**new 操作符具体做了什么(或者问为什么new一个构造函数就能返回一个对象)?**

new的过程实际上是创建一个新对象，把新对象的原型设置为构造函数的原型,最终返回一个空对象,但这个空对象不是真空的，而是已经含有原型的引用（\_\_proto\_\_).

(1) 创建一个空对象obj:

var this = Object.create(Func.prototype),即创建了一个Func构造函数的原型对象.可以看作:

this = { \_\_proto\_\_: Func.prototype };

(2) 让空对象的\_\_proto\_\_（IE没有该属性）成员指向了构造函数的prototype成员对象

(3) 使用apply调用构造器函数，this绑定到空对象obj上。

(4) 返回空对象obj.

**对象的增删改查**

查:

var person = {

name :"ly"

}

var sn = "name";

person.name; // “ly”

person.sn; // undefined; .点语法查询后面不能是变量，只能是具体的属性名。

Person[“name”]; // “ly” 通过[]方法查询对象属性，[]里面可以是字符串。

person[sn]; // "ly",通过[]方法查询对象属性，[]里面也可以是变量。

// 查询对象属性是先看看当前对象本身是否设置了该属性，如果当前对象未设置该属性，则再沿着对象的原型链看是否设置了该属性，若都没有，则返回undefined。

增:

var person = new Object();

person.name = "lyl"; //点语法增加那么属性

person["name"] = "lyl"; // []语法增加属性

var event = “click”;

person[“on” + event] = function fn(){}; // []语法使用变量增加属性

改:

var person = {

name:"lyl"

};

person.name = "xiaowang";

person["name"] = "xiaowang"

删:

var person = {

name:"lyl"

};

delete person.name;

console.log(person.name); // undefined；

delete person["name"];

console.log(person.name) // undefined；

**对象的常用方法有哪些**

(1): Object.create() 被创建的对象继承另一个对象的原型，在创建新对象时可以指定一些属性。

(2): Object.entries() 将对象转化为数组Object.entries()是Object.keys()和Object.values()的集合

let obj = {name:'amz',age:23}

console.log( Object.entres(obj) ) // [ ['name','amz'], ['age', 23] ]

(3): Object.freeze() 冻结对象 改对象永远不变，不可修改不可删除.

var obj = {a: 2}

Object.freeze(obj)

obj.a = 10

console.log(obj.a) // 2 不可以修改的

(4) hasOwnProperty: hasOwnProperty可以判断对象本身是否拥有这个属性,不包含原型和继承的属性;利用这一点可以过滤掉原型中的属性,在枚举的过程中判断返回true,只枚举自身拥有的属性.

var obj = {

name : "wang",

age : 20

}

for (var key in obj){

if(obj.hasOwnProperty(key) == true){

console.log(obj[key]);

}

(5) in: in 只能判断对象是否能访问到该元素，继承过来的属性也能够访问.

var obj = {

name : "wang",

age : 20

}

Object.prototype.sex = "meal";

console.log("sex" in obj); // true

(6) instanceof: 用来判断某个构造函数的prototype属性是否存在于另外一个要检测对象的原型链上:

A instanceof B; //A对象是不是B构造函数构造出来的，看A对象的原型链上有没有B的原型

var obj = {

name : "wang",

age : 20

}

obj instanceof Object // true

**对象的枚举的方法(遍历enumeration)**

(1)for...in:for in会输出对象自身以及原型链上的可枚举的属性, 可通过hasOwnProperty()方法过滤掉获取到原型链上的可枚举属性:

var obj = {

name : "wang",

age : 20

}

for (var key in obj){

if(obj.hasOwnProperty(key) == true){

console.log(obj[key]);

}

(2)Object.keys获取对象自身可枚举的属性键; Object.values获取对象自身可枚举的属性值; Object.entries获取对象自身可枚举的属性键和属性值.

let obj = {name:'amz',age:23}

for(var key of Object.keys(obj)){

console.log(obj[key]);

}

// 'amz' , 23

(3)Object.getOwnPropertyNames可以用来获取对象自身的全部属性名. 甚至包括不可枚举的属性.

//不可枚举属性

var my\_obj = Object.create({}, {

getFoo: {

value: function() { return this.foo; },

enumerable: false

}

});

my\_obj.foo = 1;

for(var i in my\_obj){console.log(i)}

// foo,for in无法枚举getfoo属性

console.log(Object.getOwnPropertyNames(my\_obj)); // ["getFoo","foo"]可以遍历不可枚举属性.

