帮你彻底搞懂JS中的prototype、__proto__与constructor（图解）

原创 置顶 码飞_CC 2018-07-25 22:50:47 121459 收藏 1043 版权

分类专栏： 前端之JS 深入剖析前端中的难理解与易混淆知识 文章标签： prototype __proto__ constructor 原型链

文章目录

提示：不要排斥，静下心来，认真读完，你就搞懂了！(可以先看一下最后的总结部分再回过头来完整看完)

1. 前言

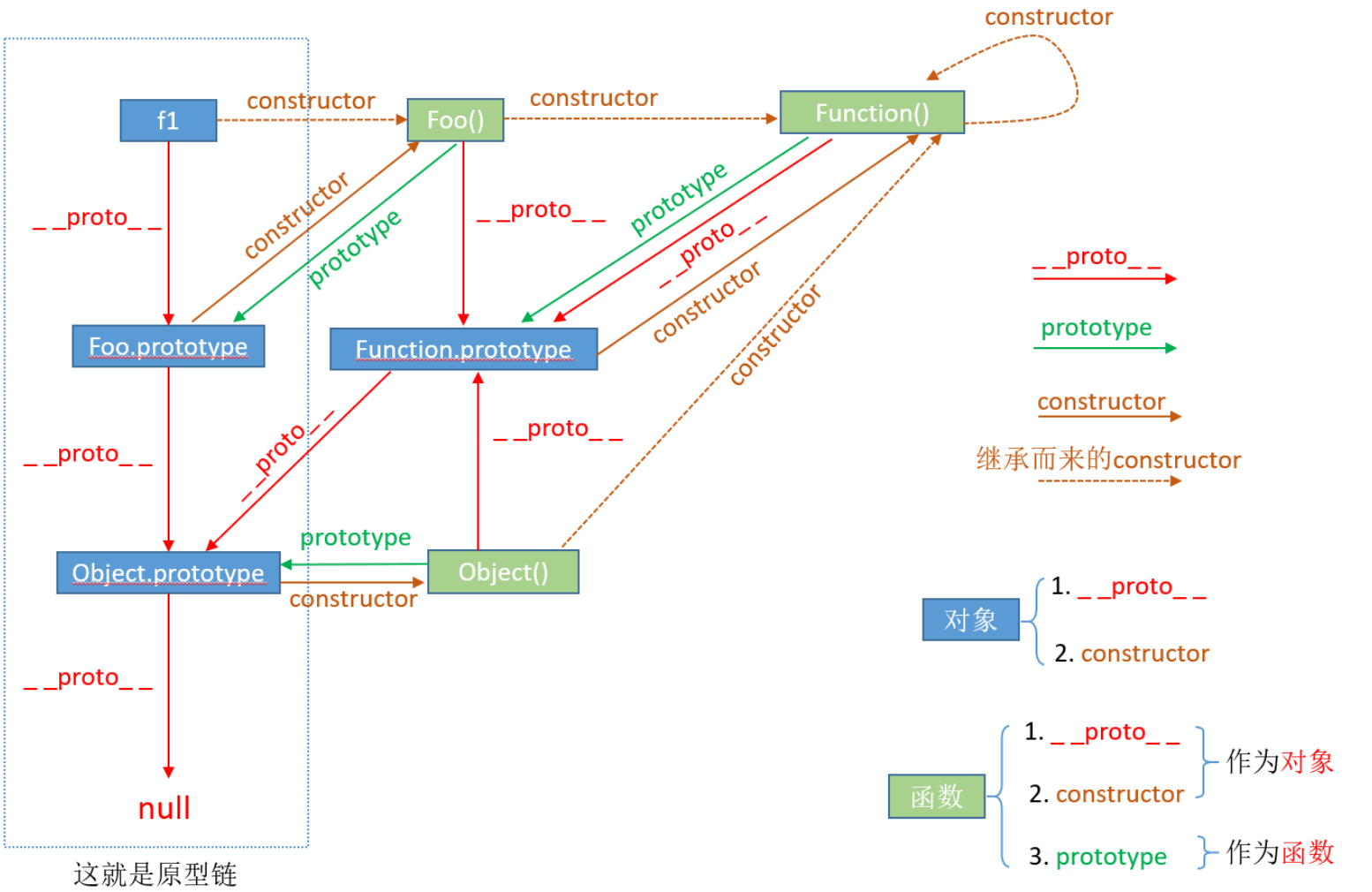
作为一名前端工程师，必须搞懂JS中的`prototype`、`__proto__`与`constructor`属性，相信很多初学者对这些属性存在许多困惑，容易把它们混淆，本文旨在帮助大家理清它们之间的关系并彻底搞懂它们。这里说明一点，`__proto__`属性的两边是各由两个下划线构成（这里为了方便大家看清，在两下划线之间加入了一个空格：`_ _proto_ _`，读作“**dunder proto**”，“**double underscore proto**”的缩写），实际上，该属性在ES标准定义中的名字应该是`[[Prototype]]`，具体实现是由浏览器代理自己实现，谷歌浏览器的实现就是将`[[Prototype]]`命名为`__proto__`，大家清楚这个标准定义与具体实现的区别即可（名字有所差异，功能是一样的），可以通过该方式检测引擎是否支持这个属性：`Object.getPrototypeOf({__proto__: null}) === null`。本文基于谷歌浏览器（版本72.0.3626.121）的实验结果所得。

