JavaScript高级程序与设计

第三章 基本概念

本章内容 ： 语法 数据类型 流控制语句 函数

3.1语法 ECMAScript借鉴了C及其其他类C语言。

3.11 区分大小写 ECMAScript 的一切（变量、函数名和操作符）都区分大小写。

3.12 标识符 就是变量、函数、属性的名字或者函数的参数。

3.2 关键字和保留字

这些关键字可用于表示控制语句的开始或结束。或用于执行特定操作。按照规则，关键字也是语言保留的，不能用于标识符。

流程控制 ：break case catch if else 等等

保留字： 没有特定用途，但是可能在将来用作关键字。

3.3 变量 ECMAScript变量为松散类型。可以保存任何类型的数据。定义变量： var 变量名

第四章 变量、作用域和内存问题

本章内容： 理解基本类型和引用类型的值 理解执行环境 理解垃圾收集

4.1 基本类型和引用类型的值

ECMA script变量可能包含两种不同数据类型的值：基本类型和引用类型值。基本类型值指的是简单的数据段，而引用类型值指那些可能由多个值构成的对象。

5种基本数据类型： Undefined、Null、Boolean、Number和 String。这五种基本数据类型是按值访问的，因为可以操作变量中实际的值。

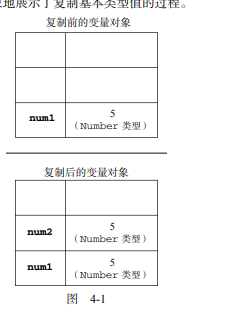
引用类型的值是保存在内存中的对象。与其他语言不同，JavaScript不允许直接访问内存中的位置，也就是说不能直接操作对象的内存空间。在操作对象时，实际上在操作对象的引用而不是实际的对象，为此，引用类型的值是按引用访问的。

4.1.2 复制变量值

从一个变量复制基本类型值和引用类型值也存在不同。

如果从一个变量向另一个变量复制基本类型的值，会在变量对象上创建一个新值，然后把该值复制到为新变量分配的位置上。例子：

Var num1=5; var num2=num1



从一个变量向另一个变量复制引用类型值时，同样也会将存储在变量对象中的值复制一份到为新变量分配的空间中，不同的是，这个值的副本实际上是一个指针，而这个指针指向存储在堆中的一个对象，当复制结束后，这两个变量实际上引用同一个对象，因此，改变其中一个变量，就会影响到另一个变量。例子：

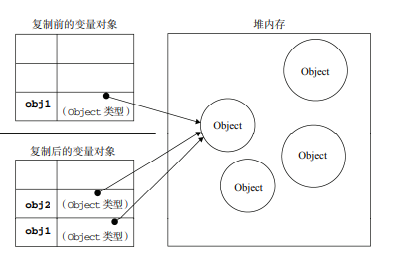
Var obj1=new Object();

Var obj2=obj1;

Obj1.name=”Ni”;

Alert(Obj2.name) ; //Ni

Obj1和obj2引用同一个对象。



4.1.3 传递参数

ECMAScript中所有函数中的参数都是按值传递的。也就是说，把函数外部的值复制给函数内部的参数，就和把值从一个变量复制到另一个变量一样。基本类型值的传递如同基本类型变量的复制一样，而引用类型值的传递，则如同引用类型变量的复制一样。

在向参数传递基本类型的值时，被传递的值会被复制给一个局部变量（即命名参数，或者用ECMAScript的概念来说，就是arguments对象中的一个元素），在向参数传递引用类型的值时，会把这个值在内存中的地址赋值给一个局部变量，因此这个局部变量的变化会反映在函数的外部。例子：

Function addTen(){

Num+=10;

Return num;}

Var count=20;

