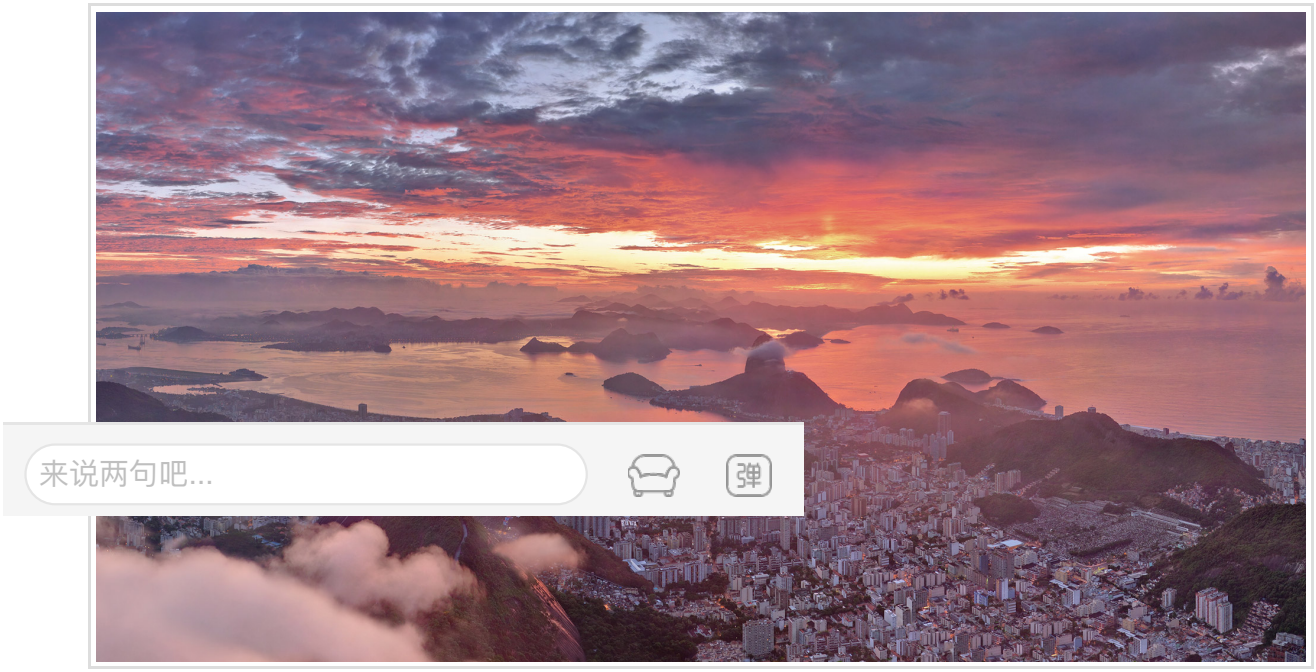
JavaScript深入理解之原型与原型链

2018-08-03 | 前端 , JavaScript



写在前面

原型和原型链一直都是 JavaScript 中很重要的概念，理解它们有助于我们理解预定义引用类型间的关系以及 JavaScript 中对象继承的实现机制，下面是我对原型和原型链的理解和总结。

原型

原型对象理解

函数对象的 prototype 属性

我们创建的每一个函数都有一个 prototype 属性，这个属性是一个指针，指向一个对象。这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法，简单来说，该函数实例化的所有对象的__proto__的属性指向这个对象，它是该函数所有实例化对象的原型。

```
function Person(){

}

// 为原型对象添加方法
Person.prototype.sayName = function(){
    alert(this.name);
}
```

