浅谈JavaScript中的面向对象编程(OOJS)

Object Oriented JavaScript

分享人: SJC

面向对象编程

面向过程编程:

假设你是汽车厂的厂长,要生产一辆汽车.面向过程的话,就如同指挥每一个工人的工作,一会儿让某车间的某工人在A机器上生产轴承,一会儿又让另一个工人在B机器上生产轮胎...

面向对象编程:

"一车间给我制造车身"

"二车间给我生产引擎"

....

车间就是一个个封装好的对象,对象中包括工人、机器、材料等成员变量,还有加工工艺这样的成员函数

对象　this　原型　继承

面向对象基本特性

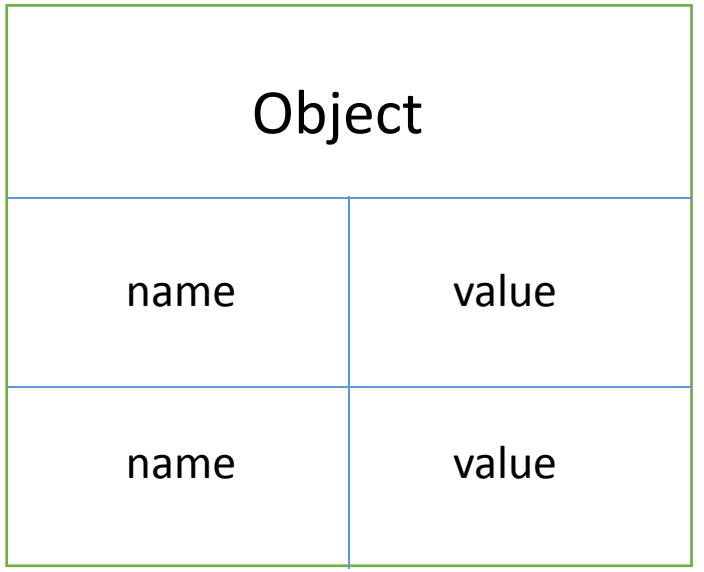
1.封装: 封装的概念好比一辆汽车,你学会开车的时候只需要会 诸如踩油门 刹车 转方向盘即可,无需去了解它发动机是如何发动的

2.继承

3.多态 举个例子,比如 "开"这个字,在不同的时候各有的意思, 比如 "开门","开窗"甚至有"开车","开饭"等,具有相同名称但含义不同

对象 ——若干属性的集合。

对象是属性的无需列表 . 属性包含键(始终是字符串)和值.如果一个属性的值是函数,它就被称为方法



把JavaScript 对象想象为上图中的哈希表可以帮助你更好的理解对象结构

函数和对象的关系

函数就是对象的一种，因为通过instanceof函数可以判断

```
var fn = function () { };
console.log(fn instanceof Object);  // true
```

函数是一种对象，但是函数与对象之间，却不仅仅是一种包含和被包含的关系，函数和对象之间的关系比较复杂，有一点鸡生蛋蛋生鸡的逻辑，咱们缕一缕。

还是先看一个小例子吧。

```
function Fn() {
    this.name = '孙继昌';
    this.age = 18;
}
var fn1 = new Fn();
```

上面的这个例子很简单，它能说明：对象可以通过函数来创建。
但是我要说——对象都是通过函数创建的——有些人可能反驳：不对！因为：

```
var obj = { a: 10, b: 20 };
var arr = [5, 'x', true];
```

但是，这个真的——是一种——“快捷方式”，在编程语言中，一般叫做“语法糖”。

其实以上代码的本质是：

```
//var obj = { a: 10, b: 20 };
//var arr = [5, 'x', true];
    var obj = new Object();
    obj.a = 10;
    obj.b = 20;

    var arr = new Array();
    arr[0] = 5;
    arr[1] = 'x';
    arr[2] = true;
```

而其中的 Object 和 Array 都是函数：

```
console.log(typeof (Object));  // function
console.log(typeof (Array));  // function
```

所以，可以很负责任的说——对象都是通过函数来创建的。

对象是函数创建的，而函数却又是一种对象——函数和对象到底是什么关？

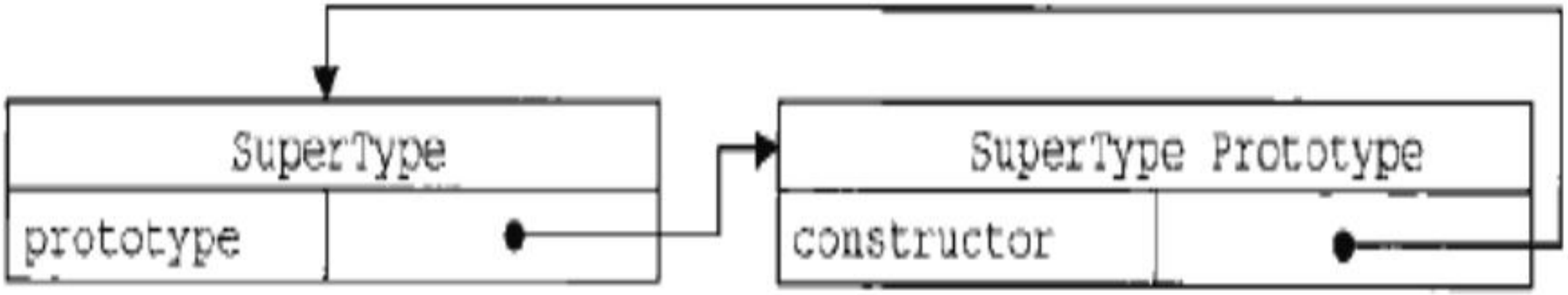
揭开这个谜底，还得先去了解一下——prototype原型。

原型

函数也是一种对象。他也是属性的集合，你也可以对函数进行自定义属性。

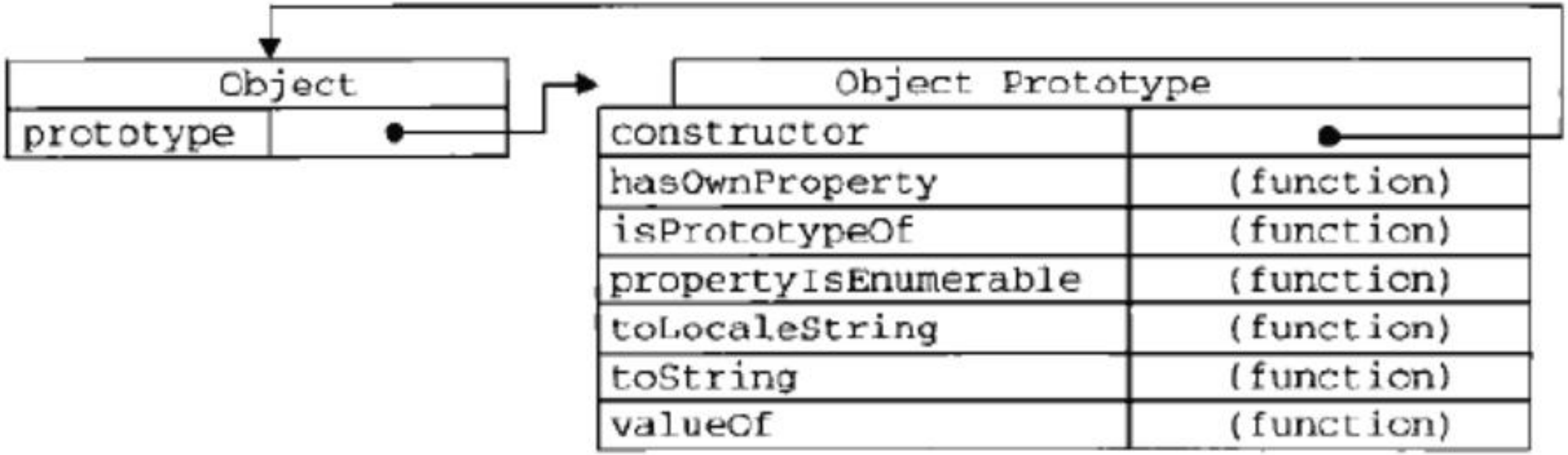
javascript默认的给函数一个属性——prototype。是的，每个函数都有一个属性叫做prototype。

