**JavaScript作用域链**

当代码在一个环境中执行时，会创建变量的一个作用域链（Scope chain）。

作用域链的作用是保持对执行环境有权访问的所有变量和函数的有序访问。

它的调用顺序始终是从前端开始，逐级调用，直到全局执行环境。

**名词解释：**

**环境（执行环境）：执行环境定义了变量或函数有权访问的其他数据，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都保存在这个对象中。**

**JavaScript没有块级作用域**

在其他类C语言中，由花括号封闭的代码都有自己的作用域。

If(true){

Var color = “blue”;

}

Alert(color); //blue

如果是类C语言，color会在if语句执行完之后被销毁。但在JavaScript中，if语句声明的变量会被添加到当前执行环境中（在这里是全局环境）。在使用for语句中尤其要注意这一点。

For(var i = 0;I < 10;i++){

}

Alert(i); //10

**JavaScript变量声明**

使用var声明的变量会自动被添加到最接近的环境中。在函数内部，最接近的环境就是函数的局部环境。在with语句中，最接近的环境就是函数环境。如果初始化变量时没有使用var声明，该变量会自动添加到全局环境。

**Function类型**

在JavaScript中，有意思的一点是，函数是一个对象，函数名是一个指针。

**Global对象**

Global对象（全局），可以算是JavaScript中最特别的一个对象了。因为 不管你是从什么角度来看，这个对象都是不存在的。事实上，没有全局函数，或者全局变量；所有在全局定义的属性和函数，都是Global对象的属性。

1. **URI编码方法**

encodeURI ()和encodeURIComponent()都可以对URI进行编码，唯一的不同是，encodeURI主要是对整个URI进行编码，而encodeURIComponent只是对URI中的某一段进行编码。

EncodeURI不会对URI中的特殊字符进行编码，如冒号，下划线，#，问号；而encodeURIComponent则会对任何非标准字符进行编码。

与它们相反的是decodeURI()和decodeURIComponent()方法。

//Global对象

//URI编码方法（encodeURI,encodeURIComponent)

var http = "http://www.baidu.com?w=湖北 武汉";

console.log("encodeURI:" + encodeURI(http));//http://www.baidu.com?w=%E6%B9%96%E5%8C%97%20%E6%AD%A6%E6%B1%89

console.log("encodeURIComponent:" + encodeURIComponent(http));//http%3A%2F%2Fwww.baidu.com%3Fw%3D%E6%B9%96%E5%8C%97%20%E6%AD%A6%E6%B1%89

1. **eval()方法**

这个方法接受一个参数，它会将传入的参数当做实际的ECMAScript语句来解析，然后把执行结果插入到原位置。

//eval会解析参数，并插入到原位置。

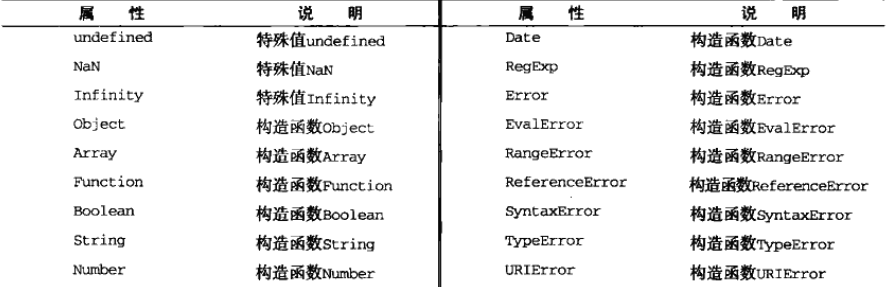
eval("console.log(sum)");//undefined

var sum = 10;

eval("console.log(sum)");//10

1. **Global属性**

这个方法接受一个参数，它会将传入的参数当做实际的ECMAScript语句来解析，然后把执行结果插入到原位置。



1. **window对象**

ECMAscript虽然没有指定如何访问Global对象，但Web浏览器都是将这个全局对象作为window对象的一部分加以实现的。因此，全局作用域中声明的所有变量和函数，就都成为了window对象的属性。

<input type="button" value="SayHello" onclick="InvokeSayHello()" />

//SayHello是一个全局函数，属于window的一个属性

function SayHello() {

console.log("Hello !!!");

}

function InvokeSayHello() {

window.SayHello();

}

**原型（Prototype）**

**1.理解原型对象**

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性。捜索首先从对象实例本身开始。如果在实例中找到了具有给定名字的属性，则返冋该属性的值；如果没有找到， 则继续搜索指针指向的原型对象.在原型对象中査找具有给定名字的属性。如果在原型对象中找到了这个属性，则返回该属性的值。也就足说在我们调用perscml.sayName()的时候.会先后执行两次搜索。首先，解析器会问：“实例personl有sayName属性吗？ ”答：“没有。”然后，它继续搜索，再 问：“personl的原型有sayName属性吗？ ”答：“有。”于是，它就读取那个保存在原型对象中的函 数。当我们调用Person2.sayName()时，将会重现相同的搜索过程，得到相同的结果。而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

**2.原型模式的问题**

原型模式也不是没有缺点的。首先它省略了构造函数初始化参数这一环节，结果所有的实例在默认情况下都将取得相同的属性值。虽然这会在一定程度上造成不便，但这不是原型的最大问题。原型模式最大的问题是由其共享的本性所导致的。

原型中所有的属性是被所有实例所共享的。这种共享对函数而言非常合适。对于那些包含基本值的属性倒也说的过去，毕竟，通过在实例上添加一个同名的属性，可以隐藏原型中的对应属性。然而，对于包含引用类型的属性来说问题就比较突出了。