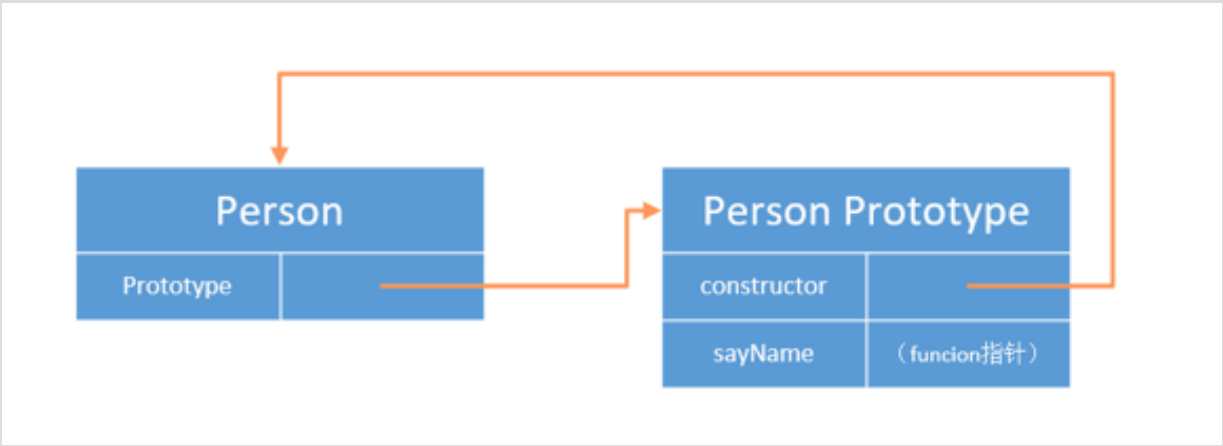
下面我们来看一下它们之间的关系。

constructor 属性

当函数创建，prototype 属性指向一个原型对象时，在默认情况下，这个原型对象将会获得一个 constructor 属性，这个属性是一个指针，指向 prototype 所在的函数对象。

拿前面的一个例子来说 Person.prototype.constructor 就指向 Person 函数对象。

下面我们来更新一下它们之间的关系图。



对象的 __proto__ 属性

当我们调用构造函数创建一个新实例后，在这个实例的内部将包含一个指针，指向构造函数的原型对象，在 ECMA-262 第五版中管这个指针叫做 [[Prototype]] 。所有的对象都含有 [[Prototype]] 属性，指向它们的原型对象，这是理解原型链的一个关键概念。

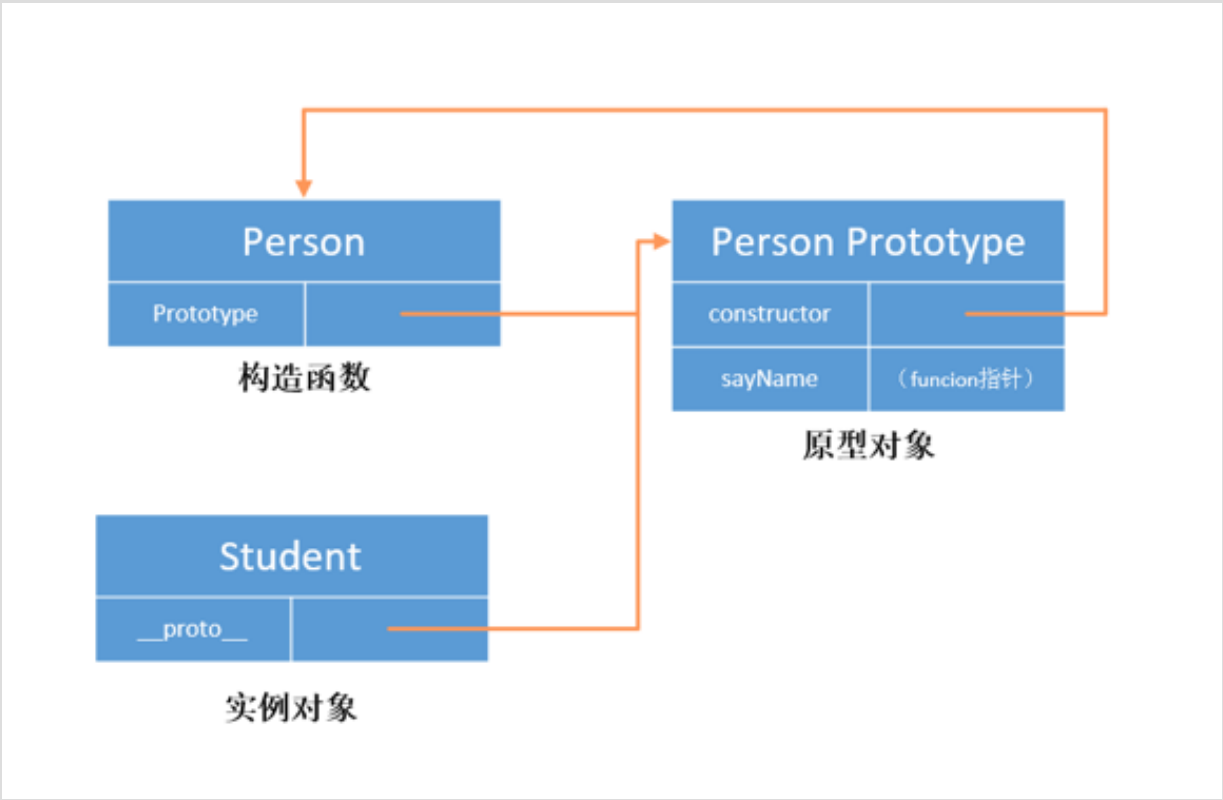
需要注意的是在脚本中没有标准的方式访问 [[Prototype]] 属性，但是在 Firefox，Safari 和 Chrome 中每个对象中都支持一个属性 __proto__ 来访问，为了区分 prototype 属性，我们在下边都使用 __proto__来表示。

