**介绍**

本文介绍的四种代码复用模式都是最佳实践，推荐大家在编程的过程中使用。

**模式1：原型继承**

原型继承是让父对象作为子对象的原型，从而达到继承的目的：

function object(o) {  
 function F() {  
 }  
  
 F.prototype = o;  
 return new F();  
}  
  
// 要继承的父对象  
var parent = {  
 name: "Papa"  
};  
  
// 新对象  
var child = object(parent);  
  
// 测试  
console.log(child.name); // "Papa"  
  
  
// 父构造函数  
function Person() {  
 // an "own" property  
 this.name = "Adam";  
}  
// 给原型添加新属性  
Person.prototype.getName = function () {  
 return this.name;  
};  
// 创建新person  
var papa = new Person();  
// 继承  
var kid = object(papa);  
console.log(kid.getName()); // "Adam"  
  
  
// 父构造函数  
function Person() {  
 // an "own" property  
 this.name = "Adam";  
}  
// 给原型添加新属性  
Person.prototype.getName = function () {  
 return this.name;  
};  
// 继承  
var kid = object(Person.prototype);  
console.log(typeof kid.getName); // "function",因为是在原型里定义的  
console.log(typeof kid.name); // "undefined", 因为只继承了原型

同时，ECMAScript5也提供了类似的一个方法叫做Object.create用于继承对象，用法如下：

/\* 使用新版的ECMAScript 5提供的功能 \*/  
var child = Object.create(parent);  
  
var child = Object.create(parent, {  
 age: { value: 2} // ECMA5 descriptor  
});  
console.log(child.hasOwnProperty("age")); // true

而且，也可以更细粒度地在第二个参数上定义属性：

// 首先，定义一个新对象man  
var man = Object.create(null);  
  
// 接着，创建包含属性的配置设置  
// 属性设置为可写，可枚举，可配置  
var config = {  
 writable: true,  
 enumerable: true,  
 configurable: true  
};  
  
// 通常使用Object.defineProperty()来添加新属性(ECMAScript5支持）  
// 现在，为了方便，我们自定义一个封装函数  
var defineProp = function (obj, key, value) {  
 config.value = value;  
 Object.defineProperty(obj, key, config);  
}  
  
defineProp(man, 'car', 'Delorean');  
defineProp(man, 'dob', '1981');  
defineProp(man, 'beard', false);

所以，继承就这么可以做了：

var driver = Object.create( man );  
defineProp (driver, 'topSpeed', '100mph');  
driver.topSpeed // 100mph

但是有个地方需要注意，就是Object.create(null)创建的对象的原型为undefined，也就是没有toString和valueOf方法，所以alert(man);的时候会出错，但alert(man.car);是没问题的。

**模式2：复制所有属性进行继承**

这种方式的继承就是将父对象里所有的属性都复制到子对象上，一般子对象可以使用父对象的数据。

先来看一个浅拷贝的例子：

/\* 浅拷贝 \*/  
function extend(parent, child) {  
 var i;  
 child = child || {};  
 for (i in parent) {  
 if (parent.hasOwnProperty(i)) {  
 child[i] = parent[i];  
 }  
 }  
 return child;  
}  
  
var dad = { name: "Adam" };  
var kid = extend(dad);  
console.log(kid.name); // "Adam"  
  
var dad = {  
 counts: [1, 2, 3],  
 reads: { paper: true }  
};  
var kid = extend(dad);  
kid.counts.push(4);  
console.log(dad.counts.toString()); // "1,2,3,4"  
console.log(dad.reads === kid.reads); // true

代码的最后一行，你可以发现dad和kid的reads是一样的，也就是他们使用的是同一个引用，这也就是浅拷贝带来的问题。

我们再来看一下深拷贝：

/\* 深拷贝 \*/  
function extendDeep(parent, child) {  
 var i,  
 toStr = Object.prototype.toString,  
 astr = "[object Array]";  
  
 child = child || {};  
  
 for (i in parent) {  
 if (parent.hasOwnProperty(i)) {  
 if (typeof parent[i] === 'object') {  
 child[i] = (toStr.call(parent[i]) === astr) ? [] : {};  
 extendDeep(parent[i], child[i]);  
 } else {  
 child[i] = parent[i];  
 }  
 }  
 }  
 return child;  
}  
  
var dad = {  
 counts: [1, 2, 3],  
 reads: { paper: true }  
};  
var kid = extendDeep(dad);  
  
kid.counts.push(4);  
console.log(kid.counts.toString()); // "1,2,3,4"  
console.log(dad.counts.toString()); // "1,2,3"  
  
console.log(dad.reads === kid.reads); // false  
kid.reads.paper = false;

