**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。

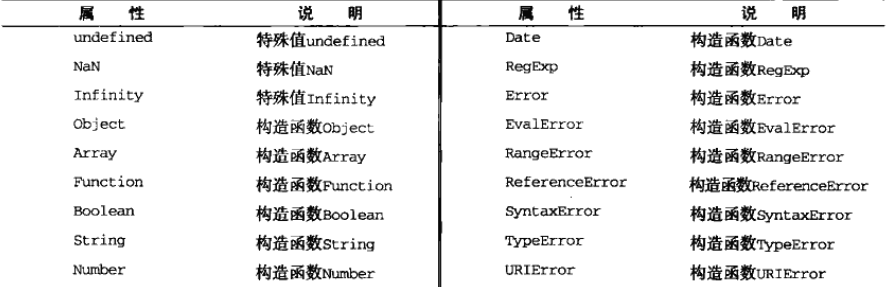
eval("console.log(sum)");//undefined

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。

**事件**

**事件冒泡与事件捕获**

IE的事件流叫做事件冒泡(event bubbing)，即事件开始时由最具体的元素(文档中嵌套最深的那个节点)接收，然后逐级向上传播到较为不具体的节点(文档)。

所有现代的浏览器都支持事件冒泡，但在具体实现上还是有一定差别的。IE5.5及更早版本的事件冒泡会跳过<html>元素(从body直接到document)。IE9,Firefox,Chrome,Safari则将事件冒泡到window对象。

Netscape Communicator团队提出的另一种事件流叫做事件捕获(event capturing)。

事件捕获的思想是不太具体的节点应该更早接收到事件，而最具体的节点应该最后接收到事件。

**DOM事件流**

“DOM2级事件”规定的事件流包括三个阶段：事件捕获阶段，处于目标阶段和事件冒泡阶段。

**事件处理程序**

事件就是用户或浏览器自身的某种动作。诸如：click，load和mouseover等，都是事件的名字。而响应某个事件的函数叫做事件处理程序。事件处理程序以“on”开头，因此click的事件处理程序是onclick。为事件指定处理函数的方法有好几种，下面我们一一介绍。

HTML事件处理程序：

<a href="javascript:void(0);" onclick="showMessage()">百度</a>

缺点有二：

1. 时差问题，用户可能会在已进入页面时，就点击元素，这时函数还可能未被完全加载。这时就会引发错误。为此，很多HTML事件处理程序都会被封装在一个try-catch块内，以便错误不会付出水面。

<a href="javascript:void(0);" onclick="try { showMessage('腾讯') }catch{ }">腾讯</a>

1. 让HTML和JavaScript紧密耦合，如果要更换事件处理函数，就要改动两个地方：HTML和JavaScript，而这正是许多开发人员摒弃HTML事件处理函数的原因。

DOM0级事件处理程序：

通过将一个函数赋值给一个事件处理程序属性。它具有简单，并具有跨浏览器的优势。

例如：

var but = document.getElementById("ButtonValid");

but.onclick = function () {

console.log(this);

}

删除这个DOM0级别的方法指定的事件处理函数，只要将null赋值给这个属性即可。

but.onclick = null;

DOM2级事件处理程序：

“DOM2级事件”定义了两个方法来用于指定和删除事件处理程序：addEventListener()和removeEventListener()。所有的DOM节点都包含这两个方法，并且它们接收3个参数：事件名（例如：click，load等），函数，bool（true表示是在捕获阶段调用事件处理函数，false表示在冒泡阶段调用事件处理函数）。

var but2 = document.getElementById("ButtonDOM2");

//1.添加一个匿名函数

but2.addEventListener("click", function () {

console.log(this);

}, false);

