创建一个类

1、类的名称，首字母要大写。

2、通过this关键字为类添加属性或方法

例如：var Book=function（id,bookname,price）{

this.id=id;

this.bookname=bookname;

this.price=price;

}

3、通过原型prototype也可以添加属性和方法

Book.prototype.display=function(){}

Book.prototype={

display:function(){}

}

4、这样就是一种封装

实例化对象的时候要用new 关键字

5、问题：通过this添加的属性和方法同在prototype中添加的属性和方法有什么区别

通过this添加的属性和方法是当前对象上添加的。然而JavaScript是一种基于原型prototype的语言，所以每创建一个对象，它都有一个原型用于指向其继承的属性和方法。通过prototype方法并不是对象自身的，所以使用这些方法时，需要通过prototype一级一级的查找通过this定义的属性或者方法是该对象本身拥有的，所以每次通过类创建一个新的对象时，this指向的属性和方法都会得到相应的创建。而通过prototype几次的属性或者方法是每个对象通过prototype访问到，所以我们每次通过类创建一个新对象时这些属性和方法不会再次创建。

6、属性与方法封装（私有属性和方法、共有属性和方法）

由于JavaScript的函数的作用域，声明在函数内部的变量以及方法在外界是访问不到的，通过此特性即可创建类的私有变量和方法。而在函数内部通过this创建的属性和方法，在类创建对象时，每个对象自身都拥有一份并且可以在外部访问到。因此通过this创建的属性可以看作是对象的共有属性和对象的共有方法，而通过this创建的方法，不但可以访问这些对象的共有属性和私有属性。我们看作为特权方法。在对象创建时通过使用特权方法可以初始化对象的一些属性，因此这些在创建对象调用的特权方法也可以看作是构造器。

例如：

var Book=function（id,name,price）

{

var num01=10;

function findid(){

};

this.getId=function(){

}

this.getName=function(){

}

this.id=id;

this.copy=function(){

}

this.setName(name);

this.setPrice(price);

}

7、类的静态公有属性和静态公有方法

Book.chinese=true;

Book.resetTime=function(){

};

静态的属性和方法属于类本身，只用通过类名调用，其对象不能调用。

8、类的公有属性和公有方法

Book.propotype={

isjsbook:false;

display:functin(){

}

}

9、创建对象安全模式

var Book=function（id,name,price）{

if(this instanceof Book)

{

this.id=id;

this.name=name;

this.price=price;

}

else

{

return new Book(id,name,price);

}

}

继承：

1、类式继承

function Person() {

this.superValue = "super";

}

Person.prototype.getSuperValue=function(){

return this.superValue;

};

function Student(){

this.studentValue="student";

}

Student.prototype=new Person();

Student.prototype.getStudentValue= function () {

return this.studentValue;

};

var stu1=new Student();

alert(stu1.getSuperValue());

alert(stu1.getStudentValue());

类式继承的缺点：

·由于子类通过其原型prototype对父类实例化，继承了父类，所以说父类中的共有属性要是引用类型，就会在子类中被所有实例共用，因此一个子类的实例更改子类原型从父类构造函数中继承来的共有属性就会直接影响到其他子类。

·由于子类实现的继承是靠其原型prototype对父类的实例化实现的，因此在创建父类的时候，是无法向父类传递参数的，因此在实例化父类的时候也无法对父类的构造函数内的属性进行初始化。

function Person(){

this.books=['java','javascript','html5'];

}

function Student()

{

}

Student.prototype=new Person();

var stu1=new Student();

var stu2=new Student();

stu2.books;

stu1.books.push('另一本书');

stu2.books;

2、构造函数继承

function Person(name)  
{  
 this.books=['java','javascript','html5'];  
 this.name=name;  
}  
  
Person.prototype.showBooks=function(){  
 return this.books;  
};  
  
function Student(name)  
{  
 Person.call(this,name);  
}  
  
var *stu1*=new Student('汪星人');  
var *stu2*=new Student('猫星人');  
  
*stu1*.books.push('白雪公主');  
  
alert(*stu1*.books);  
alert(*stu1*.name);  
  
alert(*stu2*.books);  
alert(*stu2*.name);

这里call（this,name）语句，是构造函数式继承的精华，由于call这个方法可以更改函数的作用环境，因此在子类中，对person调用这个方法就是将子类中的变量在父类中执行一遍，由于父类中是给this绑定属性的，因此子类自然也就继承了父类的共有属性，由于这种类型的继承没有涉及到原型，所以父类的原型方法自然不会被子类继承，而如果想被子类继承就必须要放在构造函数中，这样创建出来的每个实例都会单独拥有一份而不能共用，这样以来也违背了代码复用的原则。为此要结合使用:称为 组合式继承

