# 10 原型编程范式与面向对象

更新时间：2020-03-24 10:48:51

> 路漫漫其修远兮，吾将上下而求索。——屈原

JS 中的面向对象，围绕原型和原型链知识展开。

相信很多同学在阅读本节之前，一定也在各种平台、书籍中阅读过不少原型相关的文章（足见其重要性）。但在本节，我将带领大家从一个新的角度来认识原型。

大家已经知道，原型是 JavaScript 面向对象系统实现的根基。但是大家还需要知道，原型（Prototype）模式其实还是一种设计模式，同时更是一种是一种**编程范式**（programming paradigm）。

## 理解原型编程范式

很多小伙伴读到这儿还会有些迷惑：使用 JavaScript 以来，我确实离不开原型，按照上面的说法，也算是一直在应用原型编程范式了。但这个范式用得我一脸懵逼啊 —— 难道我还有除了原型以外的选择？

作为 JavaScript 开发者，我们确实没有别的选择 —— 毕竟开头我们说过，原型是 JavaScript 这门语言面向对象系统的根本。但在其它语言，比如 JAVA 中，类才是它面向对象系统的根本。

类是对一类实体的结构、行为的抽象。在基于类的面向对象语言中，我们首先关注的是**抽象** —— 我们需要先把具备通用性的类给设计出来，才能用这个类去实例化一个对象，进而关注到具体层面的东西。

而在 JS 这样的原型语言中，我们首先需要关注的就是具体 —— 具体的每一个实例的行为。根据不同实例的行为特性，我们把相似的实例关联到一个原型对象里去 —— 在这个被关联的原型对象里，就囊括了那些较为通用的行为和属性。基于此原型的实例，都能“复制”它的能力。

没错，在原型编程范式中，我们正是通过“**复制**”来创建新对象。但这个“复制”未必一定要开辟新的内存、把原型对象照着再实现一遍 —— 我们复制的是能力，而不必是实体。比如在 JS 中，我们就是通过使新对象保持对原型对象的引用来做到了“复制”。

**JavaScript 中的“类”**

这时有一部分小伙伴估计要炸毛了：啥？？？JavaScript 只能用 Prototype？我看你还活在上世纪，ES6 早就支持类了！现在我们 JavaScript 也是以类为中心的语言了。

这波同学的思想非常危险，因为 ES6 的类其实是原型继承的语法糖:

> ECMAScript 2015 中引入的 JavaScript 类实质上是 JavaScript 现有的基于原型的继承的语法糖。类语法不会为 JavaScript 引入新的面向对象的继承模型。 ——MDN

当我们尝试用 class 去定义一个 Dog 类时：

```
class Dog {
  constructor(name ,age) {
   this.name = name
   this.age = age
  }

  eat() {
    console.log('肉骨头真好吃')
  }
}
```

其实完全等价于写了这么一个构造函数:

```
function Dog(name, age) {
  this.name = name
  this.age = age
}
Dog.prototype.eat = function() {
  console.log('肉骨头真好吃')
}
```

所以说，JS 以原型作为其面向对象系统基石的本质并没有被改变。

## 理解原型与原型链

原型编程范式的核心思想就是**利用实例来描述对象，用实例作为定义对象和继承的基础**。在 JavaScript 中，原型编程范式的体现就是**基于原型链的继承**。这其中，对原型、原型链的理解是关键。

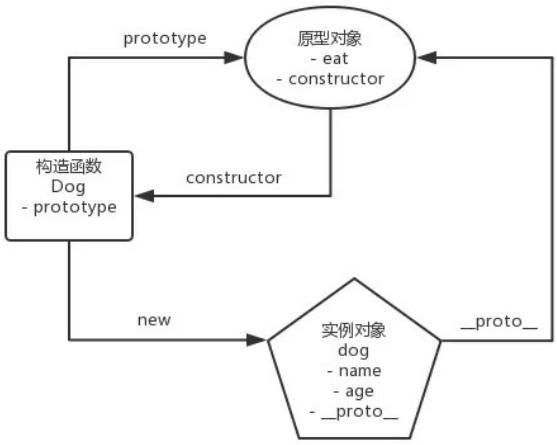
原型

在 JavaScript 中，每个构造函数都拥有一个 `prototype` 属性，它指向构造函数的原型对象，这个原型对象中有一个 construtor 属性指回构造函数；每个实例都有一个 `__proto__` 属性，当我们使用构造函数去创建实例时，实例的 `__proto__` 属性就会指向构造函数的原型对象。

