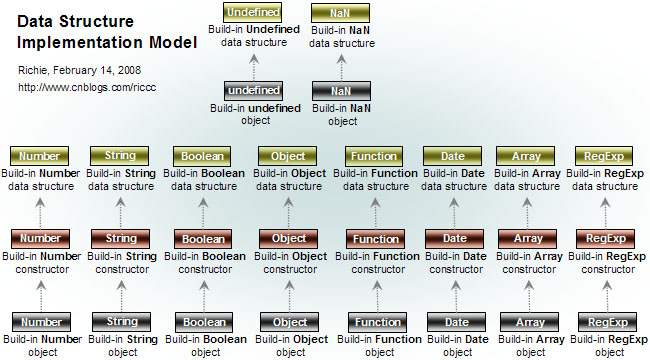
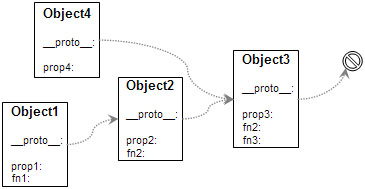
**数据类型**  
**基本数据类型**  
基本数据类型是JS语言最底层的实现。  
简单数值类型: 有Undefined, Null, Boolean, Number和String。注意，描述中的英文单词在这里仅指数据类型的名称，并不特指JS的全局对象N an, Boolean, Number, String等，它们在概念上的区别是比较大的。  
对象: 一个无序属性的集合，这些属性的值为简单数值类型、对象或者函数。同上，这里的对象并不特指全局对象Object。  
函数: 函数是对象的一种，实现上内部属性[[Class]]值为"Function"，表明它是函数类型，除了对象的内部属性方法外，还有[[Construct]]、[[Call]]、[[Scope]]等内部属性。函数作为函数调用与构造器(使用new关键字创建实例对象)的处理机制不一样(Function对象除外)，内部方法[[Construct]]用于实现作为构造器的逻辑，方法[[Call]]实现作为函数调用的逻辑。同上，这里的函数并不特指全局对象Function。  
函数在JS这个Prototype语言中可以看作是面向对象语言的类，可以用它来构造对象实例。既然函数可以看作是类，所以每一个函数可以看作是一种扩展数据类型。  
  
**内置数据类型(内置对象)**  
Function: 函数类型的用户接口。  
Object: 对象类型的用户接口。  
Boolean, Number, String: 分别为这三种简单数值类型的对象包装器，对象包装在概念上有点类似C#中的Box/Unbox。  
Date, Array, RegExp: 可以把它们看作是几种内置的扩展数据类型。  
  
首先，Function, Object, Boolean, Number, String, Date, Array, RegExp等都是JavaScript语言的内置对象，它们都可以看作是函数的派生类型，例如Number instanceof Function为true，Number instanceof Object为true。在这个意义上，可以将它们跟用户定义的函数等同看待。  
其次，它们各自可以代表一种数据类型，由JS引擎用native code或内置的JS代码实现，是暴露给开发者对这些内置数据类型进行操作的接口。在这个意义上，它们都是一种抽象的概念，后面隐藏了具体的实现机制。  
在每一个提到Number, Function等单词的地方，应该迅速的在思维中将它们实例化为上面的两种情况之一。  
  
**数据类型实现模型描述**      
Build-in \*\*\* data structure: 指JS内部用于实现\*\*\*类型的数据结构，这些结构我们基本上无法直接操作。  
Build-in \*\*\* object: 指JS内置的Number, String, Boolean等这些对象，这是JS将内部实现的数据类型暴露给开发者使用的接口。  
Build-in \*\*\* constructor: 指JS内置的一些构造器，用来构造相应类型的对象实例。它们被包装成函数对象暴露出来，例如我们可以使用下面的方法访问到这些函数对象:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
//access the build-in number constructor  
var number = new Number(123);  
var numConstructor1 = number.constructor; //or  
var numConstructor2 = new Object(123).constructor;  
//both numConstructor1 and numConstructor2 are the build-in Number constructor  
numConstructor1 == numConstructor2 //result: true  
//access the build-in object constructor  
var objConstructor1 = {}.constructor; //or  
var objConstructor2 = new Object().constructor;  
//both objConstructor1 and objConstructor2 are the build-in Object constructor  
objConstructor1==objConstructor2 //result: true

具体实现上，上图中横向之间可能也存在关联，例如对于build-in data structure和constructor，Function、 Date、 Array、 RegExp等都可以继承Object的结构而实现，但这是具体实现相关的事情了。  
  
**关于简单数值类型的对象化**  
这是一个细微的地方，下面描述对于Boolean, String和Number这三种简单数值类型都适用，以Number为例说明。  
JS规范要求: 使用var num1=123;这样的代码，直接返回基本数据类型，就是说返回的对象不是派生自Number和Object类型，用num1 instanceof Object测试为false；使用new关键字创建则返回Number类型，例如var num2=new Number(123); num2 instanceof Number为true。  
将Number当作函数调用，返回结果会转换成简单数值类型。下面是测试代码:

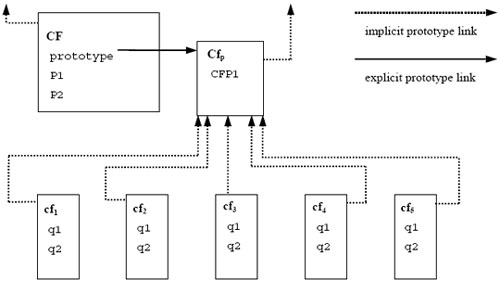
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
var num1 = new Number(123); //num1 derived from Number & Object  
num1 instanceof Number //result: true  
num1 instanceof Object //result: true  
//convert the num1 from Number type to primitive type, so it's no longer an instance of Number or Object  
num1 = Number(num1);   
num1 instanceof Number //result: false  
num1 instanceof Object //result: false  
var num2 = 123; //num2 is a primitive type  
num2 instanceof Number //result: false  
num2 instanceof Object //result: false

虽然我们得到了一个简单数值类型，但它看起来仍然是一个JS Object对象，具有Object以及相应类型的所有属性和方法，使用上基本没有差别，唯一不同之处是instanceof的测试结果。  
  
**Prototype继承  
Prototype**  
每个对象都有一个[[Prototype]]的内部属性，它的值为null或者另外一个对象。函数对象都有一个显示的prototype属性，它并不是内部[[Prototype]]属性。不同的JS引擎实现者可以将内部[[Prototype]]属性命名为任何名字，并且设置它的可见性，只在JS引擎内部使用。虽然无法在JS代码中访问到内部[[Prototype]](FireFox中可以，名字为\_\_proto\_\_因为Mozilla将它公开了)，但可以使用对象的isPrototypeOf()方法进行测试，注意这个方法会在整个Prototype链上进行判断。  
使用obj.propName访问一个对象的属性时，按照下面的步骤进行处理(假设obj的内部[[Prototype]]属性名为\_\_proto\_\_):  
1. 如果obj存在propName属性，返回属性的值，否则  
2. 如果obj.\_\_proto\_\_为null，返回undefined，否则  
3. 返回obj.\_\_proto\_\_.propName  
调用对象的方法跟访问属性搜索过程一样，因为方法的函数对象就是对象的一个属性值。  
提示: 上面步骤中隐含了一个递归过程，步骤3中obj.\_\_proto\_\_是另外一个对象，同样将采用1, 2, 3这样的步骤来搜索propName属性。  
  
例如下图所示，object1将具备属性prop1, prop2, prop3以及方法fn1, fn2, fn3。图中虚线箭头表示prototype链。  
      
这就是基于Prototype的继承和共享。其中object1的方法fn2来自object2，概念上即object2重写了object3的方法fn2。  
JavaScript对象应当都通过prototype链关联起来，最顶层是Object，即对象都派生自Object类型。  
  
类似C++等面向对象语言用类(被抽象了的类型)来承载方法，用对象(实例化对象)承载属性，Prototype语言只用实例化的对象来承载方法和属性。本质区别是前者基于内存结构的描述来实现继承，后者基于具体的内存块实现。  
  
**对象创建过程**  
JS中只有函数对象具备类的概念，因此要创建一个对象，必须使用函数对象。函数对象内部有[[Construct]]方法和[[Call]]方法，[[Construct]]用于构造对象，[[Call]]用于函数调用，只有使用new操作符时才触发[[Construct]]逻辑。  
var obj=new Object(); 是使用内置的Object这个函数对象创建实例化对象obj。var obj={};和var obj=[];这种代码将由JS引擎触发Object和Array的构造过程。function fn(){}; var myObj=new fn();是使用用户定义的类型创建实例化对象。  
  
new Fn(args)的创建过程如下(即函数对象的[[Construct]]方法处理逻辑，对象的创建过程)。另外函数对象本身的创建过程(指定义函数或者用Function创建一个函数对象等方式)虽然也使用了下面的处理逻辑，但有特殊的地方，后面再描述。  
1. 创建一个build-in object对象obj并初始化  
2. 如果Fn.prototype是Object类型，则将obj的内部[[Prototype]]设置为Fn.prototype，否则obj的[[Prototype]]将为其初始化值(即Object.prototype)  
3. 将obj作为this，使用args参数调用Fn的内部[[Call]]方法  
    3.1 内部[[Call]]方法创建当前执行上下文  
    3.2 调用F的函数体  
    3.3 销毁当前的执行上下文  
    3.4 返回F函数体的返回值，如果F的函数体没有返回值则返回undefined  
4. 如果[[Call]]的返回值是Object类型，则返回这个值，否则返回obj  
注意步骤2中， prototype指对象显示的prototype属性，而[[Prototype]]则代表对象内部Prototype属性(隐式的)。  
构成对象Prototype链的是内部隐式的[[Prototype]]，而并非对象显示的prototype属性。显示的prototype只有在函数对象上才有意义，从上面的创建过程可以看到，函数的prototype被赋给派生对象隐式[[Prototype]]属性，这样根据Prototype规则，派生对象和函数的prototype对象之间才存在属性、方法的继承/共享关系。  
  
