Javascript 高级面向对象

Javascript 得到对象的回顾

在前面我们学习了 javascript 得到对象的几种方式，下面我们首先来回顾下我们学习的几种得到 javascript 对象的方式有哪几种：

第一种：通过 new Object 得到

```
//第一种方式
var person = new Object();
person.age = 18;
person.name = "刘建宏";
person.say = function() {
    //必须加this，指向person对象所定义的属性
    alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
}
person.say();
```

使用这种定义的方式，虽然可以定义一个对象，但是因为没有类的约束，所以无法实现对象的重复使用，如存在 10 个人，就要定义 10 个 person，太过于麻烦了，在操作过程中存在问题。

第二种：使用 json 得到

我们在编程中发现，当我们需要在网络中传输一个对象数据时，上面的方式无法让我们去传输，我们知道网络中的数据是以字符串的形式传播的，所以 XML 和 json 的数据就可以辅助我们完成数据的传输。

```
//第二种对象的实现
var person = {
    name : "刘帅哥",
    age : 18,
    say : function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
```

```
};
person.say();
```

作业：使用 json 定义一组数据：一个班有 5 个人，每个人有姓名、年龄、性别、每个人都有一组朋友，每个人都有一个方法 say，会说出自己的相关信息。

```
var stus = [
    {
        name:"刘建宏",
        age:18,
        sex:"男",
        friends:["徐天岭","孙静伟","牟文斌"],
        say:function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了,我的
性别是："+sex);
        }
    },{
        name:"刘建宏",
        age:18,
        sex:"男",
        friends:["徐天岭","孙静伟","牟文斌"],
        say:function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了,我的
性别是："+sex);
        }
    },{
        name:"刘建宏",
        age:18,
        sex:"男",
        friends:["徐天岭","孙静伟","牟文斌"],
        say:function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了,我的
性别是："+sex);
        }
    },{
        name:"刘建宏",
        age:18,
        sex:"男",
        friends:["徐天岭","孙静伟","牟文斌"],
        say:function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了,我的
性别是："+sex);
        }
    },{
        name:"刘建宏",
```

```
        age:18,
        sex:"男",
        friends:["徐天岭","孙静伟","牟文斌"],
        say:function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了,我的
性别是："+sex);
        }
    }
];
```

虽然 json 的方式也可以定义对象，但是它和 new Object 一样存在了不能对象重用的缺陷，所以大家研究出了一种工厂方式来定义一个对象，下面我们来使用工厂方式来实现一个对象的定义。

第三种：使用工厂模式得到

因为上面两种方式定义对象无法让对象重复使用，所以在使用的过程中大家摸索出来一种基于工厂模式的定义方式，如下所示：

```
//基于工厂模式的定义方式定义对象
//在一个方法中定义一个对象，将传递进来的
//属性赋给了这个对象
function createOb(name,age) {
    var o = new Object();
    o.name = name;
    o.age = age;
    o.say = function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
    return o;
}

//使用工厂模式的定义方法，有效的解决了对象无法重用的问题
var p1 = createOb("刘建宏",18);
p1.say();
var p2 = createOb("牟文斌",20);
p2.say();
```

我们使用了工厂模式定义了对象，这样就很好的解决了对象无法重用的问题，但是此时又存在了另一个问题，就是我们无法判断得到的对象的类型了，如 typeof 或者 instanceof 来判断类型，仅仅得到一个 Object 类型，所以就推出了基于构造函数的方式，在前面已经告

诉大家了。

第四种：使用构造函数来创建一个对象

这种基于构造函数的创建方式，是 javascript 模拟其他面向对象语言的方式来实现对象的创建的。

```
//基于构造函数的创建对象的方式和基于工厂的方式类似
//最大的区别就是函数的名称就是类的名称，按照面向对象语句的
//潜规则，首字母大写，表示这是一个构造函数
function Person(name,age) {
    this.name = name;
    this.age = age;
    this.say = function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
}

var p1 = new Person("刘建宏",15);
p1.say();
//使用构造函数的好处就是可以使用instanceof来判断这个对象的类型了
alert(p1 instanceof Person)
```

