**概论、范式与思想**

在进行ECMAScript中的OOP技术分析之前，我们有必要掌握一些OOP基本的特征，并澄清概论中的主要概念。

ECMAScript支持包括结构化、面向对象、函数式、命令式等多种编程方式，某些情况下还支持面向方面编程；但本文是讨论面向对象编程，所以来给出ECMAScript中面向对象编程的定义:

ECMAScript是基于原型实现的面向对象编程语言。

基于原型的OOP和基于静态类的方式直接有很多差异。 让我们一起来看看他们直接详细的差异。

**基于类特性和基于原型**

注意，在前面一句很重要的一点已经指出的那样-完全基于静态类。 随着“静态”一词，我们了解静态对象和静态类，强类型（虽然不是必需的）。

关于这种情况，很多论坛上的文档都有强调这是他们反对将在JavaScript里将“类与原型”进行比较的主要原因，尽管他们在实现上的有所不同（例如基于动态类的Python和Ruby）不是太反对的重点（某些条件写，尽管思想上有一定不同，但JavaScript没有变得那么另类），但他们反对的重点是静态类和动态原型(statics + classes vs. dynamics + prototypes)，确切地说，一个静态类（例如：C + +，JAVA）和他的属下及方法定义的机制可以让我们看到它和基于原型实现的准确区别。

但是，让我们来一个一个列举一下。 让我们考虑总则和这些范式的主要概念。

**基于静态类**

在基于类的模型中，有个关于类和实例的概念。 类的实例也常常被命名为对象或范例 。

**类与对象**

类代表了一个实例（也就是对象）的抽象。在这方面有点像数学，但我们一把称之为类型（type）或分类（classification）。

例如（这里和下面的例子都是伪代码）：

C = Class {a, b, c} // 类C, 包括特性a, b, c

实例的特点是：属性（对象描述 ）和方法（对象活动）。特性本身也可视为对象：即属性是否可写的，可配置，可设置的（getter/setter）等。因此，对象存储了状态 （即在一个类中描述的所有属性的具体值），类为他们的实例定义了严格不变的结构（属性）和严格不变的行为（方法）。

C = Class {a, b, c, method1, method2}  
   
c1 = {a: 10, b: 20, c: 30} // 类C是实例：对象с1  
c2 = {a: 50, b: 60, c: 70} // 类C是实例：对象с2，拥有自己的状态（也就是属性值）

**层次继承**

为了提高代码重用，类可以从一个扩展为另一个，在加上额外的信息。 这种机制被称为（分层）继承 。

D = Class extends C = {d, e} // {a, b, c, d, e}  
d1 = {a: 10, b: 20, c: 30, d: 40, e: 50}

在类的实例上调用方的时候，通常会现在原生类本书就查找该方法，如果没找到就到直接父类去查找，如果还没找到，就到父类的父类去查找（例如严格的继承链上），如果查到继承的顶部还没查到，那结果就是：该对象没有类似的行为，也没办法获取结果。

d1.method1() // D.method1 (no) -> C.method1 (yes)  
d1.method5() // D.method5 (no) -> C.method5 (no) -> no result

与在继承里方法不复制到一个子类相比，属性总是被复杂到子类里的。 我们可以看到子类D继承自父类C类：属性a,b,c是复制过去了，D的结构是{a, b, c, d, e} } 。然而，方法{method1, method2}是没有复制过去，而是继承过去的。 因此，也就是说如果一个很深层次的类有一些对象根本不需要的属性的话，那子类也有拥有这些属性。

**基于类的关键概念**

因此，我们有如下关键概念：

1. 创建一个对象之前，必须声明类，首先有必要界定其类
2. 因此，该对象将由抽象成自身“象形和相似性”（结构和行为）的类里创建
3. 方法是通过了严格的，直接的，一成不变的继承链来处理
4. 子类包含了继承链中所有的属性（即使其中的某些属性是子类不需要的）;
5. 创建类实例，类不能（因为静态模型）来改变其实例的特征（属性或方法）;
6. 实例（因为严格的静态模型）除了有该实例所对应类里声明的行为和属性以外，是不能额外的行为或属性的。

让我们看看在JavaScript里如何替代OOP模型，也就是我们所建议的基于原型的OOP。

**基于原型**

这里的基本概念是动态可变对象。转换（完整转换，不仅包括值，还包括特性）和动态语言有直接关系。下面这样的对象可以独立存储他们所有的特性（属性，方法）而不需要的类。

object = {a: 10, b: 20, c: 30, method: fn};  
object.a; // 10  
object.c; // 30  
object.method();

