**单例模式：**很简单，就是一个对象数据类型

1、把描述同一个事物特性的东西都放在一起，起到了分组的作用,例如：person1就把第一个人的特性都放在了同一组中，person1是对象名，在单例模式中，相当于给新开辟的空间起了个名字，所以我们把对象名也叫作"命名空间"

2、我们用的时候，需要加上前缀，找到特定的命名空间下的某一个属性，例如：person2.age 就是找到person2这下面的age 这样一来避免使用全局变量时候的污染和覆盖，不在担心变量冲突了

项目中，通常用单例模式来进行模块化开发（一个页面按照不同的功能区域划分几个模块，团队间协作开发）

我们项目中通常把一个页面按照功能模块进行拆分，由团队一起协作开发

独立、相互调用

拿百度的首页来说，比如我们分为5个模块：搜索、登录、换肤、新闻、天气,需要五个人协作开发

单例模式虽然避免了变量的冲突，也起到了分组的作用，但是属于手工作业模式，效率比较低

第二次变革：**工厂模式：实现批量的生产**

工厂模式体现了函数的封装：把实现相同功能的代码放在同一个函数中，以后再用的话不需要再重新的写一遍了，直接的执行函数即可

工厂模式虽然能解决批量生产的问题，但是不能区分品牌，不能进行产品的差异化(在JS中我们把产品的差异化叫做实例识别)

第三次变革：**构造函数模式**

JS是一门轻量级的脚本编程语言-->**面向对象开发[类的继承、封装、多态]**

JS中的类都有哪些：

内置类(浏览器天生自带的)、自定义类(我们自己写的)

内置类：Window、Object(对象的内置类)、Array是数组的内置类、RegExp、Date、Number、String、Boolean、Function、HTMLCollection、NodeList、HTMLDivElement、HTMLElement、Element、Node、EventTarget、Document...

实例：每一个数组[]都是Array这类的一个实例，每一个{}都是Object这个类的一个实例、每一个数字都是Number这个类的一个实例、每一个字符串都是String这个类的一个实例...

对象:JS中万物皆对象

类:是对对象的一个具体的细分

实例:每一类中具体的事物

自然界中万物皆对象，对象仅仅是一个泛指而已

我们想要认知世界首先要把对象细分成一个个的类别：人类、动物类、植物类....

实例就是某一个类别中具体的东西，比如某一个人就是人类中的一个实例

**构造函数模式和工厂模式的区别：**

1、在执行的时候，createJsPerson()就是普通函数执行，new createJsPerson()就是构造函数模式执行，用构造函数模式执行的话，我们把createJsPerson就不叫做函数名了，叫做类名

2、在函数体中我们不需要手动的创建一个对象了，浏览器会默认的创建一个对象,这个对象是当前函数执行的行为主体(也就是this)，接下来把createJsPerson当做一个普通函数来执行(形参赋值、预解释、代码自上而下执行)，把相关的方法属性放到this上

浏览器默认开辟的那个对象(this)，是当前这个类的一个实例

3、浏览器默认把实例返回，外面的p1就是当前createJsPerson这个类的一个实例

函数体中出现的this就是这个实例

每一个实例都继承类它所属类中的私有的或者是公有的属性和方法

var ary=[];//字面量方式 ary是Array这个类的一个实例

var ary=new Array();//实例创建-->构造函数模式创建

function fn(){}

var res=fn();//普通函数执行-->fn是函数 默认返回值是undefined

var res=new fn();//构造函数模式执行-->fn是类 默认返回一个对象数据类型，这个对象数据类型就是fn这个类的一个实例

console.log(typeof res);//"object"

**function** CreateJsPerson(name, age) {  
 **var** total = 0;//私有变量  
 **this**.name = name;  
 **this**.age = age;  
 **this**.writeJs = **function** () {  
 console.log("我的名字是:" + **this**.name + "，我会写js啦！");  
 }  
}  
**var** p1 = **new** CreateJsPerson("申立莎", 18);  
console.dir(p1);  
p1.writeJs();

