

使用 HTML 和 Ajax 开发 Adobe® AIR™ 1.5 应用程序



© 2009 Adobe Systems Incorporated. All rights reserved.

使用 HTML 和 Ajax 开发 Adobe® AIR® 1.5 应用程序

If this guide is distributed with software that includes an end user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company or person names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization or person.

Adobe, the Adobe logo, Acrobat, ActionScript, Adobe AIR, ColdFusion, Dreamweaver, Flash, Flex, Flex Builder, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Apple, Macintosh, and Mac OS are trademarks of Apple Inc., registered in the United States and other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. All other trademarks are the property of their respective owners.

This work is licensed under the Creative Commons Attribution Non-Commercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/us/>

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

MPEG Layer-3 audio compression technology licensed by Fraunhofer IIS and Thomson Multimedia (<http://www.mp3licensing.com>).

Speech compression and decompression technology licensed from Nellymoser, Inc. (www.nellymoser.com)

Video compression and decompression is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. <http://www.on2.com>.

This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com>)

This product contains either BSAFE and/or TIPEM software by RSA Security, Inc.



Sorenson Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

This product includes software developed by the IronSmith Project (<http://www.ironsmith.org/>).

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users. The Software and Documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

目录

第 1 章 : Adobe AIR 安装	
安装 Adobe AIR	1
删除 Adobe AIR	2
安装和运行 AIR 范例应用程序	2
第 2 章 : 安装 HTML 开发工具	
安装 AIR Extension for Dreamweaver	3
安装 AIR SDK	3
第 3 章 : Adobe AIR 简介	
AIR 1.1 中的新增功能	7
AIR 1.5 中的新增功能	7
第 4 章 : 查找 AIR 资源	
第 5 章 : 使用 AIR SDK 创建第一个基于 HTML 的 AIR 应用程序	
创建项目文件	10
创建 AIR 应用程序描述符文件	10
创建应用程序 HTML 页	11
测试应用程序	12
创建 AIR 安装文件	12
后续步骤	13
第 6 章 : 使用 Dreamweaver 创建第一个基于 HTML 的 AIR 应用程序	
准备应用程序文件	14
创建 Adobe AIR 应用程序	14
在桌面上安装应用程序	15
预览 Adobe AIR 应用程序	16
第 7 章 : 使用 AIR Extension for Dreamweaver	
在 Dreamweaver 中创建 AIR 应用程序	17
使用数字证书对应用程序进行签名	19
编辑关联的 AIR 文件类型	20
编辑 AIR 应用程序设置	20
在 AIR 应用程序中预览网页	21
使用 AIR 代码提示和代码颜色	21
访问 Adobe AIR 文档	21
第 8 章 : 使用命令行工具创建 AIR 应用程序	
使用 AIR Debug Launcher (ADL)	22
使用 AIR Developer Tool (ADT) 打包 AIR 安装文件	24

对 AIR 文件签名以更改应用程序证书	31
使用 ADT 创建自签名证书	32
配合使用 Apache Ant 和 SDK 工具	33
第 9 章：使用 AIR HTML 内部检查器进行调试	
关于 AIR 内部检查器	36
加载 AIR 内部检查器代码	36
在控制台选项卡中检查对象	36
配置 AIR 内部检查器	38
AIR 内部检查器界面	39
对非应用程序沙箱中的内容使用 AIR 内部检查器	45
第 10 章：HTML 和 JavaScript 编程	
创建基于 HTML 的 AIR 应用程序	46
示例应用程序和安全启示	46
避免与安全相关的 JavaScript 错误	48
通过 JavaScript 访问 AIR API 类	51
关于 AIR 中的 URL	53
在 HTML 中嵌入 SWF 内容	53
在 HTML 页中使用 ActionScript 库	54
转换 Date 和 RegExp 对象	55
从 ActionScript 操作 HTML 样式表	56
跨脚本访问不同安全沙箱中的内容	56
第 11 章：关于 HTML 环境	
HTML 环境概述	60
AIR 和 Webkit 扩展	62
第 12 章：处理与 HTML 相关的事件	
HTMLLoader 事件	73
AIR 类 - 事件处理与 HTML DOM 中其他事件处理的不同之处	73
Adobe AIR 事件对象	74
使用 JavaScript 处理运行时事件	76
第 13 章：撰写 HTML 容器脚本	
HTMLLoader 对象的显示属性	79
访问 HTML 历史记录列表	81
设置在加载 HTML 内容时使用的用户代理	82
设置用于 HTML 内容的字符编码	82
为 HTML 内容定义类似于浏览器的用户界面	82
第 14 章：AIR 安全性	
AIR 安全性基础知识	89
安装和更新	89

沙箱	92
HTML 安全性	94
通过脚本访问不同域中的内容	98
写入磁盘	99
安全使用不受信任的内容	100
开发人员的最佳安全做法	100
代码签名	101
第 15 章：设置 AIR 应用程序属性	
应用程序描述符文件结构	102
在应用程序描述符文件中定义属性	103
第 16 章：针对 JavaScript 开发人员的 ActionScript 基础知识	
ActionScript 和 JavaScript 之间的差异：概述	110
ActionScript 3.0 数据类型	111
ActionScript 3.0 类、包和命名空间	112
ActionScript 3.0 函数中的必需参数和默认值	113
ActionScript 3.0 事件侦听器	114
第 17 章：使用本机窗口	
有关本机窗口的其它在线信息	115
AIR 窗口基础知识	115
创建窗口	120
管理窗口	125
侦听窗口事件	131
显示全屏窗口	132
第 18 章：屏幕	
有关屏幕的其它在线信息	133
屏幕基础知识	133
枚举屏幕	134
第 19 章：使用本机菜单	
有关本机菜单的其它在线信息	137
AIR 菜单基础知识	137
创建本机菜单	141
关于 HTML 中的上下文菜单	142
显示弹出菜单	143
处理菜单事件	143
示例：窗口和应用程序菜单	145
使用 MenuBuilder 框架	147

第 20 章：任务栏图标

有关任务栏图标的其它在线信息	158
关于任务栏图标	158
停靠栏图标	158
系统任务栏图标	159
Window 任务栏图标和按钮	160

第 21 章：使用文件系统

有关 AIR 文件 API 的其它在线信息	162
AIR 文件基础知识	162
使用 File 对象	163
获取文件系统信息	170
使用目录	171
使用文件	172
读取和写入文件	175

第 22 章：拖放

有关拖放的其它在线信息	185
拖放基础知识	185
默认的拖放行为	185
HTML 中的拖放事件	186
用于 HTML 拖放的 MIME 类型	187
HTML 中的拖动效果	187
将数据拖出 HTML 元素	188
将数据拖入 HTML 元素	189
示例：覆盖默认的 HTML 拖入行为	189
在非应用程序 HTML 沙箱中处理文件放置	191

第 23 章：复制和粘贴

有关复制和粘贴的其它在线信息	192
复制和粘贴基础知识	192
读取和写入系统剪贴板	193
HTML 复制和粘贴	193
用于复制和粘贴的菜单命令和键击	195
剪贴板数据格式	197

第 24 章：使用字节数组

读取并写入 ByteArray	201
ByteArray 示例：读取 .zip 文件	206

第 25 章：使用本地 SQL 数据库

有关本地 SQL 数据库的其它在线信息	210
关于本地 SQL 数据库	211
创建和修改数据库	213

操作 SQL 数据库数据	215
使用同步和异步数据库操作	229
对 SQL 数据库使用加密	233
使用 SQL 数据库的策略	243
第 26 章：存储加密数据	
第 27 章：添加 PDF 内容	
检测 PDF 功能	248
加载 PDF 内容	249
编写 PDF 内容的脚本	249
对 AIR 中的 PDF 内容的已知限制	250
第 28 章：处理声音	
声音处理基础知识	252
了解声音体系结构	252
加载外部声音文件	253
处理嵌入的声音	255
处理声音流文件	256
处理动态生成的音频	256
播放声音	258
处理声音元数据	261
访问原始声音数据	262
捕获声音输入	265
第 29 章：使用数字权限管理	
有关数字权限管理的其它在线信息	268
了解加密的 FLV 工作流程	268
NetStream 类中与 DRM 相关的成员和事件	271
使用 DRMStatusEvent 类	272
使用 DRMAuthenticateEvent 类	273
使用 DRMErrorEvent 类	274
使用 DRMManager 类	276
使用 DRMContentData 类	277
第 30 章：应用程序启动和退出选项	
有关启动和退出选项的其它在线信息	278
应用程序调用	278
捕获命令行参数	279
登录时启动	281
浏览器调用	282
应用程序终止	283

第 31 章：读取应用程序设置	
读取应用程序描述符文件	285
获取应用程序标识符和发行商标识符	285
第 32 章：使用运行时和操作系统信息	
管理文件关联	286
获取运行时版本和修补级别	286
检测 AIR 功能	286
跟踪用户当前状态	287
第 33 章：监视网络连接	
检测网络连接更改	288
服务监视基础知识	288
检测 HTTP 连接	289
检测套接字连接	289
第 34 章：URL 请求和网络	
网络和通信基础知识	290
使用 URLRequest 类	291
使用外部数据	293
使用 URLStream 类	296
套接字连接	297
在默认系统 Web 浏览器中打开 URL	300
向服务器发送 URL	301
第 35 章：应用程序间的通信	
关于 LocalConnection 类	302
在两个应用程序之间发送消息	302
连接到不同域中的内容和其它 AIR 应用程序	303
第 36 章：分发、安装和运行 AIR 应用程序	
从桌面安装和运行 AIR 应用程序	305
从网页安装和运行 AIR 应用程序	306
企业部署	312
对 AIR 文件进行数字签名	312
第 37 章：更新 AIR 应用程序	
关于更新应用程序	318
提供自定义应用程序更新用户界面	319
将 AIR 文件下载到用户的计算机	320
检查应用程序是否为首次运行	320
使用更新框架	321

第 38 章：查看源代码

加载、配置和打开 Source Viewer	331
Source Viewer 用户界面	334

第 39 章：本地化 AIR 应用程序

本地化简介	335
对应用程序安装程序中的应用程序名称和说明进行本地化	335
选择区域设置	336
本地化 HTML 内容	336
对日期、时间和货币进行本地化	343
索引	344

第 1 章：Adobe AIR 安装

使用 Adobe® AIR®，可以在桌面上运行 AIR 应用程序。您可以通过以下方式安装运行时：

- 单独安装运行时（不同时安装 AIR 应用程序）
- 首次安装 AIR 应用程序（系统会提示您同时安装运行时）
- 安装 AIR 开发环境，如 AIR SDK、Adobe® Flex™ Builder™ 3 或 Adobe Flex™ 3 SDK（包括 AIR 命令行开发工具）
每台计算机只需要安装一次运行时。

[Adobe AIR: 系统要求](http://www.adobe.com/cn/products/air/systemreqs/) (<http://www.adobe.com/cn/products/air/systemreqs/>) 详细介绍了安装 AIR 和运行 AIR 应用程序的系统要求。

安装 Adobe AIR

按照以下说明下载并安装 AIR 的 Windows®、Mac OS X 和 Linux 版本。

若要更新运行时，用户必须具有计算机的管理权限。

在 **Windows** 计算机上安装运行时

- 1 下载[运行时安装文件](#)。
- 2 双击运行时安装文件。
- 3 在安装窗口中，按照提示完成安装。

在 **Mac** 计算机上安装运行时

- 1 下载[运行时安装文件](#)。
- 2 双击运行时安装文件。
- 3 在安装窗口中，按照提示完成安装。
- 4 如果安装程序显示“身份验证”(Authenticate) 窗口，请输入 Mac OS 用户名和密码。

在 **Linux** 计算机上安装运行时

- 1 下载[运行时安装文件](#)。
- 2 设置文件权限，以便可以执行安装程序应用程序：

从命令行中可以用 `chmod +x installer.bin` 命令设置文件权限。某些 Linux 版本允许您在通过上下文菜单打开的“属性”对话框上设置文件权限。

- 3 从命令行中或通过双击运行时安装文件运行安装程序。
- 4 在安装窗口中，按照提示完成安装。

AIR 作为 rpm 或 dpkg 包进行安装，包名称分别为 `adobeairv.n` 和 `adobecerts`。安装需要运行 X Server。AIR 注册 mime 类型：`application/vnd.adobe.air-application-installer-package+zip`。

删除 Adobe AIR

安装运行时之后，可以通过以下步骤将其删除。

在 **Windows** 计算机上删除运行时

- 1 在 Windows 的“开始”菜单中，选择“设置”>“控制面板”。
- 2 选择“添加或删除程序”控制面板。
- 3 选择“Adobe AIR”以删除运行时。
- 4 单击“更改 / 删除”按钮。

在 **Mac** 计算机上删除运行时

- 双击“Adobe AIR Uninstaller”，其位于 /Applications/Utilities 文件夹中。

在 **Linux** 计算机上删除运行时

请执行下列操作之一：

- 从“应用程序”菜单中选择“Adobe AIR Uninstaller”命令。
- 运行 AIR 安装程序二进制文件，并加入 -uninstall 选项
- 用包管理器删除 AIR 包 (adobeairv.n 和 adobecerts)。

安装和运行 AIR 范例应用程序

某些范例应用程序用于展示 AIR 功能。您可以按照以下说明访问和安装范例应用程序：

- 1 下载并运行 [AIR 范例应用程序](#)。编译后的应用程序及源代码都可用。
- 2 若要下载并运行范例应用程序，请单击范例应用程序的“Install Now”按钮。系统会提示您安装并运行该应用程序。
- 3 如果您选择下载范例应用程序并稍后运行，请选择下载链接。您可以随时通过以下方式运行 AIR 应用程序：
 - 在 Windows 中，双击桌面上的应用程序图标或从 Windows 的“开始”菜单中选择该应用程序。
 - 在 Mac OS 中，双击应用程序图标，默认情况下该应用程序安装在用户目录的 Applications 文件夹中（例如，在 Macintosh HD/Users/JoeUser/Applications/ 中）。
 - 在 Linux 中，双击桌面上的应用程序图标或从“应用程序”菜单中选择该应用程序图标。AIR 应用程序安装在其自身文件夹的 /opt 目录下。

注：检查 AIR 发行说明以查看是否有这些说明的更新，发行说明位于以下位置：

http://www.adobe.com/go/learn_air_relnotes_cn。

第 2 章：安装 HTML 开发工具

若要开发基于 HTML 的 Adobe® AIR® 应用程序，可以使用 Adobe® AIR® Extension for Dreamweaver、AIR SDK 命令行工具或其它支持 Adobe AIR 的 Web 开发工具。本主题介绍如何安装 Adobe AIR Extension for Dreamweaver 和 AIR SDK。

安装 AIR Extension for Dreamweaver

借助于 AIR Extension for Dreamweaver，可以为桌面创建丰富的 Internet 应用程序。例如，您可能使用一组彼此交互的网页来显示 XML 数据。您可以使用 Adobe AIR Extension for Dreamweaver，将这组页面打包为一个可安装在用户计算机上的小应用程序。当用户从其桌面上运行应用程序时，应用程序将加载网站并在其自己的应用程序窗口中显示，而与浏览器无关。用户可随后在其计算机中本地浏览该网站，而无需建立 Internet 连接。

Adobe® ColdFusion® 和 PHP 等动态页面无法在 Adobe AIR 中运行。运行时只能使用 HTML 和 JavaScript。但是，可以在页面中使用 JavaScript，通过 Ajax 方法（如 XMLHttpRequest）或特定于 Adobe AIR 的 API 来调用 Internet 上公开的任何 Web 服务（包括 ColdFusion 或 PHP 生成的服务）。

有关使用 Adobe AIR 可以开发的应用程序类型的详细信息，请参阅第 6 页的“[Adobe AIR 简介](#)”。

系统要求

若要使用 Adobe AIR Extension for Dreamweaver，必须安装并正确配置以下软件：

- Dreamweaver CS3 或 Dreamweaver CS4
- Adobe® Extension Manager CS3
- Java JRE 1.4 或更高版本（必须具备此软件才能创建 Adobe AIR 文件）。可以在 <http://java.sun.com/> 上找到 Java JRE。

上述要求仅适用于使用 Dreamweaver 创建和预览 Adobe AIR 应用程序。若要在桌面上安装和运行 Adobe AIR 应用程序，计算机上还必须安装 Adobe AIR。若要下载运行时，请参阅 www.adobe.com/go/air_cn。

安装 Adobe AIR Extension for Dreamweaver

- 1 Adobe AIR Extension for Dreamweaver 可从以下网址下载：<http://www.adobe.com/products/air/tools/ajax/>。
- 2 在 Windows 资源管理器（Windows）或 Finder（Macintosh）中双击 .mxp 扩展文件。
- 3 按照屏幕上的说明安装扩展。
- 4 安装完成后，重新启动 Dreamweaver。

有关使用 Adobe AIR Extension for Dreamweaver 的信息，请参阅第 17 页的“[使用 AIR Extension for Dreamweaver](#)”。

安装 AIR SDK

Adobe AIR SDK 中包含以下命令行工具，可用于启动和打包应用程序：

AIR Debug Launcher (ADL) 允许您在不进行安装的情况下运行 AIR 应用程序。请参阅第 22 页的“[使用 AIR Debug Launcher \(ADL\)](#)”。

AIR Development Tool (ADT) 将 AIR 应用程序打包为可分发的安装包。请参阅第 24 页的“[使用 AIR Developer Tool \(ADT\) 打包 AIR 安装文件](#)”。

AIR 命令行工具要求必须在计算机上安装 Java。可以使用 JRE 或 JDK (1.4 或更高版本) 中的 Java 虚拟机。可以在 <http://java.sun.com/> 上找到 Java JRE 和 Java JDK。

注: 最终用户运行 AIR 应用程序时不需要 Java。

下载并安装 AIR SDK

可以按照以下说明下载并安装 AIR SDK:

在 Windows 中安装 AIR SDK

- 1 下载 AIR SDK 安装文件。
- 2 AIR SDK 按标准归档文件进行分发。若要安装 AIR, 请将 SDK 的内容提取到计算机上的一个文件夹 (例如: C:\Program Files\Adobe\AIRSDK 或 C:\AIRSDK) 中。
- 3 ADL 和 ADT 工具包含在 AIR SDK 的 bin 文件夹中; 请将此文件夹的路径添加到 PATH 环境变量中。

在 Mac OS X 中安装 AIR SDK

- 1 下载 AIR SDK 安装文件。
- 2 AIR SDK 按标准归档文件进行分发。若要安装 AIR, 请将 SDK 的内容提取到计算机上的一个文件夹 (例如: /Users/<userName>/Applications/AIRSDK) 中。
- 3 ADL 和 ADT 工具包含在 AIR SDK 的 bin 文件夹中; 请将此文件夹的路径添加到 PATH 环境变量中。

有关 AIR SDK 工具使用入门的信息, 请参阅第 22 页的“[使用命令行工具创建 AIR 应用程序](#)”。

AIR SDK 中的内容

下表介绍了 AIR SDK 中所包含文件的用途:

SDK 文件夹	文件 / 工具描述
BIN	adl.exe - 使用 AIR Debug Launcher (ADL), 可以在不对 AIR 应用程序进行打包和安装的情况下运行该应用程序。 有关使用此工具的信息, 请参阅第 22 页的“ 使用 AIR Debug Launcher (ADL) ”。
	adt.bat - AIR Developer Tool (ADT) 将应用程序打包为 AIR 文件以便分发。有关使用此工具的信息, 请参阅第 24 页的“ 使用 AIR Developer Tool (ADT) 打包 AIR 安装文件 ”。
FRAMEWORKS	AIRAliases.js - 提供 ActionScript 运行时类的“别名”定义, 通过这些定义可以访问 ActionScript 运行时类。有关使用此别名文件的信息, 请参阅第 52 页的“ 使用 AIRAliases.js 文件 ”。 servicemonitor.swf - 为 AIR 应用程序提供一种基于事件的响应方式, 对指定主机网络连接的更改做出响应。有关使用此框架的信息, 请参阅第 288 页的“ 监视网络连接 ”。
LIB	adt.jar - adt 可执行文件, 由 adt.bat 文件调用。 Descriptor.1.0.xsd - 应用程序架构文件。
RUNTIME	AIR 运行时 - 在对 AIR 应用程序进行打包或安装之前, ADL 使用该运行时启动这些应用程序。

SDK 文件夹	文件 / 工具描述
SAMPLES	此文件夹包含范例应用程序描述符文件、无缝安装功能 (badge.swf) 范例以及默认的 AIR 应用程序图标；请参阅第 305 页的“ 分发、安装和运行 AIR 应用程序 ”。
SRC	此文件夹中包含无缝安装范例的源文件。
TEMPLATES	descriptor-template.xml - 应用程序描述符文件的模板，每个 AIR 应用程序都需要该模板。有关应用程序描述符文件的详细描述，请参阅第 102 页的“ 设置 AIR 应用程序属性 ”。

第 3 章：Adobe AIR 简介

Adobe® AIR® 是一种跨操作系统的运行时，通过它可以利用现有 Web 开发技术（Adobe® Flash® CS3 Professional、Adobe® Flash® CS4 Professional、Adobe® Flex™、Adobe® ActionScript® 3.0、HTML、JavaScript®、Ajax）构建丰富 Internet 应用程序 (RIA) 并将其部署到桌面。

在 Adobe AIR 开发人员中心 (<http://www.adobe.com/cn/devnet/air/>) 可以找到有关 Adobe AIR 入门和使用的详细信息。

借助 AIR，您可以在熟悉的环境中工作，可以利用您认为用起来最舒适的工具和方法，并且由于它支持 Flash、Flex、HTML、JavaScript 和 Ajax，您可以创造满足您需要的可能的最佳体验。

例如：可以使用以下技术之一或其某一组合开发应用程序：

- Flash/Flex/ActionScript
- HTML/JavaScript/CSS/Ajax
- PDF 可以由任何应用程序利用

因此，AIR 应用程序可以：

- 基于根内容为 Flash/Flex (SWF) 的 Flash 或 Flex 应用程序
- 基于具有 HTML 或 PDF 的 Flash 或 Flex。根内容为包含 HTML (HTML、JS、CSS) 或 PDF 内容的 Flash/Flex (SWF) 的应用程序
- 基于 HTML。根内容为 HTML、JS、CSS 的应用程序
- 基于具有 Flash/Flex 或 PDF 的 HTML。根内容为包含 Flash/Flex (SWF) 或 PDF 内容的 HTML 的应用程序

用户与 AIR 应用程序交互的方式和他们与本机桌面应用程序交互的方式相同。在用户计算机上安装一次此运行时之后，即可像任何其他桌面应用程序一样安装和运行 AIR 应用程序。

此运行时通过在不同桌面间确保一致的功能和交互来提供用于部署应用程序的一致性跨操作系统平台和框架，从而消除跨浏览器测试。不是针对特定操作系统进行开发，而是以此运行时为目标，它具有以下优点：

- 针对 AIR 开发的应用程序可以在多个操作系统上运行，同时不需要进行额外的工作。此运行时确保在由 AIR 支持的所有操作系统上进行一致并可预知的呈现和交互。
- 可以更快地构建应用程序，因为此运行时让您可以利用现有 Web 技术和设计模式以及将您的基于 Web 的应用程序扩展到桌面，而不需要学习传统的桌面开发技术或复杂的本机代码。
- 与使用诸如 C 和 C++ 之类的较低级别的语言相比，使用此运行时可以更轻松地开发应用程序。无需管理特定于每个操作系统的复杂的低级别 API。

当针对 AIR 开发应用程序时，可以利用一组丰富的框架和 API：

- 由此运行时提供的特定于 AIR 的 API 和 AIR 框架
- SWF 文件中使用的 ActionScript API 和 Flex 框架（以及其他基于 ActionScript 的库和框架）
- HTML、CSS 和 JavaScript
- 大多数 Ajax 框架

AIR 在很大程度上改变了应用程序的创建、部署和使用方式。您获得了更富有创造性的控制能力，并可以将您的基于 Flash、Flex、HTML 和 Ajax 的应用程序扩展到桌面，而不需要学习传统的桌面开发技术。

AIR 1.1 中的新增功能

Adobe AIR 1.1 引入了以下新功能：

- 已将安装对话框和其它运行时对话框翻译为：
 - 巴西葡萄牙语
 - 简体中文和繁体中文
 - 法语
 - 德语
 - 意大利语
 - 日语
 - 韩语
 - 俄语
 - 法语
 - 西班牙语
- 为国际化应用程序的构建提供支持，其中包括双字节语言的键盘输入。请参阅第 335 页的“[本地化 AIR 应用程序](#)”。
 - 支持本地化应用程序描述符文件中的名称和描述属性。
 - 支持本地化 SQLite 数据库中的错误消息，如 `SQLError.detailID` 和 `SQLError.detailArguments`。
 - 添加了 `Capabilities.languages` 属性，用于获取操作系统设置的首选 UI 语言的数组。
 - 已将 HTML 按钮标签和默认菜单（如上下文菜单和 Mac 菜单栏）本地化为所有支持的语言。
- 支持将证书从自签名应用程序迁移至与证书颁发机构 (CA) 相关联的应用程序。
- 支持 Microsoft Windows XP Tablet PC Edition；支持 Windows Vista® Home Premium、Business、Ultimate 或 Enterprise 的 64 位版本。
- 添加了 `File.spaceAvailable` API，用于获取磁盘上可用的磁盘空间量。
- 添加了 `NativeWindow.supportsTransparency` 属性，用于确定当前操作系统能否绘制透明窗口。

有关 AIR 1.1 发行版的详细信息，请参阅《Adobe AIR 1.1 发行说明》

(http://www.adobe.com/go/learn_air_relnotes_cn)。

AIR 1.5 中的新增功能

Adobe AIR 1.5 引入了以下新功能：

- 支持 Flash Player 10 的以下功能。
 - 自定义滤镜和效果
 - 增强的绘图 API
 - 动态生成声音
 - 矢量数据类型
 - 增强的文件上传和下载 API
 - 实时媒体流协议 (RTMFP)

- 3D 效果
- 高级文本支持
- 色彩管理
- 文本引擎
- 动态流
- Speex 音频编解码器

有关这些功能的详细信息，请参阅 <http://www.adobe.com/cn/products/flashplayer/features/>。

- AIR 1.5 安装程序和其它运行时对话框中支持的其它语言包括：捷克语、荷兰语、瑞典语、土耳其语和波兰语。
- 数据库加密。

在 AIR 1.5 中可以加密数据库文件。包括元数据在内的所有数据库内容都可以加密，这样便无法在用于加密数据的 AIR 应用程序之外读取这些数据。此功能将使开发人员可以加密、解密和重新加密数据库文件。请参阅第 246 页的“[存储加密数据](#)”。

- Adobe AIR 所用 WebKit 版本已更新，现在支持 SquirrelFish JavaScript 解释器。
- 新的 XML 签名验证 API，可用于帮助验证数据或信息的完整性和签名者身份。请参阅 [XML 签名验证](#)。

有关 AIR 1.5 发行版的详细信息，请参阅《Adobe AIR 1.5 发行说明》
(http://www.adobe.com/go/learn_air_relnotes_cn)。

第 4 章：查找 AIR 资源

有关开发 Adobe® AIR® 应用程序的详细信息，请参阅以下资源：

源	位置
《针对 HTML 开发人员的 Adobe AIR 语言参考》	http://www.adobe.com/go/learn_air_html_jslr_cn
《HTML 的 Adobe AIR 快速入门》	http://www.adobe.com/go/learn_air_html_qs_cn

可以在 Adobe AIR 开发人员中心 (<http://www.adobe.com/cn/devnet/air/>) 上找到由 Adobe 专家和社区专家撰写或制作的文章、范例和演示文稿。还可以从此处下载 Adobe AIR 和相关软件。

可以在 <http://www.adobe.com/cn/devnet/air/ajax/> 上找到专门为 HTML 和 Ajax 开发人员开辟的一方天地。

访问 Adobe 支持网站 (<http://www.adobe.com/cn/support/>)，查找有关您产品的疑难解答信息，并了解免费和付费技术支持选项。通过“培训”链接可访问 Adobe 出版社书籍、各种培训资源以及 Adobe 软件认证计划等。

第 5 章：使用 AIR SDK 创建第一个基于 HTML 的 AIR 应用程序

为了通过实际操作较快地说明 Adobe® AIR® 的工作原理，请遵循以下说明创建并打包一个简单的基于 HTML 的 AIR“Hello World”应用程序。

开始前，必须已安装运行时并设置了 AIR SDK。本教程将涉及使用 AIR Debug Launcher (ADL) 和 AIR Developer Tool (ADT)。ADL 和 ADT 是命令行实用工具程序，可在 AIR SDK 的 bin 目录中找到（请参阅第 3 页的“[安装 HTML 开发工具](#)”）。本教程假定您已经熟悉从命令行运行程序并了解了如何针对您的操作系统设置所需的路径环境变量。

创建项目文件

每个基于 HTML 的 AIR 项目必须包含以下两个文件：可指定应用程序元数据的应用程序描述符文件，以及顶级 HTML 页。除了这两个必要文件外，此项目还包含一个 JavaScript 代码文件 AIRAliases.js，该文件可为 AIR API 类定义好记的别名变量。

开始前：

- 1 创建一个名为 HelloWorld 的目录来放置项目文件。
- 2 创建一个名为 HelloWorld-app.xml 的 XML 文件。
- 3 创建一个名为 HelloWorld.html 的 HTML 文件。
- 4 将 AIR SDK 的 frameworks 文件夹中的 AIRAliases.js 复制到项目目录中。

创建 AIR 应用程序描述符文件

开始构建 AIR 应用程序前，应创建一个具有以下结构的 XML 应用程序描述符文件：

```
<application>
    <id>...</id>
    <version>...</version>
    <filename>...</filename>
    <initialWindow>
        <content>...</content>
        <visible>...</visible>
        <width>...</width>
        <height>...</height>
    </initialWindow>
</application>
```

- 1 打开要编辑的 HelloWorld-app.xml。
- 2 添加根 `<application>` 元素，包括 AIR 命名空间属性：

`<application xmlns="http://ns.adobe.com/air/application/1.5">` 该命名空间的最后一部分“1.5”指定了应用程序所需的运行时版本。

- 3 添加 `<id>` 元素：

`<id>examples.html.HelloWorld</id>` 应用程序 ID 与发布者 ID（AIR 从对应用程序包进行签名时使用的证书中获取）一起可以标识唯一的应用程序。建议采用的形式为以点分隔的反向 DNS 样式的字符串，如 “com.company.AppName”。应用程序 ID 可用于安装、访问专用应用程序文件系统存储目录、访问专用加密存储以及应用程序间的通信。

4 添加 `<version>` 元素：

`<version>0.1</version>` 可帮助用户确定安装哪个版本的应用程序。

5 添加 `<filename>` 元素：

`<filename>HelloWorld</filename>` 用于操作系统中应用程序可执行文件、安装目录和对应用程序的其它引用的名称。

6 添加包含下列子元素的 `<initialWindow>` 元素，为初始应用程序窗口指定属性：

`<content>HelloWorld.html</content>` 标识 AIR 要加载的根 HTML 文件。

`<visible>true</visible>` 使窗口立即可见。

`<width>400</width>` 设置窗口宽度（以像素为单位）。

`<height>200</height>` 设置窗口高度。

7 保存该文件。完整的应用程序描述符文件应如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://ns.adobe.com/air/application/1.5">
    <id>examples.html.HelloWorld</id>
    <version>0.1</version>
    <filename>HelloWorld</filename>
    <initialWindow>
        <content>HelloWorld.html</content>
        <visible>true</visible>
        <width>400</width>
        <height>200</height>
    </initialWindow>
</application>
```

此示例仅设置了几个可能的应用程序属性。有关应用程序属性的完整设置（利用这些设置可以指定窗口镶边、窗口大小、透明度、默认安装目录、关联文件类型以及应用程序图标等），请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

创建应用程序 HTML 页

现在需要创建一个简单的 HTML 页以用作 AIR 应用程序的主文件。

1 打开要编辑的 `HelloWorld.html` 文件。添加以下 HTML 代码：

```
<html>
<head>
    <title>Hello World</title>
</head>
<body onLoad="appLoad()">
    <h1>Hello World</h1>
</body>
</html>
```

2 在该 HTML 的 `<head>` 部分，导入 `AIRAliases.js` 文件：

```
<script src="AIRAliases.js" type="text/javascript"></script>
```

AIR 针对 HTML 窗口对象定义一个名为 `runtime` 的属性。该 `runtime` 属性使用类的完全限定的包名称来提供对内置 AIR 类的访问。例如，若要创建 `AIR File` 对象，您可以在 JavaScript 中添加以下语句：

```
var textFile = new runtime.flash.filesystem.File("app:/textfield.txt");
```

`AIRAliases.js` 文件为最常用的 AIR API 定义了好记的别名。使用 `AIRAliases.js`，可以将对 `File` 类的引用缩短为以下形式：

```
var textFile = new air.File("app:/textfield.txt");
```

3 在 `AIRAliases` 脚本标签下方，添加另一个包含 JavaScript 函数的脚本标签来处理 `onLoad` 事件：

```
<script type="text/javascript">
function appLoad(){
    air.trace("Hello World");
}
</script>
```

appLoad() 函数仅调用 air.trace() 函数。使用 ADL 运行应用程序时，跟踪消息会输出到命令控制台。跟踪语句对于调试非常有用。

4 保存该文件。

现在 HelloWorld.html 文件应如下所示：

```
<html>
<head>
    <title>Hello World</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript">
        function appLoad(){
            air.trace("Hello World");
        }
    </script>
</head>
<body onLoad="appLoad()">
    <h1>Hello World</h1>
</body>
</html>
```

测试应用程序

若要从命令行运行和测试应用程序，请使用 AIR Debug Launcher (ADL) 实用程序。ADL 可执行文件可以在 AIR SDK 的 bin 目录中找到。如果尚未设置 AIR SDK，请参阅第 3 页的“[安装 HTML 开发工具](#)”。

- ❖ 首先，打开命令控制台或解释程序。更改到为此项目创建的目录。然后，运行以下命令：

```
adl HelloWorld-app.xml
```

AIR 窗口将会打开，以显示应用程序。同时，控制台窗口会显示由于 air.trace() 调用而产生的消息。

有关详细信息，请参阅第 22 页的“[使用 AIR Debug Launcher \(ADL\)](#)”。

创建 AIR 安装文件

在成功运行应用程序后，可以使用 ADT 实用程序将应用程序打包到一个 AIR 安装文件中。AIR 安装文件是包含所有应用程序文件的存档文件，您可以将其分发给用户。必须先安装 Adobe AIR，然后才能安装打包的 AIR 文件。

为了确保应用程序安全，所有 AIR 安装文件必须经过数字签名。为便于开发，您可以使用 ADT 或其他证书生成工具生成一个基本的自签名证书。还可以从 VeriSign 或 Thawte 等商用证书颁发机构购买商用代码签名证书。用户安装自签名 AIR 文件时，发行商在安装过程中会显示为“未知”。这是因为自签名证书仅确保 AIR 文件自创建后没有被更改。而无法阻止某人自签名一个伪装的 AIR 文件并将其显示为您的应用程序。对于公开发布的 AIR 文件，强烈建议使用可验证的商用证书。有关 AIR 安全问题的概述，请参阅 第 89 页的“[AIR 安全性](#)”。

生成自签名证书和密钥对

- ❖ 从命令提示符处，输入以下命令（ADT 可执行文件位于 AIR SDK 的 bin 目录中）：

```
adt -certificate -cn SelfSigned 1024-RSA sampleCert.pfx samplePassword
```

ADT 会生成一个包含证书和相关私钥的名为 `sampleCert.pfx` 的 **keystore** 文件。

此示例使用了证书允许设置的最少数量的属性。您可以为斜体部分的参数指定任何值。密钥类型必须为 1024-RSA 或 2048-RSA（请参阅第 312 页的“[对 AIR 文件进行数字签名](#)”）。

创建 AIR 安装文件

- ❖ 在命令提示符下，输入以下命令（在一行中）：

```
adt -package -storetype pkcs12 -keystore sampleCert.pfx HelloWorld.air  
HelloWorld-app.xml HelloWorld.html AIRAliases.js
```

系统将提示您输入 **keystore** 文件密码。

`HelloWorld.air` 参数表示 ADT 生成的 AIR 文件。`HelloWorld-app.xml` 表示应用程序描述符文件。后面的参数表示应用程序所使用的文件。此示例仅使用了两个文件，但可以包含任意数量的文件和目录。

在创建 AIR 包后，可以通过双击该包文件来安装和运行应用程序。也可在解释程序或命令窗口中键入 AIR 文件名作为命令。

后续步骤

在 AIR 中，通常 HTML 和 JavaScript 代码的行为与其在典型的 Web 浏览器中的行为相同。（事实上，AIR 和 Safari Web 浏览器都使用相同的 WebKit 呈现引擎。）但是，如果在 AIR 中开发 HTML 应用程序，则必须了解一些重要差异。有关这些差异和其它重要主题的详细信息，请参阅：

第 6 章：使用 Dreamweaver 创建第一个基于 HTML 的 AIR 应用程序

为了通过实际操作较快地说明 Adobe® AIR® 的工作原理，请按照以下说明使用 Adobe® AIR® Extension for Dreamweaver® 创建并打包一个简单的基于 HTML 的 AIR“Hello World”应用程序。

如果还从未进行过这样的创建，请下载并安装 Adobe AIR，其位置为：www.adobe.com/go/air_cn。

有关安装 Adobe AIR Extension for Dreamweaver 的说明，请参阅第 3 页的“[安装 AIR Extension for Dreamweaver](#)”。

有关包括系统要求在内的扩展插件的概述，请参阅第 17 页的“[使用 AIR Extension for Dreamweaver](#)”。

准备应用程序文件

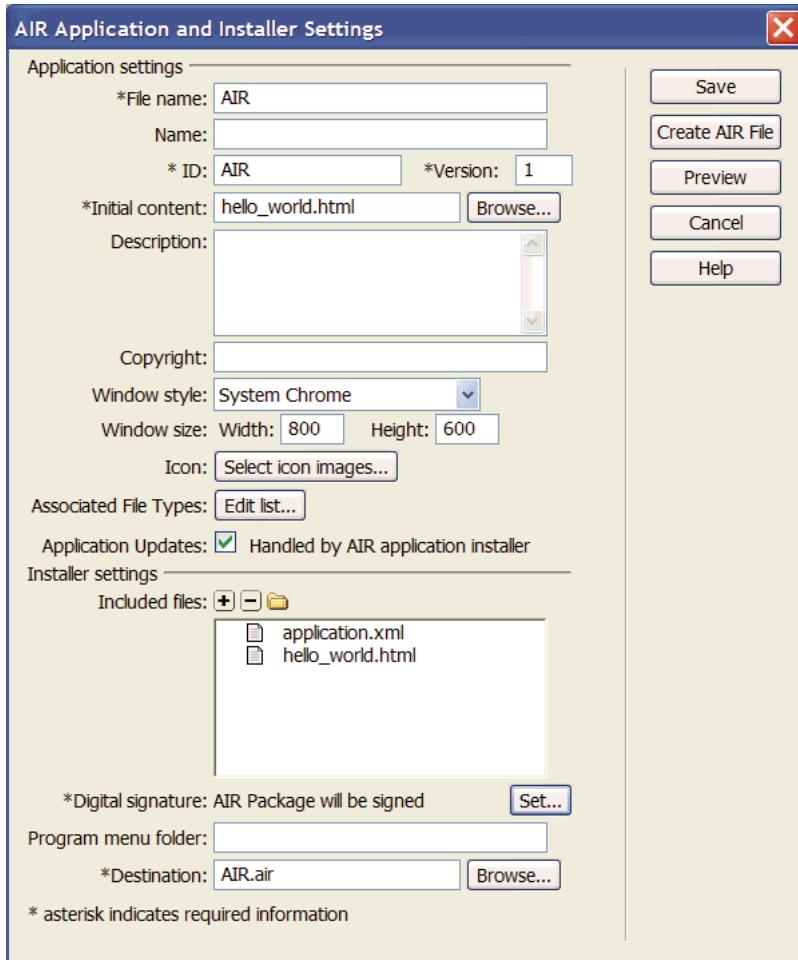
Adobe AIR 应用程序要求在 Dreamweaver 站点中定义一个起始页和其所有相关页。(有关 Dreamweaver 站点的详细信息，请参阅 Dreamweaver 帮助信息。) 若要为简单“Hello World”AIR 应用程序创建起始页，请按照以下说明进行操作：

- 1 启动 Dreamweaver 并确保定义一个站点。
- 2 打开一个新 HTML 页，方法是选择“文件”>“新建”，然后在“页面类型”列中选择 HTML，在“布局”列中选择“无”，最后单击“创建”。
- 3 在此新页面中，键入 **Hello World!**
此示例极其简单，但如果需要，也可以根据喜好设定文本样式，将其他内容添加到此页面中，将其他页面链接到此起始页，等等。
- 4 将此页面另存为 hello_world.html (“文件”>“保存”)。确保将此文件保存在 Dreamweaver 站点中。

创建 Adobe AIR 应用程序

- 1 确保在 Dreamweaver 文档窗口中打开 hello_world.html 页。(有关创建此页的说明，请参阅上一部分。)
- 2 选择“网站”>“Air 应用程序设置”。
“AIR 应用程序和设置”对话框中的大多数所需设置都自动为您填充。但是，您必须选择应用程序的起始内容（或起始页）。
- 3 单击“起始内容”选项旁的“浏览”按钮，导航到 hello_world.html 页，然后选择此页。
- 4 在“数字签名”选项的旁边，单击“设置”按钮。
数字签名用于保证自软件作者创建应用程序起应用程序的代码未经更改或未受损坏，所有 Adobe AIR 应用程序均需要数字签名。
- 5 在“数字签名”对话框中，选择“使用数字证书为 AIR 包签名”，然后单击“创建”按钮。（如果已具有对某一数字证书的访问权限，则可以单击“浏览”按钮选择此数字证书。）
- 6 完成“自签名数字证书”对话框中的所需字段。您将需要输入您的名称，输入密码并确认，以及输入数字证书文件的名称。Dreamweaver 将在站点根目录中保存此数字证书。
- 7 单击“确定”以返回“数字签名”对话框。
- 8 在“数字签名”对话框中，输入您为您的数字证书指定的密码并单击“确定”。

完成后的“AIR 应用程序和安装程序设置”对话框可能类似于如下内容：



有关所有对话框选项以及如何编辑这些选项的详细说明，请参阅第 17 页的“[在 Dreamweaver 中创建 AIR 应用程序](#)”。

9 单击“创建 AIR 文件”按钮。

Dreamweaver 将创建相应的 Adobe AIR 应用程序文件并将其保存在站点根文件夹中。Dreamweaver 还将创建 application.xml 文件并将其保存在相同的位置中。此文件的作用像清单一样，定义了应用程序的各种属性。

在桌面上安装应用程序

既然已创建应用程序文件，那就可以在任何桌面上安装该应用程序。

1 将相应的 Adobe AIR 应用程序文件移出 Dreamweaver 站点，并移到您的桌面或其他桌面上。

此步骤为可选步骤。实际上，如果愿意，可以从 Dreamweaver 站点目录直接将新建应用程序安装到计算机上。

2 双击应用程序可执行文件 (.air 文件) 以安装应用程序。

预览 Adobe AIR 应用程序

可以随时预览将归属于 AIR 应用程序的页面。即，不必打包应用程序即可查看其安装后的样子。

- 1 确保在 Dreamweaver 文档窗口中打开 hello_world.html 页。
- 2 在“文档”工具栏上，单击“在浏览器中预览 / 调试”按钮，然后选择“在 AIR 中预览”。
也可以按 Ctrl+Shift+F12 (Windows) 或 Cmd+Shift+F12 (Macintosh)。

当您预览此页时，您看到的实际上就是当用户在桌面上安装应用程序之后他们将看到的应用程序起始页。

第 7 章：使用 AIR Extension for Dreamweaver

可以通过 Adobe® AIR® Extension for Dreamweaver® 将基于 Web 的应用程序转换为桌面应用程序。这样，用户就可以在其桌面上运行该应用程序，并且在某些情况下无需 Internet 连接也可运行。

此扩展可用于 Dreamweaver CS3 和 Dreamweaver CS4。它与 Dreamweaver 8 不兼容。

有关安装此扩展的信息，请参阅第 3 页的“[安装 AIR Extension for Dreamweaver](#)”。

注 Adobe AIR 不支持 Adobe InContext Editing。如果您使用 AIR Extension for Dreamweaver 来导出包含 InContext Editing 区域的应用程序，则 InContext Editing 功能将无法正常工作。

在 Dreamweaver 中创建 AIR 应用程序

若要在 Dreamweaver 中创建基于 HTML 的 AIR 应用程序，请选择一个现有站点以将其打包成 AIR 应用程序。

- 1 请确保要打包成应用程序的网页包含在定义的 Dreamweaver 站点中。
- 2 在 Dreamweaver 中，打开要打包的页面集的主页。
- 3 选择“网站”>“Air 应用程序设置”。
- 4 完成“AIR 应用程序和安装程序设置”对话框，然后单击“创建 AIR 文件”。

有关详细信息，请参阅下面列出的对话框选项。

第一次创建 Adobe AIR 文件时，Dreamweaver 会在您站点的根文件夹中创建一个 application.xml 文件。此文件的作用像清单一样，定义了应用程序的各种属性。

下面将介绍“AIR 应用程序和安装程序设置”对话框中的选项：

应用程序文件名 是应用程序可执行文件所使用的名称。默认情况下，此扩展使用 Dreamweaver 站点的名称来为该文件命名。如果您愿意，可以更改此名称。不过，此名称只能包含对文件或文件夹名称有效的字符。（也就是说，它只能包含 ASCII 字符，且不能以句点结尾。）此设置是必需的。

应用程序名称 是当用户安装应用程序时出现在安装屏幕上的名称。同样，默认情况下此扩展会指定 Dreamweaver 站点的名称。此设置不存在字符限制，也不是必需的。

应用程序 ID 通过一个唯一的 ID 标识应用程序。如果您愿意，可以更改默认的 ID。请勿在 ID 中使用空格或特殊字符。唯一有效的字符是 0-9、a-z、A-Z、.（点）和 -（连字符）。此设置是必需的。

版本 指定应用程序的版本号。此设置是必需的。

初始内容 指定应用程序的起始页。单击“浏览”按钮可浏览到起始页并选择该页。所选的文件必须位于站点根文件夹内。此设置是必需的。

说明 用于指定在用户安装应用程序时显示的应用程序说明。

版权 用于指定安装在 Macintosh 上的 Adobe AIR 应用程序的“关于”信息中显示的版权。此信息不适用于安装在 Windows 上的应用程序。

窗口样式 指定当用户在其计算机上运行应用程序时所使用的窗口样式（或镶边）。系统镶边使用操作系统的标准窗口控件围绕应用程序。自定义镶边（不透明）会消除标准系统镶边，这样，您就可以为应用程序创建自己的镶边。（可以在打包的 HTML 页中直接构建自定义镶边。）自定义镶边（透明）类似于自定义镶边（不透明），但向页面边缘增加了透明功能，可用于非矩形的应用程序窗口。

窗口大小 指定应用程序窗口在打开时的尺寸。

图标 用于为应用程序图标选择自定义图像。(默认图像是此扩展附带的 Adobe AIR 图像。) 若要使用自定义图像, 请单击“选择图标图像”按钮。然后在出现的“图标图像”对话框中, 单击每个图标大小的文件夹, 选择要使用的图像文件。AIR 仅支持将 PNG 文件用于应用程序图标图像。

注: 所选的自定义图像必须位于应用程序站点内, 且其路径必须是相对于站点根目录的。

关联的文件类型 用于使文件类型与应用程序相关联。有关详细信息, 请参阅后面的部分。

应用程序更新 确定 Adobe AIR 应用程序的安装程序或应用程序本身是否对新版本的 Adobe AIR 应用程序执行更新。默认情况下此复选框是已选中, 这使得 Adobe AIR 应用程序的安装程序执行更新。如果希望应用程序执行它自己的更新, 请取消选中该复选框。请记住, 如果您取消选中该复选框, 则需要编写可以执行更新的应用程序。

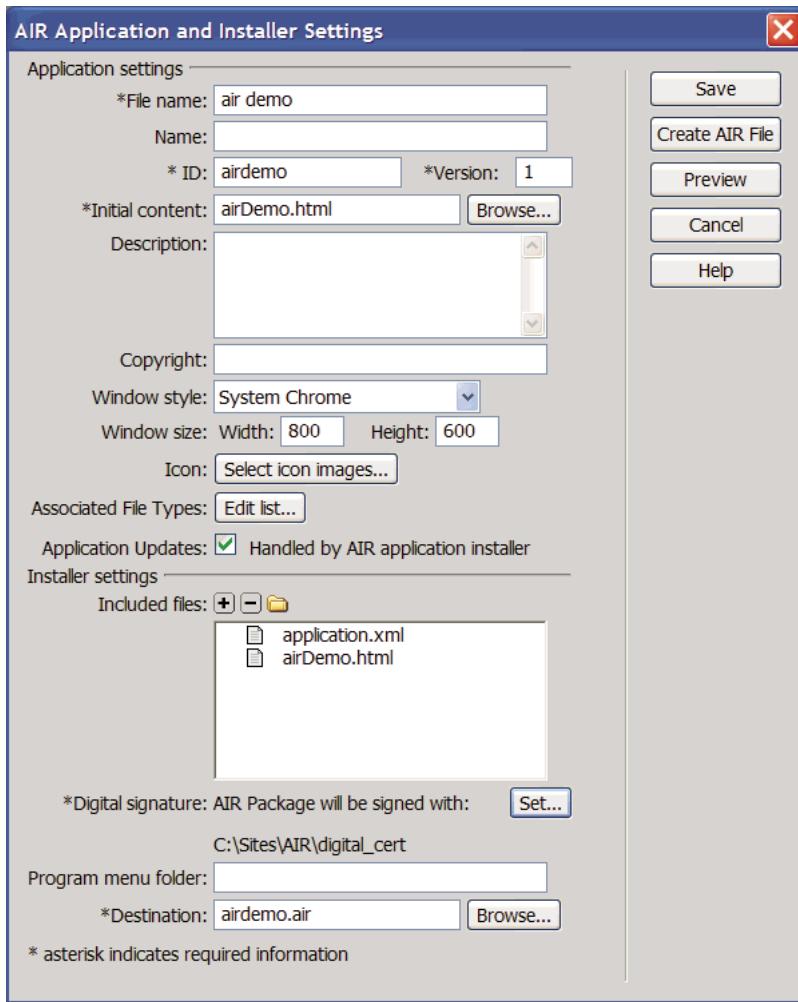
包括的文件 指定要在应用程序中包括哪些文件或文件夹。您可以添加 HTML 和 CSS 文件, 图像文件以及 JavaScript 库文件。单击加号 (+) 按钮可以添加文件, 单击文件夹图标可以添加文件夹。不应包括某些文件, 如 _mmServerScripts、_notes 等。若要从列表中删除某个文件或文件夹, 请选择该文件或文件夹, 然后单击减号 (-) 按钮。

数字签名 单击“设置”可使用数字签名对应用程序进行签名。此设置是必需的。有关详细信息, 请参阅后面的部分。

程序菜单文件夹 指定 Windows“开始”菜单中的一个子目录, 以便在其中创建应用程序的快捷方式。(不适用于 Macintosh。)

目标 指定新应用程序安装程序 (.air 文件) 的保存位置。默认位置为站点根目录。单击“浏览”按钮可以选择其他位置。默认的文件名是以站点名称为基础并添加一个 .air 扩展名形成的。此设置是必需的。

下面是设置了某些基本选项的此对话框的示例：



使用数字证书对应用程序进行签名

数据签名可确保软件作者创建应用程序代码后，该代码未遭更改或未损坏。所有 Adobe AIR 应用程序都需要具有数字签名，并且在没有数字签名的情况下无法安装。您可以使用购买的数字证书对应用程序进行签名，创建您自己的证书，或准备 Adobe AIRI 文件（一种 Adobe AIR 中间文件）以便您以后对它进行签名。

- 1 在“AIR 应用程序和安装程序设置”对话框中，单击“数字签名”选项旁边的“设置”按钮。
- 2 在“数字签名”对话框中，执行下列操作之一：
 - 若要使用预先购买的数字证书对应用程序进行签名，请单击“浏览”按钮，选择证书，输入对应的密码，然后单击“确定”。
 - 若要创建您自己的自签名数字证书，请单击“创建”按钮并完成该对话框。“证书类型”选项是指安全级别 1024-RSA 使用 1024 位密钥（不太安全），2048-RSA 使用 2048 位密钥（较为安全）。完成后，请单击“确定”。随后，在“数字签名”对话框中输入对应的密码并单击“确定”。
 - 选择“准备稍后将要为其签名的 AIRI 包”，然后单击“确定”。在没有数字签名的情况下，可以通过此选项创建 AIR 中间 (AIRI) 应用程序。不过，在添加数字签名之前，用户无法安装应用程序。

关于时间戳

当您使用数字证书对 Adobe AIR 应用程序进行签名时，打包工具将查询时间戳机构的服务器，以获取可独立验证的签名日期和时间。所获取的时间戳嵌入在 AIR 文件中。只要签名时签名证书有效，即使在证书过期后也可以安装 AIR 文件。另一方面，如果未获取时间戳，则在证书过期或被吊销之后，AIR 文件将变得不可安装。

默认情况下，Adobe AIR Extension for Dreamweaver 会在创建 Adobe AIR 应用程序时获取时间戳。不过，您可以通过在“数字签名”对话框中取消选中“时间戳”选项来禁用时间戳。（例如，在时间戳服务不可用时，您可能想这样做。）Adobe 建议使所有公开分发的 AIR 文件都包含一个时间戳。

AIR 打包工具所采用的默认时间戳机构是 Geotrust。有关时间戳和数字证书的详细信息，请参阅

编辑关联的 AIR 文件类型

您可以使不同的文件类型与 Adobe AIR 应用程序相关联。例如，如果希望在用户双击扩展名为 .avf 的文件时在 Adobe AIR 中打开这种类型的文件，则可以将 .avf 扩展名添加到关联的文件类型列表中。

1 在“AIR 应用程序和安装程序设置”对话框中，单击“关联的文件类型”选项旁边的“编辑列表”按钮。

2 在“关联的文件类型”对话框中，执行下列操作之一：

- 选择一种文件类型，然后单击减号 (-) 按钮以删除该文件类型。
- 单击加号 (+) 按钮以添加文件类型。

如果您通过单击加号按钮添加文件类型，将显示“文件类型设置”对话框。完成该对话框，然后单击“确定”将其关闭。

以下是选项列表：

名称 指定显示在“关联的文件类型”列表中的文件类型的名称。此选项是必需的，且只能包含 ASCII 字母数字字符（a-z、A-Z、0-9）和点（例如，adobe.VideoFile）。此名称必须以字母开头。最大长度为 38 个字符。

扩展名 指定文件类型的扩展名。请勿包含前面的点。此选项是必需的，且只能包含 ASCII 字母数字字符（a-z、A-Z、0-9）。最大长度为 38 个字符。

说明 用于指定对文件类型的可选说明。

内容类型 指定文件的 MIME 类型或媒体类型（例如，text/html、image/gif 等）。

图标文件位置 用于为关联的文件类型选择自定义图像。（默认图像是此扩展附带的 Adobe AIR 图像。）

编辑 AIR 应用程序设置

您随时可以编辑 Adobe AIR 应用程序的设置。

◆ 选择“站点”>“AIR 应用程序设置”(Air Application Settings)，然后进行更改。

在 AIR 应用程序中预览网页

您可以在 Dreamweaver 中预览 HTML 页在 Adobe AIR 应用程序中的显示情况。在不创建整个应用程序的情况下查看网页在应用程序中的显示情况时，预览非常有用。

- ◆ 在“文档”工具栏上，单击“在浏览器中预览 / 调试”按钮，然后选择“在 AIR 中预览”。
也可以按 Ctrl+Shift+F12 (Windows) 或 Cmd+Shift+F12 (Macintosh)。

使用 AIR 代码提示和代码颜色

Adobe AIR Extension for Dreamweaver 还在 Dreamweaver 的“代码”视图中为 Adobe AIR 语言元素添加了代码提示和代码颜色。

- 在“代码”视图中打开 HTML 或 JavaScript 文件，然后输入 Adobe AIR 代码。

注：代码提示机制只在 `<script>` 标记内部或 .js 文件内起作用。

有关 Adobe AIR 语言元素的详细信息，请参阅本指南其余部分的开发人员文档。

访问 Adobe AIR 文档

Adobe AIR 扩展在 Dreamweaver 中添加了“帮助”菜单项，可使用该菜单项访问“使用 HTML 和 Ajax 开发 AIR 应用程序”。

- ◆ 选择“帮助”>“Adobe AIR 帮助”。

另请参阅

第 14 页的“[使用 Dreamweaver 创建第一个基于 HTML 的 AIR 应用程序](#)”

第 8 章：使用命令行工具创建 AIR 应用程序

使用 Adobe® AIR® 命令行工具，您可以测试和打包 Adobe AIR 应用程序。还可以在自动构建过程中使用这些工具。Adobe® Flex™ 和 AIR SDK 中都包含这些命令行工具。

使用 AIR Debug Launcher (ADL)

使用 AIR Debug Launcher (ADL) 可以在开发期间运行基于 SWF 和基于 HTML 的应用程序。使用 ADL，您可以在不首先打包和安装应用程序的情况下运行该应用程序。默认情况下，ADL 使用 SDK 随附的运行时，这表示您无需单独安装运行时即可使用 ADL。

ADL 将 trace 语句和运行时错误输出到标准输出，但不支持断点或其他调试功能。如果您开发的是基于 SWF 的应用程序，请使用 Flash Debugger 解决复杂的调试问题。在用 ADL 运行应用程序之前，可以通过启动调试器程序连接到 Flash Debugger。

使用 ADL 启动应用程序

应使用以下语法：

```
adl [-runtime runtime-directory] [-pubid publisher-id] [-nodebug] application.xml [root-directory] [--arguments]
```

-runtime runtime-directory 指定包含要使用的运行时的目录。如果未指定，则使用 ADL 程序所在的相同 SDK 中的运行时目录。如果将 ADL 移出其 SDK 文件夹，则必须指定运行时目录。在 Windows 和 Linux 中，指定包含 Adobe AIR 目录的目录。在 Mac OS X 中，指定包含 Adobe AIR.framework 的目录。

-pubid publisher-id 分配指定值作为 AIR 应用程序的发行商 ID，以便完成此运行。通过指定临时的发行商 ID，您可以使用该发行商 ID 帮助唯一标识 AIR 应用程序，进而测试该应用程序的功能，例如通过本地连接进行通信。最终发行商 ID 由对 AIR 安装文件进行签名的数字证书确定。

-nodebug 关闭调试支持。如果使用此参数，应用程序进程则无法连接到 Flash Debugger，并禁止显示未处理异常对话框。（但是，跟踪语句仍将输出到控制台窗口。）关闭调试可以使应用程序的运行速度略有提高，并可以更准确地模拟已安装的应用程序的执行模式。

application.xml 应用程序描述符文件。请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

root-directory 指定要运行的应用程序的根目录。如果未指定，则使用应用程序描述符文件所在的目录。

-- arguments “--” 后显示的任何字符串均作为命令行参数传递到应用程序。

注：如果启动的 AIR 应用程序已在运行，则不会启动该应用程序的新实例，而是对正在运行的实例调度 invoke 事件。

输出 trace 语句

若要将 trace 语句输出到用于运行 ADL 的控制台，请使用 air.trace() 函数向代码添加 trace 语句：

```
trace("debug message");
air.trace("debug message");
```

在 JavaScript 中，可以使用 alert() 和 confirm() 函数显示来自应用程序的调试消息。此外，语法错误的行号以及任何未捕获的 JavaScript 异常均输出到该控制台。

ADL 示例

在当前目录中运行应用程序：

```
adl myApp-app.xml
```

在当前目录的子目录中运行应用程序：

```
adl source/myApp-app.xml release
```

运行应用程序，并传入“tick”和“tock”两个命令行参数：

```
adl myApp-app.xml -- tick tock
```

使用特定运行时运行应用程序：

```
adl -runtime /AIRSDK/runtime myApp-app.xml
```

运行不支持调试的应用程序：

```
adl myApp-app.xml -nodebug
```

连接 Flash Debugger (FDB)

若要使用 Flash Debugger 调试基于 HTML 的 AIR 应用程序，请启动一个 FDB 会话，然后使用 ADL 启动该应用程序。

- 1 启动 FDB。可以在 Flex SDK 的 bin 目录中找到 FDB 程序。

控制台显示 FDB 提示符：<fdb>

Flex SDK 位于 <http://opensource.adobe.com>。

- 2 执行 run 命令：<fdb>run [Enter]

- 3 在不同命令或解释程序控制台中，启动应用程序的调试版本：

```
adl myApp.xml
```

- 4 使用 FDB 命令根据需要设置断点。

- 5 类型：continue [Enter]

如果 AIR 应用程序基于 SWF，则调试器只控制 ActionScript 代码的执行。如果 AIR 应用程序基于 HTML，则调试器只控制 JavaScript 代码的执行。

若要在不连接到调试器的情况下运行 ADL，请加入 -nodebug 选项：

```
adl myApp.xml -nodebug
```

有关 FDB 命令的基本信息，请执行 help 命令：

```
<fdb>help [Enter]
```

有关 FDB 命令的详细信息，请参阅 Flex 文档中的 [Using the command-line debugger commands](#)（使用命令行调试器命令）。

ADL 退出和错误代码

下表列出了 ADL 输出的退出代码：

退出代码	说明
0	启动成功。ADL 在 AIR 应用程序退出后退出。
1	成功调用已在运行的 AIR 应用程序。ADL 立即退出。
2	用法错误。提供给 ADL 的参数错误。

退出代码	说明
3	无法找到运行时。
4	无法启动运行时。当应用程序中指定的版本或修补级别与运行时版本或修补级别不匹配时，通常会出现此情况。
5	发生未知原因错误。
6	无法找到应用程序描述符文件。
7	应用程序描述符内容无效。此错误通常指示 XML 格式错误。
8	无法找到主应用程序内容文件（在应用程序描述符文件的 <content> 元素中指定）。
9	主应用程序内容文件不是有效的 SWF 或 HTML 文件。

使用 AIR Developer Tool (ADT) 打包 AIR 安装文件

使用 AIR Developer Tool (ADT) 可以为基于 SWF 和基于 HTML 的 AIR 应用程序创建 AIR 安装文件。(如果您使用适用于 Dreamweaver® 的 Adobe® AIR® 扩展创建应用程序，则还可以使用“AIR 应用程序和安装程序设置”对话框中的“创建 AIR 文件”命令生成 AIR 包。请参阅第 17 页的“[使用 AIR Extension for Dreamweaver](#)”。)

ADT 是一个可从命令行运行的 Java 程序，也可以是一个生成工具（例如 Ant）。SDK 包含为您执行 Java 程序的命令行脚本。有关配置系统以运行 ADT 工具的信息，请参阅第 3 页的“[安装 HTML 开发工具](#)”。

打包 AIR 安装文件

每个 AIR 应用程序必须至少包含一个应用程序描述符文件和主 SWF 或 HTML 文件。安装的任何其他应用程序资源必须打包在 AIR 文件中。

必须使用数字证书对所有 AIR 安装程序文件签名。AIR 安装程序使用该签名验证应用程序文件自签名之后是否未发生任何更改。可以使用证书颁发机构颁发的代码签名证书，也可以使用自签名证书。由受信任证书颁发机构颁发的证书向应用程序用户提供一定的发行商身份保证。自签名证书无法用于验证发行商身份。此缺点还会削弱未对包进行更改的保证，因为合法安装文件在到达用户之前可能会被伪造安装文件取代)。

使用 ADT -package 命令，一步即可对 AIR 文件进行打包和签名。还可以使用 -prepare 命令创建未签名的中间包，然后在另一步中使用 -sign 命令对中间包签名。

当对安装包签名时，ADT 将自动连接时间戳签发机构服务器以验证时间。时间戳信息包含在 AIR 文件中。您可以在将来任一时间安装包含已经过验证的时间戳的 AIR 文件。如果 ADT 无法连接到时间戳服务器，则取消打包。您可以覆盖时间戳设置选项，但是如果没有任何时间戳，在对安装文件签名所使用的证书过期后，AIR 应用程序将停止安装。

如果要创建一个包以便更新现有 AIR 应用程序，则必须使用与原始应用程序相同的证书或具有相同标识的证书对此包进行签名。若要具有相同标识，两个证书必须具有相同的识别名称（所有信息字段均匹配），并且根证书必须具有相同的证书链。因此，只要未更改任何标识信息，则可以使用通过证书颁发机构续订的证书。

从 AIR 1.1 起，可以使用 -migrate 命令迁移应用程序，使其使用新证书。迁移证书需要使用新证书和旧证书对 AIR 文件进行签名。通过证书迁移，您可以将自签名证书更改为商用代码签名证书，或者将自签名证书或商用证书更改为其他自签名证书或商用证书。当迁移证书时，现有用户无需在安装现有应用程序的新版本之前卸载该应用程序。默认情况下，迁移签名带有时间戳。

注：应用程序描述符文件中的设置确定 AIR 应用程序的标识及其默认安装路径。请参阅第 102 页的“[应用程序描述符文件结构](#)”。

在一个步骤中对 AIR 文件进行打包和签名

❖ 在一个命令行中使用以下语法的 -package 命令：

```
adt -package SIGNING_OPTIONS air_file app_xml [file_or_dir | -C dir file_or_dir | -e file dir ...] ...
```

SIGNING_OPTIONS 该签名选项标识用于对 AIR 文件签名的私钥和证书所在的 **keystore**。若要使用 ADT 生成的自签名证书对 AIR 应用程序签名，使用的选项包括：

```
-storetype pkcs12 -keystore certificate.p12
```

在本例中，**certificate.p12** 为 **keystore** 的文件名。（由于未在命令行中提供密码，因此 ADT 提示您提供密码。）第 29 页的“[ADT 命令行签名选项](#)”详细介绍了签名选项。

air_file 创建的 AIR 文件的名称。

app_xml 指向应用程序描述符文件的路径。指定的路径可以是相对于当前目录的相对路径，也可以是绝对路径。（应用程序描述符文件在 AIR 文件中重命名为“**application.xml**”。）

file_or_dir 要在 AIR 文件中打包的文件和目录。可以指定任意数目的文件和目录，这些文件和目录以空格分隔。如果列出目录，则将该目录中的所有文件和子目录添加到该包中，但隐藏文件除外。（此外，如果指定应用程序描述符文件，则无论该文件是直接指定还是使用通配符或目录扩展指定的，都将忽略该文件，并且不会将其再次添加到包中。）指定的文件和目录必须位于当前目录或其子目录之一。使用 **-C** 选项更改当前目录。

重要说明：在 **file_or_dir** 参数中，不能在 **-C** 选项后使用通配符。（命令解释程序先展开通配符，然后再将该参数传递到 ADT，这将导致 ADT 在错误位置中查找文件。）但是，您仍可以使用点字符“.”表示当前目录。例如，“**-C assets.**”将资源目录中包括任何子目录在内的所有内容都复制到应用程序包的根级别。

-C dir 将工作目录更改为 **dir** 的值，然后处理添加到应用程序包的后续文件和目录。将文件和目录添加到应用程序包的根目录中。**-C** 选项可以使用任意次，以便包含文件系统多个点的文件。如果为 **dir** 指定相对路径，该路径则始终从原始工作目录解析。

由于 ADT 处理包包含的文件和目录，因此将存储当前目录和目标文件之间的相对路径。安装包时，这些路径将展开为应用程序目录结构。因此，指定 **-C release/bin lib/feature.swf** 会将 **release/bin/lib/feature.swf** 文件放置在根应用程序文件夹的 **lib** 子目录中。

-e file dir 将指定文件放置到指定包目录中。

注：应用程序描述符文件的 **<content>** 元素必须指定主应用程序文件在应用程序包目录树中的最终位置。

ADT 示例

打包当前目录中的特定应用程序文件：

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html AIRAliases.js image.gif
```

打包当前工作目录中的所有文件和子目录：

```
adt -package -storetype pkcs12 -keystore ../cert.p12 myApp.air myApp.xml .
```

注：**keystore** 文件包含用于对应用程序签名的私钥。切勿在 AIR 包中包含签名证书！如果在 ADT 命令中使用通配符，请将 **keystore** 文件放置到其他不同位置，使其不包含在包中。在本例中，**keystore** 文件 **cert.p12** 驻留在父目录中。

仅打包主文件和 **images** 子目录：

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html AIRAliases.js images
```

打包基于 HTML 的应用程序和 HTML、scripts 和 images 子目录中的所有文件：

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml index.html AIRAliases.js html scripts images
```

打包位于工作目录 (src) 中的 **application.xml** 文件和主 HTML 文件：

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air src/myApp.xml -C src myApp.html
```

对生成文件系统的多个位置的资源打包。在本例中，应用程序资源在打包前位于以下文件夹：

```
/devRoot
  /myApp
    /release
      /bin
        myApp.xml
        myApp.html
  /artwork
    /myApp
      /images
        image-1.png
        ...
        image-n.png
  /libraries
    /release
      /libs
        lib-1.js
        ...
        lib-n.js
        AIRAliases.js
```

从 /devRoot/myApp 目录运行以下 ADT 命令：

```
adt -package -storetype pkcs12 -keystore cert.p12 myApp.air release/bin/myApp.xml
-C release/bin myApp.swf
-C release/bin myApp.html
-C ../artwork/myApp images
-C ../libraries/release libs
```

生成以下包结构：

```
/myAppRoot
  /META-INF
    /AIR
      application.xml
      hash
  myApp.swf
  mimetype
  /images
    image-1.png
    ...
    image-n.png
  /libs
    lib-1.swf
    ...
    lib-n.swf
    AIRAliases.js

/myAppRoot
  /META-INF
    /AIR
      application.xml
      hash
  myApp.html
  mimetype
  /images
    image-1.png
    ...
    image-n.png
  /libs
    lib-1.js
    ...
    lib-n.js
    AIRAliases.js
```

作为 Java 程序运行 ADT (未设置类路径) :

```
java -jar {AIRSDK}/lib/ADT.jar -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf
java -jar {AIRSDK}/lib/ADT.jar -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html
AIRAliases.js
```

作为 Java 程序运行 ADT (将 Java 类路径设置为包含 ADT.jar 包) :

```
java com.adobe.air.ADTool -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.swf
java com.adobe.air.ADTool -package -storetype pkcs12 -keystore cert.p12 myApp.air myApp.xml myApp.html
AIRAliases.js
```

ADT 错误消息

下表列出了 ADT 程序可能报告的错误以及可能的原因:

应用程序描述符验证错误

错误代码	说明	备注
100	无法分析应用程序描述符	检查应用程序描述符文件中是否有标签未封闭等 XML 语法错误。
101	缺少命名空间	添加缺少的命名空间。
102	命名空间无效	检查命名空间拼写。
103	意外的元素或属性	删除引起错误的元素和属性。描述符文件中不允许使用自定义值。 检查元素和属性名称的拼写。 确保将元素放置在正确的父元素内，且使用属性时对应着正确的元素。
104	缺少元素或属性	添加所需的元素或属性。
105	元素或属性所含的某个值无效	纠正引起错误的值。
106	窗口属性组合非法	如 transparency = true 和 systemChrome = standard 等某些窗口设置不能在一起使用。更改其中某个不兼容的设置。
107	窗口最小大小大于窗口最大大小	更改最小大小或最大大小设置。

有关命名空间、元素、属性及其有效值的信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

应用程序图标错误

错误代码	说明	备注
200	无法打开图标文件	检查指定路径是否存在该文件。 使用另一个应用程序确保可以打开该文件。
201	图标大小错误	图标大小（以像素为单位）必须与 XML 标签相匹配。例如，假设有应用程序描述符元素： <image32x32>icon.png</image32x32> icon.png 中的图像必须刚好为 32x32 像素。
202	图标文件包含的某种图像格式不受支持	仅支持 PNG 格式。将应用程序打包之前转换其他格式的图像。

应用程序文件错误

错误代码	说明	备注
300	缺少文件，或无法打开文件	找不到或无法打开命令行中指定的文件。
301	缺少或无法打开应用程序描述符文件	在指定路径找不到应用程序描述符文件，或无法打开该文件。
302	包中缺少根内容文件	必须向包添加应用程序描述符的 <content> 元素中引用的 SWF 或 HTML 文件（通过将该文件加入 ADT 命令行中列出的文件中）。
303	包中缺少图标文件	必须向包添加应用程序描述符中指定的图标文件，方法是将这些图标加入到 ADT 命令行中列出的文件中。不会自动添加图标文件。
304	初始窗口内容无效	无法将应用程序描述符的 <content> 元素中引用的文件识别为有效的 HTML 或 SWF 文件。
305	初始窗口内容的 SWF 版本超出命名空间的版本	描述符命名空间中指定的 AIR 版本不支持应用程序描述符的 <content> 元素中所引用文件的 SWF 版本。例如，尝试将 SWF10 (Flash Player 10) 文件作为 AIR 1.1 应用程序的初始内容进行打包就会产生这种错误。

其他错误的退出代码

退出代码	说明	备注
2	用法错误	检查命令行参数中是否有错
5	未知错误	此错误表示所发生的情况无法按常见的错误条件作出解释。可能的根源包括 ADT 与 Java 运行时环境之间不兼容、ADT 或 JRE 安装损坏以及 ADT 内有编程错误。
6	无法写入输出目录	确保指定的（或隐含的）输出目录可访问，并且所在驱动器有足够的磁盘空间。
7	无法访问证书	确保正确指定了密钥存储库的路径。 检查能否访问密钥存储库中的证书。可以使用 Java 1.6 Keytool 实用程序帮助排除证书访问权限方面的问题。
8	证书无效	证书文件格式错误、被修改、已到期或被撤销。
9	无法为 AIR 文件签名	验证传递给 ADT 的签名选项。
10	无法创建时间戳	ADT 无法与时间戳服务器建立连接。如果通过代理服务器连接到 Internet，则可能需要配置 JRE 的代理服务器设置。
11	创建证书时出错	验证用于创建签名的命令行参数。
12	输入无效	验证命令行中传递给 ADT 的文件路径和其他参数。

ADT 命令行签名选项

ADT 使用 Java 加密体系结构 (JCA) 访问对 AIR 应用程序签名所使用的私钥和证书。签名选项标识 **keystore** 以及该 **keystore** 中的私钥和证书。

keystore 必须包含私钥和关联的证书链。证书链用于建立应用程序的发行商 ID。如果签名证书链接到某计算机上的受信任证书，则在“**AIR 安装**”对话框中显示该证书的公共名称作为发行商名称。

ADT 要求证书符合 x509v3 标准 ([RFC3280](http://tools.ietf.org/html/rfc3280))，并在扩展密钥用法扩展中包含代码签名的相应值。应遵循证书中的约束，并避免使用某些证书对 AIR 应用程序签名。

注：ADT 根据需要使用 Java 运行时环境代理设置连接 Internet 资源，以便检查证书吊销列表和获取时间戳。如果在使用 ADT 连接 Internet 资源时遇到问题，并且网络需要特定的代理设置，则可能需要配置 JRE 代理设置。

指定 AIR 签名选项

- ❖ 若要为 **-package** 和 **-prepare** 命令指定 ADT 签名选项，请使用以下语法：

```
[-alias aliasName] [-storetype type] [-keystore path] [-storepass password1] [-keypass password2] [-providerName className] [-tsa url]
```

-alias aliasName — **keystore** 中的密钥的别名。当 **keystore** 仅包含一个证书时，则不必指定别名。如果未指定任何别名，ADT 则使用 **keystore** 中的第一个密钥。

并非所有 **keystore** 管理应用程序都允许向证书分配别名。例如，当使用 Windows 系统 **keystore** 时，则使用证书的识别名称作为别名。使用 Java Keytool 实用程序可以列出可用证书以便确定别名。例如，运行以下命令：

```
keytool -list -storetype Windows-MY
```

将为证书生成如下输出：

```
CN=TestingCert,OU=QE,O=Adobe,C=US, PrivateKeyEntry,  
Certificate fingerprint (MD5): 73:D5:21:E9:8A:28:0A:AB:FD:1D:11:EA:BB:A7:55:88
```

若要在 ADT 命令行中引用此证书，请将别名设置为：

```
CN=TestingCert,OU=QE,O=Adobe,C=US
```

在 Mac OS X 中，Keychain 中的证书别名与在 Keychain Access 应用程序中显示的名称相同。

-storetype type — **keystore** 类型，由 **keystore** 实现确定。大多数 Java 安装随附的默认 **keystore** 实现支持 JKS 和 PKCS12 类型。Java 5.0 包含对 PKCS11 类型和 Keychain 类型的支持，前者用于访问硬件标记中的 **keystore**，后者用于访问 Mac OS X keychain。Java 6.0 包含对 MSCAPI 类型的支持（在 Windows 中）。如果安装和配置了其他 JCA 提供程序，则可能还可以使用其他 **keystore** 类型。如果未指定任何 **keystore** 类型，则使用默认 JCA 提供程序的默认类型。

存储类型	Keystore 格式	最低 Java 版本
JKS	Java keystore 文件 (.keystore)	1.2
PKCS12	PKCS12 文件 (.p12 或 .pfx)	1.4
PKCS11	硬件标记	1.5
KeychainStore	Mac OS X Keychain	1.5
Windows-MY 或 Windows-ROOT	MSCAPI	1.6

-keystore path — 基于文件的存储类型的 keystore 文件路径。

-storepass password1 — 访问 keystore 所需的密码。如果未指定，ADT 则提示提供密码。

-keypass password2 — 访问用于对 AIR 应用程序签名的私钥所需的密码。如果未指定，ADT 则提示提供密码。

-providerName className — 指定 keystore 类型的 JCA 提供程序。如果未指定，ADT 则使用该 keystore 类型的默认提供程序。

-tsa url — 指定符合 [RFC3161](#) 的时间戳服务器的 URL，以便对数字签名进行时间戳设置。如果未指定任何 URL，则使用 Geotrust 提供的默认时间戳服务器。对 AIR 应用程序签名设置时间戳时，仍可以在签名证书过期之后安装该应用程序，这是因为时间戳验证该证书在签名时是否有效。

如果 ADT 无法连接到时间戳服务器，则取消签名，并且不会生成任何包。指定 **-tsa none** 可以禁用时间戳。但是，对于打包的没有时间戳的 AIR 应用程序，该应用程序将在签名证书过期后停止安装。

注：签名选项类似于 Java Keytool 实用程序的等效选项。您可以使用 Keytool 实用程序在 Windows 中检查和管理 keystore。也可以在 Mac OS X 上使用 Apple® 安全实用程序实现此目的。

签名选项示例

使用 .p12 文件签名：

```
-storetype pkcs12 -keystore cert.p12
```

使用默认 Java keystore 签名：

```
-alias AIRcert -storetype jks
```

使用特定 Java keystore 签名：

```
-alias AIRcert -storetype jks -keystore certStore.keystore
```

使用 Mac OS X keychain 签名：

```
-alias AIRcert -storetype KeychainStore -providerName Apple
```

使用 Windows 系统 keystore 签名：

```
-alias cn=AIRCert -storetype Windows-MY
```

使用硬件标签签名（请参考标记制造商提供的 Java 配置相关说明，以便使用该标记并获取正确的 providerName 值）：

```
-alias AIRCert -storetype pkcs11 -providerName tokenProviderName
```

在不嵌入时间戳的情况下签名：

```
-storetype pkcs12 -keystore cert.p12 -tsa none
```

使用 ADT 创建未签名的 AIR 中间文件

使用 `-prepare` 命令可创建未签名的 AIR 中间文件。必须使用 ADT `-sign` 命令对 AIR 中间文件签名，才能生成有效的 AIR 安装文件。

`-prepare` 命令采用的标签和参数与 `-package` 命令相同（但签名选项除外）。唯一区别在于前者的输出文件未签名。生成的中间文件的文件扩展名为：airi。

若要对 AIR 中间文件签名，请使用 ADT `-sign` 命令。（请参阅[使用 ADT 对 AIR 中间文件进行签名](#)。）

ADT 示例

```
adt -prepare unsignedMyApp.airi myApp.xml myApp.swf components.swc

adt -prepare unsignedMyApp.airi myApp.xml myApp.html AIRAliases.js image.gif
```

使用 ADT 对 AIR 中间文件进行签名

若要用 ADT 对 AIR 中间文件签名，请使用 `-sign` 命令。`sign` 命令仅对 AIR 中间文件（扩展名为 airi）起作用。不能对 AIR 文件进行两次签名。

若要创建 AIR 中间文件，请使用 `adt -prepare` 命令。（请参阅第 31 页的“[使用 ADT 创建未签名的 AIR 中间文件](#)”。）

对 AIRI 文件签名

❖ 采用以下语法使用 ADT `-sign` 命令：

```
adt -sign SIGNING_OPTIONSairi_fileair_file
```

SIGNING_OPTIONS 该签名选项标识用于对 AIR 文件签名的私钥和证书。第 29 页的“[ADT 命令行签名选项](#)”介绍了这些选项。

airi_file 要进行签名的未签名 AIR 中间文件的路径。

air_file 要创建的 AIR 文件的名称。

ADT 示例

```
adt -sign -storetype pkcs12 -keystore cert.p12 unsignedMyApp.airi myApp.air
```

有关详细信息，请参阅第 312 页的“[对 AIR 文件进行数字签名](#)”。

对 AIR 文件签名以更改应用程序证书

若要更新现有 AIR 应用程序，使其使用新的签名证书，请使用 ADT `-migrate` 命令。

在下列情形中，证书迁移可能会很有用：

- 从自签名证书升级到由证书颁发机构颁发的证书
- 将即将过期的自签名证书更改为新的自签名证书
- 从一个商用证书更改为另一个商用证书，例如，当您的企业标识发生变化时

为了应用迁移签名，原始证书必须仍保持有效。一旦该证书过期，则不会应用迁移签名。应用程序用户将必须卸载现有版本才能安装更新版本。请注意，迁移签名默认带有时间戳标记，因此，即使证书已过期，使用迁移签名进行签名的 AIR 更新仍将保持有效。

注：续订以商业形式颁发的证书时，通常无需迁移证书。除非识别名称发生更改，否则续订后的证书将保留与原始证书相同的发行商身份。有关用于确定识别名称的证书属性的完整列表，请参阅第 313 页的“[关于 AIR 发行商标识符](#)”。

迁移应用程序以便使用新的证书：

- 1 创建应用程序更新
- 2 将 AIR 更新文件打包并使用新证书对它进行签名
- 3 通过 -migrate 命令，再次用原始证书对 AIR 文件签名

用 -migrate 命令签名的 AIR 文件可用于安装应用程序的新版本和更新任何早期版本，其中包括用旧证书签名的版本。

迁移 AIR 应用程序以便使用新的证书

- ❖ 采用以下语法使用 ADT -migrate 命令：

```
adt -migrate SIGNING_OPTIONS air_file_in air_file_out
```

SIGNING_OPTIONS 该签名选项标识用于对 AIR 文件签名的私钥和证书。这些选项必须标识原始签名证书，第 29 页的“[ADT 命令行签名选项](#)”介绍了这些选项。

air_file_in 使用新证书签名的要更新的 AIR 文件。

air_file_out 要创建的 AIR 文件。

ADT 示例

```
adt -migrate -storetype pkcs12 -keystore cert.p12 myApp.air myApp.air
```

有关详细信息，请参阅第 312 页的“[对 AIR 文件进行数字签名](#)”。

注 在 AIR 1.1 版本中，已将 -migrate 命令添加到 ADT。不支持将迁移签名应用于利用 AIR 1.0 附带的 ADT 版本生成的 AIR 包。

使用 ADT 创建自签名证书

使用自签名的证书，您可以生成有效的 AIR 安装文件，但由于无法验证自签名证书的可靠性，因此该证书只能向用户提供有限的安全保证。当安装自签名 AIR 文件时，发行商信息将对用户显示为“未知”。ADT 生成的证书有效期为五年。

如果对使用自签名证书签名的 AIR 应用程序创建更新，则必须使用相同证书对原始和更新 AIR 文件签名。即使使用相同的参数，ADT 生成的证书始终唯一。因此，如果希望使用 ADT 生成的证书对更新进行自签名，请将原始证书保存到安全位置。此外，在 ADT 生成的原始证书过期后，您无法生成更新的 AIR 文件。（您可以使用不同证书发布多个新应用程序，但是不能发布同一应用程序的多个新版本。）

重要说明：鉴于自签名证书的限制，Adobe 强烈建议使用由信誉良好的证书颁发机构所颁发的商用证书对公开发行的 AIR 应用程序进行签名。

ADT 生成的证书和关联私钥存储在 PKCS12 类型的 **keystore** 文件中。指定的密码是针对密钥自身（而不是 **keystore**）设置的。

生成数字 ID 证书以便对 AIR 文件进行自签名

- ❖ 在一个命令行中使用 ADT -certificate 命令：

```
adt -certificate -cn name [-ou org_unit] [-o org_name] [-c country] key_type pfx_file password
```

-cn name 分配的作为新证书公共名称的字符串。

-ou org_unit 分配的作为证书颁发组织单位的字符串。(可选。)

-o org_name 被分配作为证书颁发组织的字符串。(可选。)

-c country 双字母 ISO-3166 国家 / 地区代码。如果提供的代码无效，则不会生成证书。(可选。)

key_type 用于证书的密钥类型，即“1024-RSA”或“2048-RSA”。

pfx_file 证书文件的生成路径。

password 新证书的密码。当使用此证书对 AIR 文件签名时需要提供密码。

证书生成示例

```
adt -certificate -cn SelfSign -ou QE -o "Example, Co" -c US 2048-RSA newcert.p12 39#wnetx3tl
adt -certificate -cn ADigitalID 1024-RSA SigningCert.p12 39#wnetx3tl
```

若要使用这些证书对 AIR 文件签名，请将以下签名选项与 ADT -package 或 -prepare 命令一起使用：

```
-storetype pkcs12 -keystore newcert.p12 -keypass 39#wnetx3tl
-storetype pkcs12 -keystore SigningCert.p12 -keypass 39#wnetx3tl
```

配合使用 Apache Ant 和 SDK 工具

本主题提供使用 Apache Ant 生成工具测试和打包 AIR 应用程序的示例。

注：此讨论不会尝试提供对 Apache Ant 的全面概述。有关 Ant 文档，请访问 <http://Ant.Apache.org>。

将 Ant 用于简单项目

本示例演示如何使用 Ant 和 AIR 命令行工具生成 AIR 应用程序。简单项目结构用于单个目录中存储的所有文件。

为了更方便地重复使用生成脚本，以下示例使用定义的多个属性。其中一组属性标识命令行工具的安装位置：

```
<property name="SDK_HOME" value="C:/AIRSDK"/>
<property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
<property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>
```

另一组属性则特定于项目。在这些属性假定的命名约定中，应用程序描述符和 AIR 文件根据根目录源文件命名。还可以轻松地支持其他约定。

```
<property name="APP_NAME" value="ExampleApplication"/>
<property name="APP_ROOT" value=". "/>
<property name="APP_DESCRIPTOR" value="${APP_ROOT}/${APP_NAME}-app.xml"/>
<property name="AIR_NAME" value="${APP_NAME}.air"/>
<property name="STORETYPE" value="pkcs12"/>
<property name="KEYSTORE" value="ExampleCert.p12"/>
```

调用 ADL 以测试应用程序

若要使用 ADL 运行应用程序，请使用 exec 任务：

```
<target name="test" depends="compile">
<target name="test">
<exec executable="${ADL}">
<arg value="${APP_DESCRIPTOR}"/>
</exec>
</target>
```

调用 ADT 以打包应用程序

若要打包应用程序，请使用 Java 任务运行 `adt.jar` 工具：

```
<target name="package">
  <java jar="${ADT.JAR}" fork="true" failonerror="true">
    <arg value="-package"/>
    <arg value="-storetype"/>
    <arg value="${STORETYPE}"/>
    <arg value="-keystore"/>
    <arg value="${KEYSTORE}"/>
    <arg value="${AIR_NAME}"/>
    <arg value="${APP_DESCRIPTOR}"/>
    <arg value="${APP_NAME}.html"/>
    <arg value="*.png"/>
  </java>
</target>
```

如果应用程序有多个要打包的文件，则可以添加附加的 `<arg>` 元素。

将 Ant 用于较复杂项目

同单个目录相比，典型应用程序的目录结构较复杂。下面的示例演示用于编译、测试打包其项目目录结构更切合实际的 AIR 应用程序的生成文件。

release 该示例项目将应用程序源文件以及图标文件等其他资源存储到 `src` 目录中。生成脚本创建 `release` 目录以便存储最终的 AIR 包。

当 AIR 工具使用当前工作目录外部的文件时，该工具需要使用一些附加选项：

Testing 传递到 ADL 的第二个参数指定 AIR 应用程序的根目录。若要指定应用程序根目录，请将以下行添加到测试任务中：

```
<arg value="${debug}"/>
```

Packaging 打包最终包结构不应包含的子目录中的文件要求使用 `-C` 指令更改 ADT 工作目录。当使用 `-C` 指令时，新工作目录中的文件和目录即被复制到 AIR 包文件的根级别中。因此，`-C build file.png` 将 `file.png` 复制到应用程序的根目录。同样地，`-C assets icons` 将 `icon` 文件夹复制到根级别，并复制 `icons` 文件夹中的所有文件和目录。例如，打包任务中的以下行序列将 `icons` 目录直接添加到应用程序包文件的根级别中：

```
<arg value="-C"/>
<arg value="${assets}"/>
<arg value="icons"/>
```

注：如果需要将大量资源和资产移至不同相对位置，通常可以使用 Ant 任务将这些内容封装到临时目录中，这比为 ADT 生成复杂参数列表更简单。组织资源之后，可以使用简单 ADT 参数列表打包这些资源。

```
<project>
    <!-- SDK properties -->
    <property name="SDK_HOME" value="C:/AIRSDK"/>
    <property name="ADL" value="${SDK_HOME}/bin/adl.exe"/>
    <property name="ADT.JAR" value="${SDK_HOME}/lib/adt.jar"/>

    <!-- Project properties -->
    <property name="PROJ_ROOT_DIR" value=". "/>
    <property name="APP_NAME" value="ExampleApplication"/>
    <property name="APP_ROOT_DIR" value="${PROJ_ROOT_DIR}/src/html"/>
    <property name="APP_ROOT_FILE" value="${APP_NAME}.html"/>
    <property name="APP_DESCRIPTOR" value="${PROJ_ROOT_DIR}/${APP_NAME}-app.xml"/>
    <property name="AIR_NAME" value="${APP_NAME}.air"/>
    <property name="release" location="${PROJ_ROOT_DIR}/release"/>
    <property name="assets" location="${PROJ_ROOT_DIR}/src/assets"/>
    <property name="STORETYPE" value="pkcs12"/>
    <property name="KEYSTORE" value="ExampleCert.p12"/>

    <target name="init" depends="clean">
        <mkdir dir="${release}"/>
    </target>

    <target name="test">
        <exec executable="${ADL}">
            <arg value="${APP_DESCRIPTOR}"/>
            <arg value="${APP_ROOT_DIR}"/>
        </exec>
    </target>

    <target name="package" depends="init">
        <java jar="${ADT.JAR}" fork="true" failonerror="true">
            <arg value="-package"/>
            <arg value="-storetype"/>
            <arg value="${STORETYPE}"/>
            <arg value="-keystore"/>
            <arg value="${KEYSTORE}"/>
            <arg value="${release}/${AIR_NAME}"/>
            <arg value="${APP_DESCRIPTOR}"/>
            <arg value="-C"/>
            <arg value="${APP_ROOT_DIR}"/>
            <arg value="${APP_ROOT_FILE}"/>
            <arg value="-C"/>
            <arg value="${assets}"/>
            <arg value="icons"/>
        </java>
    </target>

    <target name="clean" description="clean up">
        <delete dir="${release}"/>
    </target>
</project>
```

第 9 章：使用 AIR HTML 内部检查器进行调试

Adobe® AIR® SDK 包含 `AIRIntrospector.js` JavaScript 文件，您可以在应用程序中包含该文件，协助调试基于 HTML 的应用程序。

关于 AIR 内部检查器

Adobe AIR HTML/JavaScript 应用程序内部检查器（称为 AIR HTML 内部检查器）提供了一些有用的功能，可有助于进行基于 HTML 的应用程序的开发和调试工作：

- 它包含一个内部检查器工具，借助此工具，您可以指向应用程序中的用户界面元素，并且可以查看元素的标记和 DOM 属性。
- 它包含用来发送对象引用以进行内部检查的控制台，并且您可以调整属性值和执行 JavaScript 代码。您还可以针对控制台为对象进行序列化，这将对数据编辑加以限制。您还可以从该控制台复制并保存文本。
- 它包含 DOM 属性和函数的树视图。
- 可用于编辑 DOM 元素的属性和文本节点。
- 它列出了应用程序中加载的链接、CSS 样式、图像和 JavaScript 文件。
- 可用于查看用户界面的初始 HTML 源代码和当前标记源代码。
- 可用于访问应用程序目录中的文件。（此功能仅当为应用程序沙箱打开 AIR HTML 内部检查器控制台时才可用。当为非应用程序沙箱内容打开控制台时不可用。）
- 它包含适用于 XMLHttpRequest 对象及其属性的查看器，其中包括 `responseText` 和 `responseXML` 属性（如果可用）。
- 您可以在源代码和文件中搜索匹配文本。

加载 AIR 内部检查器代码

AIR 内部检查器代码包含在 `AIRIntrospector.js` JavaScript 文件中，该文件包含在 AIR SDK 的框架目录中。若要在应用程序中使用 AIR 内部检查器，请将 `AIRIntrospector.js` 复制到应用程序的项目目录中，并在应用程序的 HTML 主文件中通过脚本标签加载该文件：

```
<script type="text/javascript" src="AIRIntrospector.js"></script>
```

此外，还要在与应用程序中不同本机窗口相对应的每个 HTML 文件中包含该文件。

重要说明：仅当在开发和调试应用程序时才包含 `AIRIntrospector.js` 文件。在分发的打包 AIR 应用程序中删除该文件。

`AIRIntrospector.js` 文件定义一个 `Console` 类，您可以通过从 JavaScript 代码调用 `air.Introspector.Console` 来访问该类。

注：使用 AIR 内部检查器的代码必须位于应用程序安全沙箱（在应用程序目录的文件中）中。

在控制台选项卡中检查对象

`Console` 类定义五种方法：`log()`、`warn()`、`info()`、`error()` 和 `dump()`。

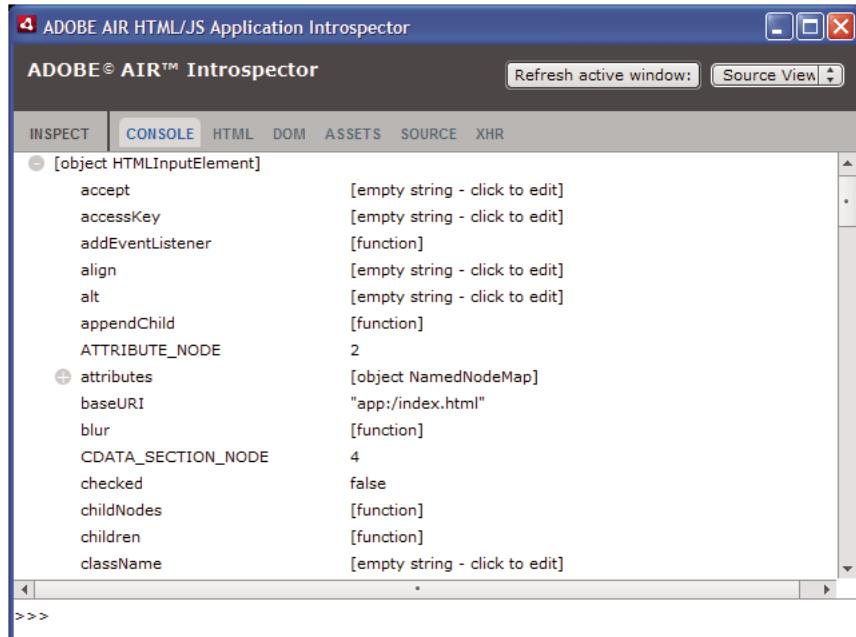
`log()`、`warn()`、`info()` 和 `error()` 方法均可用于向“控制台”选项卡发送对象。这些方法中最基本的是 `log()` 方法。以下代码将 `test` 变量表示的简单对象发送给“控制台”选项卡：

```
var test = "hello";
air.Introspector.Console.log(test);
```

不过，在向“控制台”选项卡发送复杂对象时，它更为有用。例如，以下 HTML 页面包含按钮 (`btn1`)，此按钮会调用向“控制台”选项卡发送按钮对象本身的函数：

```
<html>
  <head>
    <title>Source Viewer Sample</title>
    <script type="text/javascript" src="scripts/AIRIntrospector.js"></script>
    <script type="text/javascript">
      function logBtn()
      {
        var button1 = document.getElementById("btn1");
        air.Introspector.Console.log(button1);
      }
    </script>
  </head>
  <body>
    <p>Click to view the button object in the Console.</p>
    <input type="button" id="btn1"
      onclick="logBtn()"
      value="Log" />
  </body>
</html>
```

当您单击该按钮时，“控制台”选项卡会显示 `btn1` 对象，然后您可以展开该对象的树视图以检查其属性：



您可以通过单击属性名称右侧的列表并修改文本列表来编辑对象的属性。

`info()`、`error()` 和 `warn()` 方法与 `log()` 方法非常类似。不过，当您调用这些方法时，控制台会在行的开头显示图标：

方法	图标
info()	
error()	
warn()	

`log()`、`warn()`、`info()` 和 `error()` 方法仅发送实际对象的引用，因此可用属性是那些查看时的属性。如果要对实际对象进行序列化，请使用 `dump()` 方法。该方法具有两个参数：

参数	说明
<code>dumpObject</code>	要进行序列化处理的对象。
<code>levels</code>	要在对象树中检查的最大级别数（除根级别之外）。默认值为 1（表示显示树的除根级别之外的一个级别）。此参数是可选的。

调用 `dump()` 方法会在将对象发送至“控制台”选项卡之前对其进行序列化，这样您就无法编辑对象属性。例如，请考虑以下代码：

```
var testObject = new Object();
testObject.foo = "foo";
testObject.bar = 234;
air.Introspector.Console.dump(testObject);
```

当执行此代码时，控制台会显示 `testObject` 对象及其属性，但您无法在控制台中编辑相应属性值。

配置 AIR 内部检查器

您可以通过设置 `AIRIntrospectorConfig` 全局变量的属性来配置控制台。例如，以下 JavaScript 代码将 AIR 内部检查器配置为按照 100 个字符换行：

```
var AIRIntrospectorConfig = new Object();
AIRIntrospectorConfig.wrapColumns = 100;
```

请务必先设置 `AIRIntrospectorConfig` 变量的属性，然后再加载 `AIRIntrospector.js` 文件（通过 `script` 标签）。

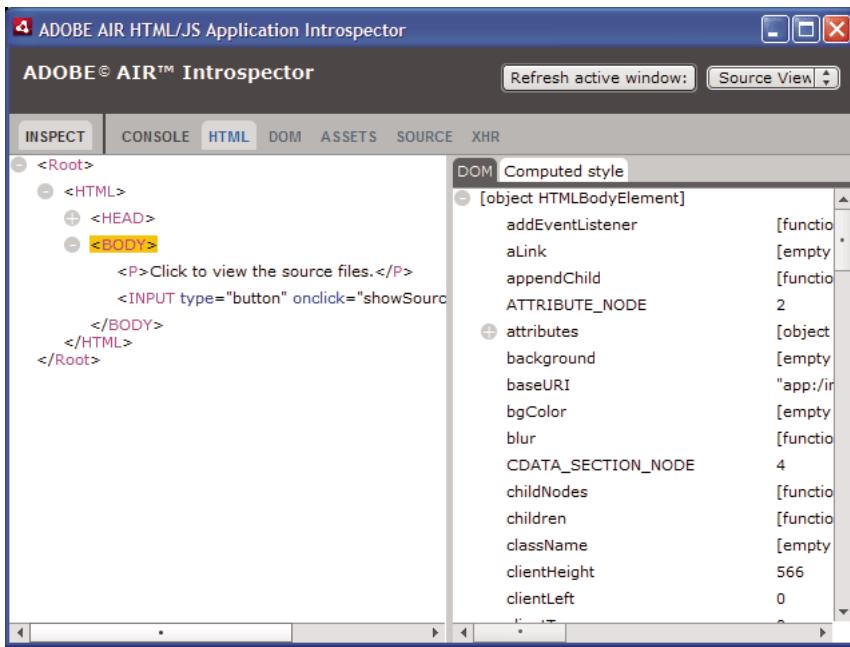
`AIRIntrospectorConfig` 变量具有八个属性：

属性	默认值	说明
<code>closeIntrospectorOnExit</code>	<code>true</code>	将检查器窗口设置为在应用程序的所有其它窗口都关闭后关闭。
<code>debuggerKey</code>	123 (F12 键)	用于显示和隐藏 AIR 内部检查器窗口的键盘快捷键的键控代码。
<code>debugRuntimeObjects</code>	<code>true</code>	将内部检查器设置为除了展开在 JavaScript 中定义的对象之外，还展开运行时对象。
<code>flashTabLabels</code>	<code>true</code>	将“控制台”选项卡和 <code>XMLHttpRequest</code> 选项卡设置为闪烁，以便它们的内容发生变化时（例如，当在这些选项卡中记录文本时）进行指示。
<code>introspectorKey</code>	122 (F11 键)	用于打开“检查”(Inspect) 面板的键盘快捷键的键控代码。
<code>showTimestamp</code>	<code>true</code>	将“控制台”选项卡设置为在每行的开头显示时间戳。
<code>showSender</code>	<code>true</code>	将“控制台”选项卡设置为在每行的开头显示有关发送消息的对象的信息。
<code>wrapColumns</code>	2000	对源文件换行时的列数。

AIR 内部检查器界面

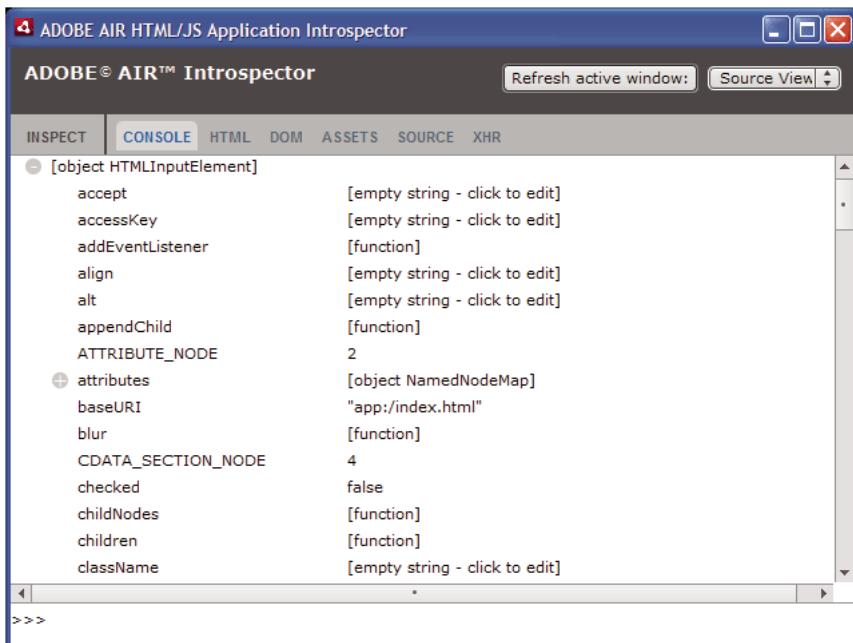
在调试应用程序时，若要打开 AIR 内部检查器窗口，请按 F12 键或调用 Console 类的方法之一（请参阅第 36 页的“[在控制台选项卡中检查对象](#)”）。您可以将热键配置为 F12 键之外的键；请参阅第 38 页的“[配置 AIR 内部检查器](#)”。

AIR 内部检查器窗口具有六个选项卡：“控制台”选项卡、HTML 选项卡、DOM 选项卡、“资源”选项卡、“源”选项卡和 XHR 选项卡，如下图所示：



控制台选项卡

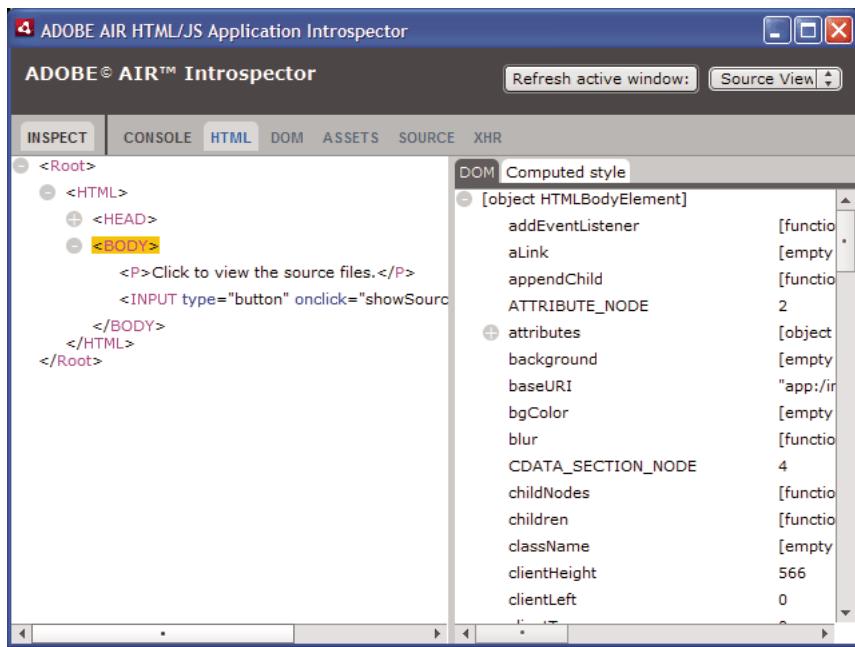
“控制台”选项卡显示以参数形式传递给 `air.Introspector.Console` 类方法之一的属性的值。有关详细信息，请参阅第 36 页的“[在控制台选项卡中检查对象](#)”。



- 若要清除控制台，请右键单击文本并选择“清除控制台”(Clear Console)。
- 若要将“控制台”选项卡中的文本保存到文件，请右键单击“控制台”选项卡并选择“将控制台保存到文件”(Save Console To File)。
- 若要将“控制台”选项卡中的文本保存到剪贴板，请右键单击“控制台”选项卡并选择“将控制台保存到剪贴板”(Save Console To Clipboard)。若仅将选定的文本复制到剪贴板，请右键单击相应文本并选择“复制”。
- 若要将 `Console` 类中的文本保存到文件，请右键单击“控制台”选项卡并选择“将控制台保存到文件”(Save Console To File)。
- 若要搜索该选项卡中显示的匹配文本，请单击 Ctrl+F (Windows) 或 Command+F (Mac OS)。(仅搜索可见树节点。)

HTML 选项卡

HTML 选项卡可用于以树结构查看整个 HTML DOM。单击某个元素可在该选项卡的右侧查看其属性。单击 + 和 - 图标可展开和折叠树中的节点。



您可以在 HTML 选项卡中编辑任何属性或文本元素，编辑的值会在应用程序中反映出来。

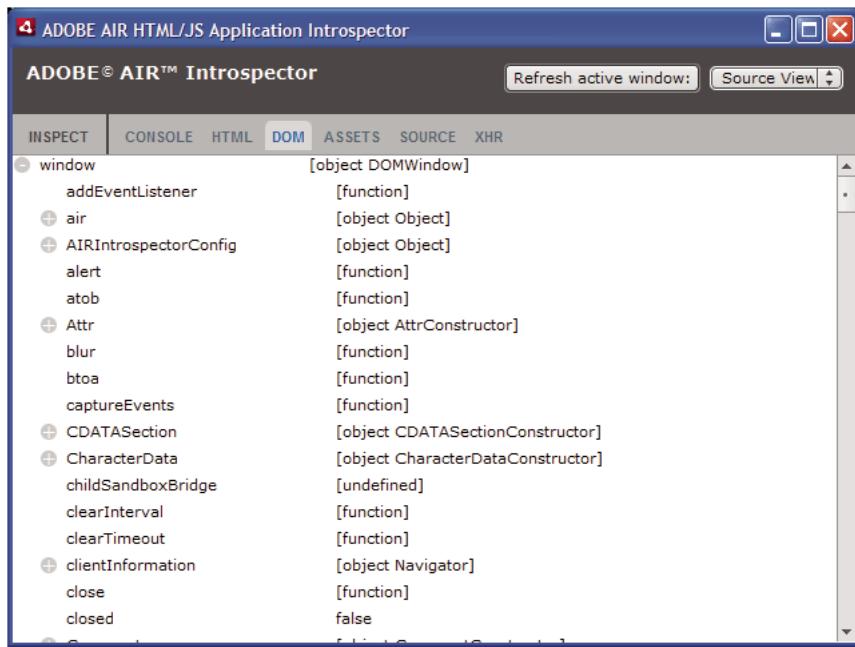
单击“检查”(Inspect)按钮（在 AIR 内部检查器窗口中选项卡列表的左侧）。您可以单击主窗口的 HTML 页面中的任何元素，关联的 DOM 对象会显示在 HTML 选项卡中。如果主窗口具有焦点，则您还可以通过按键盘快捷键打开或关闭“检查”(Inspect)按钮。默认情况下，F11 为键盘快捷键。您可以将键盘快捷键配置为 F11 键之外的键；请参阅第 38 页的“[配置 AIR 内部检查器](#)”。

单击“刷新活动窗口”(Refresh Active Window)按钮（在 AIR 内部检查器窗口的顶部）可刷新 HTML 选项卡中显示的数据。

单击 Ctrl+F (Windows) 或 Command+F (Mac OS) 可搜索该选项卡中显示的匹配文本。（仅搜索可见树节点。）

DOM 选项卡

DOM 选项卡以树结构显示窗口对象。您可以编辑任何字符串和数值属性，编辑的值会在应用程序中反映出来。

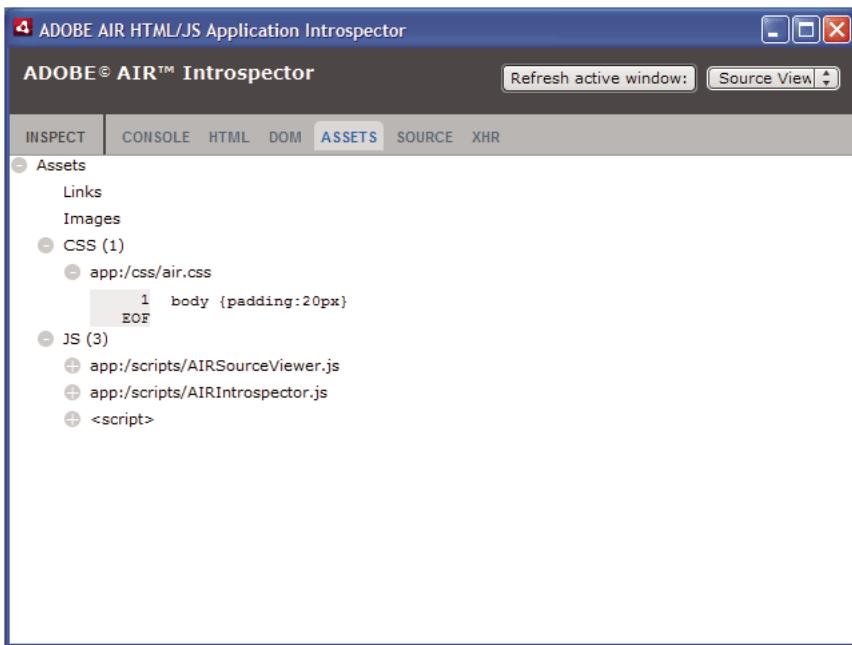


单击“刷新活动窗口”(Refresh Active Window) 按钮（在 AIR 内部检查器窗口的顶部）可刷新 DOM 选项卡中显示的数据。

单击 Ctrl+F (Windows) 或 Command+F (Mac OS) 可搜索该选项卡中显示的匹配文本。（仅搜索可见树节点。）

资源选项卡

“资源”选项卡可用于检查本机窗口中加载的链接、图像、CSS 和 JavaScript 文件。展开这些节点之一将显示文件的内容或显示所使用的实际图像。



单击“刷新活动窗口”(Refresh Active Window)按钮（在 AIR 内部检查器窗口的顶部）可刷新“资源”选项卡中显示的数据。

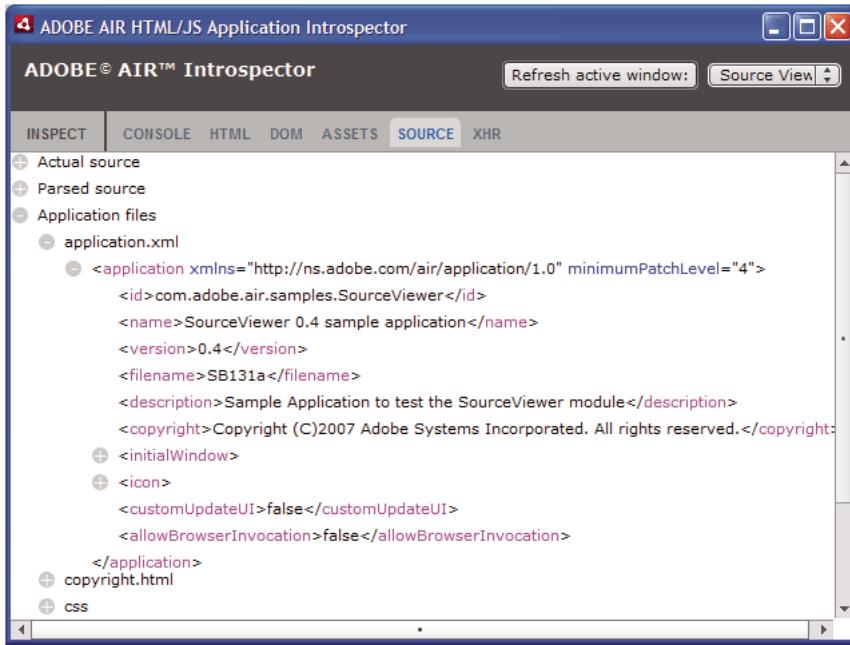
单击 Ctrl+F (Windows) 或 Command+F (Mac OS) 可搜索该选项卡中显示的匹配文本。（仅搜索可见树节点。）

源选项卡

“源”选项卡包含以下三个部分：

- 实际源代码 - 显示应用程序启动时作为根内容加载的页面的 HTML 源代码。
- 已解析源代码 - 显示构成应用程序 UI 的当前标记，它可能与实际源代码不同，因为应用程序采用 Ajax 技术动态生成标记代码。

- 应用程序文件 - 列出应用程序目录中的文件。此列表仅当从应用程序安全沙箱中的内容启动 AIR 内部检查器时才可供 AIR 内部检查器使用。在本部分中，您既可以查看文本文件的内容，也可以查看图像。

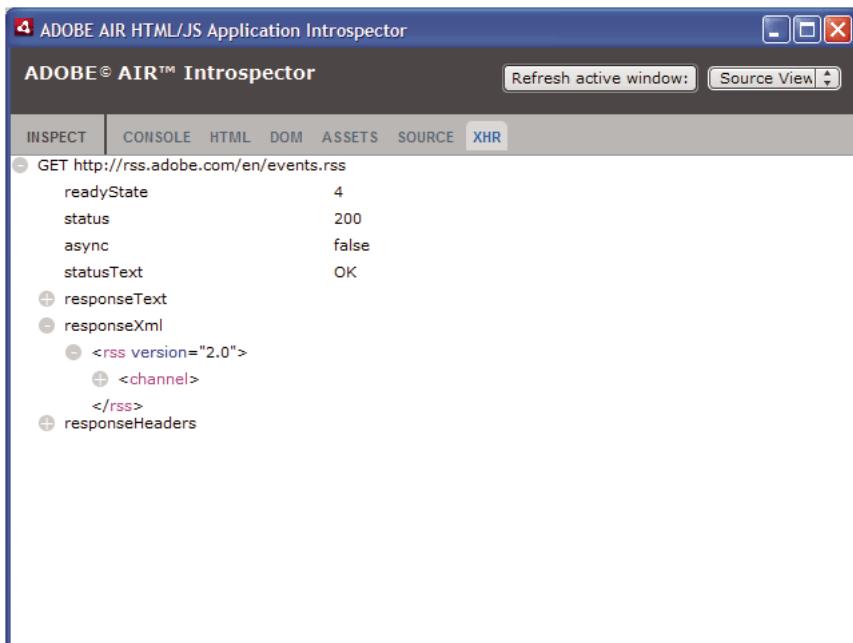


单击“刷新活动窗口”(Refresh Active Window) 按钮（在 AIR 内部检查器窗口的顶部）可刷新“源”选项卡中显示的数据。

单击 Ctrl+F (Windows) 或 Command+F (Mac OS) 可搜索该选项卡中显示的匹配文本。（仅搜索可见树节点。）

XHR 选项卡

XHR 选项卡截获应用程序中的所有 XMLHttpRequest 通信并记录相关信息。这样，您即可在树视图中查看 XMLHttpRequest 属性（包括 responseText 和 responseXML（如果可用））。



单击 Ctrl+F (Windows) 或 Command+F (Mac OS) 可搜索该选项卡中显示的匹配文本。（仅搜索可见树节点。）

对非应用程序沙箱中的内容使用 AIR 内部检查器

您可以将应用程序目录中的内容载入映射到非应用程序沙箱的 `iframe` 或帧中（请参阅第 94 页的“[HTML 安全性](#)”）。您可以对此类内容使用 AIR 内部检查器，但要遵循以下规则：

- `AIRIntrospector.js` 文件必须同时包含在应用程序沙箱和非应用程序沙箱 (`iframe`) 内容中。
- 请勿覆盖 `parentSandboxBridge` 属性；AIR 内部检查器代码将使用此属性。根据需要添加属性。因此，请不要按如下方式编写代码：

