**原始值和引用值**

在ECMAScript中，变量可以存放两种类型的值，即原始值和引用值。

原始值指的就是代表原始数据类型（基本数据类型）的值，即Undefined,Null,Number,String,Boolean类型所表示的值。

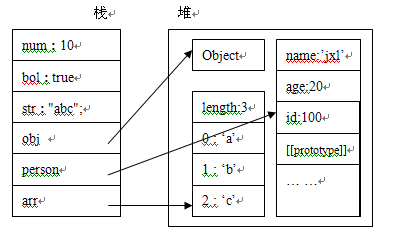
引用值指的就是复合数据类型的值，即Object,Function,Array,以及自定义对象,等等

**栈和堆**

与原始值与引用值对应存在两种结构的内存即栈和堆

原始值是存储在栈中的简单数据段，也就是说，他们的值直接存储在变量访问的位置。

堆是存放数据的基于散列算法的数据结构，在JavaScript中，引用值是存放在堆中的。



变量num,bol,str为基本数据类型，它们的值，直接存放在栈中，obj,person,arr为复合数据类型，他们的引用变量存储在栈中，指向于存储在堆中的实际对象。我们无法直接操纵堆中的数据，也就是说我们无法直接操纵对象，但我们可以通过栈中对对象的引用来操作对象。

堆比栈大，栈比堆的运算速度快,对象是一个复杂的结构，并且可以自由扩展，如：数组可以无限扩充，对象可以自由添加属性。将他们放在堆中是为了不影响栈的效率。而是通过引用的方式查找到堆中的实际对象再进行操作。相对于简单数据类型而言，简单数据类型就比较稳定，并且它只占据很小的内存。不将简单数据类型放在堆是因为通过引用到堆中查找实际对象是要花费时间的，而这个综合成本远大于直接从栈中取得实际值的成本。所以简单数据类型的值直接存放在栈中。

Null**和**Undefined**的比较**

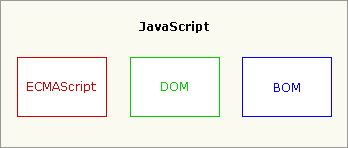
在 ECMAScript 的原始类型中，是有Undefined 和 Null 类型的。 这两种类型都分别对应了属于自己的唯一专用值，即undefined 和 null。值 undefined 实际上是从值 null 派生来的，因此 ECMAScript 把它们定义为相等的。

尽管这两个值相等，但它们的含义不同。undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象。Udefined代表没有赋值的基本数据类型，Null代表没有赋值的引用数据类型。

null 参与数值运算时其值会自动转换为 0 ，undefined参与任何数值计算时，其结果一定是NaN。

当声明的变量未初始化时，该变量的默认值是undefined，但是undefined并不同于未定义的值。Typeof运算符无法区分这两种值，因此对于变量是否存在的判断操作是通过if(typeof var == ‘undefined’){ //code here } 来进行判断的,这样既完全兼容**未定义(undefined)**和**未初始化(uninitialized)**两种情况的。

JavaScript**全局观**



JavaScript核心包括一下三部分：

1. 核心（ECMAScript)：定义了脚本语言的所有对象，属性和方法。
2. 文档对象模型（DOM)：HTML和XML应用程序接口。
3. 浏览器对象模型(BOM)：对浏览器窗口进行访问操作。

**关于**ECMAScript

1. ECMAScript是一种由[欧洲计算机制造商协会](http://baike.baidu.com/view/2233504.htm)（[ECMA](http://baike.baidu.com/view/786648.htm)）通过ECMA-262标准化的脚本程序设计语言。
2. ECMAScript的工作是定义语法和对象，从最基本的语法、数据类型、条件语句、关键字、保留字到异常处理和对象定义都是它的范畴。
3. JavaScript实现了ECMAScript，Adobe ActionScript和OpenView ScriptEase同样也实现了ECMAScript
4. 在ECMAScript范畴内定义的对象也叫做原生对象，如Object、Array、Function等等。
5. 由ECMA-262定义的ECMAScript与Web浏览器没有依赖关系。其实上它就是一套定义了语法规则的接口，然后由不同的浏览器对其进行实现,最后我们输写遵守语法规则的程序，完成应用开发需求。