3、组合式继承

前两种继承模式的特点：

1）类式继承是通过子类的原型prototype对父类实例化来实现的。

2）构造函数式继承是通过在子类的构造函数作用环境中执行父类的构造函数来实现的。

组合式继承就是实现上述两点：

function Person(name)  
{  
 this.books=['java','javascript','html5'];  
 this.name=name;  
}  
  
Person.prototype.getName=function(){  
 return this.name;  
};  
  
function Student(name,time)  
{  
 Person.call(this,name);  
 this.time=time;  
}  
  
Student.prototype=new Person();  
Student.prototype.getTime=function(){  
 return this.time;  
};  
  
var *stu1*=new Student('汪星人',2016);  
 *stu1*.books.push('新的一本书');  
 alert(*stu1*.books);  
 *stu1*.getName();  
 *stu1*.getTime();  
  
var *stu2*=new Student('猫星人',2017);  
  
alert(*stu2*.books);  
*stu2*.getName();  
*stu2*.getTime();

此类型的继承，父类的构造方法调用了两次，所有还不是完美的继承方式。

4、原型式继承

道格拉斯·克罗福德说：借助原型prototype可以根据已有的对象创建一个新的对象，同时不必创建新的自定义对象类型。

function inheritObject(o)  
{  
 function F(){};  
 F.prototype=o;  
 return new F();  
}  
  
var *book*={  
 name:"jsbook",  
 alikeBook:['css','html']  
};  
  
var *newBook1*=inheritObject(*book*);  
*newBook1*.name='hello';  
*newBook1*.alikeBook.push('xml');  
  
var *newBook2*=inheritObject(*book*);  
*newBook2*.name='hello111';  
*newBook2*.alikeBook.push('xml111');  
  
alert(*newBook1*.name);  
alert(*newBook1*.alikeBook);  
  
alert(*newBook2*.name);  
alert(*newBook2*.alikeBook);  
  
alert(*book*.name);  
alert(*book*.alikeBook);

原型式继承是对类式继承的一个封装，其实其中的过渡对象就相当于类式继承中的子类，只不过在原型式中作为一个过渡对象出现的，目的是为了创建返回的新的实例化对象。所以类式继承中的问题在这里也会出现，不过这种方式由于F过渡类的构造函数中无内容，所以开销比较小，使用起来比较方便。如果感觉有必要可以将F过渡类缓存起来，不必每次创建一个新的过渡类F。而这种过滤也是不必要的 。

跟类式继承一样，父类对象book中的值类型的属性被复制，引用类型的属性被共用。

5、寄生式继承

声明一个基础对象

function inheritObject(o)

{

function F(){};

F.prototype=o;

return new F();

}

var book={

name：'童话',

alikeBook：["一千零一夜","白雪公主"]

};

function createBook(obj){

var o=new inheritObject(obj);

o.getName=function(){

return name;

}

return o;

}

寄生式继承就是对原型继承的第二次封装，并且在这第二次封装过程中对继承对象进行拓展，这样新创建的对象不仅仅有父类中的属性和方法而且还添加新的属性和方法。

6、最终的继承------寄生组合式继承

我们使用过组合式继承，但是将类是继承对构造函数继承结合使用，但这种方式有一个问题

就是子类不是父类的实例，而子类的原型是父类的的实例，所以才有了寄生组合式继承。

寄生当然就是寄生式继承，寄生式继承依托于原型继承，原型继承又于类式继承想象，另外一种就不应该是这些模式了，所以另外一种继承模式应该是构造函数继承把，当然，子类不是父类实例的问题是由于类式继承引起的，正是这两种继承，但是这里寄生式继承有些特殊，这里它处理的不是对象，而是类的原型。寄生式继承的改造。

function inheritPrototype(subClass,superClass)

{

var p=inheritObject(supperClass.prototype);

p.constructor=subClass;

subClass.prototype=p;

