Chapter 7.2. OOP ECMAScript implementation

原文地址
http://dmitrysoshnikov.com/ecmascript/chapter-7-2-oop-ecmascript-implementation/

汤姆大叔解读
https://www.cnblogs.com/TomXu/archive/2012/02/06/2330609.html

知乎解读
https://zhuanlan.zhihu.com/p/150448758

数据类型

标准规范里定义了9种数据类型，但只有6种是在ECMAScript程序里可以直接访问的,它们是：**Undefined**、**Null**、**Boolean**、**String**、**Number**、**Object**。

另外3种类型只能在实现级别访问（ECMAScript对象是不能使用这些类型的）并用于规范来解释一些操作行为、保存中间值。这3种类型是：Reference、List和Completion。

原始值类型

alert(typeof null); // "object"

显示"object"原因是因为规范就是这么规定的：对于Null值的typeof字符串值返回"object“。
规范没有想象解释这个，但是Brendan Eich (JavaScript发明人)注意到null相对于undefined大多数都是用于对象出现的地方，例如设置一个对象为空引用。但是有些文档里有些气人将之归结为bug，而且将该bug放在Brendan Eich也参与讨论的bug列表里，结果就是任其自然，还是把typeof null的结果设置为object（尽管262-3的标准是定义null的类型是Null，262-5已经将标准修改为null的类型是object了）。

Object类型

对象是一个包含**key**-**value**对的无序集合

动态性

内置对象、原生对象及宿主对象

所有**ECMAScript实现的对象都是原生对象**（其中一些是内置对象、一些在程序执行的时候创建，例如用户自定义对象）。

内置对象是原生对象的一个子集、是在程序开始之前内置到ECMAScript里的（例如，parseInt, Match等）。

所有的**宿主对象是由宿主环境提供**的，通常是浏览器，并可能包括如window、alert等。

Boolean，String 和 Number对象

另外，规范也定义了一些原生的特殊包装类，这些对象是：（注意：类也是对象）

1. **布尔对象**
2. **字符串对象**
3. **数字对象**

字面量Literal

对于三个对象的值：**对象**（object）,**数组**（array）和**正则表达式**（regular expression），他们分别有简写的标示符称为：对象初始化器、数组初始化器、和正则表达式初始化器：

var *array* = [1, 2, 3]
var *object* = {**a**: 1, **b**: 2, **c**: 3}
var *re* = /^\d+$/g

关联数组

```
var a = {x: 10};
a['y'] = 20;
a.z = 30;

var b = new Number(1);
b.x = 10;
b.y = 20;
b['z'] = 30;

var c = new Function('');
c.x = 10;
c.y = 20;
c['z'] = 30;

// 等等，任意对象的子类型"subtype"
```

对象转换

将对象转化成原始值可以用**valueOf**方法，

valueOf的默认值会根据根据**对象的类型改变**（如果不被覆盖的话），对某些对象，他返回的是this。

Object.prototype上定义的**toString**方法具有特殊意义，它返回的我们下面将要讨论的内部**[[Class]]**属性值。

和转化成原始值（ToPrimitive）相比，将值转化成对象类型也有一个转化规范（ToObject）。

一个显式方法是使用内置的Object构造函数作为function来调用ToObject（有些类似通过new关键字也可以）：

属性的特性

所有的属性（property） 都可以有很多特性（attributes）。

1. {ReadOnly}——忽略向属性赋值的写操作尝，但只读属性可以由宿主环境行为改变——也就是说不是“恒定值”；
2. {DontEnum}——属性不能被for..in循环枚举
3. {DontDelete}——糊了delete操作符的行为被忽略（即删不掉）；
4. {Internal}——内部属性，没有名字（仅在实现层面使用），ECMAScript里无法访问这样的属性。

注意，在*ES5*里*{ReadOnly}*，*{DontEnum}*和*{DontDelete}*被重新命名为[[*Writable*]]，[[*Enumerable*]]和[[*Configurable*]]，可以手工通过*Object.defineProperty*或类似的方法来管理这些属性。

内部属性和方法

对象也可以有内部属性（实现层面的一部分），并且ECMAScript程序无法直接访问（但是下面我们将看到，一些实现允许访问一些这样的属性）。 这些属性通过嵌套的中括号[[ ]]进行访问。我们来看其中的一些，这些属性的描述可以到规范里查阅到。
每个对象都应该实现如下内部属性和方法：