1、执行：

普通函数-->createJsPerson();

构造函数-->new createJsPerson(); 通过new执行的，我们把函数名叫做类名，把执行的返回结果叫做这个类的一个实例

2、运行：

普通函数--> 形成一个私有的作用域--形参赋值--预解释--代码执行

构造函数--> 形成一个私有的作用域--在私有作用域中,首先浏览器会默认的创建一个对象数据类型的数据(就是我们这个类的实例)--形参赋值--预解释--代码执行(以当前实例为执行的主体[说明函数中的this是当前的实例]，把属性和方法赋值给这个实例)

构造函数模式既有普通函数执行时候的那一面，又有自己单独的一面

以上就是我们的构造函数模式，CreateJsPerson是一个类，p1是这个类的一个实例，p1继承CreateJsPerson这个类中的所有的属性和方法（私有和公有的），函数体中写在this上的是私有的属性

细节点:

1、console.log(p1.total);//total是CreateJsPerson函数中的一个私有的变量和实例没啥关系,和实例有关系的都是定义在this上的

2、构造函数模式中，我们浏览器会默认创建一个实例，并且返回，不需要手写return也可以；如果我们手写了return，返回的是基本数据类型值没有任何的影响，但是如果是引用数据类型值，会把默认返回的实例进行覆盖(构造函数模式中最好别加返回值)

类的数据类型："function" typeof Array=="function"

实例的数据类型："object" typeof p1=="object"

对象数据类型具体的细分是每一个类的实例

**构造函数模式中this就是当前的实例**

**instanceof:**检测某一个实例是否属于这个类，是的话返回true，不是的话返回false

f1 instanceof Fn ==>true

f1 instanceof Object ==>true 因为实例是对象数据类型的，而所有的对象数据类型都是Object这个内置类的一个实例

console.log(f1.b == f2.b);//false b是f1和f2两个实例的私有的属性，即使看上去一模一样，但是也是自己私有的，所以f1.b和f2.b不相等

检测b是否为f1的一个属性

**in:**检测某一个属性是不是这个对象的（既可以检查私有，也可以检查公有的）

console.log("b" in f1);

**hasOwnProperty:**检测某一个属性是否为这个对象的私有属性(只能检查私有的)

f1.hasOwnProperty("b") ==>true

自己编写一个检测是否为某个对象公有属性的方法 hasPubProperty

function hasPubProperty(obj, attr) {

var isAtr = attr in obj;

var isOwnAtr = obj.hasOwnProperty(attr);

var isPubAtr = false;

if (isAtr == true && isOwnAtr == false) {

isPubAtr = true;

}

return isPubAtr;

//return (attr in obj) && !obj.hasOwnProperty(attr);

}

f1.propertyIsEnumerable() 检测某一个属性是否为可枚举的

构造函数模式虽然解决了实例识别的问题，但是实例间的属性都是私有的，不能公有

第四次变革：**基于构造函数模式出来的原型链模式**

原型链模式-->属性公有

我们在类的原型链(prototype)上定义公有的属性

**function** Fn() {  
 **this**.a = **function** () {  
 console.log("a");  
 }  
 **this**.c = **function** () {  
 console.log(**this**);  
 }  
}  
Fn.prototype.b = **function** () {  
 console.log("b");  
}  
Fn.prototype.c = **function** () {  
 console.log(**this**);  
}  
**var** f1 = **new** Fn;  
**var** f2 = **new** Fn;  
console.log(f1.a == f2.a);//false a是私有的属性  
console.log(f1.b == f2.b);//true b是公有的属性  
  
console.dir(Fn.prototype.constructor);//Fn本身  
console.dir(Fn.prototype);