现在正式开始！ 让我们从如下一个简单的例子展开讨论，并配以相关的图帮助理解：

```
function Foo() {...};
let f1 = new Foo();
```

以上代码表示创建一个构造函数`Foo()`，并用`new`关键字实例化该构造函数得到一个实例化对象`f1`。这里稍微补充一下new操作符将函数作为构造器进行调用时的过程：函数被调用，然后新创建一个对象，并且成了函数的上下文（也就是此时函数内部的this是指向该新创建的对象，这意味着我们可以在构造器函数内部通过this参数初始化值），最后返回该新对象的引用，详细请看：详解JavaScript中的new操作符。虽然是简简单单的两行代码，然而它们背后的关系却是错综复杂的，如下图所示：

function Foo() { ...};
let f1 = new Foo();



这就是原型链

看到这图**别怕**，让我们一步步剖析，彻底搞懂它们！

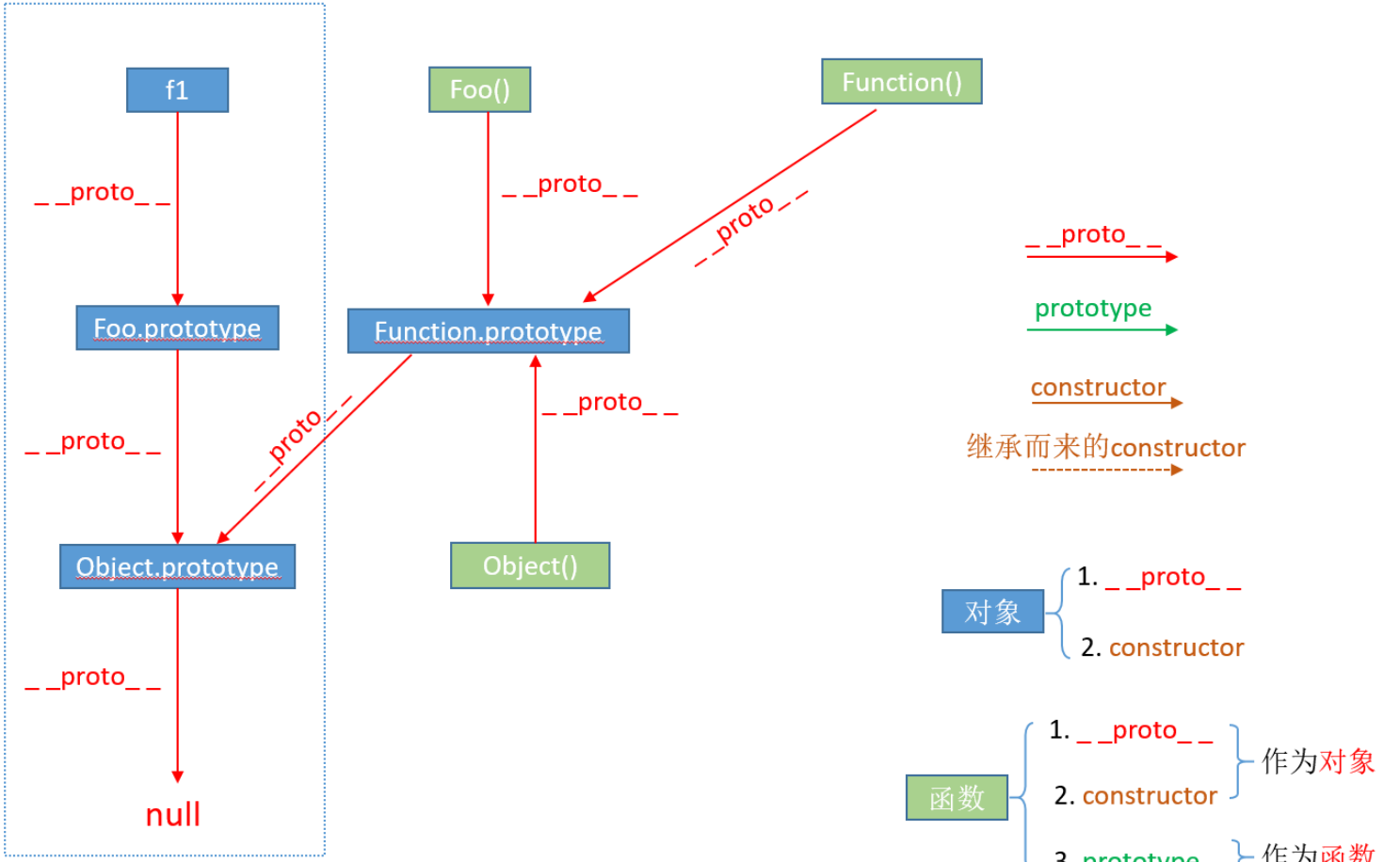
图的说明：右下角为图例，红色箭头表示`__proto__`属性指向、绿色箭头表示`prototype`属性的指向、棕色实线箭头表示本身具有的`constructor`属性的指

向，棕色虚线箭头表示继承而来的`constructor`属性的指向；蓝色方块表示对象，浅绿色方块表示函数（这里为了更好看清，Foo()仅代表是函数，并不是指执行函数Foo后得到的结果，图中的其他函数同理）。图的中间部分即为它们之间的联系，图的最左边即为例子代码。

2. _ _ proto _ _ 属性

首先，我们需要牢记两点：① `__proto__`和`constructor`属性是**对象**所独有的；② `prototype`属性是**函数**所独有的。但是由于JS中函数也是一种对象，所以函数也拥有`__proto__`和`constructor`属性，这点是致使我们产生困惑的很大原因之一。上图有点复杂，我们把它按照属性分别拆开，然后进行分析：

```
function Foo() { ...};
let f1 = new Foo();
```



