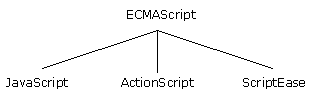
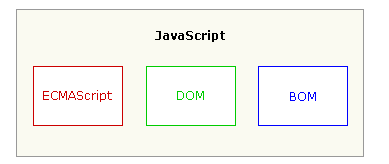
# 一.JavaScript的历史

* 1. 1992年Nombas开发出C-minus-minus(C--)的嵌入式脚本语言(最初绑定在CEnvi软件中).后将其改名ScriptEase.(客户端执行的语言)
  2. Netscape(网景)接收Nombas的理念,(Brendan Eich)在其Netscape Navigator 2.0产品中开发出一套livescript的脚本语言.Sun和Netscape共同完成.后改名叫Javascript
  3. 微软随后模仿在其IE3.0的产品中搭载了一个JavaScript的克隆版叫Jscript.
  4. 为了统一三家,ECMA(欧洲计算机制造协会)定义了ECMA-262规范.国际标准化组织及国际电工委员会（ISO/IEC）也采纳 ECMAScript 作为标准（ISO/IEC-16262）。从此，Web 浏览器就开始努力（虽然有着不同的程度的成功和失败）将 ECMAScript 作为 JavaScript 实现的基础。EcmaScript是规范.



# 1.1,ECMAScript

尽管 ECMAScript 是一个重要的标准，但它并不是 JavaScript 唯一的部分，当然，也不是唯一被标准化的部分。实际上，一个完整的 JavaScript 实现是由以下 3 个不同部分组成的：

* 核心（ECMAScript） 一天到一天半
* 文档对象模型（DOM） Document object model(整合js和html和css)(1天半)
* 浏览器对象模型（BOM） Broswer object model(整合js和浏览器)(半天)
* Javascript 在开发中绝大多数情况是基于对象的.也是面向对象的.

# 1.2ECMAScript

Web 浏览器对于 ECMAScript 来说是一个宿主环境，但它并不是唯一的宿主环境。事实上，还有不计其数的其他各种环境（例如 Nombas 的 ScriptEase，以及 Macromedia 同时用在 Flash 和 Director MX 中的 ActionScript）可以容纳 ECMAScript 实现。那么 ECMAScript 在浏览器之外规定了些什么呢？

简单地说，ECMAScript 描述了以下内容：

* 语法
* 类型
* 语句
* 关键字
* 保留字
* 运算符
* 对象 (封装 继承 多态)基于对象的语言.使用对象.

# 二ECMAScript 基础

## 2.1.ECMAScript 语法

1. **变量区分大小写**

a>可一次声明多个变量,并且变量不必具有相同的类型.例如:var test = "hi", age = 25;

b>命名规范:

* 第一个字符必须是字母、下划线（\_）或美元符号（$）
* 余下的字符可以是下划线、美元符号或任何字母或数字字符

c>ECMAScript 的解释程序遇到未声明过的标识符时，用该变量名创建一个全局变量，并将其初始化为指定的值。

这是该语言的便利之处，不过如果不能紧密跟踪变量，这样做也很危险。最好的习惯是像使用其他程序设计语言一样，总是声明所有变量。

1. **变量是弱类型的**
2. **每行的分号可有可无.没有就按折行符的末尾作为结尾.**
3. **注释与java,c,php相同**
4. **使用{}来封装代码块**

|  |
| --- |
| 著名的变量命名规则(附录) 只是因为变量名的语法正确，并不意味着就该使用它们。变量还应遵守以下某条著名的命名规则： Camel 标记法 首字母是小写的，接下来的字母都以大写字符开头。例如：  var myTestValue = 0, mySecondValue = "hi"; Pascal 标记法 首字母是大写的，接下来的字母都以大写字符开头。例如：  VarMyTestValue = 0, MySecondValue = "hi"; 匈牙利类型标记法 在以 Pascal 标记法命名的变量前附加一个小写字母（或小写字母序列），说明该变量的类型。例如，i 表示整数，s 表示字符串，如下所示“  Var iMyTestValue = 0, sMySecondValue = "hi"; |

## 2.2原始值和引用值

在 ECMAScript 中，变量可以存在两种类型的值，即原始值和引用值。

原始值

存储在栈（stack）中的简单数据段，也就是说，它们的值直接存储在变量访问的位置。

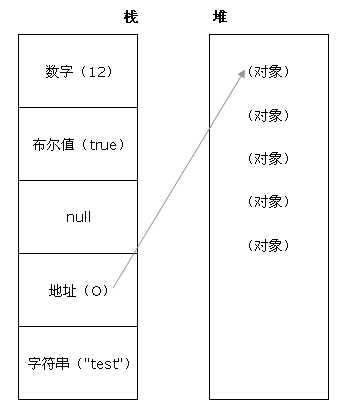
引用值

存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存处。

5种原始值类型如下:

* Undefined、
* Null、
* Boolean、
* Number
* String

**在许多语言中，字符串都被看作引用类型，而非原始类型，因为字符串的长度是可变的。ECMAScript 打破了这一传统。**



**Undefined 类型**

Undefined 类型只有一个值，即 undefined。当声明的变量未初始化时，该变量的默认值是 undefined。

当函数无明确返回值时，返回的也是值 "undefined";

Null 类型

另一种只有一个值的类型是 Null，它只有一个专用值 null，即它的字面量。值 undefined 实际上是从值 null 派生来的，因此 ECMAScript 把它们定义为相等的。

尽管这两个值相等，但它们的含义不同。undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象（在讨论 typeof 运算符时，简单地介绍过这一点）。如果函数或方法要返回的是对象，那么找不到该对象时，返回的通常是 null。

Boolean 类型

Boolean 类型是 ECMAScript 中最常用的类型之一。它有两个值 true 和 false （即两个 Boolean 字面量）。即使 false 不等于 0，0 也可以在必要时被转换成 false.

Number 类型

ECMA-262 中定义的最特殊的类型是 Number 类型。这种类型既可以表示 32 位的整数，还可以表示 64 位的浮点数。

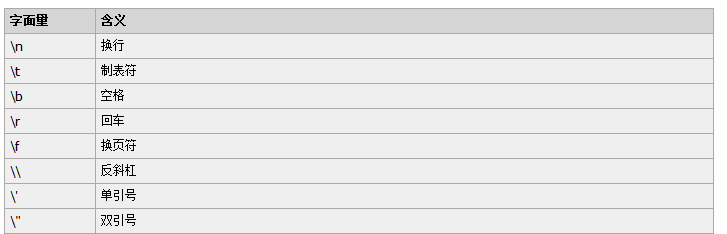
* 定义十进制整数var iNum = 86;
* 定义八进制var iNum = 070;(了解)
* 定义十六进制var iNum = 0xAB;(了解)
* 科学计数法var fNum = 5.618e7;var fNum = 8-e17(了解)
* 定义浮点数var fNum = 5.0;

**对于浮点字面量的有趣之处在于，用它进行计算前，真正存储的是字符串。**

* 数字的最大值**:**Number.MAX\_VALUE
* 数字的最小值:Number.MIN\_VALUE
* 超过最大值:Number.POSITIVE\_INFINITY(不可参与运算)
* 小于最小值:Number.NEGATIVE\_INFINITY(不可参与运算)
* 特殊值:NaN(Not A Number).一般说来，这种情况发生在类型（String、Boolean 等）转换失败时。NaN不等于NaN,可以使用isNaN()方法.

