

ActionScript: The Definitive Guide

Gary Grossman  
(ActionScript 权威指南)  
作者

# ActionScript

权威指南



O'REILLY®

机械工业出版社

China Machine Press



Colin Moock 著

赵声攀 等译

# ActionScript 权威指南



Macromedia Flash 是全球 25 000 万用户在 Web 上发布多媒体信息的实际标准。本书是关于 ActionScript (Flash 的面向对象编程语言) 的完整而深入说明, 它并不仅仅要给 Web 开发者提供创建高级 web 站点的基础工具。其目标读者既包括初来乍到的 Flash 开发者, 又包括那些要将技术转移到 ActionScript 上的 JavaScript 程序员 (两种语言都是以 ECMAScript 标准为基础的)。

**Colin Moock** 将全部的实际知识和恰当的表达方式结合起来的能力, 使他在 Flash 界广受赞誉。本书的写作既表现了他清晰的叙述性语调, 又没有自负和谦虚的痕迹。本书第一部分“ActionScript 基础”, 描述了核心编程概念 (变量、数据类型、操作符、语句、函数、事件、数组和对象) 以及它们的详细用法, 并特别介绍了影片剪辑。第二部分“ActionScript 应用”, 包括了普通的应用程序, 比如处理在线表单。第三部分“语言参考”详细列举了 ActionScript 的全局函数、属性、方法、事件处理器和对象, 并包括广泛的应用示例。

本书可以让新的 ActionScript 程序员获得迅速的进步。它用传统的形式来说明 ActionScript, 给读者打下坚实的理论基础。富有经验的程序员可以在学习 Flash 的复杂部分时利用他们的 JavaScript 知识。在理论之上, 本书还包含了很多实际技巧和现实的例子, 包括了滚动文本域、菜单按钮、多项选择测试、XML 驱动的站点、物理视频游戏和实际的多用户环境等等。本书还谨慎地讨论了一些未证明或者正在证明的主题。详细的语言参考可以说是很有价值的日常伙伴。广受欢迎的站点——ActionScript 代码库 (<http://www.moock.org/asdg>), 包括了所有的源.fla 文件和很多附加的示例, 还有一个用 Java 编写的 XMLSocket 服务器范例和一个 Perl 文本数据库范例。

“这是我所看到的最好的 ActionScript 书籍。你将很难在其他地方找到如此之多的 ActionScript 信息。”

—— Slavik Lozben, Macromedia, Flash 5 的主工程师

“Moock 已经撰写了 Flash 脚本编写方面惟一的、真正的指南。你在其他任何地方都不能再找到如此详尽、实际而精确的指南了。”

—— Jeremy Clark, Macromedia Flash 产品经理

ISBN 7-111-11085-4



9 787111 110859 >

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

ISBN 7-111-11085-4

定价：88.00 元

---

# ActionScript 权威指南

*Colin Moock* 著

赵声攀 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

ActionScript 权威指南 / (美) 莫克 (Moock, C.) 著; 赵声攀等译. - 北京: 机械工业出版社, 2003.1

书名原文: ActionScript: The Definitive Guide

ISBN 7-111-11085-4

I. A... II. ①莫 ... ②赵 ... III. 动画 - 设计 - 图形软件, ActionScript IV. TP391.41

中国版本图书馆 CIP 数据核字 (2002) 第 080941 号

北京市版权局著作权合同登记

图字: 01-2002-1832 号

©2001 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, 2001 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 2001。

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 / ActionScript 权威指南

书 号 / ISBN 7-111-11085-4

责任编辑 / 李欣

封面设计 / Ellie Volckhausen、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 / 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 48 印张 714 千字

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

印 数 / 0001-4000 册

定 价 / 88.00 元

(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和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。

# 目录

序言 ..... 1

前言 ..... 5

## 第一部分 ActionScript 基础

第一章 针对非程序员的简单介绍 ..... 17

一些基础习语 ..... 19

更为深入的 ActionScript 概念 ..... 28

创建多项选择测试 ..... 36

小结 ..... 51

第二章 变量 ..... 52

创建变量（声明） ..... 53

变量赋值 ..... 55

变量值的修改和获取 ..... 56

值的类型 ..... 58

变量作用域 .....	60
应用举例 .....	71
小结 .....	73
<b>第三章 数据和数据类型 .....</b>	<b>74</b>
数据和信息 .....	74
用数据类型来保持数据的意义 .....	75
数据的创建和分类 .....	76
数据类型转换 .....	78
原始数据和复合数据 .....	87
小结 .....	89
<b>第四章 原始数据类型 .....</b>	<b>90</b>
数字类型 .....	90
整数和浮点数字 .....	90
数值直接量 .....	91
数字处理 .....	95
串类型 .....	96
串的处理 .....	100
布尔类型 .....	120
undefined .....	122
null .....	123
小结 .....	123
<b>第五章 操作符 .....</b>	<b>124</b>
操作符的一般特点 .....	124
赋值操作符 .....	129
算术操作符 .....	130
等于和不等操作符 .....	135
比较操作符 .....	140

---

串操作符 .....	144
逻辑操作符 .....	144
组合操作符 .....	150
逗号操作符 .....	150
空(void)操作符 .....	151
其他操作符 .....	151
小结 .....	155

## 第六章 语句 ..... 156

语句的类型 .....	156
语句语法 .....	157
ActionScript 语句 .....	159
语句和动作 .....	167
小结 .....	168

## 第七章 条件语句 ..... 169

if 语句 .....	170
else 语句 .....	172
else if 语句 .....	173
模拟 switch 语句 .....	175
简化的条件语句语法 .....	176
小结 .....	177

## 第八章 循环语句 ..... 178

while 循环 .....	178
循环术语 .....	182
do-while 循环 .....	183
for 循环 .....	184
for-in 循环 .....	186
提前终止循环 .....	187

时间线和剪辑事件循环 .....	190
小结 .....	197
<b>第九章 函数 .....</b>	<b>198</b>
函数的创建 .....	199
函数的运行 .....	200
向函数传递参数 .....	201
退出函数并返回值 .....	204
函数直接量 .....	207
函数的可用性和生命周期 .....	207
函数的作用域 .....	209
再论函数参数 .....	213
递归函数 .....	217
内部函数 .....	219
函数对象 .....	220
代码的集中 .....	221
再看多项选择测试 .....	222
小结 .....	226
<b>第十章 事件和事件处理器 .....</b>	<b>227</b>
同步代码的执行 .....	227
基于事件的异步代码执行 .....	228
事件的类型 .....	228
事件处理器 .....	229
事件处理器语法 .....	230
创建事件处理器 .....	230
事件处理器作用域 .....	234
按钮事件 .....	238
影片剪辑事件综述 .....	242
针对影片播放的影片剪辑事件 .....	243

针对用户输入的影片剪辑事件 .....	250
执行的顺序 .....	256
复制剪辑事件处理器 .....	258
用 updateAfterEvent 更新屏幕 .....	259
代码的重复使用性 .....	260
动态的影片剪辑事件处理器 .....	260
事件处理器应用 .....	261
小结 .....	263
<b>第十一章 数组 .....</b>	<b>264</b>
什么是数组 .....	264
数组的分析 .....	265
数组的创建 .....	267
引用数组元素 .....	269
确定数组的大小 .....	271
命名数组元素 .....	273
向数组添加元素 .....	274
删除数组中的元素 .....	280
通用数组操作工具 .....	283
多维数组 .....	289
多项选择测试的第三版本 .....	290
小结 .....	291
<b>第十二章 对象和类 .....</b>	<b>292</b>
对象的分析 .....	294
实例化对象 .....	295
对象属性 .....	296
方法 .....	298
类和面向对象的编程 .....	302

---

内置 ActionScript 类和对象 .....	318
小结 .....	320
<b>第十三章 影片剪辑.....</b>	<b>321</b>
影片剪辑的对象性 .....	322
影片剪辑的类型 .....	323
创建影片剪辑.....	326
影片和实例的堆栈顺序 .....	333
实例和主影片的引用 .....	339
删除剪辑实例和主影片 .....	353
内置影片剪辑属性 .....	356
影片剪辑方法 .....	357
影片剪辑应用举例 .....	362
最后的测试 .....	366
小结 .....	369
<b>第十四章 词法结构.....</b>	<b>370</b>
空 白 .....	370
语句终结符（分号）.....	372
注 释 .....	373
保 留 字 .....	375
标 识 符 .....	376
大 小 写 区 分 .....	377
小 结 .....	378
<b>第十五章 高级主题.....</b>	<b>379</b>
复 制、比 较 和 传 递 数据 .....	379
位 逻 辑 编 程 .....	382
高 级 函 数 作 用 域 问 题 .....	394

影片剪辑数据类型 .....	396
小结 .....	398

## 第二部分 ActionScript 应用

### 第十六章 ActionScript 制作环境 ..... 401

动作面板 .....	401
为帧添加脚本 .....	404
对按钮添加脚本 .....	405
为影片剪辑添加脚本 .....	406
代码都在哪里 .....	407
生产力 .....	408
外在化 ActionScript 代码 .....	409
组件打包成智能剪辑 .....	411
小结 .....	418

### 第十七章 Flash 表单 ..... 419

Flash 表单数据循环 .....	419
创建 Flash 填充表单 .....	422
小结 .....	428

### 第十八章 屏幕文本域 ..... 430

动态文本域 .....	430
用户输入文本域 .....	432
文本域选项 .....	433
文本域属性 .....	437
HTML 支持 .....	440
关于文本域选择 .....	449
空文本域和 for-in 语句 .....	449
小结 .....	450

<b>第十九章 调试 .....</b>	<b>451</b>
调试工具 .....	452
调试方法 .....	457
小结 .....	462

## 第三部分 语言参考

<b>ActionScript 语言参考 .....</b>	<b>467</b>
--------------------------------	------------

## 第四部分 附录

<b>附录一 资源 .....</b>	<b>731</b>
---------------------	------------

<b>附录二 Latin1 字符指令表和键控代码 .....</b>	<b>736</b>
------------------------------------	------------

<b>附录三 向后兼容 .....</b>	<b>743</b>
-----------------------	------------

<b>附录四 ECMA-262 和 JavaScript 之间的差别 .....</b>	<b>748</b>
--	------------

<b>词汇表 .....</b>	<b>751</b>
------------------	------------

# 序言

1998年的夏天，当我加入 Macromedia 的 Flash 队伍的时候，这个小而精干的团体已经生产出了令人震惊的产品。Flash 3 作为 Web 矢量动画的标准已经基本上得到了全世界的承认。它那些有着艺术天赋的忠实而活跃的用户们创建出的惊世骇俗的视觉作品，越来越多地展示在 Web 上。

ActionScript（动作脚本）的开端可以追溯到 Flash 4 功能计划列表上的一条内容，标题为“增强交互性”。Flash 3 中有一套关于动作的基础功能，可以用来控制 Flash 的影片剪辑和按钮，并提供一定的交互性。不过，我对一个井字游戏的印象颇深，尽管这在大多数编程语言中是一个很简明的任务，但是如果用 Flash 3 的动作功能来执行就会变得很困难，而且非常耗时。

这就是 ActionScript 产生之前的情况。今天，如果你看到一个完全用 Flash 4 来创建的动态网站并不会觉得有什么可吃惊的。而且，站点正趋向于采用 Flash 5 中那些更加练达的 ActionScript 功能。

ActionScript 的一个关键目标是易于使用。非程序员可以容易地使用 ActionScript，这一点非常重要。我们所提供的并不是一个苍白的脚本编辑窗口，而是在 Flash 4 中创建了一个可视的、容易理解的界面，用来为 Flash 影片添加交互功能。Flash 4 中 ActionScript 非常易学，并且使得 Flash 播放器可以保持很小，这是一个重要的考虑因素。

Flash 播放器即使在低带宽的连接上也能快速地下载。Flash 团队在给它增加任何一项功能之前都会自问一句：“这会给播放器增加多少代码？”对于这个标准，ActionScript 也不例外。ActionScript 的目标和播放器的每一个新功能一样，可以说是事半功倍，也就是以播放器尺寸的最小增加带来功能上的最大收益。

我们知道，用户会将 ActionScript 付诸到一些无法预料的用途中去，但看到用户用它做出来的东西毕竟是让人欣慰的事情。在 Flash 4 发布后的一个月内，用 ActionScript 制作出来的精彩站点开始出现在 Web 上——商业站点、聊天室、留言板、娱乐游戏、棋盘游戏，甚至一些用来创建 Flash 站点的 Flash 站点。闸门打开了，一种新的、动态的、交互的、高图形化的 Web 内容奔涌而来。

当设计 Flash 5 时，我最希望做的事情就是将 ActionScript 发展成为一种成熟的脚本语言，它将拥有程序员们在诸如 JavaScript 语言中所熟悉的特征——函数、对象、完善的控制流语句以及多种数据类型。这些东西是帮助程序员更好地使用其他语言的“动力工具”，我希望 ActionScript 也同样能支持它们。我并没有白手起家开始设计这门语言，而是选择在很大程度上模仿 JavaScript，它是 Internet 上客户端脚本的实际标准。更为特殊的是，ActionScript 还模仿了 ECMA（欧洲计算机制造商协会）脚本标准（ECMA-262）。因此，转手从事 Flash 的 JavaScript 程序员会马上发现，他们对 ActionScript 非常熟悉，而且，ActionScript 程序员可以将他们的 ActionScript 知识用在 JavaScript 编程中，并可在两种语言之间轻松地共享代码。

方便易用和播放器尺寸最小化的要求其实是等价的。JavaScript 是一门精细复杂的语言，我们要设法向高级用户提供它的全部功能，并同时保持 Flash 4 ActionScript 的易用性能。最后，新的 Flash 5 动作面板就有了两个模式：普通模式，这是 Flash 4 ActionScript 编辑器的一个简化版本；专家模式，这是针对高级用户的标准文本编辑器。为了最小化播放器尺寸，只能牺牲 ActionScript 的 ECMA 脚本兼容性。例如，ActionScript 不支持在运行时间里用 eval() 来编译代码，这个特点要求将所有的 ActionScript 编译器合并到播放器中，这就造成了播放器尺寸增大，让人难以接受。由于同样的原因，它也不能支持正则表达式匹配。这两个功能都是非常有用的，这说明 Flash 团队要对播放器尺寸和功能需求这对矛盾做出较为平衡的决定真的是很困难。

为了满足这两个要求，我们添加了第三个东西：兼容性。我们所设计的 Flash 5 ActionScript 可以将 Flash 4 的脚本平稳地升级到 Flash 5。而且，Flash 5 将 Flash 4

的 ActionScript 作为一个子集而给予支持，这样一来，Flash 5 实际上就成为建立 Flash 4 影片的出色途径。Colin 已经在附录三以及附录四中描述了向后兼容的问题，以及 ActionScript 和 JavaScript 之间的主要差别（通常是由兼容性的原因）。

纵览整个发展过程，Flash 队伍得到了来自 Flash 用户的巨大支持，这是一个思想自由、联系紧密、有着惊人才智和激情的集体。Flash 用户的指引在塑造产品功能特点方面扮演了重要的角色。Macromedia 的目标是生产出满足消费者需求的产品，它通过倾听消费者的心声，了解他们的工作方式而做到了这一点。

最后，Flash 是一个正在继续的传奇，一项充满生命力的事业，我们将以不懈的努力来满足您的需求。Flash 开发者们是信息时代的艺术家，Flash 队伍的工作是尽可能生产出最好的画笔和刻刀。本书是第一本全部致力于 ActionScript 语言的详尽指南和参考。同样，它标志着 ActionScript 发展过程中的一个关键点：ActionScript 现在已经是一个足够成熟的领域，它能够和这本涵盖了最新材料、技术无一遗漏的书籍相得益彰。

好好享受这本书、享受 Flash 5 ActionScript。我们大家将对你的作品翘首以待！

—— Gary Grossman  
Macromedia 公司 Flash 组，主工程师  
2001 年 3 月



# 前言

本书所要讨论的内容包括ActionScript的基础知识及高级用法。在接下来的内容中，我们将接触 ActionScript 语言的所有细枝末节——从变量和影片剪辑控制基础到诸如对象和类，服务器通信以及 XML 的高级主题。最后，我们就会了解关于 Flash 编程的全部内容。

本书并不是仅为程序员准备的。虽然内容转换得很快，但阅读的时候并不需要具备编程知识。你所需要的只是 ActionScript 之外的 Flash 经验以及学习的热情。当然，如果你已经是一个程序员，那就最好不过了，你马上可以把编码技能用在 ActionScript 上。

本书提供了未被 Macromedia 或第三方书籍公开、或者出版的材料。Flash 因为口述式的技术和深奥的功能而恶名远扬。层、影片剪辑和载入影片如何堆放到播放器里？（参见第十三章）在任何给定的一帧中，是什么在支配代码的执行顺序？（参见第十三章）事件处理有局部作用范围吗？（参见第十章）为什么数字 90 有时会变成 89.9999999997？（参见第四章）我的目的就是在这些未知海域引航。当然，在书中也包含了任何语言都要求的基础编程技术，比如如何让一个代码段重复执行。（参见第八章）

本书的设计目的是书中介绍的技术能被你放在桌上使用，而不是束之高阁。书中的第三部分无一遗漏地包裹了 ActionScript 中所有的对象、类、属性、方法和事件处理器。你可以有条理地学习新东西，并且记住那些容易遗忘的部分。

首先，本书是一个权威性指南。它是在经过了多年的研究，发送了数千个向 Macromedia 工作人员请教的电子邮件，并接收了所有层次用户的反馈信息后才产生的。我希望你们能了解我对此书倾注的极大的热情，对于你们马上就将受益的东西，我在现实世界中付出了艰辛的努力。本书对 ActionScript 的讨论有着绝对的权威性——这是由于 ActionScript 的创建者 Gary Grossman 所给予的技术审查——以及无比的准确性。

## ActionScript 能做什么？

坦白地说，像 Flash 5 ActionScript 这样一种完善的语言，对于它能实现什么目的并没有实际的限制。让我们来看一些 ActionScript 明确给出的性能，以便了解本书所涵盖的主题。首先，你可以想想如何将这些技术合并到你所要达到的目标中去。

### 时间线（Timeline）控制

Flash 影片是由位于一个叫做时间线的线性序列上的多帧构成的。使用 ActionScript，我们可以控制影片时间线的播放，可以播放电影片段，显示特定的某一帧，停止影片播放，建立动画循环，以及对动态内容同步化。

### 交互性

Flash 影片可以接受用户的输入并且作出响应。我们可以用 ActionScript 创建如下的交互元素：

- 对鼠标点击动作作出反应的按钮（例如，一个典型的导航按钮）。
- 基于鼠标移动而发生动作的内容（例如，一个鼠标追踪器）。
- 能够通过鼠标或者键盘而移动的对象（例如，驾驶游戏里的一辆汽车）。
- 允许用户对影片提供输入信息的文本域（例如，一个填充表单）。

### 视频和音频内容控制

ActionScript 可以用来检查或者修改影片中音频和视频内容的属性。比如，我们可

以改变一个对象的颜色和位置、减小声音的音量，或者设置文本块的字体。我们也可以按照时间顺序重复地修改这些属性，产生独特的动作，比如物理运动和冲突检测。

## 内容的程序化生成

使用 ActionScript，我们可以直接从影片库中生成音频和视频内容，或者复制场景中现有内容。程序化的生成内容可以作为严格的静态元素，比如一个任意的视频图案，或者一个交互性元素，例如视频游戏中的敌方太空船或者下拉菜单中的一个选项。

## 服务器通讯

ActionScript 提供了多种工具，能够发送信息到服务器并接收来自服务器的信息。下面的应用中都包括了服务器通讯：

- 链接到一个 Web 页。
- 客户登记。
- 聊天应用程序。
- 多人网络游戏。
- 商业事务。
- 包括用户注册和登录的个人站点。

当然，这些例子只是 ActionScript 潜在应用的一部分。本书的目的是教你一些基本的技巧，让你能独立地去探询其他数不清的可能性。这不是一本食谱，它是一门让你能白手起家处理代码的课程。菜单上最终能有什么美味佳肴则取决于你自己。

## 代码库

在以后的章节中，我们会碰到数十个代码范例。要想获得本书没有包括的有关源文件和其他许多指南文件，可以访问在线代码库，网址为：

<http://www.moock.org/asdg>

代码库是一个发展中的资源站点，它包括现实世界中的 ActionScript 应用程序和代码基础。下面给出了你可以在代码库中找到的一个范例选择列表（你可以单独下载它们，也可以下载整个 .zip 文件）：

- 多项选择测试。
- 以 XML 为基础的聊天应用程序。
- 客户登记应用程序。
- 自定义鼠标指针和按钮。
- 行星游戏代码库。
- 程序化动作效果。
- HTML 文本域演示。
- 预装载程序。
- 串操作。
- 界面小装置，比如滑动条和文本滚动。
- 鼠标追踪器和其他视觉效果。
- 音量和声音控制。

另外，任何有关本书的消息、更新情况、技术注解以及勘误表也会放到前述的 URL 下，或者本书的网站中。

## 站点展示

实际上，现有的每一个 Flash 站点中都至少有一点点 ActionScript。但是一些站点，我们可以说，拥有的 ActionScript 不只一点点。表 P-1 给出了一系列的网址，也许会对你的工作有所帮助。也可以参见附录一中所列的站点。

表 P-1 ActionScript 展示

项目	URL
设计、交互性和脚本编写方面的实验	<i>http://www.yugop.com</i> <i>http://www.praystation.com<sup>a</sup></i> <i>http://www.presstube.com</i> <i>http://www.pitaru.com</i> <i>http://www.flight404.com</i> <i>http://www.bzort-12.com</i> <i>http://kaluzhny.nm.ru/3D.html<sup>a</sup></i> <i>http://www.protocol7.com<sup>a</sup></i> <i>http://www.uncontrol.com<sup>a</sup></i> <i>http://www.digitalnotions.com/dev/flash_5<sup>a</sup></i> <i>http://flash.onego.ru<sup>a</sup></i> <i>http://www.figleaf.com/development/flash_5<sup>a</sup></i>
游戏	<i>http://www.gigablast.com</i> <i>http://www.sadisticboxing.com</i> <i>http://www.flashkit.com/arcade<sup>a</sup></i> <i>http://www.huihui.de</i>
界面和动态内容	<i>http://www.mnh.si.edu/africanvoices</i> <i>http://www.curiousmedia.com</i>

a. 提供可下载的 *.fla* 文件。否则，只可以访问 *.swf* 文件。

## 版式约定

为了表示出 ActionScript 不同的语法成分，本书使用下面的约定：

- 对于代码举例、剪辑实例名称、帧标签、属性名称和变量名使用等宽字体 (**Constant width**)。
- 对函数名称、方法名称、类名、层名、文件名以及诸如 *.swf* 的文件后缀使用斜体。
- 在跟随一个按部就班的程序时你必须键入的代码使用等宽粗体 (**Constant width bold**)。

- 对于你必须用一个恰当的值来代替的代码（例如，在这里写你的名字），或者在代码注释里提到的变量和属性名，使用等宽斜体(*Constant width italic*)。
- 方法和函数名后面要跟括号。

## 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

对于本书我们还有一个专门的 Web 页，在那里列出了勘误表、范例或附加信息。你可以访问下面的网址：

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

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](mailto: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

我非常荣幸认识了下面的这些人，并有幸和他们一起工作，现在要向他们致以深深的谢意：

- Macromedia 才华横溢的 Flash 队伍，他们以不懈的创新精神塑造了一个媒体，他们重视终端后面构成了 Web 的用户。Macromedia 将职业技巧、钻研精神和个人热情融为一体，这在公司体系中是非常罕见的。
- O'Reilly 完美的专业人员：Tim O'Reilly, Troy Mott, Mike Sierra, Rob Romano, Edie Freedman 和许多排版人员、索引编辑人员、校对者以及市场和销售人员，是他们帮助这本书最终得以出版。
- Derek Clayton，我的编程工作的良师益友。除了几乎每天提供代码方面的建议之外，Derek 还贡献了第十七章“Flash 表单”中的 Perl 代码，并且多次教我 Quake 方面的东西。他还用 Java 编写了一个 *XMLSocket* 服务器，用 Perl 写了一个通用的普通文件数据库系统，两者都可以在在线代码库中找到。
- Wendy Schaffer，她不仅为我校对了初稿，还在编写本书的整个阶段无怨无悔地用生命和爱来支持我。
- Bruce Epstein，他是一个高级编辑，帮助我润色了原稿中的几乎每一个句子，经常为主题提供恰当的补充内容。Bruce 出色的编辑工作和指导起了难以估价的作用。
- Gary Grossman，Macromedia 公司 ActionScript 的创建者，他总是抽出时间来解答疑难，讲解深奥的概念，甚至在 ActionScript 的发展阶段还持有争论观点。Gary 是本书的技术编辑指导，协助阐明一些重要的概念和细节。由于他的参与保证了本书的准确性。
- Slavik Lozben，Macromedia 公司的 Flash 工程师，对他创建的影片剪辑事件和 swapDepths 的成果我真是感激不尽！没有 Slavik 的才智和积极性来加入我们的讨论，我恐怕现在还在写事件和事件处理器的章节。Slavik 还是本书的技术编辑，为本书作出了很大的贡献。
- Erica Norton，Macromedia 公司 ActionScript 的 QA（质量评估）工程师，她非常爽快地解答了很多问题。除了接受正式的讨论之外，Erica 还从百忙中抽出时间来担任技术编辑的工作。

- Jeremy Clark, Macromedia 的 Flash 产品经理, 他热情地支持本书的编写, 提出了很多想法、建议, 并且回答我无穷无尽的问题。Eric Wittman, Macromedia 的 Flash 产品管理主管, 多年来他远见卓识的领导塑造出了 Flash。Janice Pearce, Macromedia 公司 Flash 的 QA 队伍成员, 他阐明了各种 Flash 生产问题, 并且友好地提供了 Flash 5 的早期结构。Matt Wobensmith, Macromedia 公司的 Flash 团队经理, 他不断地提供宝贵的经验。Troy Evans, Macromedia 公司的 Flash 播放器生产经理, 他是 Flash 播放器的领导者和拥护者。Bentley Wolfe, 来自 Macromedia 的技术支持队伍, 他仿佛永远也不肯离开他的键盘。
- 来自 O'Reilly & Associates 的 Richard Koman, 他在本书的拟定期间以及早期起草工作中担任编辑工作的领导。
- David Fugate, 来自 Waterside Productions 的我的文学代理, 他的勤奋和自信几乎让工作本身相形见绌。
- D.Joe Duong, 他少年老成, 经常把时间花在教别人上并且对此毫不在意。我非常幸运是他所教过的人之一。Mike Linkovich, 代码专家, 颇具启发性思维。James Porter 和 Andrew Murphy, 他们所做的工作就是阅读、检查, 然后还是阅读、检查, 再阅读、再检查。当然还有 Graham Barton。
- Doug Keeley, Terry Maguire 和 Jon Nicholls, 他们创建了 ICE, 在这个公司中, 工作和热情共存。
- Paul Beam 教授, 他从 WaterLoo 大学令人讨厌的英语教学计划中看到了文学、通信和计算机之间的联系。
- Jack Gray 教授, 我要感谢他的睿智和友善。
- Andrew Harris, David Luxton, Micheal Kavanagh, Stephen Burke, Cheryl Gula, Christine Nishino, Stephen Mumby, Karin Trgovac 和 Judith Zissman, 他们的艺术、思想和友爱值得尊敬。
- Flash 同仁, 我从他们那里汲取了灵感和领悟, 他们包括 James Patterson, Yugo Nakamura, Naoki Mitsuse, Joshua Davis, James Baker, Marcell Mars, Phillip Torrone, Robert Reinhardt, Mark Fennell, Branden Hall, Josh Ulm, Darrel Plant, Todd Purgason, John Nack, Jason Krogh, Hillman Curtis, Glenn Thomas 和其他我可能遗漏了的所有人。

- Moock 一家 (Margaret, Michael, Jane 和 Biz) 教给了我思考、梦想、探索以及爱。
- 感谢 Schaffer 一家多年来给予我的家庭温暖和友爱。

最后，我想感谢的是你——尊敬的读者，是你在百忙中抽时间来阅读这本书。我希望它能够把我的激情传递给你。

Colin Moock

加拿大，多伦多

2001 年 4 月



# 第一部分

## ActionScript 基础

这一部分包括了 ActionScript 语言的核心语法和原理：变量，数据，语句，函数，事件处理，数组，对象和影片剪辑。学完第一部分，你将了解关于编写 ActionScript 程序的一切事项。

- 第一章，针对非程序员的简单介绍
- 第二章，变量
- 第三章，数据和数据类型
- 第四章，原始数据类型
- 第五章，操作符
- 第六章，语句
- 第七章，条件语句
- 第八章，循环语句
- 第九章，函数
- 第十章，事件和事件处理
- 第十一章，数组
- 第十二章，对象和类
- 第十三章，影片剪辑
- 第十四章，词法结构
- 第十五章，高级话题



# 第一章

## 针对非程序员 的简单介绍

我会教你如何同 Flash 对话。

我所指的不仅仅是在 Flash 中编程，而且还要对它说点什么，并且聆听它会如何答复。这并非比喻，也不是某种简单的修辞手法。它是编程的理性方法。

编程语言可以给计算机发送信息、并接收来自计算机的信息。它们是用来通信的词汇和语法的集合，就如同人类语言一样。通过使用编程语言，我们可以告诉计算机应该做什么，或者向它索取信息。它会倾听你的声音，并且试图执行要求的动作，然后给出响应。所以，虽然你也许会觉得读本书是为了学习编程方法，但实际上你是在学习如何同 Flash 通信。当然，Flash 并不会说英语、法语、德语或者粤语，Flash 的自然语言是 ActionScript，你要学会如何讲它。

学习说一种计算机语言有的时候会被认为和学习编程是一码事。但是，编程比学习某种语言的语法更为复杂。如果 Flash 会讲英语——而我们也不需要学习 ActionScript 来和它沟通，情况会怎么样呢？

会发生的事情就是我们对着计算机说：“Flash，你做一个在屏幕内弹跳的球好不好？”

Flash 不可能满足我们的要求，因为它根本不懂“球”这个词，这是一个语文学的问题。Flash 希望我们描述的东西是它所知范围内的对象：影片剪辑、按钮、帧，如此

种种。因此，让我们按照 Flash 的认知情况来改述一下要求，现在发生的情况是：“Flash，制作一个名为 ball\_one 的屏幕内的弹球影片剪辑。”

Flash还是沉默不语，不会满足我们的要求。这个球要多大？它应该被放在哪里？它应该朝哪个方向开始运动？它应该以多快的速度运动？它弹跳的时候应该在屏幕的哪个部分？跳多久？是二维的还是三维的？唔……我们不希望有这么多的问题。实际上，Flash 并不会问我们这些问题。相反，当 Flash 不懂我们的意思的时候，它只是不理会我们的要求而已，或者它会给出一个错误提示。现在，我们要以更明确的说法来模仿 Flash 向我们提出的问题，并且按照以下的步骤重新列出我们的要求：

1. 一个球就是一个名叫 ball 的圆形影片剪辑符号。
2. 一个方形就是名为 square 的四边影片剪辑符号。
3. 制作一个直径为 50 像素的新绿球。
4. 将新的球命名为 ball\_one。
5. 制作一个新的黑方形，边长为 300 像素，将它放到场景中央。
6. 将 ball\_one 放到方形内部顶端的某个地方。
7. 将 ball\_one 以任意方向按照每秒 75 像素的速度移动。
8. 如果 ball\_one 碰到了方形的某边，就让它弹回（逆过程）。
9. 继续执行，直到我让你停下来。

即使用英语给出指令，我们仍然需要搞清控制弹球的所有逻辑，以便让 Flash 懂得我们的意思。很明显，编程比简单的编程语言语法要复杂。正如英语一样，认识很多单词并不一定表示你就能很好地与人交流。

假设 Flash 能说英语的例子表明了编程中四个重要的方面：

- 不管是什么语言，编程的艺术在于逻辑步骤的明确表述。
- 在你想用计算机语言说话之前，先试着用英语进行表述常常很有用。
- 将一种语言翻译为另外一种语言时，它仍然由相同的基础语句构成。
- 计算机并不擅长于设想。它们的词汇量非常有限。

大部分的编程工作和编写代码无关。在你写下哪怕一行 ActionScript 代码之前，要认真思考你究竟要做什么，把你的系统功能制成流程图或者计划表。一旦你的程序在概念层次上得到了充分的描述，你就可以将它转换为 ActionScript。

在编程中（就如在感情、政治和商业中一样）有效的沟通是成功的关键。为了让 Flash 读懂你的 ActionScript，你必须做到语法正确，包括引号、等号和分号。为了确保 Flash 理解你所说的东西，你只能按照它认识的形式表达它所知道的东西。对你来说显而易见的东西计算机并不一定了解。可以这样认为，对计算机编程如同跟小孩说话一样：对任何东西都不要想当然，对每个细节都应清楚明白，对所要完成的任务应列出所有必要的步骤。但是要记住，Flash 又不像小孩，它对你的指示会准确地做到令行禁止。

## 一些基础习语

在任何语言学校第一天的课程中，你都会学习一些基础习语（“早上好”、“你好吗？”等等）。即使你只记住了短语本身而不知道每个词的具体意思，你也同样可以理解其作用，并且能重复引用，起到该词的作用。你一旦学会了语法规则，扩展了词汇量，并且能在更多的上下文环境中使用你所记住的习语，那么，你就可以从更高的层次来理解以前学过的习语了。本章后面的部分会如同语言学校的头天课程一样——你会看到零零碎碎的代码，也会向你介绍一些基础编程语法。本书其余的部分将建立在这些基础之上。你读完全书之后可以回到本章来，看一看你究竟前进了多远。

## 创建代码

我们的第一个练习，是学习如何在 Flash 影片中添加简单的四行代码。几乎所有的 ActionScript 编程都发生在动作（Actions）面板里。我们添加到动作面板的任何指令在影片播放的时候都会被 Flash 执行。现在，我们执行下面的步骤打开动作面板：

1. 打开 Flash，产生一个新的空白文档。
2. 在主时间线上，选择第一层的第一帧。
3. 选择 Window（窗口）→ Actions（动作）命令。

动作面板被分为两个部分：脚本框（右边）和工具箱（左边）。脚本框中包含所有的代码。从工具箱可快速访问 ActionScript 中的动作（Actions）、操作符（Operators）、函数（Functions）、属性（Properties）和对象（Objects）。你可能已经在图 1-1 中认出了 Flash 以前版本里的基本动作。

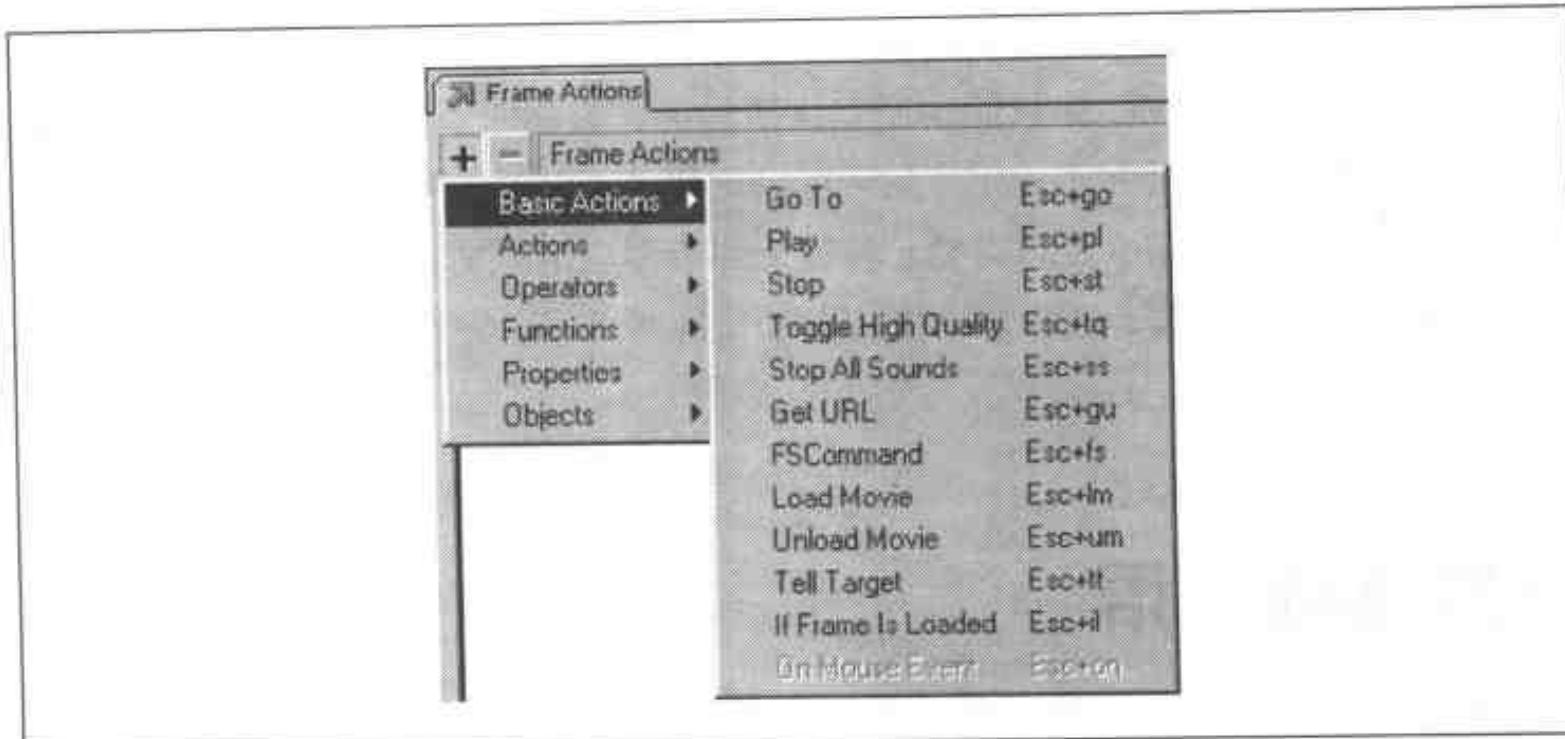


图 1-1 Flash 5 中的基本动作

工具箱中还有更多的东西值得研究：图 1-2 给出了所有可用的动作，包括 Flash 2、3 和 4 中的一些旧命令。如果你继续观察工具箱，甚至还会发现诸如声音、数组和 XML 之类的东西。本书中，我们会介绍所有这些内容。

工具箱部分的菜单可以用来创建 ActionScript 代码。但是，为了学习 ActionScript 的语法、原理和结构组成，我们将亲手编写代码。

---

**注意：** 所谓的动作（Actions）其实不仅仅单纯指动作本身——它们还包括各种各样的基本编程语言工具：变量、条件、循环、注释、函数调用等等。虽然它们在一个菜单里，但是通用名 Actions 使其具有的编程功能显示的不十分清晰。

---

我们会将动作分解，让你从程序员的角度认知这些结构。贯穿本书，我会使用一些恰当的编程术语来描绘该处的动作。例如，我不会写“添加一个 *While* 动作”，而是写“创建一个 *while* 循环”；不写“添加一个 *if* 动作”，而写“建立一个新的条件”；不写“添加一个 *play* 动作”，而写“调用 *play ()* 函数（或者方法）”。这些差别对于学习怎样使用 ActionScript 是非常重要的。

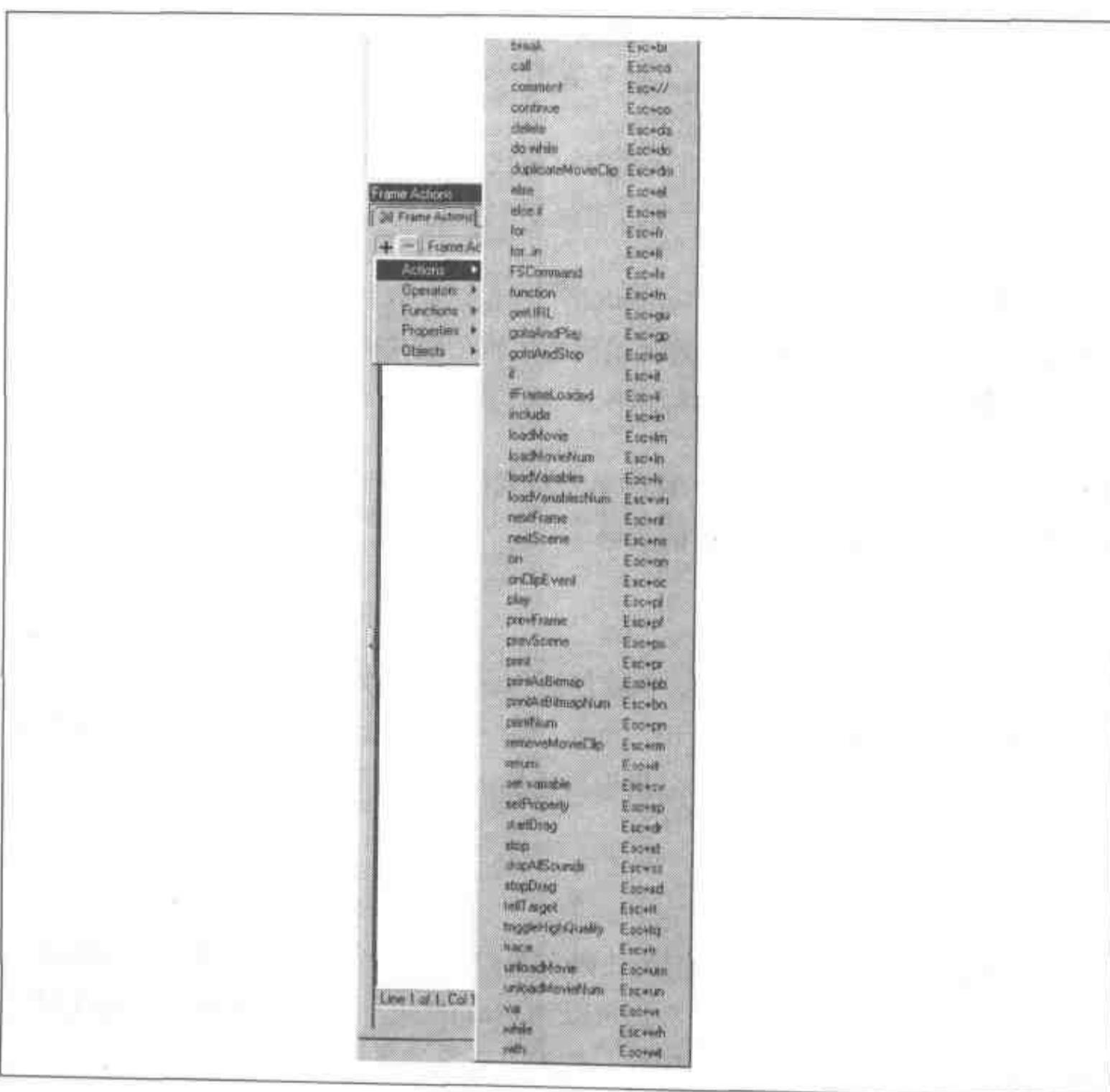


图 1-2 扩展动作命令

你准备好了吗？我们现在该向 Flash 致意了。

## 问候 Flash

在将代码输进动作面板之前，必须按照下面的步骤脱离 ActionScript 自动控制：

1. 选择 Edit (编辑) → Preferences (参数选择)。
2. 在 General (常规) 标签下，选择 Actions Panel (动作面板) → Mode (模式) → Expert Mode (专家模式)。

3. Expert Mode 也可以通过点击动作面板最右边的箭头，从弹出菜单中选择，不过这只能选择当前帧的模式。参见第十六章。

这样如何？你已经是专家了。当进入到专家模式，动作面板下部的参数框就消失了。别着急——我们不用菜单编程，所以也就不需要它了。

下面，选择层 1 的第 1 帧。你的 ActionScript（也就是代码）只能附着在帧、影片剪辑或者按钮上。选择第 1 帧，是为了稍后为该帧创建代码。在专家模式中，可以直接将代码输入到动作面板右边的脚本框中去，我们将在那里完成所有的编程工作。

现在，到了激动人心的时刻——你的第一行代码即将产生。你应该对 Flash 做点自我介绍了！在脚本框中输入下面的代码：

```
var message = "Hi there, Flash!";
```

这行代码构成了一个完整的指令，也就是语句。你可紧接其后键入第二行和第三行代码，如本段之后所示。在“*your name here*”字样处代以你的名字（本书中但凡出现这种斜体字的地方，都可以用你自己的内容来代替这部分代码）：

```
var firstName = "your name here";
trace (message);
```

现在什么事情也没有发生。那是因为代码在我们导出一个.swf文件然后播放影片之前，什么也不会做。在此之前，我们可以让 Flash 还礼。在你刚才所键入的代码后面键入如下所示的第四行代码（朋友，我们可来真的了……）：

```
trace ("Hi there, " + firstName + ", nice to meet you.");
```

好，Flash 准备要迎接你了。选择 Control（控制）→ Test Movie（测试影片），看看产生的效果。输出窗口中会出现一些文字，如图 1-3 所示。

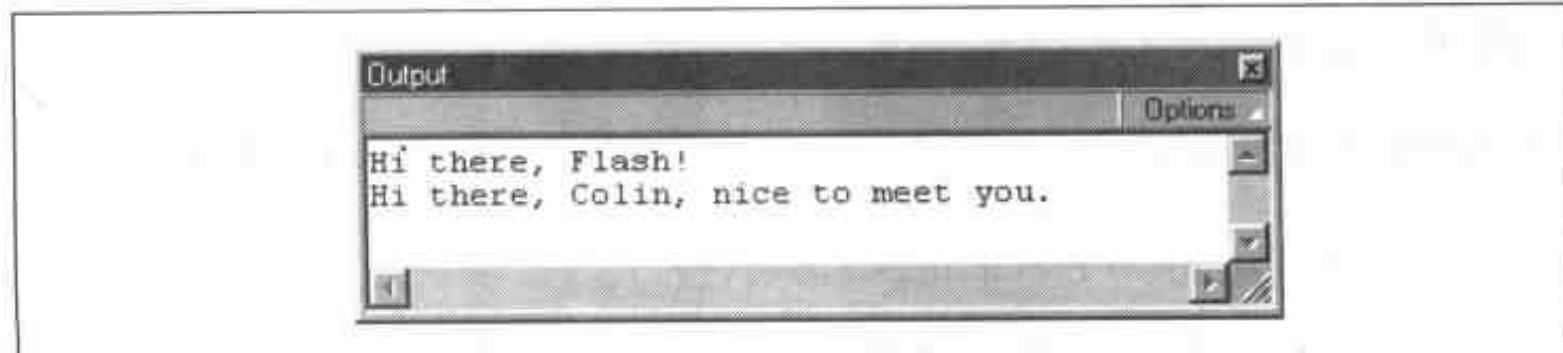


图 1-3 友好的 Flash

非常简洁，对不对？！让我们来研究一下它究竟是如何发生的。

## 记忆能力（变量）

还记得我说过编程实际上就如同和计算机通信一样吗？它的确如此，不过它可能比我迄今为止所描述的要稍微个性化一点。在第一行代码中：

```
var message = "Hi there, Flash!" ;
```

你并没有真的向 Flash 问好。你所说的东西更像是：

Flash，请帮我记忆一个信息——明确的说，就是“Hi there, Flash！”我以后可能需要这个信息，所以请你给它一个叫做message的标签。如果以后我要 message，你就把“Hi there, Flash!”给我就行了。

这虽然似乎不如问好那样礼貌，但它表明了编程的一个实际基础：Flash 可以为你记忆某些东西，并且可以为其添加标签，以便将来找到它。例如，在第二行代码中，我们让 Flash 记住你的第一个名字，然后将其引用命名为 firstName。Flash 记住了你的名字，并且当你测试影片的时候把它显示在输出窗口中。

Flash 能为我们记忆东西的功能在编程中是非常重要的。Flash 可以记住任何数据类型，包括文本（比如你的名字）、数字（比如 3.14159）以及我们后面将要讨论的更复杂的数据类型。

### 正规变量术语

现在，我们来看看关于描述 Flash 如何记忆事物的正规用语。到现在为止，你已知道 Flash 能够记忆数据。一条单独的数据称为一个数据（datum）。一个数据（例如，“Hi there, Flash!”）和标识它的标签（例如，message）合成了变量（variable）。一个变量的标签就是它的名称（name），而变量的数据则是它的值（value）。我们说，变量存储或者包含它的值。注意，“Hi there, Flash!”被双引号包围，表示它是一个文本串（string），而不是一个数字或者其他数据类型。

在第一行代码中，你设定了变量 message 的值。为变量指定一个变量值的动作称为变量赋值，或者简单点，就叫赋值。但是，在你对变量赋值之前，必须首先创建这个变量。我们使用特定关键字 var（你已经在前面使用过了）来声明变量。

因此，在实践中，我可以用更正规的用语来指导你创建前面所讨论的第一行代码：定义一个名为 message 的新变量，然后将它的初始值赋为“Hi there, Flash!”你可以这样写：

```
var message = "Hi there, Flash!";
```

## 幕布后面的魔术师（解释程序）

回忆一下你的前两行代码：

```
var message = "Hi there, Flash!";
var firstName = "your name here";
```

在每个语句中，你创建了一个变量并且为其赋值。但第三行和第四行代码却有所不同：

```
trace(message);
trace("Hi there, " + firstName + ", nice to meet you.");
```

这些语句使用了 *trace()* 命令。你已经见识过了这个命令的作用——它会将文本显示在输出窗口中。在第三行代码中，Flash 显示了变量 message 的值。而在最后一行，Flash 将变量 firstName 转化为它的值（你输入的任何文字），并且把这个值显示在“Hi there”后面。也就是说，*trace()* 命令会将任何指定的数据显示在输出窗口中（这可以让你容易了解程序运行时的情况）。

问题在于，究竟是什么让 *trace()* 命令将文字放到输出窗口中呢？当创建一个变量或者执行一个命令的时候，实际上在使用 ActionScript 解释程序，是它运行程序、管理代码、倾听指令、执行任何 ActionScript 命令，处理语句，存储数据，发送信息，计算数值，甚至在 Flash 播放器装载影片的时候启动基础编程环境。

解释程序将你的 ActionScript 翻译为计算机能够理解并用来执行代码的语言。在影片播放期间，解释程序总是积极、忠实地试图理解你所给出的命令。如果解释程序能够理解你的命令，它就会将命令发送到计算机处理器以供执行。如果某个命令产生了结果，解释程序会对你作出响应。如果解释程序不能理解命令，它就会给你发一个错误信息。因此，解释程序其实就相当于 ActionScript 的交换处理器——你编写代码时它是忠实的听众，同时它又是一个使者，可以将 Flash 的回应反馈给你。

我们可以通过检查解释程序如何处理一个简单的*trace()*动作，以深入研究它的工作情况。

你可以站在解释程序的角度考虑这个命令：

```
trace ("Nice night to learn ActionScript. ");
```

解释程序马上就从特定的法定命令名称列表中认出了关键字*trace*。解释程序还知道*trace()*是用来在输出窗口中显示文字的，因此它还希望能被告知要显示哪些文字。它在*trace*之后的两个括号之间发现了“Nice night to learn ActionScript”，就想：“啊哈，这不就是我要的吗？我要马上把它发送给输出窗口！”

注意，命令是以一个分号（;）来结束的。分号的作用是作为句子末尾的句点，每一个ActionScript语句都以分号结束，只有极少的例外。当语句成功地得到理解，并且得到了所有需要的信息，解释程序就会把命令翻译为处理器可以执行的形式，并将文字显示在输出窗口中。

对于计算机处理器和解释程序究竟如何工作，我们上面所说的只不过是一个简单的描述，但它却表明了如下概念：

- 解释程序总是接受你的指令。
- 解释程序必须一个字一个字地读你的代码，并试图理解它们。这和你读书时努力理解书中的句子一样。
- 解释程序按照严格的规则来读你的ActionScript。例如，如果*trace()*语句中的括号被写丢了，解释程序就不能理解这是什么意思，这样，命令就失败了。

虽然你现在只简单了解了解释程序，但它与你将像相交多年的爱人一样亲密：无数的争吵，无数的叫喊——“你为什么不听我的？！”——以及当你们互相理解时那些美妙的时刻。非常奇怪，我父亲总是告诉我说，学习一门新语言的最好方法就是找一个说这种语言的爱人。所以，对于你和ActionScript解释程序的新关系来说，我或许是第一个祝福你一帆风顺的人。从现在开始，我会在描述如何执行ActionScript指令时正式用解释程序来代替Flash。

## 所需的额外信息（参数）

你已经看到了在执行 *trace()* 命令时向解释程序提供显示文字的情况，这种方法非常普通。我们经常执行一个命令，然后向解释程序提供执行该命令所需的辅助信息。对于发送给命令的数据，有一个特殊的名称：参数或者参量。要向命令提供参数，就要把参数用括号括起来，如下所示：

```
command(argument);
```

对一个命令提供多个参数的时候，用逗号将它们分隔开，如下所示：

```
command(argument1, argument2, argument3);
```

向命令提供参数被称为传递参数。例如，在代码 *gotoAndPlay(5)* 中，*gotoAndPlay* 是命令的名称，而 5 是被传递的参数（在本例中这个数字表示帧号）。有一些参数，比如 *stop()*，虽然要求带括号，但却不接受任何参数。我们会在第九章中讲解。

## ActionScript 的粘合剂（操作符）

现在我们再来看看你的第四行代码，它包括 *trace()* 语句：

```
trace ('Hi there, ' + firstName + ', nice to meet you.');
```

看见 +（加号）符号了吗？它是用来把我们的文字连接（串联）在一起的，它只是诸多可用操作符之一。编程语言中的操作符类似于人类语言中的连接词（“和”、“或者”、“但是”等等）。它们是联合及处理代码短语的工具。在 *trace()* 例子中，加号把引号内的文字 “Hi there,” 和变量 *firstName* 所代表的文字连接起来。

所有的操作符都将代码短语连接到一起，然后在处理过程中操作这些短语。不管短语是文字、数字或者其他某种数据类型，一种操作符几乎总是执行某种规定的转换。通常，操作符将两个东西合并到一块，就像加号那样。但是也有一些操作符会对值进行比较，执行赋值动作，协助逻辑判断，确定数据类型，创建新对象，以及对变量提供其他方便的服务。

和两个数字操作数一起使用的时候，加号（+）和减号（-）执行基本的数学计算。下面的代码会在输出窗口中显示数字 3：

```
trace (5-2);
```

小于符号操作符将检查哪一个数字更小，或者按字母表顺序来确定哪个字母在前边：

```
if (3<300) {  
    // 完成某些工作……  
}  
  
if ('a'<'z') {  
    // 完成其他某些工作……  
}
```

由操作符来执行的联合、比较、赋值或者其他操作称为“运算”。数字计算是最容易理解的运算，因为它们遵循基本的运算法则：加（+）、减（-）、乘（\*）、除（/）。但是对一些操作符会稍显陌生，因为它们执行的是特殊的编程任务。比如，我们来看看 *typeof* 操作符，它会告诉我们变量里所存储的数据是何种类型的。所以，如果我们创建了一个变量 *x*，然后将它赋值为 4，我们就可以问解释程序 *x* 包含的数据类型了，如下所示：

```
var x = 4;  
trace (typeof x);
```

当这行代码在 Flash 中执行的时候，我们会在输出窗口中得到一个词 *number*。注意，我们为 *typeof* 操作符提供了一个要操作的值，但是却没有用括号：*typeof x*。这样一来，你可能会对 *x* 究竟是不是 *typeof* 的参数感到疑惑。实际上，*x* 充当的就是参数的角色（它是代码短语计算所需的辅助数据），但在和操作符一起使用的时候，诸如 *x* 这种变量的正规名称是“操作数”。一个操作数就是一个操作符所计算的一个项目。例如，在表达式 *4+9* 中，数字 4 和 9 就是 + 操作符的操作数。

第五章中包括了 ActionScript 中所有操作符的具体内容。现在你只需要记住操作符能在某种转换中将代码短语连接起来就行了。

## 总结

现在，我们来回顾一下你所学到的东西。下面再次给出第一行代码：

```
var message = "Hi there, Flash!";
```

关键字 *var* 告诉解释程序，我们现在做的工作是声明（创建）一个新的变量。词 *message* 是变量的名字，等号是一个用来将文本串（“Hi there, Flash!”）赋值给变

量 message 的操作符、文本 “Hi there, Flash!” 就成为了 message 的值，最后，分号（;）将告诉解释程序，我们已经结束了这一行代码。

第二行和第一行非常相似：

```
var firstName = 'your name here';
```

在这里，我们将把你输入的用来代替 your name here 的文本串赋值给变量 firstName，并用分号结束第二行语句。

然后，我们在第三行和第四行中使用了变量 message 和 firstName：

```
trace (message);
trace ("Hi there, " + firstName + ", nice to meet you.");
```

关键字 trace 通知解释程序，我们想在输出窗口中显示某些文字。我们将要显示的文字作为变量进行传递。前括号标志着变量的开始。在第四行中，变量本身包含两个运算，都用了加号操作符。第一个运算将头一个操作数 “Hi there” 连接到第二个操作数 firstName 的值。而第二个运算则将 “nice to meet you” 和第一个运算产生的结果相连接。后括号标志着参数的结束，而分号仍然用来表示语句的结束。

好了！这就是你的第一个 ActionScript 程序。你已经有了一个很好的开端，这可以算是一个重要的里程碑。

## 更为深入的 ActionScript 概念

我已经向你介绍了很多构成 ActionScript 的基本元素：数据、变量、操作符、语句、函数以及参数。在我们对这些概念进行深一步的讨论之前，先来简略看看 ActionScript 的其他核心功能。

## Flash 程序

对于大多数的计算机用户来说，一个程序和一个应用软件是等同的，比如 Adobe Photoshop 或者 Macromedia Dreamweaver。显然，这不是我们在 Flash 中编程时所要创建的东西。程序员的工作，换句话说，就是定义一个程序，它是代码的集合（一系列语句），但这也仅仅只是我们要构造的东西的一部分。

一个 Flash 影片不只是些代码行。Flash 中的代码和 Flash 中诸如帧和按钮之类的元素是混合在一起的。我们将代码添加到这些元素中，这样，它就能和元素互相作用。

最后，在标准的观念中确实没有 Flash 程序这样的东西。我们有的不是用 ActionScript 写的完整程序而是脚本：给影片予计划性动作的代码段。比如 JavaScript 脚本，它会给 HTML 文档一些计划性的动作。我们真正构筑的产品不是一个程序，而是一个完整的影片（包括它的代码、时间线、视频、声音以及其他元素）。

我们的脚本包括在传统程序里所看见的大部分东西，但不包括用诸如 C++ 或者 Java 这类语言来编写的程序，这类程序可用来把图形放到屏幕上或插入声音。我们不需要做管理图形和声音编程的琐碎事务，这就让我们可以把大部分注意力集中到影片动作的设计上。

## 表达式

脚本的语句，正如我们所学到的，包括了脚本的指令。但是大部分指令如果没有数据都是没有用的。例如，我们建立一个变量的时候，要用数据来为它赋值。当我们使用 *trace()* 命令的时候，要将数据作为参数传递，以便其能在输出窗口中显示。数据是我们在 ActionScript 代码中所操作的内容。在整个脚本中，你将要做获取、给予、存储数据，以及其他诸多围绕数据的工作。

在程序中，程序运行时能产生一个单一数据的任何代码短语都叫做表达式。数字 7 和串 “Welcome to my web site” 都是简单的表达式。它们表示在程序运行时将要按原状态使用的简单数据。同样，这些表达式被称为直接量表达式，或者简称为直接量。

直接量只是一种类型的表达式。变量也可以是一个表达式（变量表示的是数据，因此它们可以作为表达式）。表达式被操作符连接起来的时候就更有意思了。例如，表达式  $4+5$  是一个有两个操作数（4 和 5）的表达式，但是加号操作符让整个表达式产生出单一的值，也就是 9。复杂的表达式可能包含其他较短小的表达式，规则是：整个代码短语仍然能够被转换为一个单一的值。

下面，我们来看一下变量 message：

```
var message = "Hi there, Flash!";
```

如果我们喜欢，可以将变量表达式 `message` 和直接量表达式 “How are you?” 连接起来，如下所示：

```
message + " How are you?"
```

当程序运行的时候，它就会变成 “Hi there, Flash! How are you?” 在处理数字计算的时候，经常可以看到含有短表达式的长表达式，比如：

```
(2 + 3) * (4 / 2.5) - 1
```

在刚涉足编程的时候，表达式是很重要的内容，因为这个词经常被用来描述编程概念。例如，我会写“要对变量赋值，先输入变量名称，然后输入等号，后边接任何表达式。”

## 两个重要的语句类型：条件和循环

在几乎所有的程序里，我们都会使用条件来对程序添加逻辑控制，而用循环来执行有重复性的任务。

### 用条件来作选择

Flash 编程一个真正的优点是可以让你的影片更加机敏。我所说的“机敏”是指：设想一个叫作 Wendy 的女孩，她不想把衣裳弄湿。每天早上在 Wendy 离家之前，她都看看窗外的天气，如果在下雨，她就带把伞。Wendy 就很机敏。她使用基本的逻辑——能够从一系列选项中根据环境情况选择究竟要做什么。我们在创建交互动画的时候使用同样的基本逻辑。

下面是一些 Flash 影片的逻辑举例：

- 设想我们的影片中有三个片段。每当用户到达某一个片段的时候，我们用逻辑来决定是否要向她显示该片段的介绍。如果她已经看过该片段，我们就跳过介绍。否则，就显示介绍内容。
- 假设我们有一个受限制的电影片段。为了进入受限制区域，用户必须输入密码。如果用户输入了正确的密码，就为她显示限制内容。否则，就不播放。

- 假设我们正在大街上玩球，我们想让球从墙面上弹回来。如果球的线路正确，我们就反转它的运动方向。否则，就让球按照原路线运动。

这些影片逻辑例子要求使用特殊类型的语句，称为条件语句。条件句允许我们指定其后的代码段应该——或者不应该——得到执行。这里是一个条件语句的例子：

```
if (userName == "James Bond") {  
    trace ("Welcome to my web site, 007.");  
}
```

条件语句的逻辑结构是：

```
if (该条件得到满足) {  
    // 那么就执行这里的代码  
}
```

你可以在第七章中学习更多的语法细节。现在，要记住：条件语句可以让 Flash 进行逻辑选择。

## 利用循环来执行重复任务

我们的影片不仅需要作选择，我们还想让它执行一些单调乏味的重复性任务。也就是说，直到它们接管了整个世界，奴役我们，把我们养在小小的豆荚里就像……等等……忘了我曾经告诉你的……啊嗨。假设你想要在输出窗口中显示五个数字的序列，而且你希望这个序列以某个指定的数字开始。如果开始数字是 10，你可以这样来显示：

```
trace (10);  
trace (11);  
trace (12);  
trace (13);  
trace (14);
```

但是如果你想要从 513 开始序列的显示，你必须重新输入所有的数字，如下所示：

```
trace (513);  
trace (514);  
trace (515);  
trace (516);  
trace (517);
```

我们可以基于一个变量来使用 *trace ()* 语句，从而避免重新输入，就像这样：

```
var x =1;
trace (x);
x = x+1;
trace (x);
x = x+1;
trace (x);
x = x+1;
trace (x);
```

在第1行中，我们将变量 `x` 的值设置为 1。在第2行中，我们将该值发送给输出窗口。在第3行中，我们获取 `x` 的当前值并将其加1，然后将结果返回变量 `x`，这样，`x` 就变成了 2。最后我们将 `x` 的值再次发送给输出窗口。我们将这个过程重复了三次。代码运行结束后，就会在输出窗口中显示这五个数字构成的序列。这样做的妙处在于，如果我们现在想要改变序列的起始数字，只需要改变 `x` 的初始值就可以了。因为剩余的代码是以 `x` 为基础的，当程序运行的时候整个序列都将得到改变。

这对于我们的第一个方法来说是一个进步，当我们只显示五个数字的时候它会运行得很好，但是如果我们要想显示 500 个数字就不实用了。为了执行更高的重复性任务，我们要用循环（loop）——这是一种可以让代码块重复执行任意次的语句。循环有好几种类型，每种类型都有它自己的语法。最常用的循环类型是 `while` 循环。如果我们的例子不用一系列重复语句而使用 `while` 循环语句，它就会像这个样子：

```
var x =1;
while (x<=5) {
    trace (x);
    x = x+1;
}
```

关键字 `while` 表示我们要开始循环了。表达式(`x<=5`)控制循环执行的次数（只要 `x` 小于或等于 5），语句 `trace (x);` 和 `x = x+1;` 在循环的每次重复（或者说一次反复）中都会得到执行。这样，循环只帮助我们节省了 5 行代码，但是如果你要计数到更高的数字，它就能节省数百行代码了。循环是非常灵活的。为了让循环可以计数到 500，我们只需要简单地将表达式(`x<=5`)修改为(`x<=500`)：

```
var x =1;
while (x<=500) {
    trace (x);
    x = x+i;
}
```

和条件句一样，循环是编程中使用最频繁、最重要的语句类型之一。

## 模块化代码（函数）

到现在为止，你的最长的脚本只由区区4行代码所组成。但很快，4行代码就会变成400行甚至4 000行了。迟早你都会想方设法来管理你的代码，减少工作量，让你的代码可以很容易地在多种环境下使用。这时候，你就会开始喜欢函数了。一个函数就是一个语句系列包。在实践中，函数基本上就是可以复用的代码块。

假设你想要快速编写一个脚本来计算四边形的面积。如果不用函数，你的脚本就会是这样的：

```
var height = 10;
var width = 15;
var area = height * width;
```

现在设想一下，你想计算5个四边形的面积。你的代码就会增长5倍：

```
var height1=10;
var width1=15;
var area1 = height1 * width1;
var height2 =11;
var width2 =16;
var area2 = height2 * width2;
var height3=12;
var width3 =17;
var area3 = height3 * width3;
var height4 =13;
var width4 =18;
var area4 = height4 * width4;
var height5 =20;
var width5 =5;
var area5 = height5 * width5;
```

因为我们一次又一次地重复面积的计算，我们最好将它放到一个函数里，然后多次执行该函数：

```
function area (height, width) {
    return height * width;
}
area1 = area (10,15);
area2 = area (11,16);
area3 = area (12,17);
area4 = area (13,18);
area5 = area (20,5);
```

我们首先用函数语句来创建面积计算函数，它定义（声明）了一个函数，正如var声明一个变量一样。然后，我们将函数命名为area，正如我们给变量命名一样。在

括号中，我们列出函数每次使用所需的参数：height和width。在花括号({})中，我们写上要执行的语句：

```
return height * width;
```

在创建函数之后，我们可以通过使用它的名字而在影片的任何地方运行它所包含的代码。在本例中，我们要调用area()函数5次，每次都要传递它所需要的参数：area(10,15), area(11,16)，如此类推。每次计算的结果会存储在变量area1到area5中。这样就显得又清晰、又简洁，比没有函数的代码要省事多了。

如果你对函数举例还有不明之处先不要着急，我们在第九章中还要学习有关函数的更多内容。现在，只需记住函数给了你一个强有力的方法去创建复杂的系统。函数帮助我们重新使用代码，封装它的功能性、突破了实际创建的限制。

## 内置函数

注意，函数像trace()动作那样需要参数。调用函数area(4,5);看起来和trace(x);这类语句执行trace()命令非常相似。这种相似并非偶然。正如我们前边所说的，许多动作，包括trace()动作，实际上都是函数。但它们是一种特殊类型的函数，是建立在ActionScript内的（和area()这样的用户自定义函数相反）。因此，我们说“调用gotoAndStop()函数”比说“执行gotoAndStop动作”更为合理，也更准确。内置函数是和ActionScript共生的可再度使用的代码块，给了我们很多方便。内置函数可让我们完成从执行数字运算到控制影片剪辑的各种工作。所有的内置函数都在第三部分中列举出来了。我们在学习ActionScript基础知识的时候也遇到了很多内置函数。

## 影片剪辑实例

在有关编程基础知识的所有讨论中，我希望你还没有忘记Flash的基础。Flash中视觉编程的一个关键就是影片剪辑实例。作为一个Flash设计者或者说开发者，你应该对影片剪辑已经很熟悉了，但是你可能不会将影片剪辑当作一种编程元件。

每一个影片剪辑都有符号定义，它位于Flash影片库中。我们可以在Flash影片中添加单个影片剪辑符号的很多拷贝，或者说实例，只要将剪辑从库中拖曳到场景中就

可以实现。高级 Flash 编程的一个重要方面就是影片剪辑实例的控制问题。比如一个弹跳的球，无非是一个被放在场景中的影片剪辑实例。我们可以在影片播放期间通过 ActionScript 来改变一个实例的位置、尺寸、当前帧、旋转等等情况。

如果你对影片剪辑及其实例不甚熟悉，可以先参考 Flash 文件或者帮助文件，然后再继续阅读本书。

## 基于事件的执行模式

在对 ActionScript 基础知识的纵览中，我们应该讨论的最终话题是执行模式 (execution model)，它规定了影片中的代码什么时候运行（也就是得到执行）。在影片中，你可以将代码添加到各种帧、按钮和影片剪辑上。但它们什么时候才能真正开始运行呢？要回答这个问题，我们先来简要地回顾一下计算机历史中存储器的发展过程。

在计算机时代的初期，程序的指令按照它们出现的顺序来连续执行，从第一行开始，到最后一行结束。程序就意味着执行一些动作，然后停止。这种程序被称为批处理程序，它不能够处理如 Flash 这种基于事件的编程环境所需要的交互性。

基于事件的程序不像批处理程序那样按照线性顺序来运行代码。它们连续运行（在一个事件循环中），等待事情（事件）的发生，然后执行代码段作为对这些事件的响应。在为视觉交互环境设计的语言（比如 ActionScript 或者 JavaScript）中，事件就是指典型的用户动作，比如鼠标点击或者击键动作。

当一个比如鼠标点击之类的事件发生时，解释程序就会发出警报。然后，程序会响应该警报，要求解释程序执行一个相应的代码段。例如，如果用户点击影片中的按钮，我们可以执行一些代码来显示影片的其他片段（标准的导航）或者将变量递交到数据库（标准的表单提交）。

但是程序并不会对事件作出反应，除非我们创建事件处理器 (event handler)。下面的模仿代码说明了一般情况下事件处理器的建立情况：

```
when (this event happens) {
    execute these lines of code
}
```

代表性的编写形式如下所示：

```
on (event) {  
    statements  
}
```

实际上，一个将播放头移动到第 200 帧的按钮事件会被这样处理：

```
on (press) {  
    gotoAndStop (200);  
}
```

因为基于事件的程序总是运行一个事件循环，准备对下一事件作出反映，因此，它们好像是活的系统。事件是 Flash 影片中一个至关重要的部分。没有事件，我们的脚本根本不会做什么事情——只有一个例外：当播放头进入某帧的时候，Flash 执行该帧的代码。隐藏事件就是指播放头进入特定的帧，这是 Flash 的隐含性质，不需要显式的事件处理器。

确切地说，是事件促使情节的发生，这就是你在 ActionScript 语言学校头一天的最后时刻才接触到它们的原因。你已经学习了脚本所包含的东西，以及控制这些脚本得以实际执行的因素（比如说事件）。我想说，你已经有了充分的准备可以开始你的第一次真正对话了。

## 创建多项选择测试

现在，我们已经接触了 ActionScript 的基本原理，让我们把这些原理放到真实的 Flash 影片环境中去。我们要用非常简单的编程技术（大部分你都已经学过了）来创建一个多项选择测试，开始 Flash 编程应用研究。我们在后面的章节中会再次观察这里所要创建的测试，看看在你学习了更高级的编程概念之后能有什么提高。最后，我们可以将代码编得更讲究，以便更容易扩展和维护，而且，我们要给测试添加更多的功能，以便它能更容易地处理试题，不受数量限制。

该测试的完整 .fla 文件可以在在线代码库中找到。这是 Flash 编程，而非 Flash 制作的一个课程。它首先假定你已经对按钮、层、帧、关键帧和文本工具的创建和使用驾轻就熟了。测试显示了 ActionScript 编程在如下几个方面的实际应用：

- 变量。
- 用函数来控制影片的播放头。
- 按钮事件处理器。
- 简单条件句。
- 屏幕显示信息的文本域变量。

## 测试总述

图 1-4 显示出了测试的部分情况。这个测试只有两个问题，每个问题有三个多选答案。用户点击和他们的选择相对应的按钮，就可以提交答案。选择情况被记录在变量里，可以用来评定用户的成绩。所有的问题都回答完毕之后，正确答案的数目被计算出来，然后显示用户的得分。

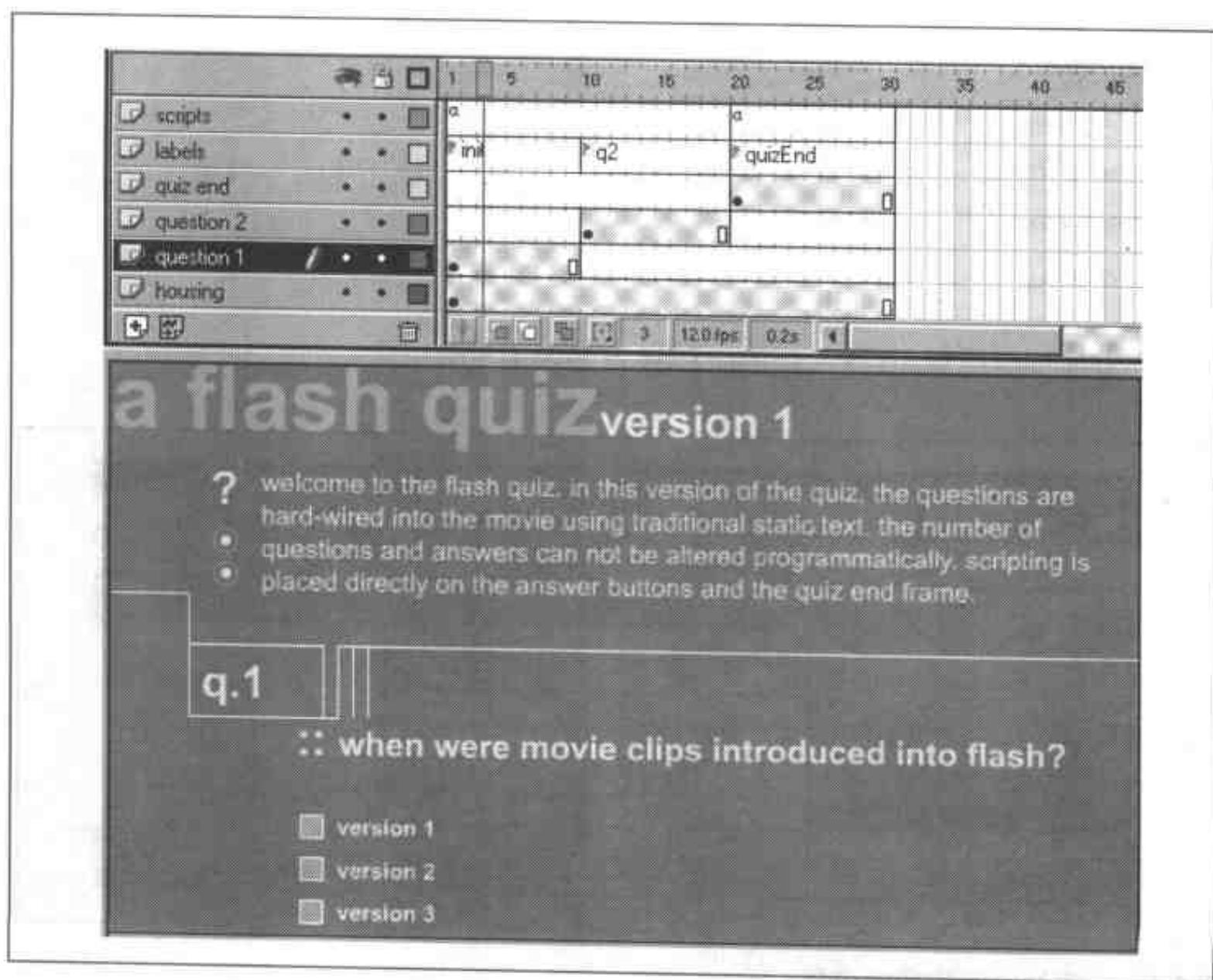


图 1-4 一个用 Flash 制作的测试

## 建立层结构

制作 Flash 影片的时候，将影片内容划分到易于管理的不同部分是非常重要的，你可以将不同的元素内容放到各自的层中。对影片内容分层通常是一种非常好的制作技术，但它在 Flash 编程中是非常基本的。在我们要制作的测试和绝大部分的脚本化影片中，我们会将时间线脚本放到一个单独的层中，成为 *scripts* 层并把该层放在层堆栈的第一个，这样就会好找一点。

我们还要将所有的帧标签放在一个独立的层中，叫做 *labels* 层。*labels* 层在你的所有时间线上都处在 *scripts* 层之下。除了这两个标准层 (*scripts* 和 *labels*) 之外，我们的测试影片有一系列的内容层，分别容纳各种独立的元素。

开始创建我们的测试影片时，要创建并命名下面的层，并按照它们在这里的出现顺序进行安排：

*scripts*  
*labels*  
*quiz end*  
*question 2*  
*question 1*  
*housing*

现在，为每个层添加 30 帧，你的时间线将如图 1-5 所示。

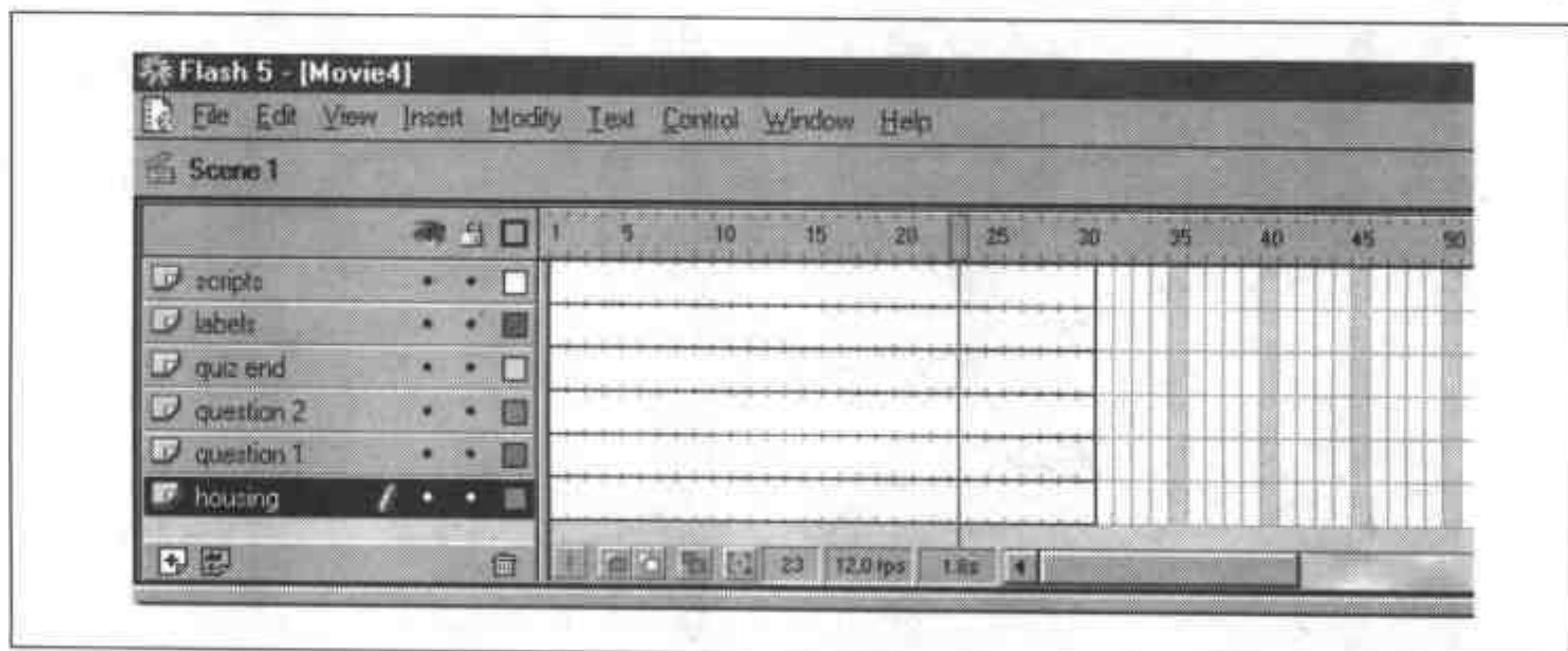


图 1-5 测试时间线初始设置

## 创建界面和问题

在我们运行测试的脚本之前，需要构建让用户进行测试的问题和界面。

下面是你所要执行的步骤：

1. 选择 *housing* 层的第 1 帧，使用文本工具直接在场景上输入你的测试题目。
2. 在 *question 1* 层的第 1 帧中，对第 1 道试题添加题号“1”和文字“*When were movie clips introduced into flash?* (是什么时候把影片剪辑引入 Flash 中的)”在问题下面为答案文字和按钮留出空间。
3. 创建一个简单的按钮，看起来就如同一个单选框或者单选按钮一样，其高度不要超过一行文字（如图 1-6 所示）。
4. 在问题文字下面（还是在 *question 1* 层），添加 3 个多选答案的文字：Version 1, Version 2 和 Version 3，每个都单独占一行。
5. 在 3 个答案的每一个旁边，放置一个单选框按钮的实例。
6. 我们可以把问题 1 当作问题 2 的模板。选择 *question 1* 层的第 1 帧，然后选择菜单命令 Edit (编辑) → Copy Frames (复制帧)。
7. 选择 *question 2* 层的第 10 帧，然后执行菜单命令 Edit (编辑) → Paste Frames (粘贴帧)。第一个问题的复本就出现在 *question 2* 层的第 10 帧处。
8. 仍在 *question 2* 层的第 10 帧上，将问题号码从 1 改为 2，并将问题文字改为 *When was MP3 audio support added to Flash?* (Flash 是什么时候增加 MP3 音频支持的?) 将多选答案改成 Version 3、Version 4 和 Version 5。
9. 最后，为了防止第 1 个问题出现在第 2 个问题下面，可以在 *question 1* 层中的第 10 帧处添加一个空白关键帧。

图 1-6 给出了为测试添加第一个问题之后的 Flash 影片。图 1-7 显示了在添加第二个问题之后时间线的情况。

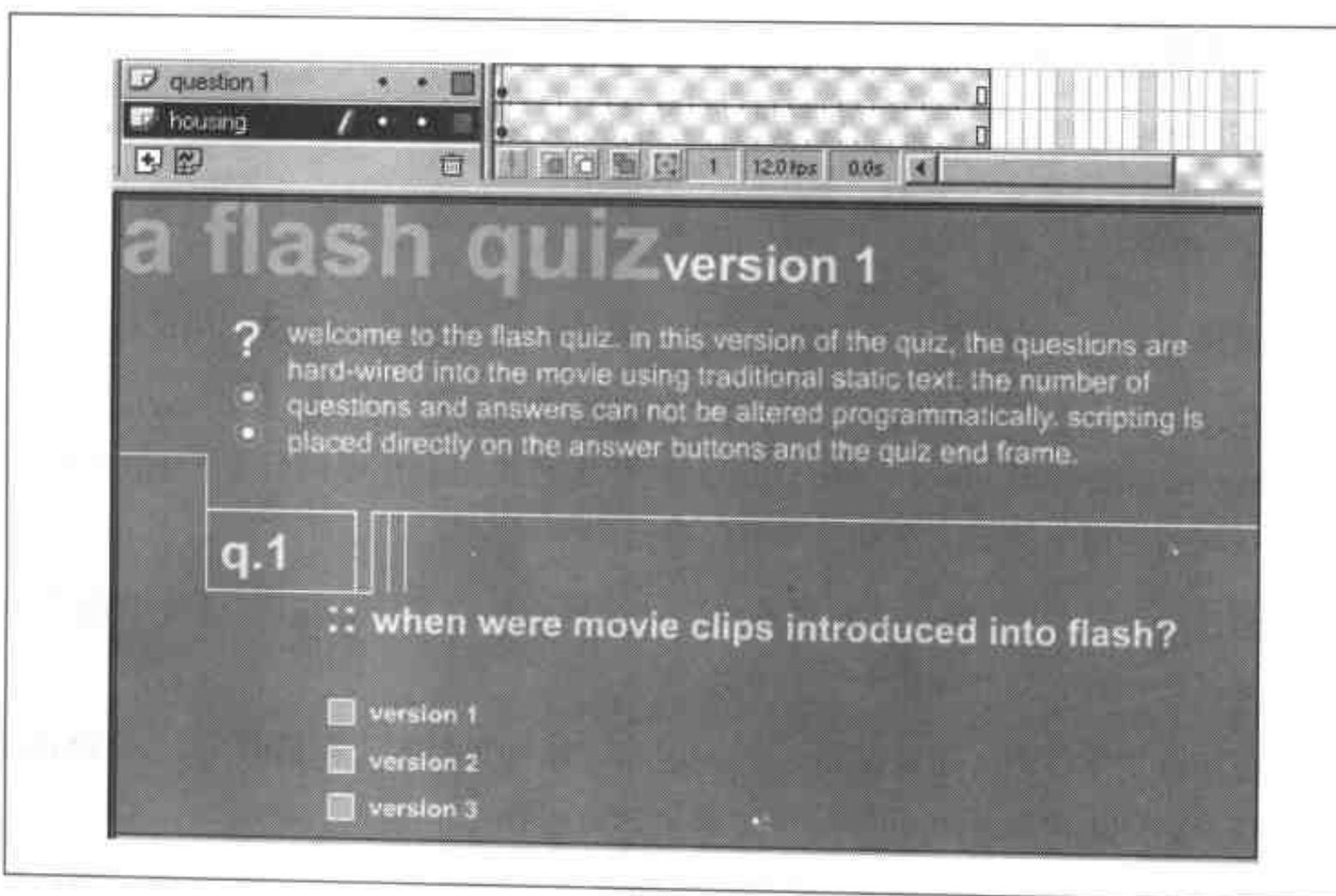


图 1-6 测试标题和第 1 道题目

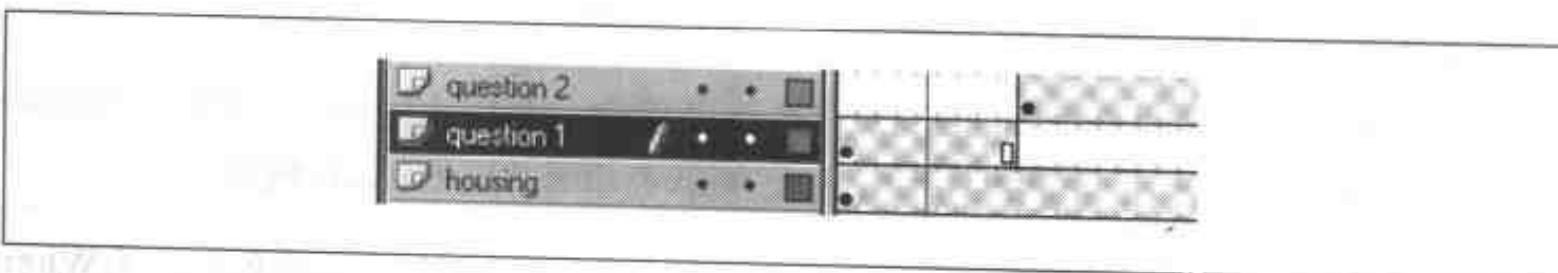


图 1-7 测试里分别容纳两个题目的时间线

## 测试初始化

测试脚本中（也是大多数脚本中）要做的第一件事情就是创建主时间线变量，它的使用将贯穿整个影片。在测试中，我们在影片的第 1 帧中完成这个工作，但在其他影片里，我们通常在载入部分或全部影片之后再这样做。不管怎么说，我们要在任何脚本出现之前进行变量的初始化。一旦变量得到定义，我们可以调用 `stop()` 函数来让用户停留在第一帧上（也就是测试开始的地方）。

对于更为复杂的影片，我们也可以通过调用函数或者进行变量赋值来设置初始条件，

为影片的剩余部分作准备。这一步骤被称为初始化。用来开始动作过程或者定义系统操作所需初始条件的函数通常被命名为 *init*。

我们的测试初始化代码，如例 1-1 所示，它被添加到影片 *scripts* 层的第 1 帧中。

#### 例 1-1：测试的初始化代码

```
// 初始化主时间线变量
var q1answer;           // 用户对第 1 道题目的答案
var q2answer;           // 用户对第 2 道题目的答案
var totalCorrect = 0;   // 正确答案的数目
var displayTotal;       // 显示用户成绩的文本域

// 将影片停在第一个问题上
stop();
```

初始化序列的第 1 行是一个代码注释。代码注释是你添加到代码中的注解，用来解释所发生动作。单行的注释以两个正斜杠和一个空格开始，后面跟一行文字：

```
// 这是一个注释
```

注意，注释可以和代码放在同一行中，如下所示：

```
x = 5;      // 这也是一个注释
```

例 1-1 的第 2 行创建了一个名为 *q1answer* 的变量。回忆一下，创建一个变量的时候，我们用 *var* 关键字，后面跟一个变量名，比如：

```
var favoriteColor;
```

这样，代码中的第 2 到第 5 行就定义了我们所需要的变量，分别以解释它们用途的注释来结束：

- *q1answer* 和 *q2answer* 包含用户答案（1, 2 或 3，表示对于每一个问题，3 个选项中哪一个被选中）的值。我们将用这些值来检查用户的答案是否正确。
- *totalCorrect* 在测试结束的时候用来记录用户答对的题目数量。
- *displayTotal* 是我们用来在屏幕上显示 *totalCorrect* 值的文本域名称。

仔细看一看例 1-1 中的第 4 行代码：

```
var totalCorrect = 0;      // 正确答案的数目
```

第4行完成了两个任务：它首先定义了变量 `totalCorrect`，然后用赋值操作符`=`将这个变量赋值为0。我们要将 `totalCorrect` 的默认值设置为0，以满足用户一道题也没有答对的情况。其他变量并不需要默认值，因为在测试中都会得到明确的设置。

定义变量之后，我们调用函数 `stop()`，它将影片的播放停止在第1帧，也就是测试开始的地方：

```
// 将影片停在第1个问题上  
stop();
```

`stop()` 函数和你在 Flash 4 或者更早版本中可能使用的任何 `stop` 动作有着完全相同的效果（它将播放头停止在当前帧上）。

---

**注意：**再观察一下 `stop()` 函数调用前边的注释。这个注释用来解释它后面的代码将会产生的结果。注释是可有可无的，但如果我们在编程时暂时离开，回来时需要一个协助回忆的东西，或者我们将代码给另外的开发者看，那么它将有助于阐明代码。注释也可以让代码更易于审查，这在调试中是非常重要的。

---

既然你知道了初始化代码的作用，现在我们就可以把它添加到测试影片里去了：

1. 选择 *scripts* 层的第1帧。
2. 选择 Windows (窗口) → Actions (动作) 命令。Frame Actions (帧动作) 面板就出现了。
3. 确定你使用的是 Expert Mode，使用 Edit (编辑) → Preferences (参数选择) 命令可以让它成为一个长久的选择。
4. 进入 Frame Actions 面板的右边，输入如例 1-1 所示的初始化代码。

## 添加帧标签

我们已经完成了测试的初始化脚本和问题创建。现在应该添加一些帧标签，这样我们就能控制测试的播放。

## 变量命名风格

到现在为止，我们已经见到了一些变量名，你可能会对大写字母的使用感到疑惑不解。如果你以前从来没有编过程序，那么，一个大写字母出现在单词的中部，比如 `firstName`，或者 `totalCorrect`，这可能看起来有点奇怪。将变量名第二个词（以及后面的任何词）的首字母大写在视觉上清楚地划分开了名称内的各个词。我们使用这个技法，是因为空格和斜杠不能用在变量名里。但是，不能将变量名头一个词的首字母大写——将第一个字母大写通常用来给对象类命名，而非变量。

如果你使用下划线来代替首字母大写，以区分变量名中的词，比如 `first_name` 和 `total_correct`，这也是可以的。但是不要用 `firstName` 来命名一些变量，又用 `second_name` 来命名其他变量。只使用一种命名风格会让其他程序员觉得你的代码很容易看懂。一些语言中的变量命名是非常敏感的，也就是说 `firstName` 和 `firstname` 将被认为是两个不同的变量。但是，ActionScript 将它们看成一个东西。不过，用不同的方法来表示同一个变量是非常糟糕的，如果你将一个变量叫作 `xPOS`，那么就不要在其他地方把它叫作 `xpos`。

通常，给你的变量和函数起有意义的名称会帮助你记住它们所表示的东西。要避免使用没有意义的名字，比如 `foo`。单字母的变量，比如 `x` 或者 `i` 只用来表示简单的东西，比如循环中的循环控制变量（即计数变量）。

为了每次只引导用户做一个题目，我们已经将题 1 和题 2 的内容隔开了，分别放在第 1 帧和第 10 帧。将播放头移动到这些关键帧上，我们就可以创建幻灯片放映的效果，每个幻灯片包含一个问题。我们知道，问题 2 在第 10 帧上，因此，当我们要显示问题 2 的时候，可以调用 `gotoAndStop()` 函数，如下所示：

```
gotoAndStop(10);
```

这将使播放头前进到第 10 帧，也就是问题 2 的位置。真是一段明智的代码，对吗？错！尽管把特定的数字 10 和 `gotoAndStop()` 用在一起是有效的，但它并不具有灵活性。例如，如果我们在第 10 帧前往时间线添加了 5 帧，那么问题 2 就会突然跑到第 15 帧去，`gotoAndStop(10)` 命令就不会把用户领到正确的帧。为了让代码在时间线上的帧发生移动的情况下也能起作用，我们要用帧标签来代替帧号。帧标签是富有

表现力的名称，比如`q2`或者`quizEnd`，利用帧标签我们可以指向时间线上特定的地方。一旦标注了某个点，我们就可以用标签按照名称而非数字来指向该帧。

帧标签的灵活性是绝对的。我以前把帧号用在比如`gotoAndStop()`这样的播放头控制函数中，这非常困难。现在我们来添加测试所需的所有标签，这样，以后就能用标签来引导用户进行测试了。

1. 在`labels`层中，点击第1帧。
2. 选择 Modify（修改）→ Frame（帧）命令，Frame面板就出现了。
3. 在标签文本域中输入`init`。
4. 在`labels`层的第10帧，添加一个空白关键帧。
5. 在帧面板的标签文本域中输入`q2`。
6. 在`labels`层的第20帧，添加一个空白关键帧。
7. 在帧面板的标签文本域中输入`quizEnd`。

## 为答案按钮编写脚本

我们的问题现在已经就位了，变量也得到了初始化，帧也有了标签。如果我们要测试影片，我们会看到问题1下面的3个答案按钮在点击的时候毫无反应，并且没有办法进入到第2个题目。我们需要对答案按钮添加一些代码，这样，它们就会引导用户继续测试，并记录用户的答题情况。

为了方便起见，我们将多项选择按钮称为按钮1、按钮2和按钮3，如图1-8所示。

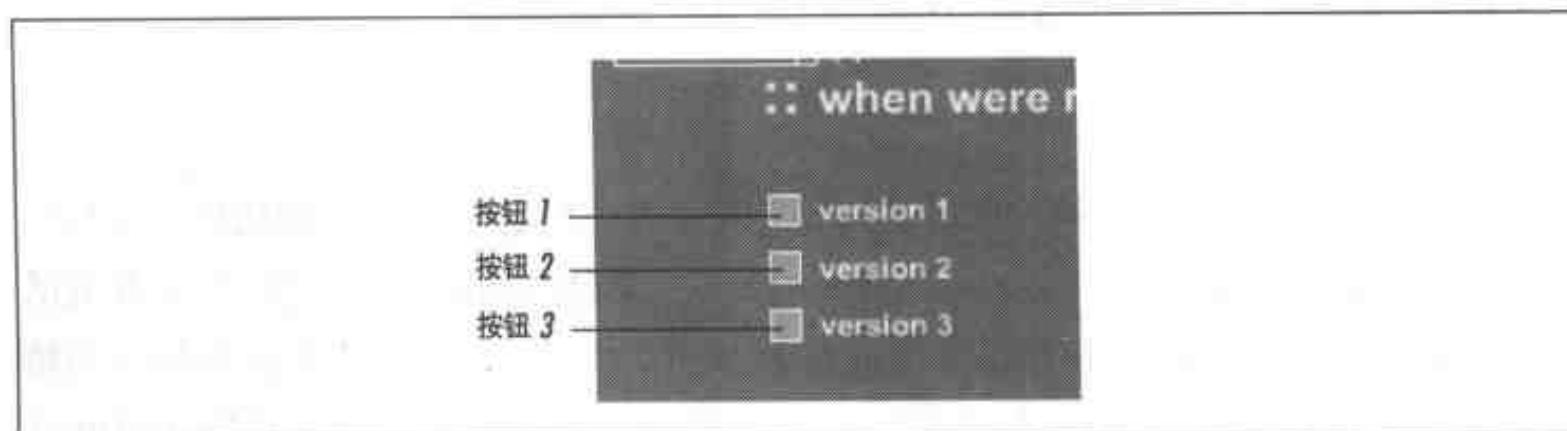


图1-8 答案按钮

我们的 3 个按钮有相似的脚本。例 1-2 到例 1-4 给出了每个按钮的代码。

#### 例 1-2：问题 1，按钮 1 的代码

```
on ( release ) {  
    q1answer = 1;  
    gotoAndStop ("q2");  
}
```

#### 例 1-3：问题 1，按钮 2 的代码

```
on ( release ) {  
    q1answer = 2;  
    gotoAndStop ("q2");  
}
```

#### 例 1-4：问题 1，按钮 3 的代码

```
on ( release ) {  
    q1answer = 3;  
    gotoAndStop ("q2");  
}
```

按钮代码包括两个语句（第 2 和第 3 行），只有当鼠标点击按钮的时候才能执行。在自然语言中，每个按钮的代码的意义为：当用户点击按钮的时候，对他选择的答案 1, 2 或 3 进行标注，然后进入第 2 道题目。下面将给出它的工作情况。

第 1 行代码是事件处理器的开始：

```
on ( release ) {
```

事件处理器耐心地等待用户点击按钮。回忆一下我们说过的，事件处理器会在影片播放的时候聆听发生的情况（比如鼠标点击）。当某个事件发生的时候，包含在响应处理器中的代码就会得到执行。

让我们剖析一下在第 1 行开始的事件处理器。关键字 *on* 标志着事件处理器的开始。（如果 *on* 这个词让你感到不方便，在你熟悉它之前你可以暂时把它当作 *when*。）关键字 *release* 被包在括号中，它表示事件处理器接收到的事件类型，在这里，我们收到的是一个 *release* 事件，当用户在按钮上点击并放开鼠标的时候这个事件就发生了。前花括号 ( ( ) 标志着 *release* 事件发生时应该执行的语句块的开始。代码块的结束由第 4 行中的后花括号 ( ) ) 来表示，这就是事件处理器的结尾。

第 2 行代码是 *release* 事件发生时所应执行的语句的开始。第 2 行的代码对你来说应该相当熟悉了：

```
q1answer = 1;
```

它将变量 q1answer 设置为 1（另外的答案按钮会将它设置为 2 或 3）。q1answer 变量存储了用户对第 1 个问题的答案。我们一旦记录了用户对第 1 道题的答案，就会通过第 3 行的按钮代码进入到第 2 个题：

```
gotoAndStop ("q2");
```

第 3 行调用 *gotoAndStop()* 函数，将帧标签 “q2” 作为参数进行传递，它将播放头推进到帧 q2，也就是第 2 道题的地方。

既然你已经了解了按钮代码的工作情况，现在我们将代码添加到问题 1 的按钮：

1. 打开 Actions 面板，选择 stage（场景）中的按钮 1。Frame Actions 面板变成了 Object Actions（对象动作）。现在，你添加的任何代码都会属于按钮 1（Stage 上选中的对象）。
2. 进入 Actions 面板的右边，输入例 1-2 中的代码。
3. 重复 1 到 2 步，对按钮 2 和按钮 3 添加代码。在按钮 2 上，将 q1answer 设置为 2；在按钮 3 上则将 q1answer 设置为 3，如例 1-3 和例 1-4 所示。

问题 2 的按钮代码和问题 1 的按钮代码在结构上是相同的（我们只消修改答案变量的名称和 *gotoAndStop()* 调用的目的地）。例 1-5 给出了问题 2 按钮 1 的代码。

#### 例 1-5：问题 2，按钮 1 的代码

```
on ( release ) {  
    q2answer = 1;  
    gotoAndStop ("quizEnd");  
}
```

我们用变量 q2answer 来代替 q1answer，因为我们要让按钮记录用户对于问题 2 的答案的选择情况。我们用 quizEnd 作为 *gotoAndStop()* 函数的参数，在用户回答完第 2 题之后将播放头推进到测试的结束（也就是标签为 quizEnd 的帧）。

我们来为问题 2 的按钮添加按钮代码：

1. 点击 *question 2* 层的第 10 帧。
2. 点击按钮 1。

3. 进入动作面板，输入例 1-5 给出的代码。
4. 重复步骤 2 和 3，为按钮 2 和按钮 3 添加按钮代码。在按钮 2 上将 q2answer 设置为 2。在按钮 3 上，将 q2answer 设置为 3。

为 6 个按钮添加完了按钮代码，你肯定会发现代码的重复性太高了。每个按钮上的代码和其他按钮代码的不同之处仅仅在于一些文字符号。这种编程的效率可不算高。我们的按钮代码大声疾呼，要求用一些集中的实体来记录答案和将测试推进到下一层。在第九章中，我们会看到如何用函数来集中代码。

## 建立测试结束部分

我们的测试差不多已经完成了。现在，我们有两个与问题跟踪脚本有关的问题，该脚本会让用户回答问题，并且通过测试让用户取得进步。我们仍然需要一个测试结束屏，以便告诉用户他的答题结果。

为了建立测试结束屏，我们需要进行一些基本的 Flash 制作，以及一些脚本的编写。首先要做的是：

1. 在 *question 2* 层的第 20 帧上，添加一个空白关键帧。这可以防止问题 2 出现在测试结束屏的内容下面。
2. 在 *quiz end* 层的第 20 帧上，添加一个空白关键帧。
3. 仍然停留在该帧上，将下面的文字放到场景中：“Thank you for taking the quiz. Your final score is: /2。”在 is 和 /2 之间确保留一定的空格距离。我们要把用户的成绩放在这里。
4. 在 *scripts* 层的第 20 帧上，添加一个空白关键帧。

这将测试结束屏的创建工作。你的结束屏将如图 1-9 所示。

现在我们来着手测试结束部分的脚本。当播放头到达我们的 *quizEnd* 帧时，我们要计算用户的成绩。播放头到了第 20 帧的时候，我们需要一个计算脚本来执行这个动作。因为放在时间线上关键帧中的任何脚本在播放头进入该帧的时候都会自动得到执行，我们只需要将计算脚本粘贴到添加的 *scripts* 层第 20 帧的关键帧中。

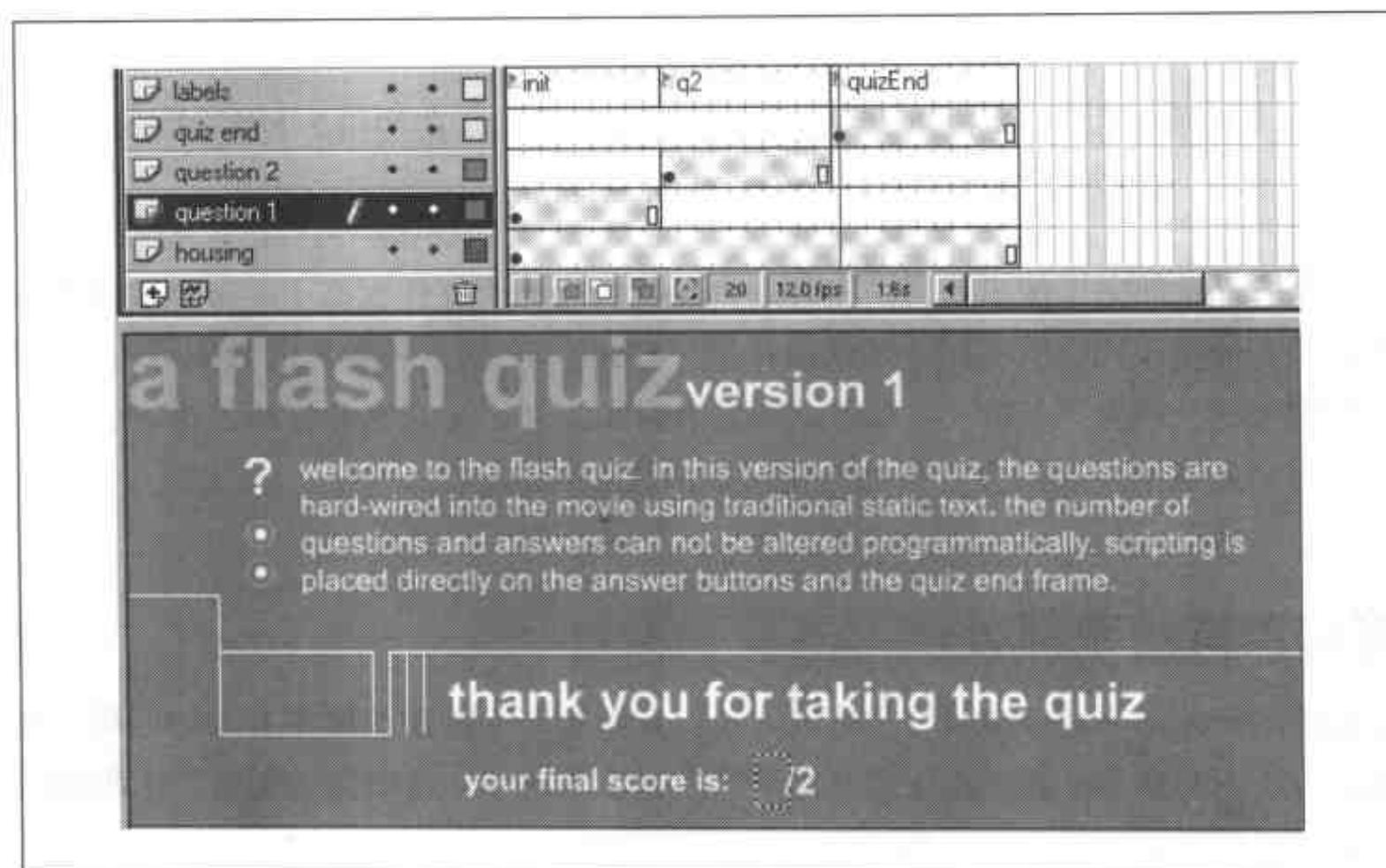


图 1-9 判决日

在计算脚本里，我们首先要确定用户的成绩，然后在屏幕上显示这个成绩：

```
// 计算用户的正确答案数
if (q1answer==3) {
    totalCorrect = totalCorrect + 1;
}
if (q2answer==2) {
    totalCorrect++;
}
// 在屏幕文字域中显示用户的成绩
displayTotal = totalCorrect;
```

第1行和第8行的注释代码说明了两个脚本部分的简要功能。第2行是计算脚本中2个条件语句中的第1个。在其中，我们使用了 q1answer：

```
if (q1answer==3){
```

关键字 if告诉解释程序，我们要提供的一系列语句只有在条件得到满足的时候才能执行。条件的内容描述在 if 关键字后面的括号中：(q1answer==3)，前花括号是条件执行语句的开始。因此，第2行的代码可以翻译为：如果 q1answer 的值等于 3，那么就执行包括在花括号中的语句。

但是条件 `q1answer==3` 究竟是如何发生作用的呢？好吧，我们将短语分开来看。我们知道 `q1answer` 是一个变量，其中存储的是用户对问题 1 的答案。而数字 3 表示问题 1 的正确答案，因为影片剪辑首先出现在 Flash 3 中。变量和数字 3 之间的双等号 (`= =`) 是相等比较操作符，它将两个表达式进行比较。如果左边的表达式 (`q1answer`) 等于右边的 (3)，我们的条件就满足了，花括号中的语句就可以得到执行。如果条件没有得到满足，花括号中的语句就被跳过。

Flash 并不知道我们的测试题目的正确答案。检查 `q1answer` 是否等于 3 是我们告诉 Flash 检查用户的第 1 道题目是否答对的方法。如果答对了，我们就告诉 Flash 要在他的总成绩上加 1，如下所示：

```
totalCorrect = totalCorrect + 1;
```

第 3 行代码的意思是：让 `totalCorrect` 的新值等于 `totalCorrect` 的旧值加上 1 (也就是说，将 `totalCorrect` 加 1)。将变量加 1 是非常常见的，它甚至有自己的特殊操作符：`++`。

因此，我们不用这样的代码：

```
totalCorrect = totalCorrect + 1;
```

我们通常这样写：

```
totalCorrect++;
```

这实际上是一样的，但是后者要简洁一点。

在第 4 行，我们结束了条件得到满足的情况下要执行的语句：

```
}
```

第 5 行到第 7 行是另外一个条件句：

```
if (q2answer==2){  
    totalCorrect++;  
}
```

在这里，我们检查用户对第 2 道题的答案是否正确 (MP3 音频支持首次出现在 Flash 4 中)。如果用户选择了第 2 个答案，我们就用变量增值操作符 `++` 将 `totalCorrect` 加 1。

因为测试中只有 2 个问题，所以我们已经完成了对用户成绩的结算。对用户答对的每一个问题，我们都要将 totalcorrect 加 1，因此，现在的 totalCorrect 就表示用户的最终成绩。剩下的惟一事情就是显示用户的成绩了，我们用测试结束脚本的最后一行，也就是第 9 行来实现：

```
displayTotal = totalCorrect;
```

你对变量已经知道得足够多了，应该懂得第 9 行的语句是将 totalCorrect 的值赋给变量 displayTotal。但是如何将成绩显示在屏幕上？到现在为止，它还不能完成显示。为了让成绩出现在屏幕上，我们需要创建一个特殊类型的变量，也就是文本域（text field）变量，它可以在屏幕上有一个物理的表现。要实现这个目的可以参照下面的方法：

1. 选择 Text（文本）工具。
2. 在 *quiz end* 层中点击第 20 帧。
3. 将你的鼠标箭头放在你先前创建的 “/2” 前面，然后点击 Stage。
4. 拖出一个文本框，其大小要足够容纳一个单一的数字。
5. 选择 Text（文本）→ Options（选项）命令。
6. 在 Text Options（文本选项）面板中，将 Static Text（静态文本）改为 Dynamic Text（动态文本）。
7. 在 Variable（变量）文本域，输入 `displayTotal`。

变量 `displayTotal` 现在在屏幕上表示出来。如果我们在脚本中改变 `displayTotal`，那么相应的文本域变量将会在屏幕上改变。

## 测试我们的测试

好，就是这样。我们的测试完成了。你现在可以用 Control（控制）→ Test Movie（影片测试）来检查测试是否运行良好。在不同的题目中点击答案，看看你的测试是否能正确地保存成绩。你甚至可以在一个新的按钮上粘贴如下的代码，创建一个重启按钮：

```
on (release) {
```

```
    gotoAndStop('init');
```

因为totalCorrect在init帧的代码中被设置为0，因此，你每次将播放头放到init帧，成绩就会自动重新设置。

如果你发现你的测试不能运行，可以和在线代码库中所提供的测试范例对照一下。你也许还想研究一下故障排除技术，这在第九章中有所描述。

## 小结

感觉如何？你已经学了很多短语、一些语法和词汇，甚至已经和Flash进行了一次很长的对话（多项选择测试）。我要说的是，语言学校内容丰富的第一天结束了。

正如你所看到的，对于ActionScript有很多要学的东西，但是，有一点点知识你也可以做很多事情了。甚至，你现在所知道的东西都已经颇够你玩味。本书余下的部分，将对你已经学到的东西进行深入的探讨，并把它们放在实际的例子中来观察，以加强你对基础知识的掌握。当然，我们也会涉及一些现在还没有接触到的概念。

记住：要考虑交流，考虑合作，言谈要清楚。如果你觉得自己在做一些特别有意思的工作或者艺术创作，你愿意和别人分享，可以将它们发给我，发到<http://www.moock.org/webdesign/flash/contact.html>。

既然你对内容目录已经有了一个实际的框架印象，那么你会在以后的章节中对基础知识有一个具体的概念。它会让你对ActionScript有更深的理解，让你能够创建更多复杂而精彩的影片。

## 第二章

# 变量

在典型的脚本化影片中，我们必须跟踪和处理所有东西，从帧号到用户的密码，其速度简直像是从宇宙飞船上发射的光子。为了管理和恢复所有的信息，我们需要将它们存储到变量（variable）中，它是 ActionScript 中重要的信息存储容器。

一个变量就如同一个银行账户，而不是手中拿着的钱，它拥有信息内容（数据）。创建一个新的变量就如同设立一项新的账户，开辟一块地方来存储以后需要用的东西。而且，正如所有的银行账户都有一个账号一样，每一个变量都有它自己的名称，这个名称被用来访问变量中的数据。

一旦创建了变量，在需要的时候就可以将新的数据放进去——正如将钱存到账下一样。或者可以用变量名找出变量中所存储的东西——正如账目结算一样。如果不再需要某个变量，可以用删除该变量的方法“关闭户头”。

要注意的关键点是：变量可让我们在播放影片的时候查看数据的变化和计算情况。正如一个银行账号在账目结算改变后仍然保持相同一样，变量名称在它所保存的数据发生改变的情况下，也仍然是不变的。使用固定的引用来访问不断变化的内容，我们就可以执行复杂的计算，比如在扑克游戏中掌握牌的情况，保存客户的账簿记录，或者基于变化的条件将播放头送到不同的位置。

这是不是让你觉得有点兴奋了？好，我想我再继续说银行，你就不想再读下去了。现在我们转入正题，看看变量的创建，以开始我们对变量的探讨。

## 创建变量（声明）

创建变量的动作称为声明（declaration）。声明在我们的银行比喻里就是“开户”，通过声明，我们让变量正式产生。当一个变量首次被声明的时候，它还是空的——一个空白的，等待落笔的页面。在这种情况下，变量包含一个特殊的值，称为 `undefined`（无定义）（表示没有数据）。

为了声明一个新的变量，我们使用 `var` 语句。例如：

```
var speed;  
var bookTitle;  
var x;
```

词语 `var` 会告诉解释程序，我们正在声明一个变量，后面的文本，比如 `speed`, `bookTitle` 或者 `x`, 就是我们所声明的新变量的名称。我们可以在能够附着代码的任何地方创建变量：在关键帧中、按钮上或者一个影片剪辑中。

我们也可以用一个单独的 `var` 语句声明数个变量，如下所示：

```
var x,y,z;
```

但是，这样做的缺点是很难在每个变量后面添加别的内容。

一旦创建了变量，我们就可以为它赋值，但是，在学习如何赋值之前，还应先考虑变量声明过程中的一些细节。

## 自动创建变量

很多编程语言都要求变量在声明之后才能够存储数据，如果不这样做就会发生错误。ActionScript 并没有这么严格。如果我们对一个并不存在的变量赋值，解释程序会为我们创建一个新的变量。我们继续拿银行作比喻，如果你第一次来存款，它会自动为你开户。

不过，这种方便是有代价的。如果我们不自己声明变量，在检查代码的时候就没有核心目录可供参考。而且，用 `var` 语句显式地声明一个变量，与让变量隐式地得到声明（也就是自动创建）有时会产生不同的结果。最安全的方式就是先声明，后使用（也就是显式声明），正如本书中所采用的那样。

## 合法的变量名

在开始创建任何变量之前，要注意变量的名称：

- 必须只由字母、数字和下划线组成。（不允许空格，连字符和其他标点符号。）
- 必须以字母或者下划线开始。
- 不得超过 255 个字符。（如果你的变量名称超出了 255 个字符，你最好重新斟酌一下命名方案。）
- 不计大小写（虽然大小写都被当作是相同的，但是你还是应该有统一的表达方式）。

下面给出的是合法的变量名称：

```
var first_name;  
var counter;  
var reallyLongVariableName;
```

下面的变量名称不合法，会引起错误：

```
var 1first_name;           // 以数字开头  
var variable name with spaces; // 包含了空格  
var another-illegal-name;    // 包含连字符
```

## 动态地创建命名变量

虽然你很少用（如果曾经用过）动态的方式来创建变量名称，但要在运行中按计划产生变量名也是可能的。为了从表达式创建变量名，可以使用 *set* 语句。例如，下面我们要把值 **bruce** 赋给 **player1name**：

```
var i = 1;  
set ("player" + i + "name", 'bruce');
```

在后面的章节中讨论的数组和对象，向我们提供了更多追踪动态命名数据的有效方式，用来代替动态的变量名称。

## 在开始处声明变量

在每个影片主脚本的开头声明变量是一个很好的习惯，也就是指影片预载入之后的

第一个关键帧。注意要为每一个变量添加注释来说明它的用途，以便你以后更容易识别。一个组织良好的脚本开头可能是这样子：

```
// 球的速度，最大值为10
// 变量初始化
// 玩家的当前成绩
// 高分（不在不同的会话之间保存）
var ballSpeed;           // 球的速度，最大值为10
var score;                // 玩家的当前成绩
var hiScore;              // 高分（不在不同的会话之间保存）
var player1;              // 第一个玩家的名字，由用户提供
```

我们可以在创建的同时给变量一个初始值，如下所示：

```
var ballSpeed=5;          // 球的速度，最大值为10
var score=0;                // 玩家的当前成绩
var hiScore=0;              // 高分（不在不同的会话之间保存）
```

## 变量赋值

现在我们到了比较有趣的一个部分——将数据放到你的变量中去。如果你仍然记得我们的银行比喻，这个过程就是“存款”。要为变量赋值，我们使用：

```
variableName = value;
```

variableName 是一个变量的名称，而 value 是我们给变量赋予的数据。下面是一个实际的例子：

```
bookTitle = "ActionScript : The Definitive Guide";
```

在等号的左边，词 bookTitle 是变量名（它的标识符）。等号的右边，短语 “ActionScript: The Definitive Guide” 是变量的值——你所存入的数据。等号本身称为赋值操作符。它告诉 Flash，你要将等号右边的东西交给（也就是存入）左边的变量。如果左边的变量并不存在，那么 Flash 就创建这个变量（虽然依靠解释程序来隐式创建变量的方式不值得推崇）。

下面是另外两个变量赋值的例子：

```
speed = 25;
output = "thank you";
```

第一个例子将整数 25 赋给变量 speed，说明变量除了能容纳文本之外同样也能存储数字。我们很快就会看到，它们也能保存其他类型的数据。第二个例子将文字“thank you”赋给变量 output。注意，在 ActionScript 中我们一直用双引号（" "）来定义文本串。

现在，我们来看看稍为复杂的例子，它用表达式  $1+5$  来为变量 y 赋值：

```
y = 1 + 5;
```

当语句  $y=1+5;$  得到执行的时候，1 首先和 5 相加，得到 6，然后将 6 赋值给 y。等号右边的表达式首先被求值（也就是计算或者说处理），然后将结果赋值给等号左边的变量。在这里，我们将包含变量 y 的表达式赋给另外的变量 z：

```
z = y + 4;
```

再次计算等号右边的表达式，并把结果赋值给 z。解释程序得到 y 的当前值（好比检查它的账目结算），然后将其加 4。因为 y 等于 6，所以 z 将会变成 10。

将任何数据（数字、文本或者其他类型）赋值给变量的过程（不管数据类型为何）都是极其类似的。例如，我们还没有学习数组，但是应该可以看出，下面是一个变量赋值语句：

```
myList = ["John", "Joyce", "Sharon", "Rick", "Megan"];
```

和以前一样，我们将变量名放在左边，赋值操作符（等号）在中间，我们需要的值在右边。

为了很快将一个相同的值赋给多个不同的变量，我们可以用以下连等的方式：

```
x = y = z = 10;
```

变量赋值通常是从右到左进行的。上面的语句将 10 赋给 z，然后将 z 的值赋给 y，再将 y 的值赋给 x。

## 变量值的修改和获取

创建了变量之后，可以在任何时间对其进行赋值和修改，如例 2-1 所示。

### 例 2-1：修改变量值

```
var firstName;           // 声明变量firstName  
firstName = 'Graham';   // 设置变量firstName的值  
firstName = 'Gillian';   // 改变firstName的值  
firstName = 'Jessica';   // 再次修改firstName的值  
firstName = 'James';    // 再次修改firstName的值  
var x = 10;              // 声明变量x，并为其赋一个数字值  
x = 'loading...please wait...'; // 将x赋为文本值
```

注意，我们将变量x的数据类型从数字修改为文本数据。只要将它赋成所要的数据类型就可以了。一些编程语言不允许变量数据类型的转换，而ActionScript则不然。

当然，如果你以后不能获取变量的值，那么创建变量和对变量赋值是没有用的。要获取变量的值，只要在需要用它的值的地方使用变量名称就可以了。变量名一出现（除了在声明中或者在等号左边），它的名称就会转变为变量的值。下面是几个简单的例子：

```
newX = oldX + 5;        // 将 newX 的值设置为 oldX 的值加上 5  
ball._x = newX;         // 将球影片剪辑的水平位置设置为 newX 的值  
trace(firstName);       // 在输出窗口中显示 firstName 的值
```

注意，在表达式ball.\_x中，ball是一个影片剪辑的名称，而.\_x表示它的x轴坐标属性（也就是在场景中的水平位置）。我们在后面要学习关于属性的更多内容。最后一行，trace(firstName)；是在脚本运行中显示一个变量的值，这有助于代码调试。

## 检查变量是否有值

有时，我们会希望在使用变量之前检查一下它是否被赋值。正如我们前边所学到的，一个得到声明但是没有赋值的变量包含特殊的“无值”属性（undefined）。要确定一个变量是否被赋值，我们可以将变量的值同 undefined关键字进行比较。例如：

```
if (someVariable!=undefined) {  
    // 放在这里的任何代码仅当 someVariable 有值的时候才能执行  
}
```

要注意不等操作符 != 的使用，它确定两个值是否不相等。

## 值的类型

我们在 ActionScript 编程中所使用的数据有各种各样的类型。到目前为止，我们已经见过了数字和文本，其他类型还包括布尔（Boolean），数组（array），函数（function）和对象（object）。在我们深入探讨每一种数据类型之前，先来看一些和变量使用有关的数据类型问题。

### 自动定型

任何 ActionScript 变量都能包含任何类型的数据，这看起来似乎只是微不足道的小事，但是，能在任何变量中存储任何类型的数据实际上非常罕见。C++ 和 Java 这样的语言使用定型变量，每一个变量只能接受一种类型的数据，这在声明变量的时候就要做出规定。ActionScript 变量是自动定型的——当我们将一个数据赋给一个变量，解释程序就会为我们设定变量的数据类型。

ActionScript 的变量不仅能包含任何数据类型，它们还能自动改变数据类型。如果我们为变量赋一个新的值，它和变量先前的值有不同的数据类型，那么变量就会自动重新定型。因此，下面的代码在 ActionScript 中是合法的：

```
x = 1;                      // x 是一个数字
x = "Michael";                // x 现在是一个串
x = [4, 6, "hello"];          // x 现在是一个数组
x = 2;                        // x 又变成了一个数字
```

在诸如 C++ 或者 Java 语言中不支持自动定型，错误类型的数据将会被转变为变量现有类型（如果不能实现转换的时候就引发错误）。自动的动态定型有一些重要的分类，我们会在后面的部分讨论。

### 值的自动转换

在一些情况下，ActionScript 希望有一个特殊的数据类型。如果我们使用的变量，其类型不符合所需要的类型，解释程序就会试图对数据进行转换。例如，如果我们在需要数字的地方使用了文本变量，解释程序就会试图将变量的文本值转化为数字值，以适应当前操作的需要。在例 2-2 中，*z* 被设置为 2，为什么？因为减号操作符需要的是一个数字，因此，*y* 的值从串 “4” 被转换为数字 4，它成为 6 (*x* 的值) 的减数，得到结果 2。

**例 2-2：由串到数字的类型自动转换**

```
x = 6;           // x 是数字 6  
y = "4";          // y 是串 "4"  
z = x - y;        // 将 z 设置为数字 2
```

相反，如果我们在需要串的地方使用了一个数字变量，解释程序就会将数字转换为串。在例 2-3 中，*z* 被设置为串 “64”，而不是数字 10。这又是为什么呢？因为表达式 *x+y* 中的第二个操作数是一个串。因此，(+) 执行的是串的连接而非数字加法。*x* 的值 (6) 被转换为串 “6”，然后和串 “4” (*y* 的值) 联结起来，得到了结果 “64”。

**例 2-3：从数字到串的类型自动转换**

```
x = 6;           // x 是数字 6  
y = "4";          // y 是串 "4"  
z = x + y;        // 将 z 设置为串 "64"
```

作为表达式一部分的变量进行求值的时候所发生的自动类型转换，是在变量数据的拷贝上执行的——它不影响原来的变量类型。变量的类型只有当变量被赋予一个和原先数据类型不同的数据时，才会改变。因此，对于例 2-2 和例 2-3 的结果，*y* 还是一个串，而 *x* 也是一个数字。

注意，第 3 行中的操作符（在例 2-2 中的 *-*，例 2-3 中的 *+*）对于 *z* 的赋值结果有着很大的影响。在例 2-2 中，串 “4” 变成数字 4，而例 2-3 中却相反（数字 6 变成了串 “6”），因为数据类型的转换规则对于操作符 *+* 和操作符 *-* 来说是不同的。我们在第三章中会讨论关于数据转换的规则，在第五章中会涉及操作符的内容。

## 手动确定数据类型

数据的自动定型和转换是非常方便的，但是正如例 2-2 和例 2-3 中所表现出来的那样，它又可能没产生期望的结果。在执行混合类型数据操作的命令之前，你也许希望用 *typeof* 操作符来确定变量的数据类型：

```
productName = 'Macromedia Flash';    // 串值  
trace (typeof productName);           // 显示: "string"
```

我们一旦知道变量的类型，就能有条件地继续下去。例如，我们在下面的代码中将要检查一个变量是否为数字类型。

```
if (typeof age == 'number') {
```

```
// 可以顺利继续  
} else {  
    trace ('Age isn't a number');           // 显示错误信息  
}
```

关于 `typeof` 的详细内容, 请参见第五章。

## 变量作用域

我们在前边已经学习了如何创建变量, 以及如何用附着在 Flash 文档主时间线上某个单独帧中的变量来获取它们的值。当一个文档包括多个帧、多个影片剪辑时间线的时候, 变量的创建和值的获取就显得有些复杂了。

为了说明原因, 我们分析几种特定情况。

### 情况 1

假设我们要在主时间线的第 1 帧上创建一个变量 `x`。创建好 `x` 之后, 我们将它的值设置为 10:

```
var x;  
x=10;
```

然后, 在接下来的一帧 (第 2 帧) 中, 我们粘贴如下的代码:

```
trace(x);
```

当我们播放影片的时候, 这些东西都能出现在输出窗口中吗? 我们在第 1 帧中创建了变量, 但是却试图在第 2 帧中获取它的值, 我们的变量还存在吗? 是, 还存在。

---

**注意:** 当我们在时间线上定义一个变量的时候, 该变量可以从该时间线上的其他所有帧中访问到。

---

### 情况 2

假设我们创建并设置了 `x`, 如同在情况 1 中所做的那样, 但是我们不打算直接将变

量设置代码放在第1帧中，而是放在第1帧中的某个按钮上。然后，在第2帧中，我们像前边那样添加同样的代码：

```
trace(x);
```

情况2的这些代码有效吗？是的。因为x被添加到了我们的按钮上，而按钮附着在主时间线上，因此，我们的变量是直接属于主时间线的。这样，我们就可以从第2帧来访问这个变量，就像前边那样。

## 情况3

假设我们在主时间线的第1帧中创建了一个名为secretPassword的变量。当影片播放的时候，用户必须输入密码才能看到影片的指定部分。

除了在第1帧声明变量secretPassword之外，我们还创建了一个函数，用它来比较用户的输入和实际的密码。代码如下所示：

```
var secretPassword;
secretPassword = 'yppah';

function checkPassword () {
    if (userPassword == secretPassword) {
        gotoAndStop ('accessGranted');
    } else {
        gotoAndStop ('accessDenied');
    }
}
```

假设我们让用户在第30帧上输入密码。她将密码输入到名为userPassword的输入文本域变量，我们在第1帧上用checkPassword()函数将输入信息同secretPassword相比较。如果我们的密码检测代码是在第1帧定义的，但是userPassword在第30帧之前并没有得到定义，那么userPassword变量在我们调用checkPassword()函数的时候是否存在呢？

答案还是：是。即使userPassword在比checkPassword()函数靠后的帧中得到定义，它仍然是同一个时间线上的一分子。

---

**注意：** 在同一个时间线上定义的任何变量，只要该时间线存在，那么它上面的任何脚本都可以访问到该变量。

---

## 变量的可访问性（作用域）

前边所列举的三种情况揭示了作用域的问题。一个变量的作用域描述的是变量在什么时候，什么地方能被影片中的代码操作。一个变量的作用域表现了它的生存范围以及对脚本中其他代码块的可访问性。要确定一个变量的作用域，我们必须回答两个问题：(a) 这个变量会存在多久？(b) 在代码的什么地方，我们能够设置和获取变量的值？

在传统的编程中，变量通常被分为两种作用域类型：全局和局部。在整个程序内都能被访问到的变量称为全局变量。只有在程序的某个部分才能访问的变量则为局部变量。虽然 Flash 支持传统的局部变量，但是它不支持真正的全局变量。我们来看一看原因所在。

## 影片剪辑变量

正如我们在前边所说的三种情况里见到的那样，在时间线上定义的变量对于该时间线上所有的脚本都是可以访问的——从第一帧到最后一个帧——不管变量的定义是在一个帧中，还是在一个按钮上。但是如果我们在一个影片中有多个时间线，如情况 4 所描述的那样，又会如何呢？

## 情况 4

假设我们有两个基本的几何形状，一个方形和一个圆形，它们是作为影片剪辑符号来定义的。

在方形剪辑符号的第 1 帧中，我们将变量 x 设置为 3：

```
var x;  
x = 3;
```

在圆形剪辑符号的第 1 帧中，我们将变量 y 设置为 4：

```
var y;  
y = 4;
```

我们将这些剪辑的实例放在影片主时间线的第 1 层第 1 帧中，将实例分别命名为 square 和 circle。

第一个问题：如果我们将下面的代码添加到影片主时间线的第1帧（square和circle已经被放在这里了），那么输出窗口中会出现什么东西？代码如下：

```
trace(x);  
trace(y);
```

答案：输出窗口中将空无一物。变量x和y是在影片剪辑的时间线上得到定义的，不是在我们的主时间线上。

---

注意：在影片剪辑时间线（比如square或者circle）上定义的变量，其作用域将被限制在该时间线上。其他时间线上的脚本（比如我们的影片主时间线）不能直接访问到它们。

---

第二个问题：如果我们将trace(x)和trace(y)语句放到square影片剪辑的第1帧上，而不是影片主时间线的第1帧，那么输出窗口的内容又将如何？

答案：会出现x的值，也就是3，但没有其他东西。x的值能够得到显示是因为x就是在square的时间线上定义的，因此能够被该时间线内的trace()命令访问到。而y的值4没有出现在输出窗口中，这是因为y是在circle中定义的，它是另外一个时间线。

你现在就明白为什么我说ActionScript不支持真正的全局变量了。全局变量是整个程序的任何地方都可以访问的变量，但是在Flash中，一个属于单独时间线的变量只能被该时间线上的脚本访问到。由于Flash中所有的变量都是定义在时间线上的，没有哪个变量能担保可以被影片中的所有脚本直接访问到。因此，没有什么变量能堂而皇之地被称为全局变量。

为了防止混淆，我们将属于时间线的变量称为时间线变量（timeline variable）或者影片剪辑变量（movie clip variable）。但是，也可以用对象（Object）类来模仿全局变量。要创建一个在所有时间线上都可以被访问到的变量，可以使用下面的语句：

```
object.prototype.myGlobalVariable = myValue;
```

例如：

```
Object.prototype.msg = "Hello world";
```

这种技术（以及它的原理）在第十二章中将有所涉及。

## 访问不同时间线上的变量

尽管一个时间线上的变量不能被其他时间线的脚本直接访问，但它们可以被间接访问。要创建、获取或者设置一个单独时间线上的变量，我们使用点号语法（dot syntax），这是一种在面向对象编程语言（比如 Java, C++ 和 JavaScript）中常见的标准符号。下面给出的是普通的点号语法用法，我们用来访问一个单独时间线上的变量：

```
movieclipInstanceName.variableName
```

也就是说，我们用包含该变量的剪辑的名称，后面跟一个点号，然后是变量名称，就可以访问其他时间线上的变量。例如在前边列举的情况下，我们可以从主时间线上访问 square 剪辑中的变量 x，如下所示：

```
square.x
```

我们同样可以从主时间线上访问 circle 剪辑中的变量 y：

```
circle.y
```

我们可以用这种参考方式，来从影片主时间线上设置和获取 square 中的变量，如下所示：

```
square.z = 5;           // 将 square 中的 z 赋为 5
var mainz;             // 在主时间线上创建 mainz
mainZ = square.z;      // 将 mainZ 赋为 square 中 z 的值
```

但是，只用 `clip.variable` 语法，我们不能从 circle 剪辑中访问 square 里的变量。如果我们在 circle 剪辑的某一帧中放上一个对 `square.x` 的引用，解释程序就会努力在 circle 中寻找一个名为 square 的剪辑，但是 square 却在主时间线上。因此，我们需要一个途径让我们能从 circle 剪辑访问包含了 square 剪辑的时间线（在本例中也就是主时间线）。这个途径来自两个特殊的属性：`_root` 和 `_parent`。

### `_root` 和 `_parent` 属性

`_root` 属性直接指向影片的主时间线。从影片剪辑结构的任何嵌套深度，我们都可以访问影片主时间线上的变量，只要使用 `_root` 即可，如下所示：

```
_root.mainZ; // 访问主时间线上的变量mainZ  
_root.firstName; // 访问主时间线上的变量firstName
```

我们甚至能够将一个指向`_root`的引用同一个指向影片剪辑实例的引用合并起来使用，以这种方式深入到影片嵌套结构的深处。例如，我们可以访问剪辑`square`中的变量`x`，而`square`在影片的主时间线上：

```
_root.square.x
```

这种引用方式在影片的任何地方都是有效的，不管剪辑嵌套有多深，这是因为引用是从影片的主时间线上开始，`_root`。这里是另外一个嵌套的例子，给出了访问`triangle`实例中的`area`变量的方法，`triangle`的位置在`shapes`实例的时间线上：

```
_root.shapes.triangle.area
```

任何从关键字`_root`开始的变量引用都称为绝对引用（absolute reference），因为它描述了我们的变量对于影片中一个固定点（主时间线）的位置。

但是，如果我们要访问其他时间线上的变量而不想涉及主时间线的时候，就会受到限制了。要想这样做，我们可使用`_parent`属性，它指向当前影片剪辑实例所处时间线的上一级时间线。例如，从`square`剪辑的某帧代码，我们可以访问包含`square`的时间线上的变量，语法为：

```
_parent.myVariable
```

从关键字`_parent`开头的引用被称为相对引用（relative reference），因为它们利用的是所在剪辑的相对位置。

回到我们前边的例子，假设我们有一个变量`size`，定义在影片的主时间线上。我们将一个名为`shapes`的剪辑放在影片主时间线上，而在`shapes`时间线上，我们定义了变量`color`。我们还在`shapes`时间线上放了一个名为`triangle`的剪辑，如图2-1所示。

要在`triangle`的时间线代码中显示变量`color`的值（它在剪辑`shapes`中），我们可以用绝对引用的方法，从主时间线开始，如下所示：

```
trace(_root.shapes.color);
```

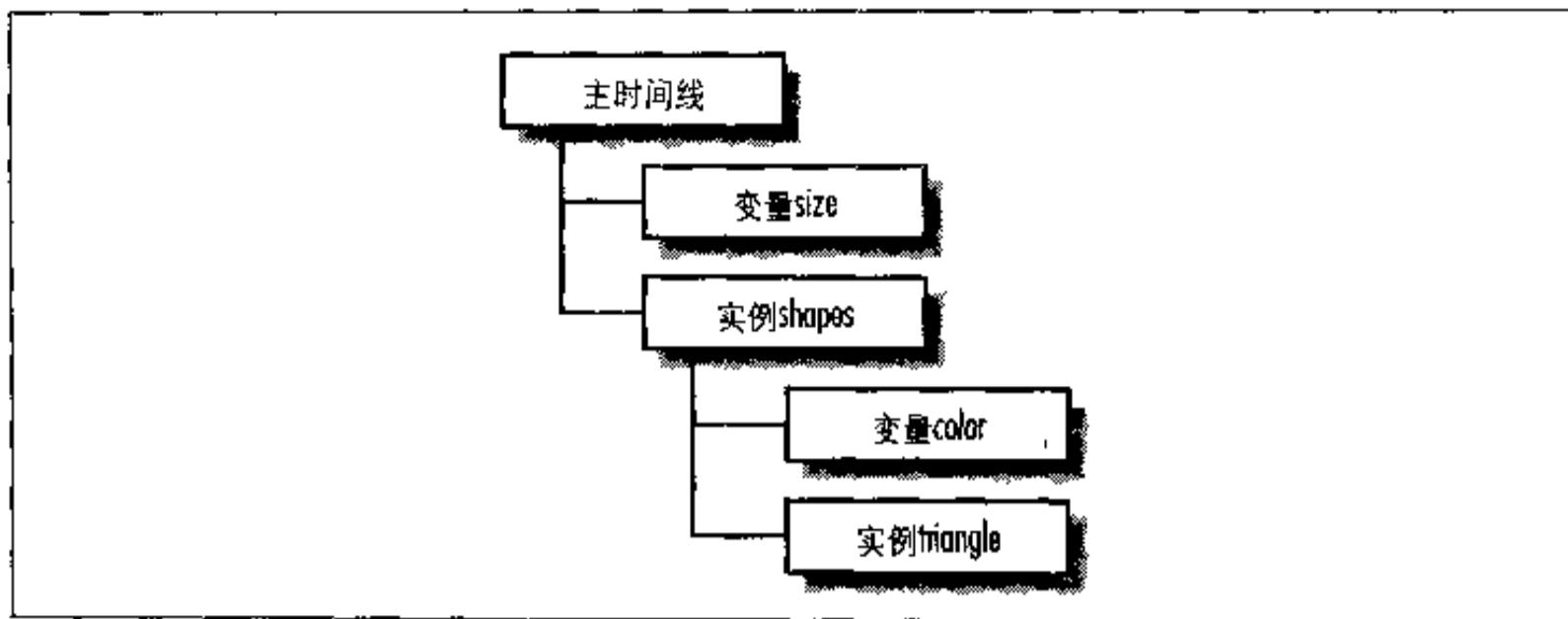


图 2-1 一个简单的影片剪辑层次图

但是这会让我们的代码徒然奔波到影片主时间线。要让代码具有灵活性，我们可以转而使用 `_parent` 属性，来创建一个相对引用，如下所示：

```
trace(_parent.color);
```

我们的第一种方法（使用 `_root`）是按照从上到下的顺序来处理的，它从主时间线开始，按照影片剪辑层次往下走，直到到达 `color` 变量。第二种方法（使用 `_parent`）却按照从下往上的顺序，它从包含 `trace()` 语句的剪辑（`triangle` 剪辑）开始，然后在剪辑结构中依次上升，最后找到 `color` 变量。

我们可以在一行代码中两次使用 `_parent`，以便可以在剪辑结构中上升到主时间线的 `size` 变量。我们把下面的代码添加到 `triangle` 中，以引用到影片主时间线的 `size` 变量：

```
trace(_parent._parent.size);
```

连续两次使用 `_parent` 属性可以让我们上升两层，在本例中也就是将我们带到了影片的主时间线上。

使用何种变量引用的方法，取决于你在不同时间线上放置影片剪辑符号实例时要达到的目的。在关于 `triangle` 的例子中，如果我们想让对 `color` 的引用总是指向在 `shapes` 剪辑中定义的 `color`，那么我们就要用 `_root` 语法，它提供了一个对 `shapes` 中 `color` 的固定引用。但是，如果我们想根据哪一个时间线包含了指定的 `triangle` 实例，而让对 `color` 的引用指向不同的 `color` 变量，我们就要使用 `_parent` 语法。

## 在不同的文件级上访问变量

`_root` 属性指向当前层级（也就是当前文档）的影片主时间线，但是 Flash 播放器在它的文档堆栈里可以容纳多个文档。载入播放器文档堆栈的任何影片主时间线都可以用 `_leveln` 来引用，在这里，`n` 是影片所处的层级数。层级数从 0 开始，比如 `_level0`, `_level1`, `_level2`, `_level3`，如此类推。关于装载多个影片的信息，可以参见第十三章。下面给出的几个例子表示的是多层级变量的地址：

```

_level1.firstName           // 在 level1 上时间线上的 firstName
_level4.ball.area           // 在 level4 上时间线中 ball 剪辑里的 area
_level0.guestBook.email     // 在 level0 上时间线上 guestBook 剪辑里的 email

```

---

**警告：**用点号语法经过影片剪辑实例时间线对变量寻址的时候，要确保你已经在场景中对剪辑实例命名了，并且在代码中引用它们的时候名称没有输入错误。如果你的实例没有被命名，你的代码就无效，即使它在语法上是正确的也不行。没有命名的实例和名称拼错的实例非常容易产生问题。

---

## Flash 4 和 Flash 5 在变量访问语法上的比较

Flash 4 风格的斜线和冒号句型，比如 `/square:area`，已经被 Flash 5 的点号语法代替了，后者在变量和时间线的引用上更为便利。旧的语法遭到打击，也不再受欢迎。表 2-1 给出了 Flash 4 和 Flash 5 在变量访问方面起相同作用的语法。附录三介绍了其他方面的语法差别。

表 2-1 Flash 4 和 Flash 5 在变量访问语法上的比较

Flash 4 语法	Flash 5 语法	指向……
<code>/</code>	<code>_root</code>	影片的主时间线
<code>/:x</code>	<code>_root.x</code>	影片主时间线上的变量 x
<code>/clip1:x</code>	<code>_root.clip1.x</code>	影片主时间线上 clip1 实例中的变量 x
<code>/clip1/clip2:x</code>	<code>_root.clip1.clip2.x</code>	在影片主时间线上包含在 clip1 实例内的 clip2 实例中的变量 x
<code>../</code>	<code>_parent</code>	当前剪辑所在时间线的上层时间线（当前剪辑时间线的上层 <sup>a</sup> ）
<code>.../:x</code>	<code>_parent.x</code>	当前剪辑所在时间线的上层时间线中的变量 x (当前时间线的上层)

表 2-1 Flash 4 和 Flash 5 在变量访问语法上的比较（续）

Flash 4 语法	Flash 5 语法	指向……
<code>.../.../x</code>	<code>_parent._parent.x</code>	包含了当前剪辑的剪辑所处时间线中的变量 x (当前剪辑时间线的上两层)
<code>clip1:x</code>	<code>clip1.x</code>	实例 clip1 中的变量 x, clip1 存在于当前时间线上
<code>clip1/clip2:x</code>	<code>clip1.clip2.x</code>	实例 clip2 中的变量 x, clip2 在 clip1 内, 而 clip1 处在当前时间线上
<code>_level1:x</code>	<code>_level1.x</code>	载入层级 1 的影片主时间线的变量 x
<code>_level2:x</code>	<code>_level2.x</code>	载入层级 2 的影片主时间线的变量 x

a. “当前剪辑时间线”是指包含变量引用代码的时间线

## 影片剪辑变量的存在期限

前面我们说过, 变量的作用域要回答两个问题: (a) 变量存在的时间有多长? (b) 我们在代码的什么地方能够设置或获取变量的值? 对于影片剪辑变量, 我们现在已经知道了包含在第二个问题答案中的东西。但是我们还没有回答第一个问题。现在我们就转回去, 看看最后一种变量编码情况。

## 情况 5

假设我们创建了一个新的影片, 它有两个关键帧。在第 1 帧中, 我们放了一个剪辑实例, ball。在 ball 时间线上, 我们创建了一个变量, radius。主时间线的第 2 帧是空的 (ball 实例还没有出现在这里)。

从影片主时间线的第 1 帧, 我们可以用下面的代码得到 radius 的值:

```
trace(ball.radius);
```

现在的问题是: 如果我们将这行代码从主时间线的第 1 帧移到第 2 帧, 那么影片播放的时候输出窗口会出现什么情况?

答案: 空无一物。当 ball 剪辑被移动到了主时间线上的第 2 帧时, 它的所有变量在此过程中都遭到了破坏。

---

注意：影片剪辑变量只存在于它们所在的剪辑处于场景上的时候。在 Flash 文档主时间线上定义的变量将存在于每一个文档之内，但是，如果文档从播放器（通过 *unloadMovie()* 函数或者因为其他影片载入到了该影片的层级中）里被卸载掉，它就会消失。

---

当包含特定影片剪辑的脚本化影片跨越了不同时间线的多个帧时，一个变量的存在期限是很重要的。在你试图使用某个剪辑中的变量之前，总是要确保你所要用的这个剪辑已经存在于时间线上。

## 局域变量

影片剪辑变量的作用域包括这个影片剪辑，并且，只要定义该变量的影片剪辑存在，变量也就存在。有的时候，它存在的时间超出了我们需要的期限。在我们只需要一个临时变量的情况下，ActionScript 提供了局域变量，它比一般的影片剪辑变量存在的时间要短很多。

局域变量用在函数中，以及旧 Flash 4 风格的子程序中。如果你以前没有接触过函数或者子程序，可以跳过本节剩余的部分，在你读了第九章之后再回头来看。

函数通常使用函数外面所不需要的变量。例如，假设我们有一个函数，它的作用是显示一个指定数组中的所有元素：

```
function displayElements (theArray) {
    var counter = 0;
    while (counter < theArray.length) {
        trace ("Element " + counter + ": " + theArray[counter]);
        counter++;
    }
}
```

counter 变量用来显示数组，但是以后就没有什么用途了。我们可以让它在时间线上定义，但这种做法却有两个缺点：(a) 如果 counter 继续存在，在影片剩余的部分它就会白白占据内存；(b) 如果 counter 在函数之外也可以被访问到，它就可能和其他名为 counter 的变量混淆起来。因此，我们希望变量 counter 在函数 *displayElements()* 结束之后就死亡。

要让 counter 在函数末尾自动消失，我们可以将它定义为局域变量。和影片剪辑变量不同，局域变量在定义它的函数结束的时候会被解释程序自动从存储器中删除（释放）。

要指定一个变量为局部变量，就要在函数内部用 *var* 关键字来定义它，如前面 *displayElements()* 的例子那样。

注意这个过程：当 *var* 语句被放在函数外面的时候，它创建的是普通的时间线变量，而不是局部变量。正如例 2-4 中所示，由于 *var* 语句所在的位置不同而产生了差别。

函数内的变量不必非得是局部的。可以通过省略 *var* 关键字从函数内部创建或者修改一个影片剪辑变量。如果不使用 *var* 关键字，而仅仅为一个变量赋值，Flash 在某些条件下就会将该变量当作一个非局部变量。在函数内部考虑这种变量赋值：

```
function setHeight () {
    height = 10;
}
```

语句 `height = 10;` 的结果依赖于 `height` 是局部变量还是影片剪辑变量。如果 `height` 在前面被定义为局部变量（不是本例中的情况），那么语句 `height = 10;` 就仅仅修改局部变量的值。如果没有名为 `height` 的局部变量，就像这里给出的例子一样，那么解释程序就会创建一个名叫 `height` 的影片剪辑变量（非局部），并将它的值设置为 10。作为一个非局部变量，`height` 在函数结束后依然存在。

例 2-4 给出了局部和非局部变量的使用情况。

#### 例 2-4：局部和非局部变量

```
var x = 5;                                // 新的非局部变量 x 的值现在为 5
function variableDemo () {
    x = 10;                                 // 非局部变量 x 的值现在为 10
    y = 20;                                 // 新的非局部变量 y 的值现在为 20
    var z = 30;                               // 新的局部变量 z 的值现在为 30
    trace(x + "," + y + "," + z);           // 将变量的值发送到输出窗口
}
variableDemo ();                           // 调用函数，显示: 10,20,30
trace(x);                                // 显示: 10 (函数中的重新赋值是持久性的)
trace(y);                                // 显示: 20 (非局部变量 y 仍然存在)
trace(z);                                // 什么也不显示 (局部变量 z 已经释放了)
```

注意，在一个脚本内同时拥有两个同名但作用域不同的变量，一个是局部的，一个是非局部的，这种情况是可能的（虽然会引起一定的混乱）。例 2-5 给出了这样的一个例子。

### 例 2-5：同名局部和非局部变量

```
var myColor = 'blue';
function hexRed () {
    var myColor = "#FF0000";
    return myColor;
}
trace(hexRed()); // 显示: #FF0000 (局部变量myColor)
trace(myColor); // 显示: 'blue' (将局部变量myColor设置为#FF0000并不会影响非局部变量)
```

### 子程序中的局部变量

虽然函数是产生轻便代码模块的首选结构，但 Flash 仍然支持 Flash 4 风格的子程序。在 Flash 4 中，一个子程序的创建可以通过将代码附着在帧上，然后添加一个标签而实现。然后，子程序可以通过调用动作而得到执行。但是在 Flash 4 中，任何在子程序中声明的变量都是非局部的，和定义它的时间线共存。在 Flash 5 中，你可以用和在函数中相同的方法在子程序中创建局部变量——使用 *var* 语句。但是，用 *var* 在子程序中定义的变量只有当子程序通过调用函数来执行的时候才能作为局部变量来创建。如果子程序帧中的脚本只作为播放头进入帧的结果而得到执行，那么 *var* 语句就声明一个普通的时间线非局部变量。不管怎么说，更为现代的函数和局部函数变量将用来代替子程序的使用。

## 应用举例

我们已经有了很多关于变量理论的知识，现在我们将这些概念付诸实际运用。在下面的例子里，我们将给出三个以变量为核心的代码范例，后面跟随的注释将对代码的用途作出说明。

### 例 2-6 为影片的播放头选择了一个随机的目的地。

#### 例 2-6：将播放头放到当前时间线的任意帧上

```
var randomFrame;           // 存储一个随机选择的帧号
var numFrames;             // 存储时间线所拥有的帧的总数
numFrames = _totalframes; // 将 _totalframes 属性赋给 numFrames

// 选择一个任意的帧
randomFrame = Math.floor(Math.random() * numFrames + 1);

gotoAndStop (randomFrame); // 将播放头放到选定的帧上
```

例 2-7 用来确定两个剪辑之间的距离。本例的有效版本可以在在线代码库中找到。

### 例 2-7：计算两个影片剪辑之间的距离

```

var c;                      // 对 circle 剪辑对象的简单引用
var s;                      // 对 square 剪辑对象的简单引用
var deltaX;                  // c 和 s 之间的水平距离
var deltaY;                  // c 和 s 之间的垂直距离
var dist;                    // c 和 s 之间的总距离

c = _root.circle;           // 获得对 circle 剪辑的引用
s = _root.square;           // 获得对 square 剪辑的引用
deltaX = c._x - s._x;       // 计算剪辑之间的水平距离
deltaY = c._y - s._y;       // 计算剪辑之间的垂直距离

// 距离就是 deltaX 的平方与 deltaY 的平方之和的根
dist = Math.sqrt ((deltaX * deltaX)+(deltaY * deltaY));

// 你也可以选择使用下面的代码，它的易读性稍差
dist = Math.sqrt (((_root.circle._x - _root.square._x) * (_root.circle._x - _root.square._x)) + ((-_root.circle._y - _root.square._y) * (_root.circle._y - _root.square._y)));

```

例 2-8 在华氏温度和摄氏温度之间进行转换。你可以在在线代码库中找到它的版本。

### 例 2-8：华氏 / 摄氏温度之间的转化

```

var fahrenheit;             // 华氏温度
var celsius;                // 摄氏温度
var convertDirection;        // 我们要转向的系统，合法的值是 "fahrenheit" 和 "celsius"
fahrenheit = 45;            // 设置一个华氏温度
celsius = 20;                // 设置一个摄氏温度
convertDirection = "celsius"; // 在本例中转换为摄氏温度
if (convertDirection == "fahrenheit") {
    result = (celsius * 1.8)+32; // 计算摄氏温度的值
    // 显示结果
    trace (celsius + " degrees Celsius is " + result + " degrees Fahrenheit.");
} else if (convertDirection == "celsius") {
    result = (fahrenheit - 32)/1.8; // 计算华氏温度的值
    // 显示结果
    trace (fahrenheit + " degrees Fahrenheit is " + result + " degrees Celsius.");
} else {
    trace ("Invalid conversion direction.");
}

```

## 小结

我们已经了解了将信息存储在变量中的有关知识，现在，我们来学习更多关于变量所存储的内容方面的知识，那就是数据。在下面的三章里，我们要学习数据究竟是什么，应如何操作，以及它为什么是我们在 ActionScript 中创建的几乎所有东西的基础。

第三章

# 数据和数据类型

我们在第二章中已经接触了变量值的内容，对脚本里所操作的数据信息也已经作了简单的介绍。在本章里，我们将深入探究关于数据的更多细节，了解 ActionScript 是如何定义、分类和存储数据的。我们还要学习如何创建和分类数据。

数据和信息

广义而言，数据（data）是能被存储在计算机中的任何东西，从文字、数字到图像、视频以及声音。所有的计算机数据都是以 0 和 1 的序列来存储的，你也许在高科  
市场材料中见到过：

数据就是原始状态下的信息——自然的、无意义的。信息，从另一个角度讲，是有意义的。例如，数字 8008898969，作为原始数据，它并没有什么意义，但是，如果我们将它作为电话号码（800）889-8969，那么数据就变成了有意义的信息。

在本章中我们会看到如何将原始的计算机数据加上一定的意义，让它成为人们能理解的信息。

# 用数据类型来保持数据的意义

怎样才能将原始数据信息存储在计算机中而不使它丧失意义？通过对数据分类，并确定其数据类型，我们就可以给予它适当的语境以规定其意义。

例如，假设我们有三个数字：5155534，5159592 和 4593030。通过数据分类——也就是说，分为电话号码、传真号码和包裹单号码——数据的语境（以及意义）就得到了保持。经过分类，每一个原来并没有什么特征的七位数字就变得有意义了。

编程语言用数据类型 (datatype) 来表示数据的基本种类。例如，几乎所有的编程语言都定义了数据类型来存储和操作文本（也就是串）和数字。要区别不同的多个数字，我们可以使用构造良好的变量名称，比如 `phoneNumber` 和 `faxNumber`。在更复杂的情况下，我们可以用对象和对象类来创建自定义的数据类型，这会在后面讲到。在考虑数据的分类之前，我们先看看 ActionScript 中包含哪些数据类型。

## ActionScript 的数据类型

编程的时候，我们可能想存储一个产品名称，一个背景颜色，或者放在夜空中的星星数量。我们用下面的 ActionScript 数据类型来存储数据：

- 对于诸如“hi there”之类的文本序列，ActionScript 提供了串(string)数据类型。一个串就是一个字符序列（包括文字、数字和标点）。
- 对于数字，比如 351 和 7.5，ActionScript 提供了数字(number)类型。数字用在计算和数学方程中。
- 对于逻辑判断，ActionScript 提供了布尔类型 (Boolean)。用布尔数据，我们可以描述或记录一些条件或比较结果的状态。布尔数据有两个基本的值：`true` 和 `false`。
- 要描述空 (absence) 的数据，ActionScript 提供了两个特殊的数据值：`null` 和 `undefined`。你可以将它们看成是 `null` 和 `undefined` 数据类型所容许的值。
- 对于单个数据的列表，ActionScript 提供了数组 (array) 数据类型。
- 对于影片剪辑实例的操作，ActionScript 提供了影片剪辑 (movieclip) 数据类型。

- 最后，对于任意的内置的或用户自定义的数据类，ActionScript 提供了强有力的对象（object）数据类型。

我们存储在 ActionScript 中的每一个数据都会属于其中的某个类型。在进入第四章，学习每一种数据类型之前，我们要看看数据使用效果方面的常见问题。

## 数据的创建和分类

在 ActionScript 中创建一个新的数据有两种方法，这两种方法都需要用到表达式——在脚本里用来表示数据的代码短语。

一个直接量表达式（literal expression）（或者简单叫作直接量）是一个字母、数字和标点数据的序列。数据直接量则是程序源代码中对数据的逐字描述。这和变量不同，变量主要是数据的容器。每种数据类型对直接量的创建制定了它自己的规则。下面是一些直接量的例子：

```
"loading...please wait"    // 一个串直接量  
1.51                      // 一个数值直接量  
['jane', "jonathan"]      // 一个数组直接量
```

注意，影片剪辑不能够用直接量来表示，而要通过实例名来引用。

我们也可以用复杂表达式（complex expression）来计划性地产生数据。复杂表达式所表示的数据是一个代码短语，它的值必须要经过计算才能得到，并不是直接量。计算出来的值就是它所表达的数据。例如，下面每一个复杂表达式都可以得出一个单一的数据：

```
1999 + 1                  // 得到数据 2000  
"hi " + "ma!"             // 得到数据 "hi ma!"  
firstName                 // 得到变量 firstName 的值  
_currentframe              // 得到播放头当前所处位置的帧号  
new Date()                 // 得到关于当前日期和时间的新数据对象
```

注意，一个单一的直接量表达式，比如 1999 或 1 可以成为一个更大的复杂表达式（*1999+1*）的一部分。

不管我们用直接量表达式还是复杂表达式来创建数据，我们都必须存储以后要用到

的所有数据。表达式 "hi" + "ma!" 的结果是不为人知的，除非我们将它存储到变量里。例如：

```
// 这个数据是转瞬即逝的，在它创建之后就马上消失  
hi + "ma!";  
  
// 这个数据存储在变量里，以后可以通过变量 welcomeMessage 来访问它  
var welcomeMessage = "hi" + "ma!";
```

我们如何将数据划分到恰当的类型中呢？也就是说，我们如何指定某个数据是一个数字、一个串、一个数组或者其他别的？通常，我们不能自己对新的数据进行分类，ActionScript解释程序基于一系列的内部规则会自动赋值或者判断每个数据的类型。

## 直接量的自动定型

解释程序通过检查语法来判断一个直接量数据的类型，正如如下的代码段中注释所说明的一样：

```
'animal'           // 从引号上确定 'animal' 是一个串  
1.35               // 如果它只包含整数和小数，就是一个数字  
true                // 特殊关键字 true 判断它是一个布尔值  
null                // 特殊关键字 null 确定这是一个 null 类型  
undefined           // 特殊关键字 undefined 确定这是一个 undefined 类型  
  
["hello", 2, true]   // 方括号以及被逗号分隔开的值表示这是一个数组  
  
(x:234, y:456)      // 花括号以及被逗号分隔开的属性名 / 值对表示这是一个对象
```

正如你所看到的，使用正确的数据直接量语法是非常重要的。不正确的语法会产生错误，或者产生对数据内容的曲解。例如：

```
animal             // 没有引号的 animal 被当作一个变量，而不是串或文字  
"1.35"            // 引号里的数字被当作串，而不是数字  
1. 35              // 3 前边的空格将产生错误  
"animal"           // 没有后引号，将产生错误
```

## 复杂表达式的自动定型

解释程序计算一个表达式的值，以确定它的数据类型。考虑下面的例子：

```
pointerX = _xmouse;
```

因为`_xmouse`存储的是鼠标指示器当前位置的数值，因此表达式`_xmouse`的类型是数字，那么变量`pointerX`也将是一个数字。

通常，解释程序自动匹配我们所期望的东西，从而确定数据类型。但是，在一些暧昧的情况下，我们需要了解解释程序用来确定表达式数据类型的规则（参见例 2-2 和例 2-3）。考虑下面的表达式：

```
'1' + 2;
```

`+`号左边的操作数是一个串（“1”），而后边的操作数却是数字（2）。`+`操作符对数字（相加）和串（连接）都能进行操作。表达式“1”+2 的值应该是数字 3 呢还是串“12”？要排除这种不明确性，解释程序依靠一个固定的规则：加操作符（`+`）总是先匹配串，因此，表达式“1”+2 就产生了串“12”，而不是数字 3。这个规则是武断的，但它提供了一个稳定的方法来解释代码。这个规则的选取是基于加操作符的典型用途：如果有一个操作数是串，那么很可能你想做的事情是把操作数连接起来，而不是进行数字加法。例如：

```
trace ("The value of x is:" + x);
```

合并不同类型的数据，或者在某语境中使用了不匹配的数据类型都会造成不明确的状况。这将迫使解释程序自动按照武断然而可预测的规则进行数据类型的转换。我们来看一看发生自动转化的情况，以及将数据类型进行转化会产生的结果。

## 数据类型转换

我们来深入研究一下上一节中给出的例子。在那个例子中，每一个数据——“1”和 2——都有属于自己的数据类型，第一个数据是串，而第二个是数字。我们看到，解释程序将两个值连接到一起，形成了串“12”。注意，解释程序首先必须将数字 2 转化为串“2”。仅当执行自动转换之后，值“2”才能连接到串“1”上。

数据类型转换表示改变数据的类型。并非所有的数据类型转换都是自动完成的，我们也可以明确地改变数据类型，否则 ActionScript 就将实现默认数据定型。

## 自动的类型转换

只要我们在语境中使用了数据类型不匹配的值，解释程序都会试图进行转换。也就是说，如果解释程序要的是数据类型 A，而我们提供的数据却属于类型 B，那么解释程序就会把类型 B 的数据转换为类型 A 的数据。例如，在下面的代码中，我们用串“Flash”来作为减号操作符右边的操作数。由于只有数字可以进行减法运算，解释程序就会将串“Flash”转换为数字：

```
999 - "Flash";
```

当然，串“Flash”并不能成功地转换为合法数字，因此，它就被转换为特殊的数字值 NaN（也就是非数字）。NaN 是数字类型的一个合法值，专门用来应付这样的情况。“Flash”被转换为 NaN，我们的表达式对解释程序来说就成了这个样子（虽然我们永远看不到内在的步骤）：

```
999 - NaN;
```

减操作符的两个操作数现在都是数字了，因此运算可以继续进行：999-NaN 产生了值 NaN，它就是表达式的最终值。

产生数字值 NaN 的表达式根本没用，大部分的转换都能产生更有意义的结果。例如，如果一个串只包含数字字符，它就能被转化为有效的数字。表达式：

```
999 - "9"; // 数字 999 减串 "9"
```

内部程序将其转化为：

```
999 - 9; // 数字 999 减 9
```

计算表达式的时候就可以得到值 990。自动转换对于加操作符、等号操作符、比较操作符，以及条件和循环语句来说是很常见的。为了确保进行自动转换处理的任何表达式能产生有效的结果，我们必须回答三个问题：(a) 当前语境所需要的数据类型是什么？(b) 在该语境中使用了一个不匹配的数据类型会如何？(c) 转换发生的时候，得到的值将是什么？

要回答第一个和第二个问题，我们需要在本书其他的地方探讨一些相关的主题（比如，要确定在条件语句中需要何种数据类型，可以参见第七章）。

下面的三个表列出了自动转换的规则，回答了第三个问题，“转化发生的时候，得到的值将是什么？”表 3-1 给出了每一种数据类型转换为数字将得到的结果。

表 3-1 转换为数字

原来的数据	转换结果
undefined	0
null	0
布尔	如果原值为 true 则得到 1，如果原值为 false 则得到 2
数字串	如果该串只由十进制数字、空格、指数、小数点、加号或者减号（例如，“-1.485e2”）组成，则得到相同的数字值。
其他串	空串、非数字串，以及由 "x"，"0x" 或者 "FF" 开始的串都转换为 NaN
"Infinity"	Infinity (正无穷)
"-Infinity"	-Infinity (负无穷)
"NaN"	NaN
数组	NaN
对象	对象的 <i>valueOf()</i> 方法所返回的值
影片剪辑	NaN

表 3-2 给出的是将每一种数据类型转换为串时所能得到的结果。

表 3-2 转换为串

原来的数据	转换结果
undefined	" " (空串)
null	"null"
布尔	如果原值为 true 则得到 "true"，如果原值为 false 则得到 "false"
NaN	"NaN"
0	"0"
Infinity	"Infinity"
-Infinity	"-Infinity"
其他数字值	等价于数字的串。例如，944.345 将成为 "944.345"

表 3-2 转换为串 (续)

原来的数据	转换结果
数组	一个以逗号分隔的元素值列表
对象	调用对象的 <i>toString()</i> 而得到的结果值。在默认情况下，一个对象的 <i>toString()</i> 方法会返回 "[Object Object]"。 <i>toString()</i> 方法可以进行自定义，以返回更有意义的结果（例如，一个数据对象的 <i>toString()</i> 返回："Sun May 14 11:38:10 EDT 2000"）
影片剪辑	影片剪辑实例的路径，给出的是绝对形式，从在播放器中的文档层级开始。例如，"_level0.ball"

表 3-3 给出了将各种数据类型转换为布尔类型所产生的结果。

表 3-3 转换为布尔类型

原来的数据	转换结果
<code>undefined</code>	<code>false</code>
<code>null</code>	<code>false</code>
<code>NaN</code>	<code>false</code>
<code>0</code>	<code>false</code>
<code>Infinity</code>	<code>true</code>
<code>-Infinity</code>	<code>true</code>
其他数字值	<code>true</code>
非空串	如果该串能够被转换为有效的非零数字就得到 <code>true</code> ，否则得到 <code>false</code> ；在 ECMA-262 中，一个非零串总是转换为 <code>true</code> (Flash 5 违背这个规则，以维持和 Flash 4 的兼容性)
空串	<code>false</code>
数组	<code>true</code>
对象	<code>true</code>
影片剪辑	<code>true</code>

## 显式的类型转换

如果自动（隐式的）类型转换规则不合乎你的意图，我们可以手工（显式的）改变一个数据的类型。我们做了这项工作之后，必须记住，在前面的表中所列举出来的规则仍然适用。

### 用 `toString()` 方法转换为串

我们可以调用 `toString()` 方法，将任何数据转换为串。例如：

```
x.toString();      // 得到变量x的串值  
('523').toString(); // 返回 "523"。注意，我们用括号是为了防止 '.' 被当作小数点
```

我们对一个数字调用 `toString()` 方法的时候，还可以提供一个数字参数，用来表示串经过转换后所表示的数字应该使用的基数。这为十六进制、十进制和八进制之间的转换提供了便利的途径。例如：

```
var myColor = 255;  
var hexColor = myColor.toString(16);      // 将hexColor设置为 "FF"
```

### 用 `String()` 函数转换为串

`String()` 函数和 `toString()` 方法有同样的结果，但它们的语法有所不同。

```
String(x);    // 将x转换为串  
String('523'); // 将523转换为串 "523"
```

不要将全局函数 `String()` 和同名的内置类构造器混淆起来。这两者都在第三部分中有相关描述。

### 用和空串串联的方法转换为串

因为加操作符 (+) 在自动转换规则中更趋向于转换为串，因此，我们可以给任何数据连一个 "" 空串，就可以把该数据转换为串了。

```
x + "";      // 将x转换为串  
523 + "";    // 将523转换为串 "523"
```

## 用 Number() 函数转换为数字

正如 *String()* 函数将数据转换为串类型一样, *Number()* 函数也可以将它的参数转换为数字类型。如果不能转换为一个实数, 那么 *Number()* 函数就会返回一个特殊的数值值, 正如表 3-1 中所给出的那样。下面是一些例子:

```
Number(age);           // 产生转换为数字的 age 值  
Number('29');         // 产生数字 29  
Number('sara');        // 产生 NaN
```

不要把全局函数 *Number()* 和同名的内置类构造器混淆起来。第三部分中将对两者进行探讨。

因为用户在屏幕文本域输入的内容总是属于串类型, 那么在执行数字计算的时候就有必要将文本域转换为数字。例如, 如果我们想要得到文本域 *price1* 和 *price2* 的总和, 就使用下面的代码:

```
totalCost = Number(price1) + Number(price2);
```

否则, *price1* 和 *price2* 将会作为串而不是数字来相加。有关文本域的更多信息, 请参见第十八章。

## 通过减 0 而转换为数字

要让解释程序将一个数据转换为数字, 我们可以从这个数据上减 0。这里同样要遵守表 3-1 所列出来的转换规则:

```
'953' - 0          // 产生 953  
'molly' - 0        // 产生 NaN  
x - 0              // 产生 x 转换为数字后的值
```

## 用 *parseInt()* 和 *parseFloat()* 函数转换为数字

*parseInt()* 和 *parseFloat()* 函数将包含数字和字母的串转换为数字。*parseInt()* 函数提取出现在串中的第一个整数, 表示串的第一个非空字符是一个合法的数字字符。否则, *parseInt()* 就产生 NaN。用 *parseInt()* 进行的数字提取以串中第一个非空字符开始, 以第一个非数字字符或者小数点之前的字符为结束。

下面是一些 *parseInt()* 的示例：

```
parseInt('1a')           // 提取 1  
parseInt('1.3a')         // 提取 1  
parseInt(' 1a')          // 提取 1  
parseInt('1 ar 14 years old') // 产生 NaN (第一个非空字符不是数字)  
parseInt('14 years old') // 提取 14
```

*parseFloat()* 函数提取出现在串中的第一个浮点数，表示串中的第一个非空字符是有效的数字字符。（浮点数字就是包含小数值的正数或负数，比如 -10.5 或 345.678。）和 *parseInt()* 相似，如果串中第一个非空字符不是有效的数字字符，*parseFloat()* 就产生特殊的数字值 NaN。用 *parseFloat()* 提取的数字是一系列的字符，从串中第一个非空字符开始，以第一个非数字字符（除了 +、-、0-9 或小数点之外的任何字符）前边的字符结束。

下面是一些 *parseFloat()* 的示例：

```
parseFloat('1.3a');        // 提取 1.3  
parseFloat('2.75 years old'); // 提取 2.75  
parseFloat('Ince upon a time'); // 提取 1  
parseFloat('T m 3.5 zee+ tall'); // 产生 NaN
```

有关 *parseInt()* 和 *parseFloat()* 的更多信息，请参见第三部分。

## 转换为布尔类型

我们要将数据转换为布尔类型的时候，可以使用全局 *Boolean()* 函数，它的语法与 *String()* 和 *Number()* 函数的语法相似。例如：

```
Boolean(5); // 结果是 true  
Boolean(x); // 将 x 的值转变为布尔值
```

不要将全局 *Boolean()* 函数和同名的内置类构造器混淆起来。第三部分有关于这两者的描述。

## 转换期限

所有在变量、数组元素和对象属性上进行的类型转换都是临时的，除非转换是赋值的一部分。下面，我们来看一个临时转换的例子：

```

var x = "20";           // x 是一个串
y = x - 5;             // 现在是 5, x 的值被转换为数字
trace(typeof x);       // 显示 "string", 转换是临时性的

```

下面我们所看到的永久性转换是赋值的结果：

```

x = "10";              // x 是一个串
x = x-5;               // x 被转换为数字
trace(typeof x);        // 显示 "number", 转换是永久性的, 因为它是赋值的一部分

```

## 从 Flash 4 到 Flash 5 的数据类型转换

在 Flash 4 中，串操作操作符和数字操作符是截然不同的——一套操作符只对数字起作用，而另外一套操作符只对串进行操作。例如，Flash 4 中的串连接操作符是 &，但是数字加法操作符是 +。与此相似，串的比较是用 eq 和 ne 操作符来操作的，而数字的比较则用 = 和 <> 来完成。表 3-4 列出了和 Flash 4 操作符功能类似的 Flash 5 的语法。

表 3-4 Flash 4 和 Flash 5 操作符的比较

运算	Flash 4 语法	Flash 5 语法
串连接	&	或 add
串的相等	eq	==
串的不等	ne	!=
串比较	ge, gt, le, lt	>=, >, <=, <
数字加法	+	+
数字相等	=	==
数字的不等	<>	!=
数字的比较	>, >, <=, <	>=, >, <=, <

Flash 5 中的一些操作符可以操作串和数字。例如，操作串的时候，+ 操作符将其操作数连接成一个新的串。但是操作数字的时候，+ 操作符将两个操作数用数学计算的方式相加。类似的，Flash 5 中的等于操作符（==）和不等操作符（!=）可以用来对串、数字和其他数据类型进行比较。

因为许多 Flash 5 操作符都可以操作多种数据类型，而 Flash 4 操作符却不然，所以如果把 Flash 4 文件导入到 Flash 5 中的时候就会出现不确定的情况。因此，导入 Flash

4 的时候，Flash 5 在如下不明确的操作符所操作的任何操作数数据上自动使用 *Number()* 函数（除非操作数是数字直接量）：

```
+, --, !=, <, >, ==, !=, &
```

转化到 Flash 5 的 Flash 4 文件也将串连接操作符 (&) 转变为新的 *add* 操作符。表 3-5 列出了 Flash 4 操作符转换为 Flash 5 操作符的例子。

表 3-5 Flash 4 到 Flash 5 的操作符转换示例

Flash 4 语法	Flash 5 语法
loop While (count<=numRecords)	while (Number(count)<=Number(numRecords))
If (x=15)	if (Number(x) == 15)
If (y<>20)	if (Number(y) != 20)
Set Variable:"lastName"="kavanagh"	lastName = "kavanagh"
Set Variable:"name"="molly" & lastName	name = "molly" add lastName

## 确定现有数据的类型

在继续执行代码之前要确定给定的表达式所包含的数据类型，我们使用 *typeof* 操作符，如下所示：

```
typeof expression;
```

*typeof* 操作符返回一个串，告诉我们 *expression* 的类型，以表 3-6 中所列的规则为准。

表 3-6 返回 *typeof* 的值

原数据类型	<i>typeof</i> 返回的值
数字	"number"
串	"string"
布尔	"boolean"
对象	"object"
数组	"object"
null	"null"

表 3-6 返回 `typeof` 的值（续）

原数据类型	<code>typeof</code> 返回的值
影片剪辑	"movieclip"
函数	"function"
<code>undefined</code>	"undefined"

下面是一些例子：

```
trace(typeof "game over");           // 在输出窗口中显示: "string"
var x=5;
trace(typeof x);                     // 显示: "number"
var now = new Date();
trace(typeof now);                  // 显示: "object"
```

正如例 3-1 所示，和 `for-in` 语句合并的时候，`typeof` 提供了一个方便的方法来找到时间线上的所有影片剪辑实例。一旦确定，我们就可以将剪辑赋给任何数组，以进行计划性的操作。（如果你不能完全理解例 3-1，那么可以在看完第一部分之后再回来。）

### 例 3-1：用动态确定的影片剪辑来填充数组

```
var childClip = new Array();
var childClipCount = 0;

for (i in _root) {
    thisItem = _root[i];
    if(typeof thisItem == 'movieclip') {
        // 注意增量操作符的使用
        childClip[childClipCount+1]=thisItem;
    }
}

// 既然我们已经填充了数组，就可以使用它来操作其中包含的剪辑
childClip[0]._x = 0;      // 将第一个剪辑放在场景的左边
childClip[1]._y = 0;      // 将第二个剪辑放在场景的顶端
```

## 原始数据和复合数据

到现在为止，我们主要在讨论数字和串，它们是最基本的原始数据类型。原始数据

类型是语言的基础单元，每一个原始数值都包含一个单一的数据（和多项目的数组相对），并严格地对数据作出描述。原始数据是非常直接的。

ActionScript 支持这些原始数据：数字、串、布尔、*undefined* 和 *null*。ActionScript 没有 C/C++ 中那种个别的单字符数据类型（比如 *char*）。

顾名思义，原始数据类型非常简单。它们只保存文字信息、帧号、影片剪辑尺寸值等，但是它们不能够容纳更复杂的层次。对于更精细的数据处理——比如模仿一打弹球的物理运动，或者管理一个有 500 道题目和答案的测试——我们只能使用复合数据。利用复合数据，我们可以管理多个相互联系的数据，如同一个单一的数据一样。

ActionScript 支持下面的复合数据类型：数组、对象和影片剪辑。函数在理论上是一类对象，因此也可以作为复合数据类型，但是我们很少这样处理它们。第九章中可以看到更多关于函数数据类型的内容。

一个单一的数字是原始数据，而多个数字构成的列表（也就是数组）是复合数据。下面将对复合数据的用途给出一个实际的例子：假设我们要追踪一个名为 Derek 的消费者的简介。我们可以创建一系列变量来存储 Derek 的特征，每个特征是一个原始数据，如下所示：

```
var custName = "Derek";
var custTitle = "Coding Genius";
var custAge = 30;
var custPhone = "416-222-3333";
```

但是，这种格式在我们试图添加更多用户的时候会变得非常讨厌。我们被迫继续使用命名的变量来存储所有的东西——*cust1Name*, *cust2Name*, *cust1Title*, *cust2Title*, 等等。讨厌！但是如果我们将使用了数组，存储信息的时候就灵活多了：

```
cust1 = ["Derek", "Coding Genius", 30, "416-222-3333"];
```

如果我们要添加更多的用户，就只需要创建新的数组：

```
cust2 = ["Komlos", "Comic Artist", 28, "515-515-3333"];
cust3 = ["Porter", "Chef", 51, "515-999-3333"];
```

这样显得非常简洁。我们在以后的章节里将会学习有关复合数据类型的更多内容。

## 小结

我们已经介绍了 ActionScript 中的数据，下面准备进行更深入的探讨。在第四章中，我们要学习数字、串、布尔、*undefined* 和 *null* 数据类型。在第五章中，我们要学习如何操作数据。在以后的章节里，我们会学习复杂数据类型，比如影片剪辑、数组和对象。

---

## 第四章

# 原始数据类型

原始数据由简单的字符或者关键字所组成，比如数字 0, 1, 2, 3, 4, 5, 6, 7, 8, 9，或者字串 “a”, “b”, “c”。我们在前一章已经学到，ActionScript 所支持的原始数据类型包括数字、串、布尔、*undefined* 和 *null*。在本章里，我们要学习如何定义、检查和修改每一种类型的数据。

## 数字类型

数字用来进行数学计算，保存影片里的数字属性（比如影片剪辑的当前帧或者它在场景中的位置）。我们来看一下在 ActionScript 中如何对数字进行定义和操作。

## 整数和浮点数字

大部分编程语言都可以区分两种类型的数字：整数（integer）和浮点数（floating-point number）。整数是没有小数部分的完整数字。整数可以是正的，也可以是负的，还包括数字 0。浮点数字（可以简称浮点）可以包括小数点后面的小数部分，比如 0.56, 199.99 和 3.14159。因此，1, 34523, -3, 0 和 -9999999 是整数，而 223.45, -0.56 和  $1/4$  是浮点数。

## 数值直接量

我们在前边已经学过，一个直接量就是一个单一的、固定的数据值的直接表达。数字类型支持三种直接量：整数直接量、浮点直接量和特殊数字值。前两种直接量表示实数（有固定数值的数），第三种类型的常量包含那些表示数字概念的值，比如正无穷。

### 整数直接量

整数直接量，比如 1, 2, 3, 99 和 -200，大都遵循下面的规则：

- 整数不可以包含小数点或者小数值。
- 整数不能超过 ActionScript 所规定的最大和最小合法数值范围。你可以在第三部分的合法值讨论中看到数字对象的 MIN\_VALUE 和 MAX\_VALUE 属性。
- 十进制的整数不能以 0 作为开始（比如 002, 000023 和 05）。

并非所有的整数值都是十进制的。ActionScript 还支持八进制和十六进制的数字直接量。要了解十进制、八进制和十六进制的基本知识，可以访问下面的网站：

<http://www.moock.org/asdg/technotes>

我们用打头的零来表示八进制数字。例如，要表示八进制数 723，我们在 ActionScript 中就要这样写：

```
0723 // 也就是十进制中的 467 (7*64 + 2*8 + 3*1)
```

要表示十六进制整数直接量，我们在数字前边放上 0x，如下所示：

```
0x723 // 十进制中的 1827 /*256 + 2*16 + 3*1)  
0xFF // 十进制中的 255 (15*16 + 15*1)
```

十六进制数通常用来表示颜色值，但是大部分简单的程序都只需要十进制数。如果你要将串转换为数字，要注意去掉打头处不必要的 0，如例 4-1 所示。

#### 例 4-1：去除打头 0

```
function trimZeros(theString) {  
    while (theString.charAt(0) == "0" || theString.charAt(0) == " ") {
```

```
    theString = theString.substring(1, theString.length);
}
return theString;
}

testString = "00377";
trace(trimZeros(testString)); // 显示: 377
```

## 浮点直接量

浮点直接量表示的数字包含有小数部分。一个浮点直接量可以包含下面四种成分中的部分或者全部：

十进制整数

小数点 (.)