调用removeEventListener()移除时，有一点需要注意，那就是函数不能是匿名函数，并且参数相同。

使用DOM2级别事件处理函数的好处是：可以添加多个事件处理函数

支持的浏览器：IE9+，Firefox，Chrome，Opera，Safari

IE事件处理程序：

IE实现了与DOM类似的两个方法：attachEvent()和detachEvent()。这两个方法都接受两个参数：事件名称和函数。默认为冒泡阶段触发。

**事件对象**

在触发DOM的某个事件时，会产生一个事件对象event，这个对象包含所有与事件有关的信息。包括导致事件的元素，事件类型以及其他与特定事件相关的信息。

在事件处理内部this始终等于CurrentTarget的值，而taregt则只包含事件的实际目标。

要阻止事件的默认行为，可以使用preventDefault()方法(当cancelable为true时)。

StopPropagation()方法用于立即停止事件在DOM层次传播，即取消事件的捕获或冒泡。

**事件类型**

UI (User Interface) 事件，当用户与页面上的元素交互时触发

Load事件：当页面完全加载后，就会触发window上的load事件。

Unload事件：在文档被完全卸载后触发。只要用户从一个页面切换到另一个页面，就会触发。

Resize事件：当浏览器调整到一个新的高度或宽度时，就会触发resize事件。注意不要在这个事件中加入计算逻辑，避免性能问题。

EventUtil.addHandler(window, "resize", function (event) {

//防抖函数

//每次调用函数都会清除之前的计数器，并重新计数，计数完成时，执行操作。

var timer;

return function () {

if (timer) {

clearTimeout(timer);

}

timer = setTimeout(function () { //setTimeout返回唯一标识符，可以使用它来取消计数器

console.log("123");

},500);

}

}());//此处的双括号表示，立即调用返回值

Scroll事件：当用户滚动带有流动条的元素中的内容时，在该元素上面触发。

EventUtil.addHandler(window, "scroll", function (event) {

//函数节流

var timer;

return function () {

if (timer) {

clearTimeout(timer);

}

timer = setTimeout(function () { //setTimeout返回唯一标识符，可以使用它来取消计数器

if (document.compatMode == "CSS1Compat") {

//标准模式

console.log("Left:" + document.documentElement.scrollLeft);

console.log("Top:" + document.documentElement.scrollTop);

}

else {

//混杂模式

console.log("Left:" + document.body.scrollLeft);

console.log("Top:" + document.body.scrollTop);

}

}, 500);

}

}());

焦点事件，在页面获得或失去焦点时触发

Blur事件：在元素失去焦点时触发。这个事件不会冒泡；所有浏览器都支持它；

Focus事件：在元素获得焦点时触发。这个事件不会冒泡，所有浏览器都支持它；

Focusin事件：在元素获得焦点时触发。 这个事件与HTML事件focus等价，但它冒泡，支持这个事件的浏览器有IE5+ safari5.1+ Opera11.5+ 和 Chrome;

Focusout事件: 在元素获得失去时触发。 这个事件是HTML事件blur的通用版本，，支持这个事件的浏览器有IE5+ safari5.1+ Opera11.5+ 和 Chrome;

鼠标与滚轮事件

Click事件：在用户单机鼠标左键或按下回车键时触发。

DbClick事件：用户双击鼠标左键时触发。

MouseDown事件：在用户按下鼠标任意按钮时触发。

MouseEnter事件：鼠标从外部首次移入到元素范围内触发。

MouseLeave事件：鼠标移出元素范围之外触发。

MouseMove事件：当鼠标指针在元素内部移动时重复触发。

MouseOut事件：在鼠标指针位于一个元素上方，然后用户将其移入到另一个元素时触发。

MouseOver事件：在鼠标指针位于一个元素外部，然后用户将其首次移入到另一个元素边界之内时触发。

MouseUp：在用户释放鼠标指针时触发。

几个鼠标坐标位置：

1. 客户区坐标位置

鼠标事件都是在浏览器视口中的特定位置上发生的。这个位置保存在事件对象的clientX和clientY属性中。

1. 页面坐标位置

pageX和pageY能告诉你事件是在页面中什么位置发生的。换句话说，这两个属性表示光标在页面中的位置，因此坐标是从页面本身而不是视口左边或顶边计算的。

在页面没有滚动的情况下，pageX和pageY和clientX和clientY是相等的。

IE8及更早的版本不支持页面坐标位置，不过可以使用客户区坐标位置和滚动信息计算出来。

EventUtil.addHandler(document.getElementById("div1"), "click", function (event) {

