## Js面向对象学习笔记

## 第一章：原型与原型连

**一. 普通对象与函数对象**

JavaScript 中，万物皆对象！但对象也是有区别的。分为普通对象和函数对象，Object ，Function 是JS自带的函数对象。下面举例说明

function f1(){};

var f2 = function(){};

var f3 = new Function('str','console.log(str)');

var o3 = new f1();

var o1 = {};

var o2 =new Object();

console.log(typeof Object); //function

console.log(typeof Function); //function

console.log(typeof o1); //object

console.log(typeof o2); //object

console.log(typeof o3); //object

console.log(typeof f1); //function

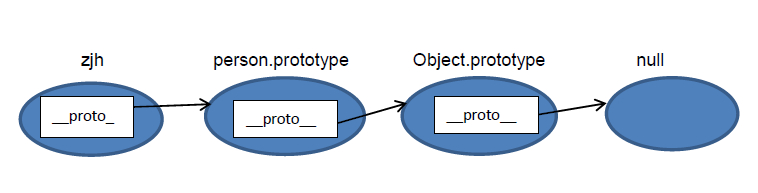
console.log(typeof f2); //function

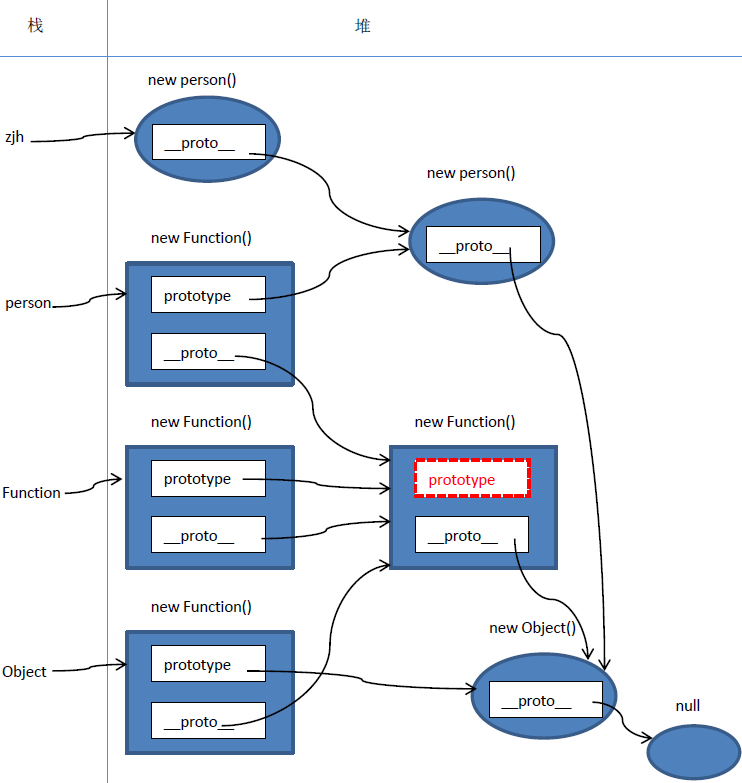
console.log(typeof f3); //function

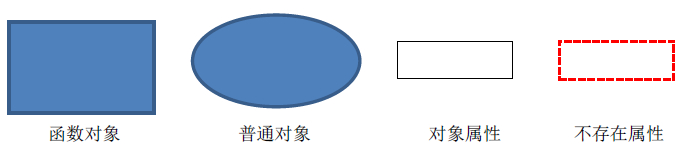
在上面的例子中 o1 o2 o3 为普通对象，f1 f2 f3 为函数对象。怎么区分，其实很简单，凡是通过 new Function() 创建的对象都是函数对象，其他的都是普通对象。f1,f2,归根结底都是通过 new Function()的方式进行创建的。Function Object 也都是通过 New Function()创建的。