这就是原型链

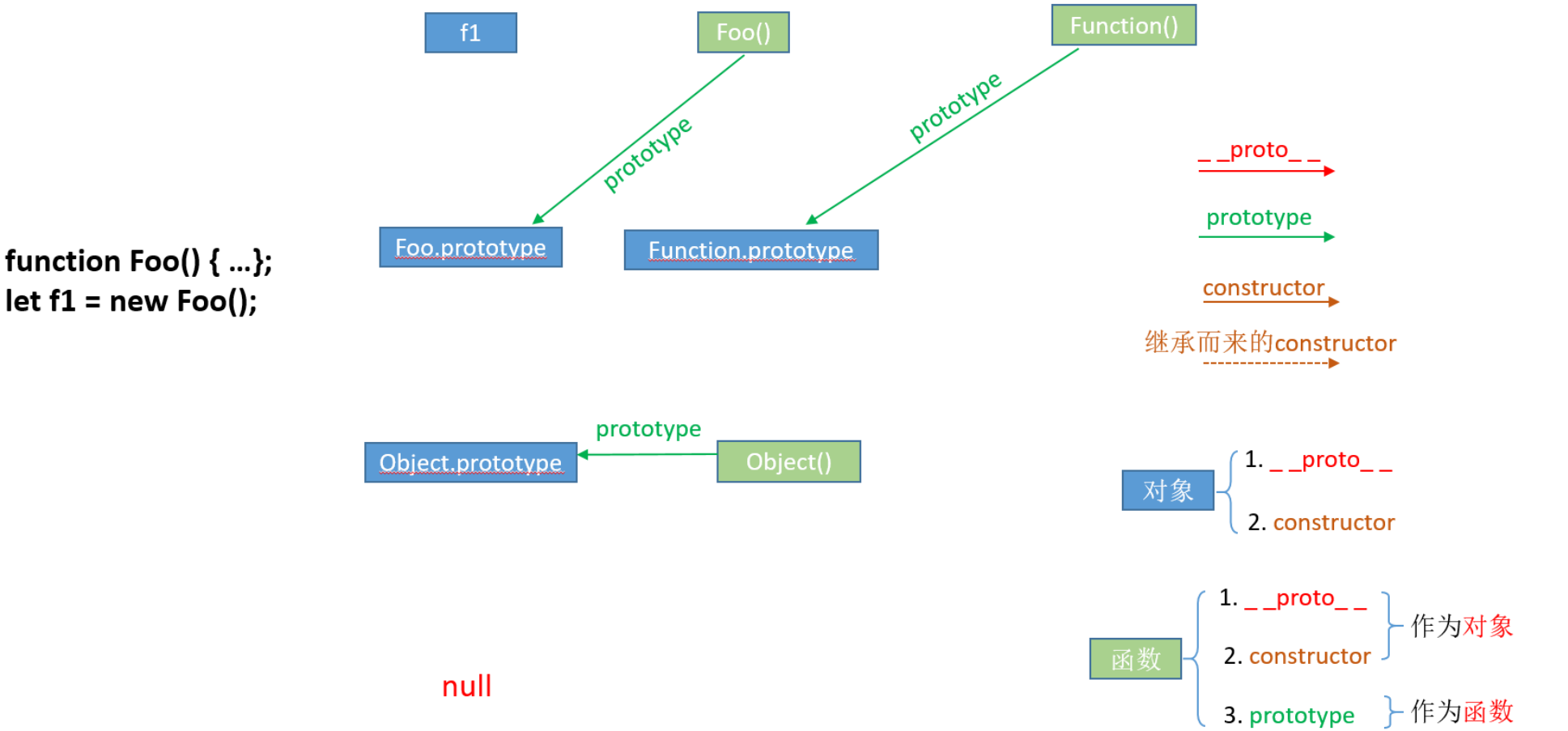
第一，这里我们仅留下`__proto__`属性，它是**对象所独有的**，可以看到`__proto__`属性都是由**一个对象指向一个对象**，即指向它们的原型对象（也可以理

解为父对象），那么这个属性的作用是什么呢？它的**作用**就是当访问一个对象的属性时，如果该对象内部不存在这个属性，那么就会去它的`__proto__`属性所指向的那个对象（可以理解为父对象）里找，如果父对象也不存在这个属性，则继续往父对象的`__proto__`属性所指向的那个对象（可以理解为爷爷对象）里找，如果还没找到，则继续往上找…直到原型链顶端**null**（可以理解为原始人。。。），再往上找就相当于在null上取值，会报错（可以理解为，再往上就已经不是“人”的范畴了，找不到了，到此结束，`null`为原型链的终点），由以上这种通过`__proto__`属性来连接对象直到`null`的一条链即为我们所谓的**原型链**。

