JavaScript

- ECMAScript
- DOM
- Bom

JavaScript 简介

JavaScript 一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为 JavaScript 引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在 HTML 网页上使用，用来给 HTML 网页增加动态功能。

在 1995 年时，由 Netscape 公司的 Brendan Eich，在网景导航者浏览器上首次设计实现而成。因为 Netscape 与 Sun 合作， Netscape 管理层希望它外观看起来像 Java，因此取名为 JavaScript。

但为了取得技术优势，微软推出了 JScript， CEnvi 推出 ScriptEase，与 JavaScript 同样可在浏览器上运行。为了统一规格，由来自 Netscape、 Sun、微软、 Borland 和其他一些对脚本编程感兴趣的公司的程序员组成的 TC39（第 39 技术委员会）锤炼出了 ECMA-262，该标准定义了名为 ECMAScript 的全新脚本语言。因为 JavaScript 兼容于 ECMA 标准，因此也称为 ECMAScript。

JavaScript 是一种基于对象和事件驱动，并具有安全性的脚本语言。它与 HTML,CSS 结合起来，用于增强功能，并提高与最终用于之间的交互性能。客户端的 JavaScript 必须要有解释器的支持。 JavaScript 代码是解释型的。不需要编译，而是作为 HTML 文件的一部分由解释器解释执行。目前，所有的浏览器都内置 JavaScript 的解释器

JavaScript 的组成部分

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：

（1）核心 ECMAScript：定义了基本的语法和一些对象。每种 Web 浏览器都有它自己对 ECMAScript 标准的实现。

（2）文档对象模型 DOM(Document Object Model)：它是 HTML 和 XML 文档的应用程序编程接口。Web 浏览器中的 DOM 把整个页面规划成由节点层级构成的文档。用 DOM API 可以轻松地删除、添加和替换节点。DOM 不是 JavaScript 专有的。许多语言都实现了它。

（3）浏览器对象模型 BOM(Browser Object Model)：描述了与浏览器窗口进行访问和操作的方法和接口（获取浏览器窗口的宽高，使用浏览器的自带提示框等）。BOM 暂时没有相关的标准，每种浏览器对 BOM 的实现有些差别。

ECMAScript 基础

语法:

熟悉 Java 和 C 这些语言的开发者会发现 ECMAScript 的语法很容易掌握，因为它借用了这些语言的语法。Java 和 ECMAScript 有一些关键的语法特性相同，也有一些完全不同。以下是一些常用的语法:

（1）变量、函数名、运算符以及其他一切东西都是区分大小写的。
（2）定义变量时只用 var 运算符，可以将它初始化为任意值。也可以直接使用一个变量，无需在使用这个变量之前对它声明。
（3）单行注释以双斜杠开头（//）多行注释以单斜杠和星号开头（/*），以星号和单斜杠结尾（*/）。
（4）没有类的概念，每个方法需要用关键字 function 声明。
（5）需要返回值的方法无需特殊声明（像 Java 中的 void），直接使用 return 语句返回值。

ECMAScript 基础

原始类型:

ECMAScript 有 5 种原始类型，即 Undefined、Null、Boolean、Number 和 String。可以用 typeOf 运算符来返回变量或值的类型。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

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。

ECMAScript 基础

原始类型:

undefined: 当一个变量未初始化时，该变量的默认值是 undefined。对 undefined 的变量做除了 typeOf 运算外的其它任何运算将会出现 错误。

null: 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。 alert(null == undefined) 输出为 true。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。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。

Number: 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。 Number.MAX_VALUE 和 Number.MIN_VALUE，它们定义了 Number 值集合的外边界。所有 ECMAScript 数都必须在这两个值之间。不过计算生成的数值结果可以不落在这两个值之间。

ECMAScript 基础

原始类型:

ECMAScript 有专门的值表示无穷大， Number.POSITIVE_INFINITY 的值为 Infinity 。 Number.NEGATIVE_INFINITY 的值为 -Infinity 。可以对任何数调用 isFinite() 方法，来判断该数是不是无穷大。

最后一个特殊值是 NaN ，表示非数。 NaN 是个奇怪的特殊值。一般说来，这种情况发生在类型（ String 、 Boolean 等）转换失败时。与无穷大一样， NaN 也不能用于算术计算。 NaN 的另一个奇特之处在于，它与自身不相等， alert(NaN == NaN); 输出为 "false" 。

所以 ECMAScript 用一个函数 isNaN() 判断一个变量是否转换为数字是否为 NaN 。 alert(isNaN("blue")); 输出为 "true" 。

ECMAScript 基础

类型转换:

ECMAScript 定义的全部对象都有 toString() 方法。 Number 的 toString() 方法比较特殊，它有两种模式，默认模式和基模式。默认模式返回的都是数字的十进制表示。

```
var iNum1 = 10; alert(iNum1.toString());    //输出 "10"
var iNum2 = 10.0; alert(iNum2.toString()); //输出 "10"
```

采用 Number 类型的 toString() 方法的基模式，可以用不同的基输出数字，例如二进制的基是 2，八进制的基是 8，十六进制的基是 16。

```
var iNum = 10;
alert(iNum.toString(2));  //输出 "1010"
alert(iNum.toString(8));  //输出 "12"
alert(iNum.toString(16)); //输出 "A"
var iNum = 18;alert(iNum.toString(2));//输出 "10010" 前面 0 省略
```

ECMAScript 基础

类型转换:

ECMAScript 提供了两种把 string 类型的值转换成数字的方法，parseInt() 和 parseFloat() 。这两个方法都是从位置 0 开始查看每个字符，直到找到第一个非有效的字符为止，然后把该字符之前的字符串转换成数字。例如，如果要把字符串 "12345red" 转换成整数，那么 parseInt() 将返回 12345 。

```
var iNum1 = parseInt("12345red");    //返回 12345
var iNum1 = parseInt("AF", 16);  //返回 175
var iNum1 = parseInt("56.9");//返回 56
var iNum1 = parseInt("red"); //返回 NaN
```

如果十进制数包含前导 0，那么最好采用基数 10，这样才不会意外地得到八进制的值。例如:

```
var iNum1 = parseInt("010"); //返回 8
var iNum2 = parseInt("010", 8);  //返回 8
var iNum3 = parseInt("010", 10); //返回 10
```

parseFloat() 方法没有基模式。

```
var fNum4 = parseFloat("11.22.33"); //返回 11.22
var fNum5 = parseFloat("0102");  //返回 102
```

ECMAScript 基础

强制类型转换:

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() 函数:
当要转换的值是至少有一个字符的字符串、非 0 数字或对象时, Boolean() 函数将返回 true。如果该值是空字符串、数字 0, undefined 或 null, 它将返回 false。

Number() 函数:
用 Number() 进行强制类型转换, 转换的是整个值, 而不是部分值。转换值 "1.2.3" 将返回 NaN, 因为整个字符串值不能转换成数字。如果字符串值能被完整地转换, Number() 将判断是调用 parseInt() 方法还是 parseFloat() 方法。

ECMAScript 运算符

一元运算符只有一个参数，即要操作的对象或值。它们是 ECMAScript 中最简单的运算符。

delete 运算符删除对以前定义的对象属性或方法的引用。例如:

```
var o = new Object;
o.name = "David";
alert(o.name);  // 输出 "David"
delete o.name;
alert(o.name);  // 输出 "undefined"
```

在这个例子中，删除了 name 属性，意味着强制解除对它的引用，将其设置为 undefined（即创建的未初始化的变量的值）。delete 运算符不能删除开发者未定义的属性和方法。下面的代码将引发错误:

```
delete o.toString;
```