小数 (以十为基数)

指数

前三种成分是非常直接的：在数字 3.14 中，“3”是十进制整数，“.”是小数点，而“14”是小数部分。第四种成分（指数）还需要进一步研究。

要将一个非常大的整数或者负数表示为浮点数，我们可以用字符 E (或 e)，为数字添加一个指数。要确定带指数的数字值，需要乘以指数所给定的 10 的幂。例如：

```
12e2 // 1200 (10的平方是100，再乘以12就得到了1200)
143E-3 // 0.143 (10的-3次方是.001，再乘以143就是0.143)
```

你会发现这其实就是标准科学计数法。如果数学不是你的强项，这里可以提供一个简单的转换技巧：如果指数是正数，那么就将小数点向右移动相应的位数；如果指数是负数，则将小数点向左移动相应的位数。

有的时候，如果结果太大或者太小，ActionScript 就会返回一个带指数的数字作为计算结果。注意，虽然如此，指数 E (或 e) 只不过是一个简便的符号而已。如果你想以任意的幂次增大一个数字，可以使用内置函数 *Math.pow()*，这将在第三部分中介绍。

## 浮点精度

Flash 使用双精度浮点数，它提供大约 15 位精度的有效数字。（任何打头的或结尾的

0，以及指数，都不属于这15个位数的范围。) 这表示：Flash可以表示数字123456789012345，但是不能表示1234567890123456。精度不限制数字的大小，但限制数字表达的精确程度。2e16比123456789012345大，但是它只精确到一位有效数字。

ActionScript的计算有时按照不理想的舍入法来进行，得到诸如0.14300000000000001这样的数字，而不是简单的0.143。发生这种情况是因为当计算机将任何进制的数字转化为内部的二进制表示时，会产生二进制的无穷小数（就象十进制里的0.3333333）。计算机的精度是有限的，它们不能精确地表示无穷小数。如果其差异会影响你的代码效果，你应该手动地进行舍入，以调整这种差异。例如，在下面的例子中，我们将myNumber舍入为小数点后三位：

```
myNumber = Math.round (myNumber * 1000) / 1000;
```

下面是一个可以重复使用的函数，用来将任何数字舍入为有任意个小数位的数字：

```
function trim(theNumber,decPlaces) {  
    if(decPlaces > 0){  
        var temp=Math.pow(10,decPlaces);  
        return Math.round(theNumber * temp)/temp;  
    }  
}  
  
// 将一个数字舍入为小数点后两位  
trace(trim(.12645,2)); // 显示: 0.13
```

## 数字类型的特殊值

整数和浮点常量几乎占据了数字类型的全部合法值，但是仍然有一些特殊的关键字值用以表示这些数字概念：非数字，最小允许值，最大允许值，正无穷，以及负无穷。

每个特殊值都可以赋给变量和属性，或者和其他数字常量一样，用在直接量表达式中。但是，特殊数值经常被解释程序作为一些表达式计算的结果而返回。

### 非数字：NaN

在某些情况下，数字计算或者数据类型转换的结果值并不是一个数字。例如，0/0就是一个不存在的计算，下面的表达式也不能够转换为有限的数字：

```
23 - "go ahead and try!"
```

为了表示那些虽属于数据类型但不是实数的数据, ActionScript 提供了 NaN 关键字值。虽然 NaN 不表示一个数, 但它仍然是一个数据类型的合法值, 比如下面的代码:

```
x=0/0;  
trace(x);           // 显示: NaN  
trace(typeof x);  // 显示: 'number'
```

由于 NaN 不是一个有限的数值, 因此它不能比较是否相等。如果两个变量的值都为 NaN, 它们通常被认为是不相等的(虽然它们看起来是一样的)。针对这个问题, 我们用内置函数 `isNaN()` 来检查一个变量所包含的值是否为 NaN:

```
x=12-'this doesn't make much sense'; // x现在的值为NaN  
trace(isNaN(x));                      // 显示: true
```

## 最小和最大允许值: MIN\_VALUE 和 MAX\_VALUE

ActionScript 对数字的表示范围虽然很宽, 但也不是无限的。最大允许值为 1.7976931348623157e+308, 而最小允许值为 5e-324。显然, 这些数字看起来并不是很方便, 因此, 我们用特殊值 `Number.MAX_VALUE` 和 `Number.MIN_VALUE` 来代替。

`Number.MAX_VALUE` 在我们检查一个计算结果是否为可表示的数字时非常适用:

```
z=x*y;  
if(z<=Number.MAX_VALUE && z>=-Number.MAX_VALUE) {  
    // 数字是合法的  
}
```

注意, `Number.MIN_VALUE` 是所允许的最小正值, 而不是最大负数值。合法的最大负数值为 `-Number.MAX_VALUE`。

## 正无穷和负无穷: Infinity 和 -Infinity

如果一个计算结果的值大于 `Number.MAX_VALUE`, ActionScript 就会用关键字 `Infinity` 来表示这个计算的结果。与此类似, 如果一个计算结果的值超过所允许的最大负值, ActionScript 就用 `-Infinity` 来表示它。`Infinity` 和 `-Infinity` 也可以直接作为直接量数值表达式。

## 无理数

除了特殊数值 NaN, Infinity, -Infinity, Number, MAX\_VALUE 和 Number.MIN\_VALUE 之外, ActionScript 还提供了 *Math* 对象, 以方便对数字常数的访问。例如:

```
Math.E          // e 的值是自然对数的底数
Math.LN10;      // 10 的自然对数
Math.LN2;       // 2 的自然对数
Math.LOG10E    // 以 10 为底, e 的对数
Math.LOG2E    // 以 2 为底 e 的对数
Math.PI         // Pi (也就是 3.1415926……)
Math.SQRT1_2   // 1/2 的平方根
Math.SQRT2     // 2 的平方根 (也就是 1.4142135……)
```

这些常数只是常用无理数的浮点值速记形式。你可以像使用其他任何对象属性那样使用这些无理数:

```
area=Math.PI * (radius * radius);
```

参见第三部分, 可以得到所支持常数的完整列表。

## 数字处理

你可以用操作符来合并数字, 形成数学表达式, 或者通过调用内置函数来执行复杂的数学运算。

## 操作符的使用

基础算术——加法、减法、乘法和除法——是用 +, -, \* 和 / 操作符来执行的。操作符可以被用在任何数值直接量或者诸如变量的数据容器上。数学表达式按照操作符的优先等级所规定的顺序来执行, 如表 5-1 所示。例如, 乘法优先于加法。下面给出的数学操作符的用法都是合法的:

```
x=3 * 5;           // 将 x 赋为 15
x=1+2-3/4;         // 将 x 赋为 2.25

x=56;
y=4 * 6+x;         // 将 y 赋为 80
y=x+(x * x)/x;    // 将 y 赋为 112
```

## 内置数学函数

要执行高级的数学运算，可以使用 *Math* 对象的内置数学函数。例如：

```
Math.abs(x)          // x 的绝对值  
Math.min(x,y)       // y 和 x 中值最小者  
Math.pow(x,y)       // x 升到 y 次幂  
Math.round(x)        // 将 x 舍入为最接近的整数
```

我们可将数学函数返回的值用在表达式中，就像使用实数那样。例如，假设要模仿一个六面的骰子，我们可以用 *random()* 函数得到一个 0 到 1 之间的随机浮点数：

```
dieRoll=Math.random();
```

然后，将这个值乘以 6，就得到一个 0 到 5.999 之间的浮点数，再加上 1：

```
dieRoll = dieRoll * 6+1; // 将 dieRoll 设置为 1 到 6.999 之间的一个数字
```

最后，用 *floor()* 函数来将得到的数字舍入为最接近的整数：

```
dieRoll=Math.floor(dieRoll); // 将 dieRoll 设置为 1 到 6 之间的一个数字
```

将这些东西放到一个单一的表达式中，我们的滚动骰子计算如下所示：

```
// 将 dieRoll 设置为 1 到 6 之间的一个整数  
dieRoll=Math.floor(Math.random()*6+1);
```

## 串类型

串 (string) 是用在文字数据（字母、标点符号和其他字符）中的数据类型。一个串直接量是包含在引号内的字符的任意组合：

```
"asdfksldfsdfeoif"      // 一个无意义的串  
"greetings"             // 一个友好的串  
"moock@moock.org"       // 一个自强的串  
"123"                   // 看似数字，其实是一个串  
'singles'               // 单引号也是可以使用的
```

在学习如何形成串直接量之前，先来查看串里允许使用的字符。

## 字符编码

和所有计算机数据一样，文字字符用一个数字代码存储在内部。它们使用一个字符集（character set）在存储时进行编码，在显示时进行解码，字符集将字符对应于它们的数字代码。字符集随着语言和字母表的不同而有所不同。西方过去的应用程序使用ASCII的一些派生集，ASCII是一个标准字符集，只包含128个字符——英语字母、数字和基本的标点符号。现代的应用程序支持一个字符集系列，总称ISO-8859。ISO-8859中的每一个字符集都包含标准的拉丁字母表（‘A’到‘Z’），以及目标语言所需要的可变字符集。ActionScript 使用 ISO-8859-1，也就是 Latin1，作为它的主要字符映射。

Latin1 字符集可以适应大部分的西欧语言——法语、德语、意大利语、西班牙语、葡萄牙语，等等——但是不支持希腊语、土耳其语、斯拉夫语和俄语。Unicode 可以映射到一百万个字符，是首选的国际编码标准，但是在 ActionScript 中不被支持（要支持 Unicode 会使 Flash 播放器变得很大）。但是，ActionScript 支持为日语字符设置的第二字符集，也就是 Shift-JIS。在 ActionScript 处理文字的时候，我们可以使用 Latin1 或者 Shift-JIS 中的任何字符。

即使 Unicode 本身不被支持，也可以使用标准 Unicode 的转义序列来表示 Latin1 或者 Shift-JIS 中的任何字符。还可以用 Unicode 类型的函数来处理字符串。从理论上讲，Unicode 在未来的某一天，在不破坏原来代码的基础上是可以添加到 Flash 中来的。

附录二中列出了每个字符的 Unicode 的编码点（code point），也就是字符在 Unicode 集中的数字位置。稍后我们将会看到如何使用这些编码点来操作脚本中的字符。

## 串直接量

要建造一个串的最普通方法，就是在从 Latin1 或 Shift-JIS 字符集中选取出来的一组字符前后加上双引号或者单引号：

```
"Hello"  
Nice night for a walk.'  
'The equation is 12+4=16, which programmers see as 12+4==16.'
```

如果使用双引号来作为串的开始，就必须仍然以双引号作为结束。同样，如果在开

头使用单引号，那么结束的时候也只能使用单引号。但是，双引号串可以包含单引号字符，反之亦然。例如，下面给出的串中单引号和双引号的使用是合法的：

```
'Nice night, isn't it?'           // 双引号中的单引号(撇号)  
'I said, "What a pleasant evening!'" // 单引号中的双引号
```

## 空串

可能的最短串就是空串 (empty string)，这种串并不包含任何字符：

```
" "  
''
```

空串在要测试一个变量是否包含有效串值的时候比较适用：

```
if (firstName == "") {  
    trace("You forgot to enter your name!");  
}
```

但是，将一个变量同空串比较并不总是能产生我们需要的效果。记住，空串和数字 0，以及布尔值 false 被认为是相等的（参见表 3-1 和表 3-3）。因此，为了确保检查的变量的确是一个空串，首先应该确定该变量属于串数据类型，如下所示：

```
if(typeof firstName == 'string' && firstName == " ") {  
    trace("You forgot to enter your name!");  
}
```

## 转义序列

我们在前边已经看到了，单引号(')可以用在双引号包括的直接量中，同时，双引号("")也可以用在单引号包括的直接量里。但是，如果我们两个都用会怎么样呢？例如：

```
'I remarked "Nice night, isn't it?"'
```

这样做的话，这行代码会产生错误，因为解释程序认为串直接量是以“isn't”中的撇号作为结束的。解释程序将它理解为：

```
'I remarked "Nice night, isn'      // 后面的部分被当作了无用信息
```

要在用单引号包括起来的串直接量中再使用单引号，我们必须用转义序列 (escape sequence)。

转义序列表示一个直接量串值的时候，采用反斜杠 (\)，后面跟着描述所需字符的代码或者该字符本身的形式。单引号和双引号的转义序列分别是：

\'  
\''

因此，要把上面的句子表示为正确的串直接量，就应该这样写：

```
'I remarked 'Nice night, isn\'t it?'" // 对单引号进行换码
```

其他的转义序列可以用来表示各种特殊的或者保留字符，如表 4-1 所示。

表 4-1 ActionScript 转义序列

转义序列	含义
\b	退格字符 (ASCII 8)
\f	换页符 (ASCII 12)
\n	换行符，执行换行操作 (ASCII 10)
\r	回车 (CR) 字符，完成换行操作 (ASCII 13)
\t	跳格符 (ASCII 9)
\'	单引号标记
\"	双引号标记
\\\	反斜杠字符，这种形式在将反斜杠用作直接量字符串的时候，可以防止\被解释程序当作转义序列的开始。

## Unicode 类型的转义序列

并非 Latin 1 和 Shift-JIS 中的所有字符都可以用键盘来产生。为了在一个串中包裹所有难以获得的字符，可以使用 Unicode 类型的转义序列。注意，Flash 实际上不支持 Unicode，它只是仿效其语法。

一个 Unicode 类型的转义序列以反斜杠和小写的 u（也就是 \u）开始，后面跟着一个四位的十六进制数，这个数和 Unicode 字符的编码点相对应，如下所示：

```
\u0040 // @标记  
\u00A9 // 版权符号  
\u0041 // 大写字母 "A"
```

编码点是属于 Unicode 字符集中每一个字符的特殊的标识数字。参见附录二可以得到一个针对 Latin1 的 Unicode 编码点列表。Shift-JIS 编码点可以在 Unicode 企业站点中找到：

<ftp://ftp.unicode.org/Public/MAPPINGS/EASTASIA/JIS/SHIFTJIS.TXT>

如果我们只从 Latin1 字符集中进行字符换码，就可以使用标准 Unicode 转义序列的一个简易格式。简易格式的组成是：以前缀 \x 开始，后面跟着一个两位的十六进制数字，表示 Latin1 对字符的换码。由于 Latin1 编码点和 Unicode 编码点的前 256 个相同，因此，你仍然可以使用附录二中的引用图表，但是要删除 u00，如下所示：

```
\u0040      // Unicode 转义序列  
\x40      // \x 便捷形式  
\u00A9      // Unicode  
\xA9      // ... 这下你懂了吧
```

除了使用 Unicode 转义序列，我们还可以通过更烦琐的内置函数 *fromCharCode()* 来把任何字符插入一个串中，这个函数在后面会有相应的描述。注意，对于 Unicode 转义序列和 *fromCharCode()* 函数，Flash 5 只支持那些映射到 Latin1 和 Shift-JIS 字符集中字符的编码点。插入其他编码点不会得到正确的 Unicode 字符，除非 Flash 以后的版本能支持 Unicode 字符系统。

## 串的处理

通过对串的处理，我们可以编写从用户输入的任何验证信息到拼字游戏的各种程序。如再加上一些创意，还可以产生美妙的视频文字效果和其他有趣的东西。

我们可以用两个操作符和一些内置函数来对串进行操作。串操作符可以用来将多个串连接在一起，或者比较两个串的字符。内置函数可以检查一个串的属性和内容，提取串的某个部分，检查字符的编码点，从一个编码点创建字符，修改串中字符的情况，甚至可将一个串转化为变量或属性名称。

## 串的连接

串的连接（从两个或者更多的串创建一个新串）称为串联。在前面你已经看到，我们可以用加操作符（+）将两个串连接起来，如下所示：

```
"Macromedia' + 'Flash'
```

这行代码将产生一个单独的串值“MacromediaFlash”。哎呀！我忘记在词之间放一个空格了。要加上一个空格，我们可以将它插入到其中一个串的引号中间，如下所示：

```
"Macromedia" + "Flash"           // 产生 "Macromedia Flash"
```

但这并不总是适用。通常，我们不想在一个公司或者产品名称中间加一个空格。因此，我们通常在中间串联一个独立的空格：

```
"Macromedia" + " " + "Flash"      // 也产生 "Macromedia Flash"
```

注意，空格字符和我们前边所看见的空串不同，因为空串在引号中间是没有字符的。

我们也可以将包含串数据的变量连接起来。看看下面的代码：

```
var company = "Macromedia";
var product = "Flash";

// 将变量sectionTitle 设置为 "Macromedia Flash"
var sectionTitle = company + " " + product;
```

在第一行和第二行代码中，我们将串值存储在变量里。然后，我们把这些值连在一起，中间有一个空格。两个串值保存在了变量中，还有一个（空格）是串直接量。没问题。始终都是这样。

有时候，我们会希望在一个已经存在的串中添加字符。例如，我们可以改变一句欢迎词的语气，如下所示：

```
var greeting = "Hello";           // 我们的欢迎消息
greeting = greeting + "?";       // 我们的询问句: 'Hello?'
```

前面的代码虽然可以达到我们的目的，但是要注意，我们在第二行代码中要两次使用greeting。如果想更有效一些，可以使用+=操作符，它可以将右边的串附加到左边的串变量上：

```
var greeting = "Hello"; // 欢迎消息
greeting += "?";      // 问句式欢迎消息
```

---

**警告：**Flash 4 的串连接操作符（&）在 Flash 5 中将执行不同的操作（逻辑运算 AND）。如果导出一个 Flash 4 .swf 文件，你必须使用 add 操作符来连接串。注意，add 虽然在向后兼容中得到支持，但 Flash 5 更趋向于使用 + 操作符。

---

## concat() 函数

*concat()* 函数用来为串添加字符，就如同 *+ = -* 一样。因为 *concat()* 是一个函数，因此它使用点操作符，如下所示：

```
var product='Macromedia'.concat("Flash");
var sentence='How are you';
var question=sentence.concat('?')
```

其中要注意 — 和 *+ = -* 不同，*concat()* 函数不改变它所处理的串，只是返回连接之后的串值。学习下面的代码，你就可以理解 *+ =* 和 *concat()* 之间的不同：

```
var greeting="Hello";
greeting.concat('?');
trace(greeting);           // 显示 'Hello'；greeting 在连接过程中不受影响

finalGreeting=greeting.concat('?');
trace(finalGreeting);      // 显示 'Hello?'
```

*concat()* 函数也可以接收多个参数（也就是说，它能够将多个独立的串合并为一个）：

```
firstName="Karsten";

// 将 finalGreeting 设置为 'Hello, Karsten?'
finalGreeting=greeting.concat(' ',firstName,"?");
```

这和下面的语句效果其实是一样的：

```
finalGreeting=greeting;
finalGreeting+=" "+firstName+"?";
```

## 串的比较

要检查两个串是否相等，我们使用相等（*==*）和不等（*!=*）操作符。在实现有条件的代码的时候，我们经常需要比较串。例如，如果用户输入了一个密码，我们就需要将他的输入串同正确的密码相比较。比较的结果控制着代码的执行。

## 使用相等（==）和不等（!=）操作符

相等操作符要带两个操作数——一个在左边，一个在右边。操作数可以是串常量，或者是任何变量、数组元素、对象属性，以及可以转换为串的表达式：

```
"hello" == "goodbye"          // 比较两个串直接量  
userGuess == "fat-cheeks"    // 将串同变量进行比较  
userGuess == password        // 比较两个变量
```

如果右边的操作数和左边的操作数有着完全相同的字符以及排列顺序，那么两个串就被认为是相等的，结果布尔值就是 true。但是，大写和小写字母在字符集中是由不同的编码点来表示的，因此，它们被认为是不等的。下面的比较都得到 false：

```
"olive-orange" == "olive orange"      // 不等  
"nighttime" == "night time"         // 不等  
"Day 1" == "day 1"                  // 不等
```

因为串的比较结果就是其布尔值为 true 或者 false，我们就可以在条件语句和循环中将它们用作测试表达式，如下所示：

```
if(userGuess == password) {  
    gotoAndStop("classifiedContent");  
}
```

如果表达式 (*userGuess == password*) 是 true，那么 *gotoAndStop("classifiedContent")*; 语句就可以得到执行。如果表达式为 false，*gotoAndStop("classifiedContent")*; 语句就被跳过。

我们在以后的章节中将会学习有关布尔值的更多内容。我们还要在第七章中学习条件语句。

要检查两个串是否相等，我们使用不等操作符，它产生和相等操作符相反的结果。例如，下面的表达式表示的值分别为 true 和 false：

```
"Jane" != "Jane"      // 因为两个串是相等的，因此得到 false  
"Jane" != "Biz"       // 因为两个串不同，所以得到 true
```

现在，我们用不等操作符来保证某些动作仅当两个串不等的时候才能执行：

```
if(userGender != "boy") {  
    // 针对女孩的代码……  
}
```

---

注意：如果导出一个 Flash 4 的 .swf 文件，你必须使用更老的 eq 和 ne 操作符来比较串的相等或者不等。虽然 eq 和 ne 可以在向后兼容中得到支持，但是 Flash 5 会优先考虑 == 和 != 操作符。

---

## 字符顺序和字母次序的比较

我们也可以在字符顺序的基础上来比较两个串。我们在前边已经看到，每一个字符都有一个数字的编码点，而这些编码点在字符集上都是有数字顺序的。我们可以用比较（comparison）操作符来检查哪个字符的顺序在前边：

```
'a' < "b"  
'2' > "&"  
'r' <= "R"  
"$" >= "@"
```

和相等与不等操作符很相似，比较操作符产生一个布尔值，true 或者 false，这取决于操作数之间的关系。每个操作数可以是能产生串值的任何东西。

由于字符 ‘A’ 到 ‘Z’，‘a’ 到 ‘z’ 都按顺序列在 Latin1 字符集的字母表中，我们常常可以使用字符顺序的比较方式来确定两个字母的先后顺序。但是要注意，在 Latin1 字符集中，任何大写字母都在小写字母之前。如果你忘记了这个，就会得到令人惊讶的结果：

```
'Z' < "a"      // 得到 true  
'z' < "a"      // 得到 false  
"Cow" < "bird" // 得到 true
```

下面我们来仔细研究一下每一种比较操作符。在下面的描述中，比较字符（comparison character）是在两个操作数中发现的第一个不等字符：

### 大于 (>)

如果左边操作数的比较字符在 Latin1 或者 Shift-JIS 字符集中的顺序比右边操作数的比较字符更靠后，那么就产生 true。如果两个操作数完全相等，> 就返回 false：

```
"b" > 'a'      // true  
'a' > 'b'      // false  
"ab" > "ac"    // false (第二个字符是比较字符)  
"abc" > "abc"  // false (串是相等的)
```

```
'ab'>"a"      // true (b是比较字符)
'A">"a"        // false ('A'的字符顺序比'a'靠前)
```

### 大于等于

如果左边操作数的比较字符比右边操作数的比较字符顺序靠后，或者两个操作数完全相等，就产生 true：

```
'b'>="a"      // true
'b'>="b"      // true
'b'>="c"      // false
'A'>="a"      // false ('A'和'a'有不同的编码点)
```

### 小于 (<)

如果左边操作数的比较字符在Latin1或者Shift-JIS字符集中的顺序比右边操作数的比较字符靠前，那么就得到 true。如果两个操作数完全相等，< 就返回 false：

```
"a" < "b"      // true
'b' < "a"      // false
"az" < "aa"    // false (第二个字符是比较字符)
```

### 小于等于 (<=)

如果左边操作数的比较字符比右边操作数的比较字符顺序靠前，或者两个操作数完全相等，就得到 true：

```
"a" <= "b"     // true
'a' <= "a"      // true
'z' <= "a"      // false
```

要确定两个没有字母顺序的字符哪一个在Latin1字符顺序里是靠前的，可以参见附录二。

下面的例子用来检查一个字符是否来自Latin字母表（不能是数字、标点符号或者其他符号）。

```
var theChar = 'w';

if ((theChar>="A" && theChar<="Z") || ((theChar>="a" && theChar<="z")) {
    trace('该字符在Latin字母表中。');
}
```

注意逻辑OR操作符（||）是如何一次检查两个条件的。我们将在第五章中学习OR操作符。

---

注意：如果导出 Flash 4 的 .swf 文件，你必须使用原来的 gt, ge, lt 和 le 串比较操作符。虽然以前的操作符在向后兼容中得到支持，但是 >、>=、< 和 <= 操作符在 Flash 5 中更正规。

---

## 使用内置串函数

除了 *concat()* 函数之外，我们到现在为止所使用的串工具都是操作符。现在，我们要来看看如何使用内置函数和属性来执行更高级的串操作。

要对串使用内置函数，必须执行一个函数调用，它的形式是：

```
string.functionName(arguments)
```

例如，下面我们要对 myString 执行 *charAt()* 函数：

```
myString.charAt(2)
```

串函数返回的数据和原始串有某种关系。我们在以后的使用中将这些返回值赋给变量或者对象属性，如下所示：

```
thirdCharacter=myString.charAt(2);
```

## 字符索引

许多串函数都使用字符索引（index）——和串的首字符有关的数字位置，从 0 开始，而不是 1。首字符是数字 0，然后是数字 1，第三个是数字 2，依此类推。例如，在串 “red” 中，*r* 是索引 0，*e* 是索引 1，而 *d* 是索引 2。

使用字符索引，我们就能识别串的部分。例如，我们可以命令解释程序“将索引 3 的字符放到索引 7 的位置上”，或者我们可以问“索引 5 的字符是什么？”

## 检查串

可以用内置 *length* 属性或者 *charAt()*, *indexOf()* 以及 *lastIndexOf()* 函数来在串内进行检查或者搜索。

## length 属性

length 属性可以告诉我们一个串中拥有多少个字符。因为它是一个属性，而不是函数，在使用的时候不能加括号或者参数。下面，我们来看看几个串的 length：

```
'Flash'.length          // length 是 5  
"skip intro".length    // length 是 10 (空格也是一个字符，每个字符都算)  
''.length                // 空串所包含的字符为 0  
  
var axiom='all that glistens will one day be obsolete';  
axiom.length             // 42
```

因为字符索引从 0 开始（也就是与 0 相对），最后一个字符的索引就总是等于串的 length 减去 1。

一个串的 length 属性可以读出来，但是不能设置。不能以下面这种方式来让串变得更长：

```
axiom.length=100;        // 虽然是很好的尝试，但是不会有效
```

---

**注意：**如果导出一个 Flash 4 的 .swf 文件，你必须使用原来的 length() 函数，如下所示。length() 函数虽然在向后兼容中得到支持，但是 Flash 5 会优先使用 length 属性。

---

下面，用 Flash 4 的 length() 函数来显示词 obsolete 中的字符数：

```
trace(length("obsolete")); // 显示: 8
```

## charAt() 函数

我们可以用 charAt() 函数来确定串中任何索引位置上的字符是什么，它的形式为：

```
string.charAt(index)
```

string 可以是一个直接量串值或者是包含一个串的标识符，而 index 是一个整数或者是能得到一个整数的表达式，这个整数描述我们要得到的字符的位置。index 的值可以是 0 到 string.length-1 之间的任何整数。如果 index 不在这个范围里，就返回空串。下面给出了一些例子：

```
"It is 10:34 pm".charAt(1)      // 返回 "t"，第二个字符  
var country="Canada";
```

```

country.charAt(2);           // 返回 "n", 第三个字符
var x=4;
fifthLetter=country.charAt(x);          // fifthLetter 是 "d"
lastLetter=country.charAt(country.length-1); // lastLetter 是 "a"

```

## indexOf()函数

我们用 *indexOf()* 函数来搜索串中的字符。如果要搜索的串包含我们的搜索序列, *indexOf()* 就返回序列在串中的第一次出现的索引 (也就是位置)。否则, 就返回值 -1。*indexOf()* 的一般形式为:

```
string.indexOf(character_sequence, start_index)
```

*string* 是任何的串值或者包含一个串的标识符, *character\_sequence* 是我们要搜索的串, 它可能是一个串直接量或者包含一个串的标识符, 而 *start\_index* 是搜索的开始位置。如果 *start\_index* 被遗漏, 那么搜索就从 *string* 的起头开始。

现在我们用 *indexOf()* 来检查一个串中是否包含字符 *W*:

```
"GWEN!".indexOf("W"); // 返回 1
```

对, *W* 是 “GWEN!” 中的第 2 个字符, 因此我们就得到了 1, 也就是 *W* 字符的索引。记住, 字符索引从 0 开始, 因此第二个字符的索引为 1。

如果我们搜索小写字符 *w* 又会如何呢? 我们来看看:

```
"GWEN!".indexOf("w"); // 返回 -1
```

在 “GWEN!” 中并没有 *w*, 因此 *indexOf()* 返回 -1。大写和小写的字符是不同的, *indexOf()* 对大小写是很敏感的!

现在我们来确定在一个邮箱地址中是否有一个 @ 符号:

```

var email="daniella2dancethenightaway.ca"; // 噢, 忘记按住 shift 键了!
// 如果没有@符号, 就通过 formStatus 文本域警告用户
if (email.indexOf("@") == -1){
    formStatus="The email address is not valid.";
}

```

我们并非总是只能搜索单一的字符。我们还可以在串中搜索整个的字符序列。现在, 我们在一个公司地址中寻找 “Canada”:

```
var iceAddress='St. Clair Avenue, Toronto, Ontario, Canada';
iceAddress.indexOf("Canada"); // 返回 36, 也就是字母 "C" 的索引
```

注意, *indexOf()*返回“Canada”第一个字符的位置。现在, 我们将*iceAddress.indexOf("Canada")*的返回值同 -1 比较一下, 并将结果赋给一个存储该公司所在国家的变量:

```
var isCanadian=iceAddress.indexOf("Canada") != -1;
```

如果 *iceAddress.indexOf("Canada")* 不等于 -1 (发现了 "Canada"), 则 *iceAddress.indexOf("Canada") != -1* 的值就会是 *true*; 如果 *iceAddress.indexOf("Canada")* 等于 -1 (没找到 "Canada"), 则为 *false*。然后, 我们将这个布尔值赋给变量 *isCanadian*, 就可以利用它来为北美创建一个指定国家的邮寄表单:

```
if(isCanadian) {
    mailDesc = "Please enter your postal code.";
} else {
    mailDesc="Please enter your zip code.";
}
```

*indexOf()*函数还可以帮助我们确定串中的哪一个部分是我们要抽取的。学习 *substring()* 函数的时候我们会看到这是如何工作的。

## lastIndexOf()函数

*indexOf()*函数返回一个字符序列在串中第一次出现的位置。*lastIndexOf()*函数返回的则是一个字符序列在串中最后一次出现的位置, 或者如果序列没有被发现就返回 -1。*lastIndexOf()*的一般形式和 *indexOf()*相似:

```
string.lastIndexOf(character_sequence, start_index)
```

惟一的不同之处是, *lastIndexOf()*是向后搜索一个串, *start\_index*表示要包括在搜索结果中的最右边一个字符 (不是最左边)。如果 *start\_index*被遗漏, 就默认是 *string.length-1* (串的最后一个字符) 的地方。

例如:

```
paradox="Pain is pleasure,pleasure is pain";
paradox.lastIndexOf("pain"); // 返回 30, indexOf()将会返回 0
```

下面的语句将会返回 0（词 pain 第一次出现的索引），因为我们在第二个 pain 出现之前开始了向后搜索：

```
paradox.lastIndexof("pain", 29); // 返回 0
```

## 没有正则表达式

注意，在 ActionScript 中不支持正则表达式（用来在文字数据中识别模式的强大工具）。

## 获取部分串

有时，一个长的串中包含了我们希望分别访问的一系列字符。例如，“Steven Sid Mumby” 中，我们可能只想获取最后一个名字“Mumby”。要得到一个稍短的串（或者说子串），我们可以分别使用这些函数：*substring()*, *substr()*, *splice()* 或者 *split()*。

### substring() 函数

我们使用 *substring()* 函数来从串中获取一个字符序列，这要基于开始和结束字符的索引。*substring()* 函数的形式如下：

```
string.substring(start_index, end_index)
```

*string* 是任何的串变量或者是包含了一个串的标识符，*start\_index* 是包含在子串中的首字符索引，而 *end\_index* 则是我们要包含在子串中的最后一个字符之后的字符索引。如果不提供这个参数，*end\_index* 就把 *string.length* 作为默认值。因此：

```
var fullName="Steven Sid Mumby";
middleName=fullName.substring(7,10); // 将 middleName 赋值为 "Sid"
firstName=fullName.substring(0,6); // 将 firstName 赋值为 "Steven"
lastName=fullName.substring(11); // 将 lastName 赋值为 "Mumby"
```

实际上，我们不知道第一个名字、中间的名字以及最后的名字是从哪里开始，在何处结束，因此，我们要寻找一些代表性的分隔符，比如说空格就可以帮助我们猜出何处是词的间隔。现在，我们要搜索名字中的最后一个空格，然后就可以假设其后的串就是我们要的最后一个名字：

```
fullName="Steven Sid Mumby";
lastSpace=fullName.lastIndexOf(" ");           // 返回 10

// 从 11 开始到最后的串可以推测为最后一个名字
lastName=fullName.substring(lastSpace+1);
trace('Hello Mr. '+lastName);
```

如果 `start_index` 比 `end_index` 大，那么在函数执行之前这两个参数就自动交换。虽然下面的函数调用会得到相同的结果，但是你不应该在使用 `substring()` 函数的时候养成参数颠倒的习惯，因为它会让你的代码不容易被看懂：

```
fullName.substring(4, 6);           // 返回 "en"
fullName.substring(6, 4);           // 返回 "en"
```

### substr() 函数

`substr()` 函数用开始索引和长度（和 `substring()` 相反，它使用的是开始和结束索引）从一个串中截取字符序列。`substr()` 的一般格式为：

```
string.substr(start_index, length)
```

`string` 和前边一样，是任何直接量串值或者包含了一个串的标识符，`start_index` 是要包含在子串中的第一个字符，`length` 指定子串中要包含的字符数目，从 `start_index` 开始，向右计数。如果 `length` 被遗漏，子串就从 `start_index` 开始，到原串的最后一个字符结束。例如：

```
var fullName="Steven Sid Mumby";
middleName=fullName.substr(7,3);           // 将 middleName 赋值为 "Sid"
firstName=fullName.substr(0,6);           // 将 firstName 赋值为 "Steven"
lastName=fullName.substr(11);            // 将 lastName 赋值为 "Mumby"
```

如果使用负数，`start_index` 即被设置为与串的结尾相应。最后的字符是 `-1`，倒数第二的是 `-2`，依此类推。因此，前面的三个 `substr()` 例子可以写成这样：

```
middleName=fullName.substr(-9,3);           // 将 middleName 赋值为 "Sid"
firstName=fullName.substr(-16,6);           // 将 firstName 赋值为 "Steven"
lastName=fullName.substr(-5);              // 将 lastName 赋值为 "Mumby"
```

但是，`length` 不允许为负数。

---

**注意：** 在 Flash 5 中，`substr()` 函数是一个串抽取函数，它和 Flash 4 中的 `substring()` 函数非常接近，都是用一个开始索引和一个长度来获得一个子串。

---

## slice()函数

和 *substring()*相似，*slice()*也是基于开始和结束字符的索引，从一个串中截取一个字符系列。*substring()*可以只指定和原串的开始字符相关的索引，而*Slice()*分别指定和原串的开始与结束字符相关的索引。

*slice()*函数有下面的形式：

```
string.slice(start_index, end_index)
```

*string*是任意的直接量串值，或者是包含一个串的标识符，*start\_index*是所要截取的子串中的第一个字符。如果 *start\_index*是一个正数，它就是一个普通的字符索引；如果 *start\_index*是一个负数，则表示要从串的结尾开始倒数相应的字符索引 (*string.length+start\_index*)。最后，*end\_index*是包含在子串中的最后一个字符后面的字符索引。如果 *end\_index*被遗漏，它的默认值就是 *string.length*。如果 *end\_index*是负数，它所表示的字符索引就是从串的结尾倒数过去的（也就是 *string.length+end\_index*）。

在 *slice()*中使用非负的索引和 *substring()*相同。如果使用了负索引，要记住，你得到的并不是字符颠倒的子串，字符顺序也不是从 *end\_index*到 *start\_index*的。你只不过是在确定索引的时候将它和串结尾处的字符相对应起来。还要记住，串中的最后一个字符是 -1，*end\_index*参数指定的是子串最后一个字符后面的字符，因此，用负的 *end\_index*来将原串中最后一个字符指定为子串的最后一个字符是不可能的。认真看看我们是如何利用负索引来抽取下面的子串的：

```
var fullName="Steven Sid Mumby";
middleName=fullName.slice(-9,-6); // 将middleName赋值为"Sid"
firstName=fullName.slice(-16,-10); // 将firstName赋值为"Steven"
lastName=fullName.slice(-5,-1); // 将lastName赋值为"Mumb";
                                // 不是我们要得到的,
                                // 但是在要使用负数的情况下我们已经尽力了
lastName=fullName.slice(-5,16) // 将'Mumby'赋值给lastName。
                                // 注意：我们将正负索引放在一起使用
```

## split()函数

到现在为止，我们所见的串抽取函数一次只能抽取一个字符序列。如果我们想一下子就得到很多子串，可以使用功能强大的*split()*函数。（因为*split()*函数是针对数组的，你可能会想先跳过这个函数，在看了第十一章之后再回过头来看。）

*split()*函数将一个串分散为多个子串，然后将这些子串存储到一个数组中作为返回。*split()*函数的格式如下：

```
string.split(delimiter)
```

*string*是任何直接量串值或者包含一个串的标识符，*delimiter*是用来分隔*string*的一个或者多个字符。典型的分隔符是逗号，空格和退格符。例如，我们要按照每一个逗号来分隔一个串，可以这样使用：

```
theString.split(',')
```

使用*split()*有一个小技巧，就是可以将一个句子分散为单个的词。我们讨论*substring()*，*substr()*和*slice()*函数的时候，要人工从串“Steven Sid Mumby”中抓取一个一个的名字。现在，如果我们使用*split()*，以空格（“ ”）作为*delimiter*，事情会变得非常简单：

```
var fullName="Steven Sid Mumby";
var names=fullName.split(' ');
// 多简单啊！

// 现在我们将数组中单个的名字赋值给单独的变量
firstName=names[0];
middleName=names[1];
lastName=names[2];
```

## 串抽取的性能问题

Flash 5 中的*substr()*和*slice()*函数实际上只是 Flash 4 中的*substring()*函数的变体，因此要花稍微多一点的时间来执行。速度的差别可以以毫秒来计，但是，在高强度的串处理中该问题就可以被注意到。对于重复度非常高的操作，*substring()*是最佳选择，如下所示：

```
fullName='Steven Sid Mumby';
// 用Flash 5的substr()函数将"Sid"赋值给middleName
middleName=fullName.substr(7,3);
// 用Flash 4的substring()函数将"sid"赋值给middleName。
// 注意，Flash 4的substring()字符索引是从1开始的。
middleName=substring(fullname,8,3);
```

## 串检查和子串抽取的结合

我们已经学习了如何搜索串中的字符，以及如何从串中抽取字符。这两个任务如果被放到一起，就会产生强大的功能。

迄今为止我们所看到的大部分例子都将直接量表达式作为参数，如下所示：

```
var msg='Welcome to my website!';
var firstWord=msg.substring(0,7);           // 0和7是数值直接量
```

这是 *substring()* 应用的一个恰当范例，但是，它不代表真实世界里 *substring()* 的典型用法。我们通常不能预先知道串的内容，而只能动态地产生我们的参数。例如，我们不会按静态的方式说“给我从索引0到索引7的子串”，我们通常以动态的方式说“给我这个串中从开始字符到第一个空格出现之前的那个字符所组成的子串”。这种较为灵活的方法并不要求我们预先知道串中的内容。下面，我们要抽取变量 *msg* 的第一个单词，把 *substring()* 和 *indexOf()* 合并起来使用：

```
var firstWord=msg.substring(0,msg.indexOf(" "));
```

表达式 *msg.indexOf("")* 计算出 *msg* 中第一个空格的数字索引。这种技术并不管空格的位置究竟在哪里。这让我们可以处理随着程序运行而改变的串，并且为我们节省了很多字符计算工作，那些计算是很容易导致错误的。

串检查和串抽取的结合实际上应用非常广泛。在例 4-2 中，我们抽取 *msg* 变量中的第二个单词，而不用费力寻找字符的索引。在自然语言里，我们要“从 *msg* 里抽取一个从首个空格之后的字符开始，到第二个空格出现之前的字符结束的子串”。我们将第一个和第二个空格的位置存储到变量中，以便接下来的工作更为明了。

### 例 4-2：获取串中的第二个单词

```
var msg="Welcome to my website!";
firstSpace=msg.indexOf(" ");                  // 找到第一个空格
secondSpace=msg.indexOf(" ",firstSpace+1); // 找到第二个空格

// 现在开始抽取第二个单词
var secondWord=msg.substring(firstSpace+1,secondSpace);
```

## 字符的大小写转换

我们可以用内置 *toUpperCase()* 和 *toLowerCase()* 函数将一个串转换为大写或者小写

的形式。当希望以更好的形式显示串，或者比较不同大小写形式的串时，经常使用这种方法。

### toUpperCase()函数

*toUpperCase()*函数将串中所有的字符都转换为大写形式（也就是大写字母），并返回转换之后的版本。如果某个给定的字符没有大写形式，那么返回的版本就没有变化。*toUpperCase()*的一般格式为：

```
string.toUpperCase()
```

*string*是任意的直接量串值或者包含一个串的标识符。下面是一些例子：

```
"listen to me".toUpperCase(); // 产生串 "LISTEN TO ME"  
var msg1="Your Final Score:234";  
var msg2=msg1.toUpperCase(); // 将msg2设置为 "YOUR FINAL SCORE:234"
```

注意，*toUpperCase()*不影响调用它的串，它只是返回原串的大写形式副本。下面的例子有不同之处：

```
var msg="Forgive me, I forgot to bring my spectacles.";  
msg.toUpperCase();  
trace(msg); // 显示 "Forgive me, I forgot to bring my spectacles."  
// msg 并不会因为 toUpperCase() 的转换作用而受到影响
```

### toLowerCase()函数

*toLowerCase()*函数将串中的字符从大写转变为小写。例如：

```
// 将normal设置为 "this sentence has mixed caps!"  
normal="This Sentence Has Mixed Caps!".toLowerCase();
```

要抛开大小写的差别比较两个串，可以把它们都转换成同样的大小写形式，如下所示：

```
if(userEntry.toLowerCase() == password.toLowerCase()) {  
    // 它们得到了秘密的访问  
}
```

例4-3显示了我们如何用大小写的转换和串函数来让文本域中的文字具有动态效果，这挺有意思的。要这样做，你需要一个三帧的影片，并且有一个称为msgOutput的文本域，它处在自己的层中。例4-3中的代码位于*scripts*层。

### 例 4-3：字符大小写的动画

```
// 帧1中的代码
var i = 0;
var msg='my what fat cheeks you have';
function caseAni() {
    var part1=msg.slice(0,i);
    var part2=msg.charAt(i);
    var part2=part2.toUpperCase();
    var part3=msg.slice(i+1,msg.length);
    msg=part1+part2+part3;
    msgOutput=msg;
    msg=msg.toLowerCase();
    i++;
    if(i>(msg.length-1)) {
        i=0;
    }
}

// 帧2中的代码
caseAni();

// 帧3中的代码
gotoAndPlay(2);
```

## 字符编码函数

在前面，我们学习了如何将字符插入串中作为转义序列。ActionScript 也包括两个内置函数，用来处理串中的字符编码：*fromCharCode()*和*charCodeAt()*。

### fromCharCode()函数

我们可以调用*fromCharCode()*函数来创建任何字符或者字符序列。和其他的串函数不同，*fromCharCode()*并不由串常量或者包含串的标识符来调用，而是作为特殊*String*对象的一个方法，如下所示：

```
String.fromCharCode(code_point1,code_point2,...)
```

*fromCharCode()*的每一次调用都以*String.fromCharCode*作为开头。然后提供一个或者多个编码点（表示我们要创建的字符）作为参数。和 Unicode 类型的转义序列不同，*fromCharCode()*调用中的编码点表示为十进制的整数，而不是十六进制。如果你不熟悉十六进制数字，那么你会发现*fromCharCode()*比 Unicode 类型的转义序列更容易使用。下面给出了一些例子：

```
// 将lastName设置为'mock'  
lastName=String.fromCharCode(109,111,111,99,107);  
  
// 为了比较一下，我们需要做和Unicode类型的转义序列相同的事情  
lastName="\u006D\u006F\u006F\u0063\u006B";  
  
// 制作一个版权符号  
copyNotice=String.fromCharCode(169)+" 2001";
```

---

**注意：**如果你导出一个Flash 4的.swf文件，必须使用原Flash 4的字符创建函数*chr()*和*mbchr()*。虽然这些函数在向后兼容中得到支持，但是Flash 5首选*fromCharCode()*。

---

## charCodeAt()函数

要确定串中任何字符的编码点，我们使用*charCodeAt()*函数，它的格式如下：

```
string.charCodeAt(index)
```

*string*是任何直接量串值或者包含一个串的标识符，*index*是我们要检查的字符的位置。*charCodeAt()*函数返回一个十进制的整数，和*index*所指示字符的Unicode编码点相匹配。例如：

```
var msg="A is the first letter of the Latin alphabet.";  
trace(msg.charCodeAt(0));           // 显示：65 ("A"的编码点)  
trace(msg.charCodeAt(1));           // 显示：32 (空格的编码点)
```

我们通常使用*charCodeAt()*来处理那些我们不能直接从键盘输入的字符。例如，在下面所给出的代码中，我们检查一个字符是否为版权符号：

```
msg=String.fromCharCode(169)+" 2000";  
if(msg.charCodeAt(0)==169) {  
    trace("The first character of msg is a copyright symbol.");  
}
```

---

**注意：**如果你导出一个Flash 4的.swf文件，必须使用原Flash 4的编码点函数*ord()*和*mbord()*。虽然这些函数在向后兼容中得到支持，但是Flash 5首选*charCodeAt()*。

---

## 用 eval 执行串中的代码

在 ActionScript 中，*eval()* 函数将一个串转换为一个标识符。但是，要想深入了解 ActionScript 中的 *eval()* 函数，我们必须学习 JavaScript 中与之相似的 *eval()* 函数是如何运作的。在 JavaScript 中，*eval()* 是一个顶层的内置函数，它将任何串转换为一个代码块，然后执行这个代码块。JavaScript 中的 *eval()* 语法如下：

```
eval(string)
```

当 *eval()* 在 JavaScript 中得到执行的时候，解释程序将 *string* 转换为代码，并运行这些代码，然后返回结果值（如果能产生值的话）。考虑下面的 JavaScript 例子：

```
eval("parseInt('1.5')");           // 调用 parseInt() 函数，返回 1
eval('var x=5');                  // 创建一个新的变量 x，并将它的值设置为 5
```

如果你以前从来没有看到过 *eval()*，你可能会想：“我干嘛要把代码写成串放在里头呢？我何不将代码直接写在外面”？这是因为 *eval()* 能够让你在需要的时候动态地创建代码。例如，假设你有 10 个函数，它们是按顺序命名的：*func1*, *func2*, *func3*, ……, *func10*。你就可以用 10 个函数调用语句来执行这些函数：

```
func1();
func2();
func3();
// 依此类推
```

但是你同样可以在循环中使用 *eval()*，使它们的执行更加灵活，如下所示：

```
for(i=1;i<=10;i++) {
eval("func"+i+"()");
}
```

ActionScript 的 *eval()* 函数只支持 JavaScript 中 *eval()* 的一个功能子集：它只在参数是标识符的时候起作用。因此，ActionScript 的 *eval()* 函数只能获取和指定标识符对应的数据。例如：

```
var num=1;
var person1="Eugene";
trace(eval("person"+num));      // 显示 'Eugene'
```

虽然功能受到削减，但是 *eval()* 的用途还是很大的。下面，我们要在循环中动态地生成一系列的影片剪辑。我们用 *eval()* 作指示，将剪辑放在数组中：

```

for(var i=0;i<10;i++) {
    duplicateMovieClip('ballParent','ball'+i,i);
    balls[i]=eval('ball'+i);
}

```

但是要注意，*eval()*对处理器的要求很高。在需求更多的情况下，我们最好使用数组访问操作符来产生动态的剪辑引用。例如：

```

duplicateMovieClip("ballParent","ball"+i,i);
balls[ballCount]=_root["ball"+i];

```

在 Flash 4 中，*eval()*常常通过变量名的动态生成和引用来模仿数组。这种技术在 Flash 5 没有得到提倡，因为 Flash 5 本来就支持数组。参见第二章，可以看到更多的细节。

## Flash 4 和 Flash 5 的串操作符和函数

贯穿对串操作符和函数的所有描述，我们也看到了 Flash 4 中的类似技术。当我们使用 Flash 5 来创建 Flash 4 影片的时候，我们应该在所有工作中使用 Flash 4 的串操作符和函数。但是当我们创建 Flash 5 影片的时候，又要使用新潮的 Flash 5 操作符。如果你习惯于 Flash 4 的语法，可以参考表 4-2 中 Flash 5 的对应技术。

表 4-2 Flash 4 的操作符和函数，以及 Flash 5 的对应项

Flash 4 语法	Flash 5 语法	描述
''	" " 或 ''	串直接量
&	+ (或者是向后兼容中的 add)	串连接操作符
eq	= =	相等操作符
ge	> -	大于等于比较
gt	>	大于比较
le	< =	小于等于比较
lt	<	小于比较
ne	!=	不等操作符
<i>chr()</i> 或 <i>mbchr()</i>	<i>fromCharCode()</i> <sup>a</sup>	从编码数字创建一个字符
<i>length()</i> 或 <i>mblength()</i>	<i>length</i> <sup>a</sup>	Flash 4 中的函数，Flash 5 中为属性，它给出串中的字符数

表 4-2 Flash 4 的操作符和函数，以及 Flash 5 的对应项（续）

Flash 4 语法	Flash 5 语法	描述
<i>mbsubstring()</i>	<i>substr()</i>	从串中抽取字符序列
<i>ord()</i> 或 <i>mbord()</i>	<i>charCodeAt()</i> <sup>a</sup>	给出指定字符的编码点
<i>substring()</i>	<i>substr()</i> <sup>a</sup>	从串中抽取字符序列

a. 因为所有的 Flash 5 串操作符和函数都处理多字节的字符，因此在 Flash 5 中不像在 Flash 4 里那样能产生单字节的操作数。例如，*fromCharCode()* 函数就相当于 Flash 5 中的 *chr()*。*mblength()* 和 *length*，*mbsubstring()* 和 *substr()*，以及 *mbord()* 和 *charCodeAt()* 也是如此。

## 布尔类型

布尔数据用来表示正确或者错误的逻辑状态。因此，布尔数据类型只有两个合法值：*true* 和 *false*。注意，单词 *true* 和 *false* 并不需要使用引号，因为布尔数据并不是串数据。关键字 *true* 和 *false* 是保留的原始数据值，不能用作变量名或者标识符。

我们用布尔值来为代码的执行添加逻辑性。例如，我们可以将值 *true* 赋给一个变量，以追踪太空船的火力状态：

```
shipHasDoubleShots = true;
```

将 *shipHasDoubleShots* 和布尔直接量 *true* 进行比较，我们就可以确定击中目标的时候能够造成多大的损害：

```
if(shipHasDoubleShots==true) {
    // 用两倍火力击中目标
    // 如果比较结果为 true，这些代码就可以得到执行
} else {
    // 用单火力击中目标
    // 如果比较结果为 false，则执行这些代码
}
```

当双火力用完了的时候，我们可以将变量设置为 *false*：

```
shipHasDoubleShots = false;
```

这会让表达式 *shipHasDoubleShots==true* 变成 *false*，而在击中目标之后，单火力的脚本就可以得到执行。

所有的比较操作符都表示布尔值。当我们问：“用户的输入和密码一致吗？”答案就会以一个布尔值给出来：

```
// userGuess==password 将会产生值 true 或者 false
if(userGuess==password) {
    gotoAndStop("secretContent");
}
```

当我们问：“影片剪辑的旋转超过 90 度了吗？”答案还是一个布尔值：

```
// myClip._rotation>90 将会产生 true 或者 false
if(myClip._rotation>90) {
    // 如果旋转超过了 90 度就淡出
    myClip..alpha=50;
}
```

许多 ActionScript 的内部属性和方法都按照布尔形式来描述 Flash 影片环境。例如，如果我们问：“空格键按下了吗？”解释程序会回答一个布尔值：true(是)或false(不是)。

```
// key.isDown() 是一个函数，返回 true 或 false
if (key.isDown(Key.SPACE)) {
    // 空格键被按住了，因此要让你的太空船开火
}
```

在第五章中，我们会学习如何用布尔操作符来表示复杂的逻辑表达式。

## 用布尔值来创建预装载

我们来介绍一个布尔的应用实例。假设我们有一个文档，它有 500 个帧以及许多的内容。影片的真正开始处在第 20 帧，标签为 intro。我们将下面的代码放在影片主时间线的第 2 帧上：

```
if (_framesloaded>=_totalframes) {
    gotoAndPlay('intro');
} else {
    gotoAndPlay(1);
}
```

影片播放的时候，播放头进入到第 2 帧。ActionScript 解释程序到达这个条件语句，计算布尔表达式 `_framesloaded>=_totalframes` 的值。如果影片仍然在装载，那么 `_framesloaded` 就小于影片的总帧数 (`_totalframes`)。如果 `_framesloaded` 小于 `_totalframes`，那么表达式 `_framesloaded>=_totalframes` 就产生 `false`。因

此，语句 `gotoAndPlay("intro")` 就被跳过，执行的是语句 `gotoAndPlay(1)`。`gotoAndPlay(1)` 语句将播放头送回第 1 帧，播放影片。当播放头进入第 2 帧，我们的代码又按照上面的描述来执行。播放头保持这种循环，直到最终导致 `_framesloaded>=_totalframes` 产生 `true` 值（也就是所有的帧都已经装载完毕了）。这时候，语句 `gotoAndPlay("intro")` 得到执行，将播放头送到标签 `intro` 的地方。这样我们就能安全地开始播放影片，因为所有的帧都已经装载好了。

哇！你已经用布尔表达式创建了一个预装载。很好用的。我们将在第七章学习与布尔有关的条件语句和影片控制。

## undefined

我们到现在为止所探讨的大部分数据类型都是用来存储和操作信息的。`undefined` 数据类型的用途非常有限：它是用来检查一个变量是否存在，或者一个变量是否被设置了某个值。`undefined` 数据类型只有一个合法值，就是初始值 `undefined`。

当我们最初定义一个变量的时候，它的值会被默认为 `undefined`：

```
var velocity;
```

对于解释程序来说，前面的语句也就是：

```
var velocity-undefined;
```

要检查一个变量是否有值，我们可以将变量同 `undefined` 进行比较，如下所示：

```
if(myVariable!=undefined) {  
    // myVariable 是有值的，因此可以按照预想的处理过程进行下去……  
}
```

注意，`undefined` 值在作为串使用的时候，可以变成空串。例如，如果 `firstName` 是 `undefined`，那么下面的 `trace()` 语句将会显示 “”（空串）：

```
var firstName;  
trace(firstName);      // 什么也不显示（空串）
```

在 JavaScript 中，和上面相同的代码将会显示串 “`undefined`” 而不是空串。ActionScript 因为向后兼容的缘故，将 `undefined` 转化为 “”。

因为在 Flash 4 的 ActionScript 中没有 `undefined` 类型，许多 Flash 4 的程序都用空串来检查一个变量是否具有有效值。下面给出的代码是很常见的：

```
if (myVar eq "") {  
    // 什么也不能做： myVar 是无值的  
}
```

如果 Flash 5 将 `undefined` 转换为非 “” 的其他东西，上面的那种代码在 Flash 5 播放器中就会被中断。

注意，ActionScript 对不存在的变量以及虽定义但没有赋值的变量都返回 `undefined`。这与 JavaScript 也有所不同，在 JavaScript 中，引用一个不存在的变量会产生错误。

## null

从理论上来说，`null` 类型和 `undefined` 类型是一样的。和 `undefined` 数据类型一样，`null` 数据类型用来表示没有数据的情况，它只有一个合法值，即原始值 `null`。`null` 值不是由解释程序自动赋值的，而需要我们的人为处理。

我们将 `null` 赋给变量、数组元素或者对象属性，以表示不包含数字、串、布尔、数组或者对象合法值的特定数据容器。

注意，`null` 只等于自身和 `undefined`：

```
null==undefined; // true  
null==null; // true
```

## 小结

我们不知不觉又学习了很多东西。我们需要打下一个坚实的基础，以便理解后面章节中更高级的内容。如果你还没有掌握所有的细节，也不要着急。只要需要，你随时可以重新开始训练。在以后的章节中我们将学习如何合并和转换数据，并研究一些形式更完善的应用事例。

# 第五章

## 操作符

一个操作符就是一个符号或者关键字，它可以操作、合并或者转换数据。如果你是一个编程新手，你会注意到对一些数学操作符，比如 +（加法）和 -（减法）操作符非常熟悉。在其他情况下，即使你非常熟悉概念，也仍然需要学习特定的编程语法。例如，要让两个数字相乘，ActionScript 使用符号 \*（乘法操作符）来代替你在高中所学到的 x 符号。下面的代码表示 5 和 6 相乘：

```
5 * 6;
```

### 操作符的一般特点

虽然每种操作符都有它自己的指定任务，但是它们有许多共同特征。在我们讨论操作符之前，还是先来看看它们的基本特性。

### 操作符和表达式

操作符对所提供的数据（操作数）执行一些动作。例如，在运算  $5 * 6$  中，数字 5 和 6 就是乘法操作符 (\*) 的操作数。操作数可以是任何类型的表达式，例如：

```
player1score+bonusScore;           // 操作数是变量  
(x+y) (Math.PI * radius * radius); // 操作数是复杂表达式
```

观察第二个例子，-操作符两边的操作数都是表达式，它们各自又包括其他的运算。我们可以用复杂表达式来创建更大的表达式，比如：

```
((x+y)-(Math.PI * radius * radius))/2 // 将整个东西除以 2
```

当表达式变得更大的时候，可以考虑使用变量来存储临时结果，这样不但方便，而且更清楚。记住在命名变量的时候要对其进行描述，比如：

```
var radius=10;
var height=25;
var circleArea=(Math.PI * radius * radius);
var cylinderVolume=circleArea * height;
```

## 操作数的数目

操作符有的时候按照所带操作数的数量来进行分类。一些 ActionScript 中的操作符只带一个操作数，一些带两个，有的甚至带三个：

```
-x // 一个操作数
x * y // 两个操作数
(x==y) ? "true result" : "false result" // 三个操作数
```

单操作数操作符被称为一元操作符，带两个操作数的操作符叫做二元操作符，带三个操作数的操作符就称为三元操作符。从我们的角度来说，关心的是操作符究竟能做什么，而不是它带多少操作数。

## 操作符优先规则

操作符优先规则可以确定在一个有多操作符的表达式中应该先执行哪一个运算。例如，当乘法和加法出现在同一个表达式中的时候，乘法是优先执行的：

```
4 + 5 * 6 // 产生 34，因为 4+30=34
```

表达式  $4 + 5 * 6$  是作为  $4 + (5 * 6)$  来执行的，因为 \* 操作符比 + 操作符有更高的优先级。如果搞不清楚优先级，或者要确保运算的顺序，可以使用括号，它的优先级是最高的：

```
(4 + 5) * 6 // 产生 54，因为 9*6=54
```

虽然没有严格的必要，括号也可以让一个复杂的表达式看起来清楚一点。表达式：

```
// x比y大或者y等于z
x>y || y==z
```

如果不借助优先级列表，也许很难判断这个表达式的优先情况。但是加上括号就清楚多了：

```
(x>y) || (y==z) // 好多了！
```

表 5-1 给出了每一个操作符的优先级。优先级最高的操作符（在表的顶层）首先执行。同优先级的操作符按照从左到右的顺序来执行。

表 5-1 ActionScript 操作符的结合顺序及优先级

操作符	优先级	顺序	描述
<code>x++</code>	16	从左到右	后缀增量
<code>x--</code>	16	从左到右	后缀减量
<code>.</code>	15	从左到右	对象属性访问
<code>[]</code>	15	从左到右	数组元素访问
<code>()</code>	15	从左到右	括号
<code>function()</code>	15	从左到右	函数调用
<code>++x</code>	14	从右到左	前缀增量
<code>--x</code>	14	从右到左	前缀减量
<code>-</code>	14	从右到左	一元非
<code>~</code>	14	从右到左	位逻辑 NOT
<code>!</code>	14	从右到左	逻辑 NOT
<code>new</code>	14	从右到左	创建对象 / 数组
<code>delete</code>	14	从右到左	删除对象 / 属性 / 数组元素
<code>typeof</code>	14	从右到左	确定数据类型
<code>void</code>	14	从右到左	返回 <code>undefined</code> 值
<code>*</code>	13	从左到右	乘法
<code>/</code>	13	从左到右	除法
<code>%</code>	13	从左到右	模除法
<code>+</code>	12	从左到右	加法或者串连接
<code>-</code>	12	从左到右	减法

表 5-1 ActionScript 操作符的结合顺序及优先级（续）

操作符	优先级	顺序	描述
<<	11	从左到右	位逻辑左移
>>	11	从左到右	带符号的位逻辑右移
>>>	11	从左到右	无符号的位逻辑右移
<	10	从左到右	小于
<=	10	从左到右	小于等于
>	10	从左到右	大于
>=	10	从左到右	大于等于
==	9	从左到右	等于
!=	9	从左到右	不等于
&	8	从左到右	位逻辑 AND
^	7	从左到右	位逻辑 XOR
	6	从左到右	位逻辑 OR
&&	5	从左到右	逻辑 AND
	4	从左到右	逻辑 OR
? :	3	从右到左	条件
=	2	从右到左	赋值
+=	2	从右到左	加和重赋值
-=	2	从右到左	减和重赋值
*=	2	从右到左	乘和重赋值
/=	2	从右到左	除和重赋值
%=	2	从右到左	模除法和重赋值
<<=	2	从右到左	左位移和重赋值
>>=	2	从右到左	右位移和重赋值
>>>=	2	从右到左	右位移（无符号）和重赋值
&=	2	从右到左	位逻辑 & 和重赋值
^=	2	从右到左	位逻辑 XOR 和重赋值
=	2	从右到左	位逻辑 OR 和重赋值
,	1	从左到右	逗号

## 操作符的结合

正如我们已经学到的，操作符优先级表示操作符的运算优先顺序：优先级高的比优先级低的操作符先执行。但是如果多个操作符同时出现，并具有同样的优先级时应该怎么办呢？在这种情况下，我们给出了操作符的结合顺序规则，它表示运算的方向。操作符是左结合（从左到右执行）或者右结合（从右到左执行）的。例如，我们看看下面的表达式：

```
a=b * c / d
```

\* 和 / 操作符是左结合的，因此，左边的 \* 运算 ( $b * c$ ) 先执行。前边的例子就等于是：

```
a=(b * c)/d
```

相反，等号 = 操作符是右结合的，因此表达式：

```
a=b=c=d
```

是说“将 d 赋给 c，然后将 c 赋 b，然后将 b 赋给 a，”也就是：

```
a= (b= (c=d))
```

操作符的结合是非常直观的，但是如果你觉得其值有点无法意料，可以参照表 5-1 或者添加括号。我们会注意到，在本章的后面，操作符的结合是非常容易引起错误的。

## 数据类型和操作符

一些操作符可以接受多种类型的数据来作为操作数。操作数的数据类型不同，操作符的结果可能会变化。例如，+ 操作符在处理数字操作数的时候执行的就是数学加法，但在处理串操作数的时候执行的是串的连接。如果操作数属于不同的类型，或者属于错误的类型，ActionScript 会按照第三章中描述的规则来执行类型的转换，这会对代码产生很大的影响。

## 赋值操作符

我们已经多次使用了赋值操作符。它可以将一个值放到变量、数组元素或者对象属性中。赋值操作符的格式为：

```
identifier = expression
```

*identifier*可以是要赋值的变量、数组元素或者对象属性。*expression*表示要用 来存储的值。例如：

```
x=4;           // 将 4 赋值给变量 x
x=y;           // 将 y 的值赋给变量 x
name='d'; duong'; // 将一个串赋给变量名
products[3]='Flash'; // 将一个串赋给 products 数组的第 4 个元素

// 将一个数字赋给 square 的 area 属性
square.area=square.width * 2;
```

我们也可以一次执行多个赋值操作，如下所示：

```
x=y=4; // 将 x 和 y 都赋值为 4
```

记住，赋值操作符有从右到左的结合顺序，因此，4 首先赋值给 y，然后将 y 的值（现在是 4）赋值给 x。

## 运算和赋值的合并

赋值运算通常用来设置变量的新值，有时候新值会和它原来的值有一定关系。例如：

```
counter=counter+10; // 将 counter 的当前值加上 10
xPosition=xPosition+xVelocity; // 在 xPosition 上加 xVelocity
score=score/2; // 用 2 来除 score
```

ActionScript 支持赋值的一个速记形式，称为复合赋值 (compound assignment)，它将诸如 +、-、/ 这些操作符同赋值操作符合并起来，执行一个“计算兼赋值”操作。这样，如果要把加法和赋值结合起来，就使用 +=。要将除法和赋值结合起来，就使用 /=。因此，前边的例子可以用复合赋值表达得更简单点，如下所示：

```
counter+=10;  
xPosition+=xVelocity;  
score/=2;
```

表 5-1 中列出了复合赋值操作符。

## 算术操作符

算术操作符对数字操作数执行算术运算。如果你对算术操作符使用非数字的操作数，ActionScript 会将数据自动转换为数字。例如，`false - true` 将得到 `-1`，因为 `false` 被转换为数字 `0`，而 `true` 被转换为数字 `1`。相似的，表达式 `"3" * "5"` 结果为 `15`，因为串 `"3"` 和 `"5"` 在执行操作之前分别被转换成了数字 `3` 和数字 `5`。但是，`+` 操作符有一个特殊的情况：用于至少有一个串操作数的运算时，它执行的是串的连接运算，而不是数字加法。

如果将非数字操作数转换为数字操作数失败了，操作数就会被设置为特殊的数字值 `Nan`。整个运算的结果也为 `Nan`。表 3-1 中就数字的转化给出了详细的规则。

### 加法

加法操作符返回两个操作数的总数：

```
operand1 + operand2
```

为了返回有意义的数字结果，`+` 操作符的操作数应该是能产生数字值的表达式，比如：

```
234+5          // 返回 239  
(2 * 3 * 4)+5 // 返回 29
```

如果加法操作符有一个以上的操作数是串，那么它在算术操作符中是独特的，它执行的将是串的连接。这在第四章中有所描述。

### 增量

作为一种方便的加法工具，增量操作符接收一个单一的操作数，然后对它的当前值加 `1`。增量运算通常有两种形式：前缀增量和后缀增量，如下所示：

```
++operand          // 前缀增量  
operand++         // 后缀增量
```

在两种形式中，增量都是对变量、数组元素或者对象属性加 1，如下所示：

```
var x=1;  
x=x+1;           // x现在为2(使用的语法比较冗长)  
x++;             // 加1; x现在是3  
++x;             // 加1; x现在为4
```

如果单独使用，前缀和后缀增量并没有什么区别，只不过后缀增量更常用一些。

但是，如果用在较大的表达式中，前缀和后缀增量有不同的作用：前缀增量将 *operand* 加 1，然后返回 *operand+1* 的值；后缀增量将 *operand* 加 1，然后返回 *operand* 自身的值，而不是 *operand+1*：

```
var x=1;  
// 后缀增量: y被设置为1, 然后x增值为2  
var y=x++;  
  
var x=1;  
// 前缀增量: x首先增值, 然后y被设置为2  
var y=++x;
```

我们将会在第八章中重新看到增量操作符。

## 减法

减法操作符从第一个操作数中减去第二个操作数。它的一般格式为：

*operand1 - operand2*

操作数可以是任何有效的表达式。如果其中有一个操作数不属于数字类型，并且无法转换为实数，那么运算就产生 NaN：

```
234 - 5      // 产生229  
5 - 234      // 产生-229
```

要确定两个数字之间的绝对值（也就是正数）差异，可以参见第三部分中的 *Math.abs()* 方法。

## 减量

减量操作符和增量操作符类似，是从它的操作数当前值上减1而不是加1。和增量一样，减量有两个常用格式，称为前缀减量和后缀减量，如下所示：

```
--operand      // 前缀减量  
operand--      // 后缀减量
```

在两种形式中，它们都从变量、数组元素或者对象属性上减去1。前缀减量从 *operand* 上减去1，返回 *operand - 1* 的值；后缀减量从 *operand* 上减1，但返回的是 *operand* 本身的价值，而不是 *operand - 1*。和增量操作符的情况一样，减量的形式只在操作数是表达式的一个部分时才会变得重要：

```
var x=10;  
var y;  
x-=1;           // x 现在是 9  
x--;           // x 现在是 8  
--x;           // x 现在为 7  
y=-x;           // y 现在为 6, x 为 6  
y-=x--;        // y 仍然是 6, x 是 5
```

## 乘法

乘法操作符将两个数字操作数相乘，返回结果（也就是乘积）。乘法的一般格式如下：

```
operand1 * operand2
```

操作数可以是任何有效的表达式。用来相乘的 \* 符号也就是传统算术中所使用的 ×（乘以）符号。如果有一个操作数不是数字类型，而且不能转换为实数，那么运算的结果为 NaN：

```
6 * 5 // 返回 30
```

## 除法

除法操作符用第二个操作数（除数）来除第一个操作数（被除数），返回结果（也就是商）。除法的格式如下：

```
operand1 / operand2
```

操作数必须是有效的数字表达式。用在除法中的 / 符号也就是传统算术计算中的 ÷ 号。如果有一个操作数不是数字类型的，并且不能转换为实数，则运算产生 NaN。如果要表示小数结果，那么即使两个操作数都是整数，商也将是一个浮点数字。

```
20/5 // 返回 4  
5.1 // 返回 1.25，在其他语言中，结果可能会是 1，而不是 1.25
```

注意，一些其他的语言，比如 Director 的 Lingo 语言，返回的是一个整数，除非操作数中至少有一个是浮点数。

如果除数（分母）为 0：

若分子为 0 或为非数字，则结果为 NaN；  
若分子为一正数，则结果为 Infinity；  
若分子为一负数，则结果为 -Infinity。

例如：

```
trace(0/0); //NaN  
trace("a"/0); //NaN  
trace(1/0); //Infinity  
trace(-1/0); //-Infinity
```

注意，如果分子为 Infinity：

若除数为 Infinity、-Infinity 或非数字，则结果为 NaN；  
若除数为 0 或为有限正数，则结果为 Infinity；  
若除数为负的有限数，则结果为 -Infinity。

比如：

```
trace(Infinity/Infinity); //NaN  
trace(Infinity/-Infinity); //NaN  
trace(Infinity/"a"); //NaN  
trace(Infinity/0); //Infinity  
trace(Infinity/1); //Infinity  
trace(Infinity/^000); //Infinity  
trace(Infinity/-1); // -Infinity  
trace(-Infinity/2); // -Infinity  
trace(-Infinity/-2); // Infinity
```

最后要注意，任何有限数字被 Infinity 来除都等于 0：

```
trace(0/Infinity); //0  
trace(1/Infinity); //0  
trace(-1000/Infinity); //0
```

如果除数有可能为 0，那么应该在执行除法之前检查它的值，如下所示：

```
if (numItems != 0) {  
    trace ("Average is' + total / numItems);  
} else  
    trace ("There are no items for which to calculate the average");  
}
```

注意，在有的语言里，除数为 0 会引起错误。

## 模除法

模操作符执行所谓的模除法。它返回第一个操作数被第二个操作数所除时产生的余数（也就是模）。模除法的格式为：

```
operand1 % operand2
```

例如， $14 \% 4$  返回值 2，因为 4 除 14 得到商 3，还剩下 2 为余数。

模的操作数可以是任何有效的数字表达式，包括整数（不像 C 和 C++）和浮点数字。例如， $5 \% 4$  得到 1，而  $5 \% 4.5$  得到 0.5。如果有某个操作数不是数字类型，而且不能被转化为实数，那么运算就得到 NaN。

如果数字为偶数，那么我们用 2 来除的时候模就将是 0。我们可以用例 5-1 给出的技巧来测试一个数字是奇数还是偶数。

### 例 5-1：用模除法来测试偶数

```
var x = 3;  
if (x%2 == 0) {  
    trace('x is even');  
} else {  
    trace("x is odd");  
}
```

## 一元非

一元非操作符只有一个操作数。它将转换操作数的符号（也就是将正的变成负的，负的变成正的）。一元非的格式如下：

-operator

操作数可以是任何有效的表达式。下面，测试一下某个东西的水平坐标是否比正数限制大，或者比负数限制小：

```
if(xPos>xBoundary || xPos < xBoundary) {  
    // 我们已经走得远了  
}
```

## 等于和不等操作符

等于操作符（`==`）用来测试两个表达式是否拥有相同的值。等于测试的格式如下：

`operand1 == operand2`

`operand1` 和 `operand2` 可以是任何有效的表达式。等于操作符可以比较任何类型的操作数。当 `operand1` 和 `operand2` 相等的时候，表达式就返回布尔值 `true`。否则，返回布尔值 `false`。例如：

```
var x=2;  
x-=1 // false  
x-=2 // true
```

---

**注意：**等于操作符是用两个连续的等号来创建的（`==`）。它确定两个表达式是否相等，不能和赋值操作符（`=`）混同起来，后者用来为变量赋一个新的值。

---

考虑下面的例子：

```
if (x = 5) //  
    trace ("X is equal to 5")  
}
```

前面的例子并不检查 `x` 是否等于 5。它只会将 `x` 设置为 5。正确的表达方式如下：

```
// 用 ==, 而不用 =  
if(x==5) {  
    trace("x is equal to 5")  
}
```

## 原始数据类型的等同式

对于原始数据类型来说，大部分等于测试的结果都是很直观的。表 5-2 列出了每一种原始数据的等于控制规则。

表 5-2 原始数据的等同性

类型	等同性形式（两个操作数必须是给定的类型）
数字	<p>如果 <code>operand1</code> 和 <code>operand2</code> 是相同的数字，那么结果就是 <code>true</code>。如果两个操作数都是正无穷或者都是负无穷，那么结果为 <code>true</code>。如果两个操作数都是 <code>-0</code> 或者 <code>+0</code>，结果是 <code>true</code>。对于其他所有的情况，包括操作数中至少有一个为 <code>NaN</code> 的情况，都将得到 <code>false</code>：</p> <pre>1 == 4           // false 4 == 4           // true NaN == NaN      // false +Infinity == -Infinity // false</pre>
串 <sup>a</sup>	<p>执行区分大小写的串比较。如果 <code>operand1</code> 和 <code>operand2</code> 有相同的长度，并且包括完全一样的字符序列，那么结果就是 <code>true</code>；否则，得到 <code>false</code>：</p> <pre>"Flash" == "Flash"      // true "O'Reilly" == "O Reilly" // false "Moock" == "moock"       // false ("m" 和 "M" 不是相同的字符)</pre>
布尔	<p>如果两个操作数都是 <code>true</code> 或者都是 <code>false</code>，那么结果就是 <code>true</code>；否则，结果是 <code>false</code>：</p> <pre>true == true // true false == false // true true == false // false</pre>
<code>undefined</code>	<p>如果两个操作数都是 <code>undefined</code> 或者其中有一个是 <code>undefined</code>，另外一个是 <code>null</code>，那么结果就是 <code>true</code>；否则，结果是 <code>false</code></p>
<code>null</code>	<p>如果两个操作数都是 <code>null</code>，或者一个是 <code>undefined</code> 另一个 <code>null</code>，结果就为 <code>true</code>；否则，结果为 <code>false</code></p>
复合数据类型	参见下面“复合数据类型的等同性”一节

a. Flash 4 的串等于操作符是 `eq`，而 `eq` 在 Flash 5 的向后兼容中可以得到支持。除非要导出为 Flash 4 的 `.swf` 格式，否则不建议使用。

## 复合数据类型的等同性

因为包含了复合数据（对象、数组、函数或者影片剪辑）的变量存储的是数据的引用而不是数据本身，因此，有可能两个变量指向同一个基层项目。两个这样的操作数当且仅当它们指向相同的基层复合数据时才被认为是相等的，如果两个操作数指向不同的项目，即使这两个项目包含相同的内容，操作数也不相等。虽然两个操作数可以被转换为同样的原始值，但它们仍然会被认为不相等。

下面的例子展示了，ActionScript如何比较指向复合数据的引用，而非数据本身。在第一个例子中，操作数（nameList1 和 nameList2）指向的两个数组虽然有相同的元素，但实际上是指向两个不同的数组。因此，引用是不同的，比较的结果就是 false：

```
nameList1=['Linkovich', 'Harris', 'Sadler'];
nameList2=["Linkovich", "Harris", "Sadler"];
nameList1==nameList2 // false
```

在这个例子中，cities 和 canadianCities 都指向同一个数组：

```
canadianCities=[ 'Toronto', 'Montreal', 'Vancouver'];
cities=canadianCities;
cities==canadianCities // true
```

在本例中，myFirstBall 和 mySecondBall 有相同的结构（也就是都来源于同一个类），但是它们作为单独（不等）的实例而存在：

```
myFirstBall=new Ball();
mySecondBall=new Ball();
myFirstBall==mySecondBall // false
```

因此，对复合数据值的等于测试也就是引用的比较，而不是值的比较。对于其中差别的更多细节，可以参见第十五章。

要复制一个数组的内容，而不复制数组的引用，我们可以使用 *Array.slice()* 方法。在本例中，我们从 dishes 数组中复制元素到 kitchenItems：

```
dishes=["cup", "plate", "spoon"];
kitchenItems=dishes.slice(0,dishes.length);
trace(kitchenItems==dishes); // 显示: false
```

现在，kitchenItems 和 dishes 都包含自己独有的数组元素拷贝，它们各自发生变化并不会影响对方。

## 等同性和数据类型的转换

我们已经看到了两个操作数有相同的数据类型时的等于测试，但是，当我们把两个不同类型的数据进行比较的时候又会怎么样呢？比如将数字和串进行比较，如下所示：

```
"asdf" == 13;
```

当操作数有不同的类型时，解释程序在比较之前要执行一个数据类型的转换。下面是解释程序在执行类型转化中要遵循的规则：

1. 如果两个操作数属于同样的类型，就对它们进行比较，然后返回结果。（如果 `null` 和 `undefined` 比较，结果返回 `true`。）
2. 如果有一个操作数是数字，而另一个操作数是串，就将串转换为数字，然后回到第 1 步。
3. 如果有一个操作数是布尔值，就将布尔值转换为数字（`true=1, false=0`），然后回到第 1 步。
4. 如果一个操作数是对象，调用对象的 `valueOf()` 方法将它转换为一个原始类型。如果无法进行转换就返回 `false`。否则，回到第 1 步。
5. 如果前面的步骤不能获得一个有效的结果就返回 `false`。

注意，如果一个操作数是对象而另外一个是布尔，那么布尔值就被转换为数字，同对象的原始值进行比较。这意味着 `someObject==true` 这样的等式通常都将得到 `false`，即使 `someObject` 存在，因为 `someObject` 被转换为数字或者串来进行比较，而 `true` 被转换为数字 1。要强制 `someObject` 在比较中被当做布尔，我们可以使用 `Boolean()` 函数，如下所示：

```
Boolean(someObject)==true           // 如果 someObject 存在就返回 true  
                                  // 如果不存在就返回 false
```

由等于运算引起的类型转换更青睐于数字类型。如果你对刚才描述的各种类型转换结果不大清楚，可以参见第三章中的介绍。

注意，在比较的时候引起的类型转换不影响项目原来存储的值或者其数据类型。一旦表达式求值结束，临时转换的结果就被丢弃了。

## 不等操作符

不相等或者不等于（或者不等）操作符返回等于操作符比较结果的布尔值的反值。我们说“如果  $x$  不等于  $y$ , 就做这个”比说“如果  $x$  等于  $y$  就什么也别做, 否则做这个”要容易理解一点。例 5-2 中表明了这个意思。不等操作符的格式为:

```
operand1 != operand2
```

例如:

```
var a=5;
var b=6;
var c=6;
a!=b          // true
b!=c          // false
```

不等操作符遵循与等于操作符同样的类型转换规则, 它总是产生相反的结果, 包括使用 NaN 作为一个操作数的时候:

```
NaN!=7        // true
NaN!=NaN      // true!
```

在一些语言中, 包括 Flash 4 ActionScript, <> 操作符被用作不等操作符。稍后还可以参见 NOT 操作符 (!) 的讨论。

## 等于操作符的一般使用

我们常常要用等于运算来在条件语句中构成布尔表达式, 或者将布尔值赋给变量。例 5-2 给出了 != 和 == 操作符在实际使用中的情况和示范。

### 例 5-2: 使用等于和不等操作符

```
version=getVersion();           // 获取播放器版本

// 检查串 "WIN" 是否在版本中。如果是, 将 isWin 设置为 true, 否则设置为 false
isWin=(version.indexOf("WIN") != -1);

// 如果 isWin 为 true...
if (isWin==true) {
    // ...在这里执行 Windows 特定的动作
    trace("Please use IE4 or later on Windows.");
}
```

经验丰富的程序员很快就可以发现，我们将 `if(isWin==true)` 减缩成了 `if(isWin)`，因为 `isWin` 包含一个布尔值，这正是 `if` 语句所需要的东西。虽然这是正确的，但是，要作为`==`的例子就似乎有点不合适了，对吗？而且，一个新的程序员常常会觉得繁琐的那种形式看起来更清楚一点。两种方法都是可以接受的。我在第七章中会对这个主题给出更多内容。

## 比较操作符

比较操作符（也叫做关系操作符）用来确定两个值中哪一个在给定的某种顺序里应该先出现。和等于以及不等操作符一样，比较操作符返回一个布尔值，`true` 或 `false`，描述比较中的关系是正确的 (`true`) 或者是错误的 (`false`)。

比较操作符只处理串和数字。当比较操作符的两个操作数是数字的时候，比较的执行是数学性的：`5<10` 得到 `true`，`-3<-6` 得到 `false`，依此类推。当一个比较操作符的两个操作数都是串时，比较的执行按照字符编码点来进行，如附录二中所示。也可以参见第四章以获得与此相关的更多内容。

解释程序会将用在比较运算中的任何非串或非数字数据转换为串或者数字类型。我们在讨论了比较操作符本身之后，再考虑比较运算中数据类型转换的作用。

### 小于操作符

小于操作符的格式如下：

```
operand1<operand2
```

如果操作数是数字，`operand1` 在数学意义上比 `operand2` 小，小于操作符就返回布尔值 `true`：

```
5<6          // true
5<5          // false, 它们是相等的, 5 并不小于 5
-3<-6        // false, -3 比 -6 大
6<-3         // true, 6 比 -3 小
```

如果操作数是串，`operand1` 在字母表顺序中排在 `operand2` 前边（参见附录二），小于操作符就返回 `true`；否则返回 `false`：

```
"a" < "z"          // true, 小写的'a'在小写'z'之前出现  
"A" < "a"          // true, 大写字母都在小写字母之前  
"Z" < "a"          // true, 大写字母都在小写字母之前  
"hello" < "hi"    // true, "e" 小于 "i"
```

## 大于操作符

大于操作符的格式为:

```
operand1 > operand2
```

如果操作数是数字, *operand1*在数学意义上比*operand2*大, 大于操作符就返回布尔值 *true*:

```
5 > 6              // false  
5 > 5              // false, 它们是相等的, 5 并不大于 5  
-3 > -6            // true, -3 比 -6 大  
-6 > -3            // false, -6 比 -3 小
```

如果操作数是串, *operand1*在字母表顺序中排在*operand2*后边(参见附录二), 大于操作符就返回 *true*; 否则返回 *false*:

```
"a" > "z"          // false, 小写的'a'在小写'z'之前出现  
"A" > "a"          // false, 大写字母都在小写字母之前  
"Z" > "a"          // false, 大写字母都在小写字母之前  
"hello" > "hi"    // false, "e" 小于 "i"
```

## 小于等于操作符

小于等于操作符的格式为:

```
operand1 <= operand2
```

如果操作数是数字, *operand1*在数学意义上小于等于*operand2*, 小于等于操作符就返回布尔值 *true*:

```
5 <= 6              // true  
5 <= 5              // true, 注意和 5 < 5 的不同  
-3 <= -6            // false  
-6 <= -3            // true
```

如果操作数是串, *operand1*在字母表顺序中排在*operand2*前边, 或者按照第四章中所描述的规则操作数是相等的, 小于等于操作符就返回 *true*; 否则返回 *false*:

```
'a'<="z"           // true, 小写的'a'在小写'z'之前出现
'A'<="a"           // true, 大写字母都在小写字母之前
"Z"<="a"           // true, 大写字母都在小写字母之前
"hello"<="hi"      // true, 'e' 小于 'i'
```

注意，`<=`操作符的写法是在小于号后面加一个等号，这种操作符是非法的：`=<`。

## 大于等于操作符

大于等于操作符的格式为：

```
operand1>=operand2
```

如果操作数是数字，`operand1`在数学意义上大于等于`operand2`，大于等于操作符就返回布尔值`true`：

```
5>=6               // false
5>=5               // true, 注意和<>5的不同
-3>= -6            // true
-6>= -3            // false
```

如果操作数是串，`operand1`在字母表顺序中排在`operand2`后边，或者按照第四章中所描述的规则操作数是相等的，大于等于操作符就返回`true`；否则返回`false`：

```
'a'>='z'          // false; 小写的'a'在小写'z'之前出现
'A'>='a'          // false; 大写字母都在小写字母之前
"Z">='a'          // false; 大写字母都在小写字母之前
"hello">="hi"     // false; "e" 小于 "i"
```

注意，`>=`操作符的写法是在小于号后面加一个等号，这种操作符是非法的：`=>`。

## 比较操作符和数据类型的转换

在大部分情况下，当我们使用比较操作符的时候，我们比较的都是数字。因此，比较运算中的类型转换倾向于转换为数字类型。当任何比较操作符的两个操作数属于不同的类型时，或者当两个操作数都不是数字或串的时候，类型的转换按照下面的步骤进行：

1. 如果两个操作数都是数字，就按照数学的规则比较两个数，返回结果。如果其中某个数字是（或者两个都是）NaN，比较的结果就是false，除非使用的是!=操作符。
2. 如果两个操作数都是串，可以用附录三中给出的编码点来比较两个数的字母顺序，然后返回结果。
3. 如果操作数一个为数字一个为串，那么就将串转换为数字，然后回到第1步。
4. 如果某个操作数是布尔值、null或者undefined，就将该操作数转换为数字，然后返回第1步。
5. 如果某个操作数是对象，先调用valueOf()方法将对象转换为原始值，然后返回第1步。如果valueOf()方法转换失败或者不能返回一个原始值，就返回false。
6. 返回false。

注意，比较中所执行的类型转换不改变原来项目所存储的值或者数据类型。临时转换的结果一旦表达式求值完成就会被丢弃。

下面是一个简单的转换例子，对两个布尔值进行比较：

```
false<true           // true: 0 小于 1
```

比较操作符总是将复合数据类型转换为串或者数字来进行比较。在下面给出的例子中，因为someObj和someOtherObj都是Object类的成员，它们的串值“[objectObject]”就是相同的：

```
someObj=new Object();
someOtherObj=new Object();
someObj<=someOtherObj;           // true
```

在下面的例子中，即使“A”的编码点为65，将“A”转换为数字还是将产生NaN，这意味着整个表达式将产生false。使用charCodeAt()函数来检查一个串的编码点：

```
"A"<=9999             // false
"A".charCodeAt(0)<9999 // true
```

## 串操作符

在Flash 4中使用串数据的时候，要求用特殊的串操作符——连接(&)、等于(eq)、不等(ne)和比较(lt, gt, le, ge)。在Flash 5及以后的版本中，并不提倡特殊的串操作符（也就是说它们虽然也可以得到支持，但是已经过时了）。一般使用!=, ==, <, >, <= 和 >= 来进行串的比较，除非要导出 Flash 4 的.swf 格式文件，那时必须使用原来的操作符(eq, ne, lt, gt, le, ge)。用+来进行串的连接，但在导出 Flash 4 的格式文件时例外，必须要使用 add 来代替&（因为&是 Flash 5 及其以后版本中所使用的位逻辑 AND 操作符）。

参见第四章和表 4-2 可以得到更多的细节，以及 Flash 4 与 Flash 5 串操作对比列表。

## 逻辑操作符

在第四章中，我们学习了如何使用布尔值来作出逻辑决定。我们的决定基于单独的因素：如果影片还没有载入就不开始播放；如果一个太空船有双火力，就提高它击中目标时的损害程度。依此类推。但是，并非所有的编程逻辑都这么简单。我们经常需要在分支逻辑（也就是做出决定）中考虑多种因素。

例如，假设我们正在创建一个个人的 Flash 站点，要求用户登录。如用户作为客人来登录，他可以看到的内容就是有限的。当一个已经注册的用户登录到站点的时候，我们就设置一个变量，用来表示在影片播放期间该用户可以访问特权内容：

```
var userStatus="registered";
```

要决定是否可以访问限制内容，可使用一个布尔值来检查，如下所示：

```
if(userStatus=="registered") {  
    // ok, 你可以进来了……  
}
```

假设我们想向其他用户展示该站点，但又不强迫他们注册，同时不让他们访问到为注册用户提供的一些保留内容。那么可以设置一个新的用户类别“specialGuest”，它允许访问者看到比一般客人更多的内容，但同时又不及注册用户。要将某个人定为特殊客人时，可以将 userStatus 设置为“specialGuest”。

既然有两种类型的用户都可以进来，那么如何执行布尔检查呢？下面的代码段是很不明智的：

```
if (userStatus=="registered") {  
    // 执行合法用户代码……  
}  
  
if (userStatus=="specialGuest") {  
    // 执行同样的合法用户代码……  
}
```

显然，这会让人非常头痛。我们添加到脚本中的每一种复杂情况都会增加脚本长度，我们最后将陷入严重的版本控制问题。

我们可以执行这样一个单独的布尔检查：“如果用户已经注册，或者是特殊客人，都允许进入。”这样，就可以用布尔逻辑操作符产生复合的逻辑短语。下面，逻辑 OR 操作符（||）可以在一个单独的表达式中检查多个布尔操作的状态：

```
if (userStatus=="registered" || userStatus=="specialGuest") {  
    // 执行合法用户代码……  
}
```

很简洁，是不是？有的时候编程是非常讲究的。比如独特的垂直线……比如括号的提供……

## 逻辑 OR

逻辑 OR 操作符通常用来在至少有一个条件得到满足的时候执行某些行动。例如，“如果我饿了或者我渴了，我就去厨房。”逻辑 OR 的符号用两条垂直线（||）来表示。同时按下键盘上的 Shift 键和键盘右上角的 (\) 键（它在那里表现为一个直杠）就可以得到 | 符号。逻辑 OR 的格式为：

operand1 || operand2

当两个操作数都是布尔表达式的时候，逻辑 Or 在其中某个操作数为 true 的时候返回 true，在两个操作数都为 false 的时候返回 false。比如：

```
true || false      // true, 因为第一个操作数为 true  
false || true      // true, 因为第二个操作数为 true  
true || true       // true, (两个操作数都为 true)  
false || false     // false, 因为两个操作数都是 false
```

前面的例子只表示出了很简单的情况，也就是两个操作数都是布尔类型的时候，但是操作数其实可以是包括非布尔类型在内的任何有效表达式。在这种情况下，逻辑 OR 运算的返回值不一定是布尔值（true 或 false）。从理论上讲，处理逻辑 OR 的第一个步骤是将 operand1 转换为布尔，如果转换的结果是 true，那么逻辑 OR 就返回 operand1 求出的值；否则逻辑 OR 就转向 operand2 进行求值。下面是一些例子：

```

null || "hi there!"    // Operand1 不是 true，因此运算返回
                        // operand2 的值 "hi there!"

true || false          // Operand1 为 true，因此运算返回 operand1 的值 true

"hey" || "dude"        // Operand1 是非空串，因此它转换为 false。
                        // 运算返回 operand2 的值："dude"

false || 5+5           // Operand1 不能转换为 true，因此返回 operand2 的值 (10)

```

实际工作中，很少使用逻辑 OR 表达式返回的非布尔值。通常使用的是作出布尔决定的条件语句的结果。看一下例 5-3：

### 例 5-3：逻辑 OR 运算作为条件测试表达式

```

var x=10;
var y=15;
if (x || y) {
    trace("One of either x or y is not zero");
}

```

在第 3 行中，逻辑 OR 运算 ( $x \text{ || } y$ ) 用在需要布尔值来作为 if 语句测试表达式的地方。要确定  $x \text{ || } y$  的值，第一步就是将 10（第一个操作数 x 的值）转换为布尔值。如表 3-3 所示，任何非零的有限数字都将转换为 true。当第一个操作数转换为 true 之后，逻辑 OR 返回第一个操作数的值，也就是 10。因此，对于解释程序而言，if 语句就是这样的了：

```

if(10) {
    trace("One of either x or y is not zero");
}

```

但是 10 是一个数字，而不是布尔值。这样会如何呢？if 语句将逻辑 OR 运算的返回值转换为布尔值。在本例中，10 被转换为布尔值 true（还是按照表 3-3 的规则），解释程序会将代码变成这样：

```
if(true) {  
    trace('One of either x or y is not zero');  
}
```

这样就行了。测试表达式为 true，因此花括号中间的 *trace()* 语句将得到执行。

注意，如果逻辑 OR 运算的第一个操作数已经可以转换为 true，那么就没有必要再理会第二个操作数了，那样是没有什么作用的。因此，ActionScript 只当第一个操作数的值为 false 的时候才处理第二个操作数。除非第一个操作数为 false，你才会去处理第二个操作数，这种做法是非常有用的。在本例中，我们检查数字是否越界。如果数字太小，就没有必要去执行第二个测试来证明数字是否偏大：

```
if (xPos<0 || xPos>100) {  
    trace ('xPos is not between 0 and 100 inclusive.');
```

## 逻辑 AND

和逻辑 OR 一样，逻辑 AND 主要用来有条件地执行一个代码块，但是需要两个条件都得到满足才能执行。逻辑 AND 操作符的格式如下：

*operand1 && operand2*

*operand1* 和 *operand2* 都可以是任何有效的表达式。在最简单的情况下，两个操作数都是布尔表达式，逻辑 AND 当其中一个操作数为 false 的时候就返回 false，只在两个操作数都为 true 的时候才返回 true。例如：

```
true && false      // false, 因为第一个操作数为 true  
false && true       // false, 因为第二个操作数为 true  
true && true        // true, 因为两个操作数都为 true  
false && false       // false, 因为两个操作数都是 false
```

我们来看看逻辑 AND 操作符在真实的例子中是如何使用的。在例 5-4，只当两个变量都大于 50 的时候才执行 *trace()* 语句。

### 例 5-4：逻辑 AND 操作符作为条件测试表达式

```
x=100;  
y=51;  
if(x>50 && y>50) {  
    trace("Both x and y are greater than 50");  
}
```

因为表达式 `x>50` 和 `y>50` 都是 `true`, 因此执行 `trace()` 语句。

在例 5-5 中, 我们又回到了网站的例子, 访问限制在注册用户和特殊客人的范围内。但是这一次, 我们只在日期不是 1 月 1 日的时候才让用户进入。注意用 `Date` 对象来确定当前日期的用法。注意 `getMonth()` 返回 0 来表示 1 月。

### 例 5-5: 复合逻辑表达式

```
userStatus="registered";
now=new Date();                                // 创建一个新的 Date 对象
day=now.getDate();                            // 返回一个 1 到 31 之间的整数
month=now.getMonth();                          // 返回一个 0 到 11 之间的整数

// 如果语句因为保证易读性而可以跨行的话, 注意缩进格式和括号的使用……
if((userStatus=='registered') || (userStatus=="specialGuest"))
    && (month+day)>1) {
    // 让用户进入……
}
```

逻辑 AND 操作符和逻辑 OR 操作符的技术特点非常相似。首先, `operand1` 被转换为布尔类型。如果转化的结果为 `false`, `operand1` 的值就被返回。如果转化的结果为 `true`, 就返回 `operand2` 的值。

如果逻辑 AND 运算中的第一个操作数得出 `false`, 就没有必要再计算第二个操作数了。ActionScript 只当第一个操作数为 `true` 的时候才处理第二个操作数。这是很有效的, 你不需要处理第二个操作数, 除非第一个操作数为 `true`。在本例中, 只当除数为非零数字的时候才执行除法:

```
if (numItems != 0 && total / numItems>3) {
    trace ('You\'ve averaged more than 3 dollars per transaction');
}
```

## 逻辑 NOT

逻辑 NOT 操作符 (!) 返回和它的操作数相反的布尔值。它的格式如下:

`! operand`

`operand` 可以是任何合法的表达式。如果 `operand` 是 `true`, 逻辑 NOT 就返回 `false`。如果 `operand` 是 `false`, 逻辑 NOT 返回 `true`。如果 `operand` 不是布尔值, 它的值就会先转化为布尔值, 然后返回其反值。

和不等操作符 ( $\neq$ ) 一样，逻辑 NOT 操作符对于测试一样东西不是什么要比测试它是什么更方便。回忆一下例 5-5 中，我们要在日期不是 1 月 1 日的条件下执行一些动作。凭借第一反应，下面的写法似乎是正确的：

```
month != 0 && day != 1
```

但是我们发现，`month!=0` 在月份不为一月的任何时候都是 `true`（到目前为止一切顺利），但是`day!=1` 实际上在每个月的头一天都将是 `false`。因此，表达式`month!=0 && day!=1` 将在所有月份的头一天都拒绝用户进入，而不仅仅是 1 月 1 日。这与我们的原意是相违背的。

要检查今天是否是 1 月 1 日非常简单：

```
month==0 && day==1
```

表达式只当 1 月 1 日的时候才会产生 `true`。一旦确定了这一步，那么惟一需要的就是用 NOT 操作符来确定什么时候不是 1 月 1 日：

```
!(month==0 && day==1)
```

显然，代码的作用比用在例 5-5 中的 `month+day>1` 要明显。

NOT 操作符另外一个典型的用途是将一个变量的值从 `true` 变到 `false`，反之亦然。例如，假设有一个单独的按钮，用来控制声音的开关。你可以使用下面的代码：

```
soundState=!soundState           // 反转当前的声音状态
if(soundState) {
    // 如果声音被打开，确认声音能够听见
} else {
    // 如果声音被关上，就将音量设置为 0
}
```

注意，`!` 也可用在不等操作符 ( $\neq$ ) 中。作为一个编程记号（也就是符号），`!` 字符通常都表示非，或者是反面。它和数学中的阶乘符号并没有关系。

## 逻辑表达式和数据值

使用逻辑操作符来计算一个布尔结果相当于使用数学操作符来计算数字结果。例如，下面的四个运算和三个操作符是用来计算单一结果的：

```
userStatus=='registered' || userStatus=="specialGuest"
```

如果用户没有注册，但却是一个特殊客人，那么前面的表达式就可以简化为：

```
false || true
```

然后可以得出布尔结果 true。

下面是一个简化为单一数字的模拟数学运算：

```
(1+2) * (3+4)      // 四个值 (1, 2, 3, 4) 会变成……  
3 * 7              // 两个值 (3, 7), 会变成……  
21                 // 一个单一的值
```

记住，将复杂表达式简化为单一布尔值的方法可以用在任何需要单一布尔值的地方（比如条件语句的测试中）。

## 组合操作符

除了用在函数调用之外，括号()还可以用来组合一个代码短语，其优先级在各种操作符之上。在一些语句中也需要括号，特别是 if 语句。括号的格式如下：

```
(expression)
```

括号里的 expression 将得到计算，并且返回值。下面是一些例子：

```
if(x==y) {           // if 语句构成的需要  
    trace('x and y are equal'); // 函数调用的需要  
}  
(5+6) * 7          // 形成不标准的运算顺序  
(x>=100) || (y<=50) // 其实可以不用，但用了括号之后比较清楚  
x>=100 || y<=50    // 没有括号，看起来不那么明白
```

## 逗号操作符

虽然很少用，但是逗号操作符(,)让我们可以在只要一个表达式的地方对两个表达式进行计算。它的一般格式为：

```
operand1, operand2
```

两个操作数都必须是合法的表达式。当一个执行逗号运算的时候，先计算 *operand1*，再计算 *operand2*，然后返回 *operand2* 的值。实际上，逗号操作符的返回值通常被忽略。

逗号操作符主要用在 *for* 循环语句中，用来初始化多个变量，如例 5-6 中所示。注意，在例子的第一行中，表达式 *i=0* 和 *j=10* 都是逗号操作符的操作数。（这些表达式让 *i* 和 *j* 被初始化为 0 和 10。）表达式 *i++* 和 *j--* 都是第二个逗号操作符的操作数。（这些表达式在每轮循环中让 *i* 加 1，而 *j* 减 1。）逗号操作符用来压缩 *for* 循环语句语法范围内的多个表达式。

#### 例 5-6：在 *for* 循环中使用逗号操作符

```
for (var i=0, j=10; i!=j; i++, j--) {  
    trace("i: " +i+ ' j: '+j);  
}  
  
// 产生下面的输出……  
i: 0 j: 10  
i: 1 j: 9  
i: 2 j: 8  
i: 3 j: 7  
i: 4 j: 6
```

## 空(*void*)操作符

空操作符用来计算一个表达式但是不返回值。它的语法为：

```
void expression
```

*expression* 是用来计算的表达式。在 JavaScript 中，*void* 用来计算 JavaScript 超文本链接中的表达式，而不让结果出现在浏览器的地址栏里。在 ActionScript 很少这样用，它包括在对 ECMA-262 的兼容性能中。

## 其他操作符

本节中包括了本章主题下其他的操作符。我们只给出一个简要的参考，而在其他的章节中再给出详细的使用方法。

## 位逻辑操作符

如果你打算发展大规模的系统、存储器、计算速度和传输速率每一方面都会给性能造成很大的影响，那么你可以阅读第十五章中所述的位逻辑操作符。否则，可以使用布尔逻辑操作符，它执行和位逻辑操作符同样的任务，虽然优化程度要稍微差一些。

## typeof 操作符

*typeof* 操作符用来确定一个表达式的数据类型。它有一个操作数，如下所示：

```
typeof operand;
```

*operand* 可以是任何合法的表达式。*typeof* 运算的返回值是一个串，描述所计算的 *operand* 的数据类型，请参见第三章。

## new 操作符

*new* 操作符可用来创建一个新的复合对象——数组或者对象。对象可能是某个内置类中的成员，或者是用户自定义类的成员。*new* 的语法为：

```
new constructor
```

*constructor* 必须是一个定义新创建对象属性的函数。请参见第十一章，以及第十二章。

## delete 操作符

使用 *delete* 操作符可删除一个对象、一个对象属性、一个数组元素，或者从脚本中删除变量。*delete* 的语法为：

```
delete identifier
```

如果 *identifier* 不是一个数据容器（变量、属性或者元素），那么 *delete* 运算就失败，返回值 *false*；否则，它返回 *true*，表示成功。请参见第十一章。

## 数组元素 / 对象属性操作符

正如我们将在第十一、十二章中看到的那样，我们使用[]操作符获取和设置一个数组元素，或者一个对象属性的值。在访问数组的时候它的格式为：

```
array[element]
```

*array*是一个数组名称或者一个数组常量，*element*是一个表达式，从0开始，为非负的整数，描述要访问的数据元素的索引。

在访问一个对象的时候，它的格式为：

```
object[property]
```

*object*是一个对象名称或者对象直接量，*property*是任何表达式，是用来描述要访问的对象属性的串。

用在等号（=）左边的时候，元素或者属性被赋为表达式右边所给出的新值：

```
var colors=new Array();           // 创建一个新的数组
colors[0]='orange';              // 设置它的第一个元素
colors[1]='green';               // 设置它的第二个元素

var ball=new Object();            // 创建一个新的对象
var myProp='xVelocity';          // 在变量中存储一个串
ball['radius']=150;              // 设置 radius 属性
ball[myProp]=10;                 // 通过 myProp 设置 xVelocity 属性
```

使用在其他地方的时候，表达式返回指定元素或属性的值：

```
diameter=ball['radius'] * 2;    // 将 diameter 的值设置为 300
trace(colors[0]);                // 显示 "orange"
```

## 点操作符

点操作符是我们指向对象属性和嵌套影片剪辑的主要工具。从功能上说，点操作符和[]操作符的功能是一样的——它让我们设置和获取对象属性值。但是两种操作符在语法上有所不同，使它们各自独立。点操作符的一般语法为：

```
object.property
```

*object* 必须是对象的名称或者对象直接量, *property* 必须是表示 *object* 属性的一个标识符。注意, *property* 可能不是一个任意的表达式或者串直接量, 它必须是属性的名称。因为数组元素是编号的, 它们不被命名, 因此, 点操作符不能用来访问数组元素。

使用在赋值操作符的左边时, 点操作符用来给属性设置一个新的值:

```
var ball=new Object();
ball.radius=150;
ball.xVelocity=10;
```

使用在其他地方的时候, 点操作符返回命名对象属性的值:

```
diameter=ball.radius*2;
newX=ball.xPosition + ball.xVelocity;
```

请参见第十二章和第十三章。

## 条件操作符

条件操作符是一个便利的语法构造工具, 可以帮助我们简便地表现单一的条件语句。这个操作符有三个操作数, 格式为:

```
condition ? result_if_true : result_if_false;
```

执行一个条件运算的时候, 先计算第一个操作数 (*condition*)。如果 *condition* 为 *true* 或者可以被转换为 *true*, 那么就返回第二个操作数 (*result\_if\_true*) 的值。否则, 返回第三个操作数 (*result\_if\_false*) 的值。请参见第七章。

## 函数调用操作符

正如我们在 *trace()* 函数和串操作函数中所看到的那样, 可使用函数调用操作符 () 来调用一个函数。函数调用的一般格式为:

```
function_name(argument_list)
```

第一个操作数 *function\_name* 是某个函数的名称, 它必须是一个标识符, 而不能是一个表达式。函数必须存在, 否则解释程序就会产生错误。*argument\_list* 是传递

给函数的一系列参数，它们之间用逗号隔开，也可以不用参数。函数调用运算的返回值就是函数本身的结果值。

函数调用操作符可以调用任何内置的或者用户自定义的函数：

```
trace("an internal function");           // 内置函数，一个参数  
myClip.play();                          // 影片剪辑的方法，没有参数  
myRectangle.area(6,9);                  // 用户自定义方法，两个参数  
init();                                // 用户自定义函数，没有参数
```

## 小结

和所有的语言一样，ActionScript有自己的语法以及不同的词类。在前面几章中，你已经学习了句型结构、名词、形容词和关联词的相应内容。现在，我们要开始学习语句，它和动词一样，可以执行任务。语句让我们有足够的能力去实现类似句子的指令。

# 第六章

## 语句

我们在前面的章节中已经知道，ActionScript 可以存储和操作数据。在本章中，我们将学习如何用数据来做点什么。我们要按照语句（statement）或者代码短语的形式来对 Flash 发出指令，让解释程序完成某些任务。

要让 Flash 做点什么事情——不管是声音的停止、影片的播放、函数的运行，还是代码的循环——可使用语句。实际上，ActionScript 程序只能定义为一个语句的列表，它告诉解释程序我们想让 Flash 做什么。例如，下面是一个完整的 ActionScript 程序，它由四个语句构成，要让解释程序将一个网页装载到浏览器中：

```
var protocol = "http";           // 语句1  
var domain='www.moock.org';     // 语句2  
var path = "webdesign/flash/";   // 语句3  
getURL(protocol + "://" +domain+ "/" +path); // 语句4
```

用 ActionScript 为影片编写脚本其实就是为帧、影片剪辑和按钮添加语句的问题。本章揭示了语句的语法结构，并列出了一般的语句种类。我们在本章中要接触所有的 ActionScript 语句，但是，其中一些比较重要的语句将在以后的章节中详加描述。

## 语句的类型

从概念上说，共有五种核心的语句类型：

用来控制程序执行过程的语句

- loops (循环)
- conditionals (条件)
- ifFrameLoaded (如果帧已载入)

变量声明语句

- var
- set

函数的声明、调用、值返回语句

- function
- function call
- call
- return

对象处理语句

- with
- for...in

表示数据值的语句

- 任何表达式 (尤其是有副作用的表达式)

这些非正式的种类可以帮助我们了解使用语句究竟能让解释程序为我们做些什么。初看之下，这个列表内容似乎很有限，但是，应该看到循环和条件还有许多变体，通过函数调用也能做无数的事情。首先，我们来看看语句的形成。

## 语句语法

语句主要是由关键字和表达式构成的。我们已经看到过了 `var` 语句，它用来声明一个变量，同时也可以将它初始化。下面是一个 `var` 语句一般语法的例子：

```
var numFrames;
```

关键字 (本例中是 `var`) 标志着语句的开始，接着是句子的语法构造，在我们的例子中，是简单的变量名 `numFrames`，最后，分号标志着语句的结束。

---

注意：在ActionScript中，每一个语句都应该以分号结束。（使用分号是很好的形式，但并没有正式的规定。在第十四章中，我们会就分号的使用进一步展开讨论。）

---

一些语句可以有多种格式。比如，在使用`var`时可以选择要不要同时进行初始化工作（在本例中，`10`是一个简单的表达式，它的值被赋给了`x`）：

```
var x;          // 简单的声明  
var x=10;       // 带赋值的声明
```

我们会在本章后面的部分学习每一种语句的特殊语法。

## 语句块

一些语句实际上可以包括其他语句，或者子语句，来作为它们语法结构的一部分。例如，`if`语句的语法为：

```
if (expression) substatement;
```

`substatement`只有当`expression`可以转换为`true`的时候才能得到执行，它可以是如下所示的简单语句，像变量声明语句：

```
if(x==5) var numFrames=2;
```

或者可以很多语句组合在一起成为语句块：

```
if(x==5) {  
    var numFrames;  
    numFrames=10;  
    play();  
}
```

正如你所看到的，一个语句块可以是任何数量的语句，在花括号所包括的一行或者多行之内：

```
{ statement1; statement2; statement3... }
```

通过将语句块用作`if`语句的子语句，我们就可以在通常只要一个语句的地方列出很多语句。非常方便。

在 ActionScript 需要一个单独语句的地方，我们可以使用语句块。实际上，语句块有时候是必要的。例如，*function* 语句总是必须包括一个语句块，即使被声明的函数中只包含一个语句：

```
function aheadTwoFrames() {  
    gotoAndStop(_currentframe+2);  
}
```

为了使程序容易阅读，语句块的典型格式如下所示：

```
parent_syntax (  
    substatement1;  
    substatement2;  
    substatement3;  
)
```

*parent\_syntax* 表示我们要为其定义一个语句块的语句。

注意，前花括号出现在第一行的末尾，也就是在 *parent\_syntax* 之后。语句块的子语句出现时要缩进两个空格，每个单独的行上都应缩进。后花括号自己为一行，位置和开始的主语句相对应。语句块内的子语句应该以分号结束，但是花括号所在的行不用分号。

缩进格式只是一种习惯，而不是严格的语法要求。本书将支持 Flash ActionScript 编辑者所使用的风格。

## ActionScript 语句

既然你已经了解了语句的典型构造，浏览一下表 6-1 可以让你熟悉 ActionScript 语句的功能。

表 6-1 ActionScript 语句

语句	语法	使用
<code>break</code>	<code>break;</code>	取消一个循环
<code>call</code>	<code>call (frame);</code>	在远程帧上执行脚本
<code>continue</code>	<code>continue;</code>	重新开始当前的循环

表 6-1 ActionScript 语句 (续)

语句	语法	使用
do-while	do { <i>statements</i> } while ( <i>expression</i> );	<i>while</i> 循环的一个变体
empty statement	;	在需要语句的地方保持一个空位, 用于新手模式中的 evaluate
for	for ( <i>init</i> ; <i>test</i> ; <i>increment</i> ) { <i>statements</i> }	重复执行一些代码 (一个 <i>for</i> 循环)
for-in	for ( <i>property</i> in <i>object</i> ) { <i>statements</i> }	列举出对象的属性
function	function <i>name</i> ( <i>parameters</i> ) { <i>statements</i> }	声明一个函数
if-else if-else	if ( <i>expression</i> ) { <i>statements</i> } else if ( <i>expression</i> ) { <i>statements</i> } else { <i>statements</i> }	基于一个或者一系列的条件而执行一些代码
ifFrame-loaded	ifFrameLoaded ( <i>frame</i> ) { <i>statements</i> }	如果载入了特定的某帧就执行一些代码, 在 Flash 5 中不提倡
return	return; return <i>expression</i> ;	退出一个函数, 或者从函数返回一个值
set	set( <i>variable</i> , <i>value</i> );	将一个值赋给某个动态命名的变量
var	var <i>variableName</i> ; var <i>variableName</i> = <i>expression</i> ;	声明并可选择地为一个变量赋值
while	while( <i>expression</i> ) { <i>statements</i> }	重复地执行一些代码 (一个 <i>while</i> 循环)

表 6-1 ActionScript 语句（续）

语句	语法	使用
with	with(objectName) { statements }	在给定对象的语境里执行一些代码

## 循环和条件

我们已经同循环和条件有了好些次非正式的接触。这两种语句类型，占据了我们程序中复杂流程的一大部分。循环让我们可以重复地执行语句，条件让我们可以在某些特定的环境下执行语句。参见第七章以及第八章可以获得详尽的内容。

## 表达式语句

虽然任何表达式都可以独立地用做合法语句，但是，如果表达式不执行任何动作，我们也不能使用其结果，那么就没有意义了：

```
"hi there"; // 这没有多大用  
345+5; // 这个也一样
```

有一些表达式能通过对变量或者属性的赋值，或者通过物理地改变 Flash 环境而产生副作用，改变系统的环境。在下面的例子中，表达式改变了 x 的值：

```
x=345+5; // 有用得多
```

函数调用是表达式语句中功能最强大的。即使一个函数不返回有用的值，它也可以产生很有用的副作用。例如，*gotoAndStop()*返回值 *undefined*，但是它却有一个重要的副作用——它将播放头移动到了另外的帧：

```
gotoAndStop(5); // _currentframe 的值变成了 5
```

我们在第九章中会学到更多的关于函数调用的内容。

## var 语句

*var* 语句声明一个新的变量：

```
var x;
```

可以将对新变量的赋值作为 *var* 语句的一个部分：

```
var x = 10;
```

在函数外面使用的时候，*var* 语句创建的变量其活动范围在包含该语句的时间线之内。在函数内部使用的时候，*var* 语句创建的是该函数的局部变量（也就是函数结束后它就死亡）。参见第二章。

## set 语句 (set 变量)

对大部分函数赋值来说，我们在语句格式中使用一个赋值表达式，如下所示：

```
x = 30;
```

但是，这种表达式要求我们预先知道变量的名字。如果我们想在赋值的时候动态地产生变量的名字，可以使用 *set* 语句，它的语法如下：

```
set (variable, expression);
```

*variable* 是一个串表达式，在赋值中用作变量的名字，*expression* 是要赋给这个变量的新值。例如：

```
var x;
set ("x", 10);           // x 现在为 10

var firstName;
set ("first" + "Name", "jane"); // firstName 现在为 "jane"
```

下面所给出的例子更加巧妙，*y* 并不是在 *set* 语句中得到设置的。先获取 *y* 的值 ("x")，然后把这个值作为要设置的变量名称：

```
// 要特别注意这个例子……
var y="x";
var x;
set (y,15); // x 现在为 15, y 仍然是 "x"
```

在 Flash 4 中，*set* 被称为 *Set* 变量。在 ActionScript 的这个版本中，它通常用来动态地设置那些按连续的名字有计划地生成的变量。这可以让编程人员模仿数组，数组并不是 Flash 4 ActionScript 原有的部分。由于数组在 Flash 5 中被添加进来，*set* 就很少使用了。用 *set* 来模拟数组的更多信息，请参见第二章的内容。

## function 语句

正如 `var` 语句被用来声明变量（也就是创建）那样，`function` 语句是用来声明函数的。`function` 语句的语法如下：

```
function funcName (param1,param2,param3,...paramn) {  
    statements  
}
```

关键字 `function` 标志着语句的开始，`funcName` 是所声明函数的名称，`param1` 到 `paramn` 表示函数执行的时候所需要的参数，`statements` 是一个或多个语句构成的列表，当函数被调用的时候它们就会得到执行。

`function` 语句只创建一个供以后使用的函数，但不执行该函数。要执行一个函数，要在函数调用语句中使用函数的名称。

## 函数调用语句

一个函数调用语句执行一个内置或者用户自定义的函数，只要给出函数的名称，并且给出函数所需要的参数就可以了。调用和运行通常交替使用，意味着函数的名称是可以被执行的。函数调用语句的语法如下：

```
funcName (arg1, arg2, ... argn);
```

`funcName` 是要执行的函数名称，`arg1` 到 `argn` 是函数运行时所需要的参数的列表（也就是输入的值）。

函数调用语句功能非常强大，它是控制 Flash 影片最主要的手段。如果想要按照某种方式操作一个影片，通常可以调用函数。下面给出了几个简单的例子：

```
play();                      // 播放当前影片  
gotoAndStop(5);              // 将播放头送到第 5 帧  
startDrag("crosshair",true); // 让 "crosshair" 实例跟着鼠标箭头移动
```

函数调用也可以用来调用对象中的方法：

```
circle.area();  
today.getDate();
```

因为影片剪辑实例是对象，我们经常在脚本中使用方法的调用：

```
ball.play();
intro.gotoAndStop("end");
```

我们将在第九章中学习关于函数使用的更多内容，在第十二章中，学习函数是如何成为对象方法的。

## call()语句

简单地说，函数就是一系列可以重复使用的语句，它在程序运行期间的任何时候都可以得到执行。虽然 Flash 4 不支持真正的函数，但是，它能通过远程脚本动作提供一些类似函数的方便功能。在 Flash 4 中，我们可以在帧中添加一系列语句，并附加标签，从而创建类似函数的东西。创建的模拟函数以后可以通过 *call()* 语句来执行：

```
call (frame);
```

*call()* 语句执行由 *frame* 指定的帧上的脚本，*frame* 可以是任何帧标签或者帧号。如果指定的帧没有被载入，*call()* 函数就失败了。

显然，Flash 4 的模拟函数在真正的函数面前相形见绌。因此，我们在创建 Flash 5 及其以后版本的时候没有理由要去使用 *call()* 语句。但是，当创建 Flash 4 影片的时候，我们需要使用旧的子程序和 Flash 4 的 *call()* 语句。

## 返回 (*return*) 语句

我们调用一个函数的时候，可以有选择地传递一个或者多个值（也就是参数），以在执行的时候进行操作。函数同样可以发送一个返回值（函数执行所得到的结果值，它将被返回给调用者）。在函数体内，我们使用 *return* 语句来结束函数的执行，并且有选择地返回一个值。*return* 语句可以是下面的格式之一：

```
return;
return expression;
```

如果包含了可选择的 *expression*，它就是函数的返回值。没有返回值的 *return* 语句将结束函数，并返回 *undefined*。所有的 *return* 语句都将退出当前的函数。注意，*return* 语句在函数内不是必需的，一个没有 *return* 语句的函数会在函数体的最后一

个语句执行之后结束，并返回 `undefined`。参见第九章可以了解更多关于函数的创建、调用和结束的内容。

## with 语句

`with` 语句提供一个简化的方法来引用对象的属性，而无需再次重复地键入对象的名称。`with` 语句的一般格式为：

```
with (object) {  
    statements  
}
```

当一个属性在 `with` 语句块中出现的时候，要检查 `object` 是否确实包含这个属性。如果属性在 `object` 中存在，那么 `object` 的属性就用来形成该属性的引用。如果 `object` 中不存在这样一个属性，当前时间线或者函数就对该属性产生质疑。

下面的例子表明了在 `with` 语句内执行一个语句，以及在 `with` 语句外执行语句时二者的不同之处：

```
PI=10;                      // 设置一个时间线变量, PI  
with( Math ) {              // 在Math中执行语句  
    trace ('pi is: ' +PI);   // 显示: 3.14159... (PI是Math的属性)  
}  
trace ("PI is: " +PI);      // 显示: 10 (Math不再能访问到了)
```

除了可便捷地访问对象属性外，`with` 还可以调用对象的方法：

```
x=10;  
y=11;  
with (Math) {  
    larger = max(x,y);  
}  
trace(larger);    // 显示: 11
```

当某个对象作为 `with` 语句的目标时，不能在其中为该对象定义一个新的属性。注意，在前面的例子中，变量 `larger` 并不是在 `Math` 对象中定义的，因此，属性引用将影响包含该 `with` 语句的时间线或者函数。用下面的代码在 `myClip` 内设置变量是不恰当的：

```
with (myClip) {  
    var x = 10;           // x在当前时间线上进行设置, 而不是在myClip  
}
```

但是，我们可以按照其他方法，在影片剪辑实例中合理地使用 *with*。它可以提供一个方便的途径去处理深层嵌套的实例结构。例如，我们可以将下面的代码做一些改变：

```
_root.form.userProfile.userID='U346BX';
_root.form.userProfile.gotoAndPlay('questionnaire');
```

改变为：

```
with (_root.form.userProfile) {
    userID='U346BX'; // 返回一个已经存在于 userProfile 实例中的变量
    gotoAndPlay('questionnaire'); // 应用于 userProfile 实例中的函数
}
```

但是 *with* 并非是我们达到这个目的的惟一途径。我们也可以简单地将实例赋给一个变量，并在引用中使用该变量：

```
var userForm=_root.form.userProfile;
userForm.userID='U346BX';
userForm.gotoAndPlay('questionnaire');
```

许多开发者发现，变量的方法更容易理解和控制，但是两种方法都有效。有关将影片剪辑作为对象来处理的内容，可以参见第十三章。

## ifFrameLoaded 语句

Flash 5 的 *ifFrameLoaded* 语句代替了使用在更早版本中的 *If Frame Is Loaded* 动作。和 *if* 语句一样，*ifFrameLoaded* 有一个子语句，子语句只在特定的环境之下执行。下面是它的语法：

```
ifFrameLoaded (expression) {
    statements
}
```

*expression* 必须是帧号或者表示帧标签的串。如果用 *expression* 表示的帧已经装载到播放器中了，那么 *statements* 就可以得到执行。如果没有，语句块就被跳过。

*ifFrameLoaded* 语句不使用在脚本的预下载中，因为它缺少 *else* 子句。因此，它并没有得到提倡，只当创建 Flash 3 或更早版本的时候才使用。在 Flash 4 以及以后的

版本中，应该在 *if-else* 语句中使用 `_totalframes` 和 `_framesloaded` 属性，来创建一个更通用的预下载。例如：

```
if (_totalframes>0 && _framesloaded == _totalframes) {  
    gotoAndPlay('beginMovie');  
} else {  
    gotoAndPlay(_currentframe-1);  
}
```

## 空语句

为了完整性起见，我们可以说，用一个单独的分号来表示一个语句是合法的：

;

空语句在实际中并没有多少用途，不过它可以用来在语句需要的任何地方作为一个占位符。如果只是想在代码中加一个空行则没有必要。ActionScript 将忽略空行，空行只是用来提高程序的易读性。

在新手创建模式中，当 *evaluate* 动作添加到代码块中的时候，就会出现单独的分号。然后就可以将任意的语句输入到参数面板中的表达式域里。

## 语句和动作

如果你浏览过 Flash ActionScript 的编辑环境，你将不会发现任何涉及“语句”的词。甚至在 Macromedia 的 *Flash 5 ActionScript* 参考指南中，“动作”和“语句”也是可互换的。

将“动作”作为“语句”的同义词，混淆了好几种不同的 ActionScript 工具之间的差别。要知道为什么，你可以打开 ActionScript 编辑器，看看 Actions 文件夹，如图 1-2 所示。在文件夹中的 Actions 列表下面，你会发现我们在表 6-1 中所看见的语句。在散落的语句中，你会注意到好几个函数：`gotoAndPlay()`, `getURL()`, `startDrag()`, 等等。虽然被列为动作的函数可以在语句中使用，但是，它们在理论上并不是一个独特的语句类型——它们只是内置函数。语句、一些内置函数，以及事件处理器都被 Macromedia 称为动作。贯穿本书，我们不使用“动作”这种说法。反之，每

一个动作都用和它在语言中的正式角色相匹配的词语来描述：语句、函数，或者是事件处理器。

## 小结

虽然我们已经学习了很多关于语句的知识，但是，有两种重要的语句类型——条件和循环——还没有讲到。我们将在后面的两章中讨论这两种语句。

---

# 第七章

# 条件语句

迄今为止，我们所看到的大部分代码示例都是线性的——每一行语句按照顺序来执行，从第一行开始，到最后一行结束。线性代码总是做相同的事情。但是，条件语句则相反，它是一种只当指定条件得到满足的时候才执行某个动作的语句。在线性代码中，解释程序会先执行语句 A，然后执行语句 B，最后才执行语句 C。使用条件语句，我们可以让解释程序先执行语句 A，然后按照条件 X 来决定要执行语句 B 还是语句 C。

我们可以用条件语句来创建和控制有多个潜在结果的环境。例如，假设我们要创建一个密码保护站点。当用户要登录的时候，要么密码正确，用户可以进入站点，要么密码错误，用户看到一个错误消息。两种结果要求在我们的影片脚本中有各自不同的代码块。一个代码块需要将 Flash 播放头发送到一个包含欢迎页面的帧，而另外一个代码块要求将播放头发送到一个错误消息帧。但是当用户登录的时候，只应该有一个代码块可以得到执行。条件语句允许我们执行恰当的代码块，而跳过不合适的代码块。

解释程序如何知道要执行哪个代码块呢？当我们定义一个条件语句的时候，我们指定：如果条件得到满足，就执行第一个代码块；如果条件不满足，就执行另外一个（依此类推，这个块可能还有它自己的条件）。从本质上讲，我们在程序上要确定一个流程图一样的逻辑，模拟代码如下所示：

```
if (第一个条件得到满足) {
```

```
// 执行这里的代码  
} else if(第二个条件满足) {  
    // 执行这里的代码  
} ... 否则 {  
    // 执行这里的代码  
}
```

当然，我们必须按照解释程序懂得的方式来描述每一个条件。这并不难——无非是语法的问题，我们将在后面学习。从概念上说，所有的条件语句基于给定的条件，要么允许要么禁止代码块的执行。现在，我们来看看条件语句的实际工作情况。

## if 语句

*if* 语句是我们每天常见的，用途广泛的条件语句。使用 *if* 可以在代码中创建一个两叉的分支，就和岔路口一样。*if* 语句包含一个或多个语句，只当给定的条件得到满足的时候才执行这些语句。*if* 语句的语法为：

```
if (condition) {  
    substatements  
}
```

*if* 语句以关键字 *if* 开始，这是我们司空见惯的了。*condition* 必须在得到满足的情况下才能 *substatements*，*substatements* 是用花括号括起来的。*substatements* 是一个或者多个 ActionScript 语句。每一个子语句都应该占一行，以分号结束。整个 *if* 语句以后花括号 () 来结束。后面不必加分号。

*if* 语句中的 *condition* 可以是任何有效的表达式。当一个 *if* 语句得到执行的时候，解释程序会检查表达式（也叫作测试表达式）的值。如果它是 *true*，*substatements* 就执行；否则，*substatements* 就不被执行。下面，我们用简单的布尔值来作为测试表达式：

```
if (true) {  
    trace("The condition was met!"); // 这个语句将被执行  
}  
if (false) {  
    trace("The condition was met!"); // 这个语句不会被执行  
}
```

当然，实际上没有必要将布尔直接量用作测试表达式，因为它们的值是永远不会改

变的。我们要使用复杂表达式来返回布尔值。例如，包括一个复合运算的表达式如果可以返回一个布尔值，它就很适合用作条件语句的测试表达式：

```
var pointerX = _xmouse; // 鼠标的水平位置  
  
// 如果pointerX>300就得到true……  
if (pointerX > 300) {  
    // ...执行这里的语句  
    trace("The mouse is past the 300 pixel mark");  
}
```

现在是最酷的部分：条件语句的测试表达式不一定要求出布尔值——任何表达式都可以。我们可以使用串或者数字作为条件语句的测试表达式：

```
if ("hi") {  
    trace("The condition was met!");  
}  
if (4) {  
    trace("The condition was met!");  
}
```

如果表达式“hi”和4不是布尔值，该怎么办呢？答案就在表3-3所给出的数据类型转换之中。当条件语句的测试表达式不是布尔值的时候，解释程序会将表达式转换为布尔值。例如，解释程序将“hi”转换为false，因为所有的非数字串用在布尔语境中的时候都要转换为false。因此，条件没有得到满足，第一个*trace()*语句也就不会得到执行。类似的，解释程序将数字4转换为true（任何非零的数字都转换为true），因此，第二个*trace()*语句就可以得到执行了。

我们前边对数据类型转换内容的学习现在有了回报啦！下面是一些基础应用示例，你猜猜哪些子语句可以得到执行：

```
x=3;  
if (x) {  
    trace("x is not zero");  
}
```

这个例子用了OR操作符，该操作符在第五章中已经讨论过了：

```
lastName="";  
firstName="";  
if (firstName != " " || lastName != " ") {  
    trace("Welcome "+firstName+" "+lastName);  
}
```

最后，我们测试一个影片剪辑是否存在：

```
if(myClip) {  
    myClip._x=0; // 如果myClip存在，就将它放到场景的左边  
}
```

## else语句

用一个单独的*if*语句，可以让单独的一个代码块有选择地执行。如果添加一个*else*分支，可以选择两个代码块中的哪个可以执行。在语法上，*else*语句是*if*语句的扩展：

```
if (condition) {  
    substatements1  
} else {  
    substatements2  
}
```

*condition*可以是任何有效的表达式。如果*condition*为true，就可以执行*substatements1*；如果*condition*为false，就执行*substatements2*。换句话说，*else*语句适合用来描述一个互斥的决定：一个代码块可以执行，而另外一个不可以。

下面是一些代码示例：

```
var lastName="Grossman";  
var gender="male";  
if (gender=="male") {  
    trace('Good morning, Mr. '+lastName+'.');  
} else {  
    trace("Good morning, Ms. "+lastName+'.');  
}
```

*else*分支通常作为*if*语句的候补计划。回忆一下我们的密码保护站点的例子。如果密码是正确的，就让用户进入站点；否则，显示错误信息。我们可以用下面给出的代码来进行密码检查（假设*userName*和*password*是用户的登录输入，*validUser*和*correctPassword*是正确的登录值）：

```
if (userName==validUser && password == correctPassword) {  
    gotoAndPlay("intro");  
} else {  
    gotoAndStop("loginError");  
}
```

## 条件操作符

只有两个部分的条件语句可以用条件操作符 (`? :`) 方便地表达。条件操作符有三个操作数，可以将它们看作 `if-else` 语句系列的类似部分：

```
condition ? expression1 : expression2;
```

在一个条件操作符中，如果 `condition` 为 `true`，`expression1` 就被求值并返回结果；否则，对 `expression2` 进行求值并返回结果。条件操作符就如同是下面的代码的简化形式：

```
if (condition) {
    expression1
} else {
    expression2
}
```

和条件语句一样，条件操作符可以用来控制程序的流程：

```
// 如果命令为 'go'，就播放影片；否则，就停止
command == "go" ? play() : stop();
```

条件操作符也提供了一个为变量赋值的便捷途径：

```
var guess = "c";
var answer = "d";
var response = (guess == answer) ? "Right" : "Wrong";
```

它等价于：

```
var guess = "c";
var answer = "d";
if (guess == answer) {
    response = "Right";
} else {
    response = "Wrong";
}
```

## else if 语句

使用 `if` 和 `else`，可以选择性地执行一个或者两个代码块。而使用 `if` 和 `else if`，可以选择性地执行无限多个代码块。和 `else` 一样，`else if` 是 `if` 语句的扩展：

```

if (condition1) {
    substatements1
} else if (condition2) {
    substatements2
} else if (condition3) {
    substatements3
} else {
    substatements4           // 包括了以上条件全都未能满足的所有情况
}

```

*condition1*, *condition2*和*condition3*必须是有效的表达式。*substatements1*在*condition1*为true的时候可以得到执行。如果*condition1*为false,*substatements2*在*condition2*为true的情况下可以得到执行。否则，对*condition3*求值，依此类推，不管*else if*语句有多少。如果这些测试表达式没有一个是true，最后的*else*语句就被执行。例如，我们可以编写一个能够检查错误信息的登录检查程序，如下所示：

```

if(userName!=validUser) {
    message="User not found. Please try again.";
    gotoAndStop("loginError");
} else if(password!=correctPassword) {
    message="Password incorrect. Please try again.";
    gotoAndStop('loginError');
} else {
    gotoAndPlay('intro');
}

```

注意，一个*else if*语句只是一个*else*和一个嵌套*if*语句的结合。虽然下面的两段代码是一样的，但是第一段的易读性更好：

```

// 一般的"else if"语法
if(x>y) {
    trace("x is larger than y");
} else if (x<y) {
    trace("x is smaller than y");
} else {
    trace("x and y are equal");
}

// 扩展if/else链
if(x>y) {
    trace("x is larger than y");
} else {
    if (x<y) {
        trace("x is smaller than y");
    } else {
        trace("x and y are equal");
    }
}

```

```
    }
}
```

## 模拟 switch 语句

虽然 ActionScript 不支持 *switch* 语句（有的时候也叫作 *case* 语句），但是，这种复杂条件句的普遍形式还是值得仿效的。*switch* 语句让我们基于单一测试表达式的值执行一系列可能的代码块。例如，在下面的 JavaScript 的 *switch* 语句中，我们基于测试表达式 *gender* 的值给用户发出问候消息：

```
var surname='Porter';
var gender='male';

switch(gender) {
  case "femaleMarried" :
    alert('Hello Mrs. '+surname);
    break;
  case "femaleGeneric" :
    alert("Hello Ms. "+surname);
    break;
  case "male" :
    alert("Hello Mr. "+surname);
    break;
  default :
    alert("Hello "+surname);
}
```

在 JavaScript 例子中，*switch* 将 *gender* 的值同 *case* 表达式进行匹配比较：“femaleMarried”，“femaleGeneric” 或者 “male”。因为 *gender* 和表达式 “male” 相匹配，子语句 *alert("Hello Mr. "+surname);* 就得到执行。如果测试表达式没有和任何 *case* 表达式相匹配，那么默认的语句 —— *alert("Hello "+surname);* —— 就可以得到执行。

在 ActionScript 中，我们可以模拟一个 *switch* 语句，用一个 *if-else if-else* 语句链来实现，如下所示：

```
var surname='Porter';
var gender="male";

if(gender=="femaleMarried") {
  trace("Hello Mrs. "+surname);
} else if(gender=="femaleGeneric") {
  trace("Hello Ms. "+surname);
```

```

} else if(gender=="male") {
    trace("Hello Mr. "+surname);
} else {
    trace("Hello "+surname);
}

```

在更高级的方法中，可以将 *switch* 模仿为一系列的函数，这些函数存储在一般对象的属性中。例 7-1 显示了这个技术。要特别注意注释的内容，以便了解它是怎么工作的。也要注意条件操作符的使用，这我们已经在前面接触过了。

#### 例 7-1：一个模拟的 switch 语句

```

var surname='Porter';      // 用户的名字
var gender="male";         // 用户的性别（测试表达式）

// 创建一个对象，以便像我们模拟的 switch 语句那样运作
var mySwitch=new object();

// 将 "case expression" 属性赋给 mySwitch 对象。
// 每一个 "case expression" 属性都包括一个函数。
mySwitch.femaleMarried=function() {
    trace("Hello Mrs. "+surname);
};

mySwitch.femaleGeneric=function() {
    trace("Hello Ms. "+surname);
};

mySwitch.male=function() {
    trace("Hello Mr. "+surname);
};

mySwitch.default=function() {
    trace("Hello "+surname);
};

// 现在要基于 gender (在我们的例子中为 'male' ) 的值来执行适当的函数。
// 如果命名属性不存在，就执行默认函数。
mySwitch[gender] ? mySwitch[gender]() : mySwitch["default"]();

```

## 简化的条件语句语法

贯穿本书，我们在所有的条件语句中都使用语句块（花括号内的语句），即使一个块中只包含一个语句：

```

if (x==y) {
    trace("x and y are equal");
}

```

为了简洁，ActionScript 在条件语句只有一个子语句的时候不要求使用花括号。一个单独的子语句可以直接放在 *if* 或者 *else if* 语句之后，而不用花括号，这是合法的，如下所示：

```
if (x==y) trace ('x and y are equal');
```

或者像这样：

```
if (x==y)
    trace ("x and y are equal");
```

对一些程序员来说，这种形式易读性稍差，而且看起来似乎很容易出错，但是，它的确节省了许多源代码。

## 小结

条件语句是 ActionScript 中极其重要的部分。它们可以准确地控制代码的执行。在下一章中，我们要探讨另一种非常重要的执行控制语句，循环语句。

---

# 第八章

# 循环语句

在上一章的学习中，我们知道，条件语句中的语句块仅当其测试表达式的值为 `true` 时才会得到执行。而一个循环语句，换句话说，只要其测试表达式的值保持为 `true`，就可以让一个语句块得到重复执行。

循环包括多种精炼的类型：`while`, `do-while`, `for`, 以及 `for-in`。前三种类型有着非常相似的作用，只不过语法不同罢了。最后一种循环类型 `for-in` 是专门适用于对象的循环类型。我们将从 `while` 语句开始循环的讨论，这个循环类型最容易理解。

## while 循环

从结构上来看，`while` 语句的构造和 `if` 语句颇为相似：一个主语句中套入一个语句块，这个语句块只在给定的条件为 `true` 时才会得到执行：

```
while(condition) {  
    substatements  
}
```

如果条件为 `true`, `substatements` 就会被执行。但与 `if` 语句不同的是，当最后一个子语句结束的时候，将从 `while` 语句的开头重新执行（这是因为解释程序“循环”到了 `while` 语句的开头）。第二遍执行 `while` 语句的过程和第一遍类似：对条件表达式

求值，如果它仍然为 true，就执行 *substatements*。这个过程会继续下去，直到 condition 变为 false，然后，就会执行脚本中跟在 while 语句之后的任何语句。

下面是一个非常简单的循环示例：

```
var i=3;
while (i<5){
    trace ("x is less than 5");
}
```

这个例子贴切地表现出了 while 循环的正确语法，但它却好像有一点错误。要知道为什么，让我们跟随解释程序来看看这个例子的语句执行过程。

我们从 while 语句之前的语句开始，*var i=3*，这个语句将变量 *i* 的值设置为 3。因为变量 *i* 被用在循环的测试表达式中，所以这一步通常被叫做循环初始化 (initialization)。然后，我们通过处理测试表达式而开始执行 while 语句：*i<5*。因为 *i* 的值为 3，而 3 明显小于 5，那么测试表达式的值就为 true，因此，将执行循环中的 *trace()* 语句。

这些动作完成之后，就会开始新一轮循环。再一次检查测试表达式的值。变量 *i* 的值并没有改变，因此测试表达式依然为 true，同样应该执行 *trace()* 语句。执行完循环体，又重新开始新的循环。猜猜会发生什么？变量 *i* 的值还是没有改变，因此测试表达式还是为 true，必须再一次执行 *trace()* 语句，一次又一次，永远没有尽头。因为测试表达式总是返回 true，就没有办法退出循环——永远陷在了一个无限的循环里，不能执行跟在 while 语句后面的任何其他语句。在 ActionScript 中，无限循环将引起错误，正如我们后面将会看到的。

上述的循环变成了无限循环，是因为它缺少更新语句 (*update statement*) 来改变用在测试表达式中的变量值。更新语句会让测试表达式最终变成 false，这样就可以终止循环。我们现在来添加一个更新语句改正无限循环：

```
var i=3;
while (i<5) {
    trace ("x is less than 5");
    i++;
}
```

更新语句 *i++* 出现在循环体的结束部分。当解释程序通过我们的循环，它会像以前那样执行 *trace()* 语句，不过它还要执行语句 *i++*，即把变量 *i* 加 1。随着循环的每

一次重复，*i* 的值就会增加。在第二次重复之后，*i* 的值变成了 5，因此测试表达式 *i*<5 就将变成 false。这样，循环就安全地结束了。

循环更新语句执行一个基本的循环动作：它起着计数的作用。变量 *i*（可以叫做计数器）贯穿一个可预知的数字序列——对于有秩序的任务来说可以起到非常完美的作用，比如复制影片剪辑或者访问数组元素。下面，我们不用循环的方法来复制 5 次 square 影片剪辑：

```
// 按顺序为每个新剪辑命名，并放在各自的层上
duplicateMovieClip ("square", "square1", 1);
duplicateMovieClip ("square", "square2", 2);
duplicateMovieClip ("square", "square3", 3);
duplicateMovieClip ("square", "square4", 4);
duplicateMovieClip ("square", "square5", 5);
```

然后，我们试着用循环功能来完成相同的工作：

```
var i = 1;
while (i <= 5) {
    duplicateMovieClip ("square", "square" + i, i);
    i++;
}
```

如果我们要做的是复制 100 次 square，你就会对两种做法之间的差别有很深的印象了。

循环对于数字操作非常有效，尤其是那些存储在数组中的数据。例 8-1 中，将利用循环功能将数组中的所有元素显示在输出窗口中。要注意的是，第一个元素是数字 0，而不是 1。

#### 例 8-1：利用 while 循环显示数组元素

```
var people = ["John", "Joyce", "Margaret", "Michael"]; // 创建一个数组
var i = 0;
while (i < people.length) {
    trace ("people element " + i + " is " + people[i]);
    i++;
}
```

输出窗口中得到的结果是：

```
people element 0 is John
people element 1 is Joyce
people element 2 is Margaret
```

```
people element 3 is Michael
```

注意，变量 *i* 不但使用在测试表达式中，而且还作为数组索引数字使用，这是非常典型的。现在，我们再次用 *i* 作为 *charAt()* 函数中的自变量：

```
var city = 'Toronto';
trace ("The letters in the variable 'city' are: ");
var i = 0;
while (i<city.length){
    trace (city.charAt (i) );
    i++;
}
```

输出窗口的显示为：

```
The letters in the variable 'city' are:
t
o
r
o
n
t
o
```

最后，我们不切割数据，而是使用一个循环，从存储在数组中的一系列词语中构造一个句子：

```
var words = ["Toronto", 'is', 'not', 'the', 'capital', 'of', "canada"];
var sentence;
var i = 0;
while (i<words.length) {
    sentence += words[i]; // 将当前词添加到句子。
    // 如果它不是最后一个词……
    if (i<words.length - 1) {
        sentence += " "; // ……添加空格。
    } else {
        sentence += ". "; // ……否则，以一个句号结束。
    }
    i++;
}
trace (sentence); // 显示: "toronto is not the capital of Canada."
```

几乎所有的循环都包括一些计数器（有时也叫循环因子或者索引变量）。计数器让我们按顺序进行数据循环。如果我们用想要操作的数组或串的 *length* 属性来确定计数器的最大值限制时，会显得特别方便，正如我们在前边的例子中所做的—样。

也有可能创建一个其结束点不依靠计数器的循环。只要循环的测试表达式最后变成 `false`, 循环就会结束。比如, 我们在下面的例子中通过检查 Flash Player 的堆栈来确定第一个空白单元:

```
var i = 0;
while (typeof eval("_level" + i) == "movieclip") {
    i++;
}
trace ("The first vacant level is " + i);

// 现在装载一个电影到堆栈, 它是独立的
loadMovie ("myMovie.swf", i);
```

## 循环术语

在前边的小节里, 我们遇到了几个新名词。现在来正式地看一下这些词, 这样, 处理循环时就能更好地理解它们。

### 初始化 (*Initialization*)

这种语句或者表达式的作用是定义一个或多个变量, 通常用在循环的测试表达式中。

### 测试表达式 (*Test expression*)

这是个必须具备的条件, 目的是让循环体内的语句能被执行。它通常被叫做条件或者测试, 有时候也叫做控制。

### 更新 (*Update*)

这个语句用来在下一次测试之前修改用在测试表达式中的变量。典型的更新语句将增加或者减少循环的计数变量。

### 重复 (*Iteration*)

这是指测试表达式和循环体语句的一次完整执行。有时候也叫做一次循环或者一个周期。

### 循环嵌套 (*Nesting or nested loop*)

指一个循环中包含另外的循环, 这样就能扫描一些二维的数据类型。比如, 可以扫描表格每列中的每一行。外层的或者说顶层的循环将逐次通过各列, 而内层的循环将扫描每列中的每一行。

### 循环因子或索引变量 (*Iterator or index variable*)

这种变量的值会随着循环的每次重复而增加或者减少，通常用来计数或者顺序扫描一些数据。循环因子通常被称为计数器。循环因子通常命名为 i, j 和 k 之类，有的时候也使用 x, y 和 z。在一系列的嵌套循环中，i 通常是顶层循环的循环因子，而 j 是第一个嵌套循环的循环因子，k 又是第二个嵌套循环的循环因子，依此类推。你可以使用任何你所喜欢的变量名，以方便操作。例如，你可以使用 charNum 作为变量名，来让自己记住它表示的是串中的当前字符。

### 循环体 (*Loop body*)

当循环的条件得到满足而执行的语句块就叫做循环体。有的循环体可能根本就不会被执行，也有的可能被执行几千次。

### 循环头或者循环控制 (*Loop header or loop control*)

这是循环的一部分，包含循环语句的关键字 (*while*, *for*, *do-while*, 或者 *for-in*) 以及循环控制。循环控制随着循环的类型而变化，控制部分包含初始化、测试，以及更新。在 *while* 循环中，控制只包含测试表达式。

### 无限循环 (*Infinite loop*)

一个循环永远重复是因为它的测试表达式根本不可能变成 *false*。无限循环在 ActionScript 中会导致错误，这个情况我们会在后面讨论。

## do-while 循环

正如我们先前说到的，只要指定的条件保持为 *true*, *while* 循环允许解释程序重复执行一个代码块。基于 *while* 循环的结构，如果循环条件在第一次测试的时候没有得到满足，那么它的循环体就会完全被跳过。*do-while* 语句则可以确保循环体至少被执行一次。*do-while* 循环的循环体总是能在第一次循环的时候得到执行。*do-while* 语句的语法很像颠倒过来的 *while* 语句：

```
do {  
    substatements  
} while (condition);
```

以关键字 *do* 开始，后面跟着循环体 *substatements*。当循环第一次通过 *do-while* 语句的时候，*substatements* 在 *condition* 尚未被测试之前就得到了执行。在 *substatements* 块结束以后，如果 *condition* 值为 *true*，那么循环就可以重新开

始，而 *substatements* 又可以得到执行。循环反复执行，直到 *condition* 的值变为 *false*，这时候，*do-while* 语句就结束了。注意，在包含 *condition* 的括号后面应该加分号。

显然，如果我们想要一次或多次地执行某个任务，那么使用 *do-while* 循环方式是很方便的。在例 8-2 中，我们要从一个名为 *starParent* 的剪辑中复制一系列星星闪烁的影片剪辑，并把它们随机放置在场景中。即使 *numStars* 被设置为 0，我们要创作的星河也至少应该有一颗星星。

#### 例 8-2：使用 *do-while* 循环

```
var numStars = 5;
var i = 1;
do {
    // 复制 starParent 剪辑
    duplicateMovieClip(starParent, "star" + i, i)

    // 将复制的剪辑随机放置在场景中
    _root["star"+i]._x = Math.floor(Math.random() * 551);
    _root["star"+i]._y = Math.floor(Math.random() * 401);
} while (i++ < numStars);
```

你是否注意到，我们已经偷偷地在测试表达式中修改了变量 *i* 的值。记得在第五章中我们就讲过，后增量操作符不仅返回其操作数的值，而且还将操作数加 1。在处理循环时，增量操作符非常方便（也经常用到）。

## for 循环

*for* 循环实际上和 *while* 循环相同，但它的语法更为复杂一些。值得注意的是，循环头除了测试表达式之外，还可以同时包含初始化和更新语句。

下面给出的是 *for* 循环的语法：

```
for (initialization; condition; update) {
    substatements
}
```

*for* 循环将循环的关键部分整齐地放在了循环头处，互相之间用分号隔开。在 *for* 循环的第一次循环之前，*initialization* 语句得到执行（仅一次）。它专门用来设置

循环变量的初始值。和其他的循环类型一样，如果 *condition* 的值为 *true*，那么 *substatements* 就可以执行；否则，循环结束。在每一次循环的最后，*update* 语句都要执行，然后才重新测试 *condition*，以确定循环是否应该继续。这里是一个典型的 *for* 循环，它简单地从 1 计数到 10：

```
for (var i = 1; i <= 10; i++) {  
    trace ("Now serving number " + i);  
}
```

如果你用 *while* 循环语句来构造与此相同的功能，就比较容易理解 *for* 循环是如何工作的了。

```
var i = 1;  
while (i <= 10) {  
    trace ("Now serving number " + i);  
    i++;  
}
```

一旦习惯了 *for* 语法，就会发现它比较节省空间，并且更容易检查循环体和循环控制部分。

## for 循环中的多循环因子

如果我们想要在一个循环中控制更多的因素，完全可以用一个以上的循环变量。*while* 循环如果有多个循环因子，看起来就会像这样：

```
var i = 1;  
var j = 10;  
while (i <= 10){  
    trace("Going up " + i);  
    trace("Going down " + j);  
    i++;  
    j--;  
}
```

在 *for* 语句中使用逗号操作符也可以达到同样的结果：

```
for (var i = 1, j = 10; i <= 10; i++, j--) {  
    trace("Going up " + i);  
    trace("Going down " + j);  
}
```

## for-in 循环

*for-in* 语句是专门用于罗列对象属性的循环方式。新手可能想马上跳过这一部分，在读了第十二章之后再回过头来看。

*for-in* 并非重复一系列语句直到给定的测试条件变成 `false`，它对于指定对象的每个属性只循环一次。因此，*for-in* 语句并不需要明确的更新语句，因为循环重复次数是由所处理对象的属性数目所决定的。*for-in* 的语法如下所示：

```
for (var thisProp in object) {  
    substatements; // 语句按照某种方式明显地使用 thisProp  
}
```

*substatements* 对于 *object* 的每一个属性只执行一次，*object* 是任何有效对象的名称，*thisProp* 是任意的变量名或者标识符。在每一次循环过程中，*thisProp* 变量用来临时保存一个串，这个串是当前所列举对象属性的名称。串值可以在每一次循环期间被用来访问和操作当前的属性。*for-in* 循环最简单的例子是一个列举对象所有属性的脚本。在这里，我们创建了一个对象，然后用 *for-in* 循环逐条列举它的属性：

```
var ball = new Object();  
ball.radius = 12;  
ball.color = "red";  
ball.style = "beach";  
  
for (var prop in ball) {  
    trace("ball has the property " + prop);  
}
```

因为 *prop* 以串的形式存储了 *ball* 的属性名称，我们就可以用 *prop* 加 `[]` 算子来获得这些属性的值，就像这样：

```
for (var prop in ball) {  
    trace("ball." + prop + " is " + ball [prop]);  
}
```

用 *for-in* 循环来获得属性值其实提供了一个很好的检查时间线上当前的影片剪辑的方式。例 3-1 就是一个用 *for-in* 作影片剪辑探测器的范例。

注意，*for-in* 中所检查的对象属性并不是按照可预知的顺序来列举的。同时，*for-in* 语句并非总是能列举出对象的所有属性。如果对象是用户定义的，那么所有的属性

都可以被罗列，包括任何继承属性。但是，内置对象的一些属性就会被*for-in*语句跳过。例如，内置对象的方法就不会被*for-in*列出来。如果想用*for-in*语句来操作内置对象的属性，那么可以首先建立一个测试循环，以确定对象中可访问的属性。

---

**警告：***for-in*循环不会列出没有默认值的输入文本域。因此，表单确认代码如果发现了空文本域就可能出错，除非这些文本域被所在的时间线明确规定为一般变量。请参见第十八章。

---

在获取表单的时候，*for-in*语句也能用来选取数组中的元素。

```
for (var thisElem in array) {  
    substatements; // 语句按照某种方式明确地使用 thisElem  
}
```