**查找的机制：**首先在自己私有的属性上找，有的话就是私有的；没有的话，通过\_\_proto\_\_找到所属类的prototype上(这个是公有的)，如果有就是公有的属性；如果还没有，在通过所属类的prototype上的\_\_proto\_\_找到Object基类的prototype上，如果还没有就报错了！

f1.a();  
f1.b();  
f1.hasOwnProperty();  
  
f1.a == f2.a;//false  
f1.b == f2.b;//true  
f1.\_\_proto\_\_.b==f2.\_\_proto\_\_.b;//true  
f1.c==f2.c;//false  
f1.\_\_proto\_\_.c==f2.\_\_proto\_\_.c;//true  
f1.b==Fn.prototype.b;//true  
f1.\_\_proto\_\_==Fn.prototype; //true  
f1.\_\_proto\_\_.b==Fn.\_\_proto\_\_.b;//false

**在原型模式上的this问题：方法执行，方法名前面有点，点前面的是谁this就是谁**

f1.c(); //this-->f1  
f1.\_\_proto\_\_.c();//this-->f1.\_\_proto\_\_或者Fn.prototype  
Fn.prototype.c();//this-->Fn.prototype

原型模式中的新增加属性

f1.d=**function**(){}//给f1新增加一个私有的属性d  
f2.d(); //f1增加的私有属性 f2并没有  
f1.\_\_proto\_\_.m=**function**(){}//在f1所属类的原型上增加一个公有的属性m  
f2.m(); //f1往公有里面增加的属性 f2也能用  
  
console.log(f1.\_\_proto\_\_.\_\_proto\_\_.hasOwnProperty("c"));//false

当前hasOwnProperty方法中的this是f1.\_\_proto\_\_.\_\_proto\_\_，即Object.prototype，而hasOwnProperty中查找的是当前this里面是否有"c"，而Object.prototype没有c，所以是false

用constructor来检测数据类型：

var ary=[];

console.log(ary.constructor==Array);//true

从下面开始，每一句话都是规定，不要问为什么

1、类(Fn)是一个函数数据类型，在JS中函数即使函数数据类型也是对象数据类型

function f() {}

说它是函数数据类型，因为它可以执行:f() 而且形成私有的作用域、arguments、return、形参...

说它是对象数据类型，因为它可以像我们的对象一样增加属性名和属性值，例如：

f.zhuFeng = "我虽然是一个函数，我也是一个对象！";

console.log(f.zhuFeng);

这个属性和我们之前讲函数中的变量等没有关系

2、每一个函数(类也是函数)都有一个天生自带的属性:prototype，这个属性存储的是一个对象数据类型的值，并且这个对象数据类型是浏览器天生给它开的空间

在prototype属性对应的堆内存中，有一个天生自带的属性constructor，这个属性指向函数本身

3、每一个对象数据类型也有一个天生自带的属性：\_\_proto\_\_，这个属性指向所属类的prototype

**在IE下不支持\_\_proto\_\_**

**function** Fn() {  
 **this**.num = 100;  
}  
**var** pro = Fn.prototype;//把原型对应的堆内存的地址给我们的pro  
pro.a = **function** () {  
};

用这种方法给Fn的原型对象上添加一个公有方法a，Fn.prototype方法上默认的constructor属性存在，指向的Fn，不会向上查找Object.prototype上的constructor。

**function** Fn() {  
 **this**.num = 100;  
}  
Fn.prototype.b = **function** () {  
 console.log("b");  
}  
Fn.prototype = {  
 constructor: Fn,  
 a: **function** () {  
 console.log("a");  
 }  
};  
Fn.prototype.c = **function** () {  
 console.log("c");  
}  
**var** f = **new** Fn;

f.a();  
f.b();//Error  
f.c();

1、console.log(f.constructor);//没加之前是Object 加了之后是Fn

2、我们自己给Fn的prototype开辟的那个堆内存不会自带constructor属性(只有浏览器默认开辟的那个才有constructor)，这样如果我们用constructor，找到的是Object.protytype上的constructor，所以：用这种方式增加公有的属性，需要手动添加constructor属性，手动指向当前函数本身

3、自己给Fn的prototype开辟的那个堆内存会覆盖之前浏览器自己默认开辟的那个，所以如果用这种办法来增加公有属性的话，在之前就不要给原型增加任何的属性，否则都会被替换掉(但是写在自己开辟的那个之后是没有问题的，例如c) --->引用数据类型地址操作的机制

