**JavaScript**

## 基础概述

**1. 什么是JavaScript？**

JavaScript是一门脚本语言。但是我们的JavaScript既可以说他是一门脚本语言，又可以说他不是一门脚本语言。在ES5之前，JavaScript一直作为脚本语言存在的。但是到ES6，我们的JavaScript已经变成了一门通用编程语言。这门语言可用于 HTML 和 web，更可广泛用于服务器、PC、笔记本电脑、平板电脑和智能手机等设备。

**JavaScript是一门高端的，动态的，弱类型的编程语言，非常适合面向对象和函数式的编程风格。**JavaScript是早已超出了其"脚本语言"scripting-language,而成为一种集健壮性，高效性和通用性为一身的编程语言。JavaScript是网景(Netscape)公司创建，JavaScript是Sun Microsystem 公司现在的Oracle的注册商标，后来网景在浏览器大战中失败，把语言作为标准提交给了ECMA由于商标冲突使用ECMAScript来指代语言标准。

每种语言都有自己的开发平台，标准库或API函数。Javascript语言包括语法和针对文本，数组，日期和正则表达式的操作很少API，其它功能有"宿主环境",web浏览器提供。

**2 JavaScript和Java之间的关系**

首先要明确，JavaScript和Java没有任何的关系。事实上，我们的JavaScript最早叫做liveScript。但是因为当时的Java非常非常火，所以为了抱大腿，临时将liveScript改为了JavaScript；JavaScript诞生于1995年。最早事实上这门语言是用来做表单验证的。最初Netscape（翻译成中文：网景）公司发明。

**3 JavaScript和ECMAScript之间的关系**

ECMA：欧洲计算机制造商协会。它负责将JavaScript进行标准化。标准化出来之后的产物就是ECMAScript。

**4 JavaScript可以做什么**

① 表单验证

② 响应用户在网页上的操作

③ 对网页进行增加，删除，修改元素。（DOM）

④ 检测访问者的浏览器。（BOM）

**5 JavaScript的实现**

**① ECMAScript：**所谓ECMAScript，就是网景公司将JavaScript提交给欧洲计算机制造商协会之后，标准化出来的脚本语言的语法和定义。该

协会指定39号技术委员会（T39）来对这门语言进行标准化。T39的成员有网景公司的员工，也有微软等公司的员工组成。第一版ECMAScript出来后，版本号为ECMAScript-262。

**② BOM：**翻译成中文就是浏览器对象模型。BOM主要是用来控制用户浏览器本身。BOM到目前为止还缺少标准。（JavaScript由ECMA来定标准，H

TML和CSS由W3C来定制标准），目前WHATWG小组正在努力的为BOM来定制标准。BOM能够改变窗口大小，打开和关闭新窗口，获取浏览器版本，屏幕分辨率等。

**③DOM：**翻译成中文就是文档对象类型。DOM主要是用于控制网页里面的元素。例如：新增，修改，删除网页里面的某个元素。

**④事件处理模型：**主要就是讲JavaScript代码和事件进行绑定来响应用户的操作。

**⑤原生对象和内置对象：**主要就是JavaScript里面的内部类。

**6 JavaScript语言特点**

**① 脚本语言。**它是解释型运行。之前我们所学的C语言是编译型语言。所谓解释型，就是代码执行一条，我们就翻译一条。而编译型则会一次性将代码全部进行翻译。编译型的运行速度比解释性语言要快。

**② 基于对象：**JavaScript里面一切皆为对象。

**③ 弱类型语言：**声明变量的时候不需要声明数据类型

**④ 跨平台：**JavaScript脚本语言不依赖操作系统，仅需要得到浏览器的支持就可以了。目前市场上99%的浏览器都支持JavaScript。

**7 JavaScript版本**

目前最新的就是ECMAScript5，但是ES6（在ES6里面ECMAScript已经变成了一门通用的编程语言）已经被提出，但是很多浏览器还没有支持到ES6。

**JavaScript 脚本每个浏览器都有解释器。** 通常”开发者面板“有多选项卡，可以查看HTML文档结构，css样式，网络请求等。其中Javascript控制台，可以直接输入javascript代码并运行出结果，可以通过console.log(）来向控制台输出消息。

## JS核心技术

**1 赋值**

var length = 16; // Number 通过数字字面量赋值

var points = x \* 10; // Number 通过表达式字面量赋值

var lastName = "Johnson"; // String 通过字符串字面量赋值

var cars = ["Saab", "Volvo", "BMW"]; // Array 通过数组字面量赋值

var person = {firstName:"John", lastName:"Doe"}; // Object 通过对象字面量赋值

**2 Value = undefined**

在计算机程序中，经常会声明无值的变量。未使用值来声明的变量，其值实际上是 undefined。

在执行过以下语句后，变量 carname 的值将是 undefined：

var carname;

重新声明 JavaScript 变量

如果重新声明 JavaScript 变量，该变量的值不会丢失：

在以下两条语句执行后，变量 carname 的值依然是 "Volvo"：

var carname="Volvo";

var carname;

**3 JSON相关**

JavaScript Object Notation

JSON 是一种轻量级的数据交换格式。

JSON是独立的语言 \*

JSON 易于理解。

var person=

{

firstname : "John",

lastname : "Doe",

id : 5566

};

name=person.lastname;

name=person["lastname"];

var arr1 = new Array('a', 'b', 'c');

arr1.push(person);

eval("delete person.firstname");

delete person.firstname;

var text = '{ "sites" : [' +

'{ "name":"Runoob" , "url":"www.runoob.com" },' +

'{ "name":"Google" , "url":"www.google.com" },' +

'{ "name":"Taobao" , "url":"www.taobao.com" } ]}';

obj = JSON.parse(text);

**4 几种domReady**

(function()

{

//doSome();

})();

$(function (){ //doSome(); });

**5 typeof "John"**

**6 正则**

/e/.test("The best things in life are free!");

/e/.exec("The best things in life are free!");

/\*是否带有小数\*/

function isDecimal(strValue )

{

var objRegExp= /^\d+\.\d+$/;

return objRegExp.test(strValue);

}

/\*校验是否中文名称组成 \*/

function ischina(str)

{

var reg=/^[\u4E00-\u9FA5]{2,4}$/; /\*定义验证表达式\*/

return reg.test(str); /\*进行验证\*/

}

/\*校验是否全由8位数字组成 \*/

function isStudentNo(str)

{

var reg=/^[0-9]{8}$/; /\*定义验证表达式\*/

return reg.test(str); /\*进行验证\*/

}

/\*校验电话码格式 \*/

function isTelCode(str)

{

var reg= /^((0\d{2,3}-\d{7,8})|(1[3584]\d{9}))$/;

return reg.test(str);

}

/\*校验邮件地址是否合法 \*/

function IsEmail(str)

{

var reg=/^([a-zA-Z0-9\_-])+@([a-zA-Z0-9\_-])+(\.[a-zA-Z0-9\_-])+/;

return reg.test(str);

}

**7 Exception**

var txt="";

function message()

{

try {

adddlert("Welcome guest!");

} catch(err) {

txt="本页有一个错误。\n\n";

txt+="错误描述：" + err.message + "\n\n";

txt+="点击确定继续。\n\n";

alert(txt);

}

}

**8 调试**

var x = 15 \* 5;

debugger;

document.getElementbyId("demo").innerHTML = x;

**9 use strict**

使用 "use strict" 指令

"use strict" 指令在 JavaScript 1.8.5 (ECMAScript5) 中新增。

它不是一条语句，但是是一个字面量表达式，在 JavaScript 旧版本中会被忽略。

"use strict" 的目的是指定代码在严格条件下执行。

严格模式下你不能使用未声明的变量。

@消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

@消除代码运行的一些不安全之处，保证代码运行的安全；

@提高编译器效率，增加运行速度；

@为未来新版本的Javascript做好铺垫。

"use strict";

myFunction();

function myFunction()

{

y = 3.14;

}

**10 href="#"与href="javascript:void(0)"的区别**

# 包含了一个位置信息，默认的锚是#top 也就是网页的上端。

而javascript:void(0), 仅仅表示一个死链接。

在页面很长的时候会使用 # 来定位页面的具体位置，格式为：# + id。

如果你要定义一个死链接请使用 javascript:void(0) 。

**11 DOM**

var x=document.getElementById("intro");

var x=document.getElementById("main");

var y=x.getElementsByTagName("p");

var x=document.getElementsByClassName("intro");

**12 Event**

document.getElementById("myBtn").addEventListener("click", displayDate);

element.addEventListener("mouseover", myFunction);

element.addEventListener("click", mySecondFunction);

element.addEventListener("mouseout", myThirdFunction);

window.addEventListener("resize", function()

{

document.getElementById("demo").innerHTML = sometext;

});

element.removeEventListener("mousemove", myFunction);

var para=document.createElement("p");

var node=document.createTextNode("这是一个新段落。");

para.appendChild(node);

parent.removeChild(child);

**13 优点**

<1>.JavaScript减少网络传输。

在JavaScript这样的用户端脚本语言出现之前，传统的数据提交和验证工作均由用户端浏览器通过网络传输到服务器上进行。如果数据量很大，这对于网络和服务器的资源来说实在是一种无形的浪费。而使用JavaScript就可以在客户端进行数据验证。

<2>.JavaScript方便操纵HTML对象。

JavaScript可以方便地操纵各种页面中的对象，用户可以使用JavaScript来控制页面中各个元素的外观、状态甚至运行方式，JavaScript可以根据用户的需要“定制”浏览器，从而使网页更加友好。

<3>.JavaScript支持分布式运算。

JavaScript可以使多种任务仅在用户端就可以完成，而不需要网络和服务器的参与，从而支持分布式的运算和处理。

**14 缺点**

<1>.各浏览器厂商对JavaScript支持程度不同。

目前在互联网上有很多浏览器，如Firefox、Internet Explorer、Opera等，但每种浏览器支持JavaScript的程度是不一样的，不同的浏览器在浏览一个带有JavaScript脚本的主页时，由于对JavaScript的支持稍有不同，其效果会有一定的差距，有时甚至会显示不出来。

<2>.“Web安全性”对JavaScript一些功能牺牲。

当把JavaScript的一个设计目标设定为“Web安全性”时，就需要牺牲JavaScript的一些功能。因此，纯粹的JavaScript将不能打开、读写和保存用户计算机上的文件。其有权访问的唯一信息就是该JavaScript所嵌入的那个Web主页中的信息，简言之，JavaScript将只存在于它自己的小小世界—Web主页里。

**15 使用场景**

1） 表单校验

2）事件

## JS深度复制

**1 在谈javascript的浅复制和深复制**之前，我们有必要在来讨论下js的数据类型。我们都知道有Number，Boolean，String，Null，Undefined，Object五种类型。而Object又包含Function，Array和Object自身。前面的五种类型叫做基本类型，而Object是引用类型。

**2 js中基本类型的赋值为深复制，而引用类型的赋值为浅复制。**

浅复制：就是把数据的地址赋值给对应变量，而没有把具体的数据复制给变量，变量会随数据值的变化而变化。

深复制：就是把数据赋值给对应的变量，从而产生一个与源数据不相干的新数据(数据地址已变化)

var a = "dengkunming";

var a1 = a;

alert(a1);//dengkunming

a="abc";

alert(a1);//dengkunming;

var a = [0,1,2,3];

var a1= a;

alert(a1);//[0,1,2,3]

a[1]="变";

alert(a1);//[0,"变",2,3]

**3 浅复制不会随着存储数据地址的变化而变化，只会随着数据值的变化而变化。**

var a = {w1:2,w2:3}

var a1= a;

alert(a1);//{w1:2,w2:3}

var a={x1:7,x2:8}

alert(a1);//{w1:2,w2:3}

按照我们上面的理论来讲，这里是浅复制。a1应该随着a的变化而变化呀，可在这里为什么会事与愿违了？这就是引用类型惹的祸了。对象赋值其实都是引用传值，传递的是一个地址。那么实验五中的第四行其实就是把变量a指向了一个新的地址。而a1还是指向的原来那个地址，原来地址中的值没变，所以a1就不会变。所以请记住：浅复制不会随着存储数据地址的变化而变化，只会随着数据值的变化而变化。

**4 数组和对象的浅拷贝**

var a = [0,1,2,3];

var a1= a;

**5 数组的浅拷贝 和 深度拷贝**

**（1）js的slice函数**

arrayObj.slice(start, [end]) 返回一个数组的一段。（仍为数组）

var arr = ["One","Two","Three"];

var arrtoo = arr.slice(0,2);

arrtoo[1] = "set Map";

document.writeln("数组的原始值：" + arr + "<br />");//Export:数组的原始值：One,Two,Three

document.writeln("数组的新值：" + arrtoo + "<br />");//Export:数组的新值：One,set Map,Three

**（2）js的concat方法**

arrayObject.concat(arrayX,arrayX,......,arrayX)连接两个或多个数组。该方法不会改变现有的数组，而仅仅会返回被连接数组的一个副本。

var arr = ["One","Two","Three"];

var arrtooo = arr.concat();

arrtooo[1] = "set Map To";

document.writeln("数组的原始值：" + arr + "<br />");//Export:数组的原始值：One,Two,Three

document.writeln("数组的新值：" + arrtooo + "<br />");//Export:数组的新值：One,set Map To,Three