此外，由于动态的，他们可以很容易地改变（添加，删除，修改）自己的特性：

object.method5 = function () {...}; // 添加新方法  
object.d = 40; // 添加新属性 "d"  
delete object.c; // 删除属性 "с"  
object.a = 100; // 修改属性 "а"  
   
// 结果是: object: {a: 100, b: 20, d: 40, method: fn, method5: fn};

也就是说，在赋值的时候，如果某些特性不存在，则创建它并且将赋值与它进行初始化，如果它存在，就只是更新。

在这种情况下，代码重用不是通过扩展类来实现的，（请注意，我们没有说类没办法改变，因为这里根本没有类的概念），而是通过原型来实现的。

原型是一个对象，它是用来作为其他对象的原始copy，或者如果一些对象没有自己的必要特性，原型可以作为这些对象的一个委托而当成辅助对象。

**基于委托**

任何对象都可以被用来作为另一个对象的原型对象，因为对象可以很容易地在运行时改变它的原型动态。

注意，目前我们正在考虑的是概论而不是具体实现，当我们在ECMAScript中讨论具体实现时，我们将看到他们自身的一些特点。

例（伪代码）：

x = {a: 10, b: 20};  
y = {a: 40, c: 50};  
y.[[Prototype]] = x; // x是y的原型  
   
y.a; // 40, 自身特性  
y.c; // 50, 也是自身特性  
y.b; // 20 – 从原型中获取: y.b (no) -> y.[[Prototype]].b (yes): 20  
   
delete y.a; // 删除自身的"а"  
y.a; // 10 – 从原型中获取  
   
z = {a: 100, e: 50}  
y.[[Prototype]] = z; // 将y的原型修改为z  
y.a; // 100 – 从原型z中获取  
y.e // 50, 也是从从原型z中获取  
   
z.q = 200 // 添加新属性到原型上  
y.q // 修改也适用于y

这个例子展示了原型作为辅助对象属性的重要功能和机制，就像是要自己的属性一下，和自身属性相比，这些属性是委托属性。这个机制被称为委托，并且基于它的原型模型是一个委托的原型（或基于委托的原型 ） 。引用的机制在这里称为发送信息到对象上，如果这个对象得不到响应就会委托给原型来查找（要求它尝试响应消息）。

在这种情况下的代码重用被称为基于委托的继承或基于原型的继承。由于任何对象可以当成原型，也就是说原型也可以有自己的原型。 这些原型连接在一起形成一个所谓的原型链。 链也像静态类中分层次的，但是它可以很容易地重新排列，改变层次和结构。

x = {a: 10}  
   
y = {b: 20}  
y.[[Prototype]] = x  
   
z = {c: 30}  
z.[[Prototype]] = y  
   
z.a // 10  
   
// z.a 在原型链里查到:  
// z.a (no) ->  
// z.[[Prototype]].a (no) ->  
// z.[[Prototype]].[[Prototype]].a (yes): 10

如果一个对象和它的原型链不能响应消息发送，该对象可以激活相应的系统信号，可能是由原型链上其它的委托进行处理。

该系统信号，在许多实现里都是可用的，包括基于括动态类的系统：Smalltalk中的＃doesNotUnderstand，Ruby中的​​method\_missing；Python中的\_\_getattr\_\_，PHP中的\_\_call；和ECMAScript中的\_\_noSuchMethod\_\_实现，等等。

例（SpiderMonkey的ECMAScript的实现）：

var object = {  
   
 // catch住不能响应消息的系统信号  
 \_\_noSuchMethod\_\_: function (name, args) {  
 alert([name, args]);  
 if (name == 'test') {  
 return '.test() method is handled';  
 }  
 return delegate[name].apply(this, args);  
 }  
   
};  
   
var delegate = {  
 square: function (a) {  
 return a \* a;  
 }  
};  
   
alert(object.square(10)); // 100  
alert(object.test()); // .test() method is handled

也就是说，基于静态类的实现，在不能响应消息的情况下，得出的结论是：目前的对象不具有所要求的特性，但是如果尝试从原型链里获取，依然可能得到结果，或者该对象经过一系列变化以后拥有该特性。