1. **原型对象**  
      在JavaScript 中，每当定义一个对象（函数）时候，对象中都会包含一些预定义的属性。其中函数对象的一个属性就是原型对象 prototype。注：普通对象没有prototype,但有\_\_proto\_\_属性。  
     
     原型对象其实就是普通对象（Function.prototype除外,它是函数对象，但它很特殊，他没有prototype属性（前面说道函数对象都有prototype属性））。看下面的例子：  
    function f1(){};  
    console.log(f1.prototype) //f1{}  
    console.log(typeof f1. prototype) //Object  
    console.log(typeof Function.prototype) // Function，这个特殊  
    console.log(typeof Object.prototype) // Object  
    console.log(typeof Function.prototype.prototype) //undefined  
     
    从这句console.log(f1.prototype) //f1 {} 的输出就结果可以看出，f1.prototype就是f1的一个实例对象。就是在f1创建的时候,创建了一个它的实例对象并赋值给它的prototype，基本过程如下：  
    var temp = new f1();  
    f1. prototype = temp;  
     
     所以，Function.prototype为什么是函数对象就迎刃而解了，上文提到凡是new Function ()产生的对象都是函数对象，所以temp1是函数对象。  
    var temp1 = new Function ();  
    Function.prototype = temp1;  
     
   那原型对象是用来做什么的呢？主要作用是用于继承。举了例子：  
     var person = function(name){  
      this.name = name  
     };  
     person.prototype.getName = function(){  
        return this.name;   
     }  
     var zjh = new person(‘zhangjiahao’);  
     zjh.getName(); //zhangjiahao  
     
      从这个例子可以看出，通过给person.prototype设置了一个函数对象的属性，那有person实例（例中：zjh）出来的普通对象就继承了这个属性。具体是怎么实现的继承，就要讲到下面的原型链了。