## String 类型

它是唯一没有固定大小的原始类型。声明时可以使用单引号或者双引号.因为没有字符类型.



## typeof 运算符

ECMAScript 提供了 typeof 运算符来判断一个值是否在某种类型的范围内。可以用这种运算符判断一个值是否表示一种原始类型：如果它是原始类型，还可以判断它表示哪种原始类型。

对变量或值调用 typeof 运算符将返回下列值之一：

* undefined - 如果变量是 Undefined 类型的
* boolean - 如果变量是 Boolean 类型的
* number - 如果变量是 Number 类型的
* string - 如果变量是 String 类型的
* object - 如果变量是一种引用类型或 Null 类型的

**注释**：您也许会问，为什么 typeof 运算符对于 null 值会返回 "Object"。这实际上是 JavaScript 最初实现中的一个错误，然后被 ECMAScript 沿用了。现在，null 被认为是对象的占位符，从而解释了这一矛盾，但从技术上来说，它仍然是原始值。

# 三ECMAScript 语句

## if 语句

if 语句是 ECMAScript 中最常用的语句之一，事实上在许多计算机语言中都是如此。

if 语句的语法：

if (*condition*) *statement1* else *statement2*

其中 *condition* 可以是任何表达式，计算的结果甚至不必是真正的 boolean 值，ECMAScript 会把它转换成 boolean 值。

如果条件计算结果为 true，则执行 *statement1*；如果条件计算结果为 false，则执行 *statement2*。

还可以串联多个 if 语句。就像这样：

if (*condition1*) *statement1* else if (*condition2*) *statement2* else *statement3*

## do-while 语句

do-while 语句是后测试循环，即退出条件在执行循环内部的代码之后计算。这意味着在计算表达式之前，至少会执行循环主体一次。

它的语法如下：

do {*statement*} while (*expression*);

## while 语句

while 语句是前测试循环。这意味着退出条件是在执行循环内部的代码之前计算的。因此，循环主体可能根本不被执行。

它的语法如下：

while (*expression*) *statement*

## for 语句

for 语句是前测试循环，而且在进入循环之前，能够初始化变量，并定义循环后要执行的代码。

它的语法如下：

for (*initialization; expression; post-loop-expression*) *statement*

## switch 语句

switch 语句是 if 语句的兄弟语句。

开发者可以用 switch 语句为表达式提供一系列的情况（case）。

switch 语句的语法：

switch (*expression*)

case *value*: *statement*;

break;

case *value*: *statement*;

break;

case *value*: *statement*;

break;

case *value*: *statement*;

break;

...

case *value*: *statement*;

break;

default: *statement*;

每个情况（case）都是表示“如果 *expression* 等于 *value*，就执行 *statement*”。

关键字 break 会使代码跳出 switch 语句。如果没有关键字 break，代码执行就会继续进入下一个 case。

关键字 default 说明了表达式的结果不等于任何一种情况时的操作（事实上，它相对于 else 从句）。

# 四ECMAScript 运算符

## 4.1ECMAScript 一元运算符

#### void(函数后讲解)

void 运算符对任何值返回 undefined。该运算符通常用于避免输出不应该输出的值，例如，从 HTML 的 <a> 元素调用 JavaScript 函数时。要正确做到这一点，函数不能返回有效值，否则浏览器将清空页面，只显示函数的结果。例如：

|  |
| --- |
| <a href="javascript:window.open('about:blank')">Click me</a> |

如果把这行代码放入 HTML 页面，点击其中的链接，即可看到屏幕上显示 "[object]"。

这是因为 window.open() 方法返回了新打开的窗口的引用。然后该对象将被转换成要显示的字符串。

要避免这种效果，可以用 void 运算符调用 window.open() 函数：

|  |
| --- |
| <a href="javascript:void(window.open('about:blank'))">Click me</a> |

这使 window.open() 调用返回 undefined，它不是有效值，不会显示在浏览器窗口中。

提示：请记住，没有返回值的函数真正返回的都是 undefined。

## 4.2i++,i--,++i,--i

Var I = 1’

Alert(i++);//1

Alert(++i);//3

Alert(i--);//3

Alert(--i);//1

## 4.3一元加法,一元减法(了解)

一元加法本质上对数字无任何影响：

|  |
| --- |
| var iNum = 20;  iNum = +iNum;  alert(iNum); //输出 "20" |

尽管一元加法对数字无作用，但对字符串却有有趣的效果，会把字符串转换成数字。

|  |
| --- |
| var sNum = "20";  alert(typeof sNum); //输出 "string"  var iNum = +sNum;  alert(typeof iNum); //输出 "number" |

这段代码把字符串 "20" 转换成真正的数字。当一元加法运算符对字符串进行操作时，它计算字符串的方式与 parseInt() 相似，主要的不同是只有对以 "0x" 开头的字符串（表示十六进制数字），一元运算符才能把它转换成十进制的值。因此，用一元加法转换 "010"，得到的总是 10，而 "0xB" 将被转换成 11。

### 4.4ECMAScript 位运算符(了解)跟java一模一样.

#### 4.4.1位运算 NOT🡺~

位运算 NOT 由否定号（~）表示，位运算 NOT 实质上是对数字求负，然后减 1，因此 25 变 -26。

|  |
| --- |
| var iNum1 = 25; //25 等于 00000000000000000000000000011001  var iNum2 = ~iNum1; //转换为 11111111111111111111111111100110  alert(iNum2); //输出 "-26" |

还可以使用下面的方式:

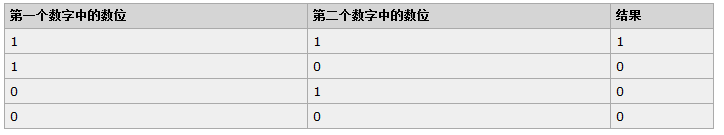
|  |
| --- |
| var iNum1 = 25;  var iNum2 = -iNum1 -1;  alert(iNum2); //输出 -26 |

## 4.4.2位运算 AND🡺&

|  |
| --- |
| var iResult = 25 & 3;  alert(iResult); //输出 "1" |

为什么得到这个结果?

|  |
| --- |
| 25 = 0000 0000 0000 0000 0000 0000 0001 1001  3 =0000 0000 0000 0000 0000 0000 0000 0011  ---------------------------------------------  AND =0000 0000 0000 0000 0000 0000 0000 0001 |

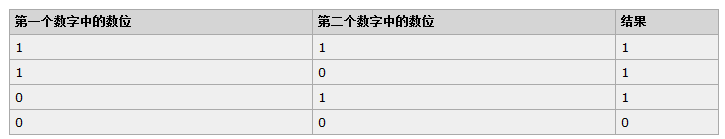


## 4.4.3位运算 OR🡺|

|  |
| --- |
| var iResult = 25 | 3;  alert(iResult); //输出 "27" |

为什么得到这个结果?