//兼容IE8及之前版本

event = EventUtil.getEvent(event);

var pageX = event.pageX,

pageY = event.pageY;

if (pageX === undefined) {

pageX = event.clientX + (document.body.scrollLeft || document.documentElement.scrollLeft);

}

if (pageY === undefined) {

pageY = event.clientY + (document.body.scrollTop || document.documentElement.scrollTop);

}

//console.log("pageX:" + pageX);

//console.log("pageY:" + pageY);

});

1. 屏幕坐标位置

screenX和screenY属性可以确定鼠标事件发生时鼠标指针相对于整个屏幕的坐标信息。

**性能优化**

在JavaScript中，添加到页面上的事件处理程序数量将直接关系到页面整体的运行性能。导致这一问题是多方面的。首先，每个函数都是一个对象，都会占用内存；内存中的对象越多，性能就越差。其次，必须指定所有事件处理程序而导致DOM访问次数，会延迟 整个页面交互就绪时间。

对于“时间处理程序过多”问题的解决方案就是**事件委托**。事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一个类型所有的事件。例如，click事件会一直冒泡到document层次。也就是说，我们可以为整个页面指定一个onclick事件，而不必给每个可单击的元素分别添加事件处理程序。（减少页面中事件处理程序的数量）

EventUtil.addHandler(document.getElementById("table1"), "click", function (event) {

event = EventUtil.getEvent(event);

var target = EventUtil.getTarget(event);//获取目标

if (target.id && target.id.substr(0, 4) == "link") {

//这样就可以只声明一个事件了，不用再像传统的方法，每一个元素都声明一次事件。。。

}

});

每当将事件处理函数指定给元素时，运行中的浏览器代码与支持页面交互的JavaScript代码之间就会建立一个连接。这种连接越多，页面执行起来就越慢。我们可以在不需要的时候移除事件处理程序。

在两种情况下，可能会造成性能问题。第一种是从文档中移除带有事件处理函数的程序元素时，例如使用removeChild()和replaceChild()方法，但更多的使用了innerHTML替换页面中的某一部分。

导致“空事件处理程序”的另一种情况是，在卸载页面的时候。如果在页面卸载之前没有干净的清除事件处理程序，那它们就会滞留在内存中。每次加载完页面再卸载，内存中滞留的对象数量就会增加，因为事件处理程序没有被清除。

一般来说，最好的做法是在页面卸载之前，先通过onunload事件处理程序移除所有的事件处理程序。

//注册事件处理程序

document.getElementById("link1").onclick = function () {

console.log(window.event.target);

}

document.getElementById("link2").onclick = function () {

console.log(window.event.target);

}

window.onunload = function () {

//清除事件处理程序

document.getElementById("link1").onclick = null;

document.getElementById("link2").onclick = null;

}

**错误处理**

**常见错误处理**

由于JavaScript是松散类型的，而且也不会验证函数的参数，因此错误只会在代码运行期间出现。一般来说，有三种错误：

1. 类型转换错误
2. 数据类型错误
3. 通信错误

**类型转换错误**常常发生在可能会自动转换数据类型的语言结构上。在使用==或!=操作符，或者在if,for,while等流程控制语句使用非布尔值时，最常发生的错误。

我们可以使用**全等（===）**和**不全等（!==）操作符**，以避免类型转换。因为它们首先比较数据类型，然后才比较值是否相等，因此我们强烈推荐使用。

容易发生类型转换出错的另一个地方，就是流程控制语句。想if之类的语句在确定下一步操作之前会自动把任何值转成布尔值。例如：

function Fun1(str1, str2, str3) {

var str = str1 + str2;

if (str3) { //这样写容易出现bug

str = str + str3;

}

return str;