其实我们平时调用的字符串方法、数组方法、对象方法、函数方法等都是靠`__proto__`继承而来的。

3. prototype属性

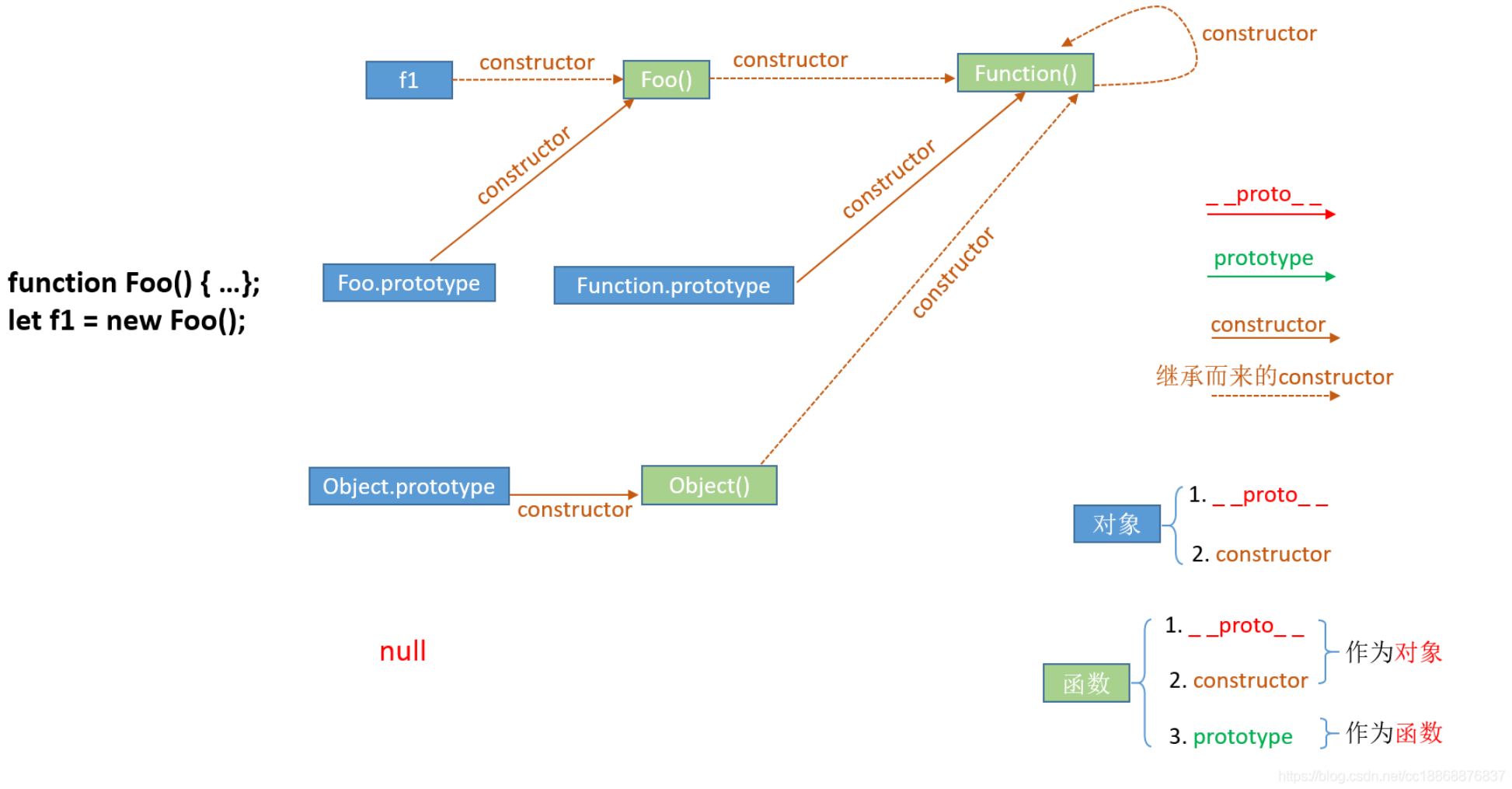
第二，接下来我们看`prototype`属性：



`prototype`属性，别忘了一点，就是我们前面提到要牢记的两点中的第二点，它是**函数所独有的**，它是从**一个函数指向一个对象**。它的含义是**函数的原型对象**，也就是这个函数（其实所有函数都可以作为构造函数）所创建的实例的原型对象，由此可知：`f1.__proto__ === Foo.prototype`，它们两个完全一样。那`prototype`属性的作用又是什么呢？它的**作用**就是包含可以由特定类型的所有实例共享的属性和方法，也就是让该函数所实例化的对象们都可以找到公用的属性和方法。**任何函数在创建的时候，其实会默认同时创建该函数的prototype对象。**