基于构造函数的定义的方式最大的好处除了对象重复使用外，就是让我们还可以判断它的类型。

此时我们发现基于构造函数的定义对象的方式看似已经很完美了，我们需要的问题它都可以解决了，但是如果我们仔细的分析这段代码的话，就会发现这样的代码是存在问题的，为什么呢？

我们通过代码分析得知：say 方法在每个对象创建后都存在了一个方法拷贝（但是我们发现代码在只有调用时，say 方法才会在堆中创建，基于闭包的原理），这样就增加了内存的消耗了，如果在对象中有大量的方法时，内存的消耗就会高，这样不行了。

解决这个问题的就是我们可以把这个方法放到全局函数，这样就所有的对象指向了一个方法。

```
//解决方案就是将方法全部放在外面，成为全局函数
function Person(name,age) {
    this.name = name;
    this.age = age;
    //可以将方法成为全局函数
```

```
    this.say = say;
}
function say() {
    alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
}
var p1 = new Person("刘建宏",15);
p1.say();
var p2 = new Person("老徐",30);
p2.say();
```

但是这样写的话，会带来另一个问题，就是方法一点定义为全局函数，那么 window 对象就可以调用，这样就破坏了对象的封装性。而且如果有大量的方法，这样写导致整体代码充斥着大量的全局函数，这样将不利于开发。所以我们急需一种可以完美的解决上述问题的方案，javascript 给我们提供了一种解决这些问题的方案，就是基于原型的对象创建方案。

封装--Javascript 的原型（prototype）

Prototype，原型的初览

以上方式在创建对象都不太理想，所以我们可以使用 prototype 的方式来完成对象的创建。

如何使用原型创建对象呢？首先写段代码让大家看看：

```
//定义了一个对象
function Person() {
}
//使用原型来给对象赋值
//这样就讲一个对象的属性和方法放在了该对象的原型中
//外界是无法访问到这样数据的
Person.prototype.name = "刘帅哥";
Person.prototype.age = 18;
Person.prototype.say = function() {
    alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
}
var p1 = new Person();
p1.say();//正常访问了
say();//报错了
```

这样我们发现 window 就无法访问到 say 方法了，此时 say 方法只属于 Person 对象独有的方法。很好的解决了封装破坏的情况。

什么是原型

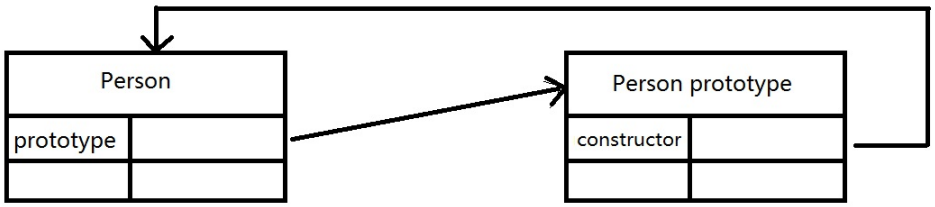
上面我们看了基于 prototype 创建对象的方式很好的解决了我们前面遇到的一系列问题，那么到底什么是原型，原型又是如何解决如上的问题的呢？我们下面来研究研究。

原型是 js 中非常特殊一个对象，当一个函数创建之后，会随之就产生一个原型对象，当通过这个函数的构造函数创建了一个具体的对象之后，在这个具体的对象中就会有一个属性指向原型。这就是原型的概念。

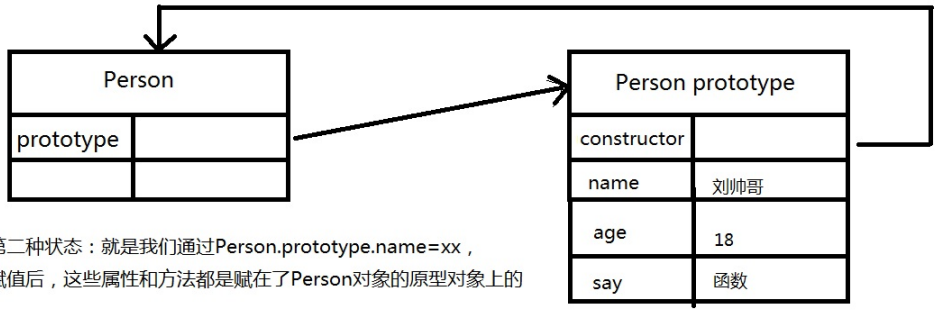
鉴于原型的概念比较难以理解，我们就以上面的代码为例，画图为大家讲解：

```
//第一种状态
//定义了一个对象
function Person() {
}

//第二种状态，这样赋值就会赋在原型对象中
//使用原型来给对象赋值
//这样就讲一个对象的属性和方法放在了该对象的原型中
//外界是无法访问到这样数据的
Person.prototype.name = "刘帅哥";
Person.prototype.age = 18;
Person.prototype.say = function() {
    alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
}

//第三种状态
var p1 = new Person();
//此时调用的是原型中的，因为自己中没有这些属性和方法
p1.say();//正常访问了

//          say();//报错了
//可以通过如下的方式检测p1是不是指向Person的原型对象
//          alert(Person.prototype.isPrototypeOf(p1))
var p2 = new Person();
p2.name = "张三";
p2.age = 20;
p2.say();
```