在JS中:

1、把相同的代码放到同一个函数中，以后用的时候不需要重新的写了，只需要执行函数就可以--->**类的封装**

2、一个函数通过传递参数的不一样实现了不同的功能，我们就说这个方法有多种的形态--->**类的多态**

3、B这个类继承了A这个类中的所有的属性和方法(公有和私有的都有) --->**类的继承**

**封装、继承、多态 组成了我们的面向对象开发**

**function** A() {  
 **this**.x = 10;  
}  
A.prototype.getX = **function** () {  
 console.log(**this**.x);  
};  
**function** B() {  
 **this**.x = 20;  
}  
//需求：想让B继承A的私有和公有的属性，需要让B的原型等于A的一个实例  
B.prototype = **new** A;  
**var** a = **new** A;  
**var** b = **new** B;  
  
//B这个类继承类A的私有和公有的属性和方法，所以B的实例b可以用A里面的属性和方法  
b.getX();//this-->b 20  
console.log(b.x); //20  
console.log(b.\_\_proto\_\_.x);//10  
  
//JS中的原型继承：子类不仅可以继承父类的所有的属性和方法，而且还可以把父类原型上的属性和方法进行修改-->JS中的继承是相互的  
b.\_\_proto\_\_.\_\_proto\_\_.c = **function** () {  
 console.log("c");  
};  
b.c();  
a.c();

所有的对象数据类型都是Object这个基类的一个实例

所有的函数数据类型都是Function这个基类的一个实例

所有的函数数据类型都天生自带两个属性：\_\_proto\_\_和prototype，\_\_proto\_\_指向所属类的原型，prototype是自己的原型

所有的对象数据类型天生自带一个属性：\_\_proto\_\_，指向所属类的原型

所有浏览器默认给prototype开辟的那个对象空间都有一个天生的constructor属性指向函数本身

**继承的四种方式：**

**function** A() {  
 **this**.x = 100;  
}  
A.prototype.getX = **function** () {  
 console.log(**this**.x);  
};

1、原型链继承

B继承A的私有的和公有的属性方法：B的原型链等于A的一个实例即可

B把A的私有的和公有的属性方法都放在了B的公有属性上

**function** B() {  
 **this**.x = 200;  
}  
B.prototype = **new** A;  
B.prototype.constructor = B;  
**var** b = **new** B;  
b.getX();//this -->b console.log(b.x)

2、call继承

B的私有作用域中，我们执行A方法，把里面的this改变成B的一个实例即可

只能继承A中私有的属性和方法，公有的不可以，并且把继承过来的属性和的方法当做B的私有的属性和方法

**function** B() {  
 A.call(**this**);//this-->b 执行A方法，把this改成B的一个实例b，A中代码执行的时候，this.x=100，就相当于b.x=100;  
 // 换个思路，我们执行A的时候，让里面的this变成了B的一个实例，也就是A中写的所有的this.xxx=xxx都是相当于给B的实例增加了  
}  
**var** b = **new** B;  
console.log(b.x);//A的私有的可以继承  
//b.getX();//公有的不能继承 Error

3、混合继承模式：原型和call同时使用

**function** B() {  
 A.call(**this**);  
}  
B.prototype = **new** A;  
B.prototype.constructor = B;  
//会存在A的私有的在B的私有中存在，也在B的公有中存在  
//一般项目中不需要解决这个问题，如果你要解决如何的处理  
// var a = new A;  
// for (var key in B.prototype) {  
// if (a.hasOwnProperty(key)) {  
// delete B.prototype[key];  
// }  
// }  
**var** b = **new** B;  
console.dir(b);

4、和原型链正好相反，把A的私有和公有的都变为B的私有的

**function** B() {  
 **this**.method = **new** A;  
 **for** (**var** key **in this**.method) {  
 **this**[key] = **this**.method[key];  
 }  
 **delete this**.method;  
}  
**var** b = **new** B;  
console.dir(b);