这个prototype的属性值是一个对象（属性的集合，再次强调！），默认的只有一个叫做 constructor 的属性，指向这个函数本身。



如上图，SuperType是一个函数，右侧的方框就是它的原型。

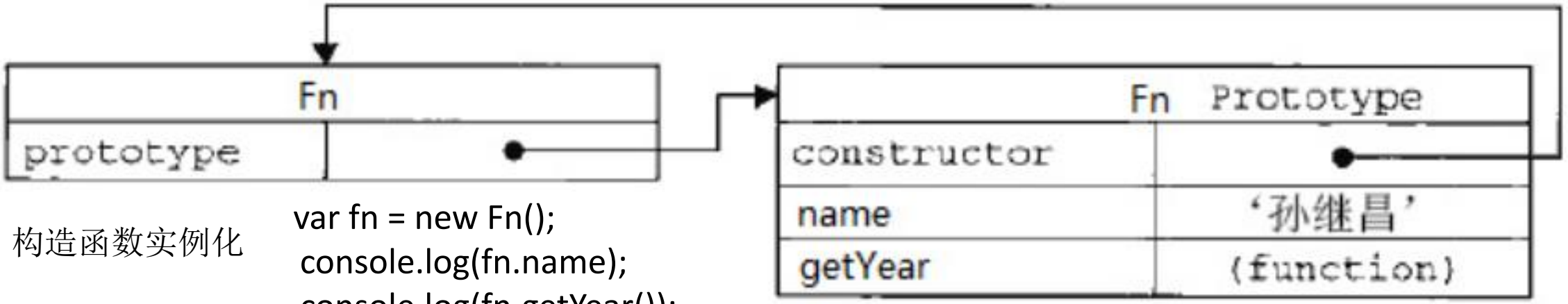
原型既然作为对象，属性的集合，不可能就只有一个constructor，肯定可以自定义的增加许多属性。例如Object，它的prototype里面，就有好几个其他属性。



你也可以在自己自定义的方法的prototype中新增自己的属性

构造函数

```
function Fn() { }
    Fn.prototype.name = '孙继昌';
    Fn.prototype.getYear = function () {
        return 1990;
    };
```



构造函数实例化

```
var fn = new Fn();
console.log(fn.name);
console.log(fn.getYear());
```

即，Fn是一个函数，fn对象是从Fn函数new出来的，这样fn对象就可以调用Fn.prototype中的属性。

每个对象都有一个隐藏的属性——“__proto__”，这个属性引用了创建这个对象的函数的prototype。
即：fn.__proto__ === Fn.prototype

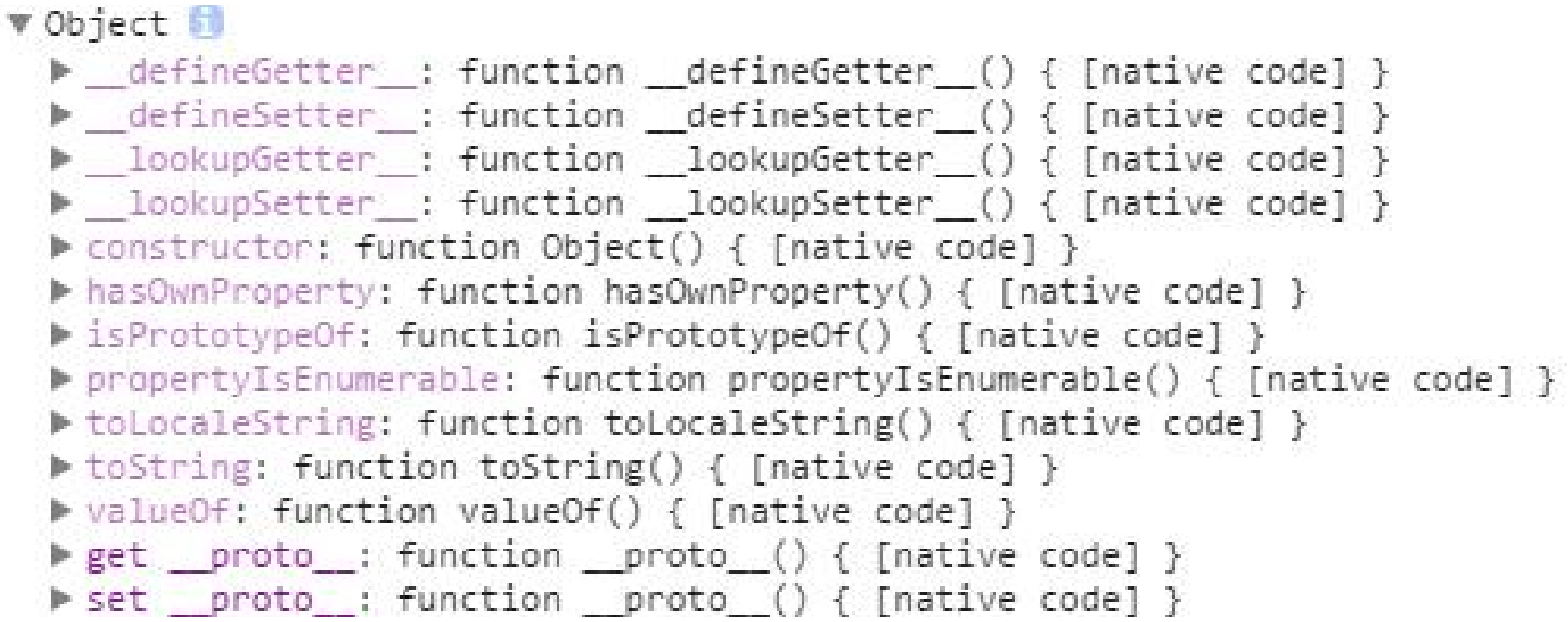
这里的"__proto__"成为“隐式原型”

隐式原型 __proto__

每个函数function都有一个prototype，即原型。
每个对象都有一个__proto__，可成为隐式原型。

注意：这个__proto__是一个隐藏的属性，javascript不希望开发者用到这个属性值，有的低版本浏览器甚至不支持这个属性值。但是你不用管它，直接写出来就是了。

