1、 创建object实例的方式有两种：

①用new操作符后跟object()构造函数

var person=new Object();

person.name="Paul";

person.age=15;

②使用对象字面量表示法

var person={

name:"Paul",

age:15

};

属性名也可以使用字符串，例如：

var person={

"name":"Paul",

"age":15,

5:true

};

这里的数值属性名自动转换为字符串。

使用这种方法时可以留空花括号，var person={};person.name="Paul";person.age=15;这种方式与第一种方法相同

## 2.1 <script>标签

1、type属性：表示编写代码使用的脚本语言的内容类型(也成为MIME类型)。如果没有指定这个属性，那么其默认值为text/javascript。

2、在使用<script>标签嵌入JavaScript代码时，记住不要在代码中的任何地方出现”</script>”字符串，例如alert(“</script>”)，如果真要输出</script>，应该写成alert(“<\/script>”)。

3、带有src属性的<script>元素，不应该在其<script>和</script>标签之间再包含额外的JavaScript代码。如果包含了嵌入的代码，则只会下载并执行外部脚本文件，嵌入的代码会被忽略。

4、另外src属性还可以包含来自外部域的JavaScript文件，即src中的URL可以是任何的网址+\*.js，这样位于外部域中的js也会被加载执行，这一点经常用在XSS攻击中。

### 2.1.1 标签的位置

最好的位置是放在<body>标签的内容后面，这样可以保证前面的标签都加载完成之后再加载JavaScript代码，可以让使用者感到页面加载变快。

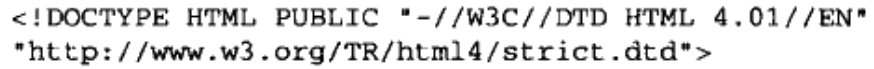
### 2.1.2 延迟脚本

使用defer属性可以使脚本在整个页面解析完成以后再执行。使用就是defer=”defer”。该属性只适用于外部脚本文件。

### 2.1.3 异步脚本

async属性也是只适用于外部脚本文件。该属性的目的就是不让页面等待该脚本的下载和执行，从而异步加载其他页面内容。这样在脚本当中就不适合修改DOM了。使用就是async=”async”。

## 2.3 文档模式



主要影响CSS内容的呈现。

# 第3章 基本概念

1、大小写敏感

## Identifiers

1、标识符就是变量、函数、属性或者函数参数的名称。

2、JavaScript中的标识符只能包含字母、下划线、数字或者美元符号$，但是不能以数字开头。

按照通常的习惯，会采用骆驼命名法。

## VARIABLES

1. 用var定义类型，但是没有初始化，例如var message;，这里message的值就是undefined。
2. 初始化变量var message=”hi”;并不意味着变量成为了字符串类型，它只是分配给变量的值，之后仍然可能会改变变量的值甚至改变变量的类型，例如message=100;
3. 上面用var定义的变量只会在本地域起作用，例如在一个函数中声明这个变量，当在函数外面调用该变量时就会报错。如果去掉关键字var，直接在函数中定义变量，message=19;这样这个变量就是一个全局变量(在浏览器中提示未定义)，可以在函数外面使用。但是这种方法并不提倡，因为会给后续的维护带来麻烦。

## DATA TYPES

基本数据类型：Undefined，Null，Boolean，Number，String。

复杂数据类型：Object，是一种无序的键/值对列表。

## The typeof Operator

typeof就是确定变量的数据类型。用该操作符会确定6种类型：undefined,boolean,string,number,function,object。

typeof null会返回object类型，因为null被认为是空的对象引用。

## The Undefined Type

1、当一个变量用var声明但是没有初始化时，该变量的值就是undefined。

var message; alert(message==undefined); //true

改代码是完全一样的和var message=undefined; alert(message==undefined); //true

2、一个变量未声明是不能使用的，例如alert(age); //cause an error

但是对这样的变量使用typeof却返回undefined，alert(typeof age); //undefined

## The Null Type

null是一个空的对象指针。

undefined是一个null的派生，所以alert(null==undefined); //true

## The Boolean Type

Boolean()这个函数能将任何类型数据转换成boolean类型。

var message=”hi”;

var messageBoolean=Boolean(message);

## The Number Type

八进制：以数字0开头，后面是0-7组成的数字串，如果后面的数字超过7，那么该八进制数就被作为十进制数对待。

十六进制：以0x(大小写不敏感)，

#### NaN

这个数值用于表示一个本来要返回数值的操作数未返回数值的情况(这样就不会抛出错误了)。比如在其他语言中任何数除以0都会抛出错误，但是在JavaScript中任何数除以0都会返回NaN，因此不会影响其他代码的执行。

1. 任何涉及NaN的操作都会返回NaN，例如(NaN/10)。
2. NaN与任何值都不相等，例如alert(NaN==NaN) //false

isNaN()函数可以接受任何类型的参数，来判断这个数是否”不是数值”。isNaN()在接收到一个值之后，会尝试将这个值转换为数值。某些不是数值的值会直接转换为数值，同时函数返回true或者false。比如：

var param="10";

alert(isNaN(param)); //false

alert(param); //10

#### 3.4.5.4 数值转换

有3个函数可以把非数值转换为数值：Number()、parseInt()和parseFloat()。Number()可以用于任何数据类型，另外两个可以专门把字符串转换成数值。

Number()函数的转换规则：

true和false会被转换为1和0

null返回0

undefined返回NaN

如果是字符串则遵循下列规则：

如果字符串中是有效的十六进制，则转换为等值的十进制。

如果字符串中是数值且前面有0，则0被忽略。

如果字符串是空的，则转换为0

其他的都转换为NaN

如果是对象，则调用对象的valueOf()方法，然后依照前面的规则转换值。如果转换的结果是NaN，则调用对象的toString()方法，然后再依照前面的规则转换。

parseInt()函数：

忽略字符串前面的空格，直接找到第一个非空格字符。如果第一个字符不是数字字符或者符号，就返回NaN。如果第一个是数字字符则接着往后直到是非数字字符。例如”1234blue”会转换成1234，”22.5”会被转换成22，因为小数点并不是有效的数字字符。

var num3=parseInt(“0xA”); //10，十六进制

var num4=parseInt(“070”); //56，八进制

可以添加第二个参数来表示是多少进制，可以添加2,8,10,16等。

如果添加了第二个参数，那么十六进制和八进制就不用添加前面的0x和0。

parseFloat()也是从第一个字符开始解析，直到遇到一个无效的浮点数字字符为止。例如，字符”22.34.5”将会被转换为22.34

parseFloat()可以识别所有浮点数格式，包括科学计数法(用e表示的)。该函数始终会忽略前导的0，所以十六进制始终会识别为0。小数点后都是0也只会保留整数部分。

### 3.4.6 String类型

#### 1 字符字面量

用单引号和用双引号没有区别。

任何字符串的长度都可以访问其length属性获得。

#### 2 字符串的特点

ECMAScript中的字符串是不可变的，也就是字符串一旦创建，它的值就不能改变。

#### 3 转换为字符串

### 3.4.7 Object类型

创建对象的语法一般为var o=new Object();但是如果不给构造函数传递参数，那么可以省略后面的括号。

### 3.5.7 相等操作符

2 全等和不全等

全等符号由===表示，它只在两个操作数未经转换就相等的情况下返回true。例如，

var result=(“55”===55); //false

### 3.6.5 for-in语句

列举出window对象的所有属性：

for(var propName in window)

{

document.write(propName);

}

但是：

var a=["a","b","c"];

for(var el in a){

document.write(el);

}

//输出结果：123。输出的是下标。

如果对象为null或者undefined，for-in会抛出异常。

### 3.7.1 理解参数

在ECMAScript中，即使你定义的函数只接收两个参数，在调用这个函数时也未必传递两个参数，可以传递一个，也可以传递三个，这是因为ECMAScript中的参数在函数内部是用一个数组来表示的，实际上，在函数内部可以使用arguments这个对象来访问这个参数数组，从而获取传递给函数的每一个参数。

ECMAScript中的所有参数传递的都是值，不可能通过引用传递参数。

没有传递值的参数会自动赋值为undefined，这就跟定义了变量但是没有初始化一样。

arguments对象中的值会自动映射到对应的参数上，所以修改arguments[1]，也会修改num2。

### 3.7.2 没有重载

ECMAScript中没有函数重载。

如果定义了两个名字相同的函数，则后面的函数会覆盖前面的函数。

# 第4章 变量、作用域和内存问题

### 4.1.2 赋值变量值

对于基本数据类型值：

var num1=5;

var num2=num1;

这里num1和num2是完全独立的两块内存空间，互不影响。

当复制引用数据类型时，复制的其实一个指针，两者指向堆内存中的同一块空间区域，所以改变一个对象的属性另外一个的属性也会改变。

### 4.1.4 类型检测

用typeof关键字检测变量类型，例如：

var name=null;

alert(typeof name); //object

一般在检测引用类型值时使用instanceof操作符，来判断是什么引用类型。例如，

alert(person instanceof Object); //person是Object类型吗？

alert(colors instanceof Array); //colors是Array类型吗？

alert(pattern instanceof RegExp); //pattern是RegExp类型吗？

如果用instanceof检测基本类型变量则始终返回false。

### 4.2.2 没有块级作用域

JavaScript中没有块级作用域只有函数作用域。比如：

for(var i=0;i<3;i++)

{}

alert(i); //仍然可以输出3

还有if语句

If(true){var name=”lizan”;}

alert(name); //仍然可以输出lizan

这是因为JavaScript中没有块级作用域。

但是如果变量是在函数中定义的，那么出了函数作用域就不可访问变量了。

如果变量没有用var声明而是直接进行赋值，那么该变量就是全局变量。

# 第5章 引用类型

## 5.1 Object类型

创建Object类型的实例有两种方法。第一种是使用new操作符。

var person=new Object();

person.name=”lizan”;

person.age=21;

另一种是使用对象字面量表示法，

var person={name:”lizan”,age:21};

这里属性也可以是字符串，var person={“name”:”lizan”, ”age”:21, 5:true};//这里5会自动转换为字符串

另外如果{}里面是空的，那效果和用new创建对象一样，只包含默认对象和方法。

var person={};

person.name=”lizan”;

person.age=21;

在使用对象字面量方法创建对象时，实际上不会调用Object构造函数。

访问对象属性也可以使用方括号表示法。

alert(person[“name”]); //lizan

使用方括号表示法时，也可以使用变量来访问属性，例如：

var propertyName=”name”;

alert(person[propertyName]); //lizan

## 5.2 Array类型

创建数组的方式有两种。第一种是使用Array构造函数，例如，

var colors=new Array();

colors[0] = “lizan”;

可以给构造函数传递数量和应该包含的项。

var colors=new Array(20);

var colors=new Array(“blue”,”green”,”red”);

另外也可以省略new操作符。

第二种方法是使用数组字面量方法。

var colors=[“blue”,”green”,”red”];

在使用数组字面量表示法时，也不会调用Array构造函数。

数组的length属性不是只读的，因此可以通过设置这个值来从数组末尾删除或者增加新项。

## 5.3 Date类型

要创建一个日期对象，使用new操作符和Date()构造函数即可。

var now=new Date();

## 5.4 RegExp类型

以字面量方式创建正则表达式的语法：

var expression=/pattern/flags

flags有三个值：

g:表示全局模式，即模式将被应用于所有字符串，而非在发现第一个匹配项时立即停止。

i:表示不区分大小写，

m:表示多行模式，即在到达一行文本末尾时还会查找下一行中是否存在于模式匹配的项。

例如：

//匹配所有以”at”结尾的3个字符的组合、不区分大小写。

var pattern3=/.at/gi;

