\* 编程思想:

\* 面向过程:所有的事情都是亲力亲为,注重的是过程

\* 面向对象:提出需求,找对象,对象解决,注重的是结果

\* js不是一门面向对象的语言,是基于对象的语言,js来模拟面向对象

\* 面向对象的特性:封装,继承,多态,(抽象性)

\* 封装:就是包装,把一些重用的内容进行包装,在需要的时候,直接使用

\* 把一个值,存放在一个变量中,把一些重用的代码放在函数中,把好多相同功能的函数放在一个对象中,把好多功能的对象,放在一个文件中,把一些相同的内容放在一个对象中

\* 继承:类与类之间的关系,js中没有类的概念,js中有构造函数的概念,是可以有继承的,是基于原型

\* 多态:同一个行为,针对不同的对象,产生了不同的效果

创建对象三种方式:

1 字面量的方式

var per1={

name:"卡卡西",

age:20,

sex:"男",

eat:function () {

console.log("吃臭豆腐");

}

};

2 调用系统的构造函数

var per2=new Object();

per2.name="大蛇丸";

per2.age=30;

per2.sex="男";

per2.eat=function () {

console.log("吃榴莲");

};

per2.play=function () {

console.log("这个小蛇真好玩");

};

var dog=new Object();

3 自定义构造函数方式

function Person(name,age,sex) {

this.name=name;

this.age=age;

this.sex=sex;

this.play=function () {

console.log("天天打游戏");

};

}

var per=new Person("雏田",18,"女");

创建对象---->实例化一个对象,的同时对属性进行初始化: var per=new Person("小红",20);

\* 1.开辟空间存储对象

\* 2.把this设置为当前的对象

\* 3.设置属性和方法的值

\* 4.把this对象返回

工厂模式和自定义构造函数:



\*共同点:都是函数,都可以创建对象,都可以传入参数

\* 工厂模式:

\* 函数名是小写

\* 有new,

\* 有返回值

\* new之后的对象是当前的对象

\* 直接调用函数就可以创建对象

\*

\* 自定义构造函数:

\* 函数名是大写(首字母)

\* 没有new

\* 没有返回值

\* this是当前的对象

\* 通过new的方式来创建对象

\*

实例对象和构造函数之间的关系:

\* 1. 实例对象是通过构造函数来创建的---创建的过程叫实例化

\* 2.如何判断对象是不是这个数据类型?

\* 1) 通过构造器的方式 实例对象.构造器==构造函数名字 构造器：constructor

\* 2) 对象 instanceof 构造函数名字

\* 尽可能的使用第二种方式来识别,为什么?原型讲完再说

通过原型来添加方法,解决数据共享,节省内存空间

不需要共享的数据写在构造函数中,需要共享的数据写在原型中

Person.prototype.eat=function () {

console.log("吃凉菜");

};

原型?

\* 实例对象中有\_\_proto\_\_这个属性,叫原型,也是一个对象,这个属性是给浏览器使用,不是标准的属性----->\_\_proto\_\_----->可以叫原型对象

\* 构造函数中有prototype这个属性,叫原型,也是一个对象,这个属性是给程序员使用,是标准的属性------>prototype--->可以叫原型对象

\* 实例对象的\_\_proto\_\_和构造函数中的prototype相等--->true

\* 又因为实例对象是通过构造函数来创建的,构造函数中有原型对象prototype

\* 实例对象的\_\_proto\_\_指向了构造函数的原型对象prototype

\* 原型的作用:共享数据,节省内存空间

构造函数和实例对象和原型对象之间的关系：

\*构造函数可以实例化对象

\* 构造函数中有一个属性叫prototype,是构造函数的原型对象

\* 构造函数的原型对象(prototype)中有一个constructor构造器,这个构造器指向的就是自己所在的原型对象所在的构造函数

\* 实例对象的原型对象(\_\_proto\_\_)指向的是该构造函数的原型对象

\* 构造函数的原型对象(prototype)中的方法是可以被实例对象直接访问的

实例对象使用属性和方法层层的搜索

\* 实例对象使用的属性或者方法,先在实例中查找,找到了则直接使用,找不到则,去实例对象的\_\_proto\_\_指向的原型对象prototype中找,找到了则使用,找不到则报错。

页面加载后.这个自调用函数的代码就执行完了；一次性的函数--声明的同时,直接调用了

(function (形参) {

var num=10;//局部变量

})(实参);

如何把局部变量变成全局变量?

