---

theme: juejin

highlight: androidstudio

---

这篇文章源于这几年的\*\*学习总结与大厂面试笔记\*\*，不知不觉，十万+的字数应该是有的

这些笔记总结帮助我从一个菜鸟，一步步蜕变为高级开发、“前端砖家”，并助力我拿到一些一线大厂的offer

\*\*文章面对的群体：\*\*

1. 1到3年的初中级前端工程师

2. 备战大厂的朋友

![text1.gif](https://p6-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/a3eabc9e59ca494d869d41de6d146539~tplv-k3u1fbpfcp-watermark.image?)

## 前言

入行这几年一直很焦虑与迷茫，总会遇到各种瓶颈，不知道如何深入下去。

这篇文章，结合自己的实际经历与面试情况，整理出的一些学历路线，希望能帮助正处于摸索期的前端朋友

本文只是我个人成长路线的总结，这篇文章提供的学习路线仅供大家参考。最重要的是希望大家都能找到适合自己路线，`适合自己的才是最好的`

前端知识体系繁杂，文章中总结的知识点难免有纰漏。即便是同一个知识点，不同阶段的理解也可能完全不一样。\*\*希望大家多多指正，一起交流学习\*\*

> 唯有在路上才能消除心中的迷茫与焦虑

## JavaScript 基础

### 执行上下文和执行栈

- 什么是执行上下文？

Javascript 代码在运行的时候，它都是在执行上下文中运行的

#### 全局、函数、Eval执行上下文

1）全局执行上下文（浏览器环境下，为全局的 `window` 对象；

2）函数执行上下文，每当一个函数被调用时, 都会为该函数创建一个新的上下文

3）Eval 函数执行上下文，如`eval("1 + 2")`

- 对于每个执行上下文，都有三个重要属性：变量对象、作用域链(`Scope chain`)、`this`

- 执行上下文的特点：

1）单线程，只在主线程上运行；

2）同步执行，从上向下按顺序执行；

3）全局上下文只有一个，也就是`window`对象；

4）函数每调用一次就会产生一个新的执行上下文环境。

#### 执行栈

是一种先进后出的数据结构，用来存储代码运行的所有执行上下文

1）当 JavaScript 引擎第一次遇到js脚本时，会创建一个全局的执行上下文并且压入当前执行栈。

2）每当引擎遇到一个函数调用，它会为该函数创建一个新的执行上下文并压入栈的顶部。

3）当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

4）一旦所有代码执行完毕，JavaScript 引擎从当前栈中移除全局执行上下文。

- 示例 ：

```

var a = 1; // 1. 全局上下文环境

function bar (x) {

console.log('bar')

var b = 2;

fn(x + b); // 3. fn上下文环境

}

function fn (c) {

console.log(c);

}

bar(3); // 2. bar上下文环境

```

- 图解：

![执行上下文.png](https://p6-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/5bffb09739624bcdb3572c6ae963d8a2~tplv-k3u1fbpfcp-watermark.image?)

[理解 JavaScript 中的执行上下文和执行栈](https://juejin.cn/post/6844903682283143181)

[JavaScript进阶-执行上下文](https://juejin.cn/post/6844903983438381069)

### 作用域

- 作用域

可访问变量的集合，作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。

#### 作用域类型

全局作用域、函数作用域、ES6中新增了块级作用域

函数作用域：是指声明在函数内部的变量，函数的作用域在函数定义的时候就决定了

- 块作用域

1）块作用域由` { } `包括，`if`语句和`for`语句里面的`{ }`也属于块作用域

2）块级作用域中，可通过let和const声明变量，该变量在指定块的作用域外无法被访问

#### `var、let、const`的区别

1）`var`定义的变量，没有块的概念，可以跨块访问, 可以变量提升

2）`let`定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问，无变量提升，不可以重复声明

3）`const`用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，而且不能修改，无变量提升，不可以重复声明

4）`let` 声明的变量只在块级作用域内有效

```

function func() {

if (true) {

let i = 3;

}

console.log(i); // 报错 "i is not defined"

}

func();

```

#### `var与let`的经典案例

1） 用`var`定义`i`变量，循环后打印`i`的值

```

// 案例1

// i是var声明的，在全局范围内都有效，全局只有一个变量i，输出的是最后一轮的i的值，也就是 10

let a = [];

for (var i = 0; i < 10; i++) {

a[i] = function() {

console.log(i);

};

}

a[0](); // 10

```

2） 用`let`定义`i`变量，循环后打印`i`的值

```

// 案例2

// 用let声明i，for循环体内部是一个单独的子作用域，相互独立，不会相互覆盖

let a = [];

for (let i = 0; i < 10; i++) {

a[i] = function() {

console.log(i);

};

}

a[0](); // 0

```

#### let 实现原理

\*\*借助闭包和函数作用域来实现块级作用域的效果\*\*

```

// 用var实现案例2 的效果

var a = [];

var \_loop = function \_loop(i) {

a[i] = function() {

console.log(i);

};

};

for (var i = 0; i < 10; i++) {

\_loop(i);

}

a[0](); // 0

```

#### 作用域链

当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做\*\*作用域链\*\*

[JavaScript深入之作用域链](https://github.com/mqyqingfeng/Blog/issues/6)

[js块级作用域和let，const，var区别](https://www.cnblogs.com/moumoon/p/10985250.html)

### this

- `this`的5种绑定方式

1）默认绑定(非严格模式下`this`指向全局对象，严格模式下`this`会绑定到`undefined`)

2）隐式绑定(当函数引用有上下文对象时, 如 `obj.foo()`的调用方式, `foo`内的`this`指向obj)

3）显示绑定(通过`call`或者`apply`方法直接指定`this`的绑定对象, 如`foo.call(obj)`)

4）`new`绑定，`this`指向新生成的对象

5）箭头函数，`this`指向的是定义该函数时，外层环境中的`this`，\*\*箭头函数的`this`在定义时就决定了，不能改变\*\*

- 题目1：

```

"use strict";

var a = 10; // var定义的a变量挂载到window对象上

function foo () {

console.log('this1', this) // undefined

console.log(window.a) // 10

console.log(this.a) // 报错，Uncaught TypeError: Cannot read properties of undefined (reading 'a')

}

console.log('this2', this) // window

foo();

```

开启了严格模式，只是说使得函数内的`this`指向`undefined`，它并不会改变全局中`this`的指向。因此`this1`中打印的是`undefined`，而`this2`还是`window`对象。

- 题目2

```

let a = 10

const b = 20

function foo () {

console.log(this.a) // undefined

console.log(this.b) // undefined

}

foo();

console.log(window.a) // undefined

```

如果把`var`改成了`let 或者 const`，变量是不会被绑定到`window`上的，所以此时会打印出三个`undefined`

- 题目3

```

var a = 1

function foo () {

var a = 2

console.log(this) // window

console.log(this.a) // 1

}

foo()

```

`foo()`函数内的`this`指向的是`window`，因为是`window`调用的`foo`，打印出的`this.a`是`window`下的`a`

由于函数作用域的原因我们知道`window`下的`a`还是`1`。

- 题目4

```

var obj2 = {

a: 2,

foo1: function () {

console.log(this.a) // 2

},

foo2: function () {

setTimeout(function () {

console.log(this) // window

console.log(this.a) // 3

}, 0)

}

}

var a = 3

obj2.foo1()

obj2.foo2()

```

对于`setTimeout`中的函数，这里存在隐式绑定的隐式丢失，也就是当我们将函数作为参数传递时会被隐式赋值，回调函数丢失`this`绑定，因此这时候`setTimeout`中的函数内的`this`是指向`window`

- 题目5

```

var obj = {

name: 'obj',

foo1: () => {

console.log(this.name) // window

},

foo2: function () {

console.log(this.name) // obj

return () => {

console.log(this.name) // window

}

}

}

var name = 'window'

obj.foo1()

obj.foo2()()

