27-第二十七章 面向对象oop 构造函数(一)

> Javascript是一种基于对象（object-based）的语言。但是，它又不是一种真正的面向对象编程（OOP）语言，因为它的语法中没有class（类）——es6以前是这样的
> 面向对象程序设计具有许多优点：
> 1、开发时间短，效率高，可靠性高，所开发的程序更强壮。由于面向对象编程的可重用性，可以在应用程序中大量采用成熟的类库，从而缩短了开发时间。
> 2、应用程序更易于维护、更新和升级。继承和封装使得应用程序的修改带来的影响更加局部化。

一、 面向对象四个基本特征

1. `抽象`：对象这个词，本身就是抽象的，也就是把客观事物抽象的成类
2. `封装`：封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。
3. `继承`:通过继承创建的新类称为“子类”或“派生类”。继承的过程，就是从一般到特殊的过程。
4. `多态`：对象的多功能,多方法，及方法

二、对象是什么？

对象有”`属性`“（property）和”`方法`“（method），

三 对象实例化方式

1、原始模式

```
var Car = {
    color: 'red',//车的颜色
    wheel: 4,//车轮数量
}

alert(Car.color);//red
```

生成两个实例对象

```
var Car = {
    color: 'red',
    wheel: 4,
}
var Car2 = {
    color: 'blue',
    wheel: 4,
}
```

这样的写法有两个缺点，一是如果多生成几个（100个！）实例，写起来就非常麻烦；二是实例与原型之间，没有任何办法，可以看出有什么联系。

二、 原始模式的改进

我们可以写一个函数，解决代码重复的问题。

```
function createCar(color,wheel) {
       return {
           color:color,
           wheel:wheel
       }
   }
//然后生成实例对象，就等于是在调用函数：

var cat1 = createCar("红色","4");
var cat2 = createCar("蓝色","4");

alert(cat1.color);//红色
```

三、 工厂模式

```
function createCar(color,wheel){//createCar工厂
    var obj = new Object;//或obj = {} 原材料阶段
     obj.color = color;//加工
     obj.wheel = wheel;//加工
     return obj;//输出产品
}
//实例化
var cat1 = createCar("红色","4");
var cat2 = createCar("蓝色","4");

alert(cat1.color);//红色
```

四、 构造函数模式

> 为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。
> 所谓"构造函数"，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

```
function CreateCar(color,wheel){//构造函数首字母大写
    //不需要自己创建对象了
     this.color = color;//添加属性，this指向构造函数的实例对象
     this.wheel = wheel;//添加属性

    //不需要自己return了

}

//实例化
var cat1 = new CreateCar("红色","4");
var cat2 = new CreateCar("蓝色","4");
alert(cat1.color);//红色
```

> new 函数构造内部变化：
>
> - 自动生成一个对象
> - this指向这个对象
> - 函数自动返回这个对象

五、 构造函数注意事项

1、此时CreateCar称之为`构造函数`，也可以称之`类`，构造函数就是类
2 、cat1，cat2均为CreateCar的`实例对象`
3、 CreateCar 构造函数中`this`指向CreateCar实例对象即 new CreateCar( )出来的对象
4、 必须带`new`
5、 构造函数`首字母大写`，这是规范，官方都遵循这一个规范，如Number() Array()

6、 `contructor`

> 这时cat1和cat2会自动含有一个constructor属性，指向它们的构造函数,即CreateCar。

```
1.      alert(cat1.constructor == CreateCar); //true
2.     alert(cat2.constructor == CreateCar); //true
```

7、 `instanceof` 运算符

> object instanceof constructor 运算符，验证构造函数与实例对象之间的关系。

```
1.  alert(cat1 instanceof CreateCar ); //true
2.  alert(cat2 instanceof CreateCar ); //true
```

六、 JavaScript值类型(基本类型)和引用类型：

（1）值类型：`数字`、`字符串`、`布尔值`、`null`、`undefined`。

（2）引用类型：`对象`、`数组`、`函数`。

> （1）值类型理解：对变量赋值`会开辟新的内存地址`
> 变量的交换等于在一个新的地方按照连锁店的规范标准（统一店面理解为相同的变量内容）新开一个分店，这样新开的店与其它旧店互不相关、各自运营。
> （2）引用类型理解：对变量赋值`指向相同的内存地址`
> 变量的交换等于把现有一间店的钥匙（变量引用地址）复制一把给了另外一个老板，此时两个老板同时管理一间店，两个老板的行为都有可能对一间店的运营造成影响。

七、 构造函数的问题：

构造函数方法很好用，但是存在一个浪费内存的问题。
请看，我们现在为再添加一个方法showWheel。那么，CreateCar就变成了下面这样：

```
function CreateCar(color,wheel){

      this.color = color;
      this.wheel = wheel;
      this.showWheel = function(){//添加一个新方法
         alert(this.wheel);
     }
}

//还是采用同样的方法，生成实例：
var cat1 = new CreateCar("红色","4");
var cat2 = new CreateCar("蓝色","4");
```

表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。那就是对于每一个实例对象，type属性和eat()方法都是一模一样的内容，每一次生成一个实例，都必须为重复的内容，多占用一些内存。这样既不环保，也缺乏效率。

```
alert(cat1.showWheel == cat2.showWheel); //false
```

八、 Prototype 原型

> Javascript规定，`每一个构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。`
> 这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。

```
function CreateCar(color,wheel){
    //属性写构造函数里面
     this.color = color;
     this.wheel = wheel;

}
    //方法写原型里面
 CreateCar.prototype.showWheel = function(){
     alert(this.wheel);
 }
 CreateCar.prototype.showName = function(){
     alert('车');
 }
//然后，生成实例。
var cat1 = new CreateCar("红色","4");
var cat2 = new CreateCar("蓝色","4");
alert( cat1.showName );//'车'
alert(cat1.showWheel == cat2.showWheel );//true
alert(cat1.showName == cat2.showName );//true
```

这时所有实例的showWheel属性和showName方法，其实都是同一个内存地址，指向prototype对象，因此就提高了运行效率。

```
alert(cat1.showWheel == cat2.showWheel); //true
```

九、面向过程 改写面向对象:

> 要求:不要出现函数嵌套
> `变量` -> `属性`
> `函数` -> `方法`
> 难点： `this指向`

案例： `用面向对象改写 拖拽`
案例： `用面向对象改写 轮播`