作者：爱前端不爱恋爱  
链接：https://zhuanlan.zhihu.com/p/256195603  
来源：知乎  
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。

**1. 防抖**

function debounce(func, ms = 1000) {

let timer;

return function (...args) {

if (timer) {

clearTimeout(timer)

}

timer = setTimeout(() => {

func.apply(this, args)

}, ms)

}

}

// 测试

const task = () => { console.log('run task') }

const debounceTask = debounce(task, 1000)

window.addEventListener('scroll', debounceTask)

**2. 节流**

function throttle(func, ms = 1000) {

let canRun = true

return function (...args) {

if (!canRun) return

canRun = false

setTimeout(() => {

func.apply(this, args)

canRun = true

}, ms)

}

}

// 测试

const task = () => { console.log('run task') }

const throttleTask = throttle(task, 1000)

window.addEventListener('scroll', throttleTask)

**3. new**

function myNew(Func, ...args) {

const instance = {};

if (Func.prototype) {

Object.setPrototypeOf(instance, Func.prototype)

}

const res = Func.apply(instance, args)

if (typeof res === "function" || (typeof res === "object" && res !== null)) {

return res

}

return instance

}

// 测试

function Person(name) {

this.name = name

}

Person.prototype.sayName = function() {

console.log(`My name is ${this.name}`)

}

const me = myNew(Person, 'Jack')

me.sayName()

console.log(me)

**4. bind**

Function.prototype.myBind = function (context = globalThis) {

const fn = this

const args = Array.from(arguments).slice(1)

const newFunc = function () {

const newArgs = args.concat(...arguments)

if (this instanceof newFunc) {

// 通过 new 调用，绑定 this 为实例对象

fn.apply(this, newArgs)

} else {

// 通过普通函数形式调用，绑定 context

fn.apply(context, newArgs)

}

}

// 支持 new 调用方式

newFunc.prototype = Object.create(fn.prototype)

return newFunc

}

// 测试

const me = { name: 'Jack' }

const other = { name: 'Jackson' }

function say() {

console.log(`My name is ${this.name || 'default'}`);

}

const meSay = say.bind(me)

meSay()

const otherSay = say.bind(other)

otherSay()

**5. call**

Function.prototype.myCall = function (context = globalThis) {

// 关键步骤，在 context 上调用方法，触发 this 绑定为 context，使用 Symbol 防止原有属性的覆盖

const key = Symbol('key')

context[key] = this

let args = [].slice.call(arguments, 1)

let res = context[key](...args)

delete context[key]

return res

};

// 测试

const me = { name: 'Jack' }

function say() {

console.log(`My name is ${this.name || 'default'}`);

}

say.myCall(me)

**6. apply**

Function.prototype.myApply = function (context = globalThis) {

// 关键步骤，在 context 上调用方法，触发 this 绑定为 context，使用 Symbol 防止原有属性的覆盖

const key = Symbol('key')

context[key] = this

let res

if (arguments[1]) {

res = context[key](...arguments[1])

} else {

res = context[key]()

}

delete context[key]

return res

}

// 测试

const me = { name: 'Jack' }

function say() {

console.log(`My name is ${this.name || 'default'}`);

}

say.myApply(me)

**7. deepCopy**

function deepCopy(obj, cache = new WeakMap()) {

if (!obj instanceof Object) return obj

// 防止循环引用

if (cache.get(obj)) return cache.get(obj)

// 支持函数

if (obj instanceof Function) {

return function () {

obj.apply(this, arguments)

}

}

// 支持日期

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

// 支持正则对象

if (obj instanceof RegExp) return new RegExp(obj.source, obj.flags)

// 还可以增加其他对象，比如：Map, Set等，根据情况判断增加即可，面试点到为止就可以了

// 数组是 key 为数字素银的特殊对象

const res = Array.isArray(obj) ? [] : {}

// 缓存 copy 的对象，用于处理循环引用的情况

cache.set(obj, res)

Object.keys(obj).forEach((key) => {

if (obj[key] instanceof Object) {

res[key] = deepCopy(obj[key], cache)

} else {

res[key] = obj[key]

}

});

return res

}

// 测试

const source = {

name: 'Jack',

meta: {

age: 12,

birth: new Date('1997-10-10'),

ary: [1, 2, { a: 1 }],

say() {

console.log('Hello');

}

}

}

source.source = source

const newObj = deepCopy(source)

console.log(newObj.meta.ary[2] === source.meta.ary[2]);

**8. 事件总线 | 发布订阅模式**

class EventEmitter {

constructor() {

this.cache = {}

}

on(name, fn) {

if (this.cache[name]) {

this.cache[name].push(fn)

} else {

this.cache[name] = [fn]

}

}

off(name, fn) {

const tasks = this.cache[name]

if (tasks) {

const index = tasks.findIndex((f) => f === fn || f.callback === fn)

if (index >= 0) {

tasks.splice(index, 1)

}

}

}

emit(name) {

if (this.cache[name]) {

// 创建副本，如果回调函数内继续注册相同事件，会造成死循环

const tasks = this.cache[name].slice()

for (let fn of tasks) {

fn();