另一种创建正则表达式的方式是使用RegExp构造函数。接收两个参数：一个是要匹配的字符串模式，另一个是可选的标志字符串。

正则表达式字面量始终会共享同一个RegExp实例，而使用构造函数创建的每一个RegExp实例都是一个新实例。

## 5.5 function类型

函数实际上是对象。每个函数都是Function类型的实例，而且也具有属性和方法。函数名是一个指向函数对象的指针，不会与某个函数绑定。

语法是：

function sum(num1,num2){

return num1+num2;

}

与下面使用函数表达式定义函数的方式相差无几：

var sum=function(num1,num2){

return num1+num2;

};

因为函数名是指针，所以一个函数可能会有多个名称，例如：

function sum(num1,num2){

return num1+num2;

}

var anotherSum=sum;

alert(anotherSum(10,10)); //20

使用不带圆括号的函数名是访问函数指针，而不是调用函数。

### 5.5.2 函数声明和函数表达式

alert(sum(10,10));

function sum(num1,num2){

return num1+num2;

}

上面代码可以执行。

alert(sum(10,10));

var sum=function(num1,num2){

return num1+num2;

}

但是上面的代码会产生错误。

这就是函数表达式和函数声明的区别。

### 5.5.3 作为值的函数

有一个这样的函数，如下

function callSomeFunction(someFunction,someArgument){

return someFunction(someArgument);

}

该函数的第一个参数是一个函数，第二个参数是第一个函数的参数。

function add10(num){

return num+10;

}

var result1=callSomeFunction(add,10);

alert(result1); //20

注意，第一个参数不要带括号，因为，要访问函数的指针而不执行函数的话，必须去掉函数名后面的那对圆括号。

### 5.5.4 函数内部属性

在函数内部有两个特殊的对象：arguments和this。arguments是一个类数组对象，包含着传入函数中的所有参数。

该arguments还有一个名叫callee的属性，指向拥有这个arguments对象的函数。

在递归函数中，可以将函数内部的函数名改为arguments.callee()，这样可以随便将函数赋给任何其他的变量而不影响函数功能。

另一个特殊对象是this。this引用的是函数赖以执行的环境对象，例如当在网页的全局作用域下调用函数时，this对象引用的就是window。

### 5.5.5 函数属性和方法

每个函数都包含两个属性：length和prototype。length表示接受的命名参数的个数。

最耐人寻味的是prototype属性。

对于引用类型来说，它们所有的实例方法其实都保存在prototype属性中，例如toString()和valueOf()方法都保存在prototype名下，只不过是通过各自对象的实例访问罢了。在创建自定义对象和继承中，该属性很重要。

每个函数都包含两个非继承而来的方法apply()和call()方法。

1. 首先，apply()接受两个参数：一个是在其中运行该函数的作用域(用this表示，该函数在另一个函数内，作用域就表示另一个函数，也就是this表示的作用域是另一个函数)；另一个是参数数组，可以使Array的实例也可以是arguments对象。
2. 这两个方法的用途都是在特定的作用域上调用函数，实际上等于设置函数体内this对象的值。例如，

function sum(num1,num2){

return num1+num2;

}

function callSum1(num1,num2){

return sum.apply(this,arguments); //传入arguments对象

}

function callSum2(num1,num2){

return sum.apply(this,[num1,num2]); //传入数组实例

}

alert(callSum1(10,10)); //20

alert(callSum2(10,10)); //20

从上面的例子可以看出，使用函数的apply()方法实际上相当于在另外一个函数中调用了该函数，而且其中this的值也变为了某个函数内部。

1. 在使用call()函数时，传递给函数的参数必须逐个列举出来。例如

function callSum2(num1,num2){

return sum.apply(this,num1,num2);

}

1. 事实上，这两个函数真正强大的地方是能够扩充函数赖以运行的作用域。例如，

window.color=”red”;

var o={color:”blue”};

function sayColor(){

alert(this.color);

}

sayColor(); //red

sayColor.call(this); //red

sayColor.call(window); //red

sayColor.call(o); //blue 从这里可以看出，call(o)的作用就是将sayColor的作用域定为o，所以sayColor中的this就表示的是o对象。

⑤还可以通过bind()函数来修改函数作用域

对于上述代码可以添加var objectSayColor=sayColor.bind(o); objectSayColor(); //blue

该函数的作用就是将新创建的函数的this值绑定给bind()的参数。

## 5.6 基本包装类型

每当读取一个基本类型值的时候，后台就会创建一个对应的基本包装类型的对象，从而可以调用一些方法来操作这些数据。例如，

var s1=”some text”;

var s2=s1.substring(2);

众所周知，基本数据不是对象所以不会有方法，但是当执行到第二行时，后台会进行这样一些操作：

1. 创建String类型的实例；
2. 在实例上调用指定的方法；
3. 销毁这个实例。

可以将以上三个步骤想象成执行了下列代码：

var s1=new String(“some text”);

var s2=s1.substring(2);

s1=null;

引用类型与基本包装类型的主要区别就是生存期：自动创建的基本包装类型的对象，只存在于一行代码的执行瞬间，所以不能在运行时为基本类型值添加属性和方法。例如：

var s1=”some text”;

s1.color=”red”;

alert(s1.color); //undefined

这里就是因为第二行创建的String对象在第三行时已经销毁了，第三行又创建自己的String对象，而该对象没有color属性。

### 5.6.1 Boolean类型

var booleanObject=new Boolean(true);

### 5.6.2 Number类型

var numberObject=new Number(10);

### 5.6.3 String类型

var stringObject=new String(“hello world”);

#### 1、字符方法

charAt()

#### 字符串操作方法

concat()用于将一个或多个字符串拼接起来，返回拼接得到的新字符串。

#### 3、字符串位置方法

indexOf()

#### 4、trim()

所有字符串都拥有一个trim()方法，用于返回一个去掉前后空格的字符串副本。

#### 5、字符串大小写转换方法

toLowerCase()

#### 6、字符串模式匹配方法

match()接受的参数可以是正则表达式，或者是RegExp对象。与RegExp对象的exec()方法相同。

#### 7、localeCompare()方法

### 5.7 单体内置对象

除了前面说的Object、Array和String等，还有两个单体内置对象：Global和Math

### 5.7.1 Global对象

isNaN()、parseInt()、parseFloat()实际上都是Global对象的方法，除此之外还有其他一些方法。

#### 1、URI编码方法

encodeURI()和encodeURIComponent()可以对URI进行编码

#### 2、eval()方法

#### 3、Global对象的属性

#### 4、window对象

ECMAScript虽然没有指出如何访问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(Global对象在浏览器中就是window对象)。因此在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。

var color=”red”;

function sayColor(){

alert(window.color);

}

window.sayColor(); //red

另一种取得Global对象的方法是使用以下代码：

var global=function(){

return this;

}();

以上代码创建了一个立即调用的函数表达式，返回this值。在没有给函数明确指定this值的情况下，this值等于Global。

### 5.7.2 Math对象

1、Math对象的属性

2、min()和max()方法

3、舍入方法

4、random()方法

# 第6章 面向对象的程序设计

## 6.1 理解对象

## 6.2 创建对象

### 6.2.1 工厂模式

用工厂模式创建对象。

function createPerson(name,age,job){

var o=new Object();

o.name=name;

o.age=age;

o.job=job;

o.sayName=function(){

alert(o.name);

};

return o;

}

### 6.2.2 构造函数模式

构造函数模式可用来创建特定类型的对象，例如Object和Array。也可以创建自定义的构造函数：

function Person(name,age,job){

this.name=name;

this.age=age;

this.job=job;

this.sayName=function(){

alert(this.name);

};

}

var person1=new Person("lizan",34,"programmer");

var person2=new Person("lizan",34,"programmer");

与上述的区别是：

* 没有显示的创建对象
* 直接将属性和方法赋给了this
* 没有return语句

每一个对象都有一个constructor属性，该属性指向Person。

alert(person1.constructor==Person); // ture

alert(person2.constructor==Person); // ture

1. 将构造函数当做函数

任何函数，只要通过new操作符来调用，那它就可以作为构造函数；而任何函数，如果不通过new操作符来调用，那它就是普通函数。

//作为普通函数调用

Person("Paul",24,"Software Engineer");

window.sayName(); //Paul

//在另一个对象的作用域中调用

var o=new Object();

Person.call(o,"Paul",24,"Software Engineer");

o.sayName(); //Paul

### 6.2.3 原型模式

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，这个对象的用途是包含可以由特定类型的所有实例共享的属性和方法。prototype就是通过调用构造函数而创建的那个对象实例的原型对象。使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法。

function Person(){

}

Person.prototype.name="Paul";

Person.prototype.age=24;

Person.prototype.job="Software Engineer";

Person.prototype.sayName=function(){

alert(this.name);

};

var person1=new Person();

person1.sayName(); //Paul

var person2=new Person();

person2.sayName(); //Paul

alert(person1.sayName==person2.sayName); //true

#### 理解原型对象

无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个prototype属性，这个属性指向函数的原型对象。在默认情况下，所有原型对象都会自动获得一个constructor属性，这个属性包含一个指向prototype属性所在函数的指针。就前面的例子，Person.prototype.constructor指向Person.

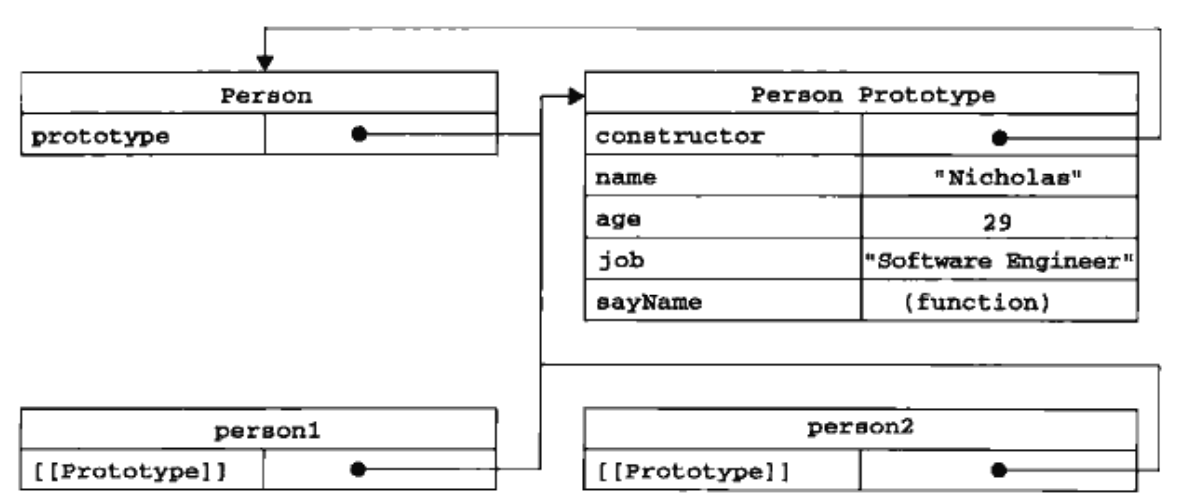
创建自定义的构造函数之后，其原型对象默认只会取得constructor属性。当调用构造函数创建一个新实例之后，该实例的内部将包含一个指针，指向构造函数原型对象。下面图示各个之间的关系。

function People(){

}

People.prototype=new Person();

alert( People.prototype.constructor==Person);//true 为原型对象赋值新对象，其constructor属性就指向了新赋值的构造函数



使用Object.getPrototypeOf()方法可以方便的取得一个对象的原型。

每当代码读取某个对象的某个属性时，都会进行搜索。首先会从对象实例本身开始，如果在对象实例中找到了给定名称的对象属性，则返回该属性值；如果没有找到，则继续搜索该对象的原型对象的属性。

如果实例对象的一个属性和其原型对象的属性同名，那么实例对象的属性就会屏蔽原型对象属性(注意这里是屏蔽并不是覆盖，原型对象的值并没有消失，只是不能访问了，即使实例对象将同名属性设为null)。可以使用delete person1.name来删除实例对象的属性，从而可以重新访问原型对象的属性。

