1. promise
2. generator
3. call,apply,bind
4. 閉包
5. 原型原型鏈
6. Eventloop
7. 作用域鏈
8. Flex
9. 深拷貝淺拷貝
10. 防抖節流區別實現
11. This指向
12. vue導航守衛
13. 設計模式（工廠設計模式、代理模式）
14. 執行上下文棧（https://github.com/mqyqingfeng/Blog/issues/8）
15. Vue生命週期
16. 跨域(1.哪三種情況會造成跨域？協議(http和https)、端口、域名)
17. 重排和重繪
18. 網頁性能優化
19. Hooks(useEffect, useRef,useState, useMemo, useCallBack)
20. 和瀏覽器輸入url的全過程
21. Websocket和http區別
22. 前端刷新404如何處理（後端指向前端的index.html頁面和修改ngnix指向）
23. Js的封裝、繼承、多態(https://www.cnblogs.com/goloving/p/13069740.html)
24. 對於V8引擎的了解
25. Var、let、const （變量提升、暫時性死區、閉包）
26. Typeof instanceof及isArray,及需要注意的地方（暫時性死區）
    1. **promise**
27. Promise的三種狀態

1.1 pending(進行中)

1.2 fulfilled(已成功)

1.3rejected(已失敗)

1. 鏈式調用的方法

2.1 then：獲取成功的信息（resolve的內容）

2.2 catch：獲取失敗的信息（rejected的內容）

2.3finally：無論成功失敗一定會指定的內容

1. Promise.all() 將多個promise實例包裝成一個新的Promise實例

Const p = Promise.all([p1,p2,p3])

3.1當三個狀態都完成，才返回p

3.2只要有一個失敗，p的狀態為失敗，返回第一個被reject的實例

1. Promise.race()

4.1獲取最先改變的那個實例的返回值

4.2可以和超時一起，若超過多久時間沒有得到結果，則返回失敗的狀態

1. const p = Promise.race([
2. fetch('/resource-that-may-take-a-while'),
3. new Promise(function (resolve, reject) {
4. setTimeout(() => reject(new Error('request timeout')), 5000)
5. })
6. ]);
7. p.then(response => console.log(response));
8. p.catch(error => console.log(error));
9. async和await轉異步為同步（解決回調地獄）

async函数就是将 Generator函数的星号（\*）替换成async，将yield替换成await

學習地址：<https://juejin.cn/post/6844904077537574919>

**二．Generator**

1.與async類似，一種解決異步編程的方案，將函數分佈阻塞、需要調用next才進入下一步

2.使用yield配合next()執行，直到done為true

function\* showName(){

var a = yield ‘第一次返回值’;

console.log(a);

var b = yield ‘第二次返回值’;

console.log(b);

var c = yield ‘最後一次返回值’;

console.log(c);

}

const it = showName();

console.log(it.next());

console.log(it.next(‘傳遞a’));

console.log(it.next(‘傳遞b’));

console.log(it.next());

輸出：

{value:’ 第一次返回值’,done:false}

傳遞a

{value:’ 第二次返回值’,done:false}

傳遞b

{value:’ 最後一次返回值’,done:false}

{value:undefined,done:true}

（1）遇到yield表达式，就暂停执行后面的操作，并将紧跟在yield后面的那个表达式的值，作为返回的对象的value属性值。

（2）下一次调用next方法时，再继续往下执行，直到遇到下一个yield表达式。

（3）如果没有再遇到新的yield表达式，就一直运行到函数结束，直到return语句为止，并将return语句后面的表达式的值，作为返回的对象的value属性值。

（4）如果该函数没有return语句，则返回的对象的value属性值为undefined。

**三．call,apply,bind**

1.作用：都是改變this指向

2.區別：用法不同

3.例子

Var objMM={

name:”MM”,

age:18,

getPersonInfo:function(address){

console.log(this.name+”年齡：”+this.age+”地址：”+address)

}

}

Var objzz={

name:”zz”,

age:18

}

用objzz調用getPersonInfo如何調用

objMM.getPersonInfo.bind(objzz,’我愛在哪在哪’)();

objMM.getPersonInfo.apply(objzz,[’我愛在哪在哪’]);

objMM.getPersonInfo.call(objzz,’我愛在哪在哪’);