}

这个判断的问题在于并非只有string类型的值可以被转成true，假如str3的值是1，也可以通过if语句。我们可以使用下面的方法避开这个漏洞。

function Fun1(str1, str2, str3) {

var str = str1 + str2;

if (typeof str3 == "string") { //需要加上类型判断

str = str + str3;

}

return str;

}

由于JavaScript是松散类型的，所以在不使用变量之前，不能确保它们的数据类型。为了保证不会发生数据类型的错误，只能依靠开发人员编写适当的数据类型检测代码。

function GetQueryString(url) {

if (typeof url == "string") { //确保类型安全

var pos = url.indexOf("?");

if (pos > -1) {

return pos.substring(pos + 1);

}

}

return "";

}

在确切知道什么类型的情况下，推荐使用instanceof来检测其数据类型。

function ReverseArray(arr) {

if (arr instanceof Array); {

arr.sort();

arr.reverse();

}

}

大体来说，基本类型的值应该使用typeof运输符来检测，引用类型则应该用instanceof来检测。

**抛出错误**

抛出错误也是一个调试代码的好办法。如果错误信息具体，基本上就可以把它当做错误来源的依据。

**JSON**

关于JSON最重要的是理解它是一种数据结构，而不是一种编程语言。虽然具有相同的语法格式，但JSON并不从属于JavaScript。而且并非只有JavaScript能使用JSON，毕竟JSON只是一种数据格式，很多语言都支持JSON的解析器和序列化器。

**语法**

JSON的语法可以表示以下三种类型的值：

1. 简单值：表示字符串，数字，NULL，布尔值。但是它不支持undefined。例如：3，”hello wolrd”,true等。。
2. 对象：对象作为一种复杂的数据类型，表示一组有序的键值对。每个键值对中的值可以是简单值，也可以是复杂值。例如：{“name”:”Wang”,”age”:18}
3. 数组：数组也是一种复杂的数据类型，表示一组有序的列表。数组的值也可以是简单或复杂类型。例如[{“name”:”Wang”},{“name”:”Cheng”}]

与JavaScript不同的是，JSON中对象的属性名任何时候都必须加双引号。手工编写JSON时，忘记给对象属性名加上双引号或者把双引号写成单引号是常见的错误！

**解析与序列化**

早期的JSON解析器基本上就是使用JavaScript的eval()函数。ECMAScript 5对解析JSON的行为进行了规范，定义了全局对象JSON。支持它的浏览器包含：IE8+,Firefox 3.5 + ，Safari 4+,Chrome,Opera10.5+。对于较早的浏览器可以使用一个shim:http://github.com/douglascrockford/JSON-js在较低版本浏览器中。

JSON对象有两个方法：stringify()和parse()，这两个方法分别用于JS对象转JSON字符串和JSON字符串转JS对象。

JSON.stringify()方法除了序列化JSON对象外，还可以接收两个参数，第一个参数是过滤器，它可以是一个数组（用于包含需要显示的属性名），也可以是一个函数（该函数有两个参数，一个是键名，一个是键值）。

第三个参数表示缩进的空格数量。

JSON.parse()方法除了反序列化JSON对象外，还可以接收一个函数参数，它有两个参数，一个是键一个是值。

**最佳实践**

**变量名注释风格**

由于自JavaScript中变量是松散类型的，很容易就忘记变量所表示的数据类型，有三种方式表示数据类型的方式。

第一种是初始化时指定变量类型

var num = 0; //int

var name = ""; //string

var isDelete = false; //Boolean

初始化化时指定一个特定数据类型可以很好地指明变量的类型。但缺点是它无法用于函数声明中函数的参数。

第二种是使用匈牙利命名法，在变量名前面添加一个或多个字符表示数据类型。如：i表示int，o表示object，s表示string，f表示float，d表示double。

var iNum; //int

var sName; //string

var bIsDelete; //Boolean

匈牙利命名法的好处是函数参数一样可以使用，但它的缺点是让代码某种程度上难以阅读。

第三种方式是使用类型注释。类型注释放在变量名右边，但是在初始化左边

var num /\* int \*/ = 0;

var name /\* string \*/ = "";

var isDelete /\* boolean \*/ = false;

