**JavaScript学习笔记**

何清 2016-10-12

概览

JavaScript 是一种面向对象的动态语言，它包含类型、运算符、标准内置（ built-in）对象和方法。它的语法来源于 Java 和 C，所以这两种语言的许多语法特性同样适用于 JavaScript。需要注意的一个主要区别是 **JavaScript 不支持类**，类这一概念在 JavaScript 通过对象原型（object prototype）。另一个主要区别是 **JavaScript 中的函数也是对象**，JavaScript 允许函数在包含可执行代码的同时，能像其他对象一样被传递。JavaScript 中不存在类和实例之间的这种差异。

1. 相等比较== 与 ===

由两个“=（等号）”组成的相等运算符有类型自适应的功能，具体例子如下：

123 == "123" // true

1 == true; // true

如果在比较前不需要自动类型转换，应该使用由三个“=（等号）”组成的相等运算符：

1 === true; //false

123 === "123"; // false

JavaScript 还支持 != 和 !== 两种不等运算符，具体区别与两种相等运算符的区别类似。

1. 多重分支switch

在需要多重分支时可以使用 基于一个**数字或字符串**的switch 语句：

default 语句是可选的。switch 和 case 都可以使用需要运算才能得到结果的表达式；在 switch 的表达式和 case 的表达式是使用 **=== 严格相等运算符进行比较的**。

1. 创建一个空对象

有两种简单方法可以创建一个空对象：

var obj = new Object();

和：

var obj = {};

这两种方法在语义上是相同的。第二种更方便的方法叫作“对象字面量（object literal）”法。这种也是 JSON 格式的核心语法，一般我们优先选择第二种方法。

1. JavaScript数组

JavaScript 中的数组是一种特殊的对象。它的工作原理与普通对象类似（以数字为属性名，但**只能通过[] 来访问**），但数组还有一个特殊的属性——length（长度）属性。这个属性的值通常比数组最大索引大 1。

创建数组的传统方法是：

var a = new Array();

a[0] = "dog";

a[1] = "cat";

a[2] = "hen";

a.length; // 3

使用数组字面量（array literal）法更加方便：

var a = ["dog", "cat", "hen"];

a.length; // 3

注意，Array.length **并不总是等于数组中元素的个数**，如下所示：

var a = ["dog", "cat", "hen"];

a[100] = "fox";

a.length; // 101

记住：**数组的长度是比数组最大索引值多一的数**。

如果试图访问一个不存在的数组索引，会得到 undefined：

typeof(a[90]); // undefined

可以通过如下方式遍历一个数组：

for (var i = 0; i < a.length; i++) {

// Do something with a[i]

}

遍历数组的另一种方法是使用 **for...in** 循环。注意，如果有人向 Array.prototype 添加了新的属性，使用这样的循环这些属性也同样会被遍历。所以并不推荐这种方法：

for (var i in a) {

// Do something with a[i]

}

ECMAScript 5 增加了遍历数组的另一个方法 **forEach()**：

["dog", "cat", "hen"].forEach(function(currentValue, index, array) {

// Do something with currentValue or array[index]

});

如果想在数组后**追加元素**，只需要：

a.push(item);

Array（数组）类自带了许多方法。查看<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array>。

| **方法名称** | **描述** |
| --- | --- |
| a.toString() | 返回一个包含数组中所有元素的字符串，每个元素通过逗号分隔。 |
| a.toLocaleString() | 根据宿主环境的区域设置，返回一个包含数组中所有元素的字符串，每个元素通过逗号分隔。 |
| a.concat(item1[, item2[, ...[, itemN]]]) | 返回一个数组，这个数组包含原先 a 和 item1、item2、……、itemN 中的所有元素。 |
| a.join(sep) | 返回一个包含数组中所有元素的字符串，每个元素通过指定的 sep 分隔。 |
| a.pop() | 删除并返回数组中的最后一个元素。 |
| a.push(item1, ..., itemN) | 将 item1、item2、……、itemN 追加至数组 a。 |
| a.reverse() | 数组逆序（会更改原数组 a）。 |
| a.shift() | 删除并返回数组中第一个元素。 |
| a.slice(start, end) | 返回子数组，以 a[start] 开头，以 a[end] 前一个元素结尾。 |
| a.sort([cmpfn]) | 依据 cmpfn 返回的结果进行排序，如果未指定比较函数则按字符顺序比较（即使元素是数字）。 |
| a.splice(start, delcount[, item1[, ...[, itemN]]]) | 从 start 开始，删除 delcount 个元素，然后插入所有的item。 |
| a.unshift([item]) | 将 item 插入数组头部，返回数组新长度（考虑undefined）。 |