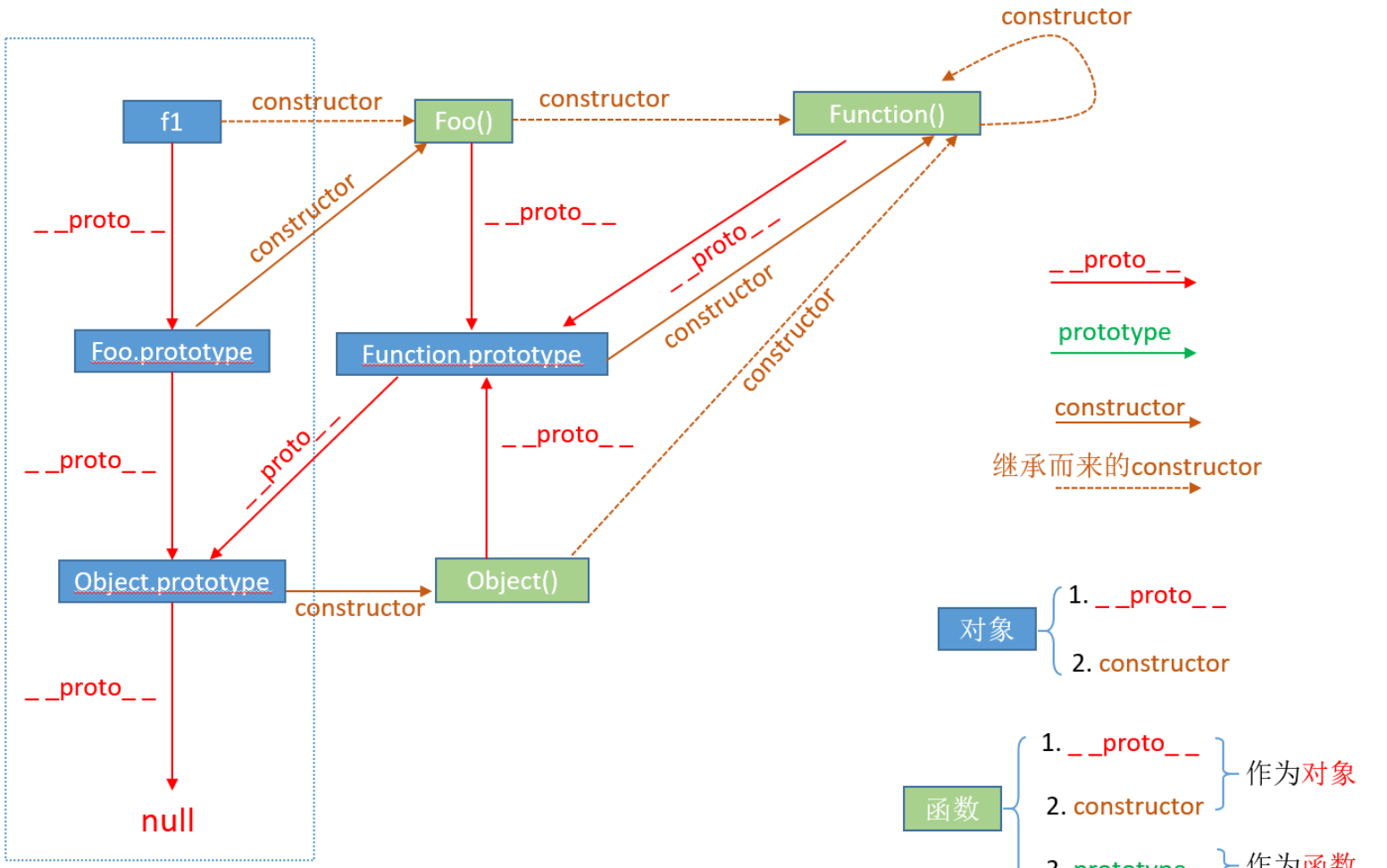
4. constructor属性

最后，我们来看一下`constructor`属性：



`constructor`属性也是**对象才拥有的**，它是从**一个对象指向一个函数**，含义就是**指向该对象的构造函数**，每个对象都有构造函数（本身拥有或继承而来，继承而来的要结合`__proto__`属性查看会更清楚点，如下图所示），从上图中可以看出