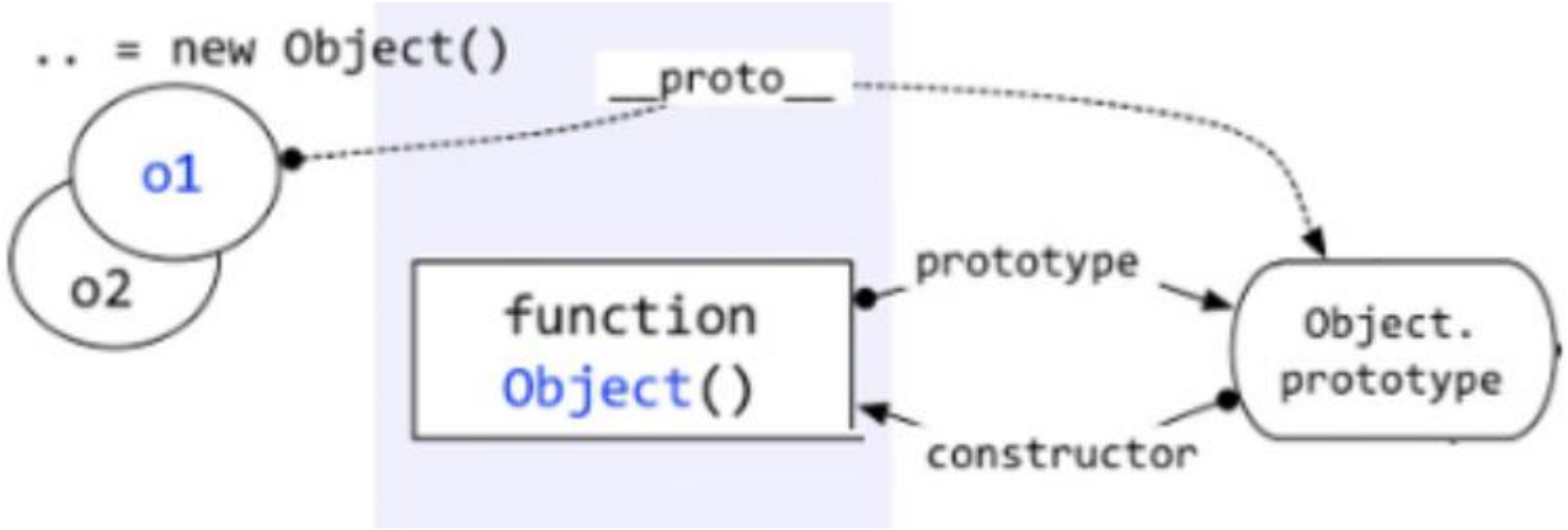
```
var obj = {};
console.log(obj.__proto
```



上面截图看来，obj.__proto__和Object.prototype的属性一样！这么巧！

答案就是一样。

obj这个对象本质上是被Object函数创建的，因此obj.__proto__===Object.prototype。我们可以用一个图来表示。

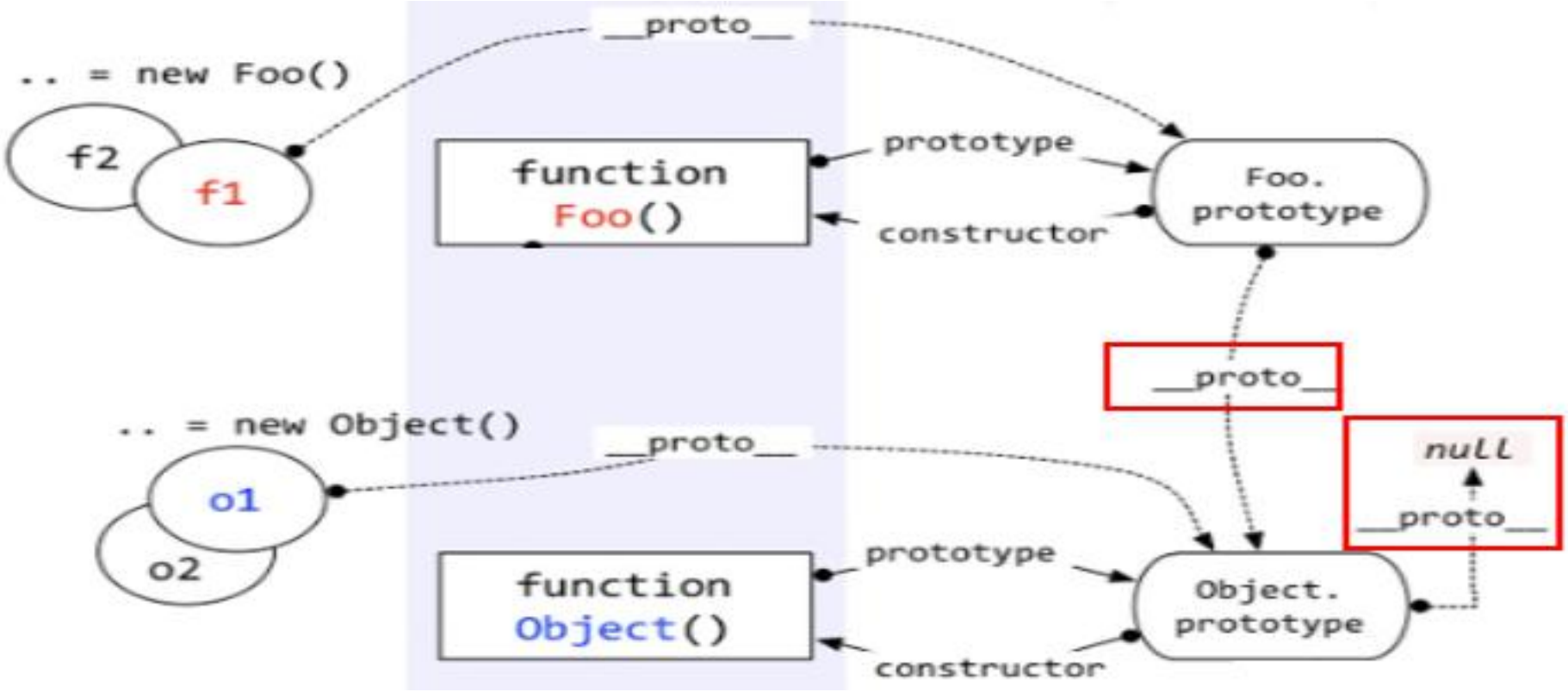


即，每个对象都有一个__proto__属性，指向创建该对象的函数的prototype。

那么上图中的“Object prototype”也是一个对象，它的__proto__指向哪里？

在说明“Object prototype”之前，先说一下自定义函数的prototype。自定义函数的prototype本质上就是和var obj = {} 是一样的，都是被Object创建，所以它的__proto__指向的就是Object.prototype。

