instanceof运算符

验证原型对象与实例对象之间的关系。

alert(cat1 instanceof Cat); //true

 isPrototypeOf()

这个方法用来判断，某个proptotype对象和某个实例之间的关系。

alert(Cat.prototype.isPrototypeOf(cat1)); //true

hasOwnProperty()

每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

alert(cat1.hasOwnProperty("name")); // true

 in运算符

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

alert("name" in cat1); // true

in运算符还可以用来遍历某个对象的所有属性。

for(var prop in cat1) {

alert("cat1["+prop+"]="+cat1[prop]);

}

## 原型

作用之一:数据共享,为了节省内存空间  
方法被共享,方法就添加到原型(prototype)中

### 案例

自定义构造函数  
 function ChangeColor(btnId, divId, color) {  
 //设置属性  
 this.btnObj = document.getElementById(btnId);//按钮对象  
 this.divObj = document.getElementById(divId);//div对象  
 this.color = color;//颜色存储  
 }  
 通过原型添加方法  
 ChangeColor.prototype.setBackgroundColor = function () {

var that = this;

//把this这个实例对象存储到that这个变量中  
 this.btnObj.onclick = function () { // 事件处理函数  
 //设置div的背景颜色  
 that.divObj.style.backgroundColor = that.color;  
 };  
 };  
 //实例化对象  
 var cc = new ChangeColor("btn", "dv", "green");  
 //对象调用方法  
 cc.setBackgroundColor();

### 想要改变一组属性的时候

function ChangeStyle(btnObj, divObj, json) {  
 this.btnObj = btnObj;//按钮  
 this.divObj = divObj;//div  
 this.json = json;//json格数的数据对象  
  
}  
//通过原型添加方法  
ChangeStyle.prototype.changeAll = function () {  
 //点击按钮,设置div的各种样式属性值  
 var that = this;//把this存储到that  
 //为按钮注册点击事件-设置div的样式  
 this.btnObj.onclick = function () {  
 for (var key in that.json) {  
 that.divObj.style[key] = that.json[key];  
 }  
 };  
};  
var btnObj = document.getElementById("btn");//按钮  
var divObj = document.getElementById("dv");//div  
var json1 = { //改变后的属性  
 "backgroundColor": "green",  
 "width": "500px",  
 "height": "300px",  
 "border": "2px solid red"  
};  
//实例化对象  
var cs = new ChangeStyle(btnObj, divObj, json1);  
//调用方法  
cs.changeAll();

### 通过原型添加方法、属性

简单的原型语法：

方便添加属性和方法,但是这么做,缺少构造器,需要手动添加构造器

function Person() {  
}  
Person.prototype= {  
 constructor:Person,  
 name:'xiaohei',  
 age:18,  
 sex:'nan',  
 say: function () {  
 console.log('你好呀')  
 }  
};  
var per=new Person();  
console.log(per.age);  
console.log(per.name);  
per.say();

## 重名问题

· 实例对象中的属性和原型对象中属性重名了,使用的是实例对象中的属性  
· 如果实例对象中没有这个属性,但是原型对象中有这个属性,结果用就是原型对中的这个属性  
· 先在实例对象中找属性.找到了就用,找不到就去原型对象中找属性,找到了就用,找不到,结果就是undefined  
· 因为js是一门动态类型的语言,对象中如果没有这个属性,如果点了,那么这个对象就有这个属性了,如果没有赋值,结果就是undefined,  
· 实例对象和原型对象中方法如果重名了,用的是实例对象的,如果实例对象中没有这个方法,去原型对象中找,没有找到,就报错

补充： num=100; 全局变量----隐式全局变量

## 实现继承的方式

继承的目的:为了实现多态,由于js不是一门面向对象的语言,所以,js模拟面向对象的编程思想,可以有封装,可以有继承,但是js中的继承不是为了多态,是为了数据共享  
  
继承的方式:  
1. 改变原型指向实现继承,属性可以被继承,方法也可以被继承,但是,属性之会相同  
2. 借用构造函数实现继承,属性可以被继承,方法不可以被继承.  
3. 组合继承,属性可以继承,方法也可以继承,而且,属性值不相同

### 1.通过改变子类prototype指向实现

方法：改变子类prototype的指向父类的构造函数的实例

子类.prototype = new 父类();