根据前面的 Person 构造函数我们新建一个实例

```
var student = new Person();

console.log(student.__proto__ === Person.prototype); // true
```

从上面我们可以看出，这个连接是存在与实例与构造函数的原型对象之间的，而不是存在于实例和构造函数之间的。

下面我们来看一下现在这几个对象之间的关系



isPrototypeOf()

虽然我们在脚本中没有办法访问到[[Prototype]]属性，但是我们可以通过 isPrototypeOf 方法来确定对象之间是否存在这种关系。

isPrototypeOf() 方法用于测试一个对象是否存在于另一个对象的原型链上。

```
1  console.log(Person.prototype.isPrototypeOf(student)); // true
```

Object.getPrototypeOf()

在 ECMAScript 5 中新增了一个方法叫 Object.getPrototypeOf()，这个方法可以返回[[Prototype]]的值，如下

```
1  console.log(Object.getPrototypeOf(student) === Person.prototype); //
```

原型属性

属性访问

每当代码读取对象的某个属性时，首先会在对象本身搜索这个属性，如果找到该属性就返回该属性的值，如果没有找到，则继续搜索该对象对应的原型对象，以此类推下去。

因为这样的搜索过程，因此我们如果在实例中添加一个属性时，这个属性就会屏蔽原型对象中保存的同名属性，因为在实例中搜索到该属性后就不会再向后搜索了。

属性判断

既然一个属性既可能是实例本身的，也有可能是其原型对象的，那么我们该如何来判断呢?

在属性确认存在的情况下，我们可以使用 hasOwnProperty() 方法来判断一个属性是存在与实例中，还是存在于原型中。注意这个方法只有在给定属性存在于实例中时，才会返回 true 。

```
function Person() {};

Person.prototype.name = "laker" ;

var student = new Person();

console.log(student.name); // laker
console.log(student.hasOwnProperty("name")); // false


student.name = "xiaoming";
console.log(student.name); //xiaoming 屏蔽了原型对象中的 name 属性
console.log(student.hasOwnProperty("name")); // true
```

上面主要是针对属性确认存在的情况下，如果这个属性不一定存在的话，这样判断就不够准确，因此我们需要首先判断这个属性是否存在，然后再进行上面的判断操作。

判断一个属性是否存在，我们可以使用 in 操作符，它会在对象能够访问给定属性时返回 true，无论该属性存在于实例还是原型中。

因此我们可以封装这样一个函数，来判断一个属性是否存在于原型中

```
function hasPrototypeProperty(object, name){
    return !object.hasOwnProperty(name) && (name in object);
}
```