**三．原型链**  
   JS在创建对象（不论是普通对象还是函数对象）的时候，都有一个叫做\_\_proto\_\_的内置属性，用于指向创建它的函数对象的原型对象prototype。以上面的例子为例：  
  
  console.log(zjh.\_\_proto\_\_ === person.prototype) //true  
  
同样，person.prototype对象也有\_\_proto\_\_属性，它指向创建它的函数对象（Object）的prototype  
  
  console.log(person.prototype.\_\_proto\_\_ === Object.prototype) //true  
  
继续，Object.prototype对象也有\_\_proto\_\_属性，但它比较特殊，为null  
  
  console.log(Object.prototype.\_\_proto\_\_) //null  
  
我们把这个有\_\_proto\_\_串起来的直到Object.prototype.\_\_proto\_\_为null的链叫做原型链。如下图：

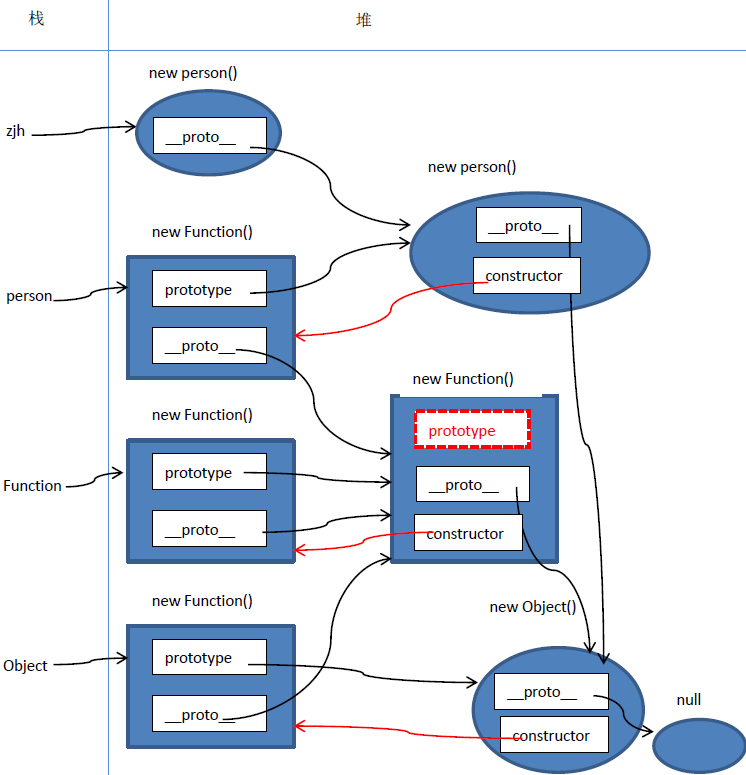


**四．内存结构图**  
为了更加深入和直观的进行理解，下面我们画一下上面的内存结构图：  


画图约定：  
  
  
疑点解释：  
1.Object.\_\_proto\_\_ === Function.prototype // true  
  Object是函数对象，是通过new Function()创建，所以Object.\_\_proto\_\_指向Function.prototype。  
  
2.Function.\_\_proto\_\_ === Function.prototype // true  
  Function 也是对象函数，也是通过new Function()创建，所以Function.\_\_proto\_\_指向Function.prototype。  
  
自己是由自己创建的，好像不符合逻辑，但仔细想想，现实世界也有些类似，你是怎么来的，你妈生的，你妈怎么来的，你姥姥生的，……类人猿进化来的，那类人猿从哪来，一直追溯下去……，就是无，（NULL生万物）  
正如《道德经》里所说“无，名天地之始”。  
  
3.Function.prototype.\_\_proto\_\_ === Object.prototype //true  
其实这一点我也有点困惑，不过也可以试着解释一下。  
Function.prototype是个函数对象，理论上他的\_\_proto\_\_应该指向 Function.prototype，就是他自己，自己指向自己，没有意义。  
JS一直强调万物皆对象，函数对象也是对象，给他认个祖宗，指向Object.prototype。Object.prototype.\_\_proto\_\_ === null，保证原型链能够正常结束。

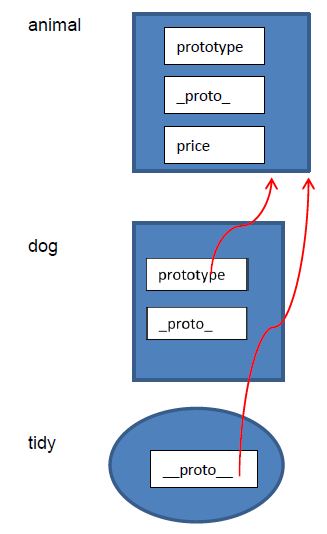
**五．constructor**  
  原型对象prototype中都有个预定义的constructor属性，用来引用它的函数对象。这是一种循环引用  
  person.prototype.constructor === person //true  
  Function.prototype.constructor === Function //true  
  Object.prototype.constructor === Object //true  
  
完善下上面的内存结构图：

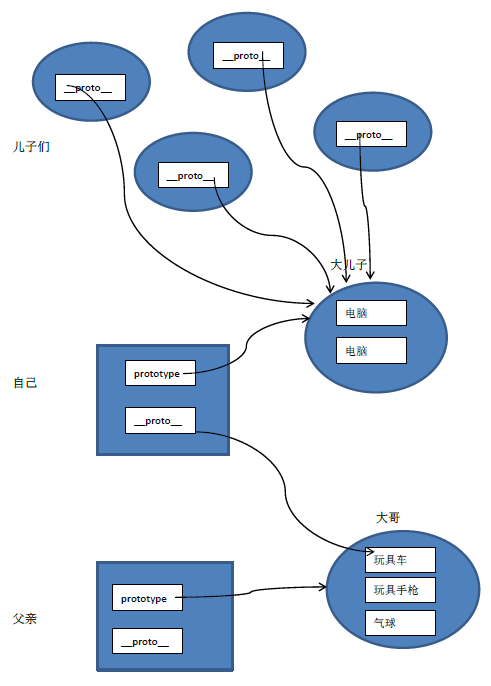
没有显示的创建对象



有两点需要注意：  
（1）注意Object.constructor===Function；//true 本身Object就是Function函数构造出来的   
（2）如何查找一个对象的constructor，就是在该对象的原型链上寻找碰到的第一个constructor属性所指向的对象