```
parentSandboxBridge = mytrace: function(str) {runtime.trace(str)} ;
```

使用如下语法：

```
parentSandboxBridge.mytrace = function(str) {runtime.trace(str)};
```

- 从非应用程序沙箱内容，您无法通过按 F12 键或调用 `air.Introspector.Console` 类中的方法之一打开 AIR 内部检查器。您只能通过单击“打开内部检查器”(Open Introspector) 按钮来打开内部检查器窗口。默认情况下，该按钮添加在 `iframe` 或帧的右上角。（由于对非应用程序沙箱内容施加安全限制，因此只能通过用户执行一定的动作来打开新窗口，如单击按钮。）
- 您可以为应用程序沙箱和非应用程序沙箱打开单独的 AIR 内部检查器窗口。您可以通过 AIR 内部检查器窗口中显示的标题来区分二者。
- 当从非应用程序沙箱运行 AIR 内部检查器时，“源”选项卡不显示应用程序文件。
- AIR 内部检查器只能查看从其打开它的沙箱中的代码。

第 10 章 : HTML 和 JavaScript 编程

许多编程主题是专门针对使用 HTML 和 JavaScript 开发 Adobe® AIR® 应用程序而编写的。无论是要编写基于 HTML 的 AIR 应用程序，还是要编写使用 HTMLLoader 类（或 mx:HTML Flex™ 组件）运行 HTML 和 JavaScript 的基于 SWF 的 AIR 应用程序，以下信息都非常重要。

创建基于 HTML 的 AIR 应用程序

开发 AIR 应用程序的过程与开发基于 HTML 的 Web 应用程序的过程几乎相同。应用程序结构仍为基于页面的形式，使用 HTML 提供文档结构，使用 JavaScript 提供应用程序逻辑。此外，AIR 应用程序需要应用程序描述符文件，该文件包含应用程序的有关元数据，用于标识应用程序的根文件。

如果使用的是 Adobe® Dreamweaver®，则可以从 Dreamweaver 用户界面中直接测试 AIR 应用程序并对其进行打包。如果使用的是 AIR SDK，则可以使用命令行 ADL 实用程序来测试 AIR 应用程序。ADL 将读取应用程序描述符并启动该应用程序。可以使用命令行 ADT 实用程序将应用程序打包为 AIR 安装文件。

创建 AIR 应用程序的基本步骤：

- 1 创建应用程序描述符文件。内容元素可标识应用程序的根页面，在启动应用程序时将自动加载指定的根页面。（有关详细信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。）
- 2 创建应用程序页面和代码。
- 3 使用 ADL 实用程序或 Dreamweaver 测试应用程序。
- 4 使用 ADT 实用程序或 Dreamweaver 将应用程序打包为 AIR 安装文件。

有关这些步骤的分步指导，请参阅第 10 页的“[使用 AIR SDK 创建第一个基于 HTML 的 AIR 应用程序](#)”或第 14 页的“[使用 Dreamweaver 创建第一个基于 HTML 的 AIR 应用程序](#)”。

示例应用程序和安全启示

在第 10 页的“[使用 AIR SDK 创建第一个基于 HTML 的 AIR 应用程序](#)”中，特意将 Hello World 示例设置的简单。该示例仅调用一个特定于 AIR 的 API (`air.trace()` 方法)。以下 HTML 代码使用了更高级的 AIR API；该代码使用文件系统 API 来列出用户桌面目录中的文件和目录。



以下是该应用程序的 HTML 代码：

```
<html>
  <head>
    <title>Sample application</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script>
      function getDesktopFileList()
      {
        var log = document.getElementById("log");
        var files = air.File.desktopDirectory.getDirectoryListing();
        for (i = 0; i < files.length; i++)
        {
          log.innerHTML += files[i].name + "<br/>";
        }
      }
    </script>
  </head>
  <body onload="getDesktopFileList(); " style="padding: 10px">
    <h2>Files and folders on the desktop:</h2>
    <div id="log" style="width: 450px; height: 200px; overflow-y: scroll;" />
  </body>
</html>
```

还必须创建应用程序描述符文件，并使用 AIR Debug Launcher (ADL) 应用程序对该应用程序进行测试。（请参阅第 10 页的“[使用 AIR SDK 创建第一个基于 HTML 的 AIR 应用程序](#)”）。

大多数示例代码可在 Web 浏览器中使用。但是，有一些代码行是特定于运行时的。

getDesktopFileList() 方法使用 File 类，该类在运行时 API 中进行定义。应用程序中的第一个 script 标签加载 AIRAliases.js 文件（随 AIR SDK 一起提供），通过该文件可轻松访问 AIR API。（例如，示例代码使用 air.File 语法访问 AIR File 类。）有关详细信息，请参阅第 52 页的“[使用 AIRAliases.js 文件](#)”。

File.desktopDirectory 属性为 File 对象（一类由运行时定义的对象）。File 对象是对用户计算机中的文件或目录的引用。

File.desktopDirectory 属性是对用户桌面目录的引用。对任何 File 对象均定义了 getDirectoryListing() 方法，该方法返回 File 对象的数组。File.desktopDirectory.getDirectoryListing() 方法返回表示用户桌面上文件和目录的 File 对象的数组。

每个 File 对象都具有一个 name 属性，该属性表示文件名，以字符串的形式表示。getDesktopFileList() 方法中的 for 循环遍历用户桌面目录上的文件和目录，并将其名称追加到应用程序中 div 对象的 innerHTML 属性中。

在 AIR 应用程序中使用 HTML 的重要安全规则

随 AIR 应用程序一起安装的文件能够访问 AIR API。出于安全方面的考虑，来自其它源的内容不能访问 AIR API。例如，此限制将阻止远程域（例如 <http://example.com>）中的内容读取用户桌面目录中的内容（也可能是更严重的情况）。

由于存在通过调用 eval() 函数（及相关 API）可以利用的安全漏洞，因此默认情况下限制随应用程序安装的内容使用这些方法。但是，某些 Ajax 框架会调用 eval() 函数和相关 API。

为确保结构内容在 AIR 应用程序中能够正常工作，必须考虑对来自不同源的内容制订相应的安全限制规则。来自不同源的内容按不同的安全等级放置在沙箱（请参阅第 92 页的“[沙箱](#)”）中。默认情况下，随应用程序一起安装的内容安装在称为应用程序的沙箱中，这将授予该内容访问 AIR API 的权限。应用程序沙箱通常是最安全的沙箱，设计了一些限制，可阻止不受信任代码的执行。

运行时允许将随应用程序一起安装的内容加载到应用程序沙箱之外的沙箱中。非应用程序沙箱中的内容在类似于典型 Web 浏览器的安全环境中运行。例如，非应用程序沙箱中的代码可以使用 eval() 和相关方法（但不允许该代码访问 AIR API）。运行时包含有相关方法，可以让不同沙箱中的内容安全地进行通信（例如，不将 AIR API 公开给非应用程序内容）。有关详细信息，请参阅第 56 页的“[跨脚本访问不同安全沙箱中的内容](#)”。

如果出于安全方面的考虑，限制在沙箱中使用所调用的代码，则运行时将发出 JavaScript 错误：“Adobe AIR runtime security violation for JavaScript code in the application security sandbox”（应用程序安全沙箱中存在针对 JavaScript 代码的 Adobe AIR 运行时安全侵犯）。

为了避免此错误，请按照下一部分第 48 页的“[避免与安全相关的 JavaScript 错误](#)”中介绍的代码编写方法进行操作。

有关详细信息，请参阅第 94 页的“[HTML 安全性](#)”。

避免与安全相关的 JavaScript 错误

如果由于这些安全限制而限制在沙箱中使用所调用的代码，则运行时将发出 JavaScript 错误：“Adobe AIR runtime security violation for JavaScript code in the application security sandbox”（应用程序安全沙箱中存在针对 JavaScript 代码的 Adobe AIR 运行时安全侵犯）。为了避免此错误，请按照这些代码编写方法进行操作。

产生与安全相关的 JavaScript 错误的原因

一旦触发文档 load 事件并退出所有 load 事件处理函数，将限制应用程序沙箱中执行的代码执行涉及计算和执行字符串的大多数操作。如果尝试使用以下类型的可计算和执行潜在不安全字符串的 JavaScript 语句，则会生成 JavaScript 错误：

- [eval\(\)](#) 函数
- [setTimeout\(\)](#) 和 [setInterval\(\)](#)
- [Function](#) 构造函数

此外，以下类型的 JavaScript 语句也会失败，但不会生成不安全的 JavaScript 错误：

- [javascript: URL](#)
- [innerHTML](#) 和 [outerHTML](#) 语句中通过 [onevent](#) 属性分配的事件回调
- 从应用程序安装目录外部加载 JavaScript 文件
- [document.write\(\)](#) 和 [document.writeln\(\)](#)
- [load](#) 事件之前或 [load](#) 事件处理函数中的同步 XMLHttpRequests
- 动态创建的脚本元素

注：在某些受限制的情况下，允许执行字符串运算。有关详细信息，请参阅第 96 页的“[对不同沙箱中的内容的代码限制](#)”。

Adobe 维护了一个已知的支持应用程序安全沙箱的 Ajax 框架列表，可以通过http://www.adobe.com/go/airappsandboxframeworks_cn 访问该列表。

以下部分介绍了如何针对应用程序沙箱中运行的代码改写脚本，以避免这些不安全的 JavaScript 错误和无提示失败。

将应用程序内容映射到其它沙箱

在大多数情况下，可以改写或重构应用程序以避免与安全相关的 JavaScript 错误。但是，如果无法进行改写或重构，则可以采用第 57 页的“[将应用程序内容加载到非应用程序沙箱](#)”中介绍的技术将应用程序内容加载到其它沙箱。如果该内容还必须访问 AIR API，则可以按照第 58 页的“[设置沙箱桥接口](#)”中的说明创建一个沙箱桥。

eval() 函数

在应用程序沙箱中，[eval\(\)](#) 函数只能用在页面 [load](#) 事件之前或用在 [load](#) 事件处理函数中。在页面加载之后，调用 [eval\(\)](#) 将不会执行代码。但是，在下面的情况下，可以通过改写代码来避免使用 [eval\(\)](#)。

将属性分配给对象

不再通过分析字符串来构建属性存取器：

```
eval("obj." + propName + " = " + val);
```

而是使用中括号记号来访问属性：

```
obj[propName] = val;
```

创建从上下文中获得变量的函数

将如下所示的语句：

```
function compile(var1, var2){  
    eval("var fn = function(){ this."+var1+"(var2) }");  
    return fn;  
}
```

替换为：

```
function compile(var1, var2){  
    var self = this;  
    return function(){ self[var1](var2) };  
}
```

创建使用类名称作为字符串参数的对象

假设有一个 JavaScript 类，其代码定义如下：

```
var CustomClass =  
{  
    Utils:  
    {  
        Parser: function(){ alert('constructor') }  
    },  
    Data:  
    {  
    }  
};  
var constructorClassName = "CustomClass.Utils.Parser";
```

最简单的实例创建方法是使用 eval()：

```
var myObj;  
eval('myObj=new ' + constructorClassName +'()')
```

但是，通过分析类名称的各个部分并使用中括号表示法构建新对象，可以避免调用 eval()：

```

function getter(str)
{
    var obj = window;
    var names = str.split('.');
    for(var i=0;i<names.length;i++){
        if(typeof obj[names[i]]=='undefined'){
            var undefstring = names[0];
            for(var j=1;j<=i;j++)
                undefstring+=".+"+names[j];
            throw new Error(undefstring+" is undefined");
        }
        obj = obj[names[i]];
    }
    return obj;
}

```

若要创建实例，可使用：

```

try{
    var Parser = getter(constructorClassName);
    var a = new Parser();
} catch(e){
    alert(e);
}

```

setTimeout() 和 setInterval()

将作为处理函数传递的字符串替换为函数引用或对象。例如，将以下语句：

```
setTimeout("alert('Timeout')", 100);
```

替换为：

```
setTimeout(function(){alert('Timeout')}, 100);
```

或者，如果函数要求 this 对象由调用方设置，则将以下语句：

```
this.appTimer = setInterval("obj.customFunction()", 100);
```

替换为：

```

var _self = this;
this.appTimer = setInterval(function(){obj.customFunction.apply(_self);}, 100);

```

Function 构造函数

对 new Function(param, body) 的调用可以替换为内联函数声明或仅在处理完页面 load 事件之前使用。

javascript: URL

在应用程序沙箱中，将忽略在使用 javascript: URL 方案的链接中定义的代码。不会生成任何不安全的 JavaScript 错误。可以将如下所示的使用 javascript: URL 的链接：

```
<a href="javascript:code()">Click Me</a>
```

替换为：

```
<a href="#" onclick="code()">Click Me</a>
```

innerHTML 和 outerHTML 语句中通过 onevent 属性分配的事件回调

使用 innerHTML 或 outerHTML 向文档的 DOM 中添加元素时，将忽略在语句内分配的任何事件回调（如 onclick 或 onmouseover）。不会生成任何安全错误。可以改为向新元素分配 id 属性，并使用 addEventListener() 方法设置事件处理函数回调函数。

例如，在文档中给定一个目标元素，如下所示：

```
<div id="container"></div>
```

将如下所示的语句：

```
document.getElementById('container').innerHTML =  
'<a href="#" onclick="code()">Click Me.</a>';
```

替换为：

```
document.getElementById('container').innerHTML = '<a href="#" id="smith">Click Me.</a>';  
document.getElementById('smith').addEventListener("click", function() { code(); });
```

从应用程序安装目录外部加载 JavaScript 文件

不允许从应用程序沙箱外部加载脚本文件。不会生成任何安全错误。在应用程序沙箱中运行的所有脚本文件都必须安装在应用程序目录中。若要在页面中使用外部脚本，必须将页面映射到其它沙箱。请参阅第 57 页的“[将应用程序内容加载到非应用程序沙箱](#)”。

document.write() 和 document.writeln()

处理页面 load 事件之后，将忽略对 document.write() 或 document.writeln() 的调用。不会生成任何安全错误。作为一种替代方法，可以加载新文件，或者使用 DOM 操作技术替换文档的正文。

load 事件之前或 load 事件处理函数中的同步 XMLHttpRequests

在页面 load 事件之前或在 load 事件处理函数中启动的同步 XMLHttpRequests 不会返回任何内容。可以启动异步 XMLHttpRequests，但在 load 事件之前不会返回内容。在处理 load 事件之后，同步 XMLHttpRequests 才能正常工作。

动态创建的脚本元素

将忽略动态创建的脚本元素，如使用 innerHTML 或 document.createElement() 方法创建的元素。

通过 JavaScript 访问 AIR API 类

除 Webkit 的标准元素和扩展元素之外，HTML 和 JavaScript 代码还可以访问运行时提供的主机类。通过这些类，可以访问 AIR 提供的高级功能，包括：

- 访问文件系统
- 使用本地 SQL 数据库
- 控制应用程序和窗口菜单
- 访问网络套接字
- 使用用户定义的类和对象
- 声音功能

例如，AIR 文件 API 包含一个 File 类，该类包含在 flash.filesystem 包中。可以在 JavaScript 中创建一个如下所示的 File 对象：

```
var myFile = new window.runtime.flash.filesystem.File();
```

runtime 对象是一个特殊的 JavaScript 对象，可用于在 AIR 应用程序沙箱中运行的 HTML 内容。使用该对象，可以通过 JavaScript 访问运行时类。flash 对象的 runtime 属性提供了对 Flash 包的访问。同样，flash.filesystem 对象的 runtime 属性提供了对 flash.filesystem 包（此包包含 File 类）的访问。包是一种对 ActionScript 中使用的类进行组织的方式。

注：不会自动向 frame 或 iframe 中加载的页面的窗口对象添加 runtime 属性。但是，只要子级文档位于应用程序沙箱中，子级文档就可以访问父级文档的 runtime 属性。

由于运行时类的包结构要求开发人员键入长字符串的 JavaScript 代码字符串（如 window.runtime.flash.desktop.NativeApplication）来访问各个类，因此，AIR SDK 提供了一个 AIRAliases.js 文件，使用该文件，可以更方便地访问运行时类（例如，只需键入 air.NativeApplication 即可）。

本指南主要讨论 AIR API 类。对于 HTML 开发人员可能感兴趣的其它 Flash Player API 类，将在《针对 HTML 开发人员的 Adobe AIR 语言参考》中予以介绍。SWF (Flash Player) 内容所使用的语言为 ActionScript。但是，JavaScript 和 ActionScript 语法是类似的。（它们都基于 ECMAScript 语言版本。）JavaScript（在 HTML 内容中）和 ActionScript（在 SWF 内容中）均包含所有内置类。

注：JavaScript 代码无法使用 Dictionary、XML 和 XMLList 类，但这些类在 ActionScript 中是可用的。

注：有关详细信息，请参阅第 112 页的“[ActionScript 3.0 类、包和命名空间](#)”以及第 110 页的“[针对 JavaScript 开发人员的 ActionScript 基础知识](#)”。

使用 AIRAliases.js 文件

运行时类采用包结构的形式进行组织，如下所示：

- window.runtime.flash.desktop.NativeApplication
- window.runtime.flash.desktop.ClipboardManager
- window.runtime.flash.filesystem FileStream
- window.runtime.flash.data.SQLDatabase

AIR SDK 中包含的 AIRAliases.js 文件提供了“别名”定义，使用这些定义，只需键入很短的内容就可以访问运行时类。例如，只需键入以下内容就可以访问上面列出的类：

- air.NativeApplication
- air.Clipboard
- air.FileStream
- air.SQLDatabase

此列表只列出了 AIRAliases.js 文件中一小部分类。《针对 HTML 开发人员的 Adobe AIR 语言参考》中提供了类和包级别函数的完整列表。

除了常用的运行时类之外，AIRAliases.js 文件还包括以下常用包级别函数的别名：window.runtime.trace()、window.runtime.flash.net.navigateToURL() 和 window.runtime.flash.net.sendToURL()，这些函数对应的别名为 air.trace()、air.navigateToURL() 和 air.sendToURL()。

若要使用 AIRAliases.js 文件，请在 HTML 页中包括以下 script 引用：

```
<script src="AIRAliases.js"></script>
```

根据需要调整 src 引用中的路径。

重要说明：如果未明确声明，此文档中的 JavaScript 示例代码均假定已在 HTML 页中包含 AIRAliases.js 文件。

关于 AIR 中的 URL

在 AIR 中运行的 HTML 内容中，可以在定义 img、frame、iframe 和 script 标签的 src 属性时、在 link 标签的 href 属性中以及可以提供 URL 的任何其它地方，使用以下任意 URL 方案：

URL 方案	说明	示例
file	相对于文件系统根目录的相对路径。	file:///c:/AIR Test/test.txt
app	相对于应用程序根安装目录的相对路径。	app:/images
app-storage	相对于应用程序存储目录的相对路径。AIR 为安装的每个应用程序都定义了唯一的应用程序存储目录，这些目录对于存储特定于各个应用程序的数据非常有用。	app-storage:/settings/prefs.xml
http	标准 HTTP 请求。	http://www.adobe.com
https	标准 HTTPS 请求。	https://secure.example.com

有关在 AIR 中使用 URL 方案的详细信息，请参阅第 292 页的“[在 URL 中使用 AIR URL 方案](#)”。

许多 AIR API（包括 File、Loader、URLStream 和 Sound 类）使用的是 URLRequest 对象，而不是包含 URL 的字符串。URLRequest 对象本身使用字符串进行初始化，在字符串中可以使用以上任意 URL 方案。例如，以下语句创建的 URLRequest 对象可用于请求 Adobe 主页：

```
var urlReq = new air.URLRequest("http://www.adobe.com/");
```

有关 URLRequest 对象的信息，请参阅 第 290 页的“[URL 请求和网络](#)”。

在 HTML 中嵌入 SWF 内容

在 AIR 应用程序中，可以像在浏览器中那样，在 HTML 中嵌入 SWF 内容。使用 object 标签、embed 标签或同时二者可嵌入 SWF 内容。

注：常见的 Web 开发做法是同时使用 object 标签和 embed 标签在 HTML 页中显示 SWF 内容。此做法在 AIR 中毫无益处。您可以只使用 W3C 标准的 object 标签在 AIR 中显示内容。同时，对于浏览器中显示的 HTML 内容，您可以继续同时使用 object 和 embed 标签（如果需要）。

以下示例说明如何使用 HTML object 标签在 HTML 内容中显示 SWF 文件。SWF 文件加载自应用程序目录，但您可以使用 AIR 支持的任何 URL 方案。（SWF 文件的加载位置决定了 AIR 放置内容的安全沙箱。）

```
<object type="application/x-shockwave-flash" width="100%" height="100%>
  <param name="movie" value="app:/SWFFile.swf"></param>
</object>
```

还可以使用脚本动态地加载内容。以下示例创建了一个 object 节点，用于显示 urlString 参数中指定的 SWF 文件。该示例将此节点添加为页面元素的子元素，并使用 elementID 参数来指定 ID：

```

<script>
function showSWF(urlString, elementID) {
    var displayContainer = document.getElementById(elementID);
    displayContainer.appendChild(createSWFOBJECT(urlString, 650, 650));
}

function createSWFOBJECT(urlString, width, height) {
    var SWFOBJECT = document.createElement("object");
    SWFOBJECT.setAttribute("type", "application/x-shockwave-flash");
    SWFOBJECT.setAttribute("width", "100%");
    SWFOBJECT.setAttribute("height", "100%");
    var movieParam = document.createElement("param");
    movieParam.setAttribute("name", "movie");
    movieParam.setAttribute("value", urlString);
    SWFOBJECT.appendChild(movieParam);
    return SWFOBJECT;
}
</script>

```

在 HTML 页中使用 ActionScript 库

AIR 对 HTML 脚本元素进行了扩展，以便页面可以导入编译的 SWF 文件中的 ActionScript 类。例如，若要导入名为 myClasses.swf 的库（位于应用程序根文件夹的 lib 子目录中），则应在 HTML 文件中包含以下 script 标签：

```
<script src="lib/myClasses.swf" type="application/x-shockwave-flash"></script>
```

重要说明：类型属性必须为 type="application/x-shockwave-flash"，才能正确加载库。

如果将 SWF 内容编译为 Flash Player 10 或 AIR 1.5 SWF，则必须将应用程序描述符文件的 XML 命名空间设置为 AIR 1.5 命名空间。有关详细信息，请参阅第 103 页的“[在应用程序描述符文件中定义属性](#)”。

在对 AIR 文件进行打包时，也必须包含 lib 目录和 myClasses.swf 文件。

通过 JavaScript Window 对象的 runtime 属性访问导入的类：

```
var libraryObject = new window.runtime.LibraryClass();
```

如果 SWF 文件中的类已组织到包中，则同时还必须包含包名称。例如，如果 LibraryClass 定义位于名为 utilities 的包中，则需要使用以下语句来创建该类的实例：

```
var libraryObject = new window.runtime.utilities.LibraryClass();
```

注：若要编译 ActionScript SWF 库使其作为 AIR 中的 HTML 页的一部分，请使用 acompc 编译器。acompcc 实用程序是 Flex 3 SDK 的一部分，有关此程序的信息，请参阅 [Flex 3 SDK 文档](#)。

从导入的 ActionScript 文件访问 HTML DOM 和 JavaScript 对象

若要在使用 <script> 标签导入页面的 SWF 文件中从 ActionScript 访问 HTML 页中的对象，请将对 JavaScript 对象（如 window 或 document）的引用传递给在 ActionScript 代码中定义的函数。在函数中使用引用来访问 JavaScript 对象（或通过传入的引用可访问的其它对象）。

例如，请看以下 HTML 页：

```
<html>
<script src="ASLibrary.swf" type="application/x-shockwave-flash"></script>
<script>
    num = 254;
    function getStatus() {
        return "OK.";
    }
    function runASFunction(window) {
        var obj = new runtime.ASClass();
        obj.accessDOM(window);
    }
</script>
<body onload="runASFunction">
    <p id="p1">Body text.</p>
</body>
</html>
```

这个简单的 HTML 页包含名为 num 的 JavaScript 变量和名为 getStatus() 的 JavaScript 函数。两者均为该页面的全局 window 对象的属性。此外，window.document 对象还包括一个名为 P 的元素（ID 为 p1）。

该页加载了一个 ActionScript 文件“ASLibrary.swf”，其中包含类 ASClass。ASClass 定义了一个名为 accessDOM() 的函数，该函数可以轻松跟踪这些 JavaScript 对象的值。accessDOM() 方法将 JavaScript 的 Window 对象作为一个参数。使用此 Window 引用，可访问页面中的其它对象，包括变量、函数和 DOM 元素，如下所示：

```
public class ASClass{
    public function accessDOM(window:*):void {
        trace(window.num); // 254
        trace(window.document.getElementById("p1").innerHTML); // Body text..
        trace(window.getStatus()); // OK.
    }
}
```

可以通过导入的 ActionScript 类同时获取和设置 HTML 页的属性。例如，以下函数设置了页面上 p1 元素的内容，并设置了页面上 foo JavaScript 变量的值：

```
public function modifyDOM(window:*):void {
    window.document.getElementById("p1").innerHTML = "Bye";
    window.foo = 66;
```

转换 Date 和 RegExp 对象

JavaScript 和 ActionScript 语言均定义了 Date 和 RegExp 类，但这些类型的对象并不能自动在两种执行上下文之间进行转换。必须将 Date 和 RegExp 对象转换为等效类型，然后才能在替代执行上下文中使用它们来设置属性或函数参数。

例如，以下 ActionScript 代码可将名为 jsDate 的 JavaScript Date 对象转换为 ActionScript Date 对象：

```
var asDate:Date = new Date(jsDate.getMilliseconds());
```

以下 ActionScript 代码可将名为 jsRegExp 的 JavaScript RegExp 对象转换为 ActionScript RegExp 对象：

```
var flags:String = "";
if (jsRegExp.dotAll) flags += "s";
if (jsRegExp.extended) flags += "x";
if (jsRegExp.global) flags += "g";
if (jsRegExp.ignoreCase) flags += "i";
if (jsRegExp.multiline) flags += "m";
var asRegExp:RegExp = new RegExp(jsRegExp.source, flags);
```

从 ActionScript 操作 HTML 样式表

当 HTMLLoader 对象调度 complete 事件之后，则可以检查和操作页面中的 CSS 样式。

例如，请看以下简单的 HTML 文档：

```
<html>
<style>
    .style1A { font-family:Arial; font-size:12px }
    .style1B { font-family:Arial; font-size:24px }
</style>
<style>
    .style2 { font-family:Arial; font-size:12px }
</style>
<body>
    <p class="style1A">
        Style 1A
    </p>
    <p class="style1B">
        Style 1B
    </p>
    <p class="style2">
        Style 2
    </p>
</body>
</html>
```

当 HTMLLoader 对象加载此内容后，可以通过 `cssRules` 数组的 `window.document.styleSheets` 数组来操作页面中的 CSS 样式，如下所示：

```
var html:HTMLLoader = new HTMLLoader();
var urlReq:URLRequest = new URLRequest("test.html");
html.load(urlReq);
html.addEventListener(Event.COMPLETE, completeHandler);
function completeHandler(event:Event):void {
    var styleSheet0:Object = html.window.document.styleSheets[0];
    styleSheet0.cssRules[0].style.fontSize = "32px";
    styleSheet0.cssRules[1].style.color = "#FF0000";
    var styleSheet1:Object = html.window.document.styleSheets[1];
    styleSheet1.cssRules[0].style.color = "blue";
    styleSheet1.cssRules[0].style.fontFamily = "Monaco";
}
```

此代码调整了 CSS 样式，从而生成如下所示的 HTML 文档：

Style 1A

Style 1B

Style 2

请记住，当 HTMLLoader 对象调度 complete 事件之后，该代码可以向页面中添加样式。

跨脚本访问不同安全沙箱中的内容

运行时安全模型将代码与不同的源隔离开来。通过跨脚本访问不同安全沙箱中的内容，可允许一个安全沙箱中的内容访问另一个沙箱中的所选属性和方法。

AIR 安全沙箱和 JavaScript 代码

AIR 强制实施同源策略，以防止一个域中的代码与另一个域中的内容进行交互。所有文件根据其来源放置在相应的沙箱中。通常，应用程序沙箱中的内容不能违反同源原则，并且不能跨脚本访问从应用程序安装目录外部加载的内容。但是，AIR 提供了一些方法，可让您跨脚本访问非应用程序内容。

一种方法是使用 `frame` 或 `iframe` 将应用程序内容映射到其它安全沙箱。从应用程序的沙箱区域加载的任何页的行为与从远程域加载该页的行为相同。例如，通过将应用程序内容映射到 `example.com` 域，该内容可以跨脚本访问从 `example.com` 加载的页。

由于此方法将应用程序内容放置到其它沙箱中，因此该内容中的代码也不再受计算出的字符串中对代码执行的限制。即使不需要跨脚本访问远程内容，也可以使用这种沙箱映射方法来减弱这些限制。当使用多个 JavaScript 框架中的一个框架或者使用依赖于计算字符串的现有代码时，采用此方法映射内容特别有用。但是，应考虑并防止运行应用程序沙箱以外的内容时可能插入和执行不可信内容的额外风险。

同时，映射到其它沙箱的应用程序内容将失去访问 AIR API 的权利，因此沙箱映射方法不能用于向在应用程序沙箱外部执行的代码公开 AIR 功能。

另一种跨脚本访问的方法，是在非应用程序沙箱中的内容与其在应用程序沙箱中的父级文档之间创建一个名为沙箱桥的接口。沙箱桥允许子级内容访问父级内容所定义的属性和方法，或允许父级内容访问子级内容所定义的属性和方法，或者两者同时允许。

最后，还可以从应用程序沙箱和其它沙箱（可选）执行跨域 XMLHttpRequest。

有关详细信息，请参阅第 68 页的“[HTML frame 和 iframe 元素](#)”、第 94 页的“[HTML 安全性](#)”以及第 63 页的“[XMLHttpRequest 对象](#)”。

将应用程序内容加载到非应用程序沙箱

若要允许应用程序内容安全地跨脚本访问从应用程序安装目录外部加载的内容，可以使用 `frame` 或 `iframe` 元素将应用程序内容加载到与外部内容相同的安全沙箱。如果不需要跨脚本访问远程内容，但仍希望加载应用程序沙箱外部的应用程序页，则可以使用同一方法，指定 `http://localhost/` 或其它某些无不利影响的值作为源域。

AIR 将向 `frame` 元素添加新的 `sandboxRoot` 和 `documentRoot` 属性，以便允许您指定是否应将加载到该框架中的应用程序文件映射到非应用程序沙箱。对于解析为 `sandboxRoot URL` 之下的路径的文件，将改为从 `documentRoot` 目录加载。出于安全方面的考虑，使用此方法加载的应用程序内容将被视为是从 `sandboxRoot URL` 实际加载的。

`sandboxRoot` 属性指定用于确定放置框架内容的沙箱和域的 URL。必须使用 `file:`、`http:` 或 `https:` URL 方案。如果您指定的是相对 URL，则内容将保留在应用程序沙箱中。

`documentRoot` 属性指定从中加载框架内容的目录。必须使用 `file:`、`app:` 或 `app-storage:` URL 方案。

以下示例对要在远程沙箱中运行的应用程序的 `sandbox` 子目录中安装的内容以及 `www.example.com` 域进行了映射：

```
<iframe
    src="http://www.example.com/local/ui.html"
    sandboxRoot="http://www.example.com/local/"
    documentRoot="app:/sandbox/">
</iframe>
```

`ui.html` 页可以使用以下脚本标签从本地的 `sandbox` 文件夹加载 javascript 文件：

```
<script src="http://www.example.com/local/ui.js"></script>
```

它还可以使用以下脚本标签从远程服务器的目录加载内容：

```
<script src="http://www.example.com/remote/remote.js"></script>
```

`sandboxRoot URL` 将遮盖远程服务器上位于相同 URL 中的所有内容。在上例中，不能访问位于 `www.example.com/local/`（或其任何子目录）的任何远程内容，这是因为 AIR 将请求重新映射到本地应用程序目录。无论是页面导航、`XMLHttpRequest` 还是采用其它内容加载手段所派生的请求，都会重新映射。

设置沙箱桥接口

如果应用程序沙箱中的内容必须访问非应用程序沙箱中的内容所定义的属性或方法，或者如果非应用程序内容必须访问应用程序沙箱中的内容所定义的属性或方法，则可以使用沙箱桥。使用任何子级文档的 `window` 对象的 `childSandboxBridge` 和 `parentSandboxBridge` 属性可创建沙箱桥。

建立子级沙箱桥

`childSandboxBridge` 属性允许子级文档向父级文档中的内容公开接口。若要公开接口，需将 `childSandbox` 属性设置为子级文档中的函数或对象。然后，可以从父级文档中的内容访问该对象或函数。以下示例显示了子级文档中运行的脚本如何向其父级文档公开包含函数和属性的对象：

```
var interface = {};
interface.calculatePrice = function(){
    return ".45 cents";
}
interface.storeID = "abc"
window.childSandboxBridge = interface;
```

如果此子级内容加载到 `iframe`（分配的 ID 为“`child`”），则可以通过读取 `frame` 的 `childSandboxBridge` 属性从父级内容来访问接口：

```
var childInterface = document.getElementById("child").contentWindow.childSandboxBridge;
air.trace(childInterface.calculatePrice()); //traces ".45 cents"
air.trace(childInterface.storeID)); //traces "abc"
```

建立父级沙箱桥

`parentSandboxBridge` 属性允许父级文档向子级文档中的内容公开接口。若要公开接口，父级文档需将子级文档的 `parentSandbox` 属性设置为父级文档中定义的函数或对象。然后，可以从子级文档中的内容访问该对象或函数。以下示例显示了父级 `frame` 中运行的脚本如何向其子级文档公开包含函数的对象：

```
var interface = {};
interface.save = function(text){
    var saveFile = air.File("app-storage:/save.txt");
    //write text to file
}
document.getElementById("child").contentWindow.parentSandboxBridge = interface;
```

使用此接口，子级 `frame` 中的内容可以将文本保存到名为 `save.txt` 的文件，但对文件系统不具备任何其它访问权利。子级内容可以调用 `save` 函数，如下所示：

```
var textToSave = "A string.";
window.parentSandboxBridge.save(textToSave);
```

应用程序内容向其它沙箱公开的接口应越窄越好。应考虑到非应用程序内容本身并不可靠，因为它可能遭到意外或恶意代码注入。必须采取适当的防护措施，以防止误用通过父级沙箱桥公开的接口。

在页面加载过程中访问父级沙箱桥

为了使子级文档中的脚本能够访问父级沙箱桥，必须先设置沙箱桥，然后才能运行脚本。创建新页 DOM 之后，在分析任何脚本或添加 DOM 元素之前，`Window`、`frame` 和 `iframe` 对象将调度 `dominitialize` 事件。可以使用 `dominitialize` 事件按照适当的页面构造顺序尽早建立沙箱桥，以便页面中定义的所有脚本均能访问该沙箱桥。

以下示例说明如何创建父级沙箱桥，以响应从子级 `frame` 调度的 `dominitialize` 事件：

```
<html>
<head>
<script>
var bridgeInterface = {};
bridgeInterface.testProperty = "Bridge engaged";
function engageBridge() {
    document.getElementById("sandbox").contentWindow.parentSandboxBridge = bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
        src="http://www.example.com/air/child.html"
        documentRoot="app:/"
        sandboxRoot="http://www.example.com/air/"
        ondominitialize="engageBridge()"/>
</body>
</html>
```

以下 `child.html` 文档说明子级内容如何访问父级沙箱桥:

```
<html>
<head>
<script>
    document.write(window.parentSandboxBridge.testProperty);
</script>
</head>
<body></body>
</html>
```

若要侦听子级窗口（而非框架）上的 `dominitialize` 事件，必须向由 `window.open()` 函数创建的新子级 `window` 对象添加侦听器:

```
var childWindow = window.open();
childWindow.addEventListener("dominitialize", engageBridge());
childWindow.document.location = "http://www.example.com/air/child.html";
```

在这种情况下，无法将应用程序内容映射到非应用程序沙箱。只有在从应用程序目录外部加载 `child.html` 时，此方法才有用。仍可将窗口中的应用程序内容映射到非应用程序沙箱，但必须首先加载一个中间页，在中间页中使用框架来加载子级文档并将其映射到所需的沙箱。

如果使用 `HTMLLoader` 类的 `createRootWindow()` 函数来创建窗口，则新窗口不是从中调用 `createRootWindow()` 的文档的子级。因此，无法从调用窗口建立到加载到新窗口中的非应用程序内容的沙箱桥。而是必须在新窗口中加载一个中间页，在中间页中使用框架来加载子级文档。这样，就可以在新窗口的父级文档与加载到框架中的子级文档之间建立沙箱桥。

第 11 章：关于 HTML 环境

在 HTML 内容中使用 AIR API 是完全可选的。您可以完全使用 HTML 和 JavaScript 编写 AIR 应用程序。大多数现有 HTML 应用程序只需少量更改即可运行（假定它们使用的 HTML、CSS、DOM 和 JavaScript 功能与 WebKit 兼容）。

AIR 授予您对应用程序外观的完全控制权限。您可以使应用程序的外观类似于本机桌面应用程序。还可以关闭由操作系统提供的窗口边框，并实现您自己的用于移动窗口、调整窗口大小和关闭窗口的控件。您甚至可以在没有窗口的情况下运行。

由于 AIR 应用程序直接在桌面上运行，且具有对文件系统的完全访问权限，因此，对应的安全模型比典型 Web 浏览器的安全模型更加严格。在 AIR 中，只有从应用程序安装目录加载的内容才会被放置到应用程序沙箱中。应用程序沙箱具有最高级别的权限，且允许访问 AIR API。AIR 根据其他内容的来源将这些内容放置到隔离沙箱中。从文件系统加载的文件放置到本地沙箱中。使用 `http:` 或 `https:` 协议从网络加载的文件则根据远程服务器的域放置到相应沙箱中。禁止这些非应用程序沙箱中的内容访问任何 AIR API，且其运行方式与在典型 Web 浏览器中几乎一样。

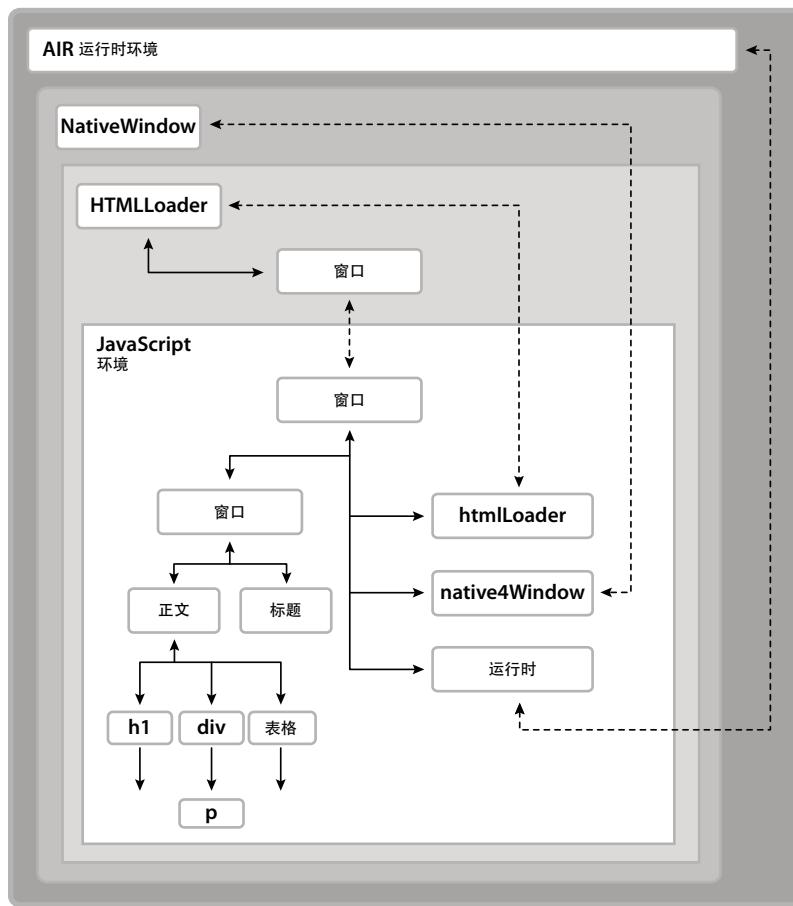
如果应用 Alpha、缩放或透明度设置，则 AIR 中的 HTML 内容不显示 SWF 或 PDF 内容。有关详细信息，请参阅[在 HTML 页中加载 SWF 或 PDF 内容时的注意事项](#)和第 118 页的“[窗口透明度](#)”。

HTML 环境概述

Adobe AIR 使用 HTML 渲染器、文档对象模型和 JavaScript 解释器提供与浏览器完全相似的 JavaScript 环境。JavaScript 环境通过 AIR HTMLLoader 类表示。在 HTML 窗口中，HTMLLoader 对象包含所有 HTML 内容，而该对象又包含在 NativeWindow 对象中。通过 NativeWindow 对象，应用程序可以为用户桌面上显示的本机操作系统窗口的属性和行为撰写脚本。

关于 JavaScript 环境及其与 AIR 之间的关系

下图演示了 JavaScript 环境和 AIR 运行时环境的关系。虽然只显示了一个本机窗口，但是一个 AIR 应程序可以包含多个窗口。（并且一个窗口可以包含多个 HTMLLoader 对象。）



JavaScript 环境具有其自己的 Document 和 Window 对象。JavaScript 代码可以通过 runtime、nativeWindow 和 htmlLoader 属性与 AIR 运行时环境交互。ActionScript 代码可以通过 HTMLLoader 对象的 window 属性与 JavaScript 环境交互，该属性是对 JavaScript Window 对象的引用。此外，ActionScript 和 JavaScript 对象均可以侦听由 AIR 和 JavaScript 对象调度的事件。

runtime 属性提供了对 AIR API 类的访问权限，使您可以新建 AIR 对象和访问类（也称为静态）成员。若要访问 AIR API，请向 runtime 属性添加该类的名称和包。例如，若要创建 File 对象，您将使用以下语句：

```
var file = new window.runtime.filesystem.File();
```

注：AIR SDK 提供了一个 JavaScript 文件，即 AIRAliases.js，该文件为最常用的 AIR 类定义了更便于使用的别名。导入该文件时，您可以使用 air.Class 简短形式替换 window.runtime.package.Class。例如，您可以使用 new air.File() () 创建 File 对象。

NativeWindow 对象提供了用于控制桌面窗口的属性。您可以使用 window.nativeWindow 属性从 HTML 页内部访问包含的 NativeWindow 对象。

HTMLLoader 对象提供了用于控制内容的加载方式和呈现方式的属性、方法和事件。您可以使用 window.htmlLoader 属性从 HTML 页内部访问父 HTMLLoader 对象。

重要说明：仅当作为应用程序的一部分安装的页作为顶级文档加载时，该页才具有 htmlLoader、nativeWindow 或 runtime 属性。将文档加载到 frame 或 iframe 中时不会添加这些属性。（只要子文档与父文档位于相同安全沙箱中，子文档即可访问父文档的这些属性。例如，帧中加载的文档可以使用 parent.runtime 访问其父级的 runtime 属性。）

关于安全性

AIR 根据原始域在安全沙箱中执行所有代码。应用程序内容限制为从应用程序安装目录加载的内容，该内容放置到应用程序沙箱中。只有在此沙箱中运行的 HTML 和 JavaScript 才能访问运行时环境和 AIR API。同时，在页 load 事件的所有处理函数返回之后，将在应用程序沙箱中阻止 JavaScript 的大多数动态计算和执行操作。

通过将应用程序页加载到 frame 或 iframe 中，并对 frame 设置特定于 AIR 的 sandboxRoot 和 documentRoot 属性，可以将该应用程序页映射到非应用程序沙箱中。通过将 sandboxRoot 值设置为实际远程域，您可以使沙箱中的内容跨脚本访问该域中的内容。当加载远程内容或撰写远程内容脚本时（例如，在 mash-up 应用程序中），使用上述方法映射页非常有用。

允许应用程序和非应用程序内容相互跨脚本访问的另一方法是创建沙箱桥，这也是向 AIR API 授予非应用程序内容访问权限的唯一方法。使用父级到子级桥，子 frame、iframe 或窗口中的内容能够访问在应用程序沙箱中定义的指定方法和属性。反之，使用子级到父级桥，应用程序内容能够访问在子级的沙箱中定义的指定方法和属性。通过设置窗口对象的 parentSandboxBridge 和 childSandboxBridge 属性可以建立沙箱桥。有关详细信息，请参阅第 94 页的“[HTML 安全性](#)”以及第 68 页的“[HTML frame 和 iframe 元素](#)”。

关于插件和嵌入对象

AIR 支持 Adobe® Acrobat® 插件。用户必须安装有 Acrobat 或 Adobe® Reader® 8.1（或更高版本）才能显示 PDF 内容。HTMLLoader 对象提供了用于查看用户系统是否可以显示 PDF 的属性。SWF 文件内容也可以在 HTML 环境中显示，但 AIR 中构建有此功能，因此无需使用外部插件。

AIR 中不支持任何其他 Webkit 插件。

另请参阅

[第 94 页的“HTML 安全性”](#)

[第 63 页的“HTML 沙箱”](#)

[第 68 页的“HTML frame 和 iframe 元素”](#)

[第 67 页的“JavaScript Window 对象”](#)

[第 63 页的“XMLHttpRequest 对象”](#)

[第 248 页的“添加 PDF 内容”](#)

AIR 和 Webkit 扩展

Adobe AIR 和 Safari Web 浏览器均使用开放源代码 Webkit 引擎。AIR 添加了若干扩展功能，以便允许访问运行时类和对象，并增强了安全性。此外，Webkit 自身还针对 HTML、CSS 和 JavaScript 添加了 W3C 标准中不包括的功能。

此处仅包含 AIR 新增功能和最显著的 Webkit 扩展功能；有关非标准 HTML、CSS 和 JavaScript 的其它文档，请参阅 www.webkit.org 和 developer.apple.com。有关标准信息，请参阅 W3C 网站。Mozilla 还提供了有关 HTML、CSS 和 DOM 主题的[重要一般参考](#)（当然，Webkit 引擎不同于 Mozilla 引擎）。

注：AIR 不支持下列标准和扩展 WebKit 功能：JavaScript Window 对象 print() 方法；除 Acrobat 或 Adobe Reader 8.1+ 以外的插件；可缩放矢量图形 (SVG) 的 CSS opacity 属性。

AIR 中的 JavaScript

AIR 对通用 JavaScript 对象的典型行为进行了若干更改。其中，很多更改都是为了在 AIR 中更方便地编写安全应用程序。同时，这些行为差异表示某些通用 JavaScript 编码模式和使用这些模式的现有 Web 应用程序在 AIR 中可能始终不会按预期方式执行。有关更正这些问题类型的信息，请参阅第 48 页的“[避免与安全相关的 JavaScript 错误](#)”。

HTML 沙箱

AIR 根据内容的原始位置将该内容放置到隔离沙箱中。沙箱规则与大多数 Web 浏览器实现的具有相同原始位置的策略一致，并且与 Adobe Flash Player 实现的沙箱规则一致。此外，AIR 还提供了一个包含并保护应用程序内容的新应用程序沙箱类型。有关在开发 AIR 应用程序时可能遇到的沙箱类型的详细信息，请参阅第 92 页的“[沙箱](#)”。

只有在应用程序沙箱中运行的 HTML 和 JavaScript 才能访问运行时环境和 AIR API。但同时，出于安全原因，动态计算和执行各种形式的 JavaScript 在应用程序沙箱内大幅受限。无论您的应用程序实际上是否从服务器直接加载信息，这些限制都将发挥作用。（即使是文件内容、粘贴的字符串和直接用户输入也可能不可靠。）

页内容的原始位置确定要将该内容放置到哪个沙箱。只有从应用程序目录（app: URL 方案引用的安装目录）加载的内容放置在应用程序沙箱中。从文件系统加载的内容则放置到与本地文件系统内容交互的沙箱或受信任的本地沙箱中，以便访问本地文件系统上的内容（而不是远程内容），并与其进行交互。从网络加载的内容则放置到与其原始域对应的远程沙箱中。

若要使应用程序页与远程沙箱中的内容自由交互，则可以将该页映射到远程内容所在的相同域中。例如，如果编写显示 Internet 服务的映射数据的应用程序，则可以将加载和显示该服务内容的应用程序页映射到该服务域中。用于将页映射到远程沙箱和域的属性是 frame 和 iframe HTML 元素的新增属性。

若要使非应用程序沙箱中的内容安全地使用 AIR 功能，则可以设置父沙箱桥。若要使应用程序内容安全地调用其他沙箱中的内容的方法并访问其属性，则可以设置子沙箱桥。此处所述的安全性意味着远程内容不能意外获取对未显式公开的对象、属性或方法的引用。通过沙箱桥只能传递简单数据类型、函数和匿名对象。但是，您仍然必须避免显式公开潜在的危险函数。例如，如果公开的接口允许远程内容读取或写入用户系统中任意位置的文件，则可能会使远程内容严重危害用户。

JavaScript eval() 函数

加载一页完毕后，应用程序沙箱内就会限制使用 eval() 函数。在某些情况下允许使用该函数，以便可以安全地分析 JSON 格式数据，但是，生成可执行语句的任何计算将生成错误。第 96 页的“[对不同沙箱中的内容的代码限制](#)”介绍了允许使用 eval() 函数的情况。

函数构造函数

在应用程序沙箱中，可以在完成加载页之前使用函数构造函数。在所有页 load 事件处理函数完成之后，无法创建新的函数。

加载外部脚本

应用程序沙箱中的 HTML 页无法使用 script 标签从应用程序目录外部加载 JavaScript 文件。为使应用程序中的页能够从应用程序目录外部加载脚本，必须将该页映射到非应用程序沙箱。

XMLHttpRequest 对象

AIR 提供了应用程序可用于执行数据请求的 XMLHttpRequest (XHR) 对象。下面的示例演示简单数据请求：

```
xmlhttp = new XMLHttpRequest();
xmlhttp.open("GET", "http://www.example.com/file.data", true);
xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4) {
        //do something with data...
    }
}
xmlhttp.send(null);
```

与浏览器不同，AIR 允许在应用程序沙箱中运行的内容请求任何域中的数据。对于包含 JSON 字符串的 XHR 结果，除非该结果还包含可执行代码，否则可以计算出该结果的数据对象。如果 XHR 结果中存在可执行语句，则会引发错误，且计算尝试失败。

若要防止从远程源意外注入代码，在完成加载页之前执行同步 XHR 时将返回空结果。异步 XHR 将始终在加载页之后返回。

默认情况下，AIR 阻止在非应用程序沙箱中执行跨域 XMLHttpRequest。应用程序沙箱中的父窗口可以选择在包含非应用程序沙箱内容的子 frame 中允许跨域请求，方法是在包含非应用程序沙箱内容的 frame 或 iframe 元素中将 AIR 添加的 allowCrossDomainXHR 属性设置为 true：

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    allowCrossDomainXHR="true"
/>
```

注：还可以根据需要使用 AIR URLStream 类下载数据。

如果从包含已映射到远程沙箱的应用程序内容的 frame 或 iframe 对远程服务器调度 XMLHttpRequest，请确保映射 URL 不会遮蔽在 XHR 中使用的服务器地址。例如，请考虑以下 iframe 定义，该定义将应用程序内容映射到 example.com 域所对应的远程沙箱：

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    documentRoot="app:/sandbox/"
    sandboxRoot="http://www.example.com/"
    allowCrossDomainXHR="true"
/>
```

由于 sandboxRoot 属性重新映射 www.example.com 地址的根 URL，因此所有请求都将从应用程序目录加载，而不是从远程服务器加载。无论请求是派生自页导航还是 XMLHttpRequest，都将重新映射这些请求。

若要避免意外阻止对远程服务器的数据请求，请将 sandboxRoot 映射到远程 URL 的子目录，而不是根目录。该目录不一定存在。例如，若要允许从远程服务器加载对 www.example.com 的请求，而不是从应用程序目录加载，请将上面的 iframe 更改为以下内容：

```
<iframe id="mashup"
    src="http://www.example.com/map.html"
    documentRoot="app:/sandbox/"
    sandboxRoot="http://www.example.com/air/"
    allowCrossDomainXHR="true"
/>
```

在本例中，仅在本地加载 air 子目录中的内容。

有关沙箱映射的详细信息，请参阅第 68 页的“[HTML frame 和 iframe 元素](#)”以及第 94 页的“[HTML 安全性](#)”。

Canvas 对象

Canvas 对象定义用于绘制直线、弧形、椭圆和多边形等几何形状的 API。若要使用 Canvas API，请首先向文档添加一个 Canvas 元素，然后使用 JavaScript Canvas API 在该元素中绘制。在许多其他方面，Canvas 对象的行为与图像相似。

下面的示例使用 Canvas 对象绘制一个三角形：

```

<html>
<body>
<canvas id="triangleCanvas" style="width:40px; height:40px;"></canvas>
<script>
    var canvas = document.getElementById("triangleCanvas");
    var context = canvas.getContext("2d");
    context.lineWidth = 3;
    context.strokeStyle = "#457232";
    context.beginPath();
        context.moveTo(5,5);
        context.lineTo(35,5);
        context.lineTo(20,35);
        context.lineTo(5,5);
        context.lineTo(6,5);
    context.stroke();
</script>
</body>
</html>

```

有关 Canvas API 的更多文档, 请参阅 Apple 提供的 [Safari JavaScript Reference](#)。请注意, Webkit 项目近期已开始更改 Canvas API, 以便根据 Web 超文本应用程序技术工作组 (WHATWG) 和 W3C 建议的 [HTML 5 工作草案](#) 标准化。因此, Safari JavaScript Reference 中的某些文档可能与 AIR 中提供的 Canvas 版本不一致。

Cookie

在 AIR 应用程序中, 只有远程沙箱中的内容 (从 `http:` 和 `https:` 源加载的内容) 才能使用 cookie (即 `document.cookie` 属性)。在应用程序沙箱中, AIR API 提供了其他几种存储永久数据的方式 (例如, `EncryptedLocalStore` 和 `FileStream` 类)。

Clipboard 对象

WebKit Clipboard API 由以下事件驱动: `copy`、`cut` 和 `paste`。在这些事件中传递的事件对象通过 `clipboardData` 属性提供对剪贴板的访问。使用 `clipboardData` 对象的以下方法可以读取或写入剪贴板数据:

方法	说明
<code>clearData(mimeType)</code>	清除剪贴板数据。将 <code>mimeType</code> 参数设置为要清除的数据的 MIME 类型。
<code>getData(mimeType)</code>	获取剪贴板数据。该方法只能在 <code>paste</code> 事件的处理函数中调用。将 <code>mimeType</code> 参数设置为要返回的数据的 MIME 类型。
<code>setData(mimeType, data)</code>	将数据复制到剪贴板。将 <code>mimeType</code> 参数设置为数据的 MIME 类型。

应用程序沙箱外部的 JavaScript 代码只能通过上述事件访问剪贴板。但是, 应用程序沙箱中的内容可以使用 AIR Clipboard 类直接访问系统剪贴板。例如, 您可以使用以下语句获取剪贴板上的文本格式数据:

```
var clipping = air.Clipboard.generalClipboard.getData("text/plain",
    air.ClipboardTransferMode.ORIGINAL_ONLY);
```

有效的数据 MIME 类型为:

MIME 类型	值
文本	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
位图	"image/x-vnd.adobe.air.bitmap"
文件列表	"application/x-vnd.adobe.air.file-list"

重要说明：只有应用程序沙箱中的内容才能访问剪贴板上的文件数据。如果非应用程序内容尝试访问剪贴板上的文件对象，则会引发安全错误。

有关使用剪贴板的详细信息，请参阅第 192 页的“[复制和粘贴](#)”以及 [Using the Pasteboard from JavaScript \(Apple 开发人员中心\)](#)。

拖放

进出 HTML 的拖放动作生成下列 DOM 事件：dragstart、drag、dragend、dragenter、dragover、dragleave 和 drop。在这些事件中传递的事件对象通过 dataTransfer 属性提供对被拖动数据的访问。dataTransfer 属性引用的对象提供与剪贴板事件关联的 clipboardData 对象相同的方法。例如，您可以使用以下函数获取 drop 事件中的文本格式数据：

```
function onDrop(dragEvent) {
    return dragEvent.dataTransfer.getData("text/plain",
        air.ClipboardTransferMode.ORIGINAL_ONLY);
}
```

dataTransfer 对象包含下列重要成员：

成员	说明
clearData(mimeType)	清除数据。将 mimeType 参数设置为要清除的数据表示形式的 MIME 类型。
getData(mimeType)	获取拖动的数据。该方法只能在 drop 事件的处理函数中调用。将 mimeType 参数设置为要获取的数据的 MIME 类型。
setData(mimeType, data)	设置要拖动的数据。将 mimeType 参数设置为数据的 MIME 类型。
类型	一个字符串数组，其中包含 dataTransfer 对象中当前可用的所有数据表示形式的 MIME 类型。
effectsAllowed	指定是否可以复制、移动、链接拖动数据，或者是否可以对拖动数据执行任何组合操作。设置 effectsAllowed 事件的处理函数中的 dragstart 属性。
dropEffect	指定拖动目标支持哪些允许的拖动效果。设置 dropEffect 事件的处理函数中的 dragEnter 属性。拖动期间，光标将发生更改，以便指示在用户释放鼠标后将产生的效果。如果未指定任何 dropEffect，则选择 effectsAllowed 属性效果。复制效果的优先级高于移动效果，而移动效果自身的优先级又高于链接效果。用户可以使用键盘修改默认优先级。

有关向 AIR 应用程序添加拖放支持的详细信息，请参阅第 185 页的“[拖放](#)”和 [Using the Drag-and-Drop from JavaScript \(Apple 开发人员中心\)](#)。

innerHTML 和 outerHTML 属性

对于在应用程序沙箱中运行的内容，AIR 会对 innerHTML 和 outerHTML 属性的使用施加一些安全限制。在执行页 load 事件之前以及在执行任何 load 事件处理函数期间，innerHTML 和 outerHTML 属性的使用不受限制。但是，一旦完成页加载，则只能使用 innerHTML 或 outerHTML 属性向文档添加静态内容。分配给 innerHTML 或 outerHTML 的字符串中计算结果为可执行代码的任何语句都将被忽略。例如，如果在元素定义中包含事件回调属性，则不会添加事件监听器。同样，不会计算嵌入的 <script> 标签。有关详细信息，请参阅第 94 页的“[HTML 安全性](#)”。

Document.write() 和 Document.writeln() 方法

在执行页的 load 事件之前，可以在应用程序沙箱中不受限制地使用 write() 和 writeln() 方法。但是，一旦完成页加载，调用上述任一方法将不会清除页或创建新页。在非应用程序沙箱中，在加载一页完毕之后调用 document.write() 或 writeln() 将清除当前页并打开一个新的空白页，这与大多数 Web 浏览器相同。

Document.designMode 属性

将 `document.designMode` 属性设置为值 `on` 可以编辑文档中的所有元素。内置编辑器支持包括文本编辑、复制、粘贴和拖放。

将 `designMode` 设置为 `on` 等效于将 `body` 元素的 `contentEditable` 属性设置为 `true`。您可以对大多数 HTML 元素使用 `contentEditable` 属性，以便定义可以编辑文档的哪些部分。有关其他信息，请参阅第 71 页的“[HTML contentEditable 属性](#)”。

unload 事件（对于 body 和 frameset 对象）

在窗口（包括应用程序主窗口）的顶级 frameset 或 `body` 标签中，切勿使用 `unload` 事件响应要关闭的窗口（或应用程序），而应使用 `NativeApplication` 对象的 `exiting` 事件检测关闭某一应用程序的时间。或者使用 `NativeWindow` 对象的 `closing` 事件检测关闭某一窗口的时间。例如，用户关闭应用程序时，下面的 JavaScript 代码将显示消息（“Goodbye.”）：

```
var app = air.NativeApplication.nativeApplication;
app.addEventListener(air.Event.EXITING, closeHandler);
function closeHandler(event)
{
    alert("Goodbye.");
}
```

但是，脚本能够成功响应因导航 `frame`、`iframe` 或顶级窗口内容而引起的 `unload` 事件。

注：Adobe AIR 的将来版本可能会删除这些限制。

JavaScript Window 对象

`Window` 对象在 JavaScript 执行上下文中保持为全局对象。在应用程序沙箱中，AIR 向 JavaScript `Window` 对象添加了新属性，以便提供对 AIR 内置类和重要主机对象的访问。此外，某些方法和属性的行为会有所不同，这取决于它们是否位于应用程序沙箱中。

Window.runtime 属性 通过 `runtime` 属性，您可以从应用程序沙箱内部实例化和使用内置运行时类。这些类包括 AIR 和 Flash Player API，但不包括 Flex 框架等。例如，以下语句可创建一个 AIR 文件对象：

```
var preferencesFile = new window.runtime.flash.filesystem.File();
```

AIR SDK 提供的 `AIRAliases.js` 文件所包含的别名定义使您可以缩短这些引用。例如，将 `AIRAliases.js` 导入到页后，可以使用以下语句创建 `File` 对象：

```
var preferencesFile = new air.File();
```

`window.runtime` 属性仅针对应用程序沙箱中的内容和带有 `frame` 或 `iframe` 的父页文档定义。

请参阅第 52 页的“[使用 `AIRAliases.js` 文件](#)”。

Window.nativeWindow 属性 `nativeWindow` 属性提供对基础本机窗口对象的引用。使用该属性，您可以为屏幕位置、大小和可见性等窗口函数和属性撰写脚本，并处理关闭、调整大小和移动等窗口事件。例如，以下语句可关闭窗口：

```
window.nativeWindow.close();
```

注：`NativeWindow` 对象提供的窗口控制功能与 JavaScript `Window` 对象提供的功能重叠。在这种情况下，您可以根据需要选择其中一种方法。

`window.nativeWindow` 属性仅针对应用程序沙箱中的内容和带有 `frame` 或 `iframe` 的父页文档定义。

Window.htmlLoader 属性 `htmlLoader` 属性提供对包含 HTML 内容的 AIR `HTMLLoader` 对象的引用。使用该属性，您可以为 HTML 环境的外观和行为撰写脚本。例如，您可以使用 `htmlLoader.paintsDefaultBackground` 属性确定该控件是否绘制默认的白色背景：

```
window.htmlLoader.paintsDefaultBackground = false;
```

注：`HTMLLoader` 对象自身具有一个 `window` 属性，该属性引用该对象包含的 HTML 内容的 JavaScript `Window` 对象。您可以使用该属性通过对包含 `HTMLLoader` 的引用访问 JavaScript 环境。

window.htmlLoader 属性仅针对应用程序沙箱中的内容和带有 frame 或 iframe 的父页文档定义。

Window.parentSandboxBridge 和 Window.childSandboxBridge 属性 使用 parentSandboxBridge 和 childSandboxBridge 属性，您可以定义父帧和子帧之间的接口。有关详细信息，请参阅第 56 页的“[跨脚本访问不同安全沙箱中的内容](#)”。

Window.setTimeout() 和 Window.setInterval() 函数 AIR 对 setTimeout() 和 setInterval() 函数在应用程序沙箱内的使用施加了一些安全限制。当调用 setTimeout() 或 setInterval() 时，您不能将要执行的代码定义为字符串。您必须使用函数引用。有关详细信息，请参阅第 50 页的“[setTimeout\(\) 和 setInterval\(\)](#)”。

Window.open() 函数 open() 方法由非应用程序沙箱中运行的代码调用时，如果调用该方法是用户交互（例如鼠标单击或按键）的结果，则该方法在被调用时仅打开一个窗口。此外，窗口标题采用应用程序标题作为前缀，以免远程内容打开的窗口模拟应用程序打开的窗口。有关详细信息，请参阅第 98 页的“[调用 JavaScript window.open\(\) 方法的限制](#)”。

air.NativeApplication 对象

NativeApplication 对象提供有关应用程序状态的信息，调度若干重要应用程序级别的事件，并提供用于控制应用程序行为的有用函数。将自动创建 NativeApplication 对象的单个实例，并且可通过用户定义的 NativeApplication.nativeApplication 属性访问该实例。

若要通过 JavaScript 代码访问该对象，您可以使用：

```
var app = window.runtime.flash.desktop.NativeApplication.nativeApplication;
```

或者，如果已导入 AIRAliases.js 脚本，则可以使用以下简短形式：

```
var app = air.NativeApplication.nativeApplication;
```

NativeApplication 对象只能从应用程序沙箱内部访问。与操作系统交互第 286 页的“[使用运行时和操作系统信息](#)”详细介绍了 NativeApplication 对象。

JavaScript URL 方案

应用程序沙箱内不准执行以 JavaScript URL 方案（如以 href="javascript:alert('Test')") 定义的代码。不引发错误。

HTML 扩展

AIR 和 WebKit 定义了一些非标准的 HTML 元素和属性，其中包括：

第 68 页的“[HTML frame 和 iframe 元素](#)”

第 70 页的“[HTML Canvas 元素](#)”

第 70 页的“[HTML 元素事件处理函数](#)”

HTML frame 和 iframe 元素

AIR 向应用程序沙箱中的内容的 frame 和 iframe 元素添加了以下新属性：

sandboxRoot 属性 sandboxRoot 属性为帧的 src 属性指定的文件指定一个替代非应用程序原始域。该文件加载到与指定域对应的非应用程序沙箱中。该文件中的内容和从指定域加载的内容可以相互跨脚本访问对方。

重要说明：如果将 sandboxRoot 的值设置为该域的基本 URL，则从应用程序目录而不是远程服务器加载对该域中的内容的所有请求（无论相应请求是通过页导航、 XMLHttpRequest 还是任何其他内容加载方式生成）。

documentRoot 属性 documentRoot 属性指定本地目录，将通过该目录加载解析到 sandboxRoot 指定的位置内文件的 URL。

当解析帧的 src 属性中或加载到帧中的内容中的 URL 时，与 sandboxRoot 中的指定值匹配的 URL 部分将替换为 documentRoot 中的指定值。因此，在以下 frame 标签中：

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/">
```

child.html 从应用程序安装文件夹的 `sandbox` 子目录加载。child.html 中的相对 URL 基于 `sandbox` 目录进行解析。请注意，在此帧中无法访问位于 `www.example.com/air` 的远程服务器上的任何文件，这是因为 AIR 将尝试从 `app:/sandbox/` 目录加载这些文件。

allowCrossDomainXHR 属性 在帧起始标签中加入 `allowCrossDomainXHR="allowCrossDomainXHR"`，以使该帧中的内容可以对任何远程域发出 `XMLHttpRequest`。默认情况下，非应用程序内容只能对其自身的原始域执行这样的请求。允许跨域 XHR 涉及严重安全影响。页中的代码能够与任何域交换数据。如果由于某种原因向页注入恶意内容，则会削弱当前沙箱中可供代码访问的任何数据的安全性。仅当确实需要执行跨域数据加载时才仅针对您创建和控制的页启用跨域 XHR。此外，请仔细验证由页加载的所有外部数据，以防止代码注入或其他形式的攻击。

重要说明：如果 `allowCrossDomainXHR` 属性包含在 `frame` 或 `iframe` 元素中，则启用跨域 XHR（除非分配的值为“0”或者以字母“f”或“n”开头）。例如，将 `allowCrossDomainXHR` 设置为“deny”仍会启用跨域 XHR。如果要禁用跨域请求，请将该属性完全置于元素声明之外。

ondominitialize 属性 为帧的 `dominitialize` 事件指定事件处理函数。该事件是在创建帧的窗口和文档对象之后以及在分析任何脚本或创建文档元素之前引发的特定于 AIR 的事件。

帧会在加载序列中尽早调度 `dominitialize` 事件，以使子页中的任何脚本均可引用由 `dominitialize` 处理函数添加到子文档的对象、变量和函数。父页必须与子页位于相同沙箱中才能直接添加或访问子文档中的任何对象。但是，应用程序沙箱中的父级可以建立沙箱桥，以便与非应用程序沙箱中的内容通信。

下面的示例演示 AIR 中 `iframe` 标签的用法：

在无需映射到远程服务器上的实际域的情况下将 child.html 放置到远程沙箱中：

```
<iframe src="http://localhost/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://localhost/air/">
```

在仅允许对 child.html 执行 `XMLHttpRequest` 的情况下将 `www.example.com` 放置到远程沙箱中：

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/">
```

在允许对任何远程域执行 `XMLHttpRequest` 的情况下将 child.html 放置到远程沙箱中：

```
<iframe src="http://www.example.com/air/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="http://www.example.com/air/"
        allowCrossDomainXHR="allowCrossDomainXHR"/>
```

将 child.html 放置到只能与本地文件系统内容交互的沙箱中：

```
<iframe src="file:///templates/child.html"
        documentRoot="app:/sandbox/"
        sandboxRoot="app-storage:/templates/">
```

使用 child.html 事件建立沙箱桥，并将 `dominitialize` 放置到远程沙箱中：

```

<html>
<head>
<script>
var bridgeInterface = {};
bridgeInterface.testProperty = "Bridge engaged";
function engageBridge() {
    document.getElementById("sandbox").parentSandboxBridge = bridgeInterface;
}
</script>
</head>
<body>
<iframe id="sandbox"
        src="http://www.example.com/air/child.html"
        documentRoot="app:/"
        sandboxRoot="http://www.example.com/air/"
        ondominitialize="engageBridge()"/>
</body>
</html>

```

以下 child.html 文档说明子级内容如何访问父级沙箱桥:

```

<html>
<head>
<script>
    document.write(window.parentSandboxBridge.testProperty);
</script>
</head>
<body></body>
</html>

```

有关详细信息, 请参阅第 56 页的“[跨脚本访问不同安全沙箱中的内容](#)”和第 94 页的“[HTML 安全性](#)”。

HTML Canvas 元素

定义与 Webkit Canvas API 配合使用的绘图区。无法在标签自身中指定图形命令。若要在画布中绘制, 请使用 JavaScript 调用画布绘制方法。

```
<canvas id="drawingAtrium" style="width:300px; height:300px;"></canvas>
```

有关详细信息, 请参阅第 64 页的“[Canvas 对象](#)”。

HTML 元素事件处理函数

AIR 和 Webkit 中的 DOM 对象调度标准 DOM 事件模型中不包含的某些事件。下表列出了为这些事件指定处理函数可使用的相关事件属性:

回调属性名称	说明
oncontextmenu	当调用上下文菜单时调用, 例如通过右键单击或按住 Command 并单击所选文本。
oncopy	当复制元素中的所选内容时调用。
oncut	当剪切元素中的所选内容时调用。
ondominitialize	在创建加载到 frame 或 iframe 的文档的 DOM 之后以及在创建任何 DOM 元素或分析脚本之前调用。
ondrag	当拖动元素时调用。
ondragend	当释放拖动动作时调用。
ondragenter	当拖动动作进入元素范围时调用。

回调属性名称	说明
ondragleave	当拖动动作离开元素范围时调用。
ondragover	当拖动动作位于元素范围内时持续调用。
ondragstart	当拖动动作开始时调用。
ondrop	当在元素上释放拖动动作时调用。
onerror	当在加载元素期间出错时调用。
oninput	当在表单元素中输入文本时调用。
onpaste	将项目粘贴到元素中时调用。
onscroll	当滚动可滚动元素的内容时调用。
onsearch	当复制元素时调用 (?Apple docs correct?)
onselectstart	当开始选择时调用。

HTML contentEditable 属性

您可以向任何 HTML 元素添加 contentEditable 属性，以便允许用户编辑该元素的内容。例如，以下 HTML 示例代码将除第一个 p 元素以外的整个文档设置为可编辑。

```
<html>
<head/>
<body contentEditable="true">
    <h1>de Finibus Bonorum et Malorum</h1>
    <p contentEditable="false">Sed ut perspiciatis unde omnis iste natus error.</p>
    <p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis.</p>
</body>
</html>
```

注：如果将 document.designMode 属性设置为 on，则该文档中的所有元素都是可编辑的，而不管各元素的 contentEditable 设置如何。但是，将 designMode 设置为 off 不会禁用对 contentEditable 为 true 的元素的编辑。有关其他信息，请参阅第 67 页的“[Document.designMode 属性](#)”。

CSS 扩展

WebKit 支持多个扩展 CSS 属性。下表列出了已建立支持的扩展属性。可以在 WebKit 中使用其他非标准属性，但是 AIR 中并不完全支持这些非标准属性，这是因为这些属性在 WebKit 中尚处于开发阶段，或者是实验功能，将来可能会删除这些实验功能。

CSS 属性名称	值	说明
-webkit-border-horizontal-spacing	非负长度单位	指定边框间距的水平组件。
-webkit-border-vertical-spacing	非负长度单位	指定边框间距的垂直组件。
-webkit-line-break	after-white-space、normal	指定用于中文、日语和韩语 (CJK) 文本的换行规则。
-webkit-margin-bottom-collapse	collapse、discard、separate	定义如何折叠表单元格的下边距。
-webkit-margin-collapse	collapse、discard、separate	定义如何折叠表单元格的上下边距。
-webkit-margin-start	任意长度单位	起始边距的宽度。对于从左到右的文本，该属性将覆盖左边距。对于从右到左的文本，该属性将覆盖右边距。

CSS 属性名称	值	说明
-webkit-margin-top-collapse	collapse、discard、separate	定义如何折叠表单元格的上边距。
-webkit-nbsp-mode	normal、space	定义被包含内容中的不间断空格的行为。
-webkit-padding-start	任意长度单位	指定起始填充的宽度。对于从左到右的文本，该属性将覆盖左填充值。对于从右到左的文本，该属性将覆盖右填充值。
-webkit-rtl-ordering	logical、visual	覆盖从左到右和从右到左混合文本的默认处理。
-webkit-text-fill-color	任意指定颜色或数字颜色值	指定文本填充颜色。
-webkit-text-security	circle、disc、none、square	指定要在密码输入字段中使用的替换形状。
-webkit-user-drag	<ul style="list-style-type: none">• auto — 默认行为• element — 拖动整个元素• none — 无法拖动元素	覆盖自动拖动行为。
-webkit-user-modify	read-only、read-write、read-write-plaintext-only	指定是否可以编辑元素内容。
-webkit-user-select	<ul style="list-style-type: none">• auto — 默认行为• none — 无法选择元素• text — 只能选择元素中的文本	指定用户是否可以选择元素内容。

有关详细信息，请参阅 Apple Safari CSS Reference
(<http://developer.apple.com/documentation/AppleApplications/Reference/SafariCSSRef/>)。

第 12 章：处理与 HTML 相关的事件

利用事件处理系统，程序员可以十分方便地响应用户输入和系统事件。Adobe® AIR® 事件模型不仅方便，而且符合标准。事件模型基于文档对象模型 (DOM) 第 3 级事件规范，是业界标准的事件处理体系结构，为程序员提供了强大而直观的事件处理工具。

HTMLLoader 事件

HTMLLoader 对象调度以下 Adobe® ActionScript® 3.0 事件：

事件	说明
htmlDOMInitialize	在创建 HTML 文档时调度，调度时未分析任何脚本或未将 DOM 节点添加到页面。
complete	在响应加载操作而创建 HTML DOM 后，紧接在 HTML 页面中的 <code>onload</code> 事件后调度。
htmlBoundsChanged	在 <code>contentWidth</code> 和 / 或 <code>contentHeight</code> 属性发生了变化时调度。
locationChange	在 <code>HTMLLoader</code> 的 <code>location</code> 属性发生了变化时调度。
scroll	只要 HTML 引擎更改滚动位置，便会调度此事件。发生 <code>scroll</code> 事件可能是由于导航到页面中的锚记链接 (# 链接)，也可能是由于调用了 <code>window.scrollTo()</code> 方法。在文本输入或文本区域中输入文本也可能会引发 <code>scroll</code> 事件。
uncaughtScriptException	当在 <code>HTMLLoader</code> 中发生 JavaScript 异常，并且在 JavaScript 代码中未捕获到该异常时调度。

您也可以为 JavaScript 事件（如 `onClick`）注册 ActionScript 函数。有关详细信息，请参阅使用 ActionScript 处理 DOM 事件。

AIR 类 - 事件处理与 HTML DOM 中其他事件处理的不同之处

HTML DOM 提供了几种不同的方法来处理事件：

- 在 HTML 元素的开始标签中定义 `on` 事件处理函数，如下所示：

```
<div id="myDiv" onclick="myHandler()">
```

- 回调函数属性，例如：

```
document.getElementById("myDiv").onclick
```

- 使用 `addEventListener()` 方法注册的事件侦听器，如下所示：

```
document.getElementById("myDiv").addEventListener("click", clickHandler)
```

不过，由于运行时对象不会出现在 DOM 中，因此只能通过调用 AIR 对象的 `addEventListener()` 方法来添加事件侦听器。

与在 JavaScript 中一样，由 AIR 对象调度的事件可以与默认行为关联起来。（默认行为是 AIR 作为特定事件的正常后果而执行的动作。）

由运行时对象调度的事件对象是 `Event` 类或其某一个子类的实例。事件对象不但存储有关特定事件的信息，还包含便于操作此事件对象的方法。例如，如果 AIR 在异步读取文件时检测到 I/O 错误事件，则会创建用来表示该特定 I/O 错误事件的事件对象（`IOErrorEvent` 类的实例）。

无论何时编写事件处理函数代码，该代码都采用相同的基本结构：

```
function eventResponse(eventObject)
{
    // Actions performed in response to the event go here.
}

eventTarget.addEventListener(EventType.EVENT_NAME, eventResponse);
```

此代码完成两项任务。首先，它定义一个处理函数，这是指定为响应事件而要执行的动作的方法。接下来，它调用源对象的 `addEventListener()` 方法，实际上就是为指定事件订阅该函数，以便在发生事件时执行处理函数动作。当事件实际发生时，事件目标将检查其向事件监听器注册的所有函数和方法的列表。然后，它依次调用每个函数或方法，同时将事件对象作为参数传递。

默认行为

开发人员通常负责编写响应事件的代码。但在某些情况下，行为通常与某一事件关联，使得 AIR 会自动执行该行为，除非开发人员添加了取消该行为的代码。由于 AIR 会自动表现该行为，因此这类行为称为默认行为。

例如，当用户单击应用程序窗口的关闭框时，普遍期待窗口关闭，因此该行为被内置到 AIR 中。如果您不希望该默认行为发生，可以使用事件处理系统来取消它。当用户单击某个窗口的关闭框时，表示该窗口的 `NativeWindow` 对象将调度 `closing` 事件。若要防止运行时关闭该窗口，必须调用所调度的事件对象的 `preventDefault()` 方法。

并非所有默认行为都可以被阻止。例如，当 `FileStream` 对象将数据写入某个文件时，运行时将生成 `OutputProgressEvent` 对象。无法阻止的默认行为是：用新数据更新该文件的内容。

许多类型的事件对象没有关联的默认行为。例如，在读取了 MP 文件中的足量数据后，`Sound` 对象会调度 `id33` 事件以提供 ID3 信息，但没有与其关联的默认行为。`Event` 类及其子类的 API 文档列出了每一类型的事件，并说明所有关联的默认行为，以及是否可以阻止该行为。

注：默认行为仅与运行时直接调度的事件对象关联，对于通过 JavaScript 以编程方式调度的事件对象，不存在默认行为。例如，可以使用 `EventDispatcher` 类的方法调度事件对象，然而调度事件并不会触发默认行为。

事件流

在 AIR 中运行的 SWF 文件内容使用 ActionScript 3.0 显示列表体系结构来显示可视内容。ActionScript 3.0 显示列表为在父级显示对象和子级显示对象之间传播的 SWF 文件内容中的内容和事件（如鼠标单击事件）提供父子关系。HTML DOM 有它自己的只遍历 DOM 元素的独立事件流。当为 AIR 编写基于 HTML 的应用程序时，您主要是使用 HTML DOM，而不是 ActionScript 3.0 显示列表，因此您通常可以忽略运行时语言参考中出现的有关事件阶段的信息。

Adobe AIR 事件对象

在事件处理系统中，事件对象有两个主要用途。首先，事件对象通过将特定事件的有关信息存储在一组属性中来表示实际事件。其次，事件对象包含一组方法，可用于操作事件对象和影响事件处理系统的行为。

AIR API 定义了 `Event` 类，该类用作 AIR API 类调度的所有事件对象的基类。`Event` 类定义所有事件对象共有的一组基本属性和方法。

若要使用 `Event` 对象，务必要先了解 `Event` 类的属性和方法以及 `Event` 类的子类存在的原因。

了解 Event 类的属性

`Event` 类定义了提供有关事件的重要信息的多个只读属性和常量。以下项尤为重要：

- `Event.type` 描述事件对象表示的事件的类型。

- `Event.cancelable` 是一个布尔值，用于报告与事件关联的默认行为（如果有）是否可以取消。
- 事件流信息包含在其余的属性中，仅当在 AIR 中的 SWF 内容中使用 ActionScript 3.0 时才有用。

事件对象类型

每个事件对象都有关联的事件类型。数据类型以字符串值的形式存储在 `Event.type` 属性中。知道事件对象的类型是非常有用的，这样您的代码就可以区分不同类型的对象。例如，下面的代码注册一个 `fileReadHandler()` 倾听器函数以响应由 `myFileStream` 调度的 `complete` 事件：

```
myFileStream.addEventListener(Event.COMPLETE, fileReadHandler);
```

AIR 定义许多类常量（如 `COMPLETE`、`CLOSING` 和 `ID3`），以表示运行时对象调度的事件的类型。这些常量列在《针对 HTML 开发人员的 Adobe AIR 语言参考》的“Event 类”页面中。

事件常量提供了引用特定事件类型的简便方法。使用常量（而不是字符串值）可帮助您更快地识别拼写错误。如果您的代码中拼错了某个常量名，则 JavaScript 分析器将捕获到该错误。而如果您拼错了事件字符串，将会为永远都不会调度的一种事件注册事件处理函数。因此，在添加事件倾听器时，建议使用下面的代码：

```
myFileStream.addEventListener(Event.COMPLETE, htmlRenderHandler);
```

而不是使用：

```
myFileStream.addEventListener("complete", htmlRenderHandler);
```

默认行为信息

代码可通过访问 `cancelable` 属性来检查是否可以阻止任何给定事件对象的默认行为。`cancelable` 属性保存着一个布尔值，用于指示是否可以阻止默认行为。可以使用 `preventDefault()` 方法阻止（即取消）与少量事件关联的默认行为。有关详细信息，请参阅第 75 页的“[取消事件默认行为](#)”。

了解 Event 类的方法

有三种类别的 Event 类方法：

- 实用程序方法：可以创建事件对象的副本或将其转换为字符串。
- 事件流方法：用于从事件流中删除事件对象（主要是在运行时的 SWF 内容中使用 ActionScript 3.0 时使用，请参阅第 74 页的“[事件流](#)”）。
- 默认行为方法：可阻止默认行为或检查是否已阻止默认行为。

Event 类实用程序方法

Event 类有两个实用程序方法。使用 `clone()` 方法可创建事件对象的副本。使用 `toString()` 方法可生成事件对象属性及其值的字符串表示形式。

取消事件默认行为

与取消默认行为有关的两个方法是 `preventDefault()` 方法和 `isDefaultPrevented()` 方法。调用 `preventDefault()` 方法可取消与事件关联的默认行为。使用 `isDefaultPrevented()` 方法可检查是否已对事件对象调用 `preventDefault()`。

`preventDefault()` 方法仅在可以取消事件的默认行为时才起作用。您可以通过查阅 API 文档，或通过检查事件对象的 `cancelable` 属性，来检查事件是否有可以取消的行为。

取消默认行为对事件对象通过事件流的进度没有影响。使用 Event 类的事件流方法可以从事件流中删除事件对象。

Event 类的子类

对于很多事件，Event 类中定义的一组公共属性已经足够了。然而，要表示其他事件，则需要使用 Event 类中未提供的属性。AIR API 为这些事件定义了 Event 类的几个子类。

每个子类提供了对该类别的事件唯一的附加属性和事件类型。例如，与鼠标输入相关的事件提供了描述发生事件时鼠标所在位置的属性。同样，InvokeEvent 类也增加了一些属性，这些属性包含执行调用的文件的文件路径，以及在调用命令行期间作为形参传递的所有实参。

Event 子类频繁定义用来表示与该子类关联的事件类型的其他常量。例如，FileListEvent 类定义表示 directoryListing 和 selectMultiple 事件类型的常量。

使用 JavaScript 处理运行时事件

运行时类支持使用 addEventListener() 方法添加事件处理函数。若要为某个事件添加处理函数，请调用调度该事件的对象的 addEventListener() 方法，并提供事件类型和处理函数。例如，若要侦听用户单击标题栏上的窗口关闭按钮时调度的 closing 事件，请使用下面的语句：

```
window.nativeWindow.addEventListener(air.NativeWindow.CLOSING, handleWindowClosing);
```

addEventListener() 方法的 type 参数为字符串，但 AIR API 为所有运行时事件类型定义了常量。与使用字符串版本相比，使用这些常量可帮助您更快地找到在 type 参数中输入的拼写错误。

创建事件处理函数

下面的代码创建一个简单的 HTML 文件，用于显示有关主窗口位置的信息。一个名为 moveHandler() 的处理函数侦听主窗口的 move 事件（由 NativeWindowBoundsEvent 类定义）。

```
<html>
<script src="AIRAliases.js" />
<script>
    function init() {
        writeValues();
        window.nativeWindow.addEventListener(air.NativeWindowBoundsEvent.MOVE,
                                            moveHandler);
    }
    function writeValues() {
        document.getElementById("xText").value = window.nativeWindow.x;
        document.getElementById("yText").value = window.nativeWindow.y;
    }
    function moveHandler(event) {
        air.trace(event.type); // move
        writeValues();
    }
</script>
<body onload="init()" />
<table>
    <tr>
        <td>Window X:</td>
        <td><textarea id="xText"></textarea></td>
    </tr>
    <tr>
        <td>Window Y:</td>
        <td><textarea id="yText"></textarea></td>
    </tr>
</table>
</body>
</html>
```

当用户移动此窗口时，`textarea` 元素会显示此窗口的更新的 X 位置和 Y 位置：

请注意，将事件对象作为参数传递给 `moveHandler()` 方法。利用 `event` 参数，处理函数可以检查事件对象。在此示例中，使用事件对象的 `type` 属性报告该事件为 `move` 事件。

注：指定 `listener` 参数时，不要使用括号。例如，在下面对 `addEventListener()` 方法的调用中，指定 `moveHandler()` 函数时没有使用括号：`addEventListener(Event.MOVE, moveHandler)`。

`addEventListener()` 方法有其它三个参数，在《针对 HTML 开发人员的 Adobe AIR 语言参考》中对这三个参数进行了介绍；这些参数为 `useCapture`、`priority` 和 `useWeakReference`。

删除事件监听器

可以使用 `removeEventListener()` 方法删除不再需要的事件监听器。建议删除将不再使用的所有监听器。必需的参数包括 `eventName` 和 `listener` 参数，这些参数与 `addEventListener()` 方法的必需参数相同。

删除执行导航的 HTML 页面中的事件监听器

当 HTML 内容进行导航时，或者因包含 HTML 内容的窗口关闭而丢弃这些 HTML 内容时，不会自动删除引用已卸载的页面中对象的事件监听器。当对象向已卸载的处理函数调度事件时，会显示下面的错误消息：“应用程序尝试引用不再处于已加载状态的 HTML 页面中的 JavaScript 对象。”(The application attempted to reference a JavaScript object in an HTML page that is no longer loaded.)

为避免出现此错误，请在 HTML 页面退出之前删除其中的 JavaScript 事件监听器。如果发生页面导航（在 `HTMLLoader` 对象中），请在 `window` 对象的 `window` 事件发生期间删除事件监听器。

例如，下面的 JavaScript 代码删除 `uncaughtScriptException` 事件的事件监听器：

```
window.onunload = cleanup;
window.htmlLoader.addEventListener('uncaughtScriptException', uncaughtScriptException);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
                                         uncaughtScriptExceptionHandler);
}
```

为避免在关闭包含 HTML 内容的窗口时发生错误，请调用 `cleanup` 函数以响应 `NativeWindow` 对象 (`closing`) 的 `window.nativeWindow` 事件。例如，下面的 JavaScript 代码删除 `uncaughtScriptException` 事件的事件监听器：

```
window.nativeWindow.addEventListener(air.Event.CLOSING, cleanup);
function cleanup()
{
    window.htmlLoader.removeEventListener('uncaughtScriptException',
                                         uncaughtScriptExceptionHandler);
}
```

此外，只要事件监听器一运行就将其删除，这样也可以防止此错误的发生。例如，下面的 JavaScript 代码通过调用 `HTMLLoader` 类的 `createRootWindow()` 方法创建一个 `html` 窗口，并为 `complete` 事件添加一个事件监听器。在调用 `complete` 事件处理函数时，它使用 `removeEventListener()` 函数删除了自己的事件监听器：

```
var html = runtime.flash.html.HTMLLoader.createRootWindow(true);
html.addEventListener('complete', htmlCompleteListener);
function htmlCompleteListener()
{
    html.removeEventListener(complete, arguments.callee)
    // handler code..
}
html.load(new runtime.flash.net.URLRequest("second.html"));
```

如果删除不需要的事件监听器，则还会使系统垃圾回收器能回收与这些监听器关联的任何内存空间。

检查有无现有的事件侦听器

使用 `hasEventListener()` 方法可检查某个对象上是否存在事件侦听器。

没有侦听器的错误事件

异常（而不是事件）是在运行时类中处理错误的主要机制。不过，异常处理对异步操作（例如加载文件）不起作用。如果在异步操作过程中发生错误，则运行时会调度一个错误事件对象。如果您不为该错误事件创建侦听器，则 AIR Debug Launcher 将显示包含有关该错误的信息的对话框。

大多数错误事件都基于 `ErrorEvent` 类，并且都有一个名为 `text` 的属性，此属性用于存储描述性错误消息。异常属于 `StatusEvent` 类，此类具有一个 `level` 属性，而不是 `text` 属性。当 `level` 属性的值为 `error` 时，`StatusEvent` 被视为错误事件。

错误事件不会导致应用程序停止运行。它仅以一个对话框的形式显示在 AIR Debug Launcher 中。它根本不会在运行时中运行的已安装 AIR 应用程序中显示。

第 13 章：撰写 HTML 容器脚本

在 Adobe® AIR® 中，HTMLLoader 类用作 HTML 内容的容器。该类提供了许多属性和方法，用于控制对象在 ActionScript® 3.0 显示列表上的行为和外观。此外，该类还为加载 HTML 内容并与之交互以及管理历史记录等任务定义了相关属性和方法。

HTMLHost 类为 HTMLLoader 定义了一组默认行为。在创建 HTMLLoader 对象时，未提供任何 HTMLHost 实现。因此，当 HTML 内容触发某一默认行为时（如更改窗口位置或窗口标题），不会发生任何变化。可以对 HTMLHost 类进行扩展，为您的应用程序定义所需的行为。

对于 AIR 创建的 HTML 窗口，提供了 HTMLHost 的默认实现。通过将 defaultBehavior 参数设置为 true 来创建新的对象，并使用新创建的 HTMLHost 对象设置 HTMLLoader 对象的 htmlHost 属性，可以将默认 HTMLHost 实现分配给其它 HTMLLoader 对象。

HTMLHost 类只能使用 ActionScript 进行扩展。在基于 HTML 的应用程序中，可以导入包含 HTMLHost 类的实现的已编译 SWF 文件。使用 window.htmlLoader 属性分配主机类实现：

```
<script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
<script>
    window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();
</script>
```

HTMLLoader 对象的显示属性

HTMLLoader 对象继承 Adobe® Flash® Player Sprite 类的显示属性。例如，可以调整大小、移动、隐藏和更改背景颜色，也可以应用滤镜、遮罩、缩放和旋转等高级效果。在应用效果时，应考虑对易读性的影响。在应用某些效果时，无法显示加载到 HTML 页中的 SWF 和 PDF 内容。

HTML 窗口包含用于呈现 HTML 内容的 HTMLLoader 对象。此对象被限制在窗口区域内，因此，更改尺寸、位置、旋转或缩放系数并不一定能得到令人满意的结果。

基本显示属性

通过 HTMLLoader 的基本显示属性，可以定位控件在其父显示对象中的位置，设置大小以及显示或隐藏控件。不应更改 HTML 窗口的 HTMLLoader 对象的这些属性。

基本属性包括：

属性	备注
x, y	定位对象在其父容器中的位置。
width, height	更改显示区域的尺寸。
visible	控制对象及其所有内容的可见性。

在 HTML 窗口外部，HTMLLoader 对象的 width 和 height 属性的默认值为 0。必须设置宽度和高度才能看到加载的 HTML 内容。HTML 内容根据 HTMLLoader 大小进行绘制，并根据内容中的 HTML 和 CSS 属性进行布置。更改 HTMLLoader 大小会重新填充内容。

在向新的 HTMLLoader 对象（width 仍设置为 0）中加载内容时，使用 contentWidth 和 contentHeight 属性设置 HTMLLoader 的显示 width 和 height 是一种很不错的做法。此项技术适用于根据 HTML 和 CSS 流规则进行布置时具有合理的最小宽度的页。不过，在缺少 HTMLLoader 提供的合理宽度时，有些页会生成窄而长的布局。

注: 当更改 `HTMLLoader` 对象的宽度和高度时, `scaleX` 和 `scaleY` 值不会发生更改, 大多数其它类型的显示对象也存在此现象。

HTMLLoader 内容的透明度

`HTMLLoader` 对象的 `paintsDefaultBackground` 属性 (默认情况下为 `true`) 确定 `HTMLLoader` 对象是否绘制不透明背景。当 `paintsDefaultBackground` 为 `false` 时, 背景是透明的。显示对象容器或 `HTMLLoader` 对象下的其它显示对象在 HTML 内容的前景元素后是可见的。

如果 `body` 元素或 HTML 文档的任何其它元素指定了背景颜色 (例如, 使用 `style="background-color:gray"`), 则 HTML 的该部分背景是不透明的, 并呈现出指定的背景颜色。如果设置了 `HTMLLoader` 对象的 `opaqueBackground` 属性, 并且 `paintsDefaultBackground` 为 `false`, 则为 `opaqueBackground` 设置的颜色是可见的。

注: 可以使用透明的 PNG 格式的图形为 HTML 文档中的元素提供 Alpha 混合背景。不支持对 HTML 元素设置不透明样式。

缩放 `HTMLLoader` 内容

在对 `HTMLLoader` 对象进行缩放时, 缩放系数应避免超过 1.0。如果对 `HTMLLoader` 对象进行放大, 则 `HTMLLoader` 内容中的文本将以特定的分辨率呈现, 从而产生像素化效果。

在 HTML 页中加载 SWF 或 PDF 内容时的注意事项

在以下情况下, 加载到 `HTMLLoader` 对象中的 SWF 和 PDF 内容将消失:

- `HTMLLoader` 对象的缩放系数不为 1.0。
- 将 `HTMLLoader` 对象的 `alpha` 属性设置为 1.0 之外的值。
- 旋转 `HTMLLoader` 内容。

如果删除出错的属性设置并删除活动滤镜, 可重新显示内容。

注: 运行时无法在透明窗口中显示 SWF 或 PDF 内容。

有关在 `HTMLLoader` 中加载这些类型的媒体的详细信息, 请参阅第 53 页的“[在 HTML 中嵌入 SWF 内容](#)”和第 248 页的“[添加 PDF 内容](#)”。

高级显示属性

`HTMLLoader` 类继承了一些可用于生成特殊效果的方法。通常, 这些效果在用于 `HTMLLoader` 显示时存在一些限制, 但对于生成过渡或其它临时效果会非常有用。例如, 如果显示一个对话窗口来收集用户输入内容, 则可以在用户关闭对话之前模糊显示主窗口。同样, 在关闭窗口时可以淡出显示。

高级显示属性包括:

属性	限制
<code>alpha</code>	会降低 HTML 内容的易读性
<code>filters</code>	在 HTML 窗口中, 外部效果沿窗口边缘剪裁
<code>graphics</code>	使用图形命令绘制的形状显示在 HTML 内容下方 (包括默认背景)。 <code>paintsDefaultBackground</code> 属性必须为 <code>false</code> , 绘制的形状才可见。
<code>opaqueBackground</code>	不更改默认背景的颜色。 <code>paintsDefaultBackground</code> 属性必须为 <code>false</code> , 此颜色层才可见。

属性	限制
rotation	矩形 HTMLLoader 区域的各个角会沿窗口边缘剪裁。不显示 HTML 内容中加载的 SWF 和 PDF 内容。
scaleX, scaleY	当缩放系数大于 1 时，会呈现像素化效果。不显示 HTML 内容中加载的 SWF 和 PDF 内容。
transform	会降低 HTML 内容的易读性。HTML 显示会沿窗口边缘剪裁。如果转换涉及旋转、缩放或倾斜，则不显示 HTML 内容中加载的 SWF 和 PDF 内容。

下面的示例说明如何设置 filters 数组使整个 HTML 模糊显示：

```
var blur = new window.runtime.flash.filters.BlurFilter();
var filters = [blur];
window.htmlLoader.filters = filters;
```

注：在基于 HTML 的应用程序中，通常不使用 Sprite 和 BlurFilter 等显示对象类。它们既未列入[针对 HTML 开发人员的 Adobe AIR 语言参考](http://www.adobe.com/go/learn_air_html_jslr_cn) (http://www.adobe.com/go/learn_air_html_jslr_cn)，也未在 AIRAliases.js 文件中设置别名。有关这些类的文档，可参考[Flex 3 语言参考](#)。

访问 HTML 历史记录列表

在 HTMLLoader 对象中加载新页时，运行时将为该对象维护一份历史记录列表。历史记录列表对应于 HTML 页中的 window.history 对象。HTMLLoader 类包含以下属性和方法，可用于操作 HTML 历史记录列表：

类成员	说明
historyLength	历史记录列表的总长度，包括向后和向前的条目。
historyPosition	历史记录列表中的当前位置。位于此位置之前的历史记录项表示“向后”导航，位于此位置之后的项表示“向前”导航。
getHistoryAt()	返回与历史记录列表中指定位置的历史记录条目对应的 URLRequest 对象。
historyBack()	如果可能，在历史记录列表中向后导航。
historyForward()	如有可能，请在历史记录列表中向前导航。
historyGo()	在浏览器历史记录中按指示的步数导航。如果为正数，则向前导航；如果为负数，则向后导航。导航到零将重新加载页面。如果指定的位置超出末尾位置，则将导航到列表末尾。

历史记录列表中的项作为 HistoryListItem 类型的对象存储。HistoryListItem 类具有以下属性：

属性	说明
isPost	如果 HTML 页包括 POST 数据，则设置为 true。
originalUrl	在进行任何重定向之前，HTML 页的原始 URL。
title	HTML 页的标题。
url	HTML 页的 URL。

设置在加载 HTML 内容时使用的用户代理

HTMLLoader 类具有 userAgent 属性，通过该属性可以设置 HTMLLoader 使用的用户代理字符串。在调用 load() 方法之前请设置 HTMLLoader 对象的 userAgent 属性。如果对 HTMLLoader 实例设置此属性，则不使用传递给 load() 方法的 URLRequest 的 userAgent 属性。

通过设置 URLRequestDefaults.userAgent 属性，可以设置应用程序域中所有 HTMLLoader 对象使用的默认用户代理字符串。URLRequestDefaults 静态属性作为默认属性适用于所有 URLRequest 对象，不只是与 HTMLLoader 对象的 load() 方法一起使用的 URLRequest。设置 HTMLLoader 的 userAgent 属性将覆盖 URLRequestDefaults.userAgent 默认设置。

如果既未为 HTMLLoader 对象的 userAgent 属性设置用户代理值，也未为 URLRequestDefaults.userAgent 设置用户代理值，则将使用默认的 AIR 用户代理值。此默认值随着运行时操作系统（如 Mac OS 或 Windows）、运行时语言和运行时版本而变化，如下面两个示例所示：

- "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"
- "Mozilla/5.0 (Windows; U; en) AppleWebKit/420+ (KHTML, like Gecko) AdobeAIR/1.0"

设置用于 HTML 内容的字符编码

HTML 页通过包括 meta 标签可以指定其使用的字符编码，如下所示：

```
meta http-equiv="content-type" content="text/html" charset="ISO-8859-1";
```

通过设置 HTMLLoader 对象的 textEncodingOverride 属性覆盖页面设置，确保使用特定的字符编码：

```
window.htmlLoader.textEncodingOverride = "ISO-8859-1";
```

使用 HTMLLoader 对象的 textEncodingFallback 属性，指定当 HTML 页未指定字符编码设置时要对 HTMLLoader 内容使用的字符编码：

```
window.htmlLoader.textEncodingFallback = "ISO-8859-1";
```

textEncodingOverride 属性将覆盖 HTML 页中的设置。并且 textEncodingOverride 属性和 HTML 页中的设置将覆盖 textEncodingFallback 属性。

需在加载 HTML 内容之前设置 textEncodingOverride 属性或 textEncodingFallback 属性。

为 HTML 内容定义类似于浏览器的用户界面

JavaScript 提供了多个 API 来控制显示 HTML 内容的窗口。在 AIR 中，通过实现自定义 HTMLHost 类可以覆盖这些 API。

重要说明：使用 ActionScript 只能创建 HTMLHost 类的自定义实现。在 HTML 页中，可以导入和使用包含自定义实现的已编译 ActionScript (SWF) 文件。有关将 ActionScript 库导入 HTML 的详细信息，请参阅第 54 页的“[在 HTML 页中使用 ActionScript 库](#)”。

关于扩展 HTMLHost 类

AIR HTMLHost 类控制以下 JavaScript 属性和方法：

- window.status
- window.document.title
- window.location

- window.blur()
- window.close()
- window.focus()
- window.moveBy()
- window.moveTo()
- window.open()
- window.resizeBy()
- window.resizeTo()

在使用 new HTMLLoader() 创建 HTMLLoader 对象时，不会启用列出的 JavaScript 属性或方法。HTMLHost 类提供了这些 JavaScript API 的类似于浏览器的默认实现。还可以扩展 HTMLHost 类以自定义行为。若要创建支持默认行为的 HTMLHost 对象，请在 HTMLHost 构造函数中将 defaultBehaviors 参数设置为 true：

```
var defaultHost = new HTMLHost(true);
```

在 AIR 中，使用 HTMLLoader 类的 createRootWindow() 方法创建 HTML 窗口时，将自动分配支持默认行为的 HTMLHost 实例。可以通过向 HTMLLoader 的 htmlHost 属性分配不同的 HTMLHost 实现来更改主机对象行为，也可以分配 null 以禁用整个功能。

注 AIR 将默认的 HTMLHost 对象分配给基于 HTML 的 AIR 应用程序创建的初始窗口以及由 JavaScript window.open() 方法的默认实现创建的任何窗口。

示例：扩展 HTMLHost 类

下面的示例说明如何通过扩展 HTMLHost 类来自定义 HTMLLoader 对象影响用户界面的方式：

- 1 创建一个 ActionScript 文件，例如 HTMLHostImplementation.as。
- 2 在此文件中，定义一个 HTMLHost 类的扩展类。
- 3 覆盖新类的方法以处理用户界面相关设置中的更改。例如，以下 CustomHost 类定义调用 window.open() 和更改 window.document.title 的行为。调用 window.open() 将在新窗口中打开 HTML 页，更改 window.document.title（包括 HTML 页的 <title> 元素的设置）将设置该窗口的标题。

```

package {
    import flash.html.HTMLHost;
    import flash.html.HTMLLoader;
    import flash.html.HTMLWindowCreateOptions;
    import flash.geom.Rectangle;
    import flash.display.NativeWindowInitOptions;
    import flash.display.StageDisplayState;

    public class HTMLHostImplementation extends HTMLHost{
        public function HTMLHostImplementation(defaultBehaviors:Boolean = true):void{
            super(defaultBehaviors);
        }

        override public function updateTitle(title:String):void{
            htmlLoader.stage.nativeWindow.title = title + " - New Host";
        }

        override public function createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader{
            var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
            var bounds:Rectangle = new Rectangle(windowCreateOptions.x,
                windowCreateOptions.y,
                windowCreateOptions.width,
                windowCreateOptions.height);

            var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                windowCreateOptions.scrollBarsVisible, bounds);

            htmlControl.htmlHost = new HTMLHostImplementation();

            if(windowCreateOptions.fullscreen){
                htmlControl.stage.displayState =
                    StageDisplayState.FULL_SCREEN_INTERACTIVE;
            }

            return htmlControl;
        }
    }
}

```

4 使用 acmpc 组件编译器将该类编译为 SWF 文件。

```
acmpc -source-path . -include-classes HTMLHostImplementation -output Host.zip
```

注: acmpc 编译器包含在 Flex 3 SDK 中 (不是在 AIR SDK 中, AIR SDK 面向通常不需要编译 SWF 文件的 HTML 开发人员。) [使用 Adobe Flex 3 开发 AIR 应用程序](#) 中提供了有关使用 acmpc 的说明。

5 打开 Host.zip 文件并提取其中的 Library.swf 文件。

6 将 Library.swf 重命名为 HTMLHostLibrary.swf。此 SWF 文件是要导入 HTML 页中的库。

7 使用 <script> 标签将该库导入 HTML 页:

```
<script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
```

8 将 HTMLHost 实现的新实例分配到该页的 HTMLLoader 对象。

```
window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();
```

以下 HTML 页说明如何加载和使用 HTMLHost 实现。通过单击按钮打开一个新的全屏窗口, 可以测试 updateTitle() 和 createWindow() 实例。

```
<html>
<head>
<title>HTMLHost Example</title>
<script src="HTMLHostLibrary.swf" type="application/x-shockwave-flash"></script>
<script language="javascript">
    window.htmlLoader.htmlHost = new window.runtime.HTMLHostImplementation();

    function test(){
        window.open('child.html', 'Child', 'fullscreen');
    }
</script>
</head>
<body>
    <button onClick="test()">Create Window</button>
</body>
</html>
```

若要运行此示例，请在应用程序目录中提供一个名为 child.html 的 HTML 文件。

处理对 `window.location` 属性的更改

覆盖 `locationChange()` 方法以处理对 HTML 页的 URL 的更改。当某页中的 JavaScript 更改了 `window.location` 的值时，将调用 `locationChange()` 方法。以下示例仅加载了请求的 URL：

```
override public function updateLocation(locationURL:String):void
{
    htmlLoader.load(new URLRequest(locationURL));
}
```

注：可以使用 `HTMLHost` 对象的 `htmlLoader` 属性来引用当前的 `HTMLLoader` 对象。

处理对 `window.moveTo()`、`window.resizeTo()`、`window.resizeBy()`、 `window.moveTo()` 的 JavaScript 调用

覆盖 `setWindowRect()` 方法以处理 HTML 内容范围的更改。当某页中的 JavaScript 调用 `window.moveTo()`、
`window.moveTo()`、`window.resizeTo()` 或 `window.resizeBy()` 时，将调用 `setWindowRect()` 方法。以下示例仅更新了桌面窗口范围：

```
override public function setWindowRect(value:Rectangle):void
{
    htmlLoader.stage.nativeWindow.bounds = value;
}
```

处理对 `window.open()` 的 JavaScript 调用

覆盖 `createWindow()` 方法以处理对 `window.open()` 的 JavaScript 调用。`createWindow()` 方法的实现负责创建和返回新的 `HTMLLoader` 对象。通常，将在新窗口中显示 `HTMLLoader`，但不需要创建一个窗口。

以下示例说明如何通过使用 `HTMLLoader.createRootWindow()` 创建窗口和 `HTMLLoader` 对象来实现 `createWindow()` 函数。还可以单独创建一个 `NativeWindow` 对象，然后将 `HTMLLoader` 添加到窗口舞台。