即使 toString 是有效的方法名，这行代码也会引发错误，因为 toString() 方法是原始的 ECMAScript 方法，不是开发者定义的。

ECMAScript 运算符

一元加法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。另一方面，与一元加法运算符相似，一元减法运算符也会把字符串转换成近似的数字，此外还会对该值求负。

```
var sNum = "20";
var iNum = -sNum;
alert(iNum);      // 输出 "-20"
alert(typeof iNum); // 输出 "number"
```

ECMAScript 运算符

ECMAScript 比较一个数字和一个字符串时，会把字符串转换成数字，然后按照数字顺序比较它们。

```
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。如果字符串不能转换成数字比较结果返回 false。

ECMAScript 中判断两个变量相等有等号和全等号两种方法。全等号由三个等号表示（===），只有在无需类型转换运算数就相等的情况下才返回 true。

```
null == undefined   true
null === undefined  false
"NaN" == NaN        false
5 == NaN            false
NaN == NaN          false
false == 0          true
false === 0         false
true == 1           true
undefined == 0      false
Null == 0           false
"5" == 5            true
"5" === 5           false
```

ECMAScript 语句

ECMAScript 中的基本语句有:

if 语句: if(true){} else if(true){} else{}

迭代语句: while(true){}, do{} while(true)

异常捕获语句: try{} catch(error){}

switch 语句: switch(value){case value1:break; default:...;}

break 语句: 跳出当前循环

continue 语句: 跳出本次循环

标签语句: label : statement (一般和 continue 或 break 连用, 表示跳转到代码的指定位置, 样例代码如下)

```
var iNum = 0;
outermost:for (var i=0; i<10; i++) {
              for (var j=0; j<10; j++) {
                  if (i == 5 && j == 5) {
                      break outermost;
                  }
                  iNum++;
              }
          }
alert(iNum);  //输出 "55", 标签 outermost 表示的是第一个 for 语句
```

ECMAScript 函数

ECMAScript 中的函数是一组可以随时随地运行的语句。函数是 ECMAScript 的核心。函数是由这样的方式进行声明的：关键字 function、函数名、一组参数，以及置于括号中的待执行代码。

```
function functionName(arg0, arg1, ... argN) {
    statements
}
```

在函数代码中，使用特殊对象 arguments，开发者无需明确指出参数名，就能访问它们。例如，在函数 sayHi() 中，第一个参数是 message。用 arguments[0] 也可以访问这个值，即第一个参数的值（第一个参数位于位置 0，第二个参数位于位置 1，依此类推）。因此，无需明确命名参数，就可以重写函数：

```
function sayHi() {
    if (arguments[0] == "bye") {
        return;
    }
    alert(arguments[0]);
}
```

ECMAScript 函数

arguments 模拟函数重载

用 arguments 对象判断传递给函数的参数个数，即可模拟函数重载:

```
function doAdd() {
  if(arguments.length == 1) {
     alert(arguments[0] + 5);
  } else if(arguments.length == 2) {
     alert(arguments[0] + arguments[1]);
  }
}
doAdd(10);  //输出 "15"
doAdd(40, 20); //输出 "60"
```

在声明函数时不指定参数，在调用函数时给函数传入参数，函数中可以用 arguments.length 来判断传入参数的个数，然后执行对应的方法。当只有一个参数时，doAdd() 函数给参数加 5。如果有两个参数，则会把两个参数相加，返回它们的和。

ECMAScript 函数

Function 对象

在 ECMAScript 中也可以这样创建函数（最后一个参数是函数主体，即要执行的代码）：

```
var sayHi = new Function("sName", "sMessage", "alert(\"Hello \" + sName + sMessage);");
```

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。当然函数 sayHi 也可以当作参数传递给另外一个函数。

ECMAScript 中的 function 函数可以接受最多 25 个参数，它的属性 length 只是为查看默认情况下预期的参数个数。（alert(sayHi .length); 输出 2 ）

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。

ECMAScript 函数

闭包：指的是词法表示包括不被计算的变量的函数，也就是说，函数可以使用函数之外定义的变量。

闭包是 ECMAScript 中非常强大多用的一部分，可用于执行复杂的计算。在一个函数中定义另一个会使闭包变得更加复杂。例如：

