[强大的原型和原型链](http://www.cnblogs.com/TomXu/archive/2012/01/05/2305453.html)

**前言**

JavaScript 不包含传统的类继承模型，而是使用 prototypal 原型模型。

虽然这经常被当作是 JavaScript 的缺点被提及，其实基于原型的继承模型比传统的类继承还要强大。实现传统的类继承模型是很简单，但是实现 JavaScript 中的原型继承则要困难的多。

由于 JavaScript 是唯一一个被广泛使用的基于原型继承的语言，所以理解两种继承模式的差异是需要一定时间的，今天我们就来了解一下原型和原型链。

**原型**

10年前，我刚学习JavaScript的时候，一般都是用如下方式来写代码：

var decimalDigits = 2,  
 tax = 5;  
  
 function add(x, y) {  
 return x + y;  
 }  
  
 function subtract(x, y) {  
 return x - y;  
 }  
  
 //alert(add(1, 3));

通过执行各个function来得到结果，学习了原型之后，我们可以使用如下方式来**美化**一下代码。

**原型使用方式1：**

在使用原型之前，我们需要先将代码做一下小修改：

var Calculator = function (decimalDigits, tax) {  
 this.decimalDigits = decimalDigits;  
 this.tax = tax;  
 };

然后，通过给Calculator对象的prototype属性赋值**对象字面量**来设定Calculator对象的原型。

Calculator.prototype = {  
 add: function (x, y) {  
 return x + y;  
 },  
  
 subtract: function (x, y) {  
 return x - y;  
 }  
 };  
 //alert((new Calculator()).add(1, 3));

这样，我们就可以new Calculator对象以后，就可以调用add方法来计算结果了。

**原型使用方式2：**

第二种方式是，在赋值原型prototype的时候使用function立即执行的表达式来赋值，即如下格式：

Calculator.prototype = function () { } ();

就是可以封装私有的function，通过return的形式暴露出简单的使用名称，以达到public/private的效果，修改后的代码如下：

Calculator.prototype = function () {  
 add = function (x, y) {  
 return x + y;  
 },  
  
 subtract = function (x, y) {  
 return x - y;  
 }  
 return {  
 add: add,  
 subtract: subtract  
 }  
 } ();  
  
 //alert((new Calculator()).add(11, 3));

同样的方式，我们可以new Calculator对象以后调用add方法来计算结果了。

**再来一点**

**分步声明：**

上述使用原型的时候，有一个限制就是一次性设置了原型对象，我们再来说一下如何分来设置原型的每个属性吧。

var BaseCalculator = function () {  
 //为每个实例都声明一个小数位数  
 this.decimalDigits = 2;  
};  
   
//使用原型给BaseCalculator扩展2个对象方法  
BaseCalculator.prototype.add = function (x, y) {  
 return x + y;  
};  
  
BaseCalculator.prototype.subtract = function (x, y) {  
 return x - y;  
};

首先，声明了一个BaseCalculator对象，构造函数里会初始化一个小数位数的属性decimalDigits，然后通过原型属性设置2个function，分别是add(x,y)和subtract(x,y)，当然你也可以使用前面提到的2种方式的任何一种，我们的主要目的是看如何将BaseCalculator对象设置到真正的Calculator的原型上。

var BaseCalculator = function() {  
 this.decimalDigits = 2;  
};  
  
BaseCalculator.prototype = {  
 add: function(x, y) {  
 return x + y;  
 },  
 subtract: function(x, y) {  
 return x - y;  
 }  
};

创建完上述代码以后，我们来开始：

var Calculator = function () {  
 //为每个实例都声明一个税收数字  
 this.tax = 5;  
};  
   
Calculator.prototype = new BaseCalculator();

我们可以看到Calculator的原型是指向到BaseCalculator的一个实例上，目的是让Calculator集成它的add(x,y)和subtract(x,y)这2个function，还有一点要说的是，由于它的原型是BaseCalculator的一个实例，所以不管你创建多少个Calculator对象实例，他们的原型指向的都是同一个实例。

var calc = new Calculator();  
alert(calc.add(1, 1));  
//BaseCalculator 里声明的decimalDigits属性，在 Calculator里是可以访问到的  
alert(calc.decimalDigits);

上面的代码，运行以后，我们可以看到因为Calculator的原型是指向BaseCalculator的实例上的，所以可以访问他的decimalDigits属性值，那如果我不想让Calculator访问BaseCalculator的构造函数里声明的属性值，那怎么办呢？这么办：

var Calculator = function () {  
 this.tax= 5;  
};  
  
Calculator.prototype = BaseCalculator.prototype;

通过将BaseCalculator的原型赋给Calculator的原型，这样你在Calculator的实例上就访问不到那个decimalDigits值了，如果你访问如下代码，那将会提升出错。

var calc = new Calculator();  
alert(calc.add(1, 1));  
alert(calc.decimalDigits);

**重写原型：**

在使用第三方JS类库的时候，往往有时候他们定义的原型方法是不能满足我们的需要，但是又离不开这个类库，所以这时候我们就需要重写他们的原型中的一个或者多个属性或function，我们可以通过继续声明的同样的add代码的形式来达到覆盖重写前面的add功能，代码如下：

//覆盖前面Calculator的add() function   
Calculator.prototype.add = function (x, y) {  
 return x + y + this.tax;  
};  
  
var calc = new Calculator();  
alert(calc.add(1, 1));