```

override public function createWindow(windowCreateOptions:HTMLWindowCreateOptions):HTMLLoader{
    var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
    var bounds:Rectangle = new Rectangle(windowCreateOptions.x, windowCreateOptions.y,
                                         windowCreateOptions.width, windowCreateOptions.height);
    var htmlControl:HTMLLoader = HTMLLoader.createRootWindow(true, initOptions,
                                                          windowCreateOptions.scrollBarsVisible, bounds);
    htmlControl.htmlHost = new HTMLHostImplementation();
    if(windowCreateOptions.fullscreen){
        htmlControl.stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
    }
    return htmlControl;
}

```

注：本示例将自定义 HTMLHost 实现分配给使用 `window.open()` 创建的所有新窗口。如果需要，还可以对新窗口使用不同的实现或将 `htmlHost` 属性设置为 `null`。

作为参数传递给 `createWindow()` 方法的对象是 `HTMLWindowCreateOptions` 对象。`HTMLWindowCreateOptions` 类包含相关属性，可报告在对 `features()` 的调用中，`window.open()` 参数字符串中设置的值：

<code>HTMLWindowCreateOptions</code> 属性	在对 <code>window.open()</code> 的 JavaScript 调用中， <code>features</code> 字符串中的相应设置
<code>fullscreen</code>	<code>fullscreen</code>
<code>height</code>	<code>height</code>
<code>locationBarVisible</code>	<code>location</code>
<code>menuBarVisible</code>	<code>menubar</code>
<code>resizeable</code>	<code>resizable</code>
<code>scrollBarsVisible</code>	<code>scrollbars</code>
<code>statusBarVisible</code>	<code>status</code>
<code>toolBarVisible</code>	<code>toolbar</code>
<code>width</code>	<code>width</code>
<code>x</code>	<code>left</code> 或 <code>screenX</code>
<code>y</code>	<code>top</code> 或 <code>screenY</code>

`HTMLLoader` 类并不会实现可在 `feature` 字符串中指定的所有功能。您的应用程序必须在适当的时候提供滚动条、位置栏、菜单栏、状态栏和工具栏。

JavaScript `window.open()` 方法的其它参数由系统处理。`createWindow()` 实现不应在 `HTMLLoader` 对象中加载内容或设置窗口标题。

处理对 `window.close()` 的 JavaScript 调用

覆盖 `windowClose()` 方法以处理对 `window.close()` 方法的 JavaScript 调用。以下示例在调用 `window.close()` 方法时关闭桌面窗口。

```

override public function windowClose():void
{
    htmlLoader.stage.nativeWindow.close();
}

```

对 `window.close()` 的 JavaScript 调用不必关闭包含窗口。例如，可以从显示列表中删除 `HTMLLoader`，保持窗口（可能包含其它内容）处于打开状态，如以下代码所示：

```
override public function windowClose():void
{
    htmlLoader.parent.removeChild(htmlLoader);
}
```

处理对 **window.status** 属性的更改

覆盖 `updateStatus()` 方法以处理对 `window.status` 值的 JavaScript 更改。以下示例跟踪状态值：

```
override public function updateStatus(status:String):void
{
    trace(status);
}
```

请求的状态作为字符串传递给 `updateStatus()` 方法。

`HTMLLoader` 对象不提供状态栏。

处理对 **window.document.title** 属性的更改

覆盖 `updateTitle()` 方法以处理对 `window.document.title` 值的 JavaScript 更改。以下示例更改窗口标题并向标题追加“Sample”字符串：

```
override public function updateTitle(title:String):void
{
    htmlLoader.stage.nativeWindow.title = title + " - Sample";
}
```

在 HTML 页上设置 `document.title` 时，请求的标题将作为字符串传递给 `updateTitle()` 方法。

更改 `document.title` 时不必更改包含 `HTMLLoader` 对象的窗口的标题。可以更改其它界面元素，如文本字段。

处理对 **window.blur()** 和 **window.focus()** 的 JavaScript 调用

覆盖 `windowBlur()` 和 `windowFocus()` 方法以处理对 `window.blur()` 和 `window.focus()` 的 JavaScript 调用，如下例所示：

```
override public function windowBlur():void
{
    htmlLoader.alpha = 0.5;
}
override public function windowFocus():void
{
    htmlLoader.alpha = 1.0;
    NativeApplication.nativeApplication.activate(htmlLoader.stage.nativeWindow);
}
```

注：AIR 不提供用于取消激活窗口或应用程序的 API。

创建具有滚动 HTML 内容的窗口

`HTMLLoader` 类包含一个静态方法 `HTMLLoader.createRootWindow()`，使用该方法，可以打开一个包含 `HTMLLoader` 对象的新窗口（由 `NativeWindow` 对象表示）并为该窗口定义一些用户界面设置。该方法采用四个参数，可以通过这些参数来定义用户界面：

参数	说明
visible	一个布尔值，它指定窗口最初是 (true) 否 (false) 可见。
windowInitOptions	一个 NativeWindowInitOptions 对象。NativeWindowInitOptions 类为 NativeWindow 对象定义初始化选项，包括以下内容：窗口是否可最小化、可最大化或可调整大小，窗口是否有系统镶边或自定义镶边，窗口是否透明（对于不使用系统镶边的窗口）以及窗口的类型。
scrollBarsVisible	是 (true) 否 (false) 有滚动条。
bounds	一个 Rectangle 对象，用于定义新窗口的位置和大小。

例如，以下代码使用 HTMLLoader.createRootWindow() 方法创建包含使用滚动条的 HTMLLoader 内容的窗口：

```
var initOptions = new air.NativeWindowInitOptions();
var bounds = new air.Rectangle(10, 10, 600, 400);
var html2 = air.HTMLLoader.createRootWindow(true, initOptions, true, bounds);
var urlReq2 = new air.URLRequest("http://www.example.com");
html2.load(urlReq2);
html2.stage.nativeWindow.activate();
```

注：通过直接在 JavaScript 中调用 createRootWindow() 所创建的窗口仍与打开的 HTML 窗口保持独立。例如，JavaScript Window opener 和 parent 的属性为 null。不过，如果通过覆盖 HTMLHost createWindow() 方法间接地调用 createRootWindow()createRootWindow()，则 opener 和 parent 将引用打开的 HTML 窗口。

第 14 章：AIR 安全性

本主题将讨论在开发 AIR 应用程序时应考虑的安全问题。

AIR 安全性基础知识

运行 AIR 应用程序所需的用户权限与运行本机应用程序所需的用户权限相同。通常，使用这些权限可以对操作系统功能进行广泛的访问，例如读取和写入文件、启动应用程序、在屏幕上执行拖放操作以及与网络进行通信。适用于本机应用程序的操作系统限制（例如特定于用户的权限）同样适用于 AIR 应用程序。

虽然 Adobe® AIR® 安全模型是由 Adobe® Flash® Player 安全模型发展而来的，但是其安全协定与适用于浏览器中的内容的安全协定不同。此协定为开发人员提供了一种自由访问更广泛的功能以便获得丰富体验的安全方式，而这种方式并不适合基于浏览器的应用程序。

AIR 应用程序是采用编译过的字节码（SWF 内容）或解释过的脚本（JavaScript、HTML）编写的，以便运行时提供内存管理。这样可以最大程度地减少与内存管理（如缓冲区溢出和内存损坏）有关的漏洞对 AIR 应用程序产生影响的可能性。下面是一些影响用本机代码编写的桌面应用程序的最常见漏洞。

安装和更新

AIR 应用程序是通过使用 air 扩展名的 AIR 安装程序文件分发的。如果安装了 Adobe AIR 且打开了 AIR 安装程序文件，运行时就会管理安装过程。

注：开发人员可以指定版本、应用程序名称和发行商源，但初始应用程序安装流程本身无法修改。此限制对用户非常有利，因为所有 AIR 应用程序共享由运行时管理的安全、简单且一致的安装过程。如果有必要对应用程序进行自定义，则可以在首次执行应用程序时进行自定义。

运行时安装位置

AIR 应用程序首先要求在用户的计算机上安装运行时，就像 SWF 文件首先要求安装 Flash Player 浏览器插件一样。

运行时安装在用户计算机上的以下位置：

- Mac OS: /Library/Frameworks/
- Windows: C:\Program Files\Common Files\Adobe AIR
- Linux: /opt/Adobe AIR/

在 Mac OS 中，若要安装某一应用程序的更新版本，用户必须具有足够的系统权限才能将新版本安装到应用程序目录中。在 Windows 和 Linux 中，用户必须具有管理权限。

可以通过两种方式安装运行时：使用无缝安装功能（直接从 Web 浏览器安装）或通过手动安装。有关详细信息，请参阅第 305 页的“[分发、安装和运行 AIR 应用程序](#)”。

无缝安装（运行时和应用程序）

借助无缝安装功能，开发人员可以让没有 Adobe AIR 安装经验的用户体验简化的安装过程。通过无缝安装方法，开发人员可以创建用于提供应用程序安装的 SWF 文件。用户单击该 SWF 文件安装应用程序时，该 SWF 文件将尝试检测运行时。如果检测不到运行时，运行时会自行安装并且会立即激活，同时开始安装开发人员的应用程序。

手动安装

用户也可以在打开 AIR 文件之前手动下载并安装运行时。开发人员随后可以通过不同的方式（例如通过电子邮件或网站上的 HTML 链接）分发 AIR 文件。打开 AIR 文件后，运行时便开始处理应用程序安装过程。

有关此安装过程的详细信息，请参阅第 305 页的“[分发、安装和运行 AIR 应用程序](#)”

应用程序安装流程

AIR 安全模型允许用户决定是否要安装 AIR 应用程序。AIR 安装体验在本机应用程序安装技术的基础上提供了以下几个方面的改进，使用户可以更容易地做出信任安装的决定：

- 即使通过 Web 浏览器中的链接安装 AIR 应用程序，运行时也会对所有操作系统提供一致的安装体验。大多数本机应用程序安装体验根据浏览器或其它应用程序提供安全信息（如果提供了安全信息）。
- AIR 应用程序安装体验可以确定应用程序的源以及有关应用程序可用权限的信息（如果用户允许继续安装）。
- 运行时会管理 AIR 应用程序的安装过程。AIR 应用程序无法控制运行时使用的安装过程。

通常，用户不应安装来自其不信任源或无法验证源的任何桌面应用程序。与其它可安装应用程序一样，对本机应用程序执行的安全验证也适用于 AIR 应用程序。

应用程序安装目标

可以选择以下两种方式之一设置安装目录：

- 1 用户在安装过程中自定义目标。应用程序将安装到用户指定的任意位置。
- 2 如果用户未更改安装目标，则应用程序将安装到运行时确定的默认路径下：
 - Mac OS: ~/Applications/
 - Windows XP 及更低版本: C:\Program Files\
 - Windows Vista: ~/Apps/
 - Linux: /opt/

如果开发人员在应用程序描述符文件中指定了 `installFolder` 设置，则应用程序将安装到此目录的子路径下。

AIR 文件系统

AIR 应用程序的安装过程会将开发人员在 AIR 安装程序文件中包括的所有文件复制到用户的本地计算机上。安装的应用程序由以下内容组成：

- Windows: 包含 AIR 安装程序文件中的所有文件的目录。在安装 AIR 应用程序的过程中，运行时还会创建一个 `exe` 文件。
- Linux: 包含 AIR 安装程序文件中所含所有文件的目录。在安装 AIR 应用程序的过程中，运行时还会创建一个 `bin` 文件。
- Mac OS: 包含 AIR 安装程序文件的所有内容的 `app` 文件。可以使用 Finder 中的“显示包内容”选项检查该文件。运行时会在 AIR 应用程序的安装过程中创建该 `app` 文件。

AIR 应用程序的运行方式如下：

- Windows: 运行安装文件夹中的 `.exe` 文件或对应于此文件的快捷方式（如“开始”菜单或桌面上的快捷方式）。
- Linux: 启动安装文件夹中的 `.bin` 文件、从“应用程序”菜单中选择该应用程序，或者从别名或桌面快捷方式运行。
- Mac OS: 运行 `.app` 文件或指向该文件的别名。

应用程序文件系统还包括与应用程序功能相关的子目录。例如，写入加密本地存储的信息保存到以应用程序的应用程序标识符命名的目录的子目录中。

AIR 应用程序存储

AIR 应用程序具有在用户硬盘上任意位置写入的权限；但是，鼓励开发人员使用 `app-storage:/` 路径作为与其应用程序相关的本地存储。从应用程序写入 `app-storage:/` 的文件位于标准位置，具体取决于用户的操作系统：

- 在 Mac OS 中：应用程序的存储目录为 `<appData>/<appId>/Local Store/`，其中 `<appData>` 表示用户的“首选参数文件夹”，通常为 `/Users/<user>/Library/Preferences`
- 在 Windows 中：应用程序的存储目录为 `<appData>\<appId>\Local Store\`，其中 `<appData>` 表示用户的 `CSIDL_APPDATA“特殊文件夹”`，通常为 `C:\Documents and Settings\<user>\Application Data`
- 在 Linux 中为 `<appData>/<appId>/Local Store/`，其中 `<appData>` 表示 `/home/<user>/.appdata`

可以通过 `air.File.applicationStorageDirectory` 属性访问应用程序存储目录。可以使用 `File` 类的 `resolvePath()` 方法访问目录中的内容。有关详细信息，请参阅第 162 页的“[使用文件系统](#)”。

更新 Adobe AIR

如果用户安装的 AIR 应用程序需要运行时的更新版本，则运行时会自动安装所需的运行时更新。

若要更新运行时，用户必须具有计算机的管理权限。

更新 AIR 应用程序

开发和部署软件更新是本机代码应用程序面临的最大安全挑战之一。AIR API 提供了一种改进此问题的机制：可以在启动时调用 `Updater.update()` 方法来检查远程位置是否有 AIR 文件。如果存在适当的更新，则会下载并安装 AIR 文件，然后重新启动该应用程序。开发人员可以使用此类提供新功能和响应潜在安全漏洞。

注：开发人员可以通过设置应用程序描述符文件的版本属性来指定应用程序的版本。AIR 不会以任何方式解释版本字符串。因此，不会认为版本“3.0”比“2.0”新。主要由开发人员来确定是否维护有意义的版本管理。有关详细信息，请参阅第 103 页的“[在应用程序描述符文件中定义属性](#)”。

卸载 AIR 应用程序

用户可以卸载 AIR 应用程序：

- 在 Windows 中：使用“添加 / 删除程序”面板删除该应用程序。
- 在 Mac OS 中：从安装位置删除 `app` 文件。

删除 AIR 应用程序的同时也将删除应用程序目录中的所有文件。但不会删除应用程序可能已写入应用程序目录外部的文件。删除 AIR 应用程序不会撤消 AIR 应用程序对该应用程序目录外部的文件所做的更改。

卸载 Adobe AIR

可以通过以下方式卸载 AIR：

- 在 Windows 中：运行“控制面板”中的“添加 / 删除程序”，选择“Adobe AIR”，然后选择“删除”。
- 在 Mac OS 中：运行 `Applications` 目录中的 `Adobe AIR Uninstaller` 应用程序。

针对管理员的 Windows 注册表设置

在 Windows 中，管理员可以通过配置计算机来阻止（或允许）安装 AIR 应用程序和更新运行时。这些设置包含在 Windows 注册表的 `HKLM\Software\Policies\Adobe\AIR` 项中。这些设置包括以下内容：

注册表设置	说明
AppInstallDisabled	指定是否允许安装和卸载 AIR 应用程序。设置为 0 表示“允许”，设置为 1 表示“禁止”。
UntrustedAppInstallDisabled	指定是否允许安装不受信任的 AIR 应用程序（未包括受信任证书的应用程序）（请参阅第 312 页的“ 对 AIR 文件进行数字签名 ”）。设置为 0 表示“允许”，设置为 1 表示“禁止”。
UpdateDisabled	指定是否允许更新运行时，该操作可以作为后台任务执行，也可以作为显式安装的一部分执行。设置为 0 表示“允许”，设置为 1 表示“禁止”。

沙箱

AIR 提供了一个综合全面的安全体系结构，用于定义 AIR 应用程序中的每个文件的相应内部和外部权限。权限会根据文件的源授予给文件，并被分配到称为“沙箱”的逻辑安全组中。

AIR 安全模型基于 Flash Player 安全模型。此安全模型会根据内容的源将所加载内容的各个项目分类到安全沙箱中。从本地文件系统加载的内容以及从网络域加载的内容均存放在沙箱中。有关详细信息，请参阅 [《Flash Player 9 安全性白皮书》](#) (http://www.adobe.com/go/fp9_0_security_cn) 或 [《Flash Player 10 安全性白皮书》](#) (http://www.adobe.com/go/fp10_0_security_cn)。

关于 AIR 应用程序沙箱

沙箱的运行时安全模型由 Flash Player 安全模型以及应用程序沙箱组成。不在应用程序沙箱中的文件具有类似 Flash Player 安全模型指定的安全限制。

运行时使用这些安全沙箱定义代码可以访问的数据范围以及可以执行的操作。若要维护本地安全，请将各个沙箱中的文件进行隔离。例如，从外部 Internet URL 加载到 AIR 应用程序的 SWF 文件放置在远程沙箱中，该文件默认情况下不具有通过脚本访问应用程序目录中分配给应用程序沙箱的文件的权限。

下表描述了各种类型的沙箱：

沙箱	说明
应用程序	文件位于应用程序目录中，并且使用完全 AIR 权限运行。
远程	文件来自 Internet URL，并且通过基于域的沙箱规则运行，该规则与适用于 Flash Player 中的远程文件的规则类似。（每个网络域都具有单独的远程沙箱，例如 http://www.example.com 和 https://foo.example.org 。）
受信任的本地	文件为本地文件，且用户已使用“设置管理器”或 Flash Player 信任配置文件将其指定为受信任。文件可以从本地数据源读取数据并且与 Internet 通信，但不具有完全 AIR 权限。
只能与远程内容交互	文件为使用网络名称发布的本地 SWF 文件，但尚未受到用户的显式信任。文件可以与 Internet 进行通信，但不能从本地数据源读取数据。此沙箱仅可用于 SWF 内容。
只能与本地文件系统内容交互	文件为未使用网络名称发布的本地脚本文件，且尚未受到用户的显式信任。文件包含尚未受到信任的 JavaScript 文件。文件可以从本地数据源读取数据，但无法与 Internet 进行通信。

本主题重点介绍了应用程序沙箱及其与 AIR 应用程序中的其它沙箱之间的关系。开发人员在使用分配到其它沙箱的内容时，应阅读针对 Flash Player 安全模型的其它文档。请参阅 http://www.adobe.com/go/flashcs4_prog_as3_security_cn 《ActionScript 3.0 编程》 (http://www.adobe.com/go/flashcs4_prog_as3_security_cn) 文档中的“Flash Player 安全性”一章以及 [《Flash Player 9 安全性白皮书》](#) (http://www.adobe.com/go/fp9_0_security_cn) 或 [《Flash Player 10 安全性白皮书》](#) (http://www.adobe.com/go/fp10_0_security_cn)。

应用程序沙箱

安装应用程序时，AIR 安装程序文件中包括的所有文件都会安装到用户计算机的应用程序目录中。开发人员可以通过 app:/ URL 方案在代码中引用此目录（请参阅第 292 页的“[在 URL 中使用 AIR URL 方案](#)”）。在应用程序运行时，应用程序目录树中的所有文件都会分配到应用程序沙箱中。应用程序沙箱中的内容具有 AIR 应用程序的完全访问权限，包括与本地文件系统内容进行交互。

许多 AIR 应用程序只能使用这些本地安装的文件来运行应用程序。但是，不会限制 AIR 应用程序仅加载应用程序目录中的文件，它们可以加载任意源中任何类型的文件。其中包括用户计算机上的本地文件以及可用外部源中的文件（例如本地网络或 Internet 上的文件）。文件类型不会对安全限制产生影响；加载的 HTML 文件与从相同源加载的 SWF 文件具有相同的安全权限。

应用程序安全沙箱中的内容可以访问 AIR API，而其它沙箱中的内容则无法访问。例如，限制 air.NativeApplication.nativeApplication.applicationDescriptor 属性（该属性返回应用程序的应用程序描述符文件的内容）访问应用程序安全沙箱中的内容。另一个示例是受限制的 API 为 FileStream 类，其中包含用于读取和写入本地文件系统的方法。

对于 HTML 内容（位于 HTMLLoader 对象中），所有 JavaScript API（使用 AIRAliases.js 文件时通过 window.runtime 属性或 airair 对象可用）都可用于应用程序安全沙箱中的内容。其它沙箱中的 HTML 内容无法访问 window.runtime 属性，因此该内容也无法访问 AIR API。

JavaScript 和 HTML 限制

对于应用程序安全沙箱中的 HTML 内容，在加载代码后使用可将字符串动态转换为可执行代码的 API 时存在一些限制。这样就可以阻止应用程序从非应用程序源（例如潜在不安全网络域）意外插入（及执行）代码。使用 eval() 函数就是一个示例。有关详细信息，请参阅第 96 页的“[对不同沙箱中的内容的代码限制](#)”。

对 ActionScript 文本字段内容中的 img 标签的限制

为了阻止潜在的跨站脚本攻击，在应用程序安全沙箱的 SWF 内容中忽略了 ActionScript TextField 对象的 HTML 内容中的 img 标签。

asfunction 的限制

应用程序沙箱中的内容无法在 ActionScript 2.0 文本字段的 HTML 内容中使用 asfunction 协议。

无法访问跨域永久性缓存

应用程序沙箱中的 SWF 内容无法使用跨域缓存，这是一项增加到 Flash Player 9 Update 3 的功能。Flash Player 通过此功能可以永久缓存 Adobe 平台组件内容，并根据需要在加载的 SWF 内容中重复使用该内容（无需多次重新加载该内容）。

非应用程序沙箱中的内容的权限

从网络或 Internet 位置加载的文件会分配到 remote 远程沙箱中。从应用程序目录外部加载的文件会分配到 local-with-filesystem、local-with-networking 或 local-trusted 沙箱中；具体取决于文件的创建方式以及用户是否通过 Flash Player 全局设置管理器显式信任了文件。有关详细信息，请参阅

http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html。

JavaScript 和 HTML 限制

与应用程序安全沙箱中的内容不同，非应用程序安全沙箱中的 JavaScript 内容随时都可以调用 eval() 函数来执行动态生成的代码。但是，非应用程序安全沙箱中的 JavaScript 存在一些限制。这些限制包括：

- 非应用程序沙箱中的 JavaScript 代码无法访问 window.runtime 对象，也无法执行 AIR API。

- 默认情况下，非应用程序安全沙箱中的内容无法使用 XMLHttpRequest 调用从调用该请求的域之外的其它域加载数据。但是，通过设置包含 frame 或 iframe 中的 allowCrossdomainXHR 属性，应用程序代码可以授予非应用程序内容执行此操作的权限。有关详细信息，请参阅第 98 页的“[通过脚本访问不同域中的内容](#)”。
- 调用 JavaScript window.open() 方法时存在一些限制。有关详细信息，请参阅第 98 页的“[调用 JavaScript window.open\(\) 方法的限制](#)”。

有关详细信息，请参阅第 96 页的“[对不同沙箱中的内容的代码限制](#)”。

加载 CSS、frame、iframe 和 img 元素的限制

远程（网络）安全沙箱中的 HTML 内容只能从远程域（网络 URL）加载 CSS、frame、iframe 和 img 内容。

local-with-filesystem、local-with-networking 或 local-trusted 沙箱中的 HTML 内容只能从本地沙箱（不能从应用程序或网络 URL）加载 CSS、frame、iframe 和 img 内容。

HTML 安全性

运行时会强制执行规则，并提供克服 HTML 和 JavaScript 中的潜在安全漏洞的机制。不论您的应用程序为主要采用 JavaScript 编写，还是您将 HTML 和 JavaScript 内容加载到基于 SWF 的应用程序，强制执行的规则都相同。应用程序沙箱和非应用程序安全沙箱中的内容（请参阅第 92 页的“[沙箱](#)”）具有不同的权限。将内容加载到 iframe 或 frame 中时，运行时会提供一种安全的沙箱桥机制，该机制允许 frame 或 iframe 中的内容能够与应用程序安全沙箱中的内容进行安全通信。

本主题介绍了 AIR HTML 安全体系结构以及如何使用 iframe、frame 和沙箱桥来设置应用程序。

有关详细信息，请参阅第 48 页的“[避免与安全相关的 JavaScript 错误](#)”。

配置基于 HTML 的应用程序概述

Frame 和 iframe 提供了一种用于组织 AIR 中的 HTML 内容的便利结构。Frame 提供了一种用于维护数据永久性以及安全使用远程内容的方式。

由于 AIR 中的 HTML 保持其基于页面的正常组织，因此 HTML 环境在 HTML 内容的顶框架“导航”到其它页面时会完全刷新。您可以使用 frame 和 iframe 来维护 AIR 中的数据永久性，方法与维护在浏览器上运行的 Web 应用程序中的数据永久性基本相同。定义顶框架中的主应用程序对象，只要不允许框架导航到新页面，这些对象就会永久保留。使用子级 frame 或 iframe 加载并显示应用程序的临时部分。（除 frame 外，还可以使用多种方式维护数据永久性。其中包括 cookie、本地共享对象、本地文件存储、加密文件存储以及本地数据库存储。）

由于 AIR 中的 HTML 在可执行代码与数据之间保持正常的模糊界限，因此 AIR 将 HTML 环境的顶部框架中的内容放入应用程序沙箱中。页面的 load 事件之后，AIR 限制 eval() 等任何可以将文本的字符串转换为可执行对象的操作。即使应用程序未加载远程内容，也会强制实施此限制。若要允许 HTML 内容执行这些受限制的操作，必须使用 frame 或 iframe 将内容放入非应用程序沙箱中。（使用某些依赖 eval() 函数的 JavaScript 应用程序框架时，可能必须运行沙箱子级 frame 中的内容。）有关应用程序沙箱中的 JavaScript 限制的完整列表，请参阅第 96 页的“[对不同沙箱中的内容的代码限制](#)”。

由于 AIR 中的 HTML 能够加载远程潜在不安全内容，因此 AIR 会强制实施同源策略，以防止一个域中的内容与另一个域中的内容进行交互。若要允许应用程序内容与其它域中的内容进行交互，可以设置一个桥，将其用作父级和子级 frame 之间的接口。

设置父子沙箱关系

AIR 会将 sandboxRoot 和 documentRoot 属性添加到 HTML frame 和 iframe 元素中。使用这些属性，您可以将应用程序内容视为其它域中的内容：

属性	说明
sandboxRoot	用于确定在其中放置 frame 内容的沙箱和域的 URL。必须使用 file:、http: 或 https: URL 方案。
documentRoot	从中加载 frame 内容的 URL。必须使用 file:、app: 或 app-storage: URL 方案。

以下示例对要在远程沙箱中运行的应用程序的 **sandbox** 子目录中安装的内容以及 www.example.com 域进行了映射：

```
<iframe
    src="ui.html"
    sandboxRoot="http://www.example.com/local/"
    documentRoot="app:/sandbox/">
</iframe>
```

在不同沙箱或域的父级和子级 frame 之间设置桥

AIR 将 **childSandboxBridge** 和 **parentSandboxBridge** 属性添加到任何子级 frame 的 **window** 对象中。使用这些属性，您可以定义用作父级和子级 frame 之间的接口的桥。每个桥都指向一个方向：

childSandboxBridge — **childSandboxBridge** 属性允许子级 frame 向父级 frame 中的内容公开接口。若要公开接口，需将 **childSandbox** 属性设置为子级 frame 中的函数或对象。然后，可以从父级 frame 中的内容访问该对象或函数。以下示例显示了子级 frame 中运行的脚本如何向其父级 frame 公开包含函数和属性的对象：

```
var interface = {};
interface.calculatePrice = function(){
    return .45 + 1.20;
}
interface.storeID = "abc"
window.childSandboxBridge = interface;
```

如果此子级内容位于分配的 id 为 "child" 的 **iframe** 中，则可以通过读取 **frame** 的 **childSandboxBridge** 属性从父级内容来访问接口：

```
var childInterface = document.getElementById("child").childSandboxBridge;
air.trace(childInterface.calculatePrice()); //traces "1.65"
air.trace(childInterface.storeID)); //traces "abc"
```

parentSandboxBridge — **parentSandboxBridge** 属性允许父级 frame 向子级 frame 中的内容公开接口。若要公开接口，需将子级 frame 的 **parentSandbox** 属性设置为父级 frame 中的函数或对象。然后，可以从子级 frame 中的内容访问该对象或函数。以下示例显示了父级 frame 中运行的脚本如何向其子级 frame 公开包含 **save** 函数的对象：

```
var interface = {};
interface.save = function(text){
    var saveFile = air.File("app-storage:/save.txt");
    //write text to file
}
document.getElementById("child").parentSandboxBridge = interface;
```

使用此接口，子级 frame 中的内容可以将文本保存到名为 **save.txt** 的文件。但对文件系统不具备任何其它访问权限。通常，应用程序内容向其它沙箱公开的接口应越窄越好。子级内容可以调用 **save** 函数，如下所示：

```
var textToSave = "A string.";
window.parentSandboxBridge.save(textToSave);
```

如果子级内容尝试设置 **parentSandboxBridge** 对象的属性，则运行时会引发 **SecurityError** 异常。如果父级内容尝试设置 **childSandboxBridge** 对象的属性，则运行时会引发 **SecurityError** 异常。

对不同沙箱中的内容的代码限制

正如本主题的简介部分 第 94 页的“[HTML 安全性](#)”中所介绍的那样，运行时强制执行规则，并提供克服 HTML 和 JavaScript 中的潜在安全漏洞的机制。本主题列出了这些限制。如果代码尝试调用这些受限制的 API，则运行时将发出错误：“Adobe AIR runtime security violation for JavaScript code in the application security sandbox”（应用程序安全沙箱中存在针对 JavaScript 代码的 Adobe AIR 运行时安全侵犯）。

有关详细信息，请参阅第 48 页的“[避免与安全相关的 JavaScript 错误](#)”。

使用 JavaScript eval() 函数及类似技术的限制

对于应用程序安全沙箱中的 HTML 内容，加载代码后（即在调度 `onload` 元素的 `body` 事件以及 `onload` 处理函数完成执行后），使用可将字符串动态转换为可执行代码的 API 时存在一些限制。这是为了阻止应用程序从非应用程序源（例如潜在不安全网络域）意外插入（及执行）代码。

例如，如果应用程序使用远程源中的字符串数据来写入 DOM 元素的 `innerHTML` 属性，则字符串中包括的可执行（JavaScript）代码可能会执行不安全操作。但是，在加载内容时将远程字符串插入 DOM 不存在风险。

使用 JavaScript `eval()` 函数时存在一个限制。在加载应用程序沙箱中的代码且处理 `onload` 事件处理函数之后，就只能通过有限的方式使用 `eval()` 函数。以下规则适用于从应用程序安全沙箱中加载代码之后使用 `eval()` 函数：

- 允许表达式中包含文本。例如：

```
eval("null");
eval("3 + .14");
eval("'foo'");
```

- 允许使用对象文本，如下所示：

```
{ prop1: val1, prop2: val2 }
```

- 禁止 使用 `setter/getter` 对象文本，如下所示：

```
{ get prop1() { ... }, set prop1(v) { ... } }
```

- 允许使用数组文本，如下所示：

```
[ val1, val2, val3 ]
```

- 禁止 表达式中包含属性读取，如下所示：

```
a.b.c
```

- 禁止 调用函数。

- 禁止 禁止对函数进行定义。

- 禁止 设置任何属性。

- 禁止 使用函数文本。

但是，加载代码时，在 `onload` 事件之前和执行 `onload` 事件处理函数过程中，这些限制不适用于应用程序安全沙箱中的内容。

例如，加载代码后，以下代码会导致运行时引发异常：

```
eval("alert(44)");
eval("myFunction(44)");
eval("NativeApplication.applicationID");
```

如果在应用程序沙箱内允许使用动态生成的代码（如调用 `eval()` 函数时生成的代码），则会引起安全性风险。例如，应用程序可能意外执行了从网络域中加载的字符串，而该字符串可能包含恶意代码。例如，这些代码可能会删除或修改用户计算机上的文件。也可能会将本地文件的内容报告给某个不受信任的网络域。

生成动态代码的方式如下所示：

- 调用 `eval()` 函数。

- 使用 innerHTML 属性或 DOM 函数插入加载应用程序目录外部的脚本的 script 标签。
- 使用 innerHTML 属性或 DOM 函数插入具有内联代码的 script 标签（而不是通过 src 属性加载脚本）。
- 设置 src 标签的 script 属性可以加载应用程序目录外部的 JavaScript 文件。
- 使用 javascript URL 方案（如 href="javascript:alert('Test')" 所示）。
- 使用 setInterval() 或 setTimeout() 函数，其中，第一个参数（用于定义要异步运行的函数）为将求值的字符串而不是函数名（如 setTimeout('x = 4', 1000) 所示）。
- 调用 document.write() 或 document.writeln()。

加载内容时，应用程序安全沙箱中的代码只能使用这些方法。

这些限制不会阻止将 eval() 和 JSON 对象文本一起使用。这样便可以在 JSON JavaScript 库中使用应用程序内容。但是会限制您使用重载的 JSON 代码（通过事件处理函数）。

对于其它 Ajax 框架和 JavaScript 代码库，需检查框架或库中的代码是否在限制动态生成的代码时起作用。如果不起作用，则需包括在非应用程序安全沙箱中使用框架或库的所有内容。有关详细信息，请参阅第 93 页的“[非应用程序沙箱中的内容的权限](#)”和第 100 页的“[通过脚本访问应用程序和非应用程序内容](#)”。Adobe 维护一个已知的支持应用程序安全沙箱的 Ajax 框架列表，其网址为 <http://www.adobe.com/products/air/develop/ajax/features/>。

与应用程序安全沙箱中的内容不同，非应用程序安全沙箱中的 JavaScript 内容随时都可以调用 eval() 函数来执行动态生成的代码。

访问 AIR API 的限制（针对非应用程序沙箱）

非应用程序沙箱中的 JavaScript 代码无法访问 window.runtime 对象，也无法执行 AIR API。如果非应用程序安全沙箱中的内容调用以下代码，则应用程序会引发 TypeError 异常：

```
try {
    window.runtime.flash.system.NativeApplication.nativeApplication.exit();
}
catch (e)
{
    alert(e);
}
```

异常类型为 TypeError（未定义的值），由于非应用程序沙箱中的内容无法识别 window.runtime 对象，因此将其认为是未定义的值。

可以使用脚本桥将运行时功能公开给非应用程序沙箱中的内容。有关详细信息，请参阅第 100 页的“[通过脚本访问应用程序和非应用程序内容](#)”。

使用 XMLHttpRequest 调用的限制

应用程序安全沙箱中的 HTML 内容无法使用同步 XMLHttpRequest 方法，在加载 HTML 内容和执行 onLoad 事件中从应用程序沙箱外部加载数据。

默认情况下，不允许非应用程序安全沙箱中的 HTML 内容使用 JavaScript XMLHttpRequest 对象从非调用请求的域加载数据。frame 或 iframe 标签可以包括 allowcrossdomainxhr 属性。将此属性设置为任何非空值，即可允许 frame 或 iframe 中的内容使用 Javascript XMLHttpRequest 对象从非调用请求的代码域加载数据：

```
<iframe id="UI"
        src="http://example.com/ui.html"
        sandboxRoot="http://example.com/"
        allowcrossDomainxhr="true"
        documentRoot="app:/">
</iframe>
```

有关详细信息，请参阅第 98 页的“[通过脚本访问不同域中的内容](#)”。

加载 CSS、frame、iframe 和 img 元素的限制（针对非应用程序沙箱中的内容）

远程（网络）安全沙箱中的 HTML 内容只能从远程沙箱（网络 URL）加载 CSS、frame、iframe 和 img 内容。

只能与本地文件系统内容交互的沙箱、只能与远程内容交互的沙箱或受信任的本地沙箱中的 HTML 内容只能从本地沙箱（而不是应用程序或远程沙箱）加载 CSS、frame、iframe 和 img 内容。

调用 JavaScript window.open() 方法的限制

如果通过调用 JavaScript window.open() 方法创建的窗口显示非应用程序安全沙箱中的内容，则窗口的标题以主（启动）窗口的标题开头，后跟一个冒号字符。无法使用代码将窗口的标题部分从屏幕上删除。

非应用程序安全沙箱中的内容只能成功调用 JavaScript window.open() 方法来响应用户与鼠标或键盘交互所触发的事件。这会阻止非应用程序内容创建可能被欺骗使用的窗口（例如用于仿冒攻击）。此外，鼠标或键盘事件的事件处理函数无法将 window.open() 方法设置为推迟执行（例如通过调用 setTimeout() 函数）。

远程（网络）沙箱中的内容只能使用 window.open() 方法打开远程网络沙箱中的内容。无法使用 window.open() 方法打开应用程序或本地沙箱中的内容。

local-with-filesystem、local-with-networking 或 local-trusted 沙箱（请参阅第 92 页的“[沙箱](#)”）中的内容只能使用 window.open() 方法打开本地沙箱中的内容。无法使用 window.open() 打开应用程序或远程沙箱中的内容。

调用受限代码时出现的错误

如果由于这些安全限制而限制在沙箱中使用所调用的代码，则运行时将发出 JavaScript 错误：“Adobe AIR runtime security violation for JavaScript code in the application security sandbox”（应用程序安全沙箱中存在针对 JavaScript 代码的 Adobe AIR 运行时安全侵犯）。

有关详细信息，请参阅第 48 页的“[避免与安全相关的 JavaScript 错误](#)”。

从字符串中加载 HTML 内容时对沙箱的保护

通过 HTMLLoader 类的 loadString() 方法，可以在运行时创建 HTML 内容。但是，如果从不安全的 Internet 来源加载数据，则用作 HTML 内容的数据可能已损坏。由于此原因，默认情况下，使用 loadString() 方法创建的 HTML 不放置在应用程序沙箱中，并且无权访问 AIR API。但是，将 HTMLLoader 对象的 placeLoadStringContentInApplicationSandbox 属性设置为 true，即可将使用 loadString() 方法创建的 HTML 放置在应用程序沙箱中。有关详细信息，请参阅从字符串加载 HTML 内容。

通过脚本访问不同域中的内容

AIR 应用程序在安装时会被授予特殊权限。不要将相同权限泄露给其它内容（包括不属于应用程序的远程文件和本地文件），这一点很重要。

关于 AIR 沙箱桥

通常，一个域中的内容无法调用其它域中的脚本。

但是仍存在这样一些情况：主 AIR 应用程序要求远程域中的内容对主 AIR 应用程序中的脚本具有受控访问权限，反之亦然。为此，运行时提供了沙箱桥机制，沙箱桥充当两个沙箱之间的通道。沙箱桥可以在远程安全沙箱和应用程序安全沙箱之间提供显式交互。

沙箱桥公开了以下两个对象，已加载和要加载的脚本都可以访问这两个对象：

- parentSandboxBridge 对象允许要加载的内容将属性和函数公开给已加载的内容中的脚本。

- `childSandboxBridge` 对象允许已加载的内容将属性和函数公开给要加载的内容中的脚本。

通过沙箱桥公开的对象按值而不是按引用进行传递。所有数据都会序列化。这意味着由桥的一端公开的对象无法由另一端设置，并且公开的对象为无类型对象。此外，只能公开简单对象和函数；不能公开复杂对象。

如果子级内容尝试设置 `parentSandboxBridge` 对象的属性，则运行时会引发 `SecurityError` 异常。同样，如果父级内容尝试设置 `childSandboxBridge` 对象的属性，则运行时也会引发 `SecurityError` 异常。

沙箱桥示例 (HTML)

在 HTML 内容中，会将 `parentSandboxBridge` 和 `childSandboxBridge` 属性添加到子级文档的 JavaScript `window` 对象中。有关如何在 HTML 内容中设置桥函数的示例，请参阅第 58 页的“[设置沙箱桥接口](#)”。

限制 API 公开

公开沙箱桥时，公开限制沙箱桥滥用程度的高级别 API 非常重要。请注意，调用桥实施的内容可能会被破坏（例如，通过插入代码）。因此（举例来说），通过桥公开 `readFile(path)` 方法（该方法读取任意文件的内容）容易被滥用。最好公开不采用路径且读取特定文件的 `readApplicationSetting()` API。一旦应用程序部分受到损坏，语义方法越多，就越能限制应用程序产生的破坏。

另请参阅

[第 56 页的“跨脚本访问不同安全沙箱中的内容”](#)

[第 93 页的“应用程序沙箱”](#)

[第 93 页的“非应用程序沙箱中的内容的权限”](#)

写入磁盘

在 Web 浏览器中运行的应用程序只能与用户的本地文件系统进行有限的交互。Web 浏览器会实施安全策略，用于确保用户的计算机不会由于加载 Web 内容而被破坏。例如，通过 Flash Player 在浏览器中运行的 SWF 文件无法直接与用户计算机中的文件进行交互。可以将共享对象和 Cookie 写入用户的计算机，以便维护用户首选项和其它数据，但文件系统交互将受到此限制。由于 AIR 应用程序安装在本地，因此它们具有不同的安全协议，其中包括在本地文件系统间进行读取和写入的功能。

这一灵活性要求开发人员担负较高的责任。意外的应用程序不安全因素不仅会危害应用程序的功能，而且会危害用户计算机的完整性。为此，开发人员应阅读第 100 页的“[开发人员的最佳安全做法](#)”。

AIR 开发人员可以使用多个 URL 方案协议来访问文件并将文件写入本地文件系统：

URL 方案	说明
<code>app:/</code>	应用程序目录的别名。从此路径访问的文件被分配到应用程序沙箱中，并由运行时授予完全权限。
<code>app-storage:/</code>	本地存储目录的别名，由运行时进行标准化。从此路径访问的文件被分配到非应用程序沙箱中。
<code>file:///</code>	表示用户硬盘的根目录的别名。如果从此路径访问的文件位于应用程序目录中，则该文件会被分配到应用程序沙箱中，否则将被分配到非应用程序沙箱中。

注：AIR 应用程序无法使用 `app:` URL 方案修改内容。此外，由于管理员设置，只可以读取应用程序目录。

除非用户计算机存在管理员限制，否则 AIR 应用程序有权写入用户硬盘上的任意位置。建议开发人员使用 `app-storage:/` 路径作为与其应用程序相关的本地存储。从应用程序写入 `app-storage:/` 的文件放置在标准位置：

- 在 Mac OS 中：应用程序的存储目录为 `<appData>/<appId>/Local Store/`，其中 `<appData>` 表示用户的首选参数文件夹。一般此目录为 `/Users/<user>/Library/Preferences`

- 在 Windows 中：应用程序的存储目录为 <appData>\<appId>\Local Store\，其中 <appData> 表示用户的 CSIDL_APPDATA 特殊文件夹。一般此目录为 C:\Documents and Settings\<userName>\Application Data
- 在 Linux 中为 <appData>/<appId>/Local Store/，其中 <appData> 表示 /home/<user>/.appdata

如果应用程序设计用于与用户文件系统中的现有文件进行交互，请确保阅读第 100 页的“[开发人员的最佳安全做法](#)”。

安全使用不受信任的内容

未分配给应用程序沙箱的内容可以为应用程序提供其它脚本功能，但前提是满足运行时的安全条件。本主题介绍 AIR 安全协议以及非应用程序内容。

通过脚本访问应用程序和非应用程序内容

通过脚本访问应用程序和非应用程序内容的 AIR 应用程序具有更复杂的安全安排。只允许不在应用程序沙箱中的文件使用沙箱桥来访问应用程序沙箱中的文件的属性和方法。沙箱桥充当应用程序内容与非应用程序内容之间的通道，在两个文件之间提供显式交互。如果使用正确，沙箱桥会提供额外的安全层，从而限制非应用程序内容访问属于应用程序内容的对象引用。

通过示例可以更好地说明沙箱桥的优点。假设 AIR 音乐商店应用程序需要为希望创建自己的 SWF 文件的广告商提供 API，商店应用程序可以使用这些文件进行通信。该商店需要为广告商提供在商店中查找艺术家和光盘的方法，另外出于安全原因，还需要将某些方法和属性与第三方 SWF 文件进行隔离。

沙箱桥可以提供此功能。默认情况下，在运行时从外部加载到 AIR 应用程序的内容无法访问主应用程序中的任何方法或属性。通过自定义沙箱桥，开发人员可以在不公开这些方法或属性的情况下为远程内容提供服务。将沙箱桥视为受信任内容和不受信任内容之间的通道，在加载方和被加载方内容之间提供通信而不公开对象引用。

有关如何安全使用沙箱桥的详细信息，请参阅第 98 页的“[通过脚本访问不同域中的内容](#)”。

开发人员的最佳安全做法

虽然 AIR 应用程序是使用 Web 技术构建的，但开发人员应知道这些应用程序并非在浏览器安全沙箱中运行，这一点很重要。这意味着，可以构建会对本地系统有意或无意产生损害的 AIR 应用程序。AIR 会尝试最大程度降低此风险，但仍存在一些可能引入漏洞的方式。本主题介绍了重要的潜在不安全因素。

将文件导入应用程序安全沙箱的风险

位于应用程序目录中的文件会被分配到应用程序沙箱中，并具有运行时的完全权限。建议将写入本地文件系统的应用程序写入 app-storage:/ :/ 中。此目录与用户计算机上的应用程序文件位于不同的位置，因此这些文件不会分配到应用程序沙箱中，并且安全风险的程度会降低。建议开发人员考虑以下问题：

- 仅在必要时才在 AIR 文件（位于安装的应用程序中）中包含文件。
- 仅在脚本文件的行为被完全理解和信任时才在 AIR 文件（位于安装的应用程序中）中包含该脚本文件。
- 不要向应用程序目录中写入内容或修改其中的内容。运行时通过引发 SecurityError 异常，阻止应用程序使用 app:/ URL 方案写入和修改文件和目录。
- 不要将网络源中的数据用作可能引起代码异常的 AIR API 的方法的参数。这包括使用 Loader.loadBytes() 方法和 JavaScript eval() 函数。

使用外部源确定路径的风险

使用外部数据或内容可能会破坏 AIR 应用程序。因此，使用网络或文件系统中的数据时应特别小心。信任责任主要取决于开发人员以及他们建立的网络连接，但加载外来数据本身具有风险，不应将其用作敏感操作的输入。建议开发人员不要执行以下操作：

- 使用网络源中的数据确定文件名
- 使用网络源中的数据构建应用程序用来发送私人信息的 URL

使用、存储或传输不安全凭据的风险

将用户凭据存储在用户的本地文件系统中将引入可能破坏这些凭据的风险。建议开发人员考虑以下问题：

- 如果凭据必须存储在本地，请在写入本地文件系统时对凭据进行加密。运行时通过 `EncryptedLocalStore` 类提供了对每个安装的应用程序都唯一的加密存储。有关详细信息，请参阅第 246 页的“[存储加密数据](#)”。
- 不要将未加密的用户凭据传输到网络源中，除非该源受信任。
- 永远不要在创建凭据时指定默认密码，应让用户自己创建密码。保留默认值不变的用户将其凭据暴露在已了解默认密码的攻击者面前。

降级攻击的风险

在安装应用程序过程中，运行时会检查以确保应用程序的版本不是当前安装的版本。如果应用程序已经安装，则运行时会比较版本字符串与已安装的版本。如果此字符串不同，则用户可以选择升级安装。运行时不保证新安装的版本比旧版本新，仅保证版本不同。攻击者可能会向用户分发旧版本以避开安全漏洞。因此，建议开发人员在运行应用程序时检查版本。最好让应用程序检查网络中是否存在所需更新。这样，即使攻击者让用户运行旧版本，该旧版本也会识别出需要更新。此外，对应用程序使用清晰的版本控制方案可使得欺骗用户安装降级版本变得更加困难。有关提供应用程序版本的详细信息，请参阅第 103 页的“[在应用程序描述符文件中定义属性](#)”。

代码签名

所有 AIR 安装程序文件都需要进行代码签名。代码签名是一种加密过程，用于确认指定的软件源是否正确。通过从外部证书颁发机构 (CA) 链接证书或构建自己的证书，可以对 AIR 应用程序进行签名。强烈建议从已知的 CA 获得商业证书，这样的证书可以保证用户安装的是您提供的应用程序，而不是赝品。但是，可以使用 SDK 中的 `adt` 或者使用 Flash、Flex Builder 或使用 `adt` 生成证书的其它应用程序来创建自签名的证书。自签名证书不会保证要安装的应用程序为正版应用程序。

有关对 AIR 应用程序进行数字签名的详细信息，请参阅第 312 页的“[对 AIR 文件进行数字签名](#)”和第 22 页的“[使用命令行工具创建 AIR 应用程序](#)”。

第 15 章：设置 AIR 应用程序属性

除了组成 AIR 应用程序的所有文件和其他资源外，每个 AIR 应用程序还需要一个应用程序描述符文件。应用程序描述符文件是定义应用程序基本属性的 XML 文件。

开发 AIR 应用程序时，必须为每个 Adobe® AIR® 项目创建一个应用程序描述符文件。范例描述符文件 descriptor-sample.xml 位于 Adobe® AIR™ 安装的 samples 目录中。

应用程序描述符文件结构

应用程序描述符文件包含影响整个应用程序的属性，例如它的名称、版本、版权等等。任何文件名都可用于应用程序描述符文件。当使用 ADT 打包应用程序时，应用程序描述符文件会重命名为 application.xml 并放在 AIR 包内的特定目录中。当使用 Flash CS3 或 Flash CS4 中的默认设置创建 AIR 文件时，会将应用程序描述符文件重命名为 application.xml，并放置在 AIR 包中的特定目录内。

下面是一个应用程序描述符文件示例：

```
<?xml version="1.0" encoding="utf-8" ?>
<application xmlns="http://ns.adobe.com/air/application/1.5">
    <id>HelloWorld</id>
    <version>2.0</version>
    <filename>Hello World</filename>
    <name>Example Co. AIR Hello World</name>
    <description>
        <text xml:lang="en">This is a example.</text>
        <text xml:lang="fr">C'est un exemple.</text>
        <text xml:lang="es">Esto es un ejemplo.</text>
    </description>
    <copyright>Copyright (c) 2006 Example Co.</copyright>
    <initialWindow>
        <title>Hello World</title>
        <content>
            HelloWorld-debug.html
        </content>
        <systemChrome>none</systemChrome>
        <transparent>true</transparent>
        <visible>true</visible>
        <minimizable>true</minimizable>
        <maximizable>false</maximizable>
        <resizable>false</resizable>
        <width>640</width>
        <height>480</height>
        <minSize>320 240</minSize>
        <maxSize>1280 960</maxSize>
    </initialWindow>
    <installFolder>Example Co/Hello World</installFolder>
    <programMenuFolder>Example Co</programMenuFolder>
    <icon>
        <image16x16>icons/smallIcon.png</image16x16>
        <image32x32>icons/mediumIcon.png</image32x32>
    </icon>
</application>
```

```

<image48x48>icons/bigIcon.png</image48x48>
<image128x128>icons/biggestIcon.png</image128x128>
</icon>
<customUpdateUI>true</customUpdateUI>
<allowBrowserInvocation>false</allowBrowserInvocation>
<fileTypes>
  <fileType>
    <name>adobe.VideoFile</name>
    <extension>avf</extension>
    <description>Adobe Video File</description>
    <contentType>application/vnd.adobe.video-file</contentType>
    <icon>
      <image16x16>icons/avfIcon_16.png</image16x16>
      <image32x32>icons/avfIcon_32.png</image32x32>
      <image48x48>icons/avfIcon_48.png</image48x48>
      <image128x128>icons/avfIcon_128.png</image128x128>
    </icon>
  </fileType>
</fileTypes>
</application>

```

在应用程序描述符文件中定义属性

使用应用程序描述符中的 XML 元素和属性可为 AIR 应用程序定义以下类型的属性：

- 所需的 AIR 运行时版本
- 应用程序标识
- 安装和程序菜单文件夹
- 初始内容和窗口属性
- 应用程序图标文件
- 应用程序是否提供自定义更新 UI
- 用户浏览器中运行的 SWF 内容能否调用您的应用程序
- 文件类型关联

指定所需的 AIR 版本

应用程序描述符文件根元素的属性 `application` 指定了所需 AIR 运行时的版本：

```
<application xmlns="http://ns.adobe.com/air/application/1.5">
```

`xmlns` AIR 命名空间，必须将其定义为默认 XML 命名空间。该命名空间因每个 AIR 主版本而异（但不会因次要修补程序而异）。命名空间的最后一段（如“1.5”）表明应用程序所需的运行时版本。如果应用程序使用 AIR 1.5 的任何新增功能，请确保将命名空间设置为 AIR 1.5 (“<http://ns.adobe.com/air/application/1.5>”)。

对基于 SWF 的应用程序，应用程序描述符中指定的 AIR 运行时版本决定了可以作为应用程序初始内容加载的 SWF 最高版本。指定 AIR 1.0 或 AIR 1.1 的应用程序只能使用 SWF9 (Flash Player 9) 文件作为初始内容。即使运行应用程序时采用了 AIR 1.5 运行时也是如此。指定 AIR 1.5 的应用程序可以使用 SWF9 或 SWF10 (Flash Player 10) 文件作为初始内容。SWF 版本决定了 AIR 和 Flash Player API 的哪个版本可供使用。如果将 SWF9 文件用作 AIR 1.5 应用程序的初始内容，则应用程序将只能访问 AIR 1.1 和 Flash Player 9 API。此外，AIR 1.5 或 Flash Player 10 中对现有 API 行为的更改将不会生效。（此原则有一个例外，在运行时当前或今后的修补程序中，可以追溯应用对 API 在安全方面的重要更改。）

对基于 HTML 的应用程序，应用程序描述符中指定的运行时版本仅决定 AIR 和 Flash Player API 的哪个版本可供应用程序使用。HTML、CSS 和 JavaScript 的行为始终由已安装 AIR 运行时中所用 Webkit 版本决定，而非由应用程序描述符决定。

AIR 应用程序加载 SWF 内容时，可供该内容使用的 AIR 和 Flash Player API 的版本取决于内容的加载方式。下表展示了如何根据加载方法决定 API 版本：

加载内容的方式	决定 API 版本的方式
初始内容，基于 SWF 的应用程序	已加载文件的 SWF 版本
初始内容，基于 HTML 的应用程序	应用程序描述符命名空间
由 SWF 内容加载的 SWF	正在加载的内容的版本
由 HTML 内容使用 <script> 标签加载的 SWF 库	应用程序描述符命名空间
由 HTML 内容使用 AIR 或 Flash Player API（如 flash.display.Loader）加载的 SWF	应用程序描述符命名空间
由 HTML 内容使用 <object> 或 <embed> 标签（或等效的 JavaScript API）加载的 SWF	已加载文件的 SWF 版本

当正在加载的 SWF 文件与正在加载的内容具有不同的版本时，可能会遇到两种问题：

- SWF9（或更低版本）加载 SWF10 内容 - 将无法解析对所加载内容中 AIR 1.5 和 Flash Player 10 API 的引用
- SWF10 加载 SWF9（或更低版本）内容 - AIR 1.5 和 Flash Player 10 中有变化的 API 的行为可能与所加载内容的预期不符

minimumPatchLevel 可选。使用 **minimumPatchLevel** 属性可指定应用程序所需的最低 Adobe AIR 修补级别。通常，AIR 应用程序仅通过定义应用程序描述符文件中的命名空间来指定其需要的 AIR 版本。命名空间根据每个 AIR 主版本（如 1.0 或 1.5）而有所变化。该命名空间不会因修补程序版本而异。

修补程序版本仅包含一组有限的修正内容，而不包含任何 API 更改内容。通常，应用程序不指定其需要哪个修补程序版本。但是，修补程序版本中的修正内容能够修正应用程序中的某一问题。在此情况下，应用程序可以为 **minimumPatchLevel** 属性指定一个值，以确保在安装应用程序之前应用相应修补程序。如有必要，AIR 应用程序安装程序会提示用户下载并安装所需版本或修补程序。以下示例显示为 **minimumPatchLevel** 属性指定了值的应用程序元素：

```
<application xmlns="http://ns.adobe.com/air/application/1.1" minimumPatchLevel="5331">
```

定义应用程序标识

以下元素定义应用程序的 ID、版本、名称、文件名、说明和版权信息：

```
<id>TestApp</id>
<version>2.0</version>
<filename>TestApp</filename>
<name>
    <text xml:lang="en">Hello AIR</text>
    <text xml:lang="fr">Bonjour AIR</text>
    <text xml:lang="es">Hola AIR</text>
</name>
<description>An MP3 player.</description>
<copyright>Copyright (c) 2008 YourCompany, Inc.</copyright>
```

id 应用程序的标识符字符串，称为应用程序 ID。该属性的值只能使用以下字符：

- 0 - 9
- a - z
- A - Z
- . (点)
- - (连字符)

该值必须包含 1 到 212 个字符。此元素是必需的。

在所安装的应用程序中，id 字符串和发行商 ID 合起来唯一地标识应用程序。发行商 ID 以用于对应用程序进行签名的证书为基础。（有关详细信息，请参阅第 313 页的“[关于 AIR 发行商标识符](#)”。）

version 指定应用程序的版本信息。（它与运行时版本无关）。version 字符串是应用程序定义的指示符。AIR 不会以任何方式解释版本字符串。因此，不会假设版本“3.0”比版本“2.0”更新。示例：“1.0”、“.4”、“0.5”、“4.9”、“1.3.4a”。此元素是必需的。

filename 该字符串在安装应用程序时用作应用程序的文件名（不带扩展名）。该应用程序文件将在运行时启动相应 AIR 应用程序。如果未提供 name 值，则 filename 也将用作安装文件夹的名称。此元素是必需的。

filename 属性可包含任何 Unicode (UTF-8) 字符，但以下字符（在各种文件系统中都禁止将这些字符用作文件名）除外：

字符	十六进制代码
因系统而异	0x00 - x1F
*	x2A
"	x22
:	x3A
>	x3C
<	x3E
?	x3F
\	x5C
	x7C

filename 值不能以句点结尾。

name（可选，但建议使用）由 AIR 应用程序安装程序显示的标题。

如果指定单个文本节点（而非多个 text 元素），则无论系统语言为哪种语言，AIR 应用程序安装程序都将使用此名称：

```
<name>Test Application</name>
```

AIR 1.0 应用程序描述符架构只允许为该名称定义一个简单文本节点（而非多个 text 元素）。

在 AIR 1.1（或更高版本）中，您可以在 name 元素中指定多种语言。例如，以下示例用三种语言（英语、法语和西班牙语）指定名称：

```
<name>
  <text xml:lang="en">Hello AIR</text>
  <text xml:lang="fr">Bonjour AIR</text>
  <text xml:lang="es">Hola AIR</text>
</name>
```

每个文本元素的 `xml:lang` 属性用于指定语言代码，有关具体定义，请参阅 [RFC4646](http://www.ietf.org/rfc/rfc4646.txt) (<http://www.ietf.org/rfc/rfc4646.txt>)。

AIR 应用程序安装程序会使用与用户操作系统的用户界面语言最匹配的名称。例如，假如在某一安装中，应用程序描述符文件的 name 元素包含适用于 en（英语）区域设置的值。如果操作系统将 en（英语）标识为用户界面语言，则 AIR 应用程序安装程序将使用此 en 名称。如果系统用户界面语言为 en-US（美式英语），则该应用程序也使用此 en 名称。但是，如果用户界面语言为 en-US，而应用程序描述符文件同时定义了 en-US 名称和 en-GB 名称，则 AIR 应用程序安装程序将使用相应的 en-US 值。如果应用程序定义的任何名称与系统用户界面语言均不匹配，则 AIR 应用程序安装程序将使用在应用程序描述符文件中定义的第一个 name 值。

如果未指定 name 元素，则 AIR 应用程序安装程序会将 filename 显示为应用程序名称。

name 元素仅定义在 AIR 应用程序安装程序中使用的应用程序标题。AIR 应用程序安装程序支持以下多种语言：繁体中文、简体中文、捷克语、荷兰语、英语、法语、德语、意大利语、日语、朝鲜语、波兰语、巴西葡萄牙语、俄语、西班牙语、瑞典语和土耳其语。AIR 应用程序安装程序将根据系统用户界面语言来（为文本而不是应用程序标题和说明）选择其显示语言。此语言选择与应用程序描述符文件中的设置无关。

name 元素不定义可供运行的已安装应用程序使用的区域设置。有关开发多语言应用程序的详细信息，请参阅第 335 页的“[本地化 AIR 应用程序](#)”。

description（可选）将在 AIR 应用程序安装程序中显示的应用程序说明。

如果指定单个文本节点（而非多个文本元素），则无论系统语言为哪种语言，AIR 应用程序安装程序都将使用此说明：

```
<description>This is a sample AIR application.</description>
```

AIR 1.0 应用程序描述符架构只允许为该名称定义一个简单文本节点（而非多个 **text** 元素）。

在 AIR 1.1（或更高版本）中，您可以在 **description** 元素中指定多种语言。例如，以下示例用三种语言（英语、法语和西班牙语）指定说明：

```
<description>
  <text xml:lang="en">This is a example.</text>
  <text xml:lang="fr">C'est un exemple.</text>
  <text xml:lang="es">Esto es un ejemplo.</text>
</description>
```

每个文本元素的 **xml:lang** 属性用于指定语言代码，有关具体定义，请参阅 [RFC4646](#) (<http://www.ietf.org/rfc/rfc4646.txt>)。

AIR 应用程序安装程序会使用与用户操作系统的用户界面语言最匹配的说明。例如，假如在某一安装中，应用程序描述符文件的 **description** 元素包含适用于 **en**（英语）区域设置的值。如果用户系统将 **en**（英语）标识为用户界面语言，则 AIR 应用程序安装程序将使用此 **en** 名称。如果系统用户界面语言为 **en-US**（美式英语），则该应用程序也使用此 **en** 名称。但是，如果系统用户界面语言为 **en-US**，而应用程序描述符文件同时定义了 **en-US** 名称和 **en-GB** 名称，则 AIR 应用程序安装程序将使用相应的 **en-US** 值。如果应用程序定义的任何名称与系统用户界面语言均不匹配，则 AIR 应用程序安装程序将使用在应用程序描述符文件中定义的第一个 **description** 值。

有关开发多语言应用程序的详细信息，请参阅第 335 页的“[本地化 AIR 应用程序](#)”。

copyright（可选）AIR 应用程序的版权信息。在 Mac OS 中，版权文本会显示在已安装应用程序的“关于”对话框中。在 Mac OS 中，在应用程序的 Info.plist 文件中的 NSHumanReadableCopyright 字段内也使用版权信息。

定义安装文件夹和程序菜单文件夹

安装文件夹和程序菜单文件夹通过以下属性设置进行定义：

```
<installFolder>Acme</installFolder>
<programMenuFolder>Acme/Applications</programMenuFolder>
```

installFolder（可选）标识默认安装目录的子目录。

在 Windows 中，默认安装子目录为 **Program Files** 目录。在 Mac OS 中，默认安装子目录为 **/Applications** 目录。在 Linux 中为 **/opt/**。例如，如果将 **installFolder** 属性设置为 “**Acme**”，并将应用程序命名为 “**ExampleApp**”，则应用程序在 Windows 中将安装在 **C:\Program Files\Acme\ExampleApp** 中，在 Mac OS 中将安装在 **/Applications/Acme/Example.app** 中，而在 Linux 中将安装在 **/opt/Acme/ExampleApp** 中。

如果要指定嵌套子目录，请使用正斜杠（/）字符作为目录分隔符，如下所示：

```
<installFolder>Acme/Power Tools</installFolder>
```

installFolder 属性可包含任何 Unicode (UTF-8) 字符，但那些禁止在各种文件系统中用作文件夹名称的字符除外（有关例外字符的列表，请参阅上文中的 **filename** 属性）。

installFolder 属性为可选属性。如果未指定 **installFolder** 属性，则应用程序将根据 **name** 属性安装在默认安装目录的子目录中。

programMenuFolder (可选) 标识应用程序的快捷方式在 Windows 操作系统的“所有程序”菜单中或 Linux 的“应用程序”菜单中放置的位置。(目前在其他操作系统中忽略此设置。) 对该属性值中允许使用的字符的限制与对 **installFolder** 属性的限制相同。请勿将正斜杠 (/) 字符用作此值的最后一个字符。

定义初始应用程序窗口的属性

当加载 AIR 应用程序时, 运行时将使用 **initialWindow** 元素中的值为应用程序创建初始窗口。然后, 运行时会将 **content** 元素中指定的 SWF 或 HTML 文件加载到该窗口中。

以下为 **initialWindow** 元素的示例:

```
<initialWindow>
    <content>AIRTunes.html</content>
    <title>AIR Tunes</title>
    <systemChrome>none</systemChrome>
    <transparent>true</transparent>
    <visible>true</visible>
    <minimizable>true</minimizable>
    <maximizable>true</maximizable>
    <resizable>true</resizable>
    <width>400</width>
    <height>600</height>
    <x>150</x>
    <y>150</y>
    <minSize>300 300</minSize>
    <maxSize>800 800</maxSize>
</initialWindow>
```

initialWindow 元素的各子元素设置根内容文件将加载到其中的窗口的属性。

content 为 **content** 元素指定的值是应用程序主内容文件的 URL。该文件可以是 SWF 文件, 也可以是 HTML 文件。该 URL 是相对于根应用程序安装文件夹指定的。(如果使用 ADL 运行 AIR 应用程序, 则该 URL 相对于包含应用程序描述符文件的文件夹。可以使用 ADL 的 **root-dir** 参数指定其他根目录。)

注: 因为将 **content** 元素的值视为 URL, 所以必须根据 [RFC 1738](#) 中定义的规则对内容文件名称中的字符进行 URL 编码。例如, 空格字符必须编码为 %20。

title (可选) 窗口标题。

systemChrome (可选) 如果将此属性设置为 **standard**, 则将显示操作系统提供的标准系统镶边。如果将其设置为 **none**, 则不显示任何系统镶边。系统镶边设置在运行时无法更改。

transparent (可选) 如果希望应用程序窗口支持 Alpha 混合, 则设置为 "true"。透明窗口绘制起来可能比较慢且需要更多内存。透明设置在运行时无法更改。

重要说明: 只有在 **systemChrome** 为 **none** 时, 才能将 **transparent** 设置为 **true**。

visible (可选) 如果希望主窗口在创建后马上可见, 则设置为 **true**。默认值是 **false**。

您可能想要使主窗口最初保持隐藏, 以便不显示对窗口位置、窗口大小和其内容的布局所做的更改。然后, 可以通过调用窗口的 **activate()** 方法或将 **visible** 属性设置为 **true** 来显示此窗口。有关详细信息, 请参阅第 115 页的“[使用本机窗口](#)”。

x、**y**、**width**、**height** (可选) 应用程序主窗口的初始边界。如果未设置这些值, 则窗口大小将由根 SWF 文件中的设置确定, 对于 HTML, 则将由操作系统确定。**width** 和 **height** 的最大值都为 2880。

minSize、**maxSize** (可选) 窗口的最小和最大大小。如果未设置这些值, 则将由操作系统确定这些值。

minimizable、**maximizable**、**resizable** (可选) 指定窗口是否可以最小化、最大化, 以及是否可以调整大小。默认情况下, 这些设置设为 **true**。

注: 在操作系统 (例如 Mac OS X) 中, 最大化窗口是一种调整大小操作, 若要阻止窗口缩放或调整大小, **maximizable** 和 **resizable** 必须同时设置为 **false**。

指定图标文件

`icon` 属性指定一个或多个要用于应用程序的图标文件。包含图标是可选的。如果未指定 `icon` 属性，则操作系统将显示默认图标。

指定路径相对于应用程序的根目录。图标文件必须为 PNG 格式。可以指定以下所有图标尺寸：

```
<icon>
  <image16x16>icons/smallIcon.png</image16x16>
  <image32x32>icons/mediumIcon.png</image32x32>
  <image48x48>icons/bigIcon.png</image48x48>
  <image128x128>icons/biggestIcon.png</image128x128>
</icon>
```

如果存在用于指定一定尺寸的元素，则文件中的图像必须与指定尺寸完全相同。如果所有尺寸都未提供，则操作系统会将图像缩放为适合图标给定用途的最接近的大小。

注：指定图标不会自动添加到 AIR 包中。打包应用程序时，图标文件必须包含在其正确的相对位置中。

为获得最佳效果，为每种可用尺寸都提供一个图像。此外，请确保图标在 16 位和 32 位颜色模式下看上去都像一回事。

提供针对应用程序更新的自定义用户界面

AIR 使用默认安装对话框安装和更新应用程序。但是，您可以为更新应用程序提供您自己的用户界面。若要指示让应用程序自身处理更新过程，请将 `customUpdateUI` 元素设置为 `true`：

```
<customUpdateUI>true</customUpdateUI>
```

如果应用程序的已安装版本将 `customUpdateUI` 元素设置为 `true`，则当用户双击新版本的 AIR 文件或使用无缝安装功能安装应用程序的更新时，运行时将打开应用程序的已安装版本，而非默认的 AIR 应用程序安装程序。然后，您的应用程序逻辑可以确定如何继续执行更新操作。（AIR 文件中的应用程序 ID 和发行商 ID 必须与已安装应用程序中的相应 ID 匹配才能继续进行升级。）

注：仅当应用程序已经安装并且用户双击包含更新的 AIR 安装文件或使用无缝安装功能安装应用程序的更新时，`customUpdateUI` 机制才能发挥作用。无论 `customUpdateUI` 是否为 `true`，您都可以通过您自己的应用程序逻辑下载并启动更新，并在必要时显示自定义 UI。

有关详细信息，请参阅第 318 页的“[更新 AIR 应用程序](#)”。

允许应用程序的浏览器调用

如果指定以下设置，则可以通过浏览器调用功能（由用户单击 Web 浏览器中某页中的链接）来启动已安装 AIR 应用程序：

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

默认值是 `false`。

如果将该值设置为 `true`，请务必考虑安全隐患，如第 282 页的“[浏览器调用](#)”所述。

有关详细信息，请参阅第 306 页的“[从网页安装和运行 AIR 应用程序](#)”。

声明文件类型关联

利用 `fileTypes` 元素可声明 AIR 应用程序可以与其关联的文件类型。当安装某个 AIR 应用程序时，任何已声明文件类型都会注册到操作系统，并且如果这些文件类型尚未与其他应用程序关联，则它们将与该 AIR 应用程序关联。若要取代某个文件类型和其它应用程序之间的现有关联，请在运行时使用 `NativeApplication.setAsDefaultApplication()` 方法（最好以用户权限）。

注：运行时方法只能管理应用程序描述符中声明的文件类型的关联。

```
<fileTypes>
  <fileType>
    <name>adobe.VideoFile</name>
    <extension>avf</extension>
    <description>Adobe Video File</description>
    <contentType>application/vnd.adobe.video-file</contentType>
    <icon>
      <image16x16>icons/AIRApp_16.png</image16x16>
      <image32x32>icons/AIRApp_32.png</image32x32>
      <image48x48>icons/AIRApp_48.png</image48x48>
      <image128x128>icons/AIRApp_128.png</image128x128>
    </icon>
  </fileType>
</fileTypes>
```

`fileTypes` 元素为可选元素。它可以包含任意数量的 `fileType` 元素。

对于包含的每个 `fileType` 声明，`name` 和 `extension` 元素都是必需的。同一名称可用于多个扩展名。扩展名唯一标识文件类型。（请注意，指定扩展名时前面不加句点。）`description` 元素是可选元素，并且用户可通过操作系统用户界面看到该元素的内容。`contentType` 在 AIR 1.5 中为必需（在 AIR 1.0 和 1.1 中为可选）。属性有助于操作系统在某些情况下找到打开文件的最佳应用程序。该值应为文件内容的 MIME 类型。注意，如果文件类型已注册，且具有已分配的 MIME 类型，则在 Linux 中将忽略该值。

可以为文件扩展名指定图标，所使用格式与应用程序 `icon` 元素相同。图标文件也必须包含在 AIR 安装文件中（它们不会自动打包）。

如果一个文件类型与某个 AIR 应用程序关联，则只要用户打开该类型的某个文件就会调用该应用程序。如果该应用程序已经运行，则 AIR 将针对该运行实例调度 `InvokeEvent` 对象。否则，AIR 将首先启动该应用程序。在这两种情况下，均可从 `NativeApplication` 对象调度的 `InvokeEvent` 对象中检索相应文件路径。可以使用此路径打开相应文件。

有关详细信息，请参阅第 286 页的“[管理文件关联](#)”和第 279 页的“[捕获命令行参数](#)”。

第 16 章：针对 JavaScript 开发人员的 ActionScript 基础知识

Adobe® ActionScript® 3.0 是一种与 JavaScript 类似的编程语言，它们均基于 ECMAScript。ActionScript 3.0 是随 Adobe® Flash® Player 9 一起发布的，因此您可以使用它在 Adobe® Flash® CS3 Professional、Adobe® Flash® CS4 Professional 和 Adobe® Flex™ 3 中开发富 Internet 应用程序。

仅当为浏览器中的 Flash Player 9 开发 SWF 内容时，当前版本的 ActionScript 3.0 才可用。目前，它还可用于开发在 Adobe® AIR® 中运行的 SWF 内容。

[针对 HTML 开发人员的 Adobe AIR 语言参考](#)包括有关基于 HTML 的应用程序中 JavaScript 代码所能使用的那些类的文档。这些类只是运行时中整个类集中的一部分。运行时中的其他类在开发基于 SWF 的应用程序时非常有用（例如，DisplayObject 类可以定义可视内容的结构）。如果您需要在 JavaScript 中使用这些类，请参阅以下 ActionScript 文档：

- [Adobe ActionScript 3.0 编程](#)
- [Flex 3 语言参考](#)。（只有 flash 包中的顶级类和函数可用于在 AIR 中运行的 HTML 内容。mx 包中的类仅可用于基于 Flex 的 SWF 应用程序。）

ActionScript 和 JavaScript 之间的差异：概述

与 JavaScript 类似，ActionScript 也基于 ECMAScript 语言规范；因此这两种语言包含公用的核心语法。例如，以下代码在 JavaScript 和 ActionScript 中的功能是一样的：

```
var str1 = "hello";
var str2 = " world.";
var str = reverseString(str1 + str2);

function reverseString(s) {
    var newString = "";
    var i;
    for (i = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

不过，这两种语言在语法和工作方式上存在差异。例如，如果使用 ActionScript 3.0，则前面的代码示例可以写成如下形式（在 SWF 文件中）：

```
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

Adobe AIR 中的 HTML 内容所支持的 JavaScript 版本为 JavaScript 1.7。本主题介绍了 JavaScript 1.7 和 ActionScript 3.0 之间的差异。

运行时包括一些可提供高级功能的内置类。在运行时，HTML 页中的 JavaScript 可以访问这些类。相同的运行时类既可用于 ActionScript（在 SWF 文件中），也可用于 JavaScript（在浏览器上运行的 HTML 文件中）。不过，目前关于这些类的 API 文档（未包括在针对 HTML 开发人员的 Adobe AIR 语言参考中）介绍了如何使用 ActionScript 语法。换句话说，若要了解运行时的某些高级功能，请参阅《Adobe AIR ActionScript 3.0 语言参考》。了解 ActionScript 的基础知识有助于您了解如何在 JavaScript 中使用这些运行时类。

例如，以下 JavaScript 代码可播放 MP3 文件的声音：

```
var file = air.File.userDirectory.resolve("My Music/test.mp3");
var sound = air.Sound(file);
sound.play();
```

基中每个代码行通过 JavaScript 调用运行时功能。

在 SWF 文件中，ActionScript 代码可以访问这些运行时功能，如下面的代码所示：

```
var file:File = File.userDirectory.resolve("My Music/test.mp3");
var sound = new Sound(file);
sound.play();
```

ActionScript 3.0 数据类型

ActionScript 3.0 是一种强类型语言。这意味着您可以为变量指定数据类型。例如，前面示例中的第一行可以写成如下形式：

```
var str1:String = "hello";
```

其中 str1 变量被声明为 String 类型。针对 str1 变量的所有后续赋值都将向该变量赋予 String 值。

您可以为变量、函数参数和函数返回类型指定类型。因此，前面示例中的函数声明在 ActionScript 中将如下所示：

```
function reverseString(s:String):String {
    var newString:String = "";
    for (var i:int = s.length - 1; i >= 0; i--) {
        newString += s.charAt(i);
    }
    return newString;
}
```

注：s 参数和该函数的返回值都被指定为 String 类型。

尽管指定类型在 ActionScript 中是一项可选操作，但声明对象类型具有以下优势：

- 如果指定对象的类型，则不仅允许在运行时对数据进行类型检查，而且当您使用严格模式时还将允许在编译时对数据进行类型检查，这将帮助您识别错误。（严格模式属于编译器选项。）
- 使用具有类型的对象可创建更有效的应用程序。

为此，ActionScript 文档中的示例将使用数据类型。通常，只要删除类型声明（如“:String”），就可以将示例 ActionScript 代码转换为 JavaScript。

与自定义类相对应的数据类型

ActionScript 3.0 对象可以具有与顶级类相对应的数据类型，如 String、Number 或 Date。

在 ActionScript 3.0 中，您可以定义自定义类。每个自定义类同样定义一个数据类型。这意味着 ActionScript 变量、函数参数或函数返回值可以具有由该类定义的类型注释。有关详细信息，请参阅第 112 页的“[ActionScript 3.0 自定义类](#)”。

void 数据类型

void 数据类型用作符合如下条件的函数的返回值：函数实际上不返回任何值（即函数不包含 return 语句）。

* 数据类型

将星号字符 (*) 用作数据类型等同于未指定数据类型。例如，以下函数包括参数 n 和返回值，二者均未指定数据类型：

```
function exampleFunction(n:*):* {
    trace("hi, " + n);
}
```

将 * 用作数据类型表示根本没有定义数据类型。如果在 ActionScript 3.0 代码中使用星号，则表明未定义任何数据类型。

ActionScript 3.0 类、包和命名空间

ActionScript 3.0 包含 JavaScript 1.7 中所没有的类的相关功能。

运行时类

运行时包括内置类，其中许多内置类也包含在标准 JavaScript 中，如 Array、Date、Math 和 String 类（以及其他类）。不过，运行时还包括标准 JavaScript 中没有的类；这些类具有广泛的用途，例如从播放富媒体（如声音）到与套接字进行交互等。

大多数运行时类包含在 flash 包中，或者包含在 flash 包内的某个包中。包是用来组织 ActionScript 3.0 类的一种方式（请参阅第 112 页的“[ActionScript 3.0 包](#)”）。

ActionScript 3.0 自定义类

开发人员可通过 ActionScript 3.0 创建他们自己的自定义类。例如，以下代码定义了名为 ExampleClass 的自定义类：

```
public class ExampleClass {
    public var x:Number;
    public function ExampleClass(input:Number):void {
        x = input;
    }
    public function greet():void {
        trace("The value of x is: ", x);
    }
}
```

此类具有以下成员：

- 构造函数方法 ExampleClass()，可用于实例化 ExampleClass 类型的新对象。
- 公共属性 x（为 Number 类型），可以为 ExampleClass 类型的对象获取和设置该属性。
- 公共方法 greet()，可以对 ExampleClass 类型的对象调用该方法。

在此示例中，x 属性和 greet() 方法位于 public 命名空间中，从而可以从该类之外的对象和类中访问它们。

ActionScript 3.0 包

包提供了用来组织 ActionScript 3.0 类的方法。例如，与安装了 AIR 应用程序的计算机上的文件和目录处理有关的许多类包含在 flash.filesystem 包中。此时，flash 包中包含了另一个包 filesystem。该包可能包含其他类或包。实际上，flash.filesystem 包中包含以下类：File、 FileMode 和 FileStream。若要引用 ActionScript 中的 File 类，可以编写以下代码：

```
flash.filesystem.File
```

可以在包中同时包含内置类和自定义类。

当从 JavaScript 中引用 ActionScript 包时, 请使用特殊的 runtime 对象。例如, 以下代码将在 JavaScript 中对新的 ActionScript File 对象进行实例化:

```
var myFile = new air.flash.filesystem.File();
```

在这里, File() 方法是与同名类 (File) 相对应的构造函数。

ActionScript 3.0 命名空间

在 ActionScript 3.0 中, 命名空间定义类中可以访问的属性和函数的范围。

只有那些 public 命名空间中的属性和方法在 JavaScript 中可用。

例如, File 类 (在 flash.filesystem 包中) 包括 public 属性和方法, 如 userDirectory 和 resolve()。二者均作为 JavaScript 变量的属性提供, 该变量可实例化 File 对象 (通过 runtime.flash.filesystem.File() 构造函数方法)。

有四种预定义的命名空间:

命名空间	说明
public	任何用于实例化特定类型对象的代码都可以访问用来定义该类型的类的公共属性和方法。此外, 任何代码都可以访问公共类的公共静态属性和方法。
private	指定为 private 的属性和方法仅可用于该类中的代码。它们不能作为该类定义的对象的属性或方法进行访问。private 命名空间中的属性和方法在 JavaScript 中不可用。
protected	指定为 protected 的属性和方法仅可用于类定义中的代码和继承该类的类。protected 命名空间中的属性和方法在 JavaScript 中不可用。
internal	指定为 internal 的属性和方法可用于同一包中的任意调用者。默认情况下, 类、属性和方法属于 internal 命名空间。

另外, 自定义类可以使用不可用于 JavaScript 代码的其他命名空间。

ActionScript 3.0 函数中的必需参数和默认值

在 ActionScript 3.0 和 JavaScript 中, 函数可以包括参数。在 ActionScript 3.0 中, 参数可以是必需参数, 也可以是可选参数; 而在 JavaScript 中, 参数始终是可选参数。

以下 ActionScript 3.0 代码定义了一个函数, 其中一个参数 n 是必需参数:

```
function cube(n:Number) :Number {
    return n*n*n;
}
```

以下 ActionScript 3.0 代码定义了一个函数, 其中 n 参数是必需参数, 而 p 参数是可选参数, 默认值为 1:

```
function root(n:Number, p:Number = 1):Number {
    return Math.pow(n, 1/p);
}
```

ActionScript 3.0 函数还可以接收任意数量的参数, 这些参数通过在参数列表的结尾处使用 ...rest 语法来表示, 如下所示:

```
function average(... args) : Number{
    var sum:Number = 0;
    for (var i:int = 0; i < args.length; i++) {
        sum += args[i];
    }
    return (sum / args.length);
}
```

ActionScript 3.0 事件侦听器

在 ActionScript 3.0 编程中，将使用事件侦听器 处理所有事件。事件侦听器是一个函数。当对象调度事件时，事件侦听器将响应该事件。作为 ActionScript 对象的事件将以函数参数的形式传递给事件侦听器，这与 JavaScript 中使用的 DOM 事件模型不同。

例如，当您调用 Sound 对象的 load() 方法（加载一个 MP3 文件）时，Sound 对象将尝试加载声音，然后调度下列事件中的任一事件：

事件	说明
complete	成功加载数据后调度。
id3	MP3 ID3 数据可用时调度。
ioError	在出现输入 / 输出错误并由此导致加载操作失败时调度。
open	在加载操作开始时调度。
progress	在加载操作进行过程中接收到数据时调度。

任何可以调度事件的类可扩展 EventDispatcher 类或实现 IEventDispatcher 接口。（ActionScript 3.0 接口是用于定义可以由类实现的方法集的数据类型。）在 ActionScript 语言参考中，针对这些类的每个类列表中都包含一组该类可以调度的事件。

您可以注册事件侦听器函数，以使用调度该事件的对象的 addEventListener() 方法处理任何事件。例如，就 Sound 对象来说，您可以注册 progress 和 complete 事件，如以下 ActionScript 代码所示：

```
var sound:Sound = new Sound();
var urlReq:URLRequest = new URLRequest("test.mp3");
sound.load(urlReq);
sound.addEventListener(ProgressEvent.PROGRESS, progressHandler);
sound.addEventListener(Event.COMPLETE, completeHandler);

function progressHandler(progressEvent):void {
    trace("Progress " + progressEvent.bytesTotal + " bytes out of " + progressEvent.bytesTotal);
}

function completeHandler(completeEvent):void {
    trace("Sound loaded.");
}
```

在 AIR 中运行的 HTML 内容中，您可以将 JavaScript 函数注册为事件侦听器，如以下代码所示（假定 HTML 文档包含名为 progressTextArea 的 TextArea 对象）：

```
var sound = new runtime.flash.media.Sound();
var urlReq = new runtime.flash.net.URLRequest("test.mp3");
sound.load(urlReq);
sound.addEventListener(runtime.flash.events.ProgressEvent.PROGRESS, progressHandler);
sound.addEventListener(runtime.flash.events.Event.COMPLETE, completeHandler);

function progressHandler(progressEvent) {
    document.progressTextArea.value += "Progress " + progressEvent.bytesTotal + " bytes out of " +
    progressEvent.bytesTotal;
}

function completeHandler(completeEvent) {
    document.progressTextArea.value += "Sound loaded.";
```

第 17 章：使用本机窗口

使用 Adobe® AIR® 本机窗口 API 提供的类可创建和管理桌面窗口。

有关本机窗口的其它在线信息

可以从以下源中查找有关本机窗口 API 和使用本机窗口的详细信息：

快速入门（**Adobe AIR** 开发人员中心）

- [自定义窗口的外观](#)

语言参考

- [NativeWindow](#)
- [NativeWindowInitOptions](#)

Adobe 开发人员中心文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR 窗口”）](#)

AIR 窗口基础知识

AIR 提供易于使用的跨平台窗口 API，以便使用 Flash®、Flex™ 和 HTML 编程技术创建本机操作系统窗口。

使用 AIR 可使您在开发应用程序的外观时具有广泛的自由度。您创建的窗口可以类似于标准的桌面应用程序，也就是在 Mac 上运行时与 Apple 风格相媲美、在 Windows 上运行时符合 Microsoft 惯例以及在 Linux 上与窗口管理器协调一致，所有这些的实现都不需要撰写平台专用的代码。此外，无论应用程序运行于何处，都可以使用 Flex 框架提供的可设置外观、可扩展的镶边树立您自己的风格。由于完全支持针对桌面进行透明度和 Alpha 混合，因此甚至可以用矢量图和位图绘制您自己的窗口镶边。是否厌倦了矩形窗口？现在可以绘制圆形窗口。

AIR 中的窗口

AIR 支持三个不同的 API 来处理窗口：

- 面向 ActionScript 的 NativeWindow 类提供最底层的窗口 API。在 ActionScript 和 Flash CS 创作的应用程序中使用 NativeWindow。考虑扩展 NativeWindow 类，以使应用程序中使用的窗口专用化。
- Flex 框架 mx:WindowedApplication 和 mx:Window 类为 NativeWindow 类提供 Flex“包装”。用 Flex 创建 AIR 应用程序时，WindowedApplication 组件将代替 Application 组件，并且必须始终使用前者作为您的 Flex 应用程序的初始窗口。
- 在 HTML 环境中，可以使用 JavaScript Window 类，就像在基于浏览器的 Web 应用程序中的那样。对 JavaScript Window 方法的调用将转移到基础本机窗口对象。

ActionScript 窗口

使用 NativeWindow 类创建窗口时，会直接使用 Flash Player 舞台并显示列表。若要向 NativeWindow 添加视觉对象，请将该对象添加到窗口舞台的显示列表或添加到舞台上的另一个显示对象容器。

Flex Framework 窗口

Flex Framework 定义其自己的窗口组件。 `mx:WindowedApplication` 和 `mx:Window` 组件无法在框架外部使用，因此无法在基于 HTML 的 AIR 应用程序中使用。

HTML 窗口

在创建 HTML 窗口时，可使用 HTML、CSS 和 JavaScript 来显示内容。若要向 HTML 窗口添加可视对象，请将该内容添加到 HTML DOM。HTML 窗口是一种特殊类别的 NativeWindow。AIR 主机定义 HTML 窗口中的 `nativeWindow` 属性，该属性提供对基础 NativeWindow 实例的访问。使用此属性可以访问此处所述的 NativeWindow 属性、方法和事件。

注：JavaScript Window 对象还具有用于脚本访问包含窗口的方法，例如 `moveTo()` 和 `close()`。如果多个方法均可用，您可以使用最简便的方法。

初始应用程序窗口

应用程序的第一个窗口是由 AIR 自动为您创建的。AIR 使用应用程序描述符文件的 `initialWindow` 元素中指定的参数设置该窗口的属性和内容。

如果根内容是 SWF 文件，则 AIR 将创建 NativeWindow 实例，加载 SWF 文件并将其添加到窗口舞台。如果根内容是 HTML 文件，则 AIR 将创建 HTML 窗口并加载 HTML。

有关应用程序描述符中指定的窗口属性的详细信息，请参阅第 102 页的“[应用程序描述符文件结构](#)”。

本机窗口类

本机窗口 API 包含以下类：

包	类
<code>flash.display</code>	<ul style="list-style-type: none"> • <code>NativeWindow</code> • <code>NativeWindowInitOptions</code> • <code>NativeWindowDisplayState</code> • <code>NativeWindowResize</code> • <code>NativeWindowSystemChrome</code> • <code>NativeWindowType</code> <p>窗口字符串常量在以下类中定义：</p> <ul style="list-style-type: none"> • <code>NativeWindowDisplayState</code> • <code>NativeWindowResize</code> • <code>NativeWindowSystemChrome</code> • <code>NativeWindowType</code>
<code>flash.events</code>	<ul style="list-style-type: none"> • <code>NativeWindowBoundsEvent</code> • <code>NativeWindowDisplayStateEvent</code>

本机窗口事件流

本机窗口调度事件，以便通知感兴趣的组件将要发生或已发生重要更改。对于许多与窗口相关的事件的调度是成对进行的。第一个事件警告即将发生更改。第二个事件通知已完成更改。可以取消警告事件，但不能取消通知事件。以下序列说明在用户单击窗口的最大化按钮后发生的事件流：

- 1 NativeWindow 对象调度 displayStateChanging 事件。
- 2 如果已注册的侦听器均未取消该事件，则窗口将最大化。
- 3 NativeWindow 对象调度 displayStateChange 事件。

此外，NativeWindow 对象还调度对窗口大小和位置进行相关更改的事件。窗口不调度这些相关更改的警告事件。相关事件包括：

- a move 事件，如果窗口的左上角由于最大化操作而发生移动，则调度该事件。
- b resize 事件，如果窗口大小由于最大化操作而发生更改，则调度该事件。

在最小化、还原、关闭、移动窗口和调整窗口大小时，NativeWindow 对象调度相似序列的事件。

在通过窗口镶边或其它操作系统控制的机制启动更改时，仅调度警告事件。在调用窗口方法以更改窗口大小、位置或显示状态时，窗口仅调度通知更改的事件。如果需要，可以使用窗口 dispatchEvent() 方法调度警告事件，然后检查在继续进行更改之前是否取消了警告事件。

有关窗口 API 类、方法、属性和事件的详细信息，请参阅针对 HTML 开发人员的 Adobe AIR 语言参考 (http://www.adobe.com/go/learn_air_html_jslr_cn)。

有关使用 Flash 显示列表的常规信息，请参阅《[Adobe ActionScript 3.0 编程](#)》参考 (http://www.adobe.com/go/learn_fl_cs4_programmingAS3_cn) 的“显示编程”一节。

控制本机窗口样式和行为的属性

以下属性控制窗口的基本外观和行为：

- type
- systemChrome
- transparent

创建窗口时，在传递到 window 构造函数的 NativeWindowInitOptions 对象上设置这些属性。AIR 从应用程序描述符中读取初始应用程序窗口的属性（不包括 type 属性，该属性无法在应用程序描述符中设置且始终设置为 normal）。窗口创建后将无法更改这些属性。

这些属性的一些设置互不兼容：在 transparent 为 true 或 type 为 lightweight 时，systemChrome 无法设置为 standard。

窗口类型

AIR 窗口类型组合本机操作系统的镶边属性和可见性属性来创建三种功能类型的窗口。使用 NativeWindowType 类中定义的常量可引用代码中的类型名称。AIR 提供以下窗口类型：

类型	说明
Normal (普通)	典型窗口。普通窗口使用标准尺寸样式的镶边，并显示在 Windows 的任务栏中和 Mac OS X 的窗口菜单中。
Utility (实用程序)	工具面板。实用程序窗口使用较细的系统镶边，而且不显示在 Windows 的任务栏中和 Mac OS X 的窗口菜单中。
Lightweight (简单)	简单窗口没有镶边，而且不显示在 Windows 的任务栏中和 Mac OS X 的窗口菜单中。此外，Windows 中的简单窗口没有“系统”(Alt+Space) 菜单。简单窗口适用于通知气泡和控件，例如用于打开短期显示区域的组合框。在使用的 type 为简单时，systemChrome 必须设置为 none。

窗口镶边

窗口镶边是一组使用户可以在桌面环境中操作窗口的控件。镶边元素包括标题栏、标题栏按钮、边框和调整大小手柄。

系统镶边

可以将 `systemChrome` 属性设置为 `standard` 或 `none`。选择 `standard` 系统镶边可为窗口提供一组由用户的操作系统创建和设置样式的标准控件。选择 `none` 可为窗口提供您自己的镶边。使用 `NativeWindowSystemChrome` 类中定义的常量可引用代码中的系统镶边设置。

系统镶边由系统管理。应用程序无法直接访问控件本身，但在使用控件时可以响应调度的事件。对窗口使用标准镶边时，`transparent` 属性必须设置为 `false`，`type` 属性必须设置为 `normal` 或 `utility`。

自定义镶边

创建不带系统镶边的窗口时，您必须添加自己的镶边控件才能处理用户和该窗口之间的交互。还可以根据需要随意创建透明的非矩形窗口。

窗口透明度

若要允许窗口与桌面或其它窗口进行 Alpha 混合，请将该窗口的 `transparent` 属性设置为 `true`。必须在创建窗口之前设置 `transparent` 属性，否则将无法更改该属性。

透明窗口没有默认背景。不包含应用程序所绘制对象的任何窗口区域都不可见。如果所显示对象的 Alpha 设置小于一，则该对象下方的任何内容都会显示出来，包括同一窗口中的其它显示对象、其它窗口和桌面。呈现经过 Alpha 混合的较大区域可能会很慢，因此应谨慎使用该效果。

在希望创建具有不规则形状边框、“淡出”边框或显示为不可见的边框的应用程序时，透明窗口非常有用。

重要说明：在 Linux 中，不能穿过完全透明的像素传递鼠标事件。应避免用完全透明的大型区域创建窗口，因为可能会在无法察觉的情况下阻止用户访问其它窗口或其桌面上的项目。在 Mac OS X 和 Windows 中，可以穿过完全透明的像素传递鼠标事件。

不能对具有系统镶边的窗口使用透明度。此外，透明窗口中不显示 HTML 中的 SWF 内容和 PDF 内容。有关详细信息，请参阅第 80 页的“[在 HTML 页中加载 SWF 或 PDF 内容时的注意事项](#)”。

在一些操作系统中，由于硬件或软件配置或者用户显示选项的原因，可能不支持透明度。在不支持透明度时，应用程序将与黑色背景合成。在这些情况下，应用程序的任何透明区域都显示为不透明的黑色。

静态 `NativeWindow.supportsTransparency` 属性可报告窗口透明度是否可用。例如，如果此属性测试为 `false`，则会向用户显示警告对话框，或显示备用的矩形非透明用户界面。请注意，Mac 和 Windows 操作系统始终支持透明度。支持 Linux 操作系统需要使用合成窗口管理器，但即使有合成窗口管理器处于活动状态，透明度也可能因用户显示选项或硬件配置而不可用。

HTML 应用程序窗口中的透明度

默认情况下，即使包含窗口是透明的，HTML 窗口和 `HTMLLoader` 对象中所显示的 HTML 内容的背景也是不透明的。若要关闭为 HTML 内容显示的默认背景，请将 `paintsDefaultBackground` 属性设置为 `false`。以下示例创建 `HTMLLoader` 并关闭默认背景：

```
var htmlView:HTMLLoader = new HTMLLoader();
htmlView.paintsDefaultBackground = false;
```

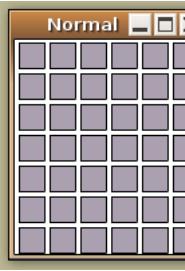
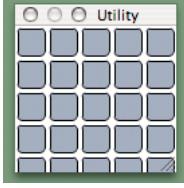
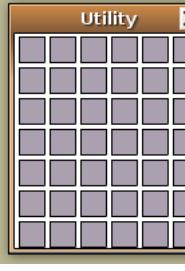
此示例使用 JavaScript 来关闭 HTML 窗口的默认背景：

```
window.htmlLoader.paintsDefaultBackground = false;
```

如果 HTML 文档中的元素设置背景颜色，则该元素的背景不透明。不支持设置局部透明度（或不透明度）值。但是，可以使用透明 PNG 格式的图形作为页面或页面元素的背景以实现相似的视觉效果。

可视窗口目录

下表说明了窗口属性设置的不同组合在 Mac OS X、Windows 和 Linux 操作系统中的视觉效果：

窗口设置	Mac OS X	Microsoft Windows	Linux*
Type: normal SystemChrome: standard Transparent: false			
Type: utility SystemChrome: standard Transparent: false			

窗口设置	Mac OS X	Microsoft Windows	Linux*
Type: Any SystemChrome: none Transparent: false			
Type: Any SystemChrome: none Transparent: true			
mxWindowedApplication 或 mx:Window Type: Any SystemChrome: none Transparent: true			

*Ubuntu (带有 Compiz 窗口管理器)

注: AIR 不支持以下系统镶边元素: Mac OS X 工具栏、 Mac OS X 代理图标、 Windows 标题栏图标以及替代系统镶边。

创建窗口

AIR 自动创建应用程序的第一个窗口, 但您可以创建所需的任何其它窗口。若要创建本机窗口, 请使用 NativeWindow 构造函数方法。若要创建 HTML 窗口, 请使用 HTMLLoader createRootWindow() 方法, 或者从 HTML 文档中调用 JavaScript window.open() 方法。

指定窗口初始化属性

在创建桌面窗口后, 无法更改窗口的初始化属性。这些不可改变的属性及其默认值包括:

属性	默认值
systemChrome	standard
type	normal
transparent	false
maximizable	true
minimizable	true
resizable	true

在应用程序描述符文件中设置 AIR 所创建的初始窗口的属性。AIR 应用程序主窗口的 type 值始终是 normal。(可以在描述符文件中指定其它窗口属性，例如 visible、width 和 height，但这些属性随时可能发生变化。)

使用 NativeWindowInitOptions 类可设置应用程序创建的其它本机窗口和 HTML 窗口的属性。在创建窗口时，必须将指定窗口属性的 NativeWindowInitOptions 对象传递给 NativeWindow 构造函数或 HTMLLoader createRootWindow() 方法。

以下代码为实用程序窗口创建 NativeWindowInitOptions 对象：

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
options.type = air.NativeWindowType.UTILITY
options.transparent = false;
options.resizable = false;
options.maximizable = false;
```

当 transparent 为 true 或 type 为 lightweight 时，不支持将 systemChrome 设置为 standard。

注：无法为使用 JavaScript window.open() 函数创建的窗口设置初始化属性。但是，您可以通过实现自己的 HTMLHost 类来覆盖这些窗口的创建方式。有关详细信息，请参阅第 85 页的“[处理对 window.open\(\) 的 JavaScript 调用](#)”。

创建初始应用程序窗口

对应用程序的初始窗口使用标准 HTML 页。此页是从应用程序安装目录中加载的并放入应用程序沙箱中。该页用作应用程序的初始入口点。

在应用程序启动时，AIR 将创建窗口、设置 HTML 环境并加载 HTML 页。在分析任何脚本或向 HTML DOM 添加任何元素之前，AIR 将 runtime、htmlLoader 和 nativeWindow 属性添加到 JavaScript Window 对象。可以使用这些属性从 JavaScript 中访问运行时类。nativeWindow 属性使您可以直接访问桌面窗口的属性和方法。

以下示例说明用 HTML 所构建 AIR 应用程序的主页的基本框架：该页等待 JavaScript 窗口 load 事件，然后显示本机窗口。

```
<html>
  <head>
    <script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
    <script language="javascript">
      window.onload=init;

      function init(){
        window.nativeWindow.activate();
      }
    </script>
  </head>
  <body></body>
</html>
```

创建 NativeWindow

若要创建 NativeWindow，请将 NativeWindowInitOptions 对象传递到 NativeWindow 构造函数：

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
options.transparent = false;
var newWindow = new air.NativeWindow(options);
```

直到将 visible 属性设置为 true 或调用 activate() 方法后，才显示窗口。

创建窗口后，可以使用舞台属性和 Flash 显示列表技术来初始化其属性和将内容加载到该窗口中。

几乎在所有情况下，都应将新本机窗口的舞台 scaleMode 属性设置为 noScale（使用 StageScaleMode.NO_SCALE 常量）。Flash 缩放模式旨在用于应用程序作者事先不知道应用程序显示区高宽比的情况。使用这些缩放模式，作者可以选择最少的内容损失：剪辑内容、拉伸或压缩内容或者用空白空间进行填充。由于您控制 AIR（窗口帧）中的显示区，因此可以在不损失内容的情况下使窗口大小适合内容或者使内容大小适合窗口。

HTML 窗口的缩放模式自动设置为 noScale。

注：若要确定当前操作系统中允许的最大窗口大小和最小窗口大小，请使用以下静态 NativeWindow 属性：

```
var maxOSSize = air.NativeWindow.systemMaxSize;
var minOSSize = air.NativeWindow.systemMinSize;
```

创建 HTML 窗口

若要创建 HTML 窗口，可以调用 JavaScript Window.open() 方法，也可以调用 AIR HTMLLoader 类的 createRootWindow() 方法。

任何安全沙箱中的 HTML 内容都可以使用标准 JavaScript Window.open() 方法。如果内容在应用程序沙箱外部运行，则只能调用 open() 方法来响应用户交互，例如鼠标单击或按键。在调用 open() 时，将创建具有系统边框的窗口以显示指定 URL 处的内容。例如：

```
newWindow = window.open("xmpl.html", "logWindow", "height=600, width=400, top=10, left=10");
```

注：可以扩展 ActionScript 中的 HTMLHost 类，以便自定义用 JavaScript window.open() 函数创建的窗口。请参阅第 82 页的“[关于扩展 HTMLHost 类](#)”。

应用程序安全沙箱中的内容可以访问更强大的窗口创建方法 HTMLLoader.createRootWindow()。使用此方法，可以指定新窗口的所有创建选项。例如，以下 JavaScript 代码创建不具有大小为 300x400 像素的系统边框的简单类型窗口：

```
var options = new air.NativeWindowInitOptions();
options.systemChrome = "none";
options.type = "lightweight";

var windowBounds = new air.Rectangle(200,250,300,400);
newHTMLLoader = air.HTMLLoader.createRootWindow(true, options, true, windowBounds);
newHTMLLoader.load(new air.URLRequest("xmpl.html"));
```

注：如果新窗口加载的内容位于应用程序安全沙箱外部，则 window 对象没有以下 AIR 属性：runtime、nativeWindow 或 htmlLoader。

使用 createRootWindow() 方法创建的窗口与打开窗口相互独立。JavaScript Window 对象的 parent 和 opener 属性为 null。打开窗口可以使用 createRootWindow() 函数返回的 HTMLLoader 引用来访问新窗口的 Window 对象。在前一个示例的上下文中，语句 newHTMLLoader.window 引用所创建窗口的 JavaScript Window 对象。

注：从 JavaScript 和 ActionScript 中均可调用 createRootWindow() 函数。

向窗口添加内容

向 AIR 窗口添加内容的方式取决于窗口类型。使用 HTML，您能够以声明方式在文本文件中定义窗口的基本内容。可以从单独的应用程序文件中加载各种资源。HTML 和 Flash 内容可以动态创建和动态添加到窗口。

在加载包含 JavaScript 的 SWF 内容或 HTML 内容时，必须考虑 AIR 安全模型。应用程序安全沙箱中的任何内容（即与应用程序一起安装且可用 app: URL 方案加载的内容）具有对所有 AIR API 的完全访问权限。从此沙箱外部加载的任何内容均无法访问 AIR API。应用程序沙箱外部的 JavaScript 内容无法使用 JavaScript Window 对象的 runtime、nativeWindow 或 htmlLoader 属性。

为允许安全的跨脚本访问，可以使用沙箱桥在应用程序内容和非应用程序内容之间提供有限的接口。在 HTML 内容中，还可以将应用程序的页面映射到非应用程序沙箱中以允许该页面上的代码跨脚本访问外部内容。请参阅 第 89 页的“[AIR 安全性](#)”。

加载 SWF 文件或图像

可以使用 flash.display.Loader 类将 Flash SWF 文件或图像加载到本机窗口的显示列表中：

```
package {
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.display.Loader;

    public class LoadedSWF extends Sprite
    {
        public function LoadedSWF(){
            var loader:Loader = new Loader();
            loader.load(new URLRequest("visual.swf"));
            loader.contentLoaderInfo.addEventListener(Event.COMPLETE,loadFlash);
        }

        private function loadFlash(event:Event):void{
            addChild(event.target.loader);
        }
    }
}
```

将 HTML 内容加载到 NativeWindow 中

若要将 HTML 内容加载到 NativeWindow 中，可以将 HTMLLoader 对象添加到窗口舞台并将 HTML 内容加载到 HTMLLoader 中，也可以使用 HTMLLoader.createRootWindow() 方法创建已包含 HTMLLoader 对象的窗口。以下示例在本机窗口舞台上的 300 x 500 像素的显示区域内显示 HTML 内容：

```
//newWindow is a NativeWindow instance
var htmlView:HTMLLoader = new HTMLLoader();
htmlView.width = 300;
htmlView.height = 500;

//set the stage so display objects are added to the top-left and not scaled
newWindow.stage.align = "TL";
newWindow.stage.scaleMode = "noScale";
newWindow.stage.addChild( htmlView );

// urlString is the URL of the HTML page to load
htmlView.load( new URLRequest(urlString) );
```

注：如果窗口使用透明度（即窗口的 transparent 属性为 true）或者缩放了 HTMLLoader 控件，则不会显示 HTML 文件中的 SWF 内容或 PDF 内容。

将 SWF 内容添加为 HTML 窗口上的覆盖图

由于 HTML 窗口包含在 NativeWindow 实例中，因此可以将 Flash 显示对象添加到显示列表中 HTML 层的上方或下方。

若要将显示对象添加到 HTML 层的上方，请使用 `window.nativeWindow.stage` 属性的 `addChild()` 方法。`addChild()` 方法将分层的内容添加到窗口中任何现有内容的上方。

若要将显示对象添加到 HTML 层的下方，请使用 `window.nativeWindow.stage` 属性的 `addChildAt()` 方法，同时为 `index` 参数传入值零。将对象放在零索引处会将现有内容（包括 HTML 显示）上移一层并在底部插入新内容。为使在 HTML 页下方分层的内容可见，必须将 `HTMLLoader` 对象的 `paintsDefaultBackground` 属性设置为 `false`。此外，该页中设置背景颜色的任何元素都将不是透明的。例如，如果设置页面 `body` 元素的背景颜色，则该页的所有内容都将不是透明的。

以下示例说明如何将 Flash 显示对象作为覆盖图或衬垫层添加到 HTML 页。该示例创建两个简单的 `shape` 对象，在 HTML 内容下方和上方各添加一个 `shape` 对象。该示例还基于 `enterFrame` 事件更新形状位置。

```
<html>
<head>
<title>Bouncers</title>
<script src="AIRAliases.js" type="text/javascript"></script>
<script language="JavaScript" type="text/javascript">
air.Shape = window.runtime.flash.display.Shape;

function Bouncer(radius, color){
    this.radius = radius;
    this.color = color;

    //velocity
    this.vX = -1.3;
    this.vY = -1;

    //Create a Shape object and draw a circle with its graphics property
    this.shape = new air.Shape();
    this.shape.graphics.lineStyle(1,0);
    this.shape.graphics.beginFill(this.color,.9);
    this.shape.graphics.drawCircle(0,0,this.radius);
    this.shape.graphics.endFill();

    //Set the starting position
    this.shape.x = 100;
    this.shape.y = 100;

    //Moves the sprite by adding (vX,vY) to the current position
    this.update = function(){
        this.shape.x += this.vX;
        this.shape.y += this.vY;

        //Keep the sprite within the window
        if( this.shape.x - this.radius < 0){
            this.vX = -this.vX;
        }
        if( this.shape.y - this.radius < 0){
            this.vY = -this.vY;
        }
        if( this.shape.x + this.radius > window.nativeWindow.stage.stageWidth){
            this.vX = -this.vX;
        }
        if( this.shape.y + this.radius > window.nativeWindow.stage.stageHeight){
            this.vY = -this.vY;
        }
    };
}

function init(){
    //turn off the default HTML background
```

```
window.htmlLoader.paintsDefaultBackground = false;
var bottom = new Bouncer(60,0xff2233);
var top = new Bouncer(30,0x2441ff);

//listen for the enterFrame event
window.htmlLoader.addEventListener("enterFrame",function(evt){
    bottom.update();
    top.update();
});

//add the bouncing shapes to the window stage
window.nativeWindow.stage.addChildAt(bottom.shape,0);
window.nativeWindow.stage.addChild(top.shape);
}

</script>
<body onload="init()">
<h1>de Finibus Bonorum et Malorum</h1>
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo.</p>
<p style="background-color:#FFFF00; color:#660000;">This paragraph has a background color.</p>
<p>At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similiqne sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga.</p>
</body>
</html>
```

注 若要访问 JavaScript Window 对象的运行时 nativeWindow 和 htmlLoader 属性，则必须从应用程序目录中加载 HTML 页。这种情况始终适用于基于 HTML 的应用程序中的根页面，但可能不适用于其它内容。此外，加载到框架（即使在应用程序沙箱中）中的文档不接收这些属性，但可以访问父级文档的那些属性。

示例：创建本机窗口

以下示例说明如何创建本机窗口：

```
function createNativeWindow() {
    //create the init options
    var options = new air.NativeWindowInitOptions();
    options.transparent = false;
    options.systemChrome = air.NativeWindowSystemChrome.STANDARD;
    options.type = air.NativeWindowType.NORMAL;

    //create the window
    var newWindow = new air.NativeWindow(options);
    newWindow.title = "A title";
    newWindow.width = 600;
    newWindow.height = 400;

    //activate and show the new window
    newWindow.activate();
}
```

管理窗口

使用 NativeWindow 类的属性和方法可管理桌面窗口的外观、行为和生命周期。

获取 NativeWindow 实例

若要操作窗口，必须首先获取窗口实例。可以从以下任一位置获取窗口实例：

- 用于创建窗口的本机窗口构造函数：

```
var nativeWin = new air.NativeWindow(initOptions);
```

- 窗口中显示对象的 `stage` 属性：

```
var nativeWin = window.htmlLoader.stage.nativeWindow;
```

- 由窗口调度的本机窗口事件的 `target` 属性：

```
function onNativeWindowEvent(event)
{
    var nativeWin = event.target;
}
```

- 窗口中显示的 HTML 页的 `nativeWindow` 属性：

```
var nativeWin = window.nativeWindow;
```

- `NativeApplication` 对象的 `activeWindow` 和 `openedWindows` 属性：