但是Object.prototype确实一个特例——它的__proto__指向的是null，切记切记！



还有——函数也是一种对象，函数也有__proto__吗？

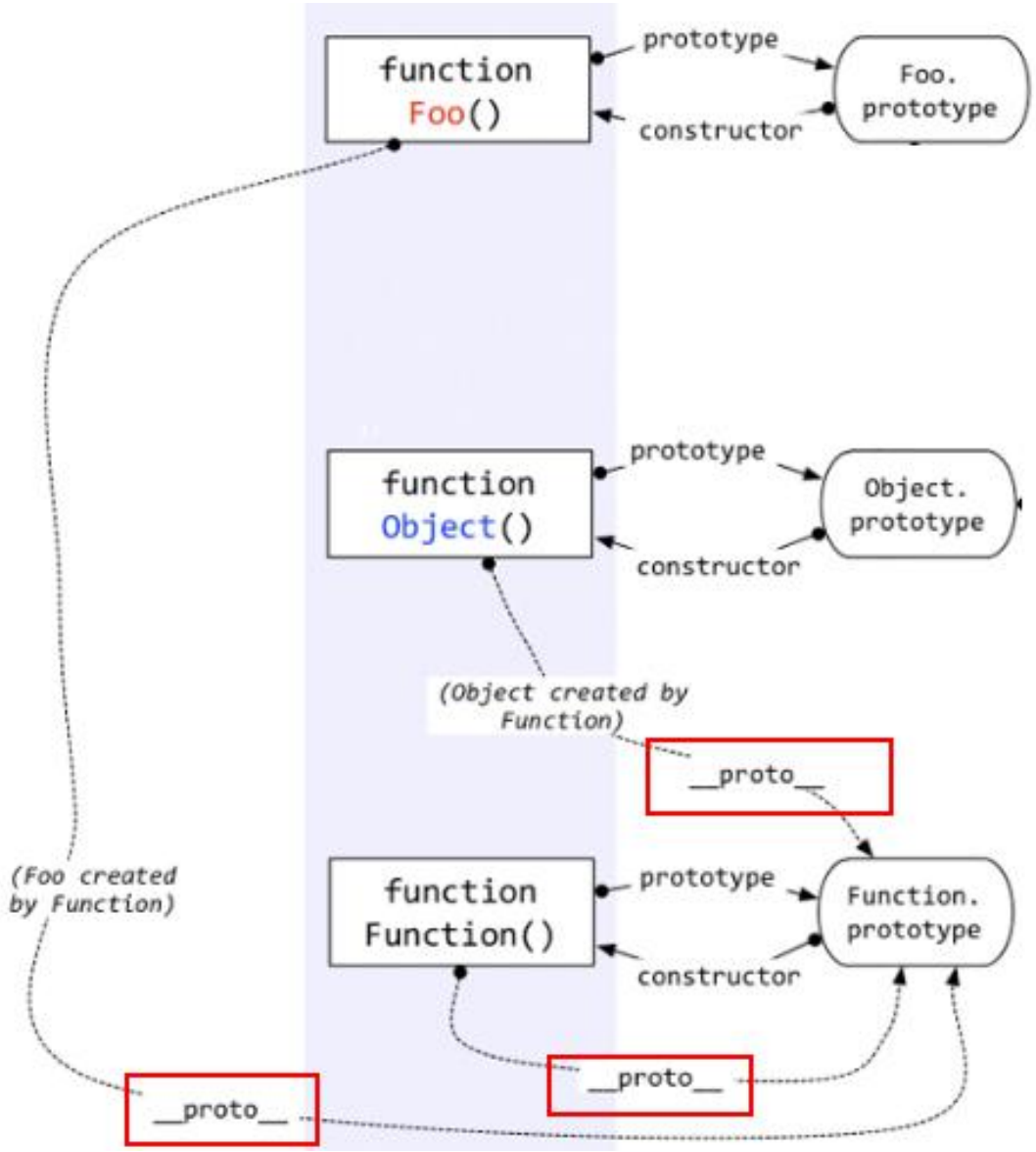
——当然有。

函数也是被创建出来的。谁创建了函数呢？——Function——注意这个大写的“F”。
且看如下代码。

```
function fn(x, y) {
    return x + y;
};
console.log(fn(10, 20));

var fn1 = new Function("x", "y", "return x + y;");
console.log(fn1(5, 6));
```

以上代码中，第一种方式是比较传统的函数创建方式，第二种是用new Functoin创建。
首先根本不推荐用第二种方式。
这里只是向大家演示，函数是被Function创建的

好了，根据上面说的一句话——对象的__proto__指向的是创建它的函数的prototype，就会出现：Object.__proto__ === Function.prototype。用一个图来表示。



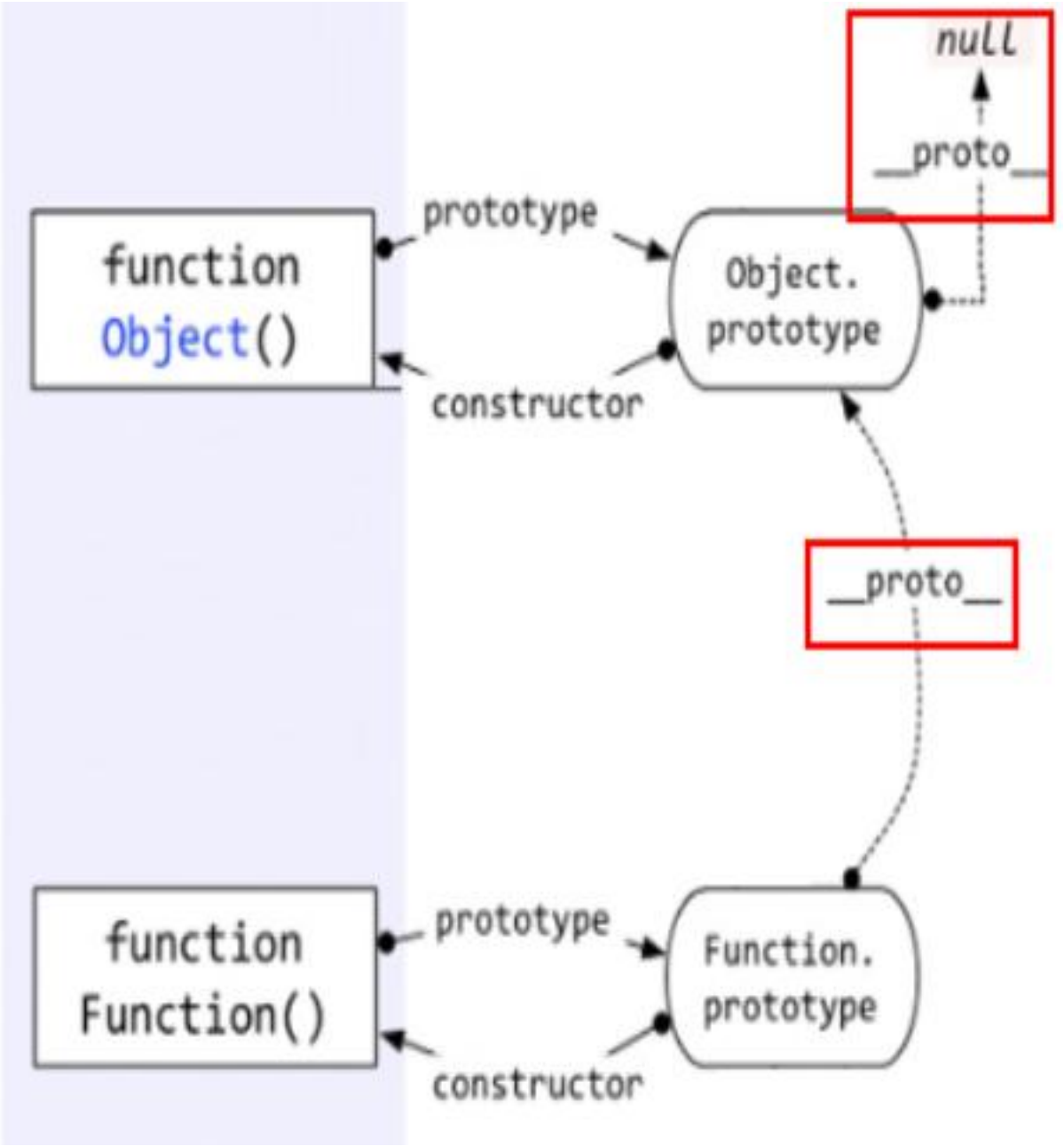
左图中，很明显的标出了：自定义函数Foo.__proto__指向Function.prototype，Object.__proto__指向Function.prototype，唉，怎么还有一个……Function.__proto__指向Function.prototype？这不成了循环引用了？