**6 对象的深度拷贝**

var obj =

{

name:'wsscat',

age:0

}

var obj2 = new Object();

obj2.name = obj.name;

obj2.age = obj.age

obj.name = 'autumns';

console.log(obj);//Object {name: "autumns", age: 0}

console.log(obj2);//Object {name: "wsscat", age: 0}

**7 通用 深度拷贝**

function clone(obj)

{

var o=[];

if(obj.constructor== Array)

{

o=obj.slice(0);

}

else

{

o={};

for(var i in obj)

{

o[i] = typeof obj[i] === "object" ? obj[i].clone() : obj[i];

}

}

return o;

}

**8 基本类型只能深度拷贝**

## JS继承

**1 JS作为面向对象的弱类型语言**，继承也是其非常强大的特性之一，由于所有的实例对象共享同一个prototype对象，那么从外界看起来，prototype对象就好像是实例对象的原型，而实例对象则好像"继承"了prototype对象一样。

**2 八种方法父类**

function Animal (name)

{

this.name = name || 'Animal';

this.sleep = function()

{

console.log(this.name + 'is asleep！');

}

}

Animal.prototype.eat = function(food)

{

console.log(this.name + 'eat:' + food);

};

**3 原型链继承,将父类的实例作为子类的原型**

**（1）代码**

function Cat(){ }

Cat.prototype = new Animal();

Cat.prototype.name = 'cat';

var cat = new Cat();

console.log(cat.name);

console.log(cat.eat('fish'));

console.log(cat.sleep());

console.log(cat instanceof Animal); //true

console.log(cat instanceof Cat); //true

**（2）特点**

非常纯粹的继承关系，实例是子类的实例，也是父类的实例；父类新增原型方法/原型属性，子类都能访问到；简单，易于实现

**（3）缺点**

@要想为子类新增属性和方法，必须要在new Animal()这样的语句之后执行，不能放到构造器中

@无法实现多继承

@来自原型对象的引用属性是所有实例共享的

@创建子类实例时，无法向父类构造函数传参

**4 构造继承,**

使用父类的构造函数来增强子类实例，等于是复制父类的实例属性给子类（没用到原型）

（1）代码

function Cat(name)

{

Animal.call(this);

this.name = name || 'Tom';

}

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true

（2）特点

@解决了1中，子类实例共享父类引用属性的问题

@创建子类实例时，可以向父类传递参数

@可以实现多继承（call多个父类对象）

（3）缺点

@实例并不是父类的实例，只是子类的实例

@只能继承父类的实例属性和方法，不能继承原型属性/方法

@无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

**5 实例继承，为父类实例添加新特性，作为子类实例返回**

（1）代码

function Cat(name)

{

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // false

（2）特点：不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

（3）缺点

@实例是父类的实例，不是子类的实例

@不支持多继承

**6 拷贝继承**

（1）代码

function Cat(name)

{

var animal = new Animal();

for(var p in animal)

{

Cat.prototype[p] = animal[p];

}

Cat.prototype.name = name || 'Tom';

}

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true

（2）特点：支持多继承

（3）缺点

@效率较低，内存占用高（因为要拷贝父类的属性）

@无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

**7 组合继承**，通过调用父类构造，继承父类的属性并保留传参的优点，然后通过将父类实例作为子类原型，实现函数复用

（1）代码

function Cat(name)

{

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // true

（2）特点

@弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法

@既是子类的实例，也是父类的实例

@不存在引用属性共享问题

@可传参

@函数可复用

（3）缺点：调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

**8 寄生组合继承**

（1）代码

function Cat(name)

{

Animal.call(this);

this.name = name || 'Tom';

}

(function()

{

var Super = function(){};

Super.prototype = Animal.prototype;

Cat.prototype = new Super();

})();

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); //true

（2）特点：堪称完美

（3）缺点：实现较为复杂

**9 apply方法**

Animal.apply(this);

**10 $.extends()，extend(dest,src1,src2,src3...);**

（1）扩展继承~带上 dest

var code1 = {"id":4,"name":"tom"};

var code2 = {"age":24};

var code3 = $.extend(code1,code2);

（2）为扩展$类本身.为类添加新的方法。 去掉 dest

$.extend

({

min: function(a, b)

{

return a < b ? a : b;

},

max: function(a, b)

{

return a > b ? a : b;

}

});

$.min(2, 3); // 2

$.max(4, 5); // 5

（3）$.fn.extend(object);给$对象添加方法。

jQuery.fn = jQuery.prototype =

{

　　　init: function( selector, context )

{}

};

$.fn.extend

({

alertWhileClick: function()

{

$(this).click(function()

{

alert($(this).val());

});

}

});

$("#input1").alertWhileClick();

**11 apply和 call和 prototype**

**（1）call方法:**

语法：call([thisObj[,arg1[, arg2[, [,.argN]]]]])

定义：调用一个对象的一个方法，以另一个对象替换当前对象。

说明： call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。 如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。

**（2）apply方法：**

语法：apply([thisObj[,argArray]])

定义：应用某一对象的一个方法，用另一个对象替换当前对象。

说明： 如果 argArray 不是一个有效的数组或者不是 arguments 对象，那么将导致一个 TypeError。 如果没有提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj， 并且无法被传递任何参数。

**（3）示例**

@示例1："劫持"别人的方法

var foo =

{

name:"mingming",

logName:function()

{

console.log(this.name);

}

}

var bar=

{

name:"xiaowang"

};

foo.logName.call(bar);//xiaowang

//此时foo中的logName方法将被bar引用 ，this指向了bar

@示例2:实现继承

function Animal(name)

{

this.name = name;

this.showName = function()

{

alert(this.name);

}

}

function Cat(name)

{

Animal.call(this, name);

}

var cat = new Cat("Black Cat");

cat.showName();

Animal.call(this) 的意思就是使用 Animal对象代替this对象，那么 Cat中不就有Animal的所有属性和方法了吗，Cat对象就能够直接调用Animal的方法以及属性了.

示例3：实际例子

有一个局部的fun方法，fun被作为普通函数调用时，fun内部的this指向了window，但我们往往是想让它指向该#test节点

window.id="window";

document.querySelector('#test').onclick = function()

{

console.log(this.id);//test

var fun = function()

{

console.log(this.id);

}

fun();//window

}

使用call,apply我们就可以轻松的解决这种问题了

window.id="window";

document.querySelector('#test').onclick = function()

{

console.log(this.id);//test

var fun = function()

{

console.log(this.id);

}

fun.call(this);//test

}

（4）对比：call 和 apply 都是为了改变某个函数运行时的 context 即上下文而存在的，换句话说，就是为了改变函数体内部 this 的指向。 call 和 apply二者的作用完全一样，只是接受参数的方式不太一样。 call 的第二个参数可以是任意类型，而apply的第二个参数必须是数组，也可以是arguments

## JS对象

**内置对象、Browser对象、DOM对象、HTML对象**

**1 String对象的属性和方法：**

prototype 能够为对象加入的属性和方法

window.String.prototype.name = 999; var sss=""; alert(sss.name);

length 返回字符串的字符长度

big() 把字符串中的文本变成大字体(<BIG>)

blink() 把字符串中的文本变成闪烁字体(<BLINK>)

bold() 把字符串中的文本变成黑字体(<B>)

fixed() 把字符串中的文本变成固定间距字体,即电报形式(<TT>)

fontcolor(color)设置字符串中文本的颜色(<FONT COLOR=>)

Fontsize(size) 把字符串中的文本变成指定大小(<FONTSIZE=>)

italics() 把字符串中的文本变成斜字体(<I>)

Link(url) 用来把字符串转换-HTML链接标记中(<A HREF=>)

small() 把字符串中的文本变成小字体(<SMALL>)

strike() 把字符串中的文本变成划掉字体(<STRIKE>)

sub() 把字符串中的文本变成下标(subscript)字体((SUB>)

sup() 把字符串中的文本变成上标(superscript)字体(<SUP>)

charAt(index) 返回指定索引处的字符

charCodeAt(index)返回一个整数,该整数表示String对象中指定位置处的字符的Unicode编

concat(string2)连接两条或多条字符串

fromCharCode(num1, num2, …, numN)获取指定的Unicode值并返回字符串

indexOf(searchString, startIndex) 返回字符串中第一个出现指定字符串的位置

lastlndexOf(searchString, startIndex) 返回字符串中最后一个出现指定字符串的位置

match(regex) 在字符串中查找指定值

replace(regex, newString)将字符串中的某些字符替换成其它字符

search(regex) 针对某执行值对字符串进行查找

slice(startIndex, endIndex)将部分字符抽出并在新的字符串中返回剩余部分

split(delimiter)将字符串分配为数组

substr(startIndex, length) 从startIndex取,取length个字符

substring(startIndex, endIndex) 从startIndex和endIndex之间的字符,不包括endIndex

toLowerCase() 把字符串中的文本变成小写

toUpperCase() 把字符串中的文本变成大写

toSource() 显示对象的源代码

valueOf() 返回字符串对象的原始值

**2 Array对象：**

constructor 所建立对象的函数参考

prototype 能够为对象加入的属性和方法

index For an array created by a regular expression match, the zero-based index of the match in the string.

input For an array created by a regular expression match, reflects the original string against which the regular expression was matched.

length 获取数组元素的个数,即最大下标加1

concat(array1,arrayn)将两个或两个以上的数组值连接起来,合并后返回结果

join(string) 将数组中元素合并为字符串,string为分隔符.如省略参数则直接合并,不再分隔

pop() 移除数组中的最后一个元素并返回该元素

push(value) 在数组的末尾加上一个或多个元素,并且返回新的数组长度值

reverse() 颠倒数组中元素的顺序,反向排列

shift() 移除数组中的第一个元素并返回该元素

slice(start, deleteCount, [item1[, item2[,...[,itemN]]]]) 返从一个数组中移除一个或多个元素,如果必要,在所移除元素的位置上插入新元素,返回所移除的元素

sort(compare Function) 在未指定排序号的情况下,按照元素的字母顺序排列,如果不是字符串类型则转换成字符串再排序,返回排序后的数组

splice() 为数组删除并添加新的元素

toSource() 显示对象的源代码

toString() 将数组所有元素返回一个字符串,其间用逗号分隔

unshift(value)为数组的开始部分加上一个或多个元素,并且返回该数组的新长度

valueOf() 返回数组对象的原始值

**3 Date对象：**

constructor 所建立对象的函数参考

prototype 能够为对象加入的属性和方法

getDay() 返回一周中的第几天(0-6)

getYear() 返回年份.2000年以前为2位,2000(包含)以后为4位

getFullYear() 返回完整的4位年份数

getMonth() 返回月份数(0-11)

getDate() 返回日(1-31)

getHours() 返回小时数(0-23)

getMinutes() 返回分钟(0-59)

getSeconds() 返回秒数(0-59)

getMilliseconds() 返回毫秒(0-999)

getTime() 返回从1970年1月1号0:0:0到现在一共花去的毫秒数

getTimezoneoffset() 返回时区偏差值,即格林威治平均时间(GMT)与运行脚本的计算机所处时区设置之间相差的分钟数)

parse(dateString) 返回在Date字符串中自从1970年1月1日00:00:00以来的毫秒数

setYear(yearInt) 设置年份.2位数或4位数

setFullYear(yearInt)设置年份.4位数

setMonth(monthInt) 设置月份(0-11)

setDate(dateInt) 设置日(1-31)

setHours(hourInt) 设置小时数(0-23)

setMinutes(minInt) 设置分钟数(0-59)

setSeconds(secInt) 设置秒数(0-59)

setMilliseconds(milliInt) 设置毫秒(0-999)

setTime(timeInt) 设置从1970年1月1日开始的时间.毫秒数

toLocaleString() 把Date对象的日期(一个数值)转变成一个字符串,使用所在计算机上配置使用的特定日期格式

toSource() 显示对象的源代码

toString() 将日期对象转换为字符串

valueOf() 返回日期对象的原始值

**4 Math对象是一个静态对象，换句话说：在使用Math对象，不需要创建实例。**

constructor 所建立对象的函数参考

prototype 能够为对象加入的属性和方法

E 欧拉常量,自然对数的底(约等于2.718)

LN2 2的自然对数(约等于0.693)

LN10 10的自然对数(约等于2.302)

LOG2E 以2为底的e的对数.(约等于1.442)

LOG10E 以10为底的e的对数(约等于0.434)

PI ∏的值(约等于3.14159)

SQRT1\_2 1/2(0.5)的平方根(即l除以2的平方根,约等于o.707)

SQRT2 2的平方根(约等于1.414)

abs(x) 返回数字的绝对值

acos(x) 返回数字的反余弦值

asin(x) 返回数字的反正弦值

atan(x) 返回位于-PI/2 和 PI/2 的反正切值

atan2(y,x) 返回（x,y）位于 -PI 到 PI 之间的角度

ceil(x) 返回 x 四舍五入后的最大整数

cos(x) 返回一个数字的余弦值

exp(x) 返回 E^x 值

floor(x) 返回 x 四舍五入后的最小整数

log(x) 返回底数为E的自然对数

max(x,y) 返回 x 和 y 之间较大的数

min(x,y) 返回 x 和 y 之间较小的数

pow(x,y) 返回 y^x 的值

random() 返回位于 0 到 1 之间的随机函数

round(x) 四舍五入后取整

sin(x) 返回数字的正弦值

sqrt(x) 返回数字的平方根

tan(x) 返回一个角度的正切值

toSource() 显示对象的源代码

valueOf() 返回数学对象的原始值

**5 Number数值对象:**

MAX\_VALUE The largest possible value a number in JavaScript can have 1.7976931348623157E+308

MIN\_VALUE The smallest possible value a number in JavaScript can have 5E-324

NaN Equal to a value that is not a number.