```
var win = NativeApplication.nativeApplication.activeWindow;
var firstWindow = NativeApplication.nativeApplication.openedWindows[0];
```

`NativeApplication.nativeApplication.activeWindow` 引用应用程序的活动窗口（但如果该活动窗口不是此 AIR 应用程序的窗口，则返回 `null`）。`NativeApplication.nativeApplication.openedWindows` 数组包含 AIR 应用程序中尚未关闭的所有窗口。

激活、显示和隐藏窗口

若要激活窗口，请调用 `NativeWindow activate()` 方法。激活窗口会将该窗口置于前面，为其提供键盘和鼠标焦点，并在必要时通过还原窗口或将 `visible` 属性设置为 `true` 来使其可见。激活窗口不会更改应用程序中其它窗口的顺序。调用 `activate()` 方法会导致窗口调度 `activate` 事件。

若要在不激活的情况下显示隐藏窗口，请将 `visible` 属性设置为 `true`。此操作使该窗口置于前面，但不会向它分配焦点。

若要从视图中隐藏窗口，请将其 `visible` 属性设置为 `false`。隐藏窗口会禁止窗口和任何相关任务栏图标的显示，并且在 Mac OS X 中还会禁止窗口菜单中条目的显示。

注：在 Mac OS X 中，无法完全隐藏在停靠栏的窗口部分拥有图标的最小化窗口。如果最小化窗口上的 `visible` 属性设置为 `false`，则仍然显示该窗口的停靠栏图标。如果用户单击该图标，则会将该窗口还原为可见状态并显示该窗口。

更改窗口显示顺序

AIR 提供几种方法来直接更改窗口的显示顺序。可以将窗口的显示顺序向前或向后移动；可以将窗口移动到其它窗口的前面或后面。同时，用户可以通过激活窗口来对窗口进行重新排序。

可以通过将窗口的 `alwaysInFront` 属性设置为 `true` 来使该窗口位于其它窗口的前面。如果多个窗口都具有此设置，则这些窗口的显示顺序是它们相互之间的排序顺序，而且它们始终排序在 `alwaysInFront` 设置为 `false` 的窗口前面。即使 AIR 应用程序未处于活动状态，最上面组中的窗口也显示在其它应用程序中窗口的前面。由于此行为会阻碍用户查看其它窗口，因此应仅在必要和适当时才能将 `alwaysInFront` 设置为 `true`。经调整的使用示例包括：

- 工具提示、弹出列表、自定义菜单或组合框等控件的临时弹出窗口。由于这些窗口在失去焦点后将关闭，因此可以避免阻碍用户查看其它窗口的不便。
- 极其紧急的错误消息和警告。在可能发生不可撤销的更改时，如果用户未及时响应，则可能调整为将警告窗口置于最前面。但是，可以按正常的窗口显示顺序处理大多数错误消息和警告。
- 短期弹出式窗口。

注: AIR 不强制要求 `alwaysInFront` 属性的正确使用。但是, 如果应用程序打断了用户的工作流, 则可能将其传递到同一用户的垃圾桶。

`NativeWindow` 类提供以下属性和方法来设置一个窗口相对于其它窗口的显示顺序:

成员	说明
<code>alwaysInFront</code> 属性	指定窗口是否显示在最上面的窗口组中。 几乎在所有情况下, <code>false</code> 都是最佳设置。将值从 <code>false</code> 更改为 <code>true</code> 会将窗口置于所有其它窗口的前面 (但不会激活该窗口)。将值从 <code>true</code> 更改为 <code>false</code> 会将窗口的顺序排在最上面组中其余窗口的后面, 但仍位于其它窗口的前面。将窗口的该属性设置为 <code>其当前值</code> 不会更改窗口显示顺序。
<code>orderToFront()</code>	将窗口置于前面。
<code>orderInFrontOf()</code>	将窗口置于紧靠特定窗口前面。
<code>orderToBack()</code>	将窗口发送到其它窗口后面。
<code>orderBehind()</code>	将窗口发送到紧靠特定窗口后面。
<code>activate()</code>	将窗口置于前面 (同时使该窗口可见并分配焦点)。

注: 如果窗口处于隐藏 (`visible` 为 `false`) 或最小化状态, 则调用显示顺序方法无效。

在 Linux 操作系统中, 不同的窗口管理器对于窗口显示顺序实施不同的规则:

- 在某些窗口管理器中, 实用程序窗口始终显示于普通窗口之前。
- 在某些窗口管理器中, 将 `alwaysInFront` 设置为 `true` 的全屏窗口始终显示于其他同样将 `alwaysInFront` 设置为 `true` 的窗口之前。

关闭窗口

若要关闭窗口, 请使用 `NativeWindow.close()` 方法。

关闭窗口会卸载该窗口的内容, 但如果其它对象引用了此内容, 则不会破坏内容对象。`NativeWindow.close()` 方法以异步方式执行, 该窗口中包含的应用程序在关闭过程中继续运行。该 `close` 方法在关闭操作完成时调度 `close` 事件。从技术角度而言, `NativeWindow` 对象仍然有效, 但访问已关闭窗口上的大多数属性和方法会生成 `IllegalOperationError`。不能重新打开已关闭窗口。检查窗口的 `closed` 属性以测试窗口是否已关闭。若要仅从视图中隐藏窗口, 请将 `NativeWindow.visible` 属性设置为 `false`。

如果 `Nativeapplication.autoExit` 属性为 `true` (默认情况), 则应用程序在其最后一个窗口关闭后退出。

允许取消窗口操作

在窗口使用系统镶边时, 可以通过侦听和取消相应事件的默认行为来取消用户与该窗口的交互。例如, 在用户单击系统镶边关闭按钮后, 将调度 `closing` 事件。如果任何已注册的侦听器调用了事件的 `preventDefault()` 方法, 则窗口不会关闭。

在窗口不使用系统镶边时, 不会在执行预期更改之前自动调度这些更改的通知事件。因此, 如果调用关闭窗口、更改窗口状态的方法, 或者设置任何窗口范围属性, 则无法取消更改。若要在做出窗口更改前通知应用程序中的组件, 应用程序逻辑可以使用窗口的 `dispatchEvent()` 方法调度相关的通知事件。

```
function onCloseCommand(event){
    var closingEvent = new air.Event(air.Event.CLOSING,true,true);
    dispatchEvent(closingEvent);
    if(!closingEvent.isDefaultPrevented()){
        win.close();
    }
}
```

如果侦听器调用事件的 `preventDefault()` 方法，则 `dispatchEvent()` 方法返回 `false`。但是，它也可能由于其它原因而返回 `false`，因此最好显式使用 `isDefaultPrevented()` 方法来测试是否应取消更改。

最大化、最小化和还原窗口

若要最大化窗口，请使用 `NativeWindow maximize()` 方法。

```
window.nativeWindow.maximize();
```

若要最小化窗口，请使用 `NativeWindow minimize()` 方法。

```
window.nativeWindow.minimize();
```

若要还原窗口（即，使其返回最小化或最大化操作之前的大小），请使用 `NativeWindow restore()` 方法。

```
window.nativeWindow.restore();
```

注：最大化 AIR 窗口导致的行为与 Mac OS X 标准行为不同。AIR 窗口不是在应用程序定义的“标准”大小和用户最后设置的大小之间切换，而是在应用程序或用户最后设置的大小和屏幕的完整可用区域之间切换。

在 Linux 操作系统中，不同的窗口管理器对于设置窗口显示状态实施不同的规则：

- 在某些窗口管理器中无法将实用程序窗口最大化。
- 如果为窗口设置了最大大小，则某些窗口不允许将窗口最大化。某些其他窗口管理器将显示状态设置为最大化，但不调整窗口大小。在这两种情况下，都不会调度显示状态更改事件。
- 某些窗口管理器不遵守窗口的 `maximizable` 或 `minimizable` 设置。

注：在 Linux 中，更改窗口属性是异步进行的。如果在程序的一行中更改窗口的显示状态，并在下一行读取该值，则对值的读取仍将受旧的设置影响。在所有平台上，当显示状态发生更改时，`NativeWindow` 对象将调用 `displayStateChange` 事件。如果您需要根据窗口的新状态执行某种动作，始终在 `displayStateChange` 事件处理函数中执行。请参阅第 131 页的“[侦听窗口事件](#)”。

示例：最小化、最大化、还原和关闭窗口

以下简短的 HTML 页演示 `NativeWindow maximize()`、`minimize()`、`restore()` 和 `close()` 方法：

```
<html>
<head>
<title>Change Window Display State</title>
<script src="AIRAliases.js"/>
<script type="text/javascript">
    function onMaximize(){
        window.nativeWindow.maximize();
    }

    function onMinimize(){
        window.nativeWindow.minimize();
    }

    function onRestore(){
        window.nativeWindow.restore();
    }

    function onClose(){
        window.nativeWindow.close();
    }
</script>
</head>

<body>
    <h1>AIR window display state commands</h1>
    <button onClick="onMaximize()">Maximize</button>
    <button onClick="onMinimize()">Minimize</button>
    <button onClick="onRestore()">Restore</button>
    <button onClick="onClose()">Close</button>
</body>
</html>
```

调整窗口大小和移动窗口

在窗口使用系统镶边时，该镶边提供用于调整窗口大小和在桌面范围内移动窗口的拖动控件。如果窗口不使用系统镶边，则必须添加您自己的控件以允许用户调整窗口大小和移动窗口。

注：若要调整窗口大小或移动窗口，必须首先获取对 **NativeWindow** 实例的引用。有关如何获取窗口引用的信息，请参阅第 126 页的“[获取 NativeWindow 实例](#)”。

调整窗口大小

若要使用户可以交互地调整窗口大小，请使用 **NativeWindow startResize()** 方法。如果此方法是从 **mouseDown** 事件中调用的，则调整大小操作由鼠标驱动并在操作系统收到 **mouseUp** 事件时完成。在调用 **startResize()** 时，传入用于指定所调整窗口大小的起始边或角的参数。

若要以编程方式设置窗口大小，请将窗口的 **width**、**height** 或 **bounds** 属性设置为所需尺寸。设置范围时，可以同时更改窗口的大小和位置。但是，无法保证发生更改的顺序。某些 **Linux** 窗口管理器不允许窗口扩展到桌面屏幕范围之外。在这些情况下，即使更改的最终效果以其他方式产生了合法的窗口，最终窗口大小也可能因属性的设置顺序而受到限制。例如，如果同时更改靠近屏幕底部的窗口的高度和 Y 轴位置，那么在 Y 轴位置更改之前应用高度更改时，可能不会进行完整的高度更改。

注：在 **Linux** 中，更改窗口属性是异步进行的。如果在程序的一行中调整窗口大小，并在下一行读取尺寸，则这些尺寸仍将受旧的设置影响。在所有平台上，当调整窗口大小时，**NativeWindow** 对象将调用 **resize** 事件。如果您需要根据窗口的新大小或状态执行某种动作（如设置窗口中控件的布局），始终在 **resize** 事件处理函数中执行。请参阅第 131 页的“[侦听窗口事件](#)”。

移动窗口

若要移动窗口而不调整其大小，请使用 **NativeWindow startMove()** 方法。与 **startResize()** 方法相似，在从 **mouseDown** 事件调用 **startMove()** 方法时，移动过程由鼠标驱动，并在操作系统收到 **mouseUp** 事件时结束。

有关 `startResize()` 和 `startMove()` 方法的详细信息，请参阅针对 HTML 开发人员的 Adobe AIR 语言参考 (http://www.adobe.com/go/learn_air_html_jslr_cn)。

若要以编程方式移动窗口，请将窗口的 `x`、`y` 或 `bounds` 属性设置为所需的位置。设置范围时，可以同时更改窗口的大小和位置。

注：在 Linux 中，更改窗口属性是异步进行的。如果在程序的一行中移动窗口，并在下一行读取该位置，则对值的读取仍将受旧的设置影响。在所有平台上，当更改位置时，`NativeWindow` 对象将调用 `move` 事件。如果您需要根据窗口的新位置执行某种动作，始终在 `move` 事件处理函数中执行。请参阅第 131 页的“[侦听窗口事件](#)”。

示例：调整窗口大小和移动窗口

以下示例说明如何对窗口启动调整大小和移动操作：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<script src="AIRAliases.js"/>
<script type="text/javascript">
    function onResize(type) {
        nativeWindow.startResize(type);
    }

    function onNativeMove() {
        nativeWindow.startMove();
    }
</script>
<style type="text/css" media="screen">

.drag {
    width:200px;
    height:200px;
    margin:0px auto;
    padding:15px;
    border:1px dashed #333;
    background-color:#eee;
}

.resize {
    background-color:#FF0000;
    padding:10px;
}
.left {
    float:left;
}
.right {
    float:right;
}

</style>
<title>Move and Resize the Window</title>
</head>

<body>
<div class="resize left" onmousedown="onResize(air.NativeWindowResize.TOP_LEFT)">Drag to resize</div>
<div class="resize right" onmousedown="onResize(air.NativeWindowResize.TOP_RIGHT)">Drag to resize</div>
<div class="drag" onmousedown="onNativeMove()">Drag to move</div>
<div class="resize left" onmousedown="onResize(air.NativeWindowResize.BOTTOM_LEFT)">Drag to resize</div>
<div class="resize right" onmousedown="onResize(air.NativeWindowResize.BOTTOM_RIGHT)">Drag to resize</div>
</body>
</html>
```

侦听窗口事件

若要侦听窗口调度的事件，请向窗口实例注册侦听器。例如，若要侦听 **closing** 事件，请按如下方式向窗口实例注册侦听器：

```
window.nativeWindow.addEventListener(air.Event.CLOSING, onClosingEvent);
```

在调度事件时，**target** 属性引用发送该事件的窗口。

大多数窗口事件都有两条相关消息。第一条消息发出即将发生窗口更改（可以取消）的信号通知，第二条消息发出更改已发生的信号通知。例如，在用户单击窗口的关闭按钮后，将调度 **closing** 事件消息。如果侦听器未取消该事件，则窗口将关闭并且 **close** 事件将调度到所有侦听器。

通常，仅在已使用系统镶边触发事件时才调度 **closing** 等警告事件。例如，调用窗口的 **close()** 方法不会自动调度 **closing** 事件，而只调度 **close** 事件。但是，可以构造 **closing** 事件对象，并使用窗口的 **dispatchEvent()** 方法调度该事件。

调度 **Event** 对象的窗口事件包括：

事件	说明
activate	在窗口收到焦点时调度。
deactivate	在窗口失去焦点时调度
closing	在窗口即将关闭时调度。仅当在按下系统镶边关闭按钮时或者在 Mac OS X 中调用 Quit 命令时，此事件才自动发生。
close	在窗口关闭时调度。

调度 **NativeWindowBoundsEvent** 对象的窗口事件包括：

事件	说明
moving	在窗口左上角由于移动窗口、调整窗口大小或更改窗口显示状态而更改位置的前一刻调度。
move	在左上角更改位置之后调度。
resizing	在窗口宽度或高度由于调整大小或显示状态更改而发生更改的前一刻调度。
resize	在窗口更改大小之后调度。

对于 **NativeWindowBoundsEvent** 事件，可以使用 **beforeBounds** 和 **afterBounds** 属性确定即将进行更改或完成更改之前和之后的窗口范围。

调度 **NativeWindowDisplayStateEvent** 对象的窗口事件包括：

事件	说明
displayStateChanging	在窗口显示状态更改的前一刻调度。
displayStateChange	在窗口显示状态更改之后调度。

对于 **NativeWindowDisplayStateEvent** 事件，可以使用 **beforeDisplayState** 和 **afterDisplayState** 属性确定即将进行更改或完成更改之前和之后的窗口显示状态。

在某些 Linux 窗口管理器中，将具有最大化大小设置的窗口最大化时并不调度显示状态更改事件。（窗口设置为最大化显示状态，但并不调整其大小。）

显示全屏窗口

将 Stage 的 displayState 属性设置为 StageDisplayState.FULL_SCREEN_INTERACTIVE 会使窗口进入全屏模式，在此模式下允许键盘输入。（在浏览器中运行的 SWF 内容中，不允许键盘输入）。若要退出全屏模式，用户需要按 Esc 键。

注：如果为窗口设置了最大大小，某些 Linux 窗口管理器不会更改窗口尺寸以适应屏幕（但会删除窗口的系统镶边）。

以下 HTML 页模拟全屏文本端点：

```
<html>
<head>
<title>Fullscreen Mode</title>
<script language="JavaScript" type="text/javascript">
function setDisplayState() {
    window.nativeWindow.stage.displayState =
        runtime.flash.display.StageDisplayState.FULL_SCREEN_INTERACTIVE;
}
</script>
<style type="text/css">
body, .mono {
    font-family: Courier New, Courier, monospace;
    font-size: x-large;
    color:#CCFF00;
    background-color:#003030;
}
</style>
</head>
<body onload="setDisplayState();">
    <p class="mono">Welcome to the dumb terminal app. Press the ESC key to exit...</p>
    <textarea name="dumb" class="mono" cols="100" rows="40">%</textarea>
</body>
</html>
```

第 18 章：屏幕

使用 Adobe® AIR® Screen 类可访问连接到计算机的桌面显示屏幕的有关信息。

有关屏幕的其它在线信息

可以从以下源中查找有关 Screen 类和使用屏幕的详细信息：

语言参考

- [Screen](#)

Adobe 开发人员中心文章和范例

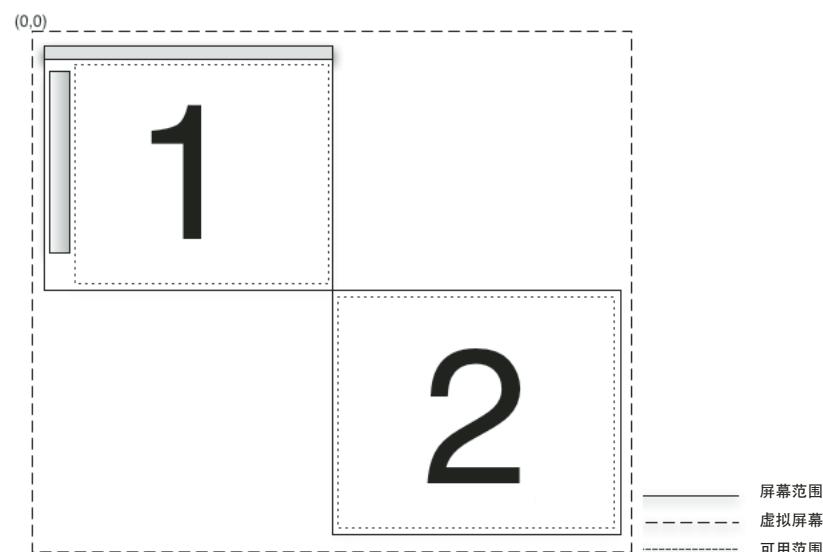
- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR 屏幕”）](#)

屏幕基础知识

屏幕 API 只包含单个 Screen 类，该类提供用于获取系统屏幕信息的静态成员以及用于描述特定屏幕的实例成员。

计算机系统可以连接多台监视器或显示器，这些监视器或显示器对应于虚拟空间中排列的多个桌面屏幕。AIR Screen 类提供有关这些屏幕、屏幕的相对排列及其可用空间的信息。如果多台监视器映射到同一屏幕，则只存在一个屏幕。如果屏幕的尺寸大于监视器的显示区域，则无法确定当前可以看到屏幕的哪个部分。

屏幕表示独立的桌面显示区域。屏幕被描述为虚拟桌面内部的矩形。被指定为主显示屏的屏幕的左上角是虚拟桌面坐标系的原点。用于描述屏幕的所有值均以像素为单位提供。



在此屏幕排列中，虚拟桌面中存在两个屏幕。主屏幕 (#1) 左上角的坐标始终为 (0,0)。如果屏幕排列更改为指定屏幕 #2 作为主屏幕，则屏幕 #1 的坐标将变为负值。报告屏幕的可用范围时将排除菜单栏、任务栏和停靠栏。

有关屏幕 API 类、方法、属性和事件的详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](http://www.adobe.com/go/learn_air_html_jslr_cn) (http://www.adobe.com/go/learn_air_html_jslr_cn)。

枚举屏幕

可以使用以下屏幕方法和属性枚举虚拟桌面的屏幕：

方法或属性	说明
Screen.screens	提供描述可用屏幕的 Screen 对象的数组。请注意，数组的顺序并不重要。
Screen.mainScreen	提供主屏幕的 Screen 对象。在 Mac OS X 中，主屏幕是指显示菜单栏的屏幕。在 Windows 中，主屏幕是指系统指定的主屏幕。
Screen.getScreensForRectangle()	提供描述与给定矩形相交的屏幕的 Screen 对象的数组。传递给此方法的矩形位于虚拟桌面上的像素坐标中。如果没有屏幕与该矩形相交，则该数组为空。可以使用此方法确定窗口显示在哪些屏幕上。

不应保存 Screen 类、方法和属性返回的值。用户或操作系统可随时更改可用屏幕及其排列方式。

以下示例使用屏幕 API 在多个屏幕间移动窗口，以响应箭头键的按键操作。该示例获取 screens 数组并将其在垂直或水平方向排序（取决于所按的箭头键），以便将窗口移动到下一个屏幕。代码随后将遍历排序后的数组，将每个屏幕与当前屏幕的坐标进行比较。为了识别窗口的当前屏幕，该示例调用 Screen.getScreensForRectangle()，并传入窗口范围。

```
<html>
<head>
<script src="AIRAliases.js" type="text/javascript"></script>
<script type="text/javascript">
function onKey(event) {
    if(air.Screen.screens.length > 1){
        switch(event.keyCode) {
            case air.Keyboard.LEFT :
                moveLeft();
                break;
            case air.Keyboard.RIGHT :
                moveRight();
                break;
            case air.Keyboard.UP :
                moveUp();
                break;
            case air.Keyboard.DOWN :
                moveDown();
                break;
        }
    }
}

function moveLeft(){
    var currentScreen = getCurrentScreen();
    var left = air.Screen.screens;
    left.sort(sortHorizontal);
    for(var i = 0; i < left.length - 1; i++){
        if(left[i].bounds.left < window.nativeWindow.bounds.left){
            window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
        }
    }
}
```

```
function moveRight(){
    var currentScreen = getCurrentScreen();
    var left = air.Screen.screens;
    left.sort(sortHorizontal);
    for(var i = left.length - 1; i > 0; i--){
        if(left[i].bounds.left > window.nativeWindow.bounds.left){
            window.nativeWindow.x += left[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += left[i].bounds.top - currentScreen.bounds.top;
        }
    }
}

function moveUp(){
    var currentScreen = getCurrentScreen();
    var top = air.Screen.screens;
    top.sort(sortVertical);
    for(var i = 0; i < top.length - 1; i++){
        if(top[i].bounds.top < window.nativeWindow.bounds.top){
            window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
            break;
        }
    }
}

function moveDown(){
    var currentScreen = getCurrentScreen();

    var top = air.Screen.screens;
    top.sort(sortVertical);
    for(var i = top.length - 1; i > 0; i--){
        if(top[i].bounds.top > window.nativeWindow.bounds.top){
            window.nativeWindow.x += top[i].bounds.left - currentScreen.bounds.left;
            window.nativeWindow.y += top[i].bounds.top - currentScreen.bounds.top;
            break;
        }
    }
}

function sortHorizontal(a,b){
    if (a.bounds.left > b.bounds.left){
        return 1;
    } else if (a.bounds.left < b.bounds.left){
        return -1;
    } else {return 0;}
}

function sortVertical(a,b){
    if (a.bounds.top > b.bounds.top){
        return 1;
    } else if (a.bounds.top < b.bounds.top){
        return -1;
    } else {return 0;}
}
```

```
}

function getCurrentScreen(){
    var current;
    var screens = air.Screen.getScreensForRectangle(window.nativeWindow.bounds);
    (screens.length > 0) ? current = screens[0] : current = air.Screen.mainScreen;
    return current;
}

function init(){
    window.nativeWindow.stage.addEventListener("keyDown",onKey);
}
</script>
<title>Screen Hopper</title>
</head>
<body onload="init()">
    <p>Use the arrow keys to move the window between monitors.</p>
</body>
</html>
```

第 19 章：使用本机菜单

使用本机菜单 API 中的类定义应用程序、窗口、上下文和弹出菜单。

有关本机菜单的其它在线信息

可以从以下源中查找有关本机菜单 API 和使用本机菜单的详细信息：

快速入门（**Adobe AIR** 开发人员中心）

- [向 AIR 应用程序中添加本机菜单](#)

语言参考

- [NativeMenu](#)
- [NativeMenuItem](#)

Adobe 开发人员中心文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR 菜单”）](#)

AIR 菜单基础知识

通过本机菜单类，可以访问运行您的应用程序的操作系统的本机菜单功能。NativeMenu 对象可用于应用程序菜单（Mac OS X 中提供）、窗口菜单（Windows 和 Linux 中提供）、上下文菜单和弹出菜单。

AIR 菜单类

Adobe® AIR® 菜单类包括：

包	类
flash.display	<ul style="list-style-type: none"> • NativeMenu • NativeMenuItem
flash.ui	<ul style="list-style-type: none"> • ContextMenu • ContextMenuItem
flash.events	<ul style="list-style-type: none"> • Event

菜单类型

AIR 支持以下类型的菜单：

应用程序菜单 应用程序菜单是应用于整个应用程序的全局菜单。Mac OS X 支持应用程序菜单，但 Windows 或 Linux 不支持。在 Mac OS X 上，操作系统自动创建应用程序菜单。可以使用 AIR 菜单 API 将项目和子菜单添加到标准菜单中。可以添加用于处理现有菜单命令的侦听器。还可以删除现有项目。

窗口菜单 窗口菜单与单个窗口关联，并显示在标题栏的下方。通过创建 NativeMenu 对象并将它分配给 NativeWindow 对象的 menu 属性，可以向窗口添加菜单。**Windows** 和 **Linux** 操作系统支持窗口菜单，但 **Mac OS X** 不支持。本机窗口菜单只能与有系统镶边的窗口一起使用。

上下文菜单 上下文菜单在响应对 SWF 内容中的交互式对象或 HTML 内容中的文档元素的右键单击或命令单击时打开。可以使用 NativeMenu 或 ContextMenu 类创建上下文菜单。在 HTML 内容中，可以使用 Webkit HTML 和 JavaScript API 向 HTML 元素添加上下文菜单。

停靠栏图标菜单和系统任务栏图标菜单 这些图标菜单类似于上下文菜单，分配给 Mac OS X 停靠栏或 Windows 和 Linux 任务栏上通知区域中的应用程序图标。停靠栏图标菜单和系统任务栏图标菜单使用 NativeMenu 类。在 Mac OS X 上，菜单中的项目添加到标准操作系统的上方。Windows 或 Linux 中没有标准菜单。

弹出菜单 AIR 弹出菜单类似于上下文菜单，但不一定与特定应用程序对象或组件关联。通过调用任意 NativeMenu 对象的 display() 方法，可以在窗口中的任意位置显示弹出菜单。

自定义菜单 本机菜单完全由操作系统绘制，因此存在于 Flash 和 HTML 呈现模型以外。可以自由使用 MXML、ActionScript 或 JavaScript 创建自己的非本机菜单。AIR 菜单类没有为控制本机菜单的绘制提供任何便利。

默认菜单

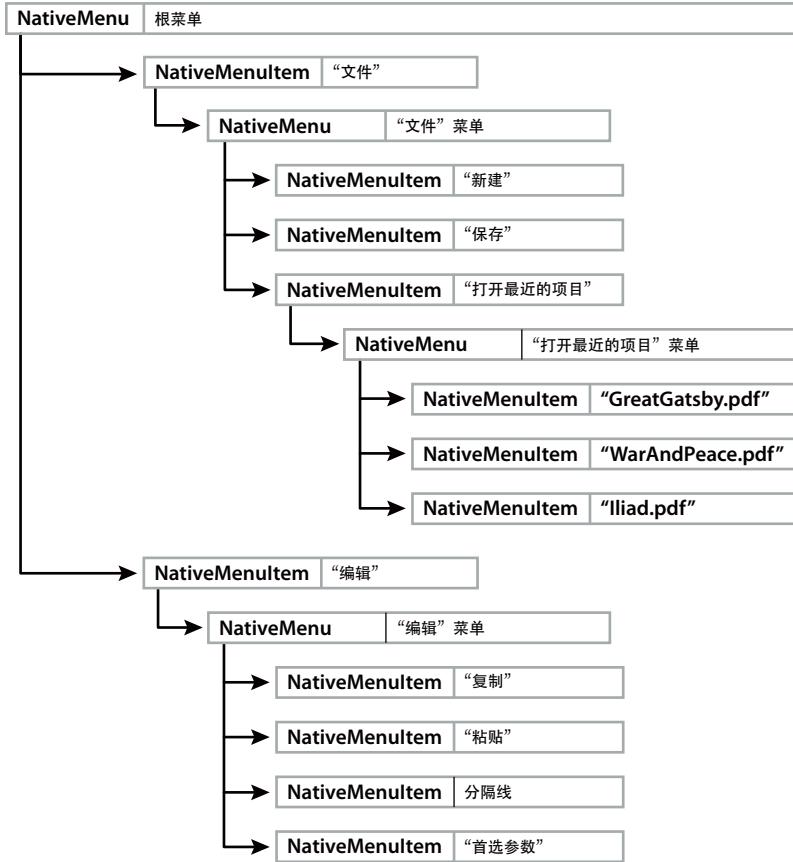
以下默认菜单由操作系统或内置 AIR 类提供：

- Mac OS X 上的应用程序菜单
- Mac OS X 上的停靠栏图标菜单
- HTML 内容中的所选文本和图像的上下文菜单
- TextField 对象（或扩展 TextField 的对象）中的所选文本的上下文菜单

菜单结构

菜单实质上是有层次结构的。NativeMenu 对象包含子级 NativeMenuItem 对象。表示子菜单的 NativeMenuItem 对象又可以包含 NativeMenu 对象。结构中的顶级或根级的菜单对象表示应用程序菜单和窗口菜单的菜单栏。（上下文、图标和弹出菜单没有菜单栏）。

下图说明了一个典型菜单的结构。根菜单表示菜单栏，它包含的两个菜单项引用“File”（文件）子菜单和“Edit”（编辑）子菜单。在此结构中，“File”（文件）子菜单包含两个命令项和一个引用项，该引用项引用“Open Recent”（打开最近的项目）子菜单，而该子菜单本身又包含三个项目。“Edit”（编辑）子菜单包含三个命令项和一个分隔符。



定义子菜单同时需要 NativeMenu 和 NativeMenuItem 对象。NativeMenuItem 对象定义在父菜单中显示的标签，并允许用户打开子菜单。NativeMenu 对象充当子菜单中的项目的容器。NativeMenuItem 对象通过 NativeMenuItem 的 submenu 属性引用 NativeMenu 对象。

若要查看创建此菜单的代码示例，请参阅第 145 页的“[示例：窗口和应用程序菜单](#)”。

菜单的事件

NativeMenu 和 NativeMenuItem 对象都将调度 displaying 和 select 事件：

Displaying: 在显示菜单的前一刻，菜单及其菜单项将 displaying 事件调度到任何注册的侦听器。displaying 事件为您提供一个在将菜单内容或项目外观显示给用户之前将其更新的机会。例如，在“Open Recent”（打开最近的项目）菜单的 displaying 事件的侦听器中，可以更改菜单项以反映最近查看过的文档的当前列表。

此事件对象的 target 属性始终是将要显示的菜单。currentTarget 是侦听器所注册的对象：或者是菜单本身，或者是其中的某一个项目。

注：每当访问菜单或其某个项目的状态时，也会调度 displaying 事件。

Select: 当用户选择命令项时，项目会将 select 事件调度到任何注册的侦听器。不能选择子菜单和分隔符项，因此永远不会调度 select 事件。

`select` 事件从菜单项上升到它的包含菜单，然后上升到根菜单。可以直接在项目上侦听 `select` 事件，也可以在菜单结构的更高位置侦听。在菜单上侦听 `select` 事件时，可以使用该事件的 `target` 属性识别所选项。当事件沿菜单层次结构上升时，该事件对象的 `currentTarget` 属性可识别当前菜单对象。

注：`ContextMenu` 和 `MenuItem` 对象调度 `menuItemSelected` 和 `menuSelect` 事件，以及 `select` 和 `displaying` 事件。

菜单命令的等效键

可以为菜单命令分配等效键（有时称为快捷键）。当按下键或组合键时，菜单项会将 `select` 事件调度到任何注册的侦听器。包含该项目的菜单必须是应用程序菜单的一部分，或者是要调用的命令的活动窗口菜单的一部分。

等效键有两部分，一部分是表示主键的字符串，另一部分是一组必须同时按下的功能键。若要分配主键，请将菜单项 `keyEquivalent` 属性设置为该键的单字符字符串。如果使用大写字母，则 `Shift` 键会自动添加到功能键组中。

在 Mac OS X 上，默认功能键是 `Command` 键 (`Keyboard.COMMAND`)。在 Windows 和 Linux 中，它是 `Ctrl` 键 (`Keyboard.CONTROL`)。这些默认键会自动添加到功能键组中。若要分配其它功能键，请将包含所需键代码的新键组分配给 `keyEquivalentModifiers` 属性。默认键组将被覆盖。无论使用默认功能键还是分配自己的功能键组，如果分配给 `keyEquivalent` 属性的字符串是大写字母，则会添加 `Shift` 键。用于功能键的键代码的常量在 `Keyboard` 类中定义。

所分配的等效键字符串将自动显示在菜单项名称的旁边。格式取决于用户的操作系统和系统首选项。

注：在 Windows 操作系统上，如果将 `Keyboard.COMMAND` 值分配给功能键组，则菜单中不显示等效键。但是，必须使用 `Ctrl` 键才能激活菜单命令。

以下示例分配 `Ctrl+Shift+G` 作为菜单项的等效键：

```
var item = new air.NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
```

此示例通过直接设置功能键组将 `Ctrl+Shift+G` 分配为等效键：

```
var item = new air.NativeMenuItem("Ungroup");
item.keyEquivalent = "G";
item.keyEquivalentModifiers = [air.Keyboard.CONTROL];
```

注：等效键仅为应用程序菜单和窗口菜单触发。如果将等效键添加到上下文或弹出菜单，则等效键将显示在菜单标签中，但永远不会调用关联的菜单命令。

助记键

助记键是菜单的操作系统键盘接口的一部分。Linux、Mac OS X 和 Windows 都允许用户通过键盘打开菜单并选择命令，但有一些细微的区别。

在 Mac OS X 中，用户键入菜单或命令的第一个或前两个字母，然后按 `Return` 键。`mnemonicIndex` 属性将被忽略。

在 Windows 上，仅一个字母是有效的。默认情况下，有效字母是标签中的第一个字符，但是，如果将助记键分配给菜单项，则有效字符将成为所指定的字母。如果一个菜单中的两个项有相同的有效字符（无论是否分配了助记键），则用户的键盘与菜单的交互稍有改变。用户必须将该字母按下所需的次数来突出显示所需项，然后按 `Enter` 完成选择，而不是仅按单个字母就能选择菜单或命令。为了保持一致的行为，应向窗口菜单中的每个菜单项都分配一个唯一的助记键。

在 Linux 中不提供默认的助记键。必须指定菜单项的 `mnemonicIndex` 属性的值才能提供助记键。

指定助记键字符作为标签字符串中的索引。标签中第一个字符的索引是 0。因此，若要使用“r”作为带“Format”标签的菜单项的助记键，可以将 `mnemonicIndex` 属性设置为等于 2。

```
var item = new air.NativeMenuItem("Format");
item.mnemonicIndex = 2;
```

菜单项状态

菜单项有两个状态属性: `checked` 和 `enabled`。

checked 设置为 `true` 将在项目标签旁边显示选中标记。

```
var item = new air.NativeMenuItem("Format");
item.checked = true;
```

enabled 在 `true` 和 `false` 之间切换值可以控制是否启用命令。禁用的项目将在视觉上“灰显”，并且不调度 `select` 事件。

```
var item = new air.NativeMenuItem("Format");
item.enabled = false;
```

将对象附加到菜单项

使用 `NativeMenuItem` 类的 `data` 属性可以引用每个项目中的任意对象。例如，在“Open Recent”（打开最近的项目）菜单中，可以将每个文档的 `File` 对象分配给每个菜单项。

```
var file = air.File.applicationStorageDirectory.resolvePath("GreatGatsby.pdf")
var menuItem = docMenu.addItem(new air.NativeMenuItem(file.name));
menuItem.data = file;
```

创建本机菜单

本主题描述如何创建 AIR 所支持的各种类型的本机菜单。

创建根菜单对象

若要创建 `NativeMenu` 对象来充当菜单的根，请使用 `NativeMenu` 构造函数：

```
var root = new air.NativeMenu();
```

对于应用程序菜单和窗口菜单，根菜单表示菜单栏，并且应当只包含打开子菜单的项目。上下文菜单和弹出菜单没有菜单栏，因此根菜单可以包含命令和分隔线以及子菜单。

在创建菜单之后，可以添加菜单项。除非使用菜单对象的 `addItemAt()` 方法在特定索引处添加项目，否则项目以其添加顺序显示在菜单中。

将菜单分配为应用程序、窗口、或图标菜单，或将其显示为弹出菜单，如以下几节所示：

设置应用程序菜单

```
air.NativeApplication.nativeApplication.menu = root;
```

注：Mac OS X 定义了一个菜单，其中包含可用于每个应用程序的标准项目。将新 `NativeMenu` 对象分配给 `NativeApplication` 对象的 `menu` 属性可以替换标准菜单。还可以使用标准菜单，而不是替换它。

设置窗口菜单

```
window.nativeWindow.menu = root;
```

设置停靠栏图标菜单

```
air.NativeApplication.nativeApplication.icon.menu = root;
```

注：Mac OS X 为应用程序停靠栏图标定义了标准菜单。在将新 `NativeMenu` 分配给 `DockIcon` 对象的 `menu` 属性时，该菜单中的项目将显示在标准项目之上。不能删除、访问或修改标准菜单项。

设置系统任务栏图标菜单

```
air.NativeApplication.nativeApplication.icon.menu = root;
```

以弹出方式显示菜单

```
root.display(window.nativeWindow.stage, x, y);
```

创建子菜单

若要创建子菜单，请将 `NativeMenuItem` 对象添加到父菜单，然后将定义子菜单的 `NativeMenu` 对象分配给该项目的 `submenu` 属性。AIR 提供了两种方式来创建子菜单项及其关联的菜单对象：

使用 `addSubmenu()` 方法，一步即可创建菜单项及其相关的菜单对象：

```
var editMenuItem = root.addSubmenu(new air.NativeMenu(), "Edit");
```

也可以创建菜单项，然后单独将菜单对象分配给其 `submenu` 属性：

```
var editMenuItem = root.addItem("Edit", false);
editMenuItem.submenu = new air.NativeMenu();
```

创建菜单命令

若要创建菜单命令，请将 `NativeMenuItem` 对象添加到菜单，然后添加一个事件侦听器来引用实现菜单命令的函数：

```
var copy = new air.NativeMenuItem("Copy", false);
copy.addEventListener(air.Event.SELECT, onCopyCommand);
editMenu.addItem(copy);
```

可以在命令项本身上侦听 `select` 事件（如本例中所示），也可以在父菜单对象上侦听 `select` 事件。

注：表示子菜单和分隔线的菜单项不调度 `select` 事件，因此不能用作命令。

创建菜单分隔线

若要创建分隔线，请创建 `NativeMenuItem`，并在构造函数中将 `isSeparator` 参数设置为 `true`。然后，将分隔符项目添加到菜单中的正确位置：

```
var separatorA = new air.NativeMenuItem("A", true);
editMenu.addItem(separatorA);
```

不显示为分隔符指定的标签（如果有）。

关于 HTML 中的上下文菜单

在 HTML 内容中，`contextmenu` 事件可用于显示上下文菜单。默认情况下，当用户调用所选文本上的上下文菜单事件时（通过右键单击或命令单击文本），将自动显示上下文菜单。若要防止打开默认菜单，请侦听 `contextmenu` 事件并调用该事件对象的 `preventDefault()` 方法：

```
function showContextMenu(event) {
    event.preventDefault();
}
```

然后可以使用 DHTML 技术或通过显示 AIR 本机上下文菜单来显示自定义上下文菜单。以下示例调用菜单的 `display()` 方法响应 HTML `contextmenu` 事件，从而显示本机上下文菜单：

```
<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="javascript" type="text/javascript">

function showContextMenu(event) {
    event.preventDefault();
    contextMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}

function createContextMenu() {
    var menu = new air.NativeMenu();
    var command = menu.addItem(new air.NativeMenuItem("Custom command"));
    command.addEventListener(air.Event.SELECT, onCommand);
    return menu;
}

function onCommand() {
    air.trace("Context command invoked.");
}

var contextMenu = createContextMenu();
</script>
</head>
<body>
<p oncontextmenu="showContextMenu(event)" style="-khtml-user-select:auto;">Custom context menu.</p>
</body>
</html>
```

显示弹出菜单

通过调用菜单的 `display()` 方法，可以随时随地在窗口上方显示任何 `NativeMenu` 对象。此方法需要对舞台的引用；因此，只有应用程序沙箱中的内容可以将菜单显示为弹出菜单。

以下方法显示由名为 `popupMenu` 的 `NativeMenu` 对象定义的菜单来响应鼠标单击：

```
function onMouseClick(event) {
    popupMenu.display(window.nativeWindow.stage, event.clientX, event.clientY);
}
```

注：不需要显示此菜单直接响应事件。任何方法都可以调用 `display()` 函数。

处理菜单事件

当用户选择菜单时，或用户选择菜单项时，菜单将调度事件。

菜单类的事件摘要

将事件侦听器添加到菜单或个别项目来处理菜单事件。

对象	调度的事件
NativeMenu	NativeMenuEvent.DISPLAYING NativeMenuEvent.SELECT (从子项目和子菜单传播)
NativeMenuItem	NativeMenuEvent.SELECT NativeMenuEvent.DISPLAYING (从父菜单传播)

选择菜单事件

若要处理菜单项上的单击，请将 `select` 事件的事件侦听器添加到 `NativeMenuItem` 对象：

```
var menuCommandX = new NativeMenuItem("Command X");
menuCommand.addEventListerner(air.Event.SELECT, doCommandX)
```

因为 `select` 事件会上升到包含菜单，所以还可以在父菜单上侦听 `select` 事件。在菜单级别上侦听时，可以使用事件对象的 `target` 属性来确定选择了哪个菜单命令。以下示例跟踪所选命令的标签：

```
var colorMenuItem = new air.NativeMenuItem("Choose a color");
var colorMenu = new air.NativeMenu();
colorMenuItem.submenu = colorMenu;

var red = new air.NativeMenuItem("Red");
var green = new air.NativeMenuItem("Green");
var blue = new air.NativeMenuItem("Blue");
colorMenu.addItem(red);
colorMenu.addItem(green);
colorMenu.addItem(blue);

if(air.NativeApplication.supportsMenu) {
    air.NativeApplication.nativeApplication.menu.addItem(colorMenuItem);
    air.NativeApplication.nativeApplication.menu.addEventListerner(air.Event.SELECT,
                                                                colorChoice);
} else if (air.NativeWindow.supportsMenu) {
    var windowMenu = new air.NativeMenu();
    window.nativeWindow.menu = windowMenu;
    windowMenu.addItem(colorMenuItem);
    windowMenu.addEventListerner(air.Event.SELECT, colorChoice);
}

function colorChoice(event) {
    var menuItem = event.target;
    air.trace(menuItem.label + " has been selected");
}
```

如果正在使用 `ContextMenu` 类，则可以侦听 `select` 事件或 `menuItemSelect` 事件。`menuItemSelect` 事件可以提供有关拥有此上下文菜单的对象的其它信息，但不能上升到包含菜单。

显示菜单事件

若要处理菜单的打开，可以为 `displaying` 事件添加一个侦听器，在显示菜单之前将调度该侦听器。可以使用 `displaying` 事件更新菜单，例如，通过添加或删除项目，或通过更新个别项目的启用或选中状态。

示例：窗口和应用程序菜单

以下示例创建在第 138 页的“[菜单结构](#)”中显示的菜单。

此菜单在设计上可同时用于 Windows（仅支持窗口菜单）和 Mac OS X（仅支持应用程序菜单）。为进行区分，`MenuExample` 类构造函数将检查 `NativeWindow` 和 `NativeApplication` 类的静态 `supportsMenu` 属性。如果 `NativeWindow.supportsMenu` 为 true，则该构造函数将为窗口创建 `NativeMenu` 对象，然后创建和添加“File”（文件）和“Edit”（编辑）子菜单。如果 `NativeApplication.supportsMenu` 为 true，则该构造函数将创建“File”（文件）和“Edit”（编辑）菜单，并将它们添加到 Mac OS X 操作系统所提供的现有菜单中。

本示例还将说明菜单事件的处理过程。`select` 事件在项目级别以及菜单级别进行处理。从包含所选项的菜单到根菜单的菜单链中的每个菜单都将响应 `select` 事件。`displaying` 事件与“Open Recent”（最近打开的项目）菜单一起使用。在打开菜单之前，将以最新的 `Documents` 数组（在此示例中实际上不发生更改）刷新菜单中的项目。尽管此示例中不显示，但还可以在个别项目上侦听 `displaying` 事件。

```
<html>
<head>
<script src="AIRAliases.js" type="text/javascript"></script>
<script type="text/javascript">
var application = air.NativeApplication.nativeApplication;
var recentDocuments =
    new Array(new air.File("app-storage:/GreatGatsby.pdf"),
              new air.File("app-storage:/WarAndPeace.pdf"),
              new air.File("app-storage:/Iliad.pdf"));

function MenuExample(){
    var fileMenu;
    var editMenu;

    if (air.NativeWindow.supportsMenu &&
        nativeWindow.systemChrome != air.NativeWindowSystemChrome.NONE) {
        nativeWindow.menu = new air.NativeMenu();
        nativeWindow.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();

        editMenu = nativeWindow.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }

    if (air.NativeApplication.supportsMenu) {
        application.menu.addEventListener(air.Event.SELECT, selectCommandMenu);
        fileMenu = application.menu.addItem(new air.NativeMenuItem("File"));
        fileMenu.submenu = createFileMenu();
        editMenu = application.menu.addItem(new air.NativeMenuItem("Edit"));
        editMenu.submenu = createEditMenu();
    }
}

function createFileMenu() {
    var fileMenu = new air.NativeMenu();
    fileMenu.addEventListener(air.Event.SELECT, selectCommandMenu);

    var newCommand = fileMenu.addItem(new air.NativeMenuItem("New"));
    newCommand.addEventListener(air.Event.SELECT, selectCommand);
    var saveCommand = fileMenu.addItem(new air.NativeMenuItem("Save"));
    saveCommand.addEventListener(air.Event.SELECT, selectCommand);
    var openFile = fileMenu.addItem(new air.NativeMenuItem("Open Recent"));
    openFile.submenu = new air.NativeMenu();
    openFile.submenu.addEventListener(air.Event.DISPLAYING, updateRecentDocumentMenu);
```

```
openFile_submenu.addEventListener(air.Event.SELECT, selectCommandMenu);

return fileMenu;
}

function createEditMenu() {
    var editMenu = new air.NativeMenu();
    editMenu.addEventListener(air.Event.SELECT,selectCommandMenu);

    var copyCommand = editMenu.addItem(new air.NativeMenuItem("Copy"));
    copyCommand.addEventListener(air.Event.SELECT,selectCommand);
    copyCommand.keyEquivalent = "c";
    var pasteCommand = editMenu.addItem(new air.NativeMenuItem("Paste"));
    pasteCommand.addEventListener(air.Event.SELECT, selectCommand);
    pasteCommand.keyEquivalent = "v";
    editMenu.addItem(new air.NativeMenuItem("", true));
    var preferencesCommand = editMenu.addItem(new air.NativeMenuItem("Preferences"));
    preferencesCommand.addEventListener(air.Event.SELECT,selectCommand);

    return editMenu;
}

function updateRecentDocumentMenu(event) {
    air.trace("Updating recent document menu.");
    var docMenu = air.NativeMenu(event.target);

    for (var i = docMenu.numItems - 1; i >= 0; i--) {
        docMenu.removeItemAt(i);
    }

    for (var file in recentDocuments) {
        var menuItem =
            docMenu.addItem(new air.NativeMenuItem(recentDocuments[file].name));
        menuItem.data = recentDocuments[file];
        menuItem.addEventListener(air.Event.SELECT, selectRecentDocument);
    }
}

function selectRecentDocument(event) {
    air.trace("Selected recent document: " + event.target.data.name);
}

function selectCommand(event) {
    air.trace("Selected command: " + event.target.label);
}

function selectCommandMenu(event) {
    if (event.currentTarget.parent != null) {
        var menuItem = findItemForMenu(event.currentTarget);
        if(menuItem != null){
            air.trace("Select event for \\" + event.target.label +
            "\\" command handled by menu: " + menuItem.label);
        }
    } else {
        air.trace("Select event for \\" + event.target.label +
        "
```

```

        "\" command handled by root menu.");
    }

}

function findItemForMenu(menu) {
    for (var item in menu.parent.items) {
        if (item != null) {
            if (itemsubmenu == menu) {
                return item;
            }
        }
    }
    return null;
}
</script>
<title>AIR menus</title>
</head>
<body onload="MenuExample()"></body>
</html>

```

使用 MenuBuilder 框架

除了标准菜单类以外，Adobe AIR 还包括一个菜单生成器 JavaScript 框架，使开发人员创建菜单更加容易。MenuBuilder 框架允许以声明方式定义 XML 或 JSON 格式的菜单结构。它还提供了帮助器方法，可以创建对 AIR 应用程序可用的任何菜单类型。若要查看如何在 AIR 中使用本机菜单的完整列表，请参阅第 137 页的“[AIR 菜单基础知识](#)”。

使用 MenuBuilder 框架创建菜单

MenuBuilder 框架允许使用 XML 或 JSON 定义菜单的结构。该框架包含的方法可以用于加载和分析包含菜单结构的文件。一旦加载了菜单结构，就可以用其它方法指定如何在应用程序中使用菜单。此方法允许将菜单设置为 Mac OS X 应用程序菜单、窗口菜单或上下文菜单。

MenuBuilder 框架未内置在运行时环境中。若要使用该框架，请在应用程序代码中包括 Adobe AIR SDK 中附带的 `AIRMenuBuilder.js` 文件，如下所示：

```
<script type="text/javascript" src="AIRMenuBuilder.js"></script>
```

MenuBuilder 框架在设计上运行于应用程序沙箱中。无法从经典沙箱调用此框架方法。

供开发人员使用的所有框架方法都被定义为 `air.ui.Menu` 类上的类方法。

MenuBuilder 基本工作流程

通常，不管要创建的菜单类型是什么，使用 MenuBuilder 框架创建菜单时都需要执行三个步骤：

- 1 定义菜单结构：创建一个文件，其中包含定义菜单结构的 XML 或 JSON。对于某些菜单类型，顶级菜单项是菜单（例如，在窗口菜单或应用程序菜单中）。对于其它菜单类型，顶级项目是单个菜单命令（比如在上下文菜单中）。若要查看定义菜单结构的格式的详细信息，请参阅第 149 页的“[定义 MenuBuilder 菜单结构](#)”。
- 2 加载菜单结构：调用相应的 `Menu` 类方法（`Menu.createFromXML()` 或 `Menu.createFromJSON()`），加载菜单结构文件并将其分析为实际的菜单对象。两种方法都返回一个 `NativeMenu` 对象，然后可以将此对象传递给该框架的菜单设置方法之一。
- 3 分配菜单：按照菜单的使用方式调用合适的 `Menu` 类方法。选项是：
 - `Menu.setAsMenu()`，用于窗口菜单或应用程序菜单
 - `Menu.setAsContextMenu()`，用于将菜单显示为 DOM 元素的上下文菜单

- `Menu.setAsIconMenu()`, 用于将菜单设置为系统任务栏或停靠栏图标的上下文菜单

确定何时执行代码很重要。尤其是，必须在创建实际操作系统窗口之前分配窗口菜单。任何将菜单设置为窗口菜单的 `setAsMenu()` 调用都必须直接在 HTML 页中执行，而不能在 `onload` 或其它事件处理函数中执行。在操作系统打开窗口之前，必须运行创建菜单的代码。同时，任何引用 DOM 元素的 `setAsContextMenu()` 调用都必须在创建该 DOM 元素之后进行。最安全的方法是将包含菜单分配代码的 `<script>` 块就放在 HTML 页末尾的结束 `</body>` 标记内。

加载菜单结构

不管菜单的用途是什么，都要将菜单结构定义为包含 XML 或 JSON 结构的单独文件。在应用程序中分配菜单之前，必须先使用框架加载和分析菜单结构文件。若要加载和分析菜单结构文件，请使用以下两个框架方法之一：

- `Menu.createFromXML()`, 用于加载和分析 XML 格式的菜单结构文件
- `Menu.createFromJSON()`, 用于加载和分析 JSON 格式的菜单结构文件

两种方法都接受一个参数：菜单结构文件的文件路径。两种方法都从该位置加载文件。它们分析文件内容，并返回 `NativeMenu` 对象，该对象中包含此文件中定义的菜单结构。例如，以下代码加载名为“`windowMenu.xml`”的菜单结构文件，该文件与加载它的 HTML 文件位于同一目录：

```
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
```

在下一个示例中，代码从名为“`menus`”的目录中加载名为“`contextMenu.js`”的菜单结构文件：

```
var contextMenu = air.ui.Menu.createFromJSON("menus/contextMenu.js");
```

注: 所生成的 `NativeMenu` 对象只能一次性用作应用程序菜单或窗口菜单。但是，所生成的 `NativeMenu` 对象可以在应用程序中多次用作上下文菜单或图标菜单。在 Mac OS X 上使用 `MenuBuilder` 框架时，如果将相同 `NativeMenu` 指定为应用程序菜单和另一种类型的菜单，则它只用作应用程序菜单。

有关 `MenuBuilder` 框架接受的特定菜单结构的详细信息，请参阅第 149 页的“[定义 MenuBuilder 菜单结构](#)”。

创建应用程序菜单或窗口菜单

使用 `MenuBuilder` 框架创建应用程序菜单或窗口菜单时，菜单数据结构中的顶级对象或节点对应于菜单栏中显示的项目。嵌套在这些顶级项目之一中的项目定义各个菜单命令。同样，这些菜单项可以包含其它项目。在此情况下，菜单项是一个子菜单，而不是一个命令。当用户选择菜单项时，它将展开自己的菜单项。

可以使用 `Menu.setAsMenu()` 方法将菜单设置为从中执行调用的窗口的应用程序菜单或窗口菜单。`setAsMenu()` 方法采用一个参数：要使用的 `NativeMenu` 对象。以下示例加载 XML 文件，并将生成的菜单设置为应用程序菜单或窗口菜单：

```
var windowMenu = air.ui.Menu.createFromXML("windowMenu.xml");
air.ui.Menu.setAsMenu(windowMenu);
```

在支持窗口菜单的操作系统中，`setAsMenu()` 调用将菜单设置为当前窗口（表示为 `window.nativeWindow` 的窗口）的窗口菜单。在支持应用程序菜单的操作系统上，此菜单用作应用程序菜单。

Mac OS X 将一组标准菜单定义为默认应用程序菜单，它们在每个应用程序中都具有一组相同的菜单项。这些菜单包括其名称与应用程序名称、“Edit”（编辑）菜单和“Window”（窗口）菜单匹配的应用程序菜单。通过调用 `Menu.setAsMenu()` 方法分配 `NativeMenu` 对象作为应用程序菜单时，`NativeMenu` 中的项目会插入到标准菜单结构中“编辑”菜单和“窗口”菜单之间。标准菜单不会被修改或替换。

如果您愿意，可以替换标准菜单，而不是补充它们。若要替换现有菜单，请将值为 `true` 的第二个参数传递给 `setAsMenu()` 调用，如本例所示：

```
air.ui.Menu.setAsMenu(windowMenu, true);
```

创建 DOM 元素上下文菜单

使用 MenuBuilder 框架创建 DOM 元素的上下文菜单涉及两个步骤。首先使用 Menu.createFromXML() 或 Menu.createFromJSON() 方法创建定义菜单结构的 NativeMenu 实例。然后，通过调用 Menu.setAsContextMenu() 方法指定该菜单作为 DOM 元素的上下文菜单。因为上下文菜单由单个菜单组成，所以菜单数据结构中的顶级菜单项充当单个菜单中的项目。包含子菜单项的任何菜单项定义子菜单。若要指定 NativeMenu 作为 DOM 元素的上下文菜单，请调用 Menu.setAsContextMenu() 方法。此方法需要两个参数：设置为上下文菜单的 NativeMenu 和分配给 DOM 元素的 ID（一个字符串）：

```
var treeContextMenu = air.ui.Menu.createFromXML("treeContextMenu.xml");
air.ui.Menu.setAsContextMenu(treeContextMenu, "navTree");
```

如果省略 DOM 元素参数，则此方法将使用调用该方法的 HTML 文档作为默认值。换句话说，该菜单将设置为 HTML 文档的整个窗口的上下文菜单。此技术通过传递 null 作为第一个参数，因此很容易从整个 HTML 窗口中删除默认上下文菜单，如本例所示：

```
air.ui.Menu.setAsContextMenu(null);
```

还可以从任何 DOM 元素中删除已分配的上下文菜单。调用 setAsContextMenu() 方法，并传递 null 和元素 ID 作为两个参数。

创建图标上下文菜单

除了应用程序窗口中的 DOM 元素的上下文菜单以外，Adobe AIR 应用程序还支持其它两个特殊的上下文菜单：停靠栏图标菜单（用于支持停靠栏的操作系统）和系统任务栏图标菜单（用于使用系统任务栏的操作系统）。若要设置这两个菜单中的任意一个，请首先使用 Menu.createFromXML() 或 Menu.createFromJSON() 方法创建一个 NativeMenu。然后，通过调用 Menu.setAsIconMenu() 方法，指定该 NativeMenu 作为停靠栏图标菜单或系统任务栏图标菜单。

此方法接受两个参数。第一个参数（必需）是要用作图标菜单的 NativeMenu。第二个参数是一个数组，其中包含要用作图标的图像文件路径的字符串，或者是包含图标的图像数据的 BitmapData 对象。除非在 application.xml 文件中指定了默认图标，否则此参数是必需的。如果在 application.xml 文件中指定了默认图标，则默认情况下这些图标将用作系统任务栏图标。

以下示例演示加载菜单数据，并将此菜单指定为停靠栏图标或系统任务栏图标的上下文菜单：

```
// Assumes that icons are specified in the application.xml file.
// Otherwise the icons would need to be specified using a second
// parameter to the setAsIconMenu() function.
var iconMenu = air.ui.Menu.createFromXML("iconMenu.xml");
air.ui.Menu.setAsIconMenu(iconMenu);
```

注：Mac OS X 为应用程序停靠栏图标定义了标准上下文菜单。将某个菜单指定为停靠栏图标上下文菜单时，此菜单中的项目将显示在标准 OS 菜单项的上方。不能删除、访问或修改标准菜单项。

定义 MenuBuilder 菜单结构

使用 Menu.createFromXML() 或 Menu.createFromJSON() 方法创建 NativeMenu 对象时，XML 元素或对象的结构决定所得菜单的结构。一旦创建了菜单，就可以在运行时更改其结构或属性。若要在运行时更改菜单项，请通过导航 NativeMenu 对象的层次结构访问 NativeMenuItem 对象。

当 MenuBuilder 框架通过菜单数据源分析时，它将查找某些 XML 属性或对象属性。这些属性 (attribute) 或属性 (property) 是否存在以及它们的值将确定所创建的菜单的结构。

使用 XML 表示菜单结构时，XML 文件必须包含根节点。根节点的子节点将用作顶级菜单项节点。XML 节点可以有任何名称。XML 节点的名称不影响菜单结构。仅节点的层次结构及其属性值用于定义菜单。

菜单项类型

菜单数据源中的每个条目（每个 XML 元素或 JSON 对象）都可以指定它所表示的菜单项的项目类型和特定于类型的信息。

Adobe AIR 支持以下菜单项类型，可以在数据源中将这些类型设置为 type 属性 (attribute) 或属性 (property) 的值：

菜单项类型	说明
normal	默认类型。选择 normal 类型的项目将触发 select 事件，并调用在数据源的 onSelect 字段中指定的函数。另外，如果项目有子项，则菜单项将调度 displaying 事件，并打开子菜单。
check	选择 check 类型的项目将使 NativeMenuItem 的 checked 属性在 true 和 false 值之间切换，并触发 select 事件，从而调用在数据源的 onSelect 字段中指定的函数。当菜单项处于 true 状态时，在菜单中此项目标签的旁边将显示复选标记。
separator	具有 separator 类型的项目将提供简单的水平线，将菜单中的项目划分到不同的可视组中。

正常菜单项被视为子菜单（如果有子项）。在使用 XML 数据源的情况下，这意味着菜单项元素包含其它 XML 元素。对于 JSON 数据源，需要为表示菜单项的对象指定一个名为 items 的属性，以包含由其它对象组成的数组。

菜单数据源属性 (attribute) 或属性 (property)

菜单数据源中的项目可以指定几个 XML 属性 (attribute) 或对象属性 (property)，用于确定项目的显示和行为方式。下表列出了可以指定的属性、其数据类型、其用途以及数据源必须如何表示它们：

属性 (attribute) 或属性 (property)	类型	说明
altKey	Boolean	指定 Alt 键是否作为项目的等效键的必要组成部分。
cmdKey	Boolean	指定 Command 键是否作为项目的等效键的必要组成部分。defaultKeyEquivalentModifiers 字段也会影响此值。
ctrlKey	Boolean	指定 Ctrl 键是否作为项目的等效键的必要组成部分。defaultKeyEquivalentModifiers 字段也会影响此值。
defaultKeyEquivalentModifiers	Boolean	指定操作系统的默认功能键（Mac OS X 的 Command 和 Windows 的 Ctrl）是否作为项目的等效键的必要组成部分。如果不指定，则 MenuBuilder 框架将该项目的值视为 true。
enabled	Boolean	指定用户是 (true) 否 (false) 可以选择菜单项。如果不指定，则 MenuBuilder 框架将该项目的值视为 true。
items	Array	(仅适用于 JSON) 指定菜单项本身是菜单。数组中的对象是包含于菜单中的子菜单项。
keyEquivalent	String	指定在按下时将触发像选择菜单项那样的事件的键盘字符。 如果此值是大写字符，则 Shift 键是项目的等效键的必要组成部分。
label	String	指定在控件中显示的文本。此项目用于除 separator 以外的所有菜单项类型。
mnemonicIndex	Integer	指定用作菜单项助记键的字符在标签中的索引位置。另外，还可以指示标签中的字符作为菜单项的助记键，方法是在紧靠该字符的左侧包括下划线。

属性 (attribute) 或属性 (property)	类型	说明
onSelect	String 或 Function	指定函数 (String) 的名称或对该函数 (Function 对象) 的引用。当用户选择菜单项时，所指定的函数将作为事件侦听器被调用。有关详细信息，请参阅第 154 页的“ 处理 MenuBuilder 菜单事件 ”。
shiftKey	String	指定 Shift 键是否作为项目的等效键的必要组成部分。 另外，keyEquivalent 值也可以指定此值。如果 keyEquivalent 值是大写字母，则 Shift 键是等效键的必要部分。
toggled	Boolean	指定是否选中了复选项。如果不指定，则 MenuBuilder 框架将该项目的值视为 false 并且未选中此项目。
type	String	指定菜单项的类型。有意义的值是 separator 和 check。MenuBuilder 框架将其它所有值、或没有 type 条目的元素或对象均视为正常菜单条目。

MenuBuilder 框架忽略其它所有对象属性 (property) 或 XML 属性 (attribute)。

示例：XML MenuBuilder 数据源

以下示例使用 MenuBuilder 框架定义文本区域的上下文菜单。它显示如何使用 XML 作为数据源来定义菜单结构。有关使用 JSON 数组指定相同菜单结构的应用程序，请参阅第 152 页的“[示例：JSON MenuBuilder 数据源](#)”。

此应用程序由两个文件组成。

第一个文件是菜单数据源，在名为“textContextMenu.xml”的文件中。尽管此示例使用的菜单项节点名为“menuItem”，但 XML 节点的实际名称是什么并不重要。前面提到过，只有 XML 的结构和属性值才会影响所生成的菜单的结构。

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menuItem label="MenuItem A"/>
    <menuItem label="MenuItem B" type="check" toggled="true"/>
    <menuItem label="MenuItem C" enabled="false"/>
    <menuItem type="separator"/>
    <menuItem label="MenuItem D">
        <menuItem label="SubMenu Item D-1"/>
        <menuItem label="SubMenu Item D-2"/>
        <menuItem label="SubMenu Item D-3"/>
    </menuItem>
</root>
```

第二个文件是应用程序用户界面的源代码（在 application.xml 文件中指定为初始窗口的 HTML 文件）：

```

<html>
    <head>
        <title>XML-based menu data source example</title>
        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="text/javascript" src="AIRMenuBuilder.js"></script>
        <style type="text/css">
            #contextEnabledText
            {
                margin-left: auto;
                margin-right: auto;
                margin-top: 100px;
                width: 50%
            }
        </style>
    </head>
    <body>
        <div id="contextEnabledText">This block of text is context menu enabled. Right click or Command-click on the text to view the context menu.</div>
        <script type="text/javascript">
            // Create a NativeMenu from "textContextMenu.xml" and set it
            // as context menu for the "contextEnabledText" DOM element:
            var textMenu = air.ui.Menu.createFromXML("textContextMenu.xml");
            air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

            // Remove the default context menu from the page:
            air.ui.Menu.setAsContextMenu(null);
        </script>
    </body>
</html>

```

示例：JSON MenuBuilder 数据源

以下示例使用 MenuBuilder 框架以 JSON 数组作为数据源定义一个文本区域的上下文菜单。有关在 XML 中指定相同菜单结构的应用程序，请参阅第 151 页的“[示例：XML MenuBuilder 数据源](#)”。

此应用程序由两个文件组成。

第一个文件是菜单数据源，在名为“textContextMenu.js”的文件中。

```
[
    {label: "MenuItem A"},
    {label: "MenuItem B", type: "check", toggled: "true"},
    {label: "MenuItem C", enabled: "false"},
    {type: "separator"},
    {label: "MenuItem D", items:
        [
            {label: "SubMenu Item D-1"},
            {label: "SubMenu Item D-2"},
            {label: "SubMenu Item D-3"}
        ]
    }
]
```

第二个文件是应用程序用户界面的源代码（在 application.xml 文件中指定为初始窗口的 HTML 文件）：

```
<html>
  <head>
    <title>JSON-based menu data source example</title>
    <script type="text/javascript" src="AIRAliases.js"></script>
    <script type="text/javascript" src="AIRMenuBuilder.js"></script>
    <style type="text/css">
      #contextEnabledText
      {
        margin-left: auto;
        margin-right: auto;
        margin-top: 100px;
        width: 50%
      }
    </style>
  </head>
  <body>
    <div id="contextEnabledText">This block of text is context menu enabled. Right click or Command-click on the text to view the context menu.</div>
    <script type="text/javascript">
      // Create a NativeMenu from "textContextMenu.js" and set it
      // as context menu for the "contextEnabledText" DOM element:
      var textMenu = air.ui.Menu.createFromJSON("textContextMenu.js");
      air.ui.Menu.setAsContextMenu(textMenu, "contextEnabledText");

      // Remove the default context menu from the page:
      air.ui.Menu.setAsContextMenu(null);
    </script>
  </body>
</html>
```

用 MenuBuilder 添加菜单键盘功能

操作系统本机菜单支持使用快捷键，这些快捷键也可在 Adobe AIR 中使用。可以在菜单数据源中指定的两类快捷键是菜单命令等效键和助记键。

指定菜单等效键

可以为窗口或应用程序的菜单命令指定等效键（有时称为快捷键）。当按下键或组合键时，NativeMenuItem 将调度 select 事件，并调用在数据源中指定的任何 onSelect 事件处理函数。行为与用户选择菜单项相同。

有关菜单等效键的完整详细信息，请参阅第 140 页的“[菜单命令的等效键](#)”。

通过使用 MenuBuilder 框架，可以在数据源的相应节点中指定菜单项的等效键。如果数据源有 keyEquivalent 字段，则 MenuBuilder 框架使用该值作为等效键字符。

还可以指定作为等效组合键的一部分的功能键。若要添加功能键，请将 altKey、ctrlKey、cmdKey 或 shiftKey 字段指定为 true。所指定的一个或多个键将成为等效组合键的一部分。默认情况下，对 Windows 指定 Ctrl 键，对 Mac OS X 指定 Command 键。若要覆盖此默认行为，请包括设置为 false 的 defaultKeyEquivalentModifiers 字段。

以下示例显示基于 XML 的菜单数据源（在名为“keyEquivalentMenu.xml”的文件中包含等价键）的数据结构：

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menuitem label="File">
        <menuitem label="New" keyEquivalent="n"/>
        <menuitem label="Open" keyEquivalent="o"/>
        <menuitem label="Save" keyEquivalent="s"/>
        <menuitem label="Save As..." keyEquivalent="s" shiftKey="true"/>
        <menuitem label="Close" keyEquivalent="w"/>
    </menuitem>
    <menuitem label="Edit">
        <menuitem label="Cut" keyEquivalent="x"/>
        <menuitem label="Copy" keyEquivalent="c"/>
        <menuitem label="Paste" keyEquivalent="v"/>
    </menuitem>
</root>
```

以下示例应用程序从“keyEquivalentMenu.xml”加载菜单结构，并使用它作为应用程序的窗口菜单或应用程序菜单的结构：

```
<html>
    <head>
        <title>XML-based menu with key equivalents example</title>
        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="text/javascript" src="AIRMenuBuilder.js"></script>
    </head>
    <body>
        <script type="text/javascript">
            // Create a NativeMenu from "keyEquivalentMenu.xml" and set it
            // as the application/window menu
            var keyEquivMenu = air.ui.Menu.createFromXML("keyEquivalentMenu.xml");
            air.ui.Menu.setAsMenu(keyEquivMenu);
        </script>
    </body>
</html>
```

指定菜单项助记键

菜单项助记键是与菜单项关联的键。如果在显示此菜单时按下该键，将触发此菜单项命令。行为与用户用鼠标选择菜单项相同。通常，操作系统通过对菜单项名称中的字符添加下划线来指示菜单项助记键。

有关助记键的详细信息，请参阅第 140 页的“[助记键](#)”。

对于 **MenuBuilder** 框架，指定菜单项助记键的最简单方式是在该菜单项的 **label** 字段中加入一个下划线字符（“_”）。请将下划线放在紧靠充当该菜单项助记键的字母左侧。例如，如果在使用 **MenuBuilder** 框架加载的数据源中使用以下 XML 节点，则此命令的助记键是第二个单词的第一个字符（字母“A”）：

```
<menuitem label="Save _As"/>
```

创建 **NativeMenu** 对象时，下划线不包括在标签中。而是以下划线后面的字符作为菜单项的助记键。若要在菜单项的名称中显示出一个下划线字符，请使用两个下划线字符（“__”）。此序列将在菜单项标签中将转换为一个下划线。

作为在 **label** 字段中使用下划线字符的替代选择，可以为助记键字符提供整数索引位置。可以在菜单项数据源对象或 XML 元素的 **mnemonicIndex** 字段中指定索引。

处理 **MenuBuilder** 菜单事件

与 **NativeMenu** 的用户交互是事件驱动的。当用户选择菜单项或打开菜单或子菜单时，**NativeMenuItem** 对象将调度一个事件。使用通过 **MenuBuilder** 框架创建的 **NativeMenu** 对象，可以将事件侦听器注册到各个 **NativeMenuItem** 对象或 **NativeMenu**。订阅和响应这些事件时，就好像您已经以手动方式而不是使用 **MenuBuilder** 框架创建了 **NativeMenu** 和 **NativeMenuItem** 对象。有关详细信息，请参阅第 139 页的“[菜单的事件](#)”。

MenuBuilder 框架补充了标准事件处理过程，从而使您可以在菜单数据源中为菜单项指定 `select` 事件处理函数。如果在菜单项数据源中指定 `onSelect` 字段，则当用户选择菜单项时，将调用所指定的函数。例如，假设以下 XML 节点包含在使用 **MenuBuilder** 框架加载的数据源中。当选择菜单项时，将调用名为 `doSave()` 的函数：

```
<menuitem label="Save" onSelect="doSave"/>
```

用于 XML 数据源时，`onSelect` 字段是 `String`。使用 JSON 数组时，此字段可以是有函数名称的 `String`。此外，仅对 JSON 数组，该字段还可以是对作为对象的函数的变量引用。但是，如果 JSON 数组使用 `Function` 变量引用，则必须在发生 `onload` 事件处理函数或 `JavaScript` 安全违规之前或在此期间创建菜单。在所有情况中，必须在全局作用域内定义所指定的函数。

调用所指定的函数时，运行时会向它传递两个参数。第一个参数是由 `select` 事件调度的事件对象。它是 `Event` 类的实例。传递给此函数的第二个参数是匿名对象，其中包含用于创建菜单项的数据。此对象有以下属性。每个属性的值均与原始数据结构中的值匹配，如果未在原始数据结构中设置该属性，则其值为 `null`：

- `altKey`
- `cmdKey`
- `ctrlKey`
- `defaultKeyEquivalentModifiers`
- `enabled`
- `keyEquivalent`
- `label`
- `mnemonicIndex`
- `onSelect`
- `shiftKey`
- `toggled`
- `type`

以下示例用于实验 `NativeMenu` 事件。该示例包括两个菜单。窗口和应用程序菜单使用 XML 数据源创建。`` 和 `` 元素所表示项目的上下文菜单使用 JSON 数组数据源创建。当用户选择菜单项时，屏幕上的文本区域将显示有关每个事件的信息。

以下代码是应用程序的源代码：

```
<html>
<head>
<title>Menu event handling example</title>
<script type="text/javascript" src="AIRAliases.js"></script>
<script type="text/javascript" src="AIRMenuBuilder.js"></script>
<script type="text/javascript" src="printObject.js"></script>
<script type="text/javascript">
    function fileMenuCommand(event, data) {
        print("fileMenuCommand", event, data);
    }

    function editMenuCommand(event, data) {
        print("editMenuCommand", event, data);
    }

    function moveItemUp(event, data) {
        print("moveItemUp", event, data);
    }

    function moveItemDown(event, data) {
        print("moveItemDown", event, data);
    }
</script>

```

```
}

function print(command, event, data) {
    var result = "";
    result += "<h1>Command: " + command + '</h1>';
    result += "<p>" + printObject(event) + "</p>";
    result += "<p>Data:</p>";
    result += "<ul>";
    for (var s in data) {
        result += "<li>" + s + ":" + printObject(data[s]) + "</li>";
    }
    result += "</ul>";

    var o = document.getElementById("output");
    o.innerHTML = result;
}

</script>
<style type="text/css">
    #contextList {
        position: absolute; left: 0; top: 25px; bottom: 0; width: 100px;
        background: #eeeeee;
    }
    #output {
        position: absolute; left: 125px; top: 25px; right: 0; bottom: 0;
    }
</style>
</head>
<body>
    <div id="contextList">
        <ul>
            <li>List item 1</li>
            <li>List item 2</li>
            <li>List item 3</li>
        </ul>
    </div>
    <div id="output">
        Choose menu commands. Information about the events displays here.
    </div>
    <script type="text/javascript">
        var mainMenu = air.ui.Menu.createFromXML("mainMenu.xml");
        air.ui.Menu.setAsMenu(mainMenu);

        var listContextMenu = air.ui.Menu.createFromJSON("listContextMenu.js");
        air.ui.Menu.setAsContextMenu(listContextMenu, "contextList")

        // clear the default context menu
        air.ui.Menu.setAsContextMenu(null);
    </script>
</body>
</html>
```

以下代码是主菜单（“mainMenu.xml”）的数据源：

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
    <menutem label="File">
        <menutem label="New" keyEquivalent="n" onSelect="fileMenuCommand"/>
        <menutem label="Open" keyEquivalent="o" onSelect="fileMenuCommand"/>
        <menutem label="Save" keyEquivalent="s" onSelect="fileMenuCommand"/>
        <menutem label="Save As..." keyEquivalent="S" onSelect="fileMenuCommand"/>
        <menutem label="Close" keyEquivalent="w" onSelect="fileMenuCommand"/>
    </menutem>
    <menutem label="Edit">
        <menutem label="Cut" keyEquivalent="x" onSelect="editMenuCommand"/>
        <menutem label="Copy" keyEquivalent="c" onSelect="editMenuCommand"/>
        <menutem label="Paste" keyEquivalent="v" onSelect="editMenuCommand"/>
    </menutem>
</root>
```

以下代码是上下文菜单（“listContextMenu.js”）的数据源；

```
[{label: "Move Item Up", onSelect: "moveItemUp"}, {label: "Move Item Down", onSelect: "moveItemDown"}]
```

以下代码包含 printObject.js 文件中的代码。在本例中，该文件包括 printObject() 函数，该函数由应用程序使用，但它不影响菜单的操作。

```
function printObject(obj) {
    if (!obj) {
        if (typeof obj == "undefined") { return "[undefined]"; };
        if (typeof obj == "object") { return "[null]"; };
        return "[false]";
    } else {
        if (typeof obj == "boolean") { return "[true]"; };
        if (typeof obj == "object") {
            if (typeof obj.length == "number") {
                var ret = [];
                for (var i=0; i<obj.length; i++) {
                    ret.push(printObject(obj[i]));
                }
                return "[" + ret.join(", ") + "]";
            } else {
                var ret = [];
                var hadChildren = false;
                for (var k in obj) {
                    hadChildren = true;
                    ret.push([k, " => ", printObject(obj[k])]);
                }
                if (hadChildren) {
                    return ["{\n", ret.join(",\n"), "\n}"].join("");
                }
            }
        }
        if (typeof obj == "function") { return "[Function]"; }
        return String(obj);
    }
}
```

第 20 章：任务栏图标

很多操作系统都提供有任务栏（例如 Mac OS X 停靠栏），任务栏中可包含表示应用程序的图标。Adobe® AIR® 提供了一个接口，可以通过 NativeApplication.nativeApplication.icon 属性与应用程序任务栏图标进行交互。

有关任务栏图标的其它在线信息

可以从以下源中查找有关使用任务栏图标的详细信息：

快速入门（**Adobe AIR** 开发人员联盟）

语言参考

- [DockIcon](#)
- [SystemTrayIcon](#)

Adobe 开发人员联盟文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员联盟（搜索“AIR 任务栏图标”）](#)

关于任务栏图标

AIR 会自动创建 NativeApplication.nativeApplication.icon 对象。对象类型可以为 DockIcon 或 SystemTrayIcon，具体取决于操作系统。可以使用 NativeApplication.supportsDockIcon 和 NativeApplication.supportsSystemTrayIcon 属性来确定 AIR 在当前操作系统上支持哪些 InteractiveIcon 子类。InteractiveIcon 基类提供了 width、height 和 bitmaps 属性，可以使用这些属性来更改图标所使用的图像。但是，在错误操作系统上访问特定于 DockIcon 或 SystemTrayIcon 的属性会生成运行时错误。

若要设置或更改图标所使用的图像，请创建一个包含一个或多个图像的数组，然后将该数组分配给

NativeApplication.nativeApplication.icon.bitmaps 属性。在不同操作系统上，任务栏图标的大小会有所不同。为了避免因缩放而导致图像质量降级，可以向 bitmaps 数组中添加多个不同大小的图像。如果提供多个图像，AIR 会选择大小与任务栏图标的当前显示大小最接近的图像，仅在需要时进行缩放。以下示例使用两个图像来设置任务栏图标的图像：

```
air.NativeApplication.nativeApplication.icon.bitmaps =
[bmp16x16.bitmapData, bmp128x128.bitmapData];
```

若要更改图标图像，请将包含新图像的数组分配给 bitmaps 属性。通过响应 enterFrame 或 timer 事件来更改图像，可以为图标添加动画效果。

若要从 Windows 和 Linux 的通知区域中删除图标，或者恢复 Mac OS X 中的默认图标外观，请将 bitmaps 设置为空数组：

```
air.NativeApplication.nativeApplication.icon.bitmaps = [];
```

停靠栏图标

AIR 在 NativeApplication.supportsDockIcon 为 true 时支持停靠栏图标。NativeApplication.nativeApplication.icon 属性表示停靠栏上的应用程序图标（而不是窗口的停靠栏图标）。

注: AIR 不支持更改 Mac OS X 停靠栏上的窗口图标。此外, 对应用程序的停靠栏图标所做的更改只有当应用程序运行时才会应用, 应用程序终止时图标将还原为其正常外观。

停靠栏图标菜单

通过创建包含命令的 NativeMenu 对象并将其分配给 NativeApplication.nativeApplication.icon.menu 属性, 可以向标准停靠栏菜单中添加命令。菜单中的项目显示在标准停靠栏图标菜单项的上方。

回弹停靠栏

通过调用 NativeApplication.nativeApplication.icon.bounce() 方法, 可以回弹停靠栏图标。如果将 bounce() priority 参数设置为 informational, 则图标将回弹一次。如果将该参数设置为 critical, 则图标将始终保持回弹状态, 直到用户激活该应用程序。用作 priority 参数的常量由 NotificationType 类定义。

注: 如果应用程序已经处于活动状态, 则不会回弹图标。

停靠栏图标事件

单击停靠栏图标后, NativeApplication 对象将调度 invoke 事件。如果应用程序尚未运行, 则系统将启动该应用程序。否则, invoke 事件将传递到正在运行的应用程序实例。

系统任务栏图标

当 NativeApplication.supportsSystemTrayIcon 为 true 时, AIR 支持系统任务栏图标 (目前只有 Windows 和大多数 Linux 发行版是这种情况)。在 Windows 和 Linux 中, 系统任务栏图标显示在任务栏的通知区域中。默认情况下不显示任何图标。要显示图标, 请将包含 BitmapData 对象的数组分配给图标的 bitmaps 属性。若要更改图标图像, 请将包含新图像的数组分配给 bitmaps。若要删除图标, 请将 bitmaps 设置为 null。

系统任务栏图标菜单

通过创建 NativeMenu 对象并将其分配给 NativeApplication.nativeApplication.icon.menu 属性, 可以向系统任务栏图标中添加菜单 (操作系统不会提供任何默认菜单)。右键单击图标即可访问系统任务栏图标菜单。

系统任务栏图标工具提示

通过设置 tooltip 属性可以向图标添加工具提示:

```
air.NativeApplication.nativeApplication.icon.tooltip = "Application name";
```

系统任务栏图标事件

NativeApplication.nativeApplication.icon 属性引用的 SystemTrayIcon 对象可以为 click、mouseDown、mouseUp、rightClick、rightMouseDown 和 rightMouseUp 事件调度 ScreenMouseEvent。可以组合使用这些事件和图标菜单, 以便用户能够在您的应用程序没有可见窗口时与应用程序进行交互。

示例: 创建不带任何窗口的应用程序

以下示例创建了一个具有系统任务栏图标但没有可见窗口的 AIR 应用程序。(在应用程序描述符中, 不得将应用程序的 visible 属性设置为 true, 否则窗口在应用程序启动时可见。)

```

<html>
<head>
<script src="AIRAliases.js" language="JavaScript" type="text/javascript"></script>
<script language="JavaScript" type="text/javascript">
    var iconLoadComplete = function(event)
    {
        air.NativeApplication.nativeApplication.icon.bitmaps = [event.target.content.bitmapData];
    }

    air.NativeApplication.nativeApplication.autoExit = false;
    var iconLoad = new air.Loader();
    var iconMenu = new air.NativeMenu();
    var exitCommand = iconMenu.addItem(new air.NativeMenuItem("Exit"));
    exitCommand.addEventListener(air.Event.SELECT, function(event){
        air.NativeApplication.nativeApplication.icon.bitmaps = [];
        air.NativeApplication.nativeApplication.exit();
    });

    if (air.NativeApplication.supportsSystemTrayIcon) {
        air.NativeApplication.nativeApplication.autoExit = false;
        iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE, iconLoadComplete);
        iconLoad.load(new air.URLRequest("icons/AIRApp_16.png"));
        air.NativeApplication.nativeApplication.icon.tooltip = "AIR application";
        air.NativeApplication.nativeApplication.icon.menu = iconMenu;
    }

    if (air.NativeApplication.supportsDockIcon) {
        iconLoad.contentLoaderInfo.addEventListener(air.Event.COMPLETE, iconLoadComplete);
        iconLoad.load(new air.URLRequest("icons/AIRApp_128.png"));
        air.NativeApplication.nativeApplication.icon.menu = iconMenu;
    }
}

</script>
</head>
<body>
</body>
</html>

```

注: 该示例假设应用程序的 icons 子目录中存在名为 AIRApp_16.png 和 AIRApp_128.png 的图像文件。(示例图标文件包含在 AIR SDK 中, 您可以将这些文件复制到项目文件夹中。)

Window 任务栏图标和按钮

窗口的图标化表示形式通常显示在任务栏或停靠栏的窗口区域中, 以便用户能够轻松访问后台窗口或最小化的窗口。Mac OS X 停靠栏会为您的应用程序显示一个图标, 并为每个最小化的窗口分别显示一个图标。Microsoft Windows 和 Linux 任务栏显示一个按钮, 其中包含您的应用程序中每个普通类型窗口的程序图标和标题。

加亮显示任务栏窗口按钮

如果窗口位于后台, 则可以通知用户发生了与该窗口相关的需要关注的事件。在 Mac OS X 中, 可以通过回弹应用程序的停靠栏图标来通知用户 (如第 159 页的“[回弹停靠栏](#)”中所述)。在 Windows 和 Linux 中, 可以通过调用 NativeWindow 实例的 notifyUser() 方法来加亮显示窗口的任务栏按钮。传递给该方法的 type 参数用于确定通知的紧急程度:

- NotificationType.CRITICAL: 在用户将窗口置于前台之前, 窗口图标一直闪烁。
- NotificationType.INFORMATIONAL: 通过更改颜色来加亮显示窗口图标。

注: 在 Linux 中, 仅支持信息性类型的通知。向 notifyUser() 函数传递任何一种类型值都会产生相同的效果。

以下语句加亮显示窗口的任务栏按钮：

```
window.nativeWindow.notifyUser(air.NotificationType.INFORMATIONAL);
```

在不支持窗口级别通知的操作系统中，调用 NativeWindow.notifyUser() 方法将不起作用。使用 NativeWindow.supportsNotification 属性可确定是否支持窗口通知。

创建不带任务栏按钮或图标的窗口

在 Windows 操作系统中，使用 utility 或 lightweight 类型创建的窗口不会显示在任务栏中。不可见窗口也不会显示在任务栏中。

由于初始窗口必定为 normal 类型，因此要创建不会在任务栏中显示任何窗口的应用程序，必须关闭初始窗口或保持初始窗口不可见。若要关闭应用程序中的所有窗口而不终止应用程序，请在关闭最后一个窗口之前，将 NativeApplication 对象的 autoExit 属性设置为 false。若要使初始窗口变得不可见，请向应用程序描述符文件的 <initialWindow> 元素添加 <visible>false</visible>（且不要将 visible 属性设置为 true 或调用窗口的 activate() 方法）。

在应用程序打开的新窗口中，将传递给窗口构造函数的 NativeWindowInitOption 对象的 type 属性设置为 NativeWindowType.UTILITY 或 NativeWindowType.LIGHTWEIGHT。

在 Mac OS X 中，最小化的窗口显示在停靠任务栏中。通过隐藏窗口而不是最小化窗口可以避免显示最小化的图标。以下示例侦听 nativeWindowDisplayState 更改事件，并在窗口最小化时取消该事件。处理函数会改为将窗口的 visible 属性设置为 false：

```
function preventMinimize(event){  
    if(event.afterDisplayState == air.NativeWindowDisplayState.MINIMIZED){  
        event.preventDefault();  
        event.target.visible = false;  
    }  
}
```

如果将 visible 属性设置为 false 后窗口最小化到 Mac OS X 停靠栏中，则无法删除该停靠栏图标。用户仍可单击图标来重新显示窗口。

第 21 章：使用文件系统

使用 Adobe® AIR® 文件系统 API 提供的类可以访问主机的文件系统。使用这些类，您可以访问和管理目录和文件、创建目录和文件、向文件中写入数据等。

有关 AIR 文件 API 的其它在线信息

可以从以下源中查找有关使用文件 API 类的详细信息：

快速入门（**Adobe AIR** 开发人员联盟）

- [构建文本文件编辑器](#)
- [构建目录搜索应用程序](#)
- [从 XML 首选参数文件中读取和写入](#)

语言参考

- [File](#)
- [FileStream](#)
- [FileMode](#)

Adobe 开发人员联盟文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员联盟（搜索“AIR 文件系统”）](#)

AIR 文件基础知识

您可以使用 Adobe AIR 提供的类访问、创建和管理文件和文件夹。这些类包含在 `flash.filesystem` 包中，用法如下所示：

您可以使用 Adobe AIR 提供的类访问、创建和管理文件和文件夹。这些类包含在 `runtime.flash.filesystem` 包中，用法如下所示：

File 类	说明
File	File 对象表示文件或目录的路径。您可以使用 file 对象创建指向文件或文件夹的指针，启动与文件或文件夹的交互。
FileMode	FileMode 类定义 FileStream 类的 open() 和 openAsync() 方法的 fileMode 参数中使用的字符串常量。这些方法的 fileMode 参数确定文件打开后 FileStream 对象可用的功能，包括写入、读取、追加和更新功能。
FileStream	FileStream 对象用于打开文件以进行读取和写入。创建了指向新文件或现有文件的 File 对象后，可以将该指针传递到 FileStream 对象，以便您可以打开文件，然后对文件中的数据进行操作。

File 类中的一些方法具有同步版本和异步版本：

- `File.copyTo()` 和 `File.copyToAsync()`
- `File.deleteDirectory()` 和 `File.deleteDirectoryAsync()`
- `File.deleteFile()` 和 `File.deleteFileAsync()`

- File.getDirectoryListing() and File.getDirectoryListingAsync()
- File.moveTo() and File.moveToAsync()
- File.moveToTrash() and File.moveToTrashAsync()

另外，FileStream 操作是以同步方式运行还是以异步方式运行取决于 FileStream 对象打开文件的方式：是通过调用 open() 方法还是通过调用 openAsync() 方法。

使用异步版本可以启动在后台运行的进程，然后在完成时（或出现错误事件时）调度事件。在运行异步后台进程的同时可以执行其他代码。操作以异步方式运行时，必须使用调用该函数的 File 或 FileStream 对象的 addEventListener() 方法设置事件侦听器函数。

使用同步版本可以编写较简单的代码，而无需设置事件侦听器。不过，由于在执行同步方法的同时无法执行其他代码，可能会暂停重要的进程（如显示对象呈现和动画）。

使用 File 对象

File 对象是指向文件系统中文件或目录的指针。

File 类扩展了 FileReference 类。在 Adobe® Flash® Player 和 AIR 中可用的 FileReference 类表示指向文件的指针，但 File 类添加了一些属性和方法，出于安全方面的考虑，Flash Player 中（在浏览器中运行的 SWF 文件中）未公开这些属性和方法。

关于 File 类

您可以使用 File 类执行以下操作：

- 获取特殊目录的路径，包括用户目录、用户的文档目录、应用程序的启动目录以及应用程序目录
- 复制文件和目录
- 移动文件和目录
- 删除文件和目录（或将它们移到垃圾桶）
- 列出目录中包含的文件和目录
- 创建临时文件和文件夹

当 File 对象指向文件路径后，您可以通过 FileStream 类使用该 File 对象读取和写入文件数据。

File 对象可以指向尚不存在的文件或目录的路径。创建文件或目录时可以使用这种 File 对象。

File 对象的路径

每个 File 对象具有两个属性，各属性可分别定义该对象的路径：

属性	说明
nativePath	指定文件在特定平台上的路径。例如，在 Windows 中，路径可能是“c:\Sample directory\test.txt”，而在 Mac OS 中，路径可能是“/Sample directory/test.txt”。nativePath 属性在 Windows 中使用反斜杠 (\) 字符作为目录分隔符，在 Mac OS 和 Linux 中使用正斜杠 (/) 字符。
url	此属性可以使用 file URL 方案指向文件。例如，在 Windows 中，路径可能是“file:///c:/Sample%20directory/test.txt”，而在 Mac OS 中，路径可能是“file:///Sample%20directory/test.txt”。除 file 之外，运行时还包括其他特殊 URL 方案，在第 168 页的“ 支持的 URL 方案 ”中将予以介绍

File 类包括用于指向 Mac OS、Windows 和 Linux 中的标准目录的静态属性。这些属性包括：

- File.applicationStorageDirectory — 每个已安装的 AIR 应用程序独有的存储目录
- File.applicationDirectory — 安装应用程序的目录（其中包括所有已安装的资源）
- File.desktopDirectory — 用户的桌面目录
- File.documentsDirectory — 用户的文档目录
- File.userDirectory — 用户目录

这些属性的值在不同操作系统中均有意义。例如，在 Mac OS、Linux 和 Windows 上，用户桌面目录的本机路径各不相同。但是，File.desktopDirectory 属性在上述每个平台上都会指向正确的桌面目录路径。要编写可以跨平台正常工作的应用程序，在需要引用应用程序使用的其它目录和文件时，请以这些属性为基础，然后使用 resolvePath() 方法来完善路径。例如，此代码会指向应用程序存储目录中的 preferences.xml 文件：

```
var prefsFile:File = air.File.applicationStorageDirectory;
prefsFile = prefsFile.resolvePath("preferences.xml");
```

尽管 File 类用于指向特定文件路径，但这会导致应用程序无法跨平台工作。例如，路径 C:\Documents and Settings\joe\ 仅适用于 Windows。出于以上原因，最好使用 File 类的静态属性，如 File.documentsDirectory。

下一节中将提供有关这些 File 属性的详细信息。

将 File 对象指向目录

可以采用多种不同方式设置 File 对象以使其指向某目录。

指向用户的主目录

您可以将 File 对象指向用户的主目录。在 Windows 中，主目录是“My Documents”目录（例如，“C:\Documents and Settings\userName\My Documents”）的父级。在 Mac OS 中，它是 Users/userName 目录。在 Linux 中为 /home/userName 目录。以下代码将设置 File 对象以使其指向主目录中的 AIR Test 子目录：

```
var file = air.File.userDirectory.resolvePath("AIR Test");
```

指向用户的文档目录

您可以将 File 对象指向用户的文档目录。在 Windows 中，默认位置为“我的文档”目录（例如“C:\Documents and Settings\userName\My Documents”）。在 Mac OS 中，默认位置为 Users/userName/Documents 目录。在 Linux 中，默认位置为 /home/userName/Documents 目录。以下代码设置 File 对象以指向文档目录中的 AIR Test 子目录：

```
var file = air.File.documentsDirectory.resolvePath("AIR Test");
```

指向桌面目录

您可以使 File 对象指向桌面。以下代码设置 File 对象以使其指向桌面的 AIR Test 子目录：

```
var file = air.File.desktopDirectory.resolvePath("AIR Test");
```

指向应用程序存储目录

您可以使 File 对象指向应用程序存储目录。对于每个 AIR 应用程序，有一个唯一的关联路径用于定义应用程序存储目录。此目录对每个应用程序和用户是唯一的。您可能希望使用此目录存储特定于用户、特定于应用程序的数据（如用户数据或首选参数文件）。例如，以下代码将使 File 对象指向应用程序存储目录中包含的首选参数文件 prefs.xml：

```
var file = air.File.applicationStorageDirectory;
file = file.resolvePath("prefs.xml");
```

应用程序存储目录的位置由用户名、应用程序 ID 和发布者 ID 共同确定：

- 在 Mac OS 中，位于：

/Users/ 用户名/Library/Preferences/ 应用程序 ID. 发行商 ID/Local Store/

例如：

```
/Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local
1 Store
```

- 在 Windows 中，位于 Documents and Settings 目录下的以下位置：

用户名/Application Data/ 应用程序 ID. 发行商 ID/Local Store/

例如：

```
C:\Documents and Settings\babbage\Application
Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store
```

- 在 Linux 中位于：

/home/ 用户名/.appdata/ 应用程序 ID. 发行商 ID/Local Store/

例如：

```
/home/babbage/.appdata/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local Store
```

通过 url 创建的 File 对象的（和 File.applicationStorageDirectoryurl 属性）将使用 app-storage URL 方案（请参阅第 168 页的“[支持的 URL 方案](#)”），如下所示：

```
var dir = air.File.applicationStorageDirectory;
dir = dir.resolvePath("prefs.xml");
air.trace(dir.url); // app-storage:/preferences
```

指向应用程序目录

您可以使 File 对象指向应用程序的安装目录，即应用程序目录。您可以使用 File.applicationDirectory 属性引用此目录。您可以使用此目录检查应用程序描述符文件或与应用程序一起安装的其他资源。例如，以下代码将使 File 对象指向应用程序目录中名为 images 的目录：

```
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
```

通过 url 创建的 File 对象的（和 File.applicationDirectoryurl 属性）将使用 app URL 方案（请参阅第 168 页的“[支持的 URL 方案](#)”），如下所示：

```
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("images");
air.trace(dir.url); // app:/images
```

指向文件系统根目录

File.getRootDirectories() 方法列出 Windows 计算机上的所有根卷，如 C: 和已装载的卷。在 Mac OS 和 Linux 中，此方法始终返回计算机的唯一根目录（“/”目录）。

指向明确的目录

通过设置 File 对象的 nativePath 属性，可以使 File 对象指向某个明确的目录，如以下示例中所示（在 Windows 中）：

```
var file = new air.File();
file.nativePath = "C:\\AIR Test\\";
```

重要说明：通过这种方式指向明确的路径会导致代码无法跨平台使用。例如，上面的示例仅适用于 Windows。您应使用 File 类的静态属性（如 File.applicationStorageDirectory）来定位跨平台工作的目录。然后使用 resolvePath() 方法（请参阅下一节）导航到某个相对路径。

导航到相对路径

可以使用 resolvePath() 方法获取相对于另一个给定路径的路径。例如，以下代码将设置 File 对象以使其指向用户主目录中的“AIR Test”子目录：

```
var file = air.File.userDirectory;
file = file.resolvePath("AIR Test");
```

您还可以使用 File 对象的 url 属性以使该对象指向基于 URL 字符串的目录，如下所示：

```
var urlStr = "file:///C:/AIR Test/";
var file = new air.File()
file.url = urlStr;
```

有关详细信息，请参阅第 167 页的“[修改文件路径](#)”。

让用户浏览以选择目录

File 类包括 browseForDirectory() 方法，它显示一个系统对话框，用户从中可以选择要分配给对象的目录。

browseForDirectory() 方法为异步方法。如果用户选择一个目录并单击“打开”按钮，它将调度一个 select 事件；或者如果用户单击“取消”按钮，它将调度一个 cancel 事件。

例如，以下代码能使用户选择一个目录，并在选择后输出目录路径：

```
var file = new air.File();
file.addEventListener(air.Event.SELECT, dirSelected);
file.browseForDirectory("Select a directory");
function dirSelected(event) {
    alert(file.nativePath);
}
```

指向从中调用应用程序的目录

通过检查调用应用程序时所调度的 InvokeEvent 对象的 currentDirectory 属性，可以获取从中调用应用程序的目录位置。有关详细信息，请参阅第 279 页的“[捕获命令行参数](#)”。

将 File 对象指向文件

可采用多种不同方式设置 File 对象所指向的文件。

指向明确的文件路径

重要说明：指向明确的路径会导致代码无法跨平台工作。例如，路径 C:/foo.txt 仅适用于 Windows。您应使用 File 类的静态属性（如 File.applicationStorageDirectory）来定位跨平台工作的目录。然后使用 resolvePath() 方法（请参阅第 167 页的“[修改文件路径](#)”）导航到某个相对路径。

您可以使用 File 对象的 url 属性以使该对象指向基于 URL 字符串的文件或目录，如下所示：

```
var urlStr = "file:///C:/AIR Test/test.txt";
var file = new air.File()
file.url = urlStr;
```

还可以将 URL 传递到 File() 构造函数，如下所示：

```
var urlStr = "file:///C:/AIR Test/test.txt";
var file = new air.File(urlStr);
```

url 属性始终返回 URI 编码形式的 URL (例如将空格替换为 "%20") :

```
file.url = "file:///c:/AIR Test";
alert(file.url); // file:///c:/AIR%20Test
```

您还可以使用 File 对象的 nativePath 属性设置明确的路径。例如，在 Windows 计算机中运行以下代码，可以设置 File 对象以使其指向 C: 驱动器的 AIR Test 子目录中的 test.txt 文件：

```
var file = new air.File();
file.nativePath = "C:/AIR Test/test.txt";
```

还可以将此路径传递到 File() 构造函数，如下所示：

```
var file = new air.File("C:/AIR Test/test.txt");
```

请使用正斜杠 (/) 字符作为 nativePath 属性的路径分隔符。在 Windows 上，还可以使用反斜杠 (\) 字符，但这会导致应用程序无法跨平台工作。

有关详细信息，请参阅第 167 页的“[修改文件路径](#)”。

枚举目录中的文件

可以使用 File 对象的 getDirectoryListing() 方法获取指向位于某目录根级的文件和子目录的 File 对象数组。有关详细信息，请参阅第 171 页的“[枚举目录](#)”。

让用户浏览以选择文件

File 类包括以下方法，它们表示系统对话框，在该对话框中用户可以选择要分配给对象的文件：

- browseForOpen()
- browseForSave()
- browseForOpenMultiple()

这些方法均为异步方法。当用户选择文件（使用 browseForSave() 方法时为选择目标路径）时，browseForOpen() 和 browseForSave() 方法将调度 select 事件。对于 browseForOpen() 和 browseForSave() 方法，在进行选择后目标 File 对象将指向所选的文件。当用户选择多个文件时，browseForOpenMultiple() 方法调度一个 selectMultiple 事件。selectMultiple 事件的类型是 FileListEvent，它具有一个 files 属性，该属性是一个 File 对象数组（指向所选的文件）。

例如，以下代码向用户显示“Open”对话框，在该对话框中用户可以选择文件：

```
var fileToOpen = air.File.documentsDirectory;
selectTextFile(fileToOpen);

function selectTextFile(root)
{
    var txtFilter = new air.FileFilter("Text", "*.as;*.css;*.html;*.txt;*.xml");
    root.browseForOpen("Open", new window.runtime.Array(txtFilter));
    root.addEventListener(air.Event.SELECT, fileSelected);
}

function fileSelected(event)
{
    trace(fileToOpen.nativePath);
}
```

当您调用浏览方法时，如果应用程序已打开了其他浏览器对话框，则运行时会引发一个错误异常。

修改文件路径

通过调用 resolvePath() 方法或通过修改对象的 nativePath 或 url 属性，还可以修改现有 File 对象的路径，如下例所示（在 Windows 中）：

```

file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
alert(file1.nativePath); // C:\Documents and Settings\userName\My Documents\AIR Test
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("../");
alert(file2.nativePath); // C:\Documents and Settings\userName
var file3 = air.File.documentsDirectory;
file3.nativePath += "/subdirectory";
alert(file3.nativePath); // C:\Documents and Settings\userName\My Documents\subdirectory
var file4 = new air.File();
file4.url = "file:///c:/AIR Test/test.txt";
alert(file4.nativePath); // C:\AIR Test\test.txt

```

使用 `nativePath` 属性时，请使用正斜杠 (/) 字符作为目录分隔符。在 Windows 上，还可以使用反斜杠 (\) 字符，但不应这样做，因为这会导致代码无法跨平台工作。

支持的 URL 方案

定义 `File` 对象的 `url` 属性时，可以使用以下任一 URL 方案：

URL 方案	说明
file	用于指定相对于文件系统根目录的路径。例如： <code>file:///c:/AIR Test/test.txt</code> URL 标准规定 <code>file</code> URL 采用 <code>file://<host>/<path></code> 形式。作为一个特例， <code><host></code> 可以是空字符串，此时将其解释为“从其解释该 URL 的计算机”。因此， <code>file</code> URL 通常具有三个斜杠 (///)。
app	用于指定相对于所安装应用程序的根目录（该目录包含所安装应用程序的 <code>application.xml</code> 文件）的路径。例如，以下路径指向所安装应用程序的 <code>images</code> 子目录： <code>app:/images</code>
app-storage	用于指定相对于应用程序存储目录的路径。对于每个安装的应用程序，AIR 定义了一个唯一的应用程序存储目录，此目录对于存储特定于该应用程序的数据很有用。例如，以下路径指向应用程序存储目录的 <code>settings</code> 子目录中的 <code>prefs.xml</code> 文件： <code>app-storage:/settings/prefs.xml</code>

查找两个文件之间的相对路径

可以使用 `getRelativePath()` 方法找出两个文件之间的相对路径：

```

var file1 = air.File.documentsDirectory
file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory
file2 = file2.resolvePath("AIR Test/bob/test.txt");

alert(file1.getRelativePath(file2)); // bob/test.txt

```

`getRelativePath()` 方法的第二个参数，即 `useDotDot` 参数，考虑到在结果中会返回 .. 语法来指示父目录：

```

var file1 = air.File.documentsDirectory;
file1 = file1.resolvePath("AIR Test");
var file2 = air.File.documentsDirectory;
file2 = file2.resolvePath("AIR Test/bob/test.txt");
var file3 = air.File.documentsDirectory;
file3 = file3.resolvePath("AIR Test/susan/test.txt");

alert(file2.getRelativePath(file1, true)); // ../..
alert(file3.getRelativePath(file2, true)); // ../../bob/test.txt

```

获取文件名的规范版本

文件名和路径名在 Windows 和 Mac OS 中不区分大小写。在以下示例中，两个 File 对象指向同一个文件：

```
File.documentsDirectory.resolvePath("test.txt");
File.documentsDirectory.resolvePath("TeSt.TxT");
```

不过，文档和目录名确实包括大小写。例如，以下代码假定在文档目录中有一个名为 AIR Test 的文件夹，如以下示例中所示：

```
var file = air.File.documentsDirectory;
file = file.resolvePath("AIR test");
trace(file.nativePath); // ... AIR test
file.canonicalize();
alert(file.nativePath); // ... AIR Test
```

canonicalize() 方法可转换 nativePath 对象，以对文件名或目录名使用正确的大小写。在区分大小写的文件系统（如 Linux）上，当多个文件的名称只有大小写不同时，canonicalize() 方法将调整路径以匹配最先找到的文件（以文件系统确定的顺序）。

在 Windows 中，还可以使用 canonicalize() 方法将短文件名（“8.3”名称）转换为长文件名，如下例所示：

```
var path = new air.File();
path.nativePath = "C:\\AIR~1";
path.canonicalize();
alert(path.nativePath); // C:\\AIR Test
```

使用包和符号链接

多种操作系统支持包文件和符号链接文件：

包 — 在 Mac OS 中，可以指定目录作为包，并且目录可以作为单个文件而非目录出现在 Mac OS Finder 中。

符号链接 — Mac OS、Linux 和 Windows Vista 支持符号链接。通过符号链接，文件可以指向磁盘上的另一个文件或目录。尽管符号链接与别名类似，不过它们并不相同。别名始终报告为文件（而不是目录），读取或写入别名或快捷方式从不影响它指向的原始文件或目录。另一方面，符号链接的行为则完全与它指向的文件或目录类似。可以将符号链接报告为文件或目录，并且读写符号链接影响的是符号链接所指向的文件或目录，而不影响其本身。此外，在 Windows 中，引用交接点（用于 NTFS 文件系统中）的 File 对象的 isSymbolicLink 属性设置为 true。

File 类包括 isPackage 和 isSymbolicLink 属性，用于检查 File 对象是否引用包或符号链接。

以下代码将遍历用户的桌面目录，列出不是包的子目录：

```
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isDirectory && !desktopNodes[i].isPackage)
    {
        air.trace(desktopNodes[i].name);
    }
}
```

以下代码将遍历用户的桌面目录，列出不是符号链接的文件和目录：

```
var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (!desktopNodes[i].isSymbolicLink)
    {
        air.trace(desktopNodes[i].name);
    }
}
```

canonicalize() 方法可更改符号链接的路径，使其指向该链接所引用的文件或目录。以下代码将遍历用户的桌面目录，报告由是符号链接的文件引用的路径：

```

var desktopNodes = air.File.desktopDirectory.getDirectoryListing();
for (i = 0; i < desktopNodes.length; i++)
{
    if (desktopNodes[i].isSymbolicLink)
    {
        var linkNode = desktopNodes[i];
        linkNode.canonicalize();
        air.trace(desktopNodes[i].name);
    }
}

```

确定卷上的可用空间

File 对象的 `spaceAvailable` 属性是 File 位置的可用空间（以字节为单位）。例如，以下代码检查应用程序存储目录中的可用空间：

```
air.trace(air.File.applicationStorageDirectory.spaceAvailable);
```

如果 File 对象引用一个目录，则 `spaceAvailable` 属性将指示可供文件使用的目录空间。如果 File 对象引用一个文件，则 `spaceAvailable` 属性将指示可供该文件使用的空间。如果 File 位置不存在，则 `spaceAvailable` 属性将设置为 0。如果 File 对象引用一个符号链接，则 `spaceAvailable` 属性将设置为符号链接指向的位置的可用空间。

通常，目录或文件的可用空间与包含该目录或文件的卷上的可用空间相同。不过，可用空间与磁盘配额及每个目录的空间限制有关。

将文件或目录添加到卷中通常需要比文件的实际大小或目录中内容的实际大小更多的空间。例如，操作系统可能需要更多空间来存储索引信息。或者，所需的磁盘扇区可能会使用额外的空间。此外，可用空间是动态变化的。因此，您不能期望为文件存储分配报告的全部空间。有关写入文件系统的信息，请参阅第 175 页的“[读取和写入文件](#)”。

获取文件系统信息

File 类包括以下可提供有关文件系统的一些有用信息的静态属性：

属性	说明
<code>File.lineEnding</code>	主机操作系统使用的行结束字符序列。在 Mac OS 和 Linux 中，这是换行符。在 Windows 中，它是回车符后跟换行符。
<code>File.separator</code>	主机操作系统的路径组件分隔符。在 Mac OS 和 Linux 中，这是正斜杠 (/) 字符。在 Windows 中，它是反斜杠 (\) 字符。
<code>File.systemCharset</code>	主机操作系统为文件使用的默认编码。此属性与操作系统使用的字符集有关，与操作系统语言相对应。

Capabilities 类还包括有用的系统信息，在使用文件时这些信息可能很有用：

属性	说明
<code>Capabilities.hasIME</code>	指定播放器是在安装有 (true) 输入法编辑器 (IME) 的系统上运行，还是在未安装 (false) IME 的系统上运行。
<code>Capabilities.language</code>	指定运行播放器的系统的语言代码。
<code>Capabilities.os</code>	指定当前的操作系统。

注：使用 `Capabilities.os` 确定系统特性时，应务必小心。如果有更加具体的属性可用来确定系统特性，请使用该属性。否则，您可能面临所写代码无法在所有平台上正常工作的风险。例如，请考虑以下代码：

```

var separator:String;
if (Capablities.os.indexOf("Mac") > -1)
{
    separator = "/";
}
else
{
    separator = "\\";
}

```

此代码将导致 Linux 上出现问题。最好只使用 File.separator 属性。

使用目录

通过运行时提供的功能，可以使用本地文件系统上的目录。

有关创建指向目录的 File 对象的详细信息，请参阅第 164 页的“[将 File 对象指向目录](#)”。

创建目录

使用 File.createDirectory() 方法可以创建目录。例如，以下代码创建名为 AIR Test 的目录以作为用户主目录的子目录：

```
var dir = air.File.userDirectory.resolvePath("AIR Test");
dir.createDirectory();
```

如果该目录存在，则 createDirectory() 方法不执行任何操作。

另外，在某些模式中， FileStream 对象在打开文件时会创建目录。如果 FileStream() 构造函数的 fileMode 参数设置为 FileMode.APPEND 或 FileMode.WRITE，则在实例化 FileStream 实例时将创建缺少的目录。有关详细信息，请参阅第 175 页的“[读取和写入文件的工作流程](#)”。

创建临时目录

File 类包括 createTempDirectory() 方法，该方法在系统的临时目录文件夹中创建一个目录，如下例所示：

```
var temp = air.File.createTempDirectory();
```

createTempDirectory() 方法自动创建一个唯一的临时目录（您无需确定新的唯一位置）。

您可以使用临时目录暂时存储应用程序会话中使用的临时文件。请注意，有一个 createTempFile() 方法可以在系统临时目录中创建新的、唯一的临时文件。

您可能需要在关闭应用程序前删除该临时目录，因为它不会自动删除。

枚举目录

可以使用 File 对象的 getDirectoryListing() 方法或 getDirectoryListingAsync() 方法获取指向目录中的文件和子文件夹的 File 对象数组。

例如，以下代码将列出用户的文档目录的内容（无需检查子目录）：

```

var directory = air.File.documentsDirectory;
var contents = directory.getDirectoryListing();
for (i = 0; i < contents.length; i++)
{
    alert(contents[i].name, contents[i].size);
}
```

当使用该方法的异步版本时， directoryListing 事件对象具有一个 files 属性，该属性是与目录有关的 File 对象数组：

```

var directory = air.File.documentsDirectory;
directory.getDirectoryListingAsync();
directory.addEventListener(air.FileListEvent.DIRECTORY_LISTING, dirListHandler);

function dirListHandler(event)
{
    var contents = event.files;
    for (i = 0; i < contents.length; i++)
    {
        alert(contents[i].name, contents[i].size);
    }
}

```

复制和移动目录

您可以使用与复制或移动文件相同的方法复制或移动目录。例如，以下代码将以同步方式复制目录：

```

var sourceDir = air.File.documentsDirectory.resolvePath("AIR Test");
var resultDir = air.File.documentsDirectory.resolvePath("AIR Test Copy");
sourceDir.copyTo(resultDir);

```

当您将 `copyTo()` 方法的 `overwrite` 参数指定为 `true` 时，会删除现有目标目录中的所有文件和文件夹，并会将其替换为源目录中的文件和文件夹（即使源目录中不存在目标文件也是如此）。

指定为 `copyTo()` 方法的 `newLocation` 参数的目录将指定所得目录的路径；它不指定将包含所得目录的父目录。

有关详细信息，请参阅第 173 页的“[复制和移动文件](#)”。

删除目录内容

`File` 类包括一个 `deleteDirectory()` 方法和一个 `deleteDirectoryAsync()` 方法。这些方法删除目录，第一个方法以同步方式运行，第二个方法以异步方式运行（请参阅第 162 页的“[AIR 文件基础知识](#)”）。两个方法都包括一个 `deleteDirectoryContents` 参数（该参数取布尔值）；当此参数设置为 `true` 时（默认值为 `false`），调用该方法将删除非空目录；否则，只删除空目录。

例如，以下代码以同步方式删除用户的文档目录中的 `AIR Test` 子目录：