}

}

}

emit(name, once = false) {

if (this.cache[name]) {

// 创建副本，如果回调函数内继续注册相同事件，会造成死循环

const tasks = this.cache[name].slice()

for (let fn of tasks) {

fn();

}

if (once) {

delete this.cache[name]

}

}

}

}

// 测试

const eventBus = new EventEmitter()

const task1 = () => { console.log('task1'); }

const task2 = () => { console.log('task2'); }

eventBus.on('task', task1)

eventBus.on('task', task2)

setTimeout(() => {

eventBus.emit('task')

}, 1000)

**9. 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))

**10. es5 实现继承**

function create(proto) {

function F() {}

F.prototype = proto;

return new F();

}

// Parent

function Parent(name) {

this.name = name

}

Parent.prototype.sayName = function () {

console.log(this.name)

};

// Child

function Child(age, name) {

Parent.call(this, name)

this.age = age

}

Child.prototype = create(Parent.prototype)

Child.prototype.constructor = Child

Child.prototype.sayAge = function () {

console.log(this.age)

}

// 测试

const child = new Child(18, 'Jack')

child.sayName()

child.sayAge()

**11. instanceof**

function isInstanceOf(instance, klass) {

let proto = instance.\_\_proto\_\_

let prototype = klass.prototype

while (true) {

if (proto === null) return false

if (proto === prototype) return true

proto = proto.\_\_proto\_\_

}

}

// 测试

class Parent {}

class Child extends Parent {}

const child = new Child()

console.log(isInstanceOf(child, Parent), isInstanceOf(child, Child), isInstanceOf(child, Array))

**12. 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)

})

**13. 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');

})()

**14. vue reactive**

// Dep module

class Dep {

static stack = []

static target = null

deps = null

constructor() {

this.deps = new Set()

}

depend() {

if (Dep.target) {

this.deps.add(Dep.target)

}

}

notify() {

this.deps.forEach(w => w.update())

}

static pushTarget(t) {

if (this.target) {

this.stack.push(this.target)

}

this.target = t

}

static popTarget() {

this.target = this.stack.pop()

}

}

// reactive

function reactive(o) {

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

Object.keys(o).forEach(k => {

defineReactive(o, k, o[k])

})

}

return o

}

function defineReactive(obj, k, val) {

let dep = new Dep()

Object.defineProperty(obj, k, {

get() {

dep.depend()

return val

},

set(newVal) {

val = newVal

dep.notify()

}

})

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

reactive(val)

}

}

// watcher

class Watcher {

constructor(effect) {

this.effect = effect

this.update()

}

update() {

Dep.pushTarget(this)

this.value = this.effect()

Dep.popTarget()

return this.value

}

}

// 测试代码

const data = reactive({

msg: 'aaa'

})

new Watcher(() => {

console.log('===> effect', data.msg);

})

setTimeout(() => {

data.msg = 'hello'

}, 1000)

**15. promise**

// 建议阅读 [Promises/A+ 标准](https://promisesaplus.com/)

class MyPromise {

constructor(func) {

this.status = 'pending'

this.value = null

this.resolvedTasks = []

this.rejectedTasks = []

this.\_resolve = this.\_resolve.bind(this)

this.\_reject = this.\_reject.bind(this)

try {

func(this.\_resolve, this.\_reject)

} catch (error) {

this.\_reject(error)

}

}

\_resolve(value) {

setTimeout(() => {

this.status = 'fulfilled'

this.value = value

this.resolvedTasks.forEach(t => t(value))

})

}

\_reject(reason) {

setTimeout(() => {

this.status = 'reject'

this.value = reason

this.rejectedTasks.forEach(t => t(reason))

})

}

then(onFulfilled, onRejected) {

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

this.resolvedTasks.push((value) => {

try {

const res = onFulfilled(value)

if (res instanceof MyPromise) {

res.then(resolve, reject)

} else {

resolve(res)

}

} catch (error) {

reject(error)

}

})

this.rejectedTasks.push((value) => {

try {

const res = onRejected(value)

if (res instanceof MyPromise) {

res.then(resolve, reject)

} else {

reject(res)

}

} catch (error) {

reject(error)

}

})

})

}

catch(onRejected) {

return this.then(null, onRejected);

}

}

// 测试

new MyPromise((resolve) => {

setTimeout(() => {

resolve(1);

}, 500);

}).then((res) => {

console.log(res);

return new MyPromise((resolve) => {

setTimeout(() => {

resolve(2);

}, 500);

});

}).then((res) => {

console.log(res);

throw new Error('a error')

}).catch((err) => {

console.log('==>', err);

})

**16. 数组扁平化**

// 方案 1

function recursionFlat(ary = []) {

const res = []

ary.forEach(item => {

if (Array.isArray(item)) {

res.push(...recursionFlat(item))

} else {

res.push(item)

}

})

return res

}

// 方案 2

function reduceFlat(ary = []) {

return ary.reduce((res, item) => res.concat(Array.isArray(item) ? reduceFlat(item) : item), [])

}

// 测试

const source = [1, 2, [3, 4, [5, 6]], '7']

console.log(recursionFlat(source))

console.log(reduceFlat(source))

**17. 对象扁平化**

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));

**18. 图片懒加载**

// <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))