用代码来做一些验证:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function fn(){}  
//the value of implicit [[Prototype]] property of those objects derived from fn will be assigned to fn.prototype  
fn.prototype={ attr1:"aaa", attr2:"bbb"};  
var obj=new fn();  
document.write(obj.attr1 + "<br />"); //result: aaa  
document.write(obj.attr2 + "<br />"); //result: bbb  
document.write(obj instanceof fn); //result: true  
document.write("<br />");  
//I change the prototype of fn here, so by the algorithm of Prototype the obj is no longer the instance of fn,  
//but this won't affect the obj and its [[Prototype]] property, and the obj still has attr1 and attr2 properties  
fn.prototype={};  
document.write(obj.attr1 + "<br />"); //result: aaa  
document.write(obj.attr2 + "<br />"); //result: bbb  
document.write(obj instanceof fn); //result: false

关于创建过程返回值的验证:

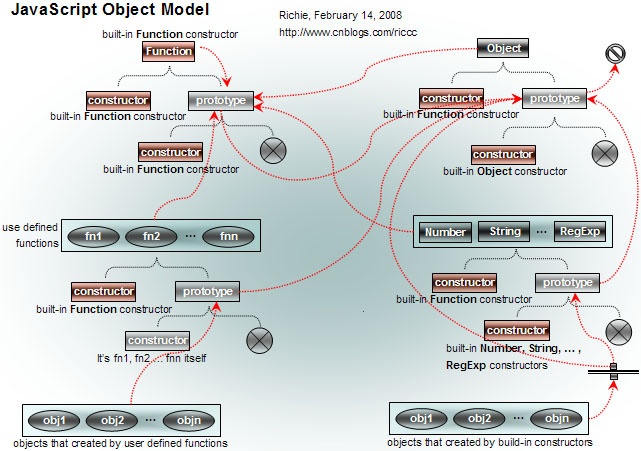
//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function fn(){  
    //according to step 4 described above,   
    //the new fn() operation will return the object { attr1: 111, attr2: 222 }, it's not an instance of fn!  
    return { attr1: 111, attr2: 222 };  
}  
fn.prototype={ attr1:"aaa", attr2:"bbb"};  
var obj=new fn();  
document.write(obj.attr1 + "<br />"); //result: 111  
document.write(obj.attr2 + "<br />"); //result: 222  
document.write(obj instanceof fn); //result: false

**做个练习**  
经过上面的理解应，请写出下面这幅图的实现代码。图中CF是一个函数，Cfp是CF的prototype对象，cf1, cf2, cf3, cf4, cf5都是CF的实例对象。虚线箭头表示隐式Prototype关系，实线箭头表示显示prototype关系。  
      
供参考的实现方案:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function CF(q1, q2){  
    this.q1=q1;  
    this.q2=q2;  
}  
CF.P1="P1 in CF";   
CF.P2="P2 in CF";  
function Cfp(){  
    this.CFP1="CFP1 in Cfp";  
}  
CF.prototype=new Cfp();  
var cf1=new CF("aaa", "bbb");  
document.write(cf1.CFP1 + "<br />"); //result: CFP1 in Cfp  
document.write(cf1.q1 + "<br />"); //result: aaa  
document.write(cf1.q2 + "<br />"); //result: bbb