1. 如何创建对象

在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。原因是这将导致每次构造器被调用，方法都会被重新赋值一次（也就是说，为每一个对象的创建）。

考虑以下虽然不切实际但却说明问题的示例：

function MyObject(name, message) {

this.name = name.toString();

this.message = message.toString();

this.getName = function() {

return this.name;

};

this.getMessage = function() {

return this.message;

};

}

上面的代码并未利用到闭包的益处，因此，应该修改为如下常规形式：

function MyObject(name, message) {

this.name = name.toString();

this.message = message.toString();

}

MyObject.prototype = {

getName: function() {

return this.name;

},

getMessage: function() {

return this.message;

}

};

或者改成：

function MyObject(name, message) {

this.name = name.toString();

this.message = message.toString();

}

MyObject.prototype.getName = function() {

return this.name;

};

MyObject.prototype.getMessage = function() {

return this.message;

};

在前面的两个示例中，继承的原型可以为所有对象共享，且不必在每一次创建对象时定义方法。

对象模型的细节

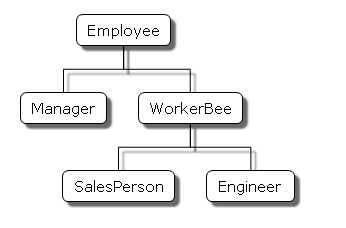
**JavaScript 是一种基于原型的面向对象语言，而不是基于类的。JavaScript 中不存在类和实例之间的这种差异。**

1. 子类和继承、示例

基于类的语言是通过对类的定义中构建类的层级结构的。在类定义中，可以指定新的类是一个现存的类的子类。子类将继承父类的全部属性，并可以添加新的属性或者修改继承的属性。例如，假设 Employee 类只有 name 和 dept 属性，而 Manager 是 Employee 的子类并添加了 reports 属性。这时，Manager 类的实例将具有所有三个属性：name，dept 和 reports。

JavaScript 通过将构造器函数与原型对象相关联的方式来实现继承。这样，您可以创建完全一样的 Employee — Manager 示例，不过需要使用略微不同的术语。首先，定义 Employee 构造器函数，指定 name 和 dept 属性；然后，定义 Manager 构造器函数，指定 reports 属性。最后，将一个新的 Employee 对象赋值给 Manager 构造器函数的 prototype 属性。这样，当创建一个新的 Manager 对象时，它将从 Employee 对象中继承 name and dept 属性。