Var result=addTen(count); alert(count); //20 没有变化 alert(result) //30

函数AddTen()有一个参数num,参数实际上是函数的局部变量。在调用这个函数时，变量count作为参数被传递给函数，这个变量的值是20.于是数值20被复制给参数num以便在AddTen（）中使用。在函数内部，参数num的值被加上了10，但这一变化不会影响函数外部的count变量。参数num和count没有关系，只是值一样。

Function setName(obj){

Obj.name=”Nicholas”;

}  
var person=new Object();

setName(person);

alert(person.name) //Nicholas

创建一个对象，将它保存到变量person之中，这个变量被传递到setName函数中被复制给了obj，obj跟person一样，都是引用类型。所以，函数内部添加name属性之后，函数外部的person也有反映。因为person指向的对象在堆内存中只有一个，而且是全局对象。

4.4.1 检测类型

要检测一个变量是不是基本数据类型，typeof操作符可以解决。

Typeof操作符是确定一个变量是字符串，数值，布尔值，还是undefined的最佳工具。如果变量是null，则typeof操作符会返回”object”

例子： var n=null; alert(typeof o) //object

检测基本数据类型时可以用typeof，检测引用类型的值时，ECMAScript提供了instanceof操作符。语法： result=variable instanceof constructor

如果变量是给定引用类型（根据原型链来识别）的实例，那么instanceof操作符就会返回true.

例子： alert(person instanceof Object) //变量Person是Object吗

Alert(colors instanceof Array) //变量colors是Array吗

Alert(pattern instanceof RegExp) //变量pattern是RegExp吗

按照规定，所有引用类型都是Object的实例，因此，在检测一个引用类型和Object构造函数时，instanceof操作符始终会返回true.

如果使用instanceof操作符检测基本类型的值，则该操作符始终会返回false,因为基本类型不是对象。

4.2 执行环境和作用域

执行环境（exeution context,简单讲，称为“环境”是JavaScript中最为重要的一个概念。执行环境定义了变量或函数有权访问的其他函数，决定了它们各自的行为）。每个执行环境都有一个与之关联的变量对象（variable object），环境中定义的所有变量和函数保存在这个对象中。（虽然我们编写的代码无法访问这个对象。但解析器在处理数据时会在后台使用它。

）

全局执行环境是最外围的一个执行环境，根据ECMAScript实现所在的宿主对象不同，表示执行环境的对象也不一样。在web浏览器中。全局执行环境被认为是window对象，因此所有全局变量和函数都是作为window对象的属性和方法创建的。某个执行环境中的所有代码执行完毕，该环境被销毁。保存在其中的所有变量和函数定义也随之销毁（全局执行环境直到应用程序退出—例如关闭网页或浏览器时才被销毁）

每个函数都有自己的执行环境，当执行流进入一个函数时，函数的环境就会被推入一个环境栈中，而在函数执行之后，栈将环境弹出，将控制权饭回之前的允许环境。

ECMASrcipt中的执行流是这个方便的机制控制者。

当代码在一个环境中执行时，会创建变量对象的一个作用域链（scopr chain

.作用域链的用途，是保证对执行环境有权访问的所有变量和函数的有序访问。作用域链的前端，始终都是当前执行代码所在环境的变量对象。如果这个环境是函数，则其活动对象（active object）作为变量对象。

活动在最开始时只包含一个变量,即arguments对象（这个对象在全局对象中不存在）。作用域链中下一个变量对象来自包含（外部环境）一直延续到全局执行环境；全局执行环境的变量对象始终都是作用域链中的最后一个对象。

标识符解析是沿着作用域链一级一级地搜索标识符的过程，搜索过程始终从作用域链的前端开始，然后逐级地向后回溯。

例子：

Var color=”blue”;

Function changeColor(){

If(color==”blue”)

{

Color=”red”;

}else{  
color=”blue”;

}}

changeColor();

alert(“color is”+color);

这个例子中，函数changeColor()可以访问两个对象arguments对象和全局变量。

例子：

Var color=”blue”;

Function changeColor(){

Var anotherColor=”red”;

Function swapColors(){

Var tempColor=anotherColor;

anotherColor=color;

color=tempColor;

//这里可以访问color\anotherColor和tempColor

}

//这里可以访问color和anotherColor,但不能访问tempColor.

swapColors();

}

changeColor(); //这里只能访问color.