**3.组合使用构造函数模式和原型模式**

创建自定义类型的最常见模式，就是组合使用构造函数模式和原型模式，构造函数用与定义实例属性，原型用于定义函数和共享的属性。这样，每个实例都有自己的实例属性副本，但同时又共享对方法的引用，最大限度节省了内存。

//组合使用构造函数模式和原型模式

//每个实例既可以有自己的属性，同时又共享这方法，最大限度节省了内存

function Person(name, age, job) {

this.Name = name

this.Age = age;

this.Job = job;

}

Person.prototype = {

constructor: Person,

SayName: function () {

console.log("Name:" + this.Name + ",Age:" + this.Age + ",Job:" + this.Job);

}

}

var jack = new Person("Jack", 18, "Student");

var Scott = new Person("Scott", 40, "Teacher");

jack.SayName();

Scott.SayName();

console.log(jack.Name + "\_" + Scott.Name);

**JavaScript模拟块级作用域**

匿名函数可以用来模拟块级作用域。

调用JavaScript将function作为一个函数声明的开始，而函数后面不能直接跟括号。然而函数表达式的后面可以跟括号。要将函数声明转换为函数表达式，只要像下面这样给它加一对括号即可。

(function(){

//这里是块级作用域

})

**数据类型隐式转Boolean**



数据类型转换还有其他套路，比如：

!!“c” => true 相当于把c转成boolean类型

+ “123” => 123相当于把字符串转成int类型

“” + 999 =>“999”相当于把数字转成string类型

**Call()和Apply()方法**

Call()方法可以用来调用方法，并能指定this的值和参数，Apply()方法和Call()方法类似。只是它要求传入的参数必须是以数组的形式。

**this用法**

在函数内部运行时，内部会自动有一个this对象可以使用

在不同的场合，this会有不同的值，this就是函数运行时所在的环境

情况一：纯粹的函数调用

这是函数的最通常用法，属于全局性调用，因此this就代表全局对象。请看下面这段代码，它的运行结果是1。

var x = 1;

function Fun1() {

console.log(this);

console.log(this.x);

}

Fun1();//1

情况二：作为对象的方法调用

函数还可以作为某个对象的方法调用，这时this就指这个上级对象。

function Fun2() {

console.log(this);

console.log(this.y);

}

var obj = {};

obj.y = 1;

obj.Fun2 = Fun2;

obj.Fun2();//1

情况三：作为构造函数调用

所谓构造函数，就是通过这个函数，可以生成一个新对象。这时，this就指这个新对象。

function Fun3() {

console.log(this);

this.x = 2;

}

var f3 = new Fun3();

console.log(f3.x);//2，这里的x是局部变量

情况四：apply(),call()调用

它的第一个参数就表示改变后的调用这个函数的对象

var z = 1;

function Fun4() {

console.log(this);

console.log(this.z);

}

Fun4.call(this);//1

var person = {

z: 2

};

Fun4.call(person);//2

Get请求有限制一般不超过2kb

Cookie请求也有限制，一般不超过4kb

Post理论上是没有限制的，但服务器也可以限制上传文件的大小

**BOM**

**Window对象**

BOM对象的核心是window，它表示浏览器的一个实例。在浏览器中，window对象有双重角色，它既是通过JavaScript访问浏览器窗口的一个接口，又是ESMAScript规定的Global对象。这意味着在网页中定义的任何一个对象，变量，函数，都以window作为其Global对象，因此有权访问parseInt()等方法。

窗口关系及框架（frame）

我们在访问窗口里的框架时，最好采用top而非window（例如：top.frame[0]）。

因为top对象始终指向最外层的框架，也就是浏览器窗口。使用它可以确保在一个框架中正确地访问另一个框架。因为对于框架中的任何编写的代码而言，其中的window对象指向的都是那个框架的特定实例，而非最高层框架。

与top相对的另一个window对象是parent。顾名思义，parent（父）对象始终指向当前框架的直接上层框架。

**浏览器检测**

浏览器检测有三种方法：能力检测，怪癖检测，用户代理(userAgent)检测

能力检测（又称特性检测）的目标不是识别特定的浏览器，而是识别浏览器的能力。

采用这种方式不必顾及特定的浏览器如何如何，只要确定浏览器支持特定的能力，就可以给出解决方案。

推荐使用typeof 操作符用于确定属性是否存在且正确。

怪癖检测的目标是识别浏览器的特殊行为 。但与能力检测不同，怪癖检测是想要知道浏览器存在什么缺陷。这通常需要一小段代码，以确定某个特性不能正常工作。

用户代理(userAgent)检测通过检测用户代理字符串来确定实际使用的浏览器。我们可以通过访问window对象的navigator.userAgent属性访问。在客户端，用户代理检测一般被当做一种万不得已采用的方法，其优先级排在能力检测和怪癖检测之后。

|  |  |
| --- | --- |
| 引擎 | 支持的浏览器 |
| Gecko | Firefox,Netscape |
| WebKit | Safari,Chrome,iOS,Android |
| KHTML | Konqueror |
| Opera | Opera |
| IE | IE8,IE11,Edge |

[^abc]表示匹配非abc的所有字符，^是取反的意思

如果想在后面取出匹配的字符，可以将要取出的正则字符用括号括起来

[?:abc] 匹配abc，但是不记住匹配项。这是为了将abc看做一个整体与后面的运算符一起使用。如表达式“[?:123]?456”可以匹配123456也可以匹配456.

在大多数情况下，识别了浏览器的呈现就足以为我们采取正确的操作提供依据了。可是只有呈现引擎还不能说明存在所需的Javascript功能