\*把局部变量给window就可以了

原型链:是一种关系,实例对象和原型对象之间的关系,关系是通过原型(\_\_proto\_\_)来联系的

如果原型指向改变了,那么就应该在原型改变指向之后添加原型方法

//因为JS是一门动态类型的语言,对象没有什么,只要点了,那么这个对象就有了这个东西,没有这个属性,只要对象.属性名字,对象就有这个属性了,但是,该属性没有赋值,所以,结果是:undefined

通过实例对象能否改变原型对象中的属性值?不能

就想改变原型对象中属性的值,怎么办?直接通过原型对象.属性=值;可以改变

var divObj=document.getElementById("dv");

console.dir(divObj);

//divObj.\_\_proto\_\_---->HTMLDivElement.prototype的\_\_proto\_\_--->HTMLElement.prototype的\_\_proto\_\_---->Element.prototype的\_\_proto\_\_---->Node.prototype的\_\_proto\_\_---->EventTarget.prototype的\_\_proto\_\_---->Object.prototype没有\_\_proto\_\_,所以,Object.prototype中的\_\_proto\_\_是null

面向对象编程思想:根据需求,分析对象,找到对象有什么特征和行为,通过代码的方式来实现需求,要想实现这个需求,就要创建对象,要想创建对象,就应该显示有构造函数,然后通过构造函数来创建对象.,通过对象调用属性和方法来实现相应的功能及需求,即可

\* 首先JS不是一门面向对象的语言,JS是一门基于对象的语言,那么为什么学习js还要学习面向对象,因为面向对象的思想适合于人的想法,编程起来会更加的方便,及后期的维护....

\* 面向对象的编程语言中有类(class)的概念(也是一种特殊的数据类型),但是JS不是面向对象的语言,所以,JS中没有类(class),但是JS可以模拟面向对象的思想编程,JS中会通过构造函数来模拟类的概念(class)

面向对象的特性:封装,继承,多态

\*

\* 封装:就是包装

\* 一个值存储在一个变量中--封装

\* 一坨重复代码放在一个函数中--封装

\* 一系列的属性放在一个对象中--封装

\* 一些功能类似的函数(方法)放在一个对象中--封装

\* 好多相类似的对象放在一个js文件中---封装

\*

\* 继承: 首先继承是一种关系,类(class)与类之间的关系,JS中没有类,但是可以通过构造函数模拟类,然后通过原型来实现继承

\* 继承也是为了数据共享,js中的继承也是为了实现数据共享

\*

\* 原型作用之一:数据共享,节省内存空间

\* 原型作用之二:为了实现继承 目的是:为了节省内存空间

\*多态:一个对象有不同的行为,或者是同一个行为针对不同的对象,产生不同的结果,要想有多态,就要先有继承,js中可以模拟多态,但是不会去使用,也不会模拟。

js中通过原型来实现继承:（改变原型指向）

//改变学生的原型的指向即可==========>学生和人已经发生关系

Student.prototype=new Person("小明",10,"男");//继承的

Student.prototype.study=function () {//自己的

console.log("学习很累很累的哦.");

};

//相同的代码太多,造成了代码的冗余(重复的代码)

var stu=new Student(100);

借用构造函数继承:

为了数据共享,改变原型指向,做到了继承---通过改变原型指向实现的继承

\*缺陷:因为改变原型指向的同时实现继承,直接初始化了属性，继承过来的属性的值都是一样的了,所以,这就是问题

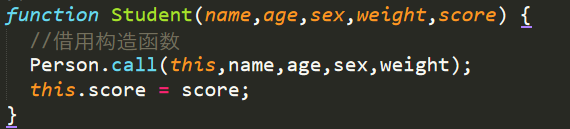
\*只能重新调用对象的属性进行重新赋值,

解决方案:继承的时候,不用改变原型的指向,直接调用父级的构造函数的方式来为属性赋值就可以了------借用构造函数:把要继承的父级的构造函数拿过来,使用一下就可以了