}

组合式继承中，通过构造函数继承的属性和方法是没有问题的，所以这里我们主要探索通过寄生式继承重新继承父类的原型。我们需要继承的仅仅是父类的原型，不在需要调用父类的构造函数。也就是在构造函数继承中我们已经调用了父类的构造函数。因此我们需要的是父类的原型对象的一个副本，而这个副本我们通过原型继承便可得到，但是这么直接赋值给子类会有问题的，因为对父类原型对象复制得到的复制对象p中的constructor指向的不是subClass子类的对象，因此寄生式继承中要对复制对象p做一次增强，修复其constructor属性指向不正确的问题最后将得到的复制对象P赋值给子类的原型，这样子类的原型就继承了父类的原型并且没有指向父类的构造函数。

测试类：

function inheritObject(o)  
{  
 function F(){}  
 F.prototype=o;  
 return new F();  
}  
  
function inheritPrototype(subClass,supperClass)  
{  
 var p=inheritObject(supperClass.prototype);  
 p.constructor=subClass;  
 subClass.prototype=p;  
}  
  
function SupperClass(name){  
 this.name=name;  
 this.colors=["red","green","blue"];  
}  
  
SupperClass.prototype.getName=function(){  
 return this.name;  
};  
function SubClass(name,time){  
 SupperClass.call(this,name);  
 this.time=time;  
}  
  
//寄生式继承父类原型  
inheritPrototype(SubClass,SupperClass);  
  
SubClass.prototype.getTime=function()  
{  
 return this.time;  
};  
var *instance1*=new SubClass("book1",2015);  
var *instance2*=new SubClass("book2",2016);  
  
*instance1*.colors.push("black");  
alert(*instance1*.colors);  
alert(*instance2*.colors);  
  
alert(*instance2*.getName());  
alert(*instance2*.getTime());

思路：

创建原型过渡函数------复制一份父类的原型副本函数-----创建父类以及父类的原型，然后创建子类，并在构造函数中实现构造函数式继承-------通过寄生式继承了父类原型，对子添加原型方法。其中（对子类原型的处理，被赋予父类原型的一个引用，这是一个对象，子类在想添加原型方法必须通过prototype.对象，通过点语法的形式一个一个添加方法了。否则直接赋予对象就会覆盖掉从父类原型继承的对象）

7、多继承：

var extend=function(target,source)

{

for(var property in source)

{

target[property]=source[property];

}

return target;

}

extend方法是一个浅复制过程，它只能复制值类型的属性，对于引用类型的属性它无能为力。而在JQuery等一些框架中实现了深复制，就是将源对象中的引用类型在属性再执行一边extend方法而实现的。

var extend=function(target,source)  
{  
 for(var property in source)  
 {  
 target[property]=source[property];  
 }  
 return target;  
};  
var *book*={  
 name:"html5",  
 alike:["red","green","blue"]  
};  
  
var *anotherBook*={  
 color:'black'  
};  
  
extend(*anotherBook*,*book*);  
alert(*anotherBook*.name);  
alert(*anotherBook*.alike);  
  
*anotherBook*.alike.push('xml');  
*anotherBook*.name="js";  
  
alert(*anotherBook*.name);  
alert(*anotherBook*.alike);  
  
alert(*book*.name);  
alert(*book*.alike);

属性复制：

多继承属性复制

var mix=function()

{

var i=1;

len=arguments.length;

target=arguments[0];

arg;

for(;i<len;i++)

{

arg=arguments[i];

for(var property in arg)

{

target[property]=arg[property];

}

}

return target;

}

mix方法的作用就是将传入的多个对象的属性复制到源对象中，这样即可以实现对象属性的继承，这种继承使用的时候需要传入目标对象（第一个参数是需要继承的对象）也可以绑定到Object上这样所有的对象就可以拥有这个方法了

Object.prototype.mix=function()

{ var i=1;

len=arguments.length;

arg;

for(;i<len;i++)

{

arg=arguments[i];

for(var property in arg)

{

target[property]=arg[property];

}

}

}

使用

newBook.mix(book1,book2);

8、多态

就是多种表示形式，在js内就是同一个方法可以调用多次。

function add()

{

var arg=arguments;

len=arg.length;

switch(len)

{

case 0:

return 5;

case 1:

return 5+arg[0];

case 2:

return arg[0]+arg[1];

}

}

类的形式：

function Add(){

function zero(){

return 5;

}

function one(num){

return 5+num;

}

function two(num1,num2)

{

return num1+num2;

}

this.add=function(){

var arg=arguments;

len=arg.length;

swith(len){

case 0: return zero();

case 1: return one(arg[0]);

case 2: return two(arg[0],arg[1])

}

}

}

var A1=new Add();

**设计模式**

1、简单的工厂模式

体用用品店

定义类各个类

篮球类

var Basketball=function(){

this.info='篮球在美国';