|  |
| --- |
| 25 = 0000 0000 0000 0000 0000 0000 0001 1001  3 = 0000 0000 0000 0000 0000 0000 0000 0011  --------------------------------------------  OR = 0000 0000 0000 0000 0000 0000 0001 1011 |

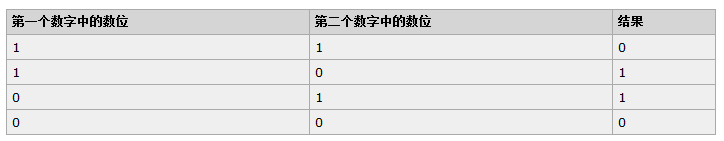


## 4.4.4位运算 XOR

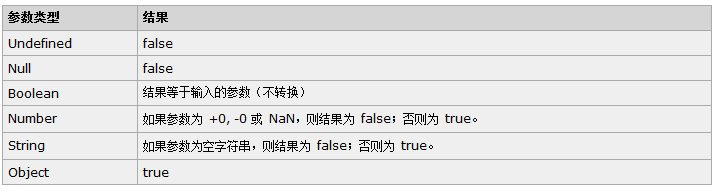
|  |
| --- |
| var iResult = 25 ^ 3;  alert(iResult); //输出 "26" |

为什么得到这个结果?

|  |
| --- |
| 25 = 0000 0000 0000 0000 0000 0000 0001 1001  3 = 0000 0000 0000 0000 0000 0000 0000 0011  ---------------------------------------------  XOR = 0000 0000 0000 0000 0000 0000 0001 1010 |

左移(<<)右移(>>)无符号右移(>>>)(了解)

### 4.4.5ECMAScript Boolean 运算符(异常重要)



##### 逻辑 NOT 运算符 (!)

* 如果运算数是对象，返回 false
* 如果运算数是数字 0，返回 true
* 如果运算数是 0 以外的任何数字，返回 false
* 如果运算数是 null，返回 true
* 如果运算数是 NaN，返回 true
* 如果运算数是 undefined，返回true

#### 逻辑 AND 运算符(&&)

**逻辑 AND 运算的运算数可以是任何类型的，不止是 Boolean 值。**

如果某个运算数不是原始的 Boolean 型值，逻辑 AND 运算并不一定返回 Boolean 值：

* 如果一个运算数是对象，另一个是 Boolean 值，返回该对象。
* 如果两个运算数都是对象，返回第二个对象。
* 如果某个运算数是 null，返回 null。
* 如果某个运算数是 NaN，返回 NaN。
* 如果某个运算数是 undefined，返回undefined。

|  |
| --- |
| * //1 false&&o ==> false * //2 true&&o ==> o ==> true * //3 o&&true ==> true * //4 o&&false ==> false * //5 o1&&o2 ==> * //6 null&&true == > null * //7 NaN && true == > NaN * //8 NaN&&null ==> NaN * //9 null&&NaN ==> null * //10 undefined&&true ==> undefined * //11 false&&undefined == > false |

#### 逻辑 OR 运算符(||)

与逻辑 AND 运算符相似，如果某个运算数不是 Boolean 值，逻辑 OR 运算并不一定返回 Boolean 值：

* 如果一个运算数是对象，并且该对象左边的运算数值均为 false，则返回该对象。
* 如果两个运算数都是对象，返回第一个对象。
* 如果最后一个运算数是 null，并且其他运算数值均为 false，则返回 null。
* 如果最后一个运算数是 NaN，并且其他运算数值均为 false，则返回 NaN。
* 如果某个运算数是 undefined，返回undefined。

|  |
| --- |
| * //1 false||o ==> o * //2 true||o ==>true * //3 o||false == > o * //4 o||true == > o * //5 o1||o2 ==>o1 * //6 null||false == > false * //7 null||true ==> true * //8 null||undefined ==>undefined * //9 undefined||null ==>null |

### 4.4.6数学运算符

+-\*/%

### 4.4.7ECMAScript 关系运算符(重要)

var bResult = "Blue" < "alpha";

alert(bResult); //输出 true

在上面的例子中，字符串 "Blue" 小于 "alpha"，因为字母 B 的字符代码是 66，字母 a 的字符代码是 97。

#### 比较数字和字符串

另一种棘手的状况发生在比较两个字符串形式的数字时，比如：

var bResult = "25" < "3";

alert(bResult); //输出 "true"

上面这段代码比较的是字符串 "25" 和 "3"。两个运算数都是字符串，所以比较的是它们的字符代码（"2" 的字符代码是 50，"3" 的字符代码是 51）。

不过，如果把某个运算数该为数字，那么结果就有趣了：

var bResult = "25" < 3;

alert(bResult); //输出 "false"

这里，字符串 "25" 将被转换成数字 25，然后与数字 3 进行比较，结果不出所料。

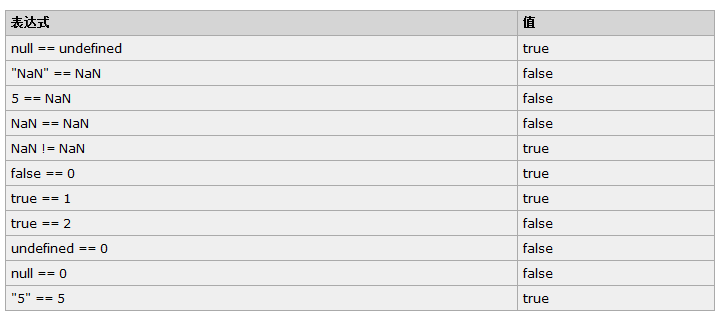
### 4.4.8ECMAScript 等性运算符

执行类型转换的规则如下：

* 如果一个运算数是 Boolean 值，在检查相等性之前，把它转换成数字值。false 转换成 0，true 为 1。
* 如果一个运算数是字符串，另一个是数字，在检查相等性之前，要尝试把字符串转换成数字。
* 如果一个运算数是对象，另一个是字符串，在检查相等性之前，要尝试把对象转换成字符串。
* 如果一个运算数是对象，另一个是数字，在检查相等性之前，要尝试把对象转换成数字。

在比较时，该运算符还遵守下列规则：

* 值 null 和 undefined 相等。
* 在检查相等性时，不能把 null 和 undefined 转换成其他值。
* 如果某个运算数是 NaN，等号将返回 false，非等号将返回 true。
* 如果两个运算数都是对象，那么比较的是它们的引用值。如果两个运算数指向同一对象，那么等号返回 true，否则两个运算数不等。



#### 全等号和非全等号

等号和非等号的同类运算符是全等号和非全等号。这两个运算符所做的与等号和非等号相同，只是它们在检查相等性前，不执行类型转换。

### 4.4.9ECMAScript 条件运算符(三元运算符)(跟java中一模一样)

*variable = boolean\_expression ? true\_value : false\_value*;

