# [详谈Javascript类与继承](http://www.cnblogs.com/chengzp/p/extends.html)

本文将从以下几方面介绍类与继承

* 类的声明与实例化
* 如何实现继承
* 继承的几种方式

## **类的声明与实例化**

类的声明一般有两种方式

[IMG_256](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

//类的声明var Animal = function () {

this.name = 'Animal';

};

//ES6中类的声明class Animal2 {

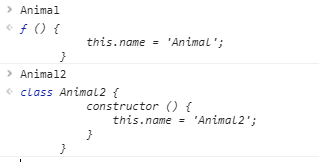
constructor () {

this.name = 'Animal2';

}

}

[IMG_257](http://www.cnblogs.com/chengzp/p/javascript:void(0);)



实例化就比较简单，直接用new运算符

new Animall()

new Animal2()

 这些比较简单，简单介绍一下就可以了。接下来，介绍本文的重点内容，继承。

## **如何实现继承**

实现继承的方式主要有两种：

**第一种借助构造函数实现继承**

先看个了例子

[IMG_259](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

function Parent1 () {

this.name = 'parent1';

}function Child1 () {

Parent1.call(this); //这里的call用apply也可以

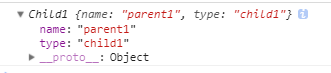
this.type = 'child1';

}

console.log(new Child1());

[IMG_260](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

 输出结果



可以看到，生成Child1里面有了父级的属性name，实现了继承。为什么就实现继承了呢？

因为在Child1里执行了这句   Parent1.call(this);  如果对this不理解的话，建议看看这个[JavaScript作用域和闭包](http://www.cnblogs.com/chengzp/p/scopechain.html" \t "http://www.cnblogs.com/chengzp/p/_blank)

在子类的函数体里执行父级的构造函数，同时改变函数运行的上下文环境（也就是this的指向），使this指向Child1这个类，从而导致了父类的属性都会挂载到子类这个类上去，如此便实现了继承。

但这种继承的方法有一个缺点，它只是把父类中的属性继承了，但父类的原型中的属性继承不了。继续上面的代码

Parent1.prototype.say = function () {

console.log("Parent1 prototype")

};

new Child1().say()

IMG_262

从结果中可以看出 Child1中是没有say方法的，因为say是加在父类的原型上的，这种继承方式只改变父类构造函数在子类函数体中的指向，继承不了原型的属性。

**第二种是借助原型链实现继承**

原型链这里直接用了，不再详细介绍了，如果对原型链还不是很了解的话，建议先看看这个，[详谈Javascript原型链](http://www.cnblogs.com/chengzp/p/prototype.html" \t "http://www.cnblogs.com/chengzp/p/_blank)

[IMG_263](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

function Parent2 () {

this.name = 'parent2';

this.play = [1, 2, 3];

}

function Child2 () {

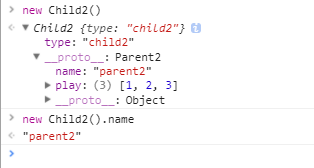
this.type = 'child2';

}

Child2.prototype = new Parent2(); //通过把Child2的原型指向Parent2来实现继承

[IMG_264](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

在浏览器中检验一下



可以看到在Child2的实例的\_\_proto\_\_的属性中有Parent2的属性，由此实现了Child2从Parent2的继承。

但这种继承方式也有不足。接着看代码

[IMG_266](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

var s1 = new Child2();var s2 = new Child2();

s1.play.push(4);

  console.log('s1.play:'+s1.play);  
  console.log('s2.play:'+s2.play);

[IMG_267](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

 打印结果

IMG_268

我们只改了s1这个实例的属性，却发现Child2的其他实例的属性都一起改变了，因为s1修改的是它原型的属性，原型的属性修改，所有继承自该原型的类的属性都会一起改变，因此Child2的实例之间并没有隔离开来，这显然不是我们想要的。

**第三种 组合方式**

组合方式就是前两种方法组合而成的，上面两种方式都有不足，这种方式就解决了上面两种方式的不足。

看代码

[IMG_269](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

function Parent3 () {

this.name = 'parent3';

this.play = [1, 2, 3];

}

function Child3 () {

Parent3.call(this); //子类里执行父类构造函数

this.type = 'child3';

}

Child3.prototype = new Parent3(); //子类的原型指向父类

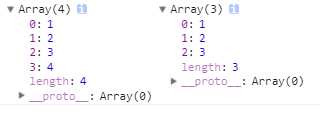
//以下是测试代码var s3 = new Child3();var s4 = new Child3();

s3.play.push(4);

console.log(s3.play, s4.play);

