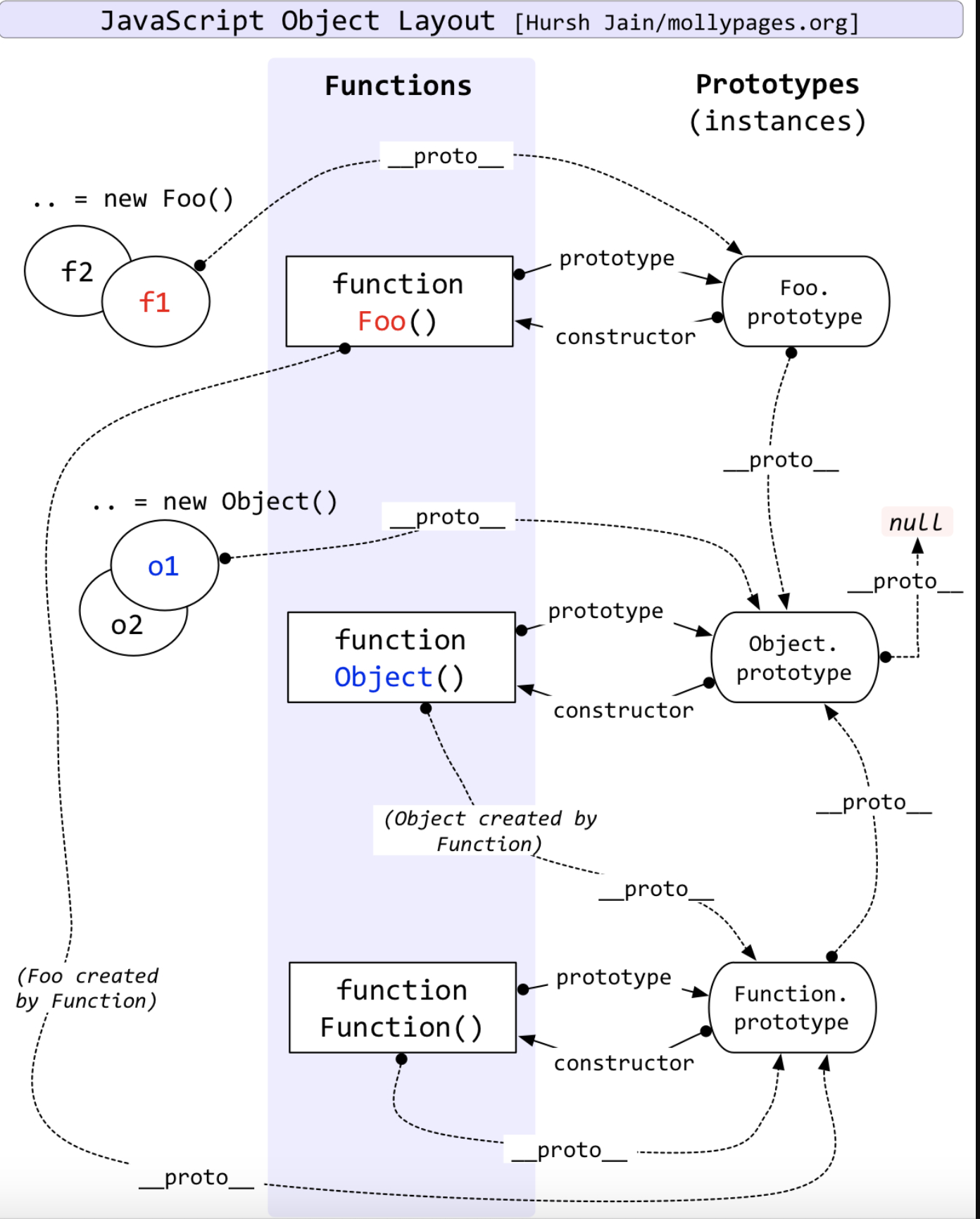
## JS继承与原型链

JavaScript是动态的，并且本身也不提供一个class的实现（ES6/ES2015）中的class仅仅是语法糖，JS仍然是基于原型的。

当谈到继承时，JS只有一种结构（对象）。每一个实例对象（object）都有一个私有属性\_\_proto\_\_指向其构造函数的原型对象（prototype）。该原型对象也有自己的\_\_proto\_\_原型对象，一层层向上直到一个对象的原型对象为null。根据定义，这个null没有原型对象，并作为这个原型链的最后一环。

几乎所有JS中的对象都是位于原型链顶端的Object的实例。



## 基于原型链的继承

**继承属性**

JS对象是动态的属性“包”（指其自己的属性）。

JS对象有一个指向原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上寻找，还会搜索该对象的原型，以及该对象的原型的原型，依次层层往上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

遵循ECMAScript标准，someObject.[[Prototype]]符号用于指向someObject的原型。

从ES6开始，一个对象的原型对象可以通过Object.getPrototypeOf()和Object.setPrototypeOf()访问器来访问。这等同于JS的非标准但许多浏览器实现的\_\_proto\_\_属性。它不应该和构造函数func的prototype属性相混淆，被构造函数创建的实例对象的[[Prototype]]属性指向func的prototype属性。

Object.prototype表示Object的原型对象。

function f() { *// 从一个函数创建一个对象，它拥有自身属性a、b*

this.a = 1;

this.b = 2;