### 4.4.10ECMAScript 赋值运算符

* 乘法/赋值（\*=）
* 除法/赋值（/=）
* 取模/赋值（%=）
* 加法/赋值（+=）
* 减法/赋值（-=）

|  |
| --- |
| var iNum = 10;  iNum = iNum + 10； |

等同于

|  |
| --- |
| var iNum = 10;  iNum += 10； |

# 五ECMA对象

## ECMAScript 引用类型

从传统意义上来说，ECMAScript 并不真正具有类。事实上，除了说明不存在类，在 ECMA-262 中根本没有出现“类”这个词。ECMAScript 定义了“对象定义”，逻辑上等价于其他程序设计语言中的类。

var o = new Object();

#### 对象分类

1.由ECMAScript定义的本地对象.独立于宿主环境的 ECMAScript 实现提供的对象.(native object)

2.ECMAScript 实现提供的、独立于宿主环境的所有对象，在 ECMAScript 程序开始执行时出现.这意味着开发者不必明确实例化内置对象，它已被实例化了。ECMA-262 只定义了两个内置对象，即 Global 和 Math （它们也是本地对象，根据定义，每个内置对象都是本地对象）。（built-in object）

3.所有非本地对象都是宿主对象（host object），即由 ECMAScript 实现的宿主环境提供的对象。所有 BOM 和 DOM 对象都是宿主对象。

### 5.1Object 对象(了解)

Object 对象自身用处不大，不过在了解其他类之前，还是应该了解它。因为 ECMAScript 中的 Object 对象与 Java 中的 java.lang.Object 相似，ECMAScript 中的所有对象都由这个对象继承而来，Object 对象中的所有属性和方法都会出现在其他对象中，所以理解了 Object 对象，就可以更好地理解其他对象。

### Object 对象具有下列属性：

constructor

对创建对象的函数的引用（指针）。对于 Object 对象，该指针指向原始的 Object() 函数。

Prototype

对该对象的对象原型的引用。对于所有的对象，它默认返回 Object 对象的一个实例。

### Object 对象还具有几个方法：

hasOwnProperty(property)

判断对象是否有某个特定的属性。必须用字符串指定该属性。（例如，o.hasOwnProperty("name")）

IsPrototypeOf(object)

判断该对象是否为另一个对象的原型。

PropertyIsEnumerable

判断给定的属性是否可以用 for...in 语句进行枚举。

ToString()

返回对象的原始字符串表示。对于 Object 对象，ECMA-262 没有定义这个值，所以不同的 ECMAScript 实现具有不同的值。

ValueOf()

返回最适合该对象的原始值。对于许多对象，该方法返回的值都与 ToString() 的返回值相同。

注释：上面列出的每种属性和方法都会被其他对象覆盖。

### 5.2Function 对象(特殊)

Function 类可以表示开发者定义的任何函数。

用 Function 类直接创建函数的语法如下：

var function\_name = new function(*arg1*, *arg2*, ..., *argN*, *function\_body*)

function sayHi(sName, sMessage) {

alert("Hello " + sName + sMessage);

}

还可以这样定义它：

var sayHi

=

new Function("sName", "sMessage", "alert(\"Hello \" + sName + sMessage);");

虽然由于字符串的关系，这种形式写起来有些困难，但有助于理解函数只不过是一种引用类型，它们的行为与用 Function 类明确创建的函数行为是相同的。

**Function 对象的 length 属性**

如前所述，函数属于引用类型，所以它们也有属性和方法。

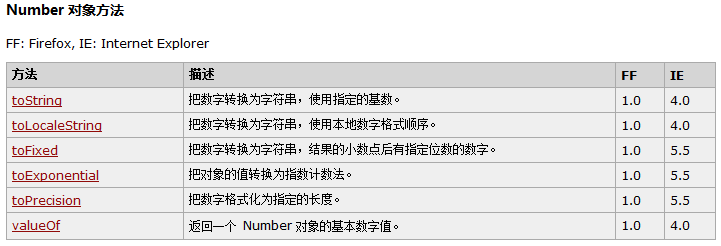
ECMAScript 定义的属性 length 声明了函数期望的参数个数。

**Function 对象的方法**

Function 对象也有与所有对象共享的 valueOf() 方法和 toString() 方法。这两个方法返回的都是函数的源代码，在调试时尤其有用。

## 5.3Number 对象





正如你可能想到的，Number 对象是 Number 原始类型的引用类型。要创建 Number 对象，采用下列代码：

var oNumberObject = new Number(68);

您应该已认出本章前面小节中讨论特殊值（如 Number.MAX\_VALUE）时提到的 Number 对象。所有特殊值都是 Number 对象的静态属性。

要得到数字对象的 Number 原始值，只需要使用 valueOf() 方法：

var iNumber = oNumberObject.valueOf();

当然，Number 类也有 toString() 方法，在讨论类型转换的小节中已经详细讨论过该方法。

除了从 Object 对象继承的标准方法外，Number 对象还有几个处理数值的专用方法。

#### toFixed() 方法

toFixed() 方法返回的是具有指定位数小数的数字的字符串表示。例如：

var oNumberObject = new Number(68);

alert(oNumberObject.toFixed(2)); //输出 "68.00"

在这里，toFixed() 方法的参数是 2，说明应该显示两位小数。该方法返回 "68.00"，空的字符串位由 0 来补充。对于处理货币的应用程序，该方法非常有用。toFixed() 方法能表示具有 0 到 20 位小数的数字，超过这个范围的值会引发错误。

#### toExponential() 方法(了解)

与格式化数字相关的另一个方法是 toExponential()，它返回的是用科学计数法表示的数字的字符串形式。

与 toFixed() 方法相似，toExponential() 方法也有一个参数，指定要输出的小数的位数。例如：

var oNumberObject = new Number(68);

alert(oNumberObject.toExponential(1)); //输出 "6.8e+1"

这段代码的结果是 "6.8e+1"，前面解释过，它表示 6.8x101。问题是，如果不知道要用哪种形式（预定形式或指数形式）表示数字怎么办？可以用 toPrecision() 方法。

#### toPrecision() 方法(了解)

toPrecision() 方法根据最有意义的形式来返回数字的预定形式或指数形式。它有一个参数，即用于表示数的数字总数（不包括指数）。例如，

var oNumberObject = new Number(68);

alert(oNumberObject.toPrecision(1)); //输出 "7e+1"

这段代码的任务是用一位数字表示数字 68，结果为 "7e+1"，以另外的形式表示即 70。的确，toPrecision() 方法会对数进行舍入。不过，如果用 2 位数字表示 68，就容易多了：

var oNumberObject = new Number(68);

alert(oNumberObject.toPrecision(2)); //输出 "68"

当然，输出的是 "68"，因为这正是该数的准确表示。不过，如果指定的位数多于需要的位数又如何呢？

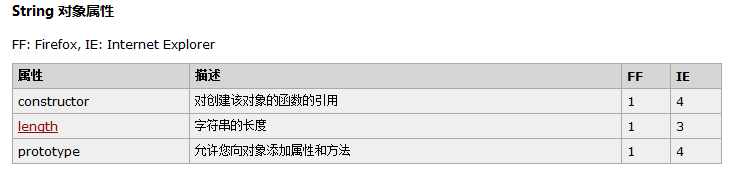
var oNumberObject = new Number(68);