关于ECMAScript，具体的实现就是：使用基于委托的原型。 然而，正如我们将从规范和实现里看到的，他们也有自身的特性。

**Concatenative模型**

老实说，有必要在说句话关于另外一种情况（尽快在ECMASCript没有用到）：当原型从其它对象复杂原来代替原生对象这种情况。这种情况代码重用是在对象创建阶段对一个对象的真正复制（克隆）而不是委托。这种原型被称为concatenative原型。复制对象所有原型的特性，可以进一步完全改变其属性和方法,同样作为原型可以改变自己（在基于委托的模型中，这个改变不会改变现有存在的对象行为，而是改变它的原型特性）。 这种方法的优点是可以减少调度和委托的时间，而缺点是内存使用率搞。

**Duck类型**

回来动态弱类型变化的对象，与基于静态类的模型相比，检验它是否可以做这些事和对象有什么类型（类）无关，而是是否能够相应消息有关（即在检查以后是否有能力做它是必须的） 。

例如：

// 在基于静态来的模型里  
if (object instanceof SomeClass) {  
 // 一些行为是运行的  
}  
   
// 在动态实现里  
// 对象在此时是什么类型并不重要  
// 因为突变、类型、特性可以自由重复的转变。  
// 重要的对象是否可以响应test消息   
if (isFunction(object.test)) // ECMAScript  
   
if object.respond\_to?(:test) // Ruby  
   
if hasattr(object, 'test'): // Python

这就是所谓的Dock类型 。 也就是说，物体在check的时候可以通过自己的特性来识别，而不是对象在层次结构中的位置或他们属于任何具体类型。

**基于原型的关键概念**

让我们来看一下这种方式的主要特点：

1. 基本概念是对象
2. 对象是完全动态可变的（理论上完全可以从一个类型转化到另一个类型）
3. 对象没有描述自己的结构和行为的严格类，对象不需要类
4. 对象没有类类但可以可以有原型，他们如果不能响应消息的话可以委托给原型
5. 在运行时随时可以改变对象的原型;
6. 在基于委托的模型中，改变原型的特点，将影响到与该原型相关的所有对象;
7. 在concatenative原型模型中，原型是从其他对象克隆的原始副本，并进一步成为完全独立的副本原件，原型特性的变换不会影响从它克隆的对象
8. 如果不能响应消息，它的调用者可以采取额外的措施（例如，改变调度）
9. 对象的失败可以不由它们的层次和所属哪个类来决定，而是由当前特性来决定

不过，还有一个模型，我们也应该考虑。

**基于动态类**

我们认为，在上面例子里展示的区别“类VS原型 ”在这个基于动态类的模型中不是那么重要，（尤其是如果原型链是不变的，为更准确区分，还是有必要考虑一个静态类）。 作为例子，它也可以使用Python或Ruby（或其他类似的语言）。 这些语言都使用基于动态类的范式。 然而，在某些方面，我们是可以看到基于原型实现的某些功能。

在下面例子中，我们可以看到仅仅是基于委托的原型，我们可以放大一个类（原型），从而影响到所有与这个类相关的对象，我们也可以在运行时动态地改变这个对象的类（为委托提供一个新对象）等等。

# Python  
   
class A(object):  
   
 def \_\_init\_\_(self, a):  
 self.a = a  
   
 def square(self):  
 return self.a \* self.a  
   
a = A(10) # 创建实例  
print(a.a) # 10  
   
A.b = 20 # 为类提供一个新属性  
print(a.b) # 20 – 可以在"a"实例里访问到  
   
a.b = 30 # 创建a自身的属性  
print(a.b) # 30  
   
del a.b # 删除自身的属性  
print(a.b) # 20 - 再次从类里获取（原型）  
   
# 就像基于原型的模型  
# 可以在运行时改变对象的原型  
   
class B(object): # 空类B  
 pass  
   
b = B() # B的实例  
   
b.\_\_class\_\_ = A # 动态改变类（原型）  
   
b.a = 10 # 创建新属性  
print(b.square()) # 100 - A类的方法这时候可用  
   
# 可以显示删除类上的引用  
del A  
del B  
   
# 但对象依然有隐式的引用，并且这些方法依然可用  
print(b.square()) # 100  
   
# 但这时候不能再改变类了  
# 这是实现的特性  
b.\_\_class\_\_ = dict # error

Ruby中的实现也是类似的：也使用了完全动态的类（顺便说一下在当前版本的Python中，与Ruby和ECMAScript的对比，放大类（原型）不行的），我们可以彻底改变对象（或类）的特性（在类上添加方法/属性，而这些变化会影响已经存在的对象），但是，它不能的动态改变一个对象的类。

