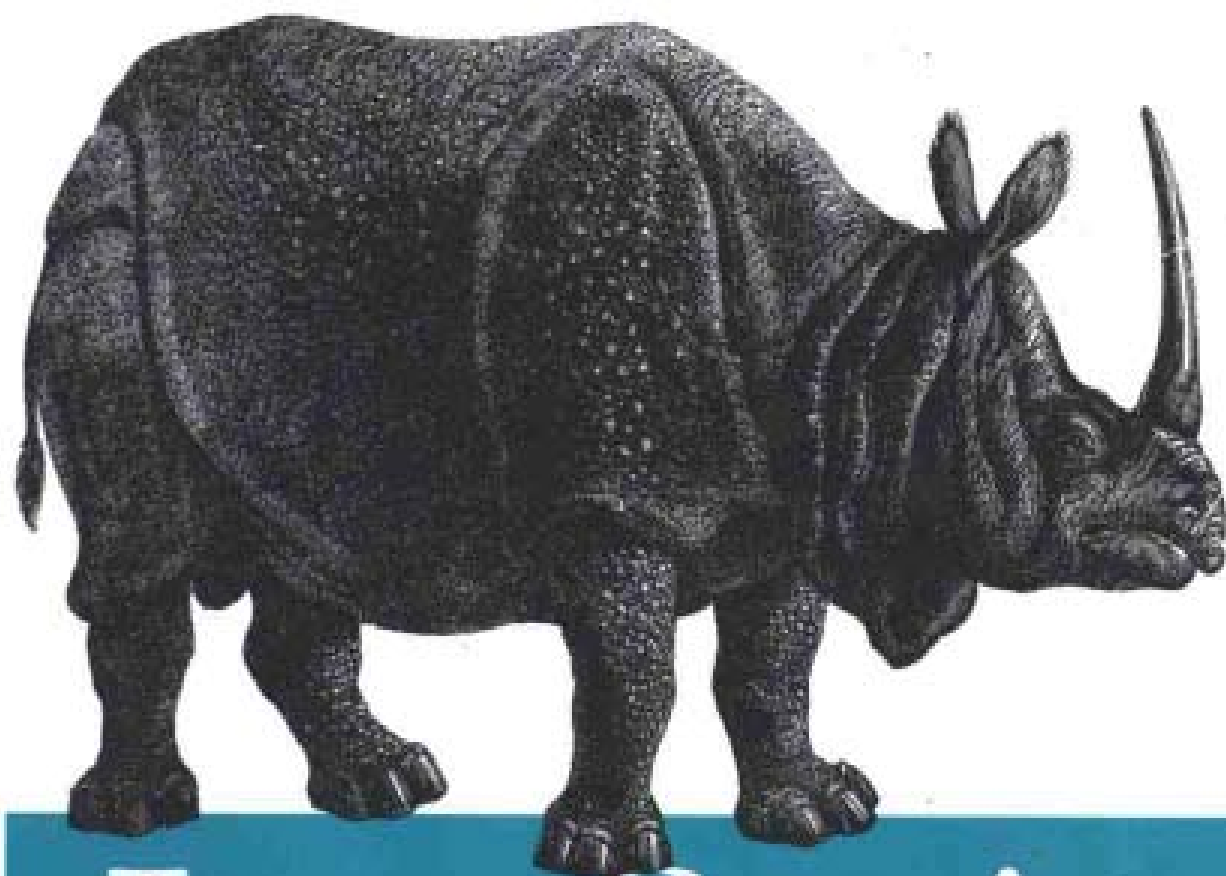


JavaScript: The Definitive Guide

第四版
涵盖 JavaScript 1.5



JavaScript

权威指南

O'REILLY®

机械工业出版社
China Machine Press



David Flanagan 著

张铭泽 等译

JavaScript 权威指南



JavaScript 是一种功能强大的基于对象的脚本语言。JavaScript 程序可以直接嵌入 HTML 页面。与 Web 浏览器定义的文档对象模型 (DOM) 一起使用时, JavaScript 可以创建动态 HTML (DHTML) 内容, 允许用户与客户端的 Web 应用程序交互。

JavaScript 语法以流行的程序设计语言 C、C++ 和 Java 为基础, 因此, 经验丰富的程序设计人员可以很快地熟悉和掌握。此外, JavaScript 是一种解释性脚本语言, 提供了比其他语言更加灵活、更加宽松的程序设计环境, 程序设计新手在这种环境中能够很快适应。

《JavaScript 权威指南》全面介绍了 JavaScript 语言的核心, 以及 Web 浏览器中实现的遗留和标准的 DOM。它运用了一些复杂的例子, 说明如何处理验证表单数据, 使用 cookie, 创建可移植的 DHTML 动画等常见任务。本书还包括详细的参考手册, 涵盖了 JavaScript 的核心 API, 遗留的客户端 API 和 W3C 标准 DOM API, 记述了这些 API 中的每一个 JavaScript 对象、方法、性质、构造函数、常量和事件处理程序。

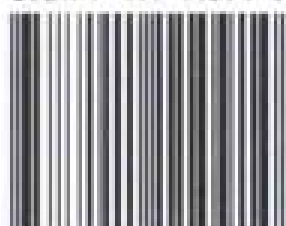
这本最畅销的 JavaScript 参考书的第四版已经进行了全面的更新, 其中涵盖 JavaScript 1.5 (ECMAScript Version 3) 的详细介绍。它还提供了 W3C DOM 标准 (1 级和 2 级) 的完整内容, 为了向后兼容, 本书保持了遗留的 0 级 DOM 的资料。

《JavaScript 权威指南》是 JavaScript 程序设计者的完整指南和参考手册。对于使用最新的、遵守标准的 Web 浏览器 (如 Internet Explorer 6、Netscape 6 和 Mozilla) 的开发者, 它尤其有用。HTML 作者可以从中学学习如何用 JavaScript 创建动态网页。经验丰富的程序设计者可以从中快速地找到编写复杂 JavaScript 程序需要的信息。本书对所有 JavaScript 程序设计者来说都是绝对必要的。

“本书是 JavaScript 程序员的必备参考……组织得很好, 而且非常详细。”

—Brendan Eich, JavaScript 之父

ISBN 7-111-11091-9



9 787111 110910 >

©Reilly & Associates, Inc. 授权机械工业出版社出版

ISBN 7-111-11091-9

定价: 99.00 元

JavaScript 权威指南

第四版

David Flanagan 著

张铭泽 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly & Associates, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目 (CIP) 数据

JavaScript 权威指南 (第四版) / (美) 弗莱 (Flanagan, D.) 著; 张铭泽等译. 北京: 机械工业出版社, 2003.1

书名原文: JavaScript: The Definitive Guide, Fourth Edition

ISBN 7-111-11091-9

I. J... II. ①弗... ②张... III. Java 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2002) 第 081228 号

北京市版权局著作权合同登记

图字: 01-2002-1831 号

©2002 by O'Reilly & Associates, Inc.

Simplified Chinese Edition, jointly published by O'Reilly & Associates, Inc. and China Machine Press, 2003. Authorized translation of the English edition, 2002 O'Reilly & Associates, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly & Associates, Inc. 出版 2002.

简体中文版由机械工业出版社出版 2003。英文原版的翻译得到 O'Reilly & Associates, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly & Associates, Inc. 的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

书 名 / JavaScript 权威指南 (第四版)

书 号 / ISBN 7-111-11091-9

责任编辑 / 朱劼

封面设计 / Edie Freedman, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮政编码 100037)

经 销 / 印刷新华书店北京发行所发行

印 刷 / 北京牛山世兴印刷厂

开 本 / 787 毫米 × 1092 毫米 16 开本 64.5 印张 964 千字

版 次 / 2003 年 1 月第 1 版 2003 年 1 月第 1 次印刷

印 数 / 0001-4 000 册

定 价 / 99.00 元 (册)

(凡购本书, 如有倒页、脱页、缺页, 由本社发行部调换)

O'Reilly & Associates 公司介绍

为了满足读者对网络 and 软件技术知识的迫切需求,世界著名计算机图书出版机构 O'Reilly & Associates 公司授权机械工业出版社,翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly & Associates 公司是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司,同时是联机出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》(被纽约公共图书馆评为二十世纪最重要的 50 本书之一)到 GNN(最早的 Internet 门户和商业网站),再到 WebSite(第一个桌面 PC 的 Web 服务器软件),O'Reilly & Associates 一直处于 Internet 发展的最前沿。

许多书店的反馈表明,O'Reilly & Associates 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比,O'Reilly & Associates 公司具有深厚的计算机专业背景,这使得 O'Reilly & Associates 形成了一个非常不同于其他出版商的出版方针。O'Reilly & Associates 所有的编辑人员以前都是程序员,或者是顶尖级的技术专家。O'Reilly & Associates 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家,而现在编写著作,O'Reilly & Associates 依靠他们及时地推出图书。因为 O'Reilly & Associates 紧密地与计算机业界联系着,所以 O'Reilly & Associates 知道市场上真正需要什么图书。

作者简介

David Flanagan是一位计算机程序设计人员,他将大部分时间用于编写JavaScript程序和Java程序。他在O'Reilly公司出版的书还包括《JavaScript Pocket Reference》、《Java in a Nutshell》、《Java Foundation Classes in a Nutshell》和《Java Examples in a Nutshell》。David拥有麻省理工学院的计算机科学和工程学位。他和妻子、儿子住在美国太平洋西北部的Seattle、Washington、Vancouver、British Columbia这些城市中。

封面介绍

本书封面上的动物是爪哇犀牛(Javan rhinoceros)。犀牛共有五种,都以其庞大的体形、粗厚如装甲的皮肤、三趾蹄以及一个或两个犀牛角著称。其中爪哇犀牛和苏门达腊犀牛栖息在森林中。爪哇犀牛与印度犀牛外表很像,但是体格要小一些、皮肤纹理有显著不同。

在人们印象中,犀牛总是站着,把犀牛角伸入水或泥中。它们经常如此。不在河里呆着时,犀牛会到泥里挖一个深深的坑,在里头打滚。它们的栖息地都有两个优点。首先是可以减轻热带的酷热,并防止吸血蚊蝇的叮咬(打滚留下的泥也可以防蚊蝇)。其次,泥坑或河水还有助于支撑犀牛庞大的身躯,减轻它们腿部和背部的负担。

长久以来,民间传说犀牛的角富于魔力,而且拥有犀牛角的人也可以获得这种魔力。这是犀牛成为偷猎者主要目标的原因之一。各种犀牛都濒临灭绝,其中爪哇犀牛最为稀少,地球上尚存不到100只。爪哇犀牛曾经遍布整个东南亚,但如今只有在印度尼西亚和越南也许还能找到。

目录

前言	1
第一章 JavaScript 概述	7
1.1 对 JavaScript 的误解	8
1.2 JavaScript 的版本	9
1.3 客户端 JavaScript	10
1.4 其他环境中的 JavaScript	12
1.5 客户端的 JavaScript: 网页中的可执行内容	13
1.6 客户端 JavaScript 的特性	16
1.7 JavaScript 的安全性	20
1.8 例子: 用 JavaScript 计算借贷支出	20
1.9 如何使用本书其余的部分	24
1.10 JavaScript 探秘	26

第一部分 JavaScript 的核心

第二章 词法结构	31
2.1 字符集	31

2.2 大小写敏感性	32
2.3 空白符和换行符	32
2.4 可选的分号	33
2.5 注释	34
2.6 直接量	34
2.7 标识符	35
2.8 保留字	36

第三章 数据类型和值 38

3.1 数字	39
3.2 字符串	43
3.3 布尔值	46
3.4 函数	47
3.5 对象	49
3.6 数组	51
3.7 null	53
3.8 undefined	53
3.9 Date 对象	54
3.10 正则表达式	55
3.11 Error 对象	55
3.12 基本数据类型的包装对象	56

第四章 变量 58

4.1 变量的类型	59
4.2 变量的声明	59
4.3 变量的作用域	60
4.4 基本类型和引用类型	63
4.5 无用存储单元的收集	65
4.6 作为属性的变量	66
4.7 深入理解变量作用域	68

第五章 表达式和运算符 70

5.1 表达式	70
5.2 运算符概述	71
5.3 算术运算符	75
5.4 相等运算符	78
5.5 关系运算符	81
5.6 字符串运算符	84
5.7 逻辑运算符	85
5.8 逐位运算符	87
5.9 赋值运算符	89
5.10 其他运算符	91

第六章 语句 97

6.1 表达式语句	97
6.2 复合语句	98
6.3 if 语句	99
6.4 else if 语句	101
6.5 switch 语句	102
6.6 while 语句	105
6.7 do/while 语句	106
6.8 for 语句	107
6.9 for/in	108
6.10 标签语句	109
6.11 break 语句	110
6.12 continue 语句	111
6.13 var 语句	113
6.14 function 语句	114
6.15 return 语句	115
6.16 throw 语句	116
6.17 try/catch/finally	117

6.18 with 语句	119
6.19 空语句	120
6.20 JavaScript 语句小结	121
第七章 函数	123
7.1 函数的定义和调用	123
7.2 作为数据的函数	128
7.3 函数的作用域: 调用对象	130
7.4 函数的实际参数: Arguments 对象	130
7.5 函数的属性和方法	132
第八章 对象	136
8.1 对象和属性	136
8.2 构造函数	139
8.3 方法	140
8.4 原型对象和继承	143
8.5 面向对象的 JavaScript	147
8.6 作为关联数组的对象	154
8.7 对象的属性和方法	156
第九章 数组	162
9.1 数组和数组元素	162
9.2 数组的方法	167
第十章 使用正则表达式的模式匹配	173
10.1 正则表达式的定义	173
10.2 用于模式匹配的 String 方法	183
10.3 RegExp 对象	186

第十一章 JavaScript 的更多主题 189

11.1 数据类型转换	189
11.2 使用值和使用引用	195
11.3 无用存储单元收集	201
11.4 词法作用域和嵌套函数	203
11.5 Function()构造函数和函数直接量	205
11.6 Netscape 公司的 JavaScript 1.2 的不兼容性	206

第二部分 客户端 JavaScript

第十二章 Web 浏览器中的 JavaScript 211

12.1 Web 浏览器环境	211
12.2 在 HTML 中嵌入 JavaScript	216
12.3 JavaScript 程序的执行	224

第十三章 窗口和框架 231

13.1 Window 对象概述	231
13.2 简单的对话框	234
13.3 状态栏	236
13.4 超时设定和时间间隔	237
13.5 错误处理	239
13.6 Navigator 对象	240
13.7 Screen 对象	242
13.8 Window 对象的控制方法	243
13.9 Location 对象	247
13.10 History 对象	249
13.11 多窗口和多框架	252

第十四章 Document 对象	259
14.1 Document 对象概览	260
14.2 动态生成的文档	264
14.3 Document 对象的颜色属性	270
14.4 Document 对象的信息属性	270
14.5 表单	271
14.6 图像	271
14.7 链接	279
14.8 锚	281
14.9 小程序	283
14.10 嵌入式数据	284
 第十五章 表单和表单元素	 285
15.1 Form 对象	286
15.2 定义表单元素	288
15.3 脚本化表单元素	292
15.4 表单验证示例	301
 第十六章 脚本化 cookie	 304
16.1 cookie 概览	304
16.2 cookie 的储存	306
16.3 cookie 的读取	308
16.4 cookie 示例	309
 第十七章 文档对象模型	 313
17.1 DOM 概览	314
17.2 使用 DOM 的核心 API	325
17.3 DOM 与 Internet Explorer 4 的兼容性	343
17.4 DOM 与 Netscape 4 的兼容性	346
17.5 简便方法: Traversal 和 Range API	347

第十八章 级联样式表和动态 HTML	355
18.1 CSS 的样式和样式表	356
18.2 用 CSS 进行元素定位	365
18.3 脚本样式	375
18.4 第四代浏览器中的 DHTML	384
18.5 关于样式和样式表的其他 DOM API	389
 第十九章 事件和事件处理	 395
19.1 基本事件处理	396
19.2 2 级 DOM 中的高级事件处理	406
19.3 Internet Explorer 事件模型	423
19.4 Netscape 4 事件模型	429
 第二十章 兼容性	 434
20.1 平台和浏览器的兼容性	435
20.2 语言版本的兼容性	440
20.3 非 JavaScript 浏览器的兼容性	444
 第二十一章 JavaScript 的安全性	 447
21.1 JavaScript 与安全性	447
21.2 受限制的特性	449
21.3 同源策略	450
21.4 安全区和签名脚本	451
 第二十二章 在 JavaScript 中使用 Java	 453
22.1 脚本化 Java 小程序	454
22.2 在 Java 中使用 JavaScript	456
22.3 直接使用 Java 类	460
22.4 LiveConnect 数据类型	462

22.5 LiveConnect 数据转换	468
22.6 JavaObject 对象在 JavaScript 中的转换	472
22.7 从 Java 到 JavaScript 的数据转换	474

第三部分 JavaScript 核心参考手册

JavaScript 核心参考手册	479
-------------------------	-----

第四部分 客户端 JavaScript 参考手册

客户端 JavaScript 参考手册	613
---------------------------	-----

第五部分 W3C DOM 参考手册

W3C DOM 参考手册	787
--------------------	-----

第六部分 类、属性、方法和事件处理程序索引

类、属性、方法和事件处理程序索引	993
------------------------	-----

词汇表	1013
-----------	------

前言

自从本书的第三版出版以来，用JavaScript进行Web程序设计的世界已经发生了巨大的改变，其中包括：

- ECMA-262标准的第二版和第三版已经发布，其中更新了JavaScript语言的核心。使Netscape公司的JavaScript解释器和Microsoft公司的JScript解释器相一致的版本也发布了。
- Netscape公司的JavaScript解释器（一个版本是用C语言编写的，另一个版本是用Java语言编写的）的源代码作为开放资源发布了，任何想把脚本语言嵌入自己应用程序的人都可以使用它。
- 万维网联盟（W3C）发布了文档对象模型（DOM）标准的两个版本（或两级）。最近的浏览器都支持这一标准（支持的程度不同），而且允许客户端的JavaScript脚本与文档内容进行交互，从而生成复杂的动态HTML（DHTML）效果。对其他W3C标准（如HTML 4、CSS1和CSS2）的支持也已经相当普遍。
- Mozilla组织利用Netscape公司提供的源代码制作了良好的第五代浏览器。在编写本书期间，Mozilla浏览器还不过处于1.0版本的水平，但是该浏览器已经足够成熟，因此Netscape公司采用Mozilla代码作为它的6.0和6.1版本的浏览器的基础。

- Microsoft 公司的 Internet Explorer 在桌面系统上已经成为占统治地位的浏览器。但是 Netscape/Mozilla 的浏览器仍然与 Web 开发者保持着密切的关系，这主要是由于它对 Web 标准提供了较高支持。除此之外，少数浏览器，如 Opera (<http://www.opera.com>) 和 Konqueror (<http://www.konqueror.org>)，也和 Web 开发者有密切的关系。
- Web 浏览器（和 JavaScript 解释器）不再局限于桌面电脑，它已经被移植到 PDA，甚至移动电话上了。

总之，JavaScript 语言的核心已经发展成熟。它已经被标准化了，而且使用范围也比以往广泛得多。Netscape 公司市场份额的暴跌促进了桌面型电脑上的 Web 浏览器的发展，而且启用 JavaScript 的 Web 浏览器在非桌面型电脑的平台上也可以使用了。虽然 Web 标准化还没有完成，但在这方面有了明显的进展。近来，浏览器中 DOM 标准的实现（部分的）给予了 Web 开发者长期期待的独立于开发商的 API，他们可以用这些 API 进行编码。

第四版中增加的新内容

《JavaScript 权威指南》第四版基于上述介绍的变化对内容进行了全面的更新。主要的新特性包括 JavaScript 1.5 和它基于的 ECMA-262 标准的第三版的完整介绍，以及 2 级 DOM 标准的完整说明。

本书的重点已经从介绍特定的 JavaScript 语言和浏览器的实现（JavaScript 1.2、Netscape 4、Internet Explorer 5，等等）转移到记述这些实现基于的（或应该基于的）标准。由于实现的激增，任何一本书想要记述每个实现版本的特性、使用范围、缺陷和错误都是不实际的。重点介绍规范而不是实现会使本书更容易使用，如果你采用同样的方法，它还可以使你的 JavaScript 代码更具可移植性且更容易维护。你还应特别注意对有关 JavaScript 语言核心和 DOM 的新材料中对标准的强调。

这一版本中另一个重要的改变是参考手册部分被明显地分成了三个部分。第一，有关 JavaScript 语言核心的材料从客户端 JavaScript 的材料（第四部分）中分离了出来，自己单独组成一部分（第三部分）。这种分离是为了方便那些在 Web 浏览器之外的环境中使用 JavaScript 语言的程序设计者和对客户端 JavaScript 不感兴趣的人。

第二, 记述W3C的DOM标准的材料从已有的客户端JavaScript材料中分离了出来, 组成第五部分。DOM标准定义的API和传统的客户端JavaScript遗留下来的API有很大不同。开发者通常会根据他们作为目标的浏览器平台选择一种API, 而且一般不会在两种API之间来回切换。保持这两种API相互独立还保留了现有客户端参考手册的架构, 这大大方便了读过本书第三版、现在升级到第四版的读者。

为了容纳所有新素材, 而又不使本书变得非常厚, 书中删除了对象的一些琐碎属性的参考页。这些属性在对象的参考页中已经说明过了, 再用它自己的参考页进行额外的说明既显得十分多余。不过需要大量说明的属性以及所有方法仍然具有它们自己的参考页。另外, O'Reilly公司的设计奇才为本书创建了一种新的结构, 不仅使它仍然保持容易阅读, 而且占用的版面更少了。

排版约定

本书使用下列排版约定:

粗体 (Bold)

用来引用计算机键盘上的特殊键或引用用户界面上的某个部分, 如按钮 **Back** 和菜单 **Options**。

斜体 (*italic*)

用于强调重点, 或者表示术语的第一次使用。此外, 它还用于电子邮件地址、网址、FTP地址、文件名、目录名和新闻组等。而且, 本书还将斜体字用于Java类的名字, 以与JavaScript类的名字区分开来。

等宽字体 (Constant width)

用于所有的JavaScript代码、HTML文本列表以及在程序设计时要输入的内容。

等宽斜体 (*Constant width italic*)

用于函数的参数名以及程序中的占位符(说明应该用一个实际的值替换这个项目)。

勘误表

请把你在本书中发现的错误、不准确处、缺陷、易误解的或混乱的语句和旧版式报告给 O'Reilly & Associates 公司，以帮助我们改进本书以后的版本。O'Reilly 公司为本书建立了一个网站，它包括一个所有已知的错误的列表。从本书的目录页可以链接到这个勘误表：

<http://www.oreilly.com/catalog/jsript4/>

这个勘误表页含有一个表单的链接，你可以通过该表单汇报找到的错误。你也可以通过电子邮件报告错误或询问有关本书的问题：

bookquestions@oreilly.com

info@mail.com.cn

如何找到在线的例子

本书中列出的例子都可以从本书的站点下载。从本书的目录页能够链接到示例：

<http://www.oreilly.com/catalog/jsript4/>

建议与评论

本书的内容都经过测试，尽管我们做了最大的努力，但错误和疏忽仍然是在所难免的。如果你发现有什么错误，或者是对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly & Associates, Inc.

101 Morris Street

Sebastopol, CA 95472

中国:

100080 北京市海淀区知春路 49 号希格玛公寓 B 座 809 室
奥莱理软件（北京）有限公司

询问技术问题或对本书的评论，请发电子邮件到:

info@mail.oreilly.com.cn

最后，您可以在 WWW 上找到我们:

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

致谢

Mozilla 公司的 Brendan Eich 是 JavaScript 的创作者和主要革新者。感谢 Brendan 开发了 JavaScript，他在百忙中花大量时间回答我们的问题甚至不断催促我们写作，对此我和许多 JavaScript 开发者感激不已。除了耐心地回答我们提出的大量问题之外，Brendan 还通读了本书的第一版和第三版，并且提出了很多有用的建议。

除了 Brendan 之外，还有其他几位顶尖的技术评论者阅读了这本书，他们的建议使本书内容更加完善、更加准确。Netscape 公司的 Waldemar Horwat 审阅了第四版中有关 JavaScript 1.5 的新资料，W3C 的 Philippe Le Hegaret、荷兰 Internet 顾问和创建公司 Netling Framfab (<http://www.netlingframfab.nl>) 的客户端程序设计部的主管 Peter-Paul Koch、SitePen 公司 (<http://www.sitepen.com>) 的 Dylan Schiemann 和独立 Web 开发者 Jeff Yates 审阅了有关 W3C DOM 的新资料。这些审阅者中有两位维护了有关用 DOM 进行 Web 设计的站点。Peter-Paul 的站点是 <http://www.xs4all.nl/~ppk/js/>。Jeff 的站点是 <http://www.pbwizard.com>。尽管 IBM Research 的 Joseph Kesselman 没有审阅过本书，但是他回答了我们提出的有关 W3C DOM 的问题，给予了大量的帮助。

本书的第三版是由 Netscape 公司的 Brendan Eich、Waldemar Horwat 和 Vidur Apparao，Microsoft 公司的 Herman Venter 以及两位独立的 JavaScript 开发者 Jay

Hodges 和 Angelo Sirigos 审阅的。CNET's Builder.COM 公司的 Dan Shafer 为本书的第三版本作了一些准备工作。虽然这一版中并没有使用他提供的资料，但是他的想法和整体大纲却给了我们极大的帮助。Netscape 公司的 Norris Boyd 和 Scott Furman 也为这一版本提供了有用的信息，还有 Netscape 公司的 Vidur Apparao 和 Microsoft 公司的 Scott Issacs，他们花费了大量的时间与我讨论即将出台的 DOM 标准。最后，Tankred Hirschmann 博士提供了有关 JavaScript 1.2 的复杂性的过人见解。

本书的第二版大大受益于 Netscape 公司的 Nick Thompson 和 Richard Yaker 与 Microsoft 公司的 Shon Katzenberger 博士、Larry Sullivan 和 Dave C. Mitchell 以及 R&B Communication 公司的 Lynn Rollin 的帮助与建议。第一版则由 Bay Networks 公司的 Neil Berkman 与 O'Reilly & Associates 公司的 Andrew Schulman 和 Terry Allen 审阅。

本书各个版本的编辑为提高本书品质做了大量工作。Paula Ferguson 是第四版和第三版的编辑，她对本书进行了全面且必要的润色，使得它更易读易懂。第二版是由 Frank Willison 编辑的，Andrew Schulman 编辑了第一版。

最后，出于多种原因，要向 Christie 致谢。

David Flanagan

2001 年 9 月

第一章

JavaScript 概述

JavaScript 是一种轻型的、解释型的程序设计语言，而且具有面向对象的能力。该语言的通用核心已经嵌入了 Netscape、Internet Explorer 和其他的 Web 浏览器中，而且它能用表示 Web 浏览器窗口及其内容的对象使 Web 程序设计增色不少。JavaScript 的客户端版本把可执行的内容添加到了网页中，这样一来，网页就不再是静态的 HTML 了，而是包含与用户进行交互的程序、控制浏览器的程序以及动态创建 HTML 内容的程序。

在句法构成上，JavaScript 的核心语言与 C、C++ 和 Java 相似，都具有诸如 if 语句、while 循环和 && 运算符这样的结构。但是，JavaScript 与这些语言的相似之处也仅限于句法上的类同。JavaScript 是一种无类型语言，这就是说，它的变量不必具有一个明确的类型。而且，与其说 JavaScript 的对象和 C 中的结构或 C++ 和 Java 中的对象相似，不如说它更像 Perl 语言中的关联数组。另外，JavaScript 面向对象的继承机制与 Self 和 NewtonScript 相似（这两种语言都不太为人知），它们的继承机制与 C++ 和 Java 中的继承机制大相径庭。JavaScript 还有一点与 Perl 语言类似，那就是它们都是解释型的语言。除此之外，JavaScript 还有多处灵感都来源于 Perl 语言，诸如正则表达式和以数组处理的特性。

本章是对 JavaScript 的一个概览，它解释了 JavaScript 能够做什么和不能够做什么，并且澄清了对这种语言的一些误解。它还区别了 JavaScript 的核心语言的嵌入版本和扩展版本，例如客户端的 JavaScript 是嵌入网页中的，而服务器端的 JavaScript

则是嵌入 Netscape 网络服务器中的（本书介绍了 JavaScript 的核心与客户端的 JavaScript）。另外，本章还用客户端 JavaScript 程序的一些例子说明了真正的网络程序设计是怎样的。

1.1 对 JavaScript 的误解

JavaScript 是一个相当容易误解和混淆的主题。在对它进行进一步的研究之前，有必要澄清一些长期存在的有关该语言的误解。

1.1.1 JavaScript 并非 Java

对 JavaScript 最常见的误解是认为它是 Sun Microsystems 公司的程序设计语言 Java 的简化版本。但是除了句法上有一些相似之处以及都能够提供网页中的可执行内容之外，JavaScript 和 Java 是完全不相干的。相似的名称纯粹是一种营销策略罢了（该语言最初被称为 LiveScript，只是到最后才被改为 JavaScript）。

但是 JavaScript 和 Java 这两种语言却是很好的搭档。它们的特性集合是不相同的。JavaScript 可以控制浏览器的行为和内容，但是却不能绘图和执行连网。而 Java 虽然不能在总体上控制浏览器，但是却可以进行绘图，执行连网和多线程。客户端的 JavaScript 可以与嵌入网页的 Java applet 进行交互，并且能够对它进行控制，从这一意义上说，JavaScript 真的可以脚本化 Java（详见第二十二章）。

1.1.2 JavaScript 并不简单

JavaScript 是作为一种脚本语言而不是作为一种程序设计语言来推广的，其中的差别在于脚本语言比较简单，它们是给非程序员提供的程序设计语言。实际上，JavaScript 最初出现的时候相当简单，其复杂程度大概与 BASIC 相同。它确实有许多特性可以使它更加灵活、让程序设计新手更加容易使用。非程序设计者可以使用 JavaScript 来执行有限的、按部就班的程序设计任务。

但是，在简单的外表之下，JavaScript 却是一种具有丰富特性的程序设计语言，它和其他所有语言一样复杂，甚至比某些语言还复杂得多。如果一个程序者对 JavaScript 没有扎实的理解，那么当他要用 JavaScript 执行较复杂的任务时，就会发

现整个进程困难重重。因此，本书对 JavaScript 进行了完整的介绍，以便你能全面地理解它。

1.2 JavaScript 的版本

JavaScript 语言已经发展几年了，Netscape 公司发布了该语言的一个版本，Microsoft 公司也发布了 JavaScript 语言的相似版本，名为 JScript。另外，ECMA (<http://www.ecma.ch>) 发布了三个版本的 ECMA-262 标准，该标准标准化了 JavaScript 语言，但采用的却是个蹩脚的名字 ECMAScript。

表 1-1 列出了这些版本，并且解释了它们的关键特性和各个版本之间的关联方法。本书用 JavaScript 来引用这些版本中的一个，包括 Microsoft 公司的 JScript。当特别地引用 ECMAScript 时，则使用术语 ECMA-262 或 ECMA。

表 1-1: JavaScript 的版本

版本	说明
JavaScript 1.0	该语言的原始版本，目前基本上已经被废弃。由 Netscape 2 实现
JavaScript 1.1	引入了真正的 Array 对象，消除了大量重要的错误。由 Netscape 3 实现
JavaScript 1.2	引入了 switch 语句、正则表达式和大量其他特性，基本上符合 ECMAv1，但是还有一些不兼容性。由 Netscape 4 实现
JavaScript 1.3	修正了 JavaScript 1.2 的不兼容性，符合 ECMA v1。由 Netscape 4.5 实现
JavaScript 1.4	只在 Netscape 的服务器产品中实现
JavaScript 1.5	引入了异常处理，符合 ECMA v3。由 Mozilla 和 Netscape 6 实现
JScript 1.0	基本上相当于 JavaScript 1.0，由 IE 3 的早期版本实现
JScript 2.0	基本上相当于 JavaScript 1.1，由 IE 3 的后期版本实现
JScript 3.0	基本上相当于 JavaScript 1.3，符合 ECMA v1。由 IE 4 实现
JScript 4.0	还没有任何 Web 浏览器能实现它
JScript 5.0	支持异常处理。部分符合 ECMA v3。由 IE 5 实现

表 1-1: JavaScript 的版本 (续)

版本	说明
JScript 5.5	基本上相当于 JavaScript 1.5 完全符合 ECMA v3。由 IE 5.5 和 IE 6 实现 (IE 6 实际实现的是 JScript 5.6, 但是 JScript 5.6 和客户端 JavaScript 程序设计者相关的部分与 5.5 没有区别)
ECMA v1	该语言的第一个标准版本, 标准化了 JavaScript 1.1 的基本特性, 并添加了一些新特性。没有标准化 switch 语句和正则表达式。与 JavaScript 1.3 和 JScript 3.0 的实现一致
ECMA v2	该标准的维护版本, 添加了说明, 但没有定义任何新特性
ECMA v3	标准化了 switch 语句, 正则表达式和异常处理。与 JavaScript 1.5 和 JScript 5.5 的实现一致

1.3 客户端 JavaScript

当把一个 JavaScript 解释器嵌入 Web 浏览器时, 就形成了客户端 JavaScript。这是迄今为止最普通的 JavaScript 变体。当人们提到 JavaScript 时, 通常所指的是客户端 JavaScript。本书介绍了客户端 JavaScript 及 JavaScript 语言的核心, 这两者是混合在一起的。

我们将在本章后面的小节中详细讨论客户端的 JavaScript 以及它的功能。简而言之, 客户端 JavaScript 将 JavaScript 解释器的脚本化能力与 Web 浏览器定义的文档对象模型 (Document Object Model, DOM) 结合在一起。因为这两种技术是以一种相互作用的方式结合在一起的, 所以产生的结果大于两部分能力之和, 即客户端 JavaScript 使得可执行的内容散布在网络中的各个地方, 它是新一代动态 DHTML (DHTML) 文档的核心。

与 ECMA-262 规范定义了 JavaScript 语言核心的标准版本一样, W3C 也发布了一个 DOM 规范 (或建议), 用来将浏览器必须在它的 DOM 中支持的特性进行标准化。我们将在第十七、十八和十九章中了解到有关该标准的更多内容。尽管 W3C DOM 标准还没有得到应有的支持, 但是对它的支持仍然足以使 Web 开发者开始编写基于它的 JavaScript 代码。

表 1-2 展示了核心语言的版本和 Netscape 公司与 Microsoft 公司提供的各种浏览器版本所支持的 DOM 功能。注意表中列出的 Internet Explorer 版本引用的是哪种浏览器的 Windows 版本。IE 的 Macintosh 版本的功能通常与相同版本号的 Windows 版本不同。另外要记住, IE 允许 JScript 解释器独立于浏览器升级, 因此遇到所安装的 IE 支持的 JScript 版本比下表列出的要高是完全可能的。

表 1-2: 各种浏览器支持的客户端 JavaScript 特性

浏览器版本	语言版本	DOM 功能
Netscape 2	JavaScript 1.0	表单操作
Netscape 3	JavaScript 1.1	图像翻转
Netscape 4	JavaScript 1.2	具有层的 DHTML
Netscape 4.5	JavaScript 1.3	具有层的 DHTML
Netscape 6/Mozilla	JavaScript 1.5	对 W3C DOM 标准的大量支持, 废止了对层的支持
IE 3	JScript 1.0/2.0	表单操作
IE 4	JScript 3.0	图像翻转, 具有 document.all[] 性质的 DHTML
IE 5	JScript 5.0	具有 document.all[] 性质的 DHTML
IE 5.5	JScript 5.5	部分支持 W3C DOM 标准
IE 6	JScript 5.5	部分支持 W3C DOM 标准, 缺乏对 W3C DOM 标准的事件模型的支持

Netscape 公司和 Microsoft 公司都提供了客户端 JavaScript, 这两种版本的不同之处与不兼容性比两家公司相应的语言核心之间的差别要大得多。不过, 这两种浏览器都支持的客户端 JavaScript 的特性子集还是比较大的。由于没有较好的名字, 所以引用客户端 JavaScript 版本时使用的都是它们引以为基础的语言核心版本。因此, 在许多环境中, 术语 JavaScript 1.2 指的就是 Netscape 4 和 Internet Explorer 4 支持的客户端 JavaScript。当使用核心语言的版本号来引用客户端的 JavaScript 版本时, 指的是 Netscape 和 Internet Explorer 都支持的特性集合。当讨论一种浏览器特有的客户端特性时, 使用浏览器的名称和版本号来引用它。

注意，支持客户端 JavaScript 的浏览器不是只有 Netscape 和 Internet Explorer。例如，Opera (<http://www.opera.com>) 也支持客户端 JavaScript。但是，由于 Netscape 和 Internet Explorer 占据了大部分的市场份额，所以本书只对它们进行了讨论。其他浏览器实现的客户端 JavaScript 应该遵循这两种浏览器的实现。

同样地，能够嵌入 Web 浏览器的程序设计语言也不是仅有 JavaScript 这一种。例如，Internet Explorer 就支持 VBScript 这种语言。它是 Microsoft 公司的 Visual Basic 语言的变体，虽然也提供了许多与 JavaScript 相同的特性，但是却只能在 Microsoft 公司的浏览器中使用。HTML 4.0 规范在讨论 HTML 标记 `<script>` 时采用了 Tcl 程序设计语言作为嵌入式脚本语言的一个例子。尽管没有什么主流浏览器为此而支持 Tcl 语言，但是，一种浏览器要支持这种语言也是很容易的。

本书以前的版本对 Netscape 浏览器的介绍比对 Microsoft 浏览器的介绍全面得多。存在这一倾向的主要原因在于 JavaScript 是 Netscape 公司发明的，而且它（至少一段时间内）在 Web 浏览器市场中占有统治地位。不过这种对 Netscape 的倾向在本书的每一个后续版本中都有所减小，当前的这个版本重点介绍标准（如 ECMAScript 和 W3C DOM），而不在于特定的浏览器。无论如何，读者仍能从旧版本沿袭下来的材料中发现这种对 Netscape 的倾向。

1.4 其他环境中的 JavaScript

JavaScript 是一种常规用途的程序设计语言，它的使用并不仅仅限于 Web 浏览器。JavaScript 能够嵌入任何应用程序，用来为程序提供脚本的功能。事实上，从一开始，Netscape 公司的 Web 服务器就含有 JavaScript 解释器，以便能够用 JavaScript 编写服务器端的脚本。同样的，Microsoft 公司除了在 Internet Explorer 中使用了 JScript 解释器外，还在它的 IIS Web 服务器和 Windows Scripting Host 产品中使用了该解释器。

Netscape 公司和 Microsoft 公司都向那些想把 JavaScript 解释器嵌入自己应用程序的公司和程序设计者开放了它们的 JavaScript 解释器。Netscape 公司的解释器是作为开放资源发布的，目前通过 Mozilla 组织可以得到它（详情请登录 <http://www.mozilla.org/js/>），Mozilla 组织实际上提供了两种不同版本的 JavaScript 1.5 解释器。

一个版本是用C语言编写的，称为SpiderMonkey。另一个版本是用Java编写的，本书称之为Rhino。

我们希望看到越来越多的应用程序将JavaScript作为嵌入式脚本语言（注1）。如果要为这样的应用程序编写脚本，那么你就会发现本书的前半部分非常有用，它介绍了JavaScript语言的核心。但是那些介绍特定Web浏览器的章节可能就不适用于你的脚本了。

1.5 客户端的 JavaScript：网页中的可执行内容

当一个Web浏览器嵌入了JavaScript解释器时，它就允许可执行的内容以JavaScript脚本的形式分布到Internet中。例1-1展示了一个简单的嵌入网页中的JavaScript程序，或者说脚本。

例 1-1：一个简单的 JavaScript 程序

```
<html>
<body>
<head><title>Factorials</title></head>
<script language="JavaScript">
document.write( <n2>Table of Factorials</n2>');
for(i = 1, fact = 1; i <= 10; i++, fact *= i) {
    document.write(i + "! = " + fact);
    document.write("<br/>");
}
</script>
</body>
</html>
```

把这个脚本装载进一个启用JavaScript的浏览器中后，就会产生如图1-1所示的输出。

注1： ActionScript是Macromedia公司的Flash5中使用的一种脚本语言，它是在ECMAScript标准发布后被模型化的，但它不是真正的JavaScript。

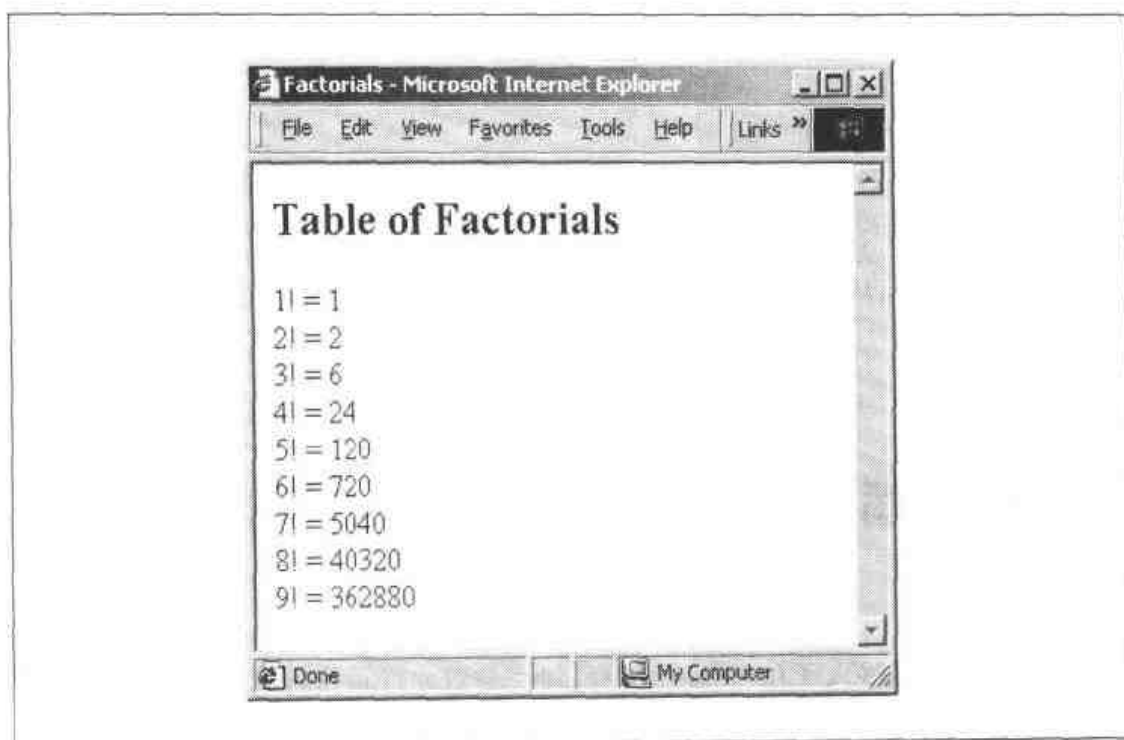


图 1-1: 用 JavaScript 生成的一个网页

在这个例子中可以看到, 标记 `<script>` 和 `</script>` 是用来在 HTML 文件中嵌入 JavaScript 代码的。我们将在第十二章中了解到更多有关 `<script>` 标记的内容。这一例子要说明的 JavaScript 的主要特性是方法 `document.write()` 的使用 (注 2), 这一方法用于动态输出由 Web 浏览器解析并显示的 HTML 文本。在本书中我们会多次见到它。

除了能够控制网页中的内容之外, JavaScript 还能够控制浏览器和出现在浏览器中的 HTML 表单的内容。我们将在本章后面的小节中详细的学习 JavaScript 的这些功能, 而且本书后面的章节中还有更加详细的介绍。

JavaScript 不仅能够控制 HTML 文档的内容, 而且能够控制这些文档的行为。也就是说, 当你在输入域中输入了一个值或者点击了文档中的一幅图像时, JavaScript 程序就会以某种方式对此做出响应。它是通过给文档定义“事件处理器” (event handler) 实现这一点的。所谓事件处理器, 就是在特定的事件 (如用户点击了一个

注 2: 方法是函数或过程使用的面向对象的术语, 在整本书中都会看到它。

按钮)发生时执行的JavaScript代码段。例1-2展示了一个非常简单的HTML表单的定义,这个表单含有一个响应按钮点击事件的事件处理器。

例1-2: 一个定义了JavaScript事件处理器的HTML表单

```
<form>
<input type='button'
      value='Click here'
      onclick="alert('You clicked the button');">
</form>
```

图1-2说明了点击按钮之后的结果。

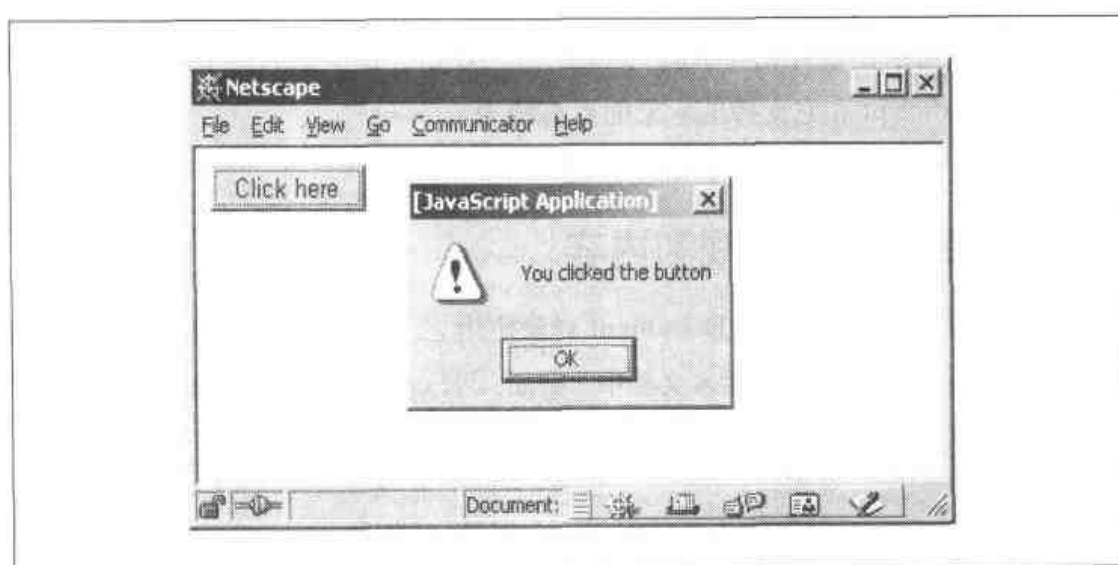


图1-2: JavaScript 对一个事件的响应

例1-2中使用的性质 onclick 原本是 Netscape 公司对 HTML 的一个扩展,是专门为客户端 JavaScript 而添加的。但是,现在 HTML 的 4.0 版本已经将这一性质和其他的事件处理器性质标准化了。所有的 JavaScript 事件处理器都是像该例一样用 HTML 性质定义的。onclick 性质的值是一个 JavaScript 代码串,它是在用户点击了按钮的时候执行的。在这种情况下,onclick 事件处理器将调用 alert() 函数。在图 1-2 中可以看到,alert() 函数会弹出一个对话框以显示指定的消息。

例1-1和例1-2只不过使用了客户端JavaScript最简单的特性。JavaScript在客户端真正的强大之处在于脚本能够访问基于网页内容的对象层次。例如,客户端JavaScript程序可以访问并操作文档中出现的每一幅图像,可以与嵌入在HTML文档中的各个Java小程序和其他对象通信与交互。一旦掌握了JavaScript语言的核心,

那么在网页中有效地使用JavaScript的关键就是学会使用浏览器显示出来的文档对象模型（DOM）的特性。

1.6 客户端 JavaScript 的特性

JavaScript 另一种可能的用途是编写执行任意计算的程序。例如，可以用它编写计算斐波纳契数列或检索素数的简单脚本。不过，在 Web 和 Web 浏览器环境中，该语言还有更加有趣的应用，那就是基于用户在 HTML 表单中提供的信息在线地计算销售税。前面已经提到过，JavaScript 的真正强大之处在于它支持的基于浏览器和文档的对象。为了让你对 JavaScript 的潜力有所了解，接下来的几节列出并解释了客户端的 JavaScript 和它支持的对象的重要能力。

1.6.1 控制文档的外观和内容

我们已经知道，JavaScript 的 Document 对象使用 write() 方法，可以在浏览器解析文档时把任何 HTML 文本写入文档中。例如，可以把当前的日期与时间添加到一个文档中，或者在不同的平台上显示不同的内容。

也可以使用 Document 对象从头开始生成一个完整文档。Document 对象的属性允许你指定文档的背景颜色、文本颜色以及文档中的超文本链接颜色。其实这就是生成动态的、有条件的 HTML 文档的能力，这种技术在多框架的文档中更加适用。实际上，在某些情况下，JavaScript 程序还可以利用动态生成框架内容这一技术完全地替换一个传统的服务器端脚本。

Internet Explorer 4 和 Netscape 4 都支持生成动态 HTML 效果的专利技术，这种效果使得文档内容能够动态生成，动态移动和动态的改变。IE 4 还支持完整的文档对象模型，这使 JavaScript 能够访问文档中的每一个 HTML 元素。另外，IE 5.5 和 Netscape 6 还支持 W3C DOM 标准（或至少支持它的关键部分），该标准定义了一种标准的、可移植的访问 HTML 文档中所有元素和文本的方法，还定义了通过操作级联样式表（CSS）的样式性质来定位元素以及修改它们外观的方法。在这些浏览器中，客户端 JavaScript 对文档内容具有绝对的控制权，这种控制可以进行无穷的本体化。

1.6.2 对浏览器的控制

有些JavaScript对象允许对浏览器的行为进行控制。Window对象支持弹出对话框以向用户显示简单消息的方法，还支持从用户那里获取简单输入信息的方法。此外，有些对象还定义了创建并打开（和关闭）全新的浏览器窗口的方法，新建的窗口的大小不限以及具有任意的用户控件组合。例如，它使你可以打开多个窗口，以便用户能够看到网站的多个视图。新的浏览器窗口对临时显示生成的HTML文档非常有用，而且，如果在创建窗口时，它没有任何菜单栏和用户控件，那么还可以将它作为对话框使用，以便显示更加复杂的消息或用户输入。

JavaScript没有定义可以在浏览器窗口中直接创建并操作框架的方法。但是，它能够动态生成HTML的能力却可以让你使用HTML的标记创建任何想要的框架布局。

JavaScript还可以控制在浏览器中显示哪个网页。Location对象可以在浏览器的任何一个框架或窗口中装载并显示出任意的URL所指的文档。History对象则可以在用户的浏览历史中前后移动，模拟浏览器的 **Forward** 按钮和 **Back** 按钮的动作。

另外，Window对象还有一个方法使你可以在任意一个浏览器窗口的状态线中向用户显示消息。

1.6.3 与 HTML 表单的交互

客户端JavaScript的另一个重要之处就是它能够与HTML表单进行交互。这种能力是由Form对象以及它含有的表单元素对象（即Button对象、Checkbox对象、Hidden对象、Password对象、Radio对象、Reset对象、Select对象、Submit对象、Text对象和Textarea对象）提供的。正是这些元素对象使得你能够对文档中某个表单的输入元素的值进行读写操作。例如，可以使用表单表示一张在线的目录，这样用户可以输入自己的订单，并且能够用JavaScript从表单中读取用户的输入，用来计算订单的总额、销售税以及运输费等。实际上，像这样的JavaScript程序在网络中是很常见的。稍后我们将会看到一个使用HTML表单和JavaScript来计算每月的住房抵押和其他借贷支出的程序。对于这样的应用程序，JavaScript与基于服务器的脚本相比有一个明显的优势，那就是JavaScript代码是在客户端执行的，所以不必把表单的内容发送给服务器，再让服务器执行较为简单的计算了。

客户端JavaScript代码在表单中的另一种常见用法是在提交表单之前对表单数据进行验证。如果客户端的JavaScript代码能够对用户的输入进行所有必要的错误检查,那么服务器端就没有必要再大费周折地检测并通知用户琐碎的输入错误了。此外,客户端JavaScript代码还能够对输入的数据进行预处理,这就大大减少了要发送给服务器的数据量。在某些情况下,客户端JavaScript甚至可以消除对服务器上的脚本的需要。(不过,JavaScript和服务器端的脚本可以很好地协同工作。例如,一个服务器端程序可以从无到有地动态创建一段JavaScript代码,就像它动态地创建HTML文件一样。)

1.6.4 与用户的交互

JavaScript的一个重要特性就是能够定义事件处理器,即在特定的事件发生时要执行的代码段。这些事件通常都是用户触发的,例如把鼠标移到一个超文本链接,在表单中输入了一个值或者点击了表单中的Submit按钮。这种处理事件的能力是至关重要的,因为具有图形界面(如HTML表单)的程序设计内在地要求一种事件驱动的模式。JavaScript可以触发任意一种类型的动作来响应用户事件。最为典型的例子莫过于当用户把鼠标移到一个超文本链接时,在状态栏中显示一条特定的消息、或者当用户提交了一个重要的表单时,弹出一个确认对话框。

1.6.5 用 cookie 读写客户的状态

cookie是客户永久性存储或暂时存储的少量状态数据。cookie将随网页被服务器发送给客户,客户在本地将它们存储起来。此后当客户请求同一个网页或与之相关的网页时,它可以把相关的cookie传回服务器,服务器能够利用这些cookie的值来改变发送回客户的内容。cookie使得网页或者网站能够“记住”有关客户的一些信息,例如,用户以前访问过该站点,或者用户已经在此注册过,已经获得了口令,甚至用户已经表明了对网页的颜色与布局的偏爱,等等。cookie提供的状态信息正是Web的无状态的HTTP协议所遗漏的。

当初发明cookie时,它是由服务器端脚本专用的,虽然它们被存储在客户端,但是却只有服务器能够对它们进行读写操作。JavaScript改变了上述的规则,因为JavaScript程序能够读写cookie的值,还可以根据cookie的值动态地生成文档内容。

1.6.6 更多的特性

除了已经提到过的特性之外，JavaScript 还有许多其他的功能：

- JavaScript 可以改变标记 `` 显示的图像，从而产生图像翻转和动画的效果。
- JavaScript 可以与 Java 小程序或其他出现在浏览器中的嵌入对象进行交互。JavaScript 代码既可以读写这些小程序或对象的属性，又可以调用它们定义的方法。这一特性真正使得 JavaScript 可以将 Java 脚本化了。
- 在本节开始的时候我提到过，JavaScript 可以执行任何计算。它具有浮点数据类型、操作这种类型的算术运算符以及所有的标准浮点运算函数。
- JavaScript 的 `Date` 对象简化了计算和使用日期与时间的方法。
- `Document` 对象支持的一个属性声明了当前文档的最后修改日期。你可以使用这一属性自动地在文档中显示一个时间戳。
- JavaScript 的 `window.setTimeout()` 方法可以确定一个 JavaScript 代码块在多少毫秒之后执行。这对于在 JavaScript 程序中建立延迟或重复的动作是非常有用的。在 JavaScript 1.2 中，`setTimeout()` 中增加了一个名为 `SetInterval()` 的方法。
- `Navigator` 对象（当然是以 Netscape Web 浏览器命名的）的一些变量声明了正在运行的浏览器的名字和版本，还有一些变量标识了运行该浏览器的平台。有了这些变量，脚本就能够根据浏览器或平台来定制自己的行为，这样一来，它们便可以利用某些版本支持的额外能力或者避开存在于某种平台上的 bug 了。
- 在客户端 JavaScript 1.2 中，`Screen` 对象提供了显示 Web 浏览器的显示器的大小和色度的信息。
- 从 JavaScript 1.1 开始，`Window` 对象的 `scroll()` 方法就允许 JavaScript 程序在 X 方向和 Y 方向滚动窗口了。在 JavaScript 1.2 中，涌现了一批用于移动窗口或调整窗口大小的方法，它们代替了上述的 `scroll()` 方法。

1.6.7 JavaScript 不能做什么

客户端 JavaScript 具有一连串给人留下深刻印象的功能。但是要注意，这些功能只

限于与浏览器相关的任务或与文档相关的任务。由于客户端 JavaScript 只能用于有限的环境中，所以它没有独立的语言所必需的特性：

- 除了能够动态地生成浏览器要显示的 HTML 文档（包括图像、表、框架、表单和字体、等等）之外，JavaScript 不具有任何图形处理能力。
- 出于安全性方面的原因，客户端 JavaScript 不允许对文件进行读写操作。显而易见，你一定不想让一个来自某个站点的不可靠程序在自己的计算机上运行，并且随意篡改你的文件。
- 除了能够引发浏览器下载任意 URL 所指的文档以及把 HTML 表单的内容发送给服务器端脚本、电子邮件地址之外，JavaScript 不支持任何形式的联网技术。

1.7 JavaScript 的安全性

只要程序（如 JavaScript 脚本、Visual Basic 程序或 Microsoft Word 宏）包含在共享文档中，尤其是通过 Internet 或电子邮件传输的文档，就可能存在病毒或其他恶意程序。JavaScript 语言的设计者意识到了这些安全方面的问题，所以没有赋予 JavaScript 程序执行破坏性操作的权利。例如，前面说明过，客户端 JavaScript 程序不能读本地文件或不能执行联网操作。

但是，由于 Web 浏览器环境非常复杂，在早期的浏览器版本中出现了大量的安全性问题。例如，在 Navigator 2 中，能够编写可以自动窃取网页访问者的电子邮件地址的 JavaScript 代码，然后不经访问者同意就以他们的名义自动发送电子邮件。不过这点和其他大量的安全漏洞已经被修正了。尽管不能保证不存在其他安全漏洞了，但是大多数有见识的用户都轻松地让现代浏览器运行页面上的 JavaScript 代码。第二十一章对客户端 JavaScript 的安全性做了完整的讨论。

1.8 例子：用 JavaScript 计算借贷支出

例 1-3 是一个重要的 JavaScript 程序的完整清单。该程序计算的是每个月的住房抵押和其他借贷的支出，其依据是借贷的数量、利息率和偿付周期。你会发现，这个程序是由一个 HTML 表单构成的，该表单可以与 JavaScript 代码进行交互。图 1-3 显

示了这个表单在浏览器中显示时的样子。不过这里的图只是程序的静态画面。程序中的 JavaScript 代码使它具有了动态效果，即当用户改变了借贷的数量、利息率或支付数目时，JavaScript 代码就会重新计算月支付额、支付总额和借贷期内支付的总利息。

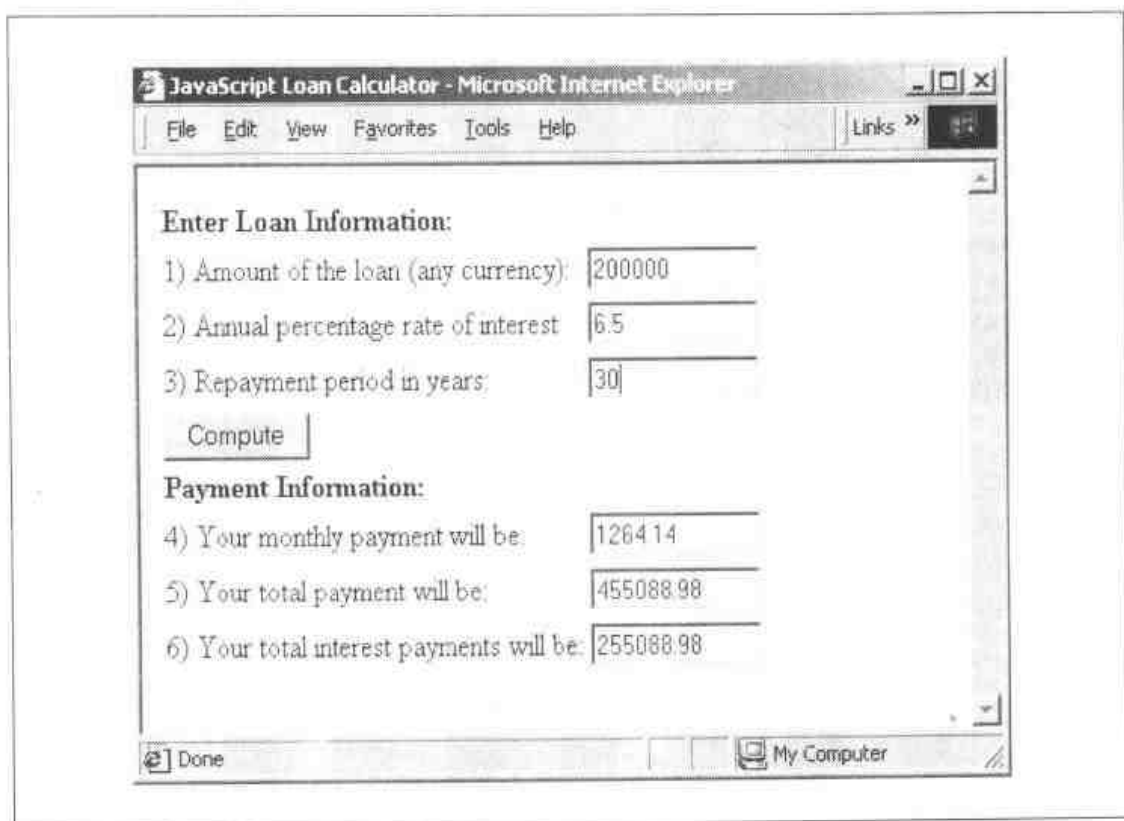


图 1-3: 用 JavaScript 编写的借贷支付金额计算器

本例的前半部分是一个 HTML 表单，它使用了一张 HTML 表将内容很好地组织了起来。注意，有几个表单元素定义了 `onchange` 或 `onclick` 事件处理器。当用户改变了输入或者点击了表单中显示的 **Compute** 按钮时，就会触发那些事件处理器。还要注意的，事件处理器性质的值是一个 JavaScript 代码串——`calculate()`。当事件处理器被触发时，它就会执行这些代码，也就是调用函数 `calculate()`。

函数 `calculate()` 是在本例的第二部分中定义的，它位于标记 `<script>` 之内。该函数先从表单中读取用户输入的信息，然后计算借贷支付金额，最后用表单最下面的三个元素显示出计算的结果。

例 1-3 虽然简单，但是却值得花时间仔细研究一番。目前你不应该期望自己能够完

全理解这里的JavaScript代码,研究这些代码只是使你对JavaScript程序的外观、事件处理器的作用以及JavaScript代码是如何嵌入到HTML表单中的有一个初步的印象。注意,注释包含在HTML标记<!--和-->之间,也有的注释在以符号//开头的JavaScript代码行中。

例 1-3: 用 JavaScript 程序计算借贷支出

```
<head><title>JavaScript Loan Calculator</title></head>
<body bgcolor= 'white'>
<!--
  以下是一个HTML表单,用户可以用它输入数据。
  JavaScript 则可以用它把计算的结果显示给用户。
  表元素嵌套在一个表中,这样可以美化它们的外观。
  表单自身名为“loandata”,表单中的域名为“interest”和“years”。
  表单后的JavaScript代码使用了这些域名。
  注意,有些表元素定义了“onchange”或“onclick”事件处理器,
  它们定义了用户输入数据或点击按钮时要执行的JavaScript代码串。
-->
<form name= 'loandata'>
  <table>
    <tr><td colspan= '3'><b>Enter Loan Information:</b></td></tr>
    <tr>
      <td>1)</td>
      <td>Amount of the loan (any currency):</td>
      <td><input type= 'text' name= 'principal' size= '12'
        onchange= 'calculate();'></td>
    </tr>
    <tr>
      <td>2)</td>
      <td>Annual percentage rate of interest:</td>
      <td><input type= 'text' name= 'interest' size= '12'
        onchange= 'calculate();'></td>
    </tr>
    <tr>
      <td>3)</td>
      <td>Repayment period in years:</td>
      <td><input type= 'text' name= 'years' size= '12'
        onchange= 'calculate();'></td>
    </tr>
    <tr><td colspan= '3'>
      <input type= 'button' value= 'Compute' onclick= 'calculate();'>
    </td></tr>
    <tr><td colspan= '3'>
      <b>Payment Information:</b>
    </td></tr>
    <tr>
      <td>4)</td>
      <td>Your monthly payment will be:</td>
      <td><input type= 'text' name= 'payment' size= '12'></td>
    </tr>
```



```

<tr>
  <td>5</td>
  <td>Your total payment will be:</td>
  <td><input type="text" name="total" size="12"></td>
</tr>
<tr>
  <td>6</td>
  <td>Your total interest payments will be:</td>
  <td><input type="text" name="totalinterest" size="12"></td>
</tr>
</table>
</form>

```

<!--

以下是使本例运行的JavaScript程序。

注意该脚本定义的calculate()函数，它由表单中的事件处理器调用。

该函数用上面的HTML代码中定义的名字引用表单中的域的值

```

<script language="JavaScript">
function calculate() {
  // 从表单中获得用户输入的数据。假定它们完全有效。
  // 把利息从百分比转换成十进制数。
  // 把年利率转换成月利率。
  // 把年支付额转换成月支付额。
  var principal = document.loandata.principal.value;
  var interest = document.loandata.interest.value / 100 / 12;
  var payments = document.loandata.years.value * 12;

  // 下面计算月支付额，使用了很多的数学函数。
  var x = Math.pow(1 + interest, payments);
  var monthly = (principal*x*interest)/(x-1);

  // 检查结果是否是无穷大的数。如果不是，就显示出结果。
  if (!isNaN(monthly) &&
      (monthly != Number.POSITIVE_INFINITY) &&
      (monthly != Number.NEGATIVE_INFINITY)) {

    document.loandata.payment.value = round(monthly);
    document.loandata.total.value = round(monthly * payments);
    document.loandata.totalinterest.value =
      round((monthly * payments) - principal);
  }
  // 否则，用户输入的数据是无效的，因此什么都不显示。
  else {
    document.loandata.payment.value = "";
    document.loandata.total.value = "";
    document.loandata.totalinterest.value = "";
  }
}

// 以下是个简单的方法，它将把数字舍入成两位小数的形式。
function round(x) {

```

```
        return Math.round(x*100)/100;
    }
</script>
</body>
</html>
```

1.9 如何使用本书其余的部分

本书的其余章节分为五部分。本章之后是第 I 部分，它介绍了 JavaScript 语言的核心。从第二章到第六章是这一部分的开始。它们介绍的内容虽然有些乏味，但是必读的，因为它们提供的是在学习一门新的程序设计语言时必须了解的信息。

- 第二章“词法结构”解释了该语言的基本结构。
- 第三章“数据类型和值”介绍了 JavaScript 支持的数据类型。
- 第四章“变量”介绍了变量、变量作用域以及相关的主题。
- 第五章“表达式和运算符”对 JavaScript 中的表达式进行了说明，并且还介绍了 JavaScript 支持的所有运算符。由于 JavaScript 的语法是模拟 Java 的，而 Java 又是模拟 C 和 C++ 的，所以熟悉 C、C++ 或 Java 的程序设计者们可以跳过该章。
- 第六章“语句”介绍了每一种 JavaScript 语句的语法和用法。同样的，熟悉 C、C++ 或 Java 的程序设计者可以跳过该章的某些内容。

第一部分接下来的五章就要有趣得多了。虽然它们介绍的仍旧是 JavaScript 语言的核心，但却是你不熟悉的那部分内容，即使你已经对 C 或 Java 了如指掌了。如果你想真正地理解 JavaScript，那么就需要对这几章仔细研究一番：

- 第七章“函数”介绍了在 JavaScript 中是如何定义函数、调用函数以及操作函数的。
- 第八章“对象”介绍了 JavaScript 中最重要的数据类型——对象。该章讨论了 JavaScript 中面向对象的程序设计技术，并且说明了如何用 JavaScript 定义你自己的对象类。
- 第九章“数组”介绍了 JavaScript 中数组的创建和使用。

- 第十章“使用正则表达式的模式匹配”说明了在 JavaScript 中如何使用正则表达式进行模式匹配和查找替换的操作。
- 第十一章“JavaScript 的更多主题”介绍了前面没有讨论过的高级主题。第一次通读全书的时候，可以略过该章，但是如果想成为一个 JavaScript 高手，那么理解它所包含的内容是至关重要的。

本书的第二部分介绍的是客户端 JavaScript。这一部分的各章介绍了客户端 JavaScript 的核心——Web 浏览器的各个对象，并且还提供了有关这些对象的用法的示例程序。要想使一个运行在 Web 浏览器中的 JavaScript 程序生动有趣，必定要依靠专用于客户端的 JavaScript 特性。

第二部分的内容包括：

- 第十二章“Web 浏览器中的 JavaScript”介绍了 JavaScript 与 Web 浏览器的集成。它把 Web 浏览器作为一种程序设计环境进行了讨论，并且还解释了把 JavaScript 代码集成到网页中以便在客户端执行的各种方法。
- 第十三章“窗口和框架”介绍了客户端 JavaScript 中最核心，也是最重要的对象——Window 对象。此外它还介绍了几个与 Window 对象有关的重要对象。
- 第十四章“Document 对象”介绍了 Document 对象以及把 HTML 文档的内容展示给 JavaScript 代码的有关对象。
- 第十五章“表单和表单元素”介绍了表示 HTML 表单的 Form 对象。此外，它还介绍了出现在 HTML 表单中的各种表单元素对象，并且展示了一些使用表单的 JavaScript 程序。
- 第十六章“脚本化 cookie”说明了如何用 cookie 在 Web 程序中保存状态。
- 第十七章“文档对象模型”说明了 W3C DOM 标准的核心片段，并且展示了 JavaScript 脚本如何访问 HTML 文档中的任意一个元素。
- 第十八章“级联样式表和动态 HTML”介绍了使 JavaScript 程序能够操作 HTML 文档中的样式表、元素的外观和位置的那部分 W3C DOM 标准。该章向你展示了如何用 CSS 属性创造大量的 DHTML 效果。
- 第十九章“事件和事件处理”介绍了 JavaScript 的事件和事件处理器，它们对

于要和用户进行交互的 JavaScript 程序来说是非常重要的。该章涵盖了传统的事件模型、W3C DOM 标准的事件模型和 Internet Explorer 专有的事件模型。

- 第二十章“兼容性”探索了 JavaScript 程序设计技术中有关兼容性的重要问题，并且讨论了编写能够在大多数 Web 浏览器上正确运行的 JavaScript 代码的方法。
- 第二十一章“JavaScript 的安全性”列举了建立在客户端 JavaScript 的安全限制，并且解释了它们的基本原理。
- 第二十二章“在 JavaScript 中使用 Java”说明了如何用 JavaScript 与 Java 小程序进行通信，并且对它们进行控制。此外它还介绍了如何进行反向的操作，即在 Java 小程序中调用 JavaScript 代码。

第三、四、五部分是参考手册，分别说明了 JavaScript 语言核心定义的对象、传统的客户端 JavaScript 定义的对象和新的 W3C DOM 标准定义的对象。

1.10 JavaScript 探秘

要真正学会一种新的程序设计语言，就要用它来编写程序。在阅读本书时，我建议你一边学习 JavaScript 的特性，一边对它们做一些尝试。有许多方法可以使 JavaScript 变的简单易行。

研究 JavaScript 最简单的方法就是编写简单的脚本。客户端 JavaScript 的好处之一是要编写 JavaScript 脚本不必购买或下载专用的软件，一个 Web 浏览器和一个简单的文本编辑器就构成了一个完整的开发环境。我们在本章开始的时候看到过一个计算阶乘的例子。假定你想对它做如下的修改以显示一个斐波纳契数：

```
<script>
document.write("<h2>Table of Fibonacci Numbers</h2>");
for (i=0, j=1, k=0, fib=0; i<50; i++, fib=j+k, j=k, k=fib){
    document.write("Fibonacci ('  + 1 + ') = " + fib);
    document.write("<br> ");
}
</script>
```

也许这段代码有些令人费解（如果你仍然不能理解，不必担心），不过其中关键的一点是，当你想试验像它一样短小的程序时，所要做的只是把代码输进去，然后在浏

览器中用一个本地的 `file:URL` 试运行这些代码即可。注意这段代码使用了 `document.write()` 方法来显示 HTML 文件的输出,以便你能够看到它的计算结果。这是试验 JavaScript 的一种重要方法。另一种方法是使用 `alert()` 方法在对话框中显示纯文本的输出:

```
alert("Fibonacci ( 1 + 1 + ... ) = " + fib);
```

还要注意的是,像这种简单的 JavaScript 试验,通常可以省略 HTML 文件中的 `<html>`、`<head>` 和 `<body>` 标记。

对于更简单的 JavaScript 试验,有时可以使用 `javascript:URL` 伪协议来计算 JavaScript 表达式并返回计算的结果。一个 JavaScript URL 是由 `javascript:` 协议说明符加上任意的 JavaScript 代码(语句之间用分号隔开)构成的。当浏览器装载了这样的 URL 时,它将执行其中的 JavaScript 代码。这样的 URL 中的最后一个表达式的值将被转换成字符串,该字符串会被作为新文档显示在 Web 浏览器中。例如,你可以在浏览器的 **Location** 字段中输入如下的 JavaScript URL 以检测你对某些 JavaScript 运算符和语句的理解:

```
javascript:5%2  
javascript:x = 3; (x < 5)? 'x is less': "x is greater"  
javascript:d = new Date(); typeof d;  
javascript:for(i=0,j=1,k=0,fib=1; i<10; i++,fib=j+k,k=j,j=fib) alert(fib);  
javascript:s=""; for(i in document) s+=i+": "+document[i]+ "\n"; alert(s);
```

在研究 JavaScript 时,可能会编写出运行结果与你的期望不相符的代码,并且应该对它进行调试。调试 JavaScript 的基本方法和调试其他多种语言的方法一样,即在代码中插入语句,输出有关变量的值,以便你能够断定到底是哪里出了问题。我们已经知道,使用方法 `document.write()` 可以实现这一点。但是,该方法是不能从事件处理器中调用的,而且还有其他的缺陷,所以使用 `alert()` 函数在一个单独的对话框中显示调试信息较为容易一些。

另外,上面提到的 `for/in` 循环(第六章中详细介绍)对调试也非常有用。例如,你可以和 `alert()` 方法一起使用它,编写出一个函数来显示对象所有属性的名字和值的列表。这种函数在研究 JavaScript 语言和调试代码时都非常有用。

祝你使用 JavaScript 时一帆风顺,度过一个有趣的探秘之旅。

第一部分

JavaScript 的核心

本书的这一部分包括第二章到第十一章的内容，介绍了Web浏览器、Web服务器和其他嵌入了JavaScript实现中的JavaScript语言的核心。它是JavaScript语言的一个参考。在学习这门语言时，读完了这一部分，你就会发觉自己常常会参考这一部分，依靠它来解决JavaScript中的某些复杂问题。

- 第二章、词法结构
- 第三章、数据类型和值
- 第四章、变量
- 第五章、表达式和运算符
- 第六章、语句
- 第七章、函数
- 第八章、对象
- 第九章、数组
- 第十章、使用正则表达式的模式匹配
- 第十一章、JavaScript的更多主题

第二章

词法结构

程序设计语言的词法结构是一套基本规则，用来详细说明如何用这种语言来编写程序。它是一种语言的最低层次的语法，指定了变量名是什么样的，注释应该使用什么字符以及语句之间是如何分隔的等规则。本章比较短，主要介绍了JavaScript的词法结构。

2.1 字符集

JavaScript程序是用Unicode字符集编写的。与7位的ASCII编码（只适用于英语）和8位的ISO Latin-1编码（只适用于英语和西欧语言）不同，16位的Unicode编码可以表示地球上通用的每一种书面语言。这是国际化的一个重要特征，对那些不讲英语的程序设计者尤为重要。

美国以及其他讲英语的国家的程序设计者通常都用仅支持ASCII码和Latin-1编码的文本编辑器编写程序，因此他们难以访问完整的Unicode字符集。但是这并不成问题，因为ASCII编码和Latin-1编码都是Unicode编码的子集，所以用这两种编码集合编写的JavaScript程序都是绝对有效的。JavaScript程序中的每个字符都是用两个字节表示的，习惯于认为字符都是用8位表示的程序设计者可能会对此感到惊讶，不过这个事实对程序设计者来说是透明的，所以可以忽略它。

虽然ECMAScript v3标准允许Unicode字符出现在JavaScript程序中的任何地方，

但是该标准的第1版和第2版都只允许Unicode字符出现在注释或用引号括起的字符串直接量中, ECMAScript v1程序的其他部分只能用ASCII字符集。ECMAScript标准化之前的JavaScript版本通常根本不支持Unicode编码。

2.2 大小写敏感性

JavaScript是一种区分大小写的语言。这就是说, 在输入语言的关键字、变量、函数名以及所有的标识符时, 都必须采取一致的字符大小写形式。例如, 关键字“while”就必须被输入为“while”, 而不能输入为“While”或者“WHILE”。同样的, “online”、“Online”、“OnLine”和“ONLINE”是四个不同的变量名。

但是要注意, HTML并不区分大小写, 由于它和客户端JavaScript紧密相关, 所以这一点是很容易混淆的。许多JavaScript对象和属性都与它们所代表的HTML标记和性质同名。在HTML中这些标记和性质名可以以任意的大小写方式输入, 但是在JavaScript中它们通常都是小写的。例如, 在HTML中, 事件处理程序性质onclick通常被声明为onClick, 但是在JavaScript代码中就只能使用onclick。

虽然JavaScript语言的核心完全、纯粹地区分大小写, 但是客户端JavaScript中却存在一些不符合这一规则的例外。例如, 在Internet Explorer 3中, 所有的客户端对象和属性都是不区分大小写的。但是这却产生了IE与Netscape的兼容问题, 所以在Internet Explorer 4及后续版本中, 所有客户端对象和属性就都区分大小写了。

2.3 空白符和换行符

JavaScript会忽略程序中记号之间的空格、制表符和换行符, 除非它们是字符串或正则表达式直接量的一部分。所谓记号(token), 就是一个关键字、变量名、数字、函数名或者其他实体, 显然你不想在它们中间插入空格或换行符。如果在一个记号中间插入了空格、制表符或换行符, 那么就将它分成了两个记号, 例如, 123是一个数字记号, 而12 3则是两个独立的记号(这就构成了一个语法错误)。

因为可以在程序中随意使用空格、制表符和换行符(除了在字符串、正则表达式和记号中), 所以你就可以采用整齐、一致的方式自由安排程序的格局, 在其中使用缩

进,这样能够使代码容易阅读和理解。但是要注意,对换行符的放置有一点小小的限制,这将在下一节中介绍。

2.4 可选的分号

JavaScript中的简单语句后通常都有分号(;),就像C、C++和Java中的语句一样。这主要是为了分隔语句。但是在JavaScript中,如果语句分别放置在不同的行中,就可以省去分号。例如,下面的代码就可以不用分号:

```
a = 3;  
b = 4;
```

但是如果代码的格式如下,那么第一个分号就是必需的:

```
a = 3; b = 4;
```

省略分号并不是一个好的编程习惯,应该习惯于使用分号。

尽管理论上说来JavaScript允许在任意两个记号之间放置换行符,但是实际上JavaScript会自动为你插入分号,使这一规则产生了异常。如果你以上述方式打断了一行,以至于使换行符之前的一行成了一个完整的语句,那么JavaScript就会认为你漏掉了分号,并会替你插入一个分号,这就改变了你的初衷。通常在使用return语句、break语句和continue语句时应该注意这一点。例如,考虑如下的语句:

```
return  
true;
```

JavaScript会假定你的意图是:

```
return;  
true;
```

但是,实际上你的意图是:

```
return true;
```

这点需要特别注意,因为这种代码并不会引起语法错误,但是却会因为产生一种不明确的状态而导致错误。如果你编写的代码如下就会发生同样的问题:

```
break  
outerloop;
```

JavaScript 会在关键字 `break` 之后插入一个分号。当它要解释下一行代码时，就会引起语法错误。出于同样的原因，后缀运算符 `++` 和 `--`（参见第五章）也要和它们所作用的表达式处于同一行中。

2.5 注释

和 Java 一样，JavaScript 也支持 C++ 型的注释和 C 型注释。JavaScript 会把处于“//”和一行结尾之间的任何文本都当作注释忽略掉。此外“/*”和“*/”之间的文本也会被当作注释。这些 C 型的注释可以跨越多行，但是其中不能有嵌套的注释。接下来的代码都是合法的 JavaScript 注释：

```
// 这是一条单行注释。  
/* 这也是一条注释 */    /* 此处是另一条注释。 */  
/*  
 * 这还是另一种注释。  
 * 它是多行的。  
 * /
```

2.6 直接量

所谓直接量 (literal)，就是程序中直接显示出来的数据值。下面列出的都是直接量：

```
12                // 数字 12  
1.2              // 数字 1.2  
'hello world'    // 一个文本串  
'Hi'            // 另一个字符串  
true             // 一个布尔值  
false            // 另一个布尔值  
/javascript/gi    // 一个“正则表达式”直接量（用于模式匹配）  
null             // 一个对象不存在
```

ECMAScript v3 还支持作为数组直接量和对象直接量的表达式。例如：

```
{ x:1, y:2 }      // 对象初始化程序  
[1,2,3,4,5]       // 数组初始化程序
```

注意，从 JavaScript 1.2 起就已经支持数组直接量和对象直接量了，但是直到 ECMAScript v3 才把它们标准化。

直接量对任何一种程序设计语言来说都是一个重要的部分，因为要编写不含直接量的程序几乎是不可能的。第三章将具体介绍 JavaScript 语言中的各种直接量。

2.7 标识符

所谓标识符 (identifier)，就是一个名字。在 JavaScript 中，标识符用来命名变量和函数，或者用作 JavaScript 代码中某些循环的标签。JavaScript 中合法的标识符的命名规则和 Java 以及其他许多语言的命名规则相同，第一个字符必须是字母、下划线 (`_`) 或美元符号 (`$`) (注 1)。接下来的字符可以是字母、数字或下划线、美元符号 (数字不允许作为首字符出现，这样 JavaScript 可以轻易地区别开标识符和数字了)。下面是合法的标识符：

```
_
my_variable_name
v13
__dummv
$str
```

在 ECMAScript v3 中，标识符中的字母和数字来自完整的 Unicode 字符集。在这个标准之前的版本中，JavaScript 标识符中字符仅限于使用 ASCII 字符集。此外，ECMAScript v3 还允许标识符中有 Unicode 转义序列。所谓 Unicode 转义序列，是字符 `\u` 后接 4 个十六进制的数字，用来指定一个 16 位的字符编码。例如，标识符 `π` 还能写作 `\u03c0`。尽管这种语法比较笨拙，但是它却可以将含有 Unicode 字符的 JavaScript 程序转换成一个表单，这样即使是不支持 Unicode 字符集的文本编辑器和其他工具也能执行该表单了。

最后要说的是，标识符不能和 JavaScript 中用于其他目的的关键字同名。下面一节列出了 JavaScript 中保留的特殊名字。

注 1： 注意，在 JavaScript 1.1 之前的版本中，使用美元符号是不合法的。只有生成代码的工具才会使用它，所以在编写代码时应尽量避免使用美元符号。

2.8 保留字

下面列出了许多 JavaScript 的保留字，它们在 JavaScript 程序中是不能被用作标识符的（变量名、函数名以及循环标签）。表 2-1 列出了 ECMAScript v3 标准化的关键字。这些关键字对 JavaScript 来说具有特殊的意义，它们是这种语言的语法自身的一部分。

表 2-1：保留的 JavaScript 的关键字

| | | | | |
|----------|----------|------------|--------|--------|
| break | do | if | switch | typeof |
| case | else | in | this | var |
| catch | false | instanceof | throw | void |
| continue | finally | new | true | while |
| default | for | null | try | with |
| delete | function | return | | |

表 2-2 列出了其他的保留关键字。虽然现在 JavaScript 已经不使用这些保留字了，但是 ECMAScript v3 保留了它们，以备扩展语言。

表 2-2：ECMA 扩展保留的关键字

| | | | | |
|----------|---------|------------|-----------|--------------|
| abstract | double | goto | native | static |
| boolean | enum | implements | package | super |
| byte | export | import | private | synchronized |
| char | extends | int | protected | throws |
| class | final | interface | public | transient |
| const | float | long | short | volatile |
| debugger | | | | |

除了上面列出的正式保留字外，当前 ECMAScript v4 标准的草案正在考虑关键字 `as`、`is`、`namespace` 和 `use` 的用法。虽然目前的 JavaScript 解释器不会阻止你将这四个关键字用作标识符，但是应该避免这样用它们。

此外，你还应该避免把 JavaScript 预定义的全局变量名或全局函数名用作标识符。如果用这些名字创建变量或函数，就会得到一个错误（如果该属性是只读的）或重定义了已经存在的变量或函数。你不应该这样做，除非你绝对明确自己正在做什么。表 2-3 列出了 ECMAScript v3 标准定义的全局变量和全局函数。不同的 JavaScript

版本可能会定义其他的全局属性，每个特定的JavaScript嵌入（客户端、服务器端，等等）会有自己的全局属性扩展列表（注2）。

表 2-3: 要避免使用的其他标识符

| | | | | |
|--------------------|--------------------|----------|----------------|-------------|
| arguments | encodeURIComponent | Infinity | Object | String |
| Array | Error | isFinite | parseFloat | SyntaxError |
| Boolean | escape | isNaN | parseInt | TypeError |
| Date | eval | Math | RangeError | undefined |
| decodeURI | EvalError | NaN | ReferenceError | unescape |
| decodeURIComponent | Function | Number | RegExp | URIError |

注2: 由客户端JavaScript定义的附加全局变量可查阅本书第四部分客户端参考手册的Window对象部分。

第三章

数据类型和值

计算机程序是通过操作值 (value) (如数字 3.14 或文本 “Hello World”) 来运行的。在一种程序设计语言中, 能够表示并操作的值的类型称为数据类型 (data type), 而程序设计语言最基本的特征之一就是它支持的数据类型的集合。JavaScript 允许你使用三种基本数据类型——数字、文本字符串和布尔值。此外, 它还支持两种小数据类型 null (空) 和 undefined (未定义), 它们各自只定义了一个值。

除了这些基本的数据类型外, JavaScript 还支持复合数据类型——对象。对象 (是数据类型对象的成员之一) 表示的是值 (既可以是基本值, 如数字和字符串, 也可以是复合值) 的集合。JavaScript 中的对象有两种, 一种对象表示的是已命名的值的无序集合, 另一种表示的是有编号的值的有序集合。后者被称为数组 (array)。虽然从根本上来说, JavaScript 中的对象和数组是同一种数据类型, 但是它们的行为却极为不同, 所以本书通常将它们看作两种不同的类型。

JavaScript 还定义了另一种特殊的对象——函数 (function)。函数是具有可执行代码的对象, 可以通过调用函数执行某些操作。和数组一样, 函数的行为与其他类型的对象不同, JavaScript 为函数定义了专用的语法。因此, 我们将函数看作独立于对象和数组的数据类型。

除了函数和数组外, JavaScript 语言的核心还定义了一些专用的对象。这些对象表示的不是新的数据类型, 而是新的对象类 (class)。Date 类定义的是表示日期的对象, RegExp 类定义的是表示正则表达式的对象 (一种强大的模式匹配工具, 将

在第十章中介绍), Error 类定义的是表示 JavaScript 程序中发生的语法和运行时错误的对象。

本章其余的小节详细说明了每种基本数据类型。此外还介绍了对象、数组和函数这些数据类型, 第七、八、九章将对它们进行完整的说明。本章的最后对 Date 对象、RegExp 对象和 Error 对象进行了概述, 在本书的语言核心的参考手册部分可以看到它们的完整说明。

3.1 数字

数字 (number) 是最基本的数据类型, 它们几乎无需解释。JavaScript 和其他程序设计语言 (如 C 和 Java) 的不同之处在于它并不区别整型数值和浮点型数值。在 JavaScript 中, 所有的数字都是由浮点型表示的。JavaScript 采用 IEEE 754 标准定义的 64 位浮点格式表示数字 (注 1), 这意味着它能表示的最大值是 $\pm 1.7976931348623157 \times 10^{308}$, 最小值是 $\pm 5 \times 10^{-324}$ 。

当一个数字直接出现在 JavaScript 程序中时, 我们称它为数值直接量 (numeric literal)。JavaScript 支持数值直接量的形式有几种, 我们将在接下来的小节中讨论这些形式。注意, 在任何数值直接量前加负号 (-) 可以构成它的负数。但是负号是 - 一元求反运算符 (参阅第五章), 它不是数值直接量语法的一部分。

3.1.1 整型直接量

在 JavaScript 程序中, 十进制的整数是一个数字序列。例如:

```
0
5
10000000
```

JavaScript 的数字格式允许精确地表示 $-9007199254740992 (-2^{53})$ 和 $9007199254740992 (2^{53})$ 之间的所有整数 (包括 $-9007199254740992 (-2^{53})$ 和

注 1: Java 语言的程序设计者应该熟悉这种格式, 即 Java 中的 double 类型的格式。它也是几乎现在所有的 C 和 C++ 实现中使用的 double 格式。

9007199254740992 (2^{53})。但是使用超过这个范围的整数，就会失去尾数的精确性。需要注意的是，JavaScript 中的某些整数运算（尤其是第五章中介绍的逐位运算符）是对32位的整数执行的，它们的范围从 -2^{31} 到 $2^{31}-1$ 。

3.1.2 八进制和十六进制的直接量

除了十进制的整型直接量，JavaScript 还能识别十六进制（以16为基数）的直接量。所谓十六进制的直接量，是以“0X”或者“0x”开头，其后跟随十六进制数字串的直接量。十六进制的数字可以是0到9中的某个数字，也可以是a (A) 到f (F) 中的某个字母，它们用来表示0到15之间（包括0和15）的某个值，下面是十六进制整型直接量的例子：

```
0xff // 15*16 + 15 = 255 (基数是10)
0xAfE911
```

尽管ECMAScript 标准不支持八进制的直接量，但是JavaScript 的某些实现却允许你采用八进制（基数为8）格式的整型直接量。八进制的直接量以数字0开头，其后跟随一个数字序列，这个序列中的每个数字都在0和7之间（包括0和7），例如：

```
0377 // 3*64 + 7*8 + 7 = 255 (基数是10)
```

由于某些JavaScript 实现支持八进制的直接量，而有些则不支持，所以最好不要使用以0开头的整型直接量，毕竟你不知道某个JavaScript 实现是将它解释为十进制，还是解释为八进制。

3.1.3 浮点型直接量

浮点型直接量可以具有小数点，它们采用的是传统科学记数法的语法。一个实数值可以被表示为整数部分后加小数点和小数部分。

此外，还可以使用指数记数法表示浮点型直接量，即实数后跟随字母e或E，后面加上正负号，其后再加一个整型指数。这种记数法表示的数值等于前面的实数乘以10的指数次幂。

为了说明得更为简洁，该语法如下：

```
[digits][.digits[(E|e)[(+|-)]digits]
```

例如:

```
3.14
2345.789
.3333333333333333
6.02e23      // 6.02 × 1023
1.4136203E-50 // 1.4738223 × 10-50
```

注意, 虽然实数有无穷多个, 但是 JavaScript 的浮点格式能够精确表示出来的却是有限的 (确切地说是 18437736874454810627 个)。这意味着当你在 JavaScript 中使用实数时, 表示出的数字通常是真实数字的近似值。不过即使是近似值也足够用了, 这并不是个实际问题。

3.1.4 数字的使用

JavaScript 是使用语言自身提供的算术运算符来进行数字运算的。这些运算符包括加法运算符 (+)、减法运算符 (-)、乘法运算符 (*) 和除法运算符 (/)。第五章将详细介绍这些和其他的算术运算符。

除了基本的算术运算外, JavaScript 还采用了大量的算术函数, 来支持更为复杂的算术运算, 这些函数是语言核心的一部分。为了方便起见, 这些函数都被保存为 Math 对象的属性, 因此我们总是使用直接量名 Math 来访问这些函数。例如, 下面列出了如何计算数字 x 的正弦值:

```
sine_of_x = Math.sin(x);
```

下面是一个计算数字的平方根的表达式:

```
hypot = Math.sqrt(x*x + y*y);
```

参阅本书核心参考手册中介绍的 Math 对象和其后的列表, 就可以了解 JavaScript 支持的所有算术函数。

还有一种有趣的方法可以用于数字。方法 toString() 可以用它的参数指定的基数或底数 (底数必须在 2 和 36 之间) 把数字转换成字符串。例如, 要将一个数字转换为二进制数字, 可以使用如下的方式:

```
var x = 33;  
var y = x.toString(2); // y是'100001'
```

要调用一个数值直接量的toString()方法,就必须使用括号,这样才能避免将“.”解释为小数点:

```
var y = (255).toString(0x10); // y是'101'
```

3.1.5 特殊的数值

JavaScript 还使用了一些特殊的数值。当一个浮点值大于所能表示的最大值时,其结果是一个特殊的无穷大值,JavaScript将它输出为Infinity。同样的,当一个负值比所能表示的最小的负值还小时,结果就是负无穷大,输出为-Infinity。

另一个特殊的JavaScript数值是当一个算术运算(如用0来除0)产生了未定义的结果或错误时返回的。在这种情况下,结果是一个非数字的特殊值,输出为NaN。这个非数字值的行为有些不寻常,因为它和任何数值都不相等,包括它自己在内,所以需要有一个专门的函数isNaN()来检测这个值。相关的函数isFinite()用来检测一个数字是否是NaN、正无穷大或负无穷大。

JavaScript 为每个特殊的数值都定义了常量,表3-1列出了这些常量。

表 3-1: 特殊数值的常量

| 常量 | 含义 |
|--------------------------|--------------------|
| Infinity | 表示无穷大的特殊值 |
| NaN | 特殊的非数字值 |
| Number.MAX_VALUE | 可表示的最大数字 |
| Number.MIN_VALUE | 可表示的最小数字(与零最接近的数字) |
| Number.NaN | 特殊的非数字值 |
| Number.POSITIVE_INFINITY | 表示正无穷大的特殊值 |
| Number.NEGATIVE_INFINITY | 表示负无穷大的特殊值 |

ECMAScript v1 标准定义了Infinity和NaN两个常量,在JavaScript 1.3 之前的版本中没有实现它们。但是从JavaScript 1.1 起就已经实现了各种Number 常量。

3.2 字符串

字符串 (string) 是由 Unicode 字符、数字、标点符号等组成的序列，它是 JavaScript 用来表示文本的数据类型。不久你就会看到，程序中的字符串直接量是包含在单引号或双引号中的。注意，JavaScript 和 C 以及 C++、Java 不同的是，它没有 char 这样的字符数据类型。要表示单个字符，必须使用长度为 1 的字符串。

3.2.1 字符串直接量

所谓字符串，就是由单引号或双引号 (' 或 ") 括起来的 Unicode 字符序列，其中可以含有 0 个或多个 Unicode 字符。由单引号定界的字符串中可以含有双引号，由双引号定界的字符串中也可以含有单引号。字符串直接量必须写在一行中，如果将它们放在两行中，可能会将它们截断。如果你必须在字符串直接量中添加一个换行符，可以使用字符序列 `\n`，下一节将对此进行说明。字符串直接量的例子如下所示：

```
"", // 空串，它的字符数为 0
'testing'
"3.14"
'name: myform'
'wouldn't you prefer O'Reilly's book?'
'This string\nhas two lines'
'π is the ratio of a circle's circumference to its diameter'
```

如上面最后一个字符串示例所示，ECMAScript v1 标准允许字符串直接量使用 Unicode 字符。但是 JavaScript 1.3 之前的版本中的字符串通常只支持 ASCII 字符和 Latin-1 字符。在下一节中我们将看到，还可以采用转义序列把 Unicode 字符添加到字符串直接量中。如果你的文本编辑器不支持完整的 Unicode 字符集，这是非常有用的。

注意，当你使用单引号来界定字符串时，必须注意英文的缩写和所有格，如 “can't” 和 “O'Reilly's”。由于撇号和单引号相同，所以必须使用反斜线符号 (\) 来转义带有单引号的字符串中出现的撇号（下一节将对此进行解释）。

在客户端 JavaScript 程序设计中，JavaScript 代码中常含有 HTML 代码串，HTML 代码中也常含有 JavaScript 代码串。和 JavaScript 一样，HTML 也使用单引号或双引号来界定字符串。因此，在同时使用 JavaScript 和 HTML 时，最好对 JavaScript 采用一种引用方式，对 HTML 采用另一种引用方式。在下面的例子中，字符串

“Thank you”在JavaScript中用单引号界定，在HTML的事件处理器性质中则用双引号界定：

```
<a href="#" onclick='alert( "Thank you" )>Click Me</a>
```

3.2.2 字符串直接量中的转义序列

在JavaScript的字符串中，反斜线（\）具有特殊的用途。在反斜线符号后加一个字符就可以表示在字符串中无法出现的字符了。例如，\n 是一个转义序列（escape sequence），它表示的是一个换行符（注2）。

另一个例子是上节中提到的“\”，它表示单引号（或撇号）。当你需要在以单引号界定的字符串直接量中使用撇号时，它就显得非常有用。现在你就会明白我们为什么称它们为转义序列了，因为反斜线符号可以使我们避免使用单引号字符的常规解释，它代表一个撇号，而不是用来标记字符串结尾的：

```
'You\'re right, it can\'t be a quote'
```

表3-2列出了JavaScript的转义序列以及它们所代表的字符。其中有两个转义序列是通用的，通过把Latin-1或Unicode字符编码表示为十六进制数，它们可以表示任意字符。例如，转义序列\xA9表示的是版权符号，它采用十六进制数A9表示Latin-1编码。同样的，\u表示的是由四位十六进制数指定的任意Unicode字符。如\u03c0表示的是字符 π 。注意，虽然ECMAScript v1标准要求使用Unicode字符转义，但是JavaScript 1.3之前的版本通常不支持转义符。有些JavaScript版本还允许用反斜线符号后加三位八进制数字来表示Latin-1字符，但是ECMAScript v3标准不支持这种转义序列，所以不应该再使用它们。

表3-2: JavaScript的转义序列

| 序列 | 所代表的字符 |
|----|-----------------|
| \0 | NUL 字符 (\u0000) |
| \b | 退格符 (\u0008) |
| \t | 水平制表符 (\u0009) |

注2: C、C++ 和 Java 程序设计者已经对此和其他 JavaScript 转义序列相当熟悉了。

表 3-2: JavaScript 的转义序列 (续)

| 序列 | 所代表的字符 |
|---------------------|---|
| <code>\n</code> | 换行符 (<code>\u000A</code>) |
| <code>\v</code> | 垂直制表符 (<code>\u000B</code>) |
| <code>\f</code> | 换页符 (<code>\u000C</code>) |
| <code>\r</code> | 回车符 (<code>\u000D</code>) |
| <code>\"</code> | 双引号 (<code>\u0022</code>) |
| <code>\'</code> | 撇号或单引号 (<code>\u0027</code>) |
| <code>\\</code> | 反斜线符 (<code>\u005C</code>) |
| <code>\xXX</code> | 由两位十六进制数值 <code>XX</code> 指定的 Latin-1 字符 |
| <code>\uXXXX</code> | 由四位十六进制数 <code>XXXX</code> 指定的 Unicode 字符 |
| <code>\XXX</code> | 由一位到三位八进制数 (1 到 377) 指定的 Latin-1 字符。ECMAScript v3 不支持, 不要使用这种转义序列 |

最后要注意, 不能在换行符前用反斜线转义符使字符串 (或其他 JavaScript) 标记跨两行或在字符串中包含一个换行直接量。如果 `\` 位于表 3-2 中所示的字符之外的字符前, 则忽略 `\` (当然, JavaScript 语言将来的版本可能定义新的转义序列)。例如, `\#` 等价于 `#`。

3.2.3 字符串的使用

JavaScript 的内部特性之一就是能够连接字符串。如果将加号 (+) 运算符用于数字, 那么它将把两个数字相加。但是, 如果你将它作用于字符串, 它就会把这两个字符串连接起来, 将第二个字符串附加在第一个之后。例如:

```
msg = 'Hello, ' + 'world';    // 生成字符串 "Hello, world"
greeting = "Welcome to my home page," + " " + name;
```

要确定一个字符串的长度 (它包含的字符数), 可以使用字符串的 `length` 属性。如果变量 `s` 包含一个字符串, 可以使用如下的方式访问它的长度:

```
s.length
```

关于字符串有许多的操作，例如，可以获取字符串 `s` 的最后一个字符：

```
last_char = s.charAt(s.length - 1)
```

可以从字符串 `s` 中抽出第二、三、四个字符：

```
sub = s.substring(1,4);
```

要在字符串 `s` 中查找第一个字母“a”的位置：

```
i = s.indexOf('a');
```

还有许多其他的方法用来操作字符串。你可以在本书核心参考手册部分的 `String` 对象和其后的列表中找到这些方法的完整说明。

你可以从上面的例子中发现，JavaScript 的字符串和 JavaScript 的数组一样，都是以 0 开始进行索引的。也就是字符串的第一个字符是字符 0。C、C++ 和 Java 的程序员会对此感到非常习惯的，但是对那些习惯于以字符 1 为数组和字符串的基数的程序员来说，要习惯这一点还得花费些精力。

在 JavaScript 的某些版本中，可以使用数组的表示法将单个字符从字符串中读出（但是这样不能写入），因此上面例子中对 `charAt()` 的调用也可以写成如下形式：

```
last_char = s[s.length - 1];
```

注意，该语法不是 ECMAScript v3 标准的一部分，也不可移植，因此应避免使用。

下面当我们讨论对象数据类型时，你将看到对象的属性和方法的用法与前面例子中字符串的属性和方法的用法是相同的。这并不意味着字符串就是一种对象类型。实际上，字符串是一种很独特的 JavaScript 数据类型。虽然它们使用对象访问属性和方法的语法，但是它们自身并非对象。我们将在本章的结尾处解释其中的原因。

3.3 布尔值

数值数据类型和字符串数据类型的值都无穷多，但是布尔数据类型只有两个值，这两个合法的值分别由直接量“true”和“false”表示。一个布尔值代表的是一个“真值”，它说明了某个事物是真还是假。

布尔值通常在 JavaScript 程序中比较所得的结果。例如：

```
a == 4
```

这行代码测试了变量 `a` 的值是否和数值 4 相等。如果相等，比较的结果就是布尔值 `true`，否则结果就是 `false`。

布尔值通常用于 JavaScript 的控制结构。例如，JavaScript 的 `if/else` 语句就是在布尔值为 `true` 时执行一个动作，而在布尔值为 `false` 时执行另一个动作。通常将一个创建布尔值的比较与使用这个比较的语句结合在一起。结果如下所示：

```
if (a == 4)
    b = b + 1;
else
    a = a + 1;
```

这段代码检测了 `a` 是否等于 4。如果相等，就给 `b` 增加 1，否则给 `a` 加 1。

有时可以把两个可能的布尔值看作是“on (true)”和“off (false)”，或者看作“yes (true)”和“no (false)”，这样比将它们看作是“true”和“false”更为方便。有时把它们看作 1 (true) 和 0 (false) 会更加有用（实际上 JavaScript 确实是这样做的，在必要时会将 `true` 转换成 1，将 `false` 转换成 0）（注 3）。

3.4 函数

函数 (function) 是一个可执行的 JavaScript 代码段，由 JavaScript 程序定义或由 JavaScript 实现预定义。虽然函数只被定义一次，但是 JavaScript 程序却可以多次执行或调用它。JavaScript 的函数可以带有实际参数或形式参数，用于指定这个函数执行计算要使用的一个或多个值，而且它还能返回一个值，以表示计算结果。JavaScript 的实现提供了许多预定义函数，如 `Math.sin()`，它用于计算角的正弦值。

注 3： C 程序员应该注意到，JavaScript 的布尔数据类型与 C 语言的截然不同，它只用整数值来模拟布尔值。Java 程序员也应该注意到，尽管 JavaScript 有布尔类型，但它不像 Java 的 `boolean` 数据类型那么纯粹，JavaScript 的布尔值可以被轻易地转换成其他数据类型，反之亦然。所以，在实践中，JavaScript 的布尔值的用法更像 C 的布尔值，而不像 Java 的布尔值。

JavaScript 程序也可以定义自己的函数，代码如下：

```
function square(x) { // 该函数名为 square 它只有一个参数 x。  
    // 函数主体从此处开始。  
    return x*x; // 该函数计算参数的平方，并返回平方值。  
} // 函数在此处结束。
```

一旦定义了函数，就可以调用它，只需要在函数名后边加上一个可选的、用逗号分隔的参数列表，该列表用括号括起来。下面是函数调用的代码：

```
y = Math.sin(x);  
y = square(x);  
d = compute_distance(x1, y1, z1, x2, y2, z2);  
move();
```

JavaScript 的一个重要特性是 JavaScript 代码可以对函数进行操作。在许多语言中（包括 Java），函数都只是语言的语法特性，它们可以被定义，被调用，但却不是数据类型。JavaScript 中的函数是一个真正的数据类型，这一点给语言带来了很大的灵活性。这就意味着函数可以被存储在变量、数组和对象中，而且函数还可以作为参数传递给其他函数，这是非常有用的。我们将在第七章中学习更多有关把函数作为数值来定义、调用以及使用的方法。

由于函数是和数字、字符串一样的数据类型，因此它也可以像其他类型的值一样被赋给对象的属性。当一个函数被赋给某个对象的属性时（对象数据类型和对象的属性将在 3.5 节进行介绍），它常常被当作那个对象的方法来引用。方法是面向对象的程序设计方法中的重要部分。我们将在第八章中了解到更多相关内容。

3.4.1 函数直接量

在前面一节中，我们看到了函数 `square()` 的定义。这里展示的语法是大多数 JavaScript 程序用来定义函数所使用的。不过，ECMAScript v3 提供了定义函数直接量的语法（在 JavaScript 1.2 及其后的版本中实现了）。函数直接量是用关键字 `function` 后加可选的函数名、用括号括起来的参数列表和用花括号括起来的函数体定义的。简而言之，函数直接量看起来就像个函数定义，只不过没有函数名。它们之间最大的差别是函数直接量可以出现在其他 JavaScript 表达式中。因此除了用函数定义来定义函数 `square()`：

```
function square(x) { return x*x; }
```

还可以用函数直接量来定义它:

```
var square = function(x) { return x*x; }
```

为了遵从LISP程序设计语言,用这种方式定义的函数有时被称为拉姆达(lambda)函数,这种语言是最先允许在程序的直接量数值中嵌入无名函数的语言之一。虽然人们选择在程序中使用函数直接量的原因时不是一目了然的,但是以后我们会发现,在高级脚本中,它非常方便、有用。

此外还有一种定义函数的方法,即把参数列表和函数体作为字符串传递给构造函数Function()。例如:

```
var square = new Function("x", "return x*x;");
```

用这种方法定义的函数通常没有用。用一个字符串来表示函数体非常笨拙。在许多JavaScript实现中,用这种方式定义的函数没有用前两种方式定义的函数有效。

3.5 对象

对象(object)是已命名的数据的集合。这些已命名的数据通常被作为对象的属性来引用(有时,它们被称为对象的“域”,但是这种称呼容易让人迷惑)。要引用一个对象的属性,就必须引用这个对象,在其后加句号和属性名。例如,如果一个名为image的对象有一个名为width和一个名为height的属性,我们可以使用如下方式引用这些属性:

```
image.width  
image.height
```

对象的属性在很多方面都与JavaScript变量相似。属性可以是任何类型的数据,包括数组、函数以及其他的对象,所以你可能会见到如下的JavaScript代码:

```
document.myForm.button
```

这里引用了一个对象的button属性,而这个对象本身又存储在对象document的myform属性中。

前面已经提到过,如果一个函数值是存储在某个对象的属性中的,那么那个函数通

常被称为方法，属性名也就变成了方法名。要调用一个对象的方法，就要使用“.”语法将函数值从对象中提取出来，然后再使用“()”语法调用那个函数。例如，要调用 Document 对象的 write() 方法，可以使用如下的代码：

```
document.write("this is a test");
```

JavaScript 中的对象可以作为关联数组使用，因为它们能够将任意的数据值和任意的字符串关联起来。如果采用这种方式使用对象，那么要访问对象的属性就要采取不同的语法，即使用一个由方括号封闭起来的、包含所需属性名的字符串。使用这个语法，我们可以访问前面代码中提到的 image 对象的属性：

```
image["width"]  
image['height']
```

关联数组是一种强大的数据类型，对于许多程序设计技术来说，它们都是非常有用的。我们将在第八章中学习更多的对象传统用法和关联数组用法。

3.5.1 创建对象

在第八章中我们将看到，对象是通过调用特殊的构造函数 (constructor function) 创建的。例如，下面的代码都创建了新对象：

```
var o = new Object();  
var now = new Date();  
var pattern = new RegExp("\\s\\sjava\\s", "i");
```

一旦你创建了属于自己的对象，那么就可以根据自己的意愿设计并使用它的属性了：

```
var point = new object();  
point.x = 2.3;  
point.y = -1.2;
```

3.5.2 对象直接量

ECMAScript v3 (JavaScript 1.2 实现) 定义了对象直接量的语法，使你能够创建对象并定义它的属性。对象直接量（也称为对象初始化程序）是由一个列表构成的，这个列表的元素是用冒号分隔的属性 / 值对，元素之间用逗号隔开了，整个列表包含在花括号之中。所以可以使用如下的方式来创建并初始化上面代码中的 point 对象：

```
var point = { x:2.3, y:-1.2 };
```

对象直接量也可以嵌套。例如：

```
var rectangle = { upperLeft: { x: 2, y: 2 },  
                  lowerRight: { x: 4, y: 4 }  
                };
```

最后要说明的是，对象直接量中的属性值不必是常量，它可以是任意的 JavaScript 表达式：

```
var square = { upperLeft: { x:point.x, y:point.y },  
               lowerRight: { x:(point.x + side), y:(point.y+side) } };
```

3.6 数组

数组（array）和对象一样是数值的集合。所不同的是，对象中的每个数值都有一个名字，而数组的每个数值有一个数字，或者说是下标。在 JavaScript 中，要获取数组中的某个值，可以使用数组名，在其后加上用方括号封闭起来的下标值即可。例如，如果一个数组名为 a，i 是一个非负整数，那么 a[i] 就是一个数组元素。因为数组的下标值是从 0 开始的，所以 a[2] 引用的是数组 a 的第三个元素。

数组可以存放任何一种 JavaScript 数据类型，包括对其他数组、对象或函数的引用。例如：

```
document.images[1].width
```

这行代码引用的是存储在数组第二个元素中的对象的 width 属性，该数组则存储在 document 对象的 images 属性中。

注意，这里所说的数组和 3.5 节中提到的关联数组不同。我们这里描述的常规数组以非负整数作为下标，而关联数组则用字符串作为下标。另外还要注意，JavaScript 并不支持多维数组，不过它的数组元素还可以是数组。最后要说的是，由于 JavaScript 是一种无类型语言，因此数组元素不必具有相同的类型（如在有类型语言 Java 中那样）。我们将在第九章中学习更多有关数组的内容。

3.6.1 数组的创建

可以使用构造函数 `Array()` 来创建数组。数组一旦被创建，就可以轻松地给数组的任何元素赋值：

```
var a = new Array();
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] = { x:1, y:3 };
```

通过把数组元素传递给 `Array()` 构造函数可以初始化数组，因此，前面创建数组和初始化代码可以写作：

```
var a = new Array(1.2, "JavaScript", true, { x:1, y:3 });
```

如果只给 `Array()` 构造函数传递一个参数，那么该参数指定的是数组的长度。因此：

```
var a = new Array(10);
```

创建的是具有 10 个未定义元素的新数组。

3.6.2 数组直接量

ECMAScript v3 (JavaScript 1.2 实现) 定义了创建并初始化数组的直接量语法。数组直接量 (或数组初始化程序) 是一个封闭在方括号中的序列，序列中的元素由逗号分隔。括号内的值将被依次赋给数组元素，下标值从 0 开始 (注 4)。例如，在 JavaScript 1.2 中，上面数组的创建和初始化都可以使用如下代码实现：

```
var a = [1.2, "JavaScript", true, { x:1, y:3 }];
```

与对象直接量一样，数组直接量也可以被嵌套：

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

注 4: Netscape 公司的 JavaScript 1.2 实现中有一个 bug: 当用一个数字指定数组的直接量，且该数字作为它唯一的元素时，该数字实际指定的是数组的长度，而不是第一个元素的值。尽管这一行为映射了 `Array()` 构造函数的行为，但在这个环境中显然不合适。

而且，与对象直接量相同，数组直接量中的元素不必仅限于常量，它可以是任意的表达式：

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

数组直接量中还可以存放未定义的元素，只要在逗号之间省去该元素的值就可以了。例如，下面的数组存放了5个元素，其中有3个是未定义的：

```
var sparseArray = [, , , 5];
```

3.7 null

JavaScript的关键字 `null` 是一个特殊的值，它表示“无值”。`null` 常常被看作对象类型的一个特殊值，即代表“无对象”的值。`null` 是个独一无二的值，有别于其他所有的值。如果一个变量的值为 `null`，那么你就会知道它的值不是有效的对象、数组、数字、字符串和布尔值（注5）。

3.8 undefined

还有一种特殊值 JavaScript 会偶尔一用，它就是值 `undefined`。在你使用了一个并未声明的变量时，或者使用了已经声明但还没有赋值的变量时，又或者使用了一个并不存在的对象属性时，返回的就是这个值。

虽然 `undefined` 和 `null` 值不同，但是 `==` 运算符却将两者看作相等。看如下的表达式：

```
my.prop == null
```

如果属性 `my.prop` 并不存在，或者它存在但是值为 `null`，那么这个比较表达式的值为 `true`。由于 `null` 和 `undefined` 都表明缺少值，所以这种相等性正是我们想要

注5： C 和 C++ 程序员应当注意，JavaScript 中的 `null` 和 0 并不相同。在某些情况下，`null` 可以转化为 0，但它并不等于 0。

的。但是，如果你必须区分 `null` 和 `undefined`，可以使用 `===` 运算符或 `typeof` 运算符（详见第五章）。

和 `null` 不同，`undefined` 不是 JavaScript 的保留字。ECMAScript v3 标准规定了名为 `undefined` 的全局变量，它的初始值是 `undefined`。因此，在符合 ECMAScript v3 的 JavaScript 实现中，可以把 `undefined` 作为关键字处理，只要不给该变量赋值即可。

如果无法确认自己使用的 JavaScript 实现有变量 `undefined`，只需要自己声明一个即可：

```
var undefined;
```

只声明这个变量，并不初始化它，就可以确保它的值为 `undefined`。`void` 运算符（参阅第五章）提供了另一种获取 `undefined` 值的方法。

3.9 Date 对象

前面几节描述了 JavaScript 支持的所有基本数据类型。日期和时间值并不属于这些基本类型。但是 JavaScript 提供了一种表示日期和时间的对象类，可以用它来操作这种类型的数据。在 JavaScript 中，可以用运算符 `new` 和构造函数 `Date()` 来创建一个 `Date` 对象（运算符 `new` 将在第五章中介绍，在第八章中我们将学习更多的有关创建对象的内容）：

```
var now = new Date(); // 创建存放当前日期和时间的对象。
// 创建表示圣诞节的 Date 对象。
// 注意，月份从 0 开始计数，所以十二月表示为 11。
var xmas = new Date(2000, 11, 25);
```

使用 `Date` 对象的方法，可以得到并设置日期和时间的值，而且还可以将 `Date` 对象转换成一个字符串（既可以使用本地时间也可以使用 GMT 时间）。例如：

```
xmas.setFullYear(xmas.getFullYear() + 1); // 把日期改为下一个圣诞节。
var weekday = xmas.getDay(); // 2001 年的圣诞节是星期二。
document.write("Today is: " + now.toLocaleString()); // 当前的日期/时间。
```

另外，`Date` 对象还定义了函数（不是方法，因为它们不是通过 `Date` 对象调用的），

该函数可以把一个由字符串或数字表示的日期值转换成内部的毫秒表示, 这对某些日期计算非常有用。

在本书的核心参考手册中可以找到有关 Date 对象和它的方法的全部文档。

3.10 正则表达式

正则表达式为描述文本模式提供了丰富、强大的语法, 它们常用于模式匹配和查找并替换的操作。JavaScript 采用了 Perl 程序设计语言的语法表示正则表达式。JavaScript 1.2 首次添加了对正则表达式的支持, ECMAScript v3 对此进行了标准化和扩展。

在 JavaScript 中, 正则表达式是由 RegExp 对象表示的, 可以使用 RegExp() 构造函数来创建它。和 Date 对象一样, RegExp 对象并不属于 JavaScript 的基本数据类型, 它只不过是所有 JavaScript 实现都支持的特殊对象类型。

但是, 与 Date 对象不同的是, RegExp 对象有一个直接量语法, 可以被直接编码到 JavaScript 1.2 程序中。一对斜线之间的文本就构成了一个正则表达式直接量。在斜线对中的第二条斜线之后还可以跟有一个或多个字母, 它们改变了模式的含义。例如:

```
/^HTML/  
/[1-9][0-9]*/  
/^\bjavascript\b/i
```

正则表达式的语法是相当复杂的, 在第十章中我们将详细介绍它。现在, 你只需要知道在 JavaScript 代码中正则表达式直接量是什么样的。

3.11 Error 对象

ECMAScript v3 定义了大量表示错误的类。当发生运行时错误时, JavaScript 解释器会抛出某个类的对象 (参阅第六章有关 throw 和 try 语句的讨论)。每个 Error 对象具有一个 message 属性, 它存放的是 JavaScript 实现特定的错误消息。预定义的

错误对象的类型有 `Error`、`EvalError`、`RangeError`、`ReferenceError`、`SyntaxError`、`TypeError` 和 `URIError`。在本书的核心参考手册中可以找到这些类的详细信息。

3.12 基本数据类型的包装对象

当我们在本章的前面讨论字符串时，曾提到数据类型的一个奇怪特性，那就是使用对象的表示法来操作字符串。例如，下面是一个对字符串的典型操作：

```
var s = "These are the times that try people's souls.";
var last_word = s.substring(s.lastIndexOf(" ") + 1, s.length);
```

如果对 JavaScript 不了解，那么就可能以为 `s` 是一个对象，你正在调用该方法并读取它的属性值。

到底怎么回事？难道字符串是对象吗？还是字符串是基本的数据类型？运算符 `typeof`（参阅第五章）可以断然地告诉我们，字符串的数据类型是“string”，而对象的类型则是“object”，两者是不同的。那么，为什么字符串的操作采用对象的表示法呢？

事实上，三个关键的基本数据类型都有一个相应的对象类。简而言之，就是 JavaScript 不仅支持数字、字符串和布尔值这些数据类型，还支持 `Number`、`String` 和 `Boolean` 类。这些类是那些基本数据类型的包装。这些包装（wrapper）不仅具有和基本类型一样的值，还定义了用来运算数据的属性和方法。

JavaScript 可以很灵活地将一种类型的值转换为另一种类型。当我们在对象环境中使用字符串时（即试图访问这个字符串的属性或方法时），JavaScript 会为这个字符串值内部地创建一个 `String` 包装对象。`String` 对象就代替了原始的字符串值。由于对象具有了属性和方法，因此就能在对象语境中使用简单的值。当然，对其他的基本类型和它们相应的包装对象来说也是同样的，只是我们使用其他类型时，不像使用字符串那样常用对象环境。

当我们在对象环境中使用字符串时，要注意被创建的 `String` 对象只不过是瞬时存在的，它使得我们可以访问属性或方法，此后就没有用了，所以系统会将它丢弃的。假设 `s` 是一个字符串，我们可以使用如下的代码来获取字符串的长度：

```
var len = s.length;
```

在这个例子中，`s` 保存了一个字符串，原始的字符串值是不会自行改变的。一个新的 `String` 对象被创建了，它是瞬时存在的，使我们能够访问 `length` 属性，之后它就被丢弃了，原始的值 `s` 并不会改变。如果你认为这个模式既精致又古怪，那么你是对的。但是不必担心，因为在 JavaScript 内可以相当有效地完成瞬态对象的转换。

如果你想在程序中显式地使用 `String` 对象，那么就必须创建一个非瞬态的对象，即不能自动地被系统丢弃的对象。`String` 对象的创建和其他对象一样，都使用了 `new` 运算符。例如：

```
var s = 'hello world';           // 原始的字符串值
var S = new String("Hello World"); // String 对象
```

创建了 `String` 对象 `S` 之后，我们可以用它做些什么呢？其实，我们可以用原始的字符串值做什么，就可以用 `String` 对象做什么。如果我们使用 `typeof` 运算符，它会告诉我们 `S` 实际上是一个对象，而不是一个字符串值，但是除此之外，我们发现根本不能区别原始字符串和 `String` 对象（注6）。我们已经看到过，无论何时，只要有必要字符串都会被自动地转换为 `String` 对象。结果证明这种转换也是对的。当我们在一处需要原始字符串值的地方使用了 `String` 对象时，JavaScript 也会自动将 `String` 对象转换为一个字符串。因此，如果我们使用 `String` 对象时用了“+”操作符，那么就会有一个瞬态的基本字符串值被创建，以便执行字符串的连接操作：

```
msg = S + '!' ;
```

记住，我们在本章中讨论的有关字符串值和 `String` 对象的所有内容都适用于数字、布尔值及其相应的 `Number` 对象及 `Boolean` 对象。要了解更多的有关这些类的内容，可以查阅本书核心参考手册部分中有关这些类的相关内容。在第十一章中，我们将介绍更多有关 JavaScript 中基本类型 / 对象的二元性以及自动数据转换的问题。

注6： 注意，`eval()` 方法处理字符串值和 `String` 对象的方式不同。如果你不小心传递给 `eval()` 方法一个 `String` 对象而不是原始的字符串值，它的行为就和你预期的不同。

第四章

变量

变量 (variable) 是一个和数值相关的名字。我们说变量“存储”了或“包含”了那个值。有了变量, 你就可以在程序中存储和运算数据了。例如, 下面的一行JavaScript代码将数值 2 赋给了一个名为 `i` 的变量:

```
i = 2;
```

下面的代码将 3 加到 `i` 上, 然后把结果赋给了一个新的变量 `sum`:

```
var sum = i + 3;
```

这两行代码说明了一切你需要了解的有关变量的内容。但是, 要全面理解在JavaScript中变量是如何工作的, 你还需要掌握更多的概念。遗憾的是, 要解释这些概念不是几行代码可以作到的。本章的余下部分将解释变量的类型规则、声明、作用域、内容和解析方法, 另外还研究了无用存储单元的收集以及变量/属性二元性的问题 (注1)。

注1: 这些都是技巧性很强的概念, 要完全理解本章介绍的内容, 还要阅读本书后面相关章节的内容。如果你是个程序设计的新手, 那么可以先阅读本章的前两个小节, 然后阅读第五、六、七章, 最后再转回阅读本章剩余的小节。

4.1 变量的类型

JavaScript 和 Java 与 C 这样的语言之间存在一个重要的差别，那就是 JavaScript 是无类型 (untyped) 的。这就意味着 JavaScript 的变量可以存放任何类型的值，而 Java 和 C 的变量都只能存放它所声明了的特定类型的数据。例如，在 JavaScript 中，可以先把一个数值赋给一个变量，然后再把一个字符串赋给它，这是完全合法的：

```
i = 10;
i = "ten";
```

在 C、C++ 和 Java 中，上边的代码都是不合法的。

有一个特性是与 JavaScript 缺少类型规则相关，即在必要时 JavaScript 可以快速、自动地将一种类型的值转换成另外一种类型。例如，如果想把一个数值连接到一个字符串上，那么 JavaScript 会自动把这个数值转换成相应的字符串，这样就可以将它连接到原来的字符串之后了。我们将在第十一章中介绍更多有关数据类型转换的内容。

由于没有类型规则，所以 JavaScript 显然是一种比较简单的语言。像 C++ 和 Java 这样的有类型语言，它们的优点在于编程时要求使用严格的程序设计规则，以便使编写、维护和重用那些较长、较复杂的程序变得更加容易。因为 JavaScript 程序多是短小的脚本，所以并不需要那么精确，而且我们从简单的语法中也获益匪浅。

4.2 变量的声明

在 JavaScript 程序中，在使用一个变量之前，必须先声明 (declare) 它 (注2)。变量是使用关键字 `var` 声明的，如下所示：

```
var i;
var sum;
```

也可以使用一个 `var` 关键字声明多个变量：

```
var i, sum;
```

注2：如果你没有显式地声明一个变量，JavaScript 会替你隐式地声明它。

而且还可以将变量声明和变量初始化绑定在一起:

```
var message = "hello";  
var i = 0, j = 0, k = 0;
```

如果没有用 var 语句给一个变量指定初始值, 那么虽然这个变量被声明了, 但是在给它存入一个值之前, 它的初始值就是 undefined。

注意, var 语句还可以作为 for 循环和 for/in 循环 (第六章将介绍这些语句) 的一部分, 这样就使循环变量的声明成为了循环语法自身的一部分, 非常简洁。例如:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");  
for(var i = 0, i=10; i < 10; i++,j--) document.write(i*j, "<br>");  
for(var i in o) document.write(i, '<br>');
```

由 var 声明的变量是永久性的, 也就是说, 用 delete 运算符来删除这些变量将会引发错误 (delete 运算符将在第五章中介绍)。

4.2.1 重复的声明和遗漏的声明

使用 var 语句多次声明同一个变量不仅是合法的, 而且也不会造成任何错误。如果重复的声明有一个初始值, 那么它担当的不过是一个赋值语句的角色。

如果尝试读一个未声明的变量的值, JavaScript 会生成一个错误。如果尝试给一个未用 var 声明的变量赋值, JavaScript 会隐式声明该变量。但是要注意, 隐式声明的变量总是被创建为全局变量, 即使该变量只在一个函数体内使用。局部变量是只在一个函数中使用, 要防止在创建局部变量时创建全局变量 (或采用已有的全局变量), 就必须在函数体内部使用 var 语句。无论是全局变量还是局部变量, 最好都使用 var 语句创建 (关于全局变量和局部变量之间的区别将在下一节中详细介绍)。

4.3 变量的作用域

一个变量的作用域 (scope) 是程序中定义这个变量的区域。全局 (global) 变量的作用域是全局性的, 即在 JavaScript 代码中, 它处处都有定义。而在函数之内声明的变量, 就只在函数体内部有定义。它们是局部 (local) 变量, 作用域是局部性的。函数的参数也是局部变量, 它们只在函数体内部有定义。

在函数体内部，局部变量的优先级比同名的全局变量高。如果你给一个局部变量或函数的参数声明的名字与某个全局变量的名字相同，那么你就有效地隐藏了这个全局变量。例如，下面的代码将输出单词“local”：

```
var scope = "global";           // 声明一个全局变量
function checkscope() {
    var scope = "local";        // 声明一个同名的局部变量
    document.write(scope);      // 使用的是局部变量，而不是全局变量
}
checkscope();                   // 输出“local”
```

虽然在全局作用域中编写代码时可以不使用var语句，但是在声明局部变量时，一定要使用var语句。下面的代码说明了如果不这样做，将会发生的情况：

```
scope = "global";               // 即使没有var，仍然声明一个全局变量
function checkscope() {
    scope = 'local';            // 我们改变了全局变量
    document.write(scope);      // 使用的是全局变量
    myscope = 'local';          // 该语句隐式声明了新的全局变量
    document.write(myscope);    // 使用的是新的全局变量
}
checkscope();                   // 输出“locallocal”
document.write(scope);          // 输出“local”
document.write(myscope);        // 输出“local”
```

一般说来，函数并不知道全局作用域中定义了什么变量，也不知道那些变量是做什么用的。因此，如果函数使用的是全局变量，而不是局部变量，那么就会有改变程序的其他部分所使用的值的危险。幸运的是，这种问题是很容易避免的，只需要在声明所有变量时都使用var语句即可。

在JavaScript 1.2（和ECMAScript v3）中，函数定义是可以嵌套的。由于每个函数都有它自己的局部作用域，所以有可能出现几个局部作用域的嵌套层。例如：

```
var scope = "global scope";     // 一个全局变量
function checkscope() {
    var scope = "local scope";   // 一个局部变量
    function nested() {
        var scope = "nested scope"; // 局部变量的嵌套作用域
        document.write(scope);     // 输出“nested scope”
    }
    nested();
}
checkscope();
```

4.3.1 没有块级作用域

注意，和 C、C++ 以及 Java 不同，JavaScript 没有块级作用域。函数中声明的所有变量，无论是在哪里声明的，在整个函数中它们都是有定义的。在下面的代码中，变量 i、j 和 k 的作用域是相同的，它们三个在整个函数体中都有定义。如果这段代码是用 C、C++ 或 Java 编写的，情形就不是这样了：

```
function test(o) {  
    var i = 0; // i 在整个函数中有定义  
    if (typeof o == "object") {  
        var j = 0; // j 到处都有定义，不仅限于这个代码块  
        for (var k = 0; k < 10; k++) { // k 到处都有定义，不仅限于该循环  
            document.write(k);  
        }  
        document.write(k); // k 仍旧有定义，输出 10  
    }  
    document.write(i); // i 仍旧有定义，但没有被初始化  
}
```

这一规则（即函数中声明的所有变量在整个函数中都有定义）可以产生惊人的结果。下面的代码说明了这一点：

```
var scope = "global";  
function f() {  
    alert(scope); // 显示 "undefined"，而不是 "global"  
    var scope = "local"; // 变量在此处被初始化，但到处都有定义  
    alert(scope); // 显示 "local"  
}  
f();
```

你可能认为对 alert() 的第一次调用会显示出 "global"，因为声明局部变量的 var 语句还没有被执行。但是，由于这个作用域规则的限制，输出的并不是 "global"。局部变量在整个函数体内都是有定义的，这就意味着在整个函数体中都隐藏了同名的全局变量。虽然局部变量在整个函数体中都是有定义的，但是在执行 var 语句之前，它是不会被初始化的，因此上面的例子中，函数 f 和下面的函数等价：

```
function f() {  
    var scope; // 局部变量在函数开头声明  
    alert(scope); // 此处该变量有定义，但值仍为 "undefined"  
    scope = "local"; // 现在我们初始化该变量，并给它赋值  
    alert(scope); // 此处该变量具有值  
}
```


这个例子说明了为什么将所有的变量声明集中起来放置在函数的开头是一个好的编程习惯。

4.3.2 未定义的变量和未赋值的变量

前面一节的例子说明了JavaScript程序设计中的一细节，那就是有两种不同类型的未定义变量。一种未定义的变量是从没有被声明过的。尝试读这种未经声明的变量会引起运行时的错误。未被声明的变量(Undeclared Variable)就是未定义的，因为这样的变量根本不存在。前面一节讲过，给未声明的变量赋值并不会引起错误，相反，程序会在全局作用域中隐式地声明它。

第二种未定义的变量是已经被声明了但是永远都不会被赋值的变量。如果要读这样的变量的值，将会得到一个默认值，即undefined。也许将这种未定义的变量称为“未赋值的变量”(unassigned)更加有用一些，这样就可以和那些未声明的、甚至根本不存在的未定义变量区别开了。

下面的代码段说明了真正的未定义变量和只是未赋值的变量之间的区别：

```
var x;           // 声明一个未赋值的变量，它的值为undefined。  
alert(u);        // 使用未声明的变量将引发错误。  
u = 3;          // 给未声明的变量赋值，将创建该变量。
```

4.4 基本类型和引用类型

我们要介绍的下一个主题是变量的内容。我们常说变量“具有”或“存放”了值，但是它存放的是什么呢？这一问题看来简单，要回答它还要再来看一看JavaScript支持的数据类型。我们可以将数据类型分为两组，即基本类型和引用类型。数值、布尔值、null和未定义的值属于基本类型。对象、数组和函数属于引用类型。

基本类型在内存中具有固定的大小。例如，一个数值在内存中占八个字节，而一个布尔值使用一位就可以表示了。数值类型是基本类型中最大的数据类型了。如果每个JavaScript变量都占有八个字节的内存，那么它们就可以直接存放任何基本类型的值了（注3）。

注3： 这种解释是非常简单的，实际的JavaScript实现不会像描述的这样简单。

但是引用类型则不同。例如，对象可以具有任意的长度，它并没有固定的大小。对数组来说也是这样，因为一个数组可以具有任意多个元素。同样的，函数可以包含任意数量的 JavaScript 代码。由于这些类型没有固定的大小，所以不能将它们的值直接存储在与每个变量相关的八字节内存中。相反，变量存储的是对这个值的引用。通常引用的形式是指针或者内存地址。虽然引用不是数据本身，但是它告诉变量在哪里可以找到这个值。

基本类型和引用类型之间的差别是很重要的，因为它们的行为是不同的。考虑下面的代码，它使用了数值（一个基本类型）：

```
var a = 3.14;      // 声明并初始化一个变量
var b = a;         // 把该变量的值复制到一个新变量
a = 4;            // 修改原始变量的值
alert(b);         // 显示 3.14：副本没有改变
```

这段代码并没有什么惊人之处。现在，考虑一下，如果我们对这段代码做了轻微的改动，使用数组（一个引用类型）代替数值，那么会出现什么情况：

```
var a = [1,2,3];   // 初始化一个引用数组的变量
var b = a;         // 把该引用复制到一个新变量
a[0] = 99;         // 用原始引用修改数组
alert(b);          // 用新的引用显示改变后的数组 [99,2,3]
```

如果结果并不让你感到意外，说明你对基本类型和引用类型之间的差别已经非常熟悉了。如果它令你感到吃惊，那么仔细看一下第二行。注意，在这个语句中，赋给 `b` 的只是对数组值的一个引用，而不是数组本身，数组已经在语句中被赋值了。执行过第二行代码之后，我们仍旧只有一个数组对象，只不过我们有了两个对它的引用。

如果基本类型和引用类型之间的差别对你来说是新内容的话，那么努力记住它吧。变量保存了基本类型的实际值，但是对引用类型的值却只保存对它的引用。有关基本类型和引用类型行为的不同之处将在 11.2 节中详细介绍。

你可能已经注意到了，我并没有指明字符串在 JavaScript 中是基本类型还是引用类型。字符串是一个特例。因为它具有可变的大小，所以显然它不能被直接存储在具有固定大小的变量中。出于效率的原因，我们希望 JavaScript 只复制对字符串的引用而不是字符串的内容。但另一方面，字符串在许多方面都和基本类型的表现相似。而字符串是不可变的这一事实（即没有办法改变一个字符串值的内容）使得字符串

是基本类型还是引用类型的问题更加令人费解。这意味着我们不能构造上面那样的例子来说明复制的是对数组的引用。其实，无论你将字符串看作是行为与基本类型相似的不可变引用类型，还是将它看作使用引用类型的内部功能实现的基本类型，结果都是一样的。

4.5 无用存储单元的收集

因为引用类型没有固定的大小，所以某些引用类型可能非常大。我们已经讨论过，变量并不能直接保存引用的值，这些值被存储在某个位置，变量保存的只是对那个位置的引用。现在，我们暂时把重点放在值的实际存储上。

由于字符串、对象和数组没有固定大小，所以当它们的大小已知时，才能对它们进行动态的存储分配。JavaScript 程序每次创建字符串、数组或对象时，解释器都必须分配内存来存储那个实体。只要像这样动态地分配了内存，最终都要释放这些内存以便它们能够被再用，否则，JavaScript 的解释器将会消耗完系统中所有可用的内存，造成系统崩溃。

在 C 和 C++ 这样的语言中，内存必须手动地释放。要跟踪已创建的所有对象，并且当不再需要这些对象时把它们销毁（释放内存空间），这都是程序设计者的责任。这是一项棘手的任务，常常是产生 bug 的根源。

JavaScript 则不要求手动地去配内存，它使用一种称为无用存储单元收集（garbage collection）的方法。JavaScript 的解释器可以检测到何时程序不再使用一个对象了。当它确定了一个对象是无用的时候（例如，程序中使用的变量再也无法引用这个对象了），它就知道不再需要这个对象，可以把它所占用的内存释放掉了。例如，考虑下面的几行代码：

```
var s = 'hello';           // 给一个字符串分配内存
var u = s.toUpperCase();   // 创建一个新字符串
s = u;                     // 覆盖对原始字符串的引用
```

运行了这些代码之后，就不能再获得原始的字符串“hello”，因为程序中没有变量再引用它了。系统检测到这一事实后，就会释放该字符串的存储空间以便这些空间可以被再利用。

无用存储单元的收集是自动进行的,对程序员来说是不可见的。你可以创建任何想要的无用对象,之后,系统会将它们都清除掉。对于无用存储单元的收集,惟一需要你做的就是相信它一定会起作用,而且不要问所有的旧对象都到哪里去了。但是,那些对此感兴趣的人,可以参阅第11.3节,该节进一步详细介绍了JavaScript的无用存储单元的收集的方法。

4.6 作为属性的变量

现在,你可能已经注意到了,JavaScript的变量和对象的属性之间有很多的相似点。例如,它们采用相同的赋值方式,而且在JavaScript表达式中的用法也相同,等等。那么变量*i*和对象*o*的属性*i*之间有什么根本的区别吗?答案是没有。在JavaScript中,变量基本上和对象的属性是一样的。

4.6.1 全局对象

当JavaScript的解释器开始运行时,它首先要做的事情之一就是在执行任何JavaScript代码之前,创建一个全局对象(global object)。这个对象的属性就是JavaScript程序的全局变量。当声明一个JavaScript的全局变量时,实际上所做的是定义了那个全局对象的一个属性。

此外,JavaScript解释器还会用预定义的值和函数来初始化全局对象的许多属性。例如,属性Infinity、parseInt和Math分别引用了数值infinity、预定义的函数parseInt()和预定义的对象Math。在本书的核心参考手册中你可以找到这些全局值。

在程序的顶层代码中(例如,不属于函数的JavaScript代码),你可以使用JavaScript的关键字this来引用这个全局对象。在函数内部,this则有别的用途,我们将在第七章介绍这一问题。

在客户端JavaScript中,Window对象代表浏览窗口,它是包含在该窗口中的所有JavaScript代码的全局对象。这个全局Window对象具有自我引用的window属性,它代替了this属性,可以用来引用这个全局对象。Window对象定义了全局的核心

属性，如 `parseInt` 和 `Math`，此外它还定义了全局的客户端属性，如 `navigator` 和 `screen`。

4.6.2 局部变量：调用对象

如果全局变量是特殊的全局对象的属性，那么局部变量又是什么呢？它们也是一个对象的属性，这个对象被称为调用对象（call object）。虽然调用对象的生命周期比全局对象的短，但是它们的用途是相同的。在执行一个函数时，函数的参数和局部变量是作为调用对象的属性而存储的。用一个完全独立的对象来存储局部变量使 JavaScript 可以防止局部变量覆盖同名的全局变量的值。

4.6.3 JavaScript 的执行环境

JavaScript 的解释器每次开始执行一个函数时，都会为那个函数创建一个执行环境（execution context）。显然，一个执行环境就是所有 JavaScript 代码段执行时所在的环境。这个环境的一个重要部分是定义变量的对象。因此，运行不属于任何函数的 JavaScript 代码的环境使用的就是全局对象。所有 JavaScript 函数都运行在自己独有的执行环境中，而且具有自己的调用对象，在调用对象中定义了局部变量。

要注意的一个有趣的地方是，JavaScript 的实现允许有多个全局执行环境，每个执行环境有不同的全局对象（但是，在这种情况下，每个全局对象就不完全是全局的了）（注4）。一个显而易见的例子是，客户端 JavaScript 的每个独立的浏览窗口或同一窗口的不同框架中都定义了独立的全局执行环境。每个框架或窗口中的客户端 JavaScript 代码都运行在自己的执行环境中，具有自己的全局对象。但是，这些独立的客户端全局对象具有将与其他对象彼此连接起来的属性。因此，一个框架中的 JavaScript 代码可以使用表达式 `parent.frames[1]` 来引用另一个框架中的 JavaScript 代码，在第二个框架中的代码可以使用表达式 `parent.frames[0].x` 来引用第一个框架中的全局变量 `x`。

你不必立刻完全理解独立的窗口和框架执行环境是如何链接在一起的。当我们在第十二章讨论 JavaScript 和 Web 浏览器的集成时会详细介绍这一主题。现在你应该理

注4：这只是题外话，如果你对此不感兴趣，可以继续阅读下一节的内容。

解的是 JavaScript 有很大的灵活性，一个 JavaScript 解释器可以在不同的全局执行环境中运行脚本，而且这些环境之间并不是完全脱节的，它们彼此可以相互引用。

最后的一点需要额外说明一下。如果一个执行环境中的 JavaScript 代码可以读写另一个执行环境中定义的属性，并且执行它的函数，那么复杂度就上升了一层，我们需要考虑安全性的问题了。以客户端 JavaScript 为例。假设浏览窗口 A 正在运行一个脚本或者它含有来自局域网的信息，而窗口 B 则正在运行来自 Internet 的某个站点的脚本。一般说来，我们不希望窗口 B 中的代码能够访问窗口 A 的属性。如果允许这样做，那么它就有可能读取公司的机密信息并窃取这些信息。所以，为了安全地运行 JavaScript 代码，必须有一定的安全机制。当从一个执行环境访问另一个执行环境时，如果这种访问是不允许的，那么就禁止执行它。我们将在第二十一章中继续讨论这一问题。

4.7 深入理解变量作用域

在我们初次讨论变量作用域这个概念时，我们只是基于 JavaScript 代码的词法结构来定义它的，即全局变量具有全局的作用域，而函数中声明的变量具有局部的作用域。如果一个函数定义嵌套在另一个函数中，那么在嵌套的函数中声明的变量就具有嵌套的局部作用域。既然我们知道全局变量是全局对象的属性，而局部变量是一个特殊的调用对象的属性，那么我们就可以再次关注一下变量作用域的表达法，对它进行再定义。有关作用域的新描述给理解多环境下的变量提供了一种有用的方法，它为 JavaScript 的工作过程提供了一种强大的、新的理解。

每个 JavaScript 执行环境都有一个和它关联在一起的作用域链 (scope chain)。这个作用域链是一个对象列表或对象链。当 JavaScript 代码需要查询变量 x 的值时（一个称为变量名解析 (variable name resolution) 的过程），它就开始查看该链的第一个对象。如果那个对象有一个名为 x 的属性，那么就采用那个属性的值。如果第一个对象没有名为 x 的属性，JavaScript 就会继续查询链中的第二个对象。如果第二个对象仍然没有名为 x 的属性。那么就继续查询下一个对象，以此类推。

在 JavaScript 的顶层代码中（例如，不属于任何函数定义的代码），作用域链只由一个对象构成，那就是全局对象。所有的变量都是在这个对象中查询的。如果一个变量并不存在，那么这个变量的值就是未定义的。在一个（非嵌套的）函数中，作用

域链是由两个对象构成的。第一个是函数的调用对象，第二个就是全局对象。当函数引用一个变量时，首先检查的是调用对象（局部作用域），其次才检查全局对象（全局作用域）。在一个嵌套函数的作用域链中可以有三个或更多的对象。图 4-1 说明了一个函数的作用域链中查找一个变量名的过程。

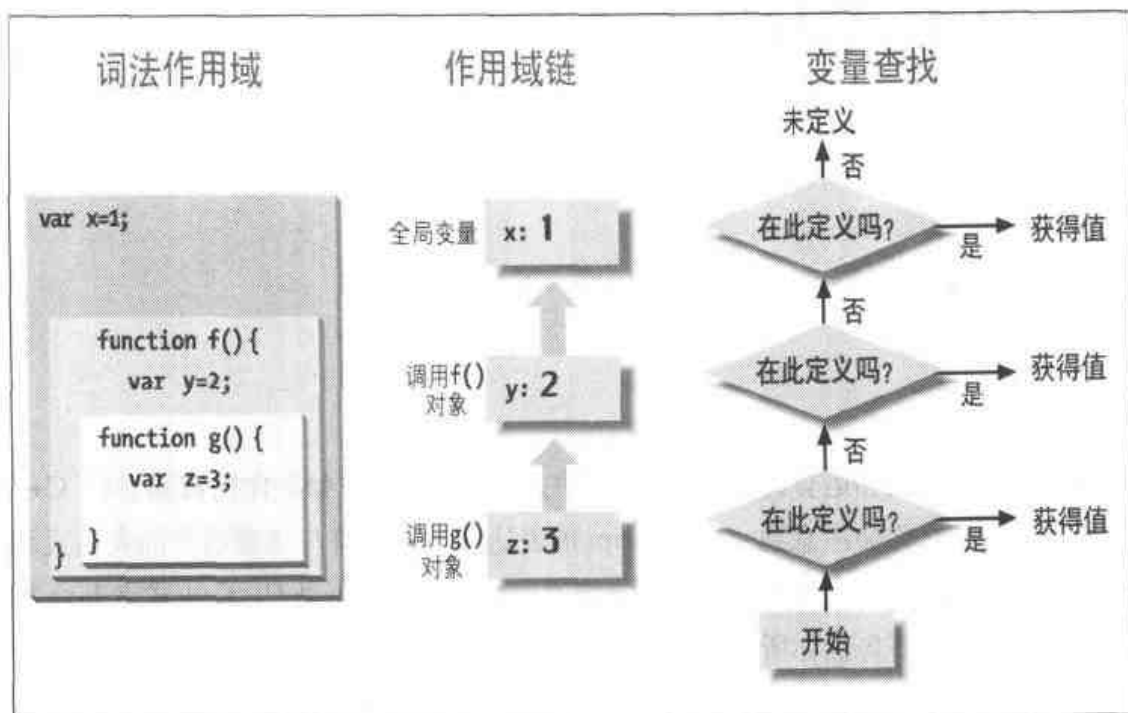


图 4-1: 作用域链和变量解析

第五章

表达式和运算符

本章阐述了表达式和运算符在JavaScript中是如何作用的。如果你比较熟悉C、C++或者Java，那么你会注意到，JavaScript的表达式和运算符与这些语言的表达式和运算符是相似的，你可以快速地浏览一下这一章。如果你不是C、C++或者Java程序员，那么本章将告诉你所有必要的JavaScript表达式和运算符的知识。

5.1 表达式

表达式 (expression) 是JavaScript的一个“短语”，JavaScript的解释器对可以计算它，从而生成一个值。最简单的表达式是直接量或者变量名，如下所示：

```
1.7           // 一个数字直接量
"JavaScript is fun!" // 一个字符串直接量
true          // 一个布尔直接量
null          // 直接量 null
/java/        // 一个正则表达式直接量
{ x:2, y:2 }   // 一个对象直接量
[2,3,5,7,11,13,17,19] // 一个数组直接量
function(x){return x*x;} // 一个函数直接量
i             // 变量 i
sum           // 变量 sum
```

直接量表达式的值就是这个直接量本身。变量表达式的值则是该变量所存放或引用的值。

上面的表达式并不是特别有趣。我们还可以通过合并简单的表达式来创建较为复杂（且有趣）的表达式。例如，我们知道 1.7 是表达式，i 也是表达式，接下来的代码所表示的同样是表达式：

```
i + 1.7
```

这个表达式的值是两个简单表达式的和。在这个例子中，“+”是一个运算符（operator），用于将两个简单的表达式合并起来以组成一个复杂的表达式。“-”也是一个运算符，它使用减法将表达式合并在一起。例如：

```
(i + 1.7) - sum
```

这个表达式使用了减法运算符，它从前面的表达式 i+1.7 中减去了变量 sum 的值。除了“+”和“-”之外，JavaScript 还支持许多其他的运算符，下一节中将详细介绍这些运算符。

5.2 运算符概述

如果你是一个 C、C++ 或者 Java 程序员，那么你应该熟悉大部分的 JavaScript 运算符。表 5-1 总结了这些运算符，你可以将此表用做参考。注意，大部分运算符是用标点符号表示的，诸如“+”和“=”，但是有些运算符则是由关键字表示的，如 delete 和 instanceof。关键字运算符和用标点符号表示的运算符一样，都是正则运算符，只不过它们是用更具有可读性、而语法却不那么简洁的方式表达的。

在这个表中，P 列给出了运算符的优先级，A 列给出了运算符的结合性。结合性可以是 L（从左到右），也可以是 R（从右到左）。如果你还不了解优先级和结合性，该表之后的部分会解释这些概念。运算符的说明则放在这个讨论之后。

表 5-1: JavaScript 的运算符

| P | A | 运算符 | 运算数类型 | 所执行的操作 |
|----|---|-----|--------|--------|
| 15 | L | . | 对象，标识符 | 属性存取 |
| | L | [] | 数组，整数 | 数组下标 |
| | L | () | 函数，参数 | 函数调用 |
| | R | new | 构造函数调用 | 创建新对象 |

表 5-1: JavaScript 的运算符 (续)

| P | A | 运算符 | 运算数类型 | 所执行的操作 |
|----|---|------------|----------|------------------|
| 14 | R | ++ | lvalue | 先递增或后递增运算 (一元的) |
| | R | -- | lvalue | 先递减或后递减运算 (一元的) |
| | R | - | 数字 | 一元减法 (负) |
| | R | + | 数字 | 一元加法 |
| | R | ~ | 整数 | 按位取补码的操作 (一元的) |
| | R | ! | 布尔值 | 取逻辑补码的操作 (一元的) |
| | R | delete | lvalue | 删除一个属性 (一元的) 的定义 |
| | R | typeof | 任意 | 返回数据类型 (一元的) |
| | R | void | 任意 | 返回未定义的值 (一元的) |
| 13 | L | *, /, % | 数字 | 乘法、除法、取余运算 |
| 12 | L | +, - | 数字 | 加法、减法运算 |
| | L | + | 字符串 | 连接字符串 |
| 11 | L | << | 整数 | 左移 |
| | L | >> | 整数 | 带符号扩展的右移 |
| | L | >>> | 整数 | 带零扩展的右移 |
| 10 | L | <, <= | 数字或字符串 | 小于或小于等于 |
| | L | >, >= | 数字或字符串 | 大于或大于等于 |
| | L | instanceof | 对象, 构造函数 | 检查对象类型 |
| | L | in | 字符串, 对象 | 检查一个属性是否存在 |
| 9 | L | == | 任意 | 测试相等性 |
| | L | != | 任意 | 测试非相等性 |
| | L | === | 任意 | 测试等同性 |
| | L | !== | 任意 | 测试非等同性 |

表 5-1: JavaScript 的运算符 (续)

| P | A | 运算符 | 运算数类型 | 所执行的操作 |
|---|---|---|------------|----------------------|
| 8 | L | & | 整数 | 按位与操作 |
| 7 | L | ^ | 整数 | 按位异或操作 |
| 6 | L | | 整数 | 按位或操作 |
| 5 | L | && | 布尔值 | 逻辑与操作 |
| 4 | L | | 布尔值 | 逻辑或操作 |
| 3 | R | ?: | 布尔值、任意、任意 | (由三个运算数构成的)
条件运算符 |
| 2 | R | = | lvalue, 任意 | 赋值运算 |
| | R | *=, /=, %=, +=,
-=, <=<=, >>=,
>>>=, &=, ^=, = | lvalue, 任意 | 带操作的赋值运算 |
| 1 | L | | 任意 | 多重计算的操作 |

5.2.1 运算数的个数

可以根据运算符需要的运算数的个数对运算符进行分类。大多数 JavaScript 运算符 (像我们在前一节中看到的 “+” 运算符) 都是二元运算符 (binary operator), 它们把两个表达式合并成一个复杂的表达式。简而言之, 就是它有两个运算数。此外 JavaScript 还支持大量的一元运算符 (unary operator), 它能将一个表达式转换成另一个更复杂的表达式。在表达式 -3 中, 运算符 “-” 就是一元运算符, 它执行的操作是对运算数取反。JavaScript 还支持三元运算符 (ternary operator) “?:”, 它可以将三个表达式合并为一个复杂的表达式。

5.2.2 运算数的类型

在构造 JavaScript 的表达式时, 一定要注意传递给运算符的数据类型和返回的数据类型。各种运算符用来计算的运算数表达式要符合某种数据类型。例如, 对字符串不能进行乘法运算, 所以在 JavaScript 中表达式 “a” * “b” 是不合法的。但是要注意, 在可能的情况下, JavaScript 会把表达式转换为适当的类型, 因此表达式 “3” * “5”

是合法的，它的值是数字 15，而不是字符串 "15"。我们将在第 11.1 节中详细地讨论 JavaScript 的类型转换。

另外，某些运算符的行为根据运算数类型的不同而有所不同。最明显的例子就是运算符 "+"，它对数字运算数执行的是加法操作，但对字符串运算数执行的却是连接操作。而且，如果传递给它的是一个数字和一个字符串，那么它会把数字转换成一个字符串，然后将两个字符串连接起来。例如，"1"+0 产生的是字符串 "10"。

注意，赋值运算符和其他几个运算符期望自己左边的表达式的值是 lvalue 类型的。lvalue 是一个历史术语，它指的是能够合法地出现在一个赋值表达式左边的表达式。在 JavaScript 中，变量、对象的属性和数组的元素都是 lvalue 型的。ECMAScript 标准允许内部函数返回 lvalue 类型的值，但是并没有定义具有这种行为的内部函数。

最后要注意的是，运算符返回的数据类型不能总是和它的运算数类型相同。比较运算符（小于、等于和大于等）的运算数类型可以不同，但是在计算比较表达式的值时，返回的结果总是布尔值，从而说明了该比较的结果是真还是假。例如，当变量 a 的值确实比 3 小的时候，表达式 a<3 就返回 true。我们发现，由比较运算符返回的布尔值常常用于 if 语句、while 循环和 for 循环，这些语句依赖于包含比较运算符的表达式的结果来控制程序的执行。

5.2.3 运算符优先级

在表 5-1 中，P 列说明了每个运算符的优先级（precedence）。运算符的优先级控制执行操作的顺序。在 P 列中数字较大的运算符的优先级大于数字较小的运算符的优先级。

考虑下面的表达式：

```
w = x + y * z;
```

乘法运算符 * 比加法运算符 (+) 的优先级高，所以先执行乘法，再执行加法。此外，赋值运算符 (=) 的优先级最低，所以在右边的操作都被执行完之后才会执行赋值操作。

使用括号可以提高运算符的优先级。在上例中，如果必须要先执行加法，我们可以编写如下的代码：

```
w = (x + y) * z;
```

在实际应用中，如果你根本不清楚所使用的运算符的优先级，最简单的办法是使用括号来明确表明计算顺序。但是要记住一条重要规则：乘法和除法是优先于加法和减法执行的，赋值操作的优先级非常低，几乎总是最后才被执行。

5.2.4 运算符的结合性

在表 5-1 中，A 列说明了运算符的结合性（associativity）。值 L 表示结合性从左到右，值 R 表示结合性从右到左。一个运算符的结合性说明了优先级相等时执行操作的顺序。从左到右的结合性表示操作是从左到右执行的。例如，加运算符的结合性是从左到右的：

```
w = x + y + z;
```

上边的表达式和下面的表达式是等同的：

```
w = ((x + y) + z);
```

但是下面的表达式（无意义的）：

```
x = ~~~y;  
w = x - y = z;  
q = a?b:c?d:e?f:g;
```

等同于：

```
x = ~(-(~y));  
w = (x - (y = z));  
q = a?b:(c?d:(e?f:g));
```

因为一元运算符、赋值运算符和三元条件运算符的结合性是从右到左的。

5.3 算术运算符

我们已经解释过运算符的优先级、结合性以及其他的背景知识了，现在可以开始讨论运算符本身了。本节将详细介绍算术运算符：

加法运算符 (+)

运算符“+”可以对数字运算数进行加法运算，也可以对字符串运算数进行连接操作。如果一个运算数是字符串，那么另一个运算数就会被转换成字符串，然后两者连接在一起。如果“+”运算符的一个运算数是对象，那么它会把这个对象转换成可以进行加法运算或者进行连接操作的数字或字符串。这一转换是通过调用对象的方法 `valueOf()` 或 `toString()` 来执行的。

减法运算符 (-)

当把运算符“-”用于二元操作时，它将从第一个运算数中减去第二个运算数。如果运算数是非数字的，那么运算符“-”会将它们转换成数字。

乘法运算符 (*)

运算符“*”会把两个运算数相乘。如果运算数是非数字的，运算符“*”会将它们转换成数字。

除法运算符 (/)

运算符“/”将用它的第二个运算数来除第一个运算数。如果运算数是非数字的，运算符“/”会将它们转换成数字。如果你熟悉区分整数和浮点型数字的编程语言，那么当你用一个整数来除另外一个整数时，你希望得到的结果仍是一个整数。但是，在 JavaScript 中，由于所有的数字都是浮点型的，所以除法的结果也都是浮点型的，如 $5/2$ 的结果是 2.5 ，而不是 2 。除数为 0 的结果为正无穷或负无穷，而 $0/0$ 的结果则是特殊值 `NaN`。

模运算符 (%)

运算符“%”计算的是第一个运算数对第二个运算数的模。简而言之，就是当第一个运算数被第二个运算数除时，返回的余数。如果运算数是非数字的，运算符“%”会将它们转换成数字。结果的符号和第一个运算数的符号相同。例如， $5\%2$ 结果是 1 。

取模操作的运算数通常都是整数，但它也适用于浮点数，如 $-4.3\%2.1 = -0.1$ 。

一元减运算符 (-)

当“-”被用于一元操作时（用于一个运算数之前），它将执行一元取反操作。简而言之，它将把一个正值转换成相应的负值，反之亦然。如果运算数是非数字的，运算符“-”会将它转换为数字。

一元加运算符 (+)

为了与一元减运算符对称, JavaScript 还有一元加运算符。如果你觉得明确指定数字直接量的符号会使代码看起来更加清楚, 可以采用该运算符:

```
var profit = +1000000;
```

在上面的代码中, “+” 运算符什么也没做, 它只是计算了参数的值。但是要注意, 对于非数字型的参数, “+” 运算符还有将参数转换成数字的功能。如果参数不能被转换, 它将返回 NaN。

递增运算符 (++)

运算符 “++” 是对它惟一的运算数进行递增操作的 (如每次加 1), 这个运算数必须是一个变量、数组的一个元素或者对象的一个属性。如果该变量、元素或属性不是数字, 运算符 “++” 首先会将它转换成数字。该运算符的实际行为是由它相对于运算数的位置来决定的。如果它位于运算数之前, 那么它将被看做前递增运算符, 即先对运算数进行递增, 然后用运算数增长后的值计算。如果该运算符位于运算数之后, 那么它将被看作后递增运算符, 虽然它增加了运算数的值, 但是计算时所用的值是运算数增长前的值。如果要进行递增操作的值不是数字, 通过这一过程它也会被转换成数字。

例如, 下面的代码将 i 和 j 都设置成了 2:

```
i = 1;  
j = ++i;
```

但是接下来的代码却把 i 设置成了 2, 把 j 设置成了 1:

```
i = 1;  
j = i++;
```

这两种形式的递增运算符通常用于增加控制循环的计数器的值。注意, 由于 JavaScript 会自动插入分号, 所以你不能在后递增运算符 (或后递减运算符) 与它之前的运算数之间插入一个换行符。如果这样做, JavaScript 会将那个运算数看做一个完整的语句, 并在运算数之前插入一个分号。

递减运算符 (--)

运算符 “--” 是对它惟一的数字运算数进行递减操作的 (如每次减 1), 这个运算数必须是一个变量、数组的一个元素或者对象的一个属性。如果该变量、元素或属性的值不是数字, 运算符 “--” 首先会将它转换成一个数字。和运算符 “++” 一样, 运算符 “--” 的实际行为是由它相对于运算数的位置决定的。

如果它位于运算数之前，它就先减少运算数的值，并且返回减少后的运算数的值。如果它位于运算数之后，它将减少运算数的值，但是返回的却是没有减少的值。

5.4 相等运算符

本节将介绍 JavaScript 的相等运算符和等同运算符。它们用于比较两个值，以此判断这两个值是相同的，还是不同的，然后根据比较的结果返回一个布尔值（true 或 false）。我们在第六章中将会发现，在像 if 语句和 for 循环这样的结构中，它们非常常用，主要用于控制程序的执行流程。

5.4.1 相等运算符（==）和等同运算符（===）

== 运算符和 === 运算符用来检测两个值是否相等，它们采用了具有同一性的两个不同定义。这两个运算符都接受任意类型的运算数，如果两个运算数相等，它们都返回 true，否则都返回 false。=== 运算符是等同运算符，它采用严格的同一性定义检测两个运算数是否完全等同。== 运算符是相等运算符，它采用比较宽松的同一性定义（即允许进行类型转换）检测两个运算数是否相等。

ECMAScript v3 对等同运算符进行了标准化，JavaScript 1.3 和其后的版本实现了它。随着等同运算符的引入，JavaScript 也支持 =、== 和 === 运算符。首先必须要明白赋值运算符、相等运算符和等同运算符之间的差别，在编码时注意采用正确的运算符。虽然我们称这三个运算符都是“等于”，但是把 = 读作“得到或赋予”，把 == 读作“等于”，把 === 读作“完全等同”，就有助于减少混淆。

在 JavaScript 中，比较数字、字符串和布尔值时使用的都是值（value）。在这种情况下，需要涉及到两个不同的值，运算符 == 和 === 将检测这两个值是否相同，也就是说，当且仅当这两个变量存放的值完全等同时，它们才称为相等或等同。例如，对两个字符串来说，只有当它们存放的字符完全相同时，它们才相等。

另一方面，比较对象、数组和函数时使用的则是引用（reference）。这就是说，只有两个变量引用的是同一个对象时，它们才是相等的。但两个不同的数组无论如何也不相等，即使它们存放的元素完全相同。对于两个存放对象、数组或函数的引用的

变量来说,只有它们引用的是同一个对象、数组或函数,它们才相等。如果想检测两个不同的对象存放的属性是否相同,或者检测两个不同的数组存放的元素是否相同,就必须分别检测每个属性或元素的相等性或等同性。(而且,如果某些属性或元素本身是对象或数组,那么你还必须决定想要比较的深度。)

下面的规则用于判定 `===` 运算符比较的两个值是否完全相等:

- 如果两个值的类型不同,它们就不相同。
- 如果两个值的是数字,而且值相同,那么除非其中一个或两个都是 `NaN` (这种情况它们不是等同的),否则它们是等同的。值 `NaN` 永远不会与其他任何值等同,包括它自身。要检测一个值是否是 `NaN`,可以使用全局函数 `isNaN()`。
- 如果两个值都是字符串,而且在串中同一位置上的字符完全相同,那么它们就完全等同。如果字符串的长度或内容不同,它们就不是等同的。注意,在某些情况下,Unicode 标准允许用多种方法对同样的字符串进行编码。但是,从效率方面考虑,JavaScript 字符串的比较操作严格地逐个字符进行比较,而且它假定在进行比较之前,所有的字符串已经被转换成了“范式”。另一种比较字符串的方法,请参阅本书核心参考手册的“`String.localeCompare()`”部分。
- 如果两个值都是布尔值 `true`,或者两个值都是布尔值 `false`,那么它们等同。
- 如果两个值引用的是同一个对象、数组或函数,那么它们完全等同。如果它们引用的是不同的对象(数组或函数),它们就不完全等同,即使这两个对象具有完全相同的属性或两个数组具有完全相同的元素。
- 如果两个值都是 `null` 或都是 `undefined`,它们完全相同。

下面的规则用于判定 `==` 运算符比较的两个值是否相等:

- 如果两个值具有相同的类型,那么就检测它们的等同性。如果这两个值完全相同,它们就相等。如果它们不完全相同,则它们不相等。
- 如果两个值的类型不同,它们仍然可能相等。用下面的规则和类型转换来检测它们的相等性:
 - 如果一个值是 `null`,另一个值是 `undefined`,它们相等。

- 如果一个值是数字，另一个值是字符串，把字符串转换为数字，再用转换后的值进行比较。
- 如果一个值为 `true`，将它转化为 1，再进行比较。如果一个值为 `false`，把它转化为 0，再进行比较。
- 如果一个值是对象，另一个值是数字或字符串，将对象转换成原始类型的值，再进行比较。可以使用对象的 `toString()` 方法或 `valueOf()` 方法把对象转化成原始类型的值。JavaScript 核心语言的内部类通常先尝试 `valueOf()` 转换，再尝试 `toString()` 转换，但是对于 `Date` 类，则先执行 `toString()` 转换。不属于 JavaScript 核心语言的对象则可以采用 JavaScript 实现定义的方式把自身转换成原始数值。
- 其他的数值组合是不相等的。

如下的代码是一个测试相等性的例子，它带有类型转换：

```
"1" == true
```

该表达式值为 `true`，说明这两个外表完全不同的值事实上相等。首先，布尔值 `true` 被转换成数字 1。然后字符串 "1" 也被转换成了数字 1。所以现在两个数字是相同的，比较运算将返回值 `true`。

在 JavaScript 1.1 中，当相等运算符尝试把一个字符串转换成数字并失败时，它将显示一条错误消息，提示字符串不能被转换，而不是将字符串转换成 `NaN`，并返回 `false` 作为比较的结果。JavaScript 1.2 已经修正了这个 bug。

5.4.1.1 Netscape 中的相等性和不等性

`==` 运算符的行为已经介绍过了，`!=` 运算符的行为将在下一节介绍，它具有一个特例。在 Netscape 4 和其后版本的客户端 JavaScript 中，当把嵌套在 `<script>` 标记中的 `language` 性质明确地指定为 JavaScript 1.2 时，相等运算符的行为和等同运算符相同，不等运算符的行为和不等运算符的行为相同。为了避免这种不兼容性，绝对不要使用 `language="JavaScript 1.2"` 性质来嵌套客户端 JavaScript 代码。第 11.6 节给出了 JavaScript 1.2 的不兼容性的完整列表。

5.4.2 不等运算符 (!=) 和不等同运算符 (!==)

运算符“!=”和“!==”检测的情况恰好与运算符“==”和“===”相反。如果两个值相等，“!=”运算符返回 false，否则返回 true。如果两个值完全相同，“!==”运算符返回 false，否则返回 true。注意该运算符是在 ECMAScript v3 中标准化，在 JavaScript 1.3 和其后的版本中实现的。

我们发现，运算符! 进行的是布尔操作 NOT。这样就很容易记住 != 代表的是“不等”，!== 代表的是“不等同”。要了解对不同的数据类型来说，相等性和等同性是如何定义的，请参阅前面的小节。

5.5 关系运算符

本节介绍 JavaScript 关系运算符。这些运算符用于测试两个值之间的关系（如“小于”或“是...的属性”），根据这些关系是否存在而返回 true 或 false。我们在第 6 章将看到，这些运算符通常用于 if 语句或 while 循环这一类的控制结构中，以控制程序的执行流程。

5.5.1 比较运算符

最常用的关系运算符是比较运算符，它们用于确定两个值的相对顺序。比较运算符包括：

小于运算符 (<)

如果运算符 < 的第一个运算数小于它的第二个运算数，它计算的值就为 true，否则它计算的值为 false。

大于运算符 (>)

如果运算符 > 的第一个运算数大于它的第二个运算数，它计算的值就为 true，否则计算的值为 false。

小于等于运算符 (<=)

如果运算符 <= 的第一个运算数小于或等于第二个运算数，那么它计算的值为 true，否则计算的值为 false。

大于等于运算符 (\geq)

如果运算符 \geq 的第一个运算数大于或等于第二个运算数, 那么它计算的值为 `true`, 否则计算的值为 `false`。

这些比较运算符的运算数可以是任意类型的。但是比较运算只能在数字和字符串上执行, 所以不是数字或字符串的运算数将被转换成数字或字符串。比较和转换的规则如下:

- 如果两个运算数都是数字, 或者都被转换成了数字, 那么将采取数字比较。
- 如果两个运算数都是字符串, 或者都被转换成了字符串, 那么将作为字符串进行比较。
- 如果一个运算数是字符串, 或者被转换成了字符串, 而另一个运算数是数字, 或者被转换成了数字, 那么运算符会将字符串转换为数字, 然后执行数字比较。如果字符串不代表数字, 它将被转换为 `NaN`, 比较的结果是 `false`。(在 JavaScript 1.1 中, 字符串到数字的转换不是生成 `NaN`, 而会引发一个错误。)
- 如果对象可以被转换成数字或字符串, JavaScript 将执行数字转换。例如, 可以从数字的角度比较 `Date` 对象, 也就是说, 比较两个日期, 以判断哪个日期早于另一个日期是有意义的。
- 如果运算数都不能被成功地转换成数字或字符串, 运算符总是返回 `false`。
- 如果某个运算数是 `NaN`, 或被转换成了 `NaN`, 比较运算符总是生成 `false`。

记住, 字符串的比较操作是严格地逐个字符进行比较, 采用的是每个字符在 Unicode 编码集中的数值。虽然在某些情况下, Unicode 标准允许采用不同的字符序列对等价的字符串进行编码, 但是 JavaScript 的比较运算符检测不出这些编码差别, 它们假定所有字符串都是以范式形式表示的。尤其要注意, 字符串比较时会区分大小写, 在 Unicode 编码中 (至少对 ASCII 码子集来说), 所有大写字母小于所有小写字母。如果你不熟悉这一规则, 它会产生令你困惑的结果。例如, 对于 $<$ 运算符, 字符串 `"Zoo"` 小于字符串 `"aardvark"`。

要了解更强壮的字符串比较算法, 请参阅方法 `String.localeCompare()`, 它考虑到了地区特定的字母顺序的定义。对于不区分大小写的比较操作, 必须首先用

`String.toLowerCase()` 方法或 `String.toUpperCase()` 方法将字符串转换成纯小写或纯大写的。

运算符 `<=` (小于等于) 和 `>=` (大于等于) 不是依赖相等运算符和等同运算符来判断两个值是否相等, 而只是将小于等于运算符定义为“不大于”, 将大于等于运算符定义为“不小于”。一个特例是当某个运算数是 (或被转换为) `NaN` 时, 四个比较运算符都返回 `false`。

5.5.2 in 运算符

`in` 运算符要求其左边的运算数是一个字符串, 或可以被转换为字符串, 右边的运算数是一个对象或数组。如果该运算符左边的值是其右边对象的一个属性名, 它返回 `true`。例如:

```
var point = { x:1, y:1 };           // 定义一个对象
var has_x_coord = "x" in point;     // 值为true
var has_y_coord = "y" in point;     // 值为true
var has_z_coord = "z" in point;     // 值为false, 它不是一个3维点
var ts = 'toString' in point;       // 继承属性; 值为true
```

5.5.3 instanceof 运算符

`instanceof` 运算符要求其左边的运算数是一个对象, 右边的运算数是对象类的名字。如果该运算符左边的对象是右边类的一个实例, 它返回 `true`, 否则返回 `false`。我们在第八章中将看到, 在 JavaScript 中, 对象类是由用来初始化它们的构造函数定义的。因此, `instanceof` 运算符右边的运算数应该是一个构造函数的名字。注意, 所有对象都是 `Object` 类的实例。例如:

```
var d = new Date();                 // 用 Date() 构造函数创建一个新对象
d instanceof Date;                   // 值为true; d 由 Date() 创建
d instanceof Object;                 // 值为true; 所有对象都是 Object 类的实例
d instanceof Number;                 // 值为false; d 不是 Number 对象
var a = [1, 2, 3];                   // 用数组直接量的语法创建一个数组
a instanceof Array;                  // 值为true; a 是一个数组
a instanceof Object;                 // 值为true; 所有数组都是对象
a instanceof RegExp;                 // 值为false; 数组不是正则表达式
```

如果 `instanceof` 运算符的左运算数不是对象, 或者右边的运算数是一个对象, 而

不是一个构造函数，它将返回 `false`。另外，如果它右边运算数根本就不是对象，它将返回一个运行时错误。

5.6 字符串运算符

我们在前几节讨论过，有几个运算符在它的运算数是字符串时具有特殊的作用。

运算符 `+` 将连接两个字符串运算数。也就是说，它将创建一个新的字符串，这个字符串是在第一个字符串之后连接上第二个字符串构成的。例如，下面的表达式的计算结果是 `"hello there"`：

```
"hello" + " " + "there"
```

如下的代码生成的是字符串 `"22"`：

```
a = "2"; b = 2;  
c = a + b;
```

运算符 `<`、`<=`、`>` 和 `>=` 将通过比较两个字符串来确定它们的顺序。这个比较采用的是字母顺序。但是要注意，这里的字母顺序是基于 JavaScript 使用的 Unicode 字符编码标准的。在这种编码标准中，所有 Latin 字母表的大写字母都位于小写字母之前（即小于），这会产生意料不到的结果。

运算符 `==` 和 `!=` 可以作用于字符串，但是我们可以发现，这两个运算符适用于所有数据类型，在用于字符串时，它们并没有什么特殊的行为。

运算符 `+` 比较特殊，它给予字符串运算数的优先级比数字运算数的高。前面已经提过，如果该运算符的一个运算数是字符串（或一个对象），那么另一个运算数将被转换为字符串（或者两个运算数都被转换成字符串），然后执行连接运算，而不是执行加法运算。另一方面，如果比较运算符的两个运算数都是字符串，那么它将只执行字符串比较。如果只有一个运算数是字符串，JavaScript 会把它转换成数字。下面的代码作了说明这种情况：

```
1 + 2           // 加法，结果为 3。  
"1" + "2"      // 连接运算，结果为 "12"。  
"1" + 2         // 连接运算；2 被转换为 "2"，结果为 "12"。
```

```
11 < 3           // 数字比较运算。结果为 false。
"11" < "3"       // 字符串比较运算。结果为 true。
11 < 3           // 数字比较运算: "11" 被转换为 11。结果为 false。
"one" < 3         // 数字比较运算: "one" 被转换成 NaN。结果为 false。
// 在 JavaScript 1.1 中, 这将引发一个错误, 而不是把 "one" 转换成 NaN。
```

最后要注意的重要一点是, 当 `+` 用于字符串和数字时, 它并不一定具有结合性。简而言之, 就是结果依赖于操作执行的顺序。从下面的例子中可以发现这一点:

```
s = 1 + 2 + " blind mice"; // 生成 "3 blind mice"
t = "blind mice: " + 1 + 2; // 生成 "blind mice: 12"
```

在行为上出现这种极大差别的原因是由于 `+` 运算符是从左到右运算的, 除非使用括号来改变这种顺序。下面的代码等同于上面的代码:

```
s = (1 + 2) + "blind mice"; // 第一个加法运算生成数字, 第二个生成字符串
t = ("blind mice: " + 1) + 2; // 两个操作都生成字符串
```

5.7 逻辑运算符

逻辑运算符通常用于执行布尔代数。它们常和比较运算符一起使用, 来表示复杂的比较运算, 这些运算要涉及多个变量, 而且常用于 `if`、`while` 和 `for` 语句。

5.7.1 逻辑与运算符 (`&&`)

当运算符 `&&` 的两个运算数都是布尔值时, 它对这两个运算数执行布尔 AND 操作, 即当且仅当它的两个运算数都是 `true` 时, 它才返回 `true`。如果其中一个或两个运算数值为 `false`, 它就返回 `false`。

这个运算符的实际行为比较复杂。首先, 它将计算第一个运算数, 也就是位于它左边的表达式。如果这个表达式的值可以被转换成 `false` (例如, 左边运算数的值为 `null`、`0` 或 `undefined`), 那么运算符将返回左边表达式的值。否则, 它将计算第二个运算数, 也就是位于它右边的表达式, 并且返回这个表达式的值 (注 1)。

注意, 该运算符既可以计算其右边表达式的值, 也可以不计算这个值, 这是由它左

注 1: 在 JavaScript 1.0 和 JavaScript 1.1 中, 如果左边表达式的计算结果为 `false`, 则 `&&` 运算符就返回 `false`, 而不是返回左边表达式中没有被转换的值。

边的表达式的值决定的。你可能会偶然见到利用 `&&` 运算符这一特性的代码。例如，接下来的两行 JavaScript 代码是等效的：

```
if (a == 0) stop();  
(a == 0) && stop();
```

虽然有些程序员（尤其是 Perl 程序员）认为这是一种自然的、有用的程序设计思想，但是我反对使用这种方法。运算符右边的代码不能保证会被计算，这实际上是一个常见的 bug。考虑如下的代码：

```
if ((a == null) && (b > 10)) stop();
```

这个语句所做的可能并不是程序员想要的，因为只要左边的比较表达式的值为 `false`，那么右边的递增运算符就不会被执行。要避免这种问题，就不要在 `&&` 运算符右边使用具有副作用（赋值、递增、递减和函数调用）的表达式，除非你非常确定地知道自己正在做什么。

尽管这种运算符的实际工作方式相当混乱，但是如果只把它看作是一个布尔代数的运算符的话，它就是最简单的，而且也相当安全。虽然它实际返回的并不是一个布尔值，但是这个值却总能够被转换成一个布尔值。

5.7.2 逻辑或运算符 (||)

当运算符 `||` 的两个运算数都是布尔值时，它对这两个运算数执行布尔 OR 操作，即如果它的两个运算数中有一个值为 `true`（或者两个都为 `true`），那么它就返回 `true`。如果它的两个运算数值都为 `false`，它就返回 `false`。

虽然 `||` 运算符常用为布尔 OR 运算符，但是它和 `&&` 运算符一样，行为是比较复杂的。首先，它要计算第一个运算数，即它左边的表达式的值。如果这个表达式的值可以被转换成 `true`，那么它就返回左边这个表达式的值。否则，它将计算第二个运算数，即位于它右边的表达式，并且返回该表达式的值（注 2）。

注 2：在 JavaScript 1.0 和 JavaScript 1.1 中，如果运算符左边的表达式可以转化为 `true`，那么运算符返回 `true`，而不是返回左边表达式未转换的值。

和&&运算符一样,在使用||运算符时,应该避免使右边的运算符产生副作用,除非你故意不计算右边的表达式。

即使||运算符的运算数不是布尔值,仍然可以将它看作布尔OR运算,因为无论它返回的值是什么类型的,都可以被转换为布尔值。

5.7.3 逻辑非运算符 (!)

运算符!是一个一元运算符,它放在一个运算数之前。它用来对运算数的布尔值取反。例如,如果变量a的值为true(或者它的值可以转换为true),那么!a的值就是false。如果表达式p&&q的值为false(或者它的值可以被转换为false),那么!(p&&q)的值就是true。注意,对任何值x应用两次该运算符(即!!x)都可以将它转换成一个布尔值。

5.8 逐位运算符

尽管在JavaScript中所有的数字都是浮点型的,但是逐位运算符却要求它的数字运算数是整型的。它们操作的这些整型运算数使用的是32位的整数表示法,而不是等价的浮点表示法。这些运算符中有四个是对运算数的每个位执行布尔代数运算,就像运算数中的每个位都是一个布尔值,执行的运算与我们在前面看到逻辑运算符的运算相似。其他三个逐位运算符用于位的左移或右移。

在JavaScript1.0和JavaScript1.1中,如果这种运算符用于非整型的运算数,或者用于太大的以至于不能用32位的整数表示的运算数,它将返回NaN。但是在JavaScript1.2和ECMAScript中,则通过舍弃运算数的小数部分或高于32位的数位来将运算数限制在32位的整数这个范围内。移位运算符要求其右边的运算数在0到31之间。当用如前所述的方法把该运算数转换成32位的整数后,它们将舍弃第5位后的数位,以生成一个位数正确的数字。

如果你还不熟悉二进制数和十进制整数的二进制表示法,那么可以跳过本节所介绍的运算符。本节并没有介绍这些运算符的用途,因为它们用于低级的二进制数操作,在JavaScript的程序设计中并不常用。逐位运算符主要包括:

按位与运算符 (&)

运算符 & 对它的整型参数逐位执行布尔 AND 操作。只有两个运算数中相应的位都为 1, 那么结果中的这一位才为 1。例如, $0x1234 \& 0x00FF = 0x0034$ 。

按位或运算符 (|)

运算符 | 对它的整型参数逐位执行布尔 OR 操作。如果其中一个运算数中的相应位为 1 或者两个运算数中的相应位都为 1, 那么结果中的这一位就为 1。例如, $9|10 = 11$ 。

按位异或运算符 (^)

运算符 ^ 对它的整型参数逐位执行布尔异或操作。异或是指第一个运算数是 true, 或者第二个运算数是 true, 但是两者不能同时为 true。如果两个运算数中只有一个数的相应位为 1 (但不能同时为 1), 那么结果中的这一位就为 1。例如, $9^10 = 3$ 。

按位非运算符 (~)

运算符 ~ 是个一元运算符, 它位于一个整型参数之前, 它将运算数的所有位取反。根据 JavaScript 中带符号的整数的表示方法, 对一个值使用 ~ 运算符相当于改变它的符号并且减 1。例如, $\sim 0x0f = 0xfffffffff0$ 或 -16 。

左移运算符 (<<)

运算符 << 左移第一个运算数中的所有位, 移动的位数由第二个运算数指定, 移动的位数应该是一个 0 到 31 的整数。例如, 在表达式 $a<<1$ 中, a 的第一位变成了它的第二位, a 的第二位变成了它的第三位, 以此类推。新的第一位用 0 来补充, 舍弃第 32 位的值。将一个值左移 1 位相当于对它乘 2, 左移 2 位相当于对它乘 4, 以此类推。例如, $7<<1=14$ 。

带符号的右移运算符 (>>)

运算符 >> 右移第一个运算数中的所有位, 移动的位数由第二个运算数指定, 移动的位数应该是一个 0 到 31 的整数。舍弃右边移出的位, 填补在左边的位由原运算数的符号位决定, 以便保持结果的符号与原操作数一致。如果第一个运算数是正的, 就用 0 填补结果的高位; 如果第一个运算数是负的, 就用 1 填补结果的高位。将一个值右移 1 位, 相当于用 2 除它 (丢弃余数), 右移 2 位, 相当于用 4 除它, 以此类推。例如, $7>>1=3$, $-7>>1=-4$ 。

用0补足的右移运算符(>>>)

运算符>>>和运算符>>一样,只是从左边移入总是0,与原运算数的符号无关。例如, $-1 \gg 4 = -1$, 但是 $-1 \ggg 4 = 268435455 (0 \times 0\text{ffffff})$ 。

5.9 赋值运算符

我们在第四章中讨论过,在JavaScript中“=”是用于给一个变量赋值的。例如:

```
i = 0
```

虽然你会认为这样一行JavaScript代码并不是一个可计算的表达式,但是实际上,它确实是一个表达式,而且从技术上说,=是一个运算符。

运算符=要求它左边的运算数是一个变量,数组的一个元素,或者是对象的一个属性。右边的运算数是一个任意的值,这个值可以是任何类型的。赋值表达式的值就是它右边的运算数的值。此外,运算符=可以将它右边的值赋给左边的变量、元素或属性,以便将来可以使用变量、元素或属性来引用这个值。

因为=被定义为一个运算符,所以可以将它用于更复杂的表达式。例如,你可以在同一个表达式中进行赋值并检测这个值。代码如下:

```
(a = b) == 0
```

如果你这样做,一定要确保自己十分清楚运算符=和==之间的区别。

赋值运算符的结合性是从右到左的,也就是说,如果一个表达式中有多个赋值运算符,将从右到左进行计算。你可以编写如下的代码,将一个值赋给多个变量:

```
i = j = k = 0;
```

记住,每个赋值表达式都有一个值,这个值就是赋值运算符右边的值。因此,在上面的代码中,第一个赋值表达式(最右边的赋值表达式)的值就是第二个赋值表达式(中间的表达式)赋值运算符右边的值,而第二个赋值表达式的值又是最后一个赋值表达式(最左边的表达式)赋值运算符右边的值。

5.9.1 带操作的赋值运算

除了常规的赋值运算符 `=` 之外，JavaScript 还支持许多其他的赋值运算符，这些运算符将赋值运算符和其他运算符联合在一起，提供了一些快捷的运算方式。例如，运算符 `+=` 执行的是加法运算和赋值操作。表达式：

```
total += sales_tax
```

和下面的表达式是等效的：

```
total = total + sales_tax
```

运算符 `+=` 可以作用于数字和字符串。如果它的运算数是数字，那么它将执行加法运算和赋值操作；如果运算数是字符串，它就执行连接操作和赋值操作。

这类运算符还包括 `-=`、`*=`、`&=` 等。表 5-2 列出了该类的所有运算符。在大多数情况下，表达式为：

```
a op= b
```

这里，`op` 表示一个运算符，这个表达式等同于：

```
a = a op b
```

这些表达式的不同之处仅仅在于它们会产生副作用，例如函数调用或增量运算。

表 5-2: 赋值运算符

| 运算符 | 示例 | 等价等式 |
|----------------------------|--------------------------------|-----------------------------------|
| <code>+=</code> | <code>a += b</code> | <code>a = a + b</code> |
| <code>-=</code> | <code>a -= b</code> | <code>a = a - b</code> |
| <code>*=</code> | <code>a *= b</code> | <code>a = a * b</code> |
| <code>/=</code> | <code>a /= b</code> | <code>a = a / b</code> |
| <code>%=</code> | <code>a %= b</code> | <code>a = a % b</code> |
| <code><<=</code> | <code>a <<= b</code> | <code>a = a << b</code> |
| <code>>>=</code> | <code>a >>= b</code> | <code>a = a >> b</code> |
| <code>>>>=</code> | <code>a >>>= b</code> | <code>a = a >>> b</code> |
| <code>&=</code> | <code>a &= b</code> | <code>a = a & b</code> |
| <code> =</code> | <code>a = b</code> | <code>a = a b</code> |
| <code>^=</code> | <code>a ^= b</code> | <code>a = a ^ b</code> |

5.10 其他运算符

JavaScript 还支持许多其他的运算符，我们将在接下来的小节中讨论这些运算符。

5.10.1 条件运算符 (?:)

条件运算符是 JavaScript 中唯一的三元运算符（带有三个运算数），有时就称它为三元运算符。这个运算符常被写为 `?:`，但是在代码中它却不是这样的，因为这个运算符具有三个运算数，第一个位于 `?` 之前，第二个位于 `?` 和 `:` 之间，第三个位于 `:` 之后。可以用如下方式来使用它：

```
x > 0 ? x*y : -x*y
```

条件运算符的第一个运算数必须是一个布尔值（或能够被转换为布尔值），通常它是一个比较表达式的结果。第二个和第三个运算数可以是任何类型的值。条件运算符的返回值是由第一个运算数的布尔值决定的。如果这个运算数的值为 `true`，那么条件表达式的值就是第二个运算数的值。如果第一个运算数的值为 `false`，那么条件表达式的值就是第三个运算数的值。

虽然使用 `if` 语句可以得到同样的结果，但是在许多情况下使用运算符 `?:` 更为快捷。下面是它的一个典型用法，即用它来检查一个变量是否被定义，如果定义了，就使用这个变量，否则就提供一个默认值：

```
greeting = "hello " + (username != null ? username : "there");
```

这和下面 `if` 语句是等价的，但是上面的代码结构更为紧凑一些：

```
greeting = "hello ";  
  
if (username != null)  
    greeting += username;  
else  
    greeting += "there";
```

5.10.2 typeof 运算符

`typeof` 是个一元运算符，放在一个运算数之前，这个运算数可以是任意类型的。它的返回值是一个字符串，该字符串说明了运算数的类型。

如果 `typeof` 的运算数是数字、字符串或者布尔值，它返回的结果就是 “number”、“string” 或 “boolean”。对对象、数组和 `null`，它返回的是 “object”。对函数运算数它返回的是 “function”。如果运算数是未定义的，它将返回 “undefined”。

当 `typeof` 的运算数是 `Number`、`String` 或 `Boolean` 这样的包装对象时，它返回的是 “object”。此外，对 `Date` 和 `RegExp` 对象，它也返回 “object”。对于那些不属于 JavaScript 核心语言，而是由 JavaScript 嵌入的环境提供的对象，`typeof` 的返回值是由实现决定的。但是，在客户端 JavaScript 中，`typeof` 对所有的客户端对象返回的都是 “object”，这与它对所有核心对象的处理是一样的。

可以采用如下方式在表达式中使用 `typeof` 运算符：

```
typeof i  
(typeof value == "string") ? "" + value + "" : value
```

注意，可以用括号将 `typeof` 的运算数括起来，这使得 `typeof` 看起来更像是个函数名，而不是一个运算符关键字：

```
typeof(i)
```

由于 `typeof` 对所有的对象和数组类型返回的都是 “object”，所以它只在区别对象和原始类型时才有用。要区别一种对象类型和另一种对象类型，必须使用其他的方法。例如 `instanceof` 运算符和 `constructor` 属性（参阅核心参考手册部分对 `Object.constructor` 的说明）。

`typeof` 运算符是由 ECMAScript v1 规范定义的，在 JavaScript 1.1 及其后的版本中实现。

5.10.3 对象创建运算符（new）

`new` 运算符用来创建一个新对象，并调用构造函数初始化它。`new` 是一个一元运算符，出现在构造函数的调用之前。它的语法如下：

```
new constructor(arguments)
```

`constructor` 必须是一个构造函数表达式，其后应该有一个用括号括起来的参数列表，列表中有零或多个参数，参数之间用逗号分隔。JavaScript 简化了该语法，即

如果函数调用时没有参数，就可以省去括号，这种简化了的语法只适用于运算符 `new`。下面是一些使用 `new` 运算符的例子：

```
o = new Object;           // 此处省略了可选的括号
o = new Date();           // 返回一个表示当前时间的 Date 对象
c = new Rectangle(3.0, 4.0, 1.5, 2.75); // 创建 Rectangle 类的对象
obj[i] = new constructors[i]();
```

运算符 `new` 首先创建一个新对象，该对象的属性都未被定义。接下来，它将调用特定的构造函数，传递指定的参数，此外还要把新创建的对象传递给关键字 `this`。这样构造函数就可以使用关键字 `this` 来初始化新对象。我们将在第八章中学到更多关于 `new` 运算符、关键字 `this` 和构造函数的内容。

利用 `new Array()` 语法，可以用 `new` 运算符创建数组。我们将在第八章和第九章中看到更多有关创建和使用对象与数组的内容。

5.10.4 delete 运算符

`delete` 运算符是个一元运算符，它将删除运算数所指定的对象的属性、数组元素或变量（注3）。如果删除操作成功，它将返回 `true`，如果运算数不能被删除，它将返回 `false`。并非所有的属性和变量都是可以删除的，某些内部的核心属性和客户端属性不能删除，用 `var` 语句声明的变量也不能被删除。如果 `delete` 使用的运算数是一个不存在的属性，它将返回 `true`（令人吃惊的是，ECMAScript 标准规定，当 `delete` 运算的运算数不是属性、数组元素或变量时，它返回 `true`）。下面是一些使用该运算符的例子：

```
var o = {x:1, y:2};      // 定义一个变量；把它初始化为一个对象
delete o.x;              // 删除一个对象属性，返回 true
typeof o.x;              // 该属性不存在；返回 "undefined"
delete o.x;              // 删除一个不存在的属性；返回 true
delete o;                // 不能删除未声明的变量；返回 false
delete 1;                // 不能删除整数；返回 true
x = 1;                   // 不用 var 关键字，隐式地声明一个变量
delete x;                // 不能删除这种类型的变量；返回 true
```

注3：如果你是一个 C++ 程序员，那么请注意，JavaScript 中的 `delete` 运算符和 C++ 中的 `delete` 运算符不同。在 JavaScript 中，内存分配是由无用存储单元的收集程序自动处理的，你无需担心显式地释放内存。因此，JavaScript 并不需要一个 C++ 式的 `delete` 运算符来释放整个对象。

```
x; // 运行时错误: x 没有被定义
```

注意, 删除属性、变量或数组元素不只是把它们值设置为 `undefined`。当删除一个属性后, 该属性将不再存在。详情请参阅 4.3.2 小节的相关讨论。

ECMAScript v1 规范对 `delete` 运算符进行了标准化, JavaScript 1.2 及其后的版本实现了它。注意, JavaScript 1.1 和 1.0 中也存在 `delete` 运算符, 但是它在这些版本中并不能执行真正的删除操作, 而只是将属性、变量或数组元素设为 `null` 罢了。

注意, `delete` 所能影响的只是属性值, 并不能影响被这些属性引用的对象。考虑如下的代码:

```
var my = new Object(); // 创建一个名为“my”的对象
my.hire = new Date(); // my.hire 引用一个 Date 对象
my.fire = my.hire;    // my.fire 引用同一个 Date 对象
delete my.hire;        // hire 属性被删除; 返回 true
document.write(my.fire); // 但 my.fire 仍然引用那个 Date 对象
```

5.10.5 void 运算符

`void` 是一个一元运算符, 它可以出现在任何类型操作数之前。这个运算符的用途比较特殊, 它总是舍弃运算数的值, 然后返回 `undefined`。这种运算符常用在客户端的 `javascript:URL` 中。在这里可以计算表达式的值, 而浏览器不会显示出这个值。

例如, 可以在 HTML 的标记中以如下方式使用 `void` 运算符:

```
<a href="javascript:void window.open();" >Open New Window</a>
```

`void` 的另一个用途是专门生成 `undefined` 值。ECMAScript v1 定义了 `void` 运算符, JavaScript 1.1 实现了它。但是全局的 `undefined` 属性则是在 ECMAScript v3 中定义的, 由 JavaScript 1.5 实现。所以, 考虑到向后兼容性, 你会发现用表达式 `void 0` 比使用 `undefined` 属性更有用。

5.10.6 逗号运算符 (,)

逗号运算符非常简单。它先计算其左边的参数, 再计算其右边的参数, 然后返回右边参数的值。因此, 如下的代码:


```
i=0, j=1, k=2;
```

等价于:

```
i = 0;  
j = 1;  
k = 2;
```

这个奇怪的运算符只在个别环境中使用,一般是在只允许出现一个表达式的地方计算几个不同的表达式时才使用的。在实际应用中,逗号运算符只和 for 循环语句联合使用,我们将在第六章中看到这个语句。

5.10.7 数组和对象存取运算符

我们在第3章中简要地提到过,你可以使用方括号([])来存取数组的元素,使用点号(.)来存取对象的元素。在JavaScript中,“[]”和“.”都是运算符。

运算符“.”左边的运算数是一个对象,右边的运算数是一个标识符(属性名)。右边的运算数既不能是字符串,也不能是存放字符串的变量,而应该是属性或方法的直接量名,而且不需要指明类型。下面是一些例子:

```
document.lastModified  
navigator.appName  
frames[0].length  
document.write('hello world')
```

如果对象中没有指定的属性,JavaScript并不会把这看作一个错误,而是返回 undefined 作为这个表达式的值。

大多数运算符允许运算数是任意的表达式,只要这个表达式的类型和运算数的类型是匹配的。而运算符“.”则是个例外,位于它右边的运算数必须是一个标识符,其他的类型都是不允许的。

运算符“[]”用于存取数组元素,它还可以用于存取对象的属性,而且没有“.”运算符对其右边运算数的限制。如果它的第一个运算数(位于左方括号之前的运算数)引用的是一个数组,那么它的第二个运算数(位于括号之内的运算数)就应该是一个值为整数的表达式。例如:

```
frames[1]
```

```
document.forms[i] = {}  
document.forms[i].elements[j++]
```

如果运算符“[]”的第一个运算数引用的是一个对象，那么第二个运算数就应该是一个值为字符串的表达式，它指明了该对象的一个属性。注意，在这种情况下，第二个运算数是一个字符串，而不是一个标识符。它应该是一个用引号括起来的常量，或者是一个变量、表达式，它们引用了一个字符串。例如：

```
document['lastModified']  
frames[0]['length']  
data['val' + i]
```

运算符“[]”通常用于存取数组元素。要存取一个对象的属性，它并没有使用运算符“.”那么方便，因为它需要引用属性名。但是，当对象用作关联数组时，由于属性名是动态生成的，所以不能使用运算符“.”，而只能使用运算符“[]”。当你使用for/in循环时，这种情况很常见，我们将在第六章中介绍这种语句。下面的JavaScript代码使用了for/in循环和“[]”运算符来输出一个对象o的所有属性的名字和值：

```
for (f in o) {  
    document.write('o.' + f + ' = ' + o[f]);  
    document.write('<br>');  
}
```

5.10.8 函数调用运算符

在JavaScript中运算符()用于调用函数。这是一个特殊的运算符，因为它没有固定数目的运算数。它的第一个运算数总是一个函数名或者是一个引用函数的表达式，其后就是左括号和数目不定的运算数，这些运算数可以是任意的表达式，它们之间用逗号隔开。右括号跟在最后一个运算数之后。()运算符将计算它的每一个运算数，然后调用第一个运算数指定的函数，而且把余下的运算数的值传递给这个函数作为它的参数。例如：

```
document.close()  
Math.sin(x)  
alert("Welcome " + name)  
Date.UTC(2000, 11, 31, 23, 59, 59)  
funcs[i].f(funcs[i].args[0], funcs[i].args[1])
```

第六章

语句

我们在上一章中见到过，表达式在 JavaScript 中是一个短语，可以用它进行计算以生成一个值。虽然表达式中的运算符可以带来各种各样的副作用，但是一般说来，表达式是什么都不做的。要想使某件事情发生，就必须使用 JavaScript 语句 (statement)，它就像一个完整的句子或命令。本章将介绍 JavaScript 的各种语句，并且解释这些语句的语法。一个 JavaScript 程序就是一些语句的集合，所以一旦你掌握了 JavaScript 语句，就可以编写 JavaScript 程序了。

在研究 JavaScript 语句之前，先回忆一下第 2.4 小节，其中介绍过 JavaScript 语句之间是以分号分隔的。但是如果把每个语句单独放在一行，JavaScript 允许省略分号。不过还是养成在每处都使用分号的习惯比较好。

6.1 表达式语句

在 JavaScript 中，最简单的语句莫过于表达式了。我们在第 5 章中见过这种语句。赋值语句是一种主要的表达式语句。例如：

```
1 s = "Hello " + name;  
2 i *= 3;
```

递增运算符 (++) 和递减运算符 (--) 都和赋值语句有关。它们的作用是改变一个变量的值，就像执行了一条赋值语句一样：

```
counter--;
```

delete运算符的重要作用是删除一个对象的属性，所以，它一般作为语句使用，而不是作为复杂表达式的一部分：

```
delete o.x;
```

函数调用是表达式语句的另一个大类。例如：

```
alert('Welcome, ' + name);  
window.close();
```

虽然这些客户端函数调用都是表达式，但是它们都对Web浏览器产生了影响，所以它们也是语句。如果一个函数没有任何作用，那么调用它就没有什么意义了，除非它是赋值语句的一部分。例如，你绝不会计算了一个正弦值，然后把结果丢弃：

```
Math.cos(x);
```

相反，你计算了这个值之后，还要把它赋给一个变量，以便将来能够使用这个值。

```
cx = Math.cos(x);
```

再次提醒你，这些示例中的每行代码都是以分号结束的。

6.2 复合语句

在第五章中我们见到过，可以用逗号运算符将几个表达式联合起来，形成一个表达式。JavaScript还有一种方法可以将几个语句联合起来，形成一个语句或者语句块(statement block)。这只需要用花括号把几个语句括起来即可。下面的几行代码就可以作为一个单独的语句，用于JavaScript中任何需要使用单个语句的地方：

```
{  
    x = Math.PI;  
    cx = Math.cos(x);  
    alert("cos(" + x + ") = " + cx);  
}
```

注意，虽然语句块可以作为一个语句，但是在它的结尾处却不需要使用分号。块中的原始语句需要以分号来结尾，但是块本身并不需要这样做。

虽然用逗号运算符将几个表达式联合起来这一方法并不常用,但是将几个语句联合起来形成一个大的语句块却是极其常见的。我们在接下来的几节中会发现,一些JavaScript语句可以包含别的语句(就像表达式可以包含别的表达式一样),这样的语句就叫做复合语句(compound statement)。正式的JavaScript语法规则规定每个复合语句可以包含一个子语句,那么使用语句块,你就可以将任意数量的语句放在这个子语句中。

JavaScript解释器执行复合语句时,只是一句一句地按照编写的顺序执行构成它的语句。通常JavaScript解释器会执行所有的语句。但在某些情况下,复合语句会意外终止。发生这种终止的情况主要是由于复合语句含有break语句、continue语句、return语句或throw语句,或者它引发了错误,或者它调用的函数引发了不能捕捉的错误或抛出了不能捕捉的异常。我们将在后面的小节中学习更多有关这些意外终止的内容。

6.3 if 语句

if语句是基本的控制语句,它使得JavaScript进行选择,更准确地说,就是有条件地执行语句。这个语句有两种形式,第一种形式是:

```
if (expression)
    statement
```

在这种形式中,expression是要被计算的,如果计算的结果是true,或者可以被转换成true,那么就执行statement。如果expression的值为false,或者可以被转换成false,那么就不执行statement。例如:

```
if (username == null)           // 如果username是null或undefined,
    username = "John Doe";      // 定义它
```

同样:

```
// 如果 username 是 null、undefined、0、"" 或 NaN, 它将被转换为 false,
// 该语句将给它赋一个新值。
if (!username) username = "John Doe";
```

虽然在这里看起来括起表达式的括号无关紧要,但是它们却是if语句的语法所必需的一部分。

我们在前面提到过,你可以使用一个语句块来替换单个的语句,所以if语句也可以用如下代码:

```
if ((address == null) || (address == '')) {  
    address = "undefined";  
    alert("Please specify a mailing address.");  
}
```

在这些例子中,格式上的缩进并不是必需的。JavaScript会把多余的空格都忽略掉。而且,由于我们在每条语句之后都用了分号,所以可以将例子中的所有代码都写到一行中。但是像上例所示的那样使用换行和缩进会使得代码更易读,也更容易理解。

if语句的第二种形式引入了else从句,当expression的值是false时,就执行这个从句。它的语法如下:

```
if (expression)  
    statement1  
else  
    statement2
```

在这种形式中,先计算expression的值,如果它是true,就执行statement1,否则就执行statement2。例如:

```
if (username != null)  
    alert("Hello " + username + "\nWelcome to my home page.");  
else {  
    username = prompt('Welcome!\n What is your name?');  
    alert("Hello " + username);  
}
```

当在具有else从句的if语句中进行嵌套时,要注意确保else语句匹配了正确的if语句。考虑如下的代码:

```
i = j = 1;  
k = 2;  
if (i == j)  
    if (j == k)  
        document.write("i equals k");  
else  
    document.write("i doesn't equal j");    // 错误!!
```

在这个例子中,内层的if语句构成了外层的if语句所需要的子语句。但是,if和else是如何匹配的并不十分清楚(只有缩进给出了一些暗示)。而且,在这个例子

中，缩进的给出暗示是错误的，因为JavaScript的解释器实际上将上面的这个例子解释为：

```
if (i == j) {  
    if (j == k)  
        document.write('i equals k');  
else  
    document.write('i doesn't equal j');    // OOPS!  
}
```

JavaScript中的规则（和大多数程序设计语言一样）是，else从句是离它最近的if语句的一部分。要使这个例子更易读、易理解、易支持和调试，你可以使用花括号：

```
if (i == j) {  
    if (j == k) {  
        document.write('i equals k');  
    }  
}  
else {    // 大括号的位置造成了多么大的差别！  
    document.write('i doesn't equal j');  
}
```

虽然这并不是本书中使用的风格，但是许多程序员都有将if和else语句的主体用花括号括起来的习惯（就像其他的复合语句一样，如while循环），即使这个主体仅由一条语句构成。坚持这样做，就可以避免出现上述的问题。

6.4 else if 语句

我们已经看到了，使用if/else语句可以根据表达式的结果来测试一个条件，然后执行两条代码中的一条。但是当我们需要执行的是多条代码中的一条时又该怎么办呢？一个解决方法是使用else if语句。else if语句并不是一个真正的JavaScript语句，它只不过是在重复使用if/else语句时经常使用的一个程序设计思想罢了：

```
if (n == 1) {  
    // 执行代码块 #1  
}  
else if (n == 2) {  
    // 执行代码块 #2  
}  
else if (n == 3) {  
    // 执行代码块 #3  
}
```

```
else {  
    // 如果所有判断都为 false, 执行代码块 #4  
}
```

这段代码并没有什么特殊之处。它只是一系列的 if 语句, 其中每个 if 语句都是前一句的 else 从句的一部分。使用 else if 语句的思想是很可取的, 而且比使用语法上和它等价的嵌套形式更易读:

```
if (n == 1) {  
    // 执行代码块 #1  
}  
else {  
    if (n == 2) {  
        // 执行代码块 #2  
    }  
    else {  
        if (n == 3) {  
            // 执行代码块 #3  
        }  
        else {  
            // 如果所有判断都为 false, 执行代码块 #4  
        }  
    }  
}
```

6.5 switch 语句

一个 if 语句会在程序的执行流程中产生一个分支。你也可以像前面一节所介绍的那样使用多个 if 语句来执行多个分支。但是, 这并不是最好的解决方案, 尤其是当所有的分支都依赖于一个变量的值时。在这种情况下, 重复检测多个 if 语句中同一个变量的值是一种浪费。

switch 语句 (在 JavaScript 1.2 中实现, 由 ECMAScript v3 标准化) 正是用来处理这种情况的, 它比重复使用 if 语句有效得多。JavaScript 的 switch 语句和 Java 或 C 的 switch 语句非常相似。关键字 switch 之后是一个表达式和一个代码块, 这和 if 语句比较相似:

```
switch(expression) {  
    statements  
}
```

但是, 完整的 switch 语句的语法比上面所示的语法要复杂得多。在代码块中, 不

同的位置要使用case关键字后加一个值和一个冒号来标记。当执行一个switch语句时，它先计算expression的值，然后查找和这个值匹配的case标签。如果找到了相应的标签，就开始执行case标签后的代码块中的第一条语句。如果没有找到和这个值匹配的case标签，就开始执行标签default（它是在特殊情况下使用的标签）后的第一条语句。如果没有default标签，它就跳过所有的代码块。

switch语句是一个解释起来容易产生混淆的语句，如果使用一个例子来解释，它的操作就变得比较清晰了。下面的switch语句和上一节中所示的重复使用if/else的语句等价：

```
switch(n) {  
  case 1: // 如果n == 1, 从此处开始  
    // 执行代码块 #1。  
    break; // 在此处停止执行  
  case 2: // 如果n == 2, 从此处开始  
    // 执行代码块 #2。  
    break; // 在此处停止执行  
  case 3: // 如果n == 3, 从此处开始  
    // 执行代码块 #3。  
    break; // 在此处停止执行  
  default: // 如果所有判断都为 false……  
    // 执行代码块 #4。  
    break; // 在此处停止执行  
}
```

注意，在上面的代码中，每一个case语句的结尾处都使用了关键字break。我们将在后面的小节中介绍break语句，它使程序跳到switch语句或循环语句的结尾处。在switch语句中，case从句只是指明了想要执行的代码的起点，但是并没有指明终点。如果没有break语句，那么switch语句就会从和expression的值匹配的case标签处的代码块开始执行，顺次执行其后的语句，直到代码块的结尾。这种由一个case标签执行到下一个case标签的代码是极少使用的，大多数情况下，应该使用break语句来终止每个case语句。（不过，如果是在函数中使用switch语句，可以使用return语句代替break语句。这两个语句都用于终止switch语句，防止一个case块执行完后继续执行下一个case块。）

下面是一个更实际的switch语句的例子，它根据值的类型，把这个值转换成了一个字符串：

```
function convert(x) {  
  switch(typeof x) {
```

```

        case 'number':           // 把数字转换成十六进制的整数
            return x.toString(16);
        case 'string':          // 返回引号包围的字符串
            return '"' + x + '"';
        case 'boolean':         // 转换成大写的 TRUE 或 FALSE
            return x.toString().toUpperCase();
        default:                // 以常规方式转换成其他类型
            return x.toString();
    }
}

```

注意，在前两个例子中，case关键字后跟随的是数字和字符串直接量。这是实际应用中 switch 语句最常用的方法，但是 ECMAScript v3 标准允许 case 语句后跟随任意的表达式（注1）。例如：

```

case 60*60*24:
case Math.PI:
case n+1:
case a[0]:

```

switch 语句首先计算 switch 关键字后的表达式，然后按照出现的顺序计算 case 后的表达式，直到找到与 switch 表达式的值相匹配的 case 表达式为止（注2）。由于匹配的 case 表达式是用 === 等同运算符判定的，而不是用 == 相等运算符判定的，所以表达式必须在没有类型转换的情况下进行匹配。

注意，用含有副作用的 case 表达式（如产生函数调用或赋值）不是好的程序设计习惯，因为每次执行 switch 语句时并不会计算所有的 case 表达式。当某些情况下出现副作用时，很难理解并预测程序的正确行为。最安全的办法是把 case 表达式限制在常量表达式的范围内。

前面提到过，如果没有一个 case 表达式与 switch 表达式匹配，switch 语句就开始执行标签为 default: 的语句体。如果没有 default: 标签，switch 语句就跳过

注1：这使得 JavaScript 的 switch 语句和 C、C++、Java 的 switch 语句有很大的不同。在 C、C++ 和 Java 中，case 表达式必须是编译时刻的常量，它们的值必须为整数或其他整数类型，而且所有值的类型必须相同。

注2：这意味着 JavaScript 的 switch 语句的执行效率比 C、C++ 和 Java 的 switch 语句低。由于 C、C++ 和 Java 中，case 表达式是编译时刻的常量，所以不必像在 JavaScript 中那样计算它们在运行时的值。而且，由于在 C、C++ 和 Java 中，case 表达式的值是整数类型的，所以可以使用高效率的“跳转表”来实现 switch 语句。

它的整个主体。注意，在前面的例子中，`default:` 标签都出现在 `switch` 主体的末尾，位于所有 `case` 标签之后。这不过是个合理的、常用的 `default:` 标签的位置，实际上，`default:` 标签可以放置在语句体的任何地方。

`switch` 语句是在 JavaScript 1.2 中实现的，但是与 ECMAScript 标准并不完全一致。在 JavaScript 1.2 中，`case` 表达式必须是直接量或编译时刻的常量，不涉及任何变量或方法调用。而且，虽然 ECMAScript 标准不限制 `switch` 表达式和 `case` 表达式的类型，但是 JavaScript 1.2 和 JavaScript 1.3 都要求它们的表达式值为原始的数字、字符串或布尔值。

6.6 while 语句

和 `if` 语句是基本的控制语句，允许 JavaScript 进行抉择一样，`while` 语句也是一个基本语句，它允许 JavaScript 执行重复的动作。它的语法如下：

```
while (expression)
    statement
```

`while` 首先计算 `expression` 的值。如果它的值是 `false`，那么 JavaScript 就转而执行程序中的下一条语句。如果值为 `true`，那么就执行构成循环体的 `statement`，然后再计算 `expression` 的值。这次如果 `expression` 的值是 `false`，那么 JavaScript 就转而执行程序中的下一条语句，否则再次执行 `statement`。这种循环会一直继续下去，直到 `expression` 的值为 `false` 为止，这时就说明 `while` 语句结束了，JavaScript 就会执行下一条语句。注意，用 `while (true)` 会创建一个无限循环。

通常说来，我们并不想让 JavaScript 反复执行同一操作，所以几乎在每一个循环中，都会有一个或者多个变量随着循环的迭代 (`iteration`) 而改变。正是由于改变了这些变量，所以每次循环执行 `statement` 的操作也有可能不同。而且，如果改变了的变量或改变所涉及到的变量是属于 `expression` 的，那么每次循环时 `expression` 的值也会不同。这一点很重要，否则一个初始值是 `true` 的表达式的值永远也不会改变，那么循环也就永远都不会结束了。下面是一个 `while` 循环的例子：

```
var count = 0;
while (count < 10) {
    document.write(count + "<br>");
    count++;
}
```

你可以发现,在这个例子中,变量 `count` 的初始值是 0,在循环运行的过程中,它的值每次都增加 1。如果循环执行了 10 次,表达式的值就变成 `false` 了(因为变量 `count` 的值不会小于 10),那么 `while` 就会结束,JavaScript 将执行程序中的下一条语句。大多数循环都有一个像 `count` 这样的计数器变量。虽然循环计数器常用 `i`、`j`、`k` 这样的变量名,但是如果想要使代码更易理解,就应该使用更具有描述性的变量名。

6.7 do/while 语句

`do/while` 循环和 `while` 循环非常相似,只不过它是在循环底部检测循环表达式,而不是在循环顶部进行检测。这就意味着循环体至少会被执行一次。它的语法如下:

```
do
    statement
while (expression);
```

JavaScript 1.2 和其后的版本实现了 `do/while` 语句,ECMAScript v3 对它进行了标准化。

`do/while` 循环并不像 `while` 循环那么常用。这是因为在实践中,那种你想要循环至少执行一次的情况并不是那么常见。例如:

```
function printArray(a) {
    if (a.length == 0)
        document.write("Empty Array");
    else {
        var i = 0;
        do {
            document.write(a[i] + "<br>");
        } while (++i < a.length);
    }
}
```

在 `do/while` 循环和普通的 `while` 循环之间有两点不同之处。首先, `do` 循环要求必须使用关键字 `do` 来标识循环的开头,用关键字 `while` 来标识循环的结尾并引入循环条件。其次,和 `while` 循环不同, `do` 循环是用分号结尾的。这是因为 `do` 循环的结尾处是循环条件,而不是简单地用花括号标识循环体的结束。

在 JavaScript 1.2 中,当 `continue` 语句(参见 6.12 小节)用在 `do/while` 语句内部

时, 存在一个 bug。考虑到这一点, 应该在 JavaScript 1.2 中避免在 do/while 语句中使用 continue 语句。

6.8 for 语句

for 语句提供了一个循环结构, 这个结构通常比 while 语句更方便。for 语句采用了大多数循环 (包括上例中的 while 循环) 常用的模式。大部分循环都具有某种类型的计数器变量, 在循环开始之前要初始化这个变量, 然后在每次循环开始之前计算 expression 的值, 将该变量作为 expression 的一部分进行检测。最后, 在下次计算 expression 的值之前, 在循环体的尾部要增加计数器变量的值, 或者更新它的值。

初始化、检测和更新是对一个循环变量的三种关键操作, for 语句就将这三步明确地声明为循环语法的一部分。这样可以很容易地理解 for 语句正在做什么, 而且也可以防止忘记初始化或者增加循环变量。for 语句的语法如下:

```
for (initialize ; test ; increment)
    statement;
```

要解释 for 循环是如何操作的, 最简单的方法莫过于列出一个和它等价的 while 循环 (注 3)。

```
initialize;
while (test) {
    statement;
    increment;
}
```

简而言之, 在循环开始之前, 先计算一次表达式 initialize, 这是一个具有副作用的表达式, 通常这个副作用是赋值。JavaScript 也规定 initialize 是一个 var 变量声明语句, 这样就可以同时声明并初始化循环计数器了。每次循环开始之前要先计算表达式 test 的值, 这个值用于判定是否执行循环体。如果 test 的值为 true, 就执行作为循环体的 statement。最后, 计算表达式 increment 的值, 这同样是一个具有副作用的表达式, 通常还是赋值表达式或者使用的是 ++、-- 运算符。

注 3: 我们可以看到, 在使用 continue 时, while 循环与 for 循环并不等价。

前面一节中的while循环的例子可以用如下的for循环重写，其循环计数从0到9：

```
for(var count = 0 ; count < 10 ; count++){  
    document.write(count + "<br>");  
}
```

注意这一语法是如何将所有有关循环变量的重要信息放置在一行中的，这使得循环的执行过程显得非常清楚。此外还要注意，把 *increment* 表达式放置在 for 语句的内部会将循环体简化为一条语句，生成一个语句块时甚至可以不用大括号。

当然，循环比这些简单的例子复杂得多，而且有时循环的一次迭代就会改变多个变量。在JavaScript中，这种情况是逗号运算符的惟一用武之地，它提供了一种方式，使得在for循环中可以将多个初始化表达式和增量表达式合并到一个表达式中。例如：

```
for(i = 0, j = 10 ; i < 10 ; i++, j--)  
    sum += i * j;
```

6.9 for/in

在JavaScript中关键字for有两种使用方式。我们刚刚见过在for循环中如何使用它，此外它还可以用于for/in语句。这个语句是有点特别的循环语句，它的语法如下：

```
for (variable in object)  
    statement
```

*variable*应该是一个变量名，声明一个变量的var语句，数组的一个元素或者是对象的一个属性（例如，它应该是一个适用于赋值表达式左边的值）。*object*是一个对象名，或者是计算结果为对象的表达式。*statement*通常是一个原始语句或者语句块，它构成了循环的主体。

可以使用while循环或者for循环，通过每次循环对下标变量加1来遍历一个数组中的所有元素。而for/in语句则提供了一种遍历对象属性的方法。for/in循环的主体对object的每个属性执行一次。在循环体执行之前，对象的一个属性名会被作为字符串赋给变量*variable*。在循环体内部，你可以使用这个变量和“[]”运算符来查询该对象属性的值。例如，下面的for/in循环将输出一个对象的所有属性名及它的值：

```
for (var prop in my_object) {  
    document.write('name: ' + prop + "; value: " + my_object[prop] + '<br>');  
}
```

注意, `for/in` 循环中的 `variable` 可以是任意的表达式, 只要它的值适用于赋值表达式的左边即可。每一次循环都会计算该表达式的值, 这意味每次计算的值都会不同。例如, 你可以采用下面的代码把一个对象的所有属性名复制到一个数组中:

```
var o = {x:1, y:2, z:3};  
var a = new Array();  
var i = 0;  
for(a[i++] in o) /* 空循环体 */;
```

JavaScript 的数组不过是一种特殊的对象。因此, `for/in` 循环像枚举对象属性一样枚举数组下标。例如, 在前面的代码块后加上下面这行代码可以枚举数组的属性 0、1、2:

```
for(i in a) alert(i);
```

`for/in` 循环并没有指定将对象的属性赋给循环变量的顺序。因为没有什么方法可以预先告之赋值顺序, 所以在不同的 JavaScript 版本或者实现中这一语句的行为可能有所不同。如果 `for/in` 循环的主体删除了一个还没有枚举出的属性, 那么该属性就不再枚举。如果循环主体定义了新属性, 那么循环是否枚举该属性则是由 JavaScript 的实现决定的。

其实, `for/in` 循环并不会遍历所有对象的所有可能的属性。对象的有些属性以相同的方式标记成了只读的、永久的 (不可删除的) 或者不可列举的, 这些属性使用 `for/in` 循环不能枚举出来。虽然所有的用户定义属性都可以枚举, 但是许多内部属性, 包括所有的内部方法都是不可枚举的。我们在第八章中会发现, 对象可以继承其他对象的属性, 那些已继承的用户定义的属性可以使用 `for/in` 循环枚举出来。

6.10 标签语句

和 `switch` 语句联合使用的 `case` 标签和 `default`: 标签不过是普通标签语句的特例罢了。在 JavaScript 1.2 中, 任何语句都可以通过在它前面加上标识符和冒号来标记:

```
identifier: statement
```

`identifier`可以是任何合法的JavaScript标识符，它不能是保留字。由于标签名不同于变量名和函数名，所以如果标签的名字和某个变量名或者函数名相同，那么也不必担心命名冲突。下面是一个加了标签的 `while` 语句：

```
parser:
  while(token != null) {
    // 此处省略了代码
  }
```

通过给一个语句加标签，就可以给这个语句起一个名字，这样在程序的任何地方都可以使用这个名字来引用它。但是被标记的语句通常是那些循环语句，即 `while`、`do/while`、`for` 和 `for/in` 语句。通过给循环命名，就可以使用 `break` 语句和 `continue` 语句来退出循环或者退出循环的某一次迭代。

6.11 break 语句

`break` 语句会使运行的程序立刻退出包含在最内层的循环或者退出一个 `switch` 语句。它的语法非常简单：

```
break;
```

由于它是用来退出循环或者 `switch` 语句，所以只有当它出现在这些语句中时，这种形式的 `break` 语句才是合法的。

ECMAScript v3 和 JavaScript 1.2 允许关键字 `break` 后跟一个标签名：

```
break labelname;
```

注意，`labelname` 只是一个标识符，此时并不像定义一个加标签的语句一样，在其后还要跟一个冒号。

当 `break` 和标签一起使用时，它将跳到这个带有标签的语句的尾部，或者终止这个语句。该语句可以是任何用括号括起来的语句，它不一定是循环语句或者 `switch` 语句，也就是说和标签一起使用的 `break` 语句甚至不必包含在一个循环语句或者 `switch` 语句之中。对 `break` 语句中的标签的唯一的限制就是它命名的是一个封闭语句。例如，这个标签命名的可以是一个 `if` 语句，甚至可以是一个用大括号组合在一起的语句块、封装块的惟一目的是用一个标签对它进行命名。

我们在第二章讨论过，在关键字 `break` 和 `labelname` 之间不允许有换行符。这是

JavaScript 语法的一个古怪之处，它是由于 JavaScript 会自动插入遗漏的分号引起的。如果你在关键字 `break` 和其后的标签之间进行了换行，那么 JavaScript 就假定你要使用的是简单的、不带标签的 `break` 语句，就会为你自动添加一个分号。

我们已经见过了在 `switch` 语句中使用 `break` 语句的例子，它通常在不再需要完成循环时（无论出于什么原因）提前退出这个循环。如果一个循环的终止条件非常复杂，那么使用 `break` 语句来实现某些条件比用一个循环表达式来表达所有的条件要容易得多。

下面的代码是在数组中检索具有特定值的元素。当它运行到数组的结尾时，循环会自然地结束，但是如果它在数组中找到了要检索的元素，那么它会用 `break` 语句来终止循环：

```
for(i = 0; i < a.length; i++) {  
    if (a[i] == target)  
        break;  
}
```

只有当使用嵌套的循环或者使用嵌套的 `switch` 语句，并且需要退出非最内层的语句时才需要使用带标签的 `break` 语句。

下面的例子显示了带标签的 `for` 语句和带标签的 `break` 语句。看一看你能否计算出它的输出：

```
outerloop:  
for(var i = 0; i < 10; i++) {  
    innerloop:  
    for(var j = 0; j < 10; j++) {  
        if (j > 3) break;           // 退出最内层循环  
        if (i == 2) break innerloop; // 与上一条语句的操作相同  
        if (i == 4) break outerloop; // 退出外层循环  
        document.write('i = ' + i + " j = " + j + "<br>");  
    }  
    document.write("FINAL i = " + i + " j = " + j + "<br>");  
}
```

6.12 continue 语句

`continue` 语句和 `break` 语句相似，所不同的是，它不是退出一个循环，而是开始循环的一次新迭代。`continue` 语句的语法和 `break` 语句的语法一样简单：

```
continue;
```

在 ECMAScript v3 和 JavaScript 1.2 中, `continue` 语句还可以和标签一起使用:

```
continue labelname;
```

`continue` 语句 (无论是带标签还是不带标签) 只能用在 `while` 语句、`do/while` 语句、`for` 语句或者 `for/in` 语句的循环体之中, 在其他地方使用它都会引起语法错误。

执行 `continue` 语句时, 封闭循环的当前迭代就会被终止, 开始执行下一次迭代。这对不同类型的循环语句含义是不同的:

- 在 `while` 循环中, 会再次检测循环开头的 `expression`, 如果它的值为 `true`, 将从头开始执行循环体。
- 在 `do/while` 循环中, 会跳到循环的底部, 在顶部开始下次循环之前, 会在此先检测循环条件。但是, JavaScript 1.2 有一个 bug, 它使 `continue` 语句直接跳转到了 `do/while` 语句的顶部, 而无需检测循环条件。因此, 如果你打算在循环中使用 `continue` 语句, 就应该避免使用 `do/while` 循环, 除非你可以确定自己用户的浏览器绝对没有 bug。但是这个问题并不严重, 因为总可以使用一个等价的 `while` 循环来替换 `do/while` 循环。
- 在 `for` 循环中, 先计算 `increment` 表达式, 然后再检测 `test` 表达式以确定是否应该执行下一次迭代。
- 在 `for/in` 循环中, 将以下一个赋给循环变量的属性名再次开始新的迭代。

注意在 `while` 循环和 `for` 循环中 `continue` 语句的行为的不同之处。在 `while` 循环中, 它直接跳转到循环条件处, 而在 `for` 循环中则要先计算 `increment` 表达式, 然后再跳转到循环条件处。前面在讨论 `for` 循环时, 使用了一个等价的 `while` 循环解释 `for` 循环的行为。因为在这两种循环中 `continue` 语句的行为不同, 所以用一个 `while` 循环来完全模拟一个 `for` 循环是不可能的。

下面的例子展示了一个不带标签的 `continue` 语句, 它用于发生错误时退出循环的当前迭代:

```
for(i = 0; i < data.length; i++) {
```

```
    if (data[i] != null)
        continue; // 不能处理 undefined 数据
    total += data[i];
}
```

和break语句一样，当要重新开始的循环不是直接封闭的循环时，在嵌套的循环中也可以使用这种带标签形式的continue语句。而且，在continue语句和labelname之间也不允许有换行符。

6.13 var 语句

var 语句允许你明确的声明一个或多个变量。它的语法如下：

```
var name_1 [= value_1] [, ..., name_n [= value_n];
```

关键字var之后跟随的是一个要声明的变量的列表，变量之间用逗号分隔。在这个列表中，每一个变量都可选地具有一个初始化表达式，用于指定它的初始值。例如：

```
var i;
var j = 0;
var p, q;
var greeting = 'hello' + name;
var x = 2.34, y = Math.cos(0.75), r, theta;
```

var 语句通过在封闭函数的调用对象中创建一个同名的属性来定义每个变量，如果变量的声明不在函数体内，那么可以在全局对象中创建同名属性来定义每个变量。由var语句创建的一个特性或多个特性不能用delete运算符来删除。注意，将var语句封闭在一个with语句中（参见6.18小节）并不会改变这一行为。

如果在var语句中没有为变量指定初始值，那么虽然定义了该变量，但是它的初始值是undefined。

注意，var 语句不能作为for循环和for/in循环的一部分。例如：

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");
for(var i = 0, j=10; i < 10; i++, j--) document.write(i*j, "<br>");
for(var i in o) document.write(i, "<br>");
```

第四章中包含了更多有关JavaScript变量和变量声明的内容。

6.14 function 语句

function 语句定义了一个 JavaScript 函数。它的语法如下：

```
function funcname([arg1 [,arg2 [..., argn]]]) {  
    statements  
}
```

funcname 是要定义的函数名，它必须是一个标识符，而不是字符串或表达式。函数名后跟随的是一个用括号括起来的参数名列表，参数名之间用逗号隔开。当函数被调用时，这些标识符可以在函数主体内部引用传递进来的参数值。

函数主体是由 JavaScript 语句构成的，其中语句的数量不限，它们用大括号括起来。这些语句在函数定义时不会被执行。相反，只有在使用函数调用运算符()调用了函数时，这些语句才会被编译，并且和用于执行的新的函数对象关联起来。注意，在 function 语句中，大括号是必需的。这和 while 循环和其他语句所使用的语句块不同，function 语句的主体必须使用大括号，即使主体只由一条语句构成。

一条函数定义创建一个新的函数对象，并且将这个函数对象存储在一个新创建的名称为 *funcname* 的属性中。下面是一些函数定义的例子：

```
function welcome() { alert("Welcome to my home page!"); }  
function print(msg) {  
    document.write(msg, '<br>');  
}  
  
function hypotenuse(x, y) {  
    return Math.sqrt(x*x + y*y); // 下面说明了返回值  
}  
  
function factorial(n) { // 递归函数  
    if (n <= 1) return 1;  
    return n * factorial(n - 1);  
}
```

函数定义通常出现在 JavaScript 代码的顶层。它们也可以嵌套在其他函数定义中，但是只能嵌套在函数顶层定义中，也就是说，函数定义不能出现在 if 语句，while 循环或其他任何语句中。

从技术上说，function 语句并非是一个语句。在 JavaScript 程序中，语句会引发动态的行为，但是函数定义描述的却是静态的程序结构。语句是在运行时执行的，而

函数则是在实际运行之前，当JavaScript代码被解析或者被编译时定义的。当JavaScript解析程序遇到一个函数定义时，它就解析并存储（而无需执行）构成函数主体的语句，然后定义一个和该函数同名的属性（如果函数定义嵌套在其他函数中，那么就在调用对象中都定义这个属性，否则在全局对象中定义这个属性）以保存它。

函数定义在解析时发生，而不是在运行时发生，这一事实产生了某些令人吃惊的作用。考虑如下的代码：

```
alert(f(4));           // 显示16。可以在定义f()前调用它。
var f = 0;             // 该语句重写属性f。
function f(x) {        // 该语句定义前两行执行的函数f。
    return x*x;        // 执行上面的语句行。
}
alert(f);              // 显示0。函数f()已经被变量f覆盖了。
```

出现这种特殊的结果是由于函数定义和变量定义发生在不同时刻。幸运的是，这种情况并不常常发生。

我们将在第七章中学习更多有关函数的内容。

6.15 return 语句

也许你还记得，用运算符`()`调用函数是一个表达式。所有表达式都有一个值，`return`语句就用于指定函数返回的值。这个值是函数调用表达式的值。`return`语句的语法如下：

```
return expression;
```

`return`语句只能出现在函数体内。出现在代码中的其他地方都会造成语法错误。在执行`return`语句时，先计算`expression`，然后返回它的值作为函数的值。当执行`return`语句时，即使函数主体中还有其他语句，函数的执行也会停止。可以采用如下的方式使用`return`语句返回值：

```
function square(x) { return x*x; }
```

`return`语句还可以不带`expression`来终止程序的执行，并不返回值。例如：

```
function display_object(obj) {  
    // 首先确保参数有效  
    // 如果参数无效, 则跳过函数其余的部分  
    if (obj == null) return;  
    // 此处开始是函数其余的部分  
}
```

如果一个函数执行了不带 *expression* 的 `return` 语句, 或者因为它执行到了函数主体的尾部而返回, 那么这个函数调用的表达式的值就是 `undefined`。

由于JavaScript会自动插入分号, 所以在 `return` 关键字和其后的表达式之间不要用换行符。

6.16 throw 语句

所谓异常 (*exception*) 是一个信号, 说明发生了某种异常状况或错误。抛出 (`throw`) 一个异常, 就是用信号通知发生了错误或异常状况。捕捉 (`catch`) 一个异常, 就是处理它, 即采取必要或适当的动作从异常恢复。在JavaScript中, 当发生运行时错误或程序明确地使用 `throw` 语句时就会抛出异常。使用 `try/catch/finally` 语句可以捕捉异常, 下一节将对它们进行介绍 (注4)。

`throw` 语句的语法如下:

```
throw expression;
```

expression 的值可以是任何类型的。但通常它是一个 `Error` 对象或 `Error` 子类的实例。抛出一个存放错误信息的字符串或代表某种错误代码的数字也很有用。下面是一些使用 `throw` 语句抛出异常的示例代码:

```
function factorial(x) {  
    // 如果输入参数无效, 则抛出该异常!  
    if (x < 0) throw new Error("x must not be negative");  
    // 否则计算一个值, 正常返回  
    for (var i = 1; x > 1; i *= x, x--) /* 空的循环体 */ ;  
    return i;  
}
```

注4: JavaScript 的 `throw` 和 `try/catch/finally` 语句与 C++ 和 Java 中的 `throw` 和 `try/catch/finally` 语句相似, 但是不完全相同。

在抛出异常时，JavaScript 解释器会立刻停止正常的程序执行，跳转到最近的异常处理器。异常处理器是用 try/catch/finally 语句的 catch 从句编写的，下一节会介绍它。如果抛出异常的代码块没有相关的 catch 从句，解释器将检查次高级的封闭代码块，看它是否有相关的异常处理器。以此类推，直到找到一个异常处理器为止。如果抛出异常的函数没有处理它的 try/catch/finally 语句，异常将向上传播到调用该函数的代码。在这种情况下，异常将沿着 JavaScript 方法的词法结构和调用堆栈向上传播。如果没有找到任何异常处理器，将把异常作为错误处理，报告给用户。

throw 语句由 ECMAScript v3 标准化，由 JavaScript 1.4 实现。Error 类和它的子类也是 ECMAScript v3 的一部分，但是直到 JavaScript 1.5 才实现了它们。

6.17 try/catch/finally

try/catch/finally 语句是 JavaScript 的异常处理机制。该语句的 try 从句只定义异常需要被处理的代码块。catch 从句跟随在 try 块后，它是在 try 块内的某个部分发生了异常时调用的语句块。finally 块跟随在 catch 从句后，存放清除代码，无论 try 块中发生了什么，该代码块都会被执行。虽然 catch 块和 finally 块都是可选的，但是 try 块中至少应该有一个 catch 块或 finally 块。try、catch 和 finally 块都以大括号开头和结尾。这是必须的语法部分，即使从句只有一条语句，也不能省略大括号。和 throw 语句一样，try/catch/finally 语句也是由 ECMAScript v3 标准化的，在 JavaScript 1.4 中实现的。

接下来的代码说明了 try/catch/finally 语句的语法和目的。尤其要注意 catch 关键字后用括号括起来的标识符。该标识符就像函数的参数，它指定了一个仅存在于 catch 块内部的局部变量。JavaScript 将把要抛出的异常对象或值赋给这个变量：

```
try {  
    // 通常，该代码从代码块的顶部运行到底部  
    // 没有任何问题，但有时它会抛出异常。  
    // 既可以用 throw 语句直接抛出，也可以调用一个抛出异常的方法间接抛出。  
}  
catch (e) {  
    // 当且仅当 try 块抛出了异常，本块中的语句才会被执行。  
    // 这些语句使用局部变量 e 引用抛出的 Error 对象或其他值。  
    // 这个块可以以某种方式处理异常，或者什么都不做，
```

```
    // 忽略异常，或者用 throw 语句再抛出一个异常。
}
finally {
    // 无论 try 块中发生了什么，这个块中的语句都会被执行。
    // try 块是否终止，它们都会执行：
    //    1) 正常地，在达到 try 块底部后
    //    2) 由于 break、continue 或 return 语句终止
    //    3) 抛出一个异常，由 catch 从句处理
    //    4) 抛出一个异常，没有被捕捉，仍旧在传播
}
```

下面是更实际的 try/catch 语句的例子。它使用了前面定义的 factorial() 方法和客户端 JavaScript 的方法 prompt() 及 alert() 进行输入和输出：

```
try {
    // 要求用户输入一个数字
    var n = prompt("Please enter a positive integer", "");
    // 计算该数字的阶乘，假定用户输入的是有效数字
    var f = factorial(n);
    // 显示结果
    alert("n + ! = " + f);
}
catch (ex) { // 如果用户输入的数字无效，在此处结束
    // 通知用户发生了什么错误
    alert(ex);
}
```

这个例子是没有 finally 从句的 try/catch 语句。虽然 finally 从句不像 catch 从句那么常用，但是它也很有用。只是它的行为需要更多的解释。只要执行了 try 块的一部分，无论 try 块的代码被完成了多少，finally 从句都会被执行。它通常在 try 从句的代码后用于清除操作。

通常情况下，控制流到达 try 块的尾部，然后开始执行 finally 块，以便进行清除操作。如果由于 return 语句、continue 语句或 break 语句使控制流离开了 try 块，那么在控制流转移到新目的地前，finally 块就会被执行。

如果异常发生在 try 块中，而且存在一个相关的 catch 块处理异常，控制流首先将转移到 catch 块，然后再转移到 finally 块。如果没有处理异常的局部 catch 块，控制流首先将转移到 finally 块，然后向上传播到最近的能够处理异常的 catch 从句。

如果 finally 块自身用 return 语句、continue 语句、break 语句或 throw 语句

转移了控制流，或者调用了抛出异常的方法改变了控制流，那么等待的控制流转移将被舍弃，并进行新的转移。例如，如果 `finally` 从句抛出了一个异常，那么该异常将代替处于抛出过程中的异常。如果 `finally` 从句运行到了 `return` 语句，那么即使已经抛出了一个异常，而且该异常还没有被处理，该方法也会正常返回。

`try` 从句可以在没有 `catch` 从句的情况下和 `finally` 从句一起使用。在这种情况下，`finally` 块中只包括清除代码，无论 `try` 从句中是否有 `break` 语句、`continue` 语句和 `return` 语句，这些代码都一定会被执行。例如，下面的代码使用 `try/finally` 语句确保循环计数器变量在每次迭代的末尾加 1，即使末次迭代由于 `continue` 语句突然终止了：

```
var i = 0, total = 0;
while(i < a.length) {
    try {
        if ((typeof a[i] != 'number') || isNaN(a[i])) // 如果它不是数字，
            continue; // 进行循环的下一迭代
        total += a[i]; // 否则把该数字加到 total 上。
    }
    finally {
        i++; // 总是增加 i 的值，即使上面的语句中使用了 continue 语句
    }
}
```

6.18 with 语句

在第四章中我们讨论过用变量的作用域和作用域链（即一个按顺序检索的对象列表）来进行变量名解析。`with` 语句用于暂时修改作用域链的。它的语法如下：

```
with (object)
    statement
```

这一语句能够有效地将 `object` 添加到作用域链的头部，然后执行 `statement`，再把作用域链恢复到原始状态。

在实际应用中，使用 `with` 语句可以减少大量的输入。例如，在客户端的 JavaScript 中，深度嵌套的对象层次很常用。例如，你可以输入如下的表达式来访问一个 HTML 表单的元素：

```
frames[1].document.forms[0].address.value
```

如果需要多次访问这个表单，可以使用 with 语句将这个表单添加到作用域链中：

```
with(frames[1].document.forms[0]) {  
    // 此处直接访问表单元素。例如：  
    name.value = '';  
    address.value = '';  
    email.value = '';  
}
```

这就减少了输入量，因为不必在每个表单属性名前都加前缀 frame[1].document.forms[0] 了。这个对象不过是作用域链的一个临时部分，当 JavaScript 需要解析像 address 这样的标识符时就会自动搜索它。

虽然有时使用 with 语句比较方便，但是人们反对使用它。使用了 with 语句的 JavaScript 代码很难优化，因此它的运行速度比不使用 with 语句的等价代码要慢得多。而且，在 with 语句中的函数定义和变量初始化可能会产生令人惊讶的、相抵触的行为（注 5）。因此，我们建议避免使用 with 语句。

注意，还有其他的、极为合理的方法可以用来节省输入。例如，上面使用 with 语句可重写为：

```
var form = frames[1].document.forms[0];  
form.name.value = '';  
form.address.value = '';  
form.email.value = '';
```

6.19 空语句

JavaScript 中的最后一个合法语句是空语句。它如下所示：

```
;
```

执行空语句显然不会产生任何作用，也不会执行任何动作。你可能认为使用这样一个语句是毫无意义的，但实践证明，当你想创建一个具有空主体的循环时，空语句是有用的。例如：

注 5：这一行为以及产生它的原因非常复杂，在这里我们不作解释。

```
// 初始化数组a
for(i=0; i < a.length; a[i++] = 0) ;
```

注意，在 for 循环、while 循环或者 if 语句的右括号后加分号会产生 bug，这些 bug 很难被检测出来。例如，下面的代码产生的结果并不是作者想要的：

```
if ((a == 0) || (b == 0)) : // 这行什么都不做
    c = null;                // 这行总会被执行
```

当你打算使用空语句时，最好在代码中使用注释，以清楚地说明是有目的地这样做。例如：

```
for(i=0; i < a.length; a[i++] = 0) /* 空函数体 */ ;
```

6.20 JavaScript 语句小结

本章介绍了 JavaScript 语言的所有语句。表 6-1 总结了这些语句以及它们的语法和用途。

表 6-1: JavaScript 语句的语法

| 语句 | 语法 | 用途 |
|----------|--|---|
| break | break;
break labelname; | 退出最内层循环或者退出 switch 语句，又或者退出 label 定指的语句 |
| case | case expression: | 在 switch 语句中标记一个语句 |
| continue | continue;
continue labelname; | 重新开始最内层循环或者重新开始 label 指定的循环 |
| default | default: | 在 switch 中标记默认语句 |
| do/while | do
statement
while(expression); | while 循环的一种替代形式 |
| 空语句 | ; | 什么都不做 |
| for | for (initialize ; test ; increment)
statement | 一种易用的循环 |

表 6-1: JavaScript 语句的语法 (续)

| 语句 | 语法 | 用途 |
|----------|---|-----------------------------------|
| for/in | for (variable in object)
statement | 遍历一个对象的属性 |
| function | function funcname([arg1[...], argn])
{
statements
} | 声明一个函数 |
| if/else | if (expression)
statement1
[else statement2] | 有条件的执行代码 |
| label | identifier: statement | 给 statement 指定一个名字
identifier |
| return | return [expression]; | 由一个函数返回或者由函数
返回 expression 的值 |
| switch | switch (expression) {
statements
} | 用 case 或者 default: 语句
标记的多分支语句 |
| throw | throw expression; | 抛出一个异常 |
| try | try {
statements
}
catch (identifier) {
statements
}
finally {
statements
} | 捕捉一个异常 |
| var | var name_1 [= value_1]
[..., name_n [= value_n]]; | 声明或者初始化变量 |
| while | while (expression)
statement | 一种基本循环结构 |
| with | with (object)
statement | 扩展当前作用域链 (不赞成) |

第七章

函数

函数是JavaScript语言一个既重要又复杂的部分。本章将详细介绍函数的各个方面。首先，我们从语法的角度来讨论函数，其中介绍了函数的定义和调用。然后，我们将函数作为一种数据类型来讨论，其中介绍了一种有用的程序设计方法的例子，这种方法只有在把函数用作数据的情况下才能使用。最后，我们讨论了在函数主体中变量的作用域，此外还要检测一些和函数相关的有用属性，执行函数可以使用这些属性。其中介绍了如何编写可以接受任意多个参数的JavaScript函数。

本章的重点是用户定义的JavaScript函数的定义和调用。另外还有一点比较重要，即JavaScript支持很多内部函数，诸如类Array的方法`eval()`、`parseInt()`和`sort()`等。客户端JavaScript还定义了其他函数，如`document.write()`和`alert()`。在JavaScript中，完全可以像使用用户定义的函数那样使用内部函数。你可以在本书的核心参考手册部分和客户端参考手册中找到更多有关内部函数的信息。

在JavaScript中，函数和对象是交织在一起的。因此，我们将某些函数特性的讨论推迟到第8章进行。

7.1 函数的定义和调用

我们在第六章中见到过，定义函数最常用的方法就是调用`function`语句。该语句是由关键字`function`构成的，它后面跟随的是：

- 函数名
- 一个用括号括起来的参数列表，这个列表是可选的，其中的参数用逗号分隔开
- 构成函数主体的 JavaScript 语句，包含在大括号中

例 7-1 展示了几个函数的定义。虽然这些函数比较短小，而且又很简单，但是它们都含有上面列出的所有元素。注意，定义函数时可以使用个数可变的参数，而且函数既可以有 return 语句，也可以没有 return 语句。我们在第六章中介绍过 return 语句，它能使函数停止运行，并且把表达式的值（如果存在这样的表达式）返回给函数调用者。如果函数不包含 return 语句，它就只执行函数体中的每条语句，然后返回给调用者 undefined。

例 7-1: JavaScript 函数的定义

```
// 有时，快捷函数比 document.write () 方法更有用
// 由于该函数没有 return 语句，所以它没有返回值
function print(msg)
{
    document.write(msg, '<br>');
}
// 以下是计算并返回两点之间距离的函数
function distance(x1, y1, x2, y2)
{
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}
// 以下是一个递归函数（调用自身的函数），它将计算阶乘的值
// 回忆一下，x! 就是 x 和所有小于它的正整数的乘积
function factorial(x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
```

如果一个函数已经被定义了，那么就可以使用运算符 () 来调用它，这个运算符在第五章中介绍过。回忆一下出现在函数名之后的括号和括号中用逗号分隔的可选的参数值（或表达式）列表。可以使用如下的代码调用例 7-1 中定义的函数：

```
print("Hello, " + name);
print("Welcome to my home page!");
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
print('The probability of that is: ' + factorial(39)/factorial(52));
```

在调用函数时，先要计算括号之间指定的所有表达式，然后把它们的结果作为函数的参数。这些值将被赋予函数定义时指定的形式参数，然后函数通过参数名来引用这些参数，对它们进行操作。注意，这些参数变量只有在执行函数的时候才会被定义，一旦函数返回，它们就不再存在。

因为JavaScript是一种无类型语言，所以你不能给函数的参数指定一个数据类型，而且JavaScript也不会检测传递的数据是不是那个函数所要求的类型。如果参数的数据类型很重要，那么可以用运算符 `typeof` 对它进行检测。JavaScript也不会检测传递给它的参数个数是否正确。如果传递的参数比函数需要的个数多，那么多余的值会被忽略掉。如果传递的参数比函数需要的个数少，那么多余的几个参数就会被赋予 `undefined`，在大多数情况下，这会使函数产生运行错误。在本章后面的小节中，我们将学习一种方法，使用它就可以检测传递给函数的参数个数是否正确。

注意，函数 `print()` 不包含 `return` 语句。所以它返回的总是 `undefined`，这对于较复杂的表达式来说是没有意义的。但是函数 `distance()` 和 `factorial()` 就可以用作较复杂的表达式的一部分，就像在前面的例子中所示的那样。

7.1.1 嵌套的函数

ECMAScript v1和JavaScript 1.2之前的实现只允许在顶层全局代码中定义函数。但是在JavaScript 1.2和ECMAScript v3中，函数定义可以嵌套在其他函数中。例如：

```
function hypotenuse(a, b) {  
    function square(x) { return x*x; }  
    return Math.sqrt(square(a) + square(b));  
}
```

注意，ECMAScript v3不允许函数定义任意出现，它们仍然被限制在顶层全局代码和顶层函数代码中。这意味着函数定义不能出现在循环或条件语句中（注1）。这些对函数定义的限制只应用于由 `function` 语句声明的函数。在本章后面的小节我们会讨论函数直接量（JavaScript 1.2引入的、ECMAScript v3标准化的另一个特性），它可以出现在任何JavaScript表达式中，也就是说，它可以出现在 `if` 语句和其他语句中。

注1：不同的JavaScript实现关于函数定义的要求比标准要求得松一些。例如，JavaScript 1.5的Netscape实现允许在 `if` 语句中使用条件函数定义。

7.1.2 Function()构造函数

function语句并非是定义新函数的惟一方法。在ECMAScript v1和JavaScript 1.1中,还可以使用Function()构造函数和new运算符动态地定义函数(我们在第五章中介绍过运算符new,将在第八章中学习构造函数)。下面是使用上述方法创建函数的例子:

```
var f = new Function('x', 'y', "return x*y;");
```

这行代码创建了一个新函数,该函数和用你所熟悉的语法定义的函数基本上是等价的:

```
function f(x, y) { return x*y; }
```

Function()构造函数可以接受任意多个字符串参数。它的最后一个参数是函数的主体,其中可以包含任何JavaScript语句,语句之间用分号分隔。其他的参数都是用来说明函数要定义的形式参数名的字符串。如果你定义的函数没有参数,那么可以只需给构造函数传递一个字符串(即函数的主体)即可。

注意,传递给构造函数Function()的参数中没有一个是用于说明它要创建的函数名。用Function()构造函数创建的未命名函数有时被称为“匿名函数”。

你可能非常想知道Function()构造函数的用途是什么。为什么不能只用function语句来定义所有的函数呢?原因是Function()构造函数允许我们动态地建立和编译一个函数,它不会将我们限制在function语句预编译的函数体中。这样做带来的负面效应就是每次调用一个函数时,Function()构造函数都要对它进行编译。因此,在循环体中或者在经常使用的函数中,我们不想频繁地调用这个构造函数。

使用Function()构造函数的另一个原因是它能将函数定义为JavaScript表达式的一部分,而不是将其定义为一个语句,这种情况下使用它就显得比较方便,甚至可以说是精致。我们将在本章后面的部分看到一些例子。在JavaScript 1.2中,当你想在一个表达式中定义一个函数,而不是在一个语句中定义它时,使用函数直接量比使用Function()构造函数更方便。我们将在下面一节中讨论函数直接量。

7.1.3 函数直接量

ECMAScript v3定义了函数直接量,JavaScript 1.2实现了它,它是第三种创建函

数的方式。我们在第三章中讨论过，函数直接量是一个表达式，它可以定义匿名函数。函数直接量的语法和 `function` 语句的非常相似，只不过它被用作表达式，而不是用作语句，而且也无需指定函数名。下面的三行代码分别使用 `function` 语句、`Function()` 构造函数和函数直接量定义了三个基本上相同的函数：

```
function f(x) { return x*x; }           // function 语句
var f = new Function("x", "return x*x;"); // Function() 构造函数
var f = function(x) { return x*x; };    // 函数直接量
```

虽然函数直接量创建的是未命名函数，但是它的语法也规定它可以指定函数名，这在编写调用自身的递归函数时非常有用。例如：

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };
```

上面的代码定义了一个未命名函数，并把对它的引用存储在变量 `f` 中。它并没有真正创建一个名为 `fact()` 的函数，只是允许函数体用这个名字来引用自身。但是要注意，JavaScript 1.5 之前的版本中没有正确实现这种命名了的函数直接量。

记住，`function` 语句在所有的 JavaScript 版本中都是有效的，而 `Function()` 构造函数只在 JavaScript 1.1 和其后的版本中有效，函数直接量只在 JavaScript 1.2 和其后的版本中有效。注意，我们说上面的代码中定义的三个函数“几乎”是等价的，但是这三种函数定义的方法还是存在差异的，我们将在 11.5 节中讨论这一点。

函数直接量的用法和用 `Function()` 构造函数创建函数的方法非常相似。由于它们都是由 JavaScript 的表达式创建的，而不是由语句创建的，所以使用它们的方式也就更加灵活，尤其适用于那些只使用一次，而且无需命名的函数。例如，一个使用函数直接量表达式指定的函数可以存储在一个变量中、传递给其他的函数甚至被直接调用：

```
a[0] = function(x) { return x*x; }; // 定义一个函数并保存它
a.sort(function(a,b){return a-b;}); // 定义一个函数：把它传递给另一个函数
var tensquared = (function(x) {return x*x;})(10); // 定义并调用一个函数
```

和 `Function()` 构造函数一样，函数直接量创建的是未命名函数，而且不会自动地将这个函数存储在属性中。但是，比起 `Function()` 构造函数来说，函数直接量有一个重要的优点。由 `Function()` 构造函数创建的函数的主体必须用一个字符串说明，用这种方式来表达一个长而复杂的函数是很笨拙的。但是函数直接量的主体使用的却是标准的 JavaScript 语法。而且函数直接量只被解析和编译一次，而作为字

字符串传递给 `Function()` 构造函数的 JavaScript 代码则在每次调用构造函数时只会被解析和编译一次。

7.2 作为数据的函数

函数最重要的特性就是它们能够被定义和调用，这一点我们在前一节中已经说明过了。函数的定义和调用是 JavaScript 和大多数程序设计语言的语法特性。但是，在 JavaScript 中，函数并不只是一种语法，还可以是数据，这意味着能够把函数赋给变量、存储在对象的属性中或存储在数组的元素中、传递给函数，等等（注 2）。

要理解函数是如何作为数据及 JavaScript 语法的，请考虑如下的函数定义：

```
function square(x) { return x*x; }
```

这个定义创建了一个新的函数对象，并且把这个对象赋给了变量 `square`。实际上，函数名并没有什么实质意义，它不过是用来保存函数的变量的名字。可以将这个函数赋给其他的变量，它仍然会以相同的方式起作用：

```
var a = square(4); // a 存放数字 16
var b = square;    // 现在 b 引用的函数和 square 的作用相同
var c = b(5);      // c 存放数字 25
```

除了赋给全局变量之外，还可以将函数赋给对象的属性。在这种情况下，我们称函数为方法：

```
var o = new Object;
o.square = new Function("x", "return x*x"); // 注意 Function() 构造函数
y = o.square(16);                          // y 等于 256
```

函数可以没有函数名，就像我们将函数赋给数组元素时那样：

```
var a = new Array(3);
a[0] = function(x) { return x*x; } // 注意函数直接量
a[1] = 20;
a[2] = a[0](a[1]);                  // a[2] 存放 400
```

注 2：除非你熟悉 Java 这样的语言，了解函数是程序的一部分，但不能由程序操作，否则这一点对你来说没有什么特别之处。

虽然上例使用的函数调用的语法比较奇怪,但它仍旧是JavaScript()运算符的合法用法。

例7-2是一个详细的例子,其中展示了将函数作为数据使用时的所有用法。它说明了如何将函数作为参数传递给其他函数,还说明了如何将它们存储在关联数组中(我们在第三章中介绍过关联数组,而且还将在第八章中对其进行详细说明)。这个例子可能比较烦琐,但是其中的注释说明了要实现的是什么,所以它还是值得仔细研究一下的。

例7-2: 将函数作为数据的用法

```
// 此处我们定义了一些简单的函数。
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }
// 以下的函数可以将上面的某个函数作为参数,
// 它的两个参数是两个运算数。
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// 我们可以调用该函数来计算值(2+3) - (4*5):
var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// 考虑到本例的需要,我们再次实现了这些函数,此次使用的函数直接量。
// 我们把这些函数存放在一个关联数组中。
var operators = new Object();
operators['add'] = function(x,y) { return x+y; };
operators["subtract"] = function(x,y) { return x-y; };
operators['multiply'] = function(x,y) { return x*y; };
operators['divide'] = function(x,y) { return x/y; };
operators['pow'] = Math.pow; // 也用于预定义函数。

// 以下的函数将运算符名作为参数,在数组中检索这个运算符,
// 然后对运算数调用检索到的函数。
// 注意调用这个运算符函数的语法。
function operate2(op_name, operand1, operand2)
{
    if (operators[op_name] == null) return "unknown operator";
    else return operators[op_name](operand1, operand2);
}

// 我们可以采用如下方式调用该函数计算值("hello" + " " + "world"):
var j = operate2('add', "hello", operate2("add", " ", "world"));
// 使用预定义的Math.pow()函数:
var k = operate2('pow', 10, 2)
```

如果前面的例子还不能使你确信能够将函数作为参数传递给其他函数或者把函数作为数值来处理，那么就考虑一下 `Array.sort()` 函数。这个函数是对数组的元素进行排序。由于排序可依据的方式有很多（如按数字顺序、字母顺序、日期顺序、升序、降序等），所以函数 `sort()` 需要另一个函数作为它的参数来告诉它以何种方式执行排序。作为参数的函数的工作非常简单，它采用两个数组元素，比较这两个元素，然后返回一个值来说明哪个元素排在前面即可。该函数参数使方法 `Array.sort()` 具有极佳的通用性和极大的灵活性，使用它就可以将任何类型的数据排成所有可能想到的顺序。

7.3 函数的作用域：调用对象

我们在第四章中介绍过，JavaScript 函数的主体是在局部作用域中执行的，该作用域不同于全局作用域。这个新作用域是通过把调用对象添加到作用域链的头部创建的。因为调用对象是作用域链的一部分，所以在函数体内可以把这个对象属性作为变量来访问。用 `var` 语句声明的局部变量创建后作为调用对象的属性，而且函数的形式参数也可用于对象的属性。

除了局部变量和形式参数外，调用对象还定义了一个特殊属性，名为 `arguments`。这个属性引用了另外一个特殊的对象——`Arguments` 对象，我们将在下一节中讨论这个对象。因为 `arguments` 属性是调用对象的一个属性，所以它的状态和局部变量以及函数的形式参数是相同的。出于这种原因，标识符 `arguments` 应该被看作保留字，不能将它作为变量名或形式参数名。

7.4 函数的实际参数：Arguments 对象

在一个函数体内，标识符 `arguments` 具有特殊含义。它是调用对象的一个特殊属性，用来引用 `Arguments` 对象。`Arguments` 对象就像数组，可以按照数字获取传递给函数的参数值。但它并非真正的 `Array` 对象。`Arguments` 对象也定义了 `callee` 属性，将在后面介绍该属性。

尽管定义 JavaScript 函数时有固定数目的命名参数，但当调用这个函数时，传递给它的参数数目却可以是任意的。数组 `arguments[]` 允许完全地存取那些实际参数

值，即使某些参数还没有被命名。假定你定义了一个函数 `f`，要传递给它一个实际参数 `x`。如果你用两个实际参数来调用这个函数，那么在函数体内，用形式参数名 `x` 或 `arguments[0]` 可以存取第一个实际参数。而第二个实际参数只能通过 `arguments[1]` 来存取。而且和所有数组一样，`arguments` 具有 `length` 属性，用于说明它所含有的元素个数。因此，在函数 `f` 的主体内，如果调用时使用的是两个实际参数，那么 `arguments.length` 的值就是 2。

数组 `arguments[]` 可以用于多个方面。下面的例子说明了如何使用它来检测调用函数时是否使用了正确数目的实际参数，因为 JavaScript 不会替你做这项检测：

```
function f(x, y, z)
{
    // 首先检查传递的参数数量是否正确。
    if (arguments.length != 3) {
        throw new Error('function f called with ' + arguments.length +
            "arguments, but it expects 3 arguments.");
    }
    // 下面运行真正的函数。
}
```

数组 `arguments[]` 还为 JavaScript 函数开发了一项重要的可能性，即可以编写函数使之能够使用任意数目的实际参数。下面的例子说明了如何编写一个简单的 `max()` 函数，让它能够接受任意数目的实际参数，然后返回其中最大的参数的值（参阅内部函数 `Math.max()`，在 ECMAScript v3 中，它也可以接受任意数目的实际参数）：

```
function max()
{
    var m = Number.NEGATIVE_INFINITY;
    // 遍历所有参数，检索并保存其中最大的参数
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > m) m = arguments[i];
    // 返回最大的参数
    return m;
}
var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

也可以使用 `arguments[]` 数组来编写一个函数，它需要固定数目的命名参数，而這些参数后还可以有任意数目的未命名参数。

在这一节中，我们多次用到了“`arguments` 数组”。但是请记住，`arguments` 并非真正的数组，它是一个 `Arguments` 对象。虽然每个 `Arguments` 对象都定义了带编码的数组元素和 `length` 属性，但是它不是数组，将它看作偶然具有了一些带编码的

属性的对象更合适一些。ECMAScript 标准没有要求 Arguments 对象实现数组的所有特殊行为。例如，虽然可以给 arguments.length 属性赋值，但是 ECMAScript 并不要求你这样做来改变对象中定义的数组元素数。（参阅第九章关于真正的 Array 对象的 length 属性的特殊行为的解释。）

Arguments 对象有一个非同寻常的特性。当函数具有命名了的参数时，Arguments 对象的数组元素是存放函数参数的局部变量的同义词。arguments[] 数组和命名了的参数不过是引用同一变量的两种不同方法。用参数名改变一个参数的值同时会改变通过 arguments[] 数组获得的值。通过 arguments[] 数组改变参数的值同样会改变用参数名获取的参数值。例如：

```
function f(x) {  
    alert(x);           // 显示参数的初始值  
    arguments[0] = null; // 改变数组元素也会改变 x  
    alert(x);           // 现在显示 "null"  
}
```

7.4.1 属性 callee

除了数组元素，Arguments 对象还定义了 callee 属性，用来引用当前正在执行的函数。这对未命名的函数调用自身非常有用。下面是一个未命名的函数直接量，用于计算阶乘：

```
function(x) {  
    if (x <= 1) return 1;  
    return x * arguments.callee(x-1);  
}
```

属性 callee 是 ECMAScript v1 定义的，由 JavaScript 1.2 实现。

7.5 函数的属性和方法

我们已经看到，在 JavaScript 程序中，函数可以用作数值，而且我们还知道可以使用 Function() 构造函数来创建函数。这些都标志着函数实际上是用一种 JavaScript 对象，即 Function 对象表示的。由于函数是对象，所有它们具有属性和方法，就像 String 对象和 Date 对象一样。既然我们已经讨论过了调用对象和函数调用环境中用到的 Arguments 对象，那么下面我们就讨论 Function 对象本身。

7.5.1 属性 length

我们已经知道，在函数主体中，arguments 数组的 length 属性指定了传递给该函数的实际参数数目。但是函数自身的 length 属性的含义却非如此，它是只读特性，返回的是函数需要的实际参数的数目，也就是在函数的形式参数列表中声明的形式参数的数目。回忆一下就会知道，调用函数时可以传递给它任意数目的实际参数，函数能够从 arguments 数组中得到这些参数，而无须考虑它所声明的形式参数的数目。Function 对象的 length 属性确切地说明了一个函数声明的形式参数的个数。注意，和 arguments.length 不同，这个 length 属性在函数体的内部和外部都有效。

接下来的代码定义了一个名为 check() 的函数，它的 arguments 数组是由另一个函数传递的。它通过比较 arguments.length 属性和 Function.length 属性（通过 arguments.callee.length 访问）来判断传递给该函数的参数个数是否符合要求。如果不是，它将抛出一个异常。函数 check() 后是一个检测函数 f()，它说明了如何使用 check() 函数：

```
function check(args) {
    var actual = args.length;           // 实际的参数个数
    var expected = args.callee.length; // 要求的参数个数
    if (actual != expected) {           // 如果它们不匹配，则抛出异常
        throw new Error("Wrong number of arguments: expected: " +
                        expected + "; actually passed " + actual);
    }
}

function f(x, y, z) {
    // 检查实际的参数个数与期望的参数个数是否匹配
    // 如果它们不匹配，则抛出异常
    check(arguments);
    // 下面正常运行函数的其余部分
    return x + y + z;
}
```

Function 对象的 length 属性由 ECMAScript v1 标准化，在 JavaScript 1.1 和其后的版本中实现（注3）。

注3： Netscape 4.0 中的一个 bug，使得该属性不能被正确使用，除非 <script> 标记的 language 属性被显式地设置为 “JavaScript 1.2”。

7.5.2 属性 prototype

每个函数都有一个 *prototype* 属性，它引用的是预定义的原型对象。原型对象在使用 `new` 运算符把函数作为构造函数时起作用，它在定义新的对象类型时起着非常重要的作用。我们将在第八章中详细的探讨这一属性。

7.5.3 定义你自己的函数属性

当函数需要使用一个在调用过程中都保持不变的值时，使用 `Function` 对象的属性比定义全局变量（这样会使名字空间变得散乱）更加方便。例如，假设我们想编写一个函数，使它在被调用时返回一个唯一的标识符。该函数不能将同一个值返回两次。为了做到这一点，它需要保存已经返回的值，而且这一信息在整个函数调用过程中必须保持不变。虽然我们可以将这一信息存储在一个全局变量中，但是由于这一信息是由函数自己使用的，所以不必使用全局变量。最好的方法莫过于将信息存储在 `Function` 对象的属性中。下面是一个例子，只要这个函数被调用，它都会返回一个唯一的整数：

```
// 创建并初始化静态变量。
// 因为函数声明在执行代码前处理，所以在函数声明前不能真正实现这个赋值运算
uniqueInteger.counter = 0;

// 下面的函数每次被调用时返回值都不同，
// 而且使用它自己的静态属性来跟踪它上次返回的值

function uniqueInteger() {
    // 给静态变量加1，然后返回它的值。
    return uniqueInteger.counter++;
}
```

7.5.4 方法 `apply()` 和 `call()`

ECMAScript v3 给所有函数定义了两个方法 `call()` 和 `apply()`。使用这两个方法可以像调用其他对象的方法一样调用函数（注意，我们还没有讨论过方法，在你读过第八章后，会觉得本节内容更容易理解）。`call()` 和 `apply()` 的第一个参数都是要调用的函数的对象，在函数体内这一参数是关键字 `this` 的值。`call()` 的剩余参数是传递给要调用的函数的值。例如，要把两个数字传递给函数 `f()`，并将它作为对象 `o` 的方法调用，可以使用如下的代码：


```
f.call(o, 1, 2);
```

这与下面的代码相似:

```
o.m = f;  
o.m(1,2);  
delete o.m;
```

`apply()` 方法和 `call()` 方法相似, 只不过要传递给函数的参数是由数组指定的:

```
f.apply(o, [1,2]);
```

例如, 要找到一个数字数组中的最大数字, 可以用 `apply()` 方法把数组元素传递给 `Math.max()` 函数 (注 4):

```
var biggest = Math.max.apply(null, array_of_numbers);
```

JavaScript 1.2 实现了 `apply()` 方法, 但是直到 JavaScript 1.5 才实现了 `call()` 方法。

注 4: 这个例子假定我们使用了 ECMAScript v3 的 `Math.max()` 函数, 该函数可以接受任意多个参数。但 ECMAScript v1 版本定义该函数只能接受两个参数。

第八章

对象

在第三章中解释过，在 JavaScript 中，对象是一种基本数据类型，而且它们也是最重要的一种数据类型。本章将详细介绍 JavaScript 的对象。虽然在接下来的小节中描述的对象的基本用法非常简单，但是后面的小节中我们会发现，对象具有更为复杂的用法和行为。

8.1 对象和属性

对象是一种复合数据类型，它们将多个数据值集中在一个单元中，而且允许使用名字来存取这些值。解释对象的另一种方式是，对象是一个无序的属性集合，每个属性都有自己的名字和值。存储在对象中的已命名的值既可以是数字和字符串这样的原始值，也可以是对象。

8.1.1 对象的创建

对象是由运算符 `new` 创建的。在这个运算符之后必须有用于初始化对象的构造函数名。例如，我们可以使用如下的方式创建一个空对象（即没有属性的对象）：

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

JavaScript还支持内部构造函数,它们以另一种简洁的方式初始化新创建的对象。例如,构造函数 `Date()` 可以初始化一个表示日期和时间的对象:

```
var now = new Date(); // 当前的日期和时间
var new_years_eve = new Date(2000, 11, 31); // 表示2000年12月31日
```

在本章后面的小节中我们将看到,定义自定义构造函数以任何希望方式初始化新创建的对象都是可能的。

对象直接量提供了另一种创建并初始化新对象的方式。我们在第三章中看到过,使用对象直接量,可以把对象的说明直接嵌入JavaScript代码,就像把文本数据作为引用的字符串嵌入JavaScript代码一样。对象直接量由属性说明列表构成,列表包含在大括号中,其中的属性说明由逗号隔开。对象直接量中的每个属性说明都由属性名加上冒号和属性值构成。例如:

```
var circle = { x:0, y:0, radius:2 };
var homer = {
    name: "Homer Simpson",
    age: 34,
    married: true,
    occupation: "plant operator",
    email: "homer@simpsons.com"
};
```

对象直接量的语法由ECMAScript v3标准定义,在JavaScript 1.2和其后的版本中实现。

8.1.2 属性的设置和查询

通常使用“.”运算符来存取对象的属性。位于“.”运算符左边的值应该是对对象的引用(通常是包含了该对象的引用的变量名)。位于“.”运算符右边的值应该是属性名,它必须是一个标识符,而不能是字符串或表达式。例如,如果要引用对象 `o` 的属性 `p`,就要使用 `o.p`,如果要引用对象 `circle` 的属性 `radius`,就要使用 `circle.radius`。对象的属性和变量的工作方式相似,既可以把值储存到其中,也可以从中读取值。例如:

```
// 创建一个对象,并把对它的引用保存在一个变量中。
var book = new Object();

// 设置该对象的属性。
```

```
book.title = "JavaScript: The Definitive Guide"

// 设置更多的属性。注意嵌套的对象。
book.chapter1 = new Object();
book.chapter1.title = "Introduction to JavaScript ;
book.chapter1.pages = 19;
book.chapter2 = { title: "Lexical Structure", pages: 6 };

// 读取该对象的某些属性值。
alert('Outline: ' + book.title + "\n" +
      'Chapter 1 ' + book.chapter1.title + '\n' +
      'Chapter 2 ' + book.chapter2.title);
```

在上面的例子中，需要注意的重要一点是，可以通过把一个值赋给对象的一个新属性来创建它。虽然通常使用关键字 `var` 来声明变量，但是声明对象的属性却不必要（决不能）这么做。而且一旦通过给属性赋值创建了该属性，就可以在任何时刻修改这个属性的值，只需要赋给它新值即可：

```
book.title = 'JavaScript: The Rhino Book'
```

8.1.3 属性的枚举

在第六章中讨论过的 `for/in` 循环提供了一种遍历（或者说枚举）对象属性的方法。在调试一个脚本或者在使用一个对象时非常有用，该对象可以具有任何我们无法事先获知的属性。下面的代码展示了一个函数，可以用它来列出对象的所有属性名：

```
function DisplayPropertyNames(obj) {
    var names = '';
    for(var name in obj) names += name + '\n';
    alert(names);
}
```

注意，`for/in` 循环列出的属性并没有特定顺序，而且虽然它能枚举出所有的用户定义的属性，但是却不能枚举出某些预定义的属性或方法。

8.1.4 未定义的属性

如果要读取一个不存在的属性（即还没有被赋值的属性）的值，那么得到的结果是一个特殊的 JavaScript 值——`undefined`（在第三章中介绍过）。

可以使用运算符 `delete` 来删除一个对象的属性：

```
delete book.chapter2;
```

注意，删除一个属性并不仅仅是把该属性设为undefined，而是真正从对象中移除了该属性。用for/in循环可以证明两者之间的区别，它枚举出的是已经被设为undefined的属性，但它并不列出被删除的属性。

8.2 构造函数

我们从前面知道，在JavaScript中，使用new运算符以及预定义的构造函数（如Object()、Date()、Function()等）可以创建并初始化一个新对象。在许多实例中，这些预定义的构造函数和它们创建的内部对象类型都很有用。但是，在面向对象的程序设计中，使用由程序定义的自定义对象类型也很普遍。例如，如果你想编写一个操作矩形的程序，那么可以用一个特殊类型或对象类来表示矩形。这个矩形类的每个对象都有一个width属性和一个height属性，因为它们定义矩形的本质特征。

要创建已经定义了的带有属性（如width和height）的对象，需要编写一个构造函数在新的对象中创建并初始化这些属性。构造函数是具有两个特性的JavaScript函数：

- 它由new运算符调用。
- 传递给它的是一个对新创建的空对象的引用，将该引用作为关键字this的值，而且它还要对新创建的对象进行适当的初始化。

例8-1说明了如何定义并调用Rectangle对象的构造函数。

例8-1：Rectangle对象的构造函数

```
// 定义构造函数。
// 注意它如何初始化this引用的对象。
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

// 调用构造函数创建两个矩形对象。
// 注意我们把宽度和高度传递给了构造函数，这样它就能正确初始化每个新对象了。
```

```
var rect1 = new Rectangle(2, 4);  
var rect2 = new Rectangle(8.5, 11);
```

注意构造函数如何使用它的参数来初始化 `this` 关键字所引用的对象的属性。记住，构造函数只是初始化了特定的对象，但并不返回这个对象。

我们已经通过定义一个适当的构造函数定义了一个对象类，现在可以确保所有由 `Rectangle()` 构造函数创建的对象都有初始化了的 `width` 和 `height` 属性。这意味着我们可以在此基础上编写程序，统一处理所有的 `Rectangle` 对象。由于每一个构造函数都定义了一个类，所以给每个构造函数一个名字以说明它所创建的对象类就显得比较重要了。例如，用 `new Rectangle(1,2)` 创建一个矩形就比用 `new init_rect(1,2)` 直观得多。

构造函数通常没有返回值。它们只是初始化由 `this` 值传递进来的对象，并且什么也不返回。但是，构造函数可以返回一个对象值，如果这样做，被返回的对象就成了 `new` 表达式的值。在这种情况下，`this` 值所引用的对象就被丢弃了。

8.3 方法

所谓方法（method），其实就是通过对象调用的 JavaScript 函数。回忆一下可以知道，函数就是数值，它们所使用的名字没有任何特殊之处，可以将函数赋给任何变量，甚至赋给一个对象的任何属性。如果有一个函数 `f` 和一个对象 `o`，就可以使用如下的代码定义一个名为 `m` 的方法：

```
o.m = f;
```

定义对象 `o` 的方法 `m()` 之后，就可以采用下面的方式来调用它：

```
o.m();
```

如果 `m()` 需要两个实际参数，可以用下面的方式调用它：

```
o.m(x, x+2);
```

方法有一个非常重要的属性，即在方法主体内部，关键字 `this` 的值就变成了调用该方法的对象。例如，在调用 `o.m()` 时，方法的主体可以使用关键字 `this` 来引用对象 `o`。

有关关键字 `this` 的讨论, 应该从说明我们为什么要使用方法开始。任何一个用作方法的函数都会得到一个额外的实际参数, 即调用该函数的对象。由于方法通常是对那个对象执行某种操作, 所以要表达函数作用于对象这一事实, 最好采用方法的调用语法。比较下面的两行代码:

```
rect.setSize(width, height);  
setRectSize(rect, width, height);
```

虽然这两行代码对对象 `rect` 执行的操作相同, 但是第一行代码采用了方法调用语法, 它在表达对象 `rect` 是操作的焦点 (或者说操作的目标) 这一观点时显然清楚得多。(如果你看来第一行代码不是很自然的话, 那么可能是由于你接触面向对象的程序设计方法的时间不长。有了一点经验后, 你就会逐渐喜欢上这种方法。)

虽然有区别地对待函数和方法比较有用, 但实际上它们之间的差别并没有最初时那么大了。回忆一下, 函数是储存在变量中的值, 而那个变量也不过是全局对象的一个属性。因此, 当你调用一个函数时, 实际上调用的是全局对象的一个方法。在这样的函数中, 关键字 `this` 引用的是全局对象。所以在函数和方法之间并没有什么技术上的差别。真正的差别存在于设计和目的上, 方法是用来对 `this` 对象进行操作的, 而函数通常是独立的, 并不需要使用 `this` 对象。

用一个例子能更清楚地说明方法的典型用法。例 8-2 返回的是例 8-1 的 `Rectangle` 对象, 并且说明了如何定义和调用一个作用于 `Rectangle` 对象的方法。

例 8-2: 方法的定义和调用

```
// 该函数使用了关键字 this, 这样它就不必自己调用自己。  
// 而成为了定义 width 属性和 height 属性的对象的方法。  
function compute_area()  
{  
    return this.width * this.height;  
}  
  
// 使用前面定义的构造函数创建一个新的 Rectangle 对象。  
var page = new Rectangle(8.5, 11);  
  
// 通过把函数赋予对象的属性, 来定义一个方法。  
page.area = compute_area;  
  
// 如下代码调用新方法:  
var a = page.area();    // a = 8.5*11 = 93.5
```

在例8-2中有一个明显的缺点，那就是在调用对象rect的方法area()之前，必须先将该方法赋给rect对象的一个属性。虽然可以调用名为page的特定对象的area()方法，但是在没有将该方法赋给其他的Rectangle对象之前，我们就不能调用任何一个Rectangle对象的area()方法。这很快就会使你感到繁琐。例8-3给Rectangle定义了另外一些方法，并且说明了如何使用一个构造函数把这些方法自动地赋予所有Rectangle对象。

例8-3：用构造函数定义方法

```
// 首先定义一些函数，它们将被用作方法。
function Rectangle_area() { return this.width * this.height; }
function Rectangle_perimeter() { return 2*this.width + 2*this.height; }
function Rectangle_set_size(w,h) { this.width = w; this.height = h; }
function Rectangle_enlarge() { this.width *= 2; this.height *= 2; }
function Rectangle_shrink() { this.width /= 2; this.height /= 2; }

// 为Rectangle对象定义一个构造函数方法。
// 构造函数不仅要初始化属性，还要给方法赋值。
function Rectangle(w, h)
{
    // 初始化对象的属性。
    this.width = w;
    this.height = h;

    // 定义对象的方法。
    this.area = Rectangle_area;
    this.perimeter = Rectangle_perimeter;
    this.set_size = Rectangle_set_size;
    this.enlarge = Rectangle_enlarge;
    this.shrink = Rectangle_shrink;
}

// 现在，一旦创建了一个Rectangle对象，就可以直接调用它的方法。
var r = new Rectangle(2,2);
var a = r.area();
r.enlarge();
var p = r.perimeter();
```

例8-3说明的方法还是有缺点。在这个例子中，构造函数Rectangle()对它所要初始化的Rectangle对象的7项属性都进行了设置，即使其中5项属性的值都是常量。每个矩形对象的这5项属性值都是相同的。由于每个属性都占用一定内存空间，所以给Rectangle类添加方法，就会使每个Rectangle对象占用的内存增加为原来的三倍。幸运的是，JavaScript引入了一种解决该问题的方法，即它允许一个对象继承原型对象的属性。下一节将详细讨论这一方法。

8.4 原型对象和继承

我们已经知道，用构造函数把方法赋予它要初始化的对象，效率非常低。如果我们这样做，那么构造函数创建的每一个对象都会有相同的方法属性的副本。有一种更为有效的方式可以用来声明方法、常量以及其他能被类的所有对象共享的属性。

JavaScript 对象都“继承”原型对象的属性（注1）。每个对象都有原型对象，原型对象的所有属性是以它为原型的对象的属性。也就是说，每个对象都继承原型对象的所有属性。

一个对象的原型是由创建并初始化该对象的构造函数定义的。JavaScript 中的所有函数都有 `prototype` 属性，它引用了一个对象。虽然原型对象初始化时是空的，但是你在其中定义的任何属性都会被该构造函数创建的所有对象继承。

构造函数定义了对象的类，并初始化了类中状态变量的属性，如 `width` 和 `height`。因为原型对象和构造函数关联在一起，所以类的每个成员都从原型对象继承了相同的属性。这说明原型对象是存放方法和其他常量属性的理想场所。

注意，继承是在查询一个属性值时自动发生的。属性并非从原型对象复制到新的对象的，它们只不过看起来像是那些对象的属性。这其中有两点重要的含义。一是使用原型对象可以大量减少每个对象对内存的需求量，因为对象可以继承许多属性。二是即使属性在对象被创建之后才添加到它的原型对象中，对象也能够继承这些属性。

每个类都有一个原型对象，这个原型对象都具有一套属性。但是实际上却有大量的类实例，每个实例都能继承原型对象的属性。由于一个原型对象能够被多个对象继承，所以 JavaScript 必须强化读写属性值之间的不对称性。在读对象 `o` 的属性 `p` 时，JavaScript 首先检查 `o` 是否具有一个名为 `p` 的属性。如果 `o` 没有这个属性，JavaScript 就会再检查 `o` 的原型对象是否具有这个属性。这样才使得以原型为基础的继承机制起作用。

而当你写一个属性的值时，JavaScript 并不使用原型对象。要明白为什么，就考虑

注 1：原型是在 JavaScript 1.1 中引入的，已经被废弃的 JavaScript 1.0 不支持它。

一下如果这样做会出现什么样的后果。假设你在对象 *o* 还没有一个名为 *p* 的属性时要设置属性 *o.p* 的值。进一步假设 JavaScript 继续在 *o* 的原型对象中查询属性 *p*，并且允许你设置原型对象的这一属性，那就改变了整个对象类的 *p* 值，而不是仅仅改变了自己想要改变的属性值。

因此，属性的继承只发生在读属性值时，而在写属性值时不会发生。如果你设置的对象 *o* 的属性 *p* 是对象 *o* 从它的原型对象继承而来的，那么结果是你直接在对象 *o* 中创建了一个新属性 *p*。现在 *o* 已经有了自己的名为 *p* 的属性，它就不会再从它的原型对象继承 *p* 的值了。当你读 *p* 的值时，JavaScript 首先查询 *o* 的属性。由于它发现了 *o* 中定义的 *p*，它就不必再查询原型对象，也就不会再发现原型对象中定义的 *p* 值了。我们有时说 *o* 中的 *p* “遮蔽”了或者说“隐藏”了原型对象中的属性 *p*。原型对象的继承是一个容易让人混淆的主题。图 8-1 说明了我们在这里讨论的概念。

因为原型对象的属性被一个类的所有对象共享，所以通常只用它们来定义类中所有对象的相同的属性。这使得原型对象适用于方法定义。另外原型属性还适合于那些具有常量值（如数学常量）的属性的定义。如果你在类中定义了一个属性，它有一个非常常用的默认值，那么你可以在原型对象中定义这个属性和它的默认值。然后，那些不想使用默认值的少数对象可以创建自己专有的、非共享的属性副本，并且将它定义为自己的、非默认的值。

让我们从对原型对象继承机制的抽象讨论转移到一个具体的例子上来。假设我们定义了一个构造函数 `Circle()`，用来创建表示圆形的对象。因为这个类的原型对象是 `Circle.prototype`（注 2），所以我们可以定义一个对所有 `Circle` 对象都有效的常量。

注 2： 构造函数的原型对象是由 JavaScript 自动创建的。在大多数 JavaScript 版本中，它都会自动给予每个函数一个空的原型对象，以防它被用作一个构造函数。但是在 JavaScript 1.1 中，在函数首次被用作构造函数之前，不会为它创建原型对象。这就是说，为了能和 JavaScript 1.1 兼容，在使用原型对象给那个类的对象赋予方法和常量之前，至少应该创建该类的一个对象。因此，如果我们已经定义了构造函数 `Circle()`，但是还没有用它来创建过任何 `Circle` 对象，那么我们应该定义如下常量属性 *pi*：

```
// First create and discard a dummy object; forces prototype object creation.
new Circle();
// Now we can set properties in the prototype.
Circle.prototype.pi = 3.14159;
```

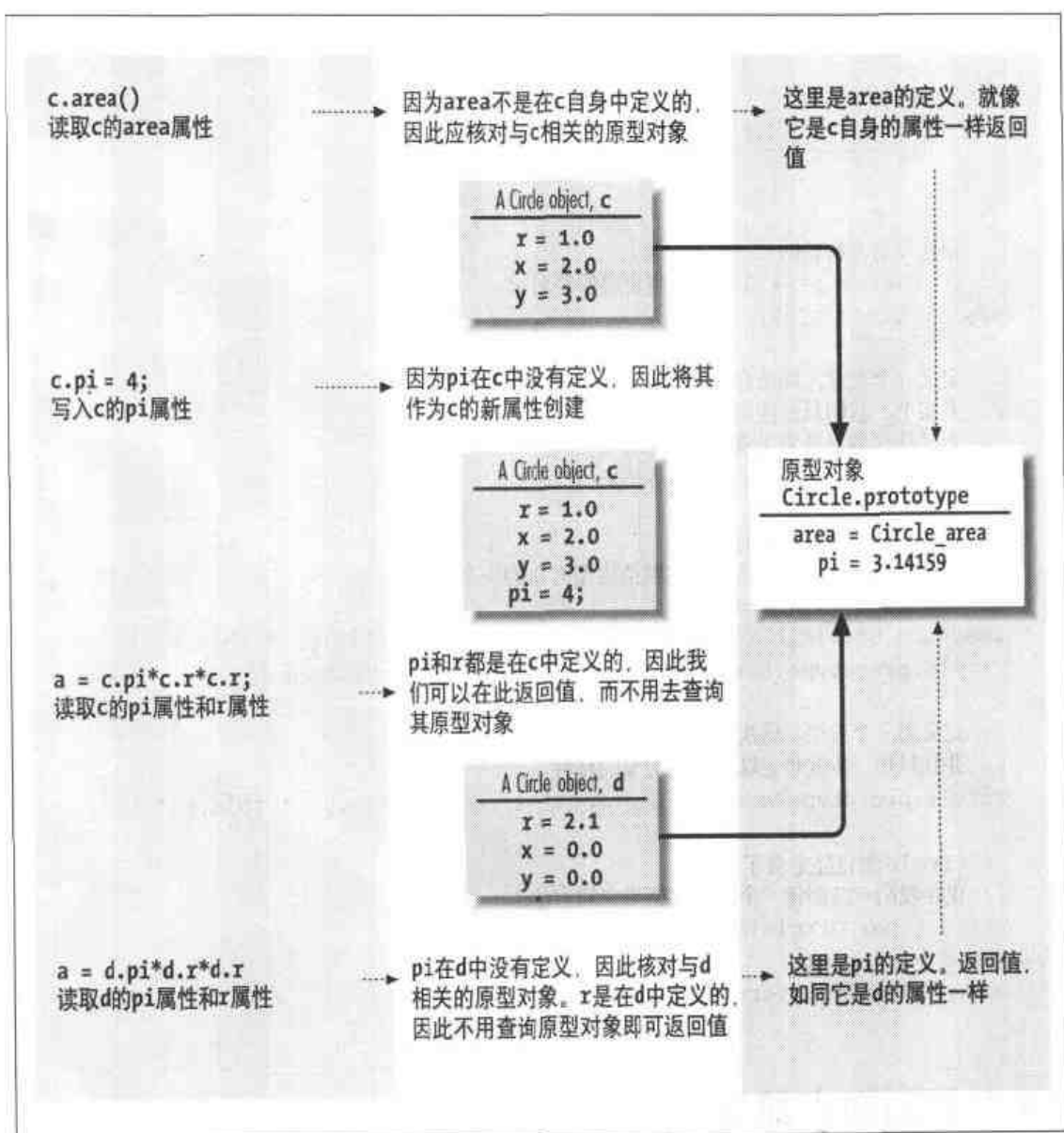


图 8-1: 对象和其原型对象

```
Circle.prototype.pi = 3.14159;
```

例 8-4 展示的 Circle 没有任何问题。这段代码定义 Circle 类时，首先定义了一个构造函数 Circle() 来初始化每个对象，然后通过设置 Circle.prototype 属性来定义所有的类实例共享的方法和常量。

例 8-4: 用原型对象定义一个 Circle 类

```
// 为我们的类定义一个构造函数方法。
// 用它初始化 Circle 对象的属性。
// 每个 Circle 对象的属性值都不同。
function Circle(x, y, r)
```

```

{
    this.x = x; // 圆心的X坐标。
    this.y = y; // 圆心的Y坐标。
    this.r = r; // 圆的半径。
}

// 创建并舍弃初始的Circle对象。
// 在JavaScript 1.1中，这将创建原型对象。
new Circle(0,0,0);

// 定义一个常量，即所有Circle对象共享的属性。
// 事实上，我们只是使用了Math.PI，
// 不过这样做只是为了满足本例的需要。
Circle.prototype.pi = 3.14159;

// 定义一个计算圆周长的方法。
// 首先声明一个函数，然后把它赋给原型对象的一个属性。
// 注意上面定义的常量的用法。
function Circle_circumference() { return 2 * this.pi * this.r; }
Circle.prototype.circumference = Circle_circumference;

// 定义另一个方法。这次使用函数直接量来定义函数。
// 并且只用一步就将其赋给原型对象的属性。
Circle.prototype.area = function() { return this.pi * this.r * this.r; }

// Circle类已经定义了。
// 现在我们可以创建一个实例并调用它的方法。
var c = new Circle(0.0, 0.0, 1.0);
var a = c.area();
var p = c.circumference();

```

8.4.1 原型和内部类

不只是用户定义的类具有原型对象，像String和Date这样的内部类同样具有原型对象，你也可以给它们赋值（注3）。例如，下面的代码定义了一个新方法，它适用于所有String对象：

```

// 如果最后一个字符是c，返回true。
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}

```

在String的原型对象中定义了新方法endsWith()后，可以用如下的方式来使用它：

注3：只适用于JavaScript 1.1及其后的版本。

```
var message = "hello world";  
message.endsWith('h') // 返回 false  
message.endsWith('d') // 返回 true
```

8.5 面向对象的 JavaScript

虽然 JavaScript 支持我们称为对象的数据类型，但是它并没有正式的类的概念。这使它有别于那些经典的面向对象程序设计语言，如 C++ 和 Java 等。在面向对象的程序设计语言中，共有的概念是强类型和支持以类为基础的继承机制。根据这个评判标准，很容易就能将 JavaScript 从真正的面向对象语言中区分出来。但是，我们又会发现 JavaScript 大量地使用了对象，而且它还有自己的以原型对象为基础的继承机制，这些又说明 JavaScript 是一种真正的面向对象语言。它从大量其他的（相对比较模糊）面向对象语言中获取了灵感，这些语言采取以原型对象为基础的继承机制，而不是采用以类为基础的继承机制。

虽然 JavaScript 不是一种以类为基础的面向对象程序设计语言，但是它在模拟像 Java 和 C 这样的以类为基础的语言时是相当出色的。在本章中，我们已经使用了类这个术语。本节将正式地探讨 JavaScript 和 Java 及 C++ 这样采用真正的以类为基础的继承机制的语言之间的共通之处（注 4）。

首先，我们来定义几个术语。我们已经知道，对象是一个数据结构，包含各种命名了的数据段，而且还能包含对这些数据段进行操作的各种方法。一个对象组将数值和方法组织到一个方便的包中，这样就增强了代码的模块性和可再用性，从而使程序设计更加容易。JavaScript 中的对象可以具有大量的属性，而且还可以动态地将这些属性添加到对象中。这在像 Java 和 C++ 这样的强类型语言中是做不到的。在那些语言中，每个对象都有一套预定义的属性（注 5），而且每个属性都有预定义的类型。在使用 JavaScript 的对象来模拟面向对象的程序设计方法时，通常是给每个对象预先定义一套属性以及这些属性的数据类型。

在 Java 和 C++ 中，类定义了对象的结构。类确定了一个对象包含有哪些域以及每个

注 4：即使你不熟悉这些语言和面向对象的程序设计方法，也应该阅读本节的内容。

注 5：在 Java 和 C++ 中，它们被称为字段（field），但是在这里我们称它为属性，因为这是 JavaScript 的术语。

域包含什么样的数据类型。类也定义了对对象进行操作的方法。虽然 JavaScript 并没有正式的类的概念，但是我们已经见到过，它用构造函数和原型对象模拟了类。

在 JavaScript 和以类为基础的面向对象语言中，同一个类可以具有多个对象。我们常常称对象是它所属的那个类的实例，所以任何类都可以有多个实例。有时我们使用实例化（*instantiate*）这个术语来描述创建一个对象（类的一个实例）的过程。

在 Java 中有一条常用的约定，即命名类时以大写字母开头，命名对象时以小写字母开头。这一约定可以帮助我们区分代码中的类和对象，JavaScript 沿用了这一有用的约定。例如，在前面几节中，我们定义了类 *Circle* 和类 *Rectangle*，而且分别创建了这两个类的实例，名为 *c* 和 *rect*。

一个 Java 类的成员有四种可能的基本类型：实例属性、实例方法、类属性以及类方法。在接下来的几节中，我们将研究这些成员类型之间的区别，并且说明在 JavaScript 中如何模拟它们。

8.5.1 实例属性

每个对象都有它自己单独的实例属性的副本。简而言之，如果有 10 个给定的类的对象，那么每个实例属性就有 10 个副本。例如，在 *Circle* 类中，每个 *Circle* 对象都有一个属性 *r*，它指定了这个圆的半径，这里 *r* 就是一个实例属性。由于每个对象都有它自己的实例属性副本，所以通过单独的对象可以存取这些变量。例如，如果对象 *c* 是 *Circle* 类的一个实例，可以这样来引用半径：

```
c.r
```

在默认情况下，JavaScript 中的任何对象属性都是实例属性。但是为了真实地模拟面向对象的程序设计语言，我们说 JavaScript 中的实例属性是那些在对象中用构造函数创建的或初始化的属性。

8.5.2 实例方法

实例方法和实例属性非常相似，只不过它是方法而不是数值（在 Java 中，函数和方法并不像它们在 JavaScript 中那样是数据，所以这一区别就更加明显了）。实例方法

是由特定对象或实例调用的。Circle类中的方法area()就是一个实例方法。它是由Circle的对象c调用的:

```
a = c.area();
```

实例方法使用了关键字this来引用它们要操作的对象或实例。虽然一个类的任何实例都可以调用实例方法,但是这并不意味着每个对象都像实例属性那样含有自己独有的方法副本。相反,每个实例方法都是由类的所有实例共享的。在JavaScript中,给类定义一个实例方法,是通过把构造函数的原型对象中的一个属性设置为函数值来实现的。这样,由那个构造函数创建的所有对象都会共享一个已继承的对函数的引用,而且使用上面所述的方法调用语法就能够调用这个函数。

8.5.3 类属性

在Java中,类属性是一个与类相关联的变量,而不是和类的每个实例相关联的变量。无论一个类创建多少个实例,每个类属性都只有一个副本。就像实例属性是通过类的实例存取的一样,类属性是通过类存取的。在JavaScript中,Number.MAX_VALUE就是类属性的一个例子,因为属性MAX_VALUE就是通过类Number来存取的。由于每个类属性都只有一个副本,所以本质上说来类属性是全局变量。但是它们与一个类关联在一起,在JavaScript的名字空间中拥有一个逻辑位置,这样它们就不会被其他的同名属性覆盖。为了尽可能说得清楚,我们模拟了一个JavaScript的类属性,简单地定义了构造函数自身的一个属性。例如,要创建一个类属性Circle.PI来存储数学常量pi,我们可以采取如下的方法:

```
Circle.PI = 3.14;
```

虽然Circle是一个构造函数,但是由于JavaScript函数是对象,所以我们可以创建函数的属性,就像创建对象的属性一样。

8.5.4 类方法

最后,我们介绍一下类方法。类方法是一个与类关联在一起的方法,而不是和类的实例关联在一起的方法。要调用类方法,就必须使用类本身,而不能使用该类的特定实例。方法Date.parse()(在本书的核心参考手册部分可以查到这一方法)就

是一个类方法。通过构造函数对象 `Date` 才能调用这个方法，而不能通过 `Date` 类的一个特殊实例来调用它。

由于类方法不能通过一个特定对象调用，所以使用关键字 `this` 对它来说没有意义（`this` 引用的是调用实例方法的对象）。和类属性一样，类方法是全局性的。因为类方法不是对特定对象进行操作的，所以类方法更容易被认为是由类调用的函数。同样的，将这些函数关联到一个类可以使它们在 JavaScript 的名字空间中有一个便利的位置，并且防止出现命名冲突。在 JavaScript 中，要定义一个类方法，只需要用合适的函数作为构造函数的属性即可。

8.5.5 例子：类 Circle

例 8-5 是我们的 `Circle` 类的再实现，它包括了前面四个例子中所有基本的成员类型。

例 8-5: `Circle` 的实例属性和类属性及实例方法的定义

```
function Circle(radius) { // 这个构造函数定义了类自身。
    // r 是构造函数定义并初始化的一个实例属性。
    this.r = radius;
}

// Circle.PI 是一个类属性，它是构造函数的一个属性
Circle.PI = 3.14159;

// 以下的函数将计算圆的面积。
function Circle_area() { return Circle.PI * this.r * this.r; }

// 下面我们通过把函数赋给构造函数的原型对象使它成为一个实例方法。
// 记作，在 JavaScript 1.2 中，我们可以用函数直接量定义一个无需命名为 Circle_area 的函数。
Circle.prototype.area = Circle_area;

// 以下是另一个函数，它以两个 Circle 对象作为实际参数，并
// 返回其中较大的一个（取半径较大的实际参数）
function Circle_max(a,b) {
    if (a.r > b.r) return a;
    else return b;
}

// 由于该函数比较的是两个 Circle 对象，所以将它看作对个别 Circle 对象进行操作的实例方法，
// 是没有意义的。但是我们不希望它成为一个独立的函数，所以我们把它赋予一个构造函数，
// 使它成为类方法。
Circle.max = Circle_max;

// 下面的代码使用了 Circle 对象的各个域：
```



```

var c = new Circle(1.0); // 创建Circle类的一个实例。
c.r = 2.2; // 设置实例属性r。
var a = c.area(); // 调用实例方法area()。
var x = Math.exp(Circle.PI); // 在我们自己的计算中使用属性PI。
var d = new Circle(1.2); // 创建另一个Circle实例。
var bigger = Circle.max(c,d); // 使用类方法max()。

```

8.5.6 例子：复数

例8-6是另一个在JavaScript中定义对象的类的例子，它比上一个例子更正规一点，其中的代码和注释值得仔细研究。注意，这个例子使用的是JavaScript 1.2的函数直接量的语法。因为它要求使用这一版本的JavaScript（或以后版本），所以不必在给原型对象赋值之前调用一次构造函数，也就是说不用担心和JavaScript 1.1的兼容。

例 8-6：一个复数类

```

/*
 * Complex.js:
 * 该文件定义了一个Complex类来表示复数。
 * 复数就是一个实数和一个虚数的和，虚数i是-1的平方根。
 */

/*
 * 定义类的第二步是定义该类的构造函数。
 * 这个构造函数要初始化对象的所有实例属性。
 * 这些属性是核心的“状态变量”，是它们使类的每个实例互不相同。
 */
function Complex(real, imaginary) {
    this.x = real; // 复数的实数部分。
    this.y = imaginary; // 复数的虚数部分。
}

/*
 * 定义类的第二步是在构造函数的原型对象中定义它的实例方法（或其他属性）。
 * 该对象定义的任何属性都将被这个类的所有实例继承。
 * 注意实例方法对this关键字的隐式操作。
 * 许多方法都不需要其他实际参数。
 */

// 返回复数的大小。
// 它定义的是从原点(0, 0)到复平面的距离。
Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// 返回复数的相反数。

```

```

Complex.prototype.negative = function() {
    return new Complex( this.x, -this.y );
};

// 以一种有效的方式将一个 Complex 对象转换成一个字符串。
// 这是在把 Complex 对象用作字符串时调用的函数。
Complex.prototype.toString = function() {
    return "(" + this.x + " + " + this.y + "i";
};

// 返回一个复数的实数部分。
// 该函数是在将 Complex 对象作为原始值处理时调用的。
Complex.prototype.valueOf = function() { return this.x; }

/*
 * 定义类的第三步是定义类方法。
 * 常量和和其他必需的类属性，作为构造函数自身的属性（而不是构造函数的原型对象的属性）。
 * 注意，类方法没有使用关键字 this，因为它们只对它们的实际参数进行操作。
 */

// 计算两个复数的和并返回结果。
Complex.add = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};

// 用一个复数减另一个复数。
Complex.subtract = function (a, b) {
    return new Complex(a.x - b.x, a.y - b.y);
};

// 两个复数相乘，返回乘积。
Complex.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
};

// 下面是一些有用的预定义复数。
// 它们被定义为类属性，这样它们就可以被用作常量。
// （注意，实际上它们并不是只读的。）
Complex.zero = new Complex(0,0);
Complex.one = new Complex(1,0);
Complex.i = new Complex(0,1);

```

8.5.7 超类和子类

在Java、C++和其他以类为基础的面向对象语言中，都有一个明确的类层次的概念。每个类都有一个超类，它们从超类中继承属性和方法。类还可以被扩展，或者说子类化，这样其他子类就能够继承它的行为。我们已经知道，JavaScript支持的是以

原型为基础的继承机制，而不是以类为基础的继承机制，但是我们仍旧能够总结出类似的类层次图。在JavaScript中，类Object是最通用的类，其他所有类都是专用化了的版本，或者说是Object的子类。另一种解释方法是Object是所有内部类的超类。所有类都继承了Object的基本方法（我们将在本章后面的小节中介绍）。

我们已经学会了对象如何从它们构造函数的原型对象中继承属性。那么它们又是如何继承类Object的属性的呢？我们知道，原型对象本身就是一个对象，它是由构造函数Object()创建的。这就意味着原型对象继承了Object.prototype属性。因此类Complex的对象就继承了Complex.prototype对象的属性，而后者又继承了Object.prototype的属性。由此可以推出，对象Complex继承了两个对象的属性。在Complex对象中查询某个属性时，首先查询的是这个对象本身。如果在这个对象中没有发现要查询的属性，就查询Complex.prototype对象。最后，如果在那个对象中还没有找到要查询的属性，就查询Object.prototype对象。

注意，由于Complex原型对象是在查询Object的原型对象之前被查询的，所以Complex.prototype的属性就隐藏了Object.prototype的同名属性。例如，在例8-6的类定义中，我们在Complex.prototype对象中定义了toString()方法，虽然Object.prototype中也定义了同名的方法，但是对象Complex永远也不会看到它，因为首先发现的是Complex.prototype中定义的toString()。

我们在本章中使用的类都是Object的直接子类。这是JavaScript的程序设计所特有的，通常它不需要生成更复杂的层次。但在必要的情况下，有可能对其他类进行子类化。例如，假定我们要生成类Complex的一个子类，以便能添加一些新方法。要做到这一点，只需要确保新类的原型对象是Complex的一个实例，这样它就能继承Complex.prototype的所有属性：

```
// 下面是子类的构造函数。
function MoreComplex(real, imaginary) {
    this.x = real;
    this.y = imaginary;
}

// 我们将它的原型对象作为Complex对象。
// 这意味着新类的实例将继承MoreComplex.prototype,
// 后者由Complex.prototype继承而来,
// Complex.prototype又由Object.prototype继承而来。
MoreComplex.prototype = new Complex(0,0);
```

```
// 下面给子类添加一个新方法或新特性。  
MoreComplex.prototype.swap = function() {  
    var tmp = this.x;  
    this.x = this.y;  
    this.y = tmp;  
}
```

上面展示的子类化方法有一点缺陷。由于我们明确地把 `MoreComplex.prototype` 设成了我们所创建的一个对象，所以就覆盖了 JavaScript 提供的原型对象，而且抛弃了给定的 `constructor` 属性。该属性（将在本章后面的小节中介绍）引用的是创建这个对象的构造函数。但是 `MoreComplex` 对象继承了它的超类的 `constructor` 属性，它自己并没有这个属性。明确地设置这一属性可以解决这个问题：

```
MoreComplex.prototype.constructor = MoreComplex;
```

但是要注意，在 JavaScript 1.1 中，`constructor` 属性是只读的，不能用这种方式设置它。

8.6 作为关联数组的对象

我们已经见到过用运算符“.”来存取一个对象属性，而数组更常用的存取属性运算符是 `[]`。这样，下面的两行代码是等价的：

```
object.property  
object['property']
```

这两条语法之间最重要的区别是，前者的属性名是标识符，后者的属性名却是一个字符串。很快我们会明白为什么这一点如此重要。

在 C、C++、Java 和其他类似的强类型语言中，一个对象的属性数是固定的，而且必须预定义这些属性的名字。由于 JavaScript 是一种松类型的语言，它并不采用这一规则，所以在用 JavaScript 编写的程序中，可以为对象创建任意数目的属性。但是当你采用“.”运算符来存取一个对象的属性时，属性名是用标识符表示的。而在 JavaScript 程序中，标识符必须被逐字地输入，它们不是一种数据类型，因此程序不能对它们进行操作。

另一方面，当用数组的 `[]` 表示法来存取一个对象的属性时，属性名是用字符串表示

的。字符串是JavaScript的一种数据类型，因此可以在程序运行的过程中操作并创建它们。例如，可以用JavaScript编写如下的代码：

```
var addr = '' ;
for(i = 0; i < 4; i++) {
    addr += customer["address" + i] + '\n' ;
}
```

这一代码读取了customer对象的属性address0、address1、address2和address3，并且将它们连接起来。

上面的代码段说明了采用数组表示法访问带有字符串表达式的对象的属性是非常灵活的。虽然我们也可以用“.”的表示法来编写那个例子，但是也有一些情况是只能用数组来解决的。例如，假定要编写一个程序，用网络资源来计算用户在股票市场上投资的当前值。这个程序要允许用户输入他所拥有的每支股票的名字以及每支股票的数量。可以使用一个名为portfolio的对象来保存这些信息。该对象为每支股票设置一个属性，其属性名就是这支股票的名字，属性值就是该支股票的数量。例如，如果一个用户具有50股IBM公司的股票，那么属性portfolio.ibm的值就是50。

这个程序还需要一个循环，它首先要提示用户输入他所拥有的股票名，然后请他输入拥有的这支股票的数量。循环内部的代码如下：

```
var stock_name = get_stock_name_from_user();
var shares = get_number_of_shares();
portfolio[stock_name] = shares;
```

由于用户是在运行过程中输入股票名，所以在此之前你无法知道这个属性名。因为你不知道这个属性名，所以在编写程序时就不能用“.”运算符来存取对象portfolio的属性。但是可以使用运算符[]来命名属性，因为它的属性名是一个字符串值（该值是动态的，可以在运行时改变），而不是一个标识符（它是静态的，在程序中必须对其进行硬编码）。

如果使用一个对象时采取的是这种形式，我们常常称它为关联数组（associative array），它是一个数据结构，允许你动态地将任意数值和任意字符串关联在一起。实际上JavaScript对象在内部是用关联数组实现的。存取属性时使用“.”的表示法使它们看来像C++和Java的静态对象，而且作为静态对象使用时，它们同样很出色。

除此之外，它们还有更强大的能力，可以将数值和任意字符串关联起来。从这个角度来看，JavaScript 比 C++ 和 Java 的对象更像 Perl 数组。

我们在第六章中介绍了 `for/in` 循环。当这个 JavaScript 语句和关联数组一起使用时，它的强大之处才真正显示了出来。现在返回到股票业务量的例子中，在用户输入了自己的业务量后，我们需要使用如下代码来计算它的当前总值：

```
var value = 0;
for (stock in portfolio) {
    // 获取有价证券一览表中每支股票的单价。
    // 然后用股份额乘以它。
    value += get_share_value(stock) * portfolio[stock];
}
```

如果没有 `for/in` 循环我们就编不出这样的代码，因为股票的名字是不可预知的。这是把属性名从名为 `portfolio` 的关联数组（或者说 JavaScript 的对象）中抽取出来的惟一方法。

8.7 对象的属性和方法

在 JavaScript 中所有的对象都由类 `Object` 继承而来。虽然一些专用的类，如内部的 `String` 类或用户定义的 `Complex` 类都定义了自己的属性和方法，但是所有对象，无论它的类是什么，都支持类 `Object` 类定义的属性和方法。由于这些属性和方法的一般性，使得它们具有特殊的重要性。

8.7.1 constructor 属性

从 JavaScript 1.1 开始，每个对象都具有 `constructor` 属性，它引用的是用来初始化该对象的构造函数。例如，如果用构造函数 `Complex()` 创建了一个对象 `o`，那么属性 `o.constructor` 引用的就是 `Complex`：

```
var o = new Complex(1,2);
o.constructor == Complex; // 值为 true
```

当然，并不是每一个 `Complex` 对象（或其他任何类型的对象）都具有自己惟一的 `constructor` 属性。相反，这个属性是从原型对象继承来的。我们在本章前面的小

节中讨论过, JavaScript 会为我们定义的每一个构造函数都创建一个原型对象, 并且将那个对象赋给构造函数的 prototype 属性。但是在前面的小节中没有说明原型对象初始时是非空的。在原型对象创建之初, 它包括一个 constructor 属性, 用来引用构造函数。也就是说, 如果有一个函数 *f*, 那么属性 *f*.prototype.constructor 就总等于 *f* (除非将它设为别的值)。

由于构造函数定义了一个对象的类, 所以属性 constructor 在确定给定对象的类型时是一个功能强大的工具。例如, 可以使用如下代码来确定一个未知的对象的类型:

```
if (typeof o == "object" && (o.constructor == Date))  
    // 然后用 Date 对象做一些操作。
```

但是, 并不能保证 constructor 属性总是存在的。例如, 一个类的创建者可以用一个全新的对象来替换构造函数的原型对象, 而新对象可能不具有有效的 constructor 属性。

8.7.2 toString()方法

方法 toString() 没有任何实际参数, 它返回的是一个字符串, 该字符串代表了调用它的对象的类型或值。当 JavaScript 需要将一个对象转换成字符串时就调用这个对象的 toString() 方法。例如, 当用运算符 “+” 把一个字符串和一个对象连接在一起时, 或者要把一个对象传递给 alert() 或 document.write() 方法时, 就会调用 toString()。

默认的 toString() 方法提供的信息并不多。例如, 下面的代码只能使浏览器显示出字符串 “[object Object]” (注 6):

```
c = new Circle(1, 0, 0);  
document.write(c);
```

由于默认的方法并不能显示出太多有用信息, 所以许多类都定义了自己的 toString() 方法。例如, 当一个数组被转换成一个字符串时, 就得到一个数组元素

注 6: 在 Netscape 公司的客户端 JavaScript 中, 如果显式地将 <script> 标记的 language 特性设置为 “JavaScript 1.2”, 那么 toString() 方法的行为会有所不同。它将对对象直接量表示法来显示对象的所有字段的名称和值。这违反了 ECMAScript 规范。

的列表，其中每个元素都被转换成了字符串。当一个函数被转换成字符串时，就会获得该函数的源代码。

`toString()`的目的是每个对象类都有自己特定的字符串表示，所以应该定义一个合适的`toString()`方法将对象转换成相应的字符串形式。当你定义一个类时，就应该为它定义一个`toString()`方法，以便能将这个类的实例转换成有意义的字符串。这个字符串应该包含被转换的对象的有用信息，因为这对调试来说非常有用。如果仔细选择字符串转换，它在程序中也会非常有用。

下面的代码展示了我们为例8-5中的`Circle`类定义的`toString()`方法：

```
Circle.prototype.toString = function () {  
    return "[Circle of radius " + this.r + ", centered at (" +  
        + this.x + ", " + this.y + ").]";  
}
```

用上面定义的`toString()`方法可以将一个典型的`Circle`对象转换成字符串“[Circle of radius 1, centered at (0, 0).]”。

如果回顾一下例8-6，就会发现它为复数类`Complex`定义了`toString()`方法。

由类`Object`定义的默认`toString()`方法有一个有趣的特性，那就是它揭示了一些有关内置对象的内部类型信息。这个默认的`toString()`方法返回的字符串形式总是：

```
[object class]
```

`class`是对象的内部类型，通常对应于该对象的构造函数名。例如，`Array`对象的`class`为“`Array`”，`Function`对象的`class`为“`Function`”，`Date`对象的`class`为“`Date`”，内部`Math`对象的`class`为“`Math`”，所有`Error`对象（包括各种`Error`子类的实例）的`class`为“`Error`”。客户端JavaScript对象和由JavaScript实现定义的其他所有对象都具有已定义实现的`class`（如“`Window`”、“`Document`”和“`Form`”）。用户定义的对象（如本章前面定义的`Circle`和`Complex`）的`class`为“`Object`”。

注意，`class`值提供了`typeof`运算符（对所有对象都返回“`Object`”或“`Function`”）不能提供的有用信息。`class`值提供的信息与前面介绍的`constructor`属性提供的值相似，但是它以字符串的形式提供这些信息，而不是以构造函数的形式提供这些信

息。但是，获取这个 `class` 值的惟一方法是调用 `Object` 定义的默认 `toString()` 方法。因为类常常会定义自己的 `toString()` 方法，所以不能只是调用某个对象的 `toString()` 方法：

```
o.toString() // 调用该对象中定制的 toString() 方法
```

但是，必须明确地引用 `Object.prototype.toString` 对象的默认 `toString()` 函数，用函数的 `apply()` 方法在想要使用的对象上调用它：

```
Object.prototype.toString.apply(c); // 总是调用默认的 toString() 方法
```

我们可以使用这种方法定义一个函数，以提供强化了了的“`type of`”功能：

```
// 增强的“type of”函数，返回描述x类型的字符串。  
// 注意，对于用户定义的对象类型，它都返回“Object”。  
function Typeof(x) {  
    // 用typeof运算符开始。  
    var t = typeof x;  
    // 如果结果清楚，则返回它。  
    if (t != "object") return t;  
    // 否则，x是对象，获取它的类值，以便弄清楚该对象的类型。  
    var c = Object.prototype.toString.apply(x); // 返回“[object class]”。  
    c = c.substring(8, c.length-1);           // 去掉“[object”和“]”。  
    return c;  
}
```

8.7.3 toLocaleString()方法

在 ECMAScript v3 和 JavaScript 1.5 中，除了 `toString()` 方法外，`Object` 类还定义了 `toLocaleString()` 方法。该方法的作用是返回该对象局部化的字符串表示。`Object` 类定义的默认 `toLocaleString()` 方法自身不做任何局部化，返回的结果与 `toString()` 方法返回的完全相同。但是 `Object` 类的子类则可能定义自己的 `toLocaleString()` 方法。在 ECMAScript v3 中，`Array`、`Date` 和 `Number` 类都定义了自己的 `toLocaleString()` 方法，返回的是局部化的值。

8.7.4 valueof()方法

方法 `valueOf()` 和方法 `toString()` 非常相似，当 JavaScript 需要将一个对象转换成字符串之外的原始类型（通常是数字）时，就需要调用它。这个函数返回的是能代表关键字 `this` 所引用的对象的值的数据。

由于对象没有定义为原始类型的值，所以大多数对象都没有等价的原始值。因此由 Object 类定义的 `valueOf()` 方法不执行任何转换，只是返回调用它的对象。像 Number 和 Boolean 这样的类具有明显的原始等价值，所以它们就覆盖了方法 `valueOf()`，让它返回合适的原始值。这就是 Number 和 Boolean 对象的行为和它们的等价原始值如此相象的原因。

有时，你还可以定义一个合理的原始等价类型的类。在这种情况下，需要为这个类定制一个 `valueOf()` 方法。回顾一下例 8-6，就会发现我们为 Complex 类定义了一个 `valueOf()` 方法。这个方法返回的是复数的实数部分，所以当在一个数字环境中使用 Complex 对象时，它的行为看起来就像是一个根本就没有虚数部分的实数一样。考虑下面的代码：

```
var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.add(a,b); // c 是复数(3,3)
var d = a - b;           // d 是数字3
```

定义方法 `valueOf()` 时需要注意的一点是，在某些环境中，当进行对象到字符串的转换时，方法 `valueOf()` 的优先级比方法 `toString()` 的优先级高。这样，当你为一个类定义了 `valueOf()` 方法时，如果你想强制性的将那个类的对象转换成字符串，就必须明确地调用方法 `toString()`。继续以 Complex 类为例：

```
alert('c = ' + c); // 用 valueOf() 方法: 显示 "c = 3"
alert('c = ' + c.toString()); // 显示 "c = (3,3)"
```

8.7.5 hasOwnProperty()方法

如果对象局部定义了一个非继承的属性，属性名是由一个字符串实际参数指定的，那么该方法将返回 true。否则，它将返回 false。例如：

```
var o = new Object();
o.hasOwnProperty('undef'); // false: 没有定义该属性
o.hasOwnProperty('toString'); // false: toString 是一个继承属性
Math.hasOwnProperty('cos'); // true: Math 对象有 cos 属性
```

8.7.6 propertyIsEnumerable()方法

如果对象定义了一个属性，属性名是由一个字符串实际参数指定的，而且该属性可以用 for/in 循环枚举出来，那么该方法返回 true，否则，返回 false。例如：

```
var o = { x:1 };
o.propertyIsEnumerable('x');           // true: 该属性存在, 而且可枚举
o.propertyIsEnumerable('');             // false: 该属性不存在
o.propertyIsEnumerable('valueOf');     // false: 该属性不可枚举
```

注意, ECMAScript 标准规定 `propertyIsEnumerable()` 方法只考虑对象直接定义的属性, 而不考虑继承的属性。这个规定降低了函数的有用程度, 因为返回 `false` 可能是因为那个属性是不可枚举的, 也可能是因为它虽然可枚举, 但却是个继承的属性。

8.7.7 isPrototypeOf() 方法

如果调用对象是实际参数指定的对象的原型对象, 该方法返回 `true`, 否则返回 `false`。该方法的用途和对象的 `constructor` 属性相似。例如:

```
var o = new Object();
Object.prototype.isPrototypeOf(o);    // true: o.constructor == Object
Object.isPrototypeOf(o);              // false
o.isPrototypeOf(Object.prototype);    // false
Function.prototype.isPrototypeOf(Object); // true: Object.constructor == Function
```

第九章

数组

第八章介绍了JavaScript的对象类型，即一种包含已命名的值的复合数据类型。本章将介绍数组类型，即一种包含已编码的值的复合数据类型。注意，本章中讨论的数组不同于上一章中介绍的关联数组。关联数组是将值和字符串关联在一起。而本章中介绍的数组只不过是常规的数字数组，它们是将值和非负整数关联在一起的。

在本书中，我们常常将对象和数组作为不同的数据类型来处理。这是一种有用而合理的简化。这样一来，你就可以在大多数的JavaScript程序设计中将对象和数组作为单独的类型来处理。但是要完全掌握对象和数组的行为，你还必须了解数组不过是一个具有额外功能层的对象。当我们使用typeof运算符时就会发现这一点，因为将之作用于一个数组的值，其返回值是字符串“object”。注意，数组的额外功能是在JavaScript 1.1中引入的。JavaScript 1.0不支持数组。

本章介绍了基本的数组语法、数组程序设计方法和对数组进行操作的方法。

9.1 数组和数组元素

数组(array)是一种数据类型，它包含或者存储了编码的值。每个编码的值称作该数组的一个元素(element)，每个元素的编码被称作下标(index)。由于JavaScript是一种无类型语言，所以一个数组的元素可以具有任意的数据类型，同一数组的不

同元素可以具有不同的类型。数组的元素甚至可以包含其他数组，这样就可以创建一个复杂的数据结构，即元素为数组的数组。

9.1.1 数组的创建

在 JavaScript 1.1 和其后的版本中，数组是用构造函数 `Array()` 和运算符 `new` 创建的。你可以用三种不同的方式来调用构造函数 `Array()`。第一种方式是无参数调用：

```
var a = new Array();
```

用这种方法创建的是一个没有元素的空数组。

调用构造函数 `Array()` 的第二种方法可以明确地指定数组前 n 个元素的值：

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

这种形式的构造函数都带有一个参数列表。每个参数都指定了一个元素值，它可以是任何类型的。给数组赋值时是从元素 0 开始的。数组的 `length` 属性值为传递给构造函数的参数个数。

调用构造函数 `Array()` 的第三种方式是传递给它一个数字参数，这个数字指定了数组的长度：

```
var a = new Array(10);
```

采用这一方法创建的数组具有指定的元素个数（每个元素的值都是 `undefined`）。它还将数组的 `length` 属性设置成了指定的值（注 1）。

最后，数组直接量提供了一种创建数组的新方式。数组直接量使你直接将一个数组的值嵌入 JavaScript 程序，就像把字符串文本放入引号之间来定义一个字符串直接量一样。要创建一个数组直接量，只需要将一个用逗号分隔的值列表放入方括号之间即可。例如：

注 1：在 Netscape 的客户端 JavaScript 中，如果 `<SCRIPT>` 标记的 `language` 性质被显式地设置为 "JavaScript 1.2"，那么 `Array()` 构造函数的第三种形式的行为与第二种形式相同，即创建一个长度为 1 的数组，并将数组元素初始化为构造函数的参数。这与 ECMAScript 标准不一致。

```
var primes = [2, 3, 5, 7, 11];  
var a = [a, true, 1.78];
```

数组直接量还可以包含对象直接量或其他的数组直接量：

```
var b = [{1, {x:1, y:2}}, [2, {x:3, y:4}]];
```

第三章提供了数组直接量的完整说明。

9.1.2 数组元素的读和写

可以使用[]运算符来存取数组元素。在方括号左边应该是对数组的引用。方括号之中是具有非负整数值的任意表达式。你既可以使用这一语法来读一个数组元素，也可以用它来写一个数组元素。下面列出的都是合法的 JavaScript 语句：

```
value = a[0];  
a[1] = 3.14;  
i = 2;  
a[i] = 3;  
a[i + 1] = 'hello';  
a[a[i]] = a[0];
```

在某些语言中，数组第一个元素的下标为 1。但是在 JavaScript 中（和 C、C++ 与 Java 一样）数组第一个元素的下标是 0。

我们在第八章中见到过，运算符[]还可以用来存取已命名对象的属性：

```
my['salary'] *= 2;
```

这是一个线索，告诉我们对象和数组本质上是相同的。

注意，数组的下标必须是大于等于 0 并小于 $2^{32}-1$ 的整数，如果你使用的数字太大，或使用了负数、浮点数（或布尔值、对象及其他值），JavaScript 会将它转换为一个字符串，用生成的字符串作为对象属性的名字，而不是作为数组下标。因此，下面的代码创建了一个名为“-1.23”的属性，而不是定义了一个新的数组元素：

```
a[-1.23] = true;
```

9.1.3 添加数组新元素

在像C和Java这样的语言中，数组是具有固定的元素个数的，你必须在创建数组时就指定它的元素数。而在JavaScript中则不同，它的数组可以具有任意个数的元素，你可以在任何时刻改变元素个数。

要给一个数组添加新的元素，只需要给它赋一个值即可：

```
a[10] = 10;
```

在JavaScript中数组是稀疏的 (sparse)。这意味着数组的下标不会落在一个连续的数字范围内，只有那些真正存储在数组中的元素才能够分配到内存。因此，当你执行下面的几行代码时，JavaScript解释器只给数组下标为0和10 000的元素分配内存，而并不给下标在0和10 000之间的那9 999个元素分配内存：

```
a[0] = 1;  
a[10000] = "this is element 10,000";
```

注意，数组元素也可以被添加到对象中：

```
var c = new Circle(1,2,3);  
c[0] = "this is an array element of an object!";
```

但是这个例子只是定义了一个名为“0”的对象属性。只将数组元素添加到一个对象中并不会使它成为数组。由构造函数Array()或由一个数组直接量创建的数组具有一些对象所不能享有的特性，下面将对此进行解释。

9.1.4 数组的长度

所有的数组（无论是由构造函数Array()创建的，还是由数组直接量创建的）都有一个特殊的属性length，用来说明这个数组包含的元素个数。更为精确地说，由于数组可能含有未定义的元素，所以属性length总是比数组的最大元素的数多1。和常规对象的属性不同，数组的length属性是自动更新的，以便在给数组添加新元素时保持不便。下面的代码说明了这一点：

```
var a = new Array();           // a.length == 0   (没有定义元素)  
a = new Array(10);            // a.length == 10 (将0-9的元素定义为空元素)  
a = new Array(1,2,3);         // a.length == 3   (定义0-2的元素)  
a = [4, 5];                   // a.length == 2   (定义元素0和元素1)
```

```
a[5] = 1;           // a.length == 6 (定义元素0、1和5)
a[49] = 0;          // a.length == 50 (定义元素0、1、5和49)
```

回忆一下，数组下标必须小于 $2^{32} - 1$ ，这意味着 length 属性的最大值是 $2^{32} - 1$ 。

一个数组的 length 属性最常见的用法就是遍历数组元素：

```
var fruits = ['mango', 'banana', 'cherry', 'pear'];
for(var i = 0; i < fruits.length; i++)
    alert(fruits[i]);
```

当然，这个例子是假定数组的元素是连续的，而且是从元素 0 开始。如果情况不是这样，那么在使用数组元素之前就需要检测一下，看每个元素是否被定义了：

```
for(var i = 0; i < fruits.length; i++)
    if (fruits[i] != undefined) alert(fruits[i]);
```

数组的 length 属性既可以读也可以写。如果给 length 设置了一个比它的当前值小的值，那么数组将会被截断，这个长度之外的元素都会被抛弃，它们的值也就丢失了。如果给 length 设置的值比当前值大，那么新的、未定义的元素就会被添加到数组末尾以使得数组增长到新指定的长度。

通过设置数组的 length 属性来截断数组是惟一种缩短数组长度的方法。如果使用 delete 运算符来删除数组中的元素，虽然那个元素变成未定义的，但是数组的 length 属性并不会改变。

注意，尽管可以将对象赋给数组元素，但是对象并没有 length 属性。就是这点特殊行为而言，length 属性成了数组最重要的特性。使数组有别于其他对象的特性还有 Array 类定义的各种方法，9.2 节将介绍它们。

9.1.5 多维数组

虽然 JavaScript 并不支持真正的多维数组，但是它允许使用元素为数组的数组，这就非常接近多维数组。要存取一个元素为数组的数组的元素，只需要使用两次 [] 运算符即可。例如，假设变量 matrix 是一个元素为数数字数组的数组，它的每个元素 matrix[x] 都是一个数数字数组。要存取这个数组中的一个数字，就要写成 matrix[x][y]。

9.2 数组的方法

除了[]运算符之外,还可以使用类Array提供的各种方法来操作数组。下面的几节对这些方法做了介绍。其中的许多方法受到了Perl程序设计语言的启示,Perl程序员可能会发现它们是如此熟悉。和前面一样,这里所讲的不过是一个概述,可以在核心参考手册中找到完整的介绍。

9.2.1 join() 方法

方法Array.join()可以把一个数组的所有元素都转换成字符串,然后再把它们连接起来。你可以指定一个可选的字符串来分隔结果字符串中的元素。如果没有指定分隔字符串,那么可以使用逗号分隔元素。例如,下面的几行代码将生成字符串“1, 2, 3”:

```
var a = [1, 2, 3];    // 用这三个元素创建一个新数组
var s = a.join();     // s == "1,2,3"
```

下面的调用指定了一个分隔符,其生成的结果稍有不同:

```
s = a.join(", ");    // s=="1, 2, 3"
```

注意,逗号后面有一个空格。方法Array.join()恰好与方法String.split()相反,后者是通过将一个字符串分割成几个片段来创建数组。

9.2.2 reverse()方法

方法Array.reverse()将颠倒数组元素的顺序并返回颠倒后的数组。它在原数组上执行这一操作,也就是说,它并不是创建一个重排了元素的新数组,而是在已经存在的数组中对数组元素进行重排。例如,下面的代码使用了方法reverse()和join()来生成字符串“3, 2, 1”:

```
var a = new Array(1,2,3);    // a[0] = 1, a[1] = 2, a[2] = 3
a.reverse();                 // 现在a[0] = 3, a[1] = 2, a[2] = 1
var s = a.join();            // s == "3,2,1"
```

9.2.3 sort() 方法

`Array.sort()`是在原数组上对数组元素进行排序，返回排序后的数组。如果调用`sort()`时不传递给它参数，那么它将按照字母顺序对数组元素进行排序（如果必要的话，可以暂时将元素转换成字符串以执行比较操作）：

```
var a = new Array('banana', 'cherry', 'apple');
a.sort();
var s = a.join(', '); // s -- "apple, banana, cherry"
```

如果数组含有未定义的元素，这些元素将被放在数组的末尾。

如果要将数组按照别的顺序来排序，必须将一个比较函数作为参数传递给`sort()`。该函数确定它的两个参数在排序数组中哪个在前，哪个在后。如果第一个参数应该位于第二个参数之前，那么比较函数将返回一个小于0的数。如果第一个参数应该出现在第二个参数之后，那么比较函数就会返回一个大于0的数。如果两个参数相等（例如它们的顺序是相等的），那么比较函数将返回0。例如，要将一个数组按照数字顺序进行排序，而不是按照字母顺序进行排序，应该使用如下的代码：

```
var a = [33, 4, 1111, 222];
a.sort(); // 按字母排序: 1111, 222, 33, 4
a.sort(function(a,b) { // 按数字排序: 4, 33, 222, 1111
    return a-b; // 根据排序返回 < 0, 0, 或 > 0
});
```

注意，在上面的代码中使用了函数直接量，这非常方便。由于这里只使用了一次比较函数，所以没有必要给它起名字。

作为对数组元素进行排序的另一个例子，还可以对一个字符串数组执行不区分大小写的字母排序操作，只需要在比较两个参数之前，传递可以将参数转换为小写的比较函数（使用`toLowerCase()`即可）。你还可能会用到那些可以将数字排成各种奇怪的顺序的比较函数，如颠倒的数字顺序、奇数排在偶数之前的顺序等等。当然，如果要比较的元素是对象而不是像数字和字符串那样的简单类型时，这种可能性就变得更加有趣了。

9.2.4 concat()方法

方法 `Array.concat()` 能创建并返回一个数组，这个数组包含了调用 `concat()` 的原始数组的元素，其后跟随的是 `concat()` 的参数。如果其中有些参数是数组，那么它将被展开，其元素将被添加到返回的数组中。但是要注意，`concat()` 并不能递归地展开一个元素为数组的数组。下面是一些例子：

```
var a = [1,2,3];
a.concat(4, 5)           // 返回[1,2,3,4,5]
a.concat([4,5]);         // 返回[1,2,3,4,5]
a.concat([1,2],[6,7])    // 返回[1,2,3,1,2,6,7]
a.concat(4, [5,[6,7]])   // 返回[1,2,3,4,5,[6,7]]
```

9.2.5 slice()方法

方法 `Array.slice()` 返回的是指定数组的一个片段 (slice)，或者说是子数组。它的两个参数指定了要返回的片段的起止点。返回的数组包含由第一个参数指定的元素和从那个元素开始到第二个参数指定的元素为止的元素，但是并不包含第二个参数所指定的元素。如果只传递给它一个参数，那么返回的数组将包含从起始位置开始到原数组结束处的所有的元素。如果两个参数中有一个是负数，那么它所指定的是相对于数组中的最后一个元素而言的元素。例如，参数值为 `-1` 指定的是数组的最后一个元素，而参数值为 `-3`，指定的是从数组的最后一个元素数起，倒数第三个元素。下面是一些例子：

```
var a = [1,2,3,4,5];
a.slice(0,3);             // 返回[1,2,3]
a.slice(3);               // 返回[4,5]
a.slice(1, 4);            // 返回[2,3,4]
a.slice(-3,-2);           // 返回[3]
```

9.2.6 splice()方法

方法 `Array.splice()` 是插入或删除数组元素的通用方法。它在原数组上修改数组，就像 `slice()` 和 `concat()` 那样并不创建新数组。注意，虽然 `splice()` 和 `slice()` 名字非常相似，但是执行的却是完全不同的操作。

`splice()` 可以把元素从数组中删除，也可以将新元素插入到数组中，或者是同时执行这两种操作。位于被插入了或删除了的元素之后的数组元素会进行必要的移动，

以便能够和数组余下的元素保持连续性。splice()的第一个参数指定了要插入或删除的元素在数组中的位置。第二个参数指定了要从数组中删除的元素个数。如果第二个参数被省略了,那么将删除从开始元素到数组结尾处的所有元素。splice()返回的是删除了元素之后的数组,如果没有删除任何元素,将返回一个空数组。例如:

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);      // 返回[5,6,7,8]; a 是 [1,2,3,4]
a.splice(1,2);    // 返回[2,3]; a 是 [1,4]
a.splice(1,1);    // 返回[4]; a 是 [1]
```

splice()的前两个参数指定了应该删除的数组元素。这两个参数之后还可以有任意多个额外的参数,它们指定的是要从第一个参数指定的位置处开始插入的元素。例如:

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b'); // 返回[]; a 是 [1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3); // 返回['a','b']; a 是 [1,2,[1,2],3,3,4,5]
```

注意,和concat()不同,splice()并不将它插入的数组参数展开,也就是说,如果传递给它的是要插入的一个数组,那么它插入的是这个数组本身,而不是这个数组的元素。

9.2.7 push()方法和pop()方法

push()和pop()方法使我们可以像使用栈那样来使用数组。方法push()可以将一个或多个新元素附加到数组的尾部,然后返回数组的新长度(注2)。方法pop()恰恰相反,它将删除数组的最后一个元素,减少数组的长度,返回它删除的值。注意,这两个方法都是在原数组上修改数组,而非生成一个修改过的数组副本。联合使用push()和pop(),就可以用JavaScript数组实现一个先进后出(FILO)栈。例如:

```
var stack = [];           // 堆栈: []
stack.push(1,2);          // 堆栈: [1,2]; 返回 2
stack.pop();              // 堆栈: [1]; 返回 2
stack.push(3);            // 堆栈: [1,3]; 返回 2
stack.pop();              // 堆栈: [1]; 返回 3
```

注2: 在Netscape中,如果<script>标记的language特性被显式地设置为“JavaScript 1.2”,那么push()方法将返回附加到数组的最后一个值,而不是数组的新长度。

```
stack.push([4,5]);      // 堆栈: [1,[4,5]]; 返回2
stack.pop()             // 堆栈: [1]; 返回[4,5]
stack.pop()             // 堆栈: []; 返回1
```

9.2.8 unshift()方法和shift()方法

unshift()和shift()方法的行为和push()与pop()非常相似,只不过它们是在数组的头部进行元素的插入和删除,而不是在尾部进行元素的插入和删除。方法unshift()会将一个或多个元素添加到数组的头部,然后把已有的元素移动到下标较大的位置以腾出空间,它返回的是数组的新长度。方法shift()会删除并返回数组的第一个元素,然后将后面的所有元素都向前移动以填补第一个元素留下的空白。

例如:

```
var a = [];              // a:[]
a.unshift(1);            // a:[1]; 返回1
a.unshift(22);           // a:[22,1]; 返回2
a.shift();               // a:[]; 返回22
a.unshift(3,[4,5]);      // a:[3,[4,5],1]; 返回3
a.shift();               // a:[[4,5],1]; 返回3
a.shift();               // a:[1]; 返回[4,5]
a.shift();               // a:[]; 返回1
```

注意使用多个参数调用unshift()时它的行为。这些参数是被同时插入的(和splice()方法一样),而不是一次只插入一个元素。这意味着参数在结果数组中的顺序和它们在参数列表中的顺序相同。如果一次只插入一个元素,那么它们在结果数组的顺序恰好与参数列表中的顺序相反。

9.2.9 toString()方法和toSource()方法

和所有的JavaScript对象一样,数组也有toString()方法。这个方法可以将数组的每个元素都转换成一个字符串(如果必要的话,就调用它的元素的toString()方法),然后输出这些字符串的列表,字符串之间用逗号隔开。注意,在输出的结果中,数组值的周围没有方括号或者其他定界符。例如:

```
[1,2,3].toString()      // 生成 '1,2,3'
['a', 'b', 'c'].toString() // 生成 'a,b,c'
[1, [2,'c']].toString()  // 生成 '1,2,c'
```

注意, `toString()` 的返回值和无参数调用方法 `join()` 时返回的字符串相同(注3)。

`toLocaleString()` 是 `toString()` 方法局部化的版本。它将调用每个元素的 `toLocaleString()` 方法把数组元素转换成字符串, 然后把生成的字符串用局部特定(和定义的实现)的分隔符字符串连接起来。

注3: 在 Netscape 中, 如果把 `<script>` 标记的 `language` 性质设置成 “JavaScript 1.2”, `toString()` 的行为就更加复杂。在这种情况下, 它将数组转换成包含方括号的字符串, 这样结果字符串就是一个有效的数组直接量表达式。

第十章

使用正则表达式 的模式匹配

正则表达式(regular expression)是一个描述字符模式的对象。JavaScript的RegExp类表示正则表达式,而String和RegEXP都定义了使用正则表达式进行强大的模式匹配和文本检索与替换的函数(注1)。

ECMAScript v3 对 JavaScript 正则表达式进行了标准化。JavaScript 1.2 实现了 ECMAScript v3 要求的正则表达式特性的子集,JavaScript 1.5 实现了完整的标准。JavaScript 的正则表达式完全以 Perl 程序设计语言的正则表达式工具为基础。坦白地说,JavaScript 1.2 实现了 Perl 4 的正则表达式,JavaScript 1.5 实现了 Perl 5 的正则表达式的大型子集。

本章定义了正则表达式用来描述文本模式的语法。它还介绍了使用正则表达式的String与RegExp方法。

10.1 正则表达式的定义

在 JavaScript 中,正则表达式由 RegExp 对象表示。当然,可以使用 RegExp() 构造

注1: “正则表达式”这个术语比较模糊,这得追溯到许多年以前。因为描述文本模式的语法确实是一种正则表达式。但是,如我们所见,该语法没有一点规则之处。有时正则表达式被称为“regex”,或“RE”。

函数创建 `RegExp` 对象，不过通常还是用特殊的直接量语法来创建 `RegExp` 对象。就像字符串直接量被定义为包含在引号内的字符一样，正则表达式直接量也被定义为包含在一对斜杠（/）之间的字符。所以，JavaScript 可能会包含如下的代码：

```
var pattern = /s$/;
```

这行代码创建了一个新的 `RegExp` 对象，并且将它赋给了变量 `pattern`。这个特殊的 `RegExp` 对象和所有的以字母“s”结尾的字符串都匹配（很快我们就会讨论模式定义的语法）。用构造函数 `RegExp()` 也可以定义一个等价的正则表达式，其代码如下：

```
var pattern = new RegExp('s$');
```

无论是用正则表达式直接量还是用构造函数 `RegExp()`，创建一个 `RegExp` 对象都比较容易。较为困难的是用正则表达式语法来描述字符的模式。JavaScript 采用的是 Perl 语言使用的正则表达式语法的相当完整的子集，因此，如果你是一个经验丰富的 Perl 程序员，那么就会知道在 JavaScript 中如何描述模式。

正则表达式的模式规范是由一系列字符构成的。大多数的字符（包括所有的字母数字字符）描述的都是按照直接量进行匹配的字符。这样说来，正则表达式 `/java/` 就和所有的包含子串“java”的字符串相匹配。虽然正则表达式中的其他字符不是按照直接量进行匹配的，但是它们都具有特殊的意义。正则表达式 `/s$/` 包含两个字符。第一个字符“s”按照直接量与自身相匹配。第二个字符“\$”是一个特殊元字符，它所匹配的是字符串的结尾。所以正则表达式 `/s$/` 匹配的就是以字母“s”结尾的字符串。

接下来的几节介绍了用于 JavaScript 正则表达式的各种字符和元字符。但是注意，有关正则表达式语法的完整介绍远远超出了本书的范围。要了解该语法的完整细节，可以参考有关 Perl 语言的书，如由 Larry Wall、Tom Christiansen 和 Jon Orwant（O'Reilly）编写的《Programming Perl》。由 Jeffrey E.F. Friedl（O'Reilly）编写的《Mastering Regular Expressions》也是一本不错的介绍正则表达式的参考书。

10.1.1 直接量字符

我们已经知道，在正则表达式中所有的字母字符和数字都是按照直接量与自身相匹

配的。JavaScript 的正则表达式语法还通过以反斜杠 (\) 开头的转义序列支持某些非字母的字符。例如, 序列 “\n” 在字符串中匹配的是直接量换行符。表 10-1 列出了这些字符。

表 10-1: 正则表达式的直接量字符

| 字符 | 匹配 |
|--------|--|
| 字母数字字符 | 自身 |
| \o | NUL 字符 (\u0000) |
| \t | 制表符 (\u0009) |
| \n | 换行符 (\u000A) |
| \v | 垂直制表符 (\u000B) |
| \f | 换页符 (\u000C) |
| \r | 回车 (\u000D) |
| \xnn | 由十六进制数 nn 指定的拉丁字符, 例如, \x0A 等价于 \n |
| \uxxxx | 由十六进制 xxxx 指定的 Unicode 字符, 例如, \u0009 等价于 \t |
| \cX | 控制字符 ^X。例如, \cJ 等价于换行符 \n |

在正则表达式中, 许多标点符号具有特殊的含义。它们是:

`^ $. * + ? - ! : | \ / () [] { }`

在接下来的几节中, 我们将学习这些符号的含义。某些符号只在正则表达式的特殊环境中才具有特殊的含义, 在其他环境中则被按照直接量进行处理。但是, 作为一个通用的原则, 如果你在正则表达式中按照直接量使用这些标点符号, 就必须加前缀 `a\`。其他标点符号 (如引号和 @) 没有特殊含义, 在正则表达式中只按照直接量匹配自身。

如果你记不清楚哪些标点符号需要用反斜杠转义, 可以在每个标点符号之前都使用反斜杠。另一方面要注意, 许多字母和数字在有反斜杠做前缀时也具有特殊含义, 所以对于想按照直接量进行匹配的字母和数字, 不要用反斜杠转义。当然, 要在正则表达式中添加按照直接量理解的反斜杠, 必须用反斜杠将其转义。例如, 正则表达式 “\V” 与任何包含反斜杠的字符串匹配。

10.1.2 字符类

将单独的直接量字符放进方括号内就可以组合成字符类 (character class)。一个字符类和它所包含的任何字符都匹配。所以正则表达式 `/[abc]/` 就和字母“a”、“b”、“c”中的任何一个字母都匹配。另外，还可以定义否定字符类，这些类匹配的是不包含在方括号之内的所有字符。定义否定字符类的时候，要将一个 `^` 符号作为左方括号后的第一个字符。正则表达式 `/[^abc]/` 匹配的是“a”、“b”、“c”之外的所有字符。字符类可以使用连字符来表示一个字符范围。要匹配拉丁字母集中的任何小写字符，可以使用 `/[a-z]/`，要匹配拉丁字母集中任何字母数字字符，则使用 `/[a-zA-Z0-9]/`。

由于某些字符类非常常用，所以 JavaScript 的正则表达式语法就包含了一些特殊字符和转义序列来表示这些常用的类。例如，`\s` 匹配的是空格符、制表符和其他 Unicode 空白符，`\S` 匹配的是非 Unicode 空白符。表 10-2 列出了这些字符，并且总结了字符类的语法。（注意，有些字符类转义序列只匹配 ASCII 字符，还没有扩展到可以处理 Unicode 字符。你可以显式地定义自己的 Unicode 字符类，例如，`/[\u0400-04FF]/` 匹配所有的 Cyrillic 字符。）

表 10-2: 正则表达式的字符类

| 字符 | 匹配 |
|---------------------|---|
| <code>[...]</code> | 位于括号之内的任意字符 |
| <code>[^...]</code> | 不在括号之中的任意字符 |
| <code>.</code> | 除换行符和其他 Unicode 行终止符之外的任意字符 |
| <code>\w</code> | 任何 ASCII 单字字符，等价于 <code>[a-zA-Z0-9_]</code> |
| <code>\W</code> | 任何 ASCII 非单字字符，等价于 <code>[^a-zA-Z0-9_]</code> |
| <code>\s</code> | 任何 Unicode 空白符 |
| <code>\S</code> | 任何非 Unicode 空白符，注意 <code>\w</code> 和 <code>\S</code> 不同 |
| <code>\d</code> | 任何 ASCII 数字，等价于 <code>[0-9]</code> |
| <code>\D</code> | 除了 ASCII 数字之外的任何字符，等价于 <code>[^0-9]</code> |
| <code>[\b]</code> | 退格直接量（特例） |

注意，在方括号之内也可以使用这些特殊的字符类转义序列。例如 `\s` 匹配的是所有

的空白符，`\d`匹配的是所有数字，那么`/[\s\d]/`就匹配任意的空白符或数字。注意，这里有一个特例。下面我们将会看到转义序列`\b`具有特殊含义，当用在字符串中时，它表示的是退格符，所以要在正则表达式中按照直接量表示一个退格符，只需要使用具有一个元素的字符类`/[\b]/`。

10.1.3 重复

用刚刚学过的正则表达式的语法，可以把两位数描述成`/\d\d/`，把四位数描述成`/\d\d\d\d/`。但是还没有一种方法可以用来描述具有任意多数位的数字，或描述字符串，这个字符串由三个字母以及跟随在字母之后的一位数字构成。这些较复杂的模式使用的正则表达式语法都指定了该正则表达式中的一个元素重复出现的次数。

指定重复的字符总是出现在它们所作用的模式之后。由于某种重复类型相当常用，所以有一些特殊的字符专门用于表示这种情况。例如，`+`号匹配前一个模式的一个或多个副本。表 10-3 总结了重复语法。下面的代码是一些例子：

```
//m{2,4}/      // 与 2 和 4 之间的数字匹配
//w{3}\d?/     // 与三个字符和一个数字匹配
//s-java\s+/   // 与字符串“java”匹配，并且该串前后可以有一个或多个空格
//.*{1,}/      // 与零个或多个非引号字符匹配
```

表 10-3: 正则表达式的重复字符

| 字符 | 含义 |
|--------------------|---|
| <code>{n,m}</code> | 匹配前一项至少 n 次，但是不能超过 m 次 |
| <code>{n,}</code> | 匹配前一项 n 次，或更多次 |
| <code>{n}</code> | 匹配前一项恰好 n 次 |
| <code>?</code> | 匹配前一项 0 次或 1 次，也就是说前一项是可选。等价于 $\{0,1\}$ |
| <code>+</code> | 匹配前一项 1 次或多次。等价于 $\{1, \}$ |
| <code>*</code> | 匹配前一项 0 次或多次。等价于 $\{0, \}$ |

在使用重复字符 `*` 和 `?` 时要注意，由于这些字符可能匹配前面字符的 0 个实例，所以它们允许什么都不匹配。例如，正则表达式 `/a*/` 实际上与字符串“bbbb”匹配，因为这个字符串含有 0 个 `a`。

10.1.3.1 非贪婪的重复

表10-3中列出的重复字符可以匹配尽可能多的字符,而且允许接下来的正则表达式继续匹配。因此,我们说重复是“贪婪的”。可以以非贪婪的方式进行重复(在JavaScript 1.5和其后的版本中可以做到,这是Perl 5的一个特性,JavaScript 1.2没有实现它),只需要在重复字符后加问号即可(如`??`、`-?`、`*?`,甚至`(1,5)?`)。例如,正则表达式`/a-/`匹配一个或多个字符`a`。将其应用到字符串“aaa”上时,它与三个字母都匹配。但是`/a+?/`只匹配一个或多个必要的字母`a`。将其应用到同样的字符串上时,该模式只匹配第一个字母`a`。

使用非贪婪的重复生成的结果并不总是与你期望的一致。考虑模式`/a*b/`,它匹配0个或多个字母`a`后跟随字母`b`。在应用到字符串“aaab”上时,它匹配整个字符串。现在使用非贪婪的重复版本`/a*b?/`,它应该匹配字母`b`,通过在字母`b`前加最少的字母`a`。在应用到同一个字符串“aaab”上时,你可能以为它只匹配最后一个字母`b`。但事实上,这个模式也匹配整个字符串,和该模式的贪婪版本一样。这是因为正则表达式模式匹配是在寻找字符串中第一个可能匹配的位置。该模式的非贪婪版本在字符串的第一个字符处不匹配,所以该匹配将返回,甚至不考虑对后面的字符进行匹配。

10.1.4 选择、分组和引用

正则表达式的语法还包括指定选择项、对子表达式分组和引用前一个表达式的特殊字符。字符“`|`”用于分隔供选择的字符。例如,`/ab|cd|ef/`匹配的是字符串“ab”,或者是字符串“cd”,又或者是字符串“ef”。`/\d{3}|[a-z]{4}/`匹配的是三位数字或四个小写字母。

注意,选择项是从左到右考虑,直到发现了匹配项。如果左边的选择项匹配,就忽略右边的匹配项,即使它产生更好的匹配。因此,把模式`/a|ab/`应用到字符串“ab”上时,它只匹配第一个字符。

在正则表达式中括号具有几种作用。一个作用是把单独的项目组合成子表达式,以便可以像处理一个独立的单元那样用`|`、`*`、`+`或`?`等来处理它们。例如,`/java(script)?/`匹配的是字符串“java”,其后既可以有“script”,也可以没有。

`/(ab|cd)+|ef)/` 匹配的既可以是字符串“ef”，也可以是字符串“ab”或者“cd”的一次或多次重复。

在正则表达式中，括号的另一个作用是在完整的模式中定义子模式。当一个正则表达式成功地和目标字符串相匹配时，可以从目标串中抽出和括号中的子模式相匹配的部分（我们将在本章后边的部分中看到如何取得这些匹配的子串）。例如，假定我们正在检索的模式是一个或多个小写字母后面跟随了一位或多位数字，则可以使用模式 `/[a-z]+\d+/`。但是假定我们真正关心的是每个匹配尾部的数字，那么如果将模式的数字部分放在括号中（`/[a-z]+(\d+)/`），就可以从所检索到的任何匹配中抽取数字了，之后我们会对此进行解释。

带括号的子表达式的另一个用途是允许我们在同一正则表达式的后部引用前面的子表达式。这是通过在字符“\”后加一位或多位数字实现的。数字指定了带括号的子表达式在正则表达式中的位置。例如，`\1` 引用的是第一个带括号的子表达式，`\3` 引用的是第三个带括号的子表达式。注意，由于子表达式可以被嵌套在别的子表达式中，所以它的位置是被计数的左括号的位置。例如，在下面的正则表达式中，嵌套的子表达式 `([Ss]cript)` 就被指定为 `\2`：

```
/(j|java|([Ss]cript)?)\.sis\s(fur\w*)
```

对正则表达式中前一子表达式的引用所指的并不是那个子表达式的模式，而是与那个模式相匹配的文本。这样，引用可以用于实施一条约束，即一个字符串各个单独的部分包含的是完全相同的字符。例如，下面的正则表达式匹配的就是位于单引号或双引号之内的0个或多个字符。但是，它并不要求开始和结束的引号匹配（例如两个都是双引号或者两个都是单引号）：

```
/(''|''|"[^"]*"|'['']*')+/
```

如果要求开始和结束的引号相匹配，可以使用如下的引用：

```
/(''|''|"[^"]*"|'['']*')\1/
```

`\1` 匹配的是第一个带括号的子表达式所匹配的模式。在这个例子中，它实施了一条约束，那就是开始的引号必须和结束的引号相匹配。正则表达式不允许用双引号括起的字符串中有单引号，反之亦然。在字符类中使用引用是不合法的，所以我们不能编写：

符，它们指定的是匹配所发生的合法位置。有时我们称这些元素为正则表达式的锚，因为它们将模式定位在检索字符串中的一个特定位置上。最常用的锚元素是 `^`，它使模式定位在字符串的开头，而锚元素 `$` 则使模式定位在字符串的末尾。

例如，要匹配单字“JavaScript”，可以使用正则表达式 `/^JavaScript$/`。如果想检索“Java”这个单字自身（不像在“JavaScript”中那样作为前缀），可以使用模式 `/\sJava\s/`，它要求在单字Java之前和之后都要有空格。但是这样做有两个问题。第一，如果“Java”出现在一个字符串的开头或结尾，该模式就会不与之匹配，除非在开头处或者结尾处有一个空格。第二个问题是，当这个模式找到了一个与之匹配的字符串时，它返回的匹配字符串的前端和后端都有空格，这并不是我们想要的。因此，我们使用单字的边界 `\b` 来代替真正的空格符 `\s` 进行匹配。结果表达式是 `/\bJava\b/`。元素 `\B` 将把匹配锚定在不是单字边界的位置。因此，模式 `/\B[Sc]ript/` 与“JavaScript”和“postscript”匹配，但是不与“script”和“Scripting”匹配。

在JavaScript 1.5中（不是JavaScript 1.2），还可以使用任意的正则表达式作为锚定条件。如果在符号（`?`和`=`）之间加入一个表达式，它就是一个前向声明，指定接下来的字符必须被匹配，但并不真正进行匹配。例如，要匹配一种常用的程序设计语言的名字，但只在其后有冒号时匹配，可以使用 `/[Jj]ava([Ss]cript)?(?=\:)/`。这个模式与“JavaScript: The Definitive Guide”中的单字“JavaScript”匹配，但是与“Java in a Nutshell”中的“Java”不匹配，因为其后没有冒号。

如果你不是用（`?!`）引入声明，它将是反前向声明，指定接下来的字符都不必匹配。例如，`/Java(?!Script)([A-Z]\w*)/`匹配的是“Java”后跟随一个大写字母和任意多个ASCII单字字符，但是不能跟随“Script”。它与“JavaBeans”匹配，不与“Javanese”匹配，与“JavaScrip”匹配，但不与“JavaScript”或“JavaScripter”匹配。

表 10-5 总结了正则表达式的锚元素。

表 10-5: 正则表达式的锚字符

| 字符 | 含义 |
|-----------------|-------------------------|
| <code>^</code> | 匹配字符串的开头，在多行检索中，匹配一行的开头 |
| <code>\$</code> | 匹配字符串的结尾，在多行检索中，匹配一行的结尾 |

表 10-5: 正则表达式的锚字符 (续)

| 字符 | 含义 |
|---------------------|--|
| <code>\b</code> | 匹配一个词语的边界。简而言之, 就是位于字符 <code>\w</code> 和 <code>\w</code> 之间的位置, 或位于字符 <code>\w</code> 和字符串的开头或结尾之间的位置 (但注意, <code>\b</code> 匹配的是退格符) |
| <code>\B</code> | 匹配非词语边界的字符 |
| <code>(? p)</code> | 正前向声明, 要求接下来的字符都与模式 <code>p</code> 匹配, 但是不包括匹配中的那些字符 |
| <code>(?! p)</code> | 反前向声明, 要求接下来的字符不与模式 <code>p</code> 匹配 |

10.1.6 标志

正则表达式的语法还有最后一个元素, 即正则表达式的标志, 它说明高级模式匹配的规则。和其他的正则表达式语法不同, 标志是在 “/” 符号之外说明的, 即它们不出现在两个斜杠之间, 而是位于第二个斜杠之后。JavaScript 1.2 支持两个标志。标志 `i` 说明模式匹配应该是大小写不敏感的字符。标志 `g` 说明模式匹配应该是全局的, 也就是说, 应该找出被检索的字符串中所有的匹配。这两种标志联合起来就可以执行一个全局的、大小写不敏感的匹配。

例如, 要执行一个大小写不敏感的检索以找到单字 “java” (或者是 “Java”、“JAVA” 等) 的第一个具体值, 可以使用大小写不敏感的正则表达式 `/\bjava\b/i`。如果要在一个字符串中找到 “java” 所有的具体值, 需要添加标志 `g`, 即 `/\bjava\b/gi`。

JavaScript 1.5 还支持一个标志 `m`, 它以多行模式执行模式匹配。在这种模式中, 如果要检索的字符串中含有换行符, `^` 和 `$` 锚除了匹配字符串的开头和结尾外还匹配每行的开头和结尾。例如, 模式 `/Java$/im` 匹配 “java” 和 “Java\nis fun”。

表 10-6 总结了这些正则表达式的标志。注意, 我们在本章后面的小节中介绍用于实际执行匹配的 `String` 和 `RegExp` 方法时还将看到更多有关标志 `g` 的用法。

表 10-6: 正则表达式的标志

| 字符 | 含义 |
|----|---|
| i | 执行大小写不敏感的匹配 |
| g | 执行一个全局匹配。简而言之, 即找到所有匹配, 而不是在找到第一个之后就停止 |
| m | 多行模式, ^ 匹配一行的开头和字符串的开头, \$ 匹配一行的结尾或字符串的结尾 |

10.1.7 JavaScript 不支持的 Perl RegExp 特性

我们说过, ECMAScript v3 定义了 Perl 5 的正则表达式工具的相对完整的子集。ECMAScript 不支持的高级 Perl 特性如下:

- s (单行模式) 和 x (扩展语法) 标志
- \a、\e、\l、\u、\L、\U、\E、\Q、\A、\Z、\z 和 \G 转义序列
- (?<= 正后向锚和 (?<! 负后向锚
- (?# 注释和其他 (? 扩展语法

10.2 用于模式匹配的 String 方法

迄今为止, 虽然我们已经讨论过了用于创建正则表达式的语法, 但是我们还没有检验过这些正则表达式在 JavaScript 代码中如何使用。在这一节中, 我们将讨论 String 对象的部分方法, 它们在正则表达式中执行模式匹配和检索与替换操作。在此后的小节中, 我们将继续讨论使用 JavaScript 正则表达式的模式匹配, 不过讨论的是 RegExp 对象和它的方法及属性。注意, 接下来的讨论只是与正则表达式相关的各种方法和属性的概述。和以往一样, 可以在本书的核心参考手册中找到完整的介绍。

类 String 支持四种利用正则表达式的方法。最简单的是 `search()`。该方法以正则表达式为参数, 返回第一个与之匹配的子串的开始字符的位置, 如果没有任何匹配的子串, 它将返回 -1。例如, 下面的调用返回的值为 4:

```
"JavaScript".search(/script/i);
```

如果 `search()` 的参数不是正则表达式，它首先将被传递给 `RegExp` 构造函数，转换成正则表达式。`search()` 不支持全局检索，因为它忽略了正则表达式参数的标志 `g`。

方法 `replace()` 执行检索与替换操作。它的第一个参数是一个正则表达式，第二个参数是要进行替换的字符串。它将检索调用它的字符串，根据指定的模式来匹配。如果正则表达式中设置了标志 `g`，该方法将用替换字符串替换被检索的字符串中所有与模式匹配的子串。否则它只替换所发现的第一个与模式匹配的子串。如果 `replace()` 的第一个参数是字符串，而不是正则表达式，该方法将直接检索那个字符串，而不是像 `search()` 那样用 `RegExp()` 构造函数将它转换成一个正则表达式。例如，我们可以用如下方法使用 `replace()` 将文本字符串中的所有 `javascript`（不区分大小写）统一为“`JavaScript`”：

```
// 无论大小写形式如何，都用正确的大小写形式替换当前的字符串。  
text.replace(/javascript/gi, 'JavaScript');
```

但是 `replace()` 的功能远比上例所示强大。回忆一下可以知道，正则表达式中用括号括起来的子表达式是从左到右进行编号的，而且正则表达式会记住与每个子表达式匹配的文本。如果在替换字符串中出现了符号 `$` 加数字，那么 `replace()` 将用与指定的子表达式相匹配的文本来替换这两个字符。这是一个非常有用的特性。例如，我们可以用它将一个字符串中的直接引号替换为大引号，这与 ASCII 字符相似：

```
// 所谓引用就是引号加任意多个非引号字符，然后再加引号构成的。  
var quote = /"([^"]*)" /g;  
// 用大引号替换直接引号，并且保留引用的内容（保存在 $1 中）不变。  
text.replace(quote, "`" + $1 + "`");
```

`replace()` 还有其他的重要特性，这些特性将在核心参考手册部分的 `String.replace()` 参考页进行介绍。最值得注意的是，`replace()` 方法的第二个参数可以是函数，该函数能够动态地计算替换字符串。

方法 `match()` 是最常用的 `String` 方法。它唯一的参数就是一个正则表达式（或把它的参数传递给构造函数 `RegExp()` 以转换成正则表达式），返回的是包含了匹配结果的数组。如果该正则表达式有标志 `g`，该方法返回的数组包含的就是出现在字符串中的所有匹配。例如：

```
1 plus 2 equals 3".match(/(d+)=/g) // 返回["1", "2", "3"]
```

如果该正则表达式没有设置标志 `g`, `match()` 进行的就不是全局性检索, 它只是检索第一个匹配。但即使 `match()` 执行的不是全局检索, 它也返回一个数组。在这种情况下, 数组的第一个元素就是匹配的字符串, 而余下的元素则是正则表达式中用括号括起来的子表达式。因此, 如果 `match()` 返回了一个数组 `a`, 那么 `a[0]` 存放的是完整的匹配, `a[1]` 存放的则是与第一个用括号括起来的表达式匹配的子串, 以此类推。为了和方法 `replace()` 保持一致, `a[n]` 存放的是 `$n` 的内容。

例如, 使用如下的代码来解析一个 URL:

```
var url = /(w.):\:\/\/([w.])-\:\/\/(\3*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0]; // 存放 'http://www.isp.com/~david'
    var protocol = result[1]; // 存放 'http'
    var host = result[2]; // 存放 'www.isp.com'
    var path = result[3]; // 存放 '~david'
}
```

最后, `match()` 还有一个特性应该介绍一下。它返回的数组和其他的数组一样具有一个 `length` 属性。如果 `match()` 是作用于一个非全局的正则表达式, 那么它返回的数组还包括另外两个属性——`index` 和 `input`, 前者包含的是在字符串中匹配开始处的字符的位置, 后者则是目标字符串的一个副本。这样, 在上面的代码中, `result.index` 属性的值应该等于 21, 因为匹配了的 URL 是从文本中的第 21 个字符的位置开始。而属性 `result.input` 则应该具有和变量 `text` 相同的字符串。对于没有标志 `g` 的正则表达式 `r`, 调用 `s.match(r)` 返回的值与 `r.exec(s)` 相同。我们将在本章后面的小节中讨论 `RegExp.exec()` 方法。

`String` 对象的最后一个有关正则表达式的方法是 `split()`。这个方法可以把调用它的字符串分解为一个子串数组, 使用的分隔符是它的参数。例如:

```
"123,456,789".split(',') // 返回["123","456","789"]
```

`split()` 方法也可以以正则表达式为参数。这种能力使该方法更强大。例如, 我们可以指定分隔符, 允许两边有任意多个空白符:

```
"1,2, 3 , 4 ,5".split(/\s*,\s*/): // 返回["1","2","3","4","5"]
```

`split()` 方法还有其他的特性，核心参考手册的 `String.split()` 条目有完整的说明。

10.3 RegExp 对象

我们在本章的开始部分提到过，正则表达式是用 `RegExp` 对象来表示的。除了构造函数 `RegExp()` 之外，`RegExp` 对象还支持三种方法和大量的属性。`RegExp` 类的一个不寻常的特性是它既定义了类（静态）属性，又定义了实例属性。也就是说，它既定义了属于构造函数 `RegExp()` 的全局属性，又定义了其他属于单独的 `RegExp` 对象的属性。我们将在接下来的两节中介绍 `RegExp` 的模式匹配方法和属性。

构造函数 `RegExp()` 有一个或两个字符串参数，它将创建一个新的 `RegExp` 对象。该构造函数的第一个参数是包含正则表达式主体的字符串，即正则表达式直接量中出现在斜线对之间的文本。注意，无论是字符串直接量还是正则表达式都使用了字符 `\` 表示转义序列，所以当你将正则表达式作为字符串直接量传递给 `RegExp()` 时，必须用 `\\` 替换所有 `\` 字符。`RegExp()` 的第二个参数是可选的。如果提供了这个参数，它说明的就是该正则表达式的标志。它应该是 “g”、“i”、“m” 或它们的组合。例如：

```
// 找到字符串中的5个数字。注意本例中的\\。
var zipcode = new RegExp('\\.d{5}', "g");
```

当要动态创建一个正则表达式，而不能用正则表达式直接量的语法来表示时，构造函数 `RegExp()` 非常有用。例如，如果检索的字符串是由用户输入的，那么就必须在运行时用 `RegExp()` 构造函数来创建正则表达式。

10.3.1 用于模式匹配的 RegExp 方法

`RegExp` 对象定义了两个用于执行模式匹配操作的方法。它们的行为和前面介绍过的 `String` 方法很相似。主要的 `RegExp` 模式匹配方法是 `exec()`。它与前面介绍过的 `String` 方法 `match()` 相似，只不过它是以字符串为参数的 `RegExp` 方法，而不是以 `RegExp` 对象为参数的字符串。`exec()` 方法对一个指定的字符串执行一个正则表达式，简而言之，就是在一个字符串中检索匹配。如果没有找到任何匹配，它将返回 `null`，但是，如果它找到了一个匹配，将返回一个数组，就像方法 `match()` 为非全

局检索返回的数组一样。这个数组的元素0包含的是与正则表达式相匹配的字符串，余下的所有元素包含的是与括号括起来的子表达式相匹配的子串。而且，属性 `index` 包含了匹配发生的字符的位置，属性 `input` 引用的是被检索的字符串。

和 `match()` 方法不同的是，`exec()` 返回的数组类型相同，无论该正则表达式是否具有全局属性 `g`。回忆一下，当传递给方法 `match()` 的是一个全局正则表达式时，它返回的是一个匹配的数组。相比之下，方法 `exec()` 返回的总是一个匹配，而且提供关于该匹配的完整信息。当一个具有 `g` 标志的正则表达式调用 `exec()` 时，它将把该对象的 `lastIndex` 属性设置到紧接着匹配子串的字符位置。当同一个 `RegExp` 对象第二次调用 `exec()` 时，它将从 `lastIndex` 属性所指示的字符处开始检索。如果 `exec()` 没有发现任何匹配，它会将 `lastIndex` 属性重置为 0（在任何时候你都可以将 `lastIndex` 属性设为 0，每当你在一个字符串中找到最后一个匹配之前就开始用同一个 `RegExp` 对象来检索另一个字符串，而放弃了原来的检索的时候就应该这样做）。这一特殊的行为使得你可以反复调用 `exec()` 遍历一个字符串中所有的正则表达式匹配。例如：

```
var pattern = /Java/g;
var text = 'JavaScript is more fun than Java!';
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched '" + result[0] + "' +
        ' at position ' + result.index +
        ' ; next search begins at ' + pattern.lastIndex);
}
```

另一个 `RegExp` 方法是 `test()`，它比 `exec()` 方法简单一些。它的参数是一个字符串，如果这个字符串包含正则表达式的一个匹配，它就返回 `true`：

```
var pattern = /java/i;
pattern.test('JavaScript'); // 返回 true
```

调用 `test()` 方法等价于调用 `exec()` 方法，如果 `exec()` 的返回值不是 `null`，它将返回 `true`。由于这种等价性，当一个全局正则表达式调用方法 `test()` 时，它的行为和方法 `exec()` 相同，即它从 `lastIndex` 指定的位置处开始检索特定的字符串，如果它发现了匹配，就将 `lastIndex` 设置为紧接在那个匹配之后的字符的位置。这样一来，我们就可以使用方法 `test()` 来遍历字符串，就像用 `exec()` 方法那样。

`String` 方法 `search()`、`replace()` 和 `match()` 都没有像 `exec()` 和 `test()` 那样使

用属性 `lastIndex`。事实上，`String` 方法只是将 `lastIndex()` 重置为 0。如果使用一个全局模式的 `exec()` 方法或 `test()` 方法来检索多个字符串，那么就必须找到每个字符串中的所有匹配，以便 `lastIndex` 属性会被自动重置为 0（这在最后的检索失败时会发生），或者必须明确地将 `lastIndex` 属性设为 0。如果你忘记了这样做，那么再检索一个新字符串时，起始位置可能就是原来的字符串中的一个任意位置，而不一定是开头。最后要记住，只有带 `g` 标志的正则表达式才会发生这种特殊的 `lastIndex` 行为。如果 `RegExp` 对象没有标志 `g`，`exec()` 和 `test()` 将忽略它的 `lastIndex` 属性。

10.3.2 RegExp 的实例属性

每个 `RegExp` 对象都有五个属性。属性 `source` 是一个只读字符串，它存放的是正则表达式的文本。属性 `global` 是一个只读的布尔值，它说明了该正则表达式是否具有标志 `g`。属性 `ignoreCase` 也是一个只读的布尔值，它说明了该正则表达式是否具有标志 `i`。属性 `multiline` 是一个只读的布尔值，它说明了正则表达式是否具有标志 `m`。最后一个属性是 `lastIndex`，它是一个可读写的整数。对于具有标志 `g` 的模式，这个属性存储了在字符串中下一次开始检索的位置。它由方法 `exec()` 和 `test()` 使用，我们在上一节中已经介绍过了。

第十一章

JavaScript 的更多主题

本章介绍了其他的 JavaScript 主题。由于这些主题较难，所以如果在前面的章节中介绍了这些主题，就有可能使那些章节无法继续下去。现在你已经通读过前面几章了，而且体验了 JavaScript 语言的核心，为理解这里所阐述的更为高级和详尽的概念做好了准备。你也可以在阅读本章之前，先跳到其他的章节，学习有关客户端 JavaScript 的详细说明。

11.1 数据类型转换

我们知道，JavaScript 是一种无类型语言（或者更为准确地说，是一种松类型的、动态类型的语言）。这就是说，在声明一个变量时无须指定它的数据类型。JavaScript 是无类型的，所以它给予了 JavaScript 所需要的灵活性和简单性（尽管这些特性有损严密性，而严密性在用 C 或 Java 这样的强类型语言编写长而复杂的程序时很重要）。JavaScript 对数据类型的灵活处理方式的一个重要特性是自动类型转换。例如，如果传递给方法 `document.write()` 的是一个数字，JavaScript 会自动将它转换成与之等价的字符串表示。同样的，如果在一个 `if` 语句的条件中检测一个字符串值，JavaScript 会自动将那个字符串转换成一个布尔值，即如果这个字符串是空的，就转换成 `false`，否则就转换成 `true`。

基本的规则是，如果某个类型的值用于需要其他类型的值的环境中，JavaScript 就自动将这个值转换成所需要的类型。例如，如果一个数字用在布尔环境中，它就会

被转换成一个布尔值。如果一个对象用在字符串环境中，那么它就会被转换成一个字符串。如果一个字符串用在数字环境中，JavaScript会将它自动转换成数字。表11-1对这些转换进行了总结，它说明了将一个特定类型用于一种特定环境下时所执行的转换。该表之后的小节对JavaScript中的类型转换进行了更详细的说明。

表 11-1: 自动数据类型转换

| 值 | 字符串 | 使用值的环境 | | |
|--------|-------------|------------------------------------|-------|------------|
| | | 数字 | 布尔值 | 对象 |
| 未定义的值 | "undefined" | NaN | false | Error |
| null | "null" | 0 | false | Error |
| 非空字符串 | As is | 字符串的数字
值或 NaN | true | String 对象 |
| 空字符串 | As is | 0 | false | String 对象 |
| 0 | "0" | As is | false | Number 对象 |
| NaN | "NaN" | As is | false | Number 对象 |
| 无穷大 | "Infinity" | As is | true | Number 对象 |
| 负无穷大 | "-Infinity" | As is | true | Number 对象 |
| 其他所有数字 | 数字的字符串值 | As is | true | Number 对象 |
| True | "true" | 1 | As is | Boolean 对象 |
| false | "false" | 0 | As is | Boolean 对象 |
| 对象 | toString() | valueOf() 或
toString() 或
NaN | true | As is |

11.1.1 对象到基本数据类型的转换

表 11-1 说明了如何把 JavaScript 对象转换成原始值。不过有关该转换还需要一些额外的说明。首先注意，只要把非空对象用在布尔环境中，它就会被转换成 true。这适用于所有对象（包括所有数组和函数），即使是将被转换为 false 的表示原始值

的包装对象也不例外。例如，下列所有对象在用于布尔环境时都将被转换成 `true`（注1）：

```
new Boolean(false) // 内部值是 false，但对象将把它转换成 true
new Number(0)
new String("")
new Array()
```

表 11-1 说明，把对象转换成数字是通过首先调用该对象的 `valueOf()` 方法来完成。大多数对象继承了 `Object` 对象的默认 `valueOf()` 方法，它只是返回对象自身。由于默认的 `valueOf()` 方法不返回原始值，所以接下来 JavaScript 通过调用对象的 `toString()` 方法，再将生成的字符串转换成数字来把对象转换成数字。

对于数组，这将产生有趣的结果。回忆一下，数组的 `toString()` 方法将把数组元素逐一转换成字符串，然后把这些字符串连接起来，每个字符串之间用逗号分隔，返回连接后的字符串。因此，没有元素的数组将返回空串，如你在上表中所见，空串将被转换成数字 0。此外，如果数组只有一个为 n 的数字元素，该数组将转换成 n 的字符串表示，它将被转换回 n 自身。如果数组元素不止一个，或它的某个元素不是数字，该数组将被转换成 `NaN`（注2）。

表 11-1 还说明了对象用于字符串环境时如何被转换，以及用于数字环境时如何被转换。但是，在 JavaScript 中有两处环境不是很明确。运算符 “+” 和比较运算符（<、<=、> 和 >=）既能作用于数字，又能作用于字符串。所以当这两个运算符作用于对象时，就不太清楚应将该对象转换为数字还是字符串。在大多数情况下，JavaScript 会先尝试调用对象的 `valueOf()` 方法对它进行转换。如果该方法返回了原始值（通常是一个数字），就使用那个值。但是 `valueOf()` 方法通常返回的都是未被转换的对象，在这种情况下，JavaScript 将调用对象的 `toString()` 方法对它进行转换。

对于这种转换规则，只有一个例外，当运算符 “+” 作用于 `Date` 对象时，首先调用 `toString()` 方法进行转换。存在这一例外的原因是 `Date` 对象既有 `toString()` 方法，又有 `valueOf()` 方法。当 “+” 作用于一个 `Date` 对象时，你想执行的几乎都是

注 1： 注意，在 JavaScript 1.1 和 1.2 中，这些对象都被转换成 `false`，ECMAScript 遵守这一规则。

注 2： 但是，在 JavaScript 1.1 和 1.2 中，当数组用于数字环境中时，它将被转换为自己的长度。

字符串连接操作。但是当比较运算符作用于一个Date对象时，你想执行的则几乎都是数字比较，以判断两个时间中哪一个更早。

大多数对象没有valueOf()方法，或者没有能够返回有用的结果的valueOf()方法。当将“+”运算符作用于一个对象时，通常进行的是字符串的连接而不是加法运算。当将比较运算符作用于一个对象时，通常进行的则是字符串的比较，而不是数字比较。

一个具有定制的valueOf()方法的对象行为可能有所不同。如果定义了一个返回数字的valueOf()方法，就可以使算术运算符和其他运算符作用于对象，但是将对象添加到一个字符串中的行为就不像你所期望的那样了，方法toString()不再被调用，valueOf()返回的数字的字符串表示将被连接到字符串中。

最后要记住，valueOf()方法不能被称为toNumber()方法，严格地说，它的工作是把对象转换成合理的原始值，所以某些对象的valueOf()方法返回的可能是字符串。

11.1.2 显式类型转换

表11-1列出了JavaScript自动执行的数据类型转换。此外还可以显式地将值从一种类型转换为另一种类型。虽然JavaScript并不像C、C++和Java那样定义了类型转换运算符，但是它提供了一个相似的数值转换工具。

从JavaScript 1.1（和ECMA-262标准）起，Number()、Boolean()、String()和Object()都可以作为函数调用，就像作为构造函数被调用一样。采取这种方式调用时，这些函数将把它们的参数转换成合适的类型。例如，可以使用String(x)将任意值x转换成一个字符串，使用Object(y)将任意值y转换成一个对象。

有一些其他的技巧对执行显式类型转换非常有用。

要把一个值转换成字符串，可以把它连接到一个空串上：

```
var x_as_string = x + "";
```

要把一个值强制转换成数字，就用它减0：

```
var x_as_number = x - 0;
```

要把一个值强制转换成布尔值，需要连用两次“!”运算符：

```
var x_as_boolean = !!z;
```

由于JavaScript有自动将数据转换成所需要的类型的趋势，所以使用这些函数进行显式的转换通常是不必要的。但是有时它们还是比较有用的，而且它们还可以使你的代码更清晰、更简洁。

11.1.3 从数字到字符串的转换

从数字到字符串的转换在JavaScript中是最常执行的。虽然它通常都是自动发生的，但是有几种有用的方法可以用于显式地执行这一转换。我们在上面已经见过两种这类转换：

```
var string_value = String(number);    // 把String()构造函数用作函数  
var string_value = number + "";       // 与空串连接
```

把数字转换成字符串的另一种方法是用toString()方法：

```
string_value = number.toString();
```

Number对象的toString()方法（原始值被转换成Number对象以便调用这一方法）具有一个可选的参数，它说明了转换的基数。如果没有指定该参数，那么转换时采取的基数就是10。也可以以其他的基数（在2和36之间）进行转换（注3）。例如：

```
var n = 17;  
binary_string = n.toString(2);        // 值为"10001"  
octal_string = "0" + n.toString(8);   // 值为"021"  
hex_string = "0x" + n.toString(16);   // 值为"0x11"
```

JavaScript1.5之前的版本有一个缺陷，它没有一种内部的方式可以将数字转换成字符串，并指定要包含的小数位的位数或指定是否采用指数计数法。这使得显示具有传统格式的数字（如表示货币值的数字）有点困难。

注3： 注意，ECMAScript标准支持toString()方法的基数参数，不过，它允许方法返回一个基数不是10的已实现的实现字符串。因此，兼容的实现可以忽略该参数，并返回以10为基数的结果。事实上，Netscape和Microsoft都采用需要的基数。

ECMAScript v3 和 JavaScript 1.5 解决了这一问题，它们给 Number 类添加了三种将数字转换成字符串的方法。方法 toFixed() 将把数字转换成字符串，并显示小数点后指定的位数。它不使用指数计数法，方法 toExponential() 用指数计数法把数字转换成字符串，该字符串中的小数点前有一位数字，小数点后有指定位数的数字。方法 toPrecision() 用指定位数的有效数字显示数字。如果有效数字的位数不足以显示数字的整数部分，它将采用指数计数法显示数字。注意，这三种方法都会对结果字符串的数字进行适当的舍入。考虑如下的示例：

```
var n = 123456.789;  
n.toFixed(0);           // "123457"  
n.toFixed(2);           // "123456.79"  
n.toExponential(1);     // "1.2e+5"  
n.toExponential(3);     // "1.235e+5"  
n.toPrecision(4);       // "1.235e+5"  
n.toPrecision(7);       // "123456.8"
```

11.1.4 从字符串到数字的转换

我们已经知道，表示数字的字符串用于数字环境时会自动转换成真正的数字。如前面所示，我们可以显式地进行这一转换：

```
var number = Number(string_value);  
var number = string_value - 0;
```

这种字符串到数字的转换问题在于它过于严格。它只能作用于基数为10的数字，而且虽然它允许数字前后有空格，但是却不允许字符串中的数字之后出现任何非空格字符。

要更灵活地进行转换，可以使用函数 parseInt() 和 parseFloat()。这两个函数将转换并返回一个字符串开头的数字，并忽略其后的所有非数字后缀。parseInt() 只能解析整数，而 parseFloat() 则既能解析整数，又能解析浮点数。如果一个字符串以 0x 或 0X 开头，那么 parseInt() 就将它解释为十六进制数（注4）。例如：

注4：ECMAScript 规范规定，如果字符串以 0 开头（不是“0x”或“0X”），parseInt() 应将它解析为八进制数或十进制数。由于这一行为没有详细说明，所以应该避免用 parseInt() 来解析任何以 0 开头的数字，除非明确地指定要使用的基数。

```
parseInt("3 blind mice"); // 返回 3
parseFloat("3.14 meters"); // 返回 3.14
parseInt("12.34"); // 返回 12
parseInt("0xFF"); // 返回 255
```

`parseInt()` 还可以具有第二个参数，指定要被解析的数的基数。其合法值在 2 到 36 之间。例如：

```
parseInt("11", 2); // 返回 3 (1*2 + 1)
parseInt("ff", 16); // 返回 255 (15*16 + 15)
parseInt("zz", 36); // 返回 1295 (35*36 + 35)
parseInt("077", 8); // 返回 63 (7*8 + 7)
parseInt("077", 10); // 返回 77 (7*10 + 7)
```

如果 `parseInt()` 或 `parseFloat()` 不能将指定的字符串转换成数字，它们将返回 `NaN`：

```
parseInt('eleven'); // 返回 NaN
parseFloat('572.4 /'); // 返回 NaN
```

11.2 使用值和使用引用

和在其他程序设计语言中一样，在 JavaScript 中可以采用三种重要方式来操作一个数值。第一，可以复制它，例如将它赋给一个新的变量。第二，可以将其作为参数传递给一个函数或方法。第三，可以把它和其他的值进行比较来看看两者是否相等。要理解任何一种程序设计语言，就必须理解在那种语言中这三种操作是如何被执行的。

操作数值时，有两种根本不同的方式。这两种方法分别称为“使用值”和“使用引用”。当使用值来操作一个数值时，重要的是那个数据的值。在一个赋值语句中，会生成一个实际值的副本，这个副本存储在变量中、对象的属性中或者数组的元素中，副本和原始数据是两个分别存放的、完全独立的值。当使用值将一个数据传递给一个函数时，传递给该函数的是这个数据的一个副本。如果函数修改了这个值，改变只会影响到这个数据的副本，并不会影响到原始的数据。最后，当一个数据和另一个数据进行比较时使用的是值，那么这两个独立的数据段必须表示完全相同的值（通常意味着要进行逐字节的比较来判断它们是否相等）。

另一种操作数据的方式是使用引用。采用这种方法时，数值的实际副本只有一份，

操作的是对那个数值的引用（注5）。如果操作数据时使用的是引用，那么变量直接保存的并不是那个值，而是对该值的一个引用、复制的、传递的以及比较的都是这些引用。因此，在一个使用引用的赋值语句中，赋予的是对数值的引用，而不是值的一个副本，更不是值本身。进行了赋值之后，新的变量保存的也是对数值的引用，它与原始变量保存的引用相同。这两个引用具有同等效力，都可以用于数据操作，如果通过其中的一个引用改变了数值，这种改变也会在原始引用中体现出来。当使用引用将数据传递给函数时，情况也是一样。传递给函数的是数值的一个引用，函数可以使用这个引用来修改数值本身，任何修改在函数外部都是可见的。使用引用将一个数值和另一个数值进行比较时，比较的是两个引用，看它们是否引用的是同一个数值的惟一副本。对两个不同数值进行引用时，尽管这两个值偶尔也会相等（例如，由相同的字节构成），但是这两个引用却是不等的。

这两种操作数据的方式大相径庭，但是它们的含义非常重要，你应该理解。表 11-2 总结了这些含义。关于使用值和使用引用来操作数据的讨论是非常通用的，其区别适用于所有的程序设计语言。接下来的几节解释了这些区别如何应用于JavaScript，其中讨论了哪些数据类型使用值进行操作，哪些使用引用进行操作。

表 11-2: 使用值和使用引用

| | 使用值 | 使用引用 |
|----|---------------------------------|--|
| 复制 | 实际复制的是值，存在两个不同的、独立的副本 | 复制的只是对数值的引用。如果通过这个新的引用修改了数值，这个改变对原始的引用来说也可见 |
| 传递 | 传递给函数的是值的一个独立的副本，对它的改变在函数外部没有影响 | 传递给函数的是对数值的一个引用。如果函数通过传递给它的引用修改了数值，这个改变在函数外部也可见 |
| 比较 | 比较的是两个独立的值（通常逐字节比较），以判断它们是否相同 | 比较的是两个引用，以判断它们引用的是不是同一个数值。对两个不同的数值的引用不相等，即使这两个数值是由相同的字节构成的 |

注5: C程序员和熟悉指针概念的人应该理解在这种环境中使用的引用概念。但是要注意，JavaScript 不支持指针。

11.2.1 基本类型和引用类型

在 JavaScript 中基本的原则是这样的：基本类型使用值来操作。顾名思义，引用的类型、使用引用来操作。在 JavaScript 中，数字和布尔值是基本类型，说它们基本，是因为它们只是由小的、固定数量的字节构成的，这些字节是在 JavaScript 解释器的低层（基本层）进行操作的。另一方面，对象就是引用类型。作为特殊对象类型的数组和函数因此也是引用类型。由于这些数据类型可以包含任意多个属性或元素，所以它们不像固定大小的基本数值那样易于操作。因为对象和数组的值可能变得非常大，所以使用值来操作这些数据很不合理，这样做可能会产生大量低效率的内存复制和比较。

那么字符串又是什么类型的呢？由于字符串的长度是任意的，所以看起来它好像应该属于引用类型。但事实上在 JavaScript 中它们通常被当作基本类型，因为它们并不是对象。从二分法的观点来看，字符串实际上并不适合基本类型。稍后我们会对字符串和它的行为进行更多的说明。

说明使用值和使用引用操作数据之间的差别的最好方法还是用例子。仔细研究下面的例子，要格外注意注释。例 11-1 对数字进行了复制、传递和比较。由于数字是基本类型，所以这个例子说明的是使用值的数据操作。

例 11-1：使用值进行复制、传递和比较

```
// 首先我们举例说明使用值的复制操作。
var n = 1; // 变量 n 的值为 1。
var m = n; // 使用值的复制操作：变量 m 存放了一个独立的值 1。

// 我们用以下函数来说明使用值的传递操作。
// 如我们所见，函数并不是按照我们想要的方式运行的。
function add_to_total(total, x)
{
    total = total + x; // 这一行只改变了内部 total 的副本
}

// 现在调用该函数，传递给它使用值存放的 n 和 m 数值。
// n 的值将被复制，复制后的值在函数中被命名为 total。
// 该函数将把 m 的一个副本加到 n 的那个副本中。
// 不过给 n 的副本加值不会影响到函数外部的 n 的原始值。
// 所以调用这个函数不会产生任何效果。
add_to_total(n, m);

// 下面我们来看看使用值的比较运算。
// 在下面的代码中，直接量 1 显然是编码到程序中的一个独立数值。
```

```
// 我们用它和变量 n 存放的值进行比较。
// 在比较值的运算中，将检查两个数字的字节，看它们是否相同。
if (n == 1) n = 2; // n 的值与直接量 1 相同，n 的值现在是 2。
```

现在，研究一下例 11-2。这个例子复制、传递和比较的是一个对象。由于对象是引用类型，所以这些操作是采用引用执行的。这个例子使用的是 Date 对象，如果有必要，你可以在本书的核心参考手册部分查找到这一对象的详细介绍。

例 11-2：使用引用进行复制、传递和比较

```
// 下面我们创建一个对象表示 2001 年的圣诞节
// 变量 xmas 存放的是对对象的引用，而不是对象本身。
var xmas = new Date(2001, 11, 25);

// 当我们对引用进行复制操作时，将得到对原始对象的一个新引用。
var solstice = xmas; // 现在两个变量引用的是同一个对象值。

// 下面我们通过新的引用改变对象
solstice.setDate(21);

// 通过原始引用也可以看到对象的变化。
xmas.getDate(); // 返回 21，而不是原始值 25。

// 当把对象或数组传递给函数时，情况也是这样。
// 接下来的函数将给数组的每个元素都加上一个值
// 传递给函数的是对数组的引用，而不是数组的副本。
// 因此，函数可以通过引用改变数组的内容，当函数返回时，可以看到这些改变。
function add_to_totals(totals, x)
{
    totals[0] = totals[0] + x;
    totals[1] = totals[1] + x;
    totals[2] = totals[2] + x;
}

// 最后我们用引用进行比较操作。
// 当我们比较上面定义的两个变量时，发现它们相等。
// 因为它们引用的是同一个对象，即使它们引用的日期不同。
(xmas == solstice) // 值为 true。

// 下面定义的两个变量引用的是两个不同的对象。
// 这两个对象表示的日期完全相同。
var xmas = new Date(2001, 11, 25);
var solstice_plus_4 = new Date(2001, 11, 25);

// 不过根据“引用比较”的原则，不同的对象不相等。
(xmas != solstice_plus_4) // 值为 true。
```

在结束使用引用来操作对象和数组的主题之前，有一点需要明确。短语“使用引用传递”有几种含义。对某些读者来说，这个短语指的是一种函数调用方法，这一方

法使得函数可以将新值赋予它的参数，而且这些修改在函数外部也可见。但这并不是本书中使用的这个术语的含义。在这里，我们的意思只是传递给函数的是对一个对象或数组引用，而不是这个对象本身。一个函数可以使用引用来修改对象的属性或数组的元素。但是如果函数用一个对新对象或数组的引用覆盖了原来的引用，那么在函数中进行的修改在函数外部就不可见了。熟悉这个术语的其他含义的读者可能会说对象和数组是用值来传递的，但是这个被传递的值实际上是一个引用，而不是对象本身。例11-3说明了这一观点。

例11-3：引用自身是使用值来传递的

```
// 这是 add_to_totals() 函数的另一个版本，但是它没有什么效用。  
// 因为它不是改变数组自身，而是尝试改变对数组的引用。  
function add_to_totals2(totals, x)  
{  
    newTotals = new Array(3);  
    newTotals[0] = totals[0] + x;  
    newTotals[1] = totals[1] + x;  
    newTotals[2] = totals[2] + x;  
    totals = newTotals;    // 这行代码在函数外部没有任何影响。  
}
```

11.2.2 字符串的复制和传递

前面提到过，从二分法的观点来看，相对于引用类型，JavaScript 的字符串并不是非常适合基本类型。由于字符串不是对象，所以自然假定它们为基本类型。如果它们是基本类型，那么根据上面给出的原则，应该使用值来操作它们。但是又因为字符串的长度是任意的，所以逐字节地来复制、传递和比较字符串的效率很低。因此，假定字符串是作为引用类型被操作的也很自然。

我们可以假定编写了一段 JavaScript 代码对字符串的操作进行实验，这样就不必对字符串进行猜测了。如果字符串是使用引用来复制和传递的，那么就应该通过存储在另一个变量中的引用或者是传递过来的引用修改字符串的内容。

但是，当我们开始编写代码来执行这一实验时，就遇到一个重要的错误，即无法修改字符串的内容。虽然方法 `charAt()` 返回的是字符串中位于指定位置的字符，但是却并没有相应的 `setCharAt()` 方法。这并不是一个疏忽之处。JavaScript 的字符串是有意地被设为不可变的，也就是说，没有一种 JavaScript 语法、方法或属性可以改变字符串中的字符。

由于字符串是不可变的，所以我们的原始问题就显得毫无实际意义了，根本就没有一种方法可以辨别字符串是使用值进行传递的还是用引用进行传递的。为了效率起见，我们假定 JavaScript 的实现使用引用来传递字符串，但实际上这并不重要，因为这与我们编写的代码没有一点实际关系。

11.2.3 字符串的比较

尽管我们不能确定复制或传递字符串时使用的是值还是引用，但是我们能够编写代码来判断字符串比较时使用的是值还是引用。例 11-4 就是我们用来进行这一判断的代码。

例 11-4：字符串比较时使用的是值还是引用

```
// 判断比较时使用的是值还是引用很容易。  
// 我们比较两个明显不同的字符串，它们恰好含有相同的字符。  
// 如果用值来比较，它们就相等。  
// 但如果用引用来比较，它们就不相等。  
var s1 = "be?ic";  
var s2 = "be?ic" + 0;  
if (s1 == s2) document.write("Strings compared by value");
```

这个实验说明比较字符串时使用的是值。这也许会使某些程序员感到吃惊。在 C、C++ 和 Java 中，字符串属于引用类型，比较时使用的是对它们的引用。如果你想比较两个字符串的实际内容，必须使用特殊的方法或函数。但是 JavaScript 是一种更高级的语言，它认为你在比较字符串时最想使用的它们的值。因此，尽管 JavaScript 为了效率起见，在复制和传递字符串时使用的是引用，但是在比较它们时使用的却是值。

11.2.4 使用值和使用引用：总结

表 11-3 总结了操作不同的 JavaScript 类型时所采取的方式。

表 11-3：JavaScript 中的数据类型操作

| 类型 | 复制所使用的 | 传递所使用的 | 比较所使用的 |
|-----|--------|--------|--------|
| 数字 | 值 | 值 | 值 |
| 布尔值 | 值 | 值 | 值 |
| 字符串 | 不可变的 | 不可变的 | 值 |
| 对象 | 引用 | 引用 | 引用 |

11.3 无用存储单元收集

在第四章中我们解释过，JavaScript 使用无用存储单元收集的方法来回收那些由字符串、对象、数组或函数占用的而且不再使用的内存。这就使程序员不必再自己显式地释放那些内存了，这也是 JavaScript 的程序设计易于 C 的程序设计的一个重要原因。

无用存储单元收集的一个关键特性是无用存储单元收集器必须要能够确定安全回收内存的时机。显而易见，它决不能回收那些仍在使用的值，而应该收集再也不会使用的值，也就是那些不会再由程序中的任何变量、对象的属性或数组的元素引用的值。如果你是一个喜欢打破沙锅问到底的人，那么一定想知道无用存储单元收集器如何区别要收集的无用单元和仍在使用的值或者有可能被使用的值。下面的几节将解释某些细节。

11.3.1 标记和清除的无用存储单元

有关无用存储单元收集的计算机著作非常多，而且技术性很强。无用存储单元收集器的实际操作则是一个特殊的实现细节，它根据语言的不同实现而有所变化。但是几乎所有重要的无用存储单元收集器使用的都是基本的无用存储单元收集算法的某种变形，这种算法称为“标记和清除算法”。

一个标记和清除的无用存储单元回收器会周期性地遍历 JavaScript 环境中的所有变量的列表，并且给这些变量所引用的值做标记。如果被引用的值是对象或数组，那么对象的属性或者数组的元素就会被递归地做上标记。通过递归地遍历所有值的树或者图，无用存储单元收集器就能够找到（并标记）仍旧使用的每个值。那些没有标记的值就是无用的存储单元。

当采用标记和清除算法的无用单元收集器给所有正在使用的变量做完了标记之后，它就会开始进行清除。在这个阶段中，它将遍历环境中所有值的列表，同时释放那些没有标记的值。经典的标记和清除无用存储单元收集器每次都进行一次完整的标记和一次完整的清除工作，这在使用无用存储单元收集过程的系统中会大大降低系统的速度。该算法较为复杂的变形使效率相对提高，它们在后台执行收集，并不影响系统的性能。

无用存储单元收集的细节是由实现指定的, 编写 JavaScript 程序无需知道它的所有细节。现代的 JavaScript 实现都使用某种类型的无用存储单元收集技术。不过 JavaScript 1.1 (在 Netscape 3 中实现的) 使用的是较为简单的无用存储单元收集模式, 所以具有一些缺陷。如果要编写与 Netscape 3 兼容的 JavaScript 代码, 接下来的一节解释了在那种浏览器中无用存储单元收集器的缺陷。Netscape 2 使用的是更简单的无用存储单元收集技术, 有严重的错误。由于那种浏览器现在已经被废弃了, 所以这里不再介绍其细节。

11.3.2 采用引用计数的无用存储单元收集

在 JavaScript 1.1 (在 Netscape 3 中实现的) 中, 无用存储单元的收集是通过引用计数执行的。这就是说, 每个对象 (无论是由 JavaScript 代码创建的用户对象还是一个由浏览器创建的内部 HTML 对象) 都记录了对它引用的次数。回忆一下, 我们讲过, 在 JavaScript 中对象赋值时使用的是引用, 而不是复制它们的完整值。

当一个对象被创建, 而且它的一个引用被存储在变量中, 引用计数就为 1。当这个对象的引用被复制, 并且存储在另一个变量中时, 引用计数就增加到 2。当保存这些引用的其中一个变量被某个新值覆盖了时, 该对象的引用计数就减为 1。如果引用计数达到了 0, 那么就没有对这个对象的引用了。由于没有了对副本的引用, 所以在程序中也就不再有对这个对象的引用。因此, JavaScript 知道此时销毁对象并且收集与之关联的内存是安全的。

遗憾的是, 使用引用计数作为无用存储单元收集的方案存在一些缺陷。事实上, 一些人甚至认为引用计数并不是真正的无用存储单元收集策略, 而将这一术语留给更好的算法, 诸如标记和清除算法。引用计数是无用存储单元收集的一种非常简单的形式, 它易于实现, 而且在很多情况下运行的都很好。但是, 有一种非常重要的情况需要格外注意, 那就是引用计数不能正确地探测并收集所有无用存储单元。

引用计数的基本错误是必须处理循环引用 (cyclical reference)。如果一个对象 A 包含对对象 B 的引用, 而对象 B 又包含对对象 A 的引用, 就形成了一个循环引用。如果 A 引用了 B, B 引用了 C, 而 C 又引用了 A, 同样形成了一个循环引用。在这些循环中, 每个元素总存在一个引用。因此即使循环中的元素不再具有任何来自循环外部的引用, 它们的引用计数也决不会小于 1, 从而永远不会成为无用存储单元。如果程序中没有其他关于这些对象的引用, 那么整个循环都是无用的存储单元, 但是

由于它们彼此进行了引用, 所以采用引用计数的无用存储单元收集器就无法探测并且释放这些不再使用的内存。

一个简单的无用存储单元收集方案必须为这一循环所产生的问题付出代价。解决这一问题的惟一方法是进行手动干涉。如果你创建了一个引用循环, 就必须认识到这一事实, 并且采取一定措施来确保当那些对象不再使用时能收集它们的空间。要使一个引用循环的对象占用的空间能够被收集, 就必须打破这个循环。你可以从这些循环的对象中选出一个, 然后将它引用下一个对象的属性设置为 `null` 即可。例如, 假定对象 A、B 和 C 都具有 `next` 属性, 而且该属性的值都被设置了, 以便这些对象能够彼此引用, 因而形成一个引用循环。当不再使用这些对象时, 将 `A.next` 设置为 `null` 就能够打破这个循环。这就是说, 对象 B 就不再具有一个来自 A 的引用了, 所以它的引用计数就降为 0, 也就可以被收集了。当 B 被收集了之后, 它就不再引用对象 C, 从而 C 的引用计数也降为 0, C 同样可以被收集了。当 C 被收集了之后, A 就能够被收集了。

当然要注意, 如果 A、B 和 C 存储在一个仍旧打开的窗口的全局变量中, 那么上面的过程就不会发生, 因为变量 A、B、C 仍旧引用了这些对象。如果它们存储在一个函数的局部变量中, 而且在函数返回之前打破了它们之间的循环, 它们就能够被收集。但是如果它们存储在全局变量中, 那么在包含它们的窗口关闭之前, 对它们的引用会一直存在。在这种情况下, 如果你想迫使它们能够被收集, 就必须打破循环, 并且将所有的变量都设置为 `null`:

```
A.next = null; // 打破循环
A = B = C = null; // 删除残余的外部引用
```

11.4 词法作用域和嵌套函数

JavaScript 的函数是词法上的作用域, 而不是动态作用域。这意味着它们运行在自己定义的作用域中, 而不是运行在执行它们的作用域中。在 JavaScript 1.2 之前的版本中, 只能在全局作用域中定义函数, 词法作用域不会产生太大的问题, 因为所有函数都在同一个全局作用域中执行 (函数的调用对象链接在全局作用域上)。

但在 JavaScript 1.2 和其后的版本中, 可以随时定义函数, 作用域的复杂问题出现了。例如, 考虑在函数 `f` 内定义的函数 `g`, `g` 总在 `f` 的作用域中执行。它的作用域链

包括三个对象：它自己的调用对象、`f()`的调用对象和全局对象。当在定义函数的同一个词法作用域中调用它们时，嵌套函数极其容易理解。例如，下面的代码说明了这一问题：

```
var x = "global";
function f() {
    var y = "local";
    function g() { alert(x); }
    g();
}
f(); // 调用该函数，显示 'local'。
```

但在JavaScript中，函数和其他值一样是数据，所以它们可以由函数返回，被赋予对象的属性，存储在数组中，等等。如果不涉及嵌套函数，这也不会引起什么大问题。考虑下面的代码，它包括一个返回嵌套函数的函数。每次调用函数，它都返回一个函数。返回的函数的JavaScript代码都一样，只是每次调用时创建这些函数的作用域有点不同，因为每次调用传给外部函数的参数不同。如果把这些返回的函数保存在数组中，依次调用每个函数，可以看到每个函数返回的值不同。由于构成每个函数的JavaScript代码完全相同，而且都是从同一个作用域中调用它们，所以使返回值不同的惟一因素只有定义函数的作用域不同：

```
// 每次调用该函数，它都返回一个函数。
// 每次调用该函数，定义它的作用域都不同。
function makeFunc(x) {
    return function() { return x; }
}

// 调用几次makeFunc()函数，把结果保存到一个数组中。
var a = [makeFunc(0), makeFunc(1), makeFunc(2)];

// 下面调用这些函数并显示它们的情。
// 尽管每个函数的函数体相同，但它们的作用域不同。
// 每次调用返回的值都不同。
alert(a[0]()); // 显示0
alert(a[1]()); // 显示1
alert(a[2]()); // 显示2
```

这个代码段的结果可能令你感到吃惊。不过它们仍然是从词法作用域规则（即函数在定义它的作用域中执行）的严格应用而得出的结果。这个作用域包括局部变量和参数的状态。即使局部变量和函数都是暂存的，它们的状态是冻结的，但它们有效

时都是定义的函数的词法作用域的一部分。为了使词法作用域与嵌套函数协同使用，JavaScript 实现还使用了“闭包”，可以将它看作定义函数时有效的作用域链和函数定义的组合。

11.5 Function()构造函数和函数直接量

我们在第七章中见到过，除了使用基本的 `function` 语句之外，还有两种方式可以用来定义函数。在 JavaScript 1.1 中，可以使用构造函数 `Function()` 来定义函数。在 JavaScript 1.2 和其后的版本中，还可以使用函数直接量来构造函数。你应该注意这两种方法之间的重要差别。

首先，构造函数 `Function()` 允许在运行时动态地创建和编译 JavaScript 代码。但是函数直接量却是程序结构的一个静态部分，就像 `function` 语句一样。

其次，作为第一个差别的必然结果，每次调用构造函数 `Function()` 时都会解析函数体并且创建一个新的函数对象。如果对构造函数的调用出现在一个循环中，或者出现在一个经常被调用的函数中，这种方法的效率就非常低。另一个方面，函数直接量或出现在循环和函数中的嵌套函数不是在每次调用时都被重新编译，而且每当遇到一个函数直接量时也不创建一个新的函数对象（但如前面提到的，可能需要一个新闭包来捕捉定义函数的词法作用域的差别）。

`Function()` 构造函数和函数直接量之间的第三点差别是，使用构造函数 `Function()` 创建的函数不使用词法作用域，相反的，它们总是被当作顶级函数来编译，就像下面代码所说明的那样：

```
var y = "global";
function constructFunction() {
    var y = "local";
    return new Function("return y"); // 不捕捉局部作用域。
}
// 这行代码将显示 "global"，因为 Function() 构造函数返回的函数并不使用局部作用域
// 假如使用一个函数直接量，这行代码则可能显示 "local"。
alert(constructFunction()); // 显示 "global"。
```

11.6 Netscape 公司的 JavaScript 1.2 的不兼容性

ECMAScript v1 标准还未完成时，发布了 JavaScript 1.2 的 Netscape 实现（作为 Netscape 4.0 浏览器的一部分）。Netscape 公司的工程师对 ECMAScript 标准中的内容进行了猜想，并基于这些猜想，对 JavaScript 的行为进行了一些改变。由于这些改变与 JavaScript 以前的版本不兼容，所以这些改变只在明确地请求使用 JavaScript 1.2 时才实现（在 Web 浏览器中，是通过把 HTML 标记 `<script>` 的 `language` 性质设置成“JavaScript 1.2”来实现的）。这是引入新行为，又不破坏旧脚本的好方法。遗憾的是，当 ECMAScript V1 完成时，Netscape 公司的工程师猜测的新行为不是新标准的一部分。这意味着 Netscape 公司的 JavaScript 1.2 实现具有专有行为，与 JavaScript 1.1 不兼容，而且与 ECMAScript 标准不一致。

为了与依赖 JavaScript 1.2 的不一致行为的脚本兼容，Netscape 公司以后的 JavaScript 实现，在明确要求使用 1.2 版本时都保留了专有行为。但要注意，如果你要求使用的版本比 1.2 高（例如，把 `language` 性质设置成了“JavaScript 1.3”），得到的行为与 ECMAScript 标准一致。因为这种专有行为只出现在 Netscape 公司的 JavaScript 实现中，所以在你的脚本中不应该信赖它，最好的方法是不要明确地指定使用版本 1.2。然而，当你必须使用 JavaScript 1.2 时，可以参考下面列出的那个版本的专有行为：

- 相等运算符和不等运算符与等同运算符和不等同运算符的行为一样，也就是说，`==` 与 `===` 作用一样，`!=` 与 `!==` 作用一样。
- 默认的 `Object.toString()` 方法显示该对象定义的所有属性值，返回一个用对象直接量的语法格式化的字符串。
- `Array.toString()` 方法用逗号和空格（而不只是逗号）分割数组元素，返回带方括号的元素列表。此外，数组的字符串元素用引号括起来，以便使结果是合法的数组直接量字符串。
- 把单个数字参数 `n` 传递给构造函数 `Array()` 时，它返回一个数组，`n` 是它的唯一元素，而不是数组的长度。

- 在数字环境中使用数组对象时，它的值为它的长度。在布尔环境中使用数组对象时，如果它的长度为 0，它的值就为 `false`，否则值为 `true`。
- `Array.push()` 方法返回最后被推入的值，而不是新的数组长度。
- 在 `Array.splice()` 方法切分单个元素 `x` 时，它返回 `x` 自身，而不是返回把 `x` 作为惟一元素的数组。在 `splice()` 没有从数组中删除任何元素时，它什么都不返回，而不是返回空数组。
- 在调用 `String.substring()` 方法时，如果传递给它的开始位置比结束位置大，它将返回空串，而不是交换两个参数，返回它们之间的子串。
- `String.split()` 方法显示的是从 Perl 继承来的特殊行为，即如果指定的分隔符是一个空格，就在分割字符串的其余部分前舍弃该字符串开头和结尾的所有空白符。

第二部分

客户端 JavaScript

第二部分包括第十二章到第二十二章的内容，描述了 Web 浏览器中实现的 JavaScript。在这些章节中引入了大量新的 JavaScript 对象，这些对象用于表示 Web 浏览器和 HTML 文档的内容。其中许多例子展示的都是这些客户端对象的典型用法。仔细研究这些例子是非常有用的。

- 第十二章，Web 浏览器中的 JavaScript
- 第十三章，窗口和框架
- 第十四章，Document 对象
- 第十五章，表单和表单元素
- 第十六章，脚本 cookie
- 第十七章，文档对象模型
- 第十八章，级联样式表和动态 HTML
- 第十九章，事件和事件处理
- 第二十章，兼容性
- 第二十一章，JavaScript 的安全性
- 第二十二章，在 JavaScript 中使用 Java

第十二章

Web 浏览器中的 JavaScript

本书第一部分描述了 JavaScript 语言的核心。现在我们要介绍在 Web 浏览器中使用的 JavaScript，通常我们称它为客户端 JavaScript（注 1）。迄今为止，我们所看到的大部分例子虽然是合法的 JavaScript 代码，但是却没有特定的环境，也就是说它们不过是一些运行在没有说明的环境中的 JavaScript 片段。本章给它们提供了这个环境。首先，我们将对 Web 浏览器的程序设计环境和基本的客户端 JavaScript 代码的概念进行了一般性介绍。然后我们将讨论怎样才能将 JavaScript 代码嵌入 HTML 文档以便让它在 Web 浏览器中运行。最后，本章详细介绍了在一个 Web 浏览器中如何执行 JavaScript 程序。

12.1 Web 浏览器环境

要理解客户端 JavaScript，必须理解 Web 浏览器所提供的程序设计环境的概念性框架。接下来的几节介绍的是程序设计环境的三个重要特性：

注 1：术语“客户端 JavaScript”是从 JavaScript 仅用于 Web 浏览器（客户端）和 Web 服务器时流传下来的。由于采用 JavaScript 作为脚本语言的环境越来越多，“客户端”这个术语也就失去了意义，因为它没有指明是什么样的客户端。不过，在本书中我们仍然使用这个术语。

- 作为全局对象的 Window 对象和客户端 JavaScript 代码的全局执行环境
- 客户端对象的层次和构成它的一部分的文档对象模型
- 事件驱动的程序设计模型

12.1.1 作为全局执行环境的 Window 对象

Web 浏览器的主要任务是在一个窗口中显示 HTML 文档。在客户端 JavaScript 中，表示 HTML 文档的是 Document 对象，Window 对象代表显示该文档的窗口（或框架）。虽然对于客户端 JavaScript 来说，Document 对象和 Window 对象都很重要，但是相比较而言，Window 对象更重要一些。本质上的原因是 Window 对象是客户端程序设计中的全局对象。

回忆一下，我们在第四章中介绍过，JavaScript 的每一个实现都有一个全局对象，该对象位于作用域链的头部。这个全局对象的属性也就是全局变量。客户端 JavaScript 的 Window 对象是全局对象，它定义了大量的属性和方法，使用户可以对 Web 浏览器的窗口进行操作。它还定义了引用其他重要对象的属性，如引用 Document 对象的 document 属性。此外 Window 对象还包括两个自我引用的属性：window 和 self。可以使用这两个全局变量来直接引用 Window 对象。

由于在客户端 JavaScript 中 Window 对象是全局对象，因此所有的全局变量都被定义为该对象的属性。例如，下面的两行代码实际上执行的是相同的功能：

```
var answer = 42;      // 声明并初始化一个全局变量
window.answer = 42;  // 创建 Window 对象的一个新属性
```

Window 对象代表的是一个 Web 浏览器窗口或者窗口中的一个框架。在客户端 JavaScript 中，顶层窗口和框架本质上是等价的。编写使用多框架的 JavaScript 应用程序很常见，而且编写使用多窗口的 JavaScript 应用程序也是可能的，尽管不那么常见。一个应用程序中出现的每个窗口和框架都对应一个 Window 对象，而且都为客户端 JavaScript 代码定义了一个唯一的执行环境。换句话说，JavaScript 代码在一个框架中声明的全局变量并不是另一个框架的全局变量，但是第二个框架却可以存取第一个框架的全局变量。我们将在第十三章中看到处理这一问题的详细说明。

12.1.2 客户端的对象层次和文档对象模型

我们知道, Window 对象是客户端 JavaScript 中的一个关键对象。其他所有的客户端对象都和这个对象连接在一起。例如, 每个 Window 对象都包含一个 document 属性, 该属性引用与这个窗口关联在一起的 Document 对象, location 属性引用与该窗口关联在一起的 Location 对象。此外 Window 对象还包含一个 frames[] 数组, 它引用代表原始窗口的框架的 Window 对象。因此, document 代表的是当前窗口的 Document 对象, 而 frames[1].document 引用的是当前窗口的第二个子框架的 Document 对象。

由当前窗口或其他 Window 对象引用的对象本身还可能引用其他的对象。例如, 每个 Document 对象都有一个 forms[] 数组, 它包含的是代表该文档中出现的所有 HTML 表单的 Form 对象。要引用这些表单, 可以编写如下的代码:

```
window.document.forms[0]
```

继续使用上面的例子, 每个 Form 对象都有一个 elements[] 数组, 该数组包含了出现在表单中的各种 HTML 表单元素 (如输入域、按钮等) 的对象。在极其特殊的情况下, 可以编写引用整个对象链底部的对象的代码, 其表达式的复杂度如下:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

我们已经知道, Window 对象是位于作用域链头部的全局对象, JavaScript 中的所有客户端对象都是作为其他对象的属性来存取的。这就是说, 存在一个 JavaScript 对象的层次, 这个层次的根是一个 Window 对象。图 12-1 说明了这一层次, 仔细研究这幅图, 理解其中的层次以及它所包含的对象, 对成功地设计客户端 JavaScript 的程序至关重要。本书余下的章节都用于描述图中所示的对象的细节。

注意, 图 12-1 仅仅显示出了那些引用其他对象的属性。图中所示的大部分对象具有的属性都比显示出来的要多。

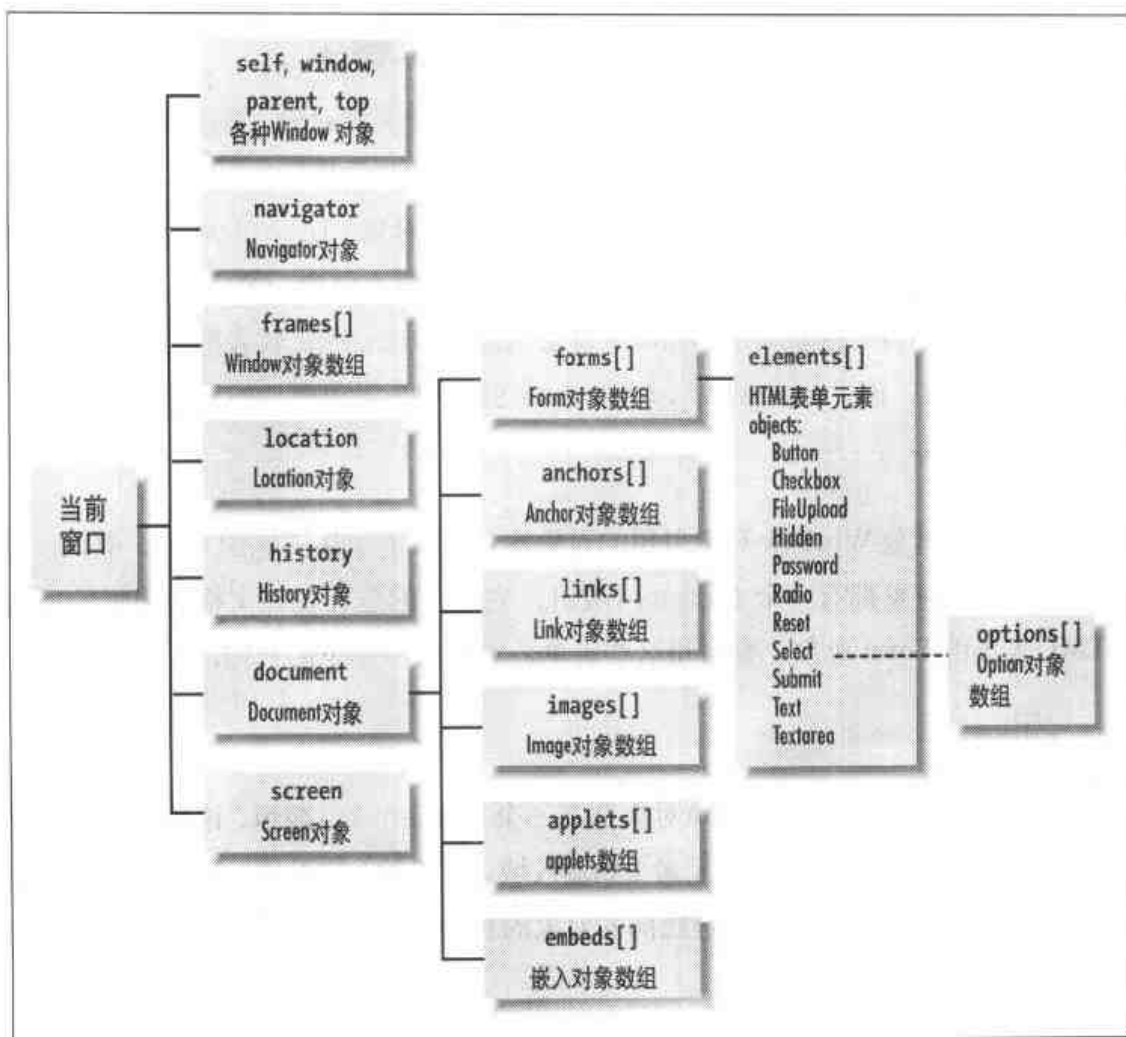


图 12-1: 客户端的对象层次和 0 级 DOM

图 12-1 中显示的许多对象都继承了 Document 对象。大型的客户端对象层次的子树被称作文档对象模型 (DOM)，它很有趣，因为它已经成为标准化进程的焦点。图 12-1 显示的 Document 对象已经成为实际标准，因为所有主流浏览器都统一实现了它。它们统称为 0 级 DOM，因为它们构成了文档功能的基本级别，JavaScript 的程序设计者在所有浏览器中都可以应用该级别。这些基本的 Document 对象是第十四章和第十五章的主题。W3C 标准化的更高级的文档对象模型是第十七章和第十八章的主题。

12.1.3 事件驱动的程序设计模型

过去，计算机程序常常以批处理的模式运行。也就是说，它们先读进来一批数据，

然后对这批数据进行计算，最后输出计算的结果。随着时间片共享和基于文本的终端的出现，便开始进行有限的交互，程序要求用户输入，用户输入数据，然后计算机对数据进行处理并且在屏幕上显示出结果。

现在，出现了图形显示和像鼠标这样的点击设备，情况就又不同了。程序通常都是事件驱动的，用户以鼠标点击和键盘敲击的方式进行输入，程序则根据鼠标指针的位置对这种异步的用户输入进行响应。Web 浏览器恰恰就是这样一个图形环境。由于一个 HTML 文档包含嵌入式 GUI（图形用户接口），因此客户端 JavaScript 使用的就是这种事件驱动的程序设计模型。

编写一个不接受用户输入、每次都完成相同工作的静态 JavaScript 程序也是完全可能的。有时这种程序非常有用，但是，大多数情况下我们需要编写能够和用户交互的动态程序。要做到这一点，必须能够响应用户输入。

在客户端 JavaScript 中，Web 浏览器使用事件（event）来通知程序有用户输入。事件的类型有很多种，例如按键事件、鼠标移动事件等等。当一个事件发生时，Web 浏览器会先尝试调用一个适合的事件处理程序来响应那个事件。因此，要编写一个动态的、交互性的客户端 JavaScript 程序，必须先定义一些适当的事件处理程序，并将它们注册到系统中，这样浏览器才能在适当的时刻调用它们。

如果你还不熟悉事件驱动的程序设计模型，那么熟练使用这种模型还需要花费一番工夫。在旧的模型中，你可以编写一个大的代码块，把它放在一些定义明确的控制流之后，并且从头到尾完整地运行一遍即可。但事件驱动的程序设计模型则有自己的模式。在事件驱动的程序设计中，可以编写大量独立的（但不是交互的）事件处理程序。你并不需要直接调用这些处理函数，而是让系统在适当的时机调用它们。由于它们是由用户输入触发的，因此事件处理程序应该在不可预知的、异步的时刻被调用。在大部分时间中，程序根本就不运行，只是等待系统调用它的某一个事件处理程序。

下面一节解释了 JavaScript 代码是如何嵌入到 HTML 文件中的。它说明了如何才能既定义从头到尾同步运行的静态代码块，又定义由系统异步调用的事件处理程序。我们将在第十九章详细讨论事件和事件处理程序。

12.2 在 HTML 中嵌入 JavaScript

把客户端 JavaScript 代码嵌入 HTML 文档有很多方法：

- 放置在标记对 `<script>` 和 `</script>` 之间
- 放置在由 `<script>` 标记的 `src` 属性指定的外部文件中
- 放置在事件处理程序中，该事件处理程序由 `onclick` 或 `onmouseover` 这样的 HTML 属性值指定
- 作为 URL 的主体，这个 URL 使用特殊的 `javascript:` 协议

下面几节详细地说明了这些 JavaScript 的嵌入技术，此外，还解释了将 JavaScript 添加到网页中的所有方法，也就是说，将解释客户端许可的 JavaScript 程序结构。

12.2.1 `<script>` 标记

客户端 JavaScript 脚本是 HTML 文件的一部分，通常放置在标记 `<script>` 和 `</script>` 之间。可以在这两个标记之间放置任意多条 JavaScript 语句。放置这些语句的顺序就是它们的执行顺序，而且执行的过程是文档装载过程的一部分。标记 `<script>` 既可以出现在 HTML 文档的 `<head>` 部分，也可以出现在 `<body>` 部分。

一个 HTML 文档可以包含任意多个不重叠的 `<script>` 和 `</script>` 标记对。这些独立的脚本的执行顺序就是它们在文档中出现的顺序。尽管在装载和解析一个 HTML 文件的过程中，各个脚本在不同时刻执行，但是这些脚本却是同一个 JavaScript 程序的组成部分，因为在一个脚本中定义的函数和变量适用于随后出现的同一文件中的所有脚本。例如，可以将下面的一行脚本放到一个 HTML 页的某个地方：

```
<script>var x = 1;</script>
```

在这个页面中该行脚本之后，你可以引用变量 `x`，即使这个引用出现在另一个脚本块中。关键的环境是那个 HTML 页，而不是脚本块：

```
<script>document.write(x);</script>
```

`document.write()` 方法是个重要且常用的方法。如这里所示, 它将把自己的输出插入到脚本在文档中所在的位置。当脚本执行完时, HTML 解析器将继续解析文档, 并开始解析 `document.write()` 方法生成的文本。

例 12-1 展示的是一个 HTML 文件, 这个文件包含一个简单的 JavaScript 程序。注意这个例子和本书前面所示的一些代码段之间的差别, 这个程序和一个 HTML 文件结合在一起, 具有明确的运行环境。还要注意 `<script>` 标记中的 `language` 属性的用法。我们将在下一小节中解释这一属性。

例 12-1: 一个 HTML 文件中的简单 JavaScript 程序

```
<html>
<head>
<title>Today's Date</title>
  <script language="JavaScript">
    // 定义一个函数, 以便以后使用
    function print_todays_date() {
      var d = new Date();           // 得到今天的日期和时间
      document.write(d.toLocaleString()); // 把它插入文档
    }
  </script>
</head>
<body>
The date and time are:<br>
  <script language="JavaScript">
    // 现在调用我们上面定义的函数
    print_todays_date();
  </script>
</body>
</html>
```

12.2.1.1 属性 `language` 和 `type`

虽然迄今为止 JavaScript 是最常用的客户端脚本语言, 但它不是惟一的一种。为了告诉 Web 浏览器编写脚本时用的是哪种语言, 标记 `<script>` 使用了一个可选的属性 `language`。理解指定的脚本语言的浏览器将运行脚本, 不知道该语言的浏览器将忽略它。

如果要编写 JavaScript 代码, 可按如下方式使用 `language` 属性:

```
<script language="JavaScript">
  // 此处为 JavaScript 代码
```

```
</script>
```

如果编写脚本时使用的是 Microsoft 公司的 Visual Basic Scripting Edition 语言（注 2）则可以采用下面的方式使用该属性：

```
<script language=VBScript">  
    此处为VBScript 代码（' 是注释符号，类似于JavaScript中的“//”符号）  
</script>
```

JavaScript 是 Web 默认的脚本语言，如果省略了 language 属性，Netscape 和 Internet Explorer 都会假定脚本是用 JavaScript 编写的。

HTML 4 规范标准化了 <script> 标记，不过它不赞成使用 language 属性，因为它没有脚本语言名的标准集合。不过该规范主张使用 type 属性，将脚本语言指定为 MIME 类型。因此，理论上说来，嵌入式 JavaScript 脚本主张使用的方法（带有一个标记）如下：

```
<script type="text/javascript">
```

事实上，对 language 属性的支持仍然比新的 type 属性好。

HTML 4 规范还定义了为整个 HTML 文件指定默认脚本语言的标准（有用）方法。如果你计划把 JavaScript 作为一个文件中使用的惟一脚本语言，只需在文档的 <head> 部分加入如下代码：

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

如果这样做，无需指定 language 属性或 type 属性，就可以安全地使用 JavaScript 脚本。

由于 JavaScript 是默认的脚本语言，因此用它编写程序时，不必用 language 属性指定用什么语言编写脚本。但是，该属性还有另一个重要用途，即可以用来指定解释脚本需要的 JavaScript 版本。如果为一个脚本指定 language='JavaScript'，那么所有启用了 JavaScript 的浏览器就都能运行这个脚本。但是假定你编写的脚本使

注 2： 该语言也被称为 VBScript。只有 Internet Explorer 支持 VBScript，所以用这种语言编写的脚本是不可移植的。VBScript 和 HTML 对象的连接方式与 JavaScript 相同，只是其核心语法和 JavaScript 不同。本书中没有详细介绍 VBScript。

用了 JavaScript 1.5 的异常处理特性，那么要避免在不支持这种语言版本的浏览器中产生语法错误，需要将如下的标记嵌入你的脚本：

```
<script language="JavaScript1.5">
```

如果这样做，那么只有支持 JavaScript 1.5（和它的异常处理特性）的浏览器才能运行该脚本，其他浏览器都将忽略它。

language 属性中使用的字符串“JavaScript 1.2”需要重点介绍一下。在 Netscape 4 准备发布前，即将出现的 ECMA-262 标准显示出可能要对 JavaScript 语言的某些特性做不兼容的改动。为了防止这些不兼容的改动破坏现有的脚本，Netscape 公司的 JavaScript 设计者明智地进行了预防，从而实现只在 language 属性被显式地指定为“JavaScript 1.2”时才实现这些改动。遗憾的是，ECMA 标准在 Netscape 4 发布前还没有定案，而在它发布后，该标准却删除了那些被提议的不兼容改动。因此，指定 language="JavaScript 1.2" 使 Netscape 4 的行为既不与以前的浏览器兼容，也不与 ECMA 标准兼容（参阅 11.6 节了解这些不兼容性的完整细节）。因此，要避免把 language 属性设置为“JavaScript 1.2”。

12.2.1.2 </script> 标记

有时，你会发现自己编写的脚本用 document.write() 方法向另一个浏览器窗口或框架中输出脚本。如果编写了这样一个脚本，就应该加上 </script> 标记来终止它。必须注意，因为 HTML 解析器不理解 JavaScript 代码，如果它看到代码中的“</script>”字符串，即使 </script> 标记出现在引号中，HTML 解析器也会认为它发现了当前运行的脚本的结束标记。要避免这种问题，只需要将标记拆成片段，用表达式 "</" + "script>" 写出即可：

```
<script>
f1.document.write("<script.");
f1.document.write('document.write("<h2>This is the quoted script</h2>")');
f1.document.write("</" + "script>");
</script>
```

另外，还可以在 </script> 中使用转义符“/”：

```
f1.document.write('<\/script>');
```

12.2.1.3 defer 性质

HTML 4 定义了 `<script>` 标记的一个性质，虽然该性质还不常用，但是仍然很重要。前面简单地提到过，脚本可以调用 `document.write()` 方法动态地给文档添加内容。因此，当 HTML 解析器遇到脚本时，它必须停止解析文档，等待脚本执行。

如果你编写的脚本不会生成任何文档输出（例如定义函数，但从不调用 `document.write()` 的脚本），就可以在 `<script>` 标记中使用 `defer` 性质，暗示浏览器可以继续解析 HTML 文档，推迟执行脚本，直到它遇到了不能推迟执行的脚本为止。这样做会提高利用 `defer` 性质的浏览器的性能。注意，`defer` 不是一个值，只是必须出现在标记中：

```
<script defer>  
    // 不调用 document.write() 方法的 JavaScript 代码  
</script>
```

12.2.2 包括 JavaScript 文件

从 JavaScript 1.1 起，`<script>` 标记就支持 `src` 性质。这个性质的值指定了一个 JavaScript 代码文件的 URL。它的用法如下：

```
<script src='../javascript/util.js'></script>
```

JavaScript 文件的扩展名通常是 `.js`，它只包含纯粹的 JavaScript 代码，其中既没有 `<script>` 标记，也没有其他 HTML 标记。

具有 `src` 性质的 `<script>` 标记的行为就像指定的 JavaScript 文件直接出现在标记 `<script>` 和 `</script>` 之间一样。支持 `src` 性质的浏览器会忽略 `<script>` 和 `</script>` 之间的代码（不过像 Netscape 2 这样不认识该性质的浏览器仍旧执行它）。注意，即使是指定了 `src` 性质，而且在 `<script>` 和 `</script>` 之间没有任何 JavaScript 代码，还是需要结束标记 `</script>`。

使用 `src` 性质有很多好处：

- 它可以把大型 JavaScript 代码块移出 HTML 文件，从而简化了 HTML 文件。
- 当某个函数或 JavaScript 代码由几个不同的 HTML 文件共享时，可以将它放置

在一个单独的文件中,然后由那些需要它的HTML文件读取。这样不仅减少了硬盘空间的使用,而且使代码更易于维护。

- 如果使用JavaScript函数的页面不止一个,那么可以将它们放置在单独的JavaScript文件中使浏览器将其缓存起来,这样装载它们时速度就更快。由多个页面共享JavaScript代码时,虽然初次打开一个JavaScript文件要求浏览器打开一个单独的网络连接,以便下载那个JavaScript文件,但是高速缓存节省的时间远远大于这个延迟。
- 由于性质src的值可以是任意的URL,因此来自一个Web服务器的JavaScript程序或网页可以使用由另一个Web服务器输出的代码(如子程序库)。

12.2.3 事件处理程序

在包含它的HTML文件被读进浏览器的时候,脚本中的JavaScript代码只执行一次。仅使用这种静态脚本的程序不能动态地响应用户。很多动态性的程序都定义了事件处理程序,当某个事件发生时(如用户点击了表单内的一个按钮),Web浏览器会自动调用相应的事件处理程序。由于客户端JavaScript的事件是由HTML对象(如按钮)引发的,因此事件处理程序被定义为这些对象的性质。例如,要定义在用户点击表单中的复选框时调用的事件处理程序,只需把处理代码作为定义复选框的HTML标记的性质:

```
<input type="checkbox" name="opts" value="ignore-case"
      onclick="ignore-case = this.checked;"
>
```

在这段代码中,我们感兴趣的是性质onclick(注3)。onclick的性质值是一个字符串,其中包含一个或多个JavaScript语句。如果其中有多条语句,必须使用分号将每条语句隔开。当指定的事件(在这里是点击)在复选框中发生时,字符串中的JavaScript代码就会被执行。

虽然可以在事件处理程序定义中加入任意多条JavaScript语句,但是当要添加的简单语句多于一或两条时,就要采取一个常用的方法,即将事件处理程序的主体定义为标记<script>和</script>之间的一个函数。然后就可以从事件处理程序中

注3: 所有的HTML事件处理程序的特性名都以“on”开头。

调用这一函数。这样一来，大部分 JavaScript 代码都存放在脚本中，从而减少了 JavaScript 和 HTML 的混合。

我们将在第十九章中更为详细地介绍事件和事件处理程序，不过在此之前，你会看到它们出现在各种示例中。尽管第十九章具有完整的事件处理程序列表，但下面仍然介绍它们中最常用的几个：

onclick

所有类似按钮的表单元素和标记 `<a>` 及 `<area>` 都支持该处理程序。当用户点击元素时会触发它。如果 onclick 处理程序返回 false，则浏览器不执行任何与按钮或链接相关的默认动作，例如，它不会进行超链接（用于标记 `<a>`）或提交表单（用于 **Submit** 按钮）。

onmousedown, onmouseup

这两个事件处理程序和 onclick 非常相似，只不过分别在用户按下和释放鼠标按钮时触发。支持 onclick 处理程序的 Document 元素也支持这两个处理程序。在 IE 4 和 Netscape 6 中，几乎所有文档元素都支持这两个处理程序。

onmouseover, onmouseout

分别在鼠标指针移到或移出文档元素时触发这两个处理程序。标记 `<a>` 最常使用它们。如果 `<a>` 标记的 onmouseover 处理程序返回 true，那么它将制止浏览器在状态栏中显示链接的 URL。

onchange

`<input>`、`<select>` 和 `<textarea>` 元素支持这个事件处理程序。在用户改变了元素显示的值，或移出了元素的焦点时触发它。

onsubmit, onreset

`<form>` 标记支持这两个处理程序，在将要提交或重置表单时引发它们。它们返回 false 可以取消提交或重置。onsubmit 处理程序通常用于执行客户端表单的验证。

作为事件处理程序用法的真实示例，可以再研究一下例 1-3 所示的交互式借贷脚本。这个例子中的 HTML 表单含有大量事件处理程序性质。这些处理程序的主体非常简单，它们只是调用在 `<script>` 中的其他地方定义的 `calculate()` 函数。

12.2.4 URL 中的 JavaScript

另一种将 JavaScript 代码添加到客户端的方法是把它放置在伪协议说明符 `javascript:` 后的 URL 中。这个特殊的协议类型声明了 URL 的主体是任意的 JavaScript 代码, 它由 JavaScript 的解释器运行。如果 `javascript:` URL 中的 JavaScript 代码含有多个语句, 必须使用分号将这些语句分隔开。这样的 URL 如下所示:

```
javascript:var now = new Date(); "<h1>The time is:</h1>" + now;
```

当浏览器装载了这样的 URL 时, 它将执行这个 URL 中包含的 JavaScript 代码, 并把最后一条 JavaScript 语句的字符串值作为新文档的内容显示出来。这个字符串值可以含有 HTML 标记, 并被格式化, 其显示与其他装载进浏览器的文档完全相同。

JavaScript URL 还可以含有只执行动作, 但不返回值的 JavaScript 语句。例如:

```
javascript:alert("Hello World!");
```

装载了这种 URL 时, 浏览器仅执行其中的 JavaScript 代码, 但由于没有作为新文档来显示的值, 因此它并不改变当前显示的文档。

通常我们想用 `javascript:URL` 执行某些不改变当前显示的文档的 JavaScript 代码。要做到这一点, 必须确保 URL 中的最后一条语句没有返回值。一种方法是用 `void` 运算符显式地把返回值指定为 `undefined`, 只需要在 `javascript:URL` 的结尾使用语句 `void 0`; 即可。例如, 下面的 URL 将打开一个新的空浏览器窗口, 而不改变当前窗口的内容:

```
javascript>window.open("about:blank"); void 0;
```

如果这个 URL 中没有 `void` 运算符, `Window.open()` 方法的返回值将被转换成字符串并被显示出来, 当前窗口将被如下所示的文档覆盖:

```
[object Window]
```

任何使用正规 URL 的地方都可以使用 `javascript:URL`。采用这种语法的一个重要方式是在浏览器的 **Location** 域中直接输入它, 你可以在这里检测任何 JavaScript 代码, 而无须借助编辑器来创建一个包含该代码的 HTML 文件。

还可以在书签中使用 javascript:URL, 在这里它们可以构成有用的微型JavaScript 程序或“小书签”, 从菜单或书签工具栏可以很容易地装入这种小程序。

javascript:URL 还可以被用作超链接的 href 值。在用户点击这样的超链接时, 指定的 JavaScript 代码将被执行。如果把 javascript:URL 指定为 <form> 标记的 action 性质的值, 那么当用户提交表单时, URL 中的 JavaScript 代码就会被执行。在这样的环境中, javascript:URL 实际上是作为事件处理程序来使用。

在某些情况下, javascript: URL 可以和那些不支持事件处理程序的对象一起使用。例如, 在 Netscape 3 中, 标记 <area> 不支持 Windows 平台上的 onclick 事件处理程序 (但 Netscape 4 支持它)。因此, 在 Netscape 3 中, 如果想在用户点击客户端的图像时执行某些 JavaScript 代码, 就必须使用 javascript:URL。

12.2.5 非标准环境中的 JavaScript

Netscape 公司和 Microsoft 公司都在它们的浏览器中实现了 JavaScript 的专有扩展, 你可能偶然会看到不在本书介绍的环境中使用的 JavaScript 代码。例如, Internet Explorer 允许在 <script> 标记中为 for 和 event 性质定义事件处理程序。Netscape 4 允许在 <style> 标记中用 JavaScript 作为定义 CSS 样式表的替代语法。Netscape 4 还扩展了整个 HTML 语法, 允许 JavaScript 代码出现在实体中 (但只限于 HTML 性质的值)。这会产生如下的 HTML 代码:

```
<table border="&{getborderwidth()}; .
```

最后, Netscape 4 还支持条件注释的表单, 这些注释以 JavaScript 的实体语法为基础。注意, Netscape 6 和 Mozilla 浏览器 (Netscape 6 以其为基础) 不再支持使用这些非标准的 JavaScript。

12.3 JavaScript 程序的执行

前面几节讨论了将 JavaScript 代码集成到一个 HTML 文件的方法。现在我们将重点介绍 JavaScript 解释器是如何执行嵌入式 JavaScript 代码的。接下来的几节解释了不同形式的 JavaScript 代码是如何执行的。虽然其中有些材料是显而易见的, 但还有大量的重要细节需要详细介绍。

12.3.1 脚本

出现在 `<script>` 和 `</script>` 标记之间的 JavaScript 语句是按照它们出现的顺序执行的。如果一个文件中有多个脚本，这些脚本也按照它们出现的顺序执行。如果脚本调用了 `document.write()` 方法，那么传递给该方法的文本将被插入文档，紧接在结束标记 `</script>` 后，当脚本运行结束时，由 HTML 解析器解析。这一规则同样适用于用 `src` 性质从分离的文件中添加的脚本。

记住，脚本的执行过程是 Web 浏览器的 HTML 解析过程的一部分。所以，如果一个脚本出现在 HTML 文档的 `<head>` 部分，那么该文档的 `<body>` 部分就还没有被定义。这就是说如果代表文档主体内容的 JavaScript 对象（如 Form 和 Link）还没有被创建，脚本就不能对它进行操作。

你所编写的脚本不应该操作还没有创建的对象。例如，如果将脚本放置在 HTML 文件的一个表单之前，那么这个脚本就不能对这个 HTML 表单的内容进行操作。还有一些类似的规则，它们都基于这条基本规则。例如，Document 对象有一些属性只能在浏览器开始解析 HTML 文档的 `<body>` 部分之前从 `<head>` 部分的脚本设置。这种特殊规则在本书的客户端参考手册部分有详细的介绍，它们被列在受其影响的对象和属性的条目中。

由于脚本在解析和显示包含它们的 HTML 文件时执行，因此每个脚本的运行时间都不应该太长。因为脚本可以用 `document.write()` 方法动态地创建文档内容，所以所有 HTML 解析器必须在 JavaScript 解释器运行脚本时停止解析文档。HTML 文档只有在它包含的所有脚本都运行结束之后才会完整地显示出来。如果某个脚本执行的是繁琐的计算性的任务，需要花费较长的运行时间，那么等文档完全显示出来时，用户早已不耐烦了。所以，如果要用 JavaScript 执行大量计算，就应该定义一个执行计算的函数，然后当用户需要它时就用事件处理程序来调用这个函数，而不是在文档初次装载时就执行所有的计算。

前面提到过，使用 `src` 性质从外部 JavaScript 文件中读取的脚本的执行方式和直接包含在 HTML 文件中的脚本完全一样。这就是说，HTML 解析器和 JavaScript 解释器都会停下来等待外部的 JavaScript 文件装载进来（与嵌入的图像不同，在把脚本装载到背景中时 HTML 解析器不能继续运行）。装载 JavaScript 代码的外部文件时，即使使用的是相对较快的 modem 连接，也会引起装载和执行网页的显著延迟。当然，如果高速缓存 JavaScript 代码，这个问题就不存在了。

12.3.2 函数

我们知道，定义函数和执行函数不同。定义一个用来操作还没有创建的对象的功能很安全。只需要注意在必要的变量、对象都存在之前不要执行或调用这个函数。以前讲过，如果一个脚本出现在HTML文件的表单之前，你不能编写一个脚本来操作这个HTML表单。但是你可以编写一个脚本，让它来定义操作表单的函数，这样就不必考虑脚本和表单的相对位置了。实际上这种方法很常用。许多JavaScript程序都以文档<head>部分的脚本开始，这些脚本除了定义由HTML文件的<body>部分使用的函数外，什么都不做。