for–in 循环

在使用 for–in 循环时，返回的是所有能够通过对象访问的、可枚举的属性，其中包括了存在于实例中的属性，也包括了存在于原型中的属性。

需要注意的一点是，屏蔽了实例中不可枚举属性的实例属性也会在 for–in 循环中返回。

所有属性获取

如果想要获得对象上所有可枚举的实例属性，可以使用 Object.keys() 方法，这个方法接收一个对象作为参数，返回一个包含所有可枚举属性的字符串数组。

如果想要获取所有的实例属性，无论它是否可以枚举，我们可以使用 Object.getOwnPropertyNames() 方法。

原型链

原型链理解

ECMAScript 中描述了原型链的概念，并将原型链作为实现继承的主要方法。其基本思想是利用的一个引用类型继承另一个引用类型的属性和方法。

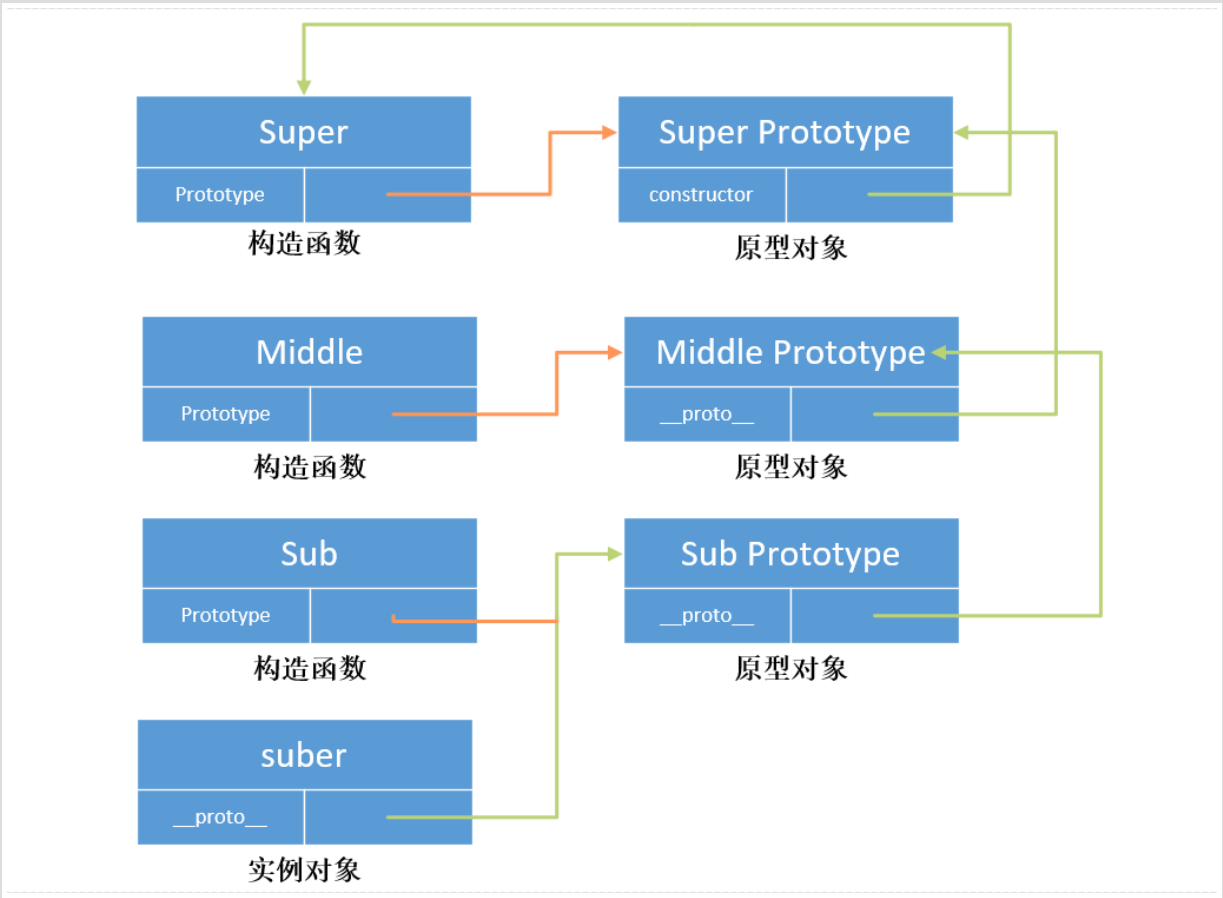
原型链的主要实现方法是让构造函数的 prototype 对象等于另一个类型的实例，此时的 prototype 对象因为是实例，因此将包含一个指向另一个原型的指针，相应地另一个原型中也包含着一个指向另一个构造函数的指针。假如另一个原型又是另一个类型的实例，那么上述关系依然成立，如此层层递进，就构成了实例与类型的链条。这就是原型链的基本概念。