```

var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.deleteDirectory(true);

```

以下代码以异步方式删除用户的文档目录中的 `AIR Test` 子目录：

```

var directory = air.File.documentsDirectory.resolvePath("AIR Test");
directory.addEventListener(air.Event.COMPLETE, completeHandler)
directory.deleteDirectoryAsync(true);

function completeHandler(event) {
    alert("Deleted.")
}

```

此外，还包括 `moveToTrash()` 和 `moveToTrashAsync()` 方法，使用这些方法可以将目录移至系统垃圾桶。有关详细信息，请参阅第 174 页的“[将文件移到垃圾桶](#)”。

使用文件

使用 AIR 文件 API，您可以向应用程序中添加基本的文件交互功能。例如，您可以读取和写入文件、复制和删除文件等。由于您的应用程序可以访问本地文件系统，因此请参阅第 89 页的“[AIR 安全性](#)”（如果您还没有阅读该章节）。

注：您可以将文件类型与 AIR 应用程序相关联（以便双击时可以打开应用程序）。有关详细信息，请参阅第 286 页的“[管理文件关联](#)”。

获取文件信息

File 类包括以下属性，这些属性提供有关 File 对象指向的文件或目录的信息：

File 属性	说明
creationDate	本地磁盘上文件的创建日期。
creator	已废弃。请使用 extension 属性。(此属性报告文件的 Macintosh 创建者类型，此属性仅用于 Mac OS X 之前的 Mac OS 版本中。)
exists	引用的文件或目录是否存在。
extension	文件扩展名，它是最后一个句点（“.”）后面的名称部分（不包括句点）。如果文件名中没有句点，则 extension 为 null。
icon	包含为文件定义的图标的 Icon 对象。
isDirectory	File 对象引用是否为对目录的引用。
modificationDate	本地磁盘上文件或目录的上一次修改日期。
name	本地磁盘上文件或目录的名称（如果存在文件扩展名，则包括文件扩展名）。
nativePath	采用主机操作系统表示形式的完整路径。请参阅 第 163 页的“ File 对象的路径 ”。
parent	包含由 File 对象表示的文件夹或文件的文件夹。如果 File 对象引用的是文件系统根目录中的文件或目录，此属性为 null。
size	本地磁盘上文件的大小（以字节为单位）。
type	已废弃。请使用 extension 属性。（在 Macintosh 中，此属性是四个字符的文件类型，它仅用于 Mac OS X 之前的 Mac OS 版本中。）
url	文件或目录的 URL。请参阅 第 163 页的“ File 对象的路径 ”。

有关这些属性的详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#) (http://www.adobe.com/go/learn_air_html_jslr_cn) 中的 File 类条目。

复制和移动文件

File 类包括两种复制文件或目录的方法：copyTo() 和 copyToAsync()。File 类包括两种移动文件或目录的方法：moveTo() 和 moveToAsync()。copyTo() 和 moveTo() 方法以同步方式运行，copyToAsync() 和 moveToAsync() 方法以异步方式运行（请参阅 第 162 页的“[AIR 文件基础知识](#)”）。

若要复制或移动文件，请设置两个 File 对象。一个对象指向要复制或移动的文件，它是调用复制或移动方法的对象；另一个对象指向目标（结果）路径。

以下代码将 test.txt 文件从用户的文档目录的 AIR Test 子目录复制到同一目录中名为 copy.txt 的文件：

```
var original = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var newFile = air.File.documentsDirectory.resolvePath("AIR Test/copy.txt");
original.copyTo(newFile, true);
```

在此例中，copyTo() 方法的 overwrite 参数（第二个参数）的值设置为 true。通过将此值设置为 true，可以覆盖现有目标文件。此参数是可选的。如果您将它设置为 false（默认值），则当目标文件存在时该操作调度一个 IOErrorEvent 事件（文件没有复制）。

复制和移动方法的“异步”版本以异步方式运行。使用 addEventListener() 方法监视任务完成或错误条件，如以下代码所示：

```

var original = air.File.documentsDirectory;
original = original.resolvePath("AIR Test/test.txt");

var destination = air.File.documentsDirectory;
destination = destination.resolvePath("AIR Test 2/copy.txt");

original.addEventListener(air.Event.COMPLETE, fileMoveCompleteHandler);
original.addEventListener(air.IOErrorEvent.IO_ERROR, fileMoveIOErrorHandler);
original.moveToAsync(destination);

function fileMoveCompleteHandler(event) {
    alert(event.target); // [object File]
}
function fileMoveIOErrorHandler(event) {
    alert("I/O Error.");
}

```

File 类还包括 `File.moveToTrash()` 和 `File.moveToTrashAsync()` 方法，这些方法可将文件或目录移至系统垃圾桶。

删除文件

File 类包括一个 `deleteFile()` 方法和一个 `deleteFileAsync()` 方法。这些方法删除文件，第一个方法以同步方式运行，第二个方法以异步方式运行（请参阅第 162 页的“[AIR 文件基础知识](#)”）。

例如，以下代码以同步方式删除用户的文档目录中的 `test.txt` 文件：

```
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.deleteFile();
```

以下代码以异步方式删除用户的文档目录中的 `test.txt` 文件：

```
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, completeHandler)
file.deleteFileAsync();

function completeHandler(event) {
    alert("Deleted.")
}
```

此外，还包括 `moveToTrash()` 和 `moveToTrashAsync` 方法，使用这些方法可以将文件或目录移至系统垃圾桶。有关详细信息，请参阅第 174 页的“[将文件移到垃圾桶](#)”。

将文件移到垃圾桶

File 类包括一个 `moveToTrash()` 方法和一个 `moveToTrashAsync()` 方法。这些方法将文件或目录发送到系统垃圾桶，第一个方法以同步方式运行，第二个方法以异步方式运行（请参阅第 162 页的“[AIR 文件基础知识](#)”）。

例如，以下代码以同步方式将用户的文档目录中的 `test.txt` 文件移到系统垃圾桶：

```
var file = air.File.documentsDirectory.resolvePath("test.txt");
file.moveToTrash();
```

创建临时文件

File 类包括 `createTempFile()` 方法，该方法在系统的临时目录文件夹中创建一个文件，如以下示例中所示：

```
var temp = air.File.createTempFile();
```

`createTempFile()` 方法自动创建一个唯一的临时文件（您无需确定新的唯一位置）。

您可以使用临时文件暂时存储应用程序会话中使用的信息。请注意，还有一个 `createTempDirectory()` 方法，可以在系统临时目录中创建一个唯一的临时目录。

您可能需要在关闭应用程序前删除该临时文件，因为它不会自动删除。

读取和写入文件

AIR 应用程序可以使用 `FileStream` 类读取和写入文件系统。

读取和写入文件的工作流程

读取和写入文件的工作流程如下所示。

初始化指向路径的 `File` 对象。

此路径是您要使用的文件（或以后要创建的文件）的路径。

```
var file = air.File.documentsDirectory;
file = file.resolvePath("AIR Test/testFile.txt");
```

此例使用 `File` 对象的 `File.documentsDirectory` 属性和 `resolvePath()` 方法来初始化 `File` 对象。不过，有许多其他方式可以将 `File` 对象指向文件。有关详细信息，请参阅第 166 页的“[将 File 对象指向文件](#)”。

初始化 `FileStream` 对象。

调用 `FileStream` 对象的 `open()` 方法或 `openAsync()` 方法。

具体调用哪个方法取决于您希望是以同步还是异步方式打开文件。使用对象作为打开方法的 `file` 参数。对于 `fileMode` 参数，请指定 `FileMode` 类中的一个常量，以指定使用文件的方式。

例如，以下代码将初始化一个 `FileStream` 对象，该对象用于创建一个文件并覆盖所有现有数据：

```
var fileStream = new air.FileStream();
fileStream.open(file, air.FileMode.WRITE);
```

有关详细信息，请参阅第 176 页的“[初始化 FileStream 对象以及打开和关闭文件](#)”和第 176 页的“[FileStream 打开模式](#)”。

如果您以异步方式打开了文件（使用 `openAsync()` 方法），请为 `FileStream` 对象添加并设置事件侦听器。

这些事件侦听器方法响应在多种情况下（如从文件中读取数据、遇到 I/O 错误或要写入的所有数据均已写入）由 `FileStream` 对象调度的事件。

有关详细信息，请参阅第 180 页的“[异步编程和以异步方式打开的 FileStream 对象所生成的事件](#)”。

根据需要包含用于读取和写入数据的代码。

`FileStream` 类中有许多与读取和写入相关的方法。（它们都以“read”或“write”开头。）具体选择使用哪个方法读取或写入数据取决于目标文件中数据的格式。

例如，如果目标文件中的数据为 UTF 编码的文本，则可以使用 `readUTFBytes()` 和 `writeUTFBytes()` 方法。如果希望将数据作为字节数组处理，则可以使用 `readByte()`、`readBytes()`、`writeByte()` 和 `writeBytes()` 方法。有关详细信息，请参阅第 180 页的“[数据格式以及选择要使用的读取和写入方法](#)”。

如果以异步方式打开了文件，请确保在调用读取方法前有足够的可用数据。有关详细信息，请参阅第 178 页的“[读取缓冲区和 FileStream 对象的 bytesAvailable 属性](#)”。

写入文件之前，如果要检查可用磁盘空间量，您可以检查 `File` 对象的 `spaceAvailable` 属性。有关详细信息，请参阅第 170 页的“[确定卷上的可用空间](#)”。

当您处理完文件后，请调用 **FileStream** 对象的 **close()** 方法。
这将使其他应用程序可以使用该文件。

有关详细信息，请参阅第 176 页的“[初始化 FileStream 对象以及打开和关闭文件](#)”。

若要查看使用 **FileStream** 类读取和写入文件的范例应用程序，请参阅 Adobe AIR 开发人员中心上的以下文章：

- [构建文本文件编辑器](#)
- [从 XML 首选参数文件中读取和写入](#)

使用 **FileStream** 对象

FileStream 类定义打开、读取和写入文件的方法。

FileStream 打开模式

FileStream 对象的 **open()** 和 **openAsync()** 方法都包括 **fileMode** 参数，该参数定义文件流的一些属性，其中包括以下属性：

- 从文件读取的能力
- 写入文件的能力
- 数据是否始终追加到文件的结尾（当写入时）
- 当文件不存在时（以及当文件的父目录不存在时）执行哪些操作

以下是各种文件模式（可以指定这些模式作为 **open()** 和 **openAsync()** 方法的 **fileMode** 参数）：

文件模式	说明
 FileMode.READ	指定只能打开文件进行读取。
 FileMode.WRITE	指定打开文件进行写入。如果文件不存在，则在打开 FileStream 对象时创建它。如果文件存在，则删除所有现有数据。
 FileMode.APPEND	指定打开文件进行追加。如果文件不存在，则创建它。如果文件存在，不覆盖现有数据，所有写入操作都从文件结尾开始。
 FileMode.UPDATE	指定打开文件进行读取和写入。如果文件不存在，则创建它。如果想对文件进行随机读取 / 写入访问，可以指定此模式。您可以从文件中的任何位置读取，当写入文件时，只有写入的字节覆盖现有字节（所有其他字节保持不变）。

初始化 **FileStream** 对象以及打开和关闭文件

当您打开 **FileStream** 对象后，即可使用该对象从文件读取数据和向文件写入数据。通过将 **File** 对象传递到 **FileStream** 对象的 **open()** 或 **openAsync()** 方法，可以打开 **FileStream** 对象：

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.READ);
```

fileMode 参数（**open()** 和 **openAsync()** 方法的第二个参数）指定打开文件的模式： **read**、**write**、**append** 或 **update**。有关详细信息，请参阅上一部分 第 176 页的“[FileStream 打开模式](#)”。

如果使用 **openAsync()** 方法打开文件进行异步文件操作，请设置事件侦听器来处理异步事件：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completeHandler);
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(air.IOErrorEvent.IOERROR, errorHandler);
myFileStream.open(myFile, air FileMode.READ);

function completeHandler(event) {
    // ...
}

function progressHandler(event) {
    // ...
}

function errorHandler(event) {
    // ...
}
```

打开文件进行同步操作还是异步操作取决于使用 `open()` 方法还是 `openAsync()` 方法。有关详细信息，请参阅第 162 页的“[AIR 文件基础知识](#)”。

如果您在 `FileStream` 对象的打开方法中将 `fileMode` 参数设置为 `FileMode.READ` 或 `FileMode.UPDATE`，则当打开 `FileStream` 对象后数据将立即读入读取缓冲区中。有关详细信息，请参阅第 178 页的“[读取缓冲区和 FileStream 对象的 bytesAvailable 属性](#)”。

可以调用 `FileStream` 对象的 `close()` 方法关闭关联的文件，从而使该文件可供其它应用程序使用。

FileStream 对象的 position 属性

`FileStream` 对象的 `position` 属性确定下一个读取或写入方法读取或写入数据的位置。

执行读取或写入操作之前，请将 `position` 属性设置为文件中的任何有效位置。

例如，以下代码在文件的位置 8 处写入字符串 "hello"（采用 UTF 编码）：

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air FileMode.UPDATE);
myFileStream.position = 8;
myFileStream.writeUTFBytes("hello");
```

当您首次打开 `FileStream` 对象时，`position` 属性设置为 0。

执行读取操作之前，`position` 的值必须至少为 0 并且小于文件中的字节数（即文件中的现有位置）。

只有在以下情况下才修改 `position` 属性的值：

- 当您显式设置 `position` 属性时。
- 当您调用读取方法时。
- 当您调用写入方法时。

当您调用 `FileStream` 对象的读取或写入方法时，`position` 属性值立即增加您读取或写入的字节数。根据您使用的读取方法，`position` 属性可以增加您指定读取的字节数，也可以增加可用的字节数。当您随后调用读取或写入方法时，它从新的位置开始读取或写入。

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air.FileMode.UPDATE);
myFileStream.position = 4000;
alert(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
alert(myFileStream.position); // 4200
```

不过，有一个例外：对于以 `append` 模式打开的 `FileStream`，调用写入方法后 `position` 属性不变。（在 `append` 模式中，数据始终写入文件的结尾，而与 `position` 属性的值无关。）

对于打开进行异步操作的文件，在下一行代码执行之前不会完成写入操作。不过，您可以连续调用多个异步方法，运行时会按顺序执行它们：

```
var myFile = air.File.documentsDirectory;
myFile = myFile.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.openAsync(myFile, air.FileMode.WRITE);
myFileStream.writeUTFBytes("hello");
myFileStream.writeUTFBytes("world");
myFileStream.addEventListener(air.Event.CLOSE, closeHandler);
myFileStream.close();
air.trace("started.");

closeHandler(event:Event):void
{
    trace("finished.");
}
```

此代码的跟踪输出如下所示：

```
started.  
finished.
```

您可以在调用读取或写入方法后立即（或在任何时间）指定 `position` 值，下一个读取或写入操作将从该位置开始执行。例如，请注意以下代码在调用 `writeBytes()` 操作后立即设置 `position` 属性，并且即使写入操作完成后也会将 `position` 设置为该值（300）：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.openAsync(myFile, air.FileMode.UPDATE);
myFileStream.position = 4000;
air.trace(myFileStream.position); // 4000
myFileStream.writeBytes(myByteArray, 0, 200);
myFileStream.position = 300;
air.trace(myFileStream.position); // 300
```

读取缓冲区和 `FileStream` 对象的 `bytesAvailable` 属性

当打开具有读取功能的 `FileStream` 对象时（在该对象中，`open()` 或 `openAsync()` 方法的 `fileMode` 参数设置为 `READ` 或 `UPDATE`），运行时将数据存储在内部缓冲区中。打开文件后，`FileStream` 对象就开始将数据读入缓冲区中（通过调用 `FileStream` 对象的 `open()` 或 `openAsync()` 方法）。

对于打开执行同步操作的文件（使用 `open()` 方法），始终可以设置 `position` 指针指向任何有效位置（在文件的范围内），并开始读取任何数量的数据（在文件的范围内），如以下代码所示（其中假定该文件至少包含 100 个字节）：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air FileMode.READ);
myFileStream.position = 10;
myFileStream.readBytes(myByteArray, 0, 20);
myFileStream.position = 89;
myFileStream.readBytes(myByteArray, 0, 10);
```

无论打开文件进行同步操作还是异步操作，读取方法始终从由 `bytesAvailable` 属性表示的“可用”字节读取。当以同步方式读取时，任何时间文件的所有字节都是可用的。当以异步方式读取时，在由 `position` 事件指示的一系列异步缓冲区填充数据中，从 `progress` 属性指定的位置开始的字节是可用的。

对于打开进行同步操作的文件，`bytesAvailable` 属性始终设置为表示从 `position` 属性到文件结尾的字节数（文件中所有字节始终是可以读取的）。

对于打开进行异步操作的文件，您需要确保在调用读取方法之前读取缓冲区已包含了足够的数据。对于以异步方式打开的文件，随着读取操作的进行，文件中从读取操作开始时指定的 `position` 开始的数据将添加到缓冲区，每读取一个字节 `bytesAvailable` 属性增加一。`bytesAvailable` 属性指示从 `position` 属性指定的位置的字节开始到缓冲区结尾的可用字节数。`FileStream` 对象定期发送一个 `progress` 事件。

对于以异步方式打开的文件，随着数据在读取缓冲区中可用，`FileStream` 对象定期调度 `progress` 事件。例如，当数据读取到缓冲区时，以下代码将把该数据读取到 `ByteArray` 对象 `bytes` 中：

```
var bytes = new air.ByteArray();
var myFile = new air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, air FileMode.READ);

function progressHandler(event)
{
    myFileStream.readBytes(bytes, myFileStream.position, myFileStream.bytesAvailable);
}
```

对于以异步方式打开的文件，只能读取读取缓冲区中的数据。而且，当您读取数据后，它即从读取缓冲区中删除。对于读取操作，您需要确保在调用读取操作之前数据在读取缓冲区中存在。例如，以下代码读取文件中从位置 4000 开始的 8000 个字节：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air FileMode.READ);
myFileStream.position = 4000;

var str = "";

function progressHandler(event)
{
    if (myFileStream.bytesAvailable > 8000 )
    {
        str += myFileStream.readMultiByte(8000, "iso-8859-1");
    }
}
```

在写入操作过程中，`FileStream` 对象不会将数据读入读取缓冲区中。当写入操作完成后（写入缓冲区中所有数据都已写入文件），`FileStream` 对象启动一个新的读取缓冲区（假定关联的 `FileStream` 对象已打开并具有读取功能），并开始将从 `position` 属性指定的位置开始的数据读入读取缓冲区中。`position` 属性可以是写入的最后一个字节的位置，也可以是其他位置（如果在写入操作后用户为 `position` 对象指定了其他值）。

异步编程和以异步方式打开的 FileStream 对象所生成的事件

当以异步方式打开文件时（使用 `openAsync()` 方法），读取和写入文件是以异步方式执行的。在将数据读入读取缓冲区以及写入输出数据时，可以执行其他 ActionScript 代码。

这表示您需要注册由以异步方式打开的 `FileStream` 对象所生成的事件。

通过注册 `progress` 事件，当有新数据可供使用时您可以收到通知，如以下代码中所示：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.ProgressEvent.PROGRESS, progressHandler);
myFileStream.openAsync(myFile, air FileMode.READ);
var str = "";

function progressHandler(event)
{
    str += myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

通过注册 `complete` 事件，您可以读取全部数据，如以下代码中所示：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air FileMode.READ);
var str = "";
function completeHandler(event)
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

就像输入数据将存储到缓冲区中以便可以执行异步操作一样，您在异步流上写入的数据也将存储到缓冲区中，然后以异步方式写入文件。随着数据写入文件，`FileStream` 对象将定期调度一个 `OutputProgressEvent` 对象。`OutputProgressEvent` 对象包括一个 `bytesPending` 属性，该属性设置为剩余的要写入的字节数。您可以注册 `outputProgress` 事件，以便当此缓冲区实际写入文件时收到通知，或者为了显示进度对话框。不过，通常情况下不需要这样做。具体而言，可以调用 `close()` 方法，而不考虑未写入的字节。`FileStream` 对象将继续写入数据，当最后一个字节写入文件并且基础文件关闭后将传递 `close` 事件。

数据格式以及选择要使用的读取和写入方法

每个文件是磁盘上的一组字节。在 ActionScript 中，文件中的数据始终可以表示为 `ByteArray`。例如，以下代码将文件中的数据读入到名为 `bytes` 的 `ByteArray` 对象中：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completeHandler);
myFileStream.openAsync(myFile, air FileMode.READ);
var bytes = new air.ByteArray();

function completeHandler(event)
{
    myFileStream.readBytes(bytes, 0, myFileStream.bytesAvailable);
}
```

同样，以下代码将名为 `bytes` 的 `ByteArray` 中的数据写入文件：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.open(myFile, air FileMode.WRITE);
myFileStream.writeBytes(bytes, 0, bytes.length);
```

不过，通常您不希望在 ActionScript `ByteArray` 对象中存储数据，并且数据文件通常是指定的文件格式。

例如，文件中的数据可能是文本文件格式，您可能希望在 `String` 对象中表示这种数据。

因此， FileStream 类包括读取和写入方法，用于读取和写入除 `ByteArray` 对象以外的类型的数据。例如，使用 `readMultiByte()` 方法可以从文件中读取数据，并将其存储到字符串中，如以下代码所示：

```
var myFile = air.File.documentsDirectory.resolvePath("AIR Test/test.txt");
var myFileStream = new air.FileStream();
myFileStream.addEventListener(air.Event.COMPLETE, completed);
myFileStream.openAsync(myFile, air FileMode.READ);
var str = "";

function completeHandler(event)
{
    str = myFileStream.readMultiByte(myFileStream.bytesAvailable, "iso-8859-1");
}
```

`readMultiByte()` 方法的第二个参数指定 ActionScript 用来解释数据的文本格式（在示例中是“iso-8859-1”）。ActionScript 支持常用的字符集编码，《ActionScript 3.0 语言参考》中（请参阅[支持的字符集](#)，网址为 http://help.adobe.com/zh_CN/AS3LCR/Flash_10.0/charset-codes.html）列出了这些编码。

`FileStream` 类还包括 `readUTFBytes()` 方法，该方法使用 UTF-8 字符集将读取缓冲区中的数据读入一个字符串中。由于 UTF-8 字符集中的字符长度可变，因此不要在响应 `progress` 事件的方法中使用 `readUTFBytes()`，因为读取缓冲区结尾的数据可能表示不完整的字符。（将 `readMultiByte()` 方法用于可变长度字符编码时，需同样遵循上述要求。）因此，当 `FileStream` 对象调度 `complete` 事件时，应读取整个数据集。

还有类似的写入方法 `writeMultiByte()` 和 `writeUTFBytes()` 可用于 `String` 对象和文本文件。

`readUTF()` 和 `writeUTF()` 方法（请不要与 `readUTFBytes()` 和 `writeUTFBytes()` 相混淆）也可以从文件读取文本数据和将文本数据写入文件，不过它们假定文本数据前面有指定文本数据长度的数据（此方式在标准文本文件中不常见）。

有些 UTF 编码的文本文件以“UTF-BOM”（字节顺序标记）字符开头，该字符定义字节顺序以及编码格式（如 UTF-16 或 UTF-32）。

有关读取和写入文本文件的示例，请参阅第 182 页的“[示例：将 XML 文件读取到 XML 对象中](#)”。

使用 `readObject()` 和 `writeObject()` 可以很方便地存储和检索复杂 ActionScript 对象的数据。数据采用 AMF (ActionScript Message Format) 编码。此格式是 ActionScript 的专有格式。除 AIR、Flash Player、Flash Media Server 以及 Flex Data Services 之外的其他应用程序没有可以与此格式的数据一起使用的内置 API。

还有其它一些读取和写入方法（如 `readDouble()` 和 `writeDouble()`）。不过，如果您使用这些方法，请确保文件格式与这些方法定义的数据的格式相匹配。

文件格式通常比简单文本格式复杂。例如，MP3 文件包括压缩的数据，这些数据只能使用特定于 MP3 文件的解压缩和解码算法解释。MP3 文件还可能包括 ID3 标签，这些标签包含有关文件的元标签信息（如歌曲的标题和艺术家）。ID3 格式有多种版本，不过最简单的版本（ID3 第 1 版）在第 183 页的“[示例：使用随机访问读取和写入数据](#)”部分中进行了介绍。

其他文件格式（用于图像、数据库、应用程序文档等）具有不同的结构，若要在 ActionScript 中使用这些格式的数据，必须了解数据的构造方式。

使用 `load()` 和 `save()` 方法

Flash Player 10 向 `FileReference` 类添加了 `load()` 和 `save()` 方法。AIR 1.5 中也有这些方法，并且 `File` 类从 `FileReference` 类继承这些方法。这些方法旨在为用户提供一种在 Flash Player 中加载和保存文件数据的安全方法。但是，AIR 应用程序还可以使用这些方法作为一种异步加载和保存文件的简便方式。

例如，以下代码将字符串保存到文本文件：

```
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
var str = "Hello.";
file.addEventListener(air.Event.COMPLETE, fileSaved);
file.save(str);
function fileSaved(event)
{
    air.trace("Done.");
}
```

`save()` 方法的 `data` 参数可以采用 `String` 或 `ByteArray` 值。当参数为 `String` 值时，该方法将文件保存为 UTF-8 编码的文本文件。

执行此代码示例时，应用程序将显示一个对话框，用户在该对话框中选择所保存文件的目标。

以下代码从 UTF-8 编码的文本文件加载字符串：

```
var file = air.File.applicationStorageDirectory.resolvePath("test.txt");
file.addEventListener(air.Event.COMPLETE, loaded);
file.load();
var str;
function loaded(event)
{
    var bytes = file.data;
    str = bytes.readUTFBytes(bytes.length);
    air.trace(str);
}
```

`FileStream` 类所提供的功能要多于 `load()` 和 `save()` 方法：

- 借助 `FileStream` 类，既可以同步读写数据，也可以异步读写数据。
- 使用 `FileStream` 类可以用增量方式写入文件。
- 使用 `FileStream` 类可以打开文件进行随机访问（读写文件的任意部分）。
- 使用 `FileStream` 类可以指定对文件具有的访问权限的类型，具体途径是设置 `open()` 或 `openAsync()` 方法的 `fileMode` 参数。
- 通过 `FileStream`，不用向用户显示“打开”或“保存”对话框即可将数据保存到文件。
- 用 `FileStream` 类读取数据时可以直接使用字节数组之外的类型。

示例：将 XML 文件读取到 XML 对象中

以下示例演示如何读取和写入包含 XML 数据的文本文件。

若要从文件读取，请初始化 `File` 和 `FileStream` 对象，调用 `FileStream` 的 `readUTFBytes()` 方法，然后将字符串转换为 XML 对象：

```
var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream = new air.FileStream();
fileStream.open(file, air.FileMode.READ);
var prefsXML = fileStream.readUTFBytes(fileStream.bytesAvailable);
fileStream.close();
```

同样，将数据写入文件也很容易，比如设置适当的 `File` 和 `FileStream` 对象，然后调用 `FileStream` 对象的写入方法。将 XML 数据的字符串版本传递到写入方法，如以下代码中所示：

```

var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
fileStream = new air.FileStream();
fileStream.open(file, air.FileMode.WRITE);

var outputString = '<?xml version="1.0" encoding="utf-8"?>\n';
outputString += '<prefs><autoSave>true</autoSave></prefs>'

fileStream.writeUTFBytes(outputString);
fileStream.close();

```

这些示例使用 `readUTFBytes()` 和 `writeUTFBytes()` 方法，这是因为它们假定文件采用 UTF-8 格式。如果不是此格式，您可能需要使用其他方法（请参阅第 180 页的“[数据格式以及选择要使用的读取和写入方法](#)”）。

前面的示例使用为进行同步操作而打开的 `FileStream` 对象。您还可以打开文件进行异步操作（这依赖于事件侦听器函数以响应事件）。例如，以下代码演示如何以异步方式读取 XML 文件：

```

var file = air.File.documentsDirectory.resolvePath("AIR Test/preferences.xml");
var fileStream= new air.FileStream();
fileStream.addEventListener(air.Event.COMPLETE, processXMLData);
fileStream.openAsync(file, air.FileMode.READ);
var prefsXML;

function processXMLData(event)
{
    var xmlString = fileStream.readUTFBytes(fileStream.bytesAvailable);
    prefsXML = domParser.parseFromString(xmlString, "text/xml");
    fileStream.close();
}

```

将整个文件读入读取缓冲区时（当 `FileStream` 对象调度 `complete` 事件时），将调用 `processXMLData()` 方法。它调用 `readUTFBytes()` 方法获取所读取数据的字符串版本，然后基于该字符串创建一个 XML 对象 `prefsXML`。

若要查看演示这些功能的范例应用程序，请参阅[从 XML 首选参数文件中读取和写入](#)。[从 XML 首选参数文件中读取和写入](#)。

示例：使用随机访问读取和写入数据

MP3 文件可以包括 ID3 标签，这种标签是位于文件开头或结尾、包含用于标识录制情况的元数据的部分。ID3 标签格式本身具有不同的修订版本。此例描述如何使用“随机访问文件数据”从包含最简单的 ID3 格式（ID3 1.0 版）的 MP3 文件读取和写入，“随机访问文件数据”表示它在文件中任意位置进行读取和写入。

包含 ID3 第 1 版标签的 MP3 文件在文件结尾最后 128 个字节中包括 ID3 数据。

当访问文件进行随机读 / 写访问时，必须指定 `FileMode.UPDATE` 作为 `open()` 或 `openAsync()` 方法的 `fileMode` 参数。

```

var file = air.File.documentsDirectory.resolvePath("My Music/Sample ID3 v1.mp3");
var fileStr = new air.FileStream();
fileStr.open(file, air.FileMode.UPDATE);

```

这样可以读取和写入文件。

打开文件时，您可以设置 `position` 指针以指向文件结尾向前 128 个字节的位置：

```
fileStr.position = file.size - 128;
```

此代码将 `position` 属性设置为指向文件中的此位置，这是因为 ID3 1.0 版格式指定 ID3 标签数据存储在文件的最后 128 个字节中。该规范还包含以下内容：

- 标签的前 3 个字节包含字符串 "TAG"。
- 接下来的 30 个字符包含 MP3 曲目的标题，为字符串。
- 接下来的 30 个字符包含艺术家的姓名，为字符串。
- 接下来的 30 个字符包含唱片的名称，为字符串。

- 接下来的 4 个字符包含年份，为字符串。
- 接下来的 30 个字符包含注释，为字符串。
- 接下来的 1 个字节包含代码，指示曲目的流派。
- 所有文本数据都采用 ISO 8859-1 格式。

读入数据后（调度 complete 事件时），`id3TagRead()` 方法将检查数据：

```
function id3TagRead()
{
    if (fileStr.readMultiByte(3, "iso-8859-1").match(/tag/i))
    {
        var id3Title = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Artist = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Album = fileStr.readMultiByte(30, "iso-8859-1");
        var id3Year = fileStr.readMultiByte(4, "iso-8859-1");
        var id3Comment = fileStr.readMultiByte(30, "iso-8859-1");
        var id3GenreCode = fileStr.readByte().toString(10);
    }
}
```

您还可以对文件执行随机访问写入。例如，您可以解析 `id3Title` 变量以确保它的大小写正确（使用 `String` 类的方法），然后将修改后的名为 `newTitle` 的字符串写入文件，如下所示：

```
fileStr.position = file.length - 125;      // 128 - 3
fileStr.writeMultiByte(newTitle, "iso-8859-1");
```

为了遵守 ID3 第 1 版标准，`newTitle` 字符串的长度应为 30 个字符，结尾以字符代码 (`String.fromCharCode(0)(0)`) 填充。

第 22 章：拖放

使用拖放 API 中的类可以支持在用户界面上执行拖放动作。此处所说 的动作 是指由用户通过操作系统和应用程序执行的一种操作，表示要复制、移动或链接信息。当用户将对象拖出组件或应用程序时，执行的就是拖出 动作。当用户从组件或应用程序外部拖入对象时，执行的就是拖入 动作。

利用拖放 API，可以允许用户在应用程序之间以及在应用程序内的组件之间拖动数据。支持的传输格式包括：

- 位图
- 文件
- HTML 格式的文本
- 文本
- URL

有关拖放的其它在线信息

可以从以下源中查找有关使用拖放 API 的详细信息：

Adobe 开发人员中心文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR 拖放”）](#)

拖放基础知识

若要将数据拖入和拖出基于 HTML 的应用程序（或者拖入和拖出 HTMLLoader 中显示的 HTML），则可以使用 HTML 拖放事件。使用 HTML 拖放 API，可以拖到和拖出 HTML 内容中的 DOM 元素。

注：还可以通过侦听包含 HTML 内容的 HTMLLoader 对象上发生的事件，来使用 AIR NativeDragEvent 和 NativeDragManager API。但是，HTML API 更好地与 HTML DOM 集成到了一起，因而您可以控制默认行为。NativeDragEvent 和 NativeDragManager API 在基于 HTML 的应用程序中不常用，因此在[针对 HTML 开发人员的 Adobe AIR 语言参考](#) (http://www.adobe.com/go/learn_air_html_jslr_cn) 中未介绍它们。有关使用这些类的详细信息，请参考[使用 Adobe Flex 3 开发 AIR 应用程序](#) ([http://www.adobe.com/go/learn_air flex3_cn](http://www.adobe.com/go/learn_air	flex3_cn)) 以及 [Flex 3 语言参考](#) (http://www.adobe.com/go/learn_flex3_aslr_cn)。

默认的拖放行为

HTML 环境为涉及文本、图像和 URL 的拖放动作提供了默认行为。利用默认行为，始终都可以将这些类型的数据拖出元素。但是，只能将文本拖入元素，并且只能拖到页面的可编辑区域内的元素。在页面的可编辑区域之间或可编辑区域内部拖动文本时，默认行为会执行移动动作。从不可编辑的区域或应用程序外部向可编辑区域拖动文本时，默认行为则会执行复制动作。

可以通过自行处理拖放事件来覆盖默认行为。若要取消默认行为，必须调用为拖放事件调度的对象的 `preventDefault()` 方法。这样，您就可以根据需要将数据插入放置目标以及从拖动源中删除数据，以便执行所选的动作。

默认情况下，用户可以选择和拖动任何文本，并可以拖动图像和链接。可以使用 WebKit CSS 属性 `-webkit-user-select` 来控制可以如何选择任何 HTML 元素。例如，如果将 `-webkit-user-select` 设置为 `none`，则元素内容是不可选择的，因而也无法拖动。还可以使用 `-webkit-user-drag` CSS 属性控制能否将元素作为一个整体拖动。不过，元素的内容则单独处理。用户仍可以拖动文本的选定部分。有关详细信息，请参阅 第 71 页的“[CSS 扩展](#)”。

HTML 中的拖放事件

从中发起拖动操作的启动器元素调度的事件有：

事件	说明
<code>dragstart</code>	在用户启动拖动动作时调度。如有必要，此事件的处理函数可以通过调用事件对象的 <code>preventDefault()</code> 方法来阻止拖动。若要控制是否可以复制、链接或移动所拖动的数据，请设置 <code>effectAllowed</code> 属性。所选的文本、图像和链接由默认行为放置到剪贴板上，但您可以使用事件对象的 <code>dataTransfer</code> 属性为拖动动作设置其他数据。
<code>drag</code>	在执行拖动动作期间持续调度
<code>dragend</code>	在用户释放鼠标按键终止拖动动作时调度。

由拖动目标调度的事件有：

事件	说明
<code>dragover</code>	当拖动动作仍然在元素边界内时持续调度。此事件的处理函数应设置 <code>dataTransfer.dropEffect</code> 属性，以指示在用户释放鼠标时放置操作是否会导致复制、移动或链接动作。
<code>dragenter</code>	在拖动动作进入元素的边界内时调度。 如果在 <code>dragenter</code> 事件处理函数中更改 <code>dataTransfer</code> 对象的任何属性，则这些更改将很快被下一个 <code>dragover</code> 事件覆盖。另一方面，在 <code>dragenter</code> 与第一个 <code>dragover</code> 事件之间有一个短暂的延迟，如果设置了不同的属性，此延迟可能导致光标闪烁。在很多情况下，都可以对这两个事件使用相同的事件处理函数。
<code>dragleave</code>	在拖动动作离开元素边界时调度。
<code>drop</code>	在用户将数据放到元素上时调度。只能在此事件的处理函数内访问所拖动的数据。

为响应这些事件而调度的事件对象与鼠标事件类似。可以使用诸如 `(clientX, clientY)` 和 `(screenX, screenY)` 等鼠标事件属性来确定鼠标位置。

拖动事件对象最重要的属性是 `dataTransfer`，此属性包含被拖动的数据。`dataTransfer` 对象本身具有以下属性和方法：

属性或方法	说明
<code>effectAllowed</code>	拖动源允许的效果。通常情况下， <code>dragstart</code> 事件的处理函数设置此值。请参阅 HTML 中的拖动效果。
<code>dropEffect</code>	由目标或用户选择的效果。如果在 <code>dragover</code> 或 <code>dragenter</code> 事件处理函数中设置 <code>dropEffect</code> ，则 AIR 会更新鼠标光标，以指示在用户释放鼠标时出现的效果。如果允许的效果中没有一种效果与 <code>dropEffect</code> 设置匹配，则不允许放置，并显示不可用光标。如果尚未设置 <code>dropEffect</code> 来响应最新的 <code>dragover</code> 或 <code>dragenter</code> 事件，则用户可以使用标准的操作系统功能键从允许的效果中进行选择。 最终的效果由 <code>dragend</code> 调度的对象的 <code>dropEffect</code> 属性报告。如果用户通过在符合条件的目标外部释放鼠标放弃放置操作，则 <code>dropEffect</code> 将设置为 <code>none</code> 。
<code>types</code>	一个数组，其中包含了 <code>dataTransfer</code> 对象中存在的每种数据格式的 MIME 类型字符串。
<code>getData(mimeType)</code>	以 <code>mimeType</code> 参数指定的格式获取数据。 只有在响应 <code>drop</code> 事件时可以调用 <code>getData()</code> 方法。

属性或方法	说明
setData(mimeType)	以 mimeType 参数指定的格式向 dataTransfer 添加数据。通过对每种 MIME 类型调用 setData()，可以添加多种格式的数据。将清除由默认拖动行为放入 dataTransfer 对象的所有数据。 只有在响应 dragstart 事件时可以调用 setData() 方法。
clearData(mimeType)	清除采用 mimeType 参数指定的格式的所有数据。
setDragImage(image, offsetX, offsetY)	设置自定义拖动图像。只有在响应 dragstart 事件，且只有通过将 -webkit-user-drag CSS 样式设置为 element 来拖动整个 HTML 元素时，才能调用 setDragImage() 方法。image 参数可以是 JavaScript 元素或 Image 对象。

用于 HTML 拖放的 MIME 类型

与 HTML 拖放事件的 dataTransfer 对象一起使用的 MIME 类型包括：

数据格式	MIME 类型
文本	"text/plain"
HTML	"text/html"
URL	"text/uri-list"
位图	"image/x-vnd.adobe.air.bitmap"
文件列表	"application/x-vnd.adobe.air.file-list"

还可以使用其他 MIME 字符串，包括应用程序定义的字符串。但是，其他应用程序可能无法识别或使用所传输的数据。以所需格式向 dataTransfer 对象添加数据的工作由您负责完成。

重要说明：只有在应用程序沙箱中运行的代码才能访问放置的文件。如果试图在非应用程序沙箱内读取或设置 File 对象的任何属性，则会产生安全错误。有关详细信息，请参阅在非应用程序 HTML 沙箱中处理文件放置。

HTML 中的拖动效果

拖动动作的启动器可以通过在 dataTransfer.effectAllowed 事件的处理函数中设置 dragstart 属性，来限制允许的拖动效果。可以使用以下字符串值：

字符串值	说明
"none"	不允许任何拖动操作。
"copy"	将数据复制到目标位置，并保留原始数据的位置不变。
"link"	使用指向原始数据的链接与放置目标共享数据。
"move"	将数据复制到目标，并将其从原始位置删除。
"copyLink"	可以复制数据，也可链接数据。
"copyMove"	可以复制数据，也可移动数据。
"linkMove"	可以链接数据，也可移动数据。
"all"	可以复制数据、移动数据，也可链接数据。当您阻止默认行为时，All 是默认效果。

拖动动作的目标可以设置 `dataTransfer.dropEffect` 属性，以指示在用户完成放置后采取的动作。如果放置效果是允许的动作之一，则系统将显示相应的复制、移动或链接光标。如果不是，则系统将显示不可用光标。如果目标未设置放置效果，则用户可以使用功能键从允许的动作中进行选择。

同时在 `dragover` 和 `dragenter` 事件的处理函数中设置 `dropEffect` 值：

```
function doDragStart(event) {
    event.dataTransfer.setData("text/plain", "Text to drag");
    event.dataTransfer.effectAllowed = "copyMove";
}

function doDragOver(event) {
    event.dataTransfer.dropEffect = "copy";
}

function doDragEnter(event) {
    event.dataTransfer.dropEffect = "copy";
}
```

注：尽管您始终都应在 `dragenter` 的处理函数中设置 `dropEffect` 属性，但要注意，下一个 `dragover` 事件会将此属性重置为其默认值。设置 `dropEffect` 以响应这两个事件。

将数据拖出 HTML 元素

默认行为允许以拖动方式复制 HTML 页面中的大部分内容。可以使用 CSS 属性 `-webkit-user-select` 和 `-webkit-user-drag` 来控制允许拖动的内容。

在 `dragstart` 事件的处理函数中覆盖默认的拖出行为。调用事件对象的 `dataTransfer` 属性的 `setData()` 方法，使您自己的数据进入拖动动作。

若要指示在不依赖默认行为时源对象支持的拖动效果，请设置为 `dragstart` 事件调度的事件对象的 `dataTransfer.effectAllowed` 属性。您可以选择任意效果组合。例如，如果源元素既支持复制效果，也支持链接效果，则将此属性设置为 "copyLink"。

设置拖动的数据

使用 `dataTransfer` 属性在 `dragstart` 事件的处理函数中为拖动动作添加数据。使用 `dataTransfer.setData()` 方法将数据放到剪贴板上，同时传入 MIME 类型和要传输的数据。

例如，如果应用程序中有一个 ID 为 `imageOfGeorge` 的图像元素，则可以使用下面的 `dragstart` 事件处理函数。此示例以多种数据格式添加 `George` 照片的表示形式，从而增加了其他应用程序能够使用拖动的数据的可能性。

```
function dragStartHandler(event) {
    event.dataTransfer.effectAllowed = "copy";

    var dragImage = document.getElementById("imageOfGeorge");
    var dragFile = new air.File(dragImage.src);
    event.dataTransfer.setData("text/plain", "A picture of George");
    event.dataTransfer.setData("image/x-vnd.adobe.air.bitmap", dragImage);
    event.dataTransfer.setData("application/x-vnd.adobe.air.file-list",
        new Array(dragFile));
}
```

注：调用 `dataTransfer` 对象的 `setData()` 方法时，默认拖放行为不会添加任何数据。

将数据拖入 HTML 元素

默认行为只允许将文本拖入页面的可编辑区域中。可以通过在元素的开始标签中包含 `contenteditable` 属性，指定可以使元素及其子级可编辑。还可以通过将文档对象的 `designMode` 属性设置为 "on"，使整个文档可编辑。

可以通过处理可接受所拖动数据的任何元素的 `dragenter`、`dragover` 和 `drop` 事件，在页面上支持其它拖入行为。

允许拖入

若要处理拖入动作，必须先取消默认行为。侦听您要用作放置目标的任何 HTML 元素上发生的 `dragenter` 和 `dragover` 事件。在这些事件的处理函数中，调用所调度的事件对象的 `preventDefault()` 方法。如果取消默认行为，则会允许不可编辑的区域接收放置。

获取放置的数据

可以在 `ondrop` 事件的处理函数中访问放置的数据：

```
function doDrop(event) {
    droppedText = event.dataTransfer.getData("text/plain");
}
```

使用 `dataTransfer.getData()` 方法将数据读入到剪贴板上，同时传入要读取的数据格式的 MIME 类型。可以使用 `dataTransfer` 对象的 `types` 属性查明哪些数据格式可用。`types` 数组包含每种可用格式的 MIME 类型字符串。

取消 `dragenter` 或 `dragover` 事件中的默认行为后，须由您将放置的任何数据插入到其在文档中的正确位置。不存在可将鼠标位置转换成元素内的插入点的 API。由于存在这一限制，因而可能很难实现插入类型的拖动动作。

示例：覆盖默认的 HTML 拖入行为

此示例实现一个放置目标，此目标显示了一个表，表中显示放置的项目中可用的每种数据格式。

默认行为用于允许在应用程序内拖动文本、链接和图像。此示例覆盖用作放置目标的 `div` 元素的默认拖入行为。使不可编辑的内容能接受拖入动作的关键步骤，就是调用为 `dragenter` 和 `dragover` 事件调度的事件对象的 `preventDefault()` 方法。为响应 `drop` 事件，处理函数将传输的数据转换成 HTML 行元素，然后将该行插入表中以进行显示。

```
<html>
<head>
<title>Drag-and-drop</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    function init() {
        var target = document.getElementById('target');
        target.addEventListener("dragenter", dragEnterOverHandler);
        target.addEventListener("dragover", dragEnterOverHandler);
        target.addEventListener("drop", dropHandler);

        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
        source.addEventListener("dragend", dragEndHandler);

        emptyRow = document.getElementById("emptyTargetRow");
    }

    function dragStartHandler(event) {
        event.dataTransfer.effectAllowed = "copy";
    }
}
```

```
function dragEndHandler(event) {
    air.trace(event.type + ": " + event.dataTransfer.dropEffect);
}

function dragEnterOverHandler(event) {
    event.preventDefault();
}

var emptyRow;
function dropHandler(event) {
    for(var prop in event){
        air.trace(prop + " = " + event[prop]);
    }
    var row = document.createElement('tr');
    row.innerHTML = "<td>" + event.dataTransfer.getData("text/plain") + "</td> +
                    "<td>" + event.dataTransfer.getData("text/html") + "</td> +
                    "<td>" + event.dataTransfer.getData("text/uri-list") + "</td> +
                    "<td>" + event.dataTransfer.getData("application/x-vnd.adobe.air.file-list") +
                    "</td>";

    var imageCell = document.createElement('td');
    if((event.dataTransfer.types.toString()).search("image/x-vnd.adobe.bitmap") > -1){
        imageCell.appendChild(event.dataTransfer.getData("image/x-vnd.adobe.bitmap"));
    }
    row.appendChild(imageCell);
    var parent = emptyRow.parentNode;
    parent.insertBefore(row, emptyRow);
}
</script>
</head>
<body onLoad="init()" style="padding:5px">
<div>
    <h1>Source</h1>
    <p>Items to drag:</p>
    <ul id="source">
        <li>Plain text.</li>
        <li>HTML <b>formatted</b> text.</li>
        <li><a href="http://www.adobe.com">URL.</a></li>
        <li></li>
        <li style="-webkit-user-drag:none;">
            Uses "-webkit-user-drag:none" style.
        </li>
        <li style="-webkit-user-select:none;">
            Uses "-webkit-user-select:none" style.
        </li>
    </ul>
</div>
<div id="target" style="border-style:dashed;">
    <h1>Target</h1>
    <p>Drag items from the source list (or elsewhere).</p>
    <table id="displayTable" border="1">
        <tr><th>Plain text</th><th>Html text</th><th>URL</th><th>File list</th><th>Bitmap Data</th></tr>
        <tr
id="emptyTargetRow"><td>&ampnbsp</td><td>&ampnbsp</td><td>&ampnbsp</td><td>&ampnbsp</td><td>&ampnbsp</td></tr>
    </table>
</div>
</body>
</html>
```

在非应用程序 HTML 沙箱中处理文件放置

非应用程序内容无法访问在将文件拖入 AIR 应用程序时产生的 File 对象。也无法将这些 File 对象中的一个对象通过沙箱桥传递给应用程序内容。(必须在序列化期间访问对象属性。)但是,您仍可以通过侦听 HTMLLoader 对象上发生的 AIR nativeDragDrop 事件,在应用程序中放置文件。

通常,如果用户将文件放入承载非应用程序内容的框架中,则放置事件不会从子级传播到父级。但是,由于 HTMLLoader (AIR 应用程序中所有 HTML 内容的容器) 调度的事件不是 HTML 事件流的一部分,因此您仍可以在应用程序内容中接收放置事件。

为了接收文件放置事件,父级文档会使用 window.htmlLoader 提供的引用向 HTMLLoader 对象添加一个事件侦听器:

```
window.htmlLoader.addEventListener("nativeDragDrop",function(event){
    var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
    air.trace(filelist[0].url);
});
```

NativeDragEvent 对象在行为上与其 HTML 事件的对应对象相似,但部分属性和方法的名称不同。例如,HTML dataTransfer 属性是 AIR clipboard 属性。在[针对 HTML 开发人员的 Adobe AIR 语言参考](#)中未介绍 NativeDragEvent 和 NativeDragManager API。有关使用这些类的详细信息,请参考[使用 Adobe Flex 3 开发 AIR 应用程序](#)和[Flex 3 语言参考](#)。

下面的示例使用将子级页面加载到远程沙箱 (<http://localhost/>) 中的父级文档:父级侦听 HTMLLoader 对象上发生的 nativeDragDrop 事件,并输出文件 URL。

```
<html>
<head>
<title>Drag-and-drop in a remote sandbox</title>
<script language="javascript" type="text/javascript" src="AIRAliases.js"></script>
<script language="javascript">
    window.htmlLoader.addEventListener("nativeDragDrop",function(event){
        var filelist = event.clipboard.getData(air.ClipboardFormats.FILE_LIST_FORMAT);
        air.trace(filelist[0].url);
    });
</script>
</head>
<body>
    <iframe src="child.html"
            sandboxRoot="http://localhost/"
            documentRoot="app:/"
            frameBorder="0" width="100%" height="100%">
    </iframe>
</body>
</html>
```

子级文档必须通过在 HTML dragenter 和 dragover 事件处理函数中阻止 Event 对象的 preventDefault() 方法来提供有效的放置目标,否则无法发生放置事件。

```
<html>
<head>
    <title>Drag and drop target</title>
    <script language="javascript" type="text/javascript">
        function preventDefault(event){
            event.preventDefault();
        }
    </script>
</head>
<body ondragenter="preventDefault(event)" ondragover="preventDefault(event)">
<div>
    <h1>Drop Files Here</h1>
</div>
</body>
</html>
```

第 23 章：复制和粘贴

使用剪贴板 API 中的类将信息复制到系统剪贴板中或从中复制信息。可以与 Adobe® AIR® 应用程序之间实现传输的数据格式包括：

- 位图
- 文件
- 文本
- HTML 格式的文本
- RTF 数据
- URL 字符串
- 序列化对象
- 对象引用（仅在源应用程序内有效）

有关复制和粘贴的其它在线信息

可以从以下源中查找有关复制和粘贴的详细信息：

快速入门（**Adobe AIR** 开发人员中心）

语言参考

- 剪贴板
- [ClipboardFormats](#)
- [ClipboardTransferMode](#)

更多信息

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR 复制和粘贴”）](#)

复制和粘贴基础知识

复制和粘贴 API 包含以下类。

包	类
flash.desktop	<ul style="list-style-type: none"> • Clipboard • ClipboardFormats • ClipboardTransferMode <p>在下列类中定义了用于复制和粘贴 API 的常量:</p> <ul style="list-style-type: none"> • ClipboardFormats • ClipboardTransferMode

静态 `Clipboard.generalClipboard` 属性表示操作系统剪贴板。 `Clipboard` 类为从 `Clipboard` 对象读取数据或向其中写入数据提供了方法。也可以创建新的 `Clipboard` 对象以通过拖放 API 传输数据。

HTML 环境提供了用于复制和粘贴的备用 API。可以通过在应用程序安全沙箱中运行的代码中使用上述各 API，但只有 HTML API 可在非应用程序内容中使用。(请参阅 第 193 页的“[HTML 复制和粘贴](#)”。)

`HTMLLoader` 和 `TextField` 类用于实现一般复制和粘贴快捷键的默认行为。若要实现自定义组件的复制和粘贴快捷键行为，您可以直接侦听这些键击。您也可以使用本机菜单命令及等效键来间接响应键击。

可以在一个 `Clipboard` 对象中包含同一信息的不同表示形式，以使其他应用程序更易于理解和使用其中的数据。例如，图像可以以图像数据形式、序列化的 `Bitmap` 对象形式和文件形式包含在其中。以某种格式呈现数据的操作可以延迟，以便直到读取此格式的数据时才真正创建此格式。

注: 在 Linux 上，当关闭 AIR 应用程序时，剪贴板数据不会永久保存。

读取和写入系统剪贴板

若要读取操作系统剪贴板，请调用 `Clipboard.generalClipboard` 对象的 `getData()` 方法，并传递要读取的格式的名称:

```
if(air.Clipboard.generalClipboard.hasFormat("text/plain")){
    var text = air.Clipboard.generalClipboard.getData("text/plain");
}
```

若要写入剪贴板，请以一种或多种格式将数据添加到 `Clipboard.generalClipboard` 对象。任何同一格式的现有数据都将被自动覆盖。然而，建议在将新数据写入系统剪贴板之前清除系统剪贴板，这样可确保任何其他格式的无关数据也将删除。

```
var textToCopy = "Copy to clipboard.";
air.Clipboard.generalClipboard.clear();
air.Clipboard.generalClipboard.setData("text/plain", textToCopy, false);
```

注: 只有在应用程序安全沙箱中运行的代码可以直接访问系统剪贴板。在非应用程序 HTML 内容中，您只能通过由 HTML 复制或粘贴事件中某一事件调度的事件对象的 `clipboardData` 属性访问剪贴板。

HTML 复制和粘贴

HTML 环境自身提供针对复制和粘贴的事件组和默认行为。只有在应用程序安全沙箱中运行的代码才能通过 AIR `Clipboard.generalClipboard` 对象直接访问系统剪贴板。在非应用程序安全沙箱中运行的 JavaScript 代码可以通过因响应 HTML 文档中某元素调度的某个复制或粘贴事件而调度的事件对象来访问剪贴板。

复制和粘贴事件包括: `copy`、`cut` 和 `paste`。为这些事件调度的对象可通过 `clipboardData` 属性提供对剪贴板的访问。

默认行为

默认情况下，AIR 将复制选定的项目以响应复制命令，该命令可通过快捷键或上下文菜单生成。在可编辑区域内，AIR 将剪切文本以响应剪切命令，或将文本粘贴到光标位置或选定位置以响应粘贴命令。

若要阻止默认行为，事件处理函数可以调用被调度事件对象的 `preventDefault()` 方法。

使用事件对象的 `clipboardData` 属性

如果事件对象是因为某个复制或粘贴事件而被调度，则该事件对象的 `clipboardData` 属性允许您读取和写入剪贴板数据。

若要在处理复制或剪切事件时写入剪贴板，请使用 `clipboardData` 对象的 `setData()` 方法，并传入要复制的数据和 MIME 类型：

```
function customCopy(event) {
    event.clipboardData.setData("text/plain", "A copied string.");
}
```

若要访问被粘贴的数据，可以使用 `clipboardData` 对象的 `getData()` 方法，并传入数据格式的 MIME 类型。可用格式由 `types` 属性报告。

```
function customPaste(event) {
    var pastedData = event.clipboardData("text/plain");
}
```

只能在 `paste` 事件调度的事件对象中访问 `getData()` 方法和 `types` 属性。

下面的示例说明如何覆盖 HTML 页中默认的复制和粘贴行为。`copy` 事件处理函数将复制的文本设置为斜体并将其作为 HTML 文本复制到剪贴板。`cut` 事件处理函数将选定的数据复制到剪贴板并将其从文档中移除。`paste` 处理函数将剪贴板内容作为 HTML 插入并将插入内容的样式设置为粗体文本。

```
<html>
<head>
    <title>Copy and Paste</title>
    <script language="javascript" type="text/javascript">
        function onCopy(event){
            var selection = window.getSelection();
            event.clipboardData.setData("text/html","<i>" + selection + "</i>");
            event.preventDefault();
        }

        function onCut(event){
            var selection = window.getSelection();
            event.clipboardData.setData("text/html","<i>" + selection + "</i>");
            var range = selection.getRangeAt(0);
            range.extractContents();

            event.preventDefault();
        }

        function onPaste(event){
            var insertion = document.createElement("b");
            insertion.innerHTML = event.clipboardData.getData("text/html");
            var selection = window.getSelection();
            var range = selection.getRangeAt(0);
            range.insertNode(insertion);
            event.preventDefault();
        }
    </script>
</head>
<body onCopy="onCopy(event)"
      onPaste="onPaste(event)"
      onCut="onCut(event)">
<p>Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt.</p>
</body>
</html>
```

用于复制和粘贴的菜单命令和键击

复制和粘贴功能通常通过菜单命令和快捷键触发。在 OS X 中，操作系统将自动创建带有复制和粘贴命令的编辑菜单，但您必须将侦听器添加到这些菜单命令中以关联自己的复制和粘贴功能。在 Windows 中，您可以将本机编辑菜单添加到使用系统镶边的任何窗口中。（您也可以使用 DHTML 创建非本机菜单，但这超出了本次讨论的范畴。）

若要通过快捷键触发复制和粘贴命令，您可以将等效键分配给本机应用程序中的相应命令项或窗口菜单，也可以直接侦听键击。

通过菜单命令启动复制或粘贴操作

若要通过菜单命令触发复制或粘贴操作，您必须为调用处理函数的菜单项上的 select 事件添加侦听器。

在 HTML 内容中，可以使用 NativeApplication 编辑命令触发默认的复制和粘贴行为。例如，NativeApplication copy() 方法可向页面发送一个复制命令，就像按下键盘上的 CMD-C 或 CTRL-C 键一样。类似命令可用于剪切、粘贴和全选。以下示例将为可编辑的 HTML 文档创建一个编辑菜单：

```

<html>
<head>
    <title>Edit Menu</title>
    <script src="AIRAliases.js" type="text/javascript"></script>
    <script language="javascript" type="text/javascript">
        function init(){
            document.designMode = "On";
            addEditMenu();
        }

        function addEditMenu(){
            var menu = new air.NativeMenu
            var edit = menu.addSubmenu(new air.NativeMenu(), "Edit");

            var copy = edit.submenu.addItem(new air.NativeMenuItem("Copy"));
            var cut = edit.submenu.addItem(new air.NativeMenuItem("Cut"));
            var paste = edit.submenu.addItem(new air.NativeMenuItem("Paste"));
            var selectAll = edit.submenu.addItem(new air.NativeMenuItem("Select All"));

            copy.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.copy();
            });
            cut.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.cut();
            });
            paste.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.paste();
            });

            selectAll.addEventListener(air.Event.SELECT, function(){
                air.NativeApplication.nativeApplication.selectAll();
            });

            copy.keyEquivalent = "c";
            cut.keyEquivalent = "x";
            paste.keyEquivalent = "v";
            selectAll.keyEquivalent = "a";

            if(air.NativeWindow.supportsMenu){
                window.nativeWindow.menu = menu;
            } else if (air.NativeApplication.supportsMenu){
                air.NativeApplication.nativeApplication.menu = menu;
            }
        }
    </script>
</head>
<body onLoad="init()">
    <p>Neque porro quisquam est qui dolorem ipsum
       quia dolor sit amet, consectetur, adipisci velit.</p>
</body>
</html>

```

前面的示例替换了 Mac OS X 中的应用程序菜单，但您也可以通过查找现有项并向其添加事件侦听器来利用默认的“编辑”菜单。

查找 Mac OS X 中的默认菜单项

若要在 Mac OS X 上的应用程序菜单中查找默认的编辑菜单和特定的复制、剪切和粘贴命令项，您可以使用 NativeMenuItem 对象的 label 属性搜索整个菜单层次结构。例如，以下函数将使用某个名称并在菜单中查找具有匹配标签的项：

```

function findItemByName(menu, name, recurse) {
    var searchItem = null;
    for (var i = 0; i < menu.items.length; i++) {
        if (menu.items[i].label == name) {
            searchItem = menu.items[i];
            break;
        }
        if ((menu.items[i].submenu != null) && recurse) {
            searchItem = findItemByName(menu.items[i].submenu, name, recurse);
        }
        if (searchItem != null) { break; }
    }
    return searchItem;
}

```

您可以将 `reurse` 参数设置为 `true` 以便将子菜单包括在搜索范围之内，或将该参数设置为 `false` 以便只包括传入菜单。

通过键击启动复制或粘贴命令

如果应用程序使用本机窗口或应用程序菜单进行复制和粘贴，则可以将等效键添加到菜单项中以实现快捷键。

在 HTML 内容中，默认情况下会实现用于复制和粘贴命令的快捷键。不可能通过使用按键事件侦听器来捕获所有常用于复制和粘贴的键击。如果需要覆盖默认行为，较好的策略是侦听 `copy` 和 `paste` 事件本身。

剪贴板数据格式

剪贴板格式描述了 `Clipboard` 对象中放置的数据。AIR 可在 AIR 对象和系统剪贴板格式之间自动进行标准数据格式的转换。此外，使用应用程序定义的格式可以在 AIR 应用程序内或多个 AIR 应用程序之间传输应用程序对象。

`Clipboard` 对象可以包含采用不同格式的同一信息的多种表示形式。例如，表示 AIR 对象的 `Clipboard` 对象可以包括在同一应用程序中使用的引用格式、由另一 AIR 应用程序使用的序列化格式、由图像编辑器使用的位图格式以及文件列表格式，还可能具有延迟呈现功能以对 PNG 文件进行编码，以便将该对象的表示形式复制或拖动到文件系统。

标准数据格式

`ClipboardFormats` 类中提供了用于定义标准格式名称的常量：

常量	说明
TEXT_FORMAT	文本格式数据与 ActionScript String 类之间的转换。
HTML_FORMAT	带有 HTML 标记的文本。
RICH_TEXT_FORMAT	RTF 格式数据与 ActionScript ByteArray 类之间的转换。不会以任何方式对 RTF 标记进行解释或转换。
BITMAP_FORMAT	位图格式数据与 ActionScript BitmapData 类之间的转换。
FILE_LIST_FORMAT	文件列表数据与 ActionScript File 对象数组之间的转换。
URL_FORMAT	URL 格式数据与 ActionScript String 类之间的转换。

因响应 HTML 内容中的 `copy`、`cut` 或 `paste` 事件而复制和粘贴数据时，必须使用 MIME 类型而不是 `ClipboardFormat` 字符串。有效的数据 MIME 类型为：

MIME 类型	说明
文本	"text/plain"
URL	"text/uri-list"
位图	"image/x-vnd.adobe.air.bitmap"
文件列表	"application/x-vnd.adobe.air.file-list"

注：如果事件对象是因为 HTML 内容中的 clipboardData 事件而被调度，则无法从该事件对象的 paste 属性中获得 RTF 格式数据。

自定义数据格式

您可以使用应用程序定义的自定义格式将对象作为引用或序列化副本传输。引用只在同一 AIR 应用程序内有效。序列化对象可以在 Adobe AIR 应用程序之间传输，但只能用于序列化和取消序列化后仍然有效的对象。如果对象的属性为简单类型或可序列化对象，则通常可以将该对象序列化。

若要将序列化对象添加到 Clipboard 对象，请在调用 Clipboard.setData() 方法时将 serializable 参数设置为 true。格式名称可以是标准格式中的某一种或由应用程序定义的任意字符串。

传输模式

使用自定义数据格式将对象写入剪贴板后，可以从剪贴板中将对象数据作为引用读取，也可以将其作为原始对象的序列化副本读取。AIR 定义了四种传输模式，这些模式确定了对象是以引用还是以序列化副本的形式传输：

传输模式	说明
ClipboardTransferModes.ORIGINAL_ONLY	仅返回引用。如果没有引用，则返回 null 值。
ClipboardTransferModes.ORIGINAL_PREFERRED	如果存在引用，将返回引用。否则返回序列化副本。
ClipboardTransferModes.CLONE_ONLY	仅返回序列化副本。如果没有可用的序列化副本，则返回 null 值。
ClipboardTransferModes.CLONE_PREFERRED	如果存在序列化副本，将返回序列化副本。否则返回引用。

读取和写入自定义数据格式

将对象写入剪贴板时，可以使用任何不以保留前缀 air: 开头的字符串作为格式参数。请使用相同字符串作为读取对象的格式。

下面的示例说明了如何从剪贴板读取对象和向其中写入对象：

```
function createClipboardObject(object) {
    var transfer = new air.Clipboard();
    transfer.setData("object", object, true);
}
```

若要从 Clipboard 对象中提取序列化对象（放置或粘贴操作之后），请使用相同格式名称和 cloneOnly 或 clonePreferred 传输模式。

```
var transfer = clipboard.getData("object", air.ClipboardTransferMode.CLONE_ONLY);
```

此时将始终向 Clipboard 对象添加一个引用。若要从 Clipboard 对象中提取引用（放置或粘贴操作之后）而不是提取序列化副本，请使用 originalOnly 或 originalPreferred 传输模式：

```
var transferredObject =
    clipboard.getData("object", air.ClipboardTransferMode.ORIGINAL_ONLY);
```

仅当 `Clipboard` 对象是源自当前 AIR 应用程序中时，引用才有效。当引用可用时，可使用 `originalPreferred` 传输模式访问该引用；当引用不可用时，则可使用该模式访问序列化副本。

延迟呈现

如果创建数据格式的计算成本高昂，则可以通过提供按需提供数据的函数来使用延迟呈现。仅当放置或粘贴操作的接收方请求延迟格式的数据时才会调用此函数。

通过 `setDataHandler()` 方法将呈现函数添加到 `Clipboard` 对象中。该函数必须返回相应格式的数据。例如，如果调用了 `setDataHandler(ClipboardFormat.TEXT_FORMAT, writeText)`，则 `writeText()` 函数必须返回字符串。

如果使用 `setData()` 方法将同一类型的数据格式添加到 `Clipboard` 对象，则该数据优先于其延迟版本（不会调用呈现函数）。如果再次访问该同一剪贴板数据，则可能会再次调用或不调用该呈现函数。

注：在 Mac OS X 中，使用标准 AIR 剪贴板格式时不会进行延迟呈现。而会立即调用呈现函数。

使用延迟呈现函数粘贴文本

下面的示例说明了如何实现延迟呈现函数。

当按下此示例中的“Copy”按钮时，应用程序将清除系统剪贴板，确保不留下来自上次剪贴板操作的数据，然后通过剪贴板的 `setDataHandler()` 方法将 `renderData()` 函数放到剪贴板上。

当按下“Paste”按钮时，应用程序将访问剪贴板并设置目标文本。由于已使用函数而不是字符串设置了剪贴板上的文本数据格式，剪贴板将调用 `renderData()` 函数。`renderData()` 函数返回源文本中的文本，该文本随后将分配给目标文本。

请注意，如果按下“Paste”按钮前编辑源文本，则此编辑操作将反映到粘贴的文本中，即使按下“Paste”按钮后再进行编辑也是如此。这是因为呈现函数直到按下“Paste”按钮后才会复制源文本。（当在实际的应用程序中使用延迟呈现时，最好以某种方式存储或保护源数据以防止出现此问题。）

```
<html>
<head>
    <title>Deferred rendering</title>
    <script src="AIRAliases.js" type="text/javascript"></script>
    <script language="javascript" type="text/javascript">
        function doCopy(){
            air.Clipboard.generalClipboard.clear();
            air.Clipboard.generalClipboard.setDataHandler(
                air.ClipboardFormats.TEXT_FORMAT, renderData);
        }

        function doPaste(){
            document.getElementById("destination").innerHTML =
                air.Clipboard.generalClipboard.getData(air.ClipboardFormats.TEXT_FORMAT);
        }

        function renderData(){
            air.trace("Rendering data");
            return document.getElementById("source").innerHTML;
        }
    </script>
</head>
<body>
    <button onClick="doCopy()">Copy</button>
    <button onClick="doPaste()">Paste</button>
    <p>Source:</p>
    <p id="source" contentEditable="true">Neque porro quisquam est qui dolorem ipsum
        quia dolor sit amet, consectetur, adipisci velit.</p>
    <hr>
    <p>Destination:</p>
    <p id="destination"></p>
</body>
</html>
```

第 24 章：使用字节数组

ByteArray 类允许读取和写入二进制数据流，该数据流本质上是字节数组。该类提供了一种访问最基本的数据的方法。因为计算机数据由字节（即包含 8 位的组）组成，因此能够以字节为单位读取数据意味着您可以访问那些不存在类和访问方法的数据。**ByteArray** 类允许您在字节级分析任何数据流，从位图到通过网络的数据流。

利用 `writeObject()` 方法可以将序列化 Action Message Format (AMF) 格式的对象写入 **ByteArray**，而利用 `readObject()` 方法可以从 **ByteArray** 中将序列化对象读取到原始数据类型的变量。可以将显示对象以外的任何对象序列化，显示对象是可以放在显示列表中的那些对象。如果自定义类可用于运行时，也可以将序列化对象重新指定给自定义类实例。将一个对象转换为 AMF 格式之后，就可以通过网络连接有效传输该对象或将该对象保存到文件中。

此处描述的 Adobe® AIR® 应用程序示例将读取一个 `.zip` 文件，以该文件为例说明处理字节流、提取 `.zip` 文件包含的文件列表并将其写入桌面的过程。

读取并写入 **ByteArray**

ByteArray 类在 `flash.utils` 包中；如果代码包括 `AIRAliases.js` 文件，您也可以使用别名 `air.ByteArray` 来引用 **ByteArray** 类。若要创建 **ByteArray**，请按以下示例中所示调用 **ByteArray** 构造函数：

```
var stream = new air.ByteArray();
```

ByteArray 方法

任何有意义的数据流均将以某种格式组织起来，以便于您分析并找到所需信息。例如，简单的员工文件中的记录可能包括 ID 号、姓名、地址、电话号码等等。MP3 音频文件包含用于标识所下载文件的标题、作者、专辑、发行日期和流派的 ID3 标签。通过格式您就可以知道数据在数据流中的预期顺序。这样，您就可以采用智能方式读取字节流。

ByteArray 类包括几种方法，可以使数据流的读写更加容易。其中几种方法包括 `readBytes()` 和 `writeBytes()`、`readInt()` 和 `writeInt()`、`readFloat()` 和 `writeFloat()`、`readObject()` 和 `writeObject()`、`readUTFBytes()` 和 `writeUTFBytes()`。利用以上方法可以将数据从数据流读入特定数据类型的变量，也可以从特定数据类型的变量直接写入二进制数据流。

例如，以下代码读取一个简单的由字符串和浮点数组成的数组并将每个元素写入 **ByteArray**。此数组的结构使代码可以调用相应的 **ByteArray** 方法 (`writeUTFBytes()` 和 `writeFloat()`) 来写入数据。重复的数据模式使循环读取该数组成为可能。

```
// The following example reads a simple Array (groceries), made up of strings
// and floating-point numbers, and writes it to a ByteArray.

// define the grocery list Array
var groceries = ["milk", 4.50, "soup", 1.79, "eggs", 3.19, "bread" , 2.35]
// define the ByteArray
var bytes = new air.ByteArray();
// for each item in the array
for (i = 0; i < groceries.length; i++) {
    bytes.writeUTFBytes(groceries[i++]); //write the string and position to the next item
    bytes.writeFloat(groceries[i]); // write the float
    air.trace("bytes.position is: " + bytes.position); //display the position in ByteArray
}
air.trace("bytes length is: " + bytes.length); // display the length
```

position 属性

position 属性存储指针的当前位置，该指针在读写过程中指向 **ByteArray**。**position** 属性的初始值为 0，如以下代码中所示：

```
var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0
```

当您读取或写入 `ByteArray` 时，您使用的方法将更新 `position` 属性以指向紧随上次所读取或写入字节后的位置。例如，以下代码将一个字符串写入 `ByteArray`，随后 `position` 属性将指向 `ByteArray` 中该字符串后面紧邻的字节：

```
var bytes = new air.ByteArray();
air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12
```

同样，读取操作会使 `position` 属性值按照读取的字节数而相应增加。

```
var bytes = new air.ByteArray();

air.trace("bytes.position is initially: " + bytes.position); // 0
bytes.writeUTFBytes("Hello World!");
air.trace("bytes.position is now: " + bytes.position); // 12
bytes.position = 0;
air.trace("The first 6 bytes are: " + (bytes.readUTFBytes(6))); // Hello
air.trace("And the next 6 bytes are: " + (bytes.readUTFBytes(6))); // World!
```

请注意，您可以将 `position` 属性设置为 `ByteArray` 中的一个特定位置从而基于该偏移量进行读取或写入。

bytesAvailable 和 length 属性

`length` 和 `bytesAvailable` 属性分别指示 `ByteArray` 的长度为多少以及从当前位置到结尾处还剩多少字节。以下示例说明了您可以通过什么方式使用这些属性。该示例将一个文本字符串写入 `ByteArray` 然后一次从 `ByteArray` 中读取一个字节，直到遇到字符 “a” 或到达结尾 (`bytesAvailable <= 0`)。

```
var bytes = new air.ByteArray();
var text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus etc./";

bytes.writeUTFBytes(text); // write the text to the ByteArray
air.trace("The length of the ByteArray is: " + bytes.length); // 70
bytes.position = 0; // reset position
while (bytes.bytesAvailable > 0 && (bytes.readUTFBytes(1) != 'a')) {
    //read to letter a or end of bytes
}
if (bytes.position < bytes.bytesAvailable) {
    air.trace("Found the letter a; position is: " + bytes.position); // 23
    air.trace("and the number of bytes available is: " + bytes.bytesAvailable); // 47
}
```

Endian 属性

在存储多字节数字（即需要超过 1 个字节的内存来存储的数字），各种计算机可能彼此有所差异。例如，一个整数可能需要占用 4 个字节，即 32 位内存。某些计算机首先将数字中的最高有效字节存储在最低的内存地址，而其他计算机则首先存储最低有效字节。计算机的这一属性（即字节顺序属性）被称为 **big endian**（最高有效字节位于最前）或 **little endian**（最低有效字节位于最前）。例如，数字 `0x31323334` 对于 **big endian** 和 **little endian** 字节顺序将分别存储为以下形式，其中 `a0` 代表 4 个字节的最低内存地址而 `a3` 代表最高内存地址：

Big Endian	Big Endian	Big Endian	Big Endian
a0	a1	a2	a3
31	32	33	34

Little Endian	Little Endian	Little Endian	Little Endian
a0	a1	a2	a3
34	33	32	31

利用 `ByteArray` 类的 `Endian` 属性可以为要处理的多字节数字表示此字节顺序。对于此属性可接受的值为 "bigEndian" 或 "littleEndian"，并且 `Endian` 类定义了常量 `BIG_ENDIAN` 和 `LITTLE_ENDIAN`，从而通过这些字符串设置 `Endian` 属性。

compress() 和 uncompress() 方法

利用 `compress()` 方法可以根据指定作为参数的压缩算法压缩 `ByteArray`。利用 `uncompress()` 方法可以根据压缩算法对压缩的 `ByteArray` 进行解压缩。调用 `compress()` 和 `uncompress()` 之后，会将字节数组的长度设置为新的长度，并将 `position` 属性设置为结尾。

`CompressionAlgorithm` 类定义了可用来指定压缩算法的常量。AIR 支持 `deflate` 和 `zlib` 这两种算法。这种 `deflate` 压缩算法用于多种压缩格式，如 `zlib`、`gzip` 及一些 `zip` 实现等。在 <http://www.ietf.org/rfc/rfc1950.txt> 中对 `zlib` 压缩数据格式进行了说明；在 <http://www.ietf.org/rfc/rfc1951.txt> 中对 `deflate` 压缩算法进行了说明。

以下示例使用 `deflate` 算法压缩名为 `bytes` 的 `ByteArray`：

```
bytes.compress(air.CompressionAlgorithm.DEFLATE);
```

以下示例使用 `deflate` 算法对压缩的 `ByteArray` 进行解压缩：

```
bytes.uncompress(CompressionAlgorithm.DEFLATE);
```

读取和写入对象

`readObject()` 和 `writeObject()` 方法可从 `ByteArray` 读取并向其写入以序列化 Action Message Format (AMF) 格式编码的对象。AMF 是 Adobe 创建并由各种 ActionScript 3.0 类使用的专有消息协议，这些类包括 `Netstream`、`NetConnection`、`NetStream`、`LocalConnection` 和 `SharedObject`。

单字节类型标记说明了编码数据遵循的类型。AMF 使用以下 13 种数据类型：

```
value-type = undefined-marker | null-marker | false-marker | true-marker | integer-type |
double-type | string-type | xml-doc-type | date-type | array-type | object-type |
xml-type | byte-array-type
```

编码数据将遵循类型标记，除非标记表示一个可能的值（例如 `null`、`true` 或 `false`），在这种情况下将不对任何数据进行编码。

存在两种版本的 AMF：AMF0 和 AMF3。AMF0 通过引用发送复杂对象并允许端点还原对象关系。AMF3 通过以下方式对 AMF0 进行了改进：通过引用发送对象 `traits` 和字符串，除对象引用外，还支持 ActionScript 3.0 中引入的新数据类型。`ByteArray.objectEncoding` 属性指定了用于对对象数据进行编码的 AMF 版本。`flash.net.ObjectEncoding` 类定义了用于指定 AMF 版本的常量：`ObjectEncoding.AMF0` 和 `ObjectEncoding.AMF3`。

以下示例调用 `writeObject()` 将 XML 对象写入 `ByteArray`，随后将该 `ByteArray` 写入桌面上的 `order` 文件。该示例在完成时将在 AIR 窗口中显示消息“Wrote order file to desktop!”

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"><html xmlns="http://www.w3.org/1999/xhtml">
<head>
<style type="text/css">
#taFiles
{
    border: 1px solid black;
    font-family: Courier, monospace;
    white-space: pre;
    width: 95%;
    height: 95%;
    overflow-y: scroll;
}
</style>
<script type="text/javascript" src="AIRAliases.js" ></script>
<script type="text/javascript">

//define ByteArray
var inBytes = new air.ByteArray();
//add objectEncoding value and file heading to output text
var output = "Object encoding is: " + inBytes.objectEncoding + "\n\n" + "order file: \n\n";

function init() {

    readFile("order", inBytes);
    inBytes.position = 0;//reset position to beginning
    // read XML from ByteArray
    var orderXML = inBytes.readObject();
    // convert to XML Document object
    var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
    document.write(myXML.getElementsByTagName("menuName") [0].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price") [0].childNodes[0].nodeValue + "<br/>");      //
burger: 3.95
    document.write(myXML.getElementsByTagName("menuName") [1].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price") [1].childNodes[0].nodeValue + "<br/>");      //
fries: 1.45
} // end of init()

// read specified file into byte array
function readFile(fileName, data) {
    var inFile = air.File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName); // name of file to read
    var inStream = new air.FileStream();
    inStream.open(inFile, air FileMode.READ);
    inStream.readBytes(data, 0, data.length);
    inStream.close();
}
</script>
</head>

<body onload = "init();">
    <div id="taFiles"></div>
</body>
</html>
```

readObject() 方法从 ByteArray 读取序列化 AMF 格式的对象，并将其存储在指定类型的对象中。以下示例从桌面将 order 文件读入 ByteArray (inBytes) 中，并调用 readObject() 将其存储在 orderXML 中，然后将其转换为 XML 对象文档 myXML，并显示两项商品和价格元素的值。该示例还显示了 objectEncoding 属性的值以及 order 文件内容的标头。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<style type="text/css">
#taFiles
{
    border: 1px solid black;
    font-family: Courier, monospace;
    white-space: pre;
    width: 95%;
    height: 95%;
    overflow-y: scroll;
}
</style>
<script type="text/javascript" src="AIRAliases.js" ></script>
<script type="text/javascript">

//define ByteArray
var inBytes = new air.ByteArray();
//add objectEncoding value and file heading to output text
var output = "Object encoding is: " + inBytes.objectEncoding + "<br/><br/>" + "order file items:" +
"<br/><br/>";

function init() {

    readFile("order", inBytes);
    inBytes.position = 0;//reset position to beginning
    // read XML from ByteArray
    var orderXML = inBytes.readObject();
    // convert to XML Document object
    var myXML = (new DOMParser()).parseFromString(orderXML, "text/xml");
    document.write(output);
    document.write(myXML.getElementsByTagName("menuName") [0].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price") [0].childNodes[0].nodeValue + "<br/>");      //
burger: 3.95
    document.write(myXML.getElementsByTagName("menuName") [1].childNodes[0].nodeValue + ": ");
    document.write(myXML.getElementsByTagName("price") [1].childNodes[0].nodeValue + "<br/>");      //
fries: 1.45
}    // end of init()

// read specified file into byte array
function readFile(fileName, data) {
    var inFile = air.File.desktopDirectory; // source folder is desktop
    inFile = inFile.resolvePath(fileName); // name of file to read
    var inStream = new air.FileStream();
    inStream.open(inFile, air FileMode.READ);
    inStream.readBytes(data, 0, data.length);
    inStream.close();
}
</script>
</head>

<body onload = "init();">
    <div id="taFiles"></div>
</body>
</html>
```

ByteArray 示例：读取 .zip 文件

该示例演示了如何读取包含若干不同类型文件的简单 .zip 文件。读取过程如下：从每个文件的元数据中提取相关数据，将每个文件解压缩至 `ByteArray` 并将该文件写入桌面。

.zip 文件的一般结构基于 PKWARE Inc. 的规范（此规范位于

<http://www.pkware.com/documents/casestudies/APPNOTE.TXT>）。首先是 .zip 归档文件中第一个文件的文件标头和文件数据，接下来依次是其余各文件的文件标头及文件数据。（文件标头的结构将在后面介绍。）接下来，.zip 文件有可能包括数据描述符记录（通常是在内存中创建输出 zip 文件而不是将其保存到磁盘的情况下包括该记录）。接下来是若干其他可选元素：归档解密标头、归档额外数据记录、中央目录结构、中央目录记录的 Zip64 结尾、中央目录定位器的 Zip64 结尾和中央目录记录的结尾。

本示例中所编写的代码仅用于分析不包含文件夹且不需要数据描述符记录的 zip 文件。它将忽略最后一个文件的数据后面的所有信息。

每个文件的文件标头的格式如下：

文件标头签名	4 字节
所需版本	2 字节
一般用途标记	2 字节
压缩方法	2 字节 (8=DEFLATE; 0=UNCOMPRESSED)
文件的最后修改时间	2 字节
文件的最后修改日期	2 字节
crc-32	4 字节
压缩后的大小	4 字节
解压缩后的大小	4 字节
文件名长度	2 字节
额外字段长度	2 字节
文件名	变量
额外字段	变量

文件标头的后面是实际的文件数据，既可以是压缩后的也可以是解压缩后的文件数据，具体取决于压缩方法标志。如果文件数据为解压缩后的数据，则此标志为 0；如果该数据为使用 DEFLATE 算法压缩的数据，则为 8；如果采用的是其他压缩算法，则为其他值。

该示例的用户界面由一个标签和一个文本区域 (`taFiles`) 组成。该应用程序将它在 `zip` 文件中遇到的各文件的以下信息写入文本区域：文件名、压缩后的大小和解压缩后的大小。以下 HTML 页定义了该应用程序的用户界面：

```

<html>
    <head>
        <style type="text/css">
            #taFiles
            {
                border: 1px solid black;
                font-family: Courier, monospace;
                white-space: pre;
                width: 95%;
                height: 95%;
                overflow-y: scroll;
            }
        </style>
        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="text/javascript">
            // The application code goes here
        </script>
    </head>
    <body onload="init();">
        <div id="taFiles"></div>
    </body>
</html>

```

程序开头将执行以下任务：

- 定义 bytes ByteArray

```
var bytes = new air.ByteArray();
```

- 定义用来存储文件标头中元数据的变量

```
// variables for reading fixed portion of file header
var fileName = new String();
var fNameLength;
var xfldLength;
var offset;
var compSize;
var uncompSize;
var compMethod;
var signature;
```

```
var output;
```

- 定义用来表示 .zip 文件的 File (zfile) 和 FileStream (zStream) 对象，并指定将从中提取文件的 .zip 文件的位置（桌面对目录下名为“HelloAIR.zip”的文件）。

```
// File variables for accessing .zip file
var zfile = air.File.desktopDirectory.resolvePath("HelloAIR.zip");
var zStream = new air.FileStream();
```

程序代码以 init() 方法开始，该方法作为 body 标签的 onload 事件处理函数调用。

```
function init()
{
```

该程序将首先以 READ 模式打开 .zip 文件。

```
zStream.open(zfile, air.FileMode.READ);
```

然后将 endian 的 bytes 属性设置为 LITTLE_ENDIAN，以指示数字字段的字节顺序为最低有效字节位于最前。

```
bytes.endian = airEndian.LITTLE_ENDIAN;
```

接下来， while() 语句开始了一个循环，直到文件流中的当前位置大于或等于文件大小时才停止循环。

```
while (zStream.position < zfile.size)
{
```

循环中的第一个语句将文件流的前 30 个字节读入 `ByteArray bytes` 中。前 30 个字节组成了第一个文件标头的固定大小部分。

```
// read fixed metadata portion of local file header
zStream.readBytes(bytes, 0, 30);
```

接下来，代码将从这 30 字节标头最前面的字节中读取一个整数 (`signature`)。ZIP 格式定义指定每个文件标头的签名位十六进制值 `0x04034b50`；如果签名不同，则表明代码已移出 `zip` 文件的文件部分且再没有可提取的文件。在此情况下，代码将立即退出 `while` 循环而不是等待到达字节数组的结尾。

```
bytes.position = 0;
signature = bytes.readInt();
// if no longer reading data files, quit
if (signature != 0x04034b50)
{
    break;
}
```

代码的下一部分将在偏移量为 8 处读取标头字节并将值存储在变量 `compMethod` 中。该字节包含指示压缩此文件时所用压缩方法的值。允许使用多种压缩方法，但实际上几乎所有 `.zip` 文件均使用 DEFLATE 压缩算法。如果当前文件以 DEFLATE 压缩方式进行压缩，则 `compMethod` 为 8；如果文件未压缩，则 `compMethod` 为 0。

```
bytes.position = 8;
compMethod = bytes.readByte(); // store compression method (8 == Deflate)
```

位于前 30 个字节之后的部分是标头的可变长度部分，包含了文件名并可能包含额外字段。变量 `offset` 用于存储此部分的大小。该大小的计算方式为将文件名长度和额外字段长度相加，这两个长度可分别从标头中偏移量为 26 和 28 的位置读取。

```
offset = 0; // stores length of variable portion of metadata
bytes.position = 26; // offset to file name length
fNameLength = bytes.readShort(); // store file name
offset += fNameLength; // add length of file name
bytes.position = 28; // offset to extra field length
xfldLength = bytes.readShort();
offset += xfldLength; // add length of extra field
```

接下来程序将读取文件标头的可变长度部分，以将该部分的字节数存储在 `offset` 变量中。

```
// read variable length bytes between fixed-length header and compressed file data
zStream.readBytes(bytes, 30, offset);
```

程序将从标头的可变长度部分中读取文件名并在文本区域中显示它，同时显示文件压缩后（已压缩）及解压缩后（原始）大小。

```
bytes.position = 30;
fileName = bytes.readUTFBytes(fNameLength); // read file name
output += fileName + "<br />"; // write file name to text area
bytes.position = 18;
compSize = bytes.readUnsignedInt(); // store size of compressed portion
output += "\tCompressed size is: " + compSize + '<br />';
bytes.position = 22; // offset to uncompressed size
uncompSize = bytes.readUnsignedInt(); // store uncompressed size
output += "\tUncompressed size is: " + uncompSize + '<br />';
```

该示例从文件流中将文件的其余部分按照压缩后大小所指定的长度读入到 `bytes` 中，同时覆盖前 30 字节中的文件标头。即使文件并未压缩，压缩后的大小也是精确的，因为在此情况下，压缩后的大小将等于文件未压缩时的大小。

```
// read compressed file to offset 0 of bytes; for uncompressed files
// the compressed and uncompressed size is the same
zStream.readBytes(bytes, 0, compSize);
```

接下来，示例将对压缩文件进行解压缩，并调用 `outfile()` 函数将其写入输出文件流。它向 `outfile()` 传递文件名和包含文件数据的字节数组。

```
if (compMethod == 8) // if file is compressed, uncompress
{
    bytes.uncompress(air.CompressionAlgorithm.DEFLATE);
}
outFile(fileName, bytes); // call outFile() to write out the file

} // end of while loop
```

```
document.getElementById("taFiles").innerHTML = output;
} // end of init() method
```

outfile() 函数以 WRITE 模式在桌面上打开输出文件，同时为其指定 filename 参数提供的名称。然后该函数将把文件数据从 data 参数中写入输出文件流 (outStream) 并关闭该文件。

```
function outFile(fileName, data)
{
    var outFile = air.File.desktopDirectory; // dest folder is desktop
    outFile = outFile.resolvePath(fileName); // name of file to write
    var outStream = new air.FileStream();
    // open output file stream in WRITE mode
    outStream.open(outFile, air FileMode.WRITE);
    // write out the file
    outStream.writeBytes(data, 0, data.length);
    // close it
    outStream.close();
}
```

第 25 章：使用本地 SQL 数据库

Adobe® AIR® 包括创建和使用本地 SQL 数据库的功能。运行时包括一个 SQL 数据库引擎，该引擎使用开放源代码 SQLite 数据库系统，支持许多标准 SQL 功能。本地 SQL 数据库可用于存储本地永久性数据。例如，它可用于应用程序数据、应用程序用户设置、文档或希望应用程序在本地保存的任何其它类型的数据。

有关本地 SQL 数据库的其它在线信息

可以从以下源中查找有关使用本地 SQL 数据库的详细信息：

快速入门（**Adobe AIR** 开发人员中心）

- [异步处理本地 SQL 数据库](#)
- [同步处理本地 SQL 数据库](#)
- [使用加密数据库](#)

语言参考

- [SQLCollationType](#)
- [SQLColumnNameStyle](#)
- [SQLColumnSchema](#)
- [SQLConnection](#)
- [SQLError](#)
- [SQLErrorEvent](#)
- [SQLErrorOperation](#)
- [SQLEvent](#)
- [SQLIndexSchema](#)
- [SQLMode](#)
- [SQLResult](#)
- [SQLSchema](#)
- [SQLSchemaResult](#)
- [SQLStatement](#)
- [SQLTableSchema](#)
- [SQLTransactionLockType](#)
- [SQLTriggerSchema](#)
- [SQLUpdateEvent](#)
- [SQLViewSchema](#)

Adobe 开发人员中心文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心（搜索“AIR SQL”）](#)

关于本地 SQL 数据库

Adobe AIR 包括一个基于 SQL 的关系数据库引擎，该引擎在运行时中运行，数据以本地方式存储在运行 AIR 应用程序的计算机上的数据库文件中（例如，在计算机的硬盘驱动器上）。由于数据库的运行和数据文件的存储都在本地进行，因此，不管网络连接是否可用，AIR 应用程序都可以使用数据库。这样，运行时的本地 SQL 数据库引擎为存储永久的本地应用程序数据提供了一种便利机制，特别是您具有 SQL 和关系数据库经验时。

本地 SQL 数据库的用途

AIR 本地 SQL 数据库功能可以用于将应用程序数据存储在用户的本地计算机上的任何目的。Adobe AIR 包括在本地存储数据的几种机制，各机制具有不同的优点。以下是本地 SQL 数据库在 AIR 应用程序中的一些可能用途：

- 对于面向数据的应用程序（例如通讯簿），数据库可以用于存储主应用程序数据。
- 对于面向文档的应用程序（用户创建要保存并可能共享的文档），可以在用户指定的位置将每个文档另存为数据库文件。（不过请注意，除非加密了数据库，否则任何 AIR 应用程序都可以打开该数据库文件。对于可能存在敏感信息的文档，建议使用加密。）
- 对于支持网络的应用程序，数据库可以用于存储应用程序数据的本地缓存，或者在网络连接不可用时暂时存储数据。可以创建一种将本地数据库与网络数据存储同步的机制。
- 对于任何应用程序，数据库都可以用于存储单个用户的应用程序设置，例如用户选项或应用程序信息（如窗口大小和位置）。

关于 AIR 数据库和数据库文件

单个 Adobe AIR 本地 SQL 数据库作为单个文件存储在计算机的文件系统中。运行时包括 SQL 数据库引擎，该引擎管理数据库文件的创建和结构化以及操作和检索数据库文件中的数据。运行时不指定在文件系统上存储数据库数据的方式或位置；相反，每个数据库完全存储在单个文件中。您指定在文件系统中存储数据库文件的位置。单个 AIR 应用程序可以访问一个或多个单独的数据库（即单独的数据库文件）。由于运行时将每个数据库作为单个文件存储在文件系统上，因此可以在需要时按照应用程序的设计和操作系统的文件访问约束查找您的数据库。每个用户都可以具有其特定数据的单独数据库文件，或者数据库文件可以在单个计算机上共享数据的所有应用程序用户访问。由于数据对单个计算机是本地的，因此在不同计算机上的用户之间并不自动共享数据。本地 SQL 数据库引擎未提供对远程数据库或基于服务器的数据库执行 SQL 语句的任何功能。

关于关系数据库

关系数据库是一种在计算机上存储（和检索）数据的机制。数据被组织到表中：行表示记录或项目，而列（有时称为“字段”）将每个记录分到各个值中。例如，通讯簿应用程序可能包含“朋友”表。表中的每个行都表示存储在数据库中的单个朋友。表的列表示名字、姓氏、出生日期等数据。对于表中的每个朋友行，数据库为每个列存储一个单独的值。

关系数据库设计用于存储复杂数据，其中一个项目与其它类型的项目关联或相关。在关系数据库中，应该将具有一对多关系（其中单个记录可以与不同类型的多个记录相关）的任何数据分到不同的表中。例如，假定您希望通讯簿应用程序为每个朋友存储多个电话号码；这就是一对多关系。“朋友”表包含每个朋友的所有个人信息。单独的“电话号码”表包含所有朋友的所有电话号码。

除了存储有关朋友和电话号码的数据外，每个表都需要一段数据来跟踪这两个表之间的关系，以便使单个朋友记录与其电话号码匹配。该数据称为主键 — 将一个表中的每个行与该表中的其它行区分开的唯一标识符。主键可以是“自然键”，这意味着它是自然区分表中每个记录的数据项目之一。在“朋友”表中，如果您知道朋友的出生日期都是不同的，则可以将出生日期列用作“朋友”表的主键（自然键）。如果没有自然键，则应单独创建一个主键列，如“朋友 id”（应用程序用于区分各行的人工值）。

使用主键，可以设置多个表之间的关系。例如，假定“朋友”表有一个“朋友 id”列，其中包含每行（每个朋友）的唯一编号。可以用以下两列来构建相关的“电话号码”表：一个列包含电话号码所属的朋友的“朋友 id”，另一列包含实际的电话号码。这样，不管单个朋友具有多少个电话号码，都可以将它们全部存储在“电话号码”表中，并可以使用“朋友 id”主键将其

链接到相关的朋友。在相关表中使用一个表的主键指定记录之间的联系时，相关表中的值称为外键。与许多数据库不同，AIR本地数据库引擎不允许您创建外键约束（即自动检查插入的或更新的外键值在主键表中是否具有对应行的约束）。然而，外键关系是关系数据库结构的重要部分，而且在数据库的表之间创建关系时应该使用外键。

关于 SQL

结构化查询语言 (SQL) 用于关系数据库以操作和检索数据。SQL 是一种描述性语言，而不是一种过程语言。SQL 语句描述您所需的一组数据，而不是提供有关它应该如何检索数据的计算机指令。数据库引擎确定如何检索该数据。

SQL 语言已由美国国家标准协会 (ANSI) 进行了标准化。Adobe AIR 本地 SQL 数据库支持 SQL-92 标准的大部分内容。有关 Adobe AIR 中支持的 SQL 语言的具体描述，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)中的附录“[本地数据库中的 SQL 支持](#)”。

关于 SQL 数据库类

要在 JavaScript 中使用本地 SQL 数据库，请使用以下类的实例。（请注意，需要在 HTML 文档中加载文件 `AIRAliases.js` 才能使用这些类的 `air.*` 别名）：

类	说明
<code>air.SQLConnection</code>	提供创建和打开数据库（数据库文件）的方式，以及执行数据库级操作和控制数据库事务的方法。
<code>air.SQLStatement</code>	表示对数据库执行的单个 SQL 语句（单个查询或命令），包括定义语句文本和设置参数值。
<code>air.SQLResult</code>	提供一种通过执行语句获取信息或结果的方法，如执行 <code>SELECT</code> 语句后的结果行、受 <code>UPDATE</code> 或 <code>DELETE</code> 语句影响的行数等。

若要获取描述数据库结构的架构信息，请使用以下类：

类	说明
<code>air.SQLSchemaResult</code>	充当通过调用 <code>SQLConnection.loadSchema()</code> 方法生成的数据库架构结果的容器。
<code>air.SQLTableSchema</code>	提供描述数据库中单个表的信息。
<code>air.SQLViewSchema</code>	提供描述数据库中单个视图的信息。
<code>air.SQLIndexSchema</code>	提供描述数据库中表或视图的单个列的信息。
<code>air.SQLTriggerSchema</code>	提供描述数据库中单个触发器的信息。

以下类提供用于 `SQLConnection` 类的常数：

类	说明
<code>air.SQLMode</code>	定义一组常数，表示 <code>SQLConnection.open()</code> 和 <code>SQLConnection.openAsync()</code> 方法的 <code>openMode</code> 参数的可能值。
<code>air.SQLColumnNameStyle</code>	定义一组常数，它们表示 <code>SQLConnection.columnNameStyle</code> 属性的可能值。
<code>air.SQLTransactionLockType</code>	定义一组常数，表示 <code>SQLConnection.begin()</code> 方法的 <code>option</code> 参数的可能值。
<code>air.SQLCollationType</code>	定义一组常数，表示 <code>SQLColumnSchema()</code> 构造函数的 <code>SQLColumnSchema.defaultCollationType</code> 属性和 <code>defaultCollationType</code> 参数的可能值。

此外，以下类表示所用的事件（和支持常数）：

类	说明
air.SQLEvent	定义其任何操作成功执行时 <code>SQLConnection</code> 或 <code>SQLStatement</code> 实例调度的事件。每个操作都具有一个在 <code>SQLEvent</code> 类中定义的关联事件类型常数。
air.SQLErrorEvent	定义其任何操作导致错误时 <code>SQLConnection</code> 或 <code>SQLStatement</code> 实例调度的事件。
air.SQLUpdateEvent	定义因执行 <code>INSERT</code> 、 <code>UPDATE</code> 或 <code>DELETE</code> SQL 语句而导致其所连接的一个数据库中表数据发生更改时 <code>SQLConnection</code> 实例调度的事件。

最后，以下类提供有关数据库操作错误的信息：

类	说明
air.SQLError	提供有关数据库操作错误的信息，包括尝试的操作和出错原因。
air.SQLErrorOperation	定义一组常数，它们表示 <code>SQLError</code> 类的 <code>operation</code> 属性（它指示导致错误的数据库操作）的可能值。

关于同步和异步执行模式

编写代码以处理本地 SQL 数据库时，会指定以两种执行模式之一执行数据库操作：异步或同步执行模式。通常，代码示例说明如何以这两种方式执行每个操作，以便您可以使用最适合您需求的示例。

在异步执行模式中，为运行时提供一个指令，运行时将在请求的操作完成或失败时调度事件。首先，通知数据库引擎执行操作。在应用程序继续运行的同时，数据库引擎在后台工作。最后，完成操作时（或者它失败时），数据库引擎调度事件。由事件触发的代码执行后续操作。此方法具有一个重要的优点：运行时在后台执行数据库操作，同时主应用程序代码继续执行。如果数据库操作花费大量的时间，则应用程序继续运行。最重要的是，用户可以继续与其交互，而屏幕不会冻结。但是，与其它代码相比，编写异步操作代码可能更加复杂。在必须将多个相关的操作分配给各个事件侦听器方法的情况下，通常会出现此复杂性。

从概念上说，将操作作为单个步骤序列（一组同步操作，而不是分到几个事件侦听器方法中的一组操作）进行编码更为简单。除了异步数据库操作外，Adobe AIR 还允许您同步执行数据库操作。在同步执行模式中，操作不在后台运行。相反，它们以与所有其它应用程序代码相同的执行序列运行。通知数据库引擎执行操作。然后，代码在数据库引擎工作时暂停。完成操作后，继续执行下一行代码。

异步还是同步执行操作是在 `SQLConnection` 级别上设置的。使用单个数据库连接，无法同步执行某些操作或语句，同时异步执行其它操作或语句。通过调用 `SQLConnection` 方法打开数据库，可以指定 `SQLConnection` 是在同步还是异步执行模式下操作。如果调用 `SQLConnection.open()`，则连接以同步执行模式操作；如果调用 `SQLConnection.openAsync()`，则连接以异步执行模式操作。使用 `open()` 或 `openAsync()` 将 `SQLConnection` 实例连接到数据库后，除非关闭并重新打开与数据库的连接，否则该实例将固定为同步或异步执行模式。

每种执行模式各有其优点。虽然每种模式的大多数方面是类似的，但是在每种模式下工作时要牢记一些差异。有关这些主题的详细信息以及在每种模式下工作的建议，请参阅第 229 页的“[使用同步和异步数据库操作](#)”。

创建和修改数据库

数据库中必须定义了应用程序可以访问的表，应用程序才可以添加或检索数据。下面说明了创建数据库和在数据库中创建数据结构的任务。虽然这些任务的使用频率低于数据插入和数据检索，但大多数应用程序都必须使用这些任务。

创建数据库

若要创建数据库文件，请首先创建 `SQLConnection` 实例。调用其 `open()` 方法以同步执行模式将其打开，或者调用其 `openAsync()` 方法以异步执行模式将其打开。`open()` 和 `openAsync()` 方法用于打开与数据库的连接。如果所传递的 `File` 实例对于 `reference` 参数（第一个参数）引用的文件位置不存在，则 `open()` 或 `openAsync()` 方法将在该文件位置创建一个数据库文件，并打开与这个新创建的数据库的连接。

无论创建数据库时调用的是 `open()` 方法还是 `openAsync()` 方法，数据库文件的名称都可以是采用任何文件扩展名的任何有效文件名。如果调用 `reference` 参数为 `null` 的 `open()` 或 `openAsync()` 方法，则将创建新的内存中数据库，而不是磁盘上的数据库文件。

以下代码清单说明使用异步执行模式创建数据库文件（新数据库）的过程。在这种情况下，数据库文件保存在应用程序的存储目录中，其文件名为“`DBSample.db`”：

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
conn.openAsync(dbFile);
function openHandler(event)
{
    air.trace("the database was created successfully");
}
function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

注：尽管 `File` 类用于指向特定本机文件路径，但这会导致应用程序无法跨平台工作。例如，路径 `C:\Documents and Settings\joe\test.db` 仅适用于 Windows。出于以上原因，最好使用 `File` 类的静态属性（如 `File.applicationDirectory`）和 `resolvePath()` 方法（如上一个示例所示）。有关详细信息，请参阅第 163 页的“[File 对象的路径](#)”。

若要同步执行操作，请在使用 `SQLConnection` 实例打开数据库连接时，调用 `open()` 方法。以下示例说明如何创建和打开同步执行其操作的 `SQLConnection` 实例：

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
try
{
    conn.open(dbFile);
    air.trace("the database was created successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

创建数据库表

在数据库中创建表涉及使用与执行 SQL 语句（如 `SELECT`、`INSERT` 等）相同的过程对该数据库执行 SQL 语句。要创建表，请使用 `CREATE TABLE` 语句，该语句包括新表的列和约束的定义。有关执行 SQL 语句的详细信息，请参阅第 216 页的“[使用 SQL 语句](#)”。

以下示例演示如何使用异步执行模式在现有数据库文件中创建一个名为“`employees`”的表。请注意，此代码假定存在一个名为 `conn`、已进行实例化并连接到数据库的 `SQLConnection` 实例。

```
// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
var createStmt = new air.SQLStatement();
createStmt.sqlConnection = conn;
var sql =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0) " +
    ")";
createStmt.text = sql;
createStmt.addEventListener(air.SQLEvent.RESULT, createResult);
createStmt.addEventListener(air.SQLErrorEvent.ERROR, createError);
createStmt.execute();
function createResult(event)
{
    air.trace("Table created");
}
function createError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

以下示例演示如何使用同步执行模式在现有数据库文件中创建一个名为“employees”的表。请注意，此代码假定存在一个名为 conn、已进行实例化并连接到数据库的 SQLConnection 实例。

```
// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
var createStmt = new air.SQLStatement();
createStmt.sqlConnection = conn;
var sql =
    "CREATE TABLE IF NOT EXISTS employees (" +
    "    empId INTEGER PRIMARY KEY AUTOINCREMENT, " +
    "    firstName TEXT, " +
    "    lastName TEXT, " +
    "    salary NUMERIC CHECK (salary > 0) " +
    ")";
createStmt.text = sql;
try
{
    createStmt.execute();
    air.trace("Table created");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

操作 SQL 数据库数据

使用本地 SQL 数据库时，会执行一些常见任务。这些任务包括连接到数据库、向数据库的表中添加数据以及从数据库的表中检索数据。执行这些任务时，还要牢记几个问题，例如使用数据类型和处理错误。

请注意，还有几个数据库任务的执行频率较低，但经常需要在执行这些更常见的任务之前执行。例如，需要创建数据库和在数据库中创建表结构，才可以连接到数据库和从表中检索数据。第 213 页的“[创建和修改数据库](#)”中讨论了这些执行频率较低的初始设置任务。

可以选择异步执行数据库操作，这意味着数据库引擎在后台运行并在操作成功或失败时通过调度事件来通知您。还可以同步执行这些操作。在这种情况下，数据库操作依次执行，整个应用程序（包括对屏幕的更新）等待操作完成后再执行其它代码。本节中的示例演示如何以异步和同步方式执行操作。有关使用异步执行模式或同步执行模式的详细信息，请参阅第 229 页的“[使用同步和异步数据库操作](#)”。

连接到数据库

在执行任何数据库操作之前，请首先打开到数据库文件的连接。SQLConnection 实例用于表示到一个或多个数据库的连接。使用 SQLConnection 实例连接的第一个数据库称为“主”数据库。此数据库是使用 open() 方法（对于同步执行模式）或 openAsync() 方法（对于异步执行模式）连接的。

如果使用异步 openAsync() 操作打开数据库，则注册 SQLConnection 实例的 open 事件，以了解 openAsync() 操作何时完成。注册 SQLConnection 实例的 error 事件，以确定操作是否失败。

以下示例说明如何为异步执行打开现有的数据库文件。数据库文件名为“DBSample.db”，位于用户的应用程序存储目录中。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
conn.openAsync(dbFile, air.SQLMode.UPDATE);
function openHandler(event)
{
    air.trace("the database opened successfully");
}
function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

以下示例说明如何为同步执行打开现有的数据库文件。数据库文件名为“DBSample.db”，位于用户的应用程序存储目录中。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
try
{
    conn.open(dbFile, air.SQLMode.UPDATE);
    air.trace("the database opened successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

请注意，在异步示例的 openAsync() 方法调用中以及同步示例的 open() 方法调用中，第二个参数是常量 SQLMode.UPDATE。如果指定的文件不存在，则为第二个参数（openMode）指定 SQLMode.UPDATE 会导致运行时调度一个错误。如果为 openMode 参数传递 SQLMode.CREATE（或者如果使 openMode 参数处于关闭状态），则在指定的文件不存在时，运行时将尝试创建数据库文件。但是如果文件存在，则将打开该文件，这与使用 SQLMode.Update 相同。也可以为 openMode 参数指定 SQLMode.READ，以便在只读模式下打开现有的数据库。在这种情况下，可以从数据库检索数据，但是不能添加、删除或更改数据。

使用 SQL 语句

在运行时中，单个 SQL 语句（一个查询或命令）表示为 SQLStatement 对象。按照以下步骤创建和执行 SQL 语句：

创建 **SQLStatement** 实例。

在您的应用程序中，**SQLStatement** 对象表示 SQL 语句。

```
var selectData = new air.SQLStatement();
```

指定对其运行查询的数据库。

为此，请将 **SQLStatement** 对象的 **sqlConnection** 属性设置为与所需数据库连接的 **SQLConnection** 实例。

```
// A SQLConnection named "conn" has been created previously
selectData.sqlConnection = conn;
```

指定实际的 **SQL** 语句。

将语句文本创建为字符串，并将其分配给 **SQLStatement** 实例的 **text** 属性。

```
selectData.text = "SELECT col1, col2 FROM my_table WHERE col1 = :param1";
```

定义函数以处理执行操作的结果（仅限异步执行模式）。

使用 **addEventListener()** 方法将函数注册为 **SQLStatement** 实例的 **result** 和 **error** 事件的侦听器。

```
// using listener methods and addEventListener();
selectData.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectData.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
function resultHandler(event)
{
    // do something after the statement execution succeeds
}
function errorHandler(event)
{
    // do something after the statement execution fails
}
```

或者，可以使用 **Responder** 对象指定侦听器方法。在这种情况下，创建 **Responder** 实例并将侦听器方法链接到该实例。

```
// using a Responder
var selectResponder = new air.Responder(onResult, onError);
function onResult(result)
{
    // do something after the statement execution succeeds
}
function onError(error)
{
    // do something after the statement execution fails
}
```

如果语句文本包括参数定义，则分配这些参数的值。

若要分配参数值，请使用 **SQLStatement** 实例的 **parameters** 关联数组属性。

```
selectData.parameters[":param1"] = 25;
```

执行 **SQL** 语句。

调用 **SQLStatement** 实例的 **execute()** 方法。

```
// using synchronous execution mode
// or listener methods in asynchronous execution mode
selectData.execute();
```

此外，如果以异步执行模式使用 **Responder** 代替事件侦听器，则将 **Responder** 实例传递给 **execute()** 方法。

```
// using a Responder in asynchronous execution mode
selectData.execute(-1, selectResponder);
```

有关演示这些步骤的特定示例，请参阅以下主题：

第 219 页的“[从数据库检索数据](#)”



第 224 页的“[插入数据](#)”



第 226 页的“[更改或删除数据](#)”

在语句中使用参数

SQL 语句参数允许您创建可重用的 SQL 语句。使用语句参数时，语句中的值可以更改（如在 INSERT 语句中添加的值），但是基本的语句文本保持不变。因此，使用参数不仅可以提高性能，而且使编制应用程序代码的工作更加轻松。

了解语句参数

应用程序经常在自身中多次使用单个 SQL 语句，只是稍有不同。以一个库存跟踪应用程序为例，用户可以在其中向数据库添加新的库存项目。向数据库添加库存项目的应用程序代码执行 SQL INSERT 语句，该语句实际上向数据库添加数据。但是，每次执行该语句时都稍有不同。具体来说，在表中插入的实际值是不同的，因为它们特定于所添加的库存项目。

在多次使用一个 SQL 语句但该语句中的值不同的情况下，最佳方法是使用包括参数的 SQL 语句而不是在 SQL 文本中包括字面值。参数是语句文本中的一个占位符，每次执行语句时都将它替换为实际的值。若要在 SQL 语句中使用参数，请照常创建 SQLStatement 实例。对于分配给 text 属性的实际 SQL 语句，使用参数占位符而不是字面值。然后通过在 SQLStatement 实例的 parameters 属性中设置元素值来定义每个参数的值。parameters 属性是一个关联数组，因此使用以下语法设置特定值：

```
statement.parameters[parameter_identifier] = value;
```

如果使用命名参数，则 parameter_identifier 是字符串；如果使用未命名参数，则它是整数索引。

使用命名参数

参数可以是命名参数。命名参数具有一个特定的名称，数据库使用该名称将参数值与语句文本中其占位符位置相匹配。参数名称由“:”或“@”字符后跟一个名称组成，如以下示例所示：

```
:itemName  
@firstName
```

以下代码清单演示命名参数的用法：

```
var sql =  
    "INSERT INTO inventoryItems (name, productCode) " +  
    "VALUES (:name, :productCode)";  
var addItemStmt = new air.SQLStatement();  
addItemStmt.sqlConnection = conn;  
addItemStmt.text = sql;  
// set parameter values  
addItemStmt.parameters[":name"] = "Item name";  
addItemStmt.parameters[":productCode"] = "12345";  
addItemStmt.execute();
```