**关于**DOM

文档对象模型（Document Object Model）定义了访问和处理文档的标准方法。根据DOM的定义(HTML和XML应用程序接口)可知DOM由两个部分组成:

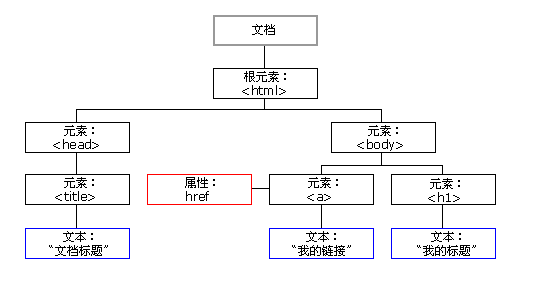
1. 针对XML的DOM即DOM Core
2. 针对HTML的DOM HTML。

DOM Core的核心概念就是节点(Node)。DOM会将文档中不同类型的元素(这里的元素并不特指<div>这种tag,还包括属性，注释，文本之类）都看作为不同的节点。DOM CORE在解析文档时，会将所有的元素、属性、文本、注释等等视为一个节点对象（或继承自节点对象的对象,多态、向上转型）,根据文本结构依次展现，最后行成了一棵"DOM树"

DOM HTML的核心概念是HTMLElement，DOM HTML会将文档中的元素（这里的元素特指<body>这种tag,不包括注释，属性，文本)都视为HTMLElement。而元素的属性，则为HTMLElement的属性。

其实上DOM Core和DOM html的外部调用接口相差并不是很大，对于html文档可以用DOM html进行操作，针对Xhtml可以用DOM Core。

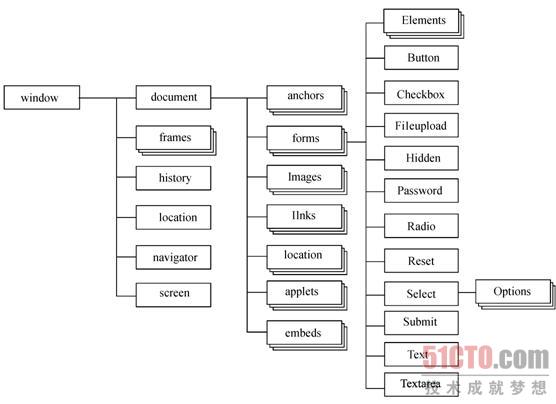
DOM模型示例：

****

**关于**BOM

BOM解析：  
1. BOM是browser object model的缩写，简称浏览器对象模型  
2. BOM提供了独立于内容而与浏览器窗口进行交互的对象  
3. 由于BOM主要用于管理窗口与窗口之间的通讯，因此其核心对象是window  
4. BOM由一系列相关的对象构成，并且每个对象都提供了很多方法与属性

BOM模型示例：



**基本的数据类型**

"基本的数据类型"与"基本数据类型"的概念不一样：

1. "基本的数据类型"指的是最常用的数据类型
2. "基本数据类型"指的是原始类型（储存在内存中的方式）
3. **原始类型（简单数据类型、基本数据类型）**

Undefined类型： 表示声明了变量但未对其初始化时赋予该变量的值。undefined为Undefined类型下的唯一的一个值。

Null类型：用于表示尚未存在的对象。Null类型下也只有一个专用值null。

Boolean类型：有两个值true和false,主要用于条件判断，控制执行流程。

Number类型：代表数字（即包括32的整数，也包括64位的浮点数）

String类型：用于代表字符串。

1. **对象**

一个无序属性的集合，这些属性的值为简单数据类型、对象或者函数。这里对象并不特指全局对象Object.