**六．总结**  
1.原型和原型链是JS实现继承的一种模型。  
2.原型链的形成是真正是靠\_\_proto\_\_ 而非prototype  
  
要深入理解这句话，我们再举个例子，看看前面你真的理解了吗？  
  var animal = function(){};  
  var dog = function(){};  
  
  animal.price = 2000;//  
  dog.prototype = animal;  
  var tidy = new dog();  
  
  
  console.log(dog.price) //undefined  
  console.log(tidy.price) // 2000  
  
为什么呢？画一下内存图：



  这说明什么问题呢，执行dog.price的时候，发现没有price这个属性，虽然prototype指向的animal有这个属性，但它并没有去沿着这个“链”去寻找。同样，执行tidy.price的时候，也没有这个属性，但是\_\_proto\_\_指向了animal，它会沿着这个链去寻找，animal中有price属性，所以tidy.price输出2000。由此得出，原型链的真正形成是靠的\_\_proro\_\_，而不是prototype。  
因此，如果在这样指定dog.\_\_proto\_\_ = animal。那dog.price = 2000。  
  
最后打个比喻，虽然不是很确切，但可能对原型的理解有些帮助。  
IMG_256

   父亲（函数对象），先生了一个大儿子（prototype），也就是你大哥，父亲给你大哥买了好多的玩具，当你出生的时候，你们之间的亲情纽带（\_\_proto\_\_）会让你自然而然的拥有了你大哥的玩具。同样，你也先生个大儿子，又给他买了好多的玩具，当你再生儿子的时候，你的小儿子会自然拥有你大儿子的所有玩具。至于他们会不会打架，这不是我们的事了。  
所以说，你是从你大哥那继承的，印证了那句“长兄如父”啊！

## 第二章：Javascript 面向对象编程

一、创建对象（封装）

1.**工厂模式**

new Object和对象字面量的方式都能创建对象但是，如果要创建成千上万个对象，就显得十分不方便。

而工厂模式提供了工厂化创建对象的方式（调用函数传递参数就可以创建一个对象）

function createPerson(name,age,job){

Var o = new Object();

o.name = name;

o.age = age;

o.job = job;

return o;

}

var person1 = createPerson(‘zhangsan’,’20’,’doctor’);

调用createPerson并传递参数，就可以得到相应的对象。

问题是工厂模式，不能解决对象识别的问题（就是对象的类型），例如：你可以使用上面的工厂函数，

创建一个人也可以创建一个狗，这就是最大的问题。

1. **构造函数模式**

构造函数可以创建特定类型的对象。下面讲工厂模式重写

function Person(name,age,job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function (){

alert(this.name);

}

}

var person1 = new Person(“zhangsan”,23,”doctor”);

var person2 = new Person(“lisi”,25,”teacher”);