下面的例子列出了数组的元素：

```
var myArr = [123, 234, 345, 456];  
for (var elem in myArr) {  
    trace(myArr[elem]);  
}
```

## 提前终止循环

在一个简单的循环中，测试表达式是确定何时终止循环的惟一因素。当一个简单循环的测试表达式变成了 `false`，循环就结束了。但是，随着循环变得越来越复杂，我们可能需要人为地终止一个正在运行的循环过程，而不管测试表达式的值如何。要达到这个目的，可使用 `break`（中断）和 `continue`（继续）语句。

### **break** 语句

`break` 语句将终止当前的循环。它的语法最为简单：

```
break
```

惟一的要求是，`break` 必须在循环体内出现。

对于那些不再需要完成的过程，`break` 语句给出了一个中断的方法。例如，我们使用

一个 *for-in* 循环来建立一个表单检查路线，它将扫描一条时间线上的输入文本变量。如果发现了一个空白输入域，就警告用户，她没有恰当地填写表单。我们可以通过执行一个 *break* 语句来终止这个进程。例 8-3 给出了相应的代码。注意，本例假定存在一个名为 *form* 的影片剪辑，它包含了一系列声明过的输入变量，名为 *input01*, *input02*, 等等。

#### 例 8-3：一个简单的表单域确认

```
for (var prop in form) {
    // 如果这是我们的“输入”文本域的属性之一
    if (prop.indexOf ("input") != -1) {
        // 如果表单条目为空，终止
        if (form [prop] == "") {
            displayMessage = "Please complete the entire form.";
            break;
        }
        // 当 break 执行的时候，任何跟在它后面命令都不能执行
    }
    // 不管是因为执行了 break 命令还是测试表达式变成了 false
    // 在循环结束后都执行此处的语句
}
```

可以使用 *break* 语句来中断一个将趋于无限的循环。也就是说，可以执行代码块的前半部分、而不必执行 *if (condition) break;* 语句后面的部分。例 8-4 给出了常用的方法。

#### 例 8-4：终止无限循环

```
while (true) {
    // 初始语句部分
    if (condition) break;
    // 后面的语句
}
```

## continue 语句

*continue* 语句在改变当前的循环过程方面，和 *break* 语句很相似，不过与 *break* 不同的是，它用来重新开始下一轮循环。*continue* 语句的语法也很简单：

```
continue
```

在所有的循环类型中，*continue* 语句会打断当前循环体的循环，不过，它重新开始新循环的方式随着循环语句类型的不同而略有变化。在 *while* 循环和 *do-while* 循环

中，测试表达式在循环重新开始之前被检查。但是在 *for* 循环中，循环更新工作在检查测试表达式之前执行。在 *for-in* 循环中，下一次循环以所处理对象的下一个属性开始（如果存在的话）。

使用 *continue* 语句，可以让循环体的执行在特定的情况下变成可选的。例如，在下面的例子中，我们要移动所有对场景左边缘不透明的影片剪辑实例，并且要将循环体跳过透明的实例。

```
for (var prop in _root) {
    if (typeof _root [prop] == "movieclip") {
        if (_root [prop]._alpha < 100) {
            continue;
        }
        _root [prop].x = 0;
    }
}
```

## 最大循环数

前面我们已经提到过，ActionScript 中的循环不允许永远执行。在 Flash 5 Player 中，循环被限制在 15 秒以内。在这个时间内，循环所能达到的次数取决于循环所包含的内容和计算机的速度。为了安全起见，你创建的循环甚至不能要求多执行短短几秒（这对于处理期限来说就等于无穷！）。大部分循环只用几毫秒的时间就结束了。如果一个循环要花更长的时间来完成（例如，因为它在初始化拼字游戏的时候要处理数百个串），还不如用一个时间线循环来重写代码，我们在下一节中会讨论这个问题。时间线循环可以修改脚本在屏幕上执行的进程，并且能避免如图 8-1 所示的显示错误消息的潜在可能。

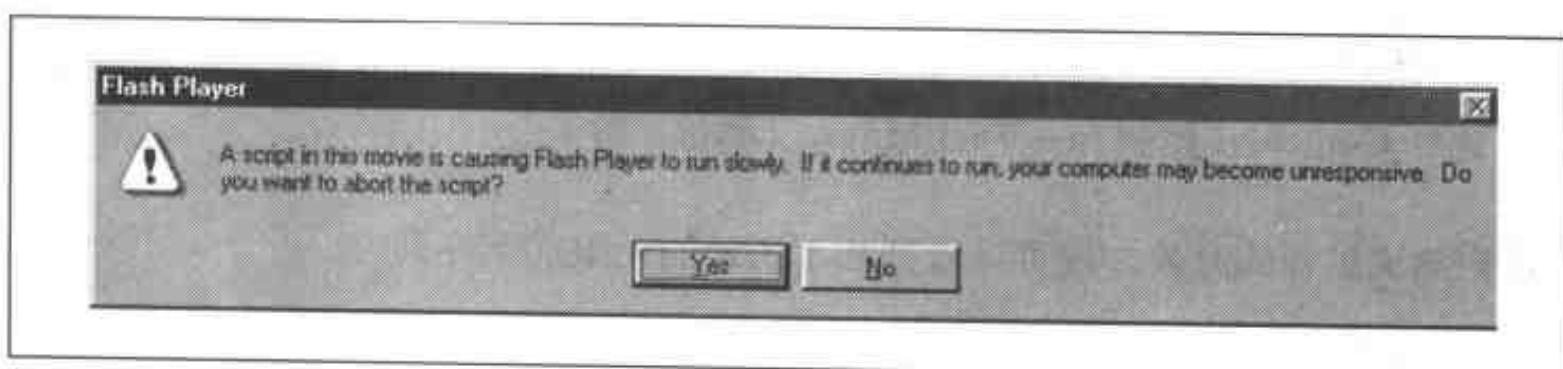


图 8-1 错误循环！停机！

当一个循环在 Flash 5 Player 中运行超过了 15 秒，将会给出一个提示框警告用户，电影脚本阻碍了影片的播放。用户可以选择等待脚本结束，或者退出脚本。

Flash 4 Player甚至更严格——它只允许200 000次循环——否则的话，所有脚本将在没有任何警告的情况下全部失效。

---

**警告：**特别注意：用户所看到的15秒限制警告中，并没有提到取消一个失控的脚本实际上会让影片里的所有脚本都停止运行！如果用户选择“是”来停止一个循环的继续运行，那么影片里所有的脚本都将失效。

---

## 时间线和剪辑事件循环

迄今为止我们所看到的所有循环都会让解释程序重复地执行代码块。大部分循环都将会是这种“ActionScript语句”类型。但有的时候，它也会通过循环Flash的播放头而创建一个时间线(Timeline)循环。要实现这个效果，就要在任意给定的帧中附加一系列语句，在它的下一帧中添加一个`gotoAndPlay()`函数，其目标指向上一帧。当影片播放的时候，播放头就会在这两个帧之间循环，使头一帧中的代码重复执行。

我们可以通过下面的步骤来制作一个简单的时间线循环：

1. 建立一个新的Flash影片。
2. 在第1帧中，添加下面的语句：

```
trace("Hi there! Welcome to frame 1");
```

3. 在第2帧中，添加下面的语句：

```
trace("This is frame 2");
gotoAndPlay(1);
```

4. 选择Control(控制)→Test Movie(测试影片)。

当我们测试影片的时候，就可以看到如下所示的无限文字序列：

```
Hi there! Welcome to frame 1
This is frame 2
Hi there! Welcome to frame 1
This is frame 2
```

时间线循环可以具有普通循环所不能具有的两个功能：

- 它们可以无限执行同一个代码块，而不会引起任何错误。
- 它们可以执行需要在循环中进行场景更新的代码块。

第二个时间线循环的功能需要稍作说明。当执行任一帧的脚本时，影片场景在视觉上要等到脚本结束才能够更新。这意味着传统的循环语句不能用来执行重复的视频和音频任务，因为任务需要的结果不能在循环期间得到实现。例如，要重新配置一个影片剪辑就需要更新场景，因此不能用普通的循环语句按计划运行影片剪辑。

你可能会用下面的代码让电影剪辑 ball 水平地滑过场景：

```
for (var i = 0; i < 50; i++) {  
    ball._x += 10;  
}
```

从概念上说，循环语句似乎可以办到——它以很小的距离不断更新 ball 的位置，仿佛的确可以造成移动的效果。但是，实际上，每次 ball 的 \_x 位置改变的时候 ball 都不会移动，因为场景并没有得到更新。反之，我们所能看到的是，在脚本结束的时候 ball 突然向右跳跃了 500 个像素——50 次循环，每次 10 像素。

为了让场景在 ball.\_x += 10; 语句的每一次执行之后得到更新，可以使用一个时间线循环，如下所示：

```
// 第一帧的代码  
ball._x += 10;  
  
// 第二帧的代码  
gotoAndPlay (1);
```

因为 Flash 在两帧之间更新场景，ball 就会动起来。但是时间线循环完全独占了它所在的时间线。当它运行的时候，不能够播放时间线上任何其他内容。一个更好的办法是将时间线循环放到一个空的，只有两帧的影片剪辑中。这样既可以在循环之间使场景更新，又不会冻结需要来做其他动画的时间线。

## 创建空白剪辑时间线循环

下面的步骤给出了创建空白剪辑时间线循环的方法：

1. 建立一个新的 Flash 影片。

2. 创建一个名为 *ball* 的影片剪辑符号，它包含一个圆形。
3. 在主场景上，将层 *Layer1* 重新命名为 *ball*。
4. 在 *ball* 层上，放置一个 *ball* 符号的实例。
5. 将 *ball* 剪辑的实例命名为 *ball1*。
6. 选择 Insert (插入) → New Symbol (新符号) 命令，创建一个空白的影片剪辑符号。
7. 将剪辑符号命名为 *process*。
8. 在 *process* 剪辑的第 1 帧中，添加下面的代码：

```
_root.ball._x += 10;
```
9. 在 *process* 剪辑的第 2 帧中，添加下面的代码：

```
gotoAndPlay (1);
```
10. 返回主影片时间线，创建一个名为 *scripts* 的层。
11. 在 *scripts* 层中，放置一个 *process* 符号的实例。
12. 将实例命名为 *processMoveBall*。
13. 选择 Control (控制) → Test Movie (测试影片)。

*processMoveBall* 实例现在将移动 *ball1*，而不妨碍 *ball1* 所在主时间线的回放。

注意，第 12 步并不是非要不可的，但它可以方便对循环的控制。给你的时间线循环实例命名，就能通过开始和停止实例的回放而停止和开始循环，如下所示：

```
processMoveBall.play();
processMoveBall.stop();
```

注意，这个例子中，只要循环有可能运行，那么，*processMoveBall* 和 *ball1* 就都必须存在于主时间线上。如果想让代码更简练，可以在 *process* 中对 *ball1* 剪辑使用一个相对引用：

```
_parent.ball._x += 10;
```

如果想要从任何时间线上控制 *ball1*，可以对 *ball1* 使用绝对引用：

```
_root.ball._x += 10;
```

---

**警告：**时间线循环并不能在一个单一的帧中进行。也就是说，如果我们将 `gotoAndPlay(5)` 函数放到影片的第 5 帧上，这个函数将被忽略。播放器知道播放头已经在第 5 帧了，因此什么也不做。

---

可以在在线代码库（Code Depot）中找到时间线循环和空白剪辑循环的范例，*.fla* 文件。

## Flash 5 剪辑事件循环

时间线循环很有效，但并不是最好的。在 Flash 5 中，可以用影片剪辑上的事件处理器来达到和时间线循环同样的目的，而且适应性更强（你只要跟随这里的范例讲解，或者去看看第十章，就可以了解关于影片剪辑事件处理器的更多细节内容）。

一个进入帧（`enterFrame`）事件处理器被放置到影片剪辑中时，影片中每过一帧，它就会执行一次代码块。我们可以用单帧空白剪辑上的进入帧事件处理器来重复执行代码块，同时允许场景在每次重复中得到更新（就像事件线循环那样）。整个步骤如下所示：

1. 执行上节中的 1~7 步。
2. 在主场景上，创建一个名叫 *scripts* 的新层。
3. 在 *scripts* 层上，放置一个 *process* 剪辑的实例。
4. 选择 *process* 实例，添加如下代码：

```
onClipEvent(enterFrame) {  
    _root.ball._x += 10;  
}
```

5. 选择 Control（控制）→ Test Movie（测试影片）命令。

*ball* 实例将会在场景中移动。

剪辑事件循环可以让我们避免将代码添加在影片剪辑中的麻烦，并且不像时间线循环那样需要两帧来循环。所有的剪辑事件循环动作都发生在一个单独的事件处理器中。但是，我们刚才看到的剪辑事件实例有一个潜在的缺点：没办法有计划地开始循环，或者循环一旦开始就没有办法按计划停止。唯一终止循环的方法就是用空白关键帧从时间线上物理地删除 *process* 实例。

为了创建能够在任意时间开始和结束的事件循环，我们必须创建一个空白的剪辑，它包含了负责事件循环的另一个空白剪辑。这样，不管我们什么时候想开始或者结束循环，都可以动态地添加或者删除整个部分。这会让人觉得有点费解，不过它的效果很不错。我们再次按照下面给出的步骤来做：

1. 执行“创建空白剪辑时间线循环”中的1~5步。
2. 两次选择 Insert（插入）→ New Symbol（新符号）命令，创建两个空白影片剪辑符号。
3. 将一个剪辑符号命名为 process，另一个为 eventLoop。
4. 在 Library（库）中，选择 process 剪辑，然后选择 Options（选项）→ Linkage（链接）命令，就会出现符号链接属性对话框。
5. 选择 Export（导出）这个符号。
6. 在标识符域中，输入 `processMoveBall`，然后点击确认按钮。
7. 在 process 剪辑的第1帧中，将 eventLoop 实例拖到场景上。
8. 选择 eventLoop 实例，然后添加下面的代码：

```
onClipEvent(enterFrame) {  
    _parent._parent.ball._x += 10;  
}
```

9. 回到主影片时间线，对第1帧添加如下的代码：

```
attachMovie("processMoveBall", "processMoveBall", 5000);
```

10. 不管想什么时候停止循环，都可以使用下面的语句：

```
_root.processMoveBall.removeMovieClip();
```

11. 选择 Control（控制）→ Test Movie（测试影片）命令。

ball 实例将再次从场景上移过，但是这一次，我们只要使用第9步和第10步所示的 `attachMovie()` 和 `removeMovieClip()` 函数，就可以随心所欲地开始或者终止这个过程。

在线代码库中还有一些正规的，可操作的剪辑时间循环范例。

## 保持事件循环的灵活性

我们刚才所见的两种剪辑事件循环都有一行用来在场景上更新ball 实例位置的代码。比如：

```
onClipEvent(enterFrame) {  
    _parent._parent.ball._x += 10; // 更新ball 的位置  
}
```

虽然这个方法有效，但是显得不太精炼。因为对剪辑事件有针对性地添加代码，就分散了代码库，也打散了影片中的逻辑和动作。为了保持代码在创建期间可以从事件循环内访问，并且能构造得更好以供再次使用，我们只有调用函数。所以，在这个例子中，我们不是真的移动ball 剪辑，而是应该调用一个函数来移动ball 剪辑，如下所示：

```
onClipEvent(enterFrame) {  
    _parent._parent.moveBall();  
}
```

用户自定义函数moveBall ()将被定义在我们添加processMoveBall 剪辑的同一条时间线上，如下所示：

```
function moveBall() {  
    ball._x += 10;  
}
```

我们将在第九章讨论更多关于函数和代码可移植性的内容。

如果应用程序比较简单，你可以不用空白事件循环剪辑。在某种情况下，我们完全可以将事件循环合理地直接添加到所处理的剪辑中。在ball 例子中，可以通过将如下所示的代码直接添加到ball 实例中，从而避免使用单独空白剪辑：

```
onClipEvent(enterFrame) {  
    _x += 10;  
}
```

这个方法确实很方便，但它并不简单。和我们的第一个例子一样，它无法开始和终止循环。

## 时间线和剪辑事件循环上的帧速率影响

因为时间线和剪辑事件循环每帧重复一次，所以它们的执行频率和影片的帧速率有关。如果用时间线或者事件循环的方法将某个对象围绕屏幕移动，那么帧速率的增加将意味着动画速度也会相应增加。

当我们在前边所说的例子中建立 ball 剪辑的动作时，已经暗示球的速度和帧速率相关。代码表示为“当每一帧过去的时候，ball 将向右移动 10 个像素”：

```
_ball += 10;
```

因此，ball 的速度依赖于帧速率。如果影片以每秒 12 帧的速度播放，那么 ball 剪辑每秒将移动 120 像素。如果影片每秒播放 30 帧，ball 剪辑每秒将移动 300 个像素！

为脚本动画定时的时候，要按照影片的帧速率来计算每次移动某个对象的距离。因此，如果一个影片每秒播放 20 帧，而我们希望对象每秒移动 100 像素，那么我们就应该将对象的速度设置为每帧 5 像素（5 像素 × 20 帧每秒 = 100 像素每秒）。这种方法有两个重大的缺陷：

- 依靠帧速率来确定对象的移动速度，若再想修改帧速率就比较困难了。如果改变了帧速率，就必须重新计算速度，并且相应地编辑代码。
- Flash 播放器并不一定按照 Flash 制作工具所规定的帧速率来播放，它通常播放得更慢。如果播放影片的计算机不能以足够快的速度渲染帧，以跟上所指定的帧速率，那么影片就被减慢了。这种减速的程度甚至会因载入系统的不同而有所不同。如果有其他程序正在运行，或者如果 Flash 正在执行一些处理器密集任务，那么帧速率可能会降低一段时间，然后又恢复正常速度。

可以用如下网址中的定时追踪工具来测试这种情况：

<http://www.moock.org/webdesign/flash/actionscript/fps-speedometer>

在有些情况下，一个播放速度稍有不同的动画是可以被接受的。但是，当视频精确性的问题非常重要，或者涉及到动作游戏的响应时，按照运行时间而非帧速率来计算对象的移动距离会更恰当一些。例 8-5 给出了一个以时间为基础的简单动画举例（也就是说，ball 的速度不依赖帧速率）。新的影片有三个帧，两个层，一个层包含 ball 实例，另外一个包含脚本。

例 8-5：以时间而非帧速率为基础，计算移动距离

```
// 第1帧中的代码
var distancePerSecond = 50;           // 每秒移动的像素
var now = getTime();                  // 当前时间
var then = 0;                         // 最后一帧得到渲染的时间
var elapsed;                          // 帧渲染之间的间隔微秒数
var numSeconds;                      // 已运行的时间
var moveAmount;                      // 每帧移动的距离

// 第2帧中的代码
then = now;
now = getTimer();
elapsed = now - then;
numSeconds = elapsed / 1000;
moveAmount = distancePerSecond * numSeconds;
ball._x += moveAmount;

// 第3帧中的代码
gotoAndPlay(2);
```

注意，如果帧速率突然改变，那么基于时间的移动可能会显得有些不平稳。我们可以用一个新的运行时间度量法来改善这种情况，新的方法是将多帧之间的时间进行平均处理，而非两帧之间。

## 小结

噢，我们好像已经走得很远了。本章的末尾将是一个里程碑——它结束了对ActionScript语句的研究。这意味着我们已经有了变量、数据、数据类型、表达式、算子，以及语句。这些语言成分是所有脚本的基础。如果到现在为止你已经理解了所有的东西，或者至少理解了大部分，那么就可以正式宣布你能“说”ActionScript了。

在第一部分“ActionScript基础”的其余部分中，我们会努力使会话更有意义，命令更强大。我们将进入一些更高级的话题，即如何使代码更灵活，如何创建驱动代码执行的事件，如何管理复杂的数据，以及如何有计划地操作影片剪辑。这些技术将会帮助我们建立更高级的应用实例。

---

# 第九章

## 函数

要讨论函数的内容真让我感到头晕，因为它们是 ActionScript 中功能如此强大的一个部分。一个函数也就是一个代码块，它在程序中可以重复使用。函数不仅给我们的脚本带来了巨大的方便和灵活性，而且还帮助我们控制 Flash 影片元素。如果没有函数，很难想像编程工作会是什么样子——它们让所有的事情变得简单，从单词排序到计算两个影片剪辑之间的距离。在本章中我们将介绍函数，然后在第十二章中学习如何利用函数和对象创建复杂的、功能强大的程序。

我们首先把焦点集中在程序函数上——也就是我们在脚本中自己创建的函数。通过学习创建我们自己的函数，就可以熟悉下面这些基本原则：

### 函数声明

创建函数，以便在脚本中使用。

### 函数调用

让函数得到执行。换句话说，就是让函数中的代码运行起来。

### 函数的变量和参数

向函数提供的数据，在调用的时候进行操作。

### 函数终止

结束函数的执行，可能返回一个结果。

### 函数的作用域

确定函数的有效范围和生命周期，以及函数体内出现的变量的可访问性。

我们只要理解了函数的这些概念，就可以考虑它们是如何应用于内部函数的，内部函数也就是建立在 ActionScript 中的内置函数。现在我们就开始吧！

## 函数的创建

要创建一个基本的函数，只需要一个函数名称和一个用来执行的语句块，如下所示：

```
function funcName() {  
    statements  
}
```

`function`关键字标志着新函数声明的开始。后面跟着的是函数的名称 `funcName`，以后要用它来调用函数。`funcName` 必须是一个合法的标识符（注 1）。然后，加上一对括号 `()`，其中可以包括参数，我们将在后面讨论这个问题。如果函数没有参数，括号空着也可以。最后，我们要提供一个函数体（也就是语句块），它所包含的代码将在调用函数的时候得到执行。

我们现在来建立一个非常简单的函数：

1. 建立一个新的 Flash 影片。
2. 在影片主时间线的第 1 帧，添加下面的代码：

```
function sayHi() {  
    trace("Hi there!");  
}
```

这非常简单——我们已经创建了一个名为 `sayHi()` 的函数。运行这个函数的时候，函数体里的 `trace()` 语句就会执行。不要关闭你已经创建的影片，我们稍后还要运行函数。

---

注 1：参见第十四章可以得到合法标识符的控制规则。

## 函数的运行

运行一个函数，就像巫师念动一段魔咒一样，只要给出函数的名称就行了，后面跟着第五章中已经介绍过的函数调用操作符`()`。

```
funcName()
```

(你几乎已经嗅到了神秘的曼德拉草根悄悄燃烧时散发到空气中的微弱气味。)

括号可以包括函数所定义的任何参数。如果函数没有定义参数，括号就空着。现在我们来试着调用`sayHi()`函数（它没有参数）以掌握基本的函数调用方法。

下面再次给出`sayHi()`函数的声明：

```
function sayHi() {  
    trace("Hi there!");  
}
```

下面所给出的就是对`sayHi()`函数的调用：

```
sayHi();
```

当这行代码执行的时候，`sayHi()`就被调用，因此，它的函数体语句就会运行，最终导致“Hi there!”出现在输出屏幕上。

可以看到，输入`sayHi();`单行语句比输入整个的`trace()`语句要方便，函数名`sayHi`对代码的作用有着更直观的描述。使用有意义的函数名称可以让代码有更好的易读性，就像人类的语句一样。

在继续下面的讨论之前，要提醒你注意的是函数调用结尾的分号：

```
sayHi();
```

我们在后面加上分号是因为函数调用是一个完整的语句，正规的格式都应该在语句后面加一个分号。

不管你信不信，我们其实只是学了函数创建和调用的基础知识而已。但也不是太寒酸。函数可让代码集中起来，便于维护，特别是当需要在程序中重复地执行相同操作的时候。函数带上参数之后功能会更为强大，我们在后面会讨论这个问题。

## 向函数传递参数

在前面的小节中，我们创建了一个函数，它执行简单的 *trace()* 语句——并不是函数中最典型的例子。下面的这个函数要有趣一些，它将影片剪辑实例 ball 移动一小段距离：

```
function moveBall() {  
    ball._x += 10;  
    ball._y += 10;  
}
```

函数 *moveBall()* 得到声明之后，我们就可以在任意时间调用 *moveBall()* 函数，将 ball 沿对角线移动：

```
moveBall();
```

小球沿着对角线往右下方向移动。[注意初始位置(0, 0)是在主场景的左上方。增加 *\_x* 的值就可以将球往右移动，但是，和笛卡儿坐标不一样的是，增加 *\_y* 的值会让小球往下移动，而不是往上。]

我们的 *moveBall()* 函数非常方便，但是它缺少灵活性。它只对影片剪辑 (ball) 起作用，只能将 ball 向一个方向移动，而且总是移动同样的距离。

一个设计良好的函数应该定义一个单一的代码段，但是能在多种环境中应用。我们可以推广 *moveBall()* 函数，这样，它就可以将任何剪辑按照任何方向移动任何距离。推广任何函数的第一步是确定何种因素在控制它的行为。在 *moveBall()* 函数中，关键因素就是影片剪辑的名称，要移动的水平和垂直距离。这种因素就叫做函数的参数——它们是让你能在调用函数时修改的信息。

## 创建带参数的函数

回忆一下简单函数声明的一般格式：

```
function funcName() {  
    statements  
}
```

要添加参数，也就是能用在函数内部的变量，我们提供一个合法的标识符，放在函数声明的括号中间。参数之间由逗号分隔，如下所示：

```
function funcName(param1,param2,param3,...paramn) {  
    statements  
}
```

一旦定义了参数，就可以从函数体内部访问函数的参数，就像使用其他参数一样。例如：

```
function say(msg) {           // 定义 msg 参数  
    trace("The message is "+msg); // 在 trace() 语句内部使用 msg  
}
```

我们的函数声明定义了参数 `msg`。`trace()` 语句就能像使用其他任何变量一样使用这个参数，并将它的值输出到输出窗口中。

在 `moveBall()` 函数中使用参数，就可以设置剪辑的名称，水平距离和垂直距离，这些值在每次运行函数的时候都可以不同。例 9-1 给出了相应的代码。

#### 例 9-1：一个普通的 moveClip 函数

```
function moveClip (theClip, xDist, yDist) {  
    theClip._x -= xDist;  
    theClip._y += yDist;  
}
```

我们将 `moveBall()` 函数重新命名为更具有普遍性的 `moveClip()`，并定义了三个参数：`theClip`（要移动的影片剪辑）、`xDist`（要移动的水平距离）和 `yDist`（要移动的垂直距离）。在函数体内，我们使用这些参数，而不使用原来例子中给出的值，因此，我们的新函数就可以按照任意的水平距离和垂直距离重新定位任何影片剪辑。

## 调用带参数的函数

创建一个函数的时候，要定义参数的名称，它在本质上是占位符；调用函数的时候，我们要为每一个参数提供对应的值。

变元或者参数这种说法在本书中是可以交替使用的，在大部分文档中也是如此。从理论上说，变元就是调用函数时所使用的值，而参数是函数中“接受”变元的占位符。

回忆一下不带参数的函数调用的一般格式：

```
funcName()
```

要提供（或者传递）变元给函数，在调用函数的时候在括号内提供一系列的值就可以了：

```
funcName(arg1, arg2, arg3,...argn)
```

用作变元的值可以是ActionScript中任何有效的表达式，包括复合表达式。例如，我们在前边定义了一个简单的函数，*say()*，它有一个单独的参数，*msg*：

```
function say(msg) {  
    trace("The message is " + msg);  
}
```

要调用*say()*，我们使用下面的语句：

```
say('This is my first argument...how touching');
```

或者像这样：

```
say(99);
```

注意，“This is my first argument...how touching” 和 99 两个值属于不同的数据类型。ActionScript 允许我们将任何类型的数据传递给函数，只要函数知道如何处理传递来的值就可以了。（诸如 C 这样的语言要求我们预先定义每一个参数的数据类型，如果提供的变元类型不匹配，就会出错。）

每一个参数在传递之前，其值会得到充分的计算。因此，我们可以在调用函数的时候传递复杂表达式。例如：

```
var name = "Gula";  
say("Welcome to my web site " + name);
```

因为表达式 “Welcome to my web site”+*name* 在传递给函数 *say()* 之前就求出值来了，因此函数接受到的是值 “Welcome to my web site Gula”。这意味着我们可以有计划地生成函数变元。这很有用。

要传递一个以上的参数给函数，可用逗号来分隔参数。回忆一下我们在例 9-1 中给出的 *moveClip()* 函数，它接受三个参数：*theClip*, *xDist*, *yDist*。因此，调用 *moveClip()* 会像这样：

```
moveClip(ball, 5, -15);
```

三个变量中的每一个都被赋给函数声明中所命名的参数的对应值：ball 赋给 theClip, 5 赋给 xDist, 而 -15 赋给 yDist。我们可以通过在调用通用 *moveClip()* 函数时使用不同的值，从而将任何剪辑移动任何距离。下面，我们将实例 square 向右移动 2 个像素，向下移动 100 个像素：

```
moveClip(square, 2, 100);
```

## 退出函数并返回值

函数通常会在解释程序执行完函数体内最后一个语句之后自然结束。如果中间使用了强制命令，就可以在执行最后的语句之前结束函数的执行。另外，一个函数可以给调用它的代码返回一个结果（发送回一个计算值）。我们来看看其中的具体内容。

### 函数的终止

*return* 语句在第六章中已经介绍过了，它可以终止一个函数，并且有选择地返回一个结果。当解释程序在函数的执行期间遇到 *return* 语句的时候，它就跳过函数中剩余的所有其他语句。看看下面的例子：

```
function say(msg) {
    return;
    trace(msg); // 这一行代码并不会得到执行
}
```

前面的例子并不现实，因为 *return* 语句在 *trace(msg)* 执行之前就终止了函数。因此，*return* 语句通常应是函数体中的最后一个语句，除非它用在中间的一个条件语句中。在下面给出的例子里，如果密码不正确，我们就用 *return* 来退出函数：

```
var correctPass="cactus";
function enterSite(pass) {
    if(pass != correctPass) {
        // 如果密码错误就退出
        return;
    }
    // 这些代码只当密码正确的時候才可以执行
    gotoAndPlay('intro');
}

enterSite("backAttack"); // 函数将过早退出
enterSite("cactus"); // 函数将自然结束
```

顾名思义，*return* 语句让解释程序返回到调用函数的地方。如果没有 *return* 语句，ActionScript 所执行的动作会像在函数体的末尾包含了一个 *return* 语句一样：

```
function say(msg) {  
    trace(msg);  
    return;           // 这行代码完全是可有可无的  
}
```

不管 *return* 语句是隐式的还是显式的，也不管函数什么时候结束，程序的执行都会从函数调用语句的后面继续进行。例如：

```
say('Something');           // 这行代码执行 say() 函数中的语句  
// 程序的执行在 say() 函数结束之后从这里继续进行  
trace('Something else');
```

## 从函数返回值

正如我们所看到的，*return* 总是用来结束一个函数。但是，它也可以将某个值发送回到调用函数的脚本，其格式如下：

```
return expression;
```

*expression* 的值就是函数调用的结果。例如：

```
// 定义一个函数，将两个数字相加  
function combine(a,b) {  
    return a + b;      // 返回两个变元的和  
}  
  
// 调用函数  
var total=combine(2,1); // 将 total 设置为 3
```

用 *return* 语句返回的表达式或者结果被称为函数的返回值。

注意，*combine()* 函数只计算并返回两个数字的和（它也可以将两个串连接起来）。它不像 *sayHi()* 函数（显示一个消息）或者 *moveClip()* 函数（将影片剪辑重新定位）那样执行某个动作。我们可以将其赋给一个变量，从而使用函数的返回值：

```
var total = combine(5,6);           // 将 total 设置为 11  
var greet = combine("Hello ","Cheryl") // greet 是 "Hello Cheryl"
```

函数调用的结果只是一个普通的表达式。因此，它也可以用在附加的表达式中。下面的例子将 `phrase` 设置为 “11 people were at the party”：

```
var phrase=combine(5,6)+" people were at the party";
```

通常可以将函数的返回值用作复杂表达式的一个部分——甚至用作其他函数调用中的变元。例如：

```
var a=3;
var b=4;
function sqr(x) { // 将一个数字求方
    return x * x;
}
var hypotenuse = Math.sqrt(sqr(a) + sqr(b));
```

注意这个例子是如何将 `sqr()` 函数的返回值传递给 `Math.sqrt()` 函数的！前面的例子也可以写成这样的一行代码：

```
var phrase = combine(combine(5,6), " people were at the party");
```

在上面的例子中，表达式 `combine(5,6)` 将得出 11，成为外面的 `combine()` 函数调用的一个变元，在这个函数中，它将同串 “people were at the party” 连接起来。

如果一个 `return` 语句不包括返回表达式，或者根本没有 `return` 语句，那么函数将返回值 `undefined`。实际上，这很容易引起错误。例如，下面的代码并不会完成任何有意义的动作，因为 `return` 语句被漏写了：

```
function combine(a, b) {
    var result = a + b; // 结果计算出来了，但是没有被返回
}
```

下面的代码同样也是不正确的：

```
function combine(a,b) {
    var result = a + b;
    return; // 你忘记指定返回值了
}
```

要创建一个需要有返回值的函数时，不要忘记写上一个 `return` 语句，用它来返回一个需要的值。否则，返回值就是 `undefined`，后来基于这个值的计算也很可能是错误的。

## 函数直接量

ActionScript 允许我们创建函数直接量，这在我们需要一个临时函数，或者要在某个需要表达式的地方使用函数的时候会非常方便。

函数直接量和标准函数声明有相同的语法，除了函数名被省略，并且在语句块后面有一个分号之外。一般的格式为：

```
function (param1, param2, ... paramn) { statements };
```

*param1, param2, ... paramn* 是一个选择性的参数列表，*statements* 是构成函数体的一行或者多行语句。因为它不包括函数名称，因此，如果不把函数直接量存储到变量（或者一个数组元素或对象属性）中，它就会被弄丢。我们可以将函数直接量存储在变量中，以供以后的访问，如下所示：

```
// 将函数直接量存储在变量中
var mouseCoords=function() { return [_xmouse,_ymouse];};

// 现在我们执行函数
mouseCoords();
```

注意，因为 ActionScript 不支持 JavaScript 的 *function()* 构造器，因此在 ActionScript 中不能像在 JavaScript 中那样在运行期间动态生成函数。

## 函数的可用性和生命周期

和变量一样，程序函数不会永远存在，也不会在整个影片中都可以直接访问到。为了准确地调用函数，我们需要知道如何从不同的影片区域来访问它们，也需要确定它们在被调用之前是存在的。

### 函数的可用性

程序函数（在代码中自己创建的函数）只可直接访问下面范围内的调用：

- 添加到影片剪辑主时间线的代码，它包含着函数的定义。
- 包含函数定义的影片剪辑时间线上的按钮。

直接访问意思是函数可以简单地通过名称来调用，而不用指向影片剪辑或者对象的引用，如下所示：

```
myFunction();
```

可以用点语法从影片内的任何位置间接访问函数（也就是远程访问），正如间接地指向一个变量一样，必须包括含有该函数的影片剪辑的路径，如下所示：

```
myClip.myOtherClip.myFunction();
_parent.myFunction();
_root.myFunction();
```

因此，假设一个名为rectangle的剪辑实例位于主时间线上，它包含一个名为area()的函数。我们可以用指向rectangle的绝对路径从影片中的任何地方调用area()，如下所示：

```
_root.rectangle.area();
```

要从一个远程影片剪辑时间线上引用到一个函数，可使用和引用远程变量同样的语法，这在第二章中已经讨论过了。（我们在第十三章中还将学习关于函数远程调用的更多内容。）

不是所有的函数都属于影片剪辑时间线。一些程序函数可能属于用户自定义的内置对象。当函数属于某个对象的时候，函数只能由它的主对象来访问。我们将在第十二章中学习关于对象方法（也就是属于对象的函数）的可用性和生命周期的更多内容。

## 函数的生命周期

一个定义在影片剪辑时间线上的函数，当剪辑从场景上被删除的时候就丢失了。要调用一个属于已经不存在的剪辑的函数显然是不行的。在主影片时间线上定义一个函数是确保函数永久存在的最好方法，因为主时间线在影片中是永远存在的。

注意，帧脚本中的函数在播放头首次进入包含该脚本的帧中时，将得到初始化，并且变成可访问的函数。因此，我们可以在函数被声明之前调用该函数，只要两者都在同一个脚本里。例如：

```
// 在 tellTime() 函数的声明语句之前调用该函数
tellTime();
```

```
// 告别 tellTime() 函数
function tellTime() {
    var now=new Date();
    trace(now);
}
```

## 函数的作用域

ActionScript语句有特定的作用域，也就是它们作为有效函数能够影响到的区域。当一个语句被加到影片剪辑上的时候，语句的作用域就被限制在包含它的剪辑中。例如，要在在一个赋值语句中使用变量 score：

```
score=10;
```

如果这个语句属于 clipA，那么解释程序将会在 clipA 内设置 score 的值，因为语句的活动范围属于 clipA。如果这个语句属于 clipB，那么解释程序会在 clipB 中设置 score 的值，因为语句的作用域在 clipB 中。语句的位置决定了它的作用域，也就是它的影响范围。

函数体里的语句在其内部单独的范围内进行操作，叫做“局部作用域”。函数的局部作用域如同为函数准备的私人电话间，与函数所属的剪辑或者对象的作用域清楚地分开。函数的局部作用域在函数被调用时得到创建，在函数执行结束时被破坏。当函数体内使用了变量的时候，解释程序首先查看函数的作用域。

例如，函数参数就是在函数的局部作用域中定义的——而不是在包含该函数的时间线的作用域。因此，参数只当函数运行的时候才能被函数体内的语句访问到。函数之外的语句不能访问函数的参数。

函数的局部作用域为临时变量提供了一个函数内的单独使用空间。这就消除了函数变量和时间线变量之间潜在的名称冲突，全面减少了存储器的使用。

## 作用域链

即使函数在它自己的局部作用域中进行操作，函数体内的语句仍可以访问普通的时间线变量。函数的局部作用域是解释程序查找变量引用的第一个地方，但是，如果

在局部作用域中没有发现变量引用，搜索就会延伸到包含该函数的影片剪辑或者对象。

例如，假设我们定义了一个变量 `firstName`，它在影片剪辑 `clipA` 中。我们还在 `clipA` 中定义了一个函数 `getName()`。在 `getName()` 里，我们在 `trace()` 语句中使用了变量 `firstName`：

```
firstName="Christine";  
  
function getName() {  
    trace(firstName);  
}  
  
getName();
```

当我们调用 `getName()` 的时候，解释程序必须找到 `firstName` 的值。在 `getName()` 的局部作用域中并没有名为 `firstName` 的变量，因此，解释程序到 `clipA` 时间线上查找 `firstName`。在那里，解释程序找到了 `firstName`，并显示它的值“Christine”。

`trace()` 语句能够从 `getName()` 函数内得到时间线变量，是因为 `getName()` 本身没有定义一个叫做 `firstName` 的参数或者变量。现在，如果我们把一个名为 `firstName` 的参数添加到 `getName()` 函数中，将会如何呢：

```
firstName="Christine";  
  
function getName (firstName) {  
    trace(firstName);  
}  
  
getName ("Kathy");
```

这一次，当调用 `getName()` 的时候，我们将“Kathy”的值赋给了参数 `firstName`。当解释程序执行 `trace()` 语句的时候，它搜索局部作用域，发现了 `firstName` 参数和它的值“Kathy”。因此，函数的输出现在就变成了“Kathy”而不是“Christine”。即使存在时间线变量 `firstName`，函数的局域变量 `firstName` 也会被优先使用。

我们的例子显示了作用域链的操作——解释程序用来引用变量和对象属性的对象层次。对属于时间线的函数来说，作用域链只有两个层次：局部作用域和包含该函数的影片剪辑的作用域。但是当我们把函数添加到自定义的对象和类中时，作用域链就会包含许多的层次，我们会在第十二章中看到相关的内容。

如果要从函数体内访问的变量和属性的作用域不同于语句的作用域，就必须使用点语法来构成精确的引用。例如：

```
function dynamicGoto() {  
    // 故意走出函数的局部作用域  
    _root.myClip.gotoAndPlay(_root.myClip.targetFrame);  
}
```

注意，函数的作用域链是按照函数的声明语句来决定的，而不是按照函数的任何调用语句。也就是说，函数体内代码的作用域在包含该函数声明的影片剪辑中，而在包含函数调用语句的影片剪辑中。

下面的例子给出的是作用域链的错误用法：

```
// 在主影片时间线上输入代码  
// 函数的作用域链包含了主影片  
function rotate(degrees) {  
    _rotation += degrees;  
}
```

如果要用 `_root.rotate` 来旋转 `clipA`，它会旋转整个的主影片，而不只是 `clipA`：

```
// 在clipA的时间线上输入代码  
_root.rotate(30);          // 哟！整个影片都旋转了！
```

在这种情况下，可以将要旋转的剪辑作为参数传递给 `rotate()` 函数，以解决这个问题：

```
function rotate(theClip, degrees) {  
    theClip._rotation += degrees;  
}  
  
// 调用rotate()函数，并使用clipA  
_root.rotate(clipA, 30);
```

## 局部变量

在函数局部作用域中赋值的变量称为局部变量。局部变量（包括参数）只能被它们所在的函数体内的语句访问，并且只当函数运行的时候才存在。要创建局部变量（参数会自动成为局部变量），可在任何函数内使用 `var` 语句，如下所示：

```
function funcName() {  
    var temp = "just testing!"; // 声明局部变量temp  
}
```

局部变量经常用于存储临时的信息。例如，使用局部变量 lastSpacePlusOne 来保存一个内部的结果。和所有的局部变量一样，它在函数结束后就死亡了：

```
function getLastWord(text) {
    var lastSpacePlusOne=text.lastIndexOf(" ")+1; // 局部
    var lastWord=text.substring(lastSpacePlusOne, text.length); // 局部
    return lastWord;
}

// 显示: "Word"
trace(getLastWord('Tell me the last word'));

// 显示: undefined, lastSpacePlusOne 是局部变量,
// 在 getLastWord() 函数之外不能被访问
trace(lastSpacePlusOne);
```

当局部变量在函数的结尾死亡时，和它们相对应的存储器空间就被释放。使用局部变量存储所有的临时值，就能在程序中避免存储器的浪费。而且，当定义一个局部变量的时候，不需要担心它会和同名的时间线变量相冲突。

当然，不是所有用在函数中的变量都是局部变量。我们在前边已经学到，可以从函数内读到一个时间线变量，也可以创建和修改它们。函数内不适用于局部变量的变量赋值语句，其作用域在时间线上而不在函数内。在下面的例子中，x 是一个局部变量，y 和 z 是时间线变量：

```
var z=1;

function createVars() {
    var x=10; // 创建局部变量 x
    y=13; // 创建时间线变量 y
    z=2; // 修改时间线变量 z
}
createVars(); // 调用函数
trace(x); // x 是 undefined (x 在函数结束的时候就已经死亡)
trace(y); // y 是 13 (y 在函数结束后依然存在)
trace(z); // z 为 2 (z 被函数永久地改变了)
```

即使函数是某个对象的方法，创建时间线变量的规则也是适用的。虽然我们要到第十二章才开始讨论对象，但是，这些东西和面向对象的编程非常相似，例 9-2 就证明了这一点。注意，x 是在时间线上定义的，因为它不存在于 newFunc() 的局部作用域中，也不是 newObj 的一个属性。

### 例 9-2：对象方法内的变量作用域

```
newObj=new object();           // 创建一个对象
newObj.newFunc=function() { x=12; }; // 添加一个新的方法
newObj.newFunc();              // 调用方法

// 现在我们来搜索 x
trace("x is "+x);           // x 为 12
trace("newObj.x is "+newObj.x); // newObj.x 为 undefined
```

## 再论函数参数

既然我们了解了函数的工作情况，现在我们再回过头来看看函数的参数。我们现在讨论要求你具有一定的对象知识，因此，新的程序员可以先读第十二章，然后再看这一节的内容。

### 参数的数目

我们在前边已经说过，创建函数的时候就要定义函数的参数。回忆下面的语法：

```
function funcName (param1, param2, param3,...paramn) {
    statements
}
```

传递给函数的参数数目可以和函数定义中指定的参数数目不同，这也许会让你感到惊讶。函数可以接受任何数目的参数，不管是多于还是少于“规定的”数目。当调用函数的时候传递了少于规定数目的参数时，每一个遗漏的参数值都会被设置为 `undefined`。例如：

```
function viewVars (x,y,z) {
    trace("x is "+x);
    trace("y is "+y);
    trace("z is "+z);
}

viewVars(10);      // 显示: "x is 10", "y is " 以及 "z is ",
                  // 因为 y 和 z 是 undefined (显示为空白)
```

当函数调用的时候传递了多于规定数目的参数时，多余的参数值可以用 `arguments` 对象来访问。（很明显，多余的参数不能像正规定义过的参数那样用名称来访问，因为它们的名字没有得到定义。）

## arguments 对象

在函数的执行中，内置的 *arguments* 对象让我们可以访问三个信息：(a) 传递给函数的参数数量；(b) 包含所有参数值的一个数组；(c) 所执行的函数的名称。*arguments* 对象其实是数组和带有其他属性的对象之间的特殊混合物。

### 从 arguments 数组中获取参数值

*arguments* 数组可以让我们检查任何函数的参数值，不管该参数是否在函数的声明语句中得到过定义。要访问一个参数，我们要查看 *arguments* 数组中的索引：

```
arguments[n]
```

*n* 是我们要访问的参数的索引。第一个参数（函数调用表达式最左边的一个）被存储为索引 0，也就是 *arguments[0]*。后面的参数按顺序进行存储，从左到右——因此第二个参数就是 *arguments[1]*，第三个参数就是 *arguments[2]*，依此类推。

从函数内部，我们可以通过检查 *arguments* 的元素数量，得知当前执行的函数传递进来了多少参数，如下所示：

```
arguments.length
```

我们可以很容易地反复向函数传递所有参数，并且把结果显示在输出窗口中，如例 9-3 所示。

#### 例 9-3：显示未知数量的参数

```
function showArgs() {
    for(var i=0; i<arguments.length; i++) {
        trace('Parameter '+ (i+1) + " is " + arguments[i]);
    }
}

showArgs(123, 23, "skip intro");

// 显示……
Parameter 1 is 123
Parameter 2 is 23
Parameter 3 is skip intro
```

*arguments* 数组让我们可以创建非常灵活的函数，它可以接受任意数量的参数。

下面是一个普通的函数，它从场景上删除任意数量的影片剪辑实例复本：

```
function killClip() {  
    for (var i = 0; i < arguments.length; i++) {  
        arguments[i].removeMovieClip();  
    }  
}
```

读者练习：对前面的 *combine()* 函数作修改，让它能接受任意数量的输入。接受任意数量的参数会给函数带来其他什么好处呢？对任意数目的数字参数求平均数的函数又将是如何的？（提示：将所有参数加起来，然后除以参数的数目。）

### callee 属性

正如我们所看到的，`arguments` 数组让我们可以获取函数的参数。`arguments` 对象有一个属性，`callee`，它返回对执行函数的引用。通常，我们知道所调用的函数的名字，但是如果我们正在执行一个用函数直接量来创建的匿名函数，`callee` 属性就很有用了。例 9-4 给出了一个用函数直接量创建的函数，它的执行是递归的，我们不知道它的名称。参见本章后面的部分。

#### 例 9-4：使用 callee 实现倒计时

```
count=function(x) {  
    trace(x);  
    if(x>1) {  
        arguments.callee(x-1);  
    }  
}  
count(25);
```

显然，可以不使用匿名的递归函数来实现倒计时。我们在本章的后面将看到一些更真实的函数递归的例子。

## 原始与复合参数值

关于参数还有一个要考虑的问题——向函数传递原始数据和复合数据之间的差别。

当将一个原始数据值作为参数值传递给函数的时候，函数接收到的是数据的一个拷贝，而不是原数据。在函数内对参数作修改并不会影响函数外面的原变量。在例

9-5 中, `variableName` 的值最初被设置为 25, 在 `setValue()` 函数中又将它的值设置为 10, 但这并不影响 `y` 的值。

#### 例 9-5: 原始数据值的传递

```
var y=25;
function setValue(variableName) {
    variableName=10;
}
setValue(y);
trace('y is '+y); // 显示: "y is 25"
```

因此, 原始数据可以说是值的传递。但是, 当我们将复合数据作为参数传递给函数的时候, 函数接收到的是指向与原变量相同的数据的引用, 而不是数据的拷贝。通过参数而改变数据将影响原来的数据, 因而也同时影响指向同一个数据的其他变量, 甚至包括函数之外的变量。因此, 复合数据可以说是引用的传递。

在例 9-6 中, 对 `myArray` 参数变量的修改会影响外面的 `boys` 数组, 因为它们都指向存储器中的相同数据。

#### 例 9-6: 修改以引用传递的复合数据

```
// 创建一个数组
var boys = ["Andrew", "Graham", "Derek"];

// setValue() 设置数组中第一个元素的值
function setValue(myArray) {
    myArray[0] = "Sid";           // 设置数组中第一个元素
}

// 将数组传递给函数
setValue(boys);

// 检查数组元素的值
trace("Boys: "+ boys);        // 显示: "Boys: Sid,Graham,Derek"
```

注意, 虽然我们可以从函数内部重写单个的数组元素值, 但给参数赋一个新的值将会打断它和原变量的联系。以后如果改变参数变量, 将不会影响原来的变量。在例 9-7 中, 虽然 `boys` 数组被作为参数传递, 但是 `myArray` 参数变量很快被设置成了 `girls`。以后再改变 `myArray` 将影响 `girls` 数组, 而不是 `boys` 数组。

#### 例 9-7: 打断变量和参数之间的联系

```
// 创建两个数组
var boys = ["Andrew", "Graham", "Derek"];
var girls = ['Alisa', 'Gillian', 'Daniella'];
```

```
// setValue() 忽略所传递的数组，而去修改 girls 数组
function setValue(myArray) {
    myArray = girls;           // 让 myArray 指向 girls，而不是 boys
    myArray[0] = "Mary";       // 改变 girls 的第一个元素
}

// 将 boys 数组传递给 setValue() 函数
setValue(boys);

trace('Boys: ' + boys);    // 显示: "Boys: Andrew,Graham,Derek"
trace("Girls: " + girls);  // 显示: "Girls: Mary,Gillian,Daniella"
```

关于原始和复合数据的更多信息可以在第十五章中看到。

## 递归函数

递归函数是可以调用自身的函数（在它的函数体内使用自己的函数名）。下面的例子给出了递归的基本原则。但是，由于代码让 *trouble()* 函数重复执行（就像一个图像在两面镜子之间无限映射一样），Flash 很快就会堆栈溢出，发生错误：

```
function trouble() {
    trouble();
}
```

实际应用中的递归函数只在给定的条件得到满足的情况下才调用自身（因此不会造成无限递归）。例 9-4 用递归的方法来计算一个数字的阶乘，但是很明显，即使不用递归的方法，这个结果也可以实现。

使用递归的一个典型实例就是计算数字的数学阶乘。3 的阶乘（在数学术语中写为  $3!$ ）是  $3 \times 2 \times 1 = 6$ 。而 5 的阶乘为  $5 \times 4 \times 3 \times 2 \times 1 = 120$ 。例 9-8 给出了一个用递归来实现的阶乘函数。

### 例 9-8：用递归来计算阶乘

```
function factorial(x) {
    if(x < 0) {
        return undefined;    // 错误条件
    } else if (x <= 1) {
        return 1;
    } else {
        return x * factorial(x-1);
    }
}
```

```
trace (factorial(3));    // 显示: 6
trace (factorial(5));    // 显示: 120
```

通常，要实现一种目的都有多种方法。我们也可以用一个循环来计算阶乘，而不用递归，如例 9-9 所示：

#### 例 9-9：不用递归的阶乘计算

```
function factorial(x) {
    if (x < 0) {
        return undefined;      // 错误条件
    } else {
        var result = 1;
        for (var i = 1; i <=x; i++) {
            result = result * i;
        }
        return result;
    }
}
```

例 9-8 和例 9-9 给出了解决相同问题的两种不同方法。递归方法认为：“6 的阶乘就是 6 乘以 5 的阶乘。而 5 的阶乘就是 5 乘以 4 的阶乘……”依此类推。而非递归的方法从 1 循环到 x，将它们全部乘在一起，形成一个大的数字。

哪种方法更好——递归或者非递归——要看所解决的问题是什么。一些问题用递归方法可以很容易地解决，但是递归会比非递归的解决方式更慢。递归方法在不知道一个数据结构的嵌套有多深的时候比较适用。例如，假设想列出某个子目录中所有的文件，包括所有嵌套子目录中的文件，层层深入。那么，如果不用递归的方法，很难写出一个能适应任何子目录数量的解决方案。递归的解决方法如下所示：

```
function listFiles(directoryName) {
    do (check the next item in directoryName) {
        if (this item is a subDirectory itself) {
            // 在新的子目录中递归调用该函数
            listFiles(subDirectoryName);
        } else {
            // 显示文件的名字
            trace(filename);
        }
    } while (there are still items to check);
}
```

当我们接触第三部分中的 XML 对象的时候，会使用递归的方法列举一个 XML 文档中的所有元素。

## 内部函数

虽然我们已经了解了创建用户自定义函数的方法，但是，别忘了 ActionScript 还有很多内置函数（类似于语言中的动词）。我们已经看到，一些内置函数可以让我们对数据进行操作。我们也已经接触到了一些控制 Flash 影片和用户环境的函数。

例如，要控制影片剪辑中的播放头，可以调用 *gotoAndPlay()* 函数，将帧号作为参数传递：

```
gotoAndPlay(5);
```

如果你是一个新程序员，你可能马上就会恍然大悟了。你会注意到，用函数调用操作符（括号）和参数列表（本例中也就是值 5）来调用内置函数其实与调用用户自定义的函数是一样的！内置函数，比如 *gotoAndPlay()*，和我们自己建立的函数非常类似。通常，内置函数所完成的工作和我们自定义的函数有所不同，我们完全没有必要建立一个自定义的函数来完成 ActionScript 内置函数已经做过的工作。但是，和任何自定义函数一样，每一个内置函数都有一个名称，有可选的参数，以及一个返回值（虽然有的时候为 *undefined*）。

虽然 Flash 长期以来将 *gotoAndPlay* 看作一个“动作”，但我们现在将它的形式看成一个内部函数。在第六章中我们知道，Flash 动作有些是语句，有些是函数。既然你两者都已经学过，那么，你应该能够分辨什么是语句，什么是函数。

如果将特定的动作看作函数，那么你就会很容易地使用它。例如，要将一个影片装载到 Flash 播放器中，我们需要知道 *loadMovie()* 函数是否存在，以及它需要什么参数。在浏览完第三部分之后，我们可以很容易地把发现的信息放在一起构成下面的语句：

```
loadMovie('myMovie.swf',1);
```

现在你又一次轻松地复习了函数的使用。

ActionScript 的内置函数不但数量多，而且种类也多。它们可以控制影片的元素，能够检查和修改所有的东西，从音量到编辑文本域中选中文本的数量。第三部分中有 ActionScript 内置函数的详尽列表。你可以经常地去熟悉一下有效的函数类型，不过没有必要记住它们的特定语法。

## 内部函数的有效性

一些内置函数在很多情况下都有效，而另外一些只能用对象来调用。如果一个函数为全局函数，它就可以在任何地方使用；如果函数是对象中的方法，那么它的使用必须和对应的对象相联系。因为我们还没有讨论到对象，而且 ActionScript 中的大部分内置函数都是对象方法，因此我们要暂停对内置函数的探讨，把它放到第十二章。

## 函数对象

在 ActionScript 中，函数从理论上说是一种特殊类型的内置对象。我们来看看这究竟是什么意思，以及在处理函数时会造成的影响。

### 将函数传递给函数

也许很让人惊奇，我们可以将任何函数作为参数传递给另外的函数，如下所示：

```
function1(function2);
```

注意，如果 *function2* 后面没有括号，那么解释程序不会执行 *function2()*，而是将它的“对象引用”作为参数传递给 *function1()*。也就是说，*function1()* 接收到了 *function2* 本身，而不是 *function2* 的返回值。因为对象是通过引用进行传递的，我们就可以将一个函数的标识符传递给另外一个函数，这个函数将完整地传递。传递函数的执行如下所示：

```
function doCommand(command) {
    command();           // 执行传递的函数
}

// --例子:
doCommand(stop);    // 传递内部 stop() 函数 (终止当前影片)
doCommand(play);    // 传递内部 play() 函数 (播放当前影片)
```

因为函数是一种类型的对象，可以将它们当作任何其他数据来对待。在下面给出的例子中，我们将内部 *gotoAndPlay* 函数赋值给变量 *gp*，它给我们提供了一个指向函数的便捷方法：

```
gp = gotoAndPlay; // 创建一个指向 gotoAndPlay() 的捷径  
gp(25); // 用引用来调用 gotoAndPlay()
```

除了将函数作为对象传递和存储之外，还可以通过对函数添加属性而使用函数的“对象状态”，如下所示：

```
// 创建一个函数  
function myFunction() {  
    trace(myFunction.x);  
}  
  
// 对它添加一个属性  
myFunction.x=15;  
  
// 调用函数以检查属性的值  
myFunction(); // 显示：15
```

---

**注意：**对函数添加属性，就可以在函数的执行中间维持信息的状态，而不扰乱时间线的变量。

---

函数属性具有局部变量所具备的优点，并且不会在函数调用完之后就消失。当一个函数需要用特定的标识符来调用的时候，这点非常有用。例如，下面给出一个普通的函数，它将复制一个影片剪辑，并且给复制的剪辑一个特殊的名称和标签：

```
makeClip.count=0; // 定义 makeClip() 的一个属性（记住，makeClip() 已经存在，  
// 因为函数在代码运行之前就已经定义了）  
  
// 复制传递进来的剪辑，并且自动给新的剪辑命名  
function makeClip (theClip) {  
    // 将剪辑的计数器加上1  
    makeClip.count++  
  
    // 现在复制剪辑，并为它赋一个独特的名称和深度  
    theClip.duplicateMovieClip(theClip._name+makeClip.count, makeClip.count);  
}  
  
makeClip(square); // 用 makeClip() 建立一个 square 的副本  
square1._x+=100; // 现在将复制的 square 移动到右边
```

## 代码的集中

函数最重要的功能也许就是它的可重复使用性——我们可以创建一个可以重复使

用的代码段，它可以在影片的任何地方得到执行。因为函数可以远程执行，它们就能存储在中央位置，使代码的维护和升级更为容易。

例如，假设我们有三个按钮，分别在不同的影片剪辑中，它们的作用都相同。我们没有必要编写三遍代码，可以将通用的代码放在函数中，将函数放在主时间线上，以备从需要的按钮上调用。这不仅节省了时间，而且减少了错误产生的可能性，要一次改变三个按钮也非常容易。因为代码是集中的，所以测试很可靠，故障查找也比较容易。如果一个按钮可以工作而另外一个不能，问题就很可能出在第二个按钮的函数调用上，而不是你的集中代码上。

## 再看多项选择测试

在例 1-1 中，我们向新程序员们介绍了一个简单的脚本化影片——一个多项选择测试。现在我们再来看看这个测试，看看我们如何能将代码进行集中。

新测试的层结构和前面差不多。我们只改变第一帧的代码，也就是 `quizEnd` 帧，以及帧上的按钮。

你可以从在线代码库中获得新旧版本测试的 `.fla` 文件。

## 将测试代码集中为函数

在第一次试图创建一个多项选择测试的时候，我们将代码分散在影片中，将测试的逻辑放在三个地方：对测试进行初始化第一帧中；追踪用户答案的按钮上；给出用户成绩的 `quizEnd` 帧中。现在，我们要将测试的逻辑集中起来，只在第一帧中就完成所有这些任务。要初始化测试，我们仍然使用简单的语句系列，就如同以前所做的那样，但是，如果要追踪用户答案，计算用户的成绩，就要使用两个函数，`answer()` 和 `gradeUser()`。

例 9-10 给出了测试第一帧中的代码。在这里存储了所有需要用来运行测试的逻辑。先看看代码的内容，然后我们再来分析。

### 例 9-10：多项测试，版本 2

```
// 在第一个问题上终止影片  
stop();
```

```
// 初始化主时间线变量
var displayTotal;           // 显示用户成绩的文本域
var numQuestions = 2;        // 测试题目的数量
var q1answer;                // 用户对问题一的答案
var q2answer;                // 用户对问题二的答案
var totalCorrect = 0;         // 回答正确的问题数目
var correctAnswer1 = 3;       // 第一题的正确选择
var correctAnswer2 = 2;       // 第二题的正确选择

// 记录用户答案的函数
function answer (choice) {
    answer.currentAnswer++;
    set ("q" + answer.currentAnswer, "answer", choice);
    if (answer.currentAnswer == numQuestions) {
        gotoAndStop ("quizEnd");
    } else {
        gotoAndStop ("q" + (answer.currentAnswer + 1));
    }
}

// 计算用户成绩的函数
function gradeUser() {
    // 计算用户答对了多少题目
    for (i = 1; i <= numQuestions; i++) {
        if (eval("q" + i + "answer") == eval("correctAnswer" + i)) {
            totalCorrect++;
        }
    }

    // 在屏幕文本域上显示用户的成绩
    displayTotal = totalCorrect;
}
```

我们的第一个任务是用 *stop()* 函数终止影片，这样它就不会播放出所有的题目。下面，我们要初始化测试的时间线变量。你会认出 *displayTotal*, *totalCorrect*, *q1answer* 和 *q2answer* 是第一个版本中的东西。我们还添加了 *numQuestions* (用在 *answer()* 及 *gradeUser()* 函数中) 和 *correctAnswer1* 和 *correctAnswer2* (在评定测试成绩的时候使用)。

初始化变量完成之后，我们可以创建 *answer()* 函数了，它将记录用户的答案，并将播放头推进到下一个题目。*answer()* 函数需要一个参数，也就是 *choice*，它是用户回答每一道题目答案的号码，因此，函数的声明应该为：

```
function answer (choice) {
```

每一次给出答案，函数就将 *currentAnswer* 加上 1，它是一个用来追踪题目回答情况的属性：

```
answer.currentAnswer++;
```

下面，将用户的选择存储在与所答题目相对应的动态命名的时间线变量中。用 `currentAnswer` 属性的值来确定时间线变量的名称（比如 `q1answer`, `q2answer`）：

```
set ("q" + answer.currentAnswer + "answer", choice);
```

将用户的选择存储在恰当的变量中之后，如果已经到了最后一道题目，那么测试就结束了；否则，要进入下一道题，它所在的帧标签为 "`q" + (answer.currentAnswer+1)`"：

```
if (answer.currentAnswer == numQuestions) {  
    gotoAndStop("quizEnd");  
} else {  
    gotoAndStop("q" + (answer.currentAnswer+1));  
}
```

这段代码控制答题逻辑。`answer()` 函数准备处理来自测试中任何题目的答案。现在，我们来建立一个计算所有这些答案的函数，`gradeUser()`。

`gradeUser()` 函数没有参数。它必须将每一个用户答案同正确的答案进行比较，然后显示用户的成绩。我们在 `for` 循环中进行比较——循环的次数也就是测试中题目的数量：

```
for (i = 1; i <= numQuestions; i++) {
```

在循环中，比较表达式会将用户的答案同正确答案进行比较。使用 `eval()`，可以动态地获得每一个用户答案变量的值，以及每一个正确答案变量。如果两个变量是相等的，`totalCorrect` 就加 1：

```
if (eval("q" + i + "answer") == eval("correctAnswer"+i)) {  
    totalCorrect++;  
}
```

在循环结束之后，`totalCorrect` 将包含用户回答正确的问题数目。我们动态地将文本域 `displayTotal` 设置为 `totalCorrect`，就可以显示这个数字：

```
displayTotal = totalCorrect;
```

哦！两个函数都完成了，我们的测试也已经初始化了，测试系统的逻辑也完成了。注意，当大部分代码都集中在单一的帧中时，要跟随测试的操作会非常简单。剩下的所有工作就是从测试中恰当的位置上调用函数了。

## 调用测试函数

我们要从用户点击的答案按钮上调用 *answer()* 函数，还要从 *quizEnd* 帧中调用 *gradeUser()* 函数。

在第 1 帧的第一个答案按钮上，添加下面的代码：

```
on(release) {  
    answer(1);  
}
```

在第二个按钮上，添加的代码为：

```
on(release) {  
    answer(2);  
}
```

对于第三个按钮，添加代码：

```
on(release) {  
    answer(3);  
}
```

然后我们将完全相同的代码添加到 q2 帧的三个按钮上。注意，这个过程是非常正规的。当需要用来处理每一个按钮的代码整齐地安排到函数中以后，我们的按钮代码就可以保持最少了。在我们以前的测试版本中不必要地重复了答案和测试进程逻辑。另外，创建新的按钮只不过是简单地改变对 *answer()* 函数传递的参数的问题。我们不需要为测试的最后一个题目做特别的工作，因为 *answer()* 函数会自动进行处理。

当按钮代码编写完成之后，只需在测试的最后评定用户的成绩就可以了，通过往 *quizEnd* 帧中添加下面的一行代码就可以实现：

```
gradeUser();
```

通过又一次将评分功能与其他函数一起放到了影片的第一帧，我们已经将系统的结构和系统的使用区分开来。因此，我们在 Flash 影片可能形成的帧和按钮的错综复杂的状态下没有丧失这些结构性。

测试你的测试影片，确保它的运行。你还可以实验一下，看看如果添加、删除测试题目，或改变其他东西会发生什么情况。在继续以后的讨论之前，思考一下我们所学的技术将如何应用于我们的测试示例。我们能为用户添加什么功能，我们能对代码作出什么改变，让它更灵活，更有效，即使用户的使用过程不发生改变。

## 小结

本章介绍了 ActionScript 中功能最强大的部分：函数。在下一章里，我们要学习一种特殊的函数，叫作事件处理器，它是由解释程序自动执行的。事件处理器是建立交互功能的关键。

# 第十章

## 事件和事件处理器

我们已经学习了很多关于执行 ActionScript 解释程序的组合指令，到现在为止，我们已经可以非常熟练地告诉解释程序我们要做什么了，但是，我们如何告诉它何时执行这些动作呢？ActionScript 代码并不会自动执行——总是需要有什么东西驱使着它执行。

这个东西就是影片播放器的同步机制，或者是预先指定的异步事件的发生。

### 同步代码的执行

当影片播放的时候，时间线的播放头从帧移动到帧。每一次当播放头进入一个新的帧时，解释程序就执行属于该帧的任何代码。帧中的代码得到执行之后，屏幕显示就会更新，并播放声音。然后，播放头又进入到下一帧。

例如，当将代码直接放在影片的第 1 帧上时，会先执行代码，然后显示第一帧内容。如果我们在同一个影片的第 5 帧关键帧中放置另外的代码块，会先显示帧 1 到帧 4 的内容，然后执行第 5 帧的代码，最后显示第 5 帧的内容。在第 1 帧和第 5 帧中执行的代码被称为是同步执行的，因为它们是以可预知的形式线性执行的。

属于影片帧的所有代码都是同步执行的。即使一些帧的播放由于使用 *gotoAndPlay()*

或者 `gotoAndStop()` 命令而次序颠倒，但是，每一帧的代码仍然是按照预先制定的顺序来执行的，和播放头的移动同步。

## 基于事件的异步代码执行

有的代码并不按照预定的顺序来执行。相反，它要在 ActionScript 解释程序注意到某个预定事件发生的时候才会执行。用户通过一些动作可引发许多事件，比如点击鼠标或者按键动作。正如播放头进入一个新的帧会同步执行属于该帧的代码一样，事件可以导致基于事件的代码得到执行。基于事件的代码（为响应事件而执行的代码）被称为是异步执行的，因为触发事件可以在任意时间发生。

同步编程要求我们预先指定代码执行的时间。而异步编程，换句话说，让我们在事件发生的时候能够动态地做出响应。异步代码的执行是 ActionScript 及其交互性的重点。

本章将讨论 Flash 中的异步（基于事件）编程，并且列出 ActionScript 所支持的不同事件。

## 事件的类型

从概念上说，事件可以被分为两类：

### 用户事件

用户产生的动作（比如鼠标的点击和按键）

### 系统事件

作为影片播放中的一部分而发生的事件（比如，一个影片剪辑出现在场景上，或者一系列变量从一个外在的文件装载进来）

ActionScript 在句法上不区分用户事件和系统事件。用户点击鼠标的事件与影片触发的内部事件一样。虽然我们一般不会把影片剪辑在场景上的移动看成一个明显的“事件”，但是能够对系统事件做出反应，可以使我们更好地控制影片。

ActionScript 事件也可以按照它们所属的对象来进行更为实际的分类。所有事件的发生都和 Flash 环境中的某个对象有关。也就是说，解释程序不会仅说“用户点击”，它会说“用户点击了这个按钮”或者“当这个影片剪辑在场景中的时候用户点击”。解释程序不会仅说“数据接收到了”，它还会说“这个影片剪辑接收了一些数据”。我们将作为事件反应的代码定义在和事件有关的对象上。

可以接收事件的 ActionScript 对象有：

- 影片剪辑
- 按钮
- *XML* 和 *XMLSocket* 类的对象

正如我们将在本章中看到的那样，ActionScript 实际上有两种不同的事件实现：一类针对和影片剪辑与按钮相关的事件；另外一类针对所有其他种类的对象。

## 事件处理器

不是所有的事件都可以触发代码的执行。规律性的事件并不会影响影片。例如，一个用户可以通过不停地点击按钮而产生无数个事件，但是这些点击会被忽略。为什么？因为这些事件并不会主动地执行代码——我们必须明确地编写对事件做出反应的代码。要命令解释程序执行某个事件的对应代码，我们添加一种所谓的事件处理器，它描述指定事件发生时要执行的动作。事件处理器之所以这么命名，是因为它能捕捉，或者说处理影片中的事件。

一个事件处理器类似于一个特殊命名的函数，当某个特殊事件发生的时候，它就被自动触发。因此，创建一个事件处理器和创建一个函数非常相似，只有下面所示的一些变化：

- 事件处理器要预先确定名称，比如 *keyDown*。不能随心所欲地对事件处理器命名，必须使用预先指定的名称，如后面的表 10-1 和 10-2 所示。
- 事件处理器不用 *function* 语句来声明。
- 事件处理器必须属于按钮，影片剪辑或者对象，而不能属于帧。

---

**警告：**大部分事件都是在 Flash 5 中首次引入的。如果要导出为 Flash 4 格式的文件，只能使用按钮事件处理器（Flash 4 中只支持按钮事件），并且要在 Flash 4 播放器中小心地测试你的作品。

---

## 事件处理器语法

事件的名称（以及它们的对应事件处理器）要由 ActionScript 预先确定。按钮事件处理器是用 `on(eventName)` 来定义的，而影片剪辑事件处理器用 `onClipEvent(eventName)` 来定义，`eventName` 是要处理的事件名称。

因此，按钮事件处理器（除了 `keyPress`，它还需要一个 `key` 参数）的格式如下：

```
on(eventName) {  
    statements  
}
```

一个单独的按钮处理可以响应多个事件，它们彼此用逗号分隔开。例如：

```
on(rollover,rollOut) {  
    // 对 rollOver 和 rollOut 事件触发一个自定义的函数  
    playRandomSound();  
}
```

所有的影片剪辑事件处理器的格式为：

```
onClipEvent(eventName) {  
    statements  
}
```

与按钮处理不同，剪辑事件处理只能对一个单独的事件做出响应。

## 创建事件处理器

要创建一个事件处理器，首先要定义处理器，然后将它添加到相应的对象上。我们可以从最普通的处理器开始——属于按钮和影片剪辑的那些处理器。

## 将事件处理器放到按钮和影片剪辑上

要将一个事件放到按钮或者影片剪辑上，我们必须物理地将处理器函数的代码放到对应的按钮或者剪辑上。我们只可以在 Flash 创建工具中这样做，只要选择场景上的对象，然后在 Actions 面板中输入对应的代码就可以了，如图 10-1 所示：

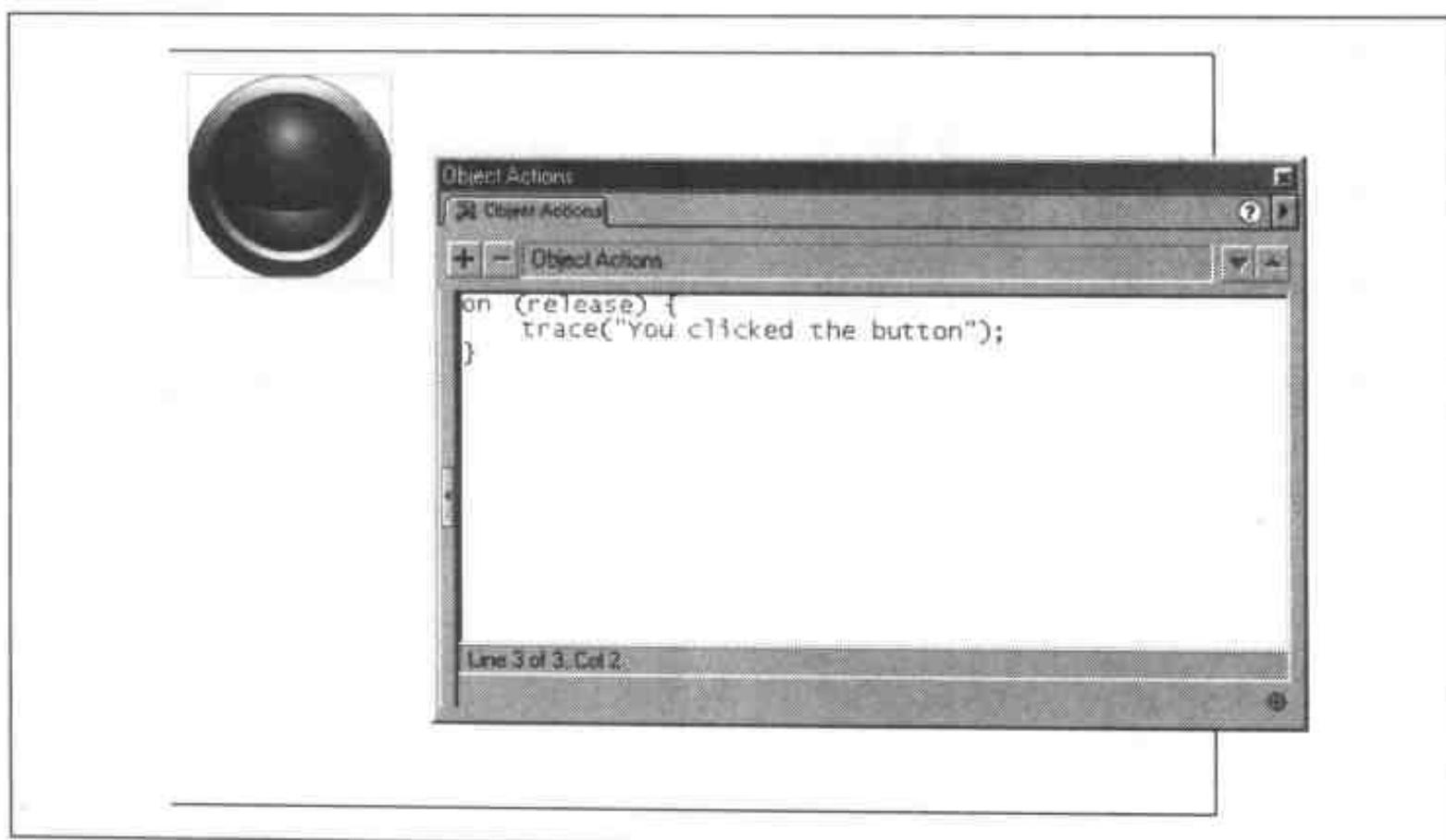


图 10-1 将事件处理器添加到一个按钮

1. 建立一个新的 Flash 影片。
2. 创建一个按钮，并拖动一个实例到主场景中。
3. 选中按钮，然后将下面的代码输入到 Actions 面板中：

```
on(release) {
    trace("You clicked the button");
}
```

4. 选择 Control (控制) → Test Movie (测试影片)。
5. 点击按钮。消息“You clicked the button”将会出现在输出窗口中。

当影片播放的时候，点击并释放按钮，*release* 事件就会被解释程序发现，它就会执行 *on (release)* 事件处理器。每一次点击和释放按钮，消息“You clicked the button”就会出现在输出窗口中。

现在我们试着在一个影片剪辑中创建一个更为有趣的事件处理器。同样要遵循下面的步骤：

1. 建立一个新的 Flash 影片。
2. 在主影片场景中拖出一个矩形。
3. 选择 Insert (插入) → Convert to Symbol (转换为符号)。
4. 在 Symbol properties (符号属性) 对话框中, 将新的符号命名为 rectangle, 并选择影片剪辑作为其类型。
5. 点击 OK, 结束 rectangle 影片剪辑的创建。
6. 选择场景上的 rectangle 剪辑, 然后在 Actions 面板中输入下面的代码:

```
onClipEvent (keyDown) {  
    _visible = 0;  
}  
  
onClipEvent (keyUp) {  
    _visible = 1;  
}
```

7. 选择 Control (控制) → Test Movie (测试影片)。
8. 点击影片, 确保它有键盘焦点, 然后按住任何键。每一次按住某个键时, rectangle 影片剪辑就会消失, 放开所按住的键时, rectangle 又会重新出现。

注意, 我们并不手工处理任何事件触发语句——解释程序会在对应的事件发生时自动触发事件处理器。

Flash 不支持在影片播放的时候通过 ActionScript 来添加或者删除处理器。事件处理器必须用 Flash 创建工具附属在按钮和影片剪辑上。因此, 下面的虚构语法是不合法的:

```
myClip.onKeyDown = function () {_visible = 0;};
```

我们会在后面看到这些缺点是如何解决的。

## 将事件处理器添加到其他对象

除了影片剪辑和按钮之外, 有两个内置对象类——*XML* 和 *XMLSocket*——支持

事件处理器。对于这些对象来说，事件处理器并不是在创建工具中添加到某些物理实体的。实际上，它们是作为方法被添加到对象实例中的。

对于 *XML* 和 *XMLsocket* 对象来说，ActionScript 用预定的属性来保存事件处理器的名称。例如，*onLoad* 属性保存的就是当外部 XML 数据被载入时所执行的处理器的名称。

要为 *XML* 对象设置 *onLoad* 属性，我们使用下面的代码：

```
myDoc = new XML();
myDoc.onLoad = function() {trace("all done loading!");};
```

作为选择，我们可以首先定义处理器函数，然后将它赋给对象的 *onLoad* 属性：

```
function doneMsg() {
    trace("all done loading!");
}
myDoc.onLoad=doneMsg;
```

这个语法很像 JavaScript，函数可以被赋给事件处理器的属性，如例 10-1 所示。

#### 例 10-1：为一个 JavaScript 事件处理器赋值

```
// 在JavaScript中将一个函数常量赋给 onload 处理器
window.onload=function() {alert("done loading");}

// 或者，作为另外的选择，可以先创建函数，然后将其赋给 onload 属性
function doneMsg() {
    alert("done loading");
}
window.onload=doneMsg;
```

在未来，更多的 ActionScript 对象可以支持用对象属性来设置事件处理器的方法，因此，现在熟悉这种形式是很不错的想法。如果你不使用 *XML* 或 *XMLSocket* 对象，仍然可以用这种方法练习用 HTML 文档和 JavaScript 建立处理器。这个方法的好处就是它具有灵活性，任何事件处理器函数都可以容易地被重新赋值甚至在影片播放期间被删除。

我们将在第十二章中学习有关将函数添加到对象的内容。关于 *XML* 和 *XMLSocket* 对象所支持事件的更多信息，可以在第三部分中查找。

---

注意：事件处理器的生命周期和它所对应的对象的生命周期有关。当一个剪辑或者按钮从场景上被删除，或者 XML 或 XMLSocket 对象死亡的时候，任何和这些对象相关的事件处理器都将一同死亡。一个对象必须出现在场景上，或者存在于时间线上，它的处理器才能保持活动。

---

## 事件处理器作用域

和任何函数一样，事件处理器中的语句在预定的作用域范围内执行。作用域表示解释程序能处理一个事件处理器体中所出现的变量、子函数、对象或者属性的区域。我们可以将事件处理器作用域和影片剪辑事件、按钮事件以及其他对象事件放在一起考虑。

### 影片剪辑事件处理器作用域

和正规函数不同，影片剪辑事件处理器不定义局部作用域！当我们为剪辑添加处理器的时候，处理器的作用域就是剪辑，而不只是处理器本身。这意味着所有的变量都可以从剪辑的时间线上获取。例如，如果我们往一个名为 `navigation` 的剪辑中添加一个 `enterFrame` 事件处理器，并且在处理器内写上 `trace(x);` 语句，解释程序就会在 `navigation` 的时间线上寻找 `x` 的值：

```
onClipEvent(enterFrame) {  
    trace(x); // 显示navigation.x的值  
}
```

解释程序不首先考虑局部作用域，因为没有局部作用域。如果我们在处理器中写 `var y=10;` 这样的语句，`y` 就是在 `navigation` 的时间线上定义的，即使 `var` 关键字用在函数中时通常都定义一个局域变量。

记住剪辑事件处理器作用域规则的最简单办法就是将处理器的语句看成是处理器所在剪辑的帧上的。例如，假设有一个名为 `ball` 的剪辑，它有一个变量 `xVelocity`。要从 `ball` 事件处理器中访问 `xVelocity`，只要直接指向它就可以了，如下所示：

```
onClipEvent(mouseDown) {  
    xVelocity+=10;  
}
```

不需要给变量提供路径 `_root.ball.xVelocity`, 因为解释程序已经假定变量 `xVelocity` 在 `ball` 中。对于属性和方法来说也是同样的, 不使用 `ball._x`, 而是使用 `_x`, 不使用 `ball.gotoAndStop(5)`, 只使用 `gotoAndStop(5)`。例如:

```
onClipEvent (enterFrame) {
    _x += xVelocity;           // 移动 ball
    gotoAndPlay(_currentframe-1); // 做一个循环
}
```

我们甚至可以在 `ball` 上用函数声明语句在处理器内定义一个函数, 如下所示:

```
onClipEvent (load) {
    function hideMe() {
        _visibility=0;
    }
}
```

剪辑事件处理器中的语句, 其作用域在剪辑的时间线上, 而不在处理函数的局部作用域, 也不在剪辑的父时间线 (剪辑所在时间线的上层时间线) 上, 这让人很容易忘记。

例如, 假设我们将 `ball` 剪辑放在影片的主时间线上, 而主时间线 (不是 `ball` 的时间线) 中定义了一个 `moveBall()` 函数。我们可能会茫然地从 `ball` 的一个事件处理器中调用 `moveBall()`, 如下所示:

```
onClipEvent (enterFrame) {
    moveBall(); // 什么也不做! ball 中并没有 moveBall() 函数
                // moveBall() 函数在 _root 上
}
```

我们必须用 `_root` 明确地指向主时间线上的 `moveBall()` 函数, 如下所示:

```
onClipEvent (enterFrame) {
    _root.moveBall(); // 现在可以了
}
```

有的时候, 我们可能需要从一个事件处理器内明确地指向当前的剪辑对象。在这个过程中, 可以使用 `this` 关键字, 当在一个事件处理器中使用它的时候, 它会指向当前的影片剪辑。因此, 下面的引用方法在剪辑事件处理器内是同义的:

```
this._x // 和下一行的代码相同
_x
```

```
this.gotoAndStop(12); // 和下一行的代码相同  
gotoAndStop(12);
```

this在我们要动态地产生一个当前剪辑属性的名称（或者一个变量名称，或者一个嵌套剪辑）时是很常用的。因此，如果要让按照ball.stripel, ball.strip2, ...序列组成的一个嵌套剪辑开始播放，就要依靠ball剪辑的当前帧：

```
onClipEvent (enterFrame) {  
    this["stripe"+_currentframe].play();  
}
```

当影片剪辑方法需要有一个显式的引用指向调用它的影片剪辑对象的时候，也经常使用this关键字。任何与ActionScript全局函数同名的影片剪辑方法必须使用一个显式的剪辑引用。因此，在事件处理器中将下面的函数作为方法调用时，this关键字非常重要：

```
duplicateMovieClip()  
loadMovie()  
loadVariables()  
print()  
printAsBitmap()  
removeMovieClip()  
startDrag()  
unloadMovie()
```

例如：

```
this.duplicateMovieClip('ball2',1);  
this.loadVariables("vars.txt");  
this.startDrag(true);  
this.unloadMovie();
```

我们很快将学习这些函数的双重性的所有内容。

注意，当一个剪辑在创建工具中没有指定实例名称，或者我们不知道剪辑名称的时候this关键字仍然能指向当前的剪辑。实际上，使用this，甚至可以将当前的剪辑作为一个引用传递给函数，而不用知道当前剪辑的名称。下面的示例代码都是合法的（而且非常正规）：

```
// 时间线上的代码  
// 下面是一个用来移动剪辑的普通函数  
function move (clip, x, y) {  
    clip._x += x;  
    clip._y += y;  
}  
  
// 剪辑上的代码  
// 调用主时间线函数，传递一个带this关键字的引用，让它移动当前的剪辑  
onClipEvent (enterFrame) {  
    _root.move(this, 10,15);  
}
```

**警告：**在Flash 5播放器的build 30中，有一个故障是：如果使用了串直接量标签，用在剪辑处理器中的`gotoAndStop()`和`gotoAndPlay()`在这个剪辑处理中就无效。这样的命令会被忽略。例如，下面的代码就是无效的：

```
onClipEvent(load) {  
    gotoAndStop("intro"); // 在Flash 5 r30中是无效的  
}
```

要防止这个故障，就要使用一个自反的剪辑引用，比如：

```
onClipEvent(load) {  
    this.gotoAndStop("intro");  
}
```

## 按钮事件处理器的作用域

按钮处理器的作用域就是按钮所在的时间线。例如，如果我们在主时间线上放了一个按钮，并且在按钮的处理器中定义变量`speed`，那么`speed`的作用域就将是主时间线（`_root`）：

```
// 按钮处理器的代码  
on(release) {  
    var speed=10; // 在_root上定义speed  
}
```

相反，如果我们在主时间线上放置一个影片剪辑`ball`，然后在`ball`的处理器上定义变量`speed`，那么`speed`的作用域将是`ball`：

```
// ball处理器上的代码  
on(load) {  
    var speed=10; // 在ball上定义speed, 而不是_root  
}
```

在按钮处理器内，`this` 关键字指向按钮所在的时间线：

```
on(release) {  
    // 将按钮所在剪辑的透明度设置为 50%  
    this._alpha=50;  
    // 将按钮所在的剪辑向右移动 10 像素  
    this._x += 10;  
}
```

## 其他对象事件处理器的作用域

和影片剪辑与按钮的处理器不同，属于内置类（比如 `XML` 和 `XMLSocket`）的事件处理器，其作用域和函数非常相似。一个 `XML` 或者 `XMLSocket` 对象的事件处理器有一个作用域链，它是在定义处理器函数的时候得到定义的。而且，`XML` 和 `XMLSocket` 事件处理器定义了一个局部作用域。所有的函数作用域规则都在第九章中描述过了，这些规则可以直接运用到属于既非按钮又非影片剪辑的对象的事件处理器函数上。

## 按钮事件

表 10-1 简要介绍了按钮可用的各种事件。使用按钮事件，我们可以轻松地创建导航器、表单、游戏和其他界面元素的代码。现在，我们来认真看一下每一种按钮事件，并了解如何对按钮进行编程以使其对鼠标和键盘元素做出反应。

表 10-1 中的每一种按钮事件都是由一个 `on (eventName)` 形式的匹配按钮事件处理器来处理的。`press` 事件用以 `on (press)` 开头的事件处理器来处理。但 `keyPress` 事件处理器例外，它的格式为 `on (keyPress key)`，`key` 是要探测的键。按钮事件只送给鼠标所影响到的按钮。如果有多个按钮重叠，那么最上层的按钮就接收所有的事件，其他的按钮都得不到响应，即使最上层按钮没有定义处理器也是如此。在下面的描述中，点击区域指的是按钮上能够在鼠标点击下让按钮做出响应的范围。（一个按钮的点击区域是在 Flash 创建工具中创建按钮的时候就定义的。）

表 10-1 按钮事件

按钮事件名称	按钮事件发生的时机
press	当鼠标指示器在按钮的点击区域中时，主鼠标键被按下。其他的鼠标键不能被检测到。

表 10-1 按钮事件（续）

按钮事件名称	按钮事件发生的时机
release	当鼠标指示器在按钮的点击区域中时，主鼠标键被按下，然后又被放开。
releaseOutside	当鼠标指示器在按钮的点击区域内时主鼠标键被按下，然后，当鼠标指示器离开点击区域之后主鼠标键又被放开。
rollOver	鼠标指示器进入按钮的点击区域，但没有按下鼠标键。
rollOut	鼠标指示器离开按钮的点击区域，但没有按下鼠标键。
dragOut	当鼠标指示器在按钮的点击区域内时主鼠标键被按下，然后，在鼠标键仍然被按住的情况下，鼠标指示器移出点击区域。
dragOver	当鼠标指示器在按钮的点击区域内时主鼠标键被按下，然后，在鼠标键仍然被按住的情况下，鼠标指示器离开又返回点击区域。
keyPress	特定的键被按下。在大多数情况下， <i>keyDown</i> 剪辑事件优先于 <i>keyPress</i> 按钮事件。

## press

鼠标点击从理论上来说包含两个步骤：鼠标按钮被按下 (*press*)，然后被放开 (*release*)。一个 *press* 事件发生在鼠标指示器位于按钮点击区域并且主鼠标键被按下时候。次要的鼠标键不会被检测到。按钮 *press* 事件适合于单选钮和游戏中的开火武器，但是，使用 *release* 事件可以让用户在释放鼠标之前改变主意。

## release

*release* 按钮事件发生在下面的一系列情况被检测到的时候：

1. 鼠标指示器在按钮的点击区域中。
2. 当鼠标指示器仍然在按钮的点击区域中时主鼠标键被按下（也就是一个 *press* 事件发生）。
3. 当鼠标指示器仍然在按钮的点击区域内时主鼠标键被释放（也就是一个 *release* 事件发生）。

用 *release* 事件来替代 *press* 事件，可以给用户一个机会，让他们在即使点击了鼠标之后仍然可以将鼠标指示器从按钮上移开，这样就可以撤回他们的决定。

## releaseOutside

*releaseOutside* 事件的典型用途是：用户在点击按钮之后，只要不将鼠标指示器从按钮上移开，那么在他释放鼠标键之前都可以改变主意。这个事件发生在下面的动作序列被检测到的时候：

1. 鼠标指示器在按钮的点击区域内。
2. 主鼠标键被按下并保持按住（*press* 事件发生）。
3. 鼠标指示器移开按钮的点击区域（*dragOut* 事件发生）。
4. 主鼠标键在非原按钮的点击区域中被释放。

你几乎不需要检测 *releaseOutside* 事件，它们通常表示用户无心做出任何动作。

## rollOver

*rollOver* 事件在鼠标指示器移动到按钮的点击区域但是没有按下鼠标键的时候发生。*rollOver* 事件很少用在 ActionScript 中，因为在创建工具中可以直接创建可视化的按钮变化，而不需要脚本。你应该在创建工具中使用给出的 *up*, *over* 和 *down* 帧，以设定按钮的高亮状态。

Flash 5 中的 *rollOver* 事件提供了很多用来获取文本域选择的方法。关于更多的细节内容，可以参见第三部分。

## rollOut

*rollOut* 事件是 *rollOver* 的搭档，它发生在鼠标指示器移离按钮点击区域，同时没有按下鼠标键的时候。和 *rollOver* 一样，*rollOut* 很少使用，因为按钮的高亮状态可以直接在制作工具中创建，因此，手动的图像交换在 ActionScript 中不要求。

## dragOut

*dragOut* 事件和 *rollOut* 很相似，除了发生的时机有所不同。它发生在鼠标键被按住时鼠标指示器离开了按钮点击区域的时候。*dragOut* 事件后面跟着 *releaseOutside* 事件（如果用户释放鼠标键）或者 *dragOver* 事件（如果用户将鼠标指示器移回按钮点击区域，中途没有释放鼠标键）。

## dragOver

*dragOver* 事件很少见。它要在下面的动作序列产生的时候才出现：

1. 鼠标指示器进入按钮的点击区域 (*rollOver* 事件发生)。
2. 主鼠标键被按下，并保持按住状态 (*press* 事件发生)。
3. 鼠标指示器离开了按钮的点击区域 (*dragOut* 事件发生)。
4. 鼠标指示器回到了按钮的点击区域 (*dragOver* 事件发生)。

因此，*dragOver* 事件表示用户将鼠标指示器移开又移回按钮点击区域，其间鼠标键保持按住状态。注意，如果鼠标指示器重新进入按钮点击区域的时候鼠标键仍然是按下的，那么就会发生 *dragover* 事件，而不发生 *rollOver* 事件。

## keyPress

*keyPress* 事件和鼠标事件无关，它是由特定按键被按下的动作触发的。我们在这里描述这个事件是因为它使用 *on (eventName)* 语法，和其他的 ActionScript 按钮事件处理器一样。这个事件处理器要求我们指定触发该事件的键：

```
on (keyPress key) {  
    statements  
}
```

*key* 是一个串，它表示和事件相关的键。这个串可以是键上的字符（比如“s”或者“S”），也可以是格式为“<Keyword>”的表示这个键的关键字。每一个处理器只可以指定一个键。要用 *keyPress* 捕捉多个键，必须创建多个 *keyPress* 事件处理器。例如：

```
// 探测 "a" 键  
on (keyPress "a") {  
    trace("The 'a' key was pressed");  
}  
  
// 探测回车键  
on(keyPress '<Enter>') {  
    trace("The Enter key was pressed");  
}  
  
// 探测下箭头键  
on(keyPress '<Down>') {  
    trace("The Down Arrow key was pressed");  
}
```

*Keyword*的合法值如下（注意，*keyPress*不支持功能键F1...F12，但是可以使用*Key*对象来探测）：

```
<Backspace>  
<Delete>  
<Down>  
<End>  
<Enter>  
<Home>  
<Insert>  
<Left>  
<PgDn>  
<PgUp>  
<Right>  
<Space>  
<Tab>  
<Up>
```

在Flash 4中，*keyPress*是我们和键盘进行交互的惟一途径。在Flash 5及以后版本中，*Key*对象与影片剪辑事件*keyDown*和*keyUp*（后面将会讨论）结合，可以对键盘交互提供更为强大的控制。*keyPress*事件一次只能探测一个单一键，而*Key*对象可以探测同时按下的多个键。

## 影片剪辑事件综述

影片剪辑事件在Flash播放器中普遍存在，从鼠标的点击到数据的下载。剪辑事件可以分为两个类型：用户输入事件和影片播放事件。用户输入事件与鼠标和键盘有关，而影片播放事件与Flash播放器中帧的出现、影片剪辑的产生和死亡，以及数据的载入有关。

注意，用户输入剪辑事件和前面所说的按钮事件在功能上有一定的重迭部分。例如，一个剪辑的 *mouseDown* 事件处理器可以探测鼠标键的点击，正如按钮的 *press* 事件处理器所做的那样。但是，影片剪辑事件没有按钮事件中的点击区域，而且也不能影响鼠标指示器的外观。

我们现在多花一点时间来看看 ActionScript 影片剪辑事件，见表 10-2。我们首先要看的是影片播放事件 (*enterFrame*, *load*, *unload* 和 *data*)，然后我们看看用户输入事件是如何工作的 (*mouseDown*, *mouseUp*, *mouseMove*, *keyDown*, *keyUp*)。每一个剪辑事件都由 *onClipEvent (eventName)* 形式的匹配剪辑事件处理器来进行处理。例如，*enterFrame* 事件就要由以 *onClipEvent (enterFrame)* 开头的一个事件处理器来处理。除了 *load*, *unload* 和 *data* 之外，影片剪辑事件将发送给场景上所有的影片剪辑，即使用户在不同的影片剪辑上点击鼠标（或没有影片剪辑）也是如此。

表 10-2 影片剪辑事件

剪辑事件名称	剪辑事件发生的时机
<i>enterFrame</i>	播放头进入一个帧（在 Flash 播放器播放该帧之前）
<i>load</i>	剪辑首次出现在场景中
<i>unload</i>	剪辑从场景中删除
<i>data</i>	剪辑中的变量装载结束，或者已载入影片的部分装载到一个剪辑中
<i>mouseDown</i>	当剪辑在场景中时主鼠标键被按下（次要的鼠标键不会被探测到）
<i>mouseUp</i>	当剪辑在场景中时主鼠标键被释放
<i>mouseMove</i>	当剪辑在场景中时鼠标指示器移动（哪怕是一点点），即使鼠标不在剪辑上
<i>keyDown</i>	当剪辑在场景中时某个键被按下
<i>keyUp</i>	当剪辑在场景中时某个被按下的键被释放

## 针对影片播放的影片剪辑事件

下面的事件在 Flash 载入和播放影片，不和用户交互的情况下发生。

## enterFrame

如果你曾经凭借一个空的、循环的影片剪辑来触发脚本，*enterFrame*会让你放松一点。*enterFrame*事件发生在每个帧进入影片的时候。例如，如果我们将下面的代码放在一个影片剪辑上，那么这个剪辑就会以每帧10像素的速度变大：

```
onClipEvent(enterFrame) {  
    _height += 10;  
    _width += 10;  
}
```

(注意，正如我们在前面已经学到过的，\_height和\_width属性是在*enterFrame*事件处理器所属剪辑的作用域中进行处理的，因此，在\_height和\_width前面不要求有剪辑实例的名称。)

---

**注意：***enterFrame*事件在每一帧的渲染之前发生，即使包含*enterFrame*处理器的剪辑停止播放。因此，*enterFrame*事件总是可以被触发。

---

所有的Flash影片在Flash播放器中显示的时候都在不变地运行，甚至当屏幕上没有东西移动，或者影片的播放头停止在某帧上的时候也一样。因此，只要剪辑还在场景上，一个独立的影片剪辑的*enterFrame*处理器就会重复地执行，而不管剪辑是播放还是停止。如果一个剪辑的播放头通过*gotoAndStop()*函数调用而被移动，剪辑的*enterFrame*事件处理器仍然会在进入每帧的时候被触发。而且，如果一个完整影片的每个播放头都被*stop()*函数所中断，所有剪辑中的*enterFrame*事件处理器函数仍然会执行。

*enterFrame*事件通常用来重复地更新影片剪辑状态。但是，一个*enterFrame*事件处理器不需要直接提供给包含它的剪辑——*enterFrame*可以和单帧的空剪辑一起使用，重复地执行代码。这个技术称为剪辑事件循环（clip event loop）（或者更简单一点，叫做一个过程），它在第八章中已经描述过了。

注意，*enterFrame*事件处理器中的代码在任何其他代码出现在包含该处理器的剪辑时间线上之前就会得到执行。

更有用的是，使用*enterFrame*可以获得对剪辑的强大控制。后面要给出的例10-7将把我们先前的剪辑放大代码扩展，建立一个影片剪辑的尺寸变换。

## load

*load* 事件在一个影片剪辑产生的时候发生——也就是说，当一个影片剪辑首次出现在场景中的时候。一个影片剪辑出现在场景中可以通过下面的途径：

- 播放头移动到一个关键帧，该帧包含一个放置在创建工具中的新剪辑实例。
- 通过 *duplicateMovieClip()* 函数从一个剪辑复制出另外一个剪辑。
- 剪辑通过 *attachMovie()* 函数程序化地被添加到场景中。
- 一个外部的 .swf 文件通过 *loadMovie()* 函数被装载到了剪辑中。
- 剪辑的内容被 *unloadMovie()* 函数卸载了。（这会导致 *load* 事件的触发，因为一个剪辑的内容被去除的时候，会有一个空的占位符剪辑被装载到该剪辑中。）

*load* 事件处理器体将在首次出现在影片剪辑时间线上的任何代码执行之后才能执行。

*load* 事件处理器经常用来初始化剪辑中的变量，或者完成一些准备性的任务（比如为动态产生的剪辑确定大小，或者定位）。*load* 处理器也可以提供一个很好的方法来防止影片剪辑的自动播放：

```
onClipEvent(load) {  
    stop();  
}
```

*load* 事件处理器也可以用来触发一些依赖于某个特定剪辑的存在而存在的函数，以便能恰当地执行。

*load* 事件和 *duplicateMovieClip()* 函数结合起来的时候会特别有意思，这个函数是用来创建新的影片剪辑的。在例 10-2 中，我们在一个层叠链中使用一个单独的 *load* 事件处理器来产生一组 star 剪辑。*load* 处理器被复制到每一个复制出来的 star 中，并导致 Star 依次自我复制。这个过程当第 100 个剪辑被复制出来的时候才停止。例 10-2 中的 .fla 文件可以在在线代码库中得到。

### 例 10-2：用 *load* 事件产生一组 Star

```
onClipEvent(load) {  
    // 将当前的剪辑放在一个任意的位置上  
    _x=Math.floor(Math.random() * 550);  
    _y=Math.floor(Math.random() * 400);
```

```
// 重新设置剪辑的大小，这样就不用继承先前的剪辑尺寸  
_xscale=100;  
_yscale=100;  
  
// 将当前的剪辑大小设置为 50%-150% 之间的任意尺寸  
randScale=Math.floor(Math.random() * 100)-50;  
_xscale += randScale;  
_yscale += randScale;  
  
// 如果我们还没有复制到第 100 个 star，就继续复制  
if(_name != 'star100') {  
    nextStarNumber=number(_name.substring(4,_name.length))+1;  
    this.duplicateMovieClip('star'+nextStarNumber,nextStarNumber);  
}  
}
```

## unload

*unload* 事件和 *load* 事件相反：它发生在-一个影片剪辑死亡的时候 —— 也就是场景上的剪辑的最后一帧播放完之后（但是在剪辑的第一帧消失之前）。

下面发生的情况将触发 *unload* 事件：

- 播放头到达剪辑所包含的帧的末尾。
- 剪辑被 *removeMovieClip()* 函数删除（它将杀死 *attachMovie()* 和 *duplicateMovieClip()* 函数所创建的剪辑）。
- 预先载入的外部 *.swf* 文件通过 *unloadMovie()* 函数从剪辑中被删除。
- 剪辑有一个外部的 *.swf* 文件装载进来。

*unload* 事件的最后一个触发条件看起来有点奇怪，但它实际上是影片载入到 Flash 中的自然结果。任何时候，当 *.swf* 文件被装载到影片剪辑中时，剪辑以前的内容就被代替，引起 *unload* 事件。

下面的例子说明了 *load* 和 *unload* 事件同 *loadMovie()* 相联系的行为：

1. 在 Flash 制作工具中，我们在影片主时间线的第 1 帧场景上放一个空的影片剪辑，并将它命名为 *emptyClip*。
2. 在主时间线的第 5 帧，我们将影片 *test.swf* 载入到 *emptyClip* 中，使用的是下面的代码：*emptyClip.loadMovie("test.swf");*

3. 我们用 Control (控制) → Play movie (播放影片) 命令来播放影片。

结果是：

1. 帧 1: emptyClip 剪辑出现，触发一个 *load* 事件。
2. 帧 5: *loadMovie()* 函数在下面两个阶段得到执行：
  - a. emptyClip 的占位符内容被删除，以腾出地方给 *test.swf*，这将引发 *unload* 事件。
  - b. 影片 *test.swf* 载入，引起 *load* 事件。

*unload* 事件的典型用法是清除代码——也就是按照某种方法来整理场景或者重新设置程序环境的代码。*unload* 处理器也提供在影片剪辑结束后执行某些动作（比如播放另一个影片）的方法。

## data

*data* 事件发生在外部数据装载到影片剪辑中的时候。*data* 事件按照装载数据类型的不同，可以通过两种完全不同的情况来触发。我们要分别讨论这些情况。

### 以 *loadVariables()* 方式使用 *data* 事件处理器

当我们用 *loadVariables()* 向服务器要求一系列变量的时候，我们必须等待它们完全装载进来才能使用它们的信息（参见第三部分）。

当一个影片剪辑接收完一批载入变量的时候，*data* 事件就被触发了，它告诉我们现在要执行的和这些变量有关的代码是安全的。

例如，假设有一个客户簿影片，其中包括来访者的信息，我们将这些内容存储在服务器里。当一个用户试图查看这些内容的时候，我们要用 *loadVariables()* 去向服务器索取。但是在能够显示这些信息之前，必须启动一个装载屏，直到我们确定所要的信息可以使用了。*data* 事件处理器会在数据完全载入的时候通知我们，这样就可以安全地向用户显示了。

例 10-3 是从一个客户簿中摘取的部分代码，展示了 *data* 事件处理器和 *loadVariables()* 的使用。在这个例子中，按钮从文本文件中装载两个 URL 编码的变量到一个影片剪辑。影片剪辑包括一个 *data* 事件处理器，它在变量装载完成之后得到执行。从处理器内，我们可以显示变量的值。我们知道变量可以安全显示，因为处理器中的代码要通过 *data* 事件的触发才能执行（也就是在接受到数据之后）。

### 例 10-3：等待 *data* 事件

```
// 我们的guestbook.txt文件的内容  
name=judith&message=hello  
  
// 剪辑内的按钮  
on (release) {  
    this.loadVariables("guestbook.txt");  
}  
  
// 剪辑上的处理器  
onClipEvent (data) {  
    trace(name);  
    trace(message);  
}
```

我们在第十七章中建立 Flash 表单的时候要再次用到 *data* 事件。

### 以 *loadMovie()* 的方式使用 *data* 事件处理器

*data* 事件的第二种使用和外部.swf 文件用 *loadMovie()* 函数装载到影片剪辑中的情况有关。当一个.swf 文件装载到主剪辑中的时候，默认情况就是文件马上开始播放，即使只装载了一部分。这总是让人不满意——有的时候我们想要确保.swf 文件在开始播放之前完全上载完毕。可以用 *data* 事件处理器和一些预装载的代码来达到这个目的。

*data* 事件在每一次主剪辑接收到一部分.swf 外部文件的时候都会发生。对于这个“一部分”的定义，其实比你预期的要复杂。为了触发 *data* 事件，至少需要外部.swf 文件所包含的一个完整的新帧被装载进来，它的开始点是：(a) 最近的一个 *data* 事件被引发；(b) 从.swf 文件开始装载的时候。（在这个时间限制中也许有不止一个的帧实际上已经被装载进来了，但是一个帧是触发 *data* 事件最小的数量要求。）

*data* 事件处理器的执行和播放器中帧的渲染有关。随着每个帧的渲染，解释程序会检查一个外部.swf 文件是否被装载到了包括 *data* 事件处理器的剪辑中。如果外部

.swf文件的一部分已经被装载到了这样的一个剪辑，那么载入的部分至少要包含一个帧，*data* 事件处理器才会被执行。这个过程在渲染一个帧的时间间隔中发生一次——仅此一次（即使播放头停止）。

注意，因为*data* 事件发生在每帧的基础上，那么帧数较多的影片就会倾向于有更平滑的预装载器，因为它们在装载.swf文件期间会接收到更频繁的更新。

在*loadMovie()*操作期间，*data* 事件被触发的准确次数依赖于所装载的.swf文件内容的发送，以及连接的速度。一个单帧的.swf文件，不管有多大，都只能引发一次*data* 事件。换句话说，一个有 100 帧的.swf 文件可能会触发单独的*data* 事件 100 次之多，这会取决于影片的帧速率，每帧的字节数，以及网络连接的速度。如果帧数太多而连接太慢，就会触发更多的*data* 事件（最高能到每帧一次）；如果帧数不多，连接速度很快，触发的*data* 事件就会少一些（全部的 100 帧可能在播放器渲染两帧的时间里就传输完，只触发一次*data* 事件）。

我们如何能用*data* 事件处理器来建立一个预装载器呢？不管与*loadMovie()* 函数调用有关的*data* 事件发生在何时，我们都应该外部.swf文件在下载中。因此，从一个*data* 事件处理器内，我们可以检查文件在允许播放之前是否下载了足够多的部分。要达到这个目的，可以使用*getBytesLoaded()* 和 *getBytesTotal()* 函数，如例 10-4 所示。（\_framesloaded 和 \_totalframes 影片剪辑属性也可以使用。）

例 10-4 也提供了影片装载时的反馈。注意，所装载的.swf文件应该在第 1 帧调用一个*stop()* 函数，以防止它在完全下载之前就自动播放。在线代码库上还提供了一个例 10-4 的变体。

#### 例 10-4：data 事件预装载器

```
onClipEvent (data) {
    trace("data received");           // 表演马上就要开始了!
    // 打开数据传输灯
    _root.transferIndicator.gotoAndStop("on");

    // 如果我们完成了装载工作，就关闭传输灯，然后播放影片
    if (getBytesTotal() > 0 && getBytesLoaded() == getBytesTotal()) {
        _root.transferIndicator.gotoAndStop("off");
        play();
    }

    // 在 _root 的文本域变量中显示一些装载细节
```

```
_root.bytesLoaded=getBytesLoaded();
_root.bytesTotal=getBytesTotal();
_root.clipURL=_url.substring(_url.lastIndexOf('/')+1,_url.length);
}
```

## 针对用户输入的影片剪辑事件

影片剪辑事件所剩余的部分和用户的交互有关。当用户输入剪辑事件发生的时候，场景上的所有的剪辑（不管在其他的剪辑中嵌套得有多深）都会接收到这个事件。因此，会有多个剪辑对单独的某个鼠标的点击、移动或者击键动作做出响应。

要基于鼠标对某个指定剪辑的接近性来执行代码，事件处理器就需要检查鼠标指示器和剪辑相关的位置。内置的 *hitTest()* 函数提供了一个简便的方法来检查鼠标点击是否发生在某个特定的范围之内，如后面要给出的例 10-9 所示。

### mouseDown

和 *press* 按钮事件一样，*mouseDown* 剪辑事件检测鼠标键的点击。*mouseDown* 事件在每一次主鼠标键被按下，而鼠标指示器在场景中时发生。

和 *press* 按钮事件不同，*mouseDown* 与按钮的点击区域无关。在同 *mouseUp* 和 *mouseMove* 事件，以及 *Mouse.hide()* 方法结合使用的时候，*mouseDown* 事件可以用来实现一个自定义的鼠标指示器，我们将在例 10-8 中介绍。

### mouseUp

*mouseUp* 事件和 *mouseDown* 事件相对应。它发生在每次主鼠标键被释放，而鼠标指示器在场景中的时候。和 *mouseDown* 一样，一个带 *mouseUp* 处理器的剪辑在鼠标键被释放的时候必须在场景上，才能使事件有一个结果。*mouseUp*，*mouseDown* 和 *mouseMove* 事件可以用来创建鼠标交互的丰富层次，而不影响鼠标指示器的外观（和按钮一样）。

### mouseMove

*mouseMove* 事件让我们可以检测鼠标指示器位置的变化。只要鼠标在移动，*mouse-*

*Move* 事件就会重复发生，速度与处理器可以产生新事件的速度一样快。一个带 *mouseMove* 处理器的剪辑必须在鼠标移动的时候位于场景上，*mouseMove* 事件才会起作用。

*mouseMove* 事件可以用于唤醒空闲应用程序的代码，它可以显示鼠标轨迹，创建自定义的指示器，我们在例 10-8 中将会看到。

## keyDown

*keyDown* 和 *keyUp* 事件是 *mouseDown* 和 *mouseUp* 在键盘上的对应物。它们都提供了基于键盘的交互性编码的基础工具。*keyDown* 事件在键盘上的任何键被按下的时候发生。当一个键保持按住状态的时候，*keyDown* 可以重复发生，这要取决于操作系统和键盘设置。和 *keyPress* 按钮事件不同，*keyDown* 剪辑事件在任何键——不是指定的键——被按下时都会发生。

要捕获（也就是探测或者捕捉）到一个 *keyDown* 事件，必须确保一个带 *keyDown* 事件处理器的影片剪辑在按键的时候位于场景上。下面的代码采用了这个技巧：

```
onClipEvent (keyDown) {  
    trace("Some key was pressed");  
}
```

你会注意到，*keyDown* 处理器不告诉我们究竟是哪个键被按下了。如果我们等待用户随便按个键以继续运行，那么可以不关心按下的是什么键。但是通常，我们需要对特定的键执行一定的动作。例如，我们可能想通过不同的键来让太空船按照不同的方向返回。

要查出是哪个键触发了 *keyDown* 事件，我们可以请教内置 *Key* 对象，它描述键盘的状态。我们所需要的信息的类型是基于我们要产生的交互的。例如，游戏需要从潜在的同时击键中做出瞬间、连续的反馈。相反，导航界面可以只要求一个单一按键的探测（例如，幻灯片的空格键会显示一些说明内容）。

*Key* 对象可以告诉我们最近按下的是哪一个键，以及某个特定的键现在是否被按下。要确定键盘的状态，可使用四个 *Key* 对象方法中的一个：

<i>Key.getCode()</i>	// 最近按键的十进制键控代码值
<i>Key.getAscii()</i>	// 最近按键的十进制 ASCII 值

```
Key.isDown(keycode)      // 如果指定的键现在被按下就返回 true  
Key.isToggled(keycode)  // 确定大写锁定和数字锁定是否被开启
```

例 10-5 给出的 *keyDown* 处理器将告诉我们最近按键的 ASCII 值。

#### 例 10-5：检查最近按下的键

```
onClipEvent (keyDown) {  
    // 获取最近按键的 ASCII 值，并将它转换为字符  
    lastKeyPressed = String.fromCharCode(Key.getAscii());  
    trace("You pressed the '" + lastKeyPressed + "' key.");  
}
```

例 10-6 是一个 *keyDown* 处理器的示例，它检查上箭头键是否是最近被按下的键。

#### 例 10-6：探测上箭头键

```
onClipEvent (keyDown) {  
    // 检查上箭头键是否是最近被按下的键  
    // 上箭头键是用 Key.UP 属性来表示的  
    if(Key.getCode() == Key.UP) {  
        trace('The up arrow was the last key depressed');  
    }  
}
```

要查询键盘的状态有多种方法，你必须选择其中最适合于应用程序的一种。例如，*Key.getAscii()*方法返回和最近按键对应的字符的 ASCII 值，这在不同的语言中可能会产生不同结果（虽然在英语中，字母和数字的键盘位置是标准的）。换句话说，*Key.getCode()*方法返回和键盘上物理键有关的值，而不是特定的字母。*Key.getCode()*可能对国际的或者跨平台的用户更为有用，比如说，你可以使用相邻的四个键来导航，而不管它们所表示的字符。在第三部分中会对这个问题进行进一步的讨论。

可以从在线代码库中下载 *keyDown* 和 *keyUp* 的示例.fla 文件。

---

**注意：**对键盘按键做出反应的事件处理器只在 Flash 播放器有鼠标焦点的时候才能执行。用户必须在影片的击键控制器开始活动之前点击影片的场景，可以规定让用户在进入影片的任何键盘控制部分之前点击一个按钮。

---

## 处理特殊的键

要取消 Flash 独立的播放器菜单命令（打开、关闭、全屏幕等等），可以将下面的代码添加到影片开始处：

```
fscommand("trapallkeys", "true");
```

这个命令也可以防止退出键 Esc 在放映机中退出全屏模式。要捕获放映机中的 Esc，可以使用：

```
onClipEvent(keyDown) {
    if (Key.getCode() == Key.ESCAPE) {
        // 对 Esc 键作出响应
    }
}
```

注意，Esc 键并非在所有的浏览器中都可以捕获。而且，没有办法取消 Alt 键或者 Windows 的 Alt-Tab 或 Ctrl-Alt-Delete 键序列。

要捕获 Tab 按键动作，可以用下面的处理器创建一个按钮：

```
on (keyPress "<Tab>") {
    // 对 Tab 按键作出响应
}
```

在独立的播放器中，Tab 键也可以用剪辑事件处理器来捕获，如下所示：

```
onClipEvent (keyDown) {
    if (Key.getCode() == Key.TAB) {
        // 对 Tab 按键作出响应
    }
}
```

在某些浏览器中，Tab 键只能通过按钮 *keyPress* 事件检测到，它甚至需要结合使用 *keyPress* 按钮事件和 *keyUp* 剪辑事件。下面的代码首先用 *keyPress* 捕获 Tab 键，然后在 *keyUp* 处理器中做出响应。注意，我们不使用 *keyDown* 是因为在 Internet Explorer 中 *Key.getCode()* 对 Tab 键来说只在释放键时才会得到设置：

```
// 主时间线按钮上的代码
on(keyPress "<Tab>") {
    // 在这里设置一个虚拟变量
    foo=0;
}
```

```
// 主时间线影片剪辑上的代码  
onClipEvent (keyUp) {  
    if (Key.getCode() == Key.TAB) {  
        // 现在在 _level0 的 myTextField 中放置指针  
        Selection.setFocus('_level0.myTextField');  
    }  
}
```

我们捕获 Tab 键是为了将插入点移动到表单中某个特定的文本域中。请参见第三部分。

要捕获一个快捷键的结合，比如 Ctrl-F，可以使用 *enterFrame* 处理器和 *Key.isDown()* 方法：

```
onClipEvent (enterFrame) {  
    if (Key.isDown(Key.CONTROL) && Key.isDown(70)) {  
        // Respond to Ctrl-F  
    }  
}
```

要捕获 Enter（或者说回车）键，可以使用按钮处理器，比如：

```
on(keyPress "<Enter>") {  
    // 对 Enter 按键（也就是表单的递交）作出响应  
}
```

也可以使用 *keyDown* 处理器，如下所示：

```
onClipEvent (keyDown) {  
    if (Key.getCode() == key.ENTER) {  
        // 对 Enter 按键（也就是表单的递交）作出响应  
    }  
}
```

参见第三部分可以获得有关捕获其他特殊键（比如功能键 F1, F2 等）或者数字键区中的键方面的更多内容。

## keyUp

当一个按键被释放的时候会触发 *keyUp* 事件。*keyUp* 事件是游戏编程的重要部分，因为它可以关闭所有在前边用 *keyDown* 事件打开的东西——太空船的攻击即为经典范例。让我们看一个更详细的例子，在 Flash 的创建工具中，按下空格键可以暂

时转换到 Hand 工具，而释放空格键又将重新存储前面的工具。这个方法可以被用来显示或者隐藏应用程序中的某种东西，比如临时菜单。

和 *keyDown* 一样，为了从 *keyUp* 事件中获得有用的信息，我们通常将它和 *Key* 对象一起使用：

```
onClipEvent (keyUp) {  
    if (!Key.isDown(Key.LEFT)) {  
        trace("The left arrow is not depressed");  
    }  
}
```

因为 *Key.isDown()* 方法可以在任何时候检查任何键的状态，所以可以用一个 *enterFrame* 事件循环来检查一个特定的键是否被按下。但是，轮流检测键盘（也就是重复地检查一个键的状态）并不比等待 *keyDown* 事件触发事件处理器更为有效。

最终采取什么方法要取决于我们建立的系统类型。在一个经常运动的系统中，比如游戏，轮流检测可能比较恰当，因为我们总是要逐帧进行主动画循环。因此，可以在循环的剩余部分检查 *Key* 对象。例如：

```
// 空剪辑上的代码  
// 让游戏进程运行  
onClipEvent (enterFrame) {  
    _root.mainLoop();  
}  
  
// 主时间线上的游戏代码  
// 这些代码在每帧中执行  
function mainLoop() {  
    if (Key.isDown(Key.LEFT)) {  
        trace("The left arrow is depressed");  
        // 将太空船向左旋转  
    }  
  
    // 检查其他键，然后继续我们的游戏循环  
}
```

在静态界面环境中，没有必要使用 *enterFrame* 循环来检查按键情况，除非你想要探测指定的键盘组合（也就是多个键同时被按下）。平常应该使用 *keyDown* 和 *keyUp* 事件处理器，它们正好每次按键或者释放的时候触发一次。使用 *keyUp* 和 *keyDown* 事件处理器的时候，不需要关心在任何给定的时刻键是否仍然被按下，这可以比较精确地探测按键情况，即使用户在两个帧之间已经释放了键，如果一个键只被按了

一次也可以避免对它检查两次。无论如何，通常都使用 `Key.getCode()` 和 `Key.getASCII()` 方法在 `keyDown` 或者 `keyUp` 事件处理器中检查最近的按键情况。

## 执行的顺序

一些影片的代码分散在多个时间线和多个剪辑事件处理器中。这是很常见的。因此，一个单独的帧会要求执行多个分散的代码块——有的在事件处理器中，有的在剪辑时间线的帧中，还有的在播放器的文档主时间线上。在这些情况下，不同代码块执行的顺序可以变得非常复杂，并且会严重影响程序的效果。熟悉与影片中不同时间线相关的事件处理器执行的顺序，可以确保代码按照意愿来运行。

异步事件处理器独立地执行影片时间线上的代码。比如，按钮事件处理器会在它们所处理的事件发生的时候立即得到执行，`mouseDown`, `mouseUp`, `mouseMove`, `keyDown` 和 `keyUp` 事件的处理器也是一样。

但是，对影片播放事件的处理就是按照影片的进展情况顺序进行的，如表 10-3 所示。

表 10-3 影片剪辑事件处理器的执行顺序

事件处理器	执行时间
<code>load</code>	在剪辑出现在场景中的第 1 帧中执行。它的执行在父时间线代码执行之后，剪辑内部的代码执行之前，并且在帧的渲染之前
<code>unload</code>	在剪辑不出现在场景中的第 1 帧中执行。它的执行在父时间线代码执行之前
<code>enterFrame</code>	在剪辑出现在场景中的第 2 帧和所有的子序列帧中执行。它的执行在父时间线代码执行之前，也在剪辑内的代码执行之前
<code>data</code>	在数据被剪辑接受到的所有帧中执行。如果被触发，它的执行在剪辑内的代码执行之前，也在 <code>enterFrame</code> 代码执行之前

要通过实际的例子来观察表 10-3 中规则的效果很容易。假设有一个单层的影片，它的主时间线上有四个关键帧，我们对每一帧添加一些代码。然后，创建第二个层，在它的第 1 帧中放置一个影片剪辑，使其延续到第 3 帧，但是不出现在第 4 帧中。我们对剪辑添加 `load`, `enterFrame` 和 `unload` 处理器。最后，在剪辑内，我们创建三个关键帧，每个帧都包含一个代码块。图 10-2 给出了影片的示意。

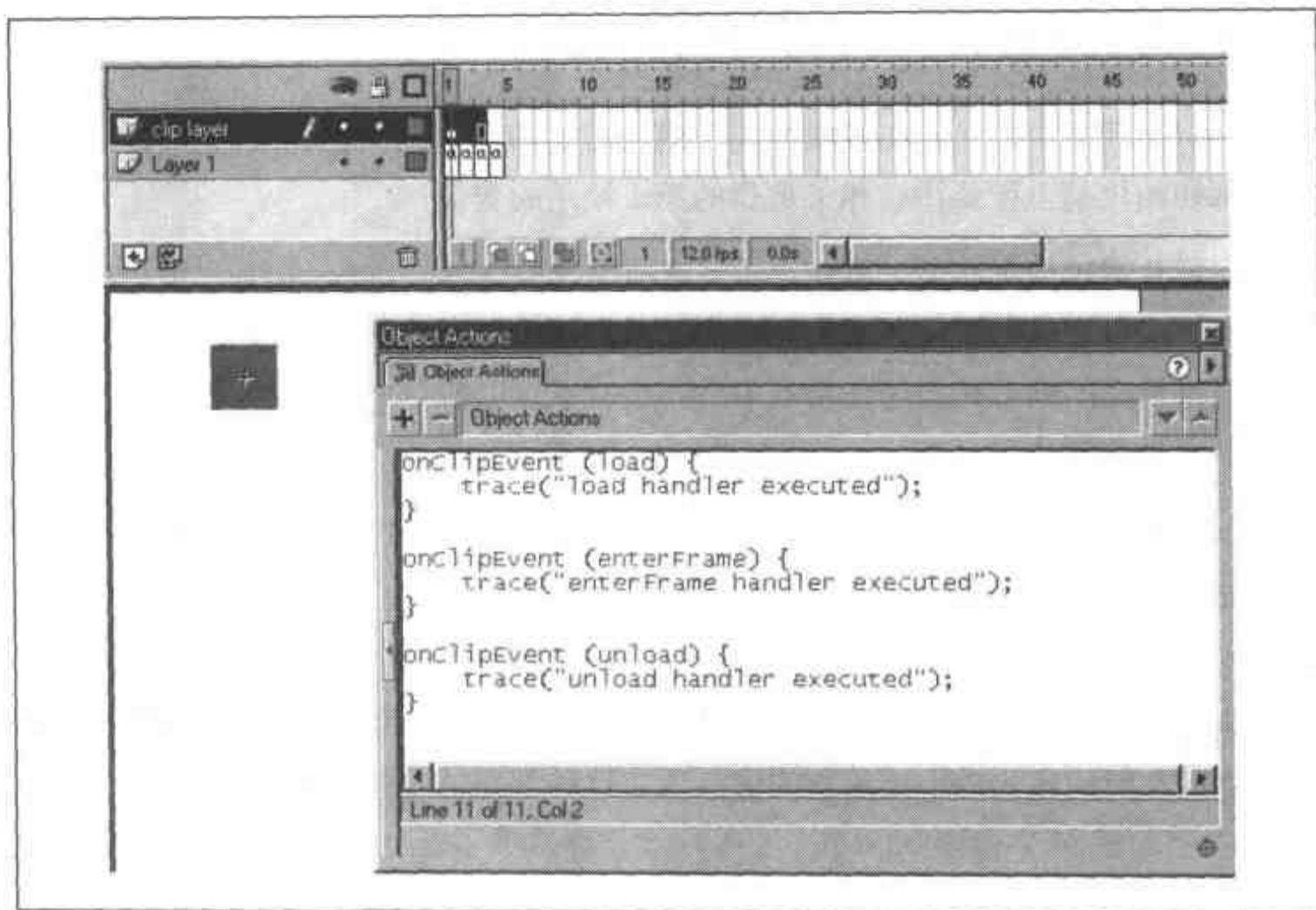


图 10-2 代码执行顺序的测试影片

当我们播放影片的时候，执行的顺序如下：

===== 第 1 帧 =====

- 1) 执行主时间线代码。
- 2) 执行 load 处理器。
- 3) 执行剪辑内的第 1 帧中的代码。

===== 第 2 帧 =====

- 1) 执行 enterFrame 处理器。
- 2) 执行剪辑内第 2 帧中的代码。
- 3) 执行主时间线代码。

===== 第 3 帧 =====

- 1) 执行 enterFrame 处理器。
- 2) 执行剪辑内第 3 帧中的代码。
- 3) 执行主时间线的代码。

===== 第 4 帧 =====

- 1) 执行 unload 处理器。
- 2) 执行主时间线的代码。

示范影片中的代码执行顺序展示了一些重要的规则，在事件处理器的编写中应记住：

- *load* 处理器中的代码在剪辑内的代码执行之前执行，因此，*load* 处理器可以用来初始化马上就要用在相关剪辑的第 1 帧中的变量。
- 在影片剪辑出现在某一帧中之前，该帧的代码会得到执行。因此，用户在影片剪辑中自定义的变量和函数在它的父时间线上的任何代码中都是无效的，直到剪辑后面的帧首次出现在场景中后才会有效，即使这些变量和函数是在剪辑的 *load* 处理器中定义的。
- *enterFrame* 事件处理器从来不与 *load* 或者 *unload* 事件占据相同的帧。*load* 和 *unload* 事件在剪辑出现或离开场景的帧上代替 *enterFrame*。
- 在每一帧中，剪辑中 *enterFrame* 处理器的代码在剪辑的父时间线代码执行前得到执行。因此使用 *enterFrame* 处理器，可以改变剪辑的父时间线的属性，然后立即在时间线代码中使用新的值，所有这些都在同一帧中完成。

## 复制剪辑事件处理器

这种快捷方法会产生一个重要的结果：当一个影片剪辑通过 *duplicateMovieClip()* 函数被复制的时候，影片剪辑事件处理器也会被复制。例如，假设在场景上有一个影片剪辑 *square*，它里面定义了一个 *load* 事件处理器：

```
onClipEvent (load) {  
    trace('movie loaded');  
}
```

当我们复制 *square* 以创建 *square2* 的时候会发生什么情况呢？

```
square.duplicateMovieClip('square2', 0);
```

因为 *load* 处理器在我们复制 *square* 的时候被复制到了 *square2* 中，因此，*square2* 的产生导致了 *load* 处理器的执行，它将在输出窗口中显示“movie loaded”。使用自动的处理器保留性能，可以创建功能强大、灵活的递归函数。例 10-2 的示例只是给出了它的简单应用。

## 用 updateAfterEvent 更新屏幕

正如我们在前面所学到的, *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* 和 *keyUp* 事件处理器在这些事件发生的时候会立刻执行。立刻的意思就是即刻——即使所讨论的事件发生在帧的渲染之间。

这种即刻性可以让影片有一个快速的反应, 但是这种反应很容易被丢失。在默认情况下, *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* 或者 *keyUp* 事件处理器的视觉效果在下一个有效的帧得到渲染之前不能被 Flash 播放器进行物理的渲染。要看到实际发生的情形, 可以创建一个单帧的影片, 帧速率为每秒一帧, 然后在场景上放置一个影片剪辑, 其中包含下面的代码:

```
onClipEvent (mouseDown) {  
    _x += 2;  
}
```

然后, 测试影片, 并且尽可能快地点击鼠标。你会看到, 所有的点击都会有结果, 但是影片剪辑还是每秒移动一下。因此, 如果你在帧之间点击 6 次, 那么剪辑在下一帧渲染之前将会向右移动 12 像素。如果你点击三次, 那么剪辑会移动 6 像素。*mouseDown* 处理器的每一次执行都记录在帧的中间, 但是结果只在每一帧得到渲染的时候才显示出来。这种方法可以在某种交互形式上制造生动活泼的效果。

幸运的是, 我们可以让 Flash 即刻渲染发生在任何用户输入事件处理器期间的任何视觉的变化, 而不用等待下一帧的出现。可以从事件处理器内使用 *updateAfterEvent()* 函数, 如下所示:

```
onClipEvent (mouseDown) {  
    _x += 2;  
    updateAfterEvent();  
}
```

*updateAfterEvent()* 函数只对 *mouseDown*, *mouseUp*, *mouseMove*, *keyDown* 和 *keyUp* 事件有效。它通常对于影片的平滑是必需的且与用户输入的相应视觉行为有关。在例 10-8 中, 我们要使用 *updateAfterEvent()* 来确保自定义指示器的平滑渲染。但是要注意, 按钮事件不要求显式的 *updateAfterEvent()* 函数调用。按钮会在帧之间自然更新。

## 代码的重复使用性

使用按钮事件和影片剪辑事件的时候，不要忘了在第九章中学习过的代码集中原则。要经常试着避免不必要的重复以及跨越多个影片元素的代码混合。如果你发现自己在多个事件处理器体内输入了相同的代码，那么直接将代码添加到对象上就是不明智的。试着集中代码，将代码从对象中拿出来，放在影片中某个代码比较集中的地方，通常，最好的地方就是主时间线。

在很多情况下，将语句隐藏在按钮或者剪辑处理器内并不是一个好的方法。记住，将代码压缩到一个函数中，然后从处理器调用这个函数，这样，代码就可以重复使用并且容易找到。按钮也是如此——我很少将函数调用语句之外的其他语句直接放到按钮上。对于影片剪辑，你要有更敏锐的判断，因为直接将代码放在剪辑上通常会产生比较好的，清楚而且完备的代码结构。你可以使用不同的方法，直到对你的需求和技能水平找到一个恰当的平衡点。这会让你对冗余和再度使用问题有较深的印象。

对于向按钮直接添加代码与从按钮调用函数之间的差别的例子，可以参见第九章。

## 动态的影片剪辑事件处理器

在本章的前边，我们已经学习了Flash中两种类型的事件——一种是输入影片剪辑和按钮的，一种是添加到其他数据对象（比如XML和XMLSocket）上的。要为数据对象创建事件处理器，可以将处理器函数的名称赋为对象的属性。回忆一下动态添加函数的语法：

```
myXMLLoader.onLoad = function () { trace('all done loading!'); };
```

动态的函数赋值可以在影片播放期间改变处理器的行为。我们需要做的所有工作就是重新设置处理器的属性：

```
myXMLLoader.onLoad = function () { gotoAndPlay("displayData"); };
```

我们甚至可以一起取消处理器：

```
myXMLLoader.onLoad = function () { return; };
```

不幸的是，影片剪辑和按钮事件的处理器并没有这么灵活，它们不能在影片播放期间被改变或者删除。而且，影片剪辑事件处理器不能添加到任何影片的主影片时间线上！为影片的 `_root` 剪辑直接创建一个事件处理器是不可能的。

为了突破这个限制，可以在 `enterFrame` 和用户输入事件情况下，用空剪辑来模仿动态的事件处理器的删除和改变。空影片剪辑甚至可以模拟 `_root` 层的事件。我们已经在第八章看到了这一技术，并学习了如何创建一个事件循环，如下所示：

1. 创建一个空影片剪辑，命名为 `process`。
2. 将另外一个名为 `eventClip` 的空剪辑放在 `process` 中。
3. 在 `eventClip` 上添加想要的事件处理器。`eventClip` 处理器中的代码应该把 `process` 剪辑的主时间线作为目标，如下所示：

```
onClipEvent (mouseMove) {  
    _parent._parent.doSomeFunction();  
}
```
4. 要导出 `process`，将它和 `attachMovie()` 函数一起使用，可以在库中选择它，然后执行 Options (选项) → Linkage (连接) 命令。对连接进行设置以导出这个符号，然后赋一个恰当的标识符（例如，“`mouseMoveProcess`”）。
5. 最后，要应用于事件处理器的时候，可以用 `attachMovie()` 将 `process` 剪辑添加到相应的时间线上。
6. 要脱离处理器，可以用 `removeMovieClip()` 来删除 `process` 剪辑。

对于如何将此技术同 `enterFrame()` 事件一起使用的详细指令，可以参见第八章。

## 事件处理器应用

我们将通过几个真实的例子来结束对 ActionScript 事件和事件处理器的研究。这些都是一些简单的应用，但是，它们会让我们体会到基于事件的编程是非常灵活的。最后两个例子可以从在线代码库中下载。

例 10-7 可以让一个剪辑产生放缩效果。

**例 10-7：影片剪辑的尺寸振荡**

```
onClipEvent (load) {  
    var shrinking = false;  
    var maxHeight = 300;  
    var minHeight = 30;  
}  
  
onClipEvent (enterFrame) {  
    if (_height < maxHeight && shrinking == false) {  
        _height += 10;  
        _width += 10;  
    } else {  
        shrinking=true;  
    }  
  
    if (shrinking == true) {  
        if (_height > minHeight) {  
            _height -= 10;  
            _width -= 10;  
        } else {  
            shrinking=false;  
            _height += 10; // 在这里进行增量就不会错过一个循环周期  
            _width += 10;  
        }  
    }  
}
```

例 10-8 模仿一个自定义的鼠标指示器，它将通常的系统指示器隐藏起来，然后制作一个剪辑，在屏幕上跟随鼠标的位置。在这个例子中，*mouseDown* 和 *mouseUp* 处理器重新设定自定义指示器的大小，以表示鼠标的点击。

**例 10-8：自定义的鼠标指示器**

```
onClipEvent (load) {  
    Mouse.hide();  
}  
  
onClipEvent (mouseMove) {  
    _x=_root._xmouse;  
    _y=_root._ymouse;  
    updateAfterEvent();  
}  
  
onClipEvent (mouseDown) {  
    _width *= .5;  
    _height *= .5;  
    updateAfterEvent();  
}  
  
onClipEvent (mouseUp) {  
    _width *= 2;
```

```
_height *= 2;
updateAfterEvent();
}
```

例10-9简单证明了ActionScript影片剪辑事件处理器的功能，它用`mouseMove`来检查图像翻转，用`mouseDown`和`mouseUp`来检查按钮点击，用`hitTest()`函数来快速记录探测结果，将一个影片剪辑转变为自定义的按钮。这个例子假设带处理器的剪辑有三个关键帧，标签分别为`up`、`down`和`over`（和一般的按钮状态对应）。

### 例 10-9：影片剪辑按钮

```
onClipEvent (load) {
    stop();
}

onClipEvent (mouseMove) {
    if (hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
        this.gotoAndStop("over");
    } else if (!hitTest(_root._xmouse, _root._ymouse, true) && !buttonDown) {
        this.gotoAndStop("up");
    }
    updateAfterEvent();
}

onClipEvent (mouseDown) {
    if (hitTest(_root._xmouse, _root._ymouse, true)) {
        buttonDown = true;
        this.gotoAndStop('down');
    }
    updateAfterEvent();
}

onClipEvent (mouseUp) {
    buttonDown = false;
    if (!hitTest(_root._xmouse, _root._ymouse, true)) {
        this.gotoAndStop('up');
    } else {
        this.gotoAndStop('over');
    }
    updateAfterEvent();
}
```

## 小结

随着对语句、操作符、函数，以及现在的事件和事件处理器的深入了解，我们已经学习了ActionScript中所有的内部工具是如何工作的。为了加深对语言的理解，在下面的三章中，我们将要学习三种非常重要的数据类型：数组、对象和影片剪辑。

---

# 第十一章

# 数组

在第三章中，我们学习了原始的数据类型——串、数字、布尔、null和undefined，它们可以表示出脚本中的基本信息。我们还知道，ActionScript 支持多种复合数据类型，它们可以将多个数据组合在一起，成为一个单独的数据。

数组类型就是我们要学习的第一种复合数据类型。数组用来存储和操作排序的信息列表，因此，对于连续的、重复的编程来说，它们是基本的工具。我们可用它们来完成从存储用户输入表单信息值，到产生下拉菜单直至追踪游戏里的敌机的所有工作。在最基本的形式里，一个数组就是一个项目列表，和你的食品列表或者支票簿账目列表一样。

## 什么是数组

一个数组就是一个数据结构，它能包含多个单独的数据值，就像一座楼房是一种物理结构，能够容纳多个层一样。和原始数据类型不同，一个数组可以包括一个以上的数据值。下面是一个简单的例子，前面两个是串，后面一个是包含两个串的一个数组：

```
"oranges"          // 一个原始串值  
'apples'          // 另外一个原始串值  
["oranges", 'apples'] // 包含两个串的一个数组
```

数组是一个有多种用途的容器。它可以包含任意数量的项目，甚至不同类型的项目。

下面给出了一个简单的例子，其中的数组同时包含串和数字。它表示的是一个购物单，以及你要买的每种东西的数量：

```
["oranges", 6, "apples", 4, "bananas", 3]
```

下面我们再给出一个关于数组概念的例子，可能更切合实际一点：一个带抽屉的柜子。一个单独的抽屉可以包含一些东西（袜子、衬衫、等等），但是柜子本身并不包含什么东西，它包含的是抽屉，抽屉再包含某些东西。柜子将抽屉组合在一起，成为独立的一个单位。

当我们面对一个数组（带抽屉的柜子）的时候，通常对数组中的值（抽屉里的东西）感兴趣。数组中包含的值就是我们想要处理的信息。但是，我们也可以对包含结构本身进行操作。例如，我们可以改变一个数组的大小（添加或者减少抽屉）或者对它的值重新排序（交换抽屉里的东西）。

虽然我们说一个数组包含很多值，但是，记住数组本身是一个单独的数据是非常重要的，它就像一个串包含多个字符、但仍然是单独的一个串，数字包含多个数位但仍然是一个单独的数字一样。作为一个单独的数据，数组可以被赋值给一个变量，或者用来作为复杂表达式的一部分：

```
product = ["ladies downhill skis", 475]      // 将数组存储在变量中
display(product);                            // 将数组传递给函数
```

## 数组的分析

存储在数组中的每一个项目被称为一个数组元素，它们都有各自的号码（索引），很容易找到。

## 数组元素

和变量一样，每一个数组元素可以存储任何合法的数据。一个完整的数组类似于一些连续命名变量的集合，但是，每一个项目都有一个元素号码（第一个元素为号码0，而不是号码1）而没有不同的名称。

图 11-1 从概念上显示了数组的结构，它包含三个元素。元素 0 存储的值为“Erica”，元素 1 存储的值为“Slavik”，而元素 2 存储的值则为“Gary”。

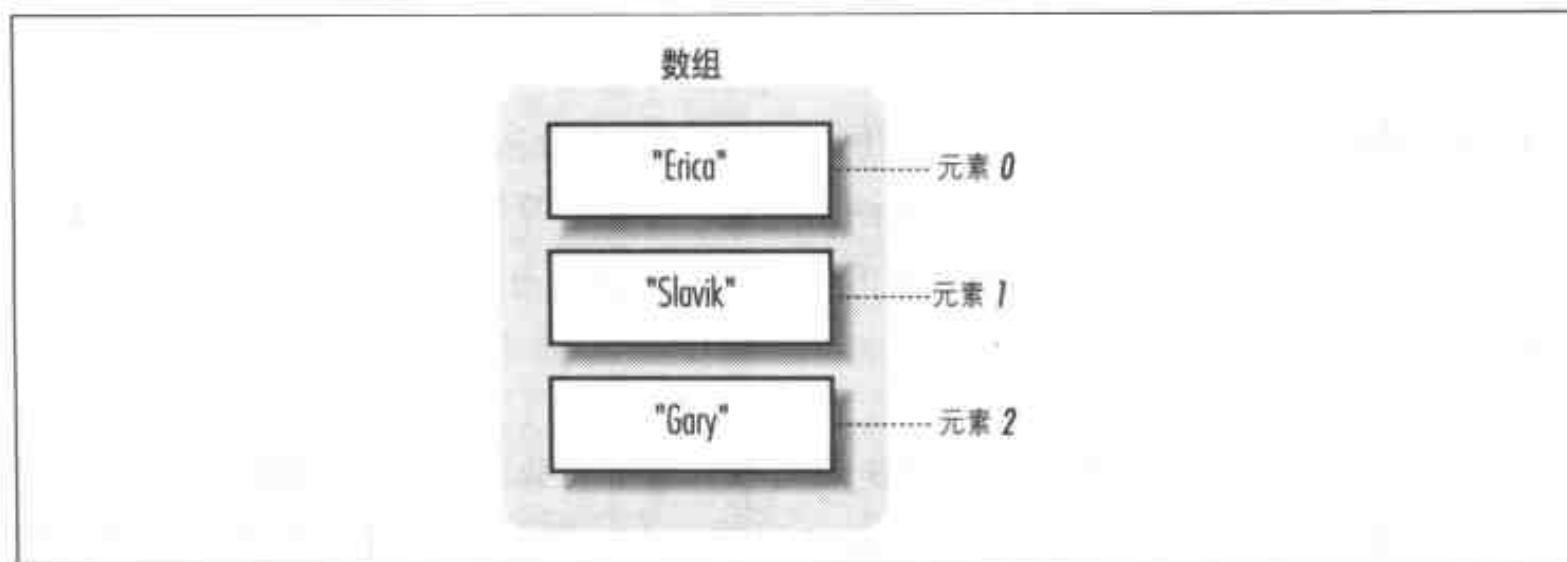


图 11-1 一个简单的数组结构

要操作数组元素中的值，可以用号码来获得元素。与我们的柜子类似，我们可以让 ActionScript 将某些东西存储在第一个抽屉里，或者获得第二个抽屉中的东西。

## 数组元素索引

数组中元素的位置被称为它的索引 (index)。正如我们可以访问一个串中的第七个字符一样，我们同样也可以访问数组中的第七个元素，通过它的索引（在本例中，也就是索引 6）就可以实现。我们使用元素的索引来设置或者获取元素的值，或者其他某种方式处理元素。例如，有一些数组处理函数使用元素索引来指定要处理的元素范围。

我们也可以在数组的开头、结尾或中部位置插入或者删除元素。数组可以有空隙(也就是说，有的元素不在了)。我们可以让元素在位置 0 和 4 上，而位置 1, 2 和 3 可以空着。带空隙的数组被称为稀疏数组 (sparse array)。

## 数组的大小

每个数组在生命周期中任何给定的时间点上都包含一定数量的元素。数组能容纳的潜在元素的数量被称为数组的长度，我们将在后面讨论。

## 数组的创建

我们可以创建一个带数据直接量的新数组(也就是简单地写出所有的元素),也可以使用特定的内置数组构造函数 *Array()*。

### 数组构造器

要用 *Array()* 构造器创建一个数组, 我们使用 *new* 操作符, 后面跟着 *Array* 关键字, 然后是括号, 它可以产生一个空的数组(没有任何元素)。通常将新创建的数据赋给某个变量或者其他数据容器, 以便将来使用。例如:

```
var myList = new Array(); // 在变量 myList 中存储一个空的数组
```

想要将单个的值赋给数组的元素, 可以通过在调用数组的时候将参数传递给 *Array()* 构造器而实现。根据我们提供的参数, 构造器的调用会产生不同的效果。

当对 *Array()* 构造器提供一个以上的参数, 或者一个非数字的参数时, 每个参数都会成为新数组中的一个元素值。例如:

```
var frameLabels = new Array("intro", "section1", "section2", "home");
```

存储在 *frameLabels* 中的数组会有下面的元素:

```
0: 'intro'  
1: 'section1'  
2: 'section2'  
3: "home"
```

当向 *Array()* 构造器提供一个精确的数字参数时, 它所创建的数组将带有指定数目的空占位符元素:

```
var myList = new Array(14); // 创建一个有 14 个空元素的数组
```

传递给 *Array()* 构造器的参数可以是任何合法的表达式, 包括复合表达式:

```
var x = 10;  
var y = 5;  
var myNumbers = new Array(x + 1, x * y, Math.random());
```

*myNumbers* 变量将存储一个有下面元素的数组中:

## 其他编程语言中的数组

几乎每一种高级计算机语言都支持某种类型的数组或者类似数组的实体。也就是说，在不同的语言中，数组的提供方式有所不同。例如，许多语言不允许数组包含不同类型的元素，数组可以包含串或者数字，但是同一个串中不允许同时包含这两种数据类型。更为有趣的是，在C语言中，并没有原始的串数据类型，但却有一个单字符数据类型，称为*char*，串被认为是一种复杂的数据类型，作为*char*的数组来对待！

在ActionScript中，数组的大小可以在项目被添加或者被删除的时候自动改变。在许多语言中，数组的大小必须在数组首次定义或者规定大小（也就是存储器被分配来存储数组数据）的时候予以指定。Lingo是一种针对Macromedia Director的脚本语言，它通过名称列表来指向它的数组。和ActionScript一样，Lingo允许数组包含不同类型的数据值，它也会在必要的时候自动调整数组的大小。与ActionScript和C语言不同的是，Lingo列表的第一个项目编号为1（也就是与1相对），而不是编号为0（也就是与0相对）。

语言的不同，导致在数组范围（限制）之外访问一个元素的时候发生的情况也不同。如果试图在数组现存的范围之外为一个元素赋值，那么ActionScript和Lingo会为你添加元素。如果试图在数组范围之外访问某个元素，ActionScript会返回*undefined*，而在Lingo中这会产生错误。C语言不关心你访问的是否为有效元素号码。它可以让你在数组的范围之外获取或者设置元素，这通常会改写存储器中的数据，或者访问数组之外的其他无意义的数据（C给你了足够长的绳索，让你把自己吊起来）。

```
0: 11  
1: 50  
2: 一个0到1之间的浮点数字
```

## 数组直接量

有的时候，用数组直接量来创建数组比用*Array()*构造器来创建数组要方便许多。回忆一下，我们说过，一个直接量就是一个确定数据的直接表示。例如：

```
"beaver" // 一个串直接量
```

```
234.2034          // 一个数字直接量  
true              // 一个布尔直接量
```

在一个数组直接量中，方括号表示数组的开始和结束。在方括号中，以逗号分隔的表达式列表提供了数组元素的值。下面是一般语法：

```
[expression1, expression2, expression3]
```

表达式被求值，然后作为所描述数组的元素存储起来。任何合法的表达式都可以使用，包括函数调用、变量、直接量，甚至可以是其他数组（一个数组在另外一个数组内称为嵌套数组）。下面是几个简单的例子：

```
[4, 5, 63]           // 简单的编号元素  
["jeremy", "janice", "eman"] // 简单的串元素  
[1, 4, 6+10]        // 带运算的数字表达式  
[firstName, lastName, "tall", "skinny"] // 变量和串作为元素  
['month end days', [31, 30, 28]]    // 嵌套的数组直接量
```

比较之下，我们可以对 *Array()* 构造器做相同的工作：

```
new Array(4, 5, 63)  
new Array('jeremy', 'janice', 'eman')  
new Array(1, 4, 6 - 10)  
new Array(firstName, lastName, "tall", "skinny")  
new Array('month end days', new Array(31, 30, 28))
```

注意，ActionScript 中单个数组的元素可以包含不同类型的数据，就如在前面提到过的一样。

## 引用数组元素

一旦创建好了数组，就不可避免地要获取或者修改元素的值。要这么做，可以使用方括号 []（也就是数组访问操作符），在第五章中已经介绍过。

### 获取元素的值

为了获得元素的值，只需在方括号中使用它的索引，就可以指向某个元素了，如下所示：

```
arrayName[elementNumber]
```

`arrayName` 必须是数组, `elementNumber` 可以是任何能产生数字值的表达式。第一个元素为号码 0, 最后一个元素的号码是数组的长度减去 1。如果指定的元素号码比最后一个有效元素号码大, 解释程序就会返回 `undefined`。例如:

```
// 用数组常量创建一个数组, 将它存储在 trees 中
var trees = ["birch", "maple", "oak", "cedar"];

// 在输出窗口中显示 trees 的第一个元素
trace(trees[0]); // 显示: "birch"

// 将第三个元素的值赋给变量 favoriteTree
// (记住, 索引是从 0 开始的, 因此索引 2 就是第三个元素!!)
var favoriteTree = trees[2]; // favoriteTree 成为了 "oak"
```

现在我们来看比较有趣的部分。由于我们可以用任何可以产生数字的表达式作为元素的索引, 因此, 可以使用变量来指定元素的索引, 就如前面我们用数字来做索引一样。例如:

```
var i = 3;
var lastTree = trees[i]; // 将 lastTree 设置为 "cedar"
```

我们甚至可以使用有数字返回值的函数调用表达式来作为数组的索引:

```
// 计算一个 0 到 3 之间的随机数, 将 randomTree 设置为从 trees 中任意选取的元素
var randomTree = trees[Math.floor(Math.random() * 4)];
```

太棒了, 真不错! 你可以使用类似的方法从一组琐碎的问题中随机抽取一个问题, 或者从表示一副扑克牌的数组中随机抽一张。

注意, 访问一个数组和访问变量的值是相似的。数组元素可以被用作复杂表达式的一部分, 如下所示:

```
var myNums = [12, 4, 155, 90];
var myTotal = myNums[0] + myNums[1] + myNums[2] + myNums[3]; // 计算数组的总和
```

这个例子中用来对数组元素值求和的方法并不是最优代码。我们在后面会用一个更快更方便的方法, 来连续地访问一个数组的元素。

## 设置元素的值

要设置一个元素的值, 我们使用 `arrayName[elementNumber]` 作为赋值表达式左边的操作数:

```
// 建立一个数组
var cities = ["Toronto", "Montreal", "Vancouver", "Waterloo"];
// cities 现在是: ['Toronto', 'Montreal', 'Vancouver', 'Waterloo']

// 对数组的第一个元素设值,
// 使 cities 变成: ["London", "Montreal", "Vancouver", "Waterloo"]
cities[0] = 'London';

// 设置数组中第四个元素的值
// cities 现在变成: ["London", "Montreal", "Vancouver", "Hamburg"]
cities[3] = "Hamburg";

// 设置数组中第三个元素的值
// cities 现在变成: ["London", "Montreal", 293.3, "Hamburg"]
cities[2] = 293.3; // 我们可以看到数据类型转换不是一件难事
```

注意，我们在设置一个数组元素的时候可以用任何数字表达式来作为索引：

```
var i = 1;
// 设置元素 i 的值
// cities 变成: ["London", "Prague", 293.3, "Hamburg"]
cities[i] = "Prague";
```

## 确定数组的大小

所有的数组都有一个内置的属性 `length`，它表示当前的元素数量（包括空的元素）。要访问数组的 `length` 属性，可以使用点操作符，如下所示：

```
arrayName.length
```

数组的 `length` 属性告诉我们数组中有多少个元素。下面是一些例子：

```
myList = [34, 45, 57];
trace(myList.length);           // 显示: 3

myWords = ["this", "that", "the other"];
trace(myWords.length);         // 显示: 3. 元素的数目，而不是词或字符的数目
frameLabels = new Array(24);    // 注意：单个的数字参数和 Array() 构造器一起使用
trace(frameLabels.length);      // 显示: 24
```

数组的 `length` 总是比它最后一个元素的索引大。例如，一个数组的元素索引如果为 0, 1, 2，那么它的长度就是 3。一个数组的元素索引如果为 0, 1, 2, 50，那么它的长度就是 51。51？对，就是 51。即使从索引 3 到索引 49 都是空的，它们也仍

然要为数组的长度作贡献。数组的最后一个元素总是`myArray[myArray.length-1]`，因为索引数字是从0开始的，而不是从1开始。

如果添加或者删除元素，数组的`length`属性就会改变，以适应我们的修改。实际上，我们甚至可以通过设置`length`属性来增加或者删除数组末尾的元素。

数组的`length`属性究竟有什么用途？使用数组的`length`属性，可以创建一个循环来访问数组中所有的元素，就如我们在例8-1中所看到的那样。循环扫描数组中的元素是编程的一个基本任务。当将数组和循环结合起来的时候，你如果想知道会发生什么情况，可以参照例11-1，它搜索`soundtracks`数组，以寻找值为“hip hop”的元素。你可以看到在第八章中学到的`for`循环，以及第五章中学到的增量操作符。至于数组的访问代码，你刚刚学过。

### 例 11-1：数组的搜索

```
// 创建一个数组
var soundtracks = ["electronic", "hip hop", "pop", "alternative", "classical"];

// 检查每一个元素，寻找包含 "hip hop" 值的元素
for (var i = 0; i < soundtracks.length; i++) {
    trace('Now examining element: '+i);
    if(soundtracks[i] == "hip hop") {
        trace('The location of "hip hop" is index: '+i);
        break;
    }
}
```

让我们来将例11-1扩展为一个具有普遍性的搜索函数，可以在任何数组中查找任何匹配元素，如例11-2所示。我们的搜索函数返回数组内找到元素的位置，如果没有发现就返回`null`。

### 例 11-2：一个扩展的数组搜索函数

```
function searchArray (whichArray, searchElement) {
    // 检查所有的元素是否包含 searchElement
    for (var i = 0; i < whichArray.length; i++) {
        if (whichArray[i] == searchElement) {
            return i;
        }
    }
    return null;
}
```

下面给出的例子将告诉你如何使用新搜索函数来检查“Fritz”是否是userNames数组中的一个元素，该数组包含的是一些经过认可的用户名：

```
if (searchArray (userNames, "Fritz") == null) {
    trace ('Sorry, that username wasn't found');
} else {
    trace ("Welcome to the game.");
}
```

现在我们振作一点！我们所有的努力工作将要集中到一个单独的系统中，是得到回报的时候了。从表面上看，微不足道的单个运算可以结合起来，形成非常强大、非常灵活的程序。就像字母表中的字母，或者一条DNA中的氨基酸一样，你可以用手上的单个积木构造出所有可以想像的东西。本章剩余的部分将介绍更多有关数组操作结构方面的内容，包括执行一些普通功能的内置函数的用法。如果你的需要不能通过内置函数来完成，可以编写你自己定义的函数，如前面例子中的*searchArray()* 函数。

## 命名数组元素

元素通常都是计数的，不过也可以命名。使用命名元素，我们可以仿效所谓的相联矩阵或者哈希。注意，命名数组元素不能用*Array*方法 (*push()*, *pop()* 等等，这些在后面将会讨论) 来操作，它们也不是编号元素列表的一部分。带一个命名元素和两个编号元素的数组，其 *length* 为 2，而不是 3。要访问数组中所有的命名元素，必须使用*for-in*循环(在第八章已讨论过)，它会将命名元素和编号元素都列举出来。

## 创建和索引命名数组元素

要添加一个可以用名称来索引的元素，我们在现有的数组上同样使用方括号，只不过带的是串，而不是数字：

```
arrayName[elementName] = expression
```

*elementName* 是一个串。例如：

```
var importantDates = new Array();
importantDates['dadsBirthday'] = "June 1";
importantDates["mumsBirthday"] = "January 16";
```

我们也可以使用点操作符，如下所示：

```
arrayName.elementName = expression
```

在这个例子中，`elementName`必须是一个标识符，而不是一个串。例如：

```
var importantDates = new Array();
importantDates.dadsBirthday = "June 1";
importantDates.mumsBirthday = "January 16";
```

假设我们知道一个元素的标识符（例如，`dadsBirthday`在`importantDates`数组中），可以有两种方法来访问它：

```
var goShopping = importantDates["dad'sBirthday"];
var goShopping = importantDates.dad'sBirthday;
```

和对命名元素的赋值一样，当用方括号访问一个元素的时候，`elementName`必须是一个串或者是一个可以产生串的表达式。同样，和点操作符一起使用时，`elementName`必须是一个标识符（也就是没有括号的元素的名字），而不是一个串。

## 删除命名元素

要删除数组中一个不再需要的命名元素，我们使用`delete`操作符，它在第五章中已经介绍过了：

```
delete arrayName.elementName
```

删除一个命名元素会同时破坏元素的值以及元素的容器，释放被元素和其内容所占据的内存空间。（它和`delete`操作符对编号元素的处理效果不同，后者仅仅清除元素的值，但是保留容器。）

## 向数组添加元素

要为数组添加一个新的元素，就要为新的元素赋值，增长数组的`length`属性，或者使用某个内置数组函数。

## 直接添加新的元素

我们可以在现有数组的指定索引中添加一个新的元素，只要简单地对该元素赋值就可以了：

```
// 创建一个数组、为它赋三个值  
var myList = ['apples', 'oranges', 'pears'];  
  
// 添加第四个值  
myList[3] = 'tangerines';
```

新的元素并不需要紧跟在原数组的最后一个元素后面。如果我们将新的元素放在数组末尾的多个元素之后，ActionScript 会自动在中间创建空元素：

```
// 从索引 4 到 38 都是空的  
myList[39] = 'grapes';  
  
trace (myList[12]);           // 显示为空，因为元素 12 是 undefined
```

## 用 length 属性添加新的元素

要扩展一个数组，而不对新的元素赋值，可以只简单地增长 `length` 属性，ActionScript 将添加足够的元素来达到这个长度：

```
// 创建一个有三个元素的数组  
var myColors = ['green', 'red', 'blue'];  
  
// 为数组添加 47 个空元素，从 3 到 49  
myColors.length = 50;
```

可以用这个方法来创建一定数目的空元素，以保存一些你以后要放的数据，比如学生的考试成绩。

## 用 Array 方法添加新的元素

我们可以使用内置 `array` 方法来进行更复杂的元素添加工作。(我们将在第十二章中学习，所谓方法就是能在对象上操作的函数。)

### push()方法

`push()`方法在数组的末尾添加一个或者多个元素。它会在数组的最后一个元素后面

自动添加数据，因此，我们不需要担心已经存在的元素有多少。*push()*方法也可以一次对数组添加多个元素。要在数组上调用*push()*，我们使用数组名称，后面跟点操作符，以及关键字*push*，括号中可以有多个参数，也可以没有参数：

```
arrayName.push(item1, item2, ...itemn);
```

*item1, item2, ...itemn*是由逗号分隔的项目列表，它们会被添加在数组的末尾，作为新的元素。下面是几个例子：

```
// 创建一个带两个元素的数组  
var menuItems = ['home', 'quit'];  
  
// 添加一个元素  
// menuItems 成为 ["home", "quit", "products"]  
menuItems.push('products');  
  
// 再添加两个元素  
// menuItems 成为 ["home", "quit", "products", "services", "contact"]  
menuItems.push("services", "contact");
```

调用的时候如果不传递参数，*push()*就不添加任何元素：

```
menuItems.push(); // 将数组长度增加1  
  
// 同样的结果  
menuItems.length++;
```

*push()*方法返回更新后数组的长度：

```
var myList = [12, 23, 98];  
trace(myList.push(28, 36)); // 对myList添加28和36，显示：5
```

注意，添加到列表的项目可以是任何表达式。表达式在被添加到列表之前要求值：

```
var temperature = 22;  
var sky = "sunny";  
var weatherListing = new Array();  
weatherListing.push(temperature, sky);  
trace(weatherListing); // 显示："22,sunny"，而不是 "temperature, sky"
```

## unshift()方法

*unshift()*方法和*push()*很相似，但是，它是在数组的开头添加一个或多个元素，顺延所有现有元素(也就是说，现有元素的索引都会增加，以对应数组开头的新元素)。*unshift()*的语法和所有的*array*方法一样，使用下面的格式：

```
arrayName.unshift(item1, item2,...itemn);
```

item1, item2, ... itemn是由逗号分隔的项目列表，要作为新的元素添加到数组的开头。注意，多个项目的添加顺序和提供它们的顺序相同。下面是一些例子：

```
var flashVersions = new Array();
flashVersions[0] = 5;
flashVersions.unshift(4);      // flashVersions 现在为[4,5]
flashVersions.unshift(2, 3);   // flashVersions 现在为[2,3,4,5]
```

*unshift()*方法和*push()*方法相似，返回增大后数组的长度。

## push, pop 和栈

*push()*方法的名字从一种叫做栈的编程概念而来。一个栈可以被看做一个垂直的数组，就像一叠盘子。如果你经常出入餐馆，看见里面的碗柜，你可能会对架子上放着的餐盘比较熟悉。要再放一个干净的盘子的时候，它们通常都被放在盘堆的顶层，原来盘子的次序就依次往下降。当一个消费者要从堆中拿一个盘子的时候，他会拿走最近放上去的盘子。这被称为后进先出（LIFO）栈，历史列表即是其典型的实例。例如，如果你在浏览器中点击了 Back 键，它会让你重新进入你最近访问过的上一个网页。如果你再次点击 Back 键，你会被带到更前面的一页，依此类推。它是通过将每个你访问的页面推入（push）栈中，然后在点击 Back 的时候弹出（pop）来实现的。

LIFO 栈也可以在现实生活中看到。飞机场里最后检查行李的人总是在飞机到达后第一个拿到他的行李，因为行李的装载顺序和卸下顺序正相反。第一个检查行李的人在飞机到达之后，会在传送带旁边等得最久，真是很倒霉。一个先进先出栈（FIFO）要公平一些——它以先来先服务为工作基础。FIFO 栈就像银行里的队列。FIFO 栈不拿数组中最后一个元素，而是每次都处理数组中的第一个元素。然后，它删除数组中的第一个元素，所有其他元素就要往上移动，就像排在你前面的人走掉以后你就可以往前边移动一样（也就是说，她得到服务后走了，或者她等累了选择离开）。因此，词 *push* 通常表示你使用的是 LIFO 栈，而 *append*（附加）则表示你使用的是 FIFO 栈。不管在什么情况下，元素都被添加到栈的“末尾”，不同之处在于，下次操作要从栈的哪一头开始。

## splice()方法

*splice()*方法可以从数据中添加或者删除元素。它的典型使用是将元素插入数组的中部（后面的元素要重新编排号码）或者从数组的中部删除元素（后面的元素要重新编排号码以弥补间隙）。当 *splice()* 在一次调用中同时执行这两项任务时，它可以有效地用新元素来代替一些原来的元素（虽然不一定是相同号码的元素）。下面是 *splice()* 的语法：

```
arrayName.splice(startIndex, deleteCount, item1, item2, ...itemn)
```

*startIndex* 是一个数字，表示要删除或者选择插入元素的开始（记住，第一个元素的索引为 0）；*deleteCount* 是可选的参数，表示要删除多少元素（包括 *startIndex* 表示的元素）。当 *deleteCount* 被省略时，*startIndex* 以及其后的元素都将被删除。可选参数 *item1, item2, ... itemn* 是由逗号分隔的项目列表，它们将作为以 *startIndex* 开始的元素被添加到数组中。

例 11-3 显示了 *splice()* 方法的多功能性。

### 例 11-3：使用 *splice()* 数组方法

```
// 建立一个数组...
months = new Array("January", "Friday", "April", "May", "Sunday", "Monday", "July");
// 哇。数组里有什么东西错了。我们来修补一下。
// 首先，我们删除 'Friday'
months.splice(1, 1);
// months 现在是: ["January", "April", "May", "Sunday", "Monday", "July"]

// 现在，我们在 April 前面添加两个月份。
// 注意，我们不想删除任何东西 (deleteCount 为 0)。
months.splice(1, 0, "February", "March");
// months 现在就是:
// ["January", "February", "March", "April", "May", "Sunday", "Monday", "July"]

// 最后，我们删除 "Sunday" 和 "Monday"，同时插入 "June"
months.splice(5, 2, "June");
// months 现在就是:
// ["January", "February", "March", "April", "May", "June", "July"]

// 现在我们的 months 数组修补好了，我们来整理一下。
// 让它只包含一年中的第一个季度
// 删除从索引 3 开始的所有元素（也就是从 "April" 开始的元素）
months.splice(3); // months 现在是: ["January", "February", "March"]
```

*splice()* 的另一个有用的功能是，它返回所删除的元素构成的数组。因此，它可以用来自从数组中摘取一系列的元素：

```
myList = ['a', 'b', 'c', 'd'];
trace(myList.splice(1, 2));      // 显示: "b, c"
                                // myList现在为['a', 'd']
```

如果没有元素被删除，*splice()*就返回一个空的数组。

## concat()方法

和 *push()*一样，*concat()*添加元素到数组的末尾。和 *push()*不同的是，*concat()*不修改调用它的数组——只返回一个新的数组。而且，*concat()*可以将作为参数传递进来的数组分为单个的元素，也可将两个数组合并为一个单独的新数组。下面是 *concat()*的语法：

```
origArray.concat(elementList)
```

*concat()*方法将包含在 *elementList* 内的元素一个接一个地添加到 *origArray* 的末尾，并将新的数组作为结果返回，而保持 *origArray* 的原状。通常，我们会将返回的数组存储到一个变量里。下面的例子中，我们将数字作为项目添加到数组：

```
var list1 = new Array(11, 12, 13);
var list2 = list1.concat(14, 15); // list2 变成[11, 12, 13, 14, 15]
```