同样的，还有一些JavaScript程序只使用脚本来定义函数，此后这些函数将由事件处理程序调用。在下面一节中我们将看到，使用这种方法必须确保两件事，一是所有函数都要在事件处理程序调用它之前定义，二是事件处理程序和它们所调用的对象都不能使用还没有定义的对象。

12.3.3 事件处理程序

把事件处理程序定义为onclick或其他HTML性质的值与定义JavaScript函数非常相似，它们的代码都不会被立刻执行。事件处理程序的执行是异步的。由于事件通常发生在用户和HTML对象交互时，因此无法预言一个事件处理程序何时会被调用。

事件处理程序和脚本都必须遵循一个重要的约束，即它们的执行时间不能太长。我们已经知道，脚本必须运行得很快，因为在脚本结束运行之前HTML解析器不能继续进行解析。而事件处理程序不能运行得过久则是因为在程序结束对事件的处理之前，用户不能继续和程序进行交互。如果一个事件处理程序执行的是耗时的操作，给用户的感觉就是程序已经挂起、冻结或者崩溃了。

如果出于某种原因你必须在事件处理程序中执行一个耗时的操作，那么要确保用户明确地请求了这一操作，而且通知用户需要短时间的等待。我们将在第十三章中看到，可以弹出alert()对话框来通知用户，或者是在浏览器的状态栏中显示文本信息。另外，如果程序需要大量的后台处理，可以用方法setTimeout()安排一个函数，在空闲时间反复调用这一函数。

要知道,事件处理程序可以在一个网页被完全装载并解析之前执行,这一点非常重要。这很容易理解,假如你使用的是非常慢的网络连接,如果装载了一半的文档显示出超级链接和一些表单元素,则用户可以和这些HTML元素进行交互,因此就可能引起在装载文档的剩余部分之前,事件处理程序被调用。

在文档完全被装载之前事件处理程序就有可能被调用这一事实有两点含义。首先,如果事件处理程序要调用一个函数,那么必须确保在处理程序调用这个函数之前,该函数就已经被定义了。要做到这一点,一个方法是在HTML文档的<head>部分定义所有函数。文档的这个部分在解析<body>部分之前总是被完全解析(而且其中所有函数都会被定义)。由于定义事件处理程序的所有对象自身都是在<body>部分定义的,因此就能确保<head>部分的函数都在调用事件处理程序之前被定义。

第二点含义是必须确保事件处理程序不会操作还没有创建和解析的HTML对象。当然,一个事件处理程序总是可以安全地操作自己的对象以及在HTML文件中在它之前定义的对象。解决这一问题的策略是在定义网页用户界面时,采用一种只允许事件处理程序引用前面定义了的对象的方式。例如,假定你定义了一个表单,其中只有**Submit**按钮和**Reset**按钮使用事件处理程序,那么只需要将这些按钮放在表单的底部即可(这是一种友好的用户界面风格,告诉用户应该继续下去)。

在较为复杂的程序中,可以不必确保事件处理程序操作的都是在它之前定义了的对象,因此你要对这样的程序格外小心。如果一个事件处理程序只是要操作和它在同一个表单中定义的对象,那么你就不会遇到什么麻烦。但是如果你要操作的对象在另一个表单中,或者在其他框架中,那么问题就出现了。一种解决方法是在操作一个对象之前检测它的存在性。要做到这一点,只需要把它(和任何父对象)与null比较一下即可。例如:

```
<script>
function set_name_other_frame(name)
{
    if (parent.frames[1] == null) return; // 还没有定义其他框架
    if (!parent.frames[1].document) return; // 文档还没有装载它
    if (!parent.frames[1].document.myform) return; // 还没有定义表单
    if (!parent.frames[1].document.myform.name) return; // 还没有定义字段
    parent.frames[1].document.myform.name.value = name;
}
</script>


```

```
onchange="set_name_order_frame(this.value)";
```

在JavaScript 1.5 和其后的版本中，可以忽略上面代码中的存在性测试，而无需在完全装载文档之前调用函数，抛出一个异常，使用try/catch语句捕捉这个异常。

事件处理程序要确保所使用的对象都被定义了，还可以使用另外一种方法，这种方法与onload事件处理程序有关。这个事件处理程序是在一个HTML文件的<body>标记或<frameset>标记中定义的，当文档或框架设置完全被装载进来时就会调用它。如果在onload事件处理程序中设置了一个标志，那么其他事件处理程序就能够通过测试这个标志来判断它们是否能够安全运行，其依据就是如果文档已经被完全装载，并且包含的所有对象已经完全被定义了。例如：

```
<body onload="window.fullyLoaded = true;">
<form>
  <input type="button" value="Do It!"
    onclick="if (window.fullyLoaded) doIt();">
</form>
</body>
```

12.3.3.1 onload()和onunload()事件处理程序

在谈及JavaScript程序的执行顺序时，事件处理程序onload()和它的搭档onunload()尤为值得一提。这两个事件处理程序都是在HTML文件的标记<body>或<frameset>中定义的（没有一个HTML文件可以合法地同时含有这两个标记）。处理程序onload()是当文档或框架被完全装载时调用的，这就是说所有图像已经被下载并显示出来了，所有子框架也已经被装载进来了，所有Java小程序和插件都已经开始运行了，等等。在使用多框架时要格外注意，各个框架的onload事件处理程序的调用顺序不能确定，只是父框架的处理程序在所有子框架的处理程序之后调用。

处理程序onunload()是在刚要卸载页面之前执行的，这种情况发生在浏览器要显示新的页面时。你可以用它取消onload处理程序或网页中其他脚本的效果。例如，如果网页打开了第二个浏览器窗口，在用户转移到其他网页时，onunload处理程序提供了关闭那个窗口的机会。onunload处理程序不应该运行任何耗时的操作，也不应该弹出对话框。它只不过是要执行一个快速清除操作，它的运行速度不应该降低，也不应该阻碍用户过渡到一个新页面。

12.3.4 JavaScript URL

位于一个 `javascript:URL` 中的 JavaScript 代码在装载包含这个 URL 的文档时不会被执行。在浏览器装载 URL 所引用的文档之前，这些代码不会被解释。在一个用户输入了一个 JavaScript URL，或者在用户点击了一个链接或客户端图像时，以及用户提交了一个表单时会发生这种情况。`javascript:URL` 通常用于代替事件处理程序，所以和事件处理程序类似，这种 URL 中的代码可以在一个文档被完全装载之前执行。因此，对于 `javascript:URL`，你必须抱有和事件处理程序一样的警觉度，要确保它们不会引用还没有定义的对象（或函数）。

12.3.5 Window 对象和变量的生存期

有关客户端 JavaScript 程序如何运行的问题，我们要研究的最后一个主题是变量的生存期。我们知道，对客户端 JavaScript 来说，Window 对象是它的全局对象，所有的全局变量都是 Window 对象的属性。那么当浏览器从一个页面转而显示下一页面时，Window 对象和它们所包含的变量会发生什么变化呢？

一个新文档被装载到窗口或框架中时，那个窗口或框架的 Window 对象会被恢复为它的默认状态，即由前一个文档中的脚本定义的所有属性和函数都将被删除，所有被改变或覆盖了的标准系统属性都会被恢复。每个文档都以“清白的历史”开始。你的脚本可以信赖这一点，它们不会继承前一个文档受损的环境。脚本定义的所有变量和函数都只存在到新的文档替换了你的文档为止。

我们这里所说的“清白的历史”是表示装载文档的窗口或框架的 Window 对象。如我们前面讨论的，这个 Window 对象是那个窗口或框架中的 JavaScript 代码的全局对象。但如果你在使用多框架或多窗口，那么某个窗口中的脚本可以引用表示其他窗口或框架的 Window 对象。所以，除了考虑 Window 对象中定义的变量和函数的持续性外，还必须考虑 Window 对象自身的持续性。

只要浏览器的顶级窗口存在，那么代表它的 Window 对象就会一直存在。无论这个窗口装载和卸载了多少页面，对它的 Window 对象的引用都有效。这个顶级窗口打开多久，它的 Window 对象就会存活多久（注4）。

注4：当窗口被关闭，这个 Window 对象也不会被销毁。如果在其他窗口中仍旧存在对这个 Window 对象的引用，那么这个对象就不会被作为无用存储单元回收。但是，引用一个已经关闭的窗口并没有什么实际价值。

代表一个框架的 Window 对象仍然有效，只要那个框架仍然存在于包含它的框架或窗口中。例如，框架 A 含有一个脚本，它引用了框架 B 的 Window 对象，即使框架 B 装载了新文档，框架 A 对 B 的 Window 对象的引用依然有效。当新文档装载进来时，在框架 B 的 Window 对象中定义的所有变量和函数都将被删除，但是 Window 对象自身仍然有效（直到包含它的框架或窗口装载了新文档并覆盖了框架 A 和框架 B 为止）。

这意味着，无论 Window 对象代表的是顶级窗口还是框架，它的存在都相当持久。Window 对象的生存期可能比它包含和显示的网页长，也可能比它显示的网页所包含的脚本长。

第十三章

窗口和框架

第十二章介绍了 Window 对象和它在客户端 JavaScript 中扮演的重要角色。我们已经知道，Window 对象是客户端 JavaScript 程序的全局对象，而且如图 12-1 所示，它还是客户端对象层次的根。

除了这些特殊的作用之外，就 Window 自身来说，它也是一个重要的对象。每个浏览器窗口以及窗口中的框架都是由 Window 对象表示的。Window 对象定义了许多属性和方法，这些属性和方法在客户端 JavaScript 程序设计中非常重要。本章对这些属性和方法进行了探讨，而且说明了用窗口和框架进行程序设计时的一些重要方法。注意，由于 Window 对象在客户端的程序设计中非常重要，所以本章的篇幅相当长。你不必急于一次掌握所有的内容，将它分成几个小块更易于学习。

13.1 Window 对象概述

首先，我们来简要介绍一下 Window 对象最常用的属性和方法。本章后面的几节会对这些内容做出详细的解释。另外，本书的客户端参考手册部分也列出了 Window 对象的全部属性和方法。

最重要的 Window 属性有：

closed

一个布尔值，只有当窗口被关闭时，它才为 true。

defaultStatus, status

在浏览器状态栏中显示的文本。

document

对 Document 对象的引用，该对象代表在窗口中显示的 HTML 文档。我们将在第十四章中详细介绍 Document 对象。

frames[]

Window 对象的数组，代表窗口中的各个框架（如果存在）。

history

对 History 对象的引用，该对象代表用户浏览窗口的历史。

location

对 Location 对象的引用，该对象代表在窗口中显示的文档的 URL。设置这个属性会引发浏览器装载一个新文档。

name

窗口的名称。可被 HTML 标记 <a> 的 target 性质使用。

opener

对打开当前窗口的 Window 对象的引用。如果当前窗口被用户打开，则它的值为 null。

parent

如果当前的窗口是框架，它就是对窗口中包含这个框架的引用。

self

自引用属性，是对当前 Window 对象的引用，与 window 属性同义。

top

如果当前窗口是框架，它就是对包含这个框架的顶级窗口的 Window 对象的引用。注意，对于嵌套在其他框架中的框架，top 不等同于 parent。

window

自引用属性，是对当前 Window 对象的引用，与 self 属性同义。

Window 对象还支持大量重要的方法:

`alert()`, `confirm()`, `prompt()`

向用户显示简单的对话框, `confirm()` 和 `prompt()` 用于获取用户的响应。

`close()`

关闭窗口。

`focus()`, `blur()`

请求或放弃窗口的键盘焦点。`focus()` 方法还通过把窗口提到堆栈顺序的最前面, 从而确保窗口可见。

`moveBy()`, `moveTo()`

移动窗口。

`open()`

打开新的顶级窗口, 用指定的特性显示指定的 URL。

`print()`

打印窗口或框架中的内容, 就像用户点击了窗口工具栏中的 **Print** 按钮一样(只有 Netscape 4 和其后的版本以及 IE 5 和其后的版本支持该方法)。

`resizeBy()`, `resizeTo()`

调整窗口大小。

`scrollBy()`, `scrollTo()`

滚动窗口中显示的文档。

`setInterval()`, `clearInterval()`

设置或者取消重复调用的函数, 该函数在两次调用之间有指定的延迟。

`setTimeout()`, `clearTimeout()`

设置或者取消在指定的若干毫秒后要调用一次的函数。

从上面的列表中可以发现, Window 对象提供了大量功能。本章余下的部分就详细探讨了其中的大部分功能。

13.2 简单的对话框

三种常用的 Window 方法是 `alert()`、`confirm()` 和 `prompt()`，它们的功能都是弹出简单的对话框。`alert()` 用于向用户显示消息。`confirm()` 要求用户点击 **OK** 或 **Cancel** 按钮来确认或取消某个操作。`prompt()` 要求用户输入一个字符串。图 13-1 显示的是由这些方法创建的示例对话框。

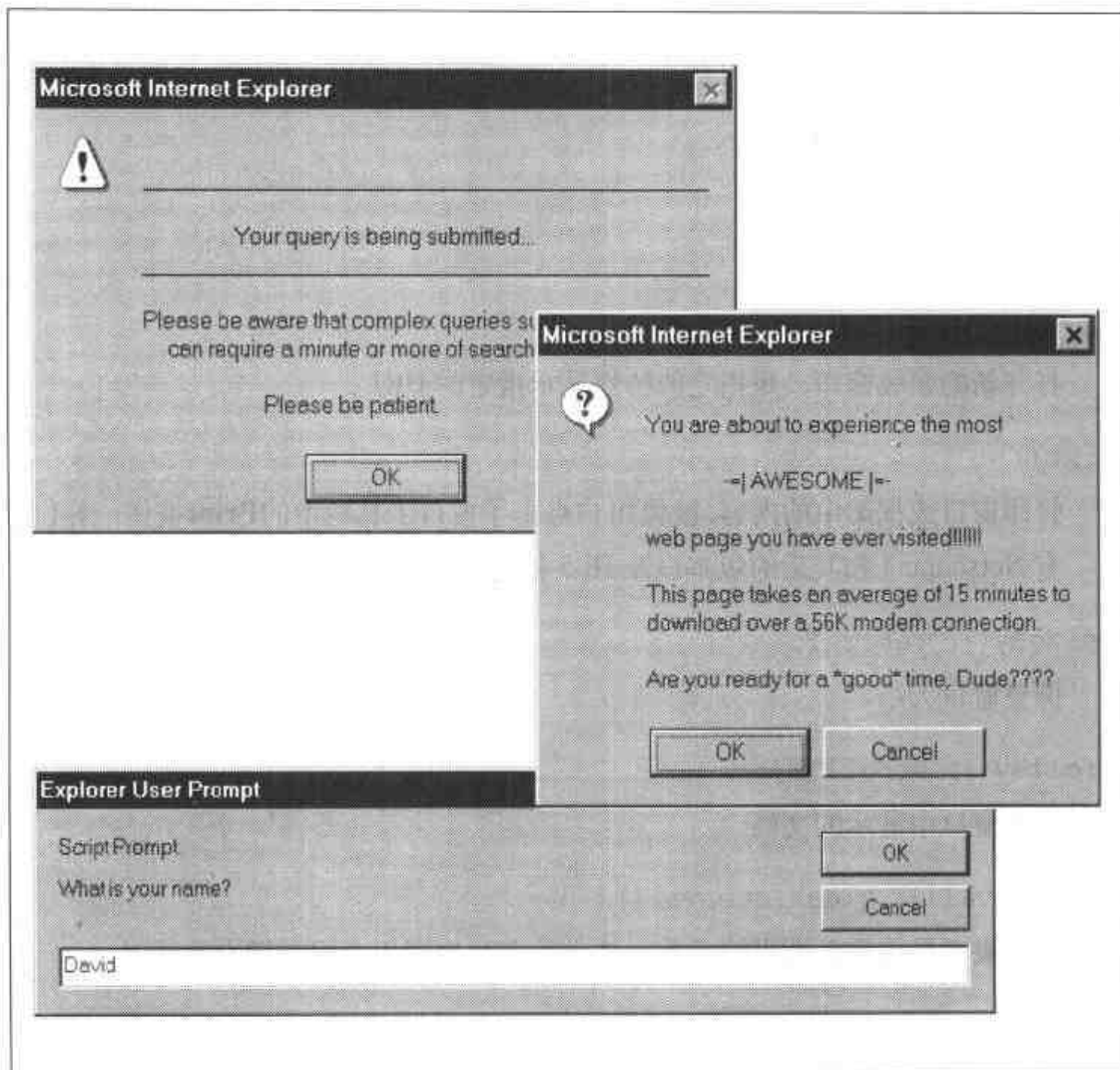


图 13-1: `alert()`、`confirm()` 和 `prompt()` 生成的对话框

注意，这些对话框中显示的文本是纯文本，而不是 HTML 格式的文本。只能使用空格、换行符和各种标点符号来格式化这些对话框。调整其布局通常要求测试，而且会出错。但是要记住，在不同的平台和不同的浏览器中的对话框看来会有所不同，因此你不能指望自己的格式化在所有的浏览器中看起来都正确。

有些浏览器（如 Netscape 3 和 4）会在 `alert()`、`confirm()` 和 `prompt()` 生成的对话框的标题栏或左上角显示“JavaScript”。虽然它看来比较讨厌，但应该将它看作一个特性，而不是一个 bug，因为就是这一点防止了编写欺骗程序，通过显示具有哄骗性的系统对话框，骗用户输入口令或做一些他们不应该做的事。

方法 `confirm()` 和 `prompt()` 都会产生阻塞，也就是说，在用户关掉它们所显示的对话框之前，它们不会返回。这就意味着在弹出一个对话框时，代码就会停止运行。如果当前正在装载文档，也会停止装载，直到用户用要求的输入进行了响应为止。没有方法可以防止这些方法产生的阻塞，因为它们的返回值是用户的输入，所以在返回之前它们必须等待用户进行输入。在大多数浏览器中，`alert()` 方法也将产生阻塞，并等待用户关闭对话框。但在某些浏览器中（尤其是 Unix 平台上的 Netscape 3 和 4），`alert()` 方法并不产生阻塞。在实践中，这点小小的不兼容性很少引起问题。

例 13-1 说明了这些方法的典型用法。

例 13-1：方法 `alert()`、`confirm()` 和 `prompt()` 的用法

```
// 下面的函数将使用 alert() 方法来通知用户提交表单需要一段时间，用户应该耐心等待。
// 该函数近于用 HTML 表单的 onsubmit 事件处理程序中。
// 注意，所有的格式都是用空格，换行符和下划线实现的。
function warn_on_submit()
{
    alert("\n_____ \n\n" +
        "                Your query is being submitted...\n" +
        "_____ \n\n" +
        "Please be aware that complex queries such as yours\n" +
        "    can require a minute or more of search time.\n\n" +
        "                Please be patient.");
}

// 下面使用了 confirm() 方法来询问用户是否真的想访问一个下载时间较长的 Web 页。
// 注意该方法的返回值，它说明了用户的答案。
// 基于这一答案可以把浏览器重新导向正确的页面。
var msg = "\nYou are about to experience the most\n\n" +
    "        -=| AWESOME |=-\n\n" +
    "web page you have ever visited!!!!\n\n" +
    "This page takes an average of 15 minutes to\n" +
    "download over a 56K modem connection.\n\n" +
    "Are you ready for a *good* time, Dude????";

if (confirm(msg))
    location.replace("awesome_page.html");
else
```

```
location.replace( home_page.html );  
// 下面的代码非常简单。它使用 prompt() 方法获取用户名。  
// 然后在动态生成的 HTML 中使用那个名字。  
n = prompt("What is your name?", "");  
document.write("<hr>Hi welcome to my home page, " + n + "</hr>");
```

13.3 状态栏

除了创建时明确不使用状态栏的浏览器窗口，在每个浏览器窗口的底部都有一个状态栏 (status line)。这是浏览器向用户显示消息的地方。例如，当用户将鼠标移动到一个超级链接上时，浏览器将显示这个链接所指的 URL。当用户将鼠标移动到浏览器的一个控制按钮上时，浏览器将显示一条简单的上下文帮助消息来解释这个按钮的作用。在程序中同样可以利用状态栏，它的内容由 Window 对象的两个属性控制，即 status 和 defaultStatus。

当用户将鼠标指针移动到一个超级链接时，浏览器会显示出这个链接的 URL，虽然这是一种通用的情况，但是在你浏览网页时，却会发现一些链接的行为并非如此，它们显示一些文本，而不是链接的 URL。这是通过 Window 对象的 status 属性和超级链接的事件处理程序 onmouseover 实现的：

```
<!-- 以下是在超链接中设置状态栏的方法。  
-- 注意，要实现这一点，事件处理函数必须返回 true。 -->  
Lost? Dazed and confused? Visit the  
<a href='sitemap.htm' onmouseover= status= Go to Site Map ; return true; ">  
  Site Map  
</a>  
<!-- 可以对客户端的图像做同样处理， -->  
<img src= 'images/irmap1.gif' usemap='#map1' >  
<map name='map1'>  
  <area coords='0,0,50,20' href='info.htm'  
    onmouseover= status='Visit our Information Center'; return true; '>  
  <area coords='0,20,50,40' href='order.html'  
    onmouseover= 'status='Place an order'; return true; '>  
  <area coords='0,40,50,60' href='help.html'  
    onmouseover= 'status= Get help fast!'; return true; ">  
</map>
```

上例中的事件处理程序 onmouseover 必须返回 true。这就通知了浏览器当事件发生时不应该执行自己默认的动作，也就是说不应该在状态栏中显示链接的 URL。如果你忘记了返回 true，那么浏览器就会用它自己的 URL 覆盖处理程序在状态栏中

显示的所有消息。如果你对本例中的事件处理程序不是十分理解，也不必担心，我们将在第十九章中对事件进行解释。

当用户将鼠标指针移动到一个超级链接上时，浏览器会显示该链接的URL。当鼠标离开它时，浏览器又会把这个URL擦掉。如果你使用事件处理程序 `onmouseover` 来设置 Window 对象的 `status` 属性，情况是相同的，即当鼠标移动到一个超级链接上时，就显示你定制的消息，当鼠标离开这个链接时，就擦掉它。

从前面的示例中我们可以知道，属性 `status` 主要用于存放瞬时消息。但有时你可能想在状态栏中显示一些非瞬时的消息，如给浏览网页的用户显示一条欢迎消息或者给新的访问者显示一条简单的帮助文本等。要实现这一点，就要设置 Window 对象的 `defaultStatus` 属性，这个属性指定了显示在状态栏中的默认文本。这条文本可以在鼠标指针移动到一个超级链接或浏览器的控制按钮上时暂时地被URL、上下文帮助消息或者其他瞬时文本所代替，但是当鼠标离开那些区域时，又会恢复到默认的文本。

可以用下面所示的方法使用 `defaultStatus` 属性向真正的新手提供一条既友好又有帮助的消息：

```
<script>
defaultStatus = 'Welcome! Click on underlined blue text to navigate.' ;
</script>
```

13.4 超时设定和时间间隔

Window 对象的方法 `setTimeout()` 用来安排一个 JavaScript 的代码段在将来的某个指定的时间运行。方法 `clearTimeout()` 则用于取消那段代码的执行。`setTimeout()` 常用于执行一个动画或其他重复的动作。如果运行了一个函数，使用方法 `setTimeout()` 进行调度，使自己再次被调用，这样我们就得到了一个无须用户干涉就可以反复执行的进程。JavaScript 1.2 添加了 `setInterval()` 和 `clearInterval()` 方法，它们的功能与 `setTimeout()` 和 `clearTimeout()` 相似，只不过它们会自动地重新调度要反复运行的代码，无需代码自己进行再调度。

方法 `setTimeout()` 常和属性 `status` 或 `defaultStatus` 一起使用，在浏览器的状

态栏中显示某条消息。一般说来,在状态栏中显示的动画华而不实,应该避免使用它们。但是,有几种状态栏显示技术很有用,而且看起来比较高雅。例 13-2 就显示了一个雅致的状态栏动画。它在状态栏中显示当前的时间,每分钟更新一次。由于更新每分钟进行一次,所以它不会像其他的动画那样在浏览器的底部生成一个长期闪烁的令人分心的事物。

注意 <body> 标记的事件处理程序 onload 的用法,它用来执行对方法 display_time_in_status_line() 的初次调用。这个事件处理程序只在 HTML 文档完全被装载进浏览器时才会被调用一次。经过初次调用之后,该方法就使用 setTimeout() 来进行调度,即每隔 60 秒调用它一次,这样它就可以更新要显示的时间。

例 13-2: 状态栏中的数字时钟

```
<html>
<head>
<script>
// 该函数将在状态栏显示时间
// 一旦调用该函数就会激活时钟,此后它将自己调用自己
function display_time_in_status_line()
{
    var d = new Date();           // 获取当前时间
    var h = d.getHours();         // 提取小时: 0~23
    var m = d.getMinutes();       // 提取分钟: 0~59
    var ampm = (h >= 12) ? 'PM' : 'AM'; // 是上午还是下午
    if (h > 12) h -= 12;          // 将 24 小时的计时形式转换成
                                // 12 小时的计时形式
    if (h == 0) h = 12;           // 把 0 点转换成午夜
    if (m < 10) m = '0' + m;      // 把 0 分转换成 00 分, 等等
    var t = h + ':' + m + ' ' + ampm; // 将所有值连接起来
    defaultStatus = t;           // 在状态栏中显示它

    // 安排 1 分钟后再重复以上操作
    setTimeout("display_time_in_status_line()", 60000); // 60000 毫秒为 1 分钟
}
</script>
</head>
<!-- 在装载完所有内容之前,不必启动时钟
      -- 毕竟装载过程中状态栏还要显示其他信息 -->
<body onload="display_time_in_status_line();" >
<!-- 此处为 HTML 文档 -->
</body>
</html>
```

在 JavaScript 1.2 中,可以使用 setInterval() 来代替例 13-2 中的 setTimeout()。这样,方法 display_time_in_status_line() 就不再调用 setTimeout(), 事件

处理程序 `onload` 将被删除。当定义完 `display_time_in_status_line()` 后, 脚本将调用 `setInterval()` 来调度那个每隔 60 000 毫秒就自动重复一次的方法。

13.5 错误处理

Window 对象的 `onerror` 属性比较特殊。如果给这个属性赋一个函数, 那么只要这个窗口中发生了 JavaScript 错误, 该函数就会被调用, 即它成了窗口的错误处理函数。

传递给错误处理程序的参数有三个。第一个是描述错误的消息, 它可以是 “missing operator in expression” (表达式中缺少运算符)、“self is read-only” (只读的) 或 “myname is not defined” (还没有定义名称)。第二个参数是一个字符串, 它存放引发错误的 JavaScript 代码所在的文档的 URL。第三个参数是文档中发生错误的行代码。错误处理程序可以利用这三个参数做任何事情。典型的错误处理程序将向用户显示错误消息, 把它记入日志或强迫性地忽略错误。

除了这三个参数外, `onerror` 处理程序的返回值也很重要。当发生错误时, 浏览器通常在一个对话框或状态栏中显示错误消息。如果 `onerror` 返回 `true`, 它通知系统处理程序将处理错误, 无需其他操作, 换句话说, 就是系统无需显示自己的错误消息。例如, 如果你不想让错误消息打扰用户, 那么无论你编写的代码中有多少 bug, 都可以在所有 JavaScript 程序的开头使用如下代码:

```
self.onerror = function() { return true; }
```

当然, 当程序运行失败, 而没有生成任何错误消息时, 这样做很难使用户给你反馈消息。

我们将在例 14-1 中看到一个使用 `onerror` 处理程序的例子。这个例子用 `onerror` 处理程序向用户显示错误细节, 并允许用户提交含有这些细节的 bug 报告。

注意, `onerror` 错误处理程序在 Netscape 6 中存在 bug。虽然在发生错误时会触发你指定的函数, 但是传递给该函数的三个参数不正确, 而且不可用。Netscape 6 和其他支持 JavaScript 1.5 的浏览器有另外一种捕捉和处理错误的方法, 它们可以使用 `try/catch` 语句 (详见第六章)。

13.6 Navigator 对象

属性 `Window.navigator` 引用的是包含 Web 浏览器总体信息（如版本和它可以显示的数据格式列表）的 `Navigator` 对象。`Navigator` 对象是在 Netscape Navigator 之后命名的，不过 Internet Explorer 也支持它。IE 还支持属性 `clientInformation`，它是 `navigator` 的同义词。遗憾的是，Netscape 和 Mozilla 不支持这一属性。

`Navigator` 对象有五个主要属性用于提供正在运行的浏览器的版本信息：

`appName`

Web 浏览器的简单名称。

`appVersion`

浏览器的版本号和（或）其他版本信息。注意，这应该被视为“内部”版本号，因为它不总是与显示给用户的版本号一致。例如，Netscape 6 的版本号是 5.0，因为从未发布过 Netscape 5。另外，IE 4 到 6 的版本号都是 4.0，这说明它们与第四代浏览器的基准功能兼容。

`userAgent`

浏览器在它的 `USER-AGENT HTTP` 标题中发送的字符串。这个属性通常包含 `appName` 和 `appVersion` 中的所有信息。

`appCodeName`

浏览器的代码名。Netscape 用代码名“Mozilla”作为这一属性的值。为了兼容，IE 也采用这种方式。

`platform`

运行浏览器的硬件平台。这一属性是 JavaScript 1.2 新加的。

下面几行 JavaScript 代码在一个对话框中显示了 `Navigator` 对象的这些属性。

```
var browser = 'BROWSER INFORMATION:\n';
for(var propname in navigator) {
    browser += propname + ': ' + navigator[propname] + '\n'
}
alert(browser);
```

图 13-2 显示的是这段代码在 IE 6 中运行时显示的对话框。

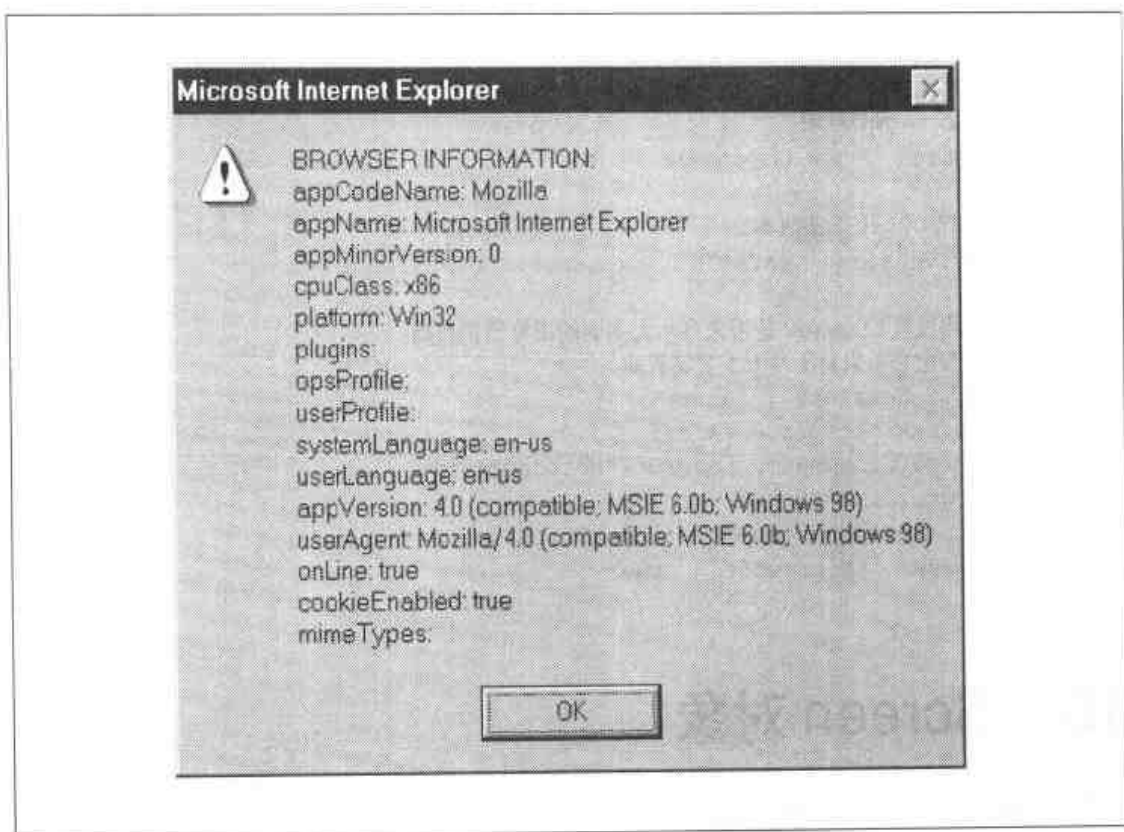


图 13-2: Navigator对象的属性

从图 13-2 可以看出, Navigator对象的属性值有时比我们所想像的要复杂得多。例如, 我们通常注意的只是appVersion属性的第一个数字。当使用Navigator对象来检测浏览器信息时, 我们通常使用parseInt()和String.indexOf()方法将我们需要的信息提取出来。例 13-3 就是实现这一功能的代码, 它对Navigator对象的属性进行了处理, 然后将它们存储在一个名为browser的对象中。这些属性的经过处理的形式用起来比原始的navigator属性更容易。形容这类代码的术语是“客户端探测器”, 你可以在Internet上找到更复杂、更通用的探测器代码(注1)。但出于多种原因, 还是例 13-3 所示的简单代码作用得较好。

例 13-3: 确定浏览器的销售商和版本

```
/*
 * 文件: browser.js
 * 由 <script SRC='browser.js'></script> 包括进来
 */
```

注 1: 例如, http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html 就是一个网站。

```
* 一个简单的探测器，用于确定浏览器的版本和销售商。  
* 它将创建一个名为 browser 的对象，该对象比 navigator 对象更容易使用。  
*  
// 创建 browser 对象。  
var browser = new Object();  
  
// 设置 browser 的版本。  
browser.version = parseInt(navigator.appVersion);  
  
// 以下将设置 browser 是否来自两大主要的浏览器销售商。  
// 首先假定它不来自于两大主要销售商。  
browser.isNetscape = false;  
browser.isMicrosoft = false;  
if (navigator.appName.indexOf("Netscape") != -1)  
    browser.isNetscape = true;  
else if (navigator.appName.indexOf("Microsoft") != -1)  
    browser.isMicrosoft = true;
```

13.7 Screen 对象

在 JavaScript 1.2 中，Window 对象的 screen 属性引用 Screen 对象。这个 Screen 对象提供有关用户显示器的大小和可用的颜色数量的信息。属性 width 和 height 指定的是以像素为单位的显示器大小。属性 availWidth 和 availHeight 指定的是实际可用的显示器大小，它们排除了像 Windows 任务栏这样的特性所占有的空间。可以使用这些属性来确定要加入文档的图像大小，或者在创建多个浏览器窗口的应用程序中确定要创建的窗口的大小。

属性 colorDepth 指定可以显示的颜色数的以 2 为底的对数。这个值通常与显示器所使用的每个像素的位数相同。例如，一个 8 位的显示器可以显示的颜色数是 256，如果所有颜色对浏览器来说都是可用的，那么 screen.colorDepth 属性的值就是 8。但是在某些环境中，浏览器可能对自身有所限制，可以显示的是可用颜色的子集。这时你就会发现，screen.colorDepth 的值小于屏幕的每个像素的位数。如果同一个图像有几个不同版本，它们分别使用不同数量的颜色，那么就可以通过测试这个 colorDepth 属性来确定文档中要使用的版本。

本章后面的例 13-4 说明了如何使用 Screen 对象。

13.8 Window 对象的控制方法

Window 对象定义了几个可以用来对窗口自身进行高级控制的方法。下面的几节就探究了如何使用这些方法来打开窗口、关闭窗口、控制窗口的位置和大小、请求或放弃键盘焦点以及滚动窗口中的内容。最后我们给出了一个例子用于说明这些特性。

13.8.1 打开窗口

使用 Window 对象的 `open()` 方法可以打开一个新的浏览器窗口。这个方法有四个可选的参数，返回的是代表新打开的窗口的 Window 对象。`open()` 的第一个参数是要在新窗口中显示的文档的 URL。如果这个参数被省略了（可以是 `null`，也可以是空字符串），那么新打开的窗口将是空的。

`open()` 的第二个参数是新打开的窗口的名字。这个名字可以作为 `<a>` 标记或 `<form>` 标记的 `target` 属性的值，我们将在本章后面的小节中讨论这一点。如果你指定的是一个已经存在的窗口的名字，那么 `open()` 使用的就只是那个已经存在的窗口，而不是再打开一个新窗口。

`open()` 的第三个参数是特性列表，这些特性声明了窗口的大小和它的 GUI 装饰。如果省略了这个参数，那么新窗口就会用一个默认的大小，而且具有一套标准的特性，即菜单栏、状态栏、工具栏等。如果指定了这个参数，就可以明确地规定新窗口的大小和要具有的特性。例如，要打开的是一个较小的、可调整的浏览器窗口，它具有状态栏，但是没有菜单栏、工具栏和地址栏，可以使用如下的 JavaScript 代码：

```
var w = window.open("smallwin.html", "smallwin",
    "width=400,height=300,status=yes,resizable=yes");
```

注意，当指定第三个参数时，所有没有明确声明的特性都会被省略。参阅本书客户端参考手册部分的 `Window.open()`，其中列出了所有可用特性和它们名称的集合。

`open()` 的第四个参数只在第二个参数命名的是一个已经存在的窗口时才有用。它是一个布尔值，声明了由第一个参数指定的 URL 是应该替换掉窗口浏览历史的当前项（`true`），还是应该在窗口浏览历史中创建一个新项（`false`），后者是默认的设置。

`open()` 的返回值是代表新创建的窗口的 `Window` 对象。你可以在自己的 JavaScript 代码中使用这个 `Window` 对象来引用新创建的窗口，就像你使用隐式的 `Window` 对象 `window` 来引用运行代码的窗口一样。但是如果在新窗口中运行的 JavaScript 代码要引用打开它的窗口又如何呢？在 JavaScript 1.1 和其后的版本中，窗口的 `opener` 属性引用的是打开它的窗口。如果当前窗口是由用户创建的，而不是由 JavaScript 代码创建的，那么它的 `opener` 属性就是 `null`。

有关 `open()` 方法的重要一点是调用它的方式总是 `window.open()`，即使 `window` 引用的是全局对象，原则上说来它是可以省略的。明确声明 `window` 是因为 `Document` 对象也有 `open()` 方法，因此指定 `window.open()` 有助于清楚地说明我们要做什么。这不仅是一个具有帮助性的好习惯，而且在某些环境下，也是必须的，在第十九章我们将会知道，事件处理程序是在定义它们的对象的作用域中执行的。例如，在执行一个 HTML 按钮的事件处理程序时，作用域链就包括 `Button` 对象、包含按钮的 `Form` 对象、包含表单的 `Document` 对象以及包含文档的 `Window` 对象。这样一来，如果该事件处理程序只引用了 `open()` 方法，该标识符在 `Document` 对象中停止解析，事件处理程序打开的是一个文档，而不是一个新的窗口。

我们将在例 13-4 中看到方法 `open()` 的用法。

13.8.2 关闭窗口

就像方法 `open()` 打开一个新窗口一样，方法 `close()` 将关闭一个窗口。如果我们已经创建了一个 `Window` 对象 `w`，可以使用如下的代码将它关掉：

```
w.close();
```

运行在那个窗口中的 JavaScript 代码则可以使用下面的代码关闭：

```
window.close();
```

再次提醒一下，要明确地使用标识符 `window`，这样可以避免混淆 `Window` 对象的 `close()` 方法和 `Document` 对象的 `close()` 方法。

大多数浏览器只允许你自动关闭由自己的 JavaScript 代码创建的窗口。如果要关闭其他窗口，可以用一个对话框提示用户，要求他对关闭窗口的请求进行确认（取消）。这样就可以防止那些不顾及别人的脚本编写者编写关闭你的主浏览器窗口的代码。

在JavaScript 1.1和其后的版本中，即使一个窗口关闭了，代表它的Window对象仍然存在。但除了检测它的closed属性外，就不应该再使用它的其他属性或方法了。如果窗口被关闭，则属性为true。记住，用户是可以在任何时候任何地点关闭窗口的，因此要避免出错，就要定期的检测你要使用的窗口是否仍然是打开的，这是个不错的建议。我们将要看到的例13-4就是这样一个例子。

13.8.3 窗口的几何大小

在JavaScript 1.2中，方法moveTo()可以将窗口的左上角移动到指定的坐标。同样，方法moveBy()可以将窗口上移、下移或者左移、右移指定数量的像素。方法resizeTo()和resizeBy()可以按照相对数量和绝对数量调整窗口的大小，它们也是在JavaScript 1.2中新出现的。注意，有些安全性攻击需要的代码在非常小或移出屏幕的窗口中运行，用户可能注意不到这样的窗口，为了防止这种攻击，浏览器会限制你不能将窗口移出屏幕的或使窗口非常小。

13.8.4 键盘焦点和可见性

方法focus()和blur()提供了对窗口的高级控制。调用focus()会请求系统将键盘焦点赋予窗口，调用blur()则会放弃键盘焦点。此外，方法focus()还会把窗口移到堆栈顺序的顶部，使窗口可见。在使用Window.open()方法打开新窗口时，浏览器会自动在顶部创建窗口。但是如果它的第二个参数指定的窗口名已经存在，open()方法不会自动使那个窗口可见。因此，在调用open()后调用focus()很常用。

focus()和blur()方法都是在JavaScript 1.1和其后的版本中定义的。

13.8.5 滚动

Window对象还具有一些在窗口或框架中滚动文档的方法。scrollBy()会将窗口中显示的文档向左、向右或者向上、向下滚动指定数量的像素。scrollTo()会将文档滚动到一个绝对位置。它将移动文档以便在窗口文档区的左上角显示指定的文档坐标。这两个方法都是在JavaScript 1.2中定义的。在JavaScript 1.1中，方法

`scroll()` 执行的功能和 JavaScript 1.2 中的方法 `scrollTo()` 一样。`scrollTo()` 是首选方法，保留 `scroll()` 只是为了向后兼容。

在 JavaScript 1.2 中，`Document` 对象的 `anchors[]` 数组的元素是 `Anchor` 对象。每个 `Anchor` 对象都有属性 `x` 和 `y`，这些属性指定了文档中的锚元素所在的位置。这样一来，将方法 `scrollTo()` 和这些值联合起来使用就可以滚动到文档中的已知位置。另外，在 IE 4 及其后的版本和 Netscape 6 及其后的版本中，文档元素都定义了 `focus()` 方法，调用某个元素的 `focus()` 方法可以对文档进行必要的滚动，确保该元素可见。

13.8.6 Window 方法示例

例 13-4 使用了我们讨论过的 `Window` 方法 `open()`、`close()` 和 `moveTo()`，此外还用到了其他一些窗口的程序设计方法。首先它创建了一个新窗口，然后使用方法 `setInterval()` 反复调用一个函数，在屏幕上不断地移动这个窗口。它还使用 `Screen` 对象确定了屏幕的大小，然后根据这一信息在窗口到达屏幕边界时将它反弹回来。

例 13-4：窗口的移动

```
<script>
// 下面是动画的初始值。
var x = 0, y = 0, w=200, h=200; // 窗口的位置和大小。
var dx = 5, dy = 5; // 窗口的速率。
var interval = 100; // 更新的毫秒数。

// 创建要移动的窗口。
// javascript: 是一种显示短文档的简单方法
// 最后一个参数指定了窗口的大小。
var win = window.open('javascript:<h1>BOUNCE!</h1>', '',
    'width=' + w + ',height=' + h);

// 设置窗口的初始位置
win.moveTo(x,y);

// 每隔指定的毫秒数就使用 setInterval() 调用 bounce() 方法。
// 保存返回值，以便我们能通过把它传递给 clearInterval() 方法来停止动画。
var intervalID = window.setInterval("bounce()", interval);

// 每隔指定的毫秒数 该函数就将窗口移动 (dx, dy) 个单位。
// 当窗口到达屏幕的边界处时，它将弹回。
function bounce() {
```

```

    // 如果用户关闭了窗口，就停止动画。
    if (win.closed) {
        clearInterval(intervalID);
        return;
    }

    // 如果到达右边界或左边界，就弹回。
    if ((x+dx > (screen.availWidth - w)) || (x+dx < 0)) dx = -dx;

    // 如果到达上边界或下边界，就弹回。
    if ((y+dy > (screen.availHeight - h)) || (y+dy < 0)) dy = -dy;

    // 更新窗口的当前位置。
    x += dx;
    y += dy;

    // 最后把窗口移到新位置。
    win.moveTo(x,y);
}
</script>

<!-- 点击该按钮来停止动画！ -->
<form>
<input type="button" value="Stop
        onclick="clearInterval(intervalID); win.close();" />
</form>

```

13.9 Location 对象

窗口的 `location` 属性引用的是 `Location` 对象，它代表该窗口中当前显示的文档的 URL。`Location` 对象的 `href` 属性是一个字符串，它包含完整的 URL 文本。这个对象的其他属性（如 `protocol`、`host`、`pathname` 和 `search` 等）则分别声明了 URL 的各个部分。

`Location` 对象的 `search` 属性比较有趣。它包含的是问号之后的那部分 URL，这部分通常是某种类型的查询字符串。一般说来，在 URL 中使用问号是一种在 URL 中嵌入参数的方法。虽然这些参数通常用于运行在服务器上的 CGI 脚本，但是没有任何理由阻止使用 JavaScript 的页面使用它们。例 13-5 展示了一个通用的函数 `getArgs()` 的定义，可以用这个函数将参数从 URL 的 `search` 属性中提取出来。它还说明了如何使用 `getArgs()` 方法来设置例 13-4 中出现的反弹窗口动画的参数的初始值。

例 13-5: 提取 URL 中的参数

```

/**
 * 该函数将把 URL 的查询串解析成 name-value 的参数列表。
 * 参数对之间用逗号分隔。它将把 name-value 对存储在一个对象的属性中，并返回该对象。
 *
function getArgs() {
    var args = new Object();
    var query = location.search.substring(1); // 获取查询串。
    var pairs = query.split(","); // 在逗号处断开。
    for(var i = 0; i < pairs.length; i++) {
        var pos = pairs[i].indexOf('='); // 查找 'name-value'。
        if (pos == -1) continue; // 如果没有找到，就跳过。
        var argname = pairs[i].substring(0,pos); // 提取 name。
        var value = pairs[i].substring(pos+1); // 提取 value。
        args[argname] = unescape(value); // 存为属性。
        // 在 JavaScript 1.5 中，使用 decodeURIComponent() 而不是 unescape()
    }
    return args; // 返回对象。
}

/**
 * 在前面弹出窗口的例子中，我们也可以使用 getArgs()。
 * 解析来自 URL 的可选动画参数。
 *
var args = getArgs(); // 获取参数。
if (args.x) x = parseInt(args.x); // 如果参数被定义了，
if (args.y) y = parseInt(args.y); // 覆盖它们的默认值
if (args.w) w = parseInt(args.w);
if (args.h) h = parseInt(args.h);
if (args.dx) dx = parseInt(args.dx);
if (args.dy) dy = parseInt(args.dy);
if (args.interval) interval = parseInt(args.interval);

```

Location 对象除了它的属性之外，自身也可以被用作一个原始字符串值。如果读取一个 Location 对象的值，得到的字符串和读取这个对象的 href 属性相同（因为 Location 对象具有一个相配的 toString() 方法）。更为有趣的是，可以将一个新的 URL 字符串赋给窗口的 location 属性。像这样把一个 URL 赋给 Location 对象具有一个重大的作用，即会引起浏览器装载并显示出那个 URL 所指的页面的内容。例如，可以使用如下的代码将一个 URL 赋给 location 属性：

```

// 如果用户使用不能显示 DHTML 内容的旧式浏览器，
// 可以重定向到只包含静态 HTML 的页面
if (parseInt(navigator.appVersion) < 4)
    location = "staticpage.html";

```

和你所设想的一样，使浏览器将一个指定的网页装载进窗口是一项非常重要的方法。当你考虑自己可以使用什么方法使浏览器显示一个新网页时，可以将一个 URL 赋给

窗口的 `location` 属性来实现这一目标。本章后面的例 13-6 中有设置 `location` 属性的示例。

虽然 `Location` 对象并没有一种方法可以直接将一个 URL 赋给窗口的 `location` 属性，但是它支持另外两种方法（JavaScript 1.1 中新添加的）。方法 `reload()` 会从 Web 服务器上再次装入当前显示的页面。方法 `replace()` 会装载并显示指定的 URL。但是为给定的 URL 调用这个方法和把一个 URL 赋给窗口的 `location` 属性不同。当调用 `replace()` 时，指定的 URL 就会替换浏览器历史列表中的当前 URL，而不是在历史列表中创建一个新条目。因此，如果使用方法 `replace()` 使一个新文档覆盖当前文档，**Back** 按钮就不能使用户返回原始文档，而通过将一个 URL 赋给窗口的 `location` 属性来装载新文档就可以做到这一点。对那些使用了框架并且显示多个临时页（可能是由 CGI 脚本生成的）的网站来说，`replace()` 通常比较有用。这样临时页面都不会存储在历史列表中，**Back** 按钮对用户就显得有用多了。

最后要注意的是，不要混淆 `Window` 对象的 `location` 属性和 `Document` 对象的 `location` 属性。前者引用一个 `Location` 对象，而后者只是一个只读字符串，并不具有 `Location` 对象的任何特性。`document.location` 与 `document.URL` 是同义的，后者在 JavaScript 1.1 中是该属性的首选名称（因为这样避免了潜在的混淆）。在大多数情况下，`document.location` 和 `location.href` 是相同的。但是，当存在服务器重定向时，`document.location` 包含的是已经装载的 URL，而 `location.href` 包含的则是原始请求的文档的 URL。

13.10 History 对象

`Window` 对象的 `history` 属性引用的是该窗口的 `History` 对象。`History` 对象最初是用来把窗口的浏览历史构造成近来访问过的 URL 的数组。但这种设计非常拙劣，出于重要的安全性和隐私性的原因，使脚本能够访问用户以前访问过的站点列表绝不合适。因此，脚本不能真正访问 `History` 对象的数组元素（除非用户在 Netscape 4 和其后的版本中对有签名的脚本进行了授权）。`History` 对象的 `length` 属性可以被访问，但是它不能提供任何有用信息。

尽管 `History` 对象的数组元素不能被访问，但它支持三种方法（所有浏览器版本中的常规、无签名脚本都可以调用它们）。方法 `back()` 和 `forward()` 可以在窗口（或框

架) 的浏览历史中前后移动, 用前面浏览过的文档替换当前显示的文档, 这与用户点击浏览器的 **Back** 和 **Forward** 按钮的作用相同。第三个方法 `go()` 有一个整数参数, 可以在历史列表中向前或向后跳过多个页。遗憾的是, 方法 `go()` 在 Netscape 2 和 3 中有 bug, 而且与 Internet Explorer 3 中的行为不兼容, 所以在第四代之前的浏览器中最好不要使用它。

例 13-6 说明了如何使用 History 对象的方法 `back()` 和 `forward()` 以及如何用 Location 对象给一个带框架的站点添加导航栏。图 13-3 展示了一个导航栏。注意, 这个例子使用了多框架的 JavaScript, 我们将在不久之后讨论这一点。它还包含一个简单的 HTML 表单, 并且使用 JavaScript 来读写这个表单中的值。我们将在第十五章中对此进行详细介绍。

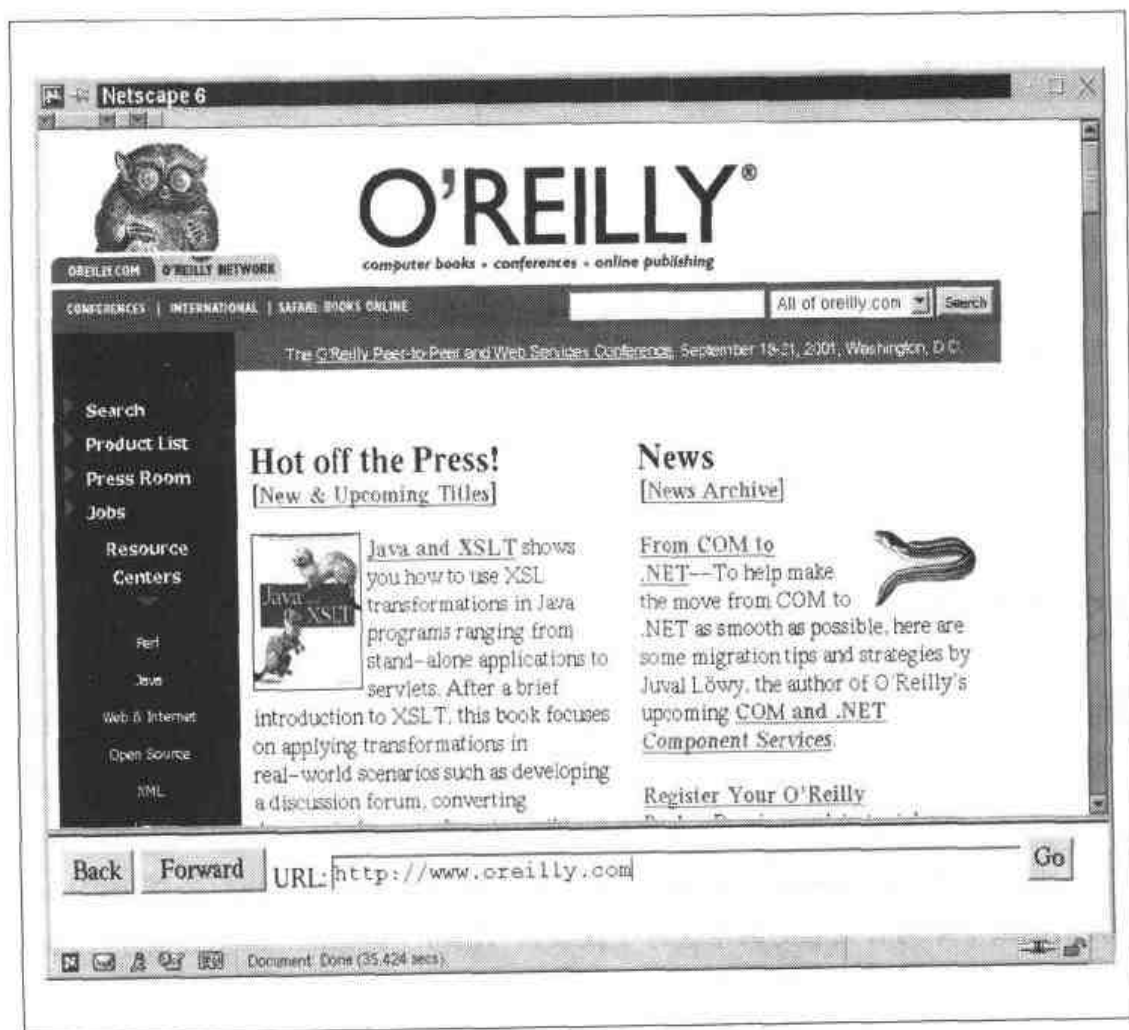


图 13-3: 导航栏

例 13-6: 使用 History 对象和 Location 对象的导航栏

<!-- 这个文件实现了一个导航栏。它被放置在窗口底部的一个框架中。

把它加入一个框架集的代码如下:

```
<frameset rows "*,75"> <iframe src="about:blank">
  <frame src="navigation.html">
</frameset>

-->

<script>
// 该函数将由导航栏的 Back 按钮调用。
function go_back()
{
  // 首先, 将表单中的 URL 输入框清空。
  document.navbar.url.value = "";

  // 然后使用主框架的 History 对象返回。
  parent.frames[0].history.back();

  // 稍等片刻, 然后把表单中的 URL 输入框更新为主框架的 location.href 属性。
  // 要使 location.href 属性配合一致, 等待片刻是必需的。
  setTimeout("document.navbar.url.value = parent.frames[0].location.href;",
    1000);
}

// 该函数将由导航栏的 Forward 按钮调用。
// 它与上一个函数的运行过程相似。
function go_forward()
{
  document.navbar.url.value = "";
  parent.frames[0].history.forward();
  setTimeout("document.navbar.url.value = parent.frames[0].location.href;",
    1000);
}

// 该函数将由导航栏的 Go 按钮调用。
// 此外, 提交表单时 (即当用户按了 Return 键时) 也会调用它。
function go_to()
{
  // 把主框架的 location 属性设置为用户输入的 URL。
  parent.frames[0].location = document.navbar.url.value;
}
</script>

<!-- 以下是表单, 它的事件处理程序将调用以上的函数。 -->
<form name="navbar" onsubmit="go_to(); return false;">
  <input type="button" value="Back" onclick="go_back();">
  <input type="button" value="Forward" onclick="go_forward();">
  URL:
  <input type="text" name="url" size="50">
  <input type="button" value="Go" onclick="go_to();" />
</form>
```

13.11 多窗口和多框架

迄今为止，我们见到过的大部分客户端JavaScript的例子都只涉及一个窗口或者一个框架。在实际应用中，大多数有趣的JavaScript应用程序都包含多个窗口或者多个框架。我们知道，窗口中的框架也是由Window对象表示的，JavaScript并不区别窗口和框架。在许多有趣的应用程序中，都有运行在各个独立的窗口中的JavaScript代码。下面一节将解释每个窗口中的JavaScript代码如何与其他窗口以及这些窗口中运行的脚本交互和协作。

13.11.1 框架之间的关系

我们已经知道，Window对象的方法`open()`返回代表新创建的窗口的Window对象。而且这个新窗口具有`opener`属性，该属性可以打开它的原始窗口。这样，两个窗口就可以互相引用，彼此都可以读取对方的属性或是调用对方的方法。框架也是这样的。一个窗口中的任何框架都可以使用Window对象的属性`frames`、`parent`和`top`属性来引用其他的框架。

每个窗口都有`frames`属性。这个属性引用一个Window对象的数组，其中每个元素代表的是这个窗口中包含的框架（如果一个窗口没有任何框架，那么`frames[]`数组就是空的，`frames.length`的值为0）。这样，窗口（或框架）就可以使用`frames[0]`来引用它的第一个子框架，使用`frames[1]`来引用第二个子框架，以此类推。同样，运行在一个窗口中的JavaScript代码可以引用它的第二个框架的第三个子框架，代码如下：

```
frames[1].frames[2]
```

每个窗口还含有一个`parent`属性，它引用包含这个窗口的Window对象。这样，窗口中的第一个框架就可以引用它的兄弟框架（即窗口中的第二个框架），代码如下：

```
parent.frames[1]
```

如果一个窗口是顶级窗口，而不是框架，那么`parent`属性引用的就是这个窗口本身：

```
parent == self; // 用于所有顶层窗口
```

如果一个框架包含在另一个框架中，而后者又包含在顶级窗口中，那么该框架就可以使用`parent.parent`来引用顶级窗口。`top`属性是一个通用的快捷方式，无论一

个框架被嵌套了几层，它的 `top` 属性引用的都是包含它的顶级窗口。如果一个 `Window` 对象代表的是一个顶级窗口，那么它的 `top` 属性引用的就是窗口自身。对于那些是顶级窗口的直接子窗口，`top` 属性就等价于 `parent` 属性。

框架通常是由 `<frameset>` 和 `<frame>` 标记创建的。但在 HTML 4（由 IE 4 及其后的版本和 Netscape 6 及其后的版本实现）中，`<iframe>` 标记也可以用来在文档中创建“内联框架”。就 JavaScript 来说，`<iframe>` 创建的框架与 `<frameset>` 和 `<frame>` 创建的框架一样。这里讨论的所有内容都适用于两种框架。

图13-4说明了框架之间的关系，此外它还展示了在每个框架中运行的代码如何使用属性 `frames`、`parent` 和 `top` 来引用其他框架。该图展示了具有两个框架的浏览器窗口，一个框架位于另一个框架之上。第二个框架（底部较大的那个）自身还含有三个并排放置的子框架。

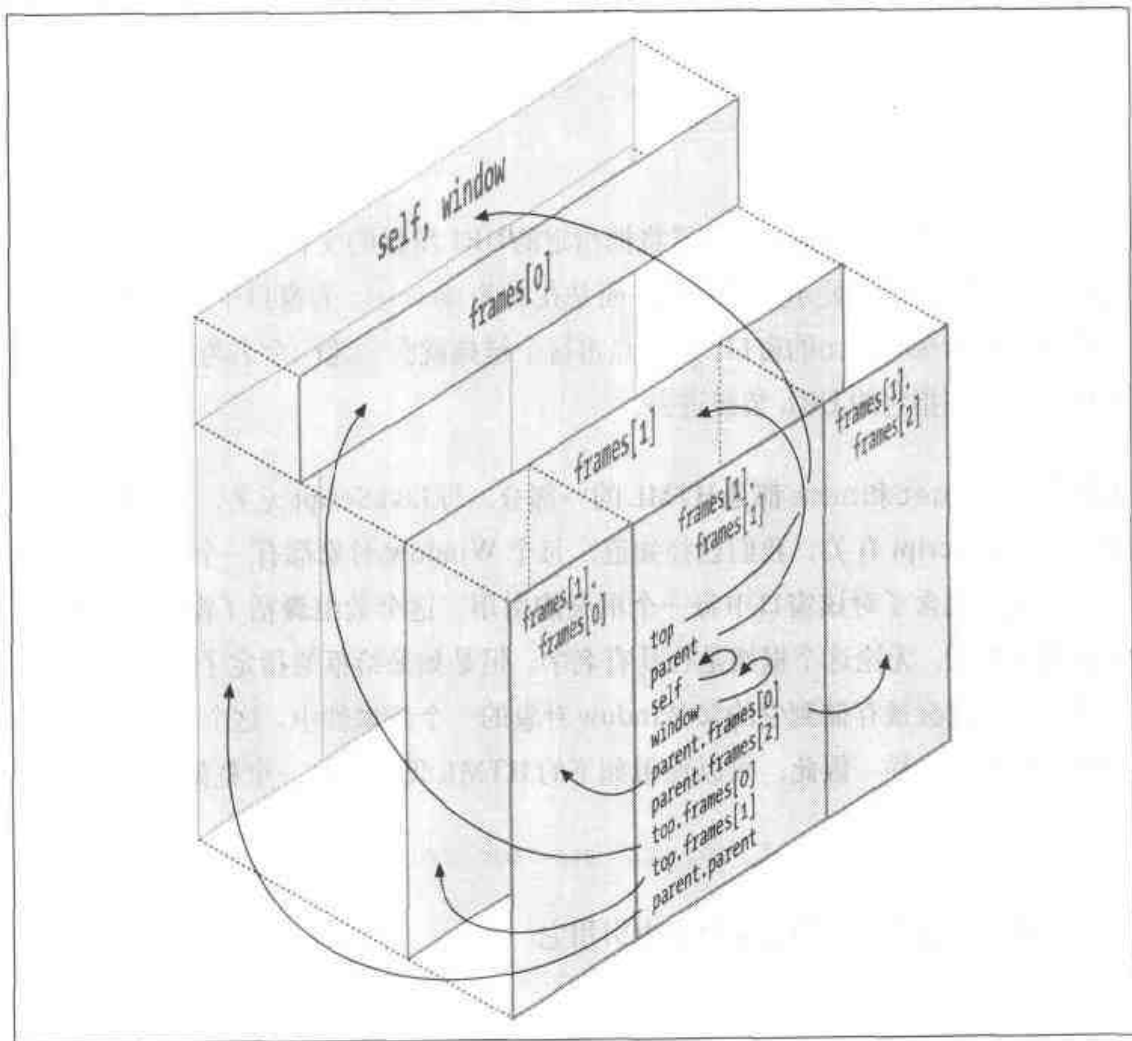


图 13-4：框架之间的关系

理解了框架之间的关系, 就可以重新研究例 13-6。这次要格外注意第二个框架引用第一个框架的 `history` 属性和 `location` 属性的代码 (该代码运行在第二个框架中)。

13.11.2 窗口和框架的名字

前面讨论过 `open()` 方法的第二个参数是新创建的窗口的名字。当用 HTML 标记 `<frame>` 创建框架时, 可以使用性质 `name` 为这个框架指定一个名字。给窗口或者框架指定名字的一个重要原因是那个名字可以用作标记 `<a>`、`<map>` 或 `<form>` 的性质 `target` 的值, 这样就可以告诉浏览器把激活链接、点击图像或者提交表单的结果显示在哪里。

例如, 假定有两个窗口, 一个名为 `table_of_contents`, 另一个名为 `mainwin`, 在名为 `table_of_contents` 的窗口中有如下 HTML 代码:

```
<a href='chapter01.html' target="mainwin">  
  Chapter 1, Introduction  
</a>
```

当用户点击这个超链接时, 浏览器将把指定的 URL 所指的文档装载进来, 但不是具有该链接的窗口中显示这个 URL, 而是在名为 `mainwin` 的窗口中将它显示出来。如果没有名为 `mainwin` 的窗口, 那么点击这个链接就会创建一个名为 `mainwin` 的新窗口, 并且将指定的 URL 装载进去。

虽然性质 `target` 和 `name` 都是 HTML 的一部分, 与 JavaScript 无关, 但是给框架命名却与 JavaScript 有关。我们已经知道, 每个 `Window` 对象都有一个 `frames[]` 数组, 该数组包含了对该窗口中每一个框架的引用。这个数组囊括了窗口 (或框架) 中的所有框架, 无论这个框架是否具有名字。但是如果给框架指定了名字, 对那个框架的引用就会被存储到它的父 `Window` 对象的一个新属性中。这个新属性的名字和框架的名字一样。因此, 可以使用如下的 HTML 代码创建一个框架:

```
<frame name="table_of_contents" src='toc.html'>
```

这样就可以从这个框架的兄弟框架中引用它:

```
parent.table_of_contents
```

比起那些使用硬编码的数组下标来处理未命名的框架的代码来说,这种代码是既容易读,又容易理解:

```
parent.frames[1]
```

任何一个 Window 对象的 name 属性都含有那个窗口的名字。在 JavaScript 1.0 中,这个属性是只读的。但在 JavaScript 1.1 和其后的版本中,就可以对该属性进行设置,从而改变一个窗口或框架的名字。这样做是为了设置初始浏览器窗口的名字。在浏览器启动时,初始窗口并没有名字,因此 target 性质就不能使用它。但如果设置了这个窗口的 name 性质,就可以在 target 性质中使用它了。

13.11.3 交互窗口中的 JavaScript

回忆一下,我们在第十二章中学过,Window 对象是客户端 JavaScript 代码的全局对象,该窗口是作为它所包含的所有 JavaScript 代码的执行环境。对框架来说,情况也是这样的,每个框架都是一个独立的 JavaScript 执行环境。由于每个 Window 对象都是它自己的全局对象,所以每个窗口都定义了自己的名字空间以及自己的一套全局变量。如果从多框架或者多窗口的角度来观察,全局变量并不见得都是全局性的。

虽然每个窗口和框架都定义了独立的 JavaScript 执行环境,但是这并不意味着在一个窗口中运行的 JavaScript 代码与另一个窗口中运行的 JavaScript 代码是完全隔离的。在两个不同框架中运行的 JavaScript 代码,其作用域链头部的 Window 对象是不同的。但是这两个框架中的代码都是在同样的 JavaScript 环境中由同一个 JavaScript 解释器执行的。我们已经知道,框架可以使用属性 frames、parent 或 top 来引用另一个框架。因此,虽然不同框架中的 JavaScript 代码是在不同的作用域链中执行的,但是这并不妨碍一个框架中的代码引用并使用另一个框架中的代码定义的变量和函数。

例如,假定框架 A 中的代码定义了一个变量 i:

```
var i = 3;
```

这个变量只是全局对象的一个属性,也是 Window 对象的一个属性。框架 A 中的代码可以使用下列任何一个表达式来明确地引用这个变量:


```
window.i  
self.i
```

假定框架 A 有一个兄弟框架 B，它想设置框架 A 中的代码所定义的变量 `i`。如果框架 B 只是设置一个变量 `i`，那么它只是给自己的 Window 对象创建了一个新的属性。因此，它必须明确地引用它的兄弟框架的属性 `i`，代码如下：

```
parent.frames[0].i = 4;
```

我们知道，定义函数的关键字 `function` 可以声明一个变量，就像关键字 `var` 所做的那样。如果框架 A 中的 JavaScript 代码声明了一个函数 `f`，那么 `f` 只在框架 A 中有定义。框架 A 中的代码可以使用如下代码来调用这个函数：

```
f();
```

但是框架 B 中的代码必须将 `f` 作为框架 A 的 Window 对象的一个属性来引用：

```
parent.frames[0].f();
```

如果框架 B 中的代码需要频繁地使用这个函数，可以将这个函数赋给框架 B 的一个变量，这样再引用这个函数就方便多了：

```
var f = parent.frames[0].f;
```

现在框架 B 中的代码就可以像框架 A 中的代码那样调用函数 `f()` 了。

当采用这种方式在框架或窗口间共享函数时，谨记词法作用域的规则非常重要。函数在定义它的作用域中执行，而不是在调用它的作用域中执行。就上面那个例子来说，如果函数 `f` 引用了全局变量，那么将在框架 A 的属性中查找这些变量，即使函数是由框架 B 调用的。

如果你没有注意这一点，程序执行的结果可能就不是你想要的。例如，假定你在多框架文档的 `<head>` 部分定义了如下函数，认为它有助于调试：

```
function debug(msg) {  
    alert('Debugging message from frame: ' + name + '\n' + msg);  
}
```

任何一个框架中的 JavaScript 代码都可以使用 `top.debug()` 来引用这个函数。但只要调用了这个函数，它都会在定义该函数的顶级窗口的环境中查找变量 `name`，而不

是在调用该函数的框架的环境中进行查找。这样一来，调试消息总是显示顶级窗口的名字，而不是像希望的那样显示发送消息的框架的名字。

我们知道构造函数也是函数的一种，所以当用构造函数和一个相关的原型对象来定义一个对象的类时，那个类只是为一个窗口定义的。回忆一下我们在第八章中定义的类 `Complex`，考虑如下的多框架 HTML 文档：

```
<head>
<script src= "Complex.js" > </script>
< body>
<frameset rows="10%,10%">
  <frame name= "frame1" src= "frame1.html">
  <frame name= "frame2" src= "frame2.html">
</frameset>
```

文件 `frame1.html` 和 `frame2.html` 中的 JavaScript 代码不能使用如下的表达式来创建 `Complex` 对象：

```
var c = new Complex(1,2);
```

在两个框架中都不能运行

相反，这两个文件中的代码必须明确地引用构造函数：

```
var c = new top.Complex(3,4);
```

另外，每个框架中的代码还可以定义自己的变量来引用构造函数，这样就更为简便了：

```
var Complex = top.Complex;
var c = new Complex(1,2);
```

和用户定义的构造函数不同，预定义的构造函数会在所有窗口中自动地进行预定义。但是要注意，每个窗口都有一个构造函数的独立副本和一个构造函数原型对象的独立副本。例如，每个窗口都有自己的 `String()` 构造函数的部分和 `String.prototype` 对象的副本。因此，如果编写一个操作 JavaScript 字符串的新方法，并且通过把它赋给 `String.prototype` 对象而使其成为 `String` 类的一个方法，那么该窗口中的所有字符串就都可以使用这个新方法。但是别的窗口中定义的字符串不能使用这个方法。注意，至于哪个窗口具有对字符串的引用并不重要，重要的是那个字符串是由哪个窗口创建的。

13.11.4 例子：有色框架

例 13-7 创建了一个框架集合，其中包含九个框架，使用的方法是我们在本章中已经讨论过的。这个框架集合的 `<head>` 部分含有一个 `<script>` 元素，其中定义了一个名为 `setcolor()` 的 JavaScript 函数。标记 `<frameset>` 的事件处理程序 `onload` 为每个框架调用了一次函数 `setcolor()`。

`setcolor()` 的参数是一个 Window 对象。它随机生成一种颜色，将这种颜色用于 `Document.write()` 方法，以创建一个只有背景颜色的空文档。最后 `setcolor()` 调用方法 `setTimeout()` 对自己进行调度，每秒调用它一次。对方法 `setTimeout()` 的调用是这个例子中最为有趣的一部分。尤其要注意如何使用 Window 对象的 `parent` 和 `name` 属性。

例 13-7：框架着色动画

```
<head>
<title>Colored Frames</title>
<script>
function setcolor(w) {
    // 生成一种随机颜色。
    var r = Math.floor(Math.random() * 256).toString(16);
    var g = Math.floor(Math.random() * 256).toString(16);
    var b = Math.floor(Math.random() * 256).toString(16);
    var colorString = 'r' + r + 'g' + g + 'b';

    // 把框架的背景设置为该随机颜色。
    w.document.write( '<body bgco or=' + colorString + " "></body>' );
    w.document.close();

    // 安排在一秒后再次调用该方法。
    // 由于调用框架的 setTimeout() 方法时，其中的字符串将在框架的环境中执行，
    // 所以必须给顶级窗口的属性加上前缀 "parent"。
    w.setTimeout('parent.setcolor(parent.' + w.name + ')', 1000);

    // 我们还可以用更简单的方式来实现这一点。
    // setTimeout( setcolor( ' + w.name + ')', 1000);
}
</script>
</head>
<frameset rows="33%,33%,34%" cols="33%,33%,34%"
  onload="for(var i = 0; i < 9; i++) setcolor(frames[i]);">
  <frame name="f1" src="javascript:''"><frame name="f2" src="javascript:''">
  <frame name="f3" src="javascript:''"><frame name="f4" src="javascript:''">
  <frame name="f5" src="javascript:''"><frame name="f6" src="javascript:''">
  <frame name="f7" src="javascript:''"><frame name="f8" src="javascript:''">
  <frame name="f9" src="javascript:''">
</frameset>
```

第十四章

Document 对象

每个 Window 对象都有 document 属性。该属性引用表示在窗口中显示的 HTML 文档的 Document 对象。Document 对象可能是客户端 JavaScript 中最常用的对象。我们在本书中已经见过几个使用 Document 对象的 write() 方法在文档被解析时将动态内容插入文档。除了常用的 write() 方法之外，Document 对象还定义了提供文档整体信息的属性，如文档的 URL、最后修改日期、文档要链接到的 URL、显示文档的颜色，等等。

客户端 JavaScript 可以把静态 HTML 文档转换成交互式的程序，因为 Document 对象给了它交互访问静态文档的内容的能力。除了提供文档整体信息的属性外，Document 对象还有大量的重要属性，这些属性提供文档内容的信息。例如，forms[] 数组存放的 Form 对象表示文档中的所有表单。images[] 数组和 applets[] 数组存放的是表示文档中的图像和小程序的对象。这些数组和它们存放的对象使客户端 JavaScript 程序创造了各种可能性。本章会详细介绍这一点。

本章介绍了 Document 对象的核心特性，几乎所有启用 JavaScript 的浏览器都能实现 Document 对象。较新的浏览器（如 IE 4 及其后的版本和 Netscape 6 及其后的版本）实现了完整的文档对象模型（DOM），它使 JavaScript 能完全访问和控制文档的所有内容。第十七章将介绍这些高级 DOM 特性。

14.1 Document 对象概览

为了说明Document对象的作用域和重要性,本章先简要介绍Document对象的方法和属性。然后解释了其他重要的资料,它们对理解本章其余部分的内容非常重要。

14.1.1 Document 对象的方法

Document对象定义了四个关键方法。其中的write()方法我们已经见过,其他三个方法如下:

close()

关闭或结束open()方法打开的文档。

open()

产生一个新文档,擦掉已有的文档内容。

write()

把文本附加到当前打开的文档。

writeln()

把文本输出到当前打开的文档,并附加一个换行符。

14.1.2 Document 对象的属性

Document对象定义了如下的属性:

alinkColor, linkColor, vlinkColor

这些属性描述了超级链接的颜色。linkColor是未被访问过的链接的正常颜色。vlinkColor是被访问过的链接的正常颜色。alinkColor是被激活的链接(即用户点击了该链接)的颜色。这些属性对应于标记<body>的属性alink、link和vlink。

anchors[]

Anchor对象的一个数组,该对象代表文档中的锚。

applets[]

Applet对象的一个数组,该对象代表文档中的Java小程序。

bgColor, fgColor

文档的背景颜色和前景（即文本）颜色。这两个属性对应于标记 <body> 的性质 bgcolor 和 text。

cookie

一个特殊属性，允许 JavaScript 程序读写 HTTP cookie。有关细节请参阅第十六章。

domain

该属性使处于同一 Internet 域中的相互信任的 Web 服务器在网页间交互时能协同地放松某项安全性限制。详情请参阅第二十一章。

forms[]

Form 对象的一个数组，该对象代表文档中的 <form> 元素。

images[]

Image 对象的一个数组，该对象代表文档中的 元素。

lastModified

一个字符串，包含文档的修改日期。

links[]

Link 对象的一个数组，该对象代表文档中的超文本链接的 Link 对象。

location

等价于属性 URL，其使用是遭反对的。

referrer

文档的 URL，包含把浏览器带到当前文档的链接（如果存在这样的链接）。

title

位于文档的标记 <title> 和 </title> 之间的文本。

URL

一个字符串。声明了装载文档的 URL。除非发生了服务器重定向，否则该属性的值与 Window 对象的属性 location.href 相同。

14.1.3 Document 对象和标准

Document 对象和它呈现给 JavaScript 程序的元素集合（如表单、图象和链接）构成了文档对象模型。历史上，不同的浏览器销售商实现了不同的 DOM，这使 JavaScript 程序设计者很难转而去使用销售商特定的 DOM 高级特性。幸运的是，W3C (<http://www.w3.org>) 标准化了 DOM，发行了该标准的两个版本，分别为 1 级 DOM 和 2 级 DOM。近来的浏览器（如 Netscape 6 及其后的版本和 IE 5 及其后的版本）实现了这两个标准的大部分内容。第十七章有详细说明。

本章介绍的 DOM 早于 W3C 的标准。但它是个通用的实现，所以它是真正的标准，通常被称为 0 级 DOM。在任何启用 JavaScript（除了像 Netscape 2 这样的旧版本）的浏览器中都可以使用本章介绍的技术。此外，前面列出的 Document 对象的属性和方法已经成为 1 级 DOM 的一部分，所以将来的浏览器一定会支持它们。

理解 W3C DOM 标准的重要一点是，它是 XML 和 HTML 文档通用的文档对象模型。在这个标准中，Document 对象提供了两种文档通用的功能。HTML 专用的功能由 HTMLDocument 子类提供。本章介绍的所有 Document 属性和方法是 HTML 专用的，在客户端参考手册部分的 Document 条目中可以找到更多细节。此外，在 DOM 参考手册部分的 Document 和 HTMLDocument 条目下还可以找到相关的信息。

14.1.4 Document 对象的命名

在我们开始讨论 Document 对象和它呈现的各种对象之前，先介绍一个通用的规则，记住它非常有用。一个 HTML 文档中的每个 <form> 元素都会在 Document 对象的 forms[] 数组中创建一个带编码的元素。同样，每个 元素也会创建一个 images[] 数组的元素。这一规则还适用于标记 <a> 和 <applet>，它们分别定义了 links[] 数组和 applets[] 数组的元素。

除了这些数组之外，如果与 Form 对象、Image 对象或 Applet 对象对应的 HTML 标记中设定了 name 性质，就可以用名字来引用这些对象。出现 name 性质时，它的值将被用作 Document 对象的属性名，用来引用相应的对象。例如，假定 HTML 文档含有如下表单：

```
<form name="f1">  
<input type="button" value="Push Me">
```

```
</form>
```

假定`<form>`是文档中的第一个元素,那么你的JavaScript代码就可以使用下面两个表达式中的任何一个来引用生成的Form对象:

```
document.forms[0]    // 用表单在文档中的位置来引用它  
document.f1          // 用名字引用表单
```

事实上,设置`<form>`的`name`性质还可以将Form对象作为`forms[]`数组的命名属性来访问,所以可以用下面的表达式来引用表单:

```
document.form=f1      // 使用属性语法引用表单  
document.forms["f1"]  // 使用数组语法引用表单
```

这个规则同样适用于图像和小程序,用HTML中的`name`性质便可以通过名字引用JavaScript代码中的对象。

给经常使用的Document对象命名很方便,这样在脚本中引用它就非常容易了。在本章后面的小节中,我们将大量使用这一方法。

14.1.5 Document 对象和事件处理程序

要使一个HTML文档具有交互性,这个文档及其中包含的元素必须响应用户事件。我们在第十二章中简要地讨论过事件和事件处理程序,而且已经见到了几个使用简单的事件处理程序的例子。在本章中,我们将会看到更多事件处理程序的例子,因为它们是使用Document对象的关键所在。

遗憾的是,事件和事件处理程序的完整讨论在第十九章进行。目前我们只要记住事件处理程序是由HTML元素的性质`onclick`或`onmouseover`定义的就足够了。这些性质的值都是JavaScript代码串。当指定的事件发生在HTML元素上时,这些代码就会被执行。

此外,有一种定义事件处理程序的方法,在本章和后面的章节中我们会见到。在本章我们看到,像Form和Image这样的对象都有与标记`<form>`和``的HTML性质相匹配的JavaScript属性。例如,HTML标记``有`src`性质和`width`性质,JavaScript的Image对象就有相应的`src`属性和`width`属性。事件处理程序也是这样。例如,HTML标记`<a>`支持`onclick`事件处理程序,表示超链接的JavaScript

对象 Link 有相应的 onclick 属性。另一个例子是 <form> 元素的 onsubmit 性质。在 JavaScript 中, Form 对象有相应的 onsubmit 属性。记住, HTML 不区分大小写, 它的性质可以用大写、小写或大小写混合的形式, 而 JavaScript 的所有事件处理程序属性则都必须用小写形式。

在 HTML 中, 通过把 JavaScript 代码赋予一个事件处理程序性质可以定义事件处理程序。但在 JavaScript 中, 则通过把函数赋予一个事件处理程序属性来定义它们。考虑下面的 <form> 元素和它的 onsubmit 事件处理程序:

```
<form name= myform onsubmit='return validateform();'>... form;
```

在 JavaScript 中, 除了使用调用函数并返回它的值的 JavaScript 代码串, 还可以直接把函数赋予事件处理程序属性, 如下所示:

```
document.myform.onsubmit = validateform;
```

注意, 函数名后没有括号, 这是因为此处我们不想调用函数, 只想把一个引用赋予它。另一个例子如下, 考虑 <a> 标记和它的 onmouseover 事件处理程序:

```
<a href= help.html" onmouseover= status='Get Help Now!';'helpa/a>
```

如果我们碰巧知道这个 <a> 标记是文档的第一个链接元素, 可以用 document.links[0] 引用相应的 Link 对象, 采用如下方式设置事件处理程序:

```
document.links[0].onmouseover = function() { status = 'Get Help Now!'; }
```

第十九章详细讨论了如何用这种方法给事件处理程序赋值。

14.2 动态生成的文档

Document 对象 (一般说来, 可能是客户端 JavaScript) 的一个最重要的特性是方法 write(), 它使你可以从 JavaScript 程序中动态地生成网页的内容。使用这种方法有两种方式。第一种方式 (也是最简单的方式) 是, 在脚本中使用它, 把 HTML 输入到当前正在被解析的文档中。第十二章讨论过这种方法。考虑如下的代码, 它使用 write() 方法把当前日期和文档的最后修改日期添加到另一个静态 HTML 文档:

```
<script>
```

```
var today = new Date();  
document.write("<p>Document accessed on: " + today.toString());  
document.write("<br>Document modified on: " + document.lastModified);  
</script>
```

以这种方式使用 write() 方法是极其常用的 JavaScript 程序设计技术，你会在许多脚本中见到这种方式。

但要注意，只能在当前文档正在解析时使用 write() 方法向其输出 HTML 代码。简而言之，就是只能在标记 <script> 中调用方法 document.write()，因为这些脚本的执行是文档解析过程的一部分。如果是从一个事件处理程序中调用 document.write()，而且一旦文档被解析，该处理程序被调用，那么结果将会覆盖当前文档（包括它的事件处理程序），而不是将文本添加到其中。当我们检测使用 write() 方法的第二种方式时会明白其中的原因。

除了在当前文档被解析时动态地给它添加内容，write() 方法还可以与 open() 方法及 close() Document 方法一起在窗口或框架中创建全新文档。虽然不能有效地从事件处理程序中改写当前文档，但是可以把文档写入另一个窗口或框架，这在多窗口或多框架的网站中非常有用。例如，某个多框架站点的 JavaScript 代码可以用如下代码在另一个框架中显示消息：

```
<script>  
// 开始一个新文档，擦掉 frames[0] 中已有的内容  
parent.frames[0].document.open();  
// 给新文档添加内容  
parent.frames[0].document.write("<hr>Hello from your sibling frame!<hr>");  
// 添加完内容后，关闭该文档  
parent.frames[0].document.close();  
</script>
```

要创建新文档，首先需要调用 Document 对象的 open() 方法，然后多次调用 write() 方法在文档中写入内容，最后调用 Document 对象的方法 close() 以说明创建过程结束了。最后一步非常重要，如果忘记了关闭文档，浏览器就不能制止它显示的文档装载动画。而且，浏览器可以将你写入的 HTML 缓存起来，这样在调用方法 close() 明确地结束文档之前，缓存输出不会显示出来。

与必需的 close() 调用不同，open() 方法的调用则是可选的。如果调用了一个已经关闭了的文档的 write() 方法，JavaScript 会隐式地打开一个新的 HTML 文档，就像已经调用过 open() 方法一样。这就解释了在同一文档中从事件处理程序调用

`document.write()`时会发生什么,即JavaScript会打开一个新文档。但是在这个过程中,当前文档(它的内容,包括脚本、事件处理程序)就被丢弃了。这不是你想做的,而且这样还可能使早期的浏览器(如Netscape 2)崩溃。作为一条经验,一个文档绝不应该从事件处理程序中调用它自己的`write()`方法。

方法`write()`还有两点需要注意。第一,许多人不知道`write()`可以具有多个参数。在传递给它多个参数时,这些参数将被依次写入文档,就像它们已经连接在一起一样。因此,下面的代码:

```
document.write("Hello, " + username + " Welcome to my home page!");
```

与下面的代码等价:

```
var greeting = "Hello, ";  
var welcome = " Welcome to my home page!";  
document.write(greeting, username, welcome);
```

第二点要注意的是,Document对象还支持`writeln()`方法,它几乎与`write()`方法完全相同,只不过会在输出的参数之后附加一个换行符。由于HTML忽略换行,所以这个换行符不会产生什么不同,不过在和非HTML文档一起使用时,使用`writeln()`方法比较方便,我们不久就会看到这样的例子。

例14-1展示的是如何使用Window对象的`open()`方法和Document对象的方法来创建一个复杂的对话框。这个例子将为窗口注册一个事件处理程序`onerror`,当发生JavaScript错误时调用该函数。错误处理函数将创建一个新窗口,并且用Document对象的方法在这个窗口中创建一个HTML表单。用户可以在这个表单中看到错误的详细信息,并且还可以将一份bug报告用email发给这段JavaScript代码的作者。

图14-1展示了示例窗口。回忆一下第十三章中对`onerror`错误处理函数的讨论,Netscape 6不能给它传递正确的参数。因此,Netscape 6中的输出与这里的显示不同。

例14-1: 对话框窗口的动态创建

```
<script>  
// 一个变量,我们用它来确保创建的每个错误窗口是唯一的。  
var error_count = 0;
```

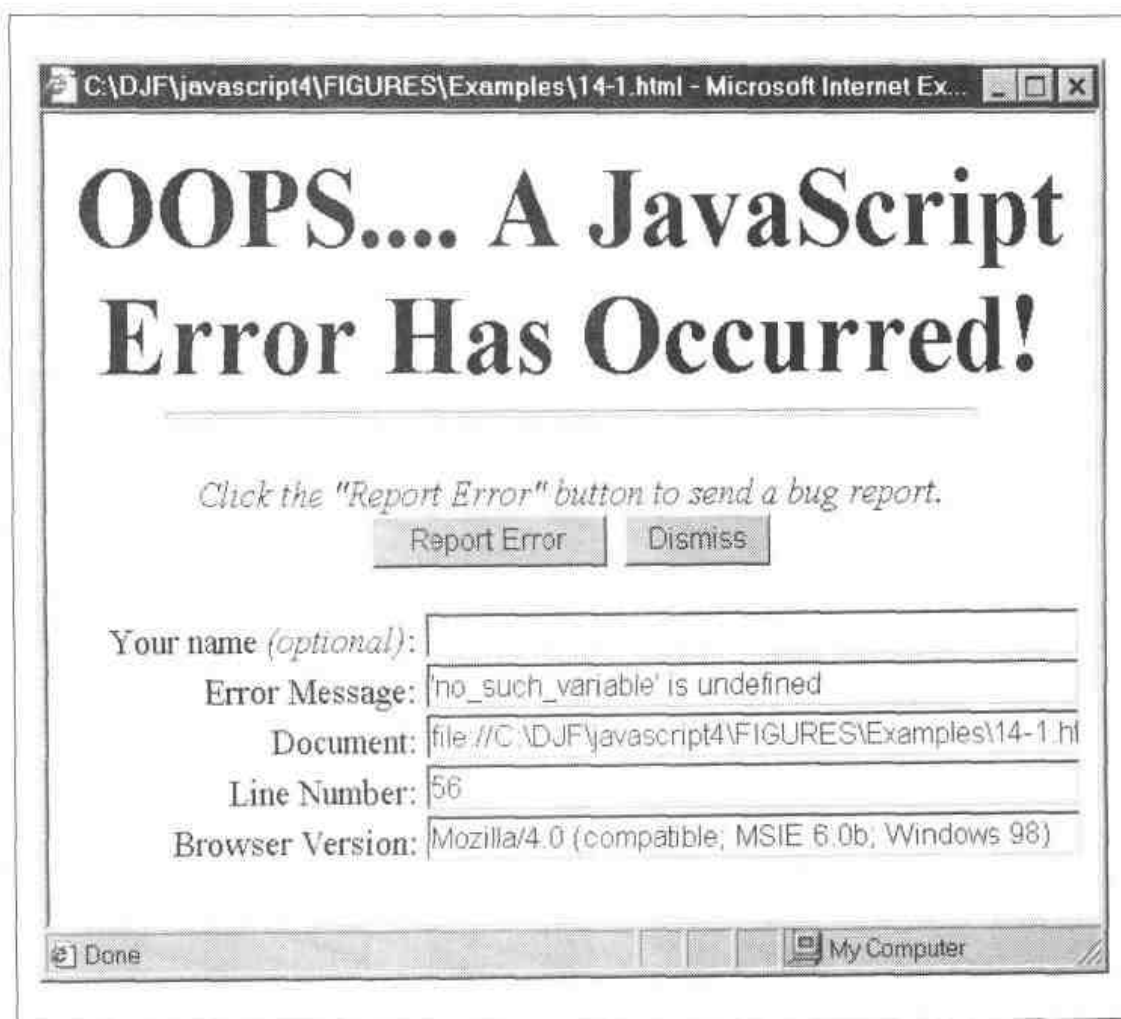


图 14-1: 将浏览器窗口作为一个对话框使用

```
// 把该变量设置为你的 email 地址。
var email = "myname@mydomain.com";

// 定义错误处理程序。它将生成一个 HTML 表单以使用户能将错误报告给作者。
function report_error(msg, url, line)
{
    var w = window.open("", // URL (未定义)。
                        "error"+error_count++, // Name (迫使它是唯一的)。
                        "resizable,status,width=625,height=400"); // 特征。
    // 获取新窗口的 Document 对象。
    var d = w.document;

    // 在新窗口中输出一个 HTML 文档，其中包括一个表单。
    // 注意我们省略了对 document.open() 方法的可选调用。
    d.write('<div align="center">');
    d.write('<font size="7" face="helvetica"><b>');
    d.write('OOPS.... A JavaScript Error Has Occurred!');
    d.write('</b></font><br><hr size="4" width="80%">');
```

```

d.write('<form action="mailto:' + email + '&#32;method=post );
d.write('&#32;enctype="text/plain">');
d.write(' <font size="3">');
d.write('</i>Click the "Report Error" button to send a bug report.</i><br> ');
d.write(' <input type="submit" value="Report Error">&#32;&#32; ');
d.write(' <input type="button" value="Dismiss" onclick="self.close(); > ');
d.write('< /div><div align="right">');
d.write('<br>Your name <i>(optional)</i>: ');
d.write(' <input size="42" name="name" value=" ">');
d.write('<br>Error Message: ');
d.write(' <input size="42" name="message" value=" ' + msg + ' ">');
d.write('<br>Document: <input size="42" name="url" value=" ' + url + ' ">');
d.write('<br>Line Number: <input size="42" name="line" value=" ' + line + ' ">');
d.write('<br>Browser Version: ');
d.write(' <input size="42" name="version" value=" ' + navigator.userAgent + ' ">');
d.write(' < /div></font>');
d.write(' < /form>');
// 写完文档后记得关闭它。
d.close());

// 由该错误处理程序返回 true, 以便 JavaScript 不再显示它自己的错误对话框
return true;
}

// 在事件处理程序发生作用之前, 我们必须把它注册为一个特殊的窗口。
self.onerror = report_error;
</script>

<script>
// 以下的代码将有意引发一个错误, 以便测试。
alert(no_such_variable);
</script>

```

14.2.1 非 HTML 文档

如果调用 Document 方法 open() 时没有使用参数, 那么它将打开一个新的 HTML 文档。但是要记住, 除了 HTML 文本之外, Web 浏览器还可以显示很多其他格式的数据。当你想动态创建并显示一个使用其他数据格式的文档时, 就要给方法 open() 传递一个参数, 指定你想使用的 MIME 类型 (注 1)。

HTML 的 MIME 类型是 text/html。除了 HTML 之外, 最常用的格式是纯文本, 它的 MIME 类型是 text/plain。如果希望在 write() 方法输出文本中使用换行符,

注 1: W3C DOM 没有标准化 open() 方法的这一参数。在 IE 4 及其后的版本和 Netscape 3 和 4 中可以使用它。令人吃惊的是, 在 Netscape 6 中不能使用这种方法, 它只支持 HTML 文档。

空格和制表符以进行格式化，就应该在打开文档时将字符串“text/plain”传递给方法 open()。例 14-2 说明了，可以用来实现这一目标的一个方法。它实现了一个函数 debug()，可以使用这个函数将来自脚本的纯文本调试消息输出到一个单独的窗口中，该窗口在必要时就会弹出。图 14-2 显示了这个窗口。

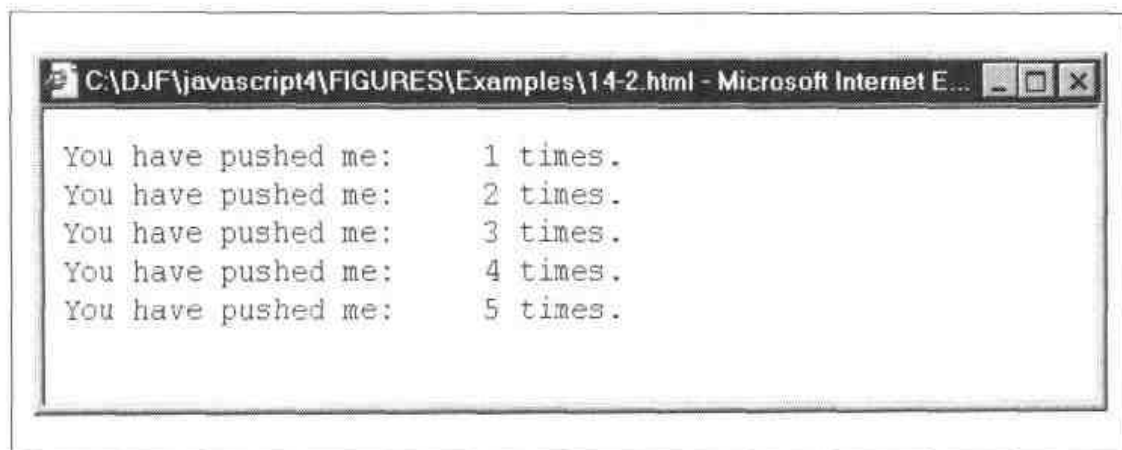


图 14-2：纯文本调试输出窗口

例 14-2：一个纯文本文档的创建

```
<script>
var _console = null;

function debug(msg)
{
    // 首次调用该函数或关闭现有控制台窗口时，打开一个窗口。
    if ((_console == null) || (_console.closed)) {
        _console = window.open("", "console", "width=600,height=300,resizable");
        // 在窗口中打开一个文档以显示纯文本。
        _console.document.open("text/plain");
    }

    _console.focus(); // 使窗口可见。
    _console.document.writeln(msg); // 给它输出消息。
    // 注意，我们有意不调用 close() 方法。
    // 使文档保持打开状态，以便以后给它添加内容。
}
</script>

<!-- 下面是一个使用该脚本的例子 -->
<script>var n = 0;</script>
<form>
<input type="button" value="Push Me"
        onclick="debug('You have pushed me:\t' + (n + ' times.');" />

</form>
```

14.3 Document 对象的颜色属性

Document 对象的 `bgColor` 属性、`fgColor` 属性、`linkColor` 属性、`aLinkColor` 属性和 `vLinkColor` 属性分别指定了文档的前景颜色、背景颜色和链接颜色。虽然这些属性是可读可写的，但是只能在解析 `<body>` 标记之前设置它们。可以在文档的 `<head>` 部分使用 JavaScript 代码对它们进行动态设置，也可以将它们作为标记 `<body>` 的性质进行静态设置。除此之外，不能在别的地方设置这些属性。这一规则有一个例外，那就是属性 `bgColor`。在许多浏览器中，可以随时设置这个属性，这样会引发浏览器窗口的背景颜色改变（注2）。除了 `bgColor` 属性，Document 对象的其他颜色属性只影响 `<body>` 标记的性质，基本上没有其他作用。

每个颜色属性都有一个字符串值。可以使用预定义的 HTML 颜色名来设置颜色，也可以使用 6 位十六进制数 `#RRGGBB` 把颜色设置成红、绿、蓝的调和值。例 13-7 就将 `<body>` 标记的特性 `bgColor` 设置成以这种形式表示的一个颜色串。

在 W3C DOM 标准中，建议不要使用 Document 对象的颜色属性，而选用表示 `<body>` 标记的 Element 对象的属性。不过，HTML 4 标准则不赞成使用 `<body>` 标记的颜色性质，而选用 CSS 样式表，这意味着你不应该编写严重依赖这些遭双重反对的颜色属性的脚本。

14.4 Document 对象的信息属性

Document 对象的几个属性提供了文档的整体信息。例如，下面的代码展示了如何使用属性 `lastModified`、`title` 和 `URI` 给文档添加一个自动的时间戳。这使用户能够判断文档是最近的还是过时的，打印文档时，这个信息也非常有用。

```
<html><font size= 12>
Document: <script>document.write(document.title);</script></font>
URL: </><script>document.write(document.URI);</script></font>
Last Updated: <script>document.write(document.lastModified);</script></font>
</html>
```

注 2：在 UNIX 平台的 Netscape 3 中存在一个 bug，这样会改变背景颜色，从而使页面内容消失（通常在滚动了窗口或重画窗口时才会再出现）。在 Netscape 6 中，只能设置一次 `bgColor` 属性一次，而其他设置都会被忽略。

referrer 是另一个比较有趣的属性，它存放了使用户链接到当前文档的文档的 URL。一种可能的用法是将它保存在网页内表单的一个隐藏域中。当用户提交这个表单时（无论怎样，页面首先包含的总是表单），就可以将这个引用者的数据保存在服务器上。这样就可以分析那个引用页面的链接，还可以跟踪各种链接的命中率。该属性的另一种用法是一个小技巧，可以使对页面的未授权链接不能进行正确操作。例如，假定你只想让别的站点链接到你的站点的主页，那么可以使用 referrer 属性和 Window 对象的属性 location，将来自外部站点的链接重定向到主页：

```
<script>
// 如果从外部站点链接，首先去主页
if (document.referrer != "" || document.referrer.indexOf('mysite.com') != -1)
    window.location = "http://home.mysite.com";
</script>
```

当然，千万不要用这个小技巧来衡量重大的安全问题。它有一个显而易见的缺陷，即对于不支持 JavaScript 的浏览器和禁用 JavaScript 的用户来说，它不起作用。

14.5 表单

Document 对象的 forms[] 数组包含 Form 对象，该对象代表文档中的 <form> 元素。由于 HTML 表单含有按钮、文本输入字段以及其他输入元素，而 Web 应用程序的 GUI 通常是由这些元素构成的，所以在客户端 JavaScript 中，Form 对象非常重要。Form 对象有一个 elements[] 属性，该属性包含代表表单中具有 HTML 输入元素的对象。就是这些 Elements 对象使得 JavaScript 程序可以在表单中设置默认值，而且能够从表单中读取用户输入。此外它们还是加载事件处理程序的重要位置，正是事件处理程序给程序带来了交互性。

由于表单和它的元素是客户端 JavaScript 的程序设计中一个庞大而重要的部分，所以用整个一章来介绍它们一点也不为过。我们将在第十五章中再对 forms[] 数组和 Form 对象进行介绍。

14.6 图像

Document 对象的 images[] 属性是一个 Image 元素的数组，每个元素代表文档中含有的一个内联图像，这些对象由标记 创建。images[] 数组和 Image 对象都

是 JavaScript 1.1 中添加的。虽然浏览器总是可以显示标记 `` 创建的图像，但是 Image 对象的出现却是一大进步，它使程序可以动态地对图像进行操作了。

14.6.1 使用 src 属性的图像置换方法

Image 对象的主要特性是它的 src 属性既可读又可写。可以读取这一属性获得图像来源的 URL。更重要的是，还可以设置 src 属性使浏览器装载一个新图像并显示在同一个地方。要实现这一点，新图像必须和原始图像具有相同的宽度和高度。

在实际应用中，图像置换最常见的用法是实现图像翻转，即当鼠标指针移到图像上时，图像就改变。当把图像放在超链接中使它们可以点击时，翻转效果是使用户点击图像的有力方式。下面是一个简单的 HTML 代码段，它在标记 `<a>` 中显示图像，使用 onmouseover 和 onmouseout 事件处理程序中的 JavaScript 代码创建翻转效果：

```
<a href="help.html"
  onmouseover="document.helpimage.src='images/help_collover.gif';"
  onmouseout="document.helpimage.src='images/help.gif';">
  
</a>
```

在这个代码段中，给 `` 标记指定了 name 性质，这样在标记 `<a>` 的事件处理程序中引用相应的 Image 对象会更容易。我们用 border 性质使浏览器不在图像周围显示蓝色的超链接边线。标记 `<a>` 的事件处理程序负责所有工作，通过把图像的 src 属性设置成要显示的图像的 URL 来改变显示的图像。

动态地将一个静态 HTML 文档中的图像替换为另一个图像可以产生许多特殊效果，如动画效果，或实时进行自我更新的数字时钟。细想一下，你还能想象出这一方法的更多用法。

14.6.2 屏外的图像和缓存

要使图像置换技术切实可行，还要动画或其他特殊效果都能作出响应。这就是说，我们还需要一种方法，用来确保所需的图像被“预载”进了浏览器的缓存中。要迫使图像被缓存起来，需要用 Image() 构造函数创建一个屏外图像，然后把它的 src 属性设置成想要的 URL（与在屏幕上创建的图像完全一样）。此后，当要在屏幕上使用同一个 URL 时，我们知道它能够从浏览器的缓存中被迅速地装载进来，而不是

通过网络慢腾腾地下载。注意，我们从不用屏外图像做任何事，尤其不会把屏外 Image 对象赋给文档的 images[] 数组。

前一节所示的翻转代码段没有预载要使用的翻转图像，所以当用户第一次把鼠标移到该图像上时，可能会发现翻转效果有一些延迟。要修正这个错误，需要对代码做如下修改：

```
<script>
// 创建一个屏外图像，并预载翻转图像。
// 注意，我们没有保存对屏外图像的引用，
// 因为此后用它什么也不能做。
img = Image(80,20); src = "images/help_rollover.gif";
</script>
<a href= "help.html"
    onmouseover="document.helpImage.src='images/help_rollover.gif'";
    onmouseout="document.helpImage.src='images/help.gif'";">
    img name= "helpImage" src="images/help.gif" width="80" height="20" border="0" >
</a>
```

例 14-3 展示的代码用图像置换方法执行简单动画，用屏外图像预载该动画的帧。注意，在这个例子中我们保留了创建的屏外图像，因为用它们保存构成动画的图像的 URL 很方便。我们通过把作为动画主题的屏幕上图像的 src 属性设置为屏外图像的 src 属性来执行动画。

例 14-3：使用图像置换方法的动画

```
<!-- 这个图像将动起来。为了方便起见，给它一个名字。 -->


<script>
// 创建一组屏外图像，预先把动画的“帧”存放在图像中。
// 这样当我们需要它们时，它们已经被缓存了。
var animframes = new Array(10);
for(var i = 0; i <= 10; i++) {
    animframes[i] = new Image(); // 创建一个屏外图像、
    animframes[i].src = "images/" + i + ".gif"; // 告诉它要装载的 URL。
}

var frame = 0; // 帧计数器：跟踪当前的帧。
var timeout_id = null; // 允许我们用 clearTimeout() 方法停止动画。

// 该函数将执行动画。调用它开始执行动画。
// 注意我们用 name 性质引用屏内图像。
function animate() {
    document.animation.src = animframes[frame].src; // 显示当前帧。
    frame = (frame + 1)%10; // 更新帧计数器。
```

```

        timeout_id = setTimeout( animate(), 250);           // 此后显示下一帧。
    }
</script>

<form> <!-- 该表单包含控制动画的按钮 -->
    <input type="button" value="Start"
           onclick="if (timeout_id != null) animate();" />
    <input type="button" value="Stop"
           onclick="if (timeout_id) clearTimeout(timeout_id); timeout_id=null;" />
</form>

```

14.6.3 Image 对象的事件处理程序

在例 14-3 中，动画是在用户点击了 **Start** 按钮之后才开始执行的，这样就有足够的时间把图像装载进缓存。但是如果我们想让动画在所有必要的图像都装载完毕时就自动开始执行，对于这种更为一般的情况又会怎样呢？结果是，无论一个 Image 对象是在屏幕上由标记 创建，还是在屏外由构造函数 Image() 创建的，它都具有事件处理程序 onload，当图像完全被装载后该程序就会被调用。

接下来的代码段说明了如何修改例 14-3，以使用这个事件处理程序对已经装载的图像计数并在所有图像装载完毕时自动启动动画。由于屏外图像不是 HTML 文档的一部分，所以不能把这个事件处理程序赋给一个 HTML 属性，相反，只能把一个函数赋给我们创建的每个 Image 对象的 onload 属性。在装载每个图像时，浏览器将调用这个函数：

```

var aniframes = new Array(10); // 存放屏外动画的帧。
var num_loaded_images = 0;     // 迄今为止，装载了多少帧？

// 该函数被用作事件处理程序，它对装载的图像进行计数，
// 在所有图像装载完毕时，它将启动动画。
function countImages() {
    if (++num_loaded_images == aniframes.length) animate();
}

// 创建屏外图像，并赋予它们 URL。
// 给每个图像赋予事件处理程序，以便我们跟踪装载了多少图像。
// 注意，我们在赋予图像 URL 之前赋予它们事件处理程序，
// 否则可能在我们赋予图像处理程序之前，图像已经结束装载（即它已经被缓存了）。
// 这样处理程序就永远不会被触发了。
for (var i = 0; i < 10; i++) {
    aniframes[i] = new Image();           // 创建屏外图像。
    aniframes[i].onload = countImages;    // 赋予事件处理程序。
    aniframes[i].src = "images/" + i + ".gif"; // 告诉它要装载的 URL。
}

```

除了事件处理程序 `onload` 之外, `Image` 对象还支持另外两个事件处理程序。事件处理程序 `onerror` 是在图像装载过程中出现了错误时调用的, 例如当指定的 URL 引用了一个被破坏的图像数据时。`onabort` 处理程序是在用户在图像装载完之前就取消了它 (例如点击了浏览器的 **Stop** 按钮) 时被调用的。任何图像都可以调用这三个事件处理程序中的一个, 而且只能调用一个。

除了这些事件处理程序之外, 每个 `Image` 对象都有一个 `complete` 属性。当图像处于正在装载的状态时, 这个属性值为 `false`, 当图像装载完毕后, 或者浏览器停止了装载时, 它的值就是 `true`。也就是说, 如果上述的三个事件处理程序中的一个被调用了, 属性 `complete` 的值就为 `true`。

14.6.4 Image 对象的其他属性

`Image` 对象还具有其他几个属性。其中大部分属性都是只读的, 只是反映创建图像的标记 `` 的属性值。`width`、`height`、`border`、`hspace` 和 `vspace` 都是只读的整数, 它们分别声明了图像的大小、边界的宽度以及垂直边缘和水平边缘的尺寸。这些属性都是由 `` 的同名性质设置的。在 Netscape 3 和 4 中, 该属性是只读的, 但在 IE 4 及其后的版本中和 Netscape 6 及其后的版本中, 可以给这些属性赋值, 动态地改变图像的大小、边界和边缘。

`Image` 对象的属性 `lowsrc` 反映了 `` 标记的性质 `lowsrc`。它指定在低分辨率的设备上浏览页面时显示的图像的 URL, 这个图像是可选的。该属性和 `src` 一样是个可读可写的字符串, 但和 `src` 属性不同的是, 设置 `lowsrc` 不会引发浏览器装载并显示出那个新指定的低分辨率的图像。如果你想用使用高分辨率的图像那样的方式使用低分辨率的图像来执行一个动画, 或者制造其他特殊效果, 就要记住在设置属性 `src` 之前一定要更新属性 `lowsrc`。如果设置 `src` 时浏览器是在一个低分辨率的设备上运行, 那么它将装载新的 `lowsrc` 图像。

14.6.5 图像置换方法示例

由于图像置换方法的多样性, 我们用一个扩展了的例子来结束对 `Image` 对象的讨论。例 14-4 定义了一个类 `ToggleButton`, 它使用图像置换方法模拟了一个图像化的复选框。因为这个类使用的图像是我们提供的, 所以我们可以不采用标准 HTML

checkbox对象的平面图形,而采用加粗了的图形。图14-3展示的就是切换按钮在网页中的样子。这个一个复杂而真实的例子,值得仔细研究。

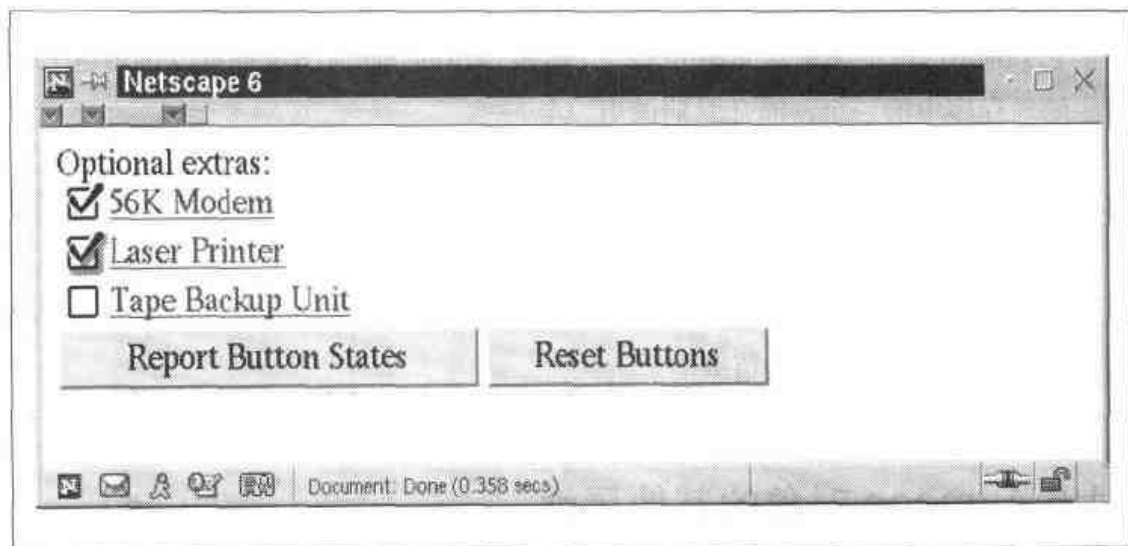


图 14-3: 用图像置换方法实现的切换按钮

例 14-4: 用图像置换方法实现一个切换按钮

```
<script language="JavaScript1.1">
// 这是用于新类 ToggleButton 的构造函数。
// 调用它将创建一个 ToggleButton 对象,
// 并把必需的 <a> 和 <img> 标记输出到当前位置的指定文档。
// 因此, 不要从事件处理程序为当前文档调用它。
// 参数:
//   document: 要在其中创建按钮的 Document 对象。
//   checked:   布尔变量, 说明按钮初始时是否被选中了。
//   label:    可选字符串, 指定了按钮上显示的文本。
//   onclick:  一个可选的函数, 在切换按钮被点击时调用。
//             可将一个布尔值传递给它, 说明按钮的新状态。
//             也可以传递给它一个字符串, 该串将被转换成一个函数,
//             这个函数将得到一个名为 state 的布尔参数。
function ToggleButton(document, checked, label, onclick)
{
    // 初次被调用时 (只限于第一次), 我们必须进行一些特殊处理。
    // 首先, 既然已经创建了原型对象, 就可以设置它的方法。
    // 其次, 必须装载要用的图像。
    // 这样在我们要用图像时, 它们已经在缓存中了。
    if (!ToggleButton.prototype.over) {
        // 初始化原型对象来创建我们的方法。
        ToggleButton.prototype.over = _ToggleButton_over;
        ToggleButton.prototype.out = _ToggleButton_out;
        ToggleButton.prototype.click = _ToggleButton_click;

        // 下面我们创建一个 Image 对象的数组, 并赋予这些对象 URL。
        // 图像的 URL 是可配置的, 它们存放在构造函数自身的数组属性中。
```

```

// 以下将对它们进行初始化。
// 由于katscape 中有一个bug,
// 我们必须支持对这些图像的引用, 因此我们把这个数组保存为构造函数的一个属
// 性, 而不是局部变量。
ToggleButton.prototype.images = new Array(4);
for(var i = 0; i < 4; i++) {
    ToggleButton.prototype.images[i] = new Image(ToggleButton.width,
                                                    ToggleButton.height);
    ToggleButton.prototype.images[i].src = ToggleButton.imageNames[i];
}

// 保存传递过来的某些参数。
this.document = document;
this.checked = checked;

// 记住, 鼠标当前并不在顶层。
this.highlighted = false;

// 保存 onclick 参数, 以便点击按钮时调用它
// 如果它还不是一个函数, 则将它转换成函数, 唯一的参数名为 state。
this.onclick = onclick;
if (typeof this.onclick != "string")
    this.onclick = new Function("state", this.onclick);

// 在 document.images 数组中设置存放该复选框图像的条目。
var index = document.images.length;

// 下面输出复选框的HTML代码, 使用<a>和<img>标记。
// 虽然这里输出的事件处理程序有些混乱, 但对该类的操作至关重要。
// 以下定义了 _tb 属性和 over()、out()、click() 方法。
document.write('&nbsp;&nbsp;&nbsp;<a href="about:blank" ' +
    'onmouseover="document.images[' + index + ']._tb.over();return true; ' +
    'onmouseout="document.images[' + index + ']._tb.out()" ' +
    'onclick="document.images[' + index + ']._tb.click(); return false;">');
document.write('');
if (label) document.write(label);
document.write('</a><br>');

// 既然我们已经输出了<img>标记,
// 那么就保存 ToggleButton 对象中创建的对 Image 对象的引用。
this.image = document.images[index];

// 创建一个从 Image 对象到 ToggleButton 对象的链接。
// 通过给 Image 对象定义 _tb 属性, 可以实现这一点。
this.image._tb = this;
}

// 这将成为 over() 方法。

```

```

function _toggleButton_over()
{
    // 更换图像, 记得要高亮显示。
    this.image.src = ToggleButton.imageNames[this.checked + 2];
    this.highlighted = true;
}

// 这将成为 out() 方法。
function _toggleButton_out()
{
    // 更换图像, 记得取消高亮显示。
    this.image.src = ToggleButton.imageNames[this.checked + 0];
    this.highlighted = false;
}

// 这将成为 click() 方法
function _toggleButton_click()
{
    // 切换按钮的状态, 更换图像。
    // 如果指定了 ToggleButton 对象的 onclick 方法, 就调用它。
    this.checked = !this.checked;
    this.image.src = ToggleButton.imageNames[this.checked + this.highlighted * 2];
    if (this.onclick) this.onclick(this.checked);
}

// 初始化描述复选框图像的静态类属性。
// 这些只是默认值。通过给它们赋新值, 可以覆盖它们。
// 但是只能在创建 ToggleButton 之前覆盖它们。
ToggleButton.imageNames = new Array(4); // 创建一个数组。
ToggleButton.imageNames[0] = "images/button0.gif"; // 未被选中的复选框。
ToggleButton.imageNames[1] = "images/button1.gif"; // 具有复选标记的复选框。
ToggleButton.imageNames[2] = "images/button2.gif"; // 未被选中但高亮显示的复选框。
ToggleButton.imageNames[3] = "images/button3.gif"; // 被选中且高亮显示的复选框。
ToggleButton.width = ToggleButton.height = 25; // 所有图像的大小。
</script>

<!-- 以下是如何使用 ToggleButton 类的代码 -->
Optional extras:<br>
<script language= JavaScript1.1">
// 创建 ToggleButton 对象, 输出实现它们的 HTML 代码。
// 一个按钮没有 Click 处理程序, 一个具有函数, 一个具有字符串。
var tb1 = new ToggleButton(document, true, "56K Modem");
var tb2 = new ToggleButton(document, false, "Laser Printer",
    function(clicked) {alert('printer: ' + clicked);});
var tb3 = new ToggleButton(document, false, "Tape Backup Unit",
    "alert('Tape backup: ' + state)");
</script>

<!-- 以下是如何从事件处理程序使用 ToggleButton 对象。 -->
<form>
<input type="button" value="Report Button States"
    onClick= alert(tb1.checked + "\n" + tb2.checked + "\n" + tb3.checked);>

```

```


</form>

```

14.7 链接

Document 对象的 links[] 数组包含代表文档中的超文本链接的 Link 对象。我们知道, HTML 的超文本链接是由标记 <a> 的性质 href 设置的。在 JavaScript 和其后的版本中, 客户端图像中的标记 <area> 也可以在 Document 对象的 links[] 数组中创建 Link 对象。

Link 对象代表超文本链接的 URL, 此外还含有 Location 对象 (第十三章中介绍过) 的所有属性。例如, Link 对象的 href 属性存储的是该对象要链接到的 URL 的完整文本, 而 hostname 属性存储的只是那个 URL 中的主机名部分。要了解这些与 URL 有关的属性请参阅客户端参考手册部分的完整列表。

例 14-5 说明了一个函数, 它能够生成一个具有文档中所有链接的列表。注意, 该函数使用 Document 对象的 write() 和 close() 方法动态地生成了一个文档, 就像我们在本章前面的小节中讨论的那样。

例 14-5: 列出文档中的所有链接

```

/*
 * FILE: listlinks.js
 * 在新窗口中列出指定文档内的所有链接。
 */
function listlinks(d) {
    var newwin = window.open("", "linklist",
                              'menubar,scrollbars,resizable,width=600,height=300');

    for (var i = 0; i < d.links.length; i++) {
        newwin.document.write('<a href="' + d.links[i].href + '>')
        newwin.document.write(d.links[i].href);
        newwin.document.writeln('</a><br>');
    }
    newwin.document.close();
}

```


14.7.1 链接、网络爬虫和 JavaScript 安全性

Link对象和links[]数组的一种常见用法是编写网络爬虫程序。这种程序在一个浏览器窗口或框架中运行,读取的却是另一个窗口或框架中的网页(通过设置Window对象的location属性实现的)。对于每个读进的页面,它都会查询links[]数组,并且递归地显示出要链接到的文档。如果谨慎地编写这样的程序(不要让它陷入无限的递归或者循环中),它是非常有用的,如可以用来生成从给定页面开始的所有可访问页面的列表。这在网站维护中极其有用。

但是不要以为使用这些方法就可以在整个Internet中爬行了。出于安全性的原因,JavaScript不允许一个窗口或框架中的未签名脚本读另一个窗口或框架的属性(如document.links),除非这两个窗口显示的文档来自同一个服务器。这条限制可以防止出现重大的安全漏洞。假定在一个有安全性意识的公司中的某个职员一边通过公司的防火墙浏览Internet,一边又使用自己的浏览器浏览intranet上的企业内部信息。如果没有我们介绍的安全性规约,那么一个来自随机的Internet站点的不可信赖的脚本都能够窥探另一个窗口中的情况。也许窥探脚本的作者并不能从私有文档的links[]数组中得到非常有用的信息,但仍然是一个严重的安全性漏洞。

虽然我们所介绍的网络爬虫程序并不能对Internet安全性或保密性造成威胁,但是它仍然受制于JavaScript常规的安全规约,这使得它不能爬行到远离自己所在的站点的地方(当爬虫程序要从另一个站点装载网页时,就好象根本不存在对那个网页的链接一样)。要查阅有关JavaScript安全性的完整讨论,请参阅第二十一章,其中有如何使用带签名的脚本避免安全性制约的介绍。

14.7.2 Link 对象的事件处理程序

Link对象支持大量有趣的事件处理程序。我们已经在13.3节见到过事件处理程序onmouseover,它和标记<a>和<area>一起用于在鼠标移动到链接上时改变状态栏中的消息。事件处理程序onclick是在用户点击超文本链接时调用的。在JavaScript 1.1和其后的版本中,如果该事件处理程序返回了false,浏览器就不会显示要链接的文档。从JavaScript 1.1起,<a>和<area>标记都支持onmouseout。这个事件处理程序和onmouseover恰好相反,它在鼠标指针离开超文本链接时运行。

事件处理程序模型在 JavaScript 1.2 中更加通用。此外，链接还支持许多其他的事件处理程序，要对其进行详细了解，请参阅第十九章。

最后，值得一提的是 Link 对象的属性 href 和 URL 都是可读可写的。这样就可以编写 JavaScript 程序，动态地修改超文本链接的目标文件了。有许多 JavaScript 扩展了的 HTML 都使用 Link 事件处理程序来设置 href 属性，创建从文档中的其他链接集合随机选出目标的超文本链接：

```
<a href="#about":  
  onmouseover= status = 'Take a chance... Click me.' ; return true;  
  onclick= this.href =  
    document.links[Math.floor(Math.random()*document.links.length)];  
>  
Random link  
</a>
```

这个例子说明了我们讨论过的 Link 对象的所有特性，即 links[] 数组、Link 事件处理程序的用法以及对一个链接的目标文件的动态设置。注意，本例中仅仅设置了链接属性 href，并没有读那些随机选择的链接的 href 属性。相反，它调用 Link 对象的 toString() 方法返回 URL。

14.8 锚

Document 对象的数组 anchors[] 包含了代表 HTML 文档中已命名位置的 Anchor 对象，这些位置都由标记 <a> 和它的是性质 name 标识。虽然从 JavaScript 1.0 开始就已经存在 anchors[] 数组了，但是 Anchor 对象却是在 JavaScript 1.2 中新加入的。在以前的版本中，anchors[] 数组的元素都是未定义的，只有 length 属性有用。

Anchor 对象是一种比较简单的对象。它所定义的唯一标准属性是 name，即 HTML 性质 name 的值。和 Link 对象一样，出现在锚的标记 <a> 和 之间的文本在 Netscape 4 中是由属性 text 指定的，在 Internet Explorer 4 中由 innerText 指定。W3C DOM 标准不支持这些属性，不过在第十七章可以看到其他获取元素的文本内容的方法。

例 14-6 展示的函数为指定的文档创建了一个新的导航窗口。在这个窗口中将显示该

文档包含的所有锚元素的 text 属性、innerText 属性或 name 属性。这些锚文件或名称是在这个锚超文本链接中显示的, 这样点击任何一个锚元素都会使窗口滚动并显示这个锚。如果你编写的 HTML 文档要将所有子标题都放入锚元素中 (示例如下), 那么这个例子是非常有参考价值的。例如:

```
<a name="sect14.6"><h2>The Anchor Object</h2></a>
```

例 14-6: 列出所有的锚元素

```
/*
 * IIFE: listanchors.js
 * 函数 listanchors() 的参数是一个文档, 它将打开一个新窗口作为那个文档的导航窗口。
 * 新窗口将显示该文档中所有锚元素的列表, 点击这个列表中任何锚元素都会使
 * 文档滚动到那个锚的位置。
 * 在文档被完全解析或至少所有锚元素被解析之前, 文档自身不应该调用该函数。
 */
function listanchors(d) {
    // 打开新窗口
    var newwin = window.open("", 'navwin',
        "menubar=yes,scrollbars=yes,resizable=yes," +
        "width=600,height=300");

    // 赋给它标题
    newwin.document.writeln("<h1>Navigation Window:<br>" +
        document.title + "</h1>");

    // 列出所有锚元素
    for(var i = 0; i < d.anchors.length; i++) {
        // 对于每个锚对象, 确定要显示的文本。
        // 首先, 用浏览器决定的属性获取 <a> 和 </a> 之间的文本
        // 如果没有这样的文本, 用 name 属性代替。
        var a = d.anchors[i];
        var text = null;
        if (a.text) text = a.text; // Netscape 4.
        else if (a.innerText) text = a.innerText; // IE 4+.
        if ((text == null) || (text == '')) text = a.name; // 默认值。

        // 下面用链接形式输出文本。
        // 注意原始窗口的 location 属性的用法。
        newwin.document.write('<a href="#" + a.name + "' +
            ' onclick="opener.location.hash=' + a.name +
            '"; return false;"> ');
        newwin.document.write(text);
        newwin.document.writeln('</a><br> ');
    }
    newwin.document.close(); // 不要忘记关闭文档。
}
```

14.9 小程序

Document 对象的 `applets[]` 数组表示由标记 `<applet>` 或 `<object>` 嵌入在文档中的小程序的对象。所谓小程序就是通过 Internet 装载的、由浏览器执行的、具有可移植性和安全性的 Java 程序。Netscape 和 Internet Explorer 都支持 Java (尽管 IE 6 不再对 Java 提供默认的支持)。

从 Netscape 3 和 Internet Explorer 3 起,这两种浏览器都允许 JavaScript 调用 Java 小程序的公用方法,并且读写它们的公用属性(我们将在第二十二章中看到,Netscape 还支持 JavaScript 和 Java 之间的双向交互)。虽然所有的小程序都从它们的超类那里继承了标准的公用方法,但是最有趣的方法和属性是随实际情况而变化的。如果你是那个小程序的作者,想通过 JavaScript 控制这些小程序,就应该了解它所定义的公用属性和方法。如果你不是它的作者,就应该查询它的文档以确定可以用它做些什么。

下面的代码说明了如何用标记 `<applet>` 将一个 Java 小程序嵌入到网页中,然后从 JavaScript 事件处理程序中调用那个小程序的 `start()` 方法和 `stop()` 方法:

```
<applet name="animation" code="Animation.class" width="500" height="200">
</applet>
<form>
<input type="button" value="Start" onclick="document.animation.start();">
<input type="button" value="Stop" onclick="document.animation.stop();">
</form>
```

所有小程序都定义了 `start()` 方法和 `stop()` 方法。在上述假定的例子中,该方法可以启动和停止某种动画,通过定义 HTML 表单,我们可以控制小程序的启动或停止。注意,由于我们使用了 `<applet>` 标记的 `name` 性质,所以可以用名字引用那个小程序,而不必将它作为 `applets[]` 数组的一个编码元素。

这个例子并不能完全说明 Java 小程序在 JavaScript 脚本中的威力,因为从 JavaScript 的事件处理程序中调用 Java 方法既不能传递参数也没有返回值。事实上,JavaScript 可以给 Java 方法传递数字、字符串和布尔值这些参数,而且也可以接受这些函数返回的数字、字符串和布尔值(在第二十二章中我们可以看到,Netscape 还可以把 JavaScript 对象或 Java 对象传递给 Java 方法,Java 方法也可以返回这些对象)。JavaScript 和 Java 之间这种自动的数据转换使两种程序设计环境之间的大量交互成

为了可能。例如，一个小程序可以实现一个返回JavaScript代码中的方法，JavaScript可以使用 `eval()` 方法来计算那段代码。

小程序也可以实现不对自身进行操作的方法，但是这种方法只能作为JavaScript环境和Java环境之间的管道。例如，小程序可以定义一个方法为给定的字符串参数调用方法 `System.getProperty()`。这使得JavaScript能够查询Java系统的属性值并且确定那个浏览器支持的Java版本。

14.10 嵌入式数据

数组 `embeds[]` 代表由标记 `<embed>` 或 `<object>` 嵌入文档的数据（除了小程序）的对象。被嵌入的数据可以是任何形式的（音频数据、视频数据或表，等等）。浏览器必须具有合适的查看程序才能显示这些数据。在Netscape中，显示嵌入式数据的特殊模块被称为“插件”。在Internet Explorer中，嵌入式数据则是由ActiveX控件显示的。无论是插件还是ActiveX控件都会在必要时自动地从网络上下载并安装。

尽管 `applets[]` 数组的元素代表的都是Java小程序，但是 `embeds[]` 数组的元素却是多样的，很难将它们概括起来。这些对象的属性和方法都与显示嵌入式数据所使用的插件或ActiveX控件有关。对于你所使用的插件或ActiveX控件，应该参考销售商提供的有关插件或ActiveX控件的文档。如果它还支持JavaScript以外的其他脚本，文档会有所说明，而且它还会描述你可以使用的JavaScript属性和方法。例如，来自Netscape的插件LiveVideo的文档中记述的就是 `embeds[]` 数组中的LiveVideo对象，它支持四种方法，它们分别是 `play()`、`stop()`、`rewind()` 和 `seek()`。有了这些信息，你就可以编写简单的脚本来控制如何用插件显示嵌入在网页中的电影了。注意，虽然某些销售商可能会生产定义同一公用API的插件（在Netscape中）和ActiveX控件（在IE中），但是情况并不总是这样的。嵌入了对象的脚本通常都会涉及到特定平台的JavaScript代码。

第十五章

表单和表单元素

我们已经看到，在本书的例子中，HTML表单的使用是JavaScript程序的基本要素。本章详细介绍了如何在JavaScript中使用表单进行程序设计。但阅读本章的前提是你已经掌握了HTML表单的创建，了解它包含的输入元素。如果不是这样，你可以参考一本有关HTML的书（注1）。本书的客户端参考手册部分除列出了有关表单和表单元素的JavaScript语法外，还列出了相应的HTML语法，你会发现这些快速参考非常有用。

如果你对使用HTML表单的服务器端程序设计已经略知一二，那么就会发现在JavaScript中使用表单的情况与前者完全不同。在服务器端模型中，具有输入数据的表单会被立刻提交给Web服务器。它的重点在于处理整批输入的数据，然后动态地生成一个新网页作为响应。而JavaScript的程序设计模型不是这样。在JavaScript的程序中，重点并不在于表单的提交和处理，而在于事件处理。一个表单及其所有输入元素都具有事件处理程序，JavaScript可以使用这些处理程序响应用户与表单的交互。例如，如果用户选中了一个复选框，JavaScript程序就会通过事件处理程序收到一个通知，然后它可能会改变其他表单元素的显示值进行响应。

在服务器端程序中，如果一个HTML表单没有**Submit**按钮（或者没有一个文本输

注1：如《HTML: The Definitive Guide》，Chuck Musciano 和 Bill Kennedy 著，O'Reilly 公司出版。

入框，不允许用户以回车键作为提交的快捷键)，那么它就没有任何用途。但是在 JavaScript 中，**Submit** 按钮则不是必须的（当然，如果 JavaScript 程序和服务器端程序联合使用，那么 **Submit** 按钮还是必须的）。而且，在 JavaScript 中，一个表单可以具有任意多个按钮，这些按钮具有的事件处理程序在按钮被点击时可以执行任意的动作。在前面一章中，我们已经看到了这样的按钮可能触发的几种动作，即用一个图像替换另一个图像，使用 `location` 属性装载并显示出一个新的网页，打开一个新的浏览器窗口以及在另一个窗口或框架中动态地生成一个新的 HTML 文档。在本章后面的部分中，我们将看到，JavaScript 的事件处理程序甚至可以触发一个表单的提交。

我们已经看到，在本书的例子中，事件处理程序几乎是所有有趣的 JavaScript 程序的中心元素。而且最常用的事件处理程序（除了 `Link` 对象的事件处理程序）是由表单和表单元素使用的。本章介绍了 JavaScript 的 `Form` 对象和代表表单元素的各种 JavaScript 对象。最后它以一个例子作为总结，这个例子说明了如何使用 JavaScript 在把用户输入提交给运行在 Web 服务器上的服务器端程序之前对它进行验证。

15.1 Form 对象

JavaScript 的 `Form` 对象代表了一个 HTML 表单。通常在 `Document` 对象的属性 `forms[]` 数组的元素中可以找到 `Form` 对象。在这个数组中，`Form` 对象是按照它们在文档中出现的顺序存放的。所以，`document.forms[0]` 指的就是文档中的第一个表单。可以使用如下的代码引用文档中的最后一个表单：

```
document.forms[document.forms.length - 1]
```

`Form` 对象最有趣的属性是 `elements[]` 数组，它包含表示各种表单输入元素的 JavaScript 对象（各种类型的）。而且，这个数组中的元素也是按照它们在文档中出现的顺序存放的。因此可以用下面的代码引用当前窗口中的文档内的第二个表单的第三个元素：

```
document.forms[1].elements[2]
```

其余的 `Form` 对象的属性就不那么重要了。它们是 `action`、`encoding`、`method` 和 `target`，直接对应于 HTML 标记 `<form>` 的性质 `action`、`encoding`、`method`

和target。这些属性和性质都用于控制如何将表单数据提交给网络服务器以及在哪里显示结果,因此只有在表单被真正提交给服务器端程序时它们才会有用。要了解这些属性,可以参阅本书的客户端参考手册部分,要得到有关这些性质的完整讨论,请查阅有关HTML或CGI程序设计的书(注2)。值得注意的是,这些Form属性都是可读可写的字符串,因此JavaScript程序可以动态地设置它们的值来改变提交表单的方式。

在JavaScript出现之前,表单靠专用的**Submit**按钮提交,表单元素则靠专用的**Reset**按钮重置。JavaScript的Form对象支持的两种方法,submit()(在JavaScript 1.1中添加的)和reset()就是专用于上述目的的。通过调用Form对象的submit()方法可以提交表单,调用reset()方法可以重置表单元素。

为了配合submit()方法和reset()方法,Form对象还提供了事件处理程序onsubmit和onreset,前者用来探测表单的提交(在JavaScript 1.1中添加的),后者用来探测表单的重置。onsubmit是在表单提交之前调用的,如果它返回false,就取消表单的提交。这给JavaScript程序提供了一个机会来检查用户的输入是否有错,以避免通过Web给服务器端程序提交不完整的或无效的数据。我们将在本章的结尾部分看到一个错误检测的例子。注意,只有真正点击**Submit**按钮才会触发onsubmit事件处理程序,调用表单的submit()方法则不会触发它。

事件处理程序onreset与处理程序onsubmit相似。它在重置表单时被调用,如果它返回false,可以阻止重置表单元素。这使JavaScript程序可以请求重置的确认信息,如果表单又长又复杂,那么这是个不错的主意。可以使用如下所示的事件处理程序来请求这种类型的确认信息:

```
<form...  
  onreset="return confirm('Really erase ALL data and start over?')"  
...
```

与onsubmit一样,只有真正点击**Reset**按钮才会触发onreset事件处理程序。调用表单的reset()方法不会触发它。

注2: 如《CGI Programming on the World Wide Web》, Shishir Gundavaram 著, O'Reilly 公司出版。

15.2 定义表单元素

HTML的表单元素是原始对象，我们用这些对象为JavaScript程序创建图形用户界面。图15-1展示了一个复杂表单，其中基本的表单元素至少出现过一次。如果你还不熟悉HTML表单元素，该图具有标识每个元素类型的编码。本节结尾处有一个例子（例15-1），说明了创建图15-1所示的表单的HTML代码和JavaScript代码，它把每个表单元素和事件处理程序联系了起来。

表15-1列出了HTML设计者和JavaScript程序员可用的表单元素的类型。该表的第一列是表单元素的类型名，第二列显示了用于定义那种类型的元素的HTML标记，第三列列出了每个元素类型的type属性值。我们知道，每个Form对象都有elements[]数组，用来存放表示表单元素的对象。每个元素都有type属性，用于区别不同类型的元素。通过检测一个未知表单元素的type属性，JavaScript代码可以确定该元素的类型并估计出用该元素可以做什么（在本章结尾的例15-2中可以看到这一点）。该表的最后一列简短描述了每个元素，还列出了它们最重要或最常用的事件处理程序。

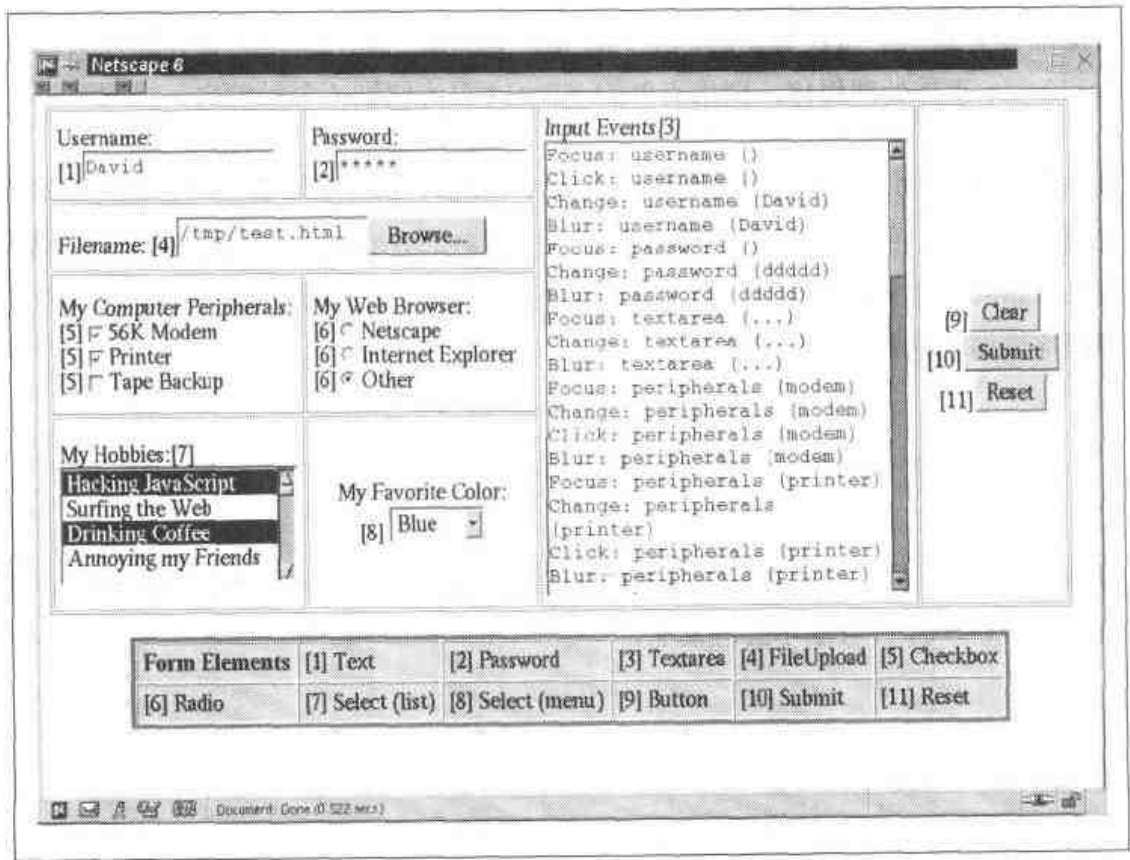


图15-1：HTML表单元素

我们将在本章后面的部分讨论更多有关表单元素的内容。在本书的客户端参考手册部分对各种类型的元素进行了完整描述，其参考页在表 15-1 的第一列列出的名字下。虽然每种表单元素类型都有独立的参考页，但要注意，大部分元素都是由HTML 标记<input>创建的，实际都是Input 对象。名为Input 的客户端参考页列出了这些元素共有的特性，指定类型的参考页列出了使用特定类型的表单元素的细节。注意，表中第一列诸如Button、Checkbox 这样的元素在客户端JavaScript 的实现中可能没有相应的真实对象，此外，DOM 标准没有定义使用这些名字的接口。每种类型的表单元素都具有不同的外观和不同的行为，把它们作为不同类型来分别处理它们比较有用，至少对于客户端参考手册部分来说是这样。在DOM 参考手册部分，你会在HTMLFormElement、HTMLInputElement、HTMLTextAreaElement、HTMLSelectElement 和HTMLOptionElement 名下会找到有关表单和表单元素的资料。

表 15-1: HTML 表单元素

对象	HTML 标记	type 属性	描述和事件
Button	<input type="button"> 或 <button type="button">	"button"	按钮; onclick
Checkbox	<input type="checkbox">	"checkbox"	不具有单选按钮行为的切换按钮; onclick
FileUpload	<input type="file">	"file"	用于输入要上载到 Web 服务器的文件名的输入框; onchange
Hidden	<input type="hidden">	"hidden"	随表单提交的数据，对用户不可见; 没有事件处理程序
Option	<option>	无	Select 对象中的一个项目; 事件处理程序属于 Select 对象，而不属于单独的 Option 对象
Password	<input type="password">	"password"	输入口令的输入框，键入的字符不可见; onchange
Radio	<input type="radio">	"radio"	具有单选按钮行为（即一次只能选择一项）的切换按钮; onclick

表 15-1: HTML 表单元素 (续)

对象	HTML 标记	type 属性	描述和事件
Reset	<code><input type="reset"></code> 或 <code><button type="reset"></code>	"reset"	重置表单的按钮; onclick
Select	<code><select></code>	"select-one"	列表或下拉式菜单, 可以选中其中一个项目; onchange (参阅 Option 对象)
Select	<code><select multiple></code>	"select-multiple"	列表, 可以选中其中多个项目; onchange (参阅 Option 对象)
Submit	<code><input type="submit"></code> 或 <code><button type="submit"></code>	"submit"	提交表单的按钮; onclick
Text	<code><input type="text"></code>	"text"	单行的文本输入框; onchange
Textarea	<code><textarea></code>	"textarea"	多行的文本输入框; onchange

我们已经浏览了各种类型的表单元素和创建它们的 HTML 标记, 例 15-1 展示了用来创建图 15-1 所示的表单的 HTML 代码。虽然这个例子主要由 HTML 代码构成, 但它也含有 JavaScript 代码, 这些 JavaScript 代码用于定义每个表单元素的事件处理程序。注意, 事件处理程序不是被定义为 HTML 性质, 它们是 JavaScript 函数, 要赋予表单的 `elements[]` 数组中的对象的属性。这些事件处理程序都调用了函数 `report()`, 它含有使用各种表单元素的代码。接下来的一节解释了要理解 `report()` 函数做什么时需要知道的内容。

例 15-1: 包含所有表单元素的 HTML 表单

```
<form name='everything' >                                <!-- 一个包含所有表单元素的表单 ... -->
<table border="border" cellpadding="5">                <!-- 包含在一个大的 HTML 表中 -->
  <tr>
    <td>Username:<br>[1]<input type="text" name="username" size="15"></td>
    <td>Password:<br>[2]<input type="password" name="password" size="15"></td>
    <td rowspan="4">Input Events[3]<br>
      <textarea name="textarea" rows="20" cols="28"><|textarea></td>
    <td rowspan="4" align="center" valign="center">
      [9]<input type="button" value="Clear" name="clearbutton"><br>
      [10]<input type="submit" name="submitbutton" value="Submit"><br>
      [11]<input type="reset" name="resetbutton" value="Reset"></td></tr>
```

```

<tr>
  <td colspan="2">
    Filename: [4]<input type="file" name="file" size="15"></td></tr>
</tr>
<td>My Computer Peripherals:<br>
  [5]<input type="checkbox" name="peripherals" value="modem">Modem<br>
  [5]<input type="checkbox" name="peripherals" value="printer">Printer<br>
  [5]<input type="checkbox" name="peripherals" value="tape">Tape Backup</td>
<td>My Web Browser:<br>
  [6]<input type="radio" name="browser" value="nn">Netscape<br>
  [6]<input type="radio" name="browser" value="ie">Internet Explorer<br>
  [6]<input type="radio" name="browser" value="other">Other</td></tr>
</tr>
<td>My Hobbies: [7]<br>
  <select multiple="multiple" name="hobbies" size="4">
    <option value="programming">Hacking JavaScript
    <option value="surfing">Surfing the Web
    <option value="caffeine">Drinking Coffee
    <option value="annoying">Annoying my Friends
  </select></td>
<td align="center" valign="center">My Favorite Color:<br>[8]
  <select name="color">
    <option value="red">Red          <option value="green">Green
    <option value="blue">Blue        <option value="white">White
    <option value="violet">Violet    <option value="peach">Peach
  </select></td></tr>
</table>
</form>

<div align="center">      <!-- 另一个表, 上表的关键 -->
  <table border="4" bgcolor="pink" cellspacing="1" cellpadding="4">
    <tr>
      <td align="center"><b>Form Elements</b></td>
      <td>[1] Text</td> <td>[2] Password</td> <td>[3] Textarea</td>
      <td>[4] FileUpload</td> <td>[5] Checkbox</td></tr>
      <tr>
        <td>[6] Radio</td> <td>[7] Select (list)</td>
        <td>[8] Select (menu)</td> <td>[9] Button</td>
        <td>[10] Submit</td> <td>[11] Reset</td></tr>
    </table>
  </div>

<script>
// 这个通用函数将把一个事件的细节添加到上面表单的大 Textarea 元素中,
// 它从各种事件处理程序中调用。
function report(element, event) {
  var elmntname = element.name;
  if (element.type == "select-one" || element.type == "select-multiple"){
    value = " ";
    for(var i = 0; i < element.options.length; i++)
      if (element.options[i].selected)
        value += element.options[i].value + " ";
  }
}

```

```

    else if (element.type == "textarea") value = "...";
    else value = element.value;
    var msg = event.target.id + " " + element.name + " : " + value + "\n";
    var t = document.getElementById("textArea");
    t.value = t.value + msg;
}

// 该函数将给表单中的每个元素添加事件处理程序。
// 它不查看该元素是否支持指定的事件处理程序，只是添加事件处理程序。
// 注意事件处理程序调用了上面的 report() 函数。
// 注意，我们通过把函数赋予 JavaScript 对象的属性定义事件处理程序，而不是通过给 HTML
// 元素的性质赋予字符串来定义事件处理程序。
function addhandlers(f) {
    // 遍历表单中的所有元素
    for(var i = 0; i < f.elements.length; i++) {
        var e = f.elements[i];
        e.onclick = function() { report(this, 'Click'); };
        e.onchange = function() { report(this, 'Change'); };
        e.onfocus = function() { report(this, 'Focus'); };
        e.onblur = function() { report(this, 'Blur'); };
        e.onselect = function() { report(this, 'Select'); };
    }

    // 为三个按钮定义专用的事件处理程序：
    f.clearbutton.onclick = function() {
        this.form.textArea.value= ''; report(this, 'Click');
    }
    f.submitbutton.onclick = function() {
        report(this, 'Click'); return false;
    }
    f.resetbutton.onclick = function() {
        this.form.reset(); report(this, 'Click'); return false;
    }
}

// 最后，通过添加所有可能的事件处理程序来激活表单。
addhandlers(document.everything);
</script>

```

15.3 脚本化表单元素

前面一节列出了 HTML 提供的表单元素，并解释了如何把这些元素嵌入 HTML 文档。本节将介绍在 JavaScript 程序中如何使用这些元素。

15.3.1 命名表单和表单元素

每个表单元素都有 name 性质，如果要将表单提交给服务器端程序，必须在相应的

HTML 标记中设置这一性质。虽然 JavaScript 程序通常对表单提交不感兴趣，但还有一个设置 name 性质的原因，不久你就会看到。

除了上述 name 性质之外，<form> 标记自身还有一个 name 性质。这个性质与表单提交没有任何关系。它的存在不过是为了方便 JavaScript 程序的设计者。如果在 <form> 标记中定义了 name 性质，那么当代表那个表单的 Form 对象被创建时，它除了会作为一个 Document 对象的数组 forms[] 的元素被存储外，还会被存储在一个 Document 对象的个人属性中。这个新定义的属性名就是 name 性质的值。在例 15-1 中，我们采用如下标记定义了一个表单：

```
<form name= 'everything'>
```

这使我们可以使用如下的表达式引用那个表单：

```
document.everything
```

通常你会觉得这样比使用数组表示法要方便得多：

```
document.forms[0]
```

另外，可以使用表单名使代码具有位置独立性，即使文档重排，改变了表单出现的顺序，表单也会正常运行。

注意，、<applet> 和其他 HTML 标记也具有 name 性质，它们的作用与 <form> 的 name 性质相同。不过，在表单中，这种命名风格影响更为深远，因为表单中包含的所有元素都有 name 性质。设置了一个表单元素的 name 性质，就创建了一个引用该元素的 Form 对象的新属性，这个属性的名字就是 name 性质的值。所以，可以使用下面的表达式引用“address”的表单中的“zipcode”元素：

```
document.address.zipcode
```

如果合理地选择了元素名，那么这种语法比使用硬编码（位置依赖）的数组下标要简洁得多：

```
document.forms[1].elements[4]
```

为了使 HTML 表单中的一组 Radio 元素表现出“单选钮”互斥式行为，它们必须具有相同的 name 性质。例如，在例 15-1 中，我们定义了三个 name 性质为“browser”

的 Radio 元素。虽然不是严格要求,但给相关的 Checkbox 元素组定义相同的 name 性质非常常见。对于服务器端的程序设计,这样共享 name 性质很自然,但在客户端则有点难使用。解决办法很简单,当表单中有多个元素具有相同的 name 性质时,JavaScript 就将这些元素存放到一个数组中,这个数组的名字就是 name 性质的值,数组元素的顺序就是它们在文档中出现的顺序。因此,可以使用如下的表达式来引用例 15-1 中的 Radio 对象:

```
document.everything.browser[0]  
document.everything.browser[1]  
document.everything.browser[2]
```

15.3.2 表单元素的属性

所有(或大部分)表单元素都有下列属性。有些元素还有其他特定属性,当我们考虑各种类型的表单元素时将介绍它们。

type

一个只读字符串,标识表单元素的类型。表 15-1 的第三列列出了每个表单元素的 type 属性的值。

form

对包含该元素的 Form 对象的只读引用。

name

由 HTML 的 name 性质指定的只读字符串。

value

一个可读可写的字符串,指定了表单元素包含或表示的“值”。在提交表单时,将把这个字符串发送到 Web 服务器,JavaScript 程序只是偶尔对它有兴趣。对于 Text 元素和 Textarea 元素来说,该属性存放的是用户输入的文本。对于 Button 元素来说,该属性指定了在按钮上显示的文本,有时你可能想从脚本改变该文本,但对于 Radio 元素和 Checkbox 元素,无论如何都不能编辑 value 属性或将它显示给用户。它只是 HTML 的 value 性质设置的字符串,在提交表单时要传递给 Web 服务器。我们将在本章后面讨论不同类别的表单元素时讨论 value 属性。

15.3.3 表单元素的事件处理程序

大多数表单元素支持如下的事件处理程序：

`onclick`

当用户在元素上点击鼠标时触发，对于 `Button` 元素和与之相关的表单元素来说，这个处理程序格外有用。

`onchange`

当用户改变了元素表示的值（通过输入文本或选择一个选项）时触发。`Button` 和其相关元素通常不支持这个事件处理程序，因为它们没有可编辑的值。注意，该事件处理程序不是用户每次在文本框中敲击一个键都会触发。它只在用户改变了一个元素的值，并把输入焦点移到了其他表单元素时才会触发。也就是说，这个事件处理程序的调用说明发生了完整的改变。

`onfocus`

在表单元素收到输入焦点时触发。

`onblur`

在表单元素失去输入焦点时触发。

例15-1说明了如何为表单元素定义这些事件处理程序。这个例子通过在一个较大的 `Textarea` 元素中列出这些事件，在事件发生时报告事件，这使例15-1成了实验表单元素和它们触发的事件处理程序的好方法。

关于事件处理程序，需要知道的重要一点是，在事件处理程序的代码中，关键字 `this` 引用触发该事件的文档元素。由于所有表单元素都有引用包含表单的 `form` 属性，所以表单元素的事件处理程序可以用 `this.form` 引用 `Form` 对象。进一步来说，这意味着表单元素的事件处理程序可以用 `this.form.x` 引用名为 `x` 的兄弟表单元素。

注意本节列出的四个表单元素事件处理程序，它对所有表单元素都格外重要。表单元素还支持（几乎）所有 `HTML` 元素支持的各种事件处理程序（如 `onmousedown`）。参阅第十九章对事件和事件处理程序的完整讨论。

15.3.4 按钮

Button是最常用的表单元素之一,因为它给用户触发脚本的动作提供了一种清晰可见的方法。Button对象没有自己的默认行为,除非它具有onclick(或其他)事件处理程序,否则它在表单中没有用处。Button元素的value属性控制在按钮上显示的文本。在第四代浏览器中,可以设置这一属性来改变出现在按钮上的文本(只是纯文本,而不是HTML)。有时这样做会有用。

注意,超链接提供了与按钮作用一样的onclick事件处理程序,任何按钮对象都可以用一个链接替换,只要该链接在被点击时进行与按钮被点击时相同的操作。在你想使用与图形化的按钮类似的元素时,应该使用按钮。当onclick处理程序触发的动作可以被概念化为“沿着链接前进”时,应该使用链接。

Submit元素和Reset元素与Button元素相似,只是它们有相关的默认动作(提交和重置表单)。因为它们有默认动作,所以即使没有onclick事件处理程序,它们也有用。另一方面,因为它们的默认动作,所以它们对提交给Web服务器的表单比对纯客户端的JavaScript程序更有用。如果onclick处理程序返回false,这两种按钮的默认动作就不会被执行。可以使用Submit元素的onclick处理程序执行表单验证,但用Form对象自身的onsubmit处理程序进行表单验证更常用一些。

在HTML 4中,可以用<button>标记代替传统的<input>标记创建Button、Submit和Reset按钮。<button>标记更灵活一些,因为它不显示由value性质指定的纯文本,而是显示出现在<button>和</button>之间的HTML内容(格式化的文本或图像)。虽然<button>标记创建的Button对象在技术上有别于<input>标记创建的Button对象,但它们的type域值相同,而且行为非常相似。主要的差别是,因为<button>标记不使用value性质定义按钮的外观,所以不能通过设置value属性改变按钮的外观。在本书中,对于<input>标记和<button>标记创建的对象,我们都用术语Button、Submit和Reset来引用。

15.3.5 切换按钮

Checkbox元素和Radio元素都是切换按钮,它们有两种不同的视觉状态,即可以被选中或取消。用户可以通过点击切换按钮来改变它的状态。Radio元素总被组合在相关元素的组中,这些元素具有相同的name性质值。用这种方法创建的Radio元素

是互斥的, 在选中一个元素后, 前面被选中的元素就会被取消。Checkbox 也常用于共享 name 性质的组中, 在使用名字引用这些元素时, 必须记住用名字引用的对象是与元素同名的数组。在例 15-1 中, 我们见过三个名为 “peripherals” 的 Checkbox 对象。在这个例子中, 我们可以用如下代码引用这三个 Checkbox 对象的数组:

```
document.everything.peripherals
```

要单独引用 Checkbox 元素, 必须使用数组下标:

```
document.everything.peripherals[0] // 第一个名为 'peripherals' 的表单元素
```

Radio 元素和 Checkbox 元素都定义了 checked 属性。这个可读可写的布尔值指定元素当前是否被选中。属性 defaultChecked 是只读的布尔值, 它具有 HTML 的 checked 性质值。声明装载页面时该元素是否被选中。

Radio 元素和 Checkbox 元素自身不显示任何文本, 通常显示临近的 HTML 文本 (在 HTML 4 中, 与 <label> 标记关联使用)。这意味着设置 Radio 元素和 Checkbox 元素的 value 属性不会改变它们的外观 (像 Button 元素那样)。虽然可以设置 value 属性, 但这只能改变在提交表单时发送给 Web 服务器的字符串。

在用户点击切换按钮时, Radio 元素或 Checkbox 元素将触发它的 onclick 事件处理程序, 以通知 JavaScript 程序改变状态。较新的 Web 浏览器还能触发这两个元素的 onchange 处理程序。这两个事件处理程序携带的基本信息相同, 只不过 onclick 处理程序具有更强的可移植性。

15.3.6 文本框

Text 元素可能是 HTML 表单和 JavaScript 程序中最常用的元素。它允许用户输入简短的、单行的文本串。它的 value 属性表示用户输入的文本。可以把该属性明确地设置为应该在文本框中显示的文本。在用户输入新文本或编辑已有的文本, 然后通过把输入焦点移出文本框说明输入完毕时, 将触发 onchange 事件处理程序。

Textarea 元素和 Text 元素相似, 只是它允许用户输入 (和用 JavaScript 程序显示) 多行文本。Textarea 元素是由 <textarea> 标记创建的, 使用的语法与创建 Text 元素的 <input> 标记语法稍有不同。不过这两种元素的行为很相似, 虽然从技术上说

Textarea元素没有继承HTMLInputElement元素,但可以这样认为。可以像使用Text元素的value属性和onchange事件处理程序一样使用Textarea元素的这些属性。

Password元素是修改过的Text元素,在用户输入时它显示星号。顾名思义,这对用户输入口令非常有用,使他们不必担心其他人偷窥自己的密码。Password元素像Text元素一样触发它的onchange处理程序,只在使用value属性时有一定约束(或者说是bug)。有些旧浏览器(如Netscape 3)的防止JavaScript读取用户在Password框中输入的值的措施效率很低。在其他浏览器中(如Netscape 4),可以设置value属性,但设置操作不能引起表单元素的外观发生任何改变。注意,Password元素可以保护用户的输入不被偷窥,但在提交表单时,输入值没有加密(除非通过安全的HTTPS连接提交它),所以当它在网络上传递时,便可能被看到。

最后,FileUpload对象用于输入要上载到服务器的文件名。它实质上是与弹出选择文件对话框的内部按钮组合在一起的Text元素。FileUpload元素像Text元素一样有onchange事件处理程序。但与Text元素不同的是,FileUpload元素的value属性是只读的。这可以阻止恶意的JavaScript程序欺骗用户,使他们不能上载不应该共享的文件。

Netscape 4及其后的版本和Internet Explorer 4及其后的版本还定义了onkeypress、onkeydown和onkeyup事件处理程序(但要注意,它们不是DOM标准的一部分)。任何Document对象都可以设置这些处理程序,但当在Text或相关的表单元素中设置它们最有用(在Netscape 4中也很有用),这些元素真正接收键盘输入。从onkeypress或onkeydown事件处理程序返回false可以阻止用户的键盘输入被记录下来。例如,当你想强迫用户只输入数字时,这就非常有用。参阅客户端参考手册部分的HTMLElement参考页和DOM参考手册可以了解更多有关HTML元素都支持的事件处理程序。

15.3.7 Select元素和Option元素

Select元素表示用户可以选择的选项(由Option元素表示)集合。浏览器通常用下拉式菜单或列表框表示Select元素。Select元素可以用两种截然不同的方式操作,type属性的值由它的配置决定。如果<select>标记有multiple性质,就允许用户进行多项选择,这种Select对象的type属性为“多选”。否则,如果没有multiple性质,那么用户只能选择一个选项,type属性为“单选”。

“多选”元素有点像 Checkbox 元素的集合，“单选”元素则有点像 Radio 元素的集合。但 Select 元素不同于切换按钮元素，因为一个 Select 元素表示所有选项的集合。这些选项由 HTML 标记 `<option>` 设置，在 JavaScript 中它们由 Option 对象表示，这些对象存储在 Select 元素的 `options[]` 数组中。因为 Select 元素表示选项的集合，所以它像其他表单元素那样没有 value 属性。不过我们曾经说过，Select 对象存放的每个 Option 对象都定义了 value 属性。

当用户选中或取消一个选项时，Select 元素将触发它的 `onchange` 事件处理程序。对于“单选”型的 Select 元素，可读可写的属性 `selectedIndex` 用数字指定了当前被选中的选项。对于“多选”型的 Select 元素，一个 `selectedIndex` 属性不足以表示被选中的项的整个集合。在这种情况下，要确定选中了哪些选项，必须遍历 `options[]` 数组的所有元素，检查每个 Option 对象的 `selected` 属性的值。

除了 `selected` 属性外，Option 元素还有 `text` 属性，用于指定 Select 元素显示那个选项使用的纯文本串。设置这一属性可以改变显示给用户的文本。`value` 属性也是可读可写的字符串，指定了提交表单时要发送给 Web 服务器的字符串。即使你编写了一个纯粹的客户端程序，并且从不提交表单，`value` 属性（或与它相应的 HTML `value` 性质）仍是一个存储数据的好地方，它可以存储用户选中了一个特殊选项时需要的数据。注意，Option 元素没有定义与表单有关的事件处理程序，而是使用包含 Select 元素的 `onchange` 处理程序。

除了设置 Option 对象的 `text` 属性外，还有其他方法可以动态改变 Select 元素中显示的选项。可以通过把 `options.length` 设置为想要的选项数来截取 Option 元素的数组，把 `options.length` 设置为 0 可以删除所有 Option 对象。假定在名为“address”的表单中有一个名为“country”的 Select 对象，则可以用如下代码删除该元素中的所有选项：

```
document.address.country.options.length = 0; // 删除所有选项
```

可以把 `options[]` 数组的某个元素设置为 `null`，从而在 Select 元素中删除一个 Option 对象。在删除一个 Option 对象后，`options[]` 数组中的位于这个 Option 对象后的元素会被自动前移，以填充空位：

```
// 从 Select 元素中删除一个 Option 对象
// 原来位于 options[10] 的 Option 对象移到 options[10]...
document.address.country.options[10] = null;
```

最后, Option 元素定义了构造函数 Option(), (在 JavaScript 1.1 和其后的版本中) 可以动态创建新的 Option 元素, 把它们附加在 options[] 数组的尾部可以给 Select 元素增加新选项。例如:

```
// 创建新的 Option 对象
var zaire = new Option('zaire',    // text 属性
                      'zaire',    // value 属性
                      false,      // defaultSelected 属性
                      false);     // selected 属性
// 把它附加到 Select 元素的 options 数组, 在 select 元素中显示它
var countries = document.address.country; // 获取 Select 对象
countries.options[countries.options.length] = zaire;
```

在 HTML 4 中, 可以用 <optgroup> 标记对 Select 元素中的相关选项分组。<optgroup> 标记具有 label 性质, 它指定显示在 Select 元素中的文本。尽管 <optgroup> 标记可见, 但是用户不能选择它, HTMLOptGroupElement 对象不会出现在 Select 元素的 options[] 数组中。

15.3.8 Hidden 元素

顾名思义, Hidden 元素在表单中不可见。它的作用是在提交表单时把任意的文本发送给服务器。服务器端的程序用它来保存表单提交时返回给它们的状态信息。由于 Hidden 元素设有可视化的外观, 所以它不能生成事件, 没有事件处理程序。value 属性允许你读写与 Hidden 元素相关的文本, 但客户端 JavaScript 程序通常不用 Hidden 元素。

15.3.9 Fieldset 元素

HTML 4 标准给出现在表单中的元素集合添加了 <fieldset> 标记和 <label> 标记。在 IE 5 和其后的版本中, 把 <fieldset> 标记放到一个表单中, 会将相应的对象添加到表单的 elements[] 数组中。Fieldset 元素不像其他表单元素那样脚本化, 它们的对象没有 type 属性。因此, Fieldset 对象的表示在 elements[] 数组中看起来像错误的设计方案。因为 <label> 标记不能使相应的对象被添加到 elements[] 数组中, 所以这一点显得尤其正确。Mozilla 和 Netscape 6 浏览器选择了采用 Microsoft 公司的这一做法, 以便与 IE 兼容。

这意味着, 如果你定义了含有 fieldset 元素的表单, 那么在近来支持 HTML 4 的浏

浏览器中，elements[]数组的内容与其在旧的不支持 HTML 4 的浏览器中的内容不同。在这种情况下，elements[]数组中基于位置的数字下标的用法不可移植，所以应该为所有表单元素定义 name 性质，而且用名字引用它们。

15.4 表单验证示例

我们将用一个例子结束有关表单的讨论，这个例子说明了我们已经讨论过的几个概念。例 15-2 说明了如何使用 Form 对象的事件处理程序 onsubmit 来执行输入验证，以便能够在表单含有缺失的或无效的数据时通知用户并且阻止表单的提交。研究过这个例子之后，你可以再研究一下例 1-3，就是我们本书开始使用的表单示例程序。现在你已经是 JavaScript 高手了，那段代码看来可能更有意义了。

例 15-2 定义了一个 verify() 函数，它可以作为一个通用的表单验证器。该函数可以检测必需的输入域是否为空。另外，它还可以检测一个数字值是否真的是数字，而且还可以检测这个数字是否在指定的范围之内。verify() 靠一个表单元素的 type 属性来判断该元素的类型。它还可以根据用户定义的属性来区分可有可无的域和必需的域，并且能够根据这些属性设置数字域的允许范围。注意该函数是如何读取输入域的 value 属性的，又是如何在报告错误的时候使用一个域的 name 属性的。

图 15-2 展示了一个示例表单，它就使用了这种验证模式，其中有错误信息，这些信息是在用户试图提交填写不正确的表单时显示出来的。

例 15-2: 执行表单验证

```
<script language="JavaScript1.1">
// 工具函数，如果一个字符串只含有空白符，它返回 true。
function isblank(s) {
    for(var i = 0; i < s.length; i++) {
        var c = s.charAt(i);
        if ((c != ' ') && (c != '\n') && (c != ' ')) return false;
    }
    return true;
}

// 这是执行表单验证的函数，用 onsubmit 事件处理程序调用。
// 这个处理程序应该返回该函数返回的值。
function verify(f) {
    var msg;
    var empty_fields = '';
    var errors = '';
```

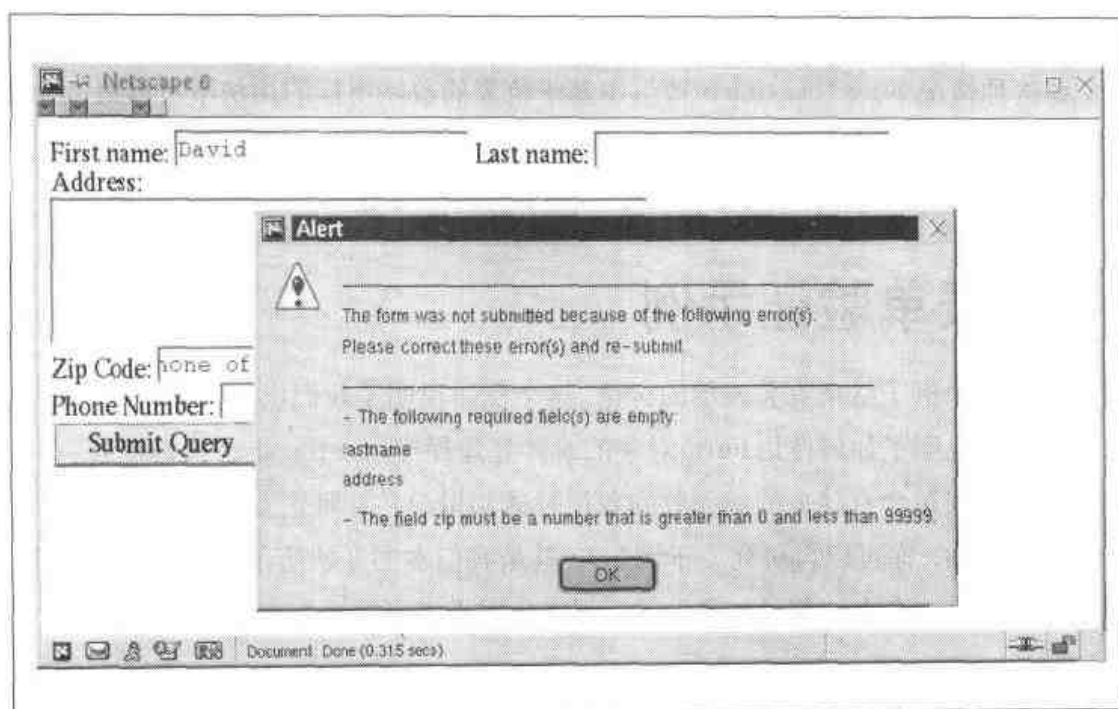


图 15-2: 没有通过验证的表单

```

// 遍历表单元素, 查找所有没有定义 optional 属性的 Text 和 Textarea 元素。
// 然后检查空的域, 生成它们的列表。
// 如果这些元素定义了 "min" 或 "max" 属性, 验证它们是数字, 而且在范围的右边界之内。
// 如果该元素定义了 "numeric" 属性, 验证它是否是数字, 但不检查它的范围。
// 收集有错误的域的错误消息。
for(var i = 0; i < f.length; i++) {
    var e = f.elements[i];
    if ((e.type == "text") || (e.type == "textarea") && !e.optional) {
        // 首先检查该域是否为空。
        if ((e.value == null) || (e.value == "") || isblank(e.value)) {
            empty_fields += '\n' + e.name;
            continue;
        }
    }

    // 下面检查假定为数字的域。
    if (e.numeric || (e.min != null) || (e.max != null)) {
        var v = parseFloat(e.value);
        if (!isNaN(v) ||
            ((e.min != null) && (v < e.min)) ||
            ((e.max != null) && (v > e.max))) {
            errors += "- The field " + e.name + " must be a number";
            if (e.min != null)
                errors += " that is greater than " + e.min;
            if (e.max != null && e.min != null)
                errors += " and less than " + e.max;
            else if (e.max != null)
                errors += " that is less than " + e.max;
            errors += ".\n";
        }
    }
}

```

```

    }
    }
}

// 现在, 如果有错误, 则显示错误消息, 并且返回 false, 阻止表单提交。
// 否则, 返回 true。
if (!empty_fields && !errors) return true;

msg = ' _____ \n\n'
msg += "The form was not submitted because of the following error(s).\n";
msg += "Please correct these error(s) and re submit.\n";
msg += ' _____ \n\n'

if (empty_fields) {
    msg += " The following required field(s) are empty:"
        + empty_fields + '\n';
    if (errors) msg += '\n';
}
msg += errors;
alert(msg);
return false;
}
</script>

```

下面是检测我们的验证函数的示例表单。注意, 我们从 onsubmit 事件处理程序调用 verify() 函数, 返回该函数的返回值, 还要注意, verify() 进行验证需要设置 Form 对象的属性, 我们用 onsubmit 处理函数设置它们。

```

<form onsubmit='
    this.firstname.optional = true;
    this.phonenumber.optional = true;
    this.zip.min = 0;
    this.zip.max = 99999;
    return verify(this);
">

First name: <input type="text" name="firstname">
Last name: <input type="text" name="lastname"><br>
Address:<br><textarea name="address" rows="4" cols="40"></textarea><br>
Zip Code: <input type="text" name="zip"><br>
Phone Number: <input type="text" name="phonenumber"><br>
<input type="submit">
</form>

```

第十六章

脚本化 cookie

Document 对象含有一个 cookie 属性，我们在第十四章中并没有对它进行讨论。表面上看来，这个属性不过是一个简单的字符串值，但实际上它控制了 Web 浏览器的一个重要特性，因此有足够的理由用完整的一章对它进行讨论。

16.1 cookie 概览

cookie 是 Web 浏览器存储的少量命名数据，它与某个特定的网页或网站关联在一起（注 1）。cookie 用来给 Web 浏览器提供内存，以便脚本和服务端程序可以在一个页面中使用另一个页面的输入数据，或者在用户离开页面并返回时恢复用户的优先级及其他状态变量。cookie 最初是为 CGI 程序设计的，而且在最低级别上作为 HTTP 协议的扩展。因为 cookie 数据可以自动地在 Web 浏览器和 Web 服务器之间传递，所以位于服务器端的 CGI 脚本可以读写存储在客户端的 cookie 值。我们还将看到，JavaScript 能够用 Document 对象的 cookie 属性对 cookie 进行操作。

注 1: “cookie”这个名字并没有什么特殊含义，但它的使用并非没有先例。在某个无名的计算历史记录中，已经使用“cookie”或“magic cookie”来引用一小块数据了，尤其是一块有特权的数据或保密的数据，以便提供身份或允许访问，这有些类似于口令。在 JavaScript 中，cookie 用来保存状态，并建立浏览器某种类型的身份。JavaScript 中的 cookie 不使用任何加密技术，因而也没有安全性。

cookie是一个字符串属性，可以用它对当前网页的cookie进行读操作、创建操作、修改操作和删除操作。尽管cookie最初呈现出来的不过是一个可读可写的普通字符串属性，但是它的行为却要复杂得多。当对cookie进行读操作时，可以得到一个字符串，这个字符串包含了应用到当前文档的所有cookie的名字和值。通过设置属性cookie的值可以创建、修改或删除一个cookie。本章后面的小节会详细解释这些操作是如何进行的。但是，要有效地使用属性cookie，还需要对cookie以及它们的工作过程做进一步了解。

除了名字与值外，每个cookie都有四个可选的性质，分别控制它的生存期、可见性和安全性。第一个性质是`expires`，它指定了cookie的生存期。默认情况下，cookie是暂时存在的，它们存储的值只在浏览器会话期间存在。当用户退出浏览器后，这些值也就丢失了。如果想让一个cookie存在的时间超过一个浏览会话期，可以用`expires`性质指定一个终止日期，这样就会使浏览器把cookie保存到一个本地文件中，以便用户下次访问这个网页时能够再将它读出来。一旦超过了终止日期，那个cookie就会自动地从cookie文件中删除掉。

cookie的第二个性质是`path`，它指定了与cookie关联在一起的网页。默认情况下，cookie会和创建它的网页以及与这个网页处于同一个目录下的网页和处于该目录的子目录下的网页关联在一起。例如，如果一个cookie由位于`http://www.acme.com/catalog/index.html`的网页创建，那么它对位于`http://www.acme.com/catalog/order.html`和位于`http://www.acme.com/catalog/widgets/index.html`的网页也可见。但是对位于`http://www.acme.com/about.html`的网页就不可见了。

这种默认的可见性通常就是你所需要的。不过有时你可能想使用整个多页面网站中的cookie值，而不管这个cookie是由哪个页面创建的。例如，假定用户在一个页面的表单中输入了他的电子邮件地址，你想保存那个地址，作为他下次返回这个页面时的默认值，此外还想将这个地址作为另一个完全无关的表单的默认值，这个表单位于另一个要求用户输入账单地址的页面中。要做到这一点，需要指定cookie的`path`性质。然后，来自同一个网络服务器的网页，只要它的URL中含有指定的路径，就可以共享这个cookie。例如，假定一个由`http://www.acme.com/catalog/widgets/index.html`设置的cookie将自己的路径设置成了`"/catalog"`，那么这个cookie对位于`http://www.acme.com/catalog/order.html`的页面也是可见的。如果将这个cookie的路径设置成了`"/"`，那么它就对位于`www.acme.com`服务器上的所有网页都可见。

默认情况下，只有和设置 cookie 的网页来自同一 Web 服务器的页面才能访问这个 cookie。但是，大的网站可能需要由多个 Web 服务器共享 cookie。例如，位于 *order.acme.com* 的服务器就可以读取 *catalog.acme.com* 设置的 cookie 值。这里就引入了第三个 cookie 性质 domain。假定由位于 *catalog.acme.com* 的页面创建的 cookie 把自己的 path 性质设置成了 “/”，把 domain 性质设置成了 “.acme.com”，那么所有位于 *catalog.acme.com* 的网页和所有位于 *orders.acme.com* 网页以及所有位于 *acme.com* 域的其他服务器上的网页都能够访问这个 cookie。如果没有设置 cookie 的 domain 性质，该性质的默认值就是创建 cookie 的网页所在的服务器的主机名。注意，不能将一个 cookie 的域设置成服务器所在的域之外的域。

cookie 的第四个性质，也是最后一个性质是名为 secure 的布尔值，它指定了在网络上如何传输 cookie 值。默认情况下，cookie 是不安全的，也就是说，它们是通过一个普通的、不安全的 HTTP 连接传输的。但是如果将 cookie 标记为安全的，那么它将只在浏览器和服务器通过 HTTPS 或其他的安全协议连接在一起时才被传输。

注意，expires、path、domain 和 secure 都是 cookie 的性质，而不是 JavaScript 对象的属性。在本章后面的小节中，我们将会看到如何设置这些性质。

如果你对 cookie 使用的完整技术细节感兴趣，可以查阅 http://www.netscape.com/newsref/std/cookie_spec.html。这里提供的文档是对 HTTP 的 cookie 的正式说明，它含有的基本介绍比之 JavaScript 的程序设计来说更适用于 CGI 的程序设计。下面的几节讨论了在 JavaScript 中如何设置和查询 cookie 的值以及如何设置 cookie 的 expires、path、domain 和 secure 性质。

16.2 cookie 的储存

要把一个暂时的 cookie 值与当前文档关联起来，只需要将 cookie 属性以下面的形式设置成一个字符串即可：

```
name=value
```

例如：

```
document.cookie = "version=" + escape(document.lastModified);
```

当下次读取 cookie 属性时, 存储的 name/value 属性对就会被添加到文档的 cookie 列表中。cookie 值不能含有分号、逗号或空白符。因此, 需要使用 JavaScript 的函数 `escape()` 在把值存入 cookie 之前对它进行编码。如果使用了 `escape()` 函数, 那么在读 cookie 值时, 就必须使用相应的 `unescape()` 函数。

用上述方法编写的 cookie 只能在当前的 Web 浏览会话期中存活, 当用户退出浏览器时它就会丢失。要创建可以在多个浏览器会话期中持续存在的 cookie, 可以设置性质 `expires`, 给它加上一个终止日期。可以采用下面的表达式来设置属性 cookie:

```
name=value; expires=date
```

使用这样的形式设置终止日期时, `date` 采用的日期规范应该是方法 `Date.toGMTString()` 编写的格式。例如, 要创建一个能够持续存在一年的 cookie, 可以采用如下代码:

```
var nextyear = new Date();
nextyear.setFullYear(nextyear.getFullYear() + 1);
document.cookie = "version=" + document.lastModified +
    "; expires=" + nextyear.toGMTString();
```

同样, 也可以把如下形式的字符串在将 cookie 值写入 cookie 属性之前附加到 cookie 值中, 这样就可以设置性质 `path`、`domain` 和 `secure`:

```
; path=path
; domain=domain
; secure
```

要改变一个 cookie 的值, 使用同一个名字和新的值再设置一次 cookie 值即可。新的值可以是任何适合于 `expires`、`path` 和其他性质的值。要删除一个 cookie, 只需要使用同一个名字和一个任意的值以及一个已经超过了的终止日期再设置它一次即可。注意, 由于浏览器不必立刻删除过期的 cookie, 所以即使过了终止日期, cookie 还可以保存在浏览器的 cookie 文件中。

16.2.1 cookie 的局限性

cookie 主要用于少量数据的不经常存储。它们既不是一种通用的通信机制, 也不是一种通用的数据传输机制, 所以使用它们时一定要适度。Web 浏览器需要保存的 cookie 总数不过 300 个, 为每个 Web 服务器保存的 cookie 数不过 20 个 (是对整个

服务器而言,而不仅仅指服务器上的页面或站点),而且每个 cookie 保存的数据不能超过4千字节(即名字和值的总量不能超过4千字节的限制)。这些限制中最为严格的是每个服务器保存的 cookie 数不能超过20个,为避免超过这个限制,为每个想要保存的状态变量都设置一个单独的 cookie 就不是个好主意。相反,你应该将多个相关的状态变量存储到一个 cookie 中。本章后面的例16-1展示了这样一个例子。

16.3 cookie 的读取

当在一个JavaScript表达式中使用 cookie 属性时,它返回的值是一个字符串,这个字符串存放的是当前文档应用的所有 cookie (注2)。它是一个 *name=value* 对的列表,数对之间用分号隔开,其中 *name* 是 cookie 的名字, *value* 是它的字符串值。这个值不包括已经设置的 cookie 属性。要特定的已命名的 cookie 的值,可以使用方法 `String.indexOf()` 和 `String.substring()`,或者可以使用方法 `String.split()` 将字符串分割成单独的 cookie。

如果已经从 cookie 属性中提取出了一个 cookie 的值,就必须基于那个 cookie 创建器所使用的格式或编码方法来解释得到的值。例如, cookie 可能存为多个信息段,各段之间用冒号分隔。在这种情况下,就要使用适当的字符串方法将各个段的信息提取出来。如果给 cookie 值编码时使用了函数 `escape()`,那么不要忘记还要使用函数 `unescape()`。

下面的代码说明了如何对属性 cookie 进行读操作,如何从中提取出一个独立的 cookie 以及如何使用那个 cookie 的值:

```
// 读 cookie 属性。这将返回文档的所有 cookie。
var allcookies = document.cookie;

// 查找名为 "version" 的 cookie 的开始位置。
var pos = allcookies.indexOf('version=');

// 如果找到了具有该名字的 cookie, 那么提取并使用它的值。
if (pos != -1) {
```

注2: 在 Internet Explorer 3 中, cookie 属性只用于 Document 对象,该对象是使用 HTTP 协议获得的。从本地文件系统或通过其他协议获得的 Document (如 FTP) 没有与之相关联的 cookie。

```
var start = pos + 8; // cookie 值的开始。
var end = allcookies.indexOf('";', start); // cookie 值的结尾。
if (end == -1) end = allcookies.length;
var value = allcookies.substring(start, end); // 提取 cookie 的值。
value = unescape(value); // 对它解码。

// 既然我们已经有了 cookie 的值，就可以使用它。
// 在这种情况下，cookie 之前被设置为文档的修改日期，
// 所以我们可以用它查看上次用户访问后文档是否改变了。
if (value != document.lastModified)
    alert('This document has changed since you were last here');
}
```

注意，当对 cookie 属性进行读操作时，得到的字符串不包含任何有关各种 cookie 属性的信息。虽然 cookie 属性允许你设置这些性质，但是却不允许你读取它们的值。

16.4 cookie 示例

例 16-1 综合了我们讨论过的 cookie 的各个方面的知识。首先，这个例子定义了一个 cookie 类。在创建一个 cookie 对象时，就为这个 cookie 指定了一个 Document 对象和一个名字，另外还可以指定它的终止日期、路径、域以及说明它是否安全的布尔值。创建了一个 cookie 对象之后，就可以设置这个对象的任意字符串属性，这些属性的值就是存储在 cookie 中的值。

cookie 类定义了三种方法。方法 store() 可以遍历 cookie 对象的所有由用户定义的属性，并且将这些属性的名字和值连接到一个字符串，这个字符串就是 cookie 的值。方法 load() 可以读取 Document 对象的 cookie 属性以得到那个文档的所有 cookie 值。首先，它要搜索读到的字符串以找到命名了的 cookie 的值，然后将这个值解析成独立的名字和值，并将它们存储为 cookie 对象的属性。cookie 对象的最后一个方法是 remove()，它可以把指定的 cookie 从文档中删除。

定义了类 cookie 后，例 16-1 展示了一种使用 cookie 的方法，这种方法既实用又简洁。虽然这段代码有点复杂，但是还是值得仔细研究。你可以从例子结尾处的测试程序开始研究，它说明了 cookie 类的一种典型用法。

例 16-1：使用 cookie 时的一个工具类

```
<script language="JavaScript1.1">
```

```

// 构造函数：用指定的名字和可选的性质为指定的文档创建一个 cookie 对象。
// 参数：
//   document: 保存 cookie 的 Document 对象，必需的。
//   name:      指定 cookie 名的字符串，必需的。
//   hours:     一个可选的数字，指定从现在起到 cookie 过期的小时数。
//   path:      一个可选的字符串，指定了 cookie 的路径性质。
//   domain:    一个可选的字符串，指定了 cookie 的域性质。
//   secure:    一个可选的布尔值，如果为 true，需要一个安全的 cookie。
//
function Cookie(document, name, hours, path, domain, secure)
{
    // 该对象所有预定义的属性都以 '$' 开头，
    // 这是为了与存储在 cookie 中的属性值区别开。
    this.$document = document;
    this.$name = name;
    if (hours)
        this.$expiration = new Date((new Date()).getTime() + hours*3600000);
    else this.$expiration = null;
    if (path) this.$path = path; else this.$path = null;
    if (domain) this.$domain = domain; else this.$domain = null;
    if (secure) this.$secure = true; else this.$secure = false;
}

// 该函数是 cookie 对象的 store() 方法。
Cookie.prototype.store = function () {
    // 首先遍历 cookie 对象的属性，并且将 cookie 值连接起来。
    // 由于 cookie 将等号和分号作为分隔符，
    // 所以我们使用冒号和 & 来分隔存储在单个 cookie 值中的状态变量。
    // 注意，我们对每个状态变量的值进行了转义，以防它含有标点符号或其他非法字符。
    var cookieval = '';
    for (var prop in this) {
        // 忽略所有名字以 $ 开头的属性和所有方法。
        if (!prop.charAt(0) == '$' || (typeof this[prop]) == 'function')
            continue;
        if (cookieval != '') cookieval += '&';
        cookieval += prop + ':' + escape(this[prop]);
    }

    // 既然我们已经有了 cookie 值，就可以连接完整的 cookie 串，
    // 其中包括名字和创建 cookie 对象时指定的各种性质。
    var cookie = this.$name + '=' + cookieval;
    if (this.$expiration)
        cookie += '; expires=' + this.$expiration.toGMTString();
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    if (this.$secure) cookie += '; secure';
    // 下面设置 Document.cookie 属性来保存 cookie。
    this.$document.cookie = cookie;
}

// 该函数是 cookie 对象的 load() 方法。
Cookie.prototype.load = function () {
    // 首先得到属于该文档的所有 cookie 的列表。

```

```

// 通过读 Document.cookie 属性可以实现这一点
var allcookies = this.$document.cookie;
if (allcookies == "") return false;

// 下面从该列表中提取已命名的 cookie。
var start = allcookies.indexOf(this.$name + '=');
if (start == -1) return false; // 该页未定义 cookie
start = this.$name.length + 1; // 跳过名字和等号。
var end = allcookies.indexOf(';', start);
if (end == -1) end = allcookies.length;
var cookieval = allcookies.substring(start, end);

// 既然我们已经提取出了已命名的 cookie 的值，
// 就可以把它分割存储到状态变量名和值中。
// 名字/值对由 & 分隔，名字和值之间则由逗号分隔。
// 我们使用 split() 方法解析所有数据。
var a = cookieval.split('&'); // 分割成名字/值对。
for(var i=0; i < a.length; i++) // 把每对值存入数组。
    a[i] = a[i].split(':');

// 既然我们已经解析了 cookie 值，
// 就可以设置 cookie 对象中的状态变量的名字和值。
// 注意我们对属性值调用了 unescape() 方法，因为存储它们时调用了 escape() 方法。
for(var i = 0; i < a.length; i++) {
    this.a[i][0] = unescape(a[i][1]);
}

// 完成了，返回成功代码。
return true;
}

// 该函数是 cookie 对象的 remove() 方法。
Cookie.prototype.remove = function() {
    var cookie;
    cookie = this.$name + '=';
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    cookie += '; expires=Fri, 02-Jan-1970 00:00:00 GMT';
    this.$document.cookie = cookie;
}

//-----
// 上面的代码是 Cookie 类的定义。下面的代码是使用该类的一个实例。
//-----

// 创建 cookie，用来保存该 Web 页的状态。
// 由于我们使用了默认路径，所以同一目录中的所有 Web 页都可以访问该 cookie。
// 因此，它的名字在这些页中应该是惟一的。
// 注意我们设置的过期时间是 10 天后。
var visitordata = new Cookie(document, "name_color_count_state", 240);

// 首先读 cookie 中存储的数据。

```



```
// 如果 cookie 没有被定义, 或者它没有我们需要的数据, 就向用户查询数据。
if (!visitordata.load() || !visitordata.name || !visitordata.color) {
    visitordata.name = prompt('What is your name:', '');
    visitordata.color = prompt('What is your favorite color:', '');
}

// 跟踪这个用户访问了多少次该页面:
if (visitordata.visits == null) visitordata.visits = 0;
visitordata.visits++;

// 保存 cookie 的值, 即使它们已经被保存过了,
// 以便为这位经常访问的用户将过期日期重置为 10 天,
// 此外, 注意再次保存它们以保存更新的访问状态变量,
visitordata.store();

// 下面我们可以使用读到的状态变量:
document.write('<font size="7" color=" ' + visitordata.color + ' "> ' +
    'Welcome, ' + visitordata.name + '! ' +
    '</font> ' +
    '<p>You have visited ' + visitordata.visits + ' times.'');

</script>

<form>
<input type='button' value='Forget My Name' onclick='visitordata.remove();'>
</form>
```

第十七章

文档对象模型

文档对象模型 (Document Object Model, DOM) 是表示文档 (如 HTML 文档) 和访问、操作构成文档的各种元素 (如 HTML 标记和文本串) 的应用程序接口 (API)。启用 JavaScript 的 Web 浏览器都定义了文档对象模型。例如, Web 浏览器 DOM 规定, 通过 Document 对象的 `forms[]` 数组可以访问 HTML 文档中的表单。

在本章中, 我们将讨论 W3C DOM, 它是由 World Wide Web 委员会定义的标准文档对象模型, 由 Netscape 6 和 Internet Explorer 5 及 6 实现 (至少部分地实现)。DOM 标准 (注 1) 是传统 Web 浏览器 DOM 的所有特性的超集。它以树形结构表示 HTML 文档 (和 XML 文档), 定义了遍历这个树和检查、修改树的节点的方法和属性。该标准的其他部分为文档的各个节点定义了事件处理程序、使用文档的样式表和操作文档邻接范围的方法。

本章先概括介绍 DOM 标准, 接着介绍了该标准用于处理 HTML 文档的核心部分, 此后简要地解释了 Internet Explorer 4 和 Netscape 4 中与 DOM 相似的特性。结尾概览了 DOM 标准两个可选的部分, 这两个部分与核心部分紧密相关。本章之后的章节介绍了使用样式表和事件的高级 DOM 特性。

注 1: 从技术上来说, W3C 将其看做“推荐标准”。但这些推荐标准与国际标准的目的和权威性机同, 所以本书称其为“标准”。

17.1 DOM 概览

虽然 DOM API 不太复杂，但在开始讨论使用 DOM 的程序设计方法前，需要了解一些 DOM 体系结构。

17.1.1 把文档表示为树

在 DOM 中，HTML 文档的层次结构被表示为树形结构。树的节点表示文档中的各种内容。HTML 文档的树形表示主要包含表示元素或标记（如 `<body>` 和 `<p>`）的节点和表示文本中的节点构成。HTML 文档还含有表示 HTML 注释的节点（注 2）。考虑下列简单的 HTML 文档：

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

这个文档的 DOM 的树形表示如图 17-1 所示。

如果你对计算机程序设计中的树形结构还不熟悉，那么了解一些术语会有所帮助，这些术语是从它们的家族树中借用的。直接位于一个节点之上的节点是该节点父节点（parent）。直接位于一个节点之下的节点是该节点的子节点（children）。位于同一层次，具有相同父节点的节点是兄弟节点（sibling）。一个节点的下一个层次的节点集合是那个节点的后代（descendant）。一个节点的父节点、祖父节点及其他所有位于它之上的节点都是那个节点的祖先（ancestor）。

注 2：DOM 还可以用于表示 XML 文档，它的语法比 HTML 文档的语法复杂得多，这类文档的树形表示包含表示 XML 实体引用的节点、表示处理指令的节点以及表示 CDATA 段的节点，等等。大多数客户端 JavaScript 程序设计者不需要用 DOM 表示 XML 文档，尽管在 DOM 参考手册部分列出了特定的 DOM 特性，但本章的重点不在此。

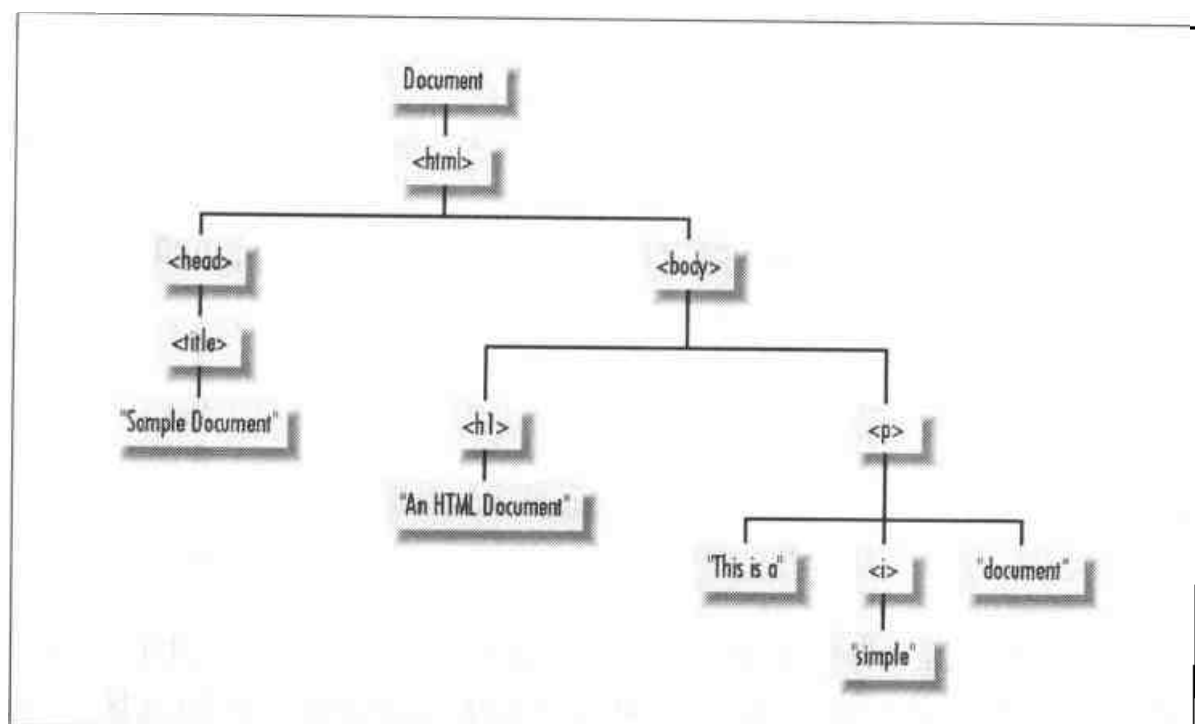


图 17-1: 一个HTML文档的树形表示

17.1.2 节点

图 17-1 显示的 DOM 树形结构是各种类型的 Node 对象的树。Node 接口（注 3）为遍历和操作树定义了属性和方法。Node 对象的 `childNodes` 属性将返回子节点的列表，`firstChild`、`lastChild`、`nextSibling`、`previousSibling` 和 `parentNode` 属性提供了遍历树的方法。`appendChild()`、`removeChild()`、`replaceChild` 和 `insertBefore()` 方法使你能给文档树添加节点或从文档树中删除节点。在本章后面的小节中可以看到这些属性和方法的使用的例子。

17.1.2.1 节点的类型

文档树中不同类型的节点由特定的 Node 子接口表示。每个 Node 对象都有 `nodeType` 属性，这些属性指定节点的类型。例如，如果一个节点的 `nodeType` 属性等于常量 `Node.ELEMENT_NODE`，你就知道这个 Node 对象还是一个 Element 对象，可以对它

注 3: DOM 标准定义了接口，没有定义类。如果你不熟悉面向对象的程序设计方法中的术语“接口”，那么你可以将它看作一种抽象。我们将在本节的后面详细讨论两者之间的差别。

使用Element接口定义的所有方法和属性。表17-1列出了HTML文档中常见的节点类型和它们的nodeType值。

表 17-1: 常用节点类型

接口	nodeType 常量	nodeType 值
Element	Node.ELEMENT_NODE	1
Text	Node.TEXT_NODE	3
Document	Node.DOCUMENT_NODE	9
Comment	Node.COMMENT_NODE	8
DocumentFragment	Node.DOCUMENT_FRAGMENT_NODE	11
Attr	Node.ATTRIBUTE_NODE	2

DOM树的根节点是个Document对象。该对象的documentElement属性引用表示文档根元素的Element对象。对于HTML文档来说,这是在文档中隐式或显式出现的<html>标记(Document节点除了根元素外还可能还有其他子节点,如Comment节点)。大部分DOM树由表示标记(如<html>和<i>)的Element对象和表示文本串的Text对象构成。如果文档解析器保留了注释,那么这些注释在DOM树中由Comment对象表示。图17-2展示了上述这些和其他DOM核心接口的部分类层次。

17.1.2.2 性质

用Element接口的getAttribute()方法、setAttribute()方法和removeAttribute()方法可以查询、设置并删除一个元素的性质(如标记的src性质和width性质)。

另一种使用性质的方式是调用getAttributeNode()方法(但该方法使用起来不够方便),它将返回一个表示性质和它的值的Attr对象(使用这种方法的原因之一是Attr接口定义了specified属性,使你可以判断文档中是否直接指定了该性质,或判断它的值是否是默认值)。图17-2中出现了Attr接口,它是一种节点类型。但要注意,Attr对象不出现在元素的childNodes[]数组中,不像Element和Text节点那样是文档树的一部分。DOM标准允许通过Node接口的attributes[]数组访问Attr节点,但Microsoft公司的Internet Explorer定义了不兼容的attributes[]数组,要可移植地使用这种特性是不可能的。

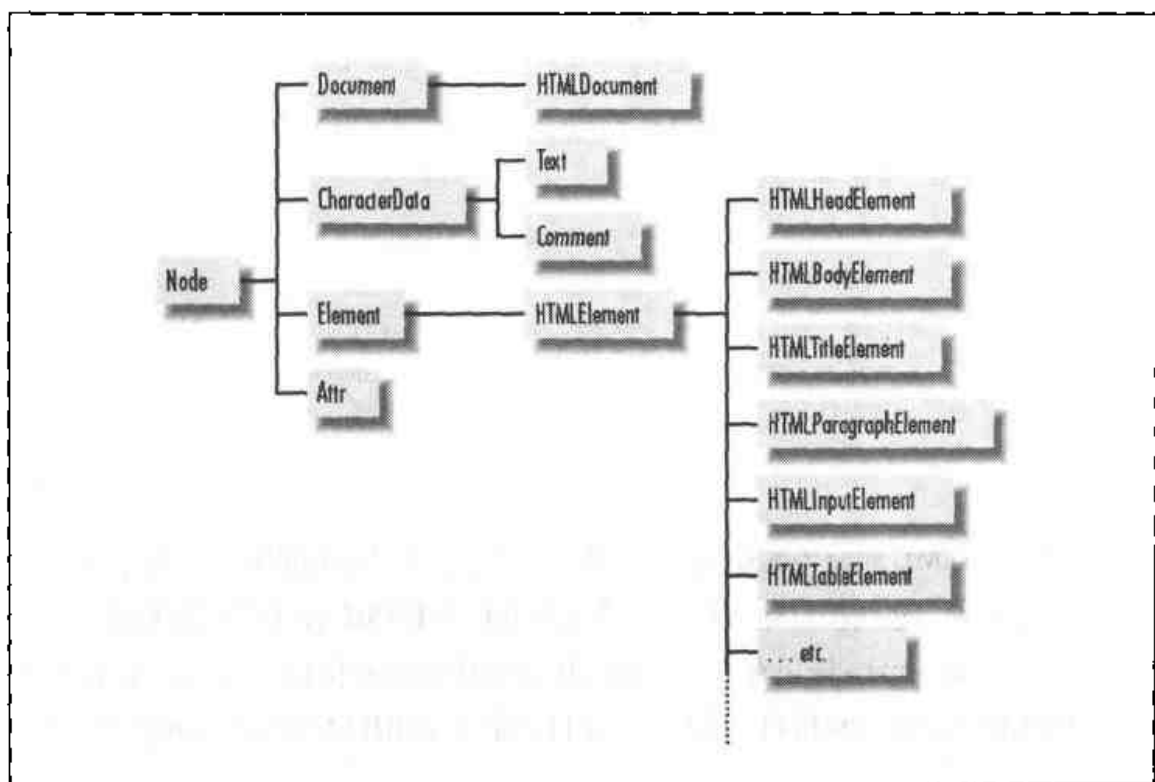


图 17-2: 核心 DOM API 的部分层次

17.1.3 DOM HTML API

DOM 标准可以与 XML 文档和 HTML 文档一起使用。DOM 的核心 API (Node、Element、Document 和其他接口) 相对通用一些, 可以应用于这两种类型的文档。DOM 标准还包括 HTML 文档专有的接口。从图 17-2 中可以看到, HTMLDocument 是 HTML 专有的 Document 接口的子接口, HTMLElement 是 HTML 专有的 Element 接口的子接口。另外, DOM 为许多 HTML 元素定义了标记专有的接口。这些标记专有的接口 (如 HTMLBodyElement 和 HTMLTitleElement) 通常定义了镜像那个 HTML 标记性质的属性集合。

HTMLDocument 接口定义了 W3C 标准化之前的浏览器支持的各种文档属性和方法。其中包括 location 属性、forms[] 数组和 write() 方法, 第十三章、第十四章和第十五章介绍过这些方法。

HTMLElement 接口定义了 id、style、title、lang、dir 和 className 属性。用这些属性访问 HTML 标记的 id、style、title、lang、dir 和 class 性质非常方便。

除了这6个性质外，表 17-2 中列出的大部分 HTML 标记不接受其他的性质，所以 HTML 接口完全可以表示它们。

表 17-2: 简单的 HTML 标记

<abbr>	<acronym>	<address>		<bdo>
<b_g>	<center>	<cite>	<code>	<dd>
<dfn>	<dl>		<i>	<kbd>
<noframes>	<noscript>	<s>	<samp>	<small>
	<strike>		<sub>	<sup>
<tt>	<u>	<var>		

DOM 标准的 HTML 部分为其他所有 HTML 标记都定义了相应的接口。对于大部分 HTML 标记来说，这些接口除了提供一套镜像它们的 HTML 性质的属性集合外，什么都不做。例如， 标记有对应的 HTMLUListElement 接口，<body> 标记有对应的 HTMLBodyElement 接口。因为这些接口只定义了由 HTML 标准标准化的属性，所以本书没有详细介绍它们。你可以安全地假定表示特殊 HTML 标记的 HTML 接口对象具有该标记的所有标准性质所使用的属性（请参阅下一节介绍的命名规则）。注意，DOM 标准为 HTML 性质定义属性是为了给脚本编写者提供方便。查询和设置性质值的通用方法（可能是首选方法）是调用 Element 对象的 `getAttribute()` 方法和 `setAttribute()` 方法。

除了镜像 HTML 性质值的属性外，HTML DOM 标准定义的某些接口还定义了其他的属性和方法。例如，HTMLInputElement 接口定义了 `focus()` 和 `blur()` 方法，HTMLFormElement 接口定义了 `submit()` 和 `reset()` 方法及 `length` 属性。这样的方法和属性通常在 DOM 标准化之前就出现了，为了向后兼容现有的版本，它们已经成为 DOM 标准的一部分。DOM 参考手册部分说明了这些接口。在客户端的参考手册部分还可以找到这些接口的已有版本的信息，通常在先于 DOM 标准化的名字下可以找到这些信息。例如，可以在客户端参考手册的“Form”和“Input”条目下找到有关 HTMLFormElement 和 HTMLInputElement 的信息。

17.1.3.1 HTML 命名规则

在使用 HTML 专有的 DOM 标准时，应该注意一些简单的命名规则。HTML 专有的接口的属性应该以小写字母开头。如果属性名由多个单词构成，第二个单词以及接

下来的每个单词的首字母都要大写。因此, `<input>` 标记的 `maxLength` 性质将被转换成 `HTMLInputElement` 的 `maxLength` 属性。

当 HTML 性质名与 JavaScript 关键字发生冲突时, 应在性质名前加前缀 “html” 来避免冲突。因此, `<label>` 标记的性质 `for` 将被转换成 `HTMLLabelElement` 的属性 `htmlFor`。这个规则的一个例外是 `class` 性质 (可以指定任何 HTML 元素的该性质), 它还可以转换成 `HTMLElement` 的 `className` 属性 (注 4)。

17.1.4 DOM 级别和特性

DOM 标准有两个版本 (或说 “级别”)。1998 年 10 月标准化了 1 级 DOM。它定义了 DOM 的核心接口, 如 `Node`、`Element`、`Attr` 和 `Document`, 还定义了各种 HTML 专有的接口。2000 年 11 月标准化了 2 级 DOM (注 5)。除了一些对核心接口的升级外, DOM 的这个新版本极大地扩展, 可以定义使用文档事件和 CSS 样式表的标准 API, 而且提供了处理文档范围的新工具。在本书编写过程中, W3C 的 DOM 工作组正在标准化 3 级 DOM。有时, 你可能需要查阅 0 级 DOM 标准的参考页。这个术语不代表任何正规标准, 只用于引用正式地引用 W3C 标准化之前的 Netscape 和 Internet Explorer 版本实现的 HTML 文档对象模型的通用特性。

从 2 级 DOM 开始, DOM 标准被模块化了。它的核心模块用 `Document`、`Node`、`Element` 和 `Text` 接口定义了文档的基础树结构, 它是惟一必须的模块。其他的模块都是可选的, 可能被支持, 也可能不被支持, 这由实现的需要决定。一个 Web 浏览器的 DOM 实现显然支持 HTML 模块, 因为 Web 文档是以 HTML 编写的。支持 CSS 样式表的浏览器通常支持 `StyleSheets` 和 CSS 模块, 因为 CSS 样式在动态 HTML 程序设计中扮演关键角色 (我们将在第十八章中看到)。同样, 由于所有有趣的客户端 JavaScript 程序设计方法都要具有事件处理的能力, 所以你会期望 Web 浏览器支持 DOM 标准的 `Events` 模块。遗憾的是, 只有 2 级 DOM 规范近来定义了 `Events` 模块, 在本书编

注 4: `className` 这个名字容易让人误解, 因为除了指定单个类名之外, 该属性 (以及它表示的 HTML 特性) 还指定了用空格分隔的类名列表。

注 5: 标准中的 HTML 专有的部分还没有标准化, 直到 2001 年 11 月, 它仍然处于 “工作草案” 阶段。幸运的是, 当前的工作草案被假定为稳定的, 它只对 1 级标准中的 HTML 专有部分做了很小的修改 (本书将对其进行介绍)。

写的过程中还没有得到广泛支持，下一节我们会看到一个2级DOM的模块完整列表。

17.1.5 DOM 一致性

在本书编写过程中，还没有浏览器与DOM标准完全一致。近来发布的Mozilla与之非常接近，实现完全的2级DOM一致性是Mozilla项目的目标。Netscape 6.1为与最重要的2级模块达到一致做了很好的努力，Netscape 6.0虽然做了足够的工作，但它的覆盖面仍存在差距。Internet Explorer 6最接近1级DOM（但还有一点讨厌的异常），但不支持许多2级模块，特别是Events模块，它是第十九章的主题。Internet Explorer 5和5.5在一致性上存在巨大差距，但它们对关键的1级DOM方法的支持，足够运行本章大部分示例。IE 5的Macintosh版本对DOM的支持比它的Windows版本好得多。

除了Mozilla、Netscape和Internet Explorer外，其他浏览器都在一定程度上对DOM提供了支持。可用的浏览器越来越多，标准支持的领域的改变速度也越来越快，以至于本书不能断言哪个浏览器支持哪些特定的DOM特性。因此，你必须依靠其他的信息资源来确定特定Web浏览器中的DOM实现的一致性。

这种一致性信息的资源之一是实现自身。在一致性实现中，Document对象的implementation属性引用一个DOMImplementation对象，它定义了名为hasFeature()的方法。用这个方法（如果它存在）可以查询一个实现是否支持特定的DOM特性（或模块）。例如，要判断一个Web浏览器中的DOM实现是否支持使用HTML文档的基本1级DOM接口，可以使用如下代码：

```
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("html", "1.0")) {
    // 该浏览器要求支持1级Core和HTML接口
}
```

hasFeature()方法有两个参数，第一个参数是要检查的特性名，第二个参数是用字符串表示的版本号。如果支持指定的版本的指定特性，它将返回true。表17-3列出了1级DOM和2级DOM标准定义的特性名/版本号对。注意，特性名不区分大小写，你可以将任意一个字母大写。该表的第四列说明了支持该特性必需的其他特性以及返回值true的含义。例如，如果hasFeature()方法表明支持

MouseEvents 模块，则暗示也支持 UIEvents 模块，这又暗示了支持 Events、View 和 Core 模块。

表 17-3: 可以用 hasFeature() 方法检测的特性

特性名	版本	描述	暗示
HTML	1.0	1 级 Core 和 HTML 接口	
XML	1.0	1 级 Core 和 XML 接口	
Core	2.0	2 级 Core 接口	
HTML	2.0	2 级 HTML 接口	Core
XML	2.0	2 级 XML 专有接口	Core
Views	2.0	AbstractView 接口	Core
StyleSheets	2.0	通用样式表遍历	Core
CSS	2.0	CSS 样式	Core, Views
CSS2	2.0	CSS2Properties 接口	CSS
Events	2.0	事件处理基础结构	Core
UIEvents	2.0	用户接口事件（加上 Events 和 Views）	Events, Views
MouseEvents	2.0	Mouse 事件	UIEvents
HTMLEvents	2.0	HTML 事件	Events
MutationEvents	2.0	文档变种事件	Events
Range	2.0	文档范围接口	Core
Traversal	2.0	文档遍历接口	Core

在 Internet Explorer 6 (Windows 上的) 中，hasFeature() 方法只为特性 HTML 和版本 1.0 返回 true。它不能报告与表 17-3 中列出的其他特性的一致性（虽然我们将在第十八章看到，它支持最常用的 CSS2 模块）。在 Netscape 6.1 中，hasFeature() 方法为大多数特性名和版本号返回 true，但对 Traversal 特性和 MutationEvents 特性例外。它对版本号为 2 的 Core 特性和 CSS2 特性返回 false，说明了其不完整的支持（即使对这些特性支持得很好）。

本书说明了表 17-3 列出的所有 DOM 模块的接口。其中 Core、HTML、Traversal 和 Range 模块在本章介绍。StyleSheets、CSS 和 CSS2 模块在第十八章中介绍，各种 Events 模块（除了 MutationEvents 模块）将在第十九章中介绍。DOM 参考手册部分包括所有模块的完整说明。

`hasFeature()` 方法并不总是完全可靠。前面提到过, 在 IE 6 中, 即使存在某些一致性问题, 它也会报告 1 级 DOM 与 HTML 特性一致。另一方面, 在 Netscape 6.1 中, 即使非常一致, 它也会报告与 2 级 Core 特性不一致。在这两种情况中, 你需要更多的详细信息来判断到底与哪些特性一致, 与哪些特性不一致。对于一本书来说, 这种类型的信息量太大, 也太易变。

如果你是积极的 Web 开发者, 无疑你已经主动地要求知道或将要发现许多浏览器专有的支持。Web 上也有许多对你有帮助的资源。最重要的是, W3C (与 U.S National Institute of Standards and Technology 合作) 正在致力于开发 DOM 实现的开放资源测试组。在本书编写过程中, 对这个测试套件的研制刚刚开始, 但它应该是 DOM 实现的一致性测试的无价资源。详情请参见 <http://www.w3c.org/DOM/Test/>。

Mozilla 组织有一套用于各种标准 (包括 1 级 DOM, http://www.mozilla.org/quality/browser_sc.html) 的测试套件。Netscape 公司发布了一个测试套件, 包括一些 2 级 DOM 的测试 (<http://developer.netscape.com/evangelism/tools/testsuites/>)。Netscape 公司还发布了早期 Mozilla 版本和 IE 5.5 的 DOM 一致性的比较 (<http://home.netscape.com/browser/future/standards.html/>)。最后, 你还可以在 Web 上的独立站点找到兼容性和一致性的信息。一个著名的站点是 Peter-Paul Koch 发布的。从他的 JavaScript 主页 (<http://www.xs4all.nl/~ppk/js/>) 可以链接到 DOM 兼容性表。

17.1.5.1 Internet Explorer 中的 DOM 一致性

因为 IE 是最常用的 Web 浏览器, 所以这里就它与 DOM 标准的一致性做出一些特殊说明。IE 5 和其后的版本对 1 级 Core 特性和 HTML 特性的支持足够运行本章中的所有例子, 它们对关键的 2 级 CSS 特性的支持足够运行第十八章中的大部分例子。遗憾的是, IE 5、5.5 和 6 都不支持 2 级 DOM 的 Events 模块, 即使 Microsoft 公司参与了该模块的定义, 并有足够的时间在 IE 6 中实现它。我们将在第十九章中看到, 事件处理对客户端的事件处理来说至关重要, IE 缺少对标准事件模型的支持, 极大地阻止了高级客户端 Web 应用程序的开发。

虽然 IE 6 (通过它的 `hasFeature()` 方法) 声称支持 1 级 DOM 标准的 Core 接口和 HTML 接口, 但这一支持并不完善。最惊人的 (也是你最可能遇到的) 问题非常小, 但很讨厌, 即 IE 不支持 Node 接口定义的节点类型常量。回忆一下, 文档中的每个节点都有 `nodeType` 属性, 该属性声明了节点的类型。DOM 标准也声明, Node 接

口定义了常量, 该常量表示每个被定义的类型。例如, 常量 `Node.ELEMENT_NODE` 表示一个 `Element` 节点。在 IE (至少在 IE 6 这样的版本) 中, 这些常量都不存在。

本章的例子已经被修改了, 用整数直接量代替相应的符号常量, 可以避免这一问题。例如, 你将看到如下的代码:

```
if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // 检查 n 是否是 Element 节点
```

在代码中, 用常量代替硬编码的整数直接量是一种好的程序设计风格。如果你想可移植地做这一点, 可以添加下列代码, 在没有这些常量时可以定义它们:

```
if (!window.Node) {  
    var Node = {} // 如果不存在 Node 对象, 则定义一个 Node 对象。  
    ELEMENT_NODE: 1, // 采用下列的属性和值。  
    ATTRIBUTE_NODE: 2, // 注意, 这些只是 HTML 节点类型。  
    TEXT_NODE: 3, // 对于 XML 专有的节点, 需要在此处添加其他常量。  
    COMMENT_NODE: 8,  
    DOCUMENT_NODE: 9,  
    DOCUMENT_FRAGMENT_NODE: 11  
}
```

17.1.6 独立于语言的 DOM 接口

虽然 DOM 标准源于为动态 HTML 程序设计方法制定统一 API 的想法, 但不只是 Web 脚本编写者对 DOM 感兴趣。事实上, 当前服务器端的 Java 和 C++ 程序都大量使用了 DOM 来解析和操作 XML 文档。由于 DOM 被大量使用, 所以它被定义为独立于语言的标准。本书只介绍了 DOM API 的 JavaScript 规约, 不过你应该注意其他几点。首先, 要注意 JavaScript 规约中的对象属性, 它们通常被映射到其他语言规约中的 `get/set` 方法对。因此, 当 Java 程序员询问你 Node 接口的 `getFirstChild()` 方法时, 你要知道, Node API 的 JavaScript 规约没有定义 `getFirstChild()` 方法, 它只定义了 `firstChild` 属性, 在 JavaScript 程序中读这个属性的值相当于在 Java 程序中调用 `getFirstChild()` 方法。

DOM API 的 JavaScript 规约的另一个重要特性是某些 DOM 对象的行为与 JavaScript 数组类似。如果一个接口定义了名为 `item()` 的方法, 那些实现该接口的对象的行为就和只读的数字数组非常相似。例如, 假定通过读一个节点的 `childNodes` 属性获得了一个 `NodeList` 对象, 那么把想要的节点编号传递给 `item()` 方法, 或者更简单

一些,把 NodeList 对象作为数组和下标直接处理,可以得到列表中的一个 Node 对象。下列代码说明了这两种方法:

```
var n = document.documentElement; // 这是一个 Node 对象。
var children = n.childNodes;      // 这是一个 NodeList 对象。
var head = children.item(0);       // 这是使用 NodeList 对象的一种方法。
var body = children[1];            // 但这种方法更容易。
```

同样,如果一个 DOM 对象有 namedItem() 方法,那么可以传递给该方法一个字符串,与把字符串用作对象的数组下标一样。例如,下面几行代码是访问表单元素的等价方法:

```
var f = document.forms.namedItem("myform");
var g = document.forms["myform"];
var h = document.forms.myform;
```

因为使用 DOM 标准有多种方式,所以该标准的体系结构谨慎地定义了 DOM API,不会限制其他人以自己认为合适的方式实现 API。特别的,DOM 标准定义了接口,而不是类。在面向对象的程序设计方法中,类是一种固定的数据类型,必须完全按照规定实现它。而接口是必须一起实现的方法和属性的集合。因此,一种 DOM 实现可以随意定义它认为合适的类,但这些类必须定义各种 DOM 接口的方法和属性。

这种结构有两项意义。首先,实现中使用的类名可以不与 DOM 标准中使用的接口名直接对应。其次,一个类可以实现多个接口。例如,考虑 Document 对象。该对象是 Web 浏览器的实现定义的某个类的实例。我们不知道它是哪种特殊类,只知道它实现了 Document 接口,也就是说,用 Document 对象可以访问 Document 接口定义的所有属性和方法。因为 Web 浏览器使用 HTML 文档,所以我们还知道 Document 对象实现了 HTMLDocument 接口,我们同样可以访问那个接口定义的所有属性和方法。另外,如果 Web 浏览器支持 CSS 样式表,并实现了 DOM CSS 模块,那么 Document 对象还实现了 DocumentStyle 和 DocumentCSS DOM 接口。如果浏览器支持 Events 模块和 Views 模块,那么 Document 对象还实现了 DocumentEvent 和 DocumentView 接口。