缺点 子类的constructor指向改变(指向了父类)需要手动设置

所有的对象的属性之都是一样,还要自己手动的修改

子类的实例对象的constructor默认调用prototype对象的constructor也指向父类

优化:子类.prototype.constuctor = 子类;

例

//父类构造函数

function Person(name) {

this.name = name ;

}

Person.prototype.live = function () {

console.log("Per - - Live");

};

//子类

function Student(age) {

this.age = age ;

}

//需要赋值，父类函数的形参

Student.prototype = new Person("Leon");

var stu = new Student(15);

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

stu.live();

### 2.借用构造函数实现继承

方法：子类构造函数中绑定父类的构造函数

父类.call(this,属性名,属性名);

缺点：不能绑定父类的prototype对象上的方法

例

function Person(name) {

this.name = name ;

}

Person.prototype.eat = function () {

console.log('Person - - eat');

};

function Student(age,name) {

Person.call(this,name);

//借用了Person的构造函数,借用这个函数中的属性了

this.age = age ;

}

var stu = new Student(18,"Leon");

console.log(stu.name); //Leon

console.log(stu.age); //18

stu.eat(); //error 方法没有继承

### 3.组合继承

改变原型指向实现继承+借用构造函数继承

//创建父类

function Person (name,age ){

this.name = name ;

this.age = age ;

}

//为父类添加方法

Person.prototype.sayHi = function () {

console.log("Person -- Hi!");

};

//子类

function Teacher (name,age,gender){

Person.call(this,name,age); //借用构造函数实现继承，属性的继承

this.gender = gender ;

}

// 改变原型指向实现继承,主要是方法的继承

Teacher.prototype = new Person(); //括号里不用赋值

//实例化teacher对象

var tea = new Teacher("Leon",10,"man");

tea.sayHi();

console.log(tea.name,tea.age,tea.gender);

4.直接继承

方法：将父类的prototype对象赋值给子类的prototype

子类.prototype = 父类.prototype;

缺点:Animal的prototype 的constructor的指向也改变了

例

//动物类

function Animal(){}

//为动物类添加方法

Animal.prototype.kind = "fly";

Animal.prototype.run = function () {

console.log('gogogo!!');

};

//猫类

function Cat() {}

//---改变猫类prototype的指向父类的prototype

Cat.prototype = Animal.prototype;

//实例化猫类

var cat = new Cat();

console.log(cat.kind);

cat.run();

5.利用空对象作为中介

改进直接继承方式

优点不会改变父类的prototype的constructor的指向

例

//动物类

function Animal() {}

Animal.prototype.kind = "From Animal---small";

Animal.prototype.color = "From Animal---blue";

Animal.prototype.run = function () {

console.log('gogogo!!');

};

//中介对象

function obj(){}

//猫类

function Cat(){}

//将Animal的prototype传给obj.prototype对象

obj.prototype = Animal.prototype;

//通过实例化obj中介对象，赋值给子类的prototype对象

Cat.prototype = new obj();

//使Cat构造函数的prototype对象的constructor指向自己,Animal不会受影响

Cat.prototype.constructor=Cat;

//实例化Cat 调用父类的方法和属性。

var cat = new Cat();

console.log(cat.kind);

console.log(cat.color);

cat.run();

//Cat和Animal都不会受影响

console.log(Cat.prototype.constructor===Cat);

console.log(Animal.prototype.constructor===Animal);

console.dir(Cat);

console.dir(Animal);

6.拷贝继承

遍历将父类的prototype上的属性分别赋值给子类

例

function Person(name) {//父类

this.name = name ;

}

Person.prototype.live = function () {//父类的方法

console.log("per - - live");

};

function Student(age) {//子类

this.age = age ;

}

for(var key in Person.prototype){//循环遍历

Student[key] = Person.prototype[key];

}

原型

实例对象中有\_\_proto\_\_,是原型对象

构造函数中有prototype,是原型对象

对象中的\_\_proto\_\_指向的是该对象的构造函数的prototype

原型链------重点-----理解的方式

是一种关系,是实例对象的\_\_proto\_\_和原型对象prototype的关系

原型的指向是否可以改变------可以

很深的原型链

divObj.\_\_proto\_\_指向的是HTMLDivElement.prototype,

中\_\_proto\_\_指向的是HTMLElement.prototype

中的\_\_proto\_\_指向的是Element.prototype

中\_\_proto\_\_指向的是Node.prototype

中\_\_proto\_\_指向是EventTarget.prototype

中\_\_proto\_\_是Object.prototype

中\_\_proto\_\_--->null

面向对象的思想:

封装

继承

继承的方式

原型的作用

函数的角色和区别-----重点

所有的函数都是Function的实例对象

函数是对象,那么函数的\_\_proto\_\_指向的是Object构造函数的prototype

各种函数中的this----重点

首先分析this所在的函数是当做哪个对象的方法调用的，则该对象就是this所引用的对象。

作为构造函数调用。所谓构造函数，就是生成一个新的对象。这时，这个this就是指这个对象。

作为普通函数调用。这时，这个this指向window对象。

在事件处理函数中调用时，指向触发事件的目标。

函数也是对象

高阶函数

函数作为参数 callback

返回值 return

函数声明方式

函数声明 function fn(){};

表达式声明 var fn = function(){};

推荐使用表达式方式

if(true){

function f1() { console.log("乔峰真的很厉害"); }

}else{

function f1() { console.log("助教好厉害"); }

}

//在IE中输出第二个，IE会跳过解析if语句。第二个覆盖了第一个声明

var ff;

if(true){

ff=function () { console.log("您好"); };

}else{

ff=function () { console.log("我好"); };

}

ff();

## This

❀

普通函数中的this,是 window,如果是严格模式下,就是undefined

实例方法中的this,是 当前的实例对象

原型对象中的this,是 当前的实例对象

构造函数中的this,是 当前的实例对象(new)

构造函数直接调用 this是window

定时器中的this,是 window

事件中的this,是当前的对象

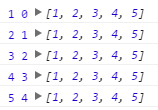
## forEach

var arr1 = [1,2,3,4,5];  
arr1.forEach(function (x,y,z) {  
 console.log(x,y,z);  
});

X:数组的每一项的值

Y:数组元素下标

Z:数组本人



forEach(数组名/字符串名，function(x){

//要执行的函数

});

## Call Apply

调用别人的函数或方法

函数名.apply()

对象.方法名.apply()

改变this指向

例：

Student 想借用Person的eat

Per.eat.apply(stu);// 把per的eat借给stu

此时this 指向Person

若原方法有形参 则在后面传参

.call(stu,‘女’,18);

原方法有ruturn时 要接收返回值

Apply 在传参的时候 写成数组形式

.apply (stu,[‘女’,18]);

## Bind

### 作用：

复制 并改变this指向

（复制的时候改变，所以复制语句的括号里要写要改变的对象）

### 使用语法：

Var 变量=需要改变的函数.bind(需要复制的对象);

Var 变量=对象.方法名.bind(需要复制的对象);

参数可以在复制的时候传，也可以在调用的时候传

此时的变量就代表了复制的那个函数 需要加括号才能调用

若传的值为null 则window

## 高阶函数

函数的比较高级的用法

函数普通用法,函数的调用

function f1(){

}

f1();

函数作为参数

function f1(fn){

fn();

}

f1(

function(){

});

函数作为返回值

function f2(){

return function (){

}

}

var ff=f2();

ff();

## 闭包

一个函数A中有一个函数B,函数B可以使用函数A中的数据(访问这个变量的值);

闭包的作用:缓存数据 缺点是数据不能及时释放

### 函数模式的闭包

function f1() {  
 var num = 10;  
 return function () {  
 console.log(num); 返回的是 函数 的形式  
 }  
}  
  
var ff = f1(10);  
ff();

### 对象模式的闭包

function f2() {  
 var num = 100;  
 return {  
 age: num 返回的是 对象：值 的形式  
 }  
}  
var obj = f2();  
console.log(obj.age);

## 沙箱 important

就是一个虚拟的环境， 不影响外面的世界

在一个自调用函数中写代码---自调用函数就是沙箱

作用：区分全局变量和局部变量，放沙箱里可避免别人声明的变量和自己的冲突

沙箱的写法：

①

自调用函数就是沙箱

（function（）{

Var num=10; // 外面接收不到

}）（）；

一般前面加一个 ; 避免出错

；（function（）{

需要执行的代码放进来

}）（）；

②

（functiom(){

}（））;