对！是一个环形结构。

其实稍微想一下就明白了。Function也是一个函数，函数是一种对象，也有__proto__属性。既然是函数，那么它一定是被Function创建。所以——Function是被自身创建的。所以它的__proto__指向了自身的Prototype。

最后一个问题：Function.prototype指向的对象，它的__proto__是不是也指向Object.prototype？

答案是肯定的。因为Function.prototype指向的对象也是一个普通的被Object创建的对象，所以也遵循基本的规则。



instanceof

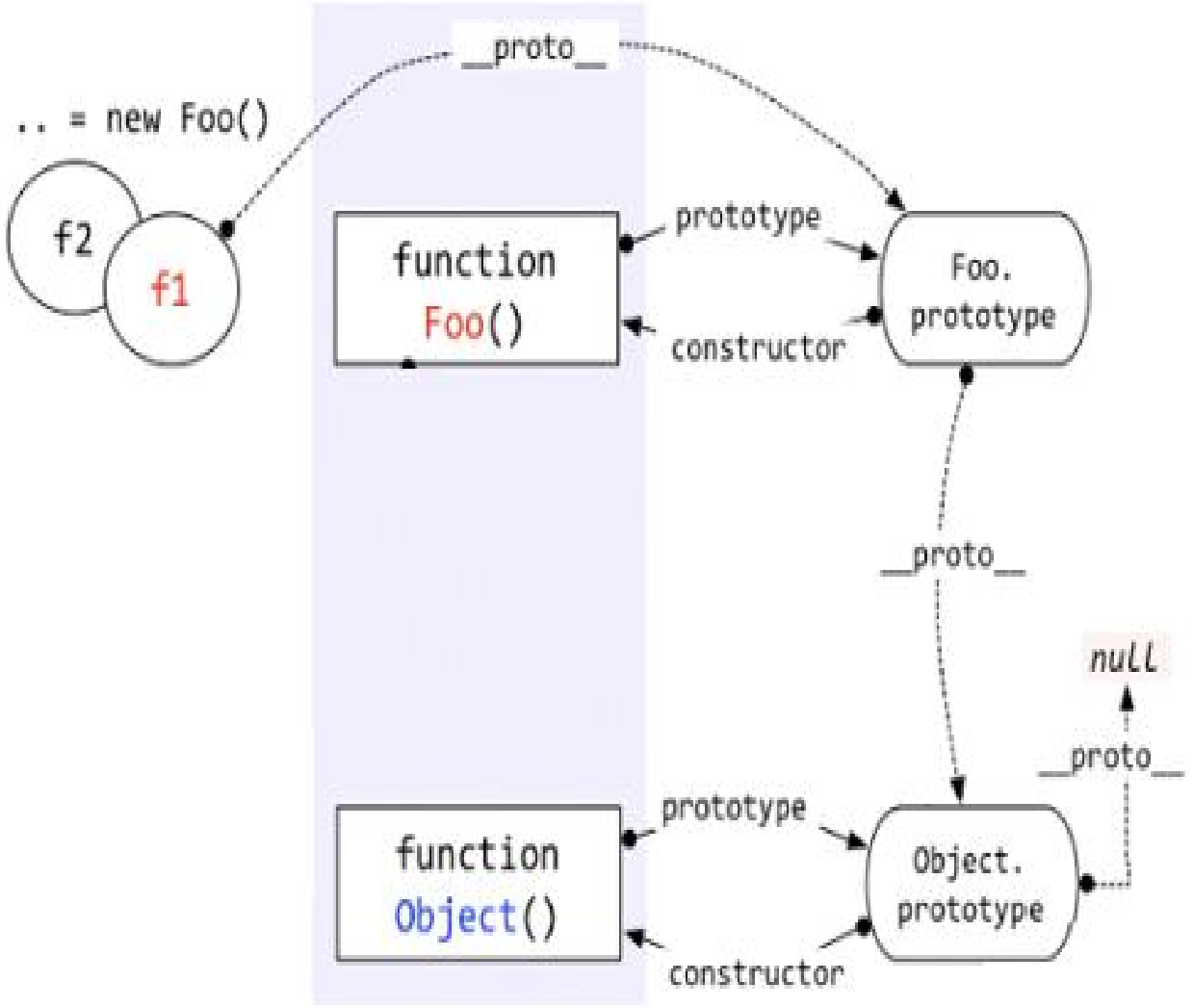
对于值类型，你可以通过typeof判断，string/number/boolean都很清楚，但是typeof在判断到引用类型的时候，返回值只有object/function，你不知道它到底是一个object对象，还是数组，还是new Number等等。

这个时候就需要用到instanceof。例如：

```
function Foo() { }
var f1 = new Foo();

console.log(f1 instanceof Foo);  // true
console.log(f1 instanceof Object);  // true
```

上图中，f1这个对象是被Foo创建，但是“f1 instanceof Object”为什么是true呢？

至于为什么过会儿再说，先把instanceof判断的规则告诉大家。根据以上代码看下图：



Instanceof运算符的第一个变量是一个对象，暂时称为A；第二个变量一般是一个函数，暂时称为B。

Instanceof的判断队则是：沿着A的__proto__这条线来找，同时沿着B的prototype这条线来找，如果两条线能找到同一个引用，即同一个对象，那么就返回true。如果找到终点还未重合，则返回false。

