JavaScript的创世神话——一切源于对象

类 继承 原型 javascript **teren** 1 天前发布

> 《圣经》里的第一章创世纪中其中有一段经典记载上帝是如何创造人的。神说：“我们要照着我们的形象，按照我们的样式造人”。不谋而合的是，JavaScript中似乎也遵循着上帝的旨意去创造程序世界，一切皆对象的认知里背后是世间万物皆源于一个原型，一个统一的形式，一个柏拉图口中的理念.....

JavaScript的编程艺术也由此演绎开来~~~

目录：

1.面向对象编程的一些初步思考

"面向对象编程"（Object Oriented Programming，缩写为OOP）本身是一种编程的思维模式，它把世界的一切看作是对象的集合，世界的运转就是靠一个个对象分工、合作的结果。

有了这么一种宏观维度的编程认知，那么我们在编程时也就不会困在代码的死胡同了出不来。

给一段小代码做初步解释：我们在网页开发中，经常要进行事件绑定。

```
var btn = document.getElementById('button');
btn.addEventListener('click',function(){console.log('click me')})
```

初学时，笔者也是无意识的看到addEventListener()方法可以进行事件绑定以实现点击特定元素后，就可以实现需求就没有再往深层次去想太多；

随着学习的深入，在进入OOP的章节，我们就会发现“一切皆对象”这句话的深刻性。

上面的代码中，btn本身就一个对象，点击时该对象将会调用自身的方法addEventListener()去干事。

一个完整的解释就是谁（对象）干了什么（方法），btn被点击时干了输出‘click me ’。

当然这只是其中一个案例，JS的编程中这种编程模式贯通始终，如：

```
//1、定义一个对象
var sheep = {
  color: white,
  echo: function(){
      console.log('mae~~~')
  }
}
sheep.echo()
//这只羊发出mae的叫声

//2.设置对象的样式
$('.passage').css({
      width: '100px',
      color: 'pink'
  }
```

```
)
//这个名为passge的jquery对象被设置宽和色
......
```

总之，JavaScript的编程无法脱离对象而独存

2.类与对象

说了那么久，如果连对象是什么都没一个清晰的界定，那么在编程过程中还是存在模糊地带。

- **什么是对象**
  对象可以说是**对现实事物的抽象**，对象封装了属性和方法，属性值指的是对象的状态，方法指的是对象的行为。

比如，我们把NBA的巨星kobe抽象为javascript里的对象：

```
var kobe = {
    team: 'Los Angeles Lakers',
    position: 'shooting guard',
    ability :  function(){
        console.log('impress those who like basketball')
  }
}
```

现实世界的科比抽象为js中kobe这一对象，效力于洛杉矶湖人和位置是得分后卫是他的属性，能力是影响许多爱好篮球的人是他的方法。

- 什么是'类'(构造函数)

按照圣经的记载，在第一章的创世纪中写道：“神按照着自己的形象造人。”

现实世界的万物（对象）的演化不是凭空诞生的，它需要按照一个模板去构造出各种实例出来。

因此，类（构造函数）就是提供这么一种模板的'对象'（函数本身在js中就是对象），它是**对象的抽象**。

但是，js中并没有类的概念，而是通过构造函数替代了类的功能，为某一类的对象提供共同的属性和方法。

通过构造函数能够创造各种具有特定类的属性和方法的实例对象。

比如，定义一个篮球运动员的类(构造函数)：

```
//定义一个类，该类包含了一个篮球运动员应有的属性和方法
var BasketballPlayer = function(){
   this.height = '180+',
   this.muscle = true,
   this.ability = function(){
        console.log('shooting and passing ')
    }
}

//由类（构造函数）创造出3个实例对象
var kobe = new BasketballPlayer();
var james =new BasketballPlayer();
var curry =new BasketballPlayer();
```

这里有个小问题，构造函数BasketballPlayer又是有谁构造出来呢？看了下面的代码便知~~~

```
BasketballPlayer instanceof Object//true
构造函数或者是函数在js中天生就是构造函数Object的实例
```

所以说，所有的实例对象都有类（构造函数）创造出来，而所有的构造函数都是由**最为一般的类**，即Object构造出来，故**一切皆对象**。

【注】类在js中表现为构造函数，为了准确起见，下面统一称为构造函数，我们只需要知道二者起到的作用是一致就行。

3.构造函数

前面我们了解到，世间万物（实例对象）都是按照特定的模板（类或构造函数）构造的，而所有的构造函数都是由最一般的构造函数Object构造的。

但是，我们或许看到下面的代码会产生这么一种疑惑：
1.构造函数中的Book的this是干嘛的，有什么门道？
2.利用构造函数Book构造实例javascript时new起到什么作用
3.为什么我构造出一个实例对象后，这个构造函数能够返回一个对象给我
总结成一句就是：在整个造物的过程中，构造函数的运作机制是怎么样的？？？

```
var Book = function(){
    this.material = 'paper';
    this.color = 'white';
    this.utility = function(){
```

```
        console.log('IQ+');
  }
    this.whoAmI = function(){
      console.log(this)
  }
}

var javascript = new Book()
```