1. **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。

这里的函数并不特指全局对象Function。

**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。

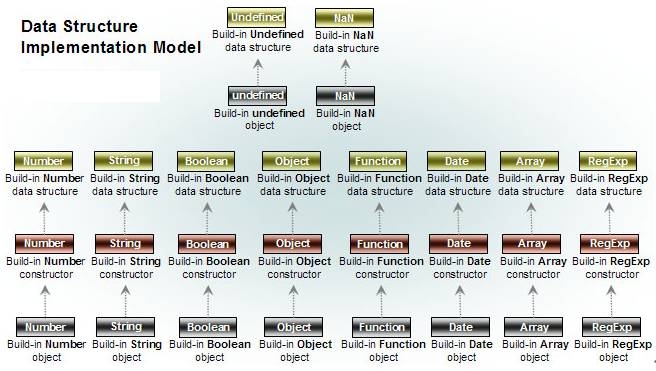
Boolean, Number, String: 分别为这三种简单数值类型的对象包装器，对象包装在概念上有点类似C#/Java中的Box/Unbox。

Date, Array, RegExp: 可以把它们看作是几种内置的扩展数据类型。

注：

1. 它们都是JavaScript语言的内置对象，都可以看作是函数的派生类型，在这个意义上，可将它们跟用户自定义的函数等同看待。
2. 它们各自可以代表一种数据类型，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。

**数据类型实现模型描述**



标准注解：

Build-in \*\*\* data structure: 指JS内部用于实现\*\*\*类型的数据结构，由宿主环境(浏览器)提供，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

可理解：

data structure:存储在内存中的数据

object：对于存储在内存中的数据的一种包装（也存放在内存中），提供各种接口以供程序语言对存储在内存中的数据进行操作。

constructor:将存储在内存中的数据包装的方法。

**关于简单数据类型的对象化**

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。

示例代码：

var num1 = new Number(123);

num1 instanceof Number//result: true

num1 instanceof Object//result: true

num1 = Number(num1);

num1 instanceof Number //result: false

num1 instanceof Object //result: false

var num2 = 123;

num2 instanceof Number //result: false

num2 instanceof Object //result: false

结论：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。由此也就产生了一个概念"Literal Syntax"

Literal Syntax**-字面量定义方法**

可以理解为在定义一个变量的同时初始化赋值。

1、简单数据类型：

数值：var i = 100; //替代var i = new Number(100);

布尔值：var b = true;  //替代var b = new Boolean(true);

字符：var str = 'this is a string.'; //替代var str = new String('this is a string');

2、复合数据类型

数组：var arr = ['笨蛋的座右铭',25]

对象：var obj  = {name:'笨蛋的座右铭',age:25}

正则表达式：var reg = /\d+/;

Json: var obj ={name:’测试’，

age:25，

Type:[‘测试’，’25’，[‘c1’,’c2’,’c3’,’c4’]，function(){//code}，{name:'笨蛋的座右铭',age:25}]，

getname: function (){//code }

}

**原型继承原理**

什么是prototype？

JavaScript中对象的prototype属性，可以返回对象类型原型的引用。在面向对象领域里，实例与类型不是唯一的一对可描述的抽象关系，在JavaScript中，另外一种重要的抽象关系是类型(Type)与原型(prototype)。这种关系是一种更高层次的抽象关系，它恰好和类型与实例的抽象关系构成了一个三层的链。

实例、类型、原型实例：

// Animal对象类型

function Animal (name){

this.name=name;

}

// Animal对象原型对象

Animal.prototype = {

id:"Animal",

sleep:function(){

alert("sleep");

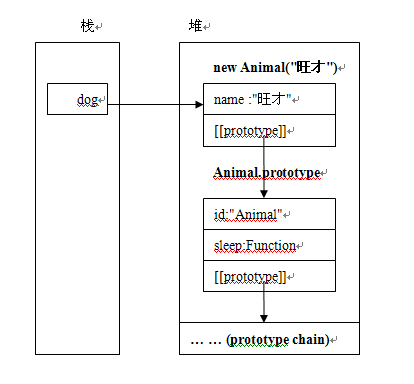
}

}

// Animal实例

Var dog=new Animal();