NEGATIVE\_INFINITY A value that is less than MIN\_VALUE.

POSOTIVE\_INFINITY A value that is greater than MAX\_VALUE.

prototype A static property of the Number object

toString Returns a string representing the specified object

valueOf() 返回数学对象的原始值

**6 Event 对象**

onabort 图像加载被中断

onblur 元素失去焦点

onchange 用户改变域的内容

onclick 鼠标点击某个对象

ondblclick 鼠标双击某个对象

onerror 当加载文档或图像时发生某个错误

onfocus 元素获得焦点

onkeydown 某个键盘的键被按下

onkeypress 某个键盘的键被按下或按住

onkeyup 某个键盘的键被松开

onload 某个页面或图像被完成加载

onmousedown 某个鼠标按键被按下

onmousemove 鼠标被移动

onmouseout 鼠标从某元素移开

onmouseover 鼠标被移到某元素之上

onmouseup 某个鼠标按键被松开

onreset 重置按钮被点击

onresize 窗口或框架被调整尺寸

onselect 文本被选定

onsubmit 提交按钮被点击

onunload 用户退出页面

**7 RegExp正则表达式**

[abc] 查找方括号之间的任何字符。

[^abc] 查找任何不在方括号之间的字符。

[0-9] 查找任何从 0 至 9 的数字。

[a-z] 查找任何从小写 a 到小写 z 的字符。

[A-Z] 查找任何从大写 A 到大写 Z 的字符。

[A-z] 查找任何从大写 A 到小写 z 的字符。

[adgk] 查找给定集合内的任何字符。

[^adgk] 查找给定集合外的任何字符。

(red|blue|green) 查找任何指定的选项。

\w 查找单词字符。

\W 查找非单词字符。

\d 查找数字。

\D 查找非数字字符。

\s 查找空白字符。

\S 查找非空白字符。

\b 匹配单词边界。

\B 匹配非单词边界。

\0 查找 NUL 字符。

\n 查找换行符。

\f 查找换页符。

\r 查找回车符。

\t 查找制表符。

\v 查找垂直制表符。

n+ 匹配任何包含至少一个 n 的字符串。

n\* 匹配任何包含零个或多个 n 的字符串。

n? 匹配任何包含零个或一个 n 的字符串。

n{X} 匹配包含 X 个 n 的序列的字符串。

n{X,Y} 匹配包含 X 至 Y 个 n 的序列的字符串。

n{X,} 匹配包含至少 X 个 n 的序列的字符串。

n$ 匹配任何结尾为 n 的字符串。

^n 匹配任何开头为 n 的字符串。

global RegExp 对象是否具有标志 g。

ignoreCase RegExp 对象是否具有标志 i。

lastIndex 一个整数，标示开始下一次匹配的字符位置。

multiline RegExp 对象是否具有标志 m。

source 正则表达式的源文本。

compile 编译正则表达式。

exec 检索字符串中指定的值。返回找到的值，并确定其位置。

test 检索字符串中指定的值。返回 true 或 false。

search 检索与正则表达式相匹配的值。

match 找到一个或多个正则表达式的匹配。

replace 替换与正则表达式匹配的子串。

split 把字符串分割为字符串数组。

**8 Window对象，顶级对象，下面包含几个子对象。**

closed 返回窗口是否已被关闭。

defaultStatus 设置或返回窗口状态栏中的默认文本。

document 对 Document 对象的只读引用。请参阅 Document 对象。

history 对 History 对象的只读引用。请参数 History 对象。

innerheight 返回窗口的文档显示区的高度。

innerwidth 返回窗口的文档显示区的宽度。

length 设置或返回窗口中的框架数量。

location 用于窗口或框架的 Location 对象。请参阅 Location 对象。

name 设置或返回窗口的名称。

Navigator 对 Navigator 对象的只读引用。请参数 Navigator 对象。

opener 返回对创建此窗口的窗口的引用。

outerheight 返回窗口的外部高度。

outerwidth 返回窗口的外部宽度。

pageXOffset 设置或返回当前页面相对于窗口显示区左上角的 X 位置。

pageYOffset 设置或返回当前页面相对于窗口显示区左上角的 Y 位置。

parent 返回父窗口。

Screen 对 Screen 对象的只读引用。请参数 Screen 对象。

self 返回对当前窗口的引用。等价于 Window 属性。

status 设置窗口状态栏的文本。

top 返回最顶层的先辈窗口。

window window 属性等价于 self 属性，它包含了对窗口自身的引用。

screenLeft

screenTop

screenX

screenY

alert() 显示带有一段消息和一个确认按钮的警告框。

blur() 把键盘焦点从顶层窗口移开。

clearInterval() 取消由 setInterval() 设置的 timeout。

clearTimeout() 取消由 setTimeout() 方法设置的 timeout。

close() 关闭浏览器窗口。

confirm() 显示带有一段消息以及确认按钮和取消按钮的对话框。

createPopup() 创建一个 pop-up 窗口。

focus() 把键盘焦点给予一个窗口。

moveBy() 可相对窗口的当前坐标把它移动指定的像素。

moveTo() 把窗口的左上角移动到一个指定的坐标。

open() 打开一个新的浏览器窗口或查找一个已命名的窗口。

print() 打印当前窗口的内容。

prompt() 显示可提示用户输入的对话框。

resizeBy() 按照指定的像素调整窗口的大小。

resizeTo() 把窗口的大小调整到指定的宽度和高度。

scrollBy() 按照指定的像素值来滚动内容。

scrollTo() 把内容滚动到指定的坐标。

setInterval() 按照指定的周期（以毫秒计）来调用函数或计算表达式。

setTimeout() 在指定的毫秒数后调用函数或计算表达式。

**9 Navigator**

appCodeName 返回浏览器的代码名。

appMinorVersion 返回浏览器的次级版本。

appName 返回浏览器的名称。

appVersion 返回浏览器的平台和版本信息。

browserLanguage 返回当前浏览器的语言。

cookieEnabled 返回指明浏览器中是否启用 cookie 的布尔值。

cpuClass 返回浏览器系统的 CPU 等级。

onLine 返回指明系统是否处于脱机模式的布尔值。

platform 返回运行浏览器的操作系统平台。

systemLanguage 返回 OS 使用的默认语言。

userAgent 返回由客户机发送服务器的 user-agent 头部的值。

userLanguage 返回 OS 的自然语言设置。

javaEnabled() 规定浏览器是否启用 Java。

taintEnabled() 规定浏览器是否启用数据污点 (data tainting)。

**10 Screen**

availHeight 返回显示屏幕的高度 (除 Windows 任务栏之外)。

availWidth 返回显示屏幕的宽度 (除 Windows 任务栏之外)。

bufferDepth 设置或返回调色板的比特深度。

colorDepth 返回目标设备或缓冲器上的调色板的比特深度。

deviceXDPI 返回显示屏幕的每英寸水平点数。

deviceYDPI 返回显示屏幕的每英寸垂直点数。

fontSmoothingEnabled 返回用户是否在显示控制面板中启用了字体平滑。

height 返回显示屏幕的高度。

logicalXDPI 返回显示屏幕每英寸的水平方向的常规点数。

logicalYDPI 返回显示屏幕每英寸的垂直方向的常规点数。

pixelDepth 返回显示屏幕的颜色分辨率（比特每像素）。

updateInterval 设置或返回屏幕的刷新率。

width 返回显示器屏幕的宽度。

**11 History**

length 返回浏览器历史列表中的 URL 数量。

back() 加载 history 列表中的前一个 URL。

forward() 加载 history 列表中的下一个 URL。

go() 加载 history 列表中的某个具体页面。

**12 Location**

hash 设置或返回从井号 (#) 开始的 URL（锚）。

host 设置或返回主机名和当前 URL 的端口号。

hostname 设置或返回当前 URL 的主机名。

href 设置或返回完整的 URL。

pathname 设置或返回当前 URL 的路径部分。

port 设置或返回当前 URL 的端口号。

protocol 设置或返回当前 URL 的协议。

search 设置或返回从问号 (?) 开始的 URL（查询部分）。

assign() 加载新的文档。

reload() 重新加载当前文档。

replace() 用新的文档替换当前文档。

**13 console 对象**

.info(); // 以->信息方式\_输出参数对象

.warn(); // 以->警告方式\_输出参数对象

.error(); // 以->错误方式\_输出参数对象

.log(); // 以->日志方式\_输出参数对象

.table(); // 以->表格方式\_输出参数对象

.exception(); // 以->异常方式\_输出参数对象.异常

.debug(); // 以->排查方式\_输出参数对象.排查出来的[错误,漏洞]

.assert(); // 以->真假方式\_输出参数对象.真假的结果[假:断言失败,真:输出响应内容]

.dir(); // 以->目录方式\_输出参数对象.所有[属性，方法]