使用hasOwnProperty()方法可以检测一个属性是在实例对象中还是在原型对象中。当存在于实例对象中时返回true，

#### 2、原型与in操作符

无论属性是存在实例对象中还是原型对象中，”name” in person1都会返回true。

在使用for-in循环时，返回的是所有能够通过对象访问的、可枚举的属性，其中既包括存在于实例中的属性又包括存在于原型中的属性。

var o={

toString:function(){

return "My Object";

}

};

for(var prop in o){

if(prop=="toString"){

alert("Found toString");

}

}

#### 3、更简单的原型语法

使用对象字面量：

function Person(){

}

Person.prototype={

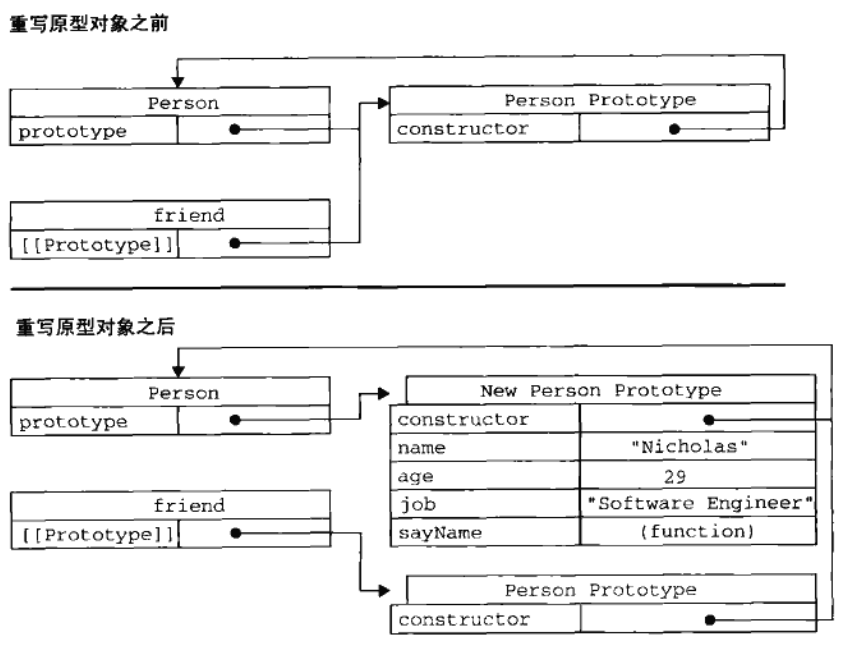
name:”Paul”,

age:29,

};

注意：以这种方式创建的原型对象，其constructor属性不在指向Person了。

使用这种方法本质上完全重写了默认的prototype对象，因此constructor属性也就变成了新对象的constructor属性(指向Object构造函数)，不再指向Person函数。



#### 5、原生对象的原型

#### 6、原型对象的问题

### 6.2.4 组合使用构造函数模式和原型模式

### 6.2.5 动态原型模式

### 6.2.6 寄生函数构造模式

## 6.3 继承

### 6.3.1 原型链

核心概念就是为一个对象的原型对象赋值为一个新对象实例，这样就实现了继承新对象。示例代码：

function SuperType(){

this.property=true;

}

SuperType.prototype.getSuperValue=function(){

return this.property;

};

function SubType(){

this.subproperty=false;

}

SubType.prototype=new SuperType();

SubType.prototype.getSubValue=function(){

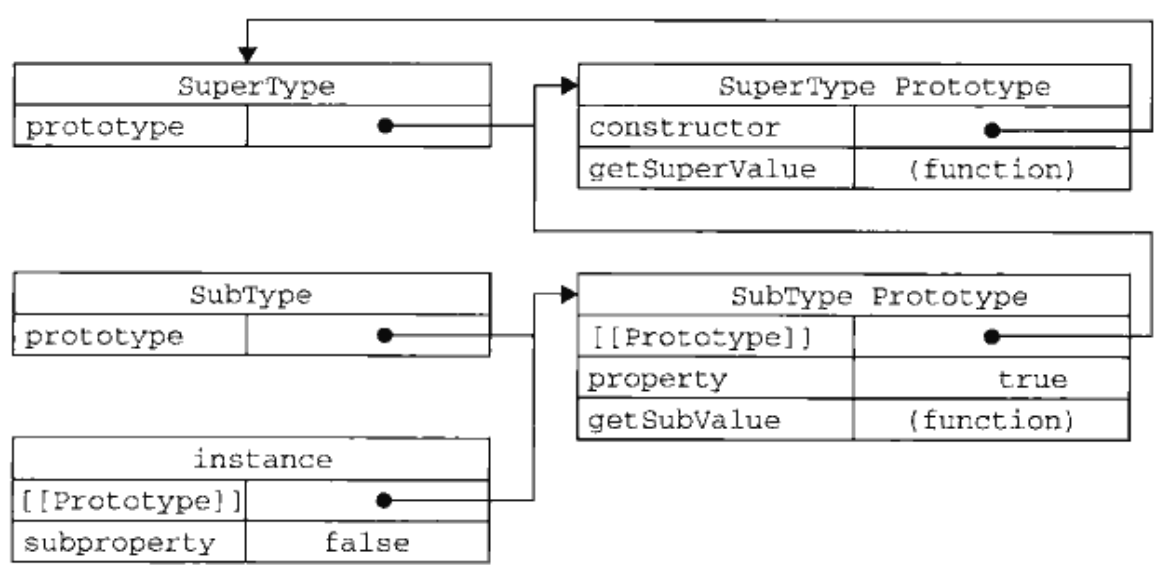
return this.subproperty;

};

var instance=new SubType();

alert(instance.getSuperValue());

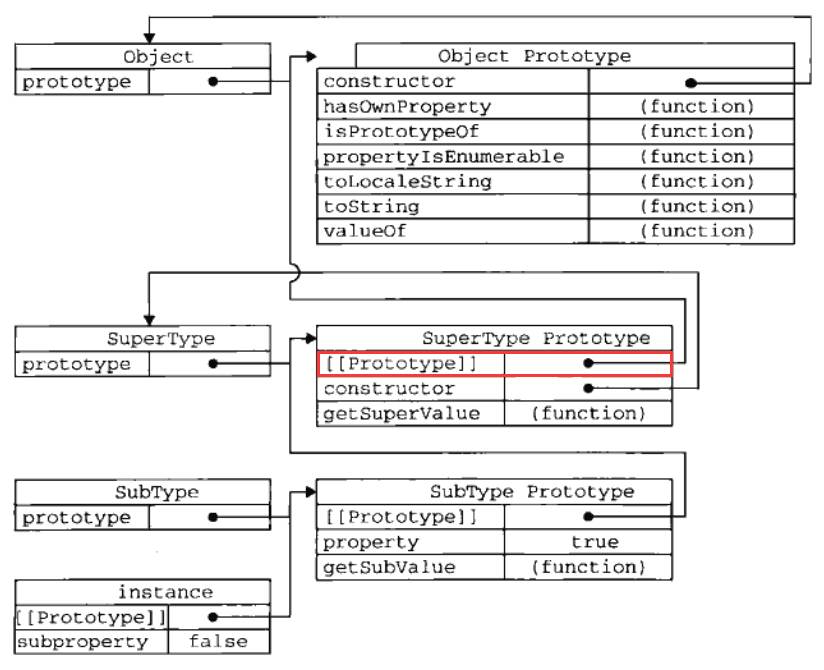
子类型的原型对象直接指向了父类型的实例，而父类型的实例又指向父类型的原型对象，所以子类型的实例可以访问所有父类型的属性和方法。关系图为：



instance.constructor指向为SuperType()

#### 1、别忘记默认的原型

所有函数的默认原型都包含一个内部指针，指向Object.prototype。这也正是所有自定义类型都会继承toString()、valueOf()等默认方法的根本原因。所以上面例子展示的原型链中还应该包含另外一个继承层次，如下：



#### 2、确认原型和实例的关系

可以通过两种方式来确定原型和实例之间的关系。

第一种是使用instanceof操作符，只要用这个操作符来测试实例与原型链中出现过的构造函数，结果就返回true。例如

alert(instance instanceof Object); //true

alert(instance instanceof SuperType); //true

alert(instance instanceof SubType); //true

第二种方法是使用isPrototypeOf()方法。

alert(Object.prototype.isPrototypeOf(instance)); //true

alert(SuperType.prototype.isPrototypeOf(instance)); //true

alert(SubType.prototype.isPrototypeOf(instance)); //true

#### 3、谨慎地定义方法

给原型添加方法的代码一定要放在替换原型之后。

function SuperType(){

this.property=true;

}

SuperType.prototype.getSuperValue=function (){

return this.property;

};

function SubType(){

this.subproperty=false;

}

//继承了SuperType

SubType.prototype=new SuperType();

//添加新方法

**SubType.prototype.getSubValue=function(){**

**return this.subproperty;**

**};**

//重写超类中的方法

**SubType.prototype.getSuperValue=function(){**

**return false;**

**};**

对于第二个加粗代码的理解：新的属性与SuperType对象原型中的属性同名，这样SubType的实例在访问其原型对象的属性时(采取向上搜索)，就不会搜索到SuperType原型对象中，所以其原型对象的属性相当于屏蔽。但是新的SuperType对象实例还是可以搜索到原来原型对象的方法的。

还有一点需要注意，在通过原型链实现继承时，不能使用对象字面量来创建原型方法，因为这样会重写原型链。

例如：

SubType.prototype={

getSubValue:function(){

return this.subproperty;

},

someOtherMethod:function(){

return false;

}

};

var instance=new SubType();

alert(instance.getSuperValue()); //error

这样的话SubType的原型对象就是一个Object实例，而不是SuperType实例，这样原型链就会断掉————SubType和SuperType之间已经没有关系了。

### 6.3.2 借用构造函数

function SuperType(){

this.colors=["red","blue","green"];

}

function SubType(){

//继承SuperType

SuperType.call(this); //将SuperType的执行环境放到了SubType里

}

var instance1=new SubType();

instance1.colors.push("black");

alert(instance1.colors); //red,blue,green,black

var instance2=new SubType();

alert(instance2.colors); //red,blue,green

通过上面的继承方法和下面的两个输出结果，可以看出，每一个SubType的实例都会有一个自己的colors属性的副本。

#### 传递参数

#### 2、借用构造函数的问题

### 6.3.3 组合继承

指将原型链和借用构造函数的方法组合在一起。

### 6.3.4 原型式继承

### 6.3.5 寄生式继承

### 6.3.6 寄生组合式继承

### 6.3.7 JavaScript中的内存模型

function Person(id,name,age){

this.id = id;

this.name = name;

this.age = age;

}

var num = 10;

var bol = true;

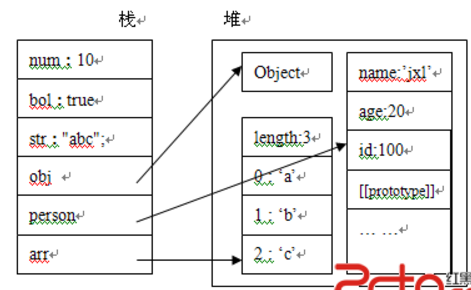
var str = "abc";

var obj = new Object();

var arr = ['a','b','c'];

var person = new Person(100,"jxl",20);

对于上述代码，内存模型图为：



# 第7章 函数表达式

## 7.1 递归

arguments.callee在哪一个函数中运行就表示哪一个函数。例如，用匿名函数实现递归，没有函数名不好实现就可以用arguments.callee来表示函数名。