这种方式维持了代码的可读性，同时注入了类型信息。它的缺点是你不能用多行注释一次注释大量代码，因为类型注释也是多行注释，两者会冲突。

**松散耦合**

只要应用的某个部分过分依赖与另一部分，代码就是耦合过紧，难于维护。紧密耦合的代码难于维护并且需要经常重写。

因为Web应用所涉及的技术，有多种情况会使它变的耦合过紧。必须小心这些情况，并尽可能维护耦合弱的功能。

1. 解耦HTML/JavaScript

在Web上，HTML和JavaScript各自代表了解决方案中的不同层次：HTML表示数据，JavaScript是行为。有很多种方法可以将它们连接起来，但有的方法会使得它们耦合过紧。比如直接写在HTML中的JavaScript，使用内联代码的<script>元素或是使用HTML属性来声明事件处理程序，都是过于紧密的耦合。我们应该采用添加外部文件引用和使用DOM附加的方式来包含JavaScript。

HTML和JavaScript的紧密耦合也可以在相反的方向上成立：JavaScript包含HTML。这通常会出现在需要用innerHTML插入一段HTML文本。你可以在页面中先隐藏这段待添加的HTML，必要时只需激活它就可以了。这样JavaScript中就不必嵌入HTML了。

将HTML和JavaScript解耦可以节省调试过程中花费的时间，更加容易确定错误的来源，也减轻了维护的难度：更改行为只需在JavaScript文件中进行，而更改标记只需在HTML中进行。

1. 解耦CSS/JavaScript

另一个Web层则是CSS,它主要负责页面的显示。JavaScript和CSS也是非常紧密相关的：他们都是HTML之上的层次，因此常常一起使用。但是，和HTML与JavaScript的情况一样，CSS和JavaScript 也可能会过于紧密地耦合在一起。最常见的紧密耦合的例子是使用JavaScript来更改某些样式，如下所示：

s.style.color = "red";

s.style.background = "white";

由于CSS负责页面的显示，当显示出现任何问题时都应该只是查看css文件来解决。然而，当使用了 JavaScript来更改某些样式的时候，比如颜色，就出现广第二个可能需要修改和必须检查的地方。结果是JavaScript也在某种程度负责了页面的显示，并与CSS紧密耦合了。如过未来需要更改样式表， CSS和JavaScript文件可能都需要修改。这就绐开发人员造成了维护上的噩梦。所以在这两个层次之间必须有清晰的划分。

现代Web应用常常要使用JavaScript来修改样式，所以虽然不可能完全将CSS和JavaScript解耦， 何是还是能让耦合更松散的。这是通过动态修改样式类而非特定样式来实现的，如下例所示：

s.className = "MainColor";

好的层次是非常重要的。显示问题的唯一来源应该是CSS，行为问题的唯一来源应该是JavaScript。在层次之间保持耦合可以让你的整个应用更加易于维护。

1. 解耦应用逻辑/事件处理程序

如果把事件处理和应用逻辑写在一个函数中，会让调试变得困难，并难以维护。较好的办法是将应用逻辑和事件处理程序分开，这样两者分别各自处理相关的信息。采用事件处理程序提取信息，然后调用应用逻辑的方式。

**性能优化**

因为JavaScript最初是一面解释型语言，执行速度比编译型语言低得多。Chrome引擎是一款内置优化引擎的浏览器，将JavaScript编译成本地代码。此后主流的浏览器都陆续实现了编译执行。

**优化作用域**

随着作用域链的增加，访问当前作用域链以外变量作用域的时间也在增加。而访问全局作用域比局部作用域要慢。所以减少访问作用域链的时间，就能提高性能。

1. **避免全局查找**

我们可以将函数中多次用到的全局变量存储为局部变量，这样就不用每次都访问全局的变量了。例如：

function Fun1() {

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

var num = document.getElementById("num").getAttribute("value"); //访问了10次全局变量document，时间复杂度O(n)

console.log(num);