下面我们来看一个例子

```
function Super(){

};


function Middle(){

};

```

```
function Sub(){

};

Middle.prototype = new Super();

Sub.prototype = new Middle();

var suber = new Sub();
```

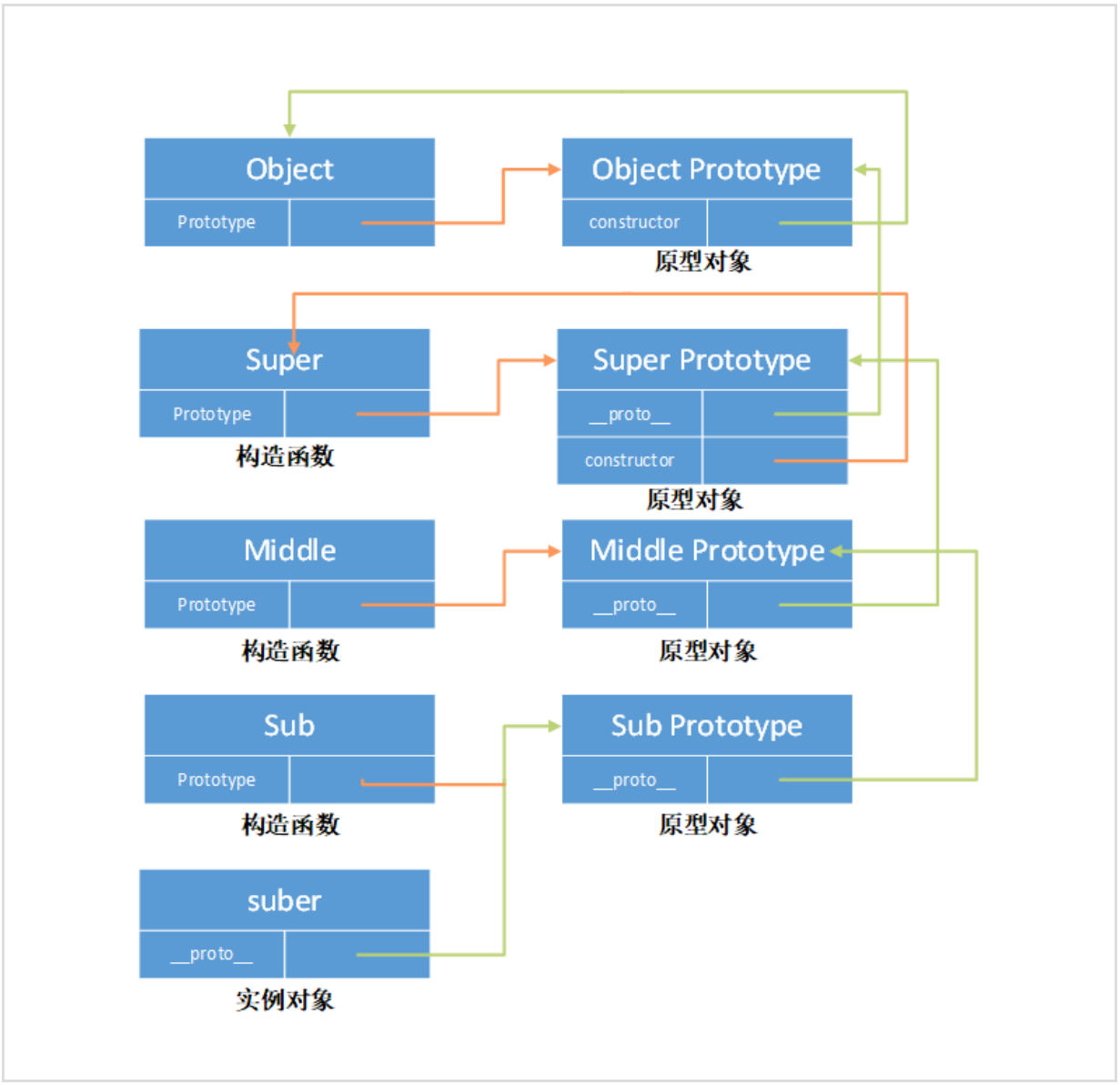
下面我们来看看这几个对象间的关系



默认的原型

其实我们上面这个原型链是不完整的，还记得我们以前说过所有的引用类型都继承了 Object 吗？这个继承就是通过原型链来实现的。我们一定要记住，所有函数的默认原型都是 Object 的实例，因此默认原型都会包含一个内部指针，指向 Object.Prototype 。这也正是所有的自定义类型都会继承 toString() 、valueOf() 等默认方法的根本原因。

