JavaScript核心

2013/11/25 | 分类： IT技术 | 2 条评论 | 标签： JAVASCRIPT

原文出处： Dmitry Soshnikov 译文出处： 魏志峰（@JeremyWei）翻译+投稿

这篇文章是「深入ECMA-262-3」系列的一个概览和摘要。面向读者：经验丰富的程序员，专家。我们以思考对象的概念做为开始，这是ECMAScript的基础。

对象

ECMAScript做为一个高度抽象的面向对象语言，是通过对象来交互的。即使ECMAScript里边也有基本类型，但是，当需要的时候，它们也会被转换成对象。

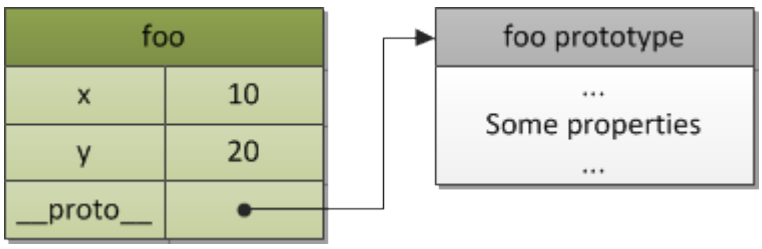
一个对象就是一个属性集合，并拥有一个独立的`prototype`（原型）对象。这个`prototype`可以是一个对象或者`null`。

让我们看一个关于对象的基本例子。一个对象的prototype是以内部的[[Prototype]]属性来引用的。但是，在示意图里边我们将会使用`__<internal-property>__`下划线标记来替代两个括号，对于prototype对象来说是：`__proto__`。

对于以下代码：

```
var foo = {
  x: 10,
  y: 20
};
```

我们拥有一个这样的结构，两个明显的自身属性和一个隐含的`__proto__`属性，这个属性是对`foo`原型对象的引用：



这些prototype有什么用？让我们以原型链（prototype chain）的概念来回答这个问题。

原型链

原型对象也是简单的对象并且可以拥有它们自己的原型。如果一个原型对象的原型是一个非null的引用，那么以此类推，这就叫作原型链。

原型链是一个用来实现继承和共享属性的有限对象链。

考虑这么一个情况，我们拥有两个对象，它们之间只有一小部分不同，其他部分都相同。显然，对于一个设计良好的系统，我们将会重用相似的功能/代码，而不是在每个单独的对象中重复它。在基于类的系统中，这个代码重用风格叫作类继承－你把相似的功能放入类A中，然后类 B和类 C继承类 A，并且拥有它们自己的一些小的额外变动。

ECMAScript中没有类的概念。但是，代码重用的风格并没有太多不同（尽管从某些方面来说比基于类（class-based）的方式要更加灵活）并且通过原型链来实现。这种继承方式叫作委托继承(delegation based inheritance)（或者，更贴近ECMAScript一些，叫作原型继承(prototype based inheritance)）。

跟例子中的类A，B，C相似，在ECMAScript中你创建对象：a，b，c。于是，对象a中存储对象b和c中通用的部分。然后b和c只存储它们自身的额外属性或者方法。

```
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
```

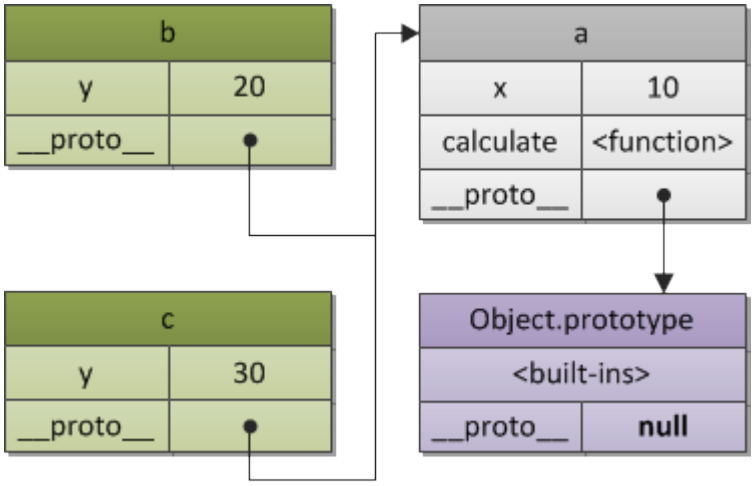
足够简单，是不是？我们看到b和c访问到了在对象a中定义的calculate方法。这是通过原型链实现的。

规则很简单：如果一个属性或者一个方法在对象自身中无法找到（也就是对象自身没有一个那样的属性），然后它会尝试在原型链中寻找这个属性/方法。如果这个属性在原型中没有查找到，那么将会查找这个原型的原型，以此类推，遍历整个原型链（当然这在类继承中也是一样的，当解析一个继承的方法的时候－我们遍历*class*链（class chain））。第一个被查找到的同名属性/方法会被使用。因此，一个被查找到的属性叫作继承属性。如果在遍历了整个原型链之后还是没有查找到这个属性的话，返回undefined值。