```

这道题非常经典，它明确了\*\*箭头函数内的`this`是由外层作用域决定的\*\*

解析：

1）对于`obj.foo1()`函数的调用，它的外层作用域是`window`，对象`obj`当然不属于作用域了(作用域只有全局作用域、函数作用域、块级作用域)，所以会打印出`window`

2）`obj.foo2()()`，首先会执行`obj.foo2()`，这不是个箭头函数，所以它里面的`this`是调用它的`obj`对象，因此打印出`obj`，而返回的匿名函数是一个箭头函数，\*\*它的`this`由外层作用域决定\*\*，那也就是函数`foo2`，那也就是它的`this`会和`foo2`函数里的`this`一样，就也打印出了`obj`

### `call apply bind`

- 三者的区别：

1）三者都可以显示绑定函数的`this`指向

2）三者第一个参数都是`this`要指向的对象，该参数为`undefined`或`null`，则默认指向全局`window`

3）传参不同：`apply`是数组、`call`是参数列表，而`bind`可以分为多次传入，实现参数的合并

4）`call、apply`是立即执行，`bind`返回绑定`this`之后的函数。如果这个新的函数作为构造函数被调用，那么`this`不再指向传入给`bind`的第一个参数，而是指向新生成的对象

- 手写`call apply bind`

```

// 手写call

Function.prototype.Call = function(context, ...args) {

// context为undefined或null，则默认指向全局window

if (!context || context === null) {

context = window;

}

// 利用symbol创建一个唯一的key值，防止新增加的属性与obj中的属性名重复

let fn = Symbol();

context[fn] = this; // this指向调用call的函数

// 隐式绑定this，如执行obj.foo(), foo内的this指向obj

let res = context[fn](...args);

delete context[fn]; // 执行完以后，删除新增加的属性

return res;

};

// apply与call函数相似，只有第二个参数是一个数组，

Function.prototype.Apply = function(context, args) {

if (!context || context === null) {

context = window;

}

let fn = Symbol();

context[fn] = this;

let res = context[fn](...args);

delete context[fn];

return res;

};

// bind要考虑返回的函数，作为构造函数被调用的情况

Function.prototype.Bind = function(context, ...args) {

if (!context || context === null) {

context = window;

}

let fn = this;

let f = Symbol();

const result = function(...args1) {

if (this instanceof fn) {

// result如果作为构造函数被调用，this指向的是new出来的对象

// this instanceof fn，判断new出来的对象是否为fn的实例

this[f] = fn;

this[f](...args1, ...args);

delete this[f];

} else {

// bind返回的函数作为普通函数被调用

context[f] = fn;

context[f](...args1, ...args);

delete context[f];

}

};

// 如果绑定的是构造函数 那么需要继承构造函数原型属性和方法

// 实现继承的方式: 使用Object.create

result.prototype = Object.create(fn.prototype);

return result;

};

```

[再来40道this面试题酸爽继续(1.2w字用手整理)](https://juejin.cn/post/6844904083707396109)

### 闭包

- 闭包： 就是函数引用了外部作用域的变量

- 闭包常见的两种情况：

一是函数作为返回值； 另一个是函数作为参数传递

- 闭包的作用：

可以让局部变量的值始终保持在内存中；对内部变量进行保护，使外部访问不到。

最常见的案例：函数节流和防抖

- 闭包的垃圾回收：

副作用：不合理的使用闭包，会造成内存泄露(就是该内存空间使用完毕之后未回收)

闭包中引用的变量直到闭包被销毁时才会被销毁

```

// 原始题目

for (var i = 0; i < 5; i++) {

setTimeout(function() {

console.log(i); // 1s后打印出5个5

}, 1000);

}

// 利用闭包，将上述题目改成1s后，打印0,1,2,3,4

// 方法一：

for (var i = 0; i < 5; i++) {

(function(j) {

setTimeout(function timer() {

console.log(j);

}, 1000);

})(i);

}

// 方法二：

// 利用setTimeout的第三个参数，第三个参数将作为setTimeout第一个参数的参数

for (var i = 0; i < 5; i++) {

setTimeout(function fn(i) {

console.log(i);

}, 1000, i); // 第三个参数i,将作为fn的参数

}

// 将上述题目改成每间隔1s后，依次打印0,1,2,3,4

for (var i = 0; i < 5; i++) {

setTimeout(function fn(i) {

console.log(i);

}, 1000 \* i, i);

}

```

[发现 JavaScript 中闭包的强大威力](https://juejin.cn/post/6844903769646317576)

[破解前端面试（80% 应聘者不及格系列）：从闭包说起](https://juejin.cn/post/6844903474212143117)

### 原型/原型链

- 原型的作用：

原型被定义为给其它对象提供共享属性的对象。函数实例可以共享原型上的属性和方法

- 原型链：

它的作用就是当你在访问一个对象上属性的时候，如果该对象内部不存在这个属性，那么就会去它`\_\_proto\_\_`属性所指向的对象（原型对象）上查找。如果原型对象依旧不存在这个属性，那么就会去其原型的`\_\_proto\_\_`属性所指向的原型对象上去查找。以此类推，直到找到`nul`，而这个查找的过程，也就构成了我们常说的\*\*原型链\*\*

\*\*原型链和作用域的区别：\*\* 原型链是查找对象上的属性，作用域链是查找当前上下文中的变量

- `\_\_proto\_\_、prototype、constructor`：

1）js中对象分为两种，`普通对象` 和 `函数对象`

2）`\_\_proto\_\_`和`constructor`是对象独有的。`prototype`属性是函数独有的，它的作用就是包含可以给特定类型的所有实例提供共享的属性和方法；但是在 JavaScript 中，函数也是对象，所以函数也拥有`\_\_proto\_\_`和 `constructor`属性

3）`constructor`属性是对象所独有的，它是一个对象指向一个函数，这个函数就是该对象的构造函数

`函数.prototype.constructor === 该函数本身`

4）一个对象的`\_\_proto\_\_`指向其构造函数的`prototype`

`函数创建的对象.\_\_proto\_\_ === 该函数.prototype`

5）特殊的`Object`、`Function`

```

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

console.log(Object.\_\_proto\_\_ === Function.prototype); // true

console.log(Function.prototype.\_\_proto\_\_ === Object.prototype); // true

console.log(Object.prototype.\_\_proto\_\_ === null); // true

```

#### instanceof

- `instanceof` 的基本用法，它可以判断一个对象的原型链上是否包含该构造函数的原型，经常用来判断对象是否为该构造函数的实例

- 特殊示例

```

console.log(Object instanceof Object); //true

console.log(Function instanceof Function); //true

console.log(Function instanceof Object); //true

console.log(function() {} instanceof Function); //true

```

- 手写`instanceof`

```

function instanceOf(obj, fn) {

let proto = obj.\_\_proto\_\_;

if (proto) {

if (proto === fn.prototype) {

return true;

} else {

return instanceOf(proto, fn);

}

} else {

return false;

}

}

// 测试

function Dog() {}

let dog = new Dog();

console.log(instanceOf(dog, Dog), instanceOf(dog, Object)); // true true

```

- `instanceof`与`typeof`的区别

1）`typeof`一般被用于来判断一个变量的类型

`typeof` 来判断`number`、`undefined`、`symbol`、`string`、`function`、`boolean`、`object` 这七种数据类型，特殊情况：`typeof null === 'object'`

2）`instanceof`判断一个对象的原型链上是否包含该构造函数的原型

[一文吃透所有JS原型相关知识点](https://juejin.cn/post/6844903984335945736)

#### new 关键字

- `new`一个对象，到底发生什么？

1）创建一个对象，该对象的原型是`fn.prototype`

2）该对象的原型指向构造函数的原型

3）调用构造函数，构造函数的`this`指向生成的对象

4）判断构造函数是否有返回值，如果有返回值则返回值是一个对象或是一个方法，则返回该值；否则返回新生成的对象

- 构造函数有返回值的案例

```

function Dog(name) {

this.name = name;

return { test: 1 };

}

let obj = new Dog("ming");

console.log(obj); // {test:1}