- this在构造函数的作用

  关键字this在js中显得十分重要，它在不同的运行环境（属性和方法的调用者）指向的环境（对象）不同，也就是说this的指向是动态的，但是无论this的指向是谁，只要清楚属性和方法的调用者是谁那么this就指向谁。

```
//在浏览器全局环境下，即window对象下
var print = function(){
      console.log(this)
 }
print()//this指向Window，因为这是Window对象调用了print方法

//在特定对象的环境下
var o = {
    print: function(){
      console.log(this)
  }
}
o.print()//this指向o，因为这是o对象调用print方法
```

因此，回到构造函数中的this来，当执行 var javascript = new Book()时，此时是javascript这个实例对象调用了构造函数Book，函数内部的this指向javascript这一实例对象

```
javascript.whoAmI()//this此时指向javascript对象
```

【注】更多this的知识详见what's this???

- new命令的原理

  接下来谈一谈new命令的原理。

new命令的作用，就是执行构造函数，返回一个实例对象

与普通的正常调用函数不同，在函数执行前面附加new命令，函数执行以下步骤：

> 1.创建一个空对象，作为将要返回的实例对象；
>
> ```
> 2.将这个空对象的原型``__proto__``指向构造函数的prototype属性以实现继承机制
> 3.将这个空对象赋值给函数内部的this关键字
> 4.开始执行构造函数内部的代码
> ```

- 原型对象

  上面我们基本了解构造函数在创造实例对象时的部分运作机制，明白了this关键字和new命令在构造实例时所起的作用。

现在有一个**最最重要的疑问**是实例对象究竟是如何继承构造函数中设定好的属于该类的共有的属性和方法呢？

prototype object

JavaScript中每个实例对象继承自另一个对象，后者被称为原型对象，原型对象上的属性和方法都能被派生对象共享，这就是JavaScript著名的继承机制的基本设计。

先上一段代码用于讲解：

```
function Dog(name,color){
  this.name = name;
  this.color = color;
}

Dog.prototype.spark = function(){
    console.log('Wow~~~')
}
var tony = new Dog('tony','white')
```

1.通过构造函数Dog生成的实例对象tony，会自动为实例对象tony分配原型对象；

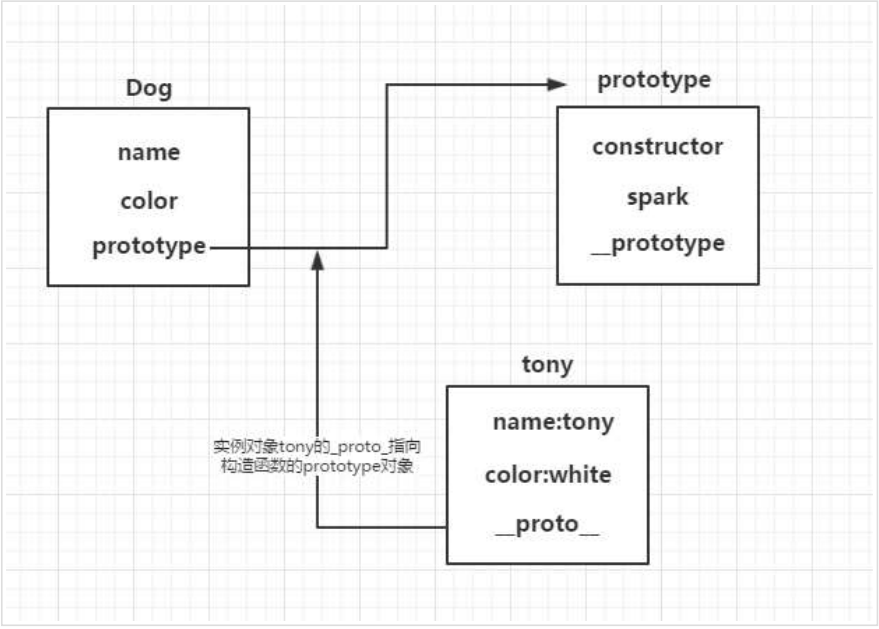
2.每一个构造函数都有一个prototype属性，该属性就是实例对象的原型对象

```
console.dir(Dog)
▼function Dog(name, color)
    arguments: null
    caller: null
    length: 2
    name: "Dog"
  ▼prototype: Object
    ▶constructor: function Dog(name,color)
    ▶spark: function ()
    ▶__proto__: Object
  ▶__proto__: function ()
  ▶<function scope>
```

3.实例对象内部有一个 `__proto__` 属性，该属性在被构造函数一创造出来就指向构造函数的protype属性

```
▼Dog
    color: "white"
    name: "tony"
  ▼__proto__: Object
    ▶constructor: function Dog(name,color)
    ▶spark: function ()
    ▶__proto__: Object
```