```
var iBaseNum = 10;
Function thisNumber(){     // 简单的闭包
  alert(iBaseNum);
}
function addNum(iNum1, iNum2) {    // 复杂的闭包
  return Function() {
    return iNum1 + iNum2 + iBaseNum;
  }
}
```

ECMAScript 对象

ECMA-262 把对象（object）定义为“属性的无序集合，每个属性存放一个原始值、对象或函数”。严格来说，这意味着对象是无特定顺序的值的数组。ECMAScript 中的类并不真正存在，我们也把对象定义叫做类，因为大多数开发者对此术语更熟悉，而且从功能上说，两者是等价的。一种面向对象语言需要向开发者提供四种基本能力：

封装 - 把相关的信息（无论数据或方法）存储在对象中的能力
聚集 - 把一个对象存储在另一个对象内的能力
继承 - 由另一个类（或多个类）得来类的属性和方法的能力
多态 - 编写能以多种方法运行的函数或方法的能力

ECMAScript 支持这些要求，因此可被是看做面向对象的。

对象的创建方式是用关键字 new 后面跟上实例化的类的名字：

```
var oObject = new Object();
```

废除一个对象时需要把对象的所有引用都设置为 null。

```
oObject = null;
```

ECMAScript 对象

一般来说，可以创建并使用的对象有三种：本地对象、内置对象和宿主对象。

ECMA-262 把本地对象定义为“独立于宿主环境（用户的机器环境和浏览器等）的 ECMAScript 实现提供的对象”。简单来说，本地对象就是 ECMA-262 定义的类（引用类型）。它们包括：

Object， Function， Array， String， Boolean， Number， Date
RegExp， Error， EvalError， RangeError， ReferenceError
SyntaxError， TypeError， URIError

ECMA-262 把内置对象（built-in object）定义为“由 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）。

所有非本地对象都是宿主对象，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。

ECMAScript 对象

Function对象提供的内置函数有（常用的）：

decodeURI()　解码某个编码的 URI。
decodeURIComponent() 解码一个编码的 URI 组件。
encodeURI()　把字符串编码为 URI。
encodeURIComponent() 把字符串编码为 URI 组件。
escape()　对字符串进行编码。
eval() 计算 JavaScript 字符串，并把它作为脚本代码来执行。
getClass() 返回一个 JavaObject 的 JavaClass。
isFinite() 检查某个值是否为有穷大的数。
isNaN()检查某个值是否是数字。
Number()　把对象的值转换为数字。
parseFloat()　解析一个字符串并返回一个浮点数。
parseInt() 解析一个字符串并返回一个整数。
String()　把对象的值转换为字符串。
unescape() 对由 escape() 编码的字符串进行解码。

ECMAScript 对象

Array 对象提供的内置函数有（常用的）：

concat()　连接两个或更多的数组，并返回结果。
join()　把数组的所有元素放入一个字符串。元素通过指定的分隔符进行分隔。
pop()　删除并返回数组的最后一个元素
push()　向数组的末尾添加一个或更多元素，并返回新的长度。
reverse()　颠倒数组中元素的顺序。
shift()删除并返回数组的第一个元素
slice()从某个已有的数组返回选定的元素
sort()　对数组的元素进行排序
splice()　删除元素，并向数组添加新元素。
toSource()返回该对象的源代码。
toString()把数组转换为字符串，并返回结果。
toLocaleString()　把数组转换为本地数组，并返回结果。
unshift()　向数组的开头添加一个或更多元素，并返回新的长度。
valueOf()　返回数组对象的原始值

ECMAScript 对象

Date 对象提供的内置函数有（常用的）：

Date() 返回当日的日期和时间。
getDate() 从 Date 对象返回一个月中的某一天 (1 ~ 31)。
getDay() 从 Date 对象返回一周中的某一天 (0 ~ 6)。
getMonth() 从 Date 对象返回月份 (0 ~ 11)。
getFullYear() 从 Date 对象以四位数字返回年份。
getYear() 请使用 getFullYear() 方法代替。
getHours() 返回 Date 对象的小时 (0 ~ 23)。
getMinutes() 返回 Date 对象的分钟 (0 ~ 59)。
getSeconds() 返回 Date 对象的秒数 (0 ~ 59)。
getMilliseconds() 返回 Date 对象的毫秒 (0 ~ 999)。
getTime() 返回 1970 年 1 月 1 日至今的毫秒数。

ECMAScript 对象

String 对象提供的内置函数有（常用的）：

toLocaleLowerCase()　把字符串转换为小写。
toLocaleUpperCase()　把字符串转换为大写。
toLowerCase() 把字符串转换为小写。
toUpperCase() 把字符串转换为大写。
substr()　从起始索引号提取字符串中指定数目的字符。
substring()　提取字符串中两个指定的索引号之间的字符。
split()把字符串分割为字符串数组。
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。
lastIndexOf() 从后向前搜索字符串。
indexOf()　检索字符串。

javascript 的 string 支持使用正则表达式匹配，这样便可以使用正则来检查一些字符串（例如用户输入的邮箱）是符合法，正则表达式的使用这里就不再详细介绍了。

ECMAScript 对象

javascript 在传递一些参数或者调用变量时，会看到使用 this 关键字。为什么使用 this 呢？因为在实例化对象时，总是不能确定开发者会使用什么样的变量名。使用 this，即可在任何多个地方重用同一个函数。