```

- 手写`new `

```

function selfNew(fn, ...args) {

// 创建一个对象，该对象的原型是fn.prototype

let instance = Object.create(fn.prototype);

// 调用构造函数，this指向新生成的对象，给新生成的对象上添加对应的属性

let res = fn.apply(instance, args);

// 如果fn函数有返回值，并且返回值是一个对象，则返回该对象，否则返回新生成的对象

return typeof res === "object" || typeof res === "function" ? res : instance;

}

```

#### 继承

- 多种继承方式

1）原型链继承，缺点：引用类型的属性被所有实例共享

2）借用构造函数(经典继承)

3）原型式继承

4）寄生式继承

5）组合继承

6）寄生组合式继承

- 寄生组合式继承的优势

借用父类构造函数，在不需要生成父类实例的情况下继承了父类原型上的属性和方法

- 手写寄生组合式继承

```

// 精简版

class Child {

constructor() {

// 调用父类的构造函数

Parent.call(this);

// 利用Object.create生成一个对象，该对象的原型是父类的原型，并将该对象作为子类构造函数的原型

Child.prototype = Object.create(Parent.prototype);

// 原型的constructor指向子类的构造函数

Child.prototype.constructor = Child;

}

}

// 通用版

function Parent(name) {

this.name = name;

}

Parent.prototype.getName = function() {

console.log(this.name);

};

function Child(name, age) {

Parent.call(this, name); // 调用父类的构造函数，继承父类上的属性和方法

this.age = age;

}

function createObj(o) {

// 目的是为了继承父类原型上的属性和方法，在不需要实例化父类构造函数的情况下，避免调用父类的构造函数 如new Parent()

function F() {}

F.prototype = o;

return new F(); // 创建一个空对象，该对象原型指向父类的原型对象

}

Child.prototype = createObj(Parent.prototype); // 等同于 Child.prototype = Object.create(Parent.prototype)

Child.prototype.constructor = Child;

let child = new Child("tom", 12);

child.getName(); // tom

```

[一文吃透所有JS原型相关知识点](https://juejin.cn/post/6844903984335945736)

[最详尽的 JS 原型与原型链终极详解](https://www.jianshu.com/p/dee9f8b14771)

### Class 类

1） class 类可以看做是构造函数的语法糖

```

class Point {}

console.log(typeof Point); // "function"

console.log(Point === Point.prototype.constructor); // true

```

2） class 类中定义的方法，都是定义在该构造函数的原型上

```

class Point {

constructor() {}

toString() {}

}

// 等同于

Point.prototype = { constructor() {}, toString() {} };

```

3）使用new 生成一个对象，默认会调用 constructor方法

4）使用static关键字，作为静态方法（静态方法，只能通过类调用，实例不能调用）

```

class Foo {

static classMethod() {

return "hello";

}

}

Foo.classMethod(); // 'hello'

```

5）实例属性的简写写法

```

class Foo {

bar = "hello";

baz = "world";

}

// 等同于

class Foo {

constructor() {

this.bar = "hello";

this.baz = "world";

}

}

```

6）extends 关键字，底层也是利用的寄生组合式继承

```

class Parent {

constructor(age) {

this.age = age;

}

getName() {

console.log(this.name);

}

}

class Child extends Parent {

constructor(name, age) {

super(age);

this.name = name;

}

}

let child = new Child("li", 16);

child.getName(); // li

```

[Class 的基本语法](https://es6.ruanyifeng.com/#docs/class)

### Promise

- `promise`的底层原理：`callback回调函数 + 发布订阅模式`

#### 链式调用：

1）`promise`的回调只能被捕获一次

2） 在`then`函数加上`return`，后面的`then`函数才能继续捕获到

```

// 只有第一个then函数能获取到结果，第二个then打印undefined

let pro = new Promise((resolve, reject) => resolve(1));

pro.then(res => {

console.log(res);

})

.then(res => {

console.log(res);

});

```

#### 手写`promise`

```

class Promise {

constructor(fn) {

// resolve时的回调函数列表

this.resolveTask = [];

// reject时的回调函数列表

this.rejectTask = [];

// 记录当前状态,共有pending、fulfilled、rejected 3种状态

this.state = "pending";

let resolve = value => {

// state状态只能改变一次，resolve和reject只会触发一种

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

this.state = "fulfilled";

this.data = value;

// 模拟异步，保证resolveTask事件先注册成功，要考虑在Promise里面写同步代码的情况

setTimeout(() => {

this.resolveTask.forEach(cb => cb(value));

});

};

let reject = err => {

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

this.state = "rejected";

this.error = err;

// 保证rejectTask事件先注册成功

setTimeout(() => {

this.rejectTask.forEach(cb => cb(err));

});

};

// 关键代码，执行fn函数

try {

fn(resolve, reject);

} catch (error) {

reject(error);

}

}

then(resolveCallback, rejectCallback) {

// 解决链式调用的情况，继续返回Promise

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

// 将then传入的回调函数注册到resolveTask中

this.resolveTask.push(() => {

// 重点：判断resolveCallback事件的返回值

// 假如用户注册的resolveCallback事件又返回一个Promise，将resolve和reject传进去，这样就实现了控制了链式调用的顺序

// 假如返回值为普通值，resolve传递出去

const res = resolveCallback(this.data);

if (res instanceof Promise) {

res.then(resolve, reject);

} else {

resolve(res);

}

});

this.rejectTask.push(() => {

// 同理：判断rejectCallback事件的返回值

// 假如返回值为普通值，reject传递出去

const res = rejectCallback(this.error);

if (res instanceof Promise) {

res.then(resolve, reject);

} else {

reject(res);

}

});

});

}

}

// 测试

// 打印结果1、2

new Promise((resolve, reject) => {

setTimeout(() => {

resolve(1);

}, 500);

}).then(

res => {

console.log(res);

return new Promise(resolve => {

setTimeout(() => {

resolve(2);

}, 1000);

});

}

).then(data => {

console.log(data);

});

```

#### 手写`race、all`

`race`：返回`promises`列表中第一个执行完的结果

`all`：返回`promises`列表中全部执行完的结果

```

class Promise {

// race静态方法， 返回promises列表中第一个执行完的结果

static race(promises) {

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

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

// Promise.resolve包装一下，防止promises[i]不是Promise类型

Promise.resolve(promises[i])

.then(res => {

resolve(res);

})

.catch(err => {

reject(err);

});

}

});

}

// all静态方法， 返回promises列表中全部执行完的结果

static all(promises) {

let result = [];

let index = 0;

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

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

Promise.resolve(promises[i])

.then(res => {

result[i] = res; // 输出结果的顺序和promises的顺序一致

index++;

if (index === promises.length) {

resolve(result);

}

})

.catch(err => {

reject(err);

});

}

});

}

}

```

#### 手写`retry`

`retry`的作用，当接口请求失败后，每间隔几秒，再重发几次

```

/\*

\* @param {function} fn - 方法名

\* @param {number} delay - 延迟的时间

\* @param {number} times - 重发的次数

\*/

function retry(fn, delay, times) {

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

function func() {

Promise.resolve(fn()).then(res => {

resolve(res);

})

.catch(err => {

// 接口失败后，判断剩余次数不为0时，继续重发

if (times !== 0) {

setTimeout(func, delay);

times--;

} else {

reject(err);

}

});

}

func();

});

}

```

[史上最最最详细的手写Promise教程](https://juejin.cn/post/6844903625769091079)