那么我更新一下我们上面的原型链



Object.prototype 就是原型链的终点了，我们可以试着打印一下 Object.prototype.__proto__，我们会发现返回的是一个 null 空对象，这就意味着原型链的结束。

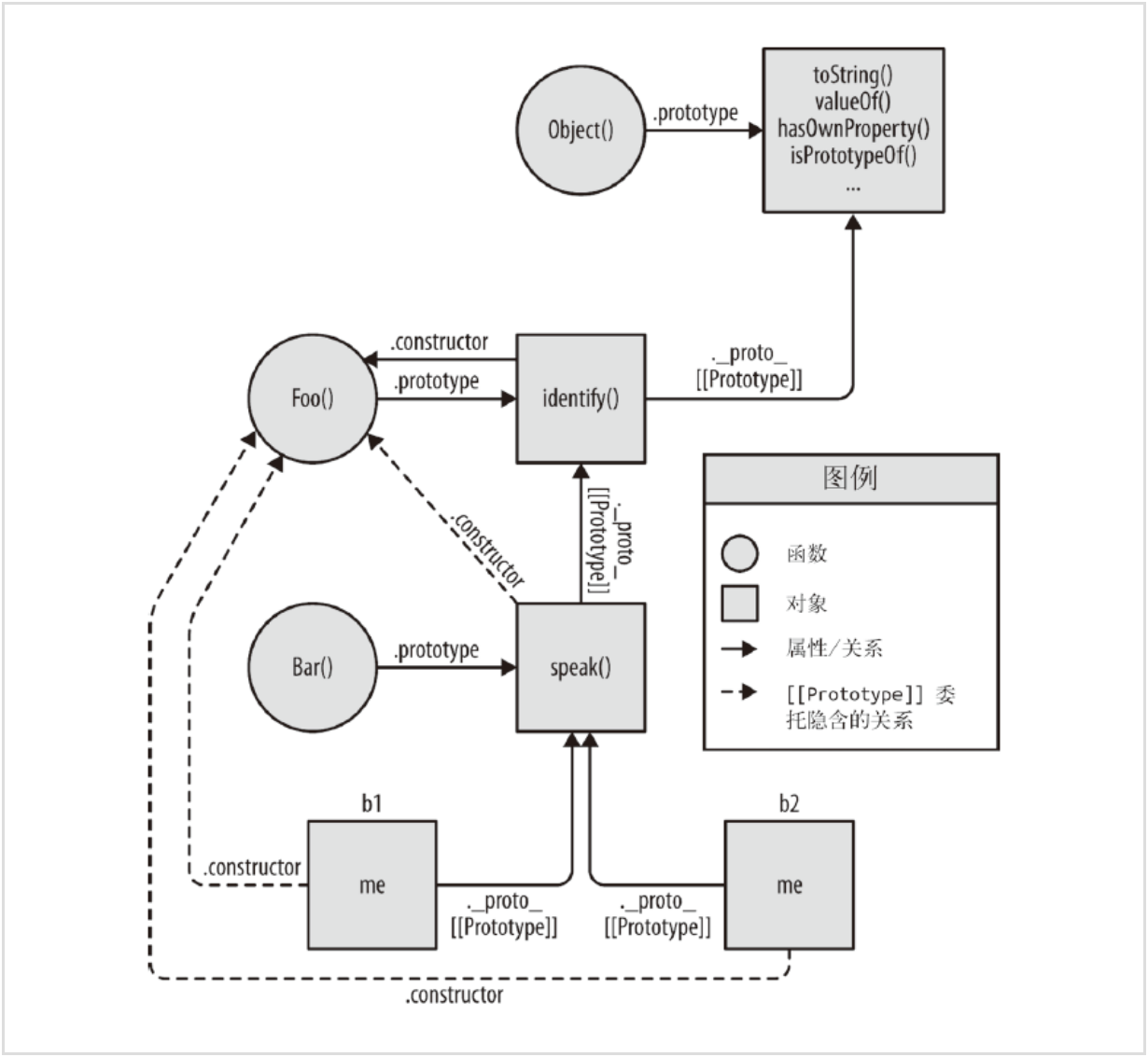
函数也是对象

其实上面的原型链还少了一部分，为什么呢。因为我们知道在 JavaScript 中，函数也是对象，那么函数也应该有它的原型对象。下面我们通过一个实例来了解它们之间的关系。