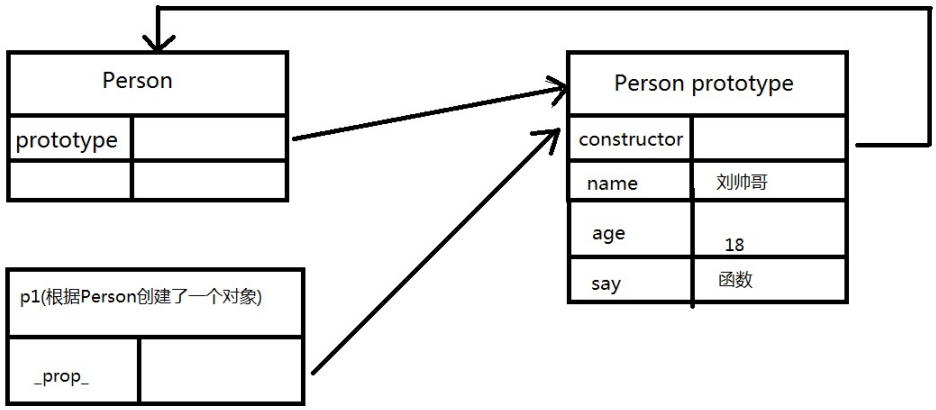
原型的内存模型图如下：



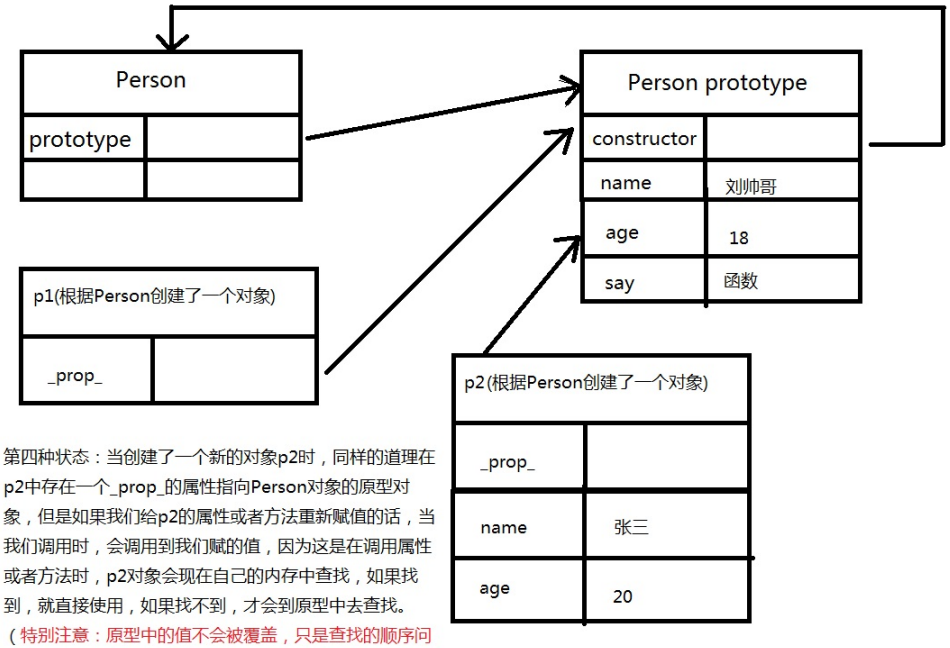
第一步：`function Person() {}`之后，内存中创建了一个Person对象，有一个prototype属性，指向了Person对象的原型对象，而原型对象中存在了一个constructor的属性，指向了Person对象。