在这个例子中，我们用 *concat()* 来合并两个数组

```
var guests = ["Panda", "Dave"];
var registeredPlayers = ["Gray", "Doomtrooper", "TRK9"];
var allUsers = registeredPlayers.concat(guests);
// allUsers 现在是: ["Gray", "Doomtrooper", "TRK9", "Panda", "Dave"]
```

注意，*concat()* 将 *guests* 数组的元素分开所采用的方法与 *push()* 不同。如果我们使用这样的代码：

```
var allUsers = registeredPlayers.push(guests);
```

我们就完成了这个嵌套数组：

```
["Gray", "Shift", "TRK9", ["Panda", "Dave"]]
```

还有，*push()* 会改变 *registeredPlayers* 数组，而 *concat()* 不会。

但是要注意，*concat()* 不会将嵌套数组分开（原来的数组保持在主数组中），如下面的代码所示：

```
var x = [1, 2, 3];
var y = [[5, 6], [7, 8]];
var z = x.concat(y);      // 结果是[1, 2, 3, [5, 6], [7, 8]]
                        // y的元素0和1没有被分开
```

## 删除数组中的元素

可以使用 *delete* 操作符从数组中删除元素，也可以通过减小数组的 *length* 属性来实现，或者使用一个内置的 *array* 方法。

### 用 *delete* 操作符删除元素

*delete* 操作符可将任何数组元素设置为 *undefined*，使用下面的格式：

```
delete arrayName[index]
```

*arrayName* 可以是任何数组，*index* 是我们要设置为 *undefined* 的元素的号码或者名称。名字 *delete* 其实是一种误导，它并不会从数组中删除某个元素，它只是将目标元素的值设置为 *undefined*。因此，*delete* 操作符也同样只是将元素值赋值为 *undefined*。我们可以在删除其中某个元素后检查数组的 *length* 属性，来核实这种情况：

```
var myList = ["a", "b", "c"];
trace(myList.length);      // 显示: 3
delete myList[2];
trace(myList.length);      // 仍然显示3...索引2的元素是 undefined 而不是 "c"
                          // 但它毕竟是存在的
```

要真的删除元素，可使用 *splice()*（从数组的中间删除它们），或者使用 *shift()* 和 *pop()*（从数组的开头和结尾删除）。注意，*delete* 对对象属性和命名元素的作用比对编号元素的作用更强。对它们使用 *delete*，会永久地破坏属性和命名元素，从而无法找到它们。

### 用 *length* 属性来删除元素

我们在前面使用 *length* 属性为数组添加元素。我们也可以将数组的 *length* 属性设置为一个比当前长度小的数字，这样就可以从数组中删除元素（也就是截短一个数组）：

```
var toppings = ["pepperoni", "tomatoes", "cheese", "green pepper", "broccoli"];
toppings.length = 3;
trace(toppings); // 显示: "pepperoni,tomatoes,cheese"
                  // 我们去掉元素 3 和 4 (最后两个)
```

## 用 Array 方法来删除元素

对于数组，有多种内置方法可以删除元素。我们已经看到 *splice()* 是如何从数组的中部删除一系列元素的。*pop()* 和 *shift()* 方法用来从数组的开头或结尾删除元素。

### *pop()*方法

*pop()* 和 *push()* 相反 —— 它删除数组的最后一个元素。*pop()* 的语法很简单：

```
arrayName.pop()
```

(我不知道是什么，但是我总是觉得“弹出”一个数组有点好笑。) 不管怎么说，*pop()* 将数组的 *length* 减 1，然后返回它所删除的元素值。例如：

```
x = [56, 57, 58];
x.pop(); // x 现在是 [56, 57]
```

正如我们在前面所学到的，*pop()* 通常和 *push()* 结合起来，执行 LIFO 栈操作。在例 11-4 中，我们用 *siteHistory* 数组来追踪用户在站点中的导航情况。当用户到了一个新的框架时，我们将他的位置用 *push()* 添加到数组。当用户返回的时候，我们用 *pop()* 方法删去他在 *siteHistory* 中的最后一个位置，并且将他送到前一个位置。例 11-4 可以从在线代码库中下载。

### 例 11-4：Back 按钮的历史记录

```
// 影片第1帧中的代码
stop();
var siteHistory = new Array();
function goto(theLabel) {
    // 如果我们已经在被请求的框架中了...
    if(theLabel != siteHistory[siteHistory.length - 1]) {
        // ... 将请求添加到历史记录，然后进入请求框架
        siteHistory.push(theLabel);
        gotoAndStop(siteHistory[siteHistory.length - 1]);
    }
    trace(siteHistory);
}
```

```
function goBack() {
    // 删除历史记录中的最后一个项目
    siteHistory.pop();
    // 如果历史记录中什么也没有了...
    if (siteHistory.length > 0) {
        // ... 到最近的框架
        gotoAndStop(siteHistory[siteHistory.length - 1]);
    } else {
        // ... 否则就回到主页
        gotoAndStop("home");
    }
    trace(siteHistory);
}

// 导航按钮上的代码
on (release) {
    goto("gallery");
}

// Back按钮上的代码
on (release) {
    goBack();
}
```

## shift()方法

*unshift()*方法用来在数组开头添加元素。它的搭档 *shift()* 用来从数组开头处删除元素：

```
arrayName.shift()
```

它没有 *pop* 那么有趣。

和 *pop()* 相似，*shift()* 返回它所删除的元素的值。剩下的元素都向前移动。例如：

```
var sports = ["hackey sack", "snowboarding", "inline skating"];
sports.shift();      // 现在["snowboarding", 'inline skating']
sports.shift();      // 现在['inline skating']
```

因为 *shift()* 实际上删除了一个元素，用它删除数组中的第一个元素比用 *delete* 更为有效。我们也可以用 *shift()* 来限制一个列表的范围。例如，假设我们在计算影片的帧速率，在每帧渲染之后，我们用 *push()* 方法将当前的时间加到一个数组中。为了限制数组的大小，只能保持最近的 10 次时间抽样，我们需要将最远的时间用 *shift()* 方法删除出去。要得到帧速率，我们将数组的时间进行平均。例 11-5 显示了这个技术。

### 例 11-5：计算影片的帧速率

```
// 在影片剪辑中创建时间计量数组
onClipEvent(load) {
    var elapsedTime = new Array();
}

// 用一个enterFrame 剪辑事件在每一帧之后测量时间
onClipEvent(enterFrame) {
    // 将当前的时间加到elapsedTime
    elapsedTime.push(getTimer());

    // 如果我们有足够的抽样来计算平均值……
    if(elapsedTime.length > 10) {
        // ...将最远的时间从elapsedTime中删除
        elapsedTime.shift();

        // 将每帧所用的毫秒数进行平均
        elapsedAverage = (elapsedTime[elapsedTime.length - 1] -
                           elapsedTime[0]) / elpasedTime.length;

        // 要得到帧速率，用1秒来除以时间平均值
        fps = 1000 / elapsedAverage;
        trace("current fps " + fps);
    }
}
```

### splice()方法

我们在前面已经学过，*splice()*可以从数组中删除和添加元素。由于我们已经了解了*splice()*的具体细节，这里就不再重复了。但是，要说明当前的内容，我们要特别给出*splice()*对元素的删除能力的范例：

```
var x = ["a", "b", "c", "d", "e", "f"];
x.splice(1,3);      // 删除元素1, 2 和 3, 留下["a", "e", "f"]
x.splice(1);       // 删除从1开始的元素, 只留下["a"]
```

## 通用数组操作工具

我们在前面已经看到了数组方法是如何用来从数组中删除和添加元素的。ActionScript 还提供了一些内置方法对元素进行重新排序以及分类，将数组元素转换为串，以及从其他数组中摘取数组。

## reverse()方法

顾名思义，*reverse()*方法将颠倒数组中元素的顺序。既简单又有效。它的语法为：

```
arrayName.reverse()
```

下面给出它的运用实例：

```
var x = [1, 2, 3, 4];
x.reverse();
trace(x); // 显示: "4,3,2,1"
```

我们通常用*reverse()*来对一个排序数组进行重新排序。例如，如果有一个产品列表，是按照价格从低到高来排序的，我们就可以按照从最便宜到最贵的顺序来显示它们，或者可以反转列表，按照从最贵的到最便宜的顺序来显示。

读者练习：编写你自定义的函数来反转数组中的元素。它不仅没有看上去那么简单，而且你很可能你会发现，内置的*reverse()*方法要快得多。

## sort()方法

*sort()*方法按照我们提供的任意规则重新安排数组元素的顺序。如果不提供某种规则，*sort()*就按照字母表的顺序进行重新排序。按照字母表的顺序对数组进行排序很简单，我们首先来看看它是如何工作的：

```
arrayName.sort()
```

当调用一个数组的*sort()*方法而不给出参数的时候，它的元素就临时转换为串，按照附录二所给出的编码点来进行排序（参见第四章内容）：

```
// 这些代码按照预想的效果来运行
var animals = ['zebra', 'ape'];
animals.sort();
trace(animals); // 显示: "ape, zebra"
                // 酷！多方便的一个小方法

// 当心，排序顺序不是严格的字母表顺序...
// zebra中的大写字母'Z'在"ape"中的小写'a"之前
var animals = ["Zebra", "ape"];
animals.sort();
trace(animals); // 显示: "Zebra, ape"。参见附录二
```

我们也可以使用 `sort()` 按照自己选择的规则对元素进行排列。这种技术稍有难度，但是却更有用。我们可以从创建一个比较函数开始，来揭示解释程序会如何整理数组中的任何两个元素。然后，我们在调用的时候将这个函数传递给 `sort()` 方法，如下所示：

```
arrayName.sort(compareFunction)
```

`compareFunction` 是告诉解释程序如何进行排序决策的函数的名称。

要建立一个比较函数，我们从一个接收两个参数的函数开始（它们描述数组中的任何两个元素）。在函数体中，虽然原来的顺序没有什么不妥，但我们还是要确定在使用 `sort()` 之后的列表中，哪个元素要出现在前边。如果我们想让第一个元素出现在第二个元素之前，那么就从函数中返回一个负数。如果我们要让第一个元素出现在第二个元素后边，就从函数中返回一个正数。如果我们想让元素保持原来的位置，就从函数中返回 0。描述这种方法的模拟代码如下：

```
function compareElements (element1,element2) {  
    if (element1 要出现在 element2 之前) {  
        return -1;  
    } else if (element1 要出现在 element2 之后) {  
        return 1;  
    } else {  
        return 0;      // 元素的位置保持原状  
    }  
}
```

例如，要将元素按照从小到大的数字顺序进行排列，我们可以使用下面所示的函数：

```
function sortAscendingNumbers (element1,element2) {  
    if (element1 < element2) {  
        return -1;  
    } else if (element1 > element2) {  
        return 1;  
    } else {  
        return 0;      // 元素是相等的  
    }  
}  
  
// 既然我们的比较函数已经准备好了，我们就来试验一下  
var x = [34, 55, 33, 1, 100];  
x.sort(sortAscendingNumbers);  
trace(x);           // 显示: "1,33,34,55,100"
```

数字排序函数实际上可以更简便地表达。前述的 *sortAscendingNumbers()* 函数可以这样写：

```
function sortAscendingNumbers (element1, element2) {  
    return element1 - element2;  
}
```

在优化版本中，如果 *element1* 小于 *element2* 就返回一个负数，如果 *element1* 大于 *element2* 就返回一个正数，如果两个数字相等就返回 0。现在就简洁了。下面是一个执行从大到小排序的版本：

```
function sortDescendingNumbers (element1, element2) {  
    return element2 - element1;  
}
```

例 11-6 给出了一个比较函数，调整 *sort()* 的默认字母表排序，因此，大小写不作区分。

#### 例 11-6：不区分大小写的字母表数字排序

```
var animals = ["Zebra", "ape"];  
  
function sortAscendingAlpha (element1, element2) {  
    return (element2.toLowerCase() < element1.toLowerCase());  
}  
  
animals.sort(sortAscendingAlpha);  
trace(animals); // 显示 "ape,Zebra"
```

当然，比较所包括的串和数字并不总是这么简单。下面，我们将一个影片剪辑数组按照它们的像素区域以上升的顺序进行排列：

```
var clips = [square1, square2, square3];  
  
function sortByClipArea (clip1, clip2) {  
    clip1area = clip1._width * clip1._height;  
    clip2area = clip2._width * clip2._height;  
    return clip1area - clip2area;  
}  
  
clips.sort(sortByClipArea);
```

这是一个很好的排序工具！

## slice()方法

*slice()*可以作为*splice()*的子集，它从数组中获取一系列元素。和*splice()*不同，*slice()*只获取元素。它会创建一个新的数组，不影响调用它的数组。*slice()*方法的语法如下所示：

```
origArray.slice(startIndex, endIndex)
```

*startIndex*指定要获取的第一个元素，*endIndex*指定要获取的最后一个元素后面的元素。*slice()*方法返回的新数组包含 *origArray[startIndex]* 到 *origArray[endIndex - 1]* 的元素序列副本。如果 *endIndex* 被遗漏，它就被默认为 *origArray.length*，返回的数组就包含从 *origArray[startIndex]* 到 *origArray[origArray.length-1]* 的元素。下面是两个例子：

```
var myList = ["a", "b", "c", "d", "e", "f"];
myList.slice(1, 3); // 返回["b", "c"] 而不是["b", "c", "d"]
myList.slice(2);   // 返回["c", "d", "e", "f"]
```

## join()方法

我们可以用*join()*来产生一个描述数组中所有元素的串。*join()*方法从将指定数组中的每一个元素转换为串开始。空的元素被转化为空串（''）。然后，*join()*将所有的小串连接成一个大串，用一个称为定界符的字符（或者字符序列）将它们分隔开。最后，*join()*返回结果串。*join()*的语法为：

```
arrayName.join(delimiter)
```

*delimiter*用来分隔 *arrayName*转化后元素的串。如果没有指定，*delimiter*就默认为一个逗号。*join()*语句的结果很容易理解，我们看下面的例子：

```
var siteSections = ['animation', 'short films', 'games'];

// 将siteTitle设置为"animation>> short films>> games"
var siteTitle = siteSections.join(">> ");

// 将siteTitle设置为"animation:short films:games"
var siteTitle = siteSections.join(":");
```

注意，*join()*不修改调用它的数组，它只基于该数组返回一个串。

调用的时候如果不带定界符, *join()*的效果就和 *toString()*很相似了。因为 *toString()*在用来分隔的逗号后面不带空格, 我们如果希望输出内容有更好的格式, 就可以使用 *join()*:

```
var x = [1, 2, 3];
trace(x.join(", "));      // 显示: "1, 2, 3", 而不是 "1,2,3"
```

*join()*方法如果使用在包含的元素本身为数组的数组上, 就有可能产生惊人的结果。由于 *join()*将元素转换为串, 而数组通过它们的 *toString()*方法转换为串, 那么嵌套数组转换为串的时候就会使用逗号作为定界符, 而不使用作为参数提供给 *join()*的定界符。换句话说, 任何提供给 *join()*的定界符都不影响嵌套数组。例如:

```
var x = [1, [2, 3, 4], 5];
x.join(", ");           // 返回 "1[2,3,4]5"
```

## toString()方法

*toString()*是一个普遍使用的对象方法, 我们将会在第十二章里学到, 它返回调用它的任何对象的串描述。在 *Array*对象上, *toString()*方法返回一个转换为串的数组元素的列表, 中间由逗号分隔。*toString()*方法可以被显式调用:

```
arrayName.toString()
```

我们一般不显式地使用 *toString()*, 只要 *arrayName*使用在一个串环境中, 它就会被自动调用。例如, 我们如果写 *trace(arrayName)*, 一个由逗号分隔的值的列表就会出现在输出窗口中, *trace(arrayName)*相当于 *trace(arrayName.toString())*。*toString()*方法在调试期间需要查看数组元素的快速、无格式的形式时非常有用。例如:

```
var sites = ["www.moock.org", "www.macromedia.com", "www.oreilly.com"];
// 在文本域中显示数组
debugOutput = "the sites array is "+sites.toString();
```

## 数组对象

读了第十二章之后, 你会认识到数组是一种类型的对象, 是 *Array*类的特殊实例。和其他对象一样, 数组对象可以添加属性。当我们把命名元素添加到数组的时候, 我

们其实是在对 *Array* 对象添加属性。但是，正如我们在前边学到的，要使用 *Array* 类的内置方法，必须是在处理编号元素的时候。

在第九章中，我们看到了一种有用的“数组 – 对象”混种，参数对象，它将编号元素同属性混合起来。（回忆一下，参数对象有一个 *callee* 属性，也将参数数组作为数组元素来存储。）现在，我们可以看到，所有的数组包含它们的编号元素列表，内置属性 *length*，以及我们添加的任何自定义属性。

## 多维数组

到现在为止，我们的讨论只限于一维数组中，它相当于电子数据表中的单行或者单列。但是，如果要创建类似于电子数据表的两行或者两列的东西呢？这时需要第二维。

ActionScript 本来只支持一维数组，但是我们可以通过创建数组构成的数组来模仿多维数组。也就是说，我们可以创建一个数组，它包含的元素本身也是数组（有的时候称为嵌套数组）。

多维数组最简单的类型就是二维数组，它的元素在概念上可以安排到一个由行和列构成的表格中——行是数组的第一维，列是第二维。

我们来考虑一下二维数组在实际中的例子。假设我们正在处理一个包含三种产品的定单，每种产品都有数量和价格。我们想模仿一个有三行（每行一个产品）和两列（一个为数量，一个为价格）的电子数据表。我们为每一行创建一个单独的数组，将列中的数值作为它的元素：

```
var row1 = [6, 2.99]; // 数量为6, 价格2.99  
var row2 = [4, 9.99]; // 数量为4, 价格9.99  
var row3 = [1, 59.99]; // 数量为1, 价格59.99
```

下面，将所有的行放入一个名为 *spreadsheet* 的数组：

```
var spreadsheet = [row1, row2, row3];
```

现在，我们可以通过将每行的数量和价格相乘，然后加到一起，从而得出总价钱。我们用两个索引（一个给行，一个给列）来访问二维数组的元素。例如，表达式

`spreadsheet[0]`表示第一行的包含两列的数组。因此，要访问 `spreadsheet` 第一行中的第二列，就要使用 `spreadsheet[0][1]`：

```
// 创建一个变量来存储定单的总价  
var total;  
  
// 现在求出定单的总价。对于每一行，将各列的数相乘，然后把所有的乘积相加即为总数  
for(var i = 0; i < spreadsheet.length; i++) {  
    total += spreadsheet[i][0] * spreadsheet[i][1];  
}  
  
trace(total); // 显示: 117.89
```

除了存储和访问之外，多维数组的操作和一般的数组差不多。我们可以对存储在多维数组中的数据使用所有的 *Array* 方法来进行操作。

## 多项选择测试的第三版本

在例 9-10 中，我们修改了例 1-1 中的多项测试。在那个示例中，我们创建了可再度使用的、集中的函数，改进了测试代码。既然我们现在又学习了数组，我们可以再来看看这个测试，对其再进行一些改善。这一次，我们会研究如何使用数组来存储用户的两个答案，以及每一道题目的正确答案，让我们的代码更简洁、更清晰，在要添加新题目的时候也更容易扩展。

这一次，我们需要改变的测试都发生在第 1 帧的 *scripts* 层中，那里放置着主要的测试代码。前边的两个版本和这一次修改后的测试版本都可以在在线代码库中找到。要查看这一次我们做了什么改善，可以读一下例 11-7 中的代码，要特别注意注释的内容。

### 例 11-7：多项选择测试的第三版本

```
stop();  
// *** 初始化主时间线变量  
var displayTotal; // 用户最终成绩的显示文本域  
var numQuestions = 2; // 测试里的题目数  
var totalCorrect = 0; // 正确答案的数目  
var userAnswers = new Array(); // 包含用户答案的数组  
var correctAnswers = [3,2]; // 包含每道题目正确答案的数组  
// 注意，我们不再需要两个长长的变量列表。  
// 命名的 q1answer, q2answer, 以及 correctAnswer1, correctAnswer2 等等。  
// 我们现在将这些信息存储在数组中。
```

```
// *** 登记用户答案的函数
function answer (choice) {
    // 由于我们现在可以通过 userAnswers 的 length 属性来检查已经回答了多少题目,
    // 就不再需要用 answer.currentAnswer 属性来手动地追踪这些信息。
    // 还要注意，我们已经废除了笨拙的 set 语句，
    // 它以前用来将用户答案动态地存储到命名变量中。

    // 将用户答案放到数组中
    userAnswers.push(choice);
    // 完成一点导航工作
    if (userAnswers.length == numQuestions) {
        gotoAndStop ("quizEnd");
    } else {
        gotoAndStop ("q'" + (userAnswers.length + 1));
    }
}

// *** 用来计算用户成绩的函数
function gradeUser() {
    // 计算答对了多少个题目
    // 我们的 userAnswer/correctAnswer 元素和前一版本中我们用来完成
    // 相同工作的混乱无比的 eval() 函数相比起来，要简洁多了。
    for (var i = 0; i < userAnswers.length; i++) {
        if (userAnswers[i] == correctAnswers[i]) {
            totalCorrect++;
        }
    }
    // 在动态文本域中显示用户的成绩
    displayTotal = totalCorrect;
}
```

读者练习：试着用 correctAnswers.length 来代替变量 numQuestions，以使基于数组的测试版本更一般化。

## 小结

本章介绍了第一种复合数据类型——数组，以及一些实际使用的例子。既然你已经理解多个数据是如何在一个单独的数据容器中表示的，你就有了坚实的基础，以进入下一章的学习。在下一章，我们要看一个功能更为强大的复合数据类型——ActionScript 的基础成分——对象。

## 第十二章

# 对象和类

本章讨论面向对象编程(OOP)，这对于很多读者来说都是一个新的领域。我们要讨论一些术语，并以实例来说明它。你可能觉得OOP很神秘，或者很难理解。其实相反，它的概念非常直观，OOP过程简单得要超过你的想象。作为它的核心，OOP意味着你将程序的某一部分当做完备的对象来看待。只要你能认识到现实世界中与你打交道的所有东西都是完备的对象，就很容易理解它。你的狗，父母，汽车，以及计算机，都是完备的对象，这表示他们可以独立地完成某些工作，而且可以在你的要求下完成另外一些工作，即使你不知道他们工作时的内部细节。

要让你的狗抓住一根棍子，不需要你是生物学家；要开动你的汽车也不需要你是机械工程师；要和你的父母交流不需要你是心理学家；要检查你的电子邮件，不需要你是计算机科学家。你需要知道的所有事情是对象愿意遵从的命令（称为方法）以及这些命令能够产生的结果。例如，如果你压下汽车的气动踏板，你会希望它加速。如果你让狗坐下，你会希望它乖乖坐着。具有了对象在现实世界里是什么的常识性知识，我们来看看如何将这些概念同ActionScript联系起来。

编程对象的一个经典实例就是弹跳的球。和一个真实的球一样，一个球对象可以有属性，来描述它的性质，比如它的直径、颜色、质量、位置和弹性。要在程序中描述我们的弹球，我们要创建一个ball对象，它的属性为radius等等。对象的属性描述它在任何给定时间内的状态，但是，一些属性会随着时间而变化。例如，要让球移动，我们需要模拟牛顿的运动定律，也就是说，我们需要以计算机的语言来描

述一个球随着时间所发生的运动。在最简单的例子里，速度乘以时间可以得到距离，我们就可以用下面的代码来确定球在以后某个时间的水平位置：

```
ball.xPosition += ball.xVelocity * (elapsedTime)
```

这个等式以球的当前位置开始，加上它经过的距离（基于它的速度和运动时间），就得到了新的位置。对象的行为（behavior）就是控制它的一些规则，比如我们用来计算球经过一定时间所处位置的等式。我们通常把这些行为放在所谓的方法里，方法其实就是实现对象行为的函数。例如，我们可以创建一个 *move()* 方法，使用上面的等式。

因此，方法可以被看作一个对象所遵循的命令。当然，一个对象可以有多个方法。我们要让球反弹，可以创建一个 *bounce()* 方法，它将反转球的方向，降低它的速度（我们的球并非完全弹性的）。*bounce()* 方法可以实现下面的等式：

```
ball.xVelocity = -(ball.xVelocity) * 0.95 // 球的弹性系数为 0.95
```

在讨论对象创建、属性添加，以及方法实现的深奥内容之前，我们将所说的几个定义正规化一下。一个对象在理论上就是一个数据结构，它将有关的属性和方法（函数）集合在一起。一个对象要封装它的行为，这意味着其内部对函数的执行细节没有必要让外边的对象看见。实际上，一个程序要通过所谓的接口（也就是说，可以让外边的对象访问到的方法）来和一个对象进行交互。程序其他的部分不必担心对象如何完成它应该完成的工作，而只对对象提供输入（input），并在适当的时候检查输出（output）（也就是结果）。你还将听到类（class）和实例（instance）。一个类就是一个普通的对象类别，而一个实例就是对象的特定情况（例如一个复本）。例如，你的宠物狗就是 *Dog* 类的一个实例。*Dog* 类中所有的狗都会叫，都有四条腿，但是，你的特定的狗有用来描述它的高度、重量和颜色的特定值。

面向对象编程（OOP）只是使用了对象的程序的名称。对象和 OOP 都是 ActionScript 内在的东西，我们已经用到过，也许你没有注意到。影片剪辑是 Flash 中熟悉的对象，和所有对象一样，它作为属性和方法的集合来实现。当用 *someClip.\_height* 来确定一个影片剪辑的高度时，我们是在访问这个剪辑对象的 *\_height* 属性。当用 *someClip.play()* 来播放一个影片剪辑的时候，我们是在调用这个剪辑对象的 *play()* 方法。

---

注意：通常，一个对象类的所有实例都有相同的方法和属性名称，用属性的值可将一个实例与相同类的其他实例区分开来。

---

不管我们是自己创建对象，还是使用 ActionScript 的内置对象，OOP 都会将程序的一个成分同其他成分清楚地分隔开来（封装），并允许它们在互相不知道其细节的情况下混合。这可让对象改变它们的内部功能，只要对象的方法（也就是说，它对外界的接口）不改变，就不会影响使用该对象的程序的其他部分。例如，回到我们的 ball 对象中来，我们不关心球运动的物理规律是否发生变化，我们只调用球的 *move()* 方法，让对象自己来关注它的细节。

OOP 的另一优点是：可以用统一的方法来对待拥有不同行为的不同对象，只要它们实现同名的方法。例如，假设我们有一个 *circle* 对象和一个 *square* 对象。只要两个对象都实现返回形体面积的 *area()* 方法，我们就可以调用它们的 *area()* 方法，而不必关心每一个对象是如何计算它的面积的。

在本章中，我们将要学习如何建立基本的对象，以及如何定义对象的类别（也就是类）。一旦我们掌握了基础知识，就可以开始探讨如何使用继承（inheritance）来在类和对象之间分享相同的特性 [也就是创建一个家族树（family tree）]。例如，我们可以实现一个 *Horse* 类，它和 *Dog* 类一样都是 *Mammal*（哺乳动物）类的后代。*Mammal* 类可以实现对所有哺乳动物都相同的方法和属性，比如它们有毛发，有乳汁，是暖血动物的事实。最后，我们将学习如何使用 OOP，通过 ActionScript 的内置对象和类来控制 Flash 环境。

我希望这个介绍能够起到对对象和 OOP 提纲挈领的作用。现在我们就开始细节内容的讨论。

## 对象的分析

和数组一样，单独的对象是容器的容器。数组在单个的元素中容纳多个数据值，与之相似，对象通过单个的属性来容纳多个数据值。但是，对象的属性是命名的，不是编号的。数组在一个编号的列表中存储一组元素，对象按照没有特别顺序的独特的标识符来存储属性。要访问一个数组元素，我们需要知道它的号码位置，但是要访问一个对象属性，我们需要知道它的名称（也就是标识符）。

图 12-1 描述了简单对象 ball 的属性。ball 对象包含两个属性：radius 和 color。这些属性的值分别是 50 和 0xFF0000（红颜色的十六进制值）。用独特标识符命名的属性和变量很类似。即使每一个属性都有自己的名称，它们也都包括在单独的对象 ball 中。



图 12-1 对象结构示意

显然，对象定义的属性都是有一定关系的。应特别指出，对象属性的选择应该有助于将对象的实例相互区别开来。例如，影片剪辑对象的属性就是独特的，比如它们的帧号（\_totalframes）和位置（\_x 和 \_y）。

因为对象属性是命名的而不是编号的，所以对象没有数组的元素管理工具（*shift()*, *unshift()*, *push()*, *splice()*, 等等）。对象属性一般通过对对象方法的传统手段来进行设置，以保存对象所封装的特性。也就是说，在严格的 OOP 中，对象应该设置它自己的属性。如果一个外部实体要设置对象的属性，它必须调用相应的对象方法。例如，一个有修辞癖的人会主张，最好让 *Array* 对象自己设置 *length* 属性，而对象之外的代码要这样做必须以非直接的方式，通过调用 *Array* 对象的某个方法来实现。在这种情况下，*Array* 类的维护者就会将 *length* 属性的名称改成 *len*，以不影响其他用户。

## 实例化对象

虽然听起来显得很土，但我们还是假设已经创建了一个对象类。这个假设并不滑稽，因为 ActionScript 提供了很多内置的类，我们定义的类也有相似的行为。假设我们已经有了一个对象类，我们必须基于这个类创建一个特殊的对象实例（也就是一个复本）。例如，ActionScript 提供了 *Array* 对象类，它可以创建单个的数组。

要创建一个对象的实例（即实例化对象），我们使用 `new` 操作符和一个构造器函数，后者是用来初始化对象的。常用的语法为：

```
new ConstructorFunction()
```

我们用一个 ActionScript 内置的构造器来实例化第一个对象：`Object()` 构造器。`Object()` 构造器创建完整类属对象（它因此得名），类属对象是其他所有对象类型所基于的基础对象类型。下面的代码将创建一个新的一般对象：

```
new Object()
```

当我们实例化一个对象的时候，通常将得到的实例存储在一个变量、数组元素或者对象属性中，以备以后的访问。例如：

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

按这种方法实例化一个对象会给我们一个没有属性的空对象和两个一般的方法。因此，类属对象的使用是有限的——真正具有强大功能的是特殊对象类。在我们学习如何建立自己的类之前，先来看看如何使用现有类的对象属性和方法。

## 对象属性

属性是与对象有关的命名的数据容器。它们由对象类定义，然后单独地设置每个对象实例。和变量一样，对象属性可以包含任何类型的数据——串、数字、布尔、`null`、`undefined`、函数、数组、影片剪辑或者其他对象。

## 属性的引用

可以使用点操作符来访问对象属性。我们用一个点将属性的名字和它所属的对象分开，如下所示：

```
objectName.propertyName
```

`objectName` 是对象的名字，`propertyName` 必须是一个合法的标识符，与 `objectName` 的某个属性名称相匹配。

例如，如果有一个 ball 对象实例，它有一个 radius 属性，我们可以用下面的代码来访问 radius：

```
ball.radius
```

另一种方法是用 [] 操作符来指向属性，如下所示：

```
objectName[property]
```

[] 操作符可以用任何能生成串的表达式构成属性的名字。例如：

```
trace(ball["radius"]);  
  
var prop="radius";  
trace(ball[prop]); // prop产生的是 "radius"
```

内置 ActionScript 属性可以用相同的方法进行访问。回忆一下获取 pi 值的语法：

```
Math.PI
```

在这个表达式中，我们访问的是内置 Math 对象的 PI 属性。但是，在单纯的 OOP 中，我们几乎从不直接访问对象的属性，而是使用方法来访问属性的值。例如，要检查内置 Sound 类的一个实例的 volume 属性，我们使用下面的代码：

```
trace(mySound.getVolume());
```

而不用：

```
trace(mySound.volume);
```

## 使用 for-in 循环来访问对象的属性

在第八章中，我们学习了 *for-in* 语句，它可以用来列举一个对象的属性。既然我们已经了解了一些关于对象的知识，就可以回到 *for-in* 语句，来重新看看它是如何操作对象属性的。

和所有的循环一样，*for-in* 语句包括一个头部和一个循环体。*for-in* 语句的循环体会对指定对象中的每一个属性自动执行一次。因为在每一轮循环执行的时候，“循环”变量会自动变成下一个属性的名称，所以我们不需要知道属性的数量和它们的名称。我们因此而可以访问对象的属性，如下所示：

```
// 列出ball 对象的所有属性
for(var prop in ball) {
    trace("Property " + prop + " has the value " + ball[prop]);
}
```

注意，前面例子中的循环变量 `prop` 不像在 `for` 循环中那样是一个整数。也就是说，不要将一个通常用来访问编号数组元素的标准 `for` 循环同用来访问对象属性的 `for-in` 混淆起来。要了解关于 `for-in` 循环的更多细节，可以参见第八章。

## 方法

方法是和对象相联系的函数，通常被用来执行任务或者访问对象的数据。我们用函数调用操作符`()`来调用方法、用点操作符来分隔对象和方法的名称：

```
objectName.methodName()
```

例如：

```
ball.getArea(); // 调用 ball 的 getArea()
```

和属性一样，方法被定义给类，然后在单独的对象实例上调用。但是，在我们学习如何在类中创建方法之前，先要讨论一下面向对象编程和对内置 `Object` 类的单个对象实例添加方法的规则。我们只要理解了一种方法是如何在独立的对象中工作的，就可以把这一概念应用到我们自己创建的类中。

方法实际上就是一个存储在对象属性中的函数。要将一个函数赋给对象属性，需要将函数转换为对象的方法：

```
// 创建一个对象
myObject = new Object();

// 创建一个函数
function greet() {
    trace("hello world");
}

// 现在将 greet 赋给属性 sayHello
myObject.sayHello = greet;
```

一旦对象的属性中有了函数，就可以将函数作为方法调用，如下所示：

```
myObject.sayHello();      // 显示: "hello world"
```

你也可以将一个函数赋给数组元素(而不是对象属性),然后用调用操作符来调用这个函数,如下所示:

```
var myList = new Array();
myList[0] = greet;
myList[0]();           // 显示: "hello world"
```

但是,当一个函数作为对象的方法来调用的时候,函数会发生一些特殊情况——它能获取或者设置它所属对象的属性。我们来看看在sayHello属性上发生的这种情况。首先,添加msg属性到类属对象(重申一次,我们为了演示的成功而曲解了OOP规则……将一个自定义属性添加到单独的对象实例上,这种形式并不好):

```
myObject.msg = 'Nice day, isn't it?';
```

现在,我们调整greet(),让它显示msg的值:

```
function greet() {
    trace(this.msg);
}

// 现在再次调用 sayHello
myObject.sayHello();      // 显示: "Nice day, isn't it?"
```

注意到第二行的关键字this了吗?这是我们和myObject的连接。作为myObject的一个方法来执行的时候,sayHello()有效地传递一个无形的参数——this,它包含对myObject的一个引用。在函数体内,我们用this来访问myObject的msg属性,形式为this.msg。当调用myObject.sayHello()的时候,表达式this.msg就变成了myObject.msg,也就是“Nice day, isn't it?”。

方法可以获取和设置主对象的属性值。例12-1显示的方法获取了一个对象的两个属性值,基于这些值进行计算,然后将这个值设置给第三个新的属性。(再说一次,因为我们当前的焦点是方法,我们使用的对象只是一个对象实例。在下一节中,我们将学习如何正确地用类来添加对象方法。)

#### 例 12-1: 实现一个设置属性的方法

```
// 建立一个新的对象,将它存储在rectangle中
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
```

```

// 定义一个函数来计算矩形的面积，然后设置一个对应的属性
// 注意this关键字的用法
function rectArea() {
    this.area = this.width * this.height;
}

// 将函数赋为rectangle的方法
rectangle.setArea = rectArea;

// 现在调用rectangle的setArea()方法
rectangle.setArea(); // 将rectangle.area设置为50

// 最后，检查由setArea()生成的新属性
trace(rectangle.area); // 显示：50

```

让一个方法返回一个值而不是设置一个属性的用法相当典型。我们修改一下rectangle的*setArea()*方法来完成这个工作，如例12-2所示：

#### 例12-2：从方法返回一个值

```

// 创建并给rectangle赋值。注意，在rectArea出现
// 在代码中之前给area的方法赋值是合法的
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

// 这一次只返回area，而不将它赋为属性
function rectArea() {
    return this.width * this.height;
}

// 现在使用起来就更方便了
trace("The area of rectangle is: " + rectangle.area());
// 显示："The area of rectangle is:50"

```

*this*关键字可能是一个棘手的概念，因为它包含一些解释程序技巧。要说明前面的是代码如何工作的，我们来看例12-3，它揭示了*this*和它的对象以及方法之间的关系。（例12-3是虚构的，因为*this*是一个保留的关键字，如果像下面这么使用，就会引起错误。）

#### 例12-3：显式使用*this*的虚构示例

```

// 创建并设定rectangle对象
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;
rectangle.area = rectArea;

```

```
// 明确要求this作为rectArea()函数的参数
function rectArea(this) {
    return this.width * this.height;
}

// 用对rectangle对象的明确引用来调用方法
trace("The area of rectangle is: " + rectangle.area(rectangle));
```

方法也可以作为函数直接量赋给对象属性。使用函数直接量来创建方法，可以不用先创建函数然后再将它赋给属性。例 12-4 重新处理了例 12-2 中的 rectangle 对象，给出了函数直接量的用法。

#### 例 12-4：用函数直接量来实现方法

```
var rectangle = new Object();
rectangle.width = 10;
rectangle.height = 5;

// 下面是一个函数直接量的赋值
rectangle.area = function() {
    return this.width * this.height;
}; // 在直接量的后面使用分号是很好的形式

// 我们仍然可以像以前那样调用函数
trace("The area of rectangle is: " + rectangle.area());
```

方法可以很好地控制对象。当使用在内置对象上的时候，方法是我们用来控制 Flash 影片环境的主要工具。

在本节中，我们已经学习了如何将函数存储在属性中（将它变成一个方法），也学习了如何通过属性来调用一个函数。这两种方式都有相似的语法，但是结果却大相径庭，应着重理解它们的不同之处。

---

**注意：**将一个方法赋给一个属性的时候，可省去括号：

```
myObj.myMethod = someFunction;
```

调用一个方法的时候，就要使用括号：

```
myObj.myMethod();
```

---

它们在语法上有非常细微的差别，要确保书写正确。

---

## 类和面向对象的编程

要创建无数个复杂的对象，用它们存储从购物车系统中的产品到视频游戏中有人工智能的坏蛋等所有东西的丰富信息，并不是什么稀罕的事情。要加速对象的创建，定义对象的层次体系（两个对象之间的关系），可以使用对象类。一个类（class）就是对整个一类对象的模板式的定义。正如我们在前面介绍的那样，类描述一个特定对象类的共同特征，比如“所有的狗都有四条腿”。

### 对象类

在讨论如何使用类之前，先看看在不使用类的情况下要怎么做。假设我们需要一个ball对象，但不使用类来生成它，我们只需要简单地修改一下内置Object类的类属对象。我们给出对象下面的属性：radius、color、xPosition和yPosition。然后，我们添加两种方法——*moveTo()*和*area()*——用来重新定位对象，以及确定它所占据的空间大小。

代码如下：

```
var ball = new Object();
ball.radius = 10;
ball.color = 0xFF0000;
ball.xPosition = 59;
ball.yPosition = 15;
ball.moveTo = function(x, y) { this.xPosition = x; this.yPosition = y; };
ball.area = function() { return Math.PI * (this.radius * this.radius); };
```

这种方法虽然可以完成任务，但是有一定的局限性。每一次我们需要一个新的和ball类似的对象时，必须重新使用ball的初始化代码，这样做很单调而且容易出错。另外，按照这种方法创建许多ball对象时，需要在每一个对象里重复地复制同样的*moveTo()*和*area()*函数代码，这会造成存储器的浪费。

要有效地创建一系列有共同属性的对象，我们可以创建一个类。使用类可以定义所有ball对象都拥有的属性。此外，可以在ball类的所有实例中共享任何属性。在自然语言中，我们的Ball类会被解释程序描述成这样：

*Ball*是一个对象类型。所有的ball对象实例拥有的属性为radius、color、xPosition、yPosition，它们会针对每一个ball进行单独的设置。所有的

*ball* 对象也共享方法 *moveTo()* 和 *area()*，它们对 *Ball* 类的所有成员都是一样的。

现在我们来看看这个理论在实践中的运用。

## 建立类

在 ActionScript 中并没有特定的“类定义”，也没有用来创建新类的与创建新变量的 *var* 语句相似的 *class* 语句。我们定义了一种特殊类型的函数，称为构造器函数，它可以产生类的新实例。通过定义构造器函数，可以有效地创建类模板或者类定义。

在句法上，构造器函数（或者简称为构造器）的格式和普通函数类似。例如：

```
function Constructor() {  
    statements  
}
```

类构造器函数的名称可以是有效的函数名称，但是传统上它首字母大写，以表示这是一个类构造器。构造器的名称用来描述它所创建的对象类，比如 *Ball*, *Product* 或者 *Vector2d*。构造器函数中的 *statements* 用来初始化它所创建的对象。

开始时，我们会让 *Ball* 构造器尽可能简单。我们迄今为止所要做的工作就是创建一个空的对象：

```
// 建立一个 Ball 构造器  
function Ball() {  
    // 这里要完成一些动作  
}
```

这并不费力。现在，我们来看看用 *Ball* 构造器函数如何生成 *ball* 对象。

## 创建类的成员

构造器函数可以用来定义类，也可以用来实例化类的新实例。正如我们在前边看到的，和 *new* 操作符在一起调用的时候，构造器函数创建然后返回一个对象实例。回忆一下基于构造器函数来创建新对象的常用语法：

```
new Constructor();           // 返回 Constructor 类的一个实例
```

因此，要用 *Ball* 类来创建 ball 对象（也就是一个实例），可这样写：

```
myBall = new Ball(); // 将 Ball 类的一个实例存储在 myBall 中
```

*Ball* 类仍然没有对它所创建的对象添加属性或者方法。我们马上就要学习这个内容。

### 将自定义的属性赋给类的对象

要在对象创建期间定制一个对象的属性，我们开始转向特定的 *this* 关键字。在构造器函数中，*this* 关键字存储一个对当前产生的对象的引用。使用这个引用，可以将任何想要的属性赋给新生的对象。下面的语法给出的常用的手法：

```
function Constructor() {
    this.propertyName = value;
}
```

*this* 是被创建的对象，*propertyName* 是要添加给对象的属性，*value* 是要赋给属性的数据值。

现在在 *Ball* 例子中使用这个技术。首先，我们假定 *Ball* 类的属性会有 *radius*, *color*, *xPosition* 和 *yPosition*。下面的代码中，*Ball* 类构造器将这些属性赋给它的实例（注意 *this* 关键字的用法）：

```
function Ball() {
    this.radius = 10;
    this.color = 0xFF0000;
    this.xPosition = 59;
    this.yPosition = 15;
}
```

*Ball* 构造器准备好了之后，就可以通过调用具有 *new* 操作符的 *Ball()*，创建带有这些属性 (*radius*, *color*, *xPosition* 和 *yPosition*) 的对象实例（也就是类的成员），就像我们前边所做的那样。例如：

```
// 建立 Ball 的一个新实例
bouncyBall = new Ball();

// 现在访问 bouncyBall 的属性。这些属性是用 Ball() 构造器建立对象的时候设置的
trace(bouncyBall.radius); // 显示:10
trace(bouncyBall.color); // 显示:16711680
trace(bouncyBall.xPosition); // 显示:59
trace(bouncyBall.yPosition); // 显示:15
```

有意思吗？

不幸的是，我们的 *Ball()* 构造器在将属性赋给它所创建的对象的时候使用的是固定的值（比如，`this.radius=10`）。因此，每一个 *Ball* 类的对象都会有相同的属性值，它和面向对象编程的目标是相对立的（我们并不需要一个只会产生同样对象的类，让它们互不相同才是真正有趣的地方）。

要动态地将属性值赋给类的实例，我们要调整构造器函数，让它能够接受参数。我们来介绍一下常用的语法，然后回到 *Ball* 的例子：

```
function Constructor (value1, value2, value3) {  
    this.property1 = value1;  
    this.property2 = value2;  
    this.property3 = value3;  
}
```

在 *Constructor* 函数中，我们用 `this` 关键字指向被创建的对象，正如在前边所做的那样。但是这一次，我们不再费力地编写对象属性的值，而是将参数 `value1`, `value2` 和 `value3` 的值赋给对象的属性。当要创建一个新的、独特的类成员时，就将初始化的属性值传递给构造函数，以建立新的实例：

```
myObject = new Constructor (value1, value2, value3);
```

我们来看看它在 *Ball* 类上的应用，如例 12-5 所示：

#### 例 12-5：一个普遍的 *Ball* 类

```
// 建立 Ball() 构造器，它将属性值作为参数接收  
function Ball (radius, color, xPosition, yPosition) {  
    this.radius = radius;  
    this.color = color;  
    this.xPosition = xPosition;  
    this.yPosition = yPosition;  
}  
  
// 调用构造器，向它传递函数，作为对象属性的值  
myBall = new Ball(10, 0x00FF00, 59, 15);  
  
// 现在我们来看看具体的工作情况……  
trace(myBall.radius); // 显示: 10, :) 真酷...
```

我们基本上建立了 *Ball* 类。但是你会注意到，我们仍然还没有涉及到前面所提到的 *moveTo()* 和 *area()* 方法。要将方法添加到类对象有两种途径。我们现在要学习的是较为简单的、效果不那么强烈的一种方法，在我们讨论了继承之后再回到方法创建的主题上来。

## 将方法赋给类的对象

为类添加一个方法的最简单办法就是在构造器中设置一个包含函数的属性。下面是常用的语法：

```
function Constructor() {  
    this.methodName = function;  
}
```

*function*可以以多种方式提供，可以通过为 *Ball* 类添加 *area()* 方法来检查。注意，我们为了简洁已经从类中删除了 *color*, *xPosition* 和 *yPosition*。

*function* 可以是一个函数常量，比如：

```
function Ball (radius) {  
    this.radius = radius;  
    // 添加area方法...  
    this.area = function() { return Math.PI * this.radius * this.radius; };
```

另外，*function*可以在构造器中声明：

```
function Ball (radius) {  
    this.radius = radius;  
    // 添加area方法...  
    this.area = getArea;  
    function getArea () {  
        return Math.PI * this.radius * this.radius;  
    }  
}
```

最后，*function*可以在构造器外声明，但是在构造器内进行赋值：

```
// 声明getArea()函数  
function getArea () {  
    return Math.PI * this.radius * this.radius;  
}  
  
function Ball (radius) {  
    this.radius = radius;  
    // 添加area方法...  
    this.area = getArea;  
}
```

三种方法之间并没有实际的差别——都是合理合法的。函数直接量的使用通常显

得更方便，但是不如在构造器外定义一个函数那样可以在其他构造器中再度使用。不管怎么说，这三种方法效率都不高。

到现在为止，我们已经为 *Ball* 类的每一个对象添加了独特的属性值。每一个 ball 对象需要用它自己的 radius 和 color 属性值来区别于其他 ball。但是，当我们用刚才学习的技术为类的对象赋一个固定的方法时，就不需要在类的每一个对象上复制方法。*area* 公式对所有的 ball 都是一样的，因此，执行这个任务的代码可以集中产生。

使用继承可用固定值向类中有效地添加任何属性或者方法，这是下一个讨论的主题。

## 对象属性继承性

继承的属性不添加到类的单个对象实例上，它们被一次性地添加到类构造器中，然后在需要的时候被对象借用。这些属性被称为继承的，是因为它们是被传递给对象的，而不是在每个对象内定义的。继承属性看起来似乎属于对象（属性通过这些对象被引用），但是实际上它是对象的类构造器的一部分。

图 12-2 给出了继承属性常用的模型，以 *Ball* 为例。因为 *Ball* 的 *moveTo()* 和 *area()* 方法在实例中没有变化，这些方法作为继承方法来实现很恰当。它们属于类本身，每一个 ball 对象只能通过引用访问它们。换句话说，*Ball* 类的 radius、color、xPosition 和 yPosition 属性被作为普通属性赋给每一个对象，因为每一个 ball 需要它自己的这些属性值。

继承在一个继承链中运作，就像一个家族树一样。当调用单个对象的方法时，解释程序会查看该对象是否定义了一个 *area()* 方法。如果该对象没有定义 *area()* 方法，解释程序就会检查 *Ball* 类是否定义了 *area()* 方法。

例如，如果我们执行 *ball1.area()*，解释程序就会查看 *ball1* 是否定义了一个 *area()* 方法，如果 *ball1* 没有定义 *area()* 方法，解释程序就会检查 *Ball* 类是否定义了 *area()* 方法，如果的确定义了，解释程序就调用 *area()*，就像它是 *ball1* 对象的方法而不是 *Ball* 类的方法一样。这种做法允许方法在 *ball1* 对象上操作（而不仅是类），在需要的时候获取或者设置 *ball1* 的属性，这是 OOP 的一个关键优点：在一个地方（*Ball* 类）定义函数却可从很多地方（任何 *ball* 对象）使用它。

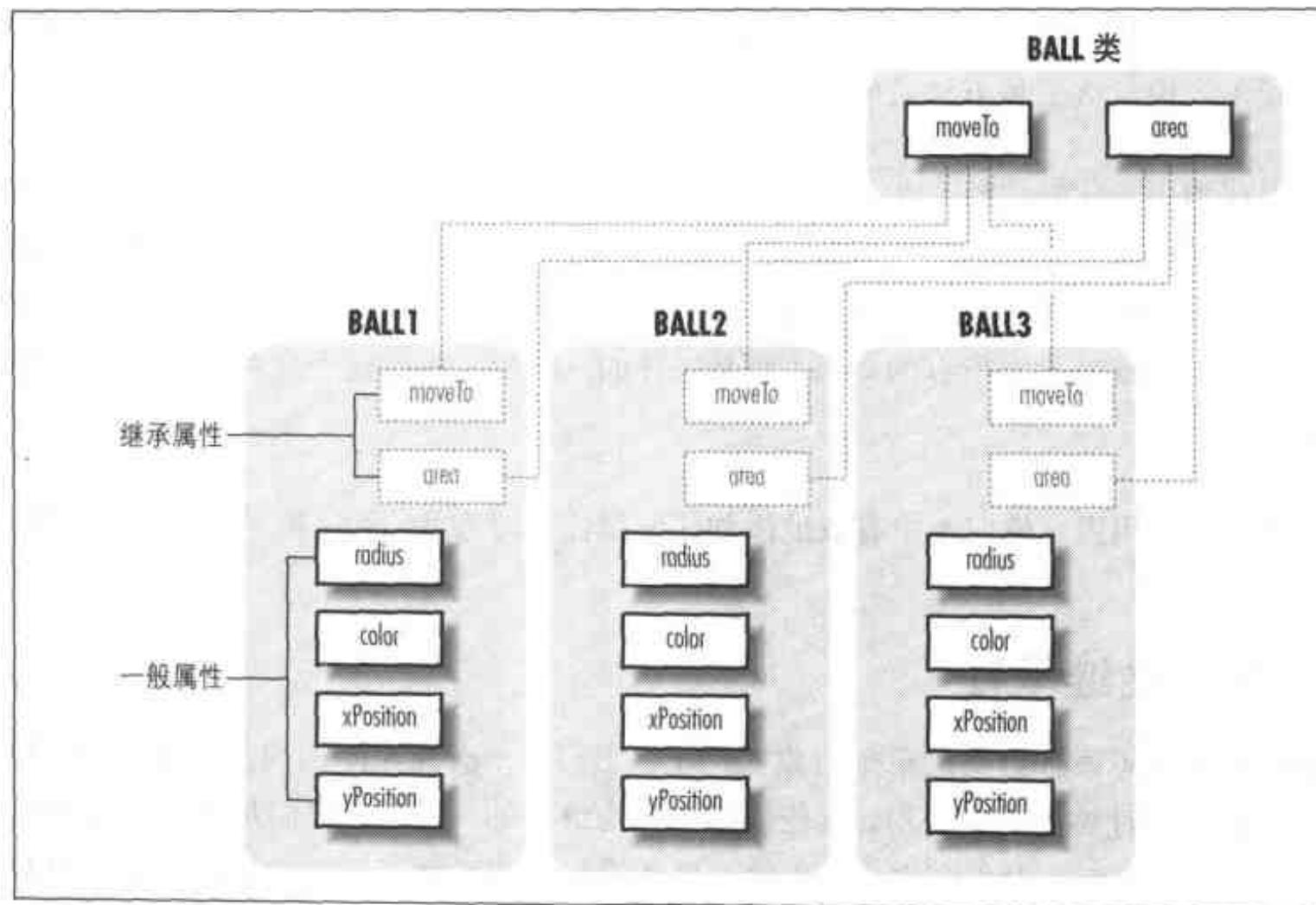


图 12-2 继承和普通属性

和普通的属性不同，继承属性只能被一个对象获取而不能设置。

现在，我们用一些代码来印证这些概念。

### 用原型属性创建继承属性

我们先创建一个类构造器函数，比如例 12-5 中的 *Ball* 类，以开始继承属性的创建过程：

```
function Ball (radius, color, xPosition, yPosition) {
    this.radius = radius;
    this.color = color;
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}
```

我们在第九章中说过，函数同时也是对象，可以带属性。

---

注意：创建构造器函数的时候，解释程序自动赋值一个称为原型（prototype）的属性。在原型属性中，解释程序放置一个类属对象。任何添加到原型对象的属性都可以被构造器函数类的所有实例继承。

---

要创建可以被类的所有对象继承的属性，我们简单地将这个属性赋给类构造函数预制的原型对象。下面是常用的语法：

```
Constructor.prototype.propName = value;
```

*Constructor*是类构造器函数（我们的例子中就是*Ball*）。*prototype*是自动生成的属性，我们用它来包含继承属性，*propName*是继承属性的名称，*value*是继承属性的值。例如，我们利用下面的代码来为*Ball*对象的整个类添加一个全局*gravity*属性：

```
Ball.prototype.gravity = 9.8;
```

*gravity*属性添加好后就可以从*Ball*类的任何成员访问*gravity*了：

```
// 创建Ball类的一个新的实例  
myBall = new Ball(5, 0x003300, 34, 220);  
  
// 现在显示继承属性gravity的值  
trace(myBall.gravity); // 显示: 9.8
```

*gravity*属性可以通过*myBall*来访问，因为*myBall*继承了*Ball*的*prototype*对象属性。

因为一个类的所有实例都可以使用相同的方法，它们都存储在继承属性里。我们来为*Ball*类添加一个继承方法*area()*：

```
Ball.prototype.area = function() {  
    return Math.PI * this.radius * this.radius;  
}; // 要加分号是因为这是一个函数常量
```

真有意思，我们来添加一个继承方法*moveTo()*，这次使用一个预制函数，而不是函数直接量：

```
function moveTo (x, y) {  
    this.xPosition = x;  
    this.yPosition = y;  
}
```

```
Ball.prototype.moveTo = moveTo;
```

一旦函数作为继承属性来定义，就可以像使用其他方法一样调用它：

```
// 建立一个新的Ball  
myBall = new Ball(15, 0x33FFCC, 100, 50);  
  
// 现在调用myBall的继承方法area()  
trace(myBall.area()); // 显示: 706.858347057703
```

注意，也可以用一个新的对象来完全代替构造器的prototype对象，从而以单一的动作添加许多继承属性。但是，这么做会改变继承链，我们会在后面讨论。

## 覆盖继承属性

要为单一的对象自定义一个继承属性，可以使用和继承属性相同的名字在对象上赋一个属性。例如，我们也许想给某个球以小于其他球的重力：

```
// 创建Ball类构造器  
function Ball (radius, color, xPosition, yPosition) (...) // 不显示了  
  
// 设置一个继承属性gravity  
Ball.prototype.gravity = 9.8  
  
// 创建一个Ball对象  
lowGravBall = new Ball ( 200, 0x22DD99, 35, 100 );  
  
// 覆盖继承属性gravity  
lowGravBall.gravity = 4.5;
```

设置在对象上的属性通常可以覆盖同名的任何继承属性，这是继承链工作方式的一个简单结果。同样，对象内局部定义的方法也会覆盖类中的那些方法。如果我们执行ball1.area()，而ball1定义了一个area()方法，这个方法就会被使用，解释程序根本不会去查看Ball类的prototype，寻找它是否也定义了area()方法。

## 构造器属性

当创建构造器的prototype对象的时候，解释程序自动赋给它一个称为constructor的特定属性。constructor属性是对prototype的类构造器函数的一个引用。例如，下面的表达式都可以产生对Ball构造器函数的一个引用：

```
trace(Ball); // 显示: [Function]
trace(Ball.prototype.constructor); // 也显示: [Function]
// 和上面相同的引用
```

注意，constructor 属性包含对构造器函数的一个引用，而不是一个表示函数名称的串。

### \_\_proto\_\_ 属性

创建任何对象的时候，解释程序自动赋给它一个特殊的属性，称为`__proto__`（注意，名称两边分别有两个下划线）。对象的`__proto__`属性是对该对象的构造器函数`prototype`属性的一个引用。例如，当创建一个名为`myBall` 的 `Ball` 实例时，`myBall.__proto__` 被设置为 `Ball.prototype`：

```
myBall = new Ball(6, 0x00FF00, 145, 200);
trace(myBall.__proto__ == Ball.prototype); // 显示: true
```

`__proto__` 属性主要被 ActionScript 解释程序用来查找对象的继承属性。例如，当我们通过`myBall` 调用继承方法`area()`的时候，比如`myBall.area()`，ActionScript 就通过`myBall.__proto__` 来访问 `Ball.prototype.area`。

我们也可以用`__proto__` 来直接检查一个对象是否属于一个指定的类，如例 12-6 所示。

#### 例 12-6：确定一个对象所属的类

```
function MyClass (prop) {
    this.prop = prop;
}

myObj = new MyClass();
if (myObj.__proto__ == MyClass.prototype) {
    trace("myObj is an instance of MyClass");
}
```

## 超类和子类

在高级面向对象编程中，类的一个重要功能就是能够共享属性。也就是说，整个类可以从其他类中继承属性，或者传递属性。在复杂的情况下，必须有多级的继承。（即使你自己不使用多级继承，理解它也可以帮助你处理内置的 ActionScript 类。）

我们已经知道对象是如何从类构造器中的 prototype 对象中继承属性的。继承并不限制在单独的对象 / 类关系中。一个类本身可以从其他类继承属性。例如，我们有一个称为 *Circle* 的类，它定义了一个普通的方法 *area()*，用来求所有 *circle* 对象的面积。我们不在 *Ball* 这样的类中定义单独的 *area()* 方法，而让 *Ball* 继承可以从 *Circle* 访问到的 *area()* 方法。因此，*Ball* 的实例通过 *Ball* 来继承 *Circle* 的 *area()* 方法。注意层次——最简单的类 *Circle*，定义了最通用的方法和属性。另外一个类 *Ball* 依靠简单的 *Circle* 类来为 *Ball* 类的实例添加功能细节，而依靠 *Circle* 来建立所有 *Circle* 对象共享的基础属性。在传统的面向对象编程中，*Ball* 类将被认为是扩展了 *Circle* 类。也就是说，*Ball* 是 *Circle* 的一个子类，而 *Circle* 是 *Ball* 的超类。

## 建立一个超类

我们在前面学习了在类构造器函数的 prototype 对象上定义继承属性。要对给定的类创建一个超类，我们要用所要超类的一个新实例来完全替代类的 prototype 对象。常用的语法如下：

```
Constructor.prototype = new SuperClass();
```

通过用 *SuperClass* 来代替 *Constructor* 的 prototype 对象，迫使所有的 *Constructor* 实例继承在 *SuperClass* 实例中定义的属性。在例 12-7 中，我们首先创建一个类 *Circle*，它将 *area()* 方法赋给它所有的实例。然后，我们将 *Circle* 的一个实例赋给 *Ball.prototype*，让所有的 *ball* 对象都从 *Circle* 继承 *area()*。

### 例 12-7：创建一个超类

```
// 创建 Circle (超类) 构造器
function Circle() {
    this.area = function() { return Math.PI * this.radius * this.radius; }
}

// 创建普通的 Ball 类构造器
function Ball (radius, color, xPosition, yPosition) {
    this.radius = radius;
    this.color = color;
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}

// 我们在这里将 Circle 的一个实例赋给 Ball 类的构造器函数，以建立超类
Ball.prototype = new Circle();
```

```
// 现在我们建立一个Ball 的实例，然后检查它的属性  
myBall = new Ball ( 16, 0x445599, 34, 5);  
  
trace(myBall.xPosition);           // 34、一个Ball 的普通属性  
trace(myBall.area());             // 804.24...，area()是从Circle中继承的
```

但是，现在我们的类继承的结构还很差——*Ball* 定义了 *radius*，但是 *radius* 实际上是所有 *circle* 对象的一个通用属性，因此它应该归入我们的 *Circle* 类。*xPosition* 和 *yPosition* 也是如此。要有一个固定的结构，可以将 *radius*、*xPosition* 和 *yPosition* 移到 *Circle* 中，只在 *Ball* 中留下 *color*。（为了示范，我们将 *color* 当作一个只有 *ball* 才能有的属性。）

下面的代码从概念上建立了改进过的 *Circle* 和 *Ball* 构造器：

```
// 创建Circle(超类) 构造器  
function Circle(radius,xPosition,yPosition) {  
    this.area = function() { return Math.PI * this.radius * this.radius; };  
    this.radius = radius;  
    this.xPosition = xPosition;  
    this.yPosition = yPosition;  
}  
  
// 创建Ball 类构造器  
function Ball ( color ) {  
    this.color = color;  
}
```

移动了属性之后，我们又面临一个新的问题，如何能够在用 *Ball* 而非 *Circle* 来创建对象的时候，为 *radius*、*xPosition* 和 *yPosition* 提供值呢？我们必须对 *Ball* 构造器代码作一个调整。首先，建立 *Ball* 来接收所有作为参数的所需属性：

```
function Ball ( color, radius, xPosition, yPosition ) {
```

然后，在 *Ball* 构造器内，将 *Circle* 构造器定义为被继承的 *ball* 对象的方法：

```
this.superClass = Circle;
```

最后，在 *ball* 对象上调用 *Circle* 构造器，为 *radius*、*xPosition* 和 *yPosition* 传递它的值：

```
this.superClass(radius, xPosition, yPosition);
```

完成后的类 / 超类代码如例 12-8 所示。

### 例 12-8：一个类和它的超类

```
// 创建Circle(超类)的构造器
function Circle ( radius, xPosition, yPosition ) {
    this.area = function() { return Math.PI * this.radius * this.radius; };
    this.radius = radius;
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}

// 创建Ball 类构造器
function Ball ( color, radius, xPosition, yPosition ) {
    // 将Circle 超类定义为被继承的ball 的方法
    this.superClass = Circle;
    // 在ball 对象上调用Circle 构造器，将作为Ball 构造器的参数所提供的值进行传递
    this.superClass(radius, xPosition, yPosition);
    // 设置ball 对象的color
    this.color = color;
}

// 将Circle 超类的实例赋给我们的Ball 类构造器原型
Ball.prototype = new Circle();

// 现在我们来建立一个Ball 的实例，然后检查它的属性
myBall = new Ball ( 0x445599, 16, 34, 5 );
trace(myBall.xPosition);      // 34
trace(myBall.area());        // 804.24...
trace(myBall.color);         // 447836
```

注意，*Ball* 中的 *superClass* 一词并不是保留的或者特殊的，它只是超类构造器函数的一个合适的名称。此外，*Circle* 的 *area()* 方法可以在 *Circle.prototype* 中定义。当在真实世界中开始用类和对象编程的时候，你会明显地注意到，ActionScript 提供来建立类继承和实现继承功能的工具非常灵活。你可能需要修改本章所描述的方法来适应你制作的特定的应用程序。

### 多态现象

继承也许带来了 OOP 的另外一个关键概念，多态现象。多态现象是一种特别的说法，意思是“多种形式”。它只意味着你让一个对象做什么，而将细节留给对象自己（也就是因人而异的意思）。我们举个例子来说明一下。假设你创建一个警察和小偷的游戏。屏幕上同时显示多个警察和小偷，以及几个无辜的旁观者，他们都可以按照不同的规则独立地移动。警察要抓小偷，而小偷要逃跑，旁观者的移动是任意的，他们惊慌失措。在这个游戏的代码中，假设我们创建一个对象类来表示每一种人：

```
function Cop() { ... }
function Robber() { ... }
function Bystander() { ... }
```

另外，我们创建一个超类 *Person*，类 *Cop*, *Robber* 和 *Bystander* 都要继承于它：

```
function Person() { ... }
Cop.prototype = new Person();
Robber.prototype = new Person();
Bystander.prototype = new Person();
```

Flash影片的每一个帧中，屏幕上的每一个人都应该按照类中制定的规则来移动。要达到这个目的，我们在每个类中定义方法 *move()*(*move()*方法对每个类来说都是自定义的)：

```
Person.prototype.move = function() { ... default move behavior ... }
Cop.prototype.move = function() { ... move to chase robber ... }
Robber.prototype.move = function() { ... move to run away from cop ... }
Bystander.prototype.move = function() { ... confused, move randomly ... }
```

在Flash影片的每个帧中，我们要让屏幕上的每一个人移动。为了管理所有的人，我们创建一个 *Person* 对象的管理数组。下面是 *persons* 数组的代码：

```
// 创建警察
cop1 = new Cop();
cop2 = new Cop();

// 创建小偷
robber1 = new Robber();
robber2 = new Robber();
robber3 = new Robber();

// 创建旁观者
bystander1 = new Bystander();
bystander2 = new Bystander();

// 创建一个包含警察、小偷和旁观者的数组
persons = [cop1, cop2, robber1, robber2, robber3, bystander1, bystander2];
```

在 Flash 影片的每一个帧中，我们调用函数 *moveAllPersons()*，它的定义如下：

```
function moveAllPersons() {
    for(var i=0; i < persons.length; i++) {
        persons[i].move();
    }
}
```

当 *moveAllPersons()* 被调用的时候，所有的警察、小偷和旁观者都会按照各自的规则来移动，这些规则是用 *move()* 方法在相关的类中制定的。这就是多态现象在现实中的反映——有共同特征的对象可以被组织到一起，但是他们有各自的特性。警察、小偷和旁观者有很多共同特征，由超类来表示。他们有共同的动作，比如知道如何移动。但是，他们可以以不同的方式实现共有的操作，也可以支持其他特殊类的操作和数据。多态现象允许不同的对象以统一的方式来对待。我们使用 *move()* 函数来让所有的人移动，即使每一个类的 *move()* 函数都是独特的。

### 确定一个类是否属于超类

假设有一个 *Shape* 类和一个 *Rectangle* 类，前者是后者的超类。要检查一个类是否为 *Shape* 的后代，必须调整例 12-6 中的方法，使用原型对象链（称为原型链）。例 12-9 的代码显示了这个方法。

#### 例 12-9：使用原型链

```
// 这个函数检查 theObj 是否为 theClass 的后代
function objectInClass(theObj, theClass) {
    while (theObj.__proto__ != null) {
        if (theObj.__proto__ == theClass.prototype) {
            return true;
        }
        theObj = theObj.__proto__;
    }
    return false;
}

// 建立 Rectangle 的一个新实例
myObj = new Rectangle();

// 现在检查 myRect 是否是从 Shape 继承的
trace (objectInClass(myRect, Shape)); // 显示: true
```

### 继承链的末尾

所有的对象都继承顶层 *Object* 类。因此，所有的对象都继承在 *Object* 构造器中定义的属性——也就是 *toString()* 和 *valueOf()* 方法。我们可以通过对 *Object* 类添加新的属性，而为影片中的每一个对象添加新属性。随着属性被添加到 *Object* 中，*prototype* 将会在整个类层次体系中增生扩散，包括所有的内部对象和 *movieclip* 对象！这种方式会产生一种真正的全局变量或方法。例如，在下面的代码中，我们将

在 *Object* 类的 *prototype* 中规定场景的高度和宽度尺度，然后就可以从任何影片剪辑中访问这些信息：

```
Object.prototype.mainstageWidth = 550;
Object.prototype.mainstageHeight = 400;
trace(anyClip.mainstageWidth);           // 显示: 550
```

*object.prototype* 属性不仅能由影片剪辑继承，也能被所有的对象继承，因此，我们自己定义的对象以及其他内置类（比如 *Date* 和 *Sound*）的实例，也能够继承赋给 *object.prototype* 的属性。

读者练习：我们可以对任何内置类添加新的属性和方法。试着将例 11-6 中不分大小写的字母排序函数作为一个继承方法添加到 *Array* 类。

## 比较 Java 的术语

在对类和类继承的讨论中，我们已经了解了定义在对象、类构造器和类原型中的属性。在 Java 和 C++ 中，对类和对象属性的不同类型有特定的名称。为了方便 Java 的编程者，表 12-1 对 Java 和 ActionScript 做了大致比较。

表 12-1 Java 和 ActionScript 属性的等价性

Java 描述	ActionScript	ActionScript 举例	
实例变量	对象实例中的局部变量	在类构造器函数中定义的属性，要复制给对象	function Square (side) {   this.side = side; }
实例方法	在对象实例上调用的方法	在类构造器函数的 <i>prototype</i> 对象上定义的方法，在实例上调用的时候自动通过 <i>prototype</i> 来访问	Square.prototype.area = squareArea; mySquare.area();
类变量	在一个类的实例中有相同值的变量	作为函数属性而在类构造器函数上定义的属性	Square.numSides = 4;
类方法	通过类来调用的方法	在类构造器函数上作为函数属性定义的方法	Square.findSmaller = function (square1, square2) { ... }

## 面向对象编程概要

类和继承为多个对象之间的信息共享开启了一片新的天地。面向对象编程是所有 ActionScript 的基础。不管你是否在脚本中使用类和对象的观念，理解其中包含的基本概念是理解 Flash 编程环境的一个关键。正如我们在本章的最后一节将看到的，我们对面向对象编程的信心一般说来在很大程度上会影响我们对 ActionScript 的信心。

要更进一步了解起源于 ECMA-262 的语言的面向对象编程知识，请参见 Netscape 的 JavaScript 文档，“Details of the Object Model”（对象模型的细节）：

*<http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm>*

David Flanagan 的经典著作，《JavaScript 权威指南》(O'Reilly & Associates 公司) 还提供了 JavaScript 中 OOP 的有关信息，很有参考价值。要得到从 Java 观点给出的 OOP 的介绍，参见 Sun 的“Object Oriented Programming Concepts”（面向对象编程概念）（摘自 Java 技术手册指南）：

*<http://java.sun.com/docs/books/tutorial/java/concepts>*

## 内置 ActionScript 类和对象

我们从第一章开始到现在，真的已经走过了很长的路。我们对 ActionScript 的第一次揭示是对动作面板中 + 按钮下的项目的简单讨论。从那以后，我们又学习了数据和表达式，以及操作符、语句和函数，刚才又讨论了类和对象的概念。现在可以在我们的 ActionScript 图画上画上最后的一笔了。

ActionScript 有多种语法工具：表达式包含数据；操作符对数据进行操作；语句给出指令；函数将指令组装成可以移动的命令。这些都是工具，但它们仅是工具而已。它们是我们用来构成脚本指令的文法。然而我们一直没有正面接触目标主题，到现在为止，我们完全知道了如何同 ActionScript 对话，但是我们没有什么可说的。ActionScript 中内置的类和对象填补了这个空白。

## 内置类

正如我们定义自己的类来描述和操作按照我们的规范所创建的对象一样，ActionScript 定义它自己的数据类。预制类的各种种类，包含 *Object* 类，都直接建立在 ActionScript 中。内置的类可以控制一个 Flash 影片的物理环境。

例如，内置 *Color* 类定义了可以检测或设置影片剪辑颜色的方法。要使用这些方法，我们首先用 *Color()* 类构造器创建一个 *Color* 对象，如下所示：

```
clipColor = new Color(target);
```

*clipColor* 是变量、数组元素或者对象属性，它有储 *Color* 对象。*Color()* 构造器有一个参数 *target*，它指定其颜色需要被设置或检测的影片剪辑的名称。

因此，假设有一个影片剪辑 *square*，我们要改变它的颜色。我们创建一个新的 *Color* 对象类，如下所示：

```
squareColor = new Color(square);
```

然后，要设置 *square* 剪辑的颜色，我们在 *squareColor* 对象上调用一个 *Color* 方法：

```
squareColor.setRGB(0x999999);
```

*setRGB()* 方法设置 *Color* 对象的 *target* 的 RGB 颜色，*target* 在本例就是 *square*，因此，前边的方法调用将会把 *square* 的颜色设置为灰色。

因为 *Color* 对象是内置的类，它可以直接设置影片剪辑的颜色。其他类可以让我们控制声音、数据和 XML 文档。我们会在第三部分中学习有关内置类的内容。

## 内置对象

和内置类一样，内置对象控制影片的环境。例如 *Key* 对象，它定义一系列的属性和方法，这些东西可以告诉我们计算机键盘的状态。要使用这些属性和方法，我们不实例化一个 *Key* 对象，只是直接使用 *Key* 对象。内置对象在 Flash 播放器起动的时候会由解释程序自动建立，并在影片播放过程中保持有效。

例如，调用 *Key* 对象的 *getCode()* 方法来显示当前按下的键的键控代码：

```
trace(Key.getCode());
```

下面，使用`isDown()`方法来检查空格键是否被按下，将空格键的键控代码作为参数：

```
trace(Key.isDown(Key.SPACE));
```

在第三部分中，我们要学习其他内置的类——*Math*, *Mouse* 和 *Selection*，它们提供对数学信息、鼠标指示器以及文本域选择情况的访问。

## 窍门揭示

学习编写合法的ActionScript代码只走完了学习Flash编程的一半路程。另一半是学习可用的内置类和对象，以及它们的属性和方法。我们会在第三部分中完成这个任务。但是，我们并不需要学习所有的类和对象。首先学习影片剪辑，然后按照需要进行扩展。渐渐地，你就会领悟到对于完成特定的工作来说什么才是最本质的。最重要的是：你要理解面向对象编程的不同结构。一旦知道了系统的规则，学习一个新的对象或者一个新的类就是一件简单的事情了，只不过看看它的方法和属性名称而已。

## 小结

我们已经走过了漫漫长路，我希望你仍然像我一样，对以后的路程充满热情。下面，我们将要讨论 ActionScript 中实用的、最重要的对象类——影片剪辑。

# 第十三章

## 影片剪辑

每一个 Flash 文档都包含一个场景（放置形状、文本和其他视觉元素的地方）以及一个主时间线，在时间线上定义场景内容随时间所发生的变化。场景（也就是主影片）可以包含单独的子影片，命名的影片剪辑（简称剪辑）。每一个影片剪辑有它自己独立的时间线和画布（场景就是主影片的画布），甚至能够包含其他影片剪辑。一个包含其他剪辑的剪辑被称为所包含剪辑的主剪辑或者父剪辑。

一个单独的 Flash 文档可以包含一个相关影片剪辑的层次。例如，主影片可以包含一个庞大的场景。一个包含动态人物的独立影片剪辑可以在场景上移动，表现人物的行走动作。人物剪辑里的另外一个影片剪辑可以独立地完成人物中的眨眼。当卡通人物中的独立元素一起播放的时候，它们会显得像是一个整体的内容。此外，每一个成分可以快速地响应其他成分——我们可以在人物停止移动的时候让眼睛开始眨动，或者当人物开始移动的时候让腿开始行走。

ActionScript 提供了影片剪辑的控制细节：我们可以播放一个剪辑或停止，在它的时间线内移动播放头，在程序中计划性地设置它的属性（比如尺寸、旋转、透明程度，以及在场景中的位置）以及将它作为真实的编程对象来操作。作为 ActionScript 语言的常见成分，影片剪辑可以被看成用来产生 Flash 的程序化生成内容的原料。例如，一个影片剪辑可以作为小游戏中的一个球或者一片桨，作为购物网站中的一个定货单，或者作为动画中背景声音的容器。在本章的最后，我们会将影片剪辑用作闹钟上的指针以及多项选择测试中的答案。

## 影片剪辑的对象性

在 Flash 5 中，影片剪辑可以像我们在第十二章中学到的那样，作为对象来进行操作。我们可以获取和设置剪辑的属性，也可以在剪辑上调用内置的或者自定义的方法。和其他对象不同，剪辑上所执行的操作可以在播放器中产生可见的或者可听的结果。

影片剪辑并不真是一种对象，它既没有 *MovieClip* 类构造器，也不能在代码中用一个对象常量来实例化一个影片剪辑。既然影片剪辑不是对象，那么它又是什么呢？它们属于一种类似对象的数据类型，称为 *movieclip*（我们可以在影片剪辑上执行 *typeof* 来证明这一点，它会返回串“*movieclip*”）。影片剪辑和真正的对象之间的主要区别是它们的分配（创建）方式以及解除分配（除去，或者说释放）的方式。要了解详细的内容，请参见第十五章。但是，不管这个技术如何，我们几乎总是将影片剪辑当做对象来看待。

那么，影片剪辑的对象性会对我们在 ActionScript 中使用影片剪辑时产生什么样的影响呢？最值得注意的是，它表示我们控制剪辑以及检查其属性的方法。影片剪辑可以通过内置方法进行直接的控制。例如：

```
eyes.play();
```

可以用点操作符来获取和设置一个影片剪辑的属性，正如可以访问任何对象的属性一样：

```
ball._xscale = 90;  
var radius = ball._width / 2;
```

影片剪辑中的一个变量只是剪辑的一个属性，可以使用点操作符来设置和获取变量的值：

```
myClip.myVariable = 14;  
x = myClip.myVariable;
```

子影片剪辑可以作为其父影片剪辑的对象属性来对待，因此，可使用点操作符来访问嵌套的剪辑：

```
clipA.clipB.clipC.play();
```

可使用保留的 `_parent` 属性来指向包含当前剪辑的剪辑：

```
_parent.clipC.play();
```

将剪辑作为对象为我们提供了无数简便语法，以及对播放的方便控制。但是，将剪辑作为对象也让我们把剪辑作为数据来管理。我们可以将一个影片剪辑存储在数组元素，或者变量中，甚至将剪辑的引用作为参数传递给函数！例如，下面的代码是一个函数，它将一个剪辑移动到屏幕上指定的位置：

```
function moveClip (clip, x, y) {  
    clip._x = x;  
    clip._y = y;  
}  
moveClip(ball, 14, 399);
```

在本章剩余的部分中，我们要学习将剪辑作为数据对象进行引用、控制以及操作的规则。

## 影片剪辑的类型

并非所有的影片剪辑的创建都是相同的。实际上，在 Flash 中有三种类型的剪辑：

- 主影片。
- 正规影片剪辑。
- 智能剪辑。

除了这三种正式的类型之外，还可以基于对正规影片剪辑的使用将其分为四类：

- 过程剪辑。
- 脚本剪辑。
- 链接剪辑。
- 种子剪辑。

后面这些非正式的分类并不是 ActionScript 中使用的正规形式，它们只是提供了一种方便的途径来处理影片剪辑的编程。下面，我们来看看每一种影片剪辑类型。

## 主影片

Flash文档的主影片包含基础时间线和出现在每个文档中的场景。主时间线是文档所有内容的基础，包括所有其他影片剪辑。我们有的时候将主影片称为主时间线、主影片时间线、主场景，或者简单叫做 *root*。

主影片的操作方法和正规影片剪辑非常类似，但是：

- 一个主影片不能从.swf文件中被删除（虽然一个.swf文件本身可以从Flash播放器中删除）。
- 下面的影片剪辑方法在主影片上调用的时候无效：*duplicateMovieClip()*, *removeMovieClip()*, *swapDepths()*。
- 事件处理器不能被添加到主影片上。
- 主影片可以通过内置的全局属性 *\_root* 和 *\_leveln* 来引用。

注意，每一个.swf文件只包含一个主影片，而Flash播放器中一次可以包含多个.swf文件，我们可以将多个.swf文档（因此有多个主影片）通过*loadMovie()*和*unloadMovie()*函数装载到*levels*的栈中，我们将在后面学习这个内容。

## 正规影片剪辑

正规影片剪辑是最常用、最基础的内容容器，它们包括视觉和听觉元素，还可以通过事件处理器对用户输入和影片的播放起反应。对熟悉DHTML的JavaScript编程人员来说，将主影片看成是和HTML文档对象类似的东西，将正规影片剪辑看成是文档的层对象会有所帮助。

## 智能剪辑

在Flash 5中推出的智能剪辑是一个包含用来在创建工具中自定义剪辑属性的图形用户界面的正规影片剪辑。智能剪辑一般由高级程序员来开发，它为缺少经验的Flash创作者提供一个在不知道剪辑代码是如何工作的情况下自定义影片剪辑行为的简便方法。我们将在第十六章讨论智能剪辑的细节内容。

## 过程剪辑

过程剪辑并不用来包容影片内容，只是用来重复地执行一个代码块。过程剪辑可以用 *enterFrame* 事件处理器或者我们在第八章中学习的时间线循环来建立。

过程剪辑是 ActionScript 中对 JavaScript 窗口对象的 *setTimeout()* 和 *setInterval()* 方法的非正式选择。

## 脚本剪辑

和过程剪辑一样，脚本剪辑不用来包含内容，而用来跟踪一些变量或者执行某些脚本的空影片剪辑。例如，可以使用一个脚本剪辑来包含对击键和鼠标事件起反应的事件处理器。

## 链接剪辑

链接剪辑是一种从影片库中既可以导入又可以导出的影片剪辑。导出和导入设置在每一个影片库中的剪辑链接选项中是可以选择的。在使用 *attachMovie()* 剪辑方法直接从符号库中动态生成一个剪辑实例的时候，我们经常使用链接剪辑，这方面的内容我们马上就会看到。

## 种子剪辑

在 *attachMovie()* 方法被引入到 Flash 5 中之前，我们使用 *duplicateMovieClip()* 函数，基于某些现存的剪辑（称为种子剪辑），来创建新的影片剪辑。种子剪辑是场景中只通过 *duplicateMovieClip()* 来进行复制的影片剪辑。随着 *attachMovie()* 的引入，就不再需要种子剪辑了。但是，在希望保留一个剪辑的事件处理器和复制过程中的转换的时候，仍然使用种子剪辑和 *duplicateMovieClip()*。

在一个大量使用 *duplicateMovieClip()* 来动态生成内容的影片中，经常可以在影片画布的外缘看到很多种子剪辑。种子剪辑只用来复制剪辑，因此不出现在场景中。

## 创建影片剪辑

我们经常将影片剪辑当成数据对象——用点操作符来设置它们的属性；用函数调用操作符（括号）来调用它们的方法；将它们存储在变量、数组元素和对象属性中。但是我们不能用创建对象的方法来创建影片剪辑。我们不能在代码中像描述一个对象一样来直接描述一个影片剪辑。不能用下面的影片剪辑构造器函数来生成影片剪辑：

```
myClip = new MovieClip(); // 虽然是很好的尝试，但它是无效的
```

我们只能在创建工具中亲手直接创建影片剪辑。一旦剪辑得到创建，就可以用诸如 *duplicateMovieClip()* 和 *attachMovie()* 这样的命令来为它建立新的、独立的复本。

## 影片剪辑符号和实例

正如所有的对象实例都基于一个或者其他的类一样，所有的影片剪辑实例都是基于模板影片剪辑的，也就是符号（symbol）（有的时候称为定义）。影片剪辑符号用来作为剪辑内容和结构的模型。我们在生成一个特定的剪辑对象之前，总是必须有一个影片剪辑符号。使用符号，我们可以用手动和程序化两种方式来创建在影片中渲染的剪辑。

在场景中渲染的影片剪辑称为一个实例。实例是单个的剪辑对象，可以用 ActionScript 来操作，符号是所有特定影片剪辑实例起源的模板。影片剪辑符号在 Flash 创建工具中创建。要建立一个新的、空白的剪辑，应执行下面的步骤：

1. 选择 Insert → New Symbol（新符号），就会出现 Symbol Properties（符号属性）对话框。
2. 在 Name（名称域）中，输入符号的标识符。
3. 选择 Movie Clip（影片剪辑）单选钮。
4. 点击 OK。

通常，下面要执行的步骤就是用影片剪辑的内容来填充符号的画布和时间线。一旦一个符号得到创建，它就存在于库中，等着我们用它来制作一个实际的影片剪辑实

例。但是，也可以将一批已经存在于场景中的形状和对象转换为影片剪辑符号。要这么做，我们执行下面的步骤：

1. 选择要进行转换的形状和对象。
2. 选择 Insert → Convert to Symbol (转换为符号)。
3. 在 Name 中，输入符号的标识符。
4. 选择 Movie Clip 单选钮。
5. 点击 OK。

我们用来创建新影片剪辑符号的形状和对象可以被新剪辑的未命名实例所代替。对应的影片剪辑符号会出现在库中，准备用来创建其他实例。

## 创建实例

要基于影片剪辑符号来创建新的实例，有三种方法。其中两种是程序化的，另外一种是手动的，要在 Flash 的创建工具中完成。

### 手动创建实例

可以使用 Flash 制作环境中的库来手动地创建影片剪辑实例。物理地将影片剪辑符号从库中拖到场景上，这样就生成了一个新的实例。一个这样创建的实例要通过实例面板手动地进行命名。（我们会在后面学习关于实例名称的更多内容。）如果你从来没有在 Flash 中接触过影片剪辑，请参考 Macromedia Flash 帮助中的 **Using Symbols and Instances**（使用符号和实例）。

### 用 `duplicateMovieClip()` 来创建实例

任何已经存在于 Flash 影片场景中的实例都可以用 ActionScript 来复制。然后，我们可以将单独的复本当作一个完全独立的剪辑。不管是程序创建的还是手动创建的实例都可以被复制。换句话说，复制一个复本是合法的。

实际上，用 `duplicateMovieClip()` 来复制实例有两种方法：

- 可以将 `duplicateMovieClip()` 作为一个全局函数来调用，使用下面的语法：

```
duplicateMovieClip(target, newName, depth);
```

*target*是我们想复制的实例名称的串; *newName*参数是一个串, 它为新实例指定标识符; *depth*是一个整数, 指定我们要将新的实例放在程序化生成剪辑栈中的什么地方。

- 还可以将 *duplicateMovieClip()*作为一个现有剪辑的方法来调用:

```
myClip.duplicateMovieClip(newName, depth);
```

*myClip*是我们要复制的剪辑的名称, *newName*和*depth*都和上面的方法一致。

通过*duplicateMovieClip()*进行复制的时候, 一个实例最初可以直接放到它的种子剪辑的顶层。因此, 复制后的第一个任务, 通常是将复制的剪辑移动到一个新的位置。例如:

```
ball1.duplicateMovieClip("ball2", 0);
ball2._x += 100;
ball2._y += 50;
```

其种子剪辑被ActionScript或者Flash中的制作工具变换(例如, 上色、旋转或者重新设置大小)的复制实例继承种子剪辑的初始变换。种子剪辑后来的变换不影响复制实例。同样, 每一个实例可以被单独转变。例如, 如果一个种子剪辑旋转了45度, 然后复制一个实例, 那么实例的初始旋转度就是45度:

```
seed._rotation = 45;
seed.duplicateMovieClip("newClip", 0);
trace(newClip._rotation); // 显示: 45
```

如果我们继续将实例旋转10度, 它的旋转角就是55度, 但是种子剪辑的旋转角仍然是45度:

```
newClip._rotation += 10;
trace(newClip._rotation); // 显示: 55
trace(seed._rotation); // 显示: 45
```

连续复制许多实例, 并稍微调整每一个复本的转变, 我们就可以得到有趣的复合转变(这个技术在例10-2的*load*事件中有所表现)。

使用*duplicateMovieClip()*通过ActionScript来复制剪辑, 与手动地将剪辑放到影片中相比有一些优点:

- 当剪辑出现在和程序执行有关的场景中时可以精确控制。
- 当剪辑从和程序执行有关的场景中被删除时可以精确控制。
- 赋给一个和其他复制剪辑相关的复制剪辑的层深度（这一点和 Flash 4 有很大关系，Flash 4 不允许影片的层栈被改变。）
- 复制一个剪辑的事件处理器。

这些能力可以提高对影片内容的程序化控制。太空船游戏就是一个显著的例子。当太空船的火力按钮被按下时候，需要复制一个飞弹影片剪辑，这个飞弹剪辑可以程序化地移动，然后放在影片中障碍物的后面，最后，在和敌舰相撞后被删除。手动剪辑就没有这种灵活性。用手动创建的剪辑，必须用时间线来预先指定该剪辑的产生和死亡，在 Flash 4 中，还不能改变剪辑的层。

### 用 `attachMovie()` 创建实例

和 `duplicateMovieClip()`一样，`attachMovie()`方法可创建一个影片剪辑实例，但是，和 `duplicateMovieClip()`不同，它不要求预先创建实例——它直接在影片库中从符号创建实例。为了使用 `attachMovie()` 来创建一个符号的实例，必须首先从库中导出这个符号。请看下面的步骤：

1. 在库中选择所要的符号。
2. 在 Library's Options (库选项) 菜单中，选择 Linkage (链接)，就会出现 Symbol Linkage Properties (符号的联接属性) 对话框。
3. 选择 Export This Symbol (导出该符号) 单选钮。
4. 在 Identifier (标识符) 域中，为剪辑符号输入一个独特的名称。名称必须是串（通常都和符号本身同名）但是应该和所有其他导出的剪辑符号不同。
5. 点击 OK。

一旦导出了剪辑符号，就可以将该符号的新实例添加到已经存在的剪辑中，用下面的语法调用 `attachMovie()` 就可以了：

```
myClip.attachMovie(symbolIdentifier, newName, depth);
```

*myClip*是要添加新实例的剪辑的名称。如果*myClip*被遗漏，*attachMovie()*就会将新的实例添加到当前剪辑(*attachMovie()*语句所在的剪辑)。*symbolIdentifier*参数是一个串，它包含要用来产生实例的符号，和库中链接选项的标识符域所指定的一样；*newName*也是串，它指定我们创建的新实例的标识符；*depth*是个整数，它指定要把新实例放到主剪辑分层栈的什么地方。

当将实例添加到另外的剪辑时，实例就被放在剪辑的中央，剪辑分层栈之间(我们马上就要讨论剪辑栈)。当将实例添加到文档主影片上的时候，实例被放到场景的左上角，坐标为(0, 0)。

## 实例名称

我们创建实例的时候，给它们赋了标识符，或者说是实例的名称，这可以让我们以后引用它们。注意这和正规对象的不同。当我们创建一个平常的数据对象(不是影片剪辑)的时候，必须将这个对象赋给一个变量或者其他数据容器，以便让该对象继续存在，我们也可以在以后用名称来引用它。例如：

```
new Object();           // 对象在创建之后立即死亡，不能引用它  
var thing = new Object(); // 对象引用存储在thing中，以后可以作为thing引用
```

影片剪辑实例不需要被存储在变量中以便引用。和普通的数据对象不同，剪辑实例在ActionScript中可以通过它们的实例名称在创建之后检索。例如：

```
ball._y = 200;
```

每一个剪辑实例名称都存储在它的内置属性`_name`中，它可以被获取，也可以被设置：

```
ball._name = "circle"; // 将ball的名称改为circle
```

当改变实例的`_name`属性时，以后指向该实例的全部引用都必须使用这个新的名称。例如，在前边的代码执行之后，`ball`引用就不存在了，我们要使用`circle`来检索这个实例。

实例最初得到它的实例名称的方式取决于它是如何创建的。程序化产生的实例用创建它的函数来命名。手动创建的实例通常在制作工具中通过实例面板赋给显式的实例名称，如下所示：

1. 选择场景上的一个实例。
2. 选择 Modify (修改) → Instance (实例)。
3. 将实例名称输入到 Name 域中。

如果手动创建的剪辑没有给予实例名称，它会在运行的时候由 Flash 播放器自动赋给一个。自动产生像 instance1, instance2, instance3...instancen 这样的实例名称，但是，这些名称对剪辑的内容没有什么有意义的描述（我们必须猜想自动产生的这些名称究竟是指什么）。

---

**警告：**因为实例名称是标识符，必须按照合法标识符的创建规则对它们进行调整，如第十四章中所描述的那样。更为特别的是，实例名称不应该以数字开头，也不应该包括连字符号和空格。

---

## 导入外部的影片

我们已经讨论了在单个文档中创建影片剪辑实例的内容，但是，Flash 播放器也可以同时显示多个.swf 文档。我们可以用 *loadMovie()*（作为全局函数或者影片剪辑方法）将一个外部的.swf 文件导入到播放器中，并把它放在剪辑实例中或者编号层次中的基础影片之上（也就是和基础影片有关的前景中）。通过对这些独立文件内容的管理，即可对下载过程有精确的控制。例如，假设有一个影片，它包含一个主导航菜单和五个子片段。在用户进入第五个片段之前，片段 1 到片段 4 必须已经完成下载。但是，如果我们将每个片段放在单独的.swf 文件中，片段就可以以任意的顺序来装载，可以让用户直接访问每一个片段。

当外在的.swf 文件被装载到一个层中的时候，它的主影片时间线就成为了该层的根时间线，它将代替在前面载入到这个层中的任何影片。当外部影片装载到剪辑中的时候也是如此，载入影片的主时间线将代替剪辑的时间线，将该剪辑现有的图形、声音和脚本卸载掉。

和 *duplicateMovieClip()* 一样，*loadMovie()* 可以用来作为一个独立的函数，也可以作为实例的方法。*loadMovie()* 的独立语法如下：

```
loadMovie(URL, location)
```

URL 指定要装载的外在.swf文件地址。location参数是一个串，它表示到现有剪辑的路径，或者是要容纳新的.swf文件的文档层（也就是说装载的影片要放在什么地方）。例如：

```
/  
loadMovie('circle.swf', '_level1');  
loadMovie("photos.swf", "viewClip");
```

因为一个影片剪辑引用在作为串使用的时候转换成了路径，location也可以作为影片剪辑的引用来提供，比如用\_level1来代替"\_level1"。但是使用引用的时候要小心。如果提供的引用并没有指向一个合法的剪辑，*loadMovie()*函数就不会产生期望的结果——它将外部.swf文件装载到当前的时间线。参见第三部分可以得到更多的细节内容，或者参见本章后面的内容。

*loadMovie()*的剪辑方法版本的语法如下：

```
myClip.loadMovie(URL);
```

作为方法来使用的时候，*loadMovie()*假设我们正在将外部.swf文件装载到*myClip*中，因此，不需要独立的*loadMovie()*函数所要求的 location参数，所以，我们只通过 URL参数提供对.swf的装载路径。自然，URL可以是一个局部文件名，比如：

```
viewClip.loadMovie("photos.swf");
```

被放置到一个实例中的时候，装载影片采用该剪辑的属性（比如剪辑的缩放、旋转、颜色变换等等）。

注意，*myClip*必须存在，以便*loadMovie()*能以它的方法形式来使用。例如，下面的代码中，如果\_level1为空，*circle.swf*的装载就会失败：

```
_level1.loadMovie('circle.swf');
```

## 装载影片的执行顺序

*loadMovie()*函数出现在一个语句块中的时候并不会立刻执行。实际上，它要等到块中所有其他语句都执行完了以后才执行。

---

注意：调用*loadMovie()*来将外部影片装载到播放器中的时候，我们不能在同一个语句块中访问这个外在影片的属性或者方法。

---

## 当 attachMovie()和 loadMovie()一起使用的时候

将一个外部.swf文件用 *loadMovie()*装载到一个剪辑实例中会产生惊人的结果——不能用 *attachMovie()*对这个剪辑添加实例。只要某个剪辑里装载了一个外部的.swf文件，就不能再从这个剪辑所起源的库中添加任何影片到这个剪辑。例如，如果 *movie1.swf*包含了一个名为 *clipA* 的实例，我们将 *movie2.swf* 装载到 *clipA* 中，我们就不能再从 *movie1.swf* 库中向 *clipA* 添加实例了。

为什么？ *attachMovie()*方法只在单个的文档中有效。也就是说，不能从一个文档的库中添加剪辑到其他的文档。当我们为剪辑装载了一个.swf文件时，我们就为这个剪辑组装了一个新的文档，也就是一个新的（不同的）库。以后还想从原来的文档中添加实例到这个剪辑就会失败，因为剪辑的库不再和原来的文档库相匹配了。但是，如果我们通过 *unloadMovie()*卸载剪辑里的文档，就又可以从它自己的文档库中添加影片到这个剪辑了。

类似的，将一个.swf文件用 *loadMovie()*装载到剪辑中就不能再通过 *duplicateMovieClip()*来复制这个剪辑。

因为 *loadMovie()*装载了一个外部文件（通常是通过网络），它的执行就是异步的。也就是说，*loadMovie()*可以在任何时间结束，这取决于文件转换的速度。因此，在访问一个装载影片之前，我们总是要检查影片是否全部转移到了播放器。我们通常用所谓的预装载器来完成这个工作——这是一种简单的代码，它在允许某些动作发生之前检查文件装载了多少。预装载可以用 *\_totalframes* 和 *\_framesloaded* 影片剪辑属性，以及 *getBytesLoaded()* 和 *getBytesTotal()* 影片剪辑方法来建造。参见第三部分示例代码中对应的条目，也可以参见例 10-4，它给出了如何使用 *data* 剪辑事件来构造预装载器的方法。

## 影片和实例的堆栈顺序

所有的影片剪辑实例和显示在播放器中的外部载入影片都按照一个可视的堆栈顺序存放，就跟一副纸牌类似。当实例或者外部载入.swf文件在播放器中产生重叠的时候，一个剪辑（两者中的“高”者）总是掩盖了其他剪辑（两者中的“低”者）。原则上很简单，但是主堆栈包含了所有的实例和.swf文件，它实际上被分为很多更小

的子栈。我们首先要分别查看这些子栈，观察它们是如何组合形成主堆栈的。(我们讨论的栈和在第十一章中所讨论的 LIFO 与 FIFO 栈并没有直接的关系。)

## 内部层堆栈

在 Flash 制作工具中手动创建的实例存在于一个称为内部层堆栈的栈中。这个栈的顺序由影片时间线上实际的层来控制。当两个不同时间线层上手动创建的实例发生重迭的时候，最上层的实例就遮盖了下层的实例。

而且，因为多个剪辑可能存在于同一个时间线层中，内部层堆栈中的每一个层实际上包含了它自己的小栈。存在上时间线一个层中的重叠剪辑通过 Modify (修改) → Arrange (排列) 命令被堆叠到制作工具中。

在 Flash 5 中，如果两个实例存在于同一个时间线上(也就是说，两个实例的 `_parent` 属性必须相等)，我们可以用 `swapDepths()` 方法来交换它们在内部层堆栈中的位置。Flash 5 之前的版本中，不能通过 ActionScript 来改变内部层堆栈。

## 程序化生成的剪辑堆栈

程序化生成的剪辑被堆叠在内部层堆栈中，和手动创建的实例相分离。每一个实例有它自己的程序化生成剪辑堆栈，容纳着通过 `duplicateMovieClip()` 和 `attachMovie()` 创建的剪辑。这些剪辑的堆栈顺序根据它们的创建方式而有不同。

### 通过 `attachMovie()` 生成的剪辑如何添加到堆栈

一个通过 `attachMovie()` 生成的新的实例总是堆叠到它所添加到的实例上面(也就是说相对在前)。例如，假设在影片的内部层结构中有两个剪辑 X 和 Y，X 所在的层在 Y 上面。现在假设添加一个新的剪辑 A 到 X 上，添加一个新剪辑 B 到 Y 上：

```
x.attachMovie('A', "A", 0);
y.attachMovie("B", "B", 0);
```

在我们的假定里，剪辑会按照从上到下的顺序出现：A, X, B, Y，如图 13-1 所示。

一旦一个剪辑产生，它还在内容之上提供了一个单独的空间，以程序化地生成剪辑。也就是说，可以将剪辑添加到已经添加的剪辑。

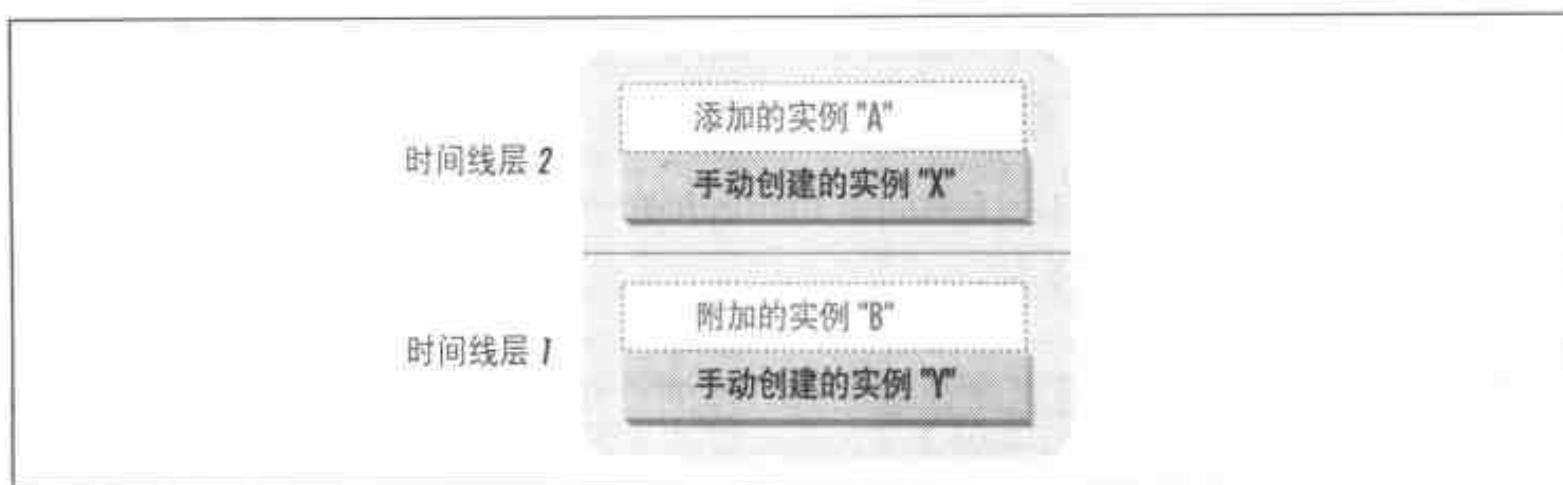


图 13-1 一个实例堆栈示意

添加到 Flash 文档影片的 `_root` 中的剪辑被放在 `_root` 影片程序化生成剪辑堆栈中，它会出现在 `_root` 影片的所有剪辑之前，甚至在那些包含程序化生成内容的剪辑之前。

让我们来扩展一下上面的例子。如果我们要将剪辑 C 添加到包含剪辑 X、Y、A 和 B 的影片的 `_root` 中，那么剪辑 C 会出现在所有其他剪辑之前。图 13-2 显示了扩展后的结构。

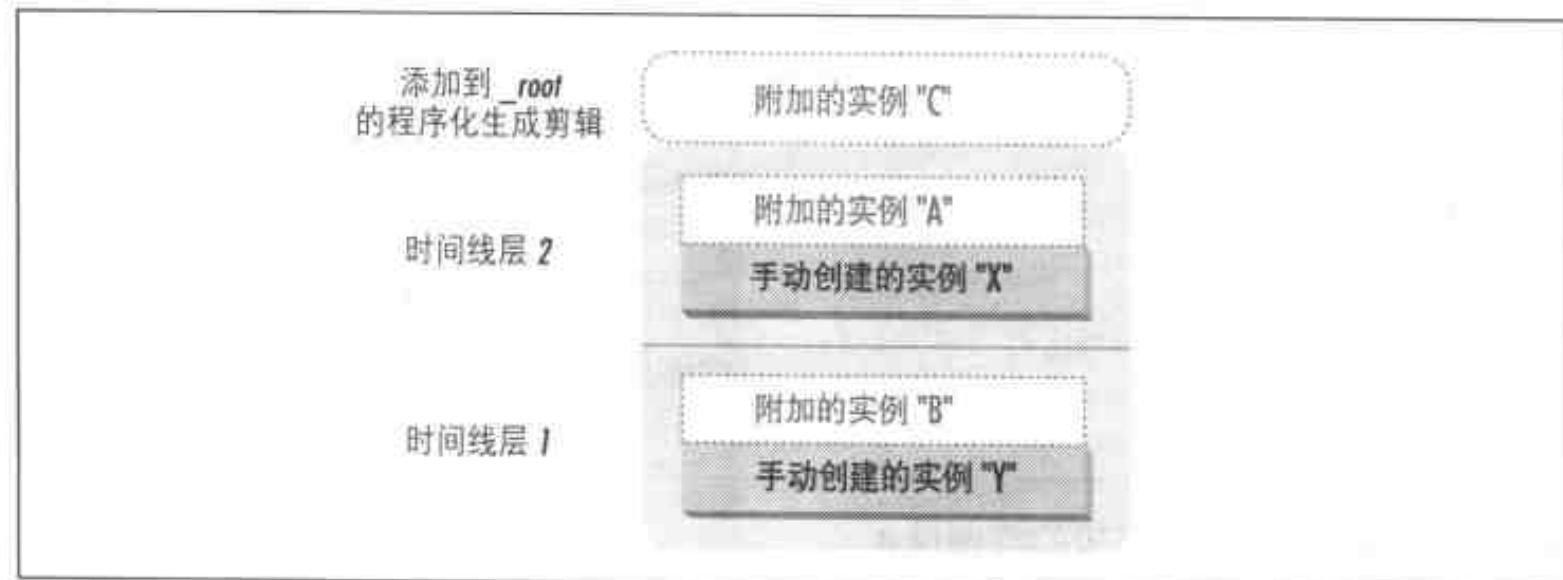


图 13-2 将一个实例添加到 `_root` 之后的实例栈

### 通过 `duplicateMovieClip()` 生成的剪辑如何添加到堆栈

每一个通过 `duplicateMovieclip()` 复制的剪辑都被放在程序化堆栈中，和实例的种子剪辑的创建方式相一致：

- 如果实例的种子剪辑是手动创建的（或者是用 `duplicateMovieClip()` 从手动创建的剪辑中复制的），那么新的实例就被放到 `_root` 上的堆栈里。
- 另一方面，如果实例的种子剪辑是用 `attachMovie()` 创建的，那么新的剪辑就被放到它的种子剪辑堆栈中。

我们回到上面的例子来看看它究竟是如何工作的。如果我们通过复制剪辑 X（手动创建的）而创建出剪辑 D，那么剪辑 D 就被放到 `_root` 上的堆栈，和剪辑 C 在一起。与此相似，如果我们通过复制剪辑 D（从手动创建的剪辑 X 而来）而创建出剪辑 E，那么 E 也被放到 `_root` 上的堆栈，和 C 与 D 在一起。但是如果我们通过复制剪辑 A（用 `attachMovie()` 创建的）而创建出剪辑 F，那么 F 就被放在 X 上的堆栈里，和剪辑 A 在一起。可以参见图 13-3 中的示意。

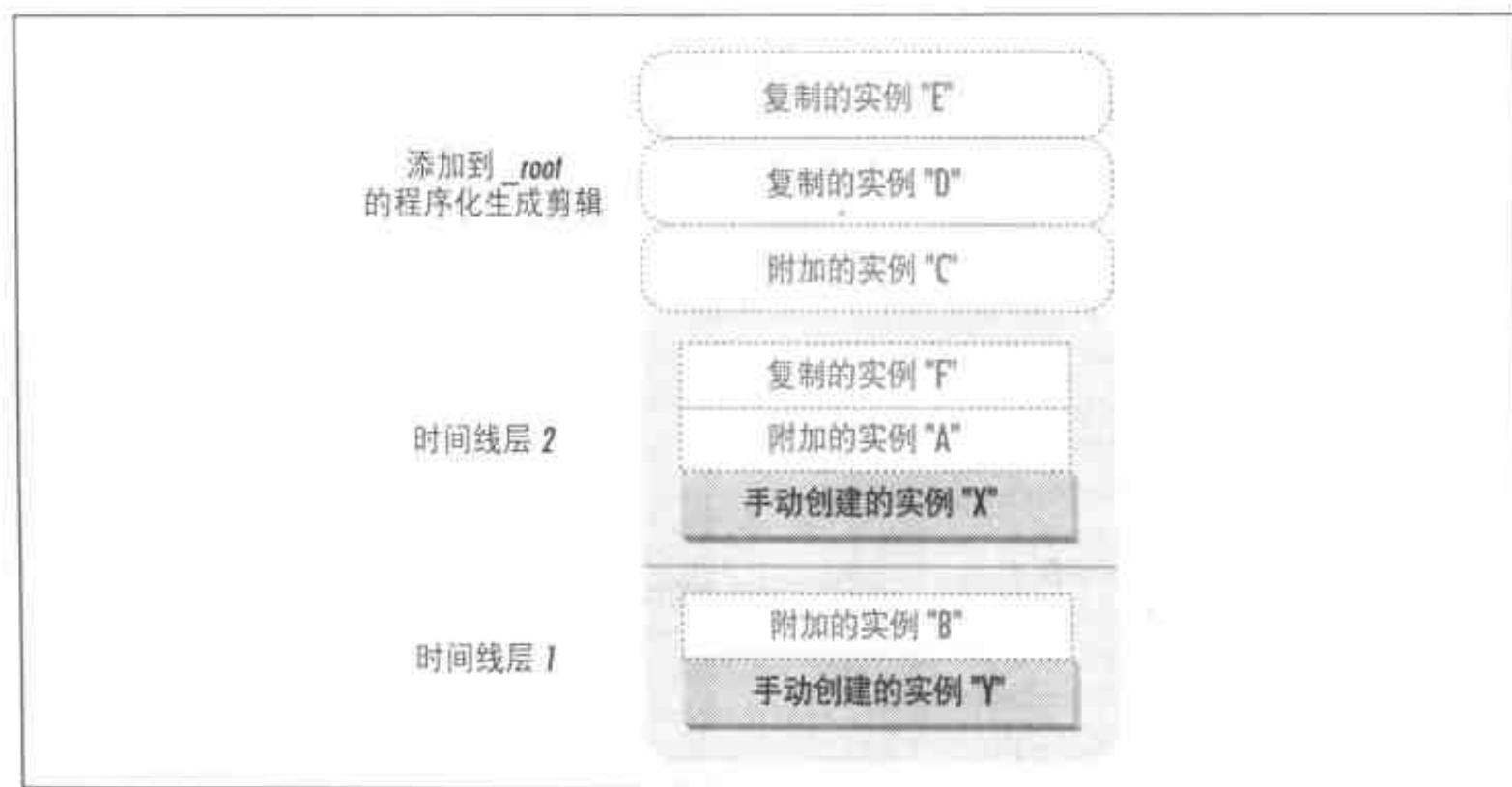


图 13-3 包含不同复制剪辑的实例堆栈

### 在程序化生成剪辑堆栈中给实例的深度赋值

你也许会感到疑惑，是什么确定了图 13-3 中剪辑 C、D 和 E 的堆栈顺序，或者剪辑 A 和 F 的顺序。程序化生成剪辑的堆栈顺序是由传递给 `attachMovie()` 或者 `duplicateMovieClip()` 函数的参数 `depth` 来确定的，也可以用 `swapDepths()` 函数在任何时候进行修改。每一个程序化生成剪辑的 `depth`（有的时候称为 z- 索引）确定了指定程序化生成堆栈中实例应处的位置。

剪辑的 *depth* 可以是任何整数，从底部开始度量，因此，-1 在 0 的下面，1 在 0 上面（也就是在前面），2 就更高，依此类推。当两个程序化生成剪辑占据了屏幕上的同一个位置的时候，*depth* 较大的一个就在另一个之前渲染。

层是单占的结构，每次在同一时间只有一个剪辑能占据堆栈中的一个层——将一个剪辑放到被占据的层中会代替（删除）该层原来的占据者。

剪辑的深度可以有间隙：可以让一个剪辑在深度 0，另外一个在深度 500，第三个在深度 1000。深度的赋值出现间隙并不会产生执行问题，也不会造成存储器的浪费。

## .swf 文档的 \_level 堆栈

除了内部层堆栈和程序化生成剪辑堆栈之外，还存在第三种（最后一种）堆栈，也就是文档堆栈（或者说文件级堆栈），它不控制实例的重叠，而控制通过 *loadMovie()* 装载到播放器中的所有 .swf 文件。

第一个装载到 Flash 播放器中的 .swf 文件被放在文档堆栈的最底层（用全局属性 *\_level0* 来表示）。如果在载入了第一个文档之后再载入任何其他的 .swf 文件到播放器中，就可以将它们设置到文档堆栈中 *\_level0* 上面的某一个层中，随意地将它们放在原来文档之前的某个位置上。文件级堆栈中高层文档中的所有内容将会在低层文档之前显示，不管每一个文档里的影片剪辑堆栈顺序如何。

正如程序化生成剪辑堆栈在每层只允许存在一个剪辑一样，文档堆栈每层也只允许有一个文档。如果将一个 .swf 文件装载到已经被占据的层级中，该层原来的占据者就会被新装载的文档所代替。例如，可以通过装载一个新的 .swf 文件到 *\_level0* 中而代替原来的文档。装载新的 .swf 文件到 *\_level1* 将会在视觉上遮盖 *\_level0* 中的影片，但是并不会把它从播放器中删除。

图 13-4 简要地给出了 Flash 播放器中所包含的多个文档之间的关系。

## 堆栈和执行顺序

影片剪辑的分层和时间线层次会影响到代码的执行顺序。规则如下：

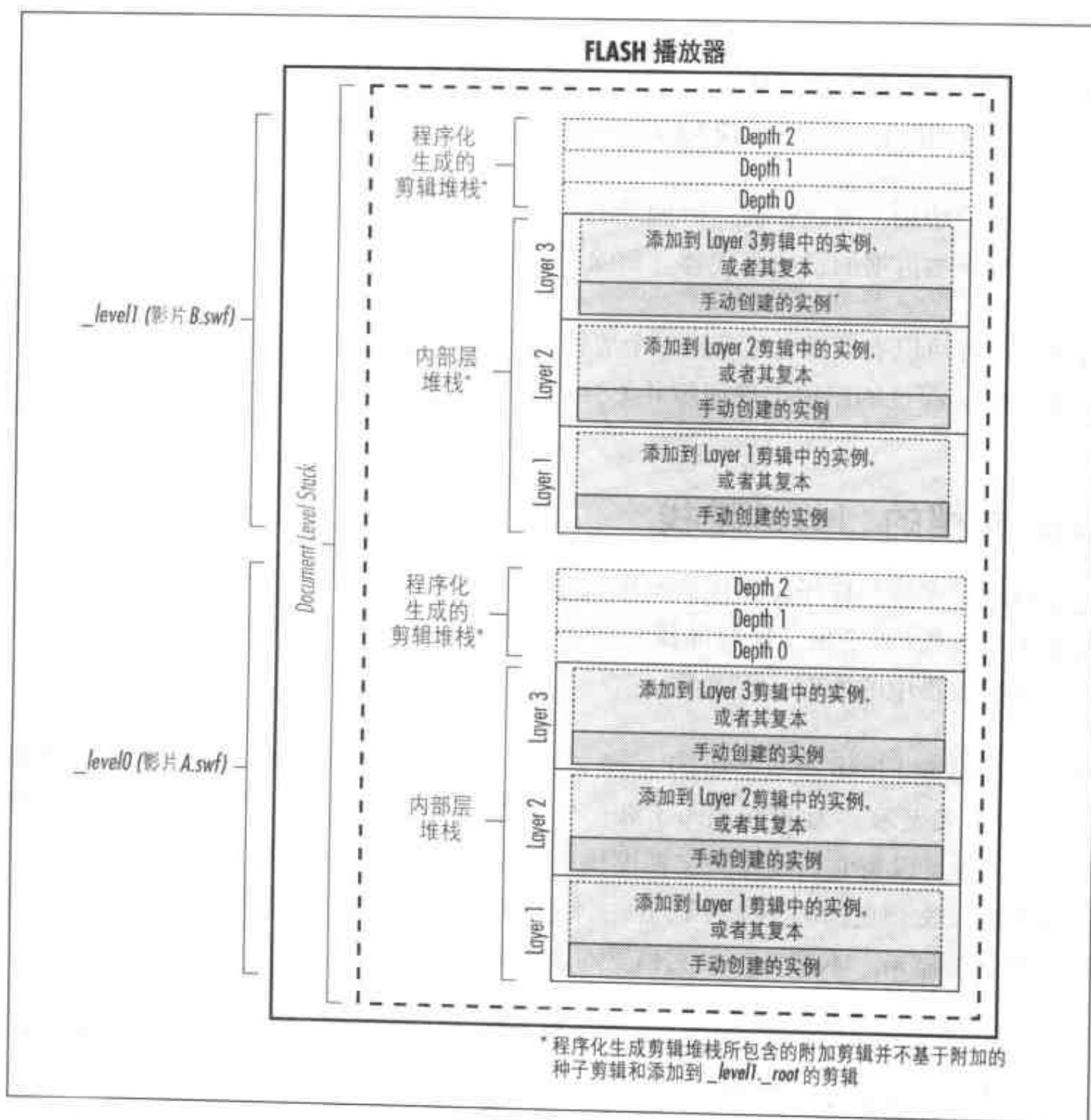


图 13-4 完整的 Flash 播放器影片剪辑堆栈

- 在不同时间线层中的帧上的代码总是按照从上到下的顺序来执行。
- 当手动创建的实例最初被载入的时候，它们的时间线上和`load`事件处理器中的代码按照 Flash 文档发布设置中所给出的装载顺序来执行——默认顺序为从下到上，也可以从上到下。

例如，假设有一个包含两层的时间线，`top` 和 `bottom` 两层，`top` 在层堆栈中位于 `bottom` 之上。我们将剪辑 X 放在 `top` 层上，将剪辑 Y 放在 `bottom` 层上。如果文档的装载顺序被设置为从下到上，那么剪辑 Y 中的代码就会在剪辑 X 中的

代码之前执行。如果文档的装载顺序被设置为从上到下，那么剪辑 X 中的代码就会在剪辑 Y 之前执行。这个执行顺序只针对 X 和 Y 首次出现的帧。

- 一旦载入，影片的所有实例都被设置一个执行顺序，它和载入顺序是相反的，最近添加到影片的实例，其代码总是首先得到执行。

对这些规则应该谨慎使用。层是变幻不定的，因此，你应该避免依靠相对的关系来产生代码。要争取让创建的代码能够安全地执行，而不依靠堆栈中剪辑的执行顺序。我们可以将所有的代码放在 *scripts* 层中，它处于每一个有代码的时间线最顶层，从而可避免由执行堆栈所引发的一些问题。

## 实例和主影片的引用

在前面的小节中，我们学习了如何在 Flash 播放器中对影片剪辑实例和外部 .swf 文件进行创建和分层。我们必须能够引用这些内容，才能用 ActionScript 来有效地控制它们。

我们在下面四种常见环境中引用实例和主影片：

- 获取或者设置剪辑或影片的属性。
- 创建或调用剪辑或影片的方法。
- 对剪辑或影片提供一些函数。
- 操作剪辑或者影片数据。例如，将它存储在变量中，或者将它作为参数传递给函数。

进行实例引用的环境和影片的环境尽管很相似，但是，我们用来建立引用的工具却是各种各样的。我们在本节剩余的部分就要介绍 ActionScript 中的实例和影片引用工具。

### 使用实例名称

在前边我们学到，影片剪辑是通过名称来引用的。例如：

```
trace(myVariable);           // 引用一个变量  
trace(myClip);             // 引用一个影片剪辑
```

为了直接引用一个实例（如前面的 *trace()* 例子所示），这个实例必须存在于代码所属的时间线上。例如，如果有一个名为 clouds 的实例放在文档的主时间线上，就可以从属于主时间线的代码中引用到 clouds，如下所示：

```
// 设置实例的一个属性  
clouds._alpha=60;  
// 调用实例上的一个方法  
clouds.play();  
// 将实例放进一个其他相关实例构成的数组中  
var backgrounds=[clouds,sky,mountains];
```

如果要引用的实例和代码不在同一个时间线上，就必须使用更多精细的语法，这些语法将在后面介绍。

## 引用当前实例或影片

在引用剪辑的时候并不总是需要使用实例的名称。属于实例时间线的某帧的代码可以直接引用该实例的属性和方法，而不用任何实例名称。

例如，要设置剪辑 cloud 的属性 *\_alpha*，可以将下面的代码放在 cloud 时间线的某帧上：

```
_alpha = 60;
```

类似的，要从 cloud 时间线的某帧上调用 cloud 的 *play()* 方法，可以简单地这样写：

```
play();
```

这个技术可以在任何时间线上使用，包括主影片的时间线。例如，如果下面的两个语句都属于 Flash 文档主时间线上的某一个帧，那么它们就是同义的。第一个语句隐式地指向主时间线，而第二个则通过全局属性 *\_root* 显式地指向主影片：

```
gotoAndStop(20);  
_root.gotoAndStop(20);
```

正如我们在第十章中所学到的，一个实例的事件处理器上的代码和时间线代码一样，也直接指向属性和方法。例如，我们可以将下面的事件处理器添加到 cloud 中。这

个处理器设置 cloud 的一个属性，然后调用 cloud 的一个方法，而不显式地指向 cloud 实例：

```
onClipEvent(load) {
    _alpha = 60;
    stop();
}
```

但是，不是所有的方法都可以隐式地指向一个影片剪辑。任何同对象的全局函数同名的影片剪辑方法（比如 *duplicateMovieClip()*），都必须用显式的实例引用来自调用。因此，在不太明确的时候就要使用显式引用。我们在后面将讨论有关方法和全局函数矛盾的更多问题。

### 使用 this 关键字的自引用

当我们要从某个实例时间线上的某帧，或者从它的某个事件处理器中显式地引用当前实例的时候，可以使用 *this* 关键字。例如，下面的语句如果属于 cloud 实例时间线上的某帧，它们就是同义的：

```
_alpha = 60;           // 对当前时间线的隐式引用
this._alpha = 60;      // 对当前时间线的显式引用
```

当我们能够直接指向某个剪辑的时候还要使用 *this* 来指向该剪辑有两个原因：如果不使用显式实例引用，特定的影片剪辑方法就会被解释程序误认为是全局函数；如果我们遗漏了 *this* 引用，解释程序就会以为我们要调用类似的全局函数，然后告诉，我们遗漏了“target”影片剪辑参数。要解决这个问题，我们使用 *this*，如下所示：

```
this.duplicateMovieClip('newClouds', 0);    // 在实例上调用一个方法
// 如果我们遗漏了 this 引用，就会引发错误
duplicateMovieClip("newClouds", 0);    // 噢！
```

使用 *this*，可以方便地将一个引用传递给在影片剪辑上操作的函数所在的时间线：

```
// 这里是一个对剪辑进行操作的函数
function moveTo(theClip, x, y) {
    theClip._x = x;
    theClip._y = y;
}

// 现在我们在当前时间线上调用它
moveTo(this, 150, 125);
```

如果我们要进行很多的面向对象编程工作，那么在使用 `this` 关键字来引用实例和影片的时候要小心。记住，在一个自定义的方法或者对象构造器内，`this` 的意思有很大不同，它不是对当前时间线的引用。参见第十二章可以获得相关的细节内容。

## 引用嵌套实例

正如我们在本章的介绍中所学到的，影片剪辑实例通常会嵌套在另外的实例中。也就是说，一个剪辑可以包含另外剪辑的实例，这个实例本身又包含其他剪辑的实例。例如，一个游戏中的 `spaceship` 剪辑可以包含 `blinkingLights` 剪辑或者 `burningFuel` 剪辑的实例。或者一个字符的 `face` 剪辑可以包含单独的 `eyes`, `nose` 和 `mouth` 剪辑。

在前面我们简要地介绍了如何在剪辑实例的层次中的任何一点开始进行向上或向下的导航，就和你对硬盘上的一系列子目录进行上下导航一样。现在，我们来看看其中的细节内容，以及一些实例。

我们首先来考虑如何指向一个嵌套在当前实例内的实例。当一个剪辑放在其他剪辑的时间线上时，它就成为那个剪辑的属性，可以像访问任何对象属性那样访问它（使用点操作符）。例如，假设将 `clipB` 放在 `clipA` 的画布上。要从 `clipA` 的时间线上访问 `clipB`，我们使用对 `clipB` 的直接引用：

```
clipB._x = 30;
```

现在假设 `clipB` 包含另外一个实例 `clipC`。要从 `clipA` 的时间线上的某一帧指向 `clipC`，我们将 `clipC` 作为 `clipB` 的一个属性来访问，如下所示：

```
clipB.clipC.play();
clipB.clipC._x = 20;
```

很不错，不是吗？这种方法的扩展性是无限的。因为每个放在其他剪辑时间线上的剪辑实例都变成其主剪辑的属性，可以用点操作符来分离各个实例，从而贯穿这个层次，如下所示：

```
clipA.clipB.clipC.clipD.gotoAndStop(5);
```

既然我们已经知道如何在实例层次中向下导航，现在，我们来看看如何向上导航，指向包含当前实例的实例或者影片。正如我们在前面所看到的，每一个实例都有一个内置的 `_parent` 属性，它指向包含这个实例的剪辑或主影片。我们使用 `_parent` 属性，如下所示：

```
myClip._parent
```

回忆一下我们刚才使用的 `clipA` 在主时间线上的例子，`clipB` 在 `clipA` 内，而 `clipC` 在 `clipB` 内，我们来看看如何使用 `_parent` 和点操作符来指向层次中不同的剪辑。假设下面的代码是放在 `clipB` 时间线的某一帧中的：

```
_parent          // 一个对 clipA 的引用  
this            // 一个对 clipB (当前剪辑) 的引用  
this._parent    // 对 clipA 的另外一个引用  
  
// 哦，我喜欢这种方式，我们再来试试……  
_parent._parent // 一个对 clipA 的父引用 (clipB 的祖父)，  
                // 也就是本例中的主时间线
```

注意，下面的代码虽然是合法的，但是如果用指向 `clipB` 的 `clipC` 属性的引用在层次中向下导航只是为了用 `_parent` 向上导航的话，就是一种没有必要的迂回。这些迂回的引用虽然没有必要，但是它的确显示了点号的灵活性：

```
clipC._parent          // 一个指向 clipB (当前时间线) 的迂回引用  
clipC._parent._parent // 一个指向主时间线的迂回引用
```

请注意如何使用点操作符来在剪辑层次中下降，以及如何使用 `_parent` 属性来上升。如果这些内容对你来说太生疏，你也许应该尝试在 Flash 中建立 `clipA`, `clipB`, `clipC` 层次，使用我们的例子中的代码。恰当的剪辑引用是 ActionScript 编程的基本技巧之一。

注意，剪辑的层次就像是一棵家族树。和那种每个后代都有两个双亲的两性繁殖不同，我们的剪辑家族树被扩展为无性的。也就是说，每一个后代只有一个单亲，这个单亲可以有很多孩子。任何剪辑（也就是树中的任何节点）都可以有一个且只有一个父亲（包含它的剪辑），但是可以有多个孩子（它所包含的剪辑）。当然，每一个剪辑的父亲依次也可以有一个单独的父亲，这意味着每一个剪辑只能有一个祖父（不像人类那样有四个祖辈）。参见图 13-5。

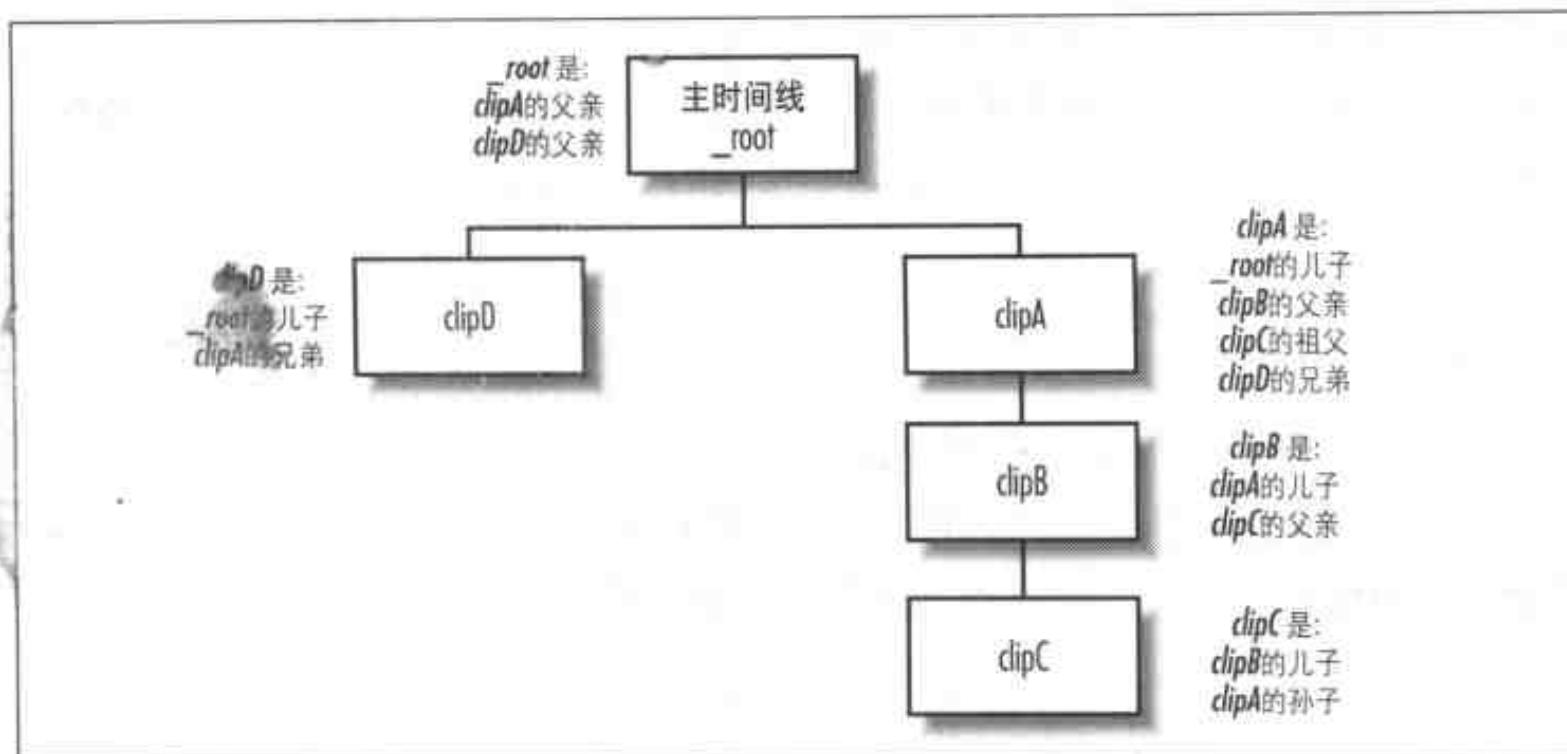


图 13-5 剪辑层次的示意

因此，不管你在家族树中有多深入，如果你按照相同的步骤向上走，就必定会回到原来开始的地方，如果你向下走是为了向上走的话，就毫无意义。但是，先在层次中上升、然后按照不同的路径向下走就不是没有意义了。例如，假设主时间线也包含 `clipD`，那么 `clipA` 和 `clipD` 就算是兄弟，因为它们两个的 `_parent` 都是主时间线。在这个例子中，可以从属于 `clipB` 的脚本这样来指向 `clipD`，如下所示：

```
_parent._parent.clipD // 这将指向 clipD, 主时间线(clipA的_parent)
// 的一个孩子, 因此它是 clipA的兄弟
```

注意，主时间线没有 `_parent` 属性（主影片是任何剪辑层次的顶层，不能被包含在其他时间线中），要指向 `_root._parent` 只会得到 `undefined`。

## 用 `_root` 和 `_levelIn` 指向主影片

既然我们已经了解了如何在层次中进行和当前剪辑相关的向上和向下的导航，现在，我们就来看看沿着绝对路径，或在存储于播放器文档堆栈的其他层级中的其他文档之间进行导航的方法。在前边的章节中，我们看到了这些技术在变量和函数上的应用，在这里，我们将学习它们是如何来控制影片剪辑的。

### 用 `_root` 指向当前层级的主影片

当一个实例在剪辑层次中嵌套得很深的时候，我们可以重复使用 `_parent` 属性来

在层次中上升，直到到达主影片时间线。但是，为了减少从深层嵌套剪辑指向主时间线的麻烦，我们也可以使用内置的全局属性 `_root`，它是指向主影片时间线的简短引用。例如，我们现在要播放主影片：

```
_root.play();
```

`_root` 属性被称为剪辑层次中某个已知点的绝对路径，因为它和与当前剪辑相关的 `_parent` 以及 `this` 属性不同，`_root` 属性不管从哪个剪辑开始引用都一样。下面这些引用都是等价的：

```
_parent._root  
this._root  
_root
```

因此，当你不知道一个给定的剪辑在层次内嵌套在什么地方的时候，你可以，也应该使用 `_root`。例如，考虑下面的层次结构，`circle` 是主影片时间线的孩子，而 `square` 是 `circle` 的孩子：

```
main timeline  
  circle  
    square
```

现在考虑下面这个在 `circle` 和 `square` 的帧中都有的脚本：

```
_parent._x += 10 // 将这个剪辑的父剪辑向右移动 10 个像素
```

当这些代码从 `circle` 内执行的时候，它会让主影片向右移动 10 个像素。当它从 `square` 内执行的时候，会让 `circle`（不是主影片）向右移动 10 个像素。为了让脚本不管在哪执行都可以使主影片向右移动 10 个像素，我们可以这样写：

```
_root._x += 10 // 将主影片向右移动 10 个像素
```

此外，`_parent` 属性如果在主时间线上就是非法的，而使用 `_root` 的脚本版本在主时间线上出现的时候却是合法的。

`_root` 属性可以同普通的实例引用结合使用，以在一个嵌套剪辑层次中下降：

```
_root.clipA.clipB.play();
```

以 `_root` 开头的引用不管在文档的什么地方使用，都指向同一个开始点，完全没有必要猜测。

## 用 `_leveln` 在播放器中引用其他文档

如果在 Flash 播放器的文档堆栈中载入了多个.swf文件，就可以用内置的全局属性系列`_level0`到`_leveln`来指向多个文档的主时间线，`n`表示我们想要引用的文档的层级。

因此，`_level0`表示文档堆栈中最底层的文档（更高层级的文档将在前面渲染）。除非一个影片是通过`loadMovie()`被装载到`_level0`中的，否则`_level0`就在播放开始的时候被最初装载的影片所占据。

下面的例子播放播放器文档堆栈中层级 3 里的文档主影片时间线：

```
_level3.play();
```

和`_root`属性一样，`_leveln`属性可以通过点操作符和普通实例引用结合使用：

```
_level1.clipA.stop();
```

正如对`_root`的引用一样，对`_leveln`属性的引用被称为绝对引用，因为它们会从文档中的任何点到达相同的目的地。

注意，`_root`和`_leveln`并不相同。`_root`属性总是当前文档的主时间线，不管当前文档所处的文档层级为何；而`_leveln`属性是对指定文档层级的主时间线的引用。例如，假设我们将代码`_root.play()`放在`myMovie.swf`中，当我们装载`myMovie.swf`到层级 5 的时候，我们的代码就播放`_level5`的主影片时间线。相反，如果我们将代码`_level2.play()`放到`myMovie.swf`中，然后将`myMovie.swf`装载到层级 5，那么我们的代码就播放`_level2`的主影片时间线，而不是`_level5`的。当然，从层级 2 中，`_root`和`_level2`是等价的。

## 用插入目标路径建立实例引用

当影片的实例结构变得非常复杂的时候，要构成对影片剪辑和主影片的引用可能非常费力。我们不能总是回忆一系列剪辑准确的层次结构，因此，我们应该结束那种只是为了确定嵌套结构而频繁地在制作工具中选择和编辑剪辑的工作。ActionScript 编辑器提供了一个插入目标路径工具（如图 13-6 所示），它可以产生剪辑的可视引用，从而减轻手动创建的负担。

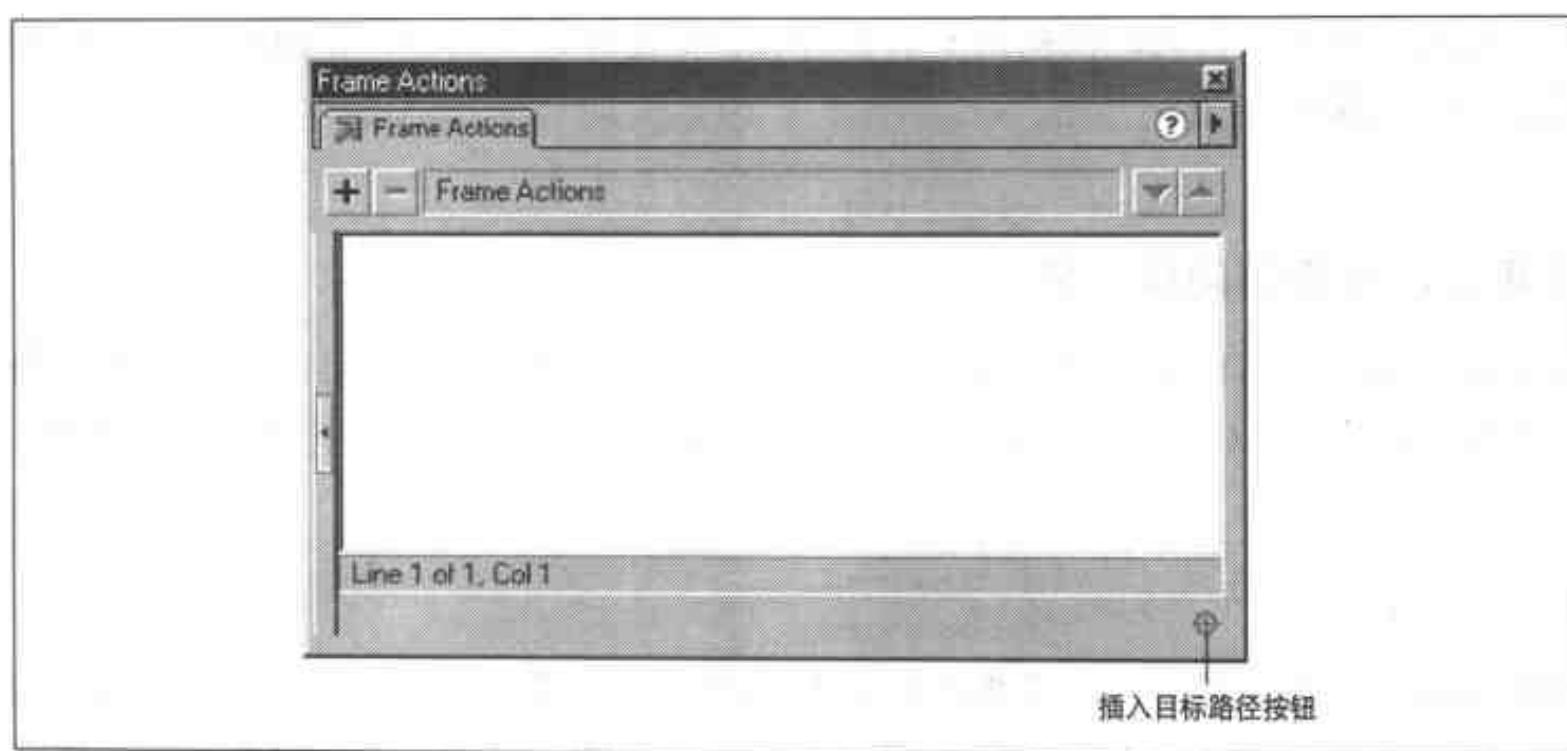


图 13-6 插入目标路径按钮

使用插入目标路径，要执行下面的步骤：

1. 将光标放在要插入剪辑引用的代码中。
2. 点击 Insert Target (插入目标路径) 按钮，如图 13-6 所示。
3. 在 Path 对话框中，选择你要引用的剪辑。
4. 选择是否插入绝对引用或相对引用。绝对引用是从 \_root 开始的，相对引用指的是和包含当前代码的剪辑相关的目标剪辑的引用。
5. 可如果你要导出 Flash 4 格式的文件，选择 Slashes Notation (斜线符号) 按钮，以与 Flash 4 兼容。（Dot Notation (点符号) 按钮是默认选项，它构成的引用对 Flash 4 无效。）参见表 2-1。

插入目标路径工具并不会产生在剪辑层次中上升的引用，也就是说，这个工具不能用来指向包含当前剪辑的剪辑（除非从 \_root 开始路径，然后往下走）。要创建在剪辑层次中上升的引用，必须在代码中使用 \_parent 属性手动地输入对应的引用。

## 对剪辑对象的动态引用

通常，我们知道要操作的指定剪辑或影片的名称，但是有的时候，我们也会需要控制不知名的剪辑。例如，我们也许想要用一个循环来缩小一系列的剪辑，或者建立

一个按钮，每次点击的时候指向不同的剪辑。要处理这些情况，必须在运行的时候动态地创建剪辑引用。

## 使用数组元素访问操作符

正如我们在第五章以及第十二章中看到的那样，一个对象的属性可以通过点操作符来获取，或者通过数组元素访问操作符`[]`来获取。例如，下面的两个语句是等价的：

```
myObject.myProperty = 10;  
myObject["myProperty"] = 10;
```

数组元素访问操作符有一个重要的特征是点操作符所没有的，它让我们（实际上是要求我们）用串表达式而不是标识符来指向一个属性。例如，下面的串连接表达式就是属性`myProperty`的合法引用：

```
myObject["myProp" + "erty"];
```

我们可以用相同的技术来动态地创建实例和影片引用。我们已经学过，剪辑实例是作为它们的父剪辑的属性来存储的。在前边，我们用点操作符来指向这些实例属性。例如，从主时间线开始，我们可以指向`clipB`，它嵌套在另外一个实例`clipA`中，如下所示：

```
clipA.clips;           // 指向 clipA 中的 clips  
clipA.clipB.stop();   // 调用 clipB 上的方法
```

因为实例是属性，我们也可以合法地用`[]`操作符来指向它们，比如：

```
clipA["clipB"];        // 指向 clipA 中的 clipB  
clipA['clipB'].stop(); // 调用 clipB 上的方法
```

注意，当我们使用`[]`操作符来指向`clipB`的时候，我们将`clipB`的名称作为一个串，而不是一个标识符。串引用可以是任何合法的、可以产生串的表达式。例如，下面的对`clipB`的引用就包含了串的连接运算：

```
var clipCount = "B";  
clipA['clip' + clipCount];           // 指向 clipA 中的 clipB  
clipA['clip' + clipCount].stop();    // 调用 clipB 上的方法
```

我们可以动态地创建剪辑引用来指向一系列连续的命名剪辑：

```
// 停止clip1,clip2,clip3和clip4
for (var i = 1; i <= 4; i++) {
    _root['clip' + i].stop();
}
```

真是太精彩了！

## 将剪辑的引用存储在数据容器中

本章开始的时候我们说过，影片剪辑是 ActionScript 有力的数据对象。我们可以将影片剪辑实例的引用存储在变量、数组元素或者对象属性中。

回忆前边的嵌套实例层次（clipC 嵌套在 clipB 中，clipB 又嵌套在 clipA 中）的例子，它是放在一个文档的主时间线上的。如果将这些不同的剪辑存储在数据容器中，就可以用容器而不是剪辑的显式引用来动态地控制它们。例 13-1 给出的代码是放在主时间线上的某帧中的，它使用数据容器来存储和控制实例。

### 例 13-1：将剪辑引用存储在变量和数组中

```
var x = clipA.clipB;          // 将 clipB 的引用存储在变量 x 中
x.play();                      // 播放 clipB

// 现在我们将剪辑存储在一个数组的元素中
var myClips = [clipA, clipA.clipB, clipA.clipB.clipC];
myClips[0].play();            // 播放 clipA
myClips[1]._x = 200;          // 将 clipB 放在场景左边界的 200 像素处

// 用一个循环来停止数组中所有的剪辑
for (var i = 0; i < myClips.length; i++) {
    myClips[i].stop();
}
```

通过将剪辑引用存储在数据容器中，我们就可以在操作这些剪辑（比如播放、旋转或者终止）的时候不用知道或影响文档的层次结构。

## 用 for-in 来访问影片剪辑

在第八章中，我们学习了如何使用 *for-in* 循环来列举对象的属性。一个 *for-in* 循环的循环变量自动扫描所有的对象属性，因此，这个循环对每一个属性都执行一次：

```
for (var prop in someObject) {
    trace("the value of someObject." + prop + " is " + someObject[prop]);
}
```

例 13-2 展示了如何使用 *for-in* 循环来列举给定时间线内的所有剪辑。

#### 例 13-2：寻找时间线上的影片剪辑

```
for (var property in myClip) {
    // 检查myClip的当前属性是否为一个影片剪辑
    if (typeof myClip[property] == "movieclip") {
        trace("Found instance: " + myClip[property]._name);

        // 现在为剪辑做点什么
        myClip[property]._x = 300;
        myClip[property].play();
    }
}
```

*for-in* 循环为我们在访问由指定剪辑实例或主影片所包含的剪辑时带来了巨大的方便。使用 *for-in*，不管我们是否知道剪辑的名称，也不管剪辑是手动创建的还是程序化生成的，我们都可以控制任何时间线上的任何剪辑。

例 13-3 给出了前边例子的递归版本。它找出时间线上的所有剪辑实例，将剪辑实例加到所有嵌套时间线上。

#### 例 13-3：递归地寻找时间线上的所有子剪辑

```
function findClips (myClip, indentSpaces) {
    // 用空格在连续的行中缩进子剪辑
    var indent = " ";
    for(var i = 0, i < indentSpaces; i++) {
        indent += ' ';
    }
    for(var property in myClip) {
        // 检查当前的myClip属性是否为影片剪辑
        if(typeof myClip[property] == "movieclip") {
            trace(indent + myClip[property]._name);
            // 检查这个剪辑是否为其他任何剪辑的父亲
            findClips(myClip[property], indentSpaces+4);
        }
    }
}
findClips(_root, 0); // 从主时间线往下寻找所有的剪辑实例
```

要知道函数递归的更多信息，请参见第九章。

### \_name 属性

正如我们在前面所学到的，每个实例的名称都被存储为内置属性 `_name` 中的串。我

我们可以使用这些属性（正如在例13-2中看到的那样）来确定当前剪辑的名称或者实例层次中其他剪辑的名称：

```
_name; // 当前实例的名称  
_parent._name // 包含当前剪辑的剪辑的名称
```

`_name` 属性在我们想要按照剪辑的标识符来执行条件性操作的时候非常方便。例如，我们在 `seedClip` 剪辑载入的时候对它进行复制：

```
onClipEvent (load) {  
    if (_name == "seedClip") {  
        this.duplicateMovieClip("clipCopy", 0);  
    }  
}
```

通过精确地检查 `seedClip` 的名称，可以防止无限递归——如果没有条件语句，每个复制剪辑的 `load` 处理器会让剪辑复制自身。

### `_target` 属性

每一个影片剪辑实例都有内置的 `_target` 属性，它是一个串，用来指定剪辑的绝对路径，路径中使用了不被支持的 Flash 4 “斜线” 符号。例如，如果 `clipB` 放在 `clipA` 内，而 `clipA` 是放在主时间线上的，那么这些剪辑的 `_target` 属性如下所示：

```
_root._target // 包含: '/'  
_root.clipA._target // 包含: "/clipA"  
_root.clipA.clipB._target // 包含: "/clipA/clipB"
```

### `targetPath()` 函数

`targetPath()` 函数返回一个串，串中包含剪辑的绝对引用路径，用点符号来表示。

`targetPath()` 函数是 Flash 5 语法中 `_target` 的等价物。它的格式为：

```
targetPath(movieClip)
```

`movieClip` 是我们要获取其绝对引用的标识符。下面的例子使用当前例子中的层次：

```
targetPath(_root); // 包含: "_level0"  
targetPath(_root.clipA); // 包含: "_level0.clipA"  
targetPath(_root.clipA, clipB); // 包含: "_level0.clipA.clipB"
```

*targetPath()* 函数给了我们剪辑的完整路径，*\_name* 属性只给了我们剪辑的名称。（这和有一个完整的文件路径和只有文件名称的情况是类似的。）因此，可以使用 *targetPath()* 来构成控制剪辑的代码，它不仅仅基于名称，而且还有它们的位置。例如，可以创建一个一般的导航按钮，通过检查它的 *targetPath()*，而设置它的颜色使其与它所在的内容部分相匹配。参见第三部分 *targetPath()* 的实际范例。

## Tell Target 将何去何从？

在 Flash 4 中，*Tell Target* 是引用影片剪辑的主要工具。*Tell Target* 是一种笨拙的工具，被 Flash 5 中推出的更多精致的对象模型所淘汰。*Tell Target* 函数已经不被支持了（即从使用需求中退出了）。虽然我们在 Flash 4 方式下仍然使用 *tellTarget()* 函数来编码，但是 *tellTarget()* 以后很可能不再使用。

考虑下面的代码，它使用 *Tell Target* 来播放一个名为 *closingSequence* 的实例：

```
Begin Tell Target ('closingSequence')
    Play
End Tell Target
```

在 Flash 5 中，我们调用更为方便易懂的 *play()* 方法来播放 *closingSequence* 实例：

```
closingSequence.play();
```

*Tell Target* 也可以在一个代码块中对某个实例执行多个操作，比如：

```
Begin Tell Target ('ball')
    (Set Property: ("ball", x Scale) = '5')
    Play
End Tell Target
```

在 Flash 5 中，第六章所描述的 *with()* 语句是执行相同操作的恰当方法：

```
with (ball) {
    _xscale = 5;
    play();
}
```

参见附录三可以获得关于不被支持的 Flash 4 ActionScript 项目以及 Flash 5 中的首选设置的更多内容。

## 删除剪辑实例和主影片

我们已经学习了创建和引用影片剪辑，现在，我们来看看如何将它们变成无数可以再度利用的电子（换句话说，就是将它们打发掉）。

我们创建一个实例或者影片的方式决定了我们后来删除这些实例或影片的技术。我们可用 *unloadMovie()* 和 *removeMovieClip()* 显式地删除影片和实例。另外，我们可以通过装载、添加或者复制一个新的剪辑到原剪辑的位置，而隐式地除去一个剪辑。我们来分别看看这些技术。

### 用 *unloadMovie()* 来处理实例和层级

内置的 *unloadMovie()* 函数可以删除任何剪辑实例或者主影片——包括手动创建的和通过 *loadMovie()*, *duplicateMovieClip()* 以及 *attachMovie()* 创建的。它可以作为全局函数或者方法来调用：

```
unloadMovie(clipOrLevel);      // 全局函数  
clipOrLevel.unloadMovie();    // 方法
```

按照全局函数的形式，*clipOrLevel* 是一个串，它表示到达被卸载的剪辑或者层级的路径。由于自动的值转换作用，*clipOrLevel* 也可以是一个影片剪辑引用（在用做串的时候，影片剪辑被转化为路径）。在用做方法的时候，*clipOrLevel* 必须是对一个影片剪辑对象的引用。*unloadMovie()* 的准确行为按照它是用在层级中还是实例上而有所不同。

#### 对层级使用 *unloadMovie()*

应用在文档堆栈（例如 *\_level0*, *\_level1*, *\_level2*）的层级上时，*unloadMovie()* 将把目标层级和该层级所包含的影片完全删除。后来对该层级的引用将产生 *undefined*。删除文档层级是 *unloadMovie()* 函数最普通的用法：

```
unloadMovie("_level1");  
_level1.unloadMovie();
```

## 对实例使用 `unloadMovie()`

应用在实例（不管是手动创建的还是程序化创建的）中时，`unloadMovie()`将删除剪辑的内容，但是它不删除剪辑本身！剪辑的时间线和画布被删除，但是空的外壳还留在场景上。这个空壳仍然可以被引用，直到实例通过`removeMovieClip()`被永久删除（或者直到实例所处的帧死亡）。此外，空壳中任何事件处理器都还会存在。

这种实例的部分删除会产生一个有趣的可能性：它让我们保持一个一般的剪辑容器，其内容可以被`loadMovie()`和`unloadMovie()`重复更换。例如，我们完全可以在一个名为`clipA`的实例上正当地调用下面的函数系列（虽然在真正的应用程序中，这样的语法要包含恰当的预装载代码）：

```
clipA.loadMovie('section1.swf');           // 将一个文档载入到 clipA  
clipA.unloadMovie();                      // 卸载文档，保持 clipA 的完整  
clipA.loadMovie('section2.swf');           // 将另外一个文档装入 clipA
```

使用该方法要注意：用在一个实例上的时候，`unloadMovie()`将删除包含在该实例中剪辑的所有自定义属性，物理属性（比如`_x`和`_alpha`）会继续存在，但是自定义的变量和函数都会丢失。

---

**警告：**如果使用`unloadMovie()`的全局函数形式，将一个不存在的剪辑或者层级实例作为它的参数，那么调用`unloadMovie()`函数的剪辑本身就会被卸载。

---

例如，如果`_level1`是未定义的，在`_level0`的主时间线上使用下面的代码，那么`_level0`就会被卸载：

```
unloadMovie(_level1);
```

这种结果是有一定的逻辑的，我们要在后面讨论。可以在指定`unloadMovie()`的`clipOrLevel`参数时使用串，或者在卸载之前明确地检查`clipOrLevel`是否存在，以避免这种问题。下面对每一种方法给出一个示例：

```
unloadMovie("_level1");                  // 将 clipOrLevel 作为串来指定  
if (_level1) {                          // 明确检查以确保 level 存在  
    unloadMovie(_level1);  
}
```

## 使用 `removeMovieClip()` 来删除实例

要从播放器中删除添加或者复制的实例，可以使用 `removeMovieClip()`。注意，`removeMovieClip()` 只对添加或者复制的实例起作用。它不删除手动创建的实例或者主影片。和 `unloadMovie()` 一样，`removeMovieClip()` 可以使用方法和全局函数形式（虽然语法不同，但结果是一样的）：

```
removeMovieClip(clip)      // 全局函数  
clip.removeMovieClip()    // 方法
```

在全局函数形式下，`clip` 是一个串，它表示要删除的剪辑的路径。由于自动的值转换作用，`clip` 也可以是一个影片剪辑引用（在用做串的时候，影片剪辑被转化为路径）。在方法形式里，`clip` 必须是对影片剪辑对象的引用。

和 `unloadMovie()` 不同，通过 `removeMovieClip()` 删除一个实例将完全删除整个剪辑对象，不会留下什么空壳或者痕迹，属性也都荡然无存了。当执行 `clip.removeMovieClip()` 的时候，以后对 `clip` 的引用就会产生 `undefined`。

## 删除手动创建的实例

在 Flash 制作工具中手动创建的剪辑实例其生命周期也是有限的——当播放头进入一个不包含它们的帧中时，它们就被删除了。因此，手动创建的影片剪辑生活在全能空白关键帧所带来的恐惧中。

记住，当一个影片剪辑从时间线上消失的时候，它作为一个数据对象就不存在了。其中定义的所有变量、函数、方法和属性都会丢失。因此，如果想让一个剪辑的信息或者函数继续存在，就应该留神手动的剪辑删除，并要保证剪辑所在的帧的活动周期在我们还需要剪辑信息的时候一直存在。（实际上，要完全避免这种错误，应该在主影片时间线的帧中添加最持久的代码。）要在剪辑还出现在时间线上的时候隐藏该剪辑，只要将剪辑放在场景中看不见的区域就可以了，并且将剪辑的 `_visible` 属性设置为 `false`。将剪辑的 `_x` 属性设置为一个非常大的正数或者非常小的负数也可以让它在用户的视野中消失，但是又不被从存储器中删除。

## 内置影片剪辑属性

和内置属性非常少的一般 *Object* 类对象不同，每一个影片剪辑配备了大量的内置属性。这些属性描述了剪辑的物理特征，可以对其进行修改。它们是 ActionScript 程序员工具箱中的基本工具。

所有的内置影片剪辑属性名称都以下划线开头，将它们同用户定义或者自定义的属性区分开。内置属性的格式为：

*\_property*

内置属性名称可以为小写的形式。但是，因为 ActionScript 中的标识符是不区分大小写的，因此也可能（虽然不是好的形式）将属性名称大写。

我们不打算马上对内置属性进行深入的讨论，这些信息将在第三部分中给出。但是，我们要探讨属性和它们的应用，表 13-1 是对内置影片剪辑属性和基本功能的描述。

表 13-1 内置影片剪辑属性

属性名称	属性描述
<i>_alpha</i>	透明程度
<i>_currentframe</i>	播放头的位置
<i>_droptarget</i>	拖动剪辑所处剪辑或影片的路径
<i>_framesloaded</i>	下载帧的数量
<i>_height</i>	物理高度，单位为像素（是实例的，而不是原来符号的）
<i>_name</i>	剪辑的标识符，作为串返回
<i>_parent</i>	指向包含该剪辑的时间线的对象
<i>_rotation</i>	旋转角度
<i>_target</i>	到该剪辑的完整路径，使用斜线符号
<i>_totalframes</i>	在时间线上帧的总数
<i>_url</i>	.swf 的网络位置
<i>_visible</i>	表示影片剪辑是否被显示的布尔值
<i>_width</i>	物理宽度，以像素为单位（是实例的，而不是原来符号的）
<i>_x</i>	水平像素位置，从场景的左边缘开始计算
<i>_xmouse</i>	鼠标指示器在剪辑坐标空间的水平位置

表 13-1 内置影片剪辑属性（续）

属性名称	属性描述
_xscale	水平尺寸，是和原来符号（或者对影片来说是主时间线）相关的百分数
_y	垂直像素位置，从场景的顶部开始
_ymouse	鼠标指示器在剪辑坐标空间的垂直位置
_yscale	垂直尺寸，是和原来符号（或者对影片来说是主时间线）相关的百分数

没有直接的颜色属性附属于实例或主影片。我们不通过属性来控制颜色，而必须使用*Color*类来创建用于控制剪辑颜色的对象。*Color*对象的方法让我们设置或检查指定剪辑的 RGB 值和变化。要了解相关的细节，请参见第三部分。

## 影片剪辑方法

在第十二章中，我们学习了称为方法的特殊属性类，它是属于对象的函数。方法经常被用来操作和控制它们所属的对象，并进行交互。要以不同的程序方法控制影片剪辑，可以用内置影片剪辑方法，也可以在影片剪辑中单独的实例或者符号库中定义我们自己的影片剪辑方法。

### 创建影片剪辑方法

要将一个新的方法添加到影片剪辑，可在剪辑的时间线上（或者在剪辑的一个事件处理器中）定义一个函数或者将一个函数赋给剪辑的一个属性。例如：

```
// 在剪辑的时间线上定义一个函数以创建方法
function halfSpin() {
    _rotation += 180;
}

// 将函数常量赋给剪辑的一个属性以创建方法
myClip.coords = function() { return [_x, _y]; };
// 这个方法对剪辑进行自定义的转换
myClip.myTransform = function() {
    _rotation += 10;
    _xscale -= 25;
    _yscale -= 25;
    _alpha -= 25;
}
```

## 调用影片剪辑方法

在影片剪辑上调用一个方法与在任何对象上调用一个方法是完全一样的，只需给出剪辑的名称和方法的名称，如下所示：

```
myClip.methodName();
```

如果方法要求参数，我们就在调用的时候传递：

```
_root.square(5); // 将参数 5 传递给 square() 方法
```

正如我们在前边所学到的，在剪辑的时间线上或者剪辑的事件处理器中工作的时候，可以在当前剪辑上直接调用大部分方法，而不指定实例的标识符：

```
square(10); // 调用当前剪辑的自定义方法 square()  
play(); // 调用当前剪辑的内置 play() 方法
```

但是，一些内置方法要求一个实例标识符，参见后面的介绍。

## 内置影片剪辑方法

回忆一下，一般的 *Object* 类用内置方法 *toString()* 和 *valueOf()* 来装配它的所有成员。同样，其他类定义的内置函数可以被它们的成员对象使用：*Date* 对象有 *getHours()* 方法，*Color* 对象有 *setRGB()* 方法，*Array* 对象有 *push()* 和 *pop()* 方法，依此类推。影片剪辑没有什么不同，我们用它们所装配的一系列方法来控制影片剪辑的出现和行为，以检查它们的特征，甚至创建新的影片剪辑。影片剪辑方法是 ActionScript 的中心功能之一。表 13-2 给出了影片剪辑方法的简述，第三部分将作详细讨论。

表 13-2 内置影片剪辑方法

方法名称	方法描述
<i>attachMovie()</i>	创建一个新的实例
<i>duplicateMovieClip()</i>	创建实例的复本
<i>getBounds()</i>	描述被剪辑占据的视觉区域
<i>getBytesLoaded()</i>	返回实例或影片的下载字节数目
<i>getBytesTotal()</i>	返回实例或影片的物理字节大小
<i>getURL()</i>	装载一个外部的文档（通常是 <i>.html</i> 文件）到浏览器

表 13-2 内置影片剪辑方法（续）

方法名称	方法描述
<i>globalToLocal()</i>	将主场景坐标转换为剪辑坐标
<i>gotoAndPlay()</i>	将播放头移动到一个新的帧，并播放影片
<i>gotoAndStop()</i>	将播放头移动到一个新的帧，并停住
<i>hitTest()</i>	表示一个点是否在剪辑内
<i>loadMovie()</i>	将一个外部.swf文件装入播放器
<i>loadVariables()</i>	将一个外部变量装到剪辑或影片中
<i>localToGlobal()</i>	将剪辑坐标转换为主场景坐标
<i>nextFrame()</i>	将播放头往前移动一帧
<i>play()</i>	播放剪辑
<i>prevFrame()</i>	将播放头向后移动一帧
<i>removeMovieClip()</i>	删除一个复制或添加的实例
<i>startDrag()</i>	让实例或影片在场景中物理地跟随鼠标指示器移动
<i>stop()</i>	停止实例或影片的播放
<i>stopDrag()</i>	终止当前正在进行的任何拖动操作
<i>swapDepths()</i>	在实例堆栈中改变实例的层次
<i>unloadMovie()</i>	从一个文档层级或者主剪辑中删除一个实例或影片
<i>valueOf()</i>	一个以绝对方法表示到实例的路径的串，使用点符号

## 方法和全局函数重叠问题

我们在本章中已经提到了多次，一些影片剪辑方法和对应的全局函数同名。你可以在 Flash 制作工具中看到。打开 Actions 面板，确定你在 Expert Mode 中，然后看看动作文件夹。你会看到一长串动作列表，包括 *gotoAndPlay()*, *gotoAndStop()*, *nextFrame()* 和 *unloadMovie()*。这些动作也可以用做影片剪辑方法。这种重复并不单纯是种类的问题，动作是全局函数，和相应的影片剪辑方法完全不同。

因此，当我们执行

```
myClip.gotoAndPlay(5);
```

的时候，我们是在访问名为 *gotoAndPlay()* 方法。但是当我们执行

```
gotoAndPlay(5);
```

的时候，我们又是在访问名为 *gotoAndPlay()* 的全局函数。这两个命令同名，但是它们却不是一回事。*gotoAndPlay()* 全局函数在当前的实例或影片上进行操作。*gotoAndPlay()* 方法在调用它的剪辑对象上操作。大部分时间里，这种微小的区别并不重要。但是，对一些重叠方法/函数对来说，它们之间的区别是比较大的。

一些全局函数要求参数 *target*，用它指定函数要操作的剪辑。在其对应的方法中不需要 *target* 参数，因为方法会自动在调用它们的剪辑上操作。例如，*unloadMovie()* 在它的方法格式中如下所示：

```
myClip.unloadMovie();
```

作为一个方法，*unloadMovie()* 调用的时候不需要参数，它会自动作用于 *myClip*。但是在全局函数形式里，*unloadMovie()* 就会像这样：

```
unloadMovie(target);
```

全局函数要求 *target* 作为它的参数，指定要下载哪个影片。为什么这会产生问题？第一个原因是可能错误地希望通过使用没有任何参数的 *unloadMovie()* 的全局版本来下载当前的文档。就像使用 *gotoAndPlay()* 而不用参数一样：

```
unloadMovie();
```

这种形式不会下载当前的文档。它会引发一个参数量出错的错误。全局函数中的 *target* 参数能引发错误的第二个原因稍微有点复杂，如果你不注意很可能找不到。要对一个要求 *target* 参数的全局函数提供一个 *target* 剪辑，可以使用一个串，它表示到我们要操作的剪辑的路径，或者给出一个剪辑引用。例如：

```
unloadMovie(_level1); // 目标剪辑是一个引用  
unloadMovie("_level1"); // 目标剪辑是一个串
```

我们可以只使用引用，因为对剪辑对象的引用在使用于串语境中的时候会转换为影片剪辑路径。很简单，但是如果 *target* 参数转换成一个空串，或者 *undefined* 值，函数就会在当前时间线上操作！例如：

```
unloadMovie(x); // 如果 x 不存在，x 就产生 undefined,  
// 因此函数就在当前时间线上进行操作  
  
unloadMovie(""); // target 是一个空串，因此函数在当前时间线上操作
```

这会引起一些难以预料的结果。考虑一下，如果我们指向一个不存在的层级会发生什么事：

```
unloadMovie(_level1);
```

如果 \_level1 是空的，解释程序就将引用当做一个没有定义的变量，就会产生 undefined，因此函数是在当前的时间线上操作，而不是在 \_level1 上！那么如何解决这个问题呢？有一些办法可供选择。我们可以在执行函数之前检查目标是否存在：

```
if (_level1) {  
    unloadMovie(_level1);  
}
```

我们也可以一直使用串来表示到达目标的路径。如果串中指定的路径不能到达一个真正的剪辑，函数就会失败：

```
unloadMovie('_level1');
```

在有的情况下，我们可以使用等价的数字函数来进行操作：

```
unloadMovieNum(1);
```

或者总是选择使用方法来避免所有这些问题：

```
_level1.unloadMovie();
```

对于引用来说，下面的函数都容易引起错误（Flash 5全局函数要求有 target 参数）：

*duplicateMovieClip()*  
*loadMovie()*  
*loadVariables()*  
*print()*  
*printAsBitmap()*  
*removeMovieClip()*  
*startDrag()*  
*unloadMovie()*

如果你在影片中遇到了莫名其妙的问题，可以检查列表，看是否误用了一个全局函数。将一个剪辑引用作为 target 参数传递的时候，要检查你的语法。

## 影片剪辑应用举例

我们现在已经学习了影片剪辑编程的基础。现在要用我们已有的知识来创建两个不同的应用程序，两个都是将影片剪辑作为基本内容容器的典型例子。

### 用剪辑制作一个钟

在本章中，我们学习了如何用 `attachMovie()` 来创建影片剪辑，以及如何用点操作符来设置影片剪辑的属性。借助于这些简单工具以及 `Date` 与 `Color` 类的知识，我们具备了用来制作钟的所有条件。我们要制作的钟有时针、分针和秒针。