}

let o = new f(); *// {a: 1, b: 2}*

*// 在f函数的原型上定义属性*

f.prototype.b = 3;

f.prototype.c = 4;

*// 这时原型链为：{a: 1, b: 2}--->{c: 1, d: 2}--->Object.prototype--->null*

*// 是o自身属性吗？->是*

console.log(o.a); *// 1*

*// 是o自身属性吗？->是。*

*// 原型上也有一个属性b，但是不会被访问到，这种情况被称为“属性遮蔽”（property shadowing）*

console.log(o.b); *// 2*

*// 是o自身属性？->否->看看它的原型上有没有->是o.[[Prototype]]的属性吗？->是*

console.log(o.c); *// 4*

*// 是o自身属性？->否->看看它的原型上有没有->是o.[[Prototype]]（即f.prototype）的属性吗？->否--*

*// -->是o.[[Prototype]].[[Prototype]]（即Object.prototype）的属性吗？->否--*

*// -->是o.[[Prototype]].[[Prototype]].[[Prototype]]（即null）的属性吗？->否*

console.log(o.d); *// undefined*

**继承方法**

JS并没有其他基于类的语言所定义的“方法”。在JS中，任何函数都可以添加到对象上作为对象的属性。函数的继承和其他属性的继承没有区别，包括上边的“属性遮蔽”（这种情况相当于其他语言的方法重写）。

当继承的函数被调用时，this指向的是当前继承的对象，而不是被继承的函数所在的原型对象。

let o = {

a: 2,

m: function() {

return this.a + 1;

}

}

*// 调用o.m时，this指向了o*

console.log(o.m()); *// 3*

let p = Object.create(o);

p.a = 4;

*// 调用p.m时，this指向了p*

*// 所以，此时的this.a即p.a就是p的自身属性a*

console.log(p.m()); *// 5*

在JS中函数（function）是允许拥有属性的，所有的函数都有一个特别的属性prototype

## 使用不同的语法结构来创建对象和生成原型链

**使用语法结构创建的对象**

let obj = {

name: 'obj'

}

*// obj继承了Object.prototype上的所有属性*

*// 原型链：obj--->Object.prototype--->null*

let arr = [1, 2, 3]

*// 数组都继承自Array.prototype*

*// Array。prototype中包含indexOf、forEach等方法*

*// 原型链：arr--->Array.prototype--->Object.prototype--->null*

function func() {

return 0;

}

*// 函数都继承自Function.prototype*

*// Function.prototype中包含call、apply、bind等方法*

*// 原型链：func--->Function.prototype--->Object.prototype--->null*

**使用构造器创建的对象**

在JS中，构造器就是一个普通的函数。当使用new操作符来作用于这个函数时，它就可以被称为构造方法（函数）。

function Graph() {

this.vertices = []; *// 顶点*

this.edges = []; *// 边*

}

Graph.prototype.addVertex = function(v) {

this.vertices.push(v);

}

let g = new Graph();

*// g是生成的对象，它的自身属性有vertices和edges*

*// 在g被实例化时，g.[[Prototype]]指向了Graph.prototype*

**使用Object.create创建的对象**

ES5中新引入的方法，使用该方法创建一个新对象，新对象的原型就是方法接收的第一个参数

let a = {a: 1};

*// a--->Object.ptototype--->null*

let b = Object.create(a);

*// b--->a--->Object.ptototype--->null*

console.log(b.a); *// 1 继承而来*

let c = Object.create(b);

*// c--->b--->a--->Object.ptototype--->null*

let d = Object.create(null);

*// d--->null*

console.log(d.hasOwnProperty); *// undefined 因为d并没有继承Object.prototype*

**使用class关键字创建的对象**

ES6引入了一套新的关键字用来实现class。但它们仍是基于原型的。这些新的关键字包括class、constructor、static、extends、super

'use strict'

class Polygon {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

class Square extends Polygon {

constructor(sideLegnth) {

super(sideLegnth, sideLegnth);

}

get area() {

return this.height \* this.width;

}

set sideLegnth(length) {

this.height = length;

this.width = length;

}

}

let square = new Square(2);

console.log(square.area); *// 4*

square.sideLength = 3;

console.log(square.area); *// 9*

**性能**

在原型链上查找属性比较耗时，对性能有副作用，这在性能要求苛刻的情况下很重要。试图访问不存在的属性会遍历整个原型链。

遍历对象的属性时，原型链上的**每个**可枚举属性都会被枚举出来。要检查对象是否具有自己定义的属性，而不是原型链上的某个属性，则必须使用所有对象从Object.prototype继承来的hasOwnProperty方法

console.log(g.hasOwnProperty('vertices')); *// true*

console.log(g.hasOwnProperty('nope')); *// false*

console.log(g.hasOwnProperty('addVertex')); *// false*

console.log(g.\_\_proto\_\_.hasOwnProperty('addVertex')); *// true*

hasOwnProperty是JS中唯一一个处理属性，并且**不会**遍历原型链的方法。

另一种这样的方法Object.keys()

注意⚠️：检查属性是否为undefined**并不能**检查其是否存在，因为该属性可能存在且刚好被设置为了undefined。