**本地属性与继承属性**  
对象通过隐式Prototype链能够实现属性和方法的继承，但prototype也是一个普通对象，就是说它是一个普通的实例化的对象，而不是纯粹抽象的数据结构描述。所以就有了这个本地属性与继承属性的问题。  
首先看一下设置对象属性时的处理过程。JS定义了一组attribute，用来描述对象的属性property，以表明属性property是否可以在JavaScript代码中设值、被for in枚举等。  
obj.propName=value的赋值语句处理步骤如下:  
1. 如果propName的attribute设置为不能设值，则返回  
2. 如果obj.propName不存在，则为obj创建一个属性，名称为propName  
3. 将obj.propName的值设为value  
可以看到，设值过程并不会考虑Prototype链，道理很明显，obj的内部[[Prototype]]是一个实例化的对象，它不仅仅向obj共享属性，还可能向其它对象共享属性，修改它可能影响其它对象。  
用上面CF, Cfp的示例来说明，实例对象cf1具有本地属性q1, q2以及继承属性CFP1，如果执行cf1.CFP1=""，那么cf1就具有本地属性CFP1了，测试结果如下:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
var cf1=new CF("aaa", "bbb");  
var cf2=new CF(111, 222);  
document.write(cf1.CFP1 + "<br />"); //result: CFP1 in Cfp  
document.write(cf2.CFP1 + "<br />"); //result: CFP1 in Cfp  
//it will result in a local property in cf1  
cf1.CFP1="new value for cf1";  
//changes on CF.prototype.CFP1 will affect cf2 but not cf1, because there's already a local property with  
//the name CFP1 in cf1, but no such one in cf2  
CF.prototype.CFP1="new value for Cfp";  
document.write(cf1.CFP1 + "<br />"); //result: new value for cf1  
document.write(cf2.CFP1 + "<br />"); //result: new value for Cfp

**语义上的混乱?**  
还是使用上面CF, Cfp示例的场景。  
根据Prototype的机制，我们可以说对象cf1, cf2等都继承了对象Cfp的属性和方法，所以应该说他们之间存在继承关系。属性的继承/共享是沿着隐式Prototype链作用的，所以继承关系也应当理解为沿着这个链。  
我们再看instanceOf操作，只有cf1 instanceOf CF才成立，我们说cf1是CF的实例对象，CF充当了类的角色，而不会说cf1是Cfp的实例对象，这样我们应当说cf1继承自CF? 但CF充当的只是一个第三方工厂的角色，它跟cf1之间并没有属性继承这个关系。  
把CF, Cfp看作一个整体来理解也同样牵强。  
  
Prototype就是Prototype，没有必要强把JavaScript与面向对象概念结合起来， JavaScript只具备有限的面向对象能力，从另外的角度我们可以把它看成函数语言、动态语言，所以它是吸收了多种语言特性的精简版。  
  
**对象模型**  
**Where are we?**  
1. 了解了JavaScript的数据类型，清楚了象Number这样的系统内置对象具有多重身份: a)它们本身是一个函数对象，只是由引擎内部实现而已，b)它们代表一种数据类型，我们可以用它们定义、操作相应类型的数据，c)在它们背后隐藏了引擎的内部实现机制，例如内部的数据结构、各种被包装成了JavaScript对象的构造器等。  
2. 了解了Prototype机制，知道对象是如何通过它们继承属性和方法，知道了在创建对象过程中JS引擎内部是如何设置Prototype关系的。  
  
接下来对用户自定义函数对象本身的创建过程进行了解之后，我们就可以对JavaScript的对象模型来一个整体性的overview了。  
  
**函数对象创建过程**  
JavaScript代码中定义函数，或者调用Function创建函数时，最终都会以类似这样的形式调用Function函数:var newFun=Function(funArgs, funBody); 。创建函数对象的主要步骤如下:  
1. 创建一个build-in object对象fn  
2. 将fn的内部[[Prototype]]设为Function.prototype  
3. 设置内部的[[Call]]属性，它是内部实现的一个方法，处理逻辑参考对象创建过程的步骤3  
4. 设置内部的[[Construct]]属性，它是内部实现的一个方法，处理逻辑参考对象创建过程的步骤1,2,3,4  
5. 设置fn.length为funArgs.length，如果函数没有参数，则将fn.length设置为0  
6. 使用new Object()同样的逻辑创建一个Object对象fnProto  
7. 将fnProto.constructor设为fn  
8. 将fn.prototype设为fnProto  
9. 返回fn  
步骤1跟步骤6的区别为，步骤1只是创建内部用来实现Object对象的数据结构(build-in object structure)，并完成内部必要的初始化工作，但它的[[Prototype]]、[[Call]]、[[Construct]]等属性应当为null或者内部初始化值，即我们可以理解为不指向任何对象(对[[Prototype]]这样的属性而言)，或者不包含任何处理(对[[Call]]、[[Construct]]这样的方法而言)。步骤6则将按照前面描述的对象创建过程创建一个新的对象，它的[[Prototype]]等被设置了。  
从上面的处理步骤可以了解，任何时候我们定义一个函数，它的prototype是一个Object实例，这样默认情况下我们创建自定义函数的实例对象时，它们的Prototype链将指向Object.prototype。  
另外，Function一个特殊的地方，是它的[[Call]]和[[Construct]]处理逻辑一样。  
  