首先，我们要制作钟面和指针，按照下面的步骤来进行（注意，我们不将钟的各部分放在场景上——我们的钟要完全通过 ActionScript 来生成）：

1. 建立一个新的 Flash 影片。
2. 创建一个名为 `clockFace` 的影片剪辑符号，它包含一个 100 像素宽的黑色圆形。
3. 创建一个名为 `hand` 的影片剪辑符号，它包含一个 50 像素长的水平红线。
4. 选择 `hand` 线，然后选择 Window (窗口) → Panels (面板) → Info (信息)。
5. 将线的 x 坐标设置为 0，y 坐标设置为 -50，从而将线的末端放在剪辑的中央。

现在要导出 `clockFace` 和 `hand` 符号，以便让它们的实例可以动态地添加到影片中来：

1. 在 Library 中，选择 `clockFace` 剪辑，然后选择 Options → Linkage → 就会出现 Symbol Linkage Properties 对话框。
2. 选择 Export 该符号。
3. 在 Identifier box 中，输入 `clockFace`，然后点击 OK。
4. 重复步骤 1 ~ 3 以导出 `hand` 剪辑，将它的标识符设置为 `hand`。

钟面和指针就完成了，准备添加到我们的影片中。现在我们来编写脚本，将钟放在场景中，并确定位置，现在对每个剪辑执行下面的两个步骤：

1. 添加例 13-4 所示的脚本到主时间线中第 1 层的第 1 帧。
2. 将 *Layer 1* 重新命名为 *scripts*。

首先浏览一下例 13-4，然后我们再分析。

#### 例 13-4：一个类似的钟

```
// 创建钟面和指针
attachMovie("clockFace", "clockFace", 0);
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);

// 对钟面进行定位和大小设置
clockFace._x = 275;
clockFace._y = 200;
clockFace._height = 150;
clockFace._width = 150;

// 对指针进行定位、大小设置和上色
secondHand._x = clockFace._x;
secondHand._y = clockFace._y;
secondHand._height = clockFace._height / 2.2;
secondHandColor = new color(secondHand);
secondHandColor.setRGB(0xFFFFFFFF);
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
minuteHand._height = clockFace._height / 2.5;
hourHand._x = clockFace._x;
hourHand._y = clockFace._y;
hourHand._height = clockFace._height / 3.5;

// 在每帧中修改指针的旋转度
function updateClock() {
    var now = new Date();
    var dayPercent = (now.getHours() > 12 ?
                      now.getHours() - 12 : now.getHours()) / 12;
    var hourPercent = now.getMinutes() / 60;
    var minutePercent = now.getSeconds() / 60;
    hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);
    minuteHand._rotation = 360 * hourPercent;
    secondHand._rotation = 360 * minutePercent;
}
```

代码太多，现在我们来复习一下。

我们首先添加了 *clockFace* 剪辑，并将它的深度赋为 0（我们想让它出现在指针的后面）：

```
attachMovie('clockFace', 'clockFace', 0);
```

然后我们添加了三个 hand 实例，将它们命名为 secondHand, minuteHand 和 hourHand。每一个指针都在程序化生成剪辑堆栈主时间线上各自的层中。secondHand (深度 3) 在 minuteHand (深度 2) 上，后者又在 hourHand (深度 1) 上：

```
attachMovie("hand", "secondHand", 3);
attachMovie("hand", "minuteHand", 2);
attachMovie("hand", "hourHand", 1);
```

现在，代码会将钟放在场景的左上角。下面，我们将 clockFace 剪辑移动到场景中央，并用 \_height 和 \_width 属性将它放大：

```
clockFace._x = 275;           // 设置水平位置
clockFace._y = 200;           // 设置垂直位置
clockFace._height = 150;       // 设置高度
clockFace._width = 150;        // 设置宽度
```

然后我们将 secondHand 剪辑移动到钟上，让它和 clockFace 剪辑的半径一样长：

```
// 将 secondHand 放在 clockFace 的上面
secondHand._x = clockFace._x;
secondHand._y = clockFace._y;
// 设置 secondHand 的大小
secondHand._height = clockFace._height / 2.2;
```

hand 符号上的线是红色的，因此我们所有的 hand 实例迄今都是红色的。要使用 secondHand 剪辑，就要用 *Color* 类将它变成白色。注意十六进制颜色值 0xFFFFFFF 的使用（参见第三部分可以获得颜色操作方面的更多信息）：

```
// 创建一个新的 Color 对象来控制 secondHand
secondHandColor = new Color(secondHand);
// 将 secondHand 设为白色
secondHandColor.setRGB(0xFFFFFF);
```

下面，我们设置 minuteHand 和 hourHand 的位置和大小，正如我们对 secondHand 所做的那样：

```
// 将 minuteHand 放在 clockFace 上面
minuteHand._x = clockFace._x;
minuteHand._y = clockFace._y;
// 将 minuteHand 调整得比 secondHand 稍微短一些
minuteHand._height = clockFace._height / 2.5;
```

```
// 将hourHand放在clockFace上面  
hourHand._x = clockFace._x;  
hourHand._y = clockFace._y;  
// 让hourHand比其他指针都短  
hourHand._height = clockFace._height / 3.5;
```

现在我们要按照当前的时间来设置指针的旋转度。但是不只设置一次，我们要重复地进行设置，以便时钟随着时间的推移而动作。因此，我们将旋转代码放在函数 *updateClock()* 中，以便重复调用：

```
function updateClock() {  
    // 存储当前的时间  
    var now = new Date();  
    // getHours() 对 24 小时的时钟才有效。如果当前的小时数大于 12，  
    // 就减去 12，转换为正规的 12 小时时钟  
    var dayPercent = (now.getHours() > 12 ?  
        now.getHours() - 12 : now.getHours()) / 12;  
    // 确定当前的小时内，时间已经过了多少分钟，用百分比表示  
    var hourPercent = now.getMinutes() / 60;  
    // 确定当前的分钟内，时间已经过了多少秒，用百分比表示  
    var minutePercent = now.getSeconds() / 60;  
    // 将指针围绕时钟旋转相应的角度  
    hourHand._rotation = 360 * dayPercent + hourPercent * (360 / 24);  
    minuteHand._rotation = 360 * hourPercent;  
    secondHand._rotation = 360 * minutePercent;  
}
```

*updateClock()* 的第一个任务就是获取和存储当前的时间。我们创建一个 *Date* 类的实例，并将它放在局部变量 *now* 中。下面要确定每个指针应该围绕时钟旋转角度的百分比——很像确定从哪里下手切蛋糕。当前的小时总是 12 的一部分，而当前的分钟和秒钟总是 60 的一部分。我们将每个指针的 *\_rotation* 基于这些百分比来进行设置。对于 *hourHand*，我们所要反映的不仅是当前日的百分比，还有当前小时的百分比。

时钟基本上完成了，剩余的事情就是在每帧中调用 *updateClock()* 函数。如下所示：

1. 添加两个关键帧到 *scripts* 层。
2. 在第 2 帧中，添加下面的代码：*updateClock();*
3. 在第 3 帧中，添加下面的代码：*gotoAndPlay(2);*

测试影片，查看时钟是否运行。如果它不动，就将它和在线代码库中给出的 *.fla* 时钟示范文件相比较，或者检查例 13-4 中的代码。可以想想怎样来扩展时钟应用程序：

可以将主时间线循环（第 2 帧和第 3 帧之间）转换为剪辑事件循环吗？可以将时钟做得更精美一些吗？在 clockFace 上动态添加分钟和小时标记会如何？

## 最后的测试

这是我们在第一章中开始讨论的多项选择测试的最后一个版本。这个测试的更新版本将用影片剪辑动态生成所有的测试问题和答案，因此，我们的测试可以无限升级，并高度结构化。实际上，我们距离非程序员所能做的最迷人的剪辑已经不远了。

测试的代码如例 13-5 所示，也可以在在线代码库中得到。因为现在测试全部是动态生成的，99% 的代码都出现在一个帧中，我们不再需要用问题来填充时间线了。（我们遗漏的只是一个预装载器，它可以确保影片在网络上的平滑播放。）注意，我们已经使用了 #include 来从外部的文本文件中导入代码块。关于 #include 的更多信息，参见第三部分和第十六章。作为一个练习，你可以尝试创建新的对象并将它们放在题目数组中，以添加新的问题。

虽然最后测试的代码相对来说比较短，但是它囊括了很多重要的技术。除了 #include 之外，我们已经在前面单独学过所有的内容，但这个扩展的现实示例显示的是它们如何共同工作的。认真看看注释——当你全部理解了这个测试版本的时候，就可以用 ActionScript 来创建高级的应用程序了。

关于下面代码的更多的说明可以在下面的网址访问到：

<http://www.moock.org/webdesign/lectures/ff2001sfWorkshop>

### 例 13-5：多项选择测试的最后版本

```
// 主时间线第1帧中的代码
// 停止影片
stop();

// 初始化主时间线变量
var displayTotal; // 显示用户最终成绩的文本域
var totalCorrect = 0; // 回答正确的题目数
var userAnswers = new Array(); // 包含用户答案的数组
var currentQuestion = 0; // 用户所在的题目号码

// 导入包含题目对象数组的源文件
#include "questionsArray.as" // 参见本例中后面的说明
```

```
// 开始测试
makeQuestion(currentQuestion);

// Question()构造器
function Question (correctAnswer, questionText, answers) {
    this.correctAnswer = correctAnswer;
    this.questionText = questionText;
    this.answers = answers;
}

// 用来将每个题目显示在屏幕上的函数
function makeQuestion (currentQuestion) {
    // 清屏
    questionClip.removeMovieClip();

    // 创建和放置上题目剪辑
    attachMovie("questionTemplate", "questionClip", 0);
    questionClip._x = 277;
    questionClip._y = 205;
    questionClip.qNum = "question\n" + (currentQuestion+1);
    questionClip.qText = questionsArray[currentQuestion].questionText;

    // 在题目剪辑里分别创建答案剪辑
    for(var i = 0; i < questionsArray[currentQuestion].answers.length; i++) {
        // 从库中添加链接的answerTemplate 剪辑
        // 它包含一个一般按钮和一个题目文本域
        questionClip.attachMovie("answerTemplate", "answer" + i, i);
        // 将答案剪辑放在题目下面
        questionClip["answer" + i]._y -= 70 + (i * 15);
        questionClip["answer" + i]._x -= 100;
        // 在答案剪辑中设置文本域以容纳这个题目相应的答案数组元素
        questionClip["answer" + i].answerText =
            questionsArray[currentQuestion].answers[i];
    }
}

// 用来记录用户答案的函数
function answer(choice) {
    userAnswers.push(choice);
    if(currentQuestion+1 == questionsArray.length) {
        questionClip.removeMovieClip();
        gotoAndStop("quizEnd");
    } else {
        makeQuestion(++currentQuestion);
    }
}

// 统计用户成绩的函数
function gradeUser() {
    // 计算用户回答对了多少题目
    for(var i = 0; i < questionsArray.length; i++) {
        if(userAnswers[i] == questionsArray[i].correctAnswer) {
```

```

        totalCorrect++;
    }
}

// 在屏幕文本域中显示用户成绩
displayTotal = totalCorrect + " / " + questionsArray.length;
}

// 程序化生成答案按钮上的代码
// 答案剪辑可以动态生成，按照 "answer0", "answer1", ..., "answern" 的序列来命名
// 每一个答案剪辑包含一个按钮，在点击的时候检查其中的答案剪辑名称，以确定用户的选择
on (release) {
    // 除去剪辑名称中的 "answer" 前缀
    choice = _name.slice(6, _name.length);
    _root.answer(choice);
}

// quizEnd帧上的代码
gradeUser();

```

*questionsArray.as* 文件中的内容如下：

```

// questionsarray.as文件的代码
// -----
// 包含一个题目对象的数组，它组装测试中的问题和答案
// 按照下面的例子来构成新的题目对象

***** 问题对象示例 *****
// 调用 Question 构造器，带三个参数
// 给正确答案一个和 0 相关的号码
// 给问题文本一个串
// 一个包含多项选择答案的数组
new Question
(
    1,
    "question goes here?",
    ["answer 1", "answer 2", "answer 3"]
)
***** */
// 记着在数组中的对象之间放一个逗号
questionsArray = [new Question (2,
    "Which version of Flash first introduced movie clips?",
    ["version 1", "version 2", "version 3",
    "version 4", "version 5", "version 6"]),
new Question (2,
    "When was ActionScript formally declared a scripting language?",
    ["version 3", "version 4", "version 5"]),
new Question (1,
    "Are regular expressions supported by Flash 5 ActionScript?",
```

```
i 'yes', "no"]),
new Question (0,
  'Which sound format offers the best compression?',
  ['mp3', "aiff", "wav"]),
new Question (1,
  'True or False: The post-increment operator(++) returns the value
  of its operand+1.',
  ["true", "false"]),
new Question (3,
  'ActionScript is based on...',
  ["Java", "JavaScript", 'C++', 'ECMA-262', "Perl"]);
```

## 小结

我们已经学了许多 ActionScript 的知识，一旦深入理解了对象和影片剪辑，你就能自己完成大部分 ActionScript 课题了。但是我们还有很多东西要学习：下两章将讨论“词法结构”（ActionScript 语法的精确细节）和各种高级主题。最后，我们将学习第二和第三部分。

---

# 第十四章

# 词法结构

语言的词法结构是一系列规则，它控制语法成分。我们在编写脚本源代码的时候必须遵循这些规则。

## 空白

Tab, 空格, 以及回车（也就是分行）字符在 ActionScript 中的用法就像它们在英语中用来分隔单词一样，让单词不至于粘连在一起。从程序员的角度来说，这些字符就是空白，在源代码中使用以便分隔符号（和英语中的单词、短语与句子相似的关键字，标识符和表达式）。下面的例子给出了空白正确和错误的使用方法：

```
varx // 在关键字 var 和变量 x 之间没有空白  
var x // 这就好了……因为使用了空白，解释程序现在就能读懂代码了
```

如果有其他分隔符能告诉 ActionScript 在何处中断，从何处开始，空白就是可有可无的。下面的代码是合法的，因为操作符 =, + 和 / 将 x, 10, 5 和 y 互相分隔开了：

```
x-10+5/y; // 有点拥挤，但是是合法的  
x = 10 + 5 / y; // 很容易读懂，但是和上面是一样的
```

类似的，空白在有其他符号（比如方括号、括号、花括号、逗号和大于或小于号）用来作分隔的时候也是可有可无的。下面这些代码都是合法的，虽然可能会让人感到不舒服：

```
for(var i=0;i<10;i++){trace(i);}
if(x==7){y=[1,2,3,4,5,6,7,8,9,10];}
myMeth=function(arg1,arg2,arg3){trace(arg1+arg2+arg3);}
```

较多的空白只不过是一种风格形式而已，因为它会被ActionScript解释程序所忽略。也就是说，你可以遵循一些惯例而让代码显得更为易读。例如，下面将用另外一个合法的方法来重写前面的赋值表达式，但是显然不容易看懂：

```
x =
  10
  + 5
  / y;
```

注意，语句是由分号来结束的，而不是分行符。在几乎所有的情况下，分行符都是没有意义的，并不作为语句的终止符号。因此，经常使用一个或者多个分行符（同空格或tab键相一致）可让复杂的语句更容易被读懂：

```
myNestedArray = [[x,y,z],
                  [1,2,3],
                  ["joshua davis", "yugo nakamura", "james patterson"]];
// 比下面的形式要整洁得多
myNestedArray = [[x, y, z], [1, 2, 3], ["joshua davis", "yugo nakamura",
"james patterson"]];
// 也比这种形式整洁
myNestedArray = [[x, y, z],
                  [1, 2, 3],
                  ["joshua davis", "yugo nakamura", "james patterson"]];
```

如果想把一个语句分成多行，并不需要做特殊的工作，只要添加一个回车，然后继续输入就可以了。要想得到关于如何提高代码易读性的建议，可参见 Steve McConnell 著的《Code Complete》(Microsoft Press)。但是，有的时候分行符也会被误解为语句的分隔符号，我们会在下一节看到这种情况。

注意，用来划分串界限的引号内的空白和引号外边被忽略的空白是不同的。比较下面的两个例子：

```
x = 5;
trace('The value of x is' + x);      // 显示: "The value of x is5"
trace("The value of x is "+x);       // 显示: "The value of x is 5"
```

## 语句终结符（分号）

正如我们在第六章中学习到的，分号用来终止一个ActionScript语句。虽然为了方便起见，你总会用分号来结束一个语句，但是，它们在ActionScript中并没有严格的要求。解释程序会在没有使用分号的时候推断语句的结尾。例如：

```
// 这些都是很恰当的
var x = 4;
var y = 5;
// 但是这些也是合法的
var x = 4
var y = 5
```

ActionScript解释程序假定前面的代码是用分行符号来作为语句终结符的。（在要求严格的语言中（比如C），编译器就会报错。）但是，从代码语句中省略分号有点像普通的书写中省略了句号一样——读者可能会理解你大部分的句子，但是总会有导致误解的情况，更不要说读起来有多费力了。例如，考虑我们在return语句后面省略了分号的时候会发生什么情况：

```
function addOne (value) {
    return
    value + 1
}
```

ActionScript会假定我们要表达的意思是：

```
function addOne (value) {
    return;
    value + 1;
}
```

函数就不会返回value+1，而通常会返回undefined，因为关键字return单独存在也是一个合法语句。甚至如果return单独出现在一行中，即使你在value+1后面添加了一个分号，return语句仍然会被当作一个完整的语句来使用。

要避免这种含混的情况，最好使用分号。此外，在return语句的特定情况下，不要将关键字return同它的表达式用分行符分开，这会改变语句的意思。因此，上面的语句应该这样书写：

```
function addOne(value) {
    return value + 1;
}
```

注意，分号用来终结单独的语句，但是，在语句块结束的时候却不需要。例如：

```
for (var i=0;i<10;i++) {           // 这里没有分号
    trace(i);
}

if(x == 10) {                      // 这里没有分号
    trace('x is ten');
} else {
    trace('x is not ten');         // 这里有分号
}                                    // 这里没有分号

on (release) {                     // 这里没有分号
    trace('Click');
}
```

但是，在函数常量后面必须有分号：

```
function (param1, param2, ...paramn) { statements };
```

如果使用了特殊的 #include 指示，不带分号就会引起错误。参见第三部分。

## 注释

所谓的注释就是被解释程序忽略、程序员故意输入以便增加代码易读性的文本。注释经常用来解释代码所完成的工作，提供版本控制信息，或者描述其他有关信息（比如数据结构如何使用，为什么要做一定的程序选择）。注释可以在概念的层次上描述代码，而不只是反映代码本身的语法。例如，下面的注释是没有用的：

```
// 将 i 设置为 5
i = 5;
```

但是，下面的注释告诉了我们为什么要将 i 设置为 5，这可以帮助我们跟随代码的流程：

```
// 初始化计数器，用来搜索从索引 5 开始的密码串
var i = 5;
```

注意，可以使用描述性的变量名和处理器名为代码作“自我注释”。如下所示：

```
x = y / z;                      // 这种代码很模糊
average = sum / numberOfItems;    // 这几乎不需要解释了
```

ActionScript 支持单行和多行的注释。单行注释我们在本书中已经看过，是以两个斜杠来开始的（//）：

```
// 啊，人类的交流……这永远不会被解释程序读到
```

单行的注释由行末的分行符号自动结束，必须重复使用//符号将更多的注释文本添加到以后的行中：

```
// 这里是一个注释的开始……  
// ……这里还有一些注释内容
```

单行注释也可以跟在真正的代码后面，在同一行中：

```
var x; // 这是一个合法的注释，它和代码在同一行中
```

ActionScript 也支持多行注释，用来适应大块代码的说明，它不被分行符所打断。多行注释以两个符号构成的序列 /\* 开始，一直持续到用 \*/ 来结束，比如：

```
/* ----- 开始版本控制信息 -----  
名称:MyApplication, 版本: 1.3.1  
作者: Killa Programma  
最后修改: 2000 年 8 月 7 日  
----- 结束版本控制信息 -----  
*/
```

多行注释只能在 Flash 5 ActionScript 编辑器的专家模式里，或者通过 #include 指示包括进来的外部.as源文件中可以使用。多行注释在你要从专家模式转换为普通模式的时候会转换为单行注释。注意，嵌套的注释是合法的：

```
/* 这是  
* 一个嵌套的 // 注释 */
```

下面的代码使用起来并不方便，但它也是合法的：

```
/* 这个注释后面跟着真正的代码 */ var x = 5;
```

在对代码进行大修改（或者小修改）的时候将最初的和改变的日期放到注释中是很常见的，比如：

```
// 在财政计算中修正错误 BAE 12-04-00. V1.01
```

注释可以临时让代码失效（或者永久地让旧代码失效），而不是删除它。将代码变为

注释就可以让代码失效，这被称为批出代码。只要删除了注释分隔符号，就可以很容易地恢复代码功能：

```
/* 让代码块失效，直到我改变主意
duplicateMovieClip("character", "newCharacter", 1);
newCharacter._rotation = 90;
*/
```

可以简单地在单行代码前面加两个斜杠让其失效：

```
// newCharacter._rotation = 90;
```

## 保留字

ActionScript 解释程序用一些保留字来表示特殊的内置语言功能，比如语句和操作符。由于它们是留给解释程序所使用的，我们必须避免将它们用做代码中的标识符。不是为了实现保留的内部意图而使用了保留字，在大多数情况下会引起错误。ActionScript 的保留字如表 14-1 所示。

表 14-1 ActionScript 的保留字

add <sup>a</sup>	for	lt <sup>a</sup>	tellTarget <sup>a</sup>
and <sup>a</sup>	function	ne <sup>a</sup>	this
break	ge <sup>a</sup>	new	typeof
continue	gt <sup>a</sup>	not <sup>a</sup>	var
delete	if	on	void
do	ifFrameLoaded <sup>a</sup>	onClipEvent	while
else	in	or <sup>a</sup>	with
eq <sup>a</sup>	le <sup>a</sup>	return	

a. 在 Flash 5 中不支持 Flash 4 的保留字。

你也应该避免使用表 14-2 列出的保留字。它们不是 Flash 5 ActionScript 中的一部分，但是以后却会成为语言的一部分，因为 ECMA-262 可能要使用它们。

表 14-2 潜在的未来保留字

abstract	extends	private
boolean	final	protected
byte	finally	public
case	float	short
catch	goto	static
char	implements	super
class	import	switch
const	instanceof	synchronized
debugger	int	throws
default	interface	transient
double	long	try
enum	native	volatile
export	package	

除了一般定义的保留字之外，还应该避免使用内置属性、方法和对象的名称来作为代码中的标识符。这样做很可能会覆盖属性、方法或对象的默认行为。例如：

```
Date = new Object(); // 哦！我们让Date()构造器失效了
```

现在我们不能再创建 *Date* 对象：

```
var now = new Date(); // 将now设置为undefined
trace(now); // 显示空串，而不是当前的时间和日期
```

## 标识符

所有的变量、函数和对象属性都用标识符来命名。ActionScript 标识符必须按照下面的规则来构成：

- 标识符只能包含字母（A-Z 或 a-z）、数字、下划线和美元符号。要特别注意不要在标识符中使用空格、句点、反斜杠或者其他标点符号。
- 标识符必须以字母、下划线或者美元符号开始（不能是数字）。
- 标识符不能和保留字相同。

虽然没有严格的要求，但是最好在构成影片剪辑实例名称、帧标签和层名的时候遵循上面的规则。

## 大小写区分

当一种语言全部要区分大小写的时候，语言中的所有符号（包括所有的标识符和关键字）都必须以正确的大小写形式书写。例如，在区分大小写的语言中，语句：

```
if (x == 5) {  
    x = 10;  
}
```

会引起错误，因为关键字 if 被写成了 If。此外，在区分大小写的语言中，下面的两个语句会声明两个不同的变量（一个名为 firstName，另外一个为 firstname）：

```
var firstName = "doug";  
var firstname = "lerry";
```

ActionScript 所基于的 ECMA-262 规范要求全部区分大小写。但是 ActionScript 在这方面没有和标准保持一致，这是为了保持和 Flash 4 影片的向后兼容。在 ActionScript 中，表 14-1 中的关键字都是区分大小写的，但是标识符却不然。例如，在下面的代码中，onClipEvent 关键字写做 onclipevent 就是不正确的，会引起错误：

```
onclipevent (enterFrame) // 应该是 onClipEvent (enterFrame)
```

但是和关键字不同，标识符在 ActionScript 中不区分大小写，因此下面的语句会将一个值赋给同一个变量：

```
var firstName = 'margaret';  
var firstname = "richael";  
trace(firstName); // 产生 "richael"  
trace(firstname); // 也产生 'richael' (变量是同一个)
```

在 ActionScript 中，甚至属性名称和函数名称这样的内部标识符也不区分大小写。下面的代码在 JavaScript 中会发生错误，但是在 ActionScript 中却可以成功地创建一个数组：

```
myList = new array(); // 应该是 new Array();
```

将 JavaScript 代码放到 ActionScript 中或者 JavaScript 程序员学习 ActionScript 的时候这可能会是一个问题。在 JavaScript 中，经常用区分大小写的方式来为不同的目的提供相同的名称。例如，语句：

```
date = new Date(); // 在 JavaScript 中是有效的，但是在 ActionScript 中则不然
```

会在 ActionScript 造成破坏性的结果，因为标识符 date 不能同对象类 Date 区分开。在 ActionScript 中，前面的代码会让内置 Date 类失效。我们因此必须确保标识符不仅在大小写方式上，而且在任何预定义的标识符（比如 Date 或 Array）之间有所区别。下面，我们在 ActionScript 中重写前面的代码：

```
myDate = new Date(); // 在 ActionScript 中使用这种方式
```

所有情况的关键是必须一致——甚至当语言中不严格要求一致性的时候。保持变量、函数、实例和其他项目的大小写一致可以让你的代码更为易读，也防止 ActionScript 以后在大小写规则上的改变会影响你艰苦的劳动成果。

## 小结

我们已经学完了 ActionScript 语言基础，你现在已经完全具备了 ActionScript 的读写能力。下一章将介绍一些更为高级和深奥的主题，它不会给我们以前的讨论造成混乱。

# 第十五章

## 高级主题

本章介绍了各种 ActionScript 高级编程技术和一些问题。

### 复制、比较和传递数据

操作数据有三种方法。复制（如将变量 *x* 的值赋给变量 *y*）、比较（如检查 *x* 是否等于 *y*）和传递（如将一个变量作为参数提供给函数）。原始数据值被复制、比较和传递的方式和复合数据有很大不同。当原始数据被复制给一个变量的时候，这个变量就得到了它自己独特的、私有的数据拷贝，独立地存储在存储器中。因此下面的代码会让串 “Dave” 在存储器中存储两次，一次是为 *name1* 占据存储空间，另一次是为 *name2* 占据空间：

```
name1 = "Dave";
name2 = name1;
```

原始数据是通过值进行复制的，因为数据直接量的值存储在了分配给变量的存储空间里。相反，当复合数据被复制给一个变量的时候，只有数据的引用（不是真正的数据）被存储在分配给变量的存储空间中。这个引用告诉解释程序真正的数据放在哪里（也就是它在存储器中的访问地址）。当包含复合数据的变量被复制给其他变量的时候，同样是引用（通常称为指针）而不是数据本身被复制。因此，复合数据被称为通过引用进行的复制。

这会产生良好的设计感觉，因为复制庞大的数组和其他复合数据类型效率非常低，但它会对代码产生一个很重要的结果。当同一个复合数据被赋给多个变量的时候，每一个变量并不存储该数据的独特拷贝（和数据是原始数据的时候会发生的情况不同）。实际上只存在一份数据拷贝，所有的变量都指向它。如果数据的值改变了，所有的变量值都会跟着改变。

我们来看看这对实际的应用程序会产生什么影响。当两个变量指向同一个原始数据的时候，每一个变量都得到自己的数据拷贝。下面，我们将值 12 赋给变量 x：

```
var x = 12;
```

现在我们将 x 的值赋给新的变量 y：

```
var y = x;
```

你也许可以猜到，y 现在等于 12。但是 y 有它自己的值 12 的拷贝，和 x 的拷贝截然分开。如果我们改变 x 的值，那么 y 的值并不会受到影响：

```
x = 15;  
trace(x); // 在输出窗口中显示 15  
trace(y); // 在输出窗口中显示 12
```

y 的值在 x 改变的时候不发生变化，这是因为当我们把 x 赋给 y 的时候，y 接收到了自己的数字 12 的拷贝（也就是 x 所包含的原始数据）。

现在我们来看看相同的事情发生在复合数据上会如何。我们创建一个新的数组，它有三个元素，然后我们将数组赋给变量 x：

```
var x = ["first element", 234, 18.5];
```

现在，正如我们在前面所做的那样，我们将 x 的值赋给 y：

```
var y = x;
```

y 的值现在和 x 的值相等。但是 x 的值是什么呢？还记得我们说过，因为 x 指向一个数组，而数组是复合数据，因此 x 的值不是数组直接量 ["first element", 234, 18.5]，而只是一个指向该数据的引用。因此，当我们把 x 赋给 y 的时候，拷贝给 y 的就不是数组本身，而是包含在 x 中的指向数组的引用。因此，x 和 y 都指向存储在存储器某处的同一个数组。

如果我们通过变量 x 改变数组，比如：

```
x[0] = "list element";
```

这个变化也会影响到 y：

```
trace(y[0]); // 显示: list element
```

类似的，如果我们通过 y 修改数组，x 也会受到影响：

```
y[0] = "second element";
trace(x[0]); // 显示: second element
```

要打断这种联系，可以使用 *slice()* 函数来创建全新的数组：

```
var x = ["first element", 231, 18.1];
// 将 x 的每一个元素复制给 y 中存储的数组
var y = x.slice(0);
y[0] = "hi there";

trace(x[0]);           // 显示: 'first element' (不是 'hi there')
trace(y[0]);           // 显示: "hi there" (不是 "first element")
```

我们将这个例子扩展一下，看看原始数据和复合数据值是如何进行比较的。下面，我们将赋给 x 和 y 同样的原始数据值，后比较这两个变量：

```
x = 10;
y = 10;
trace(x == y);        // 显示: true
```

因为 x 和 y 包含的是原始数据，它们之间就通过值进行比较。在一个以值为基础的比较中，数据是以直接量值进行比较的。x 中的数字 10 被认为和 y 中的数字 10 相等，因为它们的数字都是由相同的数位构成。

现在，赋给 x 和 y 同一个复合数据的相同版本，并且再次将两个变量进行比较：

```
x = [10, "hi", 5];
y = [10, "hi", 5];
trace(x == y);        // 显示: false
```

这次，x 和 y 包含的是复合数据，因此它们之间进行的是引用的比较。我们赋给 x 和 y 的数组有相同的字节值，但是变量 x 和 y 却不等，这是因为它们存储的并不是指向同一个复合数据的引用。但是，当我们将 x 中的引用复制给 y 的时候，来看看会如何：

```
x = y;
trace(x == y); // 显示: true
```

既然引用是相同的，那么值也就被认为是相等的。因此，比较的结果根据的是变量中的引用，而不是数组中的实际值。

原始数据和复合数据在传递给函数的时候也不一样，这在第九章中讨论过。应当特别指出，当一个原始变量作为参数传递给一个函数的时候，函数内进行的对数据的任何改变都不会影响原来的变量。但是，当传递一个复合变量的时候，在函数内所做的改变就会影响到原来的变量。也就是说，如果将一个整数变量 `x` 传递给一个函数，在函数内修改它并不会影响其原来的值。但是如果将数组 `y` 传递给函数，在函数内对数组所做的任何改变都会影响函数外边 `y` 的原有值（这是因为对数组的改变影响到了 `y` 所指向的数据）。

## 位逻辑编程

现在我们来换一换脑子，讨论一个不相关的、深奥的主题——使用位逻辑操作符的位逻辑编程。这个问题在第五章中说得并不清楚，但是，我们现在会和那些富有经验的程序员和一些勇敢的新手一起再讨论一下这方面的内容。

为了以高度优化的方法跟踪和处理一系列的选项，我们可以使用位逻辑操作符。从理论上说，位逻辑操作符是数学操作符，但是它们的典型使用是在逻辑环境中，而不是数学环境中。位逻辑操作符可以访问整数中单个的二进制数（位）。要理解这是怎么回事，需要知道数字是如何被表示为二进制数的。

一个二进制数作为 0 和 1 构成的序列来存储，它表示基数为 2 的数字体系（也就是二进制体系）。数字中的每一个数位表示数字系统的基数的某次幂。二进制数使用数字 2 作为基数，因此一个二进制数的头四位从右到左表示，1 的位 ( $2^0$ )，2 的位 ( $2^1$ )，4 的位 ( $2^2$ )，以及 8 的位 ( $2^3$ )。下面给出的是一些二进制数的例子，并说明怎样将它们的数位值换算成等价的十进制数：

```
1      // 十进制数 1:(1 x 1) 等于 1
10     // 十进制数 2:(1 x 2) + (0 x 1) 等于 2
11     // 十进制数 3:(1 x 2) + (1 x 1) 等于 3
100    // 十进制数 4:(1 x 4) + (0 x 2) + (0 x 1) 等于 4
1000   // 十进制数 8:(1 x 8) + (0 x 4) + (0 x 2) + (0 x 1) 等于 8
1001   // 十进制数 9:(1 x 8) + (0 x 4) + (0 x 2) + (1 x 1) 等于 9
```

在二进制中，我们讨论的一个数位就是比特（“二进制”位的简称）。例如，一个四位数，就是一个有四个数位的数字（每个数位包含0或者1）。最右边的比特被认为是第0位，它左边的是第1位，依此类推。下面是一个8位的数字，它在第0、第6和第7比特位上为1，我们已经把比特位标在数字上面了：

位： 76543210  
11000001

和所有的数字系统一样，单个数位的最大值就是基数（也被称为根）减去1。例如，在十进制中，最大的单个数字是9。注意，因为我们使用2作为基数，因此，每一个二进制数位必须是0或者1。我们知道，数字0和1可以等同于布尔值 `false` 和 `true`，因此，使用二进制数字作为一系列的开关转换是非常方便的！这正是位逻辑操作符让我们做的。

一个值为1的比特可以说被设置了（也就是开，或者 `true`）。一个值为0的比特可以说是被清除了（也就是关，或者 `false`）。比特有的时候被当作标记或者开关，这意味着它表示某种可以有两种状态的东西（比如开/关或者 `true/false`）。

位逻辑编程几乎总是包含一系列特征可以被设置/清除的情况。使用位逻辑操作符，可以在单个的数据值内简明地表示多个选项，而不需使用多个变量。这可以提供更高的执行效率和更低的存储器要求。

假设我们在建立一个销售汽车的 Flash 站点。为了简洁，我们说只卖一种汽车，但是用户可以用四种选项的任意组合来自定义他们所需要的汽车：空调，CD 播放器，遮阳篷顶和皮座位。为包括所有选项的汽车计算总价是 Flash 程序的任务，它还要负责一个服务器端的程序，以跟踪用户介绍的部分信息。

我们可以将汽车的选项存储在四个单独的布尔变量中，如下所示：

```
var hasAirCon = true;  
var hasCD = true;  
var hasSunRoof = true;  
var hasLeather = true;
```

本质上，我们得到了四个开关——每一个开关都包含汽车选项内容中的每一项——每一个开关都要求一个变量。这很有效，但是它意味着我们在存储器中需要四个变量，在服务器中需要四个地方来放置用户信息数据库。当我们把汽车选项记录

为单个的二进制数时，就可以将所有四个选项存储在一个单一的4位二进制数中：空调为第0位（1的位），CD播放器为第1位（2的位），遮阳篷顶为第2位（4的位），皮座位为第3位（8的位）。下面给出了简单的构造例子，显示一个单一的数字是如何表现四种选项的任意组合的：

```
var options;
options = 1 // 1是0001; 0位打开: 空调
options = 2 // 2是0010; 1位打开: CD播放器
options = 4 // 4是0100; 2位打开: 遮阳篷顶
options = 8 // 8是1000; 3位打开: 皮座位

// 下面是最酷的部分: 组合选项
options = 5 // 5是0101: 空调(1) 和遮阳篷顶(4)
options = 10 // 10是1010: CD播放器(2) 和皮革(8)
options = 15 // 15是1111: 全部都选了!
```

不管我们什么时候要添加或者删除选项，都只需要加上或者减去对应的比特值：

```
var options = 0; // 开始的时候没有选项
options += 4; // 加上遮阳篷顶 (options 为 4, 或者 0100)
options += 1; // 加上空调 (options 为 5, 或者 0101)
options += 2; // 添加CD播放器 (options 为 7, 或者 0111)
options -= 4; // 删除遮阳篷顶 (options 为 3, 或者 0011)
options += 8; // 添加皮革座位 (options 为 11, 或者 1011)
```

这样，我们就知道了如何将多个选项存储为单个数字中一系列的比特位。我们如何检查这些比特来计算汽车的价钱呢？这需要使用位逻辑操作符。我们首先看看操作符，然后再来讨论这个例子。

## 位逻辑 AND

位逻辑 AND 操作符（&）在数字的每一个比特位上执行逻辑 AND 操作，将两个数字的比特位合并起来。操作将合并的结果作为一个数字返回。

位逻辑 AND 表达式的格式为：

*operand1 & operand2*

位逻辑 AND 的操作数可以是任何数字，但是它们会在进行操作之前被转换为 32 位的二进制整数。如果一个操作数有小数值，比如 2.5，那么小数部分就会被抛弃。

注意，位逻辑 AND 使用单字符操作符 `&`，而且操作要在其操作数的单个比特位中进行，但是第五章所讨论的逻辑 AND 操作符使用两个字符的操作符 `&&`，将每个操作数整体使用。

位逻辑 AND 返回的数字值是通过比较数字操作数 `operand1` 和 `operand2` 中的单个比特位来决定的，一次比较一个。如果一个比特位在两个操作数中都包含数字 1，那么结果中对应的比特位也应该设置为 1；否则，结果中的比特位就会被设置为 0。

位逻辑 AND 操作其实很容易描述，我们可以将数字操作数的二进制等价数字水平放置，然后将它们的比特位上下对齐。这样，很容易判断哪些比特位上的两个操作数都包含 1。

在本例中，第 2 位（从右边开始的第三位）的两个操作数都是 1，因此结果要设置为 1。结果中的其他位被设置为 0：

$$\begin{array}{r} 1111 \\ \& 0100 \\ \hline 0100 \end{array}$$

在这个例子中，第 0 位和第 3 位的操作数都是 1，因此结果中该位为 1。第 1 位和第 2 位要设置为 0：

$$\begin{array}{r} 1101 \\ \& 1011 \\ \hline 1001 \end{array}$$

ActionScript 使用十进制数字来代替二进制数字，这样，要形象化位逻辑运算就比较困难。下面是前面的操作在真实代码中的样子：

```
15 & 4      // 结果为 4  
13 & 11     // 结果为 9
```

在实际中，位逻辑 AND 操作符用来检查一个特殊的标志或者标志设置（也就是比特）是否为 `true` 或者 `false`。

下面的例子要检查第 2 位（值为 4）是否被设置为 `true`：

```
if (x & 4) {
```

```
// 完成某种工作  
}
```

或者，我们可以检查第 2 位或者第 3 位（值为 8）是否有一位被设置为 true：

```
if (x & (4|8)) {  
    // 完成某种工作  
}
```

注意，前面的例子用后面就要讨论的 | 操作符来检查第 2 位或第 3 位是否被设置。要检查第 2 位和第 3 位是否都被设置，可以使用：

```
if (x & (4|8) == (4|8)) {  
    // 完成某种工作  
}
```

位逻辑 AND 操作符也可用来将数字中单个的比特设置为 false，参见本章后面位逻辑 NOT 操作符部分。

## 位逻辑 OR

位逻辑 OR 操作符 (|) 在数字的每一位上执行一个逻辑 OR 运算，将两个数字的比特位合并起来。和位逻辑 AND 一样，位逻辑 OR 返回合并后的数字结果。位逻辑 OR 表达式的格式为：

*operand1* | *operand2*

操作数可以是任何数字，但是会在操作之前转换为 32 位的二进制整数。如果操作数有小数部分，会将其丢弃。

注意，位逻辑 OR 使用的是单字符操作符 |，并且在数字内单个的比特上进行操作，但是第五章中讨论的逻辑 OR 操作符使用双字符操作符 ||，将每个操作数作为整体来对待。

结果中每一个比特位是通过对两个操作数中的比特位进行逻辑 OR 计算来确定的。因此，如果一个比特位在 *operand1* 或 *operand2* 的某个（或者两个）操作数中被设置为 1，那么结果中该比特位就被设置为 1。将下面的例子同前面在位逻辑 AND 操作符中所举的例子相比较。

在这个例子中，结果中只有第1位被设置为0，因为这个比特位在两个操作数中都是0。其他比特位被设置为1：

```
1101  
+ 0100  
-----  
1101
```

在这个例子中，结果中所有的比特位都被设置为1，因为所有的比特位在两个操作数中都至少包含一个1：

```
1101  
+ 1011  
-----  
1111
```

在真实的代码中，这也就是：

```
13 | 4      // 结果为 13  
13 | 11     // 结果为 15
```

在实际中，我们经常使用位逻辑 OR 来把表示单个选项的多个数字合并起来，变成表示所有系统选项的一个单一数值。例如，下面的代码将2位（值为4）和3位（值为8）合并起来：

```
options = 4 | 8;
```

位逻辑OR操作符也用在一个已经存在的值中将选项设置为true。下面的例子将由3位（值为8）所表示的选项设置为true。如果3位的值已经为true，它就保持原状：

```
options = options | 8;
```

多个比特位也可以一次进行设置：

```
options = options | 4 | 8;
```

## 位逻辑 XOR

我们已经在操作符中正式使用了各种奇怪的标点符号。位逻辑XOR（互斥OR）操

操作符是一个脱字符号`^`（在大部分键盘中同时按 Shift 和 6 键就可以创建）。位逻辑 XOR 表达式的格式为：

```
operand1 ^ operand2
```

操作数可以是任何数字，但是在进行操作之前，它们会被转换为 32 位的二进制数字。如果操作数存在小数部分，会将其丢弃。

位逻辑 XOR 操作符同位逻辑 OR 操作符是不同的，对于两个操作数都为 1 的比特位，结果中得到的是 0 而不是 1。换句话说，对于任何在两个操作数中都相同的比特位，结果就为 0，对于两个操作数中包含不同数字的比特位，结果位就为 1。

在这个例子中，第 0 位和第 3 位在两个操作数中都是相同的，因此结果中的这些数位就为 0。第 1 位和第 2 位在两个操作数中不同，因此在结果中就为 1：

```
1011  
^ 1101  
-----  
0110
```

在这个例子中，两个操作数的所有位都相同，因此结果全部为 0：

```
0010  
^ 0010  
-----  
0000
```

在这个例子中，第 0、第 2 和第 3 比特位在两个操作数中是不同的，因此这些位在结果中就是 1。但是 1 位在两个操作数中是相同的，因此结果中的第 1 位就是 0：

```
0110  
^ 1011  
-----  
1101
```

转换为十进制数字，前面的例子就是：

```
11 ^ 13      // 结果为 6  
2 ^ 2        // 结果为 0  
6 ^ 11       // 结果为 13
```

位逻辑 XOR 操作符的典型使用是在 0 和 1 (`false` 和 `true`) 之间进行选项的来回切换。要切换用第 2 个位表示的选项 (值为 4)，我们可以使用：

```
options = options ^ 4;
```

## 位逻辑 NOT

和位逻辑 AND, OR 和 XOR 不同, 它们都从两个操作数产生数字结果, 而位逻辑 NOT 将改变单个数字的比特位。它使用代字号 (~), 在一般键盘的左上角都可以找到, 它的格式为:

```
~operand
```

操作数可以是任何数字, 但是在进行操作之前会转换成 32 位的二进制整数。如果操作数有小数部分, 会将其丢弃。

位逻辑 NOT 只改变操作数各个位的值。例如:

```
~000000000000000000000000000010  
// 结果是: 111111111111111111111111111101  
  
-11111111111111111111111111010  
// 结果是: 0000000000000000000000000000101
```

在十进制里就是:

```
-2      // 结果为 -3。参见下面的说明。  
--6     // 结果为 5。参见下面的说明。
```

在这里讨论负的二进制数表示系统是不合适的, 但是高级程序员会注意到, 位逻辑操作使用二进制补码系统来表示负的二进制数。对这些符号不熟悉的读者, 只需要记住: 位逻辑 NOT 操作的返回值是通过将原来的操作数取负, 然后减去 1 所得到的值。例如:

```
--10    // 改变 -10 的符号, 将其变为 10、然后减 1, 结果为 9。
```

位逻辑 NOT 操作符的典型使用是和位逻辑 AND 操作符一起, 清除指定比特位 (也就是将它们设置为 0)。例如, 要清除第 2 位, 我们使用:

```
options = options & ~4;
```

表达式 ~4 返回一个 32 位的整数, 除了 2 位上是 0 之外, 其他位都是 1。位逻辑 AND 对这个返回的数字和 options 变量进行操作, options 的第 2 位被清除, 其他位不变。前边的代码可以写得更简洁:

```
options &= ~4;
```

我们可以用同样的技术来一次清除多个比特位。下面的例子将清除 2 和 3 位：

```
options &= ~(4 | 8);
```

## 位逻辑位移操作符

正如我们看到的，位逻辑编程将二进制数当作一个开关系列，它常常有效地变换这些开关。例如，如果第 0 位是打开的，我们决定将它关上，并且将第 2 位打开，我们可以简单地将第 0 位向左移动两位。如果要知道数字中的第 5 位是否打开，我们可以将这个位向右移动五位，然后检查第 0 位的值。位逻辑位移操作符让我们可以执行这样的移动。

位逻辑位移操作符也可以让我们快速地乘以或者除以 2 的倍数。如果想要将一个十进制数除以 10，可以简单地将小数点向左移动一位。类似的，要乘以 10，可以将小数点向右移动一位，要乘以  $10^3$ （也就是 1000），可以将小数点向右移动三位。位逻辑位移操作符让我们对二进制数字执行类似的操作。将数位往右边移动一位相当于除以数字 2。将数位向左移动一位相当于乘以数字 2。

### 带符号右移

带符号右移操作符可以用来除以 2 的某次方。它使用 `>>` 符号（用两个大于符号来创建），一般格式为：

```
operand >> n
```

`n` 指定要将 `operand` 的数位向右移动多少位。其结果等于将 `operand` 除以 2 的 `n` 次方。如果有余数，会将其省略。原理如下：

所有的数位右移由 `n` 指定的位数。任何移出了右边边界的数位都会被丢弃。新的数位被添加到左边，以填补因为移动而造成的数位空缺。如果 `operand` 是正数，那么新添加的数位都为 0。如果 `operand` 是负数，那么新添加的数位都为 1（因为负数是用二进制补码来表示的）。下面是模拟代码示意：

```
// 最右边的数位 (0) 被丢弃，在左边补上 0  
// 结果为 00000000000000000000000000000000100  
000000000000000000000000000000001000 >> 1
```

将数字向右移动一位就相当于除以2。在十进制中也就是：

```
8 >> 1 // 结果为4
```

注意，任何余数都会被丢弃：

```
9 >> 1 // 结果仍然是4
```

对于负数来说，>> 仍然相当于每移动一位就是除以2：

```
-16 >> 2 // 结果为 -4 (-16 除以 2 的平方)
```

## 无符号右移

无符号右移操作符是用三个连续的大于符号构成的 (>>>)，格式如下：

```
operand >>> n
```

它和带符号右移操作符类似，只是移动中腾出的左边的空位通常都用0来填补（不管 *operand* 是正数还是负数）。对于正数来说，它和带符号的右移没有什么区别。

## 左移

左移操作符可以用来将一个数字乘以2的某次方。它使用<< 符号（使用两个连续的小于符号来创建），一般格式为：

```
operand1 << n
```

n 指定要将 *operand* 向左移动多少位。结果等于将 *operand* 乘以2的n次方。下面是原理说明：

所有的数位向左移动由n指定的位数。移出左边界的数字都会被丢弃。因为左移空出来的数位用0来填补。例如：

```
01000000000000000000000000000001001 << 4  
// 结果为 000000000000000000000000000000010010000
```

将一个数字向左移动四位相当于乘以2的4次方（也就是16）。在十进制中也就是：

```
9 << 4 // 结果为 9 * 16，也就是 144
```

注意，在前边的例子中，我们“手动”指定和每个数位相联系的值：第0位为1，第1位为2，第2位为4，第3位为8，依此类推。左移操作符对计算数位等价的值非常方便：

```
(1 << 0)      // 第0位等于1  
(1 << 1)      // 第1位等于2  
(1 << 2)      // 第2位等于4  
(1 << 3)      // 第3位等于8  
(1 << 15)     // 比记住第15位为32768要容易多了!
```

左移操作符使用数字引用而不是数位值来进行动态选择比特位也很方便。例15-1统计数字中1的个数。

#### 例 15-1：使用左移方法来计算被设置的数位总数

```
myNumber = 27583; // 就是我们要计算的数字  
count = 0;  
for (var i=0; i<32; i++) {  
    if (myNumber & (1 << i)) {  
        count++;  
    }  
}
```

例 15-2 是例 15-1 的一个变体，它使用的是右移操作符。可以将值重复右移，检查它最右边的数位(第0位)，而不是使用左移操作符来计算和每一个数位相关的位值。

#### 例 15-2：使用右移计算位数

```
myNumber = 27583  
count = 0;  
temp = myNumber; // 建立一个临时使用的副本  
for (var i=0; i < 32; i++) {  
    if (temp & 1) {  
        count++;  
    }  
    temp = temp >> 1;  
}
```

变量 myNumber 被复制到临时变量 temp 中，因为右移运算是具有破坏性的，变量 temp 最后结束的时候值将为0。

## 位逻辑运算的应用

我们开始讨论位逻辑操作符的时候举了一个Flash站点的例子，它是销售汽车的。既

然我们已经学习了位逻辑操作符的工作原理，现在我们要用它来确定汽车的价钱，如例 15-3 所示。可以从在线代码库中下载这个例子的 .fla 文件。

### 例 15-3：位逻辑运算的真实演练

```
// 首先，要设置选项（通常都基于填充表单的选择来添加或者删减数字，  
// 但是我们现在把它写在下面）  
var hasAirCon = (1<<0) // 第 0 位：0 代表否，1 代表是  
var hasCDplayer = (0<<1) // 第 1 位：0 代表否，2 代表是  
var hasSunRoof = (1<<2) // 第 2 位：0 代表否，4 代表是  
var hasLeather = (1<<3) // 第 3 位：0 代表否，8 代表是  
  
// 现在使用位逻辑 OR 将选项合并为一个单独的数字  
var carOptions = hasAirCon | hasCDplayer | hasSunRoof | hasLeather;  
  
// 这是一个函数，用来计算价钱  
function totalPrice(carOptions) {  
    var price = 0;  
    if (carOptions & 1) { // 如果第一个数位被设置就加 1000 美元  
        price += 1000;  
    }  
    if (carOptions & 2) { // 如果第二个数位被设置就加 500 美元  
        price += 500;  
    }  
    if (carOptions & 4) { // 如果第三个数位被设置就加 1200 美元  
        price += 1200;  
    }  
    if (carOptions & 8) { // 如果第四个数位被设置就加 800 美元  
        price += 800;  
    }  
    return price;  
}  
  
// 万事俱备：我们来调用函数，看看它是否会运行！  
trace(totalPrice(carOptions)); // 返回 3000 真酷...
```

要避免在代码中费力地输入数位值，可以将和选项相对应的数位值存储在变量中，如下所示：

```
var airConFLAG = 1<<0; // 第 0 位，值为 1  
var cdPlayerFLAG = 1<<1; // 第 1 位，值为 2  
var sunroofFLAG = 1<<2; // 第 2 位，值为 4  
var leatherFLAG = 1<<3; // 第 3 位，值为 8
```

读者练习：使用变量和左移操作符重新编写例 15-3，不费力地输入表示选项的数位值。

## 为什么使用位逻辑？

虽然例15-3作为一个布尔运算系列会比较容易理解，但是位逻辑运算显得更快更简洁。在任何时候都可以用计算机特有的二进制方式和它交流，可以节省空间，提高速度。

如果要进行对比，可以考虑这样的一个情况，就是我们正在追踪一个用户的信息，每个用户都有 32 个设置可以开关。在普通的数据库中，我们对每个用户需要有 32 个域。如果我们有一百万个用户，那么就需要 32 个域的一百万份拷贝。但是，如果我们使用位逻辑编程，就可以将 32 项设置存储在一个单独的数字中，每个用户在数据库中只需要一个域！这不仅可以节约硬盘空间，而且每次访问用户信息的时候，只需要传输一个单独的整数，而不是 32 个布尔值。如果进行的是一百万份的传输，每一份传输就可以节省好几毫秒，这样就可以大大提高我们的执行效率。

如果你要深入学习这方面的内容，请参见 Gene Myers 献给 C 程序员的优秀文章，《Becoming Bit Wise》，在 <http://www.cscene.org/CS9/CS9-02.html> 上可以找到。

## 高级函数作用域问题

这一节要描述另外一个高级主题，它对于我们最初讨论的函数作用域来说是非常普通的。既然我们已经学习了影片剪辑、函数作用域和对象，现在我们就重新来看看这个问题。

我们在第九章中学到，一个函数的作用域链通常是由函数的声明语句来确定的。但是，这个规则中可以有一个微妙的扩展。当一个时间线上的函数被赋给不同剪辑时间线上的某个变量的时候，这个赋值动作也会影响函数的作用域链。如果原来的函数是被直接调用的，那么它的作用链就包括它最初的时间线，但是，如果函数是通过变量来调用的，那么它的作用域链就包括变量的时间线。

例如，假设我们创建一个名为 *transformClip()* 的函数，它对当前剪辑进行旋转和缩放操作。我们在变量 *rotateAmount* 和 *widthAmount* 中设置旋转和缩放的数量：

```
var rotateAmount = 45;
var widthAmount = 50;

function transformClip() {
```

```
_rotation = rotateAmount;  
_xscale = widthAmount;  
}  
  
// 调用函数  
transformClip();
```

下面，我们将 transformClip 赋给一个变量 tc，它在剪辑 rect 中：

```
rect.tc = transformClip;
```

当通过 rect.tc 调用 transformClip 函数的时候，并不会产生什么结果，如下所示：

```
rect.tc();
```

为什么？存储在 tc 中的函数的作用域链包含 rect，而不是原函数的时间线，因此，就找不到 rotateAmount 和 widthAmount。但是，当我们把 rotateAmount 和 widthAmount 变量加到 rect 中的时候，函数就可以找到变量了，因此它就是有效的：

```
rect.widthAmount = 10;  
rect.rotateAmount = 15;  
rect.tc(); // 将 rect 设置为 10% 宽度，并旋转 15 度
```

相反，当同一时间线上的正规数据对象在函数赋值中被调用的时候，函数的作用域链并不会改变；实际情况是，函数的作用域链的确定永远和函数的声明有关。例 15-4 给出了示例。

#### 例 15-4：对象方法固定的作用域

```
// 设置我们的变量  
var rotateAmount = 45;  
var widthAmount = 50;  
  
// 创建一个 transformClip() 函数  
// 该函数用来显示 rotateAmount 和 widthAmount 的值  
function transformClip () {  
    trace(rotateAmount);  
    trace(widthAmount);  
}  
  
// 创建一个对象，和前边例子中的 rect 剪辑一致  
var rectObj = new Object();  
  
// 将 transformClip 复制给 rectObj 的一个属性  
rectObj.tc = transformClip;
```

```
// 在rectObj中设置rotateAmount 和 widthAmount 属性  
rectObj.rotateAmount = 15;  
rectObj.widthAmount = 10;  
  
// 现在调用rectObj.tc  
rectObj.tc();           // 显示 45 和 50，而不是 15 和 10。  
                        // rectObj.tc 的作用域和 transformClip() 相同
```

赋给一个对象属性的时候，函数的作用域在包含函数声明的时间线上。但是，被赋给一个远程影片剪辑的时候，函数的作用域就是远程剪辑的时间线。

注意，这种行为实际上是和 JavaScript 相违背的，在 JavaScript 中，函数的作用域永久属于函数声明所在的对象。例如，如果我们假定 HTML 框架结构中的框架类似于 Flash 影片中的一个剪辑，那么我们就可以看出差异。在 JavaScript 中，赋给一个远程框架的函数，其作用域仍然在包含函数声明的框架中，而不是远程框架里，如例 15-5 所示。

#### 例 15-5：JavaScript 的静态函数作用域

```
// 一个框架结构中框架0中的代码  
// 在框架 0 中给一个变量赋值  
var myVar = "frame 0";  
  
// 在框架结构的框架 1 中设置一个函数，将函数复制回框架 0  
parent.frames[1].myMeth = function() { alert(myVar); };  
myMeth = parent.frames[1].myMeth;  
  
// 从框架 0 调用 myMeth()  
myMeth();           // 显示 'frame 0'  
  
// 同一个框架结构中框架1中某个按钮上的代码  
<FORM>  
    <INPUT type="button" value="click me" onClick="myMeth(); ">  
</FORM>  
  
// 现在点击按钮，从框架 1 调用函数 myMeth()。  
// 显示 'frame 0'。  
// myMeth() 的作用域并没有到框架 1，而是包含函数声明语句的框架 0
```

## 影片剪辑数据类型

我们又要进入另外一个深奥的主题，以掌握 ActionScript 的基础。在第十三章中，我们学习了影片剪辑的行为，它在很大程度上都和对象类似。但是，影片剪辑并不只

是另外一个类——它们属于自己的独特数据类型。ActionScript 的创建者 Gary Grossman 解释了影片剪辑和对象数据类型内在实现之间的差异，如下所示：

影片剪辑在播放器中的实现是和对象分开的，虽然两者在 ActionScript 中显得几乎是相同的。主要的差别在于它们被分配和解除分配的方式不同。正规对象采用的是引用计数和垃圾回收的方式，而影片剪辑的生命周期是由时间线控制的，或者由 *duplicateMovieClip()* 和 *removeMovieClip()* 函数来精确控制。

如果用 `x=new Array()` 来声明一个数组，然后设置 `x=null`，ActionScript 会马上探测出没有其他引用指向该 `Array` 对象（也就是没有指向它的变量），然后就会对它进行回收（也就是释放它所使用的存储器空间）。定期做记录和清扫的垃圾回收消除了包含循环引用的对象（也就是说，用这个高级技术来确保当两个无用的对象互相指向对方的时候将它们的空间释放掉）。

影片剪辑的情况有所不同。它们依靠时间线上对象的放置而出现或者消失。如果它们是动态创建的（用 *duplicateMovieClip()* 创建），它们就只有当使用 *removeMovieClip()* 的时候才会被除去。

指向对象的是指针（存储器地址引用），引用追踪和垃圾回收保护用户不让引用和存储空间泄露。但是，指向影片剪辑的引用是“软”引用——这个引用其实包含的是一个绝对目标路径。如果你有一个影片剪辑 `foo`，并设置 `x=foo`（将 `x` 设置为指向 `foo` 的引用），然后用 *removeMovieClip()* 来删除 `foo`，并创建另外一个名叫 `foo` 的剪辑，引用 `x` 就又可用了（它会指向新的 `foo` 剪辑）。

正规对象则不同——对象引用防止对象被首先删除。因此，如果影片剪辑是对象，只要变量 `x` 指向它，*removeMovieClip()* 就不会从存储空间中删除对象。此外，如果你创建了第二个名为 `foo` 的影片剪辑，旧的 `foo` 和新的 `foo` 可以同时存在，虽然旧的 `foo` 不会再得到渲染。

因此，一个独特的 `movieclip` 类型是恰当的，因为它和 `object` 类型有这么大的差别。由于类似的原因，`typeof` 操作符对函数会返回“function”，虽然函数在很大程度上和对象类似。

## 小结

第一部分“ActionScript 基础”结束了。在第二部分中，我们要学习一些重要的实际创作技巧，以及如何建立 Flash 表单，如何调试代码，要像说母语一样说 *ActionScript*。你也需要浏览第三部分，来扩充你的词汇量。第三部分描述了 *ActionScript* 的内置函数、属性、对象和类。

---

## 第二部分

# ActionScript 应用

这个部分描述了 Flash 编程中更实际的方面，比如使用制作环境和调试器。本部分还讨论了 Flash 编程中两个特殊的领域：建立 Flash 表单和使用屏幕文本域。

- 第十六章 ActionScript 制作环境
- 第十七章 Flash 表单
- 第十八章 屏幕文本域
- 第十九章 调试



# 第十六章

## ActionScript 制作环境

本章将讨论制作 ActionScript 代码的实际细节，主要有以下内容：

- 用动作（Actions）面板将代码放置到按钮、影片剪辑和帧上。
- 从外部文件载入代码。
- 用智能剪辑将代码封装为可重复使用的制作成分。

### 动作面板

动作面板是 Flash 的 ActionScript 编辑环境。影片的所有脚本都是在动作面板中创建的，可以通过 Window → Actions 命令打开动作面板。

动作面板被分为两个部分：工具箱（Toolbox）框（在左边）和脚本（Script）框（在右边），如图 16-1 所示。

脚本框包含添加到当前所选帧、按钮或影片剪辑的代码。工具箱既是一个快速参考指南，又是对脚本框添加代码的一种途径。双击工具箱中的任何项目，就会将该项目添加到脚本框。项目也可以从工具箱里拖到脚本框中。

动作面板的标题表明了当前的代码是存在于一个帧上（帧动作），还是在按钮或者影片剪辑上（对象动作）。当选择了一个帧的时候，动作面板的标题就自动转换为帧动

作 (Frame Actions); 当选择了一个影片剪辑或一个按钮的时候, 动作面板的标题就变成对象动作 (Object Actions)。

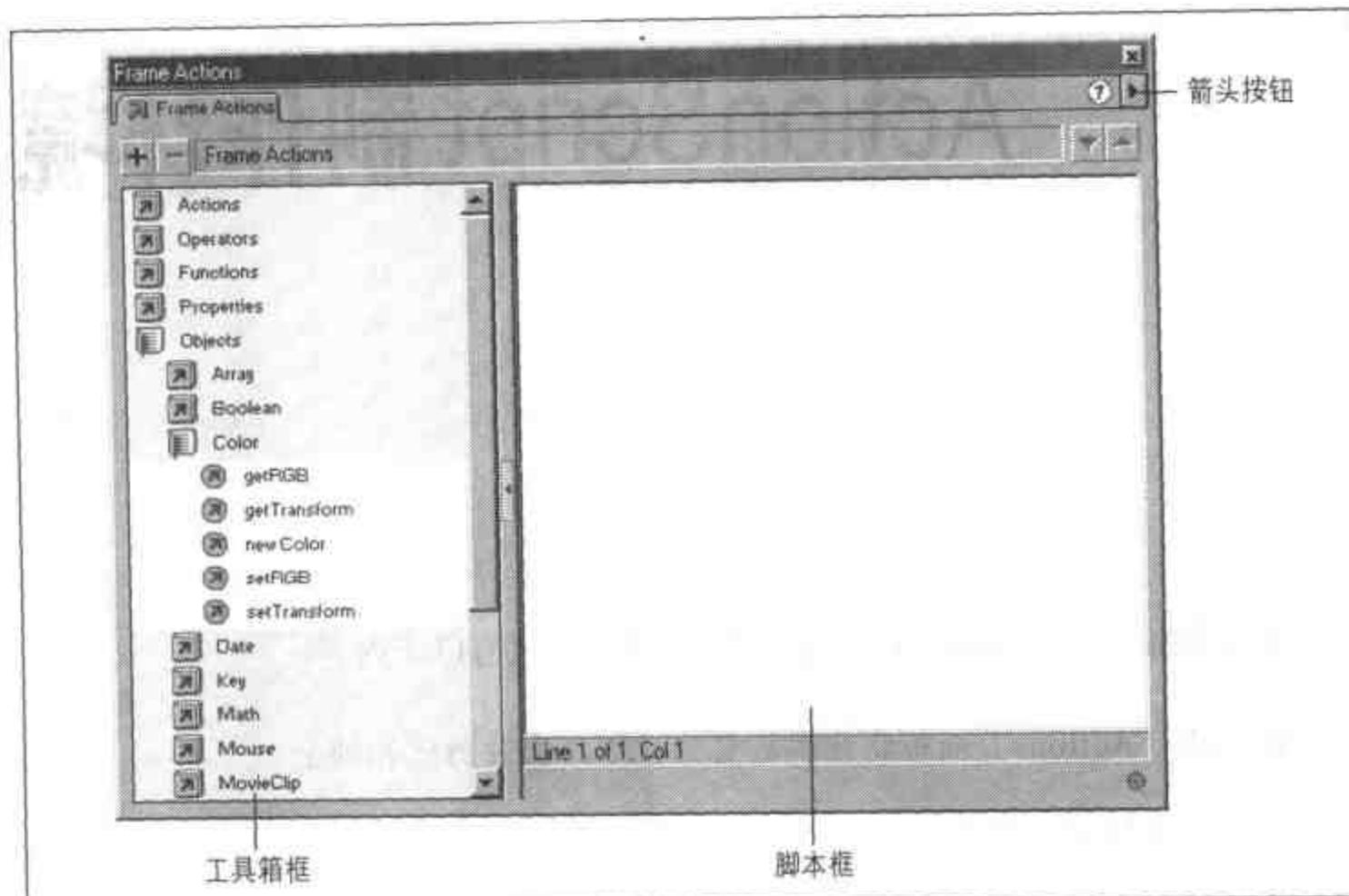


图 16-1 动作面板

工具箱中各项目的组织和本书中用来描述 ActionScript 功能的种类有所不同。特别是, 在工具箱中, 语句没有被分离在自己的文件夹中, 类和对象被一起放到了 *Objects* 文件夹中。在本书中, 我们将语句、类和对象区分开, 以将编程技术更为正式地集中起来。

## 编辑模式

动作面板有两种不同的操作模式, 普通模式 (Normal Mode) 和专家模式 (Expert Mode), 它们控制我们为脚本框添加代码的方式。

### 普通模式

在普通模式中, 工具箱用来建立新的语句, 而脚本框是这些语句的浏览器。要在普

通模式里添加一个新的语句到脚本框，可双击所要执行的动作，或者从工具箱中拖动一个动作到脚本框。要添加任意代码的一个语句，可选择 *evaluate* 动作，然后将语句输入到动作面板底部的参数框 (Parameters pane) 中的表达式 Expression域 (图中没有显示出来)。

在对脚本框添加一个语句之后，可以通过动作面板底部的参数框来定制它。参数框的规划和内容对应于脚本框中被选择的语句而发生改变。注意，当动作面板在普通模式中操作的时候，代码不可以输入到脚本框中。在普通模式里，脚本框不是文本编辑环境，而是一个只读的语句列表。要改变脚本框中的语句，应通过参数框来完成。

虽然 Flash 5 中的普通模式看起来和 Flash 4 中的动作面板非常相似，但是它不应该被误解为是向后兼容的模式。相反，在任何一种模式中创建的代码都有可能是和 Flash 4 不兼容的（参见附录三）。普通模式对于新的程序员来说很不错，但是却压缩了一些最重要的编程设计。因此，在本书中，我们只使用专家模式来建立代码。但是，除了多行注释（它只可以在专家模式中使用）之外，任何在某个模式中创建的代码都可以在另外一个模式中产生。

## 专家模式

在专家模式中，脚本框将作为一个传统的文本编辑窗口。当我们在专家模式中制作或修改代码的时候，可以直接将代码输入到脚本框里。但是，我们仍然可以双击工具箱中的项目将它们添加到脚本框。在专家模式中不使用参数框，而要将参数直接输入到脚本框中。

## 设置编辑模式

动作面板的编辑模式是在每帧或每个对象的基础上设置的。也就是说，Flash 会记住影片中每个对象和帧所选择的动作面板编辑模式。如果我们为第 2 帧选择了普通模式，为第 3 帧选择了专家模式，动作面板会在我们编辑第 2 帧代码的时候自动转换到普通模式，在编辑第 3 帧代码的时候又转换到专家模式。

要为单个的帧或对象选择模式，可从动作面板右上角的箭头按钮中选择 Expert Mode 或者 Normal Mode（见图 16-1）。要为影片中所有的帧和对象选择默认模式，我们

在 Edit → Preferences → General → Actions Panel → Mode 中选择 Expert Mode 或者 Normal Mode。注意，默认的模式设置只影响那些其编辑模式还没有设置的帧和对象。因此，默认模式可以在影片产生的开始进行设置。我们没有办法对已经存在的帧和对象所预先设置的单个模式进行全局的覆盖。

---

**警告：**从专家模式转换到普通模式会破坏所有的源代码格式。代码会按照 Flash 的标准格式规则进行重新格式化——过宽的空白会丢失，注释被放在它们自己的行中，语句块缩排被调整，就像本书所使用的格式一样。

---

## 为帧添加脚本

Flash 文档结构基本上都是围绕动画的概念来建立的。每一个 Flash 文档由一个帧的线性序列构成，或者由视觉和听觉内容的片段构成。大量的 ActionScript 代码依赖于这种以帧为基础的结构。我们将一个代码块放到帧上，就规定了该块同影片播放头有关的执行方式。不管我们什么时候对帧添加代码，都必须考虑我们想让代码做什么，以及我们要让代码在什么时候执行。

例如，我们想编写一些代码，在影片第 20 帧开始显示的时候将一个新的影片剪辑放到场景上：

```
_root.attachMovie("myClip", "c_1pl", 0);
```

为了让代码能在显示第 20 帧的时候执行，必须将它添加到影片第 20 帧上的关键帧中。要将代码添加到关键帧，我们选择时间线上的对应关键帧，然后打开动作面板，将所需的代码输入到脚本框。在播放时，关键帧中的代码会在帧的内容显示之前执行。

关键帧上的代码用来执行和影片播放同步的任务，建立影片剪辑或影片需要使用的变量、函数和对象。时间线循环就是同步任务的简单例子。假设将下面的代码添加到了影片的第 15 帧上：

```
gotoAndPlay(10);
```

当影片的播放头到达第 15 帧的时候，代码得到执行，影片又从第 10 帧开始播放。当

第二次到达第 15 帧的时候，代码又被执行，影片又重新从第 10 帧开始播放。这会让影片在第 10 帧和第 15 帧之间无限循环。

但是，关键帧上的代码并不总是控制或者同步于影片的播放头。也可以将关键帧只用作函数、变量、对象和其他程序实体的存储设备。例如，可以将函数 *moveTo()* 添加到影片剪辑的第 1 帧，然后从影片后面的按钮调用 *moveTo()* 函数：

```
function moveTo (x, y) {
    _x = x;
    _y = y;
}
```

警告：因为帧上代码的执行是由影片播放头决定的，我们必须确保变量、函数和其他程序元素在它们被访问的时候是可用的。例如，如果一个函数要到第 10 帧才会得到定义，那么我们不能在第 3 帧调用这个函数。因此，在影片中全局使用的代码应该放在主时间线的第 1 帧。

我们还必须确保没有代码试图访问还没有装载的影片部分。要检查影片中一个特定的部分是否被装载，可以使用影片剪辑属性 *\_framesloaded* 或 *getBytesLoaded()* 方法。对于示例代码，参见第三部分中关于 *Movieclip*, *\_framesloaded* 的条目。

## 对按钮添加脚本

我们为按钮添加代码，以便将代码的执行同用户事件联系起来。例如，如果要让按钮的点击将影片播放头推进到帧 section1，可以将下面的代码添加到按钮上：

```
on (release) {
    gotoAndStop('section1');
}
```

要将代码添加到按钮，我们选择场景上的按钮，然后在动作面板中将代码添加到脚本框中。按钮上的代码必须总是放在能识别代码执行环境的事件处理器上。例如，大部分触发按钮动作的事件是 *release*。我们也可以用 *rollOver* 事件，它在鼠标从按钮上移过的时候执行相应的代码，而不是在按钮被按下的时候执行：

```
on (rollOver) {
    gotoAndStop("section1");
}
```

对于按钮事件处理器的详细描述，请参见第十章。

虽然在按钮上放置几千行代码是合法的，但是让按钮的代码超载并不好。只要可能，我们应该将按钮代码的行为集中，封装到函数中，将函数放在按钮时间线上。例如，可以将下面的代码添加到按钮上：

```
on (release) {
    title._xscale = 20;
    title._yscale = 20;
    title._alpha = 50;
    title.gotoAndPlay("fadeout");
}
```

但是最好将代码放到函数中，然后从按钮调用这个函数：

```
// 按钮时间线第1帧中的代码
function transformClip(clip, scale, transparency, framelabel) {
    clip._xscale = scale;
    clip._yscale = scale;
    clip._alpha = transparency;
    clip.gotoAndPlay(framelabel);
}

// 按钮上的代码
on (release) {
    transformClip(title, 20, 50, "fadeout");
}
```

这个方法可以让代码显得很集中、容易维护，并且可以以最小的努力将按钮行为快速应用到多个按钮上。

---

**注意：**按钮代码必须被添加到场景中的对象上，必须包括事件处理器。没有事件处理器的按钮代码会引起错误，不能将代码添加到按钮符号的内部 UP、OVER、DOWN 和 HIT 帧中。

---

## 为影片剪辑添加脚本

我们已经知道可以为影片剪辑时间线上的帧添加代码。其实，为影片剪辑对象本身添加代码也是可以的。要这么做，我们先选择场景上的影片剪辑实例，然后在动作面板的脚本框中为它添加代码。和按钮一样，所有属于影片剪辑对象的代码包含在事件处理器内。事件处理器告诉解释程序什么时候执行这些代码。例如，下面的代码在某个影片剪辑载入的时候将变量 x 设置为 10：

```
onClipEvent (load) {  
    var x = 10;  
}
```

影片剪辑事件处理器可以对鼠标和键盘的动作、数据的装载、帧的渲染、以及影片剪辑的产生和死亡作出反应。关于影片剪辑和事件处理器的全面论述，请参见第十章以及第十三章。

## 代码都在哪里

即使经验丰富的 Flash 用户，在影片中将代码定位有时候也有困难。因为代码可以添加到任何时间线的任何帧上，可以添加到任何按钮、场景上的任何影片剪辑中，因此，它经常会被淹没在影片内容的海洋中。显然，一个高度组织化的结构和详细的代码文件意味着可以在设计中节省很多时间。但是，如果你的影片看起来似乎丢失了什么重要代码，你可以打开动作面板，使用下面的方法来寻找：

- 点击那些在时间线上有一个小圆圈图标的帧。小圆圈表示有 ActionScript 代码的存在。
- 寻找并选择场景上任何带黑边的白圆。这些圆表示空的影片剪辑，它们通常只包含一些代码。如果在空剪辑本身中没有代码，那么就双击剪辑来进行编辑并研究一下它的帧。
- 选择影片中所有的按钮，一次一个。一些程序员不习惯将代码进行集中处理，他们会将长长的、重要的脚本直接放在按钮上。
- 检查隐藏层和蒙板层的时间线。后面带红色 X 图标的层在创作期间隐藏了起来，但是可能包含一些会在影片播放期间出现的带代码的剪辑和按钮。类似的，蒙板层可能包括带代码的隐含对象。将蒙板层解锁，就可以得到它们的内容。
- 将所有的层解锁。空的影片剪辑（有带黑边的小圆圈）当所在的层被锁住的时候就会被隐藏起来。

有的时候，即使这些技术用尽了我们仍然找不到所要的代码。如果某人决定要隐藏某些代码，在 Flash 中有很多地方可以这样做。例如，一个空的剪辑可以放在场景限制之外很远的地方，几乎无法被找到。幸运的是，即使看起来所有的东西都丢了，

我们的代码也不会丢——我们总是可以用影片浏览器来捕获影片中的任何脚本。选择 Window → MovieExplorer（影片浏览器）浏览一下影片中的项目，包括属于帧、按钮或者影片剪辑的所有脚本。脚本有一个蓝箭头动作图标标记，和动作面板工具箱中项目里的图标是一致的。你甚至可以过滤显示，只显示脚本，选择 Explorer（浏览器）面板顶部 show（显示）菜单下的 Actions 图标，然后取消对所有其他图标的选 择。

## 生产力

下面是流线型的 ActionScript 源代码创建的一些技巧：

- 将所有时间线脚本放在独立的 *scripts* 层中。不要将任何影片内容放在这个层上，专门用它来容纳代码。将这个层存储在层结构的顶部，这样，它的代码就会在所有其他层都载入之后才执行（或者放在底部，如果将载入顺序设置为从上到下的话）。如果你总是将 *scripts* 层放在同样的位置，就很容易找到你的代码。（我们从前面对的章节中知道这有多么重要！）
- 将所有的帧标签放在一个独立的层 *labels* 中。不要在 *labels* 层放置其他内容，专门用它来放帧标签。
- 看一看动作面板右上角箭头按钮下的东西。你会发现诸如搜索和替换、源代码指示和脚本框字体控制这样一些方便的工具。
- 当处理用于多个设计的代码库时，可以将代码保存到外部文件中。参见下一节以获得详细的内容。
- 要节省输入工作，使用动作快捷键（例如，Esc-G-P 代表 *gotoAndPlay()*）。快捷键序列被列在动作面板的 + 按钮下。
- 改装动作面板，再放上其他的核心代码面板：实例、帧和文本选项。只要在 Window → Panels 下打开任何面板，然后将它拖动到动作面板上。
- 注意，工具箱是可以重新设置大小的。要在编码的时候节省空间，可以将工具箱部分或者全部隐藏起来，只要拖动工具箱和脚本框中间的边界即可。

## 外在化 ActionScript 代码

ActionScript 代码可以存储在外部文本文件中（通常使用 *.as* 扩展名）或者 *.swf* 文件中，然后导入到 Flash 文档。通过将代码放在外部文件里，我们就可以推进跨越多个设计的标准代码库的使用。我们可以使用 Import From File（从文件导入）命令、*#include* 指示，或者用共享库将外部代码导入到 Flash。

### 从文件导入（制作中的导入）

在编辑一个 *.fla* 文件时，可以使用“从文件导入”命令将代码导入到动作面板的脚本框，这个命令在动作面板右上角的箭头按钮下（参见图 16-1）。“从文件导入”一次只能操作一个文件，它将外部文件的内容复制到动作面板中，代替原本在那里的任何脚本。通过“从文件导入”导入的代码并非一直和 *.fla* 文件长久链接。要附加或者插入脚本文字，而不是代替原有的脚本，需要从外部文本编辑应用程序中手动剪切和粘贴文本。

### #include（编辑中的导入）

当我们从 *.fla* 文件导出（编辑）一个 *.swf* 文件的时候，可能会使用 *#include* 指示从外部文本文件中导入代码。对于 *#include* 的使用，参见第三部分。

### 共享库（运行中的导入）

要在影片播放的时候从外部导入代码，必须创建一个共享库 *.swf* 文件，它有一个包含导入代码的影片剪辑。运行中的导入提供了最灵活的共享影片代码的方法，因为它不要求导入共享代码的影片改编，当共享库 *.swf* 文件被更新的时候，链接到它的影片自动对更新做出反应。

下面的程序描述了如何从影片 *codeLibrary.swf* 到影片 *myMovie.swf* 共享一个简单的测试函数。首先创建 *codeLibrary.swf* 影片：

1. 创建一个新的 Flash 文档。
2. 创建一个新的影片剪辑符号 *sharedFunctions*。

3. 在 sharedFunctions 剪辑的第 1 帧添加下面的代码:

```
function test () {  
    trace("The shared function, test, was called.");  
}
```

4. 在 Library 中选择 sharedFunctions 剪辑。
5. 从 Library 面板中选择 Options → Linkage。
6. 选择 Export This Symbol。
7. 在 Identifier (标识符) 框中, 输入 sharedFunctions。
8. 将文档保存为 *codeLibrary.fla*。
9. 使用 File → Export Movie, 以从 *codeLibrary.fla* 创建 *codeLibrary.swf*。
10. 关闭 *codeLibrary.swf* 和 *codeLibrary.fla* 文件。

现在, 我们要创建 *myMovie.swf* 文件, 它将执行从 *codeLibrary.swf* 中导入的代码:

1. 创建一个新的 Flash 文档。
2. 将新文档存储为 *myMovie.fla*, 和 *codeLibrary.fla* 在同一个文件夹中。
3. 将 Layer 1 重新命名为 *sharedCode*。
4. 选择 File → Open As Shared Library (打开为共享库), 然后选择 *codeLibrary.fla*。就会出现 *codeLibrary.fla* 库。
5. 从 *codeLibrary.fla* 库中拖动一个 sharedFunctions 剪辑的实例到 *myMovie.fla* 第 1 帧的场景上。
6. 选择场景上的实例, 然后选择 Modify → Instance。
7. 将实例命名为 *sharedFunctions*。
8. 在 *myMovie.fla* 的主时间线上, 创建一个新的层 *scripts*。
9. 在 *scripts* 层的第 2 帧添加一个关键帧。
10. 在 *sharedCode* 层添加一个新的帧。
11. 在 *scripts* 层的第 2 帧, 添加下面的代码:

- ```
stop();
sharedFunctions.test();
```
12. 导出 *myMovie.swf*。
  13. 下面的文本将出现在输出窗口中：“The Shared function, test, was called”（共享函数 *test* 被调用）。

注意，共享库是一个链接设备，就像一个.gif图像被链接到一个.html文件一样。因此，共享库.swf文件必须被使用它们的文件正确上载。要改变从导入符号到共享库的链接的位置，执行下面的步骤：

1. 在库中选择一个符号。
2. 从 Library 面板中选择 Options → Linkage。
3. 在 Import This Symbol from URL (从 URL 输入该符号) 下面，设置新的位置。

## 组件打包成智能剪辑

智能剪辑是一种其中的某些代码可以通过 Flash 制作工具中的特殊图形用户界面来进行设置的影片剪辑。智能剪辑允许非程序员来自定义程序化控制的影片剪辑。智能剪辑将行为确定变量同剪辑代码分离开来，让人们将它们当作“黑盒”——它们的操作是不可知的，而它们的输入、输出和行为却是可知的。

通常，变量初始化发生在影片剪辑的源代码中。例如，我们要建立用来控制焰火效果的变量：

```
// 用户定义的变量
var numSparks = 10;           // 爆发出的火花剪辑数量
var randomDispersion = true; // 爆发类型 (true 表示随机式, flase 表示平均式)
var duration = 1300;          // 爆发的时间长度, 以毫秒为单位
```

要修改这种源代码对非程序员来说很困难。但是，如果将我们的体系建立为一个智能剪辑，非程序员就可以通过一个熟悉的应用程序风格的界面来设定焰火效果。图 16-2 给出了和变量初始化代码等同的智能剪辑界面。

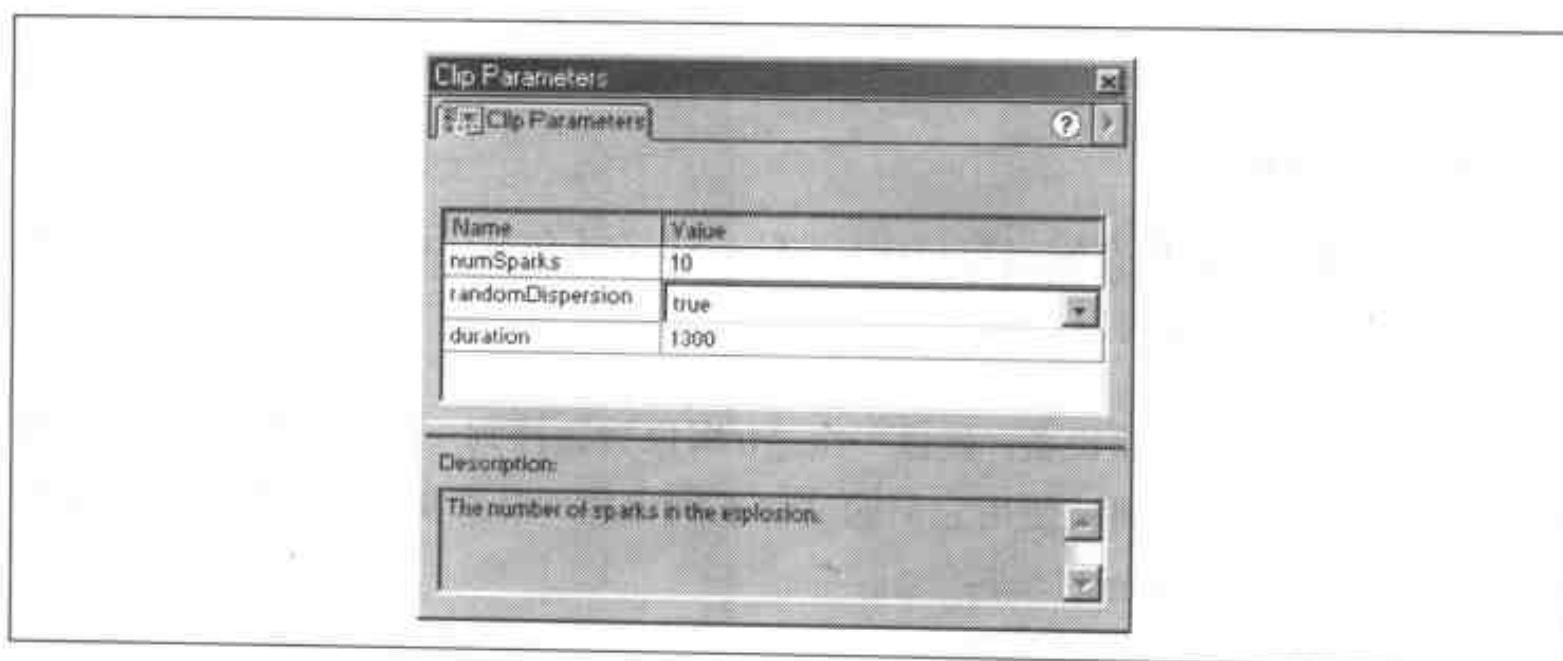


图 16-2 一智能剪辑示例——界面构造

在智能剪辑界面中，每一个变量的名称和值都在单独的行和列中清楚地区分开了。变量名称是不能编辑的，因此也就避免了由于输入有误而对系统造成影响。每一个变量也有它自己的详细描述，说明它要做什么，以及应该如何设置。最后，可以通过下拉菜单对带合法值设置限制的变量（比如 randomDispersion）赋值。

对于非程序员来说，图 16-2 中所给出的界面的确比源代码更容易使用，而且智能剪辑的用户友好程度实际上更高。我们可以用自定义的界面来代替默认的智能剪辑界面，如图 16-3 所示。请注意自定义的智能剪辑界面如何完全隐藏系统变量，允许非程序员利用文本域和下拉菜单来制作焰火的每一个实例。这个界面甚至提供了一个生动的效果预览！

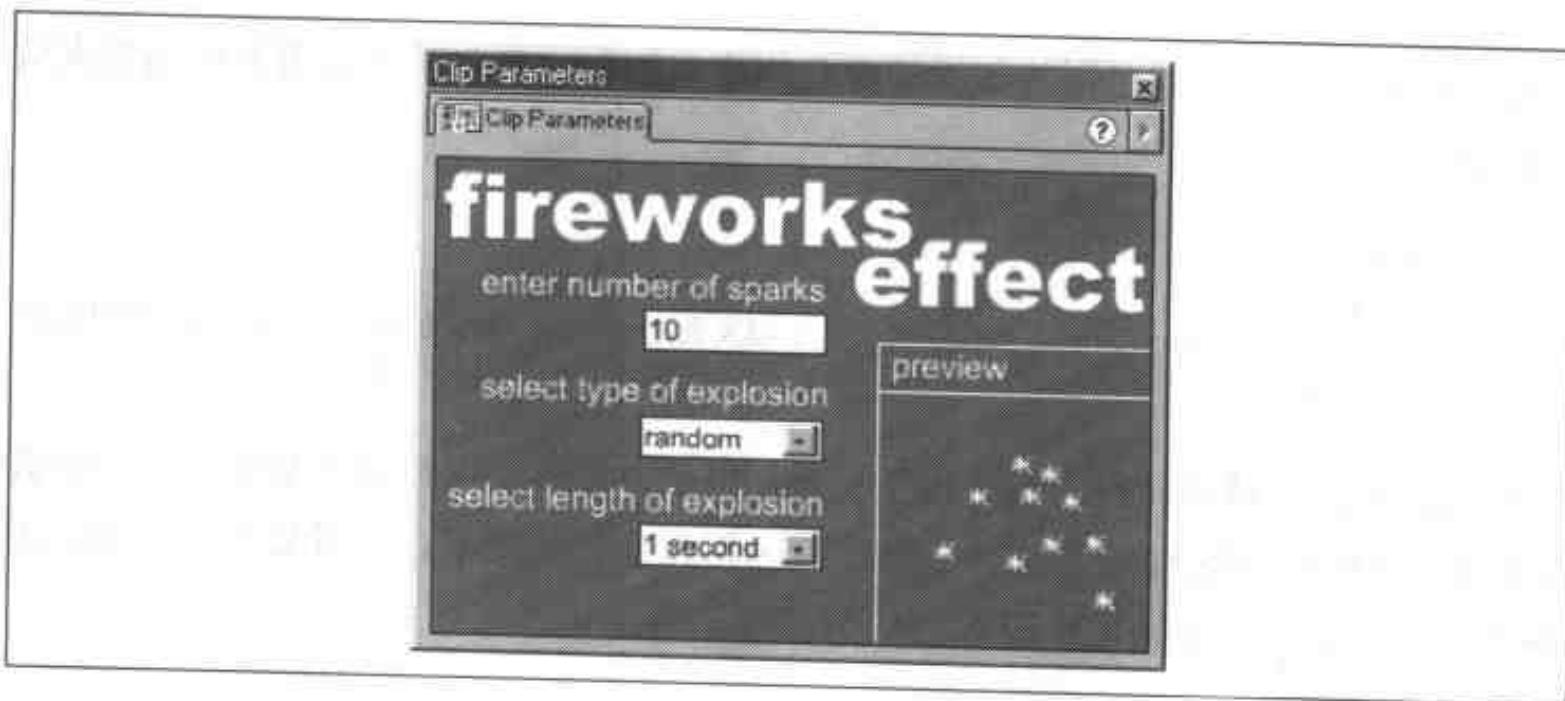


图 16-3 自定义的智能剪辑——界面构造

我们来看看它是怎么实现的。

## 建立一个标准界面的智能剪辑

正如我们刚才所学到的，智能剪辑可以有默认的系统界面，也可以有自定义的界面。我们首先来学习如何建立标准类型。

建立智能剪辑的第一步就是创建一个正规的影片剪辑，它由一个或者多个变量的值来控制。例如，在下面的代码中，变量 `xPos` 和 `yPos` 确定剪辑在场景上的位置：

```
_x = xPos;  
_y = yPos;
```

当我们为自己或者其他人建立一个作为智能剪辑的影片剪辑以供使用的时候，我们希望当剪辑放在场景上的时候，能通过它来设置一些指定的变量。这些变量也就是剪辑参数（clip parameter）。一旦创建了其行为由一个或者多个剪辑参数来表示的剪辑，就必须给剪辑一个智能剪辑的界面，通过这个界面对这些参数进行设置。

### 为智能剪辑添加一个标准界面

要将一个默认的智能剪辑界面添加到一个影片剪辑中，执行下面的步骤：

1. 选择库中的剪辑。
2. 选择 Options → Define Clip Parameters（定义剪辑参数），定义剪辑参数对话框就会出现。
3. 在 Parameters pane 中，点击 + 按钮添加一个剪辑参数。
4. 对智能剪辑里的所有参数重复第 3 个步骤。
5. 设定剪辑参数，如下面小节中所描述的那样。

### 设置标准剪辑参数

在为智能剪辑添加一个剪辑参数之后，我们必须给参数一个名称以及（可选择）一个默认的值。和变量一样，剪辑参数可以包含不同的数据类型。但是，剪辑参数所

支持的数据类型和变量所支持的那些类型并不完全相同。剪辑参数可以包含串、数字、数组、对象和列表。这些类型和变量支持的数据类型相比有两方面的差别：

- 剪辑参数支持称为 *list* (列表) 的类型，它只对界面有效。*list* 用来将参数值的设置限制在一系列预先给出的选项中。例如，参数 `difficulty` 可以将值设定为列表 "hard", "normal", "easy" 中的某个值。列表可以防止智能剪辑的用户对剪辑参数提供一个非法值。
- 原始数据类型布尔、`null` 和 `undefined` 不会直接作为剪辑参数的值来设置，这是智能剪辑界面的一个限制，而不是剪辑参数本身的限制。剪辑内的代码可以将布尔、`null` 或 `undefined` 值赋给初始化为剪辑参数的变量。要用剪辑参数模拟布尔值 `true` 和 `false`，我们使用数字 1 和 0，而不是串 "true" 和 "false"。数字 1 和 0 可以在用于布尔语境中时转换为布尔值 `true` 和 `false`。

要给剪辑参数一个名称和可选的默认值，执行下面的步骤：

1. 双击 Parameter Name (参数名称)，然后为参数输入一个合法的标识符。
2. 双击 Parameter type (参数类型)，然后选择下面的某种类型：
  - a. Default，串或数字类型的参数默认值。
  - b. Array，参数的数组类型值。
  - c. Object，参数的对象类型值。
  - d. List，参数预先设定的串或者数字值序列。
3. 双击参数值，如果需要输入，就输入默认的值。这个值会出现在智能剪辑界面中，作为参数的初始值。默认值输入的方法取决于参数的类型：
  - a. 对于默认参数，双击参数值然后输入串或数字。
  - b. 对于 Array, Object 和 List 参数，双击参数值，在 Values 对话框中，用加、减和箭头按钮来添加、删除或者排列项目。点击 OK，接受你的设置。
4. 要添加参数功能的信息，在 Description 框中输入一个说明。
5. 要防止参数名称被智能剪辑用户所改变，选择 Lock in Instance (在实例中选择锁定)。

6. 在 Define Clip Parameters (定义剪辑参数) 对话框中，点击 OK，结束你的参数设置。

### 删除和重排序标准剪辑参数

有的时候需要删除或者重新排序智能剪辑参数。

要删除一个剪辑参数，执行下面的步骤：

1. 在 Library 中，选择 Smart Clip (智能剪辑) 进行修改。
2. 选择 Options → Define Clip Parameters (定义剪辑参数)。
3. 选择要删除的参数。
4. 点击 (-) 按钮。
5. 点击 OK。

要重新排列剪辑参数，执行下面的步骤：

1. 在 Library 中，选择 Smart Clip 进行修改。
2. 选择 Options → Define Clip Parameters。
3. 选择要移动的参数。
4. 点击箭头按钮，直到参数到了想要的位置上。
5. 点击 OK。

### 建立自定义界面的智能剪辑

要建立自定义界面的智能剪辑，我们首先创建一个正规的影片剪辑，它的行为由前面描述的一系列剪辑参数来控制。然后，创建一个独立的.swf 文件（所谓的界面.swf），它用来作为剪辑参数面板的界面。我们可以代表性地创建一个带图形界面的.swf 文件，允许用户输入参数值（通过文本框、菜单、按钮等）。这些值被自动收集，然后作为参数传递给智能剪辑。

智能剪辑通过 xch 实例（*exchange* 的简写）来和界面.swf 进行通信，xch 是界面.swf 中一种特殊的命名实例。（稍后会看到如何创建 xch 实例）图 16-4 显示了从界面.swf 传递到智能剪辑的参数名称和值。

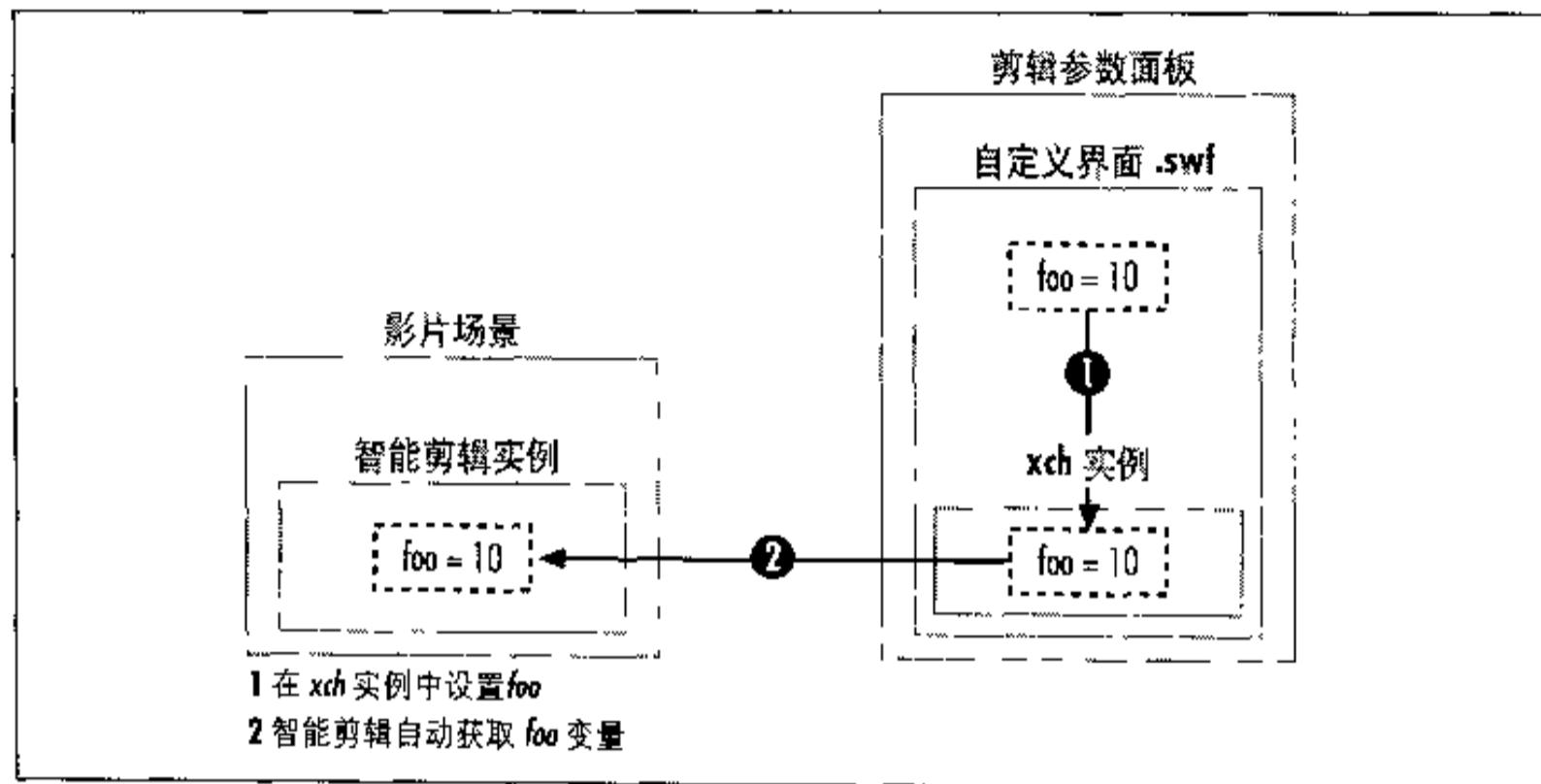


图 16-4 自定义智能界面通信

界面.swf 和智能剪辑之间的通信以循环的方式发生。当一个智能剪辑实例在场景上被选中，相应的界面.swf 就装入剪辑参数面板中。智能剪辑实例中当前的参数就被传递给了.swf 文件的 xch 实例。.swf 文件要接收这些参数，然后相应地设置界面状态。后面的在 xch 中由.swf 文件设置的变量就会自动作为参数传递给智能剪辑。当智能剪辑实例被取消选择的时候，实例.swf 就从剪辑参数面板中被删除。但是，参数值没有丢失，它们由智能剪辑保持。每次智能剪辑实例被选中，它就将参数传递回.swf 文件的 xch 剪辑。这个循环允许界面.swf 文件和智能剪辑参数保持同步。

下面的小节将说明如何创建自定义界面.swf，以及如何将它同智能剪辑联系起来。有标准用户界面的智能剪辑示例可以从在线代码库的“播放头”控制中找到。

## 创建自定义界面.swf 文件

要为用户创建一个.swf 文件作为智能剪辑的自定义界面，我们执行下面的步骤：

1. 建立一个新的 Flash 文档。
2. 创建一个新的层 *xchLayer*。
3. 选择 Insert → New Symbol，创建一个空白的影片剪辑符号。
4. 将新的符号命名为 *xchClip*。
5. 在 *xchLayer* 层中，放一个 *xchClip* 符号的实例。
6. 将实例命名为 *xch*。
7. 提供一系列的按钮、文本域和其他界面元素，用来设置变量的值。
8. 在 *xch* 实例中设置的变量会作为参数自动添加到智能剪辑中。例如，下面的代码为两个自定义参数设置值：

```
xch.param1 = value1;      // 值可以是任何数据类型  
xch.param2 = value2;
```
9. 要在自定义界面.swf 中读出已存在的智能剪辑参数的值，可以将该参数作为 *xch* 的属性。例如，下面的代码将用从智能剪辑得到的 *param1* 的值来初始化 *param1Input* 文本域：

```
param1Input = xch.param1;
```
10. 导出.swf 文件。

### 为智能剪辑添加自定义界面

既然我们已经知道如何创建界面.swf，现在我们来将它添加到智能剪辑，如下所示：

1. 关闭自定义界面.swf，然后返回最初的.flx 文件，它包含智能剪辑。
2. 选择 Library 中的智能剪辑。
3. 选择 Options → Define Clip Parameters，就会出现定义剪辑参数对话框。
4. 在 Link to Custom UI box（链接到自定义 UI 框）中，输入要用作自定义界面、并与当前.flx 文件相关的.swf 文件的位置。（也可以使用文件夹按钮来选择.swf 文件。）

## 使用智能剪辑

一旦剪辑参数被赋给一个影片剪辑，影片剪辑就正式成为了一个“智能剪辑”。智能剪辑在库中用一个特殊的图标来标识。

要在影片中使用智能剪辑实例，执行下面的步骤：

1. 从 Library 中将智能剪辑拖动到场景上。
2. 选择 Window → Panels → Clip parameters。
3. 如果剪辑有一个标准界面，如下设置每个参数的值：
  - a. 对于默认参数，双击参数值，然后输入串或者数字。
  - b. 对于数组参数，双击参数值。在 Values (值) 对话框中，双击每一个元素值，然后输入串或者数字。点击 OK，接受数组元素值。
  - c. 对于对象参数，双击参数值。在 Values 对话框中，双击每一个对象属性值，然后输入一个串或者数字。点击 OK，接受你的对象属性。
  - d. 对于 List 参数，双击参数值，然后选择一个选项。
4. 如果剪辑有一个自定义界面，使用自定义界面中提供的工具来设置剪辑的参数。

## 小结

在下一章中，我们要将 ActionScript 同简单的服务器应用程序结合起来，以产生 Flash 表单。

## 第十七章

# Flash 表单

Web 中的交互性或许在填充表单中达到了极致。这听起来也许有点夸张，但也许不是。表单在界面上的出现显得非常一般，但是它们却是在线社区（聊天室和留言板）、数据风格（人格化），当然还有如潮的电子商务（在线买卖）的心脏。

Flash 4 是第一个支持表单的 Flash 版本。从那以后，Flash 即获得了同服务器进行通信的强大手段。这一章将要讨论 Flash 中表单的使用基础：从获取用户输入和在 Flash 中显示输出，到向服务器发送数据以及从服务器接收数据。像 XML 处理这些更高级的主题，将在第三部分中介绍。

## Flash 表单数据循环

在我们进行详细的讨论之前，先来粗略看看表单提交过程的典型步骤：

1. Flash 接收用户数据输入。
2. Flash 为提交给 Web 服务器的数据作准备（收集和确认变量）。
3. Flash 通过 HTTP（或者说 HTTPS）将数据发送给 Web 服务器。
4. Web 服务器接收数据，将它传递给服务器端数据处理应用程序（如 Perl 脚本，PHP 脚本，Cold Fusion，或者 ASP）。

5. 数据处理应用程序分析并处理所提交的数据。
6. 数据处理应用程序将结果传递给 Web 服务器，服务器将结果又发送给 Flash。
7. Flash 存储或者有选择地显示结果。

因此，一个活动 Flash 表单要求：

- 一个前端（用户所看到的）。
- 一些用来将表单内容提交给服务器端脚本或应用程序的 Flash 脚本。
- 一个服务器端脚本或者应用程序。
- 一些 Flash 脚本，负责处理从服务器返回的数据。

我们来深入看一看这些内容。

## Flash 客户端用户输入

要提供输入，用户通常在文本域中输入文本，然后点击表单的提交按钮。只有文本域才能作为预制表单成分提供。我们还要手动建立其他所有的东西，包括提交按钮（我们会在后边看到）。参见第十八章，可以得到有关文本域方面的信息。

因为 Flash 有一个完整的脚本语言，因此可以创建智能表单，它会在将数据提交给服务器之前进行一定的预处理。我们要先确认用户的输入，然后才将它提交给 Web 服务器，以确保数据处理应用程序总是接收到有效的数据。普通的确认包括检查所有要求的域是否都被填充，并检测输入的数据类型是否正确。例如，一个 e-mail 地址应该包括一个名称，后面跟着一个 @ 符号，再后面是域名。

## 将数据发送给服务器端进行处理

一旦确认了数据，就可以安全地将它传递给 Web 服务器。ActionScript 提供了许多工具来将表单数据传送给 Web 服务器：

- *loadVariables()* 函数，在本章后面描述。
- *XML* 类的 *send()* 和 *sendAndLoad()* 方法，在第三部分描述。

- 全局函数 `getURL()`, 在第三部分描述。

Web服务器将Flash数据传送给服务器端应用程序, 应用程序会对数据进行处理, 比如一个中间设备数据库(例如, Allaire 的 ColdFusion 或者 Microsoft 的 ASP)或者一个CGI脚本(如Perl脚本、PHP脚本或者Java servlet)。

在描述Web客户/服务器数据循环的时候, 要区分Web服务器和数据处理应用程序。通常, 这个区别是不明显的——客户端必须提出一个HTTP请求, 以将数据发送到数据处理应用程序, 因此, Web服务器被包括在其中是很自然的。但是, 在Flash表单的制作中, 必须留心Web服务器和数据处理应用程序之间的移交管理。数据从Flash传递到服务器, 或者到达URL的目标端(使用GET), 或者在一个变量名称和值的序列中(使用POST)。当Web服务器遇到错误的时候, Flash不显示服务器发送的HTTP错误信息(和浏览器不同)。例如, 如果Web服务器没有找到一个CGI脚本, 它就发送一个“404 Not Found”消息, 但是Flash并不显示这个消息。类似的, 如果一个CGI脚本的许可没有得到正确设置, 我们就看不到任何执行错误的消息。为了在使用Flash的时候隔离客户端/服务器问题, 在要运行脚本的时候, 应监控Web服务器的HTTP错误日志。你会发现, Web服务器会试图告诉你一些Flash没有表示的东西。

## 数据处理应用程序

在接收一个数据体的时候, 数据处理应用程序必须分析它(也就是说, 对它作一些说明, 并且如果需要的话, 将它分离为更容易管理的片段)。在分析数据之后, 服务器应用程序可以有无数的方法来操作它。通常, 数据处理包括将内容存储在数据库或者文本文件中, 以便以后获取。

一旦数据处理完成, 数据处理应用程序就产生一个结果, 传送给Flash。结果可以是任何东西, 从简单确认消息(“谢谢你提交信息”)到数据库或者当前产品价格的记录列表。

应用程序将结果传送给Web服务器, 服务器会将这些结果发送回Flash, 以便存储或者显示。

注意：数据处理应用程序开发者会注意到，他们的应用程序必须将结果的 MIME 类型设置为 application/X-www-form-urlencoded。如果 MIME 类型被丢失，结果到达 Flash 的时候就很可能无效的。

---

## Flash 接收和解释结果

我们正在接近 Flash 表单循环的末尾，但是我们毕竟还没有完成。首先，我们必须确定 Flash 在服务器端应用程序处理数据和传送结果的时候正耐心等待着。考虑一个查询股价的影片。用户输入一个股票行情符号，然后点击 Get Stock Price（获取股票价格）按钮。在显示价格之前，股票检索应用程序必须鉴定并返回价格。在影片等待的时候，它显示一个 Loading 消息。当得到价格后，影片又开始动作。

在响应 *loadVariables()* 调用的时候由 Flash 获取的数据存储在一个指定的目标剪辑或层级中。一旦数据被获取，Flash 表单循环就完成了，我们就可以随心所欲地处置宝贵的数据传输内容。现在，我们将知识应用于实践，创建一个简单的填充表单。

## 创建 Flash 填充表单

我们的例子包括了 Flash 表单的所有必要成分，在前边已经说过了，但是都是在最简单的层次上。这个指南将演示如何将单一的文本域变量从 Flash 发送到名为 *echo.pl* 的 Perl 脚本，以及如何在 Flash 中接收 *echo.pl* 发回来的响应。这个示例文件的版本可以在在线代码库中找到。现在我们就来看看吧。

### 建立前端

和 HTML 不同，Flash 没有一个创建表单的完整的机制。在 HTML 中，创建下拉菜单和单选钮只需要使用 <SELECT>、<OPTION> 和 <INPUT TYPE="RADIO"> 标签就可以了。在 Flash 中，这些设备必须手动创建。Flash 惟一的内置表单设备是用户输入文本域（与 HTML 中的 <INPUT TYPE="TEXT"> 或 <INPUT TYPE="TEXTAREA"> 等价）。

---

注意：虽然表单小部件不能直接在 Flash 制作工具中建立，但是单选钮、检验栏和下拉菜单可以作为该产品包含的智能剪辑来使用。要访问表单部件智能剪辑，选择 Window → Common Libraries → Smart Clips。

---

在我们的表单中，会有一个用户输入的文本域和一个提交按钮。我们会将这两个元素放到影片剪辑中，这样就可以容易地识别要发送给服务器的变量。首先，我们要创建一个新的文档和 formClip 影片剪辑，如下所示：

1. 建立一个新的 Flash 文档。
2. 选择 Insert → New Symbol，Symbol Properties (符号属性) 对话框就会出现。
3. 在 Name 框里，输入 **formClip**。
4. 点击 OK。
5. 从 Library 中拖一个 formClip 的实例到主场景。

下面，我们将一个用户输入文本域添加到 formClip，执行下面的步骤：

1. 在 Library 中，双击 formClip 符号，以编辑这个剪辑。
2. 选择 Text (文字) 工具。
3. 在剪辑画布上，拖动矩形框，直到它的大小可以让用户输入一行文本。
4. 选择 Window → Panels → Text Options。
5. 对文本类型，选择 Input Text。
6. 对于行显示，选择 Single Line。
7. 在 Variable 框中，输入 **input**。
8. 选择 Border (边框) /Bg (背景)。

现在，我们要对 input 文本域赋一个默认的值，确保在用户不输入任何数据的时候它会被发送给服务器：

1. 在 formClip 时间线上选择第 1 帧。
2. 在动作面板中输入下面的代码：**input = " ";**

最后，我们将一个提交按钮添加到 formClip，执行下面的步骤：

1. 选择 Window → Common Libraries → Buttons。

2. 从 *Buttons.fla* 库中，拖动 *Push Bar* 按钮的一个实例到 *formClip* 场景上。

## 将数据发送到服务器

有很多方法可以将数据从 Flash 发送到服务器应用程序，包括 *loadVariables()*, *getURL()*, *loadMovie()*, *XML.load()*, *XML.sendAndLoad()* 和 *XMLSocket.send()* 方法。在我们的例子中使用 *loadVariables()* 方法。对于其他方法的有关信息，请参见第三部分。

在前面的小节中，我们将一个用户输入文本域和一个提交按钮放置到影片剪辑 *formClip*。要让提交按钮被点击的时候发送 *formClip* 的变量到 *echo.pl*，我们执行下面的步骤：

1. 在 Library 中，双击 *formClip* 符号（对符号进行编辑）。
2. 选择 *Push Bar* 按钮实例。
3. 选择 Window → Actions。
4. 将下面的代码输入到动作面板中：

```
on (release, keyPress "<Enter>") {
    loadVariables ("http://www.yourserver.com/cgi-bin/echo.pl",
                  '_root.response',
                  "GET");

    _root.response.gotoAndStop("loading");
}
```

提交按钮上的代码用 *loadVariables()* 来将 *formClip* 的变量发送到 *echo.pl*，它会让 *response* 剪辑显示一个装载信息。我们会在稍后建立 *response* 剪辑。现在，我们来查看 *loadVariables()* 调用是如何工作的。

*loadVariables()* 调用的第一个参数要指定 *echo.pl* 在服务器上的位置（服务器端的脚本通常存储在 *cgi-bin* 文件夹中）。确定要按照服务器的域名和目录结构正确地设置位置。当 *loadVariables()* 执行的时候，*formClip* 中所有的变量都被发送到这个位置。

*loadVariables()* 调用的第二个参数表示到剪辑的路径，这个路径要用来存储 *echo.pl* 发送的返回值，也就是 “*\_root.response*”。

*loadVariables()*调用的第三个参数指定提交 formClip 变量到服务器的 HTTP 方法——在本例中也就是 GET 方法。ActionScript 支持 GET 和 POST 操作，如第三部分 *loadVariables()* 中所描述的那样。

## Perl 脚本，echo.pl

当用户点击影片中 formClip 里的提交按钮时，Flash 发出一个 HTTP 的 GET 请求。这个请求执行 Perl 脚本 *echo.pl*。为了让表单有效，*echo.pl* 必须被放在 Web 服务器的 CGI 有效目录中，由服务器管理员建成，如下所示：

- 脚本必须是可以执行的（通常意味着将文件许可设置为 755）。
- 在 UNIX 上，到 Perl 解释程序的路径必须在脚本里设置。

例 17-1 显示了 *echo.pl* 的源代码。注意，# 符号表示 Perl 中的注释。

### 例 17-1：echo.pl 的源文件

```
#! /usr/local/bin/perl
#-----
# 名称: 简单的 Flash 反应
# 版本: 1.2.0
# 作者: Derek Clayton  derick_clayton@iceinc.com
# 描述: 将从 Flash 的 GET 或 POST 所接收到的名称 / 值对反射回来
#-----
# 主程序
#-----
use CGI;                                # 用 CGI.pm 作简单的分析
$query=new CGI;                          # 查询对象
$echoString="output=";                   # 对输出串进行初始化
getInput;                                 # 得到从 Flash 接收到的输入
writeResponse;                           # 编写对 Flash 的信息返回
exit;                                    # 退出脚本
#-----
sub getInput {
    # 对每一个 key 获取相关的值，然后添加到 echo 串
    foreach $key ($query->param) {
        $value = $query->param($key);
        $echoString .= "$key:$value\n";
    }
    # 在编写回应之前删除后缀的换行符号 (\n)
    chomp($echoString);
}

sub writeResponse {
```

```
# 为Flash设置目录类型  
print "Content-type:application/x-www-form-urlencoded\n\n";  
# Write the output  
print $echoString;  
1
```

*echo.pl* 脚本通常执行以下三种任务：

- 接收从 Flash 发送的数据，将数据解析为一系列的变量名称和值。
- 将这些变量名和值装配到串中，返回 Flash。这个串的格式为：  
`output=name1:value1\n.name2:value2\n...namenvaluen`
- 将串返回 Flash。

接收从 *echo.pl* 返回的串之后，Flash 自动将它解释为一个 URL 编码的变量序列（在第三部分中介绍）。因此，*output* 变成了 *response* 剪辑时间线上的一个变量。通过在 Flash 影片中检查 *output* 的值，可以知道最初是哪些变量名和值被发送给 *echo.pl*。

显然，*echo.pl* 不是世界上最有趣的 Web 应用程序，它只是一个概念的验证。但是在应用的时候，概念可以产生有趣的和有效的结果。“服务器通信”下在线代码库中的平面数据库示例中有 Perl 系统开发完整的示例。

## 从服务器接收结果

回忆一下，当我们把 *formClip* 的变量发送到 *echo.pl* 的时候，我们要求 *echo.pl* 的返回值要被存储在影片剪辑 *response* 中：

```
loadVariables ("http://www.yourserver.com/cgi-bin/echo.pl",  
              "_root.response",  
              "GET");
```

我们现在要建立 *response* 剪辑；它有三个状态： *idle*（闲置），*loading*（装载）和 *done loading*（装载完毕）。在闲置状态中，*response* 对用户是不可见的，它在等待数据装载。在装载状态中，*response* 告诉用户，数据已经提交给服务器，Flash 正在等待答复。在装载完毕状态下，*response* 已经接收到服务器的答复，通过文本域对用户显示结果。*response* 的这三种状态控制着它的时间线结构。每个状态是用一个标签关键帧 *idle*，*loading* 和 *doneLoading* 来表示的。帧的显示规定如下：

- 当影片装载的时候，response 显示 idle 帧。
- 当变量被提交的时候，提交按钮发送 response 给 loading 帧。
- 当变量被接收到的时候，response 的 *data* 事件处理器（我们将要创建）发送 response 给 doneLoading 帧。

要建立 response，我们执行下面的步骤：

1. 选择 Insert → New Symbol。Symbol Properties 对话框就会出现。
2. 在 Name 框中，输入 responseClip。
3. 点击 OK。
4. 从 Library 中拖动 responseClip 的一个实例到主场景上。
5. 将 responseClip 的实例命名为 response。
6. 在 Library 中，双击 responseClip 符号（对符号进行编辑）。
7. 创建四个命名的时间线层，从上到下依次为：scripts, labels, loading 和 outputField。
8. 在每层中创建三个关键帧。
9. 在 labels 层上，对第 1,2 和 3 帧分别添加标签 idle, loading 和 doneLoading。
10. 在 scripts 层上的第 1 帧添加下面的代码：stop();
11. 在 loading 层的第 2 帧中添加静态文本，“loading, please wait”。
12. 选择 Text 工具。
13. 在 outputField 层的第 3 帧中，拖出一个文本框。
14. 选择 Window → Panels → Text Options。
15. 对于文字类型，选择 Dynamic Text。
16. 对于行显示，选择 Multiline。
17. 在 Variable 框中，输入 outputField。

现在，我们来添加*data*事件处理器，它会在服务器将数据发送到Flash之后被触发。执行下面的步骤：

1. 在主场景中，选择 response 实例。
2. 选择 Window → Actions。
3. 在动作面板中输入下面的代码：

```
onClipEvent (data) {  
    this.gotoAndStop('doneLoading');  
    outputField = output;  
}
```

当 Flash 接收到 *echo.pl* 发送的内容之后，response 的 *data* 事件处理器就自动执行。在 *data* 处理器中，我们将 response 的播放头移动到 *doneLoading* 帧，然后在 *outputField* 中显示 *output* 的值（由 *echo.pl* 提供）。使用 *data* 事件处理器以确保 *output* 变量总是在我们试图将它的值显示在 *outputField* 文本域中之前装载完毕。

现在剩余的所有工作就是测试表单了！导出影片，在 *input* 文本域中输入一些文本，然后点击提交按钮。如果你的表单开始的时候不能运行，确认一下你的服务器脚本设置是否正确。还要记住，可以在在线代码库中学习粘贴在那里的实用版本。

读者练习：试着添加一个重启按钮给你的表单，用它来清除输入域中的值。在 *formClip* 中创建一个新的按钮，然后将下面的代码添加到上面：

```
on (release) {  
    input = '';  
}
```

在你的表单中试着添加更多的输入域，Perl 脚本会如实地返回你发送给它的同样数量的变量。你可以将它们分离开，单独显示每一个变量的值吗？

## 小结

好的，我们可以说本章的内容并不是太拙劣。考虑一下你能用 Flash 表单来完成的所有这些工作，比如记载用户的参数选择，提供一个讨论会，或者为目录动态地载

入产品信息。在线代码库中的普通文件 Perl 数据库可以带领你深入到这种类型的应用程序的创建中。在下一章，我们要学习复杂的文本域和它们在 Flash 表单中的应用。

# 第十八章

## 屏幕文本域

因为Flash基本上是一个视觉环境，所以影片通常表示要呈现给用户的屏幕信息。类似的，因为 Flash 是一个交互环境，所以影片通常通过 GUI（图形用户界面）而获取用户的信息输入。要在屏幕上显示变量的值，或者让用户将数据输入到 Flash 影片中，我们使用文本域（text fields）。

文本域提供了设置和获取那些有视觉表现的变量值的途径。文本域有两种类型——动态文本域，用来向用户显示信息，以及用户输入文本域，用来获取用户的输入信息。

### 动态文本域

动态文本域就像一个变量视口一样——它显示指定变量的文本串值。动态文本域是在 Flash 中用文字工具创建的。但是，和正规的静态文本不同，动态文本域的内容和某个变量相联系，并且可以通过 ActionScript 来修改或获取。

通过获取一个文本域值，我们就可以在脚本中获得屏幕上的用户信息。通过设置文本域的值，我们又可以让结果值显示在屏幕上。

## 创建动态文本域

要建立一个新的动态文本域，执行下面的步骤：

1. 选择 Text 工具。
2. 在场景上点击并拖动出一个矩形框。你所创建的轮廓可以定义新文本域的大小。
3. 选择 Text → Options。Text Options 对话框出现。
4. 在 Text Type 菜单中，选择 Dynamic Text（动态文本）（其他选项包括 Static Text（静态文本）和 Input Text（输入文本））。
5. 在 Variable 下面，输入动态文本域的名称，遵循我们在第十二章中所学习的规则，以构成合法的变量名称。

在创建一个动态文本域之后，你通常要设置新的域选项，我们将在后面描述。

## 改变动态文本域的内容

一旦文本域创立，我们可以用它来在屏幕上显示一个值。例如，如果我们创建了一个动态文本域 myText，就可以用下面的语句来设置这个文本域的内容：

```
myText = 10;                      // 在文本域myText中显示一个数字  
myText = "Welcome to my web site"; // 显示一个串  
  
var msg = 'Please make a selection';  
myText = msg;                      // 在myText中显示msg的值
```

只要变量 myText 的值改变，myText 动态文本域的内容也就会被更新。但是，在一个值被传送到动态文本域进行显示之前，首先要转换为一个串。因此，实际的显示内容是由表 3-2 中所描述的串转换规则所控制的。

和普通变量一样，文本域和它们所在的影片剪辑时间线相联系。要访问一个远程影片剪辑时间线上的动态文本域，我们使用第二章中所描述的技巧。

## 获取动态文本域的值

我们可以简单地使用动态文本域的名称来获取它的值。例如，如果 myTextField 是影片中的一个动态文本域，我们可以获取它的值，或者将它的值赋给另外的变量，如下所示：

```
welcomeMessage = myTextField;
```

文本域的赋值和获取通常都合并在一个语句中。可以使用 += 操作符来为文本域当前的内容添加文字：

```
// 设置文本域的值  
myTextField = "Today's Headlines...";  
// 创建一个新的消息  
var newText = "Update!The Party Has Been Cancelled!"  
// 对已经存在的文本域内容添加新的消息  
myTextField += newText;
```

## 用户输入文本域

用户输入文本域和动态文本域的不同之处仅在于前者可以在影片播放期间被用户编辑。也就是说，用户可以在用户输入文本域中输入文字，改变它的值。ActionScript 然后就可以获取和操作用户输入的值。用户输入文本域在客户簿、定货表单、密码输入域或者其他需要用户信息的地方非常有用。

### 创建一个用户输入文本域

要创建一个用户输入文本域，可使用“创建动态文本域”中所描述的步骤，但是，在文本类型菜单中要选择“输入文本”，而不是“动态文本”。

### 改变输入文本域的内容

和动态文本域类似，用户输入文本域可以在任何时间改变，只要用一个赋值语句设置命名文本域的值就可以了：

```
myInputText = 'Type your name here';
```

因为用户输入文本域通常用来接收数据而不是显示数据，我们一般不设置它的内容，除了要给用户的输入提供一个默认值以外。

## 获取和使用输入文本域的值

你可以简单地在脚本里用文本域的名称来指向它，从而获取其中的值。例如，要显示一个输入文本域 myInput 的值，我们用：

```
trace(myInput);
```

因为用户输入到用户输入文本域中的数据通常是串类型的，我们可以在非串环境中使用它之前对其进行适当的转换。比如，在下面的计算例子中，我们计算两个用户输入文本域的总和：

```
// 假设用户将myFirstInput 设置为 5，将mySecondInput 设置为 10  
// 然后我们计算域的总和  
// 错误：将两个域 相加 '会将myOutput 设置为 "Total:510"  
// 因为 + 操作符被当作了串连接符  
myOutput = Total: " + (myFirstInput+mySecondInput);  
// 正确：首先将域中的值转化为数字，然后才能得到正确的结果  
myOutput = 'Total: " + (parseFloat(myFirstInput) + parseFloat(mySecondInput));
```

## 用户输入文本域和表单

用户输入文本域通常用来填充要提交给服务器端应用程序（比如Perl脚本）的表单。当变量通过 *loadVariables()* 被提交给服务器时，只有在当前影片剪辑中定义的变量才被发送。但是，当一个表单包含用户输入域的时候，这个域应该被存储在一个单独的影片剪辑中，这样，它们就能作为一个组很容易地发送。参见第十七章可获得 *loadVariables()* 的相关细节内容。

## 文本域选项

动态文本域和用户输入文本域共同使用大部分（但不是全部）选项，这些选项用来设定它们的显示和输入特征。图 18-1 给出了用户输入和动态文本域的文本选项面板。

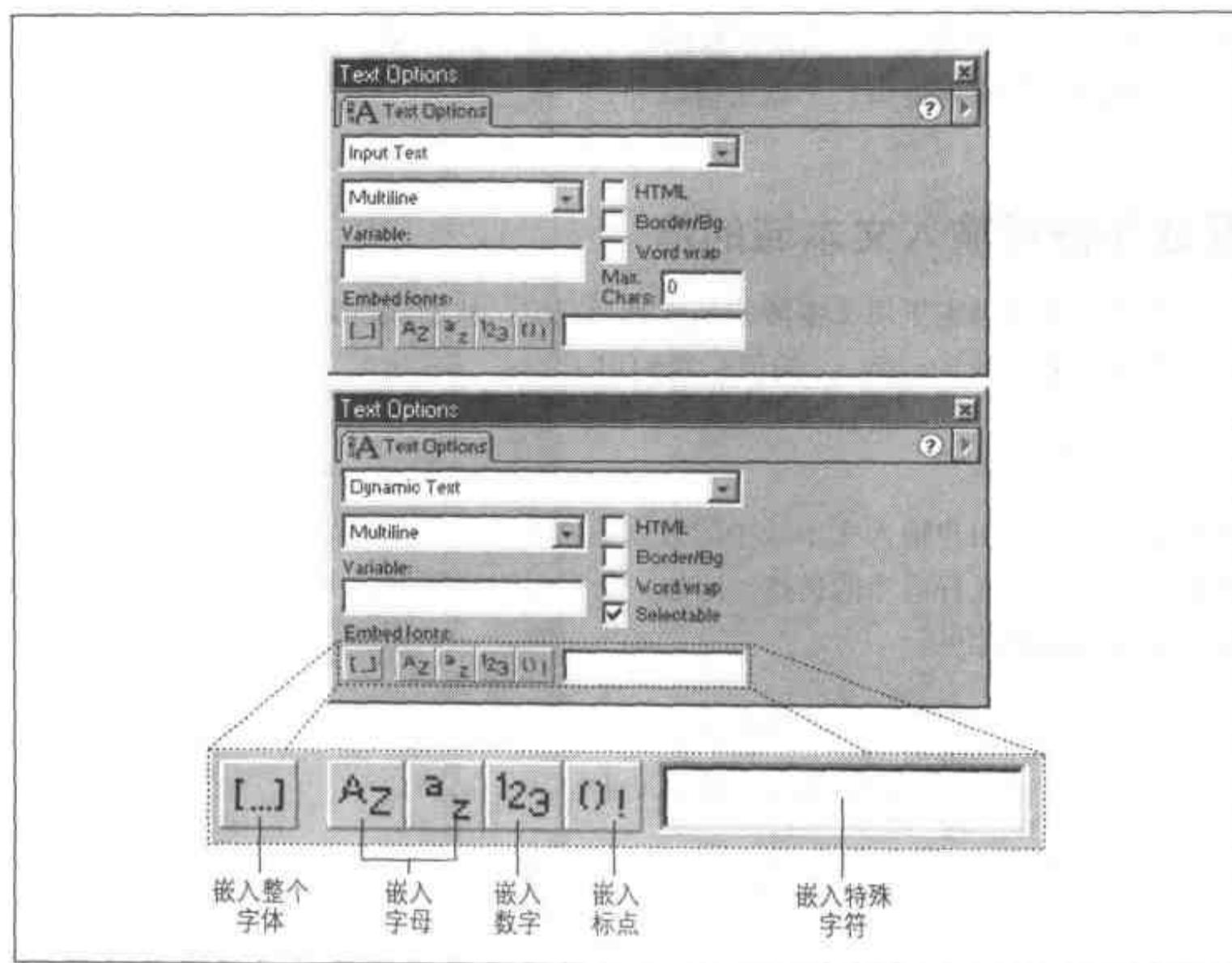


图 18-1 文本选项面板

## 行显示

要设置文本域的格式和输入类型，或者隐藏用户的输入，我们使用 LineDisplay (行显示) 菜单。其中有三个行显示选项：

### 单行

单行选项防止用户在域中输入多于一行的文本，在文本输入期间不必使用回车键。

单行选项设置对在制作工具中没有分行输入的文本也起作用。在制作期间自动进行“软件换行”的文本不会在播放器中被分行，而是将文本显示在一行中，即使它超出了右边边界。但是，在制作的时候使用回车对于单行设置是无效的，带回车的文本会显示在播放器中，就如同它出现在制作工具中一样。

单行选项主要应用于用户输入文本域。当使用在动态文本域上时，它和多行动态文本域差不多，除了“自动换行”选项不被选中之外。

\n转义序列和newline关键字在插入到行中的时候，会造成分行的结果，不管是不是设置了单行选项。例如，如果我们将一个文本域变量设置为值“this is\na test”，文本“this is”和“a test”就会显示在不同的行中。

### 多行 (*Multiline*)

多行设置允许用户在文本域中输入超过一行的文本。选择了多行选项之后，用户输入时就可以使用回车。

多行设置对动态文本域的输出基本没有影响，除非和“自动换行”选项一起使用。如果没有选择自动换行，多行文本域的效果和单行文本域差不多。

### 密码 (*Password*)

密码选项用来隐藏输入到表单中的字符，它只应用于用户输入文本域。除了所有的字符（包含空格在内）都用星号（\*）来隐藏以外，它的效果和单行文本域类似。例如，词“hi there”会被显示为“\*\*\*\*\*”。

由于Flash界面里的突发情况而有可能将密码文本域中的词隐藏起来。如果将行显示设置为多行，并且选择了自动换行，然后再将行显示设置为密码，自动换行将会保留选择。但是，我们不提倡使用多行密码项目，因为它会让大部分用户混淆。

## 变量 (*Variable*)

文本选项 (Text Options) 面板中的变量选项 (Variable Options) 用来命名一个动态的或者用户输入文本域。文本域必须命名，以便用ActionScript来操作。在命名文本域的时候，遵循第二章以及第十四章中所描述的合法变量名称构造规则。

## 边框 / 背景 (Border/Bg)

在文本选项面板中进行设置的时候，边框/背景选项会让文本域边缘显示一个黑框，并在域中可视区域后面放置一个白色的背景。这些颜色和类型不可以自定义。要产生一个自定义的文本域背景，就得清除边框/背景选项，然后手动地在文本域后面绘制一个形状。

## 自动换行 (Word Wrap)

在行显示选项和多行设置结合使用的时候，自动换行可以对文本进行换行，以免超出域的宽度。这个设置对 ActionScript 的用户输入文本和文本显示都有效。

如果你在选择多行选项的时候设置了自动换行功能，然后选择了单行选项，那么自动换行功能会保持被选择状态。如果你不想让文本换行，就要注意取消自动换行选择。

## 可选 (Selectable)

动态文本域中的文本仅当域的“可选”项被设置的时候才可以被用户选中。尽管如此，动态文本可以被复制，但是不能被剪切或编辑。用户输入文本域总是可选的，它们的文本总可以复制、剪切或者编辑。

---

注意：文本必须通过 Flash 中的 Windows 右键菜单（或者是 Macintosh 机器上的 Ctrl-click）被复制、剪切和粘贴。像 Ctrl-C 和 Ctrl-V (Windows) 或者 Cmd-C 和 Cmd-V (Macintosh) 这样的键盘快捷键将被忽略。

---

## 最多字符 (Max Characters)

“最多字符”选项只用于用户输入文本域，它限制用户可以输入到文本域中的文本数量。默认情况下，最多字符被设置为 0，它允许输入的字符无限多。其他设置只允许输入有限的字符数。

最多字符通常用在要求特定数据格式的表单中。例如，可以使用它来限制输入的日期和月份为两位数，输入的年份为四位数。

## 嵌入字体 (Embed Fonts)

在默认情况下，所有的动态和输入文本域都使用设备字体（安装在用户系统中的字体）。使用这种字体时，如果用户在文本域字符面板中有特定字体，那么文本出现在用户系统中就和它出现在制作期间是一样的（但是没有保真）。如果用户没有这种字体，就会使用选择字体，它不一定符合你的需要。

要确保文字能被渲染为特殊的字体，可以使用“嵌入字体”选项，将某种字体嵌入到影片中，参见图 18-1 中的放大示意。

我们可以：

- 使用 [...] 按钮嵌入整个字体。
- 使用 AZ, az, 123 和 ()! 按钮嵌入字母、数字或者标点符号的任意组合。
- 将它们输入到所提供的域中，以嵌入特殊的字符。

嵌入完整的 Roman 字体通常会造成影片增加 20-30KB (Asian 字体可能会更大)。如果只使用字体的一个子集，就能通过只嵌入需要的字符而节省文件空间。不嵌入的字符不能通过 ActionScript 被用户输入或者显示。我们可以使用这个技术来将输入文本限制为特定的字符。

必须为使用特殊字体的每一个文本域单独设置“嵌入字体”(Embed Fonts) 选项，甚至多个文本域使用同一字体也是如此。但是，文件大小在多个文本域嵌入同一字体的时候并不会受到影响——只有一个字体复本和影片一起下载。要一次为多个文本域提供相同的嵌入字体，可先选择所要的各域，然后像平常那样设置嵌入字体选项就可以了。

以嵌入字体显示在文本域中的文字通常都使用保真。因此，不推荐使用尺寸小于 10 点的嵌入字体，因为保真文字在大部分字体中如果小于 10 点，就不容易看清。要取消字体保真，可以使用设备字体(也就是系统字体)，只要不选择所有的嵌入字体选项即可。设备字体从来不使用保真。

---

**警告：**被旋转或者遮盖的文本域的内容不会显示在屏幕上，除非它的字体是嵌入的。也就是说，不能旋转或遮盖使用设备字体的文本域。

---

本章后面会介绍更多关于文本域字体的细节内容。

## 文本域属性

当文本域中的文字体跨越数行，超过了域中的物理可视区域，超出部分的文字行就

会被隐藏。但是，超出部分的行仍然是文本域的一部分。要浏览这些行，可以在域中点击，然后按下箭头按键，直到那些行的文字出现。显然，我们不能指望用户使用箭头按键来滚动文本域中的文本。我们应该用 scroll 和 maxscroll 属性来提供控制文本滚动的按钮，这两个属性都使用一个索引数字来指向文本域中的行。第一行为号码 1，行号在文本域中逐行增加，包括那些看不见的部分。图 18-2 是一个文本域行索引值的示意。

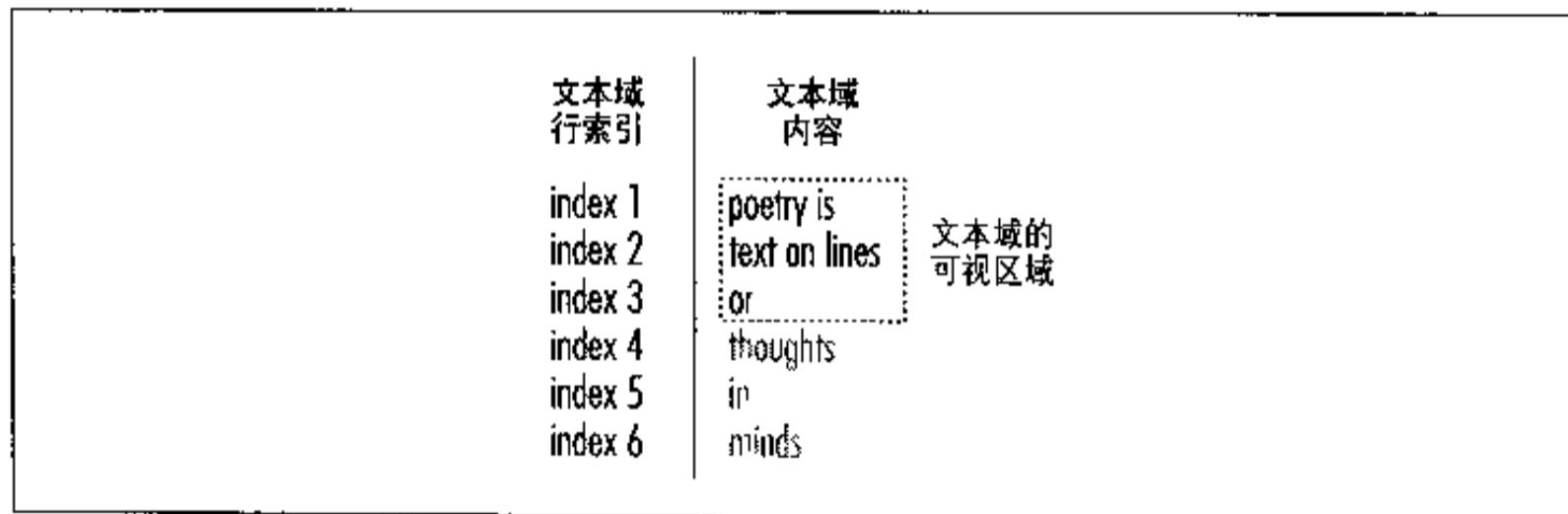


图 18-2 文本域行索引

## scroll 属性

scroll 属性表示当前显示在文本域中的最顶端的行数，可以用 `textFieldName.scroll` 来访问。

当一个文本域包含的行数超过它一次所能显示的范围时，可以通过设置 scroll 属性来改变可以显示在域中可视区域内的行。例如，如果我们要将图 18-2 中显示的文本域的 scroll 属性设置为 3，那么文本域就会显示：

```
or  
thoughts  
in
```

## maxscroll 属性

maxscroll 属性告诉我们一个域可以被滚动多少距离（也就是说，在最后一行出现之前它可以滚动多少）。它总是等于域中最后一行的行索引减去可视区域中一次所能

显示的行数，再加上1。例如，图18-2中文本域的maxscroll属性是4（最后一行为6，减去可视区域中的3行，再加1）。注意，maxscroll不等于文本行数。

我们可以用 `textFieldName.maxscroll` 来获取（但不是设置）maxscroll属性。

## 典型的文本滚动代码

`scroll`和`maxscroll`属性结合使用的时候可以用来滚动一个文本域。下面的代码在每次点击按钮的时候会向下滚动一行：

```
on (press) {  
    if (textField.scroll < textField.maxscroll) {  
        textField.scroll++;  
    }  
}
```

下面的代码在每次点击的时候向上滚动一行：

```
on (press) {  
    if (textField.scroll > 1) {  
        textField.scroll--;  
    }  
}
```

使用在影片中的滚动按钮的示例可以从在线代码库中下载。

Flash 5 播放器的Build 30，放弃了Flash 5的制作工具，它有一个文本域显示错误。当保真文本域被滚动的时候，并不能总是消除先前文本的痕迹。要解决这个问题，可以在文本域周围放置一个框，以掩盖残留的痕迹。这个错误在2000年12月发布的Flash 5播放器的build 41中得到了修正。可以使用全局函数 `getVersion()` 函数来查看播放器的这个版本。

## `_changed`事件

在Flash 4和Flash 5中，对用户输入文本域内容的修改可以用非正式的`_changed`事件探测到。`_changed`事件在用户从输入文本域中添加或者删除文本的时候会触发一个特殊命名的Flash 4类型的子程序。要为文本域创建`_changed`事件，执行下面的步骤：

1. 在任何时间线上创建一个输入文本域。
2. 将该文本域命名为 myField。
3. 在文本域的同一个时间线上，标注一个帧 myField\_changed。
4. 在 myField\_changed 帧中添加任何代码。例如：

```
trace("myField was changed");
```
5. 用 control → Test Movie 来导出该影片。
6. 在 myField 文本域中输入字符，帧 myField\_changed 上的代码就会被执行，输出窗口中将出现 “myField was changed”。

当然，名称 myField 是任意的，可以使用你所喜欢的任何文本域名称，只要对应的帧标签设置为相同的名称就可以了。注意，用 ActionScript 设置文本域的值不会触发域的 *\_changed* 事件。只有用户的击键动作可以触发 *\_changed* 事件。

*\_changed* 事件是一个非正式的功能。在 Flash 以后的版本中，会产生新的、更标准的文本域事件处理器方法。

## HTML 支持

字符面板让我们可以设置一个文本域的字体大小、字体和字体类型，但是，它只设置文本域整个的属性。要以字符为基础来设置属性，以及添加超文本链接，可以使用 HTML（它是作为 Flash 5 的文本域功能被添加的）。

虽然 HTML 可以用于动态文本域和用户输入文本域，我们通常只用 HTML 文本域来进行显示。要为文本域添加 HTML 支持，可以在文本选项面板中选择 HTML 选项。

文本域支持的 HTML 标签设置限制为：<B>，<I>，<U>，<FONT>，<P>，<BR> 和 <A>。

### <B>（粗体）

假如存在加粗字体，<B> 标签将文字显示为粗体：

```
<B>This is bold text</B>
```

## <I> (斜体)

假如存在斜体，<I> 标签将字体显示为斜体：

```
<I>This is italic text</I>
```

## <U> (下划线)

<U> 标签在文字下面加下划线。例如：

```
<U>This is underlined text</U>
```

因为链接文字在 Flash 中没有下划线，可以用 <U> 标签来标注链接：

```
<A HREF="http:// www.thesquarerootof-1.com"><U>click here</U>
</A> to visit a neat site.
```

## <FONT> (字体控制)

<FONT> 标签支持下面的三种属性：

### FACE

FACE 属性表示使用的字体名称。注意，在 Flash 中不像在 HTML 中那样支持多种字体的列表。Flash 只渲染列在 FACE 属性中的第一种字体。例如，在代码 <FONT FACE="Arial, Helvetica">my text</FONT> 中，如果 Arial 被遗漏，Flash 不会将 my text 显示为 Helvetica 字体。文字将被显示为默认的字体。

### SIZE

SIZE 属性将标注的文字表示为固定的点尺寸（比如 <FONT SIZE="18"> 或者相对尺寸。相对点尺寸前边的 + 或 - 用来表示和字符面板中的文字大小的关系。例如，如果字符面板中的点尺寸为 14，那么 <FONT SIZE="-2"> 将会以 12 点的大小来显示标注文本。

### COLOR

COLOR 属性表示标注文本的颜色，是一个十六进制数，以 # 符号开头。例如：<FONT COLOR="#FF0000">this is red text</FONT>。十六进制数字是由三个两位数字的 RGB 值构成的，两位数字从 00 到 FF。注意，Flash 对 COLOR

属性的实现比 HTML 更严格——要求有#符号，并且诸如 green 和 blue 这样的颜色名称不能用作 COLOR 值。

下面是一些 <FONT> 的例子：

```
<FONT FACE="Arial">this is Arial</FONT>
<FONT FACE="Arial" SIZE="12">this is 12pt Arial</FONT>
<FONT FACE="Lucida Console" SIZE="+4" COLOR="#FF0000">this is red,
+4pt Lucida Console</FONT>
```

本章后面介绍了有关 Flash 字体的更多细节内容。

## <P> (分段)

<P> 标签为段落划分界限，但是在 Flash 中，它的效果和在 HTML 中完全不同。首先，没有终止的 <P> 标签在 Flash 中并不会像在正规 HTML 中那样产生换行效果。注意 Flash 和 Web 浏览器输出之间的不同：

```
I hate filling out forms. <P>So sometimes I don't.
// Flash输出:
I hate filling out forms. So sometimes I don't.
// Web浏览器的输出:
I hate filling out forms.
So sometimes I don't.
```

在 Flash 中要求结束的 </P> 标签，以便进行分行。例如：

```
<P> I hate filling out forms.</P>So sometimes I don't.
```

此外，在 Flash 中，<P> 可产生一个单独的分行，和 <BR> 一样，但是在 Web 浏览器中，<P> 传统上会引起两次分行。考虑下面的代码：

```
<P>This is line one.</P><P>This is line two.</P>
```

在 Flash 中显示的这个代码，两行之间并没有空隙，比如：

```
This is line one.
This is line two.
```

在 Web 浏览器中显示的代码，两行之间会有空行，比如：

```
This is line one.
```

This is line two.

