**js原型和原型链你只要看这一篇**

**一、什么是原型：**

任何对象都有一个原型对象，这个原型对象由对象的内置属性\_proto\_指向它的构造函数的prototype指向的对象，即任何对象都是由一个构造函数创建的，但是不是每一个对象都有prototype，只有方法才有prototype。

function Person() {

}

var p = new Person();

//方法才有prototype,普通对象无prototype

console.log(Person.prototype); // Object{}

console.log(p.prototype); // undifined

//任何对象都是有构造函数的,Person这种方法的构造函数是Function。

//备注:constructor很容易被改变，一般不用它，此处只是打印下列对象的构造//函数是什么。

console.log(p.constructor); //function Person(){}

console.log(Person.constructor); //function Function(){}

console.log({}.constructor); // function Object(){}

console.log(Object.constructor); // function Function() {}

console.log([].constructor); //function Array(){}

**那什么是构造函数呢？**

  用function声明的都是函数，而如果直接调用的话，那么Person()就是一个普通函数，只有用函数new产生对象时，这个函数才是new出来对象的构造函数。

**二、创建对象的过程（采用new关键）**

**2.1 、声明方法的过程**

  首先，当我们声明一个function关键字的方法时，会为这个方法添加一个prototype属性，指向默认的原型对象，并且此prototype的constructor属性也指向方法对象。此二个属性会在创建对象时被对象的属性引用。

function Hello() {

}

console.log(Hello.prototype);

// Object {} -- > 内部的constructor 指向Hello方法

console.log(Hello.prototype.constructor); // function Hello(){}

**2.2、我们如果用Hello创建一个对象h，看这个对象有什么属性。**

console.log(h.constructor); // function Hello(){}

console.log(Object.getPrototypeOf(h)==Hello.prototype); // true

备注：getPrototypeOf是获取\_proto\_

  我们惊喜的发现，new出来的对象，它的constructor指向了方法对象，它的\_proto\_和prototype相等。

  即new一个对象，它的\_proto\_属性指向了方法的prototype属性，并且constructor指向了prototype的constructor属性。

**2.3 、创建一个对象的过程**

function Hehe(name) {

this.name = name;

}

var h = new Hehe("笑你妹");

//伪代码:

function newObj(name){

var obj = {};

obj.\_\_proto\_\_ = Hehe.prototype;

obj.constructor = Hehe.prototype.constructor;

var result = Hehe.call(obj, name);

return typeof result==='object'&& result!=null ? result : obj;

//当无返回对象或默认时返回obj。

}

var hh = newObj("笑你妹");

console.log(hh);

console.log(h);

//虽然hh!=h,但是可以看到这个hh就和h的结构一样了。

过程：先创建一个空对象，设置一个\_proto\_指向方法的原型，设置constructor，用新对象做this指向方法，返回新对象。

**2.4、延伸**

  从上面说明的过程中，我们发现只要是对象就是有构造函数来创建的，并且内部二个属性是从构造函数的prototype衍生的一个指向，而构造函数的prototype也是一个对象，那么它应该肯定也有一个构造函数，首先它是一个Object {} 对象，那么它的构造函数肯定是Object,所以就会有一个指针\_proto\_指向Object.prototype。最后Object.prototype因为没有\_proto\_，指向null，这样就构成了一个原型链。



**三、原型链**

**(1)什么是原型链？**

原型链的核心就是依赖对象的\_proto\_的指向，当自身不存在的属性时，就一层层的扒出创建对象的构造函数，直至到Object时，就没有\_proto\_指向了。

**(2)如何分析原型链?**

    因为\_proto\_实质找的是prototype，所以我们只要找这个链条上的构造函数的prototype。其中Object.prototype是没有\_proto\_属性的，它==null。

**3.1、最简单的原型链分析**

function Person(name){

this.name = name;

}

var p = new Person();

//p ---> Person.prototype --->Object.prototype---->null

属性搜索原则：  
1.当访问一个对象的成员的时候，会现在自身找有没有,如果找到直接使用。  
2.如果没有找到，则去**原型链指向的对象的构造函数的prototype**中找，找到直接使用，没找到就返回undifined或报错。

**3.2、原型链上属性查询的案例**

     //略。。。

**3.3、原型继承**

//原型继承的基本案例

function Person(name, age) {

this.name = name;

this.age = age;

}

//1.直接替换原型对象

var parent = {

sayHello : function() {

console.log("方式1：替换原型对象");

}

}

Person.prototype = parent;

var p = new Person("张三", 50);

p.sayHello();

//2.混入式原型继承

console.log(".............混入式原型继承..............");

function Student(name, age) {

this.name = name;

this.age = age;

}

var parent2 = {

sayHello : function() {

console.log("方式2：原型继承之混入式加载成员");

}

}

for ( var k in parent2) {

Student.prototype[k] = parent2[k];

}

var p = new Student("张三", 50);

p.sayHello();