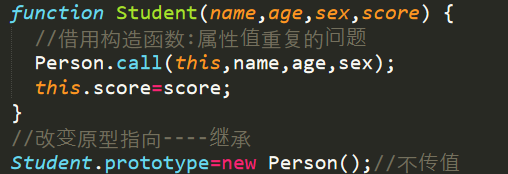
\*借用构造函数: 构造函数名字.call(当前对象,属性,属性,属性....);

\*解决了属性继承,并且值不重复的问题

\*缺陷:父级类别中的方法不能继承



组合继承:



拷贝继承；把一个对象中的属性或者方法直接复制到另一个对象中

var obj2={};

for(var key in obj1){

obj2[key]=obj1[key];

}

//改变了地址的指向

var obj2=obj1;

console.log(obj2.name,obj2.age);

obj2.sleep();

//面向对象特性:封装,继承,多态

//继承,类与类之间的关系,面向对象的语言的继承是为了多态服务的,

//js不是面向对象的语言,但是可以模拟面向对象.模拟继承.为了节省内存空间

//继承:

/\*

\* 原型作用: 数据共享 ，目的是:为了节省内存空间,

\* 原型作用: 继承 目的是:为了节省内存空间

\*

\* 原型继承:改变原型的指向

\* 借用构造函数继承:主要解决属性的问题

\* 组合继承:原型继承+借用构造函数继承

既能解决属性问题,又能解决方法问题

\* 拷贝继承:就是把对象中需要共享的属性或者犯法,直接遍历的方式复制到另一个对象中

函数的角色:

\*函数的声明

function f1() {

console.log("我是函数");

}

f1();

\*函数表达式

var ff=function () {

console.log("我也是一个函数");

};

ff();

函数声明如果放在if-else的语句中,在IE8的浏览器中会出现问题

以后宁愿用函数表达式,都不用函数声明

函数中的this的指向:

\* 普通函数中的this是谁?-----window

\* 对象.方法中的this是谁?----当前的实例对象

\* 定时器方法中的this是谁?----window

\* 构造函数中的this是谁?-----实例对象

\* 原型对象方法中的this是谁?---实例对象

严格模式:

"use strict";//严格模式

function f1() {

console.log(this);//window

}

f1();

//BOM:中顶级对象是window,浏览器中所有的东西都是window的

函数的不同的调用方式

\*普通函数

function f1() {

console.log("文能提笔控萝莉");

}

f1();

\*构造函数---通过new 来调用,创建对象

function F1() {

console.log("我是构造函数,我骄傲");

}

var f=new F1();

//对象的方法

function Person() {

this.play=function () {

console.log("玩代码");

};

}

var per=new Person();

per.play();

函数是对象,对象不一定是函数

对象中有\_\_proto\_\_原型,是对象

函数中有prototype原型,是对象

如果一个东西里面有prototype，又有\_\_proto\_\_,说明是函数,也是对象

所有的函数实际上都是Function的构造函数创建出来的实例对象

f1.\_\_proto\_\_==Function.prototype

apply和call的使用：

\* apply的使用语法

\* 函数名字.apply(对象,[参数1,参数2,...]);

\* 方法名字.apply(对象,[参数1,参数2,...]);

\* call的使用语法

\* 函数名字.call(对象,参数1,参数2,...);

\* 方法名字.call(对象,参数1,参数2,...);

\*

\* 作用:改变this的指向

\* 不同的地方:参数传递的方式是不一样的

\*

\* 只要是想使用别的对象的方法,并且希望这个方法是当前对象的,那么就可以使用apply或者是call的方法改变this的指向

\*

\*apply和call方法实际上并不在函数这个实例对象中,而是在Function的prototype中

\*apply和call是调用的时候改变this指向

bind方法:

\*是赋值一份的时候,改变了this的指向

使用的语法:

\* 函数名字.bind(对象,参数1,参数2,...);---->返回值是复制之后的这个函数

\* 方法名字.bind(对象,参数1,参数2,...);---->返回值是复制之后的这个方法

var ff=f1.bind(null);

ff(10,20);

var ff = f1.bind(per, 10, 20);

ff();

函数中几个成员:

\*函数中有一个name属性----->函数的名字,name属性是只读的,不能修改

\*函数中有一个arguments属性--->实参的个数

\*函数中有一个length属性---->函数定义的时候形参的个数

\*函数中有一个caller属性---->调用(f1函数在f2函数中调用的,所以,此时调用者就是f2)