注意，继承方法中所使用的this的值被设置为原始对象，而并不是在其中查找到这个方法的（原型）对象。也就是，在上面的例子中this.y取的是b和c中的值，而不是a中的值。但是，this.x是取的是a中的值，并且又一次通过原型链机制完成。

如果没有明确为一个对象指定原型，那么它将会使用__proto__的默认值－Object.prototype。Object.prototype对象自身也有一个__proto__属性，这是原型链的终点并且值为null。

下一张图展示了对象a，b，c之间的继承层级：



注意： ES5标准化了一个实现原型继承的可选方法，即使用`Object.create`函数：

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

你可以在对应的章节获取到更多关于ES5新API的信息。 ES6标准化了 `__proto__`属性，并且可以在对象初始化的时候使用它。

通常情况下需要对象拥有相同或者相似的状态结构（也就是相同的属性集合），赋以不同的状态值。在这个情况下我们可能需要使用构造函数(constructor function)，其以指定的模式来创造对象。

构造函数

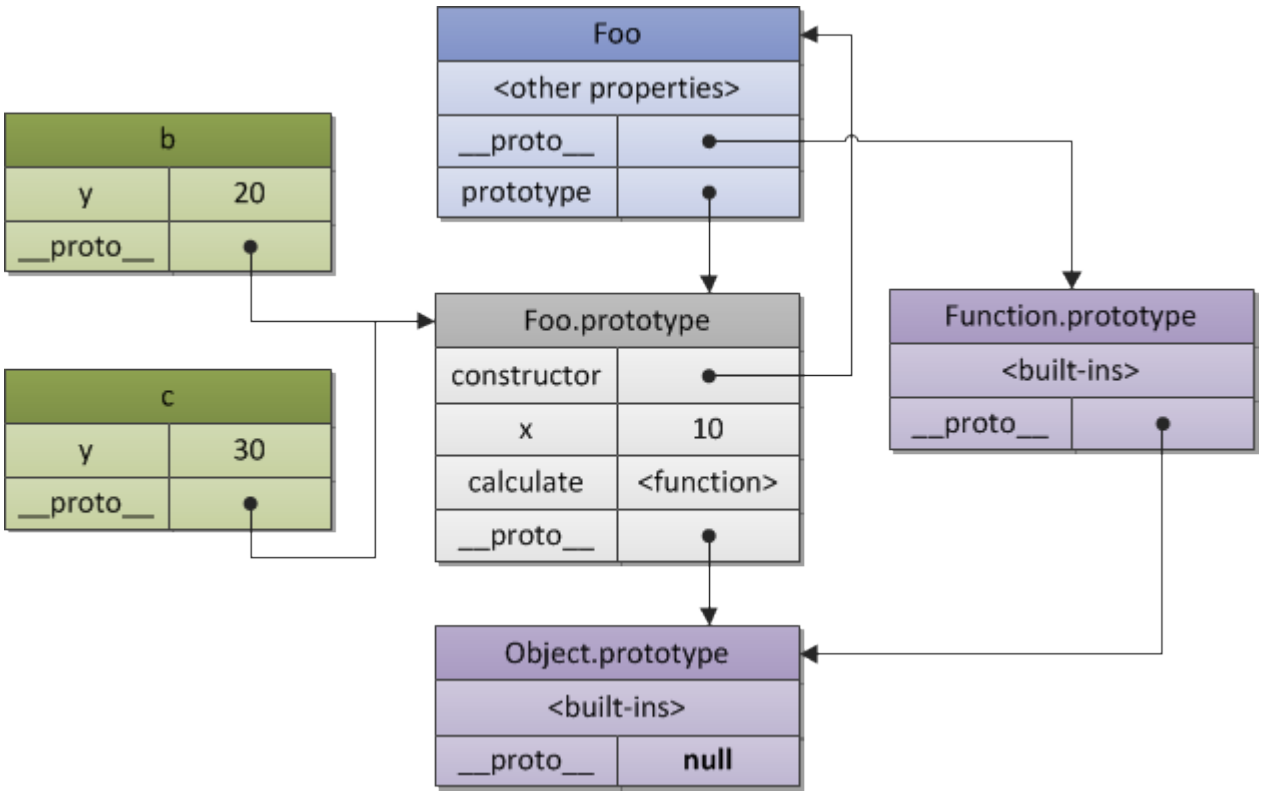
除了以指定模式创建对象之外，构造函数也做了另一个有用的事情－它自动地为新创建的对象设置一个原型对象。这个原型对象存储在`ConstructorFunction.prototype`属性中。

换句话说，我们可以使用构造函数来重写上一个拥有对象`b`和对象`c`的例子。因此，对象`a`（一个原型对象）的角色由`Foo.prototype`来扮演：

```
// a constructor function
function Foo(y) {
  // which may create objects
  // by specified pattern: they have after
  // creation own "y" property
  this.y = y;
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:

// inherited property "x"
Foo.prototype.x = 10;

// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
};

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

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