1. [[Prototype]]——对象的原型（将在下面详细介绍）
2. [[Class]]——字符串对象的一种表示（例如，Object Array ，Function Object，Function等）;用来区分对象
3. [[Get]]——获得属性值的方法
4. [[Put]]——设置属性值的方法
5. [[CanPut]]——检查属性是否可写
6. [[HasProperty]]——检查对象是否已经拥有该属性
7. [[Delete]]——从对象删除该属性
8. [[DefaultValue]]返回对象对于的原始值（调用valueOf方法，某些对象可能会抛出TypeError异常）。

通过Object.prototype.toString()方法可以间接得到内部属性[[Class]]的值，该方法

应该返回下列字符串： "[object " + [[Class]] + "]" 。

构造函数

构造函数是**一个函数**，用来**创建并初始化**新创建的对象。

因此，具有**内部 [[Call]] 属性**的对象被称为 *functions*。

内部的 **[[Construct]]** 方法由用于构造函数的 **new 操作符**激活。

对象创建的算法

内部的 [[Constuct]] 方法的行为可以被表示为：

备注：F 等于JavaScript函数

```
F.[[Construct]](initialParameters):

O = new NativeObject();

// 属性[[Class]]被设置为"Object"
O.[[Class]] = "Object"

// 引用F.prototype的时候获取该对象g
var __objectPrototype = F.prototype;

// 如果__objectPrototype是对象，就:
O.[[Prototype]] = __objectPrototype
// 否则:
O.[[Prototype]] = Object.prototype;
// 这里O.[[Prototype]]是Object对象的原型

// 新创建对象初始化的时候应用了F.[[Call]]
// 参数和F里的initialParameters是一样的
R = F.[[Call]](initialParameters);

// 将this设置为新创建的对象O
this === O;

// 这里R是[[Call]]的返回值
// 在JS里看，像这样:
// R = F.apply(O, initialParameters);

// 如果R是对象
return R
```

```
// 否则
return O
```

请注意两个主要特点：

第一，创建对象的 *prototype* 属性是来自于当前时刻函数的 prototype 属性
第二，我们上面提到，如果在对象初始时 [[Call]] 返回了一个对象，则将其作为整个 new 表达式的结果：

原型（指的是函数的原型）

每个对象都有一个原型（一些系统对象除外）。通过内部的，隐式的和不可访问的 [[Protorype]] 属性来组织与原型的通信。原型可以是一个对象，也可以是null值。

Property `constructor`(指的是函数的原型的constructor属性)

constructor属性属于原型，并且通过继承来访问对象。

通过继承的 constructor 属性实例可以间接的获取原型对象的引用：

```
function A() {}
A.prototype.x = new Number(10);

var a = new A();
console.log(a.constructor.prototype); // [object Object]

console.log(a.x); // 10, via delegation
// the same as a.[[Prototype]].x
console.log(a.constructor.prototype.x); // 10

console.log(a.constructor.prototype.x === a.x); // true
```

注意，函数的 constructor 和 prototype 可以在对象创建后重新定义。这种情况下，对象失去通过上面机制的引用。

显示 prototype 与隐式的 [[Prototype]] 属性

通过函数的prototype属性直接引用原型

```
a.[[Prototype]] ----> Prototype <---- A.prototype
```

```

而且，在对象创建阶段，实例的 [[Prototype]] 完全从构造函数的prototype属性上获取值。

然而，替换构造函数的prototype属性**不影响已经创建的对象的原型**。仅仅只是它的构造函数的prototype属性改变了。这意味着新对象会有一个新的原型。但是已经创建的对象（在 prototype 属性修改之前），拥有旧的原型的引用，并且引用不能被修改。

**非标准的__proto__属性**

然而，一些实现中，例如，SpiderMonkey，通过非标准的 __proto__ 属性提供对象原型的显示引用

**对象独立于构造函数**

因为实例的原型独立于构造函数和构造函数的prototype属性，构造函数在完成它的主要目的（创建对象）后，可以移除。原型对象将会继续存在，通过 [[Prototype]] 属性被引用。

**instanceof操作符的特性**

通过构造函数的prototype属性显示引用原型，与 instanceof 操作符的工作相关。

**原型可以存放方法并共享属性**

## 读写属性

正如我们提到，读取和写入属性值是通过内部的[[Get]]和[[Put]]方法。这些内部方法是通过属性访问器激活的：点标记法或者索引标记法：

**[[Get]]方法**

[[Get]]也会从原型链中查询属性，所以通过对象也可以访问原型中的属性。

**[[Put]]方法**

[[Put]]方法可以创建、更新对象自身的属性，并且掩盖原型里的同名属性。

请注意，不能掩盖原型里的只读属性，赋值结果将忽略，这是由内部方法[[CanPut]]控制的。

**属性访问器**
```