因为 DOM 被分割成独立的模块,所以它定义了大量小型附加接口,如 DocumentStyle、DocumentEvent 和 DocumentView,这些接口只定义了一两个方法。这样的接口绝不会独立于核心的 Document 接口实现,所以我没有单独说明它们。当你浏览 DOM 参考手册部分的 Document 接口时,会发现它还列出了 Document 接口的各

种附加接口的属性和方法。同样，如果浏览其中一个附加接口，也会发现对与它相关的核心接口的交叉引用。这个规则的一个例外是，附加接口是一个复杂接口。例如，HTMLDocument接口都是由实现Document对象的对象实现的，但由于它添加了大量新功能，所以本书给它设置了一个单独的参考页。

另一个重要事实是，由于DOM标准定义了接口，而不是类，所以它没有定义任何构造函数方法。例如，如果想创建一个新的Text对象，把它插入文档，不能用如下这样简单的代码：

```
var t = new Text('this is a new text node'); // 没有这样的构造函数。
```

因为DOM没有定义构造函数，所以DOM标准在Document接口中定义了大量有用的工厂方法（factory method）来创建对象。因此，要为文档创建一个新Text节点，可以用下列代码：

```
var t = document.createTextNode('this is a new text node');
```

DOM定义的工厂方法名以单词“create”开头。除了Document定义的工厂方法的外，DOMImplementation也定义了一些代理方法，可以通过document.implementation访问这些工厂方法。

17.2 使用 DOM 的核心 API

既然我们已经研究过文档的树形结构，知道Node对象如何构成树，那么就可以开始详细地研究Node对象和文档树了。前面我提到过，DOM的核心API不是很复杂。接下来的几节有几个例子，说明了如何用它实现普通任务。

17.2.1 遍历文档

我们已经讨论过，DOM把一个HTML文档表示为Node对象的树。对于任何一个树形结构来说，最常做的事情之一就是遍历树，依次检查树的每个节点。例17-1说明了如何遍历树。它是一个JavaScript函数，递归检查一个节点和它的所有子节点，在遍历过程中增加它遇到的HTML标记（即Element节点）数。注意节点的childNodes属性的用法。这个属性的值是一个NodeList对象，该对象的行为（在JavaScript中）

和 Node 对象的数组很类似。因此，该函数通过循环遍历 `childNodes[]` 数组的每个元素，枚举一个给定节点的所有子节点。通过递归，该函数不是枚举一个给定节点的所有子节点，而是枚举树中的所有节点。注意，该函数还示范了用 `nodeType` 属性来判断每个节点的类型。

例 17-1: 遍历文档的节点

```
<head>
<script>
// 该函数的参数是一个 DOM Node 对象，它将检查该节点是否表示一个 HTML 标记。
// 如该节点是否是 Element 对象，它将对该节点的每个子节点递归调用自身，
// 以同样的方式检测它们，它将返回遇到的 Element 对象数。
// 如果调用该函数时传递给它的是 Document 对象，它将遍历整个 DOM 树。
function countTags(n) {
    // n 是一个 Node 对象。
    var numtags = 0;           // 初始化标记计数器。
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/) // 检查 n 是否是 Element 对象。
        numtags++;           // 如果是，则给计数器加 1。
    var children = n.childNodes; // 以下获取 n 的所有子节点。
    for(var i=0; i < children.length; i++) { // 遍历子节点。
        numtags += countTags(children[i]); // 在每个子节点上进行递归操作。
    }
    return numtags;           // 返回标记的总数。
}
</script>
</head>
<!-- 以下是使用 countTags() 函数的例子 -->
<body onload='alert( "This document has " + countTags(document) + " tags. ")'>
This is a <!--sample--> document.
</body>
```

关于例 17-1 需要注意的是，它定义的 `countTags()` 函数是从事件处理程序 `onload` 中调用的，所以在文档被装载完毕前，不会调用它。在使用 DOM 时，这是一条通用的要求，即在装载完文档前，不能遍历或操作文档树。

除了 `childNodes` 属性，Node 接口还定义了其他几个有用的属性。`firstChild` 和 `lastChild` 属性分别引用一个节点的第一个和最后一个子节点，`nextSibling` 和 `previousSibling` 属性引用一个节点相邻的兄弟节点(如果两个节点有共同的父节点，那么它们就是兄弟节点)。这些属性提供了另一种遍历子节点的方法，例 17-2 示范了该方法。这个例子将计算文档 `<body>` 标记中的所有 Text 节点的字符数。注意 `countCharacters()` 函数用 `firstChild` 和 `nextSibling` 属性遍历一个节点的子节点。

例 17-2：另一种遍历文档的方法

```

<head>
<script>
// 该函数的参数是一个 DOM Node 对象，它将检查该节点是否表示一个文本串，
// 例如该节点是否是一个 Text 对象。
// 如果是，它将返回该字符串的长度。
// 如果不是，它将在该节点的每个子节点上递归调用自身，把找到的文本的总长度加起来。
// 注意，它用 firstChild 和 nextSibling 属性枚举该节点的子节点。
// 还要注意，该函数在找到一个 Text 节点后就不再递归，因为 Text 节点没有子节点。
function countCharacters(n) { // n 是一个 Node 对象。
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) // 检查 n 是否是 Text 对象。
        return n.length; // 如果是，则返回它的长度。
    // 否则，n 具有子节点，我们需要计数它们的字符数。
    var numchars = 0; // 用于存放子节点的字符数。
    // 该循环用 firstChild 和 nextSibling 属性检测 n 的子节点，
    // 而不是用 childNodes 来检测。
    for (var m = n.firstChild; m != null; m = m.nextSibling) {
        numchars += countCharacters(m); // 合计找到的字符数。
    }
    return numchars; // 返回总字符数。
}
</script>
</head>
<!--
onload 事件处理程序是使用 countCharacters() 函数的示例。
注意，我们只想计数文档的 <body> 标记中的字符数，
所以我们将 document.body 传递给该函数。
-->
<body onload="alert('Document length:  + countCharacters(document.body))'">
This is a sample document.<p>How long is it?
</body>

```

17.2.2 搜索文档中的特定元素

遍历文档树中所有节点的功能使我们可以找到特定的节点。在用 DOM API 进行程序设计时，需要文档中的一个特殊节点，或文档中具有特定类型的节点列表，这种情况相当常见。DOM API 提供了这样的函数。

在例 17-2 中，我们用 JavaScript 表达式 `document.body` 引用一个 HTML 文档中的 `<body>` 元素。Document 对象的 `body` 属性是一种便利的专用属性，它是引用 HTML 文档的 `<body>` 标记的首选方法。但如果不存在这个便利的属性，可以用下列代码引用 `<body>` 标记：

```
document.getElementsByTagName("body")[0]
```


这个表达式调用 Document 对象的 `getElementsByTagName()` 方法，选择了返回的数组的第一个元素。调用 `getElementsByTagName()` 方法将返回一个数组，该数组元素是文档中的所有 `<body>` 元素。由于 HTML 文档只能有一个 `<body>` 元素，所以我们只对返回数组的第一个元素感兴趣（注 6）。

可以用 `getElementsByTagName()` 方法获取任何类型的 HTML 元素的列表。例如，要找到文档中的所有表，可以使用如下代码：

```
var tables = document.getElementsByTagName('table');
alert("This document contains " + tables.length + " tables");
```

注意，因为 HTML 标记不区分大小写，所以传递给 `getElementsByTagName()` 方法的字符串也不区分大小写。也就是说，上面的代码即使编码为 `<TABLE>`，也可以找到 `<table>` 标记。`getElementsByTagName()` 方法返回元素的顺序就是它们出现在文档中的顺序。如果把特殊字符串 `*` 传递给 `getElementsByTagName()` 方法，它将返回文档中所有元素的列表，元素排列的顺序就是它们在文档中出现的顺序。（IE 5 和 IE 5.5 不支持这种特殊用法。详情可参阅客户端参考手册部分的 IE 专有的 `Document.all[]` 数组。）

有时，你不想要元素列表，而想操作文档中的一个特定元素。如果你对文档结构有充分的了解，就可以使用 `getElementsByTagName()` 方法。例如，如果想处理文档中的第四个段落，可以用下列代码：

```
var myParagraph = document.getElementsByTagName("p")[3];
```

但这不是最好的（也不是最有效的）方法，因为它与文档的结构有非常重要的关系，插入文档开头的一个新段落就会破坏这一代码。相反，在操作文档的一个特定元素时，最好给该元素一个 `id` 性质，为它指定一个（在文档中）惟一的名称，然后就可以用 ID 查找想要的元素。例如，可以用如下代码给文档的第四段编码：

```
<p id='specialParagraph'>
```

接着可以用下列 JavaScript 代码查询那个段落的节点：

注 6：从技术上说，DOM API 规定 `getElementsByTagName()` 返回一个 `NodeList` 对象。在 DOM 的 JavaScript 规约中，`NodeList` 对象的行为与数组相似，通常被用作数组。

```
var myParagraph = document.getElementById("specialParagraph");
```

注意, `getElementById()` 方法不像 `getElementsByTagName()` 那样返回一个数组。因为每个 `id` 性质的值惟一 (或假定为惟一), 所以 `getElementById()` 方法只返回一个元素, 该元素具有匹配的 `id` 属性。`getElementById()` 是一个重要的方法, 在 DOM 程序设计方法中, 它的使用非常常见。

`getElementById()` 方法和 `getElementsByTagName()` 方法都是 `Document` 对象的方法。但 `Element` 对象也定义了 `getElementsByTagName()` 方法, 它的行为与 `Document` 对象的 `getElementsByTagName()` 相似, 只是它返回一个元素, 该元素是调用它的那个元素的后代。该方法不会检索整个文档来查找特定类型的元素, 而是只检索给定的元素。例如, 可以用 `getElementById()` 方法找到特定的元素, 然后用 `getElementsByTagName()` 方法在那些特定标记中找到所有给定类型的后代, 代码如下:

```
// 找到文档中特定的 table 元素, 统计它的列数
var tableOfContents = document.getElementById("TOC");
var rows = tableOfContents.getElementsByTagName("tr");
var numRows = rows.length;
```

最后要注意, 对于 HTML 文档来说, `HTMLDocument` 对象还定义了 `getElementsByName()` 方法。该方法与 `getElementById()` 方法相似, 但它查询的是元素的 `name` 性质, 而不是 `id` 性质。另外, 因为一个文档中的 `name` 性质可能不惟一 (如 HTML 表单中的单选按钮通常具有相同的 `name` 性质), 所以 `getElementsByName()` 方法返回的是元素的数组, 而不是一个元素。例如:

```
// 找到 <a name="top">
var link = document.getElementsByName("top")[0];
// 找到所有的 <input type="radio" name="shippingMethod"> 元素
var choices = document.getElementsByName("shippingMethod");
```

17.2.3 修改一个文档

虽然遍历一个文档的节点很有用, 但 DOM 核心 API 的威力不止于此, 它还能用 JavaScript 动态修改文档的特性。接下来的例子示范了修改文档的基本方法, 并说明了一些可能性。

例 17-3 包括一个名为 `reverse()` 的 JavaScript 函数, 一个示例文档和一个 HTML

按钮。在该按钮被点击时，将调用 `reverse()` 函数，并传递给它表示文档 `<body>` 元素的节点（注意，按钮的事件处理程序调用了 `getElementsByTagName()` 方法来查找 `<body>` 元素）。`reverse()` 函数向后循环遍历给定节点的子节点，并用 `Node` 对象的 `removeChild()` 方法和 `appendChild()` 方法颠倒这些子节点的顺序。

例 17-3: 颠倒文档的节点顺序

```
<head><title>Reverse</title>
<script>
function reverse(n) {           // 颠倒Node n的子节点的顺序
    var kids = n.childNodes;     // 获取子节点的列表
    var numkids = kids.length;   // 统计有多少子节点
    for(var i = numkids-1; i >= 0; i--) { // 反向遍历子节点
        var c = n.removeChild(kids[i]); // 删除一个子节点
        n.appendChild(c);             // 把它放在新位置
    }
}
</script>
</head>
<body>
<p>paragraph #1<p>paragraph #2<p>paragrapn #3  <!-- 一个示例文档 -->
<p>                                           <!-- 调用 reverse() 函数的按钮 -->
<button onclick="reverse(document.body);"
>Click Me to Reverse</button>
</body>
```

例 17-3 的结果如图 17-3 所示，当用户点击按钮时，段落和按钮的顺序将被颠倒过来。

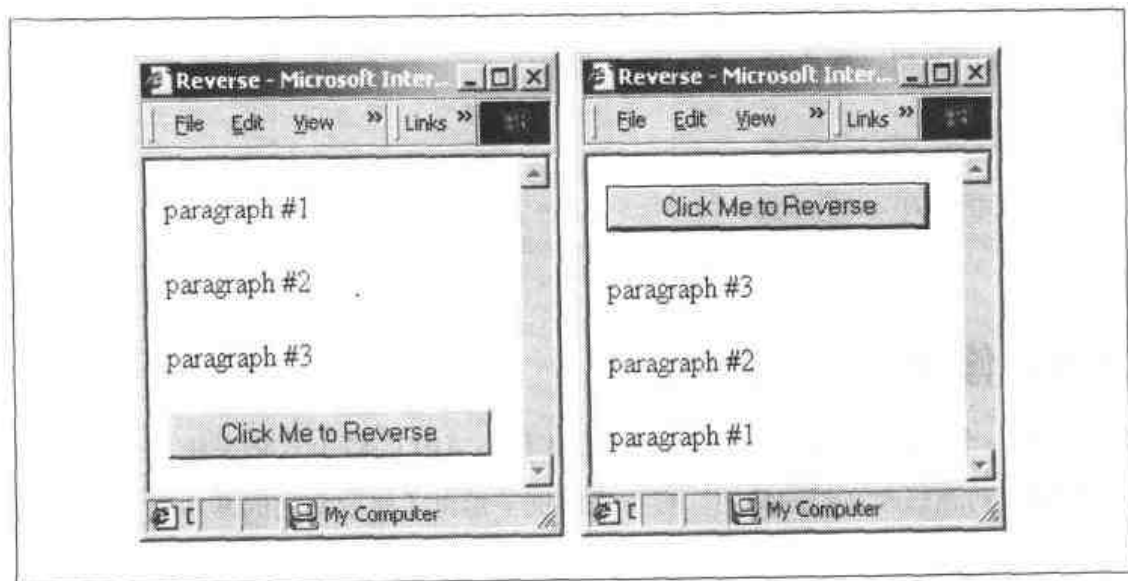


图 17-3: 颠倒前后的文档

关于例 17-3, 有两点需要注意。第一, 如果给 `appendChild()` 传递的节点已经是文档的一部分, 那么首先它将删除该节点, 所以可以省略对 `removeChild()` 方法的调用, 从而简化了代码。第二, 要记住 `childNodes` 属性 (和所有 `NodeList` 对象一样) 是“活”的, 在文档被修改时, 通过 `NodeList` 所做的改变立刻可见。这是 `NodeList` 接口的重要特性, 但实际上这使某些代码编写起来更复杂。例如, 调用 `removeChild()` 方法将改变一个子节点后所有兄弟节点的下标, 所以如果想通过 `NodeList` 对象遍历某个数组, 删除某些节点, 必须慎重编写循环代码。

例 17-4 是前一个例子中的 `reverse()` 函数的演化版本。这个函数使用递归, 不仅颠倒指定节点的子节点, 还颠倒该节点的所有子孙节点。此外, 当它遇到 `Text` 节点时, 它还会颠倒那个节点中的字符顺序。例 17-4 只展示了新 `reverse()` 函数使用的 JavaScript 代码。但在像例 17-3 所示的 HTML 文档中可以很容易地使用它。

例 17-4: 递归颠倒节点的函数

```
// 递归地颠倒Node以下的所有节点并颠倒Text节点
function reverse(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) { // 颠倒Text节点
        var text = n.data; // 获取该内容
        var reversed = '';
        for(var i = text.length-1; i >= 0; i--) // 颠倒它
            reversed += text.charAt(i);
        n.data = reversed; // 保存颠倒后的文本
    }
    else { // 对于非Text节点, 递归地颠倒它的子节点的顺序。
        var kids = n.childNodes;
        var numkids = kids.length;
        for(var i = numkids-1; i >= 0; i--) { // 遍历子节点
            reverse(kids[i]); // 递归地颠倒了节点
            n.appendChild(n.removeChild(kids[i])); // 把子节点移到新位置
        }
    }
}
```

例 17-4 展示了一种改变文档中显示的文本的方法, 即设置相应的 `Text` 节点的 `data` 域。例 17-5 展示了另一种方法。这个例子定义了一个函数 `uppercase()`, 它将递归地遍历给定节点的子节点。在找到一个 `Text` 节点时, 该函数将用新的 `Text` 节点替换它, 这个 `Text` 节点含有与原始节点相同的文本, 只是文本中的字符都被转换成大写了。注意, 该函数使用 `document.createTextNode()` 方法创建新的 `Text` 节点, 用 `Node` 对象的 `replaceChild()` 方法替换原始的 `Text` 节点。还要注意, 调用 `replaceChild()` 方法的不是要替换的节点的父节点, 而是该节点自身。`uppercase()` 函数使用了 `Node` 对象的 `parentNode` 属性来确定要替换的 `Text` 节点的父节点。

除定义了 `uppercase()` 函数外，例 17-5 还包括两个 HTML 段落和一个按钮。在用户点击按钮时，其中一个段落被转换为大写。每个段落由唯一的 `id` 名字标识，用 `<p>` 标记的 `id` 性质指定。按钮的事件处理程序调用 `getElementById()` 方法来获取表示想要的段落的 `Element` 对象。

例 17-5：用等价的大写形式替换节点

```
<script>
// 该函数将递归地查看 Node n 和它的子孙节点。
// 用等价的大写形式替换所有 Text 节点。
function uppercase(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
        // 如果该节点是 Text 节点，则创建一个新的 Text 节点，存放该节点文本的大写版本。
        // 用父节点的 replaceChild() 方法以新的大写节点替换原始节点。
        var newNode = document.createTextNode(n.data.toUpperCase());
        var parent = n.parentNode;
        parent.replaceChild(newNode, n);
    }
    else {
        // 如果该节点不是 Text 节点，遍历它的子节点。
        // 在每个子节点上递归地调用该函数。
        var kids = n.childNodes;
        for(var i = 0; i < kids.length; i++) uppercase(kids[i]);
    }
}
</script>
<!-- 下面是一些示例文本，注意标记 <p> 具有 id 性质 -->
<p id="p1">This <i>is</i> paragraph 1.</p>
<p id="p2">This <i>is</i> paragraph 2.</p>

<!-- 下面是调用上面定义的 uppercase() 函数的按钮 -->
<!-- 注意，调用 document.getElementById() 方法找到想要的节点。 -->
<button onclick="uppercase(document.getElementById('p1'))">Click Me</button>
```

前面两个例子展示了如何修改文档内容，一个方法是替换 `Text` 节点中存放的文本，另一个方法是用全新的 `Text` 节点替换原有的 `Text` 节点。此外，还可以用 `appendData()`、`insertData()`、`deleteData()` 和 `replaceData()` 方法添加、插入、删除或替换一个 `Text` 节点中的文本。虽然这些方法不是直接由 `Text` 接口定义的，但 `Text` 节点从 `CharacterData` 继承了它们。在 DOM 参考手册的“`CharacterData`”条目下可以找到更多有关的信息。

在颠倒节点的例子中，我们看到了如何用 `removeChild()` 方法和 `appendChild()` 方法重排 `Node` 对象的子节点的顺序。但要注意，我们并不限于改变父节点中的节

点顺序, 用 DOM API 可以随意在树中移动文档树中的节点 (但仅限于在同一文档中)。例 17-6 演示了这一点, 它定义了名为 `embolden()` 的函数, 该函数用表示 HTML 标记 `` 的新元素 (由 Document 对象的 `createElement()` 方法创建) 替换指定的节点, 并改变原始节点的父节点, 使它成为新的 `` 节点的子节点。在一个 HTML 文档中, 这会使该节点中的所有文本或它的子孙以粗体显示。

例 17-6: 把一个节点的父节点重定为 `` 元素

```
<script>
// 该函数的参数是 Node n, 用表示 HTML<b> 标记的 Element 节点替换它,
// 并使原始节点成为新的 <b> 的子节点。
function embolden(node) {
    var bold = document.createElement("b"); // 创建新的 <b> 元素。
    var parent = node.parentNode;           // 获取该节点的父节点。
    parent.replaceChild(bold, node);         // 用 <b> 标记替换该节点。
    bold.appendChild(node);                 // 使该节点成为标记 <b> 的子节点。
}
</script>

<!-- 两个示例段 -->
<p id="p1">This <i>is</i> paragraph #1.</p>
<p id="p2">This <i>is</i> paragraph #2.</p>

<!-- 在第一段调用 embolden() 函数的按钮 -->
<button onclick="embolden(document.getElementById('p1'))">Embolden</button>
```

除了用插入、删除、重定父节点和重排节点的方式修改文档外, 还可以通过设置文档元素的性质对文档进行较大的修改。一种方法是调用 `element.setAttribute()`。例如:

```
var headline = document.getElementById("headline"); // 找到已命名的元素
headline.setAttribute("align", "center");           // 设置 align='center'
```

表示 HTML 性质的 DOM 元素定义了对应于每个标准性质的 JavaScript 属性 (即使是被反对使用的性质, 如 `align`), 所以也可以用这段代码实现同样的效果:

```
var headline = document.getElementById("headline");
headline.align = "center"; // 设置对齐方式性质。
```

17.2.4 给文档添加内容

前面两个例子展示了如何把 Text 节点的内容改为全大写的以及如何把一个节点的父节点重定为新创建的 `` 节点的子节点。`embolden()` 函数说明可以创建新节点,

并把它们添加到文档。用 `document.createElement()` 方法和 `document.createTextNode()` 方法创建必要的 `Element` 节点和 `Text` 节点，以及把它们恰当地添加到文档，便可以给文档添加任意内容。例 17-7 演示了这一点。它定义了名为 `debug()` 的函数。该函数提供了把调试信息插入程序的便捷方式，该函数可以有效替代内部函数 `alert()`。图 17-4 显示了使用 `debug()` 函数的例子。

第一次调用 `debug()` 函数，它将用 DOM API 创建一个 `<div>` 元素，并把它插入文档的末尾。第一次调用和此后所有调用传递给 `debug()` 函数的调试消息将插入这个 `<div>` 元素。在 `<p>` 元素中创建 `Text` 节点，并把该 `<p>` 元素插入 `<div>` 元素的末尾，可以显示每条调试消息。

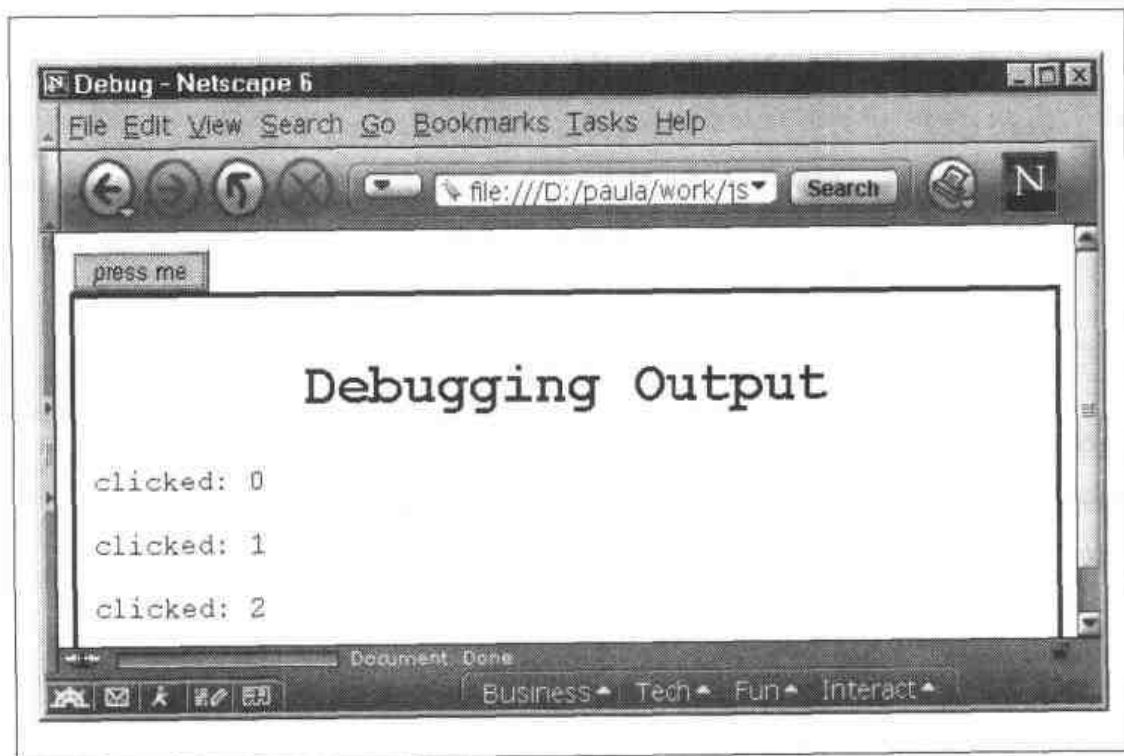


图 17-4: `debug()` 函数的输出

例 17-7 还示范了一个给文档添加新内容的方法，这种方法虽然简便但不标准。含有调试消息的 `<div>` 元素显示了一个大的标题。可以像对其他内容一样给文档创建并添加这个标题，但在这个例子中，我们采用了 `<div>` 元素的 `innerHTML` 属性。把任何元素的 `innerHTML` 属性设置为一个 HTML 文本串，都会使那段 HTML 被解析并插入以作为元素的内容。虽然这个属性不属于 DOM API，但它是 IE 4 及其后的版

本和 Netscape 6 支持的一种有用的快捷方式。尽管它不标准，但它很常用，为了完整性，这个例子采用了它（注 7）。

例 17-7：给文档添加调试输出

```
/**
 * 这个调试函数将在文档底部的特殊对话框中显示纯文本的调试消息。
 * 它是用 alert() 方法显示调试消息的有用替代方法
 **/
function debug(msg) {
    // 如果我们还没有创建显示调试消息的对话框，那么下面就创建它。
    // 注意避免使用其他全局变量，我们将该对话框节点存储为该函数的属性。
    if (!debug.box) {
        // 创建新的 <div> 元素
        debug.box = document.createElement("div");
        // 用 CSS 样式性质指定它的外观
        debug.box.setAttribute('style',
                                'background-color: white; ' +
                                'font-family: monospace; ' +
                                'border: solid black 3px; ' +
                                'padding: 10px;');

        // 把新的 <div> 元素附加到文档末尾。
        document.body.appendChild(debug.box);

        // 下面给 <div> 元素添加一个标题。
        // 注意，innerHTML 属性用于解析 HTML 片段，并把它插入文档。
        // innerHTML 属性不属于 W3C DOM 标准。
        // 但 Netscape 6 和 Internet Explorer 4 及其后的版本支持它。
        // 我们通过明确地创建 <h1> 元素，设置它的样式性质，给它添加一个 Text 节点，
        // 并把它插入文本，来避免使用 innerHTML 属性。
        // 但这不是一种好的快捷方式。
        debug.box.innerHTML = "<h1 style='text-align:center'>Debugging Output</h1>";
    }
    // 当我们进行到此处，debug.box 引用一个 <div> 元素，
    // 我们在该元素中显示调试消息。
    // 首先创建存放该消息的 <p> 节点。
    var p = document.createElement("p");
    // 下面创建存放该消息的文本节点，并把它添加到 <p> 节点。
    p.appendChild(document.createTextNode(msg));
    // 把 <p> 节点附加到存放调试输出的 <div> 节点。
    debug.box.appendChild(p);
}
```

注 7： 当你在文档中插入大型、复杂的 HTML 文本块时，innerHTML 属性极其有用。对于简单的 HTML 段，使用 DOM 方法更为有效，因为不需要 HTML 解析器。注意，用“+=”运算符附加文本到 innerHTML 属性的效率较低。

例 17-7 中列出的 `debug()` 方法可以用于如下所示的 HTML 文档中，它用于生成图 17-4 所示的文档：

```
<script src="Debug.js"></script> <!-- 包括 debug() 函数 -->
<script>var numtimes=0;</script> <!-- 定义一个全局变量 -->
<!-- 下面在一个事件处理程序中使用 debug() 函数 -->
<button onclick="debug('clicked: ' + numtimes++);">press me</button>
```

17.2.5 使用 Document 段

DOM 核心 API 定义了 `DocumentFragment` 对象，作为使用 `Document` 节点组的快捷方法。`DocumentFragment` 是一种特殊类型的节点，它自身不出现在文档中，只作为连续节点集合的临时容器，并允许将这些节点作为一个对象来操作。当把一个 `DocumentFragment` 插入文档时（用 `Node` 对象的 `appendChild()`、`insertBefore()` 或 `replaceChild()`），插入的不是 `DocumentFragment` 自身，而是它的所有子节点。

可以用 `DocumentFragment` 重写例 17-3 的 `reverse()` 方法，代码如下所示：

```
function reverse(r) { // 颠倒 Node n 的子节点的顺序
    var f = document.createDocumentFragment(); // 获取一个空的 DocumentFragment
    while(n.lastChild) // 反向遍历子节点
        f.appendChild(n.lastChild); // 把每个子节点移到 DocumentFragment
    n.appendChild(f); // 然后把它们移回（按照它们的新顺序）
}
```

创建了 `DocumentFragment` 后，就可以用下列代码使用它：

```
document.getElementsByTagName('p')[0].appendChild(fragment);
```

注意，在把 `DocumentFragment` 插入文档时，段的子节点将从段中移到文档中。进行插入操作后，段是空的，除非首次给它添加新的子节点，否则不能再利用它。在本章后面的小节中，当测试 DOM Range API 时，我们会再次见到 `DocumentFragment` 对象。

17.2.6 例子：动态创建内容表

前面几节展示了如何用核心 DOM API 遍历文档、修改文档和给文档添加新内容。在本节的结尾，例 17-8 把这些片段集合在一个较长的例子中。这个例子定义了一个方

法 `maketoc()`，它唯一的参数是一个 `Document` 节点。`maketoc()` 将遍历文档，为它创建一个内容表 (Table Of Content, TOC)，并用新创建的 TOC 替换指定的节点。TOC 是通过查找文档中的 `<h1>`、`<h2>`、`<h3>`、`<h4>`、`<h5>` 和 `<h6>` 标记而生成的，并假定这些标记标志了文档中重要段的开头。除了创建 TOC 外，`maketoc()` 函数还在每个段标题前插入了命名的锚元素 (`<a>` 元素，设置了 `name` 性质，而不是 `href` 性质)，以便 TOC 能够直接链接到每个段。最后，`maketoc()` 函数还在每个段的开头插入了返回 TOC 的链接，在读者到达一个新段时，既可以阅读下面的段，也可以用链接返回 TOC，再选择一个新段。图 17-5 展示了 `maketoc()` 函数生成的 TOC。

如果你要维护和修改的长文档用 `<h1>`、`<h2>` 和相关的标记进行了分段，那么 `maketoc()` 函数会很有用。在长文档中，TOC 非常有用，但如果经常修改文档，那么很难使 TOC 与文档自身保持一致。本书的 TOC 是通过对本书的内容进行后加工之后自动创建的。用 `maketoc()` 可以对你的 Web 文档这样做。你可以在一个 HTML 文档中用如下方式使用该函数：

```
<script src="TOC.js"></script> <!-- 装载maketoc()函数 -->
<!-- 当文档完全装载后调用maketoc()函数 -->
<body onload="maketoc(document.getElementById('placeholder'))">
<!-- 该span元素将被生成的TOC替换 -->
<span id="placeholder">Table Of Contents</span>
// ... 此处是余下的文档 ...
```

另一种使用 `maketoc()` 函数的方法是只在读者请求时才生成 TOC。添加一个链接 (或按钮)，在用户点击它时，用生成的 TOC 替换它自身即可做到这一点：

```
<a href="#" onclick="maketoc(this); return false;">Show Table Of Contents</a>
```

`maketoc()` 函数的代码如下。例 17-8 比较长，但它有很详细的注释，而且采用的技术也已经介绍过了。它值得作为 DOM API 实际应用的例子来研究。注意，`maketoc()` 函数依靠两个帮助函数。为了保持模块性，这两个帮助函数在 `maketoc()` 函数中定义。这样可以防止在全局名字空间中出现不必要的额外函数。

例 17-8：自动生成内容表

```
/**
 * 创建该文档的内容表。通过替换参数指定的节点，把 TOC 插入文档。
 */
function maketoc(replace) {
    // 创建一个 <div> 元素，作为 TOC 树的根元素。
```

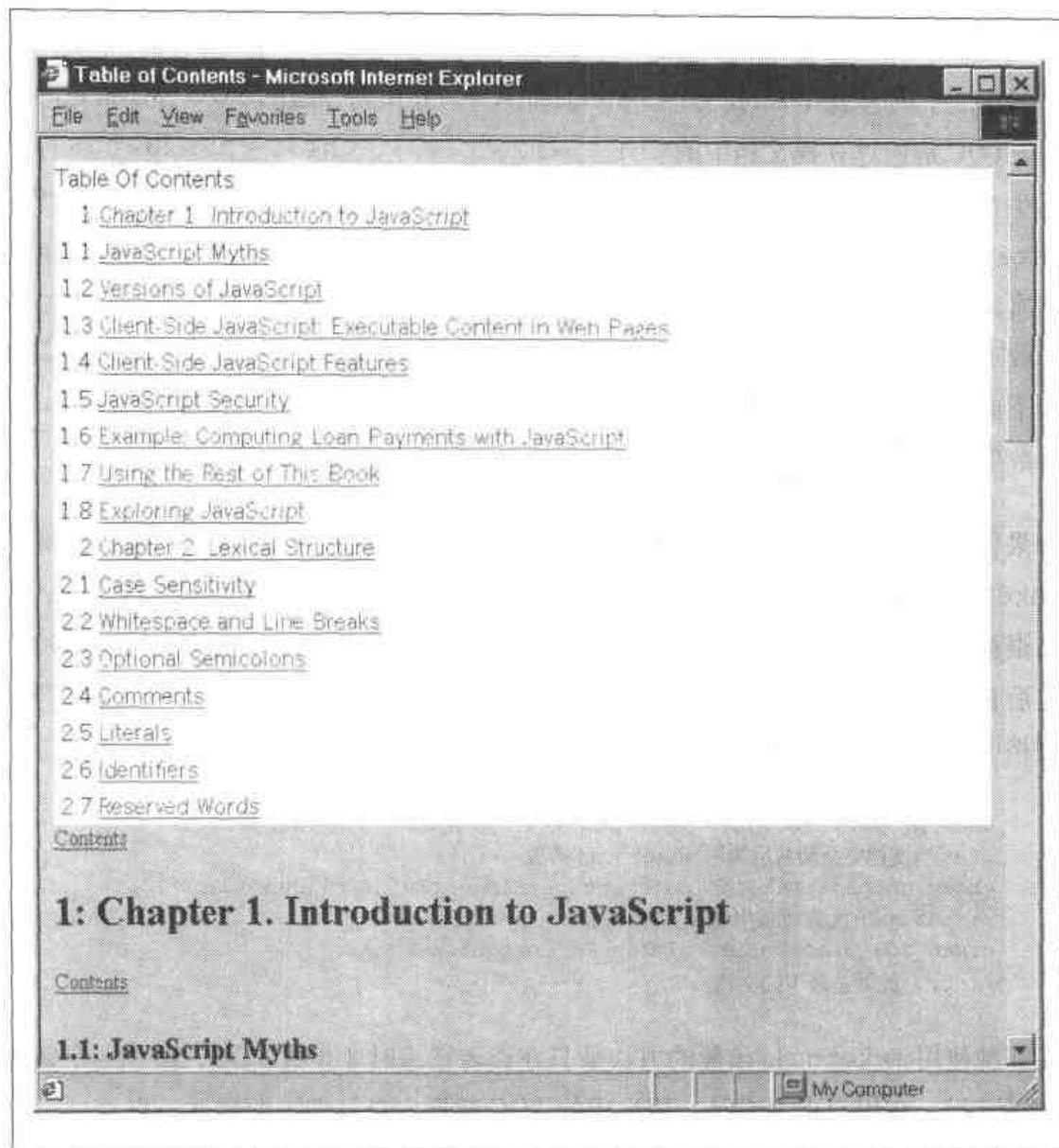


图 17-5: 动态创建的内容表

```
var toc = document.createElement('div');

// 设置 TOC 的背景颜色和字体。
// 下一章我们将学习样式属性。
toc.style.backgroundColor = "white";
toc.style.fontFamily = "sans-serif";

// 用锚元素作为 TOC 的开端，以便我们能链接回来。
var anchor = document.createElement('a'); // 创建一个 <a> 节点，
anchor.setAttribute('name', "TOC");       // 给它命名，
toc.appendChild(anchor);                  // 把它插入 TOC。

// 使锚元素的主体成为 TOC 的标题。
```

```

anchor.appendChild(document.createTextNode('Table Of Contents'));

// 创建一个<table>元素，用来存放TOC，并加入它。
var table = document.createElement("table");
toc.appendChild(table);

// 创建一个<tbody>元素，用来存放TOC的行。
var tbody = document.createElement('tbody');
table.appendChild(tbody);

// 初始化一个数组，以跟踪段号。
var sectionNumbers = [0,0,0,0,0,0];

// 递归遍历文档的主体，查找段。
// 段标记为<h1>、<h2>...
// 通过给表添加行，用这些标记创建TOC。
addSections(document.body, tbody, sectionNumbers);

// 最后，通过用带有TOC子树的替换参数替换指定的节点，把TOC插入文档。
replace.parentNode.replaceChild(toc, replace);

// 该方法将递归地遍历根节点为n的树。
// 查找标记<h1>到<h6>，通过给TOC参数指定的HTML表添加行。
// 便可以用这些标记的内容创建表的内容。它使用sectionNumbers数组跟踪当前的段号。
// 该函数在maketoc()内定义，使它在maketoc()外部不可见。
// maketoc()是由该JavaScript模块输出的唯一函数。
function addSections(n, toc, sectionNumbers) {
    // 遍历n的所有子节点
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
        // 检查m是否是标题元素。
        // 如果我们只使用(m instanceof HTMLHeadingElement)就可以了，
        // 但标准禁止这样做，而且IE不支持它。
        // 因此，我们必须检查标记名，看它是否是H1-H6。
        if ((m.nodeType == 1) && /* Node.ELEMENT_NODE */
            (m.tagName.length == 7) && (m.tagName.charAt(0) == "H")) {
            // 检查它是什么级别的标题。
            var level = parseInt(m.tagName.charAt(1));
            if (!isNaN(level) && (level >= 1) && (level <= 6)) {
                // 为该级标题增加段号。
                sectionNumbers[level-1]++;
                // 把所有低级标题的段号重置为0。
                for(var i = level; i < 6; i++) sectionNumbers[i] = 0;
                // 下面把所有级别的标题段号组合起来。
                // 生成如"2.3.1"的段号。
                var sectionNumber = '';
                for(var i = 0; i < level; i++) {
                    sectionNumber += sectionNumbers[i];
                    if (i < level-1) sectionNumber += ".";
                }

                // 创建一个锚元素，标记该段的开端。
                // 它将成为我们添加到TOC的链接的目标。
            }
        }
    }
}

```

```

var anchor = document.createElement("a");
anchor.setAttribute('name', 'SECT'+sectionNumber);

// 创建返回 TOC 的链接, 使它成为锚元素的子节点。
var backlink = document.createElement("a");
backlink.setAttribute('href', "#TOC");
backlink.appendChild(document.createTextNode("Contents"));
anchor.appendChild(backlink);

// 把锚元素插入文档中的段标题之前。
n.insertBefore(anchor, m);

// 下面创建到该段的链接。以下代码会把它加入 TOC。
var link = document.createElement("a");
link.setAttribute('href', "#SECT" + sectionNumber);
// 用下面定义的函数获取标题文本。
var sectionTitle = getTextContent(m);
// 用标题文本作为链接的内容。
link.appendChild(document.createTextNode(sectionTitle));

// 为 TOC 创建新行。
var row = document.createElement('tr');
// 为该行创建两列。
var col1 = document.createElement("td");
var col2 = document.createElement("td");
// 使第一列右对齐, 把段号放入其中。
col1.setAttribute("align", "right");
col1.appendChild(document.createTextNode(sectionNumber));
// 把一个链接放入第二列中。
col2.appendChild(link);
// 把列添加到行, 把行添加到表。
row.appendChild(col1);
row.appendChild(col2);
toc.appendChild(row);

// 修改段标题自身元素, 添加段号作为段标题的一部分
m.insertBefore(document.createTextNode(sectionNumber+" "),
m.firstChild);
}
}
else { // 否则, 这不是标题元素, 所以进行递归。
    addSections(m, toc, sectionNumbers);
}
}

// 这个工具函数将遍历节点 n, 返回找到的所有 Text 节点的内容, 舍弃 HTML 标记。
// 它也被定义为一个嵌套函数, 所以它是该模块专用的。
function getTextContent(n) {
    var s = '';
    var children = n.childNodes;
    for(var i = 0; i < children.length; i++) {

```

```
        var child = chs[dren[i]];
        if (child.nodeType == 3 /*Node.TEXT_NODE*/) s += child.data;
        else s += getTextContent(child);
    }
    return s;
}
}
```

17.2.7 使用 XML 文档

虽然 Web 浏览器显示的是 HTML 文档, 但 XML 文档却成为越来越重要的数据资源。因为 DOM 使我们可以遍历和操作 HTML 文档和 XML 文档, 所以可以用 DOM 方法装载 XML 文档, 从中提取信息, 动态地创建这些信息的 HTML 版本, 以便在 Web 浏览器中显示这些信息。例 17-9 展示了如何在 Netscape 6.1 和 Internet Explorer 6 中实现这一目标。例 17-9 是一个 HTML 文件, 几乎全部由 JavaScript 代码构成。使用 URL 查询字符串指定要装载的数据文件的相对 URL, 便可以通过 URL 装载该文件。例如, 可以用如下的 URL 调用该示例文件:

```
file:///C:/javascript/DisplayEmployeeData.html?data.xml
```

这个示例文件名为 *DisplayEmployeeData.html*, 它使用的 XML 文件名为 *data.xml*。这个 XML 文件必须含有如下格式的数据:

```
<employees>
  <employee name="J. Doe"><job>Programmer</job><salary>32768</salary></employee>
  <employee name="A. Baker"><job>Sales</job><salary>70000</salary></employee>
  <employee name="Big Cheese"><job>CEO</job><salary>1000000</salary></employee>
</employees>
```

这个例子含有两个 JavaScript 函数。第一个函数是 `loadXML()`, 它是装载 XML 文件的通用函数。它既含有装载 XML 文档的 2 级 DOM 代码, 又有实现同样操作的 Microsoft 专有 API 代码。在这个例子中, 惟一全新的方法是用 `DOMImplementation.createDocument()` 方法创建新 `Document` 对象, 然后调用那个 `Document` 对象的 `load()` 方法。需要注意的重要一点是, 文档不是瞬间装载的, 所以对 `loadXML()` 的调用将在装载文档之前返回。因此, 我们传递给 `loadXML()` 一个引用, 它引用另一个函数, 该函数在文档装载完毕时调用。

这个例子中的另一个函数是 `makeTable()`。它是传递给 `loadXML()` 的函数。在 XML 文件装载完毕时, 将给 `makeTable()` 函数传递表示 XML 文件的 `Document` 对象和

该文件的 URI。makeTable() 将使用前面见过的 DOM 方法从 XML 文档中提取信息，并把它插入 HTML 文档的一个表中，由浏览器显示这个 HTML 文档。这个函数还说明了一些与表相关的方法的使用，它们由 HTMLTableElement、HTMLTableRowElement 和相关的接口定义。DOM 参考手册部分详细介绍了与表相关的接口和它们的方法。尽管这个函数中使用的 DOM 方法和属性都很简单，但它们的组合比较难懂。仔细研究这些代码，你应该能够理解它们。

例 17-9: 从一个 XML 文档装载并读取数据

```
<head><title>Employee Data</title>
<script>
// 该函数将从指定的 URI 装载 XML 文档，在装载完毕时，
// 把该文档和 URL 传递给指定的处理函数。
// 任何 XML 文档都可以使用该函数。
function loadXML(url, handler) {
    // 如果它受支持，则采用标准的 2 级 DOM 技术。
    if (document.implementation && document.implementation.createDocument) {
        // 创建新的 Document 对象。
        var xmldoc = document.implementation.createDocument("", "", null);
        // 设置装载完毕时发生的情况。
        xmldoc.onload = function() { handler(xmldoc, url); };
        // 告诉它装载什么 URL。
        xmldoc.load(url);
    }
    // 否则，用 Microsoft 公司专有的用于 Internet Explorer 的 API。
    else if (window.ActiveXObject) {
        var xmldoc = new ActiveXObject("Microsoft.XMLDOM"); // 创建文档。
        xmldoc.onreadystatechange = function() { // 设置 onload。
            if (xmldoc.readyState == 4) handler(xmldoc, url);
        }
        xmldoc.load(url); // 开始装载。
    }
}

// 该函数将用从传递给它的 XML 文档读取的数据建立一个雇员 HTML 表。
function makeTable(xmldoc, url) {
    // 创建一个 <table> 对象，并把它插入文档。
    var table = document.createElement("table");
    table.setAttribute("border", "1");
    document.body.appendChild(table);

    // 用 HTMLTableElement 和相关接口的方法定义表的标题和表头。
    // 给每个列命名。
    var caption = "Employee Data from " + url;
    table.createCaption().appendChild(document.createTextNode(caption));
    var header = table.createTHead();
    var headerrow = header.insertRow(0);
    headerrow.insertCell(0).appendChild(document.createTextNode("Name"));
    headerrow.insertCell(1).appendChild(document.createTextNode("Job"));
}
```

```
headerrow.insertCell(2).appendChild(document.createTextNode("Salary"));

// 下面找到 xmlDoc 文档中的所有 <employee> 元素。
var employees = xmlDoc.getElementsByTagName("employee");

// 遍历这些 <employee> 元素。
for(var i = 0; i < employees.length; i++) {
    // 对于每个雇员，用标准 DOM 方法获取名字、工作和薪水数据。
    // 名字来自一个 HTML 性质。其他值来自 <job> 和 <salary> 标记中的 Text 节点。
    var e = employees[i];
    var name = e.getAttribute("name");
    var job = e.getElementsByTagName("job")[0].firstChild.data;
    var salary = e.getElementsByTagName("salary")[0].firstChild.data;

    // 既然我们已经有了雇员的数据，便可以用表的方法创建一个新行，
    // 用行的方法创建一个新表元，用 Text 节点存放这些数据。
    var row = table.insertRow(i+1);
    row.insertCell(0).appendChild(document.createTextNode(name));
    row.insertCell(1).appendChild(document.createTextNode(job));
    row.insertCell(2).appendChild(document.createTextNode(salary));
}
}
</script>
</head>
<!--
文档的主体不包含静态文本，所有元素都是动态地由 makeTable() 函数生成。
Onload 事件处理程序以调用 loadXML() 方法装载 XML 数据文件开始。
注意使用 location.search 方法对查询串中的 XML 文件名编码。
装载具有 DisplayEmployeeData.html?data.xml 这样的 URL 的 HTML 文件。
-->
<body onload="loadXML(location.search.substring(1), makeTable)">
</body>
```

17.3 DOM 与 Internet Explorer 4 的兼容性

尽管 IE 4 不遵循 DOM，但它有一些与 DOM 核心 API 相似的特性。这些特性不属于 DOM 标准，并且与 Netscape 不兼容，但是与 IE 以后的版本兼容。下面总结了这些特性，详情请参见客户端参考手册部分。

17.3.1 遍历文档

DOM 标准规定，所有 Node 对象（包括 Document 对象和所有 Element 对象）都有 childNodes[] 数组，该数组包含那个节点的子节点。IE 4 不支持 childNodes[] 数组，但它在 Document 对象和 HTML 元素对象中提供了非常相似的

children[]数组，因此，要编写例 17-1 所示的遍历 IE 4 文档中的所有元素的递归函数是很容易的。

但 IE 4 的 children[]数组和标准 DOM 数组 childNodes[]之间有一点很重要的差别。IE 4 没有 Text 节点类型，不把文本串看作子节点。因此，在 IE 4 中，没有任何标记，只含纯文本的 <p> 标记只有空的 children[]数组。不久我们会看到，通过 IE 4 的 innerText 属性可以访问 <p> 标记的文本内容。

17.3.2 搜索文档元素

IE 4 不支持 Document 对象的 getElementById() 方法和 getElementsByTagName() 方法。而 Document 对象和所有文档元素都有数组属性 all[]。顾名思义，这个数组表示文档中的所有元素或元素中包含的所有元素。注意，all[] 不只表示文档或元素的子节点，它表示所有子孙，无论它们嵌套得多深。

all[] 数组有几种使用方法。如果用整数 n 做下标，它将返回文档或父元素的第 n+1 个元素。例如：

```
var e1 = document.all[0];    // 文档的第一个元素
var e2 = e1.all[4];          // 元素 1 的第五个元素
```

元素是按照在文档中出现的顺序进行编码的。注意，IE 4 API 和 DOM 标准之间的一个重大差别是，IE 没有 Text 节点的概念，所以 all[] 数组只存放文档元素，不存放出现在各元素中的文本。

用名字引用文档元素通常比用编号引用元素更有效。IE 4 等价于 getElementById() 的方法是用字符串做 all[] 的下标，而不用数字。这时，IE 4 将返回 id 性质或 name 性质具有指定值的元素。如果这样的元素不止一个（这种情况有可能发生，在有多 个表单元素时很常见，如具有相同 name 性质的单选钮），结果是这些元素的数组。例如：

```
var specialParagraph = document.all["special"];
var radioBoxes = form.all["shippingMethod"]; // 返回一个数组
```

JavaScript 还允许把数组下标表示为属性名：

```
var specialParagraph = document.all.special;
```

```
var radioboxes = form.all.snippingMethod;
```

以这种方式使用 `all[]` 数组可以提供与 `getElementById()` 和 `getElementsByName()` 一样的功能。主要差别是 `all[]` 数组将这两种方法的功能合并起来，如果你无意中使不相关的元素的 `id` 性质和 `name` 性质使用了相同的值，就会引发问题。

`all[]` 数组有一点异常之处，即用 `tags()` 方法可以以标记名获取一个元素数组。例如：

```
var lists = document.all.tags("ul"); // 找到文档中的所有 <ul> 标记  
var items = lists[0].all.tags("li"); // 找到第一个 <ul> 中的所有 <li> 标记
```

IE 4 的语法提供了与 DOM Document 对象和 Element 对象的 `getElementsByTagName()` 方法基本相同的功能。注意，在 IE 4 中，应该用全大写字母指定标记名。

17.3.3 修改文档

与 DOM 标准一样，IE 4 用相应的 `HTMLElement` 对象的属性表示 HTML 标记的性质。因此，可以通过动态改变 HTML 性质来修改 IE 4 中显示的文档。如果修改性质改变了元素的大小，文档将“回流”以容纳它的新尺寸。IE 4 的 `HTMLElement` 对象也定义了 `setAttribute()`、`getAttribute()` 和 `removeAttribute()` 方法。这些方法与标准 DOM API 中的 `Element` 对象定义的同名方法相似。

DOM 标准定义了 API，它可以创建新节点，给文档树插入新节点，重定节点的父节点，在树中移动节点。IE 4 不能进行这些操作。但 IE 4 中的所有 `HTMLElement` 对象都定义了 `innerHTML` 属性。把这个属性设置为一个 HTML 文本串可以使你用自己需要的内容替换一个元素的内容。由于 `innerHTML` 属性作用非常强大，所以即使它不属于 DOM 标准，Netscape 6（和它继承的 Mozilla）也实现了它。例 17-7 示范了 `innerHTML` 属性的用法。

IE 4 还定义了几个相关的属性和方法。`outerHTML` 属性将用指定的 HTML 文本串替换一个元素的内容和元素本身。`innerText` 属性和 `outerText` 属性与 `innerHTML` 属性和 `outerHTML` 属性相似，只是后者将字符串作为纯文本处理，而不作为 HTML 解析。最后，`insertAdjacentHTML()` 方法和 `insertAdjacentText()` 方法不管元素的内容，而在它附近（在它之前或之后，在内部或外部）插入新的 HTML 或纯文

本内容。这些属性和函数不像 innerHTML 那么常用，而且 Netscape 6 没有实现它们。要进一步了解有关内容，请参阅客户端参考手册部分的“HTML Element”条目。

17.4 DOM 与 Netscape 4 的兼容性

Netscape 4 远没有实现 DOM 的标准。尤其是 Netscape 4 没有提供访问或设置文档中的任意元素的性质。当然，Netscape 4 支持 0 级 DOM API，所以像表单或链接那样的元素还可以通过 forms[] 数组和 links[] 数组访问，只是没有通用的方式来遍历元素的子节点或设置它们的性质。另外，Netscape 4 不能“回流”文档的内容以响应元素大小的变化。

尽管有这些限制，Netscape 4 还是提供了访问和操作关键“动态元素”的 API，这些元素可以用于实现 DHTML 效果。在 Netscape 4 API 中，这些元素称为层(layer)，它们浮于文档其余部分之上，可以独立于其他文档元素移动、调整大小和修改。层通常由 CSS 样式表实现，第十八章将详细讨论 Netscape 4 的 Layer API。

以下内容只是一个概览，解释了如何创建、访问和修改文档中某个层元素的内容。虽然 Netscape 4 不支持 DOM 这样的标准，但用 Layer API 可以实现标准 API 能实现的动态效果。注意，尽管 Layer API 被提交到 W3C，以考虑作为 DOM 标准的一部分，但这部分 API 还没有被标准化。因为 Netscape 6 基于完全重写了的 Netscape 4，所以 Layer API 已经被废弃了，Netscape 6（或 Mozilla）不支持它。

在文档中用 <layer> 标记可以创建层，它是 Netscape 专有的 HTML 扩展。但在 Netscape 4 文档中，最常用来创建层的方法是用标准的 CSS 定位性质（详见第十八章）。Netscape 4 会把用 CSS 样式性质变为动态的元素作为层处理，并用 Layer API 来操作（注意，Netscape 4 不允许元素是动态的。为了安全起见，通常用 <div> 包装元素包装可能成为动态的元素）。JavaScript 也可以用 Layer() 构造函数创建层，在本书的客户参考手册部分可以找到有关说明。

如果你已经在自己的文档中创建了动态元素或层，那么 Netscape 允许使用简单的 0 级 DOM API 扩展来访问它们。就像通过 forms[] 数组访问表单元素，通过 images[] 数组访问图像元素一样，可以通过 Document 对象的 layers[] 数组访问层。如果文档中的第一个层有 name 性质“layer1”，则可以用下列表达式引用该层元素：

```
document.layers[0]           // 用数字作为数组下标
document.layers['layer1']     // 用元素名作为数组下标
document.layer1               // 指定的层成了文档的属性
```

如果层没有 name 性质，而有 id 性质，则这个性质的值将被用作层名。

文档中的层由 Layer 对象表示，该对象定义了大量有用的属性和方法，可以用这些属性和方法来移动层、调整层大小、显示层、隐藏层和设置层的堆叠顺序。这些属性和方法与 CSS 样式性质相关，第十八章将讨论它们。Layer 对象最有趣的一点是它含有自己的 Document 对象，层的内容被看作一个文档，它完全独立于包含该层的文档。这样，通过调用 document.write() 方法和 document.close() 方法动态地重写层内容就可以修改层显示的内容。可以用 Layer 对象的 load() 方法动态地把文档装载到层。最后要注意，层本身还可能包含层，可以用下列表达式引用这样的嵌套图层：

```
// 在该层中嵌套的第二个层名为 'mylayer'
document.mylayer.document.layers[1];
```

17.5 简便方法：Traversal 和 Range API

迄今为止，我们在本章中讨论了 DOM 的核心 API，它提供了遍历和操作文档的基本方法。DOM 标准还定义了其他可选的 API 模块，其中最重要的模块将在下一章中讨论。其中两个可选的模块是在核心 API 上构建的方便的 API。Traversal API 定义了遍历文档和筛选出用户不感兴趣节点的高级方法。Range API 定义了操作连续范围内的文档内容的方法，即使那些内容不在节点的边界处开始或结束。后面的几节简要地介绍了 Traversal 和 Range API。完整的说明请参阅 DOM 参考手册部分。Range API 由 Netscape 6.1 实现（Netscape 6 部分地实现了 Range API），Traversal API 有望被 Mozilla 1.0 完全支持，这意味着 Netscape 及以后的版本将会支持它。在本书编写的过程中，IE 还不支持任何一个 API。

17.5.1 DOM Traversal API

在本章的开头，我们见过用递归依次检查每个节点的方法来遍历文档树。这是一种重要的方法，但我们通常不需要检测文档的每个节点。我们只想检测文档中的 元素或遍历 <table> 元素的子树。对这种有选择的文档遍历，Traversal API

提供了高级的方法。前面提到过, Traversal API 是可选的, 在本书编写的过程中, 主流的第六代浏览器还没有实现它。采用下列代码可以检测一个与 DOM 兼容的浏览器是否支持 Traversal API:

```
document.implementation.hasFeature('Traversal', 2.0); // 如果被支持, 则返回 true
```

17.5.1.1 NodeIterator 和 TreeWalker

Traversal API 由两个关键对象构成, 每个对象提供了不同的文档筛选视图。NodeIterator 对象提供了文档节点被“展平”的顺序视图, 并支持筛选。你可以定义一个 NodeIterator 对象, 该对象用来筛选除 标记之外的所有文档元素, 并用列表表示这些元素。NodeIterator 对象的 `nextNode()` 方法和 `previousNode()` 方法使你可以在列表中前后移动。注意, NodeIterator 不需要递归就可以遍历文档中被选中的部分。只需要在循环中使用一个 NodeIterator 对象, 反复调用 `nextNode()` 方法, 直到找到你需要的节点, 或直到它返回 `null`, 就说明它已经到达了文档末尾。

Traversal API 中的另一个关键对象是 TreeWalker。该对象也提供了文档被筛选后的视图, 并允许调用 `nextNode()` 方法和 `previousNode()` 方法遍历筛选过的文档, 但它不能展平文档树。TreeWalker 对象保留了文档的树形结构 (尽管它的树形结构可能由于节点筛选改变了很多), 允许你用 `firstChild()`、`lastChild()`、`nextSibling()`、`previousSibling()` 和 `parentNode()` 方法在树中导航。当你想遍历被筛选过的树, 而不是调用 `nextNode()` 方法来遍历它时, 或者当你想对某些子树执行更复杂的遍历操作 (如跳过某些子树) 时, 可以使用 TreeWalker 对象代替 NodeIterator 对象。

Document 对象为创建 NodeIterator 对象和 TreeWalker 对象定义了 `createNodeIterator()` 方法和 `createTreeWalker()` 方法。可以通过测试这两种方法是否存在, 来检测一个浏览器是否支持 TreeWalker API 的方法:

```
if (document.createNodeIterator && document.createTreeWalker) {  
    /* 可以安全使用 Traversal API */  
}
```

`createNodeIterator()` 方法和 `createTreeWalker()` 方法的四个参数相同, 只是返回的对象类型不同。第一个参数是开始遍历的节点。如果你想遍历整个文档, 该参数应该是一个 Document 对象。如果你只想遍历文档的一个子树, 则该参数是

其他节点。第二个参数是一个数字，说明了 `NodeIterator` 或 `TreeWalker` 应该返回的节点类型。该参数由 `NodeFilter` 对象（下一节讨论）定义的一个或多个 `SHOW_` 常量的和构成。第三个参数是一个可选的函数，用于指定更复杂的筛选，而不只是根据节点类型包括或丢弃它们（同样参阅下一节）。最后一个参数是一个布尔值，指定了在遍历过程中是否扩展文档中的实体引用节点。在使用 XML 文档时，这个参数非常有用，但使用 HTML 文档的 Web 程序员可以忽略它，或使用 `false`。

17.5.1.2 筛选

`NodeIterator` 对象和 `TreeWalker` 对象最重要的特性是它们的选择性，它们能够筛选出你不需要的节点。前面提到过，可以用 `createNodeIterator()` 方法和 `createTreeWalker()` 方法的第二个和第三个（可选的）参数指定你需要的节点。这两个参数可以指定两级筛选。第一级只根据节点类型来接受或丢弃节点。`NodeFilter` 对象为每种节点定义了数字常量，通过把你需要的节点类型的常量加起来（或用按位 OR 运算符“|”）可以指定节点类型。

例如，如果你只需要使用文档中的 `Element` 节点和 `Text` 节点，那么可以把下列表达式用作第二个参数：

```
NodeFilter.SHOW_ELEMENT | NodeFilter.SHOW_TEXT
```

如果只需要使用 `Element` 节点，则使用下面的代码：

```
NodeFilter.SHOW_ELEMENT
```

如果需要使用所有节点，或不想只根据类型丢弃任何节点，则可以用特殊的常量：

```
NodeFilter.SHOW_ALL
```

如果对注释之外的所有类型的节点都感兴趣，则使用下面的代码：

```
~NodeFilter.SHOW_COMMENT
```

（如果你忘了“~”运算符的用法，请参阅第五章。）注意，一级筛选只适用于独立的节点，不适用于它们的子节点。如果第二个参数是 `NodeFilter.SHOW_TEXT`，那么虽然 `NodeIterator` 对象或 `TreeWalker` 对象不返回元素节点，但也不完全丢弃它们，它仍然会遍历 `Element` 节点的子树，以找到你感兴趣的 `Text` 节点。

传递这种基于类型的筛选操作的节点，可以经受二级筛选。第二次筛选由你定义的函数实现，因此可以执行任何复杂的筛选。如果你不需要这种筛选，那么可以把 `createNodeIterator()` 方法和 `createTreeWalker()` 方法的第三个参数的值指定为 `null`。但如果你想执行这种筛选，就必须传递一个函数作为第三个参数。

该函数的参数是一个节点，它应该计算该节点，并返回一个值，该值说明是否应该筛选出该节点。有三种可能的返回值，被定义为一个 `NodeFilter` 常量。如果筛选函数返回 `NodeFilter.FILTER_ACCEPT`，`NodeIterator` 对象或 `TreeWalker` 对象将返回该节点。如果函数返回 `NodeFilter.FILTER_SKIP`，该节点将被筛选出来，`NodeIterator` 对象或 `TreeWalker` 对象也不会返回它。但该节点的子节点仍然会被遍历。如果使用 `TreeWalker` 对象，那么筛选函数可能返回 `NodeFilter.FILTER_REJECT` 的值，这说明不应该返回节点，甚至不应该再遍历它。

例 17-10 示范了 `NodeIterator` 对象的创建和用法，这样使前面的讨论更清楚。但要注意，在本书编写的过程中，没有一个主流浏览器支持 `Traversal API`，所以这个例子没有经过测试。

例 17-10: 创建并使用 `NodeIterator` 对象

```
// 定义 NodeFilter 函数，只接受 <img> 元素
function imgfilter(n) {
    if (n.tagName == 'IMG') return NodeFilter.FILTER_ACCEPT;
    else return NodeFilter.FILTER_SKIP;
}

// 创建一个 NodeIterator，查找 <img> 标记
var images = document.createNodeIterator(document, // 遍历整个文档
    /* 只查看 Element 节点 */ NodeFilter.SHOW_ELEMENT,
    /* 过滤掉 <img> 之外的所有元素 */ imgfilter,
    /* 在 HTML 文档中不用 */ false);

// 用迭代器遍历所有图像，并用它们做一些操作
var image;
while((image = images.nextNode()) != null) {
    image.style.visibility = "hidden"; // 处理此处的图像
}
```

17.5.2 DOM Range API

DOM Range API 由一个接口构成，即 `Range`。`Range` 对象表示文档内容的一个连

续范围（注8），包括指定的开始位置和结束位置。许多显示文本和文档的应用程序允许用户通过拖移鼠标选择部分文档（注9）。当文档树的一个节点处在一个范围内时，我们常说那个节点“被选中”了，即使 Range 对象没有对用户作出的选择动作进行任何处理。当一个范围的开始位置和结束位置相同时，我们说范围被“折叠”了。在这种情况下，Range 对象表示文档中的一个位置或一个插入点。

Range 对象提供了定义范围的开始位置和结束位置的方法，复制和删除范围内容的方法以及在范围的开始位置插入节点的方法。对 Range API 的支持是可选的。在本书编写过程中，Netscape 6.1 支持它。IE 5 支持与 Range API 相似的专有 API，但它们不兼容。可以用下面的代码测试对 Range 的支持：

```
document.implementation.hasFeature("Range", "2.0");// 如果 Range 被支持，则为 true。
```

17.5.2.1 开始位置和结束位置

范围的开始位置和结束位置由两个值指定。第一个值是文档节点，通常是 Document 对象、Element 对象或 Text 对象。第二个值是一个数字，它表示节点中的位置。当节点是文档或元素时，该数字表示文档或元素的子节点之间的位置。例如，0 偏移量表示紧邻第一个子节点的前一个位置，偏移量为 1 表示第一个子节点之后、第二个子节点之前的位置。如果指定的节点是一个 Text 节点（或其他基于文本的节点，如 Comment），该数字表示文本中的字符间的位置。偏移量 0 表示第一个文本字符前的位置，偏移量 1 表示第一个文本字符和第二个文本字符之间的位置，以此类推。用这种方式指定开始位置和结束位置，则范围就表示开始位置和结束位置之间的所有节点和（或）字符。Range 接口的真正强大之处在于开始位置和结束位置可以处于文档的不同节点之间，因此一个范围可以跨越多个（局部的）Element 节点和 Text 节点。

我采用 DOM 规范来说明一个范围表示的文档内容的概念，从而示范各种操作范围

注 8： 也就是说，这是逻辑上的范围的连续。在 Arabic 和 Hebrew 这样的双向语言中，文档的逻辑连续范围在显示时可能是视觉不连续的。

注 9： 虽然 Web 浏览器通常允许用户选择文档内容，但当前的 DOM 2 级标准不允许 JavaScript 访问该范围中的内容，所以没有一种标准的方法可以获取对应于用户选择的 Range 对象。

的方法，文档内容以 HTML 源代码的形式显示，范围的内容以粗体显示。例如，下面的代码代表一个范围，该范围从 `<body>` 节点的位置 0 处开始，继续到 `<h1>` 节点中包含的 `Text` 节点的位置 8：

```
<body><h1>Document Title</h1></body>
```

调用 `Document` 对象的 `createRange()` 方法可以创建 `Range` 对象：

```
var r = document.createRange();
```

新创建的范围的开始位置和结束位置都初始化为 `Document` 对象的位置 0。在对范围做处理之前，必须把它的开始位置和结束位置设置为想要的文档范围。有几种方法可以做到这一点。最常用的方法是调用 `setStart()` 方法和 `setEnd()` 方法指定开始点和结束点，将其传递给每个方法一个节点和节点中的一个位置。

设置开始位置和结束位置的高级方法是调用 `setStartBefore()`、`setStartAfter()`、`setEndBefore()` 或 `setEndAfter()` 方法。这些方法都以一个节点作为参数。它们把 `Range` 对象的开始位置或结束位置设置在那个节点的父节点的、指定节点之前或之后。

最后，如果你想定义一个 `Range` 对象来表示文档的一个 `Node` 或子树，那么可以调用 `selectNode()` 方法或 `selectNodeContent()` 方法。这两个方法都以一个节点为参数。`selectNode()` 方法设置的开始位置和结束位置位于指定节点的父节点的之前或之后，定义的范围包括指定节点和它的所有子节点。`selectNodeContent()` 方法设置的开始位置是指定节点的第一个子节点之前的位置，结束位置是指定节点的最后一个子节点之后的位置，生成的范围包括指定节点的所有子节点，但不包括该节点自身。

17.5.2.2 操作范围

定义了一个范围后，可以用它做很多有趣的事情。要删除一个范围中的文档内容，只需要调用 `Range` 对象的 `deleteContents()` 方法。当范围包括部分被选中的 `Text` 节点时，删除操作有点麻烦。考虑下面的范围：

```
<p>This is <i>only</i> a test
```

在调用 `deleteContents()` 后，文本受影响的部分如下：

```
<p>This<i>ly</i> a test
```

即使 `<i>` 元素（部分地）包含在 `Range` 对象中，在删除操作后该元素仍然留在文档树中（但内容被改变了）。

如果你既想把一个范围的内容从文档中删除，又想保存提取的内容（可以作为粘贴操作的一部分再插入），则应该使用 `extractContents()` 方法而不应该使用 `deleteContents()` 方法。该方法将从文档树中删除节点，并把它插入一个 `DocumentFragment` 对象（本章前面介绍过），返回该对象。当范围包含部分被选中的节点时，该节点仍会保留在文档树中，且内容进行了必要的修改，该节点的一个副本（参阅 `Node.cloneNode()`）将被创建（或修改）以便插入 `DocumentFragment` 对象中。再次考虑前面的例子。如果调用 `extractContents()` 方法而不是 `deleteContents()` 方法，效果与前面所示的文档相同，返回的 `DocumentFragment` 对象包含下面的代码段：

```
is <i>on</i>
```

当对文档执行剪切操作时，也可以使用 `extractContents()` 方法。如果你想执行复制操作，提取内容，而且又不从文档中删除它，则应该用 `cloneContents()` 方法而不是 `extractContents()` 方法（注 10）。

除了指定要删除或复制的文本的边界，范围的开始位置还可以用于指示文档中的一个插入点。范围的 `insertNode()` 方法将把指定的节点（和它的所有子节点）插入文档中范围的开始位置。如果指定的节点已经是文档树的一部分，那么它将从当前位置移到范围指定的再插入位置。如果指定的节点是 `DocumentFragment` 对象，插入的将是它的所有子节点，而不是它自身。

`Range` 对象另一个有用的方法是 `surroundContents()`。该方法将把范围内容的父节点重定为指定的节点，并把那个节点插入范围所在处的节点中。例如，把一个新创建的 `<i>` 节点传递给 `surroundContents()` 方法，可以把范围：

注 10： 实现字处理器形式的剪切、复制和粘贴操作实际上更复杂。对复杂的文档树进行的简单的范围操作不一定会在文档的线性视图中生成希望的剪切一粘贴行为。

```
This is only a test
```

转换为:

```
This is <i>only</i> a test
```

注意, 因为开始标记和结束标记必须正确地嵌套在 HTML 文件中, 所以 `surroundContents()` 不能用于包含部分被选中的节点 (除 Text 节点外) 的范围 (否则会抛出异常)。例如, 前面用于说明 `deleteContents()` 方法的范围就不能使用 `surroundContents()` 方法。

Range 对象还有各种其他特性。可以用 `compareBoundaryPoints()` 方法比较两个不同范围的边界, 用 `cloneRange()` 方法复制范围, 用 `toString()` 提取范围内容 (不包括标记) 的纯文本副本。通过使用只读属性 `startContainer`、`startOffset`、`endContainer` 和 `endOffset` 可以访问范围的开始位置和结束位置。所有有效范围的开始点和结束点都共享文档树中的同一个祖先, 即使它是文档树根部的 **Document** 对象。用范围的 `commonAncestorContainer` 属性可以找到这个共同祖先。

第十八章

级联样式表和 动态HTML

级联样式表 (Cascading Style Sheet, CSS) 是指定 HTML 文档 (或 XML 文档) 的视觉表现的标准 (注 1)。理论上说来, 应该用 HTML 标记设置文档的结构, 而不采用 `` 这样不建议使用的 HTML 标记指定文档的外观, 而用 CSS 定义样式表, 设置显示文档的结构元素的方式。例如, 可以用 CSS 设置 `<h1>` 标记定义的一级标题, 以粗体、无衬线、居中、大写和 24 point 的字符显示它。

CSS 技术上主要由图形设计者或其他注重 HTML 文档的精确视觉效果的用户使用。客户端 JavaScript 的程序设计者会对它感兴趣, 因为文档对象模型允许将应用到每个文档元素上的样式进行脚本化。使用 CSS 和 JavaScript, 可以创造出各种视觉效果, 这些效果可以统称为动态 HTML (DHTML) (注 2)。

利用脚本化 CSS 样式的能力, 可以动态地改变颜色、字体等。更重要的是, 可以用它设置和改变元素的位置, 甚至隐藏或显示元素。这意味着可以用 DHTML 技术创造动画迁移的效果, 例如使内容从右边滑入, 或者展开、折叠大纲列表, 这样用户便可以控制列表中显示的信息量。

本章开头概述了 CSS 样式表和用 CSS 样式指定文档元素的位置和可见性的方法。然

注 1: 在 CSS2 标准中, 它同时也是听觉标准。

注 2: 许多高级 DHTML 效果还涉及事件处理方法, 参阅第十九章。

后解释了如何用2级DOM标准定义的API脚本化CSS样式。最后,它说明了Netscape 4 和 Internet Explorer 4 中用于实现 DHTML 效果的 API, 这些 API 是非标准的, 而且与浏览器有关。

18.1 CSS 的样式和样式表

CSS 样式是由一个名称/值的性质对列表指定的, 性质对之间用分号隔开, 名称性质和值性质之间用冒号隔开。例如, 下面的样式设置了粗体、蓝色、有下划线的文本:

```
font-weight: bold; color: blue; text-decoration: underline;
```

CSS 标准定义了相当多样式性质。除了音频样式表专用的性质外, 表18-1还列出了其他所有性质。你不必掌握所有的性质、它们的值和意义。但当你熟悉了CSS, 并在文档或脚本中使用它时, 会发现这个表是个快捷的参考。要看CSS更完整的说明, 可以参阅《Cascading Style Sheets: The Definitive Guide》, 由 Eric Meyer 编著, O'Reilly 公司出版, 或者参阅《Dynamic HTML: The Definitive Guide》, 由 Danny Goodman 编著, O'Reilly 公司出版。也可以阅读 CSS 规范, 在 <http://www.w3c.org/TR/REC-CSS2/> 可以找到它。

表18-1的第二列说明了每个样式性质可用的值。它采用CSS规范的语法。以 *fixed-width font* (等宽字体) 显示的项口是关键字, 应该完全以所示的形式出现。以 *italics* (斜体字) 显示的项目指定了元素类型, 如 *string* 或 *length*。注意, *length* 类型是数字后面跟单位 (如 *px* 表示像素)。其他类型详见 CSS 参考手册。以 *italic fixed-width font* (斜体等宽字体) 显示的项目表示其他 CSS 性质允许使用的值的集合。除了该表中列出的值, 每个样式性质都有值 “inherit”, 说明它应该继承自己父元素的值。

由 “|” 分隔的值是两者选其一的, 必须指定其中的一个。由 “||” 分隔的值是可选项, 至少必须指定一个, 也可以指定多个, 而且它们可以以任何顺序出现。方括号 “[]” 用于对值分组。星号 “*” 说明前面的值或值组可以出现 0 次或多次。加号 “+” 说明前面的值或值组可以出现一次或多次。问号 “?” 说明前面的项目是可选的, 可以出现 0 次或一次。大括号中的数字说明重复的次数。例如, {2} 说明前面的项目可

以重复出现两次, {1, 4} 说明前面的项目至少出现 1 次, 至多出现 4 次 (这种重复语法看来很熟悉, 因为 JavaScript 的正则表达式也用这样的语法, 第十章讨论过它)。

表 18-1: CSS 样式性质和它们的值

名称	值
background	<i>[background-color background-image background-repeat background-attachment background-position]</i>
background-attachment	scroll fixed
background-color	<i>color</i> transparent
background-image	url(<i>url</i>) none
background-position	<i>[[percentage length]{1,2} [[top center bottom] left center right]]</i>
background-repeat	repeat repeat-x repeat-y no-repeat
border	<i>[border-width border-style color]</i>
border-collapse	collapse separate
border-color	<i>color{1,4}</i> transparent
border-spacing	<i>length length?</i>
border-style	<i>[none hidden dotted dashed solid double groove ridge inset outset]{1,4}</i>
border-top	<i>[border-top-width border-top-style color]</i>
border-right	
border-bottom	
border-left	
border-top-color	<i>color</i>
border-right-color	
border-bottom-color	
border-left-color	
border-top-style	<i>none hidden dotted dashed solid double groove ridge inset outset</i>
border-right-style	
border-bottom-style	
border-left-style	

表 18-1: CSS 样式性质和它们的值 (续)

名称	值
border-top-width	thin medium thick <i>length</i>
border-right-width	
border-bottom-width	
border-left-width	
border width	{thin medium thick <i>length</i> }[1,4]
bottom	<i>length</i> percentage auto
caption-side	top bottom left right
clear	none left right both
clip	[rect({ <i>length</i> auto}{4})] auto
color	<i>color</i>
content	[<i>string</i> url(<i>url</i>) <i>counter</i> attr(<i>attribute-name</i>) open-quote close-quote no-open-cuote no-close-quote +]
counter-increment	[<i>identifier integer</i> ? + none
counter-reset	[<i>identifier integer</i> ? + none
cursor	[{url(<i>url</i>),}* [auto crosshair default pointer move e-resize ne-resize nw-resize n-resize se-resize sw-resize s-resize w-resize text wait help]]
direction	ltr rtl
display	inline block list-item run-in compact marker table inline-table table-row-group table-header-group table-footer-group table-row table-column-group table-column table-cell table-caption none
empty-cells	show hide
float	left right none
font	[{ <i>font-style</i> } <i>font-variant</i> { <i>font-weight</i> }? <i>font-size</i> [/ <i>line-height</i>]? <i>font-family</i>] caption icon menu message-box small-caption status-bar
font-family	[<i>family-name</i> serif sans-serif monospace cursive fantasy ,]+

表 18-1: CSS 样式特性和它们的值 (续)

名称	值
font-size	xx-small x-small small medium large x-large xx-large smaller larger <i>length</i> <i>percentage</i>
font-size-adjust	<i>number</i> none
font-stretch	normal wider narrower ultra-condensed extra-condensed condensed semi-condensed semi-expanded expanded extra-expanded ultra-expanded
font-style	normal italic oblique
font-variant	normal small-caps
font-weight	normal bold bolder lighter 100 200 300 400 500 600 700 800 900
height	<i>length</i> <i>percentage</i> auto
left	<i>length</i> <i>percentage</i> auto
letter-spacing	normal <i>length</i>
line-height	normal <i>number</i> <i>length</i> <i>percentage</i>
list-style	[<i>list-style-type</i> ! <i>list-style-position</i> ! <i>list-style-image</i>]
list-style-image	url(<i>url</i>) none
list-style-position	inside outside
list-style-type	disc circle square decimal decimal-leading-zero lower-roman upper-roman lower-greek lower-alpha lower-latin upper-alpha upper-latin hebrew armenian georgian cjk-ideographic hiragana katakana hiragana-iroha katakana-iroha none
margin	[<i>length</i> <i>percentage</i> auto]{1,4}
margin-top	<i>length</i> <i>percentage</i> auto
margin-right	
margin-bottom	
margin-left	
marker-offset	<i>length</i> auto
marks	[crop /cross] none

表 18-1: CSS 样式性质和它们的值 (续)

名称	值
max-height	<i>length</i> <i>percentage</i> none
max-width	<i>length</i> <i>percentage</i> none
min-height	<i>length</i> <i>percentage</i>
min-width	<i>length</i> <i>percentage</i>
orphans	<i>integer</i>
outline	[<i>outline-color</i> <i>outline-style</i> <i>outline-width</i>]
outline-color	<i>color</i> invert
outline-style	none hidden dotted dashed solid double groove ridge inset outset
outline-width	thin medium thick <i>length</i>
overflow	visible hidden scroll auto
padding	[<i>length</i> <i>percentage</i>]{1,4}
padding-top	<i>length</i> <i>percentage</i>
padding-right	
padding-bottom	
padding-left	
page	<i>identifier</i> auto
page-break-after	auto always avoid left right
page-break-before	auto always avoid left right
page-break-inside	avoid auto
position	static relative absolute fixed
quotes	[<i>string string</i>]+ none
right	<i>length</i> <i>percentage</i> auto
size	<i>length</i> {1,2} auto portrait landscape
table-layout	auto fixed
text-align	left right center justify <i>string</i>
text-decoration	none [<i>underline</i> <i>overline</i> <i>line-through</i> <i>blink</i>]

表 18-1: CSS 样式性质和它们的值 (续)

名称	值
text-indent	<i>length</i> <i>percentage</i>
text-shadow	none [<i>color</i> <i>length length length?</i> ,]* [<i>color</i> <i>length length length?</i>]
text-transform	capitalize uppercase lowercase none
top	<i>length</i> <i>percentage</i> auto
unicode-bidi	normal embed bidi-override
vertical-align	baseline sub super top text-top middle bottom text-bottom <i>percentage</i> <i>length</i>
visibility	visible hidden collapse
white-space	normal pre nowrap
widows	<i>integer</i>
width	<i>length</i> <i>percentage</i> auto
word-spacing	normal <i>length</i>
z-index	auto <i>integer</i>

CSS 标准允许用特殊快捷性质把常常一起使用的样式性质组合在一起。例如，可以用一个 `font` 性质同时设置 `font-family`、`font-size`、`font-style` 和 `font-weight` 性质：

```
font: bold italic 24pt Helvetica;
```

事实上，表 18-1 中列出的某些性质自身就是快捷属性。如 `margin` 和 `padding`，它们是用来指定元素每边的页边距、补白和边线的快捷属性。因此，可以用 `margin-left`、`margin-right`、`margin-top` 和 `margin-bottom` 代替 `margin` 属性，反之亦然。

18.1.1 给文档元素应用样式规则

把样式性质应用到文档元素的方法有很多。方法之一是在 HTML 标记的 `style` 性质中使用它们。例如，要设置一个段的页边距，可以使用如下标记：

```
<p style="margin-left: 1in; margin-right: 1in;">
```

CSS 的一个重要目标是把文档内容和文档结构与文档外观分开。用 HTML 标记的 style 性质设置样式无法实现这一点（尽管对 DHTML 来说这是一种有效的方法）。要实现文档结构和外观的分离，可以使用样式表（Style Sheet），它把所有样式信息集中在一个地方。CSS 样式表由样式规则的集合构成。每条规则以一个选择器开头，它指定要应用这条样式规则的文档元素，其后是用大括号括起来的样式属性和它们值的集合。最简单的规则是为一个或多个指定的标记名定义样式。例如，下面的规则设置 <body> 标记的页边距和背景颜色：

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff; }
```

下面的规则把标题 <h1> 和 <h2> 中的文本设置为居中显示：

```
h1, h2 { text-align: center; }
```

注意上面例子中逗号的用法，它用于分隔要应用样式的标记名。如果省略了逗号，那么选择器指定了一条环境规则，只在一个标记嵌套在另一个标记中时应用。例如，下面的规则指定用斜体字显示 <blockquote> 标记，但 <blockquote> 标记中嵌套的 <i> 标记中的文本仍然以纯文本、非斜体的样式显示：

```
blockquote { font-style: italic; }  
blockquote i { font-style: normal; }
```

另一种样式表规则使用不同的选择器，指定要应用样式的元素的类。元素的类由 HTML 标记的 class 性质定义。例如，下列规则规定，以粗体字显示所有 class 性质为 “attention” 的标记：

```
.attention { font-weight: bold; }
```

类选择器可以和标记名选择器联合使用。下面的规则规定，在 <p> 标记有 class = “attention” 性质时，除了以粗体显示该标记（前面规则设置的）外，还要用红色来显示：

```
p.attention { color: red; }
```

样式表还有只应用于具有指定的 id 性质的元素规则。下面的规则规定，不显示 id 性质为 “p1” 的元素：

```
fp { visibility: hidden; }
```

我们已经见过 `id` 性质，DOM 函数 `getElementById()` 方法用它来返回文档的元素。当我们想操作某个元素的样式时，这种特定元素的样式表规则很有用。例如，考虑上一条规则，脚本可以把 `visibility` 性质的值从 `hidden` 切换为 `visible`，从而使元素动态地显示出来。

18.1.2 关联样式表和文档

可以把样式表放置在文档 `<head>` 部分中的 `<style>` 和 `</style>` 标记之间，使它合并到 HTML 文档中。也可以把样式表存储为一个单独的文件，在 HTML 文档中用标记 `<link>` 引用它。还可以联合使用这两种方法，方法是在 `<style>` 标记之间创建一个与文档有关的样式表，让它用 `@import` 引用或导入一个独立于文档的样式表。详情请参见 CSS 参考手册部分的 `@import` 中的内容。

18.1.3 级联

CSS 中的 C 代表“cascading”（级联）。这个术语说明应用到文档中指定元素的样式规则来自不同资源的级联。每个 Web 浏览器通常有自己的一套用于 HTML 元素的默认元素样式，而且允许用户使用自己定义的样式表覆盖这些默认样式。文档的作者可以在 `<style>` 标记中定义样式表，也可以在外部文件中定义样式表，该样式表由另一个样式表链接或导入文档。文档的作者还可以用 HTML 的 `style` 性质为元素单独定义内联样式。

CSS 规范包括一套完整的规则，用于确定级联中的哪条规则优先于其他规则。但你只需要知道，用户样式表覆盖默认的浏览器样式表，作者样式表覆盖用户样式表，内联样式覆盖所有样式表。这条通用规则的一个特例是，值包括 `!important` 修饰符的用户样式性质覆盖作者样式。在一个样式表中，如果一个元素上应用了多条样式规则，最详细的规则定义的样式将覆盖不太详细的规则定义的发生冲突的样式。指定元素 `id` 的规则最详细，其次是指定 `class` 属性的规则。仅指定一个标记名的规则最不详细，指定多个嵌套标记名的规则比指定一个标记名的规则详细。

18.1.4 CSS 的版本

在本书编写的过程中，有两种版本的 CSS 标准。CSS 1 是 1996 年 10 月采用的，为设置颜色、字体、页边距和其他基本样式定义了性质。Netscape 4 和 Internet Explorer 4 至少部分实现了对 CSS 1 的支持。该标准的第二个版本 CSS2 是 1998 年 5 月采用的，它定义了许多更高级的特性，最著名的是对元素绝对定位技术的支持。只有第六代浏览器支持 CSS2 的高级特性。但是，CSS2 的关键定位特性是以单独的 CSS-Positioning (CSS-P) 研究开始标准化的，因此某些 DHTML 特性在第四代浏览器中可用。对第三代 CSS 标准的研究工作仍在继续。在 <http://www.w3.org/Style/CSS/> 中可以找到 CSS 规范和草案。

18.1.5 CSS 示例

例 18-1 是一个 HTML 文件，它定义并使用了一个样式表，它示范了前面说明了标记名、类和基于 ID 的样式规则。它还有一个用 style 性质定义的内联样式。记住，这个例子只是 CSS 语法和功能的概述。对 CSS 的全面介绍不在本书范围之内。

例 18-1: 定义并使用级联样式表

```
<head>
<style type="text/css">
/* 指定以蓝色斜体字显示标题 */
h1, h2 { color: blue; font-style: italic }

/*
 * 以大粗体文本、大边缘和带有宽红边线的黄色背景显示 class="WARNING" 的元素
 */
.WARNING {
    font-weight: bold;
    font-size: 150%;
    margin: 0 1in 0 1in; /* 右上左下 */
    background-color: yellow;
    border: solid red 8px;
    padding: 10px;      /* 四边有 10 像素的补白 */
}

/*
 * class="WARNING" 的元素中的 h1 和 h2 标题的文本用蓝色斜体字居中显示
 */
.WARNING h1, .WARNING h2 { text-align: center }
/* id="P23" 的元素以大写字母居中显示 */
#P23 {
```

```
        text-align: center;
        text-transform: uppercase;
    }
</style>
</head>
<body>
<h1>Cascading Style Sheets Demo</h1>

<div class="WARNING">
<h2>Warning</h2>
This is a warning!
Notice how it grabs your attention with its bold text and bright colors.
Also notice that the heading is centered and in blue italics.
</div>

<p id="P23">
This paragraph is centered<br>
and appears in uppercase letters.<br>
<span style="text-transform: none">
Here we explicitly use an inline style to override the uppercase letters.
</span>
</p>
</body>
```

18.2 用 CSS 进行元素定位

对于DHTML内容开发者来说，最重要的CSS特性是能用普通的CSS样式性质设置文档元素的可见性、大小和精确位置。要进行DHTML程序设计，理解这些样式性质的作用原理很重要。表18-2总结了这些性质，此后的小节对它们进行了详细说明。

表 18-2: CSS 的定位和可见性性质

性质	说明
position	设置元素应用的定位类型
top, left	设置元素上边界和左边界的位置
bottom, right	设置元素下边界和右边界的位置
width, height	设置元素的大小
z-index	设置元素相对于其他重叠元素“堆叠顺序”，定义元素位置的第三维
display	设置如何和是否显示一个元素

表 18-2: CSS 的定位和可见性性质 (续)

性质	说明
visibility	设置元素是否可见
clip	定义元素的“裁剪区”，只显示元素在这个区域中的部分
overflow	设置当元素比分配给它的空间大时应该采取什么操作

18.2.1 DHTML 的关键: position 性质

CSS 的 position 性质设置了对要应用到元素的定位类型。该性质有四个可用的值，如下所示：

static

这是个默认值，指定根据文档内容的正常流动对元素定位（对大多数西方语言而言，是从左到右，从上到下流动的）。静态定位元素不是 DHTML 元素，不能用 top、left 等性质定位。要对文档元素使用 DHTML 定位技术，必须把它的 position 性质设置为其他三个值之一。

absolute

用这个值可以设置元素相对于它的包含元素的位置。绝对定位的元素将独立于其他元素定位，但不属于静态定位的元素流。绝对定位的元素可以相对于文档的 <body> 标记进行定位。如果它嵌套在另一个绝对定位的元素中，则相对于那个元素定位。这是 DHTML 最常用的定位类型。

fixed

用这个值可以设置元素在浏览器窗口中的位置。具有 fixed 定位的元素不随文档其余的元素滚动，因此可以用来实现框架效果。与绝对定位的元素一样，固定定位的元素独立于其他元素，并且不属于文档流。固定定位是 CSS2 的特性，第四代浏览器不支持它（Netscape 6 和用于 Macintosh 的 IE 5 支持它，但用于 Windows 的 IE 5 和 IE 6 都不支持它）。

relative

当 position 性质被设为 relative 时，将根据常规流布置元素，然后相对于它在常规流中的位置进行调整。在常规文档流中分配给元素的空间仍然分配给

它，它两边的元素不会向它靠近来填充那个空间，但它们也不会从元素的新位置被挤走。相对定位对某些静态图形设计比较有用，但不常用于DHTML效果。

18.2.2 设置元素的位置和大小

如果把元素的`position`性质设置为`static`以外的值，就可以用`left`、`top`、`right`、`bottom`这些性质的组合来定义元素的位置。最常用的定位方法是设置`left`和`top`性质，这种方法可以指定元素的左边界和包含元素（通常是文档本身）的左边界之间的距离，以及元素的上边界和包容元素的上边界之间的距离。例如，要把元素放在距文档上边界100像素、左边界100像素的位置，可以用下面的代码在`style`性质中设置CSS样式：

```
<div style="position: absolute; left: 100px; top: 100px;">
```

与动态元素相对定位的包容元素不必与文档源代码中定义的包容元素相一致。因为动态元素不属于常规元素流，所以它们的位置不相对于定义它们的静态包容元素。大部分动态元素相对于文档自身（`<body>`标记）的位置定位。例外情况是定义在其他动态元素中的动态元素。在这种情况下，嵌套的动态元素将被定位在相对于动态祖先最近的位置。

虽然用`left`和`top`性质设置元素左上角的位置很常用，但也可以用`right`和`bottom`性质设置元素相对于包容元素的下边界和右边界的下边界和右边界。例如，要使元素的右下角位于文档的右下角（假定该元素没有嵌套在其他动态元素中），可以采用如下的样式：

```
position: absolute; right: 0px; bottom: 0px;
```

要使元素的上边界距离窗口顶部10像素，右边界距离窗口右边界10像素，可以采用下列样式：

```
position: fixed; right: 10px; top: 10px;
```

注意，`right`和`bottom`性质是CSS标准的新性质，第四代浏览器不支持它们，就像不支持`top`和`left`性质一样。

除了元素定位外，用CSS可以设置元素的大小。提供`width`和`height`样式性质的

值是最常用的方法。例如, 下面的HTML代码创建了一个没有内容的绝对定位元素。它的 width、height 和 background-color 性质使它出现在一个小蓝框中:

```
<div style="position: absolute; left: 10px; right: 10px;
          width: 10px; height: 10px; background-color: blue">
</div>
```

另一种设置元素宽度的方法是设置 left 和 right 性质的值。同样, 也可以通过设置 top 和 bottom 性质来设置元素的高度。但如果同时设置了 left、right 和 width 性质, width 性质将覆盖 right 性质。如果元素高度的约束过多, 则 height 的优先级比 bottom 高。

记住, 不必设置所有动态元素的大小。某些元素(如图像)本身有大小。另外, 对于含有文本或其他流内容的动态元素来说, 只设置元素的宽度, 并允许由元素内容的布局自动确定高度通常比较有效。

在前面的定位示例中, 位置和大小性质的值都有后缀 px。px 代表像素。CSS 标准允许用多种单位进行度量, 包括英寸(in)、厘米(cm)、点(pt)和 em——当前字体的行高度单位。DHTML 程序设计常以像素为单位。注意, CSS 标准要求指定单位。如果省略了单位设置, 有些浏览器会假定单位为像素, 但你不应该依赖这一行为。

除了可以用上面所示的单位设置绝对位置和大小外, CSS 还允许用包容元素大小的百分比设置元素的位置和大小。例如, 下面的HTML代码创建了一个具有黑色边线的空元素, 它的宽度和高度都是包容元素(或浏览器窗口)的一半, 并且位于包容元素的中间:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
          border: 2px solid black">
</div>
```

18.2.2.1 元素大小和位置的深入介绍

了解 left、right、width、top、bottom 和 height 性质的细节很重要。首先, width 和 height 只设置了元素的内容区的大小, 不包括元素的补白、边线或页边距占用的额外空间。要确定带边线的元素在屏幕上显示的大小, 必须把左右补白和左右边线的宽度都加到元素宽度上, 把上下补白和上下边线的宽度加到元素的高度上。

由于width和height性质只设置了元素的内容区, 所以你可能认为left和top (以及right和bottom) 性质是相对于包容元素的内容区来衡量的。事实上, CSS标准规定, 这两个值相对于包容元素的补白的外边界 (即元素边线的内边缘) 来衡量。

让我们来看一个例子, 会使你了解得更清楚一些。假定已经创建了一个动态定位的包容元素, 它的内容区的四周补白是10个像素, 补白四周的边线宽为5个像素。假定在这个容器中动态定位一个子元素。如果把子元素的left性质设为“0px”, 你会发现子元素的左边界正对着容器边线的内边界。在这种设置下, 子元素将覆盖容器的补白, 可假定补白为空 (因为这正是补白的用途)。如果想把子元素定位在容器的内容区的左上角, 则应该把left性质和top性质都设为“10px”。图18-1有助于理解这一点。

现在你已经知道width和height只设置元素内容区的大小, left、top、right和bottom性质是相对于包容元素的补白进行测量的, 此外还有一点必须注意, 即从Windows使用的Internet Explorer 4到5.5 (除了IE 5的Mac版本) 实现的width和height性质不正确, 它们包括元素的边线和补白 (但不包括页边距)。例如, 如果把元素的宽度设为100像素, 在它左右分别放置10像素的页边距和5像素的边线, 那么在那些有bug的Internet Explorer版本中, 元素的内容区只有70像素宽。

在IE 6中, 如果浏览器使用标准模式, 那么CSS的定位性质和大小性质都能正确作用。如果浏览器使用不兼容的模式, 则它们不正确 (但与早期版本兼容)。标准模式 (即CSS的“盒子模型”的正确实现) 是由文档开始处的<!DOCTYPE>标记引发的, 它声明了文档遵守HTML 4.0 (及以后的版本) 或其他版本的XHTML标准。例如, 下列三个HTML文档类型声明将使IE 6以标准模式显示文档:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Strict//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

Netscape 6和Mozilla浏览器能正确处理width和height性质。但这些浏览器也像IE一样有标准模式和兼容模式。没有<!DOCTYPE>声明, Netscape浏览器将处于怪异模式, 在这种模式中, 它模仿了Netscape 4的一些 (相对较少) 不标准的布局行为。如果有<!DOCTYPE>声明, 该浏览器将不再与Netscape 4兼容, 从而正确地实现标准。

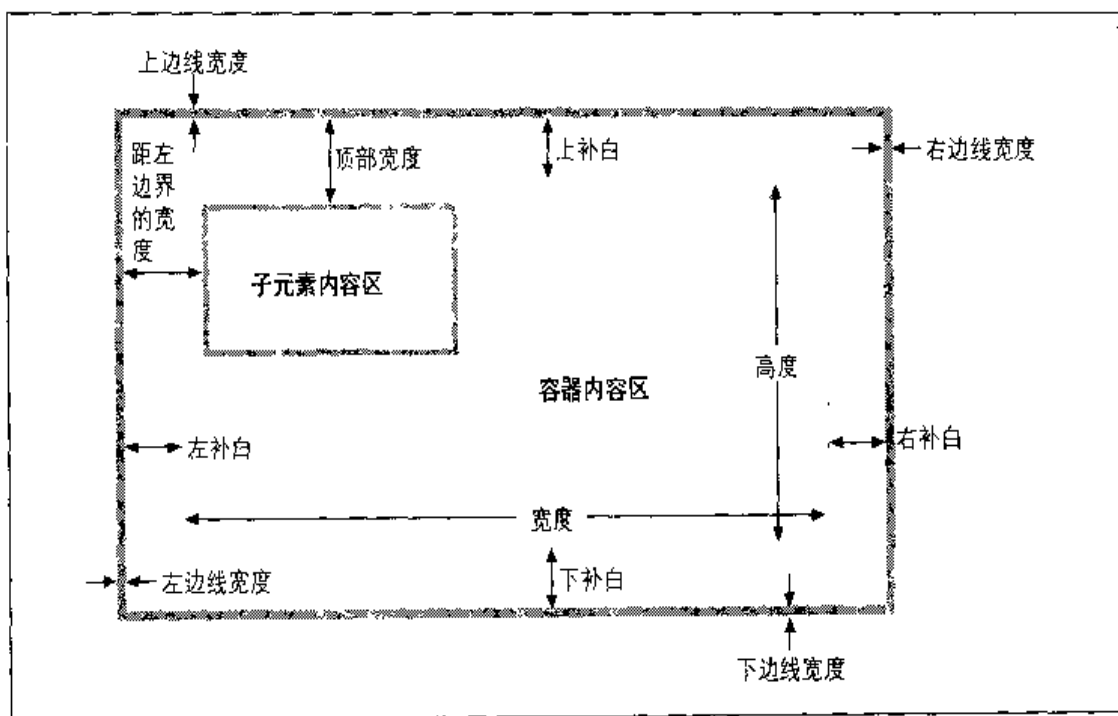


图 18-1: 动态定位的容器和具有 CSS 性质的子元素

18.2.3 第三维: z-index

我们已经知道, 用 `left`、`top`、`right` 和 `bottom` 性质可以在包容元素的二维平面内设置元素的 X 坐标和 Y 坐标。z-index 性质定义了元素的第三维, 它可以设置元素的堆叠顺序, 说明哪些元素在其他元素之上绘制。z-index 性质是个整数, 默认值为 0, 可以为正也可以为负 (第四代浏览器不支持负的 z-index 性质)。当两个或多个元素重叠时, 它们的绘制顺序是从最低的 z-index 到最高的 z-index, 具有 z-index 最大值的元素出现在所有元素的最上边。如果重叠元素的 z-index 值相同, 将以它们在文档中出现的顺序绘制这些元素, 最后一个重叠元素出现在最上边。

注意, z-index 堆叠只适用于兄弟元素 (即同一个容器的子元素)。如果两个不是兄弟的元素重叠, 设置 z-index 性质时不能指定哪个元素位于上方, 必须设置那两个重叠元素的兄弟容器的 z-index 性质。

没被定位的元素通常以防止重叠的方式放置, 所以它们不需要应用 z-index 性质。不过它们有默认的 z-index 值 0, 这意味着, z-index 性质为正数的定位元素出现在常规文档流的上边, z-index 性质为负数的定位元素出现在常规文档流底部。

最后要注意,当`z-index`性质用在`<iframe>`标记中时,有些浏览器不实现`z-index`性质,你会发现,内联框架不按照指定的堆叠顺序,浮在其他元素之上。在其他窗口元素中也会遇到同样的问题,如`<select>`下拉菜单。第四代浏览器将无视`z-index`设置,在绝对定位元素之上显示所有表单控制元素。

18.2.4 元素的显示和可见性

有两种CSS性质可以用来改变文档元素的可见性,即`visibility`和`display`。性质`visibility`比较简单,当把它设为`hidden`时,就不显示元素,当把它设为`visible`时,就显示元素。性质`display`用途更多,它用于指定显示的元素类型,它可以指定一个元素是块元素、内联元素、列表项目,等等。但当`display`性质被设为`none`时,受影响的元素不会显示出来,甚至根本不被放置。

`visibility`和`display`样式性质之间的区别在于,它们对非动态定位的元素的影响。对于出现在常规布局流中的元素(`position`性质值为`static`或`relative`),可以把`visibility`性质设为`none`,使元素不可见,但会在文档布局中保留它的空间。这样的元素可以在不改变文档布局的情况下,反复隐藏或显示。但如果元素的`display`性质被设为`none`,就不会在文档布局中为它分配空间,它两边的元素都会靠拢,就像它不存在一样(在使用绝对定位和固定定位的元素时,`visibility`和`display`性质的效果一样,因为这些元素都不是文档布局的一部分)。通常在使用动态定位元素时,可以使用`visibility`性质。在创建展开或折叠的大纲这样的元素时,用`display`元素比较有效。

注意,除非你想用JavaScript动态地设置`visibility`和`display`性质,使元素在某种情况下可见(注3),否则用它们使元素不可见就变得毫无意义。在本章后面的小节中你会看到如何实现这一点。

注3: 有一个例外,如果创建一个依赖于CSS的文档,那么可以用下面的代码警告使用不支持CSS代码的浏览器的用户:

```
<p style "display:none">
This document uses CSS, but your browser doesn't support it!
</p>
```

18.2.5 部分可见性：overflow 和 clip 性质

使用 visibility 性质可以完全隐藏一个文档元素。使用 overflow 和 clip 性质则可以显示元素的一部分。overflow 性质指定了在元素内容超过指定大小（如用 width 和 height 性质设置的大小）时会发生什么。该性质可以采用的值和它们的含义如下：

visible

内容可以溢出，如果必要可以绘制在元素框之外，这是默认值。

hidden

剪切并隐藏溢出的内容，以免在大小和定位性质定义的区域外绘制任何内容。

scroll

元素框永久具有水平滚动条和垂直滚动条。如果内容超过了元素框的大小，使用滚动条就可以查看超出的内容。这个值只在文档显示在计算机屏幕中时有用，当文档印刷在纸上时，滚动条没有任何用处。

auto

只在内容超出元素大小时才显示滚动条，而不是永久显示。

用 overflow 属性可以指定当元素内容超出元素框时发生什么情况，用 clip 属性则可以明确地设置显示元素的哪个部分（无论元素是否溢出）。该性质对脚本化的 DHTML 效果非常有用，在这些效果中，元素可以逐渐显示出来或逐渐消失。

clip 属性的值指定了元素的剪切区域。在 CSS2 中，剪切区域是矩形的，但是 clip 性质的语法为将来的版本支持其他剪切形状创造了可能性。clip 性质的语法如下：

```
rect(top right bottom left)
```

top、right、bottom 和 left 值指定了剪切矩形相对于元素框的左上角的边界（注 4）。例如，要显示一个元素的 100 × 100 像素部分，可以用如下方法设置元素的 style 性质：

注 4： CSS 2 标准最初规定，用这四个值指定剪切区域的边界和元素框每个相应边界之间的偏移量。但大部分浏览器都实现错了，它们将 right 和 bottom 的值解释为左边界和上边界之间的偏移量。因为所有的实现都不遵守 CSS 2 规范，所以修改了这个规范以匹配各个实现。

```
style="clip: rect(0px 100px 100px 0px);"
```

注意，括号中的四个值都是长度值，所以必须有单位，如用“px”表示像素。不允许使用百分比。可以使用负数来表示值，指定延伸到为元素指定的框之外的剪切区域。这四个值还可以使用关键字auto，从而指定剪切区域的边界与元素框的相应边界一致。例如，可以用下面的 style 性质来显示元素最左边的 100 个像素：

```
style="clip: rect(auto 100px auto auto);"
```

注意，各个值之间没有逗号，剪切区域的边界是从上边界开始按顺时针方向依次设置的。

18.2.6 CSS 定位示例

例 18-2 是使用 CSS 样式表和 CSS 定位性质的重要例子。在兼容 CSS 的浏览器中显示这个 HTML 文档时，它在浏览器窗口中创建“子窗口”的视觉效果。图 18-2 展示了例 18-2 中的代码创建的效果。虽然清单中没有 JavaScript 代码，但它示范了用 CSS 可以实现的效果和用 CSS 定位性质可以实现的效果。

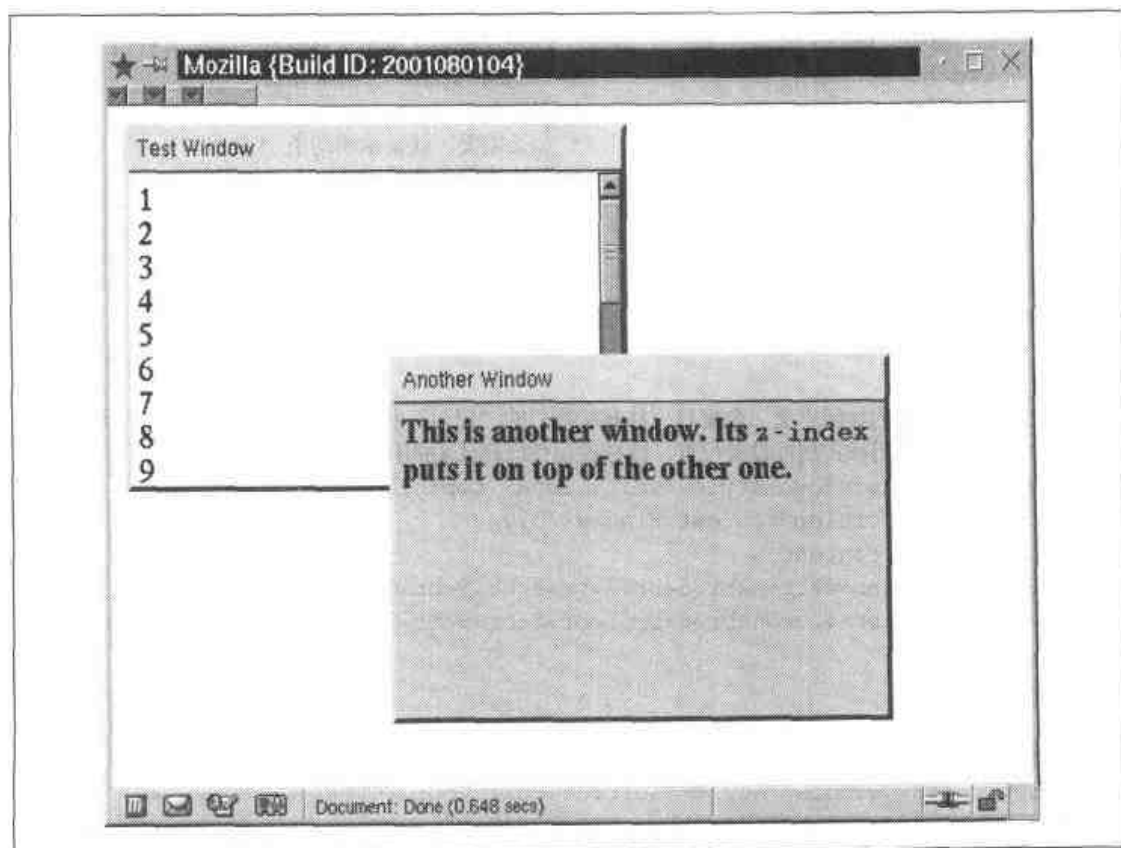


图 18-2：用 CSS 创建的窗口

例 18-2: 用 CSS 显示窗口

```

<head>
<style type="text/css">
/**
 * 这是一个 CSS 样式表, 定义了三个样式规则, 我们在文档主体中使用这些规则创建
 * “窗口”的视觉效果。这些规则使用定位性质设置窗口的整体大小并定位它的组件。
 * 改变窗口的大小要求改变三个规则中的定位性质。
 */
div.window { /* 设置窗口的大小和边线 */
    position: absolute; /* 此处设置窗口的位置 */
    width: 300px; height: 200px; /* 窗口的大小, 不包括边线 */
    border: outset gray 3px; /* 注意 3D "outset" 边线效果 */
}

div.titlebar { /* 设置标题栏的位置、大小和样式 */
    position: absolute; /* 它是一个定位元素 */
    top: 0px; height: 18px; /* Titlebar 是 18px + 补白和边线 */
    width: 290px; /* 290 + 5px 左右补白 = 300 */
    background-color: ActiveCaption; /* 采用系统标题栏的颜色 */
    border-bottom: groove black 2px; /* 标题栏只在底部有边线 */
    padding: 5px 0px 2px 5px; /* 顺时针值: 上、右、下、左 */
    font: caption; /* 采用系统标题栏的字体 */
}

div.content { /* 设置窗口内容的大小、位置和滚动 */
    position: absolute; /* 它是一个定位元素 */
    top: 25px; /* 18px 标题 + 2px 边线 + 3px + 2px 补白 */
    height: 165px; /* 200px 总值 - 25px 标题栏 - 10px 补白 */
    width: 290px; /* 300px 宽度 - 10px 补白 */
    padding: 5px; /* 四边有空白 */
    overflow: auto; /* 如果需要, 就显示滚动条 */
    background-color: #ffffff; /* 默认为白色背景 */
}

</style>
</head>

<body>
<!-- 以下是我们如何定义一个窗口, 具有标题栏的“窗口”和嵌套在它们之间的内容 div。 -->
<!-- 注意如何用样式属性设置位置, 该样式属性以样式表的样式作为参数。 -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Test Window</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- 许多线 -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- 演示滚动 -->
</div>
</div>

<!-- 以下是另一个窗口, 具有不同的位置、颜色和字体大小 -->
<div class="window" style="left: 150px; top: 140px; z-index: 20;">
<div class="titlebar">Another Window</div>

```

```
<div class="content" style="background-color:#dcdcdc; font-weight:bold;">
This is another window. It's <tt>z</tt> index, it's puts it on top of the other one.
</div>
</div>
</body>
```

这个例子的主要缺点是，样式表把所有窗口设置为固定大小。因为窗口的标题栏和内容部分都必须准确定位在整个窗口中，所以改变窗口大小要求改变样式表定义的两条规则中的所有定位性质的值。在静态HTML文档中很难做到这一点，但如果我们能用脚本设置所有必要的性质，这一点就不难实现。我们将在下一节中探讨这一主题。

18.3 脚本样式

DHTML的关键作用是能够用JavaScript动态地改变应用到文档中各个元素的样式性质。2级DOM标准定义了一个API，使这一作用非常容易实现。在第十七章中，我们用DOM API获取对文档元素的引用时，既可以采用标记名，也可以采用ID，还可以递归地遍历整个文档。一旦获取了想应用样式的元素的引用，就可以用元素的style属性获取那个元素的CSS2Properties对象。这个JavaScript对象有与CSS 1和CSS 2的每个样式性质对应的JavaScript属性。设置这些属性与设置元素的style性质中相应样式的效果一样。读取这些属性将返回元素的style性质设置的CSS性质值（如果存在）。需要知道的重要一点是，用元素的style属性获取的CSS2Properties对象只能设置元素的内联样式。不能用CSS2Properties对象的属性获取应用到元素的样式表的样式信息。设置这个对象的属性后，就能定义能有效地覆盖样式表样式的内联样式。

例如，考虑下面的脚本。它将找到文档中所有的元素，并遍历它们，根据它们的大小找到作为标题广告的元素。当它找到这样的元素时，可以用style.visibility性质把CSS visibility性质设为hidden，使该元素不可见：

```
var imgs = document.getElementsByTagName("img"); // 找到所有图像
for(var i = 0; i < imgs.length; i++) {           // 遍历它们
    var img=imgs[i];
    if (img.width == 468 && img.height == 60)      // 如果它是468 × 60的标题...
        img.style.visibility = "hidden";         // 隐藏它!
}
```


我把这个简单的脚本转换成 `javascript:URL` 后, 便把它转换成了一个书签, 存在于浏览器中。用书签把分散注意力的动画广告立刻隐藏起来是很有用的。下面是适合做书签的脚本:

```
javascript:a=document.getElementsByTagName( 'img' );for (n=0;n<a.length;n++){  
i=a[n];if (i.width==468&&i.height==60){i.style.visibility="hidden";}void 0;
```

这个书签是以非常简洁的代码编写的, 并且格式化在一行中。书签开头的 `javascript:` 标识了它是一个 URL, 它的主体是可执行的代码串。结尾处的 `void 0` 语句将使代码返回 `undefined`, 这意味着浏览器将继续显示当前 Web 页 (当然除了它的广告)。如果没有 `void 0` 语句, 浏览器将用最后执行的 JavaScript 语句返回的值覆盖当前 Web 页。

18.3.1 命名规范: JavaScript 中的 CSS 性质

许多 CSS 样式性质的名字中都有连字符, 如 `font-family`。在 JavaScript 中, 连字符被解释为减号, 所以不能编写下列表达式:

```
element.style.font-family = 'sans-serif';
```

因此, `CSS2Properties` 对象的属性名和真正的 CSS 性质名有点不同。如果一个 CSS 性质名含有一个或多个连字符, 那么 `CSS2Properties` 属性名删除了连字符, 且原来紧接在连字符后的字母改为大写。这样, 性质 `border-left-width` 可以通过 `borderLeftWidth` 属性访问, 可以用下列代码访问 `font-family` 性质:

```
element.style.fontFamily = "sans-serif";
```

CSS 性质和 JavaScript 的 `CSS2Properties` 属性间还有一点命名差别。`float` 是 Java 和其他语言中的关键字, 虽然当前 JavaScript 没有使用它, 但它被保留了以备将来使用。因此, `CSS2Properties` 对象没有与 CSS 的 `float` 性质对应的 `float` 属性。解决这个问题的办法是, 在 `float` 性质前加 `css` 前缀, 以便构成属性名 `cssFloat`。因此, 要设置或查询元素的 `float` 性质的值, 应该使用 `CSS2Properties` 对象的 `cssFloat` 属性。

18.3.2 使用样式属性

在使用 CSS2Properties 对象的样式属性时，要记住，所有值必须是字符串。在样式表或样式性质中，可以编写如下代码：

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

要用 JavaScript 对元素 e 实现同样的效果，必须用引号括起所有值：

```
e.style.position = 'absolute';  
e.style.fontFamily = "sans-serif";  
e.style.backgroundColor = "#ffffff";
```

注意，分号在字符串外。这些只是常规的 JavaScript 分号。而 CSS 样式表中使用的分号不是用 JavaScript 设置的字符串值的一部分。

另外，记住所有定位属性都要求有单位。因此，下面设置的 left 属性不正确：

```
e.style.left = 300;    // 错误：这是一个数字，不是字符串  
e.style.left = "300"; // 错误，漏写了单位
```

在 JavaScript 中设置样式属性时，单位是必需的。就像在样式表中设置样式性质一样。把元素 e 的 left 属性设置为 300 像素的正确方法是：

```
e.style.left = "300px";
```

如果想把 left 属性设置为计算值，要确保算式末尾有单位：

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

作为附加单位的一个额外作用，单位字符串将把计算值从数字转换成字符串。

还可以用 CSS2Properties 对象查询元素中 style 性质明确设置的 CSS 性质值，或读取前面由 JavaScript 代码设置的内联样式值。再提醒一遍，必须记住，这些属性的返回值是字符串，不是数字，所以下面的代码（它假定元素 e 的页边距由内联样式设置）不能实现你的要求：

```
var totalMarginWidth = e.style.marginLeft + e.style.marginRight;
```

应该改用下面的代码：

```
var totalMarginWidth = parseInt(e.style.marginLeft) + parseInt(e.style.marginRight);
```

这个表达式舍弃了两个字符串末尾返回的单位。它假定 `marginLeft` 属性和 `marginRight` 属性都用同样的单位。如果你在内联样式中只用像素做单位, 可以用这样的方式舍弃单位。

有些 CSS 性质 (如 `margin`) 是其他属性 (如 `margin-top`、`margin-right`、`margin-bottom` 和 `margin-left`) 的简化方式。CSS2Properties 对象也有相应于这些简化性质的属性。例如, 可以按如下方法设置 `margin` 属性:

```
e.style.margin = topMargin + 'px ' + rightMargin + 'px ' +  
                bottomMargin + 'px ' + leftMargin + 'px';
```

单独设置四个页边距属性更容易一些:

```
e.style.marginTop = topMargin + 'px';  
e.style.marginRight = rightMargin + 'px';  
e.style.marginBottom = bottomMargin + 'px';  
e.style.marginLeft = leftMargin + 'px';
```

也可以查询快捷属性的值, 但不值得这样做, 因为通常必需解析返回值, 把它分割成组成成分。通常这很难实现, 单独查询成分属性要简单得多。

最后, 再次强调, 在从 `HTMLElement` 的 `style` 属性获取了一个 `CSS2Properties` 对象时, 这个对象的属性表示该元素的内联样式性质。换句话说, 设置这些属性就像在元素的 `style` 性质中设置了 CSS 性质一样, 它只影响一个元素, 在 CSS 级联中比其他来源的冲突样式设置优先级高。这种对单个元素的精确控制正是我们用 JavaScript 创建 DHTML 想要的效果。

但在读这些 `CSS2Properties` 属性值时, 只有以前用 JavaScript 代码设置它们, 或在使用的 HTML 元素具有设置了它们需要的 `style` 内联性质时, 它们返回的值才有意义。例如, 文档中的一个样式表把所有段落的左页边距设置为 30 像素, 但如果读取段落元素的 `leftMargin` 属性, 那么除非那个段落具有 `style` 性质, 覆盖了样式表设置, 否则将得到空串。因此, 虽然 `CSS2Properties` 对象对于设置覆盖其他样式的样式非常有用, 但它没有提供查询 CSS 级联的方法, 也没有提供判断应用到给定元素的完整样式集合的方法。本章后面的小节将简要介绍 `getComputedStyle()` 方法, 它提供了这种功能。

18.3.3 例子：动态条形统计图

在给 HTML 文档添加图形和图表时，通常将它们实现为静态的内联图像。但由于 CSS 布局模型严重依赖矩形框，所以可以用 JavaScript、HTML 和 CSS 动态地创建条形统计图。例 18-3 展示了如何实现这一图表。这个例子定义了函数 `makeBarChart()`，这样在 HTML 文档中插入条形统计表就变得很简单。

例 18-3 的代码使用了第十七章中创建新的 `<div>` 元素，然后把它添加到文档中的方法和本章讨论的设置元素样式属性的方法。其中不涉及文本和其他内容，条形统计图不过是一组仔细调整过大小并定位在另一个矩形中的矩形。CSS 性质 `border` 和 `background-color` 用于使矩形可见。

这个例子中有些简单的数学计算，根据制图数据计算每个条形的高度（以像素为单位）。设置图表和条形的位置与大小的 JavaScript 代码还包括一些边线和补白的简单数学运算。用这个例子中的方法，可以修改例 18-2，添加一个动态创建指定大小的窗口的 JavaScript 函数。

图 18-3 展示了用 `makeBarChart()` 函数创建的条形图表：

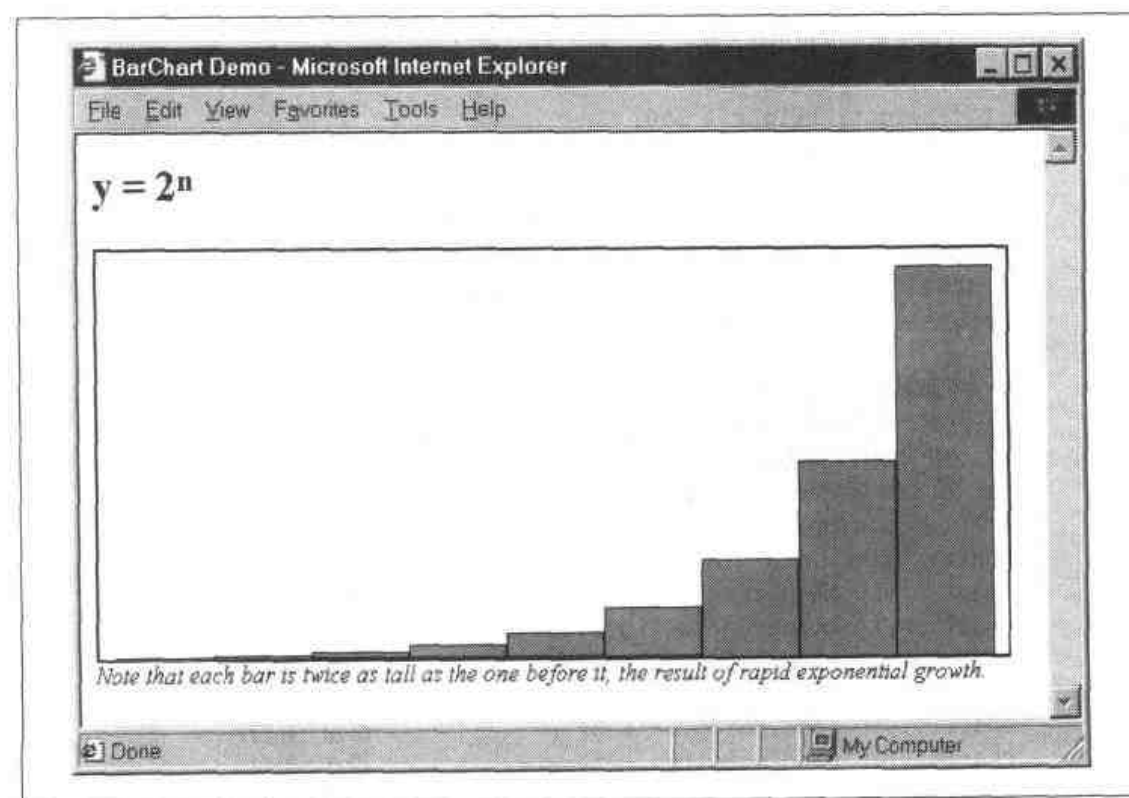


图 18-3：动态创建的条形统计图

```

<html>
<head>
<title>BarChart Demo</title>
<script src="BarChart.js"></script>
</head>
<body>
<h1>y = 2<sup>n</sup></h1>
<script>makeBarChart([2,4,8,16,32,64,128,256,512], 600, 250, 'red');</script>
<i>Note that each bar is twice as tall as the one before it,
the result of rapid exponential growth.</i>
</body>
</html>

```

例 18-3: 动态创建条形统计图

```

/**
 * BarChart.js:
 * 该文件定义了 makeBarChart() 函数，它将创建条形图来显示 data[] 数组中的数字。
 * 该图是插在当前文档结尾的块元素。
 * 图的整体大小由可选的 width 参数和 height 参数设置，包括边线和内部补白需要的空白。
 * 可选的颜色参数设置了条形的颜色。该函数将返回它创建的图元素，
 * 以便调用者可以通过设置边缘大小进一步操作它。
 *
 * 用如下代码把该函数输入到一个 HTML 文件中:
 *   <script src='BarChart.js'></script>
 * 用如下代码在 HTML 文件中使用函数:
 *   <script>makeBarChart([1,4,9,16,25], 300, 150, "yellow");</script>
 */
function makeBarChart(data, width, height, barcolor) {
    // 为可选的参数提供默认值。
    if (!width) width = 500;
    if (!height) height = 350;
    if (!barcolor) barcolor = "blue";

    // 参数 width 和 height 指定了生成的图表的整体大小。
    // 我们必须减去边线和补白的大小以得到我们创建的元素的大小。
    width -= 24; // 减 10px 补白和 2px 左右边线
    height -= 14; // 减 10px 上补白和 2px 上下边线

    // 下面创建存放图表的元素。
    // 注意，我们使图表相对定位，以便使它有绝对定位的子元素，
    // 但仍能出现在常规元素流中。
    var chart = document.createElement("DIV");
    chart.style.position = "relative"; // 设置相对定位。
    chart.style.width = width + "px"; // 设置图表宽度。
    chart.style.height = height + "px"; // 设置图表高度。
    chart.style.border = "solid black 2px"; // 设置边线。
    chart.style.paddingLeft = "10px"; // 添加左补白。
    chart.style.paddingRight = "10px"; // 添加右补白。
    chart.style.paddingTop = "10px"; // 添加上补白。
    chart.style.paddingBottom = "0px"; // 没有下补白。
    chart.style.backgroundColor = 'white'; // 使图表背景为白色。

```

```

// 计算每个栏的宽度。
var barwidth = Math.floor(width/data.length);
// 找到data[]中的最大数字。注意灵活使用Function.apply()。
var maxdata = Math.max.apply(this, data);
// 图表的缩放因子。scale*data[i]设置了条形的高度。
var scale = height/maxdata;
// 下面遍历data数组，为每个数据创建条形。
for(var i = 0; i < data.length; i++) {
    var bar = document.createElement("div"); // 为条形创建div。
    var barheight = data[i] * scale;          // 计算条形的高度。
    bar.style.position = "absolute";          // 设置条形的位置和大小。
    bar.style.left = (barwidth*i+1+10)+"px"; // 添加条形边线和图表补白。
    bar.style.top = height-barheight+10+"px"; // 添加图表补白。
    bar.style.width = (barwidth-2) + "px";    // 为条形边线-2。
    bar.style.height = (barheight-1) + "px";  // 为条形上边线-1。
    bar.style.border = "solid black 1px";      // 条形的边线样式。
    bar.style.backgroundColor = barcolor;      // 条形的颜色。
    bar.style.fontSize = "1px";               // 绕过IE bug。
    chart.appendChild(bar);                   // 把条形添加到图表。
}

// 下面把我们创建的图表添加到文档主体。
document.body.appendChild(chart);

// 最后返回图表元素，以便调用者可以操作它。
return chart;
}

```

18.3.4 DHTML 动画

用 JavaScript 和 CSS 能够实现的最强大的 DHTML 技术是动画。DHTML 动画没有什么特殊之处，所需要做的只是周期性地改变元素的一个或多个样式属性。例如，要使一幅图像从左边滑动到指定位置，需要反复增长图像的 `style.left` 属性，直到它达到了想要的位置为止。也可以反复地修改 `style.clip` 属性，逐像素地显示图像。

例 18-4 包含一个简单的 HTML 文件，它定义了一个要动画显示的 `div` 元素和一个简短的脚本，每 500 毫秒改变一次元素的背景颜色。注意，颜色改变是通过给 CSS 样式属性赋值实现的。由于颜色的反复改变使显示结果出现了动画效果，这是调用 Window 对象的 `setInterval()` 函数实现的（所有 DHTML 动画都要使用 `setInterval()` 或 `setTimeout()` 函数，阅读客户端参考手册部分有关参考页有助于你回忆起它们）。最后要注意，用取模（取余）运算符“%”可以通过循环变换颜色。如果你忘记了该运算符，可以查阅第五章。

例 18-4: 简单的颜色变化动画

```
<!-- 这个div是我们用来产生动画效果的元素-->
<div id="urgent"><h1>Red Alert!</h1>The Web server is under attack!</div>
<!-- 下面是元素的动画脚本 -->
<script>
var e = document.getElementById("urgent");           // 获取 Element 对象。
var colors = ["white", "yellow", "orange", "red"]     // 通过循环中的位置。
var nextColor = 0;                                   // 循环显示的颜色。
// 每隔 500 毫秒计算下列表达式, 使 div 元素的背景颜色产生变动
setInterval("e.style.backgroundColor=colors[nextColor++%colors.length];", 500);
</script>
```

例 18-4 生成了非常简单的动画。在实践中, CSS 动画通常同时涉及两个或多个样式属性 (如 `top`、`left` 和 `clip`) 的修改。用例 18-4 中所示的方法设置一个复杂的动画能得到相当复杂的效果。另外, 为了避免用户厌烦, 动画通常应该运行一会儿就停止, 但没有办法停止例 18-4 生成的动画。

例 18-5 展示的 JavaScript 文件定义了一个 CSS 动画函数, 使创建动画 (甚至设置复杂的动画) 变得容易得多。这个例子中定义的 `animateCSS()` 函数有 5 个参数。第一个参数指定了要用动画显示的 `HTMLElement` 对象。第二个参数和第三个参数指定了动画中的帧数和每帧显示的时间。第四个参数是一个 JavaScript 对象, 它指定要执行的动画。第五个参数是一个可选的函数, 应该在动画完成时调用。

`animateCSS()` 函数的第四个参数是它的关键参数。这个 JavaScript 对象的每个属性都必须与一个 CSS 样式属性同名, 每个属性的值必须是一个函数, 该函数返回指定样式的合法值。每显示一个新的动画帧, 这些函数就被调用一次, 以便为每个样式属性生成一个新值。当传递给这些函数的是帧号和消耗的全部时间时, 这些参数有助于返回正确的值。

用例子可以更好地理解 `animateCSS()` 的用法。下面的代码将把一个元素向上移动, 通过扩大它的剪切区使它逐渐消失:

```
// 使 id 为 'title' 的元素动起来, 每 50 毫秒显示 40 帧,
animateCSS(document.getElementById("title"), 40, 50,
{ // 设置每个帧的 top 和 clip 样式属性:
  top: function(f,t) { return 300-f*5 + 'px'; },
  clip: function(f,t) {return "rect(auto "+f*10+"px auto auto)";},
});
```

下面的代码段用 `animateCSS()` 函数在一个圆中移动 `Button` 对象。它使用

animateCSS()的第五个参数,在动画完成时,把按钮文本改为“Done”。注意,产生动画的元素将被作为参数传递给第五个参数指定的函数:

```
// 在圆中移动一个按钮,然后改变它显示的文本。
animateCSS(document.forms[0].elements[0], 40, 50, // Button, 40帧, 50ms
{ // 下面的三角函数在(200,200)处定义了一个半径为100的圆:
  left: function(f,t){ return 200 + 100*Math.cos(f/8) + "px" },
  top: function(f,t){ return 200 + 100*Math.sin(f/8) + "px" }
},
function(button) { button.value = "Done"; });
```

例18-5中的代码相当简单,我们很快就会看到,所有真正复杂的代码都嵌入到传递给animateCSS()的动画对象的属性中。animateCSS()定义了一个嵌套函数displayNextFrame(),它只是用setInterval()方法安排对displayNextFrame()函数的反复调用。displayNextFrame()函数将遍历动画对象的属性,调用各种函数来计算样式属性的新值。

注意,因为displayNextFrame()函数是在animateCSS()函数中定义的,所以它能够访问animateCSS()函数的参数和局部变量,即使是在animateCSS()返回后才调用displayNextFrame()函数。即使多次调用animateCSS()函数,同时使多个元素产生动画,也适用该规则(如果你不明白为什么使用这一规则,可以复习第11.4节)。

例 18-5: 基于 CSS 动画的框架

```
/**
 * AnimateCSS.js:
 * 该文件定义了名为animateCSS()的函数,它作为创建基于CSS动画的框架。
 * 该函数的参数如下:
 *
 *   element: 要产生动画的HTML元素。
 *   numFrames: 动画中的总帧数。
 *   timePerFrame: 显示每帧的毫秒数。
 *   animation: 定义动画的对象,后面有它的说明。
 *   whendone: 一个可选的函数,在动画结束时调用。
 *             如果设置了该函数,它的参数是element。
 *
 * animateCSS()函数只是定义了一个动画框架。
 * 它是指定要生成动画的动画对象的属性。
 * 每个属性都应该有同名的CSS样式属性。
 * 每个属性的值必须是返回相应的样式属性值的函数。
 * 每个函数的参数是帧号和消耗的总时间量,它可以使用这些参数计算应该为该帧返回的样式值。
 * 例如,要使一个图像生成动画,使它从左上角滑入,需要调用如下animateCSS:
 *
 * animateCSS(image, 25, 50, // 动画图像,每50毫秒显示25帧。
```



```

*           { // 设置每帧的 top 和 left 性质:
*               top: function(frame,time) { 返回帧 × 8 + 'px'; },
*               left: function(frame,time) { 返回帧 × 8 + 'px'; }
*           });
*
**/
function animateCSS(element, numFrames, timePerFrame, animation, whendone) {
    var frame = 0; // 存储当前帧号。
    var time = 0; // 存储消耗的总时间。

    // 安排每隔 timePerFrame 毫秒调用一次 displayNextFrame() 函数。
    // 这将显示动画的每个帧。
    var intervalId = setInterval(displayNextFrame, timePerFrame);

    // 返回对 animateCSS() 的调用,
    // 但上面一行代码确保了下面定义的嵌套函数会为每个帧调用一次。
    // 由于该函数在 animateCSS() 内定义,
    // 所以它能够访问 animateCSS() 的参数和局部变量,
    // 即使它是在函数返回之后被调用的。
    function displayNextFrame() {
        if (frame >= numFrames) { // 首先查看是否结束了。
            clearInterval(intervalId); // 如果结束了, 停止调用自身。
            if (whendone) whendone(element); // 调用 whendone 函数。
            return; // 结束。
        }
        // 下面遍历动画对象中定义的所有属性。
        for(var cssprop in animation) {
            // 对于每个属性, 调用它的动画函数, 传递给它帧号和消耗时间。
            // 用该函数的返回值作为指定元素的相应样式属性的新值。
            // 用 try/catch 语句忽略不成功返回引起的异常。
            try {
                element.style[cssprop] = animation[cssprop](frame, time);
            } catch(e) {}
        }
        frame++; // 增加帧号。
        time += timePerFrame; // 增加消耗时间。
    }
}

```

18.4 第四代浏览器中的 DHTML

Internet Explorer 4 和 Netscape 4 是把 DHTML 技术引入 Internet 的浏览器。这两种浏览器都部分支持 CSS 1 标准和 DHTML 的关键 CSS 定位性质 (整合在 CSS 2 标准中)。但是, 在开发第四代浏览器时, DOM 标准还不存在, 所以它们不遵守该标准。然而, 在这两种浏览器中都可以实现 DHTML 效果。

18.4.1 Internet Explorer 4 中的 DHTML

我们在第十七章看到, IE 4 不支持 `document.getElementById()` 方法, 也不支持动态创建新节点并把它们插入文档的 API。它提供了 `document.all[]` 数组存放文档中的所有元素, 允许用文档元素的 `innerHTML` 属性改变文档内容。虽然 IE 4 不遵守 DOM 标准, 但它提供了令人满意的替代方法。

尽管遍历和修改文档是 DHTML 的重要部分, 但本章的重点在于介绍 CSS 样式的动态用法。幸运的是, 前面介绍的通过 `style` 属性设置 CSS 样式性质的 DOM API 正是从 IE 4 API 中采纳的。因此, 如果已经用 `document.all[]` 找到了想要修改的文档元素, 就可以像使用一个完全支持 DOM API 的浏览器一样, 脚本化那个元素的样式 (记住, 由于 IE 4 不完全支持 CSS, 所以不能期望可以脚本化所有的样式属性)。

CSS 2 标准规定, 可以用 `position` 性质设置文档中元素的绝对位置和相对位置。但 IE 4 是在 CSS 2 之前实现的, 所以它只支持元素的某个子集的绝对定位技术。因此, 当你在 IE 4 中使用绝对定位方法时, 应该把需要定位或动画显示的元素包装在 `<div>` 或 `` 标记中, 它们支持 CSS `position` 性质。

18.4.2 Netscape 4 中的 DHTML

在 Netscape 4 中创建 DHTML 效果是件复杂的事。Netscape 4 不支持完整的对象模型, 所以它不允许 JavaScript 程序引用任意的 HTML 元素。因此, 它不允许访问元素的内联样式。但是它定义了特殊的 `Layer` 对象 (注 5)。绝对定位的元素 (即 `position` 属性被设为 `absolute`) 都被放在独立于文档其他部分的层中。这个层可以被独立地定位、隐藏、显示、放在其他层之下或移到其他层之上, 等等。尽管 `Layer` API 被提交给 W3C 进行标准化, 但至今都没有实现。因此, Mozilla 项目舍弃了它, Mozilla 和 Netscape 6 都不支持它。所以, 本节介绍的方法只在 Netscape 4.x 系列的浏览器中 useful。

文档中每个独立定位的层都由一个 `Layer` 对象表示。毫无疑问, `Document` 对象的

注 5: 在第十七章中讨论 Netscape 4 和核心 DOM API 的兼容性时讨论过层。这一章我们将详细介绍层并讨论层如何提供替代的核心 DOM API 和访问 CSS 样式的 DOM API。

layers[] 数组存放的是文档中的 Layer 对象的完整集合 (Layer 对象在这个数组中存放的顺序就是它们在文档中出现的顺序)。此外, 可以通过名字访问名字中带有 name 性质或 id 性质的层。例如, 如果一个层的 id="L2", 则在 Netscape 中就可以用 document.L2 或 document.layers["L2"] 来引用它。虽然 Netscape 4 没有提供引用任意文档元素的方法, 但这个 layers[] 数组提供了引用最重要的动态元素的方法。

层有点像一个独立的窗口或框架。虽然 Layer 对象不同于 Window 对象, 但它也有 document 属性, 就像窗口和框架一样。Layer 对象的 document 属性引用一个 Document 对象, 即每个层都有自己完全独立的 HTML 文档。层还可以嵌套, 可以用下列代码把一些 HTML 文本输入一个嵌套层:

```
document.layers[1].document.layers[0].document.write("Layers Are Fun!");
document.layers[1].document.layers[0].document.close();
```

Netscape 4 不允许创建或操作文档树的节点, 甚至不支持 Internet Explorer 的 innerHTML 属性。但层含有独立文档这一事实提供了动态修改文档内容的方法。

尽管 Netscape 4 把层定义为具有 CSS position 样式集合的元素, 但它没有定义直接脚本化层元素的样式的方法。不过 Layer 对象定义了可以用来动态定位图层的属性和方法。

Layer 对象的属性名与重要的 CSS 样式性质相似, 但这些层属性不完全与样式属性相同。例如, Layer 对象的 left 属性和 top 属性指定了层的像素位置, 设置层的这些属性就像设置元素的样式属性 left 和 top, 只是 Layer 属性采用的是数字像素值, 而不是包括数值和单位的字符串。层的 visibility 属性指定了层内容是否可见, 它与同名的样式属性很像, 只是它的值为 show 或 hide, 而 CSS 标准中的值是 visible 或 hidden。Layer 对象也支持 zIndex 属性, 它与 zIndex 样式属性作用相同。表 18-3 列出了关键的 CSS 样式属性和与它们最接近的 Layer 属性。注意, 这些只是 Netscape 4 允许脚本化的样式属性。

表 18-3: Netscape 4 中的 Layer 属性

CSS 属性	等价的 Layer 属性	Layer 注释
left, top	left, top	指定像素数, 没有单位。参阅 moveTo() 和 moveBy()

表 18-3: Netscape 4 中的 Layer 属性 (续)

CSS 属性	等价的 Layer 属性	Layer 注释
zIndex	zIndex	参阅moveAbove()和moveBelow()
visibility	visibility	如果把这一属性设为标准的visible和hidden, Layer对象将返回 show 或 hide
clip	clip.bottom, clip.height, clip.left, clip.right, clip.top, clip.width	指定像素数, 没有单位
backgroundColor	bgColor	
backgroundImage	background.src	设置为 URL 字符串

从表 18-3 可以发现, Layer 对象支持两个与动态定位无关的有用属性。属性 background.src 为层设置背景图像, bgColor 为层设置背景颜色。这两个属性分别对应于样式属性 backgroundImage 和 backgroundColor。

除了属性, Layer 对象还提供了大量便捷的方法。moveBy() 和 moveTo() 可以把层移动一个相对的量或移动到一个绝对位置。moveAbove() 和 moveBelow() 可以设置层相对于其他层的 zIndex。参阅本书的客户端参考手册部分, 可以看到 Layer 属性和方法的完整列表。

因为每个层含有独立的文档, 所以可以用 Document 对象的 open() 方法、write() 方法和 close() 方法动态地更新层内容, 我们在第十四章中看到过这种技术。此外, 层的 src 属性指定了要显示的文档的 URL。设置这一属性, 可以迫使浏览器装载一个全新的文档, 并在层中显示。方法 load() 与之相似, 它装载一个新 URL 所指的文档, 同时改变层的宽度。因为层常常含有动态生成的内容, 所以在 src 属性和 load() 方法中使用 javascript:URL 非常有用。

我们知道, Netscape 4 会自动为 position 样式属性设为 absolute 的元素创建 Layer 对象。Netscape 4 的 API 还允许以其他不太标准的方式创建层。例如, Netscape 4 定义了 HTML 标记 <layer>, 允许用户直接在 HTML 中定义层。<layer> 标记是 Netscape 4 的专有扩展, HTML 4 标准中没有这个标记, Mozilla 和 Netscape 6 也不支持它。更重要的是, Netscape 4 支持 Layer() 构造函数, 允许在必要时动态地创建 Layer 对象。详情请详见本书的客户端参考手册部分。

18.4.3 例子：跨平台的 DHTML 动画

尽管 DOM API、IE 4 API 和 Netscape Layer API 之间存在差别，但仍然可以创建在遵守 DOM 的浏览器、先于 DOM 的 IE 版本和 Netscape 4 中都能运行的 DHTML 效果。例 18-6 展示了这样一个例子。其中的脚本将显示单词“Hello”，并使它从浏览器窗口中的一点沿直线移动到另一点。

注意这个例子中采用的兼容性方法，即在使用关键函数、数组和属性之前，先测试它们的存在性。如果 Document 对象具有名为 getElementById 的属性，就假定使用的是遵守 DOM 的浏览器，那个属性引用的是方法 getElementById()。同样，如果 Document 对象具有名为 all 的属性，就假定该程序运行在 Internet Explorer 中，可以采用 document.all[] 数组来定位要生成动画的元素。

例 18-6：跨浏览器的 DHTML 动画脚本

```
<!-- 这是一个将要生成动画的动态元素。我们把 h1 标记包装在 div 元素中， -->
<!-- 因为如果没有 div 或 span 容器，IE 4 不移动 h1。 -->
<div id="title" style="position:absolute"><h1>Hello</h1></div>

<!-- 下面是执行动画的 JavaScript 代码 -->
<script>
// 下面是设置动画参数的变量。
var id = "title";           // 要动画显示的元素的名称。
var numFrames = 30;        // 要显示多少帧。
var interval = 100;        // 每个帧显示多久。
var x0 = 100, y0 = 100;    // 元素的开始位置。
var x1 = 500, y1 = 500;    // 元素的结束位置。
var dx = (x1 - x0) / (numFrames - 1); // 水平移动每个帧的距离。
var dy = (y1 - y0) / (numFrames - 1); // 垂直移动每个帧的距离。
var frameNum = 0;          // 当前所在的帧。
var element = null;        // 要动画显示的元素。

// 首先，找到要动画显示的元素。如果浏览器支持遵守 DOM 的方法，则采用它。
// 否则采用浏览器专用的代码。
if (document.getElementById) {           // 如果是遵守 DOM 的浏览器。
    element = document.getElementById(id); // 则使用 DOM 方法。
}
else if (document.all) {                 // 否则，如果支持 IE API。
    element = document.all[id];          // 则使用 all[] 数组查找元素。
}
else if (document.layers) {              // 否则，如果支持 Netscape API。
    element = document.layers[id];       // 则用 layers[] 数组获取元素。
}

// 如果我们使用前面的一种方法找到了要生成动画的元素，
// 就通过在一定的时间间隔内调用 nextFrame() 函数来开始动画。
```

```
if (element) {
    var intervalId = setInterval('nextFrame()', interval);
}

// 该函数将被反复调用以显示动画的每个帧。
// 它将用 DOM API 设置 CSS 样式属性来移动元素。
// 如果浏览器不支持那种 API，则采用 Netscape Layer API。
function nextFrame() {
    if (element.style) {
        // 如果浏览器支持它，通过设置 CSS 样式属性来移动元素。
        // 注意包含单位字符串。
        element.style.left = x0 + dx*frameNum + "px";
        element.style.top = y0 + dy*frameNum + "px";
    }
    else {
        // 否则，假定该元素是层，通过设置它的属性移动它。
        // 我们还可以使用 element.moveTo() 方法。
        element.left = x0 + dx*frameNum;
        element.top = y0 + dy*frameNum;
    }

    // 增加帧号，如果到达了末尾，则停止。
    if (++frameNum >= numFrames) clearInterval(intervalId);
}
</script>
```

18.5 关于样式和样式表的其他 DOM API

迄今为止，我们讨论了使用 CSS 样式的简单 DOM API，即文档中的每个 `HTMLElement` 元素都有 `style` 属性，该属性表示那个元素的内联样式性质。`style` 属性引用的是一个 `CSS2Properties` 对象，它为 CSS2 标准定义的每个 CSS 样式性质都定义了一个 JavaScript 属性。

虽然我们广泛地使用 `CSS2Properties` 对象，但它只是 CSS 的 DOM API 的一部分（注 6）。这一节提供了其他处理 CSS 样式表的 DOM API 的概览。但要注意，在本书编写的过程中，许多 CSS API 都不能被当前（第六代）浏览器很好地支持。在使用这里介绍的 API 之前，应该进行仔细的测试。

注 6：事实上，`CSS2Properties` 对象是可选的。一个 DOM 实现可以支持 CSS，但不支持 `CSS2Properties`。但在实践中，这是使用 API 以用于样式的最常用的方法，我们一般都要求 Web 浏览器中的 DOM 实现支持它。

18.5.1 样式声明

CSS2Properties接口是CSSStyleDeclaration的子接口。因此,每个文档元素的style属性还实现了CSSStyleDeclaration的属性和方法。这些方法包括setProperty()和getPropertyValue(),它们是设置和查询CSS2Properties对象的某个样式属性的替代方法。例如,下面两行代码实现了这一目标:

```
element.style.fontFamily = 'sans-serif';  
element.style.setProperty("font-family", 'sans serif', '');
```

CSSStyleDeclaration接口的另一个特性是方法removeProperty(),它可以删除指定的样式。cssText属性将返回表示所有样式性质和它们值的文本表示。因为CSSStyleDeclaration对象表示样式性质和它们值的集合,所以可以将它用作数组,用样式性质名进行迭代。

18.5.2 计算样式

本章前面提到过,文档元素的style属性只表示那个元素的style性质,不包含影响元素的其他样式的信息(来自样式表)。要确定应用到元素的完整样式集合,需要使用Window对象的getComputedStyle()方法(该方法由AbstractView接口定义,详情可参阅DOM参考手册部分的“AbstractView.getComputedStyle()”条目)。这个方法的返回值是一个CSSStyleDeclaration对象,该对象描述应用到指定元素的所有样式。可以假定返回的对象还实现了CSSProperties接口,就像其他元素的style属性一样。

要说明元素的内联样式和它的计算样式之间的差别,可考虑元素e。要判断e是否具有内联style性质设置的字体,可以用下列代码:

```
var inlinefont = e.style.fontFamily;
```

但要判断显示e的字体族(无论内联样式或样式表是否指定过它),需要用下面的代码:

```
var fontfamily = window.getComputedStyle(e, null).fontFamily;
```

getComputedStyle()方法是由AbstractView接口定义的,下面的代码能清楚地说明这一点:

```
var fontFamily = document.defaultView.getComputedStyle(e, null).fontFamily;
```

getComputedStyle()方法返回的样式值是只读的,因为它们来自样式级联的各个地方。设置这些性质对元素样式没有任何影响。此外,还应该把getComputedStyle()方法看作“昂贵的”,因为它必须遍历整个级联,并创建一个大的CSSStyleDeclaration来表示应用到元素的大量样式性质。

最后要注意,除了所有HTML元素都有的style属性外,IE 5和其后的版本还定义了非标准但非常有用的currentStyle属性,它引用一个CSS 2 Properties对象,存放那个元素的计算样式。

18.5.3 覆盖样式

CSS标准规定,Web浏览器具有默认的样式表,该样式表定义了HTML的基本显示样式。浏览器允许用户指定用户样式表来表示用户的样式首选项,覆盖默认样式表中设置的样式。作者样式表是由文档的作者定义的样式表,即包含在文档中或文档可以链接的样式。作者样式表将覆盖浏览器的默认样式和用户样式(除了!important样式)。可以把元素的style属性设置的内联样式作为作者样式表的一部分。

DOM标准引入了覆盖样式表的概念,这种样式表能覆盖作者样式表,包括内联样式。在覆盖样式表中设置元素的样式,可以在不修改文档的样式表和元素的内联样式的情况下,改变元素的显示样式。可以用Document对象的getOverrideStyle()方法获取元素的覆盖样式:

```
var element = document.getElementById("title");  
var override = document.getOverrideStyle(element, null);
```

该方法将返回一个CSSStyleDeclaration对象(它也实现了CSS 2 Properties对象),可以用来改变元素的显示样式。注意设置覆盖样式和内联样式之间的差别:

```
override.backgroundColor = "yellow"; // 设置覆盖样式  
element.style.backgroundColor = "pink"; // 设置内联样式
```


18.5.4 创建样式表

DOMImplementation 对象（用 `document.implementation` 访问）定义了创建 `CSSStyleSheet` 对象的 `createCSSStyleSheet()` 方法。`CSSStyleSheet` 对象定义了 `insertRule()` 方法，可以用来向样式表添加样式规则。遗憾的是，2 级 DOM 没有定义把样式表关联到文档的方法，所以当前没有办法使用这一方法。DOM 标准的将来的版本可能会修正这一点。

18.5.5 遍历样式表

DOM 核心 API 可以遍历 HTML（或 XML）档，检测文档的每个元素、性质和 `Text` 节点。同样，用 DOM 的样式表和 CSS 模块，可以检测文档中的所有样式和链接到文档的所有样式，也可以遍历样式表，检测构成样式表的规则、选择器和样式性质。

对于想创建 DHMLT 的脚本作者，只用前面介绍过的 API 处理元素的内联样式通常就足够了，不必遍历样式表。不过本节仍然简要地介绍遍历样式表的 DOM API。在 DOM 参考手册部分，可以找到 API 的详细介绍。在本书编写的过程中，DOM API 还没有被很好的支持，不过 Mozilla 有望很快支持它。还要注意，IE 5 为遍历样式表定义了专有的、不兼容的 API。

通过 `document.styleSheets[]` 数组，可以访问包含到文档或链接到文档的样式表。例如：

```
var ss = document.styleSheets[0];
```

这个数组的元素是 `StyleSheet` 对象，它表示通用的样式表。在使用 CSS 样式表的 HTML 文档中，这些对象都实现了 `CSSStyleSheet` 子接口（提供了 CSS 特有的属性和方法）。`CSSStyleSheet` 对象具有 `cssRules[]` 数组，该数组存放样式表的规则。例如：

```
var firstRule = document.styleSheets[0].cssRules[0]
```

`CSSStyleSheet` 接口还定义了 `insertRule()` 方法和 `deleteRule()` 方法，用于给样式表添加规则和从中删除规则：

```
document.styleSheets[0].insertRule('H1 { text-weight: bold; }', 0);
```

CSSStyleSheet.cssRules[]数组的元素是 CSSRule 对象。CSS 样式表可以包含大量不同类型的规则。除了我们在本章中看到的基本样式外,还有各种“附加规则”,这些规则由关键字 @import 和 @page 指定。在 CSS 参考手册部分可以找到这些特殊类型的 CSS 规则。

CSSRule 接口是通用的,可以表示任何类型的规则,它的子接口可以表示特定类型的规则。CSSRule 接口的 type 属性指定了特定的规则类型。CSS 样式表中的大多数规则都是基本样式规则,例如:

```
h1 { font-family: sans-serif; font-weight: bold; font-size: 24pt; }
```

这种规则的 type 属性值为 CSSRule.STYLE_RULE, 由额外实现了 CSSStyleRule 接口的 CSSRule 对象表示。CSSStyleRule 对象定义了 selectorText 属性,它含有规则选择器(在上面的规则中用字符串“h1”表示)和包含规则的样式属性与值的 style 属性(如上面规则中的 font 属性)。例如:

```
var rule = document.styleSheets[0].cssRules[0]
var styles;
if (rule.type == CSSRule.STYLE_RULE) styles = rule.style;
```

CSSStyleRule.style 属性的值是一个 CSSStyleDeclaration 对象。我们已经遇到过这个对象,它与用来表示文档元素的内联样式的对象是同一类型。它定义了 setProperty()、removeProperty() 和 getProperty Value() 这些方法。前面讨论过, CSSStyleDeclaration 对象通常都实现了 CSS2Properties 接口,因此它还定义了对应于每个 CSS 性质的属性。

CSS2Properties 对象的属性和 CSSStyleDeclaration 对象的 getProperty Value() 方法以字符串形式返回 CSS 样式性质的值。本章前面讨论过,这意味着在查询一个性质(如 font-size)的值时(或在读 CSS2Properties 对象的 fontSize 属性时),得到的是一个数字和一个单位,它可能是“24pt”或(可能不太有用)“10mm”。一般说来,在得到 CSS 性质的字符串值时,必须以某种方式解析它,从中提取出想要的。对于 clip 这样的性质来说尤其是这样,它的字符串语法很复杂。

作为解析字符串的替代方法, CSSStyleDeclaration 对象提供了另一种方法 getProperty CSSValue(), 它以 CSSValue 对象的形式返回 CSS 性质,而不是以字符串形式返回该性质。CSSValue 对象的 cssValueType 属性指定了该对象实现的一

个子接口。如果一个性质的值不止一个，那么 `CSSValue` 对象实现了 `CSSValueList` 接口并且其行为与 `CSSValue` 对象的数组相似。不过，`CSSValue` 通常是一个“原始”值，实现的是 `CSSPrimitiveValue` 接口。`CSSPrimitiveValue` 对象有一个名为 `primitiveType` 的属性，指定了值的类型或值的单位。有 26 种可能的类型，所有类型都由常量表示，如 `CSSPrimitiveValue.CSS_PERCENTAGE`、`CSSPrimitiveValue.CSS_PX` 和 `CSSPrimitiveValue.CSS_RGBCOLOR`。除了 `primitiveType` 属性和各种常量，`CSSPrimitiveValue` 对象还定义了各种设置和查询对象表示的值的方法。例如，如果 `CSSPrimitiveValue` 对象表示长度或百分比，那么调用 `getFloatValue()` 方法可以得到长度。如果 `primitiveType` 属性说明该值表示颜色，那么调用 `getRGBColorValue()` 方法可以查询颜色值。

最后，DOM CSS API 还定义了一些特殊对象类，用来表示性质值。`RGBColor` 对象表示颜色值，`Rect` 对象表示矩形值（如 `clip` 性质的值），`Counter` 对象表示 CSS2 计数器。详情参见 DOM 参考手册。

第十九章

事件和事件处理

我们在第十二章中看到过，具有交互性的 JavaScript 程序使用的是事件驱动的程序设计模型。在这种程序设计样式中，当文档或它的某些元素发生了某些事情时，Web 浏览器就会生成一个事件（event）。例如，在浏览器装载完一个文档，在用户把鼠标移到一个超链接上或者在用户点击了表单的 **Submit** 按钮时，浏览器都会生成事件。如果 JavaScript 应用程序注重特定文档元素的特定类型的事件，它就会为那个元素这种类型的事件注册一个事件处理程序（event handler），即一个 JavaScript 函数或代码段。然后，在发生特定事件时，浏览器将调用处理程序代码。所有具有图形用户界面的应用程序都是以这种方法设计的，即它们等待用户做某些事情（也就是等待事件发生），然后进行响应。

作为题外话，值得一提的是，计时器和错误处理函数（第十三章介绍过它们）都与事件驱动的程序设计模型有关。像本章介绍的事件处理程序一样，计时器和错误处理函数都向浏览器注册一个函数，允许浏览器在某个事件发生时调用那个函数。但在这两种情况中，事件是经过了指定的时间间隔或发生了 JavaScript 错误。虽然本章没有讨论计时器和错误处理函数，但将它们看做与事件处理相关非常有用，因此建议你在学习本章时重读第 13.4 和第 13.5 小节。

大多数重要的 JavaScript 程序在很大程度上基于事件处理程序。我们已经见到了大量使用简单事件处理程序的 JavaScript 程序。本章详细介绍了所有还未提到的事件和事件处理程序。遗憾的是，这些细节非常复杂，目前使用的有四种完全不同的、不兼容的事件处理模型。这些模型是：

原始事件模型。这是一种简单的事件处理模式，我们在本书中已经用过（但并未全面介绍过）。HTML 4标准已部分地把它编集到标准中，通常把它看作0级DOM API的一部分非正式内容。尽管它的特性有限，但所有启用JavaScript的浏览器都支持它，因此它具有可移植性。

标准事件模型。这是一种强大的、具有完整特性的事件模型，2级DOM标准对它进行了标准化。Netscape 6 和 Mozilla 支持它。

Internet Explorer 事件模型。这一事件模型由 IE 4 及其后版本实现，具有标准事件模型的许多高级特性，但不具有全部特性。虽然 Microsoft 公司参与了2级DOM事件模型的创建，而且有足够的时间在IE 5.5和IE 6中实现这一标准事件模型，但它们仍然坚持使用自己专有的事件模型。这意味着，想使用高级事件处理特性的JavaScript程序设计者必须为IE浏览器编写特定的代码。

Netscape 4 事件模型。这个事件模型由 Netscape 4 实现。尽管标准事件模型已经取代了它，但 Netscape 6 继续支持它（支持大部分，但不是全部）。它具有标准事件模型的某些高级特性，但不是全部特性。想使用高级事件处理特性、又想与 Netscape 4 兼容的 JavaScript 程序设计者必须理解这种模型。

本章余下的部分依次介绍了每种事件模型。

19.1 基本事件处理

迄今为止，在本书的代码中，事件处理程序都被写作JavaScript代码串，它用做某种HTML性质（如onclick）的值。虽然这是原始事件模型的关键内容，但你还应该了解更多细节，接下来的小节将对它们进行介绍。

19.1.1 事件和事件类型

发生的事情的类型不同，生成的事件类型也不同。用户把鼠标移到超链接上时引发的事件与用户点击表单的**Submit**按钮时引发的事件不同。即使发生同样的事情，但由于环境不同也可能生成不同类型的事件。例如，用户点击**Submit**按钮和点击**Reset**按钮生成的事件类型就不同。

在原始事件模型中，事件是浏览器内部提取的，JavaScript代码不能直接操作事件。

我们在原始事件模型中提到过事件类型,其真实含义是响应事件时调用的事件处理程序名。在这种模型中,用HTML元素的性质(和相关的JavaScript对象的相应性质)设置事件处理代码。因此,如果应用程序需要知道用户何时把鼠标移动到超链接上,就要使用定义超链接的标记的onmouseover性质。如果应用程序需要知道用户何时点击了Submit按钮,就要使用定义该按钮的标记的onclick性质,或使用包含该按钮的<form>元素的onsubmit性质。

在原始事件模型中,有大量不同的事件处理程序性质可供使用。表19-1列出了这些性质,还列出了何时触发这些事件处理程序,以及哪些HTML元素支持这些事件处理程序性质。

随着客户端JavaScript程序设计方法的发展,它支持的事件模型也随之发展。新的浏览器版本添加了新的事件处理程序性质。HTML 4标准编纂了HTML标记的事件处理程序性质的标准集合。表19-1的第二列指出了哪些HTML元素支持那个事件处理程序性质,还指出了哪些浏览器版本支持标记使用的该事件处理程序性质,以及该事件处理程序是否是标记使用的HTML 4的部分标准。在第三列中,“N”是Netscape的缩写,“IE”是Internet Explorer的缩写。每个浏览器版本都向后兼容,例如,“N3”表示Netscape 3和其后的所有版本。

如果仔细地研究表19-1中的各种事件处理程序性质,可以将其分为两大类事件。一类是“原始事件”(raw event)或“输入事件”(input event)。这些事件是在用户移动鼠标、点击鼠标或敲击键盘键时生成的。这些低级事件只描述用户的动作,没有其他含义。第二类事件是“语义事件”(semantic event)。这些高级事件含义较复杂,通常只在特定环境中发生,如在浏览器装载完文档或要提交表单时。语义事件通常作为低级事件的附加作用发生。例如,当用户点击了Submit按钮时,会触发三个输入处理程序,即onmousedown、onmouseup和onclick。作为鼠标点击的结果,包含该按钮的HTML表单将生成onsubmit事件。

关于表19-1,最后需要注意一点。对于原始鼠标事件处理程序,表中的三列列出了大多数元素支持的事件处理程序性质(至少在HTML 4中)。不支持这些事件处理程序的HTML元素通常是属于文档<head>部分的元素或自身没有图形表示的元素。不支持通用的鼠标事件处理程序属性的标记包括<applet>、<bdo>、
、、<frame>、<frameset>、<head>、<html>、<iframe>、<isindex>、<meta>和<style>。

表 19-1: 事件处理程序和支持它们的 HTML 元素

处理程序	触发时机	由 ... 支持
onabort	图像装载被中断	N3、IE4:
onblur	元素失去输入焦点	HTML4、N2、IE3: <button>、<input>、 <label>、<select>、<textarea> N3、IE4:<body>
onchange	选中 <select> 元素中的选项或其他表单元素失去了焦点，并且由于它获得了焦点而使值发生了改变	HTML4、N2、IE3:<input>、<select>、 <textarea>
onclick	鼠标按下并释放，发生在 mouseup 事件后。返回 false 可以取消默认动作（如进行链接、重置、提交）	N2、IE3:<a>、<area>、<input> HTML4、N6、IE4: 大多数元素
ondblclick	双击鼠标	HTML4、N6、IE4: 大多数元素
onerror	在装载图像的过程中发生了错误	N3、IE4:
onfocus	元素得到输入焦点	HTML4、N2、IE3:<button>、<input>、 <label>、<select>、<textarea> N3、IE4:<body>
onkeydown	键盘键被按下。返回 false 取消默认动作	N4:<input>、<textarea> HTML4、N6、IE4: 表单元素和 <body>
onkeypress	键盘键被释放，返回 false 取消默认动作	N4:<input>、<textarea> HTML4、N6、IE4: 表单元素和 <body>
onkeyup	键盘键被释放	N4:<input>、<textarea> HTML4、N6、IE4: 表单元素和 <body>
onload	文档装载完毕	HTML 4、N2、IE3:<body>、<frameset> N3、IE4: N6、IE4:<iframe>、<object>
onmousedown	鼠标键被按下	N4:<a>、<area>、 HTML4、N6、IE4: 大多数元素
onmousemove	鼠标移动	HTML4、N6、IE4: 大多数元素

表 19-1: 事件处理程序和支持它们的 HTML 元素 (续)

处理程序	触发时机	由 ... 支持
onmouseout	鼠标离开了元素	N3:<a>、<area> HTML4、N6、IE4:大多数元素
onmouseover	鼠标移到元素上。对于链接元素, 返回 true 可以防止 URL 出现在状态栏中	N2、IE3:<a>、<area> HTML4、N6、IE4:大多数元素
onmouseup	释放鼠标键	N4:<a>、<area>、 HTML4、N6、IE4:大多数元素
onreset	表单请求重置。返回 false 阻止重置	HTML4、N3、IE4:<form>
onresize	调整窗口大小	N4、IE4:<body>、<frameset>
onselect	选中文本	HTML4、N6、IE3:<input>、<textarea>
onsubmit	请求提交表单。返回 false 阻止提交	HTML4、N3、IE4:<form>
onunload	卸载文档或框架集	HTML4、N2、IE3:<body>、<frameset>

19.1.2 作为性质的事件处理程序

从本章之前的许多例子中我们已经看到, 事件处理程序被设置为 JavaScript 代码串 (在原始事件模型中), 作为 HTML 的性质值。例如, 要在用户点击一个按钮时执行 JavaScript 代码, 可以把这段代码设置为 <input> 标记的 onclick 性质的值:

```
<input type="button" value="Press Me" onclick="alert('thanks ');">
```

事件处理程序性质的值是一个任意的 JavaScript 代码串。如果处理程序由多个 JavaScript 语句构成, 语句之间必须用分号分隔。例如:

```
<input type="button" value="Click Here"
  onclick="if (window.numclicks) numclicks++; else numclicks=1;
  this.value= Click #    + numclicks;">
```

在事件处理程序要求多个语句时, 可以把这些语句定义在一个函数体中, 然后用 HTML 事件处理程序性质调用那个函数, 这样处理起来会更加容易。例如, 如果你想在提交表单之前验证用户的表单输入, 可以用 <form> 标记的 onsubmit 性质。表

单验证通常要求多行代码，所以定义一个表单验证函数，用 `onclick` 性质调用那个函数，比把所有代码填充在一个长长的属性值中清楚得多。例如，如果定义了一个名为 `validateForm()` 的函数来执行验证，可以采用如下方式从一个事件处理程序中调用它：

```
<form action='processform.cgi' onsubmit="return validateForm();">
```

记住，HTML 不区分大小写，所以可以用你选择的方式确定事件处理程序性质的大小写。通常采用大小写混合的形式，前缀为小写的“on”，如 `onClick`、`onLoad`、`onMouseOut`，等等。但在这本书中，一般用全小写的形式来兼容区分大小写的 XHTML。

事件处理程序性质中的 JavaScript 代码可能含有 `return` 语句，返回值对浏览器具有特殊意义。不久我们就会讨论这个问题。另外要注意，事件处理程序中的 JavaScript 代码运行在不同于全局 JavaScript 代码的作用域中（参见第四章）。此后的小节也会对这一点详细讨论。

19.1.3 作为 JavaScript 属性的事件处理程序

我们知道，文档中的每个 HTML 元素在文档树中都有一个相应的 JavaScript 对象，这个 JavaScript 对象的属性对应于那个 HTML 元素的性质。在 JavaScript 1.1 和其后的版本中，这同样适用于事件处理程序性质。所以，如果一个 `<input>` 标记具有 `onclick` 性质，那么用该表单元素对象的 `onclick` 属性就可以引用它包含的事件处理程序（JavaScript 区分大小写，所以无论 HTML 性质采用什么大小写形式，JavaScript 属性必须是全小写的）。

技术上说来，DOM 标准不支持这里介绍的原始事件模型，没有定义对应于 HTML 4 标准化的事件处理程序性质的 JavaScript 性质。尽管 DOM 没有正规的标准，但这种事件模型仍然得到了广泛应用，因此所有启用 JavaScript 的 Web 浏览器都允许引用事件处理程序来作为 JavaScript 属性。

由于 HTML 事件处理程序性质的值是一个 JavaScript 代码串，因此可能也期望相应的 JavaScript 属性的值也是字符串。但实际情况并非如此，当通过 JavaScript 进行访问时，事件处理程序属性是函数。可以用一个简单的例子验证这一点：

```
<input type='button' value='Click Here' onclick='alert(typeof this.onclick);'/>
```

如果点击该按钮，它将显示含有“function”的对话框，而不是显示“string”（注意，在事件处理程序中，关键字this引用发生事件的对象。不久我们将讨论关键字this）。

要用JavaScript把一个事件处理程序赋予一个文档元素，只需把事件处理程序属性设置为想用的函数即可。例如，考虑下面的HTML表单：

```
<form name='f1'>
<input name='b1' type='button' value='Press Me'>
</form>
```

可以用document.f1.b1引用这个表单中的按钮，这意味着可以用如下的JavaScript代码给事件处理程序赋值：

```
document.f1.b1.onclick=function() { alert( 'Thanks!'); };
```

也可以用下面的代码给事件处理程序赋值：

```
function plead() { window.status = 'Please Press Me!'; }
document.f1.b1.onmouseover = plead;
```

注意最后一行代码，在函数名后没有括号。要定义一个事件处理程序，我们应该把函数自身赋予事件处理程序属性，而不是调用函数的结果。JavaScript的程序设计新手常常会在这里碰到许多问题。

把事件处理程序表示为JavaScript属性有两点好处。第一，它减少了HTML和JavaScript的混合，增强了代码的模块性，使代码更为简洁，也更容易维护。第二，它使事件处理函数进行动态处理。与HTML的性质不同的是，它是文档的一个静态部分，HTML性质只有在创建文档时才能对它进行设置，JavaScript的属性可以在任何时候改变。在复杂的交互程序中，动态地改变注册到HTML元素的事件处理程序有时也非常有用。在JavaScript中定义事件处理程序存在一个小缺点，它把处理程序和所属元素分开了。如果用户在文档装载完之前（以及在执行所有脚本前）与文档元素进行交互，文档元素的事件处理程序可能还没有定义。

例19-1展示了如何把一个函数设置为多个文档元素的事件处理程序。这个例子是一个简单的函数，它为文档中的每个链接定义了onclick事件处理程序。这个事件处理程序在浏览器进行超链接前请求用户的确认。如果用户不确认链接，事件处理程序函数将返回false，阻止浏览器进行链接。后面会讨论事件处理程序的返回值。

例19-1: 一个函数及多个事件处理程序

```
// 该函数适合用做 <a> 元素和 <area> 元素的 onclick 事件处理程序。  
// 它使用 this 关键字引用文档元素、返回 false 阻止浏览器进行链接。  
function confirmLink() {  
    return confirm( 'Do you really want to visit ' + this.href + '?' );  
}  
  
// 该函数将遍历文档中的所有超链接，并把 confirmLink 函数赋予它们，作为事件处理程序。  
// 在解析文档、定义所有链接之前，不要调用它。  
// 最好从 <body> 标记的 onload 事件处理程序调用它。  
function confirmAllLinks() {  
    for( var i = 0; i < document.links.length; i++ ) {  
        document.links[i].onclick = confirmLink;  
    }  
}
```

19.1.3.1 显式调用事件处理程序

由于 JavaScript 事件处理程序属性的值是函数，因此可以用 JavaScript 直接调用事件处理函数。例如，如果我们使用标记 <form> 的性质 onsubmit 定义了一个表单验证函数，并想在用户提交表单之前的某个时刻验证表单，那么可以使用 Form 对象的 onsubmit 属性来调用那个事件处理函数。代码如下：

```
document.myform.onsubmit();
```

但要注意，这种直接调用事件处理程序的方法不是模拟事件发生时的真正状况。例如，如果调用一个 Link 对象的 onclick 方法，它并不能使浏览器根据那个链接把新的文档装载进来，而只能执行定义为那个属性值的函数。要使浏览器装载新的文档，还必须用第十三章介绍的方法，设置 Window 对象的 location 属性。同样，调用 Form 对象的 onsubmit 方法或者调用 Submit 对象的 onclick 方法都只能运行事件处理函数，而不能完成表单的提交（要真正地提交表单还必须调用 Form 对象的 submit() 方法）。

显式调用事件处理函数的原因之一是，可以用 JavaScript 扩展 HTML 代码定义的事件处理函数。假定你想在用户点击按钮时执行特殊的动作，但又不想破坏 HTML 文档自身定义的 onclick 事件处理程序（这是例 19-1 中代码的一个问题，通过给每个超链接添加处理程序，便可以覆盖为这些超链接定义的所有 onclick 处理程序）。用下面的代码可以实现这一点：

```
var b = document.myform.mybutton; // 这是需要的按钮
```

```
var oldHandler = b.onclick; // 保存HTML事件处理程序
function newHandler() { /* 此处是事件处理程序代码 */ }
// 下面赋予一个新的事件处理程序，由它调用新旧处理程序
b.onclick = function() { oldHandler(); newHandler(); }
```

19.1.4 事件处理程序的返回值

在许多情况下，事件处理程序（无论是HTML性质设置的，还是JavaScript属性设置的）都使用它的返回值说明事件的处理方法。例如，如果使用Form对象的事件处理程序onsubmit执行表单验证，发现用户没有填写所有的域，那么可以让处理程序返回false来阻止真正提交表单。可用下列代码阻止提交域中为空文本的表单：

```
<form action="search.cgi"
      onsubmit="if (this.elements[0].value.length == 0) return false;">
<input type="text">
</form>
```

通常，如果浏览器执行某种默认动作来响应一个事件，那么可以返回false阻止浏览器执行那个动作。除了onsubmit，其他通过返回false阻止执行默认动作的事件处理程序包括onclick、onkeydown、onkeypress、onmousedown、onmouseup和onreset。表19-1的第二列中具有关于这些事件处理程序返回值的注释。

关于返回false的规则有一个例外，即当用户把鼠标移到一个超链接（或图像）上时，浏览器的默认动作是在状态栏中显示链接的URL。要阻止这种情况发生，必须让onmouseover事件处理程序返回true。例如，用下面的代码可以显示一条不是URL的消息：

```
<a href="help.htm" onmouseover="window.status='Help!!!'; return true;">Help</a>
```

这个异常的产生是没有任何原因的。

注意，事件处理程序从来不要求明确地返回值。如果不返回值，就会发生默认的动作。

19.1.5 事件处理程序和this关键字

无论是用HTML性质定义事件处理程序，还是用JavaScript属性定义事件处理程序，都是把一个函数赋予文档元素的一个属性。换句话说，你在定义文档元素的一个新

方法。在事件处理程序被调用时，它是作为产生事件的元素的方法调用的，所以关键字 `this` 引用了那个目标元素。这种行为很有用，而且也不会使人感到意外。

但要保证你理解了其中的含义。假定有一个对象 `o`，它具有方法 `myMethod`，可以用下列代码注册一个事件处理程序：

```
button.onclick = o.myMethod;
```

这个语句使 `button.onclick` 引用与 `o.myMethod` 相同的函数。现在，这个函数既是 `o` 的方法，也是 `button` 的方法。当浏览器触发这个事件处理程序时，它将把该函数作为 `button` 对象的方法调用，而不是作为 `o` 对象的方法调用。关键字 `this` 引用 `Button` 对象，不是引用对象 `o`。不要认为你可以让浏览器把一个事件处理程序作为其他对象的方法调用。如果你想这样做，必须直接调用它，如下所示：

```
button.onclick = function() { o.myMethod(); }
```

19.1.6 事件处理程序的作用域

我们在第十一章中讨论过，JavaScript 中的函数运行在词法作用域中。这意味着函数在定义它们的作用域中运行，而不在调用它们的作用域中运行。在把一个 HTML 性质的值设置为 JavaScript 代码串，以便定义事件处理程序时，隐式地定义了一个函数（在检测 JavaScript 中相应的事件处理程序属性的类型时，可以看到这一点）。以这种方式定义的事件处理函数的作用域与用常规方法定义的全局 JavaScript 函数不同。这意味着定义为 HTML 性质的事件处理程序可以在不同于其他函数的作用域中执行（注 1）。

回忆一下，我们在第四章中讨论过，函数的作用域由作用域链或对象列表定义，变量定义时要顺次检索这个链。如果要在一个常规函数中检索或解析变量 `x`，JavaScript 首先将检查该函数的调用对象的属性中是否具有名为 `x` 的属性，来检索局部变量和参数。如果没有找到同名的属性，JavaScript 就继续检索作用域链中的下一个对象，即全局对象。它将检查全局对象的属性来判断这个变量是否是全局变量。

注 1：理解这一点很重要，尽管接下来的讨论很有趣，但它还是很难理解。你可以在第一次阅读本章时跳过这部分内容，然后再返回来阅读它。

定义为HTML性质的事件处理程序具有更加复杂的作用域链。它们的作用域链的头是调用对象，传递给事件处理程序的所有参数都是在这里定义的（我们在本章后面的小节中会看到，在某些高级事件模型中，事件处理程序具有参数），它们就和事件处理程序主体中声明的局部变量一样。但是事件处理程序的作用域链中的下一个对象却并非全局对象，而是触发事件处理程序的对象。例如，假定使用标记在HTML表单中定义了一个Button对象，然后使用onclick性质定义了一个事件处理程序。如果该事件处理程序的代码使用了一个名为form的变量，那么该变量就会被解析为Button对象的form属性。在把事件处理程序编写为HTML性质时，这是一种有用的快捷方式。

事件处理程序的作用域链不是用定义处理程序的对象终止，它还会延伸到包容层。对于上述事件处理程序onclick，它的作用域链以处理函数的调用对象开始，然后像我们所讨论的那样延伸到Button对象，之后它将延伸到包容级别的HTML元素，至少包括包含按钮的<form>元素和包含表单的Document对象。作用域链的精确构成还没有标准化，这与实现有关。Netscape 6 和 Mozilla 包括所有包容对象（甚至是<div>这样的标记），而IE 6 则坚持使用最小的集合，只包括目标元素、Form对象（如果存在）和Document对象。无论用哪种浏览器，作用域链中的最终对象都是Window对象，就像它在客户端JavaScript中一贯是最终对象一样。

事件处理程序的作用域链中有目标对象非常方便。但包括其他文档元素的扩展作用域链却非常麻烦。例如，Window对象和Document对象都定义了名为open()的方法。如果没有限定条件，只用标识符open，那么引用的几乎总是window.open()方法。但在定义为HTML性质的事件处理程序中，Document对象在作用域链中先于Window对象，所以用open引用的是document.open()方法。同样，如果给一个Form对象添加一个名为window的属性（或用name='window'定义一个输入元素）会发生什么情况如？果在那个表单中定义一个使用表达式window.open()的事件处理程序，标识符将解析为Form对象的属性，而不是全局Window对象，表单中的事件处理程序就不能容易地引用Window对象或调用window.open()方法。

在定义作为HTML性质的事件处理程序时，一定要小心。最安全的方法是，使这种处理程序尽量简单。理想的方法是，让它们只调用在别的地方定义的全局函数，并可能返回下面的结果：

```
<script>
```

```
function validateForm() {  
    * 此处是表单验证代码 *  
    return true;  
}  
</script>  
<input type= "submit" onClick="return validateForm();">
```

这样一个简单的事件处理程序仍旧用一个不寻常的作用域链执行，在某个包容元素中定义 validateForm() 方法可以破坏它。但是，假定需要的目标函数最终被调用了，该函数将在常规的全局作用域中执行。再次提醒，函数是在定义它们的作用域链中执行，而不在调用它们的作用域中执行。所以，即使从不寻常的作用域中调用了 validateForm() 方法，仍旧可以在它自己的全局作用域中执行，而不会产生任何混淆。

另外，由于事件处理程序作用域链的构成没有任何标准，因此假定它只含有目标元素和全局 Window 对象最安全。例如，用 this 引用目标元素，那么当目标元素是 <input> 时，可以随意使用 form 引用包容的 Form 对象。但不要依靠作用域链中的 Form 对象和 Document 对象。例如，不要使用没有限制条件的标识符 write 引用 Document 对象的 write() 方法，而应该清楚地使用 document.write()。

记住，关于事件处理程序作用域的完整讨论只适用于定义为 HTML 性质的事件处理程序。如果把一个函数赋予适当的 JavaScript 事件处理程序属性来设置事件处理程序，那么根本不涉及特殊的作用域链，函数在定义它的作用域中执行，这几乎总是全局作用域，除非它是一个嵌套函数，在这种情况下，作用域链又变得有趣了。

19.2 2 级 DOM 中的高级事件处理

迄今为止，我们在本章中看到的事件处理方法都是 0 级 DOM（所有启用 JavaScript 的浏览器都支持的**实际标准 API**）的一部分。2 级 DOM 标准定义了高级事件处理 API，它与 0 级 API 有很大不同（而且强大得多）。虽然 2 级标准没有把现有的 API 集成到 DOM 标准中，但舍弃 0 级 API 不会有任何危险。对于基本的事件处理任务，应该继续使用简单 API。

Mozilla 和 Netscape 6 支持 2 级 DOM 的 Event 模型，但 Internet Explorer 6 不支持它。

19.2.1 事件传播

在 0 级 DOM 事件模型中，浏览器把事件分派给发生事件的文档元素。如果那个对象具有适合的事件处理程序，就运行这个处理程序。除此之外，不用执行其他的操作。在 2 级 DOM 中，情况要复杂得多。在这种高级事件模型中，当事件发生在 Document 节点（即事件目标）时，目标的事件处理程序就被触发。此外，目标的每个祖先节点也有机会处理那个事件。事件传播分三个阶段进行。第一，在捕捉（capturing）阶段，事件从 Document 对象沿着文档树向下传播给目标节点。如果目标的任何一个祖先（不是目标自身）专门注册了捕捉事件的处理程序，那么在事件传播的过程中，就会运行这些处理程序（不久我们将学习如何注册常规的事件处理程序和捕捉事件处理程序）。

事件传播的下一个阶段发生在目标节点自身，直接注册在目标上的适合的事件处理程序将运行。这与 0 级事件模型提供的事件处理方法相似。

事件传播的第三个阶段是起泡（bubbling）阶段，在这个阶段，事件将从目标元素向上传播回或起泡回 Document 对象的文档层次。虽然所有事件都受事件传播的捕捉阶段的支配，但并非所有类型的事件都起泡。例如，把提交事件从 <form> 元素向上传播到控制它的文档元素是毫无意义的。另一方面，文档中的所有元素都对普通事件（如 mousedown）感兴趣，所以它们可以沿着文档层次起泡，触发目标元素的祖先的适当的事件处理程序。一般说来，原始输入事件起泡，而高级语义事件不起泡（参阅本章后面的表 19-3，它完整地列出了哪些事件起泡，哪些不起泡）。

在事件传播过程中，任何事件处理程序都可以调用表示那个事件的 Event 对象的 stopPropagation() 方法，停止事件的进一步传播。在本章后面的小节中，我们将看到更多有关 Event 对象和 stopPropagation() 方法的内容。

有些事件会引发浏览器执行相关的默认动作。例如，在点击 <a> 标记时，浏览器的默认动作是进行超链接。这样的默认动作只在事件传播的三个阶段都完成之后才会执行，事件传播过程中调用的任何处理程序都能通过调用 Event 对象的 preventDefault() 方法阻止默认动作发生。

虽然这种事件传播看起来很复杂，但它提供了重要的中心化事件处理代码的方法。1 级 DOM 暴露所有的文档元素，允许事件（如 mouseover 事件）在任何元素上发生。

这意味着注册事件处理程序的地方比老式的0级事件模型多得多。假定想在用户把鼠标移到文档中的<p>元素上时触发一个事件处理程序,那么不必为每个<p>标记都注册一个 `onmouseover` 事件处理程序,只在 `Document` 对象上注册一个 `onmouseover`,然后在事件传播的捕捉或起泡阶段处理这些事件即可。

关于事件传播,有一个重要的细节。在0级模型中,只能为特定对象的特定类型的事件注册一个事件处理程序。但在2级模型中,就可以为特定对象的特定类型事件注册任意多个处理函数。这同样适用于事件目标的祖先,它们的处理函数将在事件传播的捕捉阶段或起泡阶段调用。

19.2.2 事件处理程序的注册

在0级API中,通过在HTML中设置性质或在JavaScript代码中设置对象的属性,来注册事件处理程序。在2级事件模型中,可以调用对象的 `addEventListener()` 方法为特定元素注册事件处理程序(DOM标准在它的API中使用术语“监听者”,但在我们的讨论中将继续使用同义词“处理程序”)。这个方法有三个参数。第一个参数是要注册处理程序的事件类型名。事件类型应该是含有小写HTML处理程序性质名的字符串,没有前缀“on”。因此,如果用HTML性质 `onmousedown`,或在0级模型中用 `onmousedown` 属性,那么在2级事件模型中就应该使用字符串“`mousedown`”。

第二个参数是处理函数(或监听者),在指定类型的事件发生时调用该函数。在调用这个函数时,传递给它的惟一参数是 `Event` 对象。这个对象存放了有关事件(如鼠标按钮被按下)的细节,并定义了 `stopPropagation()` 这样的方法。此后我们将学习更多有关 `Event` 接口和它的子接口的内容。

`addEventListener()` 的最后一个参数是一个布尔值。如果值为 `true`,则指定的事件处理程序将在事件传播的捕捉阶段用于捕捉事件。如果该参数值为 `false`,则事件处理程序就是常规的,当事件直接发生在对象上,或发生在元素的子女上,又向上起泡到该元素时,该处理程序将被触发。

例如,可以用下列 `addEventListener()` 方法,为<form>元素的提交事件注册一个处理程序:

```
document.myform.addEventListener("submit",
    function(e) { validate(e.target); }
    false);
```

如果想捕捉 <div> 元素中发生的所有 mousedown 事件，可以使用如下 addEventListener() 方法：

```
var mydiv = document.getElementById("mydiv");
mydiv.addEventListener('mousedown', handleMouseDown, true);
```

注意，这些例子假定你在 JavaScript 代码的某些地方定义了名为 validate() 和 handleMouseDown() 的方法。

用 addEventListener() 方法注册的事件处理程序在定义它们的作用域中执行，不是由定义为 HTML 性质的事件处理程序使用的作用域链调用（参阅 19.1.6 小节）。

因为在 2 级模型中通过调用方法注册事件处理程序，而不是设置 HTML 性质或 JavaScript 属性来注册事件处理程序，所以可以给一个指定的对象的指定类型的事件注册多个事件处理程序。如果多次调用 addEventListener() 方法，给同一个对象同一类型的事件注册了多个处理函数，那么在该类型的事件在那个对象上发生（或起泡到，或被捕捉到）时，被注册的所有函数都将被调用。DOM 标准不确定调用一个对象的事件处理函数的顺序，所以不应该认为它们以注册的顺序调用：另外还要注意，如果在同一个元素上多次注册了同一个处理函数，那么第一次注册后的所有注册都将被忽略。

为什么想让同一个对象上的同一种事件具有多个处理函数呢？因为它对模块化软件非常有用。例如，假定你编写了一个可重用的 JavaScript 代码模块，该模块用图像上的 mouseover 事件执行图像翻转。再假设你有另一个模块，想使用同一个 mouseover 事件在浏览器的状态栏中显示图像的信息（或图像表示的链接）。在 0 级 API 中，必须把两个模块合并成一个，以便它们能共享 Image 对象的 onmouseover 属性。但在 2 级 API 中，每个模块都可以注册它需要的事件处理程序，而不需要了解或干涉其他模块。

与 addEventListener() 方法配对的是 removeEventListener() 方法，它的三个参数与前者相同，不过是从对象中删除事件处理函数，而不是添加它。通常，临时注册一个事件处理函数，用完后迅速删除它比较有效。例如，在得到一个 mousedown 事件时，可以为 mousemove 事件和 mouseup 事件注册临时捕捉事件处理程序，以便

查看用户是否拖动了鼠标。在 `mouseup` 事件发生时，注销这些处理程序。在这种情况下，事件处理程序删除代码如下所示：

```
document.removeEventListener('mousemove', handleMouseMove, true);
document.removeEventListener('mouseup', handleMouseUp, true);
```

`addEventListener()` 方法和 `removeEventListener()` 方法都由 `EventTarget` 接口定义。在支持 2 级 DOM Event API 的 Web 浏览器中，所有 Document 节点都实现了这个接口。要更详细地了解事件处理程序的注册和注销方法，请查阅 DOM 参考手册部分的 `EventTarget` 接口。

有关事件处理程序注册，最后需要注意的一点是，在 2 级 DOM 中，事件处理程序不仅限于文档元素，还可以给 Text 节点注册处理程序。但在实践中，给包容元素注册处理程序，允许 Text 节点的事件起泡，在容器层处理这些事件会比较简单。

19.2.3 addEventListener()方法和 this 关键字

在原始的 0 级事件模型中，当函数被注册为一个文档元素的事件处理程序时，它就成为了那个文档元素的方法。当调用这个事件处理程序时，将把它作为元素的方法调用。在函数中，关键字 `this` 引用的就是发生事件的元素。

在 Mozilla 和 Netscape 6 中，当用 `addEventListener()` 方法注册了一个事件处理程序时，将以同样的方式处理它，即当浏览器调用那个函数时，便将它作为注册该函数的文档元素的方法调用。但要注意，这是与实现有关的行为，DOM 标准没有要求这样执行。因此，在使用 2 级事件模型时，不应该依赖事件处理函数中的关键字 `this`，而应该使用传递给处理函数的 Event 对象的属性 `currentTarget`。在本章后面讨论 Event 对象时，我们会看到，`currentTarget` 属性引用一个对象，该对象注册了当前的事件处理程序，而且是以可移植的方式注册的。

19.2.4 把对象注册为事件处理程序

可以用 `addEventListener()` 方法注册事件处理函数。如前面一节讨论的，这些函数是否作为注册它们的对象的方法调用，与实现有关。对于面向对象的程序设计方法来说，你更愿意把事件处理程序定义为定制对象的方法，然后作为那个对象的方法调用它们。对于 Java 程序设计者，DOM 标准完全允许这样做，即事件处理程序

是实现了 `EventListener` 接口和 `handleEvent()` 方法的对象。在 Java 中，当注册一个事件处理程序时，传递给 `addEventListener()` 方法的是一个对象，而不是一个函数。为了简单起见，DOM API 的 JavaScript 规约不要求实现 `EventListener` 接口，而只允许给 `addEventListener()` 方法直接传递函数引用。

在编写面向对象的 JavaScript 程序时，如果想用对象作为事件处理程序，那么可以使用如下的函数来注册它们：

```
function registerObjectEventHandler(element, eventType, listener, captures) {  
    element.addEventListener(eventType,  
        function(event) { listener.handleEvent(event); },  
        captures);  
}
```

用这个函数可以把任何对象注册为事件处理程序，只要它定义了 `handleEvent()` 方法。该方法作为监听程序对象的方法而调用，关键字 `this` 引用的是监听程序对象，而不是生成事件的文档元素。这个函数可以运行，因为它使用了嵌套函数直接量，可以捕捉并记住在它的作用域链中的监听程序对象（如果你忘了这一点，可以复习第 11.4 小节）。

虽然不是 DOM 标准的一部分，但 Mozilla 0.9.1 和 Netscape 6.1（但不是 Netscape 6.0 或 6.01）都允许直接把定义了 `handleEvent()` 方法的事件监听程序对象传递给 `addEventListener()` 方法而不是函数引用。对于这些浏览器来说，不需要我们刚才定义的特殊注册函数。

19.2.5 事件模块和事件类型

前面提到过，2 级 DOM 标准是模块化的，所以一个实现可以只支持它的某些部分，省略对其他部分的支持。Event API 就是这样一个模块。可以用下面的代码测试浏览器是否支持该模块：

```
document.implementation.hasFeature("Events", "2.0")
```

但 Event 模块只有用于基本事件处理基础结构的 API。对特定类型的事件的支持则交给子模块。每个子模块提供对某类相关类型的事件的支持，并定义一个 Event 类型，该类型是传递给其他类型的事件处理程序。例如，`MouseEvents` 子模块提供对

mousedown、mouseup、click 和相关事件类型的支持。它还定义了 `MouseEvent` 接口。实现该接口的对象将传递给该模块支持的其他事件类型的处理函数。

表 19-2 列出了所有事件模块，以及它定义的事件接口和它支持的事件类型。注意，2 级 DOM 没有标准化任何类型的键盘事件，所以表中没有列出键盘事件模块。预计 3 级 DOM 标准会支持这种事件类型。

表 19-2: 事件模块、接口和类型

模块名	事件接口	事件类型
HTMLEvents	Event	abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload
MouseEvents	MouseEvent	click, mousedown, mousemove, mouseout, mouseover, mouseup
UIEvents	UIEvent	DOMActivate, DOMFocusIn, DOMFocusOut
MutationEvents	MutationEvent	DOMAttrModified, DOMCharacterDataModified, DOMNodeInserted, DOMNodeInsertedIntoDocument, DOMNodeRemoved, DOMNodeRemovedFromDocument, DOMSubtreeModified

从表 19-2 中可以看到，HTMLEvents 和 MouseEvents 模块定义的事件类型与 0 级事件模块中的事件类型相似。UIEvents 模块定义的事件类型与 HTML 表单元素支持的聚焦、取消焦点和点击事件相似，不过对它们进行了推广，任何能接收焦点或以其他方式激活的文档元素都可以生成这一事件。MutationEvents 模块定义的事件是在文档改变（这种改变是突变）时生成的。这些都是特殊的事件类型，并不常用。

前面提到过，在事件发生时，将传递给它的处理程序一个对象，该对象实现了与该事件类型相关的 `Event` 接口。这个对象的属性提供了事件的细节，这些细节对处理程序可能有用。表 19-3 再次列出了标准事件，但这次以事件类型组织它们，而不是事件以模块组织它们。对于每种事件类型来说，该表列出了传递给它的处理程序的事件对象、这种事件类型是否在事件传播过程中向文档上层起泡（B 列）以及这种事件是否具有能用 `preventDefault()` 方法取消的默认动作（C 列）。对于 HTMLEvents 模块中的事件，该表的第五列列出了哪些 HTML 元素能生成该事件。对于其他事件类型，第五列给了事件对象的哪些属性包含有意义的事件（下一节介

绍这些属性)。注意,这一列中列出的属性不包括基本的Event接口定义的属性,它们对于所有事件类型都包含有意义的值。

比较表 19-3 和表 19-1 很有用,表 19-1 列出的是 HTML 4 定义的 0 级事件处理程序。两个模块支持的事件类型大部分相同(除了 UIEvents 模块和 MutationEvents 模块)。2 级 DOM 标准添加了对 abort、error、resize 和 scroll 事件类型的支持,它们是由 HTML 4 标准化的事件类型(不久我们会看到,传递给 click 事件处理程序的对象的 detail 属性声明了已经发生的连续点击次数)。

表 19-3: 事件类型

事件类型	接口	B	C	由 ... 支持 / 详细属性
abort	Event	是	否	, <object>
blur	Event	否	否	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
change	Event	是	否	<input>, <select>, <textarca>
click	MouseEvent	是	是	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
error	Event	是	否	<body>, <frameset>, , <object>
focus	Event	否	否	<a>, <area>, <button>, <input>, <label>, <select>, <textarea>
load	Event	否	否	<body>, <frameset>, <iframe>, , <object>
mousedown	MouseEvent	是	是	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail

表 19-3: 事件类型 (续)

事件类型	接口	B	C	由 ... 支持 / 详细属性
mousemove	MouseEvent	是	否	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey
mouseout	MouseEvent	是	是	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseover	MouseEvent	是	是	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, relatedTarget
mouseup	MouseEvent	是	是	screenX, screenY, clientX, clientY, altKey, ctrlKey, shiftKey, metaKey, button, detail
reset	Event	是	否	<form>
resize	Event	是	否	<body>, <frameset>, <iframe>
scroll	Event	是	否	<body>
select	Event	是	否	<input>, <textarea>
submit	Event	是	是	<form>
unload	Event	否	否	<body>, <frameset>
DOMActivate	UIEvent	是	是	detail
DOMAttrModified	MutationEvent	是	否	attrName, attrChange, prevValue, newValue, relatedNode
DOMCharacterDataModified	MutationEvent	是	否	prevValue, newValue

表 19-3: 事件类型 (续)

事件类型	接口	B	C	由 ... 支持 / 详细属性
DOMFocusIn	UIEvent	是	否	none
DOMFocusOut	UIEvent	是	否	none
DOMNodeInserted	MutationEvent	是	否	relatedNode
DOMNodeInsertedIntoDocument	MutationEvent	否	否	none
DOMNodeRemoved	MutationEvent	是	否	relatedNode
DOMNodeRemovedFromDocument	MutationEvent	否	否	none
DOMSubtreeModified	MutationEvent	是	否	none

19.2.6 Event 接口和 Event 的深入研究

当事件发生时, 2 级 DOM API 提供了事件的额外信息 (如事件是何时何地发生的), 作为传递给事件处理程序的对象的属性。每个事件模块都有一个相关的事件接口, 该接口声明了该种事件类型的详细信息。表 19-2 (出现在本章前面的小节中) 列出了四种不同的事件模块和四种不同的事件接口。

这四个接口彼此相关, 构成了一个层次。Event 接口是这个层次的根, 所有事件对象实现了这个最基本的事件接口。UIEvent 是 Event 接口的子接口, 实现了 UIEvent 接口的事件对象也实现了 Event 接口的所有方法和属性。MouseEvent 接口是 UIEvent 接口的子接口。这意味着, 传递给 click 事件的事件处理程序的事件对象实现了 MouseEvent 接口、UIEvent 接口和 Event 接口定义的所有方法和属性。最后, MutationEvent 接口是 Event 接口的子接口。

接下来的几节介绍了各种事件接口, 并且强调了它们最重要的属性和方法。在本书的 DOM 参考手册部分可以找到每个接口的完整细节。

19.2.6.1 Event

HTMLEvent 模块定义的事件类型使用 Event 接口。其他事件类型都使用该接口的子接口, 这意味着所有事件对象都实现了 Event 接口, 并提供了适用于所有事件类型

的详细信息。Event 接口定义了如下属性（注意，这些属性和所有 Event 子接口的属性的是只读的）：

type

发生的事件的类型。该属性的值是事件类型名，与注册事件处理程序时使用的字符串值相同（例如，“click”或“mouseover”）。

target

发生事件的节点，可能与 currentTarget 不同。

currentTarget

发生当前正在处理的事件的节点（如当前正在运行事件处理程序的节点）。如果在传播过程的捕捉阶段或起泡阶段处理事件，这个属性的值就与 target 属性的值不同。前面讨论过，在事件处理函数中，应该用这个属性而不是 this 关键字。

eventPhase

一个数字，指定了当前所处的事件传播过程的阶段。它的值是常量，可能值包括 Event.CAPTURING_PHASE、Event.AT_TARGET 或 Event.BUBBLING_PHASE。

timeStamp

一个 Date 对象，声明了事件何时发生。

bubbles

一个布尔值，声明该事件（和这种类型的事件）是否在文档树中起泡。

cancelable

一个布尔值，声明该事件是否具有能用 preventDefault() 方法取消的默认动作。

除了这七个属性之外，Event 接口还定义了两个方法，即 stopPropagation() 和 preventDefault()，所有事件对象都实现了它们。调用 stopPropagation() 方法可以阻止事件从当前正在处理它的节点传播。任何事件处理程序都可以调用 preventDefault() 方法阻止浏览器执行与事件相关的默认动作。在 2 级 DOM API 中，可以调用 preventDefault() 方法，与在 0 级事件模型中返回 false 一样。

19.2.6.2 UIEvent

UIEvent接口是Event接口的子接口。它定义的事件对象类型要传递给DOMFocusIn、DOMFocusOut和DOMActivate类型的事件。这些事件类型不常用。关于UIEvent接口更重要的是，它是MouseEvent接口的父接口。除了Event接口定义的属性外，UIEvent接口还定义了两个属性。

view

发生事件的 Window 对象（在 DOM 术语中称为“视图”）。

detail

一个数字，提供事件的额外信息。对于click事件、mousedown事件和mouseup事件，这个字段代表点击的次数，1代表点击一次，2代表双击，3代表点击三次（注意，每次点击生成一个事件，但如果多次点击的间隔足够近，就可以用detail值说明它。简而言之，detail值为2的鼠标事件，前面总是有一个detail值为1的鼠标事件）。对于DOMActivate事件，这个字段的值为1，表示正常激活，2表示超级激活，如双击鼠标或同时按下**Shift**键和**Enter**键。

19.2.6.3 MouseEvent

MouseEvent接口继承Event接口和UIEvent接口的所有属性和方法，此外它还定义了下列属性：

button

一个数字，声明在mousedown、mouseup和click事件中，哪个鼠标键改变了状态。值为0表示左键，1表示中间键，2表示右键。这个属性只在鼠标键状态改变时使用，例如，在mousemove事件中，它不能用来汇报按键是否被按下并保持住了。还要注意，Netscape 6弄错了它的值，不是用0、1和2表示，而是用1、2和3来表示。Netscape 6.1修正了这个问题。

altKey、ctrlKey、metaKey和shiftKey

这四个布尔值声明在鼠标事件发生时，是否按住了**Alt**键、**Ctrl**键、**Meta**键或**Shift**键。与button属性不同，这些键盘键属性对任何鼠标事件类型都有效。

clientX、clientY

这两个属性声明鼠标指针相对于客户区或浏览器窗口的X坐标和Y坐标。注

意，这两个坐标不考虑文档滚动，如果事件发生在窗口的顶部，无论文档滚动了多远，`clientY` 都是 0。但是，2 级 DOM 没有提供把窗口坐标转换成文档坐标的标准方法。在 Netscape 6 中，加上 `window.pageXOffset` 和 `window.pageYOffset` 即可。在 Internet Explorer 中，加上 `document.body.scrollLeft` 和 `document.body.scrollTop` 即可。

`screenX`, `screenY`

这两个属性声明了鼠标指针相对于用户显示器的左上角的 X 坐标和 Y 坐标。如果你计划在鼠标事件所在地（或附近）打开一个新浏览器窗口，这两个值非常有用。

`relatedTarget`

该属性引用与事件的目标节点相关的节点。对于 `mouseover` 事件来说，它是鼠标移到目标上时所离开的那个节点。对于 `mouseout` 事件，它是离开目标时，鼠标进入的节点。对于其他类型的事件来说，这个属性没有用。

19.2.6.4 MutationEvent

`MutationEvent` 接口是 `Event` 接口的子接口，用于为 `MutationEvents` 模块定义的事件类型提供事件的详细信息。在 DHTML 程序设计方法中，这类事件不常用，所以这里没有提供有关该接口的细节。详情可参阅 DOM 参考手册。

19.2.7 例子：拖移文档元素

现在我们已经讨论过事件传播、事件处理程序注册和 2 级 DOM 事件模型中的各种事件对象接口，最后我们看一看它们是如何使用的。例 19-2 展示了一个 JavaScript 函数 `beginDrag()`，当从 `mousedown` 事件处理程序中调用它时，它允许用户拖动文档元素。

`beginDrag()` 函数有两个参数。第一个参数是要拖动的元素，它可以是发生 `mousedown` 事件的元素或包容元素（例如，允许用户拖住窗口的标题栏移动整个窗口）。但无论哪种情况，它都必须引用一个文档元素，该元素用 CSS `position` 性质绝对定位。在 `style` 性质中，CSS 性质 `left` 和 `top` 必须被明确地设置为像素值。第二个参数是与触发 `mousedown` 事件相关的事件对象。

`beginDrag()` 函数将记录 `mousedown` 事件的位置, 然后为其后将要发生的 `mousemove` 事件和 `mouseup` 事件注册事件处理程序。`mousemove` 事件的处理程序用于移动文档元素, `mouseup` 事件的处理程序用于注销它自身和 `mousemove` 处理程序。注意, `mousemove` 和 `mouseup` 事件的处理程序被注册为捕捉事件处理程序, 因为用户移动鼠标的速度比跟随它移动的文档元素快, 所以其中一些事件发生在原始目标元素外部。另外要注意, 可以注册 `moveHandler()` 函数和 `upHandler()` 函数来处理定义为函数的事件, 这些函数嵌套在 `beginDrag()` 中。因为它们定义在嵌套作用域中, 所以它们可以使用 `beginDrag()` 函数的参数和局部变量, 这些参数和变量极大地简化了它们的实现。

例 19-2: 用 2 级 DOM 事件模型拖动文档元素

```
/**
 * Drag.js:
 * 该函数由 mousedown 事件处理程序调用。
 * 它为随后发生的 mousemove 和 mouseup 事件注册了临时的捕捉事件处理程序。
 * 并用这些事件处理程序拖动指定的文档元素。
 * 第一个参数必须是绝对定位的文档元素。
 * 它可以是接收 mousedown 事件的元素, 也可以是包容元素。
 * 第二个参数必须是 mousedown 事件的事件对象。
 **/
function beginDrag(elementToDrag, event) {
    // 指出该元素当前位于何处。
    // 该元素的样式性质中必须具有 left 和 top CSS 属性。
    // 此外, 我们假定它们采用像素作单位。
    var x = parseInt(elementToDrag.style.left);
    var y = parseInt(elementToDrag.style.top);

    // 计算一个点和鼠标点击的位置之间的距离。
    // 下面嵌套的 moveHandler 函数需要这些值。
    var deltaX = event.clientX - x;
    var deltaY = event.clientY - y;

    // 注册 mousedown 事件后发生的 mousemove 和 mouseup 事件的处理程序。
    // 注意, 它们被注册为文档的捕捉事件处理程序。
    // 在鼠标按钮保持被按下的状态时, 这些事件处理程序保持活动状态,
    // 在按钮被释放时, 它们被删除。
    document.addEventListener('mousemove', moveHandler, true);
    document.addEventListener("mouseup", upHandler, true);

    // 我们已经处理了该事件。不要让别的元素看到它。
    event.stopPropagation();
    event.preventDefault();

    /**
     * 这是在元素被拖动时捕捉 mousemove 事件的处理程序。
     * 它响应移动的元素。
     */
}
```

```

    **/
    function moveHandler(event) {
        // 把元素移到当前的鼠标位置。根据初始鼠标点击的偏移量进行调整。
        elementToDrag.style.left = (event.clientX - deltaX) + "px";
        elementToDrag.style.top = (event.clientY - deltaY) + "px";

        // 不要让别的元素看到该事件。
        event.stopPropagation();
    }

    /**
     * 该处理程序将捕捉拖移结束时发生的 mouseup 事件。
     */
    function upHandler(event) {
        // 注销捕捉事件处理程序。
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
        // 不要让该事件进一步传播。
        event.stopPropagation();
    }
}

```

可以采用下列方式在 HTML 文件中使用 `beginDrag()` 函数（它是例 19-2 的简化版本）：

```

<script src="Drag.js"></script> <!-- 包括 Drag.js 脚本 -->
<!-- 定义要拖动的元素 -->
<div style="position: absolute; left: 100px; top: 100px;
        background-color: white; border: solid black;">
<!-- 定义进行拖移的处理程序。注意 onmousedown 性质， -->
<div style="background-color: gray; border-bottom: dotted black;
        padding: 3px; font-family: sans-serif; font-weight: bold;"
        onmousedown="beginDrag(this.parentNode, event);">
Drag Me <!-- 标题栏的内容 -->
</div>
<!-- 可拖动的元素的内容 -->
<p>This is a test. Testing, testing, testing.<p>This is a test.<p>Test.
</div>

```

关键在于 `<div>` 元素的 `onmousedown` 性质。虽然 `beginDrag()` 使用的是 2 级 DOM 事件模型，但为了方便，我们用 0 级模型注册它。在接下来的小节中，我们将讨论事件模型可以混合。在事件处理程序设置为 HTML 性质时，用关键字 `event` 可以访问事件对象（这不属于 DOM 标准，但它是 Netscape 4 和 IE 事件模型的一个规约。在后面会介绍它们）。

下面是另一个使用 `beginDrag()` 函数的简单例子，它定义了一个图像，用户只在按下并保持住 **Shift** 键时可以拖动该图像：

```

<script src="Drag.js">/script>


```

注意这里的 `onmousedown` 性质和上个例子中的区别。

19.2.8 混合事件模型

迄今为止，我们讨论了传统的0级事件模型和新的标准2级DOM模型。为了向后兼容，支持2级模型的浏览器将继续支持0级事件模型。这意味着可以在文档中使用混合事件模型，就像在前一节中，我们在用于示范元素拖动脚本的HTML代码中所做的那样。

支持2级事件模型的Web浏览器总是传递给事件处理程序一个事件对象，事件处理程序是通过用0级模型设置HTML性质或JavaScript属性注册的。在事件处理程序定义为HTML性质时，它将隐式地转换成的一个函数，该函数有一个参数，名为 `event`。这意味着这样一个事件处理程序可以用标识符 `event` 引用事件对象。

DOM标准从未标准化0级事件模型。对于支持 `onclick` 性质的HTML元素来说，它甚至不要求具有相应的 `onclick` 属性。但该标准规定仍然可以使用0级事件模型，并支持0级事件模型的实现，就像用 `addEventListener()` 方法注册的事件处理程序那样用该模型注册处理程序。简而言之，如果把一个函数 `f` 赋予文档元素 `e` 的 `onclick` 属性（或设置相应的HTML `onclick` 性质），等价于用下列方法注册该函数：

```
e.addEventListener("click", f, false);
```

当 `f()` 被调用时，可以传递给它一个事件对象作为参数，即使它是用0级模型注册的。另外，如果把 `onclick` 属性的值从函数 `f` 改为函数 `g`，可以使用下面的代码：

```
e.removeEventListener("click", f, false);
e.addEventListener("click", g, false);
```

但要注意，2级标准没有规定是否可以调用 `removeEventListener()` 方法删除用 `onclick` 属性赋值注册的事件处理程序。在本书编写的过程中，Mozilla/Netscape 实现不允许这样做。

19.2.9 合成事件

2级DOM标准包括创建和分派合成事件的API。这种API允许在程序控制下生成事件，而不是在用户控制下生成事件。虽然这个特性不太常用，但在制造加回归测试时，它非常有用，例如，使一个DHTML应用程序受已知的事件序列控制，然后验证结果相同。它还可以用于实现一个宏播放工具，自动执行用户界面的动作。但合成事件API不适合制造自运行的演示程序。例如，可以创建一个合成mousemove事件，把它传递给程序中合适的事件处理程序，但这不能真正使鼠标指针在屏幕上移动。

但是，在本书编写过程中，Netscape 6和Mozilla的当前版本都不支持合成事件API。

要生成一个合成事件，必须完成下面三个步骤：

- 创建一个合适的事件对象。
- 初始化这个事件对象的域。
- 把事件对象分派给想得到它的文档元素。

要创建事件对象，需要调用Document对象的createEvent()方法。该方法只有一个参数，它是应该创建的事件对象的事件模块名。例如，要创建适用于click事件的事件对象，可调用如下的createEvent()方法：

```
document.createEvent("HTMLEvents");
```

要创建适用于所有鼠标事件类型的事件对象，可调用如下方法：

```
document.createEvent("MouseEvent");
```

注意，createEvent()方法的参数是复数。虽然这让人觉得不太习惯，但它和传递给hasFeature()方法来测试浏览器是否支持某个事件模块的字符串相同。

创建了事件对象后，下一步是初始化它的属性。但事件对象的属性是只读的，所以不能直接给它们赋值，而必须调用方法执行初始化。虽然前面对Event对象、MouseEvent对象和其他事件对象的介绍都只提到了属性，但每个对象还定义了初始化事件属性的方法。这个初始化方法的名字由要初始化的事件对象的类型决定，参

数个数和要设置的属性个数相同。注意，只能在分派合成事件之前调用事件初始化方法，不能用这些方法修改传递给事件处理程序的事件对象的属性。

让我们看两个例子。我们知道，click 事件是 HTMLEvents 模块的一部分，它使用 Event 类型的事件对象。这些对象由 initEvent() 方法初始化，如下所示：

```
e.initEvent("click", "true", "true");
```

另一方面，mousedown 事件是 MouseEvents 模块的一部分，它使用 MouseEvent 类型的事件对象。这些对象由 initMouseEvent() 方法初始化，该方法具有许多参数：

```
e.initMouseEvent("mousedown", true, false, // Event 属性  
window, 1, // UIEvent 属性  
0, 0, 0, 0, // MouseEvent 属性  
false, false, false, false,  
0, null);
```

注意，只能把事件模块名传递给 createEvent() 方法。真实的事件类型名传递给事件初始化方法。DOM 标准不要求使用某个预定义的名字。可以用任何选中的事件类型名创建事件，只要它不以数字或前缀“DOM”（无论是大写、小写，还是大小写混合）开头。如果用自定义事件类型名初始化合成事件，还必须用事件类型名注册事件处理程序。

在创建并初始化了事件对象后，可以把它传递给一个合适的文档元素的 dispatchEvent() 方法来分派它。dispatchEvent() 方法是由 EventTarget 接口定义的，所以可以将它作为支持 addEventListener() 方法和 removeEventListener() 方法的文档节点的方法。被分派了事件的元素将成为事件目标，事件对象将经历常规的事件传播序列。在事件传播的每个阶段，创建的事件对象都被传递给事件处理程序，该事件处理程序为初始化事件时指定的事件类型进行注册。最后，在事件传播结束时，对 dispatchEvent() 的调用将返回。如果任何事件处理程序调用了事件对象的 preventDefault() 方法，返回值是 false，否则返回值是 true。

19.3 Internet Explorer 事件模型

Internet Explorer 4、5、5.5 和 6 支持的事件模型是中间模型，这个模型介于原始的 0 级模型和标准的 2 级 DOM 模型之间。IE 事件模型包括 Event 对象，该对象提供发

生的事件的详细情况。但 Event 对象不是传递给事件处理函数，而是作为 Window 对象的属性。IE 模型支持起泡形式的事件传播，但不支持 DOM 模型的捕捉形式的事件传播。在 IE 4 中，注册事件处理程序的方式与原始的 0 级模型注册它们的方式相同。但在 IE 5 和其后的版本中，可以用专门的（但非标准的）注册函数注册多个处理程序。

接下来的几节提供了这种事件模型的详细情况，并把它与原始的 0 级事件模型和标准的 2 级事件模型进行了比较。在阅读 IE 模型之前，应该确保已经了解了这两种事件模型。

19.3.1 IE Event 对象

与标准的 2 级 DOM 事件模型一样，IE 事件模型提供了在 Event 对象属性中发生的每个事件的详细情况。由标准模型定义的 Event 对象是由 IE Event 对象进行模型化的，所以应该注意，IE Event 对象和 DOM Event、UIEvent、MouseEvent 对象有大量相似的属性。

IE Event 对象最重要的属性如下所示：

`type`

一个字符串，声明发生的事件的类型。该属性的值是删除前缀“on”的事件处理程序名（如“click”或“mouseover”）。它与 DOM Event 对象的 `type` 属性兼容。

`srcElement`

发生事件的文档元素。与 DOM Event 对象的 `target` 属性兼容。

`button`

一个整数，声明被按下的鼠标键。值为 1 表示左键，2 表示右键，4 表示中间键。如果按下了多个键，这些值将加在一起。例如，左键和右键一起按下，则值为 3。把这个属性与 2 级 DOM 的 MouseEvent 对象的 `button` 属性比较，尽管它们的属性名相同，但属性值的解释却不同。

`clientX`, `clientY`

这两个整数属性声明事件发生时鼠标的坐标，其值是相对于包容窗口的左上角

生成的。注意，对于比窗口大的文档，这些坐标与在文档中的位置不同。可以加上 `document.body.scrollLeft` 和 `document.body.scrollTop` 属性的值来计算滚动的位置。这两个属性与 2 级 DOM 中的 `MouseEvent` 对象的同名属性兼容。

`offsetX, offsetY`

这两个整数声明鼠标指针相对于源元素的位置。用它们可以确定点击了 `Image` 对象的哪个像素。2 级 DOM 事件模型中没有与它们等价的属性。

`altKey, ctrlKey` 和 `shiftKey`

这些布尔属性声明在鼠标事件发生时，是否按住了 **Alt** 键、**Ctrl** 键或 **Shift** 键。它们与 2 级 DOM 中的 `MouseEvent` 对象的同名属性兼容。但要注意，IE Event 对象没有 `metaKey` 属性。

`keyCode`

这个整数属性声明了 `keydown` 和 `keyup` 事件的键代码以及 `keypress` 事件的 Unicode 字符。用 `String.fromCharCode()` 方法可以把字符代码转换成字符串。2 级 DOM 事件模型没有标准化按键事件（但 3 级 DOM 正致力于完成这一目标），没有与该属性等价的属性。

`fromElement, toElement`

`fromElement` 声明 `mouseover` 事件中鼠标移动过的文档元素。`toElement` 声明 `mouseout` 事件中鼠标移到的文档元素。它们等价于 2 级 DOM 中的 `MouseEvent` 对象的 `relatedTarget` 属性。

`cancelBubble`

一个布尔属性。把它设为 `true`，可以阻止当前事件进一步起泡到包容层次的元素。与 2 级 DOM 的 Event 对象的 `stopPropagation()` 方法相同。

`returnValue`

一个布尔属性。把它设为 `false` 可以阻止浏览器执行与事件相关的默认动作。它可以替代由事件处理程序返回 `false` 的老方法。它等价于 2 级 DOM 的 Event 对象的 `preventDefault()` 方法。

在本书的客户端参考手册部分可以找到 IE Event 对象的完整说明。

19.3.2 作为全局变量的 IE Event 对象

虽然 IE 事件模型提供了 Event 对象中事件的详细情况，但它不把 Event 对象作为参数传递给事件处理程序，而通过全局 Window 对象的 event 属性访问 Event 对象。这意味着 IE 中的事件处理程序可以用 window.event 或只用 event 引用 Event 对象。虽然在用函数参数的地方使用全局变量看起来比较奇怪，但 IE 模式能够运行，因为在事件驱动的程序设计模型中，一次只处理一个事件。由于从来不会并发处理两个事件，因此用全局变量存储当前在处理的事件的详细信息很安全。

虽然 Event 对象是全局变量这一事实与标准的 2 级 DOM 事件模型不兼容，但仅用一行代码就可以避开它。如果想编写任何一种事件模型都可以使用的事件处理函数，可以编写一个函数来得到一个参数。如果没有传递给它参数，就用全局变量初始化参数。例如：

```
function portableEventHandler(e) {  
    if (!e) e = window.event;    // 获取 IE 的事件细节  
    // 此处是事件处理程序的主体  
}
```

19.3.3 IE 中的事件起泡

IE 事件模型没有 2 级 DOM 模型具有的事件捕捉的概念。但在 IE 模型中，事件可以沿着包容层次向上起泡，就像它们在 2 级模型中所做的一样。在 2 级模型中，事件起泡只适用于原始事件或输入事件（主要是鼠标和键盘事件），不适用于高级的语义事件。IE 中的事件起泡和 2 级 DOM 事件模型中的事件起泡之间的差别在于停止起泡的方式。IE Event 对象没有 DOM Event 对象具有的 stopPropagation() 方法，所以要阻止事件起泡或制止它在包容层次中进一步传播，IE 事件处理程序必须把 Event 对象的 cancelBubble 属性设为 true：

```
window.event.cancelBubble = true;
```