};

Basketball.prototype={

getMember: function()

{

console.log('每个队有5名队员 组成');

},

getBallSize:function(){

console.log('篮球比较大');

}

}

足球类

var Football=function(){

this.info='足球在欧洲';

};

Football.prototype={

getMember: function()

{

console.log('每个队有11名队员 组成');

},

getBallSize:function(){

console.log('足球有点小');

}

}

工厂

var sportFactory=function(name)

{

switch(name)

{

case 'NBA':

return new Basketball();

case 'wordCup':

return new Football();

}

}

var football=SportFactory('wordCup');

console.log(football);

console.log(football.info);

football.getMember();

此种模式是通过类实例化对象创建的

2、也是一种工厂模式

function createBook(name,price)

{

var o=new Object();

o.name=name;

o.price=price;

o.getName=fucntion()

{

return this.name;

}

return o;

}

此中方法是通过创建一个新的对象然后包装增强其属性和功能来实现的

3、工厂方法模式

实例：广告投放

var Factory=function(type,content)  
{  
 if(this instanceof Factory)  
 {  
 var s= new this[type](content);  
 }else  
 {  
 return new Factory(type,content);  
 }  
};  
  
Factory.prototype={  
 Java:function(content)  
 {  
 this.content=content;  
 (  
 function(content){  
 var Odiv=document.createElement('div');  
 Odiv.innerHTML=content;  
 Odiv.style.border='1px solid red';  
 document.getElementById('container').appendChild(Odiv);  
 }  
  
 )(content);  
 },  
  
 Javascript:function(content)  
 {  
  
 },  
  
 Php: function(content)  
 {  
  
 }  
};  
  
var *data*=[{type:'Java',content:'Java哪家好'},  
 {type:'Javascript',content:'Javascript哪家好'},  
 {type:'Php',content:'php哪家好'}];  
  
for(var *i*=3;*i*>=0;*i*--)  
{  
 Factory(*data*[*i*].type,*data*[*i*].content);  
}

主要用于创建不同种类的对象

思考：创建不同的按钮

抽象工厂模式

1、抽象工厂模式--通过对类的工厂抽象使其业务用于对产品类簇的创建。而不负责创建某一类产品的实例。

模拟抽象类

var Car =function(){ };

Car.prototype={

getPrice: function(){

return new Error('抽象方法不能调用');

},

getSpeed: function(){

return new Error('抽象方法不能调用');

}

}

抽象工厂模式（代码）

//抽象工厂方法  
var VehicleFactory=function(subType,superType)  
{  
 //判断抽象工厂中是否有该抽象类  
 if(typeof VehicleFactory[superType]==='function')  
 {  
 //缓存类  
 function F(){}  
 //继承父类属性和方法  
 F.prototype=new VehicleFactory[superType]();  
 //将子类constructor指向子类  
 subType.constructor=subType;  
 //子类原型继承父类  
 subType.prototype=new F();  
 }else  
 {  
 throw new Error('未创建该抽象类');  
 }  
};  
//抽象类定义  
VehicleFactory.Car=function(){  
 this.type='Car';  
};  
  
VehicleFactory.Car.prototype= {  
 getPrice: function(){  
 return new Error('抽象方法不能调用');  
 },  
 getSpeed: function(){  
 return new Error('抽象方法不能调用');  
 }  
};  
  
  
  
VehicleFactory.Bus=function(){  
 this.type='Bus';  
};  
  
VehicleFactory.Bus.prototype= {  
 getPrice: function(){  
 return new Error('抽象方法不能调用');  
 },  
 getPassengerNum: function(){  
 return new Error('抽象方法不能调用');  
 }  
};  
  
VehicleFactory.Truck=function(){  
 this.type='Truck';  
};  
  
VehicleFactory.Truck.prototype= {  
 getPrice: function(){  
 return new Error('抽象方法不能调用');  
 },  
 getTrainload: function(){  
 return new Error('抽象方法不能调用');  
 }  
};

说明：

抽象工厂其实是一个实现子类继承父类的方法，在这个方法中我们需要通过传递子类以及要继承父类（抽象类）的名称，并且在抽象工厂方法中又增加了一次对抽象类存在性的一次判断，如果存在，则将子类继承父类的方法。然后子类通过寄生式继承。继承父类过程中有一个地方需要，就是在对过渡类的原型继承时，我们不是继承父类的原型，而是通过new 关键字复制的父类的一个实例，这么做是因为过渡类不应仅仅继承父类的原型方法，还要继承父类的对象属性，所以要通过new关键字将父类的构造函数指向一遍来复制构造函数中的属性和方法。对抽象工厂添加抽象类也很特殊，以为抽象工厂是个方法不需要实例化对象，故只需要一份，一次直接为抽象工厂添加类的属性即可。