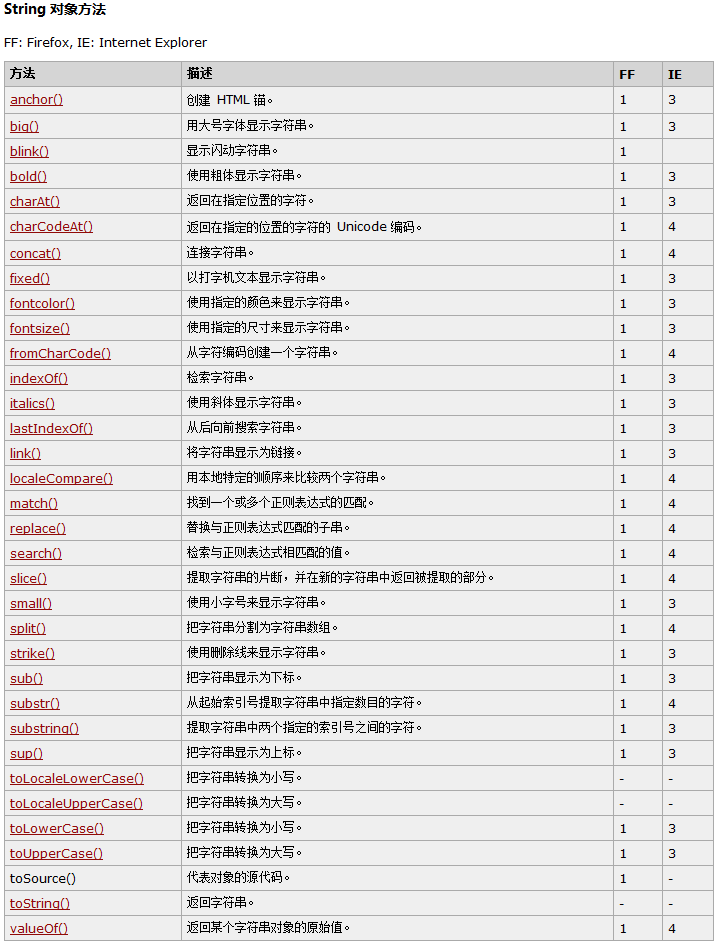
alert(oNumberObject.toPrecision(3)); //输出 "68.0"

在这种情况下，toPrecision(3) 等价于 toFixed(1)，输出的是 "68.0"。

toFixed()、toExponential() 和 toPrecision() 方法都会进行舍入操作，以便用正确的小数位数正确地表示一个数。

### 5.4String 对象





String 对象是 String 原始类型的对象表示法，它是以下方式创建的：

var oStringObject = new String("hello world");

String 对象的 valueOf() 方法和 toString() 方法都会返回 String 类型的原始值：

alert(oStringObject.valueOf() == oStringObject.toString()); //输出 "true"

如果运行这段代码，输出是 "true"，说明这些值真的相等。

注释：String 对象是 ECMAScript 中比较复杂的引用类型之一。同样，本节的重点只是 String 类的基本功能。。

#### length 属性

String 对象具有属性 length，它是字符串中的字符个数：

var oStringObject = new String("hello world");

alert(oStringObject.length); //输出 "11"

这个例子输出的是 "11"，即 "hello world" 中的字符个数。注意，即使字符串包含双字节的字符（与 ASCII 字符相对，ASCII 字符只占用一个字节），每个字符也只算一个字符。

#### charAt() 和 charCodeAt() 方法

String 对象还拥有大量的方法。

首先，两个方法 charAt() 和 charCodeAt() 访问的是字符串中的单个字符。这两个方法都有一个参数，即要操作的字符的位置。

charAt() 方法返回的是包含指定位置处的字符的字符串：

var oStringObject = new String("hello world");

alert(oStringObject.charAt(1)); //输出 "e"

在字符串 "hello world" 中，位置 1 处的字符是 "e"。在“ECMAScript 原始类型”这一节中我们讲过，第一个字符的位置是 0，第二个字符的位置是 1，依此类推。因此，调用 charAt(1) 返回的是 "e"。

如果想得到的不是字符，而是字符代码，那么可以调用 charCodeAt() 方法：

var oStringObject = new String("hello world");

alert(oStringObject.charCodeAt(1)); //输出 "101"

这个例子输出 "101"，即小写字母 "e" 的字符代码。

#### concat() 方法(几乎不用)

接下来是 concat() 方法，用于把一个或多个字符串连接到 String 对象的原始值上。该方法返回的是 String 原始值，保持原始的 String 对象不变：

var oStringObject = new String("hello ");

var sResult = oStringObject.concat("world");

alert(sResult); //输出 "hello world"

alert(oStringObject); //输出 "hello "

在上面这段代码中，调用 concat() 方法返回的是 "hello world"，而 String 对象存放的仍然是 "hello "。出于这种原因，较常见的是用加号（+）连接字符串，因为这种形式从逻辑上表明了真正的行为：

var oStringObject = new String("hello ");

var sResult = oStringObject + "world";

alert(sResult); //输出 "hello world"

alert(oStringObject); //输出 "hello "

#### indexOf() 和 lastIndexOf() 方法

迄今为止，已讨论过连接字符串的方法，访问字符串中的单个字符的方法。不过如果无法确定在某个字符串中是否确实存在一个字符，应该调用什么方法呢？这时，可调用 indexOf() 和 lastIndexOf() 方法。

indexOf() 和 lastIndexOf() 方法返回的都是指定的子串在另一个字符串中的位置，如果没有找不到子串，则返回 -1。

这两个方法的不同之处在于，indexOf() 方法是从字符串的开头（位置 0）开始检索字符串，而 lastIndexOf() 方法则是从字符串的结尾开始检索子串。例如：

var oStringObject = new String("hello world!");

alert(oStringObject.indexOf("o")); 输出 "4"

alert(oStringObject.lastIndexOf("o")); 输出 "7"

在这里，第一个 "o" 字符串出现在位置 4，即 "hello" 中的 "o"；最后一个 "o" 出现在位置 7，即 "world" 中的 "o"。如果该字符串中只有一个 "o" 字符串，那么 indexOf() 和 lastIndexOf() 方法返回的位置相同。

### slice()(了解)和 substring()

ECMAScript 提供了两种方法从子串创建字符串值，即 slice() 和 substring()。这两种方法返回的都是要处理的字符串的子串，都接受一个或两个参数。第一个参数是要获取的子串的起始位置，第二个参数（如果使用的话）是要获取子串终止前的位置（也就是说，获取终止位置处的字符不包括在返回的值内）。如果省略第二个参数，终止位就默认为字符串的长度。

与 concat() 方法一样，slice() 和 substring() 方法都不改变 String 对象自身的值。它们只返回原始的 String 值，保持 String 对象不变。

var oStringObject = new String("hello world");

alert(oStringObject.slice("3")); //输出 "lo world"

alert(oStringObject.substring("3")); //输出 "lo world"

alert(oStringObject.slice("3", "7")); //输出 "lo w"

alert(oStringObject.substring("3", "7")); //输出 "lo w"