注意，设置 cancelBubble 属性只适用于当前事件。当新事件生成时，将赋予 window.event 新的 Event 对象，cancelBubble 属性将被还原为它的默认值 false。

19.3.4 IE 事件处理程序的注册

在 IE 4 中,注册事件处理程序的注册方法与原始的 0 级模型采用的方法相同,即把它们设置为 HTML 性质或把函数赋予文档元素的事件处理程序属性。惟一的区别是,IE 4 允许访问文档中的所有元素(或给所有元素注册事件处理程序),而不仅限于 0 级 DOM 可以访问的表单、图像或链接元素。

IE 5 和其后的版本引入了 `attachEvent()` 方法和 `detachEvent()` 方法,它们提供了为指定对象事件类型注册多个处理函数的方法。它们与 `addEventListener()` 方法和 `removeEventListener()` 方法相似,只是由于 IE 事件模型不支持事件捕捉,因此它们只有两个参数,即事件类型和处理函数。另外,与 2 级 DOM 事件模型不同,传递给 IE 方法的事件处理程序名应该包括前缀“on”,例如,用“onclick”代替“click”。可以用 `attachEvent()` 方法注册事件处理程序:

```
function highlight() { /* 此处是事件处理程序代码 */ }
document.getElementById("myelt").attachEvent('onmouseover', highlight);
```

`attachEvent()` 方法和 `addEventListener()` 方法之间的另一个差别是,用 `attachEvent()` 注册的函数将被作为全局函数调用,而不是作为发生事件的文档元素的方法。也就是说,在 `attachEvent()` 注册的事件处理程序执行时,关键字 `this` 引用的是 Window 对象,而不是事件的目标元素。

19.3.5 例子: 用 IE 事件模型拖动文档元素

例 19-3 是例 19-2 中展示的 `beginDrag()` 函数的修改版本。这个版本除了包括在 2 级 DOM 事件模型中能运行的代码外,还包括在 IE 事件模型中能运行的代码。这个 `beginDrag()` 版本的设计和用途与例 19-2 中的版本相同,所以,如果你理解了那个例子,就能理解这个例子。这个例子令人感兴趣的地方在于它清楚地强调了两种事件模型的差别。

在该代码的 IE 版本中,最重要的差别是,它必须依靠事件起泡,而不是事件捕捉。虽然这样通常可以正常运行,但不是这个问题的理想解决方法。要注意的另一个重要差别是,不能给 IE 事件处理程序传递 Event 对象。注意,这个例子中的代码还区分了 IE 5 和其后的版本,它们支持 `attachEvent()` 方法,但 IE 4 不支持它。参阅例 19-2 的讨论,它提供了使用 `beginDrag()` 函数的 HTML 示例文档。

例 19-3: 用 IE 事件模型拖动文档元素

```

/**
 * PortableDrag.js:
 * beginDrag() 由 onmousedown 事件处理程序调用。
 * elementToDrag 可以是接收 mousedown 事件的元素，或者接收包容元素。
 * 事件必须是用于 mousedown 事件的 Event 对象。
 * 这种实现在 2 级 DOM 事件模型和 IE 事件模型中都能运行。
 */
function beginDrag(elementToDrag, event) {
    // 计算元素的左上角和鼠标点击位置之间的距离。
    // moveHandler 函数需要这些值。
    var deltaX = event.clientX - parseInt(elementToDrag.style.left);
    var deltaY = event.clientY - parseInt(elementToDrag.style.top);

    // 注册响应 mousemove 事件和 mousedown 事件后的 mouseup 事件的处理程序。
    if (document.addEventListener) { // 2 级 DOM 事件模型。
        // 注册捕捉事件处理程序。
        document.addEventListener('mousemove', moveHandler, true);
        document.addEventListener('mouseup', upHandler, true);
    }
    else if (document.attachEvent) { // IE 5+ 的事件模型
        // 在 IE 事件模型中，我们不能捕捉事件，所以只有当事件起泡到这些处理程序时，
        // 它们才被触发。假设不存在干涉元素，处理了事件后它们停止传播。
        document.attachEvent("onmousemove", moveHandler);
        document.attachEvent("onmouseup", upHandler);
    }
    else { // IE 4 事件模型。
        // 在 IE 4 我们不能使用 attachEvent() 方法，所以存储了以前赋予的处理
        // 程序后，直接赋予新的事件处理程序，这样还可以恢复旧的处理程序。
        // 注意，这同样依赖于事件起泡。
        var oldmovehandler = document.onmousemove;
        var olduphandler = document.onmouseup;
        document.onmousemove = moveHandler;
        document.onmouseup = upHandler;
    }

    // 我们处理了该事件，不要再让其他元素见到它。
    if (event.stopPropagation) event.stopPropagation(); // 2 级 DOM
    else event.cancelBubble = true; // IE

    // 下面禁止执行默认动作
    if (event.preventDefault) event.preventDefault(); // 2 级 DOM
    else event.returnValue = false; // IE

    /**
     * 这是在元素被拖移时捕捉 mousemove 事件的处理程序，
     * 它负责移动元素。
     */
    function moveHandler(e) {
        if (!e) e = window.event; // IE 事件模型。
        // 把元素移到鼠标当前的位置，根据初始鼠标点击的偏移量进行调整。
    }
}

```

```

elementToDrag.style.left = (e.clientX - deltaX) + "px";
elementToDrag.style.top = (e.clientY - deltaY) + "px";

// 不要再让其他元素看到该事件。
if (e.stopPropagation) e.stopPropagation(); // 2级DOM
else e.cancelBubble = true; // IE
}

/**
 * 这是捕捉拖移结束时最后发生的 mouseup 事件的处理程序。
 */
function upHandler(e) {
    if (!e) e = window.event; // IE 事件模型。

    // 注销捕捉事件处理程序。
    if (!document.removeEventListener) { // DOM 事件模型。
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
    }
    else if (document.detachEvent) { // IE 5+ 事件模型。
        document.detachEvent("onmouseup", upHandler);
        document.detachEvent("onmousemove", moveHandler);
    }
    else { // NP 4 事件模型。
        document.onmouseup = oldupHandler;
        document.onmousemove = oldmoveHandler;
    }

    // 不要再让事件进一步传播。
    if (e.stopPropagation) e.stopPropagation(); // 2级DOM
    else e.cancelBubble = true; // IE
}
}

```

19.4 Netscape 4 事件模型

Netscape 4 的事件模型与原始的 0 级事件模型相似，只是它在传递给处理函数的 Event 对象中提供事件的详细信息。它还支持启用事件捕捉的专用方法。接下来的小节将对这些特性进行解释。

19.4.1 Netscape 4 Event 对象

Netscape 4 事件模型定义了 Event 对象，用于存放发生的事件的详细情况。与 2 级 DOM 模型一样，它把 Event 对象作为参数传递给所有事件处理程序。但是，Netscape

4的Event对象的属性几乎与IE Event对象和各种2级DOM事件对象的属性完全不同。Netscape 4事件模型中的关键Event属性包括:

type

一个字符串, 声明发生的事件的类型。这个字符串是事件处理程序名删除前缀“on”构成的(如“click”或“mousedown”)。该属性与IE和2级DOM的Event对象兼容。

target

发生事件的文档元素。该属性与2级DOM的Event对象的target属性兼容, 与IE Event对象的srcElement属性兼容。

pageX、pageY

这两个属性声明了发生事件的像素坐标, 该坐标是相对于窗口的左上角而言的。对于比窗口大的文档, 应该加上window.pageXOffset和window.pageYOffset偏移量, 以便把它们转换成文档坐标。与2级DOM的MouseEvent对象和IE Event对象中的clientX和clientY属性兼容。

which

一个整数, 声明哪个鼠标键或键盘键被按住了。对于鼠标事件, 值1、2和3分别代表左键、中间键和右键。可以把该属性与2级DOM的MouseEvent对象和IE Event对象的button属性进行比较(互不相容)。对于键盘事件, 该属性存放了被按下的键的Unicode码。可以把该属性与IE Event对象的keyCode属性进行比较。

modifiers

一个整数, 声明事件发生时, 哪个键盘组合键被按下了。它的值是一个位掩码, 由下列值构成, 包括Event.ALT_MASK、Event.CONTROL_MASK、Event.META_MASK和Event.SHIFT_MASK。与2级DOM的MouseEvent对象和IE Event对象的altKey、ctrlKey、metaKey和shiftKey属性兼容。

在Netscape 4事件模型中, Event对象将被传递给所有事件处理程序。当在HTML性质中把事件处理程序定义为JavaScript代码串时, 可以将它隐式地转换成一个函数, 该函数具有名为event的参数。这意味着HTML事件处理程序可以用标识符event引用Event对象(在IE模型中, event标识符引用全局Event对象。尽管实现不同, 但实际结果相同)。

为了向后兼容, Mozilla 和 Netscape 6 使用的 Event 对象实现了 Netscape 4 Event 对象的大部分属性。不过在本书编写过程中, 有一个例外, 即 modifiers 属性。

19.4.2 Netscape 4 中的事件捕捉

Netscape 4 事件模型不支持事件起泡 (IE 事件模型支持), 但它支持形式有限的事件捕捉 (像 2 级 DOM 模型一样) (事实上, DOM 标准的事件传播模型是 Netscape 捕捉模型和 IE 起泡模型的组合)。虽然 Netscape 4 支持事件捕捉, 但它运行的方式却与 2 级 DOM 事件模型定义的非常不同。

在 Netscape 4 中, Window、Document 和 Layer 对象在生成事件的元素处理事件之前, 可以要求预览事件类型。这样的请求是由这些对象的 captureEvents() 方法发出的。这个方法的参数指定了要捕捉的事件的类型, 它是一个位掩码, 由定义为 Event 构造函数的静态属性的常量构成。例如, 如果一个程序想在目标对象处理 mousedown 和 mouseup 事件之前把它们传递给 Window 对象, 可以调用如下 captureEvents() 方法:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

发出接收事件的请求后, 程序必须为这些事件注册处理程序:

```
window.onmousedown = function(event) { ... };  
window.onmouseup = function(event) { ... };
```

当其中一个捕捉事件处理程序接收到事件时, 它必须确定接下来应该做什么。在某些程序中, 不再进一步处理或传播捕捉到的事件。但在其他环境中, 程序则想继续传递事件。如果把事件传递给 Window 对象、Document 对象或 Layer 对象的 routeEvent() 方法, 该方法将把事件传递给下一个用 captureEvents() 方法声明对那种事件类型感兴趣的 Window 对象、Document 对象或 Layer 对象。否则, 如果没有其他要传递事件的捕捉对象, 事件将被传递给它的源对象, 然后调用该对象相应的事件处理程序。例如:

```
function clickHandler(event) {  
    if (event.which == 3) { // 它是鼠标右键  
        // 此处处理该事件, 不做别的操作  
        // 该事件不再进一步传播  
    }  
    else { // 它不是鼠标右键
```



```
// 我们对该事件不感兴趣，所以让它传播到对它感兴趣的元素  
window.routeEvent(event);  
};
```

除了调用 `routeEvent()` 方法，还可以把 `Event` 对象传递给对象的 `handleEvent()` 方法（你可以把事件传递给该对象）。`handleEvent()` 方法将把事件传递给那个对象适合的事件处理程序。

当 `Window` 对象、`Document` 对象或 `Layer` 对象不再希望捕捉事件时，它应该调用 `releaseEvents()` 方法，它的参数与传递给 `captureEvents()` 方法的参数相同。

Netscape 4 的事件捕捉模型基本上与 2 级 DOM 的事件模型不兼容。例如，DOM 模型默认传播捕捉到的事件，但 Netscape 模型则不然。Mozilla 和 Netscape 6 实现了 Netscape 4 的事件捕捉 API，但这些 API 看起来没有什么作用。

19.4.3 例子：用 Netscape 4 事件模型拖动文档元素

例 19-4 是我们熟悉的 `beginDrag()` 方法的另一个实现，它采用了 Netscape 4 的事件模型（和 Netscape 4 基于层的 DOM）。它示范了如何捕捉事件，以及如何为该事件模型编写事件处理程序。这个例子包括 JavaScript 代码和使用 `beginDrag()` 方法定义用户可以拖动的图像的简单 HTML 文档。比较 `beginDrag()` 的这个实现和它的前两个版本。注意，这个例子在 `beginDrag()` 函数的开头定义它的嵌套事件处理程序，而不是在它的末尾定义。这避免了一个 bug，如果把嵌套函数放在 `beginDrag()` 函数的末尾，那么它们在 Netscape 4 中不能运行。另外还要注意例子末尾的 `onmousedown` 处理程序，它只在按下并保持住 **Shift** 键时才允许进行拖动，它用 Netscape 4 的 `Event` 对象 API 来测试这一组合键，这个 API 与 2 级 DOM 和 IE 的 API 有极大不同（注 2）。

例 19-4：在 Netscape 4 中拖动文档元素

```
<script>  
/**  
 * 该函数用于层中对象的 mousedown 事件处理程序。
```

注 2： 在本书编写过程中，Mozilla 和 Netscape 6 没有保持与 Netscape 4 的 `Event` 对象的 `modifiers` 属性的兼容性，所以这里所示的 `onmousedown` 处理程序只在 Netscape 4 中运行，不在 Netscape 6 中运行。

```

* 它的第一个参数必须是 layer 对象。
* 第二个参数必须是表示mousedown 事件的 Event 对象。
**/
function beginDrag(layerToDrag, event) {
    // 该嵌套函数响应mousemove 事件并移动层。
    function moveHandler(event) {
        // 把元素移到鼠标的当前位置, 根据初始的鼠标点击偏移量进行调整。
        layerToDrag.moveTo(event.pageX - deltaX, event.pageY - deltaY);

        // 不要采用默认动作, 不要再进一步传播。
        return false;
    }

    // 该嵌套函数处理mouseup 事件。
    // 它将停止捕捉事件, 并注销该处理程序。
    function upHandler(event) {
        // 停止捕捉和处理拖动事件。
        document.releaseEvents(Event.MOUSEMOVE | Event.MOUSEUP);
        document.onmousemove = null;
        document.onmouseup = null;

        // 不采用默认动作, 不再进一步传播。
        return false;
    }

    // 计算层左上角和鼠标点击位置之间的距离。
    // 下面的moveHandler 函数需要这些值。
    var deltaX = event.pageX - layerToDrag.left;
    var deltaY = event.pageY - layerToDrag.top;

    // 安排捕捉mousemove 和mouseup 事件。
    // 然后安排用下面定义的函数处理它们。
    document.captureEvents(Event.MOUSEMOVE | Event.MOUSEUP);
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}
</script>
<!-- 以下是如何在Netscape 4中使用beginDrag()函数的方法 -->
<!-- 用CSS性质定义一个层 -->
<div id="div1" style="position:absolute; left:100px; top:100px;">
<!-- 赋予层一些内容和mousedown 事件处理程序 -->

</div>

```

第二十章

兼容性

JavaScript 与 Java 一样是一种新的“独立于平台”的语言。也就是说，可以用 JavaScript 开发一个程序，并且要求这个程序在任何启用 JavaScript 的 Web 浏览器中都可以无改变地运行，无论那个浏览器在什么类型的机器上运行，也无论它在什么类型的操作系统上运行。不过这只是一种理想状况，目前还无法达到那种完美的状态。

兼容性的问题依然存在，而且可能一直存在下去，这是我们 JavaScript 的程序设计者们应该谨记的。我们必须牢记一个事实，网络的种类是很多的。JavaScript 程序可以在不同的操作系统上运行，这些操作系统也许要涉及三个或更多个版本的浏览器，而这些浏览器又至少来自两个不同的厂商。这对于我们这些经历过基于平台的程序开发时期的人来说，是难以理解的。那个时候我们只在特定的平台上开发。要记住，在哪个平台上开发程序无关紧要。你的程序可能在某个平台上运行得很好，但是真正要测试的是它是否在所有使用它的平台上都能运行良好。

兼容性问题主要分成两大类，一类是和平台有关的、浏览器有关的和版本有关的特性，另一类是 bug 和语言级别产生的不兼容性，其中包括 JavaScript 程序和非 JavaScript 浏览器之间的不兼容问题。本章将讨论处理这两大领域中的不兼容性问题。如果已经采用自己的方式阅读了前面的章节，那么你可能已经是一个 JavaScript 的程序设计高手了，而且可能开始编写重要的 JavaScript 程序了。但是，在阅读本章之前千万不要在 Internet 上（或各种 Intranet 上）发布那些程序。

20.1 平台和浏览器的兼容性

在开发具有成品品质的 JavaScript 代码时，与平台有关的、厂家有关的和版本有关的不兼容性测试及相关知识是你的主要助手。例如，如果知道在 Macintosh 平台上的 Netscape 2 中，系统时间与标准时间总会相差一个小时左右，你就可以采取步骤来处理它。如果知道 Windows 平台在鼠标指针离开一个超文本链接时，不能自动清除对状态栏的设置，就可以提供一个合适的事件处理程序明确地将状态栏中的信息清除掉。如果知道 Internet Explorer 4 和 Netscape 4 支持的动态 HTML 模型完全不同，就可以根据当前正在使用的浏览器，选择合适的机制来编写代码。

了解已经存在的不兼容性是编写具有兼容性的代码的关键。遗憾的是，要列出一个所有已知的厂商、版本和平台的不兼容性的明确列表，其工作量是非常巨大的，显然不会有人当真尝试这样去做。虽然可以从 Internet 上得到一些帮助，但你主要还是应该依靠自己的经验和测试结果。一旦证实了某个领域中的不兼容性，还可以使用接下来几节介绍的基本方法来处理它。

20.1.1 最小公分母法

处理不兼容性的一种方法是避开它。例如，Date 对象是 Netscape 2 中一个经常遇到的 bug。如果想让 Netscape 2 的用户能够使用你的程序，只需要避免使用 Date 对象即可（注 1）。

另一个例子是，Netscape 3 和 Internet Explorer 3 都支持 Window 对象的 opener 属性，但是 Netscape 2 不支持这一属性。所谓最小公分母法就是说，如果要与 Netscape 2 兼容，就不应该使用这一属性。相反，应该在打开一个新窗口时创建一个与之等价的属性：

```
newwin = window.open("", "new", "width=500, height=300");  
newwin.creator = self;
```

如果在所有新创建的窗口中都使用属性 creator，那么你就可以依赖这个属性，用它替换那个不可移植的 opener 属性（之后我们可以看到，另一种选择是放弃与

注 1：通常不建议这样做。在本书的编写过程中，NetScape 2 已经过时了，可以忽略它。

Netscape 2 的兼容性, 而使用支持 JavaScript 1.1 和其后版本的浏览器, 因为所有这样的浏览器都支持 opener 属性)。

使用这种方法, 就只能使用那些在所有的目标平台上都能够运行的特性。虽然它不允许你编写“尖端”程序, 但是用它生成程序却非常安全, 而且具有很强的可移植性, 能够使用很多重要的功能。

20.1.2 防御性编码

如果使用防御性编码的方法来达到兼容性, 就要为与平台有关的不兼容性编写代码, 这些代码要回避不兼容性的部分, 并且必须独立于平台。例如, 如果从 Window 对象的事件处理程序 onmouseover 中设置了 status 属性, 在状态栏中显示客户消息, 那么除了在 Netscape 2 和 3 的 Windows 版本中外, 当把鼠标从超文本链接移开时, 状态栏中的消息就会被清除。要纠正这一点, 只添加事件处理程序 onmouseout 来清除状态栏即可。这种预防措施修正了当前 (和将来) 具有 bug 的平台, 但不影响没有 bug 的平台。

20.1.3 特性检测

特性检测是处理不兼容性的有效方法。如果你想用的特性不是所有浏览器都支持, 那么就在脚本中添加代码, 测试这种特性是否被支持。如果当前的平台不支持需要的特性, 那么就不要再在该平台上使用该特性, 或者提供在所有平台上都可以运行的替代代码。

再次考虑 opener 属性。在最小公分母法中, 我们只是建议避免使用这个属性, 在所有平台上都使用替代代码。用特性检测方法, 则只在当前平台不支持 opener 属性时才使用替代代码:

```
newwin = window.open("", 'new', "width=500, height=300");  
if (!newwin.opener) newwin.opener = self;
```

注意, 我们如何检测 opener 属性的存在性。该方法同样适用于检测方法的存在性。例如, String 对象的 split() 方法只存在于 JavaScript 1.1 实现中。我们可以自己编写该函数, 使它在所有 JavaScript 版本中都能运行, 但为了效率起见, 我们在支

持它的平台上还是使用快速的内置方法。因此,采用特性检测方法对一个字符串执行 `Split()` 操作的代码如下所示:

```
if (s.split)           // 检查该方法是否存在,但不调用它
    a = s.split(':');   // 如果它存在,则可以安全地调用它
else                    // 否则,
    a = mysplit(s, ':'); // 采用我们的替代实现
```

特性检测通常用于执行只有某些浏览器支持或不同浏览器实现的不同的DHTML效果。例如,如果你设计一个包含图像翻转效果的站点,可以采用下列代码进行特性检测:

```
if (document.images) { // 如果浏览器定义了images[]数组,
                        // 我们在此加入图像翻转代码。
}
// 否则,我们省略图像翻转的效果。
```

另一个例子是,假定我们想使用动态定位的文档元素。不同的浏览器实现这一目标的API不同,所以我们首先用特性检测方法,看一看当前浏览器支持什么API,代码如下:

```
if (document.getElementById) { // 如果浏览器支持W3C DOM API,
    // 则可以用W3C DOM API实现我们的DHTML效果。
}
else if (document.all) {       // 如果浏览器支持IE 4 API,
    // 则可以用IE 4 API实现DHTML效果。
}
else if (document.layers) {    // 如果浏览器支持Netscape 4 API,
    // 则可以用Netscape 4 API实现DHTML效果(我们所能实现的最好效果)。
}
else {                          // 否则,浏览器不支持DHTML,
    // 所以提供DHTML的静态替代版本(如果我们可以做到的话)。
}
```

特性检测方法的好处是,它生成的代码不受特定的浏览器厂商或浏览器版本号的限制。这种代码不仅可以在现有的浏览器中运行,在将来的浏览器(无论它们实现了什么特性集)中也应该可以运行。

20.1.4 与平台有关的回避方法

特性检测方法很适合检测对大型功能域的支持。例如,可以用它来判断一个浏览器是否支持图像翻转或W3C DOM API。另外,有时可能应该避开某个浏览器的个别

bug 或异常行为, 可能没有测试这种 bug 存在性的简单方法。在这种情况下, 需要创建一种与平台有关的回避方法, 这些方法受特殊的浏览器厂商、版本或操作系统 (或三者的组合) 的限制。

回忆一下, 第十三章中介绍过, Window 对象的属性 navigator 能够提供浏览器的厂商和版本以及正在运行该对象的操作系统的信息。可以使用这些信息在程序中插入一些与平台有关的代码。

与平台有关的回避方法的示例涉及到 Document 对象的 bgColor 属性。在 Windows 和 Macintosh 的平台上, 可以在运行时设置这一属性, 从而改变文档的背景颜色。但是, 当在 Netscape 2 和 3 的 Unix 版本上设置该属性时, 虽然颜色也能改变, 但文档的内容却会暂时消失。如果你想通过改变背景颜色来创造一种特殊的效果, 那么可以使用 Netscape 对象来测试采用的是否是 Unix 平台, 这样就可以跳过这两种平台带来的特殊效应。使用的代码如下所示:

```
// 检测是否在 Unix 平台上的 Netscape 2 或 3 中运行
var nobg = (parseInt(navigator.appVersion) < 4) &&           // 版本
            (navigator.appName.indexOf('Netscape') != -1) && // 销售商
            (navigator.appVersion.indexOf('X11') != -1);      // OS
// 如果不是, 则使页面的背景颜色动起来
if (!nobg) animate_bg_color();
```

在编写与平台有关的回避方法时, 常用“客户探测器”来判断当前使用的是哪种平台。这种方式通常基于 navigator 对象的属性。运行一次客户探测器代码, 它就会设置描述当前平台的变量。此后不必再解析你的代码中为每个与平台有关的位设置的 navigator 属性, 只使用探测器设置的变量即可。适用于多种用途的简单探测器代码如下所示:

```
var browserVersion = parseInt(navigator.appVersion);
var isNetscape = navigator.appName.indexOf("Netscape") != -1;
var isIE = navigator.appName.indexOf('Microsoft') != -1;
var agent = navigator.userAgent.toLowerCase();
var isWindows = agent.indexOf('win') != -1;
var isMac = agent.indexOf('mac') != -1;
var isUnix = agent.indexOf('X11') != -1;
```

用上面定义的变量, 可以编写如下代码:

```
if (isNetscape && browserVersion < 4 && isUnix) {
    // 此处避开 Unix 平台上的 Netscape 3 中的 bug
}
```

在 Internet 上可以找到许多预编写的客户探测器。下面是一个内容比较全面的站点 (具有关于用法的讨论): http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html。

20.1.5 通过服务器端的脚本达到兼容性的方法

如果你的 Web 应用程序使用了服务器端的脚本, 如 CGI 脚本或服务器端的 JavaScript 程序, 那么还有一种与平台有关的实现兼容性的方法。一个位于服务器端的程序可以检查 HTTP 请求头的 User-Agent 字段, 这使得它可以判断用户正在运行的是什么浏览器。有了这一信息, 它就能够生成已知的、能够在那种浏览器上正确运行的 JavaScript 代码。如果服务器端的脚本检测到用户的浏览器不支持 JavaScript 代码, 那么它就会生成根本不需要 JavaScript 的网页。这种方法的一个重大缺陷是, 服务器端的脚本无法检测出用户何时禁用了浏览器对 JavaScript 的支持。

注意, 有关 CGI 的程序设计与服务器端的脚本的主题不在本书的讨论范围之内。

20.1.6 忽略问题

一个重要的问题是, 何时考虑“这种不兼容性的重要性”问题。如果这种不兼容性微不足道, 或者受其影响的浏览器或平台不常用, 又或者它只影响某种浏览器的过期版本, 那么就可以忽略这个问题, 把它留给那些受它影响的用户, 由那些用户来解决。

例如, 前面建议过定义一个 onmouseout 的事件处理程序来纠正 Netscape 2 和 3 的 Windows 版本不能正确清除状态栏的问题。遗憾的是, Netscape 2 不支持 onmouseout 事件处理程序, 因此上述的回避方法对 Windows 上的 Netscape 2 平台来说无效。如果你预计自己的应用程序会有许多使用 Windows 上的 Netscape 2 的用户, 而且认为将状态栏中的内容清除掉很重要, 所以必须采用其他回避方法。可以在事件处理程序 onmouseover 中使用函数 setTimeout(), 让它在两秒钟后清除状态栏中的内容。但是这种解决方案会带来新的问题, 即如果两秒钟后鼠标仍旧停留在超文本链接上, 此时不应该将状态栏中的内容清除掉, 又该怎么办? 在这种情况下, 一种较为简单的方法就是忽略这个问题。这样做很合理, 因为 Netscape 2 现在已经过期了, 所有仍旧使用它的用户都应该对它进行升级了。

20.1.7 适度停止运行

最后, 仍旧有一些不兼容性问题是不能忽略的, 而且也没有办法可以回避。在这种情况下, 你的程序就应该可以在所有提供了必需特性的平台、浏览器和版本中正确运行, 而在其他的情况下, 就适度地停止运行。所谓适度停止运行, 就是知道了必需的特性不可使用之后, 就通知用户, 不能使用你的 JavaScript 程序。

例如, 我们在第十四章中讨论过的图像置换方法在 Netscape 2 和 Internet Explorer 3 中不起作用, 而且没有一种回避方法可以用来模拟它。因此, 我们甚至不必尝试在上述两个平台上运行我们的程序。相反, 我们应该把这一不兼容性通报给用户。

实际上, 要做到适度停止运行比听起来要难得多。本章余下的大部分篇幅都用来解释这一方法。

20.2 语言版本的兼容性

前面几节讨论了通用的兼容性方法, 这些方法对于处理那些由运行在不同平台上的、由不同厂商制造的、不同版本的浏览器产生的不兼容性非常有用。本节要讨论的是另一种不兼容性关系, 即如何在不支持某种新的 JavaScript 语言特性的浏览器上使用这种特性, 而且又不会使浏览器出错。我们的目标非常简单, 即只需要阻止那些不能理解当前的 JavaScript 代码的浏览器对这些代码进行解释, 而且在浏览器中显示出特殊的消息, 通告用户他们的浏览器不能运行这些脚本。

20.2.1 language 性质

要实现第一个目标非常容易。我们在第十二章中看到过, 通过对标记 `<script>` 的 `language` 性质进行适当设置, 可以阻止浏览器运行它不能理解的代码。例如, 下面的 `<script>` 标记就指定了它包含的代码使用的是 JavaScript 1.1 的特性, 因此不支持这种脚本语言版本的浏览器就不会运行其中的代码:

```
<script language='JavaScript1.1'>  
    // 此处是 JavaScript 1.1 代码  
</script>
```

注意, `language` 性质使用的是一种通用的方法。当把它设置为 “JavaScript 1.2”

时，该性质就会阻止那些仅支持 JavaScript 1.0 或 1.1 的浏览器运行标记中的代码。在本书编写过程中，最新的浏览器（Netscape 6 和 IE 6）支持该语言的 1.0、1.1、1.2、1.3、1.4 和 1.5 版本。例如，如果你编写的 JavaScript 代码包括 try/catch 异常处理语句，那么应该在 <script> 中设置 language = "JavaScript 1.5"，以阻止不理解该语句的浏览器运行它。

但是，language 性质的通用性被一个事实破坏了，那就是把 language 设置成 "JavaScript 1.2" 会使 Netscape 的行为与 ECMA-262 标准不兼容。例如，我们在第五章中看到过，将 language 性质设置成 JavaScript 1.2 会使 "==" 运算符在执行相等性比较时不用做任何类型转换。我们在第八章中也看到过，上述设置会使方法 TOSTRING() 的行为完全不同。除非确定想使用这些新的、不兼容的行为，或者除非可以避免使用所有的不兼容特性，否则就应该避免使用 language = "JavaScript 1.2" 的设置。

注意，language 性质使用的版本号与 Netscape（现在是 Mozilla）的 JavaScript 解释器的版本号匹配。虽然 Microsoft 公司的解释器多少都有点跟随 Netscape 公司的发展，但 language 性质仍然与厂商有关，不同厂商对给定版本号的 language 特性支持不能保证一致。对 language = "JavaScript 1.2" 尤其是这样，不过其他版本也要注意。遗憾的是，不能用 language 性质指定标准的版本。也就是说，你不能编写如下代码：

```
<script language="ECMAScript3">...</script>
```

20.2.2 显式的版本测试

language 性质至少给语言版本的不兼容性问题提供了一种部分的解决方案，但是它也只是解决了一半问题。对于那些不支持想要使用的 JavaScript 版本的浏览器，我们还是需要适度停止运行。如果我们需要使用 JavaScript 1.1，那么我们最好能够通知仅支持 JavaScript 1.0 的浏览器的用户，告诉他们不能使用当前的页面。例 20-1 说明了如何实现这一点。

例 20-1：为那些不支持 JavaScript 1.1 的浏览器显示的消息

```
<!-- 设置一个变量以确定我们支持的 JavaScript 版本 -->  
<!-- 该方法可以扩展到任意语言版本号 -->  
<script language="JavaScript"> var _version = 1.0; </script>
```

```

<script language="JavaScript1.1"> _version = 1.1; </script>
<script language="JavaScript1.2"> _version = 1.2; </script>

<!-- 在启用 JavaScript 的浏览器上运行该代码 -->
<!-- 如果该版本不够高，显示一条消息 -->
<script language="JavaScript">
    if ( _version < 1.1 ) {
        document.write('<hr><h1>This Page Requires JavaScript 1.1</h1> ');
        document.write('Your JavaScript 1.0 browser cannot run this page.<h1> ');
    }
</script>

<!-- 下面是只在支持 JavaScript 1.1 的浏览器上运行的真实程序 -->
<script language="JavaScript1.1">
    // 此处是真正的 JavaScript 1.1 代码
</script>

```

20.2.3 避免与版本相关的错误

例 20-1 说明了如何编写仅支持 JavaScript 1.0 的浏览器所不能执行的 JavaScript 1.1 代码。那么，如何编写仅支持 JavaScript 1.1 的浏览器所不能执行的 JavaScript 1.2 的代码呢？虽然也可以把 language 性质明确地设置成“JavaScript 1.2”，但是前面我们讨论过，这会使 Netscape 表现出不兼容性。遗憾的是，JavaScript 1.2 还给语言添加了许多新语法。如果你编写的代码使用了 switch 语句、对象初始化程序或者函数直接量，然后在一个仅支持 JavaScript 1.1 的浏览器上运行这段代码，就会引发运行时的语法错误。

要避开这种问题，一个简单方法是减少在 JavaScript 1.1 的浏览器中发生的错误。例 20-2 说明了如何用 Window 对象的 onerror 错误处理函数（第十三章中介绍过）实现这一点。

例 20-2：避免与版本相关的错误

```

<!-- 检查是否支持 JavaScript -->
<script language="JavaScript1.2">var _js12_ = 1.2</script>

<!-- 下面避免在 Netscape 上运行 JavaScript 1.2 遇到的问题。 -->
<!-- 在支持 JavaScript 1.1 的浏览器上运行下面的代码。 -->
<!-- 如果该浏览器不支持 JavaScript 1.2，我们会显示错误消息。 -->
<!-- 抑制语法错误发生。 -->
<script language="JavaScript1.1">
    // 如果不支持 JavaScript 1.2，那么适度地停止运行。

    function suppressErrors() { return true; }

```

```
if (!_js12_) {
    window.onerror = suppressErrors;
    alert('This program requires a browser with JavaScript 1.2 support');
}
// 下面处理JavaScript 1.2代码。
</script>
```

20.2.4 为兼容性装载一个新页面

另一种实现版本兼容性的方法是，在确定浏览器是否提供对某种级别的 JavaScript 语言的支持后，装载一个使用这个级别的 JavaScript 语言的页面。例 20-3 说明了如何做到这一点，它用简单的脚本测试浏览器是否支持 JavaScript 1.2。如果浏览器支持这个版本，脚本就调用 `Location.replace()` 方法装载一个使用 JavaScript 1.2 的新页面。如果浏览器不支持 JavaScript 1.2，脚本将显示一条消息，说该页面要用 JavaScript 1.2。

例 20-3：检测 JavaScript 兼容性的页面

```
<head>
<script language="JavaScript1.2">
// 如果浏览器不支持JavaScript 1.2，从问号后的URL中提取一个新的URL，
// 并装载这个新的URL。
location.replace(location.search.substring(1));

// 输入一个长长的空循环，以便在装载新文档的过程中，不会显示文档的主体。
for(var i = 0; i < 10000000; i++);
</script>
</head>
<body>
<hr size="4">
<h1>This Page Requires JavaScript 1.2</h1>
Your browser cannot run this page. Please upgrade to a browser that
supports JavaScript 1.2, such as Netscape 4 or Internet Explorer 4.
<hr size="4">
</body>
```

这个例子最有趣的地方是，它是通用的，即要装载的 JavaScript 1.2 文件名编码在原始 URL 的检索部分，只在浏览器支持 JavaScript 1.2 时才装载该文件。因此，如果这个例子中的文件名为 `testjs12.html`，可以在超链接中使用如下 URL：

```
<a href="http://my.isp.net/~david/utills/testjs12.html?../js/cooljs12.html">
Visit my cool JavaScript 1.2 page!
</a>
```

关于例20-3, 需要注意的另一点是, 调用方法 `Location.replace()` 可以开始装载一个新页面, 但不会立刻停止装载当前页面。因此, 这个例子中的 JavaScript 代码在调用 `replace()` 方法后输入了一个长长的空循环。这可以阻止解析和显示余下的文档, 因此使用 JavaScript 1.2 浏览器的用户不会看到为不支持 JavaScript 1.2 的浏览器的用户准备的消息。

最后要注意, 例20-3中介绍的方法不仅可以区分 JavaScript 版本, 还可以区分是否支持 JavaScript 的浏览器。下一节将讨论其他兼容性方法, 它们也适用于非 JavaScript 浏览器。

20.3 非 JavaScript 浏览器的兼容性

前面几节讨论的都是不支持某种特殊 JavaScript 版本的浏览器的兼容性问题, 本节将要讨论根本不支持 JavaScript 的浏览器的兼容性问题。这些浏览器要么是不具有 JavaScript 的能力, 要么就是被用户禁用了 JavaScript (有些用户出于安全性考虑会这样做)。由于这样的浏览器仍旧大批存在, 所以应该仔细设计自己的网页, 以便在不能理解 JavaScript 的浏览器读取它后, 适度停止运行。要实现这一点有两部分事情要做, 一是要确保 JavaScript 代码不会被当作 HTML 文本显示出来, 二是要显示一条消息以通知访问者他的浏览器不能正确处理这个页面。

20.3.1 对老式浏览器隐藏脚本

支持 JavaScript 的 Web 浏览器都会执行出现在标记 `<script>` 和 `</script>` 之间的 JavaScript 语句。而不支持 JavaScript 但又能识别 `<script>` 标记的浏览器则会将 `<script>` 和 `</script>` 之间所有内容都忽略。这也正是它应该做的。但是老式浏览器 (目前仍然存在一些这样的浏览器) 甚至不能识别标记 `<script>` 和 `</script>`。这意味着它也会把标记本身也忽略掉, 然后把 `<script>` 和 `</script>` 之间的所有 JavaScript 代码都作为 HTML 文本显示出来。除非采取步骤制止它, 否则这些老式浏览器的用户就会看到 JavaScript 代码, 它们被安排在毫无意义的段落之中, 并显示为网页的内容。

为了防止发生这种情况, 应该将自己的脚本主体包括在一条 HTML 注释中, 使用的格式如例 20-4 所示。

例 20-4: 对老式浏览器隐藏的脚本

```
<script language="JavaScript">
<!-- 下面开始隐藏脚本的HTML注释
      // 此处是JavaScript语句
      .
      .
      .
      // 结束隐藏脚本的HTML注释 -->
</script>
```

不能理解标记<script>和</script>的浏览器就会忽略它们。所以例20-4中的第1行和第7行代码在那些浏览器中都没有作用。而且它们也会把第2行到第6行的代码忽略, 因为第2行的前四个字符是一条HTML注释的开始, 第6行的最后三个字符是注释的结尾, 注释之间的东西自然会被HTML解析器忽略。

脚本隐藏方法还适用于那些支持JavaScript的浏览器。第1行和第7行的代码分别标记出了一个脚本的开头和结尾。虽然客户端JavaScript解释器能够识别HTML注释的开始字符串“<!--”, 但是却会把它作为单行注释处理。因此, 支持JavaScript的浏览器就会将第2行代码作为单行注释处理。同样, 第6行代码也是以单行注释的字符串“//”开头, 所以启用JavaScript的浏览器也会把它忽略掉。这样就只剩下第3行到第5行的代码了, 它们将会被作为JavaScript语句执行。

虽然需要花费一点时间来习惯, 但是HTML注释与JavaScript注释的这种简洁的混合恰恰实现了我们需要的东西, 那就是它防止了在不支持JavaScript的浏览器中显示出JavaScript代码。尽管需要这种注释的浏览器越来越少了, 但是在Internet上的JavaScript代码中, 它仍然屡见不鲜。当然, 注释的作用不会都和例20-4相同。常见的还是如下所示的脚本:

```
<script language="JavaScript">
<!--
    document.write(new Date());
// -->
</script>
```

这种注释方法对不能运行JavaScript代码的浏览器隐藏了这些代码。适度停止作用的下一步就是给用户显示一条消息, 让他知道那个页面不能运行。

20.3.2 <noscript>

标记<noscript>和</noscript>括起了一个任意的HTML文本块, 那些不支持

JavaScript 的浏览器就应该显示这些文本。使用这两个标记可以让用户知道浏览器不能正确地显示页面了，因为它需要使用 JavaScript。例如：

```
<script language="JavaScript1.1">  
    // 此处是 JavaScript 代码  
</script>  
<noscript>  
<hr size="4">  
<h1>This Page Requires JavaScript 1.1</h1>  
This page requires a browser that supports JavaScript 1.1.<p>  
Your browser either does not support JavaScript, or it has JavaScript  
support disabled. If you want to correctly view this page, please  
upgrade your browser or enable JavaScript support.  
<hr size="4">  
</noscript>
```

使用 `<noscript>` 标记时有一个问题。它是由 Netscape 公司发布 Netscape3 时引入 HTML 的。因此，Netscape 2 不支持这一标记。由于 Netscape 2 不支持 `<noscript>` 和 `</noscript>`，所以它将忽略这对标记，只显示出它们之间包含的文本，即使它是支持 JavaScript 的。但是，在上面的代码中，结果正是我们想要的，因为我们指定了那些代码需要 JavaScript 1.1 的支持。

第二十一章

JavaScript 的安全性

由于Internet具有完全开放的性质，所以安全就成为一个非常重要的问题。引入Java与JavaScript这样的语言，尤其应该注意安全性，因为它们能够把可执行的内容嵌入静态网页中。由于装载网页可以使随机代码在你的计算机上运行，所以必须有严格的安全预防措施来防止恶意代码破坏数据或隐私。本章讨论了与JavaScript有关的Internet安全性问题。注意，本章并没有涉及到太多网络安全性的问题，诸如文档与HTML表单在Web上传输时用来保护隐私的验证与加密方法。

21.1 JavaScript 与安全性

JavaScript抵御恶意代码的第一道防线是它本身不支持某些功能。例如，客户端JavaScript没有提供在客户的计算机上写文件、删除文件或者创建目录、删除目录的方法。如果既没有File对象，又没有访问文件的函数，那么一个JavaScript程序就不能删除用户的数据，而且也不能在用户系统中种植病毒。

同样，客户端JavaScript没有任何类型的联网原语。JavaScript程序可以装载URL，可以把HTML表单发送给Web服务器、CGI脚本和电子邮件地址，但它却不能和Web上的其他任何一个主机建立直接连接。这就意味着一个JavaScript程序不能把一台客户机作为攻击平台，从该机解析另一台机器的命令（如果一个JavaScript程序已经穿过防火墙从Internet上装载了进来，并且试图侵入防火墙保护的内联网，那么这种攻击就变得危险了）。

尽管 JavaScript 核心语言和基本的客户端对象模型没有文件系统和联网功能（这些是恶意代码必须的），但情况仍然不像它表现的那么简单。在许多 Web 浏览器中，JavaScript 被用作其他软件部件的“脚本引擎”，如 Internet Explorer 中的 ActiveX 控件和 Netscape 中的插件。这些部件都具有文件系统和联网功能。JavaScript 程序能控制这些部件，增加了情况的复杂度，提高对安全性的要求。尤其是 ActiveX 控件，Microsoft 公司不时发布一些安全补丁，防止 JavaScript 程序采用脚本化的 ActiveX 对象的功能。我们将在本章结尾处再次简短介绍这一问题。

尽管客户端 JavaScript 故意省略了某些特性以提供基本的安全措施，防御恶意的攻击，但是其他安全问题仍然存在。这些问题主要是关于隐私的，在一条信息属于私有时，就不能允许 JavaScript 程序输出它的浏览器用户的信息。

在浏览 Web 时，用户默认允许发布的自身信息中包括所使用的 Web 浏览器的情况。作为 HTTP 协议的一个标准部分，每个发送给网页的请求中都会附加一个用于标识浏览器及其版本、厂商的字符串。这个信息是公开的，就像你的 IP 地址一样。但其他信息就不是公开性的了，其中包括你的电子邮件地址，除非你发送了一条电子邮件消息或者授权以你的名义自动发送电子邮件，从而选择了公开自己的邮件地址，否则它是不会被发布的。

同样，你的浏览历史（你已经访问过的站点记录）和书签列表的内容也应该保持其私有性。因为你的浏览历史与书签能够体现出你的兴趣，这种信息是营销人员和其他感兴趣的人要花钱猎取的，以便制定出更加有效的推销方式。可以肯定，如果 Web 浏览器或 JavaScript 允许这种有价值的信息被窃取，那么某些人会在你每次访问过他们的站点后窃取它，而且会以极快的速度把这些信息发布到市场上。大多数 Web 用户在知道他们访问的某个站点能够查出他们的某些喜好后都会觉得不舒服。

即使假定我们没有任何令人尴尬的兴趣要隐藏，还是有很多理由需要保护数据的隐私权。理由之一是害怕收到垃圾邮件。另一个理由则是对隐私权的保密。如果一个浏览器窗口中运行的 JavaScript 程序是从 Internet 装载进来的，而另一个浏览器窗口中的页面是从防火墙后的 Intranet 装载进来的，那么我们当然不想让第一个窗口中的 JavaScript 程序检测第二个窗口中的页面的内容。本章余下的部分解释了 JavaScript 如何防御这些攻击。

21.2 受限制的特性

我们已经提到过，客户端 JavaScript 防御恶意脚本的第一道防线是 JavaScript 语言去除了某些能力。第二道防线则是 JavaScript 为其支持的某些特性强加了一些限制。例如，客户端 JavaScript 支持 Window 对象的方法 `close()`，不过它对这一强加了一些限制，一个脚本只能用它关闭同一个 Web 服务器上的脚本打开的窗口。特别地，脚本不能关闭用户打开的窗口。如果它这样做了，就会出现一个确认框，询问他是否真的想关闭这个窗口。

最重要的安全限制是同源策略 (Same-Origin Policy)，下一节将介绍它。下面列出了大多数客户端 JavaScript 实现中使用的其他安全限制。这并不是一个十分准确的列表。每个浏览器采用的安全限制可能稍有不同，每个浏览器的专有特性可能具有专门的安全限制。

- History 对象原来是一个 URL 数组，表示浏览器的完整浏览历史。一旦它的私有性很明显，限制所有对 URL 的访问，History 对象就只剩下它的 `back()` 方法、`forward()` 方法和 `go()` 方法，可以在历史数组中移动，但不会暴露数组的内容。
- 不能设置 FileUpload 对象的 `value` 属性。如果设置了该属性，脚本就可以把它设为任意文件名，使表单把指定的文件的内容（如口令文件）上载到服务器上。
- 在没有用户（通过确认对话框）明确同意时，脚本不能把表单提交给 `mailto:` 或 `news:URL`（用 Form 对象的 `submit()` 方法）。这样的表单提交可能含有电子邮件地址，该地址在没有获得用户许可时不能把它公布于众。
- JavaScript 程序在没有用户许可时不能关闭浏览器窗口，除非是它自身打开的窗口。这阻止了恶意脚本调用 `self.close()` 方法关闭用户的浏览器窗口，因此使程序退出。
- 脚本不能打开边长小于 100 像素的窗口，也不能把窗口大小调整到边长小于 100 像素。同样，脚本不能把窗口移出屏幕，或创建比屏幕大的窗口。这可以防止脚本打开用户看不到或不容易注意到的窗口，这样的窗口含有的脚本可能在用户认为已经停止运行后仍在运行。此外，脚本不能创建没有标题栏的窗口，因为这样的窗口可以假冒操作系统对话框，欺骗用户输入区分大小的口令。

- 脚本不能创建显示 `about:URL` 的窗口或帧, 如 `about:cache`, 因为这些 URL 能暴露系统信息, 如浏览器缓存的内容。
- 脚本不能设置 Event 对象的任何属性。这可以防止脚本捏造事件。脚本不能给不同来源装载的脚本注册事件监听程序或捕捉事件。这可以防止脚本把用户的输入 (如构成口令的按键) 窃取到另一个页面。

21.3 同源策略

这是 JavaScript 中一条影响深远的安全限制, 值得我们用一节的篇幅来介绍它。这条限制称为“同源”策略, 即一个脚本只能读取与它同源 (如由同一个主机下载, 通过同一个端口下载或者下载协议相同) 的窗口或文档的属性。

实际上, 同源策略并非应用于不同源窗口中的所有对象的所有属性。不过它应用到了其中的大多数属性, 尤其是对 Document 对象的所有属性而言。出于各种目的, 我们都应该把所有客户端对象的所有预定义属性都看作对非同源脚本有限制。虽然不同的实现有不同的规定, 但非同源对象的用户定义的属性也可能受限制。

虽然同源策略是一种相当严格的限制, 但要防止不可靠的脚本窃取私有信息, 这一策略是必需的。如果没有这一制约, 窗口中的不可靠脚本 (可能是通过防火墙装载到一个安全协作的内联网上的浏览器中的脚本) 就可以用 DOM 方法读取另一个浏览器窗口中的内容, 而该窗口中可能会含有私有信息。

在某些环境中, 同源策略就显得太过严格了。它给那些使用多个服务器的大站点增加了许多麻烦。例如, 来自 `home.netscape.com` 的脚本可能会想要读从 `developer.netscape.com` 装载进来的文档的属性, 或者来自 `orders.acme.com` 的脚本可能需要读 `catalog.acme.com` 上的文档的属性, 这都是合理的。为了支持这种类型的大网站, JavaScript 1.1 引入了 Document 对象的属性 `domain`。在默认情况下, 属性 `domain` 存放的是装载文档的服务器的主机名。可以设置这一属性, 不过使用的字符串必须具有有效的域前缀。因此, 如果一个 `domain` 属性的初始值是“`home.netscape.com`”, 就可以把它设置成“`netscape.com`”, 但是不能把它设置成“`home.netscape`”或“`cape.com`”, 当然也不能设置成“`microsoft.com`” (`domain` 值中至少要有一个点号, 不能把它设为“`com`”或其他顶级域名)。

如果两个窗口（或框架）含有的脚本把 domain 设置成了相同的值，那么这两个窗口就不再受同源策略的约束，它们可以互相读取对方的属性。例如，从 *orders.acme.com* 和 *catalog.acme.com* 装载进来文档中的协作脚本可以把它们的 document.domain 属性都设置成“acme.com”，这样一来，这些文档就有了同源性，可以互相读取属性。

21.4 安全区和签名脚本

一种通用的安全策略决不会令人完全满意。如果这种策略太严格，可靠的脚本就没有我们想让它做的有趣的、有用的事情的能力。另一方面，如果策略太宽松，不可靠的脚本就会破坏系统。理想的解决方案是允许配置安全策略，使可靠的脚本受到的安全制约少于不可靠的脚本。两个主要的浏览器制造商 Microsoft 公司和 Netscape 公司都有不同的配置安全策略的方法，本节将进行简短的介绍。

Internet Explorer 定义了安全区，在这里你可以列出能够信任的 Web 站点和不能信任的 Web 站点。然后分别配置这两个安全区的安全策略，给予可靠的站点更多特权，给它较少的限制（对两个安全区中都没有明确列出的 Internet 站点和 Intranet 站点，也可以分别配置它们的特权）。

遗憾的是，这不是一种完整的或细粒度的 JavaScript 安全解决方案，因为 IE 允许你配置的大部分安全选项不与 JavaScript 直接相关。例如，在 IE 6 beta 版中，可以设置是否允许脚本控制 ActiveX 对象和 Java 小程序，是否能执行粘贴操作（即剪切并粘贴）。但没有选项可以供你为可靠站点取消同源策略或允许来自不可靠站点的脚本在没有用户确认的情况下发送 email 消息。

Netscape 6 和 Netscape 4 用“签名脚本”实现了可配置的安全策略。签名的脚本提供了完整的细粒度的安全策略配置方案，用一种加密的保护法和理论上非常有说服力的方法实现了它。遗憾的是，由于 Microsoft 公司没有与之兼容的方法，并且创建签名脚本的过程非常麻烦，而且终端用户不清楚签名脚本的用法，所以这种有希望的方法从没真正流行过。

简而言之，签名脚本有一个不会被遗忘的数字签名，声明编写该脚本或对该脚本负责的人或组织。当一个签名脚本需要避开前面介绍过的安全限制时，它首先要请求

一个特权，允许它这样做。在脚本请求一个特权时，浏览器将把它交由用户决定。用户将被告知签署脚本的人是谁，并被询问是否把请求的特权授予那个人或组织编写的脚本。一旦用户做出决定，可以使浏览器记住这一决定，以便将来浏览器不会询问同样的问题。事实上，这一过程允许用户在必要时配置细粒度的定制安全策略。

我们已经提到过，创建签名脚本的过程有点麻烦，而且 Netscape 4 和 Netscape 6 之间的创建细节已经改变了。这些细节不在本书介绍的范围之内，但你可以从 <http://developer.netscape.com/docs/manuals/signedobj/trust/index.htm> 和 <http://www.mozilla.org/projects/security/components/> 处了解到更多信息。

第二十二章

在 JavaScript 中使用 Java

我们在第十四章讨论过，Netscape 3 及其后的版本和 Internet Explorer 4 及其后的版本都允许 JavaScript 程序对嵌入 HTML 文档的 Java 小程序进行操作，既可以读写它们的公有字段，也可以调用它们的公有方法。Netscape 采用 LiveConnect 技术支持 JavaScript 和 Java 小程序之间的交互。Internet Explorer 将每个 Java 对象（包括小程序）作为一个 ActiveX 控件处理，并采用它的 ActiveX 脚本方法使 JavaScript 程序与 Java 进行交互。因为 Netscape 的方法是专为 JavaScript 程序和 Java 小程序之间的通信设计的，所以它具有一些 IE 的 ActiveX 技术不能提供的特性。但在实际应用中，这两种方法兼容。虽然本章以 Netscape 的 LiveConnect 方法为基础，但其关键特性在 IE 中也可以使用（注 1）。

本章先讨论了如何用 JavaScript 脚本化 Java 小程序，Java 小程序如何调用 JavaScript 代码，以及如何用 JavaScript 直接调用 Java 系统类（只在 Netscape 中）。然后本章介绍了 LiveConnect 方法运行的细节。要理解本章的内容，读者至少对 Java 有基本的了解（要学习 Java 的基本知识，可参阅《Java in a Nutshell》，David Flanagan 著和《Learning Java》，Patrick Niemeyer 和 Jonathan Knudsen 著，两书均由 O'Reilly 公司出版）。

注 1： 注意，Netscape 6 对 LiveConnect 的支持很少，Netscape 6.1 及以后的版本才完全实现了它。

22.1 脚本化 Java 小程序

我们在第十四章讨论过，嵌入 Web 页的 Java 小程序都属于 `Document.applets[]` 数组。如果一个小程序有给定的 `name` 或 `id`，可以直接把这个小程序作为 `Document` 对象的一个属性访问。例如，`<applet>` 标记创建的小程序的 `name` 性质值为“chart”，可以用 `document.chart` 引用它。

JavaScript 可以访问每个小程序的公有字段和方法，就像它们是 JavaScript 对象的属性和方法一样。例如，如果名为“chart”的小程序定义了名为 `lineColor` 的字段，类型为 `String`，那么 JavaScript 程序可以用下列代码查询并设置这个字段：

```
var chartcolor = document.chart.lineColor; // 读一个小程序的字段
document.chart.lineColor = "#ff00ff";      // 设置一个小程序的字段
```

JavaScript 设置可以查询并设置数组字段的值。假定 chart 小程序定义了两个字段，声明如下（Java 代码）：

```
public int numPoints;
public double[] points;
```

一个 JavaScript 程序可以用下列代码使用这两个字段：

```
for(var i = 0; i < document.chart.numPoints; i++)
    document.chart.points[i] = i*i;
```

这个例子说明了连接 JavaScript 和 Java 的麻烦之处，即进行类型转换。Java 是一种强类型语言，具有大量明确的原始类型。JavaScript 是一种松类型语言，只有一种数字类型。在上面的例子中，Java 整数被转换成 JavaScript 数字，各种 JavaScript 数字被转换成 Java 的 `double` 值。在后台需要进行大量的工作，确保根据需要正确地转换了这些值。在本章后面的小节中，我们将详细讨论类型转换这一主题。

除了能够查询和设置 Java 小程序的字段之外，JavaScript 还能调用小程序的方法。例如，假定 chart 小程序定义了名为 `redraw()` 的方法。这个方法没有参数，只用于通知小程序它的 `points[]` 数组改变了，应该刷新屏幕。JavaScript 可以像调用 JavaScript 方法那样调用这个方法：

```
document.chart.redraw();
```

JavaScript 还可以调用具有参数的方法并返回值。潜在的 LiveConnect 或 ActiveX 的脚本化方法将把 JavaScript 参数转换成合法的 Java 值，把 Java 的返回值转换成合法的 JavaScript 值。假定 chart 小程序定义了下列 Java 方法：

```
public void setDomain(double xmin, double xmax);  
public void setChartTitle(String title);  
public String getXAxisLabel();
```

JavaScript 可以用如下代码调用它们：

```
document.chart.setDomain(0, 20);  
document.chart.setChartTitle('y = x*x');  
var label = document.chart.getXAxisLabel();
```

最后要注意，Java 方法可以返回 Java 对象，作为它的返回值，JavaScript 可以读写这些对象的公有字段，调用它们的公有方法。JavaScript 还能把 Java 对象作为 Java 方法的参数。假定 Java 小程序定义了名为 getXAxis() 的方法，它返回一个 Java 对象，该对象是名为 Axis 的类的实例，另外还定义了名为 setYAxis() 的方法，它的参数类型与前者的返回值相同。现在，进一步假定 Axis 类有一个名为 setTitle() 的方法。我们可以用下面的 JavaScript 代码调用这些方法：

```
var xaxis = document.chart.getXAxis(); // 获取一个 Axis 对象  
var newyaxis = xaxis.clone(); // 复制该对象  
newyaxis.setTitle('Y'); // 调用它的一个方法  
document.chart.setYAxis(newyaxis); // 把它传递给另一个方法
```

在使用 JavaScript 调用一个 Java 对象的方法时，只有一点复杂之处。Java 允许两个或多个方法具有相同的名字，只要它们具有不同的参数类型。例如，Java 对象可以声明下面两个方法：

```
public String convert(int i); // 把一个整数转换成一个字符串  
public String convert(double d); // 转换一个浮点数字
```

JavaScript 只有一种数字类型，不能区分整数和浮点数，所以当用 JavaScript 给名为 convert 的方法传递数字时，它不能辨别想调用的是哪一个方法。在实践中，这个问题并不经常出现，根据需要给方法重命名，通常可以避免出现这个问题。LiveConnect 的最新版本（Netscape 6.1 及其后版本）还允许通过在方法名中加入参数类型来消除歧义。例如，如果上面的两个方法由 document.applets[0] 定义，可以用下列代码区分它们：


```
var iconvert = document.applets[0].convert(int)'; // 获取int方法  
iconvert(3); // 调用该方法
```

22.2 在 Java 中使用 JavaScript

前面已经探讨过如何在 JavaScript 代码中控制 Java 小程序，现在我们就讨论如何在 Java 代码中控制 JavaScript 代码。这种控制主要通过类 *netscape.javascript.JSObject* 实现，它表示 Java 程序中的一个 JavaScript 对象。前面一节介绍的 JavaScript 控制 Java 的功能在 Netscape 和 Internet Explorer 中通常都能运行得很好。相反，这里介绍的 Java 控制 JavaScript 的方法就没有那么强大的支持，它在 Netscape 和 IE 中都会遇到 bug。

22.2.1 JSObject 类

Java 与 JavaScript 的所有交互都由 *netscape.javascript.JSObject* 类的实例处理。该类的一个实例就是 JavaScript 对象的包装。该类定义的方法可以读写 JavaScript 对象的属性值与数组元素，并且调用 JavaScript 对象的方法。下面是这个类的摘要：

```
public final class JSObject extends Object {  
    // 为小程序的浏览器窗口获取初始 JSObject 对象的静态方法  
    public static JSObject getWindow(java.applet.Applet applet);  
    public Object getMember(String name);           // 读对象的属性  
    public Object getSlot(int index);               // 读数组的元素  
    public void setMember(String name, Object value); // 设置对象的属性  
    public void setSlot(int index, Object value);    // 设置数组的元素  
    public void removeMember(String name);          // 删除属性  
    public Object call(String methodName, Object args[]); // 调用方法  
    public Object eval(String str);                 // 计算字符串的值  
    public String toString();                        // 转换成字符串  
    protected void finalize();  
}
```

由于所有 JavaScript 对象都出现在当前浏览器窗口的根对象层次中，所以 JSObject 对象也必须出现在一个层次中。一个 Java 小程序要与所有 JavaScript 对象交互，必须首先含有一个表示浏览器窗口（或框架）的 JSObject 对象，小程序就显示在这个窗口（或框架）中。由于 JSObject 类没有定义构造函数，所以不能只创建一个 JSObject 对象。相反，必须调用静态方法 *getWindow()*。当传递给它一个对小程序的引用时，它就会返回一个 JSObject 对象，这个对象就表示含有那个小程序的浏览器窗口。因此，每个要与 JavaScript 交互的小程序都含有如下所示的一行代码：

```
JSObject jsroot = (JSObject) getWindow(this); // "this" 是小程序自身
```

获得了引用JavaScript对象层次的根对象的JSObject对象后,可以使用这个JSObject对象的实例方法来读取它所表示的JavaScript对象的属性值。由于这些属性中的大部分具有的值本身就是JavaScript对象,所以可以继续上述的处理来读它们的属性值。JSObject的方法getMember()返回的是一个命名属性的值,方法getSlot()返回的则是指定的JavaScript对象的一个数组元素。可以采用如下的代码来使用这些方法:

```
import netcape.javascript.JSObject; // 必须位于文件的开头  
...  
JSObject jsroot = JSObject.getWindow(this); // 自身  
JSObject document = (JSObject) jsroot.getMember("document"); // .document  
JSObject applets = (JSObject) document.getMember("applets"); // .applets  
Applet applet0 = (Applet) applets.getSlot(0); // [0]
```

使用这段代码时需要注意两点。第一,方法getMember()和getSlot()返回的是一个Object类型的值,一般必须把这个值转换成某种特定的类型,如JSObject对象。第二,从数组applets的位置0读到的值应被转换成一个Applet对象,而不是JSObject对象。这是因为JavaScript的applets[]数组中的元素是表示Java Applet对象的JavaObject对象。当Java读JavaScript的JavaObject对象时,它会打开那个对象,然后返回它所包含的Java对象(在本例中是一个Applet对象)。本章后面的小节介绍了通过JSObject接口发生的数据转换。

JSObject类还支持设置JavaScript对象的属性和数组元素的方法。方法setMember()和setSlot()与方法getMember()和getSlot()相似。这两个方法可以把一个命名属性或一个已编码的数组元素设置成一个指定的值。但需要注意的是,要设置的值必须是一个Java对象。如果要设置原始类型的值,必须使用相应的Java包装类。例如,用Integer对象代替int值。最后,用方法removeMember()可以从JavaScript对象中删除一个命名属性的值。

除了读写JavaScript对象的属性和数组元素之外,用JSObject类还可以调用JavaScript对象的方法。JSObject的call()方法就是调用一个指定的JavaScript对象的命名方法,传递给它的参数是一个Java对象的指定数组。我们知道,在设置JavaScript属性时,不必把原始的Java值作为参数传递给JavaScript方法,必须使用相应的Java对象类型来代替它。例如,在Java代码中,可以使用下面的代码调用方法call()来打开一个新的浏览器窗口:

```
public JSObject newwin(String url, String window_name)
{
    Object[] args = { url, window_name };
    JSObject win = JSObject.getWindow(this);
    return (JSObject) win.call("open", args);
}
```

JSObject 类还有一个更重要的方法 `eval()`。这个 Java 方法的作用与同名的 JavaScript 函数相同，即执行一个含有 JavaScript 代码的字符串。你会发现，使用 `eval()` 方法通常比使用 JSObject 类的其他方法都容易得多。因为所有的代码都作为字符串传递，所以可以使用你需要的数据类型的数据类型的字符串表示，不必再把 Java 的原始类型转换成它们相应的对象类型。例如，比较下面的两行代码，它们都用于设置浏览器主窗口的属性：

```
jsroot.setMember("i", new Integer(0));
jsroot.eval( "self.i = 0 );
```

第二行代码显然比第一行代码容易理解。考虑下面这个例子，它使用 `eval()` 方法编写一个要在浏览器窗口中显示的框架：

```
JSObject jsroot = JSObject.getWindow(this);
jsroot.eval("parent.frames[1].document.write('Hello from Java!')");
```

如果不用 `eval()` 方法，要实现与之等价的效果就困难得多了：

```
JSObject jsroot = JSObject.getWindow(this);
JSObject parent = (JSObject) jsroot.getMember("parent");
JSObject frames = (JSObject) parent.getMember("frames");
JSObject frame1 = (JSObject) frames.getSlot(1);
JSObject document = (JSObject) frame1.getMember("document");
Object[] args = { "Hello from Java!" };
document.call("write", args);
```

22.2.2 在小程序中使用 JSObject 对象

例 22-1 展示了一个小程序的 `init()` 方法，它使用 LiveConnect 来与 JavaScript 交互。

例 22-1：在小程序的方法中使用 JavaScript

```
import netscape.Javascript;

public void init()
{
```

```
// 获取表示小程序的浏览器窗口的 JSObject 对象。
JSObject win = JSObject.getWindow(this);

// 用 eval() 方法运行 JavaScript 代码。注意嵌套的引号。
win.eval("alert('The CPUlog applet is now running on your computer.  +
        'You may find that your system slows down a bit.');");
}
```

为了使用小程序，首先要对它进行编译，然后把它嵌入 HTML 文件。当小程序要与 JavaScript 进行交互时，上述的两步还需要特殊的指令。

22.2.2.1 编译使用 JSObject 类的小程序

所有与 JavaScript 交互的小程序都使用了类 *netscape.javascript.JSObject*。因此，要编译这样的小程序，Java 编译器必须知道从哪里找到这个类的定义。由于这个类是由 Netscape 公司定义并发布的，而不是由 Sun 公司定义的，所以 Sun 公司提供的 *javac* 编译器就识别不出它。本节解释了如何使编译器找到这个必需的类。如果你没有使用 Sun 提供的 JDK，那么只需要参阅你的 Java 编译器或 Java 开发环境的厂家提供的文档即可。

要告诉 JDK 的编译器在哪里可以找到类，需要设置环境变量 *CLASSPATH*。这个环境变量指定一个目录列表，其中有编译器应该检索的类定义（除了系统类的标准目录之外）的目录和 JAR 文件（或 ZIP 文件）。关键在于配置系统上的哪些 JAR 文件存放 *netscape.javascript.JSObject* 类的定义。在 Netscape 6.1 中，该文件位于 Netscape 的安装目录 *plugins/java2/javaplugin.jar* 中。在 Netscape 4 中，该文件位于 Netscape 的安装目录 *java/classes/java40.jar* 下。例如，对于 Windows 系统上的 Netscape 4 来说，可以在 *C:\Program Files\Netscape\Communicator\Program\Java\Classes\java40.jar* 处找到 *Java 40.Jar* 文件。

对于 Internet Explorer 来说，需要的类定义通常存放在 *c:\Windows\Java\Packages* 下的 ZIP 文件中。问题是该目录含有大量的 ZIP 文件，它们的名字非常杂乱，不同的版本有不同的名字。通常，最大的文件就是你需要的。可以用解压缩工具验证它是否含有文件 *netscape/javascript/JSObject.class*。

一旦找到了需要的 JAR 文件或 ZIP 文件，便可以通过设置环境变量 *CLASSPATH* 把它告诉编译器。对于 Unix 系统，可以用如下方法设置路径：

```
setenv CLASSPATH .:/usr/local/netscape/plugins/java2/javaplugin.jar
```

对于 Windows 系统，可以设置如下路径：

```
set CLASSPATH=.;C:\Windows\Java\packages\5rpnz7t.zip
```

设置了 CLASSPATH 后，便可以用 javac 编译小程序了。

22.2.2.2 mayscript 性质

要运行与 JavaScript 交互的 Java 小程序还需要一个属性。作为一种安全预防措施，除非网页作者（可能不是 Java 小程序的作者）明确地允许 Java 小程序使用其中的 JavaScript 代码，否则它不能这样做。要给予 Java 小程序这种许可，必须在 HTML 文件的小程序标记 `<applet>` 中添加一个新性质 `mayscript`。

例 22-1 展示过一段使用 JavaScript 来显示警报对话框的小程序。如果成功地编译了这个小程序，就可以使用如下代码把它添加到一个 HTML 文件中：

```
<applet code= J2UHog.class width="300" height="300" mayscript></applet>
```

如果忘记了使用性质 `mayscript`，这个小程序就不能使用 `JSObject` 类。

22.3 直接使用 Java 类

如前两节所述，Netscape 和 Internet Explorer 都允许 JavaScript 代码与 Java 小程序交互，也允许 Java 小程序与 JavaScript 交互。Netscape 的 LiveConnect 技术还允许 JavaScript 程序实例化自己的 Java 对象并使用它们（甚至可以不通过 Java 小程序）。Internet Explorer 不具备相似的能力。

在 Netscape 中，`Package` 对象可以访问 Netscape 知道的所有 Java 包。表达式 `Packages.java.lang` 引用 `java.lang` 包，表达式 `Packages.java.lang.System` 引用 `java.lang.System` 类。为了方便起见，`java` 是 `Packages.java` 的缩写。在 Netscape 中，JavaScript 代码可以用如下方法调用 `java.lang.System` 类的静态方法：

```
// 调用静态 Java 方法 System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

LiveConnect 的用途并不仅限于系统类，因为 LiveConnect 允许我们用 JavaScript 的 new 运算符创建新的 Java 类实例（像我们在 Java 中所做的一样）。例 22-2 展示的 JavaScript 代码使用了标准的 Java 类弹出一个窗口（事实上，这段 JavaScript 代码看来几乎与 Java 代码完全一样），显示某些文本。图 22-1 展示了该程序的结果。

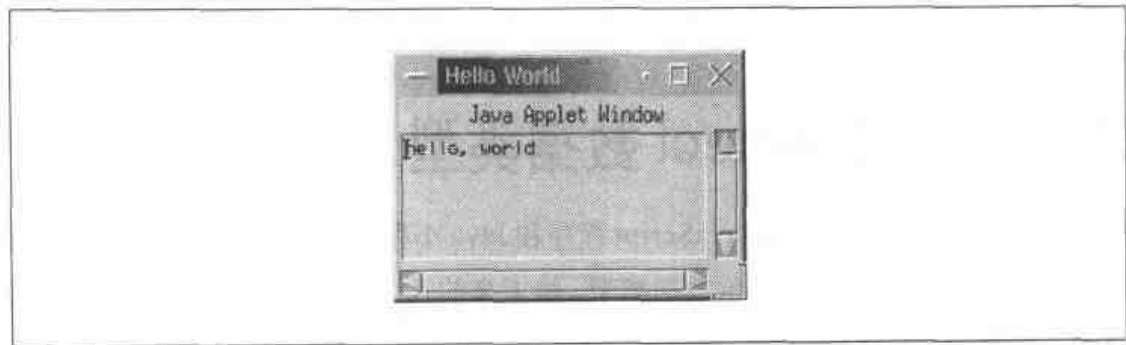


图 22-1: 在 JavaScript 中创建的 Java 窗口

例 22-2: 脚本化内部 Java 类

```
var f = new java.awt.Frame("Hello World");
var ta = new java.awt.TextArea("hello, world", 5, 20);
f.add("Center", ta);
f.pack();
f.show();
```

例 22-2 中的代码创建了一个简单的 Java 用户界面。但它没有任何形式的事件处理或用户交互。这样的程序只能进行输出，因为在用户与 Java 窗口交互时，它不包括任何通知 JavaScript 的方法。尽管用 JavaScript 定义能够响应事件的 Java 用户接口非常复杂，但是这是有可能的。在 Java 1.1 和其后的版本中，通过调用 `EventListener` 对象的方法可以执行事件通知。由于 Java 小程序可以执行任意的 JavaScript 代码串，所以可以定义一个 Java 类，实现适当的 `EventListener` 接口，并且在它被通知事件已经发生时，调用指定的 JavaScript 代码串。如果用创建 `EventListener` 对象的方法创建了一个小程序，就可以用 JavaScript 把用 JavaScript 定义的事件处理程序的 Java GUI 组合在一起。

注意，LiveConnect 没有对 Java 系统提供完全、无限制的访问。换句话说，有些事

用 LiveConnect 不能实现。例如，用 LiveConnect 不能在 JavaScript 中定义新的 Java 类或子类，也不能创建 Java 数组（注 2）。

除了这些限制外，出于安全的原因，还限制对标准 Java 类的访问。例如，不可靠的 JavaScript 程序不能使用 `java.io.File` 类，否则它便具有读、写和删除主机系统上的文件的权利。不可靠的 JavaScript 代码只能以不可靠的小程序使用 Java 的方法使用它。

22.4 LiveConnect 数据类型

要理解 LiveConnect 如何将 JavaScript 程序和 Java 小程序连接在一起，就必须了解 LiveConnect 使用的 JavaScript 数据类型。接下来的几节解释了这些 JavaScript 数据类型。虽然 Internet Explorer 使用的技术不同，但理解 LiveConnect 如何运行，也有助于理解 IE 的运行方式。这里介绍的某些数据类型在 IE 中有相似的类型。

22.4.1 JavaPackage 类

Java 中的包（package）是一组相关的 Java 类的集合。JavaPackage 类是表示 Java 包的一种 JavaScript 数据类型，它的属性是 Java 包中含有的类（不久我们就会知道，包中的 Java 类由 JavaClass 类表示）以及包含的其他包。关于 JavaPackage 类有一条约定，即不能使用 JavaScript 的 `for/in` 循环来获取一个 JavaPackage 包含的所有包和类的完整列表。这是 Java 虚拟机的基本约定产生的结果。

所有的 JavaPackage 对象都包含在一个父 JavaPackage 对象中，Window 对象中名为 `Packages` 的属性引用了包层次的根的顶级 JavaPackage 对象。它的属性有 `java`、`sun` 和 `netscape`，它们都是 JavaPackage 对象，分别代表浏览器可以使用的各种 Java 类层次。例如，JavaPackage 对象 `Packages.java` 包含的是 JavaPackage 对象 `Packages.java.awt`。为了方便起见，每个 Window 对象都还有 `java`、`sun` 和 `netscape` 属性，它们就是引用 `Packages.java`、`Packages.sun` 和 `Packages.netscape` 的快捷方式。因此，可以只键入 `java.awt`，而不必再输入 `Packages.java.awt`。

注 2： 用 Java 1.1 的 `java.lang.reflect.Array.newInstance()` 方法可以在 JavaScript 程序中间接地创建数组。

继续使用上面的例子, `java.awt` 是一个含有 `JavaClass` 对象的 `JavaPackage` 对象, 例如 `java.awt.Button` 就是 `java.awt` 含有的一个 `JavaClass` 对象, 表示 `java.awt.Button` 类。除此之外, `java.awt` 还含有另一个 `JavaPackage` 对象 `java.awt.image`, 这个 `JavaPackage` 对象表示 Java 中的 `java.awt.image` 包。

你会发现, `JavaPackage` 对象层次的属性命名模式镜像出了 Java 包的命名模式。但要注意, `JavaPackage` 类和它所表示的真正 Java 包之间有一点重大区别, 即 Java 中的包是类的集合, 而不是其他包的集合。简而言之, `java.lang` 是一个 Java 包的名字, 但 `java` 不是。因此, 名为 `java` 的 `JavaPackage` 对象表示的不是 Java 包, 只是包层次中的一个占位符, 替代那些表示 Java 包的 `JavaPackage` 对象。

在大多数系统中, Java 类都安装在与它们所属的包的名字相应的目录层次下。例如, 类 `java.lang.String` 存放在文件 `java/lang/String.class` 中。实际上, 这个文件通常存放在一个 ZIP 文件中, 但目录层次仍旧存在, 在文档中编码。因此, 与其认为 `JavaPackage` 对象表示 Java 包, 不如认为它代表的是 Java 类的目录层次中的一个目录或子目录。

`JavaPackage` 类几个缺点。第一, `LiveConnect` 不能预先辨别出 `JavaPackage` 对象的属性引用的是 Java 类还是另一个 Java 包, 因此 `JavaScript` 假定它引用的是一个类, 并且要尝试装载这个类。所以, 当使用诸如 `java.awt` 这样的表达式时, `LiveConnect` 会首先检索类文件 `java/awt.class`。它甚至还会通过网络检索这个类, 这样就会使网络服务器把一个 “404 File Not Found” 的错误写入日志。如果 `LiveConnect` 没有找到这个类, 它就会假定这个属性引用的是一个包, 不过它不能确定那个包是否真正存在, 包中是否真的含有类。这就引发了第二个缺陷, 即如果拼错了类名, `LiveConnect` 就会胡乱地把它作为包的名字来处理, 而不是通知你要使用的类不存在。

22.4.2 JavaClass 类

`JavaClass` 类是 `JavaScript` 的一种数据类型, 它表示一个 Java 类。一个 `JavaClass` 对象没有任何属于它自己的属性, 它所有的属性都表示它所代表的 Java 类的公有静态字段和方法。有时, 这些公有的静态字段和方法被称为类字段 (class field) 和类方法 (class method), 说明它们与类相关, 而不是与对象实例相关。与 `JavaPackage`

类不同, `JavaClass` 类允许使用循环 `for/in` 来枚举它的属性。注意, `JavaClass` 对象没有表示 Java 类的实例字段和方法的属性, 单独的 Java 类实例由 `JavaObject` 类表示, 不久我们就会对它进行说明。

我们已经知道, `JavaClass` 对象包含在 `JavaPackage` 对象中。例如, `java.lang` 是一个 `JavaPackage` 对象, 它含有 `System` 属性, 所以 `java.lang.System` 就是一个 `JavaClass` 对象, 表示 Java 类 `java.lang.System`。 `JavaClass` 对象还含有诸如 `out` 和 `in` 这样的属性, 这些属性代表类 `java.lang.System` 的静态字段。可以采用这种方式来引用标准的 Java 系统类。例如 `java.lang.Double` (或 `Package.java.lang.Double`) 命名为 `java.lang.Double`, `java.awt.Button` 类命名为 `java.awt.Button`。

另一种使用 JavaScript 来获取 `JavaClass` 对象的方法是使用函数 `getClass()`。给定任何一个 `JavaObject` 对象, 都可以获得表示那个 Java 对象的类的 `JavaClass` 对象, 只需要把 `JavaObject` 对象传递给函数 `getClass()` 即可 (注 3)。

一旦有了 `JavaClass` 对象, 可以用它来实现几种功能。 `JavaClass` 类实现了 `LiveConnect` 的功能, 这种功能使 JavaScript 程序能够对 Java 类的公有静态字段进行读写, 并且调用它的公有静态方法。例如, `java.lang.System` 是一个 `JavaClass` 类, 可以采用如下的表达式来读取 `java.lang.System` 的静态字段的值:

```
var java_console = java.lang.System.out;
```

同样, 可以采用下面的一行代码来调用 `java.lang.System` 的静态方法:

```
var java_version = java.lang.System.getProperty('java.version');
```

前面说过, Java 是一种强类型语言, 也就是说它的所有字段以及所有方法的参数都要具有类型。如果把字段设置成一种错误的类型或者传递给方法的参数的是类型错误, 就会抛出异常 (在 JavaScript 1.5 之前的版本中, 则发生 JavaScript 错误。)

`JavaClass` 类还有一个重要的特性, 即可以用带有 `new` 运算符的 `JavaClass` 对象创建 Java 类的一个新实例, 例如创建 `JavaObject` 对象。实现这个操作的语法和 JavaScript 的语法一样 (也和 Java 的语法一样):

注 3: 不要将 JavaScript 的 `getClass` 函数和 Java 方法 `getClass` 函数混为一谈。前者返回一个 `JavaClass` 对象, 而后者返回一个 `java.lang.Class` 对象。

```
var d = new java.lang.Double(1.23);
```

用这种方法创建了 `JavaObject` 对象后，我们返回 `getClass()` 函数，展示一个使用它的例子：

```
var d = new java.lang.Double(1.23); // 创建一个JavaObject对象
var d_class = getClass(d);           // 获取JavaObject的JavaClass对象
if (d_class == java.lang.Double) ...; // 该比较结果为true
```

当用这种方法使用标准的系统类时，通常可以直接使用系统类的名字，而不必调用函数 `getClass()`。这个函数在获取非系统对象（如一个小程序实例）的类时更有用。

可以定义一个变量作为引用 `JavaClass` 对象的快捷方式，这样就不必使用像 `java.lang.Double` 这样冗长的表达式了：

```
var Double = java.lang.Double;
```

上述语句模拟了 Java 的 `import` 语句，而且可以提高程序的效率，因为 `LiveConnect` 不必再检索 `java` 的 `lang` 属性和 `java.lang` 的属性 `Double` 了。

22.4.3 JavaObject 类

`JavaObject` 类是 JavaScript 的一种数据类型，表示一个 Java 对象。`JavaObject` 类在许多方面都与 `JavaClass` 类相似。与 `JavaClass` 类一样，`JavaObject` 类没有自己的属性，它的所有属性表示的（而且名字也相同）都是它所代表的 Java 对象的公有实例字段和公有实例方法。另外，也可以用 JavaScript 循环 `for/in` 来枚举一个 `JavaObject` 对象的所有属性。`JavaObject` 对象类也实现了 `LiveConnect` 功能，该功能使我们可以读写一个 Java 对象的公有实例字段，并且可以调用它的公有方法。

例如，如果 `d` 是一个 `JavaObject` 对象，它表示类 `java.lang.Double` 的一个实例，则可以使用如下的 JavaScript 代码来调用那个 Java 对象的方法：

```
n = d.doubleValue();
```

同样，我们在前面看到过，`java.lang.System` 类有静态字段 `out`，该字段引用 `java.io.PrintStream` 类的一个 Java 对象。在 JavaScript 中，可以使用下面的表达式引用相应的 `JavaObject` 对象：

```
java.lang.System.out
```

而且还可以用下面所示的方法调用这个方法（注4）：

```
java.lang.System.out.println("Hello world! ");
```

用 `JavaObject` 对象还能读写它所代表的 Java 对象的公有实例字段。不过前面例子中使用的 `java.io.PrintStream` 类和 `java.lang.Double` 类都没有任何公有实例字段。假定用 JavaScript 创建了一个 `java.awt.Rectangle` 类的实例：

```
r = new java.awt.Rectangle();
```

那么就可以使用如下的 JavaScript 代码来读写它的公有实例字段：

```
r.x = r.y = 0;
r.width = 4;
r.height = 5;
var perimeter = 2*r.width + 2*r.height;
```

LiveConnect 的优点在于，它使用 Java 对象 `r` 时就像在使用一个 JavaScript 对象。但是还有一点需要注意，即 `r` 是一个 `JavaObject` 对象，它的行为与正规的 JavaScript 对象不完全相同。至于其间的差别，后面我们将做详细介绍。另外，还要记住，与 JavaScript 不同的是，Java 对象的各个字段和它的方法的参数都有类型。如果给它们设置的 JavaScript 值类型不正确，就会引发一个 JavaScript 错误或异常。

在 Netscape 6.1 和其后的版本中，可以使用名字或名字加参数类型的方法访问 `JavaObject` 类，当两个或多个方法具有相同的名字，但参数类型不同时，这种方法非常有用。在本章前面我们见到过，如果 `JavaObject` 对象 `o` 表示具有两个名为“convert”的方法的对象，那么 `o` 的 `convert` 属性可以引用这两个方法中的任何一个。但在 LiveConnect 的最新版本中，`o` 还定义了包括参数类型的属性，可以通过添加类型信息指定你想调用的方法：

```
var iconvert = o['convert(int)']; // 获取我们想要的方法
iconvert(3);                      // 调用它
```

因为属性名包括括号，所以不能用正则符号“.”访问它，必须把它表示为带方括号的字符串。`JavaClass` 类型的功能与被覆盖的静态方法相同。

注4： 这行代码的输出结果不会显示在浏览器中，而是显示在 Java 控制台中。在 Netscape 6 中，可以选择 **Tasks → Tools → Java Console** 来查看这个窗口。

22.4.4 JavaArray 类

JavaScript 最后一种用于 LiveConnect 的数据类型是 JavaArray 类。也许你已经料到了，这个类的实例表示的是 Java 数组，而且提供了 LiveConnect 的功能，允许 JavaScript 读取这个 Java 数组的元素。与 JavaScript 数组一样（也与 Java 数组一样），JavaArray 对象具有 length 属性，它指定了该对象含有的元素个数。读取 JavaArray 对象的元素时，可以使用标准 JavaScript 数组的下标运算符[]。此外，也可以使用 JavaScript 循环 for/in 将这些元素枚举出来。还可以用 JavaArray 对象访问多维数组（其实是元素为数组的数组），就像在 JavaScript 中或 Java 中进行的操作一样。

例如，假定创建了 *java.awt.Polygon* 类的一个实例：

```
p = new java.awt.Polygon();
```

JavaObject 对象 p 具有属性 xpoints 和 ypoints，它们都是 JavaArray 对象，表示整型的 Java 数组（要了解这些属性的名字和类型，请参阅 Java 参考手册中对 *java.awt.Polygon* 的说明）。可以使用这些 JavaArray 对象随机地初始化一个 Java 多边形，代码如下：

```
for(var i = 0; i < p.xpoints.length; i++)  
    p.xpoints[i] = Math.round(Math.random()*100);  
for(var i = 0; i < p.ypoints.length; i++)  
    p.ypoints[i] = Math.round(Math.random()*100);
```

22.4.5 Java 方法

JavaClass 和 JavaObject 类分别允许我们调用 Java 的静态方法和实例方法。在 Netscape 3 中，Java 方法是用 JavaMethod 对象在内部表示的。但是在 Netscape 4 中，Java 方法只是本地的方法，像 String 和 Date 这样的内部 JavaScript 对象的方法一样。

在使用 Java 方法时，要记住，它们期望得到的是数量和类型都固定的参数。如果传递给它们的参数个数不对，或者传递给它们的参数类型不对，就会引发一个 JavaScript 错误。

22.5 LiveConnect 数据转换

Java 是一种强类型的语言，具有相对较多的数据类型，而 JavaScript 是一种无类型语言，具有的数据类型相对较少。由于两种语言存在这种主要的结构性差别，所以 LiveConnect 的核心任务之一就是转换数据。当用 JavaScript 设置一个 Java 类、一个实例字段，或者给一个 Java 方法传递参数时，必须把 JavaScript 值转换成等价的 Java 值。当用 JavaScript 读一个 Java 类、一个实例字段，或者获取一个 Java 方法的返回值时，那个获得的 Java 值也必须被转换成兼容的 JavaScript 值（注 5）。

图 22-2 和图 22-3 分别说明了在 JavaScript 写 / 读 Java 值时如何执行数据转换。

有关图 22-2 说明的数据转换，需要注意以下几点：

- 图 22-2 没有展示 JavaScript 类型与 Java 类型之间所有可能的转换。这是因为在 JavaScript 到 Java 的转换发生之前，可能会进行 JavaScript 到 JavaScript 的类型转换。例如，如果把一个 JavaScript 数字传递给了一个 Java 方法，而这个方法希望得到的是 *java.lang.String* 类的参数，那么 JavaScript 首先会将这个参数转换成一个 JavaScript 字符串，然后再将它转换成一个 Java 字符串。
- JavaScript 数字可以被转换成任意一种原始的 Java 数字类型。当然，真正的转换要根据设置的 Java 字段或要传递的方法参数的类型进行。注意，这样做会失去精确性，例如，把一个很大的数字传递给一个 *short* 型的 Java 字段，或者把一个浮点值传递给一个整型的 Java 字段时就会出现这种情况。
- JavaScript 数字还可以转换为 Java 类 *java.lang.Double* 的实例，但是却不能转换成一个相关类（如 *java.lang.Integer* 或 *java.lang.Float*）的实例。
- 因为 JavaScript 没有字符数据的表示方法，所以 JavaScript 数字可以转换成 Java 的原始类型 *char*。

注 5： 另外，当 Java 读写一个 JavaScript 域或者调用一个 JavaScript 方法时，也必须进行数据转换。但是进行这些转换的方式不同，在本章后面的小节中，当我们讨论如何在 Java 中使用 JavaScript 时再详细讨论这些转换。现在，我们只考虑 JavaScript 代码与 Java 代码进行交互时发生的数据转换。

- JavaScript 中的 `JavaScript` 对象在传递给 Java 时是“打开”的，它将转换成它所表示的 Java 对象。但要注意，JavaScript 中的 `JavaScript` 对象不会转换成 `java.lang.Class` 类的实例。
- JavaScript 数组不会转换成 Java 数组。

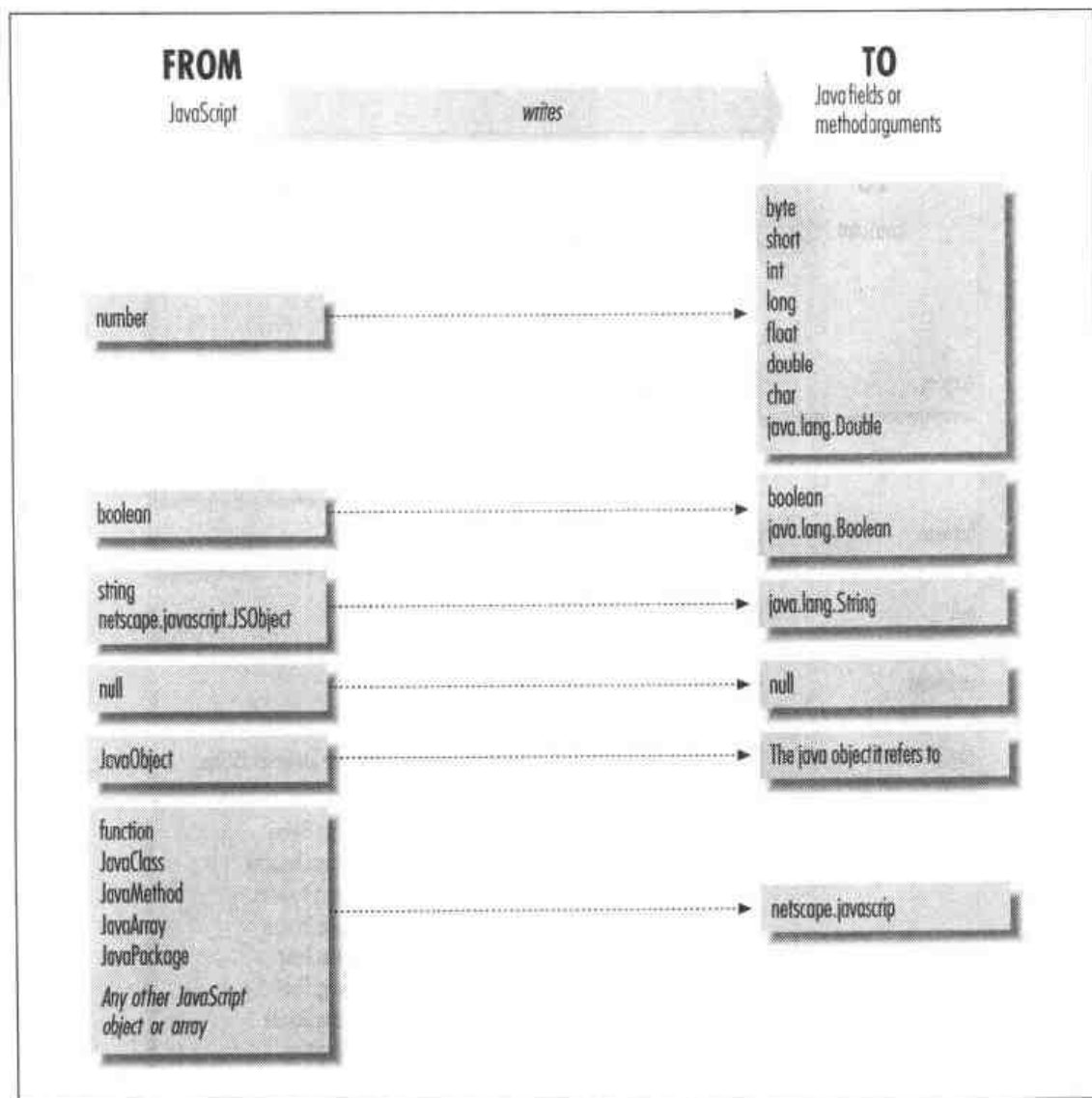


图 22-2: 当 JavaScript 写 Java 值时执行的数据转换

有关图 22-3 说明的数据转换，需要注意以下几点：

由于 JavaScript 没有字符数据类型，所以 Java 的基本数据类型 `char` 将转换成 JavaScript 数字，而不是像预计的那样转换成一个字符串。

类 `java.lang.Double` 和类 `java.lang.Integer` 以及其他相似的类的 Java 实例不是转换成 JavaScript 数字，而是像所有的 Java 对象一样，转换成 JavaScript 中的一个 `JavaScript` 对象。

Java 字符串是类 `java.lang.String` 的一个实例，因此与其他 Java 对象一样，它会转换成一个 `JavaScript` 对象，而不是转换成一个真正的 JavaScript 字符串。

任何类型的 Java 数组都可以转换成 JavaScript 的 `JavaScript` 对象。

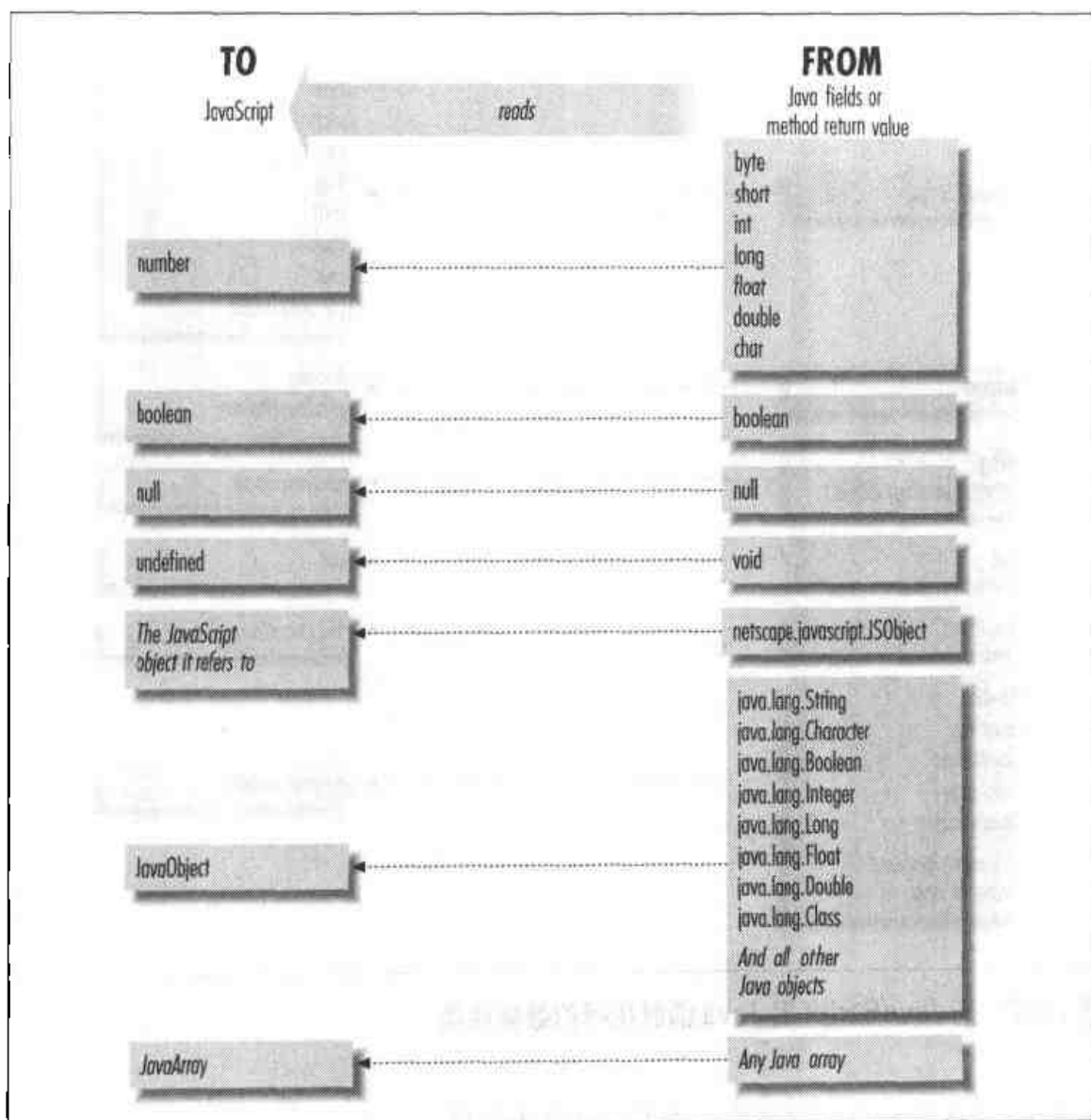


图 22-3: 当 JavaScript 读 Java 值时执行的数据转换

22.5.1 包装对象

要完全理解图 22-2 和图 22-3, 必须掌握的另一个重要概念是“包装对象”。尽管大部分 JavaScript 原始类型和 Java 原始类型之间的转换都可能实现, 但是, 一般说来, 两种对象类型之间的转换却不可能实现。这就是 LiveConnect 在 JavaScript 中定义 `JavaObject` 对象的目的, `JavaObject` 对象代表了不能直接转换成 JavaScript 对象的 Java 对象。从某种意义上说, `JavaObject` 对象是 Java 对象上的一层 JavaScript 包装。当 JavaScript 读一个 Java 值 (一个字段或一个方法的返回值) 时, 所有的 Java 对象都被包装了起来, JavaScript 能看到的只是一个 `JavaObject` 对象。

当 JavaScript 将一个 JavaScript 对象写入一个 Java 字段时, 或者当 JavaScript 将一个 JavaScript 对象传递给一个 Java 方法时, 同样的事情就会发生。由于不能把 JavaScript 对象转换成 Java 对象, 所以对象就被包装了起来。JavaScript 对象的 Java 包装是 Java 类 `netscape.javascript.JSObject`。

当要将这些包装传递回来时, 事情就变得有趣得多了。如果 JavaScript 将一个 `JavaObject` 对象写入一个 Java 字段或者将它传递给一个 Java 方法, LiveConnect 首先会打开对象的包装, 把那个 `JavaObject` 对象转换成它表示的 Java 对象。同样, 如果 JavaScript 读取了一个 Java 字段或者得到了一个 Java 方法的返回值, 而且这个值是类 `netscape.javascript.JSObject` 的一个实例, 那么 `JSObject` 对象也会被打开, 从而返回原始的 JavaScript 对象。

22.5.2 Netscape 3 中的 LiveConnect 数据转换

在 Netscape 3 中, LiveConnect 把 Java 值转换成 JavaScript 值的方法中存在一个 bug, 即 Java 对象的原始字段的值将转换成一个 JavaScript 对象, 而不是被转换成一个 JavaScript 的原始值。例如, 如果 JavaScript 读取一个 `int` 型 Java 字段的值, Netscape 3 中的 LiveConnect 会把那个值转换成 `Number` 对象, 而不是转换成一个原始的数字值。同样, LiveConnect 也会将 Java 的 `boolean` 字段的值转换成 JavaScript 的 `Boolean` 对象, 而不是转换成原始的 JavaScript 布尔值。注意, 这个 bug 只在查询 Java 字段的值时才会发生, 在 LiveConnect 转换一个 Java 方法的返回值时不会发生。

JavaScript 的 Number 对象和 Boolean 对象的行为与原始数字值和布尔值非常近似，但又不完全相同。一个重要的差别是 Number 对象把 “+” 号作为字符串连接的运算符，而不是作为加法运算符，这与所有 JavaScript 对象一样。由于上述原因，下面列出的代码使用了 Netscape 3 中的 LiveConnect，它将会产生出人意料的结果：

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width;      // 这是一个 Number 对象，不是原始数字。
var new_w = w + 1;    // new_w 现在是 "51"，不是预计的值 6。
```

要避免这一问题，可以明确地调用方法 `valueOf()`，把一个 Number 对象转换成与它相应的数字值。例如：

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width.valueOf(); // 现在我们得到了原始数字。
var new_w = w + 1;        // 这次 new_w 的值是预计的 6。
```

22.6 JavaObject 对象在 JavaScript 中的转换

前面集中介绍了数据转换，也许你希望能实现一些数据转换。遗憾的是，有关 JavaScript 如何把 JavaObject 对象转换成各种的 JavaScript 原始类型的这一主题还有许多内容需要讨论。注意图 22-3，其中相当多的 Java 数据类型（包括 Java 字符串，即 `java.lang.String` 类的实例）在 JavaScript 中都被转换成了 JavaObject 对象，而不是转换成真正的 JavaScript 原始类型（如字符串）。这意味着在使用 LiveConnect 时，通常都是使用 JavaObject 对象。

让我们再回来看一下表 11-1，它说明在不同的环境中使用时，各个 JavaScript 数据类型是如何被转换的。例如，在字符串环境中使用一个数字时，它就会被转换成一个字符串。在布尔环境中使用一个对象时，它就会被转换成值 `false`（如果它为 `null`），否则便转换为 `true`。这些转换规则并不适用于 JavaObject 对象，它们的转换使用的是自己特有的规则：

- 在一个数字环境中使用 JavaObject 对象时，通过调用这个对象所代表的 Java 对象的方法 `doubleValue()`，可以把它转换成一个数字。如果那个 Java 对象没有定义方法 `doubleValue()`，就会引发一个 JavaScript 错误。

- 在一个布尔环境中使用 `JavaObject` 对象时, 通过调用这个对象所代表的 `Java` 对象的方法 `booleanValue()`, 可以把它转换成一个布尔值。如果那个 `Java` 对象没有定义方法 `booleanValue()`, 就会引发一个 `JavaScript` 错误。
- 在一个字符串环境中使用 `JavaObject` 对象时, 通过调用这个对象所代表的 `Java` 对象的方法 `toString()`, 可以把它转换成一个字符串。由于所有的 `Java` 对象都定义或继承了方法 `toString()`, 所以这种转换一定会成功。
- 在一个对象环境中使用 `JavaObject` 时, 没有必要进行任何转换, 因为它本身就是一个 `JavaScript` 对象。

由于转换规则之间存在的差别以及其他的一些原因, `JavaObject` 对象的行为与其他 `JavaScript` 对象有很大的不同, 其间有一些易犯的错误, 你应该格外注意。首先, 表示类 `java.lang.Double` 或其他数字对象的 `JavaObject` 对象很常见。这样一个 `JavaObject` 对象在许多方面都与原始数字值一样, 不过在使用 “+” 运算符时多加注意, 这时设置了一个字符串环境, 所以 `JavaObject` 对象 (以及其他所有的 `JavaScript` 对象) 会被转换成一个字符串, 以便进行字符串的连接运算, 而不是转换成一个数字, 以便用于加法运算。

如果想明确地把一个 `JavaScript` 对象转换成一个原始值, 通常可以调用这个对象的 `valueOf()` 方法。注意, 该方法不适用于 `JavaObject` 对象。前面我们已经讨论过, `JavaObject` 类没有定义任何属于自己的属性, 它的所有属性都表示它所代表的 `Java` 对象的字段和方法。这意味着 `JavaObject` 对象不支持通用的 `JavaScript` 方法, 如 `valueOf()` 等。对于用 `JavaObject` 对象包装的 `java.lang.Double` 对象, 当必须将它转换成一个原始值时, 应该调用 `Java` 方法 `doubleValue()`。

`JavaObject` 对象与 `JavaScript` 其他的数据类型之间的另一个区别是, 如果 `JavaObject` 对象定义了方法 `booleanValue()`, 就只能在一个布尔环境中使用它。假定 `button` 是一个 `JavaScript` 变量, 它的值可以是 `null`, 也可以是一个表示 `java.awt.Button` 类的实例的 `JavaObject` 对象。如果想检测这个变量的值是否为 `null`, 可以使用如下的代码:

```
if (!button) { ... }
```

如果 `button` 值为 `null`, 这行代码就会正常运行。但是如果 `button` 的值是一个表示 `java.awt.Button` 实例的 `JavaObject` 对象, `LiveConnect` 就会调用方法 `boolean-`

Value()。当它发现类`java.awt.Button`没有定义这样一个对象时，它就会引发一个JavaScript错误。避免这种问题的方法是明确地指出要测试的是什么，这样就避免了在一个布尔环境中使用`JavaObject`对象：

```
if (button != null) { ... }
```

无论如何，这都是一个好习惯，因为它可以使代码更易读，也更容易理解。

22.7 从 Java 到 JavaScript 的数据转换

在前两节中，我们讨论了当JavaScript读写Java的字段或调用Java方法时进行数据转换的规则。这些规则解释了JavaScript的`JavaObject`对象、`JavaArray`对象和`JavaClass`对象转换数据所使用的方式，它们仅适用于JavaScript代码操作Java的情况。当Java程序要操作JavaScript时，数据转换是由Java的`JSObject`类执行的，而且转换的规则也有所不同。图22-4和图22-5说明了这些转换。

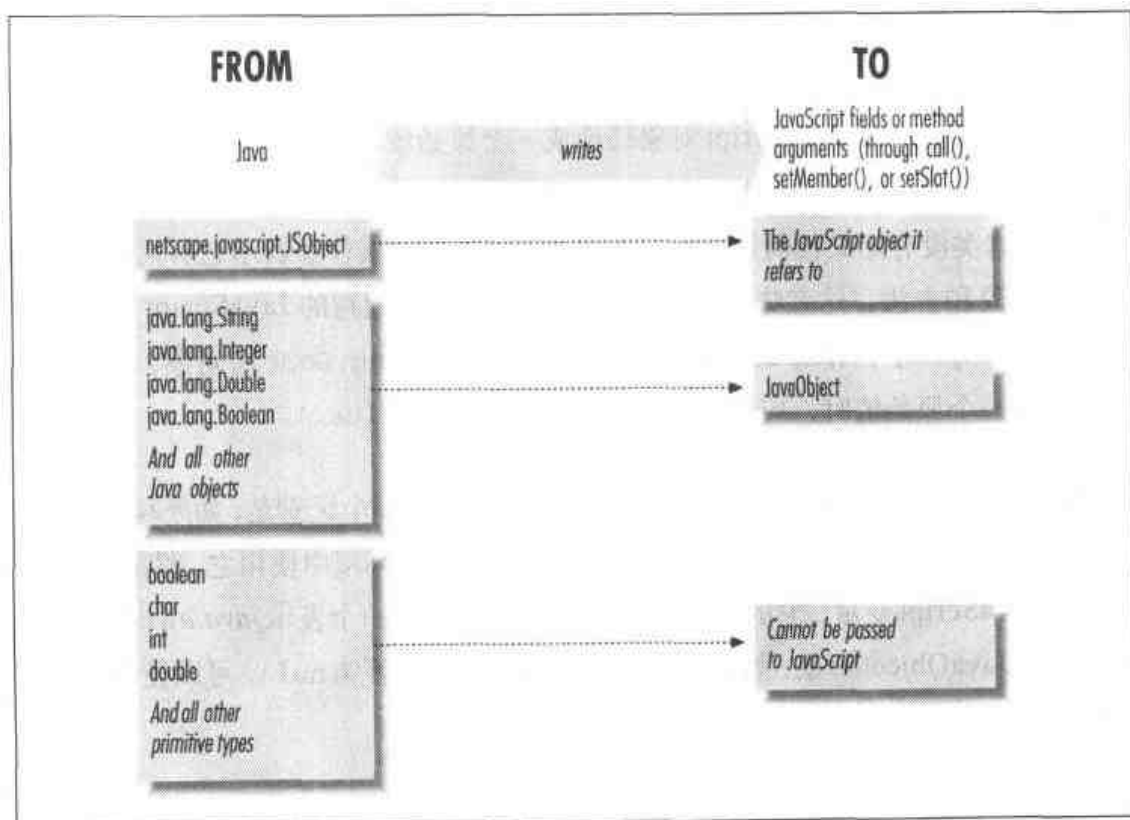


图 22-4：当 Java 小程序写 JavaScript 值时执行的数据转换

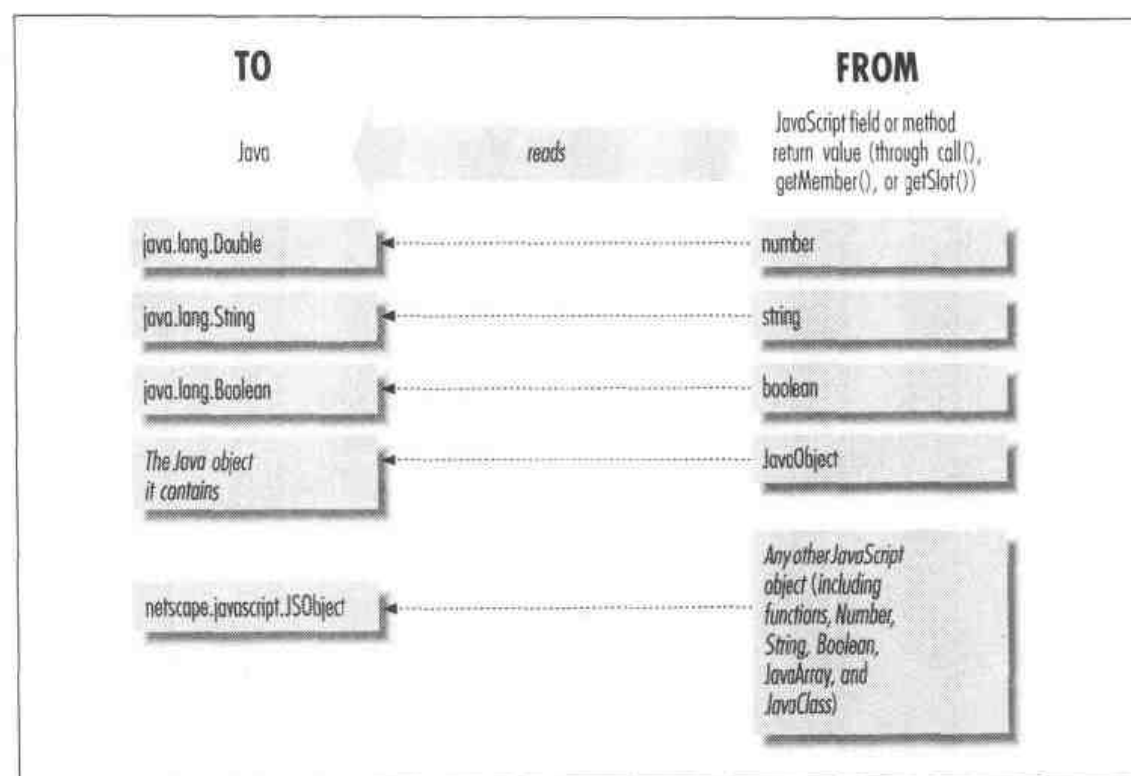


图 22-5: 当 Java 小程序读 JavaScript 值时执行的数据转换

研究这些图时需要记住, Java 只能通过 JSObject 类提供的 API 与 JavaScript 进行交互。由于 Java 是一种强类型语言, 所以由类 JSObject 定义的方法只能使用 Java 对象, 而不能使用原始类型的值。例如, 当要读一个 JavaScript 数字的值时, 方法 `getMember()` 返回的是一个 `java.lang.Double` 对象, 而不是一个原始的双精度值。

在编写一个要由 Java 程序调用的 JavaScript 函数时要记住, Java 传递的参数要么是打开 JSObject 对象得到的 JavaScript 对象, 要么是 JavaScript 对象。LiveConnect 只是不允许 Java 把原始值作为方法的参数来传递。在本章的前面部分我们见到过, JavaScript 对象的行为与其他对象有些不同之处。例如, `java.lang.Double` 类的实例行为就与原始的 JavaScript 数字不同, 甚至与 JavaScript 的 `Number` 对象也不同。当使用由 Java 设置值的 JavaScript 属性时, 也要注意同样的问题。

一种避免所有数据转换问题的方法是, 只要 Java 代码要与 JavaScript 通信, 都可以使用 JSObject 类的方法 `eval()`。为了实现这一点, Java 代码必须把所有方法的参数或属性值都转换成字符串的形式。然后, 要运行的字符串就会被无改变地传递给 JavaScript 程序, 该程序能够把数据的字符串形式转换成适当的 JavaScript 数值。

第三部分

JavaScript 核心 参考手册

本书的这一部分是 JavaScript 核心语言中的所有对象、属性、函数、方法和事件处理程序的完整参考。这一部分先解释了如何使用参考手册。

JavaScript 核心 参考手册

这一部分是一个参考手册，记载了 JavaScript 核心语言定义的类、方法和属性。引言和示例参考解释了如何使用这个参考手册，并从中获得最大的收益。仔细阅读这份资料，你会更容易找到和使用你需要的信息。

这份参考手册中的条目是按照字母顺序排列的。类的属性和方法都以它们全名的字母顺序排列，这个全名包含定义它们的类的名字。例如，如果想使用 `String` 类的方法 `replace()` 方法，应该查询“`String.replace`”，而不只是“`replace`”。

JavaScript 核心语言定义了一些全局函数和属性，如 `eval()` 和 `NaN`。从技术上说，它们是全局对象的属性。但因为全局对象没有名字，所以它们列在自己的非限定名的参考页中。为了方便起见，JavaScript 核心语言中的全局函数和属性的完整集合总结在一个特殊的参考页中，该页的名字为“`Global`”（尽管没有以 `Global` 命名的对象或类）。

有时，你会发现，自己不知道想查找的方法或属性所属的类或接口名，或者不确定应该在哪个参考手册中查找一个类或接口。本书的第六部分是一个专用的索引，这可以解决这些问题。查找一个类、方法或属性的名字，它将告诉你应该在哪个参考手册中查找，以及在该参考手册的哪个类下。例如，如果查找“`Date`”，它将告诉你 `Date` 类位于核心参考手册部分。如果查找名字“`match`”，它将告诉你，`match()` 是 `String` 类的方法，也位于核心参考手册中。

一旦找到了要查询的参考页，那么找到你需要的信息就应该没有太大的困难了。不过，如果知道参考页是如何编写和组织，就能更好地使用参考手册部分。接下来是

一个示例参考页，标题为“示例条目”，它示范了每个参考页的结构，告诉你在哪里可以找到什么类型的信息。在研究参考手册的其余部分之前，应该花时间阅读该参考页。

示例条目

可用性

如何阅读 JavaScript 核心参考手册

从 ... 继承 / 覆盖

标题和简述

每个参考条目都以一个四部分的标题块开头，如上所示。各个条目按照标题的字母顺序排列。标题行下的简短描述说明了该条目中记叙的项目，它可以帮助你快速地判断是否对该参考页余下的部分感兴趣。

可用性

标题块的右上角是可用性信息。这些信息将告诉你该项目（类、方法和属性）引入了哪个版本的 Netscape JavaScript 解释器和 Microsoft JScript 解释器。如果项目在 ECMAScript 中进行标准化，则它会告诉你引入的是哪个标准版本。可以假设在 JavaScript 某个版本中有效的内容，在其后版本中也有效。但要注意，如果这一部分说明不可以使用该项目，那么将来的版本可能会把它删除，你就应该避免使用它。

从 ... 继承 / 覆盖

如果一个类继承了超类，或一个方法覆盖了超类中的方法，该信息将显示在标题块的右下角。

第八章说过，JavaScript 的类可以继承其他类的属性和方法。例如，String 类继承 Object 类，RangeError 类继承 Error 类，Error 类继承 Object 类。在看到这些继承信息时，便可以查询列出的超类的信息。

当一个方法与超类中的方法同名时，该方法覆盖超类的方法。例如，参阅“Array.toString()”。

构造函数

如果参考页记叙一个类，通常具有“构造函数”部分，该部分说明如何使用构造方法创建类的实例。由于构造函数是一种方法，因此“构造函数”部分与方法参考页的“摘要”部分看起来很像。

摘要

函数、方法和属性的参考页都有“摘要”部分，该部分展示了如何在代码中使用这些函数、方法和属性。例如，`Array.concat()` 方法的摘要如下：

```
array.concat(value, ...)
```

斜体字说明要替换的文本。*array*应该用变量或存放数组及计算数组值的表达式来替换。*value*只表示要连接到数组上的任意值。省略号(...)说明该方法可以有任意多个 *value* 参数。因为 `concat` 和开括号及闭括号不是以斜体显示的，所以在 JavaScript 代码中，必须完全采用它们。

参数

如果参考页描述的函数、方法或类具有构造函数方法，那么“构造函数”或“摘要”后还有“参数”部分，说明这个函数、方法或构造函数的参数。如果没有参数，这个部分将被省略。

arg1

参数部分中的参数用一个列表描述。例如，这就是对参数 *arg1* 的描述。

arg2

这是对参数 *arg2* 的描述。

返回值

如果构造函数、函数或方法具有返回值，就用这一部分来解释它们的返回值。

抛出

如果构造函数、函数或方法可以抛出异常，这一部分列出了可能被抛出的异常类型，并解释了在什么环境下会抛出异常。

属性

如果参考页说明的是一个类，“属性”部分就会列出这个类定义的属性并且对每个属性进行简短的解释。在核心参考手册部分，每个属性还具有自己的参考页。例如，`Array` 类在这个部分中列出了 `length` 属性，并给出了简短解释，不过在“`Array.length`”参考页中，还有该属性的完整说明。属性列表如下：

prop1

这是属性 prop1 的概述，包括该属性的类型、作用以及它是只读的还是可读可写的。

prop2

同 prop1。

方法

如果一个类定义了方法，那么它的参考页就具有“方法”部分。这一部分与“属性”部分相似，只不过它说明的是方法而不是属性。所有方法还具有自己的参考页。

描述

大多数参考页都具有“描述”部分，该部分对要说明的类、方法、函数或属性进行了描述。它是参考页的核心。如果你是第一次学习类、方法或属性，可以直接跳到这个部分，然后再返回前面查看“参数”、“属性”和“方法”部分。如果你已经熟悉了类、方法或属性，可能不必再阅读这一部分，只需要快速查看某些特定的信息（例如“参数”部分或“属性”部分）即可。

在某些参考页中，这一部分只是一小段。但是在某些参考页中，这一部分可能会占用一页甚至更多的篇幅。对于那些非常简单的方法，“参数”和“返回值”部分已经对它进行了充分的说明，所以它的“描述”部分就被省略了。

示例

有些参考页还包括一个示例，该示例展示了该项目的习惯用法。但大多数参考页没有示例部分，可以在本书的前半部分找到它们的示例。

Bug

当一个项目不能正常运行时，这一部分描述了导致这种现象的 bug。但要注意，本书并不打算列出所有 JavaScript 版本和实现中的每一个 bug。

参阅

许多参考页的结尾都包含对相关参考页的交叉引用。有时它们还会引用本书中的某个主要章节。

arguments[]

JavaScript 1.1;JScript 2.0;ECMAScript v1

函数参数的数组

摘要

arguments

描述

arguments[] 数组只在函数体内定义。在函数体内，arguments 引用该函数的 Arguments 对象。该对象有带编号的属性，并作为一个存放传递给函数的所有参数的数组。标识符 arguments 本质上是一个局部变量，在每个函数中都会被自动声明并初始化。它只在函数体中才能引用 Arguments 对象，在全局代码中没有定义。

参阅

Arguments、第七章

Arguments

JavaScript 1.1;JScript 2.0;ECMAScript v1

一个函数的参数和其他属性

从 Object 中继承

摘要

arguments
arguments[n]

元素

Arguments 对象只在函数体中定义。虽然技术上说来，它不是数组，但 Arguments 对象有带编号的属性，这些属性可以作为数组元素，而且它有 length 属性，该属性声明了数组元素的个数。它的元素是作为参数传递给函数的值。元素 0 是第一个参数，元素 1 是第二个参数，以此类推。所有作为参数传递的值都会成为 Arguments 对象的数组元素，无论函数声明中是否有这些参数的名字。

属性

callee

对当前正在执行的函数的引用。

length

传递给函数的参数个数，同时也是 Arguments 对象中的数组元素个数。

描述

当一个函数被调用时，会为该函数创建一个 Arguments 对象，局部变量 arguments 也会自动地初始化以便引用那个 Arguments 对象。Arguments 对象的主要用途是提供一种方法，用来确定传递给函数的参数个数并且引用未命名的参数。除了数组元素和属性 length 之外，属性 callee 可以使未命名的函数引用自身。

大多数情况下，可以将 Arguments 对象看做是具有 callee 属性的数组。但它不是 Array 类的实例，Arguments.length 属性没有 Array.length 属性的专有行为，所以不能用它来改变数组的大小。

Arguments 对象有一个非常特殊的特性。当函数具有命名的参数时，Arguments 对象的数组元素是存放函数参数的局部变量的同义词。Arguments 对象和参数名提供了引用同一个变量的两种不同方法。用参数名改变参数值，会改变用 Arguments 对象得到的值，改变用 Arguments 对象得到的参数值，也会改变用参数名得到的值。

参阅

Function 和第七章

Arguments.callee

JavaScript 1.2;JScript 5.5;ECMAScript v1

当前正在运行的函数

摘要

arguments.callee

描述

属性 arguments.callee 引用当前正在运行的函数。它给未命名的函数提供了一种自我引用的方式。该属性只在函数体内被定义。

例子

```
// 一个未命名的函数直接量使用 callee 属性引用它自身，
// 以便它能够递归
var factorial = function(x) {
    if (x < 2) return 1;
    else return x * arguments.callee(x-1);
}
var y = factorial(5); // 返回120
```

Arguments.length

JavaScript 1.1;JScript 2;ECMAScript v1

传递给函数的参数个数

摘要

`arguments.length`

描述

Arguments 对象的属性 `length` 声明了传递给当前函数的参数个数。该属性只在函数体内被定义。

注意，这个属性声明的是实际传递给函数的参数个数，而不是期望传递的参数个数。有关声明的参数个数，请参阅属性 “`Function.length`”。还要注意，该属性不具备 `Array.length` 属性的专有行为。

例子

```
// 使用 Arguments 对象来检查是否正确传递了 * 参数
function check(args) {
    var actual = args.length;           // 参数的实际个数
    var expected = args.caller.length; // 希望的参数个数
    if (actual != expected) {           // 如果它们不匹配，则抛出异常
        throw new Error("Wrong number of arguments: expected: " +
                        expected + "; actually passed: " + actual);
    }
}
// 一个演示如何使用以上函数的函数
function f(x, y, z) {
    check(arguments); // 检查参数个数是否正确
    return x + y + z; // 继续使用函数
}
```

参阅

`Array.length` 和 `Function.length`

Array

JavaScript 1.1;JScript 2.0;ECMAScript v1

对数组的内部支持

构造函数

```
new Array()
new Array(size)
new Array(element0, element1, ..., elementn)
```

参数

size

期望的数组元素个数。返回的数组，length域将被设为 *size* 的值。

element0, ...elementn

两个或多个值的参数列表。当使用这些参数来调用构造函数 `Array()` 时，新创建的数组的元素就会被初始化为这些值，它的length域也会被设置为参数的个数。

返回值

新创建并被初始化了的数组。如果调用构造函数 `Array()` 时没有使用参数，那么返回的数组为空，length域为0。当调用构造函数时只传递给它一个数字参数，该构造函数将返回具有指定个数、元素为 `undefined` 的数组。当用其他参数调用 `Array()` 时，该构造函数将用参数指定的值初始化数组。当把构造函数作为函数调用，不使用 `new` 运算符时，它的行为与使用 `new` 运算符调用它时的行为完全一样。

抛出

`RangeError`

当只传递给 `Array()` 构造函数一个整数参数 *size* 时，如果 *size* 是负数，或者大于 $2^{32} - 1$ ，将抛出 `RangeError` 异常。

直接量语法

ECMAScript v3 规定了数组直接量的语法，JavaScript 1.2 和 JScript 3.0 实现了它。可以把一个用逗号分隔的表达式列表放在方括号中，创建并初始化一个数组。这些表达式的值将成为数组元素。例如：

```
var a = [1, true, 'abc'];
var b = [a[0], a[0]*2, f(x)];
```

属性

`length`

一个可读可写的整数，声明了数组中的元素个数。如果数组中的元素不连续，它就是比数组中的最后一个元素的下标大1的整数。改变这个属性的值将截断或扩展数组。

方法

`concat()`

给数组添加元素。

<code>join()</code>	将数组中的所有元素都转换成字符串，然后连接起来。
<code>pop()</code>	从数组尾部删除一个项目。
<code>push()</code>	把一个项目添加到数组的尾部。
<code>reverse()</code>	在原数组上颠倒数组中元素的顺序。
<code>shift()</code>	将数组头部的元素移出数组头部。
<code>slice()</code>	返回数组的一个子数组。
<code>sort()</code>	在原数组上对数组元素进行排序。
<code>splice()</code>	插入、删除或替换一个数组元素。
<code>toLocaleString()</code>	把数组转换成一个局部字符串。
<code>toString()</code>	把数组转换成一个字符串。
<code>unshift()</code>	在数组的头部插入一个元素。

描述

数组是 JavaScript 的基本句法特性。第九章详细说明过它们。

参阅

第九章

Array.concat() JavaScript 1.2;JScript 3.0;ECMAScript v3

连接数组

摘要

`array.concat(value, ...)`

参数

`value, ...`

要添加到 `array` 中的值，可以是任意多个。

返回值

一个新数组，是把指定的所有参数添加到 `array` 中构成的。

描述

方法 `concat()` 将创建并返回一个新数组，这个数组是将所有参数都添加到 `array` 中生成的。它并不修改 `array`。如果要进行 `concat()` 操作的参数是一个数组，那么添加的是数组中的元素，而不是数组。

例子

```
var a = [1,2,3];  
a.concat(4, 5);           // 返回 [1, 2, 3, 4, 5]  
a.concat([4,5]);          // 返回 [1, 2, 3, 4, 5]  
a.concat([4,5],[6,7]);    // 返回 [1, 2, 3, 4, 5, 6, 7]  
a.concat(4, [5,[6,7]]);  // 返回 [1, 2, 3, 4, 5, [6, 7]]
```

参阅

Array.join()、Array.push()和 Array.splice()

Array.join()

JavaScript 1.1;JScript 2.0;ECMAScript v1

将数组元素连接起来以构建一个字符串

摘要

```
array.join()  
array.join(separator)
```

参数

separator

在返回的字符串中用于分隔数组元素的字符或字符串,它是可选的。如果省略了这个参数,用逗号作为分隔符。

返回值

一个字符串,通过把 *array* 的每个元素转换成字符串,然后把把这些字符串连接起来,在两个元素之间插入 *separator* 字符串而生成。

描述

方法 `join()` 将把每个数组元素转换成一个字符串,然后把把这些字符串连接起来,在两个元素之间插入指定的 *separator* 字符串。返回生成的字符串。

可以用 `String` 对象的 `split()` 方法执行相反的操作,即把一个字符串分割成数组元素。详情参见“`String.split()`”参考页。

例子

```
a = new Array(1, 2, 3, "testing");  
s = a.join('+'); // s 是字符串 "1+2+3+testing"
```

参阅

String.split()

Array.length

JavaScript 1.1;JScript 2.0;ECMAScript v1

数组的大小

摘要

array.length

描述

数组的 length 属性总是比数组中定义的最后 一个元素的下标大一。对于那些具有连续元素, 而且以元素 0 开始的常规数组来说, 属性 length 声明了数组中的元素个数。

数组的 length 属性在用构造函数 Array() 创建数组时初始化。给数组添加新元素时, 如果必要, 将更新 length 的值:

```
a = new Array();           // a.length 被初始化为 0
b = new Array(10);         // b.length 被初始化为 10
c = new Array('one', 'two', 'three'); // c.length 被初始化为 3
c[3] = "four";             // c.length 被更新为 4
c[10] = "blasted!";        // c.length 变为 11
```

设置属性 length 的值可以改变数组的大小。如果设置的值比它的当前值小, 数组将被截断, 其尾部的元素将丢失。如果设置的值比它的当前值大, 数组将增大, 新元素被添加到数组尾部, 它们的值为 undefined。

Array.pop()

JavaScript 1.2;JScript 5.5;ECMAScript v3

删除并返回数组的最后一个元素

摘要

array.pop()

返回值

array 的最后一个元素

描述

方法 pop() 将删除 array 的最后一个元素, 把数组长度减 1, 并且返回它删除的元素的值。如果数组已经为空, 则 pop() 不改变数组, 返回 undefined。

例子

方法 `pop()` 和方法 `push()` 可以提供先进后出 (FILO) 栈的功能。例如:

```
var stack = []; // 栈: []
stack.push(1, 2); // 栈: [1, 2]      返回值为 2
stack.pop();      // 栈: [1]        返回值为 2
stack.push(4, 5); // 栈: [1, 4, 5]   返回值为 [4, 5]
stack.pop();      // 栈: [1, 4, 5]   返回值为 [4, 5]
stack.pop();      // 栈: [1]        返回值为 5
stack.pop();      // 栈: []          返回值为 1
```

参阅

`Array.push()`

`Array.push()`

JavaScript 1.2; JScript 5.5; ECMAScript v3

给数组添加元素

摘要

`array.push(value, ...)`

参数

`value, ...`

要添加到 `array` 尾部的值, 可以是一个或多个。

返回值

把指定的值添加到数组后的新长度。

描述

方法 `push()` 将它的参数顺次添加到 `array` 的尾部。它直接修改 `array`, 而不是创建一个新的数组。方法 `push()` 和方法 `pop()` 用数组提供先进后出栈的功能。参阅“`Array.pop()`”中的示例。

Bug

在 JavaScript 的 Netscape 实现中, 如果把语言版本明确地设置为 1.2, 该函数将返回最后添加的值, 而不是返回新数组的长度。

参阅

`Array.pop()`

Array.reverse()

JavaScript 1.1;JScript 2.0;ECMAScript v1

颠倒数组中的元素顺序

摘要

`array.reverse()`

描述

Array 对象的方法 `reverse()` 将颠倒数组中元素的顺序。它在原数组上实现这一操作，即重排指定的 `array` 的元素，但并不创建新数组。如果对 `array` 有多个引用，那么通过所有引用都可以看到数组元素的新顺序。

例子

```
a = new Array(1, 2, 3);    // a[0] == 1, a[2] == 3;
a.reverse();              // 现在 a[0] == 3, a[2] == 1;
```

Array.shift()

JavaScript 1.2;JScript 5.5;ECMAScript v3

将元素移出数组

摘要

`array.shift()`

返回值

数组原来的第一个元素。

描述

方法 `shift()` 将把 `array` 的第一个元素移出数组，返回那个元素的值，并且将余下的所有元素前移一位，以填补数组头部的空缺。如果数组是空的，`shift()` 将不进行操作，返回 `undefined`。注意，该方法不创建新数组，而是直接修改原有的数组。

方法 `shift()` 和方法 `Array.pop()` 相似，只不过它在数组头部操作，而不是在尾部操作。该方法常常和 `unshift()` 一起使用。

例子

```
var a = [1, [2,3], 4]
a.shift();    // 返回值 1; a = [[2, 3], 4]
a.shift();    // 返回值 [2, 3]; a = [4]
```

参阅

`Array.pop()`和`Array.unshift()`

`Array.slice()`

JavaScript 1.2;JScript 3.0;ECMAScript v3

返回数组的一部分

摘要

```
array.slice(start, end)
```

参数

start

数组片段开始处的数组下标。如果是负数，它声明从数组尾部开始算起的位置。也就是说，-1 指最后一个元素，-2 指倒数第二个元素，以此类推。

end

数组片段结束处的后一个元素的数组下标。如果没有指定这个参数，切分的数组包含从 *start* 开始到数组结束的所有元素。如果这个参数是负数，它声明的是从数组尾部开始算起的元素。

返回值

一个新数组，包含从 *start* 到 *end*（不包括该元素）指定的 *array* 元素。

描述

方法 `slice()` 将返回数组的一部分，或者说是一个子数组。返回的数组包含从 *start* 开始到 *end* 之间的所有元素，但是不包括 *end* 所指的元素。如果没有指定 *end*，返回的数组包含从 *start* 开始到原数组结尾的所有元素。

注意，该方法并不修改数组。如果想删除数组中的一段元素，应该使用方法 `Array.splice()`。

例子

```
var a = [1,2,3,4,5];
a.slice(0,3);      // 返回值 [1, 2, 3]
a.slice(3);        // 返回值 [4, 5]
a.slice(1,-1);     // 返回值 [2, 3, 4]
a.slice(-3,-2);    // 返回值 [3]。由于1E 4中的bug，所以返回值为[1, 2, 3]
```

Bug

在 Internet Explorer 4 中, 参数 `start` 不能为负数。

参阅

`Array.splice()`

`Array.sort()`

JavaScript 1.1;JScript 2.0;ECMAScript v1

对数组元素排序

摘要

```
array.sort()
```

```
array.sort(orderfunc)
```

参数

`orderfunc`

用来指定按什么顺序进行排序的函数, 可选。

返回值

对数组的引用。注意, 数组在原数组上进行排序, 不制作副本。

描述

方法 `sort()` 将在原数组上对数组元素进行排序, 即排序时不创建新的数组副本。如果调用方法 `sort()` 时没有使用参数, 将按字母顺序 (更为精确地说, 是按照字符编码的顺序) 对数组中的元素进行排序。要实现这一点, 首先应把数组的元素都转换成字符串 (如果有必要的话), 以便进行比较。

如果想按照别的顺序进行排序, 就必须提供比较函数, 该函数要比较两个值, 然后返回一个用于说明这两个值的相对顺序的数字。比较函数应该具有两个参数 `a` 和 `b`, 其返回值如下:

- 如果根据你的评判标准, `a` 小于 `b`, 在排序后的数组中 `a` 应该出现在 `b` 之前, 就返回一个小于 0 的值。
- 如果 `a` 等于 `b`, 就返回 0。
- 如果 `a` 大于 `b`, 就返回一个大于 0 的值。

注意，数组中 `undefined` 的元素都排列在数组末尾。即使你提供了自定义的排序函数，也是这样，因为 `undefined` 值不会被传递给你提供的 `orderfunc`。

例子

下面的代码展示了如何编写按数字顺序，而不是按字母顺序对数组进行排序的比较函数：

```
// 按照数字顺序排序的排序函数
function numberorder(a, b) { return a - b; }

a = new Array(33, 4, 1111, 222);
a.sort();           // 按照字母顺序的排序结果为: 1111, 222, 33, 4
a.sort(numberorder); // 按照数字顺序的排序结果为: 4, 33, 222, 1111
```

Array.splice()

JavaScript 1.2;JScript 5.5;ECMAScript v3

插入、删除或替换数组的元素

摘要

`array.splice(start, deleteCount, value, ...)`

参数

start

开始插入和/（或）删除的数组元素的下标。

deleteCount

从 *start* 开始，包括 *start* 所指的元素在内要删除的元素个数。这个参数是可选的，如果没有指定它，`splice()` 将删除从 *start* 开始到原数组结尾的所有元素。

value, ...

要插入数组的零个或多个值，从 *start* 所指的下标处开始插入。

返回值

如果从 *array* 中删除了元素，返回的是含有被删除的元素的数组。但是要注意，由于存在一个 bug，因此在 JavaScript 1.2 的 Netscape 实现中，返回的并不总是数组。

描述

方法 `splice()` 将删除从 *start* 开始（包括 *start* 所指的元素在内）的零个或多个元素，并且用参数列表中声明的一个或多个值来替换那些被删除的元素。位于插入或删除

除的元素之后的数组元素都会被移动,以保持它们与数组其他元素的连续性。注意,虽然 `splice()` 方法与 `slice()` 方法名字相似,但作用不同,方法 `splice()` 直接修改数组。

例子

读了下面的例子,就很容易理解 `splice()` 的操作了:

```
var a = [1,2,3,4,5,6,7,8]
a.splice(4);           // 返回值[5, 6, 7, 8]。a的值为[1, 2, 3, 4]
a.splice(1,2);         // 返回值[2, 3]。a的值为[1, 4]
a.splice(1,1);         // 应该返回[4],但Netscape/JavaScript 1.2则返回4
a.splice(1,0,2,3);     // 应该返回[],但Netscape/JavaScript 1.2则返回undefined
```

Bug

方法 `splice()` 假定在各种情况下均返回一个包含已删除元素的数组。但是,在 Netscape 的 JavaScript 1.2 解释器中,如果删除的是单个元素,那么该方法返回的是元素,而不是包含那个元素的数组。如果没有删除任何元素,它不是返回一个空数组,而是什么都不返回。只要把语言版本明确地设置为 1.2, JavaScript 的 Netscape 实现都有这种 bug 行为。

参阅

`Array.slice()`

`Array.toLocaleString()`

JavaScript 1.5;JScript 5.5;ECMAScript v1

把数组转换成局部字符串

覆盖 `object.toLocaleString()`

摘要

`array.toLocaleString()`

返回值

数组 `array` 的局部字符串表示。

抛出

`TypeError`

调用该方法时,若对象不是 `Array`,则抛出该异常。

描述

数组的方法 `toString()` 将返回数组的局部字符串表示。它首先调用每个数组元素的

toLocaleString() 方法, 然后地区特定的分隔符把生成的字符串连接起来, 形成一个字符串。

参阅

Array.toString(), Object.toLocaleString()

Array.toString()

JavaScript 1.1; JScript 2.0; ECMAScript v1

将数组转换成一个字符串

覆盖 Object.toString()

摘要

array.toString()

返回值

array 的字符串表示。

抛出

TypeError

调用该方法时, 若对象不是 Array, 则抛出该异常。

描述

数组的 toString() 方法将把数组转换成一个字符串, 并且返回这个字符串。当数组用于字符串环境中时, JavaScript 会调用这一方法将数组自动转换成一个字符串。但在某些情况下, 需要明确地调用这个方法。

toString() 在把数组转换成字符串时, 首先要将数组的每个元素都转换成字符串 (通过调用这些元素的 toString() 方法)。当每个元素都被转换成字符串时, 它就以列表的形式输出这些字符串, 字符串之间用逗号分隔。返回值与没有参数的 join() 方法返回的字符串相同。

Bug

在 Netscape 实现中, 如果把语言版本明确地设置为 1.2, toString() 将会返回用逗号和空格分隔的数组元素列表, 这个列表采用数组直接量表示法, 用方括号括起元素。例如, 在把 <script> 标记的 language 性质明确地设置为 “JavaScript 1.2” 时, 就会发生这种情况。

参阅

`Array.toLocaleString()`, `Object.toString()`

`Array.unshift()` JavaScript 1.2; JScript 5.5; ECMAScript v3

在数组头部插入元素

摘要

```
array.unshift(value, ...)
```

参数

`value, ...`

要插入数组头部的一个或多个值。

返回值

数组的新长度。

描述

方法 `unshift()` 将它的参数插入 `array` 的头部，并将已经存在的元素顺次地移到较高的下标处，以便留出空间。该方法的第一个参数将成为数组新的元素 0，如果还有第二个参数，它将成为新的元素 1，以此类推。注意，`unshift()` 不创建新数组，而是直接修改原有的数组。

例子

方法 `unshift()` 通常和方法 `shift()` 一起使用。例如：

```
var a = [];           // a: []
a.unshift(1);          // a: [1]           返回值: 1
a.unshift(22);         // a: [22, 1]       返回值: 2
a.shift();             // a: [1]           返回值: 22
a.unshift(33, 4, 5);   // a: [33, 4, 5, 1]  返回值: 3
```

参阅

`Array.shift()`

Boolean

JavaScript 1.1;JScript 2.0;ECMAScript v1

对布尔值的支持

从 Object 继承

构造函数

```
new Boolean(value)    // 构造函数
Boolean(value)        // 转换函数
```

参数

value

由布尔对象存放的值或者要转换成布尔值的值。

返回值

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）调用的，Boolean() 只将它的参数转换成一个原始的布尔值，并且返回这个值。

0、NaN、null、空字符串 “” 和 undefined 都将转换成 false。其他的原始值，除了 false（但包括字符串 “false”），以及其他的对象和数组都会被转换成 true。

方法

toString()

根据 Boolean 对象代表的布尔值返回 “true” 或 “false”。

valueOf()

返回 Boolean 对象中存放的原始布尔值。

描述

在 JavaScript 中，布尔值是一种基本的数据类型。Boolean 对象是一个将布尔值打包的布尔对象。Boolean 对象主要用于提供将布尔值转换成字符串的 toString() 方法。当调用 toString() 方法将布尔值转换成字符串时（通常是由 JavaScript 隐式地调用），JavaScript 会内在地将这个布尔值转换成一个临时的 Boolean 对象，然后调用这个对象的 toString() 方法。

参阅

Object

Boolean.toString()

JavaScript 1.1;JScript 2.0;ECMAScript v1

将布尔值转换成字符串

覆盖 Object.toString() 方法

摘要`b.toString()`**返回值**根据原始布尔值或者 Boolean 对象 *b* 的值返回字符串 “true” 或 “false”。**抛出**

TypeError

如果调用该方法时，对象不是 Boolean，则抛出该异常。

Boolean.valueOf()

JavaScript 1.1;JScript 2.0;ECMAScript v1

Boolean 对象的布尔值

覆盖 Object.valueOf() 方法

摘要`b.valueOf()`**返回值**Boolean 对象 *b* 存放的原始布尔值。**抛出**

TypeError

如果调用该方法时，对象不是 Boolean，则抛出该异常。

Date

JavaScript 1.0;JScript 1.0;ECMAScript v1

操作日期和时间的对象

继承 Object 对象

构造函数`new Date()``new Date(milliseconds)``new Date(datestring)``new Date(year, month, day, hours, minutes, seconds, ms)`

没有参数的构造函数 `Date()` 将把创建的 `Date` 对象设置为当前的日期和时间。如果传递给它的参数是一个数字, 那么这个数字将被作为日期的内部数字表示, 其单位是毫秒, 就像方法 `getTime()` 的返回值一样。如果传递给它的参数是一个字符串, 它就是日期的字符串表示, 其格式就是方法 `Date.parse()` 接受的格式。否则, 传递给该构造函数的参数是 2 ~ 7 个数字, 它们分别指定了日期和时间的各个字段。除了前两个字段 (年和月字段) 外, 其他所有字段都是可选的。注意, 声明这些日期和时间的字段使用的都是本地时间, 而不是 UTC 时间 (类似于 GMT 时间)。参阅静态方法 `Date.UTC()`。

`Date()` 还可以作为普通函数被调用, 而不带有运算符 `new`。以这种方式调用时, `Date()` 将忽略传递给它的所有参数, 返回当前日期和时间的字符串表示。

参数

milliseconds

期望的日期距 1970 年 1 月 1 日午夜 (UTC) 的毫秒数。例如, 假定传递的参数值为 5000, 那么创建的 `Date` 对象代表日期的就是 1970 年 1 月 1 日午夜过 5 秒。

datestring

一个字符串, 声明了日期, 也可以同时声明时间。这个字符串的格式应该是方法 `Date.parse()` 能接受的。

year

年份, 一个四位数。例如, 2001 指的是 2001 年。为了与早期的 JavaScript 实现兼容, 如果它的值在 0 ~ 99 之间, 则给它加上 1900。

month

月份, 0 (代表一月) 到 11 (代表十二月) 之间的一个整数。

day

一个月的某一天, 1 ~ 31 之间的一个整数。注意, 这个参数将 1 作为它的最小值, 而其他参数则以 0 为最小值。该参数是可选的。

hours

小时, 0 (午夜) 到 23 (晚上 11 点) 之间的一个整数。该参数是可选的。

minutes

分钟, 0 ~ 59 之间的一个整数。该参数是可选的。

`seconds`

秒，0 - 59 之间的一个整数。该参数是可选的。

`ms`

毫秒，0 - 999 之间的一个整数。该参数是可选的。

方法

`Date` 对象没有可以直接读写的属性，所有对日期和时间值的访问都是通过方法执行的。`Date` 对象的大多数方法采用两种形式，一种是使用本地时间进行操作，另一种是使用世界（UTC 或 GMT）时进行操作。如果方法的名字中有“UTC”，它将使用世界时进行操作。下面将这些方法对列在一起。例如，`get[UTC]Day()` 指方法 `getDay()` 和 `getUTCDay()`。

只有 `Date` 对象才能调用 `Date` 方法，如果用其他类型的对象调用这些方法，将抛出异常 `TypeError`。

`Get[UTC]Date()`

返回 `Date` 对象所代表的月中的某一天，采用本地时间或世界时。

`get[UTC]Day()`

返回 `Date` 对象所代表的一周中的某一天，采用本地时间或世界时。

`get[UTC]FullYear()`

返回日期中的年份，用四位数表示，采用本地时间或世界时。

`get[UTC]Hours()`

返回 `Date` 对象的小时字段，采用本地时间或世界时。

`get[UTC]Milliseconds()`

返回 `Date` 对象的毫秒字段，采用本地时间或世界时。

`get[UTC]Minutes()`

返回 `Date` 对象的分钟字段，采用本地时间或世界时。

`get[UTC]Month()`

返回 `Date` 对象的月份字段，采用本地时间或世界时。

`get[UTC]Seconds()`

返回 `Date` 对象的秒字段，采用本地时间或世界时。

`getTime()`

返回 `Date` 对象的内部毫秒表示。注意，该值独立于时区，所以没有单独的 `getUTCtime()` 方法。

`getTimezoneoffset()`

返回这个日期的本地时间和 UTC 表示之间的时差，以分钟为单位。注意，是否是夏令时或在指定的日期中夏令时是否有效，将决定该方法的返回值。

`getFullYear()`

返回 `Date` 对象的年份。一般不使用这种方法，推荐使用的方法是 `getFullYear()`。

`set[UTC]Date()`

设置 `Date` 对象的月中的某一天，采用本地时间或世界时。

`set[UTC]FullYear()`

设置 `Date` 对象的年份字段（月份和天数字段可选），采用本地时间或世界时。

`set[UTC]Hours()`

设置 `Date` 对象的小时字段（分钟、秒和毫秒字段可选），采用本地时间或世界时。

`set[UTC]Milliseconds()`

设置 `Date` 对象的毫秒字段，采用本地时间或世界时。

`set[UTC]Minutes()`

设置 `Date` 对象的分钟字段（秒和毫秒字段可选），采用本地时间或世界时。

`set[UTC]Month()`

设置 `Date` 对象的月份字段（一个月的天数字段可选），采用本地时间或世界时。

`set[UTC]Seconds()`

设置 `Date` 对象的秒字段（毫秒字段可选），采用本地时间或世界时。

`setTime()`

使用毫秒的形式设置 `Date` 对象的各个字段。

`setYear()`

设置 `Date` 对象的年份字段。该方法被反对使用，推荐使用的方法是 `setFullYear()`。

`toString()`

返回日期的日期部分的字符串表示，采用本地时间。

`toGMTString()`

将 `Date` 对象转换成一个字符串，采用 GMT 时间区。该方法被反对使用，推荐使用的的方法是 `toUTCString()`。

`toLocaleDateString()`

返回表示日期的日期部分的字符串，采用地方日期，使用地方日期的格式化规约。

`toLocaleString()`

将 `Date` 对象转换成一个字符串，采用本地时间和地方日期的格式化规约。

`toLocaleTimeString()`

返回日期的时间部分的字符串表示，采用本地时间，使用本地时间的格式化规约。

`toString()`

将 `Date` 对象转换成一个字符串，采用本地时间。

`getTimeString()`

返回日期的时间部分的字符串表示，采用本地时间。

`toUTCString()`

将 `Date` 对象转换成一个字符串，采用世界时。

`valueOf()`

将 `Date` 对象转换成它的内部毫秒格式。

静态方法

除了上面列出的实例方法之外，`Date` 对象还定义了两个静态方法。这两个方法由构造函数 `Date()` 自身调用，而不是由 `Date` 对象调用：

`Date.parse()`

解析日期和时间的字符串表示，返回它的内部毫秒表示。

`Date.UTC()`

返回指定的 UTC 日期和时间的毫秒表示。

描述

`Date` 对象是 JavaScript 语言的一种内部数据类型。它由语法 `new Date()` 语法创建，我们在前面的构造函数部分中已经说明了这种语法。

创建了Date对象后,就可以使用多种方法来操作它。大多数方法只能用来设置或者获取对象的年份字段、月份字段、天数字段、小时字段、分钟字段以及秒字段,采用本地时间或UTC(世界时或GMT)时间。方法toString()以及它的变种可以把日期转换成人们能够读懂的字符串。所谓Date对象的内部表示就是距1970年1月1日午夜(GMT时间)的毫秒数,方法getTime()可以把Date对象转换为内部表示,方法setTime()可以把它从内部表示转换成其他形式。采用标准的毫秒格式时,日期和时间由一个整数表示,这使得日期算术变得格外简单。ECMAScript标准要求Date对象能够把1970年1月1日前后10亿天中的任意日期和时间表示为毫秒。这个范围在正负273~785年之间,所以JavaScript的时钟不会超过275755年。

例子

一旦创建了Date对象,就可以用各种方法操作它:

```
d = new Date(); // 获取当前日期和时间
document.write('Today is: ' + d.toLocaleDateString() + ' '); // 显示日期
document.write('The time is: ' + d.toLocaleTimeString(); // 显示时间
var dayOfWeek = d.getDay(); // 它是一周第几天
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6); // 是周末吗
```

Date对象的另一种常见用法是用某个时间的毫秒表示减去当前时间的毫秒表示来判断两个时间的时差。下面的例子说明了这种用法:

```
<script language="JavaScript">
today = new Date(); // 为今天的日期创建对象
christmas = new Date(); // 获取的具有当前年份的日期
christmas.setMonth(11); // 把月份设为11月
christmas.setDate(25); // 把日期设为25号

// 如果圣诞节已经过了,计算当前日期和圣诞节之间的毫秒数
// 把它转换成天数
// 输出消息
if (today.getTime() < christmas.getTime()) {
    difference = christmas.getTime() - today.getTime();
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
    document.write('Only ' + difference + ' days until Christmas!<br>');
}
</script>
```

// 此处是其余的HTML文档

```
<script language="JavaScript">
// 下面我们用Data对象计时
// 用1000 除它,把毫秒转换成秒
now = new Date();
document.write('<p>It took ' +
```

```
(now.getTime()-today.getTime())/1000  
'seconds to load this page. ');  
</script>
```

参阅

Date.parse()和 Date.UTC()

Date.getDate

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回一个月中的某一天

摘要

date.getDate()

返回值

指定 Date 对象 *date* 所指的月份中的某一天，使用本地时间。返回值是 1 ~ 31 之间的一个整数。

Date.getDay()

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回一周中的某一天

摘要

date.getDay()

返回值

指定 Date 对象 *date* 所指的一个星期中的某一天，使用本地时间。返回值是 0（周日）到 6（周六）之间的一个整数。

Date.getFullYear

JavaScript 1.2;JScript 3.0;ECMAScript v1

返回年份

摘要

date.getFullYear()

返回值

当 *date* 用本地时间表示时返回的年份。返回值是一个四位数，表示包括世纪值在内的完整年份，而不是两位数的缩写形式。

Date.getHours()JavaScript 1.0;JScript 1.0;ECMAScript v1

返回 Date 对象的小时字段

摘要

```
date.getHours()
```

返回值

指定 Date 对象 *date* 的小时字段，以本地时间表示。返回值是 0（午夜）到 23（晚上 11 点）之间的一个整数。

Date.getMilliseconds()JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 Date 对象的毫秒字段

摘要

```
date.getMilliseconds()
```

返回值

date 的毫秒字段，用本地时间表示。

Date.getMinutes()JavaScript 1.0;JScript 1.0;ECMAScript v1

返回 Date 对象的分钟字段

摘要

```
date.getMinutes()
```

返回值

指定的 Date 对象 *date* 的分钟字段，以本地时间表示。返回值在 0 ~ 59 之间。

Date.getMonth()JavaScript 1.0;JScript 1.0;ECMAScript v1

返回 Date 对象的月份字段

摘要

```
date.getMonth()
```

返回值

指定的 `Date` 对象 `date` 的月份字段，以本地时间表示。返回值在 0（一月）到 11（十二月）之间。

`Date.getSeconds()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回 `Date` 对象的秒字段

摘要

```
date.getSeconds()
```

返回值

指定的 `Date` 对象 `date` 的秒字段，以本地时间表示。返回值在 0 ~ 59 之间。

`Date.getTime()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回 `Date` 对象的毫秒表示

摘要

```
date.getTime()
```

返回值

指定的 `Date` 对象 `date` 的毫秒表示，也就是 `date` 指定的日期和时间距 1970 年 1 月 1 日午夜（GMT 时间）之间的毫秒数。

描述

方法 `getTime()` 可以将日期和时间转换成一个整数。这在比较两个 `Date` 对象或者要判断两个日期之间的时差时非常有用。注意，日期的毫秒表示独立于时区，所以除了这个方法外，没有 `getUTCTime()` 方法。不要混淆 `getTime()` 方法和 `getDay()` 及 `getDate()` 方法，`getDay()` 及 `getDate()` 返回的分别是一周的第几天和一个月的第几天。

可以使用 `Date.parse()` 或 `Date.UTC()` 将日期和时间转换成它们的毫秒表示，在此之前无须先创建一个 `Date` 对象。

参阅

`Date`、`Date.parse()`、`Date.setTime()` 和 `Date.UTC()`

Date.getTimezoneOffset() JavaScript 1.0;JScript 1.0;ECMAScript v1

判断与 GMT 的时间差

摘要

`date.getTimezoneOffset()`

返回值

本地时间与 GMT 时间之间的时差，以分钟为单位。

描述

`getTimezoneOffset()` 返回的是本地时间和 GMT 时间或 UTC 时间之间相差的分钟数。实际上，该函数告诉了你运行 JavaScript 代码的时区，以及指定的时间是否是夏令时。

返回值以分钟计，而不是以小时计，原因是某些国家所占有的时区甚至不到一个小时的间隔。

Date.getUTCDate() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回该天是一个月的哪一天（世界时）

摘要

`date.getUTCDate()`

返回值

当 `date` 对象用世界时表示时，返回值是该月中的哪一天（是 1 ~ 31 的一个值）。

Date.getUTCDay() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回该天是星期几（世界时）

摘要

`date.getUTCDay()`

返回值

当 `date` 对象用世界时表示时，返回值是该星期中的哪一天。该值在 0（星期天）到 6（星期六）之间。

Date.getUTCFullYear() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回年份（世界时）

摘要

`date.getUTCFullYear()`

返回值

当 `date` 对象是用世界时表示时所代表的年份。该值是四位数，而不是两位数的缩写。

Date.getUTCHours() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 `Date` 对象的小时字段（世界时）

摘要

`date.getUTCHours()`

返回值

当 `date` 对象用世界时表示时的小时字段，该值在 0（午夜）到 23（晚上 11 点）之间。

Date.getUTCMilliseconds() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 `Date` 对象的毫秒字段（世界时）

摘要

`date.getUTCMilliseconds()`

返回值

当 `date` 对象是用世界时表示时的毫秒字段。

Date.getUTCMinutes() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 `Date` 对象的分钟字段（世界时）

摘要

`date.getUTCMinutes()`

返回值

当 `date` 对象用世界时表示时的分钟字段，该值是 0 ~ 59 之间的整数。

Date.getUTCMonth() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 Date 对象的月份（世界时）

摘要

`date.getUTCMonth()`

返回值

当 `date` 对象用世界时表示时的月份，该值是 0（一月）到 11（十二月）之间的一个整数。注意，Date 对象以 1 代表某个月的第一天，而不是像月份字段那样使用 0 代表一年的第一个月。

Date.getUTCSeconds() JavaScript 1.2;JScript 3.0;ECMAScript v1

返回 Date 对象的秒字段（世界时）

摘要

`date.getUTCSeconds()`

返回值

当 `date` 对象用世界时表示时的秒字段，该值是 0 ~ 59 之间的一个整数。

Date.getYear() JavaScript 1.0;JScript 1.0;ECMAScript v1;ECMAScript v3 反对使用

返回 Date 对象的年份字段（世界时）

摘要

`date.getYear()`

返回值

指定 Date 对象 `date` 的年份字段减去 1900。

描述

方法 `getYear()` 返回的是指定的 Date 对象的年份字段减去 1900 后的值。从 ECMAScript v3 起，JavaScript 的实现就不再要求使用该函数，而使用 `getFullYear()` 函数代替它。

Bug

在 JavaScript 1.0 到 1.2 的 Netscape 实现中，只将 1900 和 1999 之间的年份减去 1900。

Date.parse() JavaScript 1.0;JScript 1.0;ECMAScript v1
解析日期/时间字符串

摘要

`Date.parse(date)`

参数

date

含有要解析的日期和时间的字符串

返回值

指定的日期和时间距 1970 年 1 月 1 日午夜 (GMT 时间) 之间的毫秒数。

描述

`Date.parse()` 是 `Date` 对象的静态方法。一般通过 `Date` 构造函数, 采用 `Date.parse()` 的形式调用它, 而不是通过 `date` 对象, 采用 `date.parse()` 调用该方法。`Date.parse()` 只有一个字符串型的参数。它将解析这个字符串中的日期, 然后返回它的毫秒形式, 这种形式可以直接使用, 也可以用于创建一个新的 `Date` 对象, 还可以用 `Date.setTime()` 方法来设置一个已经存在的日期。

ECMAScript 标准没有规定 `Date.parse()` 方法解析的字符串的格式, 只是说该方法能解析 `Date.toString()` 方法和 `Date.toUTCString()` 方法返回的字符串。但是, 这些函数根据实现来格式化日期, 所以以某种方式编写所有 JavaScript 实现都能理解的日期是不可能的。

参阅

`Date`、`Date.setTime()`、`Date.toGMTString()` 和 `Date.UTC`

Date.setDate() JavaScript 1.0;JScript 1.0;ECMAScript v1
设置一个月的某一天

摘要

`date.setDate(day_of_month)`

参数

day_of_month

1 ~ 31 之间的整数，作为 *date* 中月中某一天字段的新值（以本地时间计）。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化之前，该方法什么都不返回。

Date.setFullYear()

JavaScript 1.2;JScript 3.0;ECMAScript v1

设置年份，也可以设置月份和天

摘要

date.setFullYear(year)

date.setFullYear(year, month)

date.setFullYear(year, month, day)

参数

year

在 *date* 中设置的年份，用本地时间表示。该参数应该是一个包含世纪值的完整年份，如 1999，而不仅是年份的缩写，如 99。

month

可选的整数，在 0 ~ 11 之间，用作 *date* 的月份字段的新值（以本地时间计）。

day

可选的整数，在 1 ~ 31 之间，用作 *date* 的天数字段的新值（以本地时间计）。

返回值

调整过的日期的内部毫秒表示。

Date.setHours()

JavaScript 1.0;JScript 1.0;ECMAScript v1

设置 Date 对象的小时字段、分钟字段、秒字段和毫秒字段

摘要

date.setHours(hours)

date.setHours(hours, minutes)

date.setHours(hours, minutes, seconds)

```
date.setHours(hours, minutes, seconds, millis)
```

参数

hours

0（午夜）到 23（晚上 11 点）之间的整数，用作 *date* 的小时字段的新值（以本地时间计）。

minutes

可选的整数，在 0 ~ 59 之间，用作 *date* 的分钟字段的新值（以本地时间计）。ECMAScript 标准化前，不支持该参数。

seconds

可选的整数，在 0 ~ 59 之间，用作 *date* 的秒字段的新值（以本地时间计）。ECMAScript 标准化前，不支持该参数。

millis

可选的整数，在 0 ~ 999 之间，用作 *date* 的毫秒字段的新值（以本地时间计）。ECMAScript 标准化前，不支持该参数。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化前，该方法不返回值。

Date.setMilliseconds() JavaScript 1.2; JScript 3.0; ECMAScript v1

设置 Date 对象的毫秒字段

摘要

```
date.setMilliseconds(millis)
```

参数

millis

用于设置 *date* 的毫秒字段，用本地时间表示。该参数是 0 ~ 999 之间的整数。

返回值

调整过的日期的内部毫秒表示。

Date.setMinutes()

JavaScript 1.0;JScript 1.0;ECMAScript v1

设置 Date 对象的分钟字段和秒字段

摘要

```
date.setMinutes(minutes)
date.setMinutes(minutes, seconds)
date.setMinutes(minutes, seconds, millis)
```

参数

minutes

0 ~ 59 之间的整数，用于设置 Date 对象 *date* 的分钟字段（以本地时间计）。

seconds

可选的整数，在 0 ~ 59 之间，用做 *date* 的秒字段的新值（以本地时间计）。

ECMAScript 标准化前，不支持该参数。

millis

可选的整数，在 0 到 999 之间，用作 *date* 的毫秒字段的新值（以本地时间计）。

ECMAScript 标准化前，不支持该参数。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化前，该方法不返回值。

Date.setMonth()

JavaScript 1.0;JScript 1.0;ECMAScript v1

设置 Date 对象的月份字段和天字段

摘要

```
date.setMonth(month)
date.setMonth(month, day)
```

参数

month

0（一月）到 11（十二月）之间的整数，用于设置 Date 对象 *date* 的月份字段。

注意，月份从 0 开始编号，而月中的某一天则从 1 开始编号。

day

可选的整数，在 1 ~ 31 之间，用做 *date* 的天数字段的新值（以本地时间计）。ECMAScript 标准化前，不支持该参数。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化前，该方法不返回值。

Date.setSeconds() JavaScript 1.0; JScript 1.0; ECMAScript v1

设置 Date 对象的秒字段和毫秒字段

摘要

`date.setSeconds(seconds)`

`date.setSeconds(seconds, millis)`

参数

seconds

0 ~ 59 之间的一个整数，用于设置 Date 对象 *date* 的秒字段。

millis

可选的整数，在 0 ~ 999 之间，用做 *date* 的毫秒字段的新值（以本地时间计）。ECMAScript 标准化前，不支持该参数。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化前，该方法不返回值。

Date.setTime() JavaScript 1.0; JScript 1.0; ECMAScript v1

以毫秒设置 Date 对象

摘要

`date.setTime(milliseconds)`

参数

milliseconds

要设置的日期和时间距 GMT 时间 1970 年 1 月 1 日午夜之间的毫秒数。这种类型的毫秒值可以传递给 `Date()` 构造函数，可以通过调用 `Date.UTC()` 和 `Date.parse()` 方法获得该值。以毫秒形式表示日期可以使它独立于时区。

返回值

参数 *milliseconds*。在 ECMAScript 标准化前，该方法不返回值。

Date.setUTCDate() JavaScript 1.2;JScript 3.0;ECMAScript v1

设置一个月中的某一天（世界时）

摘要

date.setUTCDate(day_of_month)

参数

day_of_month

要给 *date* 设置的一个月中的某一天，用世界时表示。该参数是 1 ~ 31 之间的整数。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCFullYear() JavaScript 1.2;JScript 3.0;ECMAScript v1

设置年份、月份和天（世界时）

摘要

date.setUTCFullYear(year)

date.setUTCFullYear(year, month)

date.setUTCFullYear(year, month, day)

参数

year

要给 *date* 设置的年份值，用世界时表示。该参数应该是含有世纪值的完整年份，如 1999，而不只是缩写的年份值，如 99。

month

可选的整数，在 0 ~ 11 之间，用作 *date* 的月份字段的新值（以世界時計）。

day

可选的整数，在 1 ~ 31 之间，用作 *date* 的天字段的新值（以世界時計）。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCHours() JavaScript 1.2;JScript 3.0;ECMAScript v1

设置 Date 对象的小时字段、分钟字段、秒字段和毫秒字段（世界时）

摘要

```
date.setUTCHours(hours)
```

```
date.setUTCHours(hours, minutes)
```

```
date.setUTCHours(hours, minutes, seconds)
```

```
date.setUTCHours(hours, minutes, seconds, millis)
```

参数

hours

要设置的 *date* 的小时字段的值，用世界时表示。该参数应该是 0（午夜）到 23（晚上 11 点）之间的一个整数。

minutes

可选的整数，在 0 ~ 59 之间，用作 *date* 的分钟字段的新值（以世界時計）。

seconds

可选的整数，在 0 ~ 59 之间，用作 *date* 的秒字段的新值（以世界時計）。

millis

可选的整数，在 0 ~ 999 之间，用作 *date* 的毫秒字段的新值（以世界時計）。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCMilliseconds() JavaScript 1.2;JScript 3.0;ECMAScript v1

设置 Date 对象的毫秒字段（世界时）

摘要

```
date.setUTCMilliseconds(millis)
```

参数

millis

要设置的 *date* 的毫秒字段的值，用世界时表示。该参数是 0 ~ 999 之间的整数。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCMinutes() JavaScript 1.2; JScript 3.0; ECMAScript v1

设置 Date 对象的分钟字段和秒字段（世界时）

摘要

```
date.setUTCMinutes(minutes)
```

```
date.setUTCMinutes(minutes, seconds)
```

```
date.setUTCMinutes(minutes, seconds, millis)
```

参数

minutes

要设置的 *date* 的分钟字段的值，用世界时表示。该参数应该是 0 ~ 59 之间的一个整数。

seconds

可选的整数，在 0 ~ 59 之间，用作 *date* 的秒字段的新值（以世界時計）。

millis

可选的整数，在 0 ~ 999 之间，用作 *date* 的毫秒字段的新值（以世界時計）。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCMonth() JavaScript 1.2; JScript 3.0; ECMAScript v1

设置 Date 对象的月份字段和天数字段（世界时）

摘要

```
date.setUTCMonth(month)
```

```
date.setUTCMonth(month, day)
```

参数

month

要设置的 *date* 的月份字段的值，用世界时表示。该参数是 0（一月）到 11（十二月）之间的整数。注意，月份从 0 开始编码，而月中的某一天则从 1 开始编码。

day

可选的整数，在 1 ~ 31 之间，用作 *date* 的天字段的新值（以世界時計）。

返回值

调整过的日期的内部毫秒表示。

Date.setUTCSeconds() JavaScript 1.2; JScript 3.0; ECMAScript v1

设置 Date 对象的秒字段和毫秒字段（世界时）

摘要

date.setUTCSeconds(seconds)

date.setUTCSeconds(seconds, millis)

参数

seconds

要设置的 *date* 的秒字段的值，用世界时表示。该参数应该是 0 ~ 59 之间的一个整数。

millis

可选的整数，在 0 ~ 999 之间，用作 *date* 的毫秒字段的新值（以世界時計）。

返回值

调整过的日期的内部毫秒表示。

Date.setYear() JavaScript 1.0; JScript 1.0; ECMAScript v1; ECMAScript v3 反对使用

设置 Date 对象的年份字段

摘要

date.setYear(year)

参数

year

要设置的 Date 对象 *date* 的年份字段的值，是一个整数。如果这个值在 0 ~ 99 之间，包括 0 和 99，将给它加上 1900，作为 1900 ~ 1999 间的值处理。

返回值

调整过的日期的毫秒表示。在 ECMAScript 标准化前，该方法不返回值。

描述

方法 `setYear()` 可以设置指定的 Date 对象的年份字段，对于 1900 ~ 1999 之间的年份，带有特殊的行为。

从 ECMAScript v3 起，JavaScript 实现不再要求使用该函数，而使用 `setFullYear()` 函数代替它。

Date.toString() JavaScript 1.5; JScript 5.5; ECMAScript v3

返回 Date 对象日期部分作为字符串

摘要

`date.toString()`

返回值

date 的日期部分的字符串表示，由实现决定，人们可以读懂，以本地时间表示。

参阅

`Date.toLocaleDateString()`、`Date.toLocaleString()`、`Date.toLocaleTimeString()`、`Date.toString()`、`Date.toTimeString()`

Date.toGMTString() JavaScript 1.0; JScript 1.0; ECMAScript v1; ECMAScript v3 反对使用

将 Date 转换为世界时字符串

摘要

`date.toGMTString()`

返回值

Date 对象 *date* 所指定的日期和时间的字符串表示。这个日期在转换成字符串之前由本地时区转换成了 GMT 时区。

描述

不赞成使用方法 `toGMTString()`，而赞同使用 `Date.toUTCString()`。

从 ECMAScript v3 起，JavaScript 的实现不再要求使用该函数，而用 `toUTCString()` 代替它。

参阅

`Date.toUTCString()`

`Date.toLocaleDateString()` JavaScript 1.5;JScript 5.5;ECMAScript v3

返回 Date 对象的日期部分作为本地已格式化的字符串

摘要

`date.toLocaleDateString()`

返回值

date 的日期部分的字符串表示，由实现决定，人们可以读懂，以本地时间表示，根据本地规约格式化。

参阅

`Date.toString()`、`Date.toLocaleString()`、`Date.toLocaleTimeString()`、`Date.toString()`、`Date.getTimeString()`

`Date.toLocaleString()` JavaScript 1.0;JScript 1.0;ECMAScript v1

将 Date 转换为本地已格式化的字符串

摘要

`date.toLocaleString()`

返回值

Date 对象 *date* 指定的日期和时间的字符串表示。该日期和时间用本地时间区表示，根据本地规约格式化。

用法

方法 `toLocaleDateString()` 可以将日期转换成用本地时间区表示的字符串。该方法的日期和时间格式还使用地方规约, 所以在不同平台上以及不同国家之间, 日期和时间的格式都有所不同。它返回的字符串格式通常都是用户想要的日期和时间格式。

参阅

`Date.toLocaleDateString()`、`Date.toLocaleTimeString()`、`Date.toString()`、`Date.toUTCString()`

`Date.toLocaleTimeString()` JavaScript 1.5;JScript 5.5;ECMAScript v3

返回 `Date` 对象的时间部分作为本地已格式化的字符串

摘要

`date.toLocaleTimeString()`

返回值

`date` 的时间部分的字符串表示, 由实现决定, 人们可以读取的, 以本地时区表示, 根据本地规约格式化。

参阅

`Date.toDateString()`、`Date.toLocaleDateString()`、`Date.toLocaleString()`、`Date.toString()`、`Date.toTimeString()`

`Date.toString()` JavaScript 1.0;JScript 1.0;ECMAScript v1

将 `Date` 转换为字符串 覆盖 `Object.toString()`

摘要

`date.toString()`

返回值

`date` 的字符串表示, 是人们可以读取的, 用本地时间表示。

描述

方法 `toString()` 返回一个人们可以读取的、由实现决定的日期的字符串表示。它与 `toUTCString()` 的不同, `toString()` 以本地时间表示日期。而它与 `toLocaleString()` 的不同之处在于它不使用地方规约采用的形式表示日期和时间。

参阅

`Date.parse()`、`Date.toDateString()`、`Date.toLocaleString()`、`Date.toTimeString()`、`Date.toUTCString()`

`Date.toTimeString()` JavaScript 1.5; JScript 5.5; ECMAScript v3

返回 `Date` 对象日期部分作为字符串

摘要

`date.toTimeString()`

返回值

`date` 的时间部分的字符串表示，由实现决定，人们可以读取，以本地时间表示。

参阅

`Date.toString()`、`Date.toDateString()`、`Date.toLocaleDateString()`、`Date.toLocaleString()`、`Date.toLocaleTimeString()`

`Date.toUTCString()` JavaScript 1.2; JScript 3.0; ECMAScript v1

将 `Date` 转换为字符串（世界时）

摘要

`date.toUTCString()`

返回值

`date` 的字符串表示，人们可以读取，用世界时表示。

参阅

`Date.toLocaleString()`、`Date.toString()`

`Date.UTC()` JavaScript 1.0; JScript 1.0; ECMAScript v1

将 `Date` 规范转换成毫秒数

摘要

`Date.UTC(year, month, day, hours, minutes, seconds, ms)`

参数

year

四位数表示的年份值。如果该参数在 0 ~ 99 之间（包括 0 和 99），它将加上 1900，作为 1900 ~ 1999 之间的年份处理。

month

月份值，是 0（一月）到 11（十二月）之间的整数。

day

一个月中的某一天，是 1 ~ 31 之间的整数。注意，该参数的最小值是 1，而其他参数的最小值则是 0。该参数是可选的。

hours

小时值，是 0（午夜）到 23（晚上 11 点）之间的整数。该参数是可选的。

minutes

分钟值，是 0 ~ 59 之间的整数。该参数是可选的。

seconds

秒值，是 0 ~ 59 之间的整数。该参数是可选的。

ms

毫秒数。该参数是可选的。在 ECMAScript 标准化前，忽略该参数。

返回值

指定的世界时的毫秒表示。简而言之，该方法返回指定的时间距 GMT 时间 1970 年 1 月 1 日午夜的毫秒数。

描述

`Date.UTC()` 是一种静态方法，它通过构造函数 `Date()` 调用，而不是通过某个 `Date` 对象调用。

`Date.UTC()` 方法的参数指定日期和时间，它们都是 UTC 时间，处于 GMT 时区。指定的 UTC 时间将转换成毫秒的形式，这样构造函数 `Date()` 和方法 `Date.setTime()` 就可以使用它了。

`Date.UTC()` 能接受的日期和时间格式，构造函数 `Date()` 也可以接受。区别在于构造函数 `Date()` 假定这些参数是本地时间，而 `Date.UTC()` 却假定它们是世界时（GMT 时间）。要创建使用 UTC 时间规约的 `Date` 对象，可以使用如下的代码：

```
d = new Date(Date.UTC(1996, 4, 8, 15, 39));
```

参阅

Date、Date.parse()和 Date.setTime()

Date.valueOf()

JavaScript 1.1;ECMAScript v1

将 Date 转换成毫秒表示

摘要

`date.valueOf()`

返回值

`date` 的毫秒表示。返回值和方法 `Date.getTime()` 返回的值相等。

decodeURI()

JavaScript 1.5;JScript 5.5;ECMAScript v3

URI 中未转义的字符

摘要

`decodeURI(uri)`

参数

`uri`

一个字符串，含有编码的 URI 或要其他要解码的文本。

返回值

`uri` 的副本，其中十六进制的转义序列被它们表示的字符替换了。

抛出

URIError

说明 `uri` 中的一个或多个转义序列被错误地格式化，不能被正确解码。

描述

`decodeURI()` 是一个全局函数，它返回参数 `uri` 解码后的副本。它将保留 `encodeURIComponent()` 方法执行的编码操作，详见该函数。

参阅

`decodeURIComponent()`、`encodeURI()`、`encodeURIComponent()`、`escape()`、`unescape()`

`decodeURIComponent()` JavaScript 1.5; JScript 5.5; ECMAScript v3

URI 组件中的未转义字符

摘要

`decodeURI(s)`

参数

s 一个字符串，含有编码 URI 组件或其他要解码的文本。

返回值

s 的副本，其中十六进制的转义序列被它们所表示的字符替换。

抛出

`URIError`

说明 *s* 中的一个或多个转义序列被错误地格式化，不能被正确解码。

描述

`decodeURIComponent()` 是一个全局函数，它返回参数 *s* 解码后的副本。它将保留 `encodeURIComponent()` 方法执行的编码操作，详见该函数的参考页。

参阅

`decodeURI()`、`encodeURI()`、`encodeURIComponent()`、`escape()`、`unescape()`

`encodeURI()` JavaScript 1.5; JScript 5.5; ECMAScript v3

URI 中的转义字符

摘要

`encodeURI(uri)`

参数

uri

一个字符串，含有 URI 或其他要编码的文本。

返回值

`uri` 的副本，其中某些字符被十六进制的转义序列替换了。

抛出

`URIError`

说明 `uri` 中含有格式化错误的 Unicode 替代对，不能被编码。

描述

`encodeURIComponent()` 是全局函数，返回参数 `uri` 的编码副本。ASCII 的字母和数字不编码，此外下面的 ASCII 标点符号也不编码：

```
! , ' ~ * ( )
```

因为 `encodeURIComponent()` 的目的是给 URI 进行完整的编码，所以以下在 URI 中具有特殊含义的 ASCII 标点符号也不转义：

```
; : @ > - + = , #
```

`uri` 中的其他字符都将转换成它的 UTF-8 编码字符，然后用十六进制的转义序列（形式为 `%xx`）对生成的一个、两个或三个字节的字符编码，用它们替换 `uri` 中原有的字符。在这种编码模式中，ASCII 字符由一个 `%xx` 转义字符替换，在 `\u0080` 到 `\u07ff` 之间编码的字符由两个转义序列替换，其他的 16 位 Unicode 字符由三个转义序列替换。

如果使用该方法编码 URI，应该确保 URI 组件（如查询字符串）中不含有 URI 分隔符，如 `?` 和 `#`。如果组件中含有这些符号，应该用 `encodeURIComponent()` 方法分别对各个组件编码。

用方法 `decodeURI()` 可以对该方法进行解码操作。在 ECMAScript v3 之前，可以用 `escape()` 和 `unescape()` 方法（反对使用）执行相似的编码解码操作。

例子

```
// 返回 http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURIComponent('http://www.isp.com/app.cgi?arg1=1&arg2=hello world');
encodeURIComponent("\u00a9"); // 版权字符编码为 %C2%A9
```

参阅

`decodeURI()`、`decodeURIComponent()`、`encodeURIComponent()`、`escape()`、`unescape()`

encodeURIComponent() JavaScript 1.5;JScript 5.5;ECMAScript v3

转义 URI 组件中的字符

摘要

`encodeURIComponent (s)`

参数

s 一个字符串，含有 URI 的一部分或其他要编码的文本。

返回值

s 的副本，其中某些字符被十六进制的转义序列替换了。

抛出

URIError

说明 *s* 中含有格式化错误的 Unicode 替代对，不能被编码。

描述

`encodeURIComponent()` 是全局函数，返回参数 *s* 的编码副本。ASCII 的字母和数字不编码，此外下面的 ASCII 标点符号也不编码：

`- _ . ! ~ * ' ()`

其他字符（像 “/”、“:”、“#” 这样用于分隔 URI 各种组件的标点符号），都由一个或多个十六进制的转义序列替换。关于使用的编码模式，请参阅 “`encodeURIComponent()`” 的描述。

注意 `encodeURIComponent()` 和 `encodeURIComponent()` 之间的差别，前者假定它的参数是 URI 的一部分（如协议、主机名、路径或查询字符串）。因此，它将转义用于分隔 URI 各个部分的标点符号。

例子

```
encodeURIComponent('hello world?'); // 返回 hello%20world%3F
```

参阅

`decodeURI()`、`decodeURIComponent()`、`encodeURIComponent()`、`escape()`、`unescape()`

Error

JavaScript 1.5;JScript 5.5;ECMAScript v3

普通异常

继承 Object 类

构造函数

`new Error()``new Error(message)`

参数

`message`

提供异常的详细信息的错误消息，可选。

返回值

新构造的 Error 对象。如果指定了参数 `message`，该 Error 对象将它作为 `message` 属性的值；否则，它将用实现定义的默认字符串作为该属性的值。如果把 Error 构造函数当作函数调用时不使用 `new` 运算符，它的行为与使用 `new` 运算符调用时一样。

属性

`message`

提供异常详细信息的错误消息。该属性存放传递给构造函数的字符串，或实现定义的默认字符串。

`name`

声明异常类型的字符串。对于 Error 类的实例和所有子类来说，该属性声明了用于创建实例的构造函数名。

方法

`toString()`

返回一个表示 Error 对象的字符串，该字符串由实现定义。

描述

Error 类的实例表示错误或异常，通常与 `throw` 语句和 `try/catch` 语句一起使用。属性 `name` 声明了异常的类型，`message` 属性可提供人们能够读取的异常的详细信息。

JavaScript 解释器从不直接抛出 Error 对象，而是抛出 Error 子类（如 `SyntaxError` 或 `RangeError`）的实例。在代码中，你会发现抛出 Error 对象指示异常非常方便，或者也可以用原始字符串或数字的形式抛出错误消息或错误代码。

注意, ECMAScript 标准为 Error 类定义了 toString() 方法 (Error 的所有子类都继承了该方法), 但并不要求该方法返回含有 message 属性的字符串。因此, 不能期望 toString() 方法可以把 Error 对象转换成人们可以读懂的字符串。要把错误消息显示给用户, 应该明确地使用 Error 对象的 name 属性和 message 属性。

例子

可以用下列代码指示一个异常:

```
function factorial(x) {  
    if (x < 0) throw new Error('factorial: x must be >= 0');  
    if (x <= 1) return 1; else return x * factorial(x-1);  
}
```

如果捕捉到了一个异常, 可以用下列代码把它显示给用户 (这段代码使用了客户端 Window.alert() 方法):

```
try { /*(&,* an error is thrown here */  
catch(e) {  
    if (e instanceof Error) { // 它是 Error 或子类的一个实例吗?  
        alert(e.name + ': ' + e.message);  
    }  
}
```

参阅

EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError

Error.message

JavaScript 1.5;JScript 5.5;ECMAScript v3

人们可以读取的错误消息

摘要

error.message

描述

Error 对象 (或 Error 子类的实例) 的 message 属性用于存放包含发生的错误或异常的详细情况的字符串, 该字符串是人们可以读取的。如果传递给 Error() 构造函数一个消息参数, 该消息将成为 message 属性的值。如果没有消息参数传递给 Error() 参数, Error 对象将继承实现为该属性定义的默认值 (可能是空串)。

Error.name

JavaScript 1.5;JScript 5.5;ECMAScript v3

错误的类型

摘要

`error.name`

描述

Error 对象（或 Error 子类的实例）的 `name` 属性声明了发生的错误或异常的类型。所有 Error 对象都从它们的构造函数中继承这一属性。该属性的值与构造函数名相同。因此，`SyntaxError` 对象的 `name` 属性值为 “SyntaxError”，`EvalError` 对象的 `name` 属性为 “EvalError”。

Error.toString()

JavaScript 1.5;JScript 5.5;ECMAScript v3

把 Error 对象转换成字符串

覆盖 `Object.toString()`

摘要

`error.toString()`

返回值

实现定义的字符串。ECMAScript 标准除了规定该方法的返回值是字符串外，没有再做其他规定。尤其是，它不要求返回的字符串包含错误名或错误消息。

escape()

 JavaScript 1.0;JScript 1.0;ECMAScript v1;ECMAScript v3 反对使用

对字符串编码

摘要

`escape(s)`

参数

s 要被转义或编码的字符串。

返回值

编码了的 *s* 的副本，其中某些字符被替换成了十六进制的转义序列。

描述

`escape()` 是全局函数。它返回一个含有 `s` 的编码版本的新字符串。`s` 自身并没有被修改。

在 `escape()` 返回的字符串中，除了 ASCII 字母、数字和标点符号 `@`、`*`、`_`、`+`、`-`、`.` 和 `\` 之外，所有字符都由形为 `%xx` 或 `%uxxxxx` (`x` 表示十六进制的数字) 的转义序列替代。从 `\u0000` 到 `\u00ff` 的 Unicode 字符由转义序列 `%xx` 替代，其他所有 Unicode 字符由 `%uxxxxx` 序列替代。

使用函数 `unescape()` 可以对 `escape()` 编码的字符串进行解码。

在客户端 JavaScript 中，`escape()` 通常是对 cookie 值编码，它们含有的标点符号具有限制。参阅客户端参考手册部分的“`Document.cookie`”参考页。

虽然 ECMAScript 的第一个版本标准化了 `escape()` 函数，但是 ECMAScript v3 反对使用该方法，并从标准中删除了它。ECMAScript 的实现可能实现了该函数，但它不是必需的。在 JavaScript 1.5 和 JScript 5.5 及其后的版本中，应该用 `encodeURIComponent()` 和 `encodeURIComponent()` 代替 `escape()`。

例子

```
escape("Hello World!"); // 返回 "Hello%20World%21"
```

参阅

`encodeURIComponent()`、`encodeURIComponent()`、`String`、`unescape()` 和客户端参考手册的 `Document.cookie`

eval()

JavaScript 1.0;JScript 1.0;ECMAScript v1

执行字符串中的 JavaScript 代码

摘要

```
eval(code)
```

参数

`code`

字符串，含有要计算的 JavaScript 表达式或要执行的语句。

返回值

计算 `code` 得到的值（如果存在的话）。

抛出

`SyntaxError`

说明 `code` 中没有合法的 JavaScript 表达式或语句。

`EvalError`

说明非法调用了 `eval()`，例如使用的标识符不是“eval”。参阅下面描述的对该函数的限制。

其他异常

如果传递给 `eval()` 的 JavaScript 代码生成了一个异常，`eval()` 将把那个异常传递给调用者。

描述

`eval()` 是全局方法，它将执行含有 JavaScript 代码的字符串。如果 `code` 含有一个表达式，`eval()` 将计算这个表达式，并返回它的值。如果 `code` 含有一个或多个 JavaScript 语句，`eval()` 将执行这些语句，如果最后一个语句有返回值，它还会返回这个值。如果 `code` 没有返回任何值，`eval()` 将返回 `undefined`。最后，如果 `code` 抛出了一个异常，`eval()` 将把这个异常传递给调用者。

虽然 `eval()` 给 JavaScript 语言提供了非常强大的功能，但在实际程序中极少用它。常见的用法是编写作为递归的 JavaScript 解释器的程序，或者编写动态生成并计算 JavaScript 代码的程序。

大部分使用字符串参数的 JavaScript 函数和方法都会接受其他类型的参数，在继续操作之前把这些参数值转换成字符串。但 `eval()` 的行为不是这样。如果 `code` 参数不是原始的字符串，它将不作任何改变地返回。所以，要注意，当打算传递给 `eval()` 原始字符串值时，不要粗心地给它传递 `String` 对象。

考虑到实现的效率，ECMAScript v3 标准给 `eval()` 方法的使用加上了一条与众不同的限制。如果试图覆盖 `eval` 属性或把 `eval()` 方法赋予另一个属性，并通过该属性调用它，则 ECMAScript 实现允许抛出一个 `EvalError` 异常。

例子

```
eval("1+2"); // 返回 3
```

```
// 这段代码用客户端 JavaScript 的方法提示用户输入表达式并显示出计算结果。  
// 详见客户端的方法 window.alert() 和 window.prompt()。  
try {  
    alert("Result: " + eval(prompt("Enter an expression: ", "")));  
}  
catch(exception){  
    alert(exception);  
}  
  
var myeval = eval; // 抛出 EvalError 异常。  
myeval("1+2");    抛出 EvalError 异常。
```

EvalError

JavaScript 1.5; JScript 5.5; ECMAScript v3

在不正确使用 eval() 时抛出

继承 Error 类

构造函数

```
new EvalError()  
new EvalError(message)
```

参数

message

提供异常的详细信息的错误消息，可选。如果设置了该参数，它将用作 EvalError 对象的 message 属性的值。

返回值

新构造的 EvalError 对象。如果指定了参数 message，Error 对象将用它作为 message 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果把 EvalError() 构造函数当作函数调用且不带有 new 运算符，它的行为与使用 new 运算符调用时一样。

属性

message

提供异常的详细信息的错误消息。该属性存放传递给构造函数的字符串，或存放实现定义的默认字符串。详见“Error.message”。

name

声明异常类型的字符串。所有 EvalError 对象的 name 属性都继承值“EvalError”。

描述

当在其他名称下调用全局函数 `eval()` 时, `EvalError` 类的一个实例就会被抛出。关于调用 `eval()` 函数的限制, 请参阅“`eval()`”。关于抛出和捕捉异常的细节, 请参阅“`Error`”。

参阅

`Error`、`Error.message`、`Error.name`

Function

JavaScript 1.0;JScript 1.0;ECMAScript v1

JavaScript 的函数

继承 `Object` 类

摘要

```
function functionname(argument_name_list)    // 函数定义语句
{
    body
}
function (argument_name_list) { body; }      // 未命名的函数直接量
                                              // JavaScript 3.2 引入
functionname(argument_value_list)           // 函数调用
```

构造函数

```
new Function(argument_names..., body)    // JavaScript 1.1 和其后的版本支持它
```

参数

`argument_names...`

任意多个字符串参数, 每个字符串命名一个或多个要创建的 `Function` 对象的参数。

`body`

一个字符串, 指定函数的主体, 可以含有任意多条 JavaScript 语句, 这些语句之间用分号隔开, 可以给该构造函数引用前面的参数设置的任何参数名。

返回值

新创建的 `Function` 对象。调用该函数, 将执行 `body` 指定的 JavaScript 代码。

抛出

`SyntaxError`

说明在参数 `body` 或某个 `argument_names` 参数中存在 JavaScript 语法错误。

属性

`arguments[]`

一个参数数组，元素是传递给函数的参数。反对使用该属性。

`caller`

对调用当前函数的 `Function` 对象的引用，如果当前函数由顶层代码调用，这个属性的值为 `null`。反对使用该属性。

`length`

在声明函数时指定的命名参数的个数。

`prototype`

一个对象，用于构造函数，这个对象定义的属性和方法由构造函数创建的所有对象共享。

方法

`apply()`

将函数作为指定对象的方法来调用，传递给它的是指定的参数数组。

`call()`

将函数作为指定对象的方法来调用，传递给它的是指定的参数。

`toString()`

返回函数的字符串表示。

描述

函数是 JavaScript 的一种基本数据类型。本书的第七章解释了如何定义和使用函数，第八章介绍了函数方法、构造函数以及 `prototype` 属性等相关主题。要了解详细情况，请参看这两章。注意，虽然可以用这里介绍的 `Function()` 构造函数创建函数对象，但这样做效率不高，在大多数情况下，建议使用函数定义语句或函数直接量来定义函数。

在 JavaScript 1.1 及以后版本中，函数主体会被自动地给予一个局部变量 `arguments`，它引用一个 `Arguments` 对象。该对象是一个数组，元素是传递给函数的参数值。不要将这一属性和上面介绍的反对使用的属性 `arguments[]` 相混淆。详见“`Arguments`”的参考页。

参阅

Arguments、第七章和第八章

Function.apply()

JavaScript 1.2;JScript 5.5;ECMAScript v3

将函数作为一个对象的方法调用

摘要

```
function.apply(thisobj, args)
```

参数

thisobj

调用 *function* 的对象。在函数主体中，*thisobj* 是关键字 *this* 的值。

args

一个数组，它的元素是要传递给函数 *function* 的参数值。

返回值

调用函数 *function* 的返回值。

抛出

`TypeError`

如果调用该函数的对象不是函数，或参数 *args* 不是数组和 Arguments 对象，则抛出该异常。

描述

`apply()` 将指定的函数 *function* 作为对象 *thisobj* 的方法来调用，传递给它的是存放在数组 *args* 中的参数，返回的是调用 *function* 的返回值。在函数体内，关键字 *this* 引用 *thisobj* 对象。

参数 *args* 必须是数组或 Arguments 对象。如果想单独指定传递给函数的参数，而不是指定数组元素，请使用 `Function.call()` 方法。

例子

```
// 在对象上应用默认的 Object.toString() 方法。  
// 该对象用该方法的版本覆盖了它。注意，没有参数。  
Object.prototype.toString.apply(0);
```

```
// 使用数组中用来查找最大元素的方法来调用 Math.max()。  
// 注意本例中第一个参数没有什么作用。  
var data = [1,2,3,4,5,6,7,8];  
Math.max.apply(null, data);
```

参阅

Function.call()

Function.arguments[] JavaScript 1.0;JScript 1.0;ECMAScript v1;ECMAScript v3 反对使用
传递给函数的参数

摘要

```
function.arguments[i]  
function.arguments.length
```

描述

Function对象的arguments属性是一个参数数组，它的元素是传递给函数的参数。只
在执行函数时，它才被定义。arguments.length声明的是数组中的元素个数。

反对使用该属性，赞成使用Arguments对象。虽然ECMAScript v1支持Function.
arguments属性，但ECMAScript v3删除了它，遵守该标准的实现不再支持该属性。
因此，在新的JavaScript代码中，不再应该使用它。

参阅

Arguments

Function.call() JavaScript 1.5;JScript 5.5;ECMAScript v3
将函数作为对象的方法调用

摘要

```
function.call(thisobj, args...)
```

参数

thisobj

调用 *function* 的对象。在函数主体中，*thisobj* 是关键字 *this* 的值。

args...

任意多个参数，这些参数将传递给函数 *function*。

返回值

调用函数 *function* 的返回值。

抛出

`TypeError`

如果调用该函数的对象不是函数，则抛出该异常。

描述

`call()` 将指定的函数 *function* 作为对象 *thisobj* 的方法来调用，把参数列表中 *thisobj* 后的参数传递给它，返回值是调用函数后的返回值。在函数体内，关键字 *this* 引用 *thisobj* 对象。

如果指定数组中传递给函数的参数，请使用 `Function.apply()` 方法。

例子

```
// 在对象上调用默认的 Object.toString() 方法。该对象用该方法的版本覆盖了自己。  
// 注意没有参数。  
Object.prototype.toString.call(0);
```

参阅

`Function.apply()`

Function.caller

JavaScript 1.0;JScript 2.0;ECMAScript 反对使用

调用当前函数的函数

摘要

`function.caller`

描述

在 JavaScript 的早期版本中，`Function` 对象的 `caller` 属性是对调用当前函数的函数的引用。如果该函数是从 JavaScript 程序的顶层调用的，`caller` 的值就为 `null`。该属性只能在函数内部使用（例如，`caller` 属性只有在函数执行时才会有定义）。

`Function.caller` 属性不属于 ECMAScript 标准。在遵守该标准的实现中，不需要该属性。不应该再使用它。

Function.length

JavaScript 1.1;JScript 2.0;ECMAScript v1

已声明的参数的个数

摘要*function.length***描述**

函数的 `length` 属性在定义函数时声明已命名的参数的个数。实际调用函数时，传递给它的参数个数既可以比它多，也可以比它少。不要混淆了 `Function` 对象和 `Arguments` 对象的 `length` 属性，后者声明的是实际传递给函数的参数个数。参阅“`Arguments.length`”中的示例。

参阅`Arguments.length`

Function.prototype

JavaScript 1.1;JScript 2.0;ECMAScript v1

对象类的原型

摘要*function.prototype***描述**

属性 `prototype` 是在函数作为构造函数时使用的。它引用的是作为整个对象类的原型的对象。由这个构造函数创建的任何对象都会继承属性 `prototype` 引用的对象的所有属性。

要了解 JavaScript 中的构造函数、`prototype` 属性和类定义的完整讨论，请参阅第八章。

Bug

JavaScript 1.1 要求，在给原型对象赋值之前，要先使用一次该构造函数。

参阅

第八章

Function.toString()

JavaScript 1.0;JScript 2.0;ECMAScript v1

把函数转换成字符串

摘要

```
function.toString()
```

返回值

表示函数的字符串。

抛出

`TypeError`

如果调用该函数的对象不是 `Function`，则抛出该异常。

描述

`Function`对象的方法 `toString()` 可以以一种与实现相关的方式将函数转换成字符串。在 Netscape 实现中，该方法返回一个含有有效 JavaScript 代码的字符串，即包括关键字 `function`、参数列表和函数的完整主体的代码。

Global

JavaScript 1.0;JScript 1.0;ECMAScript v1

全局对象

摘要

```
this
```

全局属性

全局对象不是一个类，所以下面的全局属性在自己名称下有单独的参考条目。也就是说，在“`undefined`”名下可以找到 `undefined` 属性的详细信息，而不是在“`Global. undefined`”下寻找。注意，所有顶层变量也都是全局对象的属性。

`Infinity` 表示正无穷大的数值。

`NaN` 非数字值。

`undefined` 未定义的值。

全局函数

全局对象是一个对象，而不是类。下面列出的全局函数不是任何对象的方法，它们的参考条目出现在函数名下。例如，在“`parseInt()`”下可以找到`parseInt()`函数的详细信息，在“`Global.parseInt()`”下就无法找到该函数的详细信息。

<code>decodeURI()</code>	对 <code>encodeURIComponent()</code> 转义的字符串解码。
<code>decodeURIComponent()</code>	对 <code>encodeURIComponent()</code> 转义的字符串解码。
<code>encodeURI()</code>	通过转义某些字符对 URI 编码。
<code>encodeURIComponent()</code>	通过转义某些字符对 URI 的组件编码。
<code>escape()</code>	用转义序列替换某些字符来对字符串编码。
<code>eval()</code>	计算 JavaScript 代码串，返回结果。
<code>isFinite()</code>	检测一个值是否是无穷大的数字。
<code>isNaN()</code>	检测一个值是否是非数字的值。
<code>parseFloat()</code>	从字符串解析一个数字。
<code>parseInt()</code>	从字符串解析一个整数。
<code>unescape()</code>	对用 <code>escape()</code> 编码的字符串解码。

全局对象

除了上面列出的全局属性和全局函数以外，全局对象还定义了引用 JavaScript 所有预定义对象的属性。除了 `Math` 外，这些属性都是定义类的构造函数，`Math` 引用的对象不是构造函数。

<code>Array</code>	构造函数 <code>Array()</code> 。
<code>Boolean</code>	构造函数 <code>Boolean()</code> 。
<code>Date</code>	构造函数 <code>Date()</code> 。
<code>Error</code>	构造函数 <code>Error()</code> 。
<code>EvalError</code>	构造函数 <code>EvalError()</code> 。
<code>Function</code>	构造函数 <code>Function()</code> 。
<code>Math</code>	对定义算术函数的对象的引用。

Number	构造函数 Number()。
Object	构造函数 Object()。
RangeError	构造函数 RangeError()。
ReferenceError	构造函数 ReferenceError()。
RegExp	构造函数 RegExp()。
String	构造函数 String()。
SyntaxError	构造函数 SyntaxError()。
TypeError	构造函数 TypeError()。
URIError	构造函数 URIError()。

描述

全局对象是预定义的对象，作为JavaScript的全局属性和全局函数的占位符。通过使用全局对象，可以访问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，所以它没有名字（之所以选择Global作为这个参考页的标题，只是为了方便组织，并不是说全局对象名为“Global”）。在顶层JavaScript代码中，可以用关键字this引用全局对象。但通常不必用这种方式引用全局对象，因为全局对象是作用域链的头，这意味着所有非限定性的变量和函数名都会作为该对象的属性来查询。例如，当JavaScript代码引用parseInt()函数时，它引用的是全局对象的parseInt属性。全局对象是作用域链的头，还意味着在顶层JavaScript代码中声明的所有变量都将成为全局对象的属性。

全局对象只是一个对象，而不是类。既没有Global()构造函数，也无法实例化一个新的全局对象。

当JavaScript代码嵌入一个特殊环境中时，全局对象通常具有环境特定的属性。实际上，ECMAScript标准没有规定全局对象的类型，JavaScript的实现或嵌入的JavaScript都可以把任意类型的对象作为全局对象，只要该对象定义了这个列出的基本属性和函数。例如，在客户端JavaScript中，全局对象是Window对象，表示运行JavaScript代码的Web浏览器窗口。

例子

在JavaScript核心语言中，全局对象的预定义属性都是不可枚举的，所以可以用for/in循环列出所有隐式或显式声明的全局变量，如下所示：

```
var variables =  
for(var name in this)  
    variables += name + " ";
```

参阅

客户端参考手册部分的 Window，第四章

Infinity

JavaScript 1.3;JScript 3.0;ECMAScript v1

表示无穷大的数字属性

摘要

Infinity

描述

Infinity是一个全局属性，它用来存放表示正无穷大的特殊的数值。用for/in循环不能枚举Infinity属性，用delete运算符不能删除它。注意，Infinity不是常量，它可以设为其他值。（但Number.POSITIVE_INFINITY是常量。）

参阅

isFinite()、NaN、Number.POSITIVE_INFINITY

isFinite()

JavaScript 1.2;JScript 3.0;ECMAScript v1

判断一个数字是否是有限的

摘要

isFinite(n)

参数

n 要检测的数字。

返回值

如果*n*是有限数字（或者可以转换为有限数字），那么返回值就是true。否则，如果

`n` 是 NaN (非数字), 或者是正、负无穷大的数, 则返回值就是 `false`。

参阅

Infinity、isNaN()、NaN、Number.NaN、Number.NEGATIVE_INFINITY、Number.POSITIVE_INFINITY

isNaN()

JavaScript 1.1;JScript 1.0;ECMAScript v1

检测非数字值

摘要

isNaN(*x*)

参数

x 要检测的值。

返回值

如果 *x* 是特殊的非数字值 NaN (或者能被转换为这样的值), 返回值就是 `true`。如果 *x* 是其他值, 返回值是 `false`。

描述

isNaN() 可以通过检测参数来判断值是否是 NaN, 该值表示一个非法的数字 (如被 0 除后得到结果)。这个函数是必需的, 因为把 NaN 与任何值 (包括它自身) 进行比较得到的结果都是 `false`, 所以要检测一个值是否是 NaN, 不能使用 `==` 或 `===` 运算符。

isNaN() 通常用于检测方法 parseFloat() 和 parseInt() 的结果, 以判断它们表示的是否是合法数字。也可以用 isNaN() 来检测算术错误, 如用 0 作除数。

例子

```
!isNaN(0);           // 返回 false
isNaN(0/0);          // 返回 true
isNaN(parseInt('3')); // 返回 false
isNaN(parseInt('hello')); // 返回 true
isNaN(3);            // 返回 false
isNaN('hello');      // 返回 true
isNaN(true);          // 返回 false
isNaN(undefined);    // 返回 true
```

参阅

isFinite()、NaN、Number.NaN、parseFloat()、parseInt()

Math

JavaScript 1.0;JScript 1.0;ECMAScript v1

算术函数和常量

摘要`Math.constant``Math.function()`**常量**`Math.E` 常量 e ，自然对数的底数。`Math.LN10` 10 的自然对数。`Math.LN2` 2 的自然对数。`Math.LOG10E` 以 10 为底的 e 的对数。`Math.LOG2E` 以 2 为底的 e 的对数。`Math.PI` 常量 π 。`Math.SQRT1_2` 2 的平方根除以 1。`Math.SQRT_2` 2 的平方根。**静态函数**`Math.abs()` 计算绝对值。`Math.acos()` 计算反余弦值。`Math.asin()` 计算反正弦值。`Math.atan()` 计算反正切值。`Math.atan2()` 计算从 X 轴到一个点的角度。`Math.ceil()` 对一个数上舍入。`Math.cos()` 计算余弦值。`Math.exp()` 计算 e 的指数。`Math.floor()` 对一个数下舍入。`Math.log()` 计算自然对数。

`Math.max()` 返回两个数中较大的一个。

`Math.min()` 返回两个数中较小的一个。

`Math.pow()` 计算 x^y 。

`Math.random()` 计算一个随机数。

`Math.round()` 舍入为最接近的整数。

`Math.sin()` 计算正弦值。

`Math.sqrt()` 计算平方根。

`Math.tan()` 计算正切值。

描述

`Math` 是一个对象，定义了引用有用的算术函数和常量的属性。`Math` 对象对这些函数和常量进行了分组，使用如下的语法就可以调用它们：

```
y = Math.sin(x);  
area = radius * radius * Math.PI;
```

`Math` 并不像 `Date` 和 `String` 那样是对象的类。没有 `Math()` 构造函数，像 `Math.sin()` 这样的函数只是函数，不是对象的方法。

参阅

`Number`

`Math.abs()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

计算绝对值

摘要

`Math.abs(x)`

参数

`x` 任意数。

返回值

`x` 的绝对值。

Math.acos()JavaScript 1.0;JScript 1.0;ECMAScript v1

计算反余弦值

摘要`Math.acos(x)`**参数** x -1.0 和 1.0 之间的数。**返回值**指定的值 x 的反余弦值。返回的是 0 到 π 之间的弧度值。

Math.asin()JavaScript 1.0;JScript 1.0;ECMAScript v1

计算反正弦值

摘要`Math.asin(x)`**参数** x -1.0 和 1.0 之间的数。**返回值**指定的值 x 的反正弦值。返回的是 $-\pi/2$ 到 $\pi/2$ 之间的弧度值。

Math.atan()JavaScript 1.0;JScript 1.0;ECMAScript v1

计算反正切值

摘要`Math.atan(x)`**参数** x 任意数。**返回值**指定的值 x 的反正切值。返回的是 $-\pi/2$ 到 $\pi/2$ 之间的弧度值。

Math.atan2()

JavaScript 1.0;JScript 1.0;ECMAScript v1

计算从 X 轴到一个点之间的角度

摘要

`Math.atan2(y, x)`

参数

x 指定点的 X 坐标。

y 指定点的 Y 坐标。

返回值

$-\pi$ 到 π 之间的值，是从 X 轴正向逆时针旋转到点 (*x*, *y*) 时经过的角度。

描述

函数 `Math.atan2()` 计算的是 y/x 的反正切值。参数 *y* 可以看作一个点的 Y 坐标，*x* 可以看作该点的 X 坐标。注意这个函数的参数顺序，Y 坐标在 X 坐标之前传递。

Math.ceil()

JavaScript 1.0;JScript 1.0;ECMAScript v1

对一个数上舍入

摘要

`Math.ceil(x)`

参数

x 任意数或表达式。

返回值

大于等于 *x*，并且与它最接近的整数。

描述

`Math.ceil()` 执行的是向上取整计算，它返回的是大于等于函数参数，并且与之最接近的整数。`Math.ceil()` 执行的操作不同于 `Math.round()`，`Math.ceil()` 总是上舍入，而 `Math.round()` 可以上舍入或下舍入到最接近的整数。还要注意，`Math.ceil()` 不会将负数舍入为更小的负数，而是向 0 舍入。

例子

```
a = Math.ceil(1.99);    // 结果为 2.0
b = Math.ceil(1.01);    // 结果为 2.0
c = Math.ceil(1.0);     // 结果为 1.0
d = Math.ceil(-1.99);   // 结果为 -1.0
```

Math.cos()

JavaScript 1.0;JScript 1.0;ECMAScript v1

计算余弦值

摘要

`Math.cos(x)`

参数

`x` 一个角的弧度值。要把角度转换成弧度，只需把角度值乘以 0.017453293 ($2\pi/360$) 即可。

返回值

指定的值 `x` 的余弦值。返回的是 -1.0 到 1.0 之间的数。

Math.E()

JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 `e`

摘要

`Math.E`

描述

`Math.E` 代表算术常量 `e`，即自然对数的底数，其值近似于 2.71828。

Math.exp()

JavaScript 1.0;JScript 1.0;ECMAScript v1

计算 e^x

摘要

`Math.exp(x)`

参数

`x` 数值或表达式，被用作指数。

返回值

e^x ，即 e 的 x 次幂。这里 e 代表自然对数的底数，其值近似为 2.71828。

Math.floor()

JavaScript 1.0;JScript 1.0;ECMAScript v1

对一个数下舍入

摘要

`Math.floor(x)`

参数

x 任意的数值或表达式。

返回值

小于等于 x 、并且与它最接近的整数。

描述

`Math.floor()` 执行的是向下取整计算，它返回的是小于等于函数参数，并且与之最接近的整数。

`Math.floor()` 将一个浮点值下舍入为最接近的整数。`Math.floor()` 执行的操作不同于 `Math.round()`，它总是进行下舍入，而不是上舍入或下舍入到最接近的整数。还要注意，`Math.floor()` 将负数舍入为更小的负数，而不是向 0 进行舍入。

例子

```
a = Math.floor(1.79);    // 结果为 1.0
b = Math.floor(1.01);    // 结果为 1.0
c = Math.floor(1.0);      // 结果为 1.0
d = Math.floor(-1.01);   // 结果为 -2.0
```

Math.LN10

JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 $\log_e 10$

摘要

`Math.LN10`

描述

`Math.LN10` 就是常量 $\log_e 10$ ，即 10 的自然对数，其值近似为 2.3025850929940459011。

Math.LN2

JavaScript 1.0; JScript 1.0; ECMAScript v1

算术常量 $\log_e 2$

摘要

`Math.LN2`

描述

`Math.LN2` 就是常量 $\log_e 2$ ，即 2 的自然对数，其值近似为 0.69314718055994528623。

Math.log()

JavaScript 1.0; JScript 1.0; ECMAScript v1

计算一个数的自然对数

摘要

`Math.log(x)`

参数

x 任何大于 0 的数值或表达式。

返回值

x 的自然对数。

描述

`Math.log()` 计算 $\log_e x$ 的值，即它的参数的自然对数。参数值必须大于 0。

可以使用下面的公式，计算一个以 10 为底的对数值和以 2 为底的对数值：

$$\log_{10} x = \log_{10} e \cdot \log_e x$$

$$\log_2 x = \log_2 e \cdot \log_e x$$

这些公式可以转化成如下的 JavaScript 函数：

```
function log10(x) { return Math.LOG10E * Math.log(x); }  
function log2(x) { return Math.LOG2E * Math.log(x); }
```

Math.LOG10E JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 $\log_{10}e$

摘要

`Math.LOG10E`

描述

`Math.LOG10E`表示常量 $\log_{10}e$,即以10为底 e 的对数。其值近似为0.4342944819 0325181667。

Math.LOG2E JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 \log_2e

摘要

`Math.LOG2E`

描述

`Math.LOG2E`表示常量 \log_2e ,即以2为底 e 的对数。其值近似为1.442695040888963387。

Math.max() JavaScript 1.0;JScript 1.0;ECMAScript v1;在 ECMAScript v3 中增强

返回最大的参数

摘要

`Math.max(args...)`

参数

`args...`

0 个或多个值。在 ECMAScript v3 之前,该方法只有两个参数。

返回值

参数中最大的值。如果没有参数,返回`-Infinity`。如果有一个参数为NaN,或是不能转换成数字的非数字值,则返回NaN。

Math.min() JavaScript 1.0;JScript 1.0;ECMAScript v1;在 ECMAScript v3 中增强

返回最小的参数**摘要**`Math.min(args...)`**参数**`args...`

0 个或多个值。在 ECMAScript v3 之前，该函数只有两个参数。

返回值

参数中最小的值。如果没有参数，返回 `Infinity`。如果有一个参数为 `NaN`，或是不能转换成数字的非数字值，则返回 `NaN`。

Math.PI JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 π **摘要**`Math.PI`**描述**

`Math.PI`表示的是常量 π , 即圆的周长和它的直径之比。这个值近似为3.14159265 358979。

Math.pow() JavaScript 1.0;JScript 1.0;ECMAScript v1

计算 x^y **摘要**`Math.pow(x, y)`**参数**

x 底数。

y 幂数。

返回值

x 的 y 次幂，即 x^y 。

描述

`Math.pow()` 计算 x 的 y 次幂。 x 和 y 可以是任意值。但如果结果是虚数或复数，`Math.pow()` 将返回 NaN。在实际应用中，这就意味着如果 x 是一个负数，那么 y 就应该是一个正整数或是一个负整数。还要记住，指数过大会引起浮点溢出，此时该方法将返回 Infinity。

Math.random()

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回一个伪随机数

摘要

`Math.random()`

返回值

0.0 到 1.0 之间的一个伪随机数。

Math.round()

JavaScript 1.0;JScript 1.0;ECMAScript v1

舍入到最接近的整数

摘要

`Math.round(x)`

参数

x 任意数。

返回值

与 x 最接近的整数。

描述

`Math.round()` 将它的参数上舍入或下舍入到与它最接近的整数。对于 .5，它将上舍入。例如，2.5 将被舍入为 3，-2.5 将被舍入为 -2。

Math.sin()JavaScript 1.0;JScript 1.0;ECMAScript v1

计算正弦值

摘要`Math.sin(x)`**参数**

x 一个以弧度表示的角。将角度乘以 0.017453293 ($2\pi/360$) 即可转换成弧度。

返回值

x 的正弦值。

Math.sqrt()JavaScript 1.0;JScript 1.0;ECMAScript v1

计算平方根

摘要`Math.sqrt(x)`**参数**

x 大于等于 0 的数。

返回值

x 的平方根。如果 *x* 小于 0，返回 NaN。

描述

`Math.sqrt()` 计算数字的平方根。注意，用 `Math.pow()` 可以计算一个数的任意次根。例如：

```
Math.cuberoot = function(x){ return Math.pow(x,1/3); }  
Math.cuberoot(8); // 返回 2
```

Math.SQRT1_2JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 $1/\sqrt{2}$ **摘要**`Math.SQRT1_2`

描述

`Math.SQRT1_2` 就是 $1/\sqrt{2}$ ，即 2 的平方根的倒数。它的值近似于 0.7071067811865476。

Math.SQRT2

JavaScript 1.0;JScript 1.0;ECMAScript v1

算术常量 $\sqrt{2}$

摘要

`Math.SQRT2`

描述

`Math.SQRT2` 就是常量 $\sqrt{2}$ ，即 2 的平方根。它的值近似于 1.414213562373095。

Math.tan()

JavaScript 1.0;JScript 1.0;ECMAScript v1

计算正切值

摘要

`Math.tan(x)`

参数

`x` 一个以弧度表示的角。将角度乘以 0.017453293 ($2\pi/360$) 即可转换成弧度。

返回值

指定的角 `x` 的正切值。

NaN

JavaScript 1.3;JScript 3.0;ECMAScript v1

非数字属性

摘要

`NaN`

描述

`NaN` 是全局属性，引用特殊的非数字值。用 `for/in` 循环不能枚举出 `NaN` 属性，用 `delete` 运算符不能删除它。注意，`NaN` 不是常量，可以用任意值设置它。

要判断一个值是否是数字，只能用 `isNaN()` 函数，因为 `NaN` 与所有值都不相等，包括它自己。

参阅

`Infinity`、`isNaN()`、`Number.NaN`

Number

JavaScript 1.1;JScript 2.0;ECMAScript v1

对数字的支持

构造函数

`new Number(value)`

`Number(value)`

参数

`value`

要创建的 `Number` 对象的数值，或是要转换成数字的值

返回值

当 `Number()` 和运算符 `new` 一起作为构造函数使用时，它返回一个新创建的 `Number` 对象。如果不用 `new` 运算符，把 `Number()` 作为一个函数来调用，它将把自己的参数转换成一个原始的数值，并且返回这个值（如果转换失败，返回 `NaN`）。

常量

`Number.MAX_VALUE`

可表示的最大的数。

`Number.MIN_VALUE`

可表示的最小的数。

`Number.NaN`

非数字值。

`Number.NEGATIVE_INFINITY`

负无穷大；溢出时返回该值。

`Number.POSITIVE_INFINITY`

正无穷大；溢出时返回该值。

方法

`toString()`

把数字转换成字符串，使用指定的基数。

`toLocaleString()`

把数字转换成字符串，使用本地数字格式规约。

`toFixed()`

把数字转换成字符串，结果的小数点后有指定位数的数字。

`toExponential()`

把数字转换成字符串，结果采用指数计数法，小数点后有指定位数的数字。

`toPrecision()`

把数字转换成字符串，结果中包含指定位数的有效数字。采用指数计数法或定点计数法，由数字的大小和指定的有效数字位数决定采用哪种方法。

描述

在 JavaScript 中，数字是一种基本的基本数据类型。在 JavaScript 1.1 中，JavaScript 还支持 `Number` 对象，该对象是原始数值的包装对象。JavaScript 在必要时会自动地进行原始数据和对象之间的转换。在 JavaScript 1.1 中，可以用构造函数 `Number()` 明确地创建一个 `Number` 对象，尽管这样做并没有什么必要。

构造函数 `Number()` 还可以不与运算符 `new` 一起使用，而直接作为转换函数来使用。以这种方式调用 `Number()` 时，它会把自己的参数转换成一个数字，然后返回转换后的原始数值（或 `NaN`）。

构造函数 `Number()` 通常还用作 5 个有用的数字常量的占位符，这 5 个有用的数字常量分别是可表示的最大的数、可表示的最小的数、正无穷大、负无穷大和特殊的非数字值。注意，这些值都是构造函数 `Number()` 自身的属性，而不是单独的 `Number` 对象的属性。例如，可以采用如下的形式使用属性 `MAX_VALUE`：

```
var biggest = Number.MAX_VALUE
```

但是却不能使用：

```
var n = new Number(2);  
var biggest = n.MAX_VALUE
```


作为比较,我们看一下 `toString()` 和 `Number` 对象的其他方法,它们是每个 `Number` 对象的方法,而不是 `Number()` 构造函数的方法。前面提到过,在必要时,JavaScript 会自动地把原始数值转换成 `Number` 对象。这就是说,调用 `Number` 方法的既可以是 `Number` 对象,也可以是原始数字值。

```
var value = 1234;  
var binary value = value.toString(2);
```

参阅

`Infinity`、`Math` 和 `NaN`

Number.MAX_VALUE	JavaScript 1.1;JScript 2.0;ECMAScript v1
-------------------------	--

最大数值

摘要

`Number.MIN_VALUE`

描述

`Number.MAX_VALUE` 是 JavaScript 中可表示的最大的数。它的值近似为 $1.79E+308$ 。

Number.MIN_VALUE	JavaScript 1.1;JScript 2.0;ECMAScript v1
-------------------------	--

最小数值

摘要

`Number.NaN`

描述

`Number.MIN_VALUE` 是 JavaScript 中可表示的最小的数(接近 0,但不是负数)。它的值近似为 $5E-324$ 。

Number.NaN	JavaScript 1.1;JScript 2.0;ECMAScript v1
-------------------	--

特殊的非数字值

摘要

`Number.NaN`

描述

`Number.NaN` 是一个特殊值，说明某些算术运算（如求负数的平方根）的结果不是数字。方法 `parseInt()` 和 `parseFloat()` 在不能解析指定的字符串时就返回这个值。对于一些常规情况下返回有效数字的函数，也可以采用这种方法，用 `Number.NaN` 说明它的错误情况。

JavaScript 以 `NaN` 的形式输出 `Number.NaN`。注意，`NaN` 与其他数值进行比较的结果总是不相等的，包括它自身在内。因此，不能通过与 `Number.NaN` 比较来检测一个值是不是数字，而只能调用函数 `isNaN()` 来比较。在 ECMAScript v1 和其后的版本中，还可以用预定义的全局常量 `NaN` 代替 `Number.NaN`。

参阅

`isNaN()`、`NaN`

Number.NEGATIVE_INFINITY JavaScript 1.1; JScript 2.0; ECMAScript v1

负无穷大

摘要

`Number.NEGATIVE_INFINITY`

描述

`Number.NEGATIVE_INFINITY` 是一个特殊值，它在算术运算或函数生成了一个比 JavaScript 能表示的最小负数还小的数（也就是比 `-Number.MAX_VALUE` 还小的数）时返回。

JavaScript 显示 `NEGATIVE_INFINITY` 时使用的是 `-Infinity`。这个值的算术行为和无穷大非常相似。例如，任何数乘无穷大结果仍为无穷大，任何数被无穷大除的结果为 0。在 ECMAScript v1 和其后的版本中，还可以用 `-Infinity` 代替 `Number.NEGATIVE_INFINITY`。

参阅

`Infinity`、`isFinite()`

Number.POSITIVE_INFINITY	JavaScript 1.1;JScript 2.0;ECMAScript v1
---------------------------------	--

正无穷大

摘要

`Number.POSITIVE_INFINITY`

描述

属性 `Number.POSITIVE_INFINITY` 是一个特殊值，它在算术运算或函数生成了一个比 JavaScript 能表示的最大的数还大的数（也就是比 `Number.MAX_VALUE` 还大的数）时返回。注意，当数字向下溢出或比 `Number.MIN_VALUE` 还小时，JavaScript 将它们转换成零。

JavaScript 显示 `POSITIVE_INFINITY` 时使用的是 `Infinity`。这个值的算术行为和无穷大非常相似。例如，任何数乘无穷大结果仍为无穷大，任何数被无穷大除结果为 0。在 ECMAScript v1 和其后的版本中，还可以用 `Infinity` 代替 `Number.POSITIVE_INFINITY`。

参阅

`Infinity` 和 `isFinite()`

Number.toExponential()	JavaScript 1.5;JScript 5.5;ECMAScript v3
-------------------------------	--

用指数计数法格式化数字

摘要

`number.toExponential(digits)`

参数

digits

小数点后的数字位数，值在 0 ~ 20 之间，包括 0 和 20，有些实现可能支持更大的数值范围。如果省略了该参数，将使用尽可能多的数字。

返回值

number 的字符串表示，采用指数计数法，即小数点之前有一位数字，小数点后有 *digits* 位数字。该数字的小数部分将被舍入，必要时用 0 补足，以便它达到指定的长度。

抛出

RangeError

digits 太小或太大时抛出的异常。0 ~ 20 之间（包括 0 和 20）的值不会引发 **RangeError**。有些实现允许支持更大范围或更小范围内的值。

TypeError

调用该方法的对象不是 **Number** 时抛出的异常。

例子

