**不是原型继承那么简单！！prototype的深度探索**

1 什么是prototype  
  
       JavaScript中对象的prototype属性，可以返回对象类型原型的引用。这是一个相当拗口的解释，要理解它，先要正确理解对象类型(Type)以及原型(prototype)的概念。  
        前面我们说，对象的类（Class）和对象实例（Instance）之间是一种“创建”关系，因此我们把“类”看作是对象特征的模型化，而对象看作是类特征的具体化，或者说，类(Class)是对象的一个类型(Type)。例如，在前面的例子中，p1和p2的类型都是Point，在JavaScript中，通过instanceof运算符可以验证这一点：  
        p1 instanceof Point  
        p2 instanceof Point  
  
        但是，Point不是p1和p2的唯一类型，因为p1和p2都是对象，所以Obejct也是它们的类型，因为Object是比Point更加泛化的类，所以我们说，Obejct和Point之间有一种衍生关系，在后面我们会知道，这种关系被叫做“继承”，它也是对象之间泛化关系的一个特例，是面向对象中不可缺少的一种基本关系。  
        在面向对象领域里，实例与类型不是唯一的一对可描述的抽象关系，在JavaScript中，另外一种重要的抽象关系是类型(Type)与原型(prototype)。这种关系是一种更高层次的抽象关系，它恰好和类型与实例的抽象关系构成了一个三层的链，下图描述了这种关系：  
        //TODO:  
  
        在现实生活中，我们常常说，某个东西是以另一个东西为原型创作的。这两个东西可以是同一个类型，也可以是不同类型。习语“依葫芦画瓢”，这里的葫芦就是原型，而瓢就是类型，用JavaScript的prototype来表示就是“瓢.prototype =某个葫芦”或者“瓢.prototype= new 葫芦()”。  
        要深入理解原型，可以研究关于它的一种设计模式——prototype pattern，这种模式的核心是用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。JavaScript的prototype就类似于这种方式。  
  
        关于prototype pattern的详细内容可以参考《设计模式》（《Design Patterns》）它不是本文讨论的范围。  
  
        注意，同类型与实例的关系不同的是，原型与类型的关系要求一个类型在一个时刻只能有一个原型（而一个实例在一个时刻显然可以有多个类型）。对于JavaScript来说，这个限制有两层含义，第一是每个具体的JavaScript类型有且仅有一个原型（prototype），在默认的情况下，这个原型是一个Object对象（注意不是Object类型！）。第二是，这个对象所属的类型，必须是满足原型关系的类型链。例如p1所属的类型是Point和Object，而一个Object对象是Point的原型。假如有一个对象，它所属的类型分别为ClassA、ClassB、ClassC和Object，那么必须满足这四个类构成某种完整的原型链，例如：  
        //TODO:  
          
  
        下面这个图描述了JavaScript中对象、类型和原型三者的关系：  
        //TODO:  
  
        有意思的是，JavaScript并没有规定一个类型的原型的类型（这又是一段非常拗口的话），因此它可以是任何类型，通常是某种对象，这样，对象-类型-原形（对象）就可能构成一个环状结构，或者其它有意思的拓扑结构，这些结构为JavaScript带来了五花八门的用法，其中的一些用法不但巧妙而且充满美感。下面的一节主要介绍prototype的用法。  
  
  
  
2 prototype使用技巧  
  
      在了解prototype的使用技巧之前，首要先弄明白prototype的特性。首先，JavaScript为每一个类型(Type)都提供了一个prototype属性，将这个属性指向一个对象，这个对象就成为了这个类型的“原型”，这意味着由这个类型所创建的所有对象都具有这个原型的特性。另外，JavaScript的对象是动态的，原型也不例外，给prototype增加或者减少属性，将改变这个类型的原型，这种改变将直接作用到由这个原型创建的所有对象上，例如：  
  
   提示：您可以先修改部分代码再运行  
  
如果给某个对象的类型的原型添加了某个名为a的属性，而这个对象本身又有一个名为a的同名属性，则在访问这个对象的属性a时，对象本身的属性“覆盖”了原型属性，但是原型属性并没有消失，当你用delete运算符将对象本身的属性a删除时，对象的原型属性就恢复了可见性。利用这个特性，可以为对象的属性设定默认值，例如：  
  