**构造函数与工厂模式的不同点：**

* 没有显示的创建对象
* 直接将属性和方法赋值给this对象
* 没有return语句

**注意：**使用构造函数是第一个字母要大写，创建实例时要使用new关键字

使用new 创建一个实例时的执行步骤：

* 创建一个新对象
* 将构造函数的作用域赋给新对象（就是this指向新对象）
* 执行构造函数的代码（为对象添加属性）
* 返回新对象

实例的construtor属性:每个实例都有construtor属性，这个属性指向实例的构造函数。

person1.construtor == Person;//true

* + 1. **将构造函数当做函数**

任何函数只要通过new操作符来调用都是构造函数，任何函数不通过new来调用就是普通函数

这里继续使用上面的代码

var person1 = new Person(“zhangsan”,23,”teacher”);//构造函数，this指向person1

Person(‘zhangsan’,23,’teacher’);//普通函数添加到window，this指向window

* + 1. **构造函数的问题**

上面介绍过使用构造函数创建对象，每创建一个实例就要为实例创建一个构造函数的方法。

这样就会造成浪费内存。

1. **原型模式**

我们每创建一个函数都有一个prototype（原型）属性，这个属性是一个指针，指向一个对象，而这个对

象的用途就是包含可以由特定类型的所有实例共享的属性和方法。可以理解为prototype就是通过构造

函数而创建的那个对象实例的原型对象（共享的属性和方法）。

function Person(){

}

Person.prototype.name = “zhangsan”;

Person.prototype.age = 23;

Person.prototype.job = “taecher”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

关于原型和原型链的知识参考js高级程序设计

**原型的问题：**

* 无法传递参数
* 因为原型里的属性和方法是每个实例共享的，这就会导致改变一个实例里的属性和方法其他实例里的属性和方法都会改变。

1. **组合使用原型和构造函数模式（推荐）**

这个方法就是结合构造函数和原型对象的优点而诞生的，具体实现就是封装是把属性放在构造函数里面，

把方法放在原型对象里面。

function Person(){

thisn.ame = “zhangsan”;

this.age = 23;

this.job = “taecher”;

}

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

1. **动态原型模式**

动态原型模式是对组合模式的一种变形，原理和功能都是一样。主要目的是从视觉上看这样更像

一个类的封装。

function Person(){

thisn.ame = “zhangsan”;

this.age = 23;

this.job = “taecher”;

If(typeof this.sayName != “function” || !this.sayName){//判断sayName是否定义过

Person.prototype.sayName = function(){

alert(this.name);

};

}

}

var person1 = new Person();

上面的代码的目的是把所有代码都封装在类中。

6.寄生构造函数模式

7.稳妥构造函数模式

二、继承

1.**原型链**

每一个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原

型对象的指针。让一个构造函数的原型对象等于另一个构造函数的实例，这样原型对象就会拥有另一个构

造函数的所有方法和属性。这样就做到了继承，不过这相当于从写原型对象。这时原型对象的construtor

属性指向的不是原先的构造函数。

function A(){

this.name = "a";

}

function B(){

this.age = 23;

}

B.prototype = new A();

var xx = new B();

alert(xx.name);//a

alert(xx.age);//23

alert(xx.construtor);//A(),并不是B();

**注意：**如果想B类继承A类并使用原型链的方式那就要注意，想放在B类的prototype里的属性或是方

法，一定要在B类继承完A类后再进行添加。

**原生连继承的问题：**

原生链继承和利用原生对象封装类的问题相似，一是无法传递参数，二是属性和方法共享会造成

修改一个实例下的引用对象就会影响到其他实例。

function A(){

this.name = "a";

}

A.prototype.friend = ["111","222","333"];

function B(){

this.age = 23;

}

B.prototype = new A();

var xx = new B();

var yy = new B();

xx.friend.push("444");

console.log(xx.friend);//["111", "222", "333", "444"]

console.log(yy.friend);//["111", "222", "333", "444"]

1. 借用构造函数

使用构造函数的继承，其实是对象冒充的方式进行的。通常使用call()和apply()进行继承。

function A(){

this.name = "zhangsan";

this.age = 23;

this.sayName = function(){

console.log(this.name);

}

}

A.prototype.job = "teacher";

function B(){

A.call(this);

}

var xx = new B();

xx.sayName();//”zhangsan”

console.log(this.job);//undefinde

其实上面的代码就是相当于：把A里面的this指向B。等同于下面的代码：

function A(){

this.name = "zhangsan";

this.age = 23;

this.sayName = function(){

console.log(this.name);

}

}

function B(){

this.newmethod = A;//为B添加一个新方法A

this.newmethod();//执行新方法，执行过程中A里面的this指向B，然后为B添加属性。

delete this.newmethod;//删除新方法

}

var xx = new B();

xx.sayName();//”zhangsan”