第二种状态：就是我们通过Person.prototype.name=xx，
赋值后，这些属性和方法都是赋在了Person对象的原型对象上的



第三种状态：当根据Person构造函数创建一个对象后，该对象中存在一个_prop_的属性，也指向了Person对象的原型对象，当我们调用该对象的属性或者方法时，首先在自己里面找，找不到的话，就会去Person对象的原型对象中找



第四种状态：当创建了一个新的对象p2时，同样的道理在p2中存在一个_prop_的属性指向Person对象的原型对象，但是如果我们给p2的属性或者方法重新赋值的话，当我们调用时，会调用到我们赋的值，因为这是在调用属性或者方法时，p2对象会现在自己的内存中查找，如果找到，就直接使用，如果找不到，才会到原型中去查找。（特别注意：原型中的值不会被覆盖，只是查找的顺序问题）。

原型的基本知识到这里也就差不多了，只有对上面的图和代码能够很好地理解，那么原型的理解就没问题，下面介绍几种原型的检测方式。

常见的原型检测方式

```
可以通过如下的方式检测p1是不是指向Person的原型对象
alert(Person.prototype.isPrototypeOf(p1))
//检测p1的构造器是否指向Person对象
alert(p1.constructor == Person)
//检测某个属性是不是自己内存中的
alert(p1.hasOwnProperty("name"));
alert(p2.hasOwnProperty("name"))
```

同样我们可以使用 delete 语句来删除我们赋予对象的自己属性（注意：原型中的是无法删除的），如

```
//可以使用delete语句删除对象中自己的属性，那么就会找到原型中的值
delete p2.name;
p2.say();
alert(p2.hasOwnProperty("name"));
```

检测在某个对象自己或者对应的原型中是否存在某个属性。

```
alert("name" in p1);//true
delete p2.name;//虽然删除了自己的name属性，但是原型中有
alert("name" in p2);//true
//原型和自己中都没有sex属性
alert("sex" in p1);//false
```

那么问题来了？如果检测只在原型中，不在自己中的属性呢？（提问）

```
//我们可以自己写代码来测试属性不在自己，在原型中
function hasPrototypeProperty(obj,prop) {
    if (!obj.hasOwnProperty(prop)) {
        if (prop in obj) {
            return true;
        }
    }
    return false;
}
alert(hasPrototypeProperty(p1,"name"));
alert(hasPrototypeProperty(p2,"name"));
```

原型重写

在上面的写法中，我们已经解决了大量的问题，使用原型。但是如果我们的对象中存在大量的属性或者方法的时候，使用上面的方式，感觉要写大量的【对象.prototype.属性名 】这样的代码，感觉不是很好，那么我们可以使用 json 的方式来写：

```
function Person() {
}

Person.prototype = {
    name : "刘帅哥",
    age : 18,
    say : function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
}

var p1 = new Person();
p1.say()
var p2 = new Person();
p2.name = "张三";
p2.age = 20;
p2.say();
```