这样，我们计算得出的结果就比原来多出了一个tax的值，但是有一点需要注意：那就是重写的代码需要放在最后，这样才能覆盖前面的代码。

**原型链**

在将原型链之前，我们先上一段代码：

function Foo() {  
 this.value = 42;  
}  
Foo.prototype = {  
 method: function() {}  
};  
  
function Bar() {}  
  
// 设置Bar的prototype属性为Foo的实例对象  
Bar.prototype = new Foo();  
Bar.prototype.foo = 'Hello World';  
  
// 修正Bar.prototype.constructor为Bar本身  
Bar.prototype.constructor = Bar;  
  
var test = new Bar() // 创建Bar的一个新实例

var test1 = new Bar();  
  
// 原型链  
test [Bar的实例]  
 Bar.prototype [Foo的实例]   
 { foo: 'Hello World' }  
 Foo.prototype  
 {method: ...};  
 Object.prototype  
 {toString: ... /\* etc. \*/};

上面的例子中，test 对象从 Bar.prototype 和 Foo.prototype 继承下来；因此，它能访问 Foo 的原型方法 method。同时，它也能够访问那个定义在原型上的 Foo 实例属性value。需要注意的是 new Bar() 不会创造出一个新的 Foo 实例，而是重复使用它原型上的那个实例；因此，所有的 Bar 实例都会共享相同的 value 属性。

**属性查找：**

当查找一个对象的属性时，JavaScript 会向上遍历原型链，直到找到给定名称的属性为止，到查找到达原型链的顶部 - 也就是 Object.prototype - 但是仍然没有找到指定的属性，就会返回 undefined，我们来看一个例子：

function foo() {  
 this.add = function (x, y) {  
 return x + y;  
 }  
 }  
  
 foo.prototype.add = function (x, y) {  
 return x + y + 10;  
 }  
  
 Object.prototype.subtract = function (x, y) {  
 return x - y;  
 }  
  
 var f = new foo();  
 alert(f.add(1, 2)); //结果是3，而不是13  
 alert(f.subtract(1, 2)); //结果是-1

通过代码运行，我们发现subtract是安装我们所说的向上查找来得到结果的，但是add方式有点小不同，这也是我想强调的，就是属性在查找的时候是先查找自身的属性，如果没有再查找原型，再没有，再往上走，一直插到Object的原型上，所以在某种层面上说，用 for in语句遍历属性的时候，效率也是个问题。

还有一点我们需要注意的是，我们可以赋值任何类型的对象到原型上，但是不能赋值原子类型的值，比如如下代码是无效的：

function Foo() {}  
Foo.prototype = 1; // 无效

**hasOwnProperty函数：**

hasOwnProperty是Object.prototype的一个方法，它可是个好东西，他能判断一个对象是否包含自定义属性而不是原型链上的属性，因为hasOwnProperty 是 JavaScript 中唯一一个处理属性但是不查找原型链的函数。

// 修改Object.prototype  
Object.prototype.bar = 1;   
var foo = {goo: undefined};  
  
foo.bar; // 1  
'bar' in foo; // true  
  
foo.hasOwnProperty('bar'); // false  
foo.hasOwnProperty('goo'); // true

只有 hasOwnProperty 可以给出正确和期望的结果，这在遍历对象的属性时会很有用。 没有其它方法可以用来排除原型链上的属性，而不是定义在对象自身上的属性。

但有个恶心的地方是：JavaScript 不会保护 hasOwnProperty 被非法占用，因此如果一个对象碰巧存在这个属性，就需要使用外部的 hasOwnProperty 函数来获取正确的结果。

var foo = {  
 hasOwnProperty: function() {  
 return false;  
 },  
 bar: 'Here be dragons'  
};  
  
foo.hasOwnProperty('bar'); // 总是返回 false  
  
// 使用{}对象的 hasOwnProperty，并将其上下为设置为foo  
{}.hasOwnProperty.call(foo, 'bar'); // true

当检查对象上某个属性是否存在时，hasOwnProperty 是唯一可用的方法。同时在使用 for in loop 遍历对象时，推荐总是使用 hasOwnProperty 方法，这将会避免原型对象扩展带来的干扰，我们来看一下例子：

// 修改 Object.prototype  
Object.prototype.bar = 1;  
  
var foo = {moo: 2};  
for(var i in foo) {  
 console.log(i); // 输出两个属性：bar 和 moo  
}

我们没办法改变for in语句的行为，所以想过滤结果就只能使用hasOwnProperty 方法，代码如下：

// foo 变量是上例中的  
for(var i in foo) {  
 if (foo.hasOwnProperty(i)) {  
 console.log(i);  
 }  
}

这个版本的代码是唯一正确的写法。由于我们使用了 hasOwnProperty，所以这次只输出 moo。如果不使用 hasOwnProperty，则这段代码在原生对象原型（比如 Object.prototype）被扩展时可能会出错。

总结：推荐使用 hasOwnProperty，不要对代码运行的环境做任何假设，不要假设原生对象是否已经被扩展了。

**总结**

原型极大地丰富了我们的开发代码，但是在平时使用的过程中一定要注意上述提到的一些注意事项。

参考内容：http://bonsaiden.github.com/JavaScript-Garden/zh/