.dirxml(); // 以->内容方式\_输出参数对象.[html/xml]标签内容

.count(); // 以->计算方式\_输出所在位置被执行的次数

.trace(); // 以->记录方式\_输出参数对象=>函数.被调用的记录

.group(); // 以->分组方式\_输出参数对象

.groupCollapsed(); // 以->分组方式\_输出参数对象,与 group() 相同,不过默认是折叠显示

.groupEnd(); // 表明分组结束

.time(); // 开始计时

.timeEnd(); // 结束计时

.profile(); // 开始分析性能

.profileEnd(); // 结束分析性能

.timeStamp(); // 时间戳

.clear(); // 清空 console 中所有的信息

console.log(); --日志

console.info(); --信息

console.error(); --错误

console.warn(); --警告

console.debug(); --漏洞

**14 DOM Document**

all[] 提供对文档中所有 HTML 元素的访问。

anchors[] 返回对文档中所有 Anchor 对象的引用。

applets 返回对文档中所有 Applet 对象的引用。

forms[] 返回对文档中所有 Form 对象引用。

images[] 返回对文档中所有 Image 对象引用。

links[] 返回对文档中所有 Area 和 Link 对象引用。

body 提供对 <body> 元素的直接访问。

对于定义了框架集的文档，该属性引用最外层的 <frameset>。

cookie 设置或返回与当前文档有关的所有 cookie。

domain 返回当前文档的域名。

lastModified 返回文档被最后修改的日期和时间。

referrer 返回载入当前文档的文档的 URL。

title 返回当前文档的标题。

URL 返回当前文档的 URL。

close() 关闭用 document.open() 方法打开的输出流，并显示选定的数据。

getElementById() 返回对拥有指定 id 的第一个对象的引用。

getElementsByName() 返回带有指定名称的对象集合。

getElementsByTagName() 返回带有指定标签名的对象集合。

open() 打开一个流，以收集来自任何 document.write() 或 document.writeln() 方法的输出。

write() 向文档写 HTML 表达式 或 JavaScript 代码。

writeln() 等同于 write() 方法，不同的是在每个表达式之后写一个换行符。

**15 DOM Element**

element.accessKey 设置或返回元素的快捷键。

element.appendChild() 向元素添加新的子节点，作为最后一个子节点。

element.attributes 返回元素属性的 NamedNodeMap。

element.childNodes 返回元素子节点的 NodeList。

element.className 设置或返回元素的 class 属性。

element.clientHeight 返回元素的可见高度。

element.clientWidth 返回元素的可见宽度。

element.cloneNode() 克隆元素。

element.compareDocumentPosition() 比较两个元素的文档位置。

element.contentEditable 设置或返回元素的文本方向。

element.dir 设置或返回元素的文本方向。

element.firstChild 返回元素的首个子。

element.getAttribute() 返回元素节点的指定属性值。

element.getAttributeNode() 返回指定的属性节点。

element.getElementsByTagName() 返回拥有指定标签名的所有子元素的集合。

element.getFeature() 返回实现了指定特性的 API 的某个对象。

element.getUserData() 返回关联元素上键的对象。

element.hasAttribute() 如果元素拥有指定属性，则返回true，否则返回 false。

element.hasAttributes() 如果元素拥有属性，则返回 true，否则返回 false。

element.hasChildNodes() 如果元素拥有子节点，则返回 true，否则 false。

element.id 设置或返回元素的 id。

element.innerHTML 设置或返回元素的内容。

element.insertBefore() 在指定的已有的子节点之前插入新节点。

element.isContentEditable 设置或返回元素的内容。

element.isDefaultNamespace() 如果指定的 namespaceURI 是默认的，则返回 true，否则返回 false。

element.isEqualNode() 检查两个元素是否相等。

element.isSameNode() 检查两个元素是否是相同的节点。

element.isSupported() 如果元素支持指定特性，则返回 true。

element.lang 设置或返回元素的语言代码。

element.lastChild 返回元素的最后一个子元素。

element.namespaceURI 返回元素的 namespace URI。

element.nextSibling 返回位于相同节点树层级的下一个节点。

element.nodeName 返回元素的名称。

element.nodeType 返回元素的节点类型。

element.nodeValue 设置或返回元素值。

element.normalize() 合并元素中相邻的文本节点，并移除空的文本节点。

element.offsetHeight 返回元素的高度。

element.offsetWidth 返回元素的宽度。

element.offsetLeft 返回元素的水平偏移位置。

element.offsetParent 返回元素的偏移容器。

element.offsetTop 返回元素的垂直偏移位置。

element.ownerDocument 返回元素的根元素（文档对象）。

element.parentNode 返回元素的父节点。

element.previousSibling 返回位于相同节点树层级的前一个元素。

element.removeAttribute() 从元素中移除指定属性。

element.removeAttributeNode() 移除指定的属性节点，并返回被移除的节点。

element.removeChild() 从元素中移除子节点。

element.replaceChild() 替换元素中的子节点。

element.scrollHeight 返回元素的整体高度。

element.scrollLeft 返回元素左边缘与视图之间的距离。

element.scrollTop 返回元素上边缘与视图之间的距离。

element.scrollWidth 返回元素的整体宽度。

element.setAttribute() 把指定属性设置或更改为指定值。

element.setAttributeNode() 设置或更改指定属性节点。

element.setIdAttribute()

element.setIdAttributeNode()

element.setUserData() 把对象关联到元素上的键。

element.style 设置或返回元素的 style 属性。

element.tabIndex 设置或返回元素的 tab 键控制次序。

element.tagName 返回元素的标签名。

element.textContent 设置或返回节点及其后代的文本内容。

element.title 设置或返回元素的 title 属性。

element.toString() 把元素转换为字符串。

nodelist.item() 返回 NodeList 中位于指定下标的节点。

nodelist.length 返回 NodeList 中的节点数。

**16 DOM Attribute**

attr.isId 如果属性是 id 类型，则返回 true，否则返回 false。

attr.name 返回属性的名称。

attr.value 设置或返回属性的值。

attr.specified 如果已指定属性，则返回 true，否则返回 false。

nodemap.getNamedItem() 从 NamedNodeMap 返回指定的属性节点。

nodemap.item() 返回 NamedNodeMap 中位于指定下标的节点。

nodemap.length 返回 NamedNodeMap 中的节点数。

nodemap.removeNamedItem() 移除指定的属性节点。

nodemap.setNamedItem() 设置指定的属性节点（通过名称）。

attr.appendChild() 属性没有子节点。

attr.attributes 属性没有属性。

attr.baseURI 使用 document.baseURI 代替。

attr.childNodes 属性没有子节点。

attr.cloneNode() 使用 attr.value 代替。

attr.firstChild 属性没有子节点。

attr.hasAttributes() 属性没有属性。

attr.hasChildNodes 属性没有子节点。

attr.insertBefore() 属性没有子节点。

attr.isEqualNode() 没有意义。

attr.isSameNode() 没有意义。

attr.isSupported() 始终为 true。

attr.lastChild 属性没有子节点。

attr.nextSibling 属性没有同级节点。

attr.nodeName 使用 attr.name 代替。

attr.nodeType 始终为 2 (ATTRIBUTE\_NODE)。

attr.nodeValue 使用 attr.value 代替。

attr.normalize() 属性无法被正常化。

attr.ownerDocument 始终是您的 HTML 文档。

attr.ownerElement 这是您用来访问该属性的 HTML 元素。

attr.parentNode 这是您用来访问该属性的 HTML 元素。

attr.previousSibling 属性没有同级节点。

attr.removeChild 属性没有子节点。

attr.replaceChild 属性没有子节点。

attr.textContent 使用 attr.value 代替。

**17 DOM Event**

onabort 图像的加载被中断。

onblur 元素失去焦点。

onchange 域的内容被改变。

onclick 当用户点击某个对象时调用的事件句柄。

ondblclick 当用户双击某个对象时调用的事件句柄。

onerror 在加载文档或图像时发生错误。

onfocus 元素获得焦点。

onkeydown 某个键盘按键被按下。

onkeypress 某个键盘按键被按下并松开。

onkeyup 某个键盘按键被松开。

onload 一张页面或一幅图像完成加载。

onmousedown 鼠标按钮被按下。

onmousemove 鼠标被移动。

onmouseout 鼠标从某元素移开。

onmouseover 鼠标移到某元素之上。

onmouseup 鼠标按键被松开。

onreset 重置按钮被点击。

onresize 窗口或框架被重新调整大小。

onselect 文本被选中。

onsubmit 确认按钮被点击。

onunload 用户退出页面。

**18 a标签**

accessKey 设置或返回访问一个链接的快捷键。

charset 设置或返回被链接资源的字符集。

coords 设置或返回逗号分隔列表，包含了图像映射中链接的坐标。

href 设置或返回被链接资源的 URL。

hreflang 设置或返回被链接资源的语言代码。

id 设置或返回一个链接的 id。

innerHTML 设置或返回一个链接的内容。

name 设置或返回一个链接的名称。

rel 设置或返回当前文档与目标 URL 之间的关系。

rev 设置或返回目标 URL 与之间当前文档的关系。

shape 设置或返回图像映射中某个链接的形状。

tabIndex 设置或返回某个链接的 Tab 键控制次序。

target 设置或返回在何处打开链接。

type 设置或返回被链接资源的 MIME 类型。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

**19 body**

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

id 设置或返回 body 的 id。

lang 设置或返回元素的语言代码。

title 设置或返回元素的咨询性的标题。

**20 button**

accessKey 设置或返回访问某个按钮的快捷键。

disabled 设置或返回是否禁用按钮。

form 返回对包含按钮的表单的引用。

id 设置或返回按钮的 id。

name 设置或返回按钮的名称。

tabIndex 设置或返回按钮的 Tab 键控制次序。

type 返回按钮的表单类型。

value 设置或返回显示在按钮上的文本。

**21 Dialog**

open 设置或返回对话是否是打开的。

returnValue 设置或返回对话的返回值。

close() 关闭对话。

show() 显示对话。

showModal() 显示对话，并使其成为 top-most 型对话。

**22 form**

acceptCharset 服务器可接受的字符集。

action 设置或返回表单的 action 属性。

enctype 设置或返回表单用来编码内容的 MIME 类型。

id 设置或返回表单的 id。

length 返回表单中的元素数目。

method 设置或返回将数据发送到服务器的 HTTP 方法。

name 设置或返回表单的名称。

target 设置或返回表单提交结果的 Frame 或 Window 名。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

reset() 把表单的所有输入元素重置为它们的默认值。

submit() 提交表单。

onreset 在重置表单元素之前调用。

onsubmit 在提交表单之前调用。

**23 IFrame**

align 根据周围的文字排列 iframe。

contentDocument 容纳框架的内容的文档。

frameBorder 设置或返回是否显示 iframe 周围的边框。

height 设置或返回 iframe 的高度。

id 设置或返回 iframe 的 id。

longDesc 设置或返回描述 iframe 内容的文档的 URL。

marginHeight 设置或返回 iframe 的顶部和底部的页空白。

marginWidth 设置或返回 iframe 的左侧和右侧的页空白。

name 设置或返回 iframe 的名称。

scrolling 设置或返回 iframe 是否可拥有滚动条。

src 设置或返回应载入 iframe 中的文档的 URL。

width 设置或返回 iframe 的宽度。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

**24 Img**

align 设置或返回与内联内容的对齐方式。

alt 设置或返回无法显示图像时的替代文本。

border 设置或返回图像周围的边框。

complete 返回浏览器是否已完成对图像的加载。

height 设置或返回图像的高度。

hspace 设置或返回图像左侧和右侧的空白。

id 设置或返回图像的 id。

isMap 返回图像是否是服务器端的图像映射。

longDesc 设置或返回指向包含图像描述的文档的 URL。

lowsrc 设置或返回指向图像的低分辨率版本的 URL。

name 设置或返回图像的名称。

src 设置或返回图像的 URL。

useMap 设置或返回客户端图像映射的 usemap 属性的值。

vspace 设置或返回图像的顶部和底部的空白。

width 设置或返回图像的宽度。

className 设置或返回元素的 class 属性。

title 设置或返回元素的 title。

onabort 当用户放弃图像的装载时调用的事件句柄。

onerror 在装载图像的过程中发生错误时调用的事件句柄。

onload 当图像装载完毕时调用的事件句柄。

**25 input button**

accessKey 设置或返回访问按钮的快捷键。

alt 设置或返回当浏览器无法显示按钮时供显示的替代文本。

disabled 设置或返回是否禁用按钮。

form 返回对包含该按钮的表单对象的引用。

id 设置或返回按钮的 id。

name 设置或返回按钮的名称。

tabIndex 设置或返回按钮的 tab 键控制次序。

type 返回按钮的表单元素类型。

value 设置或返回在按钮上显示的文本。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

blur() 把焦点从元素上移开。

click() 在某个按钮上模拟一次鼠标单击。

focus() 为某个按钮赋予焦点。

**26 input checkbox**

accessKey 设置或返回访问 checkbox 的快捷键。

alt 设置或返回不支持 checkbox 时显示的替代文本。

checked 设置或返回 checkbox 是否应被选中。

defaultChecked 返回 checked 属性的默认值。

disabled 设置或返回 checkbox 是否应被禁用。

form 返回对包含 checkbox 的表单的引用。

id 设置或返回 checkbox 的 id。

name 设置或返回 checkbox 的名称。

tabIndex 设置或返回 checkbox 的 tab 键控制次序。

type 返回 checkbox 的表单元素类型。

value 设置或返回 checkbox 的 value 属性的值

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

blur() 从 checkbox 上移开焦点。

click() 模拟在 checkbox 中的一次鼠标点击。

focus() 为 checkbox 赋予焦点。

**27 Select**

disabled 设置或返回是否应禁用下拉列表。

form 返回对包含下拉列表的表单的引用。

id 设置或返回下拉列表的 id。

length 返回下拉列表中的选项数目。

multiple 设置或返回是否选择多个项目。

name 设置或返回下拉列表的名称。

selectedIndex 设置或返回下拉列表中被选项目的索引号。

size 设置或返回下拉列表中的可见行数。

tabIndex 设置或返回下拉列表的 tab 键控制次序。

type 返回下拉列表的表单类型。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

add() 向下拉列表添加一个选项。

blur() 从下拉列表移开焦点。

focus() 在下拉列表上设置焦点。

remove() 从下拉列表中删除一个选项。

onchange 当改变选择时调用的事件句柄。

**28 Table**

cells[] 返回包含表格中所有单元格的一个数组。

rows[] 返回包含表格中所有行的一个数组。

tBodies[] 返回包含表格中所有 tbody 的一个数组。

align 表在文档中的水平对齐方式。（已废弃）

bgColor 表的背景颜色。（已废弃）

border 设置或返回表格边框的宽度。

caption 对表格的 <caption> 元素的引用。

cellPadding 设置或返回单元格内容和单元格边框之间的空白量。

cellSpacing 设置或返回在表格中的单元格之间的空白量。

frame 设置或返回表格的外部边框。

id 设置或返回表格的 id。

rules 设置或返回表格的内部边框（行线）。

summary 设置或返回对表格的描述（概述）。

tFoot 返回表格的 TFoot 对象。如果不存在该元素，则为 null。

tHead 返回表格的 THead 对象。如果不存在该元素，则为 null。

width 设置或返回表格的宽度。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

createCaption() 为表格创建一个 caption 元素。

createTFoot() 在表格中创建一个空的 tFoot 元素。

createTHead() 在表格中创建一个空的 tHead 元素。

deleteCaption() 从表格删除 caption 元素以及其内容。

deleteRow() 从表格删除一行。

deleteTFoot() 从表格删除 tFoot 元素及其内容。

deleteTHead() 从表格删除 tHead 元素及其内容。

insertRow() 在表格中插入一个新行。

**29 Textarea**

accessKey 设置或返回访问 textarea 的键盘快捷键。

cols 设置或返回 textarea 的宽度。

defaultValue 设置或返回文本框中的初始内容。

disabled 设置或返回 textarea 是否应当被禁用。

form 返回对包含该 textarea 的表单对象的引用。

id 设置或返回某个 textarea 的 id。

name 设置或返回 textarea 的名称。

readOnly 设置或返回 textarea 是否应当是只读的。

rows 设置或返回 textarea 的高度。

tabIndex 设置或返回 textarea 的 tab 键控制次序。

type 返回该文本框的表单类型。

value 设置或返回在 textarea 中的文本。

className 设置或返回元素的 class 属性。

dir 设置或返回文本的方向。

lang 设置或返回元素的语言代码。

title 设置或返回元素的 title 属性。

blur() 从 textarea 移开焦点。

focus() 在 textarea 上设置焦点。

select() 选择 textarea 中的文本。

onchange 当输入值改变时调用的事件句柄

## JS框架对比

**1 Dojo**

**（1）Dojo 是目前最为强大的JS框架**，它在自己的 Wiki 上给自己下了一个定义，Dojo 是一个用 JavaScript 编写的开源的DHTML工具箱。Dojo 很想做一个“大一统”的工具箱，不仅仅是浏览器层面的，野心还是很大的。Dojo 包括 Ajax、Browser、Event、Widget 等跨浏览器 API，包括了 JS 本身的语言扩展，以及各个方面的工具类库，和比较完善的 UI 组件库，也被广泛应用在很多项目中，他的 UI 组件的特点是通过给 HTML 标签增加 TAG 的方式进行扩展，而不是通过写 JS 来生成，Dojo 的 API 模仿 Java 类库的组织方式。用 Dojo 写 Web OS 可谓非常方便。Dojo 现在已经 4.0 了，Dojo 强大的地方在于界面和特效的封装，可以让开发者快速构建一些兼容标准的界面。