```

```
// let's show that we reference
// properties we expect

console.log(

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

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is a
  // reference to the constructor function itself;
  // instances "b" and "c" may found it via
  // delegation and use to check their constructor

  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

);
```

这个代码可以表示为如下关系：



这张图又一次说明了每个对象都有一个原型。构造函数Foo也有自己的__proto__，值为Function.prototype，Function.prototype也通过其__proto__属性关联到Object.prototype。因此，重申一下，Foo.prototype就是Foo的一个明确的属性，指向对象b和对象c的原型。

正式来说，如果思考一下分类的概念（并且我们已经对Foo进行了分类），那么构造函数和原型对象合在一起可以叫作「类」。实际上，举个例子，Python的第一级（first-class）动态类（dynamic classes）显然是以同样的属性/方法处理方案来实现的。从这个角度来说，Python中的类就是ECMAScript使用的委托继承的一个语法糖。

注意: 在ES6中「类」的概念被标准化了，并且实际上以一种构建在构造函数上面的语法糖来实现，就像上面描述的一样。从这个角度来看原型链成为了类继承的一种具体实现方式：

```
// ES6
class Foo {
  constructor(name) {
    this._name = name;
```

```
  }

  getName() {
    return this._name;
  }
}

class Bar extends Foo {
  getName() {
    return super.getName() + ' Doe';
  }
}

var bar = new Bar('John');
console.log(bar.getName()); // John Doe
```

有关这个主题的完整、详细的解释可以在ES3系列的第七章找到。分为两个部分：7.1 面向对象.基本理论，在那里你将会找到对各种面向对象范例、风格的描述以及它们和ECMAScript之间的对比，然后在7.2 面向对象.ECMAScript实现，是对ECMAScript中面向对象的介绍。

现在，在我们知道了对象的基础之后，让我们看看运行时程序的执行（runtime program execution）在ECMAScript中是如何实现的。这叫作执行上下文栈（execution context stack），其中的每个元素也可以抽象成为一个对象。是的，ECMAScript几乎在任何地方都和对象的概念打交道;)

执行上下文堆栈

这里有三种类型的ECMAScript代码：全局代码、函数代码和*eval*代码。每个代码是在其执行上下文（execution context）中被求值的。这里只有一个全局上下文，可能有多个函数执行上下文以及*eval*执行上下文。对一个函数的每次调用，会进入到函数执行上下文中，并对函数代码类型进行求值。每次对`eval`函数进行调用，会进入*eval*执行上下文并对其代码进行求值。

注意，一个函数可能会创建无数的上下文，因为对函数的每次调用（即使这个函数递归的调用自己）都会生成一个具有新状态的上下文：

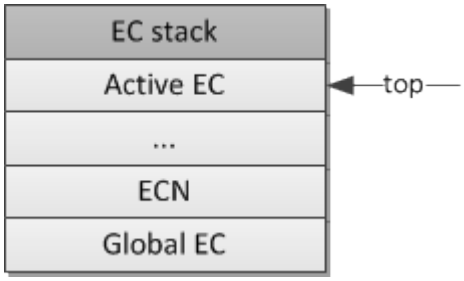
```
function foo(bar) {}

// call the same function,
// generate three different
// contexts in each call, with
// different context state (e.g. value
// of the "bar" argument)

foo(10);
foo(20);
foo(30);
```

一个执行上下文可能会触发另一个上下文，比如，一个函数调用另一个函数（或者在全局上下文中调用一个全局函数），等等。从逻辑上来说，这是以栈的形式实现的，它叫作执行上下文栈。

一个触发其他上下文的上下文叫作*caller*。被触发的上下文叫作*callee*。callee在同一时间可能是一些其他callee的caller（比如，一个在全局上下文中被调用的函数，之后调用了一些内部函数）。

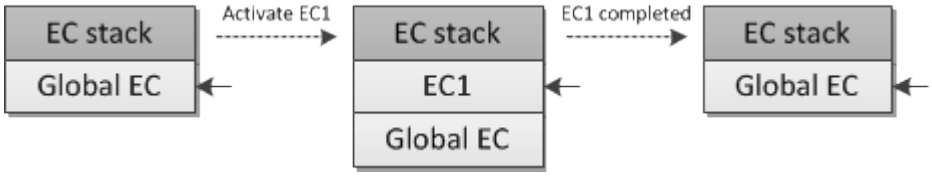
当一个caller触发（调用）了一个callee，这个caller会暂缓自身的执行，然后把控制权传递给callee。这个callee被push到栈中，并成为一个运行中（活动的）执行上下文。在callee的上下文结束后，它会把控制权返回给caller，然后caller的上下文继续执行（它可能触发其他上下文）直到它结束，以此类推。callee可能简单的返回或者由于异常而退出。一个抛出的但是没有被捕获的异常可能退出（从栈中pop）一个或者多个上下文。

