**js的继承机制**

[**开发部知识管理空间**](http://wiki.cregcloud.com/document/index?document_id=2) **/**[**WEB组**](http://wiki.cregcloud.com/document/index?document_id=6) **/**[**前端**](http://wiki.cregcloud.com/document/index?document_id=11)

 被 [xiaozongjie（肖宗杰）](http://wiki.cregcloud.com/system/main/index" \t "_parent) 创建于 2019/04/02 10:02:45 ， 被 [xiaozongjie（肖宗杰）](http://wiki.cregcloud.com/system/main/index" \t "_parent)最后更新于 2019/04/03 13:43:38 (查看修改历史)

[编辑](http://wiki.cregcloud.com/page/edit?document_id=57) 收藏[分享](http://wiki.cregcloud.com/page/display?document_id=57" \t "_blank) 导出

面向对象编程很重要的一个方面，就是对象的继承。A 对象通过继承 B 对象，就能直接拥有 B 对象的所有属性和方法。这就使得代码可以被需要时拿来复用。javascript语言传统上是通过原型对象（prototype）和构造函数的实例化来实现，ES6（2015年6月正式发布）中给出了更接近其他面向对象的编程语言的class语法来实现继承的操作。基本上，ES6 的class可以看作只是一个语法糖，它的绝大部分功能，ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。  
举例说明：  
传统地，通过构造函数生成实例对象的写法如下：

1. function Point(x, y) {
2. this.x = x;
3. this.y = y;
4. }
5. Point.prototype.toString = function () {
6. return '(' + this.x + ', ' + this.y + ')';
7. };
8. var p = new Point(1, 2);

现在改用ES6语法，通过构造函数生成实例对象的写法如下：

1. class Point {
2. constructor(x, y) {
3. this.x = x;
4. this.y = y;
5. }
6. toString() {
7. return '(' + this.x + ', ' + this.y + ')';
8. }
9. }

ES6 的类，完全可以看作构造函数的另一种写法。具体语法，参考[http://es6.ruanyifeng.com/#docs/class](http://es6.ruanyifeng.com/" \l "docs/class" \t "_parent)

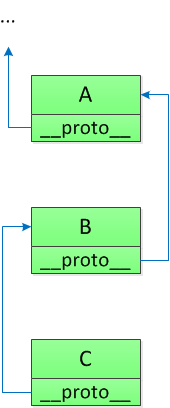
JavaScript 语言继承的核心是通过“原型对象”（prototype）实现，新旧语法都是这样，下面介绍 JavaScript 的通过原型对象怎么实现继承。  
为了叙述方便，下文有的叙述中‘原型对象’简写为‘原型’。  
JavaScript 继承机制的设计思想就是，原型对象的所有属性和方法，都能被实例对象共享。也就是说，如果属性和方法定义在“原型对象”（prototype）上，那么所有实例对象就能共享，不仅节省了内存，还体现了实例对象与实例对象之间的联系（继承的是同一个原型）。

