原型链

> 本文旨在花很少的篇幅讲清楚JavaScript语言中的原型链结构，很多朋友认为JavaScript中的原型链复杂难懂，其实不然，它们就像树上的一串猴子。

1.1 理解原型链

JavaScript中几乎所有的东西都是对象，我们说数组是对象、DOM节点是对象、函数等也是对象，创建对象的Object也是对象（本身是构造函数），那么有一个重要的问题： 对象从哪里来?

这是一句废话，对象当然是通过一定方式创建出来的，根据实际类型不同，对象的创建方式也千差万别。比如函数，我们可以声明函数、使用Function构造函数创建等，比如数组，我们可以直接通过var arr = [] 的方式创建空数组，也可以通过new Array的方式创建，比如普通的对象，我们可以字面量创建、使用内置构造函数创建等等，花样太多了，以至于我们学习的时候头昏脑涨、不得要领。

其实，归根结底所有“类型”的对象都可以认为是由相应构造函数创建出来的。 函数由Function构造函数实例化而来，普通对象由Object构造函数实例化而来，数组对象由Array构造函数实例化而来，至于Object | Array | Function等他们本身是函数，当然也有自己的构造函数。

理解了上面一点，那么接下来我们在理解原型链的时候就会容易得多。

> 请看刺激的推导过程
>
> 前提 所有对象都由构造函数实例化而来，构造函数默认拥有与之相关联的原型对象
> ❒① 构造函数的原型对象也是对象，因此也有自己的构造函数
> ❒② 构造函数原型对象的构造函数，也有与之相关连的原型对象
> ❒③ 构造函数原型对象的原型对象（ `__proto__` ）也有自己的构造函数，其也拥有关联的原型对象
> ☞ 以上就形成了一种链式的访问结构，是为 原型链 。

> ! 其实构造函数也是对象，所以构造函数本身作为对象而言也有自己的构造函数，而这个构造函数也拥有与之相关联的原型对象，以此类推。那么，这就是另一条原型链了。综上，我们可以得出 原型链并不孤单 的结论。

1.2 原型链结构

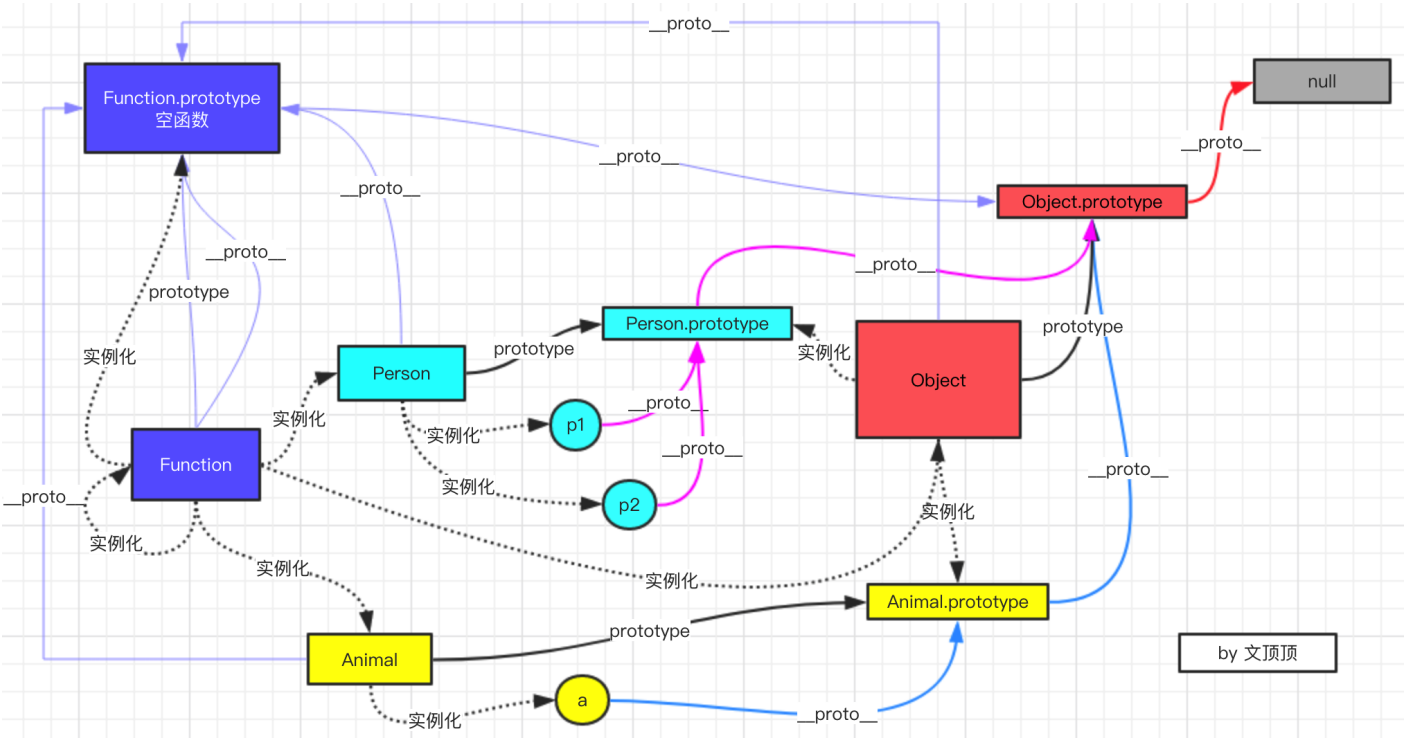
现在我们基本上把原型链的由来说清楚了，那么接下来通过具体的代码来分析原型链的整体结构。

