**原型对象**

　　每个javascript对象都有一个原型对象，这个对象在不同的解释器下的实现不同。比如在firefox下，每个对象都有一个隐藏的\_\_proto\_\_属性，这个属性就是“原型对象”的引用。

**原型链**

　　由于原型对象本身也是对象，根据上边的定义，它也有自己的原型，而它自己的原型对象又可以有自己的原型，这样就组成了一条链，这个就是原型链，JavaScritp引擎在访问对象的属性时，如果在对象本身中没有找到，则会去原型链中查找，如果找到，直接返回值，如果整个链都遍历且没有找到属性，则返回undefined.原型链一般实现为一个链表，这样就可以按照一定的顺序来查找。

示例1

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

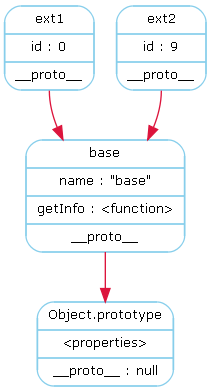
|  |
| --- |
| var base = {      name : "base",      getInfo : function(){         returnthis.name;      }  }    var ext1 = {      id : 0,      \_\_proto\_\_ : base  }    varext2 = {      id : 9,      \_\_proto\_\_ : base  }    print(ext1.id);  print(ext1.getInfo());  print(ext2.id);  print(ext2.getInfo()); |

 结果

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| 0  base  9  base |

 图1



　　可以看到，当执行ext1.id时，引擎在ext1对象本身中就找到了id属性，因此返回其值0，当执行ext1.getInfo时，ext1对象中没有找到，因此在其原型对象base中查找，找到之后，执行这个函数，得到输出”base”。

我们将上例中的ext1对象稍加修改，为ext1对象加上name属性：

示例2

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| varbase = {      name : "base",      getInfo : function(){         returnthis.name;      }  }    varext1 = {      id : 0,      name : "ext1",      \_\_proto\_\_ : base  }    print(ext1.id);  print(ext1.getInfo()); |

 结果

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| 0  ext1 |

　　这个运行效果同样验证了原型链的运行机制：从对象本身出发，沿着\_\_proto\_\_查找，直到找到属性名称相同的值(没有找到，则返回undefined)。

我们对上例再做一点修改，来更好的演示原型链的工作方式：

示例3

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| varbase = {      name : "base",      getInfo : function(){         returnthis.id + ":"+ this.name;      }  }    varext1 = {      id : 0,      \_\_proto\_\_ : base  }    print(ext1.getInfo()); |

 结果

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| 0:base |

 　　应该注意的是，getInfo函数中的this表示原始的对象，而并非原型对象。上例中的id属性来自于ext1对象，而name来自于base对象。这个特性的机制在10.3小节再做讨论。如果对象没有显式的声明自己的”\_\_proto\_\_”属性，这个值默认的设置为Object.prototype,而Object.prototype的”\_\_proto\_\_”属性的值为”null”，标志着原型链的终结。

**构造器**

　　我们在来讨论一下构造器，除了上边提到的直接操作对象的\_\_proto\_\_属性的指向以外，JavaScript还支持构造器形式的对象创建。构造器会自动的为新创建的对象设置原型对象，此时的原型对象通过构造器的prototype属性来引用。

我们以例子来说明，将Task函数作为构造器，然后创建两个实例task1, task2：

示例4

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| function Task(id){      this.id = id;  }    Task.prototype.status = "STOPPED";  Task.prototype.execute = function(args){      return"execute task\_"+this.id+"["+this.status+"]:"+args;  }    var task1 = new Task(1);  var task2 = new Task(2);    task1.status = "ACTIVE";  task2.status = "STARTING";    print(task1.execute("task1"));  print(task2.execute("task2")); |

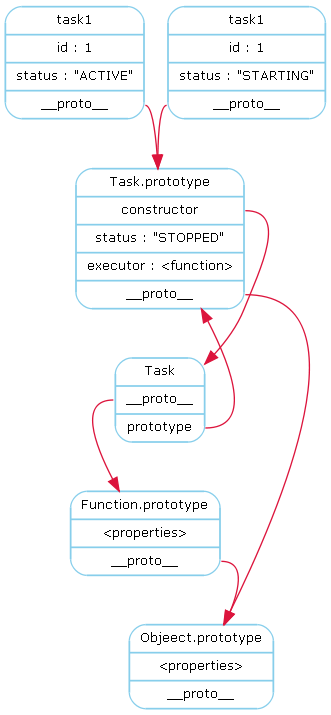
 结果

[?](http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html)

|  |
| --- |
| execute task\_1[ACTIVE]:task1  execute task\_2[STARTING]:task2 |

 　　构造器会自动为task1,task2两个对象设置原型对象Task.prototype，这个对象被Task(在此最为构造器)的prototype属性引用，参看下图中的箭头指向。

图2



　　由于Task本身仍旧是函数，因此其”\_\_proto\_\_”属性为Function.prototype, 而内建的函数原型对象的”\_\_proto\_\_”属性则为Object.prototype对象。最后Obejct.prototype的”\_\_proto\_\_”值为null。

资源链接：<http://www.cnblogs.com/xqhppt/archive/2012/02/01/2334355.html>