**原型的含义**是指：如果构造器有个原型对象A，则由该构造器创建的实例(Object Instance)都必然复制于A。““在JavaScript中，对象实例(Object Instance)并没有原型，而构造器(Constructor)有原型，属性’<构造器>.prototype’指向原型。对象只有“构 造自某个原型”的问题，并不存在“持有（或拥有）某个原型”的问题。””如何理解这一句话？

代码1：

|  |  |
| --- | --- |
| 01 | function myFunc() { |
| 02 | var name = "stephenchan"; | |

|  |  |
| --- | --- |
| 03 | var age = 23; |
| 04 | function code() { | |

|  |  |  |
| --- | --- | --- |
| 05 | alert("Hello World!"); | |
| 06 | }; |

|  |  |
| --- | --- |
| 07 | } |
| 08 | var obj = new myFunc(); | |

|  |  |  |
| --- | --- | --- |
| 09 | //输出undefined，对象实例没有原型 | |
| 10 | alert(obj.prototype); |

|  |  |  |
| --- | --- | --- |
| 11 | //输出myFunc的函数代码，obj由myFunc构造出来的 | |
| 12 | alert(obj.constructor); |

|  |  |
| --- | --- |
| 13 | //输出true |
| 14 | alert(obj.constructor == myFunc); | |

|  |  |
| --- | --- |
| 15 |  |
| 16 | //输出[object Object]，说明myFunc的原型是一个对象 | |

|  |  |
| --- | --- |
| 17 | alert(myFunc.prototype); |
| 18 | //输出function Function() { [native code] }，[native code]的意思是JavaScript引擎的内置函数 | |

|  |  |
| --- | --- |
| 19 | alert(myFunc.constructor); |
| 20 | //输出true，函数原型的构造器默认是该函数 | |

|  |  |
| --- | --- |
| 21 | alert(myFunc.prototype.constructor == myFunc); |

构造器与函数的概念是一致的，即代码1中，**myFunc就是一个构造器**，因为通过new myFunc()就可以构造出一个对象实例了。因此，”alert(obj.prototype)”输出undefined说明了**对象实例是没有原型的**，”alert(myFunc.prototype)”输出[object Object]说明了**构造器有原型**，而“obj.constructor==myFunc”返回true说明obj的构造器是myFunc。  
  
原型其实也是一个对象实例。再强调一下**原型的含义**是：如果构造器有个原型对象A，则由该构造器创建的实例 (Object Instance)都必然复制于A，而且采用的读遍历机制复制的。读遍历复制的意思是：仅当写某个实例的成员时，将成员的信息复制到实例映像中。即当构造 一个新的对象时，新对象里面的属性指向的是原型中的属性，读取对象实例的属性时，获取的是原型对象的属性值。而当对象实例对一个属性进行写操作时，才会将 属性写到新对象实例的属性列表中。

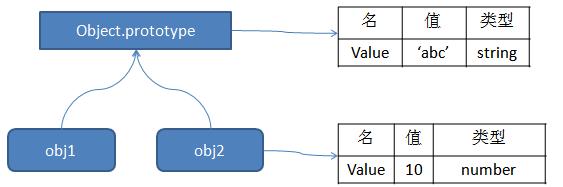


图1 JavaScript使用读遍历机制实现的原型继承

代码2：

|  |  |  |
| --- | --- | --- |
| 01 | Object.prototype.value = "abc"; | |
| 02 | var obj1 = new Object(); |

|  |  |  |
| --- | --- | --- |
| 03 | var obj2 = new Object(); | |
| 04 | obj2.value = 10; |

|  |  |  |
| --- | --- | --- |
| 05 | //输出abc，读取的是原型Object中的value | |
| 06 | alert(obj1.value); |

|  |  |  |
| --- | --- | --- |
| 07 | //输出10，读取的是obj2成员列表中的value | |
| 08 | alert(obj2.value); |

|  |  |  |
| --- | --- | --- |
| 09 | //删除obj2中的value，即在obj2的成员列表中将value删除掉 | |
| 10 | delete obj2.value; |

|  |  |  |
| --- | --- | --- |
| 11 | //输出abc，读取的是原型Object中的value | |
| 12 | alert(obj2.value); |