示例代码

```
//01 自定义构造函数Person和Animal
```

```
function Person() {}
function Animal() {}

//02 使用构造函数创建实例对象
var p1 = new Person();
var p2 = new Person();
var a = new Animal();

 //03 创建数组对象
var arrM = ["demoA","demoB"];
```

上面的代码非常简单，其中p1，p2和a它们是自定义构造函数的实例化对象。其次，我们采用快捷方式创建了arrM数组，arrM其实是内置构造函数Array的实例化对象。另外，Person和Animal这两个构造函数其实是Function构造函数的实例对象。理解以上几点后，我们就可以来看一下这几行代码对应的原型链结构图了。



原型链结构图说明：

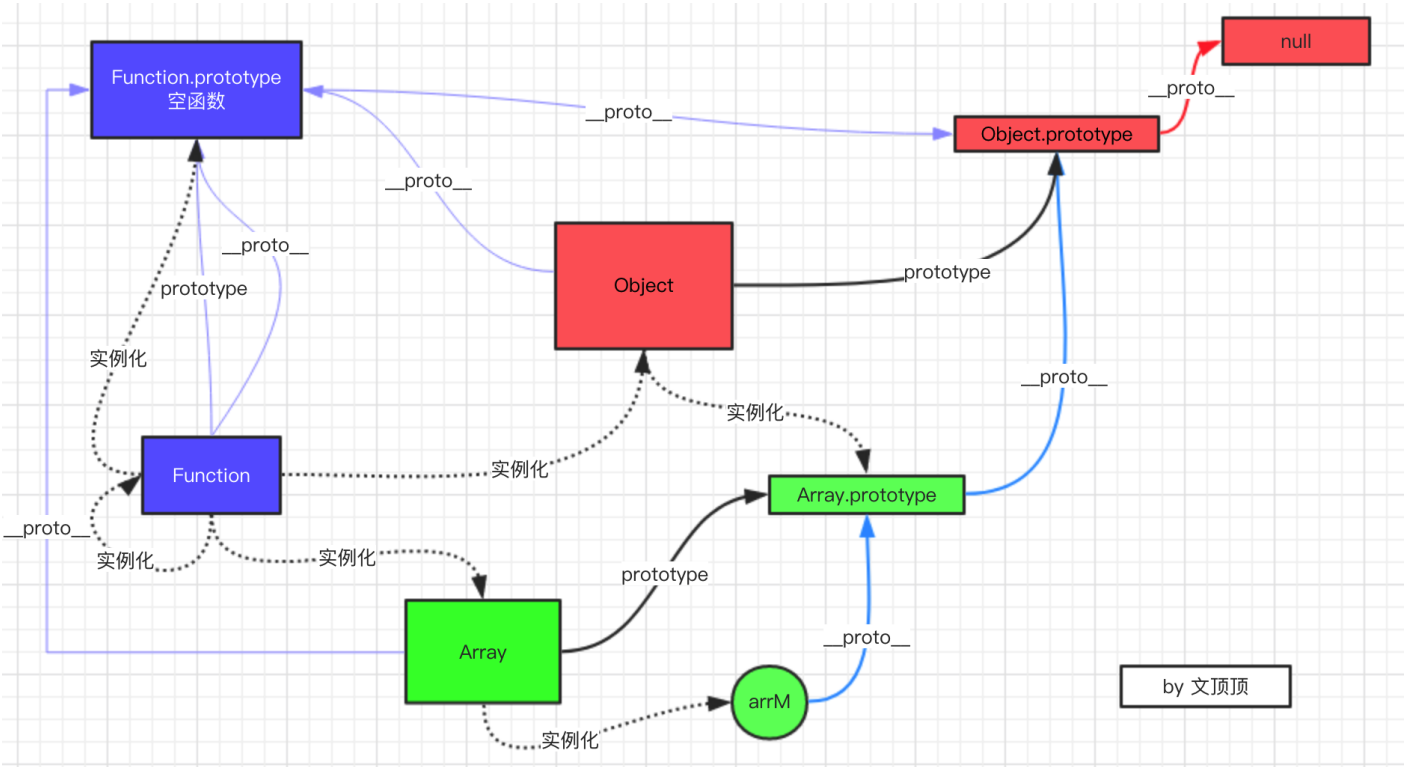
① 因为复杂度关系，arrM对象的原型链结构图单独给出。
② Object.prototype是所有原型链的顶端，终点为null。