**JavaScript对象模型**  
      
红色虚线表示隐式Prototype链。  
 这张对象模型图中包含了太多东西，不少地方需要仔细体会，可以写些测试代码进行验证。彻底理解了这张图，对JavaScript语言的了解也就差不多了。下面是一些补充说明:  
1. 图中有好几个地方提到build-in Function constructor，这是同一个对象，可以测试验证:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
Function==Function.constructor //result: true  
Function==Function.prototype.constructor //result: true  
Function==Object.constructor //result: true  
//Function also equals to Number.constructor, String.constructor, Array.constructor, RegExp.constructor, etc.  
function fn(){}  
Function==fn.constructor //result: true

这说明了几个问题: Function指向系统内置的函数构造器(build-in Function constructor)；Function具有自举性；系统中所有函数都是由Function构造。  
  
2. 左下角的obj1, obj2...objn范指用类似这样的代码创建的对象: function fn1(){}; var obj1=new fn1();  
    这些对象没有本地constructor方法，但它们将从Prototype链上得到一个继承的constructor方法，即fn.prototype.constructor，从函数对象的构造过程可以知道，它就是fn本身了。  
    右下角的obj1, obj2...objn范指用类似这样的代码创建的对象: var obj1=new Object();或var obj1={};或var obj1=new Number(123);或obj1=/\w+/;等等。所以这些对象Prototype链的指向、从Prototype链继承而来的constructor的值(指它们的constructor是build-in Number constructor还是build-in Object constructor等)等依赖于具体的对象类型。另外注意的是，var obj=new Object(123);这样创建的对象，它的类型仍然是Number，即同样需要根据参数值的类型来确定。  
    同样它们也没有本地constructor，而是从Prototype链上获得继承的constructor方法，即build-in \*\*\* constructor，具体是哪一个由数据类型确定。  
  
3. 关于图中Prototype链的补充说明:  
Object.prototype是整个链的终结点，它的内部[[Prototype]]为null。  
所有函数的Prototype链都指向Function.prototype。  
Function的Prototype链指向Function.prototype，这是规范要求的，因为设计者将Function设计为具有自举性。Function的Prototype链这样设计之后，Function.constructor==Function, Function instanceOf Function都为true。另外Function已经是最顶层的构造器，但Function本身也是一个函数对象，它必然是由某个东西创建出来的，这样自举在语义上合情合理。  
Function.prototype的Prototype链指向Object.prototype，这也是规范强制要求的。首先Function.prototype是Function的一个实例对象(typeof Function.prototype可以知道它是一个Function，instanceOf无法通过测试，因为Prototype链在内部被额外设置了），所以按照Prototype的规则，Function.prototype的内部[[Prototype]]值应当为Function.prototype这个对象，即它的Prototype链指向自己本身。这样一方面在Prototype链上造成一个死循环，另一方面它本身成为了一个终结点，结果就是所有函数对象将不是派生自Object了。加上这个强制要求之后，Prototype链只有唯一的一个终结点。  
  
4. 因为Function.prototype是一个函数对象，所以它应当具有显示的prototype属性，即Function.prototype.prototype，但只有FireFox中可以访问到，IE、Opera、Safari都无法访问。所以图中用了个表示不存在的符号。  
  
5. 用户自定义函数(user defined functions)默认情况下[[Prototype]]值是Object.prototype，即它的隐式Prototype链指向Object.prototype，所以图中就这样表示了，但并不代表总是这样，当用户设置了自定义函数的prototype属性之后，情况就不同了。  
  
**执行模型**  
**执行上下文(Execution Context)简介**  
JavaScript代码运行的地方都存在执行上下文，它是一个概念，一种机制，用来完成JavaScript运行时作用域、生存期等方面的处理。执行上下文包括Variable Object、Variable Instatiation、Scope/Scope Chain等概念，在不同的场景/执行环境下，处理上存在一些差异，下面先对这些场景进行说明。  
  
函数对象分为用户自定义函数对象和系统内置函数对象，对于用户自定义函数对象将按照下面描述的机制进行处理，但内置函数对象与具体实现相关，ECMA规范对它们执行上下文的处理没有要求，即它们基本不适合本节描述的内容。  
  
执行的JavaScript代码分三种类型，后面会对这三种类型处理上不同的地方进行说明:  
1. Global Code，即全局的、不在任何函数里面的代码，例如一个js文件、嵌入在HTML页面中的js代码等。  
2. Eval Code，即使用eval()函数动态执行的JS代码。  
3. Function Code，即用户自定义函数中的函数体JS代码。  
  
*基本原理*  
在用户自定义函数中，可以传入参数、在函数中定义局部变量，函数体代码可以使用这些入参、局部变量。背后的机制是什么样呢？  
当JS执行流进入函数时，JavaScript引擎在内部创建一个对象，叫做Variable Object。对应函数的每一个参数，在Variable Object上添加一个属性，属性的名字、值与参数的名字、值相同。函数中每声明一个变量，也会在Variable Object上添加一个属性，名字就是变量名，因此为变量赋值就是给Variable Object对应的属性赋值。在函数中访问参数或者局部变量时，就是在variable Object上搜索相应的属性，返回其值。  
一般情况下Variable Object是一个内部对象，JS代码中无法直接访问。规范中对其实现方式也不做要求，因此它可能只是引擎内部的一种数据结构。  
  