```
function showColor() {
  alert(this.color);
};
var oCar1 = new Object;
oCar1.color = "red";
oCar1.showColor = showColor;

var oCar2 = new Object;
oCar2.color = "blue";
oCar2.showColor = showColor;

oCar1.showColor();        //输出 "red"
oCar2.showColor();        //输出 "blue"
```

ECMAScript 定义类

Javascipt 语法不支持＂类＂，导致传统的面向对象编程方法无法直接使用。程序员们做了很多探索，研究如何用 Javascript 模拟＂类＂。在面向对象编程中，类是对象的模板，定义了同一组对象（又称＂实例＂）共有的属性和方法。Javascript 语言不支持＂类＂，但是可以用一些变通的方法，模拟出＂类＂。本章总结了 Javascript 定义＂类＂的三种方法（其实有很多种，这里只说明三种方法）：构造函数法，Object.create() 法，极简主义法（个人比较推荐的方法）。

ECMAScript 定义类

构造函数法

这是经典方法，也是教科书必教的方法。它用构造函数模拟＂类＂，在其内部用 this 关键字指代实例对象。

```
function Cat(name) {
    this.name = name;
}
```

生成实例的时候，使用 new 关键字。

```
var cat1 = new Cat("大毛");
alert(cat1.name); // 输出“大毛”
```

类的属性和方法，还可以定义在构造函数的 prototype 对象之上。这样避免了在 new 多个对象时重复的创建这个（值一样）属性。

```
Cat.prototype.sound = "喵喵喵";
var cat1 = new Cat("大毛");
alert(cat1.sound); // 输出“喵喵喵”
```

ECMAScript 定义类

Object.create() 法

用这个方法，"类"就是一个对象，不是函数。

```
var Cat = {
    name: "大毛",
    setName: function(){ this.name="喵喵喵"; }
};
```

然后，直接用 Object.create() 生成实例，不需要用到 new 。