**（2）优点：**库相当完善，发展时间也比较长，功能强大，据说利用 Dojo 的 io.bind() 可以实现 comet 看见其功能强大非一般，得到 IBM 和 SUN 的支持。

**（3）缺点：**文件体积比较大，200多KB，初次下载相当慢，此外，Dojo 的类库使用显得不是那么易用，JS语法增强方面不如 Prototype

**2 Prototype**

（1）它是一个非常优雅的 JS 库，定义了 JS 的面向对象扩展，DOM 操作API，事件等等，以 Prototype 为核心，形成了一个外围的各种各样的 JS 扩展库，是相当有前途的 JS 底层框架，值得推荐，感觉也是现实中应用最广的库类（RoR 集成的 AJAX JS 库），之上还有 Scriptaculous 实现一些JS组件功能和效果。

（2）优点：基本底层，易学易用，甚至是其他一些js特效开发包的底层，体积算是最小的了。

（3）缺点：如果说缺点，可能就是功能是他的弱项

**3 dScriptaculous**

（1）Scriptaculous 是基于prototype.js 框架的 JS 效果。包含了 6 个 js 文件，不同的文件对应不同的 js 效果，所以说，如果底层用 prototype 的话，做js效果用 Scriptaculous 那是再合适不过的了，连大名鼎鼎的 digg 都在用他，可见不一般

（2）优点：基于prototype 是最大的优点，由于使用 prototype 的广泛性，无疑对用户书锦上添花。

（3）缺点：刚刚兴起，需要时间的磨练

**4 Yui-ext**

（1）基于 Yahoo UI 的扩展包 yui-ext 是具有 CS 风格的 Web 用户界面组件，能实现复杂的 Layout 布局，界面效果可以和 backbase 媲美，而且使用纯 javascript 代码开发。真正的可编辑的表格 Edit Grid，支持 XML 和 Json 数据类型，直接可以迁入 Grid。许多组件实现了对数据源的支持，例如动态的布局，可编辑的表格控件，动态加载的 Tree 控件、动态拖拽效果等等。从 1.0 beta 版开始同 Jquery 合作，推出基于 jQuery 的 Ext 1.0，提供了更多有趣的功能。

（2）优点：结构化，类似于 java 的结构，清晰明了，底层用到了 Jquery 的一些函数，使整合使用有了选择，最重要的一点是界面太让让人震撼了。

（3）缺点：太过复杂，整个界面的构造过于复杂。

**5 Jquery**

（1）JQuery 是一款同 prototype 一样优秀 js 开发库类，特别是对 css 和 XPath 的支持，使我们写 js 变得更加方便！如果你不是个 js 高手又想写出优秀的 js 效果，那么 JQuery 可以帮你达到目的！并且简介的语法和高的效率一直是jQuery追求的目标。

（2）优点：注重简介和高效，js 效果有 yui-ext 的选择，因为 yui-ext 重用了很多 jQuery 的函数

（3）缺点：据说太嫩，历史不悠久。

**6 Mochikit**

（1）MochiKit 自称为一个轻量级的 js 框架。MochiKit 主要受到 Python 和 Python 标准库提供的很多便利之处的启发，另外还缓解了浏览器版本之间的不一致性。其中的 MochiKit.DOM 尤其方便，能够以比原始 JavaScript 更友好的方式处理 DOM 对象。MochiKit.DOM 大部分都是针对 XHTML 文档定制的，如果与 MochiKit 和 Ajax 结合在一起，使用 XHTML 包装的微格式尤其方便。Mochikit 可以直接对字符串或者数字格式化输出，比较实用和方便。它还有自己的 js 代码解释器

（2）优点：MochiKit.DOM 这部分很实用，简介也是很突出的

（3）缺点：轻量级的缺点

**7 Mootools**

（1）MooTools 是一个简洁，模块化，面向对象的 JavaScript 框架。它能够帮助你更快，更简单地编写可扩展和兼容性强的 JavaScript 代码。Mootools 跟 prototypejs 相类似，语法几乎一样。但它提供的功能要比 prototypejs 多，而且更强大。比如增加了动画特效、拖放操作等等。

（2）优点：可以定制自己所需要的功能，可以说是 prototypejs 的增强版。

（3）缺点：不大不小，具体应用具体分析。

**8 Moo.fx**

（1）Moo.fx是 一个超级轻量级的 javascript 特效库（7k），能够与 prototype.js 或mootools 框架一起使用。它非常快、易于使用、跨浏览器、符合标准，提供控制和修改任何 HTML 元素的 CSS 属性，包括颜色。它内置检查器能够防止用户通过多次或疯狂点击来破坏效果。moo.fx整体采用模块化设计，所以可以在它的基础上开发你需要的任何特效。

（2）优点：小块头有大能耐

（3）缺点：这么小了，已经不错了

## Json

**1 JSON:** JavaScript Object Notation(JavaScript 对象表示法)；JSON 是存储和交换文本信息的语法。类似 XML；JSON 比 XML 更小、更快，更易解析。JSON 文件的文件类型是 ".json"。JSON 文本的 MIME 类型是 "application/json"

**2 JSON 特点：**

（1）JavaScript 对象表示法（JavaScript Object Notation）

（2）JSON 是轻量级的文本数据交换格式

（3）JSON 独立于语言

（4）JSON 具有自我描述性，更易理解

（5）JSON 使用 Javascript语法来描述数据对象，但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。 目前非常多的动态（PHP，JSP，.NET）编程语言都支持JSON。

**3 常见代码**

**(1)表达式**

{ "flag":true }

{ "runoob":null }

var sites =

[

{ "name":"runoob" , "url":"www.runoob.com" },

{ "name":"google" , "url":"www.google.com" },

{ "name":"微博" , "url":"www.weibo.com" }

];

x = myObj.name;

x = myObj["name"];

**（2）循环数组**

var myObj = { "name":"runoob", "alexa":10000, "site":null };

for (x in myObj)

{

document.getElementById("demo").innerHTML += x + "<br>";

}

**（3）删除元**素

delete myObj.sites.site1;

delete myObj.sites["site1"]

delete myObj.sites[1];

**（4）JSON.parse函数**

var obj = JSON.parse('{ "name":"runoob", "alexa":10000, "site":"www.runoob.com" }');

主流浏览器都支持 JSON.parse() 函数：Firefox 3.5；Internet Explorer 8；Chrome；Opera 10

**（5）JSON.stringify函数**

var obj = { "name":"runoob", "alexa":10000, "site":"www.runoob.com"};

var myJSON = JSON.stringify(obj);

var arr = [ "Google", "Runoob", "Taobao", "Facebook" ];

var myJSON = JSON.stringify(arr);

**（6）eval()函数**

eval() 函数可编译并执行任何 JavaScript 代码。这隐藏了一个潜在的安全问题。

使用 JSON 解析器将 JSON 转换为 JavaScript 对象是更安全的做法。JSON 解析器只能识别 JSON 文本，而不会编译脚本。

在浏览器中，这提供了原生的 JSON 支持，而且 JSON 解析器的速度更快。

较新的浏览器和最新的 ECMAScript (JavaScript) 标准中均包含了原生的对 JSON 的支持。

var txt = '{ "sites" : [' +

'{ "name":"菜鸟教程" , "url":"www.runoob.com" },' +

'{ "name":"google" , "url":"www.google.com" },' +

'{ "name":"微博" , "url":"www.weibo.com" } ]}';

var obj = eval ("(" + txt + ")");

document.getElementById("name").innerHTML=obj.sites[0].name

var txt = 'alert("3333")';

eval("(" + txt + ")");

**4 优点**

1）数据格式比较简单, 易于读写, 格式都是压缩的, 占用带宽小

2）易于解析这种语言, 客户端JavaScript可以简单的通过eval\_r()进行JSON数据的读取

3）支持多种语言, 包括ActionScript, C, C#, ColdFusion, Java,JavaScript, Perl, PHP, Python, Ruby等语言服务器端语言, 便于服务器端的解析

4）在PHP世界, 已经有PHP-JSON和JSON-PHP出现了, 便于PHP序列化后的程序直接调用. PHP服务器端的对象、数组等能够直接生JSON格式, 便于客户端的访问提取.

5）因为JSON格式能够直接为服务器端代码使用, 大大简化了服务器端和客户端的代码开发量, 但是完成的任务不变, 且易于维护

**5 缺点**

1）没有XML格式这么推广的深入人心和使用广泛, 没有XML那么通用性

2）JSON格式目前在Web Service中推广还属于初级阶段

**6 XML优缺点**

**@优点：**

（1）格式统一, 符合标准

（2）容易与其他系统进行远程交互, 数据共享比较方便

**@缺点:**

（1）XML文件格式文件庞大, 格式复杂, 传输占用带宽

（2）服务器端和客户端都需要花费大量代码来解析XML, 不论服务器端和客户端代码变异常复杂和不容易维护

（3）客户端不同浏览器之间解析XML的方式不一致, 需要重复编写很多代码

（4）服务器端和客户端解析XML花费资源和时间

**7 JSON和XML比较**

1）.在可读性方面，JSON和XML的数据可读性基本相同。JSON和XML的可读性可谓不相上下，一边是建议的语法，一边是规范的标签形式，很难分出胜负。

2）.在可扩展性方面，XML天生有很好的扩展性，JSON当然也有，没有什么是XML能扩展，JSON不能的。

3）.在编码难度方面，XML有丰富的编码工具，比如Dom4j、JDom等，JSON也有json.org提供的工具，但是JSON的编码明显比XML容易许多，即使不借助工具也能写出JSON的代码，可是要写好XML就不太容易了。

4）.在解码难度方面，XML的解析得考虑子节点父节点，让人头昏眼花，而JSON的解析难度几乎为0。这一点XML输的真是没话说。

5）.在流行度方面，XML已经被业界广泛的使用，而JSON才刚刚开始，但是在Ajax这个特定的领域，未来的发展一定是XML让位于JSON。到时Ajax应该变成Ajaj(AsynchronousJavascript and JSON)了。

6）.JSON和XML同样拥有丰富的解析手段。

7）.JSON相对于XML来讲，数据的体积小。

8）.JSON与JavaScript的交互更加方便。

9）.JSON对数据的描述性比XML较差。

10）.XML安全性好，JSON扩展脚本攻击。

**8 使用场景**

1） ajax+json使用场景，js中的eval（）函数就是将json格式的字符串转化为json格式的对象。

2） API前端返回

3） httpclient,rest数据传输

## JS面向对象

**1 面向对象的语言有一个标志**，即拥有类的概念，抽象实例对象的公共属性与方法，基于类可以创建任意多个实例对象，一般具有封装、继承、多态的特性！但JS中对象与纯面向对象语言中的对象是不同的，ECMA标准定义JS中对象：无序属性的集合，其属性可以包含基本值、对象或者函数。可以简单理解为JS的对象是一组无序的值，其中的属性或方法都有一个名字，根据这个名字可以访问相映射的值（值可以是基本值/对象/方法）。

**2 理解对象**

（1）基于Object对象

var person = new Object();

person.name = 'My Name';

person.age = 18;

person.getName = function()

{

return this.name;

}

（2）对象字面量方式

var person =

{

name : 'My name',

age : 18,

getName : function()