有三个执行环境

4.2.1 延长作用域链

1.try-catch 语句的catch块

2.with语句

这两个语句都会在作用域链的前端添加一个变量对象。对with语句来说，会将制定的对象添加到作用域链中，对catch语句，创建一个新的变量对象，其中包含的是被抛出的错误对象的声明。

4.2.2 没有块级作用域

JavaScript没雨块级作用域经常会导致理解上的困惑。

1. 声明变量

使用var声明的变量会自动被添加到最接近的环境中。在with语句中，最接近的环境是函数环境。如果初始化变量没雨使用var，该变量会自动被添加到全局环境。

1. 查询标识符

当某个环境中为了读取或写入而引用一个标识符时，必须通过搜索来确定标识符实际代表什么。搜索过程从作用域链的前端开始，向上逐级查询与给定名字匹配的标识符，如果在局部环境中搜索到了该标识符，搜索过程停止，变量就绪。如果在局部环境中没有找到该变量，搜索过程一直追溯到全局环境的变量对象。如果全局环境也没有找到，则意味着该变量尚未声明。

4.3 垃圾收集

Javascripy自动垃圾收集。

离开作用域自动标记为可以回收。标记清除。 引用计数。

第六章 面向对象的程序设计

6.2 创建对象

Object构造函数和对象字面量可以用来创建单个对象，这些方式有明显的缺点：使用同一个接口创建多个对象，会产生大量的重复代码。

6.2.1 工厂模式

工厂模式抽象了创建具体对象的过程。开发人员发明一种函数，用函数来封装特定接口创建对象的细节。例子：

Function createPerson(name,age,job)

{

Var o=new Object();

o.name=name;

o.age=age;

o.job-job;

o.sayName=function(){

alert(this.name);};

return o;

}

Var person1=createPerson(“Ni”,29,”a”);

var person2 = createPerson("Greg", 27, "Doctor");

工厂模式解决了创造多个相似对象的问题，但是没有解决对象识别的问题（即怎样知道一个对象的类型）

6.2.2构造函数模式

ECMAScript中的构造函数可用来创建特定类型的对象，像Object和Array这样的原生构造函数。例子重写：

Function Person(name,age,job){

This.name=name;

This.job=job;

This.sayName=function(){  
alert(this.name);

};}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

6.2.3原型模式

我们创建的每一个函数都有prototype(原型)属性，这个属性是一个指针，指向一个对象，而这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。

Prototype就是调用构造函数而创建的那个对象实例的原型对象。使用原型对象的好处是让所有对象实例共享它所包含的属性和方法。

Function Person(){}

Person.prototype.name=”NI”

Person.prototype.sayName=function{

Alert(this.name);

};

Var person1=new Person();

Person1.sayName(); //”NI”

虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值。

2.原型与in操作符

有两种方式使用in操作符，单独使用和在for-in循环中使用。在单独使用时，in操作符会在通过对象能够访问给定属性时返回true.例子：

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

alert(person1.hasOwnProperty(“name”)); //false

alert(“name” in person1) true

1. 更简单的原型语法

为了从视觉上更好地封装原型的功能。常见做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象。例子：

Function Person(){

}

Person.prototype={

Name:”Nicholas”,

Age:29,

Job:”Software EEngineer”

sayName:function(){

alert(this.name);

} };