其对应的简易内存分配结构图：



prototype**与**[[prototype]]

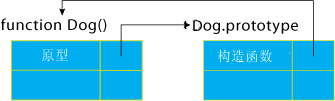
[[prototype]]与prototype并不是同一个东西:

1. 每个函数对象都有一个显示的prototype属性,它代表了对象的原型，更明确的说是代表了由函数对象(构造函数)所创建出来对象原型。
2. 每个对象都有一个名为[[Prototype]]的内部属性，指向于它所对应的原型对象。对象的原型对象必然也有[[prototype]]属性指向于它所对应的原型对象，由此便构成了一种链表的结构，这就是原型链的概念。
3. 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。
4. prototype链的根为Object.prototype,对象Object.prototype的内部[[prototype]]属性为null.

constructor**实现原理**

constructor指的就是对象的构造函数。

在 JavaScript 中，每个函数都有名为“prototype”的属性，用于引用原型对象。此原型对象又有名为“constructor”的属性，它反过来引用函数本身。这是一种循环引用。



constructor是Function在创建函数对象时产生的，它是函数对象prototype链中的一个属性。

constructor的原理就是在对象的原型链上寻找constructor属性。

示例代码：

function Animal(){}

function Person(){}

Person.prototype = new Animal();

var person = new Person();

alert(person.constructor); //Animal

由于将Person.prototype指向了new Animal。此时，Person的prototype指向的是Animal的实例，所以person的constructor为Animal这个构造函数。

instanceof**原理**

instanceof可以判断一个对象是不是某个类或其子类的实例。

instanceof 检测一个对象A是不是另一个对象B的实例的原理是：查看对象B的prototype指向的对象是否在对象A的[[prototype]]链上。如果在，则返回true,如果不在则返回false。

不过有一个特殊的情况，当对象B的prototype为null将会报错(类似于空指针异常)。

**函数对象的创建过程**

函数就是对象,代表函数的对象就是函数对象。函数对象是由Function这个函数对象构造出来的。Function对象本身也是一个函数，因此它也一个函数对象。

函数对象详细创建步骤如下：

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,处理函数调用的逻辑。

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。

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)。

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。

**函数对象构造过程的分析**

function Animal () {};和var dog = new Animal ();

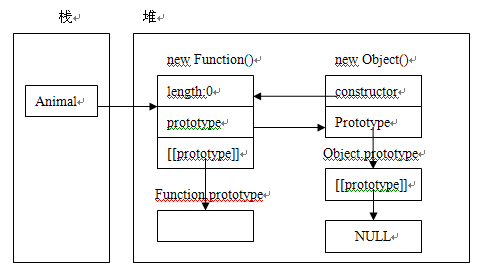
上述两行简单的语句的实际构造过程可以等价于以下的代码：

function Animal() {};等价于：

{

Animal.prototype = {constructor: Animal };

}



var dog =new Animal ()等价于：

var dog =({

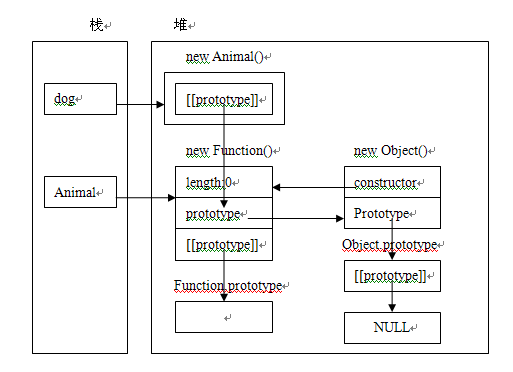
var o ={};

o.{[[prototype]]= Animal. Prototype;

Animal.call(o);

Return o;

})();