内部方法[[Get]]和[[Put]]在ECMAScript里是通过**点符号**或者**索引法**来激活的，如果属性标示符是合法的名字的话，可以通过“.”来访问，而索引运行动态定义名称。

这里有一个非常重要的特性——属性访问器总是使用**ToObject**规范来对待“.”左边的值。

并且由于左边的值是隐式转换，可能被粗糙的说成“JavaScript一切皆对象”（但是，正如我们早就知道的，当然不是所有的东西，因为还有原始的东西）。

继承

众所周知，ECMAScript使用基于原型的委托继承。

原型链

Prototype chain

Let's show how to create these prototype chains for the *user-defined* objects. It is quite simple:

```
function A() {
  console.log('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
console.log([a.x, a.y]); // 10 (own), 20 (inherited)

function B() {}

// the easiest variant of prototypes
// chaining is setting child
// prototype to new object created,
// by the parent constructor
B.prototype = new A();

// fix .constructor property, else it would be A
B.prototype.constructor = B;

var b = new B();
console.log([b.x, b.y]); // 10, 20, both are inherited

// [[Get]] b.x:
// b.x (no) -->
// b.[[Prototype]].x (yes) - 10

// [[Get]] b.y
// b.y (no) -->
// b.[[Prototype]].y (no) -->
// b.[[Prototype]].[[Prototype]].y (yes) - 20

// where b.[[Prototype]] === B.prototype,
// and b.[[Prototype]].[[Prototype]] === A.prototype
```

这种原型继承写法有两个特性：

首先，B.prototype将包含x属性。尽管原型继承正常情况是没问题的，但B构造函数有时候可能**不需要x属性**，与基于class的继承相比，所有的属性都复制到后代子类里了。

尽管如此，如果有需要（模拟基于类的继承）将x属性赋给B构造函数创建的对象上，有一些方法，我们后来来展示其中一种方式。

其次，B子类**原型创建的时候，A构造函数的代码也执行了**，这不是**一个特征**而是**缺点。**我们可以看到消息"A.[[Call]] activated"显示了两次——当用A构造函数创建对

象赋给B.prototype属性的时候，另外一场是a对象创建自身的时候！

此外，在父类的构造函数**有太多代码的话也是一种缺点**。

要解决这些“**功能**”和**问题**，程序员使用**标准模式来连接原型**（下面展示），主要目标就是**创建中间包装器构造函数**，链接所需的原型。

```
function A() {
  alert('A.[[Call]] activated');
  this.x = 10;
}
A.prototype.y = 20;

var a = new A();
alert([a.x, a.y]); // 10 (自身), 20 (集成)

function B() {
  // 或者使用A.apply(this, arguments)
  B.superproto.constructor.apply(this, arguments);
}

// 继承：通过空的中间构造函数将原型连在一起
var F = function () {};
F.prototype = A.prototype; // 引用
B.prototype = new F();
B.superproto = A.prototype; // 显示引用到另外一个原型上, "sugar"

// 修复原型的constructor属性，否则的就是A了
B.prototype.constructor = B;

var b = new B();
alert([b.x, b.y]); // 10 (自身), 20 (集成)
```

注意我们是如何在 **b 实例上创建自己的 x 属性**的：在新创建对象的中，我们通过 B.superproto.constructor 引用调用父级构造函数。
我们也修复了**无须使用父级构造函数来创建后代原型的问题**。现在，当需要的时候， "A.[[Call]].activated" 消息会显示。

而且不要每次都**重复原型链的相同操作**（创建中间构造器，设置 superproto 糖，恢复原来的 constructor 等），这个模板可以封装在方便的工具函数中，不管构造函数的名字是什么，目的都是链接原型：

```
function inherit(child, parent) {
  var F = function () {};
  F.prototype = parent.prototype
  child.prototype = new F();
  child.prototype.constructor = child;
  child.superproto = parent.prototype;
  return child;
}
```

注意，ES5为原型链标准化了这个工具函数，那就是Object.create方法。ES3可以使用以下方式实现：

```
Object.create ||
Object.create = function (parent, properties) {
  function F() {}
  F.prototype = parent;
  var child = new F;
  for (var k in properties) {
    child[k] = properties[k].value;
  }
  return child;
}

// 用法
var foo = {x: 10};
var bar = Object.create(foo, {y: {value: 20}});
console.log(bar.x, bar.y); // 10, 20
```