大致处理方式就这样，但作用域的概念不只这么简单，例如函数体中可以使用全局变量、函数嵌套定义时情况更复杂点。这些情况下怎样处理？JavaScript引擎将不同执行位置上的Variable Object按照规则构建一个链表，在访问一个变量时，先在链表的第一个Variable Object上查找，如果没有找到则继续在第二个Variable Object上查找，直到搜索结束。这就是Scope/Scope Chain的大致概念。  
  
下面是各个方面详细的处理。  
  
**Global Object**  
JavaScript的运行环境都必须存在一个唯一的全局对象-Global Object，例如HTML中的window对象。Global Object是一个宿主对象，除了作为JavaScript运行时的全局容器应具备的职责外，ECMA规范对它没有额外要求。它包Math、String、Date、parseInt等JavaScript中内置的全局对象、函数(都作为Global Object的属性)，还可以包含其它宿主环境需要的一些属性。  
  
**Variable Object**  
上面简述了Variable Object的基本概念。创建Variable Object，将参数、局部变量设置为Variable Object属性的处理过程叫做Variable Instatiation-变量实例化，后面结合Scope Chain再进行详细说明。  
  
*Global Code*  
Variable Object就是Global Object，这是Variable Object唯一特殊的地方(指它是内部的无法访问的对象而言)。

var globalVariable = "WWW";  
document.write(window.globalVariable); //result: WWW

上面代码在Global Code方式下运行，根据对Variable Object的处理，定义变量globalVariable时就会在Global Object(即window)对象上添加这个属性，所以输出是WWW这个值。  
  
*Function Code*  
Variable Object也叫做Activation Object(因为有一些差异存在，所以规范中重新取一个名字以示区别，Global Code/Eval Code中叫Variable Object，Function Code中就叫做Activation Object)。  
每次进入函数执行都会创建一个新的Activation Object对象，然后创建一个arguments对象并设置为Activation Object的属性，再进行Variable Instantiation处理。  
在退出函数时，Activation Object会被丢弃(并不是内存释放，只是可以被垃圾回收了)。  
  
*附arguments对象的属性:*  
length: 为实际传入参数的个数。注意，参考函数对象创建过程，函数对象上的length为函数定义时要求的参数个数；  
callee: 为执行的函数对象本身。目的是使函数对象能够引用自己，例如需要递归调用的地方。  
function fnName(...) { ... }这样定义函数，它的递归调用可以在函数体内使用fnName完成。var fn=function(...) { ... }这样定义匿名函数，在函数体内无法使用名字引用自己，通过arguments.callee就可以引用自己而实现递归调用。  
参数列表: 调用者实际传入的参数列表。这个参数列表提供一个使用索引访问实际参数的方法。Variable Instantiation处理时会在Activation Object对象上添加属性，前提是函数声明时有指定参数列表。如果函数声明中不给出参数列表，或者实际调用参数个数与声明时的不一样，可以通过arguments访问各个参数。  
  
arguments中的参数列表与Activation Object上的参数属性引用的是相同的参数对象(如果修改，在两处都会反映出来)。规范并不要求arguments是一个数组对象，下面是一个测试:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
var argumentsLike = { 0: "aaa", 1: 222, 2: "WWW", length: 3, callee: function() { } };  
document.write(argumentsLike[2] + "<br />"); //result: WWW  
document.write(argumentsLike[1] + "<br />"); //result: 222  
//convert the argumentsLike to an Array object, just as we can do this for the arguments property  
var array = [].slice.apply(argumentsLike);  
document.write(array instanceof Array); //result: true  
document.write("<br />");  
document.write(array.reverse().join("|")); //result: WWW|222|aaa

*Eval Code*  
Variable Object就是调用eval时当前执行上下文中的Variable Object。在Global Code中调用eval函数，它的Variable Object就是Global Object；在函数中调用eval，它的Variable Object就是函数的Activation Object。

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function fn(arg){  
    var innerVar = "variable in function";  
    eval(' \  
        var evalVar = "variable in eval"; \  
        document.write(arg + "<br />"); \  
        document.write(innerVar + "<br />"); \  
    ');  
    document.write(evalVar);  
}  
fn("arguments for function");

输出结果是:  
arguments for function  
variable in function  
variable in eval  
说明: eval调用中可以访问函数fn的参数、局部变量；在eval中定义的局部变量在函数fn中也可以访问，因为它们的Varible Object是同一个对象。  
  