在这个例子中，slice() 和 substring() 的用法相同，返回值也一样。当只有参数 3 时，两个方法返回的都是 "lo world"，因为 "hello" 中的第二个 "l" 位于位置 3 上。当有两个参数 "3" 和 "7" 时，两个方法返回的值都是 "lo w"（"world" 中的字母 "o" 位于位置 7 上，所以它不包括在结果中）。

为什么有两个功能完全相同的方法呢？事实上，这两个方法并不完全相同，不过只在参数为负数时，它们处理参数的方式才稍有不同。

对于负数参数，slice() 方法会用字符串的长度加上参数，substring() 方法则将其作为 0 处理（也就是说将忽略它）。例如：

var oStringObject = new String("hello world");

alert(oStringObject.slice("-3")); //输出 "rld"

alert(oStringObject.substring("-3")); //输出 "hello world"

alert(oStringObject.slice("3, -4")); //输出 "lo w"

alert(oStringObject.substring("3, -4")); //输出 "hel"

这样即可看出 slice() 和 substring() 方法的主要不同。

当只有参数 -3 时，slice() 返回 "rld"，substring() 则返回 "hello world"。这是因为对于字符串 "hello world"，slice("-3") 将被转换成 slice("8")，而 substring("-3") 将被转换成 substring("0")。

同样，使用参数 3 和 -4 时，差别也很明显。slice() 将被转换成 slice(3, 7)，与前面的例子相同，返回 "lo w"。而 substring() 方法则将两个参数解释为 substring(3, 0)，实际上即 substring(0, 3)，因为 substring() 总把较小的数字作为起始位，较大的数字作为终止位。因此，substring("3, -4") 返回的是 "hel"。

#### toLowerCase(),toUpperCase()

最后一套要讨论的方法涉及大小写转换。有 4 种方法用于执行大小写转换，即

* toLowerCase()
* toUpperCase()

从名字上可以看出它们的用途，前一种方法用于把字符串转换成全小写的，后一种方法用于把字符串转换成全大写的。

toLowerCase() 和 toUpperCase() 方法是原始的，是以 java.lang.String 中相同方法为原型实现的。

var oStringObject = new String("Hello World");

alert(oStringObject.toUpperCase()); //输出 "HELLO WORLD"

alert(oStringObject.toLowerCase()); //输出 "hello world"

这段代码中，toUpperCase()输出的是 "HELLO WORLD"，toLowerCase()输出的都是 "hello world"。

提示：记住，String 对象的所有属性和方法都可应用于 String 原始值上，因为它们是伪对象。

### 5.5.Boolean 对象

Boolean 对象表示两个值："true" 或 "false"。

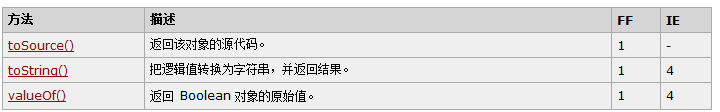
语法:

|  |
| --- |
| new Boolean(*value*); //构造函数  Boolean(*value*); //转换函数 |

当作为一个构造函数（带有运算符 new）调用时，Boolean() 将把它的参数转换成一个布尔值，并且返回一个包含该值的 Boolean 对象。

如果作为一个函数（不带有运算符 new）调用时，Boolean() 只将把它的参数转换成一个原始的布尔值，并且返回这个值。

方法:



## 5.11ECMAScript 类型转换

## 转换成字符串

ECMAScript 的 Boolean 值、数字和字符串的原始值的有趣之处在于它们是伪对象，这意味着它们实际上具有属性和方法。

例如:

* + String的length属性
  + Boolean 值、数字和字符串都有 toString() 方法,数字的基模式,(参数中输入进制)

## 转换成数字

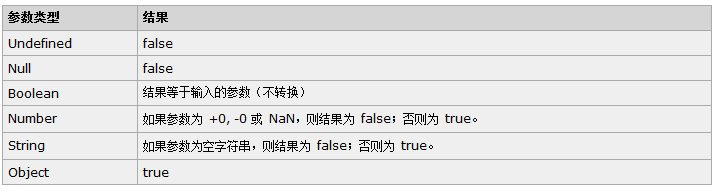
* + ECMAScript 提供了两种把非数字的原始值转换成数字的方法，即 parseInt() 和 parseFloat().parseInt()方法.方法首先查看位置 0 处的字符，判断它是否是个有效数字；如果不是，该方法将返回 NaN，不再继续执行其他操作。
    - 1. var iNum1 = parseInt("12345red"); //返回 12345
      2. var iNum1 = parseInt("0xA"); //返回 10
      3. var iNum1 = parseInt("56.9"); //返回 56
      4. var iNum1 = parseInt("red"); //返回 NaN
* parseFloat()方法

1. var fNum1 = parseFloat("12345red"); //返回 12345
2. var fNum2 = parseFloat("0xA"); //返回 NaN
3. var fNum3 = parseFloat("11.2"); //返回 11.2
4. var fNum4 = parseFloat("11.22.33"); //返回 11.22
5. var fNum5 = parseFloat("0102"); //返回 102
6. var fNum1 = parseFloat("red"); //返回 NaN

## ECMAScript的强转

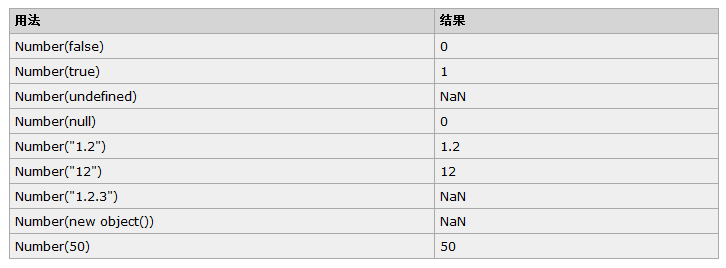
ECMAScript 中可用的 3 种强制类型转换如下：

* Boolean(value) - 把给定的值转换成 Boolean 型；
* Number(value) - 把给定的值转换成数字（可以是整数或浮点数）；
* String(value) - 把给定的值转换成字符串
* Boolean()
  + 1. var b1 = Boolean(""); //false - 空字符串
    2. var b2 = Boolean("hello"); //true - 非空字符串
    3. var b1 = Boolean(50); //true - 非零数字
    4. var b1 = Boolean(null); //false - null
    5. var b1 = Boolean(0); //false - 零
    6. var b1 = Boolean(new object()); //true - 对象



* Number()

Number() 函数的强制类型转换与 parseInt() 和 parseFloat() 方法的处理方式相似，只是它转换的是整个值，而不是部分值。



* String()