## 7.2 闭包

闭包是指有权访问另一个函数作用域中的变量的函数。

闭包的特点：

闭包函数返回时，该函数内部变量处于激活状态，函数所在栈区依然保留，内存中依然有该函数所在区域，生命周期没有停止。

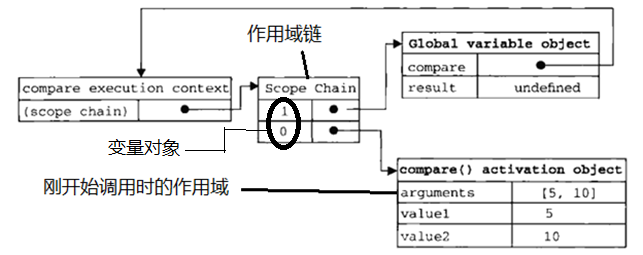
##### 函数第一次调用时会发生什么？

当某个函数第一次被调用时，会创建一个执行环境(execution context)及相应的作用域链，并把作用域链赋值给一个特殊的内部属性，然后使用this，arguments和其他命名参数的值来初始化函数的活动对象(activation object)。在作用域链中，外部函数的活动对象始终处于第二位，外部函数的外部函数的活动对象处于第三位，。。。。。。直至作为作用域链终点的全局执行环境。

function compare(value1,value2){//代码省略}

var result=compare(5,10);

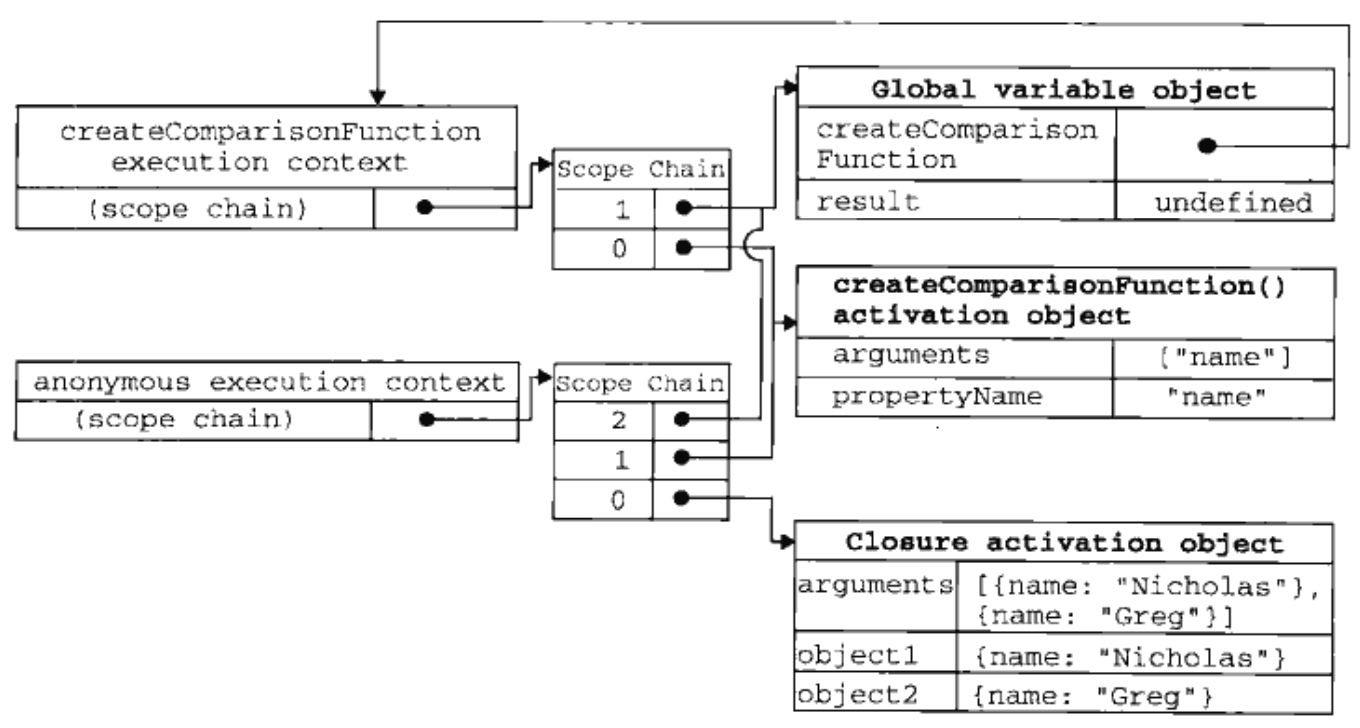
当第一次调用compare()函数时，会创建一个包含this,arguments,value1和value2的活动对象。全局执行环境的变量对象(包含this,result和compare)在compare()执行环境的作用域链中则处于第二位。如下图：



后台的执行环境都有一个表示变量的对象--------变量对象。全局环境的变量对象始终存在，而像compare()函数这样的局部环境的变量对象，则只在函数执行的过程中存在。

在创建compare()函数时，会创建一个预先包含全局变量对象的作用域链，这个作用域链会保存在内部的[[scope]]属性中。当调用compare()函数时，会为函数创建一个执行环境，然后通过复制[[scope]]属性中的对象构建起执行环境的作用域链。此后，又有一个活动对象(在此作为变量对象使用)被创建并被推入执行环境作用域链前端。对于这里的compare()的执行环境而言，其作用域链中包含两个变量对象：本地活动对象和全局变量对象。显然，作用域链本质上是一个指向变量对象的指针列表，它只引用但不实际包含变量对象。

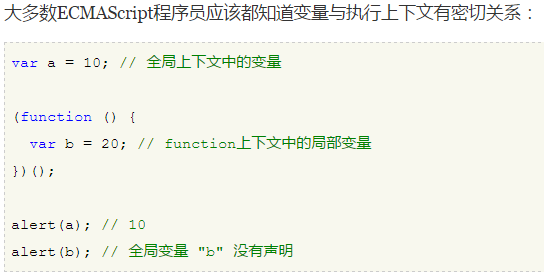
有内部函数的情况：



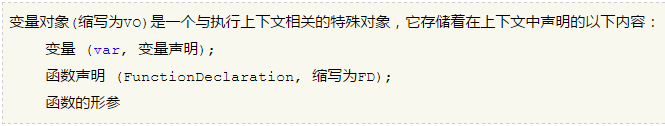
一般来讲，当函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域，但是闭包不是这样。(从最后半句可以看出闭包其实就是包含外部函数变量的匿名函数)

变量对象：

执行上下文与变量的关系：



如果变量与执行上下文相关，那变量应该知道它的数值应该存储在哪里，并且知道如何访问。这种机制称为变量对象



### 7.2.1 闭包与变量

作用域链的这种机制存在一种副作用，即闭包只能取得包含函数中任何变量的最后的值，例如下面这个例子：

function createFunctions(){

var result=new Array();

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

**result[i]=function(){**

**return i;**

**};**

}

return result;

}

函数说明---🡪注意加粗部分表示的是为每一个数组元素赋函数(并不是赋i)，当我真正调用内部函数(代码为createFunctions()[3]())，才会为第四个元素赋值，而这时闭包里面的i已经变成10(因为当调用createFunctions()时，里面的i已经变成10)。

这个例子所返回的每一个数组的值都是10。

可以使用如下的例子来解决上面的问题：

function createFunctions(){

var result=new Array();

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

**result[i]=function(num){**

**return function (){**

**return num;**

**};**

}(i); // 函数定义后面加括号表示立即调用，括号里面的值为传的参数。这样立即执行就会返回真正的值

// 给数组元素。

}

return result;

}

### 7.2.2 关于this对象

var name="The Window";

var object={

name:"The Object",

getNameFunc:function (){

var name="The Function";

return function (){

return this.name;

};

}

};

alert(object.getNameFunc()()); //The Window

这里this存在于内部函数中，该函数并非直接是object对象的成员，这样this就成为全局对象。

匿名函数的执行环境具有全局性。

相当于全局环境调用闭包，所以返回The Window。

对于上述代码有这样一条语句(object.getNameFunc=object.getNameFunc)(); // The Window。相当于在全局环境中调用函数。

内部函数在搜索this和arguments这两个变量时，根据前面的作用域链，永远也不会直接访问外部函数中的这两个变量，因为其本身就有这两个变量。

### 7.2.3 内存泄漏

## 7.3 模拟块级作用域

## 7.4 私有变量

### 7.4.1 静态私有变量

### 7.4.2 模块模式

### 7.4.3 增强的模块模式

# 第8章 BOM

## 8.1 window对象

BOM核心对象是window，它表示浏览器的一个实例。在浏览器中window是ECMAScript规定的Global对象。

### 8.1.1 全局作用域

所有在全局作用域中声明的变量、函数都会变成window对象的属性和方法。

var age=29;

function sayAge(){

var age=28;

alert(this.age);

}

alert(window.age); //29

sayAge();//29 因为sayAge()是在全局作用域中调用的，所以会输出29而不是28

window.sayAge();//29

定义全局变量与直接在window对象上定义属性还是有一点差别的：全局变量不能通过delete操作符删除，而直接在window对象上定义的操作符可以。

还要记住一件事：尝试访问未声明的变量会抛出错误，但是通过查询window对象，可以知道某个未声明的变量是否存在。例如：

//抛出异常，因为oldValue未声明

var newValue=oldValue;

//这里不会抛出异常，因为这是一次属性查询

var newValue=window.oldValue;

### 8.1.2 窗口关系及框架

如果页面中包含框架，则每个框架都拥有自己的window对象，并且保存在frames集合中。每个window对象都有一个name属性，其中包含框架的名称。

<html>

<head>

<title>Frameset Example</title>

</head>

<frameset rows="160,\*">

<frame src="frame.htm" name="topFrame">

<frameset cols="50%,50%">

<frame src="anotherframe.htm" name="leftFrame">

<frame src="yetanotherframe.htm" name="rightFrame">

</frameset>

</frameset>

</html>

top对象始终指向最高层框架，也就是浏览器窗口。

parent对象始终指向当前框架的直接上层框架。在某些情况下，parent有可能等于top；但在没有框架的情况下，parent一定等于top(此时他们都等于window)。

注意：除非最高层窗口是window.open()打开的，否则其window对象的name属性不会包含任何值。

与框架有关的最后一个对象是self，它始终指向window。self和window可以互换使用。

所有这些对象都是window对象的属性，可以通过window.parent、window.top等形式来访问。

### 8.1.3 窗口位置

IE、Safari、Opera和Chrome都提供了screenLeft和screenTop属性，分别用于表示窗口相对于屏幕左边和上边的位置。

Firefox、Safari和Chrome也支持screenX和screenY属性。

可以使用下面的代码跨浏览器取得窗口左边和上边的位置。

**var** leftPos=(**typeof** window.screenLeft=="number")?window.screenLeft:window.screenX;  
**var** topPos=(**typeof** window.screenTop=="number")?window.screenTop:window.screenY;

moveTo()和moveBy()方法

注意：这两个方法可能被浏览器禁用；而且在Opera和IE7中默认就是禁用的。另外这两个框架都不适用于框架，只能对最外层对象使用。

### 8.1.4 窗口大小

resizeTo()和resizeBy()方法可以调整浏览器窗口的大小。

### 8.1.5 导航和打开窗口

使用window.open()方法既可以导航到一个特定的URL，也可以打开一个新的浏览器窗口。接受四个参数：

1. 要加载的URL
2. 窗口目标
3. 一个特性字符串
4. 一个表示新页面是否取代浏览器历史记录中当前加载页面的布尔值。

通常只传递第一个参数，最后一个参数只在不打开窗口的情况下使用。

window.open("http://www.baidu.com","topFrame");