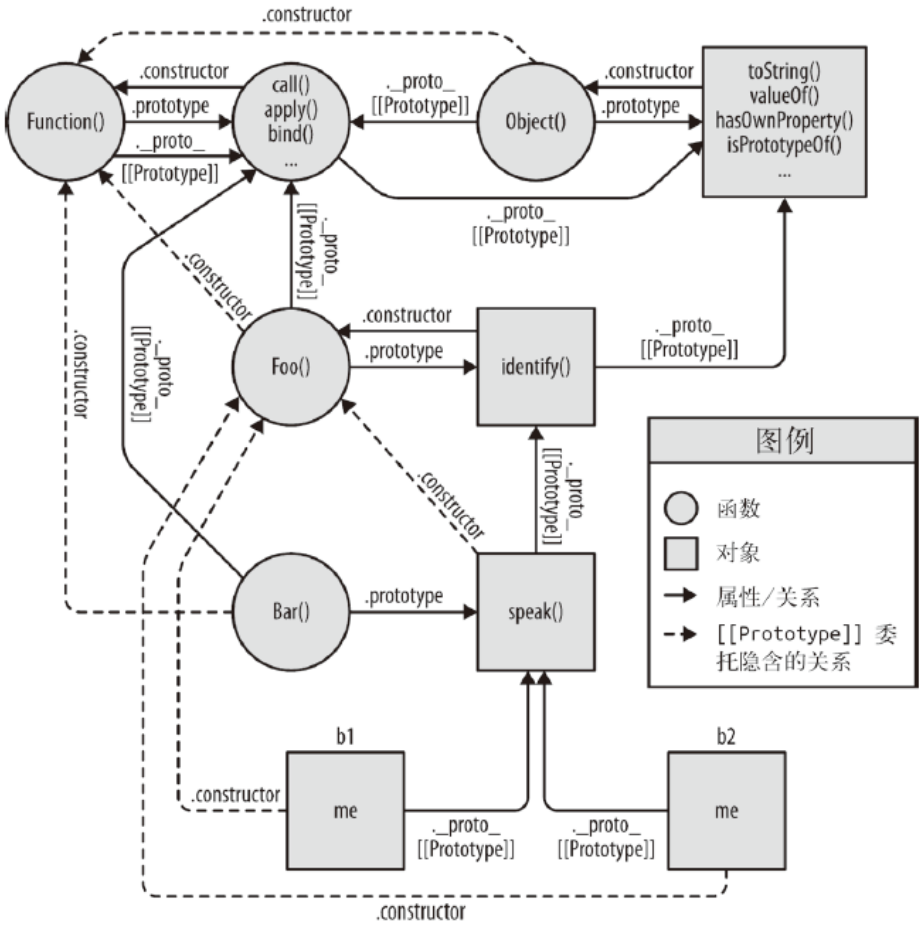
实例代码

```
function Foo(who) {
this.me = who;
}

Foo.prototype.identify = function() {
return "I am " + this.me;
};

function Bar(who) {
Foo.call( this, who );
}

Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

我们先看一下不包含函数原型的以上实例所包含的原型链的关系图



下面是加入了函数原型后的原型链的关系图。



写在最后

总结了原型和原型链的知识后，感觉对 JavaScript 语言的理解更加深刻了，也为后面理解对象的创建和继承打下了基础。其实理解原型链，对于 JavaScript 中一些预定义类型的行为和实现就很好理解了。

本篇文章纯属于个人的学习总结，如果文章中出现错误或不严谨的地方，希望大家能够指出，谢谢！

前端 JavaScript 原型链 原型

JavaScript深入理解之引用类型详解 JavaScript深入理解之对象创建