验证原型链相关的代码

```
 //[1] 验证p1、p2的原型对象为Person.prototype
//    验证a    的原型对象为Animal.prototype
console.log(p1.__proto__ == Person.prototype); //true
console.log(p2.__proto__ == Person.prototype); //true
console.log(a.__proto__ == Animal.prototype);  //true

//[2] 获取Person.prototype|Animal.prototype构造函数
//    验证Person.prototype|Animal.prototype原型对象为Object.prototype
//    先删除实例成员，通过原型成员访问
delete  Person.prototype.constructor;
```

```
delete  Person.prototype.constructor;
delete  Animal.prototype.constructor;
console.log(Person.prototype.constructor == Object);    //true
console.log(Animal.prototype.constructor == Object);    //true
console.log(Person.prototype.__proto__ == Object.prototype);    //true
console.log(Animal.prototype.__proto__ == Object.prototype);    //true

//[3] 验证Person和Animal的构造函数为Function
//    验证Person和Animal构造函数的原型对象为空函数
console.log(Person.constructor == Function);                //true
console.log(Animal.constructor == Function);                //true
console.log(Person.__proto__ == Function.prototype);        //true
console.log(Animal.__proto__ == Function.prototype);        //true

//[4] 验证Function.prototype的构造函数为Function
console.log(Function.prototype.constructor == Function);    //true


//[5] 验证Function和Object的构造函数为Function
console.log(Function.constructor == Function);              //true
console.log(Object.constructor == Function);                //true

//[6] 验证Function.prototype的原型对象为Object.prototype而不是它自己
console.log(Function.prototype.__proto__ == Object.prototype);//true

//[7] 获取原型链的终点
console.log(Object.prototype.__proto__);                    //null
```

下面贴出数组对象的原型链结构图



验证数组对象原型链结构的代码示例

```
 //[1] 验证arrM的构造函数为Array
//方法1
console.log(arrM.constructor == Array);                 //true
```

```
//方法2
console.log(Object.prototype.toString.call(arrM));      //[object Array]

//[2] 验证Array的构造函数为Function
console.log(Array.constructor == Function);              //true
//[3] 验证Array构造函数的原型对象为Function.prototype(空函数)
console.log(Array.__proto__ == Function.prototype);      //true
//[4] 验证Array.prototype的构造函数为Object,原型对象为Object.prototype
delete Array.prototype.constructor;


console.log(Array.prototype.constructor == Object);          //true
console.log(Array.prototype.__proto__ == Object.prototype); //true
```

