**笔记原地址：**

[**http://www.w3cfuns.com/notes/13600/e7f56e77304b981d30757de61f35b460.html**](http://www.w3cfuns.com/notes/13600/e7f56e77304b981d30757de61f35b460.html)

**现同步至github**

**一：工厂模式**  
  
function createPerson(name,age,job){  
     var o = new Object();  
     o.name = name;  
     o.age = age;  
     o.sayName = function(){  
     console.log(this.name);    
   }  
     return o;  
}  
//创建对象：  var person = createPerson("Tom",10,"student");  
  
**二：构造函数模式**  
  
function Person(name,age,job){  
     this.name = name;  
     this.age   = age;  
     this.job = job;  
     this.sayName = function{  
     console.log(this.name);  
    }  
}  
//创建对象：  var person = new Person("Peter",20,"engineer");  
//没有显示地创建对象；直接将属性和方法赋给了this对象；没有return 语句；  
//使用new 来创建实例对象(1.创建一个新对象 2.将构造函数的作用域赋给新对象,即this指向新对象 3.执行构造函数中的代码 4.返回新对象)  
  
**三：原型模式**  
  
function Person(){}  
Person.prototype.name = "Mary";  
Person.prototype.age = 18;  
Person.prototype.job = "teacher";  
Person.prototype.sayName = function(){console.log(this.name);}   
  
//创建对象：var person = new Person();  
  
**四：组合使用构造函数模式和原型模式**  
  
function Person(name,age,job){  
       this.name = name;  
       this.age = age;  
       this.job = job;  
       this.friends = ["tom","peter"];  
}  
  
Person.prototype = {  
       constructor:Person,  //以这种方式重写原型链需要绑定constructor，让其prototype重新指向person  
       sayName : function(){console.log(this.name);}  
}  
  
**五：动态原型模式**  
  
function Person(name,age,job){  
   //Attribute  
        this.name = name;  
        this.age = age;  
        this.job = job;  
   //method  
      
        if(typeof this.sayName != "function"){  
          //判断如果不存在sayName方法就在原型链上添加此方法，这里对原型所做的修改能立即在所有实例中得到反映  
          Person.prototype.sayName = function(){console.log(this.name);};  
    }  
}  
  
**六：寄生构造函数模式**  
  
function Person(name,age,job){  
      var o = new Object();  
      o.name = name;  
      o.age = age;  
      o.job = job;  
      o.sayName = function(){  
      console.log(this.name);  
   }  
      return o;  
}  
//发挥的对象与构造函数或者与构造函数的原型属性之间没有关系，不能依赖instanceof操作符来确定对象类型  
  
**七：稳妥构造函数模式**  
  
function Person(name,age,job){  
     var o = new Object();  
     o.sayName = function(){console.log(this.name);};  
     return o;  
}  
//创建对象：var friend = Person("Tom",29,"eigneer");  
//只能通过调用sayName方法获得name的值，安全性高  
  
  
  
**关于作用域安全的构造函数:**  
function Person(name,age,job){  
    this.name = name;  
    this.age = age;  
    this.job  = job;  
}  
var person  = new Person("Nicholas",29,"Software Engineer");//使用new时会创建一个新的Person对象，同时分配属性，this对象会指向新创建的实例  
var person2 = Person("Nicholas",29,"SoftWare Engineer");//this指向全局对象windows导致错误对象属性的意外增加，e.g: window.name//“Nicholas”  
  
**//这里可以构造作用域安全的构造函数来防止对象属性的意外增加：**  
function Person(name,age,job){  
     if(this instanceof Person){  
        this.name = name;  
        this.age = age;  
        this.job = job;  
   }  
    else{  
       return new Person(name,age,job);  
   }  
}  
  
//但是，这样就锁定了可以调用构造函数的环境，如果使用构造函数窃取模式的继承且不使用原型链，那么这个继承很可能被破坏:  
function Polygon(sides){  
     if(this instanceof Polygon){  
          this.sides = sides;  
          this.getArea = function(){return 0;}  
    }  
    else{  
         return new Polygon(sides);  
   }  
}  
function Rectangle(width,height){  
         Polygon.call(this,2);  
         this.width = width;  
         this.height = height;  
         this.getArea = function(){return this.width \* this.height;};  
}  
var rect = new Rectangle(5,10);  
console.log(rect.sides);//undefined  
//这种情况下在call的时候Polygon中判断this并非Polygon的实例，会创建并返回一个新的Polygon对象  
//Rectangle函数中的this对象没有得到增长，同时Polygon.call()返回的值也没有用到，所以Rectangle实例中不会有sides属性  
//可以通过结合使用原型链或者寄生组合则可以解决这个问题：  
Rectangle.prototype = new Polygon();//使Rectange的原型链指向Polygon