* 4. 怎么利用call、apply来求一个数组中最大或者最小值

Math.max.bind(null,1,2,3)()

Math.max.call(null,1,2,3)

Math.max.apply(null,[1,2,3])

* 5. 如何利用call、apply来做继承

function Animal(name){

this.name = name;

this.showName = function(){

console.log(this.name)

}

}

function Cat(name){

Animal.call(this,name);

}

// Animal.call(this) 的意思就是使用this对象代替Animal对象，那么 // Cat中不就有Animal的所有属性和方法了吗，Cat对象就能够直接调用Animal的方法以及属性了 var cat = new Cat("TONY"); cat.showName(); //TONY

多繼承

function Class1(){

this.showClass1 = function(a,b){

Console.log(`class1:${a},${b}`)

}

}

function Class2(){

this.showClass2 = function(a,b){

Console.log(`class2:${a},${b}`)

}

}

function Class3(){

Class1.call(this)

Class2.call(this)

}  
let arr10 = [2,2];

let demo = new Class3();

demo.showclass1.call(this,1); // class1: 1,undefined

demo.showclass1.call(this,1,2); // class1: 1,1

demo.showclass2.apply(this,arr10); // class2: 2,2

**四、閉包**

1.定義：闭包是指那些能够访问自由变量的函数

作用：讀取函數內部的變量并讓這些變量始終保存在內存里，但是大量濫用會造成內存的洩漏

2. 闭包 = 函数 + 函数能够访问的自由变量

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
   1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
   2. 在代码中引用了自由变量

例子：

var data = [];

for (var i = 0; i < 3; i++) {

data[i] = function () {

console.log(i);

};

}

data[0]();

data[1]();

data[2](); 答案是都是 3

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

// 为了凸显差异，我们将传入后的参数改名为 x

// 当然由于作用域的不同，要继续在内部沿用 i 也是可以的。

(function(x){

window.setTimeout(function() {

console.log(x);

}, 1000);

})(i);

}

**function** **outer**() {

**var** a = '123'

**return** **function** **add**(){  
　　　　*//在這裡因為作用域的關係，add是能訪問到outer的所有變數的，但是outer是訪問不到add的變數；*  
*//所以思路一轉，把add的值作為結果return出來變通實現outer外部函式訪問到了內部函式變數*

*// add就是一個閉包函式，因為他能夠訪問到outer函式的作用域，add中沒有找到變數a，則會繼續往上層作用域找*

console.log(a);

}

}

**var** inner = outer() *// 獲得add閉包函式*

inner() *//"123"*

閉包的優缺點：

優點：

1.可以讀取函式內部的變數  
2.可以避免全域性汙染

缺點：

1.閉包會導致變數不會被垃圾回收機制所清除，會大量消耗記憶體；

2.不恰當的使用閉包可能會造成記憶體洩漏的問題；

**五．原型原型鏈**

**連接地址：https://juejin.cn/post/6844903989088092174**

1js分为**函数对象**和**普通对象**，每个对象都有\_\_proto\_\_属性，**但是只有函数对象才有prototype属性**

2 Object、Function都是js内置的**函数**, 类似的还有我们常用到的Array、RegExp、Date、Boolean、Number、String

属性\_\_proto\_\_是一个对象，它有两个属性，constructor和\_\_proto\_\_；

原型对象prototype有一个默认的constructor属性，用于记录实例是由哪个构造函数创建；

1. Person.prototype.constructor == Person

// \*\*准则1：原型对象（即Person.prototype）的constructor指向构造函数本身\*\*

1. person01.\_\_proto\_\_ == Person.prototype

// \*\*准则2：实例（即person01）的\_\_proto\_\_和原型对象指向同一个地方\*\*

// 从上方 function Foo() 开始分析这一张经典之图

function Foo()  
let f1 = new Foo();

let f2 = new Foo();  
f1.\_\_proto\_\_ = Foo.prototype; // 准则2

f2.\_\_proto\_\_ = Foo.prototype; // 准则2

Foo.prototype.\_\_proto\_\_ = Object.prototype; // 准则2 (Foo.prototype本质也是普通对象，可适用准则2)

