第5章 面向对象编程应用

课程提要

- 对象常用属性和方法
- 对象的继承
- 对象冒充和克隆

5.1 对象常用属性和方法

5.1.1 常用属性

5.1.1.1 prototype

prototype用于获取和设置对象的原型属性。

语法：

```
object.prototype.name=value
```

实例：

```
function employee(name,job,born){
    this.name=name;
    this.job=job;
    this.born=born;
}

var bill=new employee("Bill Gates","Engineer",1985);
employee.prototype.salary=null;
bill.salary=20000;
document.write(bill.salary);
```

5.1.1.2 constructor

constructor用于获取实例对象的构造函数类型，经常性用于判断变量的数据类型是对象还是数组。

语法：

```
object.constructor
```

实例：

```
var test=new Array();
if (test.constructor==Array){
    document.write("This is an Array");
}

if (test.constructor==Boolean){
    document.write("This is a Boolean");
}

if (test.constructor==Date){
    document.write("This is a Date");
}

if (test.constructor==String){
    document.write("This is a String");
}
```

5.1.2 常用方法

5.1.2.1 isPrototypeOf

isPrototypeOf用于指示对象是否存在于另一个对象的原型链中。如果存在，返回`true`，否则返回`false`。

语法：

```
prototypeObject.isPrototypeOf( object )
```

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}

var lisi = new Person("李四",20);
// 这个方法用来判断，某个proptotype对象和某个实例之间的关系。
console.log(Person.prototype.isPrototypeOf(lisi)); //true
```

5.1.2.2 hasOwnProperty

每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

语法：

```
object.hasOwnProperty(proName)
```

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}
var lisi = new Person("李四",20);
//用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。
console.log(lisi.hasOwnProperty("name")); // true
```

5.1.2.3 getOwnPropertyDescriptor

getOwnPropertyDescriptor用来获取对象属性的描述信息，其中包括属性的value(值)、writable(是否可写)、configurable(是否可设置)、enumerable(是否可枚举)。

注意：必须通过Object对象调用。

语法：

```
Object.getOwnPropertyDescriptor(object, propertyname)
```

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}

var lisi = new Person("李四",20);
console.log(Object.getOwnPropertyDescriptor(lisi,"name"));
```

5.1.2.4 defineProperty

defineProperty用来修改对象属性的描述信息。

语法：

```
Object.defineProperty(object, propertyname, descriptor)
```

参数：

- `object`：必需。要在其上添加或修改属性的对象。
- `propertyname`：必需。一个包含属性名称的字符串。
- `descriptor`：必需，属性描述符。

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}

var lisi = new Person("李四",20);
```

```
Object.defineProperty(lisi,"name",{
    configurable:false,    // 能否使用delete、能否修改属性特性、或能否修改访问器属性，false为不可重新定义，默认值为true
    enumerable:false,     // 对象属性是否可通过for-in循环，false为不可循环，默认值为true
    writable:false,     // 对象属性是否可修改,false为不可修改，默认值为true
    value:'mao'  // 对象属性的默认值，默认值为undefined
});

console.log(lisi);
console.log(Object.getOwnPropertyDescriptor(lisi,"name"));
```

5.1.2.5 propertyIsEnumerable

propertyIsEnumerable用来判断改对象的属性是否可以用for...in枚举（遍历）。

语法：

```
object. propertyIsEnumerable(proName)
```

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}

var lisi = new Person("李四",20);
console.log(lisi.propertyIsEnumerable("name"));//true
console.log(lisi.propertyIsEnumerable("constructor"));//false
```

5.1.3 运算符

5.1.3.1 in

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

实例：

```
function Person(name,age){
    this.name = name
    this.age = age;
}

var lisi = new Person("李四",20);
// in 判断是否存在属性
console.log("name" in lisi); // true
console.log("sex" in lisi); // false
```

5.1.3.2 delete

delete运算符用于删除对象对其属性的引用，当属性失去引用之后会被垃圾回收，不再存在。

实例：

```
var person2 = {
    age : 28,
    title : 'fe'
};

delete person2.age;
delete person2['title'];
console.log(person2.age) // undefined
console.log(person2.title ) // undefined
//不存在或者特定属性是无法删除的
console.log(delete Object.prototype); // false
```

5.1.3.3 instanceof

instanceof运算符就是判断一个实例是否属于某种类型。

实例：

```
function Cat(){}
var cat1 = new Cat();
console.log(cat1 instanceof Cat)//true
```

5.2 对象的继承

面向对象编程很重要的一个方面，就是对象的继承。A 对象通过继承 B 对象，就能直接拥有 B 对象的所有属性和方法。这对于代码的复用是非常有用的。

5.2.1 构造继承

基本思想

构造继承的**基本思想**就是利用call或者apply把父类中通过this指定的属性和方法复制（借用）到子类创建的实例中。

核心

使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）。

缺点

方法都在构造函数中定义， 只能继承父类的实例属性和方法，不能继承原型属性/方法，无法实现函数复用，每个子类都有父类实例函数的副本，影响性能。

实例：

```
// 定义一个人类
function Person(name){
    this.name = name;
    this.sayHello = function(){
        alert(this.name);
    }
```

```
}
// 定义一个学生类
function Student(name,age){
    // 学生类调用人类去实例化，继承了人类
    Person.apply(this,new Array(name));
    this.age =age;
    this.show= function(){
        alert(this.name+"---"+this.age);
    }
}
var stu = new Student("hanfei.li",20);
stu.sayHello();
stu.show();
```

实例：

