**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。

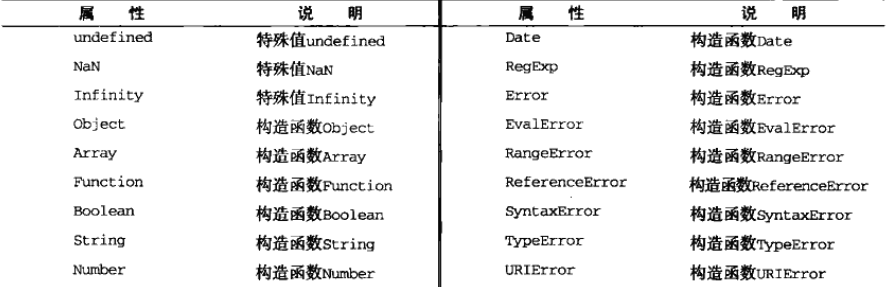
eval("console.log(sum)");//undefined

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。