图1是对代码2的描述，说明读遍历机制是如何在成员列表以至原型中管理对象成员的。只有对属性进行第一次写操作的时候，才会在对象的成员列表中添加 该属性的记录。当obj1和obj2通过new来构造出来的时候，仍然是一个指向原型的引用，在操作过程中也没有与原型相同大小的对象实例创建出来。这样 的读遍历就避免在创建新对象实例时可能的大量内存分配。当obj2.value属性被赋值为10的时候，obj2则在其成员表中添加了一个value成 员，并赋值为10，这个成员表就是记录了obj2中发生了修改的成员名、值与类型。这张表是否与原型一致并不重要，只需要遵循两条规则：(1)保证在读取 时被首先访问到。(2)如果在对象中没有指定的属性，则尝试遍历对象的整个原型链，直到原型为空或找到该属性。代码2中的delete操作是将obj2成 员表中的value删除了，因此在读取obj2的value属性的时候就遍历到Object中读取。

函数的原型总是一个标准的、系统内置的Object()构造器的实例，不过该实例创建后constructor属性总先被赋值为当前的函数。

代码3：

|  |  |  |
| --- | --- | --- |
| 01 | function MyObject() { | |
| 02 | } |

|  |  |
| --- | --- |
| 03 |  |
| 04 | //显示true，表明原型的构造器总是指向函数自身的 | |

|  |  |  |
| --- | --- | --- |
| 05 | alert(MyObject.prototype.constructor == MyObject); | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | //删除该成员 |
| 08 | delete MyObject.prototype.constructor; | |

|  |  |
| --- | --- |
| 09 |  |
| 10 | //删除操作使该成员指向了父代类原型中的值 | |

|  |  |
| --- | --- |
| 11 | //均显示为true |
| 12 | alert(MyObject.prototype.constructor == Object); | |

|  |  |
| --- | --- |
| 13 | alert(MyObject.prototype.constructor == new Object().constructor); |

从代码3中可以看出，MyObject.prototype其实与一个普通对象”new Object()”并没有本质的区别，只是在创建时将constructor赋值为当前函数MyObject。然后，当一个函数的prototype有意 义之后，它就摇身一变成了一个“构造器”，这时，如果用户试图用new运算符创建它的实例时，那么引擎就会再构造一个新的对象，并使这个新对象的原型链接 向这个prototype属性就可以了。因此，**函数与构造器并没有明显的界限**。

一个构造器产生的实例，其**constructor属性默认总是指向该构造器**，而究其根源，则在于**构造器（函数）的原型的constructor属性**指向了构造器本身。  
代码4：

|  |  |  |
| --- | --- | --- |
| 1 | function MyObject() { | |
| 2 | } |

|  |  |
| --- | --- |
| 3 | var obj = new MyObject(); |
| 4 | //输出为true，默认指向构造器 | |

|  |  |
| --- | --- |
| 5 | alert(obj.constructor == MyObject); |
| 6 | //输出为true，原型的构造器指向该构造器 | |

|  |  |
| --- | --- |
| 7 | alert(MyObject.prototype.constructor == MyObject); |

由此可见，JavaScript事实上已经为构造器维护了原型属性，因此我们可以通过实例的constructor属性来找到构造器，并进而找到它 的原型“obj.constructor.prototype”。但是，如果我们把构造器的原型修改了的话，会出现什么情况呢？如代码5，我们把 MyObjectEx的原型修改了。  
代码5：

|  |  |  |
| --- | --- | --- |
| 1 | function MyObject() { | |
| 2 | } |

|  |  |  |
| --- | --- | --- |
| 3 | function MyObjectEx() { | |
| 4 | } |

|  |  |  |
| --- | --- | --- |
| 5 | MyObjectEx.prototype = new MyObject(); | |
| 6 | var obj1 = new MyObject(); |

|  |  |
| --- | --- |
| 7 | var obj2 = new MyObjectEx(); |
| 8 | alert(obj1.constructor == obj2.constructor);    //true | |

|  |  |
| --- | --- |
| 9 | alert(MyObjectEx.prototype.constructor == MyObject.prototype.constructor);    //true |