```
var cat1 = Object.create(Cat);
alert(cat1.name); // 大毛
cat1.sound();
alert(cat1.name); // 喵喵喵
```

目前，各大浏览器的最新版本（包括 IE9 ）都部署了这个方法（版本比较旧的浏览器可以用其它写法实现，这里不再例举）。种方法比"构造函数法"简单，但是不能实现私有属性和私有方法，实例对象之间也不能共享数据，对"类"的模拟不够全面，而且实例化后修改参数比较麻烦。

ECMAScript 定义类

极简主义法

它也是用一个对象模拟"类"。在这个类里面，定义一个构造函数createNew() 用来生成实例。然后，在createNew() 里面，定义一个实例对象，把这个实例对象作为返回值。

```
var Cat = {
    createNew: function(){
        var cat = {};// 声明对象时的大括号内不能有内容
        cat.name = "大毛";
        cat.makeSound = "喵喵喵"; };
        return cat;
    }
};
```

使用的时候，调用createNew() 方法，就可以得到实例对象。

```
var cat1 = Cat.createNew();
alert ( cat1.makeSound); // 喵喵喵
```

这种方法的好处是，容易理解，结构清晰优雅，符合传统的"面向对象编程"的构造。

ECMAScript 定义类

极简主义法的数据共享

有时候，我们需要所有实例对象，能够读写同一项内部数据。这个时候，只要把这个内部数据，封装在类对象的里面、 createNew() 方法的外面即可。

```
var Cat = {
    sound : "喵喵喵",
    createNew: function(){
        var cat = {};
        cat.makeSound = function(){ alert(Cat.sound); };
        cat.changeSound = function(x){ Cat.sound = x; };
        return cat;
    }
};
```

上述代码的 sound 变量，实例对象不能直接对它进行编辑，可以通过调用方法 changeSound （）对它修改，或者用 Cat 对象本身对它进行修改。如果修改了共享的数据（ sound ），所有的实例对象都会受到影响。

ECMAScript 继承

要用 ECMAScript 实现继承机制，您可以从要继承的基类入手。所有开发者定义的类都可作为基类。出于安全原因，本地类和宿主类不能作为基类，这样可以防止公用访问编译过的浏览器级的代码，因为这些代码可以被用于恶意攻击。

创建的子类将继承超类的所有属性和方法，包括构造函数及方法的实现。记住，所有属性和方法都是公用的，因此子类可直接访问这些方法。子类还可添加超类中没有的新属性和方法，也可以覆盖超类的属性和方法。

继承的方式和其他功能一样，ECMAScript 实现继承的方式不止一种。这是因为 JavaScript 中的继承机制并不是明确规定的，而是通过模仿实现的。这意味着所有的继承细节并非完全由解释程序处理。下面将介绍五种 javascript 的继承方式，每种方式各有优缺点，在你需要实现继承的时候，你可以选择最适用的继承方式。

ECMAScript 继承

对象冒充

用构造函数定义类时，可以选中这种方法比较适合。构造函数使用 this 关键字给所有属性和方法赋值。因为构造函数只是一个函数，所以可使 ClassA 构造函数成为 ClassB 的方法，然后调用它。ClassB 就会收到 ClassA 的构造函数中定义的属性和方法。

```
function ClassA(sColor) {
    this.color = sColor;
    this.sayColor = function () {
        alert(this.color);
    };
}
function ClassB(sColor) {
    this.newMethod = ClassA;// 函数名只是指向该函数的指针
    this.newMethod(sColor);
    delete this.newMethod;
}
```

对象冒充可以支持多重继承。也就是说，一个类可以继承多个超类。如果多个超类具有同名的属性或方法，则按照后继承的的超类的属性或方法为准（后来的会覆盖前面的）。

ECMAScript 继承

call() 方法和 apply() 方法

call() 方法是与经典的对象冒充方法最相似的方法。它的第一个参数用作 this 的对象。其他参数都直接传递给函数自身。原理是将父对象的构造函数绑定在子对象上。例如:

```
function sayColor(sPrefix,sSuffix) {
        alert(sPrefix + this.color + sSuffix);
};
var obj = new Object();
obj.color = "blue";
sayColor.call(obj, "Color is ", "a very nice color.");
```

在这个例子中，函数 sayColor() 在对象外定义，即使它不属于任何对象，也可以引用关键字 this。对象 obj 的 color 属性等于 blue。调用 call() 方法时，第一个参数是 obj，说明应该赋予 sayColor() 函数中的 this 关键字值是 obj。第二个和第三个参数是字符串。它们与 sayColor() 函数中的参数 sPrefix 和 sSuffix 匹配，最后生成的消息 "Color is blue, a very nice color." 将被显示出来。

apply() 方法与 call() 方法用法一样，不同之处是 apply() 方法中传递的参数是数组的形式，用一个数组存放需要传递的参数，样例如下:

```
sayColor.apply(obj, new Array("Color is ", "a very nice color."));
```

ECMAScript 继承

原型链

prototype 对象的任何属性和方法都被传递给那个类的所有实例。原型链利用这种功能来实现继承机制。

```
function ClassA() {}
ClassA.prototype.color = "blue";
ClassA.prototype.sayColor = function () {alert(this.color);};
function ClassB() {}
ClassB.prototype = new ClassA();
```

这里把 ClassB 的 prototype 属性设置成 ClassA 的实例。这很有意思，因为想要 ClassA 的所有属性和方法，但又不想逐个将它们 ClassB 的 prototype 属性。

注意：调用 ClassA 的构造函数，没有给它传递参数。这在原型链中是标准做法。要确保构造函数没有任何参数。

与对象冒充相似，子类的所有属性和方法都必须出现在 prototype 属性被赋值后，因为在它之前赋值的所有方法都会被删除。因为 prototype 属性被替换成了新对象，添加了新方法的原始对象将被销毁。所以，为 ClassB 类添加其它属性需要放在实例化 ClassA 之后： ClassB.prototype.name = "ClassB";

任何一个 prototype 对象都有一个 constructor 属性，指向它的构造函数。原型链的缺点是改变了 constructor 指向了父类，导致继承链的紊乱。所以需要手动修改回来： ClassB.prototype.constructor = ClassB 。

ECMAScript 继承

混合方式

这种继承方式使用构造函数定义类。对象冒充的主要问题是必须使用构造函数方式，如果使用原型链，就无法使用带参数的构造函数了。所以为了更好的实现继承，就两种方式混合使用。

```
function ClassA(sColor) {this.color = sColor;}
ClassA.prototype.sayColor = function () {alert(this.color);};
function ClassB(sColor, sName) {
    ClassA.call(this, sColor);// 为构造函数赋值
    this.name = sName;
}
ClassB.prototype = new ClassA();// 原型模式继承
ClassB.prototype.sayName = function () {alert(this.name);};
```

ECMAScript 继承

前面我们提到了极简主义法定义类，这种方法的继承比较简单，只要在前者的 createNew() 方法中，调用后者的 createNew() 方法即可。

```
var Animal = {
        Color = "white";
        createNew: function(){
            var animal = {};
            animal.sleep = function(){ alert("睡懒觉"); };
            return animal;
        }
    };
```

然后，在 Cat 的 createNew() 方法中，调用 Animal 的 createNew() 方法。

```
    var Cat = {
        createNew: function(){
            var cat = Animal.createNew();
            cat.name = "大毛";
            cat.makeSound = function(){ alert("喵喵喵"); };
            return cat;
        }
```

这种方法不能继承父类中的公有方法或对象。例如上例不能继承 color 属性。

JavaScript 总结

javascript 在 web 端的应用非常广泛，并且各种浏览器对它都有支持。本 PPT 只介绍了 javascript 的核心 ECMAScript 的一部分，还有它的 DOM 和 B O M没有介绍。在 javascript 上还有许多值得我们学习的内容。只有实践，才能学到更多。

T H A N K S