但是这种写法，我们是将该对象的原型覆盖（注意：这两种写法不一样的，第一种是扩充，第二种是覆盖），就会出现如下的问题：

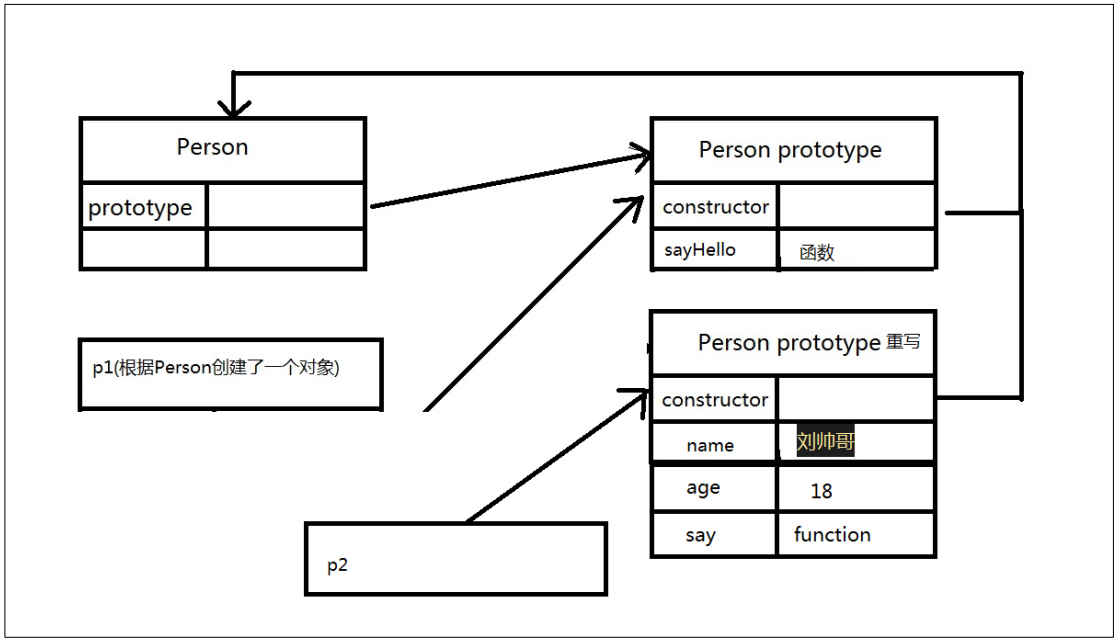
```
function Person() {
}

Person.prototype = {
    constructor:Person,//手动指向Person
    name : "刘帅哥",
    age : 18,
    say : function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
}

var p1 = new Person();
p1.say()
var p2 = new Person();
p2.name = "张三";
```

```
p2.age = 20;
p2.say();

//此时p1的构造器不在指向Person，而是指向了Object
//因为我们覆盖了Person的原型，所以如果constructor比较重要的话，
//我们可以手动指向
alert(p1.constructor == Person)
```

此时就没有问题了。但是原型重写会给我们带来一些非常有趣的现象。下面我们来研究研究。

```
function Person() {
}

var p1 = new Person();

Person.prototype.sayHello = function() {
    alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
}

//          p1.sayHello();

Person.prototype = {
    constructor:Person,//手动指向Person
    name : "刘帅哥",
    age : 18,
    say : function() {
        alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
    }
}

var p2 = new Person();
p2.name = "张三";
p2.age = 20;

p1.sayHello();//此时找不到name和age，但是代码正确
p2.say();//正确

p1.say();//错误，因为原型重写了
p2.sayHello();//错误
```

这些代码要研究明白，必须配合之前原型的图来看，下面我画图说明原因：



因为原型重写，需要大家根据原型的原理图来理解，原型的知识也就这些了。

封装--原型创建对象

因为原型存在，我们实现了对象的封装，但是这种封装也同样可能存在问题的。

1、 我们无法像使用构造函数的那样将属性传递用于设置值

2、 当属性中有引用类型，可能存在变量值的重复

如下面代码：

```
function Person() {
}

Person.prototype = {
    constructor:Person,
    name : "刘帅哥",
    age : 18,
    friends:["老孙","老牟"],
    say : function() {
        alert("我的名字是: "+this.name+",我今年"+this.age+"岁了");
    }
};

var p1 = new Person();
p1.friends.push("老刘");
```

```
alert(p1.friends);

//因为p1和p2对象指向了同一个原型链，所以当p1的friends发生变化是p2页就跟着
var p2 = new Person();
alert(p2.friends);
```

P2 的输出也和 p1 一样：

终极方案—基于组合的对象定义

为了解决原型所带来的问题，需要通过组合构造函数和原型来实现对象的创建将：属性在构造函数中定义，将方法在原型中定义。这种有效集合了两者的优点，是目前最为常用的一种方式。

```
//属性在构造方法定义
function Person(name,age,friends) {
    this.name = name;
    this.age = age;
    this.friends = friends;
}

/**
 * 此时所有的属性都是保存在自己的内存中
 * 方法都是定义在prototype（原型）中
 */
//方法在原型中定义
Person.prototype = {
    constructor:Person,
    say : function() {
        alert("我的名字是: "+this.name+",我今年"+this.age+"岁了");
    }
};

var p1 = new Person("刘帅哥",18,["老孙","老徐"]);
p1.friends.push("老刘");
```

```
alert(p1.friends);

var p2 = new Person("曾小贤",20,["小丽","小美"]);
alert(p2.friends);
```