1.3 原型链的访问

原型链的访问规则

对象在访问属性或方法的时候，先检查自己的实例成员，如果存在那么就直接使用，如果不存在那么找到该对象的原型对象，查找原型对象上面是否有对应的成员，如果有那么就直接使用，如果没有那么就顺着原型链一直向上查找，如果找到则使用，找不到就重复该过程直到原型链的顶端，此时如果访问的是属性就返回undefined，方法则报错。

```
 function Person() {
    this.name = "wendingding";
}

Person.prototype = {
    constructor:Person,
    name:"自来熟",
    showName:function () {
        this.name.lastIndexOf()
    }
};

var p = new Person();
console.log(p.name);   //访问的是实例成员上面的name属性：wendingding
p.showName();          //打印wendingding
console.log(p.age);    //该属性原型链中并不存在，返回undefined
p.showAge();           //该属性原型链中并不存在，报错
```

> 概念和访问原则说明
> ❒ 实例成员：实例对象的属性或者是方法
> ❒ 原型成员：实例对象的原型对象的属性或者是方法
> ❒ 访问原则：就近原则

1.4 getPrototypeOf、isPrototypeOf和instanceof

Object.getPrototypeOf方法用于获取指定实例对象的原型对象，用法非常简单，只需要把实例对象作为参数传递，该方法就会把当前实例对象的原型对象返回给我们。说白了，Object的这个静态方法其作用就是返回实例对象 __proto__ 属性指向的原型prototype。

```
 //01 声明构造函数F
function F() {}

//02 使用构造函数F获取实例对象f
var f = new F();

//03 测试getPrototypeOf方法的使用
console.log(Object.getPrototypeOf(f));  //打印的结果为一个对象，该对象是F相关联的原型对象
console.log(Object.getPrototypeOf(f) === F.prototype);  //true
console.log(Object.getPrototypeOf(f) === f.__proto__);  //true
```

isPrototypeOf方法用于检查某对象是否在指定对象的原型链中，如果在，那么返回结果true，否则返回结果false。

```
//01 声明构造函数Person
function Person() {}

//02 获取实例化对象p
var p = new Person();

//03 测试isPrototypeOf的使用
console.log(Person.prototype.isPrototypeOf(p)); //true
console.log(Object.prototype.isPrototypeOf(p)); //true

var arr = [1,2,3];
console.log(Array.prototype.isPrototypeOf(arr));    //true
console.log(Object.prototype.isPrototypeOf(arr));   //true
console.log(Object.prototype.isPrototypeOf(Person));//true
```

> 上述代码的原型链
> ① p–>Person.prototype –>Object.prototype –>null
> ② arr–>Array.prototype –>Object.prototype –>null
> Object.prototype因处于所有原型链的顶端，故所有实例对象都继承于Object.prototype

instanceof运算符的作用跟isPrototypeOf方法类似，左操作数是待检测的实例对象，右操作数是用于检测的构造函数。如果右操作数指定构造函数的原型对象在左操作数实例对象的原型链上面，则返回结果true，否则返回结果false。

```
 //01 声明构造函数Person
function Person() {}

//02 获取实例化对象p
var p = new Person();

//03 测试isPrototypeOf的使用
console.log(p instanceof Person);    //true
```