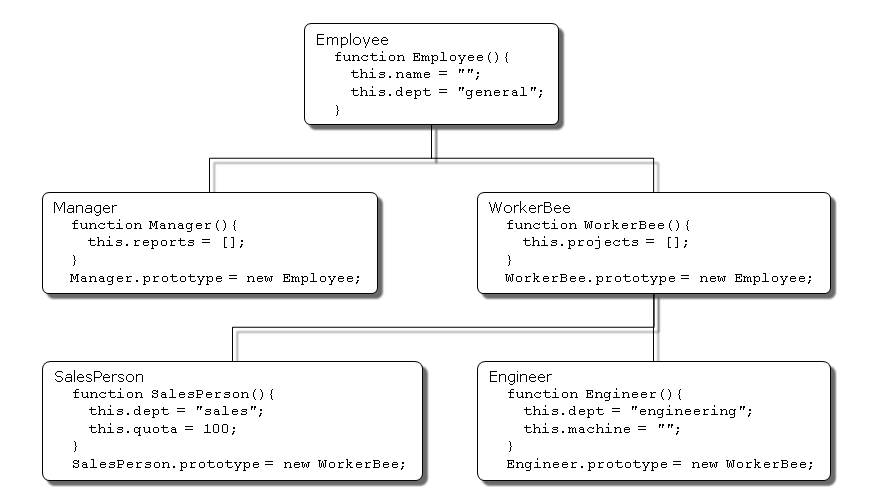
余下部分将使用如下图所示的 Employee 层级结构。

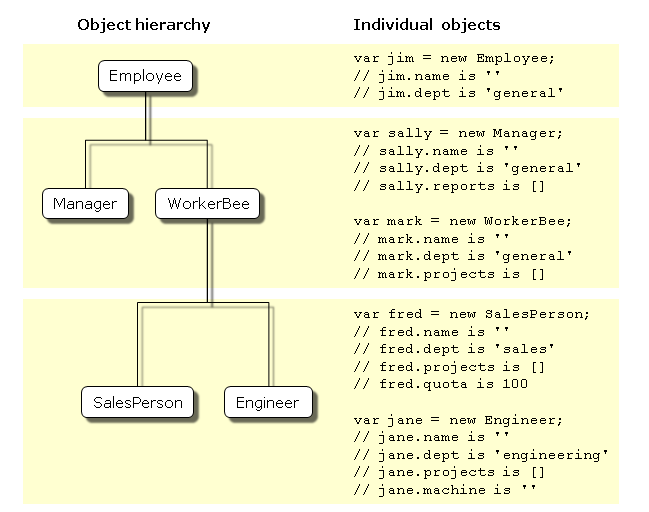


例子中会使用以下对象：

* Employee 具有 name 属性（默认值为空的字符串）和 dept 属性（默认值为 "general"）。
* Manager 是 Employee的子类。它添加了 reports 属性（默认值为空的数组，以 Employee 对象数组作为它的值）。
* WorkerBee 是 Employee的子类。它添加了 projects 属性（默认值为空的数组，以字符串数组作为它的值）。
* SalesPerson 是 WorkerBee的子类。它添加了 quota 属性（其值默认为 100）。它还重载了 dept 属性值为 "sales"，表明所有的销售人员都属于同一部门。
* Engineer 基于 WorkerBee。它添加了 machine 属性（其值默认为空的字符串）同时重载了 dept 属性值为 "engineering"。

下面介绍了如何使用非常简单的（同时也是相当不灵活的）定义，使得继承得以实现。

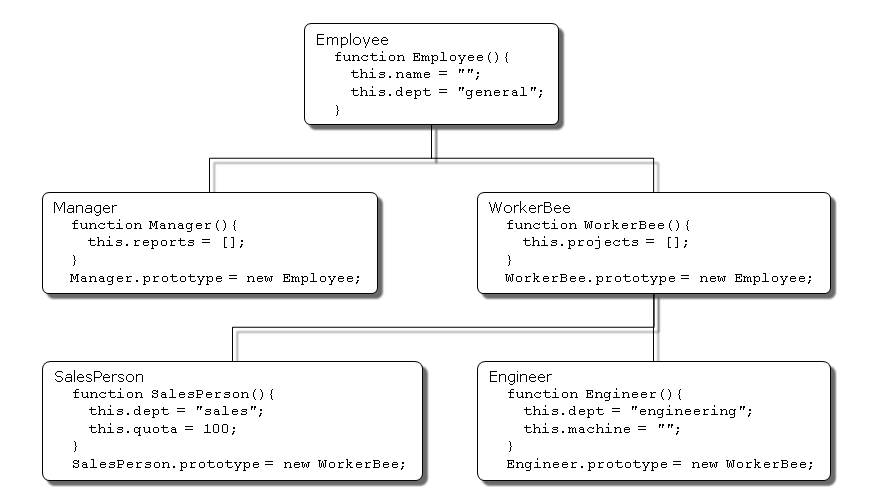




1. 添加和移除属性

在基于类的语言中，通常在编译时创建类，然后在编译时或者运行时对类的实例进行实例化。一旦定义了类，无法对类的属性进行更改。然而，在 JavaScript 中，允许运行时添加或者移除任何对象的属性。如果您为一个对象中添加了一个属性，而这个对象又作为其它对象的原型，则以该对象作为原型的所有其它对象也将获得该属性。

1. 对象的属性
2. 继承属性



假设通过如下语句创建一个 mark 对象作为 WorkerBee 的实例：

var mark = new WorkerBee;

**解析：**当 JavaScript 发现 new 操作符时，它会创建一个通用(generic)对象，并将其作为关键字 this 的值传递给 WorkerBee 的构造器函数。该构造器函数显式地设置 projects 属性的值，然后隐式地将其内部的 \_\_proto\_\_ 属性设置为 WorkerBee.prototype 的值（属性的名称前后均有两个下划线）。\_\_proto\_\_ 属性决定了用于返回属性值的原型链。一旦这些属性设置完成，JavaScript 返回新创建的对象，然后赋值语句会将变量 mark 的值指向该对象。