Function**与**Object关系

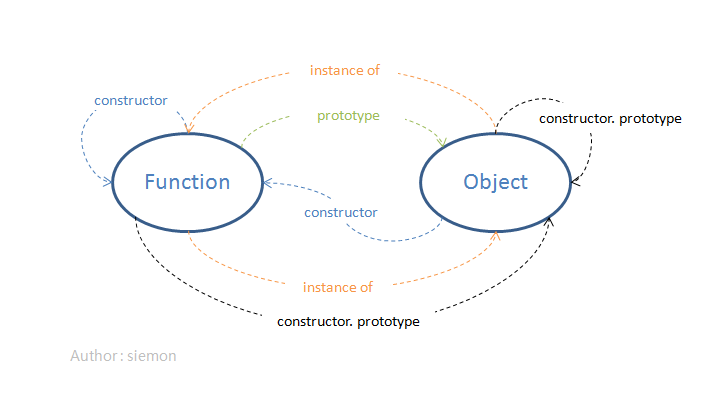
Function与Object可以总结为下图：

蓝色线为类的constructor的实例

 橙色线为类的实例

 绿色线为类的prototype的实例

黑色线为类的constructor的prototype的实例



**由上图可以得出下列结论：**

1. Function和Object各为自身的实例。
2. Object是通过Function进行构造的，而Function则由自己构造自己。
3. Function的原型对象是Object的实例。

**关于**Fuction**与**Object**的总结**：

Function

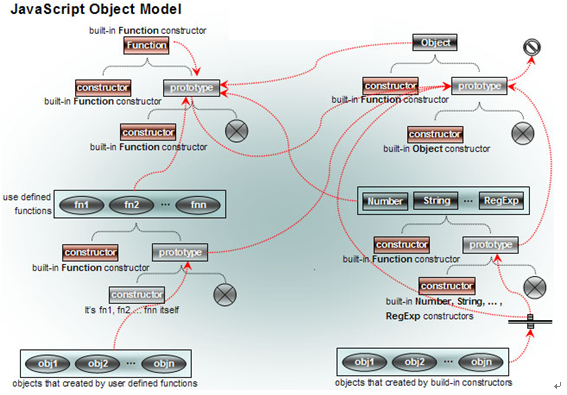
函数就是对象,代表函数的对象就是函数对象。所有的函数对象是被Function这个函数对象构造出来的。也就是说，Function是最顶层的构造器。它构造了系统中所有的对象，包括用户自定义对象，系统内置对象，甚至包括它自已。这也表明Function具有自举性(自已构造自己的能力)。这也间接决定了Function的[[call]]和[[constructor]]逻辑相同。

Object

对于Object它是最顶层的对象，所有的对象都将继承Object的原型，Object也是一个函数对象，Object是被Function构造出来的。

**对象模型**

JavaScript的对象模型如下图所示：(红色虚线表示隐式Prototype链)



总结:

1. 图中有好几个地方提到build-in Function constructor，这是同一个对象，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。
3. 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，具体是哪个由数据类型确定。
4. Object.prototype是整个链的终结点，它的内部[[Prototype]]为null。
5. 所有函数的Prototype链都指向Function.prototype。这是规范要求的，因为设计者将Function设计为具有自举性。
6. Function.prototype的Prototype链指向Object.prototype, 这也是规范强制要求的。保证Prototype链只有唯一的一个终结点。
7. 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。
8. 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。

**属性访问原则**

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

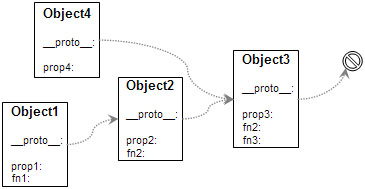
2. 如果obj.\_\_proto\_\_为null，返回undefined，否则

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。

这就是基于Prototype的继承和共享：（object1的方法fn2来自object2，概念上即object2重写了object3的方法fn2。）



**本地属性与继承属性**

看一下设置对象属性时的处理过程，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，对象的属性无法修改其原型中的同名属性，而只会自身创建一个同名属性并为其赋值。