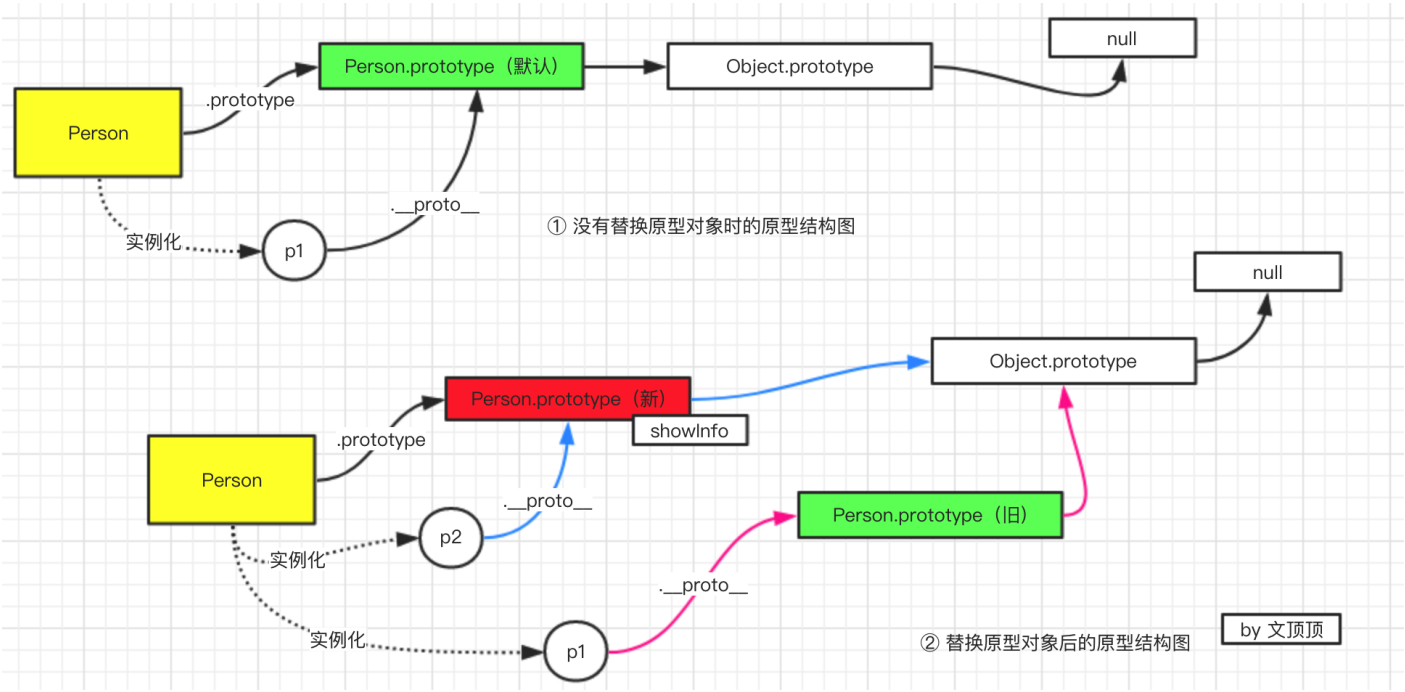
```
console.log(p instanceof Person);   //true
console.log(p instanceof Object);   //true

//04 Object构造函数的原型对象在Function这个实例对象的原型链中
console.log(Function instanceof Object); //true
//05 Function构造函数的原型对象在Object这个实例对象的原型链中
console.log(Object instanceof Function); //true
```

> ! 注意：不要错误的认为instanceof检查的是 该实例对象是否从当前构造函数实例化创建的 ,其实它检查的是实例对象是否从当前指定构造函数的原型对象继承属性。

我们可以通过下面给出的代码示例来进一步理解

```
 //01 声明构造函数Person
function Person() {}

//02 获取实例化对象p
var p1 = new Person();

//03 测试isPrototypeOf的使用
console.log(p1 instanceof Person);   //true

//04 替换Person默认的原型对象
Person.prototype = {
    constructor:Person,
    showInfo:function () {
        console.log("xxx");
    }
};

//05 重置了构造函数原型对象之后，因为Person
console.log(p1 instanceof Person); //false

//06 在Person构造函数重置了原型对象后重新创建实例化对象
var p2 = new Person();
console.log(p2 instanceof Person);   //true

//==> 建议开发中，总是先设置构造函数的原型对象，之后在创建实例化对象
```