**继承的实质是与一种查找模型搭上关系—原型链**

关于继承需要知道的三个基本前提：  
一切数据皆对象；\_\_proto\_\_属性；prototype属性：  
1.js语言中，一切数据皆对象；  
2.Javascript规定，每一个函数都有一个prototype属性，指向另一个对象（函数的原型对象）。这个对象的所有属性和方法，都会被构造函数的实例继承。  
这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。  
3.JavaScript 规定，所有对象都有自己的原型对象（prototype），每个实例对象（ object ）都有一个私有属性（称之为 \_\_proto\_\_）指向它的原型对象（ prototype ）。该原型对象也有一个自己的原型对象( \_\_proto\_\_ ) 。因此，就会形成一个“原型链”（prototype chain）：对象到原型，再到原型的原型……，某个对象的原型链，就是由该对象开始，通过\_\_proto\_\_属性连接起来的一串对象；  
如果一层层地上溯，所有对象的原型最终都可以上溯到Object.prototype，即Object构造函数的prototype属性。也就是说，所有对象都继承了Object.prototype的属性。这就是所有对象都有valueOf和toString方法的原因，因为这是从Object.prototype继承的。  
那么，Object.prototype对象有没有它的原型呢？回答是Object.prototype的原型是null。null没有任何属性和方法，也没有自己的原型。因此，原型链的尽头就是null。  
以上的三个基本前提是由于js的继承模型造成的。我们接下来会介绍这种模型，但现在先来叙述我们从这三个前提中得出的俩个基本结论：  
（1）js中，所有的对象都可以通过\_\_proto\_\_属性访问到它的原型对象；  
(2）js中，所有的函数也是对象，它同时具有\_\_proto\_\_属性和prototype属性;  
对于（2），  
Function作为对象，可以通过\_\_proto\_\_属性访问到Function prototype object,即有：Function.\_\_proto\_\_===Function.prototype //结果为true；  
同时，Function作为函数，可以通过prototype属性访问到Function prototype object,即有：  
Function.prototype===Function.prototype, //当然为true ，自身等于自身  
同时，Function prototype object (表示含义：Function的原型对象)，作为一个对象，它的原型又是谁呢？  
答案：Function.prototype.\_\_proto\_\_===Object.prototype //true

某个对象的原型链，就是由该对象开始，通过\_\_proto\_\_属性连接起来的一串对象。

如下图的例子：

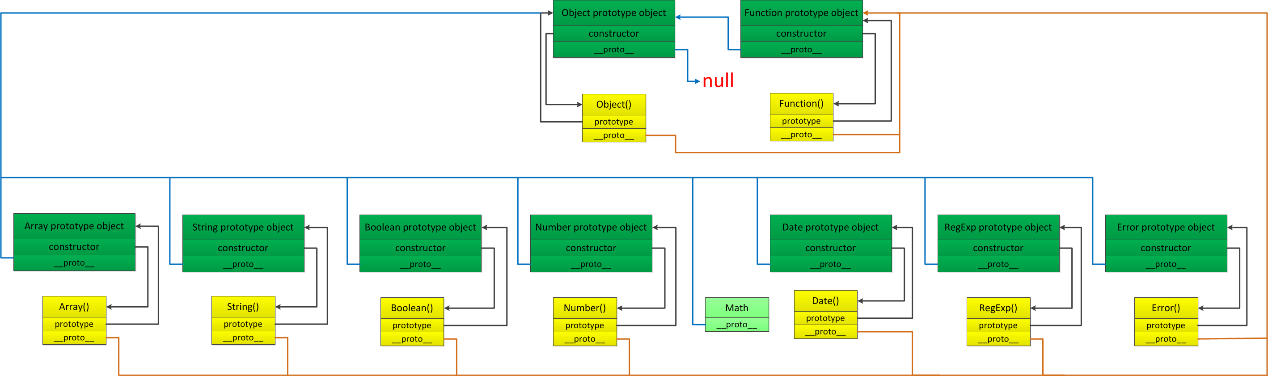


\_\_proto\_\_ 并不是语言本身的特性，这是各大厂商具体实现时添加的私有属性，虽然目前很多现代浏览器的 JS 引擎中都提供了这个私有属性，但依旧不建议在生产中使用该属性，避免对运行环境产生依赖。生产环境中，我们可以使用Object.getPrototypeOf 方法来获取实例对象的原型，然后再来为原型添加方法/属性。

上图中，A,B,C分别代表3个对象，蓝色箭头串接起来的所有对象就构成了对象C的原型链，其中C的\_\_proto\_\_属性指向B，B的\_\_proto\_\_属性指向A，A的\_\_proto\_\_属性可能指向更高层的对象，也可能指向null(表示A不继承任何对象的属性和方法)。如果我们引用了C的某个属性或者方法，那么JavaScript就会顺着C的原型链进行查找，即首先查找对象C本身，看所引用的属性名或者方法名是否存在，如果存在就停止查找直接返回，如果不存在，就通过C的\_\_proto\_\_属性找到原型链中的B对象，继续在B对象中查找，如果B对象中找到所引用的属性名或者方法名，那么就停止查找直接返回，如果B对象中也不存在，就通过对象B的\_\_proto\_\_属性找到原型链中的A对象，继续重复上述查找过程，直到找到所引用的属性或者方法为止(同时也可能查找完对象C的整个原型链也没有找到所引用的属性或者方法，那么该属性或者方法就是undefined的)。