道理很明显，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。

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。

**执行模型**

Javascript执行模型指的是一段javascript脚本从载入浏览器到显示执行都经过了哪些流程。

Javascript执行模型可简要总结如下：

1. step1. 读入第一个代码段
2. step2. 做语法分析，有错则报语法错误（比如括号不匹配等），并跳转到step5
3. step3. 创建全局执行环境(对var变量和function定义做"预解析")
4. step4. 执行代码段(调用函数、进入eval时，都会创建新的执行环境)，有错则报错（比如变量未定义）
5. step5. 如果还有下一个代码段，则读入下一个代码段，重复step2
6. step6. 结束

**执行环境**(Execution Context)

所有 JavaScript 代码都是在一个执行环境中被执行的。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。

可执行的JavaScript代码分三种类型：

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。

不同类型的JavaScript代码具有不同的Execution Context。

在一个页面中，第一次载入JS代码时创建一个全局执行环境，当调用一个 JavaScript 函数时，该函数就会进入相应的执行环境。如果又调用了另外一个函数（或者递归地调用同一个函数），则又会创建一个新的执行环境，并且在函数调用期间执行过程都处于该环境中。当调用的函数返回后，执行过程会返回原始执行环境。因而，运行中的 JavaScript 代码就构成了一个执行环境栈。

程序在进入每个执行环境的时候都会创建一个叫做Variable Object的对象。针对于函数执行环境，函数对应的每一个参数、局部变量、内部方法都会在Variable Object上创建一个属性，属性名为变量名，属性值为变量值。针对于全局执行环境，具有相同的行为。但是要强调的一点是在全局执行环境中Varible Object就是Global Object，可以简单的理解为window对象。

**全局执行环境**

在一个页面中，第一次载入JS代码时创建一个全局执行环境，全局执行环境的作用域链实际上只由一个对象,即全局对象(window),在开始JavaScript代码的执行之前，引擎会创建好这个Scope Chain结构。全局执行环境也会有变量实例化的过程，它的内部函数就是涉及大部分 JavaScript 代码的、常规的顶级函数声明。而且，在变量实例化过程中全局对象就是可变对象，这就是为什么全局性声明的函数是全局对象属性的原因。全局性声明的变量同样如此全局执行环境会使用 this 对象来引用全局对象。

Eval**执行环境**

构建Eval执行环境时的可变对象(Variable Object)就是调用eval时当前执行上下文中的可变对象(Variable Object)。在全局执行环境中调用eval函数，它的可变对象(Variable Object)就是全局对象；在函数中调用eval，它的可变对象(Variable Object)就是函数的活动对象(Activation Object)。

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。

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。

**函数执行环境**

在创建执行环境的过程中，会按照定义的先后顺序完成一系列操作:

1. 首先会创建一个'活动对象'(Activation Object)。活动对象是规范中规定的另外一种机制。之所以称之为对象，是因为它拥有可访问的命名属性，但是它又不像正常对象那样具有原型（至少没有预定义的原型），而且不能通过 JavaScript 代码直接引用活动对象。
2. 为函数调用创建执行环境的下一步是创建一个 arguments 对象，这是一个类似数组的对象，它以整数索引的数组成员一一对应地保存着调用函数时所传递的参数。这个对象也有 length 和 callee 属性。然后，会为活动对象创建一个名为“arguments”的属性，该属性引用前面创建的 arguments对象。
3. 接着，为执行环境分配作用域。作用域由对象列表（链）组成。
4. 之后会发生由 ECMA 262 中所谓'活动对象'完成的'变量实例化'(Variable Instatiation)的过程。此时会将函数的形式参数创建为可变对象的命名属性，如果调用函数时传递的参数与形式参数一致，则将相应参数的值赋给这些命名属性（否则，会给命名属性赋 undefined 值）。对于定义的内部函数，会以其声明时所用名称为可变对象创建同名属性，而相应的内部函数则被创建为函数对象并指定给该属性。变量实例化的最后一步是将在函数内部声明的所有局部变量创建为可变对象的命名属性。注：在这个过程中，除了实际参数有值外和函数定义外，其它都被'预解析'为undefined值.
5. 最后，要为使用 this 关键字而赋值。(此时的this指向的是全局对象，即window)

