36 | 编程范式游记（7）- 基于原型的编程范式

2018-2-1 陈皓

基于原型（Prototype）的编程其实也是面向对象编程的一种方式。没有class化的，直接使用对象。又叫，基于实例的编程。其主流的语言就是JavaScript，与传统的面对象编程的比较如下：

> 在基于类的编程当中，对象总共有两种类型。类定义了对象的基本布局和函数特性，而接口是“可以使用的”对象，它基于特定类的样式。在此模型中，类表现为行为和结构的集合，对所有接口来说这些类的行为和结构都是相同的。因而，区分规则首先是基于行为和结构，而后才是状态。
>
> 原型编程的主张者经常争论说，基于类的语言提倡使用一个关注分类和类之间关系的开发模型。与此相对，原型编程看起来提倡程序员关注一系列对象实例的行为，而之后才关心如何将这些对象划分到最近的使用方式相似的原型对象，而不是分成类。

因为如此，很多基于原型的系统提倡运行时进行原型的修改，而只有极少数基于类的面向对象系统（比如第一个动态面向对象的系统Smalltalk）允许类在程序运行时被修改。

> 在基于类的语言中，一个新的实例通过类构造器和构造器可选的参数来构造，结果实例由类选定的行为和布局创建模型。
>
> 在基于原型的系统中构造对象有两种方法，通过复制已有的对象或者通过扩展空对象创建。很多基于原型的系统提倡运行时进行原型的修改，而基于类的面向对象系统只有动态语言允许类在运行时被修改（Common Lisp、Dylan、Objective-C、Perl、Python、Ruby和Smalltalk）。

JavaScript的原型概念

这里，我们主要以JavaScript举例，面向对象里面要有个Class。但是JavaScript觉得不是这样的，它就是要基于原型编程，就不要Class，就直接在对象上改就行了，基于编程的修改，直接对类型进行修改。

我们先来看一个示例。

```
var foo = {name: "foo", one: 1, two: 2};

var bar = {three: 3};
```

每个对象都有一个 `__proto__` 的属性，这个就是“原型”。对于上面的两个对象，如果我们把 `foo` 赋值给 `bar.__proto__`，那就意味着，`bar` 的原型就成了 `foo`的。

```
bar.__proto__ = foo; // foo is now the prototype of bar.
```

于是，我们就可以在 `bar` 里面访问 `foo` 的属性了。

```
// If we try to access foo's properties from bar
// from now on, we'll succeed.
bar.one // Resolves to 1.

// The child object's properties are also accessible.
bar.three // Resolves to 3.

// Own properties shadow prototype properties
bar.name = "bar";
foo.name; // unaffected, resolves to "foo"
bar.name; // Resolves to "bar"
```

需要解释一下JavaScript的两个东西，一个是 `__proto__`，另一个是 `prototype`，这两个东西很容易混淆。这里说明一下：

`__proto__` 主要是安放在一个实际的对象中，用它来产生一个链接，一个原型链，用于寻找方法名或属性，等等。

`prototype` 是用 `new` 来创建一个对象时构造 `__proto__` 用的。它是构造函数的一个属性。

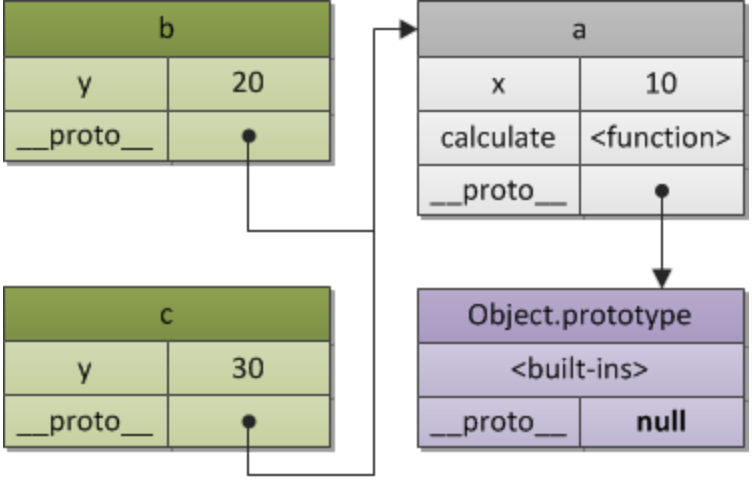
在JavaScript中，对象有两种表现形式， 一种是 Object (ES5关于Object的文档)，一种是 Function （ ES5关于Function的文档 ）。