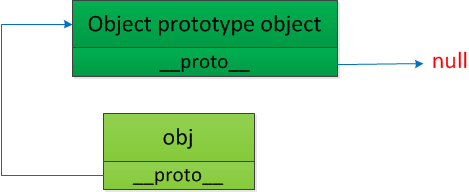
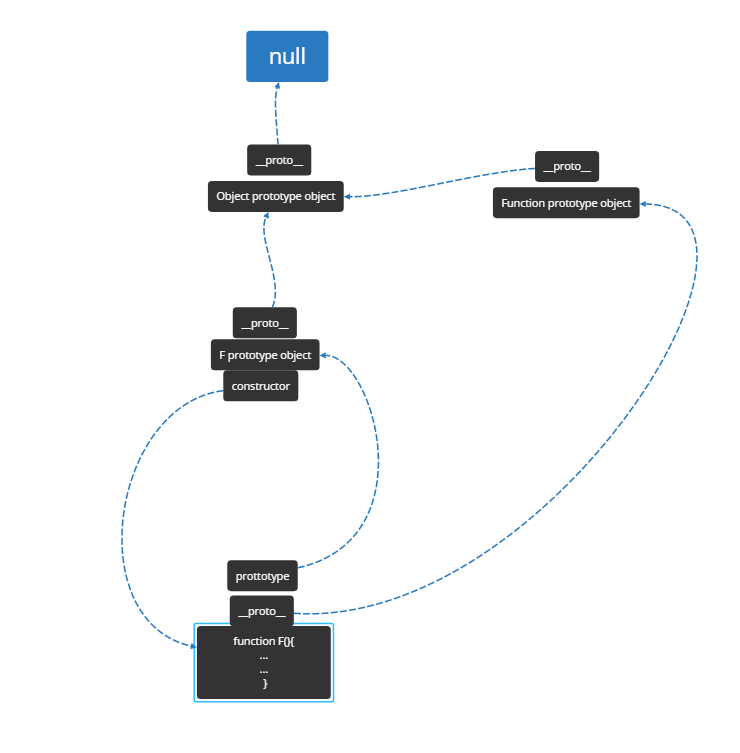
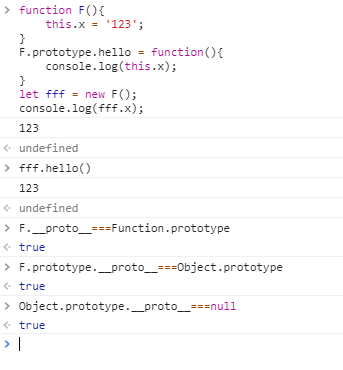
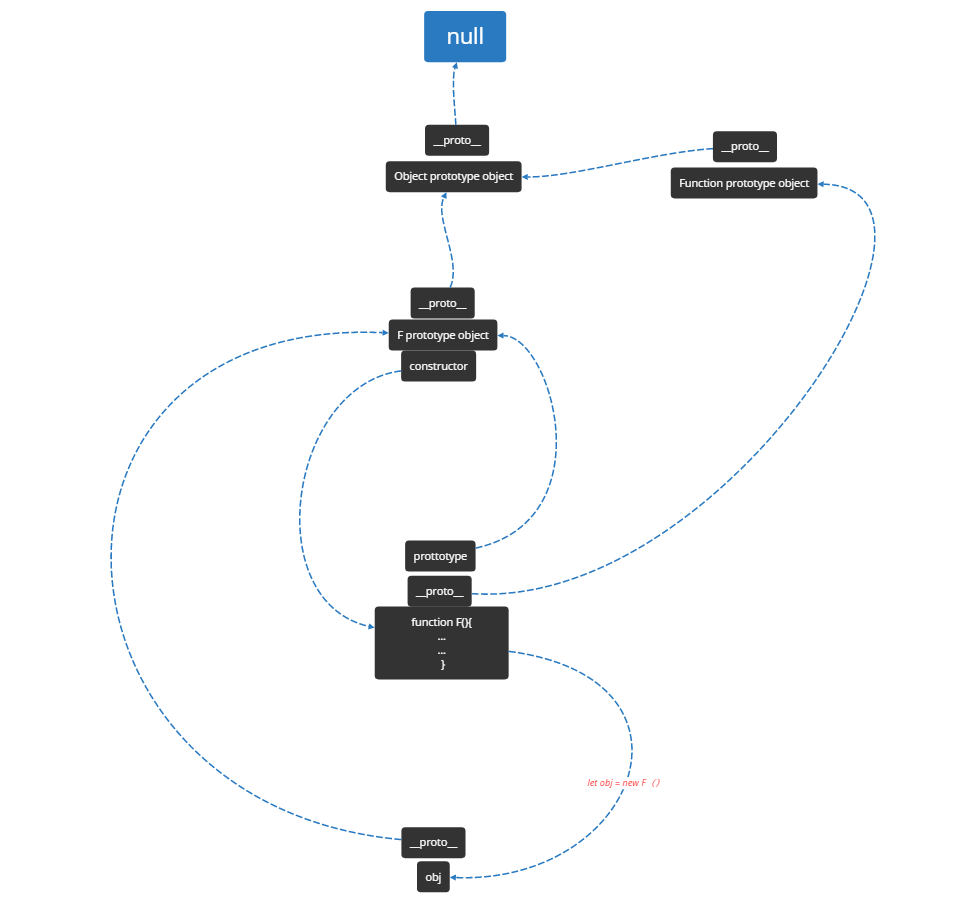
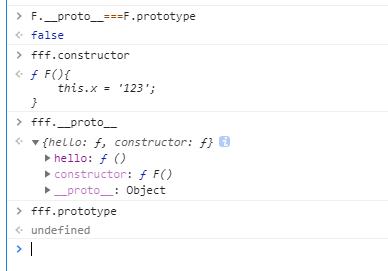
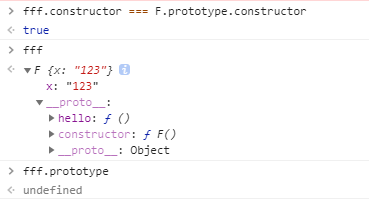
**JavaScript当中已经存在的原型链**