原型的动态性

由于在原型中查找值的过程是一次搜索，因此我们对原型对象所做的任何修改都能够立刻从实例上反映出来，即使是先创建了实例后修改原型也照样如此。

1. 原型对象的问题

原型模式最大的问题是由其共享的本性导致的，对于包含引用类型值的属性来说，问题比较突出。

**6.3 继承**

ECMAScript只支持实现继承，实现继承主要依靠原型链来实现。

call 方法就是把目标方法绑定参数对象去执行。func.call(obj),此时func的执行体如果有this的话，其this指向obj~

var inst = new SubType(); 首先，SubType构造函数的this就是指向SubType实例，然后

**原型链继承**

原型链是实现继承的主要方法，思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

构造函数、原型、实例的关系：每个构造函数都包含一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针，

原型对象（指针指向构造函数，）=> 构造函数（指针指向原型对象） =>` 实例（内部指针指向原型对象）

实现原型链基本模式，代码:

Function SuperType(){

This.property=true;

}  
SuperType.prototype.getSuperValue=function(){

Return this.property;

}

Function SubType(){  
this.subproperty=false;

}

//继承了SuperType

Subtype.prottype=new SuperType();

Subtype.prototype.getSubValue=function(){

Return this.subproperty;

};

Var instance=new SubType();

alert(instance.getSuperValue()); //true

实现的本质是重写原型对象，代之以一个新类型的实例。换句话说，原来存在于SuperType的实例中的所有属性和方法，现在也存在于SubType.prtotype中。

**构造函数继承**

借用构造函数的技术（也叫伪造对象或经典继承）。

思想：子类型构造函数内部调用超类型构造函数。

例子：

Function SuperType(){

This.color=[“red”;

}

Function subtype()

SuperType.call(this) //func.call(obj ) 此时目标方法SuperType绑定参数对象，此时this指向new过的对象instance1.

}

Var instance1=new SubType();

Instance1.colors.push(“black”) //red,black

也可以传参：SuperType.call(this,”Nicolas”)

* + 1. 组合继承

组合继承，有时候也叫作伪经典继承，指的是将原型链和借用构造函数的技术组合在一块，从而发挥二者之长的一种集成模式。

思路：使用原型链实现对原型属性和方法的继承，而通过借用构造函数实现对实例属性的继承。

function SuperType(name){

this.name = name;

this.colors = ["red", "blue", "green"];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

Function SubType(name,age)(

SuperType.call(this,name);

This.age=age;

)

//继承方法

SubType.prototype=new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){

alert(this.age);

};

var instance1 = new SubType("Nicholas", 29);

instance1.colors.push("black");

alert(instance1.colors); //"red,blue,green,black"

instance1.sayName(); //"Nicholas";

instance1.sayAge(); //29

var instance2 = new SubType("Greg", 27);

alert(instance2.colors); //"red,blue,green"

instance2.sayName(); //"Greg";

instance2.sayAge(); //27

**6.3.5 寄生式继承**

寄生式继承与原型式继承紧密相关。寄生式继承与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再真的像是它做了所有工作一样返回对象。

例子：

Function createAnother(original){

Var clone=object(original) //通过构造函数创建一个新对象

Clone.sayHi=function(){ //以某种方式来增强对象

Alert(“hi”);

}

Return clone; //返回这个对象

}

Var person={name:”Ni”,friends:[“shely”]};

Var anotherPerson=createAnother(person);

anotherPerson.sayHi() //hi

第七章 函数表达式

定义函数的方式有两种：一种是函数声明，另一种是函数表达式。

函数声明例子：

Function functionName(arg0,arg1){ //函数体}

关于函数声明，它的重要特征就是函数声明提升。在执行代码钱会先读取函数声明，这意味着可以把函数声明放在调用它的语句后面。

SayHi();

Function SayHi(){

Alert(“hi”)}

第二种创建函数的方式是使用函数表达式。例子：  
var functionName=function(arg0,arg1){ //函数体}；

这种情况下创建的函数叫做匿名函数，因为funtion关键字后面没有标识符。匿名函数的name属性是空字符串。

7.2 闭包

闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包常见方式，就是在一个函数内部创建另一个函数。

作用域链有一个副作用，闭包只能取得包含函数中任何变量的最后一个值。闭包保存的是整个变量对象，而不是某个特殊的变量。

Function createFunctions(){

Var result=new Array();

For(var i=0;i<10;i++){  
result[i]=function(){

Return I;

);

Return result;}

实际上每个函数都返回10.