### async、await

- 作用：\*\*用同步方式，执行异步操作\*\*

- 知识点：

1）`async`函数是`generator`（迭代函数）的语法糖

2）`async`函数返回的是一个`Promise`对象，有无值看有无return值

3）`await`关键字只能放在`async`函数内部，`await`关键字的作用 就是获取`Promise`中返回的`resolve`或者`reject`的值

4）`async、await`要结合`try/catch`使用，防止意外的错误

#### generator

1）`generator函数`跟普通函数在写法上的区别就是，多了一个星号`\*`

2）只有在`generator函数`中才能使用`yield`，相当于`generator函数`执行的`中途暂停点`

3）`generator`函数是不会自动执行的，每一次调用它的`next`方法，会停留在下一个`yield`的位置

- `async、await`示例

```

const getData = () => new Promise(resolve => setTimeout(() => resolve("data"), 1000));

async function test() {

const data = await getData();

console.log("data: ", data);

const data2 = await getData();

console.log("data2: ", data2);

return "success";

}

test().then(res => console.log(res))

```

- 将示例转化为`generator`函数

```

function\* testG() {

// await被编译成了yield

const data = yield getData();

console.log("data: ", data);

const data2 = yield getData();

console.log("data2: ", data2);

return "success";

}

```

- 手动执行`generator`函数

```

// 执行结果与`async、await`示例一致

const getData = () =>

new Promise(resolve => setTimeout(() => resolve("data"), 1000));

function\* testG() {

// await被编译成了yield

const data = yield getData();

console.log("data: ", data);

const data2 = yield getData();

console.log("data2: ", data2);

return "success";

}

var gen = testG();

var dataPromise = gen.next();

dataPromise.value.then(value1 => {

// data1的value被拿到了 继续调用next并且传递给data

var data2Promise = gen.next(value1);

data2Promise.value.then(value2 => {

// data2的value拿到了 继续调用next并且传递value2

gen.next(value2);

});

});

```

#### 手写`async、await`

```

function generatorToAsync(generatorFn) {

// 返回的是一个新的函数

return function() {

// 先调用generator函数 生成迭代器

// 对应 var gen = testG()

const gen = generatorFn.apply(this, arguments);

// 返回一个Promise, 因为外部是用.then的方式 或者await的方式去使用这个函数的返回值的

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

// 内部定义一个step函数 用来一步步next

function step(key, arg) {

let res;

// 这个方法需要包裹在try catch中

// 如果报错了 就把promise给reject掉 外部通过.catch可以获取到错误

try {

res = gen[key](arg); // 这里有可能会执行返回reject状态的Promise

} catch (error) {

return reject(error); // 报错的话会走catch，直接reject

}

// gen.next() 得到的结果是一个 { value, done } 的结构

const { value, done } = res;

if (done) {

// 如果done为true，说明走完了，进行resolve(value)

return resolve(value);

} else {

// 如果done为false，说明没走完，还得继续走

// value有可能是：常量，Promise，Promise有可能是成功或者失败

return Promise.resolve(value).then(

val => step("next", val),

err => step("throw", err)

);

}

}

step("next"); // 第一次执行

});

};

}

// 测试generatorToAsync

// 1秒后打印data1 再过一秒打印data2 最后打印success

const getData = () =>

new Promise(resolve => setTimeout(() => resolve("data"), 1000));

var test = generatorToAsync(function\* testG() {

// await被编译成了yield

const data = yield getData();

console.log("data1: ", data);

const data2 = yield getData();

console.log("data2: ", data2);

return "success";

});

test().then(res => console.log(res));

```

[20分钟就能搞定的async/await原理](https://juejin.cn/post/7007031572238958629)

[手写async await的最简实现](https://juejin.cn/post/6844904102053281806)

[async/await 一定要加 try/catch吗？](https://segmentfault.com/a/1190000038306050)

### 深拷贝

- 深拷贝的方式：

1）`JSON.parse(JSON.stringify()) `

\*\*缺点\*\*： 无法拷贝 函数、正则、时间格式、原型上的属性和方法等

2）递归实现深拷贝

- 手写深拷贝

```

// 解决 循环引用 和 多个属性 引用同一个对象的情况

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

// hash 存储已拷贝过的对象，避免循环拷贝和重复拷贝

if (!isObject(target)) return target;

if (hash.get(target)) return hash.get(target);

// 兼容数组和对象

let newObj = Array.isArray(target) ? [] : {};

// 关键代码，解决对象的属性循环引用 和 多个属性引用同一个对象的问题，避免重复拷贝

hash.set(target, newObj);

for (let key in target) {

if (target.hasOwnProperty(key)) {

if (isObject(target[key])) {

newObj[key] = deepClone(target[key], hash); // 递归拷贝

} else {

newObj[key] = target[key];

}

}

}

return newObj;

}

function isObject(target) {

return typeof target === "object" && target !== null;

}

// 示例

let info = { item: 1 };

let obj = {

key1: info,

key2: info,

list: [1, 2]

};

// 属性循环引用

obj.key3 = obj;

let val = deepClone(obj);

console.log(val);

```

备注：

1）使用`WeakMap`的好处是，`WeakMap`存储的`key`必须是对象，并且`key`都是弱引用，便于垃圾回收

2）循环拷贝：对象的属性引用自己

```

let target = {name: 'target'};

target.target = target

```

3）重复拷贝：对象的属性引用同一个对象

```

let obj = {};

let target = {a: obj, b: obj};

```

[JSON.parse(JSON.stringify()) 实现对对象的深拷贝](https://www.cnblogs.com/sweet-ice/p/10583192.html)

[如何实现一个深拷贝](https://github.com/yygmind/blog/issues/29)

### 事件轮询机制 event loop

#### 宏任务 微任务

1）宏任务: script全部代码（注意同步代码也属于宏任务）、setTimeout、setInterval、setImmediate等

2）微任务： Process.nextTick（Node独有）、Promise、MutationObserver等

#### 事件轮询机制

事件轮询机制执行过程：

1）代码执行过程中，宏任务和微任务放在不同的任务队列中；

2）当某个宏任务执行完后,会查看微任务队列是否有任务。如果有，执行微任务队列中的所有微任务(注意这里是执行所有的微任务)

3）微任务执行完成后，会读取宏任务队列中排在最前的第一个宏任务（注意宏任务是一个个取），执行该宏任务，如果执行过程中，遇到微任务，依次加入微任务队列

4）宏任务执行完成后，再次读取微任务队列里的任务，依次类推。

经典案例

```

Promise.resolve()

.then(function() {

console.log("promise0");

})

.then(function() {

console.log("promise5");

});

setTimeout(() => {

console.log("timer1");

Promise.resolve().then(function() {

console.log("promise2");

});

Promise.resolve().then(function() {

console.log("promise4");

});

}, 0);

setTimeout(() => {

console.log("timer2");

Promise.resolve().then(function() {

console.log("promise3");

});

}, 0);

Promise.resolve().then(function() {

console.log("promise1");

});

console.log("start");

// 执行结果： start promise0 promise1 promise5 timer1 promise2 promise4 timer2 promise3

```

\*\*案例的解释：\*\*

宏任务是一个个执行，执行一个宏任务，然后就将在任务队列中的所有微任务都执行完，再执行下一个宏任务，再执行所有微任务，依次类推

#### `async、await` 事件轮询执行时机

```

// async隐式返回 Promise，会产生一个微任务，await后面的代码是在微任务时执行

console.log("script start");

async function async1() {

await async2(); // await 隐式返回promise

console.log("async1 end"); // 这里的执行时机：在执行微任务时执行

}

async function async2() {

console.log("async2 end"); // 这里是同步代码

}

async1();

setTimeout(function() {

console.log("setTimeout");

}, 0);

new Promise(resolve => {

console.log("Promise"); // 这里是同步代码

resolve();

})

.then(function() {

console.log("promise1");

})

.then(function() {

console.log("promise2");

});

console.log("script end");

// 打印结果: script start => async2 end => Promise => script end => async1 end => promise1 => promise2 => setTimeout

```

#### `event loop` 与 更新渲染时机

1) 渲染更新渲染会在`event loop`中的`Macrotasks`和`Microtasks`完成后进行，即宏任务 → 微任务 → 渲染更新（先宏任务 再微任务，然后再渲染更新）