写了那么多的代码，目的就是为了这种定义 javascript 对象的方式，所以我们最终的定义 javascript 对象的方案就是基于组合的方式定义，将属性的定义放在构造函数中，将方法的定义放在原型中。

终极方案—基于动态原型的对象定义（选学）

上面的方案在我们看来已经相当的完美了，但是因为一些面向对象的程序员（如：java、c#）等开发人员他们认为将方法放在外面不像面向对象的写法，所以提供了另一种写法，在这里说说，经供参考：

```
//属性在构造方法定义
function Person(name,age,friends) {
    this.name = name;
    this.age = age;
    this.friends = friends;
    //判断不存在的时候写
    //如果存在就不在写，内存减少消耗
    if (!Person.prototype.say) {
        Person.prototype.say =  function() {
            alert("我的名字是："+this.name+",我今年"+this.age+"岁了");
        }
    }
}

var p1 = new Person("刘帅哥",18,["老孙","老徐"]);
p1.friends.push("老刘");
alert(p1.friends);

var p2 = new Person("曾小贤",20,["小丽","小美"]);
alert(p2.friends);
```

Javascript 面向对象对应一个对象的方式，上述两种都行，根据个人习惯而定。这也是 javascript 中面向对象的封装。将属性和方法封装到所对应的对象中，其他对象无法得到和访问。

继承--原型创建对象

在面向对象的语言中，存在了三大特性—封装、继承、多态。我们前面一直说 javascript 是面向对象的语言，那么它应该也有面向对象语言这些特性，上面我们看来封装，那么下面我们来研究继承。

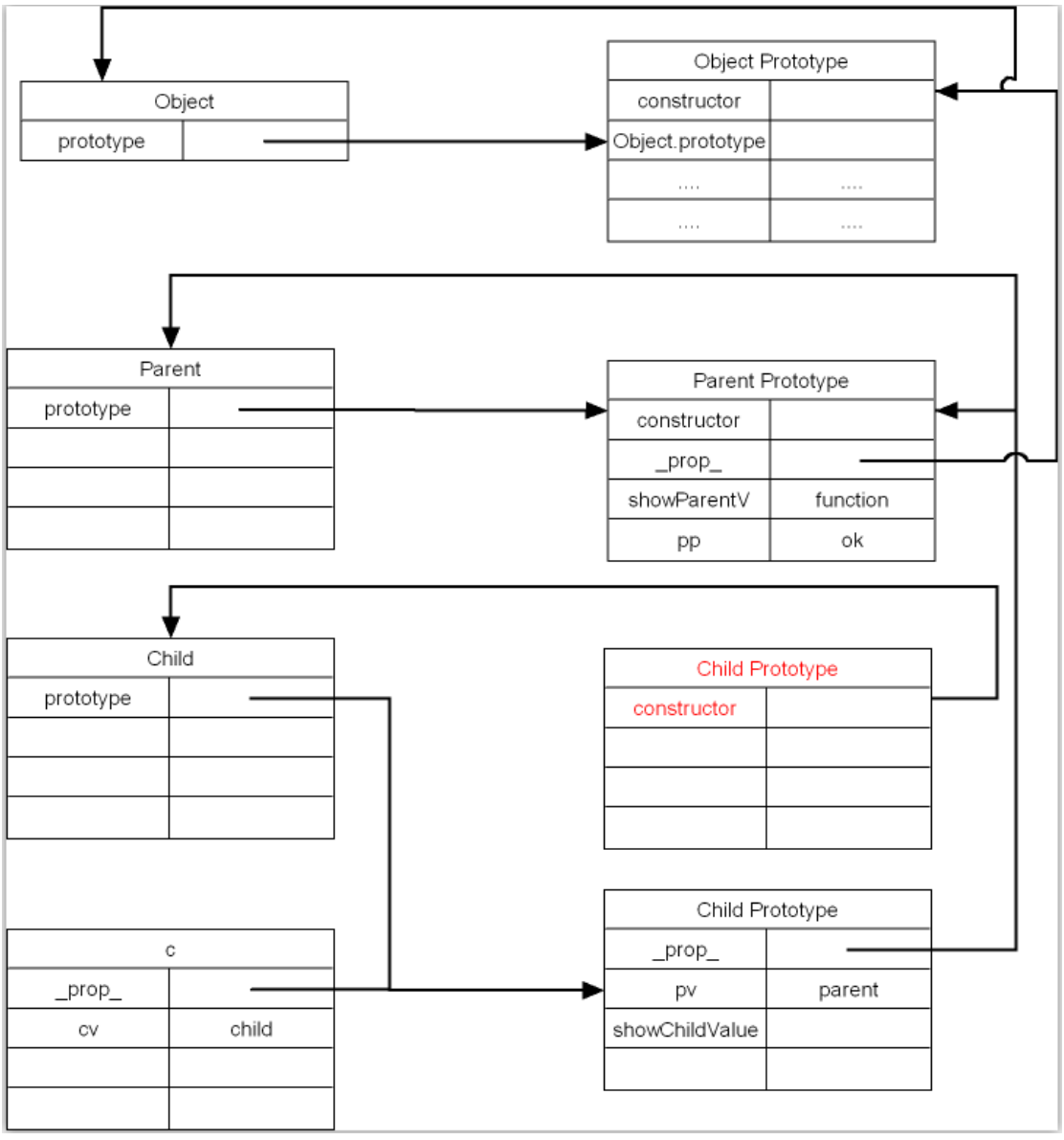
继承，望名而知意，就是我们现实社会中的子孙后代继承了父辈的财富，我们一直在说，面向对象的语言就是在模拟现实世界，通过模拟现实世界来编程，那么在 javascript 中，如何理解继承，如何实现继承呢？

