$().ready(function(){

//1 一切都是对象 一切（引用类型）都是对象，对象是**属性的集合**。

var obj = {

a:1,

b:function(){

alert(10);

},

c:{

name : "x",

age : "man"

}

}

alert(obj.a);

obj.b();

alert(obj.c.age);

function f(){

alert(111);

}

f.a=function(){

alert("aaaa");

}

f.a();

function f(){}

alert(f instanceof Object)

//2 对象都是通过函数来创建的。

function f(){

name="scott",

age="22"

}

var obj = new f();

alert(typeof(f) +"---"+typeof(obj));//function object

var a =new Object();

alert(typeof Object);//function

alert(typeof a);//object

a.b=1;

alert(a.b);//1

// 3 每个函数function都有一个prototype，即原型。这里再加一句话——每个对象都有一个\_\_proto\_\_，可成为隐式原型。

function Fn(){}

Fn.prototype.name = "徐";

Fn.prototype.age="22";

var f = new Fn();

alert(f.age);

alert(f.name);

alert(Fn.prototype);//[object Object]

//**4 函数有原型，对象有隐式原型。两者相同： fn.prototype==obj.\_\_proto\_\_ 函数原型的constructor指向本身**

**对象的\_\_proto\_\_指向的是创建它的函数的prototype**

function fn(){

}

console.log("fn.prototype "+typeof fn.prototype);

console.log(fn.prototype);

fn.a=function (){

console.log("aaaa");

}

fn.a();

fn.prototype.b=function(){

console.log("bbbb");

}

var obj=new fn();

console.log("函数有原型，对象有隐式原型。两者相同： fn.prototype==obj.\_\_proto\_\_ "+ (fn.prototype==obj.\_\_proto\_\_));//true

obj.b();

console.log(obj.\_\_proto\_\_);

console.log("函数原型的constructor指向本身：")

console.log(fn.prototype.constructor.a);

**Object是函数function (对象是函数)**

**函数的原型（对象）的隐式原型指向Object的原型 fn.prototype.\_\_proto\_\_ = Object.prototype**

**Object的原型的隐式原型为空 Object.prototype.\_\_proto\_\_ = null**

function fn(){};

var obj = new fn();

console.log(obj instanceof fn);//true

console.log(fn.prototype);

console.log(fn.prototype == obj.\_\_proto\_\_);//对象的隐式原型等于创建它的函数的原型

var Fn = new Function();

// console.log(typeof Fn);//function

var OBJ = new Object();

// console.log(typeof OBJ);//object

console.log(Fn.prototype);

console.log(OBJ.\_\_proto\_\_ == Fn.prototype);//false OBJ不是Fn创建

console.log(typeof Function);//function

console.log(typeof Object);//function Object是函数

console.log(Function.prototype);

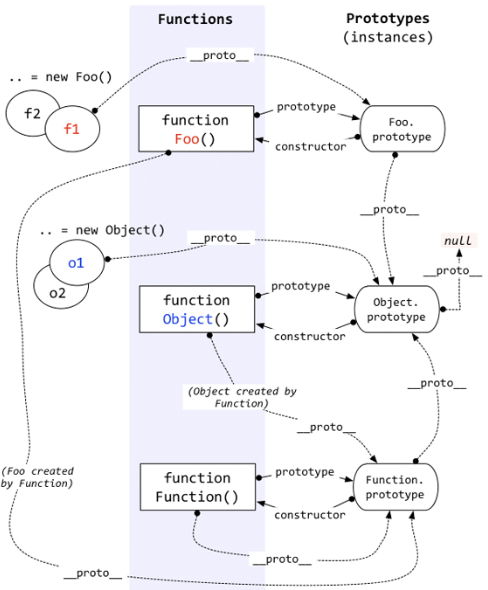
console.log(Object.prototype);

console.log("Object的原型的隐式原型为空 Object.prototype.\_\_proto\_\_ = null");

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

console.log("函数的原型（对象）的隐式原型指向Object的原型 fn.prototype.\_\_proto\_\_ = Object.prototype");

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



访问一个对象的属性时，先在基本属性中查找，如果没有，再沿着\_\_proto\_\_这条链向上找，这就是原型链。

var fn = function(){};

console.log(fn.prototype);

fn.prototype.o=222;

fn.a=1;

var f = new fn();

console.log(f.o); //f.\_\_proto\_\_指向的是fn.prototype

f.b=111;

for(item in f){

if(f.hasOwnProperty(item)){

console.log(item);//b

}

}

变量、函数表达式——变量声明，默认赋值为undefined；

this——赋值；

函数声明——赋值；

这三种数据的准备情况我们称之为“执行上下文”或者“执行上下文环境”。

console.log(f);//undefined 函数表达式

console.log(f1);//赋值 函数声明

console.log(this);

var f=function(){};//函数表达式

function f1(){}//函数声明

var fnpre=function(){

console.log("函数赋值pre ");

};

console.log(fn);

console.log(fnpre);

console.log(fnback);

console.log(this);

console.log("上下文执行 函数声明及this首先赋值");

console.log("在执行代码之前，把将要用到的所有的变量都事先拿出来，有的直接赋值了，有的先用undefined占个空");

var fnback=function(){

console.log("函数赋值back ");

}

function fn(){

console.log("声明函数");

}

var a=1,

b=2;

function co(s){

console.info(s);

}

fn1=function(){

co("javascript除了全局作用域外，只有函数可以创建作用域！");

},

fn2=function(){

co("声明变量时，要在全局代码的前端，函数体要在一开始就声明。而且建议使用单 var 形式");

};

if(a==1){

name1="name1"

var name2="name2"

}

co(name1+"--"+name2);

fn1();

fn2();;

//闭包：1、函数返回值是函数；2、函数作为参数传递

var m=10,

fn=function(){

return function(x){

if(x>m){

console.log(">>>>"+x);

}else{

console.log("<<<<"+x);

}

};

};

var fn1=fn();

fn1(15);//函数返回值是函数

(function(f){//函数作为参数

var m=1;

var fn2=f();

fn2(9);

})(fn);

})