函数作为参数的时候,如果是命名函数,那么只传入命名函数的名字,没有括号

\*排序---函数作为参数使用,匿名函数作为sort方法的参数使用,那么此时的匿名函数中有两个参数,

arr.sort(function (obj1,obj2) {//升序；从小到大

if(obj1>obj2){

return 1;

}else if(obj1==obj2){

return 0;

}else{

return -1;

}

});

arr.sort(function (obj1,obj2) {//降序 从大到小

if(obj1>obj2){

return -1;

}else if(obj1==obj2){

return 0;

}else{

return 1;

}

});

console.log(typeof num);//获取num这个变量的数据类型

console.log(obj instanceof Object);/判断这个对象是不是某个类型的

Object.prototype.toString.call(对象);//获取某个对象的数据类型的样子 得到的是字符串

[10,20,30] "[object Array]"

作用域：变量---->局部变量和全局变量,

\*作用域:就是变量的使用范围

\*局部作用域和全局作用域

\*js中没有块级作用域---一对括号中定义的变量,这个变量可以在大括号外面使用

\*函数中定义的变量是局部变量

作用域链:

变量的使用,从里向外,层层的搜索,搜索到了就可以直接使用了

层层搜索,搜索到0级作用域的时候,如果还是没有找到这个变量,结果就是报错

预解析:就是在浏览器解析代码之前,把变量的声明和函数的声明提前(提升)到该作用域的最上面

闭包：

\* 闭包的概念:函数A中,有一个函数B,函数B中可以访问函数A中定义的变量或者是数据,此时形成了闭包(这句话暂时不严谨)

\* 闭包的模式:函数模式的闭包,对象模式的闭包

\* 闭包的作用:缓存数据,延长作用域链

\* 闭包的优点和缺点:缓存数据，没有及时的释放

\* 总结:如果想要缓存数据,就把这个数据放在外层的函数和里层的函数的中间位置

\* 局部变量是在函数中,函数使用结束后,局部变量就会被自动的释放

\* 闭包后,里面的局部变量的使用作用域链就会被延长

沙箱:环境,黑盒,在一个虚拟的环境中模拟真实世界,做实验,实验结果和真实世界的结果是一样,但是不会影响真实世界。如自调用函数(function(){...})(); (function(){....}())

递归:函数中调用函数自己,此时就是递归,递归一定要有结束的条件

浅拷贝:拷贝就是复制,就相当于把一个对象中的所有的内容,复制一份给另一个对象,直接复制,或者说,就是把一个对象的地址给了另一个对象,他们指向相同,两个对象之间有共同的属性或者方法,都可以使用。

深拷贝:拷贝还是复制,深:把一个对象中所有的属性或者方法,一个一个的找到.并且在另一个对象中开辟相应的空间,一个一个的存储到另一个对象中。

\* 正则表达式的作用:匹配字符串的

\* 在大多数编程语言中都可以使用

\* 正则表达式的组成:是由元字符或者是限定符组成的一个式子

元字符:

\* . 表示的是:除了\n以外的任意的一个字符

\* [] 表示的是:范围, [0-9] 表示的是0到9之间的任意的一个数字

\* [1-7] 表示的是1到7之间的任意的一个数字

\* [a-z] 表示的是:所有的小写的字母中的任意的一个

\* [A-Z] 表示的是:所有的大写的字母中的任意的一个

\* [a-zA-Z] 表示的是:所有的字母的任意的一个

\* [0-9a-zA-Z] 表示的是: 所有的数字或者是字母中的一个

\* [] 另一个含义: 把正则表达式中元字符的意义干掉 [.] 就是一个.