2) 如果宏任务队列如果有大量的任务等待执行时，将`dom`的变动作为微任务，能更快的将变化呈现给用户，这样就可以在这一次的事件轮询中更新`dom`

#### `event loop`与 `vue nextTick`

\*\*`vue nextTick`为什么要优先使用微任务实现？\*\*

1) `vue nextTick`的源码实现，异步优先级判断，总结就是`Promise > MutationObserver > setImmediate > setTimeout `

2) 这里优先使用`Promise`，因为根据`event loop`与浏览器更新渲染时机，宏任务 → 微任务 → 渲染更新，使用微任务，本次`event loop`轮询就可以获取到更新的`dom`

3) 如果使用宏任务，要到下一次`event loop`中，才能获取到更新的`dom`

4) 使用微任务，目的就是能更快的获取到更新的dom

#### `node`中的`process.nextTick`

有很多文章把`Node`的 `process.nextTick` 和微任务混为一谈，但并不是同一个东西。`process.nextTick` 是 `Node.js` 自身定义实现的一种机制，有自己的 `nextTickQueue`

`process.nextTick` 执行顺序早于微任务

```

console.log("start");

setTimeout(() => {

console.log("timeout");

}, 0);

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

console.log("promise");

});

process.nextTick(() => {

console.log("nextTick");

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

console.log("promise1");

});

});

console.log("end");

// 执行结果 start end nextTick promise promise1 timeout ， process.nextTick 执行顺序早于微任务

```

[这一次，彻底弄懂 JavaScript 执行机制](https://juejin.cn/post/6844903512845860872)

[从event loop规范探究javaScript异步及浏览器更新渲染时机](https://github.com/aooy/blog/issues/5)

[Vue异步更新 - nextTick为什么要microtask优先](https://github.com/qingzhou729/study/issues/15)

[浏览器与Node的事件循环(Event Loop)有何区别?](https://zhuanlan.zhihu.com/p/54882306)

### 定时器

#### setTimeout/setInterval

`setTimeout` 固定时长后执行；`setInterval` 间隔固定时间重复执行

`setTimeout、setInterval` 最短时长为4ms

#### 定时器不准的原因

`setTimeout/setInterval` 的执行时间并不是确定的

`setTimeout/setInterval` 是宏任务，存在事件轮询，存在事件队列，其他任务会阻塞或延迟js的执行。

考虑极端情况，假如定时器里面的代码需要进行大量的计算，或者是 `DOM` 操作，代码执行时间，超过定时器的时间，会出现定时器不准的情况

#### setTimeout/setInterval 动画卡顿

不同设备的屏幕刷新频率可能不同， `setTimeout/setInterval` 只能设置固定的时间间隔，这个时间和屏幕刷新间隔可能不同

`setTimeout/setInterval` 通过设置一个间隔时间不断改变图像，达到动画效果。该方法在一些低端机 上会出现卡顿、抖动现象原因

#### requestAnimationFrame

`requestAnimationFrame` 是浏览器专门为动画提供的API，`requestAnimationFrame` 刷新频率与显示器的刷新频率保持一致。使用该api不会造成页面卡顿

`requestAnimationFrame`：告诉浏览器在下次重绘之前执行传入的回调函数(通常是操纵`dom`，更新动画的函数)

#### setTimeout、setInterval、requestAnimationFrame 三者的区别

1）引擎层面：

`setTimeout` 属于 `JS引擎` ，存在事件轮询。

`requestAnimationFrame` 属于 `GUI引擎` 。

`js引擎与GUI引擎`是互斥的,也就是说 `GUI引擎`在渲染时会阻塞`js引擎`计算。

这样设计的原因,如果在 `GUI` 渲染的时候,js 改变了`dom`,那么就会造成渲染不同步。

2）性能层面：

当页面被隐藏或最小化时，定时器 `setTimeout` 仍在后台执行动画任务。

当页面处于未激活的状态下，该页面的屏幕刷新任务会被系统暂停，`requestAnimationFrame` 也会停止。

#### setTimeout 模拟实现 setInterval

```

function mySetInterval(fn, t) {

let timer = null;

function interval() {

fn();

timer = setTimeout(interval, t);

}

interval();

return {

// cancel用来清除定时器

cancel() {

clearTimeout(timer);

}

};

}

```

#### setInterval 模拟实现 setTimeout

```

function mySetTimeout(fn, time) {

let timer = setInterval(() => {

clearInterval(timer);

fn();

}, time);

}

// 使用

mySetTimeout(() => {

console.log(1);

}, 2000);

```

[setTimeout/setInterval与requestAnimationFrame的区别？](https://blog.csdn.net/weixin\_40851188/article/details/89669416)

### 设计模式

设计模式是从许多优秀的软件系统中总结出的成功的、能够实现可维护性、复用的设计方案，使用这些方案将可以让我们避免做一些重复性的工作

#### 单例模式

一个类只能构造出唯一实例

应用案例：弹框

```

class Single {

constructor(name) {

this.name = name;

}

static getInstance(name) {

// 静态方法

if (!this.instance) {

// 关键代码 this指向的是Single这个构造函数

this.instance = new Single(name);

}

return this.instance;

}

}

let single1 = Single.getInstance("name1");

let single2 = Single.getInstance("name2");

console.log(single1 === single2); // true

```

#### 策略模式

根据不同参数命中不同的策略

应用案例：表单验证

```

// 策略对象

const strategies = {

isNoEmpty: function(value, errorMsg) {

if (value.trim() === "") {

return errorMsg;

}

},

minLength: function(value, length, errorMsg) {

if (value.trim().length < length) {

return errorMsg;

}

},

maxLength: function(value, length, errorMsg) {

if (value.length > length) {

return errorMsg;

}

},

isMobile: function(value, errorMsg) {

if (

!/^(13[0-9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|17[7]|18[0|1|2|3|5|6|7|8|9])\d{8}$/.test(

value

)

) {

return errorMsg;

}

}

};

// 验证类

class Validator {

constructor() {

this.cache = []; // 存储要验证的方法

this.errList = []; // 存储最终的验证结果

}

add(value, rules) {

for (let i = 0, rule; (rule = rules[i++]); ) {

let strategyAry = rule.strategy.split(":");

let errorMsg = rule.errorMsg;

this.cache.push(() => {

let strategy = strategyAry.shift();

strategyAry.unshift(value);

strategyAry.push(errorMsg);

// 执行策略对象中的不同验证规则

let error = strategies[strategy](...strategyAry);

if (error) {

this.errList.push(error);

}

});

}

}

start() {

for (let i = 0, validatorFunc; (validatorFunc = this.cache[i++]); ) {

validatorFunc();

}

return this.errList;

}

}

let validataFunc = function(info) {

let validator = new Validator();

validator.add(info.userName, [

{

strategy: "isNoEmpty",

errorMsg: "用户名不可为空"

},

{

strategy: "minLength:2",

errorMsg: "用户名长度不能小于2位"

}

]);

validator.add(info.password, [

{

strategy: "minLength:6",

errorMsg: "密码长度不能小于6位"

}

]);

validator.add(info.phoneNumber, [

{

strategy: "isMobile",

errorMsg: "请输入正确的手机号码格式"

}

]);

return validator.start();

};

// 需要验证表单的对象

let userInfo = {

userName: "王",

password: "1234",

phoneNumber: "666"

};

let errorMsg = validataFunc(userInfo);

console.log(errorMsg); // ['用户名长度不能小于2位', '密码长度不能小于6位', '请输入正确的手机号码格式']

```

#### 代理模式

代理对象和本体对象具有一致的接口

应用案例：图片预加载

```

// 代理模式

let relImage = (function() {

let imgNode = document.createElement("img");

document.body.appendChild(imgNode);

return {

setSrc(src) {

imgNode.src = src;

}

};

})();

let proxyImage = (function() {

let img = new Image();

// 实际要加载的图片 加载成功后 替换调占位图

img.onload = function() {

relImage.setSrc(img.src);

};

return {

setSrc(src) {

img.src = src;

// 设置占位图

relImage.setSrc(

"https://fuss10.elemecdn.com/e/5d/4a731a90594a4af544c0c25941171jpeg.jpeg"

);

}

};

})();

// 设置实际要加载的图片

proxyImage.setSrc(

"https://cube.elemecdn.com/6/94/4d3ea53c084bad6931a56d5158a48jpeg.jpeg"

);