我们可以简单地认为， __proto__ 是所有对象用于链接原型的一个指针，而 prototype 则是 Function 对象的属性，其主要是用来当需要new一个对象时让 __proto__ 指针所指向的地方。 对于超级对象 Function 而言， Function.__proto__ 就是 Function.prototype。

比如我们有如下的代码：

```
var a = {
  x: 10,
  calculate: function (z) {
    return this.x + this.y + z;
  }
};

var b = {
  y: 20,
  __proto__: a
};

var c = {
  y: 30,
  __proto__: a
};

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80
```

其中的“原型链”如下所示：



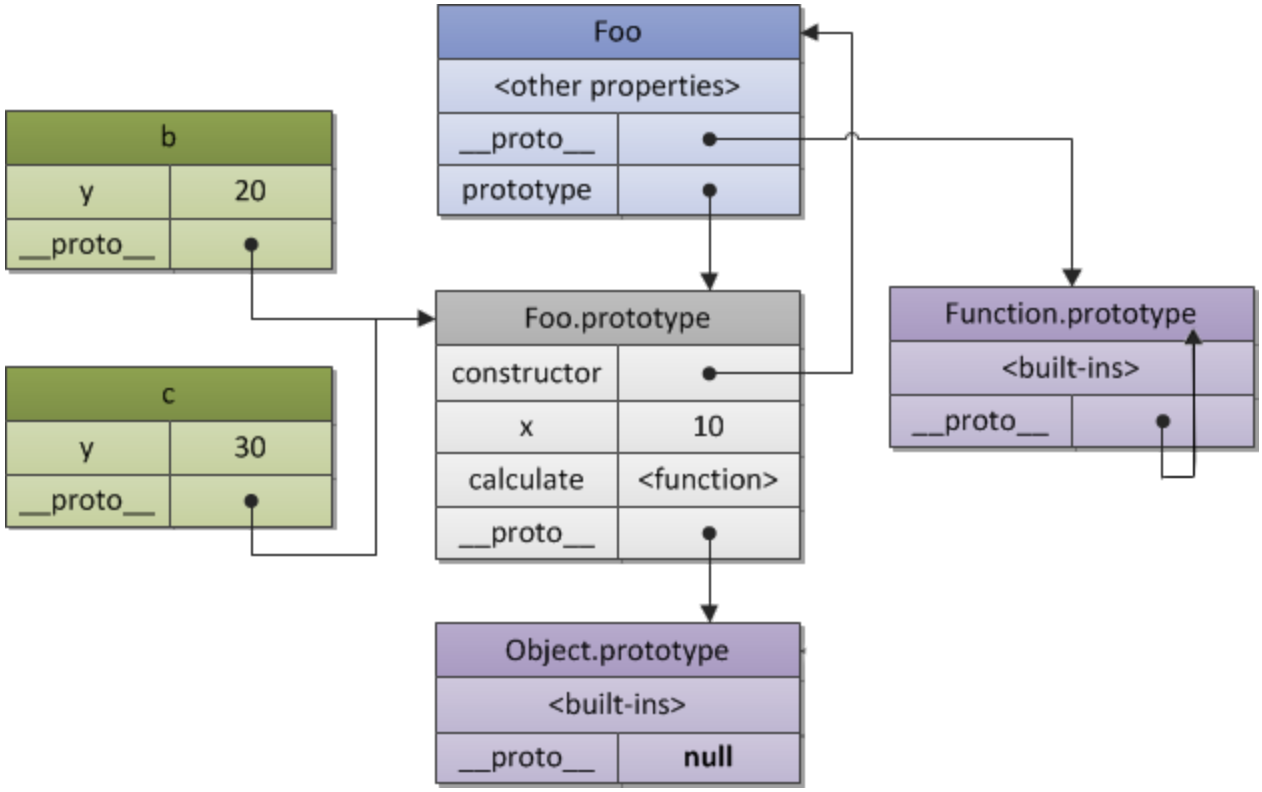
注意：ES5 中，规定原型继承需要使用 `Object.create()` 函数。如下所示：

```
var b = Object.create(a, {y: {value: 20}});
var c = Object.create(a, {y: {value: 30}});
```