Object.prototype.\_\_proto\_\_ = null; // 原型链到此停止

Foo.prototype.constructor = Foo; // 准则1

Foo.\_\_proto\_\_ = Function.prototype; // 准则2

Function.prototype.\_\_proto\_\_ = Object.protitype; // 准则2 (Function.prototype本质也是普通对象，可适用准则2)

Object.prototype.\_\_proto\_\_ = null; // 原型链到此停止

由此可以得出结论： 除了Object的原型对象（Object.prototype）的\_\_proto\_\_指向null，其他内置函数对象的原型对象（例如：Array.prototype）和自定义构造函数的 \_\_proto\_\_都指向Object.prototype, 因为原型对象本身是普通对象。 即：

Object.prototype.\_\_proto\_\_ = null;

Array.prototype.\_\_proto\_\_ = Object.prototype;

Foo.prototype.\_\_proto\_\_ = Object.prototype;

prototype

Person.prototype

Person構造函數

constructor

\_\_proto\_\_

\_\_proto\_\_

Object.prototype

person

\_\_proto\_\_

prototype

Object

null

constructor

**由**\_\_proto\_\_**構成的為原型鏈**

**Eventloop**

1. 事件循環,指瀏覽器或是node解決JS單線程阻塞的解決機制

2. 宏任務與微任務

3. 宏任務 (MacroTask)：

* + script全部代碼
  + setTimeout
  + setInterval
  + setImmediate (瀏覽器通常不支持(只有IE10支持))
  + I/O
  + UI Rendering

微任務 (MicroTask)：

* + Process.nextTick（Node才有）
  + Promise (注意executer是同步)
  + Object.observe(廢了)
  + MutationObserver

分析：

* + 執行棧在執行完同步任務後，查看執行①**棧**是否為空
  + 如果執行棧為空，就會去檢查②**微任務隊列**是否為空，若不為空執行所有微任務
  + 如果微任務對列為空，執行③**宏任務**
  + 每次單個宏任務執行完畢後，檢查④**微任務隊列**是否為空，如果不為空按先入先出規則，全部執行完微任務
  + 設置為任務隊列成null，然後執行**宏任務**，這樣一直循環

**作用域鏈**

1. 作用域：隔離變量，在es6出來前，全局為var，作用域將變量隔離，防止變量污染全局
2. **内层作用域可以访问外层作用域的变量，反之不行**
3. **自由變量：**向父级作用域寻找
4. **作用域鏈：由本層作用域一層一層往上刀父作用域去尋找自由變量，這種一層一層的關係就是作用域鏈**
5. **例子**

var a = 100

function F1() {

var b = 200

function F2() {

var c = 300

console.log(a) // 自由变量，顺作用域链向父作用域找

console.log(b) // 自由变量，顺作用域链向父作用域找 console.log(c) // 本作用域的变量 } F2()

}

F1()

**Flex**

網址：<https://www.zhangxinxu.com/wordpress/2018/10/display-flex-css3-css/>

**深拷貝淺拷貝**

網址：https://github.com/YvetteLau/Step-By-Step/issues/17

浅拷贝：复制的是其引用的地址，当原始值改变时，浅拷贝的值也进行相应变化  
深拷贝：复制的是值，当原始值改变时，深拷贝的值不会变化

淺拷貝

1.直接赋值;任何操作都会影响原数组;

let obj2 = obj1;

2.Object.assign;

拷贝属性值,假如属性值是一个对象的引用,那么也会指向那个引用;

let obj2 = Object.assign({},obj1);

Array.prototype.concat();

合并多个数组;

拷贝规则同Object.assign;

let arr2 = arr1.concat(...arr);

3.Array.prototype.slice();

提取数组;

参数(可选)：startIndex,endIndex(与索引值一致);

拷贝规则同Object.assign;

let arr2 = arr1.slice(start,end);

4.扩展运算符(...);

拷贝规则同Object.assign;

let obj2 = {obj1} or [...obj1];

lodash(\_.clone());

拷贝规则同Object.assign;

let obj2 = \_.clone(obj1);

深拷貝：1.JSON.stringify()和JSON.parse() 2. Lodash的cloneDeep

**Vue導航守衛**