```
// 定义一个人类
function Person(name){
    this.name = name;
    this.sayHello = function(){
        alert(this.name);
    }
}
// 定义一个学生类
function Student(name,age){
    // 学生类调用人类去实例化，继承了人类
    Person.call(this,name);
    this.age =age;
    this.show= function(){
        alert(this.name+"---"+this.age);
    }
}
var stu = new Student("hanfei.li",20);
stu.sayHello();
stu.show();
```

5.2.2 原型链继承

基本思想

每创建一个函数，该函数就会自动带有一个 prototype 属性。该属性是个指针，指向了原型对象，并且可以访问原型对象上的所有属性和方法。

核心 将父类的实例作为子类的原型

缺点 父类新增原型方法/原型属性，子类都能访问到，父类一变其它的都变了。

实例：

```
//定义一个Person构造函数，作为Student的父类
function Person(){}
//对原型进行扩展
Person.prototype.name = "tom";
```

```
Person.prototype.sayHello = function(){
    alert(this.name);
}
function Student(){}
//通过构造器函数创建出一个新对象，把父类的东西都拿过来。
Student.prototype = new Person();
Student.prototype.show = function(){
    alert("student--");
}
var stu = new Student();
stu.sayHello();
stu.name="韩非子";
stu.sayHello();
stu.show();
```

5.2.3 组合继承

基本思想

所有的实例都能拥有自己的属性，并且可以使用相同的方法，组合继承避免了原型链和借用构造函数的缺陷，结合了两个的优点，是最常用的继承方式。

核心

通过调用父类构造，继承父类的属性并保留传参的优点，然后再通过将父类实例作为子类原型，实现函数复用

缺点

调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

实例：

```
function Perso(name) {
    this.name = name;
    this.friends = ['小李','小红'];
};

Person.prototype.getName = function () {
    return this.name;
};

function Parent (age) {
    Person.call(this,'老明');    //这一步很关键
    this.age = age;
};

Parent.prototype = new Person('老明');    //这一步也很关键
var result = new Parent(24);
console.log(result.name);       //老明

result.friends.push("小智");    //
console.log(result.friends);    //['小李','小红','小智']
console.log(result.getName());    //老明
console.log(result.age);       //24
```

```
var result1 = new Parent(25);   //通过借用构造函数都有自己的属性，通过原型享用公共的方法
console.log(result1.name);   //老明
console.log(result1.friends);   //['小李','小红']
```

5.3 对象冒充和克隆

5.3.1 对象冒充

Javascript本身也是一种基于对象的语言，但是对继承提供的语法上的支持不完善，可以用对象冒充实现相关机制。

对象冒充是指一个对象冒充另外一个对象来实行其他对象的方法，即一个对象将其他对象的方法当做是自身的方法来执行。

实例：

```
function ClassA(name){
    this.name=name;
    this.getName=function(){
        return this.name;
    }
}

function ClassB(name,password){
    this.ClassA=ClassA;
    /*
        this.ClassA(name)
        等价于
        this.name=name;
        this.getName=function(){
            return this.name;
        },
        即将ClassA函数中的代码复制过来
   */
    this.ClassA(name);
    delete this.ClassA;
    this.password=password;
    this.getPassword=function(){
        return this.password;
    }
}

var b =new ClassB('wwww','1123');
console.log(b.getName());
```

实例：

```
function ClassA(name){
    this.name=name;
    this.getName=function(){
        return this.name;
```

```
    }
}

function ClassB(name,password){
    //此处的ClassA.call(this,name); 即将ClassA的this指向了ClassB的this.从而实现了对象冒充.
    ClassA.call(this,name);
    this.password=password;
    this.getPassword=function(){
        return this.password;
    }
}

var b = new ClassB('www','111');
console.log(b.getName());
```

5.3.2 对象克隆

对象克隆即把父对象的所有属性和方法，拷贝进子对象。对象克隆分为浅克隆和深克隆。

5.3.2.1 浅克隆

浅克隆只是拷贝了父对象的基本类型的数据。如果父对象的属性等于数组或另一个对象，那么实际上，子对象获得的只是一个内存地址，而不是真正拷贝，因此存在父对象被篡改的可能。

实例：

```
function cloneObj(obj){
    var newobj = {};
    //遍历传进的对象，将其属性赋值给新的对象
    for (var key in obj) {
        newobj[key] = obj[key];
    }
    return newobj;
}

var person = {
    name:'大毛',
    birthPlaces:["深圳","广州"],
};

var cobj = cloneObj(person);
cobj.birthPlaces.push("兰州");
console.log(person.birthPlaces);
console.log(cobj.birthPlaces);
```

5.3.2.2 深克隆

深克隆就是能够实现真正意义上的数组和对象的拷贝。它的实现并不难，只要递归调用"浅克隆"就行了。

实例：

```
//深克隆既可以克隆对象 也可以克隆数组， 对象数组
function deepCloneObj(obj) {
```

```
    //[].constructor === Array true
    var newobj = (obj.constructor === Array ? [] : {});
    //遍历传进的对象，将其属性赋值给新的对象
    for(var key in obj) {
        if(obj[key] === null) {
            //判断该值是否为null,因为typeof null也是"object"
            newobj[key] = null
        } else {
            //typeof 对象和数组 值都是 "object"
            if("object" === typeof obj[key]) { //判断该值是否是对象或者数组
                //深克隆
                newobj[key] = deepCloneObj(obj[key]);
            } else {
                //浅克隆
                newobj[key] = obj[key];
            }
        }
    }
    return newobj;
}

var person = {
    name:'大毛',
    birthPlaces:["深圳","广州"],
};
var cobj = deepCloneObj(person);
cobj.birthPlaces.push("兰州");
console.log(person.birthPlaces);
console.log(cobj.birthPlaces);
```

5.4 课程总结

- 对象常用属性和方法
- 对象的继承
- 对象冒充和克隆

5.5 实训

1、复习所讲内容。

2、将上面所讲代码自己写一遍。