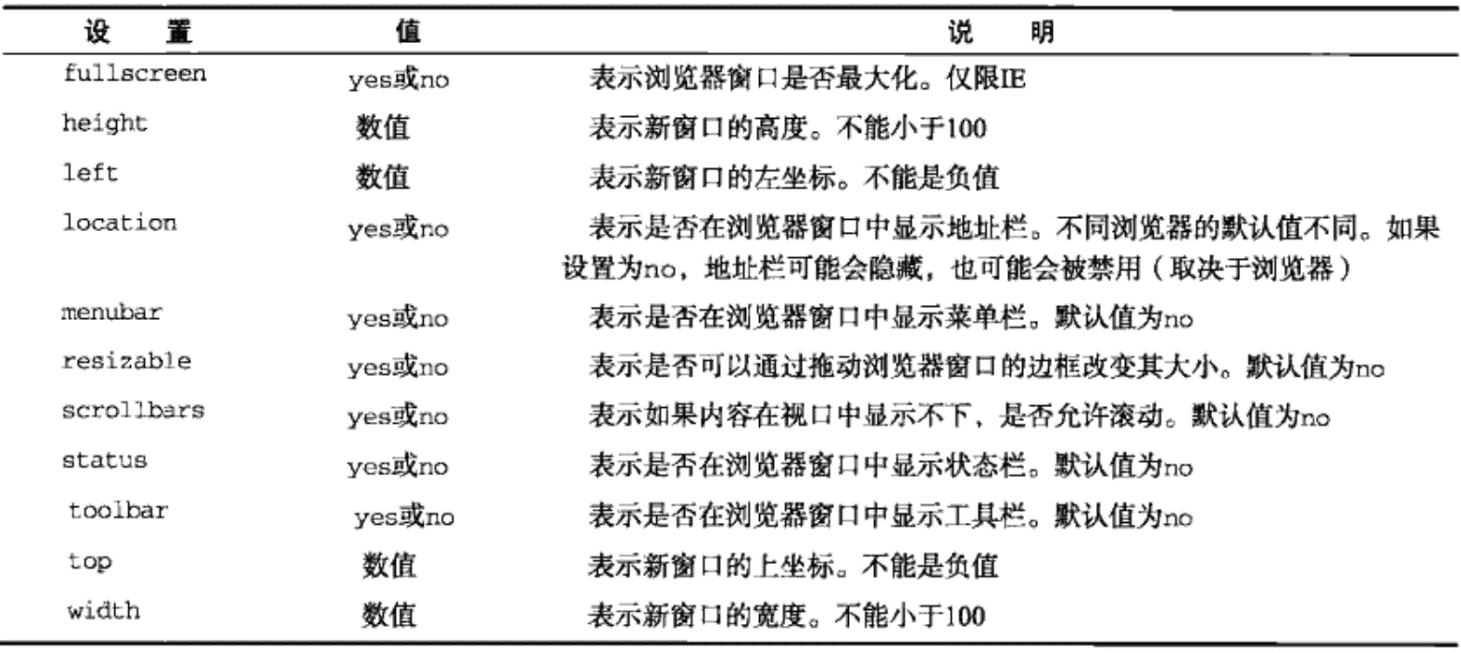
等同于<a href=”<http://www.baidu.com>” target=”topFrame”></a>

如果有一个名叫”topFrame”的窗口或者框架，就会在该窗口或者框架加载这个URL;否则，就会创建一个新窗口将其命名为”topFrame”;此外，第二个参数也可以是下列任何一个特殊的窗口名称：\_self、\_parent、\_top或\_blank。

#### 弹出窗口

如果第二个参数不是一个已经存在的窗口或框架，那么就会根据第三个参数创建一个新窗口或标签页。如果没有第三个参数，那么就会打开一个默认的窗口。

第三个参数是一个逗号分隔的设置字符串，表示在新窗口中都有哪些特性。下表表示第三个参数可以设置的选项：



#### 弹出窗口屏蔽程序

如果是浏览器内置的屏蔽程序阻止的弹出窗口，那么window.open()很可能会返回null。此时只要检测这个返回的值就可以确定弹出窗口是否被屏蔽。

如果是浏览器扩展或者是其他程序阻止的弹出窗口，那么window.open()通常会抛出一个错误。这里可以将window.open()封装到一个try-catch块中。

### 8.1.6 间歇调用与超时调用

JavaScript是单线程语言。

前者是在指定的时间过后执行代码，后者则是每隔指定的时间就执行一次代码。

超时调用需要使用window对象的setTimeout()方法。它接收两个参数：要接收的代码和以毫秒表示的时间。其中第一个参数可以是一个包含JavaScript代码的字符串(就像和eval()函数中使用的字符串一样)，也可以是一个函数。

经过指定的时间之后，指定的代码并不一定执行------JavaScript是一个单线程解释器，因此一段时间之内只能执行一段代码。为了控制要执行的代码，就有一个JavaScript任务队列，第二个参数告诉JavaScript再过多长时间把代码加到任务队列中。如果队列是空的，那么添加的代码会立即执行，如果不是空的，那么就要等到前面代码执行完成之后再执行。

||🡺调用setTimeout()之后，该方法会返回一个数值ID，表示超时调用。这个超时调用ID是计划执行代码的唯一标识符，可以通过它来取消超时调用。可以调用clearTimeout(ID)其取消超时调用。只要是在指定的时间尚未过去之前调用clearTimeout(ID)，就可以完全取消超时调用。

setInterval()设置间隔时间调用。用法和setTimeout()类似，也会返回一个ID，使用clearInterval(ID)

### 8.1.7 系统对话框

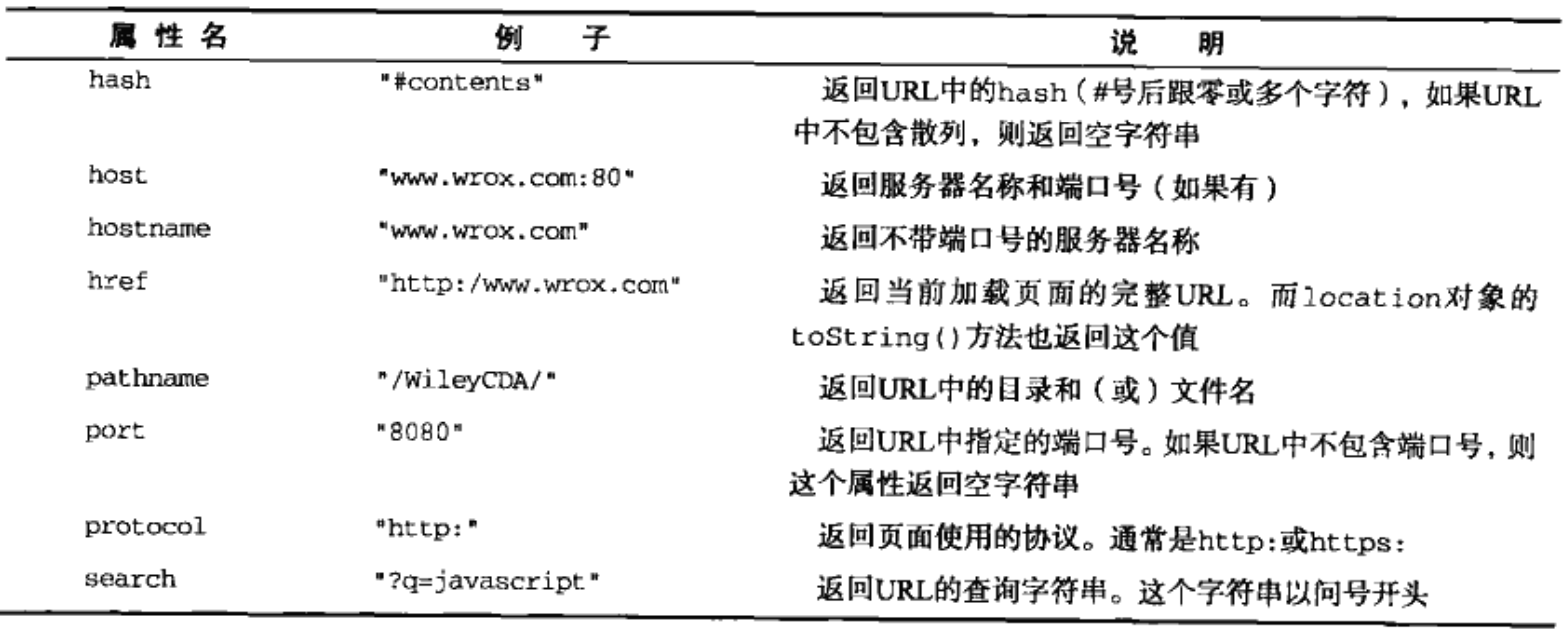
使用alert()、confirm()和prompt()可以调用系统对话框向用户调用信息。系统对话框与浏览器中的网页没有关系，也不包含HTML。通过这些方法打开的对话框都是同步和模态的，即显示这些对话框的时候，代码会停止执行，关掉这些对话框后代码又继续执行。

confirm()方法显示的对话框包含两个按钮：ok和cancel。根据用户点击的按钮不同，该方法返回的值也不同：返回true表示点击ok按钮，返回false表示点击cancel或者右上角的关闭按钮。

最后一种对话框是通过prompt()方法生成的。除了显示OK和cancel按钮之外还会显示一个文本输入域。该方法接受两个参数：第一个参数是提示信息，第二个参数是文本输入框的默认值(可以是空字符串)。

## 8.2 location对象

location对象即使window对象的属性也是document对象的属性。window.location和document.location引用的是同一个对象。location对象的用处不只表现在它保存着当前文档的信息，还表现在将URL解析为独立的片段，让开发人员可以通过不同的属性访问这些片段。下表列出了location对象的属性：



### 8.2.1 查询字符串参数

尽管location.search返回URL从问号到末尾的所有内容。但却没有办法逐个访问其中的每个查询参数。

### 8.2.2 位置操作

使用location对象可以通过方式来改变浏览器的位置。首先，也是最常用的方式，就是通过assign()方法并为其传递一个URL，例如，location.assign(“https://www.baidu.com”);这样就可以立即打开新的URL。

使用location.href或者window.location来改变URL也是调用assign()方法。例如，下面两行代码与调用assign()效果一样，window.location=” <https://www.baidu.com>”; location.href=” <https://www.baidu.com>”; 最常用的是location.href属性。

🡪通过location.replace()方法可以禁止通过点击”后退”按钮来回到前一个页面。

🡪reload()方法的作用是重新加载当前页面。如果调用该函数时不添加任何参数，页面就会以最有效的方式加载，也就是说如果页面自上次加载以来从没有改变过，页面就会从浏览器缓存中重新加载。如果要强制从服务器重新加载，需要传递参数true。

注意：位于reload()之后的代码可能会也可能不会执行，这取决于网络延迟或系统资源等因素。为此，最好将代码放在最后一行。

## 8.3 navigator对象

navigator对象是识别客户端浏览器的事实标准，是所有支持JavaScript浏览器所共有的。

### 8.3.1 检测插件

检测浏览器中是否安装了特定的插件，对于非IE浏览器来说，可以使用plugins数组来达到这个目的，该数组的每一项都包含下列属性：

①name：插件的名字

②description：插件的描述

③filename：插件的文件名

④length：插件所处理的MIME类型数量

### 8.3.2 注册处理程序

## 8.4 screen对象

## 8.5 history对象

history对象保存着用户上网的历史记录，从窗口被打开的那一刻起。因为history是window对象的属性，因此每一浏览器窗口、每个标签页乃至每个框架都有自己的history对象与特定的window对象关联。

使用go()方法可以在浏览器的历史记录中任意跳转。该方法接收一个参数，表示向后或者向前跳转的页面数的一个整数。负数表示向后跳转，正数表示向前跳转。

也可以传递一个字符串参数，此时浏览器会跳转到历史记录中包含该字符串的第一个位置，例如，

//跳转到最近的wrox.com页面

history.go(“wrox.com”);

另外还可以使用两个简写方法back()和forward()来代替go()。可以模仿浏览器的后退和前进按钮。

history还有一个length属性，保存着浏览器的历史记录数量。

对于加载到窗口、标签页或框架中的第一个页面而言，history.length等于0，可以用这个特性来检测用户是不是第一次就打开你的网页。

# 第9章 客户端检测

## 9.1 能力检测

就是查看浏览器支不支持某个属性。例如，if(object.propertyName){//使用object.propertyName}

### 9.1.2 能力检测，不是浏览器检测

!!作用就是强制将数据转换为布尔值。例如，var a; var b=!!a; //这里a为undefined，!a=true，!!a=false;