使用未命名参数

作为使用命名参数的一种替代方式，也可以使用未命名参数。若要使用未命名参数，请使用“?”字符表示 SQL 语句中的参数。按照参数在语句中的顺序，每个参数都分配有一个数字索引，数字索引从索引 0（表示第一个参数）开始。以下示例使用未命名参数演示上述示例的一个版本：

```

var sql =
    "INSERT INTO inventoryItems (name, productCode) " +
    "VALUES (?, ?)";
var addItemStmt = new air.SQLStatement();
addItemStmt.sqlConnection = conn;
addItemStmt.text = sql;
// set parameter values
addItemStmt.parameters[0] = "Item name";
addItemStmt.parameters[1] = "12345";
addItemStmt.execute();

```

使用参数的优点

在 SQL 语句中使用参数有以下几个优点：

性能更佳 与每次执行时动态创建 SQL 文本的 `SQLStatement` 实例相比，使用参数的 `SQLStatement` 实例可以更高效地执行。性能之所以得到提高，是因为只准备一次语句，然后可以使用不同的参数值多次执行它，而无需重新编译 SQL 语句。

显式数据类型指定 参数用于允许对构造 SQL 语句时未知的值进行类型替代。使用参数是保证将值的存储类传递到数据库的唯一方式。不使用参数时，运行时尝试根据相关联列的类型关联将所有值从其文本表示形式转换为存储类。有关存储类和列关联的详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)的附录“**本地数据库中的 SQL 支持**”中的“**数据类型支持**”一节。

安全性更高 使用参数有助于防止恶意技术攻击（称为 SQL 注入攻击）。在 SQL 注入攻击中，用户在用户可访问的位置（例如数据输入字段）输入 SQL 代码。如果应用程序代码通过将用户输入直接连接到 SQL 文本来构造 SQL 语句，则将对数据库执行用户输入的 SQL 代码。下面的列表显示将用户输入连接到 SQL 文本的示例。不要使用此技术：

```

// assume the variables "username" and "password"
// contain user-entered data
var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = '" + username + "' " +
    "    AND password = '" + password + "'";
var statement = new air.SQLStatement();
statement.text = sql;

```

使用语句参数而不是将用户输入的值连接到语句的文本中，可防止 SQL 注入攻击。SQL 注入无法发生的原因是，系统将参数值显式视为替代值，而不是作为字面语句文本的一部分。下面是对前面列表的建议替代方法：

```

// assume the variables "username" and "password"
// contain user-entered data
var sql =
    "SELECT userId " +
    "FROM users " +
    "WHERE username = :username " +
    "    AND password = :password";
var statement = new air.SQLStatement();
statement.text = sql;
// set parameter values
statement.parameters[":username"] = username;
statement.parameters[":password"] = password;

```

从数据库检索数据

从数据库检索数据分为以下两步。首先，执行 SQL SELECT 语句（描述要从数据库检索的一组数据）。然后，访问已检索的数据，并根据需要由应用程序显示或操作它。

执行 SELECT 语句

若要从数据库检索现有的数据，请使用 `SQLStatement` 实例。将相应的 SQL SELECT 语句分配给实例的 `text` 属性，然后调用其 `execute()` 方法。

有关 SELECT 语句的语法的详细信息，请参阅针对 HTML 开发人员的 Adobe AIR 语言参考中的附录“[本地数据库中的 SQL 支持](#)”。

下例演示使用异步执行模式执行 SELECT 语句，从名为“products”的表中检索数据：

```
// Include AIRAliases.js to use air.* shortcuts
var selectStmt = new air.SQLStatement();
// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;
selectStmt.text = "SELECT itemId, itemName, price FROM products";
// The resultHandler and errorHandler are listener functions are
// described in a subsequent code listing
selectStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
selectStmt.execute();
```

下例演示使用异步执行模式执行 SELECT 语句，从名为“products”的表中检索数据：

```
// Include AIRAliases.js to use air.* shortcuts
var selectStmt = new air.SQLStatement();
// A SQLConnection named "conn" has been created previously
selectStmt.sqlConnection = conn;
selectStmt.text = "SELECT itemId, itemName, price FROM products";
// This try..catch block is fleshed out in
// a subsequent code listing
try
{
    selectStmt.execute();
    // accessing the data is shown in a subsequent code listing
}
catch (error)
{
    // error handling is shown in a subsequent code listing
}
```

在异步执行模式下，语句完成执行时，`SQLStatement` 实例调度 `result` 事件 (`SQLEvent.RESULT`)，指示该语句已成功运行。或者，如果在 `execute()` 调用中传递 `Responder` 对象作为参数，则调用 `Responder` 对象的结果处理函数。在同步执行模式下，执行暂停，直到 `execute()` 操作完成，然后继续执行下一行代码。

访问 SELECT 语句结果数据

SELECT 语句完成执行后，下一步是访问已检索的数据。SELECT 结果集中的每行数据都将成为 `Object` 实例。该对象具有其名称与结果集的列名称匹配的属性。该属性包含结果集的列值。例如，假定 SELECT 语句指定一个结果集，该结果集具有名为“`itemId`”、“`itemName`”和“`price`”的三个列。对于结果集中的每一行，都创建一个 `Object` 实例，并含有名为 `itemId`、`itemName` 和 `price` 的属性。这些属性包含来自其相应列的值。

以下代码清单继续前面的代码清单，用于在异步执行模式下检索数据。它说明如何访问 `result` 事件侦听器方法中已检索的数据。

```

function resultHandler(event)
{
    var result = selectStmt.getResult();
    var numResults = result.data.length;
    for (i = 0; i < numResults; i++)
    {
        var row = result.data[i];
        var output = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        air.trace(output);
    }
}
function errorHandler(event)
{
    // Information about the error is available in the
    // event.error property, which is an instance of
    // the SQLError class.
}

```

以下代码清单扩展了前面的代码清单，用于在同步执行模式下检索数据。此代码扩展了之前同步执行示例中的 try..catch 块，并展示如何访问已检索的数据。

```

try
{
    selectStmt.execute();
    var result = selectStmt.getResult();
    var numResults = result.data.length;
    for (i = 0; i < numResults; i++)
    {
        var row = result.data[i];
        var output = "itemId: " + row.itemId;
        output += "; itemName: " + row.itemName;
        output += "; price: " + row.price;
        air.trace(output);
    }
}
catch (error)
{
    // Information about the error is available in the
    // error variable, which is an instance of
    // the SQLError class.
}

```

如前面的代码清单所示，结果对象包含在作为 SQLResult 实例的 data 属性提供的数组中。如果将异步执行与事件侦听器一起使用，若要检索该 SQLResult 实例，请调用 SQLStatement 实例的 getResult() 方法。如果在 execute() 调用中指定了 Responder 参数，则 SQLResult 实例将作为参数传递到结果处理函数。在同步执行模式下，调用 execute() 方法之后随时可以调用 SQLStatement 实例的 getResult() 方法。在任何情况下，在具有 SQLResult 对象后，都可以使用 data 数组属性访问结果行。

以下代码清单定义其文本为 SELECT 语句的 SQLStatement 实例。该语句从名为 firstName 的表的所有行中检索包含 lastName 和 employees 列值的行。此示例使用异步执行模式。执行完毕时，调用 selectResult() 方法，使用 SQLStatement.getResult() 访问所得的数据行，并使用 trace() 方法显示这些数据行。请注意，此列表假定存在一个名为 conn、已进行实例化并连接到数据库的 SQLConnection 实例。它还假定已创建“employees”表并为其填充了数据。

```
// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;
// define the SQL text
var sql =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;
// register listeners for the result and error events
selectStmt.addEventListener(air.SQLEvent.RESULT, selectResult);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, selectError);
// execute the statement
selectStmt.execute();
function selectResult(event)
{
    // access the result data
    var result = selectStmt.getResult();
    var numRows = result.data.length;
    for (i = 0; i < numRows; i++)
    {
        var output = "";
        for (columnName in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        air.trace("row[" + i.toString() + "]\t", output);
    }
}
function selectError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

以下代码清单演示与前面的代码清单相同的技术，但使用的是同步执行模式。该示例定义其文本为 SELECT 语句的 SQLStatement 实例。该语句从名为 firstName 的表的所有行中检索包含 lastName 和 employees 列值的行。使用 SQLStatement.getResult() 访问所得的数据行，并使用 trace() 方法显示这些数据行。请注意，此列表假定存在一个名为 conn、已进行实例化并连接到数据库的 SQLConnection 实例。它还假定已创建 “employees” 表并为其填充了数据。

```

// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
// create the SQL statement
var selectStmt = new air.SQLStatement();
selectStmt.sqlConnection = conn;
// define the SQL text
var sql =
    "SELECT firstName, lastName " +
    "FROM employees";
selectStmt.text = sql;
try
{
    // execute the statement
    selectStmt.execute();
    // access the result data
    var result = selectStmt.getResult();
    var numRows = result.data.length;
    for (i = 0; i < numRows; i++)
    {
        var output = "";
        for (columnName in result.data[i])
        {
            output += columnName + ": " + result.data[i][columnName] + "; ";
        }
        air.trace("row[" + i.toString() + "] \t", output);
    }
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}

```

定义 SELECT 结果数据的数据类型

默认情况下，由 SELECT 语句返回的每个行都创建为 **Object** 实例，以结果集的列名作为属性名，每列的值作为其关联属性的值。但是，在执行 SQL SELECT 语句之前，可以将 **SQLStatement** 实例的 **itemClass** 属性设置为一个类。通过设置 **itemClass** 属性，由 SELECT 语句返回的每个行将创建为指定类的实例。通过将 SELECT 结果集中的列名与 **itemClass** 类中的属性名相匹配，运行时将结果列值分配给属性值。

作为 **itemClass** 属性值分配的任何类都必须具有不需要任何参数的构造函数。另外，对于由 SELECT 语句返回的每个列，该类必须具有一个单一的属性。如果 SELECT 列表中的列在 **itemClass** 类中没有相匹配的属性名称，系统会将其视为错误。

检索部分 SELECT 结果

默认情况下，执行 SELECT 语句会一次检索结果集的所有行。语句完成后，通常以某种方式处理检索的数据，如创建对象或在屏幕上显示数据。如果语句返回了大量的行，则同时处理所有数据可能对计算机要求过高，这又会导致用户界面无法自行重绘。

通过指示运行时一次返回特定数量的结果行，可以提高应用程序的感知性能。这样做会使初始结果数据更快地返回。它还允许您将结果行分到各组中，以便在处理每组行后更新用户界面。请注意，只有在异步执行模式下使用此技术才是可行的。

若要检索部分 SELECT 结果，请为 **SQLStatement.execute()** 方法的第一个参数（**prefetch** 参数）指定一个值。**prefetch** 参数指示首次执行语句时检索的行数。调用 **SQLStatement** 实例的 **execute()** 方法时，将指定 **prefetch** 参数值，并且只会检索这么多行：

```

// Include AIRAliases.js to use air.* shortcuts
var stmt = new air.SQLStatement();
stmt.sqlConnection = conn;
stmt.text = "SELECT ...";
stmt.addEventListener(air.SQLEvent.RESULT, selectResult);
stmt.execute(20); // only the first 20 rows (or fewer) are returned

```

该语句调度 `result` 事件，指示第一组结果行是可用的。生成的 `SQLResult` 实例的 `data` 属性包含数据行，其 `complete` 属性指示是否存在要检索的其它结果行。若要检索其它结果行，请调用 `SQLStatement` 实例的 `next()` 方法。与 `execute()` 方法一样，使用 `next()` 方法的第一个参数指示下次调度 `result` 事件时要检索的行数。

```
function selectResult(event)
{
    var result = stmt.getResult();
    if (result.data != null)
    {
        // ... loop through the rows or perform other processing ...
        if (!result.complete)
        {
            stmt.next(20); // retrieve the next 20 rows
        }
        else
        {
            stmt.removeEventListener(air.SQLEvent.RESULT, selectResult);
        }
    }
}
```

每次 `next()` 方法返回一组后续的结果行时，`SQLStatement` 都会调度一个 `result` 事件。因此，可以使用同一侦听器函数继续处理结果（通过 `next()` 调用），直到检索了所有行为止。

有关详细信息，请参阅语言参考中对 `SQLStatement.execute()` 方法（`prefetch` 参数说明）和 `SQLStatement.next()` 方法的介绍。

插入数据

从数据库检索数据涉及执行 SQL `INSERT` 语句。语句完成执行后，如果数据库生成了键，则可以访问新插入的行的主键。

执行 `INSERT` 语句

若要向数据库中的表添加数据，请创建其文本为 SQL `INSERT` 语句的 `SQLStatement` 实例并执行它。

以下示例使用 `SQLStatement` 实例向已存在的 `employees` 表添加数据行。此示例演示如何使用异步执行模式插入数据。请注意，此列表假定存在一个名为 `conn`、已进行实例化并连接到数据库的 `SQLConnection` 实例。它还假定已创建“`employees`”表。

```

// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
// create the SQL statement
var insertStmt = new air.SQLStatement();
insertStmt.sqlConnection = conn;
// define the SQL text
var sql =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;
// register listeners for the result and failure (status) events
insertStmt.addEventListener(air.SQLEvent.RESULT, insertResult);
insertStmt.addEventListener(air.SQLErrorEvent.ERROR, insertError);
// execute the statement
insertStmt.execute();
function insertResult(event)
{
    air.trace("INSERT statement succeeded");
}
function insertError(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}

```

以下示例使用同步执行模式向已存在的 `employees` 表添加数据行。请注意，此列表假定存在一个名为 `conn`、已进行实例化并连接到数据库的 `SQLConnection` 实例。它还假定已创建“`employees`”表。

```

// Include AIRAliases.js to use air.* shortcuts
// ... create and open the SQLConnection instance named conn ...
// create the SQL statement
var insertStmt = new air.SQLStatement();
insertStmt.sqlConnection = conn;
// define the SQL text
var sql =
    "INSERT INTO employees (firstName, lastName, salary) " +
    "VALUES ('Bob', 'Smith', 8000)";
insertStmt.text = sql;
try
{
    // execute the statement
    insertStmt.execute();
    air.trace("INSERT statement succeeded");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}

```

检索已插入行的数据库生成的主键

通常，向表中插入数据行后，代码需要知道新插入行的数据库生成的主键或行标识符值。例如，在一个表中插入行后，您可能希望在相关表中添加行。在这种情况下，希望将主键值作为外键插入到相关表中。可以使用由语句执行生成的 `SQLResult` 对象检索新插入行的主键。这是在执行 `SELECT` 语句后用来访问结果数据的同一对象。与任何 SQL 语句一样，`INSERT` 语句执行完成时，运行时将创建 `SQLResult` 实例。如果使用事件侦听器，或者如果使用同步执行模式，则通过调用 `SQLStatement` 对象的 `getResult()` 方法访问该 `SQLResult` 实例。或者，如果使用的是异步执行模式，并将 `Responder` 实例传递到 `execute()` 调用，则将 `SQLResult` 实例作为参数传递到结果处理函数。在任一情况下，`SQLResult` 实例都具有属性 `lastInsertRowID`；如果执行的 SQL 语句是 `INSERT` 语句，则该属性包含最近插入的行的行标识符。

以下示例演示如何在异步执行模式下访问已插入行的主键：

```

insertStmt.text = "INSERT INTO ...";
insertStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);
insertStmt.execute();
function resultHandler(event)
{
    // get the primary key
    var result = insertStmt.getResult();
    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}

```

以下示例演示如何在同步执行模式下访问已插入行的主键：

```

insertStmt.text = "INSERT INTO ...";
try
{
    insertStmt.execute();
    // get the primary key
    var result = insertStmt.getResult();
    var primaryKey = result.lastInsertRowID;
    // do something with the primary key
}
catch (error)
{
    // respond to the error
}

```

请注意，根据以下规则，行标识符可能是也可能不是在表定义中指定为主键列的列值：

- 如果表是用其关联（列数据类型）为 INTEGER 的主键列定义的，则 lastInsertRowID 属性包含插入到该行中的值（如果它是 AUTOINCREMENT 列，则为由运行时生成的值）。
- 如果表是用多个主键列（组合键）或其关联不是 INTEGER 的单个主键列定义的，则数据库在后台生成行的行标识符值。该生成的值是 lastInsertRowID 属性的值。
- 该值始终是最近插入的行的行标识符。如果 INSERT 语句导致一个触发器激发（这将插入一行），则 lastInsertRowID 属性包含由触发器插入的最后一行的行标识符，而不是由 INSERT 语句创建的行的行标识符。因此，如果要具有显式定义的主键列，且其值通过 INSERT 属性在 SQLResult.lastInsertRowID 命令后可用，则必须将该列定义为 INTEGER PRIMARY KEY 列。但是，请注意，即使表不包括显式 INTEGER PRIMARY KEY 列，但就定义与相关表的关系而言，将数据库生成的行标识符用作表的主键也是同样可接受的。通过使用特殊的列名 ROWID、_ROWID_ 或 OID 之一，行标识符列值在任何 SQL 语句中都可用。可以在相关表中创建外键列，并将行标识符值用作外键列值，就像显式声明的 INTEGER PRIMARY KEY 列一样。在该意义上，如果使用的是任意主键而不是自然键，并且只要您不在乎生成主键值的运行时，则将 INTEGER PRIMARY KEY 列还是系统生成的行标识符用作表的主键以定义两个表之间的外键关系几乎没有差别。

有关主键和生成的行标识符的详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)的附录“[本地数据库中的 SQL 支持](#)”中标题为“[CREATE TABLE](#)”和“[表达式](#)”两节。

更改或删除数据

执行其它数据处理操作的过程与用于执行 SQL SELECT 或 INSERT 语句的过程完全相同。只需在 SQLStatement 实例的 text 属性中以不同的 SQL 语句替换即可：

- 若要更改表中的现有数据，请使用 UPDATE 语句。
- 要从表中删除一行或多行数据，请使用 DELETE 语句。

有关这些语句的描述，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)中的附录“[本地数据库中的 SQL 支持](#)”。

使用多个数据库

使用 `SQLConnection.attach()` 方法，在已具有打开的数据库的 `SQLConnection` 实例上打开与其它数据库的连接。在 `attach()` 方法调用中，使用 `name` 参数为附加的数据库提供名称。在编写语句操作该数据库时，可以在前缀中使用该名称（使用格式 `database-name.table-name`）在 SQL 语句中限定任何表名，指示运行时可以在指定的数据库中找到该表。

可以执行包括多个数据库中的表的单个 SQL 语句，这些数据库连接到同一 `SQLConnection` 实例。如果事务是在 `SQLConnection` 实例上创建的，则该事务适用于使用 `SQLConnection` 实例执行的所有 SQL 语句。不管语句运行在哪个附加的数据库上，这一点都适用。

或者，也可以在一个应用程序中创建多个 `SQLConnection` 实例，其中每个实例都连接到一个或多个数据库。但是，如果确实使用到同一数据库的多个连接，请牢记数据库事务不是跨 `SQLConnection` 实例共享的。因此，如果使用多个 `SQLConnection` 实例连接到同一数据库文件，则不能指望以预期方式应用这两个连接的数据更改。例如，如果通过不同的 `SQLConnection` 实例对同一数据库运行两个 `UPDATE` 或 `DELETE` 语句，并且在一个操作发生后出现应用程序错误，则数据库数据可能处于不可逆的中间状态，而且可能影响数据库的完整性（进而影响应用程序）。

处理数据库错误

通常，数据库错误处理与其它运行时错误处理类似。应该编写代码以备可能出现的错误，并对错误作出响应，而不是直到运行时才这样做。通常认为，可以将可能的数据库错误分为以下三类：连接错误、SQL 语法错误和约束错误。

连接错误

大多数的数据库错误是连接错误，它们可能出现在任何操作过程中。尽管存在防止连接错误的策略，但是，如果数据库是应用程序的关键部分，则几乎没有从连接错误中正常恢复的简单方法。

大多数连接错误与运行时和操作系统、文件系统及数据库文件交互的方式有关。例如，如果用户没有在文件系统上的特定位置创建数据库文件的权限，则会出现连接错误。以下策略有助于防止连接错误：

使用特定于用户的数据库文件 为每个用户提供其自己的数据库文件，而不是将单个数据库文件用于在单个计算机上使用应用程序的所有用户。该文件应该位于与用户帐户关联的目录中。例如，它可能在以下位置：应用程序的存储目录、用户的文档文件夹、用户的桌面等。

考虑不同的用户类型 在不同的操作系统上，用不同类型的用户帐户测试应用程序。请勿假定用户具有计算机上的管理员权限。此外，请勿假定安装了某应用程序的个人是运行该应用程序的用户。

考虑各个文件位置 如果允许用户指定保存数据库文件的位置或者选择要打开的文件，请考虑用户可能使用的文件位置。此外，请考虑定义对用户可以存储（或他们可以从中打开）数据库文件的位置的限制。例如，可以仅允许用户打开位于其用户帐户存储位置中的文件。

如果出现连接错误，则很可能出现在创建或打开数据库的首次尝试中。这意味着用户无法在应用程序中执行与数据库相关的任何操作。对于某些类型的错误，如只读或权限错误，一种可能的恢复技术是将数据库文件复制到其它位置。应用程序可以将数据库文件复制到用户有权创建和写入文件的其它位置，然后改用该位置。

语法错误

在 SQL 语句格式不正确，而应用程序尝试执行该语句时，会出现语法错误。由于本地数据库 SQL 语句是作为字符串创建的，因此无法进行编译时 SQL 语法检查。必须执行所有 SQL 语句才能检查其语法。使用以下策略可防止 SQL 语法错误：

全面地测试所有 SQL 语句 如有可能，在开发应用程序的过程中，将 SQL 语句编码为应用程序代码中的语句文本之前，单独对其进行测试。此外，使用代码测试方法（如单元测试）创建一组测试，在代码中运用每个可能的选项和变体。

使用语句参数和避免连接的（动态生成的）SQL 使用参数和避免动态生成的 SQL 语句，意味着每次执行语句时都使用相同的 SQL 语句文本。因此，测试语句和限制可能的变体更容易。如果必须动态生成 SQL 语句，请将语句的动态部分保持在最小限度内。此外，仔细验证任何用户输入，以确保它不会导致语法错误。

若要从语法错误中恢复，应用程序将需要复杂的逻辑才能检查 SQL 语句和更正其语法。通过遵循用于防止语法错误的上述准则，您的代码可以识别 SQL 语法错误的任何潜在运行时根源（如语句中使用的用户输入）。若要从语法错误中恢复，请为用户提供指导。指出要更正哪些内容才能使语句正确执行。

约束错误

在 INSERT 或 UPDATE 语句尝试向列添加数据时，会出现约束错误。如果新数据违反表或列的已定义约束之一，则会发生该错误。一组可能的约束包括：

唯一约束 指示对于表中的所有行，在一个列中不能有重复值。或者，将多个列组合在唯一约束中时，这些列中值的组合不得重复。换句话说，对于指定的具有唯一性的一列或多列，每个行必须是不同的。

主键约束 对于约束允许和不允许的数据，主键约束与唯一约束完全相同。

非 null 约束 指定单个列不能存储 NULL 值，因此在每个行中，该列必须具有一个值。

检查约束 允许您在一个或多个表上指定任意约束。常见的检查约束是一个规则，它定义列的值必须在某些界限内（例如，数字列的值必须大于 0）。另一种常见的检查约束类型指定列值之间的关系（例如，一个列的值必须与同一行中其它列的值不同）。

数据类型（列关联）约束 运行时强制实施列值的数据类型，尝试将类型不正确的值存储在列中时会出现错误。但是，在许多情况下，会转换值以匹配列的已声明数据类型。有关详细信息，请参阅第 229 页的“[使用数据库数据类型](#)”。

运行时不对外键值强制实施约束。换句话说，匹配现有的主键值不需要外键值。

除了预定义的约束类型外，运行时 SQL 引擎还支持使用触发器。触发器类似于事件处理函数。它是发生某个操作时执行的一组预定义指令。例如，可以定义一个触发器，它在向特定表插入数据或从中删除数据时运行。触发器的一个可能用途是检查数据更改，并在不满足指定的条件时导致出现错误。因此，触发器可以具有与约束相同的用途，防止约束错误和从中恢复的策略也适用于触发器生成的错误。但是，触发器生成的错误的错误 id 与约束错误的错误 id 不同。

在设计应用程序时，就确定了适用于特定表的一组约束。通过有意识地设计约束，可以更轻松地设计应用程序，以防止约束错误和从中恢复。但是，约束错误很难系统地预测和预防。很难预测的原因是，约束错误在添加应用程序数据后才出现。数据在创建后被添加到数据库时会出现约束错误。这些错误通常是由新数据和已经存在于数据库中的数据之间的关系导致的。以下策略可以帮助您避免许多约束错误：

仔细计划数据库结构和约束 约束的用途是强制实施应用程序规则和帮助保护数据库数据的完整性。在计划应用程序时，请考虑如何构建数据库来支持您的应用程序。作为该过程的一部分，确定数据的规则，如某些值是否必需、某值是否具有默认值、是否允许重复值等。这些规则可以指导您定义数据库约束。

显式指定列名 可以编写 INSERT 语句而不显式指定要在其中插入值的列，但是这样做会产生不必要的风险。通过显式命名要在其中插入值的列，可以允许自动生成的值、具有默认值的列和允许 NULL 值的列。此外，这样做可确保所有的 NOT NULL 列都插入了显式值。

使用默认值 每当为列指定 NOT NULL 约束时，尽可能在列定义中指定默认值。应用程序代码也可以提供默认值。例如，代码可以检查 String 变量是否为 null，并在使用它设置语句参数值之前为它分配一个值。

验证用户输入的数据 提前检查用户输入的数据，以确保它符合约束指定的限制，尤其是对于 NOT NULL 和 CHECK 约束。当然，UNIQUE 约束更难检查，因为这样做会要求执行 SELECT 查询来确定数据是否唯一。

使用触发器 可以编写一个触发器，用于验证（并可能替换）插入的数据或执行其它操作以更正无效的数据。此验证和更正可防止出现约束错误。

在许多方面，约束错误比其它类型的错误更难防范。幸运的是，有几个从约束错误恢复的策略，这样就不会使应用程序变得不稳定或不可用：

使用冲突算法 在列上定义约束时，以及创建 INSERT 或 UPDATE 语句时，您可以选择指定冲突算法。冲突算法定义在出现约束违规时数据库执行的操作。数据库引擎可以执行几种可能的操作。数据库引擎可以结束单个语句或整个事务。它可以忽略错误。它甚至可以删除旧数据，并将它替换为代码尝试存储的数据。

有关详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)的附录“[本地数据库中的 SQL 支持](#)”中的“[ON CONFLICT \(冲突算法\)](#)”一节。

提供纠正反馈 可以提前识别可能影响特定 SQL 命令的一组约束。因此，可以预期语句可能导致的约束错误。知道这一点，就可以生成应用程序逻辑来响应约束错误。例如，假定应用程序包括用于输入新产品的数据条目表单。如果数据库中的产品名称列是用 UNIQUE 约束定义的，则在数据库中插入新产品行的操作可能会导致约束错误。因此，应用程序设计用于预期约束错误。在错误发生时，应用程序将提醒用户，指出指定的产品名称已在使用中，并要求用户选择其它名称。另一种可能的响应是，允许用户查看有关同名的其它产品的信息。

使用数据库数据类型

在数据库中创建表时，用于创建表的 SQL 语句将为表中的每个列定义关联或数据类型。尽管可以省略关联声明，但是最好在 CREATE TABLE SQL 语句中显式声明列关联。

通常，在执行 SELECT 语句时，使用 INSERT 语句存储在数据库中的任何对象都将作为相同数据类型的实例返回。但是，已检索值的数据类型可能随存储该值的数据库列的关联的不同而不同。当值存储在列中时，如果其数据类型与列的关联不匹配，则数据库会尝试转换该值以便与列的关联匹配。例如，如果数据库列是用 NUMERIC 关联声明的，则在存储数据之前，数据库会尝试将插入的数据转换为数字存储类（INTEGER 或 REAL）。如果无法转换数据，则数据库将引发错误。按照此规则，如果将字符串“12345”插入到 NUMERIC 列中，则在将它存储在数据库中之前，数据库会自动将它转换为整数值 12345。使用 SELECT 语句检索该值时，它将作为数字数据类型（如 Number）的实例而不是 String 实例返回。

避免不需要的数据类型转换的最佳方式是遵循以下两个规则。首先，使用与其要存储的数据类型匹配的关联定义每个列。其次，仅插入其数据类型与定义的关联匹配的值。遵循这些规则有两个优点。插入数据时，不会意外转换它（结果是可能丢失其预期含义）。此外，检索数据时，它会按其原始数据类型返回。

有关可用的列关联类型以及在 SQL 语句中使用数据类型的详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](#)的附录“[本地数据库中的 SQL 支持](#)”中的“[数据类型支持](#)”一节。

使用同步和异步数据库操作

前面几节已描述常见的数据库操作，如检索、插入、更新和删除数据，以及在数据库中创建数据库文件和表以及其它对象。示例已演示如何以异步和同步方式执行这些操作。

需要提醒的是，在异步执行模式下，您指示数据库引擎执行操作。然后，在应用程序保持运行的同时，数据库引擎在后台工作。当操作完成时，数据库引擎调度事件以提醒您该情况。异步执行的主要优点是，在主应用程序代码继续执行的同时，运行时在后台执行数据库操作。当操作运行所用时间非常长时，这尤其有价值。

另一方面，在同步执行模式下，操作不在后台运行。通知数据库引擎执行操作。代码在数据库引擎工作时暂停。完成操作后，继续执行下一行代码。

使用单个数据库连接，无法同步执行某些操作或语句，同时异步执行其它操作或语句。打开到数据库的连接时，指定 `SQLConnection` 以同步还是异步方式运行。如果调用 `SQLConnection.open()`，则连接以同步执行模式操作；如果调用 `SQLConnection.openAsync()`，则连接以异步执行模式操作。使用 `open()` 或 `openAsync()` 将 `SQLConnection` 实例连接到数据库后，该实例将固定为同步或异步执行。

使用同步数据库操作

与异步执行模式的代码相比，使用同步执行时用于执行和响应操作的实际代码几乎没有差异。两种方法之间的主要差异体现在以下两个方面。首先，执行一个依赖于另一个操作（如 SELECT 结果行或由 INSERT 语句添加的行的主键）的操作。第二方面的差异体现在处理错误上。

为同步操作编写代码

同步执行和异步执行的主要差异在于：在同步模式下，以单个步骤系列的形式编写代码。相反，在异步代码中，注册事件侦听器，并经常在侦听器方法之间分配操作。在同步执行模式下连接数据库时，可以在单个代码块中连续执行一系列数据库操作。以下示例对此技术进行了演示：

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
// open the database
conn.open(dbFile, air.OpenMode.UPDATE);
// start a transaction
conn.begin();
// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";
insertCustomer.execute();
var customerId = insertCustomer.getResult().lastInsertRowID;
// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();
// commit the transaction
conn.commit();
```

如您所看到的，不管使用的是同步执行还是异步执行，都调用相同的方法来执行数据库操作。两种方法的主要差异在于：执行一个依赖于另一个操作的操作和处理错误。

执行一个依赖于另一个操作的操作

使用同步执行模式时，无需编写侦听事件的代码来确定操作完成的时间。相反，可以假定如果一个代码行中的操作成功完成，则继续执行下一代码行。因此，要执行一个依赖于另一个操作成功的操作，只需在紧随它所依赖的操作之后编写相关代码即可。例如，要为应用程序编码以开始事务，可执行 INSERT 语句，检索已插入行的主键，将该主键插入到不同表的其它行中，最后提交事务，可以将代码全部编写为一系列语句。以下示例对这些操作进行了演示：

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
// open the database
conn.open(dbFile, air.SQLMode.UPDATE);
// start a transaction
conn.begin();
// add the customer record to the database
var insertCustomer = new air.SQLStatement();
insertCustomer.sqlConnection = conn;
insertCustomer.text =
    "INSERT INTO customers (firstName, lastName) " +
    "VALUES ('Bob', 'Jones')";
insertCustomer.execute();
var customerId = insertCustomer.getResult().lastInsertRowID;
// add a related phone number record for the customer
var insertPhoneNumber = new air.SQLStatement();
insertPhoneNumber.sqlConnection = conn;
insertPhoneNumber.text =
    "INSERT INTO customerPhoneNumbers (customerId, number) " +
    "VALUES (:customerId, '800-555-1234')";
insertPhoneNumber.parameters[":customerId"] = customerId;
insertPhoneNumber.execute();
// commit the transaction
conn.commit();
```

在同步执行时处理错误

在同步执行模式下，不侦听错误事件来确定操作是否已失败。相反，将可能触发错误的任何代码用一组 `try..catch..finally` 代码块包围起来。将引发错误的代码包装在 `try` 块中。在单独的 `catch` 块中，编写响应每种类型的错误时要执行的操作。在 `finally` 块中放置不管成功还是失败（例如，关闭不再需要的数据库连接）都希望始终执行的任何代码。下面的示例演示使用 `try..catch..finally` 块进行错误处理。它建立在前面示例的基础之上，添加了错误处理代码：

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");
// open the database
conn.open(dbFile, air.SQLMode.UPDATE);
// start a transaction
conn.beginTransaction();
try
{
    // add the customer record to the database
    var insertCustomer = new air.SQLStatement();
    insertCustomer.sqlConnection = conn;
    insertCustomer.text =
        "INSERT INTO customers (firstName, lastName) " +
        "VALUES ('Bob', 'Jones')";

    insertCustomer.execute();
    var customerId = insertCustomer.getResult().lastInsertRowID;

    // add a related phone number record for the customer
    var insertPhoneNumber = new air.SQLStatement();
    insertPhoneNumber.sqlConnection = conn;
    insertPhoneNumber.text =
        "INSERT INTO customerPhoneNumbers (customerId, number) " +
        "VALUES (:customerId, '800-555-1234')";
    insertPhoneNumber.parameters[":customerId"] = customerId;

    insertPhoneNumber.execute();

    // if we've gotten to this point without errors, commit the transaction
    conn.commit();
}
catch (error)
{
    // rollback the transaction
    conn.rollback();
}
```

了解异步执行模式

使用异步执行模式时一个常见的问题是，如果当前在对同一数据库连接执行一个 `SQLStatement`，则假定您无法开始执行另一个 `SQLStatement` 实例。事实上，此假定是不正确的。在 `SQLStatement` 实例执行的同时，无法更改语句的 `text` 属性。但是，如果对要执行的每个不同 SQL 语句都使用单独的 `SQLStatement` 实例，则可以在一个 `SQLStatement` 实例仍执行的同时调用另一个 `SQLStatement` 的 `execute()` 方法，且不会导致错误。

在内部，使用异步执行模式执行数据库操作时，每个数据库连接（每个 `SQLConnection` 实例）都具有自己的队列或指示它执行的操作列表。运行时依次执行每个操作（按照将它们添加到队列的顺序）。创建 `SQLStatement` 实例并调用其 `execute()` 方法时，会将该语句执行操作添加到连接队列。如果在该 `SQLConnection` 实例上当前未执行操作，则语句将在后台开始执行。假定在同一代码块中，创建另一个 `SQLStatement` 实例，并且也调用该方法的 `execute()` 方法。该第二个语句执行操作将添加到队列中的第一个语句之后。在第一个语句完成执行后，运行时立即移动到队列中的下一个操作。队列中后续操作的处理发生在后台，即使在主应用程序代码中调度第一个操作的 `result` 事件也如此。以下代码对此技术进行了演示：

```
// Using asynchronous execution mode
var stmt1 = new air.SQLStatement();
stmt1.sqlConnection = conn;
// ... Set statement text and parameters, and register event listeners ...
stmt1.execute();
// At this point stmt1's execute() operation is added to conn's execution queue.
var stmt2 = new air.SQLStatement();
stmt2.sqlConnection = conn;
// ... Set statement text and parameters, and register event listeners ...
stmt2.execute();
// At this point stmt2's execute() operation is added to conn's execution queue.
// When stmt1 finishes executing, stmt2 will immediately begin executing
// in the background.
```

数据库自动执行排队的后续语句会产生另一个重要效果。如果一个语句依赖于另一个操作的结果，则直到第一个操作完毕，才能将该语句添加到队列中（换句话说，才能调用其 `execute()` 方法）。这是因为调用第二个语句的 `execute()` 方法后，就无法更改该语句的 `text` 或 `parameters` 属性。在此情况下，在开始下一个操作之前，必须等待指示第一个操作已完成的事件。例如，如果要在事务的上下文中执行语句，则该语句的执行将取决于打开事务的操作。调用 `SQLConnection.begin()` 方法打开事务后，需要等待 `SQLConnection` 实例调度其 `begin` 事件。只有这时才能调用 `SQLStatement` 实例的 `execute()` 方法。在此示例中，组织应用程序以确保正确执行操作的最简单方法是，创建一个方法，并将其注册为 `begin` 事件的监听器。调用 `SQLStatement.execute()` 方法的代码放置在该监听器方法中。

对 SQL 数据库使用加密

所有 Adobe AIR 应用程序都共享同一个本地数据库引擎。因此，任何 AIR 应用程序都可以连接到、读取和写入未加密的数据库文件。从 Adobe AIR 1.5 起，AIR 中加入了创建和连接到加密数据库文件的功能。使用加密数据库时，应用程序必须提供正确的加密密钥才能连接到数据库。如果提供的加密密钥有误（或不提供密钥），则应用程序无法连接到数据库。因此，应用程序无法从数据库中读取数据，也无法写入数据库或更改数据库中的数据。

若要使用加密数据库，创建数据库时必须将其创建为加密数据库。有了加密数据库，即可打开到数据库的连接。还可以更改加密数据库的加密密钥。除了创建和连接到加密数据库之外，处理加密数据库的方法与处理未加密数据库的方法相同。尤其是无论数据库是否加密，执行 SQL 语句的方式都相同。

加密数据库的用途

希望限制对数据库中所存储信息的访问时，加密很有帮助。Adobe AIR 的数据库加密功能可以用于多种用途。下面是需要使用加密数据库的一些示例情况：

- 从服务器下载的专用应用程序数据的只读缓存
- 与服务器进行同步（向服务器发送数据以及从服务器加载数据）的专用数据的本地应用程序存储区
- 用作由应用程序创建和编辑的文档的文件格式的加密文件。可以专用于一个用户、或可以在应用程序的所有用户中共享的文件。
- 本地数据存储区的任何其他用途（如第 211 页的“[本地 SQL 数据库的用途](#)”中所述），在这些用途中不能向有权访问计算机或数据库文件的人员公开数据。

了解需要使用加密数据库的原因有助于您确定构建应用程序的方式。尤其是会影响应用程序如何创建、获得或存储数据库的加密密钥。有关这些注意事项的详细信息，请参阅第 237 页的“[对数据库使用加密的注意事项](#)”。

除了加密数据库之外，加密本地存储区作为一种替代机制，也可以保持敏感数据的私密性。在使用加密本地存储区的情况下，可以使用 `String` 密钥存储单个 `ByteArray` 值。只有存储该值的 AIR 应用程序才能对其进行访问，而且只能在存储该值的计算机上进行访问。在采用加密本地存储区的情况下，不需要创建自己的加密密钥。出于这些原因，加密本地存储区最适合于方便地存储易于在 `ByteArray` 中编码的一个值或一组值。加密数据库最适合于需要结构化数据存储和查询的大型数据集。有关使用加密本地存储区的详细信息，请参阅第 246 页的“[存储加密数据](#)”。

创建加密数据库

若要使用加密数据库，则创建数据库文件时必须将其加密。一旦在不加密的情况下创建数据库，以后就无法再对其进行加密。同样，以后也无法对加密数据库进行解密。如有必要，可以更改加密数据库的加密密钥。有关详细信息，请参阅第 236 页的“[更改数据库的加密密钥](#)”。如果现有的数据库未加密，而您又希望使用数据库加密，则可以新建一个加密数据库，然后将现有的表结构和数据复制到新数据库中。

创建加密数据库与创建未加密数据库几乎完全相同，如第 214 页的“[创建数据库](#)”中所述。首先创建一个表示到数据库连接的 `SQLConnection` 实例。通过调用 `SQLConnection` 对象的 `open()` 方法或 `openAsync()` 方法创建数据库，并为数据库位置指定一个尚未存在的文件。创建加密数据库时的唯一区别在于要为 `encryptionKey` 参数（`open()` 方法的第五个参数和 `openAsync()` 方法的第六个参数）提供一个值。有效的 `encryptionKey` 参数值为正好包含 16 个字节的 `ByteArray` 对象。

以下示例介绍创建以异步执行模式打开的加密数据库。为了简单起见，此示例中的加密密钥在应用程序代码中采用硬编码形式。但是，由于此方法不安全，强烈建议不要使用此方法。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);

function openHandler(event)
{
    air.trace("the database was created successfully");
}

function errorHandler(event)
{
    air.trace("Error message:", event.error.message);
    air.trace("Details:", event.error.details);
}
```

以下示例介绍创建以同步执行模式打开的加密数据库。为了简单起见，此示例中的加密密钥在应用程序代码中采用硬编码形式。但是，由于此方法不安全，强烈建议不要使用此方法。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

try
{
    conn.open(dbFile, air.SQLMode.CREATE, false, 1024, encryptionKey);
    air.trace("the database was created successfully");
}
catch (error)
{
    air.trace("Error message:", error.message);
    air.trace("Details:", error.details);
}
```

有关介绍生成加密密钥的建议方式的示例，请参阅第 237 页的“[示例：生成和使用加密密钥](#)”。

连接到加密数据库

与创建加密数据库类似的是，打开到加密数据库的连接所采用的步骤类似于连接到未加密数据库。该步骤在第 216 页的“[连接到数据库](#)”中有更详细的说明。使用 `open()` 方法以同步执行模式打开连接，或使用 `openAsync()` 方法以异步执行模式打开连接。唯一的区别在于，若要打开加密数据库，要为 `encryptionKey` 参数（`open()` 方法的第五个参数和 `openAsync()` 方法的第六个参数）指定正确的值。

如果所提供的加密密钥有误，则会出错。对于 `open()` 方法，将引发 `SQLError` 异常。对于 `openAsync()` 方法，`SQLConnection` 对象将调度 `SQLErrorEvent`，其 `error` 属性包含 `SQLError` 对象。在任一情况下，由异常生成的 `SQLError` 对象的 `errorID` 属性值都为 3138。该错误 ID 对应于错误消息“所打开的文件不是数据库文件”。

以下示例介绍以异步执行模式打开加密数据库。为了简单起见，此示例中的加密密钥在应用程序代码中采用硬编码形式。但是，由于此方法不安全，强烈建议不要使用此方法。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLEvent.ERROR, errorHandler);
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

conn.openAsync(dbFile, air.SQLMode.UPDATE, null, false, 1024, encryptionKey);

function openHandler(event)
{
    air.trace("the database opened successfully");
}

function errorHandler(event)
{
    if (event.error.errorID == 3138)
    {
        air.trace("Incorrect encryption key");
    }
    else
    {
        air.trace("Error message:", event.error.message);
        air.trace("Details:", event.error.details);
    }
}
```

以下示例介绍以同步执行模式打开加密数据库。为了简单起见，此示例中的加密密钥在应用程序代码中采用硬编码形式。但是，由于此方法不安全，强烈建议不要使用此方法。

```
// Include AIRAliases.js to use air.* shortcuts
var conn = new air.SQLConnection();
var dbFile = air.File.applicationStorageDirectory.resolvePath("DBSample.db");

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

try
{
    conn.open(dbFile, air.SQLMode.UPDATE, false, 1024, encryptionKey);
    air.trace("the database was created successfully");
}
catch (error)
{
    if (error.errorID == 3138)
    {
        air.trace("Incorrect encryption key");
    }
    else
    {
        air.trace("Error message:", error.message);
        air.trace("Details:", error.details);
    }
}
```

有关介绍生成加密密钥的建议方式的示例，请参阅第 237 页的“[示例：生成和使用加密密钥](#)”。

更改数据库的加密密钥

数据库加密后，可以在以后更改数据库的加密密钥。若要更改数据库的加密密钥，请首先创建 **SQLConnection** 实例并调用其 **open()** 或 **openAsync()** 方法，从而打开与数据库的连接。连接数据库后，调用 **reencrypt()** 方法，并传递新的加密密钥作为参数。

与大多数数据库操作类似的是，**reencrypt()** 方法的行为根据数据库连接使用同步还是异步执行模式而有所不同。如果使用 **open()** 方法连接到数据库，则 **reencrypt()** 操作同步运行。操作完成后，继续执行下一行代码：

```
var newKey = new air.ByteArray();
// ... generate the new key and store it in newKey
conn.reencrypt(newKey);
```

另一方面，如果使用 **openAsync()** 方法打开数据库连接，则 **reencrypt()** 操作为异步方式。调用 **reencrypt()** 将开始重新加密的过程。操作完成后，**SQLConnection** 对象调度一个 **reencrypt** 事件。使用事件侦听器确定重新加密完成的时间：

```
var newKey = new air.ByteArray();
// ... generate the new key and store it in newKey
conn.addEventListener(air.SQLEvent.REENCRYPT, reencryptHandler);
conn.reencrypt(newKey);

function reencryptHandler(event)
{
    // save the fact that the key changed
}
```

reencrypt() 操作在其自身的事务中运行。如果操作中断或失败（例如，如果在操作完成之前关闭应用程序），则事务将回滚。在这种情况下，原始的加密密钥仍为数据库的加密密钥。

reencrypt() 方法不能用于解除对数据库的加密。向 **reencrypt()** 方法传递 **null** 值或非 16 字节 **ByteArray** 的加密密钥会导致错误。

对数据库使用加密的注意事项

第 233 页的“[加密数据库的用途](#)”部分介绍了需要使用加密数据库的几种情况。显然，不同应用程序的使用情况（包括以上这些情况和其他情况）具有不同的隐私要求。如何安排加密在应用程序中的用途，对于控制数据库的数据私密程度起着重要作用。例如，如果使用加密数据库来保持个人数据的私密性（甚至针对同一计算机的其他用户），则每个用户的数据库都需要有自己的加密密钥。为尽可能的安全起见，应用程序可以根据用户输入的密码生成密钥。加密密钥以密码为基础可以确保，即使其他人可以在计算机上模拟用户的帐户，也无法访问数据。换个角度来看隐私问题，假设您希望数据库文件可以由您的应用程序的任何用户读取，但不能由其他应用程序读取。在这种情况下，每个已安装的应用程序副本都需要具有访问共享加密密钥的权限。

可以根据希望应用程序数据所达到的隐私等级来设计应用程序，尤其是用于生成加密密钥的方法。以下列表为各种级别的数据隐私提供了设计建议：

- 若要使任何计算机上有权访问应用程序的任何用户都可以访问数据库，请使用一个密钥，该密钥对于应用程序的所有实例都可用。例如，应用程序首次运行时，可以使用一个安全协议（如 SSL）从服务器下载共享的加密密钥。然后可以将密钥保存在加密本地存储区中，以供将来使用。作为替代方法，可以按计算机上的每个用户对数据进行加密，然后将数据与远程数据存储区（如服务器）同步，以使数据可移植。
- 若要使任何计算机上的单个用户都可以访问数据库，请根据用户的保密事项（如密码）生成加密密钥。尤其是不要使用任何与特定计算机关联的值（如存储在加密本地存储区中的值）生成密钥。作为替代方法，可以按计算机上的每个用户对数据进行加密，然后将数据与远程数据存储区（如服务器）同步，以使数据可移植。
- 若要使单个计算机上的单个人可以访问数据库，请根据密码和所生成的 salt 来生成密钥。有关此方法的示例，请参阅第 237 页的“[示例：生成和使用加密密钥](#)”。

设计应用程序使用加密数据库时，还有一些安全注意事项务必要牢记，具体如下所示：

- 系统的安全性取决于其最薄弱环节的安全性。如果使用用户输入的密码生成加密密钥，则请考虑对密码施加最小长度和复杂性的限制。只使用基本字符的短密码很快就会被猜中。
- AIR 应用程序的源代码以纯文本形式（对于 HTML 内容）或易于反编译的二进制格式（对于 SWF 内容）存储在用户的计算机上。由于源代码可访问，因此有两点要牢记：
 - 切勿在源代码中对加密密钥进行硬编码
 - 始终假设用于生成加密密钥的方法（如随机字符生成器或特定的哈希算法）很容易就会被攻击者破解
- AIR 数据库加密使用高级加密标准 (AES) 及 Counter with CBC-MAC (CCM) 模式。这种加密密码需要将用户输入的密钥与 salt 值组合在一起才安全。有关此方法的示例，请参阅第 237 页的“[示例：生成和使用加密密钥](#)”。
- 如要加密数据库，则数据库引擎所使用的所有磁盘文件与该数据库一起都要进行加密。但是，在事务过程中，数据库引擎会在内存中的缓存中临时保留一些数据，以提高读写性能。驻留在内存中的任何数据都不加密。如果攻击者可以访问 AIR 应用程序所使用的内存（例如通过使用调试器），则数据库中当前公开且未加密的数据即可供其使用。

示例：生成和使用加密密钥

此示例应用程序介绍生成加密密钥的一种方法。此应用程序旨在为用户的数据提供最高级别的隐私和安全。保障私有数据安全的一个重要原则是：在应用程序每次连接到数据库时都要求用户输入密码。因此，正如此例所示，需要此种保密级别的应用程序在任何时候都不应直接存储数据库加密密钥。

应用程序由两部分组成：生成加密密钥的 ActionScript 类（EncryptionKeyGenerator 类），以及介绍如何使用该类的基础用户界面。有关完整的源代码，请参阅第 240 页的“[用于生成和使用加密密钥的完整示例代码](#)”。

使用 EncryptionKeyGenerator 类获得安全加密密钥

第 239 页的“[了解 EncryptionKeyGenerator 类](#)”一节详细介绍了 EncryptionKeyGenerator 类为数据库生成加密密钥时使用的技术。但是，在应用程序中使用 EncryptionKeyGenerator 类并不要求理解这些详细信息。

请按照以下步骤在应用程序中使用 `EncryptionKeyGenerator` 类：

- 1 下载 `EncryptionKeyGenerator` 库。`EncryptionKeyGenerator` 类包括在[开源 ActionScript 3.0 核心库 \(as3corelib\)](#) 项目中。可以下载[包括源代码和文档的 as3corelib 包](#)。还可以从项目页面下载 SWC 或源代码文件。
- 2 从 SWC 中提取 SWF 文件。若要提取 SWF 文件，请将 SWC 文件的扩展名改为 “.zip”，然后打开该 ZIP 文件。从 ZIP 文件中提取 SWF 文件，将其放在应用程序源代码可以找到的地方。例如，可以将其放在包含应用程序主 HTML 文件的文件夹中。如果愿意，可以重命名 SWF 文件。在本例中，将 SWF 文件命名为 “`EncryptionKeyGenerator.swf`”。
- 3 在应用程序源代码中，添加链接到 SWF 文件的 `<script>` 块，从而导入 SWF 代码库。第 54 页的“[在 HTML 页中使用 ActionScript 库](#)”中介绍了这种技术。以下代码使 SWF 文件可用作代码库：

```
<script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"/>
```

默认情况下，类的形式为代码 `window.runtime` 后跟完整的包名和类名。对于 `EncryptionKeyGenerator`，完整名称为：

```
window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator
```

可以为类创建别名，避免键入全名带来的繁琐。以下代码创建别名 `ekg.EncryptionKeyGenerator`，表示 `EncryptionKeyGenerator` 类：

```
var ekg;
if (window.runtime)
{
    if (!ekg) ekg = {};
    ekg.EncryptionKeyGenerator = window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
}
```

- 4 在创建数据库或与其打开连接的代码位置之前添加一段代码，通过调用 `EncryptionKeyGenerator()` 构造函数创建 `EncryptionKeyGenerator` 实例。

```
var keyGenerator = new ekg.EncryptionKeyGenerator();
```

- 5 从用户处获取密码：

```
var password = passwordInput.value;

if (!keyGenerator.validateStrongPassword(password))
{
    // display an error message
    return;
}
```

`EncryptionKeyGenerator` 实例使用此密码作为加密密钥的基础（下一步中介绍）。`EncryptionKeyGenerator` 实例对照特定的强密码验证要求测试该密码。如果验证失败，则发生错误。如示例代码所示，可以通过调用 `EncryptionKeyGenerator` 对象的 `validateStrongPassword()` 方法提前检查密码。这样可以确定密码是否符合强密码的最低要求，从而避免出错。

- 6 根据密码生成加密密钥：

```
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

`getEncryptionKey()` 方法生成并返回加密密钥（16 字节的 `ByteArray`）。然后即可使用该加密密钥创建新的加密数据库，或打开现有的加密数据库。

`getEncryptionKey()` 方法有一个必要的参数，即第 5 步中获取的密码。

注：为使数据获得最大程度的安全性和保密性，应用程序必须在每次连接到数据库时都要求用户输入密码。请勿直接存储用户的密码或数据库加密密钥。这样做将面临安全风险。正如本例所示，应用程序在创建加密数据库时和以后连接到该数据库时，应该使用相同的技术根据密码派生加密密钥。

`getEncryptionKey()` 方法还接受第二个（可选）参数，即 `overrideSaltELSKey`。`EncryptionKeyGenerator` 创建一个随机值（称为 `salt`），将其用作加密密钥的一部分。为了能够重新创建加密密钥，`salt` 值存储在 AIR 应用程序的加密本地存储区（ELS）中。默认情况下，`EncryptionKeyGenerator` 类使用特定的字符串作为 ELS 密钥。虽然可能性不大，但该密钥有可能与应用程序使用的另一个密钥发生冲突。您可能希望指定自己的 ELS 密钥，而不使用默认密钥。在这种情况下，请指定自定义密钥，方法是传递它作为第二个 `getEncryptionKey()` 参数，如下所示：

```
var customKey = "My custom ELS salt key";
var encryptionKey = keyGenerator.getEncryptionKey(password, customKey);
```

7 创建或打开数据库

通过由 `getEncryptionKey()` 方法返回的加密密钥，代码可以创建新的加密数据库，或尝试打开现有的加密数据库。在这两种情况下，都使用 `SQLConnection` 类的 `open()` 或 `openAsync()` 方法，如第 234 页的“[创建加密数据库](#)”和第 235 页的“[连接到加密数据库](#)”所述。

在此示例中，应用程序旨在以异步执行模式打开数据库。代码设置相应的事件侦听器，并调用 `openAsync()` 方法，同时传递加密密钥作为最终参数：

```
conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, openError);

conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
```

在侦听器方法中，代码取消了事件侦听器的注册。然后显示状态消息，表明是创建、打开了数据库，还是发生了错误。这些事件处理函数中，最值得注意的部分是 `openError()` 方法。在该方法中，有一个 `if` 语句检查数据库是否存在（意味着代码正在尝试连接到现有的数据库），以及错误 ID 是否与常量

`EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID` 匹配。如果这两个条件都符合，则可能表示用户输入的密码有误。（也有可能表示指定的文件根本不是数据库文件。）以下是用于检查错误 ID 的代码：

```
if (!createNewDB && event.error.errorID == ekg.EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
{
    statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
}
else
{
    statusMsg.innerHTML = "<p class='error'>Error creating or opening database.</p>";
}
```

有关示例事件侦听器的完整代码，请参阅第 240 页的“[用于生成和使用加密密钥的完整示例代码](#)”。

了解 `EncryptionKeyGenerator` 类

不必了解 `EncryptionKeyGenerator` 类的内部工作机制，也可以使用它为应用程序数据库创建安全加密密钥。第 237 页的“[使用 `EncryptionKeyGenerator` 类获得安全加密密钥](#)”中介绍了使用该类的过程。但是，您会发现非常值得了解该类使用的技术。例如，对于需要不同数据保密级别的场合，可能要改编该类或加入它的某些技术。

`EncryptionKeyGenerator` 类包括在[开源 ActionScript 3.0 核心库 \(as3corelib\)](#) 项目中。可以下载 `as3corelib` 包，其中包括[源代码](#)和[文档](#)。还可以在项目站点查看源代码，或将其下载以按照介绍进行操作。

代码创建 `EncryptionKeyGenerator` 实例并调用其 `getEncryptionKey()` 方法时，将采取若干步骤来确保只有正当的用户才能访问数据。此过程与创建数据库之前，根据用户输入的密码生成加密密钥的过程相同，亦与重新创建加密密钥，以打开数据库的过程相同。

获取并验证强密码

代码调用 `getEncryptionKey()` 方法时，将传入密码作为参数。密码用作加密密钥的基础。此设计使用只有用户知道的一条信息，从而确保只有知道密码的用户才能访问数据库中的数据。即使攻击者可以访问用户在计算机上的帐户，在不知道密码的情况下也无法进入数据库。为尽可能安全起见，应用程序从不存储密码。

在示例应用程序中，`passwordInput` 是用户从中输入密码的 HTML `<input>` 元素的 ID。应用程序并非直接操作元素的值，而是将密码复制到名为 `password` 的变量中。

```
var password = passwordInput.value;
```

然后，示例应用程序创建 `EncryptionKeyGenerator` 实例，并调用其 `getEncryptionKey()` 方法，同时使用 `password` 变量作为参数：

```
var keyGenerator = new ekg.EncryptionKeyGenerator();
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

调用 `getEncryptionKey()` 方法后 `EncryptionKeyGenerator` 类所采取的第一步为检查用户输入的密码，以确保其符合密码强度的要求。本例中，密码的长度必须为 8 到 32 个字符。必须包含大小写字母的混合形式以及至少一个数字或符号字符。

将密码扩展到 256 位

在过程的后期，密码的长度必须为 256 位。代码并不要求每个用户输入长度刚好为 256 位（32 个字符）的密码，而是通过重复密码字符来创建更长的密码。

以下是 `concatenatePassword()` 方法的代码：

如果密码长度小于 256 位，则代码将密码与密码自身连接在一起，使其达到 256 位。如果不能刚好达到这一长度，则缩短最后一次重复的内容，以便刚好得到 256 位。

生成或检索 256 位 salt 值

下一步是获得 256 位 salt 值，后面的一个步骤中会将此值与密码组合在一起。`salt` 是向用户输入的值添加或与之组合在一起而形成密码的一个随机值。结合使用 `salt` 和密码确保了即使用户选择真实单词或常用词汇作为密码，系统所使用的“密码加 `salt`”组合也是一个随机值。这有助于防御字典攻击，攻击者在字典攻击中使用单词列表尝试猜出密码。此外，通过生成 `salt` 值并将其存储在加密本地存储区中，该值与数据库文件所在计算机上的用户帐户相关联。

如果应用程序是第一次调用 `getEncryptionKey()` 方法，则代码将创建一个随机的 256 位 `salt` 值。否则，代码从加密本地存储区加载 `salt` 值。

使用 XOR 运算符组合 256 位密码和 salt

代码现在拥有一个 256 位密码和一个 256 位 `salt` 值。然后，代码使用按位 XOR 运算将 `salt` 和连接而成的密码组合为一个值。实际上，这种方法创建的 256 位密码由整个可用字符范围内的字符组成。即使实际输入的密码主要由字母数字字符组成也是如此。如此提高随机性的优点在于：无须用户输入长而复杂的密码，即可使可能密码的集合变得非常大。

对密钥进行哈希处理

将连接而成的密码与 `salt` 组合在一起后，下一步就是进一步加强对此值的保护，具体而言就是使用 SHA-256 哈希算法对此值进行哈希处理。对值进行哈希处理使攻击者更难以对其进行反向工程。

从哈希值提取加密密钥

加密密钥必须为刚好 16 个字节（128 位）长的 `ByteArray`。SHA-256 哈希算法的结果的长度始终为 256 位。因此，最后一步是从哈希处理的结果中选择 128 位作为实际的加密密钥。

并不一定要使用前 128 位作为加密密钥。可以选择从某任意点开始的一系列位，可以每隔一位选择一位，或使用某些其它方式来选择位。重要的是代码选择 128 个不同的位，并且每次都使用相同的 128 位。

用于生成和使用加密密钥的完整示例代码

以下是示例应用程序“生成和使用加密密钥”的完整代码。代码由两部分组成。

该示例使用 `EncryptionKeyGenerator` 类根据密码创建加密密钥。`EncryptionKeyGenerator` 类包括在[开源 ActionScript 3.0 核心库 \(as3corelib\) 项目](#)中。可以下载[包括源代码和文档的 as3corelib 包](#)。还可以从项目页面下载 SWC 或源代码文件。

应用程序 HTML 文件包含一个简单应用程序的源代码，该应用程序创建加密数据库或打开到加密数据库的连接：

```
<html>
    <head>
        <title>Encrypted Database Example (HTML)</title>
        <style type="text/css">
            body
            {
                padding-top: 25px;
                font-family: Verdana, Arial;
                font-size: 14px;
            }
            div
            {
                width: 85%;
                margin-left: auto;
                margin-right: auto;
            }
            .error {color: #990000}
            .success {color: #009900}
        </style>

        <script type="text/javascript" src="AIRAliases.js"></script>
        <script type="application/x-shockwave-flash" src="EncryptionKeyGenerator.swf"/>
        <script type="text/javascript">
            // set up the class shortcut
            var ekg;
            if (window.runtime)
            {
                if (!ekg) ekg = {};
                ekg.EncryptionKeyGenerator = window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
            }

            // app globals
            var dbFileName = "encryptedDatabase.db";
            var dbFile;
            var createNewDB = true;
            var conn;

            // UI elements
            var instructions;
            var passwordInput;
            var openButton;
            var statusMsg;

            function init()
            {
                // UI elements
                instructions = document.getElementById("instructions");
                passwordInput = document.getElementById("passwordInput");
                openButton = document.getElementById("openButton");
                statusMsg = document.getElementById("statusMsg");

                conn = new air.SQLConnection();
                dbFile = air.File.applicationStorageDirectory.resolvePath(dbFileName);
                if (dbFile.exists)
                {
                    createNewDB = false;
                    instructions.innerHTML = "<p>Enter your database password to open the encrypted database.</p>";
                    openButton.value = "Open Database";
                }
            }
        </script>
    </head>
    <body>
        <div>
            <form>
                <label>Enter your database password</label>
                <input type="password" id="passwordInput" />
                <input type="button" value="Open Database" id="openButton" />
            </form>
            <div id="statusMsg"></div>
        </div>
    </body>
</html>
```

```
function openConnection()
{
    var keyGenerator = new ekg.EncryptionKeyGenerator();

    var password = passwordInput.value;

    if (password == null || password.length <= 0)
    {
        statusMsg.innerHTML = "<p class='error'>Please specify a password.</p>";
        return;
    }

    if (!keyGenerator.validateStrongPassword(password))
    {
        statusMsg.innerHTML = "<p class='error'>The password must be 8-32 characters long. It
must contain at least one lowercase letter, at least one uppercase letter, and at least one number or
symbol.</p>";
        return;
    }

    passwordInput.value = "";
    passwordInput.disabled = true;
    openButton.disabled = true;
    statusMsg.innerHTML = "";

    var encryptionKey = keyGenerator.getEncryptionKey(password);

    conn.addEventListener(air.SQLEvent.OPEN, openHandler);
    conn.addEventListener(air.SQLErrorEvent.ERROR, openError);

    conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);
}

function openHandler(event)
{
    conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
    conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

    if (createNewDB)
    {
        statusMsg.innerHTML = "<p class='success'>The encrypted database was created
successfully.</p>";
    }
    else
    {
        statusMsg.innerHTML = "<p class='success'>The encrypted database was opened
successfully.</p>";
    }
}

function openError(event)
{
    conn.removeEventListener(air.SQLEvent.OPEN, openHandler);
    conn.removeEventListener(air.SQLErrorEvent.ERROR, openError);

    if (!createNewDB && event.error.errorID ==
```

```
ekg.EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
{
    statusMsg.innerHTML = "<p class='error'>Incorrect password!</p>";
}
else
{
    statusMsg.innerHTML = "<p class='error'>Error creating or opening database.</p>";
}
}
</script>
</head>

<body onload="init()">
<div id="instructions"><p>Enter a password to create an encrypted database. The next time you open the application, you will need to re-enter the password to open the database again.</p></div>
<div><input id="passwordInput" type="password"/><input id="openButton" type="button" value="Create Database" onclick="openConnection();"/></div>
<div id="statusMsg"></div>
</body>
</html>
```

使用 SQL 数据库的策略

应用程序可以通过各种方式来访问和使用本地 SQL 数据库。应用程序设计可能随应用程序代码的组织方式、操作执行方式的序列和计时等的不同而不同。所选技术可能影响开发应用程序的难易程度。它们可能影响在将来的更新中修改应用程序的难易程度。它们还可能影响从用户的角度看应用程序性能的高低。

分发预填充的数据库

在应用程序中使用 AIR 本地 SQL 数据库时，应用程序期望一个特定结构的表、列等的数据库。某些应用程序还期望在数据库文件中预填充特定数据。确保数据库具有正确结构的一种方法是，在应用程序代码中创建数据库。应用程序在加载时将检查在特定位置中是否存在其数据库文件。如果该文件不存在，则应用程序将执行一组命令来创建数据库文件，创建数据库结构并用初始数据填充表。

创建数据库及其表的代码常常很复杂。在应用程序的安装生存期内，它通常仅使用一次，但是仍增加了应用程序的大小和复杂性。作为以编程方式创建数据库、结构和数据的一种替代方法，可以随应用程序分发预填充的数据库。要分发预定义的数据库，请将数据库文件包括在应用程序的 AIR 包中。

与 AIR 包中包括的所有文件一样，捆绑的数据库文件安装在应用程序目录（由 `File.applicationDirectory` 属性表示的目录）中。但是，该目录中的文件是只读的。将 AIR 包中的文件用作“模板”数据库。用户首次运行应用程序时，会将原始数据库文件复制到该用户的应用程序存储目录（或其它位置）中，并在应用程序中使用该数据库。

提高数据库性能

通过内置于 Adobe AIR 中的几项技术，可以提高应用程序中数据库操作的性能。

除了此处描述的技术外，编写 SQL 语句的方法也影响数据库性能。通常，编写用于检索特定结果集的 SQL SELECT 语句的方法有多种。在某些情况下，不同的方法要求数据库引擎所做的工作或多或少。在 Adobe AIR 文档中不包含提高数据库性能（设计 SQL 语句以获得更佳性能）这一方面。

对每个 SQL 语句使用一个 SQLStatement 实例

在执行任何 SQL 语句之前，运行时会准备（编译）该语句，确定在内部执行的步骤并执行语句。在以前未执行的 SQLStatement 实例上调用 `SQLStatement.execute()` 时，在执行该实例之前将自动准备语句。在对 `execute()` 方法的后续调用中，只要 `SQLStatement.text` 属性未更改，就仍将准备语句。因此，它的执行速度更快。

若要充分发挥重用语句的优势，如果值在语句执行之间需要更改，请使用语句参数自定义您的语句。（语句参数是使用 `SQLStatement.parameters` 关联数组属性指定的。）与更改 `SQLStatement` 实例的 `text` 属性不同，如果更改语句参数的值，则不要求运行时再次准备语句。有关在语句中使用参数的详细信息，请参阅第 218 页的“[在语句中使用参数](#)”。

由于准备和执行语句是一项潜在要求很高的操作，因此一种好的策略是预加载初始数据，然后在后台执行其它语句。首先加载应用程序所需的数据。应用程序的初始启动操作完成时，或者在应用程序中的其它“空闲”时间执行其它语句。例如，如果应用程序为显示其初始屏幕根本未访问数据库，请等到该屏幕显示，然后打开数据库连接，最后创建 `SQLStatement` 实例并执行可以执行的任何操作。或者，假定应用程序在启动时就立即显示某些数据，如特定查询的结果。在该情况下，继续操作并执行该查询的 `SQLStatement` 实例。加载并显示初始数据后，为其它数据库操作创建 `SQLStatement` 实例，如有可能，则执行稍后需要的其它语句。

重用 `SQLStatement` 实例时，当准备 `SQLStatement` 实例后，应用程序需要保存对它的引用。要保存对该实例的引用，请将变量声明为类范围的变量而不是函数范围的变量。执行此操作的一种好方法是，构造应用程序将 SQL 语句包装在单个类中。也可以将在组合中执行的一组语句包装在单个类中。通过将一个或多个 `SQLStatement` 实例定义为类的成员变量，只要包装类的实例存在于应用程序中，它们就会一直存在。在最低限度下，只需在函数之外定义一个包含 `SQLStatement` 实例的变量，使该实例保持在内存中。例如，将 `SQLStatement` 实例声明为 ActionScript 类中的成员变量，或声明为 JavaScript 文件中的非函数变量。然后可以设置语句的参数值，并在要实际运行查询时调用其 `execute()` 方法。

在一个事务中组合多个操作

假定要执行涉及添加或更改数据的大量 SQL 语句（`INSERT` 或 `UPDATE` 语句）。通过在显式事务内执行所有语句，可以大大提高性能。如果不显式开始事务，则每个语句都运行在它自己的自动创建的事务中。每个事务（每个语句）完成执行后，运行时会生成的数据写入磁盘上的数据库文件。另一方面，应考虑显式创建事务并在该事务的上下文中执行语句时将发生的情况。运行时在内存中进行所有更改，然后在提交事务时将所有更改一次写入数据库文件。将数据写入磁盘通常是操作中最耗时的部分。因此，一次性写入磁盘而不是对每个 SQL 语句写入一次可以大大提高性能。

将运行时处理减到最少

使用以下技术，可以防止数据库引擎部分执行不需要的操作并提高应用程序的性能：

- 始终在语句中显式指定数据库名称以及表名。（如果是主数据库，请使用“`main`”）。例如，使用 `SELECT employeeId FROM main.employees` 而非 `SELECT employeeId FROM employees`。显式指定数据库名称，运行时就不必检查每个数据库来查找匹配表。这样做还消除了使运行时选择错误数据库的可能性。即使 `SQLConnection` 仅连接到单个数据库，也要遵循此规则，因为在后台 `SQLConnection` 还连接到可通过 SQL 语句访问的临时数据库。
- 始终在 `SELECT` 或 `INSERT` 语句中显式指定列名。
- 对于检索大量行的 `SELECT` 语句，分解它所返回的行：请参阅第 223 页的“[检索部分 SELECT 结果](#)”。

避免架构更改

如有可能，在向数据库的表中添加数据后，避免更改数据库的架构（表结构）。通常，数据库文件是使用该文件开头的表定义构建的。打开到数据库的连接时，运行时将加载这些定义。向数据库表添加数据时，会将该数据添加到文件中，并放置在表定义数据之后。但是，如果进行架构更改（如向表中添加列或者添加新表），则新的表定义数据将与数据库文件中的表数据相混合。如果表定义数据不都在数据库文件的开头，则打开到数据库的连接需要较长的时间，因为运行时要从文件的不同部分读取表定义数据。

如果确实需要进行架构更改，则可以在完成更改后调用 `SQLConnection.compact()` 方法。此操作将重新结建数据库文件，以便表定义数据全部位于文件的开头。但是，`compact()` 操作可能需要大量的时间，尤其当数据库文件越来越大时。

使用本地 SQL 数据库的最佳做法

下面列出了一组建议的技术，在使用本地 SQL 数据库时，可以通过这些技术提高应用程序的性能、安全性和易维护性。有关改进数据库应用程序的其它技术，请参阅第 243 页的“[提高数据库性能](#)”。

预创建数据库连接

即使应用程序在首次加载时不执行任何语句，也要提前（如在应用程序初始启动之后）实例化 `SQLConnection` 对象并调用其 `open()` 或 `openAsync()` 方法，以避免在运行语句时产生延迟。请参阅第 216 页的“[连接到数据库](#)”。

重用数据库连接

如果在应用程序的整个执行时间内访问某个数据库，请保存对 `SQLConnection` 实例的引用，并在整个应用程序中重用它，而不是先关闭再重新打开连接。请参阅第 216 页的“[连接到数据库](#)”。

推荐使用异步执行模式

编写数据访问代码时，可能很想同步执行操作而不是异步执行，因为使用同步操作通常需要更短的代码，并且代码的复杂性更低。但是，如第 229 页的“[使用同步和异步数据库操作](#)”中所述，同步操作可产生对用户而言很明显的性能影响，并损害用户对应用程序的体验。单个操作所用的时间随操作、尤其是它所涉及的数据量的不同而不同。例如，仅向数据库中添加一行的 `SQL INSERT` 语句所用的时间要比检索成千上万行数据的 `SELECT` 语句所用的时间少。但是，使用同步执行来执行多个操作时，这些操作通常串在一起。即使每个操作所用的时间非常短，但是在所有同步操作完成之前会冻结应用程序。因此，串在一起的多个操作的累积时间可能足以停止应用程序。

将异步操作用作一种标准方法，尤其是对于涉及大量行的操作。有一种技术可拆分大型 `SELECT` 语句结果集的处理，如第 223 页的“[检索部分 SELECT 结果](#)”所述。但是，此技术只能在异步执行模式下使用。只有在使用异步编程无法实现某些功能时，已考虑到应用程序的用户所要面临的性能折衷时，以及已测试应用程序以便了解对应用程序性能的影响程度时，才应使用同步操作。使用异步执行可能涉及更复杂的编码。但是，请记住只需编写一次代码，但是应用程序用户必须重复使用它（或快或慢）。

在许多情况下，通过对要执行的每个 SQL 语句使用单独的 `SQLStatement` 实例，可以同时将多个 SQL 操作排队，这将使异步代码在代码编写方式上与同步代码类似。有关详细信息，请参阅第 232 页的“[了解异步执行模式](#)”。

使用单独的 SQL 语句，且不更改 `SQLStatement` 的 `text` 属性

对于在应用程序中多次执行的任何 SQL 语句，为每个 SQL 语句创建单独的 `SQLStatement` 实例。SQL 命令在每次执行时都使用该 `SQLStatement` 实例。例如，假定您要生成一个应用程序，它包括四个多次执行的不同 SQL 操作。在此情况下，创建四个单独的 `SQLStatement` 实例，并调用每个语句的 `execute()` 方法来运行它。避免对所有 SQL 语句使用单个 `SQLStatement` 实例，每次执行语句之前都重新定义其 `text` 属性。有关详细信息，请参阅第 244 页的“[对每个 SQL 语句使用一个 SQLStatement 实例](#)”。

使用语句参数

使用 `SQLStatement` 参数 — 从不将用户输入连接到语句文本中。使用参数会使应用程序更安全，因为这样可消除 SQL 注入攻击的可能性。这样就有可能在查询中使用对象（而不是仅使用 SQL 字面值）。这样还会提高语句的运行效率，因为可以重用语句，每次执行它们时无需将其重新编译。有关详细信息，请参阅第 218 页的“[在语句中使用参数](#)”。

对列名和参数名使用常数

在没有为 `SQLStatement` 指定 `itemClass` 时，为避免拼写错误，请定义包含表的列名的 `String` 常数。从结果对象中检索值时，在语句文本中使用这些常数以及对属性名称使用它们。此外还对参数名称使用常数。

第 26 章：存储加密数据

Adobe® AIR® 运行时为安装在用户计算机上的每个 AIR 应用程序都提供一个永久性的加密本地存储区。这样，您就可以以加密格式保存和检索存储在用户本地硬盘驱动器上的数据，使其他应用程序或用户不能轻易地对这些数据进行解密。对每个 AIR 应用程序使用一个单独的加密本地存储区，且每个 AIR 应用程序对每个用户使用一个单独的加密本地存储区。

注：除了加密本地存储区之外，AIR 还可以对 SQL 数据库中存储的内容进行加密。有关详细信息，请参阅第 233 页的“[对 SQL 数据库使用加密](#)”。

您可能想使用加密的本地存储区来存储需要得到很好保护的信息，如用于 Web 服务的登录凭据。

通过在 Windows 中使用 DPAPI，在 Mac OS 中使用 KeyChain，以及在 Linux 中使用 KeyRing 或 KWallet，AIR 将加密本地存储区与每个应用程序和用户相关联。加密的本地存储区使用 AES-CBC 128 位加密。

加密的本地存储区中的信息仅可用于应用程序安全沙箱中的 AIR 应用程序内容。

使用 `EncryptedLocalStore` 类的 `setItem()` 和 `removeItem()` 静态方法可以将数据存储在本地存储区中以及从中检索数据。数据存储在哈希表中（使用字符串作为键，以字节数组的形式存储数据）。

例如，下面的代码将一个字符串存储在加密的本地存储区中：

```
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes);

var storedValue = air.EncryptedLocalStore.getItem("firstName");
air.trace(storedValue.readUTFBytes(storedValue.length)); // "foo"
```

`setItem()` 方法的第三个参数（即 `stronglyBound` 参数）是可选参数。当此参数设置为 `true` 时，加密的本地存储区通过将存储的项目绑定到执行存储的 AIR 应用程序的数字签名和位以及该应用程序的发行商 ID，提供更高级别的安全性：

```
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes, true);
```

对于在存储时将 `stronglyBound` 设置为 `true` 的项目，以后对 `getItem()` 的调用仅在执行调用的 AIR 应用程序与执行存储的应用程序相同时才会成功（前提是应用程序目录中的文件未发生数据更改）。如果执行调用的 AIR 应用程序与执行存储的应用程序不同，则当您对强绑定项目调用 `getItem()` 时，该应用程序将引发 `Error` 异常。如果您更新应用程序，则该应用程序将无法读取先前写入到加密的本地存储区中的强绑定数据。

如果将 `stronglyBound` 参数设置为 `false`（默认值），则只有发行商 ID 需要保持不变，供应用程序读取数据。应用程序的位数可能会更改（且它们需要发行商签名），但它们无需与存储数据的应用程序的位数保持完全相同。

默认情况下，AIR 应用程序无法读取其他应用程序的加密本地存储区。`stronglyBound` 设置提供额外绑定（到应用程序位中的数据），以阻止攻击应用程序通过尝试劫持应用程序发行商 ID 来读取应用程序加密本地存储区的企图。

如果您将应用程序更新为使用其他签名证书（使用迁移签名），则即使 `stronglyBound` 参数设置为 `false`，更新版本也无法访问原始存储区中的任何项目。有关详细信息，请参阅第 315 页的“[更改证书](#)”。

可以使用 `EncryptedLocalStore.removeItem()` 方法删除加密本地存储区中的值，如下例所示：

```
air.EncryptedLocalStore.removeItem("firstName");
```

可以通过调用 `EncryptedLocalStore.reset()` 方法清除加密本地存储区中的所有数据，如下例所示：

```
air.EncryptedLocalStore.reset();
```

在 AIR Debug Launcher (ADL) 中调试应用程序时，该应用程序使用的加密本地存储区不同于已安装的应用程序版本所使用的加密本地存储区。

如果存储的数据超过 10MB，则加密的本地存储区的运行速度可能变慢。

当卸载 AIR 应用程序时，卸载程序不会删除存储在加密的本地存储区中的数据。

加密的本地存储区数据存放在用户的应用程序数据目录的子目录中，该子目录的路径为 `Adobe/AIR/ELS/` (后跟应用程序 ID)。

第 27 章：添加 PDF 内容

Adobe® AIR® 中运行的应用程序不仅可以呈现 SWF 和 HTML 内容，而且还能呈现 PDF 内容。AIR 应用程序使用 HTMLLoader 类、WebKit 引擎和 Adobe® Reader® 浏览器插件来呈现 PDF 内容。在 AIR 应用程序中，PDF 内容可以沿应用程序的全高和全宽进行拉伸，也可以作为界面的一部分。Adobe Reader 浏览器插件控制 PDF 文件在 AIR 应用程序中的显示，因此，对 Reader 工具栏界面所做的修改（例如对位置、定位和可见性的修改）不会影响随后在 AIR 应用程序和浏览器中查看 PDF 文件。

重要说明：为了在 AIR 中呈现 PDF 内容，用户必须安装 Adobe Reader 或 Adobe® Acrobat® 8.1 或更高版本。

检测 PDF 功能

如果用户未安装 Adobe Reader 或 Adobe Acrobat 8.1 或更高版本，则无法在 AIR 应用程序中显示 PDF 内容。若要检测用户是否能够呈现 PDF 内容，请首先检查 `HTMLLoader.pdfCapability` 属性。此属性设置为 `HTMLPDFCapability` 类的以下常量之一：

常量	说明
<code>HTMLPDFCapability.STATUS_OK</code>	已检测到足够高的 Adobe Reader 版本（8.1 或更高版本），可以将 PDF 内容加载到 <code>HTMLLoader</code> 对象中。
<code>HTMLPDFCapability.ERROR_INSTALLED_READER_NOT_FOUND</code>	未检测到任何 Adobe Reader 版本。 <code>HTMLLoader</code> 对象无法显示 PDF 内容。
<code>HTMLPDFCapability.ERROR_INSTALLED_READER_TOO_OLD</code>	已检测到 Adobe Reader，但版本太旧。 <code>HTMLLoader</code> 对象无法显示 PDF 内容。
<code>HTMLPDFCapability.ERROR_PREFERRED_READER_TOO_OLD</code>	检测到的 Adobe Reader 版本足够高（8.1 或更高版本），但为处理 PDF 内容所安装的 Adobe Reader 版本早于 Reader 8.1。 <code>HTMLLoader</code> 对象无法显示 PDF 内容。

在 Windows 上，如果用户系统上当前正在运行 Adobe Acrobat 或 Adobe Reader 7.x 或更高版本，则即使安装了支持加载所加载 PDF 的更高版本，也会使用该版本。在这种情况下，如果 `pdfCampability` 属性的值为 `HTMLPDFCapability.STATUS_OK`，则当 AIR 应用程序尝试加载 PDF 内容时，Acrobat 或 Reader 的较早版本会显示警告（并且在 AIR 应用程序中不会引发任何异常）。如果您的最终用户有可能遇到这种情况，则可以考虑向他们提供说明，告知他们在运行应用程序的同时关闭 Acrobat。如果 PDF 内容在可以接受的时间范围内未加载，则您可能希望显示这些说明。

在 Linux 中，AIR 在由用户导出的 PATH（如果其包含 `acroread` 命令）中和 /opt/Adobe/Reader 目录中查找 Adobe Reader。

以下代码可检测用户是否能够在 AIR 应用程序中显示 PDF 内容，以及当不能显示时能否跟踪与 `HTMLPDFCapability` 错误对象对应的错误代码：

```
if(air.HTMLLoader.pdfCapability == air.HTMLPDFCapability.STATUS_OK)
{
    air.trace("PDF content can be displayed");
}
else
{
    air.trace("PDF cannot be displayed. Error code:", HTMLLoader.pdfCapability);
}
```

加载 PDF 内容

可以通过创建 `HTMLLoader` 实例、设置其尺寸以及加载 PDF 的路径，将 PDF 添加到 AIR 应用程序。

可以像在浏览器中那样，将 PDF 添加到 AIR 应用程序。例如，可以将 PDF 加载到窗口的顶级 HTML 内容、对象标签、`frame` 或 `iframe` 中。

以下示例从外部站点加载 PDF。将 `iframe` 的 `src` 属性值替换为可用外部 PDF 的路径。

```
<html>
  <body>
    <h1>PDF test</h1>
    <iframe id="pdfFrame"
      width="100%"
      height="100%"
      src="http://www.example.com/test.pdf"/>
  </body>
</html>
```

还可以从文件 URL 和特定于 AIR 的 URL 方案（例如 `app` 和 `app-storage`）加载内容。例如，以下代码可加载应用程序目录的 PDFs 子目录中的 `test.pdf` 文件：

`app:/js_api_reference.pdf`

有关 AIR URL 方案的详细信息，请参阅第 292 页的“[在 URL 中使用 AIR URL 方案](#)”。

编写 PDF 内容的脚本

可以像在浏览器的网页中那样，使用 JavaScript 控制 PDF 内容。

针对 Acrobat 的 JavaScript 扩展提供了以下功能（当然还包括其它功能）：

- 控制页面导航和缩放
- 处理文档中的表单
- 控制多媒体事件

有关 Adobe Acrobat 的 JavaScript 扩展的详细信息，请访问 Adobe Acrobat 开发人员中心：
<http://www.adobe.com/devnet/acrobat/javascript.html>。

HTML-PDF 通信基础知识

HTML 页中的 JavaScript 可以通过调用表示 PDF 内容的 DOM 对象的 `postMessage()` 方法，向 PDF 内容中的 JavaScript 发送消息。例如，请看以下嵌入的 PDF 内容：

```
<object id="PDFObj" data="test.pdf" type="application/pdf" width="100%" height="100%"/>
```

包含 HTML 内容中的以下 JavaScript 代码向 PDF 文件中的 JavaScript 发送消息：

```
pdfObject = document.getElementById("PDFObj");
pdfObject.postMessage(["testMsg", "hello"]);
```

PDF 文件可以包含 JavaScript 以便接收此消息。在某些上下文（包括文档级、文件夹级、页级、字段级和批级上下文）中，可以向 PDF 文件添加 JavaScript 代码。此处仅讨论文档级上下文，这种上下文在打开 PDF 文档时会对计算出的脚本进行定义。

PDF 文件可以向 `messageHandler` 对象添加 `hostContainer` 属性。`messageHandler` 属性是用于定义处理函数以便响应消息的一种对象。例如，以下代码定义了一个函数，用于处理 PDF 文件从主机容器（嵌入 PDF 文件的 HTML 内容）接收到的消息：

```

this.hostContainer.messageHandler = {onMessage: myOnMessage};

function myOnMessage(aMessage)
{
    if(aMessage[0] == "testMsg")
    {
        app.alert("Test message: " + aMessage[1]);
    }
    else
    {
        app.alert("Error");
    }
}

```

HTML 页中的 JavaScript 代码可以调用页面中包含的 PDF 对象的 postMessage() 方法。通过调用此方法，可向 PDF 文件中的文档级 JavaScript 发送消息 ("Hello from HTML")：

```

<html>
    <head>
        <title>PDF Test</title>
        <script>
            function init()
            {
                pdfObject = document.getElementById("PDFObj");
                try {
                    pdfObject.postMessage(["alert", "Hello from HTML"]);
                }
                catch (e)
                {
                    alert( "Error: \n name = " + e.name + "\n message = " + e.message );
                }
            }
        </script>
    </head>
    <body onload='init()'>
        <object
            id="PDFObj"
            data="test.pdf"
            type="application/pdf"
            width="100%" height="100%"/>
    </body>
</html>

```

有关更高级示例，以及有关使用 Acrobat 8 向 PDF 文件添加 JavaScript 的信息，请参阅[在 Adobe AIR 中跨脚本访问 PDF 内容](#)。

对 AIR 中的 PDF 内容的已知限制

Adobe AIR 中的 PDF 内容具有以下限制：

- 在透明窗口（NativeWindow 对象）中（transparent 属性设置为 true），不显示 PDF 内容。
- PDF 文件的显示顺序与 AIR 应用程序中的其它显示对象不同。尽管根据 HTML 显示顺序对 PDF 内容进行了正确剪辑，但在 AIR 应用程序的显示顺序中，它通常位于内容的上方。
- 全屏模式的窗口（将舞台（displayState 属性）的 window.nativeWindow.stage 属性设置为 air.StageDisplayState.FULL_SCREEN 或 air.StageDisplayState.FULL_SCREEN_INTERACTIVE 时）中不显示 PDF 内容。

- 如果更改包含 PDF 文档的 `HTMLLoader` 对象的某些视觉属性，则 PDF 文档将不可见。这些属性包括 `filters`、`alpha`、`rotation` 和 `scaling` 属性。更改这些属性将使 PDF 文件不可见，直到重置这些属性为止。如果更改包含 `HTMLLoader` 对象的显示对象容器的这些属性，也会出现这种情况。
- 只有在将包含 PDF 内容的 `NativeWindow` 对象的舞台对象 (`scaleMode` 属性) 的 `window.nativeWindow.stage` 属性设置为 `air.StageScaleMode.NO_SCALE` 时，PDF 内容才可见。将该属性设置为任何其他值时，PDF 内容均不可见。
- 单击指向 PDF 文件中内容的链接会更新 PDF 内容的滚动位置。单击指向 PDF 文件外部内容的链接会重定向包含 PDF 的 `HTMLLoader` 对象（即使链接的目标是新窗口）。
- 在 AIR 中，PDF 注释工作流不起作用。

第 28 章：处理声音

Adobe® AIR® 类包含的许多功能并不适用于在浏览器中运行的 HTML 内容，其中包括加载和播放声音内容的功能。

声音处理基础知识

在可以控制某种声音之前，需要将声音加载到 Adobe AIR 应用程序中。可以使用五种方法将音频数据加载到 AIR 中：

- 可以将外部声音文件（如 MP3 文件）加载到该应用程序中。
- 可以将声音信息嵌入 SWF 文件，加载（使用 `<script src="[swfFile].swf" type="application/x-shockwave-flash"/>`）并播放该文件。
- 可以使用连接到用户计算机上的麦克风来获取音频输入。
- 可以访问从服务器流式传输的声音数据。
- 可以动态地生成声音数据。

从外部声音文件加载声音数据时，您可以在仍加载其余声音数据的同时开始播放声音文件的开头部分。

虽然可以使用各种不同的声音文件格式对数字音频进行编码，但是 AIR 支持以 MP3 格式存储的声音文件。它不能直接加载或播放 WAV 或 AIFF 等其它格式的声音文件。

在 AIR 中处理声音时，可能会使用 `runtime.flash.media` 包中的某些类。可以使用 `Sound` 类来访问音频信息：加载声音文件或为对声音数据进行采样的事件分配函数，然后开始播放。开始播放声音后，AIR 可提供对 `SoundChannel` 对象的访问。已加载的音频文件只能是应用程序同时播放的几种声音之一。所播放的每种单独的声音都使用其自己的 `SoundChannel` 对象；混合在一起的所有 `SoundChannel` 对象的组合输出是实际通过扬声器播放的声音。可以使用此 `SoundChannel` 实例来控制声音的属性以及使其停止播放。最后，如果要控制组合音频，您可以通过 `SoundMixer` 类对混合输出进行控制。

也可以使用几个其它运行时类，在 AIR 中处理声音时执行更具体的任务。有关与声音有关的所有类的详细信息，请参阅第 252 页的“[了解声音体系结构](#)”。

Adobe AIR 开发人员中心提供了示例应用程序：[在基于 HTML 的应用程序中使用声音](http://www.adobe.com/go/learn_air_qs_sound_html_cn) (http://www.adobe.com/go/learn_air_qs_sound_html_cn)。

了解声音体系结构

应用程序可以从以下四种主要来源加载声音数据：

- 在运行时加载的外部声音文件
- SWF 文件中嵌入的声音资源
- 来自连接到用户系统上的麦克风的声音数据
- 从 Flash Media Server 等远程媒体服务器流式传输的声音数据
- 通过使用 `sampleData` 事件处理函数动态生成的声音数据

可以在播放之前完全加载声音数据，也可以进行流式传输，即在仍进行加载的同时播放这些数据。

Adobe AIR 支持以 MP3 格式存储的声音文件。它们不能直接加载或播放 WAV 或 AIFF 等其它格式的声音文件。（但 AIR 还可以使用 `NetStream` 类加载并播放 AAC 音频文件。）

AIR 声音体系结构包含以下类：

类	说明
Sound	Sound 类处理声音加载、管理基本声音属性以及启动声音播放。
SoundChannel	当应用程序播放 Sound 对象时，将创建一个新的 SoundChannel 对象来控制播放。SoundChannel 对象可控制声音的左和右播放声道的音量。播放的每种声音都具有其自己的 SoundChannel 对象。
SoundLoaderContext	SoundLoaderContext 类指定在加载声音时使用的缓冲秒数，以及运行时在加载文件时是否从服务器中查找跨域策略文件。SoundLoaderContext 对象用作 Sound.load() 方法的参数。
SoundMixer	SoundMixer 类可控制与应用程序中的所有声音有关的播放和安全属性。实际上，可通过一个通用 SoundMixer 对象将多个声道混合在一起。该 SoundMixer 对象中的属性值将影响当前播放的所有 SoundChannel 对象。
SoundTransform	SoundTransform 类包含控制音量和声相的值。可以将 SoundTransform 对象应用于单个 SoundChannel 对象、全局 SoundMixer 对象或 Microphone 对象等。
ID3Info	ID3Info 对象包含一些属性，它们表示通常存储在 MP3 声音文件中的 ID3 元数据信息。
Microphone	Microphone 类表示连接到用户计算机上的麦克风或其它声音输入设备。可以将来自麦克风的音频输入传送到本地扬声器或发送到远程服务器。Microphone 对象控制其自己的声音流的增益、采样率以及其它特性。

加载和播放的每种声音需要其自己的 Sound 类和 SoundChannel 类的实例。播放期间，SoundMixer 类将混合多个 SoundChannel 实例的输出。

Sound、SoundChannel 和 SoundMixer 类不能用于从麦克风或流媒体服务器（如 Flash Media Server）中获取的声音数据。

加载外部声音文件

Sound 类的每个实例都可加载并触发特定声音资源的播放。应用程序无法重复使用 Sound 对象来加载多种声音。若要加载新的声音资源，它应创建一个新的 Sound 对象。

创建 Sound 对象

如果要加载较小的声音文件（例如，要附加到按钮上的单击声音），应用程序可以创建一个 Sound 对象，并让其自动加载该声音文件，如下例所示：

```
var req = new air.URLRequest("click.mp3");
var s = new air.Sound(req);
```

Sound() 构造函数接受一个 URLRequest 对象作为其第一个参数。在提供 URLRequest 参数的值后，新的 Sound 对象将自动开始加载指定的声音资源。

除了最简单的情况外，应用程序都应关注声音的加载进度，并监视在加载期间出现的错误。例如，如果单击声音文件非常大，则在用户单击触发该声音的按钮时，应用程序可能没有完全加载该声音。尝试播放已卸载的声音可能会导致运行时错误。较为稳妥的作法是等待声音完全加载后，再让用户执行可以启动声音播放的动作。

关于声音事件

Sound 对象将在声音加载过程中调度多种不同的事件。应用程序可以侦听这些事件以跟踪加载进度，并确保在播放之前完全加载声音。下表列出了可以由 Sound 对象调度的事件：

事件	说明
open (air.Event.OPEN)	就在声音加载操作开始之前进行调度。
progress (air.ProgressEvent.PROGRESS)	在从文件或流接收到数据之后，在声音加载过程中定期进行调度。
id3 (air.Event.ID3)	当存在可用于 MP3 声音的 ID3 数据时进行调度。
complete (air.Event.COMPLETE)	在加载了所有声音资源的数据后进行调度。
ioError (air.IOErrorEvent.IO_ERROR)	在以下情况下进行调度：找不到声音文件，或者在收到所有声音数据之前加载过程中断。

以下代码说明了如何在完成加载后播放声音：

```
var s = new air.Sound();
s.addEventListener(air.Event.COMPLETE, onSoundLoaded);
var req = new air.URLRequest("bigSound.mp3");
s.load(req);

function onSoundLoaded(event)
{
    var localSound = event.target;
    localSound.play();
}
```

首先，该代码范例创建一个新的 Sound 对象，但没有为其指定 URLRequest 参数的初始值。然后，它通过 Sound 对象侦听 complete 事件，这将导致在加载完所有声音数据后执行 onSoundLoaded() 方法。接下来，它使用新的 URLRequest 值为声音文件调用 Sound.load() 方法。

在加载完声音后，将执行 onSoundLoaded() 方法。Event 对象的 target 属性是对 Sound 对象的引用。如果调用 Sound 对象的 play() 方法，则会启动声音播放。

监视声音加载过程

声音文件可能很大，需要花很长的时间进行加载，特别是在从 Internet 加载声音文件时。应用程序可以在完全加载声音之前播放声音。您可能需要向用户指示已加载了多少声音数据以及已播放了多少声音。

Sound 类调度以下两个事件可使显示声音加载进度变得相对简单：progress 和 complete。以下示例说明了如何使用这些事件来显示有关所加载的声音的进度信息：

```
var s = new Sound();
s.addEventListener(air.ProgressEvent.PROGRESS,
    onLoadProgress);
s.addEventListener(air.Event.COMPLETE,
    onLoadComplete);
s.addEventListener(air.IOErrorEvent.IO_ERROR,
    onIOError);

var req = new air.URLRequest("bigSound.mp3");
s.load(req);

function onLoadProgress(event)
{
    var loadedPct = Math.round(100 * (event.bytesLoaded / event.bytesTotal));
    air.trace("The sound is " + loadedPct + "% loaded.");
}

function onLoadComplete(event)
{
    var localSound = event.target;
    localSound.play();
}
function onIOError(event)
{
    air.trace("The sound could not be loaded: " + event.text);
}
```

此代码先创建一个 **Sound** 对象，然后在该对象中添加侦听器以侦听 **progress** 和 **complete** 事件。在已调用 **Sound.load()** 方法并从声音文件接收到第一批数据后，将会发生 **progress** 事件并触发 **onSoundLoadProgress()** 方法。

已加载的声音数据的小数部分等于 **ProgressEvent** 对象的 **bytesLoaded** 属性值除以 **bytesTotal** 属性后的值。**Sound** 对象上也提供了相同的 **bytesLoaded** 和 **bytesTotal** 属性。

此示例还说明了应用程序在加载声音文件时如何识别并响应出现的错误。例如，如果找不到具有给定文件名的声音文件，则 **Sound** 对象将调度一个 **ioError** 事件。在上面的代码中，当发生错误时，将执行 **onIOError()** 方法并显示一条简短的错误消息。

处理嵌入的声音

在 AIR 中，可以使用 JavaScript 访问 SWF 文件中嵌入的声音。可以使用以下任一方法将这些 SWF 文件加载到应用程序中：

- 通过在 HTML 页中使用 **<script>** 标签来加载 SWF 文件
- 通过使用 **runtime.flash.display.Loader** 类来加载 SWF 文件

将声音文件嵌入到应用程序的 SWF 文件中的具体方法因 SWF 开发环境而异。有关在 SWF 文件中嵌入媒体的信息，请参阅针对您的 SWF 开发环境的文档。

若要使用嵌入的声音，请在 ActionScript 中引用该声音的类名称。例如，通过创建自动生成的 **DrumSound** 类的一个实例来启动以下代码：

```
var drum = new DrumSound();
var channel = drum.play();
```

DrumSound 是 **flash.media.Sound** 类的子类，所以它继承了 **Sound** 类的方法和属性。它包括 **play()** 方法，如上一示例所示。

处理声音流文件

如果在仍加载声音文件或视频文件的数据的同时播放该文件，则认为是流式传输。通常，将对从远程服务器加载的声音文件进行流式传输，以使用户不必等待加载完所有声音数据再收听声音。

`SoundMixer.bufferTime` 属性表示应用程序在允许播放声音之前收集多长时间的声音数据（以毫秒为单位）。也就是说，如果将 `bufferTime` 属性设置为 5000，在开始播放声音之前，该应用程序将从声音文件中加载至少相当于 5000 毫秒的数据。`SoundMixer.bufferTime` 默认值为 1000。

通过在加载声音时显式地指定新的 `SoundMixer.bufferTime` 值，应用程序可以覆盖单个声音的全局 `bufferTime` 值。若要覆盖默认缓冲时间，请先创建一个 `SoundLoaderContext` 类实例，设置其 `bufferTime` 属性，然后将其作为参数传递给 `Sound.load()` 方法。以下示例说明了这一过程：

```
var s = new air.Sound();
var url = "http://www.example.com/sounds/bigSound.mp3";
var req = new air.URLRequest(url);
var context = new air.SoundLoaderContext(8000, true);
s.load(req, context);
s.play();
```

在播放继续进行时，AIR 尝试将声音缓冲区保持在相同大小或更大。如果声音数据的加载速度比播放速度快，播放将连续进行而不会中断。但是，如果数据加载速率由于网络限制而减慢，播放头可能会到达声音缓冲区的结尾。如果发生这种情况，播放会暂停，但播放会在已加载更多声音数据后自动恢复。

若要查明播放暂停是否是由于 AIR 正在等待加载数据，请使用 `Sound.isBuffering` 属性。

处理动态生成的音频

可以动态生成音频数据，而不是加载或流式传输现有声音。在为 `Sound` 对象的 `sampleData` 事件分配事件侦听器时，可以生成音频数据。（`sampleData` 事件在 `SampleDataEvent` 类中定义。）在这种情况下，`Sound` 对象不从文件中加载声音数据。相反，该对象将用作声音数据的套接字，声音数据通过使用您分配给此事件的函数流入该对象。

在您将 `sampleData` 事件侦听器添加到 `Sound` 对象后，该对象将定期请求数据以添加到声音缓冲区。此缓冲区包含 `Sound` 对象要播放的数据。在调用 `Sound` 对象的 `play()` 方法时，它会在请求新的声音数据时调度 `sampleData` 事件。（只有在 `Sound` 对象尚未从文件加载 mp3 数据时，此操作才会生效。）

`SampleDataEvent` 对象包含 `data` 属性。在事件侦听器中，将 `ByteArray` 对象写入此 `data` 对象。写入此对象的字节数组将添加到 `Sound` 对象播放的缓冲声音数据中。缓冲区中的字节数组是由从 -1 到 1 的浮点值组成的流。各浮点值均代表声音样本的一个声道（左声道或右声道）的幅度。声音按每秒 44,100 个样本进行采样。每个样本均包含左声道和右声道，在字节数组中以浮点数据的形式交错排列。

在处理函数中，使用 `ByteArray.writeFloat()` 方法写入 `sampleData` 事件的 `data` 属性。例如，以下代码将生成正弦波：

```
var mySound = new air.Sound();
mySound.addEventListener(air.SampleDataEvent.SAMPLE_DATA, sineWaveGenerator);
mySound.play();
function sineWaveGenerator(event)
{
    for (i = 0; i < 8192; i++)
    {
        var n = Math.sin((i + event.position) / Math.PI / 4);
        event.data.writeFloat(n);
        event.data.writeFloat(n);
    }
}
```

在调用 `Sound.play()` 时，该应用程序将调用事件处理函数，并请求声音样本数据。在播放声音时，应用程序将继续发送事件，直至您停止提供数据或调用 `SoundChannel.stop()`。

事件的滞后时间对于不同的平台会有所变化，并且在将来版本的 AIR 中也将改变。请不要依赖某个特定的滞后时间；而应计算出相应的滞后时间。若要计算滞后时间，请使用以下公式：

```
(SampleDataEvent.position / 44.1) - SoundChannelObject.position
```

向 `SampleDataEvent` 对象的 `data` 属性提供 2048 到 8192 个样本（对于每次事件侦听器调用）。为了获得最佳性能，请尽可能多地提供样本（最多可达 8192 个样本）。提供的样本越少，在播放过程中就越有可能出现单击和弹出事件。此行为对于不同的平台会有所不同，并且会在各种情况下发生。例如，在调整浏览器的大小时。在仅提供了 2048 个样本时，工作在某一个平台上的代码可能在运行于其它不同平台时将不能很好地工作。若要尽可能缩短滞后时间，请考虑允许用户选择数据量。

如果提供的样本少于 2048 个（每次 `sampleData` 事件侦听器调用），则应用程序将在播放完剩余的样本后停止。随后，它将调度 `SoundComplete` 事件。

对源自 mp3 数据的声音数据进行修改

使用 `Sound.extract()` 方法提取 `Sound` 对象中的数据。可以使用（和修改）该数据，将其写入另一个 `Sound` 对象的动态流以进行播放。例如，以下代码使用加载的 MP3 文件的字节，并通过过滤函数 `upOctave()` 将进行传递：

```
var mySound = new air.Sound();
var sourceSnd = new air.Sound();
var urlReq = new air.URLRequest("test.mp3");
sourceSnd.load(urlReq);
sourceSnd.addEventListener(air.Event.COMPLETE, loaded);
function loaded(event)
{
    mySound.addEventListener(SampleDataEvent.SAMPLE_DATA, processSound);
    mySound.play();
}
function processSound(event)
{
    var bytes = new air.ByteArray();
    sourceSnd.extract(bytes, 8192);
    event.data.writeBytes(upOctave(bytes));
}
function upOctave(bytes)
{
    var returnBytes = new air.ByteArray();
    bytes.position = 0;
    while(bytes.bytesAvailable > 0)
    {
        returnBytes.writeFloat(bytes.readFloat());
        returnBytes.writeFloat(bytes.readFloat());
        if (bytes.bytesAvailable > 0)
        {
            bytes.position += 8;
        }
    }
    return returnBytes;
}
```

有关生成声音的限制

将 `sampleData` 事件侦听器与 `Sound` 对象一起使用时，支持的其它 `Sound` 方法仅包括 `Sound.extract()` 和 `Sound.play()`。调用任何其它方法或属性将导致异常。仍启用 `SoundChannel` 对象的所有方法和属性。

播放声音

播放加载的声音非常简便，您只需为 Sound 对象调用 Sound.play() 方法，如下所示：

```
var req = new air.URLRequest("smallSound.mp3");
var snd = new air.Sound(req);
snd.play();
```

声音播放操作

播放声音时，您可以执行以下操作：

- 从特定起始位置播放声音
- 暂停声音并稍后从相同位置恢复播放
- 准确了解何时播放完声音
- 跟踪声音的播放进度
- 在播放声音的同时更改音量或声相

若要在播放期间执行这些操作，请使用 SoundChannel、SoundMixer 和 SoundTransform 类。

SoundChannel 类控制一种声音的播放。可以将 SoundChannel.position 属性视为播放头，以指示所播放的声音数据中的当前位置。

当应用程序调用 Sound.play() 方法时，将创建一个新的 SoundChannel 类实例来控制播放。

通过将特定起始位置（以毫秒为单位）作为 Sound.play() 方法的 startTime 参数进行传递，应用程序可以从该位置播放声音。它也可以通过在 Sound.play() 方法的 loops 参数中传递一个数值，指定将声音连续重播一定次数。

使用 startTime 参数和 loops 参数调用 Sound.play() 方法时，每次将从相同的起始点重复播放声音。以下代码说明了这一过程：

```
var req = new air.URLRequest("repeatingSound.mp3");
var snd = new air.Sound();
snd.play(1000, 3);
```

在此示例中，从声音开始后的 1 秒起连续播放声音三次。

暂停和恢复播放声音

如果应用程序播放很长的声音（如歌曲或播客），您可能需要让用户暂停和恢复播放这些声音。实际上，无法在播放期间暂停声音；而只能将其停止。但是，可以从任何位置开始播放声音。您可以记录声音停止时的位置，并随后从该位置开始重放声音。

例如，假定代码加载并播放一个声音文件，如下所示：

```
var req = new air.URLRequest("bigSound.mp3");
var snd = new air.Sound(req);
var channel = snd.play();
```

在播放声音时，channel 对象的 position 属性指示声音文件中当前播放的位置。应用程序可以在停止播放声音之前存储位置值，如下所示：

```
var pausePosition = channel.position;
channel.stop();
```

若要恢复播放声音，请传递以前存储的位置值，以便从声音以前停止的相同位置重新启动声音。

```
channel = snd.play(pausePosition);
```

监视播放

应用程序可能需要了解何时停止播放某种声音，然后，它可以开始播放另一种声音，或者清除在以前播放期间使用的某些资源。SoundChannel 类在其声音完成播放时将调度 soundComplete 事件。应用程序可以侦听此事件并执行相应的动作，如下例所示：

```
var snd = new air.Sound("smallSound.mp3");
var channel = snd.play();
s.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);

public function onPlaybackComplete(event)
{
    air.trace("The sound has finished playing.");
}
```

SoundChannel 类在播放期间不调度 progress 事件。若要报告播放进度，应用程序可以设置自己的计时机制并跟踪声音播放头的位置。

若要计算已播放的声音百分比，您可以将 SoundChannel.position 属性值除以所播放的声音数据长度：

```
var playbackPercent = 100 * (channel.position / snd.length);
```

但是，只有在开始播放之前加载全部声音数据，此代码才能报告精确的播放百分比。Sound.length 属性显示当前加载的声音数据的大小，而不是整个声音文件的最终大小。若要跟踪仍在加载的声音流的播放进度，应用程序应估计完整声音文件的最终大小，并在计算中使用该值。您可以使用 Sound 对象的 bytesLoaded 和 bytesTotal 属性来估计声音数据的最终长度，如下所示：

```
var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
var playbackPercent = 100 * (channel.position / estimatedLength);
```

以下代码加载一个较大的声音文件，并使用 setInterval() 函数作为计时机制来显示播放进度。此代码会定期报告播放百分比，该百分比是由当前位置值除以声音数据的总长度得出的：

```
var snd = new air.Sound();
var url = "http://www.example.com/sounds/test.mp3";
var req = new air.URLRequest(url);
snd.load(req);

var channel = snd.play();
var timer = setInterval(monitorProgress, 100);
snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);

function monitorProgress(event)
{
    var estimatedLength = Math.ceil(snd.length / (snd.bytesLoaded / snd.bytesTotal));
    var playbackPercent = Math.round(100 * (channel.position / estimatedLength));
    air.trace("Sound playback is " + playbackPercent + "% complete.");
}

function onPlaybackComplete(event)
{
    air.trace("The sound has finished playing.");
    clearInterval(timer);
}
```

在开始加载声音数据后，此代码会调用 snd.play() 方法，并将生成的 SoundChannel 对象存储在 channel 变量中。然后，此代码会添加 monitorProgress() 方法，该方法将由 setInterval() 函数重复调用。此代码将事件侦听器用于 SoundChannel 对象以侦听播放完成时发生的 soundComplete 事件。

monitorProgress() 方法会基于已加载的数据量估计声音文件的总长度。然后，此代码会计算并显示当前播放百分比。

在播放完声音后，将执行 onPlaybackComplete() 函数。此函数会删除 setInterval() 函数的回调方法，以使播放完成后应用程序不显示进度更新。

停止声音流

在进行流式传输的声音（即，在播放的同时仍在加载声音）的播放过程中，有一个奇怪的现象。当对播放声音流的 SoundChannel 实例调用 `stop()` 方法时，声音播放将会停止，随后从声音开头重新播放。发生这种情况是因为声音加载过程仍在进行当中。若要同时停止声音流加载和播放，请调用 `Sound.close()` 方法。

控制音量和声相

单个 SoundChannel 对象可同时控制声音的左立体声声道和右立体声声道。如果 MP3 声音是单声道声音， SoundChannel 对象的左立体声声道和右立体声声道将包含完全相同的波形。

可通过使用 SoundChannel 对象的 `leftPeak` 和 `rightPeak` 属性来查明所播放的声音的每个立体声声道的波幅。这些属性显示声音波形本身的峰值波幅。它们并不表示实际播放音量。实际播放音量是声音波形的波幅以及 SoundChannel 对象和 SoundMixer 类中设置的音量值的函数。

在播放期间，可以使用 SoundChannel 对象的 `pan` 属性为左声道和右声道分别指定不同的音量级别。`pan` 属性可以具有范围从 -1 到 1 的值。值 -1 表示左声道以最大音量播放而右声道处于静音状态。值 1 表示右声道以最大音量播放而左声道处于静音状态。介于 -1 和 1 之间的值可为左右声道值设置一定比例的值。值 0 表示两个声道以均衡的中音量级别播放。

以下代码示例使用 `volume` 值 0.6 和 `pan` 值 -1 创建一个 SoundTransform 对象（左声道为最高音量，右声道没有音量）。它会将 SoundTransform 对象作为参数传递给 `play()` 方法。`play()` 方法将该 SoundTransform 对象应用于为控制播放而创建的新 SoundChannel 对象。

```
var req = new air.URLRequest("bigSound.mp3");
var snd = new air.Sound(req);
var trans = new air.SoundTransform(0.6, -1);
var channel = snd.play(0, 1, trans);
```

可以在播放声音的同时更改音量和声相。设置对象的 `pan` 或 `volume` 属性，然后将该对象作为 SoundChannel 对象的 `soundTransform` 属性进行应用。

也可以通过使用 SoundMixer 类的 `soundTransform` 属性，同时为所有声音设置全局音量和声相值。以下示例说明了这一过程：

```
SoundMixer.soundTransform = new air.SoundTransform(1, -1);
```

也可以使用 SoundTransform 对象来设置 Microphone 对象的音量和声相值（请参阅第 265 页的“[捕获声音输入](#)”）。

以下示例在播放声音的同时将声音从左声道移到右声道，然后再移回来，并交替进行这一过程：