换句话说，所有ECMAScript程序的运行时可以用执行上下文（*EC*）栈来表示，栈顶是当前活跃(active)上下文：



当程序开始的时候它会进入全局执行上下文，此上下文位于栈底并且是栈中的第一个元素。然后全局代码进行一些初始化，创建需要的对象和函数。在全局上下文的执行过程中，它的代码可能触发其他（已经创建完成的）函数，这些函数将会进入它们自己的执行上下文，向栈中push新的元素，以此类推。当初始化完成之后，运行时系统（runtime system）就会等待一些事件（比如，用户鼠标点击），这些事件将会触发一些函数，从而进入新的执行上下文中。

在下个图中，拥有一些函数上下文`EC1`和全局上下文`Global EC`，当`EC1`进入和退出全局上下文的时候下面的栈将会发生变化：



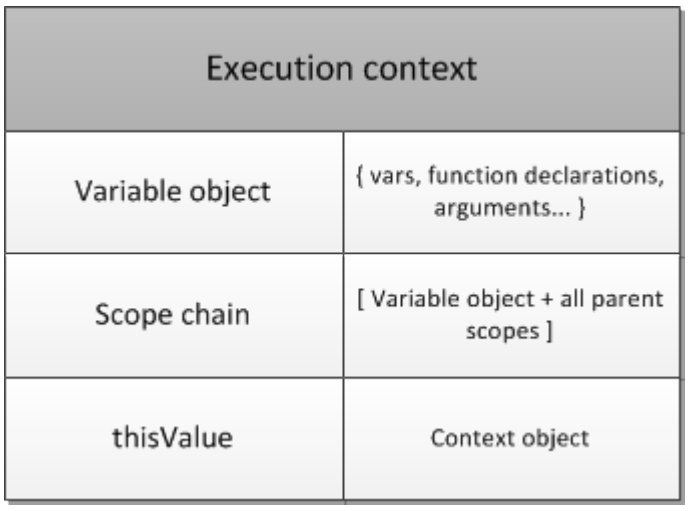
这就是ECMAScript的运行时系统如何真正地管理代码执行的。

更多有关ECMAScript中执行上下文的信息可以在对应的第一章 执行上下文中获取。

像我们所说的，栈中的每个执行上下文都可以用一个对象来表示。让我们来看看它的结构以及一个上下文到底需要什么状态（什么属性）来执行它的代码。

执行上下文

一个执行上下文可以抽象的表示为一个简单的对象。每一个执行上下文拥有一些属性（可以叫作上下文状态）用来跟踪和它相关的代码的执行过程。在下图中展示了一个上下文的结构：



除了这三个必需的属性（一个变量对象（variable objec），一个***this***值以及一个作用域链（scope chain））之外，执行上下文可以拥有任何附加的状态，这取决于实现。

让我们详细看看上下文中的这些重要的属性。

变量对象

变量对象是与执行上下文相关的数据作用域。它是一个与上下文相关的特殊对象，其中存储了在上下文中定义的变量和函数声明。

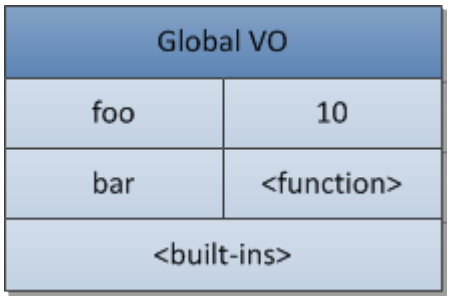
注意，函数表达式（与函数声明相对）不包含在变量对象之中。

变量对象是一个抽象概念。对于不同的上下文类型，在物理上，是使用不同的对象。比如，在全局上下文中变量对象就是全局对象本身（这就是为什么我们可以通过全局对象的属性名来关联全局变量）。

让我们在全局执行上下文中考虑下面这个例子：

```
var foo = 10;

function bar() {} // function declaration, FD
(function baz() {}); // function expression, FE

console.log(
  this.foo == foo, // true
  window.bar == bar // true
);

console.log(baz); // ReferenceError, "baz" is not defined
```

之后，全局上下文的变量对象（variable objec，简称VO）将会拥有如下属性：



再看一遍，函数baz是一个函数表达式，没有被包含在变量对象之中。这就是为什么当我们想要在函数自身之外访问它的时候会出现ReferenceError。

注意，与其他语言（比如C/C++）相比，在ECMAScript中只有函数可以创建一个新的作用域。在函数作用域中所定义的变量和内部函数在函数外边是不能直接访问到的，而且并不会污染全局变量对象。

使用eval我们也会进入一个新的（eval类型）执行上下文。无论如何，eval使用全局的变量对象或者使用caller（比如eval被调用时所在的函数）的变量对象。

那么函数和它的变量对象是怎么样的？在函数上下文中，变量对象是以活动对象（activation object）来表示的。

活动对象

当一个函数被caller所触发（被调用），一个特殊的对象，叫作活动对象（activation object）将会被创建。这个对象中包含形参和那个特殊的arguments对象（是对形参的一个映射，但是值是通过索引来获取）。活动对象之后会做为函数上下文的变量对象来使用。