## 9.2 怪癖检测

## 9.3 用户代理检测

通过检测用户代理字符串来确定是哪一种浏览器。该字符串可以通过navigator.userAgent属性来访问。在客户端，用户代理检测一般被当做一种万不得已使用的做法，其优先级排在能力检测和怪癖检测之后。

用户代理的争议性与电子欺骗有关，即在用户代理字符串中加入一些错误或误导性信息来达到欺骗浏览器的目的。

### 9.3.1 用户代理字符串历史

# 第10章 DOM

## 10.1 节点层次

### 10.1.1 Node类型

### 10.1.2 Document类型

JavaScript通过Document类型表示文档。在浏览器中，document对象表示整个HTML页面，而且document对象是window对象的一个属性。

#### 1、文档的子节点

🡪documentElement属性，该属性始终指向HTML页面中的<html>元素。

🡪还有一个body属性，直接指向<body>元素

所有浏览器都支持上述两个属性。

#### 文档信息

🡪title属性，页面的标题

🡪URL属性，页面地址栏中显示的完整URL

🡪domain属性，只包含页面的域名

🡪referrer属性，保存着链接到当前页面的那个页面的URL。

所有的这些信息都存在于请求的http头部。

在三个属性中，只有domain可以设置，但是，并不是可以设置任何值。

#### 3、查找元素

getElementById()、getElementsByTagName()和getElementsByName()

第二个方法返回一个数组，例如下面的代码：

var images=document.getElementByTagName(“img”);

可以使用方括号或item()方法来访问数组中的项。这个对象中元素的数量可以通过其length属性取得。

alert(images.length);

alert(images[0].src);

alert(images.item(0).src);

还有一个方法叫做namedItem()，使用该方法可以通过元素的name值来取得集合中的项。例如，

var myImage=images.namedItem(“myImage”);或者var myImage=images.namedItem[“myImage”];

要想取得页面中的所有元素，可以使用getElementsByTagName(“\*”);

#### 4、特殊集合

document.links🡪包含文档中所有带href的<a>元素

#### DOM一致性检测

#### 6、文档写入

将输出流写入到网页中的能力，这个能力体现在4个方面：write()、writeln()、open()和close()

writeln()方法会在字符串的末尾添加一个换行符”\n”。

在write()和writeln()中不能直接包含”</script>”，这是因为该字符串会被解释为脚本块的结束，应该写成”<\/script>”

如果在文档加载结束后再调用document.write()，那么输出的内容将会重写整个页面。例如

<script>

window.onload=function(){

document.write(“Hello World!!!”);

};

</script>

方法open()和close()分别用来打开和关闭网页的输出流。如果write()和writeln()是在页面加载期间被调用，则不需要用到这两个方法。

### 10.1.3 Element类型

要访问元素的标签名可以使用nodeName属性和tagName属性，这两个属性会返回相同的值。

**var *div***=**document**.getElementById(**"myDiv"**);  
alert(***div***.**tagName**);*//”DIV”*alert(***div***.**tagName**==***div***.**nodeName**);*//true*

在HTML中标签名称始终以大写表示

所有的HTML元素都有如下的特性：

①id:

②className:与元素的class特性对应，没有将该属性命名为class是因为class是JavaScript的保留字

③title：有关元素的附加说明信息，一般通过工具提示条显示出来。鼠标移动到元素之上会显示出来

上述这些属性都可以用来取得或修改相应的属性值。

#### 2 取得特性

操作特性的DOM方法主要有三个：getAttribute()、setAttribute()和removeAttribute()。这三个方法可以针对任何特性使用。

有两类特殊的特性，它们虽然有对应的属性名，单属性值与通过getAttribute()返回的值并不相同。

🡪第一类特性是style。通过getAttribute()访问时返回的style特性值中包含的是CSS文本；而通过属性来访问它会返回对象。

🡪第二个特性是onclick这样的事件处理程序：如果通过getAttribute()访问，则会返回相应代码的字符串。而在访问onclick属性时，则会返回一个JavaScript函数。

#### 3、设置特性

setAttribute()接受两个参数要设置的特性名和值。如果特性已经存在，则该函数会以指定的值替换现有的值；如果特性不存在setAttribute()则创建该属性并设置相应的值。

#### 4、attributes属性

#### 5、创建元素

使用document.createElement()方法可以创建新元素。这个方法只接收一个参数，即要创建元素的标签名。

### 10.1.4 Text类型

### 10.1.5 Comment类型

### 10.1.7 DocumentType类型

## 10.2 DOM操作技术

### 10.2.1 动态脚本

### 10.2.2 动态样式

### 10.2.3 操作表格

# 第11章 DOM扩展

## 11.1 选择符API

### 11.1.1 querySelector()方法

querySelector()方法接收一个CSS选择符，返回与该模式匹配的第一个元素，如果没有找到匹配的元素则返回null。

//取得body元素  
**var** body=document.querySelector("body");  
//取得id为"myDiv"的元素  
**var** myDiv=document.querySelector("#myDiv");  
//取得类为"selected"的第一个元素  
**var** selected=document.querySelector(".selected");  
//取得类为"button"的第一个图像元素  
**var** img=document.body.querySelector("img.button");

通过Document类型调用querySelector()方法时，会在文档元素的范围内查找匹配的元素。而通过Element类型调用querySelector()方法时，只会在该元素后代元素的范围内查找匹配元素。

### 11.1.2 querySelectorAll()方法

该方法返回的是所有匹配的元素而不是一个元素。

### 11.1.3 matchesSelector()方法

这个方法接收一个CSS选择符，如果调用元素与该选择符匹配，返回true；否则，返回false。

## 11.2 元素遍历

## 11.3 HTML5

## 11.4 专有扩展

### 11.4.1 文档模式

IE8引入了一个新的概念叫“文档模式”，文档模式决定了你可以使用哪个级别的CSS，可以在JavaScript中使用哪些API，以及如何对待文档类型。

要强制浏览器以某种模式渲染页面，可以使用HTTP头部信息X-UA-Compatible，或通过等价的meta标签来设置：

<meta http-equiv=”X-UA-Compatible” content=”IE=IEVersion”>

Edge版本表示始终以最新的文档模式来渲染页面

### 11.4.2 children属性

### 11.4.3 contains()方法

### 11.4.4 插入文本

### 11.4.5 滚动

# 第12章 DOM2和DOM3

## 12.1 DOM变化

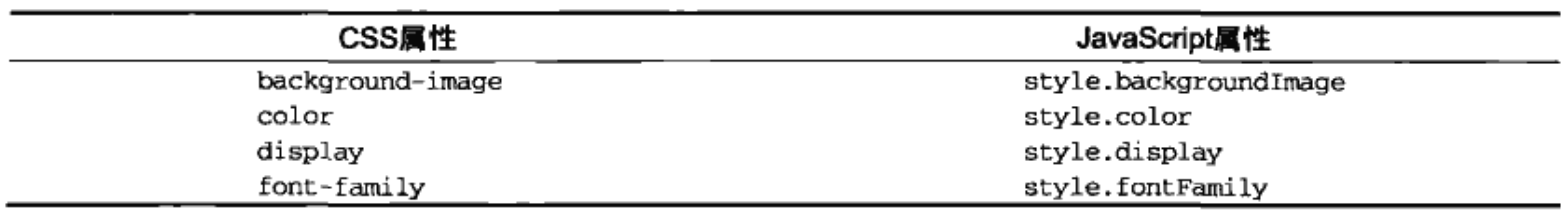
### 12.1.1 针对XML命名空间的变化

### 12.1.2 其他方面的变化

## 12.2 样式

### 12.2.1 访问元素的样式

任何支持style特性的HTML元素在JavaScript中都有一个对应的style属性。对于使用端划线的CSS属性名，必须将其转换成驼峰大小写形式，才能通过JavaScript来访问。



其中一个不能直接转换的CSS属性是float。由于float是JavaScript中的保留字，因此不能用作属性名，对应的相应的属性名应该是cssFloat;

### 12.2.2 操作样式表

### 12.2.3 元素大小

## 12.3 遍历

## 12.4 范围

# 第13章 事件

事件，就是文档或浏览器窗口中发生的一些特定的交互瞬间。可以使用侦听器(或处理程序)来预订事件，以便事件发生时，执行相应的代码。

## 13.1 事件流

在开始之前首先要搞清楚一个问题：如果单机了一个按钮，那么单机事件不仅仅发生在一个按钮上。换句话说，在单机按钮的同时，你也单机了按钮的容器元素，甚至也单机了整个页面。

事件流描述的是从页面中接受事件的顺序。

### 13.1.1 事件冒泡

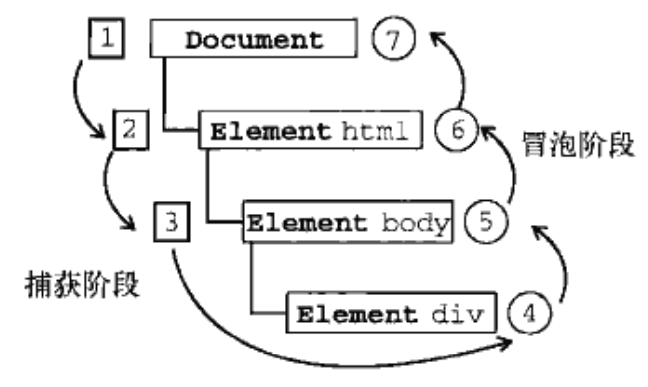
IE的事件流叫事件冒泡。即时间开始时由最具体的元素接收，然后逐级向上传播到较为不具体的节点(文档)。

### 13.1.2 事件捕获

事件捕获的思想是不太具体的节点应该更早接收到事件。事件捕获的用意在于在事件未到达预定节点之前捕获他。

### 13.1.3 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。首先发生的是事件捕获，为截获事件获得了机会。然后是实际的目标接收到事件。最后阶段是事件冒泡阶段。可以在这个阶段对事件做出响应。



## 13.2 事件处理程序

事件处理程序的名字以“on”开头，因此click事件的事件处理程序的名字就是onclick，load事件的事件处理程序的名字就是onload。

### 13.2.1 HTML事件处理程序

某个元素支持的每种事件，都可以使用与相应事件处理程序同名的HTML特性来指定。这个特性的值应该是能够执行的JavaScript代码。例如，要在按钮被单机时执行JavaScript代码，可以向下面这样编写代码：

<input type=”button” value=”Click Me” onclick=”alert(‘Clicked’)”>

当然也可以调用某个函数。这样指定事件处理程序具有一些独到之处。首先，这样会创建一个封装着元素属性值的函数。这个函数中有一个局部变量event，也就是事件对象。

<input type=”button” value=”Click Me” onclick=”alert(event.type)”>

通过event变量，你可以直接访问事件对象，你不用自己定义它。

在这个函数中，this值等于事件的目标元素。<input type=”button” value=”Click Me” onclick=”alert(this.value)”>

在HTML中指定事件处理程序有两个缺点：

①存在时差问题。即事件处理程序代码还没有加载出来，用户就点击了按钮，这样会报错。解决方法就是将事件处理程序放在try-catch块中，这样浏览器有机会处理错误之前，错误就会被捕获，用户将不会看到JavaScript错误。

②HTML代码与JavaScript代码紧密耦合，所以很多程序员选择使用JavaScript指定事件处理程序。

### 13.2.2 DOM0级事件处理程序

通过JavaScript指定事件处理程序的传统方式就是将一个函数赋值给一个事件处理程序属性。

每个元素都有自己的事件处理程序属性。这些属性通常全部小写，例如onclick。将这种属性的值设置为一个函数，就可以指定事件处理程序，如下所示：

**var** btn=document.getElementById("myBtn");  
btn.onclick=**function** (){  
 alert(**this**.id);//myBtn  
};

也可以删除事件处理程序：btn.onclick=null;//只需要将事件处理程序属性的值设置为null即可。

### 13.2.3 DOM2级事件处理程序

“DOM2级事件”定义了两个方法，用于处理指定和删除事件处理程序的操作：addEventListener()和removeEventListener()。

所有DOM节点中都包含这两个方法，并且它们都接受三个参数：要处理的事件名(click等)，作为事件处理程序的函数和一个布尔值。最后这个布尔值参数如果是true，表示在捕获阶段调用事件处理程序；如果是false，表示在事件冒泡阶段调用事件处理程序。