**创建对象的方法有哪些?各有什么优缺点?**

(1)工厂模式:

function createPerson(name,age) {

var o = new Object();

o.name = name;

o.age = age;

o.say = function(){

alert(this.name);

}

return o;

}

var person1 = createPerson("lisi",21);

解决了重复实例化多个对象的问题，但没有解决对象识别的问题（工厂模式无从识别对象的类型，因为全部都是Object，不像Date、Array等，本例中，得到的都是o对象，对象的类型都是Object，因此出现了构造函数模式）

(2) 构造函数模式:

function Person(name,age) {

this.name = name;

this.age = age;

this.family = family;

this.say = function(){

alert(this.name);

}

}

var person1 = new Person("lisi",21);

构造函数知道自己从哪里来（通过 instanceof 可以看出其既是Object的实例，又是Person的实例）

(3) 混合模式(混合原型模式和构造模式):

function Person(name,age){

this.name = name;

this.age = age;

}

Person.prototype = {

constructor: Person,，

say: function(){

alert(this.name);

}

} // 混合模式共享着对相同方法的引用，又保证了每个实例有自己的私有属性。最大限度的节省了内存

**什么是对象的原型链?原型链的作用?**

Function构造函数有一个prototype属性,属性值是一个普通对象.原型对象定义了公共的方法和属性,为每个实例对象存储共享的方法和属性.

除了自定义的属性,prototype属性默认有constructor属性和\_proto\_属性,.

constructor指向构造函数自身;

为了实现继承，还有一个原型链指针\_\_proto\_\_，该指针指向上一层的原型对象, 上一层的原型对象的结构依然类似，这样利用\_\_proto\_\_一直指向Object的原型对象上，原型链的最顶端就是Object/null，如此变形成了javascript的原型链继承，同时也解释了为什么所有的javascript对象都具有Object的基本方法(如toString()方法)。

原型可以实现数据和方法共享,而原型链主要为了实现继承,子类可以继承父类的属性和方法.

**ES5之前的实现继承的几种方式**

**(1)** **原型链继承:**

核心思想是将父类的实例化对象作为子类构造函数的原型.

function Parent(name){ // 父类

this.name = name;

}

Parent.prototype.showName = function(){

return this.name;

}; //父类方法

//子类:

function Child(){

}

// 继承父类Animal

Child.prototype = new Parent ();

Child.prototype.constructor = Child; // 需要修复Child原型上的构造器属性指向构造函数本身

// 最大的缺陷之一:无法实现多继承,因为只能对Child.prototype进行一次赋值,多次赋值会覆盖.

// 最大的缺陷之一:无法传参,创建子类的实例时，无法向父类构造函数传参

**(2)构造继承**

核心思想: 利用Fn.call()的根本功能,将子类构造函数的this当作参数传入call(),改变Fn父类函数的this指向:

function Parent(name){ // 父类

this.name = name;

}

Parent.prototype.showName = function(){

return this.name;

}; //父类方法

//子类:

function Child(name){

Parent.call(this,name)

}

// 缺陷:只能继承父类的实例属性和方法，不能继承原型属性/方法

**(3)组合式继承**

核心思想: 用到了构造继承和原型链继承,所以既可以继承父类原型上的属性和方法,也可以继承父类的实例化的属性和方法,

function Parent(name){ // 父类

this.name = name;

}

Parent.prototype.showName = function(){

return this.name;

}; //父类方法

//子类:

function Child(name){

Parent.call(this,name)

}

Child.prototype = new Parent();

Child.prototype.constructor = Child;

**分别用ES5和ES6实现继承**

ES5:

参考上面的组合式继承

ES6:

//父类:

class Parent{

curstructor(name){

this.name = name;

}

say(){

return this.name

}

}

//子类继承:

Class Child extends Parent { //通过关键词extends实现继承

constructor(name){

super(name); // super实例化调用父类,继承name属性

this.age = 18; // 子级自己增加属性

}

// 重写父类方法:

say(){

return this.age

}

}

**什么是浅拷贝?如何实现深拷贝?**

浅拷贝:js基本数据类型与对象的指针（即地址）存放在栈内存，对象存放在堆内存。

浅拷贝实际拷贝的是指针地址,所以克隆的对象与原始对象还存在关联,它们指向同一块堆内存,会互相影响.

直接将一个对象复制给另外一个对象属于浅拷贝, Object.assign(target, ...source)方法属于不完全深拷贝.

深拷贝:

深拷贝则拷贝的对象与源对象引用地址完全不同，数据存在于重新开辟的新空间. 克隆对象与原始对象无关联,不会互相影响.

实现方式:

(1)JSON序列化:

var obj = {...} ; // 这是一个复杂对象;

var obj2 = JSON.parse(JSON.stringify(obj));

至此,实现了对obj的深度克隆对象obj2.

缺陷: 数据有函数，日期，DOM节点等不适用.

(2)递归实现:

function deepClone(obj){

let newObj=Array.isArray(obj)?[]:{}; // 判断是数组还是对象,相应创建空数组或空对象.

if(obj&&typeof obj ==="object"){

for(let key in obj){

if(obj.hasOwnProperty(key)){

//判断循环中取出的值是否还是一个对象,是的话继续递归,否则直接赋值 newObj[key]=(obj&&typeofobj[key]==='object')?deepClone(obj[key]):obj[key];

}

}

}

return newObj

}