这个过程不会显式的将 mark所继承的原型链中的属性值作为本地变量存放在 mark 对象中。当请求属性的值时，JavaScript 将首先检查对象自身中是否存在属性的值，如果有，则返回该值。如果不存在，JavaScript会通过 \_\_proto\_\_对原型链进行检查。如果原型链中的某个对象包含该属性的值，则返回这个值。如果没有找到该属性，JavaScript 则认为对象中不存在该属性。这样，mark 对象中将具有如下的属性和对应的值：

mark.name = "";

mark.dept = "general";

mark.projects = [];

mark 对象从 mark.\_\_proto\_\_ 中保存的原型对象中继承了 name 和 dept 属性的值。并由 WorkerBee 构造器函数为 projects 属性设置了本地值。 这就是 JavaScript 中的属性和属性值的继承。

由于这些构造器不支持为实例设置特定的值，所以这些属性值仅仅是创建自 WorkerBee 的所有对象所共享的默认值。当然这些属性的值是可以修改的，所以您可以为 mark指定特定的信息，如下所示：

mark.name = "Doe, Mark";

mark.dept = "admin";

mark.projects = ["navigator"];

1. 添加属性

在 JavaScript 中，您可以在运行时为任何对象添加属性，而不必受限于构造器函数提供的属性。添加特定于某个对象的属性，只需要为该对象指定一个属性值，如下所示：

mark.bonus = 3000;

这样 **mark 对象就有了 bonus 属性**，而其它 WorkerBee **则没有该属性**。

如果您向某个构造器函数的原型对象中添加新的属性，那么该属性将添加到从这个原型中继承属性的所有对象的中。例如，可以通过如下的语句向所有雇员中添加 specialty 属性：

Employee.prototype.specialty = "none";

只要 JavaScript 执行了该语句，则 mark 对象也将具有 specialty 属性，其值为 "none"。

1. 设置默认值的特殊惯用法

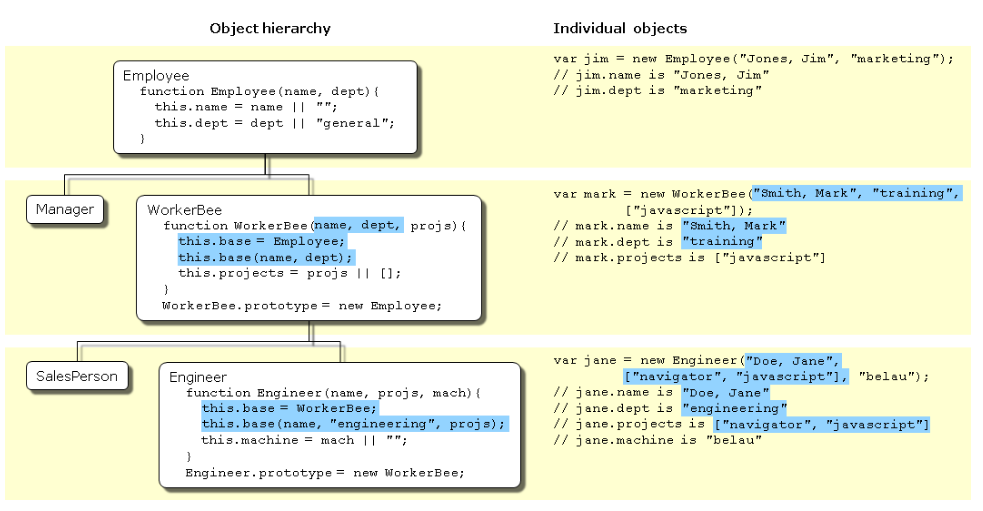
JavaScript 定义过程使用了一种设置默认值的特殊惯用法：

this.name = name || "";

JavaScript 的逻辑或操作符（||）会对第一个参数进行判断。如果该参数值运算后结果为真，则操作符返回该值。否则，操作符返回第二个参数的值。因此，这行代码首先检查 name 是否是对name 属性有效的值。如果是，则设置其为 this.name 的值。否则，设置 this.name 的值为空的字符串。尽管这种用法乍看起来有些费解，为了简洁起见，本章将使用这种习惯用法。

**注意：**如果调用构造器函数时，指定了可以转换为 false 的参数（比如 0 （零）和空字符串（""）），结果可能出乎调用者意料。此时，将使用默认值（译者注：而不是指定的参数值 0 和 ""）。

1. 属性的继承



下面是 Engineer 构造器的定义：

function Engineer (name, projs, mach) {

this.base = WorkerBee;

this.base(name, "engineering", projs);

this.machine = mach || "";