```
var n = 1.23456789;  
n.toExponential(1); // 返回 1.2e+1  
n.toExponential(5); // 返回 1.23457e+1  
n.toExponential(10); // 返回 1.2345678900e+1  
n.toExponential(); // 返回 1.23456789e+1
```

参阅

Number.toFixed()、**Number.toLocaleString()**、**Number.toPrecision()**、**Number.toString()**

Number.toFixed()

JavaScript 1.5;JScript 5.5;ECMAScript v3

采用定点计数法格式化数字

摘要

number.toFixed(digits)

参数

digits

小数点后的数字位数，是 0 ~ 20 之间的值，包括 0 和 20，有些实现可以支持更大的数值范围。如果省略了该参数，将用 0 代理。

返回值

number 的字符串表示，不采用指数计数法，小数点后有固定的 *digits* 位数字。如果必要，该数字会被舍入，也可以用 0 补足，以便它达到指定的长度。如果 *number* 大于 1e+21，该方法只调用 **Number.toString()**，返回采用指数计数法表示的字符串。

抛出

RangeError

`digits` 太小或太大时抛出的异常。0 ~ 20 之间（包括 0 和 20）的值不会引发 `RangeError`。有些实现支持更大范围或更小范围内的值。

TypeError

调用该方法的对象不是 `Number` 时抛出的异常。

例子

```
var n = 12345.6789;  
n.toFixed(0);           // 返回 12346：注意舍入，没有小数部分  
n.toFixed(1);           // 返回 12345.7：注意舍入  
n.toFixed(5);           // 返回 12345.678900：注意补零  
(1.23e+20).toFixed(2); // 返回 12300000000000000000.00  
(1.23e-10).toFixed(2) // 返回 0.00
```

参阅

`Number.toExponential()`、`Number.toLocaleString()`、`Number.toPrecision()`、`Number.toString()`

Number.toLocaleString() JavaScript 1.5; JScript 5.5; ECMAScript v3

把数字转换成本地格式的字符串

摘要

`number.toLocaleString()`

返回值

数字的字符串表示，由实现决定，根据本地规范进行格式化，可能影响到小数点或千分位分隔符采用的标点符号。

抛出

TypeError

调用该方法的对象不是 `Number` 时抛出的异常。

参阅

`Number.toExponential()`、`Number.toFixed()`、`Number.toPrecision()`、`Number.toString()`

Number.toPrecision()

JavaScript 1.5;JScript 5.5;ECMAScript v3

格式化数字的有效位

摘要

`number.toPrecision(precision)`

参数

`precision`

返回的字符串中的有效位数，是 1 ~ 21 之间（包括 1 和 21）的值。有些实现允许有选择地支持更大或更小的 `precision`。如果省略了该参数，将调用方法 `toString()`，而不是把数字转换成十进制的值。

返回值

`number` 的字符串表示，包含 `precision` 个有效数字。如果 `precision` 足够大，能够包括 `number` 整数部分的所有数字，那么返回的字符串将采用定点计数法。否则，采用指数计数法，即小数点前有一位数字，小数点后有 `precision - 1` 位数字。必要时，该数字会被舍入或用 0 补足。

抛出

`RangeError`

`digits` 太小或太大时抛出的异常。1 - 21 之间（包括 1 和 21）的值不会引发 `RangeError`。有些实现支持更大范围或更小范围内的值。

`TypeError`

调用该方法的对象不是 `Number` 时抛出的异常。

例子

```
var n = 12345.6789;  
n.toPrecision(1); // 返回 1e+4  
n.toPrecision(3); // 返回 1.23e+4  
n.toPrecision(5); // 返回 12346: 注意舍入  
n.toPrecision(10); // 返回 12345.67890: 注意补零
```

参阅

`Number.toExponential()`、`Number.toFixed()`、`Number.toLocaleString()`、`Number.toString()`

Number.toString()JavaScript 1.1;JScript 2.0;ECMAScript v1

将一个数字转换成字符串

覆盖 Object.toString()

摘要`number.toString(radix)`**参数**`radix`

可选的参数，指定表示数字的基数，是 2 - 36 之间的整数。如果省略了该参数，使用基数 10。但要注意，如果该参数是 10 以外的其他值，则 ECMAScript 标准允许实现返回任意值。

返回值

数字的字符串表示。

抛出`TypeError`

调用该方法的对象不是 `Number` 时抛出的异常。

描述

`Number` 对象的方法 `toString()` 可以将数字换成字符串。当省略了 `radix` 参数或指定它的值为 10 时，该数字将被转换成基数为 10 的字符串。如果 `radix` 是其他值，该方法将返回由实现定义的字符串。Netscape 实现和 JScript 3.0 后的 Microsoft 实现都支持 `radix` 参数，并返回以指定基数表示的字符串。

参阅

`Number.toExponential()`、`Number.toFixed()`、`Number.toLocaleString()`、`Number.toPrecision()`

Number.valueOf()JavaScript 1.1;JScript 2.0;ECMAScript v1

返回原始数值

摘要`number.valueOf()`

返回值

Number 对象的原始数值。几乎没有必要明确调用该方法。

抛出

TypeError

调用该方法的对象不是 Number 时抛出的异常。

参阅

Object.valueOf()

Object

JavaScript 1.0;JScript 1.0;ECMAScript v1

含有所有 JavaScript 对象的特性的超类

构造函数

new Object()

new Object(value)

参数

value

可选的参数，声明了要转换成 Number 对象、Boolean 对象或 String 对象的原始值（即数字、布尔值或字符串）。JavaScript 1.1 之前的版本和 ECMAScript v1 不支持该对象。

返回值

如果没有给构造函数传递 value 参数，那么它将返回一个新创建的 Object 实例。如果指定了原始的 value 参数，构造函数将创建并返回原始值的包装对象，即 Number 对象、Boolean 对象或 String 对象。当不使用 new 运算符，将 Object() 构造函数作为函数调用时，它的行为与使用 new 运算符时一样。

属性

constructor

对一个 JavaScript 函数的引用，该函数是对象的构造函数。

方法

hasOwnProperty()

检查对象是否有局部定义的（非继承的）、具有特定名字的属性。

`isPrototypeOf()`

检查对象是不是指定对象的原型。

`propertyIsEnumerable()`

检查指定的属性是否存在，以及是否能用 `for/in` 循环枚举。

`toLocaleString()`

返回对象地方化的字符串表示。该方法的默认实现只调用 `toString()` 方法，但子类可以覆盖它，提供本地化。

`toString()`

返回对象的字符串表示。`Object` 类提供的该方法的实现相当普通，并且没有提供更多有用的信息。`Object` 的子类通过定义自己的 `toString()` 方法覆盖了这一方法（`toString()` 方法能够生成更有用的结果）。

`valueOf()`

返回对象的原始值（如果存在）。对于类型为 `Object` 的对象，该方法只返回对象自身。`Object` 的子类（如 `Number` 和 `Boolean`）覆盖了该方法，返回的是与对象相关的原始数值。

描述

`Object` 类是 JavaScript 语言的内部数据类型。它是其他 JavaScript 对象的超类，因此其他对象都继承了 `Object` 类的方法和行为。第八章对 JavaScript 中的对象的基本行为进行了解释。

除了用上面所示的 `Object()` 构造函数，还可以用第八章介绍的 `Object` 直接量语法创建并初始化对象。

参阅

`Array`、`Boolean`、`Function`、`Function.prototype`、`Number`、`String`；第八章

`Object.constructor`

JavaScript 1.1; JScript 2.0; ECMAScript v1

对象的构造函数

摘要

`object.constructor`

描述

对象的 `constructor` 属性引用了该对象的构造函数。例如，如果用 `Array()` 构造函数创建一个数组，那么 `a.constructor` 引用的就是 `Array`。

```
a = new Array(1,2,3); // 创建一个对象
a.constructor == Array // 计算结果为 true
```

`constructor` 属性常用于判断未知对象的类型。给定了一个未知的值，就可以使用 `typeof` 运算符来判断它是原始的值还是对象。如果它是对象，就可以使用 `constructor` 属性来判断对象的类型。例如，下面的函数用来判断一个给定的值是否是数组：

```
function isArray(x) {
    return (typeof x == "object") && (x.constructor == Array);
}
```

但是要注意，虽然这种方法适用于JavaScript核心语言的内部对象，但对于“主对象”，如 `Window` 这样的客户端JavaScript对象，这种方法就不一定适用了。`Object.toString()` 方法的默认实现提供了另一种判断未知对象类型的方法。

参阅

`Object.toString()`

Object.hasOwnProperty() JavaScript 1.5;JScript 5.5;ECMAScript v3

检查属性是否被继承

摘要

`object.hasOwnProperty(propname)`

参数

propname

一个字符串，包含 *object* 的属性名。

返回值

如果 *object* 有 *propname* 指定的非继承属性，则返回 `true`。如果 *object* 没有名为 *propname* 指定的属性，或者它从原型对象继承了这一属性，则返回 `false`。

描述

在第八章解释过, JavaScript对象既可以有自己的属性, 又可以从原型对象继承属性。`hasOwnProperty()` 方法提供了区分继承属性和非继承的局部属性的方法。

例子

```
var o = new Object();           // 创建对象
o.x = 3, 4;                     // 定义非继承的局部属性
o.hasOwnProperty("x");          // 返回 true: x 是 o 的局部属性
o.hasOwnProperty("y");          // 返回 false: o 没有属性 y
o.hasOwnProperty("toString");   // 返回 false: toString 属性是继承的
```

参阅

`Function.prototype`、`Object.prototype.isEnumerable()`; 第八章

`Object.isPrototypeOf()`

JavaScript 1.5; JScript 5.5; ECMAScript v3

一个对象是否是另一个对象的原型

摘要

`object.isPrototypeOf(o)`

参数

`o` 任意对象。

返回值

如果 `object` 是 `o` 的原型, 则返回 `true`。如果 `o` 不是对象, 或者 `object` 不是 `o` 的原型, 则返回 `false`。

描述

在第八章解释过, JavaScript 对象继承了原型对象的属性。一个对象的原型是通过用于创建并初始化该对象的构造函数的 `prototype` 属性引用的。`isPrototypeOf()` 方法提供了判断一个对象是否是另一个对象原型的方法。该方法可以用于确定对象的类。

例子

```
var o = new Object();           // 创建一个对象
Object.prototype.isPrototypeOf(o) // true: o 是对象
Function.prototype.isPrototypeOf(o.toString); // true: toString 是函数
Array.prototype.isPrototypeOf([1,2,3]);       // true: [1,2,3] 是数组
```

· 下面是执行同样测试的另一种方法

```
o.constructor == Object; // true: o 由 Object() 构造函数创建
o.toString.constructor == Function; // true: o.toString 是函数
// 原型对象本身有原型对象。下面的调用返回 true,
// 说明函数继承了 Function.prototype 和 Object.prototype 属性
Object.prototype.isPrototypeOf(Function.prototype);
```

参阅

Function.prototype、Object.constructor; 第八章

Object.prototype.isEnumerable() JavaScript 1.5;JScript 5.5;ECMAScript v3

是否可以通过 for/in 循环看到属性

摘要

object.propertyIsEnumerable(*propname*)

参数

propname

一个字符串, 包含 *object* 原型的名字。

返回值

如果 *object* 具有名为 *propname* 的非继承属性, 而且该属性是可枚举的 (即用 for/in 循环可以枚举出它), 则返回 true。

描述

用 for/in 语句可以遍历一个对象“可枚举”的属性。但并非一个对象的所有属性都是可枚举的, 通过 JavaScript 代码添加到对象的属性是可枚举的, 而内部对象的预定义属性 (如方法) 通常是不可枚举的。propertyIsEnumerable() 方法提供了区分可枚举属性和不可枚举属性的方法。但要注意, ECMAScript 标准规定, propertyIsEnumerable() 方法不检测原型链, 这意味着它只适用于对象的局部属性, 不能检测继承属性的可枚举性。

例子

```
var o = new Object(); // 创建一个对象
o.x = 3.14;           // 定义一个属性
o.propertyIsEnumerable("x"); // true 属性 x 是局部的、可枚举的
o.propertyIsEnumerable("y"); // false: o 没有属性 y
o.propertyIsEnumerable("toString"); // false: toString 属性是继承的
Object.prototype.propertyIsEnumerable("toString"); // false 不可枚举的
```

Bug

当标准限制 `propertyIsEnumerable()` 方法只能检测非继承属性时，明显是错的。Internet Explorer 5.5 按标准实现了该方法。Netscape 6.0 实现的 `propertyIsEnumerable()` 方法考虑了原型链。虽然这种方法可取，但它与标准冲突，所以 Netscape 6.1 修改了它，以便与 IE 5.5 匹配。由于标准中有这个错误，因此该方法不是那么有用。

参阅

`Function.prototype`、`Object.hasOwnProperty()`、第八章

`Object.toLocaleString()` JavaScript 1.5; JScript 5.5; ECMAScript v3

返回对象的本地字符串表示

摘要

`object.toString()`

返回值

表示对象的字符串。

描述

该方法用于返回对象的字符串表示，本地化为适合本地的形式。`Object` 类提供的默认的 `toLocaleString()` 方法只调用 `toString()` 方法，返回非本地化的字符串。但其他类（包括 `Array`、`Date` 和 `Number`）定义了自己的 `toLocaleString()` 版本，指定本地化字符串的转换。在定义自己的类时，也可以覆盖该方法。

参阅

`Array.toLocaleString()`、`Date.toLocaleString()`、`Number.toLocaleString()`、`Object.toString()`

`Object.toString()` JavaScript 1.5; JScript 5.5; ECMAScript v3

定义一个对象的字符串表示

摘要

`object.toString()`

返回值

表示对象的字符串。

描述

这里的方法 `toString()` 并不是你在 JavaScript 程序中经常显示调用的那个 `toString()` 方法。它是你在对象中定义的一个方法，当系统需要把对象转换成字符串时就会调用它。

当在字符串环境中使用对象时，JavaScript 系统就会调用 `toString()` 方法把那个对象转换成字符串。例如，假定一个函数期望得到的参数是字符串，那么把对象传递给它时，系统就会将这个对象转换成字符串：

```
alert(my_object);
```

同样，在用运算符 “+” 连接字符串时，对象也会被转换成字符串：

```
var msg = "My object is: " + my_object;
```

调用方法 `toString()` 时不给它传递任何参数，它返回的应该是一个字符串。要使这个字符串有用，它的值就必须以调用 `toString()` 方法的对象的值为基础。

当用 JavaScript 自定义一个类时，为这个类定义一个 `toString()` 方法很有用。如果没有给它定义 `toString()` 方法，那么这个对象将继承 `Object` 类的默认 `toString()` 方法。这个方法返回的字符串形式如下：

```
[object class]
```

这里，`class` 是一个对象类，其值可以是 “Object”、“String”、“Number”、“Function”、“Window”、“Document”，等等。这种行为有时对确定未知对象的类型或类有用。但由于大多数对象都有自定义的 `toString()` 版本，所以必须明确地对对象 `o` 调用 `Object.toString()`，代码如下所示：

```
Object.prototype.toString.apply(o);
```

注意，这种识别未知对象的方法只适用于内部对象。如果你定义了自己的对象类，那么它的类是 “Object”。在这种情况下，可以用 `Object.constructor` 属性获取更多有关对象的信息。

在调试 JavaScript 程序时，`toString()` 方法非常有用，使用它可以输出对象，查看它们的值。因此，为你创建的每个对象类都定义 `toString()` 方法很有用。

虽然 `toString()` 方法通常由系统自动调用，但你也可以自己调用它。例如，在 JavaScript 不能自动把对象转换成字符串的环境中，可以明确地调用 `toString()` 方法来实现这一点：

```
y = Math.sqrt(x);           // 计算一个数
yStr = y.toString();         // 转换为一个字符串
```

注意，在这个例子中，数字具有内部的 `toString()` 方法，可以用该方法进行强制性的转换。

在其他的环境中，即使 JavaScript 可以自动地进行转换，你也可以调用 `toString()` 方法，因为对 `toString()` 的明确调用可以使代码更加清晰：

```
alert(my_obj.toString());
```

参阅

`Object.constructor`、`Object.toLocaleString()`、`Object.valueOf()`

`Object.valueOf()`

JavaScript 1.1; JScript 2.0; ECMAScript v1

指定对象的原始值

摘要

`object.valueOf()`

返回值

与对象 `object` 相关的原始值（如果存在）。如果没有值与 `object` 相关，则返回对象自身。

描述

对象的 `valueOf()` 方法返回的是与那个对象相关的原始值（如果这样的值存在）。对于类型为 `Object` 的对象来说，由于它们没有原始值，因此该方法返回的是这些对象自身。

对于类型为 `Number` 的对象，`valueOf()` 返回该对象表示的原始数值。同样，对于 `Boolean` 对象来说，该方法返回与对象相关的布尔值。对于 `String` 对象来说，返回与对象相关的字符串。

其实，几乎没有必要自己调用 `valueOf()` 方法。在期望使用原始值的地方，JavaScript

会自动地执行转换。事实上，由于方法 `valueOf()` 是被自动调用的，因此要分辨究竟是原始值还是与之相应的对象非常困难。虽然使用 `typeof` 运算符可以显示字符串和 `String` 对象之间的区别，但在实际应用中，它们在 JavaScript 代码中的作用是一样的。

`Number` 对象、`Boolean` 对象和 `String` 对象的 `valueOf()` 方法可以将这些包装对象转换成它们表示的原始值。在调用构造函数 `Object()` 时，如果把数字、布尔值或字符串作为参数传递给它，它将执行相反的操作，即将原始值打包成相应的对象。几乎在所有的环境中，JavaScript 都可以自动地实现原始值和对象之间的转换，所以一般说来没有必要用这种方法调用构造函数 `Object()`。

在某些环境中，你可以为自己的对象定制一个 `valueOf()` 方法。例如，你可以定义一个 JavaScript 对象来表示复数（即一个实数加一个虚数）。作为这个对象的一部分，要给它定义执行复数的加法、乘法等其他运算的方法。不过，还有一种功能是你想要的，即像处理常规实数一样处理复数，舍弃它的虚数部分。可以使用下面的代码实现这一点：

```
Complex.prototype.valueOf = new Function('return this.real');
```

有了这个为 `Complex` 对象定义的 `valueOf()` 方法，就可以把复数对象传递给方法 `Math.sqrt()`，它将计算复数的实数部分的平方根。

参阅

`Object.toString()`

`parseFloat()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

把字符串转换成数字

摘要

`parseFloat(s)`

参数

s 要被解析并转换成数字的字符串。

返回值

解析后的数字，如果字符串 *s* 没有以一个有效的数字开头，则返回 `NaN`。在 JavaScript 1.0 中，当 *s* 不能被解析成数字时，`parseFloat()` 返回的是 0 而不是 `NaN`。

描述

方法 `parseFloat()` 将对字符串 *s* 进行解析, 返回出现在 *s* 中的第一个数字。当 `parseFloat()` 在 *s* 中遇到了一个不是有效数字的字符时, 解析过程就停止了, 解析的结果也将在此时返回。如果 *s* 的开头是一个 `parseFloat()` 不能解析的数字, 该函数将返回 NaN。可以用函数 `isNaN()` 来检测这个值。如果只想解析数字的整数部分, 则使用 `parseInt()` 方法而不是 `parseFloat()` 方法。

Bug

NaN 不受 JavaScript 1.0 的支持, 所以在该版本中, 当它不能解析 *s* 时, `parseFloat()` 返回零。这意味着在 JavaScript 1.0 中, 如果 `parseFloat()` 的返回值为 0, 必须对 *s* 执行额外的测试, 以决定它表示为零还是根本不表示任何数。

参阅

`isNaN()`、`parseInt()`

`parseInt()`

JavaScript 1.0; JScript 1.1; ECMAScript v1

把字符串转换成整数

摘要

`parseInt(s)`

`parseInt(s, radix)`

参数

s 要被解析的字符串。

radix

可选的整数参数, 表示要解析的数字的基数。如果省略了该参数或者它的值为 0, 数字将以 10 为基数来解析。如果它以 “0x” 或 “0X” 开头, 则以 16 为基数。如果该参数小于 2 或大于 36, 则 `parseInt()` 返回 NaN。

返回值

解析后的数字, 如果字符串 *s* 不是以一个有效的整数开头, 则返回 NaN。在 JavaScript 1.0 中, 当 `parseInt()` 不能解析 *s* 时, 它返回的是 0 而不是 NaN。

描述

方法 `parseInt()` 将对字符串 *s* 进行解析, 并且返回出现在 *s* 中的第一个数字 (可以具有减号)。当 `parseInt()` 在 *s* 中遇到的字符不是指定的基数 *radix* 可以使用的有效数字时, 解析过程就停止, 解析的结果也将在此时返回。如果 *s* 的开头是 `parseInt()` 不能解析的数字, 该函数将返回 `NaN`。可以用函数 `isNaN()` 来检测这个值。

参数 *radix* 指定的是要解析成的数字的基数。如果将它设置为 10, `parseInt()` 就会将字符串解析成十进制的数。将它设置为 8, 那么解析的结果就是八进制 (使用 0 ~ 7 八个数字) 的数。将它设置为 16, 解析的结果就是十六进制 (使用数字 0 ~ 9 和字母 A ~ F 表示) 的值。*radix* 的值可以是 2 ~ 36 之间的任意一个整数。

如果 *radix* 的值为 0, 或者没有设置 *radix* 的值, 那么 `parseInt()` 将根据字符串 *s* 来判断数字的基数。如果 *s* (在可选的减号后) 以 0x 开头, 那么 `parseInt()` 将把 *s* 的其余部分解析成十六进制的整数。如果 *s* 以 0 开头, 那么 ECMAScript v3 标准允许 `parseInt()` 的一个实现把其后的字符解析成八进制的数字或十进制的数字。如果 *s* 以 1 ~ 9 之间的数字开头, `parseInt()` 将把它解析成十进制的整数。

例子

```
parseInt('19', 10);    // 返回 9   (10 + 9)
parseInt('11', 2);     // 返回 3   (2 + 1)
parseInt("17", 8);     // 返回 15  (8 + 7)
parseInt("1f", 16);    // 返回 31  (16 + 15)
parseInt('10');        // 返回 10
parseInt("0x10");      // 返回 16
parseInt("0101");      // 不明确: 返回 10 或 5
```

Bug

在没有指定 *radix* 时, ECMAScript v3 允许实现将以 “0” (但不是 “0x” 或 “0X”) 开头的字符串解析为八进制或十进制的数。要避免这种二义性, 应该明确指定基数, 或只有确定所有要解析的数字都是以 “0x” 或 “0X” 开头的十进制数或十六进制数时, 才可以不指定基数。

JavaScript 1.0 不支持 `NaN`, 所以在这个版本中, 当 `parseInt()` 不能解析 *s* 时, 返回 0 而不是 `NaN`。在 JavaScript 1.0 中, `parseInt()` 不能区分错误的输入与合法的输入 “0”。

参阅

isNaN()、parseFloat()

RangeError

JavaScript 1.5;JScript 5.5;ECMAScript v3

在数字超出合法范围时抛出

继承 Error

构造函数

`new RangeError()`

`new RangeError(message)`

参数

message

可选的错误消息，提供异常的详细情况。如果指定了该参数，它将作为 RangeError 对象的 message 属性的值。

返回值

新构造的 RangeError 对象。如果指定了参数 *message*，该 Error 对象把它作为 message 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果不用 new 运算符，把 RangeError() 构造函数当作函数调用，那么它的行为与使用 new 运算符调用时一样。

属性

message

提供异常细节的错误消息。该属性存放传递给构造函数的字符串，或实现定义的默认字符串。详见 “Error.message”。

name

声明异常类型的字符串。所有 RangeError 对象的 name 属性都继承值 “RangeError”。

描述

当数字超出合法范围时，RangeError 类的一个实例就会被抛出。例如，把数组的 length 属性设置成一个负数，就会使 RangeError 对象被抛出。关于抛出和捕捉异常的细节，请参阅 “Error”。

参阅

Error、Error.message、Error.name

ReferenceError

JavaScript 1.5;JScript 5.5;ECMAScript v3

在读取不存在的变量时抛出

继承 Error

构造函数

```
new ReferenceError()
```

```
new ReferenceError(message)
```

参数

message

可选的错误消息，提供异常的详细情况。如果指定了该参数，它将作为 ReferenceError 对象的 message 属性的值。

返回值

新构造的 ReferenceError 对象。如果指定了参数 *message*，该 Error 对象把它作为 message 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果不用 new 运算符，把 ReferenceError() 构造函数当作函数调用，它的行为与使用 new 运算符调用时一样。

属性

message

提供异常细节的错误消息。该属性存放传递给构造函数的字符串，或实现定义的默认字符串。详见“Error.message”。

name

声明异常类型的字符串。所有 ReferenceError 对象的 name 属性都继承值“ReferenceError”。

描述

在读取一个不存在的变量的值时，ReferenceError 类的一个实例就会抛出。关于抛出和捕捉异常的细节，请参阅“Error”。

参阅

Error、Error.message、Error.name

RegExp

JavaScript 1.2;JScript 3.0;ECMAScript v3

用于模式匹配的正则表达式

直接量语法

`/pattern/attributes`

构造函数

`new RegExp(pattern, attributes)`

参数

pattern

一个字符串，指定了正则表达式的模式或其他正则表达式。

attributes

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。

返回值

一个新的 RegExp 对象，具有指定的模式和标志。如果参数 *pattern* 是正则表达式而不是字符串，那么 `RegExp()` 构造函数将用与指定的 `RegExp` 相同的模式和标志创建一个新的 `RegExp` 对象。如果不用 `new` 运算符，将 `RegExp()` 作为函数调用，那么它的行为与用 `new` 运算符调用时一样，只是当 *pattern* 是正则表达式时，它只返回 *pattern*，而不再创建一个新的 `RegExp` 对象。

抛出

SyntaxError

如果 *pattern* 不是合法的正则表达式，或 *attributes* 含有“g”、“i”、“m”之外的字符，抛出该异常。

TypeError

如果 *pattern* 是 `RegExp` 对象，但没有省略 *attributes* 参数，抛出该异常。

实例属性

`global`

RegExp 对象是否具有性质 `g`。

`ignoreCase`

RegExp 对象是否具有性质 `i`。

`lastIndex`

上次匹配后的字符位置，用于在一个字符串中进行多次匹配。

`multiline`

RegExp 对象是否具有 `m` 性质。

`source`

正则表达式的源文本。

方法

`exec()`

执行强大的、通用的模式匹配。

`test()`

检测一个字符串是否含有某个模式。

描述

RegExp 对象表示一个正则表达式，它是对字符串执行模式匹配的强大工具。要了解正则表达式的语法和用法，请参阅第十章。

参阅

第十章

RegExp.exec()

JavaScript 1.2;JScript 3.0;ECMAScript v3

通用的匹配模式

摘要

`regexp.exec(string)`

参数

string

要检索的字符串。

返回值

一个数组，存放的是匹配的结果。如果没有找到匹配，值为 `null`。返回的数组的格式在下面介绍。

抛出

`TypeError`

调用该方法的对象不是 `RegExp` 时，抛出该异常。

描述

在所有的 `RegExp` 模式匹配方法和 `String` 模式匹配方法中，`exec()` 的功能最强大。它是一个通用的方法，使用起来比 `RegExp.test()`、`String.search()`、`String.replace()` 和 `String.match()` 都复杂。

`exec()` 将检索字符串 *string*，从中得到与正则表达式 *regexp* 相匹配的文本。如果 `exec()` 找到了匹配的文本，它就会返回一个结果数组。否则，返回 `null`。这个返回数组的第 0 个元素就是与表达式相匹配的文本。第 1 个元素是与 *regexp* 的第一个子表达式相匹配的文本（如果存在）。第 2 个元素是与 *regexp* 的第二个子表达式相匹配的文本，以此类推。通常，数组的 `length` 属性声明的是数组中的元素个数。除了数组元素和 `length` 属性之外，`exec()` 还返回两个属性。`index` 属性声明的是匹配文本的第一个字符的位置。`input` 属性指的就是 *string*。在调用非全局 `RegExp` 对象的 `exec()` 方法时，返回的数组与调用方法 `String.match()` 返回的方法相同。

在调用非全局模式的 `exec()` 方法时，它将进行检索，并返回上述结果。不过，当 *regexp* 是一个全局正则表达式时，`exec()` 的行为就稍微复杂一些。它在 *regexp* 的属性 `lastIndex` 指定的字符处开始检索字符串 *string*。当它找到了与表达式相匹配的文本时，在匹配之后，它将把 *regexp* 的 `lastIndex` 属性设置为匹配文本的第一个字符的位置。这就是说，可以通过反复地调用 `exec()` 方法来遍历字符串中的所有匹配文本。当 `exec()` 再也找不到匹配的文本时，它将返回 `null`，并且把属性 `lastIndex` 重置为 0。如果在另一个字符串中完成了一次模式匹配之后要开始检索新的字符串，就必须手动地把 `lastIndex` 属性重置为 0。

注意, 无论 *regexp* 是否是全局模式, `exec()` 都会将完整的细节添加到它返回的数组中。这就是 `exec()` 和 `String.match()` 的不同之处, 后者在全局模式下返回的信息要少得多。事实上, 在循环中反复地调用 `exec()` 方法是惟一一种获得全局模式的完整模式匹配信息的方法。

例子

可以在循环中使用 `exec()` 来检索一个字符串中的所有匹配文本。例如:

```
var pattern = /(bJava|w*\b)/g;
var text = "JavaScript is more fun than Java or JavaBeans!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched '" + result[0] +
        "' at position " + result.index +
        " next search begins at position " + pattern.lastIndex);
}
```

Bug

在 JScript 3.0 中, `exec()` 不能正确地使用 and 设置属性 `lastIndex`. 因此这时不能将全局模式的 `exec()` 用于循环中, 就像上面那个例子所示的一样。

参阅

`RegExp.lastIndex`, `RegExp.test()`, `String.match()`, `String.replace()`, `String.search()`;
第十章

RegExp.global

JavaScript 1.2; JScript 5.5; ECMAScript v3

正则表达式是否全局匹配

摘要

`regexp.global`

描述

`RegExp` 对象的属性 `global` 是一个只读的布尔值。它声明了给定的正则表达式是否执行全局匹配, 如创建它时是否使用了性质 `g`。

RegExp.ignoreCase

JavaScript 1.2;JScript 5.5;ECMAScript v3

正则表达式是否区分大小写

摘要

`regexp.ignoreCase`

描述

RegExp 对象的属性 `ignoreCase` 是一个只读的布尔值。它声明了一个给定的正则表达式是否执行区分大小写的匹配，例如创建它时是否使用了性质 `i`。

RegExp.lastIndex

JavaScript 1.2;JScript 5.5;ECMAScript v3

下次匹配的起始位置

摘要

`regexp.lastIndex`

描述

RegExp 对象的属性 `lastIndex` 是一个可读可写的值。对于设置了 `g` 性质的正则表达式来说，该属性存放的是一个整数，它声明了紧接着上次找到的匹配文本的字符的位置。上次匹配的结果是由方法 `RegExp.exec()` 或 `RegExp.test()` 找到的，它们都以 `lastIndex` 属性所指的位置作为下次检索的起始点。这样，就可以通过反复调用这两个方法来遍历一个字符串中的所有匹配文本。注意，不具有性质 `g` 和不表示全局模式的 RegExp 对象不能使用 `lastIndex` 属性。

由于这一属性是可读可写的，所以只要目标字符串的下一搜索开始，就可以对它进行设置。当方法 `exec()` 或 `test()` 再也找不到可以匹配的文本时，它们会自动地把 `lastIndex` 属性重置为 0。如果在成功的匹配了某个字符串之后就开始检索另一个字符串，需要明确地把这个属性设为 0。

参阅

`RegExp.exec()`、`RegExp.test()`

RegExp.source

JavaScript 1.2;JScript 3.0;ECMAScript v3

正则表达式的文本

摘要

`regexp.source`

描述

RegExp 对象的属性 `source` 是一个只读的字符串。它存放的是 RegExp 模式的文本。该文本中不包括正则表达式直接量使用的定界符，也不包括性质 `g`、`i`、`m`。

RegExp.test()

JavaScript 1.2;JScript 3.0;ECMAScript v3

检测一个字符串是否匹配某个模式

摘要

`regexp.test(string)`

参数

string

要检测的字符串。

返回值

如果字符串 *string* 中含有与 *regexp* 匹配的文本，就返回 `true`，否则返回 `false`。

抛出

`TypeError`

调用该方法的对象不是 `RegExp` 时，抛出该异常。

描述

方法 `test()` 将检测字符串 *string*，看它是否含有与 *regexp* 相匹配的文本。如果 *string* 中含有这样的文本，该方法将返回 `true`，否则，返回 `false`。调用 `RegExp` 对象 *r* 的 `test()` 方法，给它传递字符串 *s*，等价于下面的表达式：

```
(r.exec(s) != null)
```

例子

```
var pattern = /java/i;  
pattern.test('JavaScript'); // 返回 true
```

```
pattern.test("ECMAScript"); // 返回 false
```

参阅

`RegExp.exec()`、`RegExp.lastIndex`、`String.match()`、`String.replace()`、`String.substring()`；第十章

`RegExp.toString()`

JavaScript 1.2;JScript 3.0;ECMAScript v3

把正则表达式转换成字符串

覆盖 `Object.toString()`

摘要

```
regexp.toString()
```

返回值

`regexp` 的字符串表示。

抛出

`TypeError`

调用该方法的对象不是 `RegExp` 时，抛出该异常。

描述

`RegExp.toString()` 方法将以正则表达式直接量的形式返回正则表达式的字符串表示。

注意，不允许用实现添加转义序列，这样可以确保返回的字符串是合法的正则表达式直接量。考虑由表达式 `new RegExp("/", "g")` 创建的正则表达式。`RegExp.toString()` 的一种实现对该正则表达式返回 `“///g”`，此外它还能添加转义序列，返回 `“/\\/g”`。

`String`

JavaScript 1.0;JScript 1.0;ECMAScript v1

对字符串的支持

从 `Object` 继承

构造函数

```
new String(s) // 构造函数
```

```
String(s) // 构造函数
```

参数

s 要存储在 String 对象中或转换成原始字符串的值。

返回值

当 String() 与 new 运算符一起作为构造函数使用时, 返回一个新创建的 String 对象, 存放的是字符串 *s* 或 *s* 的字符串表示。当不用 new 运算符调用 String() 时, 它只把 *s* 转换成原始的字符串, 并返回转换后的值。

属性

length 字符串中的字符数。

方法

charAt() 抽取字符串中指定位置处的字符。

charCodeAt() 返回字符串中指定位置处的字符编码。

concat() 把一个或多个值连接到字符串上。

indexOf() 在字符串中检索一个字符或一个子串。

lastIndexOf() 在字符串中向后检索一个字符或一个子串。

match() 用正则表达式执行模式匹配。

replace() 用正则表达式执行查找、替换操作。

search() 检索字符串中与正则表达式匹配的子串。

slice() 返回字符串的一个片段或一个子串。

split() 把字符串分割成一个字符串数组, 在指定的分界字符处或正则表达式处执行分割。

substring() 从字符串中抽取一个子串。

substr() 从字符串中抽取一个子串。该方法是 substring() 的一个变体。

toLowerCase() 将字符串中的所有字符都转换成小写的, 然后返回一个副本。

toString() 返回原始的字符串值。

toUpperCase() 将字符串中的所有字符都转换成大写的, 然后返回一个副本。

Value Of() 返回原始字符串值。

静态方法

`String.fromCharCode()`

用作为参数而传递的字符代码创建一个新的字符串。

HTML 方法

从 JavaScript 1.0 和 JScript 1.0 起, `String` 类定义了许多方法, 返回的字符串是把它放在 HTML 标记中修改后得到的。虽然 ECMAScript 没有标准化这些方法, 但它们在客户端和服务端动态生成 HTML 的脚本代码中非常有用。用这些非标准的方法, 可以为黑体的红色超链接创建常见 HTML 源代码, 如下所示:

```
var s = 'click here!';  
var html = s.bold().link('javascript:alert( "hello" )').fontcolor("red");
```

因为这些方法没有被标准化, 所以它们没有单独的参考页:

<code>anchor(name)</code>	在 <code></code> 环境中返回字符串的一个副本。
<code>big()</code>	在 <code><big></code> 环境中返回字符串的一个副本。
<code>blink()</code>	在 <code><blink></code> 环境中返回字符串的一个副本。
<code>bold()</code>	在 <code></code> 环境中返回字符串的一个副本。
<code>fixed()</code>	在 <code><tt></code> 环境中返回字符串的一个副本。
<code>fontcolor(color)</code>	在 <code></code> 环境中返回字符串的一个副本。
<code>fontsize(size)</code>	在 <code></code> 环境中返回字符串的一个副本。
<code>italics()</code>	在 <code><i></code> 环境中返回字符串的一个副本。
<code>link(url)</code>	在 <code></code> 环境中返回字符串的一个副本。
<code>small()</code>	在 <code><small></code> 环境中返回字符串的一个副本。
<code>strike()</code>	在 <code><strike></code> 环境中返回字符串的一个副本。
<code>sub()</code>	在 <code><sub></code> 环境中返回字符串的一个副本。
<code>sup()</code>	在 <code><sup></code> 环境中返回字符串的一个副本。

描述

字符串是 JavaScript 的一种基本数据类型。`String` 类提供了操作原始字符串值的方法。`String` 对象的 `length` 属性声明了该字符串中的字符数。类 `String` 定义了大量操作字

字符串的方法，例如从字符串中提取字符或子串，或者检索字符或子串。注意，JavaScript的字符串是不可变（immutable）的，String类定义的方法都不能改变字符串的内容。像String.toUpperCase()这样的方法，返回的是全新的字符串，而不是修改原始字符串。

在JavaScript 1.2及其后版本的Netscape实现中，字符串的行为就像只读的字符数组。例如，从字符串s中提取第三个字符，可以用s[2]代替更加标准的s.charAt(2)。此外，对字符串应用for/in循环时，它将枚举字符串中每个字符的数组下标（但要注意，ECMAScript标准规定，不能枚举length属性）。因为Netscape实现中的字符串的数组行为不标准，所以应该避免使用它。

参阅

第三章

String.charAt()

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回字符串中的第n个字符

摘要

string.charAt(*n*)

参数

n 应该返回的字符在*string*中的下标。

返回值

字符串*string*的第*n*个字符。

描述

方法String.charAt()返回字符串*string*中的第*n*个字符。字符串中第一个字符的下标值是0。如果参数*n*不在0和*string.length*-1之间，该方法将返回一个空字符串。注意，JavaScript并没有一种有异于字符串类型的字符数据类型，所以返回的字符是长度为1的字符串。

参阅

String.charCodeAt()、String.indexOf()和String.lastIndexOf()

String.charCodeAt()JavaScript 1.2;JScript 5.5;ECMAScript v1

返回字符串中的第 *n* 个字符的代码

摘要

string.charCodeAt(*n*)

参数

n 返回编码的字符的下标。

返回值

string 中的第 *n* 个字符的 Unicode 编码。这个返回值是 0 ~ 65535 之间的 16 位整数。

描述

方法 `charCodeAt()` 与 `charAt()` 执行的操作相似，只不过前者返回的是位于指定位置的字符的编码，而后者返回的则是含有字符本身的子串。如果 *n* 是负数，或者大于等于字符串的长度，则 `charCodeAt()` 返回 NaN。

要了解从 Unicode 编码创建字符串的方法，请参阅 `String.fromCharCode()`。

Bug

JavaScript 1.2（例如，由 Netscape 4.0 实现）对 16 位 Unicode 字符和字符串没有提供完全的支持。

参阅

`String.charAt()` 和 `String.fromCharCode()`

String.concat()JavaScript 1.2;JScript 3.0;ECMAScript v3

连接字符串

摘要

string.concat(*value*, ...)

参数

value, ...

要连接到 *string* 上的一个或多个值。

返回值

把每个参数都连接到字符串 *string* 上得到的新字符串。

描述

方法 `concat()` 将把它的所有参数都转换成字符串（如果必要），然后按顺序连接到字符串 *string* 的尾部，返回连接后的字符串。注意，*string* 自身并没有被修改。

`String.concat()` 与 `Array.concat()` 很相似。注意，使用 “+” 运算符来进行字符串的连接运算通常更简便一些。

参阅

`Array.concat()`

`String.fromCharCode()` JavaScript 1.2;JScript 3.0;ECMAScript v1

从字符编码创建一个字符串

摘要

`String.fromCharCode(c1, c2, ...)`

参数

c1, c2, ...

零个或多个整数，声明了要创建的字符串中的字符的 Unicode 编码。

返回值

含有指定编码的字符的新字符串。

描述

这个静态方法提供了一种创建字符串的方式，即字符串中的每个字符都由单独的数字 Unicode 编码指定。注意，作为一种静态方法，`fromCharCode()` 是构造函数 `String()` 的属性，而不是字符串或 `String` 对象的方法。

`String.charCodeAt()` 是与 `String.fromCharCode()` 配套使用的实例方法，它提供了获取字符串中单个字符的编码的方法。

例子

```
// 创建字符串 hello  
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

Bug

JavaScript 1.2 (例如, 由 Netscape 4.0 实现) 没有对 16 位 Unicode 编码的字符和字符串提供完全的支持。

参阅

String.charCodeAt()

String.indexOf()

JavaScript 1.0;JScript 1.0;ECMAScript v1

检索字符串

摘要

string.indexOf(*substring*)

string.indexOf(*substring*, *start*)

参数

substring

要在字符串 *string* 中检索的子串。

start

一个可选的整数参数, 声明了在字符串 *String* 中开始检索的位置。它的合法取值是 0 (字符串中的第一个字符的位置) 到 *string.length*-1 (字符串中的最后一个字符的位置)。如果省略了这个参数, 将从字符串的第一个字符开始检索。

返回值

如果在 *string* 中的 *start* 位置之后存在 *substring*, 返回出现的第一个 *substring* 的位置。如果没有找到了子串 *substring*, 返回 -1。

描述

方法 String.indexOf() 将从头到尾的检索字符串 *string*, 看它是否含有子串 *substring*。开始检索的位置在字符串 *string* 的 *start* 处或 *string* 的开头 (没有指定 *start* 参数时)。如果找到了一个 *substring*, 那么 String.indexOf() 将返回 *substring* 的第一个字符在 *string* 中的位置。*string* 中的字符位置是从 0 开始的。

如果在 *string* 中没有找到 *substring*, 那么 `String.indexOf()` 方法将返回 `-1`。

Bug

在 JavaScript 1.0 和 1.1 中, 如果 *start* 的值大于字符串 *string* 的长度, `indexOf()` 将返回一个空字符串, 而不是返回 `-1`。

参阅

`String.charAt()`、`String.lastIndexOf()`、`String.substring()`

String.lastIndexOf()

JavaScript 1.0;JScript 1.0;ECMAScript v1

从后向前检索一个字符串

摘要

`string.lastIndexOf(substring)`

`string.lastIndexOf(substring, start)`

参数

substring

要在字符串 *string* 中检索的子串。

start

一个可选的整数参数, 声明了在字符串 *string* 中开始检索的位置。它的合法取值是 `0` (字符串中的第一个字符的位置) 到 `string.length - 1` (字符串中的最后一个字符的位置)。如果省略了这个参数, 将从字符串的最后一个字符处开始检索。

返回值

如果在 *string* 中的 *start* 位置之前存在 *substring*, 那么返回的就是出现的最后一个 *substring* 的位置。如果没有找到子串 *substring*, 那么返回的是 `-1`。

描述

方法 `String.lastIndexOf()` 将从尾到头的检索字符串 *string*, 看它是否含有子串 *substring*。开始检索的位置在字符串 *string* 的 *start* 处或 *string* 的结尾 (没有指定 *start* 参数时)。如果找到了一个 *substring*, 那么 `String.lastIndexOf()` 将返回 *substring* 的第一个字符在 *string* 中的位置。由于是从尾到头的检索一个字符

串, 所以找到的第一个 `substring` 其实是 `string` 中出现在位置 `start` 之前的最后一个 `substring`。

如果在 `string` 中没有找到 `substring`, 那么该方法将返回 `-1`。

注意, 虽然 `String.lastIndexOf()` 是从尾到头的检索字符串 `string`, 但是它返回的字符位置仍然是从头开始计算的。字符串中第一个字符的位置是 `0`, 最后一个字符的位置是 `string.length - 1`。

参阅

`String.charAt()`、`String.indexOf()`、`String.substring()`

String.length

JavaScript 1.0; JScript 1.0; ECMAScript v1

字符串的长度

摘要

`string.length`

描述

属性 `String.length` 是一个只读整数, 它声明了指定字符串 `string` 中的字符数。对于任何一个字符串 `s`, 它最后一个字符的下标都是 `s.length - 1`。用 `for/in` 循环不能枚举出字符串的 `length` 属性, 用 `delete` 运算符也不能删除它。

String.localeCompare()

JavaScript 1.5; JScript 5.5; ECMAScript v3

用本地特定的顺序来比较两个字符串

摘要

`string.localeCompare(target)`

参数

`target`

要以本地特定的顺序与 `string` 进行比较的字符串。

返回值

说明比较结果的数字。如果 `string` 小于 `target`, 则 `localeCompare()` 返回小于 `0` 的数。如果 `string` 大于 `target`, 该方法返回大于 `0` 的数。如果两个字符串相等, 或

根据本地排序规约没有区别，该方法返回 0。

描述

把 < 和 > 运算符应用到字符串时，它们只用字符的 Unicode 编码比较字符串，而不考虑当地的排序规约。以这种方法生成的顺序不一定是正确的。例如，西班牙语中，其中字母“ch”通常作为出现在字母“c”和“d”之间的字符来排序。

`localeCompare()` 方法提供的比较字符串的方法，考虑了默认的本地排序规约。ECMAScript 标准没有规定如何进行本地特定的比较操作，它只规定该函数采用底层操作系统提供的排序规约。

例子

可以用下列代码，按照地方特定的排序规约对一个字符串数组排序。

```
var strings; // 要排序的字符串数组，可以在任何地方初始化
strings.sort(function(a,b) { return a.localeCompare(b); });
```

String.match()

JavaScript 1.2;JScript 3.0;ECMAScript v3

找到一个或多个正则表达式的匹配

摘要

`string.match(regex)`

参数

regex

声明了要匹配的模式 `RegExp` 对象。如果该参数不是 `RegExp` 对象，则首先将它传递给 `RegExp()` 构造函数，把它转换成 `RegExp` 对象。

返回值

存放匹配结果的数组。该数组的内容依赖于 `regex` 是否具有全局性质 `g`。下面详细说明了这个返回值。

描述

方法 `match()` 将检索字符串 `string`，以找到一个或多个与 `regex` 匹配的文本。这个方法的行为很大程度上依赖于 `regex` 是否具有性质 `g`。关于正则表达式的完整细节，请参阅第十章。

如果 *regexp* 没有性质 *g*，那么 *match()* 就只能在 *string* 中执行一次匹配。如果没有找到任何匹配的文本，*match()* 将返回 *null*。否则，它将返回一个数组，其中存放了与它找到的匹配文本有关的信息。该数组的第 0 个元素存放的是匹配文本，其余的元素存放的是与正则表达式的子表达式匹配的文本。除了这些常规的数组元素之外，返回的数组还含有两个对象属性：*index* 属性声明的是匹配文本的起始字符在 *string* 中的位置，*input* 属性声明的是对 *string* 的引用。

如果 *regexp* 具有标志 *g*，那么 *match()* 将执行全局检索，找到 *string* 中的所有匹配子串。如果没有找到任何匹配的子串，它将返回 *null*。如果找到了一个或多个匹配子串，它将返回一个数组。不过，全局匹配返回的数组的内容与前者大不相同，它的数组元素存放的是 *string* 中的所有匹配子串，而且它也没有 *index* 属性和 *input* 属性。注意，在全局匹配的模式下，*match()* 既不提供与子表达式匹配的文本的信息，也不声明每个匹配子串的位置。如果你需要这些全局检索的信息，可以使用 *RegExp.exec()*。

例子

下面的全局匹配可以找到字符串中的所有数字：

```
1: plus 2 equals 3'.match(/\d+/g) // 返回 ["1", "2", "3"]
```

下面的非全局匹配使用了更加复杂的正则表达式，它具有几个用括号括起来的子表达式。与该表达式匹配的是一个 URL，与它的子表达式匹配的是那个 URL 的协议部分、主机部分和路径部分：

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0]; // 包含 "http://www.isp.com/~david"
    var protocol = result[1]; // 包含 "http"
    var host = result[2]; // 包含 "www.isp.com"
    var path = result[3]; // 包含 "~david"
}
```

参阅

RegExp、*RegExp.exec()*、*RegExp.test()*、*String.replace()*、*String.search()*；第十章

String.replace()

JavaScript 1.2; JScript 3.0; ECMAScript v3

替换一个与正则表达式匹配的子串

摘要

```
string.replace(regex, replacement)
```

参数

regex

声明了要替换的模式的 **RegExp** 对象。如果该参数是一个字符串，则将它作为要检索的直接量文本模式，而不是首先被转换成 **RegExp** 对象。

replacement

一个字符串，声明的是替换文本或生成替换文本的函数。详见描述部分。

返回值

一个新字符串，是用 *replacement* 替换了与 *regex* 的第一次匹配或所有匹配之后得到的。

描述

字符串 *string* 的方法 *replace()* 执行的是查找并替换的操作。它将在 *string* 中查找与 *regex* 相匹配的子串，然后用 *replacement* 替换这些子串。如果 *regex* 具有全局性质 *g*，那么 *replace()* 将替换所有的匹配子串。否则，它只替换第一个匹配子串。

replacement 可能是字符串或函数。如果它是一个字符串，那么每个匹配都将由字符串替换。但 *replacement* 中的 *\$* 字符具有特殊的含义。如下表所示，它说明从模式匹配得到的字符串将用于替换。

字符	替换文本
<i>\$1</i> , <i>\$2</i> , ..., <i>\$99</i>	与 <i>regex</i> 中的第 1 到第 99 个子表达式相匹配的文本
<i>&</i>	与 <i>regex</i> 相匹配的子串
<i>\$`</i>	位于匹配子串左侧的文本
<i>\$'</i>	位于匹配子串右侧的文本
<i>\$\$</i>	直接量 <i>\$</i> 符号。

ECMAScript v3 规定，*replace()* 方法的参数 *replacement* 可以是函数而不是字符

串。JavaScript 1.2 和 JScript 5.5 实现了这一特性。在这种情况下，每个匹配都调用该函数，它返回的字符串将作为替换文本使用。该函数的第一个参数是匹配模式的字符串。接下来的参数是与模式中的子表达式匹配的字符串，可以有 0 个或多个这样的参数。接下来参数是一个整数，声明了匹配在 *string* 中出现的位置。最后一个参数是 *string* 自身。

例子

要确保单词“JavaScript”中的大写字符是正确的，可用下列代码：

```
text.replace(/javascript/i, "JavaScript");
```

要将名字“Doe, John”转换成“John Doe”的形式，可用下列代码：

```
name.replace(/(\w+)\s*(\w+)/, "$2 $1");
```

用花引号替换直引号，可用下列代码：

```
text.replace(/'([^\']*')/g, "`$1`");
```

使字符串中所有单词的第一个字母都是大写的，可用下列代码：

```
text.replace(/\b\w+/g, function(word) {  
    return word.substring(0,1).toUpperCase() +  
           word.substring(1);  
});
```

参阅

RegExp、RegExp.exec()、RegExp.test()、String.match()、String.search(); 第十章

String.search() JavaScript 1.2; JScript 3.0; ECMAScript v3

检索与正则表达式相匹配的子串

摘要

```
string.search(regex)
```

参数

regex

要在字符串 *string* 中检索的 RegExp 对象，该对象具有指定的模式。如果该参数不是 RegExp 对象，则首先将它传递给 `RegExp()` 构造函数，把它转换成 RegExp 对象。

返回值

string 中第一个与 *regexp* 相匹配的子串的起始位置。如果没有找到任何匹配的子串, 则返回 -1。

描述

方法 *search()* 将在字符串 *string* 中检索与 *regexp* 相匹配的子串, 并且返回第一个匹配子串的第一个字符的位置。如果没有找到任何匹配的子串, 则返回 -1。

search() 并不执行全局匹配, 它将忽略标志 *g*。它也忽略 *regexp* 的 *lastIndex* 属性, 并且总是从字符串的开始进行检索, 这意味着它总是返回 *string* 的第一个匹配的位置。

例子

```
var s = 'JavaScript is fun';
s.search(/script/i) // 返回 4
s.search(/a(.))a/) // 返回 1
```

参阅

RegExp、*RegExp.exec()*、*RegExp.test()*、*String.match()*、*String.replace()*; 第十章

String.slice() JavaScript 1.2;JScript 3.0;ECMAScript v3

抽取一个子串

摘要

string.slice(start, end)

参数

start

要抽取的片段的起始下标。如果是负数, 那么该参数声明了从字符串的尾部开始算起的位置。也就是说, -1 指字符串中的最后一个字符, -2 指倒数第二个字符, 以此类推。

end

紧接着要抽取的片段的结尾的下标。如果没有指定这一参数, 那么要抽取的子串包括 *start* 到原字符串结尾的字符串。如果该参数是负数, 那么它声明了从字符串的尾部开始算起的位置。

返回值

一个新字符串，包括字符串 *string* 从 *start* 开始（包括 *start*）到 *end* 为止（不包括 *end*）的所有字符。

描述

方法 `slice()` 将返回一个含有字符串 *string* 的片段的字符串或返回它的一个子串。但是该方法不修改 *string*。

String 对象的方法 `slice()`、`substring()` 和 `substr()`（不建议使用）都返回字符串的指定部分。`slice()` 比 `substring()` 要灵活一些，因为它允许使用负数作为参数。`slice()` 与 `substr()` 有所不同，因为它用两个字符的位置指定子串，而 `substr()` 则用字符位置和长度来指定子串。还要注意的，`String.slice()` 与 `Array.slice()` 相似。

例子

```
var s = "abcdefg";
s.slice(0,4)    // 返回 "abcd"
s.slice(2,4)    // 返回 "cd"
s.slice(4)      // 返回 "efg"
s.slice(3,-1)   // 返回 "def"
s.slice(3,-2)   // 返回 "de"
s.slice(-3,-1)  // 应该返回 "ef"；在 IE 4 中返回 "abcdef"
```

Bug

在 JScript 3.0 (Internet Explorer 4) 中，参数 *start* 的值是不能是负数。负的 *start* 值指定的不是从字符串尾部开始算起的字符位置，而是指定第 0 个字符的位置。

参阅

`Array.slice()`、`String.substring()`

String.split() JavaScript 1.1;JScript 3.0;ECMAScript v1;在 ECMAScript v3 中增强

将字符串分割成字符串数组

摘要

`string.split(delimiter, limit)`

参数

delimiter

字符串或正则表达式，从该参数指定的地方分割 *string*。ECMAScript v3 标准

化了正则表达式作为定界符使用时的用法, JavaScript 1.2 和 JScript 3.0 实现了它。JavaScript 1.1 没有实现它。

limit

这个可选的整数指定了返回的数组的最大长度。如果设置了该参数, 返回的子串不会多于这个参数指定的数字。如果没有设置该参数, 整个字符串都会被分割, 不考虑它的长度。ECMAScript v3 标准化了该参数, JavaScript 1.2 和 JScript 3.0 实现了它。JavaScript 1.1 没有实现它。

返回值

一个字符串数组, 是通过在 *delimiter* 指定的边界处将字符串 *string* 分割成子串创建的。返回的数组中的子串不包括 *delimiter* 自身, 但下面列出的情况除外。

描述

方法 `split()` 将创建并返回一个字符串数组, 该数组中的元素是指定的字符串 *string* 的子串, 最多具有 *limit* 个。这些子串是通过从头到尾检索字符串中与 *delimiter* 匹配的文本, 在匹配文本之前和之后分割 *string* 得到的。返回的子串中不包括定界符文本 (下面提到的情况除外)。如果定界符从字符串开头开始匹配, 返回的数组的第一个元素是空串, 即出现在定界符之前的文本。同样, 如果定界符与字符串的结尾匹配, 返回的数组的最后一个元素也是空串 (假定与 *limit* 没有冲突)。

如果没有指定 *delimiter*, 那么它根本就不对 *string* 执行分割, 返回的数组中只有一个元素, 而不分割字符串元素。如果 *delimiter* 是一个空串或与空串匹配的正则表达式, 那么 *string* 中的每个字符之间都会被分割, 返回的数组的长度与字符串长度相等 (假定 *limit* 不小于该长度)。(注意, 这是一种特殊情况, 因为没有匹配第一个字符之前和最后一个字符之后的空串。)

前面说过, 该方法返回的数组中的子串不包括用于分割字符串的定界符文本。但如果 *delimiter* 是包括子表达式的正则表达式, 那么返回的数组中包括与这些子表达式匹配的子串 (但不包括与整个正则表达式匹配的文本)。

注意, `String.split()` 执行的操作与 `Array.join()` 执行的操作相反。

例子

在使用结构复杂的字符串时, 方法 `split()` 最有用。例如:

```
"1:2:3:4:5".split(':'); // 返回 ["1", "2", "3", "4", "5"]
```

```
"a b c".split(' '); // 返回 ['a', 'b', 'c', '']
```

`split()` 方法的另一个常见用法是解析命令和与之相似的字符串, 用空格将它们分割成单词:

```
var words = sentence.split(' ');
```

关于 `delimiter` 是一个空格的情况, 详见“Bug”小节。用正则表达式作为定界符, 很容易把字符串分割成单词:

```
var words = sentence.split(/ \b+/);
```

要把字符串分割成字符数组, 可以用空串作为定界符。如果只想把字符串的前一部分分割成字符数组, 需要使用 `limit` 参数:

```
"hello".split(''); // 返回 ['h','e','l','l','o']  
"hello".split("", 3); // 返回 ['h','e','l']
```

如果想使返回的数组包括定界符或定界符的一个或多个部分, 可以使用带子表达式的正则表达式。例如, 下面的代码将在 HTML 标记处分割字符串, 返回的数组中包括这些标记:

```
var text = 'hello <b>world</b>';  
text.split(/(<[^>]*>)/); // 返回 ["hello ", "<b>", "world", "</b>", ""]
```

Bug

在 JavaScript 的 Netscape 实现中, 如果明确地把语言版本设置为 1.2 (如把 `<script>` 标记的 `language` 性质设置为 1.2), `split()` 方法具有特殊的行为, 即如果 `delimiter` 是一个空格, 该方法将在空格处分割字符串, 忽略字符串开头和结尾处的空白。详见第 11.6 小节。

参阅

`Array.join()`、`RegExp`; 第十章

String.substr()

JavaScript 1.2; JScript 3.0; 反对使用

抽取一个子串

摘要

`string.substr(start, length)`

参数

start

要抽取的子串的起始下标。如果是一个负数, 那么该参数声明从字符串的尾部开始算起的位置。也就是说, -1 指字符串中的最后一个字符, -2 指倒数第二个字符, 以此类推。

length

子串中的字符数。如果省略了这个参数, 那么返回从 *string* 的开始位置到结尾的子串。

返回值

一个字符串的副本, 包括从 *string* 的 *start* 处(包括 *start* 所指的字符)开始的 *length* 个字符。如果没有指定 *length*, 返回的字符串包含从 *start* 到 *string* 结尾的字符。

描述

`substr()` 将在 *string* 中抽取并返回一个子串。但是它并不修改 *string*。

注意, `substr()` 指定的是子串的开始位置和长度, 它是 `String.substring()` 和 `String.splice()` 的一种有用的替代方法, 后两者指定的都是起始字符的位置。但要注意, ECMAScript 没有标准化该方法, 因此反对使用它。

例子

```
var s = 'abcdefg';
s.substr(2,2);    // 返回 'cd'
s.substr(3);      // 返回 'defg'
s.substr(-3,2);   // 应该返回 'ef'; 在 IE 4 中返回 'ab'
```

Bug

在 JScript 3.0 (Internet Explorer 4) 中, 参数 *start* 的值不能为负数。负的 *start* 值指定的不是从字符串尾部开始算起的字符位置, 而是第 0 个字符的位置。

参阅

`String.slice()`、`String.substring()`

String.substring()

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回字符串的一个子串

摘要

string.substring(from, to)

参数

from

一个整数，声明了要抽取的子串的第一个字符在 *string* 中的位置。

to

一个可选的整数，比要抽取的子串的最后 一个字符在 *string* 中的位置多 1。如果省略了该参数，返回的子串直到字符串的结尾。

返回值

一个新字符串，其长度为 *to - from*，存放的是字符串 *string* 的一个子串。这个新字符串含有的字符是从 *string* 中的 *from* 处到 *to - 1* 处复制的。

描述

`String.substring()` 将返回字符串 *string* 的子串，由 *from* 到 *to* 之间的字符构成，包括位于 *from* 的字符，不包括位于 *to* 的字符。

如果参数 *from* 与 *to* 相等，那么该方法返回的就是一个空串（即长度为 0 的字符串）。如果 *from* 比 *to* 大，那么该方法在抽取子串之前会先交换这两个参数。

要记住，该子串包括 *from* 处的字符，不包括 *to* 处的字符。虽然这样看来有违直觉，但这种系统一个值得注意重要特性是，返回的子串的长度总等于 *to - from*。

Bug

在 JavaScript 的 Netscape 实现中，如果明确地把语言版本设置为 1.2（如把 `<script>` 标记的 `language` 性质设置为 1.2），那么如果 *from* 比 *to* 大，该方法不能正确地交换它的参数，而是返回空串。

参阅

`String.charAt()`、`String.indexOf()`、`String.lastIndexOf()`、`String.slice()` 和 `String.substr()`

`String.toLocaleLowerCase()`

JavaScript 1.5; JScript 5.5; ECMAScript v3

把字符串转换小写

摘要

`string.toLocaleLowerCase()`

返回值

string 的一个副本，按照本地方式转换成小写字母。只有几种语言（如土耳其语）具有地方特有的大小写映射，所以该方法的返回值通常与 `toLowerCase()` 一样。

参阅

`String.toLocaleUpperCase()`、`String.toLowerCase()`、`String.toUpperCase()`

`String.toLocaleUpperCase()`

JavaScript 1.5;JScript 5.5;ECMAScript v3

将字符串转换成大写

摘要

`string.toLocaleUpperCase()`

返回值

string 的一个副本，按照本地方式转换成了大写字母。只有几种语言（如土耳其语）具有地方特有的大小写映射，所以该方法的返回值通常与 `toUpperCase()` 一样。

参阅

`String.toLocaleLowerCase()`、`String.toLowerCase()`、`String.toUpperCase()`

`String.toLowerCase()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

将字符串转换成小写

摘要

`string.toLowerCase()`

返回值

string 的一个副本，其中所有大写字符都被转换成了小写字符。

`String.toString()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回字符串

覆盖 `Object.toString()`

摘要

`string.toString()`

返回值

string 的原始字符串值。一般不会调用该方法。

抛出

`TypeError`

调用该方法的对象不是 `String` 时抛出该异常。

参阅

`String.valueOf()`

`String.toUpperCase()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

将字符串转换成大写

摘要

string.toUpperCase()

返回值

string 的一个副本，其中所有小写字符都被转换成了大写的。

`String.valueOf()`

JavaScript 1.0;JScript 1.0;ECMAScript v1

返回字符串

覆盖 `Object.valueOf()`

摘要

string.valueOf()

返回值

string 的原始字符串值。

抛出

`TypeError`

调用该方法的对象不是 `String` 时抛出该异常。

参阅

`String.toString()`

SyntaxError

JavaScript 1.5;JScript 5.5;ECMAScript v3

抛出该错误用来通知语法错

继承 Error 类

构造函数

```
new SyntaxError()
```

```
new SyntaxError(message)
```

参数

message

提供异常细节的错误消息（可选）。如果设置了该参数，它将作为 SyntaxError 对象的 *message* 属性的值。

返回值

新构造的 SyntaxError 对象。如果指定了参数 *message*，该 Error 对象将它作为 *message* 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果不用 *new* 运算符，把 `SyntaxError()` 构造函数当作函数调用，它的行为与使用 *new* 运算符调用时一样。

属性

message

提供异常细节的错误消息。该属性存放传递给构造函数的字符串，或存放实现定义的默认字符串。详见 “Error.message”。

name

声明异常类型的字符串。所有 SyntaxError 对象的 *name* 属性都继承值 “SyntaxError”。

描述

SyntaxError 类的一个实例会被抛出以通知 JavaScript 代码中的语法错误。`eval()` 方法、`Function()` 构造函数和 `RegExp()` 构造函数都可能抛出这种类型的异常。关于抛出和捕捉异常的细节，请参阅 “Error”。

参阅

Error、Error.message、Error.name

TypeError

JavaScript 1.5;JScript 5.5;ECMAScript v3

当一个值的类型错误时，抛出该异常

继承 Error 类

构造函数

`new TypeError()``new TypeError(message)`

参数

`message`

提供异常细节的错误消息（可选）。如果设置了该参数，它将作为 `TypeError` 对象的 `message` 属性的值。

返回值

新构造的 `TypeError` 对象。如果指定了参数 `message`，该 `Error` 对象将它作为 `message` 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果不用 `new` 运算符，则把 `TypeError()` 构造函数当作函数调用，它的行为与使用 `new` 运算符调用时一样。

属性

`message`

提供异常细节的错误消息。该属性存放传递给构造函数的字符串，或存放实现定义的默认字符串。详见 “`Error.message`”。

`name`

声明异常类型的字符串。所有 `TypeError` 对象的 `name` 属性都继承值 “`TypeError`”。

描述

当一个值的类型与要求不符时，`TypeError` 类的一个实例就会被抛出。在访问值为 `null` 或 `undefined` 的属性时，这种情况经常发生。如果由一个对象（其他类的实例）调用另一个类定义的方法，或者对于不是构造函数的值使用 `new` 运算符时，也会发生这种情况。当调用内部函数或方法时，如果传递的参数多于期望的个数，JavaScript 的实现也允许抛出 `TypeError` 异常。关于抛出和捕捉异常的细节，请参阅 “`Error`”。

参阅

`Error`、`Error.message`、`Error.name`

undefined

JavaScript 1.5;JScript 5.5;ECMAScript v3

未定义值

摘要

undefined

描述

undefined 是全局属性，存放 JavaScript 的 undefined 值。它与尝试读取不存在的对象属性的值时返回的值相同。用 for/in 循环不能枚举出 undefined 属性，用 delete 运算符也不能删除它。注意，undefined 不是常量，可以设置为其他值。

只能用 === 运算符测试一个值是否是未定义的，因为 == 运算符认为 undefined 值等价于 null。

unescape()

JavaScript 1.0;JScript 1.0;ECMAScript v1;ECMAScript v3 反对使用

给转义字符串解码

摘要

unescape(s)

参数

s 要解码或“反转义”的字符串。

返回值

s 解码后的一个副本。

描述

unescape() 是全局函数，对 escape() 编码的字符串解码。该函数是通过找到形式为 %xx 和 %uxxxx 的字符序列（这里 x 表示十六进制的数字），用 Unicode 字符 \u00xx 和 \uxxxx 替换这样的字符序列进行解码的。

虽然 ECMAScript 的第一个版本标准化了 unescape()，但 ECMAScript v3 从标准中删除了它，反对使用该方法。虽然 ECMAScript 的实现可能实现了该函数，但这不是必需的。在 JavaScript 1.5 和 JScript 5.5 及其后的版本中，应该用 decodeURI() 和 decodeURIComponent() 代替 unescape()。详见“escape()”。

参阅

`decodeURI()`、`decodeURIComponent()`、`escape()`、`String`

URIError

JavaScript 1.5;JScript 5.5;ECMAScript v3

由 URI 的编码和解码方法抛出

继承 `Error` 类

构造函数

`new URIError()`

`new URIError(message)`

参数

message

提供异常细节的错误消息（可选）。如果设置了该参数，它将作为 `URIError` 对象的 `message` 属性的值。

返回值

新构造的 `URIError` 对象。如果指定了参数 *message*，该 `Error` 对象将它作为 `message` 属性的值，否则，它将用实现定义的默认字符串作为该属性的值。如果不用 `new` 运算符，而把 `URIError()` 构造函数当作函数调用，它的行为与使用 `new` 运算符调用时一样。

属性

`message`

提供异常细节的错误消息。该属性存放传递给构造函数的字符串，或存放实现定义的默认字符串。详见“`Error.message`”。

`name`

声明异常类型的字符串。所有 `URIError` 对象的 `name` 属性都继承值“`URIError`”。

描述

如果指定的字符串含有不合法的十六进制转义序列，则 `decodeURI()` 或 `decodeURIComponent()` 方法就会抛出 `URIError` 类的实例。如果指定的字符串含有不合法的 Unicode 替代对，`encodeURI()` 或 `encodeURIComponent()` 方法也会抛出该异常。关于抛出和捕捉异常的细节，请参阅“`Error`”。

参阅

`Error`、`Error.message`、`Error.name`

第四部分

客户端 JavaScript 参考手册

本书的这一部分是客户端JavaScript中所有对象、属性、函数、方法和事件处理程序的完整参考手册。这一部分的开头解释了如何使用该参考手册。

客户端 JavaScript

参考手册

这一部分是一个参考手册，记录了支持客户端JavaScript的Web浏览器定义的类、方法、属性和事件处理程序。这些类、方法和属性构成了实际标准——0级DOM API。刚开始编写向后兼容的脚本和程序的人，需要结合第三部分JavaScript核心参考手册使用这个参考手册。引言和示例参考页解释了如何最大程度地利用这个参考手册。仔细阅读这些材料，可以更容易地找到和使用自己需要的信息。

这个参考手册是按照字母顺序安排的。类的属性和方法的参考页都以它们全称的字母顺序排列，包含定义它们的类的名字。例如，如果要阅读Form类的submit()方法，应该在“Form.submit”下查看，而不是在“submit”下查看。

为了节省本书的篇幅，这个参考手册中的大部分属性没有自己的参考页（但所有方法和事件处理程序都有自己的参考页）。简单的属性在定义它们的类的参考页中都有完整的说明。例如，可以在“Document”参考页中读到Document类的images[]属性的说明。需要大量解释的重要属性都有自己的参考页，在定义这些属性的类或接口的参考页中可以找到对这些参考页的交叉引用。例如，在“Document”参考页中查看cookie属性或在“Window”参考页中查看status属性时，可以看到对该属性的简短描述和对“Document.cookie”和“Window.status”参考页的引用。

客户端JavaScript具有大量全局属性和函数，如window、history和alert()。在客户端JavaScript中，Window对象是全局对象，客户端JavaScript的全局属性和函数实际上都是Window类的属性。因此，在客户端参考手册部分，全局属性和函数都在“Window”参考页或“Window.alert()”条目下记录。

有时,你会发现,自己不知道想查找的方法或属性定义的类或接口名,或者不确定应该在哪个参考手册中查找一个类或接口。第六部分是一个专用的索引,用来解决这些问题。查找一个类、方法或属性的名字,它将告诉你应该在哪个参考手册中查找,以及在参考手册的哪个类下。例如,如果查找“Button”,它将告诉你 Button 类位于客户端参考手册部分。如果查找名字“alert”,它将告诉你,alert()是客户端 Window 类的方法。

一旦找到了要查询的参考页,那么找到你需要的信息就没有太大的困难了。不过,如果知道参考页是如何编写和组织,就能更好地使用参考页。接下来是一个示例参考页,标题为“示例条目”,它示范了每个参考页的结构,告诉你在哪里可以找到什么类型的信息。在研究参考手册的其余部分之前,应该花时间阅读该参考页。

示例条目

可用性

如何阅读客户端参考手册

继承性

标题和简短的描述

每个参考条目都以一个四部分的标题块开头,如上所示。各个条目按照标题的字母顺序排列。标题行下的简短说明概览了该条目中记述的项目,它可以帮助你快速地判断是否需要阅读该参考页余下的部分。

可用性

标题块的右上角是可用性信息。它将告诉你类、方法或事件处理程序是何时引入的。对于可移植性差的条目,这个部分说明了哪些 Netscape 版本和 Internet Explorer 版本支持它。如果一个条目得到 web 浏览器的良好支持,而且由同一代 Netscape 和 IE 浏览器引入了对它的支持,那么这个部分说明了它在 JavaScript 核心语言版本中的可用性。可以用第一章中的表来判断这些版本对应的 Netscape 和 Internet Explorer 版本。当然,由于大多数属性没有自己的参考页,所以它们没有可用性信息。如果一个属性的可用性不同于定义它的类,在该属性的描述部分会说明这个情况。

继承性

如果一个类继承了超类,该信息将显示在标题块的右下角。第八章说过,JavaScript 类可以继承其他类的属性和方法。例如,Button 类继承 Input 类,Input 类又继承 HTMLElement 类。在看到这些继承信息时,你可能需要查询列出的超类的信息。

摘要

每个参考页都有“摘要”部分，说明了在代码中如何使用类、方法和事件处理程序。例如，Form 类的摘要如下：

```
document.form_name  
document.forms[form_number]
```

摘要说明了引用 Form 对象的两种不同方式。采用 *this* 字体的文本必须原原本本地输入。斜体字说明要替换的文本。*form_name* 应该用表单名替换，*form_number* 应该用表单在 *forms[]* 数组中的下标代替。同样，*document* 应该用对 Document 对象的引用代替。查看“Document”参考页的“摘要”部分，我们发现它也有两种形式：

```
document  
window.document
```

也就是说，可以用直接量 *document* 或 *window.document* 替换 *document*。如果你选择后者，就应该查看 Window 类的摘要，弄清楚如何引用 Window 对象，即用什么替换 *window*。

参数

如果参考页说明的是方法，那么在“摘要”后还有“参数”部分，这部分说明了这个方法的参数。如果方法没有参数，这个部分将被省略。

arg1

参数用列表描述。例如，这就是对参数 *arg1* 的描述。

arg2

这是对参数 *arg2* 的描述。

返回值

这个部分解释了方法的返回值。如果该方法没有返回值，这个部分将被省略。

构造函数

如果参考页说明的类有构造函数，则用这个部分说明如何用构造函数创建类的实例。由于构造函数是一种方法，“构造函数”部分与方法参考页的“摘要”部分看起来很像，它也包含“参数”子部分。

属性

如果参考页说明的是一个类，“属性”部分就会列出和说明这个类定义的属性。在客户端参考手册中，只有特别复杂的属性才有自己的参考页。

`prop1`

属性 `prop1` 的说明，包括它的类型、作用或含义以及它是只读的还是可读可写的。

`prop2`

同 `prop1`。

方法

定义了方法的类的参考页还包括“方法”部分，这部分列出了每个方法的名字，并提供了简短的说明。在单独的参考页中可以找到每个方法的完整说明。

事件处理程序

定义了事件处理程序的类的参考页还包括“事件处理程序”部分，这部分列出了每个处理程序的名字，并提供了简短的说明。在单独的参考页中可以找到每个事件处理程序的完整说明。

HTML 语法

许多客户端 JavaScript 类都有相应的 HTML 元素。这些类的参考页包括说明创建与 JavaScript 对象对应的 HTML 元素的 HTML 语法。

描述

大多数参考页都具有“描述”部分，该部分对要说明的类、方法、函数进行了基本描述。它是参考页的核心。如果你是第一次学习类、方法或事件处理程序，可以直接跳到这个部分，然后返回查看前面的“参数”、“属性”和“方法”部分。如果你已经熟悉了类、方法或属性，可以不必再阅读这一部分，只需要快速查看某些特定的信息（例如“参数”部分或“属性”部分）即可。

在某些参考页中，这一部分只是一小段。但是在某些参考页中，这一部分可能会占用一页的篇幅，甚至会更多。对于那些非常简单的方法，“参数”和“返回值”部分已经足以说明它，所以它的“描述”部分就被省略了。

示例

有些参考页还包括一个示例，说明该项目的典型用法。但大多数参考页没有示例部分，可以在本书的前半部分找到它们的示例。

Bug

当一个项目运行得不太正常时，这一部分描述了导致这种现象的 bug。但要注意，本书并不打算列出所有 JavaScript 版本和实现中的每一个 bug。

参阅

许多参考页的结尾都包括对相关的参考页的交叉引用。大多数交叉引用是对客户端参考手册中的其他参考页的引用，有些是对类参考页中包含的属性说明的引用，还有一些是对 DOM 参考手册中的相关参考页或本书前两个部分中的章节的引用。

Anchor

JavaScript 1.2

超文本链接的目标

继承 HTMLElement

摘要

```
document.anchors[i]  
document.anchors.length
```

属性

Anchor 对象继承了 HTMLElement 的属性，而且还定义或覆盖了以下的属性：

name

Anchor 对象的名字。该属性的值由标记 <a> 的 name 性质设置。

text [*Netscape 4*]

该属性声明了锚标记 <a> 和 之间的纯文本（如果存在）。注意，只有当标记 <a> 和 之间没有其他 HTML 标记时，该属性才能正确作用。如果还有其他 HTML 标记，text 属性只存放部分锚文本。

IE4 中的 HTMLElement.innerText 的作用和这个在 Netscape 中专用属性作用相同。

HTML 语法

锚对象是由包含 <name> 性质的标准 HTML 标记 <a> 创建的：

```
<a
  name="name"  // 链接可以使用这个名字来引用锚
>
text
</a>
```

描述

锚是HTML文档中一个命名了的地点，由具有name性质的标记<a>创建。Document对象具有包含Anchor对象的一个数组属性anchors[]，它的元素代表文档中所包含的锚。虽然这个数组从JavaScript 1.0起就已经存在了，但是直到JavaScript 1.2才实现了Anchor对象。因此，在JavaScript 1.2出现之前，anchors[]数组的元素都是null。

注意，用来创建锚的标记<a>还可以用来创建超文本链接。虽然在HTML中，超文本链接常常被称为锚，但是在JavaScript中代表它的则是Link对象，而不是Anchor对象。在本书的DOM参考手册部分，锚和链接都记录在HTMLAnchorElement下。

参阅

Document对象的anchors[]属性、Link; DOM参考手册部分的HTMLAnchorElement

Applet

JavaScript 1.1

嵌在网页中的小程序

摘要

```
document.applets[i]
document.appletName
```

属性

Applet对象的属性和它所代表的Java小程序的公有字段相同。

方法

Applet对象的方法和它所代表的Java小程序的公有方法相同。

描述

Applet对象代表嵌入在HTML文档中的Java小程序，它的属性代表这个Java小程序的公有字段，它的方法代表这个Java小程序的公有方法。Netscape的LiveConnect技术和Internet Explorer的ActiveX技术都允许JavaScript程序使用Applet对象读写相应的Java小程序的属性并调用它的方法。要了解详细的内容，请参阅第二十二章。

我们知道Java是一种强类型语言。这意味着小程序的每一个域都要明确地声明为某种数据类型,把其他类型的值赋给它会引发运行时的错误。这种规则同样适用于小程序的方法,即每个参数都有一个指定的类型,而且不能像在JavaScript中那样随意地省略参数。

参阅

JavaScript; 第二十章

Area

参看 [Link](#)

Button

JavaScript 1.0;在 JavaScript 1.1 中增强它

图形化的按钮

继承 Input, HTMLElement

摘要

`form.button_name`

`form.elements[i]`

属性

Button对象继承了Input对象和HTMLElement对象的属性,此外还定义或覆盖了下面的属性:

value

一个字符串属性,声明了要显示在Button元素上的文本。该属性的值由创建按钮的HTML标记的value性质设置。在不能回流文档内容的浏览器中,该属性是只读的。

方法

Button对象继承了Input对象和HTMLElement对象的方法。

事件处理程序

Button对象继承了Input对象和HTMLElement对象的事件处理程序,此外还定义或覆盖了下面的事件处理程序:

onclick

当按钮被点击时调用。

HTML 语法

Button 元素是由标准的 HTML 标记 `<input>` 创建的:

```
<form>
...
  <input
    type = 'button'           // 声明这是一个按钮
    value  "label"           // 要显示在按钮上的文本
                                // 设定了 value 属性
    [name = "name" ]         // 此后可以用来引用这个按钮的名字
                                // 设定了 name 属性
    [ onclick = handler" ]    // 当按钮被点击时要执行的 JavaScript 语句
  >
...
</form>
```

Button 对象也可以用 HTML4 的 `<button>` 标记创建。

```
<button id="name"
        onclick='handler'>
  label
</button>
```

描述

Button 元素代表的是 HTML 文档表单内的图形化按钮。属性 `value` 包含的是要显示在按钮上的文本。属性 `name` 是要引用该按钮时使用的名字。事件处理程序 `onclick` 是在用户点击了按钮时调用的。

习惯用法

如果想让用户在你的网页中引发某项动作, 可以使用 Button 元素。虽然也可以使用 Link 对象来实现这一点, 但是除非某个期望的动作一定要用超文本链接引发, 否则选择 Button 对象比选择 Link 对象好, 因为对用户来说, 前者更明确地指出了它可以引发的某个动作。

注意, Submit 元素和 Reset 元素都是某种类型的 Button 元素, 它们分别用于提交表单和重置表单的值。这些默认动作对于一个表单来说够用了, 不必再创建其他类型的按钮了。

例子

```
<form name= "form">
  <input type="button"
    name= "press_me_button"
    value="Press Me"
    onclick='username = prompt("What is your name?");'
  >
</form>
```

参阅

Form、HTMLElement、Input、Reset、Submit: DOM 参考手册部分的 HTMLInputElement

Button.onclick

JavaScript 1.0

当 Button 被点击时调用的处理程序

摘要

```
<input type="button" value="button-text" onclick="handler">
button.onclick
```

描述

Button对象的onclick属性引用的是用户点击按钮时要调用的事件处理程序。要了解完整的细节,请参阅HTMLElement.onclick。但是要注意,与通用化的HTMLElement.onclick不同,从JavaScript 1.0开始,就支持Button.onclick。

参阅

HTMLElement.onclick; 第十九章; DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

Checkbox

JavaScript 1.0;在 JavaScript 1.1 中增强它

图形化的复选框

继承 Input, HTMLElement

摘要

可以使用如下方法引用一个具有惟一名字的 Checkbox 元素:

```
form.checkbox_name
form.elements[i]
```

如果表单含有一组具有相同名字的复选框，那么它们存放在一个数组中，可以使用如下的方式来引用它们：

```
form.checkbox_name[j]  
form.checkbox_name.length
```

属性

Checkbox 对象继承了 Input 对象和 HTMLElement 对象的属性，此外还定义或覆盖了下面一些属性：

checked

一个可读可写的布尔属性。如果选中该复选框，checked 属性为 true。如果没有选中该复选框，该属性为 false。

如果把 checked 属性设为 true，复选框将显示为被选中。同样，如果把它设为 false，复选框将显示为没有选中。注意，设置 checked 属性，不会调用 CheckBox 元素的 onclick 事件处理程序。

defaultChecked

一个只读的布尔值，声明了复选框的初始状态。如果复选框开始时被选中，即如果 HTML 标记 <input> 中出现了性质 checked，它的值为 true。如果没有出现 checked 性质，复选框初始时没有被选中，则 defaultChecked 为 false。

value

一个可读可写的字符串属性，声明了在提交表单时如果该复选框被选中了应该传递给 Web 服务器的文本。该属性的初始值由定义复选框的 HTML 标记 <input> 的 value 性质设置。如果没有设置 value 性质，默认的 value 串是 “on”。

注意，value 域没有声明复选框是否被选中了，checked 属性才声明了复选框的当前状态。如果定义一组相关的复选框，这些复选框在提交给服务器的表单中共享一个名字，那么给每个复选框不同的 value 性质很重要。

方法

Checkbox 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Checkbox 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序，此外还定义或覆盖了下面列出的事件处理程序：

onclick

当复选框被选中时调用。

HTML 语法

Checkbox 元素是由标准的 HTML 标记 `<input>` 创建的。多个 Checkbox 元素通常是分组创建的，只要声明多个具有相同 `name` 性质的 `<input>` 标记即可。

```
<form>
...
<input
  type = "checkbox"           // 声明这是一个复选框
  [ name = "name" ]       // 此后可以用来引用这个复选框或者复选框组的名字
                           // 设定了 name 属性
  [ value = "value" ]     // 当复选框被选中时返回的值
                           // 设定了 value 属性
  [checked]               // 声明了复选框初始时被选中
                           // 设定 defaultChecked 属性
  [ onclick = 'handler' ] // 当点击复选框时要执行的 JavaScript 语句
>
label                     // 紧邻复选框的 HTML 文本
...
</form>
```

描述

Checkbox 元素代表 HTML 表单中一个图形化的复选框。注意，紧邻复选框要显示的文本不是 Checkbox 元素自身的一部分，必须在创建 Checkbox 对象的 HTML 标记 `<input>` 的外部声明它。

用事件处理程序 `onClick` 可以声明在复选框选中或取消选定时要执行的 JavaScript 代码。可以通过检测 `checked` 属性来判断复选框的状态，也可以通过设置这一属性来选定复选框或取消对它的选定。注意，设置属性 `checked` 只会改变复选框的图形化外观，并不会调用 `onClick` 事件处理程序。

声明复选框的 `name` 性质是一种很好的程序设计风格，而且如果复选框是给运行在 Web 服务器上的 CGI 脚本提交数据的表单的一部分，那么这样做是必须的。声明 `name` 性质不仅可以设置 `name` 属性，还可以在自己的 JavaScript 代码中使用名字（而不是作为表单的 `elements` 数组的元素）来引用那个复选框，这就增强了代码的模块性和可移植性。

例如，如果表单 `f` 中有一个 `name` 性质为 “`opts`” 的复选框，那么 `f.opts` 引用的就是这个 Checkbox 元素。但是 Checkbox 元素通常是在相关的组中使用的，而且组中的每

个成员都具有相同的 `name` 性质（就是这个共享的名字定义了该组的成员）。在这种情况下，JavaScript 把这个组中的所有 `Checkbox` 元素都存储在一个数组中，数组就以共享的名字来命名。例如，如果表单 `f` 中所有复选框的 `name` 性质都设置为 “`opts`”，那么 `f.opts` 就是 `Checkbox` 元素的数组，`f.opts.length` 就是这个数组中元素的个数。

可以设置 `value` 性质或复选框的 `value` 属性来声明字符串。在提交表单时如果选中了复选框，则该字符串传递给服务器。对于单一的复选框来说，默认的 `value` 值 “`on`” 就足够用了。如果使用的是多个具有相同名字的复选框，就应该给每个复选框都声明一个独立的 `value` 值，以便传递给服务器一个选中的复选框的 `value` 列表。

习惯用法

`Checkbox` 元素可以呈现给用户一个或多个选项。这种类型的元素适用于不互斥的选项，`Radio` 元素则适用于互斥的选项列表。

参阅

`Form`、`HTMLElement`、`Input`、`Radio`；DOM 参考手册部分的 `HTMLInputElement`

Checkbox.onclick

JavaScript 1.0

当复选框被选中时调用的处理程序

摘要

```
<input type='checkbox' onclick="handler">  
checkbox.onclick
```

描述

`Checkbox` 对象的 `onclick` 属性引用的是在用户点击复选框时调用的事件处理函数。要了解完整的细节，请参阅 “`HTMLElement.onclick`”。但是要注意，从 JavaScript 1.0 起，就支持处理程序 `Checkbox.onclick`，而通用化的处理程序 `HTMLElement.onclick` 却并非如此。

参阅

`HTMLElement.onclick`；第十九章；DOM 参考手册部分的 `EventListener`、`EventTarget`、`MouseEvent`

Document JavaScript 1.0;在 JavaScript 1.1、Netscape 4 和 IE4 中增强了它
代表一个 HTML 文档 继承 HTMLElement

摘要

`window.document`

`document`

属性

Document 对象继承了 HTMLElement 对象的所有属性, 此外还定义了下而列出的一些属性。Netscape 和 Internet Explorer 都定义了大量不兼容的 Document 属性, 这些属性大部分是用于 DHTML 的, 它们单独列在 Document 定义的属性之后:

`alinkColor`

一个字符串属性, 指定了 `document` 中被激活的链接的颜色。当用户在链接上按下和释放鼠标按钮的间隙中, 浏览器会显示这种颜色。HTML 标记 `<body>` 的 `alink` 性质设置该属性的初始值, 但只能在文档的 `<head>` 部分设置该属性。参阅 DOM 参考手册部分 `HTMLBodyElement` 的颜色属性。

`anchors[]`

一个 Anchor 对象数组, 每个元素代表了文档中的一个锚。锚是文档中一个命名的位置, 用作超文本链接的目标。`anchors[]` 数组具有 `anchors.length` 个元素, 从 0 到 `anchors.length - 1` 编码。不要混淆了锚和超文本链接, 后者由 `Document.links[]` 数组中的 Link 对象表示。

在 JavaScript 1.2 之前的版本中, 没有实现 Anchor 对象, `anchors[]` 数组的元素是 `null`。

`applets[]` *[JavaScript 1.1]*

一个 Applet 对象数组, 每个元素代表文档中出现的一个小程序。可以使用 Applet 对象对小程序中的所有公有变量进行读写, 也可以调用小程序的所有公有方法。如果 `<applet>` 标记具有 `name` 性质, 则可以将这个名字作为 `document` 的一个属性或者作为 `applets` 数组的下标来引用那个小程序。例如, 假定文档中的第一个小程序具有属性 `name="animator"`, 则可以采用下列方式来引用它:

```
document.applets[0]
document.animator
document.applets['animator']
```

bgColor

一个字符串属性，指定 `document` 的背景颜色，初始值由 `<body>` 标记的 `bgcolor` 性质设置。通过给 `bgColor` 赋值来改变背景颜色。和其他颜色属性不同的是，可以在任意时刻对 `bgColor` 进行设置。参阅 DOM 参考手册部分 `HTMLBodyElement` 的颜色属性。

cookie

一个字符串，它是与文档关联在一起的 `cookie` 的值。参阅“`Document.cookie`”参考页。

domain

一个字符串属性，它指定了文档所属的 Internet 域。用于安全性方面。JavaScript 1.1 及其后的版本支持该属性。参阅“`Document.domain`”参考页。

`embeds[]` [*JavaScript 1.1*]

一个对象数组，每个元素表示一个由 `<embed>` 标记嵌入文档的数据。`embeds[]` 数组中的对象并非直接引用嵌入的数据，而是引用显示这些数据对象。可以用 `embeds[]` 数组中的对象与嵌入的数据进行交互。但是交互的方式却是由嵌入数据的类型和显示它使用的插件或 ActiveX 控件指定的。要知道一个插件或者 ActiveX 控件是否能在 JavaScript 中使用，可以查询开发人员手册，如果答案是肯定的，那么还要查询 JavaScript 支持哪种 API。

`Document.plugins[]` 是 `Document.embeds[]` 的同义词。不要把它和 `Navigator.plugins[]` 相混淆。

fgColor

一个字符串属性，指定了 `document` 文本的默认颜色。该属性的初始值来自 `<body>` 标记的 `text` 性质。可以在文档的 `<head>` 标记中设置它的值。参阅 DOM 参考手册中 `HTMLBodyElement` 的颜色属性。

`forms[]`

一个数组，元素是 `Form` 对象，每个元素代表出现在 `document` 中的一个表单。`forms[]` 数组具有 `forms.length` 个元素，元素从 0 到 `forms.length - 1` 编码。

`images[]` [*JavaScript 1.1*]

一个数组，元素是 `Image` 对象，每个元素代表用 HTML 标记 `` 嵌入文档的一个图像。如果在 `` 标记中指定了图像的 `name` 性质，那么在 `Document` 对

象的属性中也存在对那个图像的引用。该属性的名字与图像名相同。所以, 如果一个图像具有 `name="toggle"` 性质, 就可以用 `document.toggle` 引用它。

`lastModified`

一个只读字符串, 声明了最后一次修改文档的日期(与Web服务器报告的一样)。参阅“`Document.lastModified`”参考页。

`linkColor`

一个字符串属性, 指定了文档中未被访问过的链接的颜色。该属性的值由 `<body>` 标记的 `link` 性质设置, 还可以由文档 `<head>` 部分中的脚本设置。参阅 DOM 参考手册中 `HTMLBodyElement` 的颜色属性。

`links`

一个数组, 元素是 `Link` 对象, 每个元素代表了文档中出现的的一个超文本链接。`links[]` 数组具有 `links.length` 个元素, 从 0 到 `links.length - 1` 编码。

`location` [反对使用]

一个 `Location` 对象, 含有当前文档的完整 URL, 是 `Window.location` 属性的同义词。在 JavaScript 1.0 中, 该属性代替了只读的字符串对象, 该对象的用途和 `Document.URL` 属性一样。

`plugins[]` [JavaScript 1.1]

数组 `embeds[]` 的同义词, 引用一个对象数组, 其中的对象表示显示文档中的嵌入数据时使用的插件或 ActiveX 控件。采用 `embeds` 属性是访问该数组的推荐方式, 因为它避免了与 `Navigator.plugins[]` 数组的混淆。

`referrer`

一个只读字符串, 含有链接到当前文档的文档的 URL (如果存在)。例如, 如果用户从文档 A 链接到了文档 B, 那么文档 B 的 `Document.referrer` 属性存放的就是文档 A 的 URL。但是, 如果用户直接键入文档 B 的 URL, 而且没有使用其他链接, 那么文档 B 的 `Document.referrer` 属性值就是一个空字符串。

`title`

一个只读字符串, 指定了当前文档的标题。该标题就是出现在文档 `<head>` 部分的标记 `<title>` 和 `</title>` 之间的文本。

`URL`

一个只读字符串, 声明了文档的 URL。参阅“`Document.URL`”参考页。

`vlinkColor`

一个字符串属性，指定了 `document` 中已经访问过的链接的颜色。该属性的值由 `<body>` 标记的 `vlink` 性质设置，也可以从文档的 `<head>` 部分的脚本设置它。参阅 DOM 参考手册中 `HTMLBodyElement` 的颜色属性。

Netscape 属性

`height` [*Netscape 4*]

文档的高度，以像素计。

`layers[]` [*仅限于 Netscape 4*]

一个数组，元素是文档中包含的表示层的所有 `Layer` 对象。每个 `Layer` 对象包含自己的子文档，通过它的 `document` 属性可以访问这些文档。该属性只在 `Netscape 4` 中可用，`Netscape 6` 已经不再采用它。

`width` [*Netscape 4*]

文档的宽度，以像素计。

Internet Explorer 属性

`activeElement` [*IE 4*]

一个只读属性，引用文档中当前活动的输入元素（即具有输入焦点的元素）。

`all[]` [*IE 4*]

一个数组，元素是文档中包含的所有元素。参阅“`Document.all`”参考页。

`charset` [*IE 4*]

文档采用的字符集。

`children[]` [*IE 4*]

一个数组，元素是文档的所有直接子元素，以它们在源代码中的顺序存放。注意，它不同于 `all[]` 数组，后者含有所有文档元素，不论它们在包容层次中使用什么位置。

`defaultCharset` [*IE 4*]

文档采用的默认字符集。

`expando` [*IE 4*]

如果将 `expando` 设置成 `false`，可以阻止客户端对象的扩展。也就是说，如果程序试图设置客户端对象的一个不存在的属性，就会引发运行时错误。把

`expando` 设置为 `false`，还有助于捕捉属性拼写错误而引起的 bug，否则这种 bug 很难发现。这一属性对于那些已经习惯了不区分大小写的语言，继而学习 JavaScript 的程序设计者们来说非常有帮助。虽然 `expando` 只在 IE 4 中有效，但是在 Netscape 中设置它也是安全的（即使没有效果）。

`ParentWindow`[IE 4]

包含文档的窗口。

`readyState`

文档的装载状态。它有四个可用的值：

`uninitialized`

还没有开始装载文档。

`loading`

正在装载文档。

`interactive`

装载的文档已经足够与用户进行交互。

`Complete`

文档已经装载完毕。

方法

`Document` 对象继承了 `HTMLElement` 对象的所有方法，此外还定义了一些方法。Netscape 和 IE 都定义了大量不兼容的 `Document` 方法，它们主要用于 DHTML，在 `Document` 定义的方法之后单独列出了这些方法。

`clear()`

擦去文档内容。JavaScript 1.1 反对使用该方法。

`close()`

关闭由方法 `open()` 打开的文档流。

`open()`

打开一个可供写入文档内容的流。

`write()`

将一个或多个指定的字符串插入当前正在解析的文档中或插入由方法 `open()` 打开的文档流中。

`writeln()`

与方法 `write()` 相似，只是在输出的字符串中插入一个换行符。

Netscape 方法

`captureEvents()`

请求指定类型的事件。

`getSelection()`

返回当前选中的文档文本。

`releaseEvents()`

停止捕捉指定类型的事件。

`routeEvent()`

根据捕捉到的事件找到下一个与之相关的元素。参看方法 `Window.routeEvent()`。

Internet Explorer 方法

`elementFromPoint()`

返回位于指定地点（X 坐标和 Y 坐标）的元素。

事件处理程序

`<body>` 标记具有 `onload` 和 `onunload` 性质。但是，原则上事件处理程序 `onload` 和 `onunload` 属于 `Window` 对象，而不属于 `Document` 对象。参看 `Window.onload` 和 `Window.onunload`。

HTML 语法

`Document` 对象中有很多相应于 HTML 标记 `<body>` 的性质的属性。而且 HTML 文档的内容都出现在 `<body>` 和 `</body>` 之间：

```
<body
  [ background = "imageURL"      // 文档的背景图像
  , bgcolor = "color"            // 文档的背景颜色
  , text = "color"               // 文档文本的前景颜色
  , link = "color"              // 未被访问过的链接的颜色
  , alink = "color"             // 激活了的链接的颜色
  , vlink = "color"             // 访问过的链接的颜色
  , onload = "handler"          // 装载文档时运行的 JavaScript 代码
  , onunload = "handler"        // 卸载文档时运行的 JavaScript 代码
>
// HTML 文档的内容
</body>
```

描述

Document 对象代表的是一个浏览器窗口或框架（或 Netscape 4 中的层）中显示的 HTML 文档。该对象的属性提供了文档各个方面的细节，从文本颜色、背景颜色、锚元素采用的颜色，到最近一次修改文档的日期，无所不有。Document 对象还具有大量的数组，用于描述文档的内容。数组 `links[]` 的元素是 Link 对象，代表文档中的一个超文本链接。同样，数组 `applets[]` 的每个元素都代表了一个嵌入文档的 Java 小程序，数组 `forms[]` 的元素是 Form 对象，代表出现在文档中的 HTML 表单。

Document 对象的 `write()` 方法极具代表性。可以在装载文档时运行的脚本中调用 `document.write()` 来给文档插入动态生成的 HTML 文本。

第十四章对 Document 对象及其引用的多个 JavaScript 对象进行了概述。第十七章对 DOM 标准进行了概述。

参阅

Form、Window 对象的 `document` 属性；第十四章；DOM 参考手册中的 Document、HTMLDocument、HTMLBodyElement

Document.all[] Internet Explorer 4

文档中的所有 HTML 元素

摘要

```
document.all[i]
document.all[name]
document.all.tags(tagname)
```

描述

`all[]` 是一个多功能的数组，它包含文档中的所有 HTML 元素。`all[]` 中的元素是按照它们在文档中出现的顺序来存放的，如果知道某个元素在数组中的数字位置，就可以直接将它从数组中提取出来。但通常用 `name` 性质或 `id` 性质从 `all[]` 数组中获取元素。如果有多个元素具有指定的名字，则使用该名字作为 `all[]` 的下标来返回共享该名字的元素数组。

传递给 `all.tags()` 方法的是标记名，它返回具有指定类型的 HTML 元素的数组。

参阅

HTMLElement

Document.captureEvents()

参阅 Window.captureEvents()

Document.clear()

JavaScript 1.0; 反对使用

清除一个文档

摘要

`document.clear()`

描述

Document 对象的方法 `clear()` 是被反对使用的，所以应该避免使用它。要清除一个文档，只需要用 `Document.open()` 打开一个新文档即可。

参阅

`Document.close()`、`Document.open()`、`Document.write()`

Document.close()

JavaScript 1.0

关闭一个输出流

摘要

`document.close()`

描述

该方法将显示出所有已经写入了 `document` 但是还没有显示出来的内容，然后关闭 `document` 的输出流。在用 `Document.write()` 生成了一个完整的 HTML 页面时，如果到达了文档的结尾，就应该调用 `Document.close()`。

在调用了 `document.close()` 后，如果还有要写入 `document` 的输出内容（如用 `document.write()` 写入的），那么文档就会被隐式地清除掉，然后再被重新打开，调用 `close()` 之前写入的内容都将被抹掉。

参阅

`Document.open()`、`Document.write()`

Document.cookieJavaScript 1.0

文档的 cookie

摘要`document.cookie`**描述**

`cookie`是一个字符串属性,使用它可以对应用于当前文档的`cookie`或`cookies`文件进行读操作、创建操作、修改操作以及删除操作。所谓`cookie`,就是浏览器存储的少量具有名字的数据。它主要是给浏览器提供一块“内存”,以便它们能够在一个页面中使用另一个页面的数据,或者通过网络浏览会话恢复用户的优先级。为了使服务器端的CGI脚本能够读写`cookie`的值,`cookie`数据会在适当的时候自动地在浏览器和服务器之间进行传输。客户端的JavaScript代码也可以使用这一属性来读写`cookie`数据。

`Document.cookie`属性的行为与常规的读/写属性不同。虽然既可以对`Document.cookie`进行读操作,也可以进行写操作,但是从这个属性读到的值通常与写入的值不一样。要了解这个极其复杂的属性的完整细节,请参阅第十六章。

习惯用法

Cookie主要用于存储少量数据不经常的。它们并不是通用的通信或程序设计机制,所以对它们的使用也是有限的。注意,Web浏览器保留的每个服务器的`cookie`值不能超过20个(是对整个服务器而言,而不仅是对服务器上的站点而言),而且保留的`cookie`数据的名字/值对的长度也不能超过4KB。

参阅

第十六章

Document.domainJavaScript 1.1

文档的安全域

摘要`document.domain`**描述**

出于安全性的原因,一个窗口中运行的没有签名的脚本不能读其他窗口的属性,除非

它要读的窗口和它来自同一个Web服务器。这给那些使用多个服务器的大网站带来了麻烦。例如，位于主机 *www.oreilly.com* 上的脚本可能想要共享位于主机 *search.oreilly.com* 上的某个脚本的属性。

属性 *domain* 解决了这一问题。开始，这个属性存放的是装载进来的文档所属的服务器的主机名。可以设置这个属性，但限制非常严格，即只能将它设置为自己的域名后缀。例如，从 *search.oreilly.com* 装载进来的脚本，可以将自己的 *domain* 属性设置为 “*oreilly.com*”。如果来自 *www.oreilly.com* 的脚本在另一个窗口中运行，而且也将自己的 *domain* 属性设置成了 “*oreilly.com*”，那么这两个脚本就可以共享属性，即使它们并非出自同一个服务器。

但要注意，来自 *search.oreilly.com* 的脚本不能将自己的 *domain* 属性设置为 “*search.oreilly.com*”。而且来自 “*snoop.spam.com*” 的脚本也不能通过把它的 *domain* 属性设置成 “*oreilly.com*” 来判断你所使用的检索关键字。

参阅

第二十一章

Document.elementFromPoint()

Internet Explorer 4

判断哪个 HTML 元素处于指定的位置

摘要

`document.elementFromPoint(x, y)`

参数

x X 坐标

y Y 坐标

返回值

出现在 *document* 中指定位置 (*x*, *y*) 的 HTML 元素。

Document.getSelection()

Netscape 4

返回选中的文本

摘要

`document.getSelection()`

返回值

文档中当前选中的文本（如果存在）。返回的文本中已经删除了 HTML 的格式标记。

Document.handleEvent()

参阅 Window.handleEvent()

Document.lastModified

JavaScript 1.0

文档的最后修改日期

摘要

`document.lastModified`

描述

属性 `lastModified` 是一个只读字符串，存放的是最后修改 `document` 的日期和时间。该数据是从服务器发送的 HTTP 头数据中派生出来的。Web 服务器通常是通过检查自身文件的修改日期得到最后修改日期。

Web 服务器不必给它们使用的文档提供最后修改日期。当 Web 服务器没有提供文档的最后修改日期时，JavaScript 就假定它为 0，换算过来也就是 GMT 时间 1970 年 1 月 1 日午夜。下面的例子说明了如何检测这种情况。

例子

让读者知道信息在 Web 上保存了多久是个好想法。可以把接下来的那段脚本添加到每个 HTML 文件的结尾，这样就在你的文档中放置了一个自动的时间戳，你就不必在每次修改文件时手动地更新修改时间了。注意，这段脚本在显示服务器提供的数据之前会先检测提供的数据的有效性：

```
<script>
if (Date.parse(document.lastModified) != 0)
    document.write('<p><hr><small><j>Last modified:
                  + document.lastModified
                  - '</j></small>' );
</script>
```

参阅

`Document.location`、`Document.referrer`、`Document.title`

Document.links[]JavaScript 1.0

文档中的 Link 对象

摘要`document.links``document.links.length`**描述**

属性 `links` 是一个数组，它的元素是 Link 对象，每个元素代表 `document` 中出现的一个超文本链接。该数组具有 `links.length` 个元素，这些元素的编码是从 0 到 `links.length - 1`。

参阅

Link

Document.open()JavaScript 1.0

打开一个新文档

摘要`document.open()``document.open(mimetype)`**参数**`mimetype`

一个可选的字符串参数，指定了要写入 `document` 并显示出来的数据的类型。该参数的值应该是浏览器可以识别的标准 MIME 类型之一（在 Netscape 中即“text/html”、“text/plain”、“text/gif”、“image/jpeg”和“image/x-bitmap”），或者是安装到浏览器上的某种插件可以处理的 MIME 类型。如果省略了这个参数，它的默认值是“text/html”。IE 3 将忽略这个参数，它总是假定文档的类型是“text/html”。W3C DOM 标准中的 `Document.open()` 方法不支持该参数。参阅 DOM 参考手册中的“HTMLDocument.open()”条目。

描述

方法 `document.open()` 会给 `document` 打开一个流，以便接下来调用的 `document.`

`write()` 能够将数据添加到文档中。参数 `mimetype` 是可选的，它指定了要写入文档的数据类型，并且告诉浏览器如何解释那些数据。

如果在调用方法 `open()` 时已经显示了一个文档，那么它通过调用方法 `close()` 或初次调用方法 `write()` 或 `writeln()`，自动地将文档内容清除掉。

习惯用法

通常可以不用参数调用 `Document.open()` 来打开一个 HTML 文档。有时，“text/plain” 类型的文档也比较有用，例如，一个显示调试信息的弹出式窗口，它就需要这种类型的文档。

参阅

`Document.close()`、`Document.write()`；DOM 参考手册中的 `HTMLDocument.open()`

Document.releaseEvents()

参阅 `Window.releaseEvents()`

Document.routeEvent()

参阅 `Window.routeEvent()`

Document.URL

JavaScript 1.1

当前文档的 URL

摘要

`document.URL`

描述

属性 `URL` 是一个只读字符串，存放了当前文档的完整 URL。`document.URL` 和包含该文档的 `Window` 的属性 `window.location.href` 是相等的。但这两个属性不一定相等，因为 URL 的重定向有可能修改属性 `Document.URL`，但 `Window.location` 存放的是被请求的文档的 URL，而 `Document.URL` 存放的却是真正获得的文档的 URL。

习惯用法

有些网页制作者喜欢将文档的 URL 存放在文档中的某个地方，这样即使文档被剪切

了，然后粘贴到了另一个文件中，或者是被打印了，对它的在线引用仍然会存在。如果将下面的脚本添加到一个文档中，它就会自动地将文档的 URL 添加进来：

```
<script>
document.write( "<p><br><small><f>URL: " + document.URL
               + "</f></small>" );
</script>
```

参阅

Document.lastModified、Document.location、Document.referrer、Document.title;
Window.location

Document.write()

JavaScript 1.0

给文档添加数据

摘要

```
document.write(value, ...)
```

参数

value

要添加到 *document* 的任意一个 JavaScript 值。如果这个值不是字符串，那么在添加到文档之前，会被转化成 一个字符串。

...

任意多个（0 个或多个）要（依次）写入文档的值。

描述

方法 *document.write()* 将把它的所有参数按照它们出现的顺序写入文档 *document*。不是字符串的参数在附加到文档结尾之前都会被转换成字符串。

使用 *Document.write()* 的方式通常有两种。一种是在 `<script>` 标记中或者在解析文档时要执行的函数中，由当前文档调用它。在这种情况下，方法 *write()* 会把它的 HTML 数据写入到调用该方法的代码所在的地点，就像那些数据是直接出现在文件中的一样。

Document.wrrite() 的另一种用法，通常是动态地生成另一个窗口（而不是当前窗口）中的内容。在这种情况下，解析进程不会处理目标文档，所以要写入的数据就不能像刚才介绍的那样出现在文档中。要使用 *write()* 将文本写入文档，首先要将那个

文档打开。可以通过明确地调用方法 `Document.open()` 打开一个文档。但是，大多数情况下不必这么做，因为当调用 `write()` 的文档处于关闭状态时，它就会隐式地将文档打开。打开文档时，文档中原有的内容都会被清除掉，而用一个空白文档来代替它。

如果一个文档已经打开，那么使用 `Document.write()` 就可以把任意多个数据添加到该文档的末尾。当使用这种方法生成了一个新文档时，还应该调用方法 `Document.close()` 关闭这个文档。注意，尽管对方法 `open()` 的调用是可选的，但是对方法 `close()` 的调用却是必须的。

调用方法 `Document.write()` 的结果也许不能立刻在目标浏览器窗口或框架中显示出来。这是因为浏览器可能会将数据缓存起来，以便形成大的数据块输出。调用 `Document.close()` 是唯一一种能明确地强迫“冲洗”所有缓存的数据，并在浏览器窗口中显示这些数据的方法。

参阅

`Document.close()`、`Document.open()`、`Document.writeln()`；第十四章

`Document.writeln()`

JavaScript 1.0

给文档添加数据和换行符

摘要

```
document.writeln(value, ...)
```

参数

value

要添加到 *document* 的任意一个 JavaScript 值。如果这个值不是字符串，那么在添加到文档之前，它会被转化成一个字符串。

...

任意多个（0 个或多个）要（依次）写入文档的值。

描述

`Document.writeln()` 与 `Document.write()` 相似，只不过在写入了所有的 *document* 参数后，`writeln()` 还会再加上一个换行符。要了解这个方法的有关信息，请参阅 `Document.write()` 参考页。

换行符在HTML文档中通常不显示出来,因此,一般说来,只有当要写入的文本出现在<pre>环境中,或者是要写到MIME类型为“text/plain”的文档中时,使用Document.writeln()才比较有用。

参阅

Document.close()、Document.open()、Document.write()

Element

参阅 Input

Event JavaScript 1.2; Netscape 4 和 Internet Explorer 4 支持的不兼容版本事件的细节

摘要

```
function handler(event) { ... } // Netscape 4 中的事件处理程序参数
window.event                    // IE 4 中的 Window 属性
```

常量

在 Netscape 4 中, Event 对象为支持的事件类型定义了大量位掩码常量。这些静态属性用于构成传递给方法 captureEvents() 和 releaseEvents() 的位掩码。可用的常量如下:

Event.ABORT	Event.BLUR	Event.CHANGE	Event.CLICK
Event.DBLCLICK	Event.DRAGDROP	Event.ERROR	Event.FOCUS
Event.KEYDOWN	Event.KEYPRESS	Event.KEYUP	Event.LOAD
Event.MOUSEDOWN	Event.MOUSEMOVE	Event.MOUSEOUT	Event.MOUSEOVER
Event.MOUSEUP	Event.MOVE	Event.RESET	Event.RESIZE
Event.SELECT	Event.SUBMIT	Event.UNLOAD	

Netscape 4 属性

height

只能由类型为“resize”的事件设置。声明了调整过的窗口或框架的新高度。

layerX, layerY

事件相对于包容图层的 X 坐标和 Y 坐标。

modifiers

声明了在事件发生时按下并保持住的组合键。这个数字值是一个位掩码,由常量

Event.ALT_MASK、Event.CONTROL_MASK、Event.META_MASK 和 Event.SHIFT_MASK 构成。由于存在 bug，所以 Netscape 6 和 6.1 没有定义该属性。

pageX, pageY

事件发生的位置相对于浏览器页面的 X 坐标和 Y 坐标。注意，这个坐标相对于顶层页面，并非相对于任何封闭层。

screenX, screenY

事件发生的位置相对于屏幕的 X 坐标和 Y 坐标。注意，和大多数 Event 属性不同的是，Internet Explorer 4 和 Netscape 4 都支持它们，并且它们在这两种浏览器中具有相同的含义。

target

对生成事件的 Window 对象、Document 对象、Layer 对象或 HTML 元素的引用。

type

一个字符串，声明了事件的类型。它的值是事件处理程序的名字去掉前缀“on”。因此，在调用事件处理程序 onclick() 时，Event 对象的 type 属性值就是“click”。

which

对于键盘事件和鼠标事件来说，which 声明的是按下或放开的键或鼠标按钮。对于键盘事件来说，这个属性存放的是按下的键的字符编码。对于鼠标事件来说，这个属性的值为 1、2 或 3，它们分别代表鼠标的左按钮、中间按钮和右按钮。

width

只在类型为“resize”的事件中设置，声明调整大小之后的窗口或框架的新宽度。

x, y

事件发生的位置的 X 坐标和 Y 坐标。在 Netscape 4 中，这个属性等价于属性 layerX 和 layerY，它们声明的都是相对于包容层（如果存在）的位置。它们在 Internet Explorer 中的含义有所不同（参阅下一节）。

Internet Explorer 4 属性

altKey

一个布尔值，指定事件发生时，Alt 键是否被按下并保持住了。

`button`

对于鼠标事件, `button` 属性声明了被按下的鼠标按钮或按钮。这个只读的整数是位掩码, 如果鼠标左按钮被按下, 设置第 1 位; 如果鼠标右按钮被按下, 设置第 2 位; 如果 (三按钮的) 鼠标的中间按钮被按下, 设置第 4 位。

`cancelBubble`

如果事件处理程序想阻止事件传播到包容对象, 必须把该属性设为 `true`。

`clientX, clientY`

声明了事件发生的位置相对于浏览器页面的 X 坐标和 Y 坐标。

`ctrlKey`

一个布尔值, 指定事件发生时, **Ctrl** 键是否被按下并保持住了。

`fromElement`

对于 `mouseover` 和 `mouseout` 事件, 该属性引用移出鼠标的元素。

`keyCode`

对于键盘事件, 该属性声明了被敲击的键生成的 Unicode 字符码。

`offsetX, offsetY`

发生事件的地点在事件源元素的坐标系统中的 X 坐标和 Y 坐标 (参见 `SrcElement`)。

`reason`

对于 `datasetcomplete` 事件, `reason` 存放数据传输事件的状态码。0 说明传输成功, 1 说明传输失败, 2 说明数据传输过程中发生了错误。

`returnValue`

如果设置了该属性, 它的值比事件处理程序真正的返回值优先级高。把这个属性设置为 `false`, 可以取消发生事件的源元素的默认动作。

`screenX, screenY`

事件发生的地点相对于屏幕的 X 坐标和 Y 坐标。注意, 和大部分 Event 属性不同的是, Internet Explorer 4 和 Netscape 4 都支持它们, 并且它们在这两种浏览器中具有相同的含义。

`shiftKey`

一个布尔值, 声明了事件发生时, **Shift** 键是否被按下并保持住了。

srcElement

对生成事件的 Window 对象、Document 对象或 HTML 元素的引用。

srcFilter

对于 filterchange 事件，该属性声明改变的过滤器。

toElement

对于 mouseover 和 mouseout 事件，该属性引用移入鼠标的元素。

type

一个字符串，声明了事件的类型。它的值是事件处理程序的名字去掉前缀“on”。因此，在调用事件处理程序 onclick() 时，Event 对象的 type 属性值就是“click”。

x, y

事件发生的位置的 X 坐标和 Y 坐标。在 Internet Explorer 4 中，它们声明了相对于用 CSS 动态定位的最内层包容元素的 X 坐标和 Y 坐标。Netscape 4 对这两个属性的解释有所不同（参阅上一节）。

描述

Event 对象提供了发生的事件的详细情况。但是，这些细节没有被标准化，Netscape 4 和 IE 4 对这些细节的定义几乎完全不兼容。除了定义的属性不同之外，Netscape 4 和 IE 4 提供的访问 Event 对象的方式也不相同。在 Netscape 中，Event 对象作为参数传递给每个事件处理程序。那些由 HTML 性质定义的事件处理程序的事件参数名就是 event。在 IE 中，最近的事件的 Event 对象存储在 Window 对象的 event 属性中。

除了 Netscape 4 和 IE 的 Event 对象的不兼容性，W3C DOM 还定义了自己的 Event 对象，它与前两者都不兼容。参阅 DOM 参考手册部分的“Event”、“UIEvent”和“MouseEvent”。

参阅

第十九章；DOM 参考手册部分的 Event、UIEvent、MouseEvent

FileUpload

JavaScript 1.0

表单输入元素中的文件上传域

继承 Input, HTMLElement

摘要

form.name

```
form.elements[i]
```

属性

FileUpload 对象继承了 Input 对象和 HTMLElement 对象的属性，此外它还定义或覆盖了以下属性：

value*[JavaScript 1.1]*

一个只读字符串，声明了用户在 FileUpload 对象中输入的文件名。用户可以直接输入这个文件名，也可以从与 FileUpload 对象关联在一起的目录浏览器中选择文件名。

为了防止恶意程序从客户端上载文件，JavaScript 代码是不能设置这一属性的。出于同样的原因，标记 <input> 不能指定这个属性的默认值。

方法

FileUpload 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

FileUpload 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序，此外它还定义或覆盖了下列事件处理程序：

onchange

当用户改变了 FileUpload 元素的值，并且将键盘焦点移到了别处时，调用该处理程序。但是并非每次 FileUpload 元素中发生了敲击键盘事件时都会调用这个事件处理程序，只有当用户完成了编辑之后才会调用它。

HTML 语法

FileUpload 元素是由标准的 HTML 标记 <input> 创建的：

```
<form enctype="multipart/form-data"
      method="post">           // 必需的性质
...
<input
  type = 'file'                // 声明这是一个 FileUpload 元素
  [ name = 'name' ]           // 此后可以用来引用这个元素的名字
                              // 指定了 name 属性
  [size= 'integer' ]          // 该元素的宽度是多少个字符
  [ maxlength = "integer" ]    // 允许输入的最多字符数
  [ onblur= 'handler' ]       // 事件处理程序 onblur()
  [ onchange= 'handler' ]     // 事件处理程序 onchange()
  [ onfocus= "handler" ]     // 事件处理程序 onfocus()
```

>
...

描述

FileUpload 元素代表表单中的文件上传输入元素。它在许多方面都与 Text 元素相似。在屏幕上，它的外形就像一个文本输入框，只不过多了一个用来打开目录浏览器的 **Browse** 按钮。在 FileUpload 元素中输入一个文件名（既可以直接输入，也可以通过浏览改变），会使 Netscape 在提交表单时提交那个文件的内容。要完成上述操作，必须使用“multipart/form-data”的编码格式以及 POST 方法。

FileUpload 元素没有 defaultValue 属性，也不能识别用来指定输入字段的初始值的 HTML 性质 value。而且 FileUpload 元素的 value 属性还是只读的。只有用户可以输入文件名，JavaScript 没有任何方法可以在 FileUpload 字段中输入文本。这就防止了恶意 JavaScript 程序从用户机器中上载任意的文件（如口令文件）。

参阅

Form、HTMLElement、Input、Text；DOM 参考手册中的 HTMLInputElement

FileUpload.onChange

JavaScript 1.0

当改变输入值时调用的处理程序

摘要

```
<input type='file' onchange="handler" ...>  
fileupload.onChange
```

描述

FileUpload 元素的 onChange 属性指定了用户改变输入字段中的值（既可以通过直接输入改变，也可以通过浏览改变），然后将输入焦点移到别处时要调用的事件处理函数。由于这个事件处理程序用于处理输入值的完整改变，所以它不是基于每次键盘敲击事件而调用的。

这个属性的初始值是一个由定义该 FileUpload 对象的 HTML 标记的性质 onChange 指定的函数，这个函数含有 JavaScript 语句，每个语句之间用分号隔开。当一个事件处理函数由一个 HTML 性质定义时，它就在 element 作用域中执行，而不是在包容窗口的作用域中执行。

在 Netscape 4 的事件模型中, Event 对象作为参数传递给事件处理函数 `onchange`。在 IE 4 事件模型中则不然, 可用的 Event 对象是作为包含 `element` 的 Window 对象的 `event` 属性使用的。

参阅

`Input.onchange`; 第十九章; DOM 参考手册中的 Event、EventListener、EventTarget

Form

JavaScript 1.0

HTML 输入表单

继承 HTMLElement

摘要

`document.form_name``document.forms[form_number]`

属性

Form 对象继承了 HTMLElement 对象的属性, 除此之外, 它还定义或覆盖了如下的属性:

`action`

一个可读可写的字符串 (在 IE 3 中是只读的), 指定不要提交的表单的 URL, 该属性的初始值由 HTML 的 `<form>` 标记的 `action` 性质设置。通常, 这个 URL 将地址声明为 CGI 脚本, 不过它还可以是 `mailto:地址` 或 `news:地址`。

`elements[]`

一个数组, 元素是表单中的输入元素, 它们可能是 Button 对象、Checkbox 对象、Hidden 对象、Password 对象、Radio 对象、Reset 对象、Select 对象、Submit 对象、Text 对象或 Textarea 对象。参阅 “Form.elements” 参考页。

`encoding`

可读可写的 (在 IE 3 中是只读的) 字符串, 指定了提交表单时传输的数据的编码形式。该属性的初始值由 `<form>` 标记的 `enctype` 性质设置。默认值是 “application/x-www-form-urlencoded”, 它几乎适用于所有用途。其他值有时也是必需的, 例如, “text/plain” 在通过 `mailto:URL` 提交表单时很方便。进一步的信息请参阅《CGI Programming on the World Wide Web》, 由 Shishir Gundavaram 编著, O'Reilly 出版。

length

表单中的元素个数。等价于 `elements.length`。

method

一个可读可写（在 IE 3 中只读）的字符串，它指定了提交表单数据所采用的方法。该属性的初始值由 HTML 标记 `<form>` 的 `method` 性质设置。它的两个合法值是 `get` 和 `post`。

`get` 是默认的提交方法。它通常用于没有其他作用的表单提交（如数据库查询）。使用这种方法可以将编码的表单数据添加到由属性 `Form.action` 指定的 URL 中。方法 `post` 适用于那些具有其他作用的表单提交（如给数据库添加条目）。使用这种方法，编码的表单数据就会在 HTTP 的查询主体中发送出去。

name

声明表单的名字，该属性可读可写，初始值就是 `<form>` 标记的 `name` 性质的值。

target

一个可读可写的字符串，指定了要显示提交表单的结果的窗口或框架的名字。初始值由性质 `target` 设置。这两个属性还支持特殊的名字 `"_top"`、`"_parent"`、`"_self"` 和 `"_blank"`。参阅“`Form.target`”参考页。

方法

`Form` 对象继承了 `HTMLElement` 对象的方法。除此之外，它还定义或覆盖了如下方法：

`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。

事件处理程序

`Form` 对象继承了 `HTMLElement` 的事件处理程序。除此之外，它还定义或覆盖了如下事件处理程序：

`onreset`

在重置表单元素之前调用。在 HTML 中，它由性质 `onreset` 设置。

onsubmit

在提交表单之前调用。在 HTML 中，它由性质 onsubmit 设置。该事件处理程序可以在提交表单之前对表单的各个条目进行验证。

HTML 语法

Form 对象是由标准的 HTML 标记 <form> 创建的。表单所含有的各种输入元素则是由 <form> 和 </form> 之间的 <input> 标记、<select> 标记和 <textarea> 标记创建的。

```
<form
  [ name = 'form_name' ]           // 在 JavaScript 中给表单命名
  [ target = 'window_name' ]       // 要响应的窗口的名字
  [ action = 'url' ]               // 表单提交的目的地 URL
  [ method = ( 'get' | 'post' ) ]   // 提交表单的方法
  [ enctype = 'encoding' ]         // 表单数据的编码方法
  [ onreset = 'handler' ]          // 当重置表单时调用的事件处理程序
  [ onsubmit = 'handler' ]         // 当提交表单时调用的事件处理程序
>
// 这里是表单文本和输入元素
</form>
```

描述

Form 对象代表文档中的一个 HTML 标记 <form>。文档中的每个表单都表示为数组 `Document.forms[]` 的一个元素。此外，已命名的表单还被表示为文档的一个属性 `form_name`，属性名 `form_name` 由标记 <form> 的性质 `name` 指定。

表单中的元素（如按钮、输入框、复选框，等等）都收集在数组 `Form.elements[]` 中。已命名的元素和已命名的表单一样，可以直接使用它的名字来引用，它的元素名就作为它在 Form 对象中的属性名。因此，要引用表单 `questionnaire` 中名为 `phone` 的 Text 对象元素，可以使用如下的 JavaScript 表达式：

```
document.questionnaire.phone
```

参阅

Button、Checkbox、FileUpload、Hidden、Input、Password、Radio、Reset、Select、Submit、Text、Textarea；第十五章；DOM 参考手册中的 HTMLFormElement

Form.elements[]

JavaScript 1.0

表单中的输入元素

摘要

`form.elements[i]`

`form.elements.length`

描述

在表单 *form* 中, `form.elements[]` 是一个数组, 其元素是表单中的输入对象。这个数组具有 `elements.length` 个元素。这些元素可以是表单输入元素 (`Button`、`Checkbox`、`Hidden`、`Input`、`Password`、`Radio`、`Reset`、`Select`、`Submit`、`Text` 和 `Textarea`) 中的任意一种, 它们在数组中出现的顺序和它们在表单的HTML源代码中出现的顺序相同。

习惯用法

如果数组 `form.elements[]` 中的某个元素具有名字, 这个名字是由HTML标记 `<input>` 的性质 `name="name"` 设置的, 那么这个元素的名字就成了表单 *form* 的一个属性, 使用这个属性可以引用该元素。因此, 使用名字, 而不是数字来引用输入对象是完全可能的:

`form.name`

一般说来, 使用名字引用元素更加简便一些, 因此, 设置所有表单元素的 `name` 性质是个好方法。

参阅

`Button`、`Checkbox`、`Form`、`Hidden`、`Input`、`Password`、`Radio`、`Reset`、`Select`、`Submit`、`Text`、`Textarea`

Form.onreset

JavaScript 1.1

在重置表单时调用的处理程序

摘要

```
<form ... onreset="handler" ... >
```

`form.onreset`

描述

`Form` 对象的 `onreset` 属性指定了一个事件处理函数。当用户点击了表单中的 **Reset** 按钮或者调用了方法 `Form.reset()` 时, 就会调用这个事件处理函数。

该属性的初始值是一个由HTML标记 `<form>` 的性质 `onreset` 设置的函数, 它含有用

分号分隔的 JavaScript 语句。当事件处理函数由一个 HTML 性质定义时，它就在定义它的 `element` 作用域中执行，而不是在包容窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给事件处理函数 `onreset`。而在 IE 4 事件模型中则没有参数传递给 `onreset`，可用的 Event 对象是作为包容 `element` 的 Window 对象的 `event` 属性被访问的。

如果 `onreset` 返回 `false`，表单的元素就不会被重置。

例子

可以使用下面的事件处理程序要求用户确认他们真的想重置表单：

```
<form ...  
  onreset="return confirm('Really erase all entered data?')"  
>
```

参阅

`Form.onsubmit`、`Form.reset()`；第十九章；DOM 参考手册部分的 `Event`、`EventListener`、`EventTarget`

Form.onsubmit

JavaScript 1.0

在提交表单时调用的事件处理程序

摘要

```
<form ... onsubmit='handler' ... >  
  
form.onsubmit
```

描述

Form 对象的属性 `onsubmit` 指定了一个事件处理函数。当用户点击了按钮 **Submit** 来提交表单时，就会调用这个处理函数。注意，在调用方法 `Form.submit()` 时，该事件处理函数并不会被调用。

该属性的初始值是一个由 HTML 标记 `<form>` 的 `onsubmit` 性质设置的函数，该函数含有用分号分隔的 JavaScript 语句。当一个事件处理函数由一个 HTML 性质定义时，这个处理函数就在 `element` 作用域中执行，而不是在包容窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给事件处理函数 `onsubmit`。而

在 IE 4 事件模型中则不给 `onsubmit` 传递任何参数，可用的 `Event` 对象作为包容 `element` 的 `Window` 对象的属性 `event` 被访问。

如果 `onsubmit` 返回的是 `false`，表单的元素就不会被提交。如果该处理函数返回了其他值，或者什么都没有返回，那么表单就会被正常提交。由于处理函数 `onsubmit` 可以取消表单提交，所以它是进行表单数据验证的理想场所。

参阅

`Form.onreset`、`Form.submit()`；第十九章；DOM 参考手册部分的 `Event`、`EventListener`、`EventTarget`

Form.reset()

JavaScript 1.1

重置表单中的元素

摘要

`form.reset()`

描述

方法 `reset()` 将重置指定的表单，把表单中的每个元素都恢复到它的默认值，就像用户点击了 **Reset** 按钮一样。首先调用的是表单的事件处理程序 `onreset()`，通过让这个处理函数返回 `false` 可以阻止重置表单。

Form.submit()

JavaScript 1.0

提交表单

摘要

`form.submit()`

描述

方法 `submit()` 用于提交指定的表单 `form`，就像用户点击了 **Submit** 按钮一样。要提交的表单由 `form` 的属性 `action`、`method` 和 `encoding` 指定（或者由标记 `<form>` 的性质 `action`、`method` 和 `enctype` 指定），提交后的结果显示在由属性 `target` 指定的窗口或框架中。

方法 `submit()` 和用户提交的表单之间有一个重要差别，即在调用 `submit()` 时不会

调用事件处理程序 `onsubmit()`。如果要用 `onsubmit()` 执行输入验证, 就必须在调用 `submit()` 之前明确地进行验证。

习惯用法

用 **Submit** 按钮让用户提交表单比自己调用 `submit()` 方法更常用。

Form.target JavaScript 1.0; 在 Internet Explorer 3 中该属性是只读的
显示表单结果的窗口

摘要

`form.target`

描述

`Form` 对象的属性 `target` 是一个可读可写的字符串。它指定了要显示表单提交结果的窗口或框架的名字。这个属性的初始值由 HTML 标记 `<form>` 的性质 `target` 设置。如果没有设置这个属性, 那么它的默认值就是表单提交结果和表单出现在同一个窗口。

注意, `target` 的值是框架或窗口的名字, 而不是框架或窗口本身。框架的名字是由标记 `<frame>` 的 `name` 性质设置的。窗口的名字是在调用方法 `Window.open()` 创建窗口时设置的。如果 `target` 指定的窗口名不存在, 浏览器会自动地打开一个新窗口来显示表单提交的结果, 而且将来任何具有相同 `target` 名称的表单都会使用这个新创建的窗口。

`target` 属性还支持四种特殊的名字。`target` 的值为 `"_blank"` 时表示应该创建一个新的、空白的浏览器窗口用来显示表单提交的结果。`"_self"` 是 `target` 的默认值, 表示表单提交的结果应该显示到表单所在的框架或窗口中。`target` 的值为 `"_parent"` 表示结果应该显示到包含表单的框架的父框架中。最后, `target` 的值为 `"_top"`, 表示结果应该显示在顶层框架中, 也就是说, 应该删除所有框架, 结果应该占据整个浏览器窗口。

在 IE 3 中也可以设置这一属性, 但是这样做并不会对表单的实际的 `target` 值产生任何影响。

参阅

`Link.target`

Frame

JavaScript 1.0

Window 对象的一种类型

摘要

```
window.frames[i]  
window.frames.length  
frames[i]  
frames.length
```

描述

尽管有时被称为 Frame 对象，但是严格上说来，这种对象不存在。浏览器窗口中的所有框架都是 Window 对象的一个实例，它们具有的属性、支持的方法和事件处理程序都与 Window 对象相同。要了解详细情况，请参阅 Window 对象和它的属性、方法以及事件处理程序。

但是，在表示顶层浏览器窗口的 Window 对象和表示浏览器窗口中的框架的 Window 对象之间还有几点差别：

- 当设置了框架的 defaultStatus 属性时，只有当鼠标在那个框架中，指定的状态消息才会显示出来。
- 顶层浏览器窗口的 top 属性和 parent 属性总是引用顶层浏览器窗口自身。这两个属性只有对框架来说才真正有用。
- 方法 close() 对表示框架的 Window 对象没有任何作用。

参阅

Window

getClass()

Netscape 3 LiveConnect

返回一个 JavaScript 的 JavaClass

摘要

```
getClass(javaobj)
```

参数

javaobj

一个 `JavaObject` 对象

返回值

javaobj 的 `JavaClass` 对象

描述

`getClass()` 是一个函数，它的参数是一个 `JavaObject` 对象 (*javaobj*)，返回的是那个 `JavaObject` 对象的 `JavaClass` 对象。也就是说，它返回的 `JavaClass` 对象代表指定的 `JavaObject` 表示的 Java 对象所属的 Java 类。

习惯用法

不要将 JavaScript 函数 `getClass()` 与所有 Java 对象的 `getClass` 方法混为一谈。同样，也不要混淆了 JavaScript 的 `JavaClass` 对象和 Java 的 `java.lang.Class` 类。

考虑用如下代码创建的 `Java Rectangle` 对象：

```
var r = new java.awt.Rectangle();
```

r 是一个 JavaScript 变量，存放了一个 `JavaObject` 对象。调用 JavaScript 函数 `getClass()` 将返回代表 `java.awt.Rectangle` 类的 `JavaClass` 对象：

```
var c = getClass(r);
```

比较这个 `JavaClass` 对象和 `java.awt.Rectangle`，就能够看出这一点：

```
if (c == java.awt.Rectangle) ...
```

Java 的 `getClass()` 方法则是以一种不同的方式调用的，而且它的执行方式也与前者完全不同：

```
c = r.getClass();
```

执行了上述代码之后，*c* 就是一个代表 `java.lang.Class` 对象的 `JavaObject` 对象。`java.lang.Class` 是一个 Java 对象，它是 `java.awt.Rectangle` 类的一个 Java 表示。要了解使用 `java.lang.Class` 的详细情况，请参阅 Java 文档。

总之，你会发现，对任何 `JavaObject` 对象 *o* 来说，下面表达式的值总是 `true`：

```
(getClass(o.getClass()) == java.lang.Class)
```

参阅

JSONArray、JavaClass、JavaObject、JavaPackage、Window 对象的 java 属性；第二十二章

Hidden

JavaScript 1.0;在 JavaScript 1.1 中增强了它

用于客户端/服务器端通信的隐藏数据

继承 Input, HTMLElement

摘要

```
form.name  
form.elements[i]
```

属性

Hidden 对象继承了 Input 对象和 HTMLElement 对象的属性,除此之外,它还定义或覆盖了如下属性:

value

一个可读可写的字符串,指定了在提交表单时要传递给服务器的数据(该表单包含 Hidden 对象),初始值由定义 Hidden 对象的 <input> 标记的 value 性质设置。

HTML 语法

一个 Hidden 元素是由标准的 HTML 标记 <input> 创建的:

```
<form>  
  ...  
  <input  
    type = "hidden"           // 声明这是一个 Hidden 元素  
    [ name = "name" ]       // 此后可以引用这个元素的名字  
                             // 指定了 name 属性  
    [value = "value"]       // 在提交表单时要传送的数据  
                             // 指定了 value 属性的初始值  
  >  
  ...  
</form>
```

描述

Hidden 是表单中的不可见元素,它在提交表单时将任意的数据传递给服务器。当你想传输给服务器的是用户输入数据以外的信息时,就可以使用 Hidden 元素。

当一个 HTML 文档由服务器临时生成时,Hidden 元素的另一个用途就是将数据从服务器传输到客户端,以便客户端的 JavaScript 程序能够对它进行处理。例如,服务器

可以以压缩的、机器可读的形式向客户传递原始数据，这些数据是由 Hidden 元素的 value 性质指定的。客户端的 JavaScript 程序（和数据一起传送或在其他帧中传送）可以读取 Hidden 元素的 value 属性值，然后对它进行处理、格式化，并且以一种非压缩的、人工可读的（还可能是用户可配置的）格式将它显示出来。

Hidden 元素对于 CGI 脚本之间的通信也非常有用，其间甚至不需要客户端 JavaScript 的介入。在这种用法中，CGI 脚本会生成一个含有隐藏数据的动态 HTML 页面，然后其中的隐藏数据就会提交给另一个 CGI 脚本。这些隐藏数据可以传达状态信息，如前一个表单的提交结果等。

Cookie 也可以用来在客户端与服务器之间传递数据。但是，Hidden 表单元素与 cookie 之间存在一个重要差别，即 cookie 是暂存于客户端的。

参阅

Document.cookie、Form、HTMLElement、Input；DOM 参考手册部分的 HTMLInputElement

History

JavaScript 1.0

浏览器的 URL 历史

摘要

window.history

frame.history

history

属性

length

这个数字属性声明了浏览器历史列表中的 URL 的个数。由于没有方法确定当前在这个列表中显示的文档的下标，所以知道列表的大小不是非常有用。

方法

back() 后退到以前已经访问过的 URL。

forward() 前进到以前已经访问过的 URL。

go() 转移到以前已经访问过的 URL。

描述

History 对象表示窗口的浏览历史，它维护了一个最近访问过的 Web 页面的列表。但出于安全性和隐私权方面的原因，脚本不能访问这个列表的内容。虽然脚本不能访问 History 对象表示的 URL 列表，但它们能用 `length` 属性确定列表中的 URL 个数，并用 `back()`、`forward()` 和 `go()` 方法再次访问数组中的 URL。

例子

下面一行代码执行的操作与点击 **Back** 按钮执行的操作一样：

```
history.back();
```

下面的代码执行的操作与点击两次 **Back** 按钮执行的操作一样：

```
history.go(-2);
```

参阅

Window 对象的 `history` 属性、Location

History.back()

JavaScript 1.0

返回到前一个 URL

摘要

```
history.back()
```

描述

方法 `back()` 会使 History 对象所属的窗口或框架再次访问位于当前 URL 之前的那个 URL（如果存在）。调用这个方法产生的效果与点击 **Back** 按钮产生的效果一样。它还等价于：

```
history.go(-1);
```

History.forward()

JavaScript 1.0

访问下一个 URL

摘要

```
history.forward()
```

描述

方法 `forward()` 会使 `History` 对象所属的窗口或框架再次访问位于当前 URL 之后的那个 URL (如果存在的话)。调用这个方法产生的效果与点击 **Forward** 按钮产生的效果一样。它还等价于:

```
history.go(1);
```

注意, 如果用户没有使用过 **Back** 按钮或 **Go** 菜单在历史记录中移动, 而且 JavaScript 没有调用过方法 `History.back()` 或 `History.go()`, 那么调用方法 `forward()` 就不会产生任何效果, 因为浏览器已经处于 URL 列表的尾部, 没有可以前进访问的 URL 了。

History.go()

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

再次访问一个 URL

摘要

```
history.go(relative_position)
```

```
history.go(target_string)
```

参数

relative_position

要访问的 URL 在 `History` 的 URL 列表中的相对位置。在 IE 3 中, 这个参数的值必须是 1、0 或 -1。

target_string

要访问的 URL 的子串。方法 `go()` 的这一版本在 JavaScript 1.1 中添加。

描述

方法 `History.go()` 的第一种形式有一个整型的参数, 指定了在 `History` 对象维护的历史列表中的位置距离, 该方法会使浏览器访问这个指定的距离处的 URL。参数值若为正数, 浏览器就会在历史列表中向前移动。参数值为负数, 浏览器就会在历史列表中向后移动。因此, 调用 `history.go(-1)` 等价于调用 `history.back()`, 而且这一调用的效果与用户点击了 **Back** 按钮相同。同样, 调用 `history.go(3)` 再次访问的 URL 与调用 `history.forward()` 三次所访问的 URL 相同。如果调用 `go()` 时传递给它的参数是 0, 就会使浏览器重新装载一次当前的页面 (在 Netscape 3 中, 采用方法 `Location.reload()` 是实现这目标的更好方式)。方法 `go()` 的这一形式在 Netscape

3 的多框架文档中是一个 bug，而且在 Internet Explorer 中，调用它时可以使用的参数值只有 1、0 和 -1。

方法 `History.go()` 的第二种形式是在 JavaScript 1.1 中实现的。它的参数是一个字符串，会使浏览器再次访问第一个含有指定的字符串的 URL（如最近一次访问过的 URL）。

HTMLElement

JavaScript 1.2

所有 HTML 元素的超类

摘要

HTMLElement 是所有表示 HTML 元素的类的超类。因此，在客户端 JavaScript 中的许多环境下都使用了 HTMLElement 对象，下面列出的是使用这一对象的所有方法：

```
document.images[i]
document.links[i]
document.anchors[i]
document.forms[i]
document.forms[i].elements[]
document.elementName
document.formName.elementName
document.all[i]
```

属性

`all[]`*[IE 4]*

该元素包含的所有元素的完整列表，以它们出现的顺序存放。该属性的行为与 `Document.all[]` 属性的行为完全一致。详情可参阅“`Document.all[]`”参考页。

`children[]`*[IE 4]*

该元素的直接子元素。

`className`*[IE 4、Netscape 6]*

一个可读可写的字符串，声明了元素的 `class` 性质的值。该属性是和级联式样表一起使用的。

`document`*[IE 4]*

对包含的 Document 对象的引用。

id[IE 4, Netscape 6]

一个可读可写的字符串，声明了元素的 id 性质的值。该属性用于给元素指定惟一的名称。

innerHTML[IE 4, Netscape 6]

一个可读可写的字符串，声明了元素含有的 HTML 文本，不包括元素自身的开始标记和结束标记。设置这一属性可以用指定的 HTML 文本替换元素的内容。注意，在装载文档时，不能设置这一属性。

innerText[IE 4]

一个可读可写的字符串，声明了元素中含有的纯文本，不包括元素自身的开始标记和结束标记。设置这一属性可以用未解析过的纯文本替换元素的内容。注意，在装载文档时，不能设置这一属性。

lang[IE 4, Netscape 6]

一个可读可写的字符串，声明了 *element* 的 HTML 性质 lang 的值。

offsetHeight[IE 4]

元素和它的内容的高度，以像素计。

offsetLeft[IE 4]

元素 *element* 的 X 坐标，相对于属性 *offsetParent* 指定的包容元素。

offsetParent[IE 4]

声明了定义坐标系统的包容元素，属性 *offsetLeft* 和属性 *offsetTop* 都是以这个坐标系统来计量的。对于大多数元素来说，*offsetParent* 指的就是包容它们的 Document 对象。但是，如果一个元素的容器是动态定位的，那么 *offsetParent* 就是这个动态定位的元素。同样，表元是相对于它们所在的行来定位的。

offsetTop[IE 4]

元素的 Y 坐标，相对于属性 *offsetParent* 所指的包容元素。

offsetWidth[IE 4]

元素和它的内容的宽度，以像素计。

outerHTML[IE 4]

一个可读可写的属性，声明了一个元素的 HTML 文本，其中包括该元素的开始标记和结束标记。把这一属性设置成 HTML 文本串可以完整地替换 *element* 和它的内容。注意，在装载文档时，不能设置该属性。

`outerText`{IE 4}

一个可读可写的属性, 声明了一个元素的纯文本, 其中包括该元素的开始标记和结束标记。通过设置这一属性, 可以把元素 `element` 和它的内容完整地替换为指定的纯文本。注意, 在装载文档时, 不能设置该属性。

`parentElement`{IE 4}

当前元素的直接父元素。该属性是只读的。

`sourceIndex`{IE 4}

元素在包容它的文档的 `Document.all[]` 数组中的下标。

`style`{IE 4、Netscape 6}

元素的内联 CSS 样式性质。设置这个 `Style` 对象的属性可以改变元素的显示样式。参阅第十八章。

`tagName`{IE 4、Netscape 6}

一个只读字符串, 声明定义 `element` 的 HTML 标记的名字。

`title`{IE 4、Netscape 6}

一个可读可写的字符串, 声明了定义 `element` 的 HTML 标记的 `title` 性质值。大多数浏览器都将这个字符串作为元素的“工具提示”的内容。

方法

<code>contains()</code>	判断当前的元素是否含有指定的元素。
<code>getAttribute()</code>	获取一个命名性质的值。
<code>handleEvent()</code>	把 <code>Event</code> 对象传递给适当的事件处理程序。
<code>insertAdjacentHTML()</code>	把 HTML 文本插入到与当前元素邻接的文档中。
<code>insertAdjacentText()</code>	把纯文本插入到与当前元素邻接的文档中。
<code>removeAttribute()</code>	从元素中删除一个性质和它的值。
<code>scrollIntoView()</code>	滚动文档, 使该元素出现在窗口的顶部或底部。
<code>setAttribute()</code>	设置元素的性质值。

事件处理程序

<code>onclick</code>	当用户点击该元素时调用。
<code>ondblclick</code>	当用户双击该元素时调用。

<code>onhelp</code>	当用户请求帮助时调用。只在 IE 4 中可用。
<code>onkeydown</code>	当用户按下 一个键时调用。
<code>onkeypress</code>	当用户按下 一个键或放开 一个键时调用。
<code>onkeyup</code>	当用户放开 一个键时调用。
<code>onmousedown</code>	当用户按下 一个鼠标按钮时调用。
<code>onmousemove</code>	当用户移动鼠标时调用。
<code>onmouseout</code>	当用户把鼠标移开当前元素时调用。
<code>onmouseover</code>	当用户把鼠标移动过 一个元素时调用。
<code>onmouseup</code>	当用户放开 一个鼠标按钮时调用。

描述

HTMLElement 是所有表示 HTML 元素（如 Anchor 类、Form 类、Image 类、Input 类、Link 类，等等）的 JavaScript 类的超类。HTMLElement 定义的事件处理程序是通过 IE 4 和 Netscape 4 中的所有元素类实现的。因为 IE 4 的文档对象模型把文档中的所有 HTML 元素都展现了出来，所以它为这些元素定义了相当多的属性和方法。Netscape 4 没有实现 IE 的属性和方法（除了 `handleEvent()`，它是 Netscape 特有的），但 Netscape 6 实现了 W3C DOM 标准化的属性和方法。参阅 DOM 参考手册部分关于 HTML 元素的标准属性和方法的完整信息。

参阅

Anchor、Form、Image、Input、Link；第十七章；第十九章；DOM 参考手册中的 Element、HTMLElement、Node

HTMLElement.contains()

Internet Explorer 4

一个元素是否包含在另一个元素中

摘要

```
element.contains(target)
```

参数

target

一个 HTMLElement 对象。

返回值

如果元素 *element* 含有元素 *target*，返回值为 `true`，否则为 `false`。

HTMLElement.getAttribute() Internet Explorer 4, Netscape 6

获取一个性质的值

摘要

```
element.getAttribute(name)
```

参数

name

性质名

返回值

元素 *element* 指定性质的值。如果 *element* 没有名为 *name* 的性质，返回值为 `null`。

HTMLElement.handleEvent()

参阅 `Window.handleEvent()`

HTMLElement.insertAdjacentHTML() Internet Explorer 4

在一个元素的周围插入 HTML 文本

摘要

```
element.insertAdjacentHTML(where, text)
```

参数

where

一个字符串，指定了插入文本的位置。值“BeforeBegin”说明要把 *text* 插入元素 *element* 的开始标记之前。值“AfterBegin”说明要把 *text* 插入元素 *element* 的开始标记之后。值“BeforeEnd”说明要把 *text* 插入 *element* 的结束标记之前。值“AfterEnd”说明要把 *text* 插入 *element* 的结束标记之后。

text

要插入的 HTML 文本。

描述

方法 `insertAdjacentHTML()` 将把 HTML 文本 `text` 插入参数 `where` 指定的位置，在 `element` 之内或周围。

HTMLElement.insertAdjacentText()

Internet Explorer 4

在元素之前或之后插入纯文本

摘要

```
element.insertAdjacentText(where, text)
```

参数

`where`

一个字符串，指定了插入文本的位置。值“BeforeBegin”说明要把 `text` 插入元素 `element` 的开始标记之前。值“AfterBegin”说明要把 `text` 插入元素 `element` 的开始标记之后。值“BeforeEnd”说明要把 `text` 插入 `element` 的结束标记之前。值“AfterEnd”说明要把 `text` 插入 `element` 的结束标记之后。

`text`

要插入的纯文本。

描述

方法 `insertAdjacentHTML()` 将把纯文本 `text` 插入到参数 `where` 指定的位置，在 `element` 之内或周围。

HTMLElement.onclick

JavaScript 1.2;HTML 4.0

当用户点击该元素时调用的事件处理程序

摘要

```
<element onclick="handler" ... >  
element.onclick
```

描述

HTMLElement 对象的 `onclick` 属性声明了用户点击 `element` 时要调用的事件处理程序。注意，`onclick` 与 `onmousedown` 不同。点击事件是在同一 `element` 上发生了鼠标按下事件之后又发生了鼠标放开事件时才发生的。

该属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义当前对象的 HTML 标记的 `onclick` 性质指定。如果一个事件处理函数由 HTML 性质定义，那么这个函数就会在元素 `element` 的作用域中执行，而不是在包容窗口的作用域中执行。

在 Netscape 4 的事件模型中，把一个 `Event` 对象作为参数传递给处理函数 `onclick`。但是在 IE 中，并不传递给它任何参数，可应用的 `Event` 对象作为含有 `element` 的 `Window` 对象的 `event` 属性出现。

在 Netscape 4 中，由属性 `Event.which` 声明按下哪个鼠标按钮。但在 IE 4 中，则由属性 `Event.button` 声明按钮的编码。

参阅

`Event`、`Input.onclick`；第十九章；DOM 参考手册部分的 `EventListener`、`EventTarget`、`MouseEvent`

HTMLElement.ondblclick

JavaScript 1.2;HTML 4.0

当用户双击该元素时调用的处理函数

摘要

```
<element ondblclick="handler" ... >  
element.ondblclick
```

描述

`HTMLElement` 对象的 `ondblclick` 属性声明了一个事件处理函数，该函数是在用户双击元素 `element` 时调用的。

该属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义元素对象的 HTML 标记的性质 `ondblclick` 指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 的事件模型中，`Event` 对象作为参数传递给 `ondblclick` 处理函数。但在 IE 4 中并不给它传递任何参数，可应用的 `Event` 对象是作为包容 `element` 的 `Window` 对象的 `event` 属性出现的。

参阅

Event; 第十九章; DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

HTMLElement.onhelp

Internet Explorer 4

当用户按 F1 键时调用的处理函数

摘要

```
<element onhelp="handler" ... >  
element.onhelp
```

描述

`element` 的 `onhelp` 属性声明了一个事件处理函数, 当用户在 `element` 具有键盘焦点时按下 **F1** 键, 就会调用该函数。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由定义元素的 HTML 标记的 `onhelp` 性质设置。如果一个事件处理函数由 HTML 性质定义, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中执行。

调用了处理函数 `onhelp` 之后, Internet Explorer 4 就会显示出内部的帮助窗口。

HTMLElement.onkeydown

JavaScript 1.2;HTML 4.0

当用户按下一个键时调用的处理函数

摘要

```
<element onkeydown="handler" ... >  
element.onkeydown
```

描述

`HTMLElement` 对象的 `onkeydown` 属性声明了一个事件处理函数, 如果在 `element` 上用户按下了某个键, 该事件处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句是由定义该对象的 HTML 标记的 `onkeydown` 性质指定的。如果一个事件处理函数是由 HTML 性质定义的, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中, `Event` 对象作为参数传递给 `onkeydown` 处理函数。但是在 IE 4 事件模型中并不给它传递任何参数, 可应用的 `Event` 对象是作为包容 `element` 的 `Window` 对象的 `event` 属性出现的。

在 Netscape 4 中, 被按下的键的字符代码存放在 `Event` 对象的 `which` 属性中, 而在 IE 4 中, 则存放在 `Event` 对象的 `keyCode` 属性中。可以用方法 `String.fromCharCode()` 将键盘代码转换成一个字符串。在 Netscape 中, 有效的转义键是通过属性 `Event.modifiers` 判断的, 而在 IE 中, 则是由 `Event.shiftKey()` 和其他方法来判断的。

在 Netscape 的事件模型中, 可以通过使该处理函数返回 `false` 来取消对按键的处理。在 IE 4 中, 则要将 `Event.returnValue` 设置为 `false` 来取消操作。另外, 在 IE 中还可以让这个处理函数返回别的键盘代码来代替用户实际按下的键。

通常, 可以使用事件处理函数 `onkeypress` 来代替 `onkeydown` 和 `onkeyup`。

参阅

`Event`、`HTMLElement.onkeypress`; 第十九章

HTMLElement.onkeypress

JavaScript 1.2;HTML 4.0

当用户按下一个键时调用的处理函数

摘要

```
<element onkeypress="handler" ... >  
element.onkeypress
```

描述

`HTMLElement` 对象的 `onkeypress` 属性声明了一个事件处理函数。如果在 `element` 上用户按下了某个键, 该处理函数就会被调用。按键事件是在键按下事件发生之后以及相应的键放开事件发生之前生成的。它与键按下事件相似。除非一定要接收单独的键放开事件, 否则最好用 `onkeypress` 代替 `onkeydown`。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由定义该对象的 HTML 标记的 `onkeypress` 属性指定。如果一个事件处理函数是由 HTML 属性定义的, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中。

在 Netscape 4 事件模型中, `Event` 对象是作为参数传递给 `onkeypress` 处理函数的。但是在 IE 事件模型中并不给它传递任何参数, 可应用的 `Event` 对象是作为包容 `element` 的 `Window` 对象的 `event` 属性出现的。

在 Netscape 中, 被按下的键的字符代码存放在 `Event` 对象的 `which` 属性中, 而在 IE 中, 则存放在 `Event` 对象的 `keyCode` 属性中。你可以用方法 `String.fromCharCode()` 将键盘代码转换成一个字符串。在 Netscape 中, 有效的组合键是通过属性 `Event.modifiers` 判断的, 而在 IE 中则是由 `Event.shiftKey()` 和相关方法来判断的。

在 Netscape 中, 使这个处理函数返回 `false` 可以取消对按键的处理。在 IE 中, 则要将 `Event.returnValue` 设置为 `false`。另外, 在 IE 中, 还可以让这个处理函数返回别的键盘代码来代替用户实际按下的键。

参阅

`Event`; 第十九章

HTMLElement.onkeyup

JavaScript 1.2; HTML 4.0

当用户放开一个键时调用的处理函数

摘要

```
<element onkeyup="handler" ... >
element.onkeyup
```

描述

`HTMLElement` 对象的 `onkeyup` 属性声明了一个事件处理函数, 如果在 `element` 上用户放开了某个键, 该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由定义该对象的 HTML 标记的 `onkeyup` 性质指定。如果一个事件处理函数是由 HTML 性质定义的, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中, `Event` 对象是作为参数传递给 `onkeyup` 处理函数的。但是在 IE 4 事件模型中并不给它传递任何参数, 可应用的 `Event` 对象是作为包容 `element` 的 `Window` 对象的 `event` 属性出现的。

参阅

Event、HTMLElement.onkeydown; 第十九章

HTMLElement.onmousedown

JavaScript 1.2;HTML 4.0

当用户按下一个鼠标按钮时调用的处理函数

摘要

```
<element onmousedown="handler" ... >  
element.onmousedown
```

描述

HTMLElement 对象的 onmousedown 属性声明了一个事件处理函数，当用户在元素 *element* 上按下了一个鼠标按钮时，该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句是由定义该对象的 HTML 标记的 onmousedown 性质指定的。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 *element* 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象是作为参数传递给 onmousedown 处理函数的。但是在 IE 4 事件模型中并不给它传递任何参数，可应用的 Event 对象是作为包容 *element* 的 Window 对象的 event 属性出现的。

在 Netscape 中，由属性 Event.which 声明按下哪一个鼠标按钮。但是在 IE 中，则由属性 Event.button 声明按钮的编码。

参阅

Event、HTMLElement.onclick; 第十九章; DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

HTMLElement.onmousemove

JavaScript 1.2;HTML 4.0

当用户在该元素中移动鼠标时调用的处理函数

摘要

```
<element onmousemove="handler" ... >  
element.onmousemove
```

描述

HTMLElement 对象的 `onmouseover` 性质声明了一个事件处理函数，当用户在元素 `element` 中移动鼠标指针时，该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句是由定义该对象的 HTML 标记的 `onmouseover` 性质指定的。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 `onmouseover` 处理函数。但是在 IE 事件模型中并不给它传递任何参数，可应用的 Event 对象是作为包容 `element` 的 Window 对象的 `event` 属性出现的。

如果你定义了一个 `onmouseover` 事件处理函数，那么当鼠标在 `element` 之中移动时就会生成大量的鼠标移动事件，它们都会被一一地汇报出来。在编写这个事件处理函数要调用的函数时，一定要谨记这一点。

在 Netscape 4 中，不能针对某个独立的元素定义这一事件处理函数，而只能通过用 Window 对象、Document 对象或 Layer 对象的 `captureEvents()` 方法捕捉鼠标移动事件来表示出你的兴趣所在。

参阅

Event、Window.captureEvents(); 第十九章；DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

HTMLElement.onmouseout

JavaScript 1.2;HTML 4.0

当用户把鼠标指针移出该元素时调用的处理函数

摘要

```
<element onmouseout="handler" ... >
element.onmouseout
```

描述

HTMLElement 对象的 `onmouseout` 属性声明了一个事件处理函数，当用户把鼠标指针移出了元素 `element` 的时候，该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义该对象的 HTML 标记的 `onmouseout` 性质指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象是作为参数传递给 `onmouseout` 处理函数的。但是在 IE 事件模型中并不给它传递任何参数，可应用的 Event 对象是作为包容 `element` 的 Window 对象的 `event` 属性出现的。

参阅

Event、Link.onmouseout；第十九章：DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

HTMLElement.onmouseover

JavaScript 1.2; HTML 4.0

当用户将鼠标指针移动过该元素时调用的处理函数

摘要

```
<element onmouseover="handler" ... >  
element.onmouseover
```

描述

HTMLElement 对象的 `onmouseover` 属性声明了一个事件处理函数，当用户把鼠标指针移动过元素 `element` 时，该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义该对象的 HTML 标记的 `onmouseover` 性质指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 中，Event 对象是作为参数传递给 `onmouseover` 处理函数的。但是在 IE 4 中并不给它传递任何参数，可应用的 Event 对象是作为包容 `element` 的 Window 对象的 `event` 属性出现的。

参阅

Event、Link.onmouseover；第十九章

HTMLElement.onmouseupJavaScript 1.2;HTML 4.0

当用户放开一个鼠标按钮时调用的处理函数

摘要

```
<element onmouseup='handler' ... >
element.onmouseup
```

描述

HTMLElement 对象的 onmouseup 属性声明了一个事件处理函数，当用户在元素 *element* 上放开一个鼠标按钮的时候，该处理函数就会被调用。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义该对象的 HTML 标记的 onmouseup 性质指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 *element* 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象是作为参数传递给 onmouseup 处理函数的。但是在 IE 事件模型中并不给它传递任何参数，可应用的 Event 对象是作为包容 *element* 的 Window 对象的 event 属性出现的。

在 Netscape 4 中，属性 Event.which 声明按下哪一个鼠标按钮。但是在 IE 中，则是由属性 Event.button 声明按钮的编码的。

参阅

Event、HTMLElement.onclick；第十九章；DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

HTMLElement.removeAttribute()Internet Explorer 4;Netscape 6

删除一个性质

摘要

```
element.removeAttribute(name)
```

参数

name

要删除的性质的名字。

返回值

如果删除成功, 则返回 `true`, 否则返回 `false`。

描述

方法 `removeAttribute()` 将从元素 `element` 中删除名为 `name` 的性质。如果 `element` 没有名为 `name` 的性质, 那么该方法将返回 `false`。

HTMLElement.scrollIntoView()

Internet Explorer 4

使元素可见

摘要

```
element.scrollIntoView(top)
```

参数

`top`

一个可选的布尔参数, 声明了元素应该滚动到屏幕的顶部还是底部。如果它的值为 `true`, 或者被省略了, 那么元素 `element` 将出现在屏幕的顶部。如果它的值为 `false`, 那么 `element` 将出现在屏幕的底部。

描述

方法 `scrollIntoView()` 可以滚动含有元素 `element` 的文档, 以便使 `element` 的顶部和显示区的顶部对齐, 或者使 `element` 的底部和显示区的底部对齐。

HTMLElement.setAttribute()

Internet Explorer 4; Netscape 6

设置一个性质的值

摘要

```
element.setAttribute(name, value)
```

参数

`name`

要设置的性质的名字。

`value`

要给性质设置的值。

描述

方法 `setAttribute()` 将把元素 `element` 的性质 `name` 的值设置为 `value`。

Image

JavaScript 1.1

嵌在 HTML 文档中的图像

继承 `HTMLElement`

摘要

```
document.images[i]  
document.images.length  
document.image-name
```

构造函数

```
new Image(width, height)
```

参数

`width` 和 `height`

指定图像的宽度和高度，这两个参数可选。

属性

`Image` 对象继承了 `HTMLElement` 对象的所有属性，此外它还定义或继承了如下一些属性，大多数属性对应于 `` 标记的 HTML 性质。在 JavaScript 1.1 和其后的版本中，`src` 属性和 `lowsrc` 属性是可读可写的，设置它们可以改变显示的图像。在不允许文档回流的浏览器（如 IE 3 和 Netscape 4）中，其他属性都是只读的。

`border`

一个整数，声明了图像边线的宽度，以像素计。它的值由性质 `border` 设置。只有当图像位于超链接中时才具有边线。

`complete`

一个只读的布尔值，声明图像是否已经被完全装载进来了。更确切地说，是浏览器是否已经完成了装载图像的过程。即使在装载过程中发生了错误，或者用户放弃了对图像的装载，`complete` 属性仍然会被设置成 `true`。

`height`

一个整数，声明了图像的高度，以像素计。它的值由性质 `height` 设置。

hspace

一个整数, 声明了插入到图像左右的水平距离, 以像素计。它的值由性质 hspace 设置。

lowsrc

一个可读可写的字符串, 声明了替代图像 (通常较小) 的 URL, 当用户的浏览器在低分辨率的显示器上运行时, 就显示该图像。该属性的初始值由标记 的 lowsrc 性质设置。

设置该属性不会即刻产生效果。但假如设置的是属性 src, 就会装载一幅新图像, 如果浏览器又是在低分辨率的系统中运行, 使用的就是属性 lowsrc 的当前值, 而不是 src 刚刚更新的值。

name

一个字符串, 由 HTML 性质 name 设置, 声明了图像的名称。当用性质 name 给图像指定名字后, 除了用 document.images[] 数组代替外, 还可以用文档的 image-name 属性代替, 来引用该图像。

src

一个可读可写的字符串, 声明了浏览器显示的图像的 URL。这个属性的初始值由标记 的 src 性质设置。当把这个属性设置为新图像的 URL 时, 浏览器就会把那幅新图像装载并显示出来 (若在一个低分辨率的系统中, 装载并显示的是由属性 lowsrc 指定的图像)。这对于更新网页的图形外观以响应用户的动作非常有用。此外, 要执行简单的动画, 也可以使用这一属性。

vspace

一个整数, 声明了插在图像上下的垂直距离。它的值由性质 vspace 设置。

width

一个整数, 声明了图像的宽度, 以像素计。它的值由性质 width 设置。

事件处理程序

Image 对象继承了 HTMLElement 对象的事件处理程序, 此外, 它还定义了如下一些事件处理程序:

onabort

如果用户放弃装载图像, 则调用该事件处理程序。

onerror

如果在装载图像的过程中发生了错误，则调用该事件处理程序。

onload

在成功地装载了图像时调用的事件处理程序。

HTML 语法

Image 对象是由标准的 HTML 标记 创建的。在下面的语法中，某些特性被省略了，这是因为 JavaScript 不使用它们，或 JavaScript 不能使用它们：

```
<img src= "url"           // 要显示的图像
      width = "pixels"    // 图像的宽度
      height = "pixels"   // 图像的高度
      ' name = 'image_name' // 一个用于表示该图像的属性名
      [lowsrc = "url"]     // 一个可选的适于在低分辨率下显示的图像
      [border = "pixels"]  // 图像边界的宽度
      [hspace = 'pixels']  // 图像周围的水平距离
      [vspace = 'pixels']  // 图像周围的垂直距离
      'onload= "handler"'  // 当图像完全装载进来时调用的事件处理程序
      'onerror= "handler"' // 在装载过程中发生错误时调用的事件处理程序
      [onabort= 'handler'] // 当用户放弃了装载时调用的事件处理程序
>
```

描述

document.images[] 数组中的 Image 对象，表示标记 嵌入 HTML 文档的图像。它的 src 属性是最有趣的属性，设置该属性可以使浏览器装载并显示新值指定的图像。

可以在 JavaScript 代码中用 Image() 构造函数动态地创建 Image 对象。注意，这个构造函数没有参数指定要装载的图像。与在 HTML 文件中创建图像一样，应该明确地把 src 属性设置成要创建的图像，这样就可以告诉浏览器要装载图像。没有一种方法可以让浏览器显示一个 Image 对象。你所能做的就是迫使 Image 对象通过设置 src 属性把一幅图像装载进来。不过这非常有用，因为这样它就把一幅图像装载到浏览器的缓存中。此后，如果指定 images[] 数组中的一幅图像的 URL 与前者相同，那么它已经被预装载进来了，很快就会显示出来。可以使用如下的代码实现这一点：

```
document.images[2].src = preloaded_image.src;
document.toggle_image.src = toggle_off.src;
```

习惯用法

设置 Image 对象的 src 属性是在网页中实现简单动画效果的一种方法。此外，当用户与页面进行交互时，也可以用它来更换图像。例如，可以使用一幅图像和一个超文本

链接创建 **Submit** 按钮。开始时, 按钮上显示的是一幅禁用的图像, 而且在用户正确地输入所有必要的表单信息之前都保持使用这一图像, 然后更换该图像, 让用户能够提交表单。

Image.onabort

JavaScript 1.1

当用户放弃图像的装载时调用的事件处理程序

摘要

```
<img ... onabort="handler" ... >  
image.onabort
```

描述

Image 对象的属性 `onabort` 声明了一个事件处理函数, 当用户放弃图像的装载 (如点击了 **Stop** 按钮) 时就会调用这个处理函数。

该属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句是由定义 Image 对象的 `` 标记的性质 `onabort` 声明的。如果一个事件处理函数是由 HTML 性质定义的, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中, Event 对象作为参数传递给 `onabort` 处理函数。但是在 IE 事件模型中并不传递给它任何参数, 可应用的 Event 对象作为包容 `element` 的 Window 对象的 `event` 属性出现。

Image.onerror

JavaScript 1.1

在装载图像的过程中发生错误时调用的事件处理程序

摘要

```
<img ... onerror="handler" ... >  
image.onerror
```

描述

Image 对象的属性 `onerror` 声明了一个事件处理函数, 当装载图像的过程中发生了错误时就会调用这个处理函数。

该属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句是由定义 Image 对象的 `` 标记的性质 `onerror` 声明的。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 `onerror` 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 Event 对象作为包容 `element` 的 Window 对象的 `event` 属性出现。

Image.onload

JavaScript 1.1

当图像装载完毕时调用的事件处理程序

摘要

```
<img ... onload="handler" ... >  
image.onload
```

描述

Image 对象的属性 `onload` 声明了一个事件处理函数，当图像装载完毕的时候就会调用这个处理函数。

该属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句是由定义 Image 对象的 `` 标记的性质 `onload` 声明的。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 `onload` 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 Event 对象作为包容 `element` 的 Window 对象的 `event` 属性出现。

参阅

第十九章；DOM 参考手册部分的 Event、EventListener、EventTarget

Input

JavaScript 1.0;在 JavaScript 1.1 中增强了它

HTML 表单中的输入元素

继承 HTMLElement

摘要

`form.elements[i]``form.name`

属性

Input 对象继承了 HTMLElement 对象的属性，此外还定义了如下的一些属性：

`checked`

一个可读可写的布尔值，声明了一个 Checkbox 元素或 Radio 元素当前是否被选中。可以通过设置这一属性的值，来设置这些按钮元素的状态。除了 Checkbox 元素和 Radio 元素外，其他表单元素不使用该属性。

`defaultChecked`

一个只读的布尔值，声明了在默认情况下一个 Checkbox 元素或 Radio 元素是否被选中了。在重置表单时，使用该属性可以把 Checkbox 元素和 Radio 元素恢复到它们的默认值。而对于其他表单元素来说，该属性没有意义。`defaultChecked` 性质对应于创建表单元素的 HTML 标记 `<input>` 的 `checked` 性质。如果存在 `checked` 性质，则 `defaultChecked` 的值就是 `true`，否则为 `false`。

`defaultValue`

声明了在该表单元素中出现的初始文本，在重置表单时可以使用这个值来恢复元素。只有 Text 元素、Textarea 元素和 Password 元素使用该属性。出于安全性的原因，FileUpload 元素没有使用该属性。对于 Checkbox 元素和 Radio 元素来说，与之等价的属性是 `defaultChecked`。

`form`

一个只读的值，引用含有该元素的 Form 对象。使用 `form` 属性，一个表单元素的事件处理程序就可以很容易地引用与它处于同一表单中的兄弟元素。在调用一个事件处理程序时，关键字 `this` 可以引用调用该处理程序的表单元素。因此，事件处理程序可以使用表达式 `this.form` 来引用含有该元素的 Form 对象，从而就可以使用名字或者使用 Form 对象的 `elements[]` 数组中的下标值来引用该元素的兄弟元素。

length

对于 Select 表单元素，这个属性声明的是 options[] 数组中存放的选项数（每个选项由一个 Option 对象表示）。参阅“Select”参考页。

name

一个只读的字符串，由 HTML 性质 name 设置，声明该元素的名字。这个名字可以用来引用该元素，就像上面摘要中说明的那样。参阅“Input.name”参考页。

options[]

对于 Select 表单元素来说，这个数组存放的是 Option 对象，每个 Option 对象表示 Select 对象显示的一个选项。这个数组中的元素数由 Select 元素的 length 属性指定。参阅“Select”参考页。

selectedIndex

对于表单元素 Select 来说，这个整数声明了当前选中的 Select 对象的选项。在 JavaScript 1.1 中，这个属性是可读可写的，但是在 JavaScript 1.0 中，它是只读的。参阅“Select”参考页。

type *[JavaScript 1.1]*

一个只读字符串，声明了表单元素的类型。参阅“Input.type”参考页。

value

一个字符串，声明了表单元素显示的值和（或）在提交表单时该表单元素要提交给服务器的值。参阅“Input.value”参考页。

方法

Input 对象继承了 HTMLElement 对象的方法，此外还定义了如下一些方法：

blur()

将键盘焦点从元素中移开。

click()

在表单元素上模拟鼠标点击。

focus()

把键盘焦点赋予该元素。

select()

对于显示可编辑文本的表单元素来说，选中其中出现的所有文本。

事件处理程序

Input对象继承了HTMLElement对象的事件处理程序, 此外还定义了如下一些事件处理程序:

`onblur`

当用户把键盘焦点从元素中移开时调用的事件处理程序。

`onchange`

对于非按钮的表单元素来说, 当用户输入了一个新值或者选择了一个新值时调用的事件处理程序。

`onclick`

对于那些按钮表单元素来说, 当用户点击或选择了该按钮时调用的事件处理程序。

`onfocus`

当用户把键盘焦点给予该元素时调用的事件处理程序。

描述

表单中的元素都存储在Form对象的`elements[]`数组中。这个数组的元素都是Input对象, 它们分别代表表单中的按钮、输入框和其他控件。许多种输入元素都由标记`<input>`创建, 还有一些由标记`<select>`、`<option>`和`<textarea>`创建。许多属性、方法和事件处理程序都是由这些表单元素共享的, 我们将在这个参考页中介绍它们。特定类型的表单元素的特有行为将在它们各自的参考页中加以说明。

Input对象定义了许多共享的属性、方法和事件处理程序, 但是它们并不是由所有表单元素共享。例如, Button对象可以触发`onclick`事件处理程序, 但是却不能触发`onchange`处理程序, 而Text对象可以触发`onchange`, 但是却不能触发`onclick`。下面的图说明了所有表单元素及与它们相关的属性。

表单元素广义上可以分为两大类。第一类是按钮, 其中包括Button对象、Checkbox对象、Radio对象、Reset对象和Submit对象。这些元素都有`onclick`事件处理程序, 但是没有`onchange`处理程序。同样, 它们能够响应`click()`方法, 但是不能响应`select()`方法。第二类表单元素是显示文本的元素, 其中包括Text对象、Textarea对象、Password对象和FileUpload对象。这些元素都有`onchange`事件处理函数, 但没有`onclick`处理函数, 它们能够响应`select()`方法, 但不能响应`click()`方法。

元素	属性	checked	defaultChecked	defaultValue	form	length	name	options	selectedIndex	type	value	blur()	click()	focus()	select()	onblur	onchange	onclick	onfocus
Button					•		•			•	•	•	•	•		•		•	•
Checkbox	•	•		•	•		•			•	•	•	•	•		•		•	•
Radio	•	•		•	•		•			•	•	•	•	•		•		•	•
Reset				•	•		•			•	•	•	•	•		•		•	•
Submit				•	•		•			•	•	•	•	•		•	•	•	•
Text			•	•	•		•			•	•	•		•	•	•			•
Textarea			•	•	•		•			•	•	•		•	•	•			•
Password			•	•	•		•			•	•	•		•	•	•			•
FileUpload			•	•	•		•			•	•	•		•	•	•			•
Select				•	•	•	•	•	•	•		•	•	•		•	•		•
Hidden				•	•		•			•	•								

Select 元素是一个特例。它不像其他表单元素那样是由 `<input>` 元素创建的，而是由 `<select>` 标记创建的。虽然技术上说来，Select 元素表示的是不同的对象类型，但是把它看作 Input 对象仍然比较方便。

参阅

Button、Checkbox、FileUpload、Form、Hidden、Password、Radio、Reset、Select、Submit、Text、Textarea；第十五章；DOM 参考手册部分的 HTMLInputElement

Input.blur()

JavaScript 1.0

将键盘焦点从表单元素中移开

摘要

`input.blur()`

描述

表单元素的方法 `blur()` 不需要调用事件处理程序 `onblur` 就可以把键盘焦点从元素中移开。它实质上与方法 `focus()` 作用相反。但是 `blur()` 不会把键盘焦点转移到别的地方，所以当你不想触发事件处理程序 `onblur` 时，真正有用的调用时机是你想要用 `focus()` 方法将键盘焦点转移到别处之前。也就是说，应该明确地把键盘焦点从元

素中移开, 这样当别的元素调用 `focus()` 方法隐式地移除键盘焦点时就不会再通知你了。

除了 `Hidden` 对象外, 所有的表单元素都支持 `blur()` 方法。遗憾的是, 并非所有的平台都能对键盘导航提供很好的支持。在 Unix 平台上的 Netscape 2 和 3 中, 只有显示文本的表单元素 (包括 `Text` 对象、`Textarea` 对象、`Password` 对象和 `FileUpload` 对象) 才能用 `blur()` 方法。

`Input.click()` JavaScript 1.0

在表单元素上模拟鼠标点击事件

摘要

`input.click()`

描述

表单元素的 `click()` 方法模拟了表单元素上的一次鼠标点击事件, 但是并不调用该元素的事件处理程序 `onclick`。

该方法并不常用。由于它不调用事件处理程序 `onclick`, 所以对于 `Button` 元素来说, 它没有太大的用途, 因为除了 `onclick` 处理程序定义的行为之外, `Button` 元素再没有其他的行为了。调用 `Submit` 对象的 `click()` 方法可以提交表单, 调用 `Reset` 对象的 `click()` 方法可以重置表单, 但是调用 `Form` 对象自身的 `submit()` 方法和 `reset()` 方法来实现这一点更直接一些。

`Input.focus()` JavaScript 1.0

把键盘焦点赋予表单元素

摘要

`input.focus()`

描述

表单元素的方法 `focus()` 无须调用事件处理程序 `onfocus` 就可以把键盘焦点转移到输入元素。也就是说, 就键盘导航和键盘输入而言, 它可以激活元素。因此, 如果调用 `Text` 元素的 `focus()` 方法, 用户输入的所有文本都会显示在那个元素中。如果调用 `Button` 元素的 `focus()` 方法, 那么用户就可以从键盘来调用那个按钮。

除了 Hidden 元素外，所有的表单元素都支持 `focus()` 方法。遗憾的是，并非所有的平台都能对键盘导航提供很好的支持。在 Netscape 的 Unix 版本中，只有显示文本的表单元素（包括 Text 对象、Textarea 对象、Password 对象和 FileUpload 对象）才能使用 `focus()` 方法。

Input.name

JavaScript 1.0

表单元素的名字

摘要

`input.name`

描述

表单元素的属性 `name` 是一个只读字符串，它的值由定义该表单元素的 HTML 标记 `<input>` 的 `name` 性质设置。

一个表单元素的名字有两个用途。第一，在提交表单时可以使用它。表单中所有元素的数据通常都是以下面这种形式提交的：

```
name=value
```

这里对 `name` 和 `value` 都进行了编码，这是传输必需的。如果没有给表单元素指定名字，那么元素的数据就不能提交给 Web 服务器。

`name` 属性的第二个用途是在 JavaScript 代码中引用表单元素。元素的名字将成为含有该元素的表单的一个属性。该属性的值就是对元素的引用。例如，假设有一个表单 `address`，它含有一个名为 `zip` 的文本输入元素，那么 `address.zip` 引用的就是那个文本输入元素。

对于 Radio 元素和 Checkbox 表单元素来说，定义多个相关的对象，每个对象具有相同的 `name` 属性是很常见的。在这种情况下，数据传递给 Web 服务器时采用如下格式：

```
name=value1,value2,...,valuen
```

同样，在 JavaScript 中，共享同一个名字的元素都会成为具有那个名字的数组中的元素。因此，如果表单 `order` 中的四个 Checkbox 对象共享名字 `options`，就可以将它们作为 `order.options[]` 数组的元素来访问。

Input.onblur

JavaScript 1.0

当表单元失去焦点时调用的事件处理程序

摘要

```
<input type="type" onblur="handler">
```

```
input.onblur
```

描述

Input对象的属性onblur声明了一个事件处理函数,这个函数是在用户把键盘焦点从该输入元素中移开时调用的。虽然调用blur()方法也可以将焦点从元素中移开,但是它并不能调用那个对象的onblur处理函数。但是要注意,调用focus()把焦点转移到其他元素可以引发当前具有焦点的元素的onblur事件处理函数被调用。

这个属性的初始值是一个含有JavaScript语句的函数,其中的JavaScript语句由定义该对象的HTML标记的onblur性质指定,各个语句之间用分号分隔。如果一个事件处理函数是由HTML性质定义的,那么它就会在element的作用域中执行,而不是在包容它的窗口的作用域中执行。

在Netscape 4事件模型中,Event对象作为参数传递给onblur处理函数。但是在IE事件模型中并不传递给它任何参数,可应用的Event对象作为包容element的Window对象的event属性出现。

除了Hidden元素外,其他所有表单元素都支持事件处理函数onblur。但是在Unix平台上的Netscape中,只有那些显示文本的表单元素(包括Text对象、Textarea对象、Password对象和FileUpload对象)才能调用这个处理函数。还要注意,在JavaScript 1.1中,Window对象也定义了onblur事件处理函数。

参阅

Window.onblur; 第十九章; DOM参考手册部分的Event、EventListener、EventTarget

Input.onchange

JavaScript 1.0

当改变表单元的值时调用的事件处理程序

摘要

```
<input type="type" onchange="handler">
```

```
input.onchange
```

描述

Input对象的属性onchange声明了一个事件处理函数,当用户改变了表单元素显示的值时就会调用这个处理函数。这种改变既可以是对Text元素、Textarea元素、Password元素或FileUpload元素中显示的文本的编辑,也可以是选择或取消选择Select元素中的某一个选项。注意,只有当用户做出上述改变时才会调用这个事件处理函数,当JavaScript程序改变元素显示的值时并不会调用它。

还要注意,并非每次用户在文本表单元素中输入一个字符或删除一个字符都会调用这个处理函数。onchange不是为逐个字符的事件处理类型设计的。相反,只有在用户的编辑全部完成后才会调用它。浏览器假定在键盘焦点从当前元素移动到其他元素(如用户点击了表单中另一个元素)时用户的编辑就完成了。要了解逐个字符的事件处理类型请参阅“HTMLInputElement onkeypress”。

Hidden元素和所有按钮元素都不能使用onchange事件处理函数。它们(Button元素、Checkbox元素、Radio元素、Reset元素和Submit元素)使用的是onclick处理函数。

这个属性的初始值是一个含有JavaScript语句的函数,其中的JavaScript语句由定义该对象的HTML标记的onchange性质指定,各个语句之间用分号分隔。如果一个事件处理函数是由HTML性质定义的,那么它就会在element的作用域中执行,而不是在包容它的窗口的作用域中执行。

在Netscape 4事件模型中,Event对象作为参数传递给onchange处理函数。但是在IE事件模型中并不给它传递任何参数,可应用的Event对象作为包容element的Window对象的event属性出现。

参阅

HTMLInputElement.onkeypress; 第十九章; DOM参考手册部分的Event、EventListener、EventTarget

Input.onclick JavaScript 1.0;在JavaScript 1.1中增强了它
点击表单元素时调用的事件处理程序

摘要

```
<input type="type" onclick="handler">  
input.onclick
```

描述

Input对象的属性 `onclick` 声明了一个事件处理函数，当用户点击该输入元素时就会调用这个处理函数。但是调用该元素的 `click()` 方法时并不会调用 `onclick` 处理函数。

只有按钮元素可以调用 `onclick` 事件处理函数。它们是 `Button` 元素、`Checkbox` 元素、`Radio` 元素、`Reset` 元素和 `Submit` 元素。其他表单元素使用的是 `onchange` 处理函数。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义该对象的 HTML 标记的 `onclick` 性质指定，各个语句之间用分号分隔。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，`Event` 对象作为参数传递给 `onclick` 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 `Event` 对象作为包容 `element` 的 `Window` 对象的 `event` 属性出现。

注意，在点击了 `Reset` 元素或 `Submit` 元素时，执行的是默认动作，即分别重置和提交包含它们的表单。可以使用这两种元素的 `onclick` 处理函数来执行默认动作之外的动作。在 JavaScript 1.1 中，还可以让 `onclick` 返回 `false` 来阻止执行这两种元素执行默认的动作。也就是说，如果 `Reset` 按钮的 `onclick` 处理函数返回的是 `false`，那么表单就不会被重置。如果 `Submit` 按钮的 `onclick` 处理函数返回的是 `false`，那么表单就不会被提交。注意，你所做的恰恰就是 `Form` 对象自身的 `onsubmit` 事件处理函数和 `onreset` 事件处理函数所实现的。

最后要注意，`Link` 对象也定义了 `onclick` 事件处理函数。

参阅

`Link.onclick`；第十九章；DOM 参考手册部分的 `EventListener`、`EventTarget`

Input.onfocus

JavaScript 1.0

当表单元元素获得焦点时调用的事件处理程序

摘要

```
<input type="type" onfocus="handler">
input.onfocus
```

描述

Input 对象的属性 `onfocus` 声明了一个事件处理函数，当用户把键盘焦点转移到该元素时就会调用这个处理函数。但是当调用了该元素的 `focus()` 方法设置焦点时并不会调用它的 `onfocus` 处理函数。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句是由定义该对象的 HTML 标记的 `onfocus` 性质指定的，各个语句之间用分号分隔。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 `element` 的作用域中执行，而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 `onfocus` 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 Event 对象作为包容 `element` 的 Window 对象的 `event` 属性出现。

除了 Hidden 元素外，其他所有表单元素都支持事件处理函数 `onfocus`。但是在 Unix 平台上的 Netscape 中，只有那些显示文本的表单元素（包括 Text 对象、Textarea 对象、Password 对象和 FileUpload 对象）才能调用这个处理函数。还要注意，在 JavaScript 1.1 中，Window 对象也定义了 `onfocus` 事件处理函数。

参阅

Window.onfocus; 第十九章; DOM 参考手册部分的 Event 和 EventTarget

Input.select()

JavaScript 1.0

选择表单元素中的文本

摘要

`input.select()`

描述

方法 `select()` 可以选择在 Text、Textarea、Password 和 FileUpload 元素中显示的文本。选择文本后产生的效果，则根据平台的不同有所区别，不过一般说来，调用这个方法制造的结果，与用户拖动鼠标移过指定 Text 对象的所有文本所产生的结果一样。在大多数平台上，产生的效果如下：

- 被选定的文本将高亮显示，通常显示使用的是相反的颜色。

- 如果用户再次输入一个字符时文本仍然保持选定的状态, 那么选定的文本将被删除, 并由新输入的字符替换。
- 在某些平台上被选定的文本可用于剪切和粘贴。

用户通常可以通过点击Text对象或移动光标取消对文本的选定。一旦取消了对文本的选定, 用户就可以添加或删除单个的字符, 无须替换整个文本值。

Input.type

JavaScript 1.1

表单元素的类型

摘要

input.type

描述

表单元素的 `type` 属性是一个只读的字符串, 声明了表单元素的类型。下面的表列出了各种表单元素可能具有的属性值。

对象类型	标记	类型属性
Button	<code><input type="button"></code>	<code>"button"</code>
Checkbox	<code><input type="checkbox"></code>	<code>"checkbox"</code>
FileUpload	<code><input type="file"></code>	<code>"file"</code>
Hidden	<code><input type="hidden"></code>	<code>"hidden"</code>
Password	<code><input type="password"></code>	<code>"password"</code>
Radio	<code><input type="radio"></code>	<code>"radio"</code>
Reset	<code><input type="reset"></code>	<code>"reset"</code>
Select	<code><select></code>	<code>"select-one"</code>
Select	<code><select multiple></code>	<code>"select-multiple"</code>
Submit	<code><input type="submit"></code>	<code>"submit"</code>
Text	<code><input type="text"></code>	<code>"text"</code>
Textarea	<code><textarea></code>	<code>"textarea"</code>

要注意的是, `Select` 元素具有两种可能的 `type` 值, 这由它是否允许多项选择决定。另外还要注意, 与其他的输入元素属性不同, `type` 在 JavaScript 1.0 中不可用。

Input.value

Netscape 2;在 Internet Explorer 3 中存在 bug

表单元素显示的值或要提交的值

摘要

`input.value`

描述

表单元素的属性 `value` 是一个可读可写的字符串, 声明了表单元素显示的值或在提交表单时该表单元素要提交的值。例如, `Text` 元素的 `value` 属性既是用户输入的值, 也是要随表单一起提交的值。但对于 `Checkbox` 对象而言, 属性 `value` 声明的就不是该对象要显示的字符串, 而是在 `Checkbox` 元素被选中的情况下, 随表单一起提交的值。

这个属性的初始值 `value` 由定义该表单元素的 `HTML` 标记的 `value` 性质设置。

对于 `Button` 对象、`Submit` 对象和 `Reset` 对象来说, `value` 属性声明的是在按钮上显示的文本。在某些平台上, 改变这些元素的 `value` 属性实际上改变的是屏幕上的按钮显示的文本。但是, 这种改变并非在所有的平台上都有效, 所以它不是一种可取的方法。改变按钮的标签还会改变按钮大小, 这会使它覆盖文档的其他部分。

`Select` 元素也有 `value` 属性, 虽然它与其他表单元素的 `value` 属性相同, 但一般不使用。`Select` 元素发送的是它含有的 `Option` 对象的 `value` 属性指定的值。

出于安全性的原因, `FileUpload` 元素的 `value` 属性是只读的。

JavaArray

Netscape 3 LiveConnect

Java 数组的 JavaScript 表示

摘要

```
javaarray.length // The length of the array
javaarray[index] // Read or write an array element
```

属性

`length`

一个只读的整数, 声明了 `JavaArray` 对象表示的 Java 数组中的元素数。

描述

JavaArray 对象是 Java 数组的 JavaScript 表示, 它使 JavaScript 代码能够使用你所熟悉的 JavaScript 数组语法对数组元素进行读写操作。而且, JavaArray 对象的 length 字段还声明了 Java 数组中的元素个数。

在对数组元素进行读写操作时, 系统会自动执行 JavaScript 表示和 Java 表示之间的数据转换。参阅第二十二章的全面介绍。

习惯用法

注意, Java 数组和 JavaScript 数组有两方面不同之处。第一, Java 数组具有固定的长度, 这个长度是在创建它时指定的。因此, JavaArray 对象的 length 字段是只读的。第二点不同之处在于 Java 数组的元素是有类型的(如它们的元素都必须具有相同的数据类型)。如果给数组元素设置了错误类型的值, 就会引发 JavaScript 错误或异常。

例子

java.awt.Polygon 是一个 JavaClass 对象。我们可以使用如下的代码创建一个 JavaObject 对象来表示这个类的实例:

```
p = new java.awt.Polygon();
```

对象 p 具有属性 xpoints 和 ypoints, 它们都是 JavaArray 对象, 代表整型的 Java 数组。我们可以用如下的 JavaScript 代码初始化这两个数组的内容:

```
for(int i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random()*100);
for(int j = 0; j < p.ypoints.length; j++)
    p.ypoints[j] = Math.round(Math.random()*100);
```

参阅

getClass()、JavaClass、JavaObject、JavaPackage、Window 对象的 java 属性; 第二十二章

JavaClass

Netscape 3 LiveConnect

Java 类的 JavaScript 表示

摘要

```
javaclass.static_member // 读写静态 Java 字段或方法
new javaclass(...)      // 创建新的 Java 对象
```

属性

每个 `JavaClass` 对象含有的都是与它表示的 Java 类的公有静态字段和方法同名的属性。用这些属性可以读写那个类的静态字段并调用它的静态方法。每个 `JavaClass` 对象的属性都不同，可以使用 `for/in` 循环来枚举一个给定的 `JavaClass` 对象的属性。

描述

`JavaClass` 对象是 Java 类的 JavaScript 表示。它的属性表示的是它所代表的类的公有静态字段和方法（有时称它们为类字段和类方法）。注意，`JavaClass` 对象没有表示 Java 类的实例字段的属性，因为 Java 类的实例都是由 `JavaObject` 对象表示的。

`JavaClass` 对象实现了 `LiveConnect` 的功能，这使得 JavaScript 程序可以使用常规的 JavaScript 语法对 Java 类的静态变量进行读写操作。它还提供了让 JavaScript 调用 Java 类的静态方法的功能。

除了允许 JavaScript 代码读写 Java 变量以及方法的值外，`JavaClass` 对象还允许 JavaScript 程序创建 Java 对象（由 `JavaObject` 对象表示），只需要使用关键字 `new` 并调用 `JavaClass` 对象的构造函数即可。

JavaScript 和 Java 之间通过 `JavaClass` 对象通信必须进行数据转换，`LiveConnect` 会自动处理这一转换。参阅第二十二章的全面介绍。

习惯用法

要记住，Java 是一种强类型的语言。这就是说，对象的每个字段都要具有特定的数据类型，只能把它们设置成属于那种类型的值。如果设置字段时采用的数据类型不对，就会引发 JavaScript 错误或异常。调用方法时传递的参数类型不正确也会引发错误或异常。

例子

`java.lang.System` 是一个 `JavaClass` 对象，表示 Java 中的 `java.lang.System` 类。可以使用如下代码对这个类的静态字段进行读操作：

```
var java_console = java.lang.System.out;
```

也可以使用下面的代码调用这个类的静态方法：

```
var version = java.lang.System.getProperty("java.version");
```

最后，用 `JavaClass` 对象还能创建新的 Java 对象，代码如下：

```
var java_date = new java.lang.Date();
```

参阅

`getClass()`、`JSONArray`、`JavaObject`、`JavaPackage`、`Window.java`；第二十二章

JavaObject

Netscape 3 LiveConnect

Java 对象的 JavaScript 表示

摘要

`javaobject.member` // 读写实例字段或方法

属性

每个 `JavaObject` 对象的属性和方法都与它表示的 Java 对象的公有实例字段和方法（而不是静态的属性和方法，或者说类方法、类属性）同名。用这些属性可以读写那个类的公有字段并调用它的公有方法。一个给定的 `JavaObject` 对象的属性显然是由它所表示的 Java 对象的类型决定的。可以使用 `for/in` 循环枚举一个给定的 `JavaObject` 对象的属性。

描述

`JavaObject` 对象是 Java 对象的 JavaScript 表示。它的属性表示它所定义的对象的所有实例字段和公有实例方法（对象的类或静态字段和对象方法由 `JavaClass` 的对象表示。）

`JavaObject` 对象实现了 `LiveConnect` 的功能，这使得 JavaScript 程序可以使用常规的 JavaScript 语法对 Java 对象的公有实例字段进行读写操作。此外，它还提供了让 JavaScript 调用 Java 对象的方法的功能。`LiveConnect` 会自动处理 JavaScript 表示和 Java 表示之间的数据转换。参阅第二十二章的全面介绍。

习惯用法

要记住，Java 是一种强类型的语言。这就是说，对象的每个字段都要具有特定的数据类型，只能把它们设置成属于那种类型的值。例如，`java.awt.Rectangle` 对象的 `width` 字段是一个整型字段，如果把它设置成一个字符串就会引发 JavaScript 错误或异常。

例子

`java.awt.Rectangle` 是一个 `JavaClass` 对象，它表示 `java.awt.Rectangle` 类。我们可以创建一个 `JavaObject` 对象来表示这个类的实例，代码如下：

```
var r = new java.awt.Rectangle(0,0,4,5);
```

然后我们就可以使用如下代码对 `JavaObject` 对象 `r` 的公有实例变量进行读操作：

```
var perimetreter = 2*r.width + 2*r.height;
```

我们还可以使用 `JavaScript` 语法来设置 `r` 的公有实例变量：

```
r.width = perimetreter / 2;  
r.height = perimetreter / 2;
```

参阅

`getClass()`、`JSONArray`、`JavaClass`、`JavaPackage`、`Window.java`；第二十二章

JavaPackage

Netscape 3 LiveConnect

Java 包的 JavaScript 表示

摘要

```
package.package_name // 引用另一个  
package.class_name   // 引用一个 JavaClass 对象
```

属性

`JavaPackage` 对象的属性是它含有的 `JavaPackage` 对象和 `JavaClass` 对象的名字。每个 `JavaPackage` 对象的属性都不同。注意，不能使用 `JavaScript` 的 `for/in` 循环遍历 `Package` 对象的属性名列表。要了解一个给定的包中含哪些包和类，请参阅 `Java` 的参考手册。

描述

`JavaPackage` 对象是 `Java` 包的 `JavaScript` 表示。在 `Java` 中，所谓包就是一组相关的类的集合。不过，在 `JavaScript` 中，一个 `JavaPackage` 对象除了可以含有类（由 `JavaClass` 对象表示）之外，还可以含有其他的 `JavaPackage` 对象。

`Window` 对象具有 `java`、`netscape` 和 `sun` 属性，分别表示包层次的 `java.*`、`netscape.*` 和 `sun.*`。这些 `JavaPackage` 对象定义了引用其他 `JavaPackage` 对象的属性。例如，

java.lang 和 java.net 引用 *java.lang* 和 *java.net* 包。JavaPackage 对象 java.awt 具有属性 Frame 和 Button，它们引用 JavaClass 对象，分别表示类 *java.awt.Frame* 和 *java.awt.Button*。

Window 对象还定义了 Packages 属性，该属性是根 JavaPackage 对象，它的属性引用了所有已知包层次的根。例如，表达式 Packages.java.awt 等价于 java.awt。

用 for/in 循环不能确定一个 JavaPackage 对象含有的包名和类的名称。必须预先获得这些信息。在任何一本 Java 参考手册中都会有这些信息，你也可以自己检测 Java 的类层次。

第二十二章对 Java 包、类和使用进行了详细的介绍。

参阅

JavaArray、JavaClass、JavaObject；Window 对象的 java、netscape、sun、Packages 属性；第二十二章

JSObject

Netscape 3;Internet Explorer 4

JavaScript 对象的 Java 表示

摘要

```
public final class netscape.javascript.JSObject extends Object
```

方法

call()	调用 JavaScript 对象的方法。
eval()	在 JavaScript 对象环境中执行一个 JavaScript 代码串。
getMember()	获取 JavaScript 对象的一个属性值。
getSlot()	获取 JavaScript 对象的一个数组元素的值。
getWindow()	获取一个“根” JSObject 对象，表示 JavaScript 中代表浏览器窗口的 Window 对象。
removeMember()	删除 JavaScript 对象的一个属性。
setMember()	设置 JavaScript 对象的一个属性的值。
setSlot()	设置 JavaScript 对象的一个数组元素的值。

`toString()` 调用 JavaScript 对象的 `toString()` 方法，并且返回该方法执行的结果。

描述

`JSObject` 是一个 Java 类，而不是一个 JavaScript 对象，在 JavaScript 程序中不能使用它。相反，`JSObject` 对象由 Java 小程序使用，这些小程序通过读写 JavaScript 属性和数组元素、调用 JavaScript 的方法以及执行 JavaScript 的代码串与 JavaScript 进行通信。显而易见，由于 `JSObject` 是一个 Java 类，所以要使用它，还要了解 Java 的程序设计方法。

要了解有关 `JSObject` 程序设计技术的详细内容，请参阅第二十二章。

参阅

第二十二章

`JSObject.call()`

Netscape 3;Internet Explorer 4

调用 JavaScript 对象的一个方法

摘要

```
public Object call(String methodName, Object args[])
```

参数

methodName

要调用的 JavaScript 方法的名字。

args[]

作为参数传递给该方法的 Java 对象数组。

返回值

一个 Java 对象，表示 JavaScript 方法的返回值。

描述

Java `JSObject` 类的方法 `call()` 将调用一个已命名的 JavaScript 方法，这个方法 `JSObject` 表示。参数将以 Java 对象数组的形式传递给该方法。JavaScript 方法的返回值是一个 Java 对象。

有关把该方法的参数从 Java 对象转换成 JavaScript 值, 以及把 JavaScript 方法的返回值从 JavaScript 值转换成 Java 对象执行的数据转换, 请参见第二十二章中的详细介绍。

JSObject.eval()

Netscape 3;Internet Explorer 4

执行一个 JavaScript 代码串

摘要

```
public Object eval(String s)
```

参数

s 一个字符串, 含有任意的 JavaScript 语句, 各语句之间用分号隔开。

返回值

s 中最后一个表达式的 JavaScript 值, 将被转换成一个 Java 对象。

描述

Java JSObject 类的 eval() 方法将在 JSObject 指定的 JavaScript 对象环境中执行字符串 *s* 中的 JavaScript 代码。它的行为与 JavaScript 的全局函数 eval() 相似。

参数 *s* 是一个含有任意多条 JavaScript 语句的字符串, 其中的语句用分号隔开, 执行顺序与它们出现的顺序相同。eval() 返回的是计算 *s* 中的最后一条语句或表达式得到的值。

JSObject.getMember()

Netscape 3;Internet Explorer 4

读 JavaScript 对象的一个属性

摘要

```
public Object getMember(String name)
```

参数

name

要读的属性的名字。

返回值

一个 Java 对象, 含有指定的 JSObject 对象命名的属性的值。

描述

Java `JSObject` 类的 `getMember()` 方法将读取 JavaScript 对象的一个已命名的属性的值，并且将它返回给 Java。返回的值是另一个 `JSObject` 对象或一个 `Double` 对象、`Boolean` 对象或者 `String` 对象，不过返回时这个值都是通用的 `Object` 对象，必须对它进行必要的类型转换。

`JSObject.getSlot()`

Netscape 3;Internet Explorer 4

读一个 JavaScript 对象的数组元素

摘要

```
public Object getSlot(int index)
```

参数

index

要读取的数组元素的下标。

返回值

位于指定的 *index* 的 JavaScript 对象的数组元素值。

描述

Java `JSObject` 类的 `getSlot()` 方法可以读取位于 JavaScript 对象的指定 *index* 处的数组元素值，并且把这个值返回给 Java。返回值是另一个 `JSObject` 对象或一个 `Double` 对象、`Boolean` 对象或者 `String` 对象，不过返回时这个值都是通用的 `Object` 对象，必须对它进行必要的类型转换。

`JSObject.getWindow()`

Netscape 3;Internet Explorer 4

返回代表浏览器窗口的初始 `JSObject` 对象

摘要

```
public static JSObject getWindow(java.applet.Applet applet)
```

参数

applet

一个 `Applet` 对象，在这个窗口中运行的对象就是要获取 `JSObject` 对象。

返回值

一个 `JSObject` 对象，表示 JavaScript 中用于代表浏览器窗口的 `Window` 对象，这个窗口含有指定的小程序 *applet*。

描述

`getWindow()` 是所有的 Java 小程序都要调用的第一个 `JSObject` 方法。由于 `JSObject` 类没有定义构造函数，所以调用静态方法 `getWindow()` 是惟一一种获得初始“根” `JSObject` 对象的方法。

`JSObject.removeMember()`

Netscape 3;Internet Explorer 4

删除 JavaScript 对象的一个属性

摘要

```
public void removeMember(String name)
```

参数

name

要从 `JSObject` 对象中删除的属性的名字。

描述

Java `JSObject` 类的方法 `removeMember()` 可以从 `JSObject` 对象表示的 JavaScript 对象中删除一个已命名的属性。

`JSObject.setMember()`

Netscape 3;Internet Explorer 4

设置 JavaScript 对象的一个属性

摘要

```
public void setMember(String name, Object value)
```

参数

name

`JSObject` 对象中要设置的属性的名字。

value

给指定的属性设置的值。

描述

Java JSObject 类的方法 `setMember()` 可以从 Java 中设置 JavaScript 对象的一个已命名的属性的值。指定的值 `value` 可以是任何类型的 Java 对象。不过原始的 Java 值不会传递给这个方法。在 JavaScript 中, 指定的值 `value` 通过一个 `JavaObject` 对象访问。

JSObject.setSlot()

Netscape 3;Internet Explorer 4

设置 JavaScript 对象的一个数组元素

摘要

```
public void setSlot(int index, Object value)
```

参数

`index`

要在 JSObject 对象中设置的数组元素的下标。

`value`

要给指定的数组元素设置的值。

描述

Java JSObject 类的方法 `setSlot()` 可以从 Java 中设置 JavaScript 对象的已编码的数组元素值。指定的值 `value` 可以是任何类型的 Java 对象。不过原始的 Java 值不会传递给这个方法。在 JavaScript 中, 指定的值 `value` 作为一个 `JavaObject` 对象访问。

JSObject.toString()

Netscape 3;Internet Explorer 4

返回一个 JavaScript 对象的字符串值

摘要

```
public String toString()
```

返回值

一个字符串, 是调用 Java JSObject 对象表示的 JavaScript 对象的 `toString()` 方法返回的。

描述

Java JSObject 类的 `toString()` 方法将调用 JSObject 表示的 JavaScript 对象的 `toString()` 方法, 并且返回调用后的结果。

Layer

只有 Netscape 4 可用; Netscape 6 不再支持

DHTML 文档中的一个独立层

摘要

`document.layers[i]`

构造函数

`new Layer(width, parent)`

参数

width

新层的宽度，以像素计。

parent

一个 Layer 对象或 Window 对象，是新创建的层的父层或父窗口。这个参数是可选的。如果省略了它，新层就是当前窗口的子窗口。

注释

构造函数 `Layer()` 将创建一个新的 Layer 对象并返回它。可以使用各种 Layer 属性和方法来设置新层的大小、位置以及其他性质，下面的列表逐一介绍了这些属性和方法。特别地，要使新层可见，就必须将 `hidden` 属性设置为 `false`。特别需要注意的是，如何使用 `src` 属性和 `load()` 方法设置层的内容。另外，还可以通过设置层的 `document` 属性动态地生成它的内容。

注意，只有在当前文档和它的所有层都装载完毕后，才能调用构造函数 `Layer()`。

属性

above

一个只读属性，引用在叠放顺序中恰好位于层 `layer` 之上的 layer 对象。如果不存在这样的层，`above` 的值为 `null`。

background

一个 Image 对象，指定了层背景中显示的图象。这个属性的初始值由 `<layer>` 标记的 `background` 性质设置。可以通过设置属性 `background.src` 改变层背景中显示的图像。如果将这个属性设为 `null`，那么层背景中就没有要显示的图像，显示的是背景颜色（由属性 `bgColor` 设置）。

`below`

一个只读属性，引用在叠放顺序中恰好位于层 `layer` 之下的那个 `layer` 对象。如果不存在这样的层，`below` 的值为 `null`。

`bgcolor`

一个可读可写的字符串属性，指定了 `layer` 的背景颜色。这个属性的初始值由 `<layer>` 标记的 `bgcolor` 性质设置。注意，由于 `layer.background` 的优先级高于 `layer.bgColor`，所以只有当 `layer` 的属性 `background.src` 的值为 `null` 时，该属性指定的颜色才会显示出来。

`clip.bottom`

层剪切区底边的 Y 坐标，相对于 `layer.top`。

`clip.height`

层剪切区的高度。设置该属性还会设置 `layer.clip.bottom` 的值。

`clip.left`

层剪切区左边界的 X 坐标，相对于 `layer.left`。

`clip.right`

层剪切区右边界的 X 坐标，相对于 `layer.right`。

`clip.top`

层剪切区上边界的 Y 坐标，相对于 `layer.top`。

`clip.width`

层剪切区的宽度。设置该属性还会设置 `layer.clip.right` 的值。

`document`

对层含有的 `Document` 对象的只读引用。

`hidden`

指定了层是可见 (`false`) 的还是隐藏 (`true`) 的。把该属性设置为 `true` 将隐藏层，把它设置为 `false`，就能够使层可见。

`layers[]`

一个数组，存放的是层中含有的子 `layer` 对象，等价于层的 `document.layers[]` 数组。

`left`

一个可读可写的整数，指定了层的 X 坐标，这个坐标相对于该层的包容层或包容文档。设置该属性可以左右移动层。它等价于属性 `x`。

`name`

一个可读可写的字符串，指定了层的名字。这个属性的初始值由创建层的 HTML 标记的 `name` 性质或 `id` 性质设置。这个值同时还作为 Document 对象的属性名，用来引用这个 Layer 对象。

`pageX, pageY`

可读可写的整数，指定了层相对于顶层文档的 X 坐标和 Y 坐标。注意，这个坐标相对于顶层页面，而不是相对于任何包容层。

`parentLayer`

对包容层（层的父层）的 Layer 对象或 Window 对象的只读引用。

`siblingAbove, siblingBelow`

引用的是在叠放顺序中恰好位于层之上和之下的兄弟 Layer 对象（即与 Layer 属于同一个父层的层）。如果不存在这样的层，它们的值为 `null`。

`src`

一个可读可写的字符串，指定了层内容的 URL（如果存在）。将这个属性设置成新 URL 将引发浏览器读取那个 URL 中的内容，并将这些内容显示出来。但是要注意，在解析当前文档的过程中，这个属性不会起作用。因此，不应该在顶层脚本中设置 `src` 属性，而应该在事件处理程序中或者由事件处理程序调用的函数中设置它。

`top`

一个可读可写的整数，指定了层 `layer` 相对于它的包容层或包容文档的 Y 坐标。设置这一属性可以上下移动层。它等价于属性 `y`。

`visibility`

一个可读可写的字符串，指定了层的可见性。这个属性的合法取值有三个，“`show`”将层指定为可见的，“`hide`”将层指定为不可见的，“`inherit`”将层指定为继承它的父层的可见性。

`window`

引用包容层的 Window 对象，无论该层在其他层中嵌套得有多深。

x, y

层相对于它的包容层或包容文档的 X 坐标和 Y 坐标。设置这一属性可以移动层。
x 等价于属性 left, y 等价于属性 top。

zIndex

声明层的 z 坐标, 或者说是层的叠放次序。当两个层交叠在一起时, zIndex 值较大的层就出现在 zIndex 值较小的层之上。如果两个兄弟层具有相同的 zIndex 值, 那么在包容文档的 layers[] 数组中位置越靠后的层就出现得越晚, 将覆盖掉它之前的层。

zIndex 属性是可读可写的。设置这一属性会改变层的叠放次序, 使它们以新的顺序重新显示出来。另外, 设置这一属性还会使包容文档的 layers[] 数组调整元素的顺序。

方法

captureEvents()	指定要捕捉的事件。
handleEvent()	将一个事件发送给适当的处理程序。
load()	装载一个新 URL, 并且调整层的大小。
moveAbove()	将层移动到另一个层之上。
moveBelow()	将层移动到另一个层之下。
moveBy()	将层移动一个相对的位置。
moveTo()	将层移动到一个相对于它的包容层的位置。
moveToAbsolute()	将层移动到一个相对于页面的位置。
offset()	等价于 moveBy()。
releaseEvents()	停止捕捉指定的事件类型。
resizeBy()	把层调整指定的数量。
resizeTo()	把层调整到指定的大小。
routeEvent()	把事件传递给下一个对它感兴趣的事件处理程序。

HTML 语法

用 Netscape 专有的 HTML 标记 <layer> 可以创建一个 Layer 对象:


```

<layer
  [ id = "layername" ]           // 层的名字
  [ left = 'x' ]                 // 相对于包容层的位置
  [ top = 'y' ]                  // 相对于顶层文档的位置
  [ pagex = 'x' ]                // 相对于顶层文档的位置
  [ pagey = 'y' ]
  [ width = "w" ]                // 层的大小
  [ height = "h" ]
  [ src = "url" ]                // 层内容的URL
  [ clip = "x, y, w, h" ]        // 层的剪切矩形
  [ clip = "w, h" ]              // 另一种语法, 默认 x 和 y 为 0
  [ zindex = 'z' ]               // 层的叠放顺序
  [ above = "layername" ]        // 另一种设置叠放顺序的语法
  [ below = "layername" ]
  [ visibility = 'vis' ]         // "show", "hide" 或 "inherit"
  [ bgcolor = "color" ]          // 层的背景颜色
  [ background = "url" ]         // 层的背景图像
  [ onmouseover = "handler" ]     // 当鼠标移入层的时候调用的事件处理程序
  [ onmouseout = "handler" ]      // 当鼠标移出层的时候调用的事件处理程序
  [ onfocus = "handler" ]        // 当层得到焦点的时候调用的事件处理程序
  [ onblur = "handler" ]          // 当层失去焦点的时候调用的事件处理程序
  [ onload = "handler" ]          // 当层的内容装载完毕之后调用的事件处理程序

```

描述

Layer 对象是 Netscape 4 支持的可动态定位 HTML 元素的技术。但要注意, Layer 对象从未被标准化, Netscape 6 也不再支持它。创建 Layer 对象的方法有三种, 它们分别是用标记 <layer> 创建、用 Layer() 构造函数创建以及使用 HTML 元素的 CSS 样式性质创建, 最后一种方式最具有可移植性。我们在第十八章中对此进行详细的介绍。

参阅

Window; 第十八章

Layer.captureEvents()

参阅 Window.captureEvents()

Layer.handleEvent()

参阅 Window.handleEvent()

Layer.load()只有 Netscape 4 可用

改变层的内容和宽度

摘要

```
layer.load(src, width)
```

参数*src*

一个字符串，指定了要装载进 *layer* 的文档的 URL。

width

一个整数，指定了 *layer* 的新宽度，以像素计。

描述

方法 `load()` 将把一个新的文档装载到图层 *layer* 中，并且指定了文档的行宽。

但是要注意，在解析当前文档的过程中 `load()` 不起作用。出于这种原因，不应该在顶层的脚本中调用这个 `load()` 方法，而应该在事件处理程序中或者由事件处理程序调用的函数中调用它。

参阅

`Layer.src`

Layer.moveAbove()只有 Netscape 4 可用

把一个层移动到另一个层之上

摘要

```
layer.moveAbove(target)
```

参数*target*

对 `Layer` 对象的引用，层 *layer* 将被移到这个层之上。

描述

`moveAbove()` 将改变层的叠放顺序，把层 *layer* 移动到层 *target* 之上。如果层 *layer* 不存在，那么它将成为层 *target* 的兄弟，具有和 *target* 相同的 `zIndex` 值，存放在包含文档的 `layers[]` 数组中的 *target* 之后。

Layer.moveBelow ()

只有 Netscape 4 可用

把一个层移动到另一个层之下

摘要

```
layer.moveBelow(target)
```

参数

target

对 Layer 对象的引用，层 *layer* 将被移到这个层之下。

描述

`moveBelow()` 将改变层的叠放顺序，把层 *layer* 移动到层 *target* 之下。如果层 *layer* 不存在，那么它将成为层 *target* 的兄弟，具有和 *target* 相同的 `zIndex` 值，存放在包容文档的 `layers[]` 数组中的 *target* 之前。

Layer.moveBy ()

只有 Netscape 4 可用

把层移动一个相对的位置

摘要

```
layer.moveBy(dx, dy)
```

参数

dx 把层右移的像素数（可以为负数）。

dy 把层下移的像素数（可以为负数）。

描述

方法 `moveBy()` 将把层从当前的位置右移 *dx* 个像素，下移 *dy* 个像素。

Layer.moveTo ()

只有 Netscape 4 可用

移动层

摘要

```
layer.moveTo(x, y)
```

参数

x 想把层移动到的位置的 X 坐标。

y 想把层移动到的位置的 Y 坐标。

描述

方法 `moveTo()` 将把 `layer` 的左上角移动到 *x* 和 *y* 指定的坐标处。注意，这两个坐标相对于该层的包容层或包容窗口。

Layer.moveToAbsolute()

只有 Netscape 4 可用

把该层移动到相对于页面的坐标处

摘要

```
layer.moveToAbsolute(x, y)
```

参数

x 想把层移动到的位置的 X 坐标。

y 想把层移动到的位置的 Y 坐标。

描述

方法 `moveToAbsolute()` 将把 `layer` 的左上角移动到 *x* 和 *y* 指定的文档坐标处。注意，这两个坐标相对于页面或顶层文档，而不是相对于任何包容层。

Layer.offset()

只有 Netscape 4 可用;反对使用

把该层移动一个相对的位置

摘要

```
layer.offset(dx, dy)
```

参数

dx 把层右移的像素数 (可以为负数)。

dy 把层下移的像素数 (可以为负数)。

描述

方法 `offset()` 将相对于当前的位置移动层。它被反对使用，推荐使用的方法是 `moveBy()`，反对使用 `offset()`。

Layer.releaseEvents()

参阅 `Window.releaseEvents()`

Layer.resizeBy()

只有 Netscape 4 可用

以相对的数量调整层

摘要

```
layer.resizeBy(dw, dh)
```

参数

`dw` 要将窗口的宽度加大的像素数（可以为负数）。

`dh` 要将窗口的高度加大的像素数（可以为负数）。

描述

方法 `resizeBy()` 将调整 `layer` 的大小，把它的 `clip.width` 属性加大 `dw` 个像素，把 `clip.height` 属性加大 `dh` 个像素。由于这样不会改变层中的内容的格式，所以缩小层会使层的部分内容被剪切掉。

Layer.resizeTo()

只有 Netscape 4 可用

调整层大小

摘要