好了，我们再来看一段代码：

```
// 一种构造函数写法
function Foo(y) {
  this.y = y;
}

// 修改 Foo 的 prototype，加入一个成员变量 x
Foo.prototype.x = 10;

// 修改 Foo 的 prototype，加入一个成员函数 calculate
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// 现在，我们用 Foo 这个原型来创建 b 和 c
var b = new Foo(20);
var c = new Foo(30);

// 调用原型中的方法，可以得到正确的值
b.calculate(30); // 60
c.calculate(40); // 80
```

那么，在内存中的布局是怎么样的呢？大概是下面这个样子。



这个图应该可以让你很好地看明白 `__proto__` 和 `prototype` 的差别了。

我们可以测试一下：

```
b.__proto__ === Foo.prototype, // true
c.__proto__ === Foo.prototype, // true

b.constructor === Foo, // true
c.constructor === Foo, // true
Foo.prototype.constructor === Foo, // true

b.calculate === b.__proto__.calculate, // true
b.__proto__.calculate === Foo.prototype.calculate // true
```

这里需要说明的是：

`Foo.prototype` **自动创建了一个属性** `constructor`**，这是一个指向函数自己的一个 reference。这样一来，对于实例** `b` **或** `c` **来说，就能访问到这个继承的** `constructor` **了。**

有了这些基本概念，我们就可以讲一下JavaScript的面向对象编程了。

> 注： 上面示例和图示来源于 JavaScript, The Core 一文。

JavaScript原型编程的面向对象

我们再来重温一下上面讲述的内容：

```
function Person(){}
var p = new Person();

Person.prototype.name = "Hao Chen";
Person.prototype.sayHello = function(){
    console.log("Hi, I am " + this.name);
}

console.log(p.name); // "Hao Chen"
p.sayHello(); // "Hi, I am Hao Chen"
```

在上面这个例子中：

- 我们先生成了一个空的函数对象 `Person()`；
- 然后将这个空的函数对象 `new` 出另一个对象，存在 `p` 中；
- 这时再改变 `Person.prototype`，让其有一个 `name` 的属性和一个 `sayHello()` 的方法；
- 我们发现，另外那个 `p` 的对象也跟着一起改变了。

注意一下：

- 当创建 `function Person(){}` 时，`Person.__proto__` 指向 `Function.prototype`;
- 当创建 `var p = new Person()` 时，`p.__proto__` 指向 `Person.prototype`;
- 当修改了 `Person.prototype` 的内容后，`p.__proto__` 的内容也就被改变了。

好了，我们再来看一下“原型编程”中面向对象的编程玩法。

首先，我们定义一个 `Person` 类。

```
//Define human class
var Person = function (fullName, email) {
  this.fullName = fullName;
  this.email = email;

  this.speak = function(){
    console.log("I speak English!");
  };
  this.introduction = function(){
    console.log("Hi, I am " + this.fullName);
  };
}
```

上面这个对象中，包含了：

属性： `fullName` 和 `email`；

方法： `speak()` 和 `introduction()`。

其实，所谓的方法也是属性。

然后，我们可以定义一个 `Student` 对象。

```
//Define Student class
var Student = function(fullName, email, school, courses) {

  Person.call(this, fullName, email);

  // Initialize our Student properties
  this.school = school;
  this.courses = courses;

  // override the "introduction" method
  this.introduction= function(){
        console.log("Hi, I am " + this.fullName +
                                ". I am a student of " + this.school +
```

```
                                ", I study "+ this.courses +".");
  };

  // Add a "exams" method
  this.takeExams = function(){
    console.log("This is my exams time!");
  };
};
```

在上面的代码中：

使用了 `Person.call(this, fullName, email)`，`call()` 或 `apply()` 都是为了动态改变 `this` 所指向的对象的内容而出现的。这里的 `this` 就是 `Student`。

