面向对象

**一种编程思想**

抽象：提取相同特性的代码，封装成一个函数（类）

**特征**：封装，抽象，继承

function Person(name,age){

this.name = name;

this.age = age;

this.say = function(){

alert(this.name);

}

}

var p2 = new Person('小明',12);

**不写new**： this指向，window必须加括号，没有return 默认return 未定义

**写new**： this指向构造函数（这个对象），可以不加括号，括号是拿来传参的，如果return的为简单类型的数据那么返回的就为这个构造函数

如果return的为复合类型的数据那么返回的就为return后面的这个对象

**Prototype：原型**

原型是所有函数的都有的一个属性，原型的作用是给这个函数的每一个实例化对象存储方法用的。

**\_\_proto\_\_** **原型链** (隐式链接)

实例与其构造函数的原型的桥梁

**原型和原型链之间的联系**

function 构造函数(){

this.name='abc';

this.age='18';

this.say=function(){

console.log('这是构造函数内部的方法')

}

}

构造函数.prototype.say=function(){

console.log('这是构造函数 原型上的方法')

}

Object.prototype.say=function(){

console.log('这是Object上的方法')

}

Function.prototype.say=function(){

console.log('这是Function上的方法')

}

var 实例=new 构造函数()

实例.say();

构造函数.say()

实例 找 say方法的过程

1.看实例的构造函数内部是否有，有就用。

2.如果构造函数上没有，就去构造函数的原型上去找，有就用。

3.如果构造函数的原型上没有，就通过 构造函数的原型的原型链去找，就是到Object的原型上去找，有就用。

console.log(构造函数.prototype.\_\_proto\_\_==Object.prototype); // true

4.如果Object的原型上还没有，就去Object的原型的原型的原型链去找，找到的是 null； 结束了

console.log(Object.prototype.\_\_proto\_\_==null); // true

函数 找 say 方法的过程

函数不能调用自身和自身原型上的方法时，也就是说构造函数自身的方法和构造函数的原型是给构造函数的实例化对象用的。

1.通过函数的原型链 去找 它的构造函数(Function)的原型上去找，有就用。

2.如果Funtion的原型上没有，就通过Funtion的原型的原型链去找，就是就是到Object的原型上去找，有就用。

console.log(Funtion.prototype.\_\_proto\_\_==Object.prototype); // true

3.如果Object的原型上还没有，就去Object的原型的原型的原型链去找，找到的是 null； 结束了

console.log(Object.prototype.\_\_proto\_\_==null); //true

注意：1.普通对象没有prototype属性，只有函数对象才有prototype属性。

2.不论是普通对象还是函数对象，都是先通过原型链（\_\_proto\_\_）去找其构造函数的prototype，然后再通过其构造函数的prototypede 的\_\_proto\_\_

找到Object的prototype，然后再通过 Object的prototype的\_\_proto\_\_找到null。

3.所有函数的原型的原型链都与Object.prototype相等，所有函数的原型链都与其构造函数的原型相等。

4.Function是所有构造函数的最高级，其自身构造自身，所以Function. Prototype和Function.\_\_proto\_\_相等。