```
layer.resizeTo(width, height)
```

参数

`width`

想要调整到的层宽度。

`height`

想要调整到的层高度。

描述

方法 `resizeTo()` 将调整层的大小，把它的 `clip.width` 属性设置为 `width`，把 `clip.height` 属性设置为 `height`。由于这样并不会改变层中的内容的格式，所以缩小层会使层的部分内容被剪切掉。

Layer.routeEvent()

参阅 `Window.routeEvent()`

Link

JavaScript 1.0;在 JavaScript 1.1 中增强了它

超文本链接

继承 `HTMLElement`

摘要

`document.links[]`

`document.links.length`

属性

`Link` 对象继承了 `HTMLElement` 对象的属性，此外还定义了如下一些属性。许多属性都表示 URL 的一部分。在接下来的每条属性说明中，给出的例子都是下面这条（虚拟）URL 的一部分：

`http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results`

`hash`

一个可读可写的字符串属性，指定了 `Link` 对象的 URL 中的锚部分，包括前导散列符（#）。例如，“#result”。URL 的锚部分只链接在文档中引用的位置。在 HTML 中，这个位置由标记 `` 创建的锚命名。

`host`

一个可读可写的字符串，指定了 `Link` 的 URL 中的主机名和端口部分。例如，`www.oreilly.com:1234`。

`hostname`

一个可读可写的字符串，指定了 `Link` 的 URL 中的主机名部分。例如，“`www.oreilly.com`”。

href

一个可读可写的字符串，指定了 Link 的完整 URL，不同于其他的 Link URL 属性，声明的只是部分 URL。

pathname

一个可读可写的字符串，声明了 Link 的 URL 的路径部分。例如，`“/catalog/search.html”`。

port

一个可读可写的字符串（不是数字），声明了 Link 的 URL 的端口部分。例如，`“1234”`。

protocol

一个可读可写的字符串，声明了 Link 的 URL 的协议部分，包括后缀冒号。例如，`“http:”`。

search

一个可读可写的字符串，它声明了 Link 的 URL 的查询部分，其中包括前导问号。例如，`“?q=JavaScript&m=10”`。

target

一个可读可写的字符串，指定了显示链接文档的 Window 对象（如一个框架或顶层的浏览器窗口）的名称。参阅“`Link.target`”参考页。

text[Netscape 4]

声明了出现在链接的标记 `<a>` 和 `` 之间的纯文本。注意，只有当标记 `<a>` 和 `` 之间不再有其他 HTML 标记时，这个属性才能正确作用。如果还有其他 HTML 标记，那么 `text` 属性存放的只是部分链接文本。这个属性是 Netscape 专有的，IE 4 中与之等价的属性是 `HTMLElement.innerText`。

方法

Link 对象继承了 HTMLElement 对象的方法。

事件处理程序

Link 对象继承了 HTMLElement 对象的事件处理程序，此外还定义了如下的三种特殊行为：

onClick

当用户点击链接时调用的事件处理程序。在 JavaScript 1.1 中，可以通过使该处理程序返回 false 阻止链接发生。在 Netscape 3 的 Windows 平台上，对于由标记 <area> 创建的链接，这个事件处理程序不起作用。

onmouseout

当用户把鼠标移出链接时调用的事件处理程序。在 JavaScript 1.1 和其后的版本中可用。

onmouseover

当用户把鼠标移动到链接上时调用的事件处理程序。在此可以设置当前窗口的 status 属性。如果要通知浏览器不要显示链接对象的 URL，只需使该处理程序返回 true 即可。

HTML 语法

Link 对象是由标准的 HTML 标记 <a> 和 创建的。href 性质对所有的 Link 对象都是必需的。如果还设置了该对象的 name 性质，那么还会创建一个 Anchor 对象：

```
<a href = "url"           // 链接的目的文件
  [ name = "anchor_tag" ] // 创建一个 Anchor 对象
  [ target = "window_name" // 显示新文档的窗口
  [ onclick = "handler" ] // 链接被点击时调用的事件处理程序
  [ onmouseover = "handler" ] // 当鼠标移动到链接上时调用的事件处理程序
  [ onmouseout = "handler" ] // 当鼠标离开链接时调用的事件处理程序
>
link text or image           // 链接中的可见部分
</a>
```

在 JavaScript 1.1 和其后的版本中，还可以用标记 <area> 在客户端的图像映射中创建 Link 对象。这里采用的也是标准的 HTML 语法：

```
<map name = "map_name">
  <area shape = "area_shape"
    coords = "coordinates"
    href = "url"           // 链接的目的文件
    [ target = "window_name" ] // 显示新文档的窗口
    [ onclick = "handler" ] // 区域被点击时调用的事件处理程序
    [ onmouseover = "handler" ] // 当鼠标移动到区域上时调用的事件处理程序
    [ onmouseout = "handler" ] // 当鼠标离开区域时调用的事件处理程序
  >
  ...
</map>
```


描述

Link 对象表示 HTML 文档中的超文本连接或客户端映射表中的可点击区域。所有由标记 `<a>` 创建的（在 Netscape 3 中还有由标记 `<area>` 创建的）链接都用 Link 对象表示，而且都存放在 Document 对象的 `links[]` 数组中。注意，无论是由标记 `<a>` 创建的链接，还是由标记 `<area>` 创建的链接，都存放在同一个数组中，它们之间没有区别。

超文本链接的目的文件由 URL 指定，Link 对象的许多属性都用来设置 URL 的内容。在具有全套 URL 属性这一点上，Link 对象与 Location 对象相似。在 Location 对象中，这些属性描述的是当前显示的文档的 URL。

除了那些属性之外，Link 对象还有三个事件处理程序。onmouseover() 设置的是在鼠标移动到超文本链接上要执行的代码，onclick() 设置的是在点击了它时要执行的代码，onmouseout() 设置的是鼠标离开了屏幕上的链接区域时要执行的代码。

参阅

Anchor、Location;DOM 参考千册部分的 HTMLAnchorElement

Link.onclick	JavaScript 1.0;在 JavaScript 1.1 中增强了它
<p>单击链接时调用的事件处理程序</p>	

摘要

```
<a ... onclick="handler" ... >
<area ... onclick='handler' ... >
link.onclick
```

描述

Link 对象的 onclick 属性声明了一个事件处理函数，当用户点击了该链接时，它就会被调用。这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义该 Link 对象的 HTML 标记 `<a>` 或 `<area>` 的 onclick 性质指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 element 的作用域中执行，而不是在包容它的窗口的作用域中执行。

onclick是在浏览器转到链接指定的页面之前调用的。这样就可以对href、target和其他的链接属性动态地进行设置(使用关键字this或引用被点击的链接)。在这个

处理函数中，还可以调用方法 `Window.alert()`、`Window.confirm()` 和 `Window.prompt()`。

在 JavaScript 1.1 中，可以通过使这个处理函数返回 `false` 来阻止浏览器进入链接所指的页面。如果它返回 `true`、其他值，或者什么都不返回，那么当 `onclick` 返回时，浏览器就会前进到链接所指的文档。还可以使用方法 `Window.confirm()` 来询问用户是否真地想进入链接所指的页面，如果用户点击了 **Cancel** 按钮，就阻止浏览器进行链接。一般说来，如果想通过链接执行某个动作，但又不想让新的 URL 显示出来，最好使用 `Button` 对象的 `onclick` 事件处理函数，而不是使用 `Link` 对象的 `onclick` 处理函数。

注意，虽然 `onclick` 处理函数通知浏览器不要执行默认的动作（沿着链接前进）时返回的是 `false`，但是 `onmouseover` 处理函数通知浏览器不要执行默认的动作（显示链接的 URL）时返回的却是 `true`。是历史原因造成了这种不兼容性。对于 `Form` 对象和表单元素的事件处理函数来说，返回值为 `false` 则阻止浏览器执行默认的动作。

在 Netscape 4 事件模型中，`Event` 对象作为参数传递给 `onclick` 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 `Event` 对象作为那个超文本链接的 `Window` 对象的 `event` 属性出现。

Bug

在 Windows 平台上的 Netscape 3 中，标记 `<area>` 的 `onclick` 事件处理函数是无效的。避开这个 bug 的方法是把 `<area>` 标记的 `href` 性质设置成一个 `javascript:URL`。

参阅

第十九章；DOM 参考手册部分的 `EventListener`、`EventTarget`、`MouseEvent`

Link.onmouseout

JavaScript 1.1

当鼠标离开该链接时调用的处理函数

摘要

```
<a ... onmouseout="handler" ... >
<area ... onmouseout="handler" ... >
link.onmouseout
```

描述

Link对象的onmouseout属性声明了一个事件处理函数,当用户把鼠标从该链接上移开时,它就会被调用。这个属性的初始值是一个含有JavaScript语句的函数,其中的JavaScript语句由定义Link对象的标记<a>或<area>的onmouseout性质指定。如果一个事件处理函数是由HTML性质定义的,那么它就会在element的作用域中执行,而不是在包容它的窗口的作用域中执行。

在Netscape事件模型4中,Event对象作为参数传递给onmouseout处理函数。但是在IE事件模型中并不给它传递任何参数,可应用的Event对象作为那个超文本链接的Window对象的事件属性出现。

参阅

第十九章; DOM 参考手册部分的EventListener、EventTarget、MouseEvent

Link.onmouseover

JavaScript 1.0

当鼠标经过该链接时调用的处理函数

摘要

```
<a ... onmouseover="handler" ... >
<area ... onmouseover="handler" ... >
link.onmouseover
```

描述

Link对象的onmouseover属性声明了一个事件处理函数,当用户把鼠标移动到该链接上时,它就会被调用。这个属性的初始值是一个含有JavaScript语句的函数,其中的JavaScript语句由定义Link对象的HTML标记<a>或<area>的onmouseover属性指定。如果一个事件处理函数是由HTML性质定义的,那么它就会在element的作用域中执行,而不是在包容它的窗口的作用域中执行。

在默认情况下,当鼠标移动到链接上时,浏览器就会在状态栏中显示出它的URL。事件处理函数onmouseover是在URL显示之前调用的。如果它返回的是true,浏览器就不显示这个URL。因此,返回值为true的事件处理函数onmouseover就可以在状态栏中显示出自定义的信息了,这只需要将属性Window.status设置成想要使用的值即可。

注意，虽然 `onmouseover` 处理函数通知浏览器不要执行默认的动作（显示链接的 URL）时返回的是 `true`，但是 `onclick` 事件处理函数通知浏览器不要执行默认的动作（沿着链接前进）时返回的却是 `false`。是历史原因造成了这种不兼容性。对于 Form 对象和表单元的事件处理函数，标准是返回值为 `false` 则阻止浏览器执行默认的动作。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 `onmouseover` 处理函数。但是在 IE 事件模型中并不给它传递任何参数，可应用的 Event 对象作为那个超文本链接的 Window 对象的 `event` 属性出现。

参阅

第十九章；DOM 参考手册部分的 `EventListener`、`EventTarget`、`MouseEvent`

Link.target

JavaScript 1.0

超文本链接的目标窗口

摘要

`link.target`

描述

Link 对象的属性 `target` 是一个可读可写的字符串，指定了一个框架或窗口的名字，Link 对象所引用的 URL 应该在这个框架或窗口中显示。这个属性的初始值是由创建该 Link 对象的标记 `<a>` 的 `target` 性质设置的。如果没有设置这个性质，那么它的默认值就是包容该链接的窗口，因此沿着这个链接前进就会覆盖含有它的那个文档。

注意，`target` 的值是框架或窗口的名字，而不是对框架或窗口自身的引用。框架的名字是由 `<frame>` 标记的 `name` 性质设置的。窗口的名字是在创建窗口时调用方法 `Window.open()` 设置的。如果 `target` 指定的窗口名不存在，浏览器会自动打开一个新窗口来显示 URL，而且此后任何具有相同的 `target` 性质的链接都会使用这个新创建的窗口。

JavaScript 支持四种特殊的 `target` 名字。“`_blank`”声明的是应该创建一个新的空浏览器窗口，用它来显示新的 URL。“`_self`”是默认的 `target` 值，它声明的是在含有链接的框架或窗口中显示的新 URL。“`_parent`”声明的是应该在含有链接的框架的父框架中显示的结果。最后一个为“`_top`”，它声明的是应该在顶层框架中显示的新的 URL，也就是说，应该将所有的框架都删除，让新的 URL 占据整个浏览器窗口。

参阅

Form.target

Location JavaScript 1.0;在 JavaScript 1.1 中增强了它
表示并控制浏览器的位置

摘要

location

window.location

属性

Location 对象的属性引用了当前文档的 URL 的各个部分。在接下来的每条属性说明中，给出的例子都是下面这条（虚拟）URL 的一部分：

```
https://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

hash

一个可读可写的字符串，指定了当前 URL 中的锚部分，包括前导散列符（#）。例如，“#result”。文档 URL 的这一部分指定了锚在文档中的名字。

host

一个可读可写的字符串，是当前 URL 中的主机名和端口部分。例如，“www.oreilly.com:1234”。

hostname

一个可读可写的字符串，声明了当前 URL 中的主机名部分。例如，“www.oreilly.com”。

href

一个可读可写的字符串，它声明了当前显示的文档的完整 URL，与其他 Location 属性只声明部分 URL 不同。把该属性设置为新的 URL 会使浏览器读取并显示新 URL 的内容。

pathname

一个可读可写的字符串，声明了当前 URL 的路径部分。例如，“/catalog/search.html”。

port

一个可读可写的字符串（不是数字），声明了当前 URL 的端口部分。例如，“1234”。

protocol

一个可读可写的字符串，声明了 URL 的协议部分，包括后缀的冒号。例如，“http:”。

search

一个可读可写的字符串，声明了当前 URL 的查询部分，包括前导问号。例如，“?q=JavaScript&m=10”。

方法

reload()

从缓存或服务器中再次把当前文档装载进来。该方法是在 JavaScript 1.1 中添加的。

replace()

用一个新文档替换当前文档，而不用在浏览器的会话历史中生成一个新的记录。该方法是在 JavaScript 1.1 中添加的。

描述

Location 对象存储在 Window 对象的 location 属性中，表示那个窗口中当前显示的文档的 Web 地址。它的 href 属性存放的是文档的完整 URL，其他的属性则分别描述了 URL 的各个部分。这些属性与 Link 对象的 URL 属性非常相似。

不过 Link 对象表示的是文档中的超链接，Location 对象表示的却是浏览器当前显示的文档的 URL（或位置）。但是 Location 对象所能做的远远不止这些，它还能控制浏览器显示的文档的位置。如果把一个含有 URL 的字符串赋给 Location 对象或它的 href 属性，浏览器就会把新的 URL 所指的文档装载进来，并显示出来。

除了设置 location 或 location.href，用完整的 URL 替换当前的 URL 之外，还可以修改部分的 URL，只需要给 Location 对象的其他属性赋值即可。这样做会创建一个新的 URL，其中的一部分与原来的 URL 不同，浏览器会将它装载并显示出来。例如，假设设置了 Location 对象的 hash 属性，那么浏览器就会转移到当前文档中的一个指定位置。同样，如果设置了 search 属性，那么浏览器就会重新装载附加了新的

查询字符串的 URL。如果 URL 引用的是一个服务器端的脚本，那么附加了新的查询字符串后产生的结果就可能与原始文档大不相同了。

除了 URL 属性外，Location 对象还定义了两个方法。reload() 可以重新装载当前文档，replace() 可以装载一个新文档而无须为它创建一个新的历史记录，也就是说，在浏览器的历史列表中，新文档将替换当前文档。

参阅

Link、Window.location

Location.reload()

JavaScript 1.1

重新装载当前文档

摘要

```
location.reload()
location.reload(force)
```

参数

force

一个布尔参数，声明了是否应该重新装载当前文档，即使服务器的报告表明自从上次装载完毕之后它还没有被修改过。如果省略了这个参数，或者它的值为 false，那么只有当从上次装载完毕之后改变文档时，该方法才会重新装载整个页面。

描述

Location 对象的方法 reload() 可以把它所在的窗口正在显示的文档重新装载一遍。如果调用时没有给它传递参数，或者传递的参数是 false，它就会用 HTTP 头 If-Modified-Since 来检测服务器上的文档是否改变了。如果文档已经改变了，该方法会把它从服务器上再次下载下来。如果文档没有改变，该方法将从缓存中把它装载进来。这与用户点击了浏览器的 **Reload** 按钮时发生的动作完全相同。

如果调用方法 reload() 时传递给它的参数是 true，那么无论文档的最后修改日期是什么，它都将绕过缓存，从服务器上重新下载该文档。这与用户在点击浏览器的 **Reload** 按钮时按住 shift 键发生的动作完全相同。

Location.replace()

JavaScript 1.1

用另一个文档替换当前显示的文档

摘要

```
location.replace(url)
```

参数

url

一个字符串，指定了要替换当前文档的新文档的 URL。

描述

Location 对象的方法 `replace()` 将下载并显示一个新文档。用这种方式装载文档与只设置属性 `location` 或 `location.href` 有一点重要的不同之处，即 `replace()` 方法不在 History 对象中生成一个新的记录。当使用 `replace()` 方法时，新的 URL 就会覆盖 History 对象中的当前记录。也就是说，在调用 `replace()` 方法之后，点击浏览器的 **Back** 按钮就不会返回到之前浏览的那个文档，而直接返回到那个文档之前的 URL。

习惯用法

当使用多框架和（或）JavaScript 生成的文档时，可能得到一个临时文档。如果存在多个这样的文档，那么使用户用 **Back** 按钮退出站点就会变得很麻烦。不过用 `replace()` 来装载这些文档，就可以避免发生这种问题。

参阅

History

MimeType

Netscape 3

代表 MIME 数据类型

摘要

```
navigator.mimeTypes[i]  
navigator.mimeTypes["type"]  
navigator.mimeTypes.length
```


属性

description

一个只读的字符串，提供了由MimeType对象描述的数据类型的英文描述，是人们可以读懂的，比属性name更加明确，也更易于理解。

enabledPlugin

对已安装的Plugin对象的引用，并启用了处理指定类型的插件。如果没有插件处理这种MIME类型，那么这个属性的值就是null。

数组navigator.mimeTypes[]可以告诉你浏览器是否以某种方式支持给定的MIME类型。enabledPlugin属性则可以告诉你一个给定的类型是否是由插件支持的（因为帮助程序也支持MIME类型，或者由浏览器直接支持它）。如果一种MIME类型是由插件支持的，那么这种数据类型就能够用标记<embed>嵌入到网页中。

suffixes

一个只读字符串，它存放的是一个文件名后缀（不包括字符“.”）的列表，各后缀之间用逗号分开。其中的文件名后缀常用于指定的MIME类型文件。例如，MIME类型text/html的后缀是“html, htm”。

type

一个只读的字符串，声明了MIME类型的名称。这种字符串是惟一的，如“text/html”或“image/jpeg”，它将各种MIME类型区别开了。此外，它还描述了数据的一般类型以及使用的数据格式。这个属性的值还可以用做访问数组navigator.mimeTypes[]元素的下标。

描述

MimeType对象表示Netscape支持的MIME类型（即数据格式）。这种格式可以是浏览器直接支持的，也可以是外部的帮助程序或插件支持的。

习惯用法

引用数组navigator.mimeTypes[]中的元素时，既可以使用该元素的下标，又可以使用MIME类型的名称（即属性type的值）。要检测Netscape支持的MIME类型，可以遍历数组navigator.mimeTypes[]。如果想检测Netscape是否支持某种指定的类型，可以使用如下的代码：

```
var show_movie = (navigator.mimeTypes['video/mpeg'] != null);
```

参阅

Navigator、Plugin

Navigator JavaScript 1.0;在 JavaScript 1.1 和 1.2 中增强了它正在使用的浏览器的信息

摘要

navigator

属性

appName

一个只读字符串，声明了浏览器的代码名。在所有 Netscape 浏览器中，它的值是“Mozilla”。为了兼容起见，在 Microsoft 的浏览器中，它的值也是“Mozilla”。

appName

一个只读字符串，声明了浏览器的名字。在 Netscape 中，这个属性的值是“Netscape”。在 IE 中，这个属性的值是“Microsoft Internet Explorer”。

appVersion

一个只读的字符串，声明了浏览器的平台和版本信息。这个字符串的第一部分是版本号。把该字符串传递给 `parseInt()` 只能获取主版本号，传递给 `parseFloat()` 可以以浮点值的形式获得主版本号和副版本号。该属性的其余部分提供了有关浏览器版本的其他细节，包括运行它的操作系统的信息。但是，不同浏览器提供的信息的格式不同。

cookieEnabled[IE 4, Netscape 6]

一个只读布尔值，如果浏览器启用了 cookie，该属性值为 `true`。如果禁用了 cookie，该属性值为 `false`。

language[Netscape 4]

一个只读字符串，声明了浏览器的版本使用的默认语言。这个属性的值是两个字母的标准语言代码，如“en”代表英语，“fr”代表法语。它还可以是具有五个字母的字符串，说明语言和它的地域，如“fr_CA”代表在加拿大讲的法语。注意，IE 4 不支持这一属性，不过支持另外两个与语言相关的 Navigator 属性。

mimeTypes[][Netscape 3]

一个 `MimeType` 对象的数组，其中的每个元素代表浏览器支持的某种 MIME 类

型 (如 “text/html” 和 “image/gif”)。虽然 `mimeType` 数组是由 IE 4 定义的, 但是在 IE 4 中它却总是空的, 因为 IE 4 不支持 `MimeType` 对象。

`platform[JavaScript 1.2]`

一个只读字符串, 声明了运行浏览器的操作系统和 (或) 硬件平台。虽然该属性没有标准的值集合, 但它有些常用值, 如 “Win32”、“MacPPC”、“Linux586” 等等。

`plugins[][Netscape 3]`

一个 `Plugin` 对象的数组, 其中的元素代表浏览器已经安装的插件。`Plugin` 对象提供的是有关插件的信息, 其中包括它所支持的 MIME 类型的列表。所谓插件 (plugin), 就是浏览器用来在浏览器窗口中显示特定数据类型的软件包的 Netscape 名称。

虽然 `plugins` 数组是由 IE 4 定义的, 但是在 IE 4 中它却总是空的, 因为 IE 4 不支持插件和 `Plugin` 对象。

`systemLanguage[IE 4]`

一个只读字符串, 指定了操作系统使用的默认语言, 使用的标准代码与 Netscape 特有的 `language` 属性使用的代码相同。

`userAgent`

一个只读的字符串, 声明了浏览器用于 HTTP 请求的用户代理头的值。一般说来, 它是在 `navigator.appCodeName` 的值之后加上斜线和 `navigator.appVersion` 的值构成的。例如:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows 95)
```

`userLanguage [IE 4]`

一个只读的字符串, 声明了用户想使用的语言。使用的标准代码与 Netscape 特有的 `language` 属性使用的代码相同。

函数

`navigator.javaEnabled()`

检测当前的浏览器是否支持并启用了 Java。它是由 JavaScript 1.1 添加进来的。

`navigator.plugins.refresh()`

检测一个新安装的插件, 把它插入数组 `plugins` 中, 并且使用这些插件有选择地重新装载文档。它是由 Netscape 3 添加进来的。

描述

Navigator 对象包含的属性描述了正在使用的浏览器。可以使用这些属性进行平台专用的配置。虽然这个对象的名字显而易见指的是 Netscape 的 Navigator 浏览器，但其他实现了 JavaScript 的浏览器也支持这个对象。

Navigator 对象的实例是唯一的，可以用 Window 对象的 navigator 属性来引用它。由于隐式地引用了窗口，所以可以直接使用 navigator 来引用 Navigator 对象。

参阅

MimeType、Plugin

Navigator.javaEnabled()

JavaScript 1.1

检测 Java 是否可用

摘要

navigator.javaEnabled()

返回值

如果当前浏览器支持 Java，并启用了它，则返回 true，否则返回 false。

描述

可以使用 navigator.javaEnabled() 来检测当前浏览器是否支持 Java，从而知道浏览器是否能显示 Java 小程序。

Navigator.plugins.refresh()

Netscape 3

启用新安装的插件

摘要

navigator.plugins.refresh([reload])

参数

reload

一个可选的布尔参数，如果值为 true，说明方法 refresh() 应该重新装载含有标记 <embed> 的页面，并且使用插件将嵌入的数据显示出来。如果省略了该参数，它的默认值为 false。

描述

方法 `refresh()` 将使 Netscape 检测是否已经安装了某种新插件。如果已经安装了新的插件, 那么 `plugins[]` 数组就会被更新, 以添加新安装的插件。如果参数 `reload` 的值为 `true`, Netscape 还会重新装载当前显示的、含有 `<embed>` 标记的文档, 使用插件将嵌入的数据显示出来。

注意, 这个方法的摘要有些特殊, `refresh()` 是 `plugins[]` 数组的方法, 不是 `Navigator` 对象的方法。不过, 在大多数情况下, 将它看作 `Navigator` 对象的方法比较简单, 这也是将它归为 `Navigator` 的方法的原因。

Option

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

Select 框中的一个选项

继承 `HTMLElement`

摘要

`select.options[i]`

构造函数

在 JavaScript 1.1 中, 可以使用 `Option()` 构造函数动态地创建 `Option` 对象:

```
new Option(text, value, defaultSelected, selected)
```

参数

`text`

一个可选的字符串参数, 指定了 `Option` 对象的 `text` 属性。

`value`

一个可选的字符串参数, 指定了 `Option` 对象的 `value` 属性。

`defaultSelected`

一个可选的布尔参数, 指定了 `Option` 对象的 `defaultSelected` 属性。

`selected`

一个可选的布尔参数, 指定了 `Option` 对象的 `selected` 属性。

属性

`Option` 对象继承了 `HTMLElement` 对象的属性, 此外还定义了如下一些属性:

defaultSelected

一个布尔值，声明了在创建包含该属性的 Select 对象时，该选项是否被选中。在包含 Select 对象的表单重置时，使用这个值恢复 Select 对象的初始状态。该属性的初始值由 `<option>` 标记的 `selected` 性质设置。

index

一个只读的整数，声明了选项在包含它的 Select 对象的 `options[]` 数组中的位置或下标。该数组中第一个 Option 对象的下标是 0，它的 `index` 属性就设置成 0。第二个 Option 对象的 `index` 属性值为 1，以此类推。

selected

一个可读可写的布尔值，声明了一个选项当前是否被选中了。可以使用该属性来检测一个给定的选项是否被选中了。还可以用它来选中一个选项（把它设置为 `true`）或取消对一个选项的选择（把它设置为 `false`）。注意，以这种方式选中一个选项或取消对一个选项的选择时，并不调用事件处理程序 `Select.onchange()`。

text

一个字符串，声明了显示给用户的选项文本。这个属性的初始值是出现在 `<option>` 标记之后，下一个 `<option>` 标记、`</option>` 标记或 `</select>` 标记之前的纯文本（而不使用 HTML 标记）。

在 JavaScript 1.0 中，`text` 属性是只读的。在 JavaScript 1.1 中，它是可读可写的。通过设置这一属性的新值，可以改变 Select 对象中的选项文本。注意，如果要在不能回流文档内容的浏览器中设置该属性，就要确保改变选项标签不会使 Select 对象加宽，如果一定要加宽 Select 对象，就要确保 Select 对象右侧的信息不会变得模糊不清。

value

一个可读可写的字符串，声明了如果在提交表单时 `option` 处于被选中的状态，要传递给服务器的文本。`value` 的初始值由 `<option>` 标记的 `value` 性质设置。如果表单是要提交给服务器的（与只在客户端 JavaScript 中使用的服务器相反），那么 Select 对象中的每个 Option 对象都要有惟一的 `value` 值。

HTML 语法

Option 对象是由 `<select>` 标记中的 `<option>` 标记创建的，其中 `<select>` 标记位于 `<form>` 标记中。一般说来，可以在 `<select>` 标记中放多个 `<option>` 标记。

```

<FORM...>
  <SELECT...>
    <option
      [ value = "value"           // 在提交表单时返回的值
      [ selected = "checked"     // 声明初始化时这个选项是否被选中了
      plain_text_label           // 该选项要显示的文本
    ] </option> .
    ...
  </select>
  ...
</FORM>

```

描述

Option对象描述的是Select对象中显示的一个选项。该对象的属性声明了在默认情况下它是否被选中了，当前它是否被选中了，它在包容的Select对象的options[]数组中的位置，它要显示的文本以及如果在提交表单时它处于选中的状态，要传递给服务器的值。

注意，虽然选项显示的文本是在标记<option>的外部设置的，但它必须是纯文本，即不能具有任何HTML标记的非格式化，这样它才能在不支持HTML格式的列表框或下拉式菜单中正确地显示出来。

在JavaScript 1.1中，可以用构造函数Option()动态地创建一个新Option对象。一旦创建了新的Option对象，它就会被添加到Select对象s的选项列表中，只需要把它赋给s.options[option.length]即可。

要了解详情，请参阅“Select.options[]”参考页。

参阅

Select、Select.options[]; DOM 参考手册部分的HTMLOptionElement、HTMLSelectElement

Password

JavaScript 1.0;在JavaScript 1.1中增强了它

用于敏感数据的文本输入框

继承 Input, HTMLElement

摘要

form.name

form.elements[i]

属性

Password 对象继承了 Input 对象和 HTMLElement 对象的属性，此外还定义或覆盖了如下的属性：

value

一个可读可写的字符串，声明了用户输入的口令。在提交表单时，将把它发送到 Net。这个属性的初始值由定义 Password 对象的元素 <input> 的 value 性质设置。注意，由于口令的敏感性，所以安全限制规约对 value 属性进行了保护。在某些浏览器中，查询该属性与用户输入的文本不匹配时，或设置该属性既不会影响显示的值，也不会影响表单提交的值时返回该字符串。

方法

Password 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Password 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序。

HTML 语法

Password 元素是由标准的 HTML 标记 <input> 创建的：

```
<form>
...
  <input
    type = 'password'           // 声明这是一个 Password 对象
    [ name = 'name' ]          // 此后可以用来引用该元素的名字
                                // 指定 name 属性
    [ value = "default" ]       // 在提交表单时要发送的默认值
    [ size = "integer" ]        // 该元素的宽度是多少个字符
  >
...
</form>
```

描述

元素 Password 是一个专门输入敏感数据（如口令）的文本框。由于在用户输入字符时只显示星号，所以旁观者就不能偷窥用户的输入数据。作为进一步的安全预防措施，对于 JavaScript 如何读写 Password 元素的 value 属性值有一定限制。要了解更多信息，参阅“Text”和“Input”参考页。

参阅

Input、Text；DOM 参考手册部分的 HTMLInputElement

Plugin

Netscape 3

描述已安装的插件

摘要

`navigator.plugins[i]``navigator.plugins['name']`

属性

`description`

一个只读字符串, 包含人们可以读懂的插件说明。该说明文本由插件的创建者提供, 包括厂商的信息、版本信息和插件功能的简短说明。

`filename`

一个只读字符串, 声明了硬盘上存放插件程序的文件名。不同平台采用的名字不同。对于标识插件, `name` 属性比 `filename` 属性更有用。

`length`

每个 `Plugin` 对象都含有 `MimeType` 数组元素, 其中的元素声明了该插件支持的数据格式。和其他所有的数组一样, 属性 `length` 声明了数组中的元素个数。

`name`

一个只读字符串, 声明了插件的名字。每个插件都应该有惟一标识它的名字。插件的名字可以用作数组 `navigator.plugins[]` 的下标。利用这一事实, 可以轻松判断出当前浏览器是否安装了指定的插件:

```
var sw_installed = (navigator.plugins['Shockwave'] != null);
```

元素

`Plugin` 对象的数组元素是 `MimeType` 对象, 它们声明了插件支持的数据格式。

描述

所谓插件, 就是一个软件模块, Netscape 可以调用这个模块显示嵌入浏览器窗口的特殊数据类型。在 Netscape 3 中, 插件由 `Plugin` 对象表示。这个对象有些特殊, 因为它既含有常规的对象属性, 又含有数组元素。`Plugin` 对象的属性提供了有关插件的各种信息, 它的数组元素是 `MimeType` 对象, 该对象声明了插件支持的数据格式。

`Plugin` 对象存放在 `Navigator` 对象的 `plugins[]` 数组中。当你想遍历已安装插件的列表, 查找符合要求的插件时 (例如, 查找一个支持你想嵌入网页的数据的 `MIME` 类型

的插件), 可以以数字作为数组 `navigator.plugins[]` 的下标。不过, 如果想检测在用户的浏览器中是否安装了指定的插件, 还可以使用插件名作为数组 `navigator.plugins[]` 的下标, 使用的代码如下:

```
document.write( navigator.plugins( "Shockwave" ) ?  
    "<embed src='movie.swf' height=100 width=100.>" :  
    "You don't have the Shockwave plugin!" );
```

在这一方法中, 用作数组下标的插件名与 `Plugin` 对象的属性 `name` 的值完全相同。

`Plugin` 对象存放在 `Navigator` 对象的数组中, 而 `Plugin` 对象自身是 `MimeType` 对象的数组, 不要混淆了它们。因为其中涉及两个数组, 所以可能会得到如下代码:

```
navigator.plugins[i][j] // 第i个插件的第j个 MIME 类型  
navigator.plugins["LiveAudio"][0] // LiveAudio 插件的第一个 MIME 类型
```

最后要注意的是, 不仅用 `Plugin` 对象的数组元素可以声明插件支持的 `MIME` 类型, 使用 `MimeType` 对象的 `enabledPlugin` 属性也可以判断插件是否支持给定的 `MIME` 类型。

参阅

`Navigator`、`MimeType`

Radio

JavaScript 1.0;在 JavaScript 1.1 中增强了它

图形化的单选按钮

继承 `Input`, `HTMLElement`

摘要

`Radio` 按钮元素通常在一个选项组中使用, 其中的选项互斥, 具有相同的名字。要引用一个分组中的 `Radio` 元素, 可以使用如下的语法:

```
form.radio_name[j]  
form.radio_name.length
```

属性

`Radio` 对象继承了 `Input` 对象和 `HTMLElement` 对象的属性, 此外还定义或覆盖了如下一些属性:

`checked`

一个可读可写的布尔值。如果单选按钮被选中了, 它的 `checked` 属性值就为 `true`。

否则为 false。如果把 checked 属性设置成 true, 单选钮就被选中, 以前选中的单选钮将被取消选中。但要注意, 把一个单选钮的 checked 属性设为 false 不会产生任何效果, 因为至少要有一个按钮必须处于选中状态, 如果没有选择其他按钮, 那么就不能取消对一个单选按钮的选定。还要注意, 对 checked 属性的设置不会调用 Radio 按钮元素的事件处理程序 onclick。如果想调用那个事件处理程序, 就必须明确地做到这一点。

defaultChecked

一个布尔值。如果在初始情况下该按钮处于被选中的状态 (但只能在定义 Radio 元素的 HTML 标记 <input> 中出现了 checked 性质), 那么它的值为 true。如果该标记没有出现, 那么在初始情况下, 该单选按钮没有被选中, 属性 defaultChecked 的值是 false。

value

一个可读可写的字符串, 声明了一段文本。如果在提交表单时该单选按钮处于被选中的状态, 那么这段文本就会被传递给服务器。value 属性的初始值由按钮的 HTML 标记 <input> 的 value 性质设置。如果表单是要提交给服务器的 (与在客户端由 JavaScript 程序使用的服务器相反), 那么选项组中的每一个单选按钮都要有惟一的 value 值。注意, 属性 value 声明的不是当前该单选按钮是否被选中了, Radio 对象的当前状态由属性 checked 声明。

方法

Radio 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Radio 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序。此外还定义或覆盖了如下的事件处理程序:

onclick

在单选钮被点击时调用的事件处理程序。

HTML 语法

Radio 元素是由标准的 HTML 标记 <input> 创建的。通过设置多个具有相同 name 性质的 <input> 标记, 可以创建一个 Radio 元素组。

```
<form>  
...
```

```

<input
  type = 'radio'                // 声明这是一个单选按钮

  [ name = "name" ]             // 此后可用于引用该按钮的名字
                                // 也可以用它来引用该按钮的分组
                                // 指定 name 属性
  [ value = value' ]           // 在选中了按钮时的返回值
                                // 指定了 value 属性
  [ checked = 'checked' ]       // 声明了按钮初始时是被选中的
                                // 指定了 defaultChecked 属性
  onclick = "handler" ]        // 在点击了按钮时要执行的 JavaScript 语句
>
// 出现在按钮旁边的 HTML 文本
...
</form>

```

描述

Radio 元素表示 HTML 表单中的一个图形化单选按钮。所谓单选按钮 (radio button)，就是一组互斥按钮中的一个按钮。当该组中的一个按钮被选中时，之前选中的那个按钮就不再处于选中状态。事件处理程序 onclick 允许指定在选中按钮时要执行的 JavaScript 代码。

可以通过检测属性 checked 来确定按钮的状态，也可以通过设置这一属性来选中一个按钮，或取消对一个按钮的选择。注意，设置 checked 属性虽然可以改变按钮的图形外观，但是却不会调用事件处理程序 onclick。checked 属性的初始值和 defaultChecked 属性的值都是由性质 checked 设置的。只有同一组中的 Radio 元素才可以具有这一性质，因为它在把某个元素的 checked 属性和 defaultChecked 属性设置成 true 时，必须把该组的其他单选按钮都设置成 false。如果没有一个 Radio 元素具有 checked 性质，就默认组中的第一个 Radio 元素的属性为 checked (及 defaultChecked)。

注意，出现在单选按钮旁边的文本并非 Radio 元素自身的一部分，必须在创建 Radio 元素的 HTML 标记 <input> 之外设置它。

Radio 元素用于选项互斥的选项组。在 HTML 表单中，一个选项互斥的选项组就是名字相同的 Radio 元素的集合。假定表单 f 中含有一个这样的选项组，其中的 Radio 元素共同使用名字 opts，那么 f.opts 指的就是一个 Radio 元素的数组，f.opts.length 是这个数组中的元素个数。

可以把一个 Radio 元素的 value 性质设置成一个字符串，这样在提交表单时，如果该 Radio 元素处于被选中的状态，就会把这个字符串传递给服务器。由于同一组的 Radio

元素都有惟一的 value 值, 所以服务器上的脚本可以据此来判断提交表单时选中的 Radio 元素。

习惯用法

使用 Radio 元素可以给用户显示一个选项列表, 这个列表中可以含有多个选项, 但是每个选项都是互斥的。如果仅要显示一个选项, 或者要显示多个选项, 但各个选项不是互斥的, 可以使用 Checkbox 元素。

参阅

Checkbox、Form、Input; DOM 参考手册部分的 HTMLInputElement

Radio.onclick

JavaScript 1.0

当单选按钮被选中时调用的事件处理程序

摘要

```
<input type="radio" onclick="handler" ... >  
radio.onclick
```

描述

Radio 对象的 onclick 属性引用了一个事件处理函数, 当用户点击了该复选框时, 就会调用这个事件处理函数。要了解完整的细节, 请参阅“HTMLElement.onclick”参考页。但要注意, 从 JavaScript 1.0 开始, JavaScript 就开始支持 Radio.onclick 属性, 这是 Radio.onclick 有别于通用的 HTMLElement.onclick 处理程序的地方。

参阅

HTMLElement.onclick; 第十九章; DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

Reset

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

用于重置表单值的按钮

继承 Input, HTMLElement

摘要

```
form.name  
form.elements[i]
```

属性

Reset 对象继承了 Input 对象和 HTMLElement 对象的属性，此外还定义或覆盖了如下的属性：

value

一个字符串，声明了出现在 Reset 按钮上的文本。这个属性的值由创建 **Reset** 按钮的 `<input>` 标记的 `value` 性质设置。如果没有指定 `value` 性质，它的默认值是“Reset”（或浏览器的默认语言采用的等价值）。在不能回流文档内容的浏览器中，该属性是只读的。

方法

Reset 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Reset 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序，此外还定义或覆盖了如下的事件处理程序：

onclick

点击 **Reset** 按钮时调用的事件处理程序。

HTML 语法

Reset 元素是由标准的 HTML 标记 `<input>` 创建的：

```
<form>
  ...
  <input
    type = 'reset'           // 声明了这是一个 Reset 按钮
    [ value = 'label' ]     // 在按钮上显示的文本
                           // 指定了属性 value
    [ name = 'name' ]       // 此后可以用来引用这个按钮的名字
                           // 指定了属性 name
    [ onclick = handler' ]  // 在点击了该按钮时要执行的 JavaScript 语句
  >
  ...
</form>
```

用 HTML 4 的 `<button>` 标记也可以创建 Reset 对象：

```
<button id = 'name'
  type = 'reset'
  onclick = handler">
  label
```

```
</button>
```

描述

Reset 元素具有与 Button 元素相同的属性和方法，不过它的作用更加专一。在点击了 Reset 元素时，含有该元素的表单中的所有输入元素都会被重置为它们的初始默认值（对于大多数的 HTML 元素来说，这个值都是由性质 value 设置的）。如果没有指定的初始值，那么点击 **Reset** 按钮就会清除用户在这些元素中输入的值。

习惯用法

如果没有给 Reset 元素指定 value 性质，那么它的标签就是“Reset”。在某些表单中，将它标记为“Clear Form”或“Defaults”更加合适。

在 JavaScript 1.1 中，可以调用 Form 对象的 reset() 方法来模拟 **Reset** 按钮的动作。此外，在 JavaScript 1.1 中，Form 对象的 onreset 事件处理程序是在重置表单之前调用的。让这个事件处理程序返回 false 可以取消对表单的重置。

参阅

Button、Form、HTMLElement、Input；DOM 参考手册部分的 HTMLInputElement

Reset.onclick

JavaScript 1.0;在 JavaScript 1.1 中增强了它

点击了 Reset 按钮时调用的事件处理程序

摘要

```
<input type='reset' onclick="handler" ... >  
reset.onclick
```

描述

Reset 对象的 onclick 属性引用一个事件处理函数，当用户点击了 **Reset** 按钮时，这个处理函数就会被调用。要了解详情，请参阅“HTMLElement.onclick”参考页。但要注意，与一般化的 HTMLElement.onclick 处理函数不同，Reset.onclick 是从 JavaScript 1.0 起就受到支持了。

Reset 按钮具有特殊的功能，即将所有表单元素都重置为它们的初始值。使用 onclick 事件处理函数可以给 **Reset** 按钮添加额外的功能。在 JavaScript 1.1 中，可以使 onclick 处理函数返回 false 来阻止 Reset 对象重置表单。（例如，onclick() 处理函数可以使用 conform() 方法请求用户对重置进行确认，如果没有得到确认，就返回 false。）

参阅

Form.onreset、Form.reset()、HTMLElement.onclick；第十九章；DOM 参考手册部分的 EventListener、EventTarget、MouseEvent

Screen

JavaScript 1.2

提供有关显示器的信息

摘要

screen

属性

availHeight

声明了显示 web 浏览器的屏幕的可用高度，以像素计。在 Windows 这样的操作系统中，这个可用的高度不包括分配给半永久特性（如屏幕底部的任务栏）的垂直空间。

availLeft[Netscape 4]

声明了屏幕最左侧的 X 坐标，这个坐标不是分配给暂存特性（如引用程序的快捷方式栏和 Windows 95 的任务栏）的空间的坐标。

availTop[Netscape 4]

声明了屏幕最顶部的 Y 坐标，这个坐标不是分配给半永久特性（如应用程序的快捷方式栏和 Windows 95 的任务栏）的空间的坐标。

availWidth

声明了显示 web 浏览器的屏幕的可用宽度，以像素计。在 Windows 这样的操作系统中，这个可用的宽度并不包括分配给半永久特性（如应用程序的快捷方式栏）的水平空间。

colorDepth

声明了浏览器分配的颜色数的以 2 为底的对数，可用于显示图像。例如，如果浏览器预分配了 128 种颜色，那么 screen.colorDepth 的值就是 7。在不分配调色板的系统中，这个值与屏幕的每像素的位数相同。

在 IE 4 中，colorDepth 声明的是屏幕的颜色深度，以每像素的位数计，而不是预分配的调色板的深度。在 Netscape 中，screen.pixelDepth 属性提供该值。

height

声明了显示 web 浏览器的屏幕的高度, 以像素计。参阅 availHeight。

pixelDepth[Netscape 4]

声明了显示浏览器的屏幕的颜色深度, 以每像素的位数计。与 colorDepth 对比。

width

声明了显示 web 浏览器的屏幕的宽度, 以像素计。参阅 availWidth。

描述

每个 Window 对象的 screen 属性都引用一个 Screen 对象。这个全局对象的静态属性存放了有关浏览器显示的屏幕的信息。JavaScript 程序将利用这些信息来优化它们的输出, 以达到用户的显示要求。例如, 一个程序可以根据显示器的尺寸选择使用大图像还是使用小图像, 它还可以根据显示器的颜色深度选择使用 16 位色的图像还是使用 8 位色的图像。另外, JavaScript 程序还能够根据有关屏幕尺寸的信息将新的浏览器窗口定位在屏幕中间。

参阅

Window.screen

Select

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

图形化选项列表

继承 Input, HTMLElement

摘要

`form.element_name`

`form.elements[i]`

属性

Select 对象继承了 Input 对象和 HTMLElement 对象的属性, 此外还定义或覆盖了如下一些属性:

length

一个只读整数, 它声明了 `options[]` 数组中的元素个数。它的值与属性 `options.length` 的值相同。

options

Option 对象的数组，其中每个元素描述了 Select 元素中显示的一个选项。有关该数组的详细情况，包括修改 Select 对象显示的选项的方法，请参阅“Select.options[]”参考页。

selectedIndex

一个整数，它声明了 Select 对象中被选中的选项下标。如果没有选中任何选项，那么该属性的值就为 -1。如果选中多个选项，那么它声明的只是第一个选项的下标。

在 JavaScript 1.0 中，selectedIndex 属性是只读的。在 JavaScript 1.1 中，这个属性是可读可写的，可以通过设置它而选中某个特定的选项。还可以取消对所有选项的选定，即使 Select 对象具有指定的 multiple 性质。在采用列表框的选择样式（而不是下拉菜单式的选择样式）时，可以把 selectedIndex 设置为 -1 而取消对所有选项的选定。注意，以这种方式改变选择不会触发事件处理程序 onchange()。

type {JavaScript 1.1}

一个只读字符串，由所有的表单元共享，声明了元素的类型。Select 对象比较特殊，因为它有两种可能的 type 属性值。如果 Select 对象只允许选择一个选项（也就是在对象的 HTML 定义中没有设置 multiple 性质），则 type 属性的值就为“select-one”（单选）。如果设置了 multiple 性质，那么 type 性质的值就为“select-multiple”（多选）。参阅“Input.type”参考页。

方法

Select 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Select 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序，此外还定义或覆盖了如下的一些事件处理程序：

onchange

在用户选中了一个选项或取消了对某个选项的选定时调用的事件处理程序。

HTML 语法

Select 元素是由标准的 HTML 标记 <select> 创建的。Select 元素中的选项由 <option> 标记创建：

```
<form>
...
<select
  name = "name"           // 用于标识该元素的名字, 指定了 name 属性
  [ size = 'integer' ]    // Select 元素中可见的选项数
  [multiple]              // 如果出现了这一属性, 那么就可以同时选择多个选项
  [ onchange = 'handler' ] // 在选择改变时调用的事件处理程序
>
<option value = "value1" [selected]> option_label1
<option value = "value2" [selected]> option_label2

// 其他的选项
</select>
...
</form>
```

描述

Select 元素表示一个图形化的选项列表, 用户可以选择其中的选项。如果在该元素的 HTML 定义中出现了 multiple 性质, 那么用户就可以从列表中选择任意多个选项。如果没有设置这个性质, 那么用户就只能选择一个选项, 此时选项的行为就和单选按钮一样, 即无论选中了哪一个选项, 在它之前选中的选项就会被取消选定。

显示 Select 元素的选项的方式有两种。如果它的 size 性质值大于 1, 或者设置了 multiple 性质, 那么可以在一个列表框中显示它的选项, 这个列表框在浏览器窗口中的高度为 size 行。如果 size 的值比选项数小, 那么列表框就会加入一个滚动条以便访问所有的选项。不过, 如果 size 的值为 1 而且又没有设置 multiple 属性, 那么就会将当前选中的选项显示在一行之中, 通过下拉式菜单可以访问其他的选项。第一种显示方式可以清楚地显示出选项, 但是占用浏览器窗口的空间比较大。第二种显示方法需要占用的空间最小, 但是又不能明确地显示出可选项。

Select 元素的 options[] 属性是最为有趣的属性。它是 Option 对象的数组, 它描述了 Select 元素中显示的一个选项。属性 length 声明了数组的长度 (与 options.length 相同)。要了解详情, 请参阅 Option 对象的说明。

在 JavaScript 1.1 中, 可以动态地修改 Select 元素显示的选项。通过设置 Option 对象的 text 属性, 可以改变一个 Option 对象显示的文本。设置 options.length 属性, 可以改变 Select 元素显示的选项数目。另外, 还可以用构造函数 Option() 创建新的选项。要了解详情, 请参阅 “Select.options[]” 和 “Option” 参考页。

注意, Select 对象是一种 Input 对象, 虽然它不是由 HTML 标记 <input> 创建的, 但它是从 Input 对象继承而来的。

参阅

Form、HTMLElement、Input、Option; DOM 参考手册部分的 HTMLSelectElement

Select.onChange

JavaScript 1.0

当改变选择时调用的事件处理程序

摘要

```
<select ... onchange="handler" ... >  
select.onChange
```

描述

Select 对象的 onChange 属性引用一个事件处理函数。当用户选中了一个选项，或者取消了对一个选项的选定时，就会调用这个处理函数。要了解这一事件处理函数的详情，请参阅“Input.onChange”参考页。

参阅

Input.onChange、Option; 第十九章; DOM 参考手册部分的 Event、EventListener、EventTarget

Select.options[]

JavaScript 1.0;在 JavaScript 1.1 中增强了它

Select 对象中的选项

摘要

```
select.options[i]  
select.options.length
```

描述

属性 options[] 存放了一个 Option 对象的数组，其中的每个元素描述了 Select 对象 select 中的一个选项。属性 options.length 声明了该数组中的元素数，它与属性 select.length 相同。要进一步了解详情，请参阅 Option 对象。

在 JavaScript 1.1 中，可以修改 Select 对象显示的选项，采用的方法有如下几种：

- 如果把 options.length 属性设置为 0，Select 对象中的所有选项都会被清除。

- 如果 `options.length` 属性的值比当前值小, `Select` 对象的选项数就会减少, 那么出现在选项数组尾部的元素就被舍弃。
- 如果把 `options[]` 数组中的一个元素设置为 `null`, 那个选项就会从 `Select` 对象中删除, 位于该元素之后的元素将被改变下标值以使它们的位置向前提, 补上数组中的空缺。
- 如果用构造函数 `Option()` 创建了一个新的 `Option` 对象 (参看 `Option` 的参考页), 可以通过把这个新建的选项添加到 `options[]` 数组的结尾, 就把那个选项添加到 `Select` 对象的选项列表中, 要实现这一点, 只需要设置 `options [options.length]` 即可。

参阅

`Option`

Style

Internet Explorer 4; Netscape 6

级联样式表性质

摘要

`htmlElement.style`

属性

`Style` 对象提供了对应于浏览器支持的 CSS 性质的属性。

描述

`Style` 对象的属性直接对应于浏览器支持的级联样式表 (CSS) 的性质。但为了与 JavaScript 语法兼容, 用连字符号连接的 CSS 性质名被替换为删除了连字符号、大小写混合的形式。因此, CSS 性质 `color` 被 `Style` 对象的 `color` 属性表示, `background-color` 性质在 `Style` 对象中就被表示为 `backgroundColor` 性质。参阅第十八章以获得有关元素样式的更多资料。

参阅

`HTMLElement.style`; 第十八章; DOM 参考手册部分的 `CSSStyleDeclaration`、`CSS2Properties`

Submit

JavaScript 1.0;在 JavaScript 1.1 中增强了它

用于提交表单的按钮

继承 Input, HTMLElement

摘要

`form.name`

`form.elements[i]`

`form.elements['name']`

属性

Submit 对象继承了 Input 对象和 HTMLElement 对象的属性, 此外还定义或覆盖了如下的属性:

`value`

一个只读字符串, 声明了出现在 **Submit** 按钮上的文本。这个属性的值是由创建 **Submit** 按钮的 `<input>` 标记的 `value` 性质设置的。如果没有指定 `value` 性质, 它的默认值是 “Submit Query” 或浏览器的默认语言中的等价字符串。在不能回流文档内容的浏览器中, 该属性是只读的。

方法

Submit 对象继承了 Input 对象和 HTMLElement 对象的方法。

事件处理程序

Submit 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序, 此外还定义或覆盖了如下的事件处理程序:

`onclick`

在 **Submit** 按钮被点击时调用的事件处理程序。

HTML 语法

Submit 对象是由标准的 HTML 标记 `<input>` 创建的:

```
<form>
...
<input
  type = 'submit'           // 声明了这是一个 Submit 按钮
  [ value = 'label' ]      // 在按钮上显示的文本
                           // 指定了属性 value
  [ name = 'name' ]        // 此后可以用来引用这个按钮的名字
                           // 指定了属性 name
```

```
    onclick = 'handler' ); // 在点击了该按钮时要执行的 JavaScript 语句
    ...
</form>
```

还可以用 HTML 4 的 <button> 标记创建 **Submit** 按钮:

```
<button id = 'name'
        type = 'submit'
        value = 'value'
        onclick = 'handler' > label
</button>
```

描述

Submit 元素具有与 Button 对象相同的属性和方法，不过它的作用更加专门。点击 **Submit** 按钮时，它就会把含有该按钮的表单中的数据提交给服务器，这个服务器是由表单的 action 性质指定的。此外它还会把服务器发送回来的 HTML 页面装载进来。在 JavaScript 1.1 中有一个例外，那就是如果事件处理程序 Submit.onclick 或 Form.onsubmit 返回 false，那么表单就不会被提交。

注意，在 JavaScript 1.1 中，Form.submit() 提供了另一种可选的提交表单的方法。

如果没有给 Submit 对象指定 value 性质，那么通常以“Submit Query”作为 Submit 按钮的标签。在某些表单中，使用“Submit”、“Done”或“Send”作为该按钮的标签更有意义。

参阅

Button、Form.onsubmit、Form.submit()、HTMLElement、Input: DOM 参考手册部分的 HTMLInputElement

Submit.onclick

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

点击 Submit 按钮时调用的事件处理程序

摘要

```
<input type="submit" onclick="handler" ... >
submit.onclick
```

描述

Submit 对象的 onclick 属性引用了一个事件处理函数。当用户点击了 **Submit** 按钮

时, 这个处理函数就会被调用。要了解详情, 请参阅“`HTMLElement.onclick`”参考页。但要注意, 与一般化的`HTMLElement.onclick`处理函数不同, `Submit.onclick`从 JavaScript 1.0 起就受到支持了。

Submit 按钮具有特殊的功能, 即把表单提交给服务器。使用 `onclick` 事件处理函数可以给 **Submit** 按钮添加额外的功能。在 JavaScript 1.1 中, 可以使 `onclick` 处理函数返回 `false` 来阻止 `Submit` 对象提交表单。(例如, 用 `onclick` 处理函数可以执行表单确认, 如果表单中必须要填写的栏目没有填写, 它可以返回 `false`。)

参阅

`Form.onsubmit`、`Form.submit()`、`HTMLElement.onclick`; 第十九章; DOM 参考手册部分的 `EventListener`、`EventTarget`、`MouseEvent`

Text

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

图形化的文本输入框

继承 `Input`, `HTMLElement`

摘要

`form.name`

`form.elements[i]`

属性

`Text` 对象继承了 `Input` 对象和 `HTMLElement` 对象的属性, 此外还定义或覆盖了如下的属性:

`value`

一个可读可写的字符串, 声明了文本输入框中显示的文本。该文本可以是用户输入的, 也可以是文档或脚本设置的默认值。它的初始值是由定义 `Text` 对象的标记 `<input>` 的 `value` 性质设置的。当用户在 `Text` 对象中输入字符时, `value` 属性的值就会随着用户的输入而更新。如果明确地设置了 `value` 属性, `Text` 对象就会显示指定的字符串。而且, `value` 属性也可以指定在提交表单时要发送给服务器的字符串。

方法

`Text` 对象继承了 `Input` 对象和 `HTMLElement` 对象的方法。

事件处理程序

Text 对象继承了 Input 对象和 HTMLElement 对象的事件处理程序，此外还定义或覆盖了如下的事件处理程序：

onchange

在用户改变了 Text 元素中的值，并且把键盘焦点移到了别处时调用的事件处理程序。并非在 Text 元素中的每一次击键都会调用 onchange 处理函数，只有当用户完成了一次编辑后，才会调用它。

HTML 语法

Text 元素是由标准的 HTML 标记 <input> 创建的：

```
<form>
...
<input
  type = "text"           // 声明了这是一个 Text 元素
  [ name = "name" ]      // 此后可以用来引用这个元素的名字
                        // 指定了属性 name
  [ value = 'default' ]   // 在提交表单时默认的发送给服务器的值
                        // 指定了属性 defaultValue
  [ size = "integer" ]    // 该元素的宽度可以容纳多少个字符
  [ maxlength = "integer" ] // 允许输入的最大字符数
  [ onchange = "handler" ] // 事件处理函数 onchange()
...
</form>
```

描述

Text 元素表示表单中的文本输入框。它的 size 性质声明了该输入框在屏幕上出现时的宽度，以字符为计量单位，maxlength 性质声明了允许用户输入的最大字符数。

除了 HTML 性质之外，令人感兴趣的主要属性是 value。通过读取这个属性，可以获得用户的输入数据，对它进行设置，则可以在输入框中显示任意的（无格式的）文本。

习惯用法

当你要用户输入的是敏感信息时，最好使用 Password 元素来代替 Text 元素。因为像口令这种数据，不应该公开地显示在屏幕上。使用 Textarea 元素可以允许用户输入多行文本。

如果一个表单只含有 Text 元素或 Password 元素，那么当输入焦点在 Text 元素或 Password 元素中时，用户敲击了回车键，表单就会被自动提交。对许多表单而言，这

是一种非常有用的快捷方式。不过，在某些情况下，如果用户敲击了回车键就提交表单，但此时还没有给其他的表单元素（如 Checkbox 元素和 Radio 元素）输入值，就会产生混乱。要将出现这种混乱的几率降到最小，只需把具有这种默认提交行为的 Text 元素放置在表单的底部即可。

参阅

Form、Input、Password、Textarea; DOM 参考手册部分的 HTMLInputElement

Text.onChange

JavaScript 1.0

当输入值改变时调用的事件处理程序

摘要

```
<input type="text" onchange="handler" ... >
```

text.onChange

描述

Text 元素的 onChange 属性引用了一个事件处理函数。当用户改变了输入框中的值并且把键盘焦点移到别处（如在别处点击了鼠标，或者敲击了 Tab 键或回车键）时就会调用这个处理函数。

注意，当使用 JavaScript 代码设置 Text 对象的 value 属性时，并不会调用 onChange 事件处理函数。另外还要注意，该处理函数用于处理输入值的完整改变，因此并非每次击键都会调用它。要了解接受每次按键事件的通知消息的处理函数，请参阅“HTMLInputElement.onkeypress”参考页。

要了解 onChange 事件处理函数的详细信息，请参阅“Input.onChange”参考页。

参阅

HTMLInputElement.onkeypress、Input.onChange; 第十九章; DOM 参考手册部分的 Event、EventListener、EventTarget

Textarea

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

多行文本输入区域

继承 Input, HTMLInputElement

摘要

form.name

```
form.elements[i]
```

属性

Textarea对象继承了Input对象和HTMLElement对象的属性, 此外还定义或覆盖了如下的属性:

value

一个可读可写的字符串。它的初始值与属性defaultValue的值相同, 它们都是出现在标记<textarea>和</textarea>之间的纯文本 (即不具有任何HTML标记)。当用户在Textarea对象中输入字符时, value属性的值就会随着用户的输入而更新。如果明确地设置了value属性, Textarea对象就会显示指定的字符串。value属性存放的字符串就是在提交表单时要发送给服务器的数据。

方法

Textarea对象继承了Input对象和HTMLElement对象的方法。

事件处理程序

Textarea对象继承了Input对象和HTMLElement对象的事件处理程序, 此外还定义或覆盖了如下的事件处理程序:

onchange

在用户改变了Textarea元素中的值, 并且把键盘焦点移到了别处时调用的事件处理程序。并非在Textarea元素中的每一次击键都会调用onchange处理函数, 只有当用户完成了一次编辑之后, 才会调用这一处理函数。

HTML 语法

Textarea元素是由标准的HTML标记<textarea>和</textarea>创建的:

```
<form>
...
< textarea
  [ name = 'name' ]           // 此后可以用这个名字引用这个元素
  [ rows = 'integer' ]       // 该元素有多少行高
  [ cols = 'integer' ]       // 该元素有多少个字符宽
  [ onchange = 'handler' ]   // 事件处理函数 onchange()
>
  plain_text                 // 初始文本, 由属性 defaultValue 设置
</textarea>
...
</form>
```

描述

Textarea 元素表示表单中的文本输入域。它的 name 性质声明了该元素的名字。在提交表单时, 必须使用这个名字来引用该元素, 而且它也给 JavaScript 代码引用 Textarea 元素提供了一种简便的方式。cols 性质声明了元素在屏幕上显示时的宽度, 它是以字符数来计的。rows 性质声明的是元素的高度, 是以文本的行数来计的。wrap 性质声明了要处理多少行, off 说明实际文本有多长, 一行就有多长, virtual 说明显示一行文本时应该有换行符, 但是传递一行文本时没有换行符, physical 说明无论显示一行文本还是传递文本都应该有换行符。

除了那些 HTML 性质之外, 令人感兴趣的属性是 value。通过读取这个属性, 可以获得用户的输入数据, 对它进行设置, 则可以在 Textarea 中显示任意的 (无格式的) 文本。该属性的初始值 (以及 defaultValue 属性的永久值) 就是出现在 <textarea> 和 </textarea> 之间的文本。

注意, Textarea 对象是一种 Input 对象, 尽管它不是由 <input> 标记创建的, 它是由 Input 对象继承而来的。

习惯用法

如果你只需要一行输入文本, 那么最好使用 Text 元素。如果要输入的文本是敏感信息 (如口令), 最好使用 Password 元素。

参阅

Form、HTMLElement、Input、Password、Text; DOM 参考手册部分的 HTMLTextAreaElement

Textarea.onChange

JavaScript 1.0

当输入值改变时调用的事件处理程序

摘要

```
<textarea onchange="handler" ... >
...
</textarea>

textarea.onChange
```

描述

Textarea 元素的 `onchange` 属性引用一个事件处理函数，当用户改变了文本域中的值并且把键盘焦点移到别处时就会调用这个处理函数。

注意，当使用 JavaScript 代码设置 Text 对象的 `value` 属性时，并不会调用 `onchange` 事件处理函数。另外还要注意，该处理函数是处理输入值的完整改变的，因此并非每次击键都会调用它。要了解接受每次按键事件的通知消息的处理函数，请参阅“`HTMLElement.onkeypress`”参考页。

要了解 `onchange` 事件处理函数的详细情况，请参阅“`Input.onchange`”参考页。

参阅

`HTMLElement.onkeypress`、`Input.onchange`；第十九章；DOM 参考手册部分的 `Event`、`EventListener`、`EventTarget`

URL

参阅 `Link`、`Location` 或 `Document.URL`

Window

JavaScript 1.0;在 JavaScript 1.1 和 1.2 中增强了它

Web 浏览器窗口或框架

摘要

`self`

`window`

`window.frames[i]`

属性

Window 对象定义了如下的属性。那些不可移植的、浏览器特定的属性单独列在下面的列表之后：

`closed`

一个只读的布尔值，声明了窗口是否已经关闭。当浏览器窗口关闭时，表示该窗口的 Window 对象并不会消失，它将继续存在，不过它的 `closed` 属性设置为 `true`。

`defaultStatus`

一个可读可写的字符串，声明了显示在状态栏中的默认消息。参阅“`Window.defaultStatus`”参考页。

`document`

对描述窗口或框架中含有的文档的 `Document` 对象的只读引用。详见 `Document` 对象。

`frames[]`

`Window` 对象的数组，每个 `Window` 对象在窗口中含有的一个框架。属性 `frames.length` 存放数组 `frames[]` 中含有的元素个数。注意，`frames[]` 数组中引用的框架自身可能还含有框架，它们自己也具有 `frames[]` 数组。

`history`

对窗口或框架的 `History` 对象的只读引用，详见 `History` 对象。

`length`

窗口或框架包含的框架个数，也是数组 `frames[]` 中的元素数。

`location`

用于窗口或框架的 `Location` 对象。该对象声明了当前装载进来的文档的 URL。把这个属性赋予一个新的 URL 字符串，会引发浏览器装载并显示出那个 URL 所指的文档。要进一步了解该属性，请参阅 `Location` 对象。

`Math`

对一个对象的引用，该对象含有各种算术函数和常量。详见 `Math` 对象。

`name`

一个字符串，存放了窗口的名字。这个名字是在 `open()` 方法创建窗口时指定的，可选。在 JavaScript 1.0 中它是只读的，在 JavaScript 1.1 中是可读可写的。参阅“`Window.name`”参考页。

`navigator`

对 `Navigator` 对象的只读引用，提供 Web 浏览器的版本信息和配置信息。详见 `Navigator` 对象。

`opener[JavaScript 1.1]`

一个可读可写的属性，是对一个 `Window` 对象的引用，该对象含有调用了 `open()` 方法的脚本以打开顶级浏览器窗口的脚本。只有表示顶层窗口的 `Window` 对象的

`opener` 属性才有效, 表示框架的 `Window` 对象的 `opener` 属性无效。属性 `opener` 之所以非常有用, 是因为新创建的窗口可以通过这个属性来引用创建它的窗口定义的变量和函数。

`parent`

对一个 `Window` 对象的只读引用, 这个 `Window` 对象包含当前的窗口或框架。如果 `window` 是顶层窗口, 所以 `parent` 引用的就是窗口自身。如果 `window` 是框架, 那么 `parent` 引用的是包含 `window` 的窗口或框架。

`screen`[JavaScript 1.2]

一个 `Screen` 对象, 由浏览器中的所有的窗口共享。这个 `Screen` 对象存放的属性声明了与屏幕有关的信息, 即可用的像素数和可用的颜色数。详见 `Screen` 对象。

`self`

对窗口自身的只读引用。等价于 `window` 属性。

`status`

一个可读可写的字符串, 声明了浏览器状态栏的当前内容。详见“`Window.status`”参考页。

`top`

对一个顶级窗口的只读引用, 顶级窗口包含了这个窗口。如果窗口自身就是一个顶层窗口, `top` 属性存放了对窗口自身的引用。如果窗口是一个框架, `top` 属性引用了包含框架的顶层窗口。比较 `parent` 属性。

`window`

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

Netscape 属性

`innerHeight, innerWidth`[Netscape 4]

可读可写的属性, 声明了窗口的文档显示区的高度和宽度, 以像素计。这里宽度和高度不包括菜单栏、工具栏以及滚动条等的高度。作为一安全规约, 不能把这些属性设置为小于 100 像素的值。

`java`[Netscape 3]

对一个 `JavaPackage` 对象的引用, 该对象是构成 Java 语言的核心 `java.*` 包的包名层次的顶层包。参阅“`JavaPackage`”参考页。

`locationbar.visible` [Netscape 4]

一个只读布尔值, 声明窗口是否显示地址栏。参阅“`Window.open()`”参考页。

`menubar.visible` [Netscape 4]

一个只读布尔值, 声明窗口是否显示菜单栏。参阅“`Window.open()`”参考页。

`netscape` [Netscape 3]

对一个 `JavaPackage` 对象的引用, 该对象是包名层次的顶层包, 这个包名层次是用于 Netscape 公司提供的 Java 包 `netscape.*` 的。参阅“`JavaPackage`”参考页。

`outerHeight, outerWidth` [Netscape 4]

可读可写的属性, 声明了整个窗口的高度和宽度, 以像素计。它们包括菜单栏、工具栏、滚动条和窗口边线等的高度和宽度。

`Packages` [Netscape 3]

对一个 `JavaPackage` 对象的引用, 该对象表示 Java 包名字层次的顶层包。例如, 用 `Packages.java.lang` 可以引用 `java.lang` 包。参阅“`JavaPackage`”参考页。

`pageXOffset, pageYOffset` [Netscape 4]

只读的整数, 声明了当前文档向右 (`pageXOffset`) 和向下 (`pageYOffset`) 滚动过的像素数。

`Personalbar.visible` [Netscape 4]

一个只读布尔值, 声明了窗口是否显示书签的“个人栏”。参阅“`Window.open()`”参考页。

`screenX, screenY` [Netscape 4]

只读整数, 声明了窗口的左上角在屏幕上的 X 坐标和 Y 坐标。如果 window 是一个框架, 这个属性声明了含有那个框架的顶层窗口的 X 坐标和 Y 坐标。

`scrollbars.visible` [Netscape 4]

一个只读布尔值, 声明窗口的滚动条是否可见。如果文档足够长或足够宽, 是否需要显示滚动条。该属性真正声明的是该窗口是否启用滚动。参阅“`Window.open()`”参考页。

`statusbar.visible` [Netscape 4]

一个只读布尔值, 声明了窗口是否具有状态栏。参阅“`Window.open()`”参考页。

`sun` [Netscape 3]

对一个 `JavaPackage` 对象的引用, 该对象是包名层次的顶层包, 这个包名层次是

用于 Sun Microsystems 公司提供的 Java 包 *sun.** 的。参阅“JavaPackage”参考页。

`toolbar.visible`*[Netscape 4]*

一个只读布尔值，声明窗口是否显示工具栏。参阅“Window.open()”参考页。

Internet Explorer 属性

`clientInformation`*[IE 4]*

IE 中与 `navigator` 属性同义。这两个属性引用的都是 `Navigator` 对象。尽管 `clientInformation` 这个名字比较好，不像 `navigator` 那样看起来是 `Navigator` 特有的，但是 Netscape 并不支持它，因此它不可移植。

`event`*[IE 4]*

一个 `Event` 对象，该对象存放窗口 `window` 中最近发生的事件的详细信息。在 Netscape 4 的事件模型中，描述事件的 `Event` 对象作为参数传递给每个事件处理程序。而在 IE 事件模型中，并不传递 `Event` 对象，事件处理程序只有从 `Window` 对象的 `event` 属性中才能得到事件的信息。

方法

`Window` 对象具有以下可移植的方法。那些不可移植的、浏览器特定的方法单独列在下面的列表之后。

<code>alert()</code>	在对话框中显示简单的消息。
<code>blur()</code>	把键盘焦点从顶层浏览器窗口中移走。在大多数平台上，这将使窗口成为背景。
<code>clearInterval()</code>	取消周期性执行的代码。
<code>clearTimeout()</code>	取消挂起超时操作。
<code>close()</code>	关闭窗口。
<code>confirm()</code>	用对话框询问一个回答为是或否的问题。
<code>focus()</code>	把键盘焦点给予顶层浏览器窗口。在大多数平台上，这将使窗口移到最前边。
<code>moveBy()</code>	把窗口移动一个相对的数量。
<code>moveTo()</code>	把窗口移动到一个绝对的位置。

<code>open()</code>	创建并打开一个新窗口。
<code>print()</code>	模拟对浏览器的 Print 按钮的点击。只有 IE 5 和 Netscape 4 支持该方法。
<code>prompt()</code>	用对话框请求输入一个简单的字符串。
<code>resizeBy()</code>	把窗口大小调整指定的数量。
<code>resizeTo()</code>	把窗口大小调整到指定的大小。
<code>scroll()</code>	滚动窗口中显示的文档。
<code>scrollBy</code>	让窗口滚动指定的数量。
<code>scrollTo()</code>	把窗口滚动到指定的位置。
<code>setInterval()</code>	周期性执行指定的代码。
<code>setTimeout()</code>	在经过指定的时间之后执行代码。

Netscape 方法

<code>back()</code>	其行为和用户点击了 Back 按钮一样。
<code>captureEvents()</code>	指定直接发送给该窗口的事件类型。
<code>forward()</code>	模拟对浏览器的 Forward 按钮的点击。
<code>handleEvent()</code>	为给定的 Event 对象调用合适的事件处理程序。
<code>home()</code>	显示浏览器的主页。
<code>releaseEvents()</code>	指定不再捕捉的事件类型。
<code>routeEvent()</code>	将 Event 对象传递给下一个对它感兴趣的对象的合适的处理程序。
<code>stop()</code>	模拟对浏览器的 Stop 按钮的点击。

Internet Explorer 方法

<code>navigate()</code>	装载并显示出指定的 URL。
-------------------------	----------------

事件处理程序

<code>onblur</code>	当窗口失去焦点时调用的事件处理程序。
---------------------	--------------------

onerror	当发生 JavaScript 错误时调用的事件处理程序。
onfocus	当窗口获得焦点时调用的事件处理程序。
onload	当文档（或框架集）完全被装载进来时调用的事件处理程序。
onmove	当移动窗口时调用的事件处理程序。只在 Netscape 4 中可用。
onresize	当调整窗口大小时调用的事件处理程序。
onunload	当浏览器离开当前文档或框架集时调用的事件处理程序。

描述

Window 对象表示一个浏览器窗口或一个框架。我们在第十三章中对它进行了详细的说明。在客户端 JavaScript 中，Window 对象是全局对象，所有的表达式都在当前 Window 对象的环境中计算。也就是说，要引用当前窗口根本不需要特殊的语法，可以把那个窗口的属性作为全局变量来使用。例如，可以只写 `document`，而不必写为 `window.document`。同样，可以把当前窗口对象的方法当作函数来使用，如只写 `alert()`，而不必写 `window.alert()`。

Window 对象的 `window` 属性和 `self` 属性引用的都是它自己。当你想明确地引用当前窗口，而不仅仅是隐式地引用它时，可以使用这两个属性。除了这两个属性之外，`parent` 属性、`top` 属性以及 `frames[]` 数组都引用了与当前 Window 对象相关的其他 Window 对象。

要引用窗口中的一个框架，可以使用如下的语法：

```
frames[i] or self.frames[i] // 当前窗口的框架
window.frames[i]           // 指定窗口的框架
```

要引用一个框架的父窗口（或父框架），可以使用下面的语法：

```
parent or self.parent // 当前窗口的父窗口
window.parent         // 指定窗口的父窗口
```

要从顶层浏览器窗口含有的任何一个框架中引用它，可以使用如下的语法：

```
top or self.top // 当前框架的顶层窗口
window.top      // 指定框架的顶层窗口
```

新的顶层浏览器窗口由方法 `Window.open()` 创建。当调用这个方法时，应该把 `open()` 调用的返回值存储在一个变量中，然后使用那个变量来引用新窗口。在 JavaScript 1.1 中，新窗口的 `opener` 属性引用了打开它的那个窗口。

一般说来, Window 对象的方法都是对浏览器窗口或框架进行某种操作。而 `alert()` 方法、`confirm()` 方法和 `prompt()` 方法则不同, 它们通过简单的对话框与用户进行交互。

要深入地了解 Window 对象, 请参阅第十三章和各个 Window 属性、方法和事件处理程序的参考页。

参阅

Document; 第十三章; DOM 参考手册的 `AbstractView`

Window.alert()

JavaScript 1.0

在对话框中显示一条消息

摘要

```
window.alert(message)
```

参数

message

要在 Window 上弹出的对话框中显示的纯文本 (而非 HTML 文本)。

描述

方法 `alert()` 将在对话框中把指定的 *message* 显示给用户。该对话框含有 **OK** 按钮, 用户可以点击该按钮来关闭对话框。

在 Windows 平台上, 由 `alert()` 方法显示的对话框是有模式的。在用户关闭该对话框之前, JavaScript 将暂停执行。但是, 在 Unix 平台的 Netscape 4 中, 由 `alert()` 方法显示的对话框是无模式的, JavaScript 将一直执行下去。

习惯用法

在用户给某些表单元素输入了无效信息时显示会错误消息, 这也许是 `alert()` 方法最常见的用法了。警报对话框可以把问题告诉用户, 并且提示用户应该修正什么才能避免这一错误。

`alert()` 方法对话框的外观由平台决定, 不过通常它都具有表示错误、警告和某种警告信息的图形。尽管 `alert()` 可以显示任何消息, 但是对话框的警告图形却意味着该

方法不适合显示简单的信息性消息, 诸如“欢迎访问我的主页”和“你是本周的第 177 位访问者!”等。

注意, 对话框中显示的 *message* 是纯文本串, 而不是 HTML 格式的字符串。可以在这个字符串中使用换行符“\n”, 将消息放在多行显示。也可以使用空格来实现一些基本的格式化, 还可以使用带下划线的字符来模拟水平标尺, 不过所能达到的最终效果很大程度上依赖于对话框使用的字体, 因此它是依赖于系统的。

参阅

Window.confirm()、Window.prompt()

Window.back()

Netscape4

返回前一个文档

摘要

`window.back()`

描述

调用方法 `back()` 将使得浏览器再次显示出窗口 *window* 中显示的前一文档, 就像用户点击了窗口的 **Back** 按钮一样。

注意, 对于有框架的文档来说, `Window.back()` 和 `History.back()` 的表现可能会有一些差异。

Window.blur()

JavaScript 1.1

把键盘焦点从顶层窗口移开

摘要

`window.blur()`

描述

方法 `blur()` 将把键盘焦点从顶层浏览器窗口中移走, 这个窗口由 **Window** 对象指定。如果 **Window** 对象是一个框架, 键盘焦点将转移到含有那个框架的顶层窗口中。在大多数平台上, 一旦顶层窗口失去了键盘焦点, 它就会成为背景 (也就是处于窗口堆栈的底部)。

参阅

`Window.focus()`

`Window.captureEvents()`

Netscape 4

声明要捕捉的事件类型

摘要

```
window.captureEvents(eventmask)
document.captureEvents(eventmask)
layer.captureEvents(eventmask)
```

参数

eventmask

一个整数，声明了窗口、文档或层应该捕捉的事件类型。这个值是 `Event` 类定义的静态事件类型常量中的一个，或是用逐位或运算符 (`|`) 或加号组合在一起的事件类型常量组。

描述

`captureEvents()` 既是 `window` 类的方法，又是 `Document` 类和 `Layer` 类的方法。对于这三种类来说，该方法的目的都相同。在 Netscape 4 的事件模型中声明所有给定类型事件，这些发生在指定的 `Window`、`document` 或 `layer` 中的事件类型都应该传递给窗口、文档或层，而不应该传递给发生这些事件的对象。

要捕捉的事件类型由参数 *eventmask* 声明，它是一个位掩码，由 `Event` 类定义的静态常量构成。要查阅这些位掩码常量的完整列表，请参阅“`Event.TYPE`”参考页。

参阅

`Event`、`Window.handleEvent()`、`Window.releaseEvents()`、`Window.routeEvent()`、第十九章；DOM 参考手册部分的 `EventTarget.addEventListener()`

`Window.clearInterval()`

JavaScript 1.2

停止周期性地执行代码

摘要

```
window.clearInterval(intervalId)
```

参数

intervalId

调用 `setInterval()` 方法返回的值。

描述

方法 `clearInterval()` 将停止周期性地执行指定的代码，对这些代码的操作是调用方法 `setInterval()` 启动的。参数 *intervalId* 必须是调用方法 `setInterval()` 后的返回值。

参阅

`Window.setInterval()`

Window.clearTimeout()

JavaScript 1.0

取消对指定的代码的延迟执行

摘要

`window.clearTimeout(timeoutId)`

参数

timeoutId

方法 `setTimeout()` 返回的值，标识了要取消的延迟执行代码块。

描述

方法 `clearTimeout()` 取消对指定代码的执行，调用方法 `setTimeout()` 可以延迟执行这些代码。参数 *timeoutId* 是调用方法 `setTimeout()` 后的返回值，它标识了要取消的延期执行代码块（可以有多个）。

参阅

`Window.setTimeout()`

Window.close()

JavaScript 1.0

关闭浏览器窗口

摘要

`window.close()`

描述

方法 `close()` 将关闭由 `window` 指定的顶层浏览器窗口。一个窗口可以通过调用 `self.close()` 或只调用 `close()` 来关闭它自身。

在 JavaScript 1.1 中, 只有通过 JavaScript 代码打开的窗口才能够由 JavaScript 代码关闭。这阻止了恶意的脚本终止用户的浏览器。

由于关闭窗口中的框架没有意义, 所以应该只调用表示顶层浏览器窗口的 `Window` 对象的 `close()` 方法, 而不能调用表示框架的 `Window` 对象的 `close()` 方法。

参阅

`Window.open()`、`Window` 对象的 `closed` 属性和 `opener` 属性

Window.confirm()

JavaScript 1.0

询问一个回答为是或否的问题

摘要

`window.confirm(question)`

参数

`question`

要在对话框中显示的纯文本 (而不是 HTML 格式的)。通常, 它应该表达你想让用户回答的问题。

返回值

如果用户点击了 **OK** 按钮, 返回值为 `true`; 如果用户点击了 **Cancel** 按钮, 返回值为 `false`。

描述

方法 `confirm()` 将在窗口 `window` 上弹出的对话框中显示指定的问题 `question`。虽然这个对话框的外观是由平台决定的, 但是通常它都含有指示性的图形, 说明要询问用户的问题。该对话框含有 **OK** 按钮和 **Cancel** 按钮, 用户可以使用这两个按钮来回答问题。如果用户点击了 **OK** 按钮, 那么 `confirm()` 将返回 `true`。如果用户点击了 **Cancel** 按钮, `confirm()` 将返回 `false`。

由 `confirm()` 方法显示的对话框是有模式的, 也就是说, 在用户点击 **OK** 按钮或

Cancel按钮把对话框关闭之前,它将阻塞用户对浏览器窗口的所有输入。由于该方法返回的值由用户的响应决定,所以在调用 `confirm()` 时,将暂停对 JavaScript 代码的执行,在用户作出响应之前,不会执行下一条 JavaScript 语句。

习惯用法

注意,对话框显示的问题 `question` 是一个纯文本串,而不是 HTML 格式的字符串。可以在这个字符串中使用换行符“`\n`”,将问题放在多行显示。也可以使用空格来实现一些基本的格式化,还可以使用带下划线的字符来模拟水平标尺,不过所能达到的最终效果很大程度上依赖于对话框使用的字体,因此它是随系统而定的。

另外,对话框按钮的标签是不可改变的(例如想让它们显示为 **Yes** 和 **No**)。因此,应该小心地编写问题或消息,让它适合于用 **OK** 或 **Cancel** 来回答。

参阅

`Window.alert()`, `Window.prompt()`

Window.defaultStatus

JavaScript 1.0

默认的状态栏文本

摘要

`window.defaultStatus`

描述

属性 `defaultStatus` 是一个可读可写的字符串,声明了要在窗口的状态栏中显示的默认文本。web 浏览器通常用状态栏在装载文件的过程中显示浏览器的进度,或者在鼠标移到超文本链接时显示链接地址。在不显示这些瞬时消息时,默认情况下,状态栏是空白的。不过,可以通过设置 `defaultStatus`, 指定在不使用状态栏时要显示的默认消息,也可以通过读取 `defaultStatus` 属性来获得默认的消息。虽然指定的文本可能会暂时被其他消息覆盖,如在鼠标移到超文本链接上时要显示的消息,但是在擦掉了瞬时消息后,会再次显示 `defaultStatus` 中的消息。

如果设置的 `defaultStatus` 属性是一个表示框架的 `Window` 对象,那么当鼠标处于那个框架之中时(无论该框架是否具有焦点),指定的消息就会显示出来。当指定的 `defaultStatus` 属性属于一个不含有框架的顶层窗口时,只要窗口是可见的,消息就是可见的。如果设置的 `defaultStatus` 属性属于一个含有框架的顶层窗口,那么只有当鼠标移动到分隔框架的边界上时,消息才是可见的。因此,在一个多框架文档

中, 要确保一条消息是可见的, 就应该把文档中所有框架的 `defaultStatus` 属性都设置为这一消息。

习惯用法

`defaultStatus` 用于显示状态栏中的半永久性消息。要显示瞬时消息, 最好使用 `status` 属性。

参阅

`Window.status`

`Window.focus()`

JavaScript 1.1

把键盘焦点给予顶层窗口

摘要

`window.focus()`

描述

方法 `focus()` 将把键盘焦点给予顶层浏览器窗口, 这个窗口由 `Window` 对象声明。如果该 `Window` 对象表示一个框架, 那么这个框架和含有它的顶层窗口都将得到键盘焦点。

在大多数平台上, 得到了键盘焦点的顶层窗口都会被前提到窗口堆栈的顶部。

参阅

`Window.blur()`

`Window.forward()`

Netscape 4

前进到下一个文档

摘要

`window.forward()`

描述

调用方法 `forward()` 将使浏览器在 `window` 中显示下一个文档, 就像用户点击了窗口中的 **Forward** 按钮一样。

注意, 对于多框架的文档, 方法 `Window.forward()` 和 `History.forward()` 的行为会有所不同。

Window.handleEvent()

Netscape 4

把事件传递给合适的处理程序

摘要

```
window.handleEvent(event)
document.handleEvent(event)
layer.handleEvent(event)
htmlElement.handleEvent(event)
```

参数

event

要处理的 Event 对象。

返回值

调用来处理事件 *event* 的事件处理程序的返回值。

描述

方法 `handleEvent()` 既是 `Window` 类的方法, 又是 `Document` 类、`Layer` 类以及所有支持事件处理程序的 HTML 元素的方法。当调用对象 *o* 的 `handleEvent()` 方法时, 它将先确定 *event* 参数的类型, 然后将 Event 对象传递给 *o* 的合适的处理程序。

参阅

`Window.routeEvent()`; 第十九章

Window.home()

Netscape 4

显示主页

摘要

```
window.home()
```

描述

调用方法 `home()` 将使浏览器显示出自己配置的主页, 就像用户点击了浏览器的 **Home** 按钮一样。

Window.moveBy()

JavaScript 1.2

把窗口移动一个相对的位置

摘要

```
window.moveBy(dx, dy)
```

参数

dx 要把窗口右移的像素数。

dy 要把窗口下移的像素数。

描述

方法 `moveBy()` 将把窗口 `window` 向右和向下分别移动 *dx* 和 *dy* 个像素。出于安全性方面的原因，浏览器限制脚本不能把窗口移出屏幕。

Window.moveTo()

JavaScript 1.2

把窗口移动到一个绝对位置

摘要

```
window.moveTo(x, y)
```

参数

x 窗口新位置的 X 坐标。

y 窗口新位置的 Y 坐标。

描述

方法 `moveTo()` 将移动窗口 `window`，使它的左上角处于 *x* 和 *y* 指定的位置处。出于安全性方面的原因，浏览器限制方法使之不能把窗口移出屏幕。

Window.name

JavaScript 1.0; 在 JavaScript 1.1 中是可读可写的

窗口的名字

摘要

```
window.name
```

描述

属性 *name* 是一个字符串，声明了窗口 *window* 的名字。在 JavaScript 1.0 中，这个属性是只读的，在 JavaScript 1.1 中是可读可写的。顶层窗口的名字初始时由方法 `Window.open()` 的 *name* 参数设置。框架的名字最初由 HTML 标记 `<frame>` 的 *name* 性质设置。

无论是顶层窗口的名字还是框架的名字都可以作为标记 `<a>` 或 `<form>` 的性质 *target* 的值。以这种方式使用 *target* 性质，声明了超链接的文档或提交表单的结果应该显示在指定的窗口中。

浏览器打开的初始窗口和用 **New Web Browser** 菜单项打开的窗口初始时没有名字（如 `name=""`），因此这些窗口都不能在单独的顶级窗口中用 *target* 性质处理。在 JavaScript 1.1 中，可以通过设置 *name* 性质来弥补这一不足。

参阅

`Form.target`、`Link.target`

Window.navigate()

Internet Explorer 3

装载新 URL

摘要

`window.navigate(url)`

参数

url

一个字符串，声明了要装载并显示的文档的 URL。

描述

Internet Explorer 的方法 `Window.navigate()` 将把指定的 *url* 所指的文档装载到指定的窗口 *window* 中（“导航到” *url*）。

Netscape 不支持该方法。要在 Netscape 和 IE 中都实现这一功能，只需要把想要装载的文档的 *url* 赋给窗口 *window* 的 *location* 属性即可。

参阅

`Location`、`Window` 对象的 *location* 属性

Window.onblurJavaScript 1.1

当窗口失去键盘焦点时调用的事件处理程序

摘要

```
<body onblur='handler' ... >
<frameset onblur="handler" ... >

window.onblur
```

描述

Window 对象的属性 onblur 声明了一个事件处理函数，这个函数是在用户把键盘焦点从该窗口中移开时调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由标记 <body> 或 <frameset> 的 onblur 性质指定的，各个语句之间用分号分隔。

在 Netscape 4 事件模型中，Event 对象作为参数传递给 onblur 处理函数。但是在 IE 事件模型中并不传递给它任何参数，可应用的 Event 对象作为包容 element 的 Window 对象的 event 属性出现。

习惯用法

如果网页具有动画效果，可以使用 onblur() 事件处理程序在窗口失去输入焦点时停止动画。从理论上说，如果窗口没有了焦点，那么用户就看不到它了，或者不再注意它了。

参阅

Window.blur()、Window.focus()、Window.onfocus；第十九章；DOM 参考手册部分的 Event、EventListener、EventTarget

Window.onerror

JavaScript 1.1; Netscape 6/6.1 中的错误

发生 JavaScript 错误时调用的处理程序

摘要

可以使用如下代码注册一个 onerror 事件处理程序：

```
window.onerror=handler-func
```

浏览器调用这个处理程序的代码如下：

```
window.onerror(message, url, line)
```

参数

message

一个字符串，声明了发生的错误的消息。

url

一个字符串，声明了发生错误的文档的 URL。

line

一个数字，声明了发生错误的代码行的行号。

返回值

如果处理程序已经解决了发生的问题，JavaScript 代码不必再作进一步处理，那么返回值是 `true`。如果 JavaScript 程序还要显示出默认的错误消息对话框，那么返回值为 `false`。

描述

Window 对象的 `onerror` 属性声明了一个错误处理函数。当窗口的代码执行时发生了 JavaScript 错误时，就会调用这一错误处理函数。在默认情况下，发生了错误时，JavaScript 将显示出一个错误对话框。你可以通过提供自己的 `onerror` 事件处理函数来定制错误处理。

要定义一个 `onerror` 事件处理函数，需要把 Window 对象的 `onerror` 属性设置成一个适当的函数。注意，与 JavaScript 的其他事件处理程序不同，`onerror` 处理程序不是由 HTML 标记定义的。

在调用 `onerror` 处理程序时，传递给它的参数有三个。第一个是声明错误消息的字符串，第二个是声明发生错误的文档的 URL 的字符串，第三个是发生错误的代码行的行号。有了这三个参数，错误处理函数就能够完成任何想要执行的操作。例如，可以显示它自己的错误对话框，或以某种方式把这个错误记入日志等。当错误处理函数执行完毕时，如果它已经完全解决了问题，不再需要 JavaScript 程序执行进一步的操作了，就返回 `true`。如果它只是以某种方式提示或者记录了错误，还需要 JavaScript 程序在它的默认对话框中显示错误消息，那么返回的就是 `false`。

注意, 虽然这个事件处理程序通过返回 true 来通知浏览器不必再采取进一步的动作了, 但是大多数 Form 事件处理程序以及表单元的事件处理程序都是返回 false 阻止浏览器执行动作 (如提交表单)。这种不一致性容易产生混乱。

可以完全关闭一个窗口的错误处理, 只需要把该窗口的 onerror 属性设置成一个函数, 该函数什么都不做, 返回的总是 true。还可以恢复默认的错误处理行为 (对话框), 只需要把 onerror 属性设置成一个函数, 该函数什么都不做, 返回的总是 false。

Bug

虽然在 Netscape 6 和 Netscape 6.1 中, 错误可以正确地触发这个事件处理程序, 但传递的消息、URL 和行号都不正确, 所以只能用它检测错误的发生, 不能用它获取任何有关错误的有用信息。

Window.onfocus

JavaScript 1.1

在窗口得到焦点时调用的处理程序

摘要

```
<body onfocus="handler" ... >
<frameset onfocus="handler" ... >
window.onfocus
```

描述

Window 对象的属性 onfocus 声明了一个事件处理函数, 这个函数是在窗口得到了键盘焦点的时候调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由标记 <body> 或 <frameset> 的 onfocus 性质指定, 各个语句之间用分号分隔。

在 Netscape 4 事件模型中, Event 对象作为参数传递给 onfocus 处理函数。但是在 IE 事件模型中并不将它传递给它任何参数。可应用的 Event 对象作为包容 element 的 Window 对象的 event 属性出现。

习惯用法

如果网页具有动画效果, 可以使用 onfocus() 事件处理程序启动动画, 使用 onblur 处理程序停止它, 这样就可以在用户注意窗口时才运行动画。

参阅

Window.blur()、Window.focus()、Window.onblur; 第十九章; DOM 参考手册部分的 Event、EventListener、EventTarget

Window.onload

JavaScript 1.0

结束文档装载时调用的处理程序

摘要

```
<body onload="handler" ... >
<frameset onload="handler" ... >
window.onload
```

描述

Window对象的属性onload声明了一个事件处理函数, 这个函数是在结束了文档(或框架集)装载时调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由标记 <body> 或 <frameset> 的 onload 性质指定, 各个语句之间用分号分隔。

-- 一旦调用了 onload 事件处理程序, 就可以确定文档已经完全装载进来了, 也就是说, 文档中所有的脚本都已经执行完毕, 脚本中的所有函数都已经被定义过, 脚本中的表单和其他文档元素也都已经解析过, 通过 Document 对象就可以访问它们。

习惯用法

如果文档中有与文档是否完全装载进来有关的事件处理程序, 那么在执行这些事件处理程序之前, 要确保文档已经被仔细检查了。如果在文档显示出了一个按钮, 但是和这个按钮有关的部分文档还没有装载进来时, 网络连接中断了, 那么当用户点击了这个按钮之后, 他将得到出乎意料的行为, 或者得到一个错误消息。检验文档是否装载完毕的一种有效方法是用 onload 处理程序把一个变量(如 loaded) 设置为 true, 然后在执行任何与文档是否装载完毕有关的操作时, 都检测这个变量。

参阅

Window.onunload; 第十九章; DOM 参考手册部分的 Event、EventListener、EventTarget

Window.onmove Netscape 4; Unix 平台上的 Netscape 4 不支持它

在移动窗口时调用的事件处理程序

摘要

```
<body onmove="handler" ... >
<frameset onmove="handler" ... >
window.onmove
```

描述

Window 对象的属性 onmove 声明了一个事件处理函数，这个函数是在用户把顶层窗口移动到了屏幕上的一个新位置时调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义窗口的 HTML 标记 <body> 或 <frameset> 的 onmove 性质指定。如果一个事件处理函数是由 HTML 性质定义的，那么它就会在 element 的作用域中执行，而不是在包容它的窗口的作用域中执行。

Event 对象作为参数传递给 onmove 处理函数。该对象的属性存放了有关窗口的新的位置的信息。

Window.onresize JavaScript 1.2

在调整窗口大小时调用的事件处理程序

摘要

```
<body onresize="handler" ... >
<frameset onresize="handler" ... >
window.onresize
```

描述

Window 对象的属性 onresize 声明了一个事件处理函数，这个函数是在用户改变窗口或框架的大小时调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数，其中的 JavaScript 语句由定义窗口的 HTML 标记 <body> 或 <frameset> 的 onresize 性质指定。如果一个事件处

理函数是由 HTML 性质定义的, 那么它就会在 `element` 的作用域中执行, 而不是在包容它的窗口的作用域中执行。

在 Netscape 4 事件模型中, `Event` 对象作为参数传递给 `onresize` 处理函数。但是在 IE 事件模型中并不传递给它任何参数, 可应用的 `Event` 对象作为包容 `element` 的 `Window` 对象的 `event` 属性出现。

在 Netscape 中, 通过 `Event` 对象的 `width` 属性和 `height` 属性可以访问窗口的尺寸。

Window.onunload

JavaScript 1.0

在浏览器卸载一个页面时调用的事件处理程序

摘要

```
<body onunload="handler" ... >
<frameset onunload="handler" ... >
window.onunload
```

描述

`Window` 对象的属性 `onunload` 声明了一个事件处理函数, 这个函数是在浏览器卸载了一个文档或框架集, 准备装载一个新文档时调用的。

这个属性的初始值是一个含有 JavaScript 语句的函数, 其中的 JavaScript 语句由标记 `<body>` 或 `<frameset>` 的 `onunload` 性质指定, 各个语句之间用分号分隔。在装载新文档之前, 应该清除浏览器的状态, `onunload` 事件处理程序为此提供了机会。

当浏览器离开使用了框架的站点时, 在调用浏览器自身的 `onunload` 处理程序之前, 会先调用框架集的 `onunload` 处理程序。这会颠倒 `onload` 事件处理程序的调用顺序。

`onunload()` 是在用户指示浏览器离开当前页面, 然后转移到其他页面时调用的。因此, 要让 `onunload` 处理程序弹出一个对话框 (调用 `Window.confirm()` 或者 `Window.prompt()`) 以延迟新页面的装载是无论如何都不合适的。

参阅

`Window.onload`; 第十九章; DOM 参考手册部分的 `Event`、`EventListener`、`EventTarget`

Window.open()

JavaScript 1.0; 在 JavaScript 1.1 中增强了它

打开一个新的浏览器窗口或者查找一个指定的窗口

摘要

```
window.open(url, name, features, replace)
```

参数

url

一个可选的字符串，声明了要在新窗口中显示的文档的 URL。如果省略这个参数，或者它的值是空字符串，那么新窗口就不显示任何文档。

name

一个可选的字符串，其中包括数字、字母和下划线，该字符串声明了新窗口的名字。这个名字可以作为标记 `<a>` 和 `<form>` 的性质 `target` 的值。如果这个参数指定一个已经存在的窗口，那么 `open()` 方法就不再创建一个新窗口，而只返回对指定窗口的引用。在这种情况下，参数 *features* 将被忽略。

features

一个字符串，声明了新窗口要显示的标准浏览器窗口的特征。在“Window features”中对这个字符串格式进行了详细说明。该参数也是可选的，如果省略了它，新的窗口将具有所有标准特征。

replace

一个可选的布尔参数，声明了是在窗口的浏览历史中给装载到新页面的 URL 创建一个新条目，还是用它替换浏览历史中的当前条目。如果这个参数的值为 `true`，那么就不创建新的历史条目。这个参数是在 JavaScript 1.1 中添加进来的。注意，对新创建的窗口使用这一参数没有太大的意义，它主要在改变已经存在的窗口的内容时使用。

返回值

对 Window 对象的引用，这个 Window 对象可能是新创建的，也可能是已经存在的，这取决于参数 *name*。

描述

方法 `open()` 将查找一个已经存在的窗口或者打开一个新的浏览器窗口。如果参数 *name* 声明了一个已经存在的窗口，那么就返回对那个窗口的引用。返回的窗口将显

示参数 *url* 指定的文档,但忽略参数 *features*。在只知道窗口名字的情况下,这是 JavaScript 获得对那个窗口的引用的惟一方式。

如果没有指定参数 *name*,或者它指定的窗口不存在,那么 *open()* 方法将创建一个新的浏览器窗口。这个新窗口将显示参数 *url* 指定的 URL,它的名字由 *name* 指定,大小以及控件都由 *features* 参数指定(下面对该参数的格式进行了详细的说明)。如果 *url* 是一个空串,那么 *open()* 将打开一个空白窗口。

参数 *name* 声明了新窗口的名字,这个名字中只能出现数字、字母或下划线。它可以作为标记 `<a>` 和 `<form>` 的性质 *target* 的值,用来迫使文档在这个指定的窗口中显示。

在 JavaScript 1.1 中,当使用方法 *Window.open()* 给指定的窗口装载新文档时,可以给它传递参数 *replace*,用来声明新文档是在窗口的浏览历史中拥有自己的条目,还是替换当前文档的条目。如果 *replace* 的值为 *true*,新文档就会替换旧文档。如果它的值为 *false*,或者省略,那么新文档会在窗口的浏览历史中拥有自己的条目。这个参数提供的功能与方法 *Location.replace()* 提供的功能非常相似。

不要混淆了方法 *Window.open()* 与方法 *Document.open()*,这两者的功能完全不同。要使你的代码清楚明白,最好使用 *Window.open()*,而不要使用 *open()*。在定义为 HTML 性质的事件处理程序中,通常把函数 *open()* 解释为 *Document.open()*,因此,在这种情况下,就必须使用 *Window.open()*。

窗口特征

参数 *features* 是一个窗口要显示的特征列表,其中各个特征之间用逗号分隔。如果这个可选参数的值是为空,或者它被省略了,那么窗口将显示出所有特征。不过,如果 *features* 声明了某个特征,那么在这个列表中没有出现的特征就不会在窗口中显示出来。要注意的是,这个字符串不含任何空格或空白符,其中每个元素的格式如下所示:

```
feature[=value]
```

对于大多数的特征来说, *value* 的值是 *yes* 或 *no*。这些特征后的等号和 *value* 值都可以省略,如果出现了该特征,就假定它的 *value* 值为 *yes*,如果没有出现,就假定 *value* 值为 *no*。不过,特征 *width* 和 *height* 的 *value* 值是必需的,一定要指定它们的像素值。

可用的特征和它们的意义如下:

`channelmode`

指定窗口是否应该以信道的模式显示。只在 IE 4 中可用。

`dependent`

如果设置成“no”，即声明了新窗口和当前窗口的子窗口无关。只在 Netscape 4 中可用。

`directories`

指南按钮，如“What's New”和“What's Cool”。只在 Netscape 中可用。

`fullscreen`

指定窗口是否应该以全屏的模式显示。只在 IE 4 中可用。

`height`

指定窗口的文档显示区的高度，以像素计。

`innerHeight`

指定窗口的文档显示区的高度，以像素计。只在 Netscape 4 中可用。

`innerWidth`

指定窗口的文档显示区的宽度，以像素计。只在 Netscape 4 中可用。

`left`

窗口的 X 坐标，以像素计。只在 IE 4 中可用。Netscape 使用的是 `screenX`。

`location`

直接在浏览器中输入 URL 的输入域。

`menubar`

菜单栏。

`outerHeight`

指定整个窗口的高度，以像素计。只在 Netscape 4 中可用。

`outerWidth`

指定整个窗口的宽度，以像素计。只在 Netscape 4 中可用。

`resizable`

如果没有出现该特征，或者设置为 no，那么窗口就没有调整自己的边界的操作

(根据平台的不同, 用户仍然可以使用其他的方法来调整窗口大小)。注意, 一个常见的错误是将这个特征误写为 “resizable”, 即多写了一个 “e”。

`screenX`

窗口的 X 坐标, 以像素计。只在 Netscape 4 中可用。IE 4 使用的是 `left`。

`screenY`

窗口的 Y 坐标, 以像素计。只在 Netscape 4 中可用。IE 4 使用的是 `top`。

`scrollbars`

在必要的情况下激活水平滚动条和垂直滚动条。

`status`

状态栏。

`toolbar`

浏览器的工具栏, 具有 **Back** 按钮和 **Forward** 按钮等。

`top`

窗口的 Y 坐标, 以像素计。只在 IE 4 中可用。Netscape 中使用的是 `screenY`。

`width`

指定窗口的文档显示区的宽度, 以像素计。

参阅

`Location.replace()`、`Window.close()`、`Window` 对象的 `closed` 属性和 `opener` 属性

`Window.print()`

Netscape 4, Internet Explorer 5

打印文档

摘要

`window.print()`

描述

调用 `print()` 方法可以打印出当前文档, 就像用户点击了浏览器的 **Print** 按钮一样。

Window.prompt()

JavaScript 1.0

获取对话框中输入的字符串

摘要

```
window.prompt(message, default)
```

参数

message

要在对话框中显示的纯文本（而不是HTML格式的文本）。它要求用户输入你想要得到的信息。

default

作为对话框中默认输入的字符串。要使prompt()显示一个空的输入框，就将这个参数设置为空串（""）。

返回值

用户输入的字符串，如果用户没有输入任何字符串，那么返回的就是空串，如果用户点击了 **Cancel** 按钮，返回的就是 `null`。

描述

方法prompt()将用一个对话框显示出指定的消息message，这个对话框中含有文本输入域、**OK** 按钮、**Clear** 按钮和 **Cancel** 按钮，其中由平台决定的图形向用户说明了需要他进行输入。

如果用户点击了 **Cancel** 按钮，prompt()将返回 `null`。如果用户点击了 **Clear** 按钮，prompt() 则会把输入域中显示的当前文本都擦掉。如果用户点击了 **OK** 按钮，prompt()将返回输入域中当前显示的值。

由prompt()方法显示的对话框是有模式的，也就是说，在用户点击**OK**按钮或**Cancel**按钮来关闭它之前，它将阻塞用户对浏览器窗口的所有输入。由于该方法返回的值由用户的响应决定，所以在调用prompt()时，将暂停对JavaScript代码的执行。在用户作出响应之前，不会执行下一条JavaScript语句。

参阅

Window.alert()、Window.confirm()

Window.releaseEvents()

Netscape 4

停止捕捉事件

摘要

```
window.releaseEvents(eventmask)
document.releaseEvents(eventmask)
layer.releaseEvents(eventmask)
```

参数

eventmask

一个整数，声明了窗口、文档或图层要停止捕捉的事件类型。这个值是 `Event` 类定义的静态事件类型常量中的一个，或是用按位或运算符 (`|`) 或加号组合在一起的事件类型常量组。

描述

`releaseEvents()` 既是 `Window` 对象的方法，又是 `Document` 对象和 `Layer` 对象的方法，它指定的操作恰好与这些类的 `captureEvents()` 方法相反。在 Netscape 4 的事件模型中，它声明了 `window`、`document` 或 `layer` 不能再捕捉的事件类型，这些类型由参数 `eventmask` 指定。要查阅 `eventmask` 可以使用的常量的完整列表，请参阅“`Event.TYPE`”。

参阅

`Event`、`Window.captureEvents()`、`Window.handleEvent()`、`Window.routeEvent()`；第十九章；DOM 参考手册部分的 `EventTarget.removeEventListener()`

Window.resizeBy()

JavaScript 1.2

将窗口大小调整一个相对的数量

摘要

```
window.resizeBy(dw, dh)
```

参数

dw 要使窗口宽度增加的像素数。

dh 要使窗口高度增加的像素数。

描述

方法 `resizeBy()` 将把窗口 `window` 的宽度和高度分别调整 `dh` 和 `dw` 个像素。出于安全性的原因，浏览器会限制该方法，使窗口的宽度和高度不能小于 100 像素。

Window.resizeTo()

JavaScript 1.2

调整窗口大小

摘要

```
window.resizeTo(width, height)
```

参数

width

想要调整到的窗口宽度。

height

想要调整到的窗口高度。

描述

方法 `resizeTo()` 将调整窗口 `window` 的大小，把它的宽度调整为 `width` 个像素，把它的高度调整为 `height` 个像素。出于安全性的原因，浏览器限制该方法，使窗口的宽度和高度不能小于 100 像素。

Window.routeEvent()

Netscape 4

把捕捉到的事件传递给下一个处理程序

摘要

```
window.routeEvent(event)
```

```
document.routeEvent(event)
```

```
layer.routeEvent(event)
```

参数

event

要传递给下一个事件处理程序的 Event 对象。

返回值

event 传递给处理程序的返回值。

描述

`routeEvent()` 既是 `Window` 类的方法，又是 `Document` 类和 `Layer` 类的方法，它在这三种类中的行为都一样。当把一个 `Event` 对象 (`event`) 传递给 `window`、`document` 或 `layer` 的一个事件处理程序时，该处理程序可以选择将这个事件传递给下一个对它感兴趣的程序（如果存在这样的处理程序）。如果窗口、文档或层中又含有窗口、文档或层，而且它们也使用 `captureEvents()` 方法注册了感兴趣的事件类型，那么事件就会传递给这些窗口、文档或层对象中合适的处理程序。

不过，如果没有包容的窗口、文档或层表示过对这种事件感兴趣，`routeEvent()` 就会把这个 `event` 对象传递给产生事件的那个对象的事件处理程序。`captureEvents()` 方法和 `routeEvent()` 方法共同构成了 Netscape 4 的“前移”事件模型的基础。

参阅

`Window.captureEvents()`、`Window.handleEvent()`、`Window.releaseEvents()`；第十九章

`Window.scroll()`

JavaScript 1.1; JavaScript 1.2 反对使用它

滚动窗口中的文档

摘要

`window.scroll(x, y)`

参数

x 要滚动到的 X 坐标。

y 要滚动到的 Y 坐标。

描述

方法 `scroll()` 将在窗口中移动它的文档，使指定的文档 *x*、*y* 坐标出现在窗口的左上角。

X 坐标是从左到右递增的，Y 坐标是从上到下递增的。因此，`scroll(0, 0)` 就是将文档的左上角放置到窗口的左上角。

在 JavaScript 1.2 中，建议使用方法 `scrollTo()` 和 `scrollBy()` 来代替方法 `scroll()`。

Window.scrollBy()JavaScript 1.2

将文档滚动一个相对的数量

摘要

```
window.scrollBy(dx, dy)
```

参数

dx 把文档向右滚动的像素数。

dy 把文档向下滚动的像素数。

描述

方法 `scrollBy()` 将把窗口中显示的文档向右滚动 *dx* 个像素，向下滚动 *dy* 个像素。

Window.scrollTo()JavaScript 1.2

滚动文档

摘要

```
window.scrollTo(x, y)
```

参数

x 要在窗口的文档显示区的左上角显示的文档的 X 坐标。

y 要在窗口的文档显示区的左上角显示的文档的 Y 坐标。

描述

方法 `scrollTo()` 将滚动窗口 *window* 中显示的文档，使 *x* 和 *y* 指定的文档中的点显示在窗口的左上角（如果可能）。

`scrollTo()` 所做的与方法 `Window.scroll()` 所做的一样，后者是 JavaScript 1.1 使用的，它的名字没有前者那么具有描述性。因此建议使用 `scrollTo()` 方法而不是 `Window.scroll()` 方法。

Window.setInterval() JavaScript 1.2; Internet Explorer 4 只支持该方法的两种形式中的一种
周期性地执行指定的代码

摘要

```
window.setInterval(code, interval)  
window.setInterval(func, interval, args...)
```

参数

code

要周期性执行的 JavaScript 代码串。如果这个字符串含有多个语句，那么每个语句之间都用分号分隔开。

func

要周期性执行的 JavaScript 函数。这种形式在 IE 4 中不可用。

interval

一个整数，声明了执行 *code* 或调用 *func* 之间的时间间隔，以毫秒计。

args...

任意多个值，是函数 *func* 的参数。

返回值

一个可以传递给 `Window.clearInterval()` 从而取消对 *code* 或 *func* 的周期性执行的值。

描述

方法 `setInterval()` 可以使字符串 *code* 指定的 JavaScript 代码周期性地执行，每次执行的间隔为 *interval* 毫秒。

在 Netscape 4 中，可以将一个函数作为第一个参数（而不是字符串）传递给该方法。在这种形式的 `setInterval()` 中，每隔 *interval* 毫秒，函数 *func* 就会被调用。除了上述参数之外，其他所有的 *args* 参数都将作为调用 `func()` 时的参数。

无论是哪种形式的 `setInterval()` 方法，它返回的值此后都可以用做方法 `Window.clearInterval()` 的参数，用来停止对 *code* 或 *func* 的反复执行。

`setInterval()` 与 `set.Timeout()` 有关。当你想延迟代码的执行，但不想重复执行时，可以使用 `setTimeout()`。

参阅

`Window.clearInterval()`; `Window.setTimeout()`

`Window.setTimeout()`

JavaScript 1.0

延迟代码的执行

摘要

`window.setTimeout(code, delay)`

参数

code

一个字符串，存放的是在延迟了 *delay* 时间后要执行的 JavaScript 代码。

delay

在执行 *code* 中的 JavaScript 语句之前要延迟的时间，以毫秒计。

返回值

一个不透明的值，可以传递给 `clearTimeout()` 方法用来取消对 *code* 的执行。

描述

方法 `setTimeout()` 可以把字符串 *code* 中含有的 JavaScript 语句的执行延迟 *delay* 毫秒。当经过了指定的毫秒数之后，就会正常执行 *code* 中的语句。注意，这些语句只被执行一次。要反复地执行这些语句，*code* 自身必须含有对 `setTimeout()` 的调用，这样才能把它自己注册为可以再次执行。在 JavaScript 1.2 中，可以使用 `window.setInterval()` 将代码注册为周期性执行。

字符串 *code* 中的语句可以在 *window* 环境中执行。这里 *window* 指的是当前窗口。如果 *code* 中含有多个语句，那么各个语句之间用分号分隔开。

参阅

`Window.clearTimeout()`; `Window.setInterval()`

Window.status

JavaScript 1.0

声明 一条瞬时的状态栏消息

摘要

`window.status`

描述

属性 `status` 是一个可读可写的字符串，声明了要在窗口的状态栏中显示的一条瞬时消息。通常显示这条消息的时间是有限的，直到其他的消息将它覆盖了，或者用户把鼠标移动到窗口中的其他区域为止。当擦除了 `status` 声明的消息时，状态栏要么是恢复为它默认的空白状态，要么是再次显示出属性 `defaultStatus` 声明的默认消息。

虽然只有顶层窗口才有状态栏，但是框架的 `status` 属性也是可以设置的。这样做也会在顶层窗口的状态栏中显示出指定的消息。无论当前哪一个框架具有焦点，也无论鼠标在哪一个框架中，由框架设置的瞬时消息都是可见的。这与属性 `defaultStatus` 的行为有所不同。

习惯用法

属性 `status` 常用于在状态栏中显示瞬时消息。要显示半永久性的消息，最好使用属性 `defaultStatus`。

一般说来，只有在事件处理程序中或者方法 `Window.setTimeout()` 延迟的代码段中使用 `status` 属性才是有用的。如果从脚本中直接设置 `status` 属性，那么消息是不会显示给用户的。因为它并不会立刻显示出来，而且当它显示出来时很可能立刻就被诸如 “Document: done” 这样的浏览器消息覆盖了。

如果想在 一个超文本链接的 `onmouseover()` 事件处理程序中设置 `status` 属性，就必须让那个处理程序返回 `true`。这是因为在默认情况下，当鼠标移动过一条链接时就显示那个链接的 URL，这样就覆盖了事件处理程序设置的所有状态消息。让该事件处理程序返回 `true`，可以取消这种默认的动作，使自己的 `status` 消息显示出来（直到鼠标离开这条链接为止）。

参阅

`Window.defaultStatus`

Window.stop()Netscape 4

停止装载文档

摘要`window.stop()`**描述**

调用 `stop()` 方法可以停止浏览器装载当前文档，就像用户点击了浏览器的 **Stop** 按钮一样。

第五部分

W3C DOM 参考手册

本书的这一部分详细介绍了W3C DOM的JavaScript实现支持的所有对象、属性、函数、方法和事件处理程序。这一部分首先介绍了如何使用该参考手册。

W3C DOM 参考手册

本部分是一个参考手册，说明了W3C的1级和2级DOM标准定义的接口、方法和属性。为最新一代遵守标准的Web浏览器编写程序的中高级程序设计者，需要结合第三部分和四部分的JavaScript核心参考手册和客户端JavaScript参考手册来使用这一部分。引言和示例参考页解释了如何最大限度地利用这个参考手册。这个参考手册和前两个参考手册有极大的不同。你应该仔细阅读引言，以便能完全理解它包含的参考信息。

与核心参考手册和客户端参考手册一样，这个参考手册的条目按照字母顺序排列。DOM接口的属性和方法的参考页都以它们全称的字母顺序排列，包含定义它们的接口的名字。例如，如果要阅读Node接口的`appendChild()`方法，应该在“Node.appendChild”下查看，而不是“appendChild”。

为了节省本书的篇幅，这个参考手册中的大多数属性没有自己的参考页（但所有接口和方法都有自己的参考手册页）。每个属性在定义它们的接口的参考页中都有完整的说明。例如，可以在Element参考页中读到Element接口的`tagName`属性的说明。

有时，你不知道想查找的方法或属性所属的接口名，或者不确定应该在哪个参考手册中查找一个类或接口。本书的第六部分是一个专用的索引，用来解决这些问题。查找一个类、方法或属性的名字，它将告诉你应该在哪个参考手册中查找，以及在哪个参考手册的哪个类下。例如，如果查找“Document”，它将告诉你客户端参考手册和DOM参考手册中都有名为Document的条目。如果查找“firstChild”，它将告诉你，firstChild是Node的属性，在DOM参考手册部分可以找到它。

一旦找到了要查询的参考页，找到你需要的信息就没有太大的困难了。因为DOM标准适用于JavaScript以外的语言，所以要记住，它是用强类型语言（如Java和C++）编写的。虽然JavaScript是无类型语言，但该标准定义的属性和方法的类型信息仍然相当有用，这些信息也包含在这个参考手册中。这意味着，这个部分的方法和属性的摘要使用的语法更像Java，而不像JavaScript。接下来是一个示例参考页，标题为“示例条目”，它说明了每个参考页的结构，告诉你在哪里可以找到什么类型的信息。即使你对本书的第三版已经相当熟悉，但在研究DOM参考手册的其余部分之前，还是应该花时间阅读该参考页。

示例条目

可用性

如何阅读 DOM 参考手册

继承性

标题和简短描述

每个参考条目都以一个四部分的标题块开头，如上所示。各个条目按照标题的字母顺序排列。标题行下的简短说明概述了该条目中记述的项目，它可以帮助你快速地判断是否需要继续阅读该参考页余下的部分。

可用性

标题块的右上角是可用性信息。它将告诉你什么级别的DOM标准和它的哪个模块定义了该接口或方法。因为属性没有自己的参考页，所以它们没有可用性信息。如果属性的可用性不同于定义它的接口，会在属性的说明中提出这一事实。

继承性

DOM接口可以继承其他接口的属性和方法。如果DOM接口继承了另一个接口，继承层次将显示在标题块的右下角。例如，HTMLElement接口的继承性信息如下所示：

Node → Element → HTMLElement

这说明HTMLElement继承了Element接口，Element接口继承了Node接口。当你看到这部分信息时，可能需要查看列出的其他接口。

子接口

这个部分是与继承性信息相反的信息，它列出了继承该接口的所有接口。例如，Element参考页的子接口部分将说明HTMLElement是Element的子接口，继承Element的方法和属性。

也实现

DOM 标准的模块结构意味着有些接口被分割成了多个独立的接口,以便某些实现只能执行它们支持的那部分模块的接口。一个对象除了实现一个接口(如 `Document`)外,还实现其他几个简单的接口(如 `DocumentCSS`、`DocumentEvent` 和 `DocumentView`),以提供其他模块特有的功能,这很常见。当接口具有要与它一起实现的次级接口时,这个部分列出了次级接口。

常量

有些 DOM 接口定义了常量集合,作为那个接口的属性值或方法的参数值。例如, `Node` 接口定义了重要的常量,作为所有 `Document` 节点的 `nodeType` 属性的合法值集合。当接口定义了常量时,这一部分列出并说明了这些常量。列表其中包括每个常量的类型、名字和值。“DOM 类型”部分讨论了这些列表使用的语法。注意,常量是接口自身的静态属性,不是接口的实例。

属性

如果参考页说明的是接口,那么用这一部分列出并说明了该接口定义的属性。列表中的条目包括属性名和类型,还包括提供有关属性的其他信息的关键字。注意,在 Java 型的语法中,属性名放在最后,属性名之前的所有信息提供了属性的类型及其他信息。例如, `HTMLTableElement` 和 `HTMLTableCellElement` 接口定义的属性如下:

`HTMLTableCaptionElement caption`

`caption` 属性,引用一个类型为 `HTMLTableCaptionElement` 的对象。

`readonly HTMLCollection rows`

`rows` 属性,引用一个 `HTMLCollection` 对象,它是只读的,即可以查询该属性的值,但不能设置它。

`deprecated String align`

`align` 属性,是一个字符串,但反对使用它,不鼓励采用它。

`readonly long cellIndex`

`cellIndex` 属性,是一个长整值(参阅“DOM 类型”部分),它是只读的。

方法

如果参考页说明的是接口,这个部分列出了接口的方法名,提供了方法的简短说明。在每个方法的参考页中可以找到它的完整说明。

摘要

如果参考页说明的是方法，这个部分说明了方法的特征或摘要。它使用Java型的语法进行说明（按顺序）：

- 方法返回值的类型。如果方法什么都不返回，则值为 `void`。
- 方法的名字。
- 方法每个参数的类型和名字。这是参数类型和名字对的列表，中间用逗号分隔，该列表用括号括起来。如果方法没有参数，那么只能看到括号（）。
- 方法抛出的异常的类型（如果存在）。

例如，`Node.insertBefore()` 方法的“摘要”部分如下所示：

```
Node insertBefore(Node newChild,  
                  Node refChild)  
    throws DOMException;
```

从这个摘要中可以得到下列信息。方法的名字是“insertBefore”。它返回 `Node` 对象。第一个参数是 `Node` 对象，名为“newChild”（可假定为要插入的节点）。第二个参数也是 `Node` 对象，名为“refChild”（可假定为位于要插入节点之前的节点）。此外，在某些情况下，该方法可能抛出类型为 `DOMException` 的异常。

摘要后的子部分提供了方法的参数、返回值和异常等其他信息。关于声明方法参数的类型所采用的Java型语法，请参阅“DOM 类型”部分。

参数

如果方法具有参数，“摘要”部分后的“参数”子部分列出了参数名，并说明了每个参数。注意，参数名用斜体字列出，表示不必逐字输入它们，而可以用其他值或JavaScript表达式代替它。继续采用上面的例子，`Node.insertBefore()` 方法的“参数”部分如下：

newChild

要插入树的节点。如果它是 `DocumentFragment` 接口，则可以插入它的子节点。

refChild

*newChild*要插入的节点之前的节点。如果该参数是 `null`，*newChild*将作为该节点的最后一个子节点插入。

返回值

“摘要”部分声明了方法返回值的数据类型，“返回值”子部分提供了更多的信息。如果方法没有返回值（即在“摘要”部分列出它返回 void），这部分将被省略。

抛出

这个部分解释了方法抛出的异常类型，以及在什么情况下它抛出异常。

DOM 类型

DOM 参考页采用 Java 型的语法声明常量、属性、方法返回值和方法参数的类型。这个部分提供了更多关于该语法的信息。注意，参考页自身没有“DOM 类型”部分。

通用语法是：

```
modifiers type name
```

常量、属性、方法或方法参数的名字通常放在最后，之前是类型和其他信息。这个参考手册部分中采用的修饰符如下（注意，它们不是合法的 Java 修饰符）：

`readonly`

声明只能查询属性值，不能设置属性值。

`deprecated`

声明反对使用该属性，应该避免使用它。

`unsigned`

声明数字常量、属性、返回值或方法参数是无符号的，即它只能是 0 或正数，不能是负数。

DOM 常量、属性、方法返回值和方法参数的类型不一定直接对应于 JavaScript 支持的类型。例如，有些属性的类型为 `short`，是 16 位的整数。虽然 JavaScript 只有一种数字类型，但这个参考部分使用 DOM 类型，因为它提供了合法数字范围的更多信息。在这个参考部分中可能遇到的 DOM 类型如下：

`String`

JavaScript 核心语言的 `String` 对象。

`Date`

JavaScript 核心语言的 `Date` 对象（不常用）。

boolean

布尔值，true 或 false。

short

16 位短整数，这种类型可以应用 unsigned 修饰符。

long

64 位长整数，这种类型可以应用 unsigned 修饰符。

float

浮点数，这种类型没有 unsigned 修饰符。

void

这种类型只适用于方法的返回值，它说明方法不返回任何值。

其他类型

在这个参考部分中见到的其他类型是其他 DOM 接口的名字（例如，Document、DOMImplementation、Element、HTMLTableElement 和 Node）。

描述

大部分参考页都具有“描述”部分，是对要说明的接口或方法的基本描述。它是参考页的核心。如果你是第一次学习接口或方法，可以直接跳到这个部分，然后返回查看前面的“摘要”、“属性”和“方法”部分。如果你已经熟悉了接口或方法，可以不必再阅读这一部分，只需要快速查看某些特定的信息（如从“属性”或“参数”部分查找属性的名字或参数的类型）即可。

在某些参考页中，这一部分只是一小段。但是在某些参考页中，这一部分可能会占用一页的篇幅，甚至会更多。对于那些非常简单的方法，“参数”和“返回值”以及“抛出”部分已经足够说明该方法了，所以它的“描述”部分就被省略了。

示例

有些常用的接口和方法的参考页还包括一个示例，用来说明该接口或方法的典型用法。但大部分参考页没有示例部分，可以在本书的前半部分找到它们的示例。

参阅

许多参考页的结尾都是对相关参考页的交叉引用。这些交叉引用引用的大部分是 DOM 参考手册中的参考页，也有一些是对接口参考页中包含的个别属性说明的引用。

其余的是对客户端参考手册中的相关参考页的引用或对本书前两部分中的章节的引用。

说明接口（不是说明方法）的参考页在“参阅”部分后还可以包括额外的段落。这些是交叉引用，说明如何使用该接口。“Type of”段列出了值为实现接口的对象的属性。“Passed to”段列出了参数为实现接口的方法。“Returned by”段列出了返回值是实现接口的对象的方法。这些交叉引用说明了如何获取接口的对象，以及得到它后可以用它做什么。

AbstractView

2 级 DOM 视图

显示文档的窗口

也实现

ViewCSS

如果DOM实现支持CSS模块，那么实现AbstractView的接口也实现了ViewCSS接口。为了方便起见，ViewCSS接口定义的方法在“方法”部分列出。

属性

readonly Document document

View对象显示的Document对象。这个Document对象还实现了DocumentView接口。

方法

getComputedStyle()*[2 级 DOM CSS]*

这个ViewCSS方法返回一个只读的CSSStyleDeclaration对象，表示指定文档元素的计算样式信息。

描述

在DOM中，视图（view）是以某种方式显示文档的对象。客户端JavaScript的Window对象就是这样的视图。AbstractView接口是向标准化Window对象的某些属性和方法迈出的第一步。它只声明了所有View对象具有名为document的属性，该属性引用了它显示的文档。此外，如果一个实现支持CSS样式表，那么所有View对象都会实现ViewCSS接口，并定义了getComputedStyle()方法，以判断某个元素在该视图中是否真的被渲染了。

`document` 属性给每个视图提供了对它显示的文档的引用。反之亦然，每个文档都有对显示它的视图的引用。如果一个 DOM 实现支持 View 模块，实现 Document 接口的实现还会实现 DocumentView 接口。这个 DocumentView 接口定义了 `defaultView` 属性，该属性引用了显示文档的窗口。

接口名字中的“Abstract”强调它只是标准化的窗口接口的开端。为了使它更加有用，将来的 DOM 标准必须引入新的扩展的 AbstractView 接口，添加其他的属性和方法。

参阅

`Document.defaultView`

Type of: `Document.defaultView`

AbastractView.getComputedStyle()

2 级 DOM CSS

获取用于渲染一个元素的 CSS 样式

摘要

```
CSSStyleDeclaration getComputedStyle(Element elt,  
                                     String pseudoElt);
```

参数

elt

想获取样式信息的文档的元素。

pseudoElt

CSS 伪元素，如果没有这样的元素则为 `null`。

返回值

一个只读的 `CSSStyleDeclaration` 对象（通常还实现了 `CSS2Properties` 接口），声明了用于渲染该视图中的指定元素的样式信息。从这个对象中查询到的所有长度值都是绝对值或像素值，而不是相对值或百分比。

描述

文档中的元素可以从内联 `style` 性质或样式表级联的样式表中获取样式信息。在视图真正显示该元素前，必须通过从适当的级联部分提取样式信息，计算它的样式。

用该方法可以访问计算出的样式。与元素的 `style` 属性进行相比，用它只能访问元素

的内联样式，不能告诉你任何有关应用到元素的样式表性质的信息。注意，用该方法还可以确定元素在视图中被渲染的像素的坐标。

`getComputedStyle()`实际上是由 `ViewCSS` 接口定义的。在支持 `View` 和 `CSS` 模块的 DOM 实现中，实现了 `AbstractView` 接口的对象都实现了 `ViewCSS` 接口。所以为了简便起见，该方法列在 `AbstractView` 接口中。

在 Internet Explorer 中，相似的函数还可以通过每个 `HTMLElement` 对象的 `currentStyle` 属性来使用。

参阅

`CSS2Properties`、`CSSStyleDeclaration`、`HTMLElement.style`

Attr	1 级核心 DOM
文档元素的性质	Node → Attr

属性

`readonly String name`
性质的名字。

`readonly Element ownerElement` [2 级 DOM]
包含该性质的 `Element` 对象，如果 `Attr` 对象当前没有关联到任何 `Element` 对象，值为 `null`。

`readonly boolean specified`
如果文档源代码中明确声明了该性质，或某个脚本设置了该性质，值为 `true`。
如果没有明确声明该性质，但在文档的 DTD 中设置了默认值，值为 `false`。

`String value`
性质的值。在读该属性时，性质值将以字符串形式返回。把该属性设置成一个字符串时，它会自动创建一个 `Text` 节点，以存放该文本，并使这个 `Text` 节点成为 `Attr` 对象的惟一子节点。

描述

`Attr` 对象表示 `Element` 节点的性质。`Attr` 对象和 `Element` 节点相关，但不直接属于文档树（并具有 `null parentNode` 属性）。可以通过 `Node` 接口的 `attributes` 属性或调用 `Element` 接口的 `getAttributeNode()` 方法，获取 `Attr` 对象。



Attr 对象是节点，它的值由 Attr 节点的子节点表示。在 HTML 文档中，它只是一个 Text 节点。但在 XML 文档中，Attr 节点可以有 Text 和 EntityReference 两个子节点。value 属性为读写性质的字符串值提供了快捷方式。

在大多数情况下，使用元素性质的最简单方法是 Element 接口的 `getAttribute()` 方法和 `setAttribute()` 方法。这两个方法将字符串作为性质名和性质值，避免了使用 Attr 节点。

参阅

Element

Passed to: `Element.removeAttributeNode()`、`Element.setAttributeNode()`、`Element.setAttributeNodeNS()`

Returned by: `Document.createAttribute()`、`Document.createAttributeNS()`、`Element.getAttributeNode()`、`Element.getAttributeNodeNS()`、`Element.removeAttributeNode()`、`Element.setAttributeNode()`、`Element.setAttributeNodeNS()`

CDATASection

1 级 DOM XML

XML 文档中的 CDATA 节

Node → CharacterData → Text → CDATASection

描述

这个常用的接口表示 XML 文档中的 CDATA 节。使用 HTML 文档的程序设计者绝不会遇到这种类型的节点，不需要使用该接口。

CDATASection 是 Text 接口的子接口，没有定义任何自己的属性和方法。通过从 Node 接口继承 `nodeValue` 属性，或通过从 CharacterData 接口继承 `data` 属性，可以访问 CDATA 节的文本内容。虽然通常可以把 CDATASection 节点作为 Text 节点处理，但要注意，`Node.normalize()` 方法不并入相邻的 CDATA 节。

参阅

CharacterData、Text

Returned by: `Document.createCDATASection()`

CharacterData

1 级核心 DOM

Text 和 Comment 节点的常用功能

Node → CharacterData

子接口

Comment、Text

属性

String data

该节点包含的文本。

readonly unsigned long length

该节点包含的字符数。

方法

appendData()

把指定的字符串添加到该节点包含的文本上。

deleteData()

从该节点删除指定的文本, 从指定位移量处的字符开始, 包括其后指定数量的字符。

insertData()

把指定的字符串插入指定位移量处的节点文本。

replaceData()

用指定的字符串替换从指定位移量处开始, 包括其后指定数量的字符。

substringData()

返回从指定位移量处的字符开始, 包括其后指定数量的字符的文本的副本。

描述

CharacterData 是 Text 节点和 Comment 节点的超接口。文档从不包含 CharacterData 节点, 它们只包含 Text 节点和 Comment 节点。但由于这两种节点类型具有相似的功能, 因此此处定义了这些函数, 以便 Text 和 Comment 可以继承它。

参阅

Comment、Text

CharacterData.appendData()1 级核心 DOM

把字符串附加到 Text 或 Comment 节点上

摘要

```
void appendData(String arg)
```

```
    throws DOMException;
```

参数

arg

要附加到 Text 或 Comment 节点的字符串。

抛出

如果调用该方法的节点是只读的, 它将抛出具有代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 异常。

描述

该方法将把字符串 *arg* 附加到节点的 data 属性的末尾。

CharacterData.deleteData()1 级核心 DOM

从 Text 或 Comment 节点删除文本

摘要

```
void deleteData(unsigned long offset,
```

```
                unsigned long count)
```

```
    throws DOMException;
```

参数

offset

要删除的第一个字符的位置。

count

要删除的字符的数量。

抛出

该方法可以抛出具有以下代码的 DOMException 异常:

INDEX_SIZE_ERR

参数 *offset* 或 *count* 是负数，或 *offset* 大于 Text 节点或 Comment 节点的长度。

NO_MODIFICATION_ALLOWED_ERR

节点是只读的，不能修改。

描述

该方法将从 *offset* 指定的字符开始，从 Text 或 Comment 节点中删除 *count* 个字符。如果 *offset* 加 *count* 大于 Text 或 Comment 节点中的字符数，那么删除从 *offset* 开始到字符串结尾的所有字符。

CharacterData.insertData()

1 级核心 DOM

把字符串插入 Text 节点或 Comment 节点

摘要

```
void insertData(unsigned long offset,  
                String arg)  
    throws DOMException;
```

参数

offset

要把字符串插入 Text 节点或 Comment 节点的字符位置。

arg

要插入的字符串。

抛出

该方法可以抛出具有以下代码的 DOMException 异常：

INDEX_SIZE_ERR

参数 *offset* 是负数，或 *offset* 大于 Text 节点或 Comment 节点的长度。

NO_MODIFICATION_ALLOWED_ERR

节点是只读的，不能修改。

描述

该方法将把指定的字符串 *arg* 插入 Text 节点或 Comment 节点的文本的指定位置 *offset*。

CharacterData.replaceData()

1 级核心 DOM

用指定的字符串替换 Text 节点或 Comment 节点的字符

摘要

```
void replaceData(unsigned long offset,  
                unsigned long count,  
                String arg)  
    throws DOMException;
```

参数

offset

要替换的第一个字符在 Text 节点或 Comment 节点中的位置。

count

要替换的字符的数量。

arg

要替换 *offset* 和 *count* 指定的字符的字符串。

抛出

该方法可以抛出具有以下代码的 DOMException 异常：

INDEX_SIZE_ERR

参数 *offset* 是负数，或 *offset* 大于 Text 节点或 Comment 节点的长度，以及 *count* 是负数。

NO_MODIFICATION_ALLOWED_ERR

节点是只读的，不能修改。

描述

该方法将用字符串 *arg* 替换从 *offset* 开始的 *count* 个字符。如果 *offset* 加 *count* 大于 Text 或 Comment 节点的长度，那么从 *offset* 开始的所有字符都将被替换。

CharacterData.substringData()

1 级核心 DOM

从 Text 或 Comment 节点中提取子串

摘要

```
String substringData(unsigned long offset,  
                      unsigned long count)  
    throws DOMException;
```

参数

offset

要返回的第一个字符的位置。

count

要返回的子串中的字符数。

返回值

一个字符串，包含 Text 或 Comment 节点中从 *offset* 开始的 *count* 个字符。

抛出

该方法可以抛出具有以下代码的 DOMException 异常：

INDEX_SIZE_ERR

参数 *offset* 是负数，或 *offset* 大于 Text 节点或 Comment 节点的长度，以及 *count* 是负数。

DOMSTRING_SIZE_ERR

指定的文本范围太长，在浏览器的 JavaScript 实现中不能填到一个字符串中。

描述

该方法将从 Text 或 Comment 节点中提取从 *offset* 开始的 *count* 个字符。只有当节点包含的文本的字符数大于浏览器的 JavaScript 实现中能填入的字符串的最大字符数，该方法才有用。在这种情况下，JavaScript 程序不能直接使用 Text 节点或 Comment 节点的 *data* 属性，而必须用节点文本的较短子串。在实际应用中，这种情况不太可能出现。

Comment

1 级核心 DOM

HTML 或 XML 注释

Node → CharacterData → Comment

描述

Comment 节点表示 HTML 或 XML 文档中的注释。使用由 CharacterData 接口继承的 data 属性，或使用由 Node 接口继承的 nodeValue 属性，可以访问注释的内容（即 <!-- 和 --> 之间的文本）。使用由 CharacterData 接口继承的各种方法可以操作注释的内容。

参阅

CharacterData

Returned by: Document.createComment()

Counter

2 级 DOM CSS

CSS counter() 或 counters 规约

属性

readonly String identifier

计数器的名字。

readonly String listStyle

计数器的列表样式。

readonly String separator

嵌套计数器的分隔符字符串。

描述

该接口表示一个 CSS counter() 或 counters() 值。详见 CSS 参考页。

参阅

Returned by: CSSPrimitiveValue.getCounterValue()

CSS2Properties

2 级 DOM CSS2

所有 CSS2 性质的快捷属性

属性

该接口定义了大量属性, 每个属性对应 CSS2 规约定义的一个 CSS 性质。属性名与 CSS 的性质名紧密对应, 但为了避免 JavaScript 中的语法错误而进行了一些改变。含有连字符的多词性质 (如 font-family) 在 JavaScript 中没有连字符, 而是每个词的第一个字符大写 (如 fontFamily)。此外, float 性质与保留字 float 冲突, 所以被转换成 cssFloat。

下表列出了完整的属性集合。由于这些属性直接对应于 CSS 性质, 因此它们没有单独的说明。有关它们的含义和合法值, 请参阅 CSS 的参考书, 如《Cascading Style Sheets: The Definitive Guide》, 由 Eric A. Meyer 著, O'Reilly 公司出版。所有属性都是字符串。设置这些属性, 会抛出异常, 其原因与调用 CSSStyleDeclaration.setProperty() 相同。

azimuth	background	backgroundAttachment	backgroundColor
backgroundImage	backgroundPosition	backgroundRepeat	border
borderBottom	borderBottomColor	borderBottomStyle	borderBottomWidth
borderCollapse	borderColor	borderLeft	borderLeftColor
borderLeftStyle	borderLeftWidth	borderRight	borderRightColor
borderRightStyle	borderRightWidth	borderSpacing	borderStyle
borderTop	borderTopColor	borderTopStyle	borderTopWidth
borderWidth	bottom	captionSide	clear
clip	color	content	counterIncrement
counterReset	cssFloat	cue	cueAfter
cueBefore	cursor	direction	display
elevation	emptyCells	font	fontFamily
fontSize	fontSizeAdjust	fontStretch	fontStyle
fontVariant	fontWeight	height	left
letterSpacing	lineHeight	listStyle	listStyleImage
listStylePosition	listStyleType	margin	marginBottom
marginLeft	marginRight	marginTop	markerOffset
marks	maxHeight	maxWidth	minHeight
minWidth	orphans	outline	outlineColor
outlineStyle	outlineWidth	overflow	padding
paddingBottom	paddingLeft	paddingRight	paddingTop
page	pageBreakAfter	pageBreakBefore	pageBreakInside
pause	pauseAfter	pauseBefore	pitch
pitchRange	playDuring	position	quotes
richness	right	size	speak
speakHeader	speakNumeral	speakPunctuation	speechRate
stress	tableLayout	textAlign	textDecoration
textIndent	textShadow	textTransform	top
unicodeBidi	verticalAlign	visibility	voiceFamily

volume	whiteSpace	widows	width
wordSpacing	zIndex		

描述

该接口为 CSS2 规约定义的所有 CSS 性质都定义了相应的属性。如果 DOM 实现支持该接口（它是 CSS2 特性的一部分），那么所有 `CSSStyleDeclaration` 对象都会实现 `CSS2Properties` 接口。读取该接口定义的一个属性，等价于调用相应的 CSS 性质的 `getPropertyValue()` 方法。设置一个属性的值，等价于对相应的性质调用 `setProperty()` 方法。`CSS2Properties` 接口定义的属性包括对应于 CSS 快捷性质的属性，`CSS2Properties` 接口可以正确地处理这些快捷属性。

参阅

`CSSStyleDeclaration`

CSSCharsetRule

2 级 DOM CSS

CSS 样式表中的 @ charset 规则

`CSSRule` → `CSSCharsetRule`

属性

String encoding

@charset 规则指定的字符编码。如果把该属性设置成非法的值，将抛出代码为 `SYNTAX_ERR` 的 `DOMException` 异常。如果规则或样式表是只读的，设置该属性会抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

描述

该接口表示 CSS 样式表中的 @charset 规则。详见 CSS 参考书。

CSSFontFaceRule

2 级 DOM CSS

CSS 样式表中的 @font-face 规则

`CSSRule` → `CSSFontFaceRule`

属性

readonly `CSSStyleDeclaration` style

该规则的样式集合。

描述

该接口表示 CSS 样式表中的 @font-face 规则。详见 CSS 参考书。

CSSImportRule

2 级 DOM CSS

CSS 样式表中的 @import 规则

CSSRule → CSSImportRule

属性

`readonly String href`

导入的样式表的 URL。该属性的值不包括 URL 值外边的“url()”定界符。

`readonly MediaList media`

导入的样式表应用的媒体类型列表。

`readonly CSSStyleSheet styleSheet`

表示导入的样式表的 CSSStyleSheet 对象。如果还没有装载样式表，或由于媒体类型不适用而没有装载样式表，则值为 null。

描述

该接口表示 CSS 样式表中的 @import 规则。styleSheet 属性表示导入的样式表。

CSSMediaRule

2 级 DOM CSS

CSS 样式表中的 @media 规则

CSSRule → CSSMediaRule

属性

`readonly CSSRuleList cssRules`

@media 规则块中嵌套的所有规则的数组（从技术上说是 CSSRuleList 对象）。

`readonly MediaList media`

嵌套规则使用的媒体类型。

方法

`deleteRule()`

删除指定位置的嵌套规则。

`insertRule()`

在 @media 规则块的指定位置插入新规则。

描述

该接口表示 CSS 样式表中的 @media 规则 and 它所有的嵌套规则。用它定义的方法可以插入和删除嵌套规则。详见 CSS 参考书。

CSSMediaRule.deleteRule()2 级 DOM CSS

删除 @media 块中的规则

摘要

```
void deleteRule(unsigned long index)
    throws DOMException;
```

参数

index

要删除的规则在 @media 规则块中的位置。

抛出

该方法可能抛出具有下列代码的 DOMException 异常：

INDEX_SIZE_ERR

index 是负数，或者大于等于 `cssRules` 中的规则数。

NO_MODIFICATION_ALLOWED_ERR

规则是只读的。

描述

该方法将删除 `cssRules` 数组中指定位置的规则。

CSSMediaRule.insertRule()2 级 DOM CSS

在 @media 块中插入新规则

摘要

```
unsigned long insertRule(String rule,
                          unsigned long index)
    throws DOMException;
```

参数

rule

要添加的规则的 CSS 串表示，它是完整、可解析的。

`index`

把新规则插入 `cssRules` 数组中的位置，或 `cssRules.length` 把新规则附加在数组末尾。

返回值

参数 `index` 的值。

抛出

该方法可能抛出具有下列代码的 `DOMException` 异常：

`HIERARCHY_REQUEST_ERR`

CSS 语法不允许指定的位置有指定的 `rule`。

`INDEX_SIZE_ERR`

`index` 是负数，或大于 `cssRules.length`。

`NO_MODIFICATION_ALLOWED_ERR`

`@media` 规则和它的 `cssRules` 数组是只读的。

`SYNTAX_ERR`

指定的 `rule` 含有语法错误。

描述

该方法将把指定的 `rule` 插入 `cssRules` 数组的指定的 `index` 处。

CSSPageRule

2 级 DOM CSS

CSS 样式表中的 `@page` 规则

`CSSRule` → `CSSPageRule`

属性

`String selector Text`

该规则的页选择器文本。把该属性设置为非法值，将抛出代码为 `SYNTAX_ERR` 的 `DOMException` 异常。当该规则是只读规则时，设置该属性将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

`readonly CSSStyleDeclaration style`

该规则的样式集合。

描述

该接口表示 CSS 样式表中的 @page 规则，通常用于为打印设置页面布局。详情参阅 CSS 参考书。

CSSPrimitiveValue

2 级 DOM CSS

一个 CSS 样式值

CSSValue → CSSPrimitiveValue

常量

下面的常量是 primitiveType 属性的合法值，它们声明了值的类型，对于数字值，还声明了表示该值的单位。

```
unsigned short CSS_UNKNOWN = 0
```

该值没有得到认可，实现不知道如何解析它。使用 cssText 属性可以访问该值的文本表示。

```
unsigned short CSS_NUMBER = 1
```

无单位的数字，用 getFloatValue() 方法查询。

```
unsigned short CSS_PERCENTAGE = 2
```

百分比。用 getFloatValue() 方法查询。

```
unsigned short CSS_EMS = 3
```

以 em（当前字体的高度）计量的相对长度。用 getFloatValue() 方法查询。

```
unsigned short CSS_EXS = 4
```

以 ex（当前字体的 X-高度）计量的相对长度。用 getFloatValue() 方法查询。

```
unsigned short CSS_PX = 5
```

以像素计量的长度。用 getFloatValue() 方法查询。像素长度是相对的度量单位，因为它们的大小由显示器的分辨率决定，所以不能把它们转换成英寸、毫米、点或其他绝对长度。但像素是最常用的单位之一，例如，把它们作为 AbstractView.getComputedStyle() 使用的绝对值处理。

```
unsigned short CSS_CM = 6
```

以厘米计量的绝对长度。用 getFloatValue() 方法查询。

```
unsigned short CSS_MM = 7
```

以毫米计量的绝对长度。用 getFloatValue() 方法查询。

`unsigned short CSS_IN = 8`

以英寸计量的绝对长度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_PT = 9`

以点 (1/72 英寸) 计量的绝对长度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_PC = 10`

以 12 点活字打印机铅字的大小 (12 点) 计量的绝对长度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_DEG = 11`

以度计量的绝对角度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_RAD = 12`

以弧度计量的绝对角度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_GRAD = 13`

以梯度计量的绝对角度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_MS = 14`

以毫秒计量的时间长度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_S = 15`

以秒计量的时间长度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_HZ = 16`

以赫兹计量的频率。用 `getFloatValue()` 方法查询。

`unsigned short CSS_KHZ = 17`

以千赫计量的频率。用 `getFloatValue()` 方法查询。

`unsigned short CSS_DIMENSION = 18`

无单位维度。用 `getFloatValue()` 方法查询。

`unsigned short CSS_STRING = 19`

字符串。用 `getStringValue()` 方法查询。

`unsigned short CSS_URI = 20`

URI。用 `getStringValue()` 方法查询。

`unsigned short CSS_IDENT = 21`

标识符。用 `getStringValue()` 方法查询。

`unsigned short CSS_ATTR = 22`

性质函数。用 `getStringValue()` 方法查询。

`unsigned short CSS_COUNTER = 23`

计数器。用 `getCounterValue()` 方法查询。

`unsigned short CSS_RECT = 24`

矩形。用 `getRectValue()` 方法查询。

`unsigned short CSS_RGBCOLOR = 25`

颜色。用 `getRGBColorValue()` 方法查询。

属性

`readonly unsigned short primitiveType`

该值的类型。该属性存放的是前面定义的常量之一。

方法

`getCounterValue()`

对于 `CSS_COUNTER` 类型的值，返回表示该值的 `Counter` 对象。

`getFloatValue()`

返回一个数字值。如果必要，把它转换成指定的单位。

`getRectValue()`

对于 `CSS_RECT` 类型的值，返回表示该值的 `Rect` 对象。

`getRGBColorValue()`

对于 `CSS_RGBCOLOR` 类型的值，返回表示该值的 `RGBColor` 对象。

`getStringValue()`

返回一个字符串值。

`setFloatValue()`

把一个数字值设置为具有指定单位的指定数字。

`setStringValue()`

把一个字符串值设置为具有指定类型的指定字符串。

描述

CSSValue 的这个子接口表示单一的 CSS 值。把它与 CSSValueList 接口进行比较，后者表示 CSS 值的列表。这个接口名字中的单词 “primitive” 是个误导，因为它可以表示某些复合类型的 CSS 值，如计数器、矩形和颜色。

primitiveType 属性存放的是前面定义的常量之一，并声明了该值的类型。用这个接口定义的各种方法可以查询各种类型的值，还能设置数字值和字符串值。

参阅

Counter、CSSValue、CSSValueList、Rect、RGBColor

Type of: RGBColor.blue、RGBColor.green、RGBColor.red、Rect.bottom、Rect.left、Rect.right、Rect.top

CSSPrimitiveValue.getCounterValue()

2 级 DOM CSS

返回计数器的值

摘要

```
Counter getCounterValue()  
    throws DOMException;
```

返回值

表示这个 CSSPrimitiveValue 值的 Counter 对象。

抛出

如果 primitiveType 属性不是 CSS_COUNTER，该方法将抛出代码为 INVALID_ACCESS_ERR 的 DOMException 异常。

描述

该方法将返回表示 CSS 计数器的 Counter 对象。虽然没有对应的 setCounterValue() 方法，但通过设置返回的 Counter 对象的属性，可以修改这个值。

CSSPrimitiveValue.getFloatValue()2 级 DOM CSS

获取一个数字值，可以转换单位

摘要

```
float getFloatValue(unsigned short unitType)
    throws DOMException;
```

参数

unitType

声明返回值的单位的 **CSSPrimitiveValue** 类型常量。

返回值

这个 **CSSPrimitiveValue** 的浮点数字值，用指定的单位表示。

抛出

如果 **CSSPrimitiveValue** 存放的是非数字值，或这个值不能转换成要求的单位类型，该方法将抛出代码为 **INVALID_ACCESS_ERR** 的 **DOMException** 异常。（关于单位转换，参阅下一节。）

描述

对于存放数字值的 **CSSPrimitiveValue** 对象，该方法将把这些值转换成指定的单位，返回转换后的值。

只有某些类型单位允许进行转换。长度可以转换成长度，角度可以转换成角度，时间可以转换成时间，频率可以转换成频率。显然，以毫米计量的长度不能转换成以千赫计量的频率。不过，并非所有的长度能进行转换。相对长度（以 **em**、**ex** 或像素计量的长度）只能转换成其他的相对长度，不能转换成绝对长度。同样，绝对长度也不能转换成相对长度。最后要注意，百分比除了能转换成颜色百分比（表示 255 分之一，并可以转换成 **CSS_NUMBER** 类型）外，不能转换成其他单位类型。

CSSPrimitiveValue.getRectValue()2 级 DOM CSS

返回 Rect 值

摘要

```
Rect getRectValue()
    throws DOMException;
```

返回值

表示这个 `CSSPrimitiveValue` 值的 `Rect` 对象。

抛出

如果 `primitiveType` 属性不是 `CSS_RECT`，该方法将抛出代码为 `INVALID_ACCESS_ERR` 的 `DOMException` 异常。

描述

该方法将返回表示 CSS 矩形的 `Rect` 对象。虽然没有对应的 `setRectValue()` 方法，但通过设置返回的 `Rect` 对象的属性，可以修改这个值。

CSSPrimitiveValue.getRGBColorValue()

2 级 DOM CSS

获得 `RGBColor` 值

摘要

```
RGBColor getRGBColorValue()  
    throws DOMException;
```

返回值

表示这个 `CSSPrimitiveValue` 值的 `RGBColor` 对象。

抛出

如果 `primitiveType` 属性不是 `CSS_RGBCOLOR`，该方法将抛出代码为 `INVALID_ACCESS_ERR` 的 `DOMException` 异常。

描述

该方法将返回表示颜色的 `RGBColor` 对象。虽然没有对应的 `setRGBColorValue()` 方法，但通过设置返回的 `RGBColor` 对象的属性，可以修改这个值。

CSSPrimitiveValue.getStringValue()

2 级 DOM CSS

查询 CSS 字符串值

摘要

```
String getStringValue()  
    throws DOMException;
```

返回值

这个 `CSSPrimitiveValue` 的字符串值。

抛出

如果 `primitiveType` 属性不是 `CSS_STRING`、`CSS_URI`、`CSS_IDENT` 或 `CSS_ATTR`，该方法将抛出代码为 `INVALID_ACCESS_ERR` 的 `DOMException` 异常。

`CSSPrimitiveValue.setFloatValue()`**2 级 DOM CSS**

设置数字值

摘要

```
void setFloatValue(unsigned short unitType,  
                  float floatValue)  
    throws DOMException;
```

参数

unitType

声明该值的数字类型单位的 `CSSPrimitiveValue` 常量之一。

floatValue

新值（以 *unitType* 指定的单位计量）。

抛出

如果与该值关联的 CSS 性质是只读的，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。如果那个 CSS 性质不允许用数字值，或不允许用指定 *unitType* 的值，该方法将抛出代码为 `INVALID_ACCESS_ERR` 的 `DOMException` 异常。

描述

该方法指定了 `CSSPrimitiveValue` 的单位类型和数字值。

`CSSPrimitiveValue.setStringValue()`**2 级 DOM CSS**

设置字符串值

摘要

```
void setStringValue(unsigned short stringType,
```

```
String stringValue)
throws DOMException;
```

参数

stringType

要设置的字符串的类型。该参数必须是 `CSSPrimitiveValue` 常量 `CSS_STRING`、`CSS_URICSS_IDENT` 或 `CSS_ATTR` 中的一个。

stringValue

要设置的新值。

抛出

如果与该值关联的 CSS 性质是只读的，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。如果那个 CSS 性质不允许用字符串值，或不允许用指定 *stringType* 的值，该方法将抛出代码为 `INVALID_ACCESS_ERR` 的 `DOMException` 异常。

描述

该方法指定了 `CSSPrimitiveValue` 的字符串类型和字符串值。

CSSRule

2 级 DOM CSS

CSS 样式表中的规则

子接口

`CSSCharsetRule`、`CSSFontFaceRule`、`CSSImportRule`、`CSSMediaRule`、`CSSPageRule`、`CSSStyleRule`、`CSSUnknownRule`

常量

这些常量表示可以出现在 CSS 样式表中的各种规则类型。它们是 `type` 属性的合法值，并且声明了该对象实现了上面的哪个子接口。

```
unsigned short UNKNOWN_RULE = 0;    // CSSUnknownRule
unsigned short STYLE_RULE = 1;      // CSSStyleRule
unsigned short CHARSET_RULE = 2;    // CSSCharsetRule
unsigned short IMPORT_RULE = 3;     // CSSImportRule
unsigned short MEDIA_RULE = 4;      // CSSMediaRule
unsigned short FONT_FACE_RULE = 5;  // CSSFontFaceRule
unsigned short PAGE_RULE = 6;       // CSSPageRule
```

属性

String cssText

规则的文本表示。如果设置该属性，它将由于下列原因抛出具有下列代码的 DOMException 异常：

HIERARCHY_REQUEST_ERR

指定的规则在样式表的该位置出现是不合法的。

INVALID_MODIFICATION_ERR

该属性的新值，是一个类型不同于原始值的规则。

NO_MODIFICATION_ALLOWED_ERR

规则或包含它的样式表是只读的。

SYNTAX_ERR

指定的字符串不是合法的 CSS 语法。

readonly CSSRule parentRule

包含该规则的规则。如果这个规则没有父规则，则值为 null。具有父规则的 CSS 规则的示例是 @media 规则中的样式规则。

readonly CSSStyleSheet parentStyleSheet

包含该规则的 CSSStyleSheet 对象。

readonly unsigned short type

该对象表示的 CSS 规则的类型。该属性的合法值是前面列出的常量。CSSRule 接口从不会直接实现，该属性的值声明了这个对象实现的子接口。

描述

该接口定义了 CSS 样式表中所有类型的规则通用的属性。没有对象直接实现该接口，它们实现的是前面列出的一个子接口。最重要的子接口是 CSSStyleRule，它说明了定义文档样式的 CSS 规则。

参阅

CSSStyleRule

Type of: CSSRule.parentRule、CSSStyleDeclaration.parentRule、CSSStyleSheet.ownerRule

Returned by: CSSRuleList.item()

CSSRuleList

2 级 DOM CSS

CSSRule 对象的数组

属性

`readonly unsigned long length`

CSSRuleList 数组中的 CSSRule 对象数。

方法

`item()`

返回指定位置的 CSSRule 对象。JavaScript 不允许明确调用该方法，而是允许把 CSSRuleList 对象作为数组处理，用标准的方括号数组表示法添加下标。如果指定的下标太大，该方法将返回 null。

描述

该接口定义了一个 CSSRule 对象的列表，该列表是只读的、有序的（如数组）。属性 `length` 声明了列表中规则的数目，用方法 `item()` 可以获取指定位置的规则。在 JavaScript 中，CSSRuleList 对象的行为像 JavaScript 数组，可以用 `[]` 数组表示法在列表中查询元素，而不必调用 `item()` 方法。（但要注意，不能用 `[]` 给 CSSRuleList 添加新节点。）

参阅

Type of: `CSSMediaRule.cssRules`、`CSSStyleSheet.cssRules`

CSSRuleList.item()

2 级 DOM CSS

获取指定位置的 CSSRule 对象

摘要

`CSSRule item(unsigned long index);`

参数

`index`

要获取的规则的位置。

返回值

指定位置的 CSSRule 对象。如果 `index` 不是有效的位置，将返回 null。

CSSStyleDeclaration

2 级 DOM CSS

CSS 样式性质和它们值的集合

也实现

如果一个实现除了支持“CSS”特性外还支持“CSS2”特性（大部分浏览器都这样），那么所有实现该接口的对象还实现了 CSS2Properties 接口。CSS2Properties 接口为设置和查询 CSS 性质的值提供了通用的快捷属性。详见“CSS2Properties”。

属性

String cssText

样式性质和它们的值的文本表示。该属性由样式规则的完整文本去掉元素选择器和括起性质和值的大括号构成。把该属性设置为不合法的值，将抛出代码为 SYNTAX_ERR 的 DOMException 异常。为只读的样式表或规则设置该属性，将抛出代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 异常。

readonly unsigned long length

样式声明中样式性质的个数

readonly CSSRule parentRule

包含这个 CSSStyleDeclaration 对象的 CSSRule 对象。如果该样式声明不属于任何 CSS 规则（如表示 HTML 内联 HTML style 性质的 CSSStyleDeclaration 对象），则该属性的值为 null。

方法

getPropertyCSSValue()

返回表示指定的 CSS 性质的值的 CSSValue 对象。如果在这个样式声明块中没有明确设置该性质，或者指定的样式是“快捷”性质，则返回 null。

getPropertyPriority()

如果在这个声明块中明确设置了指定的 CSS 性质，并且设置了 !important 优先级限定符，则返回字符串“important”。如果没有声明该性质，或者没有优先级，则返回空串。

getPropertyValue()

以字符串形式返回指定的 CSS 性质的值。如果在这个声明块中没有设置指定的性质，则返回空串。

`item()`

返回样式声明块中指定位置的 CSS 性质的名字。在 JavaScript 中，可以将 `CSSStyleDeclaration` 作为数组处理，用 `[]` 进行索引。参阅 `length` 属性。

`removeProperty()`

从声明块中删除指定的 CSS 性质。

`setProperty()`

把指定的 CSS 性质设置为指定的字符串值并为声明块设置优先级。

描述

该性质表示一个 CSS 样式声明块 (style declaration block)，即 CSS 性质和它们的值的集合，中间用分号分隔。样式声明块是 CSS 样式表中位于大括号内的样式规则的一部分。HTML style 性质的值也可以构成样式声明块。

用 `item()` 方法和 `length` 属性可以遍历声明块中设置的所有 CSS 性质名。在 JavaScript 中，可以将 `CSSStyleDeclaration` 对象作为数组处理，用 `[]` 符号索引它，而不必明确地调用 `item()` 方法。如果在声明中设置了 CSS 属性名，还可以用该接口的其他方法查询这些性质的值。`getPropertyValue()` 方法将以字符串形式返回性质的值，`getPropertyCSSValue()` 方法将以 `CSSValue` 对象的形式返回性质的值。（注意，DOM API 引用 CSS 样式性质 (attribute) 作为属性 (property)。这里使用术语“attribute”，避免与 JavaScript 对象的属性 (property) 混淆。）

在大多数 Web 浏览器中，每个实现了 `CSSStyleDeclaration` 接口的对象都实现了 `CSS2Properties` 接口，后者为 CSS2 规约定义的所有 CSS 性质定义了一个对象属性。可以读写这些便捷的对象属性的值，而不必调用方法 `getPropertyValue()` 和 `setProperty()`。

参阅

`CSS2Properties`

Type of: `CSSFontFaceRule.style`、`CSSPageRule.style`、`CSSStyleRule.style`、`HTMLElement.style`

Returned by: `Document.getOverrideStyle()`、`AbstractView.getComputedStyle()`

CSSStyleDeclaration.getPropertyCSSValue() 2 级 DOM CSS

返回一个 CSS 性质的值作为对象

摘要

```
CSSValue getPropertyCSSValue(String propertyName);
```

参数

propertyName

希望使用的 CSS 性质的名字。

返回值

如果在样式声明中明确地设置了指定的性质, 则返回表示该性质值的CSSValue对象。如果没有设置指定的性质, 则返回null。如果*propertyName*指定了CSS快捷性质, 该方法也返回null, 因为快捷性质声明了多个值, 不能用CSSValue对象表示。

CSSStyleDeclaration.getPropertyPriority() 2 级 DOM CSS

获取 CSS 性质的优先级

摘要

```
String getPropertyPriority(String propertyName);
```

参数

propertyName

CSS 性质的名字。

返回值

如果在声明块中明确地设置了CSS性质, 并且具有!important优先级限定符, 则返回字符串“important”, 否则返回空串。

CSSStyleDeclaration.getPropertyValue() 2 级 DOM CSS

获取 CSS 性质的字符串值

摘要

```
String getPropertyValue(String propertyName);
```

参数

propertyName

想获取的 CSS 性质的名字。

返回值

指定的 CSS 性质的字符串值。如果在声明块中没有明确设置该性质，则返回空串。

描述

该方法将返回指定的 CSS 性质的字符串值。与 `getPropertyCSSValue()` 方法不同，该方法可以像处理常规性质一样处理快捷性质。参阅 `CSS2Properties` 接口的各种便捷属性。

CSSStyleDeclaration.item()

2 级 DOM CSS

获取指定位置的 CSS 性质的名字

摘要

```
String item(unsigned long index);
```

参数

index

想获取的 CSS 性质名的位置。

返回值

index 处的 CSS 性质的名字。如果 *index* 是负数或大于等于 `length` 属性，则返回空串。

描述

`CSSStyleDeclaration` 接口表示 CSS 样式性质和它们的值的集合。使用该方法，可以根据位置查询 CSS 性质的名字。结合 `length` 属性还可以遍历样式声明中设置的 CSS 性质集合。注意，该方法返回的 CSS 性质的顺序不必与它们在文档或样式表源代码中出现的顺序一致。

作为 `item()` 的替代方法，JavaScript 允许将 `CSSStyleDeclaration` 对象作为 CSS 性质名的数组，并使用标准的 `[]` 数组语法获取指定位置的性质名。

CSSStyleDeclaration.removeProperty()2 级 DOM CSS

删除一个 CSS 性质声明

摘要

```
String removeProperty(String propertyName)
    throws DOMException;
```

参数

propertyName

要删除的 CSS 性质名。

返回值

指定的 CSS 性质的字符串值。如果样式声明中没有明确地设置指定的性质，则返回空串。

抛出

如果样式声明是只读的，该方法将抛出代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 异常。

描述

该方法将从样式声明块中删除指定的性质，并返回该性质的值。

CSSStyleDeclaration.setProperty()2 级 DOM CSS

设置一个 CSS 样式性质

摘要

```
void setProperty(String propertyName,
                 String value,
                 String priority)
    throws DOMException;
```

参数

propertyName

要设置的 CSS 性质的名字。

`value`

性质的新字符串值。

`priority`

性质的新优先级。如果该属性的声明是 `!important`, 则该参数应该是 `"important"`。否则它应该是空串。

抛出

如果指定的参数 `value` 格式错误, 该方法将抛出代码为 `SYNTAX_ERR` 的 `DOMException` 异常。如果样式声明或要设置的性质是只读的, 该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

描述

该方法将把指定的 CSS 性质名和它的优先级添加到样式声明中。如果样式声明中的指定性质已经有一个值, 该方法将只设置现有的性质的值和优先级。

实际上, 用 `setProperty()` 给样式声明添加新的 CSS 性质, 会把该性质插入到任意的位置, 完全打乱现有性质的顺序。因此, 在用 `item()` 方法遍历性质名集合时, 不应该使用 `setProperty()` 方法。

CSSStyleRule

2 级 DOM CSS

CSS 样式表中的一条样式规则

`CSSRule` → `CSSStyleRule`

属性

`String selectorText`

选择器文本, 指定了应用该样式规则的文档元素。如果该规则是只读的, 设置这个属性会抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。如果新值不符合 CSS 语法规则, 则抛出代码为 `SYNTAX_ERR` 的异常。

`readonly CSSStyleDeclaration style`

应该应用到 `selectorText` 指定的元素的样式值。

描述

这个接口表示 CSS 样式表中的样式规则。样式规则是样式表中最常用、最重要的规则, 它们设置了要应用到指定文档元素集合的样式信息。`SelectorText` 是该规则的

元素选择器的字符串表示。style是一个CSSStyleDeclaration对象，表示要应用到选定的元素的样式名和值的集合。

参阅

CSSStyleDeclaration

CSSStyleSheet

2 级 DOM CSS

CSS 样式表

StyleSheet → CSSStyleSheet

属性

readonly CSSRuleList cssRules

构成样式表的CSSRule对象的数组（从技术上说，是CSSRuleList对象）。除了真正的样式规则外，它还包括所有的附属规则。

readonly CSSRule ownerRule

如果该样式表由另一个样式表中的@import规则导入，则该属性存放表示那个@import规则的CSSImportRule对象。否则，它是null。当该属性非空时，继承的ownerNode属性为null。

方法

deleteRule()

删除指定位置的规则。

insertRule()

在指定位置插入一个新规则。

描述

该接口表示一个CSS样式表。cssRules属性列出了该样式表中包含的规则，用insertRule()方法和deleteRule()方法可以在列表中添加和删除规则。

参阅

StyleSheet

Type of: CSSImportRule.styleSheet、CSSRule.parentStyleSheet

Returned by: DOMImplementation.createCSSStyleSheet()

CSSStyleSheet.deleteRule()

2 级 DOM CSS

从样式表中删除一个规则

摘要

```
void deleteRule(unsigned long index)
    throws DOMException;
```

参数

index

要删除的规则在 `cssRules` 数组中的下标。

抛出

如果 *index* 是负数或大于等于 `cssRules.length`，该方法将抛出代码为 `INDEX_SIZE_ERR` 的 `DOMException` 异常。如果样式表是只读的，它将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

描述

该方法将删除 `cssRules` 数组指定 *index* 处的规则。

CSSStyleSheet.insertRule()

2 级 DOM CSS

在样式表中插入一条规则

摘要

```
unsigned long insertRule(String rule,
                          unsigned long index)
    throws DOMException;
```

参数

rule

要添加到样式表的规则的完整的、可解析的文本表示。对于样式规则，该参数包括元素选择器和样式信息。

index

要把规则插入或附加到 `cssRules` 数组中的位置。

返回值

参数 *index* 的值。

抛出

该方法在下列情况下将抛出具有下列代码的 DOMException 异常：

HIERARCHY_REQUEST_ERR

CSS 语法不允许指定的规则出现在指定的位置。

INDEX_SIZE_ERR

index 是负数或大于 `cssRules.length` 的值。

NO_MODIFICATION_ALLOWED_ERR

该样式表是只读的。

SYNTAX_ERR

指定的 *rule* 文本具有语法错误。

描述

该方法将在样式表的 `cssRules` 数组的指定 *index* 处插入（或附加）新的 `cssRule`。

CSSUnknownRule

2 级 DOM CSS

CSS 样式表中未被承认的规则

`CSSRule` → `CSSUnknownRule`

描述

该接口表示浏览器不承认或不能解析（通常因为它是由浏览器不支持的 CSS 标准版本定义的）的 CSS 样式表中的一条规则。注意，该接口没有定义任何属于自己的属性或方法。通过继承属性 `cssText` 可以访问未被承认的规则的文本文本。

CSSValue

2 级 DOM CSS

CSS 样式性质的值

子接口

`CSSPrimitiveValue`、`CSSValueList`

常量

下面的常量是 `cssValueType` 属性的有效值：

```
unsigned short CSS_INHERIT = 0
```

该常量表示特殊值“inherit”，意味着真正的CSS样式性质的值是继承的。在这种情况下，`cssText`属性是“inherit”。

```
unsigned short CSS_PRIMITIVE_VALUE = 1
```

这是原始值。该CSSValue对象还实现了更专用的CSSPrimitiveValue子接口。

```
unsigned short CSS_VALUE_LIST = 2
```

这是由值列表构成的复合值。该CSSValue对象还实现了更专用的CSSValueList子接口，它的行为像CSSValue对象的数组。

```
unsigned short CSS_CUSTOM = 3
```

定义该常量是为了扩展CSS对象模型。它说明该CSSValue对象表示的值的类型不是CSS或DOM标准定义的。如果采用的实现支持这种扩展，那么该CSSValue对象还实现了其他一些可以使用的子接口（如Scalable Vector Graphics标准定义的SVGColor接口）。

属性

```
String cssText
```

值的文本表示。设置该属性可能抛出DOMException异常。如果该异常的代码为SYNTAX_ERR，说明新值的CSS语法不合法。如果异常代码为INVALID_MODIFICATION_ERR，说明要设置的值的类型不同于原始值。如果代码为NO_MODIFICATION_ALLOWED_ERR，说明该值是只读的。

```
readonly unsigned short cssValueType
```

该对象表示的值的类型。前面的常量列表中列出了该属性的四个合法值。

描述

这个接口表示CSS性质的值。`cssText`属性给出了该值的文本形式。如果`cssValueType`属性的值为CSSValue.CSS_PRIMITIVE_VALUE，那么这个CSSValue对象还实现了更专用的CSSPrimitiveValue接口。如果`cssValueType`属性的值为CSSValue.CSS_VALUE_LIST，那么这个CSSValue对象表示值的列表，还实现了更专用的CSSValueList接口。

参阅

CSSPrimitiveValue、CSSValueList

Returned by: `CSSStyleDeclaration.getPropertyCSSValue()`, `CSSValueList.item()`

CSSValueList

2 级 DOM CSS

存放 CSSValue 对象的数组的 CSSValue 对象

CSSValue → CSSValueList

属性

readonly unsigned long length

该数组中的 CSSValue 对象的个数。

方法

item()

返回位于数组指定位置的 CSSValue 对象。如果指定的位置是负数，或它大于等于 length，则返回 null。

描述

该接口表示 CSSValue 对象的数组，并且它自身也是一种 CSSValue 接口。用方法 item() 可以获取指定位置的 CSSValue 对象。但在 JavaScript 中，用标准的 [] 语法索引数组更容易一些。

CSSValue 对象在 CSSValueList 数组中的顺序是它们在 CSS 样式声明中出现的顺序。有些值为 CSSValueList 的 CSS 性质也可能具有值 none。这个特殊的值将转换成 length 值为 0 的 CSSValueList 对象。

CSSValueList.item()

2 级 DOM CSS

获取指定位置的 CSSValue 对象

摘要

CSSValue item(unsigned long index);

参数

index

想获取的 CSSValue 对象的位置。

返回值

CSSValueList 对象中指定位置的 CSSValue 对象。如果 index 是负数，或大于等于 length，则返回 null。

Document

1 级核心 DOM

HTML 文档或 XML 文档

Node → Document

子接口

HTMLDocument

也实现

DocumentCSS

如果实现支持 CSS 模块，那么实现了 Document 接口的对象也实现了 DocumentCSS 接口和它的 `getOverrideStyle()` 方法。

DocumentEvent

如果实现支持 Events 模块，那么实现了 Document 接口的对象也实现了 DocumentEvent 接口和它的 `createEvent()` 方法。

DocumentRange

如果实现支持 Range 模块，那么实现了 Document 接口的对象也实现了 DocumentRange 接口和它的 `createRange()` 方法。

DocumentStyle

如果实现支持 StyleSheet 模块，那么实现了 Document 接口的对象也实现了 DocumentStyle 接口和它的 `styleSheets` 属性。

DocumentTraversal

如果实现支持 Traversal 模块，那么实现了 Document 接口的对象也实现了 DocumentTraversal 接口和它的 `createNodeIterator()` 方法和 `createTreeWalker()` 方法。

DocumentView

如果实现支持 Views 模块，那么实现了 Document 接口的对象还实现了 DocumentView 接口和它的 `defaultView` 属性。

因为除了 Document 接口外，通常会实现这些接口，所以这里列出了它们的属性和方法，就像这些属性和方法直接属于 Document 对象一样。

属性

`readonly AbstractView defaultView` [2 级 DOM Views]

文档的默认视图。在 Web 浏览器环境中，该属性声明了显示文档的 Window 对象（它实现了 AbstractView 接口）。

注意,从技术上说,该属性是DocumentView接口的一部分。它只能在支持Views模块的实现中由Document对象定义。

readonly DocumentType doctype

对于具有<!DOCTYPE>声明的XML文档来说,该属性声明了表示文档的DTD的DocumentType节点。对于没有<!DOCTYPE>声明的HTML文档和XML文档来说,该属性为null。注意,该属性是只读的,它引用的节点也是只读的。

readonly Element documentElement

对文档根元素的引用。对于HTML文档来说,该属性总是表示<html>标记的Element对象。通过从Node节点继承的childNodes[]数组,也可以访问根元素。

readonly DOMImplementation implementation

表示创建该文档的实现的DOMImplementation对象。

readonly StyleSheetList styleSheets [2级DOM StyleSheets]

表示嵌入或链接入文档的所有样式表的对象的集合。在HTML文档中,该集合通常包括<link>和<style>标记定义的样式表。

注意,该属性从技术上说属于DocumentStyle接口,只有在支持StyleSheets模块的实现中,Document对象才定义该属性。

方法

createAttribute()

用指定的名字创建新的Attr节点。

createAttributeNS() [2级DOM]

用指定的名字和名字空间创建新的Attr节点。

createCDATASection()

创建包含指定文本的新CDATASection节点。

createComment()

创建包含指定字符串的新Comment节点。

createDocumentFragment()

创建新的、空的DocumentFragment节点。

`createElement()`

用指定的标记名创建新的 `Element` 节点。

`createElementNS()` [2 级 DOM]

用指定的标记名和名字空间创建新的 `Element` 节点。

`createEntityReference()`

创建新的 `EntityReference` 节点，引用具有指定名字的实体。如果该文档的 `DocumentType` 对象定义了具有那个名字的 `Entity` 对象，那么新创建的 `EntityReference` 节点将具有与 `Entity` 相同的只读子节点。

`createEvent()` [2 级 DOM Events]

用指定的类型创建一个新的合成 `Event` 对象。从技术上说，该方法是由 `DocumentEvent` 接口定义的。只有在支持 `Events` 模块的实现中，`Document` 对象才实现了该方法。

`createNodeIterator()` [2 级 DOM Traversal]

创建一个 `NodeIterator` 对象。从技术上说，该方法是 `Document Traversal` 接口的一部分，只有在支持 `Traversal` 模块的实现中，`Document` 对象才实现了该方法。

`createProcessingInstruction()`

用指定的标记和数据串创建新的 `ProcessingInstruction` 节点。

`createRange()` [2 级 DOM Range]

创建一个新的 `Range` 对象。从技术上说，该方法是由 `DocumentRange` 接口定义的，只有在支持 `Range` 模块的实现中，`Document` 对象才定义了该方法。

`createTextNode()`

创建新的 `Text` 节点，表示指定的文本。

`createTreeWalker()` [2 级 DOM Traversal]

创建一个新的 `TreeWalker` 对象。从技术上来说，该方法是 `Document Traversal` 接口的一部分，只有在支持 `Traversal` 模块的实现中，`Document` 对象才定义了该方法。

`getElementById()` [2 级 DOM]

返回该文档中具有指定 `id` 性质的子孙 `Element` 节点。如果文档中没有这样的 `Element` 节点，则返回 `null`。

`getElementsByTagName()`

返回文档中所有具有指定标记名的 `Element` 节点的数组（从技术上说，是 `NodeList`）。`Element` 节点出现在返回数组中的顺序就是它们在源文档中出现的顺序。

`getElementsByTagNameNS()` [2 级 DOM]

返回所有具有指定标记名和名字空间的 `Element` 节点。

`getOverrideStyle()` [2 级 DOM CSS]

获取指定 `Element`（或选的指定伪元素）的 CSS 覆盖样式信息。从技术上来讲，该方法属于 `DocumentCSS` 接口。只有在支持 CSS 模块的实现中，`Document` 对象才实现了该方法。

`importNode()` [2 级 DOM]

对于其他适合插入该文档的文档，生成它的节点副本。

描述

`Document` 接口是文档树的根节点。一个 `Document` 节点可以具有多个子节点，但它们中只能有一个 `Element` 节点，即它是文档的根元素。通过 `documentElement` 属性最容易访问根元素。使用 `doctype` 属性和 `implementation` 属性可以访问文档的 `DocumentType` 对象（如果存在的话）和 `DOMImplementation` 对象。

`Document` 接口定义的大多数方法都是“生产方法”，用于创建可以插入文档的各种类型的节点。但 `getElementById()` 方法和 `getElementsByTagName()` 方法是例外，它们对查找文档树中指定的 `Element` 或相关 `Element` 节点的集合来说非常有用。

比较一下这个 `Document` 对象和本书的客户端参考手册部分说明的 `Document` 对象。DOM 标准在 `HTMLDocument` 接口中正式定义了客户端 `Document` 对象的 0 级属性和方法。参阅这个参考手册中的“`HTMLDocument`”参考页，它是 DOM 中等价于传统 JavaScript `Document` 对象的接口。

`Document` 接口是由 2 级 DOM 规约的 Core 模块定义的。其他许多模块定义了附加接口，由实现 `Document` 接口的对象实现。例如，如果一个实现支持 CSS 模块，那么实现该接口的对象还实现了 `DocumentCSS` 接口。在 JavaScript 中，可以像使用 `Document` 定义的方法和属性那样，使用附加接口定义的属性和方法。因此，这些属性和方法在这里列出。关于 `Document` 的附加接口的完整列表，参阅前面的“也实现”部分。

参阅

HTMLDocument

Type of: AbstractView.document、HTMLFrameElement.contentDocument、HTMLIFrameElement.contentDocument、HTMLObjectElement.contentDocument、Node.ownerDocument

Returned by: DOMImplementation.createDocument()

Document.createAttribute()

1 级核心 DOM

创建新的 Attr 节点

摘要

```
Attr createAttribute(String name)
    throws DOMException;
```

参数

name

新创建的性质的名字。

返回值

新创建的 Attr 节点，nodeName 属性设置为 *name*。

抛出

如果 *name* 含有不合法的字符，该方法将抛出代码为 INVALID_CHARACTER_ERR 的 DOMException 异常。

参阅

Attr、Element.setAttribute()、Element.setAttributeNode()

Document.createAttributeNS()

2 级核心 DOM

创建具有指定的名字或名字空间的 Attr 节点

摘要

```
Attr createAttributeNS(String namespaceURI,
    String qualifiedName)
```


throws DOMException;

参数

namespaceURI

Attr 的名字空间的唯一标识符。如果没有名字空间，则为 null。

qualifiedName

性质的限定名，应该包括名字空间前缀、冒号和本地名。

返回值

新创建的 Attr 节点，具有指定的名字和名字空间。

抛出

在下列环境中，该方法将抛出具有下列代码的 DOMException 异常：

INVALID_CHARACTER_ERR

qualifiedName 中含有不合法的字符。

NAMESPACE_ERR

qualifiedName 格式错误，或者 *qualifiedName* 与 *namespaceURI* 不匹配。

NOT_SUPPORTED_ERR

该实现不支持 XML 文档，因此没有实现该方法。

描述

`createAttributeNS()` 方法与 `createAttribute()` 方法相似，只是它创建的 Attr 节点除了具有指定的名字外，还具有指定的名字空间。只有使用名字空间的 XML 文档才会使用该方法。

Document.createCDATASection()

1 级核心 DOM

创建新的 CDATASection 节点

摘要

CDATASection createCDATASection(String data)

throws DOMException;

参数*data*

要创建的 CDATASection 的文本。

返回值

新创建的 CDATASection 节点，内容为指定的 *data*。

抛出

如果该文档是 HTML 文档，该方法将抛出代码为 NOT_SUPPORTED_ERR 的 DOMException 异常，因为 HTML 文档不支持 CDATASection 节点。

Document.createComment()

1 级核心 DOM

创建新的 Comment 节点

摘要

```
Comment createComment(String data);
```

参数*data*

要创建的 Comment 节点的文本。

返回值

新创建的 Comment 节点，文本为指定的 *data*。

Document.createDocumentFragment()

1 级核心 DOM

创建新的、空的 DocumentFragment 节点

摘要

```
DocumentFragment createDocumentFragment();
```

返回值

新创建的 DocumentFragment 节点，没有子节点

Document.createElement()1 级核心 DOM

创建新的 Element 节点

摘要

```
Element createElement(String tagName)
```

```
throws DOMException;
```

参数*tagName*

要创建的 Element 的标记名。因为 HTML 不区分大小写，所以 HTML 的标记名可以采用任意的大小写形式。XML 标记名则是区分大小写的。

返回值

新创建的 Element 节点，具有指定的标记名。

抛出

如果 *tagName* 中含有不合法的字符，该方法将抛出代码为 `INVALID_CHARACTER_ERR` 的 `DOMException` 异常。

Document.createElementNS()2 级核心 DOM

创建使用指定的名字空间的新 Element 节点

摘要

```
Element createElementNS(String namespaceURI,  
                        String qualifiedName)
```

```
throws DOMException;
```

参数*namespaceURI*

新的 Element 的名字空间的唯一标识符。如果没有名字空间，则为 `null`。

qualifiedName

新的 Element 的限定名，应该包括名字空间前缀、冒号和本地名。

返回值

新创建的 Element 节点，具有指定的标记名字和名字空间。

抛出

在下列环境中，该方法将抛出具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

`qualifiedName` 中含有不合法的字符。

`NAMESPACE_ERR`

`qualifiedName` 格式错误，或者 `qualifiedName` 与 `namespaceURI` 不匹配。

`NOT_SUPPORTED_ERR`

该实现不支持 XML 文档，因此没有实现该方法。

描述

`createElementNS()` 方法与 `createElement()` 方法相似，只是它创建的 `Element` 节点除了具有指定的名字外，还具有指定的名字空间。只有使用名字空间的 XML 文档才会使用该方法。

Document.createEntityReference()

1 级核心 DOM

创建新的 `EntityReference` 节点

摘要

```
EntityReference createEntityReference(String name)  
    throws DOMException;
```

参数

name

被引用的实体的名字。

返回值

新创建的 `EntityReference` 节点，引用具有指定名字的实体。

抛出

在下列环境中，该方法将抛出具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

指定的实体名含有不合法的字符。

NOT_SUPPORTED_ERR

这是一个 HTML 文档，不支持实体引用。

描述

该方法将创建并返回一个 EntityReference 节点，该节点引用具有指定名字的实体。注意，如果该文档是 HTML 文档，该方法就会抛出异常，因为 HTML 不支持实体引用。如果该文档具有 DocumentType 节点，而且那个 DocumentType 节点定义了具有指定的名字的 Entity 对象，那么返回的 EntityReference 节点具有与引用的 Entity 节点相同的子节点。

Document.createEvent()

2 级 DOM Events

创建新的 Event 对象

摘要

```
Event createEvent(String eventType)
    throws DOMException;
```

参数

eventType

想获取的 Event 对象的事件模块名。关于有效的事件类型的列表，请参阅“描述”部分。

返回值

新创建的 Event 对象，具有指定的类型。

抛出

如果实现不支持需要的事件类型，该方法将抛出代码为 NOT_SUPPORTED_ERR 的 DOMException 异常。

描述

该方法将创建一种新的事件类型，该类型由参数 *eventType* 指定。注意，该参数的值不是要创建的事件接口的（单数）名字，而是定义那个接口的 DOM 模块的（复数）名字。下表列出了 *eventType* 的合法值和每个值创建的事件接口。

参数	事件接口	初始化方法
HTMLEvents	Event	<code>initEvent()</code>
MouseEvents	MouseEvent	<code>initMouseEvent()</code>
UIEvents	UIEvent	<code>initUIEvent()</code>
MutationEvents	MutationEvent	<code>initMutationEvent()</code>

用该方法创建了 Event 对象后，必须用上表中所示的初始化方法初始化对象。关于初始化方法的详细信息，请参阅相关的 Event 接口参考页。

该方法实际上不是由 Document 接口定义，而是由 DocumentEvent 接口定义的。如果一个实现支持 Events 模块，那么 Document 对象就会实现 DocumentEvent 接口并支持该方法。

参阅

Event、MouseEvent、MutationEvent、UIEvent

Document.createNodeIterator()

2 级 DOM Traversal

为该文档创建 NodeIterator 节点

摘要

```
NodeIterator createNodeIterator(Node root,  
                                unsigned long whatToShow,  
                                NodeFilter filter,  
                                boolean entityReferenceExpansion)  
    throws DOMException;
```

参数

root

NodeIterator 节点要遍历的子树的根节点。

whatToShow

一个或多个 NodeFilter 标志的位掩码，指定了 NodeIterator 应该返回的节点的类型。

filter

NodeIterator 节点的一个可选的节点过滤函数。如果没有节点过滤器，则为 null。

entityReferenceExpansion

如果 *NodeIterator* 应该在 XML 文档中扩展实体引用, 则值为 `true`, 否则为 `false`。

返回值

新创建的 *NodeIterator* 对象。

抛出

如果参数 *root* 为 `null`, 该方法将抛出代码为 `NOT_SUPPORTED_ERR` 的 *DOMException* 异常。

描述

该方法将创建并返回新的 *NodeIterator* 对象, 用指定的过滤器遍历根节点为 *root* 的子树。

该方法实际上不是 *Document* 接口定义的, 而是 *DocumentTraversal* 接口定义的。如果一个实现支持 *Traversal* 模块, 那么 *Document* 对象就会实现 *DocumentTraversal* 接口, 并定义该方法。

参阅

Document.createTreeWalker()、*NodeFilter*、*NodeIterator*

Document.createProcessingInstruction()

1 级核心 DOM

创建 *ProcessingInstruction* 节点

摘要

```
ProcessingInstruction createProcessingInstruction(String target,  
                                                String data)  
  
    throws DOMException;
```

参数

target

处理指令的目标。

data

处理指令的内容文本。

返回值

新创建的 `ProcessingInstruction` 节点。

抛出

在下列环境中，该方法将抛出具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

`target` 中含有不合法的字符。

`NOT_SUPPORTED_ERR`

这是一个 HTML 文档，不支持处理指令。

Document.createRange()

2 级 DOM Range

创建 Range 对象

摘要

```
Range createRange();
```

返回值

新创建的 `Range` 对象，两个边界点都被设置为文档的开头。

描述

该方法将创建一个 `Range` 对象，可以用来表示文档的一个区域或与该文档相关的 `DocumentFragment` 对象。

注意，该方法实际上不是由 `Document` 接口定义的，而是由 `DocumentRange` 接口定义的。如果一个实现支持 `Range` 模块，那么 `Document` 对象就会实现 `DocumentRange` 接口，并定义该方法。

Document.createTextNode()

1 级核心 DOM

创建新的 Text 节点

摘要

```
Text createTextNode(String data);
```


参数*data*

Text 节点的内容。

返回值

新创建的 Text 节点，表示指定的 *data* 字符串。

Document.createTreeWalker()

2 级 DOM Traversal

为该文档创建 TreeWalker 节点

摘要

```
TreeWalker createTreeWalker(Node root,  
                             unsigned long whatToShow,  
                             NodeFilter filter,  
                             boolean entityReferenceExpansion)  
throws DOMException;
```

参数*root*

TreeWalker 节点要经过的子树的根节点。

whatToShow

一个或多个 NodeFilter 标志的位掩码，指定了 TreeWalker 应该返回的节点的类型。

filter

TreeWalker 节点的一个可选的过滤函数。如果没有节点过滤器，则为 null。

entityReferenceExpansion

如果 TreeWalker 应该在 XML 文档中扩展实体引用，则值为 true。否则为 false。

返回值

新创建的 TreeWalker 对象。

抛出

如果参数 *root* 为 null，该方法将抛出代码为 NOT_SUPPORTED_ERR 的 DOMException 异常。

描述

该方法将创建并返回新的 `TreeWalker` 对象，用指定的过滤器遍历根节点为 `root` 的子树。

该方法实际上不是 `Document` 接口定义的，而是 `DocumentTraversal` 接口定义的。如果一个实现支持 `Traversal` 模块，那么 `Document` 对象就会实现 `DocumentTraversal` 接口，并定义该方法。

参阅

`Document.createNodeIterator()`、`NodeFilter`、`TreeWalker`

`Document.getElementById()` 2 级核心 DOM; 在 1 级 DOM 中，由 `HTMLDocument` 定义
查找具有指定的唯一 ID 的元素

摘要

```
Element getElementById(String elementId);
```

参数

elementId

想获取的元素的 `id` 性质的值。

返回值

表示具有指定的 `id` 性质的文档元素的 `Element` 节点。如果没有找到这样的元素，则返回 `null`。

描述

该方法将检索 `id` 性质的值为 *elementId* 的 `Element` 节点，并将它返回。如果没有找到这样的元素 `Element`，它将返回 `null`。`id` 性质的值在文档中是惟一的，如果该方法找到多个具有指定 *elementId* 的 `Element` 节点，它将随机返回一个这样的 `Element` 节点，或者返回 `null`。

在 `HTML` 文档中，该方法总是检索具有指定 `id` 的性质。但在 `XML` 文档中，它则检索 `type` 为 `id` 的所有性质（无论性质名是什么）。如果 `XML` 性质的类型是未知的（如 `XML` 解析器不能定位文档的 `DTD`），该方法总是返回 `null`。

这是一个重要的常用方法，因为它为获取表示指定的文档元素的 `Element` 对象提供了简便的方法。注意，它提供的功能与 Internet Explorer 4 及其后版本定义的 `document.all[]` 数组相似。最后要注意，该方法的名字以 `Id` 结尾，不是 `ID`，不要拼错了。

参阅

`Document.getElementsByTagName()`、`Element.getElementsByTagName()`、`HTMLDocument.getElementsByTagName()`

`Document.getElementsByTagName()`

1 级核心 DOM

返回所有具有指定名字的 `Element` 节点

摘要

```
Node[] getElementsByTagName(String tagname);
```

参数

tagname

要返回的 `Element` 节点的标记名或通配符 `*`（返回文档中的所有 `Element` 节点，无论标记的名字是什么）。对于 HTML 文档，标记名不区分大小写。

返回值

文档树中具有指定标记名的 `Element` 节点的只读数组（从技术上说，是 `NodeList` 对象）。返回的 `Element` 节点的顺序就是它们在源文档中出现的顺序。

描述

该方法将返回一个 `NodeList` 对象（可以作为只读数组处理），该对象存放文档中具有指定标记名的 `Element` 节点，它们存放的顺序就是在源文档中出现的顺序。`NodeList` 对象是“活的”，即如果在文档中添加或删除了具有指定标记名的元素，它的内容会自动进行必要的更新。

HTML 文档不区分大小写，所以可以用任意的大小写形式指定 *tagname*，它将与文档中所有同名的标记匹配，无论这些标记在源文档中的采用的大小写形式是什么。但 XML 文档区分大小写，*tagname* 只和源文档中名字与大小写形式完全相同的标记匹配。

注意，`Element` 接口定义了一个同名的方法，该方法只检索文档的子树。另外，

HTMLDocument 接口定义了 `getElementsByName()` 方法，它基于 `name` 性质的值（而不是标记名）检索元素。

例子

可以用下列代码检索并遍历文档中的所有 `<h1>` 标记：

```
var headings = document.getElementsByTagName('h1');
for(var i = 0; i < headings.length; i++) { // 循环遍历返回的标记
    var h = headings[i];
    // 用h变量中的<h1>元素进行某些操作
}
```

参阅

`Document.getElementById()`、`Element.getElementsByTagName()`、`HTMLDocument.getElementsByName()`

Document.getElementsByTagNameNS()

2 级核心 DOM

返回所有具有指定名字和名字空间的 Element 节点

摘要

```
Node[] getElementsByTagNameNS(String namespaceURI,
                                String localName);
```

参数

namespaceURI

要获取的元素的名字空间的唯一标识符，或者“*”，匹配所有名字空间。

localName

要获取的元素的本地名（或者“*”）来匹配所有本地名字。

返回值

文档树中具有指定的名字空间和本地名的 Element 节点的只读数组（从技术上说，是 `NodeList` 对象）。

描述

该方法与 `getElementsByTagName()` 相似，只是它根据名字空间和名字来检索元素。只有使用名字空间的 XML 文档才会使用它。

Document.getOverrideStyle()2 级 DOM CSS

获取指定元素的覆盖样式

摘要

```
CSSStyleDeclaration getOverrideStyle(Element elt,  
                                     String pseudoElt);
```

参数**elt**

想获取覆盖样式的元素。

pseudoElt

elt 的伪元素。如果不存在，则返回 null。

返回值

表示指定的元素和伪元素的覆盖样式信息的 CSSStyleDeclaration 对象。返回的对象通常还实现了常用的 CSS2Properties 接口。

描述

该方法将为指定的元素和可选的伪元素返回一个 CSSStyleDeclaration 对象（通常还实现了 CSS2Properties 对象）。可以利用返回的对象改变指定元素的显示样式，无须扰乱元素的内联样式，也不必修改文档的样式表。从概念上说，返回的值表示覆盖样式表中的样式声明，这个样式表的优先级高于其他样式表和内联样式（除了用户样式表中的 !important 声明）。

注意，该方法不是由 Document 接口定义的，而是由 DocumentCSS 接口定义的。如果一个实现支持 CSS 模块，那么 Document 对象就会实现 DocumentCSS 对象，并定义该方法。

参阅

CSSStyleDeclaration、CSS2Properties、AbstractView.getComputedStyle()、
HTMLElement.style

Document.importNode()

2 级核心 DOM

把一个节点从另一个文档复制到该文档以便应用

摘要

```
Node importNode(Node importedNode,  
                boolean deep)  
    throws DOMException;
```

参数

importedNode

要导入的节点。

deep

如果为 true，还要递归复制 *importedNode* 节点的所有子孙节点。

返回值

importedNode (可能还有它的所有子孙节点) 的副本，它的 *ownerDocument* 属性设置到该文档。

抛出

如果 *importedNode* 是 Document 节点或 DocumentType 节点，该方法将抛出代码为 NOT_SUPPORTED_ERR 的 DOMException 异常，因为不能导入这些类型的节点。

描述

该方法的参数是另一个文档中定义的节点，返回值是适合插入该文档的节点的副本。如果 *deep* 值为 true，那么还要复制该节点的所有子孙节点。无论如何，原始节点和它的子孙节点都不会被修改。返回的副本的 *ownerDocument* 属性被设置为当前文档，但 *parentNode* 属性为 null，因为它还没有插入文档。在原始节点或树中注册的 *EventListener* 函数不会被复制。

当导入 Element 节点时，只有在源文档中明确设置的属性才会被导入。当导入 Attr 节点时，将自动把它的 *specified* 属性设置为 true。

参阅

`Node.cloneNode()`

DocumentCSS

参阅 `Document`

DocumentEvent

参阅 `Document`

DocumentFragment

1 级核心 DOM

邻接节点和它们的子树

`Node` → `DocumentFragment`

描述

`DocumentFragment` 接口表示文档的一部分（或一段）。更明确地说，它表示一个或多个邻接的 `Document` 节点和它们的所有子孙节点。`DocumentFragment` 节点不属于文档树，继承的 `parentNode` 属性总是为 `null`。不过它有一种特殊行为，该行为使得它非常有用，即当请求把一个 `DocumentFragment` 节点插入文档树时，插入的不是 `DocumentFragment` 自身，而是它的所有子孙节点。这使得 `DocumentFragment` 成了有用的占位符，暂时存放那些希望一次插入文档的节点。它还有利于实现文档的剪切、复制和粘贴操作，尤其是与 `Range` 接口一起使用时更是如此。

可以用 `Document.createDocumentFragment()` 方法创建新的空 `DocumentFragment` 节点，也可以用 `Range.extractContents()` 或 `Range.cloneContents()` 方法获取包含现有文档的片段的 `DocumentFragment` 节点。

参阅

`Range`

Returned by: `Document.createDocumentFragment()`、`Range.cloneContents()`、`Range.extractContents()`

DocumentRange

参阅 `Document`

DocumentStyle

参阅 Document

DocumentTraversal

参阅 Document

DocumentType

1 级 DOM XML

XML 文档的 DTD

Node → DocumentType

属性

readonly NamedNodeMap entities

NamedNodeMap 是 DTD 中声明的 Entity 对象的列表，允许通过名字查询 Entity 对象。注意，它不包括 XML 参数实体。NamedNodeMap 是不可变的，它的内容不能改变。

readonly String internalSubset [2 级 DOM]

DTD 的内部子集（即出现在文档自身的 DTD，而不是出现在外部文件中的 DTD）。内部子集的分界符 | 不属于返回值。如果没有这样的内部子集，该属性为 null。

readonly String name

文档的类型名。它是 XML 文档开头的 <!DOCTYPE> 后的标识符，它与文档根元素的标记名相同。

readonly NamedNodeMap notations

包含 Notation 对象的 NamedNodeMap 节点。其中的 Notation 对象表示 DTD 中声明的所有符号。它还允许用符号名查看 Notation 对象。NamedNodeMap 是不可变的，它的内容不能改变。

readonly String publicId [2 级 DOM]

DTD 的外部子集的公有标识符。如果没有设置这种标识符，值为 null。

readonly String systemId [2 级 DOM]

DTD 的外部子集的系统标识符。如果没有设置这种标识符，值为 null。

描述

这种不常用的接口表示 XML 文档的 DTD。专门处理 HTML 文档的程序设计者不必使用该接口。

因为 DTD 不属于文档的内容，所以 `DocumentType` 节点不会出现在文档树中。如果 XML 文档有 DTD，通过 `Document` 节点的 `doctype` 属性可以访问那个 DTD 的 `DocumentType` 节点。

`DocumentType` 是不可变的，它的内容不能改变。

参阅

`Document`、`Entity`、`Notation`

Type of: `Document.doctype`

Passed to: `DOMImplementation.createDocument()`

Returned by: `DOMImplementation.createDocumentType()`

DocumentView

参阅 `Document`

DOMException

1 级核心 DOM

通知核心 DOM 对象的异常或错误

常量

下面的常量定义了 `DOMException` 对象的 `code` 属性的合法值。注意，这些常量是 `DOMException` 的静态属性，不是个别异常对象的属性。

```
unsigned short INDEX_SIZE_ERR = 1
```

说明数组或字符串下标的溢出错误。

```
unsigned short DOMSTRING_SIZE_ERR = 2
```

说明请求的文本太大，当前 JavaScript 实现的字符串容纳不了它。

```
unsigned short HIERARCHY_REQUEST_ERR = 3
```

说明发生了要把节点放在文档树层次中的不合法位置的操作。

```
unsigned short WRONG_DOCUMENT_ERR = 4
```

说明发生了从创建节点的文档以外的文档使用该节点的操作。

`unsigned short INVALID_CHARACTER_ERR = 5`

说明（如在元素名中）使用了不合法的字符。

`unsigned short NO_DATA_ALLOWED_ERR = 6`

当前不采用。

`unsigned short NO_MODIFICATION_ALLOWED_ERR = 7`

说明发生了修改只读的、不允许修改的节点的操作。`Entity`、`EntityReference`、`Notation` 节点和它们的子孙节点都是只读的。

`unsigned short NOT_FOUND_ERR = 8`

说明在期望的位置没有找到指定的节点。

`unsigned short NOT_SUPPORTED_ERR = 9`

说明当前的 DOM 实现不支持某个属性或方法。

`unsigned short INUSE_ATTRIBUTE_ERR = 10`

说明在一个 `Attr` 节点已经关联到另一个 `Element` 节点时，发生了把一个 `Attr` 节点关联到另一个 `Element` 节点的操作。

`unsigned short INVALID_STATE_ERR = 11[2 级 DOM]`

说明使用了处于不允许使用状态或不再允许使用状态的对象。

`unsigned short SYNTAX_ERR = 12[2 级 DOM]`

说明指定的字符串含有语法错误。通常由 CSS 属性声明使用。

`unsigned short INVALID_MODIFICATION_ERR = 13[2 级 DOM]`

说明发生了修改 `CSSRule` 对象或 `CSSValue` 对象的操作。

`unsigned short NAMESPACE_ERR = 14[2 级 DOM]`

说明有涉及到元素或性质的名字空间的错误。

`unsigned short INVALID_ACCESS_ERR = 15[2 级 DOM]`

说明以一种当前的实现不支持的方法访问对象。

属性

`unsigned short code`

错误代码，提供了引发异常的原因的详细情况。该属性的合法值（和它们的含义）由前面列出的常量定义。

描述

当错误使用或在不适合的环境中使用某个 DOM 属性或方法时，就会抛出一个 DOMException 对象。code 属性的值说明了发生的异常的一般类型。注意，读写对象的属性或调用对象的方法时，都有可能抛出 DOMException。

在对象的属性和方法的描述部分，列出了它们可能抛出的异常。但要注意，这些列表省略了某些通常抛出的异常。当想要修改只读节点（如 Entity 节点或它的某个子孙节点）的操作时，代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 就会被抛出。因此，Node 接口（和它的子接口）的大多数方法和读/写属性都可能抛出该异常。因为只读节点只出现在 XML 文档中，不出现在 HTML 文档中，而且它普遍地应用于 Node 对象的方法和可写属性，所以这些方法和属性的描述部分省略了 NO_MODIFICATION_ALLOWED_ERR 异常。

同样，许多返回字符串的 DOM 方法和属性都可以抛出代码为 DOMSTRING_SIZE_ERR 的 DOMException 异常，该异常说明返回的文本太长，当前 JavaScript 实现中的字符串容纳不下它。虽然理论上许多属性和方法都可能抛出这种异常，但在实际中它很少出现，所以这些属性和方法的描述部分省略了该异常。

注意，DOM 中的所有异常都由 DOMException 通知。必须处理事件和事件处理程序的异常，会抛出 EventException 对象。涉及 DOM Range 模块的异常，会抛出 RangeException 异常。

参阅

EventException、RangeException

DOMImplementation

1 级核心 DOM

独立于任何特殊文档的方法

也实现

DOMImplementationCSS、HTMLDOMImplementation

如果 DOM 实现支持 HTML 和 CSS 模块，那么 DOMImplementation 对象还会实现 DOMImplementationCSS 接口和 HTMLDOMImplementation 接口。为了方便起见，与核心的 DOMImplementation 方法一起，列出了这些普通接口的方法。

方法

`createCSSStyleSheet()` [2 级 DOM CSS]

这个 `DOMImplementationCSS` 方法将创建新的 `CSSStyleSheet` 对象。

`createDocument()` [2 级 DOM]

创建带有指定类型的根元素(返回的 `Document` 对象的 `documentElement` 属性)的新 `Document` 对象。

`createDocumentType()` [2 级 DOM]

创建新的 `DocumentType` 节点。

`CreateHTMLDocument()` [2 级 DOM HTML]

这个 `HTMLDOMImplementation` 方法将创建新的 `HTMLDocument` 对象, 并且用 `<html>`、`<head>`、`<title>`、`<body>` 元素来填充它。

`hasFeature()`

检查当前实现是否支持具有指定特性的版本。

描述

`DOMImplementation` 接口和它的 `DOMImplementationCSS` 与 `HTMLDOMImplementation` 子接口是占位符, 存放不专属于任何特定 `Document` 对象, 对 DOM 实现来说是“全局性”的方法。可以通过任何 `Document` 对象的 `implementation` 属性获得对 `DOMImplementation` 对象的引用。

参阅

Type of: `Document.implementation`

`DOMImplementation.createCSSStyleSheet()` 2 级 DOM CSS

创建一个 `CSSStyleSheet` 对象

摘要

```
CSSStyleSheet createCSSStyleSheet(String title,  
                                   String media)  
  
throws DOMException;
```

参数

`title`

样式表的标题。

media

要应用样式表的媒体类型列表，用逗号分隔。

返回值

一个 `CSSStyleSheet` 对象。

抛出

如果参数 *media* 格式错误，将抛出代码为 `SYNTAX_ERR` 的 `DOMException` 异常。

描述

该方法将创建一个新的 `CSSStyleSheet` 对象。但要注意，从 2 级 DOM 起，DOM 标准没有再定义把新创建的 `CSSStyleSheet` 对象关联到文档的方法。

`createCSSStyleSheet()` 方法不是由 `DOMImplementation` 接口定义的，而是由 `DOMImplementationCSS` 子接口定义的。如果一个实现支持 CSS 特性，那么它的 `DOMImplementation` 对象就会实现该方法。

DOMImplementation.createDocument() 2 级核心 DOM

创建一个新 Document 对象和指定的根元素

摘要

```
Document createDocument(String namespaceURI,  
                        String qualifiedName,  
                        DocumentType doctype)  
    throws DOMException;
```

参数

namespaceURI

为文档创建的根元素的名字空间的唯一标识符。如果没有名字空间，则为 `null`。

qualifiedName

为文档创建的根元素的名字。如果 *namespaceURI* 不为 `null`，该名字应该包括名字空间前缀和冒号。

doctype

新创建的 Document 对象的 `DocumentType` 对象。如果没有想得到的 `DocumentType` 对象，则为 `null`。

返回值

一个带有 `documentElement` 属性的 `Document` 对象设置为指定类型的 `Element` 根节点。

抛出

该方法将在下列环境中抛出具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

`qualifiedName` 含有不合法的字符。

`NAMESPACE_ERR`

`qualifiedName` 格式错误，或者 `qualifiedName` 和 `namespaceURI` 不匹配。

`NOT_SUPPORTED_ERR`

当前实现不支持 XML 文档，没有实现该方法。

`WRONG_DOCUMENT_ERR`

另一个文档使用了 `doctype`，或其他 `DOMImplementation` 对象创建了该 `doctype`。

描述

该方法将创建一个新的 `Document` 对象，并为文档指定它的根对象 `documentElement`。如果参数 `doctype` 不是 `null`，那么 `DocumentType` 对象的 `ownerDocument` 属性将被设置为新创建的文档。

该方法用于创建 XML 文档，只支持 HTML 的实现可能不支持它。用 `createHTMLDocument()` 方法可以创建新的 HTML 文档。

参阅

`createDocumentType()`、`createHTMLDocument()`

`DOMImplementation.createDocumentType()`

2 级核心 DOM

创建一个 `DocumentType` 节点

摘要

```
DocumentType createDocumentType(String qualifiedName,  
                                String publicId,
```

```
String systemId)  
throws DOMException;
```

参数

qualifiedName

文档类型的名字。如果使用 XML 名字空间，该参数可能是一个限定名，用来指定名字空间前缀和本地名，两者之间用冒号分隔。

publicId

文档类型的公有标识符或 null。

systemId

文档类型的系统标识符或 null。该参数通常声明了 DTD 文件的本地文件名。

返回值

新创建的 `DocumentType` 对象，`ownerDocument` 属性为 null。

抛出

该方法将在下列环境中抛出具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

qualifiedName 含有不合法的字符。

`NAMESPACE_ERR`

qualifiedName 格式错误。

`NOT_SUPPORTED_ERR`

当前实现不支持 XML 文档，没有实现该方法。

描述

该方法将创建一个新的 `DocumentType` 节点。该方法只声明文档类型的一个外部子集。从 2 级 DOM 起，DOM 标准就不再提供声明内部子集的方法，返回的 `DocumentType` 对象不定义任何 `Entity` 节点和 `Notation` 节点。该方法只和 XML 文档一起使用，只支持 HTML 文档的实现不支持该方法。

DOMImplementation.createHTMLDocument() 2 级 DOM HTML

创建提纲式的 HTML 文档

摘要

```
HTMLDocument createHTMLDocument(String title);
```

参数*title*

文档的标题。该文本作为新创建的文档的 <title> 元素的内容。

返回值

新创建的 HTMLDocument 对象。

描述

该方法将创建新的 HTMLDocument 对象，它具有大纲式的文档树，其中包括指定的标题。返回对象的 documentElement 属性是一个 <html> 元素，这个根元素的子节点是 <head> 标记和 <body> 标记。<head> 元素的子节点是 <title> 标记，<title> 子节点子节点是指定的 *title* 字符串。

createHTMLDocument() 方法不是由 DOMImplementation 接口定义的，而是由 HTMLDOMImplementation 子接口定义的。如果一个实现支持 HTML 特性，它的 DOMImplementation 对象就会实现该方法。

参阅

DOMImplementation.createDocument()

DOMImplementation.hasFeature() 1 级核心 DOM

确定实现是否支持某个特性

摘要

```
boolean hasFeature(String feature,  
                   String version);
```

参数*feature*

特性名，用于判断哪个支持被测试。后面的表中列出了 2 级 DOM 标准支持的有效特性名的集合。特性名不区分大小写。

version

版本号，用于判断哪个支持被测试，或者为 null。如果该特性的所有版本都被支持，则为空串（""）。在 2 级 DOM 标准中，支持的版本号是 1.0 和 2.0。

返回值

如果当前实现完全支持指定特性的指定版本，返回值为 true，否则为 false。如果没有指定版本号，而且实现完全支持指定特性的所有版本，该方法也返回 true。

描述

W3C DOM 标准是模块化的，不要求每种实现都实现标准中的所有模块或特性。该方法用于检测一种 DOM 实现是否支持 DOM 标准的指定模块。在这个 DOM 参考手册中，每个条目的可用性信息都包括模块名。注意，虽然 Internet Explorer 5 和 5.5 都部分地支持 1 级 DOM 标准，但在 IE 6 之前，没有实现支持这个重要的方法。

下表列出了可以作为 *feature* 参数的模块名的完整集合。

特性	描述
Core	实现 Node、Element、Document、Text 和其他所有 DOM 实现都要求实现的基本接口；所有遵守 DOM 标准的实现都必须支持该模块
HTML	实现 HTMLElement、HTMLDocument 和其他 HTML 专有接口
XML	实现 Entity、EntityReference、ProcessingInstruction、Notation 和其他 XML 文档专用的节点类型
StyleSheets	实现描述普通样式表的简单接口
CSS	实现 CSS 样式表专有的接口
CSS2	实现 CSS2Properties 接口
Events	实现基本的事件处理接口
UIEvents	实现处理用户界面事件的接口
MouseEvents	实现处理鼠标事件的接口
HTMLEvents	实现处理 HTML 事件的接口
MutationEvents	实现处理文档变化事件的接口
Range	实现操作文档范围的接口
Traversal	实现进行高级文档遍历的接口
Views	实现处理文档视图的接口

例子

可以在代码中如下使用该方法:

```
// 检查浏览器是否支持2级 DOM Traversal API
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("Traversal", "2.0")) {
    // 如果是，使用此处的代码
}
else {
    // 如果不是，以其他方式遍历文档
}
```

参阅

Node.isSupported()

DOMImplementationCSS

参阅 DOMImplementation

Element

1 级核心 DOM

一个 HTML 元素或 XML 元素

Node → Element

子接口

HTMLElement

属性

readonly String tagName

元素的标记名。例如，对于 HTML<P> 元素，它是字符串“P”。对于 HTML 文档，无论标记名在文档源代码中的大小写形式是什么，都以大写形式返回它。XML 文档区分大小写，返回的标记名与它在文档源代码中的形式完全一致。该属性的值与 Node 接口的 nodeName 属性的值相同。

方法

getAttribute()

以字符串形式返回指定性质的值。

getAttributeNS() [2 级 DOM]

以字符串形式返回由本地名和名字空间 URI 指定的性质的值。只有使用名字空间的 XML 文档才使用该方法。

`getAttributeNode()`

以 Attr 节点的形式返回指定性质的值。

`getAttributeNodeNS()` [2 级 DOM]

以 Attr 节点的形式返回由本地名和名字空间 URI 指定的性质的值。只有使用名字空间的 XML 文档才使用该方法。

`getElementsByTagName()`

返回一个数组(从技术上说是 `NodeList`), 元素是具有指定标记名的所有 `Element` 节点的子孙节点, 它们在数组中的顺序就是出现在文档中的顺序。

`getElementsByTagNameNS()` [2 级 DOM]

与 `getElementsByTagName()` 相似, 只是元素标记名由本地名和名字空间 URI 指定。只有使用名字空间的 XML 文档才会使用该方法。

`hasAttribute()` [2 级 DOM]

如果该元素具有指定名字的性质, 则返回 `true`, 否则返回 `false`。注意, 如果在文档的源代码中明确设置了指定的性质, 或者文档的 DTD 为指定的性质设置了默认值, 该方法都返回 `true`。

`hasAttributeNS()` [2 级 DOM]

与 `hasAttribute()` 相似, 只是性质由本地名和名字空间 URI 共同指定。只有使用名字空间的 XML 文档才会使用该方法。

`removeAttribute()`

从元素中删除指定的性质。注意, 该方法只删除在文档的源代码中明确设置过的性质。如果 DTD 为该性质设置了默认值, 这个默认值将成为性质的新值。

`removeAttributeNS()` [2 级 DOM]

与 `removeAttribute()` 相似, 只是要删除的属性由本地名和名字空间 URI 共同指定。只有使用名字空间的 XML 文档才会使用该方法。

`removeAttributeNode()`

从元素的性质列表中删除指定的 Attr 节点。注意, 该方法只能删除在文档的源代码中明确设置过的性质。如果 DTD 为删除的性质设置了默认值, 那么将创建一个新的 Attr 节点, 用于表示该性质的默认值。

`setAttribute()`

把指定的性质设置为指定的字符串值。如果具有指定名字的性质还不存在, 则给元素添加一个新性质。

`setAttributeNS()` [2 级 DOM]

与 `setAttribute()` 相似，只是要设置的性质名由本地名和名字空间 URI 共同指定。只有使用名字空间的 XML 文档才会使用该方法。

`setAttributeNode()`

把指定的 Attr 节点添加到元素的性质列表。如果已经存在同名的性质，那么该性质的值将被替换。

`setAttributeNodeNS()` [2 级 DOM]

与 `setAttributeNode()` 相似，只是该方法适用于 `Document.createElementNS()` 返回的节点。只有使用名字空间的 XML 文档才会使用该方法。

描述

Element 接口表示 HTML 元素、XML 元素或标记。`tagName` 属性指定了元素的名字。`getElementsByTagName()` 方法提供了定位具有指定标记名的元素的子孙元素的方法。该接口的其他方法提供了访问元素性质的方法。如果在文档源代码中用 `id` 性质给元素一个惟一标识符，那么用 `Document.getElementById()` 方法就能容易地找到表示该文档元素的 **Element** 节点。用 `Document.createElement()` 方法，可以创建一个插入文档的新 **Element** 节点。

在 HTML 文档（和许多 XML 文档）中，所有性质都有简单的字符串值。可以用 `getAttribute()` 方法和 `setAttribute()` 方法实现需要的性质操作。

如果你在使用的 XML 文档的属性值有一部分为实体引用，那么就不得不使用 **Attr** 对象和它们的节点子树。用 `getAttributeNode()` 方法和 `setAttributeNode()` 方法可以获取和设置 **Attr** 对象，也可以在 **Node** 接口的 `attributes[]` 数组中遍历 **Attr** 节点。如果正在使用的 XML 文档使用了 XML 名字空间，则需要使用名字以“NS”结尾的各种方法。

在 1 级 DOM 标准中，`normalize()` 方法属于 **Element** 接口。在 2 级 DOM 标准中，`normalize()` 方法则属于 **Node** 接口。所有 **Element** 节点都继承了该方法，仍旧可以使用它。

参阅

HTMLElement、**Node**

Type of: `Attr.ownerElement`、`Document.documentElement`

Passed to: `Document.getOverrideStyle()`、`AbstractView.getComputedStyle()`

Returned by: `Document.createElement()`、`Document.createElementNS()`、`Document.getElementById()`

Element.getAttribute()

1 级核心 DOM

返回指定性质的字符串值

摘要

```
String getAttribute(String name);
```

参数

name

要返回值的性质的名字。

返回值

指定性质的字符串值。如果在文档中没有设置该性质的值，而且文档类型也没有设置它的默认值，那么返回值为空串 “”。

描述

方法 `getAttribute()` 将返回一个元素的指定性质的值。在 HTML 文档中，性质值都是字符串，该方法将返回完整的性质值。注意，表示 HTML 元素的对象还实现了 `HTMLElement` 接口和它的一个有特定标记的子接口。因此，像 `Element` 对象的属性一样可以直接访问标准 HTML 标记的所有标准属性。

在 XML 文档中，性质值不能直接作为元素属性，必须通过调用来查询它们。对于许多 XML 文档来说，`getAttribute()` 方法适用于实现这一点。但要注意，XML 性质可以含有实体引用，要获取这种性质的完整细节，必须调用 `getAttributeNode()` 方法获取一个 `Attr` 节点，它的子树表示完整性质值。通过从 `Node` 接口继承的 `attributes[]` 数组也可以访问元素的 `Attr` 节点。对于使用名字空间的 XML 文档来说，需要使用 `getAttributeNS()` 方法或 `getAttributeNodeNS()` 方法。

例子

接下来的代码说明了两种获取 HTML 元素的性质值的方法：

```
// 获取文档中所有图像
var images = document.body.getElementsByTagName('IMG');
// 获取第一个图像的 SRC 性质
var src0 = images[0].getAttribute("SRC");
// 通过读取属性获取第二个图像的 SRC 性质
var src1 = images[1].src;
```

参阅

Element.getAttributeNode()、Node.attributes

Element.getAttributeNode()

1 级核心 DOM

返回指定性质的 Attr 节点

摘要

Attr getAttributeNode(String name);

参数

name

想获取的性质的名字。

返回值

一个 Attr 节点，它的子孙表示指定的性质的值。如果该元素没有这样的性质，则为 null。

描述

方法 getAttributeNode() 将返回一个 Attr 节点，表示指定的性质的值。注意，通过从 Node 接口继承的 attributes 属性也可以获取该 Attr 节点。

性质值由 Attr 节点的子孙表示。在 HTML 文档中，一个 Attr 节点只有一个 Text 节点的子节点，调用 getAttribute() 方法查询性质的值更容易，它返回的值是字符串。只有在使用性质值含有实体引用的 XML 文档时，才有必要使用 getAttributeNode() 方法。

参阅

Element.getAttribute()、Element.getAttributeNodeNS()、Node.attributes

Element.getAttributeNodeNS()2 级核心 DOM

返回具有名字空间的性质的 Attr 节点

摘要

```
Attr getAttributeNodeNS(String namespaceURI,  
                        String localName);
```

参数

namespaceURI

惟一标识性质的名字空间的 URI。若没有名字空间，则该参数为 null。

localName

声明该性质在名字空间中的名字的标识符。

返回值

一个 Attr 节点，它的 value 表示指定性质的值。如果该元素没有这样的性质，则返回 null。

描述

该方法与 `getAttributeNode()` 方法相似，只是性质名由名字空间 URI 和在这个名字空间中定义的本地名共同指定。只有使用名字空间的 XML 文档才使用该方法。

参阅

`Element.getAttributeNode()`、`Element.getAttributeNS()`

Element.getAttributeNS()2 级核心 DOM

获取使用指定名字空间的性质的值

摘要