**关于作用域和作用域链**

 在访问变量时，就必须存在一个可见性的问题，这就是作用域（Scope）。更深入的说，当访问一个变量或调用一个函数时，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。这也就形成了作用域链（Scope Chain）的概念。

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。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)。

**执行环境栈和作用域链的关系示例：**

<script type="text/javascript">

function Fn1(){

function Fn2(){

alert(document.body.tagName);//BODY

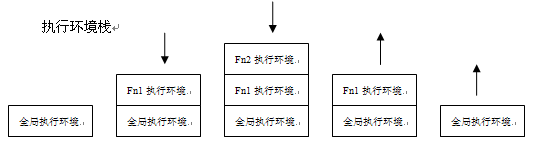
 //other code...

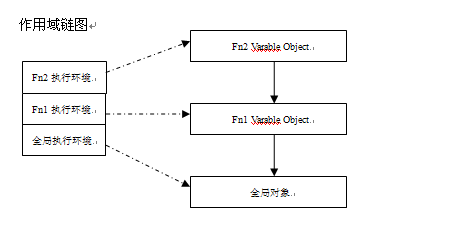
}

Fn2();

}

</script>





this**关键字处理**

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。

**作用域分配与变量访问规则**

在 ECMAScript 中，函数也是对象。函数对象在变量实例化过程中会根据函数声明来创建，或者是在计算函数表达式或调用 Function 构造函数时创建。。每个函数对象都有一个内部的 [[scope]] 属性，这个属性也由对象列表（链）组成。这个内部的[[scope]] 属性引用的就是创建它们的执行环境的作用域链，同时，当前执行环境的活动对象被添加到该对象列表的顶部。当我们在函数内部访问变量时，其实就是在作用域链上寻找变量的过程。

示例代码：

<script type="text/javascript">

function outer(){

var i = 10;

function inner(){

var j = 100;

alert(j);//100

alert(i);//10

alert(adf);

}

inner();

}

outer();

</script>

执行过程如下：

1.载入代码，创建全局执行环境，在可变对象(window)中添加outer变量，其指向于函数对象outer,此时作用域链中只有window对象.

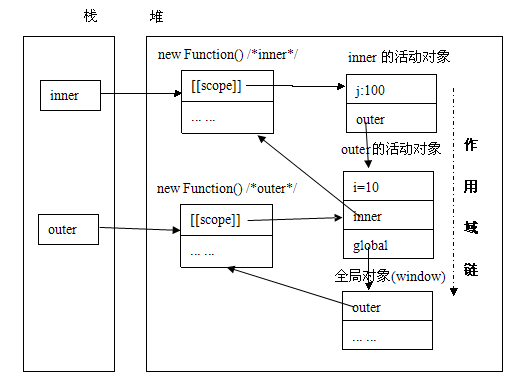
2.执行代码，当程序执行到outer()时，会在全局对象中寻找outer变量，成功调用。

3.创建outer的执行环境，此时会新创建一个活动对象，添加变量i，设置值为10,添加变量inner，指向于函数对象inner.并将活动对象压入作用域链中.并将函数对象outer的[[scope]]属性指向活动对象outer。此时作用域链为outer的活动对象+window.

4.执行代码，为 i 成功赋值。当程序执行到inner()时，会在函数对象outer的[[scope]]中寻找inner变量。找到后成功调用。

5.创建inner的执行环境，新建一个活动对象，添加变量j，赋值为100,并将该活动对象压入作用域链中，并函数对象inner的[[scope]]属性指向活动对象inner.此时作用域链为:inner的活动对象+outer的活动对象+ window.