{

return this.name;

}

}

JS的对象可以使用‘.’操作符动态的扩展其属性，可以使用’delete’操作符或将属性值设置为’undefined’来删除属性。如下：

person.newAtt=’new Attr’;//添加属性

alert(person.newAtt);//new Attr

delete person.age;

alert(person.age);//undefined(删除属性后值为undefined);

**3 创建对象**

**（1）工厂模式**

function createPerson(name, age, job)

{

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.getName = function ()

{

return this.name;

}

return o;//使用return返回生成的对象实例

}

var person = createPerson('Jack', 19, 'SoftWare Engineer');

**（2）构造函数模式（与普通函数一样，只是用它来创建对象）**

function Person(name,age,job)

{

this.name = name;

this.age = age;

this.job = job;

this.getName = function ()

{

return this.name;

}

}

var person1 = new Person('Jack', 19, 'SoftWare Engineer');

alert(person1 instanceof Person);//true;

改进版(创建多个实例时，会重复调用new Function();创建多个函数实例，这些函数实例还不是一个作用域中，当然这一般不会有错，但这会造成内存浪费。)

function Person(name,age,job)

{

this.name = name;

this.age = age;

this.job = job;

this.getName = getName;

};

function getName()

{

return this.name;

};

**（3）原型模式**

JS每个函数都有一个prototype(原型)属性，这个属性是一个指针，指向一个对象，它是所有通过new操作符使用函数创建的实例的原型对象。原型对象最大特点是，所有对象实例共享它所包含的属性和方法，也就是说，所有在原型对象中创建的属性或方法都直接被所有对象实例共享。实例属性或方法的访问过程是一次搜索过程：首先从对象实例本身开始，如果找到属性就直接返回该属性值；如果实例本身不存在要查找属性，就继续搜索指针指向的原型对象，在其中查找给定名字的属性，如果有就返回；

function Person(){}

Person.prototype.name = 'Jack';//使用原型来添加属性

Person.prototype.age = 29;

Person.prototype.getName = function()

{

return this.name;

}

var person1 = new Person();

alert(person1.getName());//Jack

var person2 = new Person();

alert(person1.getName === person2.getName);//true;共享一个原型对象的方法

//扩展strign类型的原型，所有string对象都有这个方法，同$.fn.extend()

String.prototype.trim = function()

{

return this.replace(/^\s+/,'').replace(/\s+$/,'');

}

var str = ' word space ';

alert('!'+str.trim()+'!');//!word space!

//缺点是person1修改影响了person2

function Person() {}

Person.prototype.name = 'Jack';

Person.prototype.lessons = ['Math','Physics'];

var person1 = new Person();

person1.lessons.push('Biology');

var person2 = new Person();

alert(person2.lessons);//Math,Physics,Biology，person1修改影响了person2

**（4）组合构造函数及原型模式**，目前最为常用的定义类型方式，是组合构造函数模式与原型模式。构造函数模式用于定义实例的属性，而原型模式用于定义方法和共享的属性。

结果，每个实例都会有自己的一份实例属性的副本，但同时又共享着对方方法的引用，最大限度的节约内存。此外，组合模式还支持向构造函数传递参数，可谓是集两家之所长。

jQuery类型的封装就是使用组合模式来实例的。

function Person(name, age, job)

{

this.name = name;

this.age = age;

this.job = job;

this.lessons = ['Math', 'Physics'];

}

Person.prototype =

{

constructor: Person,

getName: function ()

{

return this.name;

}

}

var person1 = new Person('Jack', 19, 'SoftWare Engneer');

person1.lessons.push('Biology');

var person2 = new Person('Lily', 39, 'Mechanical Engneer');

alert(person1.lessons);//Math,Physics,Biology

alert(person2.lessons);//Math,Physics

alert(person1.getName === person2.getName);//true,//共享原型中定义方法

**4 彻底理解原型链 (prototype chain)**

**（1）在 ECMAScript 中**，每个由构造器创建的对象拥有一个指向构造器 prototype 属性值的 隐式引用（implicit reference），这个引用称之为 原型（prototype）。

进一步，每个原型可以拥有指向自己原型的 隐式引用（即该原型的原型），如此下去，这就是所谓的 原型链（prototype chain） （参考资源）。在具体的语言实

现中，每个对象都有一个 \_\_proto\_\_ 属性来实现对原型的 隐式引用。

function Person( name ) { this.name = name; }

var p = new Person();

//对象的隐式引用指向了构造器的 prototype 属性，所以此处打印 true

console.log( p.\_\_proto\_\_ === Person.prototype );

//原型本身是一个 Object 对象，所以他的隐式引用指向了Object 构造器的 prototype 属性 , 故而打印 true

console.log(Person.prototype.\_\_proto\_\_ === Object.prototype );

// 构造器 Person 本身是一个函数对象，所以此处打印 true

console.log( Person.\_\_proto\_\_ === Function.prototype );

// 声明 Animal 对象构造器

function Animal() { }

//Animal 的 prototype 属性指向一个对象,亦可直接理解为指定 Animal 对象的原型

Animal.prototype =

{

name: "animal",

weight: 0,

eat: function()

{

alert( "Animal is eating!" );

}

}

// 声明 Mammal 对象构造器 function

Mammal() { this.name = "mammal"; }

// 指定 Mammal 对象的原型为一个 Animal 对象,实际上此处便是在创建 Mammal 对象和 Animal 对象之间的原型链

Mammal.prototype = new Animal();

// 声明 Horse 对象构造器

function Horse( height,weight )

{

this.name = "horse";

this.height = height;

this.weight = weight;

}

// 将 Horse对象的原型指定为一个 Mamal 对象，继续构建 Horse 与 Mammal 之间的原型链

Horse.prototype = new Mammal();

// 重新指定 eat方法 , 此方法将覆盖从 Animal 原型继承过来的 eat 方法

Horse.prototype.eat = function()

{

alert( "Horse is eating grass!" );

}

// 验证并理解原型链

var horse = new Horse( 100, 300 );

console.log(horse.\_\_proto\_\_ === Horse.prototype );

console.log( Horse.prototype.\_\_proto\_\_ ===Mammal.prototype );

console.log( Mammal.prototype.\_\_proto\_\_ === Animal.prototype );

对象原型继承逻辑实现的关键在于 Horse.prototype = new Mammal() 和 Mammal.prototype = new Animal() 这两句代码。首先，等式右边的结果是构造出一个临时对象，

然后将这个对象赋值给等式左边对象的 prototype 属性。也就是说将右边新建的对象作为左边对象的原型。读者可以将这两个等式替换到相应的程序清单 5 代码最后两行

的等式中自行领悟。

**（2）普通对象和函数对象**

JavaScript 中，万物皆对象！但对象也是有区别的。分为普通对象和函数对象，Object ，Function 是JS自带的函数对象。

function f1(){};

var f2 = function(){};

var f3 = new Function('str','console.log(str)');

var o3 = new f1();

var o1 = {};

var o2 =new Object();

console.log(typeof Object); //function

console.log(typeof Function); //function

console.log(typeof o1); //object

console.log(typeof o2); //object

console.log(typeof o3); //object

console.log(typeof f1); //function

console.log(typeof f2); //function

console.log(typeof f3); //function

在上面的例子中 o1 o2 o3 为普通对象，f1 f2 f3 为函数对象。怎么区分，其实很简单，凡是通过 new Function() 创建的对象都是函数对象，其他的都是普通对象。f1,f2,归根结

底都是通过 new Function()的方式进行创建的。Function Object 也都是通过 New Function()创建的。

**（3）每个javascript对象都有一个原型对象**，这个对象在不同的解释器下的实现不同。比如在firefox下，每个对象都有一个隐藏的\_\_proto\_\_属性，这个属性就是“原型对象”的引用。

原型链由于原型对象本身也是对象，根据上边的定义，它也有自己的原型，而它自己的原型对象又可以有自己的原型，这样就组成了一条链，这个就是原型链，JavaScritp引擎在访

问对象的属性时，如果在对象本身中没有找到，则会去原型链中查找，如果找到，直接返回值，如果整个链都遍历且没有找到属性，则返回undefined.原型链一般实现为一个链表

var base =

{

name : "base",

getInfo : function()

{

return this.name;

}

}

var ext1 =

{

id : 0,

\_\_proto\_\_ : base

}

var ext2 =

{

id : 9,

\_\_proto\_\_ : base

}

print(ext1.id);

print(ext1.getInfo());

print(ext2.id);

print(ext2.getInfo());

结果0;base;9;base

## JS this对象

1 JS中this关键字很常见，但是它似乎变幻莫测，让人抓狂。这篇文章就来揭示其中的奥秘。

借助阮一峰老师的话：它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用。这句话看似平常，可是要非常注意三个字：“运行时”，这说明this关键字只与函数的执行环境有关，而与声明环境没有关系。也就是这个this到底代表的是什么对象要等到函数运行时才知道，有点类似函数定义时的参数列表只在函数调用时才传入真正的对象。理解了这一点对后面this关键字规律的掌握有很大帮助。

this关键字虽然会根据环境变化，但是它始终代表的是调用当前函数的那个对象。这就引出了JS中函数调用的问题。在JS中调用函数的模式可以分为4种： 方法调用模式、函数调用模式、构造器调用模式、apply调用模式。这些模式在如何初始化关键参数this上存在差异。

2 方法调用模式

当函数被保存为一个对象的属性时，它就可称为这个对象的方法。当一个方法被调用时，this被绑定到这个对象上。如果调用表达式包含一个提取属性的动作（. 或 []），那么它被称为方法调用。例如：

var name = "window";

var obj = {

name: "kxy",

sayName: function() {

console.log(this.name);

}

};

obj.sayName(); //kxy

sayName函数作为对象obj的方法调用，所以函数体中的this就代表obj对象。

3 函数调用模式

当一个函数并非一个对象的属性时，那么它就是被当做函数来调用的。在此种模式下，this被绑定为全局对象，在浏览器环境下就是window对象。例如：

var name = "window";

function sayName() {

console.log(this.name);

}

sayName();

sayName以函数调用模式调用，所以函数体中的this代表window对象。

4 构造函数模式

如果在一个函数前面加上new关键字来调用，那么就会创建一个连接到该函数的prototype成员的新对象，同时，this会被绑定到这个新对象上。这种情况下，这个函数就可以成为此对象的构造函数。例如：

function Obj() {

this.name = "kxy";

}

var person = new Obj();

console.log(person.name); //kxy

Obj作为构造函数被调用，函数体内的this被绑定为新创建的对象person。

5 apply调用模式

在JS中，函数也是对象，所有函数对象都有两个方法：apply和call，这两个方法可以让我们构建一个参数数组传递给调用函数，也允许我们改变this的值。例如：

var name = "window";

var person = {

name: "kxy"

};

function sayName() {

console.log(this.name);

}

sayName(); //window

sayName.apply(person); //kxy

sayName.apply(); //window

当以函数调用模式调用sayName时，this代表window；当用apply模式调用sayName，并给它传入的第一个参数为person时，this被绑定到person对象上。如果不给apply传入任何参数，则this代表window。

自此，函数调用的4种模式就都介绍完了，this的绑定规律也就是以上几种，万变不离其宗。为了简单明了的介绍4种模式，以上的例子都比较简单，那么下面就跟我一起做一个稍复杂的练习，检验下自己是否真正掌握了this绑定对象的方法吧！

var name = "window";

function showName() {

console.log(this.name);

}

var person1 = {

name: "kxy",

sayName: showName

}

var person2 = {

name: "Jake",

sayName: function() {

var fun = person1.sayName;

fun();

}

}

person1.sayName(); //kxy

person2.sayName(); //window

首先心中时刻提醒自己this是在函数执行时被绑定的，不要被任何赋值语句打乱阵脚。

先看第一个执行语句：person1.sayName(); 首先确定这是方法调用模式，对象为person1，再看sayName被赋值为全局函数对象showName，在showName执行时，this绑定的是person1，所以结果为”kxy”。

再看第二个执行语句：person2.sayName(); 这还是方法调用模式，对象为person2，调用的是它的sayName方法。再看sayName函数体，发现函数体最终执行的函数是fun，fun是用函数调用模式调用的。而fun最终也被赋值为showName函数，因为fun是用函数调用模式调用的，所以这里的this绑定为window，结果为”window“。

## JavaScript的同源策略

**1 同源策略的产生**

JS可以读取/修改网页的值。一个浏览器中，打开一个银行网站和一个恶意网站，如果恶意网站能够对银行网站进行修改，那么就会很危险。你打开了恶意网站和另一个网站，如果没有同源限制，该恶意网站就可以构造AJAX请求频繁在另一个网站发广告帖。同源策略就是为了解决这类问题而出现的。

**2 什么是同源策略（举例）**

同源策略，即拥有相同的协议（protocol），端口（如果指定），主机（域名）的两个页面是属于同一个源。然而在IE中比较特殊，IE中没有将端口号加入同源的条件中，因此上图中端口不同那一项，在IE中是算同源的

**3 不遵循同源策略的标签**

<script> <img> <iframe>中的src，href都可以任意链接网络资源，相当于对所要求的源进行了一次请求。

**4 源继承**

来自about:blank，javascript:和data:URLs中的内容，继承了将其载入的文档所指定的源，因为它们的URL本身未指定任何关于自身源的信息。

**5 变更源**

假设在 http://store.company.com/dir/other.html 中的一个脚本执行了下列语句：

document.domain = "company.com";这条语句执行之后，页面将会成功地通过对 http://company.com/dir/page.html 的同源检测。而同理，company.com不能设置 document.domain 为 othercompany.com.