```
String getAttributeNS(String namespaceURI,  
                      String localName);
```

参数

namespaceURI

惟一标识性质的名字空间的 URI。若没有名字空间，则该参数为 null。

localName

声明该属性在名字空间中的名字的标识符。

返回值

指定性质的字符串值。如果在文档中没有明确设置该性质、而且文档类型也没有设置它的默认值，该方法将返回空串。

描述

该方法与 `getAttribute()` 方法相似，只是性质由名字空间 URI 和名字空间中的本地名指定。只有使用名字空间的 XML 文档才使用该方法。

参阅

`Element.getAttribute()`、`Element.getAttributeNodeNS()`

`Element.getElementsByTagName()`

1 级核心 DOM

找到具有指定标记名的子孙元素

摘要

```
Node[] getElementsByTagName(String name);
```

参数

name

想获取的元素的标记名，或者为“*”，表示应该返回所有的子孙节点，无论它们的标记名是什么。

返回值

一个数组（技术上说来，是 `NodeList` 对象），元素是 `Element` 对象，表示当前元素的子孙并具有指定标记名。

描述

该方法将遍历指定元素的子孙节点，返回一个 `Element` 节点的数组（实际上是 `NodeList` 对象），表示所有具有指定标记名的文档元素。元素在返回的数组中的顺序就是它们出现在文档源代码中的顺序。

注意，`Document` 接口也定义了 `getElementsByTagName()` 方法，它与该方法相似，但遍历整个文档，而不只是遍历某个元素的子孙节点。不要把该方法与

HTMLDocument.getElementsByName()方法并混淆,后者基于元素的name性质值检索元素,而不是基于它们的标记名检索元素。

例子

可以用下列代码找到文档中的所有<div>标记:

```
var divisions = document.body.getElementsByTagName('div');
```

用下面的代码可以找到<div>标记中的所有<p>标记:

```
var paragraphs = divisions[0].getElementsByTagName('p');
```

参阅

Document.getElementById()、Document.getElementsByTagName()、HTMLDocument.getElementsByName()

Element.getElementsByTagNameNS()

2 级核心 DOM

返回具有指定名字和名字空间的子孙元素

摘要

```
Node[] getElementsByTagNameNS(String namespaceURI,  
                               String localName);
```

参数

namespaceURI

惟一标识元素的名字空间的 URI。

localName

声明在名字空间中元素的名字的标识符。

返回值

一个数组(技术上说来,是一个NodeList对象),元素是Element对象,表示当前元素的子孙节点和具有指定的名字和名字空间的元素。

描述

该方法与getElementsByTagName()相似,只是想获取的元素的标记名被指定为名字空间URI和在名字空间中定义的本地名的组合。只有使用名字空间的XML文档才使用该方法。

参阅

`Document.getElementsByTagNameNS()`、`Element.getElementsByTagName()`

`Element.hasAttribute()`

2 级核心 DOM

判断当前元素是否具有指定的性质

摘要

```
boolean hasAttribute(String name);
```

参数

name

要使用的性质的名字。

返回值

如果当前元素具有指定的性质、或者为指定的性质设置了默认值，则返回 `true`，否则返回 `false`。

描述

该方法将判断一个元素是否具有指定的性质，但不返回那个性质的值。注意，如果文档中明确设置了指定的性质，或者文档类型为该性质设置了默认值，`hasAttribute()` 都返回 `true`。

参阅

`Attr.specified`、`Element.getAttribute()`、`Element.setAttribute()`

`Element.hasAttributeNS()`

2 级核心 DOM

判断当前元素是否具有指定的性质

摘要

```
boolean hasAttributeNS(String namespaceURI,  
                        String localName);
```

参数

namespaceURI

惟一标识属性的名字空间标识符，若没有名字空间，则为 `null`。

`localName`

声明在指定的名字空间中性质的名字。

返回值

如果该元素明确地设置了指定的值，或者给指定性质设置了默认值，则返回 `true`，否则返回 `false`。

描述

该方法与 `hasAttribute()` 方法相似，只是要检查的性质由名字空间和名字指定。只有使用名字空间的 XML 文档才使用该方法。

参阅

`Element.getAttributeNS()`、`Element.setAttributeNS()`

`Element.removeAttribute()`

1 级核心 DOM

从元素中删除指定的性质

摘要

```
void removeAttribute(String name);
```

参数

`name`

要删除的性质的名字。

抛出

如果该元素是只读的，不允许删除它的性质，则该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

描述

方法 `removeAttribute()` 将从当前元素中删除指定的性质。如果文档类型为指定的性质设置了默认值，那么接下来调用 `getAttribute()` 方法将返回那个默认值。删除不存在的性质或没有设置但具有默认值的性质的操作将被忽略。

参阅

`Element.getAttribute()`、`Element.setAttribute()`、`Node.attributes`

Element.removeAttribute()1 级核心 DOM

从元素中删除一个 Attr 节点

摘要

```
Attr removeAttributeNode(Attr oldAttr)  
    throws DOMException;
```

参数

oldAttr
要从元素中删除的 Attr 节点。

返回值

删除的 Attr 节点。

抛出

该方法将抛出具有下列代码的 DOMException 异常：

NO_MODIFICATION_ALLOWED_ERR
当前元素是只读的，不允许删除性质。

NOT_FOUND_ERR
oldAttr 不是当前元素的性质。

描述

该方法将从当前元素的性质集合中删除（并返回）一个 Attr 节点。如果 DTD 给删除的性质设置了默认值，那么该方法将添加一个新的 Attr 节点，表示这个默认值。用 `removeAttribute()` 方法代替该方法会更简单。

参阅

Attr、Element.removeAttribute()

Element.removeAttributeNS()2 级核心 DOM

删除由名字和名字空间指定的性质

摘要

```
void removeAttributeNS(String namespaceURI,  
                        String localName);
```

参数

namespaceURI

性质的名字空间的惟一标识符。若没有名字空间，则为 null。

localName

声明在指定的名字空间中性质的名字。

抛出

如果当前元素是只读的，不允许删除它的性质，该方法将抛出代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 异常。

描述

方法 `removeAttributeNS()` 与方法 `removeAttribute()` 相似，只是要删除的性质由名字和名字空间共同指定，而不是只由名字指定。只有使用名字空间的 XML 文档才可以使用该方法。

参阅

`Element.getAttributeNS()`、`Element.removeAttribute()`、`Element.setAttributeNS()`

Element.setAttribute()**1 级核心 DOM**

创建或改变元素的某个性质

摘要

```
void setAttribute(String name,  
                  String value)  
    throws DOMException;
```

参数

name

要创建或修改的性质的名字。

value

性质的字符串值。

抛出

该方法将抛出具有下列代码的 DOMException 异常：

INVALID_CHARACTER_ERR

参数 *name* 含有 HTML 性质名或 XML 性质名不允许使用的字符。

NO_MODIFICATION_ALLOWED_ERR

当前元素是只读的，不允许修改它的性质。

描述

该方法将把指定的性质设置为指定的值。如果不存在具有指定名字的性质，该方法将创建一个新性质。注意，表示HTML文档标记的Element对象还实现了HTMLElement接口和（通常实现了）它的专用于接口。作为快捷方式，这些接口定义了对应于每个标记的标准HTML性质的属性。通过设置相应的对象属性来设置HTML性质通常更容易。

参数 *value* 是纯字符串。如果你正在使用XML文档，应该用 `setAttributeNode()` 方法在性质值中加入实体引用。

例子

```
// 设置文档中所有链接的 TARGET 性质
var links = document.getElementsByTagName('A');
for (var i = 0; i < links.length; i++) {
    links[i].setAttribute('TARGET', "newwindow");
}
```

参阅

`Element.getAttribute()`、`Element.removeAttribute()`、`Element.setAttributeNode()`

Element.setAttributeNode()

1 级核心 DOM

给元素添加新的 Attr 节点

摘要

```
Attr setAttributeNode(Attr newAttr)
    throws DOMException;
```

参数

newAttr

表示要添加的性质或值要修改的性质的 Attr 节点。

返回值

被 *newAttr* 替换的 *Attr* 节点。如果没有替换任何性质，则返回 *null*。

抛出

该方法将抛出具有下列代码的 *DOMException* 异常：

INUSE_ATTRIBUTE_ERR

newAttr 已经是其他 *Element* 节点的性质集合的成员。

NO_MODIFICATION_ALLOWED_ERR

当前的 *Element* 节点是只读的，不允许修改它的性质。

WRONG_DOCUMENT_ERR

newAttr 的 *ownerDocument* 性质不同于要设置它的 *Element* 节点。

描述

该方法将给 *Element* 节点的性质集合添加新的 *Attr* 节点。如果当前 *Element* 已经具有一个同名的性质，该方法将用 *newAttr* 替换那个性质，返回被替换的 *Attr* 节点。如果不存在这样的性质，该方法将为 *Element* 定义一个新性质。

通常，用 *setAttribute()* 方法比用 *setAttributeNode()* 简单。但是，要为 XML 文档定义一个性质值包含实体引用的性质时，应该使用 *setAttributeNode()* 方法。

参阅

Attr、*Element.setAttribute()*

Element.setAttributeNodeNS()

2 级核心 DOM

给 *Element* 节点添加具有名字空间的 *Attr* 节点

摘要

```
Attr setAttributeNodeNS(Attr newAttr)  
    throws DOMException;
```

参数

newAttr

表示要添加的性质或值要修改的性质的 *Attr* 节点。

返回值

被 *newAttr* 替换的 *Attr* 节点，如果没有替换任何性质，则返回 *null*。

抛出

该方法抛出异常的原因和 *setAttributeNode()* 方法一样。此外，如果当前实现不支持 XML 文档和名字空间，该方法还将抛出代码为 *NOT_SUPPORTED_ERR* 的 *DOMException* 异常，说明该方法没有实现。

描述

该方法与 *setAttributeNode()* 方法相似，只是它适用于由名字和名字空间共同指定的表示特性的 *Attr* 节点。

只有使用名字空间的 XML 文档才使用该方法。不支持 XML 文档的浏览器可能无法实现该方法（即抛出代码为 *NOT_SUPPORTED_ERR* 的异常）。

参阅

Attr、*Element.setAttributeNS()*、*Element.setAttributeNode()*

Element.setAttributeNS()

2 级核心 DOM

创建或改变具有名字空间的性质

摘要

```
void setAttributeNS(String namespaceURI,  
                    String qualifiedName,  
                    String value)  
  
throws DOMException;
```

参数

namespaceURI

惟一标识要设置或创建的性质的名字空间的 URI。若没有名字空间，则为 *null*。

qualifiedName

性质的名字，作为名字空间的前缀，其后是冒号和名字空间中的一个名字。

value

性质的新值。

抛出

该方法将抛具有下列代码的 `DOMException` 异常：

`INVALID_CHARACTER_ERR`

`qualifiedName` 参数含有 **HTML** 性质名或 **XML** 性质名中不允许出现的字符。

`NAMESPACE_ERR`

`qualifiedName` 格式错误，或者 `qualifiedName` 和 `namespaceURI` 参数不匹配。

`NO_MODIFICATION_ALLOWED_ERR`

当前的元素是只读的，不允许修改它的性质。

`NOT_SUPPORTED_ERR`

当前的 DOM 实现不支持 XML 文档。

描述

该方法与 `setAttribute()` 方法相似，只是要创建或设置的性质由名字空间 URI 和限定名（由名字空间前缀、冒号和名字空间中的本地名构成）共同指定。除了可以改变一个性质的值以外，使用该方法还可以改变性质的名字空间前缀。

只有使用名字空间的 XML 文档才会使用该方法。不支持 XML 文档的浏览器可能不会实现该方法（即抛出代码为 `NOT_SUPPORTED_ERR` 的异常）。

参阅

`Element.setAttribute()`、`Element.setAttributeNode()`

ElementCSSInlineStyle

参阅 `HTMLElement`

Entity

1 级 DOM XML

XML DTD 中的一个实体

Node → Entity

属性

readonly String `notationName`

符号名（用于未解析的实体的），如果没有这样的符号名（对于已经解析的实

体), 则为 null。参阅 `DocumentType` 的 `notations` 属性, 它提供了通过名字查询 `Notation` 节点的方法。

`readonly String publicId`

实体的公共标识符。如果没有设置该标识符, 则为 null。

`readonly String systemId`

该实体的系统标识符。如果没有设置该标识符, 则为 null。

描述

这个不常用的接口表示 XML 文档类型定义 (DTD) 中的一个实体。HTML 文档从不使用它。

实体的名字由从 `Node` 接口继承的 `nodeName` 属性指定。实体的内容由 `Entity` 节点的子节点表示。注意, `Entity` 节点和它的子节点不属于文档树 (`Entity` 对象的 `parent-Node` 属性总为 null)。而文档包含一个或多个对实体的引用, 详见“`EntityReference`”参考页以获取更多信息。

实体是在文档的 DTD 中定义的, 既属于外部的 DTD 文件, 也属于定义当前文档专有的本地实体的“内部子集”。`DocumentType` 接口具有 `entities` 属性, 用它可以通名字查找 `Entity` 节点。这是获取对 `Entity` 节点的引用的惟一方法, 因为它们属于文档类型, 所以 `Entity` 节点绝不会出现在文档自身。

`Entity` 节点和它的所有子孙节点都是只读的, 不能对它们进行编辑和修改。

参阅

`DocumentType`、`EntityReference`、`Notation`

EntityReference

1 级 DOM XML

对 XML DTD 中定义的实体的引用

`Node` → `EntityReference`

描述

这种不常用的接口表示从 XML 文档到文档的 DTD 中定义的一个实体的引用。在 XML 文档和 HTML 文档中, 字符实体和预定义的实体 (如 `<`;) 总会被扩展, 而且 `EntityReference` 节点不会出现在 HTML 文档中, 所以专门处理 HTML 文档的程序设计者绝不会用到这个接口。还要注意, 有些 XML 解析器会扩展所有的实体引用。由这样的解析器创建的文档不包含 `EntityReference` 节点。

该接口没有定义自己的属性和方法。继承的属性 `nodeName` 指定了被引用的实体的名字。 `DocumentType` 接口的 `entities` 属性提供了用名字查找 `Entity` 对象的方法。但要注意, `DocumentType` 对象可能不包含具有指定名字的 `Entity` 节点 (例如, 无效的 XML 解析器不必解析 DTD 的“外部子集”)。在这种情况下, `EntityReference` 没有子节点。另一方面, 如果 `DocumentType` 包括具有指定名字的 `Entity` 节点, `EntityReference` 节点的子节点是 `Entity` 节点的子节点副本, 表示实体的内容。和 `Entity` 节点一样, `EntityReference` 节点和它们的子孙也是只读的, 不能对它们进行编辑和修改。

参阅

`DocumentType`

Returned by: `Document.createEntityReference()`

Event

2 级 DOM Events

一个事件的信息

子接口

`MutationEvent`, `UIEvent`

常量

这些常量是属性 `eventPhase` 的合法值, 它们表示事件传播的当前阶段:

`unsigned short CAPTURING_PHASE = 1`

事件处于它的捕捉阶段。

`unsigned short AT_TARGET = 2`

事件正由它的目标节点处理。

`unsigned short BUBBLING_PHASE = 3`

事件正在起泡。

属性

`readonly boolean bubbles`

如果事件是起泡类型的 (即只有调用 `stopPropagation()` 方法才能停止起泡), 则为 `true`, 否则为 `false`。

`readonly boolean cancelable`

如果用 `preventDefault()` 方法可以取消与事件关联的默认动作，则为 `true`，否则为 `false`。

`readonly EventTarget currentTarget`

当前处理该事件的 `Document` 节点。在捕捉和起泡的过程中，它不同于 `target` 属性。注意，所有节点都实现了 `EventTarget` 接口，`currentTarget` 属性可以引用任何节点，而不仅限于 `Element` 节点。

`readonly unsigned short eventPhase`

事件传播的当前阶段。前面的三个常量定义了该属性的合法值。

`readonly EventTarget target`

事件的目标节点，如生成事件的节点。注意，它可以是任何节点，而不仅限于 `Element` 节点。

`readonly Date timeStamp`

生成事件（技术上说来，是创建 `Event` 对象的事件）的日期和时间。实现并不是必须在这个域中提供有效的时间数据。如果它们没有提供该数据，`Date` 对象的 `getTime()` 方法将返回 0。参阅本书核心参考手册部分的 `Date` 对象。

`readonly String type`

当前 `Event` 对象表示的事件的名字。它与注册的事件处理程序同名，或者是事件处理程序属性删除前缀“on”。例如，“click”、“load”或“submit”。参阅第十九章的表 19-3，它是 DOM 标准定义的事件类型的完整列表。

方法

`initEvent()`

初始化新创建的 `Event` 对象的属性。

`preventDefault()`

通知浏览器不要执行与事件关联的默认动作（如果存在这样的动作）。如果事件类型不是可取消的，则该方法不起作用。

`stopPropagation()`

制止事件在传播过程的捕捉、目标处理或起泡阶段进一步传播。调用该方法后，该节点上处理该事件的处理程序将被调用，事件不再被分派到其他节点。

描述

这个接口表示在文档的某些节点上发生的事件，并包括该事件的详细情况。Event 的各种子接口定义了额外的属性，提供与特殊事件类型有关的情况。

许多事件类型都使用 Event 的专用子接口描述发生的事件。但 HTMLEvents 模块定义的事件类型则直接使用 Event 接口。这些事件类型包括“abort”、“blur”、“change”、“error”、“focus”、“load”、“reset”、“resize”、“scroll”、“select”、“submit”和“unload”。

参阅

EventListener、EventTarget、MouseEvent、UIEvent; 第十九章

Passed to: EventTarget.dispatchEvent()

Returned by: Document.createEvent()

Event.initEvent()

2 级 DOM Events

初始化新 Event 对象的属性

摘要

```
void initEvent(String eventTypeArg,  
               boolean canBubbleArg,  
               boolean cancelableArg);
```

参数

eventTypeArg

事件的类型。它可以是一种预定义的事件类型，如“load”或“submit”，也可以是你自己选择的一种定制类型。但以“DOM”开头的名字是保留的。

canBubbleArg

事件是否起泡。

cancelableArg

是否可以用 preventDefault() 方法取消事件。

描述

该方法将初始化 Document.createEvent() 方法创建的合成的 Event 对象的 type 属性、bubbles 属性和 cancelable 属性。只有在新创建的 Event 对象被 EventTarget.dispatchEvent() 方法分派之前，才能调用 Event.initEvent() 方法。

Event.preventDefault()

2 级 DOM Events

取消事件的默认动作

摘要

```
void preventDefault();
```

描述

该方法将通知 Web 浏览器不要执行与事件关联的默认动作（如果存在这样的动作）。例如，如果 type 属性是“submit”，在事件传播的任意阶段可以调用任意的事件处理程序，通过调用该方法，可以阻止提交表单。注意，如果 Event 对象的 cancelable 属性是 false，那么就是没有默认动作，或者不能阻止默认动作。无论哪种情况，调用该方法都没有作用。

Event.stopPropagation()

2 级 DOM Events

不再分派事件

摘要

```
void stopPropagation();
```

描述

该方法将停止事件的传播，阻止它被分派到其他 Document 节点。在事件传播的任何阶段都可以调用它。注意，虽然该方法不能阻止同一个 Document 节点上的其他事件处理程序被调用，但它可以阻止把事件分派到其他节点。

EventException

2 级 DOM Events

通知一个事件特有的异常或错误

常量

下面的常量定义了 EventException 对象的 code 属性的合法值。注意，该常量是 EventException 的静态属性，不是个别异常对象的属性。

```
unsigned short UNSPECIFIED_EVENT_TYPE_ERR = 0
```

Event 对象的 type 属性没有被初始化，或者为 null，也可以为空串。

属性

unsigned short code

错误代码，提供了引发异常的原因的详细情况。在 2 级 DOM 中，这个域只有一种可能的值，即上面定义的常量。

描述

EventException 对象由某些与事件相关的方法抛出，通知发生了某种类型的问题。（在 2 级 DOM 中，只有 EventTarget.dispatchEvent() 方法会抛出这种类型的异常。）

EventListener

2 级 DOM Events

一个事件处理函数

方法

handleEvent()

在 Java 这样的语言中，不允许将函数作为参数传递给其他函数，可以通过定义一个实现该接口的类，并包含 handleEvent() 方法的实现，来定义一个事件监听器。当事件发生时，系统将调用该方法，传递给它一个描述该事件的 Event 对象。

但在 JavaScript 中，可以只编写一个接受 Event 对象作为参数的函数，来定义一个事件处理程序。函数名无关紧要，函数本身则用于代替 EventListener 对象。参阅“示例”部分。

描述

该接口定义了事件监听器或事件处理程序的结构。在 Java 这样的语言中，事件监听器是一个对象，该对象定义了名为 handleEvent() 的方法，它只有一个参数，即一个 Event 对象。但在 JavaScript 中，以 Event 对象为参数的函数和没有参数的函数都可以作为事件监听器。

例子

```
// 该函数是用于 "submit" 事件的事件监听器
function submitHandler(e) {
    // 调用在另处定义的格式确认的函数
    if (!validate(e.target))
        e.preventDefault(); // 如果确认失败，不提交表单
}

// 用如下方法注册事件监听器
```

```
document.forms[0].addEventListener("submit", submitHandler, false);
```

参阅

Event、EventTarget; 第十九章

Passed to: EventTarget.addEventListener()、EventTarget.removeEventListener()

EventTarget

2 级 DOM Events

事件监听器的注册方法

方法

addEventListener()

给该节点的事件监听器集合添加一个事件监听器。

dispatchEvent()

把一个合成事件分派到当前节点。

removeEventListener()

从 Document 节点的监听器集合中删除一个事件监听器。

描述

在支持事件（即支持“Events”特性）的 DOM 实现中，文档树中的所有节点都会实现该接口，为每个节点维护一个事件监听器函数的集合或列表。用 addEventListener() 方法和 removeEventListener() 方法可以在这个集合中添加和删除监听器函数。

参阅

Event、EventListener; 第十九章

Type of: Event.currentTarget、Event.target、MouseEvent.relatedTarget

Passed to: MouseEvent.initMouseEvent()

EventTarget.addEventListener()

2 级 DOM Events

注册一个事件处理程序

摘要

```
void addEventListener(String type,
```



```
EventListener listener,  
boolean useCapture);
```

参数

type

调用事件监听器的事件类型。例如，“load”、“click”或“mousedown”。

listener

事件监听器函数。当指定类型的事件被分派到 Document 节点时，调用该函数。

useCapture

如果为 true，那么只有在事件传播的捕捉阶段，指定的 *listener* 才会被调用，更常用的值是 false，意味着在捕捉阶段不会调用 *listener*，而当该节点是实际的事件目标或事件从它的原始目标起泡到该节点时，调用 *listener*。

描述

该方法将把指定的事件监听器函数添加到当前节点的监听器集合中，以处理指定类型 *type* 的事件。如果 *useCapture* 为 true，则监听器被注册为捕捉事件监听器。如果 *useCapture* 为 false，它就注册为普通事件监听器。

`addEventListener()` 可能被调用多次，在同一个节点上为同一种类型的事件注册多个事件处理程序。但要注意，DOM 不能确定多个事件处理程序被调用的顺序。

如果一个事件监听器函数在同一个节点上用相同的 *type* 和 *useCapture* 注册了两次，那么第二次注册将被忽略。如果节点正在处理一个事件时注册新的事件监听器，则不会为那个事件调用新的事件监听器。

当用 `Node.cloneNode()` 方法或 `Document.importNode()` 方法复制一个 Document 节点时，不会复制为原始节点注册事件监听器。

参阅

Event、EventListener；第十九章

EventTarget.dispatchEvent()

2 级 DOM Events

给该节点分派一个合成事件

摘要

```
boolean dispatchEvent(Event evt)  
    throws EventException;
```

参数

evt

要分派的 Event 对象。

返回值

如果在事件传播过程中调用了 *evt* 的 `preventDefault()` 方法, 则返回 `false`, 否则返回 `true`。

抛出

如果 Event 对象 *evt* 没有被初始化, 或者它的 `type` 属性为 `null` 或空串, 该方法将抛出 `code` 属性为 `EventException.UNSPECIFIED_EVENT_TYPE_ERR` 的 `EventException` 对象。

描述

该方法将分派一个合成事件, 它由 `Document.createEvent()` 创建, 由 Event 接口或它的某个子接口定义的初始化方法初始化。调用该方法的节点将成为事件的目标节点, 该事件在捕捉阶段中第一次沿着文档树向下传播。如果该事件的 `bubbles` 属性为 `true`, 那么在事件的目标节点自身处理事件后, 它将沿着文档树向上起泡。

参阅

`Document.createEvent()`、`Event.initEvent()`、`MouseEvent.initMouseEvent()`

EventTarget.removeEventListener()

2 级 DOM Events

删除一个事件监听器

摘要

```
void removeEventListener(String type,  
                           EventListener listener,
```

```
boolean useCapture);
```

参数

type

要删除事件监听器的事件类型。

listener

要删除的事件监听器函数。

useCapture

如果要删除的是捕捉事件监听器，则为 true；如果要删除的是普通事件监听器，则为 false。

描述

该方法将删除指定的事件监听器函数。参数 *type* 和 *useCapture* 必须与调用 `addEventListener()` 方法的相应参数一样。如果没有找到与指定的参数匹配的事件监听器，该方法则什么都不做。

如果一个事件监听器函数被该方法删除，那么当前节点发生指定类型的事件时，就不再调用它。即使一个事件监听器被同一节点上为同类型的事件注册的另一个事件监听器删除，它也不再被调用。

HTMLAnchorElement

1 级 DOM HTML

HTML 文档中的超链接或锚 Node → Element → HTMLElement → HTMLAnchorElement

属性

下表列出了该接口定义的性质，它们对应于标记 `<a>` 的 HTML 性质。

对象属性	HTML 性质	描述
String accessKey	accessKey	快捷键
String charset	charset	目标文档的编码
String coords	coords	在 <code><map></code> 元素中使用
String href	href	超链接的 URL
String hreflang	hreflang	链接文档采用的语言
String name	name	锚的名字

对象属性	HTML 性质	描述
String rel	rel	链接类型
String rev	rev	反链接类型
String shape	shape	在 <map> 元素中使用
String tabIndex	tabindex	在跳格顺序中链接的位置
String target	target	显示目标文档的框架或窗口的名字
String type	type	目标文档的内容类型

方法

blur()

把键盘焦点从链接上移开。

focus()

滚动文档，使锚或链接可见，并将键盘焦点给予链接。

描述

该接口表示 HTML 文档中的 <a> 标记。href、name 和 target 是关键属性，表示该标记最常用的 HTML 性质。

从 HTMLDocument 接口的 links 属性和 anchors HTMLCollection 属性可以获得 HTMLAnchorElement 对象。

例子

```
// 获取文档中第一个超链接的目的地 URL
var url = document.links[0].href;
// 滚动文档，使名为 '_bottom_' 的锚可见
document.anchors['_bottom_'].focus();
```

参阅

客户端参考手册中的 Link 对象和 Anchor 对象

HTMLAnchorElement.blur()

1 级 DOM HTML

把键盘焦点从超链接上移开

摘要

void blur();



描述

对于允许超链接具有键盘焦点的Web浏览器来说,该方法将把键盘焦点从超链接上移开。

HTMLAnchorElement.focus()

1 级 DOM HTML

使链接或锚可见,并给予它键盘焦点

摘要

```
void focus();
```

描述

该方法将滚动文档,使指定的锚或超链接可见。如果当前元素是超链接,而且浏览器允许超链接具有键盘焦点,那么该方法还会给予该元素键盘焦点。

HTMLBodyElement

1 级 DOM HTML

HTML 文档的 <body> 标记 Node → Element → HTMLElement → HTMLBodyElement

属性

deprecated String aLink

性质 aLink 的值。声明了“活动的”链接的颜色,即鼠标移到某个链接上,按下还未释放时的颜色。

deprecated String background

性质 background 的值。声明了用作文档的背景纹理的图像的 URL。

deprecated String bgColor

性质 bgcolor 的值。声明了文档的背景颜色。

deprecated String link

性质 link 的值。声明了未被访问过的超链接的常规颜色。

deprecated String text

性质 text 的值。声明了文档的前景颜色(即文本的颜色)。

deprecated String vLink

性质 vLink 的值。声明了已经被访问过的超链接的常规颜色。

描述

HTMLBodyElement 接口表示文档的 <body> 标记。所有 HTML 文档都有 <body> 标记，即使它没有明确地出现在文档源代码中。可以通过 HTMLDocument 接口的 body 属性获取文档的 HTMLBodyElement 对象。

该对象的属性声明了文档的默认颜色和图像。尽管这些属性表示 <body> 性质的值，但 0 级 DOM 标准允许通过 Document 对象的（不同名）属性访问这些值。参阅客户端参考手册部分的 Document 对象。

虽然这些颜色和图像属性属于 HTMLBodyElement 接口，而不属于 Document 对象，但是注意，它们都是反对使用的，因为 HTML 4 标准反对使用它们表示的 <body> 性质。推荐的方法是用 CSS 样式设置文档的颜色和图像。

例子

```
document.body.text1 = "#ff0000";           // 以亮红色显示文本
document.fgColor = "#ff0000";              // 使用老式 0 级 DOM API
document.body.style.color = "#ff0000";      // 使用 CSS 样式
```

参阅

客户端参考手册的 Document 对象

HTMLCollection

1 级 DOM HTML

通过位置或名字访问的 HTML 元素的数组

属性

readonly unsigned long length

集合中的元素数。

方法

item()

返回集合中指定位置的元素。也可以在数组方括号中指定位置，而不是明确地调用该方法。

namedItem()

返回集合中 name 性质或 id 性质具有指定值的元素。如果没有这样的元素，则返回 null。可以在方括号中使用元素名，而不是明确地调用该方法。

描述

HTMLCollection对象是带有方法的HTML元素的集合，用它可以通_过元素在文档中的位置或他们的id性质、name性质获取元素。在JavaScript中，HTMLCollection对象的行为和只读的数组一样，可以使用JavaScript的方括号，通过编号或名字索引一个HTMLCollection对象，而不必调用item()方法和namedItem()方法。

HTMLDocument接口的许多属性（0级DOM的Document对象标准化的）都是HTMLCollection对象，它提供了访问文档元素（如表单、图像和链接）的便捷方式。HTMLCollection对象还为遍历HTML表单的元素、HTML表的行、表元和客户端图像的区域提供了便捷方式。

HTMLCollection对象是只读的，不能给它添加新元素，即使采用JavaScript的数组语法。它们是“活”的，即如果基本的文档改变了，那些改变通过所有HTMLCollection对象会立即显示出来。

例子

```
var c = document.forms;           // 这是一个表单元素的HTMLCollection
var firstform = c[0];             // 可以像数字数组一样使用它
var lastform = c[c.length-1];     // 长度属性给出了元素个数
var address = c["address"];       // 可以像关联数组一样使用它
var address = c.address;          // JavaScript也可以使用这种表示法
```

参阅

NodeList

Type of: HTMLDocument.anchors、HTMLDocument.applets、HTMLDocument.forms、HTMLDocument.images、HTMLDocument.links、HTMLFormElement.elements、HTMLMapElement.areas、HTMLSelectElement.options、HTMLTableElement.rows、HTMLTableElement.tBodies、HTMLTableRowElement.cells、HTMLTableSectionElement.rows

HTMLCollection.item()

1级DOM HTML

根据位置获取元素

摘要

Node item(unsigned long index);

参数

index

要返回的元素的位置。元素在HTMLCollection对象中出现的顺序与它们在文档源代码中出现的顺序一样。

返回值

指定 *index* 处的元素。如果 *index* 小于 0 或大于等于 *length* 属性，则返回 null。

描述

方法 *item()* 将返回 HTMLCollection 中的编码元素。在 JavaScript 中，将 HTMLCollection 作为数组处理，用数组的语法索引它更容易一些。

例子

```
var c = document.images; // 这是一个HTMLCollection
var img0 = c.item(0);    // 可以这样使用item()方法
var img1 = c[1];         // 但是这种符号比较容易且更常见
```

参阅

[NodeList.item\(\)](#)

HTMLCollection.namedItem()

1 级 DOM HTML

根据名字获取元素

摘要

`Node namedItem(String name);`

参数

name

要返回的元素的名字。

返回值

具有指定的 *name* 的元素，如果 HTMLCollection 中没有这样的元素，则返回 null。

描述

该方法将查找并返回HTMLCollection中具有指定名字的元素。如果某个元素的id性质值是指定的名字，则返回那个元素。如果没有找到这样的元素，则返回name性质为指定值的元素。如果这样的元素也不存在，namedItem()则返回 null。

注意，任何 HTML 元素都可能有 id 性质，但只有某些 HTML 元素（如表单、表单元素、图像和锚）可能具有 name 性质。

在 JavaScript 中，将 HTMLCollection 作为关联数组处理，并采用数组语法。将 name 放在 [] 之间查找元素更容易一些。

例子

```
var forms = document.forms;           // 表单的 HTMLCollection
var address = forms.namedItem("address"); // 查找 <form name='address' >
var payment = forms["payment"] // 简单语法：查找 finds <form name='payment' >
var login = forms.login;             // 所有的工作：查找 <form name='login' >
```

HTMLDocument

1 级 DOM HTML

HTML 文档树的根

Node → Document → HTMLDocument

属性

readonly HTMLCollection anchors

文档中所有锚元素的数组（HTMLCollection 对象）。为了与 0 级 DOM 兼容，该数组只包含设置了 name 性质的 <a> 元素，不包括用 id 性质创建的锚元素。

readonly HTMLCollection applets

文档中所有小程序的数组（HTMLCollection 对象）。它包括 <object> 标记和所有 <applet> 标记定义的小程序。

HTMLElement body

一个快捷属性，引用 HTMLBodyElement 对象，该对象表示当前文档的 <body> 标记。对于定义了框架集的文档，该属性引用最外层的 <frameset> 标记。

String cookie

允许设置或查询当前文档的 cookie。参阅客户端参考手册部分的“Document.cookie”。

readonly String domain

提供装载的文档的服务器的域名，如果不存在这样的域名，则为 null。可与客户端参考手册中可读可写的属性 Document.domain 进行比较。

readonly HTMLCollection forms

文档中所有 HTMLFormElement 对象的数组（HTMLCollection 对象）。

readonly HTMLCollection images

文档中所有标记的数组 (HTMLCollection 对象)。注意, 为了与0级DOM兼容, 该集合不包括由<object>标记定义的图像。

readonly HTMLCollection links

文档中所有超链接的数组 (HTMLCollection 对象)。它包括所有具有 href 性质的 <a> 标记和所有的 <area> 标记。

readonly String referrer

链接到当前文档的文档的 URL, 如果当前文档不是通过超级链接访问的, 则为 null。

String title

文档的 <title> 标记的内容。

readonly String URL

文档的 URL。

方法

close()

关闭由 open() 方法打开的文档流, 强制性地显示所有缓存的输出内容。

getElementById()

返回具有指定 id 的元素。在 2 级 DOM 中, 该方法从 Document 接口继承。

getElementsByName()

返回文档中 name 性质具有指定值的所有元素的节点的数组 (NodeList 对象)。

open()

打开一个流, 以便写入新文档的内容。注意, 该方法将擦去当前文档的所有内容。

write()

把 HTML 文本串添加到打开的文档。

writeln()

把 HTML 文本串添加到打开的文档, 后面附加一个换行符。

描述

该接口扩展了 Document 接口, 定义了 HTML 专用的属性和方法, 这些属性和方法提供了与 0 级 DOM 的 Document 对象的兼容性 (参阅客户端参考手册中的 Document 对

象)。注意，HTMLDocument 不具备 0 级 Document 对象的所有属性。设置文档颜色和背景图像的属性被改名，并转移到了 HTMLBodyElement 接口中。

最后要注意，在 1 级 DOM 中，HTMLDocument 接口定义了名为 getElementById() 的方法。在 2 级 DOM 中，该方法被转移到了 Document 接口中，现在由 HTMLDocument 接口继承，而不是由它定义。详情参见本参考手册中的“Document.getElementById()”参考条目。

参阅

Document.getElementById()、HTMLBodyElement；客户端参考手册中的 Document 对象

Returned by: HTMLDOMImplementation.createHTMLDocument()

HTMLDocument.close()

1 级 DOM HTML

关闭一个打开的文档，并显示它

摘要

```
void close();
```

描述

该方法将关闭 open() 方法打开的文档流，并强制性的显示出所有缓存的输出内容。详见客户端参考手册中的“Document.close()”参考条目。

参阅

HTMLDocument.open()、客户端参考手册中的 Document.close()

HTMLDocument.getElementById()

参阅 Document.getElementById()

HTMLDocument.getElementsByName()

1 级 DOM HTML

找到具有指定 name 性质的元素

摘要

```
Node[] getElementsByName(String elementName);
```

参数

elementName

name 性质的期望值。

返回值

name 性质具有指定值的元素的数组（实际是 NodeList 对象）。如果没有找到这样的元素，则返回空的 NodeList 对象，其 length 为 0。

描述

该方法将搜索 HTML 文档树，查找 name 性质具有指定值的 Element 节点，返回包含所有匹配元素的 NodeList 对象（可以将其作为数组）。如果没有匹配的元素，则返回的 NodeList 对象的 length 属性为 0。

不要将这个方法与 Document.getElementById() 方法混淆，后者查找的是具有唯一的 id 性质值的 Element 节点。也不要与 Document.getElementsByTagName() 方法混淆，它返回具有指定标记名的元素的 NodeList 对象。

参阅

Document.getElementById(), Document.getElementsByTagName()

HTMLDocument.open()

1 级 DOM HTML

打开一个新文档，抹去当前文档的内容

摘要

```
void open();
```

描述

该方法将抹去当前 HTML 文档的内容，开始一个新文档，新文档用 write() 方法或 writeln() 方法编写。调用 open() 方法打开一个新文档，用 write() 方法设置文档内容后，必须记住调用 close() 方法关闭文档，迫使它的内容显示出来。

属于被覆盖的文档的一部分的脚本或事件处理程序不能调用该方法，因为脚本或事件处理程序自身也会被覆盖。

参阅客户端参考手册部分的“Document.open()”条目。但要注意，该方法被标准化的版本只能用于创建新的 HTML 文档，不接受 0 级版本接受的 *mimetype* 参数。

例子

```
var w = window.open," ");          // 打开新窗口
var d = w.document;                // 获取它的 HTMLDocument 对象
d.open();                          // 打开文档以便写入
d.write("<h1>Hello world</h1> ");    // 向文档输出 HTML
d.close();                          // 结束文档并显示它
```

参阅

HTMLDocument.close()、HTMLDocument.open(); 客户端参考手册中的 Document.open()

HTMLDocument.write()

1 级 DOM HTML

向打开的文档添加 HTML 文本

摘要

```
void write(String text);
```

参数

text

要添加到文档的 HTML 文本。

描述

该方法将把指定的 HTML 文本添加到文档中，该文档必须是由 open() 方法打开的，而且还没有用 close() 方法关闭。

详见客户端参考手册部分的“Document.write()”条目，但要注意，0 级方法的标准化版本只接受一个字符串参数，不接受任意多个参数。

参阅

HTMLDocument.open(); 客户端参考手册部分的 Document.write()

HTMLDocument.writeln()

1 级 DOM HTML

把 HTML 文本和换行符添加到打开的文档

摘要

```
void writeln(String text);
```

参数

text

要添加到文档的 HTML 文本。

描述

该方法与 `HTMLDocument.write()` 方法相似。只是在附加的文本后，它还要附加一个换行符。在编写标记 `<pre>` 的内容时，该方法很有用。

参阅

客户端参考手册部分的 `Document.writeln()`

HTMLDOMImplementation

参阅 DOMImplementation

HTMLElement

1 级 DOM HTML

所有 HTML 元素的基础接口

也实现

`ElementCSSInlineStyle` 如果实现支持 CSS 样式表，那么所有实现该接口的对象都会实现 `ElementCSSInlineStyle` 接口。因为对 CSS 的支持在 Web 浏览器中非常常见，所以为了方便起见，这里列出了 `ElementCSSInlineStyle` 接口定义的 `style` 属性。

子接口

<code>HTMLAnchorElement</code>	<code>HTMLAppletElement</code>	<code>HTMLAreaElement</code>
<code>HTMLBRElement</code>	<code>HTMLBaseElement</code>	<code>HTMLBaseFontElement</code>
<code>HTMLBodyElement</code>	<code>HTMLButtonElement</code>	<code>HTMLDListElement</code>
<code>HTMLDirectoryElement</code>	<code>HTMLDivElement</code>	<code>HTMLFieldSetElement</code>
<code>HTMLFontElement</code>	<code>HTMLFormElement</code>	<code>HTMLFrameElement</code>
<code>HTMLFrameSetElement</code>	<code>HTMLHRElement</code>	<code>HTMLHeadElement</code>
<code>HTMLHeadingElement</code>	<code>HTMLHtmlElement</code>	<code>HTMLIFrameElement</code>
<code>HTMLImageElement</code>	<code>HTMLInputElement</code>	<code>HTMLIsIndexElement</code>
<code>HTMLLIElement</code>	<code>HTMLLabelElement</code>	<code>HTMLLegendElement</code>
<code>HTMLLinkElement</code>	<code>HTMLMapElement</code>	<code>HTMLMenuElement</code>

HTMLMetaElement	HTMLModElement	HTMLLOListElement
HTMLObjectElement	HTMLOptGroupElement	HTMLOptionElement
HTMLParagraphElement	HTMLParamElement	HTMLPreElement
HTMLQuoteElement	HTMLScriptElement	HTMLSelectElement
HTMLStyleElement	HTMLTableCaptionElement	HTMLTableCellElement
HTMLTableColElement	HTMLTableElement	HTMLTableRowElement
HTMLTableSectionElement	HTMLTextAreaElement	HTMLTitleElement
HTMLULListElement		

属性

readonly CSS2Properties style

为当前元素设置内联 CSS 样式的 style 性质的值。实际上，该性质不是由 HTMLElement 接口直接定义的，它由 ElementCSSInlineStyle 接口定义。如果浏览器支持 CSS 样式表，那么它的所有 HTMLElement 对象都会实现 ElementCSSInlineStyle 接口，定义这个 style 属性。该属性的值是一个对象，该对象实现了 CSSStyleDeclaration 接口和（更常用的）CSS2Properties 接口。

String className

元素的 class 性质的值，声明了 CSS 类的名字。注意，该属性名不是“class”，因为“class”是 JavaScript 中的保留字。

String dir

元素的 dir 性质的值，声明了文档文本的方向。

String id

id 性质的值。在一个文档中，没有两个元素具有相同的 id 值。

String lang

lang 性质的值，声明了文档的语言代码。

String title

title 性质的值，声明了适合显示在元素的“工具提示”中的描述性文本。

描述

该接口定义了表示所有 HTML 元素共享的性质的属性。所有 HTML 元素都实现了该接口，大多数元素实现了标记特有的子接口，这些接口为每种标记性质定义了属性。

除了这里列出的属性，客户端参考手册中的“HTMLElement”参考页还列出了文档中的所有 HTML 元素都支持的 0 级 DOM 事件处理程序属性。

HTMLElement 的属性表示所有 HTML 标记都使用的通用性，有些 HTML 标记不采用这些属性之外的属性。这些标记没有自己的标记指定的子接口，在文档树中，这种类型的元素由 HTMLElement 对象表示。没有标记指定接口的标记如下所示：

<abbr>	<acronym>	<address>	
<body>	<big>	<center>	<cite>
<code>	<dd>	<dfn>	<dt>
	<i>	<kbd>	<noframes>
<noscript>	<s>	<samp>	<small>
	<strike>		<sub>
<sup>	<tt>	<u>	<var>

从前面的“子接口”部分可以看到，许多 HTML 标记都没有标记指定的子接口。通常，名为 T 的标记的指定接口名为 HTMLTElement。例如，<head> 标记由 HTMLHeadElement 接口表示。有些情况下，两个或多个相关的标记共享一个接口，如标记 <h1> 到标记 <h6> 都由 HTMLHeadingElement 接口表示。

大多数标记指定的接口不过是为 HTML 标记的性质定义一个 JavaScript 属性。JavaScript 属性和 HTML 性质名字相同，采用全小写的形式（如 id），在 HTML 性质名由多个单词构成时，JavaScript 属性采用大小写混合的形式（如 className）。当 HTML 性质名是 Java 或 JavaScript 的保留字时，JavaScript 属性名有轻微改变。例如，所有 HTML 标记的 class 性质将变成 HTMLElement 接口的 className 属性，因为 class 是保留字。同样，<label> 标记和 <script> 标记的 for 性质将成为 HTMLLabelElement 和 HTMLScriptElement 接口的 htmlFor 属性，因为 for 是保留字。这些属性的含义直接对应于 HTML 标准定义的 HTML 性质，对它们的介绍不属于本书的范围。

下表列出了所有具有相应的 HTMLElement 子接口的 HTML 标记。对于每个标记来说，该表列出了接口名和它定义的属性及方法名。如果没有明确说明，所有属性都是可读可写的字符串。对于不是可读可写的字符串的属性，属性名前的方括号中声明了它的属性类型。因为这些接口和它们的属性直接映射到 HTML 元素和性质，所以大多数接口在本书中没有自己的参考页。要了解它们的详细情况，应该查询 HTML 的参考书。特殊的接口是定义了方法的接口和表示极其重要的标记（如 <body> 标记）的接口。表中用 * 标出了这些重要的接口，可以在这个参考手册中查找它们的详细情况。

HTML 标记	DOM 接口, 属性和方法
all tags	HTMLElement ^a : id, title, lang, dir, className
<a>	HTMLAnchorElement ^a : accessKey, charset, coords, href, hreflang, name, rel, rev, shape, [long] tabIndex, target, type, blur(), focus()
<applet>	HTMLAppletElement ^b : align ^b , alt ^b , archive ^b , code ^b , codeBase ^b , height ^b , hspace ^b , name ^b , object ^b , vspace ^b , width ^b
<area>	HTMLAreaElement: accessKey, alt, coords, href, [boolean] noHref, shape, [long] tabIndex, target
<base>	HTMLBaseElement: href, target
<basefont>	HTMLBaseFontElement ^b : color ^b , face ^b , size ^b
<blockquote>, <q>	HTMLQuoteElement: cite
<body>	HTMLBodyElement ^a : aLink ^b , background ^b , bgColor ^b , link ^b , text ^b , vLink ^b
 	HTMLBRElement: clear ^b
<button>	HTMLButtonElement: [readonly HTMLFormElement] form, accessKey, [boolean] disabled, name, [long] tabIndex, [readonly] type, value
<caption>	HTMLTableCaptionElement ^a : align ^b
<col>, <colgroup>	HTMLTableColElement ^a : align, ch, chOff, [long] span, vAlign, width
, <ins>	HTMLModElement: cite, dateTime
<dir>	HTMLDirectoryElement ^b : [boolean] compact ^b
<div>	HTMLDivElement: align ^b
<dl>	HTMLDListElement: [boolean] compact ^b
<fieldset>	HTMLFieldSetElement: [readonly HTMLFormElement] form
	HTMLFontElement ^b : color ^b , face ^b , size ^b
<form>	HTMLFormElement ^a : [readonly HTMLCollection] elements, [readonly long] length, name, acceptCharset, action, enctype, method, target, submit(), reset()
<frame>	HTMLFrameElement: frameBorder, longDesc, marginHeight, marginWidth, name, [boolean] noResize, scrolling, src, [readonly Document] contentDocument ^c
<frameset>	HTMLFrameSetElement: cols, rows

HTML 标记	DOM 接口, 属性和方法
<h1>, <h2>, <h3>, <h4>, <h5>, <h6>	HTMLHeadingElement: align ^b
<head>	HTMLHeadElement: profile
<hr>	HTMLHRElement: align ^b , [boolean] noShade ^b , size ^b , width ^b
<html>	HTMLHtmlElement: version ^b
<iframe>	HTMLIFrameElement: align ^b , frameBorder, height, longDesc, marginHeight, marginWidth, name, scrolling, src, width, [readonly Document] contentDocument ^c
	HTMLImageElement: align ^b , alt, [long] border ^b , [long] height, [long] hspace ^b , [boolean] isMap, longDesc, name, src, useMap, [long] vspace ^b , [long] width
<input>	HTMLInputElement ^a : defaultValue, [boolean] default- checked, [readonly HTMLFormElement] form, accept, accessKey, align ^b , alt, [boolean] checked, [boolean] disabled ^c , [long] maxLength, name, [boolean] readOnly, size, src, [long] tabIndex, type, useMap, value, blur(), focus(), select(), click()
<ins>	See
<isindex>	HTMLIsIndexElement ^b : [readonly HTMLFormElement] form, prompt ^b
<label>	HTMLLabelElement: [readonly HTMLFormElement] form, accessKey, htmlFor
<legend>	HTMLLegendElement: [readonly HTMLFormElement] form, accessKey, align ^b
	HTMLLIElement: type ^b , [long] value ^b
<link>	HTMLLinkElement: [boolean] disabled, charset, href, hreflang, media, rel, rev, target, type
<map>	HTMLMapElement: [readonly HTMLCollection of HTMLAreaElement] areas, name
<menu>	HTMLMenuElement ^b : [boolean] compact ^b
<meta>	HTMLMetaElement: content, httpEquiv, name, scheme
<object>	HTMLObjectElement: code, align ^b , archive, border ^b , codeBase, codeType, data, [boolean] declare, height, hspace ^b , name, standby, [long] tabIndex, type, useMap, vspace ^b , width, [readonly Document] contentDocument ^c

HTML 标记	DOM 接口, 属性和方法
	HTMLListElement: [boolean] compact ^b , [long] start ^b , type ^b
<optgroup>	HTMLOptGroupElement: [boolean] disabled, label
<option>	HTMLOptionElement ^a : [readonly HTMLFormElement] form, [boolean] defaultSelected, [readonly] text, [readonly long] index, [boolean] disabled, label, [boolean] selected, value
<p>	HTMLParagraphElement: align ^b
<param>	HTMLParamElement: name, type, value, valueType
<pre>	HTMLPreElement: [long] width ^b
<q>	See <blockquote>
<script>	HTMLScriptElement: text, htmlFor, event, charset, [boolean] defer, src, type
<select>	HTMLSelectElement ^a : [readonly] type, [long] selectedIndex, value, [readonly long] length, [readonly HTMLFormElement] form, [readonly HTMLCollection of HTMLOptionElement] options, [boolean] disabled, [boolean] multiple, name, [long] size, [long] tabIndex, add(), remove(), blur(), focus()
<style>	HTMLStyleElement: [boolean] disabled, media, type
<table>	HTMLTableElement ^a : [HTMLTableCaptionElement] caption, [HTMLTableSectionElement] tHead, [HTMLTableSectionElement] tFoot, [readonly HTMLCollection of HTMLTableRowElement] rows, [readonly HTMLCollection of HTMLTableSectionElement] tBodies, align ^b , bgColor ^b , border, cellPadding, cellSpacing, frame, rules, summary, width, createTHead(), deleteTHead(), createTFoot(), deleteTFoot(), createCaption(), deleteCaption(), insertRow(), deleteRow()
<tbody>, <tfoot>, <thead>	HTMLTableSectionElement ^a : align, ch, chOff, vAlign, [readonly HTMLCollection of HTMLTableRowElement] rows, insertRow(), deleteRow()

HTML 标记	DOM 接口, 属性和方法
<td>,<th>	HTMLTableCellElement ^a : [readonly long] cellIndex,abbr,align,axis,bgColor ^b ,ch,chOff,[long] colSpan,headers,height ^b ,[boolean] nowrap ^b ,[long] rowSpan,scope,vAlign,width ^b
<textarea>	HTMLTextAreaElement ^a : defaultValue,[readonly HTMLFormElement] form,accessKey,[long] cols,[boolean] disabled,name,[boolean] readOnly,[long] rows,[long] tabIndex,[readonly] type,value,blur(),focus(),select()
<tfoot>	See <tbody>
<th>	See <td>
<thead>	See <tbody>
<title>	HTMLTitleElement: text
<tr>	HTMLTableRowElement ^a : [readonly long] rowIndex,[readonly long] sectionRowIndex,[readonly HTMLCollection of HTMLTableCellElement] cells,align,bgColor ^b ,ch,chOff,vAlign,insertCell(),deleteCell()
	HTMLUListElement: [boolean] compact ^b , type ^b

a: 说明本书记述了这些接口。

b: 说明反对使用的元素和性质。

c: 说明添加到 HTML2 级 DOM 草案中的性质。

参阅

客户端参考手册部分的 HTMLElement

Type of: HTMLDocument.body

Passed to: HTMLSelectElement.add()

Returned by: HTMLTableElement.createCaption()、HTMLTableElement.createTfoot()、HTMLTableElement.createTHead()、HTMLTableElement.insertRow()、HTMLTableRowElement.insertCell()、HTMLTableSectionElement.insertRow()

HTMLFormElement

1 级 DOM HTML

HTML 文档中的 <form> 元素 Node → Element → HTMLFormElement → HTMLFormElement

属性

readonly HTMLCollection elements

表单中所有元素的数组（HTMLCollection 对象）。

readonly long length

表单中的表单元素数，它的值与 elements.length 相同。

除了上述属性，HTMLFormElement 还定义了下表中列出的属性，它们直接对应于 HTML 性质。

属性	HTML 性质	描述
String acceptCharset	acceptCharset	服务器可以接受的字符集
String action	action	表单处理程序的 URL
String enctype	enctype	表单的编码
String method	method	用于提交表单的 HTTP 方法
String name	name	表单的名字
String target	target	显示提交表单的结果的框架或窗口

方法

reset()

把所有表单元素重置为它们的默认值。

submit()

提交表单。

描述

该接口表示 HTML 文档中的 <form> 元素。elements 属性是一个 HTMLCollection 对象，它提供了访问所有表单元素的便捷方式。用方法 submit() 和 reset() 可以在程序控制下提交或重置表单。

详见客户端参考手册中的 Form 对象。

参阅

客户端参考手册部分的 *Form* 对象

Type of: *HTMLButtonElement.form*、*HTMLFieldSetElement.form*、*HTMLInputElement.form*、*HTMLIsIndexElement.form*、*HTMLLabelElement.form*、*HTMLLegendElement.form*、*HTMLObjectElement.form*、*HTMLOptionElement.form*、*HTMLSelectElement.form*、*HTMLTextAreaElement.form*

HTMLFormElement.reset()

1 级 DOM HTML

把表单元素重置为它们的默认值

摘要

```
void reset();
```

描述

该方法将把每个表单元素都重置为它的默认值。调用该方法的结果与用户点击 **Reset** 按钮的结果一样，只是它不会调用表单的 `onreset` 事件处理程序。

参阅

客户端参考手册部分的 *Form.reset()*

HTMLFormElement.submit()

1 级 DOM HTML

提交表单

摘要

```
void submit();
```

描述

该方法将把表单元素的值提交给表单的 `action` 属性指定的表单处理程序。它提交表单的方式与用户点击 **Submit** 按钮来提交表单的方式相同，只是它不会触发表单的 `onsubmit` 事件处理程序。

参阅

客户端参考手册部分的 *Form.submit()*

HTMLInputElement

1 级 DOM HTML

HTML 表单中的输入元素 Node → Element → HTMLElement → HTMLInputElement

属性

String accept

一个 MIME 类型的列表，其间用逗号分隔，声明了当该元素是 FileUpload 元素时，可以上载的文件类型。映射 HTML 性质 accept。

String accessKey

浏览器用于把键盘焦点转移到该输入元素的快捷键（一定是一个字符）。映射 HTML 性质 accessKey。

deprecated String align

该元素相对于周围文本的垂直对齐方式，或元素使用的左、右浮点。映射 HTML 性质 align。

String alt

不能显示该输入元素的浏览器采用的替代文本。当 type 是 image 时，该属性尤其有用。映射 HTML 性质 alt。

boolean checked

对于 Radio 和 Checkbox 输入元素，该属性声明了它是否被“选中”了。设置该属性会改变输入元素的外观。映射 HTML 性质 checked。

boolean defaultChecked

对于 Radio 和 Checkbox 输入元素，该属性存放的是 checked 性质的初始值，就像它出现在文档源代码中一样。在重置表单时，该属性将恢复到这个值。改变该属性的值，也会改变 checked 属性的值和元素的当前核选状态。

String defaultValue

对于 Text、Password 和 FileUpload 元素，该属性存放了元素显示的初始值。重置表单时，元素会恢复为这个值。改变这个属性的值也会改变 value 属性和当前显示的值。

boolean disable

如果为 true，则该输入元素将被禁用，用户输入不能访问它。映射 HTML 性质 disable。

readonly HTMLFormElement form

HTMLFormElement 对象，表示包含该输入元素的 <form> 元素。如果该输入元素不在表单中，则为 null。

long maxLength

对于 Text 元素或 Password 元素，该属性声明了允许用户输入的最多字符数。注意，它不同于 size 属性。映射 HTML 性质 maxLength。

String name

输入元素的名字，由 HTML 性质 name 设置。

boolean readOnly

如果为 true，而且该元素是 Text 元素或 Password 元素，则不允许用户在该元素中输入文本。映射 HTML 性质 readOnly。

String size

对于 Text 元素或 Password 元素，该属性声明了元素的字符宽度。映射 HTML 性质 size。参阅 maxLength。

String src

对于类型为 image 的输入元素，该属性声明了要显示的图像的 URL。映射 HTML 性质 src。

long tabIndex

在跳格顺序中，该输入元素的位置。映射 HTML 性质 tabindex。

String type

输入元素的类型。“描述”部分的表中类出了各种类型和它们的含义。映射 HTML 性质 type。

String useMap

对于类型为 image 的元素，该属性声明了为元素提供客户端图像映射的 <map> 元素的名字。

String value

在提交表单时传递给服务器端脚本的值。对于 Text、Password 和 FileUpload 元素，该属性是输入元素包含的文本。对于 Button、Submit 和 Reset 元素，该属性是显示在按钮上的文本。出于安全性的原因，FileUpload 元素的 value 属性是只读的。同样，Password 元素的该属性返回的值不包含用户实际输入的文本。

方法

`blur()`

把键盘焦点从该元素上移开。

`click()`

如果该输入元素是 `Button`、`Checkbox`、`Radio`、`Submit` 或 `Reset` 按钮，该方法将模拟在元素上的鼠标点击。

`focus()`

把键盘焦点转移到该输入元素。

`select()`

如果输入元素是 `Text`、`Password` 或 `FileUpload` 元素，该方法将选择元素显示的文本。在许多浏览器中，这意味着当用户接着输入字符时，被选中的文本将被删除，由新输入的字符代替。

描述

该接口表示一个 `<input>` 元素，它定义了一个 HTML 输入元素（通常是在 HTML 表单中）。`HTMLInputElement` 接口可以表示各种类型的输入元素，这由它的 `type` 属性的值决定。下表列出了该属性可用的值和它们的含义。

类型	输入元素类型
<code>button</code>	按钮
<code>checkbox</code>	<code>Checkbox</code> 元素
<code>file</code>	<code>FileUpload</code> 元素
<code>hidden</code>	<code>Hidden</code> 元素
<code>image</code>	图形化的 <code>Submit</code> 按钮
<code>password</code>	用于口令的伪文本输入框
<code>radio</code>	互斥的 <code>Radio</code> 钮
<code>reset</code>	<code>Reset</code> 按钮
<code>text</code> (默认值)	单行文本输入框
<code>submit</code>	<code>Submit</code> 按钮

有关 HTML 表单和表单元素的更多信息，请参阅第十五章。还要注意，在客户端参考手册中，每种类型的表单输入元素都有自己的参考页。

参阅

HTMLFormElement、HTMLOptionElement、HTMLSelectElement、HTMLTextAreaElement；第十五章：客户端参考手册部分的 Input 对象和它的子类（Button、Checkbox、FileUpload、Hidden、Password、Radio、Reset、Submit 和 Text）

HTMLInputElement.blur()

1 级 DOM HTML

把键盘焦点从该元素中移开

摘要

```
void blur();
```

描述

该方法将把键盘焦点从表单元素中移开。

HTMLInputElement.click()

1 级 DOM HTML

模拟鼠标在表单元素上的点击

摘要

```
void click();
```

描述

该方法将模拟鼠标在 Button、Checkbox、Radio、Reset 或 Submit 表单元素上的点击。它不会触发输入元素的 onclick 事件处理程序。

在调用 Button 元素的 click() 方法时，它可能（或不可能）会生成按钮点击的视觉效果，但它没有其他的效果，因为它没有触发按钮的 onclick 事件处理程序。对于 Checkbox 元素，它将切换 checked 属性。它可以使没有被选中的 Radio 元素变成被选中的状态，但不管已选中的元素。在调用 Reset 元素和 Submit 元素的 click() 时，该方法将使表单被重置或提交。

HTMLInputElement.focus()

1 级 DOM HTML

给予该元素键盘焦点

摘要

```
void focus();
```

描述

该方法将把键盘焦点转移到当前元素，使用户可以不必点击它，就能与它进行交互。

HTMLInputElement.select()

1 级 DOM HTML

选择 Text 元素的内容

摘要

```
void select();
```

描述

该方法将选中 Text 元素、Password 元素和 FileUpload 元素显示的文本。在大多数浏览器中，这意味着用户接下来输入的字符将替换当前文本，而不是把文本附加到其后。

HTMLOptionElement

1 级 DOM HTML

HTML 表单中的 <option> 元素

Node → Element → HTMLElement →

HTMLOptionElement

属性

boolean defaultSelected

<option> 元素的 selected 性质的初始值。如果表单被重置，selected 属性将被重置为这个属性的值。设置该属性还会改变 selected 属性的值。

Boolean disabled

如果为 true，该选项将被禁用，用户不能选择它。映射 HTML 性质 disable。

readonly HTMLFormElement form

对包含该元素的 <form> 元素的引用。

readonly long index

这个 <option> 元素在包含它的 <select> 元素中的位置。

String label

选项显示的文本。映射 HTML 性质 label。如果没有设置该属性，将采用 <option> 元素的纯文本内容。

boolean selected

该选项的当前状态, 如果为 true, 说明该选项被选中。该属性的初始值由 HTML 性质 selected 设置。

readonly String text

<option> 元素包含的纯文本。该文本是选项的标签。

String value

在提交表单时, 如果该选项被选中了, 该属性是要随表单一起提交的值。映射 HTML 性质 value。

描述

该接口表示 <select> 元素中的 <option> 元素。

参阅

HTMLFormElement、HTMLInputElement、HTMLSelectElement; 客户端参考手册中的 Option 和 Select: 第十五章

HTMLSelectElement

1 级 DOM HTML

HTML 表单中的 <select> 元素

Node → Element → HTMLElement →

HTMLSelectElement

属性

boolean disabled

如果为 true, 这个 <select> 元素将被禁用, 用户不能和它交互。映射 HTML 性质 disabled。

readonly HTMLFormElement form

包含该元素的 <form> 元素。

readonly long length

这个 <select> 元素包含的 <option> 元素的个数。与 options.length 相同。

boolean multiple

如果为 true, 那么这个 <select> 元素允许选中多个选项。否则, 所有选项是互斥的, 一次只能选中一个选项。映射 HTML 性质 multiple。

String name

该表单元素的名字。映射 HTML 性质 name。

readonly HTMLCollection options

表示这个 <select> 元素中包含的 <option> 元素的 HTMLOptionElement 对象的数组 (HTMLCollection 对象), 存在 <option> 元素的顺序就是它们在 <select> 中出现的顺序。

long selectedIndex

被选中的选项在 options 数组中的位置。如果没有选中选项, 该属性的值为 -1。如果多个选项被选中, 该属性将返回第一个被选中的选项的下标。

long size

同时显示的选项个数。如果该属性值为 1, <select> 元素通常用下拉菜单或列表显示选项。如果它的值大于 1, <select> 元素通常用大小固定的列表控件显示选项, 如果必要, 该控件还可以具有滚动条。映射 HTML 性质 size。

long tabIndex

该元素在跳格顺序中的位置。映射 HTML 性质 tabindex。

readonly String type

如果 multiple 属性为 true, 该属性就是 “select-multiple” (多选)。否则它为 “select-one” (单选)。

方法

add()

在 options 数组中插入新的 HTMLOptionElement 对象, 既可以附加在该数组的尾部, 也可以插入到指定的选项之前。

blur()

移开键盘焦点。

focus()

把键盘焦点转移到该元素。

remove()

删除指定位置的 <option> 元素。

描述

该接口表示 HTML 表单中的 <select> 元素。options 属性提供了访问它包含的 <option> 元素集合的便捷方式。用方法 add() 和 remove() 可以很方便地修改选项集合。

参阅

HTMLFormElement、HTMLOptionElement; 客户端参考手册部分的 Option 对象和 Select 对象; 第十五章

HTMLSelectElement.add()

1 级 DOM HTML

插入一个 <option> 元素

摘要

```
void add(HTMLElement element,  
         HTMLElement before)  
throws DOMException;
```

参数

element

要添加的 HTMLOptionElement 对象。

before

options 数组中的元素, 新的 *element* 要插入到该元素之前。如果该参数为 null, *element* 将附加到 options 数组的尾部。

抛出

如果参数 *before* 指定的对象不是 options 数组的成员, 该方法将抛出代码为 NOT_FOUND_ERR 的 DOMException 异常。

描述

该方法将把一个新的 <option> 元素添加到 <select> 元素中。 *element* 是一个 HTMLOptionElement 对象, 表示要添加的 <option> 元素。 *before* 声明的 HTMLOptionElement 对象, *element* 元素要插入到它前面。如果 *before* 是 OPTGROUP 的一部分, *element* 总是作为该组的一部分插入。如果 *before* 为空, *element* 则成为 <select> 元素的最后一个子元素。

参阅

客户端参考手册部分的 Select 对象

HTMLSelectElement.blur()1 级 DOM HTML

把键盘焦点从该元素移走

摘要

```
void blur();
```

描述

该方法将把键盘焦点从该元素移走。

HTMLSelectElement.focus()1 级 DOM HTML

给予元素键盘焦点

摘要

```
void focus();
```

描述

该方法将把键盘焦点转移到这个 <select> 元素，使用户用键盘就能与它交互，而不必使用鼠标。

HTMLSelectElement.remove()1 级 DOM HTML

删除一个 <option> 元素

摘要

```
void remove(long index);
```

参数

index

要删除的 <option> 元素在 options 数组中的位置。

描述

该方法将删除 options 数组中指定位置的 <option> 元素。如果指定的 *index* 小于 0 或大于等于选项数，*remove()* 方法将忽略它，什么都不做。

参阅

客户端参考手册中的 Select 对象

HTMLTableCaptionElement

1 级 DOM HTML

HTML 表中的 <caption> 元素

Node → Element → HTMLElement → HTMLTableCaptionElement

属性

deprecated String align

标题相对于表的水平对齐方式。align性质的值。该属性被反对使用，推荐使用 CSS 样式。

描述

HTML 表中的 <caption> 元素。

参阅

Type of: HTMLTableElement.caption

HTMLTableCellElement

1 级 DOM HTML

HTML 表中的 <td> 或 <th> 表元

Node → Element → HTMLElement → HTMLTableCellElement

属性

readonly long cellIndex

表元在它的行中的位置。

除了 cellIndex 属性，该接口还定义了下表列出的属性，它们直接对应于 <td> 和 <th> 元素的 HTML 性质。

属性	性质	描述
String abbr	abbr	参阅 HTML 标准
String align	align	表元的水平对齐方式
String axis	axis	参阅 HTML 标准
deprecated String bgColor	bgcolor	表元的背景颜色
String ch	char	对齐字符
String chOff	choff	对齐字符偏移量
long colSpan	colspan	表元跨越的列数
String headers	headers	表元头的 id 值
deprecated String height	height	表元的高度，以像素计

属性	性质	描述
deprecated boolean nowrap	nowrap	表元不自动换行
long rowspan	rowspan	表元跨越的行数
String scope	scope	头表元的作用域
String valign	valign	表元的垂直对齐方式
deprecated String width	width	表元的宽度，以像素计

描述

该接口表示 HTML 表中的 <td> 和 <th> 元素。

HTMLTableColElement

1 级 DOM HTML

HTML 表中的 <col> 或 <colgroup> 元素

Node → Element → HTMLElement →
HTMLTableColElement

属性

该接口定义了下表列出的属性，它们直接对应于 <col> 或 <colgroup> 元素的 HTML 性质。

属性	性质	描述
String align	align	默认的水平对齐方式
String ch	char	默认的对齐字符
String chOff	choff	默认的对齐字符偏移量
long span	span	该元素表示的列数
String valign	valign	默认的垂直对齐方式
String width	width	列的宽度

描述

该接口表示 HTML 表中的 <col> 或 <colgroup> 元素。

HTMLTableElement

1 级 DOM HTML

HTML 文档中的 <table>

Node → Element → HTMLElement →
HTMLTableElement

属性

HTMLTableCaptionElement caption

对表的 <caption> 元素的引用，如果不存在该元素，则为 null。

readonly HTMLCollection rows

HTMLTableRowElement 对象的数组 (HTMLCollection)，表示表中的所有列。
这包括 <thead>、<tfoot> 和 <tbody> 中定义的所有行。

readonly HTMLCollection tBodies

HTMLTableSectionElement 对象的数组 (HTMLCollection)，表示表中的所有
<tbody> 段。

HTMLTableSectionElement tFoot

表的 <tfoot> 元素，如果不存在该元素，则为 null。

HTMLTableSectionElement tHead

表的 <thead> 元素，如果不存在该元素，则为 null。

除了上面列出的属性，该接口还定义了下表列出的属性，它们表示 <table> 元素的 HTML 性质。

属性	性质	描述
deprecated String align	align	表在文档中的水平对齐方式
deprecated String bgColor	bgcolor	表的背景颜色
String border	border	表的边线的宽度
String cellPadding	cellpadding	表元内容和边线之间的距离
String cellSpacing	cellspacing	表元边线之间的距离
String frame	frame	绘制哪种表边线
String rules	rules	在表中何处绘制行线
String summary	summary	表的概述
String width	width	表的宽度

方法

createCaption()

返回表现有的 <caption> 元素，如果不存在这样的元素，则创建（并插入）一个新的 <caption>。

createTFoot()

返回表现有的 <tfoot> 元素，如果不存在这样的元素，则创建（并插入）一个新的 <tfoot>。

`createTHead()`

返回表现有的 `<thead>` 元素，如果不存在这样的元素，则创建（并插入）一个新的 `<thead>`。

`deleteCaption()`

删除表的 `<caption>` 元素（如果表具有该元素）。

`deleteRow()`

删除表中指定位置的列。

`deleteTFoot()`

删除表的 `<tfoot>` 元素（如果表具有该元素）。

`deleteTHead()`

删除表的 `<thead>` 元素（如果表具有该元素）。

`insertRow()`

在表的指定位置插入一个新的空 `<tr>` 元素。

描述

这个接口表示 HTML 的 `<table>` 元素，它为查询和修改表的各个部分定义了大量便捷的属性和方法。虽然使用这些方法和属性处理表更容易，但它们和 DOM 的核心方法重复了。

参阅

`HTMLTableCaptionElement`、`HTMLTableCellElement`、`HTMLTableColElement`、`HTMLTableRowElement`、`HTMLTableSectionElement`

HTMLTableElement.createCaption()

1 级 DOM HTML

获取或创建 `<caption>` 元素

摘要

`HTMLElement createCaption();`

返回值

一个 `HTMLTableCaptionElement` 对象，表示该表的 `<caption>` 元素。如果该表已经有了标题，该方法只返回它。如果该表还没有 `<caption>` 元素，该方法将创建一个新的空 `<caption>` 元素，把它插入表中，并返回它。

HTMLTableElement.createTFoot()

1 级 DOM HTML

获取或创建 <tfoot> 元素**摘要**

```
HTMLElement createTFoot();
```

返回值

一个 HTMLTableSectionElement 对象，表示该表的 <tfoot> 元素。如果该表已经有了脚注，该方法只返回它。如果该表还没有脚注，该方法将创建一个新的空 <tfoot> 元素，把它插入表中，并返回它。

HTMLTableElement.createTHead()

1 级 DOM HTML

获取或创建 <thead> 元素**摘要**

```
HTMLElement createTHead();
```

返回值

一个 HTMLTableSectionElement 对象，表示该表的 <thead> 元素。如果该表已经有了表头，该方法只返回它。如果该表还没有表头，该方法将创建一个新的空 <thead> 元素，把它插入表中，并返回它。

HTMLTableElement.deleteCaption()

1 级 DOM HTML

删除表的 <caption> 元素**摘要**

```
void deleteCaption();
```

描述

如果该表有 <caption> 元素，这个方法将从文档树中删除它。否则，它什么也不做。

HTMLTableElement.deleteRow()1 级 DOM HTML

删除表的一行

摘要

```
void deleteRow(long index)  
    throws DOMException;
```

参数

index

指定了要删除的行在表中的位置。

抛出

如果 *index* 小于 0 或者大于等于表中行数，该方法将抛出代码为 INDEX_SIZE_ERR 的 DOMException 异常。

描述

该方法将删除表中指定位置的行。行的编码顺序就是它们在文档源代码中出现的顺序。<thead> 和 <tfoot> 中的行与表中其他行一起编码。

参阅

HTMLTableSectionElement.deleteRow()

HTMLTableElement.deleteTFoot()1 级 DOM HTML

删除表的 <tfoot> 元素

摘要

```
void deleteTFoot();
```

描述

如果该表有 <tfoot> 元素，这个方法将从文档树中删除它。否则，它什么也不做。

HTMLTableElement.deleteTHead()1 级 DOM HTML

删除表的 <thead> 元素

摘要

```
void deleteTHead();
```

描述

如果该表有 `<thead>` 元素，这个方法将从文档树中删除它。否则，它什么也不做。

HTMLTableElement.insertRow()

1 级 DOM HTML

给表添加新的空行

摘要

```
HTMLElement insertRow(long index)  
    throws DOMException;
```

参数

index

要插入新行的位置。

返回值

一个 `HTMLTableRowElement` 对象，表示新插入的行。

抛出

如果 *index* 小于 0 或者大于等于表中行数，该方法将抛出代码为 `INDEX_SIZE_ERR` 的 `DOMException` 异常。

描述

该方法将创建一个新的 `HTMLTableRowElement` 对象，表示一个 `<tr>` 标记，并把它插入表中的指定位置。

新的行将被插入 *index* 指定的行之前的位置。如果 *index* 等于表中的行数，新行将被附加到表的末尾。如果表初始时是空的，新行将被插入到一个新的 `<tbody>` 段，该段自身会被插入表中。

可以用 `HTMLTableRowElement.insertCell()` 方法给新创建的行添加内容。

参阅

`HTMLTableSectionElement.insertRow()`

HTMLTableRowElement

1 级 DOM HTML

HTML 表中的 <tr> 元素

Node → Element → HTMLElement →
HTMLTableRowElement

属性

readonly HTMLCollection cells

HTMLTableCellElement 对象的数组 (HTMLCollection 对象), 表示该行中的所有表元。

readonly long rowIndex

该行在表中的位置。

readonly long sectionRowIndex

该行在它的段 (即 <thead>、<tbody> 或 <tfoot> 元素) 中的位置。

除了上面列出的属性, 该接口还定义了下表列出的属性, 它们直接对应于 <tr> 元素的 HTML 性质。

属性	性质	描述
String align	align	表元在行中的默认水平对齐方式
deprecated String bgColor	bgcolor	行的背景颜色
String ch	char	表元在行中的对齐字符
String chOff	choff	表元在行中的对齐字符偏移量
String vAlign	valign	表元在行中的默认垂直对齐方式

方法

deleteCell()

删除行中指定的表元。

insertCell()

在 HTML 表的一行的指定位置插入一个空的 <td> 元素。

描述

该接口表示 HTML 表中的一行。

HTMLTableRowElement.deleteCell()1 级 DOM HTML

删除表行中的一个表元

摘要

```
void deleteCell(long index)
    throws DOMException;
```

参数

index

要删除的表元在行中的位置。

抛出

如果 *index* 小于 0 或者大于等于表中行的表元数, 该方法将抛出代码为 INDEX_SIZE_ERR 的 DOMException 异常。

描述

该方法将删除表行中指定位置的表元。

HTMLTableRowElement.insertCell()1 级 DOM HTML

在表行中插入一个新的空 <td> 元素

摘要

```
HTMLCellElement insertCell(long index)
    throws DOMException;
```

参数

index

插入新表元的位置。

返回值

一个 HTMLTableCellElement 对象, 表示新创建并被插入的 <td> 元素。

抛出

如果 *index* 小于 0 或者大于等于表中行的表元数, 该方法将抛出代码为 INDEX_SIZE_ERR 的 DOMException 异常。

描述

该方法将创建一个新的 `<td>` 元素，把它插入行中指定的位置。新表元将被插入当前位于 `index` 指定的位置的表元之前。如果 `index` 等于行中的表元数，新表元将被附加在行的末尾。

注意，这种方法只能插入 `<td>` 数据表元。如果需要给行添加头表元，必须用 `Document.createElement()` 方法和 `Node.insertBefore()` 方法（或相关的方法）创建并插入一个 `<th>` 元素。

HTMLTableSectionElement

1 级 DOM HTML

表的头、脚注和主体段

Node → Element → HTMLElement →
HTMLTableSectionElement

属性

readonly HTMLCollection rows

HTMLTableRowElement 对象的数组（HTMLCollection 对象），表示表中该段中的所有行。

除了上面列出的 `rows` 属性，该接口还定义了下表列出的属性，它们直接对应于基础元素的 HTML 性质。

属性	性质	描述
String align	align	表元在段中的默认水平对齐方式
String ch	char	表元在段中的对齐字符
String chOff	chOff	表元在段中的默认字符偏移量
String vAlign	vAlign	表元在段中的默认垂直对齐方式

方法

deleteRow()

删除段中指定位置的列。

insertRow()

在段的指定位置插入一个空行。

描述

该接口表示 HTML 表中的 `<tbody>`、`<thead>` 或 `<tfoot>` 段。

参阅

Type of: `HTMLTableElement.tFoot`、`HTMLTableElement.tHead`

HTMLTableSectionElement.deleteRow()

1 级 DOM HTML

删除表段的一行

摘要

```
void deleteRow(long index)
    throws DOMException;
```

参数

index

指定了行在段中的位置。

抛出

如果 *index* 小于 0 或者大于等于段中行数，该方法将抛出代码为 `INDEX_SIZE_ERR` 的 `DOMException` 异常。

描述

该方法将删除段中指定位置的行。注意，*index* 指定的是行在段中的位置，不是行在整个表中的位置。

参阅

`HTMLTableElement.deleteRow()`

HTMLTableSectionElement.insertRow()

1 级 DOM HTML

给表段添加新的空行

摘要

```
HTMLTableSectionElement insertRow(long index)
    throws DOMException;
```

参数

index

要插入的新行在段中的位置。

返回值

一个 `HTMLTableRowElement` 对象，表示新创建并插入的 `<tr>` 元素。

抛出

如果 `index` 小于 0 或者大于等于段中行数，该方法将抛出代码为 `INDEX_SIZE_ERR` 的 `DOMException` 异常。

描述

该方法将创建一个新的 `<tr>` 元素，并把它插入段中的指定位置。如果 `index` 等于段中的行数，新行将被附加到段的末尾。否则新行将被插入当前位于 `index` 指定的位置的行之前。注意，在该方法中，`index` 指定的是行在段中的位置，不是它在整个表中的位置。

参阅

`HTMLTableElement.insertRow()`

HTMLTextAreaElement

1 级 DOM HTML

HTML 表单中的 `<textarea>` 元素

Node → Element → HTMLElement →
HTMLTextAreaElement

属性

String `accessKey`

浏览器用于把键盘焦点转移到该元素的快捷键（一定是一个字符）。映射 HTML 性质 `accessKey`。

long `cols`

该元素的宽度，以字符列计。映射 HTML 性质 `cols`。

String `defaultValue`

文本框的初始内容。在表单被重置时，文本框将被恢复为这个值。改变这个属性的值会改变文本框当前显示的文本。

boolean `disable`

如果为 `true`，则该输入元素将被禁用，用户输入不能与它交互。映射 HTML 性质 `disabled`。

readonly HTMLFormElement form

HTMLFormElement对象，表示包含该文本框的 <form> 元素，如果该输入元素不在表单中，则为 null。

String name

<textarea> 元素的名字，由 HTML 性质 name 设置。

boolean readOnly

如果为 true，该元素是只读的，用户不能编辑其中显示的文本。映射 HTML 性质 readOnly。

long rows

文本框的高度，以文本行数计。映射 HTML 性质 rows。

long tabIndex

在跳格顺序中，该元素的位置。映射 HTML 性质 tabindex。

readonly String type

元素类型，以便和HTMLInput Element对象兼容。该属性的值总是“textarea”。

String value

文本框当前显示的文本。

方法

blur()

把键盘焦点从该元素上移开。

focus()

把键盘焦点转移到该元素。

select()

选中文本框的完整内容。

描述

该接口表示<textarea>元素，它用于在HTML表单中创建多行输入的文本框。该文本框的初始内容在标记<textarea>和</textarea>之间设置。用户可以用value属性（或通过修改该元素的Text子节点）编辑这个值，或查询和设置它的文本。

参阅

HTMLFormElement、HTMLInputElement; 客户端参考手册中的 Textarea 对象; 第十五章

HTMLTextAreaElement.blur()

1 级 DOM HTML

把键盘焦点从该元素中移开

摘要

```
void blur();
```

描述

该方法将把键盘焦点从该元素中移开。

HTMLTextAreaElement.focus()

1 级 DOM HTML

给予该元素键盘焦点

摘要

```
void focus();
```

描述

该方法将把键盘焦点转移到当前元素，使用户可以不必点击它，就能编辑显示的文本。

HTMLTextAreaElement.select()

1 级 DOM HTML

选择该元素的文本

摘要

```
void select();
```

描述

该方法将选中 <textarea> 元素显示的所有文本。在大多数浏览器中，这意味着文本将高亮显示，用户输入的新文本将替换高亮的文本，而不是附加到其后。

LinkStyle

2 级 DOM StyleSheet

与节点关联的样式表

属性

`readonly StyleSheet sheet`

与节点关联的 StyleSheet 对象。

描述

在支持 StyleSheet 模块的 DOM 实现中，该接口由链接到样式表或定义了内联样式表的 Document 节点实现。sheet 属性提供了获取与该节点关联的 StyleSheet 对象的方法。

在 HTML 文档中，<style> 元素和 <link> 元素实现了该接口。它们由 HTMLStyleElement 接口和 HTMLLinkElement 接口表示，在本参考手册中，它们没有自己的参考页。有关这些接口的详细信息请参阅“HTML Element”。

在 XML 文档中，样式表包括一个处理指令。参见“ProcessingInstruction”以获得详细信息。

参阅

HTML Element、ProcessingInstruction

MediaList

2 级 DOM StyleSheet

样式表的媒体类型列表

属性

`readonly unsigned long length`

数组的长度，即列表中的媒体类型数。

`String mediaText`

完整的媒体列表的文本表示，媒体类型之间用逗号分隔。设置该属性时，如果新值包含语法错误，将抛出代码为 SYNTAX_ERR 的 DOMException 异常，如果媒体列表是只读的，将抛出代码为 NO_MODIFICATION_ALLOWED_EXCEPTION 的 DOMException 异常。

方法

`appendMedium()`

在列表尾部添加新的媒体类型。

```
deleteMedium()
```

从列表中删除指定的媒体类型。

```
item()
```

返回列表中指定位置的媒体类型。如果 `index` 无效，则返回 `null`。在 JavaScript 中，还可以把 `MediaList` 对象作为数组处理，用数组符号 `[]` 索引它，而无需调用该方法。

描述

这个接口表示样式表的媒体类型的列表或数组。`length` 属性声明了列表中的元素数，用 `item()` 方法可以根据位置获取指定的媒体类型。可以用 `appendMedium()` 和 `deleteMedium()` 将条目附加到列表中或从列表中删除条目。JavaScript 允许将 `MediaList` 对象作为数组处理，可以用 `[]` 方括号代替调用 `item()` 方法。

HTML 4 标准定义了以下的媒体类型（它们区别大小写，必须以小写字母书写）：`screen`、`tty`、`tv`、`projection`、`handheld`、`print`、`braille`、`aural` 和 `all`。`screen` 类型与台式机或笔记本电脑上的浏览器显示的文档关系最密切。`print` 类型用于打印的文档的样式。

参阅

Type of: `StyleSheet.media`

MediaList.appendMedium()

2 级 DOM `StyleSheet`

给列表添加新的媒体类型

摘要

```
void appendMedium(String newMedium)
```

throws `DOMException`;

参数

newMedium

要添加的新媒体类型。关于有效媒体类型名的集合，请参阅“`MediaList`”参考页。

抛出

如果媒体列表是只读的，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的

DOMException 异常。如果指定的参数 *newMedium* 包含不合法的字符，该方法将抛出代码为 `INVALID_CHARACTER_ERR` 的异常。

描述

该方法将把指定的 *newMedium* 附加到 *MediaList* 的末尾。如果 *MediaList* 已经包含这种指定的类型，该方法将首先从它的当前位置删除这种类型，然后把它附加到列表的末尾。

MediaList.deleteMedium()

2 级 DOM StyleSheet

把指定的媒体类型从列表中删除

摘要

```
void deleteMedium(String oldMedium)
    throws DOMException;
```

参数

oldMedium

要从列表中删除的媒体类型的名字。关于有效媒体类型名的集合，请参阅“*MediaList*”参考页。

抛出

如果列表不包括指定的 *oldMedium* 媒体类型，该方法将抛出代码为 `NOT_FOUND_ERR` 的 *DOMException* 异常。如果媒体列表是只读的，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的异常。

描述

该方法将从 *MediaList* 中删除指定的媒体类型。如果列表不包括指定的类型，该方法将抛出异常。

MediaList.item()

2 级 DOM StyleSheet

索引媒体类型的数组

摘要

```
String item(unsigned long index);
```


参数

`index`

想获取的媒体类型在数组中的位置。

返回值

`MediaList`中位于指定位置的媒体类型（一个字符串）。如果 `index` 是负数，或者大于等于 `length` 属性的值，该方法将返回 `null`。注意，在 JavaScript 中，可以将 `MediaList` 对象作为数组处理，用数组符号 `[]` 索引它，而不用调用该方法。

MouseEvent

2 级 DOM Events

鼠标事件的详细情况

Event → UIEvent → MouseEvent

属性

readonly boolean `altKey`

在鼠标事件发生时，**Alt** 键是否被按下并保持住了。所有类型的鼠标事件都定义了该属性。

readonly unsigned short `button`

在 `mousedown`、`mouseup` 或 `click` 事件中，哪个鼠标键的状态改变了。0 表示左键，2 表示右键，1 表示鼠标的中间键。注意，只有当改变鼠标键的状态时，该属性才会被定义，例如，在 `mouseover` 事件中，它不能用于报告鼠标键是否被按下并保持住了。此外，该属性不是位图，不能报告是否多个鼠标键被按下并保持住了。

Netscape 6.0 和 6.01 用 1、2、3 代替该属性的值 0、1、2。Netscape 6.1 修正了这一点。

readonly long `clientX`、`clientY`

声明鼠标指针相对于“客户区”或浏览器窗口的 X 坐标和 Y 坐标。注意，这些坐标不考虑文档滚动，如果事件发生在窗口的最上边，`clientY` 的值就是 0，无论文档向下滚动了多远。所有类型的鼠标事件都定义了这两个属性。

readonly boolean `ctrlKey`

在鼠标事件发生时，**Ctrl** 键是否被按下并保持住了。所有类型的鼠标事件都定义了该属性。

readonly boolean metaKey

在鼠标事件发生时，**Meta**键是否被按下并保持住了。所有类型的鼠标事件都定义了该属性。

readonly EventTarget relatedTarget

引用与事件的target节点相关的节点。对于mouseover事件，它是鼠标移到目标节点前要离开的节点。对于mouseout事件，它是鼠标离开目标节点后要进入的节点。其他类型的鼠标事件没有定义relatedTarget属性。

readonly long screenX、screenY

鼠标指针相对于用户显示器的左上角的X坐标和Y坐标。所有类型的鼠标事件都定义了这两个属性。

readonly boolean shiftKey

在鼠标事件发生时，**Shift**键是否被按下并保持住了。所有类型的鼠标事件都定义了该属性。

方法

initMouseEvent()

初始化新创建的MouseEvent对象的属性。

描述

该接口定义了传递给click、mousedown、mousemove、mouseout、mouseover和mouseup事件的Event对象的类型。注意，除了这里列出的属性之外，该接口还继承了UIEvent和Event接口的属性。

参阅

Event、UIEvent；第十九章

MouseEvent.initMouseEvent() 1 级 DOM Events

初始化MouseEvent对象的属性

摘要

```
void initMouseEvent(String typeArg,  
                    boolean canBubbleArg,  
                    boolean cancelableArg,
```

```
AbstractView viewArg,  
long detailArg,  
long screenXArg,  
long screenYArg,  
long clientXArg,  
long clientYArg,  
boolean ctrlKeyArg,  
boolean altKeyArg,  
boolean shiftKeyArg,  
boolean metaKeyArg,  
unsigned short buttonArg,  
EventTarget relatedTargetArg);
```

参数

该方法的许多参数用于设置 `MouseEvent` 对象的属性的初始值，其中包括从 `Event` 和 `UIEvent` 接口继承的属性。因为每个参数的名字清楚地说明了它们要设置的值的属性，所以这里不再一一列出。

描述

该方法将初始化新创建的 `MouseEvent` 对象的各种属性。只有 `Document.createEvent()` 方法创建的 `MouseEvent` 对象可以调用该方法，而且必须在把 `MouseEvent` 对象传递给 `EventTarget.dispatchEvent()` 方法之前调用。

MutationEvent

2 级 DOM Events

文档变化的详细情况

Event → MutationEvent

常量

下面的常量表示属性 `attrChange` 的可以设置值：

```
unsigned short MODIFICATION = 1
```

被修改的 Attr 节点。

```
unsigned short ADDITION = 2
```

添加的 Attr 节点。

`unsigned short REMOVAL = 3`

被删除的 Attr 节点。

属性

`readonly unsigned short attrChange`

对于 DOMAttrModified 事件, 性质是如何改变的。“常量”部分定义了它可以用的三个值。

`readonly String attrName`

对于 DOMAttrModified 事件来说, 表示改变的性质的名字。

`readonly String newValue`

对于 DOMAttrModified 事件, 表示 Attr 节点的新值。对于 DOMCharacterDataModified 事件来说, 表示 Text、Comment、CDATASection 或 ProcessingInstruction 节点的新文本值。

`readonly String prevValue`

对于 DOMAttrModified 事件, 是 Attr 节点的前一个值。对于 DOMCharacterDataModified 事件, 是 Text、Comment、CDATASection 或 ProcessingInstruction 节点的前一个值。

`readonly Node relatedNode`

对于 DOMAttrModified 事件, 是相关的 Attr 节点。对于 DOMNodeInserted 和 DOMNodeRemoved 事件, 是被插入或删除的节点的父节点。

方法

`initMutationEvent()`

初始化新创建的 MutationEvent 对象的属性。

描述

该接口定义了传递给下面列出的事件类型的 Event 对象的类型 (注意, 用 Event.preventDefault() 方法不能取消这些事件的默认动作):

DOMAttrModified

当对文档元素的属性进行添加、删除或改变操作时, 生成该对象。事件的目标是包含该性质的元素, 事件从此处开始起泡。

DOMCharacterDataModified

当 Text、Comment、CDATASection 或 ProcessingInstruction 节点的字符数据改变时，生成该对象。事件的目标是改变的节点，事件从此处开始起泡。

DOMNodeInserted

当一个节点作为另一个节点的子节点添加时，生成该对象。事件的目标是被插入的节点，事件从此处开始沿着文档树向上起泡。relatedNode 属性指定了被插入的节点的新父节点。被插入的节点的子孙节点不会生成这种类型的事件。

DOMNodeInsertedIntoDocument

在节点插入文档树后生成的对象，直接插入文档树和由于祖先节点被插入文档树而间接插入文档树的节点都会生成该对象。该事件的目标节点是被插入的节点。因为子树中的每个节点都有可能引发这种类型的事件，所以它们不会起泡。

DOMNodeRemoved

从一个节点的父节点中删除该节点之前生成的对象。该事件的目标节点是被删除的节点，事件从此处开始沿着文档树向上起泡。relatedNode 属性存放了被删除的节点的父节点。

DOMNodeRemovedFromDocument

从文档树中删除一个节点之前生成的对象。直接被删除的节点和由于祖先节点被删除而间接删除的节点都会生成该事件。该事件的目标是要删除的事件。这种类型的事件不会起泡。

DOMSubtreeModified

当调用一个 DOM 方法触发多个变化事件时生成的总结型事件。该事件的目标节点是文档中发生改变的所有嵌套最深的祖先节点，它从该节点开始沿着文档树向上起泡。如果你对改变的详细情况不感兴趣，只想知道文档的哪个部分改变了，可以为这种类型的事件注册监听器。

MutationEvent.initMutationEvent()

2 级 DOM Events

初始化新 MutationEvent 对象的属性

摘要

```
void initMutationEvent(String typeArg,  
                        boolean canBubbleArg,  
                        boolean cancelableArg,
```

```
Node relatedNodeArg,  
String prevValueArg,  
String newValueArg,  
String attrNameArg,  
unsigned short attrChangeArg);
```

参数

该方法的许多参数用于设置 `MutationEvent` 对象的属性的初始值，其中包括从 `Event` 接口继承的属性。因为每个参数的名字清楚地说明了它们要设置的属性，所以这里不再一一列出。

描述

该方法将初始化新创建的 `MutationEvent` 对象的各个属性。只有用 `Document.createEvent()` 方法创建的 `MutationEvent` 对象可以调用该方法，而且必须在把 `MouseEvent` 对象传递给 `EventTarget.dispatchEvent()` 方法之前调用。

NamedNodeMap

1 级核心 DOM

根据名字和位置索引的节点的集合

属性

`readonly unsigned long length`
数组中的节点数。

方法

`getNamedItem()`
查找指定的节点。

`getNamedItemNS()` [2 级 DOM]
查找用名字空间和名字指定的节点。

`item()`
获取 `NamedNodeMap` 中指定位置的节点。在 JavaScript 中，可以用节点的位置作为数组下标来实现这一点。

`removeNamedItem()`
从 `NamedNodeMap` 中删除指定的节点。

`removeNamedItemNS()` [2 级 DOM]

根据名字和名字空间从 `NamedNodeMap` 中删除指定的节点。

`setNamedItem()`

给 `NamedNodeMap` 添加新节点（或替换现有的节点）。Node 对象的 `nodeName` 属性将作为该节点的名字。

`setNamedItemNS()` [2 级 DOM]

给 `NamedNodeMap` 添加新节点（或替换现有的节点）。Node 对象的 `namespaceURI` 属性和 `localName` 属性将作为该节点的名字。

描述

`NamedNodeMap` 接口定义了一个能根据 `nodeName` 属性或 `namespaceURI` 属性和 `localName` 属性（对于使用名字空间的节点）查找的节点的集合。

`NamedNodeMap` 接口定义的最有用的属性是 Node 接口的 `attributes`，它是可以根据性质名查找的 Attr 节点的集合。`NamedNodeMap` 接口的许多方法与用于操作特性的 `Element` 方法相似。通过 `Element` 接口的方法操作 `Element` 性质最容易，通常不使用 `NamedNodeMap` 接口的方法。

`NamedNodeMap` 对象是“活”的，即它们会即刻反映出文档树的变化。例如，如果获取了一个表示元素性质的 `NamedNodeMap` 对象，然后给那个元素添加一个新性质，那么通过 `NamedNodeMap` 就可以访问这个新性质。

参阅

`NodeList`

Type of: `DocumentType.entities`、`DocumentType.notations`、`Node.attributes`

`NamedNodeMap.getNamedItem()`

1 级核心 DOM

根据名字查找节点

摘要

```
Node getNamedItem(String name);
```

参数

name

要查找的节点的 `nodeName` 属性值。

返回值

指定的节点，如果没有找到具有指定名字的节点，则返回 `null`。

NamedNodeMap.getNamedItemNS()

2 级核心 DOM

根据名字和名字空间查找节点

摘要

```
Node getNamedItemNS(String namespaceURI,  
                     String localName);
```

参数

namespaceURI

想获取的节点的 `namespaceURI` 属性，如果没有名字空间，该属性为 `null`。

localName

本地节点的 `localName` 属性。

返回值

`NamedNodeMap` 中具有指定的 `namespaceURI` 属性和 `localName` 属性的元素。如果没有这样的节点，则返回 `null`。

描述

该方法将根据名字空间和本地名查找 `NamedNodeMap` 中的一个元素。只有使用名字空间的 XML 文档才可以使用该方法。

NamedNodeMap.item()

1 级核心 DOM

根据位置返回 `NamedNodeMap` 中的一个元素

摘要

```
Node item(unsigned long index);
```

参数

index

想获取的节点的位置或下标。

返回值

指定位置的节点，如果 *index* 小于 0 或者大于等于 `NamedNodeMap` 的 `length` 属性，则返回 `null`。

描述

该方法将返回 `NamedNodeMap` 的一个编码元素。在 JavaScript 中，`NamedNodeMap` 对象的行为与只读数组相似，可以用节点的位置作为数组下标来访问它的元素，而不必调用该方法。

虽然用 `NamedNodeMap` 接口可以根据位置遍历它的节点，但它不表示节点的有序集合。对 `NamedNodeMap` 对象的任何改变（如调用 `removeNamedItem()` 方法或 `setNamedItem()` 方法）都会使所有的元素重新排序。因此，在遍历 `NamedNodeMap` 的元素时，不能修改它。

参阅

`NodeList`

`NamedNodeMap.removeNamedItem()`

1 级核心 DOM

根据名字删除指定的节点

摘要

```
Node removeNamedItem(String name)  
    throws DOMException;
```

参数

name

要删除的节点的 `nodeName` 属性。

返回值

被删除的节点。

抛出

如果 `NamedNodeMap` 是只读的，不允许删除操作，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。如果 `NamedNodeMap` 中不存在 *name* 指定的节点，则抛出代码为 `NOT_FOUND_ERR` 的异常。

描述

从NamedNodeMap中删除指定的节点。注意, 如果NamedNodeMap表示一个Element性质集合, 那么删除在文档中明确设置的性质的Attr节点, 会使删除的Attr节点自动替换为新的Attr节点, 该节点表示性质的默认值(如果存在的话)。

参阅

Element.removeAttribute()

NamedNodeMap.removeNamedItemNS() 2 级核心 DOM

删除由名字空间和名字指定的节点

摘要

```
Node removeNamedItemNS(String namespaceURI,  
                        String localName)  
    throws DOMException;
```

参数

namespaceURI

要删除的节点的 namespaceURI 属性, 如果没有名字空间, 则为 null。

localName

要删除的节点的 localName 属性。

返回值

被删除的节点。

抛出

该方法抛出异常的原因和 removeNamedItem() 方法一样。

描述

该方法与 removeNamedItem() 方法相似, 只是要删除的节点由名字空间和本地名指定的, 而不是仅仅由名字指定的。只有使用名字空间的 XML 文档才可以使用该方法。

NamedNodeMap.setNamedItem()1 级核心 DOM

在 NamedNodeMap 中添加或替换一个节点

摘要

```
Node setNamedItem(Node arg)
    throws DOMException;
```

参数

arg

要添加到 NamedNodeMap 的节点。

返回值

被替换的节点，如果没有节点被替换，则返回 `null`。

抛出

该方法将抛出具有下列代码的 DOMException 异常：

HIERARCHY_REQUEST_ERR

参数 *arg* 指定的节点类型不适合该 NamedNodeMap 节点（如不是 Attr 节点）。

INUSE_ATTRIBUTE_ERR

参数 *arg* 是已经关联到一个元素的 Attr 节点。

NO_MODIFICATION_ALLOWED_ERR

NamedNodeMap 是只读的。

WRONG_DOCUMENT_ERR

参数 *arg* 的 `ownerDocument` 属性不同于创建 NamedNodeMap 节点的文档。

描述

该方法将给 NamedNodeMap 添加指定的节点，并允许用节点的 `nodeName` 属性的值查找它。如果 NamedNodeMap 已经包含了具有该名字的节点，则该节点将被替换，并成为该方法的返回值。

参阅

`Element.setAttribute()`

NamedNodeMap.setNamedItemNS()

2 级核心 DOM

用名字空间给 NamedNodeMap 添加一个节点

摘要

```
Node setNamedItemNS(Node arg)
    throws DOMException;
```

参数

arg

要添加到 NamedNodeMap 的节点。

返回值

被替换的节点，如果没有节点被替换，则返回 null。

抛出

该方法抛出异常的原因和 setNamedItem() 方法一样。此外，如果调用它的实现不支持 XML 文档或 XML 名字空间，它还会抛出代码为 NOT_SUPPORTED_ERR 的 DOMException 异常。

描述

该方法与 setNamedItem() 方法相似，只是可以用 namespaceURI 属性和 localName 属性查找添加到 NamedNodeMap 的节点，而不是用 nodeName 属性。只有使用名字空间的 XML 文档才会使用该方法。注意，在不支持 XML 文档的实现中，可能不支持该方法（如它会抛出异常）。

Node

1 级核心 DOM

文档树中的一个节点

子接口

Attr、CharacterData、Document、DocumentFragment、DocumentType、Element、Entity、EntityReference、Notation、ProcessingInstruction

也实现

EventTarget 如果 DOM 实现支持 Events 模块，那么文档树中的每个节点都会实现 EventTarget 接口，而且还在它上面注册了事件监听器。这里不包括 EventTarget 接口定义的方法，详见“EventTarget”参考页和“EventListener”参考页。

常量

每个 Node 对象都实现了上面列出的一种子接口。每个 Node 对象都有 `nodeType` 属性，声明了它们实现的是哪种子接口。下面的常量是该属性的合法值，它们的名称的含义很明显，不需要解释。注意，它们是 `Node()` 构造函数的静态属性，不是个别 Node 对象属性。此外还要注意，Internet Explorer 4、5、6 不支持这些常量。

```
Node.ELEMENT_NODE = 1;           // Element
Node.ATTRIBUTE_NODE = 2;         // Attr
Node.TEXT_NODE = 3;              // Text
Node.CDATA_SECTION_NODE = 4;     // CDATASection
Node.ENTITY_REFERENCE_NODE = 5;   // EntityReference
Node.ENTITY_NODE = 6;             // Entity
Node.PROCESSING_INSTRUCTION_NODE = 7; // ProcessingInstruction
Node.COMMENT_NODE = 8;           // Comment
Node.DOCUMENT_NODE = 9;          // Document
Node.DOCUMENT_TYPE_NODE = 10;    // DocumentType
Node.DOCUMENT_FRAGMENT_NODE = 11; // DocumentFragment
Node.NOTATION_NODE = 12;         // Notation
```

属性

readonly NamedNodeMap attributes

如果该节点是一个元素，这个属性声明了该元素的性质。attributes 是一个 NamedNodeMap 对象，根据名字或编码可以查询性质，并以 Attr 对象的形式返回该属性。实际上，用 Element 接口的 `getAttribute()` 方法更容易以字符串形式获取性质的值。注意，返回的 NamedNodeMap 对象是“活”的，元素性质发生任何改变，通过它都可以立刻看到。

readonly Node[] childNodes

存放当前节点的子节点。该属性绝不会为 null，因为对于没有子节点的节点来说，childNodes 是一个 length 为 0 的数组。从技术上说来，该属性是一个 NodeList 对象，但它的行为更像 Node 对象的数组。注意，返回的 NodeList 对象是“活”的，子元素属性发生任何改变，通过它都可以立刻看到。

readonly Node firstChild

当前节点的第一个子节点，如果当前节点没有子节点，则为 null。

readonly Node lastChild

当前节点的最后一个子节点，如果当前节点没有子节点，则为 null。

readonly String localName [2 级 DOM]

在使用名字空间的 XML 文档中，它声明了该元素的本地部分或性质名。HTML 文档从不使用该属性。参阅 namespaceURI 属性和 prefix 属性。

`readonly String namespaceURI` [2 级 DOM]

在使用名字空间的XML文档中, 声明一个Element节点或Attribute节点的名字空间的URI。HTML文档从不使用该属性。参阅 `localName` 属性和 `prefix` 属性。

`readonly Node nextSibling`

在 `parentNode` 的 `childNodes[]` 数组中, 紧接着当前节点的兄弟节点。如果没有这样的节点, 则为 `null`。

`readonly String nodeName`

节点的名字。对于Element节点, 该属性声明了该元素的标记名, 用Element接口的 `tagName` 属性也可以获取它。对于其他类型的节点, 它的值由节点类型决定。参阅“描述”部分中的表。

`readonly unsigned short nodeType`

节点的类型, 如该节点实现了哪种子接口。前面列出的常量定义了它的合法值。但由于Internet Explorer不支持那些常量, 所以可以用硬编码的值来代替这些常量。在HTML文档中, 该属性的对于Element节点的值是1, 对Text节点的该属性值是3, 对Comment节点该属性的值是8, 对顶层Document节点该属性的值是9。

`String nodeValue`

节点的值。对于Text节点来说, 该属性存放的是文本的内容。对于其他类型的节点, 该属性的值由 `nodeType` 决定。参阅下面的表。

`readonly Document ownerDocument`

该节点所属的Document对象。对于Document节点来说, 该属性为 `null`。

`readonly Node parentNode`

当前节点的父节点(或包容节点), 如果没有父节点, 则为 `null`。注意, Document节点和Attr节点绝不会有父节点。另外, 从文档中删除的节点或新创建的、还没有插入文档树的节点, `parentNode` 属性都为 `null`。

`String prefix` [2 级 DOM]

对于使用名字空间的XML文档, 该属性声明了Element节点或Attribute节点的名字空间的前缀。HTML文档从不使用该属性。参阅 `localName` 属性和 `namespaceURL` 属性。设置该属性时, 如果新值包含不合法的字符、格式错误, 或者与 `namespaceURI` 属性不匹配, 都会引发异常。

readonly Node previousSibling

在parentNode的childNodes[]数组中，紧挨着当前节点、位于它之前的兄弟节点。如果没有这样的节点，则为null。

方法

appendChild()

通过把一个节点附加到当前节点的childNodes[]数组，给文档树添加节点。如果文档树中已经存在了该节点，则把它删除，然后在新位置插入它。

cloneNode()

复制当前节点，或复制当前节点及它的所有子孙节点。

hasAttributes() [2 级 DOM]

如果当前节点是Element节点，而且具有性质，则返回true。

hasChildNodes()

如果当前节点具有子节点，就返回true。

insertBefore()

在文档树中插入一个节点，插入到当前节点的指定子节点之前。如果文档树中已经存在要插入的节点，则删除该节点，重新插入到它的新位置。

isSupported() [2 级 DOM]

如果当前节点支持指定特性的特定版本，则返回true。

normalize()

通过删除当前节点的所有空Text节点，合并相邻的Text节点，“规范化”它的所有Text子孙节点。

removeChild()

从文档树中删除（并返回）指定的子节点。

replaceChild()

从文档树中删除（并返回）指定的子节点，用另一个节点替换它。

描述

文档树中的所有对象（包括Document对象自身）都实现了Node接口，它提供了遍历和操作文档树的基本属性和方法。用parentNode属性和childNodes[]数组可以在文档树中上下移动。通过遍历childNodes[]数组或使用firstChild和next-

Sibling属性进行循环操作(或用lastChild和previousSibling属性反向循环),可以枚举指定节点的子节点。调用appendChild()、insertBefore()、removeChild()和replaceChild()方法,可以通过改变一个节点的子节点修改文档树。

文档树中的每个对象都实现了Node接口和更专用的接口,如Element接口或Text接口。nodeType属性声明了该节点实现的是哪个子接口。在使用专用接口的方法和属性前,可以用这个属性检测节点的类型。例如:

```
var n; // 用来存储使用的节点
if (n.nodeType == 1) { // 或与常量Node.ELEMENT_NODE比较
    var tagName = n.tagName; // 如果节点为Element,则为标记名
}
```

属性nodeName和nodeValue声明了更多的节点信息,但它们的值由nodeType决定,如下表所示。注意,子接口通常都定义了专门的属性(如Element节点的tagName属性和Text节点的数据属性)来获取该信息。

nodeType	nodeName	nodeValue
ELEMENT_NODE	元素标记名	null
ATTRIBUTE_NODE	性质名	性质值
TEXT_NODE	#text	节点文本
CDATA_SECTION_NODE	#cdata-section	节点文本
ENTITY_REFERENCE_NODE	引用实体名	null
ENTITY_NODE	实体名	null
PROCESSING_INSTRUCTION_NODE	PI 目标	PI 的剩余部分
COMMENT_NODE	#comment	注释的文本
DOCUMENT_NODE	#document	null
DOCUMENT_TYPE_NODE	文档类型名	null
DOCUMENT_FRAGMENT_NODE	#document-fragment	null
NOTATION_NODE	符号名	null

参阅

Document、Element、Text; 第十七章

Node.appendChild() 1 级核心 DOM

插入一个节点,作为当前节点的最后一个子节点



摘要

```
Node appendChild(Node newChild)  
    throws DOMException;
```

参数

newChild

要插入文档的节点。如果该节点是 `DocumentFragment` 节点，则不会直接插入它，而是插入它的子节点。

返回值

加入的节点。

抛出

该方法将在下列环境中抛出如下代码的 `DOMException` 异常：

`HIERARCHY_REQUEST_ERR`

当前节点不能有子节点，或者它不能有指定类型的子节点，或者 *newChild* 是该节点的祖先（或是该节点自身）。

`WRONG_DOCUMENT_ERR`

newChild 的 `ownerDocument` 属性与当前节点的 `ownerDocument` 属性不同。

`NO_MODIFICATION_ALLOWED_ERR`

当前节点是只读的，不允许添加子节点，或者被添加的节点已经是文档树的一部分，它的父节点是只读的，不允许删除子节点。

描述

该方法将把节点 *newChild* 添加到文档中，使它成为当前节点的最后一个子节点。如果文档树中已经存在了 *newChild*，它将从文档树中删除，然后重新插入它的新位置。如果 *newChild* 是 `DocumentFragment` 节点，则不会直接插入它，而是把它的子节点按序插入当前节点的 `childNodes[]` 数组的末尾。注意，来自一个文档的节点（或由一个文档创建的节点）不能插入另一个文档。也就是说，*newChild* 的 `ownerDocument` 属性必须与当前节点的 `ownerDocument` 属性相同。

例子

接下来的函数将在文档末尾插入一个新段：

```
function appendMessage(message) {  
    var pElement = document.createElement("P");  
    var messageNode = document.createTextNode(message);  
    pElement.appendChild(messageNode);    // 向段中添加文本  
    document.body.appendChild(pElement); // 向文档主体添加文本  
    body  
}
```

参阅

Node.insertBefore(), Node.removeChild(), Node.replaceChild()

Node.cloneNode()

1 级核心 DOM

复制节点和它的所有子孙节点（可选）

摘要

Node cloneNode(boolean deep);

参数

deep

如果该参数为 true, cloneNode() 就会递归复制当前节点的所有子孙节点。否则, 它只复制节点自身。

返回值

当前节点的副本。

描述

该方法将复制并返回调用它的节点的副本。如果传递给它的参数是 true, 它还将递归复制当前节点的所有子孙节点。否则, 它只复制当前节点。返回的节点不属于文档树, 它的 parentNode 属性为 null。当复制的是 Element 节点时, 它的所有性质都将被复制。但要注意, 当前节点上注册的 EventListener 函数不会被复制。

Node.hasAttributes()

2 级核心 DOM

判断当前节点是否具有性质

摘要

boolean hasAttributes();

返回值

如果当前节点具有一个或多个性质，则返回 `true`，否则返回 `false`。注意，只有 `Element` 节点可以具有性质。

参阅

`Element.getAttribute()`、`Element.hasAttributes()`、`Node.attributes`

Node.hasChildNodes()

1 级核心 DOM

判断当前节点是否具有子节点

摘要

```
boolean hasChildNodes();
```

返回值

如果当前节点具有一个或多个子节点，则返回 `true`，否则返回 `false`。

参阅

`Node.childNodes`

Node.insertBefore()

1 级核心 DOM

在文档树中的指定节点前插入一个节点

摘要

```
Node insertBefore(Node newChild,  
                  Node refChild)  
    throws DOMException;
```

参数

newChild

要插入文档树的节点。如果它是 `DocumentFragment` 节点，则插入它的子节点。

refChild

位于要插入的 *newChild* 之前的子节点。如果该参数为 `null`，则把 *newChild* 节点插入到当前节点的最后一个子节点。

返回值

被插入的节点。

抛出

该方法将在下列环境中抛出如下代码的 DOMException 异常：

HIERARCHY_REQUEST_ERR

当前节点不支持子节点，或者它不能有指定类型的子节点，或者 *newChild* 是该节点的祖先节点（或是该节点自身）。

WRONG_DOCUMENT_ERR

newChild 的 *ownerDocument* 属性与当前节点的 *ownerDocument* 属性不同。

NO_MODIFICATION_ALLOWED_ERR

当前节点是只读的，不允许添加子节点，或者 *newChild* 的父节点是只读的，不允许删除子节点。

NOT_FOUND_ERR

refChild 不是当前节点的子节点。

描述

该方法将把节点 *newChild* 作为当前节点的子节点插入文档树。新节点将存放在当前节点的 *childNodes[]* 数组中，紧接在 *refChild* 节点前。如果 *refChild* 为 *null*，*newChild* 将插入到 *childNodes[]* 数组的末尾，如 *appendChild()* 方法一样。注意，如果 *refChild* 不是当前节点的子节点，调用该方法是不合法的。

如果文档树中已经存在 *newChild* 节点，那么它将从文档树中删除，然后重新插入到它的新位置。如果 *newChild* 是 *DocumentFragment* 节点，那么插入文档树的不是它自身，而是它的子节点，这些子节点将按顺序插在指定的位置上。

例子

下面的函数将把一个新段插入到文档的开头：

```
function insertMessage(message) {  
    var paragraph = document.createElement("p"); // 创建一个 <p> Element  
    var text = document.createTextNode(message); // 创建一个 Text 节点  
    paragraph.appendChild(text); // 向文本添加段落  
    // 现在在文本的第一个子元素之前插入段落  
    document.body.insertBefore(paragraph, document.body.firstChild)  
}
```

参阅

`Node.appendChild()`、`Node.removeChild()`、`Node.replaceChild()`

Node.isSupported()

2 级核心 DOM

判断当前节点是否支持某个特性

摘要

```
boolean isSupported(String feature,  
                     String version);
```

参数

feature

要检测的特性的名字。

version

要检测的特性的版本号，如果要检测对该特性的版本的支持，则为空串。

返回值

如果当前节点支持指定特性的指定版本，则返回 `true`，否则返回 `false`。

描述

W3C DOM 标准被模块化，它的实现不必实现标准规定的所有模块或特性。该方法用于检测当前节点的实现是否支持指定特性的指定版本。关于参数 *feature* 和 *version* 的值的列表，请参阅“`DOMImplementation.hasFeature()`”参考页。

参阅

`DOMImplementation.hasFeature()`

Node.normalize()

1 级核心 DOM

合并相邻的 `Text` 节点并删除空的 `Text` 节点

摘要

```
void normalize();
```

描述

该方法将遍历当前节点的所有子孙节点，通过删除空的 `Text` 节点，以及合并所有相邻

的Text节点来规范化文档。该方法在进行节点的插入或删除操作后,对于简化文档树的结构很有用。

参阅

Text

Node.removeChild()

1 级核心 DOM

删除 (并返回) 当前节点的指定子节点

摘要

```
Node removeChild(Node oldChild)  
    throws DOMException;
```

参数

oldChild
要删除的子节点。

返回值

被删除的节点。

抛出

该方法将在下列环境中抛出如下代码的 DOMException 异常:

NO_MODIFICATION_ALLOWED_ERR
当前节点是只读的, 不允许删除子节点。

NOT_FOUND_ERR
oldChild 不是当前节点的子节点。

描述

该方法将从当前节点的 *childNodes[]* 数组中删除指定的子节点。如果调用该方法时传递的节点不是当前节点的子节点, 将引发错误。该方法在删除 *oldChild* 后, 将返回它。*oldChild* 仍然是有效节点, 可以再次插入文档。

例子

可以用下面的代码删除文档主体的最后一个子节点:

```
document.body.removeChild(document.body.lastChild);
```

参阅

`Node.appendChild()`、`Node.insertBefore()`、`Node.replaceChild()`

`Node.replaceChild()`

1 级核心 DOM

用新节点替换一个子节点

摘要

```
Node replaceChild(Node newChild,  
                  Node oldChild)  
    throws DOMException;
```

参数

`newChild`

新的替换节点。

`oldChild`

要被替换的节点。

返回值

从文档树中删除和被替换的节点。

抛出

该方法将在下列环境中抛出如下代码的 `DOMException` 异常：

`HIERARCHY_REQUEST_ERR`

当前节点不能有子节点、或者它不能有指定类型的子节点，也可能 `newChild` 是该节点的祖先节点（或是该节点自身）。

`WRONG_DOCUMENT_ERR`

`newChild` 的 `ownerDocument` 属性与当前节点的 `ownerDocument` 属性不同。

`NO_MODIFICATION_ALLOWED_ERR`

当前节点是只读的，不允许替换子节点，或者 `newChild` 是子节点，它的父节点不允许删除子节点。

`NOT_FOUND_ERR`

`oldChild` 不是当前节点的子节点。

描述

该方法将用一个节点替换文档树的节点。`oldChild`是被替换的节点，它必须是当前节点の子节点。`newChild`将在当前节点的 `childNodes[]` 数组中代替 `oldChild`。

如果文档树中已经存在 `newChild` 节点，那么它将被从文档树中删除，然后重新插入它的新位置。如果 `newChild` 是 `DocumentFragment` 节点，那么插入文档树的不是它自身，而是它的子节点，按顺序插在原来由 `oldChild` 占有的位置上。

例子

下面的代码将用元素 `` 替换节点 `n`，然后把替换掉的节点插入 `` 元素，这种操作是重定节点 `n` 的父节点，使它以粗体字显示：

```
// 创建文档第一段的第一个子节点
var n = document.getElementsByTagName("p")[0].firstChild;
var b = document.createElement("b"); // 创建 <b> 元素
n.parentNode.replaceChild(b, n);      // 用 <b> 替换节点
b.appendChild(n);                      // 作为 <b> 的子节点重新插入节点
```

参阅

`Node.appendChild()`、`Node.insertBefore()`、`Node.removeChild()`

NodeFilter

2 级 DOM Traversal

过滤文档树的节点的函数

常量

下面的三个常量是节点过滤函数的合法返回值。注意，它们是名为 `NodeFilter` 的对象的静态属性，不是个别节点的过滤函数的属性：

```
short FILTER_ACCEPT = 1
```

接受当前节点。`NodeIterator` 或 `TreeWalker` 接口将返回当前节点，作为它的文档遍历的一部分。

```
short FILTER_REJECT = 2
```

拒绝当前节点。`NodeIterator` 或 `TreeWalker` 接口的行为像当前节点不存在一样。另外，该返回值还通知 `TreeWalker` 接口忽略当前节点的所有子节点。

```
short FILTER_SKIP = 3
```

跳过当前节点。`NodeIterator` 或 `TreeWalker` 接口不返回当前节点，但它仍然会递归遍历它的子节点。

接下来的常量是位标志, 用于设置 Document 对象的 `createNodeIterator()` 方法和 `createTreeWalker()` 方法的参数 `whatToShow`。每个常量对应一种 Document 节点的类型(关于节点类型的列表, 参阅“Node”参考页), 指定了 `NodeIterator` 或 `TreeWalker` 接口在遍历文档的过程中应该考虑那种类型的节点。用逻辑 OR 运算符 “|” 可以把多个常量组合起来。`SHOW_ALL` 是个特殊值, 它所有的位都被设置了, 以说明需要考虑所有类型的节点(不管其类型如何)。

```
unsigned long SHOW_ALL = 0xFFFFFFFF;
unsigned long SHOW_ELEMENT = 0x00000001;
unsigned long SHOW_ATTRIBUTE = 0x00000002;
unsigned long SHOW_TEXT = 0x00000004;
unsigned long SHOW_CDATA_SECTION = 0x00000008;
unsigned long SHOW_ENTITY_REFERENCE = 0x00000010;
unsigned long SHOW_ENTITY = 0x00000020;
unsigned long SHOW_PROCESSING_INSTRUCTION = 0x00000040;
unsigned long SHOW_COMMENT = 0x00000080;
unsigned long SHOW_DOCUMENT = 0x00000100;
unsigned long SHOW_DOCUMENT_TYPE = 0x00000200;
unsigned long SHOW_DOCUMENT_FRAGMENT = 0x00000400;
unsigned long SHOW_NOTATION = 0x00000800;
```

方法

`acceptNode()`

在 Java 这样的语言中, 不允许把函数作为参数传递给其他函数, 所以定义节点过滤器时, 需要定义一个类, 由该类实现这种接口, 并包含对这种函数的实现。该函数的参数是一个节点, 必须返回 `FILTER_ACCEPT`、`FILTER_REJECT` 和 `FILTER_SKIP` 这三个常量中的一个。但在 JavaScript 中, 只需要定义一个函数(可以是任何名字), 它的参数是一个节点, 返回三个过滤常量之一, 即可创建一个节点过滤器。详见下面的描述和示例。

描述

节点过滤器是一个对象, 可以检测一个 Document 节点, 通知 `NodeIterator` 或 `TreeWalker` 接口是否在文档遍历中包括这个节点。在 JavaScript 中, 节点过滤器只是一个函数, 唯一的参数是一个节点, 返回前面定义的两个过滤常量之一。注意, 不存在 `NodeFilter` 接口, 只有名为 `NodeFilter` 的对象, 具有定义了过滤常量的属性。要使用节点过滤器, 需要把它传递给 Document 对象的 `createNodeIterator()` 方法或 `createTreeWalker()` 方法。在使用生成的 `NodeIterator` 或 `TreeWalker` 对象遍历文档时, 节点过滤函数将被调用以便检测节点。

理论上应该编写节点过滤函数，这样它们自身才不会改变文档树，也不抛出异常。另外，节点过滤器不会根据以往的过滤器调用历史作出过滤决定。

例子

可以用如下方法定义并使用一个节点过滤函数：

```
// 定义一个节点过滤器用来过滤除 <n1> 和 <n2> 外的所有元素
var myfilter = function(n) { // 过滤节点 n
    if ((n.nodeName == "H1") || (n.nodeName == "H2"))
        return NodeFilter.FILTER_ACCEPT;
    else
        return NodeFilter.FILTER_SKIP;
}
// 现在创建一个使用过滤器的 NodeIterator
var ni = document.createNodeIterator(document.body, // 遍历文档主体
                                     NodeFilter.SHOW_ELEMENT, // 只有元素
                                     myfilter, // 通过标记名过滤
                                     false); // 没有实体扩展
```

参阅

NodeIterator、TreeWalker

Type of: NodeIterator.filter、TreeWalker.filter

Passed to: Document.createNodeIterator()、Document.createTreeWalker()

NodeIterator

2 级 DOM Traversal

遍历 Document 节点过滤后的序列

属性

readonly boolean expandEntityReferences

当前的 NodeIterator 对象是否遍历 EntityReference 节点的子节点（在 XML 文档中）。该属性的值在首次创建 NodeIterator 时，将作为 Document.createNodeIterator() 方法的参数。

readonly NodeFilter filter

节点过滤函数，在调用 Document.createNodeIterator() 方法时指定。

readonly Node root

NodeIterator 对象开始遍历的根节点。该属性的值在调用 Document.createNodeIterator() 方法时指定。

readonly unsigned long whatToShow

位标志的集合（有效标志的列表参阅“NodeFilter”参考页），指定了当前 NodeIterator 对象考虑的 Document 节点的类型。如果该属性中的某个位没有被设置，那么 NodeIterator 对象将忽略相应的节点类型。注意，该属性的值是在调用 Document.createNodeIterator() 方法时指定的。

方法

detach()

把当前的 NodeIterator 对象从它的文档中“分离”出来，以便在文档被修改时，实现不必再修改 NodeIterator 对象。在使用完 NodeIterator 对象后，才调用该方法。调用 detach() 后，对 NodeIterator 对象其他方法的调用都会引发异常。

nextNode()

返回 NodeIterator 表示的过滤后的节点序列中的下一个节点，如果 NodeIterator 已经返回了最后一个节点，该方法则返回 null。

previousNode()

返回 NodeIterator 表示的过滤后的节点序列中的前一个节点，如果没有前一个节点，该方法则返回 null。

描述

NodeIterator 接口表示 Document 节点的序列，这些节点是按照文档源代码的顺序遍历文档子树，并用两级过程过滤节点后得到的。NodeIterator 对象由 Document.createNodeIterator() 方法创建。用 nextNode() 和 previousNode() 方法可以前后遍历这个节点序列。在使用 NodeIterator 对象后，才能调用 detach() 方法，除非你确定在文档修改前，这个 NodeIterator 对象将被作为无用存储单元回收。注意，这个接口的属性都是传递给 Document.createNodeIterator() 方法的参数的只读副本。

nextNode() 和 previousNode() 方法返回的节点必须经过两步过滤。首先，节点类型必须是 whatToShow 属性指定的类型之一。可以组合起来指定 Document.createNodeIterator() 方法的 whatToShow 属性的常量列表，请参阅“NodeFilter”参考页。接下来，如果 filter 属性不为 null，那么进行 whatToShow 检测的节点还要传递给 filter 属性指定的过滤函数。如果该函数返回 NodeFilter.FILTER_ACCEPT，该节点将被返回。如果它返回 NodeFilter.FILTER_REJECT 或 NodeFilter.FILTER_SKIP，NodeIterator 将跳过该节点。注意，当节点被一个或两个过滤步骤拒绝时，只有节点自身被拒绝，它的子节点不会被自动拒绝，仍然要接受同样的过滤步骤。

即使 `NodeIterator` 遍历的文档树被修改了, `NodeIterator` 对象仍然有效。`nextNode()` 和 `previousNode()` 方法将基于文档的当前状态返回节点, 而不是创建 `NodeIterator` 对象时的文档状态。

参阅

`NodeFilter`, `TreeWalker`; 第十七章

Returned by: `Document.createNodeIterator()`

`NodeIterator.detach()`

2 级 DOM Traversal

释放 `NodeIterator` 对象

摘要

```
void detach();
```

描述

DOM 实现会跟踪为文档创建的所有 `NodeIterator` 对象, 因为在某些 `Document` 节点被删除时, 它们需要修改 `NodeIterator` 对象的状态。当你确认不再使用 `NodeIterator` 对象时, 可以调用 `detach()` 方法, 通知实现不要再跟踪它们。但要注意, 一旦调用了该方法, 那么接下来调用 `nextNode()` 和 `previousNode()` 方法都会抛出异常。

对 `detach()` 方法的调用不是必须的, 但在文档被修改, `NodeIterator` 对象不会被立刻回收时, 这样做会提高性能。

`NodeIterator.nextNode()`

2 级 DOM Traversal

遍历到下一个节点

摘要

```
Node nextNode()
    throws DOMException;
```

返回值

`NodeIterator` 对象表示的节点序列中的下一个节点。如果最后一个节点已经被返回了, 则返回 `null`。

抛出

如果在调用 `detach()` 方法之后调用该方法，它将抛出代码为 `INVALID_STATE_ERR` 的 `DOMException` 异常。

描述

该方法将前向遍历 `NodeIterator` 对象表示的节点序列。如果是第一次调用 `NodeIterator` 对象的该方法，它将返回节点序列中的第一个节点。否则，它将返回上一次返回的节点后的节点。

例子

```
// 创建一个 NodeIterator 以表示文档主体中的所有元素
var ni = document.createNodeIterator(document.body, NodeFilter.SHOW_ELEMENT,
                                     null, false);

// 向前循环迭代中的所有节点
for(var e = ni.nextNode(); e != null; e = ni.nextNode()) {
    // 使用元素 e 完成某些功能
}
```

NodeIterator.previousNode()

2 级 DOM Traversal

遍历到前一个节点

摘要

```
Node previousNode()
    throws DOMException;
```

返回值

`NodeIterator` 对象表示的节点序列中的前一个节点。如果没有前一个节点，则返回 `null`。

抛出

如果在调用 `detach()` 方法之后调用该方法，它将抛出代码为 `INVALID_STATE_ERR` 的 `DOMException` 异常。

描述

该方法将后向遍历 `NodeIterator` 对象表示的节点序列。它返回的节点位于 `nextNode()` 和 `previousNode()` 方法返回的最新章节之前。如果节点序列中不存在这样的节点，它将返回 `null`。

NodeList

1 级核心 DOM

节点的只读数组

属性

`readonly unsigned long length`

数组中的节点数。

方法

`item()`

返回数组的指定元素。

描述

`NodeList` 接口定义 `Node` 对象的只读有序列表（即数组）。`length` 属性声明了列表中有多少节点。用 `item()` 方法可以获取列表中指定位置的节点。`NodeList` 的元素都是有效的 `Node` 对象，它不会包含 `null` 元素。

在 JavaScript 中，`NodeList` 对象的行为和 JavaScript 数组相似，可以用数组符号 `[]` 从列表中查询元素，而不必调用 `item()` 方法。但不能用 `[]` 给 `NodeList` 对象添加新节点。由于把 `NodeList` 对象看做 JavaScript 只读数组进行处理更容易，所以本书采用 `Node[]`（即 `Node` 数组）代替 `NodeList`。例如，参阅“`Element.getElementsByTagName()`”参考页，它列出了返回的 `Node[]` 数组，而不是 `NodeList` 对象。同样，虽然技术上说，`Node` 对象的 `childNodes` 属性是 `NodeList` 对象，但“`Node`”参考页将它定义为 `Node[]`，属性自身通常由“`childNodes[]` 数组”引用。

注意，`NodeList` 对象是“活”的，它们不是静态的，会立刻反映出文档树的变化。例如，如果有一个 `NodeList` 对象，表示指定节点的子节点，然后删除其中一个子节点，那么该子节点将从 `NodeList` 中删除。如果在遍历 `NodeList` 的元素时，循环的主体修改了文档树（如删除节点），那么将影响 `NodeList` 的内容，所以要仔细操作。

参阅

`NamedNodeMap`Type of: `Node.childNodes`

Returned by: `Document.getElementsByTagName()`、`Document.getElementsByTagNameNS()`、`Element.getElementsByTagName()`、`Element.getElementsByTagNameNS()`、`HTMLDocument.GetElementsByTagName()`

NodeList.item()

1 级核心 DOM

获取 `NodeList` 中的元素**摘要**

```
Node item(unsigned long index);
```

参数`index`

要获取的节点在 `NodeList` 中的位置（或下标）。`NodeList` 中的第一个节点下标为 0，最后一个节点的下标为 `length - 1`。

返回值

`NodeList` 中指定位置的节点。如果 `index` 小于 0 或大于等于 `NodeList` 的长度，则返回 `null`。

描述

该方法将返回 `NodeList` 中的指定元素。在 JavaScript 中，可以用数组符号 `[]` 代替调用该方法。

Notation

1 级 DOM XML

XML DTD 中的符号

Node → Notation

属性

```
readonly String publicId
```

符号的公共标识符，如果没有指定，则为 `null`。

```
readonly String systemId
```

符号的系统标识符，如果没有指定，则为 `null`。

描述

这个不常用的接口表示 XML 文档的文档类型定义（DTD）中的符号声明。在 XML 中，符号用于指定未解析的实体的格式或正式声明一个处理指令目标。

符号的名字由继承的属性 `nodeName` 指定。因为符号出现在 DTD 中，而不是出现在文档自身，所以 `Notation` 节点不属于文档树，它的 `parentNode` 属性总为 `null`。`DocumentType` 接口的 `notations` 属性提供了根据符号名查找 `Notation` 对象的方法。

Notation 对象是只读的，不能被修改。

参阅

DocumentType

ProcessingInstruction

1 级 DOM XML

XML 文档中的处理指令

Node → ProcessingInstruction

属性

String data

处理指令的内容（即从目标开始后的第一个非空格字符到结束字符（但不包括）“>”之间的字符）。

readonly String target

处理指令的目标。它是“<?”后的第一个标识符，指定了处理指令的处理器。

描述

这个不常用的接口表示 XML 文档中的一个处理指令（或 PI）。使用 HTML 文档的程序设计者不会遇到 ProcessingInstruction 节点。

参阅

Returned by: Document.createProcessingInstruction()

Range

2 级 DOM Range

表示文档中的连续范围

常量

这些常量指定了如何比较 Range 对象的边界点。它们是 compareBoundaryPoints() 方法的参数 how 的合法值。参阅“Range.compareBoundaryPoints()”参考页。

unsigned short START_TO_START = 0

用指定范围的开始点与当前范围的开始点进行比较。

unsigned short START_TO_END = 1

用指定范围的开始点与当前范围的结束点进行比较。

unsigned short END_TO_END = 2

用指定范围的结束点与当前范围的结束点进行比较。


```
unsigned short END_TO_START = 3
```

用指定范围的结束点与当前范围的开始点进行比较。

属性

Range 接口定义了下列属性。注意，所有属性都是只读的，不能通过设置这些属性改变范围的开始点或结束点，必须调用 `setEnd()` 方法或 `setStart()` 方法实现这一点。还要注意，调用 Range 对象的 `detach()` 方法后，对这些属性的任何读操作都会抛出代码为 `INVALID_STATE_ERR` 的 `DOMException` 异常。

```
readonly boolean collapsed
```

如果范围的开始点和结束点在文档的同一位置，则为 `true`，也就是说，范围是空的，或“折叠”的。

```
readonly Node commonAncestorContainer
```

包含范围的开始点和结束点的（即它们的祖先节点）、嵌套最深的 Document 节点。

```
readonly Node endContainer
```

包含范围的结束点的 Document 节点。

```
readonly long endOffset
```

`endContainer` 中的结束点位置

```
readonly Node startContainer
```

包含范围的开始点的 Document 节点。

```
readonly long startOffset
```

开始点在 `startContainer` 中的位置。

方法

Range 接口定义了下列方法。注意，如果调用了范围的 `detach()` 方法，那么接下来调用 Range 对象的任何方法都会抛出代码为 `INVALID_STATE_ERR` 的 `DOMException` 异常。因为在该接口中的这种异常普遍存在，所以 Range 方法的参考页中没有列出它。

```
cloneContents()
```

返回新的 DocumentFragment 对象，它包含该范围表示的文档区域的副本。

```
cloneRange()
```

创建一个新 Range 对象，表示与当前的 Range 对象相同的文档区域。

`collapse()`

折叠该范围，使它的边界点重合。

`compareBoundaryPoints()`

比较指定范围的边界点和当前范围的边界点，根据它们的顺序返回 -1、0 和 1。

比较哪个边界点由它的第一个参数指定，它的值必须是前面定义的常量之一。

`deleteContents()`

删除当前 **Range** 对象表示的文档区域。

`detach()`

通知实现不再使用当前的范围，可以停止跟踪它。如果调用了范围的这个方法，那么接下来调用的该范围任何方法都会抛出代码为 `INVALID_STATE_ERR` 的 **DOMException** 异常。

`extractContents()`

删除当前范围表示的文档区域，并且以 **DocumentFragment** 对象的形式返回那个区域的内容。该方法和 `cloneContents()` 方法和 `deleteContents()` 方法的组合很相似。

`insertNode()`

把指定的节点插入文档范围的开始点。

`selectNode()`

设置该范围的边界点，使它包含指定的节点和它的所有子孙节点。

`selectNodeContents()`

设置该范围的边界点，使它包含指定节点的子孙节点，但不包含指定的节点自身。

`setEnd()`

把该范围的结束点设置为指定的节点和偏移量。

`setEndAfter()`

把该范围的结束点设置为紧邻指定节点的节点之后。

`setEndBefore()`

把该范围的结束点设置为紧邻指定节点之前。

`setStart()`

把该范围的开始点设置为指定的节点中的指定偏移量。

`setStartAfter()`

把该范围的开始点设置到紧邻指定节点的节点之后。

`setStartBefore()`

把该范围的开始点设置到紧邻指定节点之前。

`surroundContents()`

把指定的节点插入文档范围的开始点,然后重定范围中所有节点的父节点,使它们成为新插入的节点的子孙节点。

`toString()`

返回该范围表示的文档区域的纯文本内容。

描述

Range对象表示文档的连续范围或区域。如用户在浏览器窗口中用鼠标拖动选中的区域。如果一个实现支持Range模块,那么Document对象就定义了`createRange()`方法,调用它可创建新的Range对象(但要注意,Internet Explorer定义了不兼容的`Document.createRange()`方法,它返回的对象与Range接口相似,但不兼容)。Range接口为指定文档“选中”的区域定义了大量的方法,此外还有几个方法可以用于在选中的区域上进行剪切和粘贴类型的操作。

一个范围具有两个边界点,即一个开始点和一个结束点。每个边界点由一个节点和那个节点中的偏移量指定。该节点通常是Element节点、Document节点或Text节点。对于Element节点和Document节点,偏移量指该节点的子节点。偏移量为0,说明边界点位于该节点的第一个子节点之前。偏移量为1,说明边界点位于该节点的第一个子节点之后,第二个子节点之前。但如果边界节点是Text节点,偏移量则指的是文本中两个字符之间的位置。

Range接口的属性提供了获取范围的边界节点和偏移量的方法。它的方法提供了设置范围边界的方法。注意,范围的边界可以设置为Document或DocumentFragment对象中的节点。

一旦定义了范围的边界点,就可以使用`deleteContents()`、`extractContents()`、`cloneContents()`和`insertNode()`方法实现剪切、复制和粘贴的操作。

当通过插入或删除操作改变了文档时,表示文档一部分的所有Range对象都将被改变(如果必要的话),以便使它们的边界点保持有效,它们表示的文档内容不变。

要了解详细情况,请参阅每个Range方法的参考页和第十七章对Range API的讨论。

参阅

`Document.createRange()`、`DocumentFragment`；第十七章

Passed to: `Range.compareBoundaryPoints()`

Returned by: `Document.createRange()`、`Range.cloneRange()`

`Range.cloneContents()`

2 级 DOM Range

把范围的内容复制到一个 `DocumentFragment` 对象

摘要

```
DocumentFragment cloneContents()  
    throws DOMException;
```

返回值

一个 `DocumentFragment` 对象，包含该范围中的文档内容的副本。

抛出

如果当前的范围包含 `DocumentType` 节点，该方法将抛出代码为 `HIERARCHY_REQUEST_ERR` 的 `DOMException` 异常。

描述

该方法将复制当前范围的内容，把它存在在一个 `DocumentFragment` 对象中，返回该对象。

参阅

`DocumentFragment`、`Range.deleteContents()`、`Range.extractContents()`

`Range.cloneRange()`

2 级 DOM Range

复制该范围

摘要

```
Range cloneRange();
```

返回值

新创建的 `Range` 对象，与当前范围的边界点相同。

参阅

`Document.createRange()`

`Range.collapse()`

2 级 DOM Range

使范围的边界点重合

摘要

```
void collapse(boolean toStart)
    throws DOMException;
```

参数

toStart

如果该参数为 `true`, 该方法将把范围的结束点设置为与开始点相同的值。否则, 它将把范围的开始点设置为与结束点相同的值。

描述

该方法将设置范围的一个边界点, 使它与另一个边界点相同。要修改的边界点由参数 `toStart` 指定。该方法返回后, 范围将“折叠”, 即表示文档中的一个点, 没有内容。当范围被折叠后, 它的 `collapsed` 属性将被设置为 `true`。

`Range.compareBoundaryPoints()`

2 级 DOM Range

比较两个范围的位置

摘要

```
short compareBoundaryPoints(unsigned short how,
                             Range sourceRange)
    throws DOMException;
```

参数

how

声明如何执行比较操作 (即比较哪些边界点)。它的合法值是 `Range` 接口定义的常量。

sourceRange

要与当前范围进行比较的范围。

返回值

如果当前范围的指定边界点位于 `sourceRange` 指定的边界点之前, 则返回 `-1`。如果指定的两个边界点相同, 返回 `0`。如果当前范围的指定边界点位于 `sourceRange` 指定的边界点之后, 则返回 `1`。

抛出

如果 `sourceRange` 表示的文档不同于当前范围表示的文档, 该方法将抛出代码为 `WRONG_DOCUMENT_ERR` 的 `DOMException` 异常。

描述

该方法将比较当前范围的边界点和指定的 `sourceRange` 的边界点, 并返回一个值, 声明它们在源文档中的相对位置。参数 `how` 指定了比较两个范围的哪个边界点。该参数的合法值和它们的含义如下:

`Range.START_TO_START`

比较两个 `Range` 节点的开始点。

`Range.END_TO_END`

比较两个 `Range` 节点的结束点。

`Range.START_TO_END`

用 `sourceRange` 的开始点与当前范围的结束点比较。

`Range.END_TO_START`

用 `sourceRange` 的结束点与当前范围的开始点比较。

该方法的返回值是一个数字, 声明了当前范围相对于 `sourceRange` 的位置。因此, 你可能认为, 首先需要用参数 `how` 的范围常量指定当前范围的边界点, 然后再用它指定 `sourceRange` 的边界点。但事实上, 常量 `Range.START_TO_END` 指定与当前范围的 `end` 点和 `sourceRange` 的 `start` 点进行比较。同样, 常量 `Range.END_TO_START` 常量指定比较当前范围的 `start` 点和指定范围的 `end` 点。

`Range.deleteContents()`

2 级 DOM Range

删除文档的区域

摘要

```
void deleteContents()  
    throws DOMException;
```

抛出

如果当前范围表示的部分文档是只读的，该方法将抛出代码为 `NO_MODIFICATION_ALLOWED_ERR` 的 `DOMException` 异常。

描述

该方法将删除当前范围表示的所有文档内容。当该方法返回时，当前范围的边界点将重合。注意，这种删除操作可以生成相邻的 `Text` 节点，调用 `Node.normalize()` 方法可以合并这些节点。

关于复制文档内容的方法，请参阅“`cloneContents()`”。关于进行文档内容的复制和删除操作的方法，请参阅“`extractContents()`”。

参阅

`Node.normalize()`、`Range.cloneContents()`、`Range.extractContents()`

Range.detach()

2 级 DOM Range

释放一个 `Range` 对象

摘要

```
void detach()  
    throws DOMException;
```

抛出

和所有 `Range` 方法一样，如果在已经被释放了的 `Range` 对象上调用该方法，它将抛出代码为 `INVALID_STATE_ERR` 的 `DOMException` 异常。

描述

DOM 实现将跟踪为文档创建的所有 `Range` 对象，因为在修改文档时，它们需要改变范围的边界点。当确认 `Range` 对象不再被使用时，可以调用 `detach()` 方法，通知实现不必再跟踪该范围。注意，一旦调用了 `Range` 对象的 `detach()` 方法，再使用 `Range` 对象，就会抛出异常。对 `detach()` 方法的调用不是必须的，但在修改了文档，`Range` 对象不会被立刻回收的情况下，调用它可以提高性能。

Range.extractContents()

2 级 DOM Range

删除文档内容并以 DocumentFragment 对象的形式返回它

摘要

```
DocumentFragment extractContents()  
    throws DOMException;
```

返回值

一个 DocumentFragment 节点，包含该范围的内容。

抛出

如果要提取的文档内容是只读的，该方法将抛出代码为 NO_MODIFICATION_ALLOWED_ERR 的 DOMException 异常。如果当前范围包括 DocumentType 节点，该方法将抛出代码为 HIERARCHY_REQUEST_ERR 的 DOMException 异常。

描述

该方法将删除文档的指定范围，并返回包含被删除的内容（或被删除的内容的副本）的 DocumentFragment 节点。当返回该方法时，范围将折叠，文档中可能出现相邻的 Text 节点（用 Node.normalize() 方法可以合并）。

参阅

DocumentFragment、Range.cloneContents()、Range.deleteContents()

Range.insertNode()

2 级 DOM Range

在范围的开头插入一个节点

摘要

```
void insertNode(Node newNode)  
    throws RangeException,  
           DOMException;
```

参数

newNode

要插入文档的节点。

抛出

如果 `newNode` 是 `Attr`、`Document`、`Entity` 或 `Notation` 节点，该方法将抛出代码为 `INVALID_NODE_TYPE_ERR` 的 `RangeException` 异常。

在下列条件下，该方法还将抛出如下代码的 `DOMException` 异常：

`HIERARCHY_REQUEST_ERR`

包含范围的开始点的节点不能有子节点，它也不能有指定类型的子节点，或者 `newNode` 是该节点的祖先节点（或是该节点自身）。

`NO_MODIFICATION_ALLOWED_ERR`

包含范围的开始点的节点（或它的祖先节点）是只读的。

`WRONG_DOCUMENT_ERR`

`newNode` 与范围所属的文档不同。

描述

该方法将把指定的节点（和它的所有子孙节点）插入文档范围的开始点。当该方法返回时，当前范围将包括新插入的节点。如果 `newNode` 已经是文档的一部分，那么它将被从当前位置删除，然后重新插入范围的开始点。如果 `newNode` 是 `DocumentFragment` 节点，那么插入的不是它自身，而是它的所有子节点，按顺序插入范围的开始点。

如果包含当前范围的开始点的节点是 `Text` 节点，那么在发生插入操作前，它将被分割成两个相邻的节点。如果 `newNode` 是 `Text` 节点，在插入文档后，它不会与任何相邻的 `Text` 节点合并。要合并相邻的节点，需要调用 `Node.normalize()` 方法。

参阅

`DocumentFragment`、`Node.normalize()`

`Range.selectNode()`

2 级 DOM Range

把范围边界设置为一个节点

摘要

```
void selectNode(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

被选中的节点（即将成为当前范围的内容的节点）。

抛出

如果 *refNode* 是 Attr、Document、DocumentFragment、Entity 或 Notation 节点，或者 *refNode* 的祖先之一是 DocumentType、Entity 或 Notation 节点，该方法将抛出代码为 INVALID_NODE_TYPE_ERR 的 RangeException 异常。

如果 *refNode* 所属的文档与创建该范围的文档不同，该方法将抛出代码为 WRONG_DOCUMENT_ERR 的 DOMException 异常。

描述

该方法将把范围的内容设置为指定的 *refNode* 节点。也就是说，“选中”那个节点和它的子孙节点。

参阅

Range.selectNodeContents()

Range.selectNodeContents()

2 级 DOM Range

把范围的边界设置为一个节点的子节点

摘要

```
void selectNodeContents(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

其子节点将成为当前范围的内容的节点。

抛出

如果 *refNode* 或它的祖先节点是 DocumentType、Entity 或 Notation 节点，该方法将抛出代码为 INVALID_NODE_TYPE_ERR 的 RangeException 异常。

如果 *refNode* 所属的文档与创建该范围的文档不同，该方法将抛出代码为 WRONG_DOCUMENT_ERR 的 DOMException 异常。

描述

该方法将设置范围的边界点，使该范围包含 *refNode* 的子节点。

参阅

`Range.selectNode()`

`Range.setEnd()`

2 级 DOM Range

设置范围的结束点

摘要

```
void setEnd(Node refNode,  
            long offset)  
    throws RangeException,  
           DOMException;
```

参数

refNode

包含新的结束点的节点。

offset

结束点在 *refNode* 中的位置。

抛出

如果 *refNode* 或它的祖先节点是 `DocumentType`、`Entity` 或 `Notation` 节点，该方法将抛出代码为 `INVALID_NODE_TYPE_ERR` 的 `RangeException` 异常。

如果 *refNode* 所属的文档与创建该范围的文档不同，该方法将抛出代码为 `WRONG_DOCUMENT_ERR` 的 `DOMException` 异常。如果 *offset* 是负数，或者大于 *refNode* 中的子节点数或字符数，该方法将抛出代码为 `INDEX_SIZE_ERR` 的 `DOMException` 异常。

描述

该方法将把范围的结束点设置为 `endContainer` 属性和 `endOffset` 属性指定的值。

Range.setEndAfter()

2 级 DOM Range

在指定的节点后结束范围

摘要

```
void setEndAfter(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

一个节点，要设置的范围的结束点位于该节点之后。

抛出

如果 *refNode* 是 Document、DocumentFragment、Attr、Entity 或 Notation 节点，或者 *refNode* 的根包容节点不是 Document、DocumentFragment 或 Attr 节点，该方法将抛出代码为 INVALID_NODE_TYPE_ERR 的 RangeException 异常。

如果 *refNode* 所属的文档与创建该范围的文档不同，该方法将抛出代码为 WRONG_DOCUMENT_ERR 的 DOMException 异常。

描述

该方法将把范围的结束点设置为紧邻指定的 *refNode* 节点的位置。

Range.setEndBefore()

2 级 DOM Range

在指定的节点之前结束范围

摘要

```
void setEndBefore(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

一个节点，要设置的范围的结束点位于该节点之前。

抛出

该方法将在与 `Range.setEndAfter()` 方法一样的环境中抛出相同的异常。详见 `Range.setEndAfter()` 方法的参考页。

描述

该方法将把范围的结束点设置为指定的 `refNode` 节点之前的位置。

Range.setStart()

2 级 DOM Range

设置范围的开始点

摘要

```
void setStart(Node refNode,  
              long offset)  
    throws RangeException,  
           DOMException;
```

参数

refNode

包含新的开始点的节点。

offset

新开始点在 *refNode* 中的位置。

抛出

该方法抛出异常的原因和 `Range.setEnd()` 方法相同，抛出的异常也相同。详见 `Range.setEnd()` 方法的参考页。

描述

该方法将把范围的开始点设置为 `startContainer` 属性和 `startOffset` 属性指定的值。

Range.setStartAfter()

2 级 DOM Range

在指定的节点后开始范围

摘要

```
void setStartAfter(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

一个节点，要设置的范围的开始点位于该节点后。

抛出

该方法将出于和 `Range.setEndAfter()` 方法同样的原因抛出同样的异常。详见 `Range.setEndAfter()` 方法的参考页。

描述

该方法将把范围的开始点设置为紧邻指定的 *refNode* 节点的位置。

Range.setStartBefore()

2 级 DOM Range

在指定的节点之前开始范围

摘要

```
void setStartBefore(Node refNode)
    throws RangeException,
           DOMException;
```

参数

refNode

一个节点，要设置的范围的开始点位于该节点之前。

抛出

该方法将出于和 `Range.setEndAfter()` 方法同样的原因抛出同样的异常。详见 `Range.setEndAfter()` 方法的参考页。

描述

该方法将把范围的开始点设置为指定的 *refNode* 节点之前的位置。

Range.surroundContents()2 级 DOM Range

用指定的节点包围范围的内容

摘要

```
void surroundContents(Node newParent)
    throws RangeException,
           DOMException;
```

参数

newParent

将成为当前范围内容的新父节点的节点。

抛出

该方法将在下列环境中抛出具有如下代码的 DOMException 异常或 RangeException 异常：

DOMException.HIERARCHY_REQUEST_ERR

当前范围的开始点的包容节点不能有子节点，不能有 *newParent* 类型的子节点，或者 *newParent* 是包容节点的祖先节点。

DOMException.NO_MODIFICATION_ALLOWED_ERR

当前范围的边界点的祖先节点是只读的，不允许进行插入操作。

DOMException.WRONG_DOCUMENT_ERR

newParent 它的范围是用不同 Document 对象创建的。

RangeException.BAD_BOUNDARYPOINTS_ERR

当前范围部分地选择了一个节点（除了 Text 节点外的），所以不能包围文档的这个区域。

RangeException.INVALID_NODE_TYPE_ERR

newParent 是 Document、DocumentFragment、DocumentType、Attr、Entity 或 Notation 节点。

描述

该方法将把当前范围的父节点重定为 *newParent*，然后把 *newParent* 插在文档中范围的开始位置。例如，把文档的一个区域放入 或 元素中，可以使用该方法。

如果 `newParent` 已经是文档的一部分，那么它首先将从文档中删除，它的子节点也将被舍弃。当该方法返回时，该范围将以 `newParent` 之前的位置为开始点，`newParent` 之后的位置为结束点。

Range.toString()

2 级 DOM Range

以纯文本串形式获取范围的内容

摘要

```
String toString();
```

返回值

以纯文本（不含标记）的形式返回当前范围的内容。

RangeException

2 级 DOM Range

通知发生了范围特有的异常

常量

下面的常量为 `RangeException` 对象的 `code` 属性定义了合法的值。注意，这些常量是 `RangeException` 接口的静态属性，不是个别异常对象的属性。

```
unsigned short BAD_BOUNDARYPOINTS_ERR = 1
```

对于请求的操作，范围的边界点不合法。

```
unsigned short INVALID_NODE_TYPE_ERR = 2
```

尝试把范围边界点的包容节点设置为一个无效节点或具有无效祖先的节点。

属性

```
unsigned short code
```

提供引发异常的详细信息的错误代码。该属性的合法值由前面列出的常量定义。

描述

`RangeException` 对象由 `Range` 接口的某些方法抛出，用于通知某种类型的问题。注意，`Range` 方法抛出的大多数异常都是 `DOMException` 对象。只有当现有的 `DOMException` 错误常量不适合表示当前的异常时，才生成 `RangeException` 对象。

Rect2 级 DOM CSS

一个 CSS rect() 值

属性

readonly CSSPrimitiveValue bottom

矩形的底边。

readonly CSSPrimitiveValue left

矩形的左边线。

readonly CSSPrimitiveValue right

矩形的右边线。

readonly CSSPrimitiveValue top

矩形的上边线。

描述

该接口表示一个 CSS rect (top, right, bottom, left) 值，与 CSS 性质 clip 一起使用。详见 CSS 参考手册。

参阅

Returned by: CSSPrimitiveValue.getRectValue()

RGBColor2 级 DOM CSS

一个 CSS 颜色值

属性

readonly CSSPrimitiveValue blue

该颜色的蓝色成分。

readonly CSSPrimitiveValue green

该颜色的绿色成分。

readonly CSSPrimitiveValue red

该颜色的红色成分。

描述

该接口表示在 RGB 颜色空间中设置的颜色。它的属性是 CSSPrimitiveValue 对象，分

别指定了该颜色的红色、绿色和蓝色成分的值。每个 `CSSPrimitiveValue` 对象存放一个 0~255 之间的数字，或 0~100% 之间的百分比。

参阅

Returned by: `CSSPrimitiveValue.getRGBColorValue()`

StyleSheet

2 级 DOM StyleSheet

任意类型的样式表

子接口

`CSSStyleSheet`

属性

`boolean disabled`

如果该属性值为 `true`，则将禁用这个样式表，不能应用到文档。如果为 `false`，样式表将被激活，应用到文档（除非 `media` 属性指定不能将该样式表应用到这种类型的文档）。

`readonly String href`

链接到文档的样式表的 URL。对于内联样式表，该属性为 `null`。

`readonly MediaList media`

应该应用该样式表的媒体类型的列表。如果没有给样式表提供媒体信息，该属性仍然是一个有效的 `MediaList` 对象，其 `length` 属性为 0。

`readonly Node ownerNode`

把样式表链接到文档的 `Document` 节点，或包含内联样式表的节点。在 HTML 文档中，该属性引用一个 `<link>` 元素或 `<style>` 元素。对于包含在其他样式表而不是直接包含在文档中的样式表，该属性为 `null`。

`readonly StyleSheet parentStyleSheet`

包含当前样式表的样式表。如果当前样式表直接包含在文档中，则该属性为 `null`。参阅 “`CSSStyleSheet.ownerRule`”。

`readonly String title`

样式表的标题（如果存在的话）。标题必须由 `<style>` 标记或 `<link>` 标记（该样式表的 `ownerNode` 节点）的 `title` 性质设置。

readonly String type

样式表的类型，和 MIME 类型一样。CSS 样式表的类型是 “text/css”。

描述

这个接口表示关联到文档的样式表。如果这是一个 CSS 样式表，实现 StyleSheet 接口的对象还会实现 CSSStyleSheet 子接口，定义能够检测和设置构成样式表的 CSS 规则的属性和方法。详见 “CSSStyleSheet”。

如果 DOM 实现支持样式表，可以通过 Document.styleSheets 属性获取关联到文档的样式表的列表。另外，HTML 的 <style> 元素和 <link> 元素，以及 XML 样式表的 ProcessingInstruction 节点实现了 LinkStyle 接口。通过属性 sheet 提供对 StyleSheet 对象的引用。

参阅

CSSStyleSheet、Document.styleSheets、LinkStyle

Type of: LinkStyle.sheet、StyleSheet.parentStyleSheet

Returned by: StyleSheetList.item()

StyleSheetList

2 级 DOM StyleSheets

样式表的数组

属性

readonly unsigned long length

数组中的 StyleSheet 对象的个数。

方法

item()

返回数组中指定位置的 StyleSheet 对象。如果指定的位置是负数或大于等于 length 属性的值，该方法将返回 null。

描述

该接口定义了 StyleSheet 对象的数组。length 属性声明了数组中的样式表个数。item() 方法提供了获取指定位置的样式表的方法。在 JavaScript 中，可以将 StyleSheetList 对象作为只读数组处理，用数组符号[]索引它，而不必调用 item() 方法。

参阅

Type of: `DocumentStyle.styleSheets`

StyleSheetList.item()

2 级 DOM StyleSheet

索引样式表的数组

摘要

`StyleSheet item(unsigned long index);`

参数

index

想获取的样式表在数组中的位置。

返回值

数组中指定位置的 `StyleSheet` 对象。如果 *index* 是负数，或者大于等于 `length` 属性的值，则返回 `null`。在 JavaScript 中，将 `StyleSheet` 对象作为数组处理更简单，用数组符号 `[]` 索引它，而不必调用该方法。

Text

1 级核心 DOM

HTML 或 XML 文档中的一系列文本

Node → `CharacterData` → `Text`

子接口

`CDATASection`

方法

`splitText()`

在指定的字符位置把一个 `Text` 节点分割成两个，并返回新的 `Text` 节点。

描述

`Text` 节点表示 HTML 或 XML 文档中的一系列纯文本。因为纯文本出现在 HTML 和 XML 的元素和性质中，所以 `Text` 节点通常作为 `Element` 节点和 `Attr` 节点的子节点出现。`Text` 节点继承了 `CharacterData` 接口，通过从 `CharacterData` 接口继承的 `data` 属性或从 `Node` 接口继承的 `nodeValue` 属性，可以访问 `Text` 节点的文本内容。用从 `CharacterData` 继承的方法或 `Text` 接口自身定义的 `splitText()` 方法可以操作 `Text` 节点。`Text` 节点没有子节点。

关于从文档的子树中删除空 Text 节点与合并相邻的 Text 节点的方法，请参阅“Node.normalize()”参考页。

参阅

CharacterData、Node.normalize()

Returned by: Document.createTextNode()、Text.splitText()

Text.splitText()

1 级核心 DOM

把一个 Text 节点分割成两个

摘要

```
Text splitText(unsigned long offset)  
    throws DOMException;
```

参数

offset

分割 Text 节点的字符位置。

返回值

从当前节点分割出的 Text 节点。

抛出

该方法将抛出具有下列代码之一的 DOMException 异常：

INDEX_SIZE_ERR

Offset 是负数，或者大于 Text 或 Comment 节点的长度。

NO_MODIFICATION_ALLOWED_ERR

该节点是只读的，不能修改。

描述

该方法将在指定的 *offset* 处把 Text 节点分割成两个节点。原始的 Text 节点将被修改，使它包含 *offset* 指定的位置之前的文本内容（但不包括该文本内容）。新的 Text 节点将被创建，用于存放从 *offset* 位置（包括该位置上的字符）到原字符串结尾的所有字符。新的 Text 节点是该方法的返回值。此外，如果原始的 Text 节点具有 *parentNode*，新的 Text 节点将插入这个父节点，紧邻在原始节点之后。

CDATASection接口继承了Text接口, CDATASection节点也可以使用该方法, 只是新创建的节点是CDATASection节点, 而不是Text节点。

参阅

Node.normalize()

TreeWalker

2 级 DOM Traversal

遍历过滤后的文档子树

属性

Node currentNode

TreeWalker的当前位置, 是与所有TreeWalker遍历方法操作有关的节点。这个节点是由某个遍历方法最近返回的, 如果还没有任何遍历方法被调用, 那么它与root属性相同。

注意, 这是一个可读可写的属性, 可以把它设置为任何有效的Document节点, 即使该节点不是原始root节点的子孙节点, 或被TreeWalker使用的过滤器拒绝了。如果改变该属性的值, 遍历方法的操作将与设置的新节点有关。如果把该属性设置为null, 将抛出代码为NOT_SUPPORTED_ERR的DOMException异常。

readonly boolean expandEntityReferences

这个只读属性指定了该TreeWalker对象在遍历XML文档时是否扩展它遇到的实体引用。这个属性的值可以通过调用Document.createTreeWalker()方法设置。

readonly NodeFilter filter

节点过滤函数(如果存在), 调用Document.createTreeWalker()方法时为该TreeWalker对象设置的。如果没有使用节点过滤函数, 该属性为null。

readonly Node root

这个只读属性指定了TreeWalker对象开始遍历的root节点。它是currentNode属性的初始值, 调用Document.createTreeWalker()方法设置。

readonly unsigned long whatToShow

这个只读属性是位标志的集合(有效标志的列表参阅“NodeFilter”参考页), 指定了当前TreeWalker对象考虑的Document节点的类型。如果该属性中的某个位没有被设置, 那么TreeWalker对象将忽略相应的节点类型。注意, 该属性的值是在调用Document.createTreeWalker()方法时指定的。

方法

`firstChild()`

返回没有被过滤掉的当前节点的第一个子节点或 `null`。

`lastChild()`

返回没有被过滤掉的当前节点的最后一个子节点或 `null`。

`nextNode()`

返回（按源文档顺序中）没有被过滤掉的下一个节点或 `null`。

`nextSibling()`

返回没有被过滤掉的当前节点的下一个兄弟节点或 `null`。

`parentNode()`

返回没有被过滤掉的当前节点的父节点、最近的祖先节点或 `null`。

`previousNode()`

返回（按源文档顺序中）没有被过滤掉的前一个节点或 `null`。

`previousSibling()`

返回没有被过滤掉的当前节点的前一个兄弟节点或 `null`。

描述

`TreeWalker` 对象用于过滤指定的文档子树，定义了遍历过滤后的文档树（可能与原始文档树的结构有很大差异）的方法。用 `Document` 对象的 `createTreeWalker()` 方法可以创建 `TreeWalker` 对象。一旦创建了 `TreeWalker` 对象，就可以用它的 `firstChild()` 和 `nextSibling()` 方法遍历它表示的过滤后的文档子树，这与使用 `Node` 接口的 `firstChild` 属性和 `nextSibling` 属性遍历没有过滤的文档树一样。

`TreeWalker` 接口采用和 `NodeIterator` 接口一样的两步过滤步骤。 `TreeWalker` 定义的各种遍历方法只返回两步过滤都通过的节点。首先，节点类型必须是 `whatToShow` 属性指定的类型之一。可以组合起来指定 `Document.createTreeWalker()` 方法的 `whatToShow` 参数的常量列表，请参阅“`NodeFilter`”参考页。接下来，如果 `filter` 属性不为 `null`，那么进行 `whatToShow` 检测的节点还要传递给 `filter` 属性指定的过滤函数。如果该函数返回 `NodeFilter.FILTER_ACCEPT`，该节点将被返回。如果它返回 `NodeFilter.FILTER_REJECT`，`TreeWalker` 将跳过该节点和它的所有子孙节点（这点与 `NodeIterator` 过滤不同，在 `NodeIterator` 过滤中，所有子孙节点都不会被自动拒绝）。如果节点过滤函数返回 `NodeFilter.FILTER_SKIP`，`TreeWalker` 将忽略该节点，但仍然考虑它的子孙节点。

与 NodeIterator 不同，在基础文档被修改时，TreeWalker 不会被修改。TreeWalker 的当前节点保持不变，即使从文档中删除了该节点 [在这种情况下，TreeWalker 可以用于遍历包围当前节点的被删除节点的树（如果存在的话）]。

例子

```
// NodeFilter 对象拒绝 <font> 标签和具有
// class="sidebar" 性质的元素以及这种元素的子孙
var filter = function (n) {
    if (n.nodeName == "FONT") return NodeFilter.FILTER_SKIP;
    if (n.nodeType == Node.ELEMENT_NODE && n.className == "sidebar")
        return NodeFilter.FILTER_REJECT;
    return NodeFilter.FILTER_ACCEPT;
}

// 用上面的 filter 创建一个 TreeWalker 对象
var tw = document.createTreeWalker(document.body, // 遍历 HTML 文档的主体
    // 考虑除注释之外的所有节点
    NodeFilter.SHOW_COMMENT,
    filter, // 使用上面的 filter 对象
    false); // 不要扩展实体引用

// 下面是用 TreeWalker 对象遍历文档的递归函数
function traverse(tw) {
    // 记住当前节点
    var currentNode = tw.currentNode;
    // 遍历 TreeWalker 当前节点的子节点
    for(var c = tw.firstChild(); c != null; c = tw.nextSibling()) {
        process(c); // 处理子节点
        traverse(tw); // 递归处理它的子节点
    }

    // 把 TreeWalker 恢复到我们创建它时的状态
    tw.currentNode = currentNode;
}
```

参阅

NodeFilter, NodeIterator; 第十七章

Returned by: Document.createTreeWalker()

TreeWalker.firstChild()

2 级 DOM Traversal

返回没有被过滤掉的第一个子节点

摘要

Node firstChild();

返回值

没有被过滤掉的当前节点的第一个子节点。如果没有这样的子节点，则返回 `null`。

描述

该方法将 `currentNode` 属性设置为没有被过滤掉的当前节点的第一个子节点，并返回该子节点。如果没有这样的子节点，它将不改变 `currentNode` 属性，并返回 `null`。

TreeWalker.lastChild()

2 级 DOM Traversal

返回没有被过滤掉的最后一个子节点

摘要

```
Node lastChild();
```

返回值

没有被过滤掉的当前节点的最后一个子节点。如果没有这样的子节点，则返回 `null`。

描述

该方法将 `currentNode` 属性设置为没有被过滤掉的当前节点的最后一个子节点，并返回该子节点。如果没有这样的子节点，它将不改变 `currentNode` 属性，并返回 `null`。

TreeWalker.nextSibling()

2 级 DOM Traversal

返回没有被过滤掉的下一个节点

摘要

```
Node nextNode();
```

返回值

在文档源代码中位于当前节点之后、没有被过滤掉的节点。如果不存在这样的节点，则返回 `null`。

描述

该方法将把 `currentNode` 属性设置为没有被过滤掉的下一个节点(在文档源代码中)，并返回该节点。如果没有这样的节点，或者对下一个节点的检索使 `TreeWalker` 在 `root` 子树之外发生，`currentNode` 将保持不变，该方法返回 `null`。

TreeWalker.nextSibling()2 级 DOM Traversal

返回没有被过滤掉的下一个兄弟节点

摘要

```
Node nextSibling();
```

返回值

没有被过滤掉的当前节点的下一个兄弟节点。如果没有这样的节点，则返回 `null`。

描述

该方法将把 `currentNode` 属性设置为没有被过滤掉的当前节点的下一个兄弟节点，并返回该兄弟节点。如果没有这样的兄弟节点，该方法将不改变 `currentNode` 属性，并返回 `null`。

TreeWalker.parentNode()2 级 DOM Traversal

返回没有过滤掉的节点的最近的祖先节点

摘要

```
Node parentNode();
```

返回值

没有被过滤掉的当前节点的最近祖先节点，如果没有这样的节点，则返回 `null`。

描述

该方法将把 `currentNode` 属性设置为没有被过滤掉的当前节点的最近祖先节点，并返回该祖先节点。如果没有这样的祖先节点，它将不改变 `currentNode` 属性，返回 `null`。

TreeWalker.previousNode()2 级 DOM Traversal

返回没有被过滤掉的前一个节点

摘要

```
Node previousNode();
```

返回值

在文档源代码中位于当前节点之前、没有被过滤掉的节点。如果不存在这样的节点，则返回 `null`。

描述

该方法将把 `currentNode` 属性设置为没有被过滤掉的节点的前一个节点（在文档源代码中），并返回该节点。如果没有这样的节点，或者对前一个节点的检索使 `TreeWalker` 在 `root` 子树之外发生，`currentNode` 将保持不变，该方法返回 `null`。

注意，该方法将简化文档树的结构，按照节点在源文档中出现的顺序返回它们。调用该方法会使当前节点在文档树中下移或上移。用 `NodeIterator.previousNode()` 也可以执行这种简化类型的遍历。

TreeWalker.previousSibling()

2 级 DOM Traversal

返回没有被过滤掉的前一个兄弟节点

摘要

```
Node previousSibling();
```

返回值

没有被过滤掉的当前节点的前一个兄弟节点。如果没有这样的兄弟节点，则返回 `null`。

描述

该方法将把 `currentNode` 属性设置为没有被过滤掉的当前节点的前一个兄弟节点，并返回该兄弟节点。如果没有这样的兄弟节点，该方法将不改变 `currentNode` 属性，返回 `null`。

UIEvent

2 级 DOM Events

用户界面事件的详细情况

Event → UIEvent

子接口

MouseEvents

属性

`readonly long detail`

关于该事件的数字细节，对于 `click`、`mousedown`、`mouseup` 事件来说（参阅“`MouseEvent`”），这个域填写点击的次数，1 表示单击，2 表示双击，3 表示击键三次，以此类推。对于 `DOMActivate` 事件，该域为 1 表示普通激活，2 表示“超级激活”，如双击或 **Shift-Enter** 键的组合。

`readonly AbstractView view`

生成事件的窗口（视图）。

方法

`initUIEvent()`

初始化新创建的 `UIEvent` 对象的属性，包括由 `Event` 接口继承的属性。

描述

该接口是 `Event` 接口的子接口，定义了传递给 `DOMFocusIn`、`DOMFocusOut` 和 `DOMActivate` 类型的事件的 `Event` 对象的类型。在 Web 浏览器中，不常使用这些事件类型，关于 `UIEvent` 接口要记住的重要一点是，它是 `MouseEvent` 接口的父接口。

参阅

`Event`、`MouseEvent`；第十九章

`UIEvent.initUIEvent()`

2 级 DOM Events

初始化 `UIEvent` 对象的属性

摘要

```
void initUIEvent(String typeArg,  
                  boolean canBubbleArg,  
                  boolean cancelableArg,  
                  AbstractView viewArg,  
                  long detailArg);
```

参数

`typeArg`

事件类型。

canBubbleArg

事件是否可以起泡。

cancelableArg

是否可以用 `preventDefault()` 方法取消该事件。

viewArg

发生事件的窗口。

detailArg

事件的 `detail` 属性。

描述

该方法将初始化 `UIEvent` 对象的 `view` 属性和 `detail` 属性以及从 `Event` 接口继承的 `type` 属性、`bubbles` 属性、`cancelable` 属性。只有新创建的 `UIEvent` 对象才能调用该方法，而且必须在把它传递给 `EventTarget.dispatchEvent()` 方法之前调用。

ViewCSS

参阅 `AbstractView`

第六部分

类、属性、方法 和事件处理程序索引

本书的这一部分是JavaScript中所有类、属性、方法和事件处理程序的索引。用这个索引有助于找到这些项目的参考资料。

类、属性、方法 和事件处理程序索引

当你在前面的参考手册中查找资料，而又不知道在哪里查找时，可以使用下面的索引。例如，如果你想查找一个类或接口，但不知道哪个参考部分记述了它，便可以在这里查找该类或接口的名字，索引将告诉你在哪个参考部分可以找到它。符号“[Core]”表示 JavaScript 的核心参考手册，“[Client]”表示客户端的参考手册，“[DOM]”表示 DOM 参考手册。

如果你想查找方法、属性或事件处理程序，但不知道什么类定义了它，也可以在这个索引中查找方法、属性或事件处理程序的名字，它将告诉你在（哪个参考部分的）哪个或哪些类下可以找到。

A

abbr: HTMLTableCellElement[DOM]

ABORT: Event[Client]

above: Layer[Client]

abs: Math[Core]

AbstractView: [DOM]

accept: HTMLInputElement[DOM]

acceptCharset: HTMLFormElement[DOM]

accessKey: HTMLAnchorElement[DOM],

HTMLInputElement[DOM],

HTMLTextAreaElement[DOM]

acos: Math[Core]

action: Form[Client], HTMLFormElement[DOM]

add: HTMLSelectElement[DOM]

addEventListener: EventTarget[DOM]

ADDITION: MutationEvent[DOM]

alert: Window[Client]

align: HTMLInputElement[DOM],

HTMLTableCaptionElement[DOM],

HTMLTableCellElement[DOM],

HTMLTableColElement[DOM],

HTMLTableElement[DOM],

HTMLTableRowElement[DOM],
 HTMLTableSectionElement[DOM]
aLink: HTMLBodyElement[DOM]
alinkColor: Document[Client]
all: Document[Client], HTMLElement[Client]
alt: HTMLInputElement[DOM]
altKey: MouseEvent[DOM]
Anchor: [Client]
anchors: Document[Client],
 HTMLDocument[DOM]
appCodeName: Navigator[Client]
appendChild: Node[DOM]
appendData: CharacterData[DOM]
appendMedium: MediaList[DOM]
Applet: [Client]
applets: Document[Client],
 HTMLDocument[DOM]
apply: Function[Core]
appName: Navigator[Client]
appVersion: Navigator[Client]
Area: [Client]
Arguments: [Core]
arguments: [Core], Function[Core]
Array: [Core]
asin: Math[Core]
atan: Math[Core]
atan2: Math[Core]
Attr: [DOM]
attrChange: MutationEvent[DOM]
attributes: Node[DOM]
ATTRIBUTE_NODE: Node[DOM]
attrName: MutationEvent[DOM]
AT_TARGET: Event[DOM]

availHeight: Screen[Client]
availLeft: Screen[Client]
availTop: Screen[Client]
availWidth: Screen[Client]
axis: HTMLTableCellElement[DOM]
azimuth: CSS2Properties[DOM]

B

back: History[Client], Window[Client]
background: CSS2Properties[DOM],
 HTMLBodyElement[DOM], Layer[Client]
backgroundAttachment: CSS2Properties[DOM]
backgroundColor: CSS2Properties[DOM]
backgroundImage: CSS2Properties[DOM]
backgroundPosition: CSS2Properties[DOM]
backgroundRepeat: CSS2Properties[DOM]
BAD_BOUNDARYPOINTS_ERR:
 RangeException[DOM]
below: Layer[Client]
bgColor: Document[Client],
 HTMLBodyElement[DOM],
 HTMLTableCellElement[DOM],
 HTMLTableElement[DOM],
 HTMLTableRowElement[DOM], Layer[Client]
blue: RGBColor[DOM]
BLUR: Event[Client]
blur: HTMLAnchorElement[DOM],
 HTMLInputElement[DOM],
 HTMLSelectElement[DOM],
 HTMLTextAreaElement[DOM], Input[Client],
 Window[Client]
body: HTMLDocument[DOM]
Boolean: [Core]

border: CSS2Properties[DOM],
HTMLTableElement[DOM], Image[Client]
borderBottom: CSS2Properties[DOM]
borderBottomColor: CSS2Properties[DOM]
borderBottomStyle: CSS2Properties[DOM]
borderBottomWidth: CSS2Properties[DOM]
borderCollapse: CSS2Properties[DOM]
borderColor: CSS2Properties[DOM]
borderLeft: CSS2Properties[DOM]
borderLeftColor: CSS2Properties[DOM]
borderLeftStyle: CSS2Properties[DOM]
borderLeftWidth: CSS2Properties[DOM]
borderRight: CSS2Properties[DOM]
borderRightColor: CSS2Properties[DOM]
borderRightStyle: CSS2Properties[DOM]
borderRightWidth: CSS2Properties[DOM]
borderSpacing: CSS2Properties[DOM]
borderStyle: CSS2Properties[DOM]
borderTop: CSS2Properties[DOM]
borderTopColor: CSS2Properties[DOM]
borderTopStyle: CSS2Properties[DOM]
borderTopWidth: CSS2Properties[DOM]
borderWidth: CSS2Properties[DOM]
bottom: CSS2Properties[DOM], Rect[DOM]
bubbles: Event[DOM]
BUBBLING_PHASE: Event[DOM]
Button: [Client]
button: MouseEvent[DOM]

C

call: Function[Core], JSObject[Client]
callee: Arguments[Core]
caller: Function[Core]

cancelable: Event[DOM]
caption: HTMLTableElement[DOM]
captionSide: CSS2Properties[DOM]
captureEvents: Document[Client], Layer[Client],
Window[Client]
CAPTURING_PHASE: Event[DOM]
CDATASection: [DOM]
CDATA_SECTION_NODE: Node[DOM]
ceil: Math[Core]
cellIndex: HTMLTableCellElement[DOM]
cellPadding: HTMLTableElement[DOM]
cells: HTMLTableRowElement[DOM]
cellSpacing: HTMLTableElement[DOM]
ch: HTMLTableCellElement[DOM],
HTMLTableColElement[DOM],
HTMLTableRowElement[DOM],
HTMLTableSectionElement[DOM]
CHANGE: Event[Client]
CharacterData: [DOM]
charAt: String[Core]
charCodeAt: String[Core]
charset: HTMLAnchorElement[DOM]
CHARSET_RULE: CSSRule[DOM]
Checkbox: [Client]
checked: Checkbox[Client],
HTMLInputElement[DOM], Input[Client],
Radio[Client]
childNodes: Node[DOM]
children: HTMLElement[Client]
chOff: HTMLTableCellElement[DOM],
HTMLTableColElement[DOM],
HTMLTableRowElement[DOM],
HTMLTableSectionElement[DOM]

- className**: HTMLElement[Client],
 HTMLElement[DOM]
clear: CSS2Properties[DOM], Document[Client]
clearInterval: Window[Client]
clearTimeout: Window[Client]
CLICK: Event[Client]
click: HTMLInputElement[DOM], Input[Client]
clientX: MouseEvent[DOM]
clientY: MouseEvent[DOM]
clip: CSS2Properties[DOM]
clip.bottom: Layer[Client]
clip.height: Layer[Client]
clip.left: Layer[Client]
clip.right: Layer[Client]
clip.top: Layer[Client]
clip.width: Layer[Client]
cloneContents: Range[DOM]
cloneNode: Node[DOM]
cloneRange: Range[DOM]
close: Document[Client], HTMLDocument[DOM],
 Window[Client]
closed: Window[Client]
collapse: Range[DOM]
collapsed: Range[DOM]
color: CSS2Properties[DOM]
colorDepth: Screen[Client]
cols: HTMLTextAreaElement[DOM]
colSpan: HTMLTableCellElement[DOM]
Comment: [DOM]
COMMENT_NODE: Node[DOM]
commonAncestorContainer: Range[DOM]
compareBoundaryPoints: Range[DOM]
complete: Image[Client]
concat: Array[Core], String[Core]
confirm: Window[Client]
constructor: Object[Core]
contains: HTMLElement[Client]
content: CSS2Properties[DOM]
cookie: Document[Client],
 HTMLDocument[DOM]
cookieEnabled: Navigator[Client]
coords: HTMLAnchorElement[DOM]
cos: Math[Core]
Counter: [DOM]
counterIncrement: CSS2Properties[DOM]
counterReset: CSS2Properties[DOM]
createAttribute: Document[DOM]
createAttributeNS: Document[DOM]
createCaption: HTMLTableElement[DOM]
createCDATASection: Document[DOM]
createComment: Document[DOM]
createCSSStyleSheet:
 DOMImplementation[DOM]
createDocument: DOMImplementation[DOM]
createDocumentFragment: Document[DOM]
createDocumentType:
 DOMImplementation[DOM]
createElement: Document[DOM]
createElementNS: Document[DOM]
createEntityReference: Document[DOM]
createEvent: Document[DOM]
createHTMLDocument:
 DOMImplementation[DOM]
createNodeIterator: Document[DOM]
createProcessingInstruction: Document[DOM]
createRange: Document[DOM]

createTextNode: Document[DOM]
createTFoot: HTMLTableElement[DOM]
createTHead: HTMLTableElement[DOM]
createTreeWalker: Document[DOM]
CSS2Properties: [DOM]
CSSCharsetRule: [DOM]
cssFloat: CSS2Properties[DOM]
CSSFontFaceRule: [DOM]
CSSImportRule: [DOM]
CSSMediaRule: [DOM]
CSSPageRule: [DOM]
CSSPrimitiveValue: [DOM]
CSSRule: [DOM]
CSSRuleList: [DOM]
cssRules: CSSMediaRule[DOM],
 CSSStyleSheet[DOM]
CSSStyleDeclaration: [DOM]
CSSStyleRule: [DOM]
CSSStyleSheet: [DOM]
cssText: CSSRule[DOM],
 CSSStyleDeclaration[DOM], CSSValue[DOM]
CSSUnknownRule: [DOM]
CSSValue: [DOM]
CSSValueList: [DOM]
cssValueType: CSSValue[DOM]
CSS_ATTR: CSSPrimitiveValue[DOM]
CSS_CM: CSSPrimitiveValue[DOM]
CSS_COUNTER: CSSPrimitiveValue[DOM]
CSS_CUSTOM: CSSValue[DOM]
CSS_DEG: CSSPrimitiveValue[DOM]
CSS_DIMENSION: CSSPrimitiveValue[DOM]
CSS_EMS: CSSPrimitiveValue[DOM]
CSS_EXS: CSSPrimitiveValue[DOM]
CSS_GRAD: CSSPrimitiveValue[DOM]
CSS_HZ: CSSPrimitiveValue[DOM]
CSS_IDENT: CSSPrimitiveValue[DOM]
CSS_IN: CSSPrimitiveValue[DOM]
CSS_INHERIT: CSSValue[DOM]
CSS_KHZ: CSSPrimitiveValue[DOM]
CSS_MM: CSSPrimitiveValue[DOM]
CSS_MS: CSSPrimitiveValue[DOM]
CSS_NUMBER: CSSPrimitiveValue[DOM]
CSS_PC: CSSPrimitiveValue[DOM]
CSS_PERCENTAGE: CSSPrimitiveValue[DOM]
CSS_PRIMITIVE_VALUE: CSSValue[DOM]
CSS_PT: CSSPrimitiveValue[DOM]
CSS_PX: CSSPrimitiveValue[DOM]
CSS_RAD: CSSPrimitiveValue[DOM]
CSS_RECT: CSSPrimitiveValue[DOM]
CSS_RGBCOLOR: CSSPrimitiveValue[DOM]
CSS_S: CSSPrimitiveValue[DOM]
CSS_STRING: CSSPrimitiveValue[DOM]
CSS_UNKNOWN: CSSPrimitiveValue[DOM]
CSS_URI: CSSPrimitiveValue[DOM]
CSS_VALUE_LIST: CSSValue[DOM]
ctrlKey: MouseEvent[DOM]
cue: CSS2Properties[DOM]
cueAfter: CSS2Properties[DOM]
cueBefore: CSS2Properties[DOM]
currentNode: TreeWalker[DOM]
currentTarget: Event[DOM]
cursor: CSS2Properties[DOM]

D
data: CharacterData[DOM],
 ProcessingInstruction[DOM]

- Date:** [Core]
DBLCLICK: Event[Client]
decodeURI: [Core]
decodeURIComponent: [Core]
defaultChecked: Checkbox[Client],
 HTMLInputElement[DOM], Input[Client],
 Radio[Client]
defaultSelected: HTMLOptionElement[DOM],
 Option[Client]
defaultStatus: Window[Client]
defaultValue: HTMLInputElement[DOM],
 HTMLTextAreaElement[DOM], Input[Client]
defaultView: Document[DOM]
deleteCaption: HTMLTableElement[DOM]
deleteCell: HTMLTableRowElement[DOM]
deleteContents: Range[DOM]
deleteData: CharacterData[DOM]
deleteMedium: MediaList[DOM]
deleteRow: HTMLTableElement[DOM].
 HTMLTableSectionElement[DOM]
deleteRule: CSSMediaRule[DOM],
 ~CSSStyleSheet[DOM]
deleteTFoot: HTMLTableElement[DOM]
deleteTHead: HTMLTableElement[DOM]
description: MimeType[Client], Plugin[Client]
detach: NodeIterator[DOM], Range[DOM]
detail: UIEvent[DOM]
dir: HTMLElement[DOM]
direction: CSS2Properties[DOM]
disabled: HTMLInputElement[DOM],
 HTMLOptionElement[DOM],
 HTMLSelectElement[DOM],
 HTMLTextAreaElement[DOM],
 StyleSheet[DOM]
dispatchEvent: EventTarget[DOM]
display: CSS2Properties[DOM]
doctype: Document[DOM]
Document: [Client], [DOM]
document: AbstractView[DOM],
 HTMLElement[Client], Layer[Client],
 Window[Client]
DocumentCSS: [DOM]
documentElement: Document[DOM]
DocumentEvent: [DOM]
DocumentFragment: [DOM]
DocumentRange: [DOM]
DocumentStyle: [DOM]
DocumentTraversal: [DOM]
DocumentType: [DOM]
DocumentView: [DOM]
DOCUMENT_FRAGMENT_NODE: Node[DOM]
DOCUMENT_NODE: Node[DOM]
DOCUMENT_TYPE_NODE: Node[DOM]
domain: Document[Client],
 HTMLDocument[DOM]
DOMException: [DOM]
DOMImplementation: [DOM]
DOMImplementationCSS: [DOM]
DOMSTRING_SIZE_ERR: DOMException[DOM]
DRAGDROP: Event[Client]
- ## E
- E:** Math[Core]
Element: [Client], [DOM]
ElementCSSInlineStyle: [DOM]
elementFromPoint: Document[Client]

elements: Form[Client],
HTMLFormElement[DOM]
ELEMENT_NODE: Node[DOM]
elevation: CSS2Properties[DOM]
embeds: Document[Client]
emptyCells: CSS2Properties[DOM]
enabledPlugin: MimeType[Client]
encodeURIComponent: [Core]
encodeURIComponent: [Core]
encoding: CSSCharsetRule[DOM], Form[Client]
enctype: HTMLFormElement[DOM]
endContainer: Range[DOM]
endOffset: Range[DOM]
END_TO_END: Range[DOM]
END_TO_START: Range[DOM]
entities: DocumentType[DOM]
Entity: [DOM]
EntityReference: [DOM]
ENTITY_NODE: Node[DOM]
ENTITY_REFERENCE_NODE: Node[DOM]
Error: [Core]
ERROR: Event[Client]
escape: [Core]
eval: [Core], JSObject[Client]
EvalError: [Core]
Event: [Client], [DOM]
EventException: [DOM]
EventListener: [DOM]
eventPhase: Event[DOM]
EventTarget: [DOM]
exec: RegExp[Core]
exp: Math[Core]
expandEntityReferences: NodeIterator[DOM],

TreeWalker[DOM]
extractContents: Range[DOM]

F

fgColor: Document[Client]
filename: Plugin[Client]
FileUpload: [Client]
filter: NodeIterator[DOM], TreeWalker[DOM]
FILTER_ACCEPT: NodeFilter[DOM]
FILTER_REJECT: NodeFilter[DOM]
FILTER_SKIP: NodeFilter[DOM]
firstChild: Node[DOM], TreeWalker[DOM]
floor: Math[Core]
FOCUS: Event[Client]
focus: HTMLAnchorElement[DOM],
HTMLInputElement[DOM],
HTMLSelectElement[DOM],
HTMLTextAreaElement[DOM], Input[Client],
Window[Client]
font: CSS2Properties[DOM]
fontFamily: CSS2Properties[DOM]
fontSize: CSS2Properties[DOM]
fontSizeAdjust: CSS2Properties[DOM]
fontStretch: CSS2Properties[DOM]
fontStyle: CSS2Properties[DOM]
fontVariant: CSS2Properties[DOM]
fontWeight: CSS2Properties[DOM]
FONT_FACE_RULE: CSSRule[DOM]
Form: [Client]
form: HTMLInputElement[DOM],
HTMLOptionElement[DOM],
HTMLSelectElement[DOM],
HTMLTextAreaElement[DOM], Input[Client]

forms: Document[Client], HTMLDocument[DOM]

forward: History[Client], Window[Client]

Frame: [Client]

frame: HTMLTableElement[DOM]

frames: Window[Client]

fromCharCode: String[Core]

Function: [Core]

G

getAttribute: Element[DOM],

HTMLElement[Client]

getAttributeNode: Element[DOM]

getAttributeNodeNS: Element[DOM]

getAttributeNS: Element[DOM]

getClass: [Client]

getComputedStyle: AbstractView[DOM]

getCounterValue: CSSPrimitiveValue[DOM]

getDate: Date[Core]

getDay: Date[Core]

getElementById: Document[DOM],

HTMLDocument[DOM]

getElementsByName: HTMLDocument[DOM]

getElementsByTagName: Document[DOM],

Element[DOM]

getElementsByTagNameNS: Document[DOM],

Element[DOM]

getFloatValue: CSSPrimitiveValue[DOM]

getFullYear: Date[Core]

getHours: Date[Core]

getMember: JSObject[Client]

getMilliseconds: Date[Core]

getMinutes: Date[Core]

getMonth: Date[Core]

getNamedItem: NamedNodeMap[DOM]

getNamedItemNS: NamedNodeMap[DOM]

getOverrideStyle: Document[DOM]

getPropertyCSSValue:

CSSStyleDeclaration[DOM]

getPropertyPriority: CSSStyleDeclaration[DOM]

getPropertyValue: CSSStyleDeclaration[DOM]

getRectValue: CSSPrimitiveValue[DOM]

getRGBColorValue: CSSPrimitiveValue[DOM]

getSeconds: Date[Core]

getSelection: Document[Client]

getSlot: JSObject[Client]

getStringValue: CSSPrimitiveValue[DOM]

getTime: Date[Core]

getTimezoneOffset: Date[Core]

getUTCDate: Date[Core]

getUTCDay: Date[Core]

getUTCFullYear: Date[Core]

getUTCHours: Date[Core]

getUTCMilliseconds: Date[Core]

getUTCMinutes: Date[Core]

getUTCMonth: Date[Core]

getUTCSeconds: Date[Core]

getWindow: JSObject[Client]

getYear: Date[Core]

Global: [Core]

global: RegExp[Core]

go: History[Client]

green: RGBColor[DOM]

H

handleEvent: Document[Client],

HTMLElement[Client], Layer[Client],

Window[Client]
hasAttribute: Element[DOM]
hasAttributeNS: Element[DOM]
hasAttributes: Node[DOM]
hasChildNodes: Node[DOM]
hasFeature: DOMImplementation[DOM]
hash: Link[Client], Location[Client]
hasOwnProperty: Object[Core]
headers: HTMLTableCellElement[DOM]
height: CSS2Properties[DOM],
 HTMLTableCellElement[DOM], Image[Client],
 Screen[Client]
Hidden: [Client]
hidden: Layer[Client]
HIERARCHY_REQUEST_ERR:
 DOMException[DOM]
History: [Client]
history: Window[Client]
home: Window[Client]
host: Link[Client], Location[Client]
hostname: Link[Client], Location[Client]
href: CSSImportRule[DOM],
 HTMLAnchorElement[DOM], Link[Client],
 Location[Client],
 StyleSheet[DOM]
hreflang: HTMLAnchorElement[DOM]
hspace: Image[Client]
HTMLAnchorElement: [DOM]
HTMLBodyElement: [DOM]
HTMLCollection: [DOM]
HTMLDocument: [DOM]
HTMLDOMImplementation: [DOM]
HTMLElement: [Client], [DOM]

HTMLFormElement: [DOM]
HTMLInputElement: [DOM]
HTMLOptionElement: [DOM]
HTMLSelectElement: [DOM]
HTMLTableCaptionElement: [DOM]
HTMLTableCellElement: [DOM]
HTMLTableColElement: [DOM]
HTMLTableElement: [DOM]
HTMLTableRowElement: [DOM]
HTMLTableSectionElement: [DOM]
HTMLTextAreaElement: [DOM]

I

id: HTMLElement[Client], HTMLElement[DOM]
identifier: Counter[DOM]
ignoreCase: RegExp[Core]
Image: [Client]
images: Document[Client],
 HTMLDocument[DOM]
implementation: Document[DOM]
importNode: Document[DOM]
IMPORT_RULE: CSSRule[DOM]
index: HTMLOptionElement[DOM], Option[Client]
indexOf: String[Core]
INDEX_SIZE_ERR: DOMException[DOM]
Infinity: [Core]
initEvent: Event[DOM]
initMouseEvent: MouseEvent[DOM]
initMutationEvent: MutationEvent[DOM]
initUIEvent: UIEvent[DOM]
innerHTML: HTMLElement[Client]
innerText: HTMLElement[Client]
Input: [Client]

insertAdjacentHTML: HTMLElement[Client]
insertAdjacentText: HTMLElement[Client]
insertBefore: Node[DOM]
insertCell: HTMLTableRowElement[DOM]
insertData: CharacterData[DOM]
insertNode: Range[DOM]
insertRow: HTMLTableElement[DOM],
 HTMLTableSectionElement[DOM]
insertRule: CSSMediaRule[DOM],
 CSSStyleSheet[DOM]
internalSubset: DocumentType[DOM]
INUSE_ATTRIBUTE_ERR: DOMException[DOM]
INVALID_ACCESS_ERR: DOMException[DOM]
INVALID_CHARACTER_ERR:
 DOMException[DOM]
INVALID_MODIFICATION_ERR:
 DOMException[DOM]
INVALID_NODE_TYPE_ERR:
 RangeException[DOM]
INVALID_STATE_ERR: DOMException[DOM]
isFinite: [Core]
isNaN: [Core]
isPrototypeOf: Object[Core]
isSupported: Node[DOM]
item: CSSRuleList[DOM],
 CSSStyleDeclaration[DOM],
 CSSValueList[DOM], HTMLCollection[DOM],
 MediaList[DOM], NamedNodeMap[DOM],
 NodeList[DOM], StyleSheetList[DOM]

J

JavaArray: [Client]
JavaClass: [Client]

javaEnabled: Navigator[Client]
JavaObject: [Client]
JavaPackage: [Client]
join: Array[Core]
JSObject: [Client]

K

KEYDOWN: Event[Client]
KEYPRESS: Event[Client]
KEYUP: Event[Client]

L

label: HTMLOptionElement[DOM]
lang: HTMLElement[Client], HTMLElement[DOM]
language: Navigator[Client]
lastChild: Node[DOM], TreeWalker[DOM]
lastIndex: RegExp[Core]
lastIndexOf: String[Core]
lastModified: Document[Client]
Layer: [Client]
layers: Layer[Client]
left: CSS2Properties[DOM], Layer[Client],
 Rect[DOM]
length: Arguments[Core], Array[Core],
 CharacterData[DOM], CSSRuleList[DOM],
 CSSStyleDeclaration[DOM],
 CSSValueList[DOM], Form[Client],
 Function[Core],
 History[Client], HTMLCollection[DOM],
 HTMLFormElement[DOM],
 HTMLSelectElement[DOM], Input[Client],
 JavaArray[Client], MediaList[DOM],
 NamedNodeMap[DOM], NodeList[DOM],

Plugin[Client], Select[Client], String[Core],
StyleSheetList[DOM], Window[Client]
letterSpacing: CSS2Properties[DOM]
lineHeight: CSS2Properties[DOM]
Link: [Client]
link: HTMLBodyElement[DOM]
linkColor: Document[Client]
links: Document[Client], HTMLDocument[DOM]
LinkStyle: [DOM]
listStyle: Counter[DOM], CSS2Properties[DOM]
listStyleImage: CSS2Properties[DOM]
listStylePosition: CSS2Properties[DOM]
listStyleType: CSS2Properties[DOM]
LN10: Math[Core]
LN2: Math[Core]
LOAD: Event[Client]
load: Layer[Client]
localeCompare: String[Core]
localName: Node[DOM]
Location: [Client]
location: Document[Client], Window[Client]
log: Math[Core]
LOG10E: Math[Core]
LOG2E: Math[Core]
lowsrc: Image[Client]

M

margin: CSS2Properties[DOM]
marginBottom: CSS2Properties[DOM]
marginLeft: CSS2Properties[DOM]
marginRight: CSS2Properties[DOM]
marginTop: CSS2Properties[DOM]
markerOffset: CSS2Properties[DOM]

marks: CSS2Properties[DOM]
match: String[Core]
Math: [Core], Window[Client]
max: Math[Core]
MAX_VALUE: Number[Core]
maxHeight: CSS2Properties[DOM]
maxLength: HTMLInputElement[DOM]
maxWidth: CSS2Properties[DOM]
media: CSSImportRule[DOM],
CSSMediaRule[DOM], StyleSheet[DOM]
MediaList: [DOM]
mediaText: MediaList[DOM]
MEDIA_RULE: CSSRule[DOM]
message: Error[Core]
metaKey: MouseEvent[DOM]
method: Form[Client], HTMLFormElement[DOM]
MimeType: [Client]
mimeType: Navigator[Client]
min: Math[Core]
MIN_VALUE: Number[Core]
minHeight: CSS2Properties[DOM]
minWidth: CSS2Properties[DOM]
MODIFICATION: MutationEvent[DOM]
MOUSEDOWN: Event[Client]
MouseEvent: [DOM]
MOUSEMOVE: Event[Client]
MOUSEOUT: Event[Client]
MOUSEOVER: Event[Client]
MOUSEUP: Event[Client]
MOVE: Event[Client]
moveAbove: Layer[Client]
moveBelow: Layer[Client]
moveBy: Layer[Client], Window[Client]

moveTo: Layer[Client], Window[Client]**moveToAbsolute:** Layer[Client]**multiple:** HTMLSelectElement[DOM]**MutationEvent:** [DOM]

N

name: Anchor[Client], Attr[DOM],
 DocumentType[DOM], Error[Core],
 Form[Client],
 HTMLAnchorElement[DOM],
 HTMLFormElement[DOM],
 HTMLInputElement[DOM],
 HTMLSelectElement[DOM],
 HTMLTextAreaElement[DOM], Image[Client],
 Input[Client], Layer[Client], Plugin[Client],
 Window[Client]

namedItem: HTMLCollection[DOM]**NamedNodeMap:** [DOM]**namespaceURI:** Node[DOM]**NAMESPACE_ERR:** DOMException[DOM]**NaN:** [Core], Number[Core]**navigate:** Window[Client]**Navigator:** [Client]**navigator:** Window[Client]**NEGATIVE_INFINITY:** Number[Core]**newValue:** MutationEvent[DOM]**nextNode:** NodeIterator[DOM], TreeWalker[DOM]**nextSibling:** Node[DOM], TreeWalker[DOM]**Node:** [DOM]**NodeFilter:** [DOM]**NodeIterator:** [DOM]**NodeList:** [DOM]**nodeName:** Node[DOM]**nodeType:** Node[DOM]**nodeValue:** Node[DOM]**normalize:** Node[DOM]**Notation:** [DOM]**notationName:** Entity[DOM]**notations:** DocumentType[DOM]**NOTATION_NODE:** Node[DOM]**NOT_FOUND_ERR:** DOMException[DOM]**NOT_SUPPORTED_ERR:** DOMException[DOM]**noWrap:** HTMLTableCellElement[DOM]**NO_DATA_ALLOWED_ERR:**

DOMException[DOM]

NO_MODIFICATION_ALLOWED_ERR:

DOMException[DOM]

Number: [Core]

O

Object: [Core]**offset:** Layer[Client]**offsetHeight:** HTMLElement[Client]**offsetLeft:** HTMLElement[Client]**offsetParent:** HTMLElement[Client]**offsetTop:** HTMLElement[Client]**offsetWidth:** HTMLElement[Client]**onabort:** Image[Client]**onblur:** Input[Client], Window[Client]

onchange: FileUpload[Client], Input[Client],
 Select[Client], Textarea[Client], Text[Client]

onclick: Button[Client], Checkbox[Client],
 HTMLElement[Client], Input[Client],
 Link[Client], Radio[Client], Reset[Client],
 Submit[Client]

ondblclick: HTMLElement[Client]

onerror: Image[Client], Window[Client]
onfocus: Input[Client], Window[Client]
onhelp: HTMLElement[Client]
onkeydown: HTMLElement[Client]
onkeypress: HTMLElement[Client]
onkeyup: HTMLElement[Client]
onload: Image[Client], Window[Client]
onmousedown: HTMLElement[Client]
onmousemove: HTMLElement[Client]
onmouseout: HTMLElement[Client], Link[Client]
onmouseover: HTMLElement[Client], Link[Client]
onmouseup: HTMLElement[Client]
onmove: Window[Client]
onreset: Form[Client]
onresize: Window[Client]
onsubmit: Form[Client]
onunload: Window[Client]
open: Document[Client], HTMLDocument[DOM],
Window[Client]
opener: Window[Client]
Option: [Client]
options: HTMLSelectElement[DOM],
Input[Client], Select[Client]
orphans: CSS2Properties[DOM]
outerHTML: HTMLElement[Client]
outerText: HTMLElement[Client]
outline: CSS2Properties[DOM]
outlineColor: CSS2Properties[DOM]
outlineStyle: CSS2Properties[DOM]
outlineWidth: CSS2Properties[DOM]
overflow: CSS2Properties[DOM]
ownerDocument: Node[DOM]
ownerElement: Attr[DOM]

ownerNode: StyleSheet[DOM]
ownerRule: CSSStyleSheet[DOM]

P

padding: CSS2Properties[DOM]
paddingBottom: CSS2Properties[DOM]
paddingLeft: CSS2Properties[DOM]
paddingRight: CSS2Properties[DOM]
paddingTop: CSS2Properties[DOM]
page: CSS2Properties[DOM]
pageBreakAfter: CSS2Properties[DOM]
pageBreakBefore: CSS2Properties[DOM]
pageBreakInside: CSS2Properties[DOM]
pageX: Layer[Client]
pageY: Layer[Client]
PAGE_RULE: CSSRule[DOM]
parent: Window[Client]
parentElement: HTMLElement[Client]
parentLayer: Layer[Client]
parentNode: Node[DOM], TreeWalker[DOM]
parentRule: CSSRule[DOM],
CSSStyleDeclaration[DOM]
parentStyleSheet: CSSRule[DOM],
StyleSheet[DOM]
parse: Date[Core]
parseFloat: [Core]
parseInt: [Core]
Password: [Client]
pathname: Link[Client], Location[Client]
pause: CSS2Properties[DOM]
pauseAfter: CSS2Properties[DOM]
pauseBefore: CSS2Properties[DOM]
PI: Math[Core]

pitch: CSS2Properties[DOM]
pitchRange: CSS2Properties[DOM]
pixelDepth: Screen[Client]
platform: Navigator[Client]
playDuring: CSS2Properties[DOM]
Plugin: [Client]
plugins: Document[Client], Navigator[Client]
plugins.refresh: Navigator[Client]
pop: Array[Core]
port: Link[Client], Location[Client]
position: CSS2Properties[DOM]
POSITIVE_INFINITY: Number[Core]
pow: Math[Core]
prefix: Node[DOM]
preventDefault: Event[DOM]
previousNode: NodeIterator[DOM],
TreeWalker[DOM]
previousSibling: Node[DOM], TreeWalker[DOM]
prevValue: MutationEvent[DOM]
primitiveType: CSSPrimitiveValue[DOM]
print: Window[Client]
ProcessingInstruction: [DOM]
PROCESSING_INSTRUCTION_NODE:
Node[DOM]
prompt: Window[Client]
propertyIsEnumerable: Object[Core]
protocol: Link[Client], Location[Client]
prototype: Function[Core]
publicId: DocumentType[DOM], Entity[DOM],
Notation[DOM]
push: Array[Core]

Q

quotes: CSS2Properties[DOM]

R

Radio: [Client]
random: Math[Core]
Range: [DOM]
RangeError: [Core]
RangeException: [DOM]
readOnly: HTMLInputElement[DOM],
HTMLTextAreaElement[DOM]
Rect: [DOM]
red: RGBColor[DOM]
ReferenceError: [Core]
referrer: Document[Client],
HTMLDocument[DOM]
RegExp: [Core]
rel: HTMLAnchorElement[DOM]
relatedNode: MutationEvent[DOM]
relatedTarget: MouseEvent[DOM]
releaseEvents: Document[Client], Layer[Client],
Window[Client]
reload: Location[Client]
REMOVAL: MutationEvent[DOM]
remove: HTMLSelectElement[DOM]
removeAttribute: Element[DOM],
HTMLElement[Client]
removeAttributeNode: Element[DOM]
removeAttributeNS: Element[DOM]
removeChild: Node[DOM]

removeEventListener: EventTarget[DOM]
removeMember: JavaScript[Client]
removeNamedItem: NamedNodeMap[DOM]
removeNamedItemNS: NamedNodeMap[DOM]
removeProperty: CSSStyleDeclaration[DOM]
replace: Location[Client], String[Core]
replaceChild: Node[DOM]
replaceData: CharacterData[DOM]
Reset: [Client]
RESET: Event[Client]
reset: Form[Client], HTMLFormElement[DOM]
RESIZE: Event[Client]
resizeBy: Layer[Client], Window[Client]
resizeTo: Layer[Client], Window[Client]
rev: HTMLAnchorElement[DOM]
reverse: Array[Core]
RGBColor: [DOM]
richness: CSS2Properties[DOM]
right: CSS2Properties[DOM], Rect[DOM]
root: NodeIterator[DOM], TreeWalker[DOM]
round: Math[Core]
routeEvent: Document[Client], Layer[Client],
 Window[Client]
rowIndex: HTMLTableRowElement[DOM]
rows: HTMLTableElement[DOM],
 HTMLTableSectionElement[DOM],
 HTMLTextAreaElement[DOM]
rowSpan: HTMLTableCellElement[DOM]
rules: HTMLTableElement[DOM]

S

scope: HTMLTableCellElement[DOM]
Screen: [Client]

screen: Window[Client]
screenX: MouseEvent[DOM]
screenY: MouseEvent[DOM]
scroll: Window[Client]
scrollBy: Window[Client]
scrollIntoView: HTMLElement[Client]
scrollTo: Window[Client]
search: Link[Client], Location[Client],
 String[Core]
sectionRowIndex: HTMLTableRowElement[DOM]
Select: [Client]
SELECT: Event[Client]
select: HTMLInputElement[DOM],
 HTMLTextAreaElement[DOM], Input[Client]
selected: HTMLOptionElement[DOM],
 Option[Client]
selectedIndex: HTMLSelectElement[DOM],
 Input[Client], Select[Client]
selectNode: Range[DOM]
selectNodeContents: Range[DOM]
selectorText: CSSPageRule[DOM],
 CSSStyleRule[DOM]
self: Window[Client]
separator: Counter[DOM]
setAttribute: Element[DOM],
 HTMLElement[Client]
setAttributeNode: Element[DOM]
setAttributeNodeNS: Element[DOM]
setAttributeNS: Element[DOM]
setDate: Date[Core]
setEnd: Range[DOM]
setEndAfter: Range[DOM]
setEndBefore: Range[DOM]

setFloatValue: CSSPrimitiveValue[DOM]
setFullYear: Date[Core]
setHours: Date[Core]
setInterval: Window[Client]
setMember: JSObject[Client]
setMilliseconds: Date[Core]
setMinutes: Date[Core]
setMonth: Date[Core]
setNamedItem: NamedNodeMap[DOM]
setNamedItemNS: NamedNodeMap[DOM]
setProperty: CSSStyleDeclaration[DOM]
setSeconds: Date[Core]
setSlot: JSObject[Client]
setStart: Range[DOM]
setStartAfter: Range[DOM]
setStartBefore: Range[DOM]
setStringValue: CSSPrimitiveValue[DOM]
setTime: Date[Core]
setTimeout: Window[Client]
setUTCDate: Date[Core]
setUTCFullYear: Date[Core]
setUTCHours: Date[Core]
setUTCMilliseconds: Date[Core]
setUTCMinutes: Date[Core]
setUTCMonth: Date[Core]
setUTCSeconds: Date[Core]
setYear: Date[Core]
shape: HTMLAnchorElement[DOM]
sheet: LinkStyle[DOM]
shift: Array[Core]
shiftKey: MouseEvent[DOM]
SHOW_ALL: NodeFilter[DOM]
SHOW_ATTRIBUTE: NodeFilter[DOM]

SHOW_CDATA_SECTION: NodeFilter[DOM]
SHOW_COMMENT: NodeFilter[DOM]
SHOW_DOCUMENT: NodeFilter[DOM]
SHOW_DOCUMENT_FRAGMENT:
 NodeFilter[DOM]
SHOW_DOCUMENT_TYPE: NodeFilter[DOM]
SHOW_ELEMENT: NodeFilter[DOM]
SHOW_ENTITY: NodeFilter[DOM]
SHOW_ENTITY_REFERENCE: NodeFilter[DOM]
SHOW_NOTATION: NodeFilter[DOM]
SHOW_PROCESSING_INSTRUCTION:
 NodeFilter[DOM]
SHOW_TEXT: NodeFilter[DOM]
siblingAbove: Layer[Client]
siblingBelow: Layer[Client]
sin: Math[Core]
size: CSS2Properties[DOM],
 HTMLInputElement[DOM],
 HTMLSelectElement[DOM]
slice: Array[Core], String[Core]
sort: Array[Core]
source: RegExp[Core]
sourceIndex: HTMLElement[Client]
span: HTMLTableColElement[DOM]
speak: CSS2Properties[DOM]
speakHeader: CSS2Properties[DOM]
speakNumeral: CSS2Properties[DOM]
speakPunctuation: CSS2Properties[DOM]
specified: Attr[DOM]
speechRate: CSS2Properties[DOM]
splice: Array[Core]
split: String[Core]
splitText: Text[DOM]

sqrt: Math[Core]
SQRT1_2: Math[Core]
SQRT2: Math[Core]
src: HTMLInputElement[DOM], Image[Client],
Layer[Client]
startContainer: Range[DOM]
startOffset: Range[DOM]
START_TO_END: Range[DOM]
START_TO_START: Range[DOM]
status: Window[Client]
stop: Window[Client]
stopPropagation: Event[DOM]
stress: CSS2Properties[DOM]
String: [Core]
Style: [Client]
style: CSSFontFaceRule[DOM],
CSSPageRule[DOM], CSSStyleRule[DOM],
HTMLElement[Client], HTMLElement[DOM]
styleSheet: CSSImportRule[DOM]
StyleSheet: [DOM]
StyleSheetList: [DOM]
styleSheets: Document[DOM]
STYLE_RULE: CSSRule[DOM]
Submit: [Client]
SUBMIT: Event[Client]
submit: Form[Client], HTMLFormElement[DOM]
substr: String[Core]
substring: String[Core]
substringData: CharacterData[DOM]
suffixes: MimeType[Client]
summary: HTMLTableElement[DOM]
surroundContents: Range[DOM]
SyntaxError: [Core]

SYNTAX_ERR: DOMException[DOM]
systemId: DocumentType[DOM], Entity[DOM],
Notation[DOM]
systemLanguage: Navigator[Client]

T

tabIndex: HTMLAnchorElement[DOM],
HTMLInputElement[DOM],
HTMLSelectElement[DOM],
HTMLTextAreaElement[DOM]
tableLayout: CSS2Properties[DOM]
tagName: Element[DOM], HTMLElement[Client]
tan: Math[Core]
target: Event[DOM], Form[Client],
HTMLAnchorElement[DOM],
HTMLFormElement[DOM],
Link[Client], ProcessingInstruction[DOM]
tBodies: HTMLTableElement[DOM]
test: RegExp[Core]
Text: [Client], [DOM]
text: Anchor[Client], HTMLBodyElement[DOM],
HTMLOptionElement[DOM], Link[Client],
Option[Client]
textAlign: CSS2Properties[DOM]
Textarea: [Client]
textDecoration: CSS2Properties[DOM]
textIndent: CSS2Properties[DOM]
textShadow: CSS2Properties[DOM]
textTransform: CSS2Properties[DOM]
TEXT_NODE: Node[DOM]
tFoot: HTMLTableElement[DOM]
tHead: HTMLTableElement[DOM]
timeStamp: Event[DOM]

title: Document[Client], HTMLDocument[DOM],
 HTMLElement[Client],
 HTMLElement[DOM], StyleSheet[DOM]
toDateString: Date[Core]
toExponential: Number[Core]
toFixed: Number[Core]
toGMTString: Date[Core]
toLocaleDateString: Date[Core]
toLocaleLowerCase: String[Core]
toLocaleString: Array[Core], Date[Core],
 Number[Core], Object[Core]
toLocaleTimeString: Date[Core]
toLocaleUpperCase: String[Core]
toLowerCase: String[Core]
top: CSS2Properties[DOM], Layer[Client],
 Rect[DOM], Window[Client]
toPrecision: Number[Core]
toString: Array[Core], Boolean[Core], Date[Core],
 Error[Core], Function[Core],
 JSONObject[Client], Number[Core],
 Object[Core], Range[DOM], RegExp[Core],
 String[Core]
toTimeString: Date[Core]
toUpperCase: String[Core]
toUTCString: Date[Core]
TreeWalker: [DOM]
type: CSSRule[DOM], Event[DOM],
 HTMLAnchorElement[DOM],
 HTMLInputElement[DOM],
 HTMLSelectElement[DOM],
 HTMLTextAreaElement[DOM], Input[Client],
 MimeType[Client], Select[Client],
 StyleSheet[DOM]

TypeError: [Core]

U

UIEvent: [DOM]
undefined: [Core]
unescape: [Core]
unicodeBidi: CSS2Properties[DOM]
UNKNOWN_RULE: CSSRule[DOM]
UNLOAD: Event[Client]
unshift: Array[Core]
URIError: [Core]
URL: [Client], Document[Client],
 HTMLDocument[DOM]
useMap: HTMLInputElement[DOM]
userAgent: Navigator[Client]
userLanguage: Navigator[Client]
UTC: Date[Core]

V

vAlign: HTMLTableCellElement[DOM],
 HTMLTableColElement[DOM],
 HTMLTableRowElement[DOM],
 HTMLTableSectionElement[DOM]
value: Attr[DOM], Button[Client],
 Checkbox[Client], FileUpload[Client],
 Hidden[Client],
 HTMLInputElement[DOM],
 HTMLOptionElement[DOM],
 HTMLSelectElement[DOM],
 HTMLTextAreaElement[DOM], Input[Client],
 Option[Client], Password[Client],
 Radio[Client], Reset[Client], Submit[Client],
 Text[Client], Textarea[Client]

valueOf: Boolean[Core], Date[Core],
Number[Core], Object[Core], String[Core]

verticalAlign: CSS2Properties[DOM]

view: UIEvent[DOM]

ViewCSS: [DOM]

visibility: CSS2Properties[DOM], Layer[Client]

vLink: HTMLBodyElement[DOM]

vlinkColor: Document[Client]

voiceFamily: CSS2Properties[DOM]

volume: CSS2Properties[DOM]

vspace: Image[Client]

W

whatToShow: NodeIterator[DOM],

TreeWalker[DOM]

whiteSpace: CSS2Properties[DOM]

widows: CSS2Properties[DOM]

width: CSS2Properties[DOM],

HTMLTableCellElement[DOM],

HTMLTableColElement[DOM],

HTMLTableElement[DOM], Image[Client],

Screen[Client]

Window: [Client]

window: Layer[Client], Window[Client]

wordSpacing: CSS2Properties[DOM]

write: Document[Client], HTMLDocument[DOM]

writeln: Document[Client],

HTMLDocument[DOM]

WRONG_DOCUMENT_ERR:

DOMException[DOM]

X

x: Layer[Client]

Y

y: Layer[Client]

Z

zIndex: CSS2Properties[DOM], Layer[Client]

词汇表

Anchor Element

锚元素

Animation

动画

Argument

参数 (实际参数)

Array

数组

Assignment

赋值

Associative Array

关联数组

Associativity

结合性

Attribute

性质

Background Color

背景颜色

Behavior

行为

Block

数据块

Browser

浏览器

Bubbling

起泡

Class

类

Constructor Function

构造函数

CSS

级联样式表

Cyclical Reference

循环引用

Data Type

数据类型

Debug

调试

Declare

声明

Decrement	Identity
递减	等同
DOM	Implementation
文档对象模型	实现
Element	Increment
元素	递增
Embed	Index
嵌入	下标
Equality	Input Event
相等	输入事件
Event Handler	Integer
事件处理器	整型
Event	Interpreter
事件	解释器
Exception	Iteration
异常	迭代
Execution Context	Key
执行环境	键
Feature	Keyword
特性	关键字
Field	Label
字段	标签
Flag	Layer
标志	层
Form	Local Variable
表单	局部变量
Frame	Local-specific
框架	地区特定
Garbage Collection	Loop
无用存储单元收集	循环
Global Variable	Markup
全局变量	标记
Identifier	Method
标识符	方法

Multiple Thread	Reference
多线程	引用
Object	Regular Expression
对象	正则表达式
Operand	Same-origin Policy
操作数	同源策略
Operator	Scope
运算符	作用域
Parameter	Script Language
参数 (形式参数)	脚本语言
Pattern	Statement
模式	语句
Precedence	String Literals
优先级	字符串直接量
Property	Synthesizing Event
属性	合成事件
Protocol	Tag
协议	标记
Prototype	Target
原型	目标
Range	Variable Name Resolution
范围	变量名解析
Raw Event	Variable
原始事件	变量

Multiple Thread	Reference
多线程	引用
Object	Regular Expression
对象	正则表达式
Operand	Same-origin Policy
操作数	同源策略
Operator	Scope
运算符	作用域
Parameter	Script Language
参数 (形式参数)	脚本语言
Pattern	Statement
模式	语句
Precedence	String Literals
优先级	字符串直接量
Property	Synthesizing Event
属性	合成事件
Protocol	Tag
协议	标记
Prototype	Target
原型	目标
Range	Variable Name Resolution
范围	变量名解析
Raw Event	Variable
原始事件	变量