**介绍**

任何编程都提出代码复用，否则话每次开发一个新程序或者写一个新功能都要全新编写的话，那就歇菜了，但是代码复用也是有好要坏，接下来的两篇文章我们将针对代码复用来进行讨论，第一篇文避免篇，指的是要尽量避免使用这些模式，因为或多或少有带来一些问题；第二排是推荐篇，指的是推荐大家使用的模式，一般不会有什么问题。

**模式1：默认模式**

代码复用大家常用的默认模式，往往是有问题的，该模式使用Parent()的构造函数创建一个对象，并且将该对象赋值给Child()的原型。我们看一下代码：

function inherit(C, P) {  
 C.prototype = new P();  
}  
  
// 父构造函数  
function Parent(name) {  
 this.name = name || 'Adam';  
}  
// 给原型添加say功能  
Parent.prototype.say = function () {  
 return this.name;  
};  
// Child构造函数为空  
function Child(name) {  
}  
  
// 执行继承  
inherit(Child, Parent);  
  
var kid = new Child();  
console.log(kid.say()); // "Adam"  
  
var kiddo = new Child();  
kiddo.name = "Patrick";  
console.log(kiddo.say()); // "Patrick"  
  
// 缺点:不能让参数传进给Child构造函数  
var s = new Child('Seth');  
console.log(s.say()); // "Adam"

这种模式的缺点是Child不能传进参数，基本上也就废了。

**模式2：借用构造函数**

该模式是Child借用Parent的构造函数进行apply，然后将child的this和参数传递给apply方法：

// 父构造函数  
function Parent(name) {  
 this.name = name || 'Adam';  
}  
  
// 给原型添加say功能  
Parent.prototype.say = function () {  
 return this.name;  
};  
  
// Child构造函数  
function Child(name) {  
 Parent.apply(this, arguments);  
}  
  
var kid = new Child("Patrick");  
console.log(kid.name); // "Patrick"  
  
// 缺点：没有从构造函数上继承say方法  
console.log(typeof kid.say); // "undefined"

缺点也很明显，say方法不可用，因为没有继承过来。

**模式3：借用构造函数并设置原型**

上述两个模式都有自己的缺点，那如何把两者的缺点去除呢，我们来尝试一下：

// 父构造函数  
function Parent(name) {  
 this.name = name || 'Adam';  
}  
  
// 给原型添加say功能  
Parent.prototype.say = function () {  
 return this.name;  
};  
  
// Child构造函数  
function Child(name) {  
 Parent.apply(this, arguments);  
}  
  
Child.prototype = new Parent();  
  
var kid = new Child("Patrick");  
console.log(kid.name); // "Patrick"  
console.log(typeof kid.say); // function  
console.log(kid.say()); // Patrick  
console.dir(kid);  
delete kid.name;  
console.log(kid.say()); // "Adam"

运行起来，一切正常，但是有没有发现，Parent构造函数执行了两次，所以说，虽然程序可用，但是效率很低。

**模式4：共享原型**

共享原型是指Child和Parent使用同样的原型，代码如下：

function inherit(C, P) {  
 C.prototype = P.prototype;  
}  
  
// 父构造函数  
function Parent(name) {  
 this.name = name || 'Adam';  
}  
  
// 给原型添加say功能  
Parent.prototype.say = function () {  
 return this.name;  
};  
  
// Child构造函数  
function Child(name) {  
}  
  
inherit(Child, Parent);  
  
var kid = new Child('Patrick');  
console.log(kid.name); // undefined  
console.log(typeof kid.say); // function  
kid.name = 'Patrick';  
console.log(kid.say()); // Patrick  
console.dir(kid);

确定还是一样，Child的参数没有正确接收到。

**模式5：临时构造函数**

首先借用构造函数，然后将Child的原型设置为该借用构造函数的实例，最后恢复Child原型的构造函数。代码如下：

/\* 闭包 \*/  
var inherit = (function () {  
 var F = function () {  
 };  
 return function (C, P) {  
 F.prototype = P.prototype;  
 C.prototype = new F();  
 C.uber = P.prototype;  
 C.prototype.constructor = C;  
 }  
} ());  
  
function Parent(name) {  
 this.name = name || 'Adam';  
}  
  
// 给原型添加say功能  
Parent.prototype.say = function () {  
 return this.name;  
};  
  
// Child构造函数  
function Child(name) {  
}  
  
inherit(Child, Parent);  
  
var kid = new Child();  
console.log(kid.name); // undefined  
console.log(typeof kid.say); // function  
kid.name = 'Patrick';  
console.log(kid.say()); // Patrick  
var kid2 = new Child("Tom");  
console.log(kid.say());   
console.log(kid.constructor.name); // Child  
console.log(kid.constructor === Parent); // false

问题照旧，Child不能正常接收参数。

**模式6：klass**

这个模式，先上代码吧：

var klass = function (Parent, props) {  
  
 var Child, F, i;  
  
 // 1.  
 // 新构造函数  
 Child = function () {  
 if (Child.uber && Child.uber.hasOwnProperty("\_\_construct")) {  
 Child.uber.\_\_construct.apply(this, arguments);  
 }  
 if (Child.prototype.hasOwnProperty("\_\_construct")) {  
 Child.prototype.\_\_construct.apply(this, arguments);  
 }  
 };  
  
 // 2.  
 // 继承  
 Parent = Parent || Object;  
 F = function () {  
 };  
 F.prototype = Parent.prototype;  
 Child.prototype = new F();  
 Child.uber = Parent.prototype;  
 Child.prototype.constructor = Child;  
  
 // 3.  
 // 添加实现方法  
 for (i in props) {  
 if (props.hasOwnProperty(i)) {  
 Child.prototype[i] = props[i];  
 }  
 }  
  
 // return the "class"  
 return Child;  
};  
  
var Man = klass(null, {  
 \_\_construct: function (what) {  
 console.log("Man's constructor");  
 this.name = what;  
 },  
 getName: function () {  
 return this.name;  
 }  
});  
  
var first = new Man('Adam'); // logs "Man's constructor"  
first.getName(); // "Adam"  
  
var SuperMan = klass(Man, {  
 \_\_construct: function (what) {  
 console.log("SuperMan's constructor");  
 },  
 getName: function () {  
 var name = SuperMan.uber.getName.call(this);  
 return "I am " + name;  
 }  
});  
  
var clark = new SuperMan('Clark Kent');  
clark.getName(); // "I am Clark Kent"  
  
console.log(clark instanceof Man); // true  
console.log(clark instanceof SuperMan); // true

怎么样？看着是不是有点晕，说好点，该模式的语法和规范拧得和别的语言一样，你愿意用么？咳。。。

**总结**

以上六个模式虽然在某种特殊情况下实现了某些功能，但是都存在各自的缺点，所以一般情况，大家要避免使用。