贴出上面代码的原型链结构图（部分）



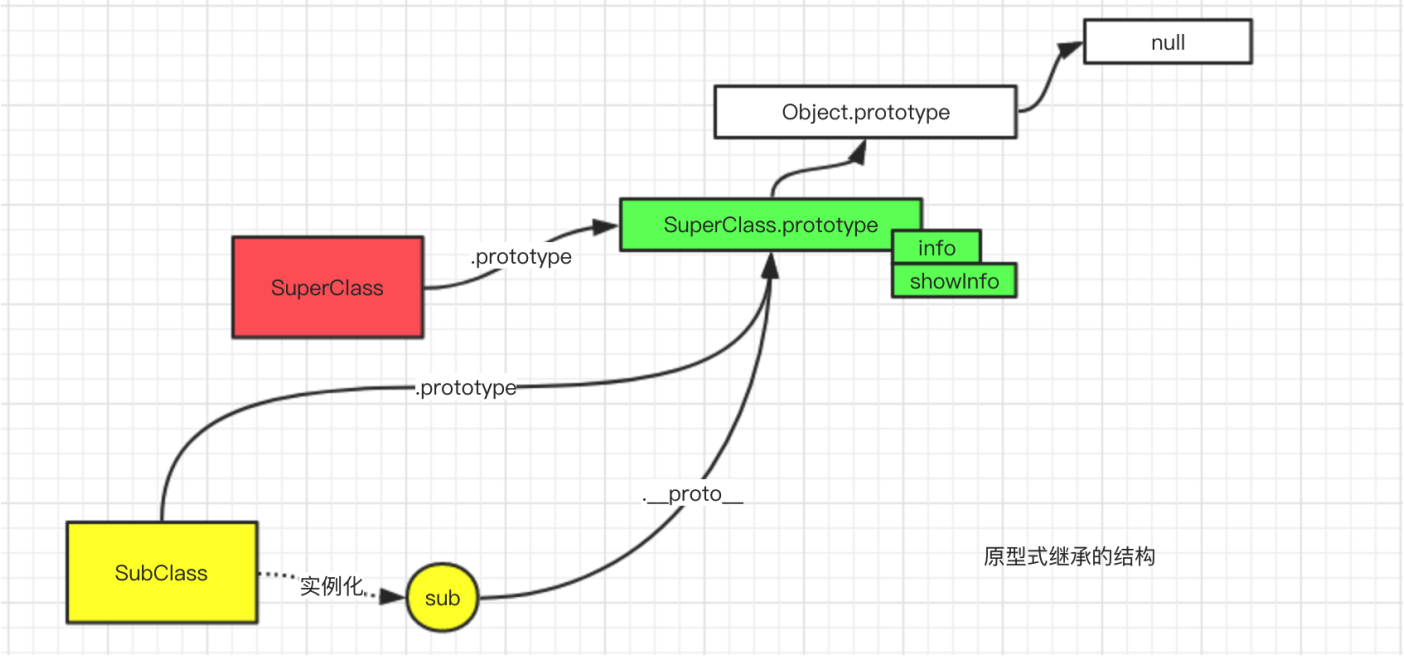
1.5 原型链相关的继承

继承是面向对象编程的基本特征之一，JavaScript支持面向对象编程，在实现继承的时候，有多种可行方案。接下来，我们分别来认识下 原型式继承、原型链继承以及在此基础上演变出来的组合继承 。

原型式继承基本写法

```
 //01 提供超类型|父类型构造函数
function SuperClass() {}

//02 设置父类型的原型属性和原型方法
SuperClass.prototype.info = 'SuperClass的信息';
SuperClass.prototype.showInfo = function () {
    console.log(this.info);
};

//03 提供子类型
function SubClass() {}

//04 设置继承(原型对象继承)
SubClass.prototype = SuperClass.prototype;
SubClass.prototype.constructor = SubClass;

var sub = new SubClass();
console.log(sub.info);          //SuperClass的信息
sub.showInfo();                 //SuperClass的信息
```