我们拿一些内置的函数和对象来举例。  
（关于所有的内置对象（函数对象和非函数对象）请参考[JavaScript的所有的标准的内置对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects" \o "JavaScript的所有的标准的内置对象" \t "_parent)）

在JavaScript当中存在Object，Function，Array，String，Boolean，Number，Date，Error，RegExp这9个函数对象和一个非函数的Math对象，通过这上述9个函数我们可以创建相应的对象，同时，这9个函数的prototype属性所指向的原型对象也是内置的（即这种语言要求内部具备的）。下面的图示解释了这几个函数各自prototype属性以及作为对象身份下\_\_proto\_\_属性所指向的原型对象之间的关系。  
  
以下，为强调函数对象的函数身份，简称函数对象为函数，但我们知道它还有对象的身份。  
上面的图示中，黄色方框代表那9个函数，深绿色方框代表函数的prototype属性指向的原型对象，名字都叫xx prototype object，浅绿色方框(即Math对象)代表普通对象（非函数对象），请注意Math由于是普通对象它没有prototype属性，蓝色箭头连接非函数对象(无论是普通对象如Math，还是原型对象)的\_\_proto\_\_属性，而土黄色箭头连接函数的\_\_proto\_\_属性。

　　通过上图可以发现，这9个函数对象的原型链最终都指向Function prototype object，所有非函数对象的原型链最终都指向Object prototype object，并且Function prototype object的\_\_proto\_\_属性也指向Object prototype object，Object prototype object的\_\_proto\_\_属性指向为null。我们知道，Object prototype object是所有原型链的顶端的必经之路，再往上是null。以此推广到更广泛的函数对象和非函数对象，通过原型链查找规则可知：  
1.Function prototype object包含了所有函数共享的属性和方法，而Object prototype object包含了所有对象都共享额属性和方法；  
2.所有函数对象同时继承了Object prototype object和Function prototype object上的属性和方法，而所有非函数对象只继承了Object prototype object上的方法。