}

假设您创建了一个新的 Engineer 对象，如下所示：

var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");

执行时，JavaScript 会有以下步骤：

1. new 操作符创建了一个新的通用对象，并将其 \_\_proto\_\_ 属性设置为 Engineer.prototype。
2. new 操作符将该新对象作为 this 的值传递给 Engineer 构造器。
3. 构造器为该新对象创建了一个名为 base 的新属性，并指向 WorkerBee 的构造器。这使得 WorkerBee 构造器成为 Engineer 对象的一个方法。base 属性的名称并没有什么特殊性，我们可以使用任何其他合法的名称来代替；base 仅仅是为了贴近它的用意。
4. 构造器调用 base 方法，将传递给该构造器的参数中的两个，作为参数传递给 base 方法，同时还传递一个字符串参数  "engineering"。显式地在构造器中使用 "engineering" 表明所有 Engineer 对象继承的 dept 属性具有相同的值，且该值重载了继承自 Employee 的值。
5. 因为 base 是 Engineer 的一个方法，在调用 base 时，JavaScript 将在步骤 1 中创建的对象绑定给 this 关键字。这样，WorkerBee 函数接着将"Doe, Jane" 和 "engineering" 参数传递给 Employee 构造器函数。当从 Employee 构造器函数返回时，WorkerBee 函数用剩下的参数设置projects 属性。
6. 当从 base 方法返回后，Engineer 构造器将对象的 machine 属性初始化为 "belau"。
7. 当从构造器返回时，JavaScript 将新对象赋值给 jane 变量。

您可以认为，在 Engineer 的构造器中调用了 WorkerBee 的构造器，也就为 Engineer 对象设置好了继承关系。事实并非如此。调用 WorkerBee 构造器确保了Engineer 对象以所有在构造器中所指定的属性被调用。但是，如果后续在 Employee 或者 WorkerBee 原型中添加了属性，那些属性不会被 Engineer 对象继承。例如，假设如下语句：

function Engineer (name, projs, mach) {

this.base = WorkerBee;

this.base(name, "engineering", projs);

this.machine = mach || "";

}

var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");

Employee.prototype.specialty = "none";

对象 jane 不会继承 specialty 属性。您必须显式地设置原型才能确保动态的继承。如果修改成如下的语句：

function Engineer (name, projs, mach) {

this.base = WorkerBee;

this.base(name, "engineering", projs);

this.machine = mach || "";

}

Engineer.prototype = new WorkerBee;

var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");

Employee.prototype.specialty = "none";

现在 jane 对象的 specialty 属性为 "none" 了。

继承的**另一种途径是使用call() / apply() 方法**。下面的方式都是等价的：

|  |  |
| --- | --- |
| function Engineer (name, projs, mach) {  this.base = WorkerBee;  this.base(name, "engineering", projs);  this.machine = mach || "";  } | function Engineer (name, projs, mach) {  WorkerBee.call(this, name, "engineering", projs);  this.machine = mach || "";  } |

使用 javascript 的 call() 方法相对明了一些，因为无需 base 方法了。

1. 再谈属性的继承

**本地值和继承值：**

正如本章前面所述，在访问一个对象的属性时，JavaScript 将执行下面的步骤：

1. 检查本地值是否存在。如果存在，返回该值。
2. 如果本地值不存在，检查原型链（通过 \_\_proto\_\_ 属性）。
3. 如果原型链中的某个对象具有指定属性的值，则返回该值。
4. 如果这样的属性不存在，则对象没有该属性。

以上步骤的结果依赖于您是如何定义的。最早的例子中具有如下定义：

function Employee () {

this.name = "";

this.dept = "general";

}

function WorkerBee () {

this.projects = [];

}

WorkerBee.prototype = new Employee;

基于这些定义，假定通过如下的语句创建 WorkerBee 的实例 amy：

var amy = new WorkerBee;

则 amy 对象将具有一个本地属性，projects。name 和 dept 属性则不是 amy 对象本地的，而是从 amy 对象的 \_\_proto\_\_ 属性获得的。因此，amy 将具有如下的属性值：

amy.name == "";

amy.dept == "general";

amy.projects == [];

现在，假设修改了与 Employee 的相关联原型中的 name 属性的值：

Employee.prototype.name = "Unknown"