但在此之前，我们还得再看一下在一些OOP里有的“语法糖”，因为很多之前关于JavaScript的文章往往会文这些问题。

本节唯一需要注意的错误句子是：“JavaScript不是类，它有原型，可以代替类”。 非常有必要知道并非所有基于类的实现都是完全不一样的,即便我们可能会说“JavaScript是不同的”，但也有必要考虑（除了“类”的概念）还有其他相关的特性呢。

**各种OOP实现的其它特性**

本节我们简要介绍一下其它特性和各种OOP实现中关于代码重用的方式，也包括ECMAScript中的OOP实现。 原因是，之前出现的关于JavaScript中关于OOP的实现是有一些习惯性的思维限制，唯一主要的要求是，应该在技术上和思想上加以证明。不能说没发现和其它OOP实现里的语法糖功能，就草率认为JavaScript不是不是纯粹的OOP语言，这是不对滴。

**多态**

在ECMAScript中对象有几种含义的多态性。

例如，一个函数可以应用于不同的对象，就像原生对象的特性（因为这个值在进入执行上下文时确定的）：

function test() {  
 alert([this.a, this.b]);  
}  
   
test.call({a: 10, b: 20}); // 10, 20  
test.call({a: 100, b: 200}); // 100, 200  
   
var a = 1;  
var b = 2;  
   
test(); // 1, 2

不过，也有例外：Date.prototype.getTime()方法，根据标准这个值总是应该有一个日期对象，否则就会抛出异常。

alert(Date.prototype.getTime.call(new Date())); // time  
alert(Date.prototype.getTime.call(new String(''))); // TypeError

所谓函数定义时的参数多态性也就等价于所有数据类型，只不过接受多态性参数（例如数组的.sort排序方法和它的参数——多态的排序功能）。顺便说一下，上面的例子也可以被视为是一种参数多态性。

原型里方法可以被定义为空，所有创建的对象应重新定义（实现）该方法（即“一个接口（签名），多个实现”）。

多态性和我们上面提到的Duck类型是有关的：即对象的类型和在层次结构中的位置不是那么重要，但如果它有所有必要的特征，它可以很容易地接受（即通用接口很重要，实现则可以多种多样）。

**封装**

关于封装，往往会有错误的看法。本节我们讨论一下一些OOP实现里的语法糖——也就是众所周知的修饰符：在这种情况下，我们将讨论一些OOP实现便捷的“糖” -众所周知的修饰符：private,protected和public（或者称为对象的访问级别或访问修饰符）。

在这里我要提醒一下封装的主要目的：封装是一个抽象的增加，而不是选拔个直接往你的类里写入一些东西的隐藏“恶意黑客”。

这是一个很大的错误：为了隐藏使用隐藏。

访问级别（private,protected和public），为了方便编程在很多面向对象里都已经实现了（真的是非常方便的语法糖），更抽象地描述和构建系统。

这些可以在一些实现里看出（如已经提到的Python和Ruby）。一方面（在Python中），这些\_\_private \_protected属性（通过下划线这个命名规范），从外部不可访问。 另一方面，Python可以通过特殊的规则从外部访问（\_ClassName\_\_field\_name）。

class A(object):  
   
 def \_\_init\_\_(self):  
 self.public = 10  
 self.\_\_private = 20  
   
 def get\_private(self):  
 return self.\_\_private  
   
# outside:  
   
a = A() # A的实例  
   
print(a.public) # OK, 30  
print(a.get\_private()) # OK, 20  
print(a.\_\_private) # 失败，因为只能在A里可用  
   
# 但在Python里，可以通过特殊规则来访问  
   
print(a.\_A\_\_private) # OK, 20

在Ruby里：一方面有能力来定义private和protected的特性，另一方面，也有特殊的方法（ 例如instance\_variable\_get，instance\_variable\_set，send等）获取封装的数据。

class A  
   
 def initialize  
 @a = 10  
 end  
   
 def public\_method  
 private\_method(20)  
 end  
   
private  
   
 def private\_method(b)  
 return @a + b  
 end  
   
end  
   
a = A.new # 新实例  
   
a.public\_method # OK, 30  
   
a.a # 失败, @a - 是私有的实例变量  
   
# "private\_method"是私有的，只能在A类里访问  
   
a.private\_method # 错误  
   
# 但是有特殊的元数据方法名，可以获取到数据  
   
a.send(:private\_method, 20) # OK, 30  
a.instance\_variable\_get(:@a) # OK, 10

最主要的原因是，程序员自己想要获得的封装（请注意，我特别不使用“隐藏”）的数据。 如果这些数据会以某种方式不正确地更改或有任何错误,则全部责任都是程序员，但不是简单的“拼写错误”或“随便改变某些字段”。 但如果这种情况很频繁，那就是很不好的编程习惯和风格 ，因为通常值用公共的API来和对象“交谈”。