```

#### 装饰者模式

在不改变对象自身的基础上，动态地给某个对象添加一些额外的职责

应用案例： 在函数执行前后添加新的方法

```

function fuc() {

console.log(2);

}

Function.prototype.before = function(beFn) {

let self = this;

return function() {

beFn.apply(this, arguments); // 先执行插入到前面的方法，类似于二叉树的前序遍历

return self.apply(this, arguments); // 后执行当前的方法

};

};

Function.prototype.after = function(afFn) {

let self = this;

return function() {

self.apply(this, arguments); // 先执行当前的方法

return afFn.apply(this, arguments); // 后执行插入到后面的方法

};

};

function fuc1() {

console.log(1);

}

function fuc3() {

console.log(3);

}

function fuc4() {

console.log(4);

}

fuc = fuc.before(fuc1).before(fuc4).after(fuc3);

fuc();

// 最终打印结果：4 1 2 3

```

#### 组合模式

组合模式在对象间形成树形结构;

组合模式中基本对象和组合对象被一致对待;

无须关心对象有多少层, 调用时只需在根部进行调用

应用案例： 打印文件目录

```

class Combine {

constructor() {

this.list = [];

}

add(fn) {

this.list.push(fn);

return this; // 链式调用

}

excute() {

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

this.list[i].excute();

}

}

}

let comb1 = new Combine();

comb1

.add({

excute() {

console.log(1);

}

})

.add({

excute() {

console.log(2);

}

});

let comb2 = new Combine();

comb2

.add({

excute() {

console.log(3);

}

})

.add({

excute() {

console.log(4);

}

});

let comb3 = new Combine();

comb3

.add({

excute() {

console.log(5);

}

})

.add({

excute() {

console.log(6);

}

});

comb2.add(comb3);

let comb4 = new Combine();

comb4.add(comb1).add(comb2);

comb4.excute();

// 最终打印结果：1 2 3 4 5 6

```

#### 工厂模式

工厂模式是用来创建对象的一种最常用的设计模式

不暴露创建对象的具体逻辑，而是将将逻辑封装在一个函数中，那么这个函数就可以被视为一个工厂

应用案例： `jquery window.$`

```

class Car {

constructor(name, color) {

this.name = name;

this.color = color;

}

}

class Factory {

static create(type) {

switch (type) {

case "car":

return new Car("汽车", "白色");

break;

case "bicycle":

return new Car("自行车", "黑色");

break;

default:

console.log("没有该类型");

}

}

}

let p1 = Factory.create("car");

let p2 = Factory.create("bicycle");

console.log(p1, p1 instanceof Car); // {name: '汽车', color: '白色'} true

console.log(p2, p2 instanceof Car); // {name: '自行车', color: '黑色'} true

```

#### 访问者模式

在不改变该对象的前提下访问其结构中元素的新方法

应用案例：`babel`插件

```

// 元素类

class Student {

constructor(name, chinese, math, english) {

this.name = name;

this.chinese = chinese;

this.math = math;

this.english = english;

}

accept(visitor) {

visitor.visit(this);

}

}

// 访问者类

class ChineseTeacher {

visit(student) {

console.log(`语文${student.chinese}`);

}

}

class MathTeacher {

visit(student) {

console.log(`数学${student.math}`);

}

}

class EnglishTeacher {

visit(student) {

console.log(`英语${student.english}`);

}

}

// 实例化元素类

const student = new Student("张三", 90, 80, 60);

// 实例化访问者类

const chineseTeacher = new ChineseTeacher();

const mathTeacher = new MathTeacher();

const englishTeacher = new EnglishTeacher();

// 接受访问

student.accept(chineseTeacher); // 语文90

student.accept(mathTeacher); // 数学80

student.accept(englishTeacher); // 英语60

```

#### 发布订阅模式

订阅者订阅相关主题，发布者通过发布主题事件的方式，通知订阅该主题的对象

应用案例：EventBus

```

// 发布订阅模式

class EventBus {

constructor() {

this.task = {};

}

on(type, fn) {

// on 注册事件

if (!this.task[type]) this.task[type] = [];

this.task[type].push(fn);

}

emit(type, ...args) {

// emit 发送事件

if (this.task[type]) {

this.task[type].forEach(fn => {

fn.apply(this, args); // 注意this指向

});

}

}

off(type, fn) {

// 删除事件

if (this.task[type]) {

this.task[type] = this.task[type].filter(item => item !== fn);

}

}

once(type, fn) {

// 只执行一次

function f(...args) {

fn(...args);

this.off(type, f);

}

this.on(type, f);

}

}

let event = new EventBus();

event.on("change", (...args) => {

console.log(args);

});

// 只执行一次

event.once("change", (...args) => {

console.log(args);

});

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

event.emit("change", 2, 3);

```

#### 观察者模式

一个对象有一系列依赖于它的观察者（watcher），当对象发生变化时，会通知观察者进行更新

应用案例： vue 双向绑定

```

let data = {

name: "ming",

age: 18

};

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

let value = data[key];

Object.defineProperty(data, key, {

get() {

console.log("get", value);

return value;

},

set(newValue) {

console.log("更新");

value = newValue;

}

});

});

data.name = "佩奇";

console.log(data.name);

// 依次打印： 更新 → get 佩奇 → 佩奇

```

#### 观察者与发布订阅模式的区别

观察者模式：一个对象有一系列依赖于它的观察者（`watcher`），当对象发生变化时，会通知观察者进行更新。

发布订阅模式： 订阅者订阅相关主题，发布者通过发布主题事件的方式通知订阅该主题的对象。发布订阅模式中可以基于不同的主题去执行不同的自定义事件

[javaScript设计模式统计](https://zhuanlan.zhihu.com/p/472719016)

[JavaScript 中常见设计模式整理](https://juejin.cn/post/6844903607452581896)

### Web Worker

#### 让前端拥有后端的计算能力

在HTML5的新规范中，实现了 `Web Worker` 来引入 js 的 `多线程` 技术, 可以让我们可以在页面主运行的 js 线程中加载运行另外单独的一个或者多个 js 线程

\*\*作用：\*\*

`Web Worker`专门处理复杂计算的，从此让前端拥有后端的计算能力

#### 页面大量计算，造成假死

浏览器有`GUI渲染线程`与`JS引擎`线程，这两个线程是互斥的关系

当js有大量计算时，会造成 `UI 阻塞`，出现界面渲染卡顿、掉帧等情况，严重时会出现页面卡死的情况，俗称假死

#### Web Worker使用案例

计算十万条数据，计算时长从35s变成6s，并且全程无卡顿

[在Vue中 使用 Web Worker](https://juejin.cn/post/7137728629986820126#heading-3)

#### web worker提高Canvas运行速度

`web worker`除了单纯进行计算外

还可以结合`\*\*离屏canvas\*\*`进行绘图，提升绘图的渲染性能和使用体验

[web worker 提高Canvas运行速度](https://juejin.cn/post/7137728629986820126#heading-8)

#### 计算时长超过多久适合用Web Worker

\*\*原则：\*\*

运算时间超过50ms会造成页面卡顿，属于`Long task`，这种情况就可以考虑使用`Web Worker`

但还要先考虑`通信时长`的问题，假如一个运算执行时长为100ms, 但是通信时长为300ms, 用了`Web Worker`可能会更慢通信时长

\*\*最终标准：\*\*

\*\*计算的运算时长 - 通信时长 > 50ms，推荐使用Web Worker\*\*

[如何让前端拥有后端的计算能力？一文彻底了解Web Worker](https://juejin.cn/post/7137728629986820126)