是最简单的

强制转换成字符串和调用 toString() 方法的唯一不同之处在于，对 null 和 undefined 值强制类型转换可以生成字符串而不引发错误：

var s1 = String(null); //"null"

var oNull = null;

var s2 = oNull.toString(); //会引发错误

## instanceof 运算符

在使用 typeof 运算符时采用引用类型存储值会出现一个问题，无论引用的是什么类型的对象，它都返回 "object"。ECMAScript 引入了另一个 Java 运算符 instanceof 来解决这个问题。

instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。例如：

var oStringObject = new String("hello world");

alert(oStringObject instanceof String); //输出 "true"

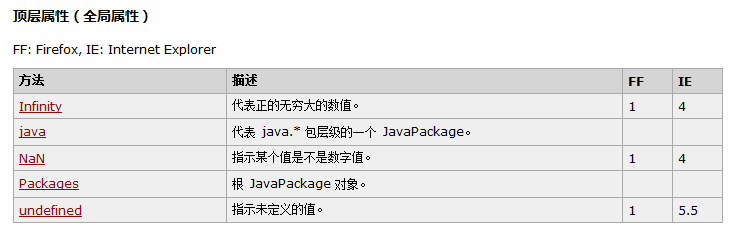
这段代码问的是“变量 oStringObject 是否为 String 对象的实例？”oStringObject 的确是 String 对象的实例，因此结果是 "true"。尽管不像 typeof 方法那样灵活，但是在 typeof 方法返回 "object" 的情况下，instanceof 方法还是很有用的。

### 5.6Global对象

全局对象是预定义的对象，作为 JavaScript 的全局函数和全局属性的占位符。通过使用全局对象，可以访问所有其他所有预定义的对象、函数和属性。全局对象不是任何对象的属性，所以它没有名称。

方法





### 5.7Math对象





### 5.8Array对象

Array 对象用于在单个的变量中存储多个值。

语法:

new Array();

new Array(*size*);

new Array(*element0*, *element1*, ..., *elementn*)

属性:



方法:



### 5.9Date 对象

Date 对象用于处理日期和时间。

语法:

var myDate=new Date()

方法:

1.new Date() 获取当前时间

2.getFullYear() 获取年份

3.getMonth() 获取月份注意 1月份结果为0

4.getHours() 小时

5.getDate() 日期

6.getMinutes() 分钟

7.getSeconds() 获取秒

8.getTime() 获取毫秒值.

9.toLocalString() 获取本地的时间格式字符串.

### 5.10RegExp对象

RegExp 对象表示正则表达式，它是对字符串执行模式匹配的强大工具。

语法:

### 直接量语法

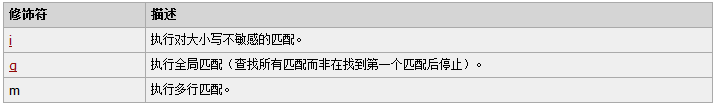
/pattern/attributes

### 创建 RegExp 对象的语法：

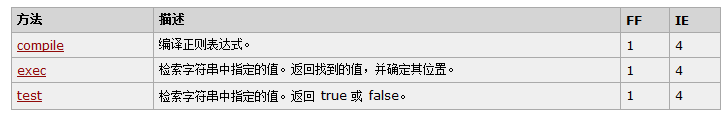
new RegExp(*pattern*, *attributes*);

参数 *pattern* 是一个字符串，指定了正则表达式的模式或其他正则表达式。

参数 *attributes* 是一个可选的字符串，包含属性 "g"、"i" 和 "m"，分别用于指定全局匹配、区分大小写的匹配和多行匹配。ECMAScript 标准化之前，不支持 m 属性。如果 *pattern* 是正则表达式，而不是字符串，则必须省略该参数。



方法:



支持正则表达式的 String 对象的方法:



# 六BOM对象

IE 3.0 和 Netscape Navigator 3.0 提供了一种特性 - BOM（浏览器对象模型），可以对浏览器窗口进行访问和操作。使用 BOM，开发者可以移动窗口、改变状态栏中的文本以及执行其他与页面内容不直接相关的动作。使 BOM 独树一帜且又常常令人怀疑的地方在于，它只是 JavaScript 的一个部分，没有任何相关的标准。

BOM 主要处理浏览器窗口和框架，不过通常浏览器特定的 JavaScript 扩展都被看做 BOM 的一部分。这些扩展包括：

* 弹出新的浏览器窗口
* 移动、关闭浏览器窗口以及调整窗口大小
* 提供 Web 浏览器详细信息的定位对象
* 提供用户屏幕分辨率详细信息的屏幕对象
* 对 cookie 的支持
* IE 扩展了 BOM，加入了 ActiveXObject 类，可以通过 JavaScript 实例化 ActiveX 对象

由于没有相关的 BOM 标准，每种浏览器都有自己的 BOM 实现。有一些事实上的标准，如具有一个窗口对象和一个导航对象，不过每种浏览器可以为这些对象或其他对象定义自己的属性和方法。

### 6.1 window对象





Self 🡪window

Top🡪window

parent🡪window



Alert();

Prompt();

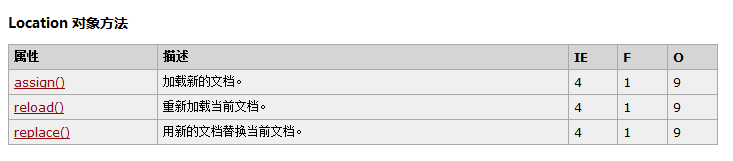
Confirm();

# 6.2History 对象

### 

### 6.3Location 对象





# 七DOM操作(DHTML)

DOM（文档对象模型）是 HTML 和 XML 的应用程序接口（API）。DOM 将把整个页面规划成由节点层级构成的文档。HTML 或 XML 页面的每个部分都是一个节点的衍生物。

## 为什么 DOM 必不可少

自从 IE 4.0 和 Netscape Navigator 4.0 开始支持不同形态的动态 HTML（DHTML），开发者首次能够在不重载网页的情况下修改它的外观和内容。这是 Web 技术的一大飞跃，不过也带来了巨大的问题。Netscape 和微软各自开发自己的 DHTML，从而结束了 Web 开发者只编写一个 HTML 页面就可以在所有浏览器中访问的时期。

业界决定必须要做点什么以保持 Web 的跨平台特性，他们担心如果放任 Netscape 和微软公司这样做，Web 必将分化为两个独立的部分，每一部分只适用于特定的浏览器。因此，负责指定 Web 通信标准的团体 W3C（World Wide Web Consortium）就开始制定 DOM。

## DOM 的各个 level

DOM Level 1 是 W3C 于 1998 年 10 月提出的。它由两个模块组成，即 DOM Core 和 DOM HTML。前者提供了基于 XML 的文档的结构图，以便访问和操作文档的任意部分；后者添加了一些 HTML 专用的对象和方法，从而扩展了 DOM Core。

注意，DOM 不是 JavaScript 专有的，事实上许多其他语言都实现了它。不过，Web 浏览器中的 DOM 已经用 ECMAScript 实现了，现在是 JavaScript 语言的一个很大组成部分。

