**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。

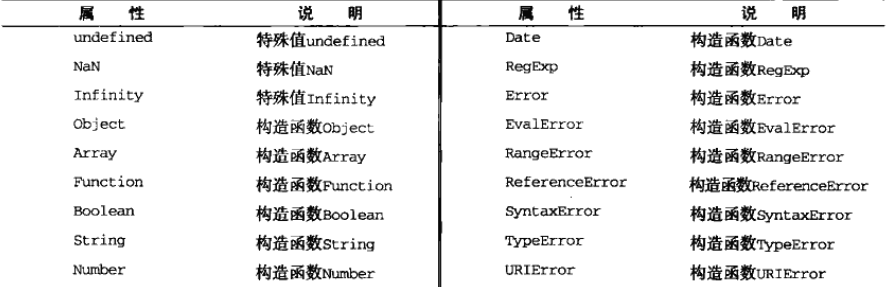
eval("console.log(sum)");//undefined

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

**DOM**

理解DOM的关键，就是理解DOM对性能的影响。DOM操作往往是javascript程序开销最大的部分，而因访问NodeList导致的问题最多。NodeList对象都是“动态的”，这就意味着每次访问NodeList对象，都会运行一次查询。有鉴于此，最好的办法就是尽量减少DOM操作。

**选择符（Selector）API**

对DOM的两个主要拓展是选择符API和HTML5。选择符API是最常用的一项功能，就是根据Css选择器和某个模式匹配DOM元素。

Selector API是由W3C发起制定的一个标准，致力于让浏览器原生支持CSS查询。

它的核心是两个方法：querySelector和querySelectorAll。在兼容的浏览器上，可以通过Document和Element类型的实例调用它们。目前支持的浏览器有IE8+,Firefox3.5+,Safari3.5+,Chrome和Opear10。

querySelector()方法接受一个CSS选择符，返回与该模式匹配的第一个元素，如果没有找到元素，则返回null。

querySelectorAll()方法接收的参数与querySelector一样，都是一个Css选择器，但返回的是所有匹配的元素，这个方法返回的是一个NodeList对象。

**HTML5**

HTML5规范围绕如何使用新增标记定义了大量javascript API

1. getElementByClassName()方法，该方法接受一个参数即包含一个或多个类名的字符串，返回带有指定类的所有元素的NodeList。

使用这个方法可以更方便地为带有某些类的元素添加事件处理函数，从而不必再局限于使用ID或者签名。不过由于它也是返回的额NodeList所以同样具有性能问题。

目前支持的浏览器有：IE9+,Firefox3.5+,safari3.5+,Chrome,Opera9.5+

1. 焦点管理

我们可以调用document.activeElement属性，获取当前的焦点元素，这样我们就不必靠猜测哪个元素获取了焦点了。

目前支持的浏览器有：IE4+,Firefox3+,Safari 4+,Chrome,Opera8+

1. HTMLDocument的变化

HTML5拓展了HTMLDocument，增加了新功能。

readyState属性，它有两个值：loading，正在加载文档；comp0lete，加载完毕。使用它最恰当的场景是通过它来实现一个指示文档已经加载完成的指示器。

支持的浏览器有：IE4+,Firefox3.6+,Safari,Chrome,Opear 9+

自从IE6开始区分渲染页面是标准还是混杂的，检测页面的兼容模式就成为了浏览器的必要功能。IE为此给document添加了一个名为compatMode属性，在标准模式下，它的值等于CSS1Compat；在混合模式下它的值为BackCompat。

支持的浏览器有：Firefox,Safari3.1+,Opera,Chrome

自定义数据属性

HTML5规定可以为元素添加非标准的属性，但要添加前缀data-。这些属性可以任意添加，随便命名。

在JS中，我们可以通过元素的dataset属性来访问自定义的属性值。每个data-name形式的属性都会有一个对应的属性，只不过属性名没有data-前缀。

例如：

Document.body.dataset.name = “abc”;

InnerHTML和OuterHTML的区别：

InnerHTML仅可以设置或获取调用元素内部的内容；

OuterHTML可以设置或获取整个标签的内容；

OuterHTML支持的浏览器包含：IE4+,Safari4+,Chrome,Opera 8+,Firefox7+

内存与性能问题：

在删除带有事件的元素时，假如没有将绑定的事件一并删除。如果反复出现这种情况，页面占用内存会明显增加。

在使用innerHTML时，如果需要频繁访问，最好的办法是单独构建字符串，然后一次性将字符串结果赋值给innerHTML；这种做法比多次调用innerHTML效率高得多。