现在创建另一个匿名函数强制让闭包行为符合预期。

function createFunctions(){

var result = new Array();

for (var i=0; i < 10; i++){

result[i]=function(num){

return function(){

return num;

};}(i)

Return result;}

7.2.2 关于this对象

闭包中使用this对象也可能导致一些问题。This对象是在运行时基于函数的执行环境绑定的。

匿名函数的执行环境具有全局性，因此其this对象通常指向window。

例子：

Var name=”the window

Var object={

Name:”My Object”,

getNameFunc:function(){

return function(){

return this.name;

};}};

Alert(object.getNameFunc()()): //”the window(非严格模式)

函数被调用自动取得两个特殊变量：this 和arguments.内部函数在搜索这两个变量时，只会搜索到活动变量为止。

var name = "The Window";

var object = {

name : "My Object",

getNameFunc : function(){

var that=this;

return function(){

return that.name;

);}};

Alert(object.getNameFunc()) //”My Object

7.2.3 内存泄漏

闭包在IE版本中会导致一些特殊的问题。具体来说，如果闭包的作用域链保存着一个HTML元素，那么就意味着该元素将无法被销毁。

把变量element设置为null,这样就能解除对DOM对象的引用·。

7.3 模仿块级作用域

JavaScript没有块级作用域的概念，意味着在块语句中定义的变量，实际上是在包含函数中而非语句中创建的。

Function outputNumbers(count){

for (var i=0; i < count; i++){

alert(i);

}

alert(i); //计数

}

下面这样错误地重新声明同一个变量，也不会改变它的值。

var i; //重新声明变量

；遇到这种情况，它只会对后续的声明视而不 见（不过，它会执行后续声明中的变量初始化）匿名函数可以模仿块级作用域解决这个问题。

(function(){

//这里是块级作用域

})()

例子：

function outputNumbers(count){  
(function(){

for (var i=0; i < count; i++){ alert(i);

}

))();

alert(i); //导致一个错误！ }

)

7.4私有变量

严格来说，JavaScript没有私有成员的概念。所有对象属性都是共有的。不过，倒是有一个私有变量的概念。私有变量包括函数的参数、局部变量和函数内部定义的其他函数。

第十三章 事件

第23章 离线应用与客户端存储

本章内容： 进行离线检测 使用离线缓存 在浏览器中保存数据

支持离线web应用开发是H5的另一个重点。离线web就是在没网的情况下也可以运行的应用。

为什么?因为前端开发人员一直希望web应用与传统的客户端通常竞技。

23.3 数据存储

随着web应用程序出现，也产生了对于能够直接在客户端上存储用户信息的要求。

23.3.3 web存储机制

Web Storage的目的是克服 由cookie带来的一些限制，当数据需要被严格控制在客户端上时，无须持续地将数据发回服务器。

Web Storage两个目标是： 提供一种在cookie之外存储绘画数据得到一种途径。 θ 提供一种存储大量可以跨会话存在的数据的机制。

1.storage类型

Storage的实例与其他对象类似，有如下方法：

Clear() : 删除所有值，

GetItem(name)

2.sesion storage对象

Session Storage对象存储于某个会话的数据，也就是数据只保持到浏览器关闭。这个对象就像会话 cookie，在浏览器关闭后消失。

存储在session Storage中的数据可以跨越页面刷新存在（浏览器支持）。

因为session Storage 对象绑定在某个服务器会话，。所以文件在本地运行不可用。

存储在sessionStorage中的数据只能由最初给对象存储数据的页面访问到，所以对多页面应用有限制。