**Scope/Scope Chain**  
首先Scope Chain是一个类似链表/堆栈的结构，里面每个元素基本都是Variable Object/Activation Object。  
其次存在执行上下文的地方都有当前Scope Chain，可以理解为Scope Chain就是执行上下文的具体表现形式。  
  
*Global Code*  
Scope Chain只包含一个对象，即Global Object。在开始JavaScript代码的执行之前，引擎会创建好这个Scope Chain结构。  
  
*Function Code*  
函数对象在内部都有一个[[Scope]]属性，用来记录该函数所处位置的Scope Chain。  
创建函数对象时，引擎会将当前执行环境的Scope Chain传给Function的[[Construct]]方法。[[Construct]]会创建一个新的Scope Chain，内容与传入的Scope Chain完全一样，并赋给被创建函数的内部[[Scope]]属性。在前面函数对象创建过程一节中，这个处理位于步骤4和5之间。  
进入函数调用时，也会创建一个新的Scope Chain，包括同一个函数的递归调用，退出函数时这个Scope Chain被丢弃。新建的Scope Chain第一个对象是Activation Object，接下来的内容与内部[[Scope]]上存储的Scope Chain内容完全一样。  
  
*Eval Code*  
进入Eval Code执行时会创建一个新的Scope Chain，内容与当前执行上下文的Scope Chain完全一样。  
  
**实例说明**  
Scope Chain的原理就上面这些，必须结合JS代码的执行、Variable Instantiation的细节处理，才能理解上面这些如何产生作用，下面用一个简单的场景来综合说明。假设下面是一段JavaScript的Global Code:

var outerVar1="variable in global code";  
function fn1(arg1, arg2){  
    var innerVar1="variable in function code";  
    function fn2() { return outerVar1+" - "+innerVar1+" - "+" - "+(arg1 + arg2); }  
    return fn2();  
}  
var outerVar2=fn1(10, 20);

执行处理过程大致如下:  
1. 初始化Global Object即window对象，Variable Object为window对象本身。创建Scope Chain对象，假设为scope\_1，其中只包含window对象。  
2. 扫描JS源代码(读入源代码、可能有词法语法分析过程)，从结果中可以得到定义的变量名、函数对象。按照扫描顺序:   
   2.1 发现变量outerVar1，在window对象上添加outerVar1属性，值为undefined；  
   2.2 发现函数fn1的定义，使用这个定义创建函数对象，传给创建过程的Scope Chain为scope\_1。将结果添加到window的属性中，名字为fn1，值为返回的函数对象。注意fn1的内部[[Scope]]就是scope\_1。另外注意，创建过程并不会对函数体中的JS代码做特殊处理，可以理解为只是将函数体JS代码的扫描结果保存在函数对象的内部属性上，在函数执行时再做进一步处理。这对理解Function Code，尤其是嵌套函数定义中的Variable Instantiation很关键；  
   2.3 发现变量outerVar2，在window对象上添加outerVar2属性，值为undefined；  
3. 执行outerVar1赋值语句，赋值为"variable in global code"。  
4. 执行函数fn1，得到返回值:  
   4.1 创建Activation Object，假设为activation\_1；创建一个新的Scope Chain，假设为scope\_2，scope\_2中第一个对象为activation\_1，第二个对象为window对象(取自fn1的[[Scope]]，即scope\_1中的内容)；  
   4.2 处理参数列表。在activation\_1上设置属性arg1、arg2，值分别为10、20。创建arguments对象并进行设置，将arguments设置为activation\_1的属性；  
   4.3 对fn1的函数体执行类似步骤2的处理过程:  
       4.3.1 发现变量innerVar1，在activation\_1对象上添加innerVar1属性，值为undefine；  
       4.3.2 发现函数fn2的定义，使用这个定义创建函数对象，传给创建过程的Scope Chain为scope\_2(函数fn1的Scope Chain为当前执行上下文的内容)。将结果添加到activation\_1的属性中，名字为fn2，值为返回的函数对象。注意fn2的内部[[Scope]]就是scope\_2；  
   4.4 执行innerVar1赋值语句，赋值为"variable in function code"。  
   4.5 执行fn2:  
       4.5.1 创建Activation Object，假设为activation\_2；创建一个新的Scope Chain，假设为scope\_3，scope\_3中第一个对象为activation\_2，接下来的对象依次为activation\_1、window对象(取自fn2的[[Scope]]，即scope\_2)；  
       4.5.2 处理参数列表。因为fn2没有参数，所以只用创建arguments对象并设置为activation\_2的属性。  
       4.5.3 对fn2的函数体执行类似步骤2的处理过程，没有发现变量定义和函数声明。  
       4.5.4 执行函数体。对任何一个变量引用，从scope\_3上进行搜索，这个示例中，outerVar1将在window上找到；innerVar1、arg1、arg2将在activation\_1上找到。  
       4.5.5 丢弃scope\_3、activation\_2(指它们可以被垃圾回收了)。  
       4.5.6 返回fn2的返回值。  
   4.6 丢弃activation\_1、scope\_2。  
   4.7 返回结果。  
5. 将结果赋值给outerVar2。  
  