这样一来，我们通过构造函数Dog中的原型对象prototype实现了实例对象tony对Dog的共有属性和方法的继承。



因此，我们可以得出的思考是，原型对象定义所有实例对象的共有的属性和方法，所有的实例对象无非是从原型对象衍生出的子对象，只不过在后来给它添加了特有的属性和方法罢了。

prototype chain

实例对象tony的 `__proto__` 指向构造函数Dog的prototype对象，因此继承了Dog中prototype的属性和方法；

而构造函数本身也存在 `__proto__` 指向更一般的函数（本质上也是对象）的prototype对象；

```
▼function Dog(name, color)
    arguments: null
    caller: null
    length: 2
    name: "Dog"
  ▶prototype: Object
  ▼__proto__: function ()
    ▶apply: function apply()
      arguments: (...)
    ▶get arguments: function ThrowTypeError()
    ▶set arguments: function ThrowTypeError()
    ▶bind: function bind()
    ▶call: function call()
      caller: (...)
    ▶get caller: function ThrowTypeError()
    ▶set caller: function ThrowTypeError()
    ▶constructor: function Function()
      length: 0
      name: ""
    ▶toString: function toString()
    ▶Symbol(Symbol.hasInstance): function [Symbol.hasInstance]()
    ▶__proto__: Object
```

更进一步，该函数也存在 `__proto__` 指向最一般的构造函数Object的prototype对象

这种层层嵌套的关系形成一条原型链（prototype chain）。

一只名叫tony的狗，首先继承了构造函数Dog的原型对象，而Dog的原型对象中的 `__proto__` 有继承了函数的原型对象，函数对象中的 `__proto__` 有继承了Oject的原型对象。

这里再一次体现了构造函数Object的威力，所有的对象无非都是Object的衍生，均继承Object.prototype的属性和方法,更加深刻表达“一切皆对象”的思想。

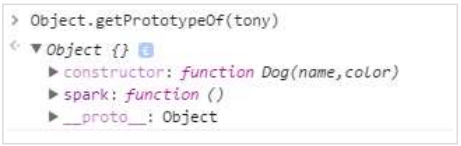
4.一些相关的方法

- **instanceof 运算符**
  instanceof运算符返回一个布尔值，表示指定对象是否为某个构造函数的实例

```
tony instanceof Dog//true
or
Dog.prototype.isPrototypeOf(tony)//true
```

- **Object.getPrototypeOf()**

Object.getPrototypeOf()返回一个对象的原型，这是**获取原型对象的标准方法**



- **Object.setPrototypeOf()**
  Object.setPrototypeOf

方法可以为现有对象设置原型，返回一个新对象，该方法接受两个参数，第一个是现有对象，第二个是原型对象。

```
var foo = { x:1 };
var bar = Object.setPrototypeOf({},foo)
bar.x === 1//true
```

我们之前以new命令去构建实例对象，本质上就是把一个空对象的**proto**指向构造函数的prototype，然后在实例对象上执行构造函数

```
var  Person = function(){
    this.race = 'monkey'
};
var Asian = new Person();
//等价于
var Asian = Object.setPrototypeOf({},Person.prototype);
Person.call(Asian)
```

- **Object.create()**
  Object.create方法用于从原型对象生成新的实例对象，可以替代new命令

```
var  Person = {
    race: 'monkey'
}
var Asian = Object.create(Person)
//等价于
var Person = function(){
    this.race='monkey'
}
var Asian = new Person()
```

- **Object.isPrototypeOf()**
  对象实例的isPrototypeOf方法，用来判断一个对象是否是另一个对象的原型。

```
var obj1 = {};
var obj2 = Object.create(obj1);
obj1.isPrototypeOf(obj2)//true
```

- **Object.prototype.hasOwnProperty()**
  对象实例的hasOwnProperty方法返回一个布尔值，用于判断某个属性定义在对象自身，还是定义在原型链上。

```
var o = {
  name:'teren'
}
o.hasOwnProperty('name')//true
o.hasOwnProperty('toString')//false
Object.getPrototypeOf(o).hasOwnProperty('toString')//true
```

参考文献：

阮一峰-JavaScript标准参考教程
饥人谷学习笔记

1 天前发布　更多

你可能感兴趣的文章

Javascript 设计模式读书笔记(三)——继承　16 收藏，2.4k 浏览
Javascript类继承浅析　11 收藏，1.7k 浏览
如何更好地理解Javascript对象的自有属性和原型继承属性　3 收藏，1.1k 浏览

讨论区


teren
13 声望
关注作者

发布于专栏

Not Just a Programer But a Artist

编程之道，更是艺术之道

0 人关注　关注专栏

相关收藏夹


JavaScript
5 个条目 | 0 人关注

分享扩散：