重复一下，封装的基本目的是一个从辅助数据的用户中抽象出来，而不是一个防止黑客隐藏数据。 更严重的，封装不是用private修饰数据而达到软件安全的目的。

封装辅助对象（局部），我们用最小的代价、本地化和预测性变化来问为公共接口的行为变化提供可行性，这也正是封装的目的。

另外setter方法​​的重要目的是抽象复杂的计算。 例如，element.innerHTML这个setter——抽象的语句——“现在这个元素内的HTML是如下内容”，而在 innerHTML属性的setter函数将难以计算和检查。 在这种情况下，问题大多涉及到抽象 ，但封装也会发生。

封装的概念不仅仅只与OOP相关。 例如，它可以是一个简单的功能，只封装了各种计算，使得其抽象（没有必要让用户知道，例如函数Math.round（... ...）是如何实现的，用户只是简单地调用它）。 它是一种封装，注意，我没有说他是“private, protected和public”。

ECMAScript规范的当前版本，没有定义private, protected和public修饰符。

然而，在实践中是有可能看到有些东西被命名为“模仿JS封装”。 一般该上下文的目的是（作为一个规则，构造函数本身）使用。 不幸的是，经常实施这种“模仿”，程序员可以产生伪绝对非抽象的实体设置“getter / setter方法”（我再说一遍，它是错误的）：

function A() {  
   
 var \_a; // "private" a  
   
 this.getA = function \_getA() {  
 return \_a;  
 };  
   
 this.setA = function \_setA(a) {  
 \_a = a;  
 };  
   
}  
   
var a = new A();  
   
a.setA(10);  
alert(a.\_a); // undefined, "private"  
alert(a.getA()); // 10

因此，每个人都明白，对于每个创建的对象，对于的getA/setA方法也创建了，这也是导致内存增加的原因（和原型定义相比）。 虽然，理论上第一种情况下可以对对象进行优化。

另外，一些JavaScript的文章经常提到“私有方法”的概念，注意：ECMA-262-3标准里没有定义任何关于“私有方法”的概念。

但是，某些情况下它可以在构造函数中创建，因为JS是意识形态的语言——对象是完全可变的并且有独特的特性（在构造函数里某些条件下，有些对象可以得到额外的方法，而其他则不行）。

此外，在JavaScript里，如果还是把封装曲解成为了不让恶意黑客在某些自动写入某些值的一种理解来代替使用setter方法，那所谓的“隐藏(hidden)”和“私有(private)”其实没有很“隐藏”，，有些实现可以通过调用上下文到eval函数（可以在SpiderMonkey1.7上测试）在相关的作用域链（以及相应的所有变量对象）上获取值）。

eval('\_a = 100', a.getA); // 或者a.setA,因为"\_a"两个方法的[[Scope]]上  
a.getA(); // 100

或者，在实现中允许直接进入活动对象（例如Rhino），通过访问该对象的相应属性可以改变内部变量的值：

// Rhino  
var foo = (function () {  
 var x = 10; // "private"  
 return function () {  
 print(x);  
 };  
})();  
foo(); // 10  
foo.\_\_parent\_\_.x = 20;  
foo(); // 20

有时，在JavaScript里通过在变量前加下划线来达到“private”和“protected”数据的目的（但与Python相比，这里只是命名规范）：

var \_myPrivateData = 'testString';

对于括号括住执行上下文是经常使用，但对于真正的辅助数据，则和对象没有直接关联，只是方便从外部的API抽象出来：

(function () {  
   
 // 初始化上下文  
   
})();

**多重继承**

多继承是代码重用改进的一个很方便的语法糖（如果我们一次能继承一个类，为什么不能一次继承10个？）。 然而由于多重继承有一些不足，才导致在实现中没有流行起来。

ECMAScript不支持多继承（即只有一个对象，可以用来作为一个直接原型），虽然其祖先自编程语言有这样的能力。 但在某些实现中(如SpiderMonkey)使用\_\_noSuchMethod\_\_可以用于管理调度和委托来替代原型链。

