原生js

1. 函数调用this指向问题

包括函数名()和匿名函数调用，this指向window

对象.方法名，this指向对象

New构造函数名，this指向构造函数

间接调用：利用call和apply来实现，this就是call和apply对应的第一个参数， 如果不传值或者第一个值为null,undefined时this指向window

1. Es6中箭头函数的this和定义时有关，和调用无关，调用就是函数的调用模式
2. Call apply bind

call和apply可以调用函数，改变this，实现继承和借用别的对象的方法

Call和apply的定义：调用方法，用一个对象替换掉另一个对象（this）

对象.call(新this对象，实参1，实参2，实参3…）

对象.apply(新this对象，[实参1，实参 2，实参3…]）

Call和apply的用法：

**间接调用函数，改变作用域的this值**

**劫持其他对象的方法**

var foo = {

name:"张三",

logName:function(){ console.log(this.name); }

}

var bar={ name:"李四" };

foo.logName.call(bar);//李四

实质是call改变了foo的this指向为bar,并调用该函数

**两个函数实现继承**

function Animal(name){

this.name=name;

this.showName = function(){ console.log(this.name); }

}

function Cat(name){ Animal.call(this, name); }

var cat = new Cat("Black Cat");

cat.showName(); //Black Cat

**为类数组（arguments和nodeList)添加数组方法push pop**

(function(){

Array.prototype.push.call(arguments,'王五');

console.log(arguments); //['张三','李四','王五'] })('张三','李四')

**合并数组**

let arr1=[1,2,3];

let arr2=[4,5,6];

Array.prototype.push.apply(arr1,arr2); //将arr2合并到了arr1中

**求数组最大值**

Math.max.apply(null,arr)

**判断字符类型**

Object.prototype.toString.call({})

1. 函数的节流和防抖

**节流**：某个时间段内只执行一次

应用：滚动条，resize事件一段时间触发一次

let throttle = function(func, delay){

let timer = null;

return function(){

if(!timer){

timer = setTimeout(function(){

func.apply(this, arguments);

timer = null;

},delay);

}

};

};

function handle(){

console.log(Math.random());

}

window.addEventListener("scroll",throttle(handle, 1000)); //事件处理函数

**防抖**：处理函数截至后一段时间依次执行

应用：scroll resize事件触发完成后一段时间触发

function debounce(fn, wait){

var timeout = null;

return function(){

if(timeout!==null) clearTimeout(timeout); //如果多次触发将上次记录延迟清除掉

timeout=setTimeout(function(){

fn.apply(this, arguments);

timer=null;

},wait);

};

}

//处理函数

function handle(){

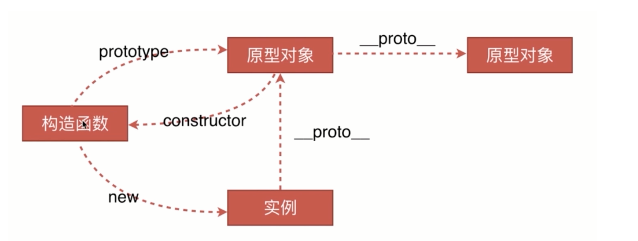
console.log(Math.random());

}

//滚动事件

window.addEventListener("onscroll",debounce(handle,1000));

1. 原型链：对象继承属性的一个链条



1. 创建实例的方法

字面量 let obj={‘name’:’张三’}

Object构造函数创建 let obj=new Object()

Obj.name=’张三’

使用工厂模式创建对象

function createPerson(name){

var o=new Object(){

o.name=name;

};

return o;

}

var person1=createPerson(‘张三’);

使用构造函数创建对象

function person(name){

this.name=name;

}

var person1=new Person(‘张三’)

1. new运算符

创建一个新对象

This指向构造函数

构造函数有返回，会替换new出来的对象，如果没有就是new出来的对象

手动封装一个new运算符

var new2 = function ( func ) {

Var o = Object.create( func.prototype ); //创建对象

Var k = func.call( o ); //改变this指向，把结果付给k

If ( typeof k === ’object’) { //判断k的类型是不是对象

return k;

} else { return o; }

1. 继承的方式：js是一门弱类型动态语言，封装和继承是它的两大特性

**原型链继承**：将父类的实例作为子类的原型

//定义一个动物类 （父类）

function Animal(name) {

//属性

this.name = name || ‘Animal’ ;

//实例方法

this.sleep = function() {

console.log(this.name + ‘is sleeping’);

}

}

//原型方法

Animal.prototype.eat = function (food) {

console.log(this.name + ‘is sleeping’);

}

//子类

function Cat(){ }

Cat.prototype = new Animal();

Cat.prototype.name='cat'; //&emsp;Test Code

var cat = new Cat();

console.log(cat.name); //cat

console.log(cat.eat('fish')); //cat正在吃：fish undefined

console.log(cat.sleep()); //cat正在睡觉！

undefined console.log(cat instanceof Animal); //true

console.log(cat instanceof Cat); //true

优缺点：简单易于实现，但是要想为子类新增属性和方法，必须要在new Animal() 这 样的语句之后执行，无法实现多继承

**构造继承**：实质是利用call来改变Cat中的this指向

function Cat (name) {

Animal.call (this);

This.name = name || ’Tom’ ;

}

优缺点：可以实现多继承，不能继承原型属性/方法

**拷贝继承：**将父类的属性和方法拷贝一份到子类中

function Cat (name) {

var animal = new Animal ();

for( var p in animal) {

Cat.prototype [p] = animal [p];

}

Cat.prototype.name = name || ‘Tom’;

}

优缺点：支持多继承，但是效率低占用内存

**组合继承**：通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实 例做为子类原型，实现函数复用

function Cat (name) {

Animal.call (this);

this.name = name || ‘Tom’ ;

}

Cat.prototype = new Animal ();

Cat.prototype.constructor = Cat;

**寄生组合继承：**

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

(function(){ // 创建一个没有实例方法的类

var Super = function(){};

Super.prototype = Animal.prototype; //将实例作为子类的原型

Cat.prototype = new Super(); })();

**ES6的extends继承：**ES6的继承机制是先创造父类的实例对象this（所以必须先调用 super方法），然后再用子类的构造函数修改this

//父类

class Person { //constructor是构造方法

constructor(skin, language) {

this.skin = skin;

this.language = language;

}

say() { console.log('我是父类') }

}

//子类

class Chinese extends Person {

constructor(skin, language, positon) {

//console.log(this);//报错

super(skin, language);

//super();相当于父类的构造函数

//console.log(this);调用super后得到了this，不报错，this指向子类，相当于调用了父类

.prototype.constructor.call(this)

this.positon = positon;

}

aboutMe() {

console.log(`${this.skin} ${this.language} ${this.positon}`); }

} //调用只能通过new的方法得到实例,再调用里面的方法

let obj = new Chinese('红色', '中文', '香港');

obj.aboutMe();

obj.say();

1. **高阶函数**：函数的参数是函数或返回函数

常见：map reduce filter sort

**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

fn(a, b, c, d) =>fn(a) (b) (c) (d)

let currying = function (fn) {

//args 获取第一个方法内的全部参数

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

return function (){

//将后面方法里的全部参数和args进行合并

var newArgs = args.concat (Array.prototype.slice.call (arguments ))

//把合并后的参数通过apply作为fn的参数并执行

return fn.apply (this, newArgs)

}

}

**反柯里化**： obj.func (arg1, arg2) =>func (obj, arg1, arg2)

Function.prototype.uncurrying = function () {

Var that = this;

Return function (){

Return function.prototype.call.apply (that, arguments);

}

};