6.执行代码为j赋值，当访问i、j时成功在作用域中找到对应的值并输出，而当访问变量adf时，没有在作用域中寻找到，访问出错。



**闭包原理**

ECMAScript 允许使用内部函数－－即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。也就是说，内部函数会在外部函数返回后被执行。而当这个内部函数执行时，它仍然必需访问其外部函数的局部变量、参数以及其他内部函数。这些局部变量、参数和函数声明（最初时）的值是外部函数返回时的值，但也会受到内部函数的影响。

示例代码：

<script type="text/javascript">

var increment = (function(){

var id = 0;

return function(){

return ++id;

}

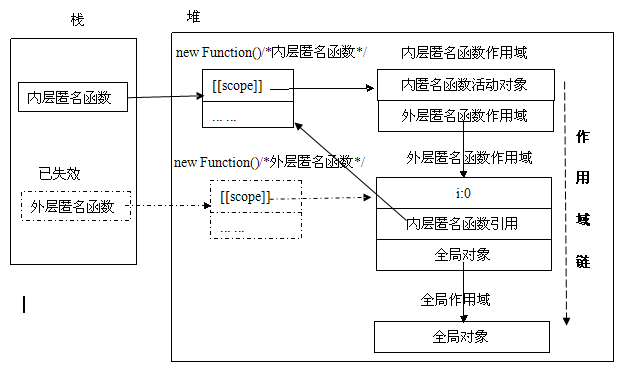
})()

alert(increment());//1

alert(increment());//2

</script>

上述代码的解释：在执行第二个alert(increment())的语句时，按理说变量id在执行完第一个alert(increment())语句后就应该销毁了，但是由于在此函数外城函数的返回值是一个内层函数，而JavaScript使用自动垃圾回收来释放对象内存，所以此时内层函数并没有销魂，同时变量id得以保留，因此在第二次执行alert(increment())时id的初始值是1而非0。



[**函数形式参数与**arguments](http://www.cnblogs.com/fool/archive/2010/10/19/1855261.html)

实例代码：

function say(msg,other,garbage){

alert(arguments[1]);//world

var other = 'nice to meet you!';

var msg;

alert(arguments.length);

alert(msg);//hello

alert(other);//nice to meet you!

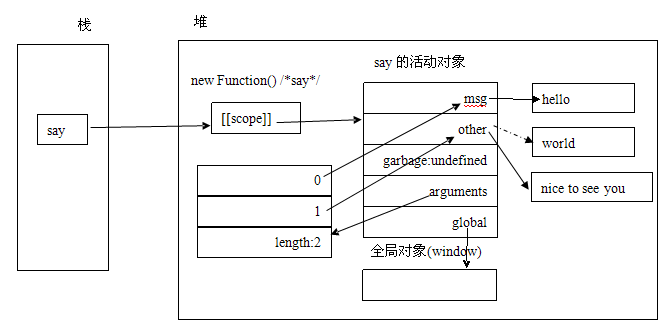
alert(arguments[1]);//nice to meet you!

alert(garbage);//undefined

}

say('hello','world');

**简单的内存图**



注：虚线表示的是曾经引用的指向。

Javascript函数有形式参数和实际参数。

形式参数是定义方法时所明确指定的参数，由于Javascript语言的灵活性，javascript不要求方法调用时，所传递参数个数与形式参数一致.。

javascript实际调用时所传递的参数就是实际参数。arguments指的就是实际参数。

从say方法中可以看出，say定义了三个形式参数，而实际调用时只传递了两个值。因此arguments.length的值为2,而不是3.接着我们来看一下arguments的特殊行为，个人感觉arguments会将所有的实际参数都当作对象来看待，对于基本数据类型的实际参数则会转换为其对应的对象类型。这是根据在函数内定义与形式参数同名的变量并赋值，arguments对应的值会跟着改变来判断的。

由于在实际调用的过程中实际参数other被重新定义了，所以在alert(other)语句执行时返回的结果是nice to meet you!而不是调用say时传递的形式参数world,而两次执行alert(arguments[1])的结果不一样是因为第一次arguments[1]指向other，而other指向world，第二次arguments[1]同样指向other，但是此时other已经由原来指向world变为指向了nice to meet you!

**总结执行模型完整实例分析：**

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。