## js继承的概念

js里常用的如下两种继承方式：

原型链继承（对象间的继承）

类式继承（构造函数间的继承）

由于js不像java那样是真正面向对象的语言，js是基于对象的，它没有类的概念。所以，要想实现继承，可以用js的原型[prototype](http://segmentfault.com/blog/trigkit4/1190000000662547)机制或者用apply和call方法去实现

在面向对象的语言中，我们使用类来创建一个**自定义对象**。然而js中所有事物都是对象，那么用什么办法来创建自定义对象呢？这就需要用到js的[原型](http://segmentfault.com/blog/trigkit4/1190000000662547" \t "https://segmentfault.com/a/_blank)：

我们可以简单的把prototype看做是一个模版，新创建的自定义对象都是这个模版（prototype）的一个拷贝 （实际上不是拷贝而是链接，只不过这种链接是不可见，新实例化的对象内部有一个看不见的\_\_Proto\_\_指针，指向原型对象）。

js可以通过构造函数和原型的方式模拟实现类的功能。 另外，js类式继承的实现也是依靠原型链来实现的。

## 原型式继承与类式继承

类式继承是在子类型构造函数的内部调用超类型的构造函数。  
严格的类式继承并不是很常见，一般都是组合着用：

function Super(){

this.colors=["red","blue"];

}

function Sub(){

Super.call(this);

}

原型式继承是借助已有的对象创建新的对象，将子类的原型指向父类，就相当于加入了父类这条原型链

## 原型链继承

为了让子类继承父类的属性（也包括方法），首先需要定义一个构造函数。然后，将父类的新实例赋值给构造函数的原型。代码如下：

<script>

function Parent(){

this.name = 'mike';

}

function Child(){

this.age = 12;

}

Child.prototype = new Parent();//Child继承Parent，通过原型，形成链条

var test = new Child();

alert(test.age);

alert(test.name);//得到被继承的属性

//继续原型链继承

function Brother(){ //brother构造

this.weight = 60;

}

Brother.prototype = new Child();//继续原型链继承

var brother = new Brother();

alert(brother.name);//继承了Parent和Child,弹出mike

alert(brother.age);//弹出12</script>

以上原型链继承还缺少一环，那就是Object，所有的构造函数都继承自Object。而继承Object是自动完成的，并不需要我们自己手动继承，那么他们的从属关系是怎样的呢？

### 确定原型和实例的关系

可以通过两种方式来确定原型和实例之间的关系。操作符instanceof和isPrototypeof()方法：

alert(brother instanceof Object)//true

alert(test instanceof Brother);//false,test 是brother的超类

alert(brother instanceof Child);//true

alert(brother instanceof Parent);//true

只要是原型链中出现过的原型，都可以说是该原型链派生的实例的原型，因此，isPrototypeof()方法也会返回true

在js中，被继承的函数称为超类型（父类，基类也行），继承的函数称为子类型（子类，派生类）。使用原型继承主要由两个问题：  
一是字面量重写原型会中断关系，使用引用类型的原型，并且子类型还无法给超类型传递参数。

伪类解决引用共享和超类型无法传参的问题，我们可以采用“**借用构造函数**”技术

### 借用构造函数（类式继承）

<script>

function Parent(age){

this.name = ['mike','jack','smith'];

this.age = age;

}

function Child(age){

Parent.call(this,age);

}

var test = new Child(21);

alert(test.age);//21

alert(test.name);//mike,jack,smith

test.name.push('bill');

alert(test.name);//mike,jack,smith,bill</script>

借用构造函数虽然解决了刚才两种问题，但没有原型，则复用无从谈起，所以我们需要原型链+借用构造函数的模式，这种模式称为**组合继承**

### 组合继承

<script>

function Parent(age){

this.name = ['mike','jack','smith'];

this.age = age;

}

Parent.prototype.run = function () {

return this.name + ' are both' + this.age;

};

function Child(age){

Parent.call(this,age);//对象冒充，给超类型传参

}

Child.prototype = new Parent();//原型链继承

var test = new Child(21);//写new Parent(21)也行

alert(test.run());//mike,jack,smith are both21</script>

组合式继承是比较常用的一种继承方法，其背后的思路是 使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又保证每个实例都有它自己的属性。

call()的用法：调用一个对象的一个方法，以另一个对象替换当前对象。

call([thisObj[,arg1[, arg2[, [,.argN]]]]])

## 原型式继承

这种继承借助原型并基于已有的对象创建新对象，同时还不用创建自定义类型的方式称为原型式继承

<script>

function obj(o){

function F(){}

F.prototype = o;

return new F();

}

var box = {

name : 'trigkit4',

arr : ['brother','sister','baba']

};

var b1 = obj(box);

alert(b1.name);//trigkit4

b1.name = 'mike';

alert(b1.name);//mike

alert(b1.arr);//brother,sister,baba

b1.arr.push('parents');

alert(b1.arr);//brother,sister,baba,parents

var b2 = obj(box);

alert(b2.name);//trigkit4

alert(b2.arr);//brother,sister,baba,parents</script>

原型式继承首先在obj()函数内部创建一个临时性的构造函数 ，然后将传入的对象作为这个构造函数的原型，最后返回这个临时类型的一个新实例。

### 寄生式继承

这种继承方式是把原型式+[工厂模式](http://segmentfault.com/blog/trigkit4/1190000000662547" \l "articleHeader15)结合起来，目的是为了封装创建的过程。

<script>

function create(o){

var f= obj(o);

f.run = function () {

return this.arr;//同样，会共享引用

};

return f;

}</script>

### 组合式继承的小问题

组合式继承是js最常用的继承模式，但组合继承的超类型在使用过程中会被调用两次；一次是创建子类型的时候，另一次是在子类型构造函数的内部

<script>

function Parent(name){

this.name = name;

this.arr = ['哥哥','妹妹','父母'];

}

Parent.prototype.run = function () {

return this.name;

};

function Child(name,age){

Parent.call(this,age);//第二次调用

this.age = age;

}

Child.prototype = new Parent();//第一次调用</script>

以上代码是之前的组合继承，那么寄生组合继承，解决了两次调用的问题。

### 寄生组合式继承

<script>

function obj(o){

function F(){}

F.prototype = o;

return new F();

}

function create(parent,test){

var f = obj(parent.prototype);//创建对象

f.constructor = test;//增强对象

}

function Parent(name){

this.name = name;

this.arr = ['brother','sister','parents'];

}

Parent.prototype.run = function () {

return this.name;

};

function Child(name,age){

Parent.call(this,name);

this.age =age;

}

inheritPrototype(Parent,Child);//通过这里实现继承

var test = new Child('trigkit4',21);

test.arr.push('nephew');

alert(test.arr);//

alert(test.run());//只共享了方法

var test2 = new Child('jack',22);

alert(test2.arr);//引用问题解决</script>

## call和apply

全局函数apply和call可以用来改变函数中this的指向，如下：

// 定义一个全局函数

function foo() {

console.log(this.fruit);

}

// 定义一个全局变量

var fruit = "apple";

// 自定义一个对象

var pack = {

fruit: "orange"

};

// 等价于window.foo();

foo.apply(window); // "apple",此时this等于window

// 此时foo中的this === pack

foo.apply(pack); // "orange"