```
var snd = new air.Sound();
var req = new air.URLRequest("bigSound.mp3");
snd.load(req);

var panCounter = 0;

var trans = new air.SoundTransform(1, 0);
var channel = snd.play(0, 1, trans);
channel.addEventListener(air.Event.SOUND_COMPLETE,
    onPlaybackComplete);

var timer = setInterval(panner, 100);

function panner()
{
    trans.pan = Math.sin(panCounter);
    channel.soundTransform = trans; // or SoundMixer.soundTransform = trans;
    panCounter += 0.05;
}

function onPlaybackComplete(event)
{
    clearInterval(timer);
}
```

此代码先加载一个声音文件，然后将音量设置为 1（最大音量）并将声相设置为 0（声音在左和右声道之间均衡地平均分布）以创建一个 `SoundTransform` 对象。接下来，此代码调用 `snd.play()` 方法，以将 `SoundTransform` 对象作为参数进行传递。

在播放声音时，将反复执行 `panner()` 方法。`panner()` 方法将使用 `Math.sin()` 函数生成一个介于 -1 和 1 之间的值。此范围对应于可接受的 `SoundTransform.pan` 属性值。此代码将 `SoundTransform` 对象的 `pan` 属性设置为新值，然后设置声道的 `soundTransform` 属性以使用更改后的 `SoundTransform` 对象。

若要运行此示例，请用本地 MP3 文件的名称替换文件名 `bigSound.mp3`。然后，运行该示例。当右声道音量变小时，您应会听到左声道音量变大，反之亦然。

在此示例中，可通过设置 `SoundMixer` 类的 `soundTransform` 属性来获得同样的效果。但是，这会影响当前播放的所有声音的声相，而不是只影响此 `SoundChannel` 对象播放的一种声音。

处理声音元数据

使用 MP3 格式的声音文件可以采用 ID3 标签格式来包含有关声音的其它数据。

并非每个 MP3 文件都包含 ID3 元数据。当 `Sound` 对象加载 MP3 声音文件时，如果该声音文件包含 ID3 元数据，它将调度 `Event.ID3` 事件。若要防止出现运行时错误，应用程序应等待接收 `Event.ID3` 事件后，再访问加载的声音的 `Sound.id3` 属性。

以下代码说明了如何识别何时加载了声音文件的 ID3 元数据：

```
var s = new air.Sound();
s.addEventListener(air.Event.ID3, onID3InfoReceived);
var urlReq = new air.URLRequest("mySound.mp3");
s.load(urlReq);

function onID3InfoReceived(event)
{
    var id3 = event.target.id3;

    air.trace("Received ID3 Info:");
    for (propName in id3)
    {
        air.trace(propName + " = " + id3[propName]);
    }
}
```

此代码先创建一个 **Sound** 对象并通知它侦听 **id3** 事件。加载了声音文件的 ID3 元数据后，将调用 **onID3InfoReceived()** 方法。传递给 **onID3InfoReceived()** 方法的 **Event** 对象的目标是原始 **Sound** 对象。该方法随后获取 **Sound** 对象的 **id3** 属性并循环访问其命名属性以跟踪它们的值。

访问原始声音数据

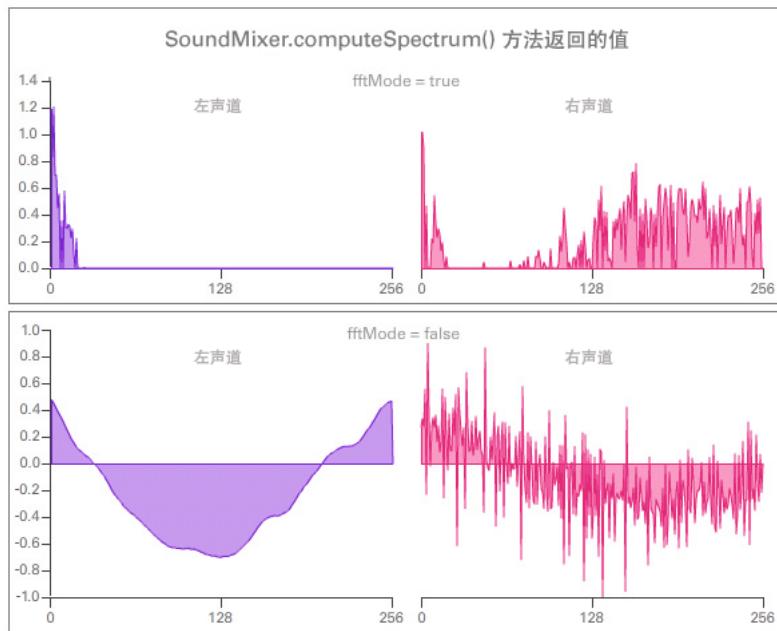
通过使用 **SoundMixer.computeSpectrum()** 方法，应用程序可以读取当前所播放的波形的原始声音数据。如果当前播放多个 **SoundChannel** 对象，**SoundMixer.computeSpectrum()** 方法将显示各个 **SoundChannel** 对象混合在一起的组合声音数据。

声音数据的返回方式

声音数据是作为 **ByteArray** 对象（包含 512 个 4 字节数组）返回的，其中的每个字节表示一个介于 -1 和 1 之间的浮点值。这些值表示所播放的声音波形中的点的波幅。这些值是分为两个组（每组包含 256 个值）提供的，第一个组用于左立体声声道，第二个组用于右立体声声道。

如果将 **FFTMode** 参数设置为 **true**，**SoundMixer.computeSpectrum()** 方法将返回频谱数据，而非波形数据。频谱显示按音频频率（从最低频率到最高频率）排列的波幅。可以使用快速傅立叶变换（FFT）将波形数据转换为频谱数据。生成的频谱值范围介于 0 和约 1.414（2 的平方根）之间。

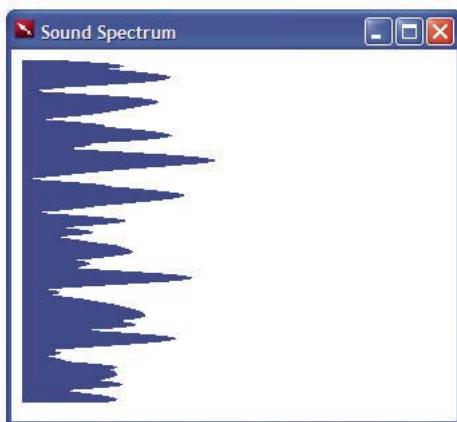
下图比较了将 `FFTMode` 参数设置为 `true` 和 `false` 时从 `computeSpectrum()` 方法返回的数据。此图所用的声音在左声道中包含很大的低音；而在右声道中包含击鼓声。



`computeSpectrum()` 方法也可以返回在较低比特率重新采样的数据。通常，这会产生更平滑的波形数据或频率数据，但会以牺牲细节为代价。`stretchFactor` 参数控制 `computeSpectrum()` 方法数据的采样率。如果将 `stretchFactor` 参数设置为 0（默认值），则以采样率 44.1 kHz 采集声音数据样本。`stretchFactor` 参数值每连续增加 1，采样率就降低一半。因此，值 1 指定采样率 22.05 kHz，值 2 指定采样率 11.025 kHz，依此类推。当使用较高的 `stretchFactor` 值时，`computeSpectrum()` 方法仍会为每个立体声声道返回 256 个浮点值。

构建简单的声音可视化程序

以下示例使用 `SoundMixer.computeSpectrum()` 方法来显示定期更新的声音波形图：



```
<html>
    <title>Sound Spectrum</title>
    <script src="AIRAliases.js" />
    <script>
        const PLOT_WIDTH = 600;
        const CHANNEL_LENGTH = 256;

        var snd = new air.Sound();
        var req = new air.URLRequest("test.mp3");
        var bytes = new air.ByteArray();
        var divStyles = new Array();

        /**
         * Initializes the application. It draws 256 DIV elements to the document body,
         * and sets up a divStyles array that contains references to the style objects of
         * each DIV element. It then calls the playSound() function.
         */
        function init()
        {
            var div;
            for (i = 0; i < CHANNEL_LENGTH; i++)
            {
                div = document.createElement("div");
                div.style.height = "1px";
                div.style.width = "0px";
                div.style.backgroundColor = "blue";
                document.body.appendChild(div);
                divStyles[i] = div.style;
            }
            playSound();
        }
        /**
         * Plays a sound, and calls setInterval() to call the setMeter() function
         * periodically, to display the sound spectrum data.
         */
        function playSound()
        {
            if (snd.url != null)
            {
                snd.close();
            }
            snd.load(req);
            var channel = snd.play();
            timer = setInterval(setMeter, 100);
            snd.addEventListener(air.Event.SOUND_COMPLETE, onPlaybackComplete);
        }

        /**
         * Computes the width of each of the 256 colored DIV tags in the document,
         * based on data returned by the call to SoundMixer.computeSpectrum(). The
         * first 256 floating point numbers in the byte array represent the data from
         * the left channel, and then next 256 floating point numbers represent the
         * data from the right channel.
         */
        function setMeter()
        {
            air.SoundMixer.computeSpectrum(bytes, false, 0);
            var n;
            for (var i = 0; i < CHANNEL_LENGTH; i++)
```

```

    {
        bytes.position = i * 4;
        n = Math.abs(bytes.readFloat());
        bytes.position = 256*4 + i * 4;
        n += Math.abs(bytes.readFloat());
        divStyles[i].width = n * PLOT_WIDTH;
    }
}
/**
 * When the sound is done playing, remove the intermediate process
 * started by setInterval().
 */
function onPlaybackComplete(event)
{
    clearInterval(interval);
}
</script>
<body onload="init()">
</body>
</html>

```

此示例首先加载并播放声音文件，然后使用 `setInterval()` 函数监视 `SoundMixer.computeSpectrum()` 方法，该方法将声音波形数据存储在 `bytes ByteArray` 对象中。

声音波形是通过设置表示条形图的 `div` 元素的宽度进行绘制的。

捕获声音输入

应用程序可通过 `Microphone` 类连接到用户系统上的麦克风或其它声音输入设备。应用程序可将输入音频广播到该系统的扬声器，或者将音频数据发送到远程服务器，如 `Flash Media Server`。您无法访问源自麦克风的原始音频数据；只能将音频发送到系统的扬声器或将压缩音频数据发送到远程服务器。对于发送到远程服务器的数据，可以使用 `Speex` 或 `Nellymoser` 编解码器。（在 AIR 1.5 中可以使用 `Speex` 编解码器。）

访问麦克风

`Microphone` 类没有构造函数方法。相反，应使用静态 `Microphone.getMicrophone()` 方法来获取新的 `Microphone` 实例，如下例所示：

```
var mic = air.Microphone.getMicrophone();
```

不使用参数调用 `Microphone.getMicrophone()` 方法时，将返回在用户系统上发现的第一个声音输入设备。

系统可能连接了多个声音输入设备。应用程序可以使用 `Microphone.names` 属性来获取所有可用声音输入设备名称的数组。然后，它可以使用 `index` 参数（与数组中的设备名称的索引值相匹配）来调用 `Microphone.getMicrophone()` 方法。

系统可能没有连接麦克风或其它声音输入设备。可以使用 `Microphone.names` 属性或 `Microphone.getMicrophone()` 方法来检查用户是否安装了声音输入设备。如果用户未安装声音输入设备，则 `names` 数组的长度为零，并且 `getMicrophone()` 方法返回值 `null`。

将麦克风音频传送到本地扬声器

可以使用参数值 `true` 调用 `Microphone.setLoopback()` 方法，以将来自麦克风的音频输入传送到本地系统扬声器。

如果将来自本地麦克风的声音传送到本地扬声器，则会存在产生音频回馈循环的风险。这可能会导致非常大的振鸣声，并且可能会损坏声音硬件。使用参数值 `true` 调用 `Microphone.setUseEchoSuppression()` 方法可降低发生音频回馈的风险，但不会完全消除该风险。Adobe 建议务必在调用 `Microphone.setLoopback(true)` 之前调用 `Microphone.setUseEchoSuppression(true)`，除非您确信用户使用耳机或除扬声器以外的某种设备来播放声音。

以下代码说明了如何将来自本地麦克风的音频传送到本地系统扬声器：

```
var mic = air.Microphone.getMicrophone();
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
```

更改麦克风音频

应用程序可以使用两种方法更改来自麦克风的音频数据。第一，它可以更改输入声音的增益，这实际上是将输入值乘以指定数值。这样可产生更大或更小的声音。`Microphone.gain` 属性接受介于 0 和 100 之间的数值。值 50 相当于乘数 1，它指定正常音量。值 0 相当于乘数 0，它可有效地将输入音频静音。大于 50 的值指定的音量高于正常音量。

应用程序也可以更改输入音频的采样率。较高的采样率可提高声音品质，但它们也会创建更密集的数据流（使用更多的资源进行传输和存储）。`Microphone.rate` 属性表示以千赫 (kHz) 为单位测量的音频采样率。默认采样率是 8 kHz。如果麦克风支持较高的采样率，您可以将 `Microphone.rate` 属性设置为高于 8 kHz 的值。例如，如果将 `Microphone.rate` 属性设置为 11，则会将采样率设置为 11 kHz；如果将其设置为 22，则会将采样率设置为 22 kHz，依此类推。采样率取决于所选择的编解码器。如果使用的是 Nellymoser 编解码器，则可以指定 5、8、11、16、22 和 44 kHz 作为采样率。如果使用的是 Speex 编解码器（在 AIR 1.5 中可用），则只能使用 16 kHz。

检测麦克风活动

为节省带宽和处理资源，运行时将尝试检测何时麦克风不传输声音。在麦克风的活动级别处于静音级别阈值以下一段时间后，运行时将停止传输音频输入并调度 `activity` 事件。如果使用 Speex 编解码器（在 AIR 1.5 中可用），请将静音级别设置为 0，以确保应用程序能够持续传输音频数据。Speex 语音活动检测将自动减少带宽。

`Microphone` 类的以下三个属性用于监视和控制活动检测：

- `activityLevel` 只读属性指示麦克风检测的音量，范围从 0 到 100。
- `silenceLevel` 属性指定激活麦克风并调度 `activity` 事件所需的音量。`silenceLevel` 属性也使用从 0 到 100 的范围，默认值为 10。
- `silenceTimeout` 属性描述活动级别必须处于静音级别以下多长时间（以毫秒为单位）后，才会调度 `activity` 事件。`silenceTimeout` 默认值为 2000。

`Microphone.silenceLevel` 属性和 `Microphone.silenceTimeout` 属性都是只读的，但可以使用 `Microphone.setSilenceLevel()` 方法来更改它们的值。

在某些情况下，在检测到新活动时激活麦克风的过程可能会导致短暂的延迟。通过将麦克风始终保持活动状态，可以消除此类激活延迟。应用程序可以调用 `Microphone.setSilenceLevel()` 方法并将 `silenceLevel` 参数设置为零。这样可将麦克风保持活动状态并持续收集音频数据，即使未检测到任何声音也是如此。反之，如果将 `silenceLevel` 参数设置为 100，则可以完全禁止激活麦克风。

以下示例显示了有关麦克风的信息，并报告 `Microphone` 对象调度的 `activity` 事件和 `status` 事件：

```
var deviceArray = air.Microphone.names;
air.trace("Available sound input devices:");
for (i = 0; i < deviceArray.length; i++)
{
    air.trace(" " + deviceArray[i]);
}

var mic = air.Microphone.getMicrophone();
mic.gain = 60;
mic.rate = 11;
mic.setUseEchoSuppression(true);
mic.setLoopBack(true);
mic.setSilenceLevel(5, 1000);

mic.addEventListener(air.ActivityEvent.ACTIVITY, this.onMicActivity);

var micDetails = "Sound input device name: " + mic.name + '\n';
micDetails += "Gain: " + mic.gain + '\n';
micDetails += "Rate: " + mic.rate + " kHz" + '\n';
micDetails += "Muted: " + mic.muted + '\n';
micDetails += "Silence level: " + mic.silenceLevel + '\n';
micDetails += "Silence timeout: " + mic.silenceTimeout + '\n';
micDetails += "Echo suppression: " + mic.useEchoSuppression + '\n';
air.trace(micDetails);

function onMicActivity(event)
{
    air.trace("activating=" + event.activating + ", activityLevel=" +
        mic.activityLevel);
}
```

在运行上面的示例时，对着系统麦克风说话或发出噪音，并观察所生成的、显示在控制台中的 trace 语句。

向媒体服务器发送音频以及从中接收音频

使用 Flash Media Server 等流媒体服务器时，可以使用额外的音频功能。

特别是应用程序可以将 Microphone 对象附加到 runtime.flash.net.NetStream 对象上，并将数据直接从用户麦克风传输到服务器。也可以将音频数据从服务器流式传输到 AIR 应用程序。

AIR 1.5 引入了对 Speex 编解码器的支持。若要设置向媒体服务器发送的压缩音频所使用的编解码器，请设置 Microphone 对象的 codec 属性。此属性可以包含两个值，可使用 SoundCodec 类对这两个值进行枚举。将 codec 属性设置为 SoundCodec.SPEEX 将选择使用 Speex 编解码器来压缩音频。将该属性设置为 SoundCodec.NELLYMOSER（默认值）将选择使用 Nellymoser 编解码器来压缩音频。

有关详细信息，请参阅 <http://www.adobe.com/support/documentation/cn> 上提供的在线 Flash Media Server 文档。

第 29 章：使用数字权限管理

媒体发行商使用 Adobe® Flash® Media Rights Management Server (FMRMS) 可以分发内容（特别是 FLV 和 MP4 文件），还可以通过客户的直接补偿（用户付费）或间接补偿（广告付费）收回生产成本。发行商以加密的 FLV 形式分发媒体，这些媒体可被下载并在 Adobe® Media Player™ 或任何使用数字权限管理 (DRM) API 的 Adobe® AIR® 应用程序中播放。

内容提供者使用 FMRMS 可以通过基于身份的许可借助用户凭据来保护自己的内容。例如，客户希望观看电视节目，但却不希望观看附加的广告。若要避免观看广告，客户可注册并向内容发行商支付额外费用。然后，用户可使用其身份验证凭据访问并播放没有商业广告的节目。另一位客户可能希望在无法访问 Internet 的旅行期间脱机观看内容。在注册并为内容发行商的额外付费服务支付费用后，该用户即可使用其身份验证凭据从发行商的网站访问并下载相应节目。然后，此用户可以在允许的时段内脱机观看相应内容。这些内容也受用户凭据保护，无法与其他用户共享。

如果用户尝试播放 DRM 加密的文件，那么应用程序将与 FMRMS 联系，FMRMS 再通过内容发行商的服务提供商接口 (SPI) 与内容发行商的系统联系，以便对用户进行身份验证并检索许可证。许可证是一个凭证，用于确定是否允许用户访问相应内容；如果允许，则访问时间有多长。该凭证还用于确定用户是否可以脱机访问内容；如果可以，则访问时间有多长。同样，需要用户凭据来确定用户是否可以访问加密的内容。

基于身份的许可也支持匿名访问。例如，提供者可使用匿名访问分发支持广告的内容或允许在指定天数内免费访问当前内容。归档资料可作为必须付费且需要用户凭据的额外付费内容。内容提供者还可指定和限制播放其内容所需的播放器类型和版本。

本章介绍如何使 AIR 应用程序能够播放受数字权限管理加密保护的内容。您不一定了解如何使用 DRM 对内容进行加密，但本章的叙述假定您可以使用 DRM 加密内容并正在与 FMRMS 联系，以便对用户进行身份验证并检索凭证。

有关包括创建策略在内的 FMRMS 概述，请参阅 FMRMS 随附文档。

有关 Adobe Media Player 的信息，请参阅 Adobe Media Player 中提供的 Adobe Media Player 帮助。

有关数字权限管理的其它在线信息

可以从以下源中查找有关数字权限管理的详细信息：

语言参考

Adobe 开发人员中心文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员中心](#) (搜索“数字权限管理”或“drm”)

了解加密的 FLV 工作流程

当 AIR 应用程序尝试播放 DRM 加密的文件时可能会调度四类事件：StatusEvent、DRMAuthenticateEvent、DRMErrorEvent 和 DRMStatusEvent。为了支持这些文件，应用程序应添加事件侦听器以处理这些 DRM 事件。

以下工作流程说明 AIR 应用程序如何检索并播放受 DRM 加密保护的内容：

- 1 应用程序尝试使用 NetStream 对象播放 FLV 或 MP4 文件。如果内容已加密，则通过代码 events.StatusEvent 调度 DRM.encryptedFLV 事件，该代码指明 FLV 已加密。

注：如果应用程序不播放 DRM 加密的文件，则它会在遇到加密内容时侦听调度的状态事件，然后通知用户相应文件不受支持并关闭连接。

- 2 如果文件是匿名加密的，即允许所有用户观看内容而不需要输入身份验证凭据，则 AIR 应用程序将进行此工作流程的最后一步。然而，如果文件需要基于身份的许可证，即需要用户凭据，则 **NetStream** 对象会调度一个 **DRMAuthenticateEvent** 对象。用户必须先提供其身份验证凭据，然后才能开始播放。
- 3 AIR 应用程序必须提供某种机制来收集必需的身份验证凭据。内容服务器提供的 **DRMAuthenticationEvent** 类的 **usernamePrompt**、**passwordPrompt** 和 **urlPrompt** 属性可用于为最终用户提供有关所需数据的信息。您可以使用这些属性构造用于检索所需用户凭据的用户界面。例如，**usernamePrompt** 值字符串可能会声明用户名必须为电子邮件地址形式。
注：AIR 不提供用于收集身份验证凭据的默认用户界面。应用程序开发人员必须编写此用户界面并处理 **DRMAuthenticateEvent** 事件。如果应用程序没有为 **DRMAuthenticateEvent** 对象提供事件侦听器，则 DRM 加密的对象将保持“等待凭据”状态，因而不提供相应内容。
- 4 应用程序获得用户凭据后，就会用 **setDRMAuthenticationCredentials()** 方法将凭据传递给 **NetStream** 对象。此操作向 **NetStream** 对象发出信号，通知该对象应尽早尝试对用户进行身份验证。然后，AIR 会将凭据传递给 FMRMS 以进行身份验证。如果用户通过身份验证，则应用程序将进行下一步。

如果身份验证失败，则 **NetStream** 对象会调度一个新的 **DRMAuthenticateEvent** 对象，并且应用程序将返回到步骤 3。此流程无限次重复。应用程序应提供一种机制以处理和限制重复身份验证尝试次数。例如，应用程序可允许用户取消尝试，而这可以关闭 **NetStream** 连接。

- 5 验证用户的身份后，或使用匿名加密时，DRM 子系统将检索凭证。凭证用于检查用户是否获得观看内容的授权。凭证中的信息既适用于经过身份验证的用户也适用于匿名用户。例如，经过身份验证的用户和匿名用户也许能够在内容过期之前的指定时段内访问内容，也许由于内容提供者不支持执行观看的应用程序的版本而无法访问内容。

如果没有发生错误，并且用户已成功获得查看内容的授权，则 **NetStream** 对象将调度一个 **DRMStatusEvent** 对象，并且 AIR 应用程序将开始播放。**DRMStatusEvent** 对象保留相关凭证信息，这些信息用于标识用户的策略和权限。例如，它保留有关是否可脱机使用内容或凭证何时过期而无法再观看内容的信息。应用程序可以使用此数据来通知用户其策略的状态。例如，应用程序可在状态栏上显示用户可以观看相应内容的剩余天数。

如果允许用户脱机访问，则将缓存凭证并将加密内容下载到用户计算机中，同时使加密内容在脱机租赁期限定义的时段内可供访问。事件中的 **detail** 属性包含“**DRM.voucherObtained**”。应用程序决定内容的本地存储位置，以便内容可脱机使用。在 AIR 1.5 中，还可以使用 **DRMManager** 类预加载凭证。

所有与 DRM 相关的错误都会导致应用程序调度 **DRMErrorEvent** 事件对象。AIR 通过重新调度 **DRMAuthenticationEvent** 对象来处理使用 **NetStream setDRMAuthenticationCredentials()** 方法时遇到的身份验证失败。所有其它错误事件应由应用程序显式处理。这包括用户输入有效凭据但保护加密内容的凭证却限制对内容的访问的情况。例如，由于尚未对权限进行付费，因此通过身份验证的用户仍无法访问内容。当作为同一媒体发行商的注册成员的两名用户试图共享仅其中一名成员付费的内容时，也会出现这种情况。应用程序应就错误（例如对内容的限制）通知用户，同时提供替代性内容，例如，如何注册和对观看内容的权限付费的说明。

预加载用于脱机播放的凭证

可以预加载播放受 DRM 保护的内容所需的凭证。通过预加载的凭证，无论是否有活动的 Internet 连接，用户都可以查看内容。（当然，预加载过程本身需要 Internet 连接。）使用 **NetStream** 类 **preloadEmbeddedMetadata()** 方法和 AIR 1.5 **DRMManager** 类预加载凭证。

以下步骤介绍为受 DRM 保护的媒体文件预加载凭证所采用的工作流：

- 1 下载并存储媒体文件。（只能从本地存储的文件中预加载 DRM 元数据。）
- 2 创建 **NetConnection** 和 **NetStream** 对象，为 **NetStream** 客户端对象的 **onDRMContentData()** 和 **onPlayStatus()** 回调函数提供实现。
- 3 创建 **NetStreamPlayOptions** 对象，并将 **stream** 属性设置为本地媒体文件的 URL。
- 4 调用 **NetStream preloadEmbeddedMetadata()**，并传入 **NetStreamPlayOptions** 对象，从而标识要分析的媒体文件。

5 如果媒体文件包含 DRM 元数据，则调用 `onDRMContentData()` 回调函数。将元数据作为 `DRMContentData` 对象传递到此函数。

6 使用 `DRMContentData` 对象通过 `DRMManager loadVoucher()` 方法获取凭证。

如果 `DRMContentData` 对象的 `authenticationMethod` 属性的值为 `userNameAndPassword`，则必须在加载凭证之前验证用户在媒体权限服务器上的权限。可以将 `DRMContentData` 对象的 `serverURL` 和 `domain` 属性传递给 `DRMManager authenticate()` 方法，附带用户的凭据。

7 文件分析完毕后将调用 `onPlayStatus()` 回调函数。如果尚未调用 `onDRMContentData()` 函数，则文件不包含获取凭证所需的元数据（并且可能不受 DRM 保护）。

以下代码示例介绍如何为本地媒体文件预加载 DRM 凭证：

```
package
{
import flash.display.Sprite;
import flash.events.DRMAuthenticationCompleteEvent;
import flash.events.DRMAuthenticationErrorEvent;
import flash.events.DRMErrorEvent;
import flash.events.DRMStatusEvent;
import flash.events.NetStatusEvent;
import flash.net.NetConnection;
import flash.net.NetStream;
import flash.net.NetStreamPlayOptions;
import flash.net.drm.AuthenticationMethod;
import flash.net.drm.DRMContentData;
import flash.net.drm.DRMManager;
import flash.net.drm.LoadVoucherSetting;
public class DRMPreloader extends Sprite
{
    private var videoURL:String = "app-storage:/video.flv";
    private var userName:String = "user";
    private var password:String = "password";
    private var preloadConnection:NetConnection;
    private var preloadStream:NetStream;
    private var drmManager:DRMManager = DRMManager.getDRMManager();
    private var drmContentData:DRMContentData;
    public function DRMPreloader():void {
        drmManager.addEventListener( DRMAuthenticationCompleteEvent.AUTHENTICATION_COMPLETE,
onAuthenticationComplete );
        drmManager.addEventListener(
DRMAuthenticationErrorEvent.AUTHENTICATION_ERROR, onAuthenticationError );
        drmManager.addEventListener(DRMStatusEvent.DRM_STATUS, onDRMStatus);
        drmManager.addEventListener(DRMErrorEvent.DRM_ERROR, onDRMError);
        preloadConnection = new NetConnection();
        preloadConnection.addEventListener(NetStatusEvent.NET_STATUS, onConnect);
        preloadConnection.connect(null);
    }

    private function onConnect( event:NetStatusEvent ):void
    {
        preloadMetadata();
    }
    private function preloadMetadata():void
    {
        preloadStream = new NetStream( preloadConnection );
        preloadStream.client = this;
        var options:NetStreamPlayOptions = new NetStreamPlayOptions();
        options.streamName = videoURL;
        preloadStream.preloadEmbeddedData( options );
    }
    public function onDRMContentData( drmMetadata:DRMContentData ):void
```

```
{  
    drmContentData = drmMetadata;  
    if ( drmMetadata.authenticationMethod == AuthenticationMethod.USERNAME_AND_PASSWORD )  
    {  
        authenticateUser();  
    }  
    else  
    {  
        getVoucher();  
    }  
}  
private function getVoucher():void  
{  
    drmManager.loadVoucher( drmContentData, LoadVoucherSetting.ALLOW_SERVER );  
}  
  
private function authenticateUser():void  
{  
    drmManager.authenticate( drmContentData.serverURL, drmContentData.domain, userName, password );  
}  
private function onAuthenticationError( event:DRMAuthenticationErrorEvent ):void  
{  
    trace( "Authentication error: " + event.errorID + ", " + event.subErrorID );  
}  
  
private function onAuthenticationComplete( event:DRMAuthenticationCompleteEvent ):void  
{  
    trace( "Authenticated to: " + event.serverURL + ", domain: " + event.domain );  
    getVoucher();  
}  
private function onDRMStatus( event:DRMStatusEvent ):void  
{  
    trace( "DRM Status: " + event.detail );  
    trace("--Voucher allows offline playback = " + event.isAvailableOffline );  
    trace("--Voucher already cached = " + event.isLocal );  
    trace("--Voucher required authentication = " + !event.isAnonymous );  
}  
private function onDRMError( event:DRMErrorEvent ):void  
{  
    trace( "DRM error event: " + event.errorID + ", " + event.subErrorID + ", " + event.text );  
}  
public function onPlayStatus( info:Object ):void  
{  
    preloadStream.close();  
}  
}  
}
```

NetStream 类中与 DRM 相关的成员和事件

NetStream 类提供 Flash Player 或 AIR 应用程序与 Flash Media Server 或本地文件系统之间的单向流连接。(NetStream 类还支持渐进式下载。) NetStream 对象是 NetConnection 对象中的一个通道。在 AIR 应用程序中，NetStream 类调度四个与 DRM 相关的事件：

事件	说明
drmAuthenticate	在 DRMAuthenticateEvent 类中定义，此事件在 NetStream 对象尝试播放使用数字权限管理 (DRM) 加密的内容 (播放前需要用户凭据以进行身份验证) 时调度。此事件的属性包括 header、usernamePrompt、passwordPrompt 和 urlPrompt，这些属性可用于获取和设置用户凭据。此事件将重复发生，直到 NetStream 对象获得有效的用户凭据。
drmError	在 DRMErrorEvent 类中定义，并当在 NetStream 对象尝试播放使用数字权限管理 (DRM) 加密的文件的过程中遇到与 DRM 相关的错误时调度。例如，当用户授权失败时，将调度 DRM 错误事件对象。这可能是由于用户没有购买观看内容的权限，也可能是由于内容提供者不支持执行观看的应用程序。
drmStatus	在 DRMSatusEvent 类中定义，在开始播放使用数字权限管理 (DRM) 加密的内容时（如果用户已通过身份验证并获得播放内容的授权）调度。DRMSatusEvent 对象包含与凭证有关的信息，例如是否可以脱机使用内容或凭证何时过期而无法再观看内容。
status	在 events.StatusEvent 中定义，并仅在应用程序尝试通过调用 NetStream.play() 方法播放使用数字权限管理 (DRM) 加密的内容时调度。status 代码属性的值为 “DRM.encryptedFLV”。

NetStream 类包含以下特定于 DRM 的方法：

方法	说明
resetDRMVouchers()	删除所有本地缓存的数字权限管理 (DRM) 凭证数据。应用程序必须重新下载凭证，用户才能访问加密内容。 例如，以下代码将删除缓存中的所有凭证： <code>NetStream.resetDRMVouchers();</code>
setDRMAuthenticationCredentials()	将一组身份验证凭据（即用户名、密码和身份验证类型）传递给 NetStream 对象以进行身份验证。有效的身份验证类型为 “drm” 和 “proxy”。使用 “drm” 身份验证类型时，针对 FMRMS 对所提供的凭据进行身份验证。使用 “proxy” 身份验证类型时，针对代理服务器对凭据进行身份验证，且这些凭据必须与代理服务器所需的凭据相符。例如，当企业要求用户在进行针对代理服务器进行身份验证这样的步骤后才能访问 Internet 时，使用 proxy 选项可使应用程序针对代理服务器进行身份验证。除非使用匿名身份验证，否则在代理身份验证之后，用户仍必须针对 FMRMS 进行身份验证才能获取凭证并播放内容。可以再次使用 setDRMAuthenticationcredentials()（带有 “drm” 选项），针对 FMRMS 进行身份验证。
preloadEmbeddedMetadata()	分析本地媒体文件，从而得到嵌入的元数据。找到与 DRM 相关的元数据后，AIR 将调用 onDRMContentData() 回调函数。

此外，调用 preloadEmbeddedMetaDate() 方法后，NetStream 对象将调用 onDRMContentData() 和 onPlayStatus() 回调函数。在媒体文件中遇到 DRM 元数据后，将调用 onDRMContentData() 函数。文件分析完毕后，将调用 onPlayStatus() 函数。在分配给 NetStream 实例的 client 对象上必须定义 onDRMContentData() 和 onPlayStatus() 函数。如果使用同一个 NetStream 对象预加载凭证并播放内容，则必须在开始播放前等待由 preloadEmbeddedMetaDate() 生成的 onPlayStatus() 调用。

在以下代码中，将设置用户名 (“administrator”)、密码 (“password”) 和 “drm” 身份验证类型以对用户进行身份验证。setDRMAuthenticationCredentials() 方法提供的凭据必须与内容提供者已知并接受的凭据相匹配（与提供了观看内容所需权限的用户凭据相同）。此处不包含用于播放视频和确保已成功连接到视频流的代码。

使用 DRMSatusEvent 类

在使用数字权限管理 (DRM) 加密保护的内容成功开始播放时（在验证凭证以及在用户经过身份验证并获得观看内容的授权时），NetStream 对象将调度 DRMSatusEvent 对象。如果允许匿名用户访问，则还会为其调度 DRMSatusEvent。将检查凭证以验证是否允许不需要身份验证的匿名用户访问并播放内容。出于各种原因，可能会拒绝匿名用户进行访问。例如，匿名用户可能无法访问已过期的内容。

DRMStatusEvent 对象包含与凭证有关的信息，例如是否可以脱机使用内容或凭证何时过期而无法再观看内容。应用程序可使用此数据来传输用户的策略状态及其权限。

DRMStatusEvent 属性

DRMStatusEvent 类包含以下属性：

属性	说明
contentData	包含内容中嵌入的 DRM 元数据的 DRMContentData 对象。
detail	说明状态事件上下文的字符串。在 DRM 1.0 中，唯一的有效值为 DRM.voucherObtained。
isAnonymous	指示受 DRM 加密保护的内容是否在不需要用户提供身份验证凭据的情况下可供使用，如果是，则为 <code>true</code> ，否则为 <code>false</code> 。值为 <code>false</code> 表示用户提供的用户名和密码必须与内容提供者已知并需要的用户名和密码相匹配。
isAvailableOffline	指示受 DRM 加密保护的内容是否可以脱机使用，如果是，则为 <code>true</code> ，否则为 <code>false</code> 。为了使数字保护的内容可脱机使用，必须将其凭证缓存到用户的本地计算机中。
isLocal	指示本地是否缓存了播放内容所需的凭证。
offlineLeasePeriod	可以脱机观看内容的剩余天数。
policies	可能包含自定义 DRM 属性的自定义对象。
凭证	DRMVoucher。
voucherEndDate	凭证到期且内容不再可观看的绝对日期。

创建 DRMStatusEvent 处理函数

以下示例将创建一个事件处理函数，该事件处理函数可为启动了事件的 NetStream 对象输出 DRM 内容状态信息。将此事件处理函数添加到指向 DRM 加密内容的 NetStream 对象。

使用 DRMAuthenticateEvent 类

DRMAuthenticateEvent 对象在 NetStream 对象尝试播放使用数字权限管理 (DRM) 加密的内容（播放前需要用户凭据以进行身份验证）时调度。

DRMAuthenticateEvent 处理函数负责收集所需凭据（用户名、密码和类型）并将这些值传递给 NetStream.setDRMAuthenticationCredentials() 方法以进行验证。每个 AIR 应用程序都必须提供用于获取用户凭据的某种机制。例如，应用程序可以为用户提供一个简单的用户界面以输入用户名和密码值，以及类型值（可选）。AIR 应用程序还应提供一种机制以处理和限制重复的身份验证尝试次数。

DRMAuthenticateEvent 属性

DRMAuthenticateEvent 类包含以下属性：

属性	说明
authenticationType	指示提供的凭据是针对 FMRMS (“drm”) 进行身份验证，还是针对代理服务器 (“proxy”) 进行身份验证。例如，当企业要求用户在进行针对代理服务器进行身份验证这样的步骤后才能访问 Internet 时，使用 “proxy” 选项可使应用程序针对代理服务器进行身份验证。除非使用匿名身份验证，否则在代理身份验证之后，用户仍必须针对 FMRMS 进行身份验证才能获取凭证并播放内容。您可以再次与 “drm” 选项一起使用 setDRMAuthenticationcredentials() 以针对 FMRMS 进行身份验证。
header	服务器提供的加密内容文件标头。它包含有关加密内容的上下文的信息。
netstream	启动此事件的 NetStream 对象。
passwordPrompt	服务器提供的密码凭据提示。该字符串可以包括所需密码类型的说明。
urlPrompt	服务器提供的 URL 字符串提示。该字符串可以提供要将用户名和密码发送到的位置。
usernamePrompt	服务器提供的用户名凭据提示。该字符串可以包括所需用户名类型的说明。例如，内容提供者可能要求用电子邮件地址作为用户名。

创建 DRMAuthenticateEvent 处理函数

以下示例将创建一个事件处理函数，该事件处理函数将一组硬编码身份验证凭据传递给启动了事件的 NetStream 对象。（此处不包含用于播放视频和确保已成功连接到视频流的代码。）

```
var connection = new air.NetConnection();
connection.connect(null);

var videoStream = new air.NetStream();

videoStream.addEventListener(air.DRMAuthenticateEvent.DRM_AUTHENTICATE,
                           drmAuthenticateEventHandler)

function drmAuthenticateEventHandler(event)
{
    videoStream.setDRMAuthenticationCredentials("administrator", "password", "drm");
}
```

创建用于获取用户凭据的界面

在 DRM 内容要求进行用户身份验证的情况下，AIR 应用程序通常需要通过用户界面获取用户的身份验证凭据。

使用 DRMErrorEvent 类

如果在 NetStream 对象尝试播放数字权限管理 (DRM) 加密的文件时遇到与 DRM 相关的错误，则 AIR 将调度 DRMErrorEvent 对象。如果是用户凭据无效，则将通过重复调度 DRMAuthenticateEvent 对象来处理此错误，直到用户输入有效的凭据，或者 AIR 应用程序将拒绝进一步的尝试。应用程序应侦听任何其他 DRM 错误事件，以检测、标识和处理与 DRM 相关的错误。

当用户输入有效凭据时，他们仍有可能无法观看加密的内容，具体取决于 DRM 凭证条款。例如，如果用户尝试使用未经授权的应用程序（即未通过加密内容发行商验证的应用程序）观看内容。此时，将调度 DRMErrorEvent 对象。如果内容已损坏或者应用程序的版本与凭证指定的版本不匹配，则也会引发错误事件。应用程序必须提供适当的机制来处理错误。

DRMErrorEvent 属性

下表列出了 DRMErrorEvent 对象报告的错误：

主要错误代码	次要错误代码	错误详细信息	说明
1001	未使用		用户身份验证失败。
1002	未使用		Flash Media Rights Management Server (FMRMS) 不支持安全套接字层 (SSL)。
1003	未使用		内容已过期，不可再观看。
1004	未使用		用户授权失败。例如，如果用户尚未购买内容从而没有观看内容的权限，则会发生这种情况。
1005	未使用	Server URL	无法连接到服务器。
1006	未使用		需要更新客户端，即 Flash Media Rights Management Server (FMRMS) 需要新的数字权限管理 (DRM) 引擎。
1007	未使用		一般内部失败。
1008	详细解密错误代码		不正确的许可证密钥。
1009	未使用		FLV 内容已损坏。
1010	未使用	publisherID:applicationID	执行观看的应用程序的 ID 与内容发行商支持的有效 ID 不匹配。
1011	未使用		应用程序版本与策略中指定的版本不匹配。
1012	未使用		与加密内容关联的凭证验证失败，指示内容可能已损坏。
1013	未使用		无法保存与加密内容关联的凭证。
1014	未使用		FLV 标头完整性验证失败，指示内容可能已损坏。

主要错误代码	次要错误 ID	错误详细信息	说明
3300	Adobe Policy Server 错误代码		应用程序检测到与内容关联的凭证无效。
3301	未使用		用户身份验证失败。
3302	未使用		Flash Media Rights Management Server (FMRMS) 不支持安全套接字层 (SSL)。
3303	未使用		内容已过期，不可再观看。
3304	未使用		用户授权失败。即使用户通过身份验证，也可能发生这种情况，例如，如果用户尚未购买观看内容所需的权限。
3305	未使用	Server URL	无法连接到服务器。
3306	未使用		需要更新客户端，即 Flash Media Rights Management Server (FMRMS) 需要新的数字权限管理客户端引擎。
3307	未使用		一般内部数字权限管理失败。
3308	详细解密错误代码		不正确的许可证密钥。
3309	未使用		Flash 视频内容已损坏。

主要错误代码	次要错误 ID	错误详细信息	说明
3310	未使用	publisherID:applicationID	执行观看的应用程序的 ID 与内容发行商支持的有效 ID 不匹配。也就是说，内容提供者不支持执行观看的应用程序。
3311	未使用	min=x:max=y	应用程序版本与凭证中指定的版本不匹配。
3312	未使用		与加密内容关联的凭证验证失败，指示内容可能已损坏。
3313	未使用		无法将与加密内容关联的凭证保存到 Microsoft。
3314	未使用		FLV 标头完整性验证失败，指示内容可能已损坏。
3315	未使用		不允许远程播放 DRM 保护的内容。
3316	未使用		缺少 AdobeCP 模块。
3317	未使用		加载 AdobeCP 模块失败。
3318	未使用		找到的 AdobeCP 版本不兼容。
3319	未使用		缺少 AdobeCP API 入口点。
3320	未使用		AdobeCP 模块未经过身份验证。

创建 DRMErrorEvent 处理函数

以下示例将针对启动了事件的 NetStream 对象创建一个事件处理函数。如果 NetStream 在尝试播放 DRM 加密的内容时遇到错误，则会调用该事件处理函数。通常，当应用程序遇到错误时，该事件处理函数将执行许多清理任务，就错误通知用户，以及提供用于解决问题的选项。

```
function drmErrorHandler(event)
{
    air.trace(event.toString());
}
```

使用 DRMManager 类

在 AIR 应用程序中使用 DRMManager 类管理凭证和媒体权限服务器会话。AIR 1.5 版或更高版本中提供 DRMManager 类。

凭证管理

用户只要播放受 DRM 保护的联机媒体文件，AIR 就会获取并缓存查看内容所需的许可证凭证。如果应用程序在本地保存文件，并且凭证允许脱机播放，则即使无法与媒体权限服务器建立连接，用户也可以查看内容。使用 DRMManager 和 NetStream preloadEmbeddedMetadata() 方法，可以将凭证预先缓存，以使应用程序不必启动播放即可获取查看内容所需的许可证。例如，应用程序可以下载媒体文件，然后在用户仍联机时获取凭证。

若要预加载凭证，请使用 NetStream preloadEmbeddedMetadata() 方法获取 DRMContentData 对象。DRMContentData 对象包含可以提供许可证的媒体权限服务器的 URL 和域，并介绍是否需要对用户进行身份验证。借助此信息，可以调用 DRMManager loadVoucher() 方法来获取和缓存凭证。第 269 页的“[预加载用于脱机播放的凭证](#)”中更详细地介绍了预加载凭证所采用的工作流。

会话管理

还可以使用 DRMManager 验证用户对媒体权限服务器的身份以及管理持久性会话。

调用 DRMManager authenticate() 方法，与媒体权限服务器建立会话。成功完成身份验证后，DRMManager 将调度 DRMAuthenticationCompleteEvent 对象。此对象包含会话令牌。可以保存此令牌供将来建立会话，以使用户不必提供其帐户凭据。将令牌传递给 setAuthenticationToken() 方法以建立通过身份验证的新会话。（令牌到期等属性由生成令牌的服务器的设置确定。令牌的数据结构无意由 AIR 应用程序代码解释，并且在未来的 AIR 更新中可能更改。）

可以将通过身份验证的令牌转移到其他计算机。若要保护令牌，可以将其存储在 AIR 加密本地存储区中。有关详细信息，请参阅第 246 页的“[存储加密数据](#)”。

DRMStatus 事件

对 loadVoucher() 方法的调用成功完成后，DRMManager 将调度 DRMStatusEvent 对象。

如果获取了凭证，则事件对象的 detail 属性的值将为““DRM.voucherObtained”，而 voucher 属性将包含该 DRMVoucher 对象。

如果未获取凭证，则 detail 属性的值将仍为““DRM.voucherObtained”；但是，voucher 属性为 null。例如，如果使用 localOnly 的 LoadVoucherSetting，并且没有本地缓存的凭证，则不能获取凭证。

如果 loadVoucher() 调用未成功完成（可能由于身份验证或通信错误），则 DRMManager 改为调度 DRModelErrorEvent 对象。

DRMAuthenticationComplete 事件

通过调用 authenticate() 方法成功验证用户的身份后，DRMManager 将调度 DRMAuthenticationCompleteEvent 对象。

在 AIR 1.5 中，DRMAuthenticationCompleteEvent 对象包含一个可重用的标记，该标记可用于在应用程序会话中保持用户的身份验证。将此标记传递给 DRMManger setAuthenticationToken() 方法以重新建立会话。（到期等标记属性由标记创建者设置。AIR 不提供用于检查标记属性的 API。）

DRMAuthenticationError 事件

用户无法通过调用 authenticate() 或 setAuthenticationToken() 方法成功验证用户的身份时，DRMManager 将调度 DRMAuthenticationErrorEvent 对象。

使用 DRMContentData 类

DRMContentData 对象包含受 DRM 保护的媒体文件的 metadata 属性。DRMContentData 属性包含获取用于查看内容的许可证凭证所需的信息。

第 30 章：应用程序启动和退出选项

本节讨论有关启动已安装的 Adobe® AIR® 应用程序的选项和注意事项，以及有关关闭正在运行的应用程序的选项和注意事项。

有关启动和退出选项的其它在线信息

可以从以下源中查找有关启动和关闭应用程序时所使用 API 的详细信息：

快速入门（Adobe AIR 开发人员联盟）

- [启动选项](#)

语言参考

- [NativeApplication](#)
- [InvokeEvent](#)
- [InvokeEventReason](#)
- [BrowserInvokeEvent](#)

Adobe 开发人员联盟文章和范例

- [HTML 和 Ajax 的 Adobe AIR 开发人员联盟](#)

应用程序调用

在用户（或操作系统）执行以下操作时，将调用 AIR 应用程序：

- 从桌面解释程序启动该应用程序。
- 使用该应用程序作为命令行解释程序中的命令。
- 打开某类型文件而该应用程序是此类型文件的默认打开程序。
- (Mac OS X) 单击停靠任务栏中的该应用程序图标（无论应用程序当前是否正在运行）。
- 选择从安装程序启动该应用程序（在新安装过程结束时，或者在双击已安装应用程序的 AIR 文件之后）。
- 在已安装版本指示其自身正在处理应用程序更新（通过在应用程序描述符文件中加入 `<customUpdateUI>true</customUpdateUI>` 声明）时开始更新 AIR 应用程序。
- 访问承载 Flash 标志或应用程序的网页，该 Flash 标志或应用程序调用为 AIR 应用程序指定识别信息的 `com.adobe.air.AIR.launchApplication()` 方法。（应用程序描述符中还必须加入 `<allowBrowserInvocation>true</allowBrowserInvocation>` 声明，浏览器调用才能成功。）请参阅第 311 页的“[从浏览器启动安装的 AIR 应用程序](#)”。

每当调用 AIR 应用程序时，AIR 都会通过单一 NativeApplication 对象调度类型为 `invoke` 的 InvokeEvent 对象。若要给应用程序留出时间来初始化自身并注册事件侦听器，将对 `invoke` 事件进行排队而非将其丢弃。一旦侦听器已注册，就会传送所有排队的事件。

注：使用浏览器调用功能来调用某个应用程序时，如果该应用程序尚未运行，则 NativeApplication 对象将仅调度 `invoke` 事件。请参阅第 311 页的“[从浏览器启动安装的 AIR 应用程序](#)”。

若要接收 `invoke` 事件，请调用 `NativeApplication` 对象 (`NativeApplication.nativeApplication`) 的 `addEventListener()` 方法。当某个事件侦听器为 `invoke` 事件进行注册后，该事件侦听器还会接收到在注册前发生的所有 `invoke` 事件。调用 `addEventListener()` 返回之后，以较短间隔一次调度一个排入队列的 `invoke` 事件。如果在此过程中发生了新的 `invoke` 事件，则可能会在一个或多个排队的事件之前调度该事件。通过该事件队列可处理在初始化代码执行之前发生的任何 `invoke` 事件。请记住，如果在执行后期（在应用程序初始化之后）添加一个事件侦听器，该事件侦听器仍将会接收自应用程序启动以来发生的所有 `invoke` 事件。

仅启动 AIR 应用程序的一个实例。当再次调用某个已经运行的应用程序时，AIR 将向该正在运行的实例调度一个新的 `invoke` 事件。AIR 应用程序负责响应 `invoke` 事件并采取适当的动作（例如，打开一个新的文档窗口）。

`InvokeEvent` 对象包含任何传递给该应用程序的参数，以及已从中调用该应用程序的目录。如果该应用程序是由于文件类型关联而被调用，则文件的完整路径将包含在命令行参数中。同样，如果该应用程序是由于某个应用程序升级而被调用，则会提供升级 AIR 文件的完整路径。

当在一次操作中打开多个文件时，在 Mac OS X 中将调度一个 `InvokeEvent` 对象。每个文件都包括在 `arguments` 数组中。在 Windows 和 Linux 中将为每个文件调度一个单独的 `InvokeEvent` 对象。

应用程序可通过以下方法处理 `invoke` 事件：即向其 `NativeApplication` 对象注册侦听器，

```
NativeApplication.nativeApplication.addEventListener(InvokeEvent.INVOKE, onInvokeEvent);
air.NativeApplication.nativeApplication.addEventListener(air.InvokeEvent.INVOKE, onInvokeEvent);
```

然后定义事件侦听器：

```
var arguments;
var currentDir;
function onInvokeEvent(invocation) {
    arguments = invocation.arguments;
    currentDir = invocation.currentDirectory;
}
```

捕获命令行参数

与 AIR 应用程序调用关联的命令行参数是在由 `NativeApplication` 对象调度的 `invoke` 事件中进行传送的。

`InvokeEvent.arguments` 属性包含在调用 AIR 应用程序时由操作系统传递的参数数组。如果这些参数包含相对文件路径，则通常可以使用 `currentDirectory` 属性来解析路径。

除非用双引号引起，否则传递给 AIR 程序的参数会视为以空白分隔的字符串：

参数	数组
tick tock	{tick,tock}
tick "tick tock"	{tick,tick tock}
"tick" "tock"	{tick,tock}
\"tick\" \"tock\"	{"tick","tock"}

`InvokeEvent.currentDirectory` 属性包含一个表示应用程序启动时所在目录的 `File` 对象。

如果调用某应用程序是由于要打开该应用程序所注册类型的文件，则该文件的本机路径将以字符串的形式包含在命令行参数中。（您的应用程序负责打开该文件或对其执行预定操作。）同样，如果已对应用程序进行编程使其能实现自我更新（而非依赖于标准 AIR 更新用户界面），则当用户双击某个 AIR 文件（该文件包含具有匹配应用程序 ID 的应用程序）时，将会包含该 AIR 文件的本机路径。

可以使用 `currentDirectory` `File` 对象的 `resolve()` 方法访问该文件：

```
if((invokeEvent.currentDirectory != null)&&(invokeEvent.arguments.length > 0)){
    dir = invokeEvent.currentDirectory;
    fileToOpen = dir.resolvePath(invokeEvent.arguments[0]);
}
```

此外，还应验证参数是否的确是文件的路径。

示例：调用事件日志

下面的示例演示如何为 `invoke` 事件注册侦听器以及如何处理该事件。该示例会记录所有接收到的调用事件并显示当前目录和命令行参数。

注：此示例使用 `AIRAliases.js` 文件，该文件可在 SDK 的 `frameworks` 文件夹中找到。

```
<html>
<head>
<title>Invocation Event Log</title>
<script src="AIRAliases.js" />
<script type="text/javascript">
function appLoad() {
    air.trace("Invocation Event Log.");
    air.NativeApplication.nativeApplication.addEventListener(
        air.InvokeEvent.INVOKE, onInvoke);
}

function onInvoke(invokeEvent) {
    logEvent("Invoke event received.");
    if (invokeEvent.currentDirectory) {
        logEvent("Current directory=" + invokeEvent.currentDirectory.nativePath);
    } else {
        logEvent("--no directory information available--");
    }

    if (invokeEvent.arguments.length > 0) {
        logEvent("Arguments: " + invokeEvent.arguments.toString());
    } else {
        logEvent("--no arguments--");
    }
}

function logEvent(message) {
    var logger = document.getElementById('log');
    var line = document.createElement('p');
    line.innerHTML = message;
    logger.appendChild(line);
    air.trace(message);
}

window.unload = function() {
    air.NativeApplication.nativeApplication.removeEventListener(
        air.InvokeEvent.INVOKE, onInvoke);
}
</script>
</head>

<body onLoad="appLoad();">
    <div id="log"/>
</body>
</html>
```

登录时启动

通过将 NativeApplication startAtLogin 属性设置为 true，可以将 AIR 应用程序设置为在当前用户登录时自动启动。设置后，每当该用户登录时该应用程序都将自动启动。除非该设置更改为 false、用户通过操作系统手动更改该设置或者卸载了该应用程序，否则该应用程序将始终在登录时启动。登录时启动是一种运行时设置。该设置仅适用于当前用户。必须安装该应用程序以将 startAtLogin 属性成功设置为 true。如果该属性是在未安装应用程序时设置的，则会引发错误（例如，通过 ADL 启动该应用程序时）。

注：该应用程序在计算机系统启动时不会启动。它会在用户登录时启动。

若要确定应用程序是自动启动还是用户操作的结果，可以检查 InvokeEvent 对象的 reason 属性。如果该属性的值等于 InvokeEventReason.LOGIN，则应用程序为自动启动。对于任何其它调用路径，reason 属性的值等于 InvokeEventReason.STANDARD。若要访问 reason 属性，您的应用程序必须指向 AIR 1.5.1（通过在应用程序描述符文件中设置正确命名空间值）。

以下简化的应用程序利用 InvokeEvent reason 属性来确定在发生 invoke 事件时如何响应。如果 reason 属性为“login”，则应用程序仍在后台运行。否则，它将显示主要应用程序。采用此方式的应用程序通常在登录时启动（从而可以执行后台处理或事件监控），并打开窗口以响应用户触发的 invoke 事件。

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
try
{
    air.NativeApplication.nativeApplication.startAtLogin = true;
}
catch ( e )
{
    air.trace( "Cannot set startAtLogin: " + e.message );
}

air.NativeApplication.nativeApplication.addEventListener( air.InvokeEvent.INVOKE, onInvoke );

function onInvoke( event )
{
    if( event.reason == air.InvokeEventReason.LOGIN )
    {
        //do background processing...
        air.trace( "Running in background..." );
    }
    else
    {
        window.nativeWindow.activate();
    }
}
</script>
</head>
<body>
</body>
</html>
```

注：若要查看行为中的差异，请打包并安装该应用程序。只能为已安装的应用程序设置 startAtLogin 属性。

浏览器调用

使用浏览器调用功能，网站可启动要从浏览器启动的已安装 AIR 应用程序。仅在应用程序描述符文件将 `allowBrowserInvocation` 设置为 `true` 时，才允许浏览器调用：

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

有关应用程序描述符文件的详细信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

当该应用程序通过浏览器调用时，该应用程序的 `NativeApplication` 对象将会调度 `BrowserInvokeEvent` 对象。

若要接收 `BrowserInvokeEvent` 事件，请在 AIR 应用程序中调用 `NativeApplication` 对象 (`NativeApplication.nativeApplication`) 的 `addEventListener()` 方法。当某个事件侦听器针对 `BrowserInvokeEvent` 事件进行注册时，该事件侦听器还将接收在注册前发生的所有 `BrowserInvokeEvent` 事件。调用 `addEventListener()` 返回之后将调度这些事件，但此时并不一定在注册后可能收到其他 `BrowserInvokeEvent` 事件之前。这样就可处理在初始化代码执行之前（如从浏览器首次调用应用程序时）已发生的 `BrowserInvokeEvent` 事件。请记住，如果在执行后期（在应用程序初始化之后）添加一个事件侦听器，它仍然会接收到自应用程序启动以来发生的所有 `BrowserInvokeEvent` 事件。

`BrowserInvokeEvent` 对象包含以下属性：

属性	说明
<code>arguments</code>	要传递给应用程序的参数（字符串）数组。
<code>isHTTPS</code>	浏览器中的内容是否使用 https URL 方案。如果是，则为 <code>(true)</code> ，否则为 <code>(false)</code> 。
<code>isUserEvent</code>	浏览器调用是否生成用户事件（如鼠标单击）。在 AIR 1.0 中，它始终设置为 <code>true</code> ；AIR 需要与浏览器调用功能有关的用户事件。
<code>sandboxType</code>	浏览器中内容的沙箱类型。定义的有效值与可用于 <code>Security.sandboxType</code> 属性的值相同，可以是以下值之一： <ul style="list-style-type: none"> • <code>Security.APPLICATION</code> — 内容位于应用程序安全沙箱中。 • <code>Security.LOCAL_TRUSTED</code> — 内容位于只能与本地文件系统内容交互的安全沙箱中。 • <code>Security.LOCAL_WITH_FILE</code> — 内容位于只能与本地文件系统内容交互的安全沙箱中。 • <code>Security.LOCAL_WITH_NETWORK</code> — 内容位于只能与远程内容交互的安全沙箱中。 • <code>Security.REMOTE</code> — 内容位于远程（网络）域中。
<code>securityDomain</code>	浏览器中内容的安全域，如 " <code>www.adobe.com</code> " 或 " <code>www.example.org</code> "。仅对于远程安全沙箱中的内容（来自网络域的内容）设置此属性，而不对位于本地或应用程序安全沙箱中的内容设置此属性。

如果使用浏览器调用功能，请务必考虑安全性问题。网站启动 AIR 应用程序时，它可通过 `BrowserInvokeEvent` 对象的 `arguments` 属性发送数据。请在任何敏感操作（例如文件或代码加载 API）中谨慎使用此数据。风险级别取决于应用程序处理数据的方式。如果只希望某个特定网站调用该应用程序，则该应用程序应检查 `BrowserInvokeEvent` 对象的 `securityDomain` 属性。也可以要求调用该应用程序的网站使用 HTTPS，这样就可通过检查 `BrowserInvokeEvent` 对象的 `isHTTPS` 属性来进行验证。

应用程序应验证传入的数据。例如，如果某个应用程序要求传入指向某个特定域的 URL，则该应用程序应验证 URL 确实指向该域。这样就能够阻止攻击者欺骗该应用程序向其发送敏感数据。

任何应用程序都不应使用可能指向本地资源的 `BrowserInvokeEvent` 参数。例如，应用程序不应基于从浏览器传递的路径创建 `File` 对象。如果预计从浏览器传递远程路径，则应用程序应确保这些路径不使用 `file://` 协议来替代远程协议。

有关从浏览器调用应用程序的详细信息，请参阅第 311 页的“[从浏览器启动安装的 AIR 应用程序](#)”。

应用程序终止

终止应用程序最快的方法是调用 `NativeApplication.nativeApplication.exit()`，在应用程序没有要保存的数据或要清理的外部资源时适于使用此方法。调用 `exit()` 将关闭所有窗口，然后终止该应用程序。但是，若要允许窗口或应用程序的其他组件中断该终止进程（也许要保存重要数据），请在调用 `exit()` 之前调度适当的警告事件。

另一种妥善关闭应用程序的方法是提供单一执行路径，而不考虑关闭进程的启动方式。用户（或操作系统）可以通过以下方式触发应用程序终止：

- 在 `NativeApplication.nativeApplication.autoExit` 为 `true` 的情况下关闭最后一个应用程序窗口。
- 从操作系统中选择应用程序退出命令；例如，用户从默认菜单中选择退出应用程序命令。这种情况仅适用于 Mac OS；Windows 和 Linux 并不通过系统边栏提供应用程序退出命令。
- 关闭计算机。

当采用上述方式之一通过操作系统执行退出命令时，`NativeApplication` 将调度 `exiting` 事件。如果没有侦听器取消 `exiting` 事件，则任何打开的窗口都将关闭。每个窗口都会先调度一个 `closing` 事件，然后调度一个 `close` 事件。如果任何窗口取消 `closing` 事件，则关闭进程将会停止。

如果需要考虑应用程序的窗口关闭顺序，则可侦听来自 `NativeApplication` 的 `exiting` 事件并自行决定以适当的顺序关闭窗口。这种情况可能存在，例如，如果某个文档窗口具有工具调板时。如果系统关闭了调板而用户却决定取消退出命令以保存某些数据，这可能会带来不便，甚至更糟糕的情形。在 Windows 中，收到 `exiting` 事件的唯一时间是在关闭最后一个窗口之后（当 `NativeApplication` 对象的 `autoExit` 属性设置为 `true` 时）。

若要在所有平台上提供一致的行为，而不考虑退出顺序是通过操作系统边栏、菜单命令还是通过应用程序逻辑启动的，请遵守用于退出应用程序的以下良好做法：

- 1 始终先通过 `NativeApplication` 对象调度 `exiting` 事件，然后再从应用程序代码中调用 `exit()`，并检查应用程序的其他组件是否没有取消该事件。

```
function applicationExit() {
    var exitingEvent = new air.Event(air.Event.EXITING, false, true);
    air.NativeApplication.nativeApplication.dispatchEvent(exitingEvent);
    if (!exitingEvent.isDefaultPrevented()) {
        air.NativeApplication.nativeApplication.exit();
    }
}
```

- 2 侦听来自 `exiting` 对象的应用程序 `NativeApplication.nativeApplication` 事件，并在处理函数中关闭任何窗口（首先调度 `closing` 事件）。在所有窗口都已关闭后执行任何所需的清理任务，例如保存应用程序数据或删除临时文件。在清理期间仅使用同步方法以确保在应用程序退出之前能完成清理任务。

如果窗口关闭的顺序无关紧要，则可以遍历 `NativeApplication.nativeApplication.openedWindows` 数组并依次关闭每个窗口。如果顺序的确很重要，则需提供一种以正确顺序关闭窗口的方法。

```
function onExiting(exitingEvent) {
    var winClosingEvent;
    for (var i = 0; i < air.NativeApplication.nativeApplication.openedWindows.length; i++) {
        var win = air.NativeApplication.nativeApplication.openedWindows[i];
        winClosingEvent = new air.Event(air.Event.CLOSING, false, true);
        win.dispatchEvent(winClosingEvent);
        if (!winClosingEvent.isDefaultPrevented()) {
            win.close();
        } else {
            exitingEvent.preventDefault();
        }
    }

    if (!exitingEvent.isDefaultPrevented()) {
        //perform cleanup
    }
}
```

3 Windows 应始终通过侦听自己的 closing 事件来处理自己的清理任务。

4 在应用程序中仅使用一个 exiting 侦听器，因为调用时间较早的处理函数无法知道随后的处理函数是否将取消 exiting 事件（依赖于执行顺序将是不明智的做法）。

另请参阅

第 102 页的“[设置 AIR 应用程序属性](#)”

第 319 页的“[提供自定义应用程序更新用户界面](#)”

第 31 章：读取应用程序设置

在运行时，您可以获取应用程序描述符文件的属性及应用程序的发行商 ID。它们是在 NativeApplication 对象的 applicationDescriptor 和 publisherID 属性中设置的。

读取应用程序描述符文件

可以通过获取 NativeApplication 对象的 applicationDescriptor 属性来读取当前运行应用程序的应用程序描述符文件，如以下代码所示：

```
var appXml:XML = air.nativeApplication.nativeApplication.applicationDescriptor;
```

可以使用 DOMParser 对象分析数据，如以下代码所示：

```
var xmlString = air.NativeApplication.nativeApplication.applicationDescriptor;
var appXml = new DOMParser();
var xmlobj = appXml.parseFromString(xmlString, "text/xml");
var root = xmlobj.getElementsByTagName('application')[0];
var appId = root.getElementsByTagName("id")[0].firstChild.data;
var appVersion = root.getElementsByTagName("version")[0].firstChild.data;
var appName = root.getElementsByTagName("filename")[0].firstChild.data;
air.trace("appId:", appId);
air.trace("version:", appVersion);
air.trace("filename:", appName);
```

有关详细信息，请参阅第 102 页的“[应用程序描述符文件结构](#)”。

获取应用程序标识符和发行商标识符

应用程序 ID 和发行商 ID 一起唯一标识 AIR 应用程序。在应用程序描述符的 <id> 元素中指定应用程序 ID。发行商 ID 派生自用于对 AIR 安装包进行签名的证书。

可以从 NativeApplication 对象的 id 属性读取应用程序 ID，如以下代码所示：

```
air.trace(air.NativeApplication.nativeApplication.applicationID);
```

可以从 NativeApplication 的 publisherID 属性读取发行商 ID：

```
air.trace(air.NativeApplication.nativeApplication.publisherID);
```

注：用 ADL 运行 AIR 应用程序时，除非在 ADL 命令行中使用 -pubID 标志暂时指定一个发行商 ID，否则该应用程序没有发行商 ID。

还可以在已安装应用程序的安装目录中的 META-INF/AIR/publisherid 文件中找到应用程序的发行商 ID。

有关详细信息，请参阅第 313 页的“[关于 AIR 发行商标识符](#)”。

第 32 章：使用运行时和操作系统信息

本部分讨论 AIR 应用程序如何管理操作系统文件关联，如何检测用户活动以及如何获取 Adobe® AIR® 运行时的有关信息。

管理文件关联

必须在应用程序描述符中声明应用程序与文件类型之间的关联。在安装过程中，AIR 应用程序安装程序会将 AIR 应用程序关联为声明的各个文件类型的默认打开应用程序，但当文件类型已有与之关联的默认打开应用程序时除外。AIR 应用程序安装过程不会覆盖现有的文件类型关联。若要接管与其它应用程序的关联，请在运行时调用 `NativeApplication.setAsDefaultApplication()` 方法。

在启动应用程序时验证所需的文件关联是否就绪是一种很好的做法。这是因为 AIR 应用程序安装程序不会覆盖现有的文件关联，并且用户系统上的文件关联随时都可能发生更改。如果文件类型当前已关联到其它应用程序，则在接管现有关联之前询问用户是否更改文件关联也是一种很有礼貌的做法。

应用程序可以通过 `NativeApplication` 类的以下方法管理文件关联。各个方法都将文件类型扩展名视为一个参数：

方法	说明
<code>isSetAsDefaultApplication()</code>	如果 AIR 应用程序当前与指定的文件类型关联，则返回 <code>true</code> 。
<code>setAsDefaultApplication()</code>	在 AIR 应用程序与文件类型的打开操作之间建立关联。
<code>removeAsDefaultApplication()</code>	删除 AIR 应用程序与文件类型之间的关联。
<code>getDefaultApplication()</code>	报告当前与文件类型关联的应用程序的路径。

AIR 只能管理最初在应用程序描述符中声明的文件类型的关联。即使用户在文件类型与您的应用程序之间手动建立了关联，您也无法获取有关未声明文件类型的关联信息。使用未在应用程序描述符中声明的文件类型的扩展名调用任何文件关联管理方法，将导致应用程序引发运行时异常。

有关在应用程序描述符中声明文件类型的信息，请参阅第 108 页的“[声明文件类型关联](#)”。

获取运行时版本和修补级别

`NativeApplication` 对象有一个 `runtimeVersion` 属性，该属性为从中运行应用程序的运行时的版本（一个字符串，例如 "1.0.5"）。`NativeApplication` 对象还包含一个 `runtimePatchLevel` 属性，该属性表示运行时的修补级别（一个数字，例如 2960）。以下代码使用了这些属性：

```
air.trace(air.NativeApplication.nativeApplication.runtimeVersion);
air.trace(air.NativeApplication.nativeApplication.runtimePatchLevel);
```

检测 AIR 功能

对于与 Adobe AIR 应用程序捆绑的文件，`Security.sandboxType` 属性设置为由 `Security.APPLICATION` 常量定义的值。可以根据文件是否位于 Adobe AIR 安全沙箱中来加载内容（可以包含也可以不包含特定于 AIR 的 API），如以下代码所示：

```
if (window.runtime)
{
    if (air.Security.sandboxType == air.Security.APPLICATION)
    {
        alert("In AIR application security sandbox.");
    }
    else
    {
        alert("Not in AIR application security sandbox.")
    }
}
else
{
    alert("Not in the Adobe AIR runtime.")
}
```

对于未随 AIR 应用程序一起安装的所有资源，将根据其源域放在相应的安全沙箱中。例如，www.example.com 提供的内容放在与该域相应的安全沙箱中。

可以检查 `window.runtime` 属性是否设置为，从而查看内容是否在运行时中执行。

有关详细信息，请参阅第 89 页的“[AIR 安全性](#)”。

跟踪用户当前状态

`NativeApplication` 对象调度两个事件，可帮助检测用户是否正在使用计算机。如果在 `NativeApplication.idleThreshold` 属性指定的间隔内未检测到任何鼠标或键盘活动，则 `NativeApplication` 将调度 `userIdle` 事件。当发生下一次键盘或鼠标输入时，`NativeApplication` 对象将调度 `userPresent` 事件。`idleThreshold` 间隔以秒为单位，默认值为 300（5 分钟）。还可以通过 `NativeApplication.nativeApplication.lastUserInput` 属性获取自上一个用户输入以来经过的秒数。

以下代码行将空闲阈值设置为 2 分钟，并同时侦听 `userIdle` 和 `userPresent` 事件：

```
air.NativeApplication.nativeApplication.idleThreshold = 120;
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_IDLE, function(event) {
    air.trace("Idle");
});
air.NativeApplication.nativeApplication.addEventListener(air.Event.USER_PRESENT, function(event) {
    air.trace("Present");
});
```

注：在任意两个 `userIdle` 事件之间，只调度一个 `userPresent` 事件。

第 33 章：监视网络连接

Adobe® AIR® 提供了一种方法，可以检查安装 AIR 应用程序的计算机的网络连接是否发生更改。如果应用程序使用的数据是从网络获取的，则此信息非常有用。而且，应用程序可以检查网络服务的可用性。

检测网络连接更改

AIR 应用程序可以在具有不确定且不断更改的网络连接的环境中运行。为了有助于应用程序管理到在线资源的连接，每当网络连接变为可用或不可用时 Adobe AIR 都会发送一个网络更改事件。应用程序的 NativeApplication 对象会调度该网络更改事件。为了响应该事件，可添加一个侦听器：

```
air.NativeApplication.nativeApplication.addEventListener(air.Event.NETWORK_CHANGE, onNetworkChange);
```

并定义一个事件处理函数：

```
function onNetworkChange(event)
{
    //Check resource availability
}
```

Event.NETWORK_CHANGE 事件不是指示所有网络活动的更改，而是仅指示网络连接已更改。AIR 不尝试解释网络更改的含义。联网的计算机可能有许多真实和虚拟的连接，因此失去某个连接并不一定意味着失去了资源。而另一方面，新建连接也无法保证改善资源的可用性。有时，新建连接甚至可能阻止对之前可用资源的访问（例如，连接到 VPN 时）。

通常，应用程序确定其是否可连接到远程资源的唯一方法是，尝试连接该远程资源。为此，air.net 包中的服务监视框架为 AIR 应用程序提供了一个基于事件的方法，该方法可响应于指定主机的网络连接的更改。

注：服务监视框架检测服务器是否对请求进行接受响应。这不保证完全连接。通常，可扩展的 Web 服务使用缓存和负载平衡设备将流量重定向到 Web 服务器群集。在这种情况下，服务提供商仅提供对网络连接的局部诊断。

服务监视基础知识

服务监视器框架独立于 AIR 框架工作。对于基于 HTML 的 AIR 应用程序，servicemonitor.swf 必须包含在 AIR 应用程序包中。servicemonitor.swf 还必须包含在 AIR 应用程序代码中，如下所示：

```
<script src="servicemonitor.swf" type="application/x-shockwave-flash"/>
```

servicemonitor.swf 文件包含在 AIR SDK 的 frameworks 目录中。

ServiceMonitor 类实现用于监视网络服务的框架并为服务监视器提供基本功能。默认情况下，ServiceMonitor 类的实例会调度有关网络连接的事件。在创建该实例后以及每当 Adobe AIR 检测到网络更改时，ServiceMonitor 对象会调度这些事件。此外，可以设置 ServiceMonitor 实例的 pollInterval 属性进而以指定的间隔（以毫秒为单位）检查连接，而不考虑一般的网络连接事件。直到调用 start() 方法时，ServiceMonitor 对象才检查网络连接。

URLMonitor 类（ServiceMonitor 类的子类）可检测针对指定的 URLRequest 的 HTTP 连接的更改。

SocketMonitor 类（也是 ServiceMonitor 类的子类）可在指定的端口检测到指定主机的连接的更改。

检测 HTTP 连接

URLMonitor 类确定是否可从端口 80 (HTTP 通信的标准端口) 向指定地址发送 HTTP 请求。以下代码使用 URLMonitor 类的实例来检测到 Adobe 网站的连接更改：

```
<script src="servicemonitor.swf" type="application/x-shockwave-flash" />

<script>
    var monitor;
    function test()
    {
        monitor = new air.URLMonitor(new air.URLRequest('http://www.adobe.com'));
        monitor.addEventListener(air.StatusEvent.STATUS, announceStatus);
        monitor.start();
    }
    function announceStatus(e) {
        air.trace("Status change. Current status: " + monitor.available);
    }
</script>
```

检测套接字连接

AIR 应用程序也可将套接字连接用于推模式连接。防火墙和网络路由器通常会因某些安全原因而对未授权端口上的网络通信进行限制。因此，开发人员必须考虑用户可能不具有建立套接字连接能力。

类似于 URLMonitor 示例，以下代码使用 SocketMonitor 类的实例在 6667 (IRC 的常用端口) 处检测套接字连接的连接更改：

```
<script src="servicemonitor.swf" type="application/x-shockwave-flash" />

<script>
    function test()
    {
        socketMonitor = new air.SocketMonitor('www.adobe.com', 6667);
        socketMonitor.addEventListener(air.StatusEvent.STATUS, socketStatusChange);
        socketMonitor.start();
    }
    function socketStatusChange(e) {
        air.trace("Status change. Current status: " + socketMonitor.available);
    }
</script>
```

第 34 章：URL 请求和网络

本部分介绍用于实现应用程序与网络通信的 Adobe® AIR® 功能。另外，还介绍如何从外部源加载数据，在服务器和 AIR 应用程序之间发送消息，以及使用 `FileReference` 和 `FileReferenceList` 类执行文件上传和下载。

本部分介绍专门为在运行时中运行的应用程序提供的 AIR 网络和通信 API 功能。对 Web 浏览器中使用的 HTML 和 JavaScript 本身固有的网络和通信功能（例如 `XMLHttpRequest` 类所提供的功能）不予介绍。

网络和通信基础知识

当构建较复杂的 AIR 应用程序时，可能需要与服务器端脚本进行通信，或者从 Internet 上的 XML 文件或文本文件加载数据。运行时包含用于通过 Internet 收发数据的类；例如，从远程 URL 加载内容、与其它 AIR 应用程序进行通信以及连接到远程网站。

在 AIR 中，可以使用 `URLLoader` 和 `URLStream` 类加载外部文件。随后，根据加载的数据类型，可使用特定的类来访问数据。例如，如果远程内容采用名称 - 值对格式，则可以使用 `URLVariables` 类来分析服务器结果。或者，如果使用 `URLLoader` 和 `URLStream` 类加载的文件是远程 XML 文档，则可以使用 `DOMParser` 类来分析 XML 文档。这样，您便可以简化您的代码，因为无论是使用 `URLVariables`、`DOMParser` 还是某个其它类来分析和处理远程数据，用于加载外部文件的代码都是相同的。

运行时还包含用于其它类型的远程通信的类。这些类包括 `FileReference` 类（用于将文件上传到服务器以及从服务器下载文件）、`Socket` 和 `XMLSocket` 类（允许您通过套接字连接与远程计算机直接通信）以及 `NetConnection` 类（用于与远程应用程序服务器通信，例如 Flash Media Server 2）。

最后，运行时包含一个 `LocalConnection` 类，允许您在同一台计算机上运行的 AIR 应用程序（以及浏览器中的 SWF 文件）之间进行通信。

网络和通信术语

下表列出了重要的 AIR 网络和通信术语：

术语	说明
外部数据	在 AIR 应用程序外部以某些形式存储的数据，需要时可加载到应用程序中。可以将此数据存储在从服务器直接加载的文件中，或存储在数据库中，或者采用可通过调用服务器上运行的脚本或程序进行检索的其它形式进行存储。
URL 编码变量	URL 编码格式提供了一种在单个文本字符串中表示多个变量（变量名和值对）的方法。各个变量采用 <code>name=value</code> 格式。各个变量（即各个名称 - 值对）之间用 & 符隔开，如下所示： <code>variable1=value1&variable2=value2</code> 。这样，便可以将不限数量的变量作为一条消息进行发送。
MIME 类型	用于在 Internet 通信中标识给定文件类型的标准代码。任何给定文件类型都具有用于对其进行标识的特定代码。发送文件或消息时，计算机（如 Web 服务器或 AIR 应用程序）将指定要发送的文件类型。
HTTP	超文本传输协议，这是一种标准格式，用于传送通过 Internet 发送的网页和其它各种类型的内容。
请求方法	当程序（例如运行时或 Web 浏览器）向 Web 服务器发送消息（称为 HTTP 请求）时，可以使用请求方法（例如 GET 和 POST）将要发送的任何数据嵌入到请求中。在服务器端，接收请求的程序需要查看相应的请求部分以查找数据，因此，用于从 AIR 应用程序发送数据的请求方法应与用于在服务器上读取该数据的请求方法相匹配。
套接字连接	用于在两台计算机之间进行通信的永久连接。

使用 URLRequest 类

许多与网络相关的运行时 API 均使用 URLRequest 类来定义如何发出网络请求。使用 URLRequest 类可定义的内容不再仅仅局限于 URL 字符串。运行时中的内容可以使用新的 URL 方案（file 和 http 等标准方案除外）来定义 URL。

URLRequest 属性

URLRequest 类包括以下只可用于 AIR 应用程序安全沙箱中的内容的属性：

属性	说明
followRedirects	指定是否要遵循重定向（默认值为 true；如果不遵循，则为 false）。仅在运行时中支持此属性。
manageCookies	指定 HTTP 协议堆栈是否应管理此请求的 cookie（默认值为 true；如果不管理，则为 false）。仅在运行时中支持此属性。
authenticate	指定是否应为此请求处理身份验证请求（如果是，则为 true）。仅在运行时中支持此属性。默认值为对请求进行身份验证 — 如果服务器要求显示凭据，则可能会显示身份验证对话框。还可以设置用户名和密码，请参阅第 291 页的“ 设置 URLRequest 默认值 ”。
cacheResponse	指定是否应为此请求缓存成功的响应数据。仅在运行时中支持此属性。默认值为缓存响应（true）。
useCache	指定在此 URLRequest 获得数据之前是否应查询本地缓存。仅在运行时中支持此属性。默认值（true）为使用本地缓存版本（如果可用）。
userAgent	指定要在 HTTP 请求中使用的用户代理字符串。

URLRequest 对象的以下属性可以通过任何沙箱（不仅是 AIR 应用程序安全沙箱）中的内容进行设置：

属性	说明
contentType	使用 URL 请求发送的任何数据的 MIME 内容类型。
data	一个对象，它包含将随 URL 请求一起传输的数据。
digest	对缓存文件的安全“摘要”，用于跟踪 Adobe® Flash® Player 缓存。
method	控制 HTTP 请求方法，例如 GET 或 POST 操作。（AIR 应用程序安全域中运行的内容可以指定“GET”或“POST”之外的字符串作为 method 属性。允许使用任何 HTTP 动词，“GET”为默认方法。请参阅第 89 页的“ AIR 安全性 ”。）
requestHeaders	要追加到 HTTP 请求的 HTTP 请求标头的数组。
url	指定要请求的 URL。

注：HTMLLoader 类具有相关属性，用于设置与 HTMLLoader 对象加载的内容有关的设置。有关详细信息，请参阅关于 HTMLLoader 类。

设置 URLRequest 默认值

通过 URLRequestDefaults 类，可以定义 URLRequest 对象的默认设置。例如，以下代码设置 manageCookies 和 useCache 属性的默认值：

```
air.URLRequestDefaults.manageCookies = false;
air.URLRequestDefaults.useCache = false;
```

URLRequestDefaults 类包含 setLoginCredentialsForHost() 方法，使用该方法可指定要用于特定主机的默认用户名和密码。该方法的 hostname 参数中定义的主机可以为域（如 "www.example.com"）或域和端口号（如 "www.example.com:80"）。请注意，"example.com"、"www.example.com" 和 "sales.example.com" 均视为各自唯一的主机。

只有在服务器要求提供凭据时才会使用这些凭据。如果用户已进行了身份验证（例如，通过使用身份验证对话框），则无法通过调用 `setLoginCredentialsForHost()` 方法来更改已进行身份验证的用户。

例如，以下代码设置要在 `www.example.com` 上使用的默认用户名和密码：

```
air.URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

`URLRequestDefaults` 设置的各个属性仅应用于设置该属性的内容所在的应用程序域。而 `setLoginCredentialsForHost()` 方法则适用于 AIR 应用程序内所有应用程序域中的内容。通过此方法，应用程序可以使用指定的凭据登录到主机，并从而登录应用程序中的所有内容。

有关详细信息，请参阅[针对 HTML 开发人员的 Adobe AIR 语言参考](http://www.adobe.com/go/learn_air_html_jslr_cn)
(http://www.adobe.com/go/learn_air_html_jslr_cn) 中的 `URLRequestDefaults` 类。

在 URL 中使用 AIR URL 方案

在 AIR 的任何安全沙箱中定义 URL 时，可以使用以下标准 URL 方案：

http: 和 **https:**

使用方法同 Web 浏览器中的用法。

file:

使用此方案可指定相对于文件系统根目录的相对路径。例如：

```
file:///c:/AIR Test/test.txt
```

为应用程序安全沙箱中运行的内容定义 URL 时，还可以使用以下方案：

app:

使用此方案可指定相对于应用程序安装根目录（包含安装的应用程序的应用程序描述符文件的目录）的相对路径。例如，以下路径指向应用程序安装目录的 `resources` 子目录：

```
app:/resources
```

当在 ADL 应用程序中运行时，应用程序资源目录设置为包含应用程序描述符文件的目录。

app-storage:

使用此方案可指定相对于应用程序存储目录的相对路径。对于安装的每个应用程序，AIR 为每个用户都定义了唯一的应用程序存储目录，这些目录对于存储特定于各个应用程序的数据非常有用。例如，以下路径指向应用程序存储目录的 `settings` 子目录中的 `prefs.xml` 文件：

```
app-storage:/settings/prefs.xml
```

应用程序存储目录的位置由用户名、应用程序 ID 和发布者 ID 共同确定：

- 在 Mac OS 中，位于：

```
/Users/ 用户名/Library/Preferences/ 应用程序 ID. 发行商 ID/Local Store/
```

例如：

```
/Users/babbage/Library/Preferences/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1/Local Store
```

- 在 Windows 中，位于 `Documents and Settings` 目录下的以下位置：

```
用户名/Application Data/ 应用程序 ID. 发行商 ID/Local Store/
```

例如：

```
C:\Documents and Settings\babbage\Application  
Data\com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store
```

- 在 Linux 中位于:

```
/home/ 用户名/.appdata/ 应用程序 ID. 发行商 ID/Local Store/
```

例如:

```
/home/babbage/.appdata/com.example.TestApp.02D88EEED35F84C264A183921344EEA353A629FD.1\Local Store
```

通过 File.applicationStorageDirectory 创建的 File 对象的 URL (以及 url 属性) 使用 app-storage URL 方案, 如下所示:

```
var dir = air.File.applicationStorageDirectory;  
dir = dir.resolvePath("prefs.xml");  
air.trace(dir.url); // app-storage:/preferences
```

mailto:

在传递给 navigateToURL() 函数的 URLRequest 对象中可以使用 mailto 方案。请参阅第 300 页的“[在默认系统 Web 浏览器中打开 URL](#)”。

在 AIR 中使用 URL 方案

可以借助使用了以上任意 URL 方案的 URLRequest 对象来定义许多其它对象 (例如 FileStream 或 Sound 对象) 的 URL 请求。还可以在 AIR 中运行的 HTML 内容中使用这些方案; 例如, 可以在 img 标签的 src 属性中使用它们。

但在应用程序安全沙箱中的内容中, 只能使用这些特定于 AIR 的 URL 方案 (app: 和 app-storage:)。有关详细信息, 请参阅第 89 页的“[AIR 安全性](#)”。

禁止的 URL 方案

某些 API 允许在 Web 浏览器中启动内容。出于安全方面的考虑, 在 AIR 中使用这些 API 时将禁止某些 URL 方案。禁止的方案列表取决于使用 API 的代码所在的安全沙箱。有关详细信息, 请参阅第 300 页的“[在默认系统 Web 浏览器中打开 URL](#)”。

使用外部数据

运行时包含用于从外部源加载数据的机制。这些源可以是静态内容 (如文本文件), 也可以是动态内容 (如从数据库检索数据的 Web 脚本所生成的内容)。可以使用多种方法来设置数据的格式, 并且运行时提供了用于解码和访问数据的相关功能。也可以在检索数据的过程中将数据发送到外部服务器。

使用 URLLoader 类和 URLVariables 类

运行时包含用于加载外部数据的 URLLoader 和 URLVariables 类。URLLoader 类以文本、二进制数据或 URL 编码变量的形式从 URL 下载数据。URLLoader 类用于下载文本文件、XML 或其它要用于 AIR 应用程序中的信息。URLLoader 类采用运行时事件处理模型, 通过该模型可以侦听 complete、httpStatus、ioError、open、progress 和 securityError 等事件。

只有在下载完成后, URLLoader 数据才可用。尽管如果文件加载速度太快, 可能不会调度 progress 事件, 但可以通过侦听要调度的 progress 事件来监视下载进度 (已加载的字节数和总字节数)。成功下载文件后, 将调度 complete 事件。加载的数据将从 UTF-8 或 UTF-16 编码解码为字符串。

注: 如果没有为 URLRequest.contentType 设置值, 则以 application/x-www-form-urlencoded 的形式发送值。

`URLLoader.load()` 方法（以及 `URLLoader` 类的构造函数，可选）采用一个参数 `request`，该参数是 `URLRequest` 实例。`URLRequest` 实例包含单个 HTTP 请求的所有信息，如目标 URL、请求方法（例如 GET 或 POST）、附加标头信息以及 MIME 类型（例如，当上载 XML 内容时）。

例如，若要将 XML 数据包上载到服务器端脚本，可以使用以下代码：

```
var secondsUTC = new Date().time;
var dataXML = "<login>" +
    + "<time>" + secondsUTC + "</time>" +
    + "<username>Ernie</username>" +
    + "<password>guru</password>" +
    + "</login>";
var request = new air.URLRequest("http://www.example.com/login.cfm");
request.contentType = "text/xml";
request.data = dataXML;
request.method = air.URLRequestMethod.POST;
var loader = new air.URLLoader();
loader.load(request);
```

上面的代码创建了一个名为 `dataXML` 的 XML 实例，其中包含要发送到服务器的 XML 数据包。接下来，将 `URLRequest` `contentType` 属性设置为 "text/xml"，并将 `URLRequest` `data` 属性设置为 XML 数据包的内容。最后，创建一个 `URLLoader` 实例，并使用 `URLLoader.load()` 方法向远程脚本发送请求。

可以使用三种方式指定要在 URL 请求中传递的参数：

- 在 `URLVariables` 构造函数中
- 在 `URLVariables.decode()` 方法中
- 作为 `URLVariables` 对象本身中的特定属性

当定义 `URLVariables` 构造函数或 `URLVariables.decode()` 方法中的变量时，需要确保对 & 号字符进行 URL 编码，因为它具有特殊含义，并可作为分隔符。例如，由于 & 号作为参数的分隔符，因此当传递 & 号时，需要对 & 号进行 URL 编码，从 & 更改为 %26。

从外部网络文档加载数据

以下代码段创建 `URLRequest` 和 `URLLoader` 对象，用于加载网络上的文本文件的内容：

```
var request = new air.URLRequest("http://www.example.com/data/params.txt");
var loader = new air.URLLoader();
loader.load(request);
```

默认情况下，如果未定义请求方法，运行时将使用 HTTP GET 方法来加载内容。如果要使用 POST 方法发送数据，则需要使用静态常量 `URLRequestMethod.POST` 将 `request.method` 属性设置为 POST，如下所示：

```
var request = new air.URLRequest("http://www.example.com/sendfeedback.cfm");
request.method = air.URLRequestMethod.POST;
```

在运行时加载的外部文档 `params.txt` 包含以下数据：

```
monthNames=January,February,March,April,May,June,July,August,September,October,November,December&dayNames
=Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
```

该文件包含两个参数，即 `monthNames` 和 `dayNames`。每个参数包含一个逗号分隔列表，该列表被分析为字符串。可以使用 `String.split()` 方法将此列表拆分为数组。

 提示：不要将保留字或语言构造作为外部数据文件中的变量名称，因为这样做会使代码的读取和调试变得困难。

加载数据后，将调度 `Event.COMPLETE` 事件，随后将可以在 `URLLoader` 的 `data` 属性中使用外部文档的内容，如以下代码所示：

```

function completeHandler(event)
{
    var loader2 = event.target;
    air.trace(loader2.data);
}

```

如果远程文档包含名称 - 值对，则可以通过传入加载文件的内容，使用 **URLVariables** 类来分析数据，如下所示：

```

function completeHandler(event)
{
    var loader2 = event.target;
    var variables = new air.URLVariables(loader2.data);
    air.trace(variables.dayNames);
}

```

外部文件中的各个名称 - 值对都创建为 **URLVariables** 对象中的一个属性。在上面的代码范例中，变量对象中的各个属性都被视为字符串。如果名称 - 值对的值是一个项目列表，则可以通过调用 **String.split()** 方法将字符串转换为数组，如下所示：

```
var dayNameArray = variables.dayNames.split(",");
```

 如果从外部文本文件加载数值数据，则需要使用顶级函数（如 **parseInt()**、**parseFloat()** 和 **Number()**）将这些值转换为数字值。

无需将远程文件的内容作为字符串加载和创建 **URLVariables** 对象，可以将 **URLLoader.dataFormat** 属性设置为在 **URLLoaderDataFormat** 类中找到的静态属性之一。**URLLoader.dataFormat** 属性的三个可能值如下：

值	说明
air.URLLoaderDataFormat.BINARY	URLLoader.data 属性包含 ByteArray 对象中存储的二进制数据。
air.URLLoaderDataFormat.TEXT	URLLoader.data 属性包含 String 对象中的文本。
air.URLLoaderDataFormat.VARIABLES	URLLoader.data 属性包含 URLVariables 对象中存储的 URL 编码变量。

以下代码演示如何将 **URLLoader.dataFormat** 属性设置为 **air.URLLoaderDataFormat.VARIABLES**，以便允许您将加载的数据自动分析为 **URLVariables** 对象：

```

var request = new air.URLRequest("http://www.example.com/params.txt");
var variables = new air.URLLoader();
variables.dataFormat = air.URLLoaderDataFormat.VARIABLES;
variables.addEventListener(air.Event.COMPLETE, completeHandler);
try
{
    variables.load(request);
}
catch (error)
{
    air.trace("Unable to load URL: " + error);
}

function completeHandler(event)
{
    var loader = event.target;
    air.trace(loader.data.dayNames);
}

```

注：**URLLoader.dataFormat** 的默认值为 **air.URLLoaderDataFormat.TEXT**。

如以下示例所示，从外部文件加载 XML 与加载 **URLVariables** 相同。可以创建 **URLRequest** 和 **URLLoader** 实例，然后使用它们下载远程 XML 文档。文件下载完毕时，将调度 **complete** 事件，且 **trace()** 函数会将该文件的内容输出到命令行。

```

var request = new air.URLRequest("http://www.example.com/data.xml");
var loader = new air.URLLoader();
loader.addEventListener(air.Event.COMPLETE, completeHandler);
loader.load(request);

function completeHandler(event)
{
    var dataXML = event.target.data;
    air.trace(dataXML);
}

```

与外部脚本进行通信

除了加载外部数据文件，还可以使用 **URLVariables** 类将变量发送到服务器并处理服务器的响应。这是非常有用的，例如，如果您正在编写游戏，想要将用户的得分发送到服务器以计算是否应添加到高分列表中，甚至想要将用户的登录信息发送到服务器以进行验证。服务器端脚本可以处理用户名和密码，向数据库验证用户名和密码，然后返回用户提供的凭据是否有效的确认。

以下代码段创建一个名为 **variables** 的 **URLVariables** 对象，该对象创建一个称为 **name** 的变量。接下来，创建一个 **URLRequest** 对象，该对象指定变量要发送到的服务器端脚本的 URL。然后，设置 **URLRequest** 对象的 **method** 属性，以便将变量作为 HTTP POST 请求发送。为了将 **URLVariables** 对象添加到 URL 请求，需要将 **URLRequest** 对象的 **data** 属性设置为早先创建的 **URLVariables** 对象。最后，创建 **URLLoader** 实例并调用 **URLLoader.load()** 方法，此方法用于启动请求。

```

var variables = new air.URLVariables("name=Franklin");
var request = new air.URLRequest();
request.url = "http://www.example.com/greeting.cfm";
request.method = air.URLRequestMethod.POST;
request.data = variables;
var loader = new air.URLLoader();
loader.dataFormat = air.URLLoaderDataFormat.VARIABLES;
loader.addEventListener(air.Event.COMPLETE, completeHandler);
try
{
    loader.load(request);
}
catch (error)
{
    air.trace("Unable to load URL");
}

function completeHandler(event)
{
    air.trace(event.target.data.welcomeMessage);
}

```

以下代码包含上例中使用的 Adobe® ColdFusion® greeting.cfm 文档的内容：

```

<cfif NOT IsDefined("Form.name") OR Len(Trim(Form.Name)) EQ 0>
    <cfset Form.Name = "Stranger" />
</cfif>
<cfoutput>welcomeMessage=#UrlEncodedFormat("Welcome, " & Form.name)#
</cfoutput>

```

使用 **URLStream** 类

URLStream 类提供对下载 URL 的低级访问。数据下载后便可立即使用，不必像使用 **URLLoader** 时那样，需要等待整个数据发送结束。并且 **URLStream** 类还允许在完成下载前关闭流。已下载文件的内容将作为原始二进制数据提供。

URLStream 中的读取操作是非阻塞模式的。这意味着在读取数据之前必须使用 bytesAvailable 属性来确定是否能够获得足够的数据。如果不能获得足够的数据，将引发 EOFError 异常。

URLStream 类在传送任何响应数据之前将调度 httpResponseStatus 事件。httpResponseStatus 事件（在 HTTPStatusEvent 类中定义）包含 responseURL 属性（从中返回响应的 URL）和 responseHeaders 属性（表示响应返回的响应标头的 URLRequestHeader 对象的数组）。

套接字连接

在运行时中，可以使用两种不同类型的套接字连接：XML 套接字连接和二进制套接字连接。

使用 XML 套接字，可以连接到远程服务器并创建服务器连接，该连接在显式关闭之前一直保持打开状态。这样，无需不断地打开新的服务器连接，就可以在服务器与客户端之间交换 XML 数据。使用 XML 套接字服务器的另一个好处是用户不需要显式请求数据。您无需请求即可从服务器发送数据，并且可以将数据发送到连接到 XML 套接字服务器的每个客户端。

二进制套接字连接与 XML 套接字类似，不同之处是客户端与服务器不需要专门交换 XML 数据包。该连接可以将数据作为二进制信息传输。这样，您就可以连接到各种各样的服务，包括邮件服务器（POP3、SMTP 和 IMAP）和新闻服务器（NNTP）。

Socket 类

使用 Socket 类，AIR 应用程序可以建立套接字连接并读取和写入原始二进制数据。它与 XMLSocket 类相似，但未指定接收和传输数据的格式。在与使用二进制协议的服务器进行交互操作时，Socket 类非常有用。使用二进制套接字连接，可以编写允许与多个不同的 Internet 协议（例如 POP3、SMTP、IMAP 和 NNTP）进行交互的代码。反过来，这又会使 AIR 应用程序能够连接到邮件和新闻服务器。

AIR 应用程序可通过使用服务器的二进制协议直接与该服务器连接。某些服务器使用 big-endian 字节顺序，某些服务器则使用 little-endian 字节顺序。Internet 上的大多数服务器使用 big-endian 字节顺序，因为“网络字节顺序”为 big-endian。您应使用与收发数据的服务器的字节顺序相匹配的 endian 字节顺序。在默认情况下，所有操作都是以 Big-endian 格式编码的；即，最高位字节位于第一位。这样做是为了匹配 Java 和官方网络字节顺序。若要更改是使用 big-endian 还是使用 little-endian 字节顺序，可以将 endian 属性设置为 airEndian.BIG_ENDIAN 或 airEndian.LITTLE_ENDIAN。



提示：Socket 类继承 IDataInput 和 IDataOutput 接口（位于 flash.utils 包中）实现的所有方法，应使用这些方法从 Socket 读取数据和向其中写入数据。

XMLSocket 类

运行时提供了一个内置的 XMLSocket 类，使用它可以打开与服务器的持续连接。这种打开的连接消除了反应时间问题，它通常用于实时的应用程序，例如聊天应用程序或多人游戏。传统的基于 HTTP 的聊天解决方案频繁轮询服务器，并使用 HTTP 请求来下载新的消息。与此相对照，XMLSocket 聊天解决方案保持与服务器的打开连接，这一连接允许服务器即时发送传入的消息，而无需客户端发出请求。

若要创建套接字连接，必须创建服务器端应用程序来等待套接字连接请求，然后向 AIR 应用程序发送响应。可以使用 Java、Python 或 Perl 等编程语言来编写这种类型的服务器端应用程序。若要使用 XMLSocket 类，服务器计算机必须运行可识别 XMLSocket 类使用的协议的守护程序。下面的列表说明了该协议：

- XML 消息通过全双工 TCP/IP 流套接字连接发送。
- 每个 XML 消息都是一个完整的 XML 文档，以一个零 (0) 字节结束。
- 通过 XMLSocket 连接发送和接收的 XML 消息的数量没有限制。

注：XMLSocket 类不能自动穿过防火墙，因为 XMLSocket 没有 HTTP 隧道功能（这与实时消息传递协议（RTMP）不同）。如果您需要使用 HTTP 隧道，应考虑改用 Flash Remoting 或 Adobe® Flash® Media® Server（支持 RTMP）。

对于应用程序安全沙箱外部的内容，如何使用 `XMLSocket` 对象连接到服务器以及连接到哪些端口受以下限制：

- 对于应用程序安全沙箱外部的内容，`XMLSocket.connect()` 方法只能连接到大于或等于 1024 的 TCP 端口号。这种限制所带来的后果之一是，向与 `XMLSocket` 对象通信的服务器守护程序分配的端口号也必须大于等于 1024。端口号小于 1024 的端口通常用于系统服务，例如 FTP (21)、Telnet (23)、SMTP (25)、HTTP (80) 和 POP3 (110)，因此，出于安全方面的考虑，禁止 `XMLSocket` 对象使用这些端口。这种端口号方面的限制可以减少不恰当地访问和滥用这些资源的可能性。
- 对于应用程序安全沙箱外部的内容，`XMLSocket.connect()` 方法只能连接到该内容所在同一域中的计算机。（此限制与 `URLLoader.load()` 的安全规则相同。）若要连接到在内容所在域之外的其它域中运行的服务器守护程序，可以在该服务器上创建一个允许从特定域进行访问的跨域策略文件。有关跨域策略文件的详细信息，请参阅 第 89 页的“[AIR 安全性](#)”。

注：将服务器设置为与 `XMLSocket` 对象进行通信可能会遇到一些困难。如果您的应用程序不需要进行实时交互，请使用 `URLLoader` 类，而不要使用 `XMLSocket` 类。

可以使用 `XMLSocket` 类的 `XMLSocket.connect()` 和 `XMLSocket.send()` 方法，通过套接字连接与服务器之间往返传输 XML。`XMLSocket.connect()` 方法与 Web 服务器端口建立套接字连接。`XMLSocket.send()` 方法将 XML 对象传递给套接字连接中指定的服务器。

调用 `XMLSocket.connect()` 方法时，运行时将打开与服务器的 TCP/IP 连接，并使该连接保持打开状态，直到发生以下情况之一：

- 调用 `XMLSocket` 类的 `XMLSocket.close()` 方法。
- 对 `XMLSocket` 对象的引用不再存在。
- 连接中断（例如，调制解调器断开连接）。

创建并连接到 Java XML 套接字服务器

下面的代码演示了一个用 Java 编写的简单 `XMLSocket` 服务器，该服务器接受传入连接并在命令提示窗口中显示接收到的消息。虽然从命令行启动服务器时可以指定其它端口号，但默认情况下，在本地计算机上的 8080 端口创建新服务器。

创建一个文本文档并添加以下代码：

```
import java.io.*;
import java.net.*;

class SimpleServer
{
    private static SimpleServer server;
    ServerSocket socket;
    Socket incoming;
    BufferedReader readerIn;
    PrintStream printOut;

    public static void main(String[] args)
    {
        int port = 8080;

        try
        {
            port = Integer.parseInt(args[0]);
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            // Catch exception and keep going.
        }

        server = new SimpleServer(port);
    }
}
```

```

private SimpleServer(int port)
{
    System.out.println(">> Starting SimpleServer");
    try
    {
        socket = new ServerSocket(port);
        incoming = socket.accept();
        readerIn = new BufferedReader(new
            InputStreamReader(incoming.getInputStream()));
        printOut = new PrintStream(incoming.getOutputStream());
        printOut.println("Enter EXIT to exit.\r");
        out("Enter EXIT to exit.\r");
        boolean done = false;
        while (!done)
        {
            String str = readerIn.readLine();
            if (str == null)
            {
                done = true;
            }
            else
            {
                out("Echo: " + str + "\r");
                if(str.trim().equals("EXIT"))
                {
                    done = true;
                }
            }
            incoming.close();
        }
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}

private void out(String str)
{
    printOut.println(str);
    System.out.println(str);
}
}

```

将文档保存到硬盘，命名为 **SimpleServer.java** 并使用 Java 编译器对其进行编译，这会创建一个名为 **SimpleServer.class** 的 Java 类文件。

您可以通过打开命令提示并键入 `java SimpleServer` 来启动 XMLSocket 服务器。**SimpleServer.class** 文件可以位于本地计算机或网络上的任何位置，不需要放置在 Web 服务器的根目录中。

 提示：如果由于文件没有位于 Java classpath 中而无法启动服务器，请尝试使用 `java -classpath . SimpleServer` 启动服务器。

若要从 AIR 应用程序连接到 XMLSocket，需要创建一个 XMLSocket 类实例，并在传递主机名和端口号时调用 `XMLSocket.connect()` 方法，如下所示：

```

var xmlsock = new air.XMLSocket();
xmlsock.connect("127.0.0.1", 8080);

```

只要从服务器接收数据，就会调度 `data` 事件：

```
xmlsock.addEventListener(window.runtime.flash.events.DataEvent.DATA, onData);
function onData(event)
{
    air.trace("[" + event.type + "] " + event.data);
}
```

若要向 XMLSocket 服务器发送数据，可以使用 `XMLSocket.send()` 方法并传递一个字符串。运行时将内容发送到 XMLSocket 服务器，后跟一个零 (0) 字节：

```
xmlsock.send(xmlFormattedData);
```

`XMLSocket.send()` 方法不返回指示数据是否成功传输的值。如果尝试发送数据时发生错误，将引发 `IOError` 错误。

 提示：发送到 XML 套接字服务器的每条消息必须以换行符 (`\n`) 结束。

在默认系统 Web 浏览器中打开 URL

可以使用 `navigateToURL()` 函数在默认系统 Web 浏览器中打开 URL。对于作为此函数的 `request` 参数传递的 `URLRequest` 对象，仅使用 `url` 属性。

`navigateToURL()` 函数的第一个参数（即 `navigate` 参数）是一个 `URLRequest` 对象（请参阅第 291 页的“[使用 URLRequest 类](#)”）。第二个参数是可选的 `window` 参数，您可以使用该参数指定窗口名称。例如，以下代码在默认系统浏览器中打开 www.adobe.com 网站：

```
var url = "http://www.adobe.com";
var urlReq = new air.URLRequest(url);
air.navigateToURL(urlReq);
```

注：使用 `navigateToURL()` 函数时，运行时将使用 POST 方法（其 `method` 属性设置为 `URLRequestMethod.POST` 的方法）的 `URLRequest` 对象视为使用 GET 方法。

使用 `navigateToURL()` 函数时，将根据调用 `navigateToURL()` 函数的代码所在的安全沙箱来允许 URL 方案。

某些 API 允许在 Web 浏览器中启动内容。出于安全方面的考虑，在 AIR 中使用这些 API 时将禁止某些 URL 方案。禁止的方案列表取决于使用 API 的代码所在的安全沙箱。（有关安全沙箱的详细信息，请参阅 第 89 页的“[AIR 安全性](#)”。）

应用程序沙箱

允许以下方案。使用方法同 Web 浏览器中的用法。

- http:
- https:
- file:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
- app:
- app-storage:

禁止其它所有 URL 方案。

远程沙箱

允许以下方案。使用方法同 Web 浏览器中的用法。

- http:
- https:

- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
禁止其它所有 URL 方案。

只能与本地文件系统内容交互的沙箱
允许以下方案。使用方法同 Web 浏览器中的用法。

- file:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
禁止其它所有 URL 方案。

只能与远程内容交互的沙箱
允许以下方案。使用方法同 Web 浏览器中的用法。

- http:
- https:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
禁止其它所有 URL 方案。

受信任的本地沙箱
允许以下方案。使用方法同 Web 浏览器中的用法。

- file:
- http:
- https:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
禁止其它所有 URL 方案。

向服务器发送 URL

可以使用 `sendToURL()` 函数向服务器发送 URL 请求。此函数忽略任何服务器响应。`sendToURL()` 函数采用一个参数 `request`，该参数是一个 `URLRequest` 对象（请参阅第 291 页的“[使用 URLRequest 类](#)”）。下面是一个示例：

```
var url = "http://www.example.com/application.jsp";
var variables = new air.URLVariables();
variables.sessionId = new Date().getTime();
variables.userLabel = "Your Name";
var request = new air.URLRequest(url);
request.data = variables;
air.sendToURL(request);
```

此示例使用 `URLVariables` 类将变量数据包含到 `URLRequest` 对象中。有关详细信息，请参阅第 293 页的“[使用 URLLoader 类和 URLVariables 类](#)”。

第 35 章：应用程序间的通信

`LocalConnection` 类支持在 Adobe® AIR® 应用程序之间以及在 AIR 应用程序和浏览器中运行的 SWF 内容之间进行通信。

关于 `LocalConnection` 类

`LocalConnection` 对象只能在同一客户端计算机上运行的 AIR 应用程序和 SWF 文件间进行通信，但是它们可以在不同的应用程序中运行。例如，两个 AIR 应用程序可以使用 `LocalConnection` 类进行通信，和 AIR 应用程序与浏览器中运行的 SWF 文件之间的通信原理相同。

最简便的 `LocalConnection` 对象使用方法是只允许位于同一个域或同一 AIR 应用程序中的 `LocalConnection` 对象之间进行通信。这样，您就不必担心安全方面的问题。但如果需要在不同域之间进行通信，则可采用多种方法来实施安全措施。有关详细信息，请参阅 [Adobe AIR 语言参考](#) 中 `LocalConnection` 类列表中对 `send()` 方法的 `connectionName` 参数以及 `allowDomain()` 和 `domain` 条目的讨论。

若要向 `LocalConnection` 对象添加回调方法，请将 `LocalConnection.client` 属性设置为具有成员方法的对象，如以下代码所示：

```
var lc = new air.LocalConnection();
var clientObject = new Object();
clientObject.doMethod1 = function() {
    air.trace("doMethod1 called.");
}
clientObject.doMethod2 = function(param1) {
    air.trace("doMethod2 called with one parameter: " + param1);
    air.trace("The square of the parameter is: " + param1 * param1);
}
lc.client = clientObject;
```

`LocalConnection.client` 属性包含可调用的所有回调方法。

在两个应用程序之间发送消息

可以使用 `LocalConnection` 类在不同的 AIR 应用程序之间以及在 AIR 应用程序与浏览器中运行的 Adobe® Flash® Player (SWF) 应用程序之间进行通信。

以下代码定义了一个用作服务器的 `LocalConnection` 对象，负责接受从其它应用程序传入的 `LocalConnection` 调用：

```
var lc = new air.LocalConnection();
lc.connect("connectionName");
var clientObject = new Object();
clientObject.echoMsg = function(msg) {
    air.trace("This message was received: " + msg);
}
lc.client = clientObject;
```

此代码首先创建一个名为 `lc` 的 `LocalConnection` 对象，然后将 `client` 属性设置为对象 `clientObject`。当其它应用程序调用此 `LocalConnection` 实例中的方法时，AIR 会查找 `clientObject` 对象中的该方法。

如果已存在具有指定名称的连接，则会引发 `ArgumentError` 异常，指出由于已经连接了该对象，连接尝试失败。

以下代码段说明如何创建名为 conn1 的 LocalConnection:

```
connection.connect("conn1");
```

从辅助应用程序连接到主应用程序需要您首先在发送方 LocalConnection 对象中创建一个 LocalConnection 对象，然后使用连接名称和要执行的方法的名称来调用 LocalConnection.send() 方法。例如，要连接到先前创建的 LocalConnection 对象，可以使用以下代码：

```
sendingConnection.send("conn1", "echoMsg", "Bonjour.");
```

此代码使用连接名称 conn1 连接到底层现有 LocalConnection 对象，并调用远程应用程序中的 doMessage() 方法。如果要向远程应用程序发送参数，请在 send() 方法中的方法名称后指定附加参数，如以下代码段所示：