**创建对象时，原型链的构造过程**

在JavaScript当中创建对象有2中方式，一种是通过定义构造函数使用new方法来构造，另一种是使用对象字面量的方式，  
1.使用字面量时，即:  
let obj = {  
name: “Jim Green”  
};  
JavaScript会为我们做两件事：分配内存存储新创建的对象；把新创建对象的\_\_proto\_\_属性指向Object prototype object ,如下图：  
  
2.通过定义构造函数使用new方法来构造：  
为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数（Constructor）模式。  
使用函数的方式构造对象分为两步:（1）首先需要定义一个函数作为构造函数；（2）然后使用new方法构造对象。  
对于（1），假设我们定义了一个函数名为F,function F（）{} //注释:老浏览器识别小写的构造函数名字，新浏览器都推荐的写法风格是构造函数的名字首字母大写。  
此时JavaScript会为我们做两件事，第一:根据我们定义的函数创建一个函数对象，第二，设置这个函数的prototype属性(函数身份)和\_\_proto\_\_属性（对象身份）。其中\_\_proto\_\_属性指向的Function prototype object，而prototype属性指向一个为函数F新创建的原型对象，这个新创建的原型对象通过调用new Object()构造出来（这意味着F.prototype.\_\_proto\_\_===Object.prototype //true），并且为这个新创建的对象添加constructor属性，该属性指向函数对象F。最后的结果如下图所示:  
  
  
对于（2），当我们使用new方法调用F函数的时候，JavaScript也会为我们做两件事，第一，分配内存作为新创建的对象，第二，将新创建的对象的\_\_proto\_\_属性指向函数F的原型对象，结果如下图:  
  
  
上图中，obj就是调用new方法通过函数F创建出来的对象，我们可以看到对象obj的原型链包含了函数F的原型对象，以及Object prototype object，这样，对象obj通过原型链查找规则，就能继承函数F的原型对象，以及Object prototype object上面定义的属性和方法了。并且如果我们想知道一个对象是由哪个方法构建的，只需要访问这个对象的constructor属性即可，上例中，只要我们访问obj.constructor，那么就知道obj是由函数F创建的。同时，由于F prototype object上文中介绍是由new Object函数创建的，根据此处介绍，F prototype object的\_\_proto\_\_属性应该指向Object函数的原型对象，即Object prototype object。  
它的原型链查找之路就是： obj自身—> F.prototype —> Object.prototype —> null  
并且我们验证了以下:  
1.实例对象是没有prototype属性的,它只有\_\_proto\_\_属性。  
2.每一个实例调用constructor属性，默认查找调用prototype对象的constructor属性。  