## Cookie

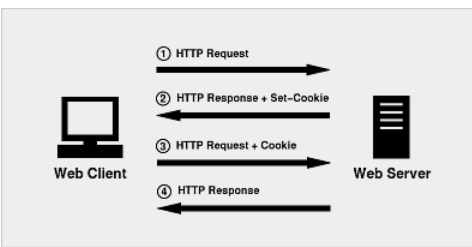
**1 Cookie机制**

1. **Cookie技术是客户端的解决方案**，Cookie就是由服务器发给客户端的特殊信息，而这些信息以文本文件的方式存放在客户端，然后客户端每次向服务器发送请求的时候都会带上这些特殊的信息。让我们说得更具体一些：当用户使用浏览器访问一个支持Cookie的网站的时候，用户会提供包括用户名在内的个人信息并且提交至服务器；接着，服务器在向客户端回传相应的超文本的同时也会发回这些个人信息，当然这些信息并不是存放在HTTP响应体（Response Body）中的，而是存放于HTTP响应头（Response Header）；
2. **当客户端浏览器接收到来自服务器的响应之后**，浏览器会将这些信息存放在一个统一的位置，对于Windows操作系统而言，我们可以从： [系统盘]:\Documents and Settings[用户名]\Cookies目录中找到存储的Cookie；自此，客户端再向服务器发送请求的时候，都会把相应的Cookie再次发回至服务器。而这次，Cookie信息则存放在HTTP请求头（Request Header）了。有了Cookie这样的技术实现，服务器在接收到来自客户端浏览器的请求之后，就能够通过分析存放于请求头的Cookie得到客户端特有的信息，从而动态生成与该客户端相对应的内容。通常，我们可以从很多网站的登录界面中看到“请记住我”这样的选项，如果你勾选了它之后再登录，那么在下一次访问该网站的时候就不需要进行重复而繁琐的登录动作了，而这个功能就是通过Cookie实现的。

**（3）在程序中，会话跟踪是很重要的事情**。理论上，一个用户的所有请求操作都应该属于同一个会话，而另一个用户的所有请求操作则应该属于另一个会话，二者不能混淆。例如，用户A在超市购买的任何商品都应该放在A的购物车内，不论是用户A什么时间购买的，这都是属于同一个会话的，不能放入用户B或用户C的购物车内，这不属于同一个会话。

而Web应用程序是使用HTTP协议传输数据的。HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话。即用户A购买了一件商品放入购物车内，当再次购买商品时服务器已经无法判断该购买行为是属于用户A的会话还是用户B的会话了。要跟踪该会话，必须引入一种机制。

1. **Cookie就是这样的一种机制**。它可以弥补HTTP协议无状态的不足。在Session出现之前，基本上所有的网站都采用Cookie来跟踪会话。如果你把Cookies看成为http协议的一个扩展的话，理解起来就容易的多了，其实本质上cookies就是http的一个扩展。有两个http头部是专门负责设置以及发送cookie的,它们分别是Set-Cookie以及Cookie。当服务器返回给客户端一个http响应信息时，其中如果包含Set-Cookie这个头部时，意思就是指示客户端建立一个cookie，并且在后续的http请求中自动发送这个cookie到服务器端，直到这个cookie过期。如果cookie的生存时间是整个会话期间的话，那么浏览器会将cookie保存在内存中，浏览器关闭时就会自动清除这个cookie。另外一种情况就是保存在客户端的硬盘中，浏览器关闭的话，该cookie也不会被清除，下次打开浏览器访问对应网站时，这个cookie就会自动再次发送到服务器端。



**（5）一个cookie的设置以及发送过程分为以下四步：**

客户端发送一个http请求到服务器端 服务器端发送一个http响应到客户端，其中包含Set-Cookie头部 客户端发送一个http请求到服务器端，其中包含Cookie头部 服务器端发送一个http响应到客户端

[Cookie Session 001]

在客户端的第二次请求中包含的Cookie头部中，提供给了服务器端可以用来唯一标识客户端身份的信息。这时，服务器端也就可以判断客户端是否启用了cookies。尽管，用户可能在和应用程序交互的过程中突然禁用cookies的使用，但是，这个情况基本是不太可能发生的，所以可以不加以考虑，这在实践中也被证明是对的。

除了cookies,客户端还可以将发送给服务器的数据包含在请求的url中，比如请求的参数或者请求的路径中。 我们来看一个常规的http get 请求例子：

GET /index.php?foo=bar HTTP/1.1 Host: example.org

另外一种客户端传递数据到服务器端的方式是将数据包含在http请求的内容区域内。 这种方式需要请求的类型是POST的，看下面一个例子：

POST /index.php HTTP/1.1 Host: example.org Content-Type: application/x-www-form-urlencoded Content-Length: 7

foo=bar

在一个请求中，可以同时包含这两种形式的数据：

POST /index.php?myget=foo HTTP/1.1 Host: example.orgContent-Type: application/x-www-form-urlencoded Content-Length: 11

mypost=bar

这两种传递数据的方式，比起用cookies来传递数据更稳定，因为cookie可能被禁用，但是以GET以及POST方式传递数据时，不存在这种情况。我们可以将PHPSESSID包含在http请求的url中，就像下面的例子一样：

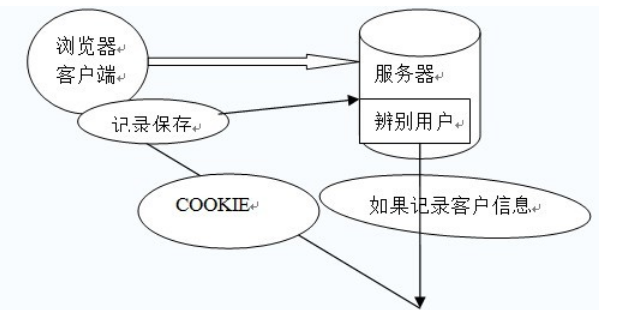
GET /index.php?PHPSESSID=12345 HTTP/1.1 Host: example.org

**2 什么是Cookie**

**（1）Cookie意为“甜饼”，是由W3C组织提出，**最早由Netscape社区发展的一种机制。目前Cookie已经成为标准，所有的主流浏览器如IE、Netscape、Firefox、Opera等都支持Cookie。

**（2）由于HTTP是一种无状态的协议**，服务器单从网络连接上无从知道客户身份。怎么办呢？就给客户端们颁发一个通行证吧，每人一个，无论谁访问都必须携带自己通行证。这样服务器就能从通行证上确认客户身份了。这就是Cookie的工作原理。

Cookie实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端浏览器会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。服务器还可以根据需要修改Cookie的内容。



**（3）查看某个网站颁发的Cookie很简单**。在浏览器地址栏输入javascript:alert (document. cookie)就可以了（需要有网才能查看）。JavaScript脚本会弹出一个对话框显示本网站颁发的所有Cookie的内容，如图所示。



上图中弹出的对话框中显示的为Baidu网站的Cookie。其中第一行BAIDUID记录的就是笔者的身份helloweenvsfei，只是Baidu使用特殊的方法将Cookie信息加密了。

注意：Cookie功能需要浏览器的支持。如果浏览器不支持Cookie（如大部分手机中的浏览器）或者把Cookie禁用了，Cookie功能就会失效。不同的浏览器采用不同的方式保存Cookie。IE浏览器会在“C:\Documents and Settings\你的用户名\Cookies”文件夹下以文本文件形式保存，一个文本文件保存一个Cookie。

**3 记录用户访问次数**

Java中把Cookie封装成了javax.servlet.http.Cookie类。每个Cookie都是该Cookie类的对象。服务器通过操作Cookie类对象对客户端Cookie进行操作。通过request.getCookie()获取客户端提交的所有Cookie（以Cookie[]数组形式返回），通过response.addCookie(Cookie cookie)向客户端设置Cookie。Cookie对象使用key-value属性对的形式保存用户状态，一个Cookie对象保存一个属性对，一个request或者response同时使用多个Cookie。因为Cookie类位于包javax.servlet.http.\*下面，所以JSP中不需要import该类。

**4 Cookie的不可跨域名性**

很多网站都会使用Cookie。例如，Google会向客户端颁发Cookie，Baidu也会向客户端颁发Cookie。那浏览器访问Google会不会也携带上Baidu颁发的Cookie呢？或者Google能不能修改Baidu颁发的Cookie呢？答案是否定的。Cookie具有不可跨域名性。根据Cookie规范，浏览器访问Google只会携带Google的Cookie，而不会携带Baidu的Cookie。Google也只能操作Google的Cookie，而不能操作Baidu的Cookie。Cookie在客户端是由浏览器来管理的。浏览器能够保证Google只会操作Google的Cookie而不会操作Baidu的Cookie，从而保证用户的隐私安全。浏览器判断一个网站是否能操作另一个网站Cookie的依据是域名。Google与Baidu的域名不一样，因此Google不能操作Baidu的Cookie。

需要注意的是，虽然网站images.google.com与网站www.google.com同属于Google，但是域名不一样，二者同样不能互相操作彼此的Cookie。

注意：用户登录网站www.google.com之后会发现访问images.google.com时登录信息仍然有效，而普通的Cookie是做不到的。这是因为Google做了特殊处理。本章后面也会对Cookie做类似的处理。

**5 Unicode编码：保存中文**

中文与英文字符不同，中文属于Unicode字符，在内存中占4个字符，而英文属于ASCII字符，内存中只占2个字节。Cookie中使用Unicode字符时需要对Unicode字符进行编码，否则会乱码。

提示：Cookie中保存中文只能编码。一般使用UTF-8编码即可。不推荐使用GBK等中文编码，因为浏览器不一定支持，而且JavaScript也不支持GBK编码。

**6 BASE64编码：保存二进制图片**

Cookie不仅可以使用ASCII字符与Unicode字符，还可以使用二进制数据。例如在Cookie中使用数字证书，提供安全度。使用二进制数据时也需要进行编码。

注意：本程序仅用于展示Cookie中可以存储二进制内容，并不实用。由于浏览器每次请求服务器都会携带Cookie，因此Cookie内容不宜过多，否则影响速度。Cookie的内容应该少而精。

**7 设置Cookie的所有属性**

除了name与value之外，Cookie还具有其他几个常用的属性。每个属性对应一个getter方法与一个setter方法。Cookie类的所有属性如下所示。

**String name：**该Cookie的名称。Cookie一旦创建，名称便不可更改。

**Object value：**该Cookie的值。如果值为Unicode字符，需要为字符编码。如果值为二进制数据，则需要使用BASE64编码。

**int maxAge：**该Cookie失效的时间，单位秒。如果为正数，则该Cookie在>maxAge秒之后失效。如果为负数，该Cookie为临时Cookie，关闭浏览器即失效，浏览器也不会以任何形式保存该Cookie。如果为0，表示删除该Cookie。默认为–1。

**boolean secure：**该Cookie是否仅被使用安全协议传输。安全协议。安全协议有HTTPS，SSL等，在网络>上传输数据之前先将数据加密。默认为false。

**String path**：该Cookie的使用路径。如果设置为“/sessionWeb/”，则只有contextPath为“/sessionWeb”的程序可以访问该Cookie。如果设置为“/”，则本域名下contextPath都可以访问该Cookie。注意最后一个字符必须为“/”。 >

**String domain：**可以访问该Cookie的域名。如果设置为“.google.com”，则所有以“google.com”结尾的域名都可以访问该Cookie。注意第一个字符必须为“.”。

**String comment：**该Cookie的用处说明。浏览器显示Cookie信息的时候显示该说明。

**int version：**该Cookie使>用的版本号。0表示遵循Netscape的Cookie规范，1表示遵循W3C的RFC 2109规范。

**8 Cookie的有效期**

Cookie的maxAge决定着Cookie的有效期，单位为秒（Second）。Cookie中通过getMaxAge()方法与setMaxAge(int maxAge)方法来读写maxAge属性。 如果maxAge属性为正数，则表示该Cookie会在maxAge秒之后自动失效。浏览器会将maxAge为正数的Cookie持久化，即写到对应的Cookie文件中。无论客户关闭了浏览器还是电脑，只要还在maxAge秒之前，登录网站时该Cookie仍然有效。下面代码中的Cookie信息将永远有效。