**注意：**call()和apply()函数的应用，他们的第一个参数都是this指向的对象。Call()的其他参数就是

函数的其他参数，而apply()只有两个参数，第二个参数是一个数组，就是其他参数组成的。

**构造函数继承的问题**：构造函数继承虽然解决了参数传递的问题，但是不能继承原生对象里面的属性和

方法。还有就是无法服用函数，造成内存浪费。

1. 组合继承（推荐）

组合继承就是使用构造函数继承的方式继承构造函数里的属性，使用原生链的方式继承原型对象的方法。

function A(name){

this.name = name;

}

A.prototype.sayName = function(){

console.log(this.name);

}

function B(name){

A.call(this,name)

}

B.prototype = new A();

B.prototype.construtor = B;

var xx = new B("zhangsan");

xx.sayName();

1. 原型继承

json格式的发明人Douglas Crockford，提出了一个object()函数，可以做到这一点。

　　function object(o) {

　　　　function F() {}

　　　　F.prototype = o;

　　　　return new F();

　　}

这个object()函数，其实只做一件事，就是把子对象的prototype属性，指向父对象，从而使得子对象

与父对象连在一起。

使用的时候，第一步先在父对象的基础上，生成子对象：

1. 浅拷贝

除了使用"prototype链"以外，还有另一种思路：把父对象的属性，全部拷贝给子对象，也能实现继承。

下面这个函数，就是在做拷贝：

　　function extendCopy(p) {

　　　　var c = {};

　　　　for (var i in p) {   
　　　　　　c[i] = p[i];  
　　　　}

　　　　c.uber = p;

　　　　return c;  
　　}

使用的时候，这样写：

　　var Doctor = extendCopy(Chinese);

　　Doctor.career = '医生';

　　alert(Doctor.nation); // 中国

但是，这样的拷贝有一个问题。那就是，如果父对象的属性等于数组或另一个对象，那么实际上，子对象获得的只是一个内存地址，而不是真正拷贝，因此存在父对象被篡改的可能。

请看，现在给Chinese添加一个"出生地"属性，它的值是一个数组。

　　Chinese.birthPlaces = ['北京','上海','香港'];

通过extendCopy()函数，Doctor继承了Chinese。

　　var Doctor = extendCopy(Chinese);

然后，我们为Doctor的"出生地"添加一个城市：

　　Doctor.birthPlaces.push('厦门');

发生了什么事？Chinese的"出生地"也被改掉了！

　　alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门

　　alert(Chinese.birthPlaces); //北京, 上海, 香港, 厦门

所以，extendCopy()只是拷贝基本类型的数据，我们把这种拷贝叫做"浅拷贝"。这是早期jQuery实现继承的方式。

6.深拷贝

所谓"深拷贝"，就是能够实现真正意义上的数组和对象的拷贝。它的实现并不难，只要递归调用"浅拷贝"

就行了。

　　function deepCopy(p, c) {

　　　　var c = c || {};

　　　　for (var i in p) {

　　　　　　if (typeof p[i] === 'object') {

　　　　　　　　c[i] = (p[i].constructor === Array) ? [] : {};

　　　　　　　　deepCopy(p[i], c[i]);

　　　　　　} else {

　　　　　　　　　c[i] = p[i];

　　　　　　}  
　　　　}

　　　　return c;  
　　}

使用的时候这样写：

　　var Doctor = deepCopy(Chinese);

现在，给父对象加一个属性，值为数组。然后，在子对象上修改这个属性：

　　Chinese.birthPlaces = ['北京','上海','香港'];

　　Doctor.birthPlaces.push('厦门');

这时，父对象就不会受到影响了。

　　alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门

　　alert(Chinese.birthPlaces); //北京, 上海, 香港

目前，jQuery库使用的就是这种继承方法。