在代码5中，obj1和obj2是由不同的两个构造器产生的实例，分别是MyObject和MyObjectEx。然而，我们看到，代码5中的两个alert都会输出true，即是说，**由两个不相同的构造器产生的实例（代码5中的MyObject和MyObjectEx），它们的constructor属性却指向了相同的构造器**， 是不是很诡异？这个正确是体现了原型继承中出出现的“原型复制”了。要注意，MyObjectEx的原型是由MyObject构造出来的对象实例，即 obj1和obj2都是从MyObject原型中复制出来的对象，因此它们的constructor指向的都是MyObject。那么怎么解决这个问题？  
代码6：

|  |  |
| --- | --- |
| 01 | function MyObject() { |
| 02 | this.constructor = arguments.callee; //arguments.callee为MyObject，正确维护constructor，以便回溯外部原型链 | |

|  |  |
| --- | --- |
| 03 | } |
| 04 | MyObject.prototype = new Object(); //人为构建外部原型链 | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | function MyObjectEx() { | |

|  |  |  |
| --- | --- | --- |
| 07 | this.constructor = arguments.callee; //正确维护constructor，以便回溯外部原型链 | |
| 08 | } |

|  |  |  |
| --- | --- | --- |
| 09 | MyObjectEx.prototype = new MyObject(); //人为构建外部原型链 | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | obj1 = new MyObjectEx(); |
| 12 | obj2 = new MyObjectEx(); |

代码6与代码5中的主要区别就是在于，在MyObjectEx的初始化中正确地维护了constructor属性，使当前的constructor属性指向了调用的构造器。代码6所描述的继承关系如图2：

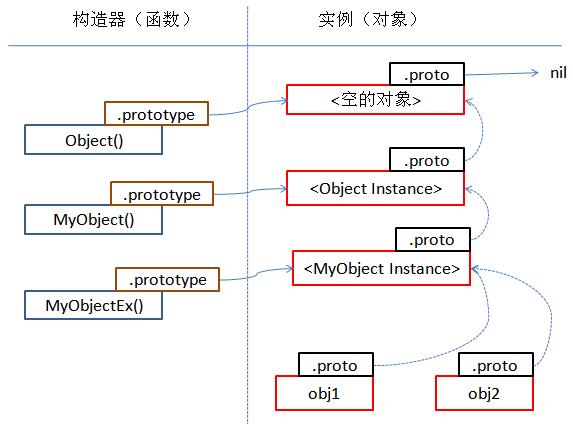


图2 构造器原型链与内部原型链

其中有[proto]属性中一个对象的私有属性，用于正确维护对象的内部原型链，在Firefox中可以通过[\_\_proto\_\_]来访问，这个后 面再讨论。我们可以看到MyObjectEx的构造器是MyObject的对象实例，而MyObject的构造器是Object的对象实例。  
接代码6：

|  |  |
| --- | --- |
| 1 | obj = new MyObject(); |
| 2 | alert(obj.constructor === MyObject);    //true | |

|  |  |
| --- | --- |
| 3 | alert(obj1.constructor === MyObjectEx);    //true |
| 4 | alert(obj.constructor === obj1.constructor);    //false | |

可以看到，obj和obj1从不同的构造器产生的实例，其constructor属性已经能够正确地指向相应的构造器，这个是由于在对象实例初始化 的时候的赋值语句”this.constructor = arguments.callee;”。你可能会疑问为什么不采用下面这种方式来实现：

|  |  |
| --- | --- |
| 1 | MyObjectEx.prototype = new MyObject(); |
| 2 | MyObjectEx.prototype.constructor = MyObjectEx; | |