\* | 或者 [0-9]|[a-z] 表示的是要么是一个数字,要么是一个小写的字母

\* () 分组 提升优先级 [0-9]|([a-z])|[A-Z]

\* ([0-9])([1-5])([a-z]) 三组, 从最左边开始计算

\* (()(())) 4组

都是元字符,但是也可以叫限定符,下面的这些

\* \* 表示的是:前面的表达式出现了0次到多次

\* [a-z][0-9]\* 小写字母中的任意一个 后面是要么是没有数字的,要么是多个数字的

\* "fdsfs3223323" [a-z][0-9]\*

\*

\* + 表示的是:前面的表达式出现了1次到多次

\* [a-z][9]+ 小写字母一个后面最少一个9,或者多个9

\* "fesfewww9fefds"

\*

\* ? 表示的是:前面的表达式出现了0次到1次,最少是0次,最多1次 ,另一个含义:阻止贪婪模式

\* [4][a-z]? "1231234ij"

\* 限定符:限定前面的表达式出现的次数

\* {} 更加的明确前面的表达式出现的次数

\* {0,} 表示的是前面的表达式出现了0次到多次,和 \*一样的

\* {1,} 表示的是前面的表达式出现了1次到多次,和 +一样的

\* {0,1} 表示的是前面的表达式出现了0次到1次,和 ?一样的

\* {5,10} 表示的是前面的表达式出现了5次到10次

\* {4} 前面的表达式出现了4次

\* {,10} 错误的========不能这么写

\* ^ 表示的是以什么开始,或者是取非(取反)

\* ^[0-9] 以数字开头

\* ^[a-z] 以小写字母开始

\* [^0-9] 取反,非数字

\* [^a-z] 非小写字母

\* [^0-9a-zA-Z\_]

\* $ 表示的是以什么结束 [0-9][a-z]$ 必须以小写字母结束

\* ^[0-9][a-z]$ 相当于是严格模式 "3f2432e"(错) "4f"

\* \d 数字中的任意一个 [0-9]

\* \D 非数字中的一个 [^0-9]

\* \s 空白符中的一个

\* \S 非空白符

\* \w 非特殊符号 [0-9a-zA-Z\_]

\* \W 特殊符号 [^0-9a-zA-Z\_]

\* \b 单词的边界

\* "what are you no sha lei"

\*

\* . 除了\n以外的任意一个单个字符

\* [] 范围

\* () 分组,提升优先级

\* | 或者

\* \* 0-多次

\* + 1-多次

\* ? 0-1次

\* {0,} 和\*一样

\* {1,} 和+

\* {0,1} 和?

\*

\* \d 数字中的一个

\* \D 非数字

\* \s 空白符

\* \S 非空白符

\* \W 特殊符号

\* \w 非特殊符号 \_

\* ^ 取反,以什么开始

\* $ 以什么结束

\*

\* \b 单词边界

邮箱的正则表达式：

[[0-9a-zA-Z\_.-]+[@][0-9a-zA-Z\_.-]+([.][a-zA-Z]+){1,2}](mailto:[0-9a-zA-Z_.-]+[@][0-9a-zA-Z_.-]+([.][a-zA-Z]+){1,2})

手机：/^\d{11}$/

座机号码：/^\d{3,4}[-]\d{7,8}$/

中文名字：/^[\u4e00-\u9fa5]{2,6}$/

创建正则表达式对象:

1、通过构造函数创建对象：

var reg=new RegExp(/\d{5}/);

var str="我的电话是10086";

var flag=reg.test(str); //调用方法验证字符串是否匹配

console.log(flag);//true

2、字面量的方式创建正则表达式对象:

var reg=/\d{1,5}/;

var flag=reg.test("小苏的幸运数字:888");

console.log(flag);//true

正则表达式中:g 表示的是全局模式匹配

正则表达式中:i 表示的是忽略大小写

识别正则表达式是否匹配：/[a-zA-Z]+/.test("hello")

通过正则表达式匹配这个字符串

var str = "中国移动:10086,中国联通:10010,中国电信:10000";

var reg=/\d{5}/g;

//通过正则表达式匹配这个字符串

var array=reg.exec(str);

console.log(array);

//["10086", index: 5, input: "中国移动:10086,中国联通:10010,中国电信:10000", groups: undefined]

while (array!=null){

//输出匹配的内容

console.log(array[0]);

array=reg.exec(str);

}

把里面所有的数字全部显示出来

var str='12366abc32155nn22hh'

var array=str.match(/\d{5}/g);

console.log(array);//["12366", "32155"]

//伪数组和数组的区别

//真数组的长度是可变的

//伪数组的长度不可变

//真数组可以使用数组中的方法

//伪数组不可以使用数组中的方法