和其他高级语言一样 javascript 中也有 new 运算符，我们知道 new 运算符是用来实例化一个类，从而在内存中分配一个实例对象。 但在 javascript 中，万物皆对象,为什么还要通过 new 来产生对象？ 本文将带你一起来探索 javascript 中 new 的奥秘...

一、认识 new 运算符：

```
function Animal(name){
    this.name = name;
}
Animal.color = "black";
Animal.prototype.say = function(){
    console.log("I'm " + this.name);
};
var cat = new Animal("cat");

console.log(
   cat.name,  //cat
   cat.height //undefined
);
cat.say(); //I'm cat

console.log(
   Animal.name, //Animal
   Animal.color //back
);
Animal.say(); //Animal.say is not a function
```

如果你能理解上面输出的结果,说明你已非常了解 js 中 new 和 this 的运行机制,请忽略本文!

我们将通过解析这个例子来加深你对 js 中 new 运算符的理解! 【如果你对 js 的 this 还不了解，请先阅读：JS 作用域和 this 关键字】

1、代码解读

1-3 行创建了一个函数 Animal,并在其 this 上定义了属性:name,name 的值是函数被执行时的形参。

第 4 行在 Animal 对象（Animal 本身是一个函数对象）上定义了一个静态属性:color,并赋值“black”

5-7 行在 Animal 函数的原型对象 prototype 上定义了一个 say()方法，say 方法输出了 this 的 name 值。

第 8 行通过 new 关键字创建了一个新对象 cat

10-14 行 cat 对象尝试访问 name 和 color 属性，并调用 say 方法。

16-20 行 Animal 对象尝试访问 name 和 color 属性，并调用 say 方法。

2、重点解析

第 8 行代码是关键：

```
var cat = new Animal("cat");
```

JS 引擎执行这句代码时，在内部做了很多工作，用伪代码模拟其工作流程如下：

```
1  var obj = {};
2  obj.__proto__ = Animal.prototype;
3  var result = Animal.call(obj,"cat");
4  return typeof result === 'obj'? result : obj;
```

（1）创建一个空对象 obj;

（2)把 obj 的__proto__ 指向 Animal 的原型对象 prototype,此时便建立了 obj 对象的原型链：obj->Animal.prototype->Object.prototype->null

【如果你不了解 JS 原型链，请先阅读：JS 原型和原型链】

（3）在 obj 对象的执行环境调用 Animal 函数并传递参数“cat”。 相当于 var result = obj.Animal("cat")。

当这句执行完之后，obj 便产生了属性 name 并赋值为"cat"。【关于 JS 中 call 的用法请阅读：JS 的 call 和 apply】

（4）考察第 3 步返回的返回值，如果无返回值或者返回一个非对象值，则将 obj 返回作为新对象；否则会将返回值作为新对象返回。

理解 new 的运行机制以后，我们知道 cat 其实就是过程（4）的返回值，因此我们对 cat 对象的认知就多了一些：

cat 的原型链是：cat->Animal.prototype->Object.prototype->null

cat 上新增了一个属性：name

分析完了 cat 的产生过程，我们再看看输出结果：

cat.name -> 在过程（3）中，obj 对象就产生了 name 属性。因此 cat.name 就是这里的 obj.name

cat.color -> cat 会先查找自身的 color，没有找到便会沿着原型链查找，在上述例子中，我们仅在 Animal 对象上定义了 color,并没有在其原型链上定义，因此找不到。

cat.say -> cat 会先查找自身的 say 方法，没有找到便会沿着原型链查找，在上述例子中，我们在 Animal 的 prototype 上定义了 say,因此在原型链上找到了 say 方法。

另外，在 say 方法中还访问 this.name，这里的 this 指的是其调用者 obj,因此输出的是 obj.name 的值。

对于 Animal 来说，它本身也是一个对象，因此，它在访问属性和方法时也遵守上述查找规则，所以：

Animal.color -> "black"

Animal.name -> "Animal"， Animal 先查找自身的 name,找到了 name,注意：但这个 name 不是我们定义的 name,而是函数对象内置的属性。

一般情况下，函数对象在产生时会内置 name 属性并将函数名作为赋值（仅函数对象）。

Animal.say -> Animal 在自身没有找到 say 方法，也会沿着其原型链查找，话说 Animal 的原型链是什么呢？

```
> Animal.__proto__
< function () {}
> Animal.__proto__ == Function.prototype
< true
> Function.prototype.__proto__ == Object.prototype
< true
> Object.prototype.__proto__
< null
>
```

从测试结果看：Animal 的原型链是这样的：

Animal->Function.prototype->Object.prototype->null

因此 Animal 的原型链上没有定义 say 方法！

二、new 存在的意义

认识了 new 运算符之后，我们再回到开篇提到的问题：JS 中万物皆对象，为什么还要通过 new 来产生对象？要弄明白这个问题，我们首先要搞清楚 cat 和 Animal 的关系。

通过上面的分析，我们发现 cat 继承了 Animal 中的部分属性，因此我们可以简单的理解：Animal 和 cat 是继承关系。

另一方面，cat 是通过 new 产生的对象，那么 cat 到底是不是 Animal 的实例对象？ 我们先来了解一下 JS 是如何来定义“实例对象”的？

```
A instanceof B
```

如果上述表达式为 true,JS 认为 A 是 B 的实例对象，我们用这个方法来判断一下 cat 和 Animal

```
cat instanceof Animal; //true
```

从执行结果看：cat 确实是 Animal 实例，要想证实这个结果，我们再来了解一下 JS 中 instanceof 的判断规则：

```
1 var L = A.__proto__;
2 var R = B.prototype;
3 if(L === R)
4   return true;
```

如果 A 的__proto__ 等价于 B 的 prototype，就返回 true

在 new 的执行过程（2）中，cat 的__proto__指向了 Animal 的 prototype，所以 cat 和 Animal 符合 instanceof 的判断结果。因此，我们认为：cat 是 Animal 的实例对象。

简单的总结语

在 javascript 中，通过 new 可以产生原对象的一个实例对象，而这个实例对象继承了原对象的属性和方法。因此，new 存在的意义在于它实现了 javascript 中的继承，而不仅仅是实例化了一个对象！

finish!

update at 2015-12-15 13:02:56

update at 2015-12-16 15:31:59