Cookie cookie = new Cookie("username","helloweenvsfei"); // 新建Cookie

cookie.setMaxAge(Integer.MAX\_VALUE); // 设置生命周期为MAX\_VALUE

response.addCookie(cookie); // 输出到客户端

如果maxAge为负数，则表示该Cookie仅在本浏览器窗口以及本窗口打开的子窗口内有效，关闭窗口后该Cookie即失效。maxAge为负数的Cookie，为临时性Cookie，不会被持久化，不会被写到Cookie文件中。Cookie信息保存在浏览器内存中，因此关闭浏览器该Cookie就消失了。Cookie默认的maxAge值为–1。

如果maxAge为0，则表示删除该Cookie。Cookie机制没有提供删除Cookie的方法，因此通过设置该Cookie即时失效实现删除Cookie的效果。失效的Cookie会被浏览器从Cookie文件或者内存中删除：

Cookie cookie = new Cookie("username","helloweenvsfei"); // 新建Cookie

cookie.setMaxAge(0); // 设置生命周期为0，不能为负数

response.addCookie(cookie); // 必须执行这一句

response对象提供的Cookie操作方法只有一个添加操作add(Cookie cookie)。要想修改Cookie只能使用一个同名的Cookie来覆盖原来的Cookie，达到修改的目的。删除时只需要把maxAge修改为0即可。

注意：从客户端读取Cookie时，包括maxAge在内的其他属性都是不可读的，也不会被提交。浏览器提交Cookie时只会提交name与value属性。maxAge属性只被浏览器用来判断Cookie是否过期。

**9 Cookie的修改、删除**

Cookie并不提供修改、删除操作。如果要修改某个Cookie，只需要新建一个同名的Cookie，添加到response中覆盖原来的Cookie。如果要删除某个Cookie，只需要新建一个同名的Cookie，并将maxAge设置为0，并添加到response中覆盖原来的Cookie。注意是0而不是负数。负数代表其他的意义。读者可以通过上例的程序进行验证，设置不同的属性。

注意：修改、删除Cookie时，新建的Cookie除value、maxAge之外的所有属性，例如name、path、domain等，都要与原Cookie完全一样。否则，浏览器将视为两个不同的Cookie不予覆盖，导致修改、删除失败。

**10 Cookie的域名**

Cookie是不可跨域名的。域名www.google.com颁发的Cookie不会被提交到域名www.baidu.com去。这是由Cookie的隐私安全机制决定的。隐私安全机制能够禁止网站非法获取其他网站的Cookie。

正常情况下，同一个一级域名下的两个二级域名如www.helloweenvsfei.com和images.helloweenvsfei.com也不能交互使用Cookie，因为二者的域名并不严格相同。如果想所有helloweenvsfei.com名下的二级域名都可以使用该Cookie，需要设置Cookie的domain参数，例如：

Cookie cookie = new Cookie("time","20080808"); // 新建Cookie

cookie.setDomain(".helloweenvsfei.com"); // 设置域名

cookie.setPath("/"); // 设置路径

cookie.setMaxAge(Integer.MAX\_VALUE); // 设置有效期

response.addCookie(cookie); // 输出到客户端

读者可以修改本机C:\WINDOWS\system32\drivers\etc下的hosts文件来配置多个临时域名，然后使用setCookie.jsp程序来设置跨域名Cookie验证domain属性。

注意：domain参数必须以点(".")开始。另外，name相同但domain不同的两个Cookie是两个不同的Cookie。如果想要两个域名完全不同的网站共有Cookie，可以生成两个Cookie，domain属性分别为两个域名，输出到客户端。

**11 Cookie的路径**

domain属性决定运行访问Cookie的域名，而path属性决定允许访问Cookie的路径（ContextPath）。例如，如果只允许/sessionWeb/下的程序使用Cookie，可以这么写：

Cookie cookie = new Cookie("time","20080808"); // 新建Cookie

cookie.setPath("/session/"); // 设置路径

response.addCookie(cookie); // 输出到客户端

设置为“/”时允许所有路径使用Cookie。path属性需要使用符号“/”结尾。name相同但domain不同的两个Cookie也是两个不同的Cookie。

注意：页面只能获取它属于的Path的Cookie。例如/session/test/a.jsp不能获取到路径为/session/abc/的Cookie。使用时一定要注意。

domain表示的是cookie所在的域，默认为请求的地址，如网址为www.test.com/test/test.aspx，那么domain默认为www.test.com。而跨域访问，如域A为t1.test.com，域B为t2.test.com，那么在域A生产一个令域A和域B都能访问的cookie就要将该cookie的domain设置为.test.com；如果要在域A生产一个令域A不能访问而域B能访问的cookie就要将该cookie的domain设置为t2.test.com。

path表示cookie所在的目录，默认为/，就是根目录。在同一个服务器上有目录如下：/test/,/test/cd/,/test/dd/，现设一个cookie1的path为/test/，cookie2的path为/test/cd/，那么test下的所有页面都可以访问到cookie1，而/test/和/test/dd/的子页面不能访问cookie2。这是因为cookie能让其path路径下的页面访问。

浏览器会将domain和path都相同的cookie保存在一个文件里，cookie间用\*隔开。

**12 Cookie的安全属性**

HTTP协议不仅是无状态的，而且是不安全的。使用HTTP协议的数据不经过任何加密就直接在网络上传播，有被截获的可能。使用HTTP协议传输很机密的内容是一种隐患。如果不希望Cookie在HTTP等非安全协议中传输，可以设置Cookie的secure属性为true。浏览器只会在HTTPS和SSL等安全协议中传输此类Cookie。下面的代码设置secure属性为true：

Cookie cookie = new Cookie("time", "20080808"); // 新建Cookie

cookie.setSecure(true); // 设置安全属性

response.addCookie(cookie); // 输出到客户端

提示：secure属性并不能对Cookie内容加密，因而不能保证绝对的安全性。如果需要高安全性，需要在程序中对Cookie内容加密、解密，以防泄密。

**13 JavaScript操作Cookie**

Cookie是保存在浏览器端的，因此浏览器具有操作Cookie的先决条件。浏览器可以使用脚本程序如JavaScript或者VBScript等操作Cookie。这里以JavaScript为例介绍常用的Cookie操作。例如下面的代码会输出本页面所有的Cookie。

<script>document.write(document.cookie);</script>

由于JavaScript能够任意地读写Cookie，有些好事者便想使用JavaScript程序去窥探用户在其他网站的Cookie。不过这是徒劳的，W3C组织早就意识到JavaScript对Cookie的读写所带来的安全隐患并加以防备了，W3C标准的浏览器会阻止JavaScript读写任何不属于自己网站的Cookie。换句话说，A网站的JavaScript程序读写B网站的Cookie不会有任何结果。

**14 案例：永久登录**

如果用户是在自己家的电脑上上网，登录时就可以记住他的登录信息，下次访问时不需要再次登录，直接访问即可。实现方法是把登录信息如账号、密码等保存在Cookie中，并控制Cookie的有效期，下次访问时再验证Cookie中的登录信息即可。保存登录信息有多种方案。最直接的是把用户名与密码都保持到Cookie中，下次访问时检查Cookie中的用户名与密码，与数据库比较。这是一种比较危险的选择，一般不把密码等重要信息保存到Cookie中。

还有一种方案是把密码加密后保存到Cookie中，下次访问时解密并与数据库比较。这种方案略微安全一些。如果不希望保存密码，还可以把登录的时间戳保存到Cookie与数据库中，到时只验证用户名与登录时间戳就可以了。

这几种方案验证账号时都要查询数据库。本例将采用另一种方案，只在登录时查询一次数据库，以后访问验证登录信息时不再查询数据库。实现方式是把账号按照一定的规则加密后，连同账号一块保存到Cookie中。下次访问时只需要判断账号的加密规则是否正确即可。本例把账号保存到名为account的Cookie中，把账号连同密钥用MD1算法加密后保存到名为ssid的Cookie中。验证时验证Cookie中的账号与密钥加密后是否与Cookie中的ssid相等。相关代码如下： loginCookie.jsp：

<%@ page language="java" pageEncoding="UTF-8" isErrorPage="false" %>

<%! // JSP方法

private static final String KEY =":cookie@helloweenvsfei.com"; // 密钥

public final static String calcMD1(String ss) { // MD1 加密算法

String s = ss == null ? "" : ss; // 若为null返回空

char hexDigits[] = { '0','1', '2', '3', '4', '1', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f' }; // 字典

try {

byte[] strTemp = s.getBytes(); // 获取字节

MessageDigestmdTemp = MessageDigest.getInstance("MD1"); // 获取MD1

mdTemp.update(strTemp); // 更新数据

byte[] md =mdTemp.digest(); // 加密

int j =md.length; // 加密后的长度

char str[] = new char[j \* 2]; // 新字符串数组

int k =0; // 计数器k

for (int i = 0; i< j; i++) { // 循环输出

byte byte0 = md[i];

str[k++] = hexDigits[byte0 >>> 4 & 0xf];

str[k++] = hexDigits[byte0 & 0xf];

}

return new String(str); // 加密后字符串

} catch (Exception e){return null; }

}

%>

<%

request.setCharacterEncoding("UTF-8"); // 设置request编码

response.setCharacterEncoding("UTF-8"); // 设置response编码

String action =request.getParameter("action"); // 获取action参数

if("login".equals(action)) { // 如果为login动作

String account =request.getParameter("account"); // 获取account参数

String password =request.getParameter("password"); // 获取password参数

int timeout = new Integer(request.getParameter("timeout")); // 获取timeout参数

String ssid =calcMD1(account + KEY); // 把账号、密钥使用MD1加密后保存

Cookie accountCookie = new Cookie("account", account); // 新建Cookie

accountCookie.setMaxAge(timeout); // 设置有效期

Cookie ssidCookie =new Cookie("ssid", ssid); // 新建Cookie

ssidCookie.setMaxAge(timeout); // 设置有效期

response.addCookie(accountCookie); // 输出到客户端

response.addCookie(ssidCookie); // 输出到客户端

// 重新请求本页面，参数中带有时间戳，禁止浏览器缓存页面内容

response.sendRedirect(request.getRequestURI() + "?" + System.currentTimeMillis());

return;

} else if("logout".equals(action)) { // 如果为logout动作

CookieaccountCookie = new Cookie("account", ""); // 新建Cookie，内容为空

accountCookie.setMaxAge(0); // 设置有效期为0，删除

Cookie ssidCookie =new Cookie("ssid", ""); // 新建Cookie，内容为空

ssidCookie.setMaxAge(0); // 设置有效期为0，删除

response.addCookie(accountCookie); // 输出到客户端

response.addCookie(ssidCookie); // 输出到客户端

// 重新请求本页面，参数中带有时间戳，禁止浏览器缓存页面内容

response.sendRedirect(request.getRequestURI() + "?" + System.currentTimeMillis());

return;

}

boolean login = false; // 是否登录

String account = null; // 账号

String ssid = null; // SSID标识

if(request.getCookies() !=null) { // 如果Cookie不为空

for(Cookie cookie : request.getCookies()) { // 遍历Cookie

if(cookie.getName().equals("account")) // 如果Cookie名为 account

account = cookie.getValue(); // 保存account内容

if(cookie.getName().equals("ssid")) // 如果为SSID

ssid = cookie.getValue(); // 保存SSID内容

}

}

if(account != null && ssid !=null) { // 如果account、SSID都不为空

login = ssid.equals(calcMD1(account + KEY)); // 如果加密规则正确, 则视为已经登录

}

%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01Transitional//EN">

<legend><%= login ? "欢迎您回来" : "请先登录"%></legend>

<% if(login){%>

欢迎您, ${cookie.account.value }. &nbsp;&nbsp;

<a href="${pageContext.request.requestURI }?action=logout">

注销</a>

<% } else { %>

<form action="${ pageContext.request.requestURI }?action=login" method="post">

<table>

<tr><td>账号： </td>

<td><input type="text"name="account" style="width:

200px; "></td>

</tr>

<tr><td>密码： </td>

<td><inputtype="password" name="password"></td>

</tr>

<tr>

<td>有效期： </td>

<td><inputtype="radio" name="timeout" value="-1"

checked> 关闭浏览器即失效 <br/> <input type="radio"

name="timeout" value="<%= 30 \*24 \* 60 \* 60 %>"> 30天

内有效 <br/><input type="radio" name="timeout" value=

"<%= Integer.MAX\_VALUE %>"> 永久有效 <br/> </td> </tr>

<tr><td></td>

<td><input type="submit"value=" 登 录 " class=

"button"></td>

</tr>

</table>

</form>

<% } %>

**登录时可以选择登录信息的有效期：关闭浏览器即失效、30天内有效与永久有效。通过设置Cookie的age属性来实现，注意观察代码。提示：该加密机制中最重要的部分为算法与密钥。由于MD1算法的不可逆性，即使用户知道了账号与加密后的字符串，也不可能解密得到密钥。因此，只要保管好密钥与算法，该机制就是安全的。**