贴出原型式继承结构图



! 提示 该方式可以继承超类型中的原型成员，但是存在和超类型原型对象共享的问题

原型链继承

实现思想

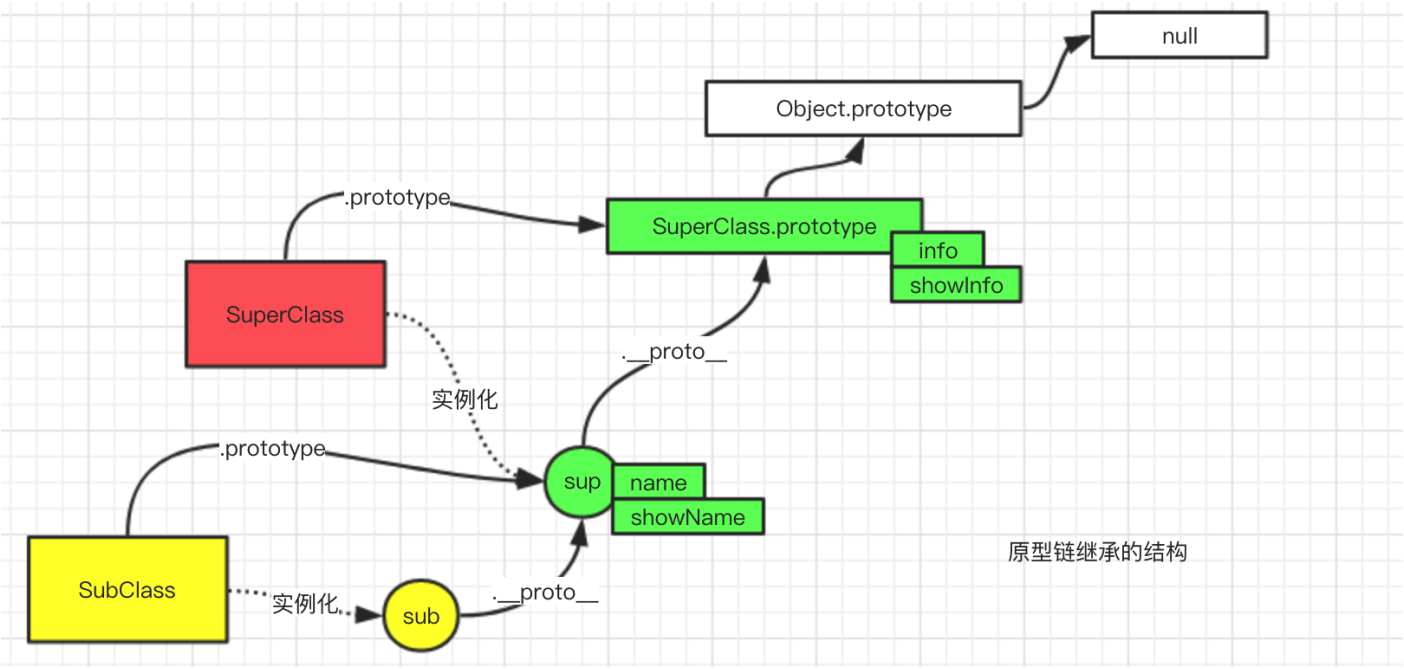
> 核心：把父类的实例对象设置为子类的原型对象 SubClass.prototype = new SuperClass();
> 问题：无法为父构造函数（SuperClass）传递参数

原型链继承基本写法

```
//01 提供超类型|父类型
function SuperClass() {
    this.name = 'SuperClass的名称';
    this.showName = function () {
        console.log(this.name);
    }
}

//02 设置父类型的原型属性和原型方法
SuperClass.prototype.info = 'SuperClass的信息';
SuperClass.prototype.showInfo = function () {
    console.log(this.info);
};

//03 提供子类型
function SubClass() {}

//04 设置继承(原型对象继承)
var sup = new SuperClass();
SubClass.prototype = sup;
SubClass.prototype.constructor = SubClass;
```

```
var sub = new SubClass();
console.log(sub.name);          //SuperClass的名称
console.log(sub.info);          //SuperClass的信息
sub.showInfo();                 //SuperClass的信息
sub.showName();                 //SuperClass的名称
```

贴出原型链继承结构图



组合继承

实现思想

> ① 使用原型链实现对原型属性和方法的继承
> ② 通过伪造(冒充)构造函数来实现对实例成员的继承，并且解决了父构造函数传参问题

组合继承基本写法

```
 //01 提供超类型|父类型
function SuperClass(name) {
    this.name = name;
    this.showName = function () {
        console.log(this.name);
    }
}

//02 设置父类型的原型属性和原型方法
SuperClass.prototype.info = 'SuperClass的信息';
SuperClass.prototype.showInfo = function () {
    console.log(this.info);
};

//03 提供子类型
function SubClass(name) {
    SuperClass.call(this,name);
```

```
    }

    //(1)获取父构造函数的实例成员  Person.call(this,name);
    //(2)获取父构造函数的原型成员  SubClass.prototype = SuperClass.prototype;
    SubClass.prototype = SuperClass.prototype;
    SubClass.prototype.constructor = SubClass;

    var sub_one = new SubClass("zhangsan");
    var sub_two = new SubClass("lisi");
    console.log(sub_one);
    console.log(sub_two);
```

最后，贴出实例对象sub_one和sub_two的打印结果