其它情况下Scope Chain、Variable Instantiation处理类似上面的过程进行分析就行了。  
  
根据上面的实例说明，就可以解释下面这个测试代码的结果:

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function fn(obj){  
    return {  
        //test whether exists a local variable "outerVar" on obj  
        exists: Object.prototype.hasOwnProperty.call(obj, "outerVar"),  
        //test the value of the variable "outerVar"  
        value: obj.outerVar  
    };  
}  
var result1 = fn(window);  
var outerVar = "WWW";   
var result2 = fn(window);  
  
document.write(result1.exists + " " + result1.value); //result: true undefined  
document.write("<br />");  
document.write(result2.exists + " " + result2.value); //result: true WWW

result1调用的地方，outerVar声明和赋值的语句还没有被执行，但是测试结果window对象已经拥有一个本地属性outerVar，其值为undefined。result2的地方outerVar已经赋值，所以window.outerVar的值已经有了。实际使用中不要出现这种先使用，后定义的情况，否则某些情况下会有问题，因为会涉及到一些规范中没有提及，不同厂商实现方式上不一致的地方。  
  
**一些特殊处理**  
1. with(obj) { ... }这个语法的实现方式，是在当前的Scope Chain最前面位置插入obj这个对象，这样就会先在obj上搜索是否有相应名字的属性存在。其它类似的还有catch语句。  
2. 前面对arguments对象的详细说明中，提到了对函数递归调用的支持问题，了解到了匿名函数使用arguments.callee来实现引用自己，而命名函数可以在函数体内引用自己，根据上面Scope Chain的工作原理我们还无法解释这个现象，因为这里有个特殊处理。  
任何时候创建一个命名函数对象时，JavaScript引擎会在当前执行上下文Scope Chain的最前面插入一个对象，这个对象使用new Object()方式创建，并将这个Scope Chain传给Function的构造函数[[Construct]]，最终创建出来的函数对象内部[[Scope]]上将包含这个object对象。创建过程返回之后，JavaScript引擎在object上添加一个属性，名字为函数名，值为返回的函数对象，然后从当前执行上下文的Scope Chain中移除它。这样函数对象的Scope Chain中第一个对象就是对自己的引用，而移除操作则确保了对函数对象创建处Scope Chain的恢复。  
  
**this关键字处理**  
执行上下文包含的另一个概念是this关键字。  
Global Code中this关键字为Global Object；函数调用时this关键字为调用者，例如obj1.fn1()，在fn1中this对象为obj1；Eval Code中this关键字为当前执行上下文的Variable Object。  
  
在函数调用时，JavaScript提供一个让用户自己指定this关键字值的机会，即每个函数都有的call、apply方法。例如:  
fn1.call(obj1, arg1, arg2, ...)或者fn1.apply(obj1, argArray)，都是将obj1作为this关键字，调用执行fn1函数，后面的参数都作为函数fn1的参数。如果obj1为null或undefined，则Global Object将作为this关键字的值；如果obj1不是Object类型，则转化为Object类型。它们之间的唯一区别在于，apply允许以数组的方式提供各个参数，而call方法必须一个一个参数的给。  
前面的测试示例代码中有多处运用到了这个方法。例如window对象并没有hasOwnProperty方法，使用Object.prototype.hasOwnProperty.call(window, "propertyName")也可以测试它是否拥有某个本地属性。  
  
**JavaScript中的闭包Closures**  
示例：

//Passed in FF2.0, IE7, Opera9.25, Safari3.0.4  
function outer(){  
    var a="aaa";  
    var b="bbb";  
    return function(){ return a + " " + b; };  
}  
var inner=outer();  
document.write(inner());

outer返回的是一个内嵌函数，内嵌函数使用了outer的局部变量a和b。照理outer的局部变量在返回时就超出了作用域因此inner()调用无法使用才对。这就是闭包Closure，即函数调用返回了一个内嵌函数，而内嵌函数引用了外部函数的局部变量、参数等这些应当被关闭(Close)了的资源。  
  
根据前面Scope Chain的理解可以解释，返回的内嵌函数已经持有了构造它时的Scope Chain，虽然outer返回导致这些对象超出了作用域、生存期范围，但JavaScript使用自动垃圾回收来释放对象内存: 按照规则定期检查，对象没有任何引用才被释放。因此上面的代码能够正确运行。  
  
关于使用Closure时的内存泄漏、效率等问题，参考http://www.jibbering.com/faq/faq\_notes/closures.html