**Function**这个对象比较特殊，它的构造函数就是它自己（因为Function可以看成是一个函数，也可以是一个对象），所有函数和对象最终都是由Function构造函数得来，所以`constructor`属性的终点就是**Function**这个函数。

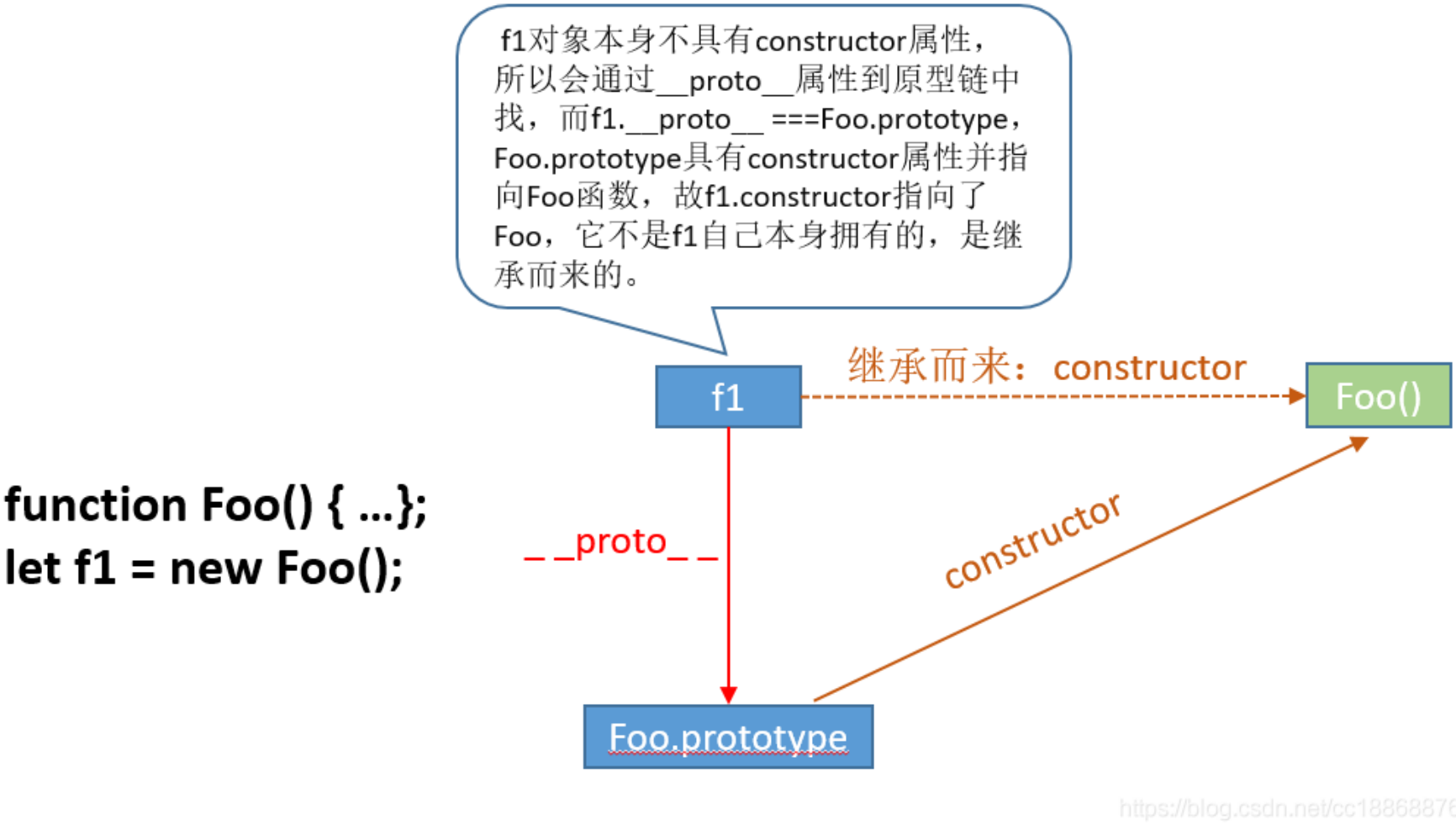
function Foo() { ...};
let f1 = new Foo();