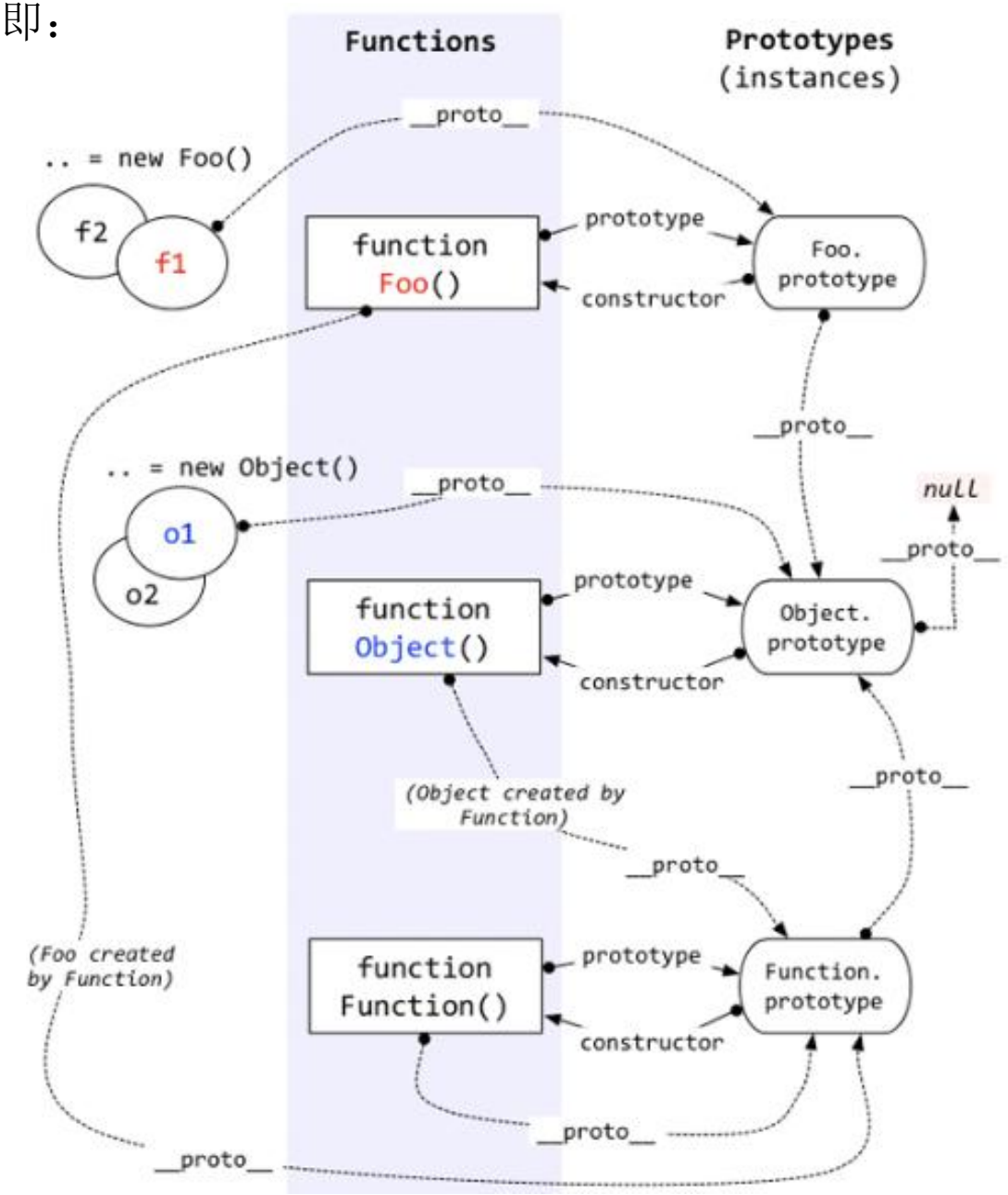
按照以上规则，大家看看“ f1 instanceof Object ”这句代码是不是true？ 根据上图很容易就能看出来，就是true。

通过上以规则，你可以解释很多比较怪异的现象，例如：

```
console.log(Object instanceof Function);   // true
console.log(Function instanceof Object);   // true
console.log(Function instanceof Function);  // true
```

这些看似很混乱的东西，答案却都是true

咱们贴了好多的图片，其实那些图片是可以联合成一个整体的，即：



问题又出来了。Instanceof这样设计，到底有什么用？到底instanceof想表达什么呢？

重点就这样被这位老朋友给引出来了——继承——原型链。

即，instanceof表示的就是一种继承关系，或者原型链的结构。

继承

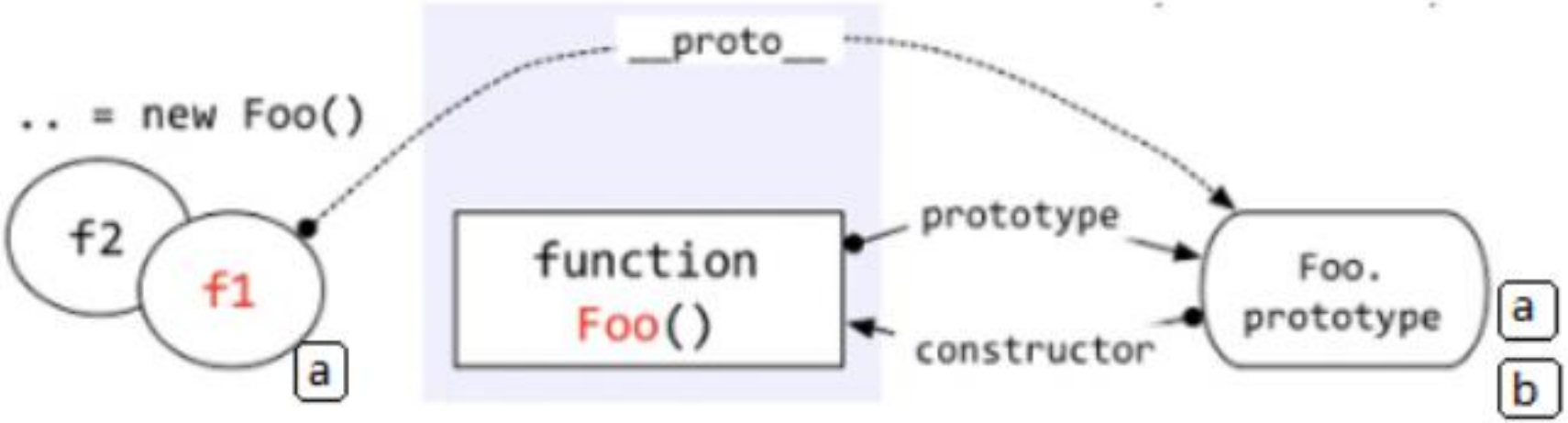
javascript中的继承是通过原型链来体现的。先看几句代码

```
function Foo() { }
var f1 = new Foo();

f1.a = 10;

Foo.prototype.a = 100;
Foo.prototype.b = 200;

console.log(f1.a);  //10
console.log(f1.b);  //200
```

以上代码中，f1是Foo函数new出来的对象，f1.a是f1对象的基本属性，f1.b是怎么来的呢？——从Foo.prototype得来，因为f1.__proto__指向的是Foo.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。