换句话说，函数的变量对象也是一个同样简单的变量对象，但是除了变量和函数声明之外，它还存储了形参和arguments对象，并叫作活动对象。

考虑如下例子：

```
function foo(x, y) {
  var z = 30;
  function bar() {} // FD
  (function baz() {}); // FE
}

foo(10, 20);
```

我们看下函数foo的上下文中的活动对象（activation object，简称AO）：

Activation object	
x	10
y	20
arguments	{0: 10, 1: 20, ..}
z	30
bar	<function>

并且函数表达式baz还是没有被包含在变量/活动对象中。

关于这个主题所有细节方面（像变量和函数声明的提升问题（hoisting））的完整描述可以在同名的章节第二章 变量对象中找到。

注意，在ES5中变量对象和活动对象被并入了词法环境模型（lexical environments model），详细的描述可以在对应的章节找到。

然后我们向下一个部分前进。众所周知，在ECMAScript中我们可以使用内部函数，然后在这些内部函数我们可以引用父函数的变量或者全局上下文中的变量。当我们把变量对象命名为上下文的作用域对象，与上面讨论的原型链相似，这里有一个叫作作用域链的东西。

作用域链

作用域链是一个对象列表，上下文代码中出现的标识符在这个列表中进行查找。

这个规则还是与原型链同样简单以及相似：如果一个变量在函数自身的作用域（在自身的变量/活动对象）中没有找到，那么将会查找它父函数（外层函数）的变量对象，以此类推。

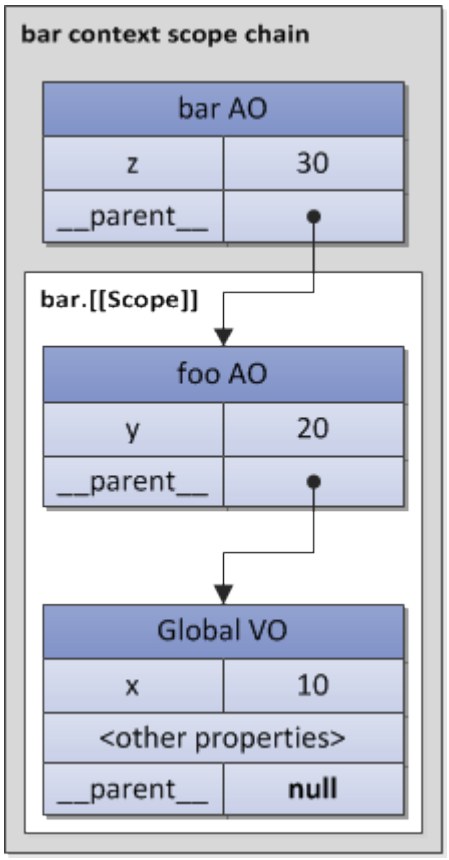
就上下文而言，标识符指的是：变量名称，函数声明，形参，等等。当一个函数在其代码中引用一个不是局部变量（或者局部函数或者一个形参）的标识符，那么这个标识符就叫作自由变量。搜索这些自由变量(free variables)正好就要用到作用域链。

在通常情况下，作用域链是一个包含所有父（函数）变量对象__加上（在作用域链头部的）函数自身变量/活动对象的一个列表。但是，这个作用域链也可以包含任何其他对象，比如，在上下文执行过程中动态加入到作用域链中的对象－像*with*对象或者特殊的*catch*从句（catch-clauses）对象。

当解析（查找）一个标识符的时候，会从作用域链中的活动对象开始查找，然后（如果这个标识符在函数自身的活动对象中没有被查找到）向作用域链的上一层查找－重复这个过程，就和原型链一样。

```
var x = 10;

(function foo() {
  var y = 20;
  (function bar() {
    var z = 30;
    // "x" and "y" are "free variables"
    // and are found in the next (after
    // bar's activation object) object
    // of the bar's scope chain
    console.log(x + y + z);
  })();
})();
```

我们可以假设通过隐式的__parent__属性来和作用域链对象进行关联，这个属性指向作用域链中的下一个对象。这个方案可能在真实的Rhino代码中经过了测试，并且这个技术很明确得被用于ES5的词法环境中（在那里被叫作outer连接）。作用域链的另一个表现方式可以是一个简单的数组。利用__parent__概念，我们可以用下面的图来表现上面的例子（并且父变量对象存储在函数的[[Scope]]属性中）：