## 手写JS面试题

除了上文\*\*JavaScript基础\*\*中的提到的一些手写题外，另外补充以下题目

这些是面试中经常会遇到，也是一个优秀前端工程师的必备技巧

### reduce函数

`reduce(callbackFn, initialValue) `

1）`callbackFn`接收4个参数，`reduce((pre,cur, index, array) => {})`

`pre` 累加器、`cur` 当前值、 `index`当前下标、`array`用于遍历的数组

2）实现`reduce`方法，内部需要判断`initialValue`是否存在

不存在，需要找到数组中第一个存在的值作为`initialValue`

```

// 如果提供了initialValue时，则作为pre的初始值，index从0开始；

// 如果没有提供initialValue，找到数组中的第一个存在的值作为pre，下一个元素的下标作为index

Array.prototype.myReduce = function(fn, initialValue) {

let pre, index;

let arr = this.slice();

if (initialValue === undefined) {

// 没有设置初始值

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

// 找到数组中第一个存在的元素，跳过稀疏数组中的空值

if (!arr.hasOwnProperty(i)) continue;

pre = arr[i]; // pre 为数组中第一个存在的元素

index = i + 1; // index 下一个元素

break; // 易错点：找到后跳出循环

}

} else {

index = 0;

pre = initialValue;

}

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

// 跳过稀疏数组中的空值

if (!arr.hasOwnProperty(i)) continue;

// 注意： fn函数，接收四个参数，pre 之前累计值、cur 当前值、 当前下标、 arr 原数组

pre = fn.call(null, pre, arr[i], i, this);

}

return pre;

};

console.log([, , , 1, 2, 3, 4].myReduce((pre, cur) => pre + cur)); // 10

```

### compose

\*\*作用：\*\*

在`函数式编程`当中有一个很重要的概念就是函数组合，实际上就是把处理数据的函数像管道一样连接起来，然后让数据穿过管道得到最终的结果

在多个框架源码中，都有用到，比如`redux`、`koa` 原理 等多次遇到这个方法

效果： 将一系列函数，通过`compose`函数组合起来，像管道一样连接起来，比如函数结合`[f, g, h ]`，通过`compose`最终达到这一的效果： `f(g(h()))`

compose函数要求可执行同步方法，也可执行异步方法，两者都可以兼容

```

function compose(list) {

// 取出第一个函数，当做reduce函数的初始值

const init = list.shift();

return function(...arg) {

// 执行compose函数，返回一个函数

return list.reduce(

(pre, cur) => {

// 返回list.reduce的结果，为一个promise实例，外部就可以通过then获取

return pre.then(result => {

// pre始终为一个promise实例，result为结果的累加值

return cur.call(null, result); // 在前一个函数的then中，执行当前的函数，并返回一个promise实例，实现累加传递的效果

});

},

// Promise.resolve可以将非promise实例，转为promise实例（一种兼容处理）

Promise.resolve(init.apply(null, arg))

);

};

}

// 同步方法案例

let sync1 = data => {

console.log("sync1");

return data;

};

let sync2 = data => {

console.log("sync2");

return data + 1;

};

let sync3 = data => {

console.log("sync3");

return data + 2;

};

let syncFn = compose([sync1, sync2, sync3]);

syncFn(0).then(res => {

console.log(res);

});

// 依次打印 sync1 → sync2 → sync3 → 3

// 异步方法案例

let async1 = data => {

return new Promise(resolve => {

setTimeout(() => {

console.log("async1");

resolve(data);

}, 1000);

});

};

let async2 = data => {

return new Promise(resolve => {

setTimeout(() => {

console.log("async2");

resolve(data + 1);

}, 1000);

});

};

let async3 = data => {

return new Promise(resolve => {

setTimeout(() => {

console.log("async3");

resolve(data + 2);

}, 1000);

});

};

let composeFn = compose([async1, async2, async3]);

composeFn(0).then(res => {

console.log(res);

});

// 依次打印 async1 → async1 → async1 → 3

```

### 数组扁平化

deep用来控制扁平的层数，默认为1

```

// deep用来控制扁平的层数，默认为1

Array.prototype.myFlat = function(deep = 1) {

let arr = this;

// deep为0 则返回，递归结束

if (deep == 0) return arr;

// 使用reduce作为累加器

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

// cur为数组，继续递归，deep-1

if (Array.isArray(cur)) {

return [...pre, ...cur.myFlat(deep - 1)];

} else {

return [...pre, cur];

}

}, []);

};

console.log([1, 2, 3, [4, [5, [6]]]].myFlat(2)); // [1, 2, 3, 4, 5, [6]]

```

### map 函数实现

map中的第二个参数作为第一个参数的this

判断稀疏数组，跳过稀疏数组中的空值

```

Array.prototype.selfMap = function(fn, content) {

// map中的第二个参数作为fn函数的this

// 将类数组转化为数组，同Array.from, this为调用的数组（arr）

let arr = Array.prototype.slice.call(this);

let mappedArr = Array(); // 创建一个空数组

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

// 判断稀疏数组，跳过稀疏数组中的空值

// 稀疏数组：数组中元素的个数小于数组的长度，比如Array(2) 长度为2的稀疏数组

if (!arr.hasOwnProperty(i)) continue;

mappedArr[i] = fn.call(content, arr[i]);

}

return mappedArr;

};

let arr = [1, 2, 3];

console.log(arr.selfMap(item => item \* 2)); // [2, 4, 6]

```

### some 函数实现

some()方法用于检测数组中的元素是否满足指定条件（函数提供）

如果有一个元素满足条件，则表达式返回true , 剩余的元素不会再执行检测

如果没有满足条件的元素，则返回false

```

Array.prototype.mySome = function(fn) {

let result = false;

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

// 判断条件是否满足，满足跳出循环

if (fn(this[i])) {

result = true;

break;

}

}

return result;

};

console.log([1, 2, 3, 4].mySome(item => item > 6)); // false

```

### 判断所有的数据类型

```

function getDataType(target) {

return Object.prototype.toString.call(target).slice(8, -1).toLowerCase();

}

// 判断所有的数据类型

console.log(getDataType(null)); // null

console.log(getDataType(undefined)); // undefined

console.log(getDataType(Symbol())); // symbol

console.log(getDataType(new Date())); // date

console.log(getDataType(new Set())); // set

```

### 实现es6模板字符串

```

let name = "小明";

let age = 20;

let str1 = "我叫${name},我的年龄 ${ age}";

function tempalteStr(str) {

return str.replace(/\$\{(.\*?)\}/g, function(str, k) {

return eval(k);

});

}

console.log(tempalteStr(str1)); // 我叫小明,我的年龄20

```

### 实现 ES6 的 class 语法

`ES6` 的 `class` 内部是基于寄生组合式继承，它是目前最理想的继承方式

`ES6` 的 `class` 允许子类继承父类的静态方法和静态属性

```

function selfClass(Child, Parent) {

// Child 为子类的构造函数， Parent为父类的构造函数

// Object.create 第二个参数，给生成的空对象定义属性和属性描述符/访问器描述符

Child.prototype = Object.create(Parent.prototype, {

// 子类继承父类原型上的属性和方法

constructor: {

enumerable: false,

configurable: false,

writable: true,

value: Child

}

});

// 继承父类的静态属性和静态方法

Object.setPrototypeOf(Child, Parent);

}

// 测试

function Child() {

this.name = 123;

}

function Parent() {}

// 设置父类的静态方法getInfo

Parent.getInfo = function() {

console.log("info");

};

Parent.prototype.getName = function() {

console.log(this.name);

};

selfClass(Child, Parent);

Child.getInfo(); // info

let tom = new Child();

tom.getName(); // 123

