javascript继承

1.我们的约定

function Fun(){

//私有属性

var a = 1; //私有基本属性

var arr = [1]; //私有引用属性

function fun(){ //私有函数（引用属性）

}

//实例属性

this.a = 1; //实例基本属性

this.arr = [1]; //实例引用属性

this.fun = function(){ //实例函数（引用属性）

}

}

/原型属性

Fun.prototype.a = 1; //原型基本属性

Fun.prototype.aa = [1]; //原型引用属性

Fun.prototype.fun = function(){}; //原型函数（引用属性）

<1>继承的第一种方式：原型链继承

核心：拿父类实例来充当子类原型对象

function Super(){

this.a = 1;

this.arr = [1];

}

function Sub(){

}

Sub.prototype = new Super(); //核心

var sub1 = new Sub();

var sub2 = new Sub();

sub1.a =2;

sub1.arr.push(2);

alert(sub1.a); //2

alert(sub2.a);//1

alert(sub1.arr);//1,2

alert(sub2.arr);//1,2

优缺点：

优点：简单，易于实现；

缺点：

1.修改sub1.arr后sub2.arr也变了，因为来自原型对象的引用属性是所有实例共享的。

可以这样理解：执行sub1.arr.push(2);先对sub1进行属性查找，找遍了实例属性（在本例中没有实例属性），没找到，就开始顺着原型链向上找，拿到了sub1的原型对象，一搜身，发现有arr属性。于是给arr末尾插入了2，所以sub2.arr也变了

2.创建子类实例时，无法向父类构造函数传参。

所以：这种继承是个坑爹货，2大致命缺点使他简直不能用于是开发了第二种继承方式

<2>继承的第二种方式：借用构造函数

核心：借父类的构造函数来增强子类实例，等于是把父类的实例属性复制了一份给子类实例装上了（完全没有用到原型）

function Super(val){

this.val = val;

this.arr = [1];

this.fun = function(){

}

}

function Sub(val){

Super.call(this, val); //核心

}

var sub1 = new Sub(1);

var sub2 = new Sub(2);

sub1.arr.push(2);

alert(sub1.val); //1

alert(sub2.val); //2

alert(sub1.arr); //1,2

alert(sub1.arr); //1

alert(sub1.fun === sub2.fun); //false

**3.优缺点**

优点：

解决了子类实例共享父类引用属性的问题

创建子类实例时，可以向父类构造函数传参

P.S.前辈就这么高效，两个缺陷瞬间修复

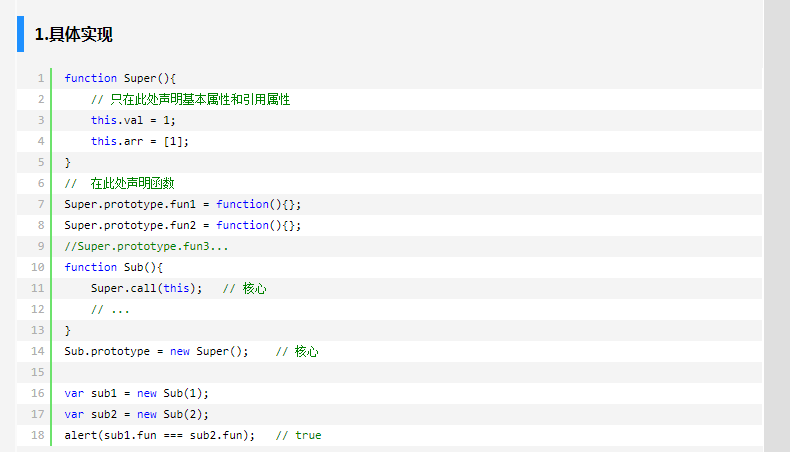
缺点：

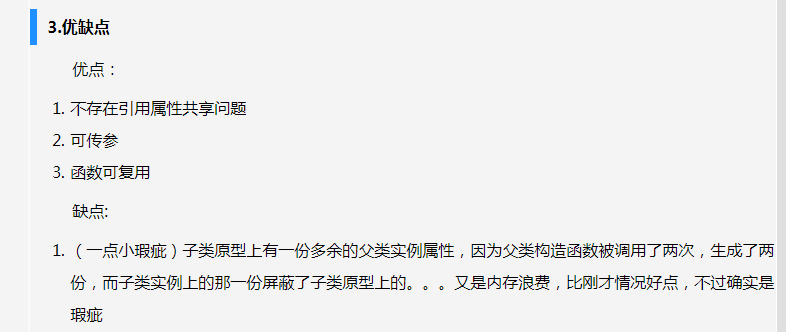
无法实现函数复用，每个子类实例都持有一个新的fun函数，太多了就会影响性能，内存爆炸。。

P.S.好吧，刚修复了共享引用属性的问题，又出现了这个新问题。。

<3>第三种继承方式;组合继承（最常用）

目前我们的借用构造函数方式还是有问题（无法实现函数复用），没关系，接着修复，jsers吭哧吭哧又搞出了组合继承





<4>第四种继承方式：寄生组合继承（最佳方式）