原型链实现继承

Javascript 实现继承有多种方式，我们来一个一个的研究。首先我们来学习基于原型链来实现继承。

```
//定义一个父类
function Parent() {
    this.pv = "parent";
}

Parent.prototype.showParent = function() {
    alert(this.pv);
}

//定义一个子类
function Son() {
    this.sv = "Son";
}

//使用原型链来实现继承
Son.prototype = new Parent();

Son.prototype.showSon = function() {
    alert(this.sv);
}

var s1 = new Son();
s1.showParent();
s1.showSon();
```

当子类的原型指向父类的对象后，子类就继承了父类，实现了继承，这就是基于原型链的继承。我们可以通过内存模型图来分析这种继承。



但是使用原型链实现继承要注意以下一些问题：

1、 不要在设定了原型链之后，再原型重写

2、 一定要在原型链赋值之后才能添加或者覆盖方法

父类方法的覆盖（重写）

当子类继承父类后，子类如果认为父类的方法不能满足自己或者不太满意父类的方法，可以使用与父类同名的方法来覆盖（也叫重写）父类方法。

注意：javascript 中存在重写，但是没有重载。

原型链继承的缺陷

原型链继承存在的缺陷就是：

1、 无法从子类中调用父类的构造函数，这样就没有办法把子类中属性赋值给父类。

2、 父类中属性是在子类的原型中的，这违背了我们前面所讲的封装的理念（属性在对象中，方法在原型中），会出现前面值的混淆问题。

所以我们一般都不会使用单纯的原型链来实现继承。

基于伪装实现继承

在前面我们学习了 call 和 apply 方法，这两个方法我们知道可以使用：函数名.call(上下文,参数列表)，或：者函数名.apply(上下文,参数数组)的方式来调用函数，这样我们可以通过第一个参数上下文来改变调用函数的对象，那么基于这两个方法，我们可以实现一个基于伪装的继承。

```
//定义一个父类
function Parent() {
    this.pv = "parent";
}

//定义一个子类
function Son() {
    this.sv = "Son";
    Parent.call(this);//注意：此时的this指的是Son的对象
                      //那么就是Son对象调用Parent函数
}

var s1 = new Son();
alert(s1.pv);
```

在子类中的 this 指的就是子类实例化后的对象本身，当我们在子类中使用 call 方法调用父类后，就相当于将父类的构造方法绑定到了子类的对象身上，这样就伪装了子类可以使用父类的构造方法，完成了继承。