深拷贝以后，两个值就不相等了，bingo！

**模式3：混合（mix-in）**

混入就是将一个对象的一个或多个（或全部）属性（或方法）复制到另外一个对象，我们举一个例子：

function mix() {  
 var arg, prop, child = {};  
 for (arg = 0; arg < arguments.length; arg += 1) {  
 for (prop in arguments[arg]) {  
 if (arguments[arg].hasOwnProperty(prop)) {  
 child[prop] = arguments[arg][prop];  
 }  
 }  
 }  
 return child;  
}  
  
var cake = mix(  
 { eggs: 2, large: true },  
 { butter: 1, salted: true },  
 { flour: '3 cups' },  
 { sugar: 'sure!' }  
 );  
  
console.dir(cake);

mix函数将所传入的所有参数的子属性都复制到child对象里，以便产生一个新对象。

那如何我们只想混入部分属性呢？该个如何做？其实我们可以使用多余的参数来定义需要混入的属性，例如mix（child,parent,method1,method2)这样就可以只将parent里的method1和method2混入到child里。上代码：

// Car   
var Car = function (settings) {  
 this.model = settings.model || 'no model provided';  
 this.colour = settings.colour || 'no colour provided';  
};  
  
// Mixin  
var Mixin = function () { };  
Mixin.prototype = {  
 driveForward: function () {  
 console.log('drive forward');  
 },  
 driveBackward: function () {  
 console.log('drive backward');  
 }  
};  
  
  
// 定义的2个参数分别是被混入的对象（reciving）和从哪里混入的对象（giving)  
function augment(receivingObj, givingObj) {  
 // 如果提供了指定的方法名称的话，也就是参数多余3个  
 if (arguments[2]) {  
 for (var i = 2, len = arguments.length; i < len; i++) {  
 receivingObj.prototype[arguments[i]] = givingObj.prototype[arguments[i]];  
 }  
 }  
 // 如果不指定第3个参数，或者更多参数，就混入所有的方法  
 else {  
 for (var methodName in givingObj.prototype) {  
 // 检查receiving对象内部不包含要混入的名字，如何包含就不混入了  
 if (!receivingObj.prototype[methodName]) {  
 receivingObj.prototype[methodName] = givingObj.prototype[methodName];  
 }  
 }  
 }  
}  
  
// 给Car混入属性，但是值混入'driveForward' 和 'driveBackward'\*/  
augment(Car, Mixin, 'driveForward', 'driveBackward');  
  
// 创建新对象Car  
var vehicle = new Car({ model: 'Ford Escort', colour: 'blue' });  
  
// 测试是否成功得到混入的方法  
vehicle.driveForward();  
vehicle.driveBackward();

该方法使用起来就比较灵活了。

**模式4：借用方法**

一个对象借用另外一个对象的一个或两个方法，而这两个对象之间不会有什么直接联系。不用多解释，直接用代码解释吧：

var one = {  
 name: 'object',  
 say: function (greet) {  
 return greet + ', ' + this.name;  
 }  
};  
  
// 测试  
console.log(one.say('hi')); // "hi, object"  
  
var two = {  
 name: 'another object'  
};  
  
console.log(one.say.apply(two, ['hello'])); // "hello, another object"  
  
// 将say赋值给一个变量，this将指向到全局变量  
var say = one.say;  
console.log(say('hoho')); // "hoho, undefined"  
  
// 传入一个回调函数callback  
var yetanother = {  
 name: 'Yet another object',  
 method: function (callback) {  
 return callback('Hola');  
 }  
};  
console.log(yetanother.method(one.say)); // "Holla, undefined"  
  
function bind(o, m) {  
 return function () {  
 return m.apply(o, [].slice.call(arguments));  
 };  
}  
  
var twosay = bind(two, one.say);  
console.log(twosay('yo')); // "yo, another object"  
  
  
// ECMAScript 5给Function.prototype添加了一个bind()方法，以便很容易使用apply()和call()。  
  
if (typeof Function.prototype.bind === 'undefined') {  
 Function.prototype.bind = function (thisArg) {  
 var fn = this,  
slice = Array.prototype.slice,  
args = slice.call(arguments, 1);  
 return function () {  
 return fn.apply(thisArg, args.concat(slice.call(arguments)));  
 };  
 };  
}  
  
var twosay2 = one.say.bind(two);  
console.log(twosay2('Bonjour')); // "Bonjour, another object"  
  
var twosay3 = one.say.bind(two, 'Enchanté');  
console.log(twosay3()); // "Enchanté, another object"