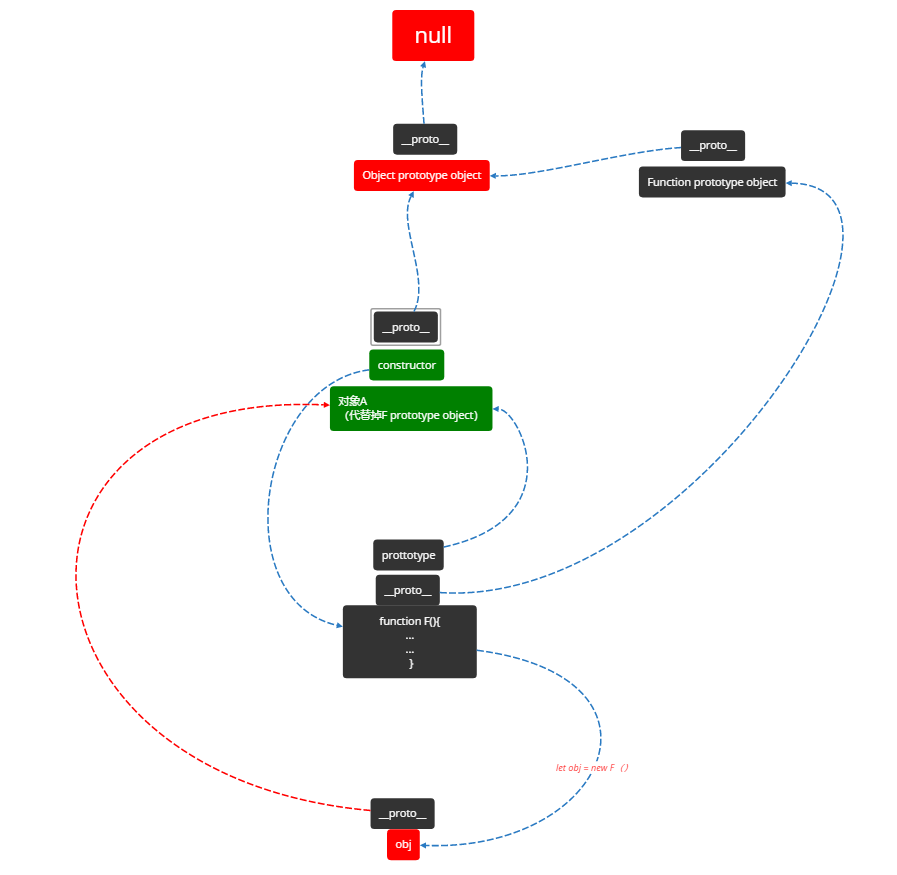
**理解了上面的原理之后，我们来理解继承**

假设目前有一个对象A(这个对象可以是任意的，包括JavaScript 的[内置对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects" \o "内置对象" \t "_parent)，任何函数对象，任何原型对象,以及我们自己new出来的对象),现在想创建一个对象obj，让obj继承A的属性和方法。通过上面的介绍，我们知道创建对象有两种方式，但是使用对象字面量创建的对象其原型链总是只包含两个对象，一个是其自己，一个是Object prototype object，无法达到重用的目的生成多个实例，让对象obj继承对象A属性和方法如果通过改变obj的\_\_proto\_\_属性的指向为A是可以做到，但是不能重用该原型链，此方法没有重用原型链生成对象的意义，所以不推荐，另外直接改变\_\_proto\_\_的做法不推荐请参看文末补充（3）。

因此，只能使用函数的方式创建对象，做法上让对象A包含在新创建对象obj的原型链中即可。

根据上面的讲解，如果是用函数的方式创建对象，那么在调用new方法时，新创建对象的\_\_proto\_\_属性会指向函数的原型对象。因此，只要在调用函数之前，将函数的原型对象换成A，然后再调用new方法，就可以将对象A包含在新创建的对象obj的原型链中，这样通过原型链查找规则，obj就继承了A的属性和方法。假设用来创建对象obj的函数为F，则相关代码为:

1. F.prototype = A;
2. F.prototype.constructor = F;
3. var obj = new F();

上面代码中第二行的F.prototype.constructor = F，是因为对象A中可能没有constructor属性，或者constructor属性不指向F，而为了让用户从实例对象的constructor属性，就可以正确的知道该对象是使用函数F构造出来的，实际应用中强烈推荐为F.prottotype设置construtor指向F。相关图示如下图:  


**补充（1） 构造函数中的继承方式与ES6的差别**

所谓”构造函数”，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。其中ES5的继承机制（构造函数+prototype方式）和ES6的继承机制（class语法)是不一样的，ES5 的继承，实质是先创造子类的实例对象this，然后再将父类的方法添加到this上面（Parent.apply(this)）。ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到this上面（所以必须先调用super方法），然后再用子类的构造函数修改this。[具体应在学习class语法的时候注意。](http://es6.ruanyifeng.com/" \l "docs/class-extends" \o "具体应在学习class语法的时候注意。" \t "_parent)

**补充（2） 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。另外，试图访问不存在的属性时会遍历整个原型链。  
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，而不是其原型链上的某个属性，则必须使用所有对象从 Object.prototype 继承的 hasOwnProperty 方法。

**补充（3）**[**为什么不用proto修改原型链**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/proto)



最后修改时间：2019-04-03 13:43:38