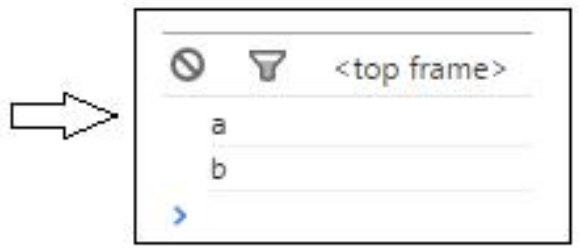
看图说话：



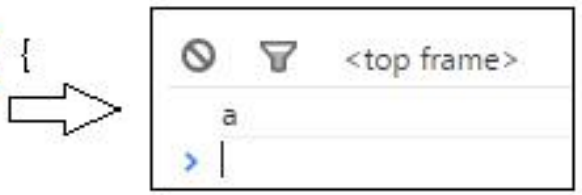
上图中，访问f1.b时，f1的基本属性中没有b，于是沿着__proto__找到了Foo.prototype.b。

那么我们在实际应用中如何区分一个属性到底是基本的还是从原型中找到的呢？大家可能都知道答案了——hasOwnProperty，特别是在for…in…循环中，一定要注意。

```
function Foo() { }
var f1 = new Foo();

f1.a = 10;

Foo.prototype.a = 100;
Foo.prototype.b = 200;

var item;
for (item in f1) {
    console.log(item);
}
```

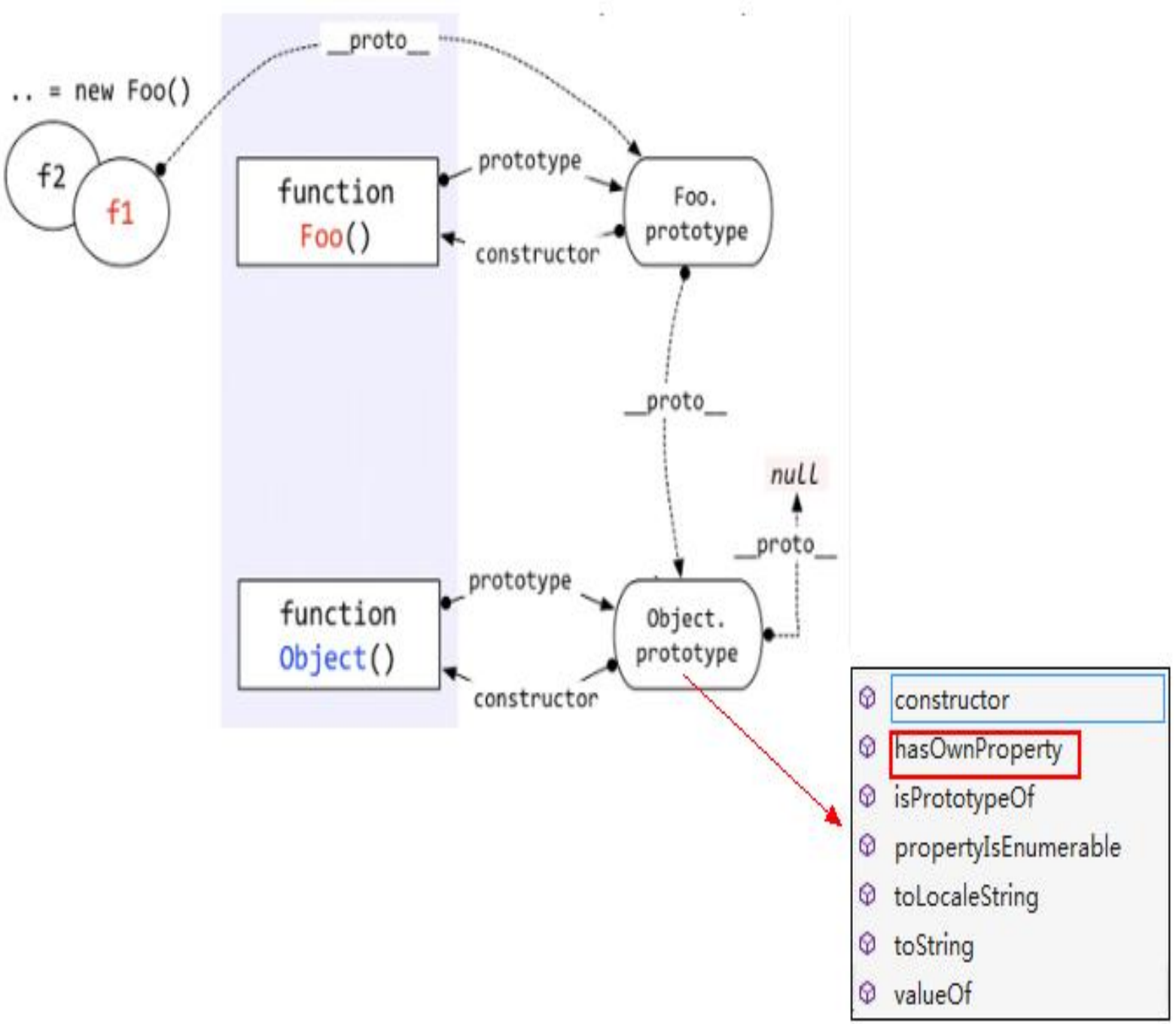



```
function Foo() { }
var f1 = new Foo();

f1.a = 10;

Foo.prototype.a = 100;
Foo.prototype.b = 200;

var item;
for (item in f1) {
    if (f1.hasOwnProperty(item)) {
        console.log(item);
    }
}
```



f1的这个hasOwnProperty方法是从哪里来的？ f1本身没有，Foo.prototype中也没有，哪儿来的？

它是从Object.prototype中来的，请看图：



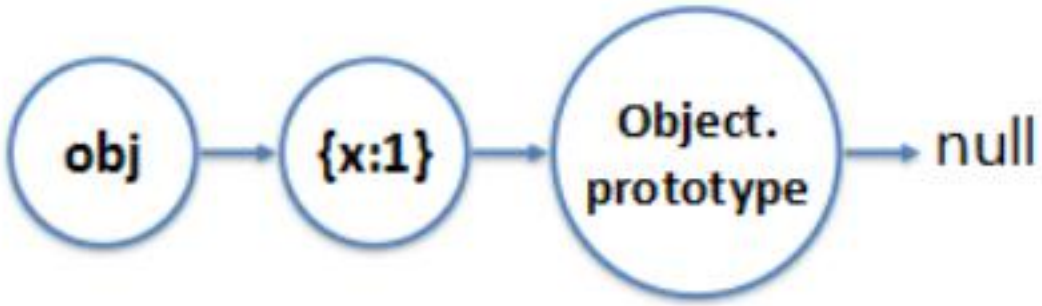
对象的原型链是沿着__proto__这条线走的，因此在查找f1.hasOwnProperty属性时，就会顺着原型链一直查找到Object.prototype。