DOM Level 1 只是一个目标，即规划文档的结构，DOM Level 2 的目标就广泛多了。对原始 DOM 的扩展添加了对鼠标和用户界面事件（DHTML 对此有丰富的支持）、范围、遍历（重复执行 DOM 文档的方法）的支持，并通过对象接口添加了对 CSS（层叠样式表）的支持。由 Level 1 引入的原始 DOM Core 也加入了对 XML 命名空间的支持。

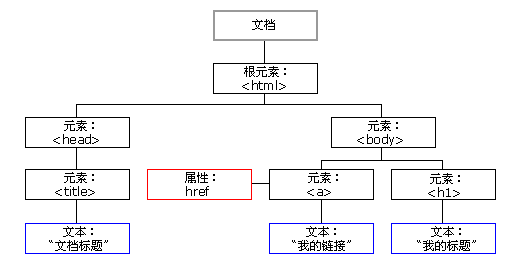
DOM Level 2 引入了几种 DOM 新模块，用于处理新的接口类型：

* DOM 视图 - 描述跟踪文档的各种视图（即 CSS 样式化之前和 CSS 样式化之后的文档）
* DOM 事件 - 描述事件的接口
* DOM 样式 - 描述处理基于 CSS 样式的接口
* DOM 遍历和范围 - 描述遍历和操作文档树的接口

DOM Level 3 引入了以统一的方式载入和保持文档的方法（包含在新模块 DOM Load and Save）以及验证文档（DOM Validation）的方法，从而进一步扩展了 DOM。在 Level 3 中，DOM Core 被扩展为支持所有的 XML 1.0 特性，包括 XML Infoset、XPath 和 XML Base。

在学习 DOM 时，可能会遇到有人引用 DOM Level 0。注意，根本没有 DOM Level 0 这个标准，它只是 DOM 的一个历史参考点（DOM Level 0 指的是 IE 4.0 和 Netscape Navigator 4.0 中支持的原始 DHTML）。

Dom树:



## 7.1什么是 DOM？

DOM 是 W3C（万维网联盟）的标准。

DOM 定义了访问 HTML 和 XML 文档的标准：

“W3C 文档对象模型（DOM）是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。”

W3C DOM 标准被分为 3 个不同的部分：

* 核心 DOM - 针对任何结构化文档的标准模型
* XML DOM - 针对 XML 文档的标准模型
* HTML DOM - 针对 HTML 文档的标准模型

注：DOM 是 Document Object Model（文档对象模型）的缩写。

* 什么是 XML DOM？

XML DOM 定义了所有 XML 元素的对象和属性，以及访问它们的方法。

* 什么是 HTML DOM？
* HTML 的标准对象模型
* HTML 的标准编程接口
* W3C 标准

HTML DOM 定义了所有 HTML 元素的对象和属性，以及访问它们的方法。

## 7.2DOM 节点

根据 W3C 的 HTML DOM 标准，HTML 文档中的所有内容都是节点(NODE)：

* 整个文档是一个文档节点
* 每个 HTML 元素是元素节点
* HTML 元素内的文本是文本节点
* 每个 HTML 属性是属性节点
* 注释是注释节点

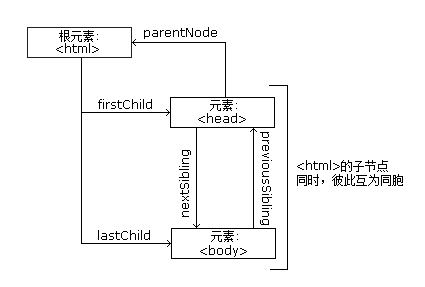
## 7.3节点父、子和同胞

节点树中的节点彼此拥有层级关系。

父（parent）、子（child）和同胞（sibling）等术语用于描述这些关系。父节点拥有子节点。同级的子节点被称为同胞（兄弟或姐妹）。

* 在节点树中，顶端节点被称为根（root）
* 每个节点都有父节点、除了根（它没有父节点）
* 一个节点可拥有任意数量的子
* 同胞是拥有相同父节点的节点

下面的图片展示了节点树的一部分，以及节点之间的关系：



注意:<title>ITCAST</title>，元素节点 <title>，包含值为 "ITCAST " 的文本节点。使用innerHTML获取.

## 7.4HTML DOM Event

HTML 4.0 的新特性之一是有能力使 HTML 事件触发浏览器中的动作（action），比如当用户点击某个 HTML 元素时启动一段 JavaScript。下面是一个属性列表，这些属性可插入 HTML 标签来定义事件动作。



## Event 对象

Event 对象代表事件的状态，比如事件在其中发生的元素、键盘按键的状态、鼠标的位置、鼠标按钮的状态。

事件通常与函数结合使用，函数不会在事件发生前被执行！



## 7.5HTML DOM 方法

导航属性:

* parentNode - 节点（元素）的父节点
* firstChild – 节点下第一个子元素
* lastChild – 节点下最后一个子元素
* childNodes - 节点（元素）的子节点

节点属性:

* attributes - 节点（元素）的属性节点
* nodeType – 节点类型
* nodeValue – 节点值
* nodeName – 节点名称
* innerHTML - 节点（元素）的文本值



## 7.5访问 HTML 元素（节点）

访问 HTML 元素等同于访问节点

您能够以不同的方式来访问 HTML 元素：

* 通过使用 getElementById() 方法
* 通过使用 getElementsByTagName() 方法
* 通过使用 getElementsByClassName() 方法

## 7.6导航节点关系

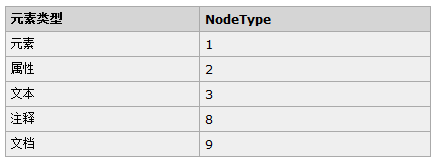
您能够使用以下节点属性：

* parentNode
* firstChild
* lastChild
* childNodes

在文档结构中进行导航。

## 7.6HTML DOM 属性

* innerHTML属性:获取元素内容
* nodeName属性:获取节点的名称。
* nodeValue属性:获取节点的值。
* nodeType属性:获取节点的类型,返回值是数字(配图)



## 7.7增删改查演示

###### 7.7.1访问 HTML 元素（节点）

* 通过使用 getElementById() 方法
* 通过使用 getElementsByTagName() 方法
* 通过使用 getElementsByClassName() 方法
* 通过使用getElementsByName()方法

###### 7.7.2 HTML 元素的增删改查

增:

* 1. createElement(name)创建元素
  2. appendChild();将元素添加

删:

1. 获得要删除的元素
2. 获得它的父元素
3. 使用removeChild()方法删除

改:

第一种方式:

* + - 1. 使用上面增和删结合完成修改

第二中方式:

使用setAttribute();方法修改属性

使用innerHTML属性修改元素的内容

查:

使用之前介绍的方法.

修改 HTML DOM 包括以下的方面：

* 改变 HTML 内容

改变元素内容的最简答的方法是使用 innerHTML 属性。

* 改变 CSS 样式

|  |
| --- |
| <p id="p2">Hello world!</p>  document.getElementById("p2").style.color="blue"; |

* 改变 HTML 属性

elementNode.setAttribute(name,value)

* 创建新的 HTML 元素

createElement(name)

* 删除已有的 HTML 元素

elementNode.removeChild(node)

* 改变事件（处理程序）