要在按钮上为click事件添加事件处理程序，可以使用下列代码：

**var** btn=document.getElementById("myBtn");  
btn.addEventListener("click",**function**(){  
 alert(**this**.id);  
},**false**)

使用这种方法可以添加多个事件处理程序。多个事件处理程序会按照添加他们的顺序触发。

通过addEventListener()方法添加的事件处理程序只能通过removeEventListener()来移除。移除时添加的参数要与添加时出入的参数相同，所以添加的匿名函数无法移除。

btn.removeEventListener("click",**function**(){//没有用  
 alert(**this**.id);  
},**false**);

### 13.2.4 IE事件处理程序

## 13.3 事件对象

在触发DOM上的某个事件时，会产生一个事件对象event，这个对象中包含着所有与事件相关的信息。包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。例如鼠标操作导致的事件对象中，会包含鼠标位置的信息，而键盘操作导致的事件对象中，会包含与按下的键有关的信息。所有浏览器都支持event对象，但支持方式不同。

### 13.3.1 DOM中的事件对象

浏览器会将一个event对象传入到事件处理程序中。无论使用事件处理程序时使用什么方法(DOM0或DOM2)，都会传入event对象。例如：

**var** btn=document.getElementById("myBtn");  
btn.onclick=**function** (event){  
 alert(event.type);//click  
};  
  
**var** btn=document.getElementById("myBtn");  
btn.addEventListener("click",**function**(event){  
 alert(event.type);//click  
},**false**);

在通过HTML特性指定事件处理程序时，变量event中保存着event对象。例如下面的例子：

<input type="button" value="Click Me" onclick="alert(event.type)">

在事件处理程序内部，对象this始终等于正在处理事件的那个元素即event对象的currentTarget属性。

**var** btn=document.getElementById("myBtn");  
btn.onclick= **function** (event) {  
 alert(event.currentTarget===**this**);//true  
 alert(event.target===**this**);//true  
}

如果事件处理程序存在于body元素中但是我点击body中的某个按钮，则会有下列结果：

document.body.onclick= **function** (event) {  
 alert(event.currentTarget===document.body);//true  
 alert(**this**===document.body);//true  
 alert(event.target===document.getElementById("myBtn"));//true  
}

从这里可以看出currentTarget、target和this在事件处理程序中各表示什么：this和currentTarget都表示事件处理程序注册的那个元素，target表示事件真正的目标。

在需要一个函数处理多个事件时，可以使用type属性，如下：

**var** btn=document.getElementById("myBtn");  
**var** handler=**function**(event){  
 **switch** (event.type){  
 **case** "click":  
 alert("Clicked");  
 **break**;  
 **case** "mouseover":  
 event.target.style.backgroundColor="red";  
 **break**;  
 **case** "mouseout":  
 event.target.style.backgroundColor="green";  
 **break**;  
 }  
}  
btn.onclick=handler;  
btn.onmouseover=handler;  
btn.onmouseout=handler;

要阻止特定事件的默认行为，可以使用preventDefault()方法。例如阻止链接转到特定的URL：

btn.onclick=**function**(event){  
 event.preventDefault();  
}

只有cancelable属性设置为true的事件，才可以使用preventDefault()来取消其默认行为。

另外stopPropagation()方法用于立即停止事件在DOM层中的传播，例如，直接添加到一个按钮的事件处理程序可以调用stopPropagation()方法，从而避免触发注册在body上面的事件处理程序。例如：

**var** btn=document.getElementById("myBtn");  
btn.onclick=**function**(event){  
 btn.style.backgroundColor="yellow";  
 event.stopPropagation();  
}  
document.body.onclick=**function**(){  
 alert("Body Clicked");  
}

事件对象的eventPhase属性可以判断事件当前正处于哪个阶段。如果是在捕获阶段，eventPhase=1；如果是在目标阶段，eventPhase=2；如果是在冒泡阶段，eventPhase=3.可以使用下面的程序验证：

btn.onclick=**function**(event){  
 alert(event.eventPhase);//2  
};  
  
document.body.addEventListener("click",**function**(event){  
 alert(event.eventPhase);//1  
},**true**);  
  
document.body.onclick=**function**(a){  
 alert(a.eventPhase);//3  
};

### 13.3.2 IE中的事件对象

## 13.4 事件类型

* UI(User Interface，用户界面)事件
* 焦点事件
* 鼠标事件
* 滚轮事件
* 文本事件
* 键盘事件
* 合成事件：当为IME(Input Method Editor，输入法编辑器)输入字符时触发；
* 变动事件：当底层DOM结构发生变动时触发。

### 13.4.1 UI事件

* load：当页面完全加载后在window上面触发，当所有框架都加载完毕时在框架集上触发，当图像加载完毕时在<img>元素上面触发，
* unload：当页面完全卸载后在window上面触发，当所有框架都卸载后在框架集上面触发。
* resize：当窗口或框架的大小发生变化时在window或框架上面触发。
* scroll：当用户滚动带滚动条的元素中的内容时，在该元素上面触发。

#### 1、load事件

JavaScript中最常用的一个事件就是load。当页面完全加载后(包括所有图像、JavaScript文件、CSS文件等外部资源)，就会触发window上面的load事件。

有两种定义onload事件处理程序的方式。第一种方式是使用如下所示的JavaScript代码：

EventUtil.addHandler(window,"load", **function** (event) {  
 alert("Loaded!");  
});

这是通过JavaScript来指定事件处理程序的方式，使用了跨浏览器的EventUtil对象。

第二种指定事件处理程序的方式是为<body>元素添加一个onload特性，如下面的例子所示：

<body onload=”alert(‘Loaded!’)”>

一般来说，在window上面发生的任何事件都可以在<body>元素中通过相应的特性来指定，因为在HTML中无法访问window元素。

图像上面也可以触发load事件，无论是在DOM中的图像元素还是HTML中的图像元素。因此可以在HTML中为任何图像指定onload事件处理程序。例如，””

<img src="smile.gif" onload="alert('Loaded!')">

同样的功能也可以使用JavaScript来实现：

var img=document.getElementById(”MyImage”);

EventUtil.addHandler(img,"load",**function** (event){  
 alert("Loaded!");  
});

在JavaScript中新图像元素不一定要从添加到文档后才开始下载，只要设置了src属性就会开始下载。

与图像不同，只有在设置了<script>元素的src属性并将该元素添加到文档后，才会开始下载JavaScript文件。

#### unload事件

这个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unload事件。而利用这个事件最多的情况是清除引用，以避免内存泄漏。

与load事件类似，也有两种方式指定unload事件，JavaScript方式和DOM方式。

#### 3、resize事件

可以通过JavaScript方式或者为<body>元素中的onresize特性来指定事件处理程序。

与其他发生在window上的事件类似，传入事件处理程序中的event对象有一个target属性，值为document；

#### 4、scroll事件

### 13.4.2 焦点事件

焦点事件会在页面获得或失去焦点时触发。

* blur：在元素失去焦点是触发
* focus：在元素获得焦点时触发

上面两个事件都不会冒泡。

### 13.4.3 鼠标与滚轮事件

DOM3级事件中定义了9种鼠标事件：

* click：鼠标左键或者是回车时触发
* dblclick:双击时触发
* mousedown：按下鼠标任意按钮触发
* mousemove：当鼠标指针在元素内部移动时重复触发
* mouseout：在鼠标指针位于一个元素上方，然后用户将其移入另一个元素时触发
* mouseover：在鼠标指针位于一个元素外部，然后用户将其首次移入另一个元素边界之内时触发
* mouseup：在用户释放鼠标按钮时触发。

只有在同一个元素上相继触发mousedown和mouseup事件才会触发click事件。如果有一个被取消，就不会触发click事件。

#### 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发生的。这个位置信息保存在事件对象的clientX和clientY属性中。它们的值表示事件发生时鼠标指针在视口中的水平和垂直坐标。

#### 3、屏幕坐标位置

鼠标事件发生时还有一个相对于整个电脑屏幕的位置，screenX和screenY。

#### 4、修改键

#### 5、相关元素

#### 6、鼠标按钮

#### 7、更多的事件信息

#### 8、鼠标滚轮事件

### 13.4.4 键盘与文本事件

* keydown:按下键盘上的任意键时触发，按住不放，会重复触发
* keypress:按下字符键时触发，按住不放，重复触发
* keyup:释放键盘上的键时触发

只有一个文本事件：textInput。

#### 键码

在发生keydown和keyup事件时，event对象的keyCode属性中会包含一个代码，与键盘上一个特定的键对应。

#### 13.4.5 复合事件

### 13.4.7 HTML5事件

### 13.5 内存和性能

### 13.5.1 事件委托

就是利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。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事件处理程序，而不必给每个可单机的元素分别指定事件处理程序。

<ul id="myLinks">  
 <li id="goSomewhere">Go somewhere</li>  
 <li id="doSomething">Do something</li>  
 <li id="sayHi">Say hi</li>  
</ul>

其中包括3个被单击后执行操作的列表项，按照传统的做法，需要为每个li指定事件处理程序。此时，可以使用事件委托技术解决这个问题。使用事件委托只需要在DOM树中尽量最高的层次上添加一个事件处理程序。如下面的例子：

**var** list=document.getElementById("myLinks");  
EventUtil.addHandler(list,"click",**function** (event){  
 **var** target=EventUtil.getTarget(event);  
 **switch**(target.id){  
 **case** "goSomewhere":  
 location.href="http://www.lizan.com";  
 **break**;  
 **case** "doSomething":  
 document.title="I changed the document's title.";  
 **break**;  
 **case** "sayHi":  
 alert("hi");  
 **break**;  
 }  
});

### 13.5.2 移除事件处理程序

## 13.6 模拟事件

# 第14章 表单脚本

### 14.1.1 提交表单

提交表单：form.submit();

### 14.1.2 重置表单

### 14.1.3 表单字段

## 14.2 文本框脚本

### 14.2.1 选择文本

### 14.2.2 过滤输入

### 14.2.3 自动切换焦点

## 14.3 选择框脚本

## 14.4. 表单序列化

# 第15章 使用Canvas绘图

## 15.1 基本用法

要使用<canvas>元素，必须先设置width和height属性，指定可以绘图的区域大小。出现在开始和结束标签之间的信息是后备信息，如果浏览器不支持<canvas>元素，就会显示这些信息。

如果不为<canvas>元素添加样式或者不绘制图形，在界面上是看不到该元素的。

要在这块画布上绘制图形，需要取得绘图上下文，要获得该引用要调用getContext()方法并传入上下文的名字，传入“2d”，就可以获得2D上下文对象。

# 第16章 HTML5脚本编程

# 第17章 错误处理与调试

### 17.2.1 try-catch语句

try(){

//错误代码

}catch(error){

//处理错误代码

}

与其他语言不同的是，即使你不想使用catch中的这个错误对象，也要给他起个名字。错误对象中有一个所有浏览器都支持的message属性。

# 第18章 JavaScript与XML

# 第20章 JSON

JSON是一种数据格式，而不是一种编程语言，很多编程语言都支持JSON，并不是JavaScript特有的。

## 20.1 语法

JSON的语法可以表示以下三种类型的值：

* 简单值：可以表示字符串、数值、布尔值和null。但是不支持undefined
* 对象：
* 数组

### 20.1.1 简单值

JSON在表示字符串时必须使用双引号。

### 20.1.2 对象

JSON中的对象与JavaScript中的字面量稍微有些不同。JavaScript中的对象字面量如下：

var person={

name:”lizan”,

age:29

};