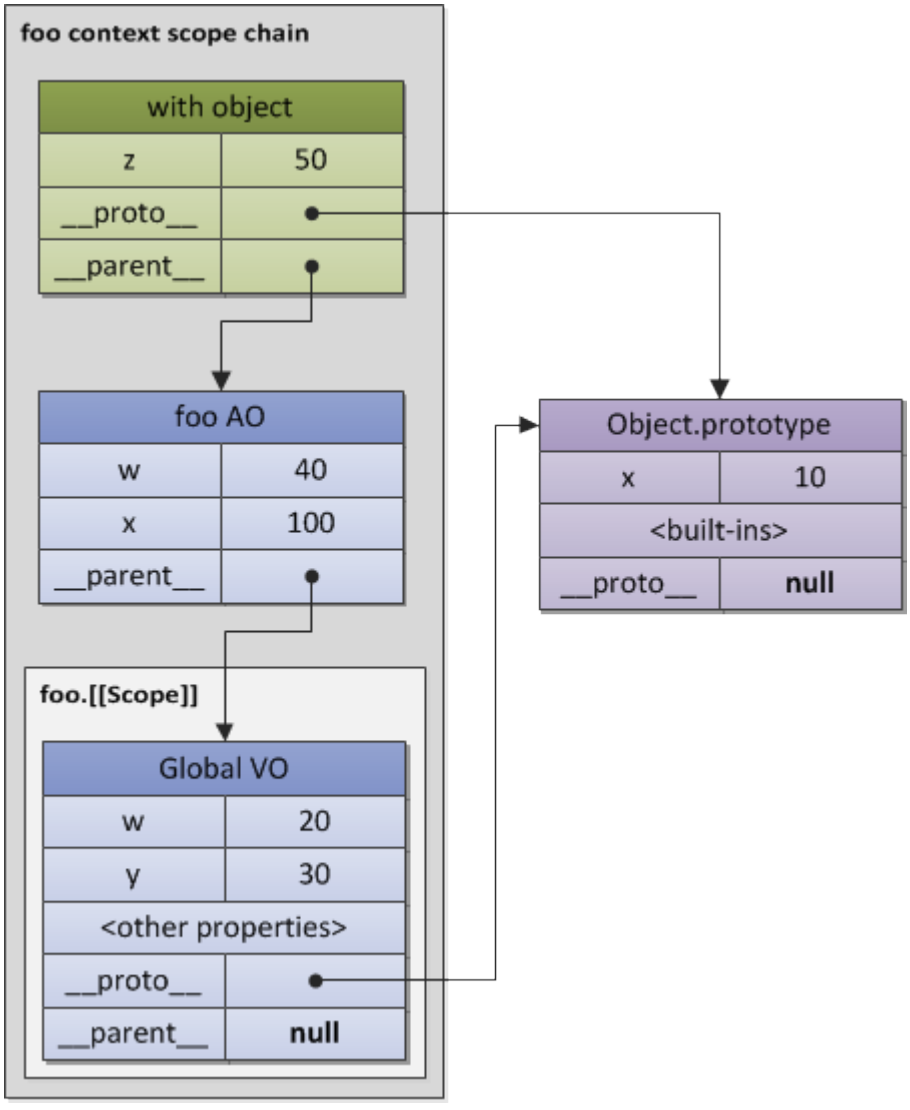
在代码执行过程中，作用域链可以通过使用with语句和catch从句对象来增强。并且由于这些对象是简单的对象，它们可以拥有原型（和原型链）。这个事实导致作用域链查找变为两个维度：（1）首先是作用域链连接，然后（2）在每个作用域链连接上—深入作用域链连接的原型链（如果此连接拥有原型）。

对于这个例子：

```
Object.prototype.x = 10;

var w = 20;
var y = 30;

// in SpiderMonkey global object
// i.e. variable object of the global
// context inherits from "Object.prototype",
// so we may refer "not defined global
// variable x", which is found in
// the prototype chain

```

```
13  console.log(x); // 10
14
15  (function foo() {
16
17    // "foo" local variables
18    var w = 40;
19    var x = 100;
20
21    // "x" is found in the
22    // "Object.prototype", because
23    // {z: 50} inherits from it
24
25    with ({z: 50}) {
26      console.log(w, x, y , z); // 40, 10, 30, 50
27    }
28
29    // after "with" object is removed
30    // from the scope chain, "x" is
31    // again found in the AO of "foo" context;
32    // variable "w" is also local
33    console.log(x, w); // 100, 40
34
35    // and that's how we may refer
36    // shadowed global "w" variable in
37    // the browser host environment
38    console.log(window.w); // 20
39
40  })();
```

我们可以给出如下的结构（确切的说，在我们查找__parent__连接之前，首先查找__proto__链）：



注意，不是在所有的实现中全局对象都是继承自Object.prototype。上图中描述的行为（从全局上下文中引用「未定义」的变量x）可以在诸如SpiderMonkey引擎中进行测试。

由于所有父变量对象都存在，所以在内部函数中获取父函数中的数据没有什么特别－我们就是遍历作用域链去解析（搜寻）需要的变量。就像我们上边提及的，在一个上下文结束之后，它所有的状态和它自身都会被销毁。在同一时间父函数可能会返回一个内部函数。而且，这个返回的函数之后可能在另一个上下文中被调用。如果自由变量的上下文已经「消失」了，那么这样的调用将会发生什么？通常来说，有一个概念可以帮助我们解决这个问题，叫作（词法）闭包，其在ECMAScript中就是和作用域链的概念紧密相关的。

闭包

在ECMAScript中，函数是第一级（first-class）对象。这个术语意味着函数可以做为参数传递给其他函数（在那种情况下，这些参数叫作「函数类型参数」（funargs，是"functional arguments"的简称））。接收「函数类型参数」的函数叫作高阶函数或者，贴近数学一些，叫作高阶操作符。同样函数也可以从其他函数中返回。返回其他函数的函数叫作以函数为值（function valued）的函数（或者叫作拥有函数类值的函数（functions with functional value））。