子类初始化父类属性

基于原型链的继承我们说了缺陷就是无法实现子类去调用父类的构造函数，这样就无法修改父类的属性，但是基于伪装的完美的解决了这个问题，下面我们来看看：

```
//定义一个父类
function Parent(name) {
    this.name = name;
}

//定义一个子类
function Son(name,age) {
    this.age = age;
    Parent.call(this,name);
}

var s1 = new Son("刘帅哥",18);
var s2 = new Son("老牟",28);
alert(s1.name+","+s1.age);
alert(s2.name+","+s2.age);
```

这样我们可以通过子类来设置父类的属性了。

伪装的缺陷

基于伪装的继承解决了基于原型链的问题，但不是说它就十分完美，它也存在了问题，如下：

由于使用伪造的方式继承，子类的原型不会指向父类，所以父类中写在原型中的方法不会被子类继承，所以子类调用不到父类的方法。

解决的办法就是将父类的方法放到子类中来，但是这样的又违背了封装的理念。

终极方案—基于组合实现继承

基于组合的继承就是：属性的继承基于伪装的方式实现，而方法的继承基于原型链的方式继承。

```
function Parent(name) {
    this.name = name;
    this.friends = ["老许","老孙"];
```

```
}

Parent.prototype.parentSay = function() {
    alert(this.name+"---->"+this.friends);
}

//定义一个子类
function Son(name,age) {
    this.age = age;
    Parent.apply(this,[name]);
}

//使用原型链来实现继承
Son.prototype = new Parent();

Son.prototype.sonSay = function() {
    alert(this.name+"*******==="+this.age);
}

var s1 = new Son("刘帅哥",18);
s1.friends.push("老刘");
s1.parentSay();
s1.sonSay();
var s2 = new Son("老牟",28);
s2.parentSay();
s2.sonSay();
```

这是我们实现继承的终极方案，当然在编程界，还存在着其他的继承方案，但是使用都比较少，所以在这里就不在讲解了，有兴趣的同学可以自己下来找资料看看，如 YUI 的寄生继承等。

我们发现不管是封装还是继承最终方案都是基于组合，就是汲取了这种方案的长处，舍去了缺点。

ECMAScript6—面向对象

ECMAScript6 是下一代 Javascript 标准，这个标准将在 2015 年 6 月得到批准。ES6 是 Javascript 的一个重大的更新，并且是自 2009 年发布 ES5 以来的第一次更新。 它将会在主要的 Javascript 引擎实现以下新的特性。

class 关键字

ES6 在面向对象使用了新特性和语法来实现，启用之前的保留字 class 来申明类，因此在 ES6 之后，JavaScript 就有了类的定义和实现：

```
/**
 * 使用class关键字申明一个类，类名为Parent
 */
class Parent {

    //constructor方法就是Parent的构造方法
    //可以使用它来初始化Parent类的属性
    constructor(name,age) {
        this.name = name;
        this.age = age;
    }

    //直接申明Parent类的方法，say
    say() {
        return this.name +"---->"+this.age;
    }

    //使用static关键字申明静态方法
    //注意静态方法属于类，而不属于对象
    static sayHell() {
        alert("Hello 刘帅哥");
    }
}

//错误
//new Parent().sayHello();
//正确，该方法数据类
//Parent.sayHell();
```

extends

在 ES6 中继承的实现使用 extends 来实现：

```
//使用extends来申明Son类继承Parent类
class Son extends Parent {

    constructor(name,age,sex) {
        //使用super关键字表示父类（超类）
        super(name,age);
        this.sex = sex;
    }

    sayWord() {
        alert(this.sex+"----->"+this.name+"----------"+this.age);
    }

    //使用父类中的同名方法，会覆盖父类方法（override）
    say() {
        return "哈哈";
    }

}

let p1 = new Son("阿里巴巴",15,"男");
//p1.sayWord();
alert(p1.say());
```

虽然 ES6 的 Class 本质上还是语法糖，但这么设计有它的目的，首先这样的书写形式是的 JavaScript 的面向对象的写法就和 Java(一门后端面向对象的语言)看起来很是类似了。其次了简化了写法，还让我们之前将代码书写在构造函数外面的那种方法不存在了。