由于所有的对象的原型链都会找到Object.prototype，因此所有的对象都会有Object.prototype的方法。这就是所谓的“继承”。

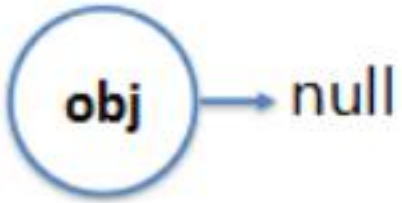
当然这只是一个例子，你可以自定义函数和对象来实现自己的继承。

创建对象 - Object.create

```
var obj = Object.create({x : 1});
obj.x // 1
typeof obj.toString // "function"
obj.hasOwnProperty('x');// false
```




```
var obj = Object.create(null);
obj.toString // undefined
```



创建一个空对象,让它的原型指向它的参数

全局作用域 this

```
console.log(this === window); // true

this.a = 37;
console.log(window.a); // 37
```

一般函数this

```
function f1(){
  return this;
}

f1() === window; // true, global
```

```
function f2(){
  "use strict"; // see strict mode
  return this;
}

f2() === undefined; // true
```

作为对象方法的函数的this

```
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};

console.log(o.f()); // logs 37
```

```
var o = {prop: 37};

function independent() {
  return this.prop;
}

o.f = independent;

console.log(o.f()); // logs 37
```

对象原型链上的this

```
var o = {f:function(){ return this.a + this.b; }};
var p = Object.create(o);
p.a = 1;
p.b = 4;

console.log(p.f()); // 5
```

构造器中的this

```
function MyClass(){
  this.a = 37;
}

var o = new MyClass();
console.log(o.a); // 37


function C2(){
  this.a = 37;
  return {a : 38};
}

o = new C2();
console.log(o.a); // 38
```

call/apply 方法与this

```
function add(c, d){
  return this.a + this.b + c + d;
}

var o = {a:1, b:3};

add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16

add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34

function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // "[object Number]"
```

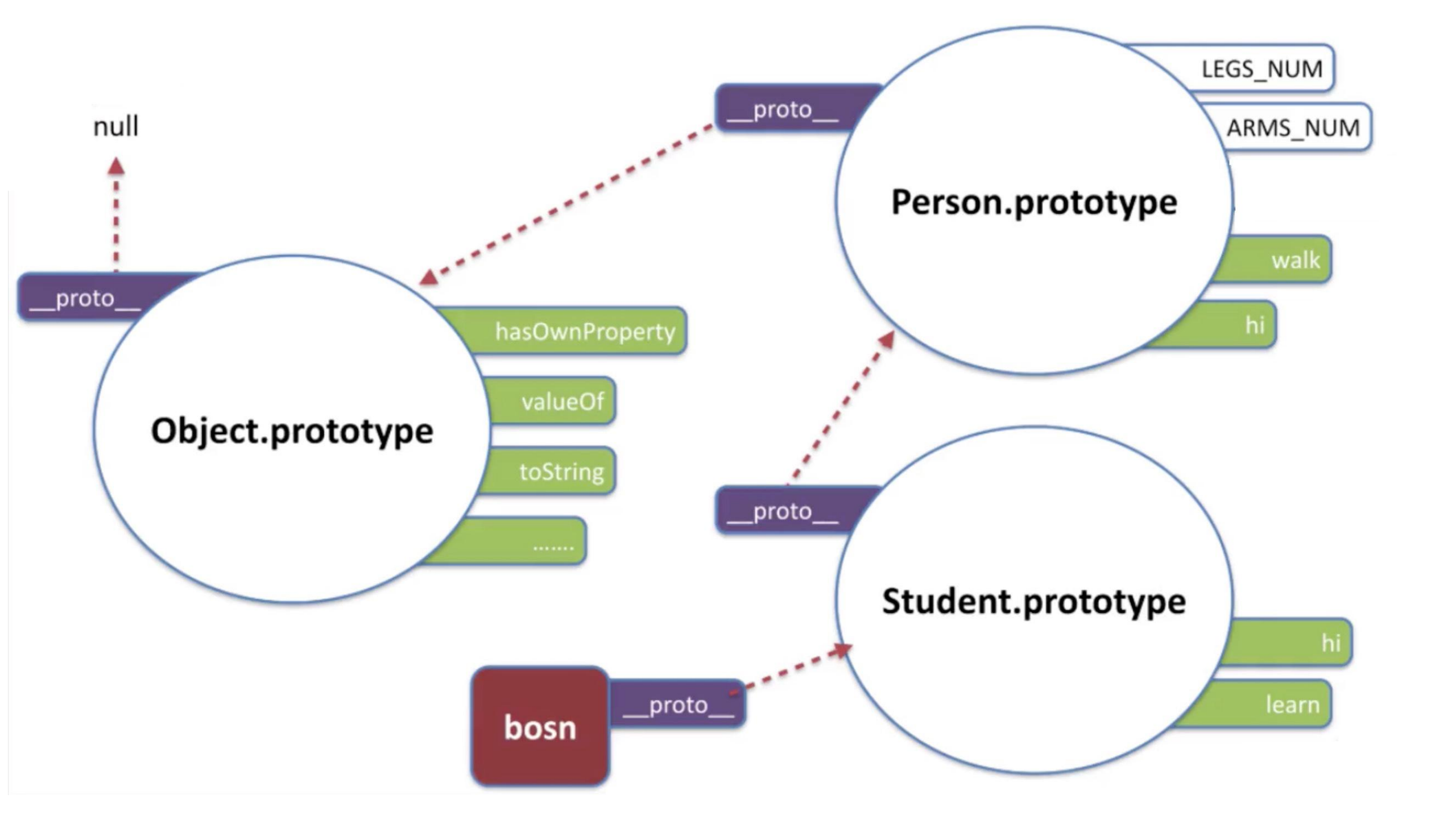
bind方法与this

```
function f(){
  return this.a;
}

var g = f.bind({a : "test"});
console.log(g()); // test

var o = {a : 37, f : f, g : g};
console.log(o.f(), o.g()); // 37, test
```

举个例子

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.hi = function() {
    console.log('Hi, my name is ' + this.name + ", I'm " +
    this.age + " years old now.");
};

Person.prototype.LEGS_NUM = 2;
Person.prototype.ARMS_NUM = 2;
Person.prototype.walk = function() {
    console.log(this.name + " is walking...");
};

function Student(name, age, className) {
    Person.call(this, name, age);
    this.className = className;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;
```

```
Student.prototype.hi = function() {
    console.log('Hi, my name is ' + this.name + ", I'm " +
        this.age + " years old now, and from " +
        this.className + ".");
};

Student.prototype.learn = function(subject) {
    console.log(this.name + ' is learning ' + subject +
        ' at ' + this.className + '.');
};

// test
var bosn = new Student('Bosn', 27, 'Class 3, Grade 2');
bosn.hi(); // Hi, my name is Bosn, I'm 27 years old now, and from Class 3, Grade 2.
bosn.LEGS_NUM; // 2
bosn.walk(); // Bosn is  walking...
bosn.learn('math'); // Bosn is learning math at Class 3, Grade 2.
```


null
__proto__
Object.prototype
hasOwnProperty
valueOf
toString
.....
__proto__
Person.prototype
LEGS_NUM
ARMS_NUM
walk
hi
__proto__
Student.prototype
hi
learn
bosn
__proto__

谢谢