抽象与实现

// 定义宝马汽车子类  
var BMW=function(price,speed){  
 this.price=price;  
 this.speed=speed;  
};  
// 抽象工厂实现对Car抽象类的继承  
VehicleFactory(BMW,'Car');  
BMW.prototype.getPrice=function(){  
 return this.price;  
};  
BMW.prototype.getSpeed=function(){  
 return this.speed;  
};  
  
// 兰博基尼汽车子类  
var Lamborghini=function(price,speed){  
 this.price=price;  
 this.speed=speed;  
};  
// 抽象工厂实现对Car抽象类的继承  
VehicleFactory(Lamborghini,'Car');  
Lamborghini.prototype.getPrice=function(){  
 return this.price;  
};  
Lamborghini.prototype.getSpeed=function(){  
 return this.speed;  
};  
  
// 兰博基尼汽车子类  
var YuTong=function(price,passenger){  
 this.price=price;  
 this.passenger=passenger;  
};  
// 抽象工厂实现对Bus抽象类的继承  
VehicleFactory(YuTong,'Bus');  
YuTong.prototype.getPrice=function(){  
 return this.price;  
};  
YuTong.prototype.getSpeed=function(){  
 return this.passenger;  
};  
  
// 陕汽德龙货车子类  
var DeLong=function(price,trainload){  
 this.price=price;  
 this.trainload=trainload;  
};  
// 抽象工厂实现对抽象类Truck的继承  
VehicleFactory(DeLong,'Truck');  
DeLong.prototype.getPrice=function(){  
 return this.price;  
};  
DeLong.prototype.getSpeed=function(){  
 return this.trainload;  
};

测试：

建造者模式

说明：将一个复杂对象的构建层与其表示层相互分离。注重的是创建细节

实例发布简历：

//创建一位人类  
var Human=function(param){  
 //技能  
 this.skill=param && param.skill||'保密';  
 //兴趣爱好  
 this.hobby=param && param.hobby||'保密';  
};  
  
Human.prototype={  
 getSkill:function(){  
 return this.skill;  
 },  
 getHobby:function(){  
 return this.hobby;  
 }  
};  
  
//实例化姓名类  
var Named=function(name){  
 var that=this;  
 //构造器函数解析姓名的姓与名  
 (  
 function(name,that){  
 that.wholeName=name;  
 if(name.indexof(' ')>-1)  
 {  
 that.FirstName=name.slice(0,name.indexOf(' '));  
 that.secondName=name.slice(name.indexOf(' '));  
 }  
 }  
  
 )(name,that);  
};  
//实例化职位  
var Work=function(work){  
 var that=this;  
 //构造函数中通过传入的职位特征来设置相应的职位以及描述  
 (  
 function(work,that){  
 switch (work)  
 {  
 case 'code':  
 that.work='工程师';  
 that.workDescript='每天沉醉于编程人';  
 break;  
 case 'UI':  
 that.work='设计师';  
 that.workDescript='设计是一种艺术';  
 break;  
 case 'teach':  
 that.work='教师';  
 that.workDescript='传道授业解惑者也';  
 break;  
 default :  
 that.work=work;  
 that.workDescript='对不起，我们还不清楚你选择的职位！';  
  
 }  
 }  
 )(work,that);  
};  
Work.prototype={  
 changeWork: function (work) {  
 this.work=work;  
 },  
 changeDescript: function (setence) {  
 this.workDescript=setence;  
 }  
};  
  
//创建应聘者类  
var Person=function(name,work){  
 //创建缓存对象  
 var \_person=new Human();  
 //创建应聘者姓名解析对象  
 \_person.name=new Named(name);  
 //创建应聘者期望职位  
 \_person.work=new Work(work);  
  
 //返回创建的应聘对象  
 return \_person;  
};  
  
//测试  
var *p1*=new Person('xiao ming','code');  
alert(*p1*.skill);  
alert(*p1*.name.FirstName);  
alert(*p1*.work.work);  
alert(*p1*.work.workDescript);  
*p1*.work.changeDescript('更改职位描述');  
alert(*p1*.work.workDescript);

卡片堆砌效果