乍一看，您可能觉得新的值会传播给所有 Employee 的实例。然而，并非如此。

在创建 Employee 对象的任意实例时，该实例的 name 属性将获得一个本地值（空的字符串）。这就意味着在创建一个新的 Employee 对象作为 WorkerBee 的原型时，WorkerBee.prototype 的 name 属性将具有一个本地值。因此，当 JavaScript 查找 amy 对象（WorkerBee 的实例）的 name 属性时，JavaScript 将找到 WorkerBee.prototype 中的本地值。因此，也就不会继续在原型链中向上找到 Employee.prototype 了。

如果想在运行时修改一个对象的属性值并且希望该值被所有该对象的后代所继承，您就不能在该对象的构造器函数中定义该属性。而应该将该属性添加到该对象所关联的原型中。例如，假设将前面的代码作如下修改：

function Employee () {

this.dept = "general";

}

Employee.prototype.name = "";

function WorkerBee () {

this.projects = [];

}

WorkerBee.prototype = new Employee;

var amy = new WorkerBee;

Employee.prototype.name = "Unknown";

在这种情况下，amy 的 name 属性将为 "Unknown"。

正如这些例子所示，如果希望对象的属性具有默认值，并且希望在运行时修改这些默认值，应该在对象的原型中设置这些属性，而不是在构造器函数中。

1. 没有多继承

JavaScript 不支持多重继承。JavaScript 属性值的继承是在运行时通过检索对象的原型链来实现的。因为对象只有一个原型与之关联，所以 JavaScript 无法动态地从多个原型链中继承。

继承与原型链

1. 使用不同的方法来创建对象和生成原型链
2. 使用普通语法创建对象

var o = {a: 1};

// o这个对象继承了Object.prototype上面的所有属性

// 所以可以这样使用 o.hasOwnProperty('a').

// hasOwnProperty 是Object.prototype的自身属性。

// Object.prototype的原型为null。

// 原型链如下:

// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];

// 数组都继承于Array.prototype

// (indexOf, forEach等方法都是从它继承而来).

// 原型链如下:

// a ---> Array.prototype ---> Object.prototype ---> null

function f(){

return 2;

}

// 函数都继承于Function.prototype

// (call, bind等方法都是从它继承而来):

// f ---> Function.prototype ---> Object.prototype ---> null

1. 使用构造器创建对象

在 JavaScript 中，构造器其实就是一个普通的函数。当使用 new 操作符 来作用这个函数时，它就可以被称为构造方法（构造函数）。

function Graph() {

this.vertexes = [];

this.edges = [];

}

Graph.prototype = {

addVertex: function(v){

this.vertexes.push(v);

}

};

var g = new Graph();

// g是生成的对象,他的自身属性有'vertexes'和'edges'.

// 在g被实例化时,g.[[Prototype]]指向了Graph.prototype.

1. 使用 Object.create 创建对象

ECMAScript 5 中引入了一个新方法：Object.create()。可以调用这个方法来创建一个新对象。新对象的原型就是调用 create 方法时传入的第一个参数：

var a = {a: 1};

// a ---> Object.prototype ---> null

var b = Object.create(a);

// b ---> a ---> Object.prototype ---> null

console.log(b.a); // 1 (继承而来)

var c = Object.create(b);

// c ---> b ---> a ---> Object.prototype ---> null

var d = Object.create(null);

// d ---> null

console.log(d.hasOwnProperty); // undefined, 因为d没有继承Object.prototype

1. 使用 class 关键字

ECMAScript6 引入了一套新的关键字用来实现 class。使用基于类语言的开发人员会对这些结构感到熟悉，但它们是不一样的。 JavaScript 仍然是基于原型的。这些新的关键字包括 class, constructor, static, extends, 和 super.

"use strict";

class Polygon {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

class Square extends Polygon {

constructor(sideLength) {

super(sideLength, sideLength);

}

get area() {

return this.height \* this.width;

}

set sideLength(newLength) {

this.height = newLength;

this.width = newLength;

}

}

var square = new Square(2);

1. 性能

在原型链上查找属性比较耗时，对性能有副作用，这在性能要求苛刻的情况下很重要。另外，试图访问不存在的属性时会遍历整个原型链。

遍历对象的属性时，原型链上的每个可枚举属性都会被枚举出来。

检测对象的属性是定义在自身上还是在原型链上，有必要使用 hasOwnProperty 方法，所有继承自 Object.proptotype 的对象都包含这个方法。

**hasOwnProperty 是 JavaScript 中唯一一个只涉及对象自身属性而不会遍历原型链的方法。**