这有两个在概念上与「函数类型参数（funargs）」和「函数类型值（functional values）」相关的问题。并且这两个子问题在*"Funarg problem"*（或者叫作"functional argument"问题）中很普遍。为了解决整个*"funarg problem"*，闭包（closure）的概念被创造了出来。我们详细的描述一下这两个子问题（我们将会看到这两个问题在ECMAScript中都是使用图中所提到的函数的`[[Scope]]`属性来解决的）。

「funarg问题」的第一个子问题是「向上*funarg*问题」（upward funarg problem）。它会在当一个函数从另一个函数向上返回（到外层）并且使用上面所提到的自由变量的时候出现。为了在即使父函数上下文结束的情况下也能访问其中的变量，内部函数在被创建的时候会在它的`[[Scope]]`属性中保存父函数的作用域链。所以当函数被调用的时候，它上下文的作用域链会被格式化成活动对象与`[[Scope]]`属性的和（实际上就是我们刚刚在上图中所看到的）：

```
Scope chain = Activation object + [[Scope]]
```

再次注意这个关键点－确切的说在创建时刻－函数会保存父函数的作用域链，因为确切的说这个保存下来的作用域链将会在未来的函数调用时用来查找变量。

```
function foo() {
  var x = 10;
  return function bar() {
    console.log(x);
  };
}

// "foo" returns also a function
// and this returned function uses
// free variable "x"

var returnedFunction = foo();

// global variable "x"
var x = 20;

// execution of the returned function

returnedFunction(); // 10, but not 20
```

这个类型的作用域叫作静态（或者词法）作用域。我们看到变量`x`在返回的`bar`函数的`[[Scope]]`属性中被找到。通常来说，也存在动态作用域，那么上面例子中的变量`x`将会被解析成`20`，而不是`10`。但是，动态作用域在ECMAScript中没有被使用。

「funarg问题」的第二个部分是「向下*funarg*问题」。这种情况下可能会存在一个父上下文，但是在解析标识符的时候可能会模糊不清。问题是：标识符该使用哪个作用域的值－以静态的方式存储在函数创建时刻的还是在执行过程中以动态方式生成的（比如*caller*的作用域）？为了避免这种模棱两可的情况并形成闭包，静态作用域被采用：

```
// global "x"
```

```
var x = 10;

// global function
function foo() {
  console.log(x);
}

(function (funArg) {

  // local "x"
  var x = 20;

  // there is no ambiguity,
  // because we use global "x",
  // which was statically saved in
  // [[Scope]] of the "foo" function,
  // but not the "x" of the caller's scope,
  // which activates the "funArg"

  funArg(); // 10, but not 20

})(foo); // pass "down" foo as a "funarg"
```

我们可以断定静态作用域是一门语言拥有闭包的必需条件。但是，一些语言可能会同时提供动态和静态作用域，允许程序员做选择－什么应该包含（closure）在内和什么不应包含在内。由于在ECMAScript中只使用了静态作用域（比如我们对于funarg问题的两个子问题都有解决方案），所以结论是：*ECMAScript*完全支持闭包，技术上是通过函数的[[Scope]]属性实现的。现在我们可以给闭包下一个准确的定义：

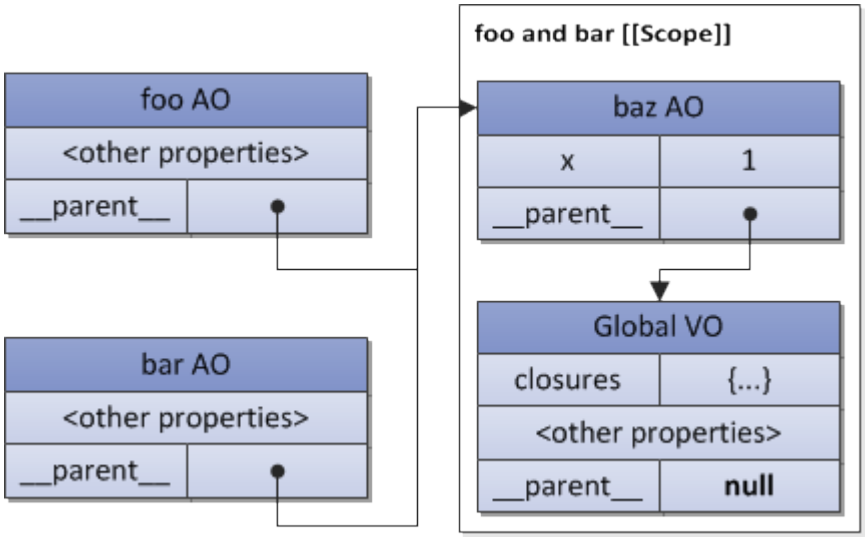
闭包是一个代码块（在ECMAScript是一个函数）和以静态方式/词法方式进行存储的所有父作用域的一个集合体。所以，通过这些存储的作用域，函数可以很容易的找到自由变量。