这就是原型链

感谢网友的指出，这里解释一下上段中**“每个对象都有构造函数”**这句话。这里的意思是每个对象都可以找到其对应的constructor，因为创建对象的前提是需要有constructor，而这个constructor可能是对象自己本身显式定义的或者通过`__proto__`在原型链中找到的。**而单从constructor这个属性来讲，只有prototype对象才有。**每个函数在创建的时候，JS会同时创建一个该函数对应的prototype对象，而`函数创建的对象.__proto__ === 该函数.prototype，该函`

数.prototype.constructor===该函数本身，故通过函数创建的对象即使自己没有constructor属性，它也能通过__proto__找到对应的constructor，所以任何对象最终都可以找到其构造函数（null如果当成对象的话，将null除外）。如下：



5. 总结

总结一下：

1. 我们需要牢记两点：① `__proto__`和`constructor`属性是**对象**所独有的；② `prototype`属性是**函数**所独有的，因为函数也是一种对象，所以函数也拥有`__proto__`和`constructor`属性。
2. `__proto__`属性的**作用**就是当访问一个对象的属性时，如果该对象内部不存在这个属性，那么就会去它的`__proto__`属性所指向的那个对象（父对象）里找，一直找，直到`__proto__`属性的终点**null**，再往上找就相当于在null上取值，会报错。通过`__proto__`属性将对象连接起来的这条链路即**我们所谓的原型链**。
3. `prototype`属性的**作用**就是让该函数所实例化的对象们都可以找到公用的属性和方法，即`f1.__proto__ === Foo.prototype`。
4. `constructor`属性的含义就是**指向该对象的构造函数**，所有函数（此时看成对象了）最终的构造函数都指向**Function**。

本文就此结束了，希望对那些对JS中的`prototype`、`__proto__`与`constructor`属性有困惑的同学有所帮助。

最后，感谢这两篇博文，本文中的部分内容参考自这两篇博文：

- 一张图理解prototype、proto和constructor的三角关系
- prototype和__proto__的关系是什么?

小彩蛋：实现继承（相对完美、优雅）

```
function inherit(Child, Parent) {
    // 继承原型上的属性
    Child.prototype = Object.create(Parent.prototype)
    // 修复 constructor
    Child.prototype.constructor = Child
    // 存储超类
    Child.super = Parent
    // 静态属性继承
    if (Object.setPrototypeOf) {
        // setPrototypeOf es6
        Object.setPrototypeOf(Child, Parent)
    } else if (Child.__proto__) {
        // __proto__ es6 引入，但是部分浏览器早已支持
        Child.__proto__ = Parent
    } else {
        // 兼容 IE10 等陈旧浏览器
```

```
        // 将 Parent 上的静态属性和方法拷贝一份到 Child 上，不会覆盖 Child 上的方法
        for (var k in Parent) {
            if (Parent.hasOwnProperty(k) && !(k in Child)) {
                Child[k] = Parent[k]
            }
        }
    }
}
```