JSON中的对象属性要求加引号，实际上JavaScript中的对象字面量中的属性名也可以加引号。

JSON表示上述对象的方式如下：

{

“name”:”lizan”,

“age”:29

}

与JavaScript中的对象字面量相比，JSON中的对象有两点不同。首先没有声明变量(JSON中没有变量的概念)。其次，没有末尾的分号(因为这不是JavaScript语句，所以没有分号)。

### 20.1.3 数组

JSON数组采用的就是JavaScript中数组字面量的形式。例如，下面是JavaScript中的数组字面量：

var values=[25,”hi”,true];

在JSON中，可以采用同样的语法表示同一个数组：

[25,”hi”,true]

同样要注意，JSON数组也没有变量和分号。

把对象和数组结合起来就可以组成更复杂的数据结构。

[{  
 "title":"professional JavaScript",  
 "authors":["lizan"],  
 "edition":3.,  
 "year":2011  
},{  
 "title":"Professional JavaScript",  
 "author":"lizan",  
 "edition":2,  
 "year":2009  
}]

## 20.2 解析与序列化

以上面一节中包含一组图书序列的JSON数据结构为例，在解析为JavaScript对象后，只需要下面一行简单的代码就可以取得第二本书的名字：

books[1].title;

### 20.2.1 JSON对象

ECMAScript5对解析JSON的行为进行了规范，定义了全局对象JSON。

JSON对象有两个方法：stringify()和parse()。这两个方法分别用于把JavaScript对象序列化为JSON字符串和把JSON字符串解析为原生JavaScript值。例如：

**var** book={  
 title:"Professional JavaScript",  
 author:"Li Zan",  
 edition:3,  
 year:2011  
};  
**var** jsonText=JSON.stringify(book);  
alert(jsonText);// {"title":"Professional JavaScript","author":"Li Zan","edition":3,"year":2011}

在序列化JavaScript对象时，所有函数及原型成员都会被有意忽略，不体现在结果中。此外，值为undefined的任何属性也都会被跳过。结果中最终都是值为有效JSON数据类型的实例属性。

将JSON字符串直接传递给JSON.parse()方法就可以得到相应的JavaScript值。例如使用下列代码就可以创建与book类似的对象：var bookCopy=JSON.parse(jsonText);

### 20.2.2 序列化选项

### 20.2.3 解析选项

# 第21章 Ajax与Comet

Ajax的核心是XMLHttpRequest对象(简称XHR)。虽然名字中包含XML的成分，但是Ajax通信与数据格式无关；这种技术就是无需刷新页面就可以从服务器取得数据，但不一定是XML数据。

## 21.1 XMLHttpRequest对象

### 21.1.1 XHR的用法

在使用XHR对象时，要调用的第一个方法是open()，它接收3个参数：要发送的请求的类型(“get”,”post”等)、请求的URL和表示是否异步发送请求的布尔值。下面就是调用这个方法的例子，

xhr.open(“get”,”example.php”,false);

关于这行代码要说明两条：①是URL相对于执行代码的当前页面(当然也可以执行绝对路径)②调用open()方法并不会真正发送请求，而只是启动一个请求以备发送。

注意：默认情况下，只能向同一个域中使用相同端口和协议的URL发送请求。如果URL与启动请求的页面有任何差别，都会引发安全错误。

要发送特定请求必须像下面这样调用send()请求：

**var** xhr=**new** XMLHttpRequest();  
xhr.open("get","example.php",**false**);  
xhr.send(**null**);

这里的send()方法就收一个参数，既要作为请求主体发送的数据。如果不需要请求主体发送数据，则必须传入null，因为这个参数对有些浏览器是必须的。

因为这次请求是同步的，JavaScript代码会等到服务器响应之后再继续执行。

在收到响应之后，响应的数据会自动填充XHR对象的属性，相关的属性简介如下：

* responseText:作为响应的主体被返回的文本。
* responseXML:如果响应的内容是“text/xml”或“application/xml”，这个属性中将保存包含着响应数据的XML DOM文档。对于非XML类型的数据，responseXML属性的值为null。
* status：响应的HTTP状态
* statusText:HTTP状态的说明

在接收到响应后，第一步是检查status属性，以确定响应已经成功返回。一般来说，可以将HTTP状态代码为200作为成功的标志。此时，responseText属性的内容已经就绪。此外状态为304表示请求的资源并没有被修改，可以直接使用浏览器中缓存的版本。为确保接收到适当的响应，应该像下面这样检查上述这两种状态代码：

xhr.open("get","example.php",**false**);  
xhr.send(**null**);  
**if**((xhr.status>=200 && xhr.status<=300) || (xhr.status==304)){  
 alert(xhr.responseText);  
}**else** {  
 alert("response was unsuccessful:"+xhr.status);  
}

像前面这样发送同步请求当然没有问题，但大多数情况下，我们还要发送异步请求。

发送异步请求时，可以检测XHR对象的readyState属性，该属性表示请求/响应过程的当前活动阶段。这个属性可取的值如下：

* 0：未初始化。尚未调用open()方法
* 1：启动。已经调用open()方法，但尚未调用send()方法
* 2：发送。已经调用send()方法，但尚未接收到响应
* 3：接收。已经接收到部分响应数据。
* 4：完成。已经接收到全部响应数据。

只要readyState的值由一个值变成另一个值，都会触发一次readystatechange事件。通常只对readyState的值为4的阶段感兴趣。

xhr.onreadystatechange=**function** (){  
 **if**(xhr.readyState==4){  
 **if**((xhr.status>=200 && xhr.status<=300) || (xhr.status==304)){  
 alert(xhr.responseText);  
 }**else** {  
 alert("response was unsuccessful:"+xhr.status);  
 }  
 }  
};  
xhr.open("get","example.php",**false**);  
xhr.send(**null**);

### 21.1.2 HTTP头部信息

每个HTTP请求和响应都会带有头部信息。XHR对象也提供了操作这两种头部(请求头部和响应头部)信息的方法。

默认情况下，在发送XHR请求的同时，还会发送下列头部信息：

* Accept：浏览器能够处理的内容类型
* Accept-Charset：浏览器能够显示的字符集
* Accept-Encoding：浏览器能够处理的压缩编码
* Accept-Language：浏览器当前设置的语言
* Connection：浏览器与服务器之间连接的类型
* Cookie：当前页面设置的任何Cookie
* Host：发出请求的页面所在的域
* Referer：发出请求的页面的URI
* User-Agent：浏览器的用户代理字符串

使用setRequestHeader()方法可以设置自定义的请求头部信息。这个方法就收两个参数：头部字段的名称和头部字段的值。要成功发送请求头部信息，必须在调用open()方法之后且调用send()方法之前调用setRequestHeader()，

xhr.onreadystatechange=**function** (){  
 **if**(xhr.readyState==4){  
 **if**((xhr.status>=200 && xhr.status<=300) || (xhr.status==304)){  
 alert(xhr.responseText);  
 }**else** {  
 alert("response was unsuccessful:"+xhr.status);  
 }  
 }  
};  
xhr.open("get","example.php",**false**);  
**xhr.setRequestHeader("MyHeader","MyValue");**  
xhr.send(**null**);

### 21.1.3 get请求

查询字符串中每个参数的名称和值都必须使用encodeURIComponent()进行编码，然后才能放在URL的末尾。

### 21.1.4 post请求

可以使用XHR来模仿表单提交，首先将Content-Type头部信息设置为application/x-www-form-urlencoded，也就是表单提交时的内容类型，其次是以适当的格式创建字符串。以post方式提交时需要将表单的数据进行序列化，然后通过XHR对象发送到服务器。

xhr.onreadystatechange=**function** (){  
 **if**(xhr.readyState==4){  
 **if**((xhr.status>=200 && xhr.status<=300) || (xhr.status==304)){  
 alert(xhr.responseText);  
 }**else** {  
 alert("response was unsuccessful:"+xhr.status);  
 }  
 }  
};  
xhr.open("get","example.php",**false**);  
xhr.setRequestHeader("Content-Type","application/x-www-form-urlencoded");  
**var** form=document.getElementById("user-info");  
xhr.send(serialize(form));

### 21.2.1 FromData

FormData类型可以序列化表单以及创建与表单格式相同的数据。下面代码创建了一个FormData对象，并向其中添加了一些数据。

**var** data=**new** FormData();  
data.append("name","LiZan");

创建FormData之后，可以直接将它传递给XHR的send()方法：

xhr.open("get","example.php",**false**);  
xhr.setRequestHeader("Content-Type","application/x-www-form-urlencoded");  
**var** form=document.getElementById("user-info");  
xhr.send(**new** FormData(form));

### 21.2.2 超时设定

### 21.2.3 overrideMimeType()方法

## 21.3 进度事件

### 21.3.1 load事件

### 21.3.2 progress事件

## 21.4 跨源资源共享

## 21.5 其他跨域技术

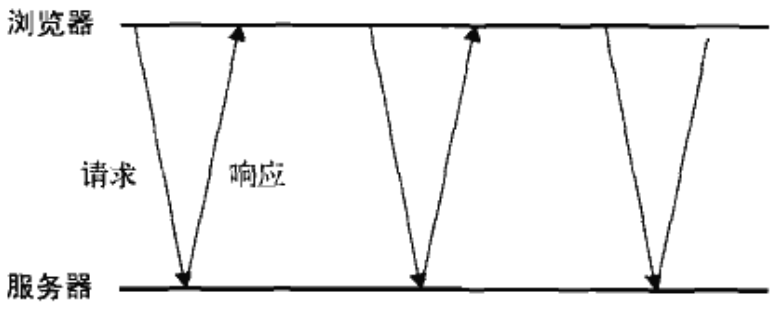
### 21.5.1 图像ping

### 21.5.2 JSONP

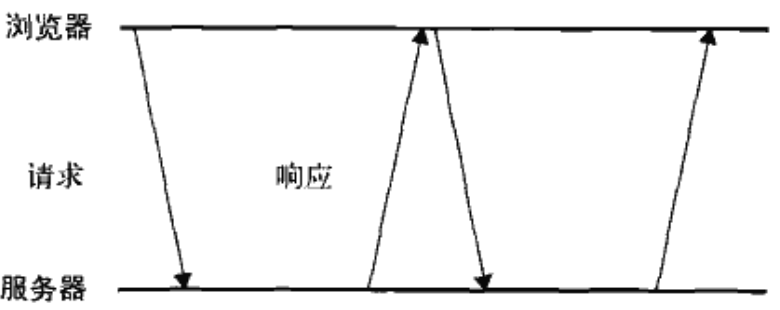
### 21.5.3 Comet

指的是一种更高级的Ajax技术(“服务器推送”)，是一种服务器向页面推送数据的技术。Comet能够让信息近乎实时的推送到页面上，非常适合处理体育比赛的分数和股票报价。

有两种实现Comet的方式：长轮询和流。长轮询是传统轮询(短轮询)的一个翻版，指的是浏览器定时向服务器发送请求，看有没有更新数据。下图展示的是短轮询的时间线：



长轮询把短轮询颠倒了一下。页面发起一个到服务器的请求，然后服务器一直保持连接打开，直到有数据可发送。发送完数据之后，浏览器关闭连接，随即又发起一个到服务器的新请求。这一过程在页面打开期间一直持续不断。下面是长轮询时间线：



第二种流行的Comet实现是HTTP流。流不同于上述两种轮询，因为它在页面的整个生命周期内只使用一个HTTP连接。具体来说，就是浏览器向服务器发送一个请求，而服务器保持连接打开，然后周期性的向浏览器发送数据。

### 21.5.5 Web Sockets

Web Sockets的目标是在一个单独的持久连接上提供全双工、双向通信。

# 第22章 高级技巧

# 第23章 离线应用与客户端存储

### 23.3.1 Cookie