[IMG_270](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

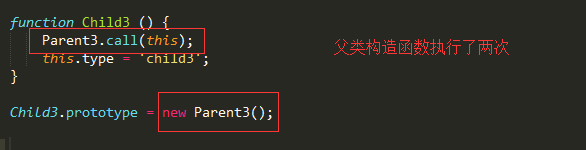
打印结果



可以看出，修改某个实例的属性，并不会引起父类的属性的变化。

这种方式的继承把构造函数和原型链的继承的方式的优点结合起来，并弥补了二者的不足，功能上已经没有缺点了。

但这种方法仍不完美，因为创建一个子类的实例的时候，父类的构造函数执行了两次。



每一次创建实例，都会执行两次构造函数这是没有必要的，因为在继承构造函数的时侯，也就是Parent3.call(this)的时候，parnet的属性已经在child里运行了，外面原型链继承的时候就没有必要再执行一次了。所以，接下来我们对这一方法再做一个优化。

**第四种 组合方式的优化**

 上面一种继承方式问题出在继承原型的时候又一次执行了父类的构造函数，所以优化就从这一点出发。

组合方式中为了解决借助构造函数继承(也就是本文中第一种)的缺点，父类的原型中的属性继承不了，所以才把子类的原型指向了父类。

但是父类的属性，在子类已经中已经存在了，子类只是缺少父类的原型中的属性，所以，根据这一点，我们做出优化。

[IMG_273](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

function Parent4 () {

this.name = 'parent4';

this.play = [1, 2, 3];

}

function Child4 () {

Parent4.call(this);

this.type = 'child4';

}

Child4.prototype = Parent4.prototype; //优化的点在这里

//以下为测试代码var s5 = new Child4();var s6 = new Child4();

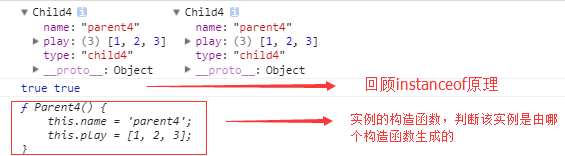
console.log(s5, s6);

console.log(s5 instanceof Child4, s5 instanceof Parent4);

console.log(s5.constructor);

[IMG_274](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

在这种继承方式中，并没有把直接把子类的原型指向父类，而是指向了父类的原型。这样就避免了父类构造函数的二次执行，从而完成了针对组合方式的优化。但还是有一点小问题，先看输出结果



可以看到s5是new Child4()出来的，但是他的constructor却是Parent4.

这是因为Child4这个类中并没有构造函数，它的构造函数是从原型链中的上一级拿过来的，也就是Parent4。所以说到这里，终于能把最完美的继承方式接受给大家啦。

接下来。。。

**第五种 组合的完美优化**

先看代码吧

[IMG_276](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

function Parent5 () {

this.name = 'parent5';

this.play = [1, 2, 3];

}

function Child5 () {

Parent5.call(this);

this.type = 'child5';

}

//把子类的原型指向通过Object.create创建的中间对象

Child5.prototype = Object.create(Parent5.prototype);

//把Child5的原型的构造函数指向自己

Child5.prototype.constructor = Child5;

//测试var s7= new Child5();

console.log(s7 instanceof Child5, s7 instanceof Parent5)

console.log(s7.constructor);

[IMG_277](http://www.cnblogs.com/chengzp/p/javascript:void(0);)

 本例中通过把子类的原型指向Object.create(Parent5.prototype)，实现了子类和父类构造函数的分离，但是这时子类中还是没有自己的构造函数，所以紧接着又设置了子类的构造函数，由此实现了完美的组合继承。

测试结果



### **总结：**

本文并没有直接把最完美的继承直接写出来，而是由浅入深循序渐进的来介绍的，如果对后面几种方法没看太懂的话，可能是原型链掌握的不够好，还是建议看看这个[详谈Javascript原型链](http://www.cnblogs.com/chengzp/p/prototype.html" \t "http://www.cnblogs.com/chengzp/p/_blank)。

类的继承就告一段落了，这部分内容确实不好理解，文章写起来也不好写，可能有的地方语言组织的也不好，有点难懂。大家凑合着看，多看几遍，敲一敲代码就懂了。

如果觉得本文对你有帮助就点个赞吧^\_^