白话原型和原型链

关于原型和原型链的介绍，网上数不胜数，但能讲清楚这两个概念的很少，大多数都是介绍各种对象、属性之间如何指来指去，最后的结果就是箭头满天飞，大脑一团糟。本文将从这两个概念的命名入手，用通俗易懂的语言，帮助你理解这两个东西到底是何方神圣。

一. 背景知识

JavaScript 和 Java、C++等传统面向对象的编程语言不同，它是没有类（class）的概念的（ES6 中的 class 也只不过是语法糖，并非真正意义上的类），而在 JavaScript 中，一切皆是对象（object）。在基于类的传统面向对象的编程语言中，对象由类实例化而来，实例化的过程中，类的属性和方法会拷贝到这个对象中；对象的继承实际上是类的继承，在定义子类继承于父类时，子类会将父类的属性和方法拷贝到自身当中。因此，这类语言中，对象创建和继承行为都是通过拷贝完成的。但在 JavaScript 中，对象的创建、对象的继承（更好的叫法是对象的代理，因为它并不是传统意义上的继承）是不存在拷贝行为的。**现在让我们忘掉类、忘掉继承，这一切都不属于 JavaScript。**

二. 原型和原型链

其实，原型这个名字本身就很容易产生误解，原型在百度词条中的释义是：指原来的类型或模型。按照这个定义解释的话，对象的原型是对象创建自身的模子，模子具备的特点对象都要具有，这俨然就是拷贝的概念。我们已经说过， JavaScript 的对象创建不存在拷贝，对象的原型实际上也是一个对象，它和对象本身是完全独立的两个对象。既然如此，原型存在的意义又是什么呢？原型是为了共享多个对象之间的一些共有特性（属性或方法），这个功能也是任何一门面向对象的编程语言必须具备的。A、B 两个对象的原型相同，那么它们必然有一些相同的特征。

JavaScript 中的对象，都有一个内置属性`[[Prototype]]`，指向这个对象的原型对象。当查找一个属性或方法时，如果在当前对象中找不到定义，会继续在当前对象的原型对象中查找；如果原型对象中依然没有找到，会继续在原型对象的原型中查找（原型也是对象，也有它自己的原型）；如此继续，直到找到为止，或者查找到最顶层的原型对象中也没有找到，就结束查找，返回 undefined。可以看出，这个查找过程是一个链式的查找，每个对象都有一个到它自身原型对象的链接，这些链接组件的整个链条就是原型链。拥有相同原型的多个对象，他们的共同特征正是通过这种查找模式体现出来的。

在上面的查找过程，我们提到了最顶层的原型对象，这个对象就是`Object.prototype`，这个对象中保存了最常用的方法，如 `toString`、`valueOf`、`hasOwnProperty` 等，因此我们才能在任何对象中使用这些方法。

1.字面量方式

当通过字面量方式创建对象时，它的原型就是 `Object.prototype`。虽然我们无法直接访问内置属性`[[Prototype]]`，但我们可以通过 `Object.getPrototypeOf()` 或对象的`__proto__`获取对象的原型。

```
var obj = {};
Object.getPrototypeOf(obj) === Object.prototype;  // true
obj.__proto__  === Object.prototype;          // true
```

复制代码

2.函数的构造调用

通过函数的构造调用（注意，我们不把它叫做构造函数，因为 JavaScript 中同样没有构造函数的概念，所有的函数都是平等的，只不过用来创建对象时，函数的调用方式不同而已）也是一种常用的创建对象的方式。基于同一个函数创建出来的对象，理应可以共享一些相同的属性或方法，但这些属性或方法如果放在 Object.prototype 里，那么所有的对象都可以使用它们了，作用域太大，显然不合

适。于是，JavaScript 在定义一个函数时，同时为这个函数定义了一个 默认的 prototype 属性，所有共享的属性或方法，都放到这个属性所指向的对象中。由此看出，通过一个函数的构造调用创建的对象，它的原型就是这个函数的 prototype 指向的对象。

```
var f = function(name) { this.name = name };
f.prototype.getName = function() { return this.name; }  //在 prototype 下存放所有对象的共享方法
var obj = new f('JavaScript');
obj.getName();                // JavaScript
obj.__proto__ === f.prototype;  // true
```

3.Object.create（）

第三种常用的创建对象的方式是使用 `Object.create()`。这个方法会以你传入的对象作为创建出来的对象的原型。

```
var obj = {};
var obj2 = Object.create(obj);
obj2.__proto__ === obj;      // true
```

这种方式还可以模拟对象的“继承”行为。

```
function Foo(name) {
        this.name = name;
}

Foo.prototype.myName = function() {
        return this.name;
};

function Bar(name,label) {
        Foo.call( this, name );  //
```

```
        this.label = label;
}

// temp 对象的原型是 Foo.prototype
var temp = Object.create( Foo.prototype );

// 通过 new Bar() 创建的对象，其原型是 temp，而 temp 的原型是 Foo.prototype，
// 从而两个原型对象 Bar.prototype 和 Foo.prototype 有了"继承"关系
Bar.prototype = temp;

Bar.prototype.myLabel = function() {
        return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
a.__proto__.__proto__ === Foo.prototype;  //true
```

三. `__proto__`和 prototype

这是容易混淆的两个属性。`__proto__`指向当前对象的原型，prototype 是函数才具有的属性，默认情况下，new 一个函数创建出的对象，其原型都指向这个函数的 prototype 属性。

四. 三种特殊情况

1.对于 JavaScript 中的内置对象，如 String、Number、Array、Object、Function 等，因为他们是 native 代码实现的，他们的原型打印出来都是 `ƒ () { [native code] }`。

2.内置对象本质上也是函数，所以可以通过他们创建对象，创建出的对象的原型指向对应内置对象的 prototype 属性，最顶层的原型对象依然指向 Object.prototype。

```
'abc'.__proto__ === String.prototype;  // true
new String('abc').__proto__ === String.prototype;  //true

new Number(1).__proto__  ==== Number.prototype;  // true

[1,2,3].__proto__ === Array.prototype;          // true
new Array(1,2,3).__proto__ === Array.prototype;  // true

({}).__proto__ === Object.prototype;            // true
new Object({}).__proto__ === Object.prototype;    // true

var f = function() {};
f.__proto__ === Function.prototype;          // true
var f = new Function('{}');
f.__proto__ === Function.prototype;          // true
```

3.`Object.create(null)` 创建出的对象，不存在原型。

```
var a = Object.create(null);
a.__proto__;            // undefined
```

此外，函数的 prototype 中还有一个 constructor 方法，建议大家就当它不存在，它的存在让 JavaScript 原型的概念变得更加混乱，而且这个方法也几乎没有作用。