因为 **<P>** 标签用在 Flash 中与用在 Web 浏览器中的效果不同，我们通常使用 **<BR>** 标签来分行。但是，**<P>** 标签的 **ALIGN** 属性对中央对齐、右对齐和左对齐仍然是有效的，比如：

```
<P ALIGN="CENTER">Centered text</P>
<P ALIGN="RIGHT">Right-justified text</P>
<P ALIGN="LEFT">Left-justified text</P>
```

## <BR> (分行)

**<BR>** 标签在文本体内可产生分行的效果，它和 **\n** 转义序列或者 **newline** 关键字是等价的。考虑下面的代码：

```
This is line one. <BR>This is line two.
This is line one. \nThis is line two.
```

两种语句在 Flash 中都会显示成：

```
This is line one.
This is line two.
```

## <A> (锚或者超文本链接)

**<A>** 标签创建一个超文本链接。当用户点击用 **<A>** 标注的文本时，标签中 **Href** 属性所指定的文档就会载入到浏览器中。如果播放器在独立模式下运行，系统中的默认 Web 浏览器就会启动，文档被载入到浏览器中。

**<A>** 标签的普通格式为：

```
<A HREF="documentToLoad.html">linked text</A>
```

例如，要链接到一个视频游戏，可以使用：

```
<A HREF="http://www.quake3arena.com">nice game</A>
```

和 HTML 一样，URL 对于当前页来说可以是相对的，也可以是绝对的。通常，用锚标签跟踪的链接会让当前的影片被锚标签中 **HREF** 所指定的文档所替代。使用 **TARGET** 属性，我们可以设置装载所链接文档的窗口名称，比如：

```
<A HREF="documentName" TARGET="WindowName>linked text</A>
```

如果一个名为 *windowName* 的窗口不存在，浏览器将建立一个新的窗口，并将它命名为 *windowName*。要在它自己的无名窗口中显示文档，可以使用 *\_blank* 关键字，比如：

```
<A HREF="mypage.html" TARGET="_blank">linked text</A>
```

注意，当我们通过 TARGET 属性设置窗口的时候，我们不能控制新窗口的大小或者工具栏排列。要在链接的时候指定窗口的大小，必须使用 JavaScript。用 JavaScript 建立自定义的附属窗口的技术，可参考：

*http://www.moock.org/webdesign/flash*

关于在 ActionScript 中与 JavaScript 通信的更多信息，请参见第三部分以及本章后面的内容。

TARGET 属性也可以用来将文档装载到框架中，比如：

```
<A HREF="documentName" TARGET="frameName>linked text</A>
```

Flash 锚标签并不总是和 HTML 中的锚标签一样。例如，不可以在 Flash 中使用锚标签的 NAME 属性。因此，不存在文本体内的内部链接。另外，Flash 链接都不使用下划线和高亮显示。链接的下划线和颜色必须用前边描述过的 <U> 和 <FONT> 标签来手动插入。

## 锚标签的 Tab 键控制顺序

在 Flash 5 中，锚标签并不包含影片的 Tab 键顺序，因此也就不能用键盘来访问。如果你的内容必须让键盘和交互输入设备可以访问，你应该对链接使用按钮，而不使用锚标签。

## 引用属性值

在 Flash 之外，HTML 属性值可以用单引号、双引号或者什么都不用来引用。下面的标签在大部分 Web 浏览器中都是合法的：

```
<P ALIGN=RIGHT>
<P ALIGN='RIGHT'>
<P ALIGN="RIGHT">
```

但是在 Flash 中，不允许没有引号的属性值。例如，语法 `<P ALIGN=RIGHT>` 在 Flash 中就是非法的。但是，单引号和双引号都可以用来界定属性值。在构成包含 HTML 属性的文本域值的时候，必须注意要正确地引用属性，使用一种类型的引号来划分串本身，使用另外一种类型的引号来划分属性值。例如：

```
// 这些例子都是合法的
myText = '<P ALIGN='RIGHT'>hi there</P>';
myText = '<P ALIGN='RIGHT">hi there</P>';
// 这个例子会引起错误，因为串和属性都用双引号来分隔
myText = "<P ALIGN="RIGHT">hi there</P>";
```

关于使用引号来形成串的更多信息，请参见第四章。

## 未被承认的标签和属性

和 Web 浏览器一样，Flash 会忽略那些它认不出的标签和属性。例如，如果将下面的值赋给 Flash 中的一个 HTML 文本域：

```
<P>Please fill in and print this form</P>
<FORM><INPUT TYPE="TEXT"></FORM>
<P>Thank you!</P>
```

输出会是：

```
Please fill in and print this form
Thank you!
```

FORM 和 INPUT 元素在 Flash 中不被支持，因此都将被忽略。类似的，如果使用诸如 `<TD>` 这样的容器元素，那么内容会被存储起来，但是置标将会被忽略。例如：

```
myTextField = '<TABLE><TR><TD>table cell text</TD></TR></TABLE>';
```

输出下面的行，但是没有表格的格式：

```
table cell text
```

## 使用 HTML 输出

使用文本工具手动地输入到文本域中的HTML文本不会被显示为HTML格式。要在屏幕上显示HTML格式的文本，必须将HTML文本通过ActionScript赋给一个动态文本域。例如：

```
myTextField = "<P><B>Error!</B> You <I>must</I> supply an email address!</P>";
```

为HTML文本域嵌入一种字体只能嵌入该种字体的一种类型。例如，一个文本域在字符面板中被设置为Arial粗体，它就只支持Arial粗体的字符类型。如果我们使用HTML来将Arial设置为其他类型（比如斜体），或者使用不同的字体（比如Garamond），那么标注的文字就会消失，除非对应的字体被嵌入到影片中！

例如，假设我们创建了一个文本域，称为output。在output文本域的字符面板中，我们选择Arial，设置为斜体。在文本选项面板中，我们嵌入整个的Arial斜体字。然后，我们设置output来显示HTML。最后，我们将下面的值赋给文本域：

```
output = '<P><I>My</I>, what<B>lovely</B>  
+   <FONT SIZE='24'>eyes</FONT>you have!</P>';
```

当影片播放的时候，下面的文本就会出现在文本域中：

*My*

我们赋给output的其他所有东西都丢失了！只有HTML中的静态文本会得到显示。其他文本要求Arial字体的其他变体，但是我们并没有嵌入——what，eyes和you have都是非静态的，lovely是粗体。

对于在HTML文本域中使用的每一种字体和变体，我们必须嵌入对应的字体。要这么做有两种方式：

- 建立一个虚文本域，从视窗中隐藏，将所要使用的字体在字符面板中选中，并且在文本选项面板中嵌入。
- 为影片库添加一个新的字体符号，将字体和影片一起导出。

下面是在影片中嵌入用在文本域中的Arial粗体字的步骤：

1. 选择 Window → Library。
2. 选择 Options → New Font。Font Symbol Properties（字体符号属性）对话框出现。
3. 在 Font 下，选择 Arial。
4. 在 Style 下，选择 Bold。
5. 在 Name 下，输入 **ArialBold**（这是一个表面的名称，只用在 Library 中）。
6. 在 Library 中，选择 ArialBold 字体符号。
7. 选择 Optoins → Linkage。
8. 在 Symbol Linkage Properties（符号链接属性）对话框中，选择 Export This Symbol。
9. 在 Identifier 框中，输入 **ArialBold**。对于我们的目的来说，在这里所输入的名称并不要紧。导出符号标识符只用来共享库。

注意，字体的每一种类型都必须单独嵌入。如果在一个文本域中使用 Arial 粗体，Arial 斜体和 Arial 粗斜体，那么必须嵌入所有这三种字体类型。下划线不表示一个字体类型，字体大小和颜色也同样。

但是，如果在文本选项面板中不使用任何嵌入字体选项，那么 Flash 就完全依赖系统的字体，如果用户在系统中装有相应的字体类型，正常、粗体、斜体几种字体就可以得到显示。

要确保文本在所有的平台和用户系统中的显示都一致，应该嵌入文本域所需要的所有字体。

## 使用 HTML 输入

尽管HTML通常用于文本域的显示工作，但是它也可通过可以使用HMTL的或者一个正规（非 HMTL）的用户输入文本域来进入到影片当中。

当正规文本进入到能使用HTML的用户输入文本域中时，HTML置标标签就会自动添加。例如，文本“Hi there”会转换为HTML值：

```
'<P ALIGN='LEFT'><FONT FACE='Arial' SIZE='10' COLOR='#000000'>Hi there</FONT></P>
```

当HTML标签被输入到可以使用HTML的用户输入文本域中的时候，<和>字符就被转换为&gt;和&lt;。例如，文本“<B>hi there</B>”会转换为值：

```
'<P ALIGN='LEFT'><FONT FACE='Arial' SIZE='10' COLOR='#000000'>&lt;B&gt;hi there&lt;/B&gt;</FONT></P>'
```

可以使用HTML的用户输入文本域可以用来创建非常简单的HTML数据输入系统。

当正规的或者HTML文本被输入到普通（非HTML）用户输入文本域中的时候，不发生文本修正动作。正规的用户输入文本域允许原始HTML代码输入到影片中而不会变形。

可以在在线代码库中访问到能使用HTML和正规的用户输入文本域数据输入的示例。

## 从HTML链接执行JavaScript

在大部分能使用JavaScript的Web浏览器中，可以从一个将javascript协议用作HREF属性值的锚标签来执行JavaScript语句。例如：

```
<A HREF=" javascript:square(5); '>Find the square of 5</A>
```

在ActionScript中，也可以从<A>标签执行JavaScript语句，如下所示：

```
myTextField = "<A HREF=' javascript:alert(5); '>display the number 5</A>";
```

但是，要在JavaScript语句中包括串的值，必须使用HTML条目“"来代表引号，比如：

```
myTextField = '<A HREF=' javascript:alert("hello world"); '>' + "display hello world</A>';
```

## 从HTML链接调用ActionScript函数

虽然一般的ActionScript代码语句不能够从Flash的<A>标签来执行，但是ActionScript的函数却可以这样。要从一个锚标签调用ActionScript函数，可以使用下面的语法：

```
<A HREF="asfunction:myFunctionName">invoke the function</A>
```

函数调用操作符()在从锚标签调用一个 ActionScript 函数的时候是不能使用的。除了从锚标签调用 ActionScript 函数之外，也可以用下面的语法将参数传递给那个函数：

```
<A HREF="asfunction:myFunctionName,myParameter">invoke the function</A>
```

myParameter 是传递的参数值。在调用的函数中，myParameter 总是一个串。要从锚传递更多的信息给函数，可在 myParameter 值中使用一个分隔符，然后在函数中我们自己将该串分隔开。例如，下面的函数调用传递两个值，由 | 符号隔开，传递给 *roleCall()* 函数：

```
<A HREF="asfunction:roleCall,megan|murray">invoke the function</A>
```

下面是 *roleCall()* 函数。注意一下它是如何用 *split()* 方法将值分隔开的：

```
function roleCall (name) {
    var bothName=name.split('|');
    trace("first name: " + bothNames[0]);
    trace("last name: " + bothName[1]);
}
```

## 关于文本域选择

当用户选择了动态或者用户输入文本域的一个部分，所选字符的位置就被存储在一个特殊的内置对象 *Selection* 对象中。使用 *Selection* 对象，可以检查用户选择的是文本域的哪个部分，甚至可程序化地选择文本域的某个部分。*Selection* 对象也可以告诉我们哪些文本域当前被用户选中。最后，可以使用 *Selection* 对象将键盘焦点设置到一个特定的文本域中，让用户在暗示的位置输入。

要学习如何处理文本域选择方面的东西，请参见第三部分。

## 空文本域和 for-in 语句

要检查时间线上所有的变量值，可以使用第六章中所描述的 *for-in* 语句。但是，未

定义的文本域（出现在屏幕上但是包含的值为 `undefined`）不会被 `for-in` 语句列举出来。（只包含空串（`""`）的或者空格的域并不认为是空的，因此会被列举出来。）

未定义的文本域在 `for-in` 循环中不可见，这给查错脚本带来了问题。使用 `for-in` 循环来扫描文本域系列的脚本必须考虑未定义文本域。例如，在这个例子中，我们试图检查一个名为 `formClip` 的影片剪辑，以查看它是否有什么变量包含空串：

```
for (i in formClip) {  
    if (formClip[i] == "") {  
        trace(i + ' is empty! don't submit the form!');  
        break;  
    }  
}
```

代码这时并不会像预想的那样运作，因为未定义文本域不会被循环列举出来，根本不会被检查到。要迫使未定义文本域被 `for-in` 循环列举出来，必须特意地将空串赋给相应的时间线变量。例如，我们可以将脚本添加到 `formClip` 的帧，以完善前面的例子：

```
// 将文本域赋予空串，这样它们就会在 for-in 循环中出现  
formField1 = "";  
formField2 = "";
```

## 小结

我们已经到了这样一个必然的阶段；指导性的漫游结束了，你开始去探究自己的项目和想法。前面的两章教会了我们如何创建表单，以及在屏幕上显示信息，还有如何获得用户输入。在参考部分之前的最后一步——调试代码——将教给你在以后未知的路程中所仅剩的技术。

# 第十九章

## 调试

到现在为止，我们已经讨论了很多技术以及实现许多目标的语法。但是，当你开始编写你自己的 ActionScript 时，难免会遇到数不清的错误（特别是开始时，还会犯语法和概念上的错误）。但是不要担心！即使是经验丰富的程序员，也要花很多时间来调试（整理有问题的代码）。

认真测试产品非常重要，这样可以首先发现错误。这意味着要在需要支持的所有平台上测试各种浏览器及其不同的版本。测试应在不同类型的 Windows 下进行，如果合适，还可以在 Flash 插件程序的旧版本中进行。可以在下面的网址找到这些版本：

*<http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm>*

关于测试和质量保证 (QA) 的讨论超过了本书的范围。可以说，你应该有一个恰当的测试和 QA 过程，以及一个测试报告单，你可以从报告单上获得详细的细节（比如平台、浏览器版本、Flash 插件程序版本，以及可复制的步骤）以重现错误，这是修正这些错误的第一步。

调试是编程的一个不可缺少的部分，它能区分程序员水平的高低。新手会因为原先看到的错误莫名其妙地消失而感到高兴。但经验丰富的程序员知道，错误会在最不该出现的时候重新出现，虽然它是断断续续的（可能尤其如此），因而需要对其进行更深入的研究。换句话说，稚嫩的程序员会逃避错误消息，并因严重的错误而丧失

信心；而熟练的程序员却会依赖错误消息。他们知道，容易再生的错误是最容易修改的。

成功的调试需要有逻辑的、系统的研究技巧，以及对错误查找工具的深入理解。在本章中，我们只简要地探讨一些调试工具基础和一些解决代码问题的一般技巧。记住，调试是对我们能力的挑战。任何给定的问题通常是由其他上游（弊病）问题所引起的。我们会使用调试工具来研究各种东西在实际上是否以起初的设想来操作，这会让我们最终理解和消除明显的错误也就是故障现象。

## 调试工具

ActionScript 配备有下面的调试工具：

- *trace()* 函数
- 列举变量命令
- 列举对象命令
- 带宽调节
- 调试器

所有这些工具都在测试影片模式中使用。要进入测试影片模式，我们用 Control → Test Movie 从制作工具中导出一个影片。

除了这些普通的调试工具之外，Flash 还在影片导出的时候或者执行语法检查的时候向输出窗口发送错误信息。（语法检查命令列在动作面板右上角的箭头按钮下）错误消息对出错点的确认通常可以精确到源代码块中的某一行。Macromedia 的 ActionScript 参考指南中给出了对不同错误类型的说明。

注意，并不是所有的错误都会发出错误消息。例如，错误的计算结果，即使它不会破坏浏览器。还要注意，有两种类型的错误消息：所谓的编译期间错误消息和运行期间错误消息。编译期间错误消息发生在导出脚本的时候，运行期间错误消息要直到你运行 Flash 影片、达到错误点的时候才会发生。编译期间错误表示有某种语法问题，比如括号丢失或者没有后引号。第三部分给出了每个命令所需要的精确语法，第十四章介绍了恰当的 ActionScript 语法。

运行期间错误有多种形式，它可能表示的不是当前运行代码中的问题，而是由于使用了以前不正确的计算结果造成的。例如，假设你要将从 *loadVariables()* 命令接收到的值发送给 Web 服务器。如果响应命令的 Perl 脚本没有以正确的格式提供正确数据，你就需要纠正 Perl 脚本。你的 Flash 脚本本身可能是正确的，但是因为接收了错误的数据而产生不正确的结果。

这就产生了一个重要的技术——防御性编程。可以通过检查潜在的问题条件而避免很多错误或者潜在的错误，这被称为“错误检查”（如果它属于用户输入，也被称为数据确认）。例如，在显示测试的题目之前，你要检查这些题目的载入是否正确。你也可以检查每一道题目，以确定它是正确的显示格式。如果所提供的数据输入不正确，你应该显示一个相应的错误消息，让编程者或者用户采用正确的做法。

## trace() 函数

在 ActionScript 中，最有效的错误源辨别工具之一是 *trace()* 函数，它也是最简单的工具之一。正如我们在本书中所看见的，*trace()* 将表达式的值发送到测试影片模式的输出窗口。例如，如果将下面的代码添加到影片：

```
trace("hello world");
```

文本“hello world”就会出现在输出窗口中。类似的，下面我们要显示一个变量的值：

```
var x = 5;  
trace(x); // 在输出窗口中显示 5
```

使用 *trace()*，我们可以检查变量、属性和对象的状态，也可以跟踪代码的进程。通常，通过确认脚本中每个操作的结果，我们就可以知道问题出在哪里。例如，假设一个函数要返回一个值，但是我们发现，使用 *trace()* 命令返回的值是 *undefined*（也就是说，它在输出窗口中什么也不显示），这时我们就知道应该检查函数中的细节，确定用 *return* 命令来返回一个有效值的动作是正确的。

## 列举变量命令

当影片在测试影片模式下运行的时候，可以检查在影片中定义的当前变量的值，使用 *Debug → List Variable* 命令就可以实现。列举变量告诉我们当前活动影片中所有

变量的名称和位置，并列出它们的值。因为函数和影片剪辑存储在变量里，列举变量命令也会告诉我们影片的函数和影片剪辑。

例19-1给出了列举变量的输出示例。注意，变量rate虽然定义了，但是却显示为undefined。这种细微的地方通常很难用*trace()*探测到，因为*trace()*将值undefined转换为空串（“”）来显示。

#### 例 19-1：列举变量输出示例

```
level #0:  
  Variable_level0.$version = 'WIN 5.0,30,0'  
  Variable_level0.calcDist = [function]  
  Variable_level0.deltaX = 194  
  Variable_level0.deltaY = 179  
  Variable_level0.rate = undefined  
  Variable_level0.dist = 264  
Movie Clip: Target = '_level0.clip1'  
Movie Clip: Target = '_level0.clip2'
```

注意，列举变量命令中的两个*trace()*都只给出某一时刻的简单状态。通常，我们需要随着时间的变化来反映变量的值，或者重复地进行检查。调试器（后面要讨论）允许你跟踪变量值的改变情况。

## 列举对象命令

列举对象命令产生影片中所定义的文本、形状、图形和影片剪辑的目录。要执行这个命令，在测试影片模式中选择 Debug → List Objects（列举对象）。注意，列举对象命令不包括程序中数据对象（类的实例）的列表，数据对象要通过列举变量命令来给出。

例19-2显示了列举对象的输出示例。注意，可编辑的文本域是明确给出标签的，自动命名的影片剪辑实例被显示出来（例如，\_level0.instance1）。

#### 例 19-2：列举对象输出示例

```
Level #0: Frame = 1  
  Shape:  
    Text: Value = "variables functions clip events startDrag stopDrag Math"  
    Text: Value = "this movie demonstrates a little math, variables, movie clip events"  
    Text: Value = "draggable distance"  
    Text: Value = "calculator"  
  Movie Clip: Frame=1 Target="_level0.instance1"
```

```
Shape:  
Text: Value = "distance between clipstoAll:horizontal:vertical:"  
Edit Text: Variable=_level0.d;st Text="322"  
Edit Text: Variable=_level0.deltaX Text="174"  
Edit Text: Variable=_level0.deltaY Text="138"  
Movie Clip: Frame=1 Target="_level0.obj1"  
    Shape:  
Movie Clip: Frame=1 Target="_level0.obj2"  
    Shape:
```

同样，列举对象只提供某一时刻的快照。如果发生变化，需要重新运行它才能得到对象当前的值。

## 带宽调节

带宽调节用来模仿不同调制解调器速度的影片下载。使用带宽调节，可以测试影片的执行性能、预下载代码，以及在影片播放期间跟踪主影片的播放头的位置。执行下面的步骤，开启带宽调节：

1. 在测试影片模式中，选择 View（视图）→ Bandwidth Profiler（带宽调节）。
2. 在调试菜单下，选择需要的下载速度。
3. 要模仿该速度下的影片下载，选择 View → Show Streaming（显示数据流）。

有很多东西会影响Flash的执行，比如使用的资源以及播放器上的渲染命令。例如，使用较大的位图，以多曲线渲染复杂的图像，以及过分使用透明通道都会降低Flash执行的质量。信息下载和渲染时间通常减小带宽和ActionScript所要求的处理器时间。也就是说，ActionScript通常比诸如C这样的编译语言慢得多。

对ActionScript来说，最费时的操作就是等待数据的上载或者下载，或者重复执行某些动作（比如检查一个大的数组）。

---

**警告：**在输出窗口中显示项目比在ActionScript中执行“看不见”的操作要慢很多。如果一个简单的影片执行得很差，可以试着取消所有的*trace()*语句，或者在测试影片模式外播放影片。

---

对于编写最优化代码的讨论超出了本书的范围，但是我们会提供一些小技巧：

- 如果某种工作可以用非循环的方法完成，并与使用循环方法具有同等功效，就不要在循环里重复地执行某个操作。
- 不要在循环中等待某个事件的发生。事件可能要等很长时间才会发生，或者永远不会发生，等待会让你的执行变慢，或者锁定整个应用程序。相反，依靠事件处理器（比如 *on (load)*）在事件发生或结束的时候触发会更好。
- 尽可能归纳你的代码（或者用智能剪辑来达到这个目的）。这会减少需要下载的代码数量。例如，我们不必编写两个几乎相等的子程序，每个长 5KB，而可编写一个集中的子程序，长为 5KB，然后用不同的参数调用它两次。（代码的集中在第九章、以及第十六章中已经讨论过了）
- 如果你使用的是 Flash 5 播放器，ActionScript 的执行性能对你的作品非常重要，可以试试使用 Flash 4 中原来的语法类型来代替新的技术。在 Flash 5 中，指定的操作用 Flash 4 的语法来实现的时候会比较快。例如，Flash 4 的 *substring()* 函数比 Flash 5 的 *substring()* 和 *substr()* 方法快，Flash 4 的 *Tell Target* 比 Flash 5 的点语法快。
- 选择 Publish Settings (发布设置) → Flash → Options → Omit Trace Actions (省略跟踪动作)，以便在导出影片的时候将 *trace()* 语句全部去掉。
- 记住，删除和重新添加一个剪辑比移动一个已经存在的剪辑要费事，应尽可能重复使用你的影片资源。
- Flash 一般优化技术的列表，可以参见 [http://www.macromedia.com/support/flash/publishexport/stream\\_optimize/stream\\_optimize.html](http://www.macromedia.com/support/flash/publishexport/stream_optimize/stream_optimize.html)。

## 调试器

调试器是一个很有效的工具，让我们可以有条理地访问影片中属性、对象和变量的值，并让我们可以在运行中改变变量的值。

要启动调试器，在 Flash 制作工具中（不是在测试影片模式里）选择 Control → Debug Movie。你也可以在 Web 浏览器中使用调试器，只要满足以下条件：

- 被检查的影片最初导出的时候有调试许可。
- 用来查看影片的播放器是调试播放器。

- Flash 制作工具在你要调试的时候是运行的。

要输出一个影片，并带有浏览器中的调试许可，选择File → Publish Settings → Flash → Debugging Permitted（允许调试），然后可以选择性地附带一个密码，以防止好奇的眼睛来打探你的代码。要在浏览器中安装调试播放器，可以用在硬盘驱动器中安装Flash时候的/Players/Debug/文件夹中的安装程序。要确保在查看影片的时候能够进行调试，在Windows中，在影片上点击鼠标右键（在Macintosh中为Ctrl-click），并选择调试器。

---

**注意：**不是所有的Flash播放器版本都有对应的调试播放器。检查Macromedia的支持站点，寻找调试播放器的最新版本，<http://www.macromedia.com/support/flash>。

---

调试器（显示列表）的上半部给出了影片剪辑层次。要检查指定影片剪辑的属性和变量，在显示列表中选中它。调试器的下半部包含三个标签：属性、变量和监视，它们显示的所选剪辑的属性和变量会动态地更新。要设置任何属性或变量的值，双击它的值，然后输入新的数值。要挑选一个或多个项目进行详细审查，可以在属性或变量标签中选中它们，然后从调试器右上角的箭头按钮中选择Add Watch（添加监视点）。所有被监视变量都被添加到监视标签中（这可让我们同时查看不同影片剪辑中的变量）。

关于使用Flash调试器的技巧的更多细节，参见《ActionScript Reference Guide》（ActionScript参考指南）中的“Troubleshooting ActionScript”下的Macromedia详细文档。如果你没有参考指南，可以参考<http://www.macromedia.com/support/flash>下的Macromedia网站，也可以参考Flash制作工具里帮助菜单下的内容。

## 调试方法

我们来简要地看一下代码调试中所包含的一些技巧。调试可以被分为三个阶段：

- 问题的识别和再生。
- 确定问题来源。
- 修正错误。

## 识别错误

我们经常将代码问题看成编程活动过程的一部分。也就是说，我们编写一些代码，测试影片，然后发现影片的运行有些问题，这样就要识别错误。

问题当然是发现得越早越好。因此，代码的编写过程就是编写和测试不断重复的过程——编写数行代码、导出影片，确定代码运行正确，然后再写几行代码、导出影片，依此类推。在对程序进行整体测试之前，要确定程序各部分都运行正确。编写一个复杂的代码体时，最好要在中途进行测试。

不要因为你自己没有发现什么错误就认为你的影片是完美的。应经常安排针对目标用户的外部测试，特别是如果你创建的代码是产品的一部分，或者是客户想要的服务时。正如前面所描述的，执行错误检查的过程，可以阻止不正确的数据输入可能造成的问题。例如，如果你编写的函数需要一个整数作为参数，可以使用 `typeof` 操作符来确定输入的参数是正确的类型，还要测试最终条件，比如特别大、特别小，或者负的值，包括 0。

不要低估发现了复制问题的最少再生步骤（minimum reproducible steps）的价值。这是重现错误的最少步骤。像“我将它播放了一个小时，然后冻结了”这类的错误报告并没有什么用。有用的错误报告包括一些编号的步骤，比如：

1. 为年份输入 0。
2. 点击计算按钮。
3. 结果显示“NaN”而不是美元数目。

## 确定错误来源

我们一旦识别了错误，就要开始寻求解决之道。我们的第一个任务就是找到错误的来源，不管要追溯多远。错误可以被看成因为多年不良的饮食习惯而造成的心脏病发作。心脏病发作是很明显的症状，但是你必须经常纠正心脏病发作以前的不良习惯。大部分错误都是由错误的假定造成的：我们假定已经正确输入了变量的名称但实际上却没有，或者我们假定一个文本域存储了数字数据但它并不是这样。通过执行一系列的 `trace()` 语句，使用调试器或者列举变量命令，我们可以依靠解释程序对代码的理解来测试我们的假定。

例如，下面的例子中有一个错误，它将 `status` 错误地设置为 “equal”：

```
var x = 11;
isTen(x);
function isTen(val) {
    if (val = 10) {
        status = "equal";
    }
}
```

要找到代码中究竟什么发生了错误，我们将我们认为代码会怎么做与它实际会怎么做相比较，一次一个步骤：

```
// 将x设置为11
var x = 11;

// 我们来看它是否真的起作用
trace(x); // 对，显示的是：11

// 调用isTen()函数
isTen(x);

// 现在是我们的函数
function isTen(val) {
    // 我们来确定我们的函数被调用了
    trace('isTen was called'); // 对，显示的是："isTen was called"

    // 现在我们来确定参数传递是否正确
    trace('val is ' + val); // 对，显示的是："val is 11"
```

暂停片刻，注意发生的情况——我们检查了我们的代码，到现在为止，一切顺利。我们的变量得到了正确的设置，`isTen()` 函数被调用，参数被正确地传递。

---

**注意：**很多错误的发生是因为代码并不像你想象的那样去执行！我们可以使用 `trace()` 语句来检测代码的指定部分是否实现了原来的目的。

---

通过消除错误的过程，我们已经知道，代码的问题要么出在条件语句 `if(val=10)` 中，要么出在文本域赋值语句 `status="equal"` 中。下面使用 `trace()` 来显示测试表达式的值，以检查条件语句（我们希望显示的是 `true` 或者 `false`）：

```
trace(val = 10);
```

找到了！输出窗口显示的是 10，而不是我们所希望的 true 或 false。在进一步的检查中，我们发现测试表达式是一个赋值语句，而不是比较语句！我们在相等比较中遗失了一个等号。表达式 if(val=10) 应该是 if(val==10)。

显然，不是所有的错误都像条件语句错误这么简单（它是一个非常普通的错误），但是，我们使用的方法对大部分的错误检测都是有效的：执行一系列 trace() 函数来创建一个运行的，逐项的报告，以反映影片代码的实际效果，将调试器用作 Macromedia 文档的说明。

## 错误的一般来源

表 19-1 列出了 ActionScript 中一些普通的错误来源。

表 19-1 ActionScript 错误来源示意

问题	描述
代码位置错误	所有的代码都必须添加到影片剪辑、帧或者按钮上。观察动作面板的标题，看看代码是否添加到了预定的地方——将代码添加到帧的时候，动作面板的标题就是帧面板；将代码添加到影片剪辑或者按钮上的时候，动作面板的标题就是对象面板。如果想将脚本添加到一个特定的帧，要在输入代码之前确定时间线的帧已经被选中了，在要放置代码的地方应该有一个关键帧。如果想在影片剪辑或者按钮上添加脚本，要在输入代码之前确定场景上的对象是被选中的。使用影片浏览器 (Window → Movie Explorer) 来跟踪代码被添加到的地方。
遗失事件处理器	属于影片剪辑和按钮的代码必须包含事件处理器。对于影片剪辑，使用： <pre>onClipEvent(event) {     // 语句 }</pre> 对于按钮，使用： <pre>on(event) {     // 语句 }</pre> event 是处理器对应的事件名称。“语句必须出现在处理器中”这个错误表示你遗漏了事件处理器。参见第十章。

表 19-1 ActionScript 错误来源示意（续）

问题	描述
错误的影片剪辑引用	引用了不存在的影片剪辑，或者影片剪辑引用错误。检查所有的实例是否被命名，以及实例的名称是否和提供的引用匹配。参见第十三章以获得合法的影片剪辑引用构成信息。
类型转换错误	数据转换没产生预期的结果。比如， <code>3 + "4"</code> 产生串“34”，而不是数字7。类似的，串“true”转换成了布尔值 <code>false!</code> 学习第三章中的类型转换规则。用 <code>typeof</code> 操作符来检查数据类型。
遗失分号	因为没有分号，因此过早结束语句。参见第十四章。
引号标记问题	包含引号字符的串扰乱了串直接量。参见第四章。
错误使用文本域数据	文本域被当作数字或其他数据类型来对待，而不是串。用户在文本域中输入的通常是一个串值，应该在其被当作其他数据类型之前手动进行转换。
作用域问题	变量、属性、剪辑或者函数在错误的区域被引用。例如，一个剪辑处理器上的语句试图调用该剪辑的父时间线上的函数。参见第十章、第二章和第九章。
全局函数和方法混淆	一些全局函数和影片剪辑方法同名。显然，这会引起重叠问题。参见第十三章。
内容尚未载入	对剪辑、属性、函数或者变量的引用不能实现，因为其内容尚未载入。要确定所有的内容都被载入，可以使用 <code>MovieClip._frames-loaded</code> 属性，如第三部分所示。
大小写不正确	一些关键字在 ActionScript 中是区分大小写的。如果你将 <code>onClipEvent</code> 错误地写成了 <code>onclipevent</code> ，ActionScript 会认为你要调用一个自定义的函数 <code>onclipevent</code> ，而不是使用内置的 <code>onClipEvent</code> 处理器关键字。这样，当它在 <code>onClipEvent</code> 语句块的开头遇到(符号的时候就会发生错误（它期望的是用分号来作为 <code>onclipevent</code> 函数调用的结束）。参见第十四章。

## 修正错误

在一些情况下，修正识别到的错误是自然要做的。例如，如果我们发现了一个因为丢失串中的引号而引起的错误，我们会添加上引号来改正。

在复杂的程序中，改正错误可能会是一个严重的挑战。如果一个错误很难改正，可以考虑下面的方法：

- 不要害怕重新编写代码。在很多情况下，对复杂代码的最好的改正方法就是重新构建整个系统，从头开始。重新建立的系统总是会比第一次的要更快、更出色。大部分专家都同意这个做法（例如，Quake III 就完全是 Quake II 工具的重写）。不过，新的代码仍然需要再调试。不要把代码完全抛弃，保持好的部分，只重新写有问题的部分。
- 将有问题的部分分隔开，进行单独的影片测试。完全单独处理每一个系统部分，然后一次合并一个。
- 回顾你的代码。不要害怕，我们都会为一年前编写的代码感到局促不安。
- 在附录一中寻求帮助。例如，FlashCoders 邮件列表都是关于 ActionScript 问题的。

对于编程问题的更多建议，参见 Kent Beck (Addison Wesley) 编写的《Extreme Programming Explained》(极限编程说明) 以及 Steve McConnell 编写的《Code Complete》(代码全编) (Microsoft 出版)。

## 小结

我们的 ActionScript 讨论已经结束了，但是你的旅程才刚刚开始。读了第一部分“ActionScript 基础”和第二部分“ActionScript 应用”之后，你会懂得如何同 ActionScript 对话——现在该将你的知识应用到实际工作中去了。在你开始动手之前，先来看看下面的最后忠告：

- 和任何艺术形式一样，学习编程是一个过程，而不是一个短期的事件。在真正开始编程之后，你将会学到更多关于编程的东西。考虑第一、九、十一和十三章中的多项选择测试——我们将它重新建立了四次！每一次，我们都重新设计方法，添加新的功能，学习一些以前没有考虑的东西。正如每一幅绘画作品都让绘画者得到关于主题的新体会一样，创建和再创建应用程序会启发你的新思维。

- 在第三部分中有实际的帮助，它包含 ActionScript 内置函数、属性、类和对象的细节描述。你可能不想将它从头读到尾，不过你应该了解每一个主题，这样你就会对 ActionScript 的能力有一个全面的认识。
- 你可以经常重新看这本书或第三部分，每次都会有新的体会，因为你会在一个更高层次上来考虑这些信息，并且能够将概念与实际经验联系起来。你可以将第三部分当作一个词典，在工作的时候放在身边。
- 有很多 Flash 开发者会提供想法和一些解决方法，以及——更重要的是——共享源代码！应尽可能地对代码进行研究。对于你不理解的东西，可以参考本书。Macromedia 在下面的网址上提供了一个网站列表，以及关于 Flash 的邮件发送单。[http://www.macromedia.com/support/flash/ts/documents/flash\\_websites.htm](http://www.macromedia.com/support/flash/ts/documents/flash_websites.htm)。
- 不要局限于对本书的 ActionScript 的研究，应从多个角度来考虑问题，参考其他资源，比如在附录一和前言中所列出来的那些。在 <http://www.moock.org/moockmarks> 下有一个 Flash 资源的列表，在 <http://www.moock.org/webdesign/books> 下有一个有价值的 Flash 书籍介绍。
- 最后，记住本书的支持站点，<http://www.moock.org/asdg>，其中有很多代码示例、技术要点和新主题的讨论。

祝你编码愉快！什么时候可以对我来一次 *while* 循环。:)



---

# 第三部分

## 语言参考

这一部分将说明 ActionScript 所支持的每一个内置类、对象、函数、属性和事件处理器的运用。在日常的创作中，你会经常用第三部分的内容来完成特定的任务。

- ActionScript 语言参考



---

# ActionScript 语言参考

ActionScript 语言参考中给出了 ActionScript 中所支持的所有类和对象，并说明了它们的一般用途，用法，属性，方法和事件处理器。并且包含了全局函数和全局属性（它们不属于类或者对象，但是在影片中可以以独立的形式来使用）。

语言参考中的所有条目都是按照字母顺序列出来的。例如，全局函数 *duplicateMovieClip()* 在 *Date* 类后面（而不是在单独的“全局函数”一节中）列出。但是，每一个条目的标题清楚地表现了所描述的条目类型，这样就不会让两个重叠的项目混淆起来。例如，*duplicateMovieClip()* 在全局函数和影片剪辑方法形式中都有出现，你会发现它不但被列为 *duplicateMovieClip()*，还被列为 *MovieClip.duplicateMovieClip()* 方法。

## 全局函数

全局函数是一种内置函数，它在整个影片中都是可用的——它们可以从影片的任何帧、按钮或者影片剪辑中调用（和方法不同，方法只能用指定对象的引用来调用）。

表 R-1 列出了 Flash 5 ActionScript 中可用的全局函数。

表 R-1 ActionScript 全局函数

<i>Boolean()</i>	<i>call()</i>	<i>Date()</i>
<i>duplicateMovieClip()</i>	<i>escape()</i>	<i>eval()</i>

表 R-1 ActionScript 全局函数（续）

<i>fscommand()</i>	<i>getProperty()</i>	<i>getTimer()</i>
<i>getURL()</i>	<i>getVersion()</i>	<i>gotoAndPlay()</i>
<i>gotoAndStop()</i>	<i>#include</i>	<i>int()</i> <sup>a</sup>
<i>isFinite()</i>	<i>isNaN()</i>	<i>loadMovie()</i>
<i>loadMovieNum()</i>	<i>loadVariables()</i>	<i>loadVariablesNum()</i>
<i>maxscroll</i>	<i>newline</i>	<i>nextFrame()</i>
<i>nextScene()</i>	<i>Number()</i>	<i>parseFloat()</i>
<i>parseInt()</i>	<i>play()</i>	<i>prevFrame()</i>
<i>prevScene()</i>	<i>print()</i>	<i>printAsBitmap()</i>
<i>printAsBitmapNum()</i>	<i>printNum()</i>	<i>random()</i> <sup>a</sup>
<i>removeMovieClip()</i>	<i>scroll</i>	<i>setProperty()</i>
<i>startDrag()</i>	<i>stop()</i>	<i>stopAllSounds()</i>
<i>stopDrag()</i>	<i>String()</i>	<i>targetPath()</i>
<i>tellTarget()</i> <sup>a</sup>	<i>toggleHighQuality()</i> <sup>a</sup>	<i>trace()</i>
<i>unescape()</i>	<i>unloadMovie()</i>	<i>unloadMovieNum()</i>
<i>updateAfterEvent()</i>		

a. 在 Flash 5 中不支持。

## 全局属性

全局属性和全局函数相似，都可以从影片中的任何脚本访问。它们存储的通常是来自于代码中的任意点的有用信息。很多全局属性作用于整个 Flash 播放器，而不是针对特定的影片剪辑或者影片。

表 R-2 列出了 Flash 5 中可用的 ActionScript 全局属性。

表 R-2 ActionScript 中的全局属性

属性名称	描述
<i>_focusrect</i>	用键盘激活的按钮的高亮状态
<i>_highquality</i>	播放器的渲染质量 <sup>a</sup>
<i>Infinity</i>	一个表示数字类型的无穷值的常量

表 R-2 ActionScript 中的全局属性（续）

属性名称	描述
<code>-Infinity</code>	一个表示数字类型的负无穷值的常量
<code>_leveln</code>	播放器中的文档层级
<code>NaN</code>	非数字，一个表示无效数字的数字类型的值
<code>_quality</code>	播放器的渲染质量
<code>_root</code>	指向当前层级的主影片时间线的引用
<code>_soundbuftime</code>	预加载的流声音的秒数
<code>\$version<sup>a</sup></code>	Flash 播放器的版本

a. 在 Flash 5 中不支持

## 内置类和对象

本语言参考假定你理解了第十二章中详尽描述的类、对象和实例。ActionScript 的内置类用来创建可以控制影片和操作数据的对象。ActionScript 中的内置类为 `Array`, `Boolean`, `Color`, `Date`, `MovieClip`, `Number`, `Object`, `Sound`, `String`, `XML`, `XMLNode` 和 `XMLSocket`。要创建特定类的实例，我们使用类的构造器函数和 `new` 操作符。例如，要建立 `Color` 类的一个新的对象，我们这样使用 `Color` 构造器：

```
myColor = new Color(_root);
```

对“语言参考”中的每一个类，构造器项目显示了如何创建特定类的新对象的方法（也就是说，它表明了每一个类的构造器函数的语法）。对于类中对象可用的属性、方法和事件处理器，与类的用途和典型用法一起进行了总结。一些类所定义的某些方法或者属性可以通过类构造器本身访问，而不是在单个的实例内才能访问。这些方法和属性作为“类方法”和“类属性”被列举出来。每个类的属性、实例和事件处理器的详细内容在类的一般介绍后面按字母顺序也列了出来。

ActionScript 的特殊内置对象——`Arguments`, `Key`, `Math`, `Mouse` 和 `Selection`——与类描述一起在本书中按字母顺序排列，但是由关键词对象区分开（例如，`Math` 对象）。和真正的类不同，真正的类将用构造器函数来实例化多个对象，而这些独立的对象是从来不实例化的（也就是说，它们不能用 `new` 操作符来构造）。而且，它们是预置的对象，它们独立存在，将有关的功能集中到一个单独的包中。例如，`Math` 对象为一般的数学函数和常量提供了方便的访问。

## 开头标题

表 R-3 列出了用来确定“语言参考”中每一个条目的标题。

表 R-3 语言参考标题

标题	描述
有效性	指出条目是什么时候添加到 ActionScript 中的（或者什么时候成为一个动作的，如果条目在正式的 ActionScript 产生之前）。也包括向后兼容和不支持的情况。
概要	说明该条目所要求的抽象语法。斜体文本表示的是必须由编程者提供的代码。
参数	只针对方法或者函数的条目。描述方法和函数要使用的参数，在概要中列出。
返回	只针对方法和函数的条目。描述方法的返回值（如果有的话）。如果方法没有返回值，这个标题就被省略。
访问	只针对属性的条目。描述属性值是否可以被获取（只读）或者既可获取又可赋值
描述	说明项目如何工作，以及它如何在特定的情况下使用。
用法	描述条目的显著特性。
故障	描述和该条目有关的已知的错误。
示例	提供该条目使用情况的范例代码。
参见	列出有关主题的前后参照。

## 按照字母顺序排列的语言参考

下面的条目列出了 ActionScript 的对象和类，以及指向本书所包括的操作数和语句的索引。

### Arguments 对象

指向函数参数和当前函数

**有效性** Flash 5

**概要** `arguments [elem]`

`arguments.propertyName`

### 属性

*callee* 指向被执行函数的引用。

*length* 传递给执行函数的参数的数目。

### 描述

*Arguments* 对象存储在每个函数的局部变量 *arguments* 中，只在函数运行的时候才可以访问。*Arguments* 既是数组又是对象。作为一个数组，*arguments* 存储了传递给当前所执行的函数的值。例如，*arguments[0]* 是传递的第一个参数，*arguments[1]* 是传递的第二个参数，依此类推。作为一个对象，*Arguments* 存储 *callee* 属性，它可以用来自确认或者调用当前的函数。

### 参见

第九章

---

**arguments.callee 属性** 指向当前执行函数的引用

**有效性** Flash 5

**概要** arguments.callee

**访问** 读 / 写

### 描述

*callee* 属性存储的是一个指向当前执行函数的引用。我们可以用这个引用来再次执行当前函数，或者通过比较确认当前函数。

### 示例

```
function someFunction () {  
    trace(arguments.callee == someFunction); // 显示: true  
}  
  
// 一个未命名的递归函数  
countToTen = function () {  
    i++;  
    trace(i);  
    if (i < 10) {  
        arguments.callee();  
    }  
};
```

---

**arguments.length 属性** 传递给 argument 的参数数目

**有效性** Flash 5

**概要** arguments.length

**访问** 读 / 写

### 描述

`length` 属性存储的整数表示 `arguments` 数组中的元素数目，它等于传递给当前执行函数的参数数目。

### 示例

我们可以用 `arguments` 的 `length` 属性来确定一个函数的调用是否传递了正确数目的参数。下面的例子检查是否将两个参数传递给了 `someFunction()`。检查传递的参数类型是否正确则留给读者练习。（提示：参见 `typeof` 操作数。）代码如下：

```
function someFunction (y, z) {
    if (arguments.length != 2) {
        trace("Function invoked with wrong number of parameters");
        return;
    }
    // 进行正常的函数执行……
}
```

## Array 类

支持排序的资料列表

**有效性** Flash 5

**构造器** `new Array()`

`new Array(len)`

`new Array(element0, element1, element2, ..., elementn)`

### 参数

`len` 一个非负整数，指定新数组的大小。

`element(),...,elementn`

一个或多个要被赋值为数组元素的初始值列表。

### 属性

`length` 数组中元素的数目（包括空元素）。

### 方法

`concat()` 添加另外的元素到已经存在的数组中，以创建一个新的数组。

`join()` 将数组转换为串。

`pop()` 删除并返回数组中的最后一个元素。

`push()` 将一个或者多个元素添加到数组的末尾。

---

<i>reverse()</i>	反转数组中元素的排列顺序。
<i>shift()</i>	删除并返回数组中的第一个元素。
<i>slice()</i>	用现有数组的元素子集来创建一个新的数组。
<i>sort()</i>	按照给定的规则对数组元素进行排序。
<i>splice()</i>	从数组删除元素或者添加元素。
<i>toString()</i>	将一个数组转换为串，元素值之间用逗号分隔。
<i>unshift()</i>	在数组的开头添加一个或者多个元素。

### 描述

我们使用 *Array* 类的属性和方法来操作存储在数组对象中的元素。参见第十一章可以了解数组的定义和使用方法的更多的细节，以及本部分中所使用术语的详细定义。还要参见 [ ] 和 . 操作符，它们是用来访问数组元素的，如第五章所示。

### 用法

如果 *Array* 构造器在调用的时候只有一个整数参数，这个参数就用来设置新数组的大小，而不是第一个元素的值。如果给构造器提供了两个或者多个参数，或者如果提供的单个参数不是数字类型的，那么参数就是数组元素的初始值，而数组的大小由参数的数目来决定。

---

## Array.concat()方法

扩展现有数组以创建一个新的数组

**有效性** Flash 5

**概要** *array.concat(value1,value2,value3,...valuEn)*

### 参数

*value1,...valuEn* 一个表达式列表，这些表达式要添加到 *array* 的末尾作为新元素。

### 返回

一个包含所有 *array* 元素，后面还跟着元素 *value1,...valuEn* 的新数组。

### 描述

*concat()* 方法返回一个新数组，它是通过在现有元素后面添加新的元素而创建的。原来的 *array* 保持不变。可使用 *push()*, *splice()* 或者 *shift()* 方法来修改原来的数组。

如果一个数组被用来作为 *concat()* 的参数，那么它的每一个元素都会被单独添加。也就是说，*arrayX.concat(arrayY)* 会将 *arrayY* 的每一个元素添加到 *arrayX* 后面。这

个结果数组的长度会等于 arrayY.length+arrayX.length。但是，嵌套的数组没有这么简单。

## 示例

```
// 创建一个数组
myListA = new Array('apples', 'oranges');

// 将 myListB 设置为['apples', 'oranges', 'bananas']
myListB = myListA.concat('bananas');

// 创建另外一个新的数组
myListC = new Array('grapes', 'plums');

// 将 myListD 设置为["apples", "oranges", "bananas", "grapes", "plums"]
myListD = myListB.concat(myListC);

// 将 myListA 设置为['apples', 'oranges', 'bananas']
myListA = myListA.concat('bananas');

// 创建一个数组
settings = ["on", 'off'];
// 添加一个包含嵌套数组的数组
options = settings.concat(["brightness", ["high", "medium", "low"]]);
// 将 options 设置为['on', 'off', 'brightness', ['high', 'medium', 'low']]
// 而不是['on', 'off', 'brightness', 'high', 'medium', 'low']
```

## 参见

*Array.push()*, *Array.shift()*, *Array splice()*, 第十一章

---

## Array.join()方法

将数组转换为串

**有效性** Flash 5

**概要** `array.join()`

`array.join(delimiter)`

**参数**

*delimiter* 一个可选串，放置在新创建的串中间。如果不提供，就默认为逗号。

**返回**

一个由 *array* 的所有元素转换为串之后所构成的串，中间由 *delimiter* 分隔。

**描述**

*join()*方法返回的是由数组的所有元素所构成的串，如下所示：

1. 将串中的每个元素都转换为串（空元素转换为空串）。
2. 在每个转换后的元素串后面添加 `delimiter`, 最后一个元素除外。
3. 将所有的元素串连接为一个长的串。

注意，本身就是数组的元素通过 `toString()` 方法转换为串，因此，嵌套数组元素总是由逗号分隔开，而不是 `join()` 调用中所传递的 `delimiter`。

### 示例

```
fruit = new Array('apples', "oranges", "bananas", "grapes", "plums");
// 将 fruitString 设置为 'apples,oranges,bananas,grapes,plums'
fruitString = fruit.join();
// 将 fruitString 设置为 "apples-oranges-bananas-grapes-plums"
fruitString = fruit.join('-');
```

### 参见

`Array.toString()`, `String.split()`, 第十一章

---

## Array.length 属性

数组中的元素个数

**有效性** Flash 5

**概要** `array.length`

**访问** 读 / 写

### 描述

`length` 属性是一个非负的整数，用来指定数组中的元素个数。一个没有元素的数组长度为 0，有两个元素的数组长度为 2。注意，数组中第一个元素的索引号码为 0，因此 `length` 总是比数组中最后一个元素的索引号码大 1。

数组的 `length` 属性表示数组当前包含多少编号元素，包括空元素在内（包含 `null` 或者 `undefined` 的元素）。例如，一个数组的元素 0,1,2 和 9 可以有值，但是元素 3 到 8 却是空的。这样的一个数组长度为 10，因为它有 10 个元素位置（从 0 到 9），虽然只有四个位置被有效的值所占据。

设置数组的 `length` 属性将改变数组中的元素数目。如果增加 `length`，那么就会在数组末尾添加新的元素；如果减少 `length`，就会从数组末尾删除已经存在的元素。通过 `Array` 类方法对数组的元素进行添加或者删除的时候，`length` 属性将自动变化。`length` 属性只反映编号的数组元素，它不包括命名数组元素，那些元素会被看成数组的属性。

## 示例

```
myList = new Array("one", "two", "three");
trace(myList.length); // 显示: 3

// 循环扫描数组的元素
for (var i=0; i < myList.length; i++) {
    trace(myList[i]);
}
```

## 参见

[第十一章](#)

## Array.pop()方法

删除数组的最后一个元素

**有效性** Flash 5

**概要** `array.pop()`

### 返回

`array` 的最后一个元素值，它同时会被删除。

### 描述

`pop()`方法删除数组中的最后一个元素，将数组的 `length` 减 1，并且返回所删除的元素。和 `shift()`方法进行比较，`shift()`删除的是数组的第一个元素。

## 示例

```
myList = new Array("one", "two", "three");
trace ("Now deleting " + myList.pop()); // myList 现在是: ["one", "two"]
```

## 参见

[Array.push\(\)](#), [Array.shift\(\)](#), [Array.splice\(\)](#), 第十一章, 第五章

## Array.push()方法

在数组的末尾添加一个或多个元素

**有效性** Flash 5

**概要** `array.push(value1,value2,...valuEn)`

### 参数

`value1,...valuEn` 一个或多个值的列表，这些值要被添加到 `array` 的末尾。

### 返回

数组的新 `length`。

## 描述

*push()*方法将一个值列表附加到一个数组的末尾作为新的元素。被添加进去的元素按照其被提供的顺序来排列。它和*concat()*不同，*push()*会修改原来的数组，但是*concat()*会创建一个新的数组。它和*unshift()*也不同，*unshift()*将元素添加到数组末尾，而不是开头。

## 示例

```
myList = new Array(5, 6);
myList.push(7);           // myList 现在为[5, 6, 7]
myList.push(10, 8, 9);    // myList 现在为[5, 6, 7, 10, 8, 9]
```

## 参见

*Array.concat()*, *Array.pop()*, *Array.unshift()*, 第十一章

---

## Array.reverse()方法

反转数组的元素顺序

**有效性** Flash 5

**概要** *array.reverse()*

## 描述

*reverse*方法将反转数组中的元素排列顺序，交换第一个和最后一个元素，第二个和倒数第二个元素和，依此类推。

## 示例

```
myList = new Array(3, 4, 5, 6, 7);
myList.reverse();          // myList 现在为[7, 6, 5, 4, 3]
```

## 参见

*Array.sort()*

---

## Array.shift ()方法

删除数组的第一个元素

**有效性** Flash 5

**概要** *array.shift()*

## 返回

*array*的第一个元素的值，它也会被删除。

## 描述

*shift()*方法将删除数组的第一个元素，然后将数组中剩下的所有元素往前移一位。数组的 *length* 被减 1。注意，*shift()*和*pop()*方法不同，*pop()*删除的是数组中的最后一个元素。

## 示例

```
myList = new Array("a", "b", "c");
myList.shift(); // myList 现在是["b", "c"]
```

## 参见

*Array.pop()*, *Array.splice()*, *Array.unshift()*, 第十一章

---

## Array.slice()方法

用现有数组元素的一个子集来创建一个新的数组

**有效性** Flash 5

**概要** `array.slice(startIndex, endIndex)`

### 参数

***startIndex*** 一个从 0 开始的整数，用来指定 *array* 中第一个要添加到新数组中的元素。如果这个数是负的，*startIndex* 就表示从 *array* 末尾开始计算的元素号码（-1 是最后一个元素，-2 是倒数第二个元素，等等）。

***endIndex*** 一个整数，用来指定 *array* 中最后添加到新数组中的元素后面的一个元素。如果这个数是负的，那么 *endIndex* 就从 *array* 的末尾开始计算（-1 是最后一个元素，-2 是倒数第二个元素，等等）。如果被忽略，*endIndex* 就默认为 *array.length*。

### 返回

一个包含 *array* 中从 *startIndex* 到 *endIndex-1* 的元素的新数组。

## 描述

*slice()*方法创建一个新数组，它的创建方法是从一个已经存在的数组中提取一系列的元素。新数组是原来的 *array* 元素的一个子集，从 *array[startIndex]* 开始，到 *array[endIndex - 1]* 结束。

## 示例

```
myList = new Array("a", "b", "c", "d", "e");
```

```
// 将myOtherList 设置为["b", "c", "d"]
myOtherList = myList.slice(1, 4);

// 将anotherList 设置为["d", "e"]
anotherList = myList.slice(3);

// 将yetAnotherList 设置为["c", "d"]
yetAnotherList = myList.slice(-3, -1);
```

## 参见

*Array.splice()*, 第十一章, 第五章

---

## Array.sort()方法

将数组元素进行排序

**有效性** Flash 5

**概要** `array.sort()`

`array.sort(compareFunction)`

**参数**

*compareFunction* 一个指定如何对 *array* 进行排序的函数。

**描述**

调用的时候如果不使用参数, *sort()*方法就将 *array* 的元素临时转换为串, 按照这些串的编码点来对元素进行排序 (和字母顺序大体相同)。字母顺序的比较和代码点在第四章中有所描述。

如果调用的时候传递了参数 *compareFunction*, 那么 *sort()*就按照 *compareFunction* 的返回值来对数组元素进行重新排序, 这个函数是用户自定义的, 表示如何排列数组中的任意两个值。用户自定义的 *compareFunction*必须接受两个数组元素作为参数。如果第一个元素排在第二个元素前面, 它应该返回一个负数; 如果第一个元素要排在第二个元素后面, 它应该返回一个正数; 如果元素的次序不改变, 它就返回 0。如果在排序完成后又有新的元素添加到数组, 它们不会自动被排序, 必须对数组进行重新排序。注意, 数字的排序默认是按照它们的 Latin1 编码点来进行的。第十一章说明了如何按照数字值进行排序。

**示例**

下面的例子按照影片剪辑在屏幕上的水平位置进行排序:

```
var clips = [clip1, clip2, clip3, clip4];

function compareXposition(element1, element2) {
```

```

    if (element1._x < element2._x) {
        return -1;
    } else if (element1._x > element2._x) {
        return 1;
    } else {
        return 0; // 剪辑有相同的 x 位置
    }
}

clips.sort(compareXposition);

```

**参见**

*Array.reverse()*, 第十一章

**Array.splice()方法**

从数组中删除或者添加元素

**有效性** Flash 5

**概要**

```

array.splice(startIndex)
array.splice(startIndex, deleteCount)
array.splice(startIndex, deleteCount, value1, ...valuem)

```

**参数**

*startIndex* 一个从 0 开始的元素索引，从这里开始删除元素，或者插入新元素。如果是负数，*startIndex* 就表示从 *array* 后面开始计算（-1 是最后一个元素，-2 是倒数第二个元素，等等）。

*deleteCount* 一个可选的非负整数，它表示要从 *array* 中删除的元素个数，包括 *startIndex* 所表示的元素。如果为 0，则没有元素被删除。如果被遗漏，那么从 *startIndex* 开始到数组最后的元素都将被删除。

*value1,...valuem* 一个或多个值的可选列表，这些值在指定元素被删除后可以被添加到 *array* 中索引 *startIndex* 开始的地方。

**返回**

一个新的数组，其中包括被删除的元素（原来的 *array* 单独被修改，以反映所需要的变化）。

**描述**

*splice()* 方法删除从 *array*[*startIndex*] 到 *array*[*startIndex+deleteCount-1*] 的元素，然后可选地从 *startIndex* 开始的地方插入新元素。*splice()* 方法在 *array* 中不留间隙，它将元素上移或者下移，以确保数组元素的邻接。

## 示例

```
myList = new Array (1, 2, 3, 4, 5);
// 从列表中删除第2和第3个元素，然后在这个位置上插入元素“x”，“y”和“z”
// 这将把myList 修改为[1, "x", "y", "z", 4, 5]
myList.splice(2, 2, "x", "y", "z");
```

## 参见

[Array.slice\(\)](#), 第十一章

---

## Array.toString()方法

将数组转换为串，元素值之间用逗号分隔

**有效性** Flash 5

**概要** array.toString()

### 返回

一个由逗号分隔的转换为串的 array 元素的列表。

### 描述

*toString()*方法创建一个串表示 array。*toString()*所返回的是一个转换为串的数组元素的列表，中间由逗号分隔（与调用*join()*方法时不使用参数所返回的形式一样）。数组的*toString()*方法只要数组使用在串语境中都会被自动调用。因此，几乎没有必要在数组上手动执行*toString()*。通常，当需要数组的精确的串表示形式时，可使用*join()*方法，它可以更好地控制所创建的数组。

## 示例

```
myList = new Array("a", "b", "c");           // 创建一个数组
trace(myList.toString());                     // 显示: "a", "b", "c"
myList = new Array([1, 2, 3], "a", "b", "c"); // 创建一个嵌套的数组
trace(myList.toString());                     // 显示: "1,2,3,a,b,c"
```

## 参见

[Array.join\(\)](#)

---

## Array.unshift()方法

从数组的开头添加或删除元素

**有效性** Flash 5

**概要** array.unshift(value1, value2, ..., valuen)

### 参数

*value1, ..., valuen* 一个或多个值的可选列表，这些值会被添加到 array 的开头。

**返回**

*array* 的新 `length`。

**描述**

`unshift()` 方法在数组的开头添加一系列新的元素。元素的添加顺序和提供顺序相同。要在数组末尾添加元素，使用 `push()`。

**示例**

```
myList = new Array(5, 6);
myList.unshift(4);           // myList 现在为 [4, 5, 6]
myList.unshift(7, 1);        // myList 变成了 [7, 1, 4, 5, 6]
```

**参见**

`Array.push()`, `Array.shift()`, 第十一章

---

**Boolean()全局函数**

将一个值转换为布尔类型

**有效性** Flash 5

**概要** `Boolean(value)`

**参数**

*value* 一个包含要被转换为布尔值的表达式。

**返回**

将 *value* 转换为原始 Boolean 值的结果 (`true` 或者 `false`)。

**描述**

`Boolean()` 全局函数将它的参数转换为原始布尔值，并返回转换后的结果。将多种数据类型转换为原始布尔值的结果如表 3-3 所示。通常不需要使用 `Boolean()` 函数，ActionScript 会在必要的时候自动将值转换为布尔类型。

**用法**

要确保不把全局函数 `Boolean()` 同 `Boolean` 类构造器混淆起来。`Boolean()` 函数将它的参数转换为 `boolean` 类型，但是 `Boolean` 类构造器创建一个新的布尔对象。注意，在 ECMA-262 中，所有的非空串都转换为 `true`。在 Flash 5 中，只有能转换为合法非 0 数字的串才能转换为 `true`。因此，即使将串“`true`”转换为布尔的时候也会得到 `false`。

**示例**

```
var x = 1;
```

```
if (Boolean(x)) {  
    trace('x is true');  
}
```

## 参见

[Boolean 类, 第三章](#)

## Boolean 类

原始 Boolean 资料的包装类

**有效性** Flash 5

**概要** new Boolean(value)

### 参数

*value* 要处理的表达式，如果需要就转换为布尔值，然后装到 Boolean 对象中。

### 方法

*toString()* 将 Boolean 对象的值转换为串。

*valueOf()* 获取 Boolean 对象的原始值。

### 描述

*Boolean* 类创建一个 *Boolean* 对象，它在一个不能访问的内部属性中包含一个原始布尔值。*Boolean* 对象只用 *Boolean* 类的方法来操作和检查原始布尔值。*Boolean* 对象因此被称为包装对象，因为它只是简单地包装一个原始布尔值，然后给它一些对象类型的方法。将布尔类和 *String* 以及 *Number* 类进行比较，后两者类似于包装串和数字原始值，但是它们的用途更为广泛。

在很大程度上，*Boolean* 对象是内部使用的。在原始布尔值上调用方法的时候，解释程序会自动创建 *Boolean* 对象，在使用完之后又自动将其删除。我们自己可以通过 *Boolean* 构造器来创建 *Boolean* 对象，但是没有必要这么做。

### 用法

注意，在实践中，将 *Boolean()* 全局函数用作数据类型转换工具比使用 *Boolean* 类更为常见。

## 参见

[Boolean\(\) 全局函数, 第四章](#)

---

**Boolean.toString()方法** 被转换为串的 Boolean 对象值**有效性** Flash 5**概要** `booleanObject.toString()`**返回**

如果 `booleanObject` 的原始值为 `true` 就返回 “`true`”，如果 `booleanObject` 的原始值为 `false` 就返回 “`false`”。布尔对象的值是在对象得到构建，并存储在内部之后被设定的。虽然不能访问布尔对象的内部值，但是可以使用 `toString()` 来将它转换为等价的串。

**描述**

`toString()` 方法获取布尔对象的原始值，并将这个值转换为串，返回结果串。

**示例**

```
x = new Boolean(true);
trace(x.toString()); // 显示: "true"
```

**参见**

*Object.toString()*

---

**Boolean.valueOf()方法** 布尔对象的原始值**有效性** Flash 5**概要** `booleanObject.valueOf()`**返回**

如果 `booleanObject` 的原始值为 `true` 就返回 “`true`”，如果 `booleanObject` 的原始值为 `false` 就返回 “`false`”。布尔对象的值是在对象得到构建，并存储在内部之后被设定的。

**描述**

`valueOf()` 方法返回和一个布尔对象对应的原始布尔值。虽然不可以访问布尔对象的内部值，但是可以用 `valueOf()` 来将它转换为等价的原始值。

**示例**

```
x = new Boolean(0);
trace(x.valueOf()); // 显示: false
```

**参见**

*Object.valueOf()*

**Call()全局函数**

执行远程帧的脚本

**有效性** Flash 4, 在 Flash 5 中不支持

**概要**

```
call(frameLabel)
call(frameNumber)
```

**参数**

*frameLabel* 一个串, 表示其脚本要被执行的帧的标签。

*frameNumber* 其脚本要被执行的帧的号码。

**描述**

*call()* 函数执行属于 *frameLabel* 或者 *frameNumber* 的帧的脚本。例如, 下面的代码将运行当前时间线的第 20 帧的脚本:

```
call(20);
```

在 Flash 4 中, *call()* 用来创建一种原始的可重复使用的子程序 (不能接收参数或者返回值)。在 Flash 5 中, 会首选使用函数语句。

注意, 在 Flash 5 中, 当一个脚本通过 *call()* 得到远程执行的时候, 任何用 *var* 关键词声明的变量都被当作该次执行的局部变量, 在执行结束后就死亡。在远程执行脚本中创建非局部变量, 不使用 *var* 关键词:

```
var x = 10;           // 局部变量, 在脚本结束之后死亡
x = 10;             // 时间线变量, 在脚本结束后依然存在
```

要在当前时间线以外的帧上调用 *call()*, 使用 *tellTarget()* 函数。下面的例子执行 box 剪辑上第 10 帧的脚本:

```
tellTarget ('box') {
    call(10);
}
```

**参见**

第九章, 附录三

---

Color 类	控制影片剪辑的颜色值
<b>有效性</b>	Flash 5
<b>概要</b>	<code>new Color(target)</code>
<b>参数</b>	
<i>target</i>	一个串或者引用，表示到影片剪辑的路径或者其颜色要由新对象来控制的文件层级（使用在串语境中的时候，引用会被转换为路径）。
<b>方法</b>	
<i>getRGB()</i>	获取 Red, Green 和 Blue 的当前偏移量值。
<i>getTransform()</i>	获取 Red, Green, Blue 和 Alpha 的当前偏移量和百分比值。
<i>setRGB()</i>	为 Red, Green 和 Blue 赋新的偏移量值，将百分比设置为 0。
<i>setTransform()</i>	为 Red, Green, Blue 和 Alpha 赋新的偏移量，以及 / 或者百分比值。

### 描述

使用 *Color* 类可程序化地规定影片剪辑或者主影片的颜色和透明值。一旦对特定的 *target* 创建了一个 *Color* 类的对象，就可以调用这个对象的方法来影响它的 *target* 的颜色和透明度。例如，假设有一个剪辑实例 *ball*，我们想让它变成红色。我们首先建立一个 *target* 为 *ball* 的 *Color* 对象，并将它存储在变量 *ballColor* 中。然后，使用 *ballColor.setRGB()* 将 *ball* 设置为红色，如下所示：

```
var ballColor = new Color("ball");
ballColor.setRGB(0xFF0000); // 为 setRGB() 传递表示红色的十六进制值
```

这个例子提供了对简单应用程序的颜色控制。但是，要想控制更为复杂的情况，我们需要知道 Flash 中颜色表示的更多内容。影片剪辑中所显示的每个单个颜色都是由四个单独的部分来定义的：Red, Green, Blue 和 Alpha（或者透明度）。这四个部分按照不同的数量合并起来，生成我们在屏幕上所看到的每一种颜色。给定颜色中 Red, Green, Blue 和 Alpha 的数量描述为 0 到 255 之间的一个数字。Red, Green 或者 Blue 这些值中的较高者将对最终生成的颜色起较大的作用。但是，计算机颜色是加色系统，而不像绘画那样是减色的，因此，较高的值就会显得更亮，而不是更暗。如果它们全部等于 0，那么结果就是黑色；如果它们全部都是 255，那么结果就是白色。Alpha 的值越高，最终颜色就越不透明。（Alpha 为 0 的颜色是完全透明的，Alpha 为 255 的颜色则是完全不透明的。）

例如，单纯的红色会被描述为下面的值：

Red:255, Green:0, Blue:0, Alpha:255

但是部分透明的红色则为：

Red:255, Green:0, Blue:0, Alpha:130

为了便于讨论，我们在说到颜色值的时候采用所谓的RGB三原色符号(*Red*, *Green*, *Blue*)。虽然ActionScript不支持十进制的三原色，比如(255, 0, 255)，但是它支持十六进制的等价形式0xRRGGBB。*RR*, *GG*和*BB*是两位的十六进制数，表示*Red*, *Green*和*Blue*。我们也采用*RGBA*四成分符号(*Red*, *Green*, *Blue*, *Alpha*)以方便下面的讨论。

影片剪辑中最初的每个颜色的*Red*, *Green*, *Blue*和*Alpha*值是在Flash制作工具中用Mixer(混合器)面板来设置的。(在Mixer面板中，*Alpha*显示为一个百分数，而不是从0~255之间的数字。)要通过ActionScript改变影片剪辑中的所有颜色，我们要对剪辑颜色的*Red*, *Green*, *Blue*和*Alpha*成分进行全体的调整(称为变换)。

有两种方法可为每个颜色成分设置变换：

- 可以设置成分原始值的一个百分比值，也就是-100到100之间的一个数字。例如，我们可以说：“将剪辑中使用的所有红色设置为初始值的80%”。
- 可以为成分的初始值指定一个偏移量(*offset*)。这个偏移量是介于-255到255之间的一个数字。例如，我们可以说：“将剪辑中所有蓝色的值加上20”，使用负数的时候我们可以说：“将剪辑中所有蓝色的值减去20”。

变换剪辑中最终的颜色值是由初始(制作时间中的)颜色成份值，以及通过Color对象所设置的变换百分比和偏移量共同决定的。如下所示：

```
R = originalRedValue * (redTransformPercentage/100) + redTransformOffset  
G = originalGreenValue * (GreenTransformPercentage/100) + GreenTransformOffset  
B = originalBlueValue * (BlueTransformPercentage/100) + BlueTransformOffset  
A = originalAlphaValue * (AlphaTransformPercentage/100) + AlphaTransformOffset
```

如果在ActionScript中没有进行变换，每个成分最初的变换百分比的默认值就是100，而最初偏移量的默认值为0。

我们来看看颜色变换的实际例子。假设一个剪辑包含一个不透明的红色三角形(R:255, G:0, B:0, A:255)以及一个不透明的绿色圆形(R:0, G:255, B:0, A:255)。现在我们进一步假设对剪辑提供一个全体性的转换：将*Green*的百分比设置为50，将

Alpha的百分比设置为80, Blue的偏移量设置为100, 但是其他的偏移量和百分比保持默认值(0或100)。下面是这个全体性的转换对红色三角形的影响:

```
R -= 255 * (100/100) + 0 == 255      // 红色没有变化  
G -= 0 * (50/100) + 0 == 0          // 绿色减少50%  
B -= 0 * (100/100) + 100 == 100     // 蓝色偏移100  
A -= 255 * (80/100) + 0 == 204      // Alpha减少到80%
```

最后, 红色三角形变换得到的颜色值(R:255, G:0, B:100, A:204)。现在我们来看看对绿色圆形的影响:

```
R -= 0 * (100/100) + 0 == 0      // 红色没有变化  
G -= 255 * (50/100) + 0 == 127.5 // 绿色减少50%  
B -= 0 * (100/100) + 100 == 100   // 蓝色偏移100  
A -= 255 * (80/100) + 0 == 204   // Alpha减少到80%
```

最终, 绿色圆形的变换结果得到颜色值(R:0, G:127.5, B:100, A:204)。

要将我们假定的变换应用于实际的剪辑, 可使用前边所学的 *Color* 对象。要设置剪辑的全体颜色偏移量以及百分比值, 我们使用 *setRGB()* 或者 *setTransform()* 方法(参见这些方法条目中的示例代码)。相反, 要检查剪辑的当前颜色变换, 我们使用 *getRGB()* 和 *getTransform()* 方法。*Color* 类方法可以产生动态的颜色效果, 比如淡入、淡出, 以及着色。此外, 因为我们可以将着色应用在单个的剪辑实例上, *Color* 类为创建不同的图形提供了一个非常有效的省力的方法。例如, 我们可以从一个影片剪辑来创建满屏的气球, 它们的颜色千变万化, 就如后面的例子所示。

## 用法

一些 *Color* 类的有趣之处:

- 用 *Color* 对象来改变影片剪辑的颜色会将影片剪辑置于 ActionScript 的控制之下, 也就是说, 一些在制作时间提供给剪辑的行为将会被运行所终止。
- 设置剪辑的 *\_alpha* 属性会影响剪辑的 Alpha 百分比, 就像用 *getTransform()* 返回的对象的 *aa* 属性所表现出来的那样。
- 颜色变换不影响影片剪辑或者影片的背景颜色。它们只应用于场景上所放置的单纯形状。
- 手动的颜色变换可以在制作工具中通过 Effect(效果)面板(Window → Panel → Effect)而应用到影片剪辑上。所有这样的变换都会从 *getTransform()* 所返回的对象属性中反映出来。效果面板在制作影片的时候是一个浏览和选择颜色变换的有效工具。

## 示例

第一个例子显示如何基于现有的剪辑 balloon 而生成一系列随机的彩色气球影片剪辑：

```
// 循环建立剪辑 balloon 的 20 个副本
for (var i = 0; i < 20; i++) {
    // 复制这个气球
    balloon.duplicateMovieClip('balloon' + i, i);

    // 将气球放置在场景上
    this['balloon' + i]._x = Math.floor(Math.random() * 550);
    this['balloon' + i]._y = Math.floor(Math.random() * 400);

    // 为这个气球创建一个 Color 对象
    balloonColor = new Color(this['balloon' + i]);
    // 使用 setRGB() 方法随机给气球的颜色赋值
    balloonColor.setRGB(Math.floor(Math.random() * 0xFFFFFF));
}
```

通过将 Red, Green, Blue 的偏移量设置为相同的值，可以有效地使影片剪辑增亮或者变暗。例如，下面的代码将把 myClip 变暗：

```
brightness = new Color('myClip');
brightnessTransform = new Object();
brightnessTransform.rb = -30;
brightnessTransform.bb = -30;
brightnessTransform.gb = -30;
brightness.setTransform(brightnessTransform);
```

最后一个例子包括的代码将按照鼠标的位置使剪辑增亮或者变暗：

```
onClipEvent (load) {
    var brightness = new Color(this);
    var brightnessTransform = new Object();
    var stageWidth = 550;
}

onClipEvent (mouseMove) {
    brightnessAmount = -255 + (_root._xmouse / stageWidth) * 510;
    brightnessTransform.rb = brightnessAmount;
    brightnessTransform.bb = brightnessAmount;
    brightnessTransform.gb = brightnessAmount;
    brightness.setTransform(brightnessTransform);
    updateAfterEvent();
}
```

**Color.getRGB()方法**

获取 Red, Green 和 Blue 的当前偏移量

**有效性** Flash 5**概要** colorObj.getRGB()**返回**

一个表示 colorObj 目标的当前 RGB 偏移量的数字。

**描述**

*getRGB()*方法返回的数字在 -16777215 到 16777215 之间，表示剪辑中 Red, Green 和 Blue 成分的当前颜色偏移量。要获取颜色的百分比，必须使用 *getTransform()*。因为颜色偏移量值通常是从 0 到 255，它将 *getRGB()* 的返回值处理为十六进制比较方便，每一个颜色偏移量都可以用两位的十六进制数来表示。要对 *getRGB()* 返回的数字进行编码，我们将它作为 0xRRGGBB 形式的六位十六进制数，RR 表示 Red 的偏移量，GG 表示 Green 的偏移量，而 BB 表示 Blue 的偏移量。例如，如果 *getRGB()* 返回数字 10092339，我们就将它转换为十六进制数 0x99FF33，从而得到颜色偏移量 R:153, G:255, B:51 的；如果 *getRGB()* 返回数字 255，我们可以将它转换为十六进制数 0x0000FF，从而得到颜色偏移量 R:0, G:0, B:255。*getRGB()* 返回的值可以用 *toString()* 转换为十六进制的，如下所示：

```
// 创建一个 Color 对象
myColor = new Color("myClip");
// 将 Red 偏移量设置为 255 (十六进制中的 FF)
myColor.setRGB(0xFF0000);
// 获取 RGB 偏移量，并把它转化为十六进制数
hexColor = myColor.getRGB().toString(16);
trace(hexColor); // 显示: ff0000
```

大部分 Web 开发者对于十六进制颜色值都很熟悉，因为它们经常用在 HTML 标签中。例如，现在有一个十六进制数字，它表示页面的背景颜色（全值的红色和蓝色形成桃红色）：

```
<BODY BGCOLOR="#FF00FF">
```

用在 HTML 标签中的十六进制颜色形式实际上和 *getRGB()* 以及 *setRGB()* 中所使用的格式是相同的。但是，不强制使用十六进制数来解释 *getRGB()* 的返回值。我们也可以从 *getRGB()* 的返回值中，使用下面的位逻辑操作符来抽取单独的 Red, Green 和 Blue 颜色偏移量：

```
var rgb = myColorObject.getRGB();
var red = (rgb >> 16) & 0xFF; // 将单独的 Red 偏移量赋给 red
```

```
var green = (rgb >> 8) & 0xFF; // 将单独的 Green 偏移量赋给 Green  
var blue = rgb & 0xFF; // 将单独的 Blue 偏移量赋给 Blue
```

将偏移量值分别存储到单独的变量中后，我们可以将它们当作十进制数字来分别进行检查和操作。但是，当我们想要对 *Color* 对象应用任何偏移量值的变化，我们必须重新将偏移量放入一个单独的数字中，如同 *setRGB()* 方法中的示范。

### 用法

当我们要直接为剪辑赋予新的颜色而不涉及剪辑的最初颜色值时使用 *getRGB()* 和 *setRGB()* 方法非常方便。但是，*getTransform()* 和 *setTransform()* 方法更适合于修改与剪辑最初颜色相关的剪辑颜色变换的 RGB 成分。

颜色偏移量通常可以用 *getTransform()* 读出，它返回单独的每一个成分，而不像 *setRGB()* 那样整个地返回。当我们用 *setTransform()* 来设置负偏移量的时候更是如此，这是由于负数在二进制中的表示方法的原因。

### 示例

```
// 为剪辑 box 创建一个新的 Color 对象  
boxColor = new Color("box");  
// 为 "box" 设置一个新的 RGB 偏移量  
boxColor.setRGB(0x333366);  
// 检查为 "box" 所设置的 RGB 偏移量  
trace(boxColor.getRGB()); // 显示: 3355494
```

### 参见

*Color.getTransform()*, *Color.setRGB()*

---

## Color.getTransform()方法

获取剪辑的 Red, Green, Blue 和 Alpha 成分的当前偏移量和百分比

**有效性** Flash 5

**概要** *colorObj.getTransform()*

### 返回

返回一个对象，它的属性包含 *colorObj* 的目标剪辑的颜色变换值。

### 描述

*getTransform()* 方法返回一个函数，它的属性可以告诉我们对于 *Color* 对象的目标来说，当前应用的是什么变换。返回对象的属性名称和值如表 R-4 所示。

表 R-4 getTransform()返回对象的属性

属性名称	属性值	属性描述
ra	-100 到 100	Red 变换百分比
rb	-255 到 255	Red 偏移量
ga	-100 到 100	Green 变换百分比
gb	-255 到 255	Green 偏移量
ba	-100 到 100	Blue 变换百分比
bb	-255 到 255	Blue 偏移量
aa	-100 到 100	Alpha 变换百分比
ab	-255 到 255	Alpha 偏移量

## 用法

注意，在表 R-4 中，百分比和偏移量都可以是负数，但是，这些只是计算 RGB 颜色成分的一个因素，它总是在 0 到 255 之间变化。这个范围之外的值也在允许的范围之内。参见 *Color* 类的描述，它对用来确定最终 RGB 和 Alpha 颜色成分的计算作了说明。

## 示例

我们可以将 *getTransform()* 和 *setTransform()* 合并使用，来单独修改颜色变换中的 Red, Green, Blue 或者 Alpha 成分。例如，在下面的代码中，我们调整剪辑 box 中的 Red 和 Alpha 成分：

```
// 为剪辑 box 创建一个新的 Color 对象
boxColor = new Color('box');

// 将 getTransform() 的返回对象赋给 boxTransform
boxTransform = boxColor.getTransform();

// 现在，作一些变换来修改对象的属性
boxTransform.rb = 200; // 将 Red 偏移量设置为 200
boxTransform.aa = 60; // 将 Alpha 百分比设置为 60

// 将新的变换通过 boxColor 应用到 box
boxColor.setTransform(boxTransform);
```

## 参见

*Color.getTransform()*

**Color.setRGB ()方法**

为 Red, Green 和 blue 赋新的偏移量

**有效性**

Flash 5

**概要**

```
colorObj.setRGB(offset);
```

**参数***offset*

一个从 0 到 16777215 (0xFFFFFFF) 之间的数字，表示 *colorObj* 目标剪辑的新 RGB 偏移量。可以是十进制整数，也可以是十六进制整数。

允许范围之外的数字被转换为允许范围之内的数字（使用二进制补码符号）。因此，*setRGB()*不能用来设置负的偏移量值（而*setTransform()*可以）。

**描述**

*setRGB()*方法为影片剪辑的 RGB 成分设置新的变换偏移量。新的 *offset* 通常很容易设置为 0xRRGGBB 形式的六位十六进制数字，RR, GG 和 BB 是从 00 到 FF 的二位数字，表示 Red, Green 和 Blue 成分。例如，RGB 值 (51, 51, 102) 等价于十六进制值：

0x333366

因此，要为剪辑 *menu* 设置一个灰色的 RGB 偏移量，我们可以使用：

```
var menuColor = new Color("menu");
menuColor.setRGB(0x999999);
```

习惯于 HTML 中的六位十六进制颜色值的 Web 开发者在使用 *getRGB()* 时用前边所说的十六进制格式应该是很轻松的。十进制、八进制和十六进制数字的初级读本，可以参见 <http://www.moock.org/asdg/technotes>。

注意，除了设置偏移量之外，*setRGB()* 也可以自动将剪辑颜色变换的 Red, Green 和 Blue 百分比设置为 0，表示通过 *setRGB()* 来执行的颜色变化是作为直接的颜色赋值（而不是剪辑中对初始颜色的调整）。要调整影片剪辑中与剪辑初始颜色相关的颜色，必须使用 *setTransform()* 方法。

**示例**

下面是一个简单的技术，用来生成一个用于 *setRGB()* 方法的数字。我们自定义的函数 *combineRGB()* 将把 *red* 和 *green* 数字移动到一个 24 比特位数字中合适的位置，然后用位逻辑操作符 OR (|) 将 *red*, *green* 和 *blue* 合并起来。我们利用结果来为影片剪辑 *box* 赋一个颜色值：

```
function combineRGB (red, green, blue) {
    // 将颜色值合并到一个单独的数字中
    var RGB = (red<<16) | (green<<8) | blue;
```

```

        return RGB;
    }
    // 创建Color 对象
    var boxColor = new Color("box");
    // 将box的颜色设置为RGB值(201, 160, 21)
    boxColor.setRGB(combineRGB(201, 160, 21));
}

```

关于位逻辑操作的更多信息，请参见第十五章。

## 参见

*Color.getRGB()*, *Color.setTransform()*

## Color.setTransform ()方法

为 Red, Green, blue 和 Alpha 赋新的偏移量和 / 或百分比

**有效性** Flash 5

**概要** `colorObj.setTransform(transformObject)`

### 参数

*transformObject* 一个对象，它的属性包含为 *colorObj* 的目标剪辑而设定的新颜色变换值。

### 描述

*setTransform()* 方法让我们可以精确地控制影片剪辑颜色的 Red, Green, Blue 和 Alpha 成分的百分比和偏移量。要使用 *setTransform()*，必须首先创建一个包含一系列预置属性的对象。我们把要应用在 *Color* 对象上的变换用这些属性来表示，它们列在表 R-4 的 *getTransform()* 方法中（差别是，*setTransform()* 指定它们的值，而不读它们的值）。

一旦创建了具有表 R-4 中所描述的属性的对象，我们就将这个对象传递给 *setTransform()* 方法。*transformObject* 上的属性值变成了 *colorObj*（因此对应到 *colorObj* 的目标影片剪辑）的新百分比和偏移量变换值。这样，剪辑中最终的颜色值就按照 *Color* 类下所讨论的计算来决定。

要检查特定 *Color* 类当前的偏移量和百分比，我们使用 *getTransform()* 方法。

### 示例

```

// 为 box 剪辑创建一个新的Color 对象
boxColor = new Color("box");

// 创建一个新的匿名对象，将它存储在 boxTransform 中
boxTransform = new Object();

```

```
// 赋给 boxTransform 所需要的属性，按照要求设置我们的变换值  
boxTransform.ra = 50; // Red 百分比  
boxTransform.rb = 0; // Red 偏移量  
boxTransform.ga = 100; // Green 百分比  
boxTransform.gb = 25; // Green 偏移量  
boxTransform.ba = 100; // Blue 百分比  
boxTransform.bb = 0; // Blue 偏移量  
boxTransform.aa = 40; // Alpha 百分比  
boxTransform.ab = 0; // Alpha 偏移量  
  
// 既然我们的变换对象已经准备好了，我们就可以将它传递给 setTransform()  
boxColor.setTransform(boxTransform);
```

用上面的方法来创建一个变换对象相当麻烦。我们可以使用 *getTransform()* 方法更容易地生成一个新的变换对象，如下所示：

```
// 为 box 剪辑创建一个新的 Color 对象  
boxColor = new Color("box");  
  
// 在 boxColor 上调用 getTransform()，获取一个构造适宜的变换对象  
boxTransform = boxColor.getTransform();  
  
// 现在只改变 boxTransform 中需要的属性，其他保持不变  
boxTransform.rb = 51; // Red 偏移量  
boxTransform.aa = 40; // Alpha 百分比  
  
// 将变换对象和 setTransform() 一起使用  
boxColor.setTransform(boxTransform);
```

## 参见

*Color.getTransform()*

---

## Date()全局函数

一个表示当前日期和时间的串

**有效性** Flash 5

**概要** Date()

**返回**

一个包含当前日期和时间的串。

**描述**

*Date()* 函数返回一个人可以读得懂的串，表示与当地时区相关的日期和时间。这个串还包含 GMT 偏移量（本地时间和格林威治平均时间之间的小时差别）。

确保不要将全局函数 *Date()* 同 *Date()* 类构造器混淆起来。*Date()* 函数返回标准简明格式的串资料。它对于人来说是很方便的，但是在你需要对资料和时间进行操作的程序

中却不那么有用。因此，对你来说，使用 *Date* 类的对象会更好一些，它可以很方便地独立访问年份、月份、日期和时间。

### 示例

要用最简单的方法将当前的时间和日期放在文本域中，可以使用 *Date()* 函数，如下所示：

```
myTextField = Date();
// 将myTextField设置为下面格式的串:
// "Mon Aug 28 16:23:09 GMT-0400 2000"
```

### 参见

*Date* 类, *Date.UTC()*

---

## Date 类

当前时间以及对日期信息的结构化支持

**有效性** Flash 5

**概要**

```
new Date()
new Date(milliseconds)
new Date(year, month, day, hours, minutes, seconds, ms)
```

**参数**

*milliseconds* 从 UTC（世界调整时间，和 GMT 类似）1970 年 1 月 1 日午夜开始到新的日期之间的毫秒数。如果为正数时间就在此之后，如果为负数则在此之前。任何必要的当地时区调整都要在 UTC 时间确定之后才能进行。例如，指定一个 *milliseconds* 参数为东部标准时间的 1000，那么将创建的日期就是 UTC 时间 1970 年 1 月 1 日午夜之后的 1 秒钟，它将转换为东部标准时间的 1969 年 12 月 31 日 7:00:01 p.m. (UTC 时间的五个小时之前)。

*year* 一个整数，表示年份。使用 *year, ..., ms* 构造器格式的时候需要这个参数。如果 *year* 是一或两位数字，它就被当作从 1900 年开始的某个年份（例如，*year* 等于 11 就表示 1911 年，而不是 2011 年）。使用四位数可以表示 2000 年或之后的年份（例如，使用 2010，而不是 10）。三位数年份被当作公元 1000 年以前。注意，当 *year* 为负数或者小于 800 的时候，计算是不可靠的。要确定在公元 1000 年之前的日期，安全的做法是使用单独的 *milliseconds* 构造器格式。

<i>month</i>	一个整数，表示月份，从 0 (1月) 到 11 (12月)，而不是从 1 到 12。使用 <i>year, ... ms</i> 构造器格式的时候需要这个参数。超出范围的月份将转到上一年或者下一年。例如， <i>month</i> 等于 13 的时候，它就被当作下一年的 2 月来看待。
<i>day</i>	一个可选的整数，指定日期，从 1 到 31。如果没有设定，就默认为 1。超出范围的日期被转到下一个月或者上个月中。例如，9月 31 日就会被当作 10 月 1 日，9 月 0 日，被当作 8 月 31 日。
<i>hours</i>	一个可选的整数，表示小时数，从 0 (午夜) 到 23 (11 p.m.)。如果没有设置就默认为 0。AM 和 PM 符号不被支持。超出范围的小时数被转到前一天或者后一天。例如， <i>hour</i> 等于 25 就表示下一天的 1 a.m.。
<i>minutes</i>	一个可选的整数，表示分钟数，从 0 到 59。如果没有设置就默认为 0。超出范围的分钟数被转到上一小时或者下一小时。例如， <i>minute</i> 等于 60 就表示下一小时的第一分钟。
<i>seconds</i>	一个可选的整数，表示秒数，从 0 到 59。如果没有设置就默认为 0。超出范围的秒数被转到上一分钟或者下一分钟。例如， <i>second</i> 等于 -1 就表示上一分钟的最后一秒。
<i>ms</i>	一个可选的整数，表示毫秒数，从 0 到 999。如果没有设置就默认为 0。超出范围的毫秒数被转到上一秒或者下一秒钟。例如， <i>ms</i> 等于 1005 就表示下一秒钟的 5 毫秒。

### 类方法

*UTC()* 获取 1970 年 1 月 1 日到所提供的 UTC 日期之间的毫秒数。

### 方法

<i>getDate()</i>	获取月份中的天数，从 1 到 31。
<i>getDay()</i>	获取星期中的天数，是一个从 0 (星期天) 到 6 (星期六) 的数字。
<i>getFullYear()</i>	获取四位数字的年份。
<i>getHours()</i>	获取一天中的小时数，从 0 到 23。
<i>getMilliseconds()</i>	获取毫秒数。
<i>getMinutes()</i>	获取分钟数。
<i>getMonth()</i>	获取一年中的月份，是一个从 0 (1月) 到 11 (12月) 的数字。

- getSeconds()* 获取秒数。
- getTime()* 获取内部格式的日期（即从 1970 年 1 月 1 日到当前日期之间的毫秒数）。
- getTimezoneOffset()*
- 获取 UTC 和当地时间之间的分钟数。
- getUTCDate()* 获取 UTC 时间月份里已过的天数。
- getUTCDay()* 获取 UTC 时间星期里已过的天数。
- getUTCFullYear()* 获取 UTC 时间四位数字的年份。
- getUTCHours()* 获取 UTC 时间一天里已过的小时数。
- getUTCMilliseconds()*
- 获取 UTC 时间的毫秒数。
- getUTCMinutes()* 获取 UTC 时间的分钟数。
- getUTCMonth()* 获取 UTC 时间一年里的月份。
- getUTCSeconds()* 获取 UTC 时间的秒数。
- getYear()* 获取与 1900 年相关的年份。
- setDate()* 指定月份中的天数。
- setFullYear()* 指定四位数字格式的年份。
- setHours()* 指定一天中的小时数。
- setMilliseconds()* 指定毫秒数。
- setMinutes()* 指定分钟数。
- setMonth()* 指定一年中的月份。
- setSeconds()* 指定秒数。
- setTime()* 指定内部格式的日期（即从 1970 年 1 月 1 日到当前日期之间的毫秒数）。
- setUTCDate()* 指定 UTC 时间一个月中的天数。
- setUTCFullYear()* 指定 UTC 时间四位数字格式的年份。
- setUTCHours()* 指定 UTC 时间一天中的小时数。
- setUTCMilliseconds()*
- 指定 UTC 时间的毫秒数。

*setUTCMinutes()* 指定 UTC 时间的分钟数。

*setUTCMonth()* 指定 UTC 时间一年中的月份。

*setUTCSeconds()* 指定 UTC 时间的秒数。

*setYear()* 指定四位数字的年份或针对 20 世纪的二位数字格式。

*toString()* 表示日期的串，我们可以读懂。

*valueOf()* UTC 中从 1970 年 1 月 1 日到当前时间之间的毫秒数。

### 描述

我们将 *Date* 类的对象用作一种确定当前时间和日期的方法，它同时也是将任意的日期和时间存储在结构化格式中的一种途径。

在 ActionScript 中，一个特定的日期是由 1970 年 1 月 1 日午夜之前或之后的毫秒数来表示的。如果毫秒数是正的，那么要表示的日期就在 1970 年 1 月 1 日的午夜之后；如果毫秒数是负的，日期就在 1970 年 1 月 1 日之前。例如，如果一个日期值是 10000，那么所描述的日期就是 1970 年 1 月 1 日 12:00:10。如果一个日期值为 -10000，描述的日期就是 1969 年 12 月 31 日 11:59:50。

但是，我们通常不需要操心对特定的日期计算毫秒数，ActionScript 解释程序会帮我们处理。当构造一个 *Date* 对象之后，我们只将日期描述为年份、月份、日期、小时、分钟、秒和毫秒。然后，解释程序在一定的时候将这些值转换为内部的 1970 毫秒形式。我们也可以让解释程序用当前时间创建一个新的 *Date* 对象。一旦 *Date* 对象得到创建，我们就可以使用 *Date* 类的方法来获取和指定资料的年份、月份、日期、时间、分钟、秒和毫秒。

要建立一个新的 *Date* 对象有三种方法：

- 调用 *Date()* 构造器，不带参数。它将新的 *Date* 对象设置为当前的时间。
- 调用 *Date()* 构造器，带 1 个数字参数——从 1970 年 1 月 1 日午夜到所创建的日期之间的毫秒数。
- 调用 *Date()* 构造器，带 2 到 7 个数字参数，对应于所创建日期的年份和月份（强制的），以及日期、小时、分钟、秒和毫秒（可选的）。

因为数据在内部存储为一个单独的数字，*Date* 类的对象没有要获取和设置的属性。不过，我们可以用一些方法来访问以人们易懂的形式表示的各种数据（也就是说，以更为方便的单位）。例如，要确定特定 *Date* 对象的月份，我们使用：

```
myMonth = myDate.getMonth();
```

不能使用：

```
myMonth = myDate.month; // 没有这样的属性！应使用方法。
```

许多 *Date* 方法包含字母 UTC，它是世界调整时间的缩写形式。在很大程度上，UTC 时间和更加通用的格林威治标准时间 (GMT) 几乎是同义的，GMT 是在格林威治子午线上测量的。UTC 是一种简单的更新标准，比 GMT 更清楚，后者在历史上有好几种意思。UTC 方法让我们可以直接在宇宙坐标时间上处理时间，而不用在两种时区之间进行转换。等价的非 UTC 方法都基于本地的、调整的时区来生成值。

## 用法

所有的日期和时间都按照 Flash 影片所运行的操作系统（不是 Web 服务器）上的设置来确定，包括地方性的偏移量。因此，时间和日期的精确性和用户的系统一致。

注意，*Date()* 构造器也可以用作全局函数，生成一个表示当前时间的串，其格式和 *myDate.toString()* 所返回的一样。

在 Flash 5 ActionScript 中，不能将一个串转换为一个 *Date* 对象，但是在 JavaScript 中却可以。

## 示例

日期可以进行加和减，以赶上累积时间或逝去的时间。假设我们的朋友 Graham 决定出去游玩，时间不到一年。当他走的时候，我们想记录一下他走了多少天，以及多少天后才回来。下面的代码可以建立倒计时：

```
// 将当前的时间指定到 now 中
var now=new Date();

// Graham 是 2000 年 9 月 7 日离开的（记住，月份是从 0 开始的，
// 因此，9 月份被表示为 8，而不是 9）
var departs = new Date(2000, 8, 7);
// Graham 是 2001 年 8 月 15 日回来的
var return=new Date(2001, 7, 15)

// 将时间转换为毫秒数，便了比较。
// 然后检查在两个时间之间过了多少毫秒。
var gone = now.getTime() - departs.getTime();

// 通过一天中的毫秒数来区分离开时间和现在时间的差额。
// 它告诉我们已经等了多少天。
var numDaysGone = Math.floor(gone/86400000);

// 用同样的技术来确定我们还要等多少天
var left = return.getTime() - now.getTime();
var numDaysLeft = Math.floor(left/86400000);
```

```
// 在文本域中显示天数  
goneOutput = numDaysGone;  
leftOutput = numDayLeft;
```

当从现在的日期添加或者减去一天时，通常将一天中的毫秒数(86400000)赋给一个变量，以便使用。下面的代码对当前的日期添加一天：

```
oneDay = 86400000;  
now = new Date();  
tomorrow=new Date(now.getTime() + oneDay);  
// 我们也可以在now上添加一天，像这样：  
now.setTime(now.getTime() + oneDay);
```

要对一个日期应用自定义的格式，必须手动变换*getDate()*, *getDay()*, *getHours()*等的返回值，使其成为自定义的串，如下面的例子所示：

```
// 用任何Date 对象，返回一个这种格式的串：  
// Saturday December 16  
function formatDate(theDate) {  
    var months = ['January', 'February', 'March', 'April',  
                 'May', 'June', 'July', 'August', 'September',  
                 'October', 'November', 'December'];  
    var days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',  
               'Friday', 'Saturday'];  
    var dateString = days[theDate.getDay()] + " "  
                    + months[theDate.getMonth()] + " "  
                    + theDate.getDate();  
    return dateString;  
}  
  
now = new Date();  
trace("Today is " + formatDate(now));
```

下面的例子显示了如何将*getHours()*所返回的24小时时钟的值转换为用AM和PM表示的12小时时钟：

```
// 用任何Date 对象，返回如下格式的一个串：  
// "2:04PM"  
function formatTime(theDate) {  
    var hour = theDate.getHours();  
    var minute = theDate.getMinutes() > 9?  
                theDate.getMinutes() : "0" + theDate.getMinutes();  
    if (hour > 12) {  
        var timeString = (hour - 12) + ":" + minute + "PM";  
    } else {  
        var timeString = hour + ":" + minute + "AM";  
    }  
    return timeString;  
}  
  
now = new Date();  
trace("The time is " + formatTime(now));
```

关于用 *Date* 类来创建类似风格的时钟，可以参见第十三章。注意，对于基于时间的编程行为，比如影片中一个10秒钟的停顿，用全局函数 *getTimer()* 可以更容易地实现。

### 参见

*Date()*, *Date.UTC()*, *getTimer()*

---

## **Date.getDate()方法**

月份中的天数

**有效性** Flash 5

**概要** `date.getDate()`

**返回**

一个从 1 到 31 之间的整数，表示 *date* 中所指的月份中的天数。

---

## **Date.getDay()方法**

星期中的天数

**有效性** Flash 5

**概要** `date.getDay()`

**返回**

一个从 0(星期天) 到 6(星期六) 之间的整数，表示 *date* 中所指的星期中的天数。

**示例**

下面的代码将一个针对星期中的当前天数的.swf文件装载到影片剪辑 *welcomeHeader* 中(7个.swf文件按照 *sun.swf*, *mon.swf* 等这样的顺序来命名):

```
now = new Date();
today = now.getDay();
days = ["sun", "mon", "tue", "wed", "thu", "fri", "sat"];
welcomeHeader.loadMovie(days[today] + ".swf");
```

---

## **Date.getFullYear()方法**

用四位数来表示的年份

**有效性** Flash 5

**概要** `date.getFullYear()`

**返回**

一个四位的整数，表示 *date* 的年份，比如 1999 或者 2000。

---

**Date.getHours()方法** ·天中的小时数**有效性** Flash 5**概要** `date.getHours()`**返回**

一个从 0 (午夜) 到 23 (11 p.m.) 的整数，表示 `date` 所指的一天中的小时数。A.M. 和 P.M. 符号不被支持，但是可以手动构建，如 `Date` 类的示例中所示。

---

**Date.getMilliseconds()方法** 日期中的毫秒数**有效性** Flash 5**概要** `date.getMilliseconds()`**返回**

一个从 0 到 999 的整数，表示 `date` 的毫秒数。注意，它不表示从 1970 年开始的毫秒数（参见 `getTime()`），而是特定 `Date` 对象所表示的秒数的小数余数。

**参见***[Date.getTime\(\)](#)*

---

**Date.getMinutes()方法** 日期中的分钟数**有效性** Flash 5**概要** `date.getMinutes()`**返回**

一个从 0 到 59 的整数，表示 `date` 中小时里的分钟数。

---

**Date.getMonth()方法** 年中的月份**有效性** Flash 5**概要** `date.getMonth()`**返回**

一个从 0 (1 月) 到 11 (12 月) 的整数，而不是从 1 到 12，它表示 `date` 中年份里的月份。

## 用法

小心，不要以为 1 表示 1 月份！*getMonth()*的返回值是从 0 而不是从 1 开始的。

## 示例

下面，我们将 *getMonth()*所返回的数字转换为大家可以看得懂的缩写形式：

```
var months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sept", "Oct", "Nov", "Dec"];
myDateObj = new Date();
trace ('The month is ' + months[myDateObj.getMonth()]);
```

## Date.getSeconds()方法

日期中的秒数

**有效性** Flash 5

**概要** `date.getSeconds()`

### 返回

一个从 0 到 59 的整数，表示 *date* 中分钟里的秒数。

## Date.getTime()方法

获取从 1970 年 1 月 1 日到 *date* 对象当前时间之间的毫秒数

**有效性** Flash 5

**概要** `date.getTime()`

### 返回

一个整数，表示从 1970 年 1 月 1 日午夜开始到 *date* 当前时间之间的毫秒数。如果当前时间在 1970 年 1 月 1 日之后，就是正数，在其之前就是负数。

## 描述

在内部，所有的日期都被表示为一个单独的数字——1970 年 1 月 1 日午夜到 *date* 当前时间之间的毫秒数。*getTime()*方法让我们可以访问这个毫秒数，这样，我们就可以用它来构造其他的日期，或者在两个日期之间比较逝去的时间。

## 示例

假设我们将下面的代码放在影片的第 10 帧上：

```
time1 = new Date();
```

然后，我们将下面的代码放在影片的第 20 帧上：

```
time2 = new Date();
```

使用下面的代码，可以确定影片中从第 10 帧到第 20 帧所用时间的毫秒数：

```
elapsedTime = time2.getTime() - time1.getTime();
```

注意，Flash 的全局函数 *getTimer()* 也可以访问影片中的逝去的时间。

## 参见

*Date.setTime()*, *getTimer()*

---

## Date.getTimezoneOffset()方法

当地时间和 UTC（也就是  
GMT）之间的分钟数

**有效性** Flash 5

**概要** `date.getTimezoneOffset()`

### 返回

一个整数，表示本地时区和实际的 UTC（格林威治子午线）时间之间的分钟数。如果本地时间在 UTC 之后则为正数；如果本地时间在 UTC 之前则为负数。包括依靠年中的天数而进行调节的夏令时。

### 示例

在东部夏季时间 (EDT) 的夏令时中调用的时候，下面的代码将返回值 240 (240 分钟即 4 个小时)：

```
myDate = new Date();
trace(myDate.getTimezoneOffset());           // 显示: 240
```

但是，在 EDT 的非夏令时时间中调用的时候，同样的代码将返回 300(300 分钟即 5 个小时)，它是 EST 和 UTC 之间真正的偏移量。

---

## Date.getUTCDate()方法

月份中的天数 (UTC 时间)

**有效性** Flash 5

**概要** `date.getUTCDate()`

### 返回

一个从 1 到 31 的整数，表示 *date* 中月份里的天数，*date* 是 UTC 时间。

---

**Date.getUTCDay()方法** 星期中的天数 (UTC 时间)**有效性** Flash 5**概要** `date.getUTCDay()`**返回**

一个从0(星期天)到6(星期六)的整数, 表示`date`中星期里的天数, `date`是UTC时间。

---

---

**Date.getUTCFullYear()方法** 四位数字的年份 (UTC 时间)**有效性** Flash 5**概要** `date.getUTCFullYear()`**返回**

一个四位的整数, 表示`date`的年份, `date`是UTC时间, 例如1999或者2000。

---

---

**Date.getUTCHours()方法** 一天中的小时数 (UTC 时间)**有效性** Flash 5**概要** `date.getUTCHours()`**返回**

一个从0(午夜)到23(11 p.m.)的整数, 表示`date`中一天里的小时数, `date`为UTC时间。

---

---

**Date.getUTCMilliseconds()方法** 日期中的毫秒数 (UTC 时间)**有效性** Flash 5**概要** `date.getUTCMilliseconds()`**返回**

一个从0到999的整数, 表示`date`的毫秒数, `date`为UTC时间。

---

---

**Date.getUTCMinutes()方法** 日期中的分钟数 (UTC 时间)**有效性** Flash 5

**概要**            `date.getUTCMinutes()`

**返回**

一个从 0 到 59 的整数，表示 `date` 的分钟数，`date` 为 UTC 时间。

---

### Date.getUTCMonth()方法

年中的月份（UTC 时间）

**有效性**        Flash 5

**概要**            `date.getUTCMonth()`

**返回**

一个从 0 (1 月) 到 11 (12 月) 的整数，而不是从 1 到 12，表示 `date` 中年份中的月份，`date` 为 UTC 时间。

**用法**

小心不要将 1 当做是 1 月份！ `getUTCMonth()` 返回的值是从 0 而不是从 1 开始的。

---

### Date.getUTCSeconds()方法

日期中的秒数（UTC 时间）

**有效性**        Flash 5

**概要**            `date.getUTCSeconds()`

**返回**

一个从 0 到 59 的整数，表示 `date` 中分钟里的秒数，`date` 是 UTC 时间。

---

### Date.getYear()方法

年份，和 1900 年相关

**有效性**        Flash 5

**概要**            `date.getYear()`

**返回**

`date.getFullYear()`-1900 的值。例如，1999 的 `getYear()` 值为 99，2001 的 `getYear()` 值为 101，而 1800 的 `getYear()` 值为 -100。这个函数对 20 世纪的日期来说最为有用。

---

### Date.setDate()方法

设置月份中的天数

**有效性**        Flash 5

**概要**`date.setDate(day)`**参数***day*

一个从 1 到 31 的整数，表示 *date* 中月份里的新天数。如果你指定的天数比当前月中总的天数大，月份就会相应增长。例如，如果当前的 *month* 为 8 (九月份)，你为新的 *day* 指定了 31，那么就会得到 10 月 1 日。*day* 会变成 1，而 *month* 会变成 9 (10 月)。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**[Date.getDate\(\)](#)**Date.setFullYear()方法**

指定四位数字的世纪和年份

**有效性**

Flash 5

**概要**`date.setFullYear(year, month, day)`**参数***year*

一个四位的整数，表示 *date* 的新年份，例如 1999 或者 2000。

*month*

一个可选的整数，从 0 (1 月) 到 11 (12 月)，而不是从 1 到 12，它表示 *date* 年份中新的月份。如果没有设置就默认为 0。

*day*

一个可选的整数，从 1 到 31，表示 *date* 的月份中的新日期。如果没有设置就默认为 1。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**[Date.setYear\(\)](#), [Date.getFullYear\(\)](#)**Date.setHours()方法**

指定一天中的小时数

**有效性**

Flash 5

**概要**`date.setHours(hour)`

**参数**

*hour* 一个从 0 (午夜) 到 23 (11 p.m.) 的整数，指定 *date* 中日期中的新的小时数。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**

*Date.getHours()*

---

**Date.setMilliseconds()方法**

指定日期的毫秒数

**有效性** Flash 5

**概要** *date.setMilliseconds(ms)*

**参数**

*ms* 一个从 0 到 999 的整数，表示 *date* 的新的毫秒数，而不是从 1970 年开始的毫秒数。大于 999 以及小于 0 的值被转换到 *date* 中的秒数上。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**

*Date.getMilliseconds()*, *Date.setTime()*

---

**Date.setMinutes()方法**

指定日期中的分钟数

**有效性** Flash 5

**概要** *date.setMinutes(minute)*

**参数**

*minute* 一个从 0 到 59 的整数，表示 *date* 的小时中的新分钟数。大于 59 的和小于 0 的值被转换到 *date* 的小时数上。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**

*Date.getMinutes()*

---

**Date.setMonth()方法**

指定年份中的月份

**有效性** Flash 5

**概要** `date.setMonth(month)`

**参数**

*month* 一个从 0 (1 月) 到 11 (12 月) 的整数，而不是从 1 到 12，表示 *date* 中年份里的新月份。大于 11 或者小于 0 的值将会被转换到 *date* 的年份上。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**用法**

注意不要将 1 当做 1 月份！月份是从 0 而不是从 1 开始的。

**参见**

*Date.getMonth()*

---

**Date.setSeconds()方法**

指定日期中的秒数

**有效性** Flash 5

**概要** `date.setSeconds(second)`

**参数**

*second* 一个从 0 到 59 的整数，表示 *date* 中分钟里的新秒数。大于 59 的或者小于 0 的值将会被转换到 *date* 的分钟上。

**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**参见**

*Date.getSeconds()*

---

## Date.setTime()方法

基于从 1970 年 1 月 1 日到新日期之间的毫秒数而指定一个新的日期

**有效性** Flash 5

**概要** Date.setTime(*milliseconds*)

**参数**

*milliseconds* 一个整数，表示新的指定日期和 1970 年 1 月 1 日午夜之间的毫秒数。如果新的日期在 1970 年 1 月 1 日之后就为正数，在其之前就为负数。

**返回**

*milliseconds* 的值。

**描述**

在内部，所有的日期都表示为从日期时间到 1970 年 1 月 1 日午夜之间的毫秒数。*setTime()* 方法用内部的毫秒数表示来指定一个新的日期。用从 1970 年开始的毫秒数来设置日期，在我们要用 *getTime()* 来确定多个日期和时间之间的差异时通常很方便。

**示例**

将 *setTime()* 和 *getTime()* 协同使用，可以通过加减毫秒数来调整现有日期的时间。例如，在下面的代码中，我们在日期上加一个小时：

```
now = new Date();
now.setTime(now.getTime() + 3600000);
```

在下面的代码中加一天：

```
now = new Date();
now.setTime(now.getTime() + 86400000);
```

要提高代码的可读性，我们创建用来表示小时内以及一天内所包含毫秒数的变量：

```
oneDay = 86400000;
oneHour = 3600000;
now = new Date();
// 减去一天零三个小时
now.setTime(now.getTime() - oneDay - (3 * oneHour));
```

**参见**

*Date.getTime()*, *Date.setMilliseconds()*, *Date.UTC()*, *getTimer()*

**Date.setUTCDate()方法**

指定月份中的天数(UTC)

**有效性** Flash 5**概要** `date.setUTCDate(day)`**参数**

*day* 一个从 1 到 31 的整数，表示 UTC 时间 *date* 中月份里的新天数。如果你指定的天数比当前月中总的天数大，月份就会相应增长。例如，如果当前的 *month* 为 8 (九月份)，你为新的 *day* 指定了 31，那么就会得到 10 月 1 日。*day* 会变成 1，而 *month* 会变成 9 (10 月)。

**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**Date.setUTCFullYear()方法** 指定四位数字格式的世纪和年份(UTC 时间)**有效性** Flash 5**概要** `date.setUTCFullYear(year)`**参数**

*year* 一个四位的整数，表示 UTC 时间 *date* 中的新年份，例如 1999 或者 2000。

**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**Date.setUTCHours()方法**

指定一天中的小时数(UTC 时间)

**有效性** Flash 5**概要** `date.setUTCHours(hour)`**参数**

*hour* 一个从 0 (午夜) 到 23 (11 p.m.) 的整数，指定 UTC 时间 *date* 中一天中新的小时数。

**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

---

**Date.setUTCMilliseconds()方法** 指定日期中的毫秒数(UTC 时间)**有效性** Flash 5**概要** `date.setUTCMilliseconds(ms)`**参数****ms** 一个从 0 到 999 的整数，表示 UTC 时间 `date` 中的新毫秒数。**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

---

**Date.setUTCMinutes()方法** 指定日期中的分钟数(UTC 时间)**有效性** Flash 5**概要** `date.setUTCMinutes(minute)`**参数****minute** 一个从 0 到 59 的整数，表示 UTC 时间 `date` 的小时中的新分钟数。**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

---

**Date.setUTCMonth()方法** 指定年份中的月份(UTC 时间)**有效性** Flash 5**概要** `date.setUTCMonth(month)`**参数****month** 一个从 0 (1 月) 到 11 (12 月) 的整数，而不是从 1 到 12，表示 UTC 时间 `date` 中年份里的新月份。**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**用法**

注意，不要将 1 当做 1 月份！月份是从 0 而不是从 1 开始的。

**Date.setUTCSeconds()方法**

指定日期中的秒数(UTC 时间)

**有效性** Flash 5**概要** `date.setUTCSeconds(second)`**参数****second** 一个从 0 到 59 的整数，表示 UTC 时间 `date` 中分钟里的新秒数。**返回**

一个整数，表示 UTC 时间新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**Date.setYear()方法**

指定年份，和 1900 年相关

**有效性** Flash 5**概要** `date.setYear(year, month, day)`**参数****year** 一个必须有的整数，表示 `date` 的新年份，例如 1999 或者 2000。如果提供了一个或者两位数字，年份就被当作是 20 世纪中的。例如，1 是 1901 年，而 99 表示 1999 年。三位数字的年份表示公元 1000 年以前的年份。用四位数字来指定 2000 年和以后的年份。**month** 一个可选的整数，从 0 (1 月) 到 11 (12 月)，而不是从 1 到 12，它表示 `date` 年份中新的月份。如果没有设置就默认为 0。**day** 一个可选的整数，从 1 到 31，表示 `date` 的月份中的新日期。**返回**

一个整数，表示新的日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**描述**

`setYear()` 和 `setFullYear()` 基本是同样的，只不过前者将提供的一位和两位数字的年份当作和 1900 年相关的，但是后者将它们当作和公元 0 年相关的。

**参见**`Date.getYear()`, `Date.setFullYear()`

---

**Date.toString()方法**

表示日期的一个可读串

**有效性** Flash 5**概要** date.toString()**返回**

一个串，依照读者可以读懂的格式，给出当前时间和 date 的日期，包括 UTC (GMT) 偏移量。要构成自定义的日期表示，可使用获取天数、小时、分钟等内容的方法，然后将这些值变换到你自己的串上。如 date 类在前面的例子中所示。

**示例**

```
trace (myDate.toString()); // 显示这种格式的日期:  
// Wed Sep 15 12:11:33 GMT-0400 1999
```

---

**Date.UTC()类方法**

获取从 1970 年 1 月 1 日到所给 UTC 日期之间的毫秒数

**有效性** Flash 5**概要** Date.UTC(*year, month, day, hours, minutes, seconds, ms*)**参数**

*year,...ms* 一系列的数字值，描述日期和时间，为 UTC 时间而非本地时间。对于每一个参数的描述，参见 *Date()* 构造器。

**返回**

指定日期和 1970 年 1 月 1 日午夜之间的毫秒数。

**描述**

*Date.UTC()* 方法和 *Date()* 构造器有相同的参数，但是它不返回指定日期的对象，而是返回内部格式中从 1970 年开始的毫秒数。返回的值通常用于构造一个 UTC 中的新 *Date* 对象，或者通过 *setTime()* 方法将 UTC 时间赋给一个现有的 *Date* 对象。

**示例**

下面的代码显示了如何计量 UTC 时间从 1970 年午夜到 2000 年午夜之间所经过的毫秒数：

```
trace(Date.UTC(2000, 0) + " milliseconds passed between 1970 and 2000.");  
// 显示: "946684800000 milliseconds passed between 1970 and 2000."
```

下面，我们用这个毫秒数来构造一个基于 UTC 时间的 *date* 对象：

```
nowUTC = new Date(Date.UTC(2000, 0));
```

如果这些代码在EST(东部标准时间)里调用,也就是比UTC晚5个小时,那么nowUTC将会表示为本地时间1999年12月31日7p.m.。当我们用非UTC方法getHours()检查小时的时候,会得到当地的小数19(24小时时钟7p.m.):

```
trace(nowUTC.getHours()); // 显示: 19
```

但是,当我们用UTC方法getUTCHours()来检查小时数的时候,会得到正确的UTC小时0(24小时时钟的午夜):

```
trace(nowUTC.getUTCHours()); // 显示: 0
```

## 参见

*Date()* *Date.setTime()*, *Date* 类

---

### Date.valueOf()方法

1970年1月1日到Date对象时间之间的毫秒数

**有效性** Flash 5

**概要** date.valueOf()

## 返回

一个整数,表示Date对象和1970年1月1日午夜之间的毫秒数。如果时间在1970年1月1日之后则为整数,在其之前则为负数。

## 描述

实际上, *Date.valueOf()* 和 *Date.getTime()* 是等同的。

## 参见

*Date.toString()*

---

### duplicateMovieClip()全局函数

创建一个影片剪辑的副本

**有效性** Flash 5

**概要** duplicateMovieClip(target, newName, depth)

## 参数

*target* 一个串,表示到要复制的影片剪辑(称为种子剪辑)的路径。嵌套的剪辑会用点语法来引用,比如 *duplicateMovieClip("\_root.my-*

*Clip*"*"myClip2",0*)。因为影片剪辑的引用在用于串语境中的时候被转换成了路径，*target* 也会是一个影片剪辑对象引用，比如 *duplicateMovieClip(myClip,"myClip2",0)*。

*newName* 一个串，会成为复制剪辑实例的名称。所使用的串必须遵循标识符创建的规则，这些规则在第十四章中有所描述。

*depth* 一个整数，表示将复制剪辑放在程序化剪辑堆栈里的哪个层级。层级较低的剪辑在视觉上处于层级较高的剪辑之后。堆栈中 *depth* 值最高的影片剪辑遮盖它下面的所有剪辑。例如，一个 *depth* 为 1 的剪辑在 *depth* 为 0 的剪辑后面，而这个剪辑又在 *depth* 为 1 的剪辑后面。如果该层级已经被占据，那么原来的占据者将被删除，新的复本将代替它的位置。参见第十三章以获得其他信息。深度为负值非常有用，但是在 ActionScript 中不被正式支持，要确保以后的兼容性，可以使用深度 0 或者数值更大的深度。

## 描述

*duplicateMovieClip()* 函数是在影片播放期间创建新的影片剪辑的一种方法（其他方法还有 *attachMovie()*, *duplicateMovieClip()*，它们创建 *target* 的一个同样的拷贝，并将拷贝放在 *target* 剪辑堆栈的 *depth* 层上。复制剪辑在第 1 帧开始播放，不管 *target* 的帧是什么，也不管复制是什么时候发生的。

复制剪辑继承了 *target* 中的所有变换（旋转、缩放等等），但是不继承 *target* 的时间线变量。全局函数 *duplicateMovieClip()* 也是一种可用的影片剪辑方法，虽然用在这种格式中的时候，不使用 *target* 参数。

## 示例

```
// 复制 ball 剪辑，并将拷贝命名为 ball2  
duplicateMovieClip(ball, "ball2", 0);  
// 移动新的 ball2 剪辑，以便我们能看到  
ball2._x += 100;
```

## 参见

*MovieClip.duplicateMovieClip()* *removeMovieClip()*, 第十三章

---

## escape()全局函数

对串进行编码，以便安全地进行网络传输

### 有效性

Flash 5

**概要** `escape(string)`

**参数**

`string` 一个要被编码的串（或者能产生串的表达式）。

**返回**

`string` 的一个 URL 编码版本（大体如此）。

**描述**

`escape()` 函数基于提供的串创建一个新的编码串。新的串包含一个十六进制的转义序列，代替串中任何非数字或非 A 到 Z、a 到 z 的 Latin 字母的字符。用来替换的十六进制转义序列的格式为 %xx，xx 是 Latin 1 字符集中字符编码点的十六进制值。shift-JIS 双字节字符被转换为格式为 %xx%xx 的十六进制转义序列。

`escape()` 函数可有效地对一个串进行 URL 编码，但是空格符被转换为 %20，而不是 +。有的时候，在 Flash 影片向服务器应用程序发送信息，或者在浏览器中记录 cookies 的时候使用 `escape()`。

要对编码串进行译码，我们使用全局函数 `unescape()`。

**示例**

```
var phoneNumber = '(222)515-1212';
escape(phoneNumber); // 产生 %28222%29%20515%2D1212
```

**参见**

`unescape()`, 附录二

---

**eval() 全局函数**

将串转换为标识符

**有效性** Flash 4 及以后的版本

**概要** `eval(stringExpression)`

**参数**

`stringExpression` 一个串或者能产生串的表达式，应该和某个标识符的名称相匹配。

**返回**

用 `stringExpression` 表示的变量值，或者由 `stringExpression` 表示的指向对象、影片剪辑或函数的引用。如果 `stringExpression` 不表示一个变量或者影片剪辑，就返回 `undefined`。

## 描述

*eval()*函数提供了对基于串或文本的标识符构造动态引用的一种方法。*eval()*将串转换为变量、影片剪辑、对象属性或者其他标识符，然后对标识符求值。例如，下面的代码中，我们用*eval()*的返回值来设置变量的值：

```
name1 = "Kathy";
count = 1;
currentName = eval("name" + count); // 将currentName设置为"Kathy"
```

下面，我们要控制一个动态命名的影片剪辑 *star1*：

```
eval("star" + count)._x += 100; // 将star1向右移动100像素
```

但是我们也可以调用*eval()*来代替赋值表达式左边的标识符，比如：

```
eval("name" + count) = "Simone"; // 将name1设置为"Simone"
```

注意，ActionScript 中的*eval()*和它的JavaScript兄弟不同，它不允许串中任意的代码块得到编译和执行。*eval()*的全部支持是在播放器中需要一个ActionScript编译器，这会让播放器尺寸变得很大。

## 用法

和 Flash 5 一样，*eval()*很少需要动态的变量访问。管理多个资料的时候，要使用数组和对象。

## 示例

*eval()*函数通常用来将 *MovieClip.\_droptarget* 串属性转换为影片剪辑对象引用。在下面的例子中，假设我们有一系列卡通脸谱剪辑。当用户将 food 剪辑放到某个脸谱上的时候，我们用 *\_droptarget* 来获取到所讨论的脸谱的路径，然后，使用*eval()*来获取该脸谱的对象引用。最后，我们将脸谱发送给显示填满食物的嘴的帧：

```
// 将_droptarget串转换为一个引用
theFace = eval(food._droptarget);
// 用转换过来的引用控制对应的脸谱剪辑
theFace.gotoAndStop("full");
```

## 参见

第四章

---

**\_focusrect 全局属性**

由键盘激活的按钮的高亮状态

**有效性**

Flash 4 及以后的版本

**概要**            `_focusrect`

**访问**            读 / 写

### 描述

当 Flash 中的鼠标从按钮上经过的时候，按钮的 Over 状态就可以显示出来。按钮也可以在用户按下 Tab 键的时候得到键盘的焦点。当一个按钮得到了键盘焦点的时候，Flash 会在按钮上放一个黄色的矩形，它在视觉效果上并不总是让人感到满意。可以用 `_focusrect` 全局属性来去掉这个黄色的矩形，比如：

```
_focusrect = false;
```

当 `_focusrect` 被设置为 `false` 的时候，Flash 显示有键盘焦点的按钮的 Over 状态。

当 `_focusrect` 被设置为 `true`（它的默认值）时，Flash 显示黄色的矩形。

### 用法

虽然 `_focusrect` 用于布尔形式，但是它实际上存储了一个数字，而不是一个布尔值。虽然没有必要这样做，但是如果我们检查 `_focusrect` 的值，它会返回 1（表示 `true`）或者 0（表示 `false`）。

---

## fscommand()全局函数

给独立的播放器或到播放器的  
主机应用程序发送一个消息

**有效性**        Flash3 及以后的版本（在 Flash 5 中得到了增强，包括了 `trapallkeys`）

**概要**            `fscommand(command, arguments)`

### 参数

*command*        一个传递给主机应用程序的串，通常是 JavaScript 的一个函数名称。

*arguments*        一个传递给主机应用程序的串，通常是由 *command* 所命名的函数的参数。

### 描述

使用 `fscommand()` 函数，一个 Flash 影片剪辑可以同独立的播放器或播放器的主机应用程序——Flash 播放器的运行环境（也就是 Web 浏览器或者 Macromedia 控制器）——通信。`fscommand()` 函数通常有以下三种应用方式：

- 将一组有限的内置命令发送到独立的 Flash 播放器。
- 将命令发送到 Web 浏览器里诸如 JavaScript 或者 VBScript 这样的脚本语言中。

- 和 Macromedia 控制器影片中的 Lingo 通信。

使用在独立播放器中的时候，*fscommand()*带一组内置的 command/argument 对，如表 R-5 所示。

表 R-5 独立播放器中的命令/参数 (command/argument) 对

命令	参数	描述
"allowscale"	"true" 或 "false"	为 "false" 的时候，可以防止影片的内容随着播放器的窗口尺寸而变化。"allowscale" 通常和 "fullscreen"一起使用，来创建全屏的放映机，保持影片的最初尺寸。
"exec"	"application_name"	启动一个外部的应用程序。到这个应用程序的路径是在串 <i>application_name</i> 中指定的。路径必须和 Flash 影片相关，除非 <i>application_name</i> 是作为绝对路径来表示的，比如："C:/WINDOWS/NOTEPAD.EXE"。注意，这个路径使用的是斜杠 (/)，而不是反斜杠 (\)。
"fullscreen"	"true" or "false"	为 "true" 的时候，可以让播放器窗口最大化，填满用户的整个屏幕。
"quit"	不使用	关闭影片，推出 Flash 控制器（独立的播放器）。
"showmenu"	"true" 或 "false"	为 "false" 的时候，禁止播放器内容菜单中的控制显示，只留下“关于 Macromedia Flash 播放器”选项。内容菜单可以通过在 Windows 中点击右键（在 Macintosh 中按 Ctrl-click）来访问。
"trapallkeys"	"true" 或 "false"	为 "true" 的时候，让所有的击键动作——甚至键盘快捷键——都发送到 Flash 影片。 <i>trapallkeys</i> 用来取消独立播放器中的控制键（也就是全屏模式的 Ctrl-F 或者 Command-F，退出的 Ctrl-Q 或 Command-Q，停止/退出全屏模式的 Esc，等等）。在 Flash 5 中还加上了 <i>fscommand</i> 。

在浏览器中使用的时候，影片中一个 *fscommand()* 函数的执行会在包含该影片的页上调用特殊的 JavaScript 函数 (Netscape) 或者 VBScript 函数 (IE)。这个特殊函数名称的一般格式为 *movieID\_DoFSCommand*，*movieID* 是在来自主机 HTML 文件的影片的 OBJECT ID 属性 (IE) 或者 EMBED NAME 属性 (Netscape) 中指定的。当 *movieID\_DoFSCommand()* 被调用的时候，*fscommand()* 的 command 和 arguments

参数的值就被传递给 `movieID_DoFSCommand()` 函数作为参数。如果主页面中不存在 `movieID_DoFSCommand()` 函数，`fscommand()` 就失败了。

注意，为了让 `fscommand()` 在 Netscape 中运行，影片的 EMBED 标签的 swLiveConnect 属性必须被设置为 "true"。例如：

```
<EMBED  
    NAME="testMovie"  
    SRC="myMovie.swf"  
    WIDTH="100%"  
    HEIGHT="100%"  
    swLiveConnect="true"  
    PLUGINDPAGE="http://www.macromedia.com/go/flashplayer/"  
</EMBED>
```

## 用法

在下面的系统配置下不能通过 `fscommand()` 与浏览器通信：

- Macintosh OS 上的 Internet Explorer
- 运行在 68K 系列 Macintosh 上的任何浏览器
- 运行在 Windows 3.1 上的任何浏览器
- Netscape 6

注意，将 `fscommand()` 用于从 Flash 与控制器影片通信并不总是最好的方法。首选的控制器通信设备是 `getURL()` 函数，它要么带 `event:`，要么带 `lingo:` 协议。至于细节的内容，请参见 `getURL()` 函数，或者下面的 Macromedia 技术要点：

[http://www.macromedia.com/support/director/ts/documents/flash\\_xtra\\_sending\\_events.htm](http://www.macromedia.com/support/director/ts/documents/flash_xtra_sending_events.htm)  
[http://www.macromedia.com/support/director/ts/documents/flash\\_xtra\\_lingo.htm](http://www.macromedia.com/support/director/ts/documents/flash_xtra_lingo.htm)  
[http://www.macromedia.com/support/director/ts/documents/flash\\_tips.htm](http://www.macromedia.com/support/director/ts/documents/flash_tips.htm)

## 示例

要退出一个独立控制器，使用：

```
fscommand('quit');
```

要创建一个以全屏模式运行的独立控制器，使用：

```
fscommand('fullscreen', "true");
```

要创建一个独立控制器，它运行全屏模式，但是保持最初影片的尺寸，使用：

```
fscommand("fullscreen", "true");
fscommand("allowscale", "false");
```

至于如何在全屏的 Web 浏览器窗口中启动一个影片的信息，参见 <http://www.moock.org/webdesign/flash/launchwindow/fullscreen>。

要启动大多数 Windows 系统中的记事本，使用：

```
fscommand("exec", "C:/WINDOWS/NOTEPAD.EXE");
```

下面的代码显示了一个 HTML 页，它需要 JavaScript 和 VBScript 从影片中对简单的 *fscommand()* 作出响应。注意，VBScript 函数只调用 JavaScript 函数，这让我们可以只用一个 JavaScript 函数就处理了 IE 和 Netscape：

```
<HTML>
<HEAD>
<TITLE>fscommand demo</TITLE>

<SCRIPT LANGUAGE='JavaScript'>
<!--
function testmovie_DoFSCommand(command, args) {
    alert("Here's the Flash message ' + command + ", " + args);
}
//-->
</SCRIPT>

<SCRIPT LANGUAGE="VBScript">
<!--
Sub testmovie_FSCmd(ByVal command, ByVal args)
    call testmovie_DoFSCommand(command, args)
end sub
//-->
</SCRIPT>

</HEAD>

<BODY BGCOLOR="#FFFFFF">

<OBJECT
    ID="testmovie"
    CLASSID="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    WIDTH="100%"
    HEIGHT="100%"
    CODEBASE="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
<PARAM NAME='MOVIE' VALUE='flash-to-javascript.swf'>

<EMBED
    NAME="testmovie"
    SRC="flash-to-javascript.swf"
    WIDTH="100%"
    HEIGHT="100%">
```

```
swLiveConnect= true"
PLUGINSPAGE="http://WWW.macromedia.com/go/flashplayer/"
</EMBED>
</OBJECT>

</BODY>
</HTML>
```

要从 *flash-to-javascript.swf* 影片中调用前边所说的 *testmovie\_DoFSCommand()* 函数，可以使用：

```
fscommand("hello", "world");
```

## 参见

对于 *fscommand()* 的更多信息，以及用 JavaScript 控制 Flash 的内容，参见：

<http://www.moock.org/webdesign/flash/fscommand>

<http://www.macromedia.com/support/flash/publishexport/scriptingwithflash>

---

## getProperty()全局函数

获取影片剪辑属性的值

**有效性** Flash 4，在 Flash 5 中不被支持

**概要** `getProperty(movieClip, property)`

**参数**

*movieClip* 一个能产生描述影片剪辑路径的串的表达式。在 Flash 5 中，也可以是一个影片剪辑引用，因为影片剪辑引用在使用于串语境中的时候会转换为路径。

*property* 要获取的内置属性的名称。必须是一个标识符而不是一个串(例如，`_x` 而不是 "`_x`" )。

**返回**

*movieClip* 的 *property* 的值。

**描述**

*getProperty()* 函数获取影片剪辑的一个内置属性的值。虽然 *setProperty()* 在 Flash 4 中是访问对象属性的惟一方法，但是 . 和 [] 操作符在 Flash 5 以及以后版本中是首选的属性访问工具。

## 示例

下面每一个对 *getProperty()* 的调用都获取影片 ball 的 \_x 属性值，ball 位于主影片时间线上：

```
getProperty(ball,_x);           // 相对影片剪辑引用  
getProperty(_root.ball,_x);    // 绝对影片剪辑引用  
getProperty("ball",_x);        // 串中的相对路径  
getProperty("_root.ball",_x);   // 串中的点路径  
getProperty("/ball",_x);        // 串中的斜杠路径
```

下面的代码显示了用点号和 [] 操作符来访问类似属性的方式：

```
ball._x;  
_root.ball._x;  
ball."_x";  
_root["ball"][_x];
```

## 参见

*setProperty()*, 第十三章

## getTimer()全局函数

确定影片的使用时间，以毫秒为单位

**有效性** Flash 4 及以后的版本

**概要** `getTimer()`

**返回**

影片播映的时间，以毫秒计数。

## 描述

*getTimer()* 函数表示一个影片被播放了多久，以毫秒为单位。可以使用多 *getTimer()* 检查来控制代码块的定时执行，或者对影片添加基于时间的一些功能，比如视频游戏中的一个倒计时。例如，当下面的代码被添加到一个包含文本域 counterOutput 的影片剪辑中时，会从 60 倒数倒到 0：

```
onClipEvent (load) {  
    // 记录当前的时间  
    var startTime = getTimer();  
    // 设置要倒数的秒数  
    var countAmount = 60;  
    // 初始化一个变量，用来跟踪过了多少时间  
    var elapsed = 0;  
}  
  
onClipEvent (enterFrame) {  
    // 检查过了多少时间
```

```

elapsed = getTimer() - startTime;
// 如果过去的时间小于倒数时间……
if (elapsed < countAmount * 1000) {
    // ... 设置文本域，显示剩下的时间
    counterOutput = countAmount - Math.floor(elapsed / 1000);
} else {
    // ... 否则，我们的倒数就结束，可以告诉用户了
    counterOutput = "Time's UP!";
}
}

```

要确定影片所经过的秒数（不是毫秒），可以将 *getTimer()* 的返回值除以 1000，然后用 *Math.floor()* 或者 *Math.round()*，或 *Math.ceil()* 来去掉小数部分。例如：

```
numSeconds=Math.floor(getTimer()/1000);
```

### 示例

下面的代码在两个帧之间循环，直到影片超过 10 秒，然后开始播放影片：

```

now = getTimer();
if (now > 10000) {
    play();
} else {
    gotoAndPlay(_currentframe - 1);
}

```

### 参见

*Date()*, *Date* 类

## getURL()全局函数

将一个文档装载到浏览器中，执行服务器端脚本，或者触发 Macromedia 控制器事件

### 有效性

Flash 2 和 Flash 3，在 Flash 4 中包括了 *method* 参数而得到加强，  
Flash 5

### 概要

```

getURL ( URL )
getURL ( URL, window )
getURL ( URL, window, method )

```

### 参数

#### URL

一个串，用来指定要装载的文档或者要运行的外部脚本的相对或者绝对位置。

#### window

一个可选串，用来指定要装载文档的浏览器窗口或者帧的名称。可

以是一个自定义的名称，或者是以下四个预设值之一：“\_blank”，“\_parent”，“\_self”，“\_top”。

*method* 一个可选的串，指定要将当前时间线的变量发送给外部脚本——“GET”或“POST”——的方法。这个参数必须是常量串，而不是变量或者其他表达式。Flash 播放器的独立版本经常使用“GET”方法，而不管指定的是什么 *method*。

### 描述

*getURL()* 函数可以用来：

- 将一个文档（通常是一个 Web 页）装载到 Web 浏览器的帧或者窗口。
- 执行服务器端的脚本，获取浏览器帧或窗口中的结果。
- 在 Web 浏览器中执行 JavaScript 代码。
- 从作为小精灵导入到控制器中的 Flash 资源来触发事件。

要将一个文档装载到当前的窗口或者帧，只要指定文档的 URL，而不用提供 *window* 或 *method* 参数。当然，Flash 会支持绝对 URL（包含诸如“http:”这样的协议，后面加上服务器名称或者硬件设备）以及相对的 URL（和当前的位置相关）：

```
getURL("http://www.moock.org/");           // 到 Web 页的绝对 URL  
getURL("file:///C|WINDOWS/Desktop/index.html"); // 到本地文件的绝对 URL  
getURL("/whatever/index.html");             // 相对 URL，假定  
   // 是 http 协议
```

要将一个文档装载到命名的窗口或者帧，要提供窗口或者帧的名称作为 *window* 参数。例如：

```
getURL("http://www.moock.org/", "contentFrame"); // 装载到命名帧  
getURL("http://www.moock.org/", "remoteWin");    // 装载到命名窗口
```

要替代包含当前影片的框架，用“\_parent”作为 *window* 参数。例如：

```
getURL("http://www.moock.org/", "_parent");
```

要用加载的文档来替代 Web 页中的所有框架，用“\_top”作为 *window* 参数的值。例如：

```
getURL("http://www.moock.org/", '_top');
```

要在一个新的、匿名的浏览器窗口中打开装载的文档，将“\_blank”用作 *window* 参数的值。例如：

```
getURL('http://www.moock.org/', '_blank');
```

注意，要用“\_blank”来启动一个新的窗口并不能让我们控制新窗口的外观（尺寸、工具栏配置、位置，等等）。要用`getURL()`启动一个自定义的窗口，必须在影片的主页面调用 JavaScript 函数。JavaScript 窗口启动技术在 <http://www.moock.org/webdesign/flash> 中有描述。

`getURL()`函数也可以用来将变量发送到远程的服务器应用程序或脚本。要将当前影片剪辑的时间线变量发送给外部的脚本，要将脚本的名称指定为 URL 参数，用“GET”或者“POST”来作为`method`参数的值。例如：

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "resultsFrame", "GET");
```

作为一个影片剪辑方法来调用的时候，`getURL()`将发送该剪辑的时间线变量，比如：

```
// 将myClip的变量发送到search.pl  
myClip.getURL("http://www.server.com/cgi-bin/search.pl", "resultsFrame", "GET");
```

脚本执行的结果会出现在窗口中或者`getURL()`的`window`参数（提交变量的时候要求使用）所指定的帧中。

要将脚本的执行结果装载到当前的帧或窗口，将“\_self”作为`window`参数的值，比如：

```
getURL("http://www.someserver.com/cgi-bin/search.pl", "_self", "GET");
```

当`method`参数为“GET”的时候，当前的影片剪辑时间线变量作为 HTTP GET 请求中属于脚本 URL 的询问串被发送。询问串由变量名/值对所组成，由和符号（&）来分隔。例如：

```
http://www.someserver.com/cgi-bin/search.pl?term=javascript&scope=entiresite
```

当`method`参数为“POST”的时候，当前的影片剪辑时间线变量就作为单独的数据块被发送给脚本，它在 HTTP POST 请求标头（实际上类似于使用 POST 方法的正规 HTML 格式）之后。注意，“POST”在独立的 Flash 播放器中是无效的。

因为大部分 Web 服务器将 URL 的长度限制在 255 和 1024 个字符之间，所以可以使用“POST”代替“GET”以传送大量的资料。

注意，`getURL()`调用的脚本所发送的任何返回信息都作为正规 Web 内容在浏览器中显示，而不在 Flash 中显示。要将一个脚本执行结果收到 Flash 中，可使用`loadVariables()`。

*getURL()* 函数也可以用于除了“http:”之外的其他协议，如表 R-6 所示。

表 R-6 *getURL* 所支持的协议

协议	格式	目的
事件	"event: <i>eventName</i> <i>params</i> "	如果 Flash 资源是一个控制器小精灵，就将事件发送到控制器。
文件	"file:///driveSpec/folder/filename"	访问一个本地文件。
ftp	"ftp://server.domain.com/folder/filename"	通过 FTP（文件传输协议）访问一个远程文件。
http	"http://server.domain.com/folder/filename"	通过 HTTP（超文本传输协议）来访问远程文件。
javascript	"javascript: <i>command</i> "	在浏览器中执行 JavaScript 命令。
lingo	"lingo: <i>command</i> "	如果 Flash 资源是一个控制器小精灵，就执行 Lingo 命令。
print	"print:", "targetClip"	在 Flash 4 中进行打印、优先于 Flash 5 中的 <i>print()</i> 函数。
vbscript	"vbscript: <i>command</i> "	在浏览器中执行 VBScript 命令。

如表 R-6 所示，如果 Flash 资源作为 Macromedia 控制器文件导入，那么 *getURL()* 可以用来触发 Lingo 事件或者执行 Lingo 命令（Lingo 是控制器脚本语言，类似 ActionScript）。例如，可以用以下格式来添加一个帧事件：

```
getURL ("event: eventName params");
```

这会在 *eventName* 上命名的 Lingo 事件处理器在控制器中得到调用。下面是一个 *getURL()* 语句，它产生名为“myEvent”的事件，然后将它传递给串“A”。注意，\* 符号可以用 \ 序列来代替：

```
getURL ('event:myEvent \'A\''); // 发送一个事件给控制器
```

下面是 Lingo 的小精灵事件处理器，它应该被添加到控制器中的 Flash 小精灵里，以便接收事件。注意，控制器小精灵基本上相当于 Flash 影片剪辑实例，Lingo 关键词 *put* 相当于 ActionScript 的 *trace()* 命令，而 *&&* 是 Lingo 的串连接操作符：

```
on myEvent msg
    put 'The message received from Flash was ' && msg
end
```

你也可以在控制器里的 Flash 小精灵中使用 "lingo:" 关键词来触发 Lingo 的执行，比如：

```
getURL ('lingo: beep'); // 让控制器播放一个嘟嘟的声音
```

注意，Director 8.0 不能导入 Flash 5 的 .swf 文件，但是更新之后的 Flash 资源 Xtra 等到你读本书的时候可能可以使用了。

最后，*getURL()* 也可以用来执行 JavaScript 代码。下面，我们用 *getURL()* 调用一个简单的 JavaScript 的 *alert*：

```
getURL ("javascript: alert('hello world');");
```

注意，从 URL 执行 JavaScript 代码在 Macintosh 的 IE 4.5 中无效。

## 示例

下面的代码是一个链接到 Web 页面的标准按钮：

```
on (release) {
    getURL('http://www.macromedia.com/');
}
```

## 故障

在 Macintosh 上，IE 4.5 和更老的版本都不支持 *getURL()* 调用的 "POST" 方法。要服务于这些浏览器，可使用 "GET" 来代替 "POST"（要服从前边所说的长度限制）。

在大部分浏览器中，*getURL()* 的相关链接都和包含 .swf 文件的 HTML 文件有关。在 Macintosh 上的 IE 4.5 以及更老版本中，相关链接和 .swf 文件而不是 HTML 文件的位置相关，这在两者位于不同目录中时会产生问题。要避免这个问题，可以将 .swf 和 .html 文件放在相同的目录中，也可以在调用 *getURL()* 的时候使用绝对路径，比如：

```
getURL ("http://WWW.someserver.com/")
```

## 参见

*loadVariables()*, *MovieClip.getURL()*, *movieClip.loadVariables()*, 第十八章

---

## getVersion()全局函数

检查 Flash 播放器的平台和版本

**有效性** Flash 5

**概要** `getVersion()`

## 返回

一个串，包含容纳当前影片的 Flash 播放器的版本和平台信息。

## 描述

*getVersion()*函数告诉我们用来浏览影片的平台和Flash播放器的版本。它可以用来针对 Flash 播放器的版本或者当前运行的操作系统而选择性地执行特定的代码。*GetVersion()*返回的串格式如下：

*platform majorVersion,minorVersion,buildNumber,patch*

*platform*用来指定平台 ("WIN", "MAC" 或者 "UNIX") 的代码，后面跟着主版本号码，次版本号码，以及制作（也叫做修订）号码。最后一个项目 *patch* 通常为 0。例如：

WIN 5,0,30,0	// Windows 上的版本 5.0，修订本 30 (5.0r30)
MAC 5,0,41,0	// Macintosh 上的版本 5.0，修订本 41 (5.0r41)
UNIX 4,0,12,0	// UNIX 上的版本 4.0，修订本 12 (4.0r12)

和 Macromedia 文件的声明相反，*getVersion()*的确能工作在 Flash 制作工具的测试影片模式中。它给出嵌入到制作工具中的播放器的版本号码（和制作工具本身的版本不同）。例如，Flash 5 制作工具嵌入了播放器的 5.0 r30 版本，因此，它的 *getVersion()* 函数给出这样的信息：

WIN 5,0,30,0  
Or  
MAC 5,0,30,0

当制作工具的主版本或者次版本创建的时候，*buildNumber*就从 0 开始。但是，在 Flash 制作工具的典型开发周期中，Flash 播放器的很多修订本都会在制作工具的最终版本发布之前产生。因此，播放器第一个新的主版本通常比 0 大。例如，Flash 5 播放器的第一次正式发布是修订本 30。当 Flash 6 产生的时候，Flash 6 播放器大概会返回这样的东西：

WIN 6.0,xx,0

但是如果在 Flash 6 中创建的影片在 Flash 5 播放器中播放，那么 *getVersion()*仍然会返回：

WIN 5,0,30,0

通常，我们只关心播放器的平台、主版本和修订本。要提取 *getVersion()*串中我们需要的部分，可以使用第四章中所描述的串操作工具，或者建立一个自定义的对象，

在其中，*getVersion()*的每一个成分都被赋给一个属性，这个对象会在下面的例子中给出。

除非我们需要对Flash播放器的特定版本编写内部的ActionScript代码块，JavaScript和VBScript都提供了较好的工具来进行版本、浏览器探测和自动的页面重新定位。此外，不能使用*getVersion()*，除非用户已经有了播放器的5.0版本或者更新的版本。有关用JavaScript和VBScript探测Flash播放器的存在和版本方面的内容，参见<http://www.moock.org/webdesign/flash/detection/moockfpi>。

### 示例

下面的代码将从*getVersion()*所返回的串中提取不同的部分，然后将它们存储为一个对象的属性，以方便访问：

```
// 将getVersion()串分裂为可用的部分
var version = getVersion();
var firstSpace = version.indexOf(' ');
var tempString = version.substring(firstSpace + 1, version.length);
var tempArray = tempString.split(',');

// 将getVersion()串的不同部分赋给我们的对象
// 注意，要将版本号码部分转换为整数
Var thePlayer = new Object();
ThePlayer.platform = version.substring(0,firstSpace);
ThePlayer.majorVersion = parseInt(tempArray[0]);
ThePlayer.minorVersion = parseInt(tempArray[1]);
ThePlayer.build = parseInt(tempArray[2]);
ThePlayer.patch = parseInt(tempArray[3]);

// 现在用我们的对象来执行特定版本代码
if (thePlayer.platform == "WIN") {
    // 执行Windows的特定代码
} else if (thePlayer.platform == "MAC") {
    // 执行Mac的特定代码
} else if (thePlayer.platform == "UNIX") {
    // 执行UNIX的特定代码
}

if ((thePlayer.majorVersion == 5) && (thePlayer.build == 30)) {
    trace ("I recommend upgrading your player to avoid text display bugs");
}
```

### 参见

\$version, 附录三

---

**gotoAndPlay()全局函数**

将播放头移动到给定的帧，然后  
播放当前的剪辑或影片

**有效性** Flash 2 及其以后的版本

**概要** `gotoAndPlay(frameNumber)`

`gotoAndPlay(frameLabel)`

`gotoAndPlay(scene, frameNumber)`

`gotoAndPlay(scene, frameLabel)`

**参数**

*frameNumber* 一个正整数，表示在播放之前当前时间线的播放头要落脚的帧。如果 *frameNumber* 小于 1 或者大于时间线上的帧数，那么播放头就被分别放到开始帧或者结束帧。

*frameLabel* 一个串，表示在播放之前当前时间线的播放头要落脚的帧的标签。如果 *frameLabel* 没有找到，那么播放头就被送到时间线的第一帧。

*Scene* 一个可选的串，表示包含指定 *frameNumber* 或者 *frameLabel* 的场景的名称。如果没有提供，就假设是当前的场景。

**描述**

调用的时候如果不带 *scene* 参数，*gotoAndPlay()* 就把当前时间线上的播放头送到 *frameNumber* 或者 *frameLabel* 所指定的帧，然后从该点开始播放时间线影片。“当前时间线”是指 *gotoAndPlay()* 函数被调用的影片剪辑或影片。

如果在 *gotoAndPlay()* 函数调用中提供了两个参数，那么第一个参数就被认为是 *scene*；如果只有一个参数，那么它就被当成是 *frameNumber* 或 *frameLabel*，场景则假设是当前的场景。

调用的时候如果只有一个 *scene* 参数，*gotoAndPlay()* 就将播放头放到指定场景中的帧号或者帧标签，然后播放该场景。如果使用了 *scene* 参数，就必须从 *\_root* 时间线调用 *gotoAndPlay()* 函数；否则，操作就会失败，播放头不会被送到预定的帧。注意，场景会在影片的播放中进行调整，变成单一的时间线。也就是说，如果场景 1 的时间线包含 20 帧，而场景 2 的时间线包含 10 帧，那么可以用 *gotoAndPlay(25)* 将播放头送到场景 2 的第 5 帧。

---

**注意：**我再次推荐在处理 ActionScript 加强影片的时候使用场景。和影片剪辑不同，场景不通过对象来表示，不能用大多数的内置函数来直接处理。在你的时间线上用标签和影片剪辑来作为伪帧，以代替 Flash 的场景功能通常会更好。

---

全局函数 *gotoAndPlay()* 只影响当前的时间线。当前时间线内其他影片剪辑的帧或者状态不受影响。要播放其他影片剪辑，必须对每一个影片剪辑使用一个单独的 *play()* 或者 *gotoAndPlay()* 命令。要对一个非当前影片剪辑的剪辑应用 *gotoAndPlay()* 函数，可使用影片剪辑方法的形式，*myClip.gotoAndPlay()*。

## 故障

在 Flash 播放器的修订版 5.0r30 中，*gotoAndPlay()* 不能工作于带 *frameLabel* 串直接量的 *onClipEvent()* 处理器。要解决这个问题，可用 *this* 来表示当前剪辑，以使用函数的影片剪辑变体，比如 *this.gotoAndPlay("myLabel")*，而不用 *gotoAndPlay("myLabel")*。

## 示例

```
// 到当前时间线的第 5 帧并播放
gotoAndPlay(5);
// 到 exitSequence 场景的第 10 帧并播放
gotoAndPlay("exitSequence", 10);
// 到 exitSequence 场景的 "goodbye" 帧并播放
gotoAndPlay("exitSequence", "goodbye");
// 小心！这将播放当前场景中的 "exitSequence" 帧
gotoAndPlay("exitSequence");
// 这将播放 exitSequence 场景的第 1 帧
gotoAndPlay("exitSequence", 1);
```

## 参见

*gotoAndStop()*, *MovieClip.gotoAndPlay()*, *play()*, *stop()*

---

## gotoAndStop() 全局函数

将播放头移动到给定的帧，  
然后停止当前剪辑

**有效性** Flash 2 以及以后的版本

**概要** *gotoAndStop(frameNumber)*

*gotoAndStop(frameLabel)*

*gotoAndStop(scene, frameNumber)*

*gotoAndStop(scene, frameLabel)*

## 参数

- frameNumber* 一个正整数，表示在播放之前当前时间线的播放头要落脚的帧。如果 *frameNumber* 小于 1 或者大于时间线上的帧数，那么播放头就被分别放到开始帧或者结束帧。
- frameLabel* 一个串，表示在播放之前当前时间线的播放头要落脚的帧的标签。如果 *frameLabel* 没有找到，那么播放头就被送到时间线的第一帧。
- Scene* 一个可选的串，表示包含指定 *frameNumber* 或 *frameLabel* 的场景的名称。如果没有提供，就假设是当前的场景。

## 描述

调用的时候如果不带 *scene* 参数，*gotoAndStop()*就把当前时间线上的播放头送到 *frameNumber* 或者 *frameLabel* 所指定的帧。“当前时间线”是指 *gotoAndStop()* 函数被调用的影片剪辑或者影片。播放头会停止在目标帧上，它到达目标帧以后不会再自动前进。

如果在 *gotoAndStop()* 函数调用中提供了两个参数，那么第一个参数就被认为是 *scene*；如果只有一个参数，那么它就被当成是 *frameNumber* 或 *frametable*，场景则假设是当前的场景。

调用的时候如果只有一个 *scene* 参数，*gotoAndStop()* 就将播放头放到指定场景中的帧号或者帧标签，然后终止播放。如果使用了 *scene* 参数，就必须从 *\_root* 时间线调用 *gotoAndStop()* 函数；否则，操作就会失败，播放头不会被送到预定的帧。

全局函数 *gotoAndStop()* 只影响当前的时间线。当前时间线内其他影片剪辑的帧或者状态不受影响。要播放其他影片剪辑，必须对每一个影片剪辑使用一个单独的 *gotoAndStop()* 命令。要对一个非当前影片剪辑的剪辑应用 *gotoAndStop()* 函数，可使用影片剪辑方法的形式，*myClip.gotoAndStop()*。

## 故障

在 Flash 播放器的修订版 5.0r30 中，*gotoAndStop()* 不能工作于带 *frameLabel* 串直接量的 *onClipEvent()* 处理器。要解决这个问题，用 *this* 来表示当前剪辑，以使用函数的影片剪辑变体，比如 *this.gotoAndStop("myLabel")*，而不用 *gotoAndStop("myLabel")*。

## 示例

```
// 到当前时间线的第 5 帧并停止  
gotoAndStop(5);  
// 到 introSequence 场景的第 20 帧并停止  
gotoAndStop("introSequence", 20);  
// 到 introSequence 场景的 "hello" 帧，并停止  
gotoAndStop("introSequence", "hello");  
// 小心！这将播放当前场景中的 "introSequence" 帧。  
gotoAndStop("introSequence");  
// 这将播放 introSequence 场景的第 1 帧  
gotoAndStop('introSequence', 1);
```

## 参见

*gotoAndPlay()*, *MovieClip.gotoAndStop()*, *play()*, *stop()*

## \_highquality 全局属性

播放器的渲染质量

**有效性** Flash 4，在 Flash 5 中因为 \_quality 而得到了增强

**概要** \_highquality

**访问** 读 / 写

### 描述

\_highquality全局属性存储0到2之间的一个整数，表示Flash播放器的渲染质量，如下所示：

0

低质量。位图和向量都没有保真（平滑）。

1

高质量。向量有保真，位图在没有动画时候可以保真。

2

最好质量。位图和向量都总是有保真。

从 Flash 5 开始，\_highquality被\_quality所代替，后者可以用来将影片的质量设置为“中等”以及“低”、“高”和“最佳”。

## 参见

\_quality, *toggleHighQuality()*

---

## #include 指令

导入外部 ActionScript 文件的文本

**有效性** Flash 5

**概要** #include *path*

**参数**

*path* 一个串，它表示要导入的脚本文件的名称和位置，可以对应.flx 文件来指定，也可以是一个绝对路径（参见下面的例子）。注意，在路径中只能用斜杠，不能用反斜杠。脚本文件应该以.as 作为文件扩展名。

### 描述

#include 指令将外部文本文件（可能是一个扩展名为.as 的文件）带进当前的脚本，并把它直接放在脚本中 #include 出现的地方。#include 操作是在编译的时候执行的，这表示影片中所包含的文本会在影片测试、导出或从制作工具打印的时候存在。如果外部文件在影片被导出之后发生变化，那么这个变化不会在影片中有反映。为了将改变添加到影片，这个影片必须重新导出。

#include 指令用来合并多个脚本，或者不同 Flash 设计（很像你使用一个外部资源库）之间的相同代码块。在版本控制系统工具（比如 CVS 或者 Microsoft 视觉资源保险库）中维护代码，或者使用你觉得比 ActionScript 编辑器更好的外部文件编辑器时，你可以通过这种方法来优化你的代码。当程序员单独处理一个创建 Flash 动画的图形作品时，这也是相当方便的。外部文件有助于代码库的使用，比如独立于当前时间线或者影片剪辑之外的函数库。对于需要和 Flash 文件紧密结合的代码来说，它们的作用不大。

### 用法

注意，#include 指令是以 # 符号开始的，它不使用括号，并且不能以分号来结束。任何以分号结束的 #include 语句都会引发错误，并且不能导入外部脚本。如果文件在指定的路径中没有被找到，那么指令会出错，不会导入所包含的外部文本。当用动作面板菜单中的检查语法命令（Ctrl-T 或者 Command-T）（在面板右上角的箭头按钮下面）执行语法检查的时候，外部文件中的文本也会被检查。

### 示例

下面的代码将一个名为.as 的外部文件导入到当前.flx 文件中。在使用相对路径的时候，*myScript.as* 必须和包含 include 指令的.flx 文件在同一个文件夹中：

```
#include "myScript.as"
```

我们可以构造一个包含子文件夹的相对路径。下面，假设 *myScript.as* 在当前 *.fla* 文件层级之下一个子文件夹 *includes* 中：

```
#include "includes/myScript.as"
```

用两个点号来表示当前文件夹之上的文件夹。下面，假设 *myScript.as* 在当前 *.fla* 文件的上一个层级：

```
#include "../myScript.as"
```

下面，假设 *myScript.as* 在一个名为 *includes* 的子文件夹中，该子文件夹和包含当前 *.fla* 文件的子文件夹相邻：

```
#include "../includes/myScript.as"
```

你也可以指定到任何文件夹的绝对路径，比如：

```
#include "C:/WINDOWS/Desktop/myScript.as"
```

但是绝对路径不能跨平台，如果你在不同的机器上用不同的目录地址来编译 *.fla* 文件，可能需要对路径进行一定的修改。注意驱动器文字规范的不同：

```
#include "C:/WINDOWS/Desktop/MyScript.as"           // Windows  
#include "Mac HD/Desktop folder/working/myScript.as" // Macintosh
```

## 参见

第十六章

---

## Infinity 全局属性

一个表示无穷大数字的常量

**有效性** Flash 5

**概要** Infinity

**访问** 读 / 写

### 描述

在 ActionScript 中，任何覆盖最大允许数字范围的数字都用数字常量 **Infinity** 来表示。在 ActionScript 中所允许的最大值是用 Number.MAX\_VALUE (1.7976931348623157e+308) 表示的。

## 示例

超过最大允许数字的计算结果就是 `Infinity`。例如：

```
Number.MAX_VALUE * 2;      // 产生 Infinity
```

在将正数除以 0 的时候，也产生 `Infinity`：

```
1000 / 0;                  // 产生 Infinity
```

## 用法

`Infinity` 是 `Number.POSITIVE_INFINITY` 的缩写。

## 参见

-`Infinity`, `Number.POSITIVE_INFINITY`, 第四章

---

## -`Infinity` 全局属性

一个表示负无穷大数字的常量

**有效性** Flash 5

**概要** -`Infinity`

**访问** 读/写

### 描述

在 ActionScript 中，任何覆盖负数允许数字范围的数字都用数字常量 `-Infinity` 来表示。在 ActionScript 中所允许的最小负数值（绝对值最大）是用 `-Number.MAX_VALUE` 来表示的，它等于 `-1.7976931348623157e+308`。

## 示例

超过最小允许负数（也就是更负）的计算结果就是 `-Infinity`。例如：

```
-Number.MAX_VALUE * 2;    // 产生 -Infinity
```

在将负数除以 0 的时候，也产生 `-Infinity`：

```
-1000 / 0;                // 产生 -Infinity
```

## 用法

`-Infinity` 是 `Number.NEGATIVE_INFINITY` 的缩写。

## 参见

`Infinity`, `Number.NEGATIVE_INFINITY`, 第四章

## Int()全局函数

去掉数字的小数部分

**有效性** Flash 4，在 Flash 5 中因为有了 *Math* 方法而得到增强

**概要** `int(number)`

**参数**

*number* 一个数字或者一个能产生数字的表达式，通常是一个带小数部分的数字。

**返回**

*number* 的整数部分。

**描述**

*int()* 函数在 Flash 4 中用作提取数字的整数部分的强制手段。它有效地对正数进行上舍入，对负数进行下舍入。*int()* 函数只对在 -2147483648 (- $2^{31}$ ) 到 2147483648 ( $2^{31} - 1$ ) 范围内的数字有效；在这个范围之外，它产生的结果为 *Undefined*。如果 *number* 是一个只包含数字的串，*int()* 会将串转换为数字再进行操作。如果 *number* 是布尔值 *true*，那么 *int()* 就返回结果 1。对于所有其他非数字数据（包括 *undefined* 和 *null*），*int()* 返回结果 0。

**用法**

因为支持更准确和标准的 *Math.floor()*, *Math.ceil()* 和 *Math.round()* 方法，*int()* 函数已经开始失宠了。用 *parseInt()* 或者 *Number()* 来将非数字数据转化为整数或者数字。

**示例**

```
int(4.5)      // 产生 4  
int(~4.5)     // 产生 -4  
int(3.999)    // 产生 3
```

用 *int()* 函数来检查一个数字是否为整数很有效，只要比较最初的数字值和 *int()* 函数的返回值就可以了：

```
if (int(x) != x) {  
    trace("Please enter a whole number for your age in years");  
}
```

**参见**

*Math.ceil()*, *Math.floor()*, *Math.round()*, *Number()*, *parseFloat()*, *parseInt()*

---

**IsFinite()全局函数**

检查一个数字是否小于 Infinity 并且大于 -Infinity

**有效性** Flash 5

**概要** `isFinite(number)`

**参数**

*number* 任何数字值或者能产生数字值的表达式

**返回**

一个布尔值。如果数字在 Number.MAX\_VALUE 和 -Number.MAX\_VALUE (包含在内) 之间就为 true, 否则为 false。如果 *number* 不属于数字类型, *number* 就会在 *isFinite()* 执行之前被转换为数字类型。

**描述**

*isFinite()* 函数只检查一个数字是否在 ActionScrit 合法的数字值范围之内。如果需要合法的数字才能保证计算正确, 可以在执行代码之前使用 *isFinite()*。

**示例**

```
if (!isFinite(x * y)) {           // 检查数字是不是在数字值范围内
    trace ('The answer is too large to display. Try again.');
}
isFinite(-2342434);              // 产生 true
isFinite(Math.PI);                // 产生 true
isFinite(Number.MAX_VALUE*2)      // 产生 false
```

**参见**

`-Infinity`, `Infinity`, `isNaN()`, `Number.MAX_VALUE`, `Number.MIN_VALUE`, 第四章

---

**isNaN()全局函数**

特殊 NaN 值的相等测试

**有效性** Flash 5

**概要** `isNaN(value)`

**参数**

*value* 要测试的表达式。

**返回**

一个布尔值。如果 *value* 是特殊数字值 NaN 就得到 true, 否则为 false。

## 描述

要测试一个值是否等于特殊数字值 NaN，必须使用 *isNaN()* 函数，因为 NaN 在一般的相等测试中都不等于它自身。例如，表达式：

```
NaN == NaN;
```

这将得到 false。*isNaN()* 函数通常用来检查代码段中是否有数学错误（比如 0 除以自身），或者在将一个值转换为合法数字的时候是否失败。因为 *isNaN()* 在表达式不是合法数字表达式的时候返回 true，所以可以经常把逻辑 NOT 操作符 (!) 和 *isNaN()* 一起使用（有的时候，非数字的非就是数字）。注意，0/0 产生 NaN，但是所有的正数除以 0 将产生 Infinity，所有的负数除以 0 将产生 -Infinity。

## 示例

```
// 设置一个值
var x = 'test123';
// 在用于数字表达式之前，检查 x 是否为合法的数字。
// 对于用户输入文本域来说这是一个简便的方法，它通常为串。
if (!isNaN(parseFloat(x))) {
    var y = parseFloat(x) * 2;
}
```

## 参见

*isFinite()*, *Nan*, 第四章

## Key 对象

确定键盘上键的状态

**有效性** Flash 5

**概要** *Key.property*

*Key.methodName()*

## 属性

表 R-7 列出了 *Key* 对象的属性。

表 R-7 Key 对象的键控代码属性

属性	等价键控代码	属性	等价键控代码
BACKSPACE	8	INSERT	45
CAPSLOCK	20	LEFT	37
CONTROL	17	PGDN	34
DELETEKEY	46	PGUP	33

表 R-7 Key 对象的键控代码属性（续）

属性	等价键控代码	属性	等价键控代码
DOWN	40	RIGHT	39
END	35	SHIFT	16
ENTER	13	SPACE	32
ESCAPE	27	TAB	9
HOME	36	UP	38

## 方法

- `getAscii()`      返回最近按键的 ASCII 值
- `getCode()`     返回最近按键的键控代码
- `isDown()`      检查指定的键当前是否被按下
- `isToggled()`    检查 Num Lock, Caps Lock 或者 Scroll Lock 是否被激活

## 描述

*Key* 对象用来确定当前哪一个键被按下，以及最近哪个键被按下。我们可以用它建造用键盘控制的界面，比如通过箭头按键来进行移动的太空船游戏。

因为不是所有的键盘都相同，因此，键盘控制界面的创建可以很灵活。但是，通过正确地选择脚本工具，我们可以确保所有的用户都有同样的切身体验。

探测键盘命令有两种常用的方法来：

- 通过 `isDown()` 方法来检查一个键现在是否被按下。这在不断需要键盘输入的时候比较适用，比如在视频游戏中。
- 用 `getCode()` 和 `getAscii()` 方法来检查哪个键是最近被按下的。对于典型的键盘驱动界面，也就是按键的时候要执行特定的操作，这种方法很不错。通常可以在 `keyDown` 事件处理器中使用这些方法，以区分不同的键。没有必要总检查（也就是轮询）最近按下的键，实际上，这么做会引起错误的操作重复，即使一个键并没有被重复按下，也就是说，应该只在 `keyDown` 事件处理器中检查 `getCode()` 和 `getAscii()`，因为处理器对于每一次按键动作只调用一次。

`getCode()` 所返回的，以及 `isDown()` 所要求的所谓 Windows 视频键控代码（或者简单称做键控代码）其实就是一个数字，它表示键盘上的物理键，而不是这些键上的符号。要使用键控代码，当影片在不同的操作系统上运行，或者两个键盘使用不同的语言或不同的符号设置时，我们都可以对键进行识别。

在大部分的键盘上，键 A 到 Z 的键控代码和等价大写 Latin 1 字母的编码点（65-90）是一样的。同样，键 0 到 9 和 Latin 1 中的这些数字值（48-57）是相等的。其他键的代码就和 Latin 1 编码点不匹配了。但是，许多非字母和非数字的键控代码作为 *Key* 的属性是可用的。例如，我们没有必要记住上箭头使用的键控代码是 38，我们只需要使用 *Key.UP* 属性。下面的代码检查上箭头现在是否被按下：

```
if (Key.isDown(Key.UP)) {  
    trace("The up arrow is being pressed");  
}
```

当我们处理的键控代码不是字母或者数字，也不是 *Key* 的属性时——比如那些功能键（F1, F2，等等）——那么创建一个快速测试影片来检查所需键的键控代码是最安全的，比如：

```
trace(Key.getCode());
```

键控代码列在附录二中。

## 参见

第十章，附录二

---

### Key.getAscii()方法

返回最近按键的 ASCII 值

**有效性** Flash 5

**概要** Key.getAscii()

**返回**

一个整数，表示最近按键的 ASCII 值。

**描述**

*getAscii()*方法返回最近按键的 ASCII 值。因为并不是所有的键都有 ASCII 值，因此，*getAscii()*通常用来探测 Latin 1 字符集（西欧语言）中的字母和数字。和 *getCode()*不同，*getAscii()*要分辨大小写字母。*getAscii()*不能分辨有相同 ASCII 值的两个键，比如主键盘上的 8 键和数字键盘上的 8 键，这一点也与 *getCode()*语言不同。

要探测和键盘物理位置相关而不是和 ASCII 值相关的特定键点按情况（比如，使用成菱形分布的四个键来控制游戏的播放是很恰当的），可以使用 *getCode()*。

**示例**

下面的例子表示一个简单的刽子手字游戏中的按键探测。它使用 *keyDown*事件处理器来识别键的点按情况，并且将该键添加到用户猜测列表：

```
onClipEvent (keyDown) {  
    var lastKey = Key.getAscii();  
    guessNum++;  
    userGuesses[guessNum] = String.fromCharCode(lastKey);  
}
```

## 参见

*Key.getCode()*, 附录二, 以及第十章

---

## Key.getCode()方法

返回最近按键的键控代码

**有效性** Flash 5

**概要** `Key.getCode()`

### 返回

一个整数, 表示最近所按键的键控代码。

### 描述

*getCode()*方法返回最近按键的键控代码, 它是一个任意的整数, 表示键在键盘上的物理位置。在非Windows操作系统中, 自然键控代码系统被Flash自动转换为Windows的等价物, 因此, *getCode()*提供了一个跨平台的途径来引用特定的按键。*getCode()*方法也可以用来区分有相同ASCII值的两个键。例如, 它可以区分主键盘上的8和数字键盘上的8, 但是 *getAscii()*就不行。但是, *getCode()*不能区分大小写 (例如, A 和 a 使用相同的键控代码, 因为它们是用相同的按键产生的)。

许多常用的键控代码值可以作为 *Key* 对象的属性来使用 (例如, *Key.UP*, *Key.BACKSPACE*)。要确定特殊按键的键控代码, 参见附录二, 或者按照下面的步骤来构造一个键控代码测试器:

1. 创建一个新的 Flash 文档。
2. 在主时间线的第 2 帧上, 添加一个帧。
3. 在帧 1 上, 添加下面的代码:  
`trace(Key.getCode());`
4. 选择 Control → Test Movie。
5. 点击影片的场景。
6. 按一个键, 这个键的键控代码就会出现在输出窗口中。

## 示例

和 *isDown()* 不同, *getCode()* 对于创造那种单个按键动作能产生单独的、直接的结果的界面来说是非常有用的。例如, 用户可以通过按空格来跳过影片的介绍内容。当空格键被按下的时候, 我们将播放头发送到影片的主界面(在主时间线上), 如下所示:

```
// intro 剪辑中的代码
onClipEvent (keyDown) {
    if (Key.getCode() == Key.SPACE){
        _root.gotoAndStop('mainInterface');
    }
}
```

## 参见

*Key.getAscii()*, *Key.isDown()*, 第十章、附录二

## Key.isDown()方法

检查特定的键现在是否被按下

**有效性** Flash 5

**概要** Key.isDown(keycode)

**参数**

*keycode* 一个数字, 表示所要检查键的键控代码。也可以是一个 *Key* 常量(例如, *Key.UP*, *Key.BACKSPACE*)。

**返回**

一个布尔值, 表示由 *keycode* 所指定的键是被按下(*true*), 还是没有被按下(*false*)。

**描述**

*isDown()* 方法告诉我们由 *keycode* 所指定的键现在是否被按下。它会对键盘的状态提供一个任意的、快速的访问, 最好使用在要求经常的按键输入或者要探测同步键点按情况的系统中。

*isDown()* 与 *getCode()* 和 *getAscii()* 相比有一个显著的优点, 就是它能够探测同步按下的多个键。例如, 检查 *Key.UP* 和 *Key.RIGHT*, 我们就可以确定游戏中的太空船应该按照对角方向移动。依靠所测试的特定键的位置, 同时探测的键最多可以达到三个。

## 示例

*isDown()* 方法通常用来创建随着每个帧而更新的系统。在下面的代码中, 我们在对应的箭头键被按下的帧中旋转和扩展太空船。注意, 如果需要同时探测两个键, 你应该使用单独的 *if* 语句。在本例中, 右箭头键的状态在左箭头被按着的情况下就会被忽略。

但是不管如何，上箭头键的状态总是在单独的 *if* 语句中进行检查的。太空船示例的版本可以从在线代码库中找到：

```
// spaceship 剪辑上的代码
onClipEvent (enterFrame) {
    if (Key.isDown(Key.LEFT)) { // 左箭头
        _rotation += 10;
    } else if (Key.isDown(Key.RIGHT)) { // 右箭头
        _rotation -= 10;
    }
    if (Key.isDown(Key.UP)) { // 上箭头
        thrust += 10;
    }
}
```

## 参见

*Key.getCode()*, 第十章

---

## Key.isToggled()方法

检查 Caps Lock、Num Lock 或者 Scroll Lock 键是否被激活

**有效性** Flash 5

**概要** Key.isToggled(keycode)

**参数**

*keycode* 一个整数键控代码，通常是 Caps Lock 键 (20), Num Lock 键 (144), Scroll Lock 键 (145)。也可以是键常量 *Key.CapsLock*。

**返回**

一个布尔值，表示由 *keycode* 所指定的键是开启 (*true*)，还是关闭 (*false*)。

**描述**

*isToggled()* 方法探测特定的 Caps Lock、Num Lock 或者 Scroll Lock 键的状态。和其他的键不同，这些键有一个“开”状态和一个“关”状态，表示它们的功能是否被激活。*isToggled()* 的返回值告诉我们键的功能是否有效。(虽然 *isToggled()* 实际上对任何键控代码都有效，但是它的返回值只对支持开关功能的特殊键有用。要探测其他键的状态，使用 *isDown()*、*getCode()* 或者 *getAscii()*。)

---

## \_leveln 全局属性

播放器中的文档层级

**有效性** Flash 3 以及以后的版本

**概要**            `_level0`  
                  `_level1`  
                  `_level2`  
                  ...  
                  `_leveln`

**访问**            只读

### 描述

我们可以将多个.swf文件导入到Flash播放器进行同步显示。每一个载入的.swf都留在文档层级栈中自己的层上。(一个层级较高的.swf文件如果和层级较低的文件占据了相同的场景位置，前者会遮盖后者。) `_leveln` 属性存储了一个载入到播放器层级中的.swf文件的主时间线的引用。每一个文档层级都是用一个数字属性来表示的，比如 `_level0`, `_level1`, `_level2`, 依此类推。

任何Flash播放器中最初装载的文档都被当作 `_level0`。

### 示例

一个 `_leveln` 引用通常用来控制文档栈中其他层级上的影片。例如，我们播放第 2 层中的影片：

```
_level2.play();
```

我们也可以将 `_leveln` 同影片剪辑引用结合使用，控制文档栈中任何层级影片中所包含的剪辑。例如：

```
_level1.orderForm.titleBar.play();
```

一个 `_leveln` 引用也可以用做多个函数的 `target` 参数值，包括 `loadMovie()`, `unloadMovie()`, `loadVariables()` 和 `print()`。如果该层还不存在，你可以在引号内指定层级的引用。如果没有使用引号，一个不存在的层级就被当作 `undefined`，可以让命令在当前时间线上，而不是在新的、未定义的层级上操作。例如，从 `_level0` 的主时间线上执行的时候，如果 `_level1` 还没有定义，下面的代码会代替 `_level0` 上的影片：

```
loadMovie('myMovie.swf', _level1);
```

如果你不能确保一个层级是否已经存在，那么使用下面的代码比较安全：

```
loadMovie('myMovie.swf', '_level1'); // 即使 _level1 不存在该命令也有效
```

当然，从其他的层中，你可以引用原先的层级，使用 `_level0`，如下所示：

```
startDrag(_level0, true);
```

## 参见

*loadMovie()*, *unloadMovie()*, *\_root*, 第十三章

---

## loadMovie()全局函数

将一个外部的.swf文件载入到播放器

**有效性** Flash 4 以及以后的版本。Flash 5 中的 *loadMovie()* 函数符合于 Flash 4 中带目标路径的 *Load Movie*。

### 概要

```
loadMovie(URL, target)  
loadMovie(URL, target, method)
```

### 参数

*URL* 一个用来指定到要装载的外部.swf文件的绝对或者相对路径。所有的 URL 必须使用正斜杠，绝对 URL 必须包含 http:// 或者 file:/// 协议引用。

*target* 一个表示要容纳外部.swf文件的影片剪辑或者文档层级的串。也可以是一个到已经存在的影片剪辑或文档层级的引用（引用在使用于串语境中的时候会转换为路径）。

*method* 一个可选的串，表示将变量发送给外部脚本所要使用的方法。合法的 method 值为 "GET" 或者 "POST"。该参数必须是一个直接量，而不是一个变量或者其他表达式。独立的 Flash 播放器版本总是使用 "GET" 方法，而不管指定的是什么 method。

### 描述

*loadMovie()* 函数将 URL 所定位的文件导入到 Flash 播放器。

如果 target 是一个对已经存在影片剪辑的引用，或者是一个表示到影片剪辑的路径的串，装载的.swf文件就被放到指定的剪辑（会清除以前的任何内容）。要将一个影片装载到当前影片剪辑中，可以使用空串来作为 target 参数，比如：

```
loadMovie("myMovie.swf", "")
```

如果 target 是一个对现有文档层级的引用（比如 \_level2）或者是一个表示文档层级路径的串（比如 "\_level2"），那么.swf文件就被放置到指定的文档层级。将一个影片装载到 \_level0 中将清除所有的播放内容，将新的.swf文件放到 \_level0 中。

在 *loadMovie()* 的调用中可以发送变量，这种情况下，URL 通常是一个脚本的位置，该脚本基于所发送的变量返回一个.swf文件。要在 *loadMovie()* 调用中发送变量，可以包括 *method* 参数（设置为“GET”或者“POST”）。“GET”将当前的影片剪辑时间线变量作为属于 URL 的问询串来发送。“POST”将当前影片剪辑时间线上的变量放在 HTTP POST 请求标头后面进行发送。“POST”方法在独立的Flash播放器中是无效的。因为大部分Web服务器严格地将URL的长度限制在255和1024个字符之内，所以用“POST”来代替“GET”可传送更多的数据内容。

通过 Web 服务器，使用“GET”方法的 *loadMovie()* 调用可以将变量传递给一个载入的影片，而不需要其他脚本的帮助。下面，我们将外部影片 *myMovie.swf* 装载到播放器文档栈中的层级 1，从当前时间线向它传递变量：

```
loadMovie("myMovie.swf", "_level1", "GET");
```

传递给装载影片的变量是在影片的主时间线上定义的。这个技术只当 *loadMovie()* 的请求要通过 Web 服务器来处理的时候才有效。如果用局部文件将“GET”方法使用在 *loadMovie()* 中，会引发一个“URL 打开错误”的错误。

## 用法

将影片剪辑和层级引用用做 *loadMovie()* 的 *target* 参数的时候要小心。如果 *loadMovie()* 的 *target* 参数产生 *undefined*，那么 *loadMovie()* 函数就将当前的时间线作为它的 *target*。同样，产生空串的 *target* 引用会让 *loadMovie()* 在当前时间线上进行操作，特别是当影片装载到新的、未被占据的层级中时会出现问题。考虑下面的代码：

```
loadMovie("myMovie.swf", _level1);
```

如果在语句执行之前不存在 *\_level1* 对象，代码就会将 *myMovie.swf* 装载到包含 *loadMovie()* 语句的时间线中，而不是 *\_level1*！要避免这个问题，可以使用 *loadMovieNum()*。另外，也可以将一个串作为 *loadMovie()* 的 *target* 参数，比如：

```
loadMovie("myMovie.swf", "_level1");
```

在这种情况下，如果层级不存在就对其进行创建（在所有的影片中只默认 *\_level0* 是存在的）。关于更多的信息，参见第十三章。

## 示例

```
loadMovie("myMovie.swf", "_level1"); // 将 myMovie.swf 放在层级 1  
loadMovie("myMovie.swf", "_level0"); // 将 myMovie.swf 放在层级 0  
loadMovie("myMovie.swf", "myClip"); // 将 myMovie.swf 放到 myClip 中
```

```
// 用coolmovie.swf来替代播放器的内容，使用绝对路径  
loadMovie("http://WWW.yourflashtite.com/coolmovie.swf", "_level0");  
// 将影片从Windows桌面装载到层级1。注意，  
// file:///协议和正斜杠  
loadMovie('file:///C:/WINDOWS/Desktop/animation.swf', '_level1');
```

## 参见

*loadMovieNum()*, *MovieClip.loadMovie()*, *unloadMovie()*, 第十三章

## **loadMovieNum()全局函数**

将一个外部的.swf文件装载到文件层级中

### 有效性

Flash 3，在Flash 4中使用了*method*参数而得到加强，在Flash 5中可用。*loadMovieNum()*函数和Flash 3的*Load Movie*相当，后者只接受层级号码。

### 概要

*loadMovieNum(URL, level)*

*loadMovieNum(URL, level, method)*

### 参数

#### *URL*

一个串，指定要装载的外部.swf文件的相对或绝对路径。

#### *level*

一个非负整数，或者能够产生一个非负整数的表达式，表示容纳外部.swf文件的文档层级。

#### *method*

一个可选的串，表示用来将变量发送到外部脚本的方法。*method*的合法值为"GET"和"POST"。这个参数必须是常量，而不能是变量或其他表达式。独立的Flash播放器版本总是使用"GET"方法，不管指定的是什么*method*。

### 描述

*loadMovieNum()*函数几乎和*loadMovie()*相同，差别在于它要求装载操作的目标*level*要作为数字定义，而不是串。这意味着*loadMovieNum()*只能将影片装载到文档层级，而不是主剪辑。如果指定的层级不存在，可以创建。如果指定的层级的确存在，它的占据者就会被新的.swf文件所代替。将影片装载到\_level2中，即使\_level1还没有创建也是合法的。

当我们希望动态设置装载影片的层级时可以使用*loadMovieNum()*函数，比如：

```
var x = 3;  
loadMovieNum("myMovie.swf", x);
```

这个效果也可以通过用正规 *loadMovie()* 函数和一个串连接表达式来实现：

```
loadMovie('myMovie.swf', '_level'+x);
```

## 参见

*loadMovie()*, *MovieClip.loadMovie()*

---

## loadVariables()全局函数

获取一系列外部变量

**有效性** Flash 4 以及以后的版本

**概要** `loadVariables(URL, target)`

`loadVariables(URL, target, method)`

### 参数

**URL** 一个串，指定变量来源的路径——返回变量的服务器端脚本或者包含变量的文本文件。

**target** 一个串，表示影片剪辑的路径，或者装载变量要被定义的文档层级。也可以是一个影片剪辑或者文档层级的引用（引用使用于一个串语境的时候会转换为路径）。

**method** 一个可选的串，表示用来将变量发送到外部脚本的方法。如果指定了，那么当前时间线的变量就被发送到脚本，*target*接收装载的变量。如果遗漏，变量虽然可以被接收但是却不发送。*method*的合法值为“GET”和“POST”。这个参数必须是直接量，而不能是变量或其他表达式。独立的 Flash 播放器版本总是使用“GET”方法，不管指定的是什么 *method*。

### 描述

通常，我们在影片里使用 ActionScript 来定义变量。但是，使用 *loadVariables()*，我们也可以将变量从文本文件或者服务器端应用程序（比如 Perl 脚本）导入到影片中。通过 *loadVariables()* 装载的变量，其作用域是影片剪辑或者由 *target* 所指定的层级，它总是串数据类型。要将装载的变量添加到当前时间线，可用空串来作为 *target* 参数的值。例如：

```
loadVariables('myVars.txt', ""); // 从 myVars.txt 将变量装载到当前时间线
```

要装载的变量不管是在文本文件中还是在脚本中，它们的格式都必须按照 URL 编码的规则，如下所示：

- 每一个变量名都要将值和赋值号分开，不使用空格，比如 `firstName=stephen`。
- 多个变量名 / 值对必须用 & 符号分开，比如 `firstName=stephen&lastName=burke`。
- 空格必须用 + 符号来代替。
- 任何非空格、非数字（1-9）的字符，或者不被接受的 Latin1 字母（a-z, A-Z）应该用格式为 %xx 的十六进制转义序列来代替，xx 是字符的十六进制 Latin1 编码点。

例如，下面的代码显示了要通过 `loadVariables()` 导入到 Flash 中的文本文件的内容。导入的变量是 `name` 和 `address`，它们的值分别为 "stephen" 和 "65 nowhere st!"：

```
name=stephen&address=65+nowhere+st%21
```

用于 `loadVariables()` 的文本文件是简单的正规文本文件，包含 URL 编码的变量，如前所示。要从外部文本文件装载变量，我们将文件的路径指定为 `loadVariables()` 函数调用中的 *URL* 参数。例如：

```
// 从 myVariables.txt 将变量装载到主影片时间线  
loadVariables("myVariables.txt", '_root');
```

`loadVariables()` 也可以用于输出 URL 编码变量的脚本或者服务器应用程序。当一个脚本将数据发送到和 `loadVariables()` 函数对应的 Flash 影片时，脚本会将数据的 MIME 类型设置为："application/x-www-form-urlencoded"。下面是一个 Perl 脚本中典型的 MIME 设置语句：

```
print "Content-type: application/x-www-form-urlencoded\n\n";
```

虽然名称 `loadVariables()` 暗示变量的传输方向只有一个，但是它也可以用来将变量发送到服务器端的脚本。要将所有定义在当前时间线上的变量发送到脚本，我们将 `loadVariables()` 函数调用的 *method* 参数设置为 "GET" 或者 "POST"。变量按照 URL 编码格式发送。如果 *method* 被设置为 "GET"，变量就作为脚本 *URL* 的问询串来发送；如果 *method* 被设置为 "POST"，变量就在 HTTP POST 请求标头之后进行发送。"POST" 方法在独立的 Flash 播放器中是不可用的。因为大部分 Web 服务器都严格地将 URL 的长度限制在 255 和 1024 个字符以内，所以，可用 "POST" 来代替 "GET" 进行大量的数据传输。

出于安全考虑，`loadVariables()` 只对装载影片的主机定义域有效。表 R-8 列出了控制 `loadVariables()` 用法的规则。这些安全测试只影响 Flash 播放浏览器插件和 ActiveX 控制，变量可以从独立播放器的任何定义域得到装载。

表 R-8 基于定义域的 loadVariables() 的安全限制

影片来源的定义域	连接的主机	是否许可
www.somewhere.com	www.somewhere.com	Yes
www.somewhere.com	other.somewhere.com	Yes
www.somewhere.com	www.somewhere-else.com	No
www.somewhere.com	somewhere.com	Yes
somewhere.com	www.somewhere.com	Yes

定义域限制是 Flash 的一个内部安全功能，但是，它可以被一个运行在站点 X 上作为 Flash 和站点 Y 的中介的代理脚本，或者一个在站点 X 上指向站点 Y 的 DNS 信号所废除。有关更多的信息，参见：

[http://www.macromedia.com/support/flash/ts/documents/loadvars\\_security.htm](http://www.macromedia.com/support/flash/ts/documents/loadvars_security.htm)

## 用法

对相同脚本 URL 的多个 *loadVariables()* 调用的结果，可以隐藏在一些从来没有从服务器装载新数据的浏览器上。要避免这个问题，可以对每一个 *loadVariables()* 调用添加一个虚拟变量，这样，URL 就是惟一的。例如，下面我们通过添加毫秒时间来产生独特的 URL：

```
loadVariables("http://www.mysite.com/cgi-bin/myScript.pl?cacheKiller=" + getTimer(), serverResponse);
```

## 故障

Macintosh 上的 IE4.5 不支持 POST 方法。这个问题在浏览器的版本 5 中得到了修正。

## 参见

*LoadVariables Num()*, *MovieClip.load Variables()*, 第十七章

## loadVariablesNum() 全局函数

将一系列外部变量添加到一个文档层级

### 有效性

Flash 5，在 Flash 4 中用装载变量动作来将变量放到文档层级中。

### 概要

`loadVariablesNum(URL, level)`

`loadVariablesNum(URL, level, method)`

### 参数

#### URL

一个串，指定变量来源的路径——返回变量的服务器端脚本或者包含变量的文本文件。

---

<i>level</i>	一个非负整数，或者能够产生一个非负整数的表达式，表示定义装载变量的文档层级。
<i>method</i>	一个可选的串，表示用来将变量发送到外部脚本的方法。如果指定了，那么当前时间线的变量就被发送到脚本， <i>level</i> 接收装载的变量。如果遗漏，变量虽然可以被接收但是却不发送。 <i>method</i> 的合法值为 "GET" 和 "POST"。这个参数必须是直接量，而不能是变量或者其他表达式。独立的 Flash 播放器版本总是使用 "GET" 方法，不管指定的是什么 <i>method</i> 。

## 描述

*loadVariablesNum()* 函数几乎和 *loadVariables()* 相同，差别在于它要求装载操作的目标 *level* 要作为数字定义，而不是串。这意味着 *loadVariablesNum()* 只能将影片装载到文档层级，而不能装载到影片剪辑。目标 *level* 可以动态指定，比如：