上面的例子中，我们重载了 `introduction()` 方法，并新增加了一个 `takeExams()`的方法。

虽然，我们这样定义了 `Student`，但是它还没有和 `Person` 发生继承关系。为了要让它们发生关系，我们就需要修改 `Student` 的原型。

我们可以简单粗暴地做赋值：`Student.__proto__ = Person.prototype` ，但是，这太粗暴了。

我们还是使用比较规范的方式：

先用 `Object.create()` 来将`Person.prototype` 和 `Student.prototype` 关联上。

然后，修改一下构造函数 `Student.prototype.constructor = Student;`。

```
// Create a Student.prototype object that inherits
// from Person.prototype.
Student.prototype = Object.create(Person.prototype);

// Set the "constructor" property to refer to Student
Student.prototype.constructor = Student;
```

这样，我们就可以这样使用了。

```
var student = new Student("Hao Chen",
                                                    "haoel@hotmail.com",
                                                    "XYZ University",
                                                    "Computer Science");
student.introduction();
student.speak();
student.takeExams();

// Check that instanceof works correctly
console.log(student instanceof Person);  // true
console.log(student instanceof Student); // true
```

上述就是基于原型的面向对象编程的玩法了。

> 注：在ECMAScript标准的第四版开始寻求使JavaScript提供基于类的构造，且ECMAScript第六版有提供"class"(类)作为原有的原型架构之上的语法糖，提供构建对象与处理继承时的另一种语法。

小结

我们可以看到，这种玩法就是一种委托的方式。在使用委托的基于原型的语言中，运行时语言可以“仅仅通过序列的指针找到匹配”这样的方式来定位属性或者寻找正确的数据。所有这些创建行为、共享的行为需要的是委托指针。

不像是基于类的面向对象语言中类和接口的关系，原型和它的分支之间的关系并不要求子对象有相似的内存结构，因为如此，子对象可以继续修改而无需像基于类的系统那样整理结构。还有一个要提到的地方是，不仅仅是数据，方法也能被修改。因为这个原因，大多数基于原型的语言把数据和方法提作“slots”。

这种在对象里面直接修改的玩法，虽然这个特性可以带来运行时的灵活性，我们可以在运行时修改一个prototype，给它增加甚至删除属性和方法。但是其带来了执行的不确定性，也有安全性的问题，而代码还变得不可预测，这有点黑科技的味道了。因为这些不像静态类型系统，没有一个不可变的契约对代码的确定性有保证，所以，需要使用者来自己保证。

以下是《编程范式游记》系列文章的目录，方便你了解这一系列内容的全貌。**这一系列文章中代码量很大，很难用音频体现出来，所以没有录制音频，还望谅解。**

精选留言 10

yun
1519018218

皓叔，是否可以分享下如何写出高质量的代码的经验？排查问题的方法经验？

fsj
1523201760

看了这篇文章，然后又去翻了《JavaScript高级程序设计》，终于把原型继承搞明白了。之前学习Objective-C，觉得这语法挺奇葩的，但是面向对象思想和Java等都是互通的，然后又学习了JS，算是更奇葩了，没有类，玩的是原型继承，也是服，为什么JS不搞成类继承，非要搞一个原型继承呢？

云学
1517531366

c++的原型模式就是一个clone虚方法，这篇文章看得是很懵逼，完全不了解Javascript

强
1517546993

老师，有时间能写写go吗

蜉蝣
1566177688

看过之前完全不懂，看完之后懵懵懂懂。不晓得什么时候才能搞懂。

edisonhuang
1561421779

基于原型的编程可以理解为是基于实例的编程，这种玩法就是一种委托的方式。在使用委托的基于原型的语言中，运行时语言可以“仅仅通过序列的指针找到匹配”这样的方式来定位属性或者寻找正确的数据。所有这些创建行为共享的行为需要的是委托指针。js使用基于原型的编程方法

杨洪林
1550761097

引用于JavaScript, The Core的图例有些问题，Function.prototype的__proto__应该是指向Object.prototype, 似乎Object.prototype应该是所有非prime类型对象的根

Kennedy
1522288225

Lua的__index也属于原型链模式

songyy
1518625979

以前一直没有完全搞清楚JavaScript的prototype是什么原理，看了那张引用自《JavaScript，the core》的图和例子，感觉清晰好多了。

后面那个inheritance的例子也很棒，是我以前不清楚的。但最后那个instanceof的部分，我没太看懂原因：如果我有三层继承，那么三个class分别去检查instanceof，都会是true么？

ajodfaj
1517801055

Javascript其实挺有意思的