具体来说，当我们这样使用构造函数创建一个对象时：

```
// 创建一个Dog构造函数
function Dog(name, age) {
  this.name = name
  this.age = age
}
Dog.prototype.eat = function() {
  console.log('肉骨头真好吃')
}
// 使用Dog构造函数创建dog实例
const dog = new Dog('旺财', 3)
```

这段代码里的几个实体之间就存在着这样的关系：



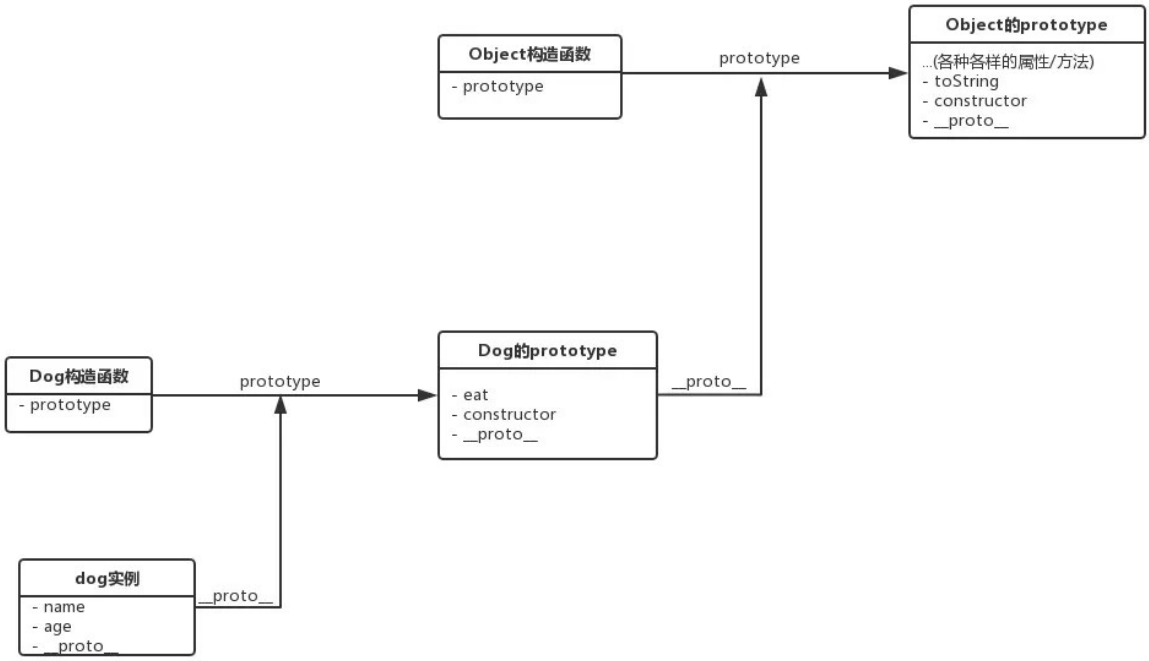

原型链

现在我在上面那段代码的基础上，进行两个方法调用:

```
// 输出"肉骨头真好吃"
dog.eat()
// 输出"[object Object]"
dog.toString()
```

明明没有在 dog 实例里手动定义 eat 方法和 toString 方法，它们还是被成功地调用了。这是因为当我试图访问一个 JavaScript 实例的属性 / 方法时，它首先搜索这个实例本身；当发现实例没有定义对应的属性 / 方法时，它会转而去搜索实例的原型对象；如果原型对象中也搜索不到，它就去搜索原型对象的原型对象，这个搜索的轨迹，就叫做原型链。

以我们的 eat 方法和 toString 方法的调用过程为例，它的搜索过程就是这样子的：




楼上这些彼此相连的 prototype，就构成了所谓的“原型链”。

注： 几乎所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例，除了 Object.prototype（当然，如果我们手动用 Object.create(null) 创建一个没有任何原型的对象，那它也不是 Object 的实例）。

以上为大家介绍了原型、原型链等 JavaScript 中核心的基础知识。基于这些知识，我们一起来做几个题吧 ^_^。

}

09 JS 更进一步：执行上下文与调用栈

11 原型与面向对象真题解析