   提示：您可以先修改部分代码再运行  
  
上面的例子通过prototype为Point对象设定了默认值(0,0)，因此p1的值为(0,0)，p2的值为(1,2)，通过delete p2.x, delete p2.y; 可以将p2的值恢复为(0,0)。下面是一个更有意思的例子：  
  
   提示：您可以先修改部分代码再运行  
  
利用prototype还可以为对象的属性设置一个只读的getter，从而避免它被改写。下面是一个例子：  
  
   提示：您可以先修改部分代码再运行  
  
将this.getFirstPoint()改写为下面这个样子：  
this.getFirstPoint = function()  
{  
        function GETTER(){};  
        GETTER.prototype = m\_firstPoint;  
        return new GETTER();  
}  
则可以避免这个问题，保证了m\_firstPoint属性的只读性。  
  
   提示：您可以先修改部分代码再运行  
  
实际上，将一个对象设置为一个类型的原型，相当于通过实例化这个类型，为对象建立只读副本，在任何时候对副本进行改变，都不会影响到原始对象，而对原始对象进行改变，则会影响到副本，除非被改变的属性已经被副本自己的同名属性覆盖。用delete操作将对象自己的同名属性删除，则可以恢复原型属性的可见性。下面再举一个例子：  
  
   提示：您可以先修改部分代码再运行  
  
注意，以上的例子说明了用prototype可以快速创建对象的多个副本，一般情况下，利用prototype来大量的创建复杂对象，要比用其他任何方法来copy对象快得多。注意到，用一个对象为原型，来创建大量的新对象，这正是prototype pattern的本质。  
下面是一个例子：  
  
   提示：您可以先修改部分代码再运行  
  
除了上面所说的这些使用技巧之外，prototype因为它独特的特性，还有其它一些用途，被用作最广泛和最广为人知的可能是用它来模拟继承，关于这一点，留待下一节中去讨论。  
  
3 prototype的实质  
  
        上面已经说了prototype的作用，现在我们来透过规律揭示prototype的实质。  
        我们说，prototype的行为类似于C++中的静态域，将一个属性添加为prototype的属性，这个属性将被该类型创建的所有实例所共享，但是这种共享是只读的。在任何一个实例中只能够用自己的同名属性覆盖这个属性，而不能够改变它。换句话说，对象在读取某个属性时，总是先检查自身域的属性表，如果有这个属性，则会返回这个属性，否则就去读取prototype域，返回protoype域上的属性。另外，JavaScript允许protoype域引用任何类型的对象，因此，如果对protoype域的读取依然没有找到这个属性，则JavaScript将递归地查找prototype域所指向对象的prototype域，直到这个对象的prototype域为它本身或者出现循环为止，我们可以用下面的图来描述prototype与对象实例之间的关系：  
        //TODO:  
  
4 prototype的价值与局限性  
  
        从上面的分析我们理解了prototype，通过它能够以一个对象为原型，安全地创建大量的实例，这就是prototype的真正含义，也是它的价值所在。后面我们会看到，利用prototype的这个特性，可以用来模拟对象的继承，但是要知道，prototype用来模拟继承尽管也是它的一个重要价值，但是绝对不是它的核心，换句话说，JavaScript之所以支持prototype，绝对不是仅仅用来实现它的对象继承，即使没有了prototype继承，JavaScript的prototype机制依然是非常有用的。  
        由于prototype仅仅是以对象为原型给类型构建副本，因此它也具有很大的局限性。首先，它在类型的prototype域上并不是表现为一种值拷贝，而是一种引用拷贝，这带来了“副作用”。改变某个原型上引用类型的属性的属性值（又是一个相当拗口的解释:P），将会彻底影响到这个类型创建的每一个实例。有的时候这正是我们需要的（比如某一类所有对象的改变默认值），但有的时候这也是我们所不希望的（比如在类继承的时候），下面给出了一个例子：  
  
   提示：您可以先修改部分代码再运行