```
var myLevel = 2;  
loadVariablesNum("myVars.txt", myLevel);
```

类似的效果也可以通过用正规 *loadVariables()* 函数和一个串连接表达式来实现：

```
loadVariables("myVars.txt", "_level" + myLevel);
```

## 参见

*loadVariables()*

---

## Math 对象

访问数学函数和常量

**有效性** Flash 5

**概要** `Math.propertyName`  
`Math.methodName()`

## 属性

*E* 常量 *e*，自然对数的底数，大约为 2.71828。

*LN10* 10 的自然对数 ( $\log_e 10$ )，大约为 2.30259。

*LN2* 2 的自然对数 ( $\log_e 2$ )，大约为 0.69315。

*LOG10E* 以 10 为底的 *e* 的对数，大约为 0.43429。

*LOG2E* 以 2 为底的 *e* 的对数，大约为 1.44270。

*PI* 圆的周长和直径的比率，大约为 3.14159。

*SQRT1\_2* 2 的平方根的倒数，大约为 0.70711。

*SQRT2* 2 的平方根，大约为 1.41421。

## 方法

*abs()* 计算一个数字的绝对值。

*acos()* 计算一个数字的反余弦。

*asin()* 计算一个数字的反正弦。

*atan()* 计算一个数字的反正切。

*atan2()* 计算一个点的角度，和 x 轴相关。

*ceil()* 将一个数字四舍五入为一个整数。

*cos()* 计算一个角度的余弦。

*exp()* 将 e 增加指定次幂。

*floor()* 对输入的数字返回小于或者等于它的最接近的整数。

*log()* 计算一个数字的自然对数。

*max()* 确定两个数字中的较大数字。

*min()* 确定两个数字中的较小数字。

*pow()* 将一个数字增大指定次幂。

*random()* 获取 0 和 1 之间的一个随机浮点数。

*round()* 计算一个数字的最接近整数。

*sin()* 计算一个角度的正弦。

*sqrt()* 计算一个数字的平方根。

*tan()* 计算一个角度的正切。

## 描述

*Math* 对象提供了对内置数学函数（通过方法的访问）和常量值（通过属性的访问）的访问。这些函数和常量可以相对容易地执行一些潜在复杂的计算。

注意，*Math* 对象的属性和方法可以用在导出为 Flash 4 格式的影片中，在这种情况下，Flash 会作近似计算。结果值为合理的近似值，但是没有必要和原来的 Flash 5 函数相同。Flash 4 的值对于精密的应用程序，比如图形显示来说已经非常精确了，但是对于精确的财政或者工程计算则是不够的。

注意，三角函数要求的角度要用弧度来表示，但是Flash中*MovieClip.\_rotation*属性却是按照角度来度量的。在一个圆周中有 $2\pi$ 弧度（一个半径大约为57.3度）。要从弧度转换为角度，用下面的公式：

```
degrees = (radians / Math.PI) * 180;
```

要从角度转换到弧度，使用公式：

```
radians = (degrees / 180) * Math.PI;
```

## 参见

*Math.atan2()*, *Math.cos()*, 第四章

---

## Math.abs()方法

计算一个数字的绝对值

**有效性** Flash 5，在输出 Flash 4 影片的时候也可以使用。

**概要** `Math.abs(x)`

**参数**

*x* 一个正数或者负数。

**返回**

*x* 的绝对值（一个表示 *x* 数量值的正数）。

**描述**

*abs()*方法计算 *x* 和 0 之间的距离（也就是 *x* 的绝对值）。它对正数不作改变，但是将负数转换为相同数值的正数。它适用于计算两个数字之间的差别，而不比较其大小。例如，当我们计算两个点之间的距离时它就很适用，因为距离总是一个正数。

**示例**

```
Math.abs(-5);           // 返回 5  
  
// 计算两个数字之间的差别  
function diff (num1, num2){  
    return Math.abs(num1-num2);  
}  
  
diff(-5, 5);           // 返回 10
```

---

## Math.acos()方法

计算一个数字的反余弦

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要**      `Math.acos(x)`

**参数**

*x*      一个在 -1.0 到 1.0 之间的数字（角度的余弦）。

**返回**

一个用弧度表示的角度。它的余弦值为 *x*。如果 *x* 不在 -1.0 到 1.0 的范围内，那么就返回 `NaN`。

**描述**

反余弦函数（有的时候写作  $\cos^{-1}$ ）是余弦函数的反转。它返回余弦为给定值的角度，用弧度来表示。返回的值在 0 到  $\pi$ （也就是 3.14159...）的范围内。

**示例**

```
trace(Math.acos(1.0)); // 显示: 0  
trace(Math.acos(0.0)); // 显示: 1.5707... (也就是 pi/2)
```

**参见**

`Math.asin()`, `Math.atan()`, `Math.cos()`

---

**Math.asin()方法**

计算一个数字的反正弦

**有效性**      Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要**      `Math.asin(x)`

**参数**

*x*      一个在 -1.0 到 1.0 之间的数字（角度的正弦）。

**返回**

一个用弧度表示的角度。它的正弦值为 *x*。如果 *x* 不在 -1.0 到 1.0 的范围内，那么就返回 `NaN`。

**描述**

反正弦函数（有的时候写作  $\sin^{-1}$ ）是正弦函数的反转。它返回正弦为给定值的角度，用弧度来表示。返回的值在  $-\pi/2$  到  $\pi/2$  的范围内。

**示例**

```
trace(Math.asin(1.0)); // 显示: 1.5707... (也就是 pi/2)  
trace(Math.asin(0.0)); // 显示: 0
```

**参见**

*Math.acos()*, *Math.atan()*, *Math.sin()*

**Math.atan()方法**

计算一个数字的反正切

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.atan(x)`

**参数**

*x* 一个在 -Infinity 到 Infinity 之间的数字, 包括一些角度的正切。

**返回**

一个用弧度表示的角度。它的正切值为 *x*。

**描述**

反正切函数 (有的时候写作  $\tan^{-1}$ ) 是正切函数的反转。它返回正切为给定值的角度, 用弧度来表示。返回的值在  $-\pi/2$  到  $\pi/2$  的范围内。

**示例**

```
trace(Math.atan(1.0));           // 显示: 0.78539...
trace(Math.atan(0.0));           // 显示: 0
trace(Math.atan(-Infinity));    // 显示: -1.5707... (也就是 -pi/2)
```

**参见**

*Math.acos()*, *Math.asin()*, *Math.tan()*

**Math.atan2()方法**

基于一个点来确定一个角度

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.atan2(y, x)`

**参数**

*y* 点的 y 坐标。

*x* 点的 x 坐标。

**返回**

一个用弧度表示的从圆心到点 (*x*, *y*) 的角度, 从圆的正水平轴 (也就是 *x* 轴) 按照反时针的方向来度量。范围从  $-\pi$  到  $\pi$ 。(负值表示角度在 *x* 轴下方)。

## 描述

*atan2()*方法，和*atan()*一样，执行一个反正切计算，但是使用x和y轴坐标来作为参数，而不是它们的比率。也就是说，用*atan2()*来计算一个反正切如下所示：

```
Math.atan2(9, 3); // 产生 1.24904577239825
```

它与使用9/3（或3）的比率用*atan()*来计算反正切效果是一样的：

```
Math.atan(3); // 同样的
```

## 用法

注意，y坐标是作为第一个参数传递到*atan2()*的，而x坐标是第二个参数。这是特意要求的。它反映了正切的结构，它是一个角度的对边（y）除以角度的邻边（x）。

## 示例

*atan2()*方法可以用来制作一个影片剪辑，指向一个移动的目标。下面的例子可以在在线代码库中找到，它将把当前的剪辑朝鼠标指示器方向旋转，用来使敌方太空船对着玩家的太空船：

```
// 将影片剪辑朝着鼠标旋转
onClipEvent (load) {
    // 将弧度转换为角度，每360角度为2*pi弧度
    function radiansToDegrees(radians) {
        return (radians/Math.PI) * 180;
    }
}

onClipEvent (enterFrame) {
    // 创建一个point 对象、将该剪辑和其父记录点相关的x和y坐标存储起来
    point = {x:_x, y:_y};
    // 将我们的局部（父）坐标转换为全局（场景）坐标
    _parent.localToGlobal(point);
    // 计算剪辑的记录点同鼠标之间的距离
    deltaX = _root._xmouse - point.x;
    deltaY = _root._ymouse - point.y;
    // 计算从剪辑记录点到鼠标的角度
    rotationRadian = Math.atan2(deltaY, deltaX);
    // 将弧度值转换为角度值
    rotationAngle = radiansToDegrees(rotationRadian); // 参见前边的函数
    // 将该剪辑的旋转角度转到指向鼠标的位置
    this._rotation = rotationAngle;
}
```

---

## Math.ceil()方法

将一个数字四舍五入为下一个整数

**有效性**

Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要**      `Math.ceil(x)`

**参数**

*x*            一个数字。

**返回**

大于或等于 *x* 的下一个整数。

**描述**

*ceil()*(也就是加顶) 方法将一个浮点数转换为第一个大于等于 *x* 的整数。

**示例**

```
Math.ceil(1.00001);        // 返回 2  
Math.ceil(5.5);            // 返回 6  
Math.ceil(-5.5);          // 返回 -5
```

**参见**

*Math.floor()*, *Math.round()*

---

## Math.cos()方法

计算一个角度的余弦

**有效性**      Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要**      `Math.cos(theta)`

**参数**

*theta*            一个角度, 以弧度 (不是角度) 表示, 范围从 0 到  $2\pi$ 。

**返回**

*theta* 的余弦值 (结果在范围 -1.0 到 1.0 之间)。

**描述**

*cos()* 函数返回一个角度的三角余弦。在一个直角三角形中, 一个角的余弦是用角的斜边边长来除邻边边长所得到的值。

**用法**

注意, *cos()* 中的角度是用弧度而不是角度来表示的。

**示例**

```
trace(cos(0));            // 显示: 1.0  
trace(cos(Math.PI));     // 显示: -1.0
```

*cos()*函数可以和*sin()*一起使用，计算圆周上的一个点。在下面的例子中，我们要使用这个技巧将影片剪辑顺着圆周路径移动。给定圆的半径 $r$ ，以及角度 $\theta$ ，该角度从正水平轴开始按反时针方向计量，一个点的位置为( $r\cos\theta, r\sin\theta$ )：

```
// 帧1中的代码
var radius = 100;           // 圆周路径的半径
var centerX = 275;          // 圆周路径的水平中心
var centerY = 200;          // 圆周路径的垂直中心
var rotAngleDeg = 0;         // 对象的角度，以角度数值表示，
                            // 从水平轴(x轴)开始以反时针方向计量
var rotAngRad;              // rotAngleDeg的弧度值

// 将角度转换为弧度。每360度有2*pi弧度
function degreesToRadians(degrees) {
    return (degrees/180) * Math.PI;
}

// 帧2上的代码
// 将旋转角度增加5
rotAngleDeg += 5;

// 放置对象。注意，Flash的Y轴是和笛卡儿坐标相反的。
// 因此我们要对y实施一定的减法来获得新的位置
rotAngRad= degreesToRadians(rotAngleDeg);
ball._x = centerX+Math.cos(rotAngRad) * radius;
ball._y = centerY-Math.sin(rotAngRad) * radius;

// 帧3上的代码
// 回到帧2再次移动球
gotoAndPlay(2);
```

## 参见

*Math.acos()*, *Math.sin()*, *Math.tan()*

---

## Math.E 属性

常量 e (自然对数的底数)

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.E`

### 描述

E 属性存储一个自然对数底数的近似值（大约为 2.71828），在数学中是用符号  $e$  来表示的。它是一个先验的数字，和  $\pi$  一样，用在包含增长或者变化的数学方程式中。不要将它同第四章“浮点数直接量”中所讨论的指数符号混淆起来，两者之间并没有什么关系。

**参见**

*Math.log()*, *Math.LN10()*, *Math.LN2()*

**Math.exp()方法**

对常数 e 进行给定的幂次运算

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.exp(x)`

**参数**

*x* 表示 `Math.E` 的幂次。

**返回**

`Math.E` 的 *x* 次方。

**Math.floor()方法**

将一个数字下舍入为前边的整数

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.floor(x)`

**参数**

*x* 一个数字。

**返回**

一个小于或等于 *x* 并与其最靠近的整数。

**描述**

*floor* 方法将一个浮点数转换为一个小于或等于 *x* 的整数。

**示例**

```
Math.floor(1.99999);      // 返回 1
Math.floor(5.5);          // 返回 5
Math.floor(-5.5);         // 返回 -6

function minutesToHMM (minutes) {
    var hours = Math.floor(minutes/60);
    minutes -= hours*60;
    minutes = minutes < 10 ? '0' + minutes : minutes;
    return hours + ":" + minutes;
}
```

**参见**

*Math.ceil()*, *Math.round()*

---

**Math.LN10 属性** 10 的自然对数( $\log_e 10$ )，大约为 2.30259

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**摘要** Math.LN10

**描述**

LN10 属性表示 10 的自然对数（以  $e$  为底 10 的对数），一个大约等于 2.30259 的常量。

---

**Math.LN2 属性** 2 的自然对数( $\log_e 2$ )，大约为 0.69315

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**摘要** Math.LN2

**描述**

LN2 属性表示 2 的自然对数（以  $e$  为底 2 的对数），一个大约等于 0.69315 的常量。

---

**Math.log()方法** 计算一个数字的自然对数

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**摘要** Math.log(x)

**参数**

x 一个正整数。

**返回**

$x$  的自然对数。

**描述**

*log()* 方法计算一个数字的自然对数值（也就是以  $e$  为底的对数）。参见下面计算以 10 为底的对数的例子。

**示例**

```
trace (Math.log(Math.E));           // 显示: 1

// 计算一个数字的以 10 为底的对数
function log10 (x) {
    return (Math.log(x) / Math.log(10));
}
```

---

**Math.LOG10E 属性**

以 10 为底的  $e$  的对数，大约为 0.43429

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** Math.LOG10E

**描述**

LOG10E 属性表示  $e$  的一般对数（以 10 为底  $e$  的对数），一个大约等于 0.43429 的常量。

---

**Math.LOG2E 属性**

以 2 为底的  $e$  的对数，大约为 1.44270

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** Math.LOG2E

**描述**

LOG2E 属性表示以 2 为底  $e$  的对数 ( $\log_2 e$ )，一个大约等于 1.44270 的常量。

**故障**

在 Flash 5r30 中，LOG2E 会错误地返回 LN2 的值 (0.69315)。

---

**Math.max()方法**

确定两个数中的较大者

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** Math.max(*x*, *y*)

**参数**

*x* 一个数字。

*y* 一个数字。

**返回**

*x* 和 *y* 中的较大者。

**示例**

```
Math.max(5, 1); // 返回 5  
Math.max(-6, -5); // 返回 -5
```

这个例子要用一个值来确定范围；

```
function constrainToRange (checkVal, minValue, maxValue) {  
    return Math.min(Math.max(checkVal, minValue), maxValue);  
}
```

```
// 对场景区域使用滑块
mySlider._x = constrainToRange (mySlider._x, 0, 550);
```

这个例子返回数组中的最大值：

```
function maxInArray (checkArray) {
    maxVal = -Number.MAX_VALUE; // 将maxVal 初始化为一个非常小的数字
    for (var i = 0; i < checkArray.length; i++) {
        maxVal = Math.max(checkArray[i], maxVal);
    }
    return maxVal;
}

trace(maxInArray([2,3,66,4,342,-90,0])); // 显示: 342
```

## 参见

*Math.min()*

## Math.Min()方法

确定两个数中的较小者

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.min(x, y)`

### 参数

*x* 一个数字。

*y* 一个数字。

### 返回

*x* 和 *y* 中的较小者。

### 示例

```
Math.min(5, 1); // 返回 1
Math.min(-5, -6); // 返回 -6
```

读者练习：修改 *Math.max()* 下的例子返回一个数组中的最小值而不是最大值。

## 参见

*Math.max()*

## Math.PI 属性

圆周长同直径的比率，大约为 3.14159

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要**      `Math.PI`**描述**

PI 属性表示常量  $\pi$ ，也就是圆的周长同直径的比率。

**示例**

PI 是计算圆面积时最重要的元素：

```
function circleArea (radius) {  
    // PI 乘以半径的平方可以写成 Math.PI * Math.pow(radius, 2)  
    return Math.PI * (radius * radius);  
}
```

**Math.pow()方法**

对给定的数字进行指定的幂次运算

**有效性**      Flash 5，在导出 Flash 4 影片的时候也可以使用。**概要**      `Math.pow(base, exponent)`**参数**

*base*      一个数字，表示指数表达式的底数。

*exponent*      一个数字，表示 *base* 的幂次（也就是指数）。

**返回**

*base* 的 *exponent* 次幂。

**描述**

*pow()* 方法可以用来对任何数字进行幂次运算。如果 *exponent* 是负数，那么 *pow()* 返回  $1 / (\text{base}^{\text{abs(exponent)}})$ 。如果 *exponent* 为分数，那么 *pow()* 可以用来计算数字的平方根或立方根（在这种情况下，它返回的根为正数，虽然在数学上也有负的根）。

**示例**

```
Math.pow(5, 2);      // 返回 25 (5 的平方)  
Math.pow(5, 3);      // 返回 125 (5 的立方)  
Math.pow(5, -2);      // 返回 0.04 (1 除以 25)  
Math.pow(8, 1/3);      // 返回 2 (8 的立方根)  
Math.pow(9, 1/2);      // 返回 3 (9 的平方根)  
Math.pow(10, 6);      // 返回 1000000 (可以写成 1e6)
```

**故障**

Flash 5 播放器的修订版 30 不能正确地计算 *base* 为负数的 *Math.pow()* 值。例如，`Math.pow(-2, 2)` 结果为 NaN，而它的正确值应该是 4。

**参见**

*Math.exp()*, *Math.sqrt()*, *Math.SQRT2*, *Math.SQRT1\_2*

**Math.random()方法**

产生一个从 0 到 1.0 的随机数字

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.random()`

**返回**

一个大于或等于 0.0 但是小于 1.0 的浮点数字。

**描述**

*random()*方法为生成随机数字提供了一个途径，它可以用来自随机选择脚本中的动作。*random()*方法生成一个随机值，在 0 到 .99999... 之间，两头的数字包含在内。我们可以按照需要来缩放这个范围。例如，要获得一个 0 到 5 之间的随机数，可以使用：

```
Math.floor(Math.random() * 6)
```

要获得一个 1 到 6 之间的随机数，可以使用：

```
Math.floor(Math.random() * 6) + 1
```

下面这个自定义的函数返回一个特定范围内的整数，而不是从 0 到 1 的浮点数：

```
// 返回一个从 minVal 到 maxVal 之间的数字，两者包含在内
function myRandom (minVal, maxVal) {
    return minVal + Math.floor(Math.random() * (maxVal + 1 - minVal));
}

// 调用函数
dieRoll = myRandom(1, 6);      // 模拟一个六面模型
trace(dieRoll);

// 注意，要模拟两个骰子，可以用：
twoDice = myRandom(2, 12);     // 最小的值为 2，不是 1

// 要单独返回各面的值，使用一个数组
function rollTwoDice () {
    return [myRandom(1, 6), myRandom(1, 6)];
}
```

因为 Flash 5 播放器的修正版 30 中的故障，这个方法有潜在的少见但是重大的错误倾向。在修正版 30 中，*random()*生成 0.0 到 1.0 范围内的值，包括 0.0 和 1.0。当将 *random()* 的返回值乘以一个整数 *n* 的时候，得到 0.0 到 *n* 之间的随机值。在我们的例子中，将 *Math.random()* 乘以 6，这样返回从 0.0 到 6 之间的值。按照调整的值来调用 *floor()*，

生成 0 到  $n$  之间的证书（本例中为 0 到 6）。这会导致随机数字的一个错误分布——产生  $n$  的机会比产生该范围内的其他值的机会要小得多。

下面代码中的 *myRandom()* 函数避免了这个问题，它是简单地丢弃碰巧被 *Math.random()* 选中的值 1.0：

```
// 返回一个从minVal 到 maxVal 之间的整数，两者包括在内
function myRandom(minVal, maxVal) {
    do{
        r = Math.random(); // 挑选一个数字，直到它不是 1。
    } while (r == 1);
    return minVal + Math.floor(r * (maxVal - minVal));
}

// 调用函数
dieRoll = myRandom(1, 6); // 在 Flash 5 的修订版 30 中安全地模拟一个六面体
```

## 用法

*Math.random()*代替了不被支持的 Flash 4 中的 *random* 函数。

## 示例

*Math.random()*通常用来让播放头随机跳到某一个帧。下面的代码从前面的例子中调用 *myRandom()* 函数，然后将播放头送到随机选择的帧：

```
// 调用函数，选择一个从 10 到 20 之间的随机数字
var destinationFrame = myRandom(10, 20);

// 现在将播放头放到该帧
gotoAndStop(destinationFrame);
```

---

## Math.round()方法

计算与一个数字最临近的整数

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.round(x)`

**参数**

*x* 一个数字。

**返回**

从数学上看和 *x* 最临近的整数（如果 *x* 就是整数即为其本身）。如果 *x* 的小数部分为 0.5 (*x* 与临近的两个整数的距离相等)，那么就返回比 *x* 大的第一个整数。

## 描述

*round()*方法执行传统的四舍五入，它将浮点数转换为最近的整数。小数部分小于 0.5 的正数以及小数部分大于 0.5 的负数，都是向下舍入的。小数部分大于等于 0.5 的正数以及小数部分小于等于 0.5 的负数是向上舍入的。

## 示例

```
Math.round(1.4);    // 返回 1  
Math.round(1.5);    // 返回 2  
Math.round(1.6);    // 返回 2  
Math.round(-5.4);   // 返回 -5  
Math.round(-5.5);   // 返回 -5  
Math.round(-5.6);   // 返回 -6
```

## 参见

*int()*, *Math.ceil()*, *Math.floor()*

## Math.sin()方法

计算一个角度的正弦

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.sin(theta)`

### 参数

*theta* 一个角度，用弧度（而不是度）表示，范围为 0 到  $2\pi$ 。

### 返回

*theta* 的正弦值（结果范围从 -1.0 到 1.0）。

## 描述

*sin()*方法返回一个角度的三角正弦。在直角三角形中，角度的正弦就是角的对边边长除以斜边边长的值。

## 用法

注意，*sin()*中的角度要用弧度而不是度表示。

## 示例

```
trace(Math.sin(0));           // 显示: 0  
trace(Math.sin(Math.PI/2));   // 显示: 1.0
```

*sin()*函数可以和*cos()*结合使用，计算圆周上的点。参见 *Math.cos()*下的例子。

**参见**

*Math.asin()*, *Math.cos()*, *Math.tan()*

**Math.sqrt()方法**

计算一个数字的平方根

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.sqrt(x)`

**参数**

*x* 一个非负的整数。

**返回**

*x* 的平方根, 如果 *x* 小于 0 就返回 NaN。

**描述**

*sqrt()* 方法返回操作数的正平方根 (虽然在数学上也可以存在合法的负根)。它等于:

`Math.pow(x, 0.5)`

**示例**

```
Math.sqrt(4);      // 返回 2, 虽然 -2 也是一个合法的根  
Math.sqrt(36);    // 返回 6, 虽然 -6 也是合法根  
Math.sqrt(-20);   // 返回 NaN
```

**参见**

*Math.pow()*, *Math.SQRT2*, *Math.SQRT1\_2*

**Math.SQRT1\_2 属性**

2 的平方根的倒数, 大约为 0.70711

**有效性** Flash 5, 在导出 Flash 4 影片的时候也可以使用。

**概要** `Math.SQRT1_2`

**描述**

SQRT1\_2 属性是一个常量, 接近值  $1/Math.SQRT2$  (2 的平方根的倒数), 大约等于 0.70711。

**参见**

*Math.SQRT2*

**Math.SQRT2()属性**

2 的平方根，大约等于 1.41421

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。**概要** Math.SQRT2**描述**

SQRT2 属性是一个常量，其值接近 2 的平方根，为无理数，大约等于 1.41421。

**参见**

Math.SQRT1\_2

**Math.tan()方法**

计算一个角度的正切

**有效性** Flash 5，在导出 Flash 4 影片的时候也可以使用。**概要** Math.tan(*theta*)**参数***theta* 一个角度，用弧度（而不是度）表示，范围为  $-\pi/2$  到  $\pi/2$ 。**返回***theta* 的正切（结果范围从 -Infinity 到 Infinity）。**描述**

*tan()* 方法返回一个角度的三角正切。在直角三角形中，角度的正切是将角度对边边长除以角度邻边边长的结果。这同 *Math.sin(theta)/Math.cos(theta)* 的比率是相等的，因此，当 *cos(theta)* 接近于 0 的时候，*tan(theta)* 就接近无穷大。因此，*tan(theta)* 不计算  $-\pi/2$ 、 $\pi/2$ 、 $-3\pi/2$ 、 $3\pi/2$  等的值。

**示例**

```
trace (Math.tan(0));           // 显示: 0
trace (Math.tan(Math.PI/4));   // 显示: 1
```

**参见***Math.atan()*, *Math.cos()*, *Math.sin()***maxscroll 属性**

文本域的最后一个合法顶行

**有效性** Flash 4 及其以后的版本

**概要** `textField.maxscroll`

**返回**

一个正整数，表示文本域中最后一个合法顶行的行号。

**描述**

`maxscroll` 属性告诉我们文本域中所允许的最大 `scroll` 值。它表示文本域中能够用来作为可见区域中顶行的最后一个行号。

`maxscroll` 属性可以和 `scroll` 属性结合使用，管理滚动文本域。

**参见**

`scroll`, 第十八章

---

## Mouse 对象

隐藏或者显示鼠标指示器

**有效性** Flash 5

**概要** `Mouse.methodName`

**方法**

`hide()` 隐藏鼠标指示器。

`show()` 显示鼠标指示器。

**描述**

`Mouse` 对象只有两个方法，没有属性。用 `Mouse.hide()` 来隐藏系统鼠标指示器，通常是为了用自定义的鼠标指示器来代替它，如后所示。也可以在全屏、键盘控制影片或者触摸式问讯机上隐藏鼠标。

注意，`Mouse` 对象不告诉我们鼠标指示器的位置。这个信息存储在每一个影片剪辑对象的 `_xmouse` 和 `_ymouse` 属性中。使用 `_root._xmouse` 和 `_root._ymouse` 就可以确定鼠标指示器在主场景上的位置。

**参见**

`MovieClip._xmouse`, `MovieClip._ymouse`

---

## Mouse.hide()方法

隐藏鼠标指示器

**有效性** Flash 5

**概要** Mouse.hide()

### 描述

*hide()*方法让正常的鼠标指示器（通常是一个箭头）在鼠标移动到播放器的任何区域上时隐藏起来。正常的系统指示器在鼠标移出 Flash 播放器的活动场景区域时又会重新出现。

### 用法

注意，在 Flash 5 中，甚至在调用了 *Mouse.hide()*之后，一般的系统文本 I 型光标会在鼠标出现在文本域上的时候出现。

### 示例

用 *hide()*方法来隐藏默认的系统鼠标指示器，通常是为了用自定义的指示器来代替它。在 Flash 中，一个自定义的指示器就是一个跟随鼠标的影片剪辑。使用 *mouseMove* 事件，让一个影片剪辑跟随鼠标，只要随着每一帧更新剪辑的 *\_x* 和 *\_y* 属性即可。下面的代码示范了这个技术：

```
// 作为自定义指示器的剪辑上的代码
onClipEvent (load) {
    Mouse.hide();
}

onClipEvent (mouseMove) {
    _x = _root._xmouse;
    _y = _root._ymouse;
    updateAfterEvent();
}
```

也可以在鼠标静止的时候隐藏鼠标指示器。因为当指示器停留在 Flash 播放器的活动场景区域中时会出现系统指示器，这样，屏幕上就会有两个指示器。下面的代码表示了如何使用前边的代码来在鼠标静止 5 秒之后隐藏指示器：

```
onClipEvent (load) {
    Mouse.hide();
}

onClipEvent (enterFrame) {
    if (getTimer() - lastMove>5000) (
        _visible = false;
    } else {
        _visible = true;
    }
}
```

```
onClipEvent (mouseMove) {
    _x = _root._xmouse;
    _y = _root._ymouse;
    lastMove = getTimer();
    updateAfterEvent();
}
```

要在鼠标经过一个按钮的时候将一个自定义指示器变为自定义的“手掌”图标，可以使用按钮的*rollOver*事件来将自定义指示器剪辑设置为一个包含自定义手掌的帧，并且用*rollOut*事件来将自定义指示器设置回默认状态，比如：

```
on (rollOver) {
    _root.customPointer.gotoAndStop("hand");
}

on (rollOut) {
    _root.customPointer.gotoAndStop("default");
}
```

不管怎么说，要记住将自定义的指示器放在影片的最高层，以让它出现在所有的内容之上。另外，也可以使用*duplicateMovieClip()*或者*attachMovie()*来动态地生成自定义指示器剪辑，并为它设置一个很高的深度。

## 参见

*Mouse.show()*, *MovieClip.\_xmouse*, *MovieClip.\_ymouse*

---

## Mouse.show()方法

显示鼠标指示器

**有效性** Flash 5

**概要** `Mouse.show()`

### 描述

*show()*方法让默认的系统鼠标指示器在*Mouse.hide()*调用之后重新出现。它可以在影片需要时提供一个普通指示器，比如当用户希望填充表单的时候。当系统指示器被隐藏起来的时候，这是处理文本 I 光标的错误出现的一个方法。

## 参见

*Mouse.hide()*

---

## MovieClip “类”

针对主影片和影片剪辑的一个类似类的数据类型

**有效性** Flash 3 以及以后的版本

## 构造器

无。影片剪辑符号是在 Flash 制作工具中手动创建的。影片剪辑实例可以用 *attachMovie()* 和 *duplicateMovieClip()* 来创建。

## 属性

影片剪辑属性给出了对各种影片剪辑和主影片功能的描述和控制。属性可以使用点操作符来访问，和任何对象一样。参见第十三章可以获得使用影片剪辑属性的细节。

要注意两个和属性有关的问题：

- 改变一个 *MovieClip* 对象的物理属性会将剪辑置于程序控制之下，打断内部时间线对它的把持。这种情况可以在当前进程的任何间隙终止。
- 一些 *MovieClip* 属性为浮点数字，但是可以在内匹配到一些格式。例如，*\_alpha* 属性就可以匹配到 0 到 255 之间的整数。这在特定属性的设置和获取中会造成精度的损失。例如，我们可以将一个属性值设置为 90，然后立刻获取该值，得到的是 89.84375。如果属性值中次要的变化会造成脚本的不同，就必须在获取之后用 *Math.round()*、*Math.floor()* 或者 *Math.ceil()* 手动舍入这些值，或者将原始值存储在和属性分开的变量中。

表 R-9 列出了影片剪辑的属性。注意，所有的内置影片剪辑属性都是以下划线开始的，这可以很容易地同你添加到影片剪辑中的自定义属性（一般不以下划线开头）区分开来。在表 R-9 的访问栏里，R/W 表示属性的值可以获取和设置（也就是读/写），而 RO 表示它只能被获取不能被设置（也就是只读）。有的时候，一些只读属性可以通过制作工具或者一些相关的函数来直接进行设置（比如 *gotoAndStop()* 可以设置 *\_currentframe* 属性），只有读/写属性可以直接通过 ActionScript 来直接设置。类型一栏中描述了每一个属性值的数据类型。属性描述栏对属性的作用作了简短的描述，但是后面具体的说明会给出一些重要的细节内容。除了 *\_name* 和 *\_parent* 属性之外，表 R-9 中的属性都可以应用在影片剪辑实例和主影片（也就是 *\_root*）上。但是，一个属性的值会因为它是否从影片剪辑实例或主影片上检测而有很大的不同，而且会依据剪辑所属的地方不同而不同。例如，一个影片剪辑的 *\_x* 属性会因为它是属于主影片时间线还是作为父剪辑的子剪辑而有很大的不同。此外，当前时间线的所有属性可以在没有显式引用的情况下访问到，比如 *\_alpha* 和 *myClip.\_alpha*。

表 R-9 影片剪辑属性要略

属性名称	访问	类型	属性描述
_alpha	R/W	数字	不透明度：0为透明，100为不透明
_currentframe	RO	数字	播放头所在的帧号
_droptarget	RO	串	被拖动剪辑经过的或者已经落脚的剪辑目标路径，用斜杠符号
_framesloaded	RO	数字	已经被下载的帧的帧号
_height	R/W	数字	当前内容的高度，以像素表示
_name <sup>a</sup>	R/W	串	实例的串标识符（不是引用）
_parent <sup>a</sup>	RO	影片剪辑引用	一个对实例或者包含当前实例的影片的引用
_rotation	R/W	数字	旋转角度
_target	RO	串	绝对目标路径，使用斜杠符号
_totalframes	RO	数字	时间线中的总帧数
_url	RO	串	.swf源文件的硬盘或者网络位置
_visible	R/W	布尔	可见度：如果显示就为 true，如果隐藏就为 false
_width	R/W	数字	当前内容的宽度，以像素表示
_x	R/W	数字	水平位置，以像素表示
_xmouse	RO	数字	鼠标指示器的水平位置，以像素表示
_xscale	R/W	数字	水平缩放比例
_y	R/W	数字	垂直位置，以像素表示
_ymouse	RO	数字	鼠标指示器的垂直位置，以像素表示
_yscale	R/W	数字	垂直缩放比例

a. 应用于影片剪辑实例，不应用于主时间线。

## 方法

影片剪辑方法可以在任何影片剪辑实例中调用，在大部分情况下，也可以在任何主影片时间线上调用。许多影片剪辑方法提供了和类似全局函数相同的功能，但是只使用方便的 *MovieClip.method()* 点操作符格式。不对应到全局函数的那些方法可以应

用于当前剪辑，而不使用显式引用，比如 `attachMovie()` 和 `myClip.attachMovie()`。表 R-10 列出了影片剪辑方法。

表 R-10 影片剪辑方法要略

方法名称	方法描述
<code>attachMovie()</code>	从当前文档库中基于一个导出符号创建一个新的实例。将新的实例放在主剪辑中，或者影片的程序化生成剪辑堆栈中
<code>duplicateMovieClip()<sup>a</sup></code>	创建一个当前实例的复本，然后将复本放在合适的程序化生成剪辑堆栈中（参见第十三章）
<code>getBounds()</code>	返回一个对象，其属性给出边界框的坐标，边界框表示被剪辑所占据的可视范围
<code>getBytesLoaded()</code>	返回实例或影片的下载比特数。不能用于内部剪辑
<code>getBytesTotal()</code>	返回实例或主影片的实际比特大小
<code>getURL()</code>	将一个外部文档（通常是网页）装载到浏览器
<code>globalToLocal()</code>	将 <code>coordinates</code> 对象的属性从场景坐标转换为实例坐标。在主影片对象（比如 <code>_root</code> ）上调用的时候无效，因为初始坐标空间和目标坐标空间是同样的
<code>gotoAndPlay()</code>	将实例或影片的播放头移动到指定的帧，然后播放实例或者影片
<code>gotoAndStop()</code>	将实例或影片的播放头移动到指定的帧，然后停止
<code>hitTest()</code>	返回一个布尔值，表示一个剪辑是否和一个给定的点或者其他剪辑相交
<code>loadMovie()</code>	将一个外部.swf 文件带入播放器
<code>loadVariables()</code>	获取由变量名称和值所合成的外部数据，然后将数据转换为 ActionScript 变量
<code>localToGlobal()</code>	将 <code>coordinates</code> 对象的属性从实例坐标转换为主影片坐标
<code>nextFrame()</code>	将实例或影片的播放头前进一个帧并停在那里
<code>play()</code>	开始实例或影片的播放（也就是播放剪辑）
<code>prevFrame()</code>	将实例或影片的播放头后退一帧并停在那里
<code>removeMovieClip()<sup>a</sup></code>	删除一个复制或添加的实例
<code>startDrag()</code>	让实例或者影片物理地跟随鼠标指示器
<code>Stop()</code>	终止实例或影片的播放

表 R-10 影片剪辑方法要略（续）

方法名称	方法描述
<i>Stop Drag()</i>	停止当前进程中的拖动操作
<i>SwapDepths()<sup>a</sup></i>	交换实例堆栈中实例的位置
<i>unloadMovie()</i>	将一个实例或主影片从文档层级中或主剪辑中删除
<i>valueOf()</i>	表示到实例的绝对路径，使用点操作符

a. 应用于影片剪辑实例，不能应用于主时间线。

## 事件

影片剪辑实例支持事件处理器，它自动对预定的事件作出反应（例如，鼠标或键盘交互，或影片播放）。表 R-11 中列出了所支持的影片剪辑事件处理器。参见第十章可得到更多的细节内容。

表 R-11 影片剪辑事件处理器概要

剪辑事件处理器	剪辑事件发生的时机
<i>onClipEvent(enterFrame)</i>	Flash 播放器的一个人口帧
<i>onClipEvent(load)</i>	剪辑首次出现在场景中
<i>onClipEvent(unload)</i>	剪辑从场景中被删除
<i>onClipEvent(data)</i>	剪辑接收到了装载变量，或一个装载影片部分的数据流结尾
<i>onClipEvent(mouseDown)</i>	当剪辑在场景上的时候鼠标主键被按下
<i>onClipEvent(mouseUp)</i>	当剪辑在场景上的时候，鼠标主键被按下，然后被释放
<i>onClipEvent(mouseMove)</i>	当剪辑在场景上的时候，鼠标指示器发生移动（哪怕是一点点）
<i>onClipEvent(keyDown)</i>	当剪辑在场景上的时候，某键被按下
<i>onClipEvent(keyUp)</i>	当剪辑在场景上的时候，某按下的键被释放

## 描述

*MovieClip* 实际上不是 ActionScript 中的一个类，而是一个独特的 ActionScript 数据类型，用来表示影片和影片剪辑的有关信息，并对其进行控制。通常，我们将影片剪辑和影片当作对象来看待。可以创建和访问影片剪辑属性，也可以创建和调用影片剪辑方法。

因为 *MovieClip* 不是一个真正的类，我们不使用构造器来创建新的 *MovieClip* 实例。我们要在制作工具中创建影片剪辑符号，并且将实例手动地放到场景中。但是，我们可以用某些方法复制现有的剪辑 (*duplicateMovieClip()*)，或程序化地为场景添加新的剪辑 (*attachMovie()*)。

不是所有的 *MovieClip* 对象都是一样的，一些 *MovieClip* 方法和属性只应用于影片剪辑实例，而不是主影片（主影片就是.swf 文档的 *\_root* 时间线）。在对 *MovieClip* 属性和方法的讨论中，我们要注意功能性被限制在一个 *MovieClip* 对象类型上的情况。注意，我们使用词 *MovieClip* 来表示对象的类，用词 *movieclip*（小写）来表示 ActionScript 数据类型，用影片剪辑、剪辑或实例来表示特定的影片剪辑，用影片来表示.swf 文件的主影片。在每一个详细的属性和方法的说明中，*mc* 表示剪辑或者主影片的名称。对于很多属性和方法，*mc* 是可选的——如果遗漏，就使用当前的时间线。

贯穿 *MovieClip* 小节，当我们谈论坐标的时候，需要指出影片剪辑的位置。在测量剪辑位置的时候要指向一个代表性的点，也就是所谓的记录点，它使用剪辑库符号中的十字准线来表示的。

当一个剪辑位于主场景中的时候，对其位置的描述和场景的左上角有关，也就是点 (0, 0)。当一个剪辑位于另外一个剪辑内的时候，对其位置的描述和父剪辑的记录点有关，同样也是点 (0, 0)。两种情况下的点 (0, 0) 都是坐标空间的初始点（或者说原点），这个空间是用来划分剪辑区域的。我们可以看到 *localToGlobal()* 和 *globalToLocal()* 方法如何能够在两个坐标空间之间进行转换。

---

**注意：**Flash 的坐标系统转换了笛卡儿坐标中的 Y 轴，也就是说，y 值按照向下的方向增长。例如，某点 y 坐标为 100 表示其在 X 轴之下 100 像素点的位置。

---

我们经常用与坐标相关的属性和方法来移动剪辑，确定剪辑的位置，或者确定它是否和另外的对象或点相交。最后一项技术被称为冲突检测，是因为它经常用来确定是否要改变剪辑动画的方向，就好像跳离了另外一个对象（参见 *MovieClip.hitTest()* 方法）。

注意，ActionScript 没有自然数据类型来表示一个点（也就是 x 和 y 坐标）。参见 *MovieClip.globalToLocal()*，可以得到关于如何从普通对象创建点对象的说明。

## 参见

对于 *MovieClip* 类的全面介绍，第十三章

**MovieClip.\_alpha 属性**

剪辑或者影片的不透明度

**有效性** Flash 4 以及以后的版本**概要** *mc.\_alpha***访问** 读 / 写**描述**

浮点 \_alpha 属性指定 *mc* 不透明度（或者是相反的透明度）的百分比——0 是完全透明，100 是完全不透明。设置 *mc* 的 \_alpha 会影响所有嵌套在 *mc* 内的剪辑的视觉透明度，但是不影响它们的 \_alpha 属性。也就是说，如果有一个剪辑 square，它包含另外一个剪辑 circle，我们将 square.\_alpha 设置为 50，那么 circle 在屏幕上的视觉透明度就会变成 50%，但是它的 \_alpha 还是 100。

**用法**

注意，设置一个影片剪辑的 \_alpha 会影响 *Color.getTransform()* 所返回的对象 aa 属性。

**示例**

```
ball._alpha = 60; // 让 ball 部分透明  
ball._alpha = 0; // 让 ball 藏起来
```

下面的剪辑事件处理器让一个剪辑随着鼠标在屏幕中的下移而更为透明：

```
onClipEvent (enterFrame) {  
    _alpha = 100 - (_root._ymouse / 400) * 100;  
}
```

**参见**

*Color* 类, *MovieClip.\_visible*

**MovieClip.attachMovie()方法**

从导出库符号创建一个新的影片剪辑实例

**有效性** Flash 5**概要** *mc.attachMovie(symbolIdentifier, newName, depth)***参数**

*symbolIdentifier* 一个串，表示导出影片剪辑符号的链接标识符，在 options → Linkage 下的库中进行设置。

---

<i>newName</i>	一个串，表示所创建的剪辑新实例名称。名称必须遵循第十四章中所描述的标识符创建规则。
<i>depth</i>	一个整数，表示新的剪辑要放置到程序化剪辑堆栈中 <i>mc</i> 上的那个层级。 <i>depth</i> 为 -1 低于 0, 1 在 0 前边, 2 在 1 前边, 依次类推。参见第十三章可获得更多的细节内容。负的 <i>depth</i> 是有效的，但是没有在 ActionScript 中得到正式的支持，要确保以后的兼容性，应使用 <i>depth0</i> 或者更大的值。

**描述**

*attachMovie()*方法基于导出的由 *symbolIdentifier* 所指定的影片剪辑符号来创建新的实例 *newName*。如果 *mc* 是主影片，新的实例就被放在场景的左上角。如果 *mc* 是一个影片剪辑实例，新的实例就被放在 *mc* 的记录点。在其他情况下，新的实例被放在程序化生成剪辑堆栈中 *mc* 的上面。

**参见**

*duplicateMovieClip()*, *movieClip.duplicateMovieClip()*, 第十三章

---

<b>MovieClip._currentframe 属性</b>	剪辑或影片的播放头所在的帧号
-----------------------------------	----------------

<b>有效性</b>	Flash 4 以及以后的版本
------------	-----------------

<b>概要</b>	<i>mc._currentframe</i>
-----------	-------------------------

<b>访问</b>	只读
-----------	----

**描述**

整数 *\_currentframe* 属性表示 *mc* 的播放头当前所在的帧号。注意，第一帧为 1，而不是 0，因此，*\_currentframe* 的范围从 1 到 *mc.\_totalframes*。

**示例**

```
// 将播放头从当前位置后退两个帧
gotoAndStop(_currentframe - 2);
```

**参见**

*MovieClip.gotoAndPlay()*, *MovieClip.gotoAndStop()*

---

<b>MovieClip._droptarget 属性</b>	到拖动剪辑所落脚的剪辑或者影片的路径
---------------------------------	--------------------

<b>有效性</b>	Flash 4 以及以后的版本
------------	-----------------

**概要** mc.\_droptarget

**访问** 只读

### 描述

如果 mc 被拖动，那么 \_droptarget 就存储一个表示到 mc 所处（如果有）剪辑路径的串。如果 mc 没有在某个剪辑上，\_droptarget 就存储 undefined。如果 mc 先被拖动，然后落在某个剪辑上，\_droptarget 就存储一个表示 mc 所处剪辑路径的串，这个路径使用斜杠符号。如果被拖动剪辑的记录点和目标剪辑的某部分发生重叠，我们就认为前者位于后者之上。

### 示例

\_droptarget 属性很适于创建拖放界面。下面的例子展示了如何用 \_droptarget 创建一个简单的购物篮界面（当一个 item 剪辑落到 basket 上的时候，item 就可以留在 basket 中，否则，item 要回到原来的地方）：

```
// item的帧1上的代码
var origX = _x;
var origY = _y;

function drag() {
    this.startDrag();
}

function drop() {
    stopDrag();
    if (_droptarget != "/basket") {
        _x = origX;
        _y = origY;
    }
}

// item中按钮上的代码
on (press) {
    drag();
}

on (release) {
    drop();
}
```

注意，\_droptarget 存储的是一个串，而不是一个剪辑引用。要将 \_droptarget 串转化为一个影片剪辑引用，可以使用 eval() 示例中所展示的技术。

### 参见

*MovieClip.startDrag()*, *MovieClip.stopDrag()*

**MovieClip.duplicateMovieClip()方法**

创建影片剪辑的副本

**有效性** 该方法是在 Flash 5 中引入的（在 Flash 4 中引入了它的全局形式）**概要** `mc.duplicateMovieClip(newName, depth)`**参数****newName** 一个串，表示所创建的剪辑新实例的名称。名称必须遵循第十四章里所描述的标识符创建规则。**depth** 一个整数，表示新的剪辑要放置到程序化剪辑堆栈中 `mc` 上的那个层级。`depth` 为 -1 低于 0, 1 在 0 前边, 2 在 1 前边, 依此类推。参见第十三章可获得更多的细节内容。负的 `depth` 是有效的，但是没有在 ActionScript 中得到正式的支持。要确保以后的兼容性，使用 `depth0` 或者更大的值。**描述**

`MovieClip.duplicateMovieClip()` 方法是全局函数 `duplicateMovieClip()` 的可选途径。作为 `MovieClip` 方法来调用的时候，`duplicateMovieClip()` 没有 `target` 参数——它复制 `mc`。方法的语法比全局函数出错的机会少。

对于方法的介绍，参见全局函数 `duplicateMovieClip()`。

**参见**`MovieClip.attachMovie()`, `duplicateMovieClip()`**MovieClip.\_framesloaded 属性**

已经下载到播放器的剪辑或影片的帧数

**有效性** Flash 4 以及以后的版本**概要** `mc._framesloaded`**访问** 只读**描述**

整数 `_framesloaded` 属性表示 `mc` 已经装载到播放器中的帧数（从 0 到 `mc._totalframes`）。它通常用来创建预装载器，直到装载了足够的帧才开始播放。对于影片剪辑来说，`_framesloaded` 属性总是等于 `_totalframes`（因为剪辑在播放之前完全装载），除非实例处在用 `loadMovie()` 调用来装载外部 `.swf` 文件的过程中。`_framesloaded` 属性因此只对主影片或者装载到实例或层级中的外部 `.swf` 文件有效。

预装载器代码可以直接放在被预装载的影片主时间线上。简单的方法是：在帧1和帧2之间循环，直到影片装载结束，这时候就可以来到影片的起始帧。例如：

```
// 帧1中的代码  
if (_framesloaded > 0 && _framesloaded == _totalframes) {  
    gotoAndPlay("beginMovie");  
}  
  
// 帧2中的代码  
gotoAndPlay(1);
```

在Flash 5中，可以选择使用*enterFrame*影片剪辑事件处理器来建立一个更为合适的预装载。在要装载的影片中，我们在要开始预装载的帧上调用*stop()*函数，然后将一个带下面代码的影片剪辑放置到影片时间线上：

```
onClipEvent (enterFrame) :  
    loaded = _parent._framesloaded;  
    if (loaded > 0 && loaded == _parent._totalframes && loading != "done") {  
        _parent.gotoAndPlay("beginMovie");  
        loading = "done";  
    }  
}
```

在前边的例子中，剪辑追踪父剪辑的装载过程，当装载完毕的时候就开始播放。将影片剪辑用作一个预装载器，可以避免在预装载影片的时间线上作循环。一个影片剪辑预装载器甚至可以转换为智能剪辑，为缺乏经验的开发者提供更容易的工作流程。

注意，在预装载器例子中，我们不仅要检查 *\_framesloaded == \_totalframes* 是否为真，还要检查 *\_framesloaded > 0* 是否为真。这很有必要，因为当一个影片从剪辑中卸载的时候，该剪辑的 *\_totalframes* 为 0。因此，如果 *\_framesloaded* 为 0（因为它可能在一个非常慢的连接上），那么比较 *\_framesloaded == \_totalframes* 的结果就是 *true*，即使还没有装载任何帧也是如此。我们的检查可以防止影片在装载完合适的内容之前就往前跳。这种防范措施对于装载到层级中的.swf文件的主影片预装载器是没有必要的，因为它们的 *\_totalframes* 属性从来不会是 0。

## 示例

预装载器经常包括一个水平的装载条和一个文本域，用来表示影片装载的百分比。装载条可以用剪辑来实现，用 *\_width* 或者 *\_xscale* 属性来调整大小。但是，注意，一个剪辑的缩放是相对于它的记录点的（剪辑记录点左右两边都会按比例缩放）。因此，要只从一边重新设定剪辑，必须将所有的剪辑内容放在剪辑符号记录点的一边。下面的例子显示了如何将一个装载条和状态文本域添加到我们前面的剪辑处理器代码：

```
// 一个带状态条的预装载器
```

```

onClipEvent (enterFrame) {
    // 测量装载了多少帧
    loaded = _parent._framesloaded;
    // 如果所有的帧都装载完毕……
    if (loaded > 0 && loaded == _parent._totalframes && loading != "done") {
        // ... 播放影片，表示我们已经装载完毕
        _parent.gotoAndPlay("beginMovie");
        loading = "done";
    }
    // 确定已经装载的字节的百分比
    percentDone = Math.floor(_parent.getBytesLoaded()
        / _parent.getBytesTotal()) * 100;
    // 在文本域loadStatus中显示已经装载的字节百分比
    loadStatus = percentDone + "% complete";
    // 设置loadBar 剪辑的大小
    loadBar._xscale = percentDone;
}

```

使用测试影片模式中的模拟带宽可以模仿预装载测试的影片下载。

## 参见

*MovieClip.getBytesLoaded()*、*MovieClip.\_totalframes*, 第十章, 第十九章

## MovieClip.getBounds()方法

确定剪辑或影片的边界框

**有效性** Flash 5

**概要** *mc.getBounds(targetCoordinateSpace)*

### 参数

*targetCoordinateSpace*

一个串，表示到 *mc* 维度测量空间的影片或剪辑的路径。因为影片剪辑引用在使用于串语境的时候会转换为路径，*targetCoordinateSpace* 也可以是一个影片剪辑对象引用，正如在 *mc.getBounds(\_root)* 和 *mc.getBounds("\_root")* 中一样。如果没有设置就默认为 *mc*。

### 返回

一个对象，它的属性 (*xMin*, *xMax*, *yMin*, *yMax*) 描述 *mc* 所占据空间的边界框。对象的这四个属性分别指定 *mc* 的左边、右边、顶边和底边的像素坐标。

### 描述

*getBounds()* 方法返回一个匿名的对象，其属性定义 *mc* 所占据的矩形区域（也就是 *mc* 的边界框）。要获取存储在返回对象中的值，必须访问对象的属性，如下面的例子所示。

剪辑的边界框维度可以相对于任何其他剪辑或者影片来进行测量。使用 `targetCoordinateSpace` 参数，我们可以提出问题：“如果 `mc` 在 `targetCoordinateSpace` 的画布内，那么它所占据的是什么区域？”答案会根据 `targetCoordinateSpace` 是主影片还是剪辑实例而不同。主影片坐标空间的原点是场景的左上角，但是实例坐标空间的原点是剪辑库符号中所标注的记录点（显示为十字线）。

`getBounds()` 方法可以执行影片或剪辑同其他点之间的冲突检测（虽然 `MovieClip.hitTest()` 对于这个工作可以完成得更为出色）。它也可以用来确定要用 `MovieClip.attachMovie()` 添加到影片的剪辑要放置的矩形区域。

### 示例

```
var clipBounds = this.getBounds();
var leftX = clipBounds.xMin;
var rightX = clipBounds.xMax;
var topY = clipBounds.yMin;
var bottomY = clipBounds.yMax;
```

### 参见

`MovieClip.hitTest()`

## MovieClip.getBytesLoaded()方法

检查已经装载到播放器的字节数

**有效性** Flash 5

**概要** `mc.getBytesLoaded()`

**返回**

一个整数，表示 `mc` 中已经装载到播放器的字节数（除以 1024 可以转换为千字节）。

### 描述

`getBytesLoaded()` 方法告诉我们一个影片已经装载到 Flash 播放器的字节数。但是，`getBytesLoaded()` 只在整个帧块中测量字节数。因此，如果影片的第一个帧有 200 字节，它的第二个帧有 3000 字节，`getBytesLoaded()` 会返回 200 和 3200，但是永远不会有中间的某个数字。直到所有的给定帧都装载完毕，`getBytesLoaded()` 的返回值才会改变。`getBytesLoaded()` 方法因此可以被看成是 `_framesloaded` 属性的一个“字节版本”。

注意，内部影片剪辑总是会在它们被完全装载之后才显示，因此，`getBytesLoaded()` 在内部影片剪辑上的返回值总是和 `getBytesTotal()` 一样（除非影片剪辑现在正以

*loadMovie()*装载外部.swf文件)。因此, *getBytesLoaded()*只当用于主影片或者外部.swf文件被装载到实例或者层级的时候有效。

和 *\_framesloaded* 一样, *getBytesLoaded()*通常用来建立预装载器。它可以和 *getBytesTotal()* 一起使用, 创建一个比 *\_framesloaded* 和 *\_totalframes* 所能创建的更为精确的进度条(因为每一帧的字节数量是不相等的——如果帧的大小差异悬殊, 那么有 10 帧的影片在上载了 3 帧的时候并不是上载了 30%)。

## 示例

```
// 帧1上的代码
if (_framesloaded > 0 && framesloaded == totalframes) {
    gotoAndPlay('beginMovie');
} else {
    // 在文本域中显示装载进度。除以 1024 就可以转换为千字节 KB。
    loadProgressOutput = this.getBytesLoaded() / 1024;
    loadTotalOutput = this.getBytesTotal() / 1024;
}

// 第2帧上的代码
gotoAndPlay(1);
```

## 参见

*MovieClip.\_framesloaded*, *MovieClip.getBytesTotal()*

---

## MovieClip.getBytesTotal()方法

检查剪辑或影片的字节数量

**有效性** Flash 5

**概要** *mc.getBytesTotal()*

### 返回

一个整数, 表示 *mc* 的字节大小, 除以 1024 可以转换为千字节。

### 描述

*getBytesTotal()*方法告诉我们一个剪辑实例或者主影片的字节大小。在主影片中调用的时候, *getBytesTotal()*会给出整个.swf文件的大小。它通常和 *getBytesLoaded()*结合使用, 产生主影片和装载到实例或者层级的.swf文件的预装载器。

## 参见

*MovieClip.getBytesLoaded()*, *MovieClip.\_totalframes*

---

**MovieClip.getURL()方法**

将一个文档装载到浏览器窗口

**有效性**

该方法是在 Flash 5 中引入的（在 Flash 2 中引入了它的全局形式，在 Flash 4 中因包括了 *method* 参数而得到增强）

**概要**

*mc.getURL(URL, window, method)*

**参数***URL*

一个串，指定要装载的文档或者要运行的外部脚本的位置。

*window*

一个可选的串，指定浏览器窗口或者装载到文档中的帧的名称。也可以是自定义名称或下面的四种设置之一：“\_blank”, “\_parent”, “\_self”或者“\_top”。

*method*

一个可选的串常量，指定用来从 *mc* 发送变量到外部脚本的方法。必须是常量“GET”或者“POST”，不允许使用其他表达式。

**描述**

*MovieClip.getURL()* 方法是 *getURL()* 全局函数的替换方式。它的方法格式只当发送变量的时候才有效，在这种情况下，*getURL()* 从 *mc* 发送变量，而不一定要从当前时间线发送变量。

**参见**

关于一般用法介绍，参见 *getURL()* 全局函数

---

**MovieClip.globalToLocal()方法**

将主场景上的一个点转换为剪辑坐标

**有效性**

Flash 5

**概要**

*mc.globalToLocal(point)*

**参数***point*

一个对对象的引用，该对象包含两个属性：*x* 和 *y*，描述播放器主场景（也就是 *\_root*）上的一点。*x* 和 *y* 都可以是浮点数。

**描述**

*globalToLocal()* 方法将 *point* 的 *x* 和 *y* 属性从主场景坐标转换为 *mc* 的空间坐标。注意，*globalToLocal()* 不返回一个新的对象，它只是修改现有 *point* 的 *x* 和 *y* 的值。

要使用 *globalToLocal()*，必须首先创建一个有 *x* 和 *y* 属性的对象。例如：

```
var myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

对象的 x 和 y 属性在主场景的水平和垂直轴上，和左上角相对。例如，x 等于 10 就是离主场景左边缘 10 像素，y 等于 20 就是在场景上边缘之下 20 像素。创建好了对象，设置了 x, y 属性，我们就可以将对象传递给 *globalToLocal()* 方法，比如：

```
myClip.globalToLocal(myPoint);
```

执行 *globalToLocal()* 的时候，*myPoint* 的 x 和 y 属性被转换为 *myClip* 空间中的一个点，从 *myClip* 的记录点开始计量。通过检查 *myPoint* 对象属性的新的值，我们回答了“主场景中的点 (x, y) 什么时候出现在 *myClip* 中？”例如：

```
xInClip = myPoint.x;
yInClip = myPoint.y;
```

## 示例

下面的例子计算从主场景左上角到当前剪辑记录点的偏移量：

```
pt = new Object();           // 创建一个普通的对象来容纳我们的点
pt.x = 0;                   // 主场景的左边缘
pt.y = 0;                   // 主场景的右边缘
this.globalToLocal(pt);    // 将 pt 转换为局部坐标

trace("From the current clip, the top-left corner of the main Stage is at ");
trace("x: " + pt.x + "y: " + pt.y);
```

## 参见

*MovieClip.localToGlobal()*

## MovieClip.gotoAndPlay()方法

跳到指定的帧，然后播放

### 有效性

该方法是在 Flash 5 中引入的（在 Flash 2 以及以后的版本中支持它的全局形式）

### 概要

```
mc.gotoAndPlay(frameNumber)
mc.gotoAndPlay(frameLabel)
```

### 参数

*frameNumber* 一个正整数，表示 *mc* 的播放头在播放之前要转到的帧号。如果 *frameNumber* 小于 1，对 *Movieclip.gotoAndPlay()* 的调用将被忽略；如果 *frameNumber* 大于 *mc* 时间线上的总帧数，播放头就被送到

最后一帧。注意，这和全局函数 *gotoAndPlay()* 是不同的，后者将小于 1 的值作为 1 来对待。

*frameLabel* 一个串，表示 *mc* 的播放头在播放之前要转到的帧标签。如果 *frameLabel* 没有找到，播放头就被送到 *mc* 时间线的第一帧。

### 描述

*MovieClip.gotoAndPlay()* 方法是全局函数 *gotoAndPlay()* 的替换方式。可以使用方法格式来控制由 *mc* 指定的远程影片剪辑或影片。

对于一般用法的介绍，参见全局函数 *gotoAndPlay()*。

### 示例

```
// 将 part1 剪辑发送到 intro 标签，然后播放 part1
part1.gotoAndPlay("intro")
```

---

## MovieClip.gotoAndStop()方法

跳到指定的帧，然后停止播放

**有效性** 该方法是在 Flash 5 中引入的（在 Flash 2 以及以后的版本中支持它的全局形式）

### 概要

```
mc.gotoAndStop(frameNumber)
mc.gotoAndStop(frameLabel)
```

### 参数

*frameNumber* 一个正整数，表示 *mc* 的播放头要转到的帧号。如果 *frameNumber* 小于 1，对 *MovieClip.gotoAndStop()* 的调用将被忽略；如果 *frameNumber* 大于 *mc* 时间线上的总帧数，播放头就被送到最后一帧。注意，这和全局函数 *gotoAndStop()* 是不同的，后者将小于 1 的值作为 1 来对待。

*frameLabel* 一个串，表示 *mc* 的播放头要转到的帧标签。如果 *frameLabel* 没有找到，播放头就被送到 *mc* 时间线的第一帧。

### 描述

*MovieClip.gotoAndStop()* 方法是全局函数 *gotoAndStop()* 的替换方式。可以使用方法格式来控制由 *mc* 指定的远程影片剪辑或主影片。

对于一般用法的介绍，参见全局函数 *gotoAndStop()*。

## 示例

```
// 将mainMenu 剪辑送到帧 6，然后停止播放  
mainMenu.gotoAndStop(6);
```

## MovieClip.\_height 属性

剪辑或影片的高度，以像素表示

**有效性** Flash 4，在 Flash 5 中得到了增强

**概要** `mc._height`

**访问** 在 Flash 4 中为只读，在 Flash 5 中为读/写

### 描述

浮点 `_height` 属性是一个非负的数字，用来指定 `mc` 的高度（用像素表示）。如果 `mc` 没有内容，`_height` 就为 0。`_height` 属性将剪辑的内容所占据的最高像素和最低像素之间的距离当作剪辑的高度进行度量，即使在这些像素之间有间隙也是一样。占据像素是指包含形状、图形、按钮、影片剪辑或其他元素内容的像素。用制作工具或 `_yscale` 对影片剪辑进行的修改可以通过 `_height` 反映出来。

我们可以设置 `_height` 的值，在垂直方向上重新设置影片剪辑的大小。将 `_height` 设置为一个负值则会被忽略。将剪辑的 `_height` 设置为 0 将不会隐藏剪辑，而会将其变成一像素高的水平线。

主影片的 `_height`（也就是 `_root._height` 或者 `_leveln._height`）不是在制作工具的 **Modify → Movie → Dimensions** 下所指定的场景高度，而是主影片内容的高度。并没有明确的场景高度属性，如果需要，必须将场景高度手动地作为一个变量。例如，如果一个影片的场景高度为 400，可以添加下面的变量：

```
_root.stageHeight = 400;
```

要让这个值在任何剪辑的时间线上有效，使用：

```
Object.prototype.stageHeight = 400;
```

### 用法

注意，当我们设置影片剪辑的高度时，线条也按比例缩放，丢失它们在描边 (Stroke) 面板中设置的初始值。但是，在描边面板中设置为 Hairline (绒线) 的线条的点尺寸在影片剪辑重新设置大小的时候不会变化。也就是说，使用绒线可以防止描边在剪辑被调用时发生变形。

## 示例

```
ball._height = 20; // 将ball的高度设置为20像素  
ball._height /= 2; // 将ball的高度减少一半
```

## 参见

*MovieClip.\_width*, *MovieClip.\_yscale*

## MovieClip.hitTest()方法 检查一个点是否在剪辑上，或者两个剪辑是否交迭

**有效性** Flash 5

**概要** `mc.hitTest(x, y, shapeFlag)`

`mc.hitTest(target)`

### 参数

*x* 要测试点的水平坐标。

*y* 要测试点的垂直坐标。

*shapeFlag* 一个可选的布尔值，表示冲突测试是要检测*mc*的边界框 (`false`) 还是 *mc* 的实际像素 (`true`)。如果没有提供就默认为 `false`。注意，*shapeFlag*只能与 *x* 和 *y* 而不能与 *target* 参数一起使用。只当 *mc* 有不规则的轮廓线或环形的洞时该参数才有意义；如果 *mc* 是一个固定的矩形对象该参数就不起作用。

*target* 一个串，指定到要测试和 *mc* 冲突的影片剪辑的路径。因为影片剪辑引用在用于串语境的时候会转换为串，因此 *target* 也可以是影片剪辑对象引用，比如 `mc.hitTest(ball)` 和 `mc.hitTest("ball")`。

### 返回

一个布尔值，表示冲突测试的结果。在下面的情况下结果为 `true`:

- 主场景上的点 (*x*, *y*) 和 *mc* 的任何一个占据像素冲突。占据像素是包含视觉元素的像素，比如形状或文本。
- *shapeFlag* 属性为 `false`，同时主场景上点 (*x*, *y*) 和 *mc* 边界框的任何像素冲突。*mc* 的边界框是可以容纳 *mc* 的所有被占据像素的最小矩形。
- *target* 边界框中的任何像素同 *mc* 边界框中的任何像素冲突。

在下面的情况下结果为 `false`:

- *shapeFlag* 属性为 `true`, 主场景上的点  $(x, y)$  不和 *mc* 的任何占据像素冲突。占据像素就是包含视觉元素的像素, 比如形状或文本。
- 主场景上的点  $(x, y)$  不和 *mc* 的边界框上的任何像素冲突。*mc* 的边界框是可以容纳 *mc* 的所有被占据像素的最小矩形。
- *target* 边界框中的任何像素都不和 *mc* 边界框中的任何像素冲突。

### 描述

*hitTest()* 方法用来确定影片剪辑或指定的点是否同 *mc* 发生冲突。

当查看一个点是否和 *mc* 冲突时, 我们给出 *hitTest()* 以及点的 *x* 和 *y* 坐标 (与主场景相对) 以便检查。也可以使用可选的 *shapeFlag* 参数, 它表示冲突测试是要使用 *mc* 的实际像素, 还是只使用 *mc* 的边界框 (能包含 *mc* 所有占据像素的最小矩形)。检查 *mc* 的实际像素可以确定点  $(x, y)$  是否是 *mc* 轮廓内的占据像素, 而不是只检查它是否为 *mc* 边界框内的一个点。

当检查影片剪辑是否同 *mc* 相冲突时, 我们使用带 *target* 参数的 *hitTest()*, 参数指定的是要检测的剪辑。*target* 和 *mc* 之间的冲突检测通常使用剪辑的边界框, *hitTest()* 不支持全像素的剪辑对剪辑的探测。手动的全像素冲突检测程序很难创建, 处理器运行也需要加强。在很多情况下 (例如, 一个简单的太空船游戏) 探测边界圆比探测实际像素更为实际。

### 用法

注意, 冲突总是按照 *mc* 在播放器主场景上的相对位置类进行检测的。因此, 当 *hitTest()* 用于一个单独的点时, 参数 *x* 和 *y* 可以用场景坐标来描述一个点。参见 *MovieClip.localToGlobal()* 以获得更多关于将剪辑坐标转换为场景坐标的信息。在剪辑对剪辑的检测中, 不同时间线上的剪辑坐标被自动转换为全局 (也就是场景) 坐标。

### 示例

这个例子显示了如何手动地探测两个圆之间的冲突, 而不使用 *hitTest()*:

```
// 检查两个圆剪辑在两个轴上的距离
var deltaX = clip1._x - clip2._x; // 水平距离
var deltaY = clip1._y - clip2._y; // 垂直距离

// 将每个圆的半径存储在方便的属性中
var radius1 = clip1._width / 2;
var radius2 = clip2._width / 2;

// 如果圆心的距离小于等于两个半径的总长, 就会发生冲突。
if ((deltaX * deltaX) + (deltaY * deltaY))
```

```
<= (radius1 + radius2) * (radius1 + radius2)) {  
    trace("intersecting");  
} else {  
    trace("not intersecting");  
}
```

下面，我们用 *hitTest()* 来检查 paddle 是否和 ball 冲突：

```
if(paddle.hitTest("ball")) {  
    trace("The paddle hit the ball.");  
}
```

下面检查鼠标指示器是否在 tractor 轮廓线的占据像素上。注意，指示器的坐标是按照和 *\_root*（主场景）相对的关系给出的。当 *shapeFlag* 为 *true* 时，影片剪辑中的间隙可以被检测到。例如，如果 tractor 用了一个空块来表示窗口，那么 *hitTest()* 会在指示器经过窗口的时候返回 *false*：

```
if (tractor.hitTest(_root._xmouse, _root._ymouse, true)) {  
    trace("You're pointing to the tractor.");  
}
```

## 参见

*MovieClip.getBounds()*, *MovieClip.localToGlobal()*

## MovieClip.loadMovie()方法

将一个外部.swf文件装载到播放器中

**有效性** 该方法是在 Flash 5 中引入的（在 Flash 4 中支持它的全局形式）

### 概要

*mc.loadMovie(URL)*

*mc.loadMovie(URL, method)*

### 参数

#### *URL*

一个串，指定要装载的外部.swf文件的位置。

#### *method*

一个可选的串直接量，表示要用来发送变量到外部脚本的方法。必须是直接量 "GET" 或者 "POST"，不能使用其他表达式。

### 描述

*MovieClip.loadMovie()* 方法是全局函数 *loadMovie()* 的替换方式。作为 *MovieClip()* 方法来调用的时候，*loadMovie()* 没有 *target* 参数，它将 *URL* 指示的.swf文件装载到 *mc*。方法的语法与全局函数形式相比不容易出错。

至于用法说明，参见全局函数 *loadMovie()*。

**参见**

*loadMovie()*, 第十三章

**MovieClip.loadVariables()方法**

获取外部文件中的一系列变量

**有效性**

该方法是在 Flash 5 中引入的（在 Flash 4 以及以后版本中支持它的全局形式）

**概要**

*mc.loadVariables(URL)*

*mc.loadVariables(URL, method)*

**参数***URL*

一个串，指定到一个或者两个变量来源的路径：一个能够产生输出变量的脚本或者一个包含变量的文本文件。

*method*

一个可选的串直接量，表示要用来从 *mc* 发送变量到外部脚本的方法。如果指定了，那么变量可以被发送和装载。如果遗漏，变量就只能被装载。必须是直接量 "GET" 或者 "POST"，不能使用其他表达式。

**描述**

*MovieClip.loadVariables()*方法是全局函数 *loadVariables()* 函数的替换方式。作为 *MovieClip* 方法来调用的时候，*loadVariables()* 没有 *target* 参数，它将 *URL* 指定的变量装载到 *mc*。方法的语法与全局函数形式相比不容易出错。

至于用法说明，参见全局函数 *loadVariables()*。

**参见**

*loadVariables()*, 第十三章

**MovieClip.localToGlobal()方法**

将剪辑中的一个点转换为主场景坐标

**有效性**

Flash 5

**概要**

*mc.localToGlobal(point)*

**参数***point*

一个指向对象的引用，该对象包含两个属性：*x* 和 *y*，表示 *mc* 坐标空间中的一个点。*x* 和 *y* 都可以是浮点数字。

## 描述

*localToGlobal()*方法将*point*的x和y属性从*mc*坐标空间转换到播放器主场景坐标。注意，*localToGlobal()*不返回一个新的对象，它只修改*point*的现有x和y值。

要使用*localToGlobal()*，必须首先创建一个所谓的点或坐标对象，以x和y为属性。要这么做，只要简单地从*Object*类创建一个普通对象，然后添加x和y属性：

```
myPoint = new Object();
myPoint.x = 10;
myPoint.y = 20;
```

对象中的x和y属性定位在*mc*的水平和垂直轴上，和*mc*的记录点相对（在*mc*库符号中表现为一个十字线）。例如，x为10则在*mc*记录点右边的10像素位置上，y为20就是在*mc*记录点下面的20像素位置处。创建好了对象，并添加了x和y属性之后，就可以将对象传递给*localToGlobal()*方法，比如：

```
myClip.localToGlobal(myPoint);
```

执行*localToGlobal()*的时候，*myPoint*的x和y属性的值就被转换，用来表示主场景中的对应点，从场景的左上角开始计量。通过检查*myPoint*对象的属性的新值，我们可以回答“影片剪辑的点(x, y)将出现在主场景中的什么地方”这样的问题。例如：

```
mainX = myPoint.x;
mainY = myPoint.y;
```

## 示例

下面的例子用来确定剪辑记录点和主场景相对的位置：

```
pt = new Object();
pt.x = 0;                                // 剪辑的水平记录点
pt.y = 0;                                // 剪辑的垂直记录点
this.localToGlobal(pt);                    // 将pt转换为主场景坐标

trace("On the main Stage, the registration point of the current clip is at: ");
trace("x: " + pt.x + "y: " + pt.y);
```

*localToGlobal()*方法可以用来将一个点转换为场景坐标，以便使用*hitTest()*方法，这个方法需要检测点在主场景坐标空间中。它也可以用来比较不同影片剪辑中使用共同坐标空间的两个点。

## 参见

*MovieClip.globalToLocal()*, *MovieClip.hitTest()*

**MovieClip.\_name 属性**

剪辑实例的标识符，是一个串

**有效性** Flash 4 及其以后的版本

**概要** `mc._name`

**访问** 读 / 写

**描述**

`_name` 属性指定影片剪辑实例的名称，以串表示。因为它最初反映的是制作期间设置在实例面板中的实例名称，因此 `_name` 属性不能用于主影片（主影片最容易通过全局属性 `_root` 来指示）。

**示例**

可以用 `_name` 来确定是否执行给定剪辑的操作，就像我们在例 10-2 中生成一系列星星影片剪辑时所做的那样。

`_name` 属性也可以用来重新设置剪辑的标识符。例如：

```
// 将 ball 转变成 circle  
ball._name = "circle";  
  
// 现在将以前的 ball 作为 circle 来控制  
circle._x = 500;
```

**参见**

`MovieClip._target`, `targetPath()`, 第十三章

---

**MovieClip.nextFrame()方法**

将剪辑或者影片的播放头推进一帧并停止

**有效性** 该方法是在 Flash 5 中引入的（在 Flash 2 及其之后版本中支持它的全局形式）

**概要** `mc.nextFrame()`

**描述**

`MovieClip.nextFrame()` 方法是全局函数 `nextFrame()` 的替换方式。可以使用方法形式来控制由 `mc` 所指定的远程影片剪辑或者主影片。

关于一般用法介绍，参见全局函数 `nextFrame()`。

## 示例

```
// 将 slideshow 推进一帧，并且停止播放头  
slideshow.nextFrame();
```

## 参见

*MovieClip.prevFrame()*, *nextFrame()*

## MovieClip.\_parent 属性

一个指向主剪辑或包含该剪辑的影片的引用

**有效性** Flash 4 及其以后的版本

**概要** *mc.\_parent*

**访问** 只读

### 描述

*\_parent* 属性存储了一个剪辑对象的引用，*mc* 就存在于该剪辑的时间线上。主影片支持 *\_parent* 属性，因为它们在最顶层（作为 *\_root* 来引用比较方便）。对 *\_root.\_parent* 的引用返回 *undefined*。*\_parent* 属性可以很好地操作和当前剪辑相关的剪辑。

### 故障

虽然在 Flash 5 中可以重新将 *\_parent* 属性设置为其他剪辑，但是这么做并没有什么意义——只有引用而不是剪辑的物理结构被改变。这种无意的行为可能在将来会有所改变。

## 示例

如果 *mc* 存在于影片的主时间线上，可以用下面的代码来从 *mc* 播放主时间线：

```
_parent.play();
```

也可以设置父时间线的属性，比如：

```
_parent._alpha = 50;
```

*\_parent* 属性也可以接连使用，也就是说，可以访问 *\_parent* 的 *\_parent*。比如：

```
_parent._parent.play(); // 播放当前剪辑上两层的剪辑
```

## 参见

*\_root*, 第十三章

**MovieClip.play()方法**

开始连续显示剪辑或者影片中的帧

**有效性** 在 Flash 5 中引入了方法形式（在 Flash 2 及其后面的版本中支持全局形式）

**摘要** `mc.play()`

**描述**

*MovieClip.play()*方法是全局函数 *play()* 的替换方式。使用方法形式可控制由 *mc* 指定的远程影片剪辑或者主影片。

关于一般用法的介绍，参见全局函数 *play()*。

**示例**

```
// 开始播放剪辑 intro  
intro.play();
```

**参见**

*MovieClip.stop()*, *play()*

---

**MovieClip.prevFrame()方法**

将剪辑或者影片的播放头  
后退一帧，然后停止

**有效性** 在 Flash 5 中引入了方法形式（在 Flash 2 及其以后的版本中支持全局形式）

**摘要** `mc.prevFrame()`

**描述**

*MovieClip.prevFrame()*方法是全局函数 *prevFrame()* 的替换方式。使用方法形式可控制由 *mc* 指定的远程影片剪辑或者主影片。

关于一般用法的介绍，参见全局函数 *prevFrame()*。

**示例**

```
// 将 slideshow 后退一帧，然后停止  
slideshow.prevFrame();
```

**参见**

*MovieClip.nextFrame()*, *prevFrame()*

---

**MovieClip.removeMovieClip()方法** 从 Flash 播放器删除一个影片剪辑

**有效性** 在 Flash 5 中引入了方法形式（在 Flash 4 及其以后的版本中支持全局形式）

**摘要** `mc.removeMovieClip()`

**描述**

*MovieClip.removeMovieClip()*方法是全局函数 *removeMovieClip()*的替换方式。作为 *MovieClip* 方法来调用的时候，*removeMovieClip()*没有 *target* 参数，它将删除 *mc*。方法的语法与对应的全局函数相比不容易出错。

关于一般用法的介绍，参见全局函数 *removeMovieClip()*。

**参见**

*duplicateMovieClip()*, *MovieClip.attachMovie()*, *MovieClip.duplicateMovieClip()*, 第十三章

---

**MovieClip.\_rotation 属性**

剪辑或者影片的旋转角度

**有效性** Flash 4 及其以后的版本

**摘要** `mc._rotation`

**访问** 读 / 写

**描述**

浮点 *\_rotation* 属性指定 *mc* 从原来的方向所旋转的角度（如果 *mc* 是剪辑，最初的方向就是它在库中的符号）。制作工具和程序化的调整都可以在 *\_rotation* 上得到反映。0 到 180.0 之间的数字将剪辑按照顺时针方向旋转。0 到 -180.0 之间的数字将剪辑向逆时针方向旋转。将剪辑旋转 *n* 度或者 *n*-360 度（*n* 为正数）的效果是一样的。例如，将一个剪辑旋转 +299.4 度或者旋转 -60.6 度效果并没有差别。同样，当 *n* 为负数的时候，*n* 度和 *n*+360 度之间也没有区别。例如，将一个剪辑旋转 -90 度和旋转 +270 度是一样的。

当 *\_rotation* 被设置为 -180 到 180 范围之外的任何值时，该值会按照下面的计算来转化到恰当的范围内：

```
x = newRotation % 360;  
if (x > 180) {  
    x -= 360;
```

```
    } else if (x < -180) {
        x += 360;
    }
    _rotation = x;
```

## 故障

在 Flash 4 播放器中，设置剪辑的 `_rotation` 会按照分数数量减小一个剪辑的比例。多次设置 `_rotation` 后，剪辑实际上会明显变小。要解决这个问题，可以在设置 `_rotation` 的时候将剪辑的 `_xscale` 和 `_yscale` 设置为 100。

## 示例

将下面的代码放在一个剪辑上，会让剪辑在每次帧渲染的时候按照顺时针的方向旋转 5 度：

```
onClipEvent (enterFrame) {
    _rotation += 5;
}
```

---

## MovieClip.startDrag()方法

让一个影片或影片剪辑跟随鼠标移动

### 有效性

在 Flash 5 中引入了方法形式（在 Flash 4 及其以后的版本中支持全局形式）

### 概要

```
mc.startDrag()  
mc.startDrag(lockCenter)  
mc.startDrag(lockCenter, left, top, right, bottom)
```

### 参数

*lockCenter* 一个布尔值，表示 *mc* 的记录点是应该处在鼠标指示器的中心 (*true*)，还是相对它的最初位置来拖动 (*false*)。

*left* 一个数字，用来指定 *mc* 记录点左边的 x 坐标。

*top* 一个数字，用来指定 *mc* 记录点上面的 y 坐标。

*right* 一个数字，用来指定 *mc* 记录点右边的 x 坐标。

*bottom* 一个数字，用来指定 *mc* 记录点下面的 y 坐标。

### 描述

*MovieClip.startDrag()* 方法是全局函数 *startDrag()* 的替换方式。作为 *MovieClip* 方法调用的时候 *startDrag()* 没有 *target* 参数，它拖动 *mc*。方法的语法与全局函数形式相比不容易出错。

关于一般用法的介绍，参见全局函数 *startDrag()*。

### 故障

注意，表示矩形坐标的正确顺序为左、上、右、下。而在 Flash 5 ActionScript 字典中，*MovieClip.startDrag()*中列出的顺序是错误的。

### 示例

```
// 用来拖动和放置当前剪辑和影片的按钮代码
on (press) {
    this.startDrag(true);
}

on (release) {
    stopDrag();
}
```

### 参见

*MovieClip.stopDrag()*, *startDrag()*, *stopDrag()*, 第十三章

---

## MovieClip.stop()方法

终止剪辑或者影片

**有效性** 在 Flash 5 中引入了方法形式（在 Flash 2 及其以后的版本中支持全局形式）

**概要** *mc.stop()*

### 描述

*MovieClip.stop()*方法是全局函数 *stop()*的替换方式。使用方法形式可控制由 *mc* 指定的远程影片剪辑或主影片。

关于一般用法的介绍，参见全局函数 *stop()*。

### 示例

```
// 停止 spinner 的播放
spinner.stop();
```

### 参见

*MovieClip.play()*, *stop()*

---

## MovieClip.stopDrag()方法

停止拖动操作

**有效性** 在 Flash 5 中引入了方法形式（在 Flash 4 及其以后的版本中支持全局形式）

**概要** *mc.stopDrag()*

### 描述

*MovieClip.stopDrag()*方法是全局函数*stopDrag()*的替换方式。但是没有必要使用方法形式，*stopDrag()*会取消进程中的任何拖动操作，不管是通过影片剪辑还是作为一个全局函数来调用。

### 参见

*MovieClip.startDrag()*, *stopDrag()*

---

## MovieClip.swapDepths()方法

交换实例堆栈中实例的图形层次

**有效性** Flash 5

**概要** *mc.swapDepths(target)*  
*mc.swapDepths(depth)*

### 参数

*target* 一个串，表示到要和*mc*进行交换的影片剪辑的路径。因为影片剪辑引用使用在串语境中会被转换为路径，因此*target*也可以是一个影片剪辑对象引用，比如*mc.swapDepths(window2)*和*mc.swapDepths("window2")*。

*depth* 一个整数，指定*mc*父剪辑堆栈中的一个层级。*depth*为-1则在0后面，为1则在0前面，为2又在1前面，依此类推。参见第十三章可以了解更多的细节。负的*depth*虽有效，但是在ActionScript中没有得到正式的支持，要确保以后的兼容性，可以使用大于或等于0的*depth*。

### 描述

.swf文档中所有的影片剪辑实例都存在于一个堆栈中，该堆栈控制播放器中实例的视觉分层。堆栈的构造就如同一副扑克，在堆栈中位置较高的剪辑会出现在位置较低的剪辑之前。堆栈中实例的位置在创建剪辑（在Flash制作工具中创建，或者通过*attachMovie()*或*duplicateMovieClip()*创建）的时候就会被初始化。使用*swapDepths()*可以改变堆栈中实例的位置。

*swapDepths()*方法有两种形式。使用*target*参数的时候，*swapDepths()*对换*mc*和*target*的堆栈位置，规定*mc*和*target*要共享相同的父剪辑（在相同的时间线内）。

使用 depth 参数的时候，*swapDepths()* 将 *mc* 放在 *mc* 的父堆栈的一个新位置上。注意，*swapDepths()* 不能移动父堆栈所容纳的范围之外的剪辑。关于影片和影片剪辑在播放器里如何堆栈的信息，请参见第十三章。

### 故障

注意，将一个复制或者添加的实例同一个手动创建的实例通过 depth 参数来进行交换，在 Flash 5 中会引起屏幕的刷新问题。请谨慎对待 depth 参数，并不断认真检查深度交换代码。

### 参见

第十三章

---

**MovieClip.\_target 属性** 一个剪辑或影片的目标路径，使用斜杠语法

**有效性** Flash 4 及其以后的版本

**概要** *mc.\_target*

**访问** 只读

### 描述

*\_target* 属性表示 Flash 4 风格的斜杠字符串所描述的到 *mc* 的路径。例如，如果剪辑 *ball* 在主影片时间线上，*ball* 的 *\_target* 属性为 “/ball”。而 *ball* 内的一个剪辑 *stripe* 的 *\_target* 就是 “/ball/stripe”。

要获取一个表示到剪辑路径的使用点符号的串，可以用 *targetPath()* 函数。

### 参见

*targetPath()*

---

**MovieClip.\_totalframes 属性** 剪辑或影片时间线中的帧数

**有效性** Flash 4 及其以后的版本

**概要** *mc.\_totalframes*

**访问** 只读

### 描述

整数 *\_totalframes* 属性表示 *mc* 时间线上的总帧数，*mc* 是一个影片剪辑或主影片。一个新的剪辑或主影片的 *\_totalframes* 总是 1。但是，如果一个剪辑的内容通过

*unloadMovie()*被卸载，那么 *\_totalframes* 就变成了 0。当 *loadMovie()* 在一个很慢的连接上操作的时候，如果当前剪辑在装载新的剪辑之前被卸载，那么它也可能立刻变成 0。*\_totalframes* 属性常和 *\_frameLoaded* 一起使用，创建 *MovieClip.\_framesLoaded* 中所示的预装载器。

### 参见

*MovieClip.\_framesLoaded*

---

## **MovieClip.unloadMovie()方法**      从播放器中删除一个影片或影片剪辑

**有效性**      Flash 5 中引入了方法形式（在 Flash 3 及其以后的版本中支持全局形式）

**概要**      *mc.unloadMovie()*

### 描述

*MovieClip.unloadMovie()*方法是全局 *unloadMovie()* 函数的替换形式。作为 *MovieClip* 方法调用的时候，*unloadMovie()* 没有 *target* 参数，它将卸载 *mc*。方法语法与全局函数形式相比不容易出错。

关于用法介绍，可以参见全局函数 *unloadMovie()*。

### 示例

```
// 从层级 1 删除一个装载进来的文档  
_level1.unloadMovie();
```

### 参见

*MovieClip.loadMovie()*, *unloadMovie()*, 第十三章

---

## **MovieClip.\_url 属性**      剪辑或影片被装载的网络地址

**有效性**      Flash 4 及其以后的版本

**概要**      *mc.\_url*

**访问**      只读

### 描述

*\_url* 属性表示的 URL（统一资源地址）描述了装载 *mc* 的 Internet 或本地硬盘的位置，形式为一个串。*\_url* 属性总是绝对的 URL，从来不使用相对的。对于主影片来说，*\_url* 是当前 *.swf* 文件的位置。*.swf* 文件里的所有影片剪辑的 *\_url* 都和该文件的

主影片的 `_url` 相同，除非外部 `.swf` 文件通过 `MovieClip.loadMovie()` 装载到单独的剪辑中。包含外部装载 `.swf` 文件的剪辑的 `_url` 就是外部装载文件的位置。

`_url` 属性有的时候被用来创建简单的安全系统，防止影片在不希望播放的地方播放。

### 示例

从一个 web 站点装载的影片的 `_url` 值如下所示：

```
'http://www.moock.org/gwen/meetgwen.swf'
```

从本地 PC 硬盘驱动器装载的影片的 `_url` 值如下所示：

```
'file:///C:/data/flashfiles/movie.swf'
```

下面，检查影片是否在恰当的位置上（如果不是，就显示一个错误信息帧）：

```
if (_url != "http://www.moock.org/gwen/meetgwen.swf") {  
    trace ("This movie is not running from its intended location.");  
    gotoAndStop("accessDenied");  
}
```

### 参见

`MovieClip.loadMovie()`

---

## MovieClip.valueOf()方法

剪辑或影片的路径串

**有效性** Flash 5

**概要** `mc.valueOf()`

### 返回

一个包含到 `mc` 的完整路径的串，使用点语法。例如：

```
"_level1.ball"  
'_level0'  
'_level0.shoppingcart.product1'
```

### 描述

`valueOf()` 方法返回一个串，该串表示到 `mc` 的绝对路径，使用点语法符号。`valueOf()` 方法会在 `MovieClip` 对象用在需要串的地方的时候自动被调用。例如，`trace(mc)` 会和 `trace(mc.valueOf())` 产生同样的结果。因此，几乎不需要显式地调用 `valueOf()`。

### 参见

`Object.valueOf()`, `targetPath()`

---

**MovieClip.\_visible()方法** 表示一个剪辑或影片是被显示还是被隐藏

**有效性** Flash 4 及其以后的版本

**概要** mc.\_visible

**访问** 读 / 写

### 描述

布尔值 `_visible` 属性表示 `mc` 当前是被显示 (`true`) 还是被隐藏 (`false`)。`_visible` 是隐藏影片剪辑或影片的一种快速方法。注意，隐藏的剪辑仍然可以通过 ActionScript 来控制，仍然可以播放、停止、接收事件，以及进行与正常情况下一样的其他操作。隐藏的剪辑只是不显示在屏幕上。

`_visible` 属性的初始值是 `true`，甚至对于那些完全透明或者完全在场景外面的剪辑也是如此。`_visible` 属性只在被脚本有意修改的时候才发生变化。应将它看成一个程序化显示和隐藏剪辑的方法，而不是对诸如能够影响剪辑可见性的位置和透明度等因素的反映。

用 `_visible` 属性将影片剪辑隐藏起来比将它设置为全透明或者将它移出场景更有效，因为 Flash 播放器不会对 `_visible` 被设置为 `false` 的图形进行绘制，因此提高了渲染的效率。

### 示例

下面的按钮代码将在按钮被按下的时候隐藏当前的剪辑，在按钮被释放的时候将其重新显示：

```
on (press) {  
    this._visible = false;  
}  
  
on (release) {  
    this._visible = true;  
}
```

### 参见

`MovieClip._alpha`

---

**MovieClip.\_width 属性** 剪辑或者影片的宽度，以像素表示

**有效性** Flash 4，在 Flash 5 中得到了增强

**概要** *mc.\_width*

**访问** 在 Flash 4 中为只读，在 Flash 5 中为读 / 写

### 描述

浮点 *\_width* 属性存储一个非负的数字，指定 *mc* 的当前宽度（以像素表示）。如果 *mc* 没有内容，*\_width* 就是 0。*\_width* 属性是以最左边的占据像素到最右边的占据像素之间的距离来计量剪辑或影片的内容，即使中间有空的像素也是如此。占据像素是指包含形状、图形、按钮、影片剪辑或其他元素内容的像素。在制作工具中对剪辑宽度所做的改变或通过 *\_xscale* 所做的修改都会影响到 *\_width*。

我们可以设置 *\_width* 的值，以便重新设置影片剪辑的水平尺寸。如果要将 *\_width* 设置为一个负数就会被忽略。将剪辑的 *\_width* 设置为 0 不会将剪辑隐藏起来，而是将它转换成一像素宽的垂直线。

主影片的 *\_width*（也就是 *\_root.\_width* 或者 *\_leveln.\_width*）不是在制作工具中 Modify → Movie → Dimensions 下所设置的场景宽度，而是主影片内容的宽度。并没有明确的场景宽度属性，如果需要，必须将场景宽度手动设置为一个变量。例如，如果一个影片的场景宽度为 550，我们可以添加下面的变量：

```
_root.stageWidth = 550;
```

要让这个值在任何剪辑的时间线上有效，使用：

```
Object.prototype.stageWidth = 550;
```

### 用法

注意，当我们设置影片剪辑宽度的时候，线条会被按比例进行缩放，丢失它们在描边面板中所设置的初始值。但是，如果在描边面板中将线条设置为“绒线”，线条的点尺寸就不会在影片剪辑重新定义大小的时候发生相应的变化。也就是说，使用绒线可以防止描边效果在剪辑被调用时发生变形。

### 示例

```
ball._width = 20; // 将 ball 的宽度设置为 20 像素  
ball._width *= 2; // 将 ball 的宽度加倍
```

### 参见

*MovieClip.\_height*, *MovieClip.\_xscale*

**MovieClip.\_x 属性**

剪辑或者影片的水平位置，以像素表示

**有效性** Flash 4 及其以后的版本**概要** *mc.\_x***访问** 读 / 写**描述**

浮点数 *\_x* 属性总是表示 *mc* 的记录点的水平位置，但是它的计量是相对于三个可能的坐标空间之一的：

- 如果 *mc* 在主时间线上，*\_x* 就相对于场景的左边缘来测量。例如，*\_x* 为 20 就表示 *mc* 的记录点在场景左边缘的右边 20 像素位置上，-20 就表示左边的 20 像素处。
- 如果 *mc* 在其他影片剪辑实例时间线上，*\_x* 就是相对于父实例的记录点来计量的。例如，*\_x* 为 20 表示 *mc* 的记录点在父实例的记录点的右边 20 像素位置上，-20 表示左边的 20 像素位置。
- 如果 *mc* 为主影片，*\_x* 是整个.swf 文档相对于场景左边缘的水平偏移量。例如，*\_x* 如果为 200 就表示场景的内容相对于制作中的位置向右偏离了 200 像素，-200 则是向左偏离了 200 像素。

*\_x* 属性（以及 Flash 中所有的水平坐标）都向右增长，向左减少。小数的 *\_x* 值在 Flash 中要转换为近似整数值。

如果 *mc* 包含在一个实例中，该实例被缩放并/或旋转，那么它所处的坐标空间也会被缩放并/或旋转。例如，如果 *mc* 的上一级放大到 200%，并且按照顺时针方向旋转了 90 度，那么 *\_x* 会向下增长，而不仅仅是向右，而 *\_x* 的一个单位也将由 1 变成 2。

因为实例和主影片坐标空间之间的转换可能非常麻烦，*MovieClip* 对象为坐标空间转换的执行提供了 *localToGlobal()* 和 *globalToLocal()* 方法。

**示例**

将下面的代码放在剪辑上，会让它逐帧向右移动 5 像素（假设它的坐标空间已经被转换到父剪辑空间而发生了变化）：

```
onClipEvent (enterFrame) {  
    _x += 5;  
}
```

通过 `_x` 和 `_y` 将剪辑进行定位是视觉 ActionScript 编程中的基本任务。下面的例子显示了一个剪辑事件处理器，它让一个剪辑按照固定的速度跟随鼠标而移动（可以在在线代码库中得到更多其他的动画示例）：

```
// 让剪辑跟随鼠标
onClipEvent(load) {
    this.speed = 10;

    // 将剪辑朝着点(leaderX, leaderY)移动
    function follow(clip, leaderX, leaderY) {
        // 只当我们还没有到达目的地的时候才继续移动
        if (clip._x != leaderX || clip._y != leaderY) {
            // 确定剪辑和引导之间的距离
            var deltaX = clip._x - leaderX;
            var deltaY = clip._y - leaderY;
            var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

            // 分配x和y轴上的速度
            var moveX = clip.speed * (deltaX / dist);
            var moveY = clip.speed * (deltaY / dist);

            // 如果剪辑有足够的速度到达目的地，就直接到达
            // 否则，按照剪辑的速度来移动
            if (clip.speed >= dist) {
                clip._x = leaderX;
                clip._y = leaderY;
            } else {
                clip._x += moveX;
                clip._y += moveY;
            }
        }
    }

    onClipEvent(enterframe) {
        follow(this, _root._xmouse, _root._ymouse);
    }
}

onClipEvent(load) {
    this.speed = 10;

    // 将剪辑朝着点(leaderX, leaderY)移动
    function follow(clip, leaderX, leaderY) {
        // 只当我们还没有到达目的地的时候才继续移动
        if (clip._x != leaderX || clip._y != leaderY) {
            // 确定剪辑和引导之间的距离
            var deltaX = clip._x - leaderX;
            var deltaY = clip._y - leaderY;
            var dist = Math.sqrt((deltaX * deltaX) + (deltaY * deltaY));

            // 分配x和y轴上的速度
            var moveX = clip.speed * (deltaX / dist);
            var moveY = clip.speed * (deltaY / dist);

            // 如果剪辑有足够的速度到达目的地，就直接到达
            // 否则，按照剪辑的速度来移动
            if (clip.speed >= dist) {
                clip._x = leaderX;
                clip._y = leaderY;
            } else {
                clip._x += moveX;
                clip._y += moveY;
            }
        }
    }

    onClipEvent(enterframe) {
        follow(this, _root._xmouse, _root._ymouse);
    }
}
```

## 参见

`MovieClip.globalToLocal()`, `MovieClip.localToGlobal()`, `MovieClip._y`

---

## MovieClip.\_xmouse 属性

鼠标指示器的水平位置

**有效性** Flash 5

**概要** `mc._xmouse`

**访问** 只读

## 描述

浮点值 `_xmouse` 属性表示鼠标指示器热点相对于 `mc` 坐标空间的水平位置。如果 `mc` 为主影片，那么 `_xmouse` 就从场景的左边缘开始计量。如果 `mc` 是一个实例，`_xmouse` 就从实例的记录点开始计算。要获得一个稳定的、总是相对于场景的 `_xmouse` 坐标，可以使用 `_root._xmouse`。

## 示例

将下面的代码放在一个剪辑上，可以让它反映鼠标指示器相对于场景的水平位置（它在一条直线上左右移动）：

```
onClipEvent (enterFrame) {
    _x = _root._xmouse;
}
```

## 参见

`MovieClip._ymouse`

---

## MovieClip.\_xscale 属性

一个剪辑或影片的宽度，以百分数表示

**有效性** Flash 4 及其以后的版本

**概要** `mc._xscale`

**访问** 读 / 写

## 描述

浮点数 `_xscale` 属性指定 `mc` 相对于初始宽度的百分比。如果 `mc` 为实例，那么初始宽度就是实例在库中的符号宽度；如果 `mc` 是一个主影片，它的初始宽度就是影片在制作中的宽度。

当 `mc` 的当前宽度和它的初始宽度一致的时候，`_xscale` 就是 100。`_xscale` 为 200 表示 `mc` 的宽度是初始宽度的两倍。`_xscale` 为 50 表示 `mc` 的宽度为初始宽度的一半。

`_xscale` 属性相对于它的记录点来进行缩放（剪辑记录点左右两边的部分都成比例缩放）。要仅从一边设定剪辑的尺寸，应将所有的剪辑内容放在剪辑符号记录点的一边（对于创建水平预装载条来说，这是非常有用的技术）。当剪辑的 `_xscale` 被设置为负数时，剪辑就会发生水平翻转，就像通过放在记录点上的垂直镜子一样（也就是变成自身的一个镜像），然后负值就被当作正值来对待。要水平翻转一个剪辑而不重新设置它的大小，可以将剪辑的 `_xscale` 设置为 -100。

## 示例

```
// 将ball的宽度加倍（高度不变）
ball._xscale *= 2;

// 创建ball的一个镜像
ball._xscale = -100;
```

## 参见

*MovieClip.\_yscale*

## MovieClip.\_y 属性

剪辑或者影片的垂直位置，以像素表示

**有效性** Flash 4 及其以后的版本

**概要** *mc.\_y*

**访问** 读 / 写

### 描述

浮点数 *\_y* 属性总是表示 *mc* 的记录点的垂直位置，但是它的计量是相对于三个可能的坐标空间之一的：

- 如果 *mc* 在主时间线上，*\_y* 就相对于场景的顶边缘来测量。例如，*\_y* 为 20 就表示 *mc* 的记录点在场景顶边缘的下边 20 像素处，-20 就表示上边的 20 像素处。
- 如果 *mc* 在其他影片剪辑实例时间线上，*\_y* 就是相对于父实例的记录点来计量的。例如，*\_y* 为 20 表示 *mc* 的记录点在父实例的记录点的下边 20 像素处，-20 表示上边的 20 像素处。
- 如果 *mc* 为主影片，*\_y* 是整个.swf 文档相对于场景顶边缘的水平偏移量。例如，*\_y* 如果为 200 就表示场景的内容相对于制作中的位置向下偏离 200 像素，-200 则是向上偏离 200 像素。

*\_y* 属性（以及 Flash 中所有的垂直坐标）都向下增长，向上减少——和笛卡儿坐标相反。小数的 *\_y* 值在 Flash 中要转换为近似整数值。

如果 *mc* 包含在一个实例中，该实例被缩放并/或旋转，那么它所处于的坐标空间也会被缩放并/或旋转。例如，如果 *mc* 的上一级放大到 200%，并且按照顺时针方向旋转了 90 度，那么 *\_x* 会向左增长，而不仅仅是向下，而 *\_y* 的一个单位也将由 1 变成 2。

因为实例和主影片坐标空间之间的转换可能非常麻烦，*MovieClip* 对象为坐标空间转换的执行提供了 *localToGlobal()* 和 *globalToLocal()* 方法。

## 示例

将下面的代码放在剪辑上，会让它逐帧向下移动 5 像素（假设它的坐标空间已经被转换到父剪辑空间而发生了变化）：

```
onClipEvent (enterFrame) {  
    _y += 5;  
}
```

## 参见

*MovieClip.globalToLocal()*, *MovieClip.localToGlobal()*, *MovieClip.\_x*

---

## MovieClip.\_ymouse 属性

鼠标指示器的垂直位置

**有效性** Flash 5

**概要** *mc.\_ymouse*

**访问** 只读

### 描述

浮点值 *\_ymouse* 属性表示鼠标指示器热点相对于 *mc* 坐标空间的垂直位置。如果 *mc* 为主影片，那么 *\_ymouse* 就从场景的上边缘开始计量。如果 *mc* 是一个实例，*\_ymouse* 就从实例的记录点开始计算。要获得一个稳定的、总是相对于场景的 *\_ymouse* 坐标，可以使用 *\_root.\_ymouse*。

## 示例

将下面的代码放在一个剪辑上，可以让它反映鼠标指示器相对于场景的垂直位置（它在一条直线上上下移动）：

```
onClipEvent (enterFrame) {  
    _y = _root._ymouse;  
}
```

## 参见

*MovieClip.\_xmouse*

---

## MovieClip.\_yscale 属性

一个剪辑或者影片的高度，以百分数表示

**有效性** Flash 4 及其以后的版本

**概要** *mc.\_yscale*

**访问** 读 / 写

#### 描述

浮点数 `_yscale` 属性指定 `mc` 相对于初始高度的百分比。如果 `mc` 为实例，那么初始高度就是实例在库中的符号高度。如果 `mc` 是一个主影片，它的初始高度就是影片在制作中的高度。

当 `mc` 的当前高度和它的初始高度一致的时候，`_yscale` 就是 100。`_yscale` 为 200 表示 `mc` 的高度是初始高度的两倍。`_yscale` 为 50 表示 `mc` 的高度为初始高度的一半。

`_yscale` 属性相对于它的记录点来进行缩放（剪辑记录点上下两边的部分都成比例缩放）。要仅从一边设定剪辑的尺寸，应将所有的剪辑内容放在剪辑符号记录点的一边（对于创建垂直预装载条来说，这是非常有用的技术）。当剪辑的 `_yscale` 被设置为负数时，剪辑就会发生垂直翻转，就像通过放在记录点上的水平镜子一样（也就是变成自身的一个镜像），然后负值就被当作正值来对待。要垂直翻转一个剪辑而不重新设置它的大小，可以将剪辑的 `_yscale` 设置为 -100。

#### 示例

```
// 将 ball 的高度加倍（宽度不变）
ball._yscale *= 2;
// 创建 ball 的一个镜像
ball._yscale = -100;
```

#### 参见

`MovieClip._xscale`

---

## NaN 全局属性

一个表示非法数字数据的常量（非数字）

**有效性** Flash 5

**概要** NaN

**访问** 只读

#### 描述

NaN 是一个特殊的数字常量，用来表示非法的数字数据（NaN 是“非数字”的缩写）。数字操作得不到一个合法数字的时候就产生 NaN。例如：

```
Math.sqrt(-1); // 一个不能表示出来的非法数字
15 - "coffee cup"; // "coffee cup" 不能转换为数字
```

注意，NaN 仍然属于数字类型，虽然它是一个不可计算的数字：

```
typeof NaN; // 产生 "number"
```

## 用法

NaN 值很难直接用在源代码中，但是可以作为返回错误条件的操作方法。因为 NaN 和自身比较是不相等的，因此要在全局函数 *isNaN()* 中进行探测。NaN 是 Number.NaN 的简写。

## 参见

*isNaN()*, Number.NaN, 第四章

---

## newline 常量

换行

**有效性** Flash 4 及其以后的版本

**概要** newline

### 返回

一个换行符号。

### 描述

常量 newline 表示一个标准的换行符号 (ASCII 10)。它和转义序列 "\n" 相同，用来在文本块中进行强制分行 (通常用来显示文本域变量)。

## 用法

虽然 newline 是 Flash 4 中的一个函数，但是它在 Flash 5 中变成了一个常量，有类似于变量和属性的语法。注意，newline 不使用括号。

## 示例

```
myOutput = "hello" + newline + "world";
```

## 参见

第四章

---

## nextFrame()全局函数

将影片或影片剪辑的播放头前移一帧，然后停止

**有效性** Flash 2 及其以后的版本

**概要** nextFrame()

**描述**

*nextFrame()* 函数将当前时间线的播放头前移一帧，然后停止在那里。当前时间线是调用 *nextFrame()* 函数的时间线。*nextFrame()* 函数和 *gotoAndStop(\_currentFrame+1)* 类似。如果在时间线的最后一帧上调用，*nextFrame()* 只将播放头停止在该帧上，除非有其他的场景接在当前场景后面，才会让 *nextFrame()* 将播放头移动到下一个场景的第一帧。

**参见**

*gotoAndStop()*, *MovieClip.nextFrame()*, *nextScene()*, *prevFrame()*

---

**nextScene()全局函数**

将影片的播放头前移到下一个场景的第 1 帧

**有效性** Flash 2 及其以后的版本

**概要** `nextScene()`

**描述**

*nextScene()* 函数将影片的主播放头移动到下一场景的第 1 帧，并且停止主影片时间线。当前场景是调用 *nextScene()* 函数的场景，它必须在一个场景的主时间线上调用才有效，也就是说，*nextScene()* 在影片剪辑或 *onClipEvent()* 处理器内是无效的。如果从影片的最后一个场景调用，*nextScene()* 将播放头发送到该场景的第 1 帧，并停在那里。

**参见**

*nextFrame()*, *prevScene()*

---

**Number()全局函数**

将一个值转换为数字类型

**有效性** Flash 5

**概要** `Number(value)`

**参数**

*value* 一个包含要转换为数字的值的表达式。

**返回**

将 *value* 转换为原始数字的结果。

## 描述

*Number()* 函数将它的参数转换为一个原始数字值，然后返回转换后的值。将不同数据类型转换为数字的结果如表 3-1 所示。通常没有必要使用 *Number()* 函数，ActionScript 会在需要的时候自动将值转换为数字类型。

要确定不要将全局函数 *Number()* 同 *Number* 类构造器混淆起来。前者是一个函数，将某个值转换为数字类型；后者是一个类，用来封装带属性和方法的对象中的原始数字数据。

## 用法

注意，*Number()* 函数常常出现在已经被转换为 Flash 5 格式的 Flash 4 的 *.fla* 文件中。关于 Flash 4 文件转换为 Flash 5 格式的时候如何处理数据类型的信息，请参见第三章。

## 参见

*Number* 类, *parseFloat()*, *parseInt()*, 第三章

---

## Number 类

原始数字数据的封装类

**有效性**      Flash 5

**概要**      `new Number(value)`

**参数**

*value*      一个要处理的表达式，如果需要，先转换为数字值，然后封装到 *Number* 对象中。

### 类属性

下面的属性都可以作为 *Number* 类的属性通过 *Number.propertyName* 来直接访问。要访问它们，不需要实例化一个新的 *Number* 对象（也就是说不必使用构造器函数）。有些属性，比如 *NaN*，甚至不需要 *Number.propertyName* 符号，可以仅用 *NaN* 作为 *Number.NaN* 的简写（后面有每个属性的细节内容）。

**MAX\_VALUE**      ActionScript 中可以表达的最大正数。

**MIN\_VALUE**      ActionScript 中可以表达的最小正数。

**NaN**      特殊的非数值，表示非法的数字数据。

**NEGATIVE\_INFINITY**

比 `-MAX_VALUE` 小的任何数字。

**POSITIVE\_INFINITY**

比 `MAX_VALUE` 大的任何数字。

**方法**

`toString()` 将数字转换为串。

**描述**

*Number* 类有两个作用：

- 可访问表示特殊数字值的内置属性 —— `MAX_VALUE`, `MIN_VALUE`, `NaN`, `NEGATIVE_INFINITY` 以及 `POSITIVE_INFINITY` —— 可以用来检查一个数字数据是否合法。
- 可用来在不同的数字系统之间进行转换, 比如十进制和十六进制。参考 *Number*.`toString()` 方法。

**用法**

如果只想访问对象所定义的数字属性, 就没有必要创建新的 *Number* 对象。实际上, 在恰当的地方, 使用全局属性的等价物 (`NaN`, `Infinity` 和 `-Infinity`) 很容易。其实, 我们几乎从不使用 *Number* 类属性。

另一方面, *Number* 类的 `toString()` 方法用于一个实例化的 *Number* 对象。但是, 当我们在一个原始数字值上调用方法的时候, 解释程序会小心地为我们创建一个 *Number* 对象。例如:

```
x = 102;  
x.toString(16); // x 被自动转换为一个 Number 对象, 以进行操作。
```

用 `toString()` 在不同的数字系统之间进行转换时你可能会这样做。其实, 很少会有这样的任务, 因此, 几乎不使用 *Number* 类。

**参见**

*Math* 对象, 第四章

---

**Number.MAX\_VALUE 属性**

ActionScript 中可以表示的最大正数

**有效性**

Flash 5

**概要** Number.MAX\_VALUE

**访问** 只读

### 描述

MAX\_VALUE 属性存储 ActionScript 可以表示的最大正数 ( $1.7976931348623157e+308$ )。当你想涉及一个很大的值时，这很方便，如下例所示。任何比 MAX\_VALUE 大的数字都不能在 ActionScript 中表示，因此会变成 POSITIVE\_INFINITY。

### 示例

下面，寻找数组里的一个最小值。我们初始化变量 minVal 为 MAX\_VALUE，就知道后来的任何值都会小于它：

```
var myArray = [-10, 12, 99]
var minVal = Number.MAX_VALUE
for (thisValue in myArray) {
    if (myArray[thisValue] < minVal) {
        minVal = myArray[thisValue];
    }
}
trace ("The minimum value is " , minVal);
```

### 参见

Number.MIN\_VALUE, Number.POSITIVE\_INFINITY, 第四章

---

## Number.MIN\_VALUE 属性

ActionScript 中所能表达的最小正数

**有效性** Flash 5

**概要** Number.MIN\_VALUE

**访问** 只读

### 描述

MIN\_VALUE 属性存储的是 ActionScript 中所能表达的最小的正数  $5e-324$  (不要同 -Number.MAX\_VALUE 混淆起来，后者是 ActionScript 所允许的最小负数)。

### 参见

Number.MAX\_VALUE, 第四章

---

## Number.NaN 属性

表示非法数字数据（非数字）的常量

**有效性** Flash 5

**概要** Number.NaN

**访问** 只读

### 描述

NaN属性存储了数字数据类型中特殊的非法数字值，用来表示一个不合逻辑的数字计算的结果（比如 $0/0$ ）或者一个不能产生合法数字的数据类型转换。

通常使用Number.NaN的更为方便的全局属性形式NaN。

### 用法

NaN值很少直接用在源代码中，但却可以作为返回错误条件操作的一个途径。检测NaN的唯一方法就是使用全局函数`isNaN()`。

### 参见

`isNaN()`, NaN, 第四章

---

## Number.NEGATIVE\_INFINITY 属性

表示比`-Number.MAX_VALUE`  
更小的任何数字的常量

**有效性** Flash 5

**概要** Number.NEGATIVE\_INFINITY

**访问** 只读

### 描述

NEGATIVE\_INFINITY属性存储一个特殊的数字值，用来表示比`-Number.MAX_VALUE`(ActionScript中所能表示的最负的值)更小的值。这被称为下溢条件，通常由数学错误引发。

通常使用Number.NEGATIVE\_INFINITY的更为方便的全局属性形式`-Infinity`。

### 参见

`-Infinity`, `isFinite()`, `Number.MAX_VALUE`, 第四章

---

## Number.POSITIVE\_INFINITY 属性

表示比`Number.MAX_VALUE`  
更大的任何数字的常量

**有效性** Flash 5

**概要** Number.POSITIVE\_INFINITY

**访问** 只读

### 描述

POSITIVE\_INFINITY 属性存储一个特殊的数字值，用来表示比 Number.MAX\_VALUE (ActionScript 中所能表示的最大值) 更大的值。这被称为上溢条件，通常由数学错误引发。

通常使用 Number.POSITIVE\_INFINITY 的更为方便的全局属性形式 Infinity。

### 示例

```
if (scor == Number.POSITIVE_INFINITY) {  
    trace ("You've achieved the highest possible score.");  
}
```

### 参见

[Infinity](#), [isFinite\(\)](#), [Number.MAX\\_VALUE](#), 第四章

---

## Number.toString()方法

将一个数字转换为串

**有效性** Flash 5

**概要** NumberObject.toString(radix)

### 参数

*radix* 一个 2 到 36 之间的整数，指定用来在串格式中表示 numberObject 的数字系统的底数。该参数是可选的，如果没有提供，就默认为 10。

### 返回

numberObject 转换为串的值。

### 描述

*toString()* 方法获取 *Number* 对象的值，将该值转换为一个串，然后返回串。可以使用 *radix* 参数在不同的底数之间（比如二进制、十进制、八进制和十六进制）进行数字值的转换。字母 A-Z 分别用来表示值为 10~35 的数字，虽然一般只使用 A 到 F（表示十六进制数字的等价数字 10 到 15）。

要在原始数字值上使用 *Number.toString()*，要用括号将值括起来，比如：

```
(204).toString(16);
```

## 示例

```
x = new Number(255);
trace(x.toString());           // 显示: "255" (也就是十进制)
trace(x.toString(16));         // 显示: "ff" (也就是十六进制)
trace(x.toString(2));          // 显示: "11111111" (也就是二进制)

// 将一个十六进制直接量转换为十进制的
trace(0xFFFFFFF).toString(10); // 显示: "16777215"
```

## 参见

*Number()*, *Object.toString()*, *parseInt()*, 第四章

---

## Object 类

所有其他类和一般类的基础

**有效性** Flash 5

**概要** `new Object()`

### 属性

*constructor* 一个到用来创建对象的类构造器函数的引用。

*\_proto\_* 一个到对象构造器函数的 *prototype* 属性的引用。

### 方法

*toString()* 将对象的值转换为串。

*valueOf()* 获取对象的原始值（如果存在）。

### 描述

*Object* 类是 ActionScript 对象模型的基础类。*Object* 有两个一般的用途：(a) 作为构造器来创建新的普通对象，(b) 作为超类，也就是新类的基础。ActionScript 中所有的类，不管是用户自定义的还是内置的，都是 *Object* 类的子孙。所有类的所有对象因此都继承 *Object* 的属性（虽然一些类会忽略这些属性）。

要直接在代码中创建一个 *Object* 类的类属对象而不使用构造器，可以用对象直接量，就如同使用一个串直接量或者一个数组直接量一样。一个对象直接量是一系列由逗号分隔的属性名/值对，写在花括号之中。下面是一般的语法：

`(property1: value1, property2: value2, property3: value3)`

对象直接量中属性的名称必须是合法的标识符，如第十四章中所描述的那样。值可以是任何合法的表达式。例如：

```
// 一个带两个数字属性的对象直接量  
myObject = { x: 30, y: 23 };  
// 用一个复杂表达式来设置 x 属性值  
myOtherObject = { x: Math.floor(Math.random() * 50 + 1) };
```

因为对象常量总是创建类属的、相同的对象。它们只用在需要对象格式的临时数据的时候，如，调用 *Sound.setTransform()*、*Color.setTransform()* 或者 *MovieClip.localToGlobal()* 时。

## 参见

第十二章

---

### Object.constructor 属性

一个指向用来创建对象的类构造器函数的引用

**有效性** Flash 5

**概要** *someObject.constructor*

**访问** 读/写（不提倡覆盖对象的 *constructor* 属性，因为它会改变类遗传的自然结构）。

## 描述

*constructor* 属性存储一个指向用来创建 *someObject* 的构造器函数的引用。例如，*Date* 对象的 *constructor* 属性是 *Date* 构造器函数：

```
now = new Date();  
now.constructor == Date; // 产生 true
```

## 参见

第十二章

---

### Object.\_\_proto\_\_ 属性

一个指向对象构造器的 *prototype* 属性的引用

**有效性** Flash 5

**概要** *someObject.\_\_proto\_\_*

**访问** 读/写（不提倡覆盖对象的 *proto\_\_* 属性，因为它会改变类继承的自然结构）。

## 描述

*\_\_proto\_\_* 属性存储一个指向 *someObject* 构造器函数的自动设置 *Prototype* 属性的

引用，用来在类继承中向下传递属性。对象的 `__proto__` 属性主要由翻译器在内部使用，用于寻找对象的继承属性，但我们也可以用它来确定对象的类，如例 12-6 和例 12-9 所示。注意，`__proto__` 的开头和结尾都是两个下划线符号。

## 参见

第十二章

## Object.toString()方法

对象的值，形式为串

**有效性** Flash 5

**概要** `someObject.toString()`

### 返回

一个内部定义的串，描述或表示一个对象。

### 描述

`toString()`方法返回一个描述 `someObject` 的串。默认情况下，`someObject.toString()` 返回一个表达式：

```
"[object " + class + "]"
```

`class` 是内部定义的 `someObject` 所属类的名称。ActionScript 解释程序在 `someObject` 用于串语境的时候自动调用 `toString()` 方法。例如：

```
x = new Object();
trace(x); // 显示: "[object Object]"
```

大部分类都覆盖 `Object` 的默认 `toString()` 方法，以便提供更多关于类中每个成员的有意义的信息。例如，`Date.toString()` 方法返回日期和时间，`Array.toString()` 方法返回一个由逗号分隔的数组元素列表。我们可以在构造自己的类时也这么做：

### 示例

这个例子显示了如何为我们在第十二章中所创建的 `Ball` 类提供一个自定义的 `toString()` 方法。

```
// 添加一个自定义 toString() 方法
// 制造 Ball 构造器
function Ball(radius, color, xPosition, yPosition) {
    this.radius = radius;
    this.color = color;
    this.xPosition = xPosition;
    this.yPosition = yPosition;
}
```

```
// 将一个函数直接量赋给 Ball 类 prototype 的 toString() 方法
Ball.prototype.toString = function() {
    return "A ball with the radius " + this.radius;
};

// 创建一个新的 ball 对象
myBall = new Ball(6, 0x00FFC0, 145, 200);

// 现在检查 myBall 的串值
trace(myBall); // 显示：“一个半径为 6 个单位的球”
```

**参见***Array.toString(), Date.toString(), Number.toString(), Object.valueOf()***Object.valueOf()方法**

对象的原始值

**有效性** Flash 5**概要** someObject.valueOf()**返回**

内部定义的 *someObject* 的原始值，如果一个原始值和 *someObject* 有关；如果没有原始值和 *someObject* 对应，就返回 *someObject* 本身。

**描述**

*valueOf()* 方法返回和某个对象对应的原始数据，如果对应关系存在。这个方法主要用于 *Number*, *String* 和 *Boolean* 类的对象，它们都有对应的原始数据。*MovieClip.valueOf()* 方法会返回一个表示到剪辑的路径的串。

注意，实际上很少需要显式调用 *valueOf()*，它会在 *someObject* 用于需要原始数据环境中时由解释程序自动调用。

**参见***Boolean.valueOf(), MovieClip.valueOf(), Number.valueOf(), Object.toString(), String.valueOf()***parseFloat()全局函数**

从串中提取浮点数字

**有效性** Flash 5**概要** parseFloat(*stringExpression*)

### 参数

*stringExpression* 提取浮点数字的串。

### 返回

提取到的浮点数字。如果提取失败，就返回特殊数字值 NaN。

### 描述

*parseFloat()* 函数将串转换为浮点数字（一个带小数部分的数字）。它只对包含浮点数字的合法串表达式有效；否则，就返回 NaN。串必须是下面的格式：

- 可选的开头空格。
- 可选的符号指示 + 或者 -。
- 0 到 9 中的至少一个数字，以及一个可选的小数点。
- 以 e 或 E 开头，后面跟随整数指数的可选指数。

不能作为前边数字形式的一部分而被解析的后面的字符将会被忽略。

### 用法

因为输入文本域的用户输入数据总是属于串数据类型，因此我们经常用 *parseFloat()* 从用户输入文本中提取数字数据。注意，*parseFloat()* 可以从包含数字和非数字的串中提取数字，但是 *Number()* 不能。

### 示例

```
parseFloat("14.5 apples");      // 产生 14.5
parseFloat('.123');            // 产生 0.123
var x = '15, 4, 23, 9';
parseFloat(x);                  // 产生 15
```

### 参见

*isNaN()*, *NaN*, *Number()*, *parseInt()*, 第三章, 第四章

---

## parseInt()全局函数

从串中提取一个整数，或者将数字转换为十进制数

**有效性** Flash 5

**概要** *parseInt(stringExpression)*  
*parseInt(stringExpression, radix)*

### 参数

*stringExpression* 提取整数的串。

*radix* 一个 2 到 36 之间的可选整数，指定要提取的整数的基数。如果没有指定，默认值取决于 *stringExpression* 的内容（如后所述）。

### 返回

提取到的整数值，它是一个十进制数，不管最初的 *radix* 是什么。如果提取失败，就返回特殊数字值 *Nan*。

### 描述

*parseInt()* 函数将串表达式转换为整数。它只对包含使用指定基数的整数的合法串表达式有效。串必须是下面的格式：

- 可选的开头空格。
- 可选的符号指示 + 或者 -。
- 一个或者多个数字，这些数字在指定的基数系统中是合法的。

不能作为前边数字形式的一部分而被解析的后面的字符将会被忽略。

从串中提取到的数字以串中第一个非空字符开始，以 *radix* 基数系统下的第一个非法数字的前一个字符为结束。例如，在十进制系统中，字符 *F* 不是一个合法的数字，因此下面的表达式产生数字 2：

```
parseInt('2F', 10);
```

但是在十六进制中，字符 *A*, *B*, *C*, *D*, *E* 和 *F* 都是合法的数字，因此下面的表达式将产生数字 47：

```
parseInt('2F', 16); // 十六进制中的 2F 就是十进制中的 47
```

*parseInt()* 的 *radix* 参数指定串中数字的基数。换句话说，使用 *radix* 参数，可以对解释程序说“将这个串作为一个十六进制的数字来处理”或者“将这个串当作一个二进制数”。

*parseInt()* 还认为，*0x* 前缀表示一个十六进制数字（就像将 *radix* 指定为 16 一样），而以 0 开头则表示一个八进制数字（就像将 *radix* 设置为 8 一样）：

```
parseInt("0xFF"); // 作为十六进制数，得到 255
parseInt("FF", 16); // 作为十六进制数，得到 255
parseInt("0377"); // 作为八进制数，得到 255 = (3 * 64) + (7 * 8) + (7 * 1)
parseInt("377", 8); // 作为八进制数，得到 255
```

显式的 *radix* 可覆盖任何隐式的 *radix*：

```
parseInt("0xFF", 10) // 作为十进制数，得到 0
```

```
parseInt("0X15", 10)      // 作为十进制数, 得到 0(不是 15, 也不是 21)
parseInt("03/7", 10)       // 作为十进制数, 得到 377
```

注意, *parseInt()*函数只提取整数值, 和*parseFloat()*不同, 后者也可以用来提取小数值, 但是只能应用于十进制数。

### 示例

我们主要用*parseInt()*从包含数字和文本的串中提取整数, 或者从数字中删除小数(和*Math.floor()*类似)。

```
parseInt("Wow, 20 people were there");      // 得到 NaN
parseInt("20 people were there");             // 得到 20
parseInt("1001", 2);                         // 得到 9(也就是二进制中的1001)
parseInt(1.5);                             // 得到 1(数字 1.5 在提取操作开始之前被转换成了串 "1.5")
```

### 参见

*Math.floor()*, *Nan*, *parseFloat()*, 第三章, 第四章

## play()全局函数

开始影片中帧的连续显示

**有效性** Flash 2 及其以后的版本

**概要** play()

### 描述

调用*play()*函数可以让当前主影片或者影片剪辑的帧连续显示。当前影片或影片剪辑是指包含了*play()*函数调用语句的部分。帧按照为整个影片而设置的帧速率(帧每秒, 或FPS)来显示, 它是在影片属性(*Modify* → *Movie* → *Frame Rate*)中设置的。

一旦开始播放, 影片或影片剪辑内容就会一直播放, 直到调用另外一个函数来终止播放。所有的影片剪辑都是在播放头到达时间线的末尾才开始循环的(又从第1帧开始播放)。但是, 在浏览器中, 只当用来嵌入到影片HTML页面中的代码指定影片要循环的时候, 主影片才会循环, 比如用LOOP属性来指定。(如果使用Publish命令来将影片嵌入到一个HTML页面中, 那么可以选择File → Publish Settings → HTML → Playback → Loop来设置LOOP属性。)

### 参见

*gotoAndPlay()*, *MovieClip.play()*, *stop()*

---

**prevFrame()全局函数**

将影片的播放头后退一帧，然后停止

**有效性** Flash 2 及其以后的版本

**概要** `prevFrame()`

**描述**

*prevFrame()* 函数将当前时间线的播放头后退一帧，并且停止。当前时间线是指 *prevFrame()* 函数被调用的时间线。*prevFrame()* 函数和 *gotoAndStop(\_currentFrame - 1)* 是一样的。如果在时间线上的第一帧调用，*prevFrame()* 会将播放头停止在该帧上，除非当前场景前面还有其他的场景。如果那样，*prevFrame()* 就将播放头移动到上一个场景的最后一帧。

**参见**

*gotoAndStop()*, *MovieClip.prevFrame()*, *nextFrame()*, *prevScene()*

---

---

**prevScene()全局函数**

将影片的播放头移动到上一场景的第 1 帧

**有效性** Flash 2 及其以后的版本

**概要** `prevScene()`

**描述**

*prevScene()* 函数将影片的播放头移动到当前场景的上一场景的第 1 帧，并且停止主影片时间线的播放。当前场景是指调用 *prevScene()* 函数的场景。它必须在场景的主时间线上调用才能有效，也就是说，*prevScene()* 在影片剪辑或 *onClipEvent()* 处理器中无效。如果从影片中的第一个场景调用，*prevScene()* 将播放头送到该场景的头一帧，并且停止影片的播放。

**参见**

*nextScene()*, *prevFrame()*

---

---

**print()全局函数**

用矢量方式打印影片或影片剪辑的一个帧

**有效性** Flash 5

**概要** `print(target, boundingBox)`

## 参数

<i>target</i>	一个串或者一个引用，表示到要打印的影片剪辑或文档层的路径（引用用在串语境中的时候会转换为路径）。
<i>boundingBox</i>	一个串，表示 <i>target</i> 的打印帧的打印范围。这个范围是通过边界框来定义的，表示整个打印页面。包括在打印页面中的 <i>target</i> 区域可以用 <i>boundingBox</i> 的三个合法值之一来进行设置（必须是一个合法的串）：
“ <i>bframe</i> ”	为每一个帧单独设置的边界框，以匹配每帧内容的大小。因此，每一个打印帧的内容都会进行适当的缩放来填满整个打印页面。
“ <i>bmax</i> ”	所有打印帧的内容所占据的区域被合并起来形成一个共同的边界框。每一个打印帧的内容都进行缩放，放置在和共同边界框相关的打印页面上。
“ <i>bmovie</i> ”	所有打印帧的边界框都被设置为 <i>target</i> 剪辑中一个单独的，指定的帧。打印帧的内容被裁剪到指定帧的边界框中。要将一个帧指定为边界框，可将它的标签设置为 #b。

## 描述

用 Web 浏览器的内置打印函数来打印 Flash 影片，效果会不一致，并且质量通常比较低。使用 *print()* 函数可以从 Flash 中直接打印精确的、高质量的影片内容。在默认情况下，*print()* 让 *target* 时间线上的所有帧都被发送到打印机，每页一帧，按照 *boundingBox* 参数来进行安排。要指定某个特定的帧来进行打印，可将标签 #P 赋给指定的帧。

*print()* 函数直接将矢量发送给 PostScript 打印机，将转换为位图的矢量发送给非 PostScript 打印机。因为 *print()* 使用矢量，因此它不能打印具有 alpha 透明或色彩变化的影片。要打印有色彩效果的影片，应使用 *printAsBitmap()*。

## 用法

在 Flash 4 r20 和之前版本中，各种 Flash 5 的 *print()* 函数只作为一个修定的 *getURL()* 动作。关于更多的细节，参见 Macromedia 的 Flash 打印 SDK，地址为：

<http://www.macromedia.com/software/flash/open/webprinting/authoring.html>

## 示例

```
// 打印主影片时间线上的每帧  
// 单独设置每帧的大小，填满整个页面  
print('_root', 'bfrate');  
  
// 打印主影片时间线上的每一帧  
// 将每帧按照所有帧的合并尺寸来设定大小  
print('_root', 'bmax');
```

当点击一个有下面代码的按钮时，Flash 会打印按钮时间线上的所有帧，按照标签 #b 的帧边界框进行裁减，并且进行大小设置以填满整个页面：

```
on (release) {  
    print(this, "smovie");  
}
```

## 参见

*getURL()*, *printAsBitmap()*, *printAsBitmapNum()*, *printNum()*

---

## printAsBitmap()全局函数

以位图方式打印影片或影片剪辑的帧

**有效性** Flash 5

**概要** *printAsBitmap(target, boundingBox)*

**参数**

*target* 一个串或引用，表示到要打印的影片剪辑或文档的路径（引用用于串语境的时候会被转换为路径）。

*boundingBox* 一个串，表示 *target* 的打印帧在打印的时候如何裁减，与 *print()* 的描述相同。

**描述**

*printAsBitmap()* 函数在功能上和 *print()* 一样，不过，它是将栅格化的内容而不是矢量输出到打印机，因此，它可以成功地打印颜色的变化效果，但是打印质量比矢量方式要低。

**用法**

参见 *print()* 函数下的用法说明

**参见**

*print()*, *printAsBitmapNum()*, *printNum()*

---

**printAsBitmapNum()全局函数**

以位图方式打印文档层级的帧

**有效性** Flash 5**概要** printAsBitmapNum(*level*, *boundingBox*)**参数***level* 一个非负的整数或表达式，可以产生一个非负整数，表示要打印的文档层级。*boundingBox* 一个串，表示 *target* 的打印帧在打印的时候如何进行裁减，如 *print()* 下的描述。**描述**

*printAsBitmapNum()* 函数基本上和 *printAsBitmap()* 类似，不过，它要求打印操作的目标层要用数字来指定，而不是一个串。这表示 *printAsBitmapNum()* 只可以打印文档层级，而不能打印影片剪辑。它通常用在希望能动态打印影片层的时候，比如：

```
var x = 3;  
printAsBitmapNum(x, "bmax");
```

这个效果也可以在 *printAsBitmap()* 函数中使用串连接来实现：

```
printAsBitmap("_level" + x, "bmax");
```

**用法**

参见 *print()* 函数下的用法说明。

**参见**

*print()*, *printAsBitmap()*, *printNum()*

---

**printNum()全局函数**

用矢量方式打印文档的帧

**有效性** Flash 5**概要** printNum(*level*, *boundingBox*)**参数***level* 一个非负的整数或表达式，可以产生一个非负整数，表示要打印的文档层级。*boundingBox* 一个串，表示 *target* 的打印帧在打印的时候如何进行裁减，如 *print()* 下的描述。

## 描述

*printNum()* 函数基本上和 *print()* 类似，不过，它要求打印操作的目标层 *level* 要用数字来指定，而不是一个串。这表示 *printNum()* 只可以打印文档层级，而不能打印影片剪辑。它通常用在希望能动态打印影片层的时候，比如：

```
var x = 3;  
printNum(x, "bmax");
```

这个效果也可以在 *print()* 函数中使用串连接来实现：

```
print("_level" + x, "bmax");
```

## 用法

参见 *print()* 函数下的用法说明。

## 参见

*print()*, *printAsBitmap()*, *printAsBitmapNum()*

---

## \_quality 全局属性

播放器的渲染质量

**有效性** Flash 5

**概要** \_quality

**访问** 读 / 写

## 描述

*\_quality* 属性存储一个串，表示 Flash 播放器的渲染质量，如下所示：

“*LOW*” 低质量。位图和矢量都不使用保真(平滑处理)。

“*MEDIUM*” 中等质量。对矢量进行适当的保真。

“*HIGH*” 高质量。矢量高度保真，位图在没有动画的时候进行保真处理。

“*BEST*” 最好质量。位图和矢量始终都有保真处理。

较低的渲染质量表示在绘制影片的帧时需要的计算较少，执行更快。对于大部分影片来说，*\_quality* 通常设置为“*HIGH*”。

## 示例

下面，我们将一个影片的渲染质量设置为它能达到的最好等级（会让播放最慢）：

```
_quality = "BEST";
```

**参见**

`_highQuality`, `toggleHighQuality()`

---

**random()全局函数**

产生一个随机数字

**有效性** Flash 4, 在 Flash 5 中因为 `Math.random()` 的出现而得到了加强。

**概要** `random(number)`

**参数**

`number` 一个正整数。

**返回**

一个大于或等于 0 并小于 `number` 的随机整数。

**描述**

在 Flash 4 中不赞成用 `random()` 来生成随机数字, 这个函数在该语言中已不再使用了, 只是为了向后兼容的原因而存在。在 Flash 5 以及更高的版本中, 使用首选的 `Math.random()` 方法。注意, `random()` 产生从 0 到 `number - 1` 的整数, 但是 `Math.random()` 产生从 0.0 到 .999 的浮点数字。

**参见**

`Math.random()`

---

**removeMovieClip()全局函数**

从 Flash 播放器中删除一个影片剪辑

**有效性** Flash 4, 在 Flash 5 中可以应用于用 `attachMovie()` 所创建的实例

**概要** `removeMovieClip(target)`

**参数**

`target` 一个串或一个引用, 表示到要从播放器删除的影片剪辑实例的路径 (引用用于串语境的时候会转换为路径)。

**描述**

`removeMovieClip()` 函数从播放器中删除指定的影片剪辑, 并且不留剪辑的任何内容或外壳。后来对该剪辑或者它的任何变量或属性的引用都将会产生 `undefined`。

`removeMovieClip()` 函数只可以用在最初通过 `duplicateMovieClip()` 或者 `attachMovie()` 所创建的影片剪辑实例上。它对在制作工具中创建的影片剪辑并没有影响。

**参见**

*attachMovie()*, *duplicateMovieClip()*, *MovieClip().removeMovieClip()*, 第十三章

**\_root 全局属性**

一个指向当前层级的主影片时间线的引用

**有效性** Flash 5 (和 Flash 4 影片中的 "/" 符号相同)

**概要** `_root`

**访问** 只读

**描述**

`_root` 属性存储了一个指向当前文档层级的主时间线的引用。可以用 `_root` 在主影片上调用方法，或者获取和设置主影片的属性。例如：

```
_root.play();      // 播放主时间线  
_root._alpha = 40; // 让影片部分透明
```

`_root` 属性也可以用来指向嵌套剪辑，例如：

```
_root.myClip.play();  
_root.shapes.square._visible = false;
```

`_root` 属性提供了对影片剪辑的绝对形式访问。也就是说，在影片的任何地方使用从 `_root` 开头的引用是合法的（不变的）。

**参见**

`_leveln`, `_parent`, 第十三章, 第二章

**Scroll 属性**

显示在文本域中的当前顶行

**有效性** Flash 4 及其以后的版本

**概要** `textfield.scroll`

**返回**

一个表示文本域中最顶层可视行的行号的正整数。

**描述**

`scroll` 文本域属性可以获取和设置。当获取文本域的 `scroll` 属性值时，它表示当前显示的是域中可视区域第一行的行号。当设置 `scroll` 的值时，它将滚动文本域，让指定的行成为文本域中可视区域的顶行。`scroll` 属性通常和 `maxscroll` 一起使用，创建文本滚动界面，如第十八章中所示。

## 用法

虽然 scroll 在 Flash 中被当做函数，但是它也可作为文本域变量的一个属性。注意调用 scroll 的时候不使用括号。

## 故障

在 Flash 播放器的修订版 5.0 r30 中，当文本域的字体被嵌入时，使用 scroll 会导致一些文本显示在文本域的可视区域之外，一些文本在文本域滚动的时候不会被删除。要解决这个问题，可以在文本域层上使用一个遮罩。这个问题在修订版 5.0 r41 中得到了修正。

## 示例

```
// 将 x 设置为 myField 文本域中所显示的顶行的索引  
var x = myField.scroll;  
// 将 myField 中的文本向上滚动一行  
myField.scroll++;
```

## 参见

maxscroll, 第十八章

---

## Selection 对象

控制文本域的选择

**有效性** Flash 5

**概要** Selection.methodName()

### 方法

*getBeginIndex()* 获取第一个所选字符的索引。

*getCaretIndex()* 获取插入点的索引。

*getEndIndex()* 获取最后一个所选字符的索引。

*getFocus()* 确定插入点当前所在的文本域。

*setFocus()* 将插入点放在指定的文本域中。

*setSelection()* 在当前焦点文本域中选择字符。

### 描述

我们使用 *Selection* 对象控制文本域用户交互，并捕获文本域的部分。在实际应用中，*Selection* 对象方法以关键词 Selection 开头；它们总是在焦点文本域上进行操作，焦点文本域在任何给定的时间里都只能有一个。*Selection* 对象的方法可以将插入点进行

定位，也可以选择或获取文本域中的内容。这些能力可以为用户输入系统提供细微然而重要的功效。例如，在一个用户登录屏幕中，可以通过将光标放在一个名称文本域中以提示用户要输入自己的名字。也可以选择有问题的文本以突出一个错误。还可以自定义所谓的文本域 Tab 顺序，如“*Selection.setFocus()*方法”下所示。

字符在文本域中的位置引用和 0 相对的索引，因此，第一个字符为索引 0，第 2 个字符为索引 1，依次类推。字符索引在第四章中有详细描述。

在 Flash 5 中，不能程序化地剪切、复制或者粘贴文本。而且，剪切、复制和粘贴的快捷键，比如 Ctrl-V 和 ctrl-C 在 Flash 播放器中都是无效的。次鼠标键（在 Windows 中的右键，在 Macintosh 中的 Ctrl-click）可以提供对剪切、复制和粘贴命令的使用。

## 用法

点击鼠标时，表单的提交按钮自动从任何先前的焦点文本域中删除焦点。要在失去焦点之前获得一个选择，可以使用按钮的 *rollover* 事件。例如：

```
on (rollOver) {  
    focusedField = Selection.setFocus();  
}
```

## 参见

*\_focusrect*, *Selection.setFocus()*, 第十八章

---

## Selection.getBeginIndex()方法

获取文本域所选的第一个字符的索引

**有效性** Flash 5

**概要** *Selection.getBeginIndex()*

### 返回

当前选择（高亮文本块）的第一个字符的索引。如果文本域没有键盘焦点，它就返回 -1。如果文本域虽然有焦点，但是没有选择任何字符，那么它就返回 *Selection.getCareIndex()* 的值。

### 描述

*getBeginIndex()* 方法确定一个选择的开头。要确定当前所选的字符域，应同时使用 *getBeginIndex()* 和 *getEndIndex()*。

### 示例

下面的例子创建了一个串，表示当前所选的字符，然后将这个串显示在输出窗口中：

```
var firstChar = Selection.getBeginIndex();
var lastChar = Selection.getEndIndex();
var currentSelection = eval(Selection.getFocus()).substring(firstChar, lastChar);
trace(currentSelection);
```

下面的代码将当前的选择向左扩展一个字符：

```
Selection.setSelection(Selection.getBeginIndex() - 1, Selection.getEndIndex());
```

## 参见

*Selection.getCaretIndex()*, *Selection.getEndIndex()*

---

**Selection.getCaretIndex()方法**      获取文本域所选的第一个字符的索引

**有效性**      Flash 5

**概要**      `Selection.getCaretIndex()`

## 返回

当前文本域插入点的索引。如果文本域没有键盘焦点，就返回 -1。如果带焦点的文本域是空的，就返回 0。

## 描述

*getCaretnIndex()*方法表示一个文本域中的插入点(也就是光标的位置)。当文本域有键盘焦点的时候，光标显示为一个 I 形。使用 *setSelection()*可以设置插入点的位置。

## 示例

因为 *getCaretnIndex()*在文本域都没有焦点的时候返回 -1，因此，可以通过检查 *getCaretnIndex()*是否等于 -1 而确定该文本域是否有焦点。比如：

```
if (Selection.getCaretIndex() == -1) {
    trace("No field has focus");
}
```

## 参见

*Selection.setSelection()*

---

**Selection.getEndIndex()方法**      获取文本域所选的最后一个字符的索引

**有效性**      Flash 5

**概要**      `Selection.getEndIndex()`

## 返回

当前选择(高亮文本块)的最后一个字符之后字符的索引。如果文本域没有键盘焦点, 它就返回 -1。如果文本域虽然有焦点, 但是没有选择任何字符, 那么它就返回 *Selection.getCaretIndex()* 的值。

## 描述

*getEndIndex()*方法确定一个选择的结尾。要确定当前所选的字符范围, 可同时使用 *getBeginIndex()* 和 *getEndIndex()*。

## 示例

下面的代码将当前的选择向右扩展一个字符:

```
Selection.setSelection(Selection.getBeginIndex(), Selection.getEndIndex() + 1);
```

## 参见

*Selection.getBeginIndex()*, *Selection.getCaretIndex()*

---

## Selection.setFocus()方法

确定光标当前所在的文本域

**有效性** Flash 5

**概要** `Selection.setFocus()`

## 返回

一个串, 表示到拥有键盘焦点的文本域变量(也就是光标当前所在的文本域)的路径, 例如, "`_level1.myTextField`"。如果文本域没有键盘焦点, 那么它就返回 `null`。

## 描述

*getFocus()*方法确定哪个文本域当前有键盘焦点(如果有焦点), 并返回到该文本域的路径串。要将该串转换为变量引用, 可以使用 *eval()*。例如, 在下面的代码中, 我们确定焦点文本里的字符数。我们调用 *getFocus()*, 获取该文本域的名称, 并且将名称用 *eval()* 转换为一个变量:

```
var numChars = eval(Selection.getFocus()).length;
```

## 示例

因为 *getFocus()*在没有选中任何文本域的时候返回 `null`, 所以可以通过检查 *getFocus()*的返回值是否为 `null`, 从而确定文本域是否拥有焦点, 比如:

```
if (Selection.getFocus() == null) {  
    trace("No field has focus");  
}
```

## 参见

*Selection.setFocus()*, *eval()*, *Selection.getCaretIndex()*

---

## Selection.setFocus()方法

为指定的文本域设置焦点

**有效性** Flash 5

**概要** `Selection.setFocus(variableName)`

**参数**

**variableName** 一个串，表示到要获取焦点的文本域变量的路径（也就是“\_root.myTextField”或“userName”）。

**返回**

一个布尔值，表示焦点设置成功 (`true`) 或者失败 (`false`)。只当 `variableName` 所指定的变量没有找到，或者它不是一个文本域变量的时候，焦点设置才会失败。

**描述**

*setFocus()*方法为文本域设置键盘焦点。它将光标放在文本域中，通常是为了突出该文本域，或者允许用户输入。例如，在在线表单中，用户可能没有输入 email 地址，要提醒用户这个错误，可以将焦点设置到 email 地址文本域中，让用户填写。也可以用 *setFocus()* 来为表单中的各文本域创建一个自定义的 Tab 键顺序，如下例所示。

**用法**

注意，文本域焦点设置将自动选择该域中的任何内容。要在设置文本域焦点的时候不选择任何字符，可以使用下面的代码：

```
// 首先，为myField设置焦点  
Selection.setFocus('myField');  
  
// 现在，将插入点放在myField的开头  
Selection.setSelection(0, 0);  
  
// 或者在myField的末尾  
Selection.setSelection(myField.length, myField.length);
```

---

**注意：**当在浏览器中浏览影片的时候，焦点文本域只当 Flash 影片本身有焦点（也就是用户在影片播放期间点击影片上的某一点）时才会接收输入。要在让用户输入某个文本域之前确保影片有焦点，方法是在影片的开头使用一个“Click Here to Start”按钮。

---

## 示例

这个例子显示了如何为填充表单中的文本域设置 Tab 顺序。对应的 fla 范例文件可以从在线代码库中下载（关于指定 Tab 键的更多信息，请参见第十章）：

```
// 自定义 Tab 顺序
// 包含表单域的影片剪辑上的代码
onClipEvent (load) {
    // 将到该剪辑的路径存储在一个串中
    // 我们可以在调用 Selection.setFocus() 的时候使用
    path = targetPath(this);
    // 用文本域的名称来创建一个数组，应用于所需的 Tab 顺序
    // 所列的第一个域是默认的
    tabOrder = ["field", "field3", "field2", "field4"];
}

onClipEvent (keyUp) {
    // 如果 Tab 键被按下……
    if (Key.getCode() == Key.TAB) {
        // ……如果域没有焦点……
        if (Selection.getFocus() == null) {
            // ……然后将焦点设置给域数组中的第一个域
            Selection.setFocus(path + '.' + tabOrder[0]);
        } else {
            // 否则，在域数组中定位当前的焦点域
            i = 0;
            focused = Selection.getFocus();
            while (i < tabOrder.length) {
                // 从 Selection.getFocus 所返回的完整路径中提取域变量的名称
                fieldName = focused.substring(focused.lastIndexOf('.') + 1);
                // 检查 tabOrder 的每一个元素，寻找焦点域
                if (tabOrder[i] == fieldName) {
                    // 发现匹配的时候就停止检查
                    currentFocus = i;
                    break;
                }
                i++;
            }
            // 既然我们知道焦点域在 tabOrder 数组中，
            // 就可以将新的焦点设置给下一个域或者上一个域。
            // 这要依靠 Shift 键是否被按下
            if (Key.isDown(Key.SHIFT)) {
                // Shift 键被按下，因此可以到前边的一个域，除非我们已经在最开头,
                // 如果我们在最开头，就会跑到最后一个域中去
                nextFocus = currentFocus-1 == -1 ? tabOrder.length-1 : currentFocus-1;
            } else {
                // Shift 键没有被按下，所以到下一个域,
                // 除非我们已经在最后，如果我们在最后，那么就到第一个域中
                nextFocus = currentFocus+1 == tabOrder.length ? 0 : currentFocus+1;
            }
            // 最后，我们要设置新的焦点
            Selection.setFocus(path + '.' + tabOrder[nextFocus]);
        }
    }
}
```

```
        }
    }

// 主时间线按钮上的代码
on (keyPress '<Tab>') {
    // 这个占位符代码只在主时间线上对 Tab 键有效
    var tabCatcher = 0;
}
```

## 参见

*Selection.setFocus()*, *Selection.setSelection()*

---

**Selection.setSelection()方法** 选择焦点文本域中的字符，或者设置插入点

**有效性** Flash 5

**概要** `Selection.setSelection(beginIndex, endIndex)`

### 参数

*beginIndex* 一个非负的整数，指定要包括在新选择中的第一个字符的索引。

*endIndex* 一个非负的整数，指定包括在新选择中的最后一个字符之后字符的索引。

### 描述

*setSelection()*方法选择（高亮）焦点文本域中从 *beginIndex* 到 *endIndex-1* 的字符。如果文本域没有焦点，那么 *setSelection()* 就无效。它一般用来突出有问题的用户输入。

### 用法

虽然 *Selection* 对象没有 *setCaretIndex* 方法，但是可以用 *setSelection()* 方法在文本域中的指定位置设置插入点。要这么做，可以指定相同的 *beginIndex* 和 *endIndex* 值。比如：

```
// 将插入点设置在第 3 个字符的后面
Selection.setSelection(3, 3);
```

### 示例

```
// 选择当前焦点文本域中的第 2 和第 3 个字符
Selection.setSelection(1, 3);
```

**参见**

*Selection.getBeginIndex()*, *Selection.getCaretIndex()*, *Selection.getEndIndex()*

**setProperty()全局函数**

为影片剪辑属性赋值

**有效性** Flash 4 及其以后的版本

**概要** `setProperty(movieClip, property, value)`

**参数**

*movieClip* 一个能产生串的表达式，表示到影片剪辑的路径。在 Flash 5 中，也可以是一个影片剪辑引用，因为影片剪辑引用应用于串语境的时候会转换为路径。

*property* 将被赋予 *value* 的内置属性的名称。必须是一个标识符，不能是一个串（例如，应是 `_alpha`，而不是 `"_alpha"`）。

*value* 一个新的数据值，要赋给指定的 *movieClip* 的 *property*。

**描述**

*setProperty()* 函数将 *value* 赋给 *movieClip* 的一个内置属性（内置属性列在 *MovieClip* 类下面）。它不能用来设置自定义属性（也就是用户定义的）的值。在 Flash 4 中，*setProperty()* 是为影片剪辑属性赋值的惟一途径；而在 Flash 5 中，`.` 和 `[]` 操作符是设置内置的和自定义的影片剪辑属性的首选方式。

**示例**

```
// Flash 4 语法，将主影片旋转 45 度。  
setProperty("_root", _rotation, 45);  
  
// Flash 5 语法，也将主影片旋转 45 度。  
_root._rotation = 45;
```

**参见**

*getProperty()*, 第十三章, 附录三

**Sound 类**

控制影片中的声音

**有效性** Flash 5

**概要** `new Sound()`  
`new Sound(target)`

### 参数

*target* 一个串，表示到要控制其声音的影片剪辑或文档层级的路径，也可以是一个到影片剪辑或文档层级的引用（引用使用于串语境的时候会转换为路径）。

### 方法

*attachSound()* 将库中的某个声音同 *Sound* 对象相联合。

*getPan()* 获取当前的概貌设置。

*getTransform()* 确定声音当前的频道分配是左边的扬声器还是右边的扬声器（也就是平衡）。

*getVolume()* 获取当前的音量。

*setPan()* 通过声音的左右频道设置概貌。

*setTransform()* 在左右扬声器之间分配左右频道（也就是平衡）。

*setVolume()* 设置声音的音量。

*start()* 开始播放所添加的声音。

*stop()* 停止所有声音或者某个指定的声音。

### 描述

*Sound* 类的对象用来控制影片中现有的声音，或者控制程序化添加到影片中的声音。

*Sound* 对象有几种不同的用途，它们可以控制：

- Flash 播放器中所有的声音。
- 特定影片剪辑实例或主影片中的声音（包括任何嵌套剪辑中的声音）。
- 程序化添加的单个声音。

要创建 *Sound* 对象来控制播放器中的所有声音（包括文档层级中的.swf文件），可以用不带参数的 *Sound* 构造器。例如：

```
myGlobalSound = new Sound();
```

要创建一个 *Sound* 对象来控制特定剪辑或主影片中的所有声音，可以用 *target* 参数来表示要控制的剪辑或影片。注意，这也将控制 *target* 中剪辑的声音。例如：

```
spaceshipSound = new Sound('spaceship'); // 控制 spaceship 剪辑中的声音  
mainSound      = new Sound("_root");    // 控制主时间线上的声音
```

要让单个的声音可以开始、停止、以及单独循环，可以先创建任意种类的 *Sound* 对象，然后用 *attachSound()* 方法为它添加一个声音。

## 参见

*stopAllSounds()*, *\_soundbuftime*

## Sound.attachSound()方法

将库中的声音和一个 *Sound* 对象相联合

**有效性** Flash 5

**概要** *soundObject.attachSound(linkageIdentifier)*

### 参数

*linkageIdentifier* 要添加的声音的名称，如同 Options → Linkage 命令下指定库中的描述一样。

### 描述

*attachSound()* 方法在运行中为影片添加一个新的声音，并且将新的声音放在 *soundObject* 的控制之下。声音一旦添加，就可以通过在 *soundObject* 上调用 *start()* 和 *stop()* 来单独地开始和停止。

为了将声音添加到 *soundObject*，声音必须从影片库中导出。要导出一个声音，执行下面的步骤：

1. 在 Library 中，选择要导出的声音。
2. 选择 Options → Linkage 命令。符号链接属性对话框就会出现。
3. 选择 Export This Symbol。
4. 在 Identifier 符框中，输入指定该声音的特定名称。

注意，所有的导出声音都在包含它们的影片的第 1 帧处装载（而不是在它们实际通过 ActionScript 而添加或者播放的时候），如果声音文件很大，会使装载时间过长。最好将声音放在外部 .swf 文件中，然后在必要的时候用 *loadMovie()* 输入它们。

### 用法

一次只能将一个声音添加到 *Sound* 对象。将一个新的声音添加到 *Sound* 对象将替代先前添加到该对象中的任何声音。注意，*attachSound()* 对通过 *loadMovie()* 装载到剪辑或者层级的影片无效，除非添加的声音在 *Sound* 对象作用域所在的文档库中有效。创建时不带 *target* 参数的全局声音对象，其作用域为 *\_level0*。

## 示例

下面的例子将一个标识符为 phaser 的声音添加到 Sound 对象 phaserSound，然后，播放和停止 phaser 声音：

```
phaserSound = new Sound();
phaserSound.attachSound("phaser");

// 开始播放 phaser 声音
phaserSound.start();

// 停止 phaser 声音的播放
phaserSound.stop("phaser");
```

## 参见

*Sound.start()*, *Sound.stop()*

---

## Sound.getPan()方法

获取最近的概貌值设置

**有效性** Flash 5

**概要** *soundObject.getPan()*

**返回**

一个数字，表示由 *setPan()* 所设置的最新值。通常范围为 -100（左频道开，右频道关）到 100（右频道开，左频道关）。默认值为 0（两边的频道为相同属性）。

**描述**

通过调节声音的概貌，可以创建运动声源的错觉。*getPan()* 方法用来确定由 *soundObject()* 所控制的声音当前的左右频道分配。通常，*getPan()* 和 *setPan()* 结合使用，可调整声音的当前概貌。

**示例**

下面，我们将声音的概貌改变 20：

```
mySound = new Sound();
mySound.setPan(mySound.getPan() + 20);
```

**参见**

*Sound.getTransform()*, *Sound.setPan()*

---

**Sound.getTransform()方法**

确定声音当前针对左右扬声器的频道分配

**有效性** Flash 5

**概要** `soundObject.getTransform()`

**返回**

一个匿名的对象，它的属性包含由 *soundObject* 控制的声音的频道百分比值。

**描述**

*getTransform()* 方法返回一个对象，它的属性告诉我们 *soundObject* 所控制的声音频道如何分配到左右扬声器。返回对象的属性为 *lL*, *lR*, *rL* 和 *rR*，与 *Sound.setTransform()* 方法下面的描述相同。

**参见**

*Sound.getPan()*, *Sound.setTransform()*

---

---

**Sound.getVolume()方法**

获取当前的音量设置

**有效性** Flash 5

**概要** `soundObject.getVolume()`

**返回**

一个数字，表示由 *setVolume()* 所设置的当前音量。通常在范围 0 (无音量) 到 100 (默认音量) 之间，但是也可以更高。

**描述**

*getVolume()* 方法用来确定 *soundObject* 所控制的声音的当前音量。通常，*getVolume()* 和 *setVolume()* 一起使用，以调整声音的当前音量。

**示例**

下面，我们将声音的音量减少 20:

```
mySound = new Sound();
mySound.setVolume(mySound.getVolume() - 20);
```

**参见**

*Sound.setVolume()*

---

## Sound.setPan()方法

设置声音的左右频道平衡

**有效性** Flash 5

**概要** `soundObject.setPan(pan)`

**参数**

*pan* 一个 -100 (左) 到 100 (右) 之间的数字，表示左右扬声器之间对于 *soundObject* 所控制的声音的分配情况。如果 *pan* 大于 100，实际的值就被设置为  $200 - pan$ 。如果 *pan* 小于 -100，实际的值就设置为  $-200 - pan$ 。

### 描述

*setPan()* 方法规定 *soundObject* 所控制的声音的左右频道平衡。通过不同时间对声音概貌的调节，可以让声音从一个扬声器移动到另外一个扬声器（也就是移动镜头）。

如只在左扬声器中播放 *soundObject* 所控制的声音，应使用 -100 的 *pan* 值。如只在右扬声器中播放 *soundObject* 所控制的声音，应使用 100 的 *pan* 值。要平衡两个频道，可让 *pan* 为 0。

注意，*setPan()* 会影响 *soundObject* 所控制的所有声音。如果 *soundObject* 是一个全局声音，*setPan()* 就影响影片中的所有声音；如果 *soundObject* 属于一个剪辑或主时间线，*setPan()* 就影响该剪辑或时间线以及它所包含的所有剪辑中的所有声音。

*setPan()* 的效果只可以通过另一个 *setPan()* 的调用来改变。一次 *setPan()* 赋值可以影响以后 *soundObject* 所控制的所有的声音，即使 *soundObject* 被删除。

### 示例

下面的剪辑事件处理器让影片剪辑中的声音在左右扬声器之间无休止地移动：

```
onClipEvent (load) {
    panEffect = new Sound(this);
    panDirection = "right";
    panIncrement = 50;
}

onClipEvent (enterFrame) {
    if (panDirection == "right") {
        newPan = panEffect.getPan() + panIncrement;
        if (newPan > 100) {
            panDirection = "left";
            panEffect.setPan(panEffect.getPan() - panIncrement);
        } else {
            panEffect.setPan(newPan);
        }
    }
}
```

```

        }
    } else {
        newPan = panEffect.getPan() - panIncrement;
        if (newPan < -100) {
            panDirection = "right";
            panEffect.setPan(panEffect.getPan() + panIncrement);
        } else {
            panEffect.setPan(newPan);
        }
    }
}

```

下面的剪辑时间处理器让剪辑中的声音对鼠标作出响应。假设一个场景的宽和高分别为 550 和 400，声音的平衡随着鼠标的水平移动而左右变换，音量随着鼠标的垂直移动而提高或降低：

```

onClipEvent (load) {
    // 创建一个新的声音对象，然后将声音 bgMusic 添加进去
    mySound = new Sound(this);
    mySound.attachSound("bgMusic");
    mySound.start(0, 999);           // 播放和循环声音
}

onClipEvent (enterFrame) {
    // 测量鼠标的水平位置，然后相应地设置 pan
    mouseX = (_root._xmouse / 550) * 200;
    mySound.setPan(mouseX - 100);
    // 测量鼠标的垂直位置，然后相应地设置音量
    mouseY = (_root._ymouse / 400) * 300;
    mySound.setVolume(30 + mouseY);
}

```

## 参见

*Sound.getPan()*

---

## Sound.setTransform()方法

在左右扬声器之间分配左右频道

**有效性** Flash 5

**概要** soundObject.setTransform(*transformObject*)

**参数**

*transformObject* 一个用户定义的对象，用来指定新的频道设置的一系列属性。

**描述**

*setTransform()*方法可以精确控制声音的频道如何输出到左右扬声器中。原则上，*setTransform()*和*setPan()*不同，它提供了对立体声的更多细节控制。

立体声是两个不同声音的合并——左频道和右频道——这两个声音通常是单独发送到左右扬声器中的。但是，使用 *setTransform()*，可以表示每个频道在每个扬声器中播放多少。例如，我们可以说“在左扬声器中播放一半左频道，不在右扬声器中播放左频道，在两个扬声器中都播放全部的右频道”或者说“在左扬声器中播放全部的左右频道”。

要使用 *setTransform()*，必须首先创建一个带一系列预置属性的对象。这些属性表现如何在左右扬声器之间分配立体声音的左右频道，如表 R-12 所述。

表 R-12 transformObject 的属性

属性名称	属性值	属性描述
l1	0 到 100	在左扬声器中播放的左频道的百分比
lr	0 到 100	在左扬声器中播放的右频道的百分比
r1	0 到 100	在右扬声器中播放的左频道的百分比
rr	0 到 100	在右扬声器中播放的右频道的百分比

一旦创建了带表 R-12 所描述的属性的对象，就可以将这个对象传递给 *Sound* 对象的 *setTransform()* 方法。*transformObject* 上的属性值将成为 *soundObject* 所控制的声音的新的频道输出百分比。

要检查特定 *Sound* 对象当前的百分比，可使用 *Sound.getTransform()* 方法。

## 示例

```
// 创建一个新的 Sound 对象
mySound = new Sound();

// 创建一个新的类属对象，用于 setTransform()
transformer = new Object();

// 设置转换对象的属性
transformer.l1 = 0;          // 在左扬声器中不播放左频道
transformer.lr = 0;          // 在左扬声器中不播放右频道
transformer.r1 = 0;          // 在右扬声器中不播放左频道
transformer.rr = 100;         // 在右扬声器中播放全部右频道

// 将变换对象传递给 setTransform() 方法，以应用新的频道分配
mySound.setTransform(transformer);
```

## 参见

*Sound.getTransform()*, *Sound.setPan()*

**Sound.setVolume()方法**

设置 Sound 对象所控制的声音的音量

**有效性** Flash 5**概要** `soundObject.setVolume(volume)`**参数**

*volume* 一个数字，表示 *soundObject* 所控制的声音的音量，0 代表没有音量（静音）。*volume* 的绝对值（不管 *volume* 是正是负）越大，*soundObject* 所控制的声音就越响亮。例如，*volume* 为 -50 和 50 的时候是一样的。*volume* 的默认值为 100。

**描述**

*setVolume()* 方法让 *soundObject* 所控制的声音更强或更弱。要完全达到静音，可以将 *volume* 设置为 0。要让一个声音更响亮，可以增加 *volume*。100 到 200 这个范围内的值会非常响亮，但是并没有预定的最大值。

注意，*setVolume()* 会影响 *soundObject* 所控制的所有声音。如果 *soundObject* 是一个全局声音，*setVolume()* 就影响影片中的所有声音；如果 *soundObject* 属于一个剪辑或者主时间线，*setVolume()* 就会影响该剪辑或时间线上的所有声音。

*setVolume()* 所产生的效果会持续到其他 *setVolume()* 调用的时候。一次 *setVolume()* 赋值将影响以后 *soundObject* 所控制的所有声音，甚至当 *soundObject* 被删除后。

**示例**

第一个例子将简单设置影片剪辑的音量：

```
var mySound = new Sound();
mySound.setVolume(65);
```

下面的例子显示了如何创建用来控制影片音量的按钮：

```
// 主影片时间线上的代码
var globalSound = new Sound();
var maxVolume = 200;
var minVolume = 0;
var volumeIncrement = 20;

// 主时间线上的音量增加代码
on(release) {
    globalSound.setVolume(Math.min(globalSound.getVolume() + volumeIncrement,
                                   maxVolume));
}
```

```
// 主时间线上的音量减小代码  
on (release) {  
    globalSound.setVolume(Math.max(globalSound.getVolume() - volumeIncrement,  
        minVolume));  
}
```

## 参见

*Sound.getVolume()*

## Sound.start()方法

开始播放添加的声音

**有效性** Flash 5

**概要** *soundObject.start(secondOffset, loops)*

### 参数

*secondOffset* 一个浮点数字，表示开始播放添加到 *soundObject* 的声音的时间，以秒计算（通常称为切入点）。例如，*secondOffset* 如果为 1，就表示从库中声音实际定义开始时间之后的一秒开始播放。默认值为 0。没有规定退出点（也就是停止播放声音的时间）。如果不手动停止，声音会一直播放到结束。

*loops* 一个正整数，表示要播放添加到 *soundObject* 的声音的次数。如果只播放一次，就使用 1（这也是默认值）；要连接播放两次，就用 2，依此类推。从 *secondOffset* 开始的部分将重复 *loops* 所规定的次数。

### 描述

*start()* 方法用来播放通过 *attachSound()* 添加到 *soundObject* 的程序化定义的声音。*start()* 方法不播放剪辑或影片中的任何声音，它只播放最近通过 *attachSound()* 添加到 *soundObject* 的声音。

如果只播放添加到 *soundObject* 的声音的一部分，可以使用 *secondOffset* 参数。要重复播放添加到 *soundObject* 的声音，可使用 *loops* 参数。

### 示例

```
// 创建一个新的 Sound 对象  
boink = new Sound();  
  
// 将一个导出为 boink 的声音添加到 sound 对象  
boink.attachSound("boink");
```

```
// 播放全部 boink, soundOffset 默认为 0  
boink.start();  
  
// 播放 boink 的一部分, 从 0.5 秒之后开始、循环次数默认为 1  
boink.start(.5);  
  
// 从头到尾播放 boink 三次  
boink.start(0, 3);
```

## 参见

*Sound.stop()*

---

## Sound.stop()方法

禁止所有的声音或指定的添加声音

**有效性** Flash 5

**概要** `soundObject.stop()`

`soundObject.stop(linkageIdentifier)`

**参数**

*linkageIdentifier* 添加到任何 *Sound* 对象的任何声音的名称, 该对象和 *soundObject* 有相同的 *target*。链接标识符是在 Options → Linkage 中指定的。

**描述**

调用的时候如果没有使用 *linkageIdentifier*, *stop()* 方法就停止 *soundObject* 所控制的所有声音。如果 *soundObject* 是一个全局声音, *stop()* 就禁止影片中的所有声音; 如果 *soundObject* 是用 *target* 参数来创建的, *stop()* 就停止 *target* 中的所有声音。