## 递归

函数A中调用函数A， 递归要有结束条件，否则形成死循环

一定要有结束条件，把要传递下去的值放在递归函数里

❀ 案例1：

//计算一个数字和他前面所有数字的累加和,  
//传入一个数字,计算这个数字的累加和  
//1+2+3+4+5------>5+4+3+2+1----- n+(n-1)  
function getSum(num) {  
 //结束的条件  
 if(num==1){  
 return 1;  
 }  
 return num+getSum(num-1);

（把还需要利用的值 放在函数里）  
 //return 5+ getSum(4);---->等待 5+10  
 //return 4+ getSum(3);---->等待-----> 4+6  
 //return 3+getSum(2);---->等待-------> 3+3  
 //return 2+1---->  
}  
  
var result=getSum(5);  
console.log(result);

❀ 案例2：

各位数字相加

function getEverySum(num) {  
 if(num<10){  
 return num;  
 }  
 //如果这个数字和10相除了结果还是大于10,那么就继续调用这个函数,  
 //此时这里,做的是:一个数字和10相除的结果+这个数字和10取余的结果  
 return parseInt(getEverySum(num/10))+parseInt(num%10);  
}  
console.log(getEverySum(1345));

这里的num/10 得到的值还要往下做进一步运算（进一步获取各个位数值）

❀ 案例3：

function getFib(num) {  
 if(num==1||num==2){  
 return 1;  
 }  
 //每次都是计算的是前一个月的数字值+前一个月的前一个月的数字值  
 return getFib(num-1)+getFib(num-2);  
}  
console.log(getFib(5));

## 正则表达式

### 常用元字符串

| **元字符** | **说明** |
| --- | --- |
| \d | 匹配数字 |
| \D | 匹配任意非数字的字符 |
| \w | 非特殊符号 匹配字母或数字或下划线 |
| \W | 特殊符号 匹配任意不是字母，数字，下划线 |
| \s | 匹配任意的空白符 |
| \S | 匹配任意不是空白符的字符 |
| . | 匹配除换行符 \n 以外的任意单个字符 |
| ^ | 表示匹配行首的文本(以谁开始) ^[0-9]以数字开头  取反 [^0-9] 非数字 |
| $ | 表示匹配行尾的文本(以谁结束) |

### 限定符

| **限定符** | **说明** |
| --- | --- |
| \* {0, } | 重复零次或更多次 （紧贴着的前面的） |
| + {1, } | 重复一次或更多次 （紧贴着的前面的） |
| ? {0,1} | 重复零次或一次 （紧贴着的前面的） |
| {n} | 重复n次 （紧贴着的前面的） |
| {n,} | 重复n次或更多次 |
| {n,m} | 重复n到m次 |

### 其他

[] 字符串用中括号括起来，表示匹配其中的任一字符，相当于或的意思

[^] 匹配除中括号以内的内容

\ 转义符

| 或者，选择两者中的一个。注意|将左右两边分为两部分，而不管左右两边有多长多乱

() 从两个直接量中选择一个，分组 提升优先级

例子：

gr(a|e)y匹配gray和grey

[\u4e00-\u9fa5] 匹配汉字

/^ $/ 表示严格模式

[3-7]{2,4}

必须有3-7连续出现两到四次 ‘7a6b5c2d’是错的，因为不连续

## 正则表达式的使用

### 验证

方法一：调用系统对象创建正则表达式

var str='哈哈哈哈。886';

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

var result=reg.test(str);

console.log(result); // 返回布尔值  
  
方法二：字面量的方式使用正则

var str2='abcd123efg';

var reg2=/[a-zA-Z]{2}/;

var result2=reg2.test(str2);

console.log(result2);

简写： /正则表达式/.test(字符串)

/^ $/ 表示严格模式

### 提取

/..../g g指的是全部的符合条件的内容 否则只选中第一个

/..../i i指的是忽略大小写

Str.match() 从字符串中提取符合条件的字符，括号内放条件

### 分组提取

前提是 必须用（）分组

提取邮件中的每一部分  
 var reg = /(\w+)@(\w+)\.(\w+)(\.\w+)?/;  
 var str = ["123123@xx.com";](mailto:\"123123@xx.com\";)

if (reg.test(str)) {  
 console.log(RegExp.$1);  
 console.log(RegExp.$2);  
 console.log(RegExp.$3);

需要分组的括起来， $后面的数字表示第几个括号

### 替换

str.replace(查找条件，替换成什么字符)

var str="HhPp";  
str=str.replace(/[a-z]/gi,"Y");  
console.log(str);