```

### 函数柯里化

函数柯里化：将使用多个参数的一个函数转换成一系列使用一个参数的函数

柯里化函数的原理：

用闭包把参数保存起来，当参数的长度等于原函数时，就开始执行原函数

```

function mycurry(fn) {

// fn.length 表示函数中参数的长度

// 函数的length属性，表示形参的个数，不包含剩余参数，仅包括第一个有默认值之前的参数个数（不包含有默认值的参数）

if (fn.length <= 1) return fn;

// 自定义generator迭代器

const generator = (...args) => {

// 判断已传的参数与函数定义的参数个数是否相等

if (fn.length === args.length) {

return fn(...args);

} else {

// 不相等，继续迭代

return (...args1) => {

return generator(...args, ...args1);

};

}

};

return generator;

}

function fn(a, b, c, d) {

return a + b + c + d;

}

let fn1 = mycurry(fn);

console.log(fn1(1)(2)(3)(4)); // 10

```

### 函数防抖

应用场景：搜索框输入文字后调用对应搜索接口

原理：

利用闭包，不管触发频率多高，在停止触发n秒后才会执行，如果重复触发，会清空之前的定时器，重新计时，直到最后一次n秒后执行

```

/\*

\* @param {function} fn - 需要防抖的函数

\* @param {number} time - 多长时间执行一次

\* @param {boolean} flag - 第一次是否执行

\*/

function debounce(fn, time, flag) {

let timer;

return function(...args) {

// 在time时间段内重复执行，会清空之前的定时器，然后重新计时

timer && clearTimeout(timer);

if (flag && !timer) {

// flag为true 第一次默认执行

fn.apply(this, args);

}

timer = setTimeout(() => {

fn.apply(this, args);

}, time);

};

}

function fn(a) {

console.log("执行:", a);

}

let debounceFn = debounce(fn, 3000, true);

debounceFn(1);

debounceFn(2);

debounceFn(3);

// 先打印：执行: 1

// 3s后打印: 执行: 3

```

### 函数节流

应用场景：下拉滚动加载

原理：

利用闭包，不管触发频率多高，每隔一段时间内执行一次

```

/\*

\* @param {function} fn - 需要防抖的函数

\* @param {number} time - 多长时间执行一次

\* @param {boolean} flag - 第一次是否执行

\*/

function throttle(fn, time, flag) {

let timer;

return function(...args) {

// flag控制第一次是否立即执行

if (flag) {

fn.apply(this, args);

// 第一次执行完后，flag变为false；否则以后每次都会执行

flag = false;

}

if (!timer) {

timer = setTimeout(() => {

fn.apply(this, args);

// 每次执行完重置timer

timer = null;

}, time);

}

};

}

// 测试

function fn() {

console.log("fn");

}

let throttleFn = throttle(fn, 3000, true);

setInterval(throttleFn, 500);

// 测试结果，一开始就打印"fn", 以后每隔3s打印一次"fn"

```

### render函数

虚拟dom转化为真实dom

```

// 虚拟dom转化为真实dom

function render(node) {

if (typeof node === "string") {

// 创建文本节点

return document.createTextNode(node);

}

// 创建对应的dom节点

let dom = document.createElement(node.tag);

if (node.attrs) {

// 设置dom属性

Object.keys(node.attrs).forEach(key => {

dom.setAttribute(key, node.attrs[key]);

});

}

// 递归生成子节点

if (node.children) {

node.children.forEach(item => {

dom.appendChild(render(item));

});

}

return dom;

}

```

### dom To json

将真实dom转化为虚拟dom

```

// 将真实dom转化为虚拟dom

function dom2Json(dom) {

if (!dom.tagName) return;

let obj = {};

obj.tag = dom.tagName;

obj.children = [];

dom.childNodes.forEach(item => {

// 去掉空的节点

dom2Json(item) && obj.children.push(dom2Json(item));

});

return obj;

}

```

### 图片懒加载

图片的懒加载：当该图片元素出现在屏幕中时再加载对应的图片

原理：

使用`IntersectionObserver`监听元素来判断是否出现在视口，判断该图片元素出现在视口时，给`img.src`赋值对应的图片链接。

```

// html内容

// <img src="./loading.jpg" data-src="https://cube.elemecdn.com/6/94/4d3ea53c084bad6931a56d5158a48jpeg.jpeg">

// <img src="./loading.jpg" data-src="https://fuss10.elemecdn.com/e/5d/4a731a90594a4af544c0c25941171jpeg.jpeg">

function observerImg() {

let imgList = document.getElementsByTagName("img"); // 获取所有的图片元素

let observer = new IntersectionObserver(list => {

// 回调的数据是一个数组

list.forEach(item => {

// 判断元素是否出现在视口

if (item.intersectionRatio > 0) {

// 设置img的src属性

item.target.src = item.target.getAttribute("data-src");

// 设置src属性后，停止监听

observer.unobserve(item.target);

}

});

});

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

// 监听每个img元素

observer.observe(imgList[i]);

}

}

```

#### IntersectionObserver

\*\*`IntersectionObserver`替代监听`scroll`事件来判断元素是否在视口中，性能更高\*\*

`IntersectionObserver`的用法

1）let observer = new IntersectionObserver( targetList=> console.log(targetList) // 通过回调函数获取元素的状态

2）observer.observe(dom) // 监听元素；

observer.unobserve(dom) // 取消对应的监听

3）intersectionRatio大于0，来判断元素是否出现在视口，intersectionRatio初始值为0

4）元素出现和离开视口，这两种情况下都可以触发回调

[IntersectionObserver API 使用教程](http://www.ruanyifeng.com/blog/2016/11/intersectionobserver\_api.html)

### 最大并发数

控制请求最大并发数

```

/\*

\* 控制并发数

\* @param {array} list - 请求列表

\* @param {number} num - 最大并发数

\*/

function control(list, num) {

function fn() {

if (!list.length) return;

// 从任务数 和 num 中 取最小值，兼容并发数num > list.length的情况

let max = Math.min(list.length, num);

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

let f = list.shift();

num--;

// 请求完成后，num++

f.finally(() => {

num++;

fn();

});

}

}

fn();

}

```

### LazyMan

考察：事件轮询机制、链式调用、队列

```

class LazyMan {

constructor(name) {

this.name = name;

this.task = []; // 任务列表

function fn() {

console.log("hi" + this.name);

this.next();

}

this.task.push(fn);

// 重点：使用setTimeout 确保所有的任务都注册到task列表中

setTimeout(() => {

this.next();

});

}

next() {

// 取出第一个任务并执行

let fn = this.task.shift();

fn && fn.call(this);

}

sleepFirst(time) {

function fn() {

console.log("sleepFirst" + time);

setTimeout(() => {

this.next();

}, time);

}

// 插入到第一个

this.task.unshift(fn);

// 返回this 可以链式调用

return this;

}

sleep(time) {

function fn() {

console.log("sleep" + time);

setTimeout(() => {

this.next();

}, time);

}

this.task.push(fn);

return this;

}

eat(something) {

function fn() {

console.log("eat" + something);

this.next();

}

this.task.push(fn);

return this;

}

}

new LazyMan("王")

.sleepFirst(3000)

.eat("breakfast")

.sleep(3000)

.eat("dinner");

```

### sleep

等待指定时间后再执行对应方法

```

// 方法一：

// 这种实现方式是利用一个伪死循环阻塞主线程。

// 因为JS是单线程的。所以通过这种方式可以实现真正意义上的sleep

function sleep1(fn, time) {

let start = new Date().getTime();

while (new Date().getTime() - start < time) {

continue;

}

fn();

}

// 方式二： 定时器

function sleep2(fn, time) {

setTimeout(fn, time);

}

// 方式三： promise

function sleep3(fn, time) {

new Promise(resolve => {

setTimeout(resolve, time);

}).then(() => {

fn();

});

}

// 方式四： async await

async function sleep4(fn, time) {

await new Promise(resolve => {

setTimeout(resolve, time);

});

fn();

}

function fn() { console.log("fn")}

sleep1(fn, 2000);

sleep2(fn, 2000);

sleep3(fn, 2000);

sleep4(fn, 2000);

```

## 算法

## 结语

\*\*一句话：还是多写，多练习吧\*\*

看十篇讲webpack的文章，不如跟着手写一个简易的webpack来的透彻

以上所有观点仅是我个人的一些经验之谈，仅供参考，欢迎一起学习交流