}

}

function Fun2() {

var num = document.getElementById("num"); //只访问一次全局变量document，时间复杂度O(1)

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

console.log(num.getAttribute("value"));

}

}

1. **避免不必要的属性查找**

下面是JavaScript中常见的算法类型：

|  |  |  |
| --- | --- | --- |
| 标记 | 名称 | 描述 |
| O(1) | 常数 | 不管有多少值，执行的时间是恒定的。一般表示简单值和存储在变量中的值。 |
| O(log n) | 对数 | 总的执行时间和值的数量有关，但是要完成算法并不一定要获取每个值。例如：二分查找 |
| O(n) | 线性 | 总执行之间和值的数量直接相关。例如：遍历某个数组的所有元素。 |
| O(n2) | 平方 | 总执行时间和值的数量有关，每个值至少要获取n次。例如：插入排序 |

使用遍历和数组要比访问对象上的属性更有效率。对象上的任何属性查找都要比访问变量或者数组花费更多的时间，因为必须在原型链中对拥有该名称的属性进行一次搜索。

1. **优化循环**

循环是编程中最常见的，在JavaScript中也随处可见。优化循环是性能优化中很重要的一部分，由于它们会反复运行一段代码，从而自动增加执行时间。一个循环的基本步骤如下所示：

1. 减值迭代 ---大多数循环使用一个0开始，增加到某个特定值的迭代器。在很多情况下，从最大值开始，在循环中不断减值的迭代器更加高效。

function Fun5() {

var obj = {

count: 1000

}

for (var i = 0; i < obj.count; i++) { //每次循环都会判断一次，访问一次count属性

console.log(i);

}

}

function Fun6() {

var obj = {

count: 1000

}

for (var i = obj.count - 1; i >= 0; i--) { //只在初始化i时访问了一次count属性

console.log(i);

}

}

1. 简化终止条件---由于每次循环都会计算终止条件，所以必须保证它尽可能快。也就是说尽量避免属性访问等其他O(n)操作。

function Fun7() {

var obj = {

count: 1000

}

var i = obj.count - 1;

if (i > -1) {

do {

console.log(i);

}while(--i >= 0)

}

}

1. **最小化语句数**

JavaScript代码中语句数量也影响所执行的操作的速度。完成多个操作的单个语句要比完成单个操作的多个语句快。

我们可以将多个var合并到一个var语句中来声明。它们之间用逗号隔开。

function Fun8() {

var a = 0;

var b = 1;

var c = 2;

}

function Fun9() {

var a = 0,

b = 1,

c = 2;

}

我们使用迭代值的时候，尽可能合并语句。

function Fun10{

var arr = new Array();

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

var name = arr[i];

i++;

}

}

function Fun11{

var arr = new Array();

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

var name = arr[i++]; //将两行代码操作合并为一行

}

}

1. **使用innerHTML**

有两种方法可以在页面上创建DOM节点：使用诸如appendChild(),CreateElement()之类的DOM方法，以及使用innerHTML。对于小的更改而言两种差别不大，但如果是大的DOM修改，使用innerHTML就比使用标准的DOM方法要快。

需要注意的是，使用innerHTML要最小化调用它的次数，可以先构造一个HTML字符串一次性调用innerHTML。

1. **最小化访问HTMLCollection**

任何时候都要注意HTMLCollection，不管它是一个属性还是一个方法，都是在一个文档上进行一个查询，这个查询的开销很昂贵。最小化访问HTMLCollection可以改进脚本的性能。

编写JavaScript的时候，一定要知道何时返回HTMLCollection对象，这样你就可以最小化它的访问了，以下情况下会返回HTMLCollection对象：

* 进行了对getElementsByTagName，getElementsByClassName的调用
* 获取了元素childNodes属性
* 获取了元素attributes属性
* 访问了特殊的集合读document.forms,document.images等