**Mixins**

Mixins是代码重用的一种便捷方式。 Mixins已建议作为多重继承的替代品。 这些独立的元素都可以与任何对象进行混合来扩展它们的功能（因此对象也可以混合多个Mixins）。 ECMA-262-3规范没有定义“Mixins”的概念，但根据Mixins定义以及ECMAScript拥有动态可变对象，所以使用Mixins简单地扩充特性是没有障碍的。

典型的例子：

// helper for augmentation  
Object.extend = function (destination, source) {  
 for (property in source) if (source.hasOwnProperty(property)) {  
 destination[property] = source[property];  
 }  
 return destination;  
};  
   
var X = {a: 10, b: 20};  
var Y = {c: 30, d: 40};  
   
Object.extend(X, Y); // mix Y into X  
alert([X.a, X.b, X.c, X.d]); 10, 20, 30, 40

请注意，我采取在ECMA-262-3中被提及过的引号中的这些定义（“mixin”，“mix”），在规范里并没有这样的概念，而且不是mix而是常用的通过新特性去扩展对象。（Ruby中mixins的概念是官方定义的，mixin创建了一个包含模块的一个引用来代替简单复制该模块的所有属性到另外一个模块上——事实上是：为委托创建一个额外的对象（原型））。

**Traits**

Traits和mixins的概念相似，但它有很多功能（根据定义，因为可以应用mixins所以不能包含状态，因为它有可能导致命名冲突）。 根据ECMAScript说明Traits和mixins遵循同样的原则，所以该规范没有定义“Traits”的概念。

**接口**

在一些OOP中实现的接口和mixins及traits类似。然而，与mixins及traits相比，接口强制实现类必须实现其方法签名的行为。

接口完全可以被视为抽象类。不过与抽象类相比（抽象类里的方法可以只实现一部分，另外一部分依然定义为签名），继承只能是单继承基类，但可以继承多个接口，节约这个原因，可以接口（多个混合）可以看做是多继承的替代方案。

ECMA-262-3标准既没有定义“接口”的概念，也没有定义“抽象类”的概念。 然而，作为模仿，它是可以由“空”的方法（或空方法中抛出异常，告诉开发人员这个方法需要被实现）的对象来实现。

**对象组合**

对象组合也是一个动态代码重用技术之一。 对象组合不同于高灵活性的继承，它实现了一个动态可变的委托。而这，也是基于委托原型的基本。 除了动态可变原型，该对象可以为委托聚合对象（创建一个组合作为结果——聚合 ），并进一步发送消息到对象上，委托到该委托上。这可以两个以上的委托，因为它的动态特性决定着它可以在运行时改变。

已经提到的\_\_noSuchMethod\_\_例子是这样，但也让我们展示了如何明确地使用委托：

例如：

var \_delegate = {  
 foo: function () {  
 alert('\_delegate.foo');  
 }  
};  
   
var agregate = {  
   
 delegate: \_delegate,  
   
 foo: function () {  
 return this.delegate.foo.call(this);  
 }  
   
};  
   
agregate.foo(); // delegate.foo  
   
agregate.delegate = {  
 foo: function () {  
 alert('foo from new delegate');  
 }  
};  
   
agregate.foo(); // foo from new delegate

这种对象关系称为“has-a”，而集成是“is-a“的关系。

由于显示组合的缺乏（与继承相比的灵活性），增加中间代码也是可以的。

**AOP特性**

作为面向方面的一个功能，可以使用function decorators。ECMA-262-3规格没有明确定义的“function decorators”的概念（和Python相对，这个词是在Python官方定义了）。 不过，拥有函数式参数的函数在某些方面是可以装饰和激活的（通过应用所谓的建议）：

最简单的装饰者例子：

function checkDecorator(originalFunction) {  
 return function () {  
 if (fooBar != 'test') {  
 alert('wrong parameter');  
 return false;  
 }  
 return originalFunction();  
 };  
}  
   
function test() {  
 alert('test function');  
}  
   
var testWithCheck = checkDecorator(test);  
var fooBar = false;  
   
test(); // 'test function'  
testWithCheck(); // 'wrong parameter'  
   
fooBar = 'test';  
test(); // 'test function'  
testWithCheck(); // 'test function'

**结论**

在这篇文章，我们理清了OOP的概论（我希望这些资料已经对你有用了）