注意，由于每个（标准的）函数都在创建的时候保存了[[Scope]]，所以理论上来讲，ECMAScript中的所有函数都是闭包。

另一个需要注意的重要事情是，多个函数可能拥有相同的父作用域（这是很常见的情况，比如当我们拥有两个内部/全局函数的时候）。在这种情况下，[[Scope]]属性中存储的变量是在拥有相同父作用域链的所有函数之间共享的。一个闭包对变量进行的修改会体现在另一个闭包对这些变量的读取上：

```
function baz() {
  var x = 1;
  return {
    foo: function foo() { return ++x; },
    bar: function bar() { return --x; }
  };
}

var closures = baz();

console.log(
  closures.foo(), // 2
  closures.bar()  // 1
);
```

以上代码可以通过下图进行说明：



确切来说这个特性在循环中创建多个函数的时候会使人非常困惑。在创建的函数中使用循环计数器的时候，一些程序员经常会得到非预期的结果，所有函数中的计数器都是同样的值。现在是到了该揭开谜底的时候了—因为所有这些函数拥有同一个`[[Scope]]`，这个属性中的循环计数器的值是最后一次所赋的值。

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = function () {
    alert(k);
  };
}

data[0](); // 3, but not 0
data[1](); // 3, but not 1
data[2](); // 3, but not 2
```

这里有几种技术可以解决这个问题。其中一种是在作用域链中提供一个额外的对象—比如，使用额外函数：

```
var data = [];

for (var k = 0; k < 3; k++) {
  data[k] = (function (x) {
    return function () {
      alert(x);
    };
  })(k); // pass "k" value
}

// now it is correct
data[0](); // 0
data[1](); // 1
data[2](); // 2
```

对闭包理论和它们的实际应用感兴趣的同学可以在第六章 闭包中找到额外的信息。如果想获取更多关于作用域链的信息，可以看一下同名的第四章 作用域链。

然后我们移动到下个部分，考虑一下执行上下文的最后一个属性。这就是关于`this`值的概念。

This

`this`是一个与执行上下文相关的特殊对象。因此，它可以叫作上下文对象（也就是用来指明执行上下文是在哪个上下文中被触发的对象）。

任何对象都可以做为上下文中的`this`的值。我想再一次澄清，在一些对ECMAScript执行上下文和部分`this`的描述中的所产生误解。`this`经常被错误的描述成是变量对象的一个属性。这类错误存在于比如像这本书中（即使如此，这本书的相关章节还是十分不错的）。再重复一次：

this是执行上下文的一个属性，而不是变量对象的一个属性。

这个特性非常重要，因为与变量相反，*this*从不会参与到标识符解析过程。换句话说，在代码中当访问this的时候，它的值是直接从执行上下文中获取的，并不需要任何作用域链查找。this的值只在进入上下文的时候进行一次确定。

顺便说一下，与ECMAScript相反，比如，Python的方法都会拥有一个被当作简单变量的self参数，这个变量的值在各个方法中是相同的的并且在执行过程中可以被更改成其他值。在ECMAScript中，给this赋一个新值是不可能的，因为，再重复一遍，它不是一个变量并且不存在于变量对象中。

在全局上下文中，this就等于全局对象本身（这意味着，这里的this等于变量对象）：

```
var x = 10;

console.log(
  x, // 10
  this.x, // 10
  window.x // 10
);
```

在函数上下文的情况下，对函数的每次调用，其中的this值可能是不同的。这个this值是通过函数调用表达式（也就是函数被调用的方式）的形式由*caller*所提供的。举个例子，下面的函数foo是一个*callee*，在全局上下文中被调用，此上下文为caller。让我们通过例子看一下，对于一个代码相同的函数，this值是如何在不同的调用中（函数触发的不同方式），由caller给出不同的结果的：

```
// the code of the "foo" function
// never changes, but the "this" value
// differs in every activation

function foo() {
  alert(this);
}

// caller activates "foo" (callee) and
// provides "this" for the callee

foo(); // global object
foo.prototype.constructor(); // foo.prototype

var bar = {
  baz: foo
};

bar.baz(); // bar

(bar.baz)(); // also bar
(bar.baz = bar.baz)(); // but here is global object
(bar.baz, bar.baz)(); // also global object
(false || bar.baz)(); // also global object

var otherFoo = bar.baz;
otherFoo(); // again global object
```

为了深入理解this为什么（并且更本质一些—如何）在每个函数调用中可能会发生变化，你可以阅读第三章 This。在那里，上面所提到的情况都会有详细的讨论。

总结

通过本文我们完成了对概要的综述。尽管，它看起来并不像是「概要」;)。对所有这些主题进行完全的解释需要一本完整的书。我们只是没有涉及到两个大的主题：函数（和不同函数之间的区别，比如，函数声明和函数表达式）和ECMAScript中所使用的求值策略(evaluation strategy)。这两个主题是可以ES3系列的在对应章节找到：第五章 函数和第八章 求值策略。