如果调用的时候带 *linkageIdentifier* 参数, *stop()* 方法就只停止 *linkageIdentifier* 所指定的声音。在这种情况下, *linkageIdentifier* 必须是一个通过 *attachSound()* 添加到 *Sound* 对象的声音。但是, 要禁止的声音不是添加到 *soundObject* 本身的, 它可能是添加到和 *soundObject* 有相同 *target* 的任何 *Sound* 对象上的。如果 *soundObject* 在创建的时候没有 *target* (也就是说, 它是一个全局 *Sound* 对象), 那么要停止的声音可能是添加到其他任何全局 *Sound* 对象上的。

**示例**

```
// 创建一个全局 Sound 对象  
mySound = new Sound();  
  
// 将声音 doorbell 添加到对象  
mySound.attachSound("doorbell");  
  
// 停止影片中的所有声音
```

```
mySound.stop();  
// 播放doorbell  
mySound.start();  
  
// 只停止doorbell  
mySound.stop("doorbell");  
  
// 创建另外一个全局 Sound 对象  
myOtherSound = new Sound();  
  
// 将doorknock 声音添加到该对象  
myOtherSound.attachSound('doorknock');  
  
// 播放doorknock  
myOtherSound.start();  
  
// 现在通过 mySound 来停止doorknock, 而不是通过 myOtherSound  
// 这是有效的, 因为两个 Sound 对象有相同的目标  
mySound.stop("doorknock");
```

## 参见

*Sound.start()*

---

### **\_soundbuftime 全局属性**

要预装载的流声音所需要的时间，用秒表示

#### 有效性

Flash 4 及其以后的版本

#### 概要

`_soundbuftime`

#### 访问

读 / 写

#### 描述

`_soundbuftime` 属性是一个整数，用来指定流声音在播放之前预装载所需要的时间（以秒表示）。默认值为 5 秒。

Flash 带流声音的同步影片播放要确保卡通人物的嘴和配音相匹配。在 `_soundbuftime` 秒的流声音下载之前，动画会保持停止，因此，在较低连接上设置较长的停顿会造成额外的启动次数。因为网络字节流会很慢，或者有暂时的中断，因此较短的 `_soundbuftime` 会让声音发生跳跃（也就是如果没有装载足够的声音数据）。理想的设置基于图形的复杂性、声音的质量设置以及用户的 Internet 连接带宽不同而在不同的影片之间有差异。默认设置（5 秒）通常会比较恰当，但是个别的情况可能需要用实验来证明什么是最好的设置。流缓冲时间可能会在播放的过程中发生变化，但它是一个全局属性，不可以对单个的声音进行设置。

## 示例

```
_soundbuftime = 10; // 音频缓冲 10 秒
```

## startDrag()全局函数

让影片或者影片剪辑跟随鼠标指示器

**有效性** Flash 4 及其以后的版本

**概要**

```
startDrag(target)
startDrag(target, lockCenter)
startDrag(target, lockCenter, left, top, right, bottom)
```

**参数**

*target* 一个串或一个引用，表示到要跟随鼠标指示器的影片或影片剪辑实例的路径（引用使用于串语境的时候会转换为路径）。

*lockCenter* 一个布尔值，表示 *target* 的记录点是应该在鼠标指示器的中心处 (*true*)，还是按照它的最初位置来拖动 (*false*)。

*left* 一个数字，指定在 *target* 的记录点左边可以被拖动的最小 x 坐标。

*top* 一个数字，指定在 *target* 的记录点上边可以被拖动的最小 y 坐标。

*right* 一个数字，指定在 *target* 的记录点右边可以被拖动的最大 x 坐标。

*bottom* 一个数字，指定在 *target* 的记录点下边可以被拖动的最大 y 坐标。

**描述**

*startDrag()* 函数让 *target* 在视觉上跟随鼠标指示器在播放器中移动（称为剪辑拖动）。拖动剪辑的移动可以被限制在一个方框内，它的坐标是作为 *startDrag()* 函数的参数给出的。方框坐标和 *target* 所在的画布有关。如果画布为主影片场景，那么 (0, 0) 就是场景的左上角。如果画布为一个影片剪辑，那么 (0, 0) 就是剪辑的画布记录点。注意，Flash 的坐标系统反转了笛卡儿的 Y 轴，y 值向屏幕的下方增加，向屏幕的上方减少。负的 y 值在原点上方（也就是在 X 轴上面）。

拖动可以在任何时间通过 *stopDrag()* 来停止。一次只能拖动一个影片剪辑或者影片，因此，将 *startDrag()* 函数设置到一个新的 *target* 上，会自动取消已经进行的任何拖动动作。并且，当一个影片或影片剪辑被拖动的时候，它所包含的所有影片剪辑也随之一起拖动。

**示例**

```
// 拖动ball，将它的移动限制在场景的左上角  
startDrag("ball", true, 0, 0, 225, 200);
```

**参见**

*MovieClip.startDrag()*, *stopDrag()*

---

**stop()全局函数**

将播放的影片停止在当前帧

**有效性** Flash 2 及其以后的版本

**概要** stop()

**描述**

*stop()*函数是一个简单然而基础的函数，可停止影片或者影片剪辑的播放。它是*MovieClip.stop()*方法的搭档，通常用来等待用户，比如，从图形菜单中进行选择。

**参见**

*MovieClip.stop()*, *play()*

---

**stopAllSounds()全局函数**

让影片静音

**有效性** Flash 3 及其以后的版本

**概要** stopAllSounds()

**描述**

*stopAllSounds()*函数将禁止当前播放的影片中的所有声音，不管其在影片剪辑中嵌套得有多深。这将应用于影片中的所有声音上，包括程序化生成的*Sound*对象。关于停止、开始以及设置声音音量的详细内容，请参见*Sound*类。

注意，*stopAllSounds()*只是临时起作用，任何在*stopAllSounds()*调用之后开始的声音都将照常播放。没有办法让影片永远地静音。

**参见**

*Sound.setVolume()*, *Sound.stop()*

---

**stopDrag()全局函数**

停止拖动

**有效性** Flash 4 及其以后的版本

**概要** stopDrag()

### 描述

*startDrag()* 函数让一个影片剪辑跟随场景上的鼠标指示器移动。*stopDrag()* 操作将停止跟随鼠标指示器的移动。因为一次只能拖动一个影片剪辑或影片，因此，*stopDrag()* 不使用 *target* 参数，它只简单地取消正在进行的拖动过程。

*startDrag()* 和 *stopDrag()* 一起使用可创建简单的 Flash 拖放界面，正如在线代码库的 Interface Widgets（界面部件）中所示的那样。

### 示例

下面的按钮代码将让一个影片剪辑在按钮被按下的时候拖动，在按钮被释放的时候放下：

```
on (press) {
    startDrag("", true);
}

on (release) {
    stopDrag();
}
```

### 参见

*MovieClip.stopDrag()*, *startDrag()*, *String.toLowerCase()*, 第四章

## String()全局函数

将某个值转换为串数据类型

**有效性** Flash 5

**概要** String(*value*)

### 参数

**value** 一个表达式，包含要转换为串的值。

### 返回

将 *value* 转换为原始串的结果。

### 描述

*String()* 函数将一个参数转换为原始串值，并且返回所转换的值。将不同数据类型转换为原始串的结果在表 3-2 中已经描述过了。通常不需要使用 *String()* 函数，ActionScript 会在必要的时候自动将值转换为串数据类型。

要确保不要将 `String()` 全局函数同 `String` 的类构造器混淆起来。前者将一个表达式转换为一个串，而后者是一个类，用来将原始串数据包装到对象中，以便应用属性和方法。

用法

注意，`String()`函数有的时候出现在并没有转换为 Flash 5 格式的 Flash 4 的 `.fla` 文件中。关于数据类型在 Flash 4 文件转换为 Flash 5 的时候应如何处理，请参见第三章。

参见

String 类, 第二章

String 类	
针对串原始类型的包装类	
有效性	Flash 5
构造器	<code>new String(value)</code>
参数	
<i>value</i>	要处理的一个表达式，如有必要，要转换为串，然后封装到一个 <i>String</i> 对象中。
属性	
<i>length</i>	串中的字符数。
类方法	
下面的方法要通过 <i>String</i> 类本身来调用，而不是通过 <i>String</i> 类的一个对象。	
<i>fromCharCode()</i>	从一个或者多个 Latin1/Shift-JIS 编码点产生一个串。
方法	
下面的对象方法可以通过 <i>String</i> 类的实例来调用：	
<i>charAt()</i>	在串中指定的位置获取一个字符。
<i>charCodeAt()</i>	获取串中指定字符的编码点。
<i>concat()</i>	将一个或多个项目合并为一个单独的串。
<i>indexOf()</i>	在串中寻找指定子串的第一次出现。
<i>lastIndexOf()</i>	在串中寻找指定子串的最后一次出现。

---

<i>slice()</i>	基于正的或负的字符位置，从串中提取一个子串。
<i>split()</i>	将串转换为数组。
<i>substr()</i>	基于开始点和长度，从串中提取一个子串。
<i>substring()</i>	基于正的字符位置从串中提取一个子串。
<i>toLowerCase()</i>	返回一个串的小写形式。
<i>toUpperCase()</i>	返回一个串的大写形式。

### 描述

*String* 类有下面几个用途：

- 允许我们访问串的 *length* 属性，执行和串相关的操作，比如 *indexOf()* 和 *slice()*。  
*String* 对象是当某个方法在一个原始串值上调用的时候，由解释程序自动创建的（最终会被删除）。
- 可以将任何类型的数据转换为串。
- 可以访问 *fromCharCode()* 类方法，以基于指定的 Latin1 或者 Shift-JIS 编码点而创建一个新的串。
- 可以创建一个 *String* 对象，该对象在一个未命名的内部属性中包含一个原始串值，但是，几乎没有必要这么做。

### 用法

在实践中，*String* 类构造器一般用来将其他数据类型转换为串。参见全局函数 *String()* 可以得到更详细的内容。

### 参见

第四章

---

## String.charAt()方法

从串中指定的位置获取一个字符

**有效性** Flash 5

**概要** *string.charAt(index)*

**参数**

*index* 要获取字符的整数位置，可以从 0(第一个字符) 到 *string.length*-1 (最后一个字符)。

**返回**

*string* 内位置 *index* 上的字符。

**描述**

*charAt()*方法确定串中指定位置 (*index*) 上的字符。

**示例**

```
trace("It is 10:34pm",charAt(1));      // 显示: 't' (第 2 个字符)
var country = 'Canada';
trace(country.charAt(0));                // 显示: 'C' (第一个字符)

// 这个函数将从串中删除所有的空格，并且返回结果
function stripSpaces(inString) {
    var outString = '';
    for (i = 0; i < inString.length; i++) {
        if (inString.charAt(i) != ' ') {
            outString += inString.charAt(i);
        }
    }
    return outString;
}
```

**参见**

*String.charCodeAt()*, *String.indexOf()*, *String.slice()*, 第四章

---

**String.charCodeAt()方法**

获取串中指定字符的编码点

**有效性** Flash 5

**概要** *string.charCodeAt(index)*

**参数**

*index* *String* 中某个字符的整数位置，可以从 0 (第一个字符) 到 *string.length-1* (最后一个字符)。

**返回**

一个整数，表示 *string* 中 *index* 位置上字符的 Latin1 或者 Shift-JIS 编码点，如附录二所示。

**示例**

```
var msg = "A is the first letter of the Latin alphabet.";
trace(msg.charCodeAt(0));      // 显示: 65 (对 "A" 字符的代码)
trace(msg.charCodeAt(1));      // 显示: 32 (对空格符的代码)
```

## 参见

*String.charAt()*, *String.fromCharCode()*, 附录二, 第四章

## String.concat()方法

将一个或多个值合并到单独的串中

**有效性** Flash 5

**概要** *string.concat(value1, value2, ...valuen)*

### 参数

*value1, ...valuen* 要转换为串（如果需要）并且要连接到 *string* 上的值。

### 返回

将 *string* 同 *value1, value2, ...valuen* 连接的结果。

### 描述

*concat()* 方法从一系列值创建一个串。它类似于在串上使用连接操作符 (+)，但是有的时候显得要清楚些，因为 + 操作符也可以用在数字上。关于 *concat()* 的详细说明，参见第四章。

### 用法

注意，*concat()* 不修改 *string*，它返回一个全新的串。

### 示例

```
var greeting = "Hello";
excitedGreeting = greeting.concat("!");
trace(greeting);           // 显示: "Hello"
trace(excitedGreeting);    // 显示: "Hello!"  
  
var x = 4;                  // 将 x 初始化为一个整数
trace(x + 5);               // 显示: 9
trace(x.concat(5));         // 因为 x 不是串，所以失败
trace(String(x).concat(5)); // 显示: "45"  
  
var x = "4";                // 将 x 初始化为一个串
trace(x.concat(5));         // 显示: "45"
trace(concat("foo", "fee")); // 因为 concat() 必须作为串的方法来调用
                           // 所以失败
```

## 参见

第五章, 第四章

---

**String.fromCharCode()类方法**      从一个或多个编码点生成一个串**有效性**      Flash 5**概要**      `String.fromCharCode(code_point1, code_point2, ...code_pointn)`**参数***code\_point1, ...code\_pointn*

一系列十进制整数，对应于 Latin1 或者 Shift-JIS 字符编码点，如附录二中所示。

**返回**

一个串，它将指定编码点所表示的字符连接起来。

**描述**

*fromCharCode()*类方法从第四章中所描述的字符编码点产生一个字符或一系列字符。

**示例**

```
// 制作一个复制的符号，后面跟着年份(2001)
copyNotice = String.fromCharCode(169) + "2001";
```

**参见**

*String.charCodeAt()*, 附录二, 第四章

---

---

**String.indexOf()方法**      在串中寻找一个子串的首次出现**有效性**      Flash 5**概要**      `string.indexOf(substring)``string.indexOf(substring, startIndex)`**参数***substring*      一个包含要搜索的字符的串。*startIndex*      一个可选的 *string* 中的整数位置，也就是开始搜索 *substring* 的地方。可以从 0 (第 1 个字符) 到 *string.length-1* (最后一个字符)。默认值为 0。**返回**

*substring* 在 *string* 中 (在 *startIndex* 之后) 首次出现的位置。如果 *substring* 没有找到，或在 *string* 中 *startIndex* 之后没有出现，就返回 -1。

## 描述

*indexOf()*方法用来搜索串中的字符，或者检查一个串是否包含了一个特定的子串。

## 示例

```
// 检查一个邮件地址中是否包含了@符号
var email = "derek@aol.com";
if (email.indexOf('@') == -1) {
    trace ('This isn t a valid email address');
}

// 检查一个邮件地址中是否有@符号、是否来自aol.com域
var email = "derek@aol.com";
var atPos = email.indexOf('@');
if (atPos != -1 && email.indexOf('aol.com') == atPos + 1) {
    gotoAndStop("AClassOfficer");
}
```

下面的代码显示了一个检查串中关键字的函数，在评估某个填空测验的时候可能会需要

```
// 用不区分大小写的比较来搜索 origStr 中出现的任何 searchStr 的函数
function search (origStr, searchStr) {
    var origStr = origStr.toLowerCase();
    var searchStr = searchStr.toLowerCase();
    return origStr.indexOf(searchStr) != -1;
}

var answer = 'einstein';
var guess = "Dr.Albert Einstein";
// 如果guess 包括 "einstein"，就增加score
if (search(guess, answer)) {
    score++;
}
```

## 参见

*String.charAt()*, *String.lastIndexOf()*, 第四章

---

## String.lastIndexOf()方法

在串中寻找一个子串的最后一次出现

**有效性** Flash 5

### 概要

*string.lastIndexOf(substring)*

*string.lastIndexOf(substring, startIndex)*

### 参数

*substring* 一个包含要搜索的字符的串。

**startIndex** 一个可选的 *string* 中的整数位置，也就是开始搜索 *substring* 的地方。搜索从 *startIndex* 开始向前进行。*startIndex* 可以从 0 (第一个字符) 到 *string.length-1* (最后一个字符)。默认值为 *string.length-1*。

#### 返回

*substring* 在 *string* 中 *startIndex* 之前最后一次出现的位置。如果 *substring* 在 *string* 中 *startIndex* 之前没有出现，就返回 -1。

#### 描述

*lastIndexOf()* 方法用来搜索串中某个子串的最后一次出现，或者检查一个串是否包含了一个特定的子串。

#### 示例

```
URL = "http://www.moock.org/webdesign/flash/fillthewindow.html";
// 寻找最后一个斜杠符号
lastSlash = URL.lastIndexOf('/');
// 从 URL 中提取文件名称
file = URL.substring(lastSlash + 1);
trace(file); // 显示: fillthewindow.html
```

#### 参见

*String.charAt()*, *String.indexOf()*, 第四章

---

## String.length 属性

串中的字符数

**有效性** Flash 5

**概要** *string.length*

**访问** 只读

#### 描述

*length* 属性返回 *string* 中的字符数。注意，空字符 (ASCII 0) 不标志串的结束，在许多语言中都是如此，但是它也不能算在串的 *length* 中。例如：

```
// 创建串 'A' + null + 'B'
var myString = String.fromCharCode(65, 0, 66);
trace(myString.length); // 显示: 2 (空字符被忽略)
```

#### 示例

```
var myString = "hello";
trace (myString.length); // 显示: 5
```

```
trace ('hello'.length);           // 显示: 5  
  
// 下面, 我们将数字 1000 转换为串, 以便检查长度  
var age = 1000;  
// 如果数字有错, 显示一个错误信息  
if (String(age).length != 2) {  
    trace ("Please enter a two-digit number");  
}
```

## 参见

*Array.length()*, 第四章

---

## String.slice()方法

基于正的或者负的字符位置, 从串中提取一个子串

**有效性** Flash 5

**概要** *string.slice(startIndex, endIndex)*

### 参数

*startIndex* 要从 *string* 中提取的第一个字符的整数位置。如果 *startIndex* 为负数, 那么这个位置就从串的末尾开始计量, -1 就是最后一个字符, -2 是倒数第 2 个字符, 依次类推 (也就是说, 负的 *startIndex* 用来指定 *string.length+startIndex* 处的字符)。

*endIndex* 要从 *string* 中提取的最后一个字符之后字符的整数位置。如果 *endIndex* 为负数, 那么这个位置就从串的末尾开始计量, -1 为最后一个字符, -2 为倒数第 2 个字符, 依次类推 (也就是说, 负的 *endIndex* 指定 *string.length+endIndex* 处的字符)。如果没有提供, 默认值为 *string.length*。

### 返回

*string* 的一个子串, 从 *startIndex* 开始, 以 *endIndex-1* 为结束, *startIndex* 和 *endIndex* 都是从 0 开始的。

### 描述

*slice()* 方法是从串中提取子串的三个方法之一 (其他两个是 *substring()* 和 *substr()*)。*slice()* 方法提供了以负数表示的开始和结束索引值, 以作为一种选择, 这让我们可以提取从串结尾开始计量的子串。

### 用法

注意, *slice()* 不修改 *string*, 它返回的是一个全新的串。

## 示例

```
var fullName = "steven Sid Mumby";
middleName = fullName.slice(7, 10); // 将 "Sid" 赋给 middleName
middleName = fullName.slice(-9, -6); // 将 "Sid" 也赋给 middleName
```

## 参见

*String.substr()*, *String.substring()*, 第四章

## String.split()方法

将一个串转换为一系列的数组元素

**有效性** Flash 5

**概要** `string.split(delimiter)`

### 参数

*delimiter* 用来分割 *string*, 以形成新数组元素的符号或者符号序列。

### 返回

一个数组, 其元素包含的子串是按照 *delimiter* 将 *string* 分割而形成的。

### 描述

*split()* 方法将串分割为子串, 将这些子串赋给一个数组的元素, 并且返回该数组。如果 *delimiter* 连续出现, 中间没有其他字符, 就会形成空元素。例如, 下面的代码:

```
owners = "terry,doug,,,jon";
ownersArray = owners.split(',');
```

会将下面的元素赋给 *ownersArray* (元素 2 和 3 包含的是 *undefined*):

```
0: terry
1: doug
2:
3:
4: jon
```

*split()* 方法通常用来将从 CGI 脚本或者文本文件中所获得的串转换为一个数组, 以便以后的操作。它在解析 HTML 文本域 *<A>* 标签的 *asfunction* 调用中的参数时也很有用, 因为这种调用只能将一个串参数传递给函数。参见第十八章可以得到示例代码。普通的分隔符号包括逗号和 Tab 符号。

## 示例

假设在文本文件 *names.txt* 中存储了一个名称列表。每一个名称都用 Tab 字符与其他名称分隔开, 就像用空格来分隔一样:

```
owners=terry doug jor
```

在影片中的第 1 帧，我们将 *names.txt* 文件装载到影片中：

```
flis.loadVariables("names.txt");
```

在确保 *names.txt* 文件充分载入之后（参见第十章），我们就可以将载入的 *owners* 变量分割为一个数组：

```
splitString = String.fromCharCode(9);           // 将 tab 字符赋给 splitString  
ownersArray = owners.split(splitString);  
trace(ownersArray[1]);                          // 显示 "doug"
```

注意，*split()*可能会花很长时间来处理很长的文本。如果要注重效率，可以将数据分为容易管理的部分，或者考虑使用 XML。参见 XML 类。

## 故障

在 Flash 5 中，使用空串作为分隔符会将所有的 *string* 添加到生成数组的第一个元素中。按照 ECMA-262，一个空串分隔符会让 *string* 在每一个字符上被打断。同样，多字符的分隔符不能被 Flash 5 识别，它会让所有的串都赋给返回数组的第一个元素。

## 参见

*Array.join()*, 第四章

---

## String.substr()方法

基于开始点和长度从一个串中提取子串

**有效性** Flash 5

**概要** *string.substr(startIndex, length)*

### 参数

*startIndex* 要从 *string* 中提取的第一个字符的整数位置。如果 *startIndex* 为负数，那么这个位置就从串的末尾开始计量，-1就是最后一个字符，-2是倒数第 2 个字符，依次类推（也就是说，负的 *startIndex* 用来指定 *string.length+startIndex* 处的字符）。

*length* 要从 *string* 中提取的字符数，从 *startIndex* 开始，包括 *startIndex* 在内。如果没有指定，就是从 *startIndex* 到 *string* 末尾的所有字符都被提取。

### 返回

*string* 的一个子串，从 *startIndex* 开始，包括 *length* 个字符。如果没有提供 *length*，就是从 *startIndex* 到 *string* 末尾的所有字符。

### 描述

*substr()* 方法是用来从串中提取子串的三个方法之一（其他两个是 *slice()* 和 *substring()*）。*substr()* 方法基于 *length* 所指定的字符数来提取子串，而不是基于两个字符索引。

### 用法

注意，*substr()* 不修改 *string*，它返回的是一个全新的串。

### 示例

```
var fullName = "steven S.d Mumby";  
  
middleName = fullName.substr(7, 3); // 将 'Sid' 赋给 middleName  
firstName = fullName.substr(0, 6); // 将 "Steven" 赋给 firstName  
lastName = fullName.substr(11); // 将 "Mumby" 赋给 lastName  
  
// 注意负的开始索引……  
middleName = fullName.substr(-9, 3); // 将 "Sid" 赋给 middleName  
firstName = fullName.substr(-16, 6); // 将 "Steven" 赋给 firstName  
lastName = fullName.substr(-5); // 将 "Mumby" 赋给 lastName
```

### 参见

*String.slice()*, *String.substring()*, 第四章

---

## String.substring()方法

基于正的符号位置从一个串中提取子串

**有效性** Flash 5

### 概要

*string.substring(startIndex, endIndex)*

### 参数

*startIndex* 要从 *string* 中提取的第一个字符的正整数位置。如果为负，就取为 0。

*endIndex* 要从 *string* 中提取的最后一个字符之后字符的正整数位置。如果没有提供，默认值为 *string.length*。如果为负，就取为 0。

### 返回

*string* 的一个子串，从 *startIndex* 开始，*endIndex-1* 为结束。*startIndex* 和 *endIndex* 都是和 0 相对的。

## 描述

*substring()*方法是用来从串中提取子串的三个方法之一（其他两个是*slice()*和*substr()*）。*substring()*函数基本等同于*slice()*，除了它不允许负的*startIndex*和*endIndex*值之外，而且如果*endIndex*小于*startIndex*，它会自动调换两个索引。

## 用法

注意，*substring()*不修改*string*，它返回的是一个全新的串。

## 示例

```
// 从串中提取名字  
var fullName = "steven Sid Mumby";  
middleName = fullName.substring(7, 10); // 将 "Sid" 赋给 middleName  
middleName = fullName.substring(10, 7); // 将 "Sid" 赋给 middleName  
// (索引自动转换)  
firstName = fullName.substring(0, 6); // 将 "Steven" 赋给 firstName  
lastName = fullName.substring(11); // 将 "Mumby" 赋给 lastName
```

下面的例子是一个有用的函数，它在串中搜索所有某个子串出现的地方，并进行替换：

```
// 一个搜索替换函数  
function replace(origStr, searchStr, replaceStr) {  
    var tempStr = "";  
    var startIndex = 0;  
    if (searchStr == "") {  
        return origStr;  
    }  
  
    if (origStr.indexOf(searchStr) != -1) {  
        while ((searchIndex = origStr.indexOf(searchStr, startIndex)) != -1) {  
            tempStr += origStr.substring(startIndex, searchIndex);  
            tempStr += replaceStr;  
            startIndex = searchIndex+searchStr.length;  
        }  
        return tempStr + origStr.substring(startIndex);  
    } else {  
        return origStr;  
    }  
}  
  
msg = 'three times three is four';  
trace(replace(msg, "three", "two")); // 显示: "two times two is four"
```

## 参见

*String.slice()*, *String.substr()*, 第四章

---

## String.toLowerCase()方法

生成一个串的小写版本

**有效性** Flash 5

**概要** *string.toLowerCase()*

**返回**

*string* 的小写形式，是一个新的串。没有小写形式的字符不改变。

**描述**

*toLowerCase()* 方法创建 *string* 的一个新的小写版本，它可以用来进行格式化，或者便于进行不区分大小写的字符比较。*toLowerCase()* 方法只转换 A-Z 之间的字符（它不转换发音符号，比如重音和元音）。

**用法**

注意，*toLowerCase()* 不修改 *string*，它返回的是一个全新的串。

**示例**

```
// 将msg 设置为 "this sentence has mixed caps!"  
msg = "This Sentence Has Mixed Caps!".toLowerCase();  
  
// 对两个串进行不区分大小写的比较  
function caseInsensitiveCompare (stringA, stringB) {  
    return (stringA.toLowerCase() == stringB.toLowerCase());  
}  
  
trace(caseInsensitiveCompare("Colin", "colin")); // 显示: true
```

**参见**

*String.toUpperCase()*, 第四章

---

## String.toUpperCase()方法

生成一个串的大写版本

**有效性** Flash 5

**概要** *string.toUpperCase()*

**返回**

*string* 的大写形式（也叫做全大写），是一个新的串。没有大写形式的字符不改变。

**描述**

*toUpperCase()* 方法创建 *string* 的一个新的大写版本，它可以用来进行格式化，或者

便于进行不区分大小写的字符比较。*toUpperCase()*方法只转换a-z之间的字符（它不转换发音符号，比如重音和元音）。

## 用法

注意，*toUpperCase()*不修改*string*，它返回的是一个全新的串。

## 示例

```
'listen to me'.toUpperCase(); // 产生串 'LISTEN TO ME'  
var msg1 = 'Your final Score:234';  
var msg2 = msg1.toUpperCase(); // 将msg2设置为 'YOUR FINAL SCORE:234'
```

## 参见

*String.toLowerCase()*, 第四章

## targetPath()全局函数

到影片或影片剪辑的绝对路径

**有效性** Flash 5

**概要** targetPath(*movieClip*)

### 参数

*movieClip* 到影片剪辑对象的引用。

### 返回

一个串，表示到*movieClip*的绝对路径，使用点符号（例如，“\_level0.myMovie”）。

## 描述

*targetPath()*函数返回影片剪辑的引用串形式，它描述到该剪辑的绝对路径（等同于*MovieClip.valueOf()*的返回值）。*targetPath()*函数有的时候用来构成区分位置的代码，这些代码在影片剪辑上进行操作，和剪辑所在的时间线相关。

## 故障

注意，在Flash 5 ActionScript词典中所给出的关于*targetPath()*的示例不表示*targetPath()*的恰当用法，和例子正相反，它不是*tellTarget()*的等同物。

## 示例

如果*square*是包含在*shapes*中的影片剪辑，而*shapes*又在层级0的主时间线上，那么在*shpaes*剪辑内，语句；

```
targetPath(square);
```

会返回

```
'_level0.shapes.square'
```

## 参见

`MovieClip._target`, `MovieClip.valueOf()`, 第十三章

## tellTarget()全局函数

执行远程影片剪辑作用域内的语句

**有效性** Flash 3 和 Flash 4，在 Flash 5 中，因为面向对象的语法或者 `with` 语句而不被支持

### 概要

```
tellTarget (target) {  
    statements  
}
```

### 参数

*target* 一个串或引用，表示到影片或影片剪辑实例的路径（引用使用于串语境的时候会转换为路径）。

*statements* 要在 *target* 的作用域内执行的语句。

### 描述

在 Flash 3 和 Flash 4 中，`tellTarget()` 是在两个影片剪辑之间通信的主要途径（也就是从一个剪辑控制另外一个剪辑）。它用来在远程影片剪辑上调用诸如 `play()`, `stop()` 和 `gotoAndStop()` 这样的函数。在 Flash 4 中，当变量添加到 ActionScript 中以后，可以用 `tellTarget()` 来获得和设置远程剪辑变量值。在 Flash 5 中，这些动作可以用点操作符 `.` 和数组访问操作符 `[]` 来更好地完成。`tellTarget()` 函数的另外一个交换选择是 `with` 语句，在第六章中描述过了。

### 用法

`tellTarget()` 函数可能作为语句来描述会更好一些，因为它要求一个子语句块。但这只是理论上的说法，`tellTarget()` 已经被淘汰了。

### 示例

```
tellTarget ('ball') {  
    gotoAndStop('redStripes');  
    _x += 300;  
}
```

**参见**

第六章、第十三章

**toggleHighQuality()全局函数**

改变播放器的渲染质量

**有效性** Flash 2 及其以后的版本，在 Flash 5 中因为 `_quality` 而被淘汰

**概要** `toggleHighQuality()`

**描述**

在渲染的高质量和低质量之间进行转换。设置为 High 的时候，Flash 播放器的渲染线条是保真（平滑）的边缘。设置为 Low 的时候，Flash 播放器的渲染线条是折叠（锯齿的）边缘。`toggleHighQuality()` 函数没有参数，它只在两种可能的设置之间进行转换——High 和 Low。这存在一定的问题，因为它不能将渲染质量精确地设置为一个已知的设置，也不允许这两种设置之外的其他设置。

在 Flash 5 中，`toggleHighQuality()` 由于全局属性 `_quality` 的使用而被淘汰，后者支持 Low, Medium, High 和 Best 四种渲染设置。

**参见**

`_highquality`, `_quality`

**trace()全局函数**

在输出窗口中显示一个值

**有效性** Flash 4 及其以后的版本

**概要** `trace(value)`

**参数**

**value** 要处理的表达式，然后显示在输出窗口中。如果 `value` 的值不是一个串，就会在发送给输出窗口之前转换为一个串，规则如表 3-2 所述。

**描述**

`trace()` 函数是一种调试工具，只用在 Flash 制作环境的测试影片模式中。虽然看起来很平常，但却是 ActionScript 编程的一个基础成分，它可以在影片播放的任何时间检查变量或表达式的值。

## 用法

不幸的是, *trace()*非常慢。在File → Publish Settings → Flash中通过Omit Trace Actions (忽略跟踪动作) 选项可以取消它。

## 示例

```
trace(firstName);           // 输出 firstName 的值  
trace(myClip);            // 输出 myClip 的路径  
trace(myClip._x);          // 输出 myClip 的 x 坐标  
trace("hello" + " there"); // 计算并输出表达式
```

## 参见

第十九章, 调试

---

## unescape()全局函数

对一个编码串进行解码

**有效性** Flash 5

**概要** `unescape(stringExpression)`

**参数**

*stringExpression* 一个串(或者能产生串的表达式), 先前通过*escape()*对其进行编码。

**返回**

一个新的串, 是 *string*的解码版本。

**描述**

*unescape()*函数基于 *string*返回一个新的串。新的串包含 *string*中带%符号的两位数十六进制转义序列的Latin1字符等价形式。*escape()*和*unescape()*函数用来对串进行编码和解码, 以便在网络上安全地传输。但是, 几乎不需要手动使用*unescape()*, 因为Flash会在通过*loadVariables()*输入URL编码文本的时候自动进行转换。

## 示例

```
var msg = "hello!";  
// 将msgCoded 设置为 "hello%21"  
msgCoded = escape(msg);  
// 将msgDecoded 设置为 "hello!"  
var msgDecoded = unescape(msgCoded);
```

## 参见

*escape()*, 附录二

---

**unloadMovie()全局函数** 从播放器中删除一个影片或影片剪辑

**有效性** Flash 4 及其以后的版本 (Flash 5 的 *unloadMovie()* 函数对应于 Flash 4 的使用目标路径的 *Unload Movie*)

**概要** `unloadMovie(target)`

**参数**

*target* 一个串或者引用，表示到要从播放器中删除的影片剪辑或文档层级的路径（引用使用于串语境的时候会转换为路径）。

**描述**

*unloadMovie()* 函数通常用于从播放器的文档层级中删除影片。例如，如果一个影片被装载到播放器的层级 1，那么可以用下面的方式从播放器中删除这个影片：

```
unloadMovie('_level1');
```

*unloadMovie()* 函数也可以用在影片剪辑实例上，在这种情况下，它删除实例的内容，但不删除实例本身。实例留在场景上，成为一个空壳，我们在以后可以将影片装载到这个空壳中来。因此，一个剪辑可以作为动态内容的容器而存在，它的内容会随着 *loadMovie()* 和 *unloadMovie()* 的执行而变化。

**参见**

*loadMovie()*, *MovieClip.unloadMovie()*, *unloadMovieNum()*, 第十三章

---

**unloadMovieNum()全局函数** 从文档层级中删除影片

**有效性** Flash 3 及其以后的版本 (Flash 5 的 *unloadMovieNum()* 函数对应于 Flash 3 中的 *unload Movie*，后者只对编号的层级有效)

**概要** `unloadMovieNum(level)`

**参数**

*level* 一个非负整数或者能产生非负整数的表达式，表示要卸载的层级。

**描述**

*unloadMovieNum()* 函数基本等同于 *unloadMovie()*，除了前者要求装载操作的目标层级要作为数字来指定，而不是作为串。这意味着 *unloadMovieNum()* 只能删除文档层级上的影片，而不能删除影片剪辑。它通常用在我们希望动态地设置卸载影片层级的时候。比如：

```
var x = 3;  
unloadMovieNum(x);
```

这也可以用串和正规 *unloadMovie()* 函数的连接来实现：

```
unloadMovie('level' + x);
```

## 参见

*loadMovieNum()*, *Movieclip.unloadMovie()*, *unloadMovie()*, 第十三章

---

## updateAfterEvent()全局函数

在帧之间渲染场景内容

**有效性** Flash 5

**概要** updateAfterEvent()

### 描述

用户输入剪辑事件处理器 (*mouseMove*, *mouseDown*, *mouseUp*, *keyDown* 和 *keyUp*) 通常发生在 Flash 播放器里帧的渲染之间。要强迫屏幕对发生在用户输入剪辑事件处理器期间的视觉改变作出响应，可以在任何这类处理器内调用 *updateAfterEvent()*。但要注意，*updateAfterEvent()* 不是一个武断的屏幕更新工具，它只在用户输入剪辑事件中有效。在 *onClipEvent()* 处理器之外它是无效的。

### 示例

下面的脚本添加到影片剪辑上，会让剪辑跟随鼠标指示器而移动，并在每次指示器移动的时候刷新屏幕。因为在每次指示器移动的时候刷新屏幕，因此跟随鼠标指示器的剪辑就显得相当平滑：

```
onClipEvent (mouseMove) {  
    _x = _root._xmouse;  
    _y = _root._ymouse;  
    updateAfterEvent();  
}
```

### 故障

注意，Flash 5 ActionScript 词典中将 *updateAfterEvent()* 函数描述为会将剪辑事件作为参数来接收，这是不正确的，它不接收任何参数。

## 参见

第十章

**\$version “全局”属性**

Flash 播放器的版本

**有效性** Flash 4 修订版 11 及其以后的版本，在 Flash 5 中因为 *get Version* 的使用而不被支持了

**概要** `root.$version`

**访问** 只读

**描述**

`$version` 属性包含和全局函数 *getVersion()* 的返回值相同的串（操作系统，后面是播放器的版本信息）。`$version` 属性在 Flash 4 播放器的生命周期中途引入，后来被 *getVersion()* 替代。它在理论上不是一个全局属性，而是主影片时间线的属性，从其他影片剪辑时间线上，可按照 `_root.$version` 的形式来访问它。

**参见**

*getVersion()*

**XML 类**

对 XML 结构数据基于 DOM 的支持

**有效性** Flash 5

**构造器** `new XML()`

`new XML(source)`

**参数**

*source* 一个可选的串，包含要分解到 XML 对象层次中的规范形式 XML（或者 HTML）的数据。

**属性**

*attributes* 一个对象，其属性存储了元素属性。

*childNodes* 一个数组，包含指向子孙节点的引用。

*contentType* 要传输给服务器的 MIME 内容类型。

*docTypeDecl* 文档的 DOCTYPE 标签。

*firstChild* 指向节点第一个子孙的引用。

*ignoreWhite* 确定是否要在 XML 解析期间忽略空白节点。

*lastChild* 指向节点最后一个后代的引用。

<i>loaded</i>	<i>load()</i> 或 <i>sendAndLoad()</i> 操作的状态。
<i>nextSibling</i>	指向对象层次当前层级的节点之后的节点的引用。
<i>nodeName</i>	当前节点的名称。
<i>nodeType</i>	当前节点的类型。
<i>nodeValue</i>	当前节点的值。
<i>parentNode</i>	指向节点直接祖先的引用。
<i>previousSibling</i>	指向对象层次中当前层级的节点之前的节点的引用。
<i>status</i>	描述将 XML 源分解入对象层次中的错误代码。
<i>xmlDecl</i>	文档的 XML 声明标签。

## 方法

<i>appendChild()</i>	为节点添加一个新的子节点。
<i>cloneNode()</i>	创建节点的复本。
<i>createElement()</i>	创建一个新的元素节点。
<i>createTextNode()</i>	创建一个新的文本节点。
<i>hasChildNodes()</i>	检查一个节点是否有后代。
<i>insertBefore()</i>	在节点之前添加同胞节点。
<i>load()</i>	从外部文档导入 XML 源代码。
<i>parseXML()</i>	解析 XML 源代码的串。
<i>removeNode()</i>	从对象层次中删除节点。
<i>send()</i>	将 XML 源代码发送到脚本。
<i>sendAndLoad()</i>	将 XML 源代码发送到脚本，并接收返回的 XML 源代码。
<i>toString()</i>	将 XML 对象转换为串。

## 事件处理器

<i>onData()</i>	在外部 XML 源文件装载结束之后执行的处理器。
<i>onLoad()</i>	在外部 XML 数据被分解到对象层次中的时候执行的处理器。

## 描述

我们用 XML 类的对象来操作面向对象方式下 XML (或者 HTML) 文档的内容，以及将 XML 格式的数据发送到 Flash，或者接收来自 Flash 的 XML 格式的数据。使用 XML

对象的方法和属性，可以建立一个 XML 结构的文档（或者读已存在的某个文档），以及有效地访问、修改或者删除文档中的信息。

XML 文档的源代码主要由一系列的元素和属性来构成。例如，在下面的 XML 片段中，元素 BOOK, TITLE, AUTHOR 和 PUBLISHER 有相同的格式，也就是 HTML 标签，我们会看到，AUTHOR 元素支持一个属性 SALUTATION：

```
<BOOK>
  <TITLE>ActionScript: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr.">Colin Moock</AUTHOR>
  <PUBLISHER>O'Reilly</PUBLISHER>
</BOOK>
```

从面向对象的角度来看，XML 文档的内容可以作为对象层次来看待，在这个层次中，每一个元素和文本块都变成了类似流程图结构中的对象节点。图 R-1 显示了简单的 XML <BOOK> 格式，在概念上是用 XML 对象层次来表示的。

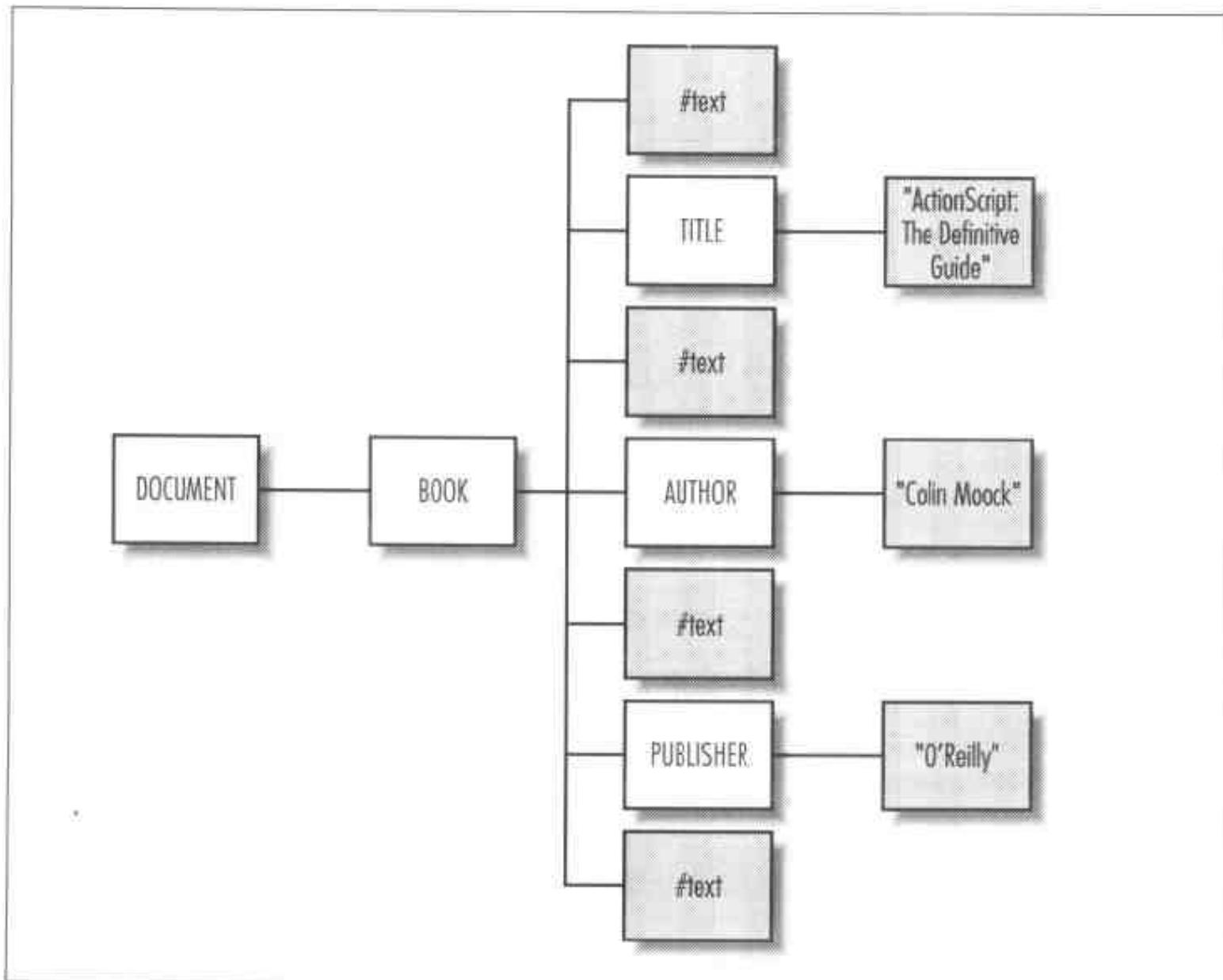


图 R-1 一个简单的 XML 节点层次

我们从左到右来考虑一下这个简单 XML 对象层次的结构和语义。我们从主 XML 对象开始，也就是图 R-1 中的 DOCUMENT，它是由 XML 构造器自动创建的，并且作为我们的 XML 对象层次中的一个容器。

在层次中右移一排，就到了 BOOK，它是 XML 源代码标签中的第一个元素，在本例中，也是 DOCUMENT 下的第一个对象节点。BOOK 节点是 XML 数据结构的根——每一个正规形式的 XML 文档必须有一个总领的根元素，比如 BOOK，它包含所有的其他元素。XML 对象层次的分支要添加到树中，可以通过解析 XML 源代码或者在层次对象上调用节点添加方法来实现。

当一个节点包含在另外一个节点当中的时候，被包含的节点就称为包含节点的子节点，而包含节点称为父节点。在我们的例子中，BOOK 就是 DOCUMENT 的第一个子节点，而 DOCUMENT 是 BOOK 的父节点。

当我们在图 R-1 中右移的时候，我们看到 BOOK 有 7 个子节点，包括 #text 节点，这个节点在最初的 XML 文档中看起来并不存在。在 XML 源代码中，元素之间每一个出现的空格都被当作 XML 对象层次中的一个对象。如果我们认真查看，会在前边的 XML 标签中的 BOOK 和 TITLE 之间发现空格——回车和 Tab 字符。这个空格在图 R-1 中用 #text 节点来表示，在 TITLE、AUTHOR 和 PUBLISHER 节点的后面也有类似的空格节点。

BOOK 的子节点互相之间是兄弟的关系（也就是说，它们存在于层次中的同一层）。例如，我们说 AUTHOR 的下一个兄弟是 #text，AUTHOR 的上一个兄弟是 #text。当我们在层次中从一个兄弟转移到另外一个兄弟的时候，可以看到 #text 节点有些碍事。可以按照下面的方法来处理这些空格节点：

- 手动地将它们从对象层次中去除（参见后面例子中的空格去除代码）。
- 先删除，然后在代码中跳过它们（参见 nextSibling 和 previousSibling 属性的删除节点方法）。
- 简单地从 XML 源代码中删除格式化空格，确保空格节点不首先出现。
- 将 XML 对象的 ignoreWhite 属性设置为 true，然后再解析我们的 XML 源代码（对于 Flash 5 播放器的修订版 41 有效）。

最后，我们来到了层次中的最后一排，在这里，TITLE、AUTHOR 和 PUBLISHER 节点都有一个子节点。每个子节点都是文本节点，对应包含在元素 TITLE、AUTHOR 和 PUBLISHER 中的文本里。注意，包含在 XML 源代码元素中的文本存在于对对象结构内该元素的子节点中。要访问被元素包含的文本，必须总是用 firstChild。

nodeValue 或者 childNodes[0].nodeValue 来指向元素的子节点，关于这一点，我们马上就会讨论到。

但是，元素的属性又如何呢？它们应该出现在 XML 对象层次的什么地方？可以设想 AUTHOR 的 SALUTATION 属性被描述为子节点 SALUTATION。但是在实践中，一个属性不能被当作定义它的元素的子节点，而应该当作该元素的一个属性。要学习如何访问这个属性，参见 XML.attributes 条目。

我们来看看如何将 XML 文档建立为一个节点对象的层次。要创建一个新的、空白的 XML 对象，我们使用 XML() 构造器：

```
myDocument = new XML();
```

然后，可以将节点添加到空的 XML 对象上，通过在对象上调用诸如 *appendChild()*, *parseXML()* 和 *load()* 这样的方法就可以实现。另外，我们也可以从脚本中现有的 XML 源代码创建一个 XML 对象，通过调用 XML 构造器，并使用 *source* 参数就可以实现：

```
myDocument = new XML(source);
```

例如：

```
myDocument = new XML('<P>hello world!</P>');
```

当我们在 XML() 构造器上应用 *source* 参数的时候，*source* 就被解析转换为新的对象层次，然后存储在构造器所返回的对象中。（在本例中，节点 P 被标注为 myDocument 的第一个子节点，而 nodeValue 为“hello world!” 的文本节点被标注为 P 的第一个子节点。）

一旦 XML 层次创建好，并存储在对象中，就可以用 XML 类的方法和属性来访问层次中的信息。例如，假设我们要获取 myDocument 中的文本“hello world!”。在面向对象形式中，假定我们可以将 P 的文本当作 myDocument 的属性来访问，如下所示：myDocument.P。实际上，这是无效的，我们不用名称来引用节点，而应用 XML 类的内置属性，比如 *firstChild* 和 *childNodes*，来访问节点。例如，要访问 P 节点，可以用：

```
myDocument.firstChild / 访问 P  
myDocument.childNodes[0] / 也可以访问 P
```

因为 *firstChild* 返回指向层次中指定节点的第一个子节点引用，myDocument.*firstChild* 就返回节点 P 的引用。但是，我们想让文本“hello world!” 包含在 P 中，而不是节点 P 本身。正如我们在前面所学到的，元素节点的文本作为节点的子节点来存储。因此，我们可以这样引用文本节点（也就是 P 的第一个后代）：

```
myDocument.firstChild.firstChild // 访问 P 下的文本节点
```

要获得节点的值，我们使用nodeValue 属性。例如，可以用下面的语句在输出窗口中显示值 “hello world!”：

```
trace(myDocument.firstChild.firstChild.nodeValue);
```

或者，可以重新设置 P 下文本节点的值，使用下面的语句：

```
myDocument.firstChild.firstChild.nodeValue = "goodbye cruel world";
```

要完全删除 P 节点，添加新节点，或将文本 “hello world!” 移到另外的节点，我们调用 XML 类中的恰当方法。例如：

```
// 删除 P
myDocument.firstChild.removeNode();

// 建立一个新的元素 P
newElement = myDocument.createElement("P");

// 将新的元素添加到文档
myDocument.appendChild(newElement);

// 建立一个新的文本节点，添加到 P
newText = myDocument.createTextNode("XML is fun");

// 将新的文本节点添加到 P
myDocument.firstChild.appendChild(newText);
```

正如你所看到的，在对象层次中处理 XML 结构数据要使用间接的方式。我们建立、破坏和处理数据的时候要在对象上调用方法，访问对象的属性。要学习处理 XML 数据的不同工具，可以参照后面列出的 XML 类的属性和方法。

ActionScript 用万维网联盟（W3C）发布的文档对象模板（DOM）标准来处理 XML 数据。关于 DOM 如何将 XML 结构数据表示为对象层次的详细内容，参见：

<http://www.w3.org/DOM>

关于核心 DOM 的语言独立规范的详细内容，参见：

<http://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>

(要特别注意 1.2 Interface Node (基础界面) 下的 *Fundamental Interfaces* (界面节点))。关于 DOM 如何在 ECMA-262 中实现的详细内容，参见：

<http://www.w3.org/TR/REC-DOM-Level-1/ecma-script-language-binding.html>

## 示例

我们已经知道，在 XML 源代码中，任何两个元素之间的空格在对应的 XML 对象层次中都用一个文本节点来表示。在 Flash 5 播放器的修订版 41 之前，不需要的空格节点必须手动地从 XML 对象层次中删除。去除特定种类的节点是 XML 处理中的一个基本任务，是树遍历（周游层次中的每一个节点）中的一个好例子。我们来看看从文档中去除空格节点的不同技术。

在第一个例子中，可以用标准的 FIFO（先进先出）堆栈在处理中将树中的所有节点添加到数组中去。*stripWhitespaceTraverse()* 函数用它所接收的参数 *theNode* 生成一个由节点元素构成的数组。然后，它开始循环，在循环中删除数组中的第一个节点，处理该节点，并且将它的子节点（如果有）添加到数组中。当数组再也没有其他元素的时候，*theNode* 的所有子孙就都被处理完了。在处理过程中，没有子节点的任何节点都被认为是潜在的空格（因为文本节点都没有子节点）。这些节点中的每一个都要经过检查，以确定是否：

- 为文本节点（由它的 *nodeType* 属性来确定）。
- 包含 ASCII 32 以上的字符，这不算是空格。

任何只包含 ASCII 32 以下的字符（即只有空格）的文本节点被删除：

```
// 用 FIFO 堆栈去除空格
// 通过树遍历而删除从 theNode 以下的任何空格节点
function stripWhitespaceTraverse (theNode) {
    // 创建一个要处理的节点列表
    var nodeList = new Array();
    // 用传递给函数的节点生成列表
    nodeList[0] = theNode;

    // 处理后代节点，直到全部处理完
    while (nodeList.length > 0) {
        // 从列表中删除第一个节点，然后将它赋给 currentNode
        currentNode = nodeList.shift();

        // 如果这个节点有子节点……
        if (currentNode.childNodes.length > 0) {
            // ... 将这个节点的子节点添加到要处理的节点列表
            nodeList = nodeList.concat(currentNode.childNodes);
        } else {
            // ... 否则，这个节点就是分支的结尾，于是检查它是否为一个文本节点
            // 如果是，就检查它是否只包含空的空格
            // nodeType 3 表示一个文本节点
            if (currentNode.nodeType == 3) {
                var i = 0;
                var emptyNode = true;
                for (i = 0; i < currentNode.nodeValue.length; i++) {
                    if (currentNode.nodeValue[i] != " ") {
                        emptyNode = false;
                    }
                }
                if (emptyNode) {
                    // 将这个文本节点从列表中删除
                    nodeList.splice(nodeList.indexOf(currentNode), 1);
                }
            }
        }
    }
}
```

```
// 有效的字符在 32 之上（空格, tab, 换行等都在下面）。
if (currentNode.nodeValue.charCodeAt(i) > 32) {
    emptyNode = false;
    break;
}
}

// 如果没有发现有用的字符，就删除节点
if (emptyNode) {
    currentNode.removeNode();
}
}
}
```

前面例子中所显示的技术在传统上非常有效。但是，在 Flash 5 播放器中，*Array.concat()*方法执行得非常缓慢。因此，用下面例子中给出的技术来去除空格更快一些。要认真学习注释内容：

```
// 用函数递归来去除空格
// 在树的每层中遍历两遍，以从 XML 文档中去除空白节点
function stripWhitespaceDoublePass(theNode) {
    // 循环遍历 theNode 的所有子孙
    for (var i = 0; i < theNode.childNodes.length; i++) {
        // 如果当前的节点为文本节点……
        if (theNode.childNodes[i].nodeType == 3) {

            // ... 检查节点中有用的字符
            var j = 0;
            var emptyNode = true;
            for (j = 0; j < theNode.childNodes[i].nodeValue.length; j++) {
                // 有用的字符在 32 之上（空格、tab、换行等都在其下）
                if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
                    emptyNode = false;
                    break;
                }
            }

            // 如果没发现有用的字符，就删除该节点
            if (emptyNode) {
                theNode.childNodes[i].removeNode();
            }
        }
    }

    // 既然所有的空格节点都从 theNode 中删除了，就可以在剩余
    // 的子孙上递归调用 stripWhitespaceDoublePass() 函数
    for (var k = 0; k < theNode.childNodes.length; k++) {
        stripWhitespaceDoublePass(theNode.childNodes[k]);
    }
}
```

## 参见

*XMLnode* 类, *XMLSocket* 类, 第十八章

### XML.appendChild()方法

为节点添加一个新的子节点,  
或者删除一个现有的节点

**有效性** Flash 5

**概要** *theNode.appendChild(childNode)*

#### 参数

*childNode* 现有的一个 XML 节点对象。

#### 描述

*appendChild()*方法将特定的 *childNode*添加到 *theNode*作为 *theNode*的最后的一个子节点。可以用 *appendChild()*来将新的节点添加给一个现有的节点，可以在文档内移动一个节点，或者在文档之间移动一个节点。在每一种情况下，*childNode*都必须是一个指向现有节点对象的引用。

要将一个新的节点添加到现有节点，必须首先用 *createElement()*, *createTextNode()* 或 *cloneNode()*，或者将XML源代码分解到一个XML对象中以创建一个新的子节点。例如，在下面的代码里，我们创建一个新的 P 节点和一个新的文本节点。我们将文本节点添加到 P 节点，然后将 P 节点和它的文本子节点添加到文档的顶层节点：

```
// 创建一个文档  
myDoc=new XML('<P>paragraph 1</P>');  
  
// 创建一个 P 节点和一个文本节点  
newP = myDoc.createElement("P");  
newText = myDoc.createTextNode("paragraph 2");  
  
// 将文本节点添加到 P 节点  
newP.appendChild(newText);  
  
// 将 P 节点（包括它的文本子节点）添加到 myDoc  
myDoc.appendChild(newP);  
trace(myDoc); // 显示：<P>paragraph 1</P><P>paragraph 2</P>
```

要在文档内移动一个节点，将 *childNode*指定为文档中现有节点的引用。在这种情况下，*childNode*表示节点的原来位置，*theNode*表示节点的新的父节点。在被添加到 *theNode*的过程中，*childNode*从它的原来父节点中被删除掉。例如，我们将节点 B 从它的父节点 P 移动到文档的根处：

```
// 创建一个新的文档  
myDoc = new XML( <P>paragraph 1<B>bold text</B></P> );  
  
// 存储一个到节点B的引用  
boldText = myDoc.firstChild.childNodes[1];  
  
// 将节点B添加到文档的根，同时从 P 中将它删除  
myDoc.appendChild(boldText);  
  
trace(myDoc); // 显示: "<P>paragraph 1</P><B>bold text</B>"
```

我们也可以跳过引用存储的步骤，直接移动节点：

```
myDoc.appendChild(myDoc.firstChild.childNodes[1]);
```

要在文档间移动一个节点，*childNode*应该是一个指向第一个文档（源文档）中某个节点的引用，*theNode*应该是指向第二个文档（目标文档）中某个节点的引用。例如，下面，将 B 节点从 myDoc 移动到 myOtherDoc：

```
myDoc = new XML( <P>paragraph .<B>bold text</B></P> );  
myOtherDoc = new XML();  
  
myOtherDoc.appendChild(myDoc.firstChild.childNodes[1]);  
trace(myDoc); // 显示: "<P>paragraph 1</P>  
trace(myOtherDoc); // 显示: "<B>bold text</B>"
```

## 参见

*XML.createElement()*, *XML.createTextNode()*, *XML.cloneNodee()*,  
*XML.insertBefore()*

---

## XML.attributes 属性

一个数组，其属性存储元素的属性

**有效性** Flash 5

### 概要

*theNode.attributes.attributeIdentifier*  
*theNode.attributes[attributeNameInQuotes]*

**访问** 读 / 写

### 描述

*attributes* 属性存储 *theNode* 所定义的属性的名称和值。例如，P 标签的 ALIGN 属性

```
<P ALIGN="CENTER">This is a paragraph</P>
```

可以用 `theNode.attributes.ALIGN` 或 `theNode.attributes["ALIGN"]` 来访问。如果 P 标签是 XML 源代码中惟一的标签，可以这样来访问 ALIGN 属性：

```
// 创建一个 XML 对象层次
myDoc = new XML('<P ALIGN="CENTER">this is a paragraph</P>');

// 访问 ALIGN 属性，显示： "CENTER"
trace(myDoc.firstChild.attributes.ALIGN);

// 设置 ALIGN 属性
myDoc.firstChild.attributes.ALIGN = 'LEFT';
```

`attributes` 属性本身为一个对象。我们可以将新的属性添加到 `attributes` 对象，因此，将新的属性添加到 `theNode` 如下所示：

```
// 将 CLASS 属性添加到 P 标签
myDoc.firstChild.attributes.CLASS = 'INTRO';

// firstChild 现在表示 XML 源代码
// <P ALIGN="CENTER" CLASS='INTRO'>this is a paragraph</P>
```

因为 `attributes` 不是数组，因此它不包含 `length` 属性。我们要用 `for-in` 循环来访问在元素上定义的所有属性：

```
var count = 0;
for (var prop in theNode.attributes) {
    trace("attribute" + prop + "has the value " + theNode.attributes[prop]);
    count++;
}
trace("The node has" + count + "attributes.");
```

如果 `theNode` 所表示的 XML 元素没有属性，`attributes` 就是一个空的对象，没有属性，上述例子将显示零属性。

## 参见

`XML.nodeType`

## XML.childNodes 属性

一个由指向节点的子节点的引用构成的数组

**有效性** Flash 5

**概要** `theNode.childNodes[n]`

**访问** 只读

## 描述

`childNodes` 属性是一个数组，它的元素包含指向 `theNode` 的直接子节点的引用。它用来访问 XML 层次中的节点。例如，如果我们创建一个对象层次，如下所示：

```
myDoc = new XML('<STUDENT><NAME>Tim</NAME><MAJOR>BIOLOGY</MAJOR></STUDENT>');
```

然后，我们可以用下面的方式访问 STUDENT：

```
myDoc.childNodes[0];
```

我们可以用下面的代码来访问 NAME 和 MAJOR 节点（它们是 STUDENT 的后代）：

```
myDoc.childNodes[0].childNodes[0]; // NAME  
myDoc.childNodes[0].childNodes[1]; // MAJOR
```

如果 `theNode` 之下的层次发生变化，`childNodes` 就自动更新以反映新的结构。例如，如果我们删除 MAJOR 节点，`myDoc.childNodes[0].childNodes[1]` 就会返回 `undefined`。

我们通常要引用节点以操作信息或者重新安排一个文档的结构。例如，我们可以改变某个学生的名字，或者添加一个新的学生，代码如下：

```
// 检查学生的名字  
trace('The student's name is: '  
+myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue);  
  
// 改变学生的名字  
myDoc.childNodes[0].childNodes[0].childNodes[0].nodeValue = 'James';  
  
// 复制 STUDENT 节点  
newStudent = myDoc.childNodes[0].cloneNode(true);  
  
// 将一个新的 STUDENT 节点添加到文档  
myDoc.appendChild(newStudent);
```

注意，出于方便，我们也可以用 `firstChild` 属性来指向 `childNodes[0]`。下面的引用是相同的：

```
myDoc.childNodes[0];  
myDoc.firstChild;
```

要扫描节点的所有子节点，可以用 `for` 语句，如下所示：

```
for (var i = 0; i < theNode.childNodes.length; i++) {  
    trace("child" + i + " is " + theNode.childNodes[i].nodeName);  
}
```

但是，我们的例子只遍历 `theNode` 层次的第一层。`XML.nextSibling` 条目下的例子显

示了如何访问 *theNode* 下的所有节点。如果 *theNode* 没有子节点, *theNode.childNodes.length* 就为 0。

## 用法

表示用来格式化 XML 源代码中空格的空文本节点, 也会在 *childNode* 列表中出现。例如, 在下面的 XML 源代码中, 空的文本节点要通过 BOOK 开始标签和 TITLE, AUTHOR 以及 PUBLISHER 结束标签之后的空格来创建:

```
<BOOK>
  <TITLE>ActionScript: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr">Colin Moock</AUTHOR>
  <PUBLISHER>O'reilly</PUBLISHER>
</BOOK>
```

因此, BOOK 的第一个子节点是一个空的文本节点, 第二个子节点是 TITLE。

## 参见

*XML.firstChild*, *XML.hasChildNodes()*, *XML.lastChild*, *XML.nextSibling*,  
*XML.previousSibling*

---

## XML.cloneNode()方法

创建节点的副本

**有效性** Flash 5

**概要** *theNode.cloneNode(deep)*

**参数**

*deep* 一个布尔值, 表示是否要在复制操作中递归地包括 *theNode* 的子节点。如果为 true, 就复制从 *theNode* 开始的整个层次; 如果为 false, 就只复制 *theNode* 本身 (以及它的属性, 如果它是一个元素节点)。

**返回**

*theNode* 对象的一个副本, 也可以包含它的子树。

**描述**

*cloneNode()* 方法创建并返回 *theNode* 的一个副本, 如果 *theNode* 是一个元素节点, 就包括所有的 *theNode* 属性和值。如果 *deep* 为 true, 返回的副本就包含从 *theNode* 往下的整个节点层次。

我们通常用 *cloneNode()* 基于现有模板 (让我们可以手动产生新的节点结构) 来创建一个新的节点。一旦复制了一个节点, 就可以自定义它, 并且用 *appendChild()* 或者

*insertBefore()*将它插入到一个现有的 XML 文档中。下面的例子复制文档的第一节，并建立一个相同结构的兄弟节点：

```
// 创建一个新的文档  
myDoc = new XML( <P>paragraph 1</P> );  
  
// 制作第一节的一个副本  
newP = myDoc.firstChild.cloneNode(true);  
  
// 自定义副本  
newP.firstChild.nodeValue='paragraph 2';  
  
// 将副本添加到文档  
myDoc.appendChild(newP);  
  
trace(myDoc); // 显示: "<P>paragraph 1</P><P>paragraph 2</P>"
```

注意，元素中的文本是存储在该元素单独的子节点中的，因此必须将 *deep* 设置为 *true* 来将元素的文本内容保存在一个复制操作中。记住，*cloneNode()*不将它所返回的元素插入到节点的文档中——必须用 *appendChild()* 或 *insertBefore()* 来完成这个任务。

## 参见

*XML.appendChild()*, *XML.createElement()*, *XML.createTextNode()*, *XML.insertBefore()*

---

## XML.contentType 属性

通过 *XML.send()* 和 *XML.sendAndLoad()*

发送的 XML 数据 MIME 内容类型

**有效性** Flash 5 修订版 41 及其以后的版本

**概要** *XMLdoc.contentType*

**访问** 读 / 写

### 描述

*contentType* 属性是在调用 *XML.send()* 或 *XML.sendAndLoad()* 的时候被发送到服务器的 MIME 类型。它的默认值为 *application/x-www-form-urlencoded*。*contentType* 属性可以针对指定的 XML 对象或 *XML.prototype* 而进行修改。可以通过修改 *contentType* 影响所有的 XML 对象。

*contentType* 属性首先出现在 Flash 5 播放器的修订版 41 中，在此之前，不能设置 MIME 类型。要检查 *contentType* 的有效性，可以将它同 *undefined* 相比较，或者使用 *getVersion()* 函数来确定播放器的版本和修订版本。

## 用法

参见 *XML.send()* 条目下设置 MIME 类型的重要内容。

## 参见

*XML.send()*, *XML.sendAndLoad()*

---

## XML.createElement()方法

创建一个新的元素节点

**有效性** Flash 5

**概要** `XMLdoc.createElement(tagName)`

### 参数

*tagName* 一个区分大小写的串，表示要创建的元素名称。例如，在标签 `<P ALIGN="RIGHT">` 中，`P` 就是标签的名称。

### 返回

一个新的元素节点对象，没有父节点，也没有子节点。

### 描述

*createElement()* 方法是生成一个 XML 文档对象层次中的新元素节点的主要途径。注意，*createElement()* 不将它所返回的元素插入到 *XMLdoc* 中——必须自己用 *appendChild()* 或 *insertBefore()* 方法来完成这个任务。例如，我们要创建一个新的 `P` 元素并将其一个插入到文档中：

```
myDoc = new XML();
newP = myDoc.createElement('P');
myDoc.appendChild(newP);
```

我们可以将的这些步骤合并起来，如下所示：

```
myDoc.appendChild(myDoc.createElement("P"));
```

*XMLdoc* 必须是 *XML* 类的一个实例，而不是 *XMLnode* 类的。

*createElement()* 方法不能用来创建文本节点，要创建文本节点，可使用 *createTextNode()* 方法。

## 参见

*XML.appendChild()*, *XML.cloneNode()*, *XML.createTextNode()*, *XML.insertBefore()*

---

**XML.createTextNode()方法**

创建一个新的文本节点

**有效性** Flash 5**概要** `XMLdoc.createTextNode( text )`**参数****text** 一个串，包含要成为新节点的 `nodeValue` 的文本。**返回**

一个新的文本节点对象，没有父节点和子节点。

**描述**

*createTextNode()*方法是生成一个 XML 文档对象层次中的新文本节点的主要途径。注意，*createTextNode()*不将它所返回的元素插入到 `XMLdoc` 中——必须自己用 *appendChild()* 或 *insertBefore()* 方法来完成这个任务。例如，我们要创建一个新 P 元素并将其插入到文档中，然后给 P 元素一个文本节点后代：

```
myDoc = new XML();
newP = myDoc.createElement("P");
myDoc.appendChild(newP);

newText = myDoc.createTextNode("This is the first paragraph");
myDoc.firstChild.appendChild(newText);

trace(myDoc); // 显示: '<P>This is the first paragraph</P>'
```

`XMLdoc` 必须是 `XML` 类的一个实例，而不是 `XMLnode` 类的。

文本节点通常存储为元素节点的子节点，元素节点是用 *createElement()* 来创建的。

**参见**

`XML.appendChild()`, `XML.cloneNode()`, `XML.createElement()`, `XML.insertBefore()`

---

---

**XML.docTypeDecl 属性**

文档的 DOCTYPE 标签

**有效性** Flash 5**概要** `XMLdoc.docTypeDecl`**访问** 读 / 写**描述**

`docTypeDecl` 串属性指定 `XMLdoc` 的 DOCTYPE 标签（如果有），否则，`docTypeDecl` 就

是 undefined。*XMLdoc* 必须是 XML 对象层次中顶层的节点（也就是 XML 类的一个实例，而不是 XMLnode 类）。

XML 文档的 DOCTYPE 指定用来确认文档的 DTD 的名称和位置。ActionScript 不执行 XML 文档的确认，它只进行解析。我们用 DOCTYPE 标签来建立可以外部确定的 XML 文档或识别载入 XML 文档的类型。

## 示例

```
var myXML = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">' + '<P>a short document</P>');

trace(myXML.docTypeDecl); // 显示: '<!DOCTYPE foo SYSTEM "bar.dtd">'

// 设置一个新的 DOCTYPE
myXML.docTypeDecl = '<!DOCTYPE baz SYSTEM "bam.dtd">';
```

## 参见

[XML.xmlDecl](#)

---

### XML.firstChild 属性

一个指向节点的第一个子节点的引用

**有效性** Flash 5

**概要** theNode.firstChild

**访问** 只读

#### 描述

`firstChild` 属性和 `childNodes[0]` 类似。它返回指向 `theNode` 下的第一个子节点的引用。如果 `theNode` 没有子节点，`firstChild` 就返回 `null`。

在 XML 源代码标签中，MESSAGE 节点的 `firstChild` 是一个文本节点，它的 `nodeValue` 为 “hey”：

```
<!-- Fragment 1 -->
<MESSAGE>hey</MESSAGE>
```

因此，HOTEL 节点的 `firstChild` 就是 ROOM 节点；

```
<!-- Fragment 2 -->
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

当 `theNode` 在对象层次的顶层时（也就是指向 XML 文档对象），`firstChild` 可能并不总是指向文档中的第一个有用元素。如果一个文档包括 XML 声明 (`<?xml`

version="1.0"?>) 或还有一个 DOCTYPE 标签，那么在 XML 层次的实际根元素之前通常会有一个空格节点。但是，如果 XML 标签没有 XML 声明和 DOCTYPE，我们就可以从文档的 `firstChild` 节点开始处理它，如下所示：

```
// 创建一个新的 XML 标签  
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>');  
  
// 将 XML 标签的第一个节点存储在变量 msg 中  
msg = myDoc.firstChild;  
  
// 将 USER 标签所包含的文本赋给文本域 userNameOutput  
userNameOutput = msg.firstChild.firstChild.nodeValue;
```

用 `nodeValue` 来访问 USER 标签所包含的文本，这种格式很好，但是实际上并不需要。当我们在串语境中使用文本节点对象的时候，`toString()` 方法会自动在节点上调用，返回节点中的文本。

## 参见

`XML.childNodes`, `XML.lastChild`, `XML.nextSibling`, `XML.previousSibling`

---

## XML.hasChildNodes()方法

检查一个节点是否有后代

**有效性** Flash 5

**概要** `theNode.hasChildNodes()`

**返回**

一个布尔值：如果 `theNode` 有子节点就为 `true`，否则为 `false`。

**描述**

`hasChildNodes()` 方法表示一个节点层次是否是从给定的节点扩展出来的。它和下面的比较表达式是同义的：

`theNode.childNodes.length > 0`

如果 `theNode` 不包含子节点，`hasChildNodes()` 就返回 `false`。

**示例**

我们可以用 `hasChildNodes()` 来确定在节点处理期间是否要在一个节点上进行操作。例如，我们要删除文档第一个后代下面的节点，直到第一个后代再也没有子节点：

```
while (myDoc.firstChild.hasChildNodes()) {  
    myDoc.firstChild.firstChild.removeNode();  
}
```

**参见**

`XML.childNodes`

---

**XML.ignoreWhite 属性** 确定在 XML 解析期间是否要忽略空格节点

**有效性** Flash 5 修订版 41 及其以后的版本

**概要** `XMLdoc.ignoreWhite`

**访问** 读 / 写

**描述**

`ignoreWhite` 属性存储了一个布尔值，表示是否要在解析过程中忽略只包含空格的文本节点。默认值为 `false`（不忽略空白节点）。这是一个全局设置，会应用到整个 XML 文档，而不仅仅是一个指定的节点。也就是说，`XMLnode` 类的实例不支持 `ignoreWhite`。

**示例**

要让单独的 XML 文档在解析的过程中去除空白结点，我们使用下面的语句：

```
myXML.ignorewhite = true;
```

要让所有的 XML 文档去除空白结点，则用下面的语句：

```
XML.prototype.ignorewhite = true;
```

`ignorewhite` 属性应该在任何解析 XML 的企图（通常由于 `load()` 或 `sendAnd Load()` 操作）发生之前得到设置。

**参见**

类对象中介绍的手动去除空格代码范例。

---

**XML.insertBefore()方法** 添加一个节点作为前面的兄弟节点

**有效性** Flash 5

**概要** `theNode.insertBefore(newChild, beforeChild)`

**参数**

`newChild` 一个现有的 XML 节点对象。

`beforeChild` `theNode` 的子节点，`newChild` 将要插入到它的前面。

## 描述

*insertBefore()*方法将 *newChild* 添加到 *theNode* 的子节点列表中，添加到 *beforeChild*的前面。*insertBefore()*方法和 *appendChild()*方法类似，但是可以在现有的 XML 对象层次中恰当地定位新的节点。

## 示例

```
// 创建一个文档
myDoc = new XML('<P>paragraph 2</P>');

// 创建一个P节点和一个文本节点
newP = myDoc.createElement('P');
newText = myDoc.createTextNode('paragraph 1');

// 将文本节点添加到P节点
newP.appendChild(newText);

// 将新的P节点(包括它的文本子节点)插入到现有的P节点之前
myDoc.insertBefore(newP, myDoc.firstChild);

trace(myDoc); // 显示: '<P>paragraph 1</P><P>paragraph 2</P>'
```

## 参见

*XML.appendChild()*

---

## XML.lastChild 属性

一个指向节点的最后一个后代的引用

**有效性** Flash 5

**概要** *theNode.lastChild*

**访问** 只读

## 描述

*lastChild*属性和 *childNodes[childNodes.length-1]*作用相同。它返回指向 *theNode*的最后一个子孙的引用。如果 *theNode*没有子节点, *lastChild*就返回null。

在下面的 XML 源代码标签中, MESSAGE 的 *lastChild* 节点是 CONTENT:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

## 示例

```
// 创建一个新的 XML 文档
myDoc = new XML('<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>');
// 将msg设置为 "hi"，因为myDoc 的 firstChild 是 MESSAGE,
```

```
// MESSAGE 的 lastChild 是 CONTENT，而 CONTENT  
// 的 firstChild 是值为 "hi" 的文本节点  
msg = myDoc.firstChild.lastChild.firstChild.nodeValue
```

## 参见

*XML.childNodes, XML.firstChild, XML.nextSibling, XML.previousSibling*

## XML.load()方法

从外部文档中导入 XML 源代码

**有效性** Flash 5

**概要** *XMLdoc.load(URL)*

### 参数

*URL* 一个串，表示要装载的 XML 的位置。

### 描述

*load()*方法导入一个外部的 XML 文档，并对其进行解析，然后将它转换为 XML 对象层次，并且将层次结构放到 *XMLdoc* 中。*XMLdoc* 的任何以前的内容都会被新装载的 XML 内容所代替。

*XMLdoc* 必须是 *XML* 类的一个实例，而不是 *XMLnode* 类的实例。

### 用法

在访问 *load()* 导入的内容之前，我们必须确定装载和解析的操作已经完成了。要这么做，可以检查 XML 文档的 *loaded* 属性值，或者将一个 *onLoad()* 复查处理器赋给文档，以便对装载的完成情况作出响应。参见 *XML.loaded* 和 *XML.onLoad()* 条目可以获得详细的内容。要确定装载数据是否被成功解析，可以检查文档的 *status* 属性。

*XML.load()* 是安全限制领域内的课题，在 *loadVariables()* 全局函数下的表 R-8 中有所描述。

### 示例

```
myDoc = new XML();  
myDoc.load("myData.xml");
```

## 参见

*XML.loaded, XML.onLoad(), XML.sendAndLoad(), XML.status*

**XML.loaded 属性***load()*或*sendAndLoad()*操作的状态**有效性** Flash 5**概要** *XMLdoc.loaded***访问** 只读**描述**

*loaded* 属性返回一个布尔值，表示前面在 *XMLdoc* 上调用的 *load()* 或 *sendAndLoad()* 操作是否完成。当 XML 的 *load()* 或 *sendAndLoad()* 操作开始的时候，它会马上被设置为 *false*。如果装载成功，*loaded* 会设置为 *true*。如果在 *XMLdoc* 上没有执行这样的操作，*loaded* 就是 *undefined*。

当 *loaded* 为 *false* 的时候，XML 数据的装载和解析仍然还在继续，如果试图在 *XMLdoc* 中访问对象层次就会失败。当 *loaded* 为 *true* 的时候，XML 数据完成了装载和解析，并存储在 *XMLdoc* 中，成为对象层次。但是注意，装载进来的 XML 数据可能解析并不成功（使用 *XMLdoc.status* 可以确定这个情况）。

*XMLdoc* 必须是 *XML* 类的一个实例，而不是 *XMLnode* 类的实例。

**示例**

下面的例子显示了一个基本的 XML 预装载器，它会在显示之前等待 XML 数据的装载（XML 预装载器也可以用 *XML.onLoad()* 处理器来建立）：

```
// 第1帧上的代码
// 创建一个新的 XML 文档
myDoc = new XML();
// 将一个外部文件装载到文档中
myDoc.load("userProfile.xml");

// 第5帧上的代码
// 检查数据是否装载完毕，如果完毕，就进入显示帧
// 如果没有，就循环回第4帧，然后播放...
// 循环直到数据装载完成...
if (myDoc.loaded) {
    if (myDoc.status == 0) {
        gotoAndStop("displayData");
    } else {
        gotoAndStop('loadingError');
    }
} else {
    gotoAndPlay(4);
}
```

## 参见

*XML.load()*, *XML.onLoad()*, *XML.sendAndLoad()*

## XML.nextSibling 属性

指向本节点之后节点的引用

**有效性** Flash 5

**概要** *theNode.nextSibling*

**访问** 只读

### 描述

*nextSibling* 属性返回 XML 对象层次中当前层的 *theNode* 之后的节点对象。如果 *theNode* 之后没有节点, *nextSibling* 就返回 null。在下面的 XML 源代码标签中, CONTENT 节点是 USER 节点的 *nextSibling*:

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

### 示例

*nextSibling* 属性的通常用法是遍历一个 XML 对象层次。例如, 要按照出现顺序浏览 *theNode* 的所有子节点, 我们可以用:

```
for (var child = theNode.firstChild; child != null; child = child.nextSibling) {  
    trace("found node: " + child.nodeName);  
}
```

要将循环扩展为一个函数, 我们可以递归地遍历 XML 对象层次中的所有节点, 如下所示:

```
function showNodes (node) {  
    trace(node.nodeName + ": " + node.nodeValue);  
    for (var child = node.firstChild; child != null; child = child.nextSibling) {  
        showNodes(child);  
    }  
}  
  
// 在我们的代码或文档中调用函数  
showNodes(myDoc);
```

注意, 在我们给出的两种遍历示范中, 出现的文本节点并没有 *nodeName* 条目下所描述的名称。

## 参见

*XML.childNodes*, *XML.firstChild*, *XML.lastChild*, *XML.nodeName*,  
*XML.nodeValue*, *XML.previousSibling*

---

**XML.nodeName 属性** 当前节点的名称**有效性** Flash 5**概要** `theNode.nodeName`**访问** 读 / 写**描述**

`nodeName` 串属性表示 `theNode` 的名称。由于 ActionScript 只支持两种节点类型（元素节点和文本节点），因此 `nodeName` 只有两个可能的值：

- 如果 `theNode` 是一个元素节点，`nodeName` 就是一个和该元素的标签名称相匹配的串。例如，如果 `theNode` 表示元素 `<BOOK>`，那么 `theNode.nodeName` 就是 `"BOOK"`。
- 如果 `theNode` 是一个文本节点，`nodeName` 就是 `null`。注意，这和 DOM 规范相背离，规范中规定，文本节点的 `nodeName` 应该是串 `"#text"`。如果你喜欢，可以使用符合 DOM 规范的 `nodeType` 属性。

**示例**

我们可以用 `nodeName` 来检查当前的节点是否是我们需要的元素类型。例如，我们检查 XML 文档第一层中 `H1` 标签的所有内容（这个例子只检查标签名字 `H1`，而不检查小写的 `h1` 标签）：

```
myDoc = new XML('<H1>first heading</H1><P>content</P>' +  
                 '<H1>second heading</H1><P>content</P>');  
for (i = 0; i < myDoc.childNodes.length; i++) {  
    if (myDoc.childNodes[i].nodeName == 'H1') {  
        trace(myDoc.childNodes[i].firstChild.nodeValue);  
    }  
}
```

**参见**`XML.nodeType`, `XML.nodeValue`

---

**XML.nodeType 属性** 当前节点的类型**有效性** Flash 5**概要** `theNode.nodeType`**访问** 只读

## 描述

`nodeType` 是一个整数属性，返回 `theNode` 的类型。由于在 ActionScript 中只支持两种节点类型——元素节点和文本节点——`nodeName` 就只有两个可能的值：1（如果节点为元素节点）和 3（如果节点为文本节点）。这些值看起来是任意的，但是它们实际上是 DOM 所规定的恰当的值。表 R-13 列出了 DOM 中其他的节点类型作为参考。

表 R-13 DOM 节点类型

节点描述	节点类型代码
ELEMENT_NODE <sup>a</sup>	1
ATTRIBUTE_NODE	2
TEXT_NODE <sup>a</sup>	3
CDATA_SECTION_NODE	4
ENTITY_REFERENCE_NODE	5
ENTITY_NODE	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE	12

a. Flash 支持的。从理论上说，ActionScript 除了元素节点和文本节点之外，还执行所谓的 *attribute*, *document* 和 *document\_type* 节点，但是不能将它们作为对象来直接访问。例如，我们可以通过 *attributes* 属性操作节点的属性，但是不能直接访问 *attribute* 节点本身。同样，我们可以通过 *docTypeDecl* 属性来访问文档的 DOCTYPE 标签，但是不能直接访问 *document\_type* 本身。

元素节点对应于 XML 或者 HTML 标签。例如，在 XML 标签 `<P>what is your favorite color?</P>` 中，`P` 标签在 XML 对象层次中会被表示为元素节点 (`nodeType` 1)。在 XML 源代码中被标签所包含的文本——例如，文本 “`what is your favorite color?`”——会被表示为文本节点 (`nodeType` 3)。

## 示例

我们可以有条件地基于节点的 `nodeType` 来操作一个节点。例如，我们将删除 `theNode` 子节点中所有空的文本节点：

```
// 循环扫描 theNode 的所有子节点
for (i = 0; i < theNode.childNodes.length; i++) {
    // 如果当前的节点是文本节点...
    if (theNode.childNodes[i].nodeType == 3) {
        // 检查节点中任何有用的字符
        var j = 0;
        var emptyNode = true;
        for (j = 0; j < theNode.childNodes[i].nodeValue.length; j++) {
            // 有用的字符编码从 ASCII 32 开始
            if (theNode.childNodes[i].nodeValue.charCodeAt(j) > 32) {
                emptyNode = false;
                break;
            }
        }
        // 没有发现有用的字符，因此删除节点
        theNode.childNodes[i].removeNode();
    }
}
```

## 参见

[XML 类](#), [XML.nodeName](#), [XML.nodeValue](#)

---

## XML.nodeValue 属性

当前节点的值

**有效性** Flash 5

**概要** theNode.nodeValue

**访问** 读 / 写

### 描述

nodeValue 属性表示 theNode 的串值。由于 ActionScript 中只支持两种节点类型（元素节点和文本节点），因此 nodeValue 只有两个可能的值：

- 如果 theNode 是一个元素节点，nodeValue 就是 null。
- 如果 theNode 是一个文本节点，nodeValue 就是包含在该节点中的文本。

实际上，nodeValue 通常只用在文本节点上。要将新的文本赋给一个现有的文本节点，我们使用 nodeValue，如下所示：

```
// 创建一个新的 XML 文档
myDoc = new XML('<H1>first heading</H1><P>content</P>');
// 修改 H1 标签所包含的内容
myDoc.firstChild.firstChild.nodeValue = "My Life Story";
```

虽然我们可以用 `nodeValue` 显式地获取文本节点的值, `toString()`方法在使用于串语境的时候却可以隐式地返回一个节点的值。因此, 下面的代码将在输出窗口中显示文本节点中的文本:

```
trace(myDoc.firstChild.firstChild);
```

## 参见

`XML.nodeName`, `XML.nodeType`

## XML.onData()事件处理器

当外部 XML 源代码装载结束  
但还没有被解析的时候执行

**有效性** Flash 5 (未文档化)

**概要** `XMLdoc.onData(src);`

### 参数

`src` 一个串, 包含装载的 XML 源代码。

### 描述

`onData()`处理器会在原始的 XML 源代码通过早前调用的 `load()`或 `sendAndLoad()`完全装载到 `XMLdoc` 中的时候自动执行。在默认情况下, `onData()`会进行下面的操作:

- 如果所接收的原始源代码为 `undefined`, 它就调用 `XMLdoc.onLoad()`, 并将 `success` 参数设置为 `false`。
- 否则, 它将源代码分解到 `XMLdoc` 中, 将 `XMLdoc.loaded` 设置为 `true`, 并且调用 `XMLdoc.onLoad()`, 同时将 `success` 参数设置为 `true`。

`onData()`处理器可以被赋予一个自定义的复查函数, 以便在 ActionScript 解析原始的 XML 源代码之前中途截取它。在特定的环境下, 手动操作原始的 XML 源代码可能比执行 ActionScript 内置解析性能更好。

### 示例

下面的例子显示了如何显示装载进来的 XML 源代码, 同时防止它被 ActionScript 解析:

```
myDoc = new XML();
myDoc.onData = function(src) {
    trace("Here's the source:\n"+src);
}
myDoc.load('book.xml');
```

**参见**

*XML.onLoad()*

**XML.onLoad()事件处理器** 当外部 XML 数据装载和解析完成的时候执行

**有效性** Flash 5

**概要** `XMLdoc.onLoad(success)`

**参数**

*success* 一个布尔值，表示装载是成功 (*true*) 还是失败 (*false*)。

**描述**

*XMLdoc* 的 *onLoad()* 处理器会在外部 XML 文件通过 *load()* 或 *sendAndLoad()* 方法装载到 *XMLdoc* 中的时候自动执行。在默认情况下，XML 文档对象的 *onLoad()* 处理器是一个空的函数。要使用 *onLoad()*，我们可以赋给它一个复查处理器（也就是子定义的函数）。例如：

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
function handleLoad(success) {
    // 在这里处理 XML...
}
```

依靠 *onLoad()* 事件可以判断什么时候处理 *XMLdoc* 才是安全的。如果 *onLoad()* 被触发，我们就知道外部 XML 数据的装载和解析操作已经结束了，因此，可以安全地访问装载进来的内容。有了 *onLoad()* 处理器，就不再需要编写在调用 XML 的 *load()* 函数之后等待数据到达的预装载代码。例如，在下面的代码中，我们装载一个 XML 文档，然后等待我们自定义的 *handleLoad()* 函数在装载完成的时候自动执行。如果装载成功，就用 *display()* 函数继续 XML 处理；否则，执行 *display()* 函数，显示错误信息。（*displayProduct()* 和 *displayError()* 函数是自定义的函数，假设你已经编写好了，用来对用户显示一些信息，但是没有给出它们。）代码如下：

```
myDoc = new XML();
myDoc.onLoad = handleLoad;
myDoc.load("productInfo.xml");

function handleLoad(success) {
    if(success) {
        output = "Product information received";
        displayProduct(); // 调用自定义的显示函数
    } else {
        output = "Attempt to load XML data failed";
    }
}
```

```
    displayError(); // 调用自定义的显示函数
}
}
```

注意，*handleLoad()*的*success*参数值会自动被解释程序设置为*true*或*false*，表示装载是否正确完成。但是，*success*参数在理论上比在实际上显得更为有用。大部分的Web服务器消息（例如：“404 File Not Found”）都是以HTML文档格式出现的。由于HTML完全可以解析为XML数据，因此，服务器错误信息页面的接收就会让该页被解析为目标XML文档对象。因为页面解析正确，装载动作就会被认为是成功的，*success*就是*true*，虽然实际上XML文件可能并没有找到，或者遇到的是其他某个服务器的错误信息。要肯定你得到的是你所想要的数据，可以显式地测试它的结构或内容，以检查某些确认特征，比如特定了节点的*nodeName*。也可以参见*XML.onData()*事件处理器，它可以用来执行自定义的解析操作。

## 参见

*XML.load()*, *XML.onData()*, *XML.sendAndLoad()*

---

## XML.parentNode 属性

指向节点的直接祖先的引用

**有效性** Flash 5

**概要** *theNode.parentNode*

**访问** 只读

### 描述

*parentNode*属性返回一个指向XML对象层次中*theNode*所继承的节点对象的引用。如果*theNode*是当前层次中的顶层节点，*parentNode*就返回*null*。

在下面的XML源代码标签中，MESSAGE节点是文本节点“hey”的*parentNode*:

```
<MESSAGE>hey</MESSAGE>
```

ROOM节点的*parentNode*为HOTEL节点:

```
<HOTEL><ROOM><SIZE>Double</SIZE></ROOM></HOTEL>
```

## 参见

*XML.childNodes*, *XML.firstChild*, *XML.lastChild*, *XML.previousSibling*

---

## XML.parseXML()方法

解析一个 XML 源代码的串

**有效性** Flash 5

**概要** `XMLdoc.parseXML(string)`

**参数**

`string` 一个 XML 源代码的串。

### 描述

*parseXML()*方法类似于内部的*load()*函数，它读出并解析包含在 `string` 中的源代码，将 XML 转换为一个对象层次，然后将结果层次结构放到 `XMLdoc` 中。`XMLdoc` 以前的任何内容都会被新的层次所代替。`XMLdoc` 必须是 `XML` 类的一个实例，而不是 `XMLnode` 类的实例。

要在文本节点中包括原始 HTML 或者 XML 源代码而不解析它，可以用 CDATA 片段，如下所示：

```
<![CDATA[ source]]>
```

例如，下面的代码创建一个 `MESSAGE` 元素，它有一个单独的子节点，为文本节点，包含文本 “`<B>Welcome</B>to my site`” (`<B>` 标签不被编译为一个 XML 标签，不会成为 XML 对象层次的一个部分)：

```
myDoc = new XML();
myDoc.parseXML("<MESSAGE><![CDATA[<B>Welcome</B>to my site]]></MESSAGE>");
trace(myDoc); // 显示: "<MESSAGE><B>Welcome</B>to my site</MESSAGE>"
```

### 示例

我们可以用 *parseXML()* 作为基于内存构成 XML 源代码（例如一些用户输入），用新的层次来代替 XML 对象中当前对象层次的一种方法。在下面的例子中，通过将文本域 `username` 和 `content` 中的输入同标高进行合并，从而创建一个简单的 XML 消息：

```
myDoc = new XML();
myXMLsource = "<MESSAGE><USER>" + username + "</USER><CONTENT>" +
    content + "</CONTENT></MESSAGE>";
myDoc.parseXML(myXMLsource);
```

### 参见

`XML.load()`, `XML.status`

---

**XML.previousSibling 属性**

指向当前节点之前节点的引用

**有效性** Flash 5**概要** theNode.previousSibling**访问** 只读**描述**

previousSibling 属性返回一个指向 XML 对象层次中当前层的 theNode 之前节点对象的引用。如果在层次的当前层中 theNode 前面没有节点存在，就返回 null。

在下面的 XML 源标签中，CONTENT 节点的 previousSibling 是 USER 节点：

```
<MESSAGE><USER>gray</USER><CONTENT>hi</CONTENT></MESSAGE>
```

**示例**

previousSibling 属性可以用来遍历一个 XML 对象层次，虽然在这方面 nextSibling 更为常用。要按照相反的顺序浏览 theNode 的所有子节点，可以使用：

```
for (var i = theNode.lastChild; i != null; i = i.previousSibling) {  
    trace('found node: ' + i.nodeName);  
}
```

**参见**

[XML.childNodes](#), [XML.firstChild](#), [XML.lastChild](#), [XML.nextSibling](#),  
[XML.nodeName](#), [XML.nodeValue](#), [XML.parentNode](#)

---

**XML.removeNode()方法**

从 XML 对象层次中删除一个节点

**有效性** Flash 5**概要** theNode.removeNode()**描述**

removeNode() 方法从 XML 文档中删除 theNode。theNode 的所有子孙（子节点、孙节点等）也会被删除。theNode 的父节点的 childNodes 属性会自动修改以反映剩余对象层次的结构。

**示例**

下面，我们删除第二个子节点，让第三个子节点代替它的位置：

```
myDoc = new XML('<P>one</P><P>two</P><P>three</P>');
```

```
myDoc.childNodes[1].removeNode();
trace(myDoc); // 显示: "<P><cre</P><P>three</P>"
```

## 参见

*XML.appendChild()*

## XML.send()方法

将 XML 源代码发送给脚本

**有效性** Flash 5

**概要** *XMLdoc.send(URL, window)*

### 参数

*URL* 一个串，指定 *XMLdoc* 要发送到的脚本或应用程序的位置。

*window* 一个要求出现的串，指定装载脚本响应信息的浏览器窗口或框架的名称。可以是一个自定义的名称，或下面四种预设置之一：“\_blank”、“\_parent”、“\_self”、“\_top”。有关详细内容，请参见全局函数 *getURL()* 下的窗口设置描述。

### 描述

*send()* 方法将 *XMLdoc* 转换为 XML 源代码的串，并且将 HTTP 请求中的代码发送给 URL 中的脚本或者应用程序。脚本或应用程序要按照某种方式处理 XML，并且可选地返回一个响应——通常是一个 Web 页面——到浏览器，然后在 *window* 中显示出来。注意，响应是被浏览器而不是 Flash 捕捉到的，可以用 *sendAndLoad()* 在 Flash 中得到响应。

当 *XML.send()* 从运行在浏览器中的 Flash 播放器被调用的时候，*XMLdoc* 就通过 POST 方法被发送；当 *XML.send()* 从作为独立应用程序的 Flash 播放器中调用的时候，*XMLdoc* 就通过 GET 方法发送。接收传递的 XML 串的服务器应用程序必须能够直接访问 HTTP 请求的原始 POST 数据，并且能够将它作为一般的名/值对来解析。在 Perl 中，POST 请求中的数据可以从 STDIN 中访问，可以在 \$buffer 中提取和存储，如下所示：

```
read(STDIN, $buffer, SENV('CONTENT_LENGTH'));
```

在 ASP 中，原始的 POST 数据可以通过 *Request.BinaryRead* 方法来访问。一些应用程序（例如 Cold Fusion）没有直接访问 POST 请求中的数据的直接途径。在这些情况下，可能有必要用 *XML.toString()* 首先将 XML 对象转换为串，然后将该串作为变量用 *loadVariables()* 传递给服务器。

发送给服务器的 XML 文本的默认 MIME 内容类型为 application/x-www-form-urlencoded。但是，这个类型只是表象——文本本身并不是 URL 编码的。在 Flash 5 播放器的修订版 41 及其以后版本中，MIME 内容类型可以用 XML.contentType 属性来修改。例如，要将 MIME 类型设置为 application/xml，我们使用：

```
myXML = new XML();
myXML.contentType = "application/xml";
```

但是，显式地将 contentType 属性设置为 application/x-www-form-urlencoded 仍然不会使发送文本变成 URL 编码的。

注意，对于 Flash 5 播放器的 41 修订版来说，当 XML 源代码被解析，字符 &、'、"、< 和 > 出现在文本节点中的时候，它们会被转换为下面的东西：&amp;、&apos;、&quot;、&gt;、&lt;。这个转换在 Flash 中是明显的，因为当 XML 对象被转换为串的时候，这些东西被转换回了原来的字符形式；但是，它们将会出现在发送给服务器的 XML 源代码中。

## 示例

```
myDoc = new XML("<SEARCH_TERM>tutorials</SEARCH_TERM>");
myDoc.send("http://www.domain.com/cgi-bin/lookup.cgi", "remoteWin");
```

## 参见

*XML.sendAndLoad()*, *XML.load()*, *XML.loaded*, *XML.onLoad()*, *XML.status*

---

## XML.sendAndLoad()方法

将 XML 源代码发送给脚本，并且接收返回的 XML 源代码

**有效性** Flash 5

**概要** `XMLdoc.sendAndLoad(URL, resultXML)`

**参数**

**URL** 一个串，指定 `XMLdoc` 要发送到的脚本或者应用程序。

**resultXML** 一个指向要接收返回的 XML 源代码的 XML 文档对象的引用。

## 描述

`sendAndLoad()` 方法将 `XMLdoc` 连接为 XML 源代码串，并且将这个串发送给 `URL` 中的脚本或应用程序。脚本或应用程序要按照某种方法处理 XML，并且将 XML 文档作为响应发送回去。响应文档被 Flash 捕获，进行解析，转换为 XML 对象层次，然后放

在 `resultXML` 中。 `resultXML` 中以前的任何内容都将被新装载的 XML 内容所代替。参见 `XML.send()` 可以获得关于将 XML 发送到服务器的更多重要信息。

## 用法

在访问 `sendAndLoad()` 所导入的内容之前，我们必须确定装载和解析操作已经完成。要这么做，可以检查 `resultXML` 的 `loaded` 属性，或者，为 `resultXML` 赋一个 `onLoad()` 事件处理器，以便对装载的完成作出响应。请参见 `XML.loaded` 和 `XML.onLoad()` 条目中的具体内容。要确定装载数据的解析结果，可以检查文档的 `status` 属性。

`XML.sendAndLoad()` 是全局函数 `loadVariables()` 条目下表 R-8 中所描述的安全显示范围内的项目。

## 示例

```
// 创建一个 XML 文档  
myDoc = new XML("<P>hello server!</P>");  
  
// 创建一个空的 XML 文档来接收服务器的响应  
serverResponse = new XML();  
  
// 将 myDoc 发送给服务器，然后将响应放在 serverResponse 中  
myDoc.sendAndLoad("http://www.domain.com/cgi-bin/readData.cgi", serverResponse);  
  
// 将 onLoad 处理器添加到 serverResponse，在文本域 output 中显示来自服务器的响应  
serverResponse.onLoad = function() {  
    output = serverResponse.toString();  
}
```

关于将 XML 发送给服务器的初级读本，参见 Macromedia 的文章“Integrating XML and Flash in a Web Application”（在 Web 应用程序中生成 XML 和 Flash），网址为：

<http://www.macromedia.com/support/flash/interactivity/xml>

## 参见

`XML.load()`, `XML.loaded`, `XML.onLoad()`, `XML.send()`, `XML.status`

---

### XML.status 属性

表示将 XML 源代码分解到对象层次中的操作是否成功

**有效性** Flash 5

**概要** `XMLdoc.status`

**访问** 只读

## 描述

`status` 属性返回一个数字的状态编码，表示在解析 XML 源代码的过程中是否有错误产生。当源 XML 为下面形式的时候要进行解析：

- 作为参数提供给 `XML()` 构造器的时候。
- 通过 `parseXML()` 方法进行显式解析的时候。
- 通过 `load()` 或者 `sendAndLoad()` 方法装载到新的 XML 对象中的时候。

`status` 代码如表 R-14 所示。如果在解析的过程中没有错误，那么 `status` 为 0，表示成功。错误是用负数来表示的。在遇到第一个错误的时候，解析就停止了，因此其他错误在你修正了前面的错误之后还会出现。

表 R-14 XML 解析状态代码

状态	描述
0	文档的解析没有错误（也就是成功）
-2	CDATA 部分没有正确结束
-3	XML 声明没有正确结束
-4	DOCTYPE 声明没有正确结束
-5	注释没有正确结束
-6	XML 元素有问题
-7	没有足够的内存空间来解析 XML 源代码
-8	某个属性值没有正确结束
-9	某个开始标签没有对应的结束标签
-10	某个结束标签没有对应的开始标签

我们通常用 `status` 来确定装载外部 XML 文件的过程是否安全。在检查 `status` 之前检查 `loaded` 属性可以确定 `load()` 或 `sendAndLoad()` 命令是否完成。注意，所有的 ActionScript 的 XML 解析器都不能确认违反 DTD 的文档，它对格式有着很高的要求。

## 示例

```
myDoc = new XML(<BOOK>Colin Moock</AUTHOR></BOOK>);  
trace(myDoc.status); // 显示: "-10" (没有开始标签)
```

## 参见

`XML.load()`, `XML.loaded`, `XML.onLoad()`, `XML.parseXML()`, `XML.sendAndLoad()`

---

## XML.toString()方法

XML 节点的源代码，为一个串

**有效性** Flash 5

**概要** `theNode.toString()`

**返回**

一个串，表示 XML 对象层次的源代码，从 `theNode` 开始。

**描述**

`toString()` 方法将 XML 节点对象或一个 XML 文档对象转换为它的等价 XML 源代码。如果 `theNode` 是顶层 XML 文档对象，那么串中就会包含所有的 DOCTYPE 和 XML 声明标签。如果文档的 `ignoreWhite` 属性是 `false`，那么空格就会被保留，文档源代码和解析时一样。

通常没有必要显式地调用 `toString()`，`toString()` 在 `theNode` 用于串语境的时候会自动调用。

**示例**

```
var myDoc = new XML('<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd"><BOOK>
<TITLE>ActionScript: The Definitive Guide</TITLE>
  <AUTHOR SALUTATION="Mr">Colin Moock </AUTHOR>
  + '<PUBLISHER>O\'reilly&Associates, Inc</PUBLISHER>  </BOOK> );
trace(myDoc.toString());
// 显示:
<?xml version="1.0"?><!DOCTYPE foo SYSTEM "bar.dtd">
<BOOK>  <TITLE>ActionScript:
The Definitive Guide</TITLE><AUTHOR SALUTATION="Mr">Colin
Moock </AUTHOR>  <PUBLISHER>O'reilly & Associates, Inc
</PUBLISHER>  </BOOK>
```

**参见**

`Object.toString()`, `XML.nodeValue`

---

## XML.xmlDecl 属性

文档的 XML 声明标签

**有效性** Flash 5

**概要** `XMLdoc.xmlDecl`

**访问** 读 / 写

## 描述

`xmlDecl` 串属性表示 `XMLdoc` 的 XML 声明标签（如果有），否则，`xmlDecl` 就为 `undefined`。`XMLdoc` 必须是 `XML` 对象层次中的顶层节点（也就是 `XML` 类的一个实例，而不是 `XMLnode` 类的实例）。

`XML` 文档的 XML 声明标签用来确定文档中使用的 XML 的版本。我们用 XML 声明标签来建立格式良好的 XML 文档，它可以进行显式确认。

## 示例

```
// 一个格式规范的文档(但是不能通过 DTD 确认)
myXML = new XML('<?xml version="1.0"?><p>this is a short document</p>');
trace(myXML.xmlDecl); // 显示: '<?xml version="1.0"?>'
// 设置一个新的 XML 声明
myXML.xmlDecl = '<?xml version="1.0" standalone="no"?>';
```

## 参见

`XML.docTypeDecl`

---

## XMLnode 类

XML 类的内部超类

**有效性** Flash 5

## 描述

`XMLnode` 类定义了 `XML` 对象层次节点的核心属性和方法。虽然 `XMLnode` 是一个内部设备，但是程序员可以用它来扩展 `XML` 对象的默认功能。

每一个 `XML` 对象层次在理论上都包含两种对象节点：

- 一个 `XML` 节点，作为层次的主要容器。
- 任意数量的 `XMLnode` 节点，它们是主容器节点的子节点。

主容器节点是 `XML` 类的实例。例如，如果我们创建 `myDoc`，如下所示：

```
myDoc = new XML();
```

那么 `myDoc` 就是 `XML` 类的一个实例。`XML` 类从 `XMLnode` 类继承而来，因此，主容器节点有 `XMLnode` 中定义的所有属性和方法，以及 `XML` 自己定义的属性和方法。相反，`myDoc` 的子节点实际上不是 `XMLnode` 类的实例，而是 `XML` 类的实例：

```
myParagraph = myDoc.createElement("P");
```

于是，`myParagraph` 就是 `XMLnode` 类的一个实例。通常，两种节点类之间的内在差

别并不影响使用 XML 对象。但是，如果希望将一个继承属性添加到所有的 XML 对象中，那么必须使用 *XMLnode* 类的 *prototype*，而不是 *XML* 类的 *prototype*（参见下面的例子），添加到 *XMLnode.prototype* 的任何方法或属性都可以由影片中所有的 XML 节点所继承。

表 R-15 给出了 *XMLnode* 和 *XML* 所定义的属性、方法和事件处理器，以供参考。注意，所有列出的项目都可以通过 *XML* 类的实例访问到，而 *XML* 所定义的方法却不能通过 *XMLnode* 的实例而访问到。例如，*load()* 方法可以在 *XML* 类的实例上调用，而不能在 *XMLnode* 类的实例上调用。关于每一个项目的具体讨论，参见对应的 *XML* 类条目。

表 R-15 XMLnode 和 XML 的属性、方法和事件处理器

XMLnode 和 XML	仅对 XML
<i>appendChild()</i>	<i>contentType</i>
<i>attributes</i>	<i>createElement()</i>
<i>childNodes</i>	<i>createTextNode()</i>
<i>cloneNode()</i>	<i>docTypeDecl</i>
<i>firstChild</i>	<i>ignoreWhite</i>
<i>hasChildNodes()</i>	<i>load()</i>
<i>insertBefore()</i>	<i>loaded</i>
<i>lastChild</i>	<i>onData()</i>
<i>nextSibling</i>	<i>onLoad()</i>
<i>nodeName</i>	<i>parseXML()</i>
<i>nodeType</i>	<i>send()</i>
<i>nodeValue</i>	<i>sendAndLoad()</i>
<i>parentNode</i>	<i>status</i>
<i>previousSibling</i>	<i>xmlDecl</i>
<i>removeNode()</i>	
<i>toString()</i>	

## 示例

下面的代码将一个自定义的 *secondChild()* 方法添加到 *XMLnode.prototype*（然后，*secondChild()* 方法可以从影片中的任何 XML 节点访问到）：

```
XMLnode.prototype.secondChild = function() {
```

```
    return this.childNodes[1];
}

myDoc = new XML('<PRODUCT>Cell Phone</PRODUCT><PRODUCT>Game Console
</PRODUCT>');
trace(myDoc.secondChild()); // 显示：'<PRODUCT>Game Console</PRODUCT>'
```

通过 `XML.prototype` 来扩展 `XML` 类也是完全合法的，但是，这样的扩展只应用在主容器节点上（`XML` 类的直接实例）。

## 参见

`XML` 类，第十二章

---

## XMLSocket 类

支持连续的服务器 / 客户 TCP/IP 连接

**有效性** Flash 5

**概要** `new XMLSocket()`

### 方法

`close()` 终止对服务器应用程序的开放连接。

`connect()` 试图建立服务器应用程序的新连接。

`send()` 将 XML 对象层次作为串发送到服务器应用程序。

### 事件处理器

`onClose()` 在服务器终止连接的时候执行。

`onConnect()` 在连接完成的时候执行。

`onData()` 在数据被接收到但是还没有解析为 XML 的时候执行。

`onXML()` 在数据已经被接收并且解析为 XML 对象层次的时候执行。

### 描述

Flash 和服务器之间的大部分连接都非常短暂。当 Flash 通过 `loadMovie()`, `loadVariables()` 或 `XML.load()` 函数请求外部数据的时候，就建立一个临时的连接通道。数据通过通道发送，然后通道就被终止了。这种短暂的连接有很多有效的应用，但是在两个重要的方面也有所限制：

- 一旦连接关闭，服务器就没有办法和 Flash 联系，Flash 必须总是启动和服务器的连接。

- 每一次 Flash 从服务器获得信息的时候，必须打开一个新的连接。用在开放的重複连接中的时间和处理器负载可以防止 Flash 忙于和服务器之间近乎实时的处理。

对于 Flash 5 来说，可以用 *XMLSocket* 类来覆盖这些限制，它允许在服务器应用程序和 Flash 之间打开一个稳定的连接。*XMLSocket* 可用来开发要求频繁的服务器更新的系统，比如聊天室或者网络多人游戏。

为了用 *XMLSocket* 连接到一个远程的应用程序，必须首先创建和存储一个 *XMLSocket* 对象，如下所示：

```
mySocket = new XMLSocket();
```

然后，调用 *connect()* 方法，它让 Flash 和远程应用程序之间建立一个连接。例如：

```
mySocket.connect("http://www.myserver.com", 8000);
```

一旦连接建立，*XMLsocket* 对象就会成为传送 / 接收者。我们通过调用接口的 *send()* 方法而将 XML 格式的数据发送给远程应用程序，当接口的 *onXML()* 事件被触发的时候，我们就知道已经接收到了 XML 格式数据。

使用 *XMLSocket* 的服务器应用程序必须：

- 在大于等于 1024 的指定端口上服务 TCP/IP 接口连接。
- 按照 zero 字节（也就是 ASCII 的 null 字符）所分隔成的片段来发送 XML 格式数据。

服务器应用程序通常是由服务器端程序员而不是 Flash 程序员创建的。将接收到的所有消息发送给所有连接客户的简单服务器应用程序举例，可以参见在线代码库中的 Java *XMLSocket* 服务器。

*XMLSocket* 连接直到下面的情况发生时才终止：

- *XMLSocket* 对象的 *clos()* 方法被调用。
- 再也没有指向 *XMLSocket* 对象的引用存在了。
- 服务器终止了连接（这将触发 *onClose()* 事件）。
- 影片被关闭或者 Flash 播放器退出。

令人高兴的是，*XMLSocket* 类还为我们提供了一个监控连接的方法。它包括三个属性——*onClose*, *onConnect*, *onXML*——允许我们确定对应事件发生的时候会触发哪一个事件处理器。这样的事件处理器通常称为复查处理器，因为它们通过 Flash 自动

触发，以响应在程序员直接控制范围之外的一些事件（从这种角度来说，它们和 ActionScript 的内置剪辑和按钮事件处理器非常类似，除了处理器函数是程序员定义的之外）。例如，当一个连接被服务器关闭的时候，*onClose* 属性所定义的处理器就会被触发。

---

**注意：**如果没有为 *onConnect* 和 *onClose* 属性定义复查处理器，你就不能执行任何错误检测或提供任何反应动作。如果没有为 *onXML* 属性定义一个复查处理器，那么，当接口从服务器端应用程序接收数据的时候，你就不会被告知，也不能获取这样的数据。

---

## 示例

下面的例子给出了执行一个简单的聊天客户程序所需要的精辟代码。可以在下面的网址看到这个客户程序的运行情况：

*http://www.moock.org/chat*

服务器和客户程序都可以在在线代码库中得到：

```
// 一个简单的聊天客户程序
// *** 普通初始化
var incomingUpdated = false; // 跟踪我们是否需要滚动到 incoming
                             // (聊天文本域) 的最后
var incoming = ''; // 为主聊天文本域赋予一个初始值

// 添加滚动控制影片，它让聊天文本域在每次添加消息的时候
// 显示下一行(最近的行)
// 注意，我们只需要滚动控制，这是由于 Flash 5 播放器修订版 30 中的一个文本域滚动故障
attachMovie('processScroll', "processScroll", 0);

// 在接收一个消息的时候添加声音
var click = new Sound();
click.attachSound("click");

// 当用户加入或离开的时候添加声音
var welcomeSound = new Sound();
welcomeSound.attachSound('welcome');

// 关闭按钮上的黄色高亮状态
_focusrect = 0;

// *** 创建一个新的接口，试图连接到服务器
function connect() {
    // 创建一个新的 XMLSocket 对象
    mySocket = new XMLSocket();

    // 将复查函数赋给 mySocket 的处理器
    mySocket.onConnect = handleConnect;
```

```
mySocket.onClose = handleClose;
mySocket.onXML = handleIncoming;

// 尝试连接，并将mySocket.connect()的返回值赋给connectSuccess
// (如果初始连接成功，Connect()就返回true)
var connectSuccess = mySocket.connect("www.myserver.com", 1025);
if (connectSuccess) {
    trace("initial connection succeeded");
} else {
    // connectSuccess 为 false，因此我们不能建立连接
    gotoAndStop('connectionFailed');
    trace('initial connection failed');
}

// *** 响应连接完成的事件处理器
function handleConnect (succeeded) {
    // 如果handleConnect()的succeeded参数为true,
    // 连接就是建立的，我们可以继续聊天
    // 否则，显示一个失败信息
    if (succeeded) {
        // 设置一个属性，表示我们有一个开放的连接是可用的
        mySocket.connected = true;
        gotoAndStop('chat');
        // 将光标放到 "send message" 文本域
        Selection.setFocus("_level0.outgoing");
    } else {
        // 连接不成功，因此显示错误消息
        gotoAndStop("connectionFailed");
        trace('connection failed');
    }
}

// *** 当服务器关闭连接的时候调用的事件处理器
function handleClose() {
    // 告诉用户，连接已经丢失
    incoming += ("The server has terminated the connection.\n");
    // 更新聊天文本域，以让滚动控制器知道
    mySocket.connected = false;
    numClients = 0;
}

// *** 接收和显示引入消息的事件处理器
function handleIncoming(messageObj) {
    // 在输出窗口中显示接收到的 XML 数据
    trace("-----new data received-----");
    trace(">>" + messageObj.toString() + "<<");
    trace("----- end of new data -----");

    // 我们要更新聊天文本域，因此要让滚动控制器知道
    incomingUpdated = true;
    lastScrollPos = incoming.scroll;
```

```
// 检查时间
var now = new Date();
var hours = now.getHours();
var minutes = now.getMinutes();
var seconds = now.getSeconds();
// 格式化时间以便输出
hours = (hours < 10 ? "0" : "")+hours;
minutes = (minutes < 10 ? "0" : "")+minutes;
seconds = (seconds < 10 ? "0" : "")+seconds;

// 如果我们在 XML 对象中发现 NUMCLIENTS,
// 那么服务器在客户连接或者不连接的任何时候发送 NUMCLIENTS...
if (messageObj.firstChild.nodeName == "NUMCLIENTS") {
    // ...然后检查引入消息窗口是否是空的, 如果是...
    if (incoming == "") {
        // ...那么用户就是刚加入的, 因此添加一个欢迎消息
        incoming += ("Welcome to moock comm 1.0.0,
            + userID + "\n"
            + " connection time: " + hours + ":" + minutes + ":" + seconds + "\n"
            + " server:clayton\5 JavaComm generic flash xml socket server\n\n");
    } else {
        // 否则, 就是已经到达或者已经离开的, 因此要告诉用户
        if (parseInt(messageObj.firstChild.firstChild.nodeValue) > numClients) {
            // 报告到达聊天窗口的客户
            incoming += (hours + ":" + minutes + ":"
                + seconds + " a new user has connected.\n");
        } else {
            // 报告离开聊天窗口的客户
            incoming += (hours + ":" + minutes + ":"
                + seconds + " a user disconnected.\n");
        }
    }
    // 最后, 跟踪新的客户数量
    // 并播放欢迎 / 欢送声音
    numClients = parseInt(messageObj.firstChild.firstChild.nodeValue);
    welcomeSound.setVolume(100);
    welcomeSound.start();
} else {
    // 没有发现 NUMCLIENTS 节点, 因此这是一个正规消息
    // 从 XML 对象中获取用户名和消息
    var user = messageObj.firstChild.firstChild.firstChild.nodeValue;
    var message = messageObj.firstChild.childNodes[1].firstChild.nodeValue;

    // 将消息添加到聊天窗口, 附带一个时间标记
    incoming += (hours + ":" + minutes
        + ":" + seconds + user + ">> " + message + "\n");
    // 现在处理新的消息声音
    // 如果从最后一个消息开始已经过了 30 秒,
    // 就发出一个较响的声音, 否则发出一个较低的声音
    trace('time since last message: ' + (now - lastIncomingMessageTime));
    if (lastIncomingMessageTime && (now - lastIncomingMessageTime) > 30000) {
        click.setVolume(200);
    } else {
```

```
        click.setVolume(30);
    }
    click.start();
}

// 如果长于 5000 个字符，就截短主聊天文本域的内容
if (incoming.length > 5000) {
    var nearestNewline = incoming.indexOf("\n", incoming.length - 5000);
    incoming = incoming.substring(nearestNewline, incoming.length);
}

// 为下一次的处理而记录消息到达的时间
lastIncomingMessageTime = now;
}

// *** 将一个新的 XML 对象发送到服务器
function sendMessage() {
    // 创建一个消息，作为 XML 源标签发送
    // 注意，<USER> 和</MESSAGE> 标签之前要求空格，
    // 以便 MESSAGE 和 USER 总是有一个文本子节点
    var message = '<USER> ' + userID + '</USER><MESSAGE>' +
        outgoing + '</MESSAGE>';

    // 将消息转换为一个 XML 对象层次
    messageObj = new XML();
    messageObj.parseXML(message);

    // 检查我们所发送的东西
    trace("Sending: " + messageObj);

    // 如果一个接口对象已经创建并且已经连接，那么就发送 XML 消息
    // 否则就警告用户，他需要先连接
    if (mySocket && mySocket.connected) {
        mySocket.send(messageObj);
        // 清除 "send message" 文本域
        outgoing = '';
    } else {
        // 服务器必须断开……
        incoming += "You are no longer connected. Please reconnect.\n";
        incomingUpdated = true;
    }
}

// *** 关闭到服务器的连接
function quit() {
    if (mySocket.connected) {
        mySocket.close();
        mySocket.connected = false;
        numClients = 0;
        incoming = "";
        gotoAndStop('login');
    }
}
```

**参见**

*loadVariables()*, XML 类

---

**XMLSocket.close()方法**

建立一个到服务器应用程序的开放连接

**有效性** Flash 5

**概要** `socket.close()`

**描述**

*close()*方法提供 *socket* 和服务器应用程序之间的连接。一旦 *close()* 在 *socket* 上执行，以后再试图在 *socket* 上调用 *send()* 就会失败。同样，服务器应用程序不能再通过 *socket* 将数据发送到 Flash。

注意，如果 *socket* 已经关闭或者还没有建立，那么 *close()* 就是无效的。此外，*close()* 不触发接口对象的 *onClose()* 处理器——*onClose()* 只被服务器端的连接关闭所触发。

**参见**

*XMLSocket.connect()*, *XMLSocket.onClose()*

---

**XMLSocket.connect()方法**

打开一个到服务器应用程序的连接

**有效性** Flash 5

**概要** `socket.connect(host, port)`

**参数**

*host* 一个串，指定主机名称（比如 "www.myserver.com"）或一个标准的 IP 地址（四个由点分隔的，8 比特位的十进制整数，比如 111.222.3.123）。如果指定的是一个空串或者 *null*，就默认认为是提供影片的服务器地址。

*port* 一个整数，指定一个 TCP 端口号，大于或者等于 1024。

**返回**

一个布尔值，表示连接的初始化成功 (*true*) 或失败 (*false*)。

**描述**

*connect()* 方法试图建立一个从 Flash 到运行在 *port* 端口 *host* 上的服务器应用程序的连接。

如果 `connect()` 返回 `true`, 那么连接的初始化就成功完成, `socket` 的 `onConnect()` 处理器会在这之后被调用。通过 `onConnect()` 处理器, 我们可以判断连接是否完全建立。注意, 建立连接所需要的时间是不同的, 特别是当连接失败的时候。你应该在调用 `connect()` 的时候告诉用户, 连接正在进行中。

如果 `connect()` 返回 `false`, 那么初始化连接装载就没有成功完成。在这种情况下, `socket` 的 `onConnect()` 处理器就不会被调用。

检查 `connect()` 方法的返回值, 以及在 `connect()` 返回 `true` 的时候检查 `onConnect()` 处理器的 `success` 参数值是很重要的。

### 用法

出于安全上的考虑, `connect()` 不能连接到任意的 Internet 主机上。它只可以连接到从其中下载影片的主机。`connect()` 的规则和应用于 `loadVariables()` 函数的规则是一样的。参见全局函数 `loadVariables()` 下的表 R-8, 它给出了一个匹配 `connect()` 方法需求的范围列表。`connect()` 方法对违反安全限制的连接会返回 `false`。注意, 安全限制不作用在独立的播放器上。

### 示例

```
// 创建一个新的接口对象
mySocket = new XMLSocket();
// 将一个复查处理器函数赋给onConnect
mySocket.onConnect = handleConnect;
// 连接到一个运行在端口 10000 上的 myserver.com 上的应用程序
if (mySocket.connect("myserver.com", 10000) == false) {
    // 跳到一个显示错误消息部分的帧
    gotoAndStop("failureDialog");
} else {
    // 跳到一个我们要等待onConnect 被触发的帧
    gotoAndPlay("connecting");
}
```

### 参见

`XMLSocket.close()`, `XMLSocket.onConnect()`

---

## XMLSocket.onClose()事件处理器

确定服务器关闭连接的时候所调用的复查处理器

**有效性**

Flash 5

**概要**

```
socket.onClose=closeHandler  
socket.closeHandler()
```

## 描述

*onClose* 属性可以指定一个复查处理器，在 *socket* 的开放连接被服务器关闭的时候自动执行。服务器的关闭通常是由服务器应用程序被关闭，或者故意地“断开”客户机所致。

## 示例

要对 *onClose* 事件作出反应，我们必须将自己的函数（也就是复查处理器）赋给 *XMLSocket* 对象的 *onClose* 属性。实际上，我们用这个复查处理器来探测一个外部的接口连接。下面的代码将函数 *handleClose()* 赋给 *mySocket* 的 *onClose* 属性。*handleClose()* 函数只警告用户由更新文本域 *status* 的值而引发的关闭动作：

```
mySocket = new XMLSocket();
mySocket.onClose = handleClose;

function handleClose() {
    status += ("\nThe server has terminated the connection.\n");
}
```

## 参见

*XMLSocket.close()*, 第十章

---

## XMLSocket.onconnect()事件处理器

定义一个事件处理器，在连接完成  
(成功或不成功) 的时候调用

**有效性** Flash 5

**概要** *socket.onConnect=connectHandler*  
*socket.connectHandler(success)*

**参数**

*success* 一个布尔值，表示连接是成功 (*true*) 还是失败 (*false*)。

## 描述

*onConnect* 属性可以指定一个复查处理器，在前面的 *connect()* 调用操作结束之后自动执行。*onConnect* 所指定的复查处理器的执行不一定表示一个连接已经成功建立——复查处理器在连接完成的时候就会执行，不管连接是否成功。*onConnect* 所指定的复查处理器有一个 *success* 参数，表示连接是否成功（也就是一个连接被建立）。如果成功，*success* 就被设置为 *true*；如果连接失败（也就是被中断、被拒绝，或者因为其他某种原因没有建立），那么 *success* 参数就被设置为 *false*。注意，

ActionScript 对于网络中断、不知名主机、或其他连接错误不加区分，因此，复查处理器依靠连接试图、连接速度、网络交通等因素中所包含的服务器设置，可能无法在 *connect()* 命令开始之后一分钟内得到执行。

## 示例

我们用 *onConnect* 指定的复查处理器来探测连接的成功或失败。实际上，我们可以用复查处理器来设置一个标识，表示如果连接成功就开始传输。我们也可以用复查处理器在连接失败的时候执行复查代码，比如警告用户问题的种类。

要对 *onConnect* 事件作出反应，我们必须将自己的函数（也就是复查处理器）赋给 *XMLSocket* 对象的 *onConnect* 属性。下面的代码将函数 *handleConnect()* 赋给 *mySocket* 的 *onConnect* 属性。通过更新文本域 *status* 的值，*handleConnect()* 告诉用户连接是否成功：

```
mySocket = new XMLSocket();
mySocket.onConnect = handleConnect;

function handleConnect (succeeded) {
    if (succeeded) {
        status += ('Successfully connected.\n');
    } else {
        status += ('Connection attempt failed.\n');
    }
}
```

显示应用在更多完整系统中的 *onConnect()* 处理器的代码，请参见 *XMLSocket* 类下的例子。

## 参见

*XMLSocket.connect()*, 第十章

---

## XMLSocket.onData()事件处理器

当外部数据被接收到，但是还没有作为 XML 来解析的时候执行

**有效性** Flash 5 (非正式)

**概要** `socket.onData(src)`

### 参数

*src* 一个串，包含装载数据，通常是 XML 源代码。

## 描述

*onData()* 处理器会在零字节 (ASCII 的 null 字符) 被传送到 *socket* 端的 Flash 时自动执行。在默认情况下, *onData()* 只从 *src* 构建一个新的 XML 对象层次, 并将这个层次传递给 *socket.onXML()*。但是, *onData()* 处理器可以设置自定义的复查函数, 以在 ActionScript 有机会将 *src* 作为 XML 进行解析之前半路截获它。在特定的环境下 (比如实时视频游戏), 手动操作 *src* 中的原始数据可以提供比 ActionScript 的内置解析操作更高的效率。

## 示例

下面的代码显示了如何将自定义的复查函数赋给 *onData()*。复查函数只显示 *mySocket* 所接收到的数据, 并且防止 ActionScript 将数据解析为 XML:

```
mySocket = new XMLSocket();  
  
mySocket.onData = function (src) {  
    trace("Received data: " + src);  
};
```

## 参见

*XMLSocket.onXML()*

---

## XMLSocket.onXML()事件处理器

定义一个复查处理器, 在数据被 *XMLSocket* 对象接收到, 并且被解析为 XML 的时候调用

**有效性** Flash 5

**概要**  
*socket.onXML = xmlHandler*  
*socket.xmlHandler(XMLObject)*

## 参数

*XMLObject* XML 对象, 将容纳引入的 XML 格式数据。

## 描述

*onXML* 属性可以指定一个复查处理器, 在 Flash 接收到传输的时候得到执行。当 *socket* 从服务器接收到一个完整的数据块 (也就是一个后面跟着 ASCII 的 null 字符的串) 时, *socket.onXML* 所指定的复查处理器就被自动调用。服务器发送数据的速度是任意的, 但是复查处理器只在跟踪 *socket* 所接收的 null 字符 (也就是零字节) 的时候可以执行。在 Java 中, 一个零字节被指定为 '\0'。当零字节被接收到的时候, 它

会让 ActionScript 在最后一个零字节被发送之后（如果这是第一个零字节，就是在初始化连接之后）将 *socket* 接收到的任何数据进行解析。解析的数据被转换为 XML 对象层次，作为 *XMLObject* 的参数被传输给复查处理器。

如果你是一个 Flash 程序员，负责客户/服务器应用程序中的客户端，那么只需要注意，*onXML* 所执行的复查处理器会接收任何新到的 XML 数据。可以通过 *XMLObject* 访问新的 XML 数据。

要访问通过接口发送的原始数据，需要覆盖接口的 *onData()* 处理器的默认行为。参见 *XMLSocket.onData()*。

### 示例

要对 *onXML* 事件作出响应，必须将我们的函数（也就是复查处理器）赋给 *XMLSocket* 对象的 *onXML* 属性。下面的代码将把函数 *handleIncoming()* 赋给 *mySocket* 的 *onXML* 属性。*handleIncoming()* 函数访问存储在 *messageObj* 中的 XML 对象层次中的一个节点，并将它的值添加到文本域 *message*：

```
mySocket = new XMLSocket();
mySocket.onXML = handleIncoming;

function handleIncoming (messageObj) {
    trace('Got some new data! ');
    // messageObj 会包含标签: <MESSAGE><text>...</text></MESSAGE>
    var message = messageObj.firstChild.firstChild;
    message += (message.nodeValue + '\n');
}
```

显示使用在更完整系统中的 *onXML* 处理器的代码，参见 *XMLSocket* 类下面的例子。

### 参见

*XMLSocket.send*, *XMLSocket.onData()*, 第十章

---

## XMLSocket.send()方法

将 XML 格式的数据传输到服务器应用程序

**有效性** Flash 5

**概要** *socket.send(XMLObject)*

**参数**

*XMLObject* 一个被转换为串，然后被发送到服务器应用程序的 XML 对象，或者任何包含 XML 格式文本的串。

## 描述

*send()*方法通过 *socket* 将一个消息从 Flash 发送到服务器应用程序。要发送的消息应该是 *XML* 类的对象而不是一个串。当 *send()* 被调用的时候，*XMLObject* 被转换为串，并且发送到远程应用程序，后面跟着一个零字节（第一个 ASCII 字符，*null*）。远程应用程序不要求响应，但是，任何响应都将触发 *socket* 的 *onXML()* 事件处理器。

## 示例

下面的代码将一个非常简单的 XML 格式消息发送给接口 *mySocket* 端的远程应用程序，*mySocket* 是一个合法的 *XMLSocket* 对象，已经建立了连接（注意，*message* 是一个 *XML* 对象，而不是一个 *XMLSocket* 对象，参见 *XMLSocket* 类条目下的例子，可以得到完整的 *XMLSocket* 示范应用程序）：

```
var message = new XML('<MESSAGE>testing...testing...</MESSAGE>');
mySocket.send(message);
```

发送包含 XML 格式文本的串，但是不将它封装到 XML 对象中也是合法的。对于简单的 XML 消息，下面这种形式通常已经足够了：

```
mySocket.send('<MESSAGE>testing...testing...</MESSAGE>');
```

## 参见

*XMLSocket.onXML()*, *XMLSocket* 类, *XML.send()*

---

# 第四部分

## 附录

- 附录一，资源
- 附录二，Latin1 指令表和键控代码
- 附录三，向后兼容
- 附录四，ECMA-262 和 JavaScript 之间的差别



# 附录一

# 资源

下面是一些有趣的资源。也可以参见前言中所引用的 URL。

## ActionScript 和编程

### 本书站点

<http://www.moock.org/asdg>

这是本书中所有范例的下载中心(代码库)，而且还有其他数十个普通 ActionScript 范例。该站点还提供了书籍更新、技术、示例章节、勘误表和消息。主要由作者来维护。

### moockmarks

<http://www.moock.org/moockmarks>

作者的书签。列出了几百个关于 Flash 和一般的 Web 设计的技术信息和设计灵感的站点。

### Macromedia 的 Flash 支持中心

<http://www.macromedia.com/support/flash>

收集了丰富的 ActionScript 和 Flash 信息。包括关于 Macromedia 的 ActionScript 词典的所有内容，以及针对 Flash 开发者的正规更新技术数据库。可以从：<http://www.macromedia.com/software/flash/trial> 下载一个关于 Flash 制作工具的测试版本。

*Macromedia 的 Flash 交流站点*

<http://www.macromedia.com/exchange/flash>

收集了专业水平的智能剪辑、.fla 设计文件，以及 ActionScript 范例。内容由 Macromedia 拥有和认可，但是是由 Flash 同仁共同开发的。要访问范例需要一种所谓的扩展管理的安装功能。

*Flash 工具包*

<http://www.flashkit.com>

Internet.com 的 Flash 工具包是关于 Flash 帮助的最大的第三方资源。站点中有关于 Flash 的一切内容：范例文件、留言板、指南、消息、功能文章、会面、聊天、图库等等。让 Flash 编程人员最感兴趣的是 ActionScript 范例.fla 文件，可以从 <http://www.flashkit.com/movies/Scripting> 上下载。

*Ultrashock*

<http://www.ultrashock.com>

对 Flash 开发者的一个共同入口，Ultrashock 拥有丰富的 ActionScript 中心.fla 文件和指南。

*Flash 编程者邮件列表*

<http://chattyfig.figleaf.com/mailman/listinfo/flashcoders>

由 Fig Leaf 软件公司的 Branden Hall 拥有。这个精力旺盛的 Flash 开发者的邮件列表将焦点主要放在高级 ActionScript 的媒介上。该站点是一个学习实际技术的好地方，还有一些由专家回答的问题，但是，它不适合普通的 Flash 问题和站点检查。它的存档文件是可以访问的，每月都会张贴出来。在你粘贴问题之前可以先检查存档文件。

*Brandon Williams 和 Ethan Kennedy 的 Flash 经验*

[http://www.homepages.go.com/~ahab\\_flash/exper/index.htm](http://www.homepages.go.com/~ahab_flash/exper/index.htm)

收集了众多的高级数学基础 ActionScript 演示。提供了数十个可供学习的.flc 文件，包括 3D、不规则形，以及许多物理视觉效果。还提供了 Flash 中的数学基础编程的具有一定深度的文章。

*图形和运算法则常见问题解答*

<http://www.faqs.org/faqs/graphics/algorithms-faq>

描述了众多图形编程问题的数学解决方式。从诸如“我怎么才能得到点到直线的距离”之类的简单问题到高级 3D 问题，通常都超出了 Flash 编程的领域。

#### 存档的 Macromedia 的 Flash 播放器

<http://www.macromedia.com/support/flash/ts/documents/oldplayers.htm>

下载 Macromedia 的 Flash 播放器旧版本以供测试。

#### 通用游戏编程资源

下面的站点提供了设计、规划和编写游戏的有用信息（它们不是 ActionScript 的专门站点）：

<http://www.gamedev.net>

<http://www.flipcode.com>

<http://www.chesworth.com/pv/games>

<http://www.javascript-games.org>

#### Flash 和活动服务器页面(ASP)

下面的条目来自 ASP 101，提供了关于将 Flash 与 ASP 结合，以创建简单 Flash 表单的精彩介绍：

<http://www.asp101.com/articles/flash>

<http://www.asp101.com/articles/flash2>

#### Macromedia 发生器

虽然讨论 Macromedia 发生器已经超出了本书的范围，但是下面的 Web 站点提供了出色的发生器开发资源：

<http://www.markme.com>

<http://www.gendev.net>

## ECMA-262 资源

因为 ActionScript 和 JavaScript 都是以 ECMA-262 为基础的脚本语言，它们有一个共同的核心语法。因此，虽然下面的资源并没有将焦点专门集中在 ActionScript 上，但是，对于 ActionScript 编程人员来说它们非常值得一读：

### ECMA-262 语言规范

<http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM>

来自 ECMA 的正规 ECMA-262 语言规范，ActionScript 所基于的语言主体。Netscape 的 JavaScript 和 Microsoft 的 Jscript 是 ECMA-262 的另外两个表现形式。这是一个高级技术文档，用来创建诸如规则 ECMA 解释程序这样的东西。

### Netscape 的核心 JavaScript 指南

<http://developer.netscape.com/docs/manuals/js/core/jsguide/contents.htm>

Netscape 提供了易读的 JavaScript 文档，有 JavaScript 的核心语言功能的详细解释。

## 面向对象编程

### Netscape 的对象模版细节

<http://developer.netscape.com/docs/manuals/js/core/jsguide/obj2.htm>

一篇详细解释 OOP 以原型为基础的实现、继承和 JavaScript 中的类的文章。其概念是用很多代码范例来说明的。为了让 C 和 Java 编码者能够明白，Netscape 在解释 JavaScript 的 OOP 时做了特别的努力。

### Sun 的面向对象编程概念（来自 Java 技术手册指南）

<http://java.sun.com/docs/books/tutorial/java/concepts>

虽然本来打算作为热情的 Java 编程者的初级读本，但它包含了 OOP 的一般形式基础，而且为任何语言中的 OOP 阅读打下了良好的基础。

## SWF 文件格式

尽管 Flash 播放器没有公开的资源，但是 .swf 文件格式本身是可以访问的（这是它获得了许多第三方支持的一个原因）。关于更多的信息，参见下面的资源站点：

### Macromedia Flash 播放器文件格式(SWF)

<http://www.macromedia.com/software/flash/open/licensing/fileformat>

你可以从 Macromedia 获得 SWF 的软件开发工具包——一系列让开发者编写 Macromedia 的 Flash 5 (SWF) 文件的工具，关于 Macromedia Flash 文件格式(SWF)的文档、以及编写 SWF 文件的代码。

#### *OpenSWF*

<http://www.openswf.org>

一个针对 SWF 实际开发者的普通信息中心。包括指南、留言板、资源和源代码。

#### *Ming*

<http://www.opaque.net/ming>

Ming 是一个公开的 C 源代码库，针对 SWF 格式影片的产生，以及一系列使用 C++ 和像 PHP，Python 和 Ruby 这些流行脚本语言所制作的东西。参见 <http://www.opensource.org>，可以得到公共资源软件的信息。

#### *Perl::Flash*

<http://www.2shortplanks.com/flash>

一个 Perl 库，允许 Flash/SWF 影片动态地、程序化地生成。由 Simon Wistow 创建。

---

## 附录二

# Latin1 字符指令表 和键控代码

表B-1列出了Latin1字符指令表中的字符，Flash所支持的基本字符集。第一列（标签为Dec）给了每一个字符的十进制编码点（标准ASCII值），第二列提供了字符的Unicode转义序列，而第三行描述了字符本身。参见第四章可以得到关于Flash中字符编码的更多信息。

关于字符编码的其他资料，参见下面的资源：

#### *ISO 8859 字母表*

<http://czyborra.com/charsets/iso8859.html>

一系列文档，详细描述Latin1字符指令表中字符的结构和含意，以及Flash所支持的基本字符集（由Roman Czyborra维护）。

#### *Shift-JIS 编码点*

<ftp://ftp.unicode.org/Public/MAPPINGS/EASTASIA/JIS/SHIFTJIS.TXT>

一个Shift-JIS字符集中字符的Unicode编码点列表，Flash的日本字符支持集。

#### *Unicode 常见问题解答*

<http://www.unicode.org/unicode/faq>

Unicode - 一个很好的问题解答格式浏览，Unicode是一种字符编码的国际标准。

表 B-1 ISO 8859-1(Latin1)字符和 Unicode 映射

Dec	Unicode	描述	Dec	Unicode	描述
0	\u0000	[null]	36	\u0024	\$
1	\u0001	[标题开始]	37	\u0025	%
2	\u0002	[文本开始]	38	\u0026	&
3	\u0003	[文本结束]	39	\u0027	'
4	\u0004	[传输结束]	40	\u0028	(
5	\u0005	[询问]	41	\u0029	)
6	\u0006	[承认]	42	\u002a	*
7	\u0007	[铃]	43	\u002b	+
8	\u0008	[退格]	44	\u002c	,
9	\u0009	[水平表格]	45	\u002d	-
10	\u000a	[线]	46	\u002e	.
11	\u000b	[垂直表格]	47	\u002f	/
12	\u000c	[表单]	48	\u0030	0
13	\u000d	[段落]	49	\u0031	1
14	\u000e	[移出]	50	\u0032	2
15	\u000f	[移入]	51	\u0033	3
16	\u0010	[数据链接溢出]	52	\u0034	4
17	\u0011	[设备控制--]	53	\u0035	5
18	\u0012	[设备控制二]	54	\u0036	6
19	\u0013	[设备控制三]	55	\u0037	7
20	\u0014	[设备控制四]	56	\u0038	8
21	\u0015	[拒绝承认]	57	\u0039	9
22	\u0016	[同步空闲]	58	\u003a	:
23	\u0017	[传输块结束]	59	\u003b	;
24	\u0018	[取消]	60	\u003c	<
25	\u0019	[媒介结束]	61	\u003d	=
26	\u001a	[代替]	62	\u003e	>
27	\u001b	[溢出]	63	\u003f	?
28	\u001c	[文件分隔符]	64	\u0040	@
29	\u001d	[组分隔符]	65	\u0041	A
30	\u001e	[记录分隔符]	66	\u0042	B
31	\u001f	[单元分隔符]	67	\u0043	C
32	\u0020	[空格]	68	\u0044	D
33	\u0021	!	69	\u0045	E
34	\u0022	"	70	\u0046	F
35	\u0023	#	71	\u0047	G

表 B-1 ISO 8859-1(Latin1)字符和 Unicode 映射(续)

Dec	Unicode	描述	Dec	Unicode	描述
72	\u0048	H	108	\u006c	l
73	\u0049	I	109	\u006d	m
74	\u004a	J	110	\u006e	n
75	\u004b	K	111	\u006f	o
76	\u004c	L	112	\u0070	p
77	\u004d	M	113	\u0071	q
78	\u004e	N	114	\u0072	r
79	\u004f	O	115	\u0073	s
80	\u0050	P	116	\u0074	t
81	\u0051	Q	117	\u0075	u
82	\u0052	R	118	\u0076	v
83	\u0053	S	119	\u0077	w
84	\u0054	T	120	\u0078	x
85	\u0055	U	121	\u0079	y
86	\u0056	V	122	\u007a	z
87	\u0057	W	123	\u007b	{
88	\u0058	X	124	\u007c	
89	\u0059	Y	125	\u007d	}
90	\u005a	Z	126	\u007e	~
91	\u005b	l	127	\u007f	[删除]
92	\u005c	\`	128	\u0080	控制字符
93	\u005d	¡	129	\u0081	控制字符
94	\u005e	^	130	\u0082	控制字符
95	\u005f	—	131	\u0083	控制字符
96	\u0060	‘	132	\u0084	控制字符
97	\u0061	a	133	\u0085	控制字符
98	\u0062	b	134	\u0086	控制字符
99	\u0063	c	135	\u0087	控制字符
100	\u0064	d	136	\u0088	控制字符
101	\u0065	e	137	\u0089	控制字符
102	\u0066	f	138	\u008a	控制字符
103	\u0067	g	139	\u008b	控制字符
104	\u0068	h	140	\u008c	控制字符
105	\u0069	i	141	\u008d	控制字符
106	\u006a	j	142	\u008e	控制字符
107	\u006b	k	143	\u008f	控制字符

表 8-1 ISO 8859-1(Latin1)字符和 Unicode 映射(续)

Dec	Unicode	描述	Dec	Unicode	描述
144	\u0090	控制字符	180	\u00b4	'
145	\u0091	控制字符	181	\u00b5	μ
146	\u0092	控制字符	182	\u00b6	¶
147	\u0093	控制字符	183	\u00b7	.
148	\u0094	控制字符	184	\u00b8	:
149	\u0095	控制字符	185	\u00b9	1
150	\u0096	控制字符	186	\u00ba	ə
151	\u0097	控制字符	187	\u00bb	»
152	\u0098	控制字符	188	\u00bc	¼
153	\u0099	控制字符	189	\u00bd	½
154	\u009a	控制字符	190	\u00be	¾
155	\u009b	控制字符	191	\u00bf	ѣ
156	\u009c	控制字符	192	\u00c0	À
157	\u009d	控制字符	193	\u00c1	Á
158	\u009e	控制字符	194	\u00c2	Â
159	\u009f	控制字符	195	\u00c3	Ã
160	\u00a0	[没有间断空格]	196	\u00c4	Ä
161	\u00a1	¡	197	\u00c5	Å
162	\u00a2	¢	198	\u00c6	Æ
163	\u00a3	£	199	\u00c7	Ҫ
164	\u00a4	¤	200	\u00c8	È
165	\u00a5	¥	201	\u00c9	É
166	\u00a6	߱	202	\u00ca	Ê
167	\u00a7	߳	203	\u00cb	Ë
168	\u00a8	ߴ	204	\u00cc	ି
169	\u00a9	ߵ	205	\u00cd	ି
170	\u00aa	߶	206	\u00ce	ି
171	\u00ab	߷	207	\u00cf	ି
172	\u00ac	߸	208	\u00d0	ଽ
173	\u00ad	߹	209	\u00d1	ି
174	\u00ae	ߺ	210	\u00d2	ଓ
175	\u00af	߻	211	\u00d3	ଓ
176	\u00b0	߻	212	\u00d4	ଓ
177	\u00b1	߻	213	\u00d5	ଓ
178	\u00b2	߻	214	\u00d6	ଓ
179	\u00b3	߻	215	\u00d7	ࡼ

表 B-1 ISO 8859-1(Latin1)字符和 Unicode 映射 (续)

Dec	Unicode	描述	Dec	Unicode	描述
216	\u00d8	ø	236	\u00ec	í
217	\u00d9	Ù	237	\u00ed	í
218	\u00da	Ú	238	\u00ee	í
219	\u00db	Û	239	\u00ef	í
220	\u00dc	Ü	240	\u00f0	ð
221	\u00dd	Ý	241	\u00f1	ñ
222	\u00de	þ	242	\u00f2	ò
223	\u00df	ß	243	\u00f3	ó
224	\u00e0	à	244	\u00f4	ô
225	\u00e1	á	245	\u00f5	ô
226	\u00e2	â	246	\u00f6	ô
227	\u00e3	ã	247	\u00f7	÷
228	\u00e4	ä	248	\u00f8	ø
229	\u00e5	å	249	\u00f9	ù
230	\u00e6	æ	250	\u00fa	ú
231	\u00e7	ç	251	\u00fb	û
232	\u00e8	è	252	\u00fc	ü
233	\u00e9	é	253	\u00fd	ý
234	\u00ea	ê	254	\u00fe	þ
235	\u00eb	ë	255	\u00ff	ÿ

表B-2列出了键盘中所选特殊键的键控代码。这些键控代码只用在Key对象上(参见第三部分)。

表 B-2 特殊键控代码的 Key 对象

Key	键控代码	Key	键控代码
;	186	Caps Lock	20
+=	187	Control	17
-~	189	Delete	46
/?	191	Down arrow	40
`~	192	End	35
{`	219	Enter	13
\	220	Escape	27
}`	221	F1	112
“”	222	F2	113

表 B-2 特殊键控代码的 Key 对象

Key	键控代码	Key	键控代码
Alt	18 (不能捕捉)	F3	114
Backspace	8	F4	115
F5	116	数字键盘 7	103
F6	117	数字键盘 8	104
F7	118	数字键盘 9	105
F8	119	数字键盘 *	106
F9	120	数字键盘 +	107
F10	121 (不能捕捉)	数字键盘回车	13 (作为 108 不能捕捉)
F11	122	数字键盘 -	109
F12	123	数字键盘 .	110
Home	36	数字键盘 /	111
Iinsert	45	下一页	34
左箭头	37	上一页	33
Num Lock	144	Pause/Break	19
数字键盘 0	96	Print Screen	44
数字键盘 1	97	右箭头	39
数字键盘 2	98	Scroll Lock	145
数字键盘 3	99	Shift	16
数字键盘 4	100	空格	32
数字键盘 5	101	Tab	9
数字键盘 6	102	上箭头	38

表 B-3 列出了键盘上数字和字母的键控代码。这些键控代码只用在 Key 对象上 (参见第三部分)。

表 B-3 Key 对象的字母和数字键盘代码

Key	键控代码	Key	键控代码	Key	键控代码
A	65	M	77	Y	89
B	66	N	78	Z	90
C	67	O	79	0	48
D	68	P	80	1	49
E	69	Q	81	2	50
F	70	R	82	3	51
G	71	S	83	4	52

表 B-3 Key 对象的字母和数字键盘代码（续）

Key	键控代码	Key	键控代码	Key	键控代码
H	72	T	84	5	53
I	73	U	85	6	54
J	74	V	86	7	55
K	75	W	87	8	56
L	76	X	88	9	57

---

## 附录三

# 向后兼容

Flash允许你按照和播放器以前的版本相兼容的格式发布.swf文件。当你读这本书的时候，大部分用户所使用的Flash插件至少已经是版本5的了，但是，你可以选择使用Flash 4插件。即使你专门使用Flash 5，本附录也可以帮助你避免使用已经淘汰的ActionScript，让你跟上Flash 5中首选的方法。关于各种Flash播放器版本的统计，参见：

<http://www.macromedia.com/software/flash/survey/whitepaper>

为Flash 4制作的时候，可以使用不支持的技术（也就是那些已经过时，但是在向后兼容中仍然可以得到支持的技术），它们都列在表C-1中。参见Macromedia的Flash ActionScript Reference Guide中Writing Scripts with Action Script（用ActionScript编写脚本）下面的Using Flash 5 to Create Flash 4 Content（用Flash 5来创建Flash 4内容）。

---

**警告：**要在Flash 4播放器上运行脚本，必须将你的.swf文件版本设置为Flash 4，在file→Publish Settings下面的Flash标签就可以设置。不能试图在Flash 4播放器上运行Flash 5或更新的.swf文件。

---

表C-1给出了向后兼容关键问题的摘要，以及Flash 4 ActionScript和Flash 5 ActionScript之间的差别。

表 C-1 向后兼容问题

主题	描述
创建变量	Flash 4 的 <i>set</i> 函数已经被 <i>var</i> 语句所代替。要动态地创建命名变量，可以使用 <i>eval()</i> 或者（更恰当）使用数组来管理数据。参见第十一章
变量和时间线引用	Flash 4 风格的斜线——冒号构造（/square:area）已经被点操作符所代替（square.area）。参见表 2-1
串比较操作符	Flash 4 的串比较操作符 —— <i>eq</i> , <i>ne</i> , <i>ge</i> , <i>gt</i> , <i>le</i> , <i>lt</i> —— 已经被以下 Flash 5 中的操作符所代替：==, !=, >=, >, <=, <。参见表 4-2
串连接操作符	在 Flash 5 及其以后的版本中创建 Flash 4 内容的时候，用 <i>add</i> 操作符来代替 Flash 4 的 & 操作符。制作 Flash 5 作品的时候，用 + 操作符来进行串的连接。参见表 4-2
串长度	Flash 4 的 <i>length()</i> 函数（也就是 <i>length(myString)</i> ）已经被 <i>length</i> 属性所替代（例如， <i>myString.length</i> ），参见表 4-2
子串的提取	Flash 4 的 <i>substring()</i> 函数（例如 <i>substring(myString,1,3)</i> ）已经被 <i>substring()</i> , <i>substr()</i> 和 <i>slice()</i> 方法所代替。注意， <i>substring()</i> 在 Flash 4 和 Flash 5 中是不同的。参见表 4-2
字符编码点函数	Flash 4 的 <i>chr()</i> 和 <i>mbchr()</i> 函数（用来从编码点创建字符）已经被 <i>String.fromCharCode()</i> 所代替。Flash 4 的 <i>ord()</i> 和 <i>mbord()</i> 函数（用来确定字符的编码点）已经被 <i>String.charCodeAt()</i> 方法所代替。参见表 4-2
数据类型转换	导入 Flash 4 文件的时候，Flash 5 会自动插入 <i>Number()</i> 函数，将用做下面操作符的潜在操作数的数字数据进行舍入：+, ==, !=, <>, <, >, >=, <=。参见表 3-5
<i>ifFrameLoaded</i> 语句	Flash 3 的 <i>ifFrameLoaded</i> 语句已经不被支持了。使用 <i>_totalframes</i> 和 <i>_framesloaded</i> 这两种 <i>MovieClip</i> 属性来创建预下载代码
无限循环	Flash 4 允许 200,000 次最大循环次数。Flash 5 允许 15 秒钟的循环，然后就会警告用户影片已停止响应。参见第八章
子程序和函数	在 Flash 4 中，可以通过将代码块添加到帧，并附加标签而创建子程序，然后，可以通过 <i>call()</i> 语句来执行。Flash 5 的函数代替了 Flash 4 的子程序
剪辑事件	Flash 4 只支持按钮事件（也就是从 <i>on()</i> 开始的函数），如表 10-1 所示。剪辑事件（也就是 <i>onClipEvent()</i> ）不能用在 Flash 4 播放器中

表 C-1 向后兼容问题 (续)

主题	描述
捕获击键动作	在 Flash 4 中, <i>keyPress</i> 是捕获击键动作的惟一途径。Flash 5 的 <i>Key</i> 对象和影片剪辑事件 <i>keyDown</i> 和 <i>keyUp</i> 结合在一起, 可以提供对键盘交互的更强控制
不被支持的 <i>Tell Target</i>	Flash 4 的 <i>Tell Target</i> (用来控制远程影片剪辑) 已经被用点操作符和 <i>with</i> 语句来访问的方法和属性所替代。参见第十三章
不被支持的 <i>Get Property</i>	Flash 4 的 <i>Get Property</i> 命令已不用于对影片剪辑属性的访问, 取而代之的是点操作符。参见第十三章
不被支持的 <i>int</i>	Flash 4 的 <i>int()</i> 函数 (用来将浮点数转换为整数) 已经被 <i>Math.floor()</i> , <i>Math.ceil()</i> 和 <i>Math.round()</i> 所代替
随机数字的产生	Flash 4 的 <i>random()</i> 函数 (用来产生随机数字) 已经被 <i>Math.random()</i> 所代替
不被支持的 <i>toggleHighQuality</i>	Flash 4 的 <i>toggleHighQuality</i> 函数 (用来设置播放器的渲染效果) 已经被全局属性 <i>_quality</i> 所代替
不被支持的 <i>_highquality</i>	Flash 4 的 <i>_highquality</i> 属性已经被 <i>_quality</i> 全局属性所代替
Flash 4 中支持的 Math 对象	Math 对象的函数和属性 (例如 <i>Math.cos()</i> , <i>Math.PI</i> ) 并不是 Flash 4 播放器本来所支持的。但是, 当影片按照 Flash 4 格式导出的时候, 值要进行约计
<i>loadMovie</i> 和 <i>loadMovieNum</i>	Flash 3 中作用于编号层次上的 <i>loadMovie()</i> 已经被 Flash 5 的 <i>loadMovieNum()</i> (它接收一个整数层级参数) 所代替了。Flash 4 的目标影片剪辑中的 <i>loadMovie()</i> 在 Flash 5 中仍然可以作为 <i>loadMovie()</i> 来使用
打印	Flash 5 支持 <i>print()</i> 函数, 它在 Flash 4 修订版 20 和之后的版本中只作为一个修改的 <i>Get URL</i> 动作
不被支持的对象 和类	Flash 4 不支持任何 Flash 5 的内置对象和类

## 更新到 Flash 5 播放器的修订版 41

下面的列表总结了 Flash 5 播放器修订版 41 (Netscape) 和 42 (IE) 中所发生的变化 (前一个发布修订版为 30, 最初和 Flash 5 制作工具一起发布):

- I型光标具有文本的颜色。
- 表格单元中的影片不会让IE 5.5崩溃。
- 文本域的 scroll 位置不会在文本域内容被修改的时候重新设置。
- 有嵌入字体的文本域在滚动的时候保持视觉上的分界。
- 添加了 XML.contentType 属性。
- 添加了 XML.ignoreWhite 属性。
- 当 XML 源代码被解析，并且字符 &、'、"、< 和 > 出现在文本节点中的时候，它们会被转换为下面的东西：&amp;、&apos;、&quot;、&lt; 和 &gt;。这个转换在 Flash 中是很明显的，因为当 XML 对象被转换为串的时候，它们就重新变成了字符；但是，它们将会出现在发送给服务器的源代码中。
- Math.random() 不返回值 1。它的最大返回值为 0.999。
- 一般性能被提高了，特别是在 Window 98 中。

## 控制影片剪辑

在第十三章中，我们学习了如何用 Flash 5 技术来控制剪辑。下面我们介绍 Flash 4 中作用相同技术。

在 Flash 5 之前，我们可以执行特殊的影片剪辑动作来控制影片剪辑。我们可以用下面的语言表示“让剪辑 eyes 播放”：

```
Begin Tell Target ('eyes')
    Play
End Tell Target
```

但是在 Flash 5 中，通过内置方法可以更直接地控制影片剪辑。例如：

```
eyes.play();
```

同样，要访问 Flash 5 之前的影片剪辑内置属性，可以用显式的属性设置和属性设置命令，比如：

```
GetProperty ('ball', _width)
```

```
Set Property ("ball", X Scale) = 90
```

在 Flash 5 中，可以用点操作符来获取和设置影片剪辑的属性，正如我们可以访问任何对象的属性一样：

```
ball._width;  
ball._xscale = 90;
```

在 Flash 5 之前，要访问影片剪辑中的变量，我们要用冒号来分隔剪辑名称和变量名称：

```
Set Variable: "x" - myClip:myVariable
```

在 Flash 5 中，影片剪辑中的一个变量就是剪辑对象的一个简单属性，因此我们现在使用点操作符来设置和获取变量的值：

```
myClip.myVariable = 14;  
x = myClip.myVariable;
```

最后，在 Flash 5 之前，我们可以访问影片剪辑的嵌套层次，用斜杠和点的类似家族树来实现：

```
clipA/clipB/clipC  
...../clipC
```

因为影片剪辑是 Flash 5 中类似对象的数据，我们可以将剪辑存储为另外一个剪辑的属性。因此，可以用点操作符来访问所谓的嵌套剪辑，我们使用剪辑的保留 `_parent` 属性来指向包含它的剪辑：

```
clipA.clipB.clipC;  
_parent._parent._parent.clipC;
```

## 附录四

# ECMA-262 和 JavaScript 之间的差别

本书本来的焦点是放在 ActionScript 的内容上，但是，如果你拥有 O'Reilly 的优秀书籍《JavaScript: The Definitive Guide》，你会注意到两本书的参考部分非常相似。

虽然 ActionScript 像 JavaScript 一样基于 ECMA-262 的标准，但它们的播放器尺寸和向后兼容显然是有差别的。如果从 JavaScript, Jscript 或其他基于 ECMA-262 的语言中输出代码，你会发现表 D-1 非常有使用价值。它总结了 ECMA-262, JavaScript 和 Flash 5 ActionScript 之间内在的差别。同样，如果从 ActionScript 将代码输出到另外一种语言，你就能很好地避免因为 ActionScript 背离 ECMA-262 标准所造成的缺陷。

表 D-1 反映了 Flash 5 ActionScript 和 ECMA-262 标准之间的差别。它不反映标准实现中所存在的故障。

表 D-1 ECMA-262, JavaScript 和 ActionScript 之间的差别

主题	描述
串到布尔的转换	在 ECMA-262 中，所有的非空串都转化为 true。在 Flash 5 中，只有可以转换为有效非零数字的串才转换为 true
区分大小写	ECMA-262 规范要求完全区分大小写。在 ActionScript 中，关键字是区分大小写的，但是标识符却不是。参见第十四章，特别是表 14-1

表 D-1 ECMA-262, JavaScript 和 ActionScript 之间的差别（续）

主题	描述
函数作用域	当某个时间线上的函数被赋给另外一个影片剪辑时间线上的变量时，被赋予的函数作用域链就转换到变量的时间线上。在 ECMA-262 中，不能通过赋值来修改函数的作用域链，作用域是按照函数声明语句的位置而永久确定的
正规表达式	ActionScript 不支持正规表达式
事件处理器名称	ActionScript 中，只有基于对象的事件处理器能得到它们自己的命名函数（例如， XML 的 <i>onLoad()</i> ）。影片剪辑事件处理器是用 <i>onClipEvent(eventName)</i> 来定义的，按钮事件处理器是用 <i>on(eventName)</i> 来定义的。参见第十章
全局变量	ActionScript 不支持真正的文档范围全局变量。全局变量可以通过将属性添加到 <i>Object.prototype</i> 来模拟，如第十二章中描述的那样
<i>eval()</i> 函数	ActionScript 的 <i>eval()</i> 函数支持 ECMA-262 功能的一个子集，它只在参数是标识符的时候有效，只用来动态地生成对标识符的引用
<i>undefined</i> 数据类型转换	在 ActionScript 中，特殊的 <i>undefined</i> 值在使用于串语境的时候转换为空串（“”），在使用于数字语境的时候转换为数字 0。在 ECMA-262 中， <i>undefined</i> 在串语境中转换为串“ <i>undefined</i> ”，在数字语境中转换为数字值 NaN
<i>Function</i> 构造器	ActionScript 不支持函数构造器，它在 JavaScript 中是用来创建函数的，语法为 <i>new Function()</i> ；
<i>Date</i> 对象的创建	ActionScript 在创建一个新的 <i>Date</i> 对象时，不接收（也就是不解析）我们可以读懂的串，比如“January 9, 2001”
<i>switch</i> 语句	ActionScript 不支持 <i>switch/case/default</i> 语句（用来指定复杂条件）。参见第七章
语言支持	ECMA-262 要求支持 Unicode 字符编码标准，而 ActionScript 却不然。ActionScript 用 Latin1 和 Shift-JIS 字符集，实现 Unicode 类型的函数和转换的一个子集（比如 \u 转义序列）
对象模板	JavaScript 包括和 Web 浏览器有关的内置类和对象，而 Flash 包括和 Flash 影片有关的这些东西。对习惯 DHTML 的 JavaScript 程序员来说，将 Flash 文档的主影片看成 HTML 文档对象的类似物，将影片剪辑看成层对象的类似物可能会有帮助

表 D-1 ECMA-262, JavaScript 和 ActionScript 之间的差别（续）

主题	描述
定时的代码执行	JavaScript 窗口对象的 <code>setTimeout()</code> 和 <code>setInterval()</code> 方法在 ActionScript 中是不可用的，但是可以用第八章中所描述的时间线和剪辑事件循环来模拟
<i>Object</i> 构造器	在 Flash 5 中，ActionScript 的对象类构造器不接收任何参数。而在 ECMA-262 中，对象接收 <code>value</code> 参数，它可以是一个布尔值、一个串，或一个原始数字

# 词汇表

action	data
动作	数据
asynchronous	document level
异步执行	文件层级
bandwidth	encode
带宽	编码
bitwise	event
位逻辑	事件
boolean	event handler
布尔	事件处理器
character set	floating-point number
字符集	浮点数
code point	frame rate
编码点	帧速率
clip event loop	function
剪辑事件循环	函数
collision detection	global scope
冲突检测	全局作用域
composite datatype	HEX
复合数据类型	十六进制

inheritance	primitive datatype
继承	原始数据类型
initialize	recursive function
初始化	递归函数
interactivity	registration point
交互性	记录点
interpreter	reversed word
解释程序	保留字
iterator	RGB color
循环因子	RGB 颜色
keyframe	scene
关键帧	场景
layer structure	scripting
层结构	脚本编写
local scope	seed clip
局部作用域	种子剪辑
method	server
方法	服务器
modulo division	smart clip
模除法	智能剪辑
movie clip	superclass and subclass
影片剪辑	超类和子类
parameter	synchronous execution
参数	同步执行
play header	timeline loop
播放头	时间线循环
preload	variable
预装载器	变量

[ General Information ]

书名 = Action Script 权威指南

作者 =

页数 = 752

SS号 = 11197496

出版日期 =

封面  
书名  
版权  
前言  
目录  
目录

序言

序言

前言

前言

第一部分 ActionScript 基础

第一章 针对非程序员的简单介绍

一些基础习语

更为深入的 ActionScript 概念

创建多项选择测试

小结

第二章 变量

创建变量（声明）

变量赋值

变量值的修改和获取

值的类型

变量作用域

应用举例

小结

第三章 数据和数据类型

数据和信息

用数据类型来保持数据的意义

数据的创建和分类

数据类型转换

原始数据和复合数据

小结

第四章 原始数据类型

数字类型

整数和浮点数字

数值直接量

数字处理

串类型

串的处理

布尔类型

`undefined`

`null`

小结

第五章 操作符

操作符的一般特点

赋值操作符

算术操作符

等于和不等操作符

比较操作符

串操作符

逻辑操作符

组合操作符

逗号操作符

空（`void`）操作符

其他操作符

小结

第六章 语句

语句的类型

语句语法

ActionScript 语句

语句和动作

小结

## 第七章 条件语句

if 语句

else 语句

else if 语句

模拟 switch 语句

简化的条件语句语法

小结

## 第八章 循环语句

while 循环

循环术语

do - while 循环

for 循环

for - in 循环

提前终止循环

时间线和剪辑事件循环

小结

## 第九章 函数

函数的创建

函数的运行

向函数传递参数

退出函数并返回值

函数直接量

函数的可用性和生命周期

函数的作用域

再论函数参数

递归函数

内部函数

函数对象

代码的集中

再看多项选择测试

小结

## 第十章 事件和事件处理器

同步代码的执行

基于事件的异步代码执行

事件的类型

事件处理器

事件处理器语法

创建事件处理器

事件处理器作用域

按钮事件

影片剪辑事件综述

针对影片播放的影片剪辑事件

针对用户输入的影片剪辑事件

执行的顺序

复制剪辑事件处理器

用 updateAfterEvent 更新屏幕

代码的重复使用性

动态的影片剪辑事件处理器

事件处理器应用

小结

## 第十一章 数组

- 什么是数组
- 数组的分析
- 数组的创建
- 引用数组元素
- 确定数组的大小
- 命名数组元素
- 向数组添加元素
- 删除数组中的元素
- 通用数组操作工具
- 多维数组
- 多项选择测试的第三版本
- 小结

## 第十二章对象和类

- 对象的分析
- 实例化对象
- 对象属性
- 方法
- 类和面向对象的编程
- 内置ActionScript类和对象
- 小结

## 第十三章影片剪辑

- 影片剪辑的对象性
- 影片剪辑的类型
- 创建影片剪辑
- 影片和实例的堆栈顺序
- 实例和主影片的引用
- 删除剪辑实例和主影片
- 内置影片剪辑属性
- 影片剪辑方法
- 影片剪辑应用举例
- 最后的测试
- 小结

## 第十四章词法结构

- 空白
- 语句终结符（分号）
- 注释
- 保留字
- 标识符
- 大小写区分
- 小结

## 第十五章高级主题

- 复制、比较和传递数据
- 位逻辑编程
- 高级函数作用域问题
- 影片剪辑数据类型
- 小结

## 第二部分ActionScript应用

### 第十六章ActionScript制作环境

- 动作面板
- 为帧添加脚本
- 对按钮添加脚本
- 为影片剪辑添加脚本
- 代码都在哪里
- 生产力
- 外在化ActionScript代码
- 组件打包成智能剪辑

小结

## 第十七章 F I a s h 表单

F I a s h 表单数据循环

创建F I a s h 填充表单

小结

## 第十八章 屏幕文本域

动态文本域

用户输入文本域

文本域选项

文本域属性

H T M L 支持

关于文本域选择

空文本域和 f o r - i n 语句

小结

## 第十九章 调试

调试工具

调试方法

小结

## 第三部分 语言参考

### A c t i o n S c r i p t 语言参考

## 第四部分 附录

### 附录一 资源

### 附录二 L a t i n 1 字符指令表和键控代码

### 附录三 向后兼容

### 附录四 E C M A - 2 6 2 和 J a v a S c r i p t 之间的差别

### 词汇表