这样虽然能使obj1和obj2的constructor属性正确地指向了MyObjectEx，但是，这样同时也使得MyObjectEx的原型 对象（MyObject构造的实例）的constructor属性没法往父代原型追溯。因为当MyObjectEx的原型对象想通过 constructor属性来获取到MyObject构造器时，会发现获取到的是MyObjectEx的构造器，而不是期待的MyObject的构造器。  
我们可以通过下面的语句来验证代码6是不是的确是如图2的关系链：

|  |  |
| --- | --- |
| 1 | alert(obj1.constructor === MyObjectEx); //true |
| 2 | alert(MyObjectEx.prototype instanceof MyObject); //true | |

|  |  |  |
| --- | --- | --- |
| 3 | alert(MyObjectEx.prototype.constructor === MyObject); //true | |
| 4 | alert(MyObject.prototype instanceof Object); //true |

|  |  |
| --- | --- |
| 5 | alert(MyObject.prototype.constructor === Object); //true |
| 6 | alert(obj1.constructor.prototype.constructor.prototype.constructor === Object); //true，完成了所有的回溯 | |

好了，刚才上面提到了有一个不可访问的属性[proto]，这个属性是JavaScript引擎内部维护的原型链属性，这个属性在Firefox里 面可以通过[\_\_proto\_\_]来访问的，一般情况下，[proto]属性指向的和prototype属性一样，指向的都是原型对象，两个有什么不同后 面会有讲述。

|  |  |
| --- | --- |
| 1 | //输出都是true，在Firefox中 |
| 2 | alert(obj.\_\_proto\_\_ instanceof Object); | |

|  |  |
| --- | --- |
| 3 | alert(obj1.\_\_proto\_\_ instanceof MyObject); |
| 4 | alert(obj2.\_\_proto\_\_ instanceof MyObject); |

这个[proto]属性是JavaScript内部维护的，外部是不可访问的，由这个属性所维护的原型链为**内部原型链**，与由prototype和constructor维护的**外部原型链**。那么这两条原型链有什么区别呢？简单来说就是，**通过prototype和constructor来维护的外部原型链是开发人员自己代码中回溯时用到的，而通过[proto]维护的内部原型链是JavaScript原型继承机制实现所需要的**。 具体来说，外部原型链就是做这种 事：”alert(obj1.constructor.prototype.constructor.prototype.constructor === Object);”，也就是说当我们开发人员想要自己去回溯整个原型继承的结构链时，也只会在我们开发人员写代码时才出现通过prototype和 constructor来访问外部原型链。而内部原型链，这个比较有意思，在［图1 JavaScript使用读遍历机制实现的原型继承］，我们看到，当我们访问一个对象实例的属性时，它如果发现在其成员列表中没有该属性，即会去访问原型 的成员列表，把原型的默认值读取出来，也就是说，这个在原型链中回溯来查询成员属性的过程，只会在内部原型链中进行，这个过程是由JavaScript引 擎自己去维护的，开发人员没法干涉。来看看代码，我觉得这个还是相当有意思的：  
接代码6：

|  |  |
| --- | --- |
| 01 | alert(obj.\_\_proto\_\_ instanceof Object);    //true |
| 02 | alert(obj1.\_\_proto\_\_ instanceof MyObject);    //true | |

|  |  |
| --- | --- |
| 03 | alert(obj2.\_\_proto\_\_ instanceof MyObject);    //true |
| 04 | //按照上面所说的，在MyObjectEx的原型上添加了value的属性，那么在访问obj1和obj2的value属性时便会往原型中查找 | |

|  |  |  |
| --- | --- | --- |
| 05 | MyObjectEx.prototype.value = "Hello World!"; | |
| 06 | //这里正确地输出"Hello World!" |

|  |  |
| --- | --- |
| 07 | alert(obj1.value); |
| 08 | //在此时，obj1和obj2都构造之后，我把原来的MyObjectEx的原型换了，变成MyObjectEx2 | |

|  |  |
| --- | --- |
| 09 | function MyObjectEx2() {} |
| 10 | MyObjectEx.prototype = new MyObjectEx2(); | |

|  |  |  |
| --- | --- | --- |
| 11 | //这句究竟会输出什么呢？[Referece Error]还是？ | |
| 12 | alert(obj1.value); |

最后的1个alert输出的”Hello World!”，有意思吧。即使我在上面把MyObjectEx的原型对象改变成新的MyObjectEx2，但是在obj1和obj2中的 [proto]属性依然指向的是原来的MyObject构造的对象实例，也就是说内部访问属性时是通过[proto]来回溯原型链的，而不是通过 prototype的（而且对象实例也没有prototype属性），这个就是内部原型链体现的威力。