```
sendingConnection.send("conn1", "doMessage", "Hello world");
```

连接到不同域中的内容和其它 AIR 应用程序

若要只允许从特定域进行通信，请调用 LocalConnection 类的 allowDomain() 或 allowInsecureDomain() 方法，并传递允许访问此 LocalConnection 对象的一个或多个域组成的列表，同时传递允许的一个或多个域名。

可以向 LocalConnection.allowDomain() 和 LocalConnection.allowInsecureDomain() 方法传递两个特殊值：* 和 localhost。星号值 (*) 表示允许从所有域进行访问。字符串 localhost 允许将从应用程序资源目录之外的本地安装内容调用应用程序。

如果 LocalConnection.send() 方法尝试从调用方代码无权访问的安全沙箱与应用程序通信，则会调度 securityError 事件 (SecurityErrorEvent.SECURITY_ERROR)。若要解决此错误，可以在接收方的 LocalConnection.allowDomain() 方法中指定调用方的域。

如果仅在同一个域中的内容之间实现通信，可以指定一个不以下划线 (_) 开头且不指定域名的 connectionName 参数（例如 myDomain:connectionName）。在 LocalConnection.connect(connectionName) 命令中使用相同的字符串。

如果要实现不同域中的内容之间的通信，可以指定一个以下划线开头的 connectionName 参数。指定下划线使具有接收方 LocalConnection 对象的内容更易于在域之间移植。下面是两种可能的情形：

- 如果 connectionName 字符串不以下划线开头，则运行时会添加一个包含超级域名和冒号的前缀（例如 myDomain:connectionName）。虽然这可以确保您的连接不会与其它域中具有同一名称的连接冲突，但任何发送方 LocalConnection 对象都必须指定此超级域（例如 myDomain:connectionName）。如果将具有接收方 LocalConnection 对象的 HTML 或 SWF 文件移动到另一个域中，则运行时会更改前缀，以反映新的超级域（例如 anotherDomain:connectionName）。必须手动编辑所有发送方 LocalConnection 对象，以指向新超级域。
- 如果 connectionName 字符串以下划线开头（例如 _connectionName），则运行时不会向该字符串添加前缀。这意味着接收方和发送方 LocalConnection 对象都将使用相同的 connectionName 字符串。如果接收方对象使用 LocalConnection.allowDomain() 指定可以接受来自任何域的连接，则可以将具有接收方 LocalConnection 对象的 HTML 或 SWF 文件移动到另一个域，而无需更改任何发送方 LocalConnection 对象。

在 connectionName 中使用下划线名称的缺点是存在潜在冲突，例如当两个应用程序使用同一 connectionName 同时尝试连接时。第二个相关缺点是安全方面的。使用下划线语法的连接名称不会标识侦听应用程序的域。出于这些原因，应优先选择使用域限定名称。

对于在 AIR 应用程序安全沙箱中运行的内容（随 AIR 应用程序安装的内容），AIR 使用字符串 app# 后跟 AIR 应用程序的应用程序 ID（应用程序描述符文件中定义）、点(.) 字符以及该应用程序的发布者 ID，来替代浏览器中运行的 SWF 内容所使用的域。例如，应用程序 ID 为 com.example.air.MyApp 且发布者 ID 为 B146A943FBD637B68C334022D304CEA226D129B4 的应用程序的 connectionName 将解析为 "app#com.example.air.MyApp.B146A943FBD637B68C334022D304CEA226D129B4:connectionName"。（有关详细信息，请参阅第 104 页的“[定义应用程序标识](#)”和第 285 页的“[获取应用程序标识符和发行商标识符](#)”。）

当您允许其它 AIR 应用程序通过本地连接与您的应用程序进行通信时，必须调用 LocalConnection 对象的 allowDomain()，并传入本地连接域名。对于 AIR 应用程序，此域名的形式与连接字符串相同，都包含了应用程序 ID 和发布者 ID。例如，如果发送方 AIR 应用程序的应用程序 ID 为 com.example.air.FriendlyApp 且发布者 ID 为 214649436BD677B62C33D02233043EA236D13934，则用于允许此应用程序进行连接的域字符串为：app#com.example.air.FriendlyApp.214649436BD677B62C33D02233043EA236D13934。

注：如果使用 ADL（或使用 Flash CS3、Flex Builder 或 Dreamweaver 等开发工具）运行应用程序，则发布者 ID 为 null，必须从域字符串中省略该 ID。在安装和运行应用程序时，必须在域字符串中包含发布者 ID。可以使用 ADL 命令行参数来分配临时发布者 ID。使用临时发布者 ID 可测试连接字符串和域名的格式是否正确。

第 36 章：分发、安装和运行 AIR 应用程序

AIR 应用程序作为单个 AIR 安装文件，此安装文件中包含应用程序代码和所有资源。您可以通过通常采用的任何方法来分发此文件，例如，通过下载、通过电子邮件或通过物理媒体（如 CD-ROM）。用户可以通过双击此 AIR 文件来安装此应用程序。您可以使用无缝安装功能，即用户单击网页中的单个链接即可安装 AIR 应用程序（如果需要，还可安装 Adobe® AIR®）。

必须先将 AIR 安装文件打包，并用代码签名证书和私钥进行签名，然后才能分发该文件。对此安装文件进行数字签名可确保应用程序自签名以来未经修改。另外，如果受信任的认证机构颁发了数字证书，则您的用户即可确认您作为发行商和签名者的身份。当使用 AIR 开发工具 (ADT) 将应用程序打包时，会对 AIR 文件进行签名。

有关如何使用 Adobe® Dreamweaver® 将应用程序打包成 AIR 文件的信息，请参阅第 17 页的“[在 Dreamweaver 中创建 AIR 应用程序](#)”。

有关如何使用 Adobe® AIR® SDK 将应用程序打包成 AIR 文件的信息，请参阅第 24 页的“[使用 AIR Developer Tool \(ADT\) 打包 AIR 安装文件](#)”。

从桌面安装和运行 AIR 应用程序

您可以直接将 AIR 文件发送给接收方。例如，您可以将 AIR 文件作为电子邮件附件或作为网页中的链接发送。

用户下载 AIR 应用程序后，就会按以下说明安装该应用程序：

1 双击 AIR 文件。

计算机上必须已安装 Adobe AIR。

2 在“安装”窗口中，保留默认设置处于选定状态不变，然后单击“继续”。

在 Windows 中，AIR 会自动执行以下操作：

- 将此应用程序安装到 Program Files 目录
- 为此应用程序创建一个桌面快捷方式
- 创建“开始”菜单快捷方式
- 在“添加 / 删除程序”控制面板中添加一个应用程序条目

在 Mac OS 中，默认情况下，会将此应用程序添加到 Applications 目录中。

如果已安装此应用程序，安装程序将让用户选择是打开此应用程序的现有版本还是更新到所下载 AIR 文件中的版本。安装程序使用 AIR 文件中的应用程序 ID 和发行商 ID 来标识此应用程序。

3 安装完成后，单击“完成”。

在 Mac OS 中，若要安装某一应用程序的更新版本，用户需要具有足够的系统权限才能将新版本安装到应用程序目录中。在 Windows 和 Linux 中，用户需要具有管理权限。

应用程序也可以通过 ActionScript 或 JavaScript 安装新版本。有关详细信息，请参阅第 318 页的“[更新 AIR 应用程序](#)”。

安装 AIR 应用程序后，用户只需双击此应用程序的图标即可运行它，就像任何其他桌面应用程序一样。

- 在 Windows 中，请双击该应用程序的图标（安装在桌面上或文件夹中），或者从“开始”菜单中选择该应用程序。
- 在 Linux 中，请双击该应用程序的图标（安装在桌面上或文件夹中），或者从应用程序菜单中选择该应用程序。
- 在 Mac OS 中，请在该应用程序的安装文件夹中双击该应用程序。默认安装目录为 /Applications 目录。

使用 AIR 无缝安装 功能，用户可通过单击网页中的链接来安装 AIR 应用程序。使用 AIR 浏览器调用 功能，用户可以通过单击网页中的链接来运行安装的 AIR 应用程序。下一节中介绍了这些功能。

从网页安装和运行 AIR 应用程序

使用无缝安装功能，可以将 SWF 文件嵌入到网页中，这样用户便可以从浏览器中安装 AIR 应用程序。如果未安装运行时，无缝安装功能将安装运行时。使用无缝安装功能，用户无需将 AIR 文件保存到其计算机，即可安装 AIR 应用程序。AIR SDK 中包含一个 badge.swf 文件，通过此文件，您可以轻松使用无缝安装功能。有关详细信息，请参阅第 306 页的“[使用 badge.swf 文件安装 AIR 应用程序](#)”。

有关如何使用无缝安装功能的演示，请参阅[通过 Web 分发 AIR 应用程序](#) (http://www.adobe.com/go/learn_air_qs_seamless_install_cn) 快速入门示例文章。

关于自定义无缝安装 badge.swf

除了使用 SDK 随附的 badge.swf 文件，您还可以创建自己的 SWF 文件以供在浏览器页面中使用。自定义的 SWF 文件可以通过以下方式与运行时进行交互：

- 它可以安装 AIR 应用程序。请参阅第 310 页的“[从浏览器安装 AIR 应用程序](#)”。
- 它可以检查是否已安装特定 AIR 应用程序。请参阅第 310 页的“[从网页检查是否已安装 AIR 应用程序](#)”。
- 它可以检查是否已安装运行时。请参阅第 309 页的“[检查是否已安装运行时](#)”。
- 它可以在用户的系统中启动安装的 AIR 应用程序。请参阅第 311 页的“[从浏览器启动安装的 AIR 应用程序](#)”。

所有这些功能都是通过在承载于 adobe.com 上的 SWF 文件 air.swf 中调用 API 提供的。本节说明如何使用和自定义 badge.swf 文件，以及如何从您自己的 SWF 文件中调用 air.swf API。

另外，在浏览器中运行的 SWF 文件可以通过使用 LocalConnection 类与正在运行的 AIR 应用程序通信。有关详细信息，请参阅第 302 页的“[应用程序间的通信](#)”。

重要说明：本节中所述的功能（以及 air.swf 文件中的 API）要求最终用户在 Windows 或 Mac OS 的 Web 浏览器中安装 Adobe® Flash® Player 9 更新 3。在 Linux 中，无缝安装功能需要 Flash Player 10 (10.0.12.36 版或更高版本)。您可以编写代码来检查已安装的 Flash Player 版本，如果未安装所需的 Flash Player 版本，可以为用户提供替代界面。例如，如果安装的是 Flash Player 的旧版本，您可以提供下载 AIR 文件版本的链接（而不是使用 badge.swf 文件或 air.swf API 来安装应用程序）。

使用 badge.swf 文件安装 AIR 应用程序

AIR SDK 中包含一个 badge.swf 文件，通过此文件，您可以轻松使用无缝安装功能。badge.swf 可以从网页中的链接安装运行时和 AIR 应用程序。为您提供了一个 badge.swf 文件及其源代码以供您在您的网站上分发。

本节中的说明介绍了如何设置 Adobe 提供的 badge.swf 文件的参数。我们还提供了 badge.swf 文件的源代码，您可以对其进行自定义。

在网页中嵌入 badge.swf 文件

1 找到以下文件（在 AIR SDK 的 samples/badge 目录中提供），并将这些文件添加到您的 Web 服务器中。

- badge.swf
- default_badge.html
- AC_RunActiveContent.js

2 在文本编辑器中打开 default_badge.html 页。

3 在 default_badge.html 页中的 AC_FL_RunContent() JavaScript 函数中，针对以下参数调整 FlashVars 参数定义：

参数	说明
appname	应用程序的名称，如果没有安装运行时，则由 SWF 文件显示。
appurl	(必需)。要下载的 AIR 文件的 URL。必须使用绝对（而非相对）URL。
airversion	(必需)。对于运行时 1.0 版，将此参数设置为 1.0。
imageurl	要在标志中显示的图像（可选）的 URL。
buttoncolor	下载按钮的颜色（以十六进制值的形式指定，例如 FFCC00）。
messagecolor	如果没有安装运行时，则为按钮下方显示的文本消息的颜色（以十六进制值的形式指定，如 FFCC00）。

4 badge.swf 文件的最小大小为 217 像素宽 x 180 像素高。调整 AC_FL_RunContent() 函数的 width 和 height 参数的值，以满足您的需要。

5 重命名 default_badge.html 文件并调整其代码（或将其包含在另一个 HTML 页中），以满足您的需要。

您也可以编辑和重新编译 badge.swf 文件。有关详细信息，请参阅第 308 页的“[修改 badge.swf 文件](#)”。

从网页中的无缝安装链接安装 AIR 应用程序

将无缝安装链接添加到页面中后，用户即可通过在 SWF 文件中单击此链接来安装 AIR 应用程序。

1 在已安装 Flash Player（Windows 和 Mac OS 中为版本 9 更新 3 或更高版本，Linux 中为版本 10）的 Web 浏览器中导航到此 HTML 页。

2 在此网页中，单击 badge.swf 文件中的链接。

- 如果您已安装运行时，请跳至下一步。
- 如果您尚未安装运行时，将显示一个对话框，询问您是否要安装它。安装运行时（请参阅第 1 页的“[Adobe AIR 安装](#)”），然后接着执行下一步。

3 在“安装”窗口中，保留默认设置处于选定状态不变，然后单击“继续”。

在 Windows 计算机中，AIR 会自动执行以下操作：

- 将应用程序安装到 c:\Program Files\ 中
- 为此应用程序创建一个桌面快捷方式
- 创建“开始”菜单快捷方式
- 在“添加 / 删除程序”控制面板中添加一个应用程序条目

在 Mac 操作系统中，安装程序会将该应用程序添加到 Applications 目录（例如，添加到 Mac 操作系统中的 /Applications 目录中）。

在 Linux 计算机上，AIR 自动执行以下操作：

- 将应用程序安装到 /opt 中。
- 为此应用程序创建一个桌面快捷方式
- 创建“开始”菜单快捷方式
- 在系统包管理器中加入该应用程序的条目

4 选择所需选项，然后单击“安装”按钮。

5 安装完成后，单击“完成”。

修改 badge.swf 文件

AIR SDK 提供了 badge.swf 文件的源文件。这些文件包含在 SDK 的 samples/badge 文件夹中：

源文件	说明
badge.fla	用于编译 badge.swf 文件的 Flash 源文件。badge.fla 文件编译成 SWF 9 文件（可以在 Flash Player 中加载）。
AIRBadge.as	定义在 badge.fla 文件中使用的基类的 ActionScript 3.0 类。

可以使用 Flash CS3 或 Flash CS4 重新设计 badge.fla 文件的可视界面。

在 AIRBadge 类中定义的 AIRBadge() 构造函数加载 <http://airdownload.adobe.com/air/browserapi/air.swf> 上承载的 air.swf 文件。air.swf 文件包含用于使用无缝安装功能的代码。

成功加载 air.swf 文件后，将调用 AIRBadge 类中的 `onInit()` 方法：

```
private function onInit(e:Event):void {
    _air = e.target.content;
    switch (_air.getStatus()) {
        case "installed" :
            root.statusMessage.text = "";
            break;
        case "available" :
            if (_appName && _appName.length > 0) {
                root.statusMessage.htmlText = "<p align='center'><font color='#" +
                    _messageColor + "'>In order to run " + _appName +
                    ", this installer will also set up Adobe® AIR®.</font></p>";
            } else {
                root.statusMessage.htmlText = "<p align='center'><font color='#" +
                    _messageColor + "'>In order to run this application,
                    " + "this installer will also set up Adobe® AIR®.</font></p>";
            }
            break;
        case "unavailable" :
            root.statusMessage.htmlText = "<p align='center'><font color='#" +
                _messageColor +
                "'>Adobe® AIR® is not available for your system.</font></p>";
            root.buttonBg_mc.enabled = false;
            break;
    }
}
```

上述代码将全局 `_air` 变量设置为所加载的 air.swf 文件的主类。此类包括以下公共方法， badge.swf 文件通过访问这些方法来调用无缝安装功能：

方法	说明
getStatus()	确定计算机上是否已安装（或是否可以安装）运行时。有关详细信息，请参阅第 309 页的“ 检查是否已安装运行时 ”。
installApplication()	在用户的计算机上安装指定的应用程序。有关详细信息，请参阅第 310 页的“ 从浏览器安装 AIR 应用程序 ”。 <ul style="list-style-type: none"> • url - 定义 URL 的字符串。必须使用绝对（而非相对）URL 路径。 • runtimeVersion - 指示要安装的应用程序所需的运行时版本（例如“1.0.M6”）的字符串。 • arguments - 要传递给此应用程序的参数（如果此应用程序在安装后启动）。如果在应用程序描述符文件中将 allowBrowserInvocation 元素设置为 true，则应用程序会在安装后启动。（有关应用程序描述符文件的详细信息，请参阅第 102 页的“设置 AIR 应用程序属性”。）如果因从浏览器进行无缝安装而导致应用程序启动（用户选择在安装后启动），则仅当传递参数时，应用程序的 NativeApplication 对象才调度 BrowserInvokeEvent 对象。请考虑您传递给应用程序的数据存在的安全隐患。有关详细信息，请参阅第 311 页的“从浏览器启动安装的 AIR 应用程序”。

url 和 runtimeVersion 的设置通过容器 HTML 页中的 FlashVars 设置传入 SWF 文件。

如果应用程序在安装后自动启动，您可以使用 LocalConnection 通信让已安装的应用程序在调用时与 badge.swf 文件联系。有关详细信息，请参阅第 302 页的“[应用程序间的通信](#)”。

还可以调用 air.swf 文件的 getApplicationVersion() 方法检查是否已安装应用程序。可以在开始应用程序安装过程之前调用此方法，亦可在安装开始之后进行调用。有关详细信息，请参阅第 310 页的“[从网页检查是否已安装 AIR 应用程序](#)”。

加载 air.swf 文件

您可以创建自己的 SWF 文件，使之使用 air.swf 文件中的 API 从浏览器中的网页与运行时和 AIR 应用程序交互。air.swf 文件承载于 <http://airdownload.adobe.com/air/browserapi/air.swf>。若要从 SWF 文件中引用 air.swf API，请将 air.swf 文件加载到 SWF 文件所在的应用程序域中。下面的代码显示了将 air.swf 文件加载到执行加载的 SWF 文件所在的应用程序域中的示例：

```
var airSWF:Object; // This is the reference to the main class of air.swf
var airSWFLoader:Loader = new Loader(); // Used to load the SWF
var loaderContext:LoaderContext = new LoaderContext();
                                         // Used to set the application domain

loaderContext.applicationDomain = ApplicationDomain.currentDomain;

airSWFLoader.contentLoaderInfo.addEventListener(Event.INIT, onInit);
airSWFLoader.load(new URLRequest("http://airdownload.adobe.com/air/browserapi/air.swf"),
                  loaderContext);

function onInit(e:Event):void
{
    airSWF = e.target.content;
}
```

一旦加载了 air.swf 文件（Loader 对象的 contentLoaderInfo 对象调度 init 事件时），您就可以调用任何 air.swf API。以下几节中介绍了这些 API：

- 第 309 页的“[检查是否已安装运行时](#)”
- 第 310 页的“[从网页检查是否已安装 AIR 应用程序](#)”
- 第 310 页的“[从浏览器安装 AIR 应用程序](#)”
- 第 311 页的“[从浏览器启动安装的 AIR 应用程序](#)”

注：AIR SDK 附带的 badge.swf 文件会自动加载 air.swf 文件。请参阅第 306 页的“[使用 badge.swf 文件安装 AIR 应用程序](#)”。本节中的说明适用于创建您自己的加载 air.swf 文件的 SWF 文件。

检查是否已安装运行时

SWF 文件通过在从 <http://airdownload.adobe.com/air/browserapi/air.swf> 加载的 air.swf 文件中调用 getStatus() 方法，可以检查是否已安装运行时。有关详细信息，请参阅第 309 页的“[加载 air.swf 文件](#)”。

加载 air.swf 文件后，SWF 文件便可以调用 air.swf 文件的 getStatus() 方法，如下所示：

```
var status:String = airSWF.getStatus();
```

getStatus() 方法根据计算机上运行时的状态，返回下列字符串值之一：

字符串值	说明
"available"	运行时可以安装在此计算机上，但当前未安装。
"unavailable"	运行时无法安装在此计算机上。
"installed"	运行时已安装在此计算机上。

如果浏览器中未安装所需的 Flash Player 版本 (Windows 和 Mac OS 中为版本 9 更新 3 或更高版本, Linux 中为版本 10) , 则 `getStatus()` 方法会引发错误。

从网页检查是否已安装 AIR 应用程序

SWF 文件通过在从 <http://airdownload.adobe.com/air/browserapi/air.swf> 加载的 air.swf 文件中调用 `getApplicationVersion()` 方法, 可以检查是否已安装 (应用程序 ID 和发行商 ID 相符的) AIR 应用程序。有关详细信息, 请参阅第 309 页的 “[加载 air.swf 文件](#)”。

加载 air.swf 文件后, SWF 文件便可以调用 air.swf 文件的 `getApplicationVersion()` 方法, 如下所示:

```
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EED35F84C264A183921344EEA353A629FD.1";
airSWF.getApplicationVersion(appID, pubID, versionDetectCallback);

function versionDetectCallback(version:String):void
{
    if (version == null)
    {
        trace("Not installed.");
        // Take appropriate actions. For instance, present the user with
        // an option to install the application.
    }
    else
    {
        trace("Version", version, "installed.");
        // Take appropriate actions. For instance, enable the
        // user interface to launch the application.
    }
}
```

`getApplicationVersion()` 方法具有以下参数:

参数	说明
appID	此应用程序的应用程序 ID。有关详细信息, 请参阅第 104 页的 “ 定义应用程序标识 ”。
pubID	此应用程序的发行商 ID。有关详细信息, 请参阅第 313 页的 “ 关于 AIR 发行商标识别符 ”。
callback	用作处理函数的回调函数。 <code>getApplicationVersion()</code> 方法异步运行, 在检测到已安装的版本 (或没有已安装的版本) 时, 会调用此回调方法。回调方法定义必须包含一个参数, 此参数为一个字符串, 设置为已安装的应用程序的版本字符串。如果未安装此应用程序, 则会将一个 <code>null</code> 值传递给此函数, 如上一代码示例所示。

如果浏览器中未安装所需的 Flash Player 版本 (Windows 和 Mac OS 中为版本 9 更新 3 或更高版本, Linux 中为版本 10) , 则 `getApplicationVersion()` 方法会引发错误。

从浏览器安装 AIR 应用程序

SWF 文件通过在从 <http://airdownload.adobe.com/air/browserapi/air.swf> 加载的 air.swf 文件中调用 `installApplication()` 方法, 可以安装 AIR 应用程序。有关详细信息, 请参阅第 309 页的 “[加载 air.swf 文件](#)”。

加载 air.swf 文件后, SWF 文件便可以调用 air.swf 文件的 `installApplication()` 方法, 如下面的代码所示:

```
var url:String = "http://www.example.com/myApplication.air";
var runtimeVersion:String = "1.0";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.installApplication(url, runtimeVersion, arguments);
```

`installApplication()` 方法在用户的计算机上安装指定的应用程序。此方法具有以下参数:

参数	说明
url	一个字符串，定义要安装的 AIR 文件的 URL。必须使用绝对（而非相对）URL 路径。
runtimeVersion	一个字符串，指示要安装的应用程序所需的运行时版本（例如“1.0”）。
arguments	要传递给此应用程序的参数数组（如果此应用程序在安装后启动）。参数中只能识别字母数字字符。如果需要传递其它值，请考虑使用编码方案。 如果在应用程序描述符文件中将 <code>allowBrowserInvocation</code> 元素设置为 <code>true</code> ，则应用程序会在安装后启动。（有关应用程序描述符文件的详细信息，请参阅第 102 页的“ 设置 AIR 应用程序属性 ”。）如果因从浏览器进行无缝安装而导致应用程序启动（用户选择在安装后启动），则仅当已传递参数时，应用程序的 <code>NativeApplication</code> 对象才调度 <code>BrowserInvokeEvent</code> 对象。有关详细信息，请参阅第 311 页的“ 从浏览器启动安装的 AIR 应用程序 ”。

仅当在用户事件（例如鼠标单击）的事件处理函数中调用 `installApplication()` 方法时，此方法才能执行。

如果浏览器中未安装所需的 Flash Player 版本（Windows 和 Mac OS 中为版本 9 更新 3 或更高版本，Linux 中为版本 10），则 `installApplication()` 方法会引发错误。

在 Mac 操作系统中，若要安装某一应用程序的更新版本，用户必须拥有足够的系统权限才能将新版本安装到应用程序目录中（如果此应用程序更新运行时，则还须拥有管理权限）。在 Windows 中，用户必须具有管理权限。

还可以调用 `air.swf` 文件的 `getApplicationVersion()` 方法检查是否已安装应用程序。可以在开始应用程序安装过程之前调用此方法，亦可在安装开始之后进行调用。有关详细信息，请参阅第 310 页的“[从网页检查是否已安装 AIR 应用程序](#)”。此应用程序运行后便可以通过使用 `LocalConnection` 类与浏览器中的 SWF 内容通信。有关详细信息，请参阅第 302 页的“[应用程序间的通信](#)”。

从浏览器启动安装的 AIR 应用程序

若要使用浏览器调用功能（使其可以从浏览器启动），目标应用程序的应用程序描述符文件必须包含以下设置：

```
<allowBrowserInvocation>true</allowBrowserInvocation>
```

有关应用程序描述符文件的详细信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

浏览器中的 SWF 文件通过在从 `http://airdownload.adobe.com/air/browserapi/air.swf` 加载的 `air.swf` 文件中调用 `launchApplication()` 方法，可以启动 AIR 应用程序。有关详细信息，请参阅第 309 页的“[加载 air.swf 文件](#)”。

加载 `air.swf` 文件后，SWF 文件便可以调用 `air.swf` 文件的 `launchApplication()` 方法，如下面的代码所示：

```
var appID:String = "com.example.air.myTestApplication";
var pubID:String = "02D88EED35F84C264A183921344EEA353A629FD.1";
var arguments:Array = ["launchFromBrowser"]; // Optional
airSWF.launchApplication(appID, pubID, arguments);
```

`launchApplication()` 方法在 `air.swf` 文件（在用户界面 SWF 文件所在的应用程序域中加载）的顶级定义。调用此方法将导致 AIR 启动指定的应用程序（如果该应用程序已安装，且通过应用程序描述符文件中的 `allowBrowserInvocation` 设置允许浏览器调用）。此方法具有以下参数：

参数	说明
appID	要启动的应用程序的应用程序 ID。有关详细信息，请参阅第 104 页的“ 定义应用程序标识 ”。
pubID	要启动的应用程序的发行商 ID。有关详细信息，请参阅第 313 页的“ 关于 AIR 发行商标识符 ”。
arguments	要传递给此应用程序的参数数组。此应用程序的 <code>NativeApplication</code> 对象调度 <code>arguments</code> 属性设置为此数组的 <code>BrowserInvokeEvent</code> 事件。参数中只能识别字母数字字符。如果需要传递其它值，请考虑使用编码方案。

仅当在用户事件（例如鼠标单击）的事件处理函数中调用 `launchApplication()` 方法时，此方法才能执行。

如果浏览器中未安装所需的 Flash Player 版本（Windows 和 Mac OS 中为版本 9 更新 3 或更高版本，Linux 中为版本 10），则 `launchApplication()` 方法会引发错误。

如果在应用程序描述符文件中将 `allowBrowserInvocation` 元素设置为 `false`，则调用 `launchApplication()` 方法将不起任何作用。

在显示用户界面以启动应用程序之前，您可能需要在 `air.swf` 文件中调用 `getApplicationVersion()` 方法。有关详细信息，请参阅第 310 页的“[从网页检查是否已安装 AIR 应用程序](#)”。

当通过浏览器调用功能调用此应用程序时，此应用程序的 `NativeApplication` 对象将调度 `BrowserInvokeEvent` 对象。有关详细信息，请参阅第 282 页的“[浏览器调用](#)”。

如果您使用浏览器调用功能，一定要考虑安全隐患（在第 282 页的“[浏览器调用](#)”中介绍）。

此应用程序运行后便可以通过使用 `LocalConnection` 类与浏览器中的 SWF 内容通信。有关详细信息，请参阅第 302 页的“[应用程序间的通信](#)”。

企业部署

IT 管理员可以使用标准的桌面部署工具以静默方式安装 Adobe AIR 运行时和 AIR 应用程序。IT 管理员可以执行以下操作：

- 使用工具（例如，Microsoft SMS、IBM Tivoli，或任何允许使用引导程序进行静默安装的部署工具）以静默方式安装 Adobe AIR 运行时
- 使用用于部署运行时的相同工具以静默方式安装 AIR 应程序

有关详细信息，请参阅《AIR 管理员指南》(http://www.adobe.com/go/learn_air_admin_guide_cn)。

对 AIR 文件进行数字签名

如果用公认的认证机构 (CA) 颁发的证书对 AIR 安装文件进行数字签名，则可以为您的用户提供他们所安装的应用程序未经无意或恶意修改的重要保证，并能证明您的签名者（发行商）身份。用可信的证书或链至安装计算机上可信证书的证书对 AIR 应用程序进行签名后，AIR 会在安装期间显示发行商名称。否则，发行商名称将显示为“未知”。

重要说明：如果恶意实体以某种方式获取您的签名 `keystore` 文件或发现您的私钥，则可以以您的身份伪造 AIR 文件。

有关代码签名证书的信息

证书实行声明 (CPS) 和由颁发证书的认证机构发布的订户协议中拟定了涉及代码签名证书使用的安全保证、限制和法律义务。有关当前颁发 AIR 代码签名证书的证书颁发机构的协议的详细信息，请参阅：

[ChosenSecurity](http://www.chosensecurity.com/products/tc_publisher_id_adobe_air.htm) (http://www.chosensecurity.com/products/tc_publisher_id_adobe_air.htm)

[ChosenSecurity CPS](http://www.chosensecurity.com/resource_center/repository.htm) (http://www.chosensecurity.com/resource_center/repository.htm)

[GlobalSign](http://www.globalsign.com/developer/code-signing-certificate/index.htm) (<http://www.globalsign.com/developer/code-signing-certificate/index.htm>)

[GlobalSign CPS](http://www.globalsign.com/repository/index.htm) (<http://www.globalsign.com/repository/index.htm>)

[Thawte CPS](http://www.thawte.com/cps/index.html) (<http://www.thawte.com/cps/index.html>)

[Thawte Code Signing Developer's Agreement](http://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/devlcertsign.pdf) (<http://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/devlcertsign.pdf>)

[VeriSign CPS](http://www.verisign.com/repository/CPS/) (<http://www.verisign.com/repository/CPS/>)

[VeriSign Subscriber's Agreement](https://www.verisign.com/repository/subscriber/SUBAGR.html) (<https://www.verisign.com/repository/subscriber/SUBAGR.html>)

关于 AIR 代码签名

对 AIR 文件进行签名后，安装文件中将包含一个数字签名。此签名包括程序包的摘要，用于证实 AIR 文件自签名以来未经修改；此签名还包括有关签名证书的信息，用于证实发行商身份。

AIR 使用通过操作系统的证书存储区支持的公钥基础结构 (PKI) 来确定证书是否可信。安装 AIR 应用程序的计算机必须直接信任用于对此 AIR 应用程序进行签名的证书，或者必须信任将该证书链接到受信认证机构的证书链，才能核实发行商信息。

如果 AIR 文件未链至其中一个受信根证书（通常，这些证书包括所有自签名证书）的证书进行签名，则无法核实发行商信息。虽然 AIR 可以确定 AIR 程序包自签名以来未经修改，但无法知道文件的实际创建者和签名者。

注：用户可以选择信任自签名证书，这样，用该证书签名的任何 AIR 应用程序就会显示该证书中的公共名称字段的值作为发行商名称。AIR 不为用户提供任何将证书指定为可信证书的方法。必须单独为用户提供证书（不包括私钥），且用户必须使用操作系统提供的某种机制或适当的工具将证书导入系统证书存储区中的正确位置。

关于 AIR 发行商标识符

在安装 AIR 文件的过程中，AIR 应用程序安装程序会生成一个发行商 ID。这是用于生成 AIR 文件的证书的唯一标识符。如果对多个 AIR 应用程序重复使用同一个证书，它们将具有相同的发行商 ID。发行商 ID 和应用程序 ID 合起来唯一地标识一个 AIR 应用程序。

应用程序需要了解另一个 AIR 应用程序的发行商 ID，才能使用 LocalConnection 类与其通信（请参阅第 302 页的“[应用程序间的通信](#)”）。您可以通过读取 NativeApplication.nativeApplication.publisherID 属性来确定已安装的应用程序的发行商 ID。

下列字段用于计算发行商 ID：Name、CommonName、Surname、GivenName、Initials、GenerationQualifier、DNQualifier、CountryName、localityName、StateOrProvinceName、OrganizationName、OrganizationalUnitName、Title、Email、SerialNumber、DomainComponent、Pseudonym、BusinessCategory、StreetAddress、PostalCode、PostalAddress、DateOfBirth、PlaceOfBirth、Gender、CountryOfCitizenship、CountryOfResidence 和 NameAtBirth。如果您续订认证机构颁发的证书，或者重新生成自签名证书，则这些字段必须保持不变才能使发行商 ID 保持不变。另外，CA 颁发的证书的根证书以及自签名证书的公钥也必须保持不变。

关于证书格式

AIR 签名工具接受任何可通过 Java 加密体系统结构 (JCA) 访问的 Keystore。这包括基于文件的 Keystore（例如 PKCS12 格式的文件，通常使用 .pfx 或 .p12 文件扩展名）、Java .keystore 文件、PKCS11 硬件 Keystore 和系统 Keystore。ADT 可以访问的 Keystore 格式取决于用于运行 ADT 的 Java 运行时的版本和配置。访问某些类型的 Keystore（例如 PKCS11 硬件令牌）可能需要安装和配置附加的软件驱动程序和 JCA 插件。

若要对 AIR 文件进行签名，可以使用大部分现有的代码签名证书，也可以获取一个专门为对 AIR 应用程序进行签名而颁发的新证书。例如，可以使用 VeriSign、Thawte、GlobalSign 或 ChosenSecurity 所颁发的以下任何类型的证书：

- [ChosenSecurity](#)
 - 用于 Adobe AIR 的 TC Publisher ID
- [GlobalSign](#)
 - ObjectSign 代码签名证书
- [Thawte](#):
 - AIR 开发人员证书 (AIR Developer Certificate)
 - Apple 开发人员证书 (Apple Developer Certificate)
 - JavaSoft 开发人员证书 (JavaSoft Developer Certificate)
 - Microsoft 验证码证书 (Microsoft Authenticode Certificate)

- [VeriSign](#):
 - Adobe AIR 数字 ID
 - Microsoft 验证码数字 ID (Microsoft Authenticode Digital ID)
 - Sun Java 签名数字 ID (Sun Java Signing Digital ID)

注: 必须创建证书才能进行代码签名。不能使用 SSL 或其它证书类型对 AIR 文件进行签名。

时间戳

对 AIR 文件进行签名时, 打包工具会查询时间戳机构的服务器, 以获取可独立验证的签名日期和时间。获取的时间戳嵌入在 AIR 文件中。只要签名时签名证书有效, 即使在证书过期后也可以安装 AIR 文件。另一方面, 如果未获取时间戳, 则在证书过期或被吊销之后, AIR 文件将变得不可安装。

默认情况下, AIR 打包工具会获取时间戳。然而, 若要允许在时间戳服务不可用时打包应用程序, 您可以禁用时间戳。Adobe 建议使所有公开分发的 AIR 文件都包含一个时间戳。

AIR 打包工具所采用的默认时间戳机构是 [Geotrust](#)。

获取证书

若要获取证书, 您通常需要访问认证机构的网站, 完成该公司的购买流程。使用何种工具生成 AIR 工具所需的 Keystore 文件, 取决于所购买的证书的类型、证书在接收计算机上的存储方式, 在某些情况下, 还取决于用于获取证书的浏览器。例如, 若要从 Thawte 获得和导出 Adobe Developer 证书, 必须使用 Mozilla Firefox。然后可以直接从 Firefox 用户界面中以 .p12 或 .pfx 文件的形式导出证书。

可以使用用于打包 Air 安装文件的 Air 开发工具 (ADT) 生成自签名证书。也可以使用某些第三方工具。

有关如何生成自签名证书的说明以及有关对 AIR 文件进行签名的说明, 请参阅第 24 页的 “[使用 AIR Developer Tool \(ADT\) 打包 AIR 安装文件](#)”。也可以使用 Flex Builder、Dreamweaver 和为 Flash 提供的 AIR 更新来导出 AIR 文件以及对其进行签名。

下面的示例说明如何从 Thawte 认证机构获取 AIR 开发人员证书并准备好将它与 ADT 搭配使用。

示例: 从 Thawte 获取 AIR 开发人员证书

注: 有众多方法可用来获取和准备代码签名证书以供使用, 此示例仅说明了其中的一种。每个证书颁发机构都有其自己的策略和程序。

若要购买 AIR 开发人员证书, Thawte 网站要求您使用 Mozilla Firefox 浏览器。此证书的私钥存储在浏览器的 Keystore 内。请确保 Firefox Keystore 受主密码保护并且计算机本身在物理上是安全的。(完成购买流程后, 您就可以从浏览器 Keystore 中导出和删除证书及私钥。)

在证书注册过程中, 将生成一个私钥 / 公钥对。私钥自动存储在 Firefox Keystore 内。从 Thawte 的网站请求和取回证书时, 必须使用相同的计算机和浏览器。

- 1 访问 Thawte 网站, 并浏览至 [Product page for Code Signing Certificates](#)。
- 2 从“Code Signing Certificates”列表中, 选择“Adobe AIR Developer Certificate”。
- 3 完成三步注册过程。您需要提供您所在单位的信息和联系信息。Thawte 随后将执行其身份验证过程, 并且可能要求提供其他信息。验证完成后, Thawte 将向您发送电子邮件, 邮件中包含有关如何取回此证书的说明。

注: 可以在此处找到有关所需文档类型的其他信息: https://www.thawte.com/ssl-digital-certificates/free-guides-whitepapers/pdf/enroll_codesign_eng.pdf。

- 4 从 Thawte 网站取回颁发的证书。证书会自动保存到 Firefox Keystore。
- 5 按照以下步骤从 Firefox Keystore 导出包含私钥和证书的 Keystore 文件:

注：从 Firefox 导出私钥 / 证书时，它将以 ADT、Flex、Flash 和 Dreamweaver 可以使用的 .p12 (pfx) 格式导出。

- a 打开 Firefox 的“证书管理器”(Certificate Manager)对话框：
 - b 在 Windows 中：打开“工具”(Tools) ->“选项”(Options) ->“高级”(Advanced) ->“加密”(Encryption) ->“查看证书”(View Certificates)
 - c 在 Mac OS 中：打开“Firefox”->“首选参数”(Preferences) ->“高级”(Advanced) ->“加密”(Encryption) ->“查看证书”(View Certificates)
 - d 在 Linux 中：打开“编辑”(Edit) ->“首选参数”(Preferences) ->“高级”(Advanced) ->“加密”(Encryption) ->“查看证书”(View Certificates)
 - e 从证书列表中选择“Adobe AIR 代码签名证书”(Adobe AIR Code Signing Certificate)，然后单击“备份”(Backup)按钮。
 - f 输入文件名和 Keystore 文件的导出位置，然后单击“保存”(Save)。
 - g 如果您使用的是 Firefox 主密码，系统将提示您输入软件安全设备的密码才能导出文件。(此密码仅由 Firefox 使用。)
 - h 在“选择证书备份密码”(Choose a Certificate Backup Password)对话框中，为 Keystore 文件创建一个密码。
重要说明：此密码用于保护 Keystore 文件，当使用该文件对 AIR 应用程序进行签名时需要提供此密码。应该选择一个安全密码。
 - i 单击“确定”。您应该会收到一条关于备份密码设置成功的消息。包含私钥和证书的 Keystore 文件以 .p12 文件扩展名保存（采用 PKCS12 格式）。
- 6 通过 ADT、Flex Builder、Flash 或 Dreamweaver 使用导出的 Keystore 文件。只要对 AIR 应用程序进行签名，就需要提供为该文件创建的密码。

重要说明：私钥和证书仍存储在 Firefox Keystore 内。虽然这样使您可以导出证书文件的其他副本，但它同时也提供了另一个访问点，必须对此访问点加以保护才能维护证书和私钥的安全。

更改证书

在某些情况下，您可能需要更改用于对 AIR 应用程序进行签名的证书。此类情况包括：

- 从自签名证书升级到认证机构颁发的证书
- 从即将到期的自签名证书更改为另一个自签名证书
- 从一个商用证书更改为另一个商用证书，例如，当您的企业标识发生变化时

由于签名证书是确定 AIR 应用程序标识的元素之一，因此并非只改用其他证书就可以对应用程序更新进行签名。若要使 AIR 将 AIR 文件识别为更新，必须使用同一证书对原始 AIR 文件及任何更新的 AIR 文件都进行签名。否则，AIR 会将新的 AIR 文件作为独立的应用程序安装，而不是更新现有的安装。

自 AIR 1.1 起，可以使用迁移签名更改应用程序的签名证书。迁移签名是一种应用于 AIR 更新文件的辅助签名。迁移签名使用原始证书，此证书可证明签名者是应用程序的原始发行商。

重要说明：证书的更改必须在原始证书过期或被吊销前进行。如果不在证书过期前创建使用迁移签名进行签名的更新，则用户在安装任何更新前，必须卸载其应用程序的现有版本。商业上颁发的证书通常都是可续订的，以免过期。自签名证书不可续订。

更改证书：

- 1 创建应用程序更新
- 2 将 AIR 更新文件打包并使用新证书对它进行签名
- 3 通过 ADT -migrate 命令，再次用原始证书对 AIR 文件签名

在第 31 页的“[对 AIR 文件签名以更改应用程序证书](#)”中介绍了迁移签名的应用过程。

安装经过更新的 AIR 文件后，应用程序的标识就会发生变化。这种标识变化会产生以下影响：

- 应用程序的发行商 ID 会发生变化以便与新证书匹配。
- 新应用程序版本无法访问现有的加密本地存储区中的数据。
- 应用程序存储目录的位置会发生变化。旧位置中的数据不会复制到新目录。（但新应用程序可以根据旧发行商 ID 找到原始目录）。
- 应用程序无法再使用旧发行商 ID 打开本地连接。
- 如果用户重新安装迁移前的 AIR 文件，AIR 会使用原始发行商 ID 将其作为独立应用程序安装。

由应用程序负责在应用程序的原始版本与新版本之间迁移所有数据。若要迁移加密的本地存储区 (ELS) 中的数据，必须在证书发生变化前导出这些数据。应用程序的新版本无法读取旧版本的 ELS。（直接重新创建数据通常要比迁移数据更容易。）

应继续对尽可能多的后续更新应用迁移签名。否则，尚未从原始版本升级的用户必须先安装一个中间迁移版本或卸载其当前版本，然后才能安装最新更新。当然，最终原始证书将过期，您将无法再应用迁移签名。（不过，除非您禁用时间戳，否则以前用迁移签名进行签名的 AIR 文件将仍然有效。为迁移签名添加了时间戳，以便在证书过期后也允许 AIR 接受签名。）

具有迁移签名的 AIR 文件在其他方面与普通 AIR 文件无异。如果应用程序安装在没有原始版本的系统中，则 AIR 会按照平常的安装方式安装新版本。

注：续订以商业形式颁发的证书时，通常无需迁移证书。除非识别名称发生更改，否则续订后的证书将保留与原始证书相同的发行商身份。有关用于确定识别名称的证书属性的完整列表，请参阅第 313 页的“[关于 AIR 发行商标识符](#)”。

术语

本节提供了一个术语表，阐释在决定如何对要公开发布的应用程序进行签名时应了解的部分关键术语。

术语	说明
认证机构 (CA)	公钥基础结构网络中的一个实体，担当受信的第三方，并最终对公钥所有者的身份进行证实。CA 通常会颁发由它自己的私钥进行签名的数字证书，以证明它已经核实了证书持有者的身份。
证书实行声明 (CPS)	规定认证机构在颁发和核实证书方面的做法和政策。CPS 是 CA 及其订户与信任方达成的合约的一部分。它还拟定了身份核实方面的政策及它们所提供的证书具备的保证程度。
证书吊销列表 (CRL)	已被吊销而不应再受到信任的已颁发证书列表。AIR 在对 AIR 应用程序进行签名时检查 CRL，如果不存在时间戳，它会在安装该应用程序时再次进行检查。
证书链	证书链是一个证书序列，链中的每个证书已由下一个证书进行签名。
数字证书	一种数字文档，包含所有者的身份、所有者的公钥以及证书本身的标识的有关信息。由认证机构颁发的证书本身由属于颁发证书的 CA 的证书进行签名。
数字签名	经过加密的消息或摘要，只能用公钥 - 私钥对的公钥部分解密。在 PKI 中，数字签名包含一个或多个最终来源于认证机构的数字证书。数字签名可用来证实消息（或计算机文件）自签名以来未经修改（在所用的加密算法提供的保证限制范围内）；此外，假如使用者信任颁发证书的认证机构，也可以用数字签名来证实签名者的身份。
Keystore	包含数字证书、在某些情况下也包含相关私钥的数据库。
Java 加密体系结构 (JCA)	一种用于管理和访问 Keystore 的可扩展体系结构。有关详细信息，请参阅 Java Cryptography Architecture Reference Guide 。
PKCS #11	由 RSA Laboratories 提出的加密令牌接口标准。也是一种基于硬件令牌的 Keystore。
PKCS #12	由 RSA Laboratories 提出的个人信息交换语法标准。也是一种基于文件的 Keystore，通常包含私钥及其关联的数字证书。
私钥	由两部分组成的公钥 - 私钥非对称加密体系的私有部分。私钥必须保密，绝不应该通过网络传送。进行数字签名的消息由签名者通过私钥进行加密。

术语	说明
公钥	由两部分组成的公钥 - 私钥非对称加密体系的公开部分。公钥是公开提供的，用于解密用私钥加密的消息。
公钥基础结构 (PKI)	认证机构用来证明公钥所有者身份的一种信任体系。网络客户端依靠受信的 CA 颁发的数字证书来核实数字消息（或文件）签名者的身份。
时间戳	包含事件发生日期和时间的经过数字签名的数据。ADT 可以将符合 RFC 3161 的时间服务器中的时间戳包含在 AIR 包中。如果存在时间戳，AIR 便在签名时使用时间戳确定证书的有效性。这样，AIR 应用程序便可在其签名证书过期后安装。
时间戳机构	颁发时间戳的机构。为了使 AIR 能够识别，时间戳必须符合 RFC 3161 ，并且时间戳签名必须链至安装计算机上的可信根证书。

第 37 章：更新 AIR 应用程序

用户可以通过双击其计算机上的 AIR 文件或从浏览器中（使用无缝安装功能）安装或更新 AIR 应用程序。Adobe® AIR® 安装应用程序将管理此安装，在用户更新现已存在的应用程序时将向其发出警告。（请参阅第 305 页的“[分发、安装和运行 AIR 应用程序](#)”。）

不过，也可以使用 `Updater` 类让安装的应用程序自行更新到新版本。（安装的应用程序可能检测到有新版本可供下载和安装。）`Updater` 类包括 `update()` 方法，通过此方法可以指向用户计算机上的 AIR 文件，并将其更新为该版本。

更新 AIR 文件的应用程序 ID 和发布者 ID 必须与要更新的应用程序匹配。发布者 ID 是从签名证书中派生的，这意味着更新和被更新应用程序必须使用同一证书进行签名。

从 AIR 1.1 及更高版本起，您可以对应用程序进行迁移以使用新的代码签名证书。对应用程序进行迁移以使用新的签名涉及使用新的和原始的证书对更新 AIR 文件进行签名。证书迁移是一个单向过程。迁移后，只有使用新证书（或同时使用新的和原始证书）进行签名的 AIR 文件才会被识别为对现有安装的更新。

管理应用程序的更新可能非常复杂。AIR 1.5 包括新的 AdobeAIR 应用程序更新框架。此框架提供的 API 可帮助开发人员在 AIR 应用程序中提供良好的更新功能。

可以使用证书迁移将自签名证书更改为商业代码签名证书，或将一个自签名证书或商业证书更改为另一个自签名证书或商业证书。如果未进行证书迁移，则现有用户必须先删除当前的应用程序版本才能安装新版本。有关详细信息，请参阅第 315 页的“[更改证书](#)”。

关于更新应用程序

`Updater` 类（在 `flash.desktop` 包中）包括 `update()` 这一方法，可以使用此方法以其它版本更新当前正在运行的应用程序。例如，如果用户桌面上有某个版本的 AIR 文件（“`Sample_App_v2.air`”），则使用以下代码可以更新该应用程序：

```
var updater = new air.Updater();
var airFile = air.File.desktopDirectory.resolvePath("Sample_App_v2.air");
var version = "2.01";
updater.update(airFile, version);
```

在应用程序使用 `Updater` 类之前，用户或应用程序必须将更新版本的 AIR 文件下载到计算机。有关详细信息，请参阅第 320 页的“[将 AIR 文件下载到用户的计算机](#)”。

调用 `Updater.update()` 方法的结果

当运行时中的应用程序调用 `update()` 方法时，运行时将关闭该应用程序，然后尝试从 AIR 文件安装新版本。运行时将检查在 AIR 文件中指定的应用程序 ID 和发布者 ID 是否与调用 `update()` 方法的应用程序的应用程序 ID 和发布者 ID 相匹配。（有关应用程序 ID 和发布者 ID 的信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。）运行时还将检查版本字符串是否与传递给 `update()` 方法的 `version` 字符串相匹配。如果安装成功完成，运行时将打开新版本的应用程序。否则（如果安装无法完成），它将重新打开应用程序的现有（预安装）版本。

在 Mac OS 中，若要安装某一应用程序的更新版本，用户必须具有足够的系统权限才能将新版本安装到应用程序目录中。在 Windows 和 Linux 中，用户必须具有管理权限。

如果应用程序的更新版本要求运行时的更新版本，则会安装新的运行时版本。若要更新运行时，用户必须具有计算机的管理权限。

在使用 ADL 测试应用程序时，调用 `update()` 方法会导致运行时异常。

关于版本字符串

指定作为 update() 方法的 version 参数的字符串必须与要安装的 AIR 文件的应用程序描述符文件的主 application 元素的 version 属性中的字符串相匹配。出于安全方面的考虑，需要指定 version 参数。通过要求应用程序验证 AIR 文件中的版本号，应用程序不会在不经意间安装较旧的版本，较旧的版本可能包含在当前安装的应用程序中已经得到修复的安全漏洞。应用程序还应检查 AIR 文件中的版本字符串与安装的应用程序中的版本字符串是否相符，以防止降级攻击。

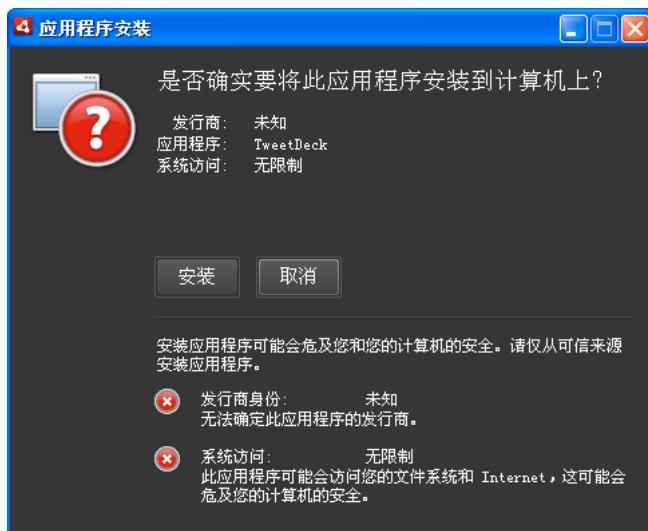
版本字符串可以为任意格式。例如，可以是“2.01”也可以是“version 2”。此字符串的格式完全由应用程序开发人员决定。运行时并不验证版本字符串；应用程序代码应在更新应用程序之前对此字符串进行验证。

如果 Adobe AIR 应用程序通过 Web 下载 AIR 文件，则运用一种机制使 Web 服务能够通知 AIR 应用程序正在下载的版本是一种很好的做法。然后，应用程序可以使用此字符串作为 update() 方法的 version 参数。如果通过其它方式获取 AIR 文件，且 AIR 文件的版本未知，则 AIR 应用程序可以检查 AIR 文件来确定版本信息。（AIR 文件是使用 ZIP 压缩的归档文件，应用程序描述符文件是该归档文件中的第二个记录。）

有关应用程序描述符文件的详细信息，请参阅第 102 页的“[设置 AIR 应用程序属性](#)”。

提供自定义应用程序更新用户界面

AIR 包括一个默认的更新界面：



此界面总是在用户首次在计算机上安装某个版本的应用程序时使用。不过，您可以定义自己的界面以供后续实例使用。如果应用程序定义自定义更新界面，则请在应用程序描述符文件中为当前安装的应用程序指定一个 customUpdateUI 元素：

```
<customUpdateUI>true</customUpdateUI>
```

当应用程序已经安装，且用户打开应用程序 ID 和发布者 ID 与安装的应用程序相匹配的 AIR 文件时，运行时将打开此应用程序，而不是打开默认的 AIR 应用程序安装程序。有关详细信息，请参阅第 108 页的“[提供针对应用程序更新的自定义用户界面](#)”。

应用程序可以决定其何时运行（当 NativeApplication.nativeApplication 对象调度 load 事件时），是否更新应用程序（使用 Updater 类）。如果决定进行更新，则它会向用户提供自己的安装界面（不同于标准的运行界面）。

将 AIR 文件下载到用户的计算机

为了使用 Updater 类，用户或应用程序必须先在本地将 AIR 文件保存到用户的计算机。

注：AIR 1.5 包括一个更新框架，该框架可帮助开发人员在 AIR 应用程序中提供良好的更新功能。使用此框架比直接使用 Updater 类的 update() 方法简单得多。有关详细信息，请参阅第 321 页的“[使用更新框架](#)”。

以下代码从一个 URL (http://example.com/air/updates/Sample_App_v2.air) 读取 AIR 文件，然后将该 AIR 文件保存到应用程序存储目录：

```
var urlString = "http://example.com/air/updates/Sample_App_v2.air";
var urlReq = new air.URLRequest(urlString);
var urlStream = new air.URLStream();
var fileData = new air.ByteArray();
urlStream.addEventListener(air.Event.COMPLETE, loaded);
urlStream.load(urlReq);

function loaded(event) {
    urlStream.readBytes(fileData, 0, urlStream.bytesAvailable);
    writeAirFile();
}

function writeAirFile() {
    var file = air.File.desktopDirectory.resolvePath("My App v2.air");
    var fileStream = new air.FileStream();
    fileStream.open(file, air FileMode.WRITE);
    fileStream.writeBytes(fileData, 0, fileData.length);
    fileStream.close();
    trace("The AIR file is written.");
}
```

有关详细信息，请参阅第 175 页的“[读取和写入文件的工作流程](#)”。

检查应用程序是否为首次运行

更新应用程序后，可能需要向用户提供“快速入门”或“欢迎”消息。在启动时，应用程序将检查是否为首次运行，以便可以确定是否显示此类消息。

注：AIR 1.5 包括一个更新框架，该框架可帮助开发人员在 AIR 应用程序中提供良好的更新功能。此框架提供简单的方法，以检查应用程序的版本是否为首次运行。有关详细信息，请参阅第 321 页的“[使用更新框架](#)”。

实现此操作的方法之一是在初始化应用程序时，在应用程序存储目录中保存一个文件。应用程序每次启动时，都会检查该文件是否存在。如果此文件不存在，则表示当前用户首次运行该应用程序。如果此文件已存在，则表示该应用程序至少已运行过一次。如果此文件已存在且包含比当前版本号旧的版本号，则可知该用户是首次运行此新版本。

下面的示例对该概念进行了演示：

```

<html>
    <head>
        <script src="AIRAliases.js" />
    </script>
        var file;
        var currentVersion = "1.2";
        function system extension() {
            file = air.File.appStorageDirectory.resolvePath("Preferences/version.txt");
            air.trace(file.nativePath);
            if(file.exists) {
                checkVersion();
            } else {
                firstRun();
            }
        }
        function checkVersion() {
            var stream = new air.FileStream();
            stream.open(file, air FileMode.READ);
            var reversion = stream.readUTFBytes(stream.bytesAvailable);
            stream.close();
            if (reversion != currentVersion) {
                window.document.getElementById("log").innerHTML
                    = "You have updated to version " + currentVersion + ".\n";
            } else {
                saveFile();
            }
            window.document.getElementById("log").innerHTML
                += "Welcome to the application.";
        }
        function firstRun() {
            window.document.getElementById("log").innerHTML
                = "Thank you for installing the application. \n"
                + "This is the first time you have run it.";
            saveFile();
        }
        function saveFile() {
            var stream = new air.FileStream();
            stream.open(file, air FileMode.WRITE);
            stream.writeUTFBytes(currentVersion);
            stream.close();
        }
    </script>
</head>
<body onLoad="system extension()">
    <textarea ID="log" rows="100%" cols="100%" />
</body>
</html>

```

如果应用程序将数据保存到本地（例如，保存到应用程序存储目录中），则您可能希望在首次运行时检查以前保存的所有数据（来自以前的版本）。

使用更新框架

管理应用程序的更新可能非常复杂。Adobe AIR 应用程序更新框架提供的 API 可帮助开发人员在 AIR 应用程序中提供良好的更新功能。AIR 更新框架中的功能帮助开发人员完成以下工作：

- 根据某一时间间隔或用户请求定期检查更新
- 从 Web 源下载 AIR 文件（更新）

- 在首次运行新安装的版本时向用户发出警告
- 确认用户希望检查更新
- 向用户显示有关新的更新版本的信息
- 向用户显示下载进度和错误信息

AIR 更新框架提供了一个可供应用程序使用的示例用户界面。该示例用户界面为用户提供了与应用程序更新相关的基本信息和选项。应用程序也可以定义其自己的自定义用户界面，与更新框架配合使用。

AIR 更新框架允许您将有关 AIR 应用程序更新版本的信息存储在简单 XML 配置文件中。对于大多数应用程序，设置这些配置文件以及包括一些基本代码可以为最终用户提供良好的更新功能。

即使不使用更新框架，AIR 应用程序仍可以使用 Adobe AIR 包括的 Updater 类升级到新的版本。通过使用此类，应用程序可以升级到用户计算机上的 AIR 文件中包含的版本。然而，升级管理可不只是进行基于本地存储的 AIR 文件的应用程序更新。

AIR 更新框架中的文件

AIR 更新框架包括以下目录：

- doc — AIR 更新框架的文档（您现在正在阅读的文档）。
- frameworks — 此目录包括 SWC 文件（用于 Flex 开发）和 SWF 文件（用于 HTML 开发）。有关详细信息，请参阅第 322 页的“[在基于 HTML 的 AIR 应用程序中包括框架文件](#)”。
- samples — 此目录包括基于 Flex 和基于 HTML 的示例，这些示例说明了如何使用应用程序更新框架。可以像对待任何 AIR 应用程序一样编译并测试这些文件。
- templates — 此目录包括示例更新描述符文件（简单和已经过本地化）以及配置文件。（有关这些文件的详细信息，请参阅“[设置 Flex 开发环境](#)”和第 322 页的“[基本示例：使用 ApplicationUpdaterUI 版本](#)”。）

在基于 HTML 的 AIR 应用程序中包括框架文件

更新框架的 frameworks/html 目录包括以下这些文件：

- ApplicationUpdater.swf — 定义更新库的基本功能，不包括任何用户界面
- ApplicationUpdater_UI.swf — 定义更新库的基本功能，包括应用程序可以用于显示更新选项的用户界面

AIR 应用程序中的 JavaScript 代码可以使用在 SWF 文件中定义的类。

若要使用更新框架，请在应用程序目录（或子目录）中加入 ApplicationUpdater.swf 或 ApplicationUpdater_UI.swf 文件。然后，在要使用该框架的 HTML 文件中（在 JavaScript 代码中）包括加载该文件的 script 标记：

```
<script src="applicationUpdater.swf" type="application/x-shockwave-flash"/>
```

或者，使用此 script 标记加载 ApplicationUpdater_UI.swf 文件：

```
<script src="ApplicationUpdater_UI.swf" type="application/x-shockwave-flash"/>
```

这两个文件中定义的 API 将在本文档的其余部分进行描述。

基本示例：使用 ApplicationUpdaterUI 版本

更新框架的 ApplicationUpdaterUI 版本提供了可以在应用程序中轻松使用的基本界面。下面是一个基本示例。

首先，创建调用更新框架的 AIR 应用程序：

- 1 如果您的应用程序为基于 HTML 的 AIR 应用程序，则会加载 ApplicationUpdaterUI.js 文件：

```
<script src="ApplicationUpdater_UI.swf" type="application/x-shockwave-flash"/>
```

- 2 在 AIR 应用程序逻辑中，实例化 ApplicationUpdaterUI 对象。

在 ActionScript 中，使用以下代码：

```
var appUpdater:ApplicationUpdaterUI = new ApplicationUpdaterUI();
```

在 JavaScript 中，使用以下代码：

```
var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
```

您可能希望在加载应用程序后执行的初始化函数中添加此代码。

- 3 创建名为 updateConfig.xml 的文本文件并在其中添加以下内容：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
    <url>http://example.com/updates/update.xml</url>
    <delay>1</delay>
</configuration>
```

编辑 updateConfig.xml 文件的 URL 元素以与 Web 服务器上的更新描述符文件的最终位置相一致（请参阅下一过程）。

delay 是应用程序在检查更新之间等待的天数。

- 4 将 updateConfig.xml 文件添加到 AIR 应用程序的项目目录中。

- 5 让 updater 对象引用 updateConfig.xml 文件，并调用该对象的 initialize() 方法。

在 ActionScript 中，使用以下代码：

```
appUpdater.configurationFile = new File("app:/updateConfig.xml");
appUpdater.initialize();
```

在 JavaScript 中，使用以下代码：

```
appUpdater.configurationFile = new air.File("app:/updateConfig.xml");
appUpdater.initialize();
```

- 6 创建另一版本的 AIR 应用程序，该版本与第一个应用程序的版本不同。（该版本将在应用程序描述符文件的 version 元素中指定。）

随后，将更新版本的 AIR 应用程序添加到 Web 服务器：

- 1 将更新版本的 AIR 文件置于 Web 服务器上。

- 2 创建名为 updateDescriptor.xml 的文本文件并在其中添加以下内容：

```
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
    <version>1.1</version>
    <url>http://example.com/updates/sample_1.1.air</url>
    <description>This is the latest version of the Sample application.</description>
</update>
```

编辑 updateDescriptor.xml 文件中的 version、URL 和 description 使其与更新 AIR 文件相匹配。

- 3 将 updateDescriptor.xml 文件添加到包含更新 AIR 文件的同一 Web 服务器目录中。

这是一个基本示例，但它提供的更新功能可以满足许多应用程序的需要。本文档的其余部分介绍了如何使用更新框架以最好地满足您的需要。

有关使用更新框架的另一个示例，请参阅 Adobe AIR 开发人员中心的以下示例应用程序：[基于 HTML 的应用程序中的更新框架](http://www.adobe.com/go/learn_air_qs_update_framework_html_cn) (http://www.adobe.com/go/learn_air_qs_update_framework_html_cn)。

定义更新描述符文件并将 AIR 文件添加到 Web 服务器

在使用 AIR 更新框架时，您可以在存储于 Web 服务器的更新描述符文件中定义关于可用更新的基本信息。更新描述符文件是简单 XML 文件。包括在应用程序中的更新框架可对此文件进行检查，以查看是否已上载新的版本。

更新描述符文件包含以下数据：

- **version** — AIR 应用程序的新版本。该数据必须与新 AIR 应用程序描述符文件中使用的版本字符串相同。如果更新描述符文件中的版本与更新 AIR 文件的版本不一致，则更新框架将引发异常。
- **url** — 更新 AIR 文件的位置。此文件包含更新版本的 AIR 应用程序。
- **description** — 有关新版本的详细信息。在更新过程中可以向用户显示此信息。

version 和 **url** 元素是必需的。**description** 元素为可选元素。

下面是一个示例更新描述符文件：

```
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
    <version>1.1a1</version>
    <url>http://example.com/updates/sample_1.1a1.air</url>
    <description>This is the latest version of the Sample application.</description>
</update>
```

如果您希望使用多种语言定义 **description** 标记，请使用定义 **lang** 属性的多个 **text** 元素：

```
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
    <version>1.1a1</version>
    <url>http://example.com/updates/sample_1.1a1.air</url>
    <description>
        <text xml:lang="en">English description</text>
        <text xml:lang="fr">French description</text>
        <text xml:lang="ro">Romanian description</text>
    </description>
</update>
```

将更新描述符文件随更新 AIR 文件一起放置在 Web 服务器上。

更新描述符随附的模板目录包括示例更新描述符文件。这些文件包括单语言和多语言版本。

实例化 **updater** 对象

在代码中加载 AIR 更新框架（请参阅第 322 页的“[在基于 HTML 的 AIR 应用程序中包括框架文件](#)”）后，需要将 **updater** 对象实例化，如下例所示：

```
var appUpdater = new runtime.air.update.ApplicationUpdater();
```

以上代码使用了 **ApplicationUpdater** 类（没有提供任何用户界面）。如果要使用 **ApplicationUpdaterUI** 类（它提供用户界面），请使用以下语句：

```
var appUpdater = new runtime.air.update.ApplicationUpdaterUI();
```

本文档中的其余代码示例假定您已实例化名为 **appUpdater** 的 **updater** 对象。

配置更新设置

ApplicationUpdater 和 **ApplicationUpdaterUI** 可以通过应用程序随附的配置文件，或者通过应用程序中的 ActionScript 或 JavaScript 进行配置。

在 XML 配置文件中定义更新设置

更新配置文件为 XML 文件。它可能包含以下元素：

- **updateURL** - 一个字符串。表示更新描述符在远程服务器上的位置。允许使用任何有效的 URLRequest 位置。您必须定义 **updateURL** 属性，可以通过配置文件也可以通过脚本进行定义（请参阅第 323 页的“[定义更新描述符文件并将 AIR 文件添](#)

加到 Web 服务器”。必须先定义此属性后，然后才能使用 `updater`（在调用 `updater` 对象的 `initialize()` 方法之前，在第 327 页的“[初始化更新框架](#)”中进行了说明）。

- `delay` — 数字。表示以天为单位给定用于检查更新的时间间隔（允许使用 0.25 等数值）。值 0（这是默认值）指定 `updater` 不执行自动定期检查。

`ApplicationUpdaterUI` 的配置文件除了包含 `updateURL` 和 `delay` 元素之外，还可包含以下元素：

- `defaultUI`: `dialog` 元素的列表。每个 `dialog` 元素都具有与用户界面中的对话框对应的 `name` 属性。每个 `dialog` 元素都具有定义该对话框是否可见的 `visible` 属性。默认值是 `true`。`name` 属性的可能值如下所示：
 - `"checkForUpdate"` — 与“检查更新”、“没有更新”和“更新错误”对话框相对应
 - `"downloadUpdate"` — 与“下载更新”对话框相对应
 - `"downloadProgress"` — 与“下载进度”和“下载错误”对话框相对应
 - `"installUpdate"` — 与“安装更新”对话框相对应
 - `"fileUpdate"` — 与“文件更新”、“文件没有更新”和“文件错误”对话框相对应
 - `"unexpectedError"` — 与“意外错误”对话框相对应

在设置为 `false` 时，对应的对话框不作为更新过程的一部分进行显示。

下面是 `ApplicationUpdater` 框架的配置文件的一个示例：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
    <url>http://example.com/updates/update.xml</url>
    <delay>1</delay>
</configuration>
```

下面是 `ApplicationUpdaterUI` 框架的配置文件的一个示例，其中包括 `defaultUI` 元素的定义：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration xmlns="http://ns.adobe.com/air/framework/update/configuration/1.0">
    <url>http://example.com/updates/update.xml</url>
    <delay>1</delay>
    <defaultUI>
        <dialog name="checkForUpdate" visible="false" />
        <dialog name="downloadUpdate" visible="false" />
        <dialog name="downloadProgress" visible="false" />
    </defaultUI>
</configuration>
```

将 `configurationFile` 属性指向该文件的位置，如下所示：

- ActionScript:

```
appUpdater.configurationFile = new File("app:/cfg/updateConfig.xml");
```

- JavaScript:

```
appUpdater.configurationFile = new air.File("app:/cfg/updateConfig.xml");
```

更新框架的模板目录包括一个示例配置文件 `config-template.xml`。

定义更新设置 ActionScript 或 JavaScript 代码

这些配置参数还可以使用应用程序中的代码进行设置，如下所示：

```
appUpdater.updateURL = " http://example.com/updates/update.xml";
appUpdater.delay = 1;
```

updater 对象的属性为 `updateURL` 和 `delay`。这些属性所定义的配置与配置文件中的 `updateURL` 和 `delay` 元素相同：更新描述符文件的 URL 以及检查更新的时间间隔。如果您在代码中指定配置文件和设置，则使用代码设置的任意属性优先于配置文件中相应的设置。

必须先通过配置文件或通过脚本定义 `updateURL` 属性（请参阅第 323 页的“[定义更新描述符文件并将 AIR 文件添加到 Web 服务器](#)”），然后才能使用 **updater**（在调用 **updater** 对象的 `initialize()` 方法之前，在第 327 页的“[初始化更新框架](#)”中进行了说明）。

ApplicationUpdaterUI 框架定义 **updater** 对象的以下这些附加属性：

- `isCheckForUpdateVisible` — 与“检查更新”、“没有更新”和“更新错误”对话框相对应
- `isDownloadUpdateVisible` — 与“下载更新”对话框相对应
- `isDownloadProgressVisible` — 与“下载进度”和“下载错误”对话框相对应
- `isInstallUpdateVisible` — 与“安装更新”对话框相对应
- `isFileUpdateVisible` — 与“文件更新”、“文件没有更新”和“文件错误”对话框相对应
- `isUnexpectedErrorVisible` — 与“意外错误”对话框相对应

每个属性都对应于 **ApplicationUpdaterUI** 用户界面中的一个或多个对话框。每个属性均为布尔值，默认值为 `true`。在设置为 `false` 时，对应的对话框不作为更新过程的一部分进行显示。

这些对话框属性优先于更新配置文件中的设置。

更新过程

AIR 更新框架可按以下步骤完成更新过程：

- 1 **updater** 初始化检查更新检查是否已在定义的延迟间隔内执行（请参阅第 324 页的“[配置更新设置](#)”）。如果更新检查已达到定义的时间，则更新过程继续进行。
- 2 **updater** 下载并解释更新描述符文件。
- 3 **updater** 下载更新 AIR 文件。
- 4 **updater** 安装应用程序的更新版本。

updater 对象在完成每个步骤时对事件进行调度。在 **ApplicationUpdater** 版本中，可以取消指示过程中某个步骤成功完成的事件。如果取消其中一个事件，则过程中的下一步也会被取消。在 **ApplicationUpdaterUI** 版本中，**updater** 提供了允许用户取消或继续进行过程中每个步骤的对话框。

如果取消该事件，则可以调用 **updater** 对象的方法继续进行此过程。

由于 **updater** 的 **ApplicationUpdater** 版本是通过更新过程执行的，因此它在 `currentState` 属性中记录了其当前的状态。此属性设置为具有以下可能值的字符串：

- “UNINITIALIZED” - **updater** 尚未初始化。
- “INITIALIZING” - **updater** 正在初始化。
- “READY” - **updater** 已初始化
- “BEFORE_CHECKING” - **updater** 尚未检查是否有更新描述符文件。
- “CHECKING” - **updater** 正在检查是否有更新描述符文件。
- “AVAILABLE” - **updater** 描述符文件可用。
- “DOWNLOADING” - **updater** 正在下载 AIR 文件。
- “DOWNLOADED” - **updater** 已下载 AIR 文件。
- “INSTALLING” - **updater** 正在安装 AIR 文件。

- "PENDING_INSTALLING" - `updater` 已初始化，但存在未处理的更新。

`updater` 对象的某些方法仅在 `updater` 处于特定的状态下才执行。

初始化更新框架

在设置配置属性（请参阅第 322 页的“[基本示例：使用 ApplicationUpdaterUI 版本](#)”）之后，调用 `initialize()` 方法对更新进行初始化：

```
appUpdater.initialize();
```

此方法执行以下操作：

- 它将初始化更新框架，以静默方式同步安装所有未处理的更新。在应用程序启动过程中需要调用此方法，因为调用此方法时可以重新启动应用程序。
- 检查是否存在推迟的更新并进行安装；
- 如果在更新过程中出现错误，则该方法会从应用程序存储区域中清除更新文件和版本信息。
- 如果延迟已到期，则该方法会启动更新过程。否则，将重新启动计时器。

调用此方法可能会导致 `updater` 对象调度以下事件：

- `UpdateEvent.INITIALIZED` — 初始化完成时调度此事件。
- `ErrorEvent.ERROR` — 初始化过程中出现错误时调度此事件。

在调度 `UpdateEvent.INITIALIZED` 事件时，更新过程已完成。

在调用 `initialize()` 方法时，`updater` 根据计时器延迟设置启动更新过程并完成所有步骤。但是，随时都可以通过调用 `updater` 对象的 `checkNow()` 方法来启动更新过程：

```
appUpdater.checkNow();
```

如果更新过程已在运行，则此方法不执行任何操作。否则，将启动更新过程。

由于调用 `checkNow()` 方法，`updater` 对象可以调度以下事件：

- `UpdateEvent.CHECK_FOR_UPDATE` 事件仅在尝试下载更新描述符文件之前调度。

如果取消 `checkForUpdate` 事件，则可以调用 `updater` 对象的 `cancelUpdate()` 方法。（请参阅下一节。）如果不取消该事件，则更新过程会继续检查更新描述符文件。

管理 ApplicationUpdaterUI 版本中的更新过程

在 ApplicationUpdaterUI 版本中，用户可以通过用户界面的对话框中的“取消”按钮来取消过程。此外，可以通过调用 `ApplicationUpdaterUI` 对象的 `cancelUpdate()` 方法，以编程方式取消更新过程。

可以设置 `ApplicationUpdaterUI` 对象的属性或定义更新配置文件中的元素来指定 `updater` 显示哪些对话框确认。有关详细信息，请参阅第 324 页的“[配置更新设置](#)”。

管理 ApplicationUpdater 版本中的更新过程

可以调用 `ApplicationUpdater` 对象调度的事件对象的 `preventDefault()` 方法来取消更新过程的步骤（请参阅第 326 页的“[更新过程](#)”）。取消默认行为让您的应用程序有机会向用户显示消息，询问其是否要继续。

以下部分将介绍在取消过程的某个步骤后，如何继续进行更新过程。

下载并解释更新描述符文件

在更新过程开始之前 ApplicationUpdater 对象调度 checkForUpdate 事件，紧接着 **updater** 就会尝试下载更新描述符文件。如果取消 checkForUpdate 事件的默认行为，则 **updater** 将不下载更新描述符文件。可以调用 checkForUpdate() 方法恢复更新过程：

```
appUpdater.checkForUpdate();
```

调用 checkForUpdate() 方法会导致 **updater** 以异步方式下载和解释更新描述符文件。由于调用 checkForUpdate() 方法，**updater** 对象可以调度以下事件：

- StatusUpdateEvent.UPDATE_STATUS — **updater** 已成功下载并解释更新描述符文件时调度此事件。此事件具有以下属性：
 - available — 一个布尔值。如果存在不同于当前应用程序版本的可用版本，则设置为 true；否则（版本相同），设置为 false。
 - version - 一个字符串。更新文件的应用程序描述符文件的版本
 - details — 一个数组。如果没有描述的本地化版本，则此数组返回一个空字符串 ("") 作为第一个元素，并返回描述作为第二个元素。
- 如果存在多个版本的描述（位于更新描述符文件中），则该数组包含多个子数组。每个数组包含两个元素：第一个元素为语言代码（例如 "en"），第二个元素为该语言的相应描述（一个字符串）。请参阅第 323 页的“[定义更新描述符文件并将 AIR 文件添加到 Web 服务器](#)”。
- StatusUpdateErrorEvent.UPDATE_ERROR — 存在错误，而且 **updater** 无法下载或解释更新描述符文件时调度此事件。

下载更新 AIR 文件

updater 成功下载并解释更新描述符文件之后，ApplicationUpdater 对象将调度 updateStatus 事件。默认行为是开始下载更新文件（如果可用）。如果取消默认行为，则可以调用 downloadUpdate() 方法恢复更新过程：

```
appUpdater.downloadUpdate();
```

调用此方法会导致 **updater** 异步下载 AIR 文件的更新版本。

downloadUpdate() 方法可以调度以下事件：

- UpdateEvent.DOWNLOAD_START — 建立到服务器的连接时调度此事件。在使用 ApplicationUpdaterUI 库时，此事件将显示带有进度栏的对话框以对下载进度进行跟踪。
- ProgressEvent.PROGRESS — 根据文件下载进度定期调度此事件。
- DownloadErrorEvent.DOWNLOAD_ERROR — 如果在连接或下载更新文件时出现错误，则调度此事件。HTTP 状态无效时也会调度此事件（例如“404 - File not found”（404 - 找不到文件））。此事件具有 errorID 属性，该属性为定义其它错误信息的整数。此外，还具有另一个属性 subErrorID，该属性可能包含更多错误信息。
- UpdateEvent.DOWNLOAD_COMPLETE — **updater** 已成功下载并解释更新描述符文件时调度此事件。如果不取消此事件，则 ApplicationUpdater 版本会继续安装更新版本。在 ApplicationUpdaterUI 版本中，向用户显示为其提供选项的对话框以继续操作。

更新应用程序

更新文件下载完毕后，ApplicationUpdater 对象将调度 downloadComplete 事件。如果取消默认行为，则可以调用 installUpdate() 方法恢复更新过程：

```
appUpdater.installUpdate(file);
```

调用此方法会导致 **updater** 安装 AIR 文件的更新版本。此方法包括一个参数 file，该参数是引用要用作更新的 AIR 文件的 File 对象。

由于调用 `installUpdate()` 方法， `ApplicationUpdater` 对象可以调度 `beforeInstall` 事件：

- `UpdateEvent.BEFORE_INSTALL` — 仅在安装更新前调度此事件。有时，阻止目前更新的安装非常有用，这样用户可以完成当前的工作，然后再继续进行更新。调用 `Event` 对象的 `preventDefault()` 方法会将安装推迟到下次重新启动，以不能再启动任何其它更新过程。（这些更新包括通过调用 `checkNow()` 方法或由于定期检查而产生的更新。）

从任意 AIR 文件进行安装

可以调用 `installFromAIRFile()` 方法，安装要从用户计算机上的 AIR 文件进行安装的更新版本：

```
appUpdater.installFromAIRFile();
```

此方法会导致 `updater` 从 AIR 文件安装应用程序的更新版本。

`installFromAIRFile()` 方法可以调度以下事件：

- `StatusFileUpdateEvent.FILE_UPDATE_STATUS` - 在 `ApplicationUpdater` 成功验证使用 `installFromAIRFile()` 方法发送的文件后调度此事件。此事件具有以下属性：
 - `available` — 如果存在不同于当前应用程序版本的可用版本，则设置为 `true`；否则（版本相同），设置为 `false`。
 - `version` — 表示新的可用版本的字符串。
 - `path` — 表示更新文件的本机路径。

如果 `StatusFileUpdateEvent` 对象的可用属性设置为 `true`，则可以取消此事件。取消该事件可阻止更新继续进行。调用 `installUpdate()` 方法可继续进行取消的更新。

- `StatusFileUpdateErrorEvent.FILE_UPDATE_ERROR` — 存在错误且 `updater` 无法安装 AIR 应用程序时调度此事件。

取消更新过程

可以调用 `cancelUpdate()` 方法取消更新过程：

```
appUpdater.cancelUpdate();
```

此方法可取消所有未处理的下载，删除所有不完整的下载文件，并可重新启动定期检查计时器。

如果 `updater` 对象正在初始化，则该方法不执行任何操作。

本地化 ApplicationUpdaterUI 界面

`ApplicationUpdaterUI` 类为更新过程提供默认的用户界面。该用户界面包括允许用户启动过程、取消过程以及执行其它相关操作的对话框。

使用更新描述符文件的 `description` 元素，您可以用多种语言定义应用程序的描述。使用定义 `lang` 属性的多个 `text` 元素，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<update xmlns="http://ns.adobe.com/air/framework/update/description/1.0">
  <version>1.1a1</version>
  <url>http://example.com/updates/sample_1.1a1.air</url>
  <description>
    <text xml:lang="en">English description</text>
    <text xml:lang="fr">French description</text>
    <text xml:lang="ro">Romanian description</text>
  </description>
</update>
```

更新框架将使用最适合最终用户的本地化链的描述。有关详细信息，请参阅“[定义更新描述符文件并将 AIR 文件添加到 Web 服务器](#)”。

Flex 开发人员可以直接将新的语言添加到 "ApplicationUpdaterDialogs" 包中。

JavaScript 开发人员可以调用 `updater` 对象的 `addResources()` 方法。此方法可动态地添加某种语言的新资源包。该资源包定义某种语言的本地化字符串。这些字符串用于各种对话框的文本字段。

JavaScript 开发人员可以使用 `ApplicationUpdaterUI` 类的 `localeChain` 属性来定义用户界面所使用的区域设置链。通常只有 JavaScript (HTML) 开发人员使用此属性。Flex 开发人员可以使用 `ResourceManager` 管理区域设置链。

例如，以下 JavaScript 代码定义了罗马尼亚语和匈牙利语的资源包：

```
appUpdater.addResources("ro_RO",
    {titleCheck: "Titlu", msgCheck: "Mesaj", btnCheck: "Buton"});
appUpdater.addResources("hu", {titleCheck: "Cím", msgCheck: "Üzenet"});
var languages = ["ro", "hu"];
languages = languages.concat(air.Capabilities.languages);
var sortedLanguages = air.Localizer.sortLanguagesByPreference(languages,
    air.Capabilities.language,
    "en-US");
sortedLanguages.push("en-US");
appUpdater.localeChain = sortedLanguages;
```

有关详细信息，请参阅语言参考中对 `ApplicationUpdaterUI` 类的 `addResources()` 方法的说明。

第 38 章：查看源代码

就像用户可以在 Web 浏览器中查看 HTML 页面的源代码一样，用户也可以查看基于 HTML 的 AIR 应用程序的源代码。Adobe® AIR® SDK 包含 `AIRSourceViewer.js` JavaScript 文件，您可以在应用程序中使用该文件，以便轻松地向最终用户显示源代码。

加载、配置和打开 Source Viewer

Source Viewer 代码包含在 JavaScript 文件 `AIRSourceViewer.js` 中，该文件包含在 AIR SDK 的 `frameworks` 目录中。若要在应用程序中使用 Source Viewer，请将 `AIRSourceViewer.js` 复制到应用程序的项目目录中，并在应用程序的 HTML 主文件中通过脚本标签加载该文件：

```
<script type="text/javascript" src="AIRSourceViewer.js"></script>
```

`AIRSourceViewer.js` 文件定义一个 `SourceViewer` 类，您可以通过从 JavaScript 代码调用 `air.SourceViewer` 来访问该类。

`SourceViewer` 类定义三种方法：`getDefault()`、`setup()` 和 `viewSource()`。

方法	说明
<code>getDefault()</code>	静态方法。返回 <code>SourceViewer</code> 实例，可用于调用其它方法。
<code>setup()</code>	对 Source Viewer 应用配置设置。有关详细信息，请参阅第 331 页的“ 配置 Source Viewer ”
<code>viewSource()</code>	打开一个新窗口，用户可以在其中浏览和打开主机应用程序的源文件。

注：使用 Source Viewer 的代码必须位于应用程序安全沙箱（在应用程序目录中的文件中）中。

例如，下面的 JavaScript 代码可以实例化 Source Viewer 对象和打开列出所有源文件的 Source Viewer 窗口：

```
var viewer = air.SourceViewer.getDefault();
viewer.viewSource();
```

配置 Source Viewer

`config()` 方法对 Source Viewer 应用给定设置。此方法使用一个参数：`configObject`。`configObject` 对象具有用于为 Source Viewer 定义配置设置的属性。这些属性包括 `default`、`exclude`、`initialPosition`、`modal`、`typesToRemove` 和 `typesToAdd`。

default

指定要在 Source Viewer 中显示的初始文件相对路径的字符串。

例如，下面的 JavaScript 代码可以打开 Source Viewer 窗口，其中将 `index.html` 文件作为初始文件显示：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.default = "index.html";
viewer.viewSource(configObj);
```

exclude

指定要从 Source Viewer 列表中排除的文件或目录的字符串数组。其路径相对于应用程序目录。不支持通配符。

例如，下面的 JavaScript 代码可以打开 Source Viewer 窗口，该窗口列出了除 `AIRSourceViewer.js` 文件以外的所有源文件以及 `Images` 和 `Sounds` 子目录中的文件：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.exclude = ["AIRSourceViewer.js", "Images" "Sounds"];
viewer.viewSource(configObj);
```

initialPosition

包含两个数字的数组，用于指定 Source Viewer 窗口的初始 x 坐标和 y 坐标。

例如，下面的 JavaScript 代码可以在屏幕坐标 [40, 60] (X = 40, Y = 60) 位置打开 Source Viewer 窗口：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.initialPosition = [40, 60];
viewer.viewSource(configObj);
```

modal

一个布尔值，用于指定 Source Viewer 应为模式 (true) 窗口还是非模式 (false) 窗口。默认情况下，Source Viewer 窗口为模式窗口。

例如，下列 JavaScript 代码将打开 Source Viewer 窗口，从而用户可以与 Source Viewer 窗口及任何应用程序窗口进行交互：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.modal = false;
viewer.viewSource(configObj);
```

typesToAdd

指定除所包含的默认类型之外，要在 Source Viewer 列表中包含的文件类型的字符串数组。

默认情况下，Source Viewer 将列出以下文件类型：

- 文本文件 - TXT、XML、MXML、HTM、HTML、JS、AS、CSS、INI、BAT、PROPERTIES、CONFIG
- 图像文件 - JPG、JPEG、PNG、GIF

如果未指定任何值，则将包含所有默认类型（那些在 typesToExclude 属性中指定的类型除外）。

例如，下面的 JavaScript 代码可以打开包含 VCF 文件和 VCARD 文件的 Source Viewer 窗口：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.typesToAdd = ["text.vcf", "text.vcard"];
viewer.viewSource(configObj);
```

对于列出的每个文件类型，必须指定“text”（对于文本文件类型）或“image”（对于图像文件类型）。

typesToExclude

指定要从 Source Viewer 中排除的文件类型的字符串数组。

默认情况下，Source Viewer 将列出以下文件类型：

- 文本文件 - TXT、XML、MXML、HTM、HTML、JS、AS、CSS、INI、BAT、PROPERTIES、CONFIG
- 图像文件 - JPG、JPEG、PNG、GIF

例如，下面的 JavaScript 代码可以打开未列出 GIF 文件或 XML 文件的 Source Viewer 窗口：

```
var viewer = air.SourceViewer.getDefault();
var configObj = {};
configObj.typesToExclude = ["image.gif", "text.xml"];
viewer.viewSource(configObj);
```

对于列出的每个文件类型，必须指定 "text" (对于文本文件类型) 或 "image" (对于图像文件类型)。

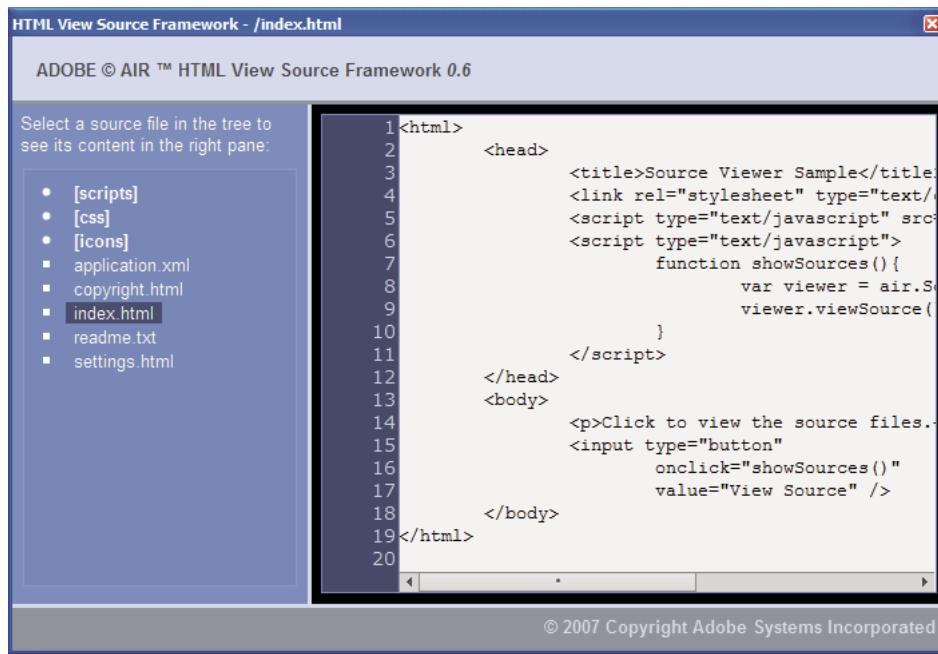
打开 Source Viewer

应该包含用户界面元素 (如，链接、按钮或菜单命令)，用于在用户选择它时调用 Source Viewer 代码。例如，以下简单的应用程序将在用户单击某个链接时 Source Viewer：

```
<html>
    <head>
        <title>Source Viewer Sample</title>
        <script type="text/javascript" src="AIRSourceViewer.js"></script>
        <script type="text/javascript">
            function showSources() {
                var viewer = air.SourceViewer.getDefault();
                viewer.viewSource()
            }
        </script>
    </head>
    <body>
        <p>Click to view the source files.</p>
        <input type="button"
            onclick="showSources()"
            value="View Source" />
    </body>
</html>
```

Source Viewer 用户界面

当应用程序调用 SourceViewer 对象的 viewSource() 方法时，AIR 应用程序将打开 Source Viewer 窗口。该窗口包含一个源文件和目录的列表（在左侧），以及显示所选文件的源代码的显示区域（在右侧）：



目录以带括号的形式列出。用户可以单击大括号来展开或折叠目录列表。

Source Viewer 可以显示带有可识别扩展名（如 HTML、HTML、JS、TXT、XML 及其它）的文本文件的源代码，也可以显示带有可识别图像扩展名（JPG、JPEG、PNG 和 GIF）的图像文件的源代码。如果用户选择不带可识别文件扩展名的文件，则会显示错误消息（“无法从此文件类型获取文本内容”(Cannot retrieve text content from this filetype)）。

不会列出通过 setup() 方法排除的任何源文件（请参阅第 331 页的“[加载、配置和打开 Source Viewer](#)”）。

第 39 章：本地化 AIR 应用程序

Adobe® AIR® 支持多种语言（从 1.1 版开始）。

本地化简介

本地化即包含资源以支持多种区域设置的过程。区域设置为语言和国家 / 地区代码的组合。例如，en_US 指美国英语，fr_FR 指法国法语。若要针对这些区域设置本地化应用程序，则应提供两组资源：一组用于 en_US 区域设置，一组用于 fr_FR 区域设置。

区域设置可以共享语言。例如，en_US 和 en_GB（英国）是不同的区域设置。在此情况下，虽然这两个区域设置都使用英语，但国家 / 地区代码指示它们为不同的区域设置，并可能因此使用不同的资源。例如，en_US 区域设置下的应用程序可能将单词拼写为“color”，而在 en_GB 区域设置下则可能拼写为“colour”。同样，根据区域设置的不同，货币单位也将以美元或英镑表示，并且日期和时间的格式可能也有所不同。

您也可以在不指定国家 / 地区代码的情况下为一种语言提供一组资源。例如，您可以为英语提供 en 资源并为 en_US 区域设置提供特定于美式英语的其它资源。

AIR SDK 提供了一个 HTML 本地化框架（包含在 AIRLocalizer.js 文件中）。该框架包括帮助处理多个区域设置的 API。有关详细信息，请参阅第 336 页的“[本地化 HTML 内容](#)”。

本地化不仅仅是翻译应用程序中使用的字符串。它还包括对任何类型的资源（例如音频文件、图像和视频）的翻译。

对应用程序安装程序中的应用程序名称和说明进行本地化

您可以为应用程序描述符文件中的 name 和 description 元素指定多种语言。例如，以下示例用三种语言（英语、法语和德语）指定应用程序名称：

```
<name>
  <text xml:lang="en">Sample 1.0</text>
  <text xml:lang="fr">Échantillon 1.0</text>
  <text xml:lang="de">Stichprobe 1.0</text>
</name>
```

每个文本元素的 xml:lang 属性用于指定语言代码，有关具体定义，请参阅 RFC4646 (<http://www.ietf.org/rfc/rfc4646.txt>)。

name 元素定义了 AIR 应用程序安装程序显示的应用程序名称。AIR 应用程序安装程序使用与操作系统设置定义的用户界面语言最匹配的本地化值。

同样，也可以在应用程序描述符文件中指定 description 元素的多种语言版本。该元素定义了 AIR 应用程序安装程序显示的说明文本。

这些设置仅适用于可在 AIR 应用程序安装程序中使用的语言。这些设置不定义可供运行的已安装应用程序使用的区域设置。AIR 应用程序可以提供支持多种语言的用户界面，包括但不限于那些可用于 AIR 应用程序安装程序的语言。

有关详细信息，请参阅第 103 页的“[在应用程序描述符文件中定义属性](#)”。

选择区域设置

若要确定应用程序使用的区域设置，您可以使用以下方法之一：

- 用户提示符 — 您可以在某些默认区域设置下启动应用程序，然后请用户选择其首选区域设置。
- Capabilities.languages — Capabilities.languages 属性列出了用户首选语言中可用语言数组，与通过操作系统设置一样。字符串包含 RFC4646 (<http://www.ietf.org/rfc/rfc4646.txt>) 定义的语言标记（如果适用，还包括脚本及区域信息）。这些字符串使用连字符作为分隔符（例如，“en-US”或“ja-JP”）。返回的数组中的第一项具有与 language 属性相同的主语言 ID。例如，如果将 languages[0] 设置为“en-US”，则会将 language 属性设置为“en”，但是，如果将 language 属性设置为“xu”（指定未知语言），则 languages 数组中的第一个元素将不同。
- Capabilities.language — Capabilities.language 提供了操作系统的用户界面语言代码。但是，此属性限制为 20 种已知语言。并且在英文系统中，此属性仅返回语言代码，而不返回国家 / 地区代码。出于以上原因，最好使用 Capabilities.languages 数组中的第一个元素。

本地化 HTML 内容

AIR 1.1 SDK 包括一个 HTML 本地化框架。AIRLocalizer.js JavaScript 文件定义了该框架。AIR SDK 的 frameworks 目录包含 AIRLocalizer.js 文件。该文件包含一个 air.Localizer 类，该类提供了帮助创建可支持多个本地化版本的应用程序的功能。

加载 AIR HTML 本地化框架代码

若要使用本地化框架，请将 AIRLocalizer.js 文件复制到您的项目中。然后使用 script 标签将其包括在应用程序的主要 HTML 文件中：

```
<script src="AIRLocalizer.js" type="text/javascript" charset="utf-8"></script>
```

后续的 JavaScript 可以调用 air.Localizer.localizer 对象：

```
<script>
    var localizer = air.Localizer.localizer;
</script>
```

air.Localizer.localizer 对象是定义使用和管理本地化资源的方法和属性的单个对象。Localizer 类包含以下方法：

方法	说明
getFile()	获取指定区域设置的指定资源包。请参阅第 341 页的“ 获取特定区域设置的资源 ”。
getLocaleChain()	返回区域设置链中的语言。请参阅第 340 页的“ 定义区域设置链 ”。
getResourceBundle()	将捆绑密钥和相应的值作为对象返回。请参阅第 341 页的“ 获取特定区域设置的资源 ”。
getString()	获取为资源定义的字符串。请参阅第 341 页的“ 获取特定区域设置的资源 ”。
setBundlesDirectory()	设置包目录的位置。请参阅第 339 页的“ 自定义 AIR HTML Localizer 设置 ”。
setLocalAttributePrefix()	设置由用于 HTML DOM 元素中的 localizer 属性使用的前缀。请参阅第 339 页的“ 自定义 AIR HTML Localizer 设置 ”。

方法	说明
setLocaleChain()	设置区域设置链中语言的顺序。请参阅第 340 页的“ 定义区域设置链 ”。
sortLanguagesByPreference()	基于操作系统设置中区域设置的顺序，对区域设置链中的区域设置进行排序。请参阅第 340 页的“ 定义区域设置链 ”。
update()	使用当前区域设置链中的本地化字符串更新 HTML DOM (或 DOM 元素)。有关区域设置链的说明，请参阅第 338 页的“ 管理区域设置链 ”。有关 update() 方法的详细信息，请参阅第 339 页的“ 更新 DOM 元素以使用当前区域设置 ”。

Localizer 类包含以下静态属性：

属性	说明
localizer	返回对应用程序单个 Localizer 对象的引用。
ultimateFallbackLocale	应用程序不支持用户首选参数时使用的区域设置。请参阅第 340 页的“ 定义区域设置链 ”。

定义资源包

HTML 本地化框架从本地化文件中读取字符串的本地化版本。本地化文件是文本文件中经过序列化的基于键的值集合。本地化文件有时被称为包。

创建一个名为 `locale` 的应用程序项目目录的子目录。(您也可以使用其它名称，请参阅第 339 页的“[自定义 AIR HTML Localizer 设置](#)”。) 该目录将包括本地化文件。该目录被称为包目录。

针对应用程序支持的每个区域设置，都创建一个包目录的子目录。命名每个子目录以与区域设置代码相匹配。例如，将法语目录命名为“`fr`”并将英语目录命名为“`en`”。您可以使用下划线(`_`)字符定义具有语言和国家 / 地区代码的区域设置。例如，将美式英语目录命名为“`en_us`”。(也可以使用连字符代替下划线，如“`en-us`”。HTML 本地化框架支持这两种形式。)

您可以向 `locale` 子目录添加任意数目的资源文件。通常，为每种语言均创建一个本地化文件(并将该文件放在该语言的目录中)。HTML 本地化框架包括 `getFile()` 方法，利用该方法可以读取文件的内容(请参阅第 341 页的“[获取特定区域设置的资源](#)”)。

具有 `.properties` 文件扩展名的文件被认为是本地化属性文件。可以使用这些文件为区域设置定义键值对。属性文件在每行都定义了一个字符串值。例如，以下代码为名为 `greeting` 的键定义了一个字符串值 "Hello in English."：

```
greeting=Hello in English.
```

包含以下文本的属性文件定义了六个键值对：

```
title=Sample Application
greeting>Hello in English.
exitMessage=Thank you for using the application.
color1=Red
color2=Green
color3=Blue
```

此示例显示了要存储在 `en` 目录中的属性文件的英语版本。

此属性文件的法语版本放置在 `fr` 目录中：

```
title=Application Example
greeting>Bonjour en français.
exitMessage=Merci d'avoir utilisé cette application.
color1=Rouge
color2=Vert
color3=Bleu
```

您可以为不同种类的信息定义多个资源文件。例如，`legal.properties` 文件可能包含法律文本样本(如版权信息)。您可能希望在多个应用程序中重复使用该资源。同样，您可以定义单独的文件来为用户界面的不同部分定义本地化内容。

对这些文件使用 UTF-8 编码以支持多种语言。

管理区域设置链

当应用程序加载 AIRLocalizer.js 文件时，该文件将检查应用程序中定义的区域设置。这些区域设置对应于包目录的子目录（请参阅第 337 页的“[定义资源包](#)”）。此可用区域设置的列表称为区域设置链。AIRLocalizer.js 文件将根据操作系统设置定义的首选顺序自动对区域设置链进行排序。（Capabilities.languages 属性按首选顺序列出操作系统用户界面语言。）

因此，如果应用程序为“en”、“en_US”和“en_UK”区域设置定义了资源，则 AIR HTML Localizer 框架将对区域设置链进行相应排序。当应用程序在将“en”报告为主区域设置的系统中启动时，区域设置链的排序顺序为["en", "en_US", "en_UK"]。此时，应用程序首先在“en”包中查找资源，然后在“en_US”包中查找。

但是，如果系统将“en-US”报告为主区域设置，则排序将使用["en_US", "en", "en_UK"]。在此情况下，应用程序首先在“en_US”包中查找资源，然后在“en”包中查找。

默认情况下，应用程序会将区域设置链中的第一个区域设置定义为要使用的默认区域设置。您可以请用户在第一次运行应用程序时选择一个区域设置。然后，您可以选择将该选择存储在首选参数文件中，并在随后的应用程序启动中使用该区域设置。

应用程序可以在区域设置链中的任何区域设置中使用资源字符串。如果特定区域设置未定义资源字符串，则应用程序将为区域设置链中定义的其它区域设置使用下一个匹配的资源字符串。

可以通过调用 Localizer 对象的 setLocaleChain() 方法自定义区域设置链。请参阅第 340 页的“[定义区域设置链](#)”。

利用本地化的内容更新 DOM 元素

应用程序中的元素可以引用本地化属性文件中的键值。例如，以下示例中的 title 元素指定了 local_innerHTML 属性。本地化框架使用此属性查找本地化值。默认情况下，框架会查找以“local_”开头的属性名称。该框架将更新名称与“local_”之后的文本相匹配的属性。在本例中，框架会设置 innerHTML 元素的 title 属性。innerHTML 属性 (attribute) 将在默认属性 (property) 文件 (default.properties) 中使用为 mainWindowTitle 键定义的值：

```
<title local_innerHTML="default.mainWindowTitle"/>
```

如果当前区域设置未定义匹配的值，则 localizer 框架将搜索区域设置链的其余部分。该框架将使用区域设置链中定义了值的下一个区域设置。

在以下示例中，p 元素的文本 (innerHTML 属性 (attribute)) 使用在默认属性 (property) 文件中定义的 greeting 键的值：

```
<p local_innerHTML="default.greeting" />
```

在以下示例中，input 元素的值属性 (attribute) (和显示文本) 使用默认属性 (property) 文件中定义的 btnBlue 键的值：

```
<input type="button" local_value="default.btnBlue" />
```

若要更新 HTML DOM 以使用当前区域设置链中定义的字符串，请调用 Localizer 对象的 update() 方法。调用 update() 方法将使 Localizer 对象分析 DOM 并在其找到本地化 ("local_...") 属性处应用操作：

```
air.Localizer.localizer.update();
```

您可以为属性（如“innerHTML”）及其相应的本地化属性（如“local_innerHTML”）都定义值。在这种情况下，如果本地化框架在本地化链中找到一个匹配值，则仅覆盖该属性值。例如，以下元素定义了 value 和 local_value 属性：

```
<input type="text" value="Blue" local_value="default.btnBlue"/>
```

也可以只更新特定的 DOM 元素。请参阅下一节第 339 页的“[更新 DOM 元素以使用当前区域设置](#)”。

默认情况下，AIR HTML Localizer 使用“local_”作为定义元素本地化设置的属性的前缀。例如，默认情况下，local_innerHTML 属性定义了用于元素的 innerHTML 值的包和资源名称。同样，默认情况下，local_value 属性定义了用于元素的 value 属性的包和资源名称。可以将 Localizer 配置为使用“local_”之外的属性前缀。请参阅第 339 页的“[自定义 AIR HTML Localizer 设置](#)”。

更新 DOM 元素以使用当前区域设置

当 Localizer 对象更新 HTML DOM 时，将导致已标记元素根据当前区域设置链中定义的字符串使用属性值。若要使 HTML localizer 更新 HTML DOM，请调用 Localizer 对象的 update() 方法：

```
air.Localizer.localizer.update();
```

若要仅更新指定的 DOM 元素，请将其作为参数传递给 update() 方法。update() 方法只有一个参数 parentNode，该参数为可选参数。指定后，parentNode 参数将定义要本地化的 DOM 元素。调用 update() 方法并指定 parentNode 参数后，将为指定本地化属性的所有子元素设置本地化值。

以下面的 div 元素为例：

```
<div id="colorsDiv">
<h1 local_innerHTML="default.lblColors" ></h1>
<p><input type="button" local_value="default.btnBlue" /></p>
<p><input type="button" local_value="default.btnRed" /></p>
<p><input type="button" local_value="default.btnGreen" /></p>
</div>
```

若要更新此元素以使用当前区域设置链中定义的本地化字符串，请使用以下 JavaScript 代码：

```
var divElement = window.document.getElementById("colorsDiv");
air.Localizer.localizer.update(divElement);
```

如果在区域设置链中未找到键值，则本地化框架会将属性值设置为 "local_" 属性的值。例如，在上一个示例中，假设本地化框架无法找到 lblColors 键（在区域设置链中的任何一个 default.properties 文件中）的值。在此情况下，它将使用 "default.lblColors" 作为 innerHTML 值。使用该值指示（开发人员）缺少资源。

当 update() 方法在区域设置链中无法找到资源时，将调度 resourceNotFound 事件。air.Localizer.RESOURCE_NOT_FOUND 常量定义了字符串 "resourceNotFound"。该事件具有三个属性：bundleName、resourceName 和 locale。bundleName 属性是在其中未找到该资源的包的名称。resourceName 属性是在其中未找到该资源的包的名称。locale 属性是在其中未找到该资源的区域设置的名称。

update() 方法在找不到指定的包时将调度 bundleNotFound 事件。air.Localizer.BUNDLE_NOT_FOUND 常量定义了字符串 "bundleNotFound"。该事件具有两个属性：bundleName 和 locale。bundleName 属性是在其中未找到该资源的包的名称。locale 属性是在其中未找到该资源的区域设置的名称。

update() 方法采用异步方式操作（并异步调度 resourceNotFound 和 bundleNotFound 事件）。以下代码将为 resourceNotFound 和 bundleNotFound 事件设置事件侦听器：

```
air.Localizer.localizer.addEventListener(air.Localizer.RESOURCE_NOT_FOUND, rnfHandler);
air.Localizer.localizer.addEventListener(air.Localizer.BUNDLE_NOT_FOUND, bnfHandler);
air.Localizer.localizer.update();
function rnfHandler(event)
{
    alert(event.bundleName + ":" + event.resourceName + ":" + event.locale);
}
function bnfHandler(event)
{
    alert(event.bundleName + ":" + event.locale);
}
```

自定义 AIR HTML Localizer 设置

利用 Localizer 对象的 setBundlesDirectory() 方法可以自定义包目录路径。利用 Localizer 对象的 setLocalAttributePrefix() 方法可以自定义包目录路径以及自定义 Localizer 使用的属性值。

默认的包目录定义为应用程序目录的区域设置子目录。可以通过调用 Localizer 对象的 setBundlesDirectory() 方法指定另一个目录。此方法将使用一个以字符串形式表示的参数 path（它是所需包目录的路径）。path 参数的值可以为以下任意值：

- 用于定义与应用程序目录有关的路径的字符串，如 "locales"

- 用于定义使用 app、app-storage 或 file URL 方案的有效 URL 的字符串，如 "app://languages"（请勿 使用 http URL 方案）
- File 对象

有关 URL 和目录路径的信息，请参阅 第 163 页的 “[File 对象的路径](#)”。

例如，下列代码将包目录设置为应用程序存储目录的语言子目录（而非应用程序目录）：

```
air.Localizer.localizer.setBundlesDirectory("languages");
```

将有效的路径作为 path 参数传递。否则，该方法将引发 BundlePathNotFoundError 异常。此错误将 "BundlePathNotFoundError" 作为其 name 属性，而其 message 属性则指定了无效的路径。

默认情况下，AIR HTML Localizer 使用 "local_" 作为定义元素本地化设置的属性的前缀。例如，local_innerHTML 属性定义了用于以下 innerHTML 元素的 input 值的包和资源名称：

```
<p local_innerHTML="default.greeting" />
```

利用 Localizer 对象的 setLocalAttributePrefix() 方法，可以使用 "local_" 之外的属性前缀。此静态方法将使用一个参数，而该参数是要用作属性前缀的字符串。例如，以下代码将本地化框架设置为使用 "loc_" 作为属性前缀：

```
air.Localizer.localizer.setLocalAttributePrefix("loc_");
```

您可以自定义本地化框架使用的属性前缀。如果默认值 ("local_") 与代码所使用的另一个属性的名称冲突，则可能要自定义该前缀。请确保调用此方法时使用对 HTML 属性有效的字符。（例如，该值不能包含空格字符。）

有关在 HTML 元素中使用本地化属性的详细信息，请参阅第 338 页的 “[利用本地化的内容更新 DOM 元素](#)”。

包目录和属性前缀设置在不同的应用程序会话之间不会保留。如果使用自定义包目录或属性前缀设置，请确保每次启动应用程序时对其进行设置。

定义区域设置链

默认情况下，加载 AIRLocalizer.js 代码时，它将设置默认的区域设置链。包目录和操作系统语言设置中可用的区域设置定义了该区域设置链。（有关详细信息，请参阅第 338 页的 “[管理区域设置链](#)”。）

可以通过调用 Localizer 对象的静态 setLocaleChain() 方法修改区域设置链。例如，如果用户为某个特定语言指示一个首选参数，则您最好调用此方法。setLocaleChain() 方法使用一个参数 chain，该参数为一个区域设置数组，如 ["fr_FR", "fr", "fr_CA"]。数组中区域设置的顺序设定了框架查找资源的顺序（在后续操作中）。如果未在链中找到第一个区域设置的资源，它将继续查找其它区域设置的资源。如果 chain 参数丢失、不是数组或为空数组，则此功能将失败并引发 IllegalArgumentsError 异常。

Localizer 对象的静态 getLocaleChain() 方法返回一个数组，它列出当前区域设置链中的区域设置。

以下代码将读取当前区域设置链并将两个法语区域设置添加到链头：

```
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
```

当 setLocaleChain() 方法更新区域设置链时，将调度 "change" 事件。air.Localizer.LOCALE_CHANGE 常量定义了字符串 "change"。该事件具有一个 localeChain 属性，该属性为新的区域设置链中的区域设置代码数组。以下代码为此事件设置了一个事件侦听器：

```
var currentChain = air.Localizer.localizer.getLocaleChain();
newLocales = ["fr_FR", "fr"];
localizer.addEventListener(air.Localizer.LOCALE_CHANGE, changeHandler);
air.Localizer.localizer.setLocaleChain(newLocales.concat(currentChain));
function changeHandler(event)
{
    alert(event.localeChain);
}
```

静态 air.Localizer.ultimateFallbackLocale 属性表示在应用程序不支持用户首选参数时所用的区域设置。默认值是 "en"。您可以将其设置为另一个区域设置，如以下代码中所示：

```
air.Localizer.ultimateFallbackLocale = "fr";
```

获取特定区域设置的资源

Localizer 对象的 `getString()` 方法返回为特定区域设置中的资源定义的字符串。调用此方法时，不必指定 `locale` 值。在此情况下，该方法将查看整个区域设置链并返回提供给定资源名称的第一个区域设置中的字符串。此方法具有以下参数：

参数	说明
<code>bundleName</code>	包含资源的包。这是不带 <code>.properties</code> 扩展名的属性文件的文件名。(例如，如果此参数设置为 "alerts"，则 Localizer 代码将在名为 <code>alerts.properties</code> 的本地化文件中查找。)
<code>resourceName</code>	资源名称。
<code>templateArgs</code>	可选。用于替换替换字符串中的编号标签的字符串数组。以调用 <code>templateArgs</code> 参数为 <code>["Raúl", "4"]</code> 且匹配的资源字符串为 "Hello, {0}. You have {1} new messages." 的函数为例。在此情况下，此函数将返回 "Hello, Raúl. You have 4 new messages."。若要忽略此设置，请传递 <code>null</code> 值。
<code>locale</code>	可选。要使用的区域设置代码 (如 "en"、"en_us" 或 "fr")。如果提供了一个区域设置但未找到匹配值，则此方法将不会继续在区域设置链中的其它区域设置中搜索值。如果未指定任何区域设置代码，则此函数将返回区域设置链中第一个区域设置中为给定的资源名称提供值的字符串。

本地化框架可以更新标记的 HTML DOM 属性。但是，可以通过其它方式使用本地化字符串。例如，可以在某些动态生成的 HTML 中使用字符串或在函数调用中将其作为参数值。例如，以下代码通过 `fr_FR` 区域设置的默认属性文件中 `error114` 资源内定义的字符串调用 `alert()` 函数：

```
alert(air.Localizer.localizer.getString("default", "error114", null, "fr_FR"));
```

当 `getString()` 方法在指定包中无法找到资源时，将调度 `resourceNotFound` 事件。`air.Localizer.RESOURCE_NOT_FOUND` 常量定义了字符串 "resourceNotFound"。该事件具有三个属性：`bundleName`、`resourceName` 和 `locale`。`bundleName` 属性是在其中未找到该资源的包的名称。`resourceName` 属性是在其中未找到该资源的包的名称。`locale` 属性是在其中未找到该资源的区域设置的名称。

`getString()` 方法在找不到指定的包时将调度 `bundleNotFound` 事件。`air.Localizer.BUNDLE_NOT_FOUND` 常量定义了字符串 "bundleNotFound"。该事件具有两个属性：`bundleName` 和 `locale`。`bundleName` 属性是在其中未找到该资源的包的名称。`locale` 属性是在其中未找到该资源的区域设置的名称。

`getString()` 方法采用异步方式操作 (并异步调度 `resourceNotFound` 和 `bundleNotFound` 事件)。以下代码将为 `resourceNotFound` 和 `bundleNotFound` 事件设置事件侦听器：

```
air.Localizer.localizer.addEventListener(air.Localizer.RESOURCE_NOT_FOUND, rnfHandler);
air.Localizer.localizer.addEventListener(air.Localizer.BUNDLE_NOT_FOUND, bnfHandler);
var str = air.Localizer.localizer.getString("default", "error114", null, "fr_FR");
function rnfHandler(event)
{
    alert(event.bundleName + ":" + event.resourceName + ":" + event.locale);
}
function bnfHandler(event)
{
    alert(event.bundleName + ":" + event.locale);
}
```

Localizer 对象的 `getResourceBundle()` 方法为给定的区域设置返回一个指定的包。该方法的返回值是其属性与包中的键相匹配的对象。(如果应用程序找不到指定的包，则该方法返回 `null`。)

该方法采用两个参数 — `locale` 和 `bundleName`。

参数	说明
locale	区域设置（如“fr”）。
bundleName	包名称。

例如，以下代码调用 `document.write()` 方法加载 fr 区域设置的默认包。然后，调用 `document.write()` 方法写入该包中 str1 和 str2 键的值：

```
var aboutWin = window.open();
var bundle = localizer.getResourceBundle("fr", "default");
aboutWin.document.write(bundle.str1);
aboutWin.document.write("<br/>");
aboutWin.document.write(bundle.str2);
aboutWin.document.write("<br/>");
```

`getResourceBundle()` 方法在找不到指定的包时将调度 `bundleNotFound` 事件。`air.Localizer.BUNDLE_NOT_FOUND` 常量定义了字符串 “bundleNotFound”。该事件具有两个属性：`bundleName` 和 `locale`。`bundleName` 属性是在其中未找到该资源的包的名称。`locale` 属性是在其中未找到该资源的区域设置的名称。

`Localizer` 对象的 `getFile()` 方法将以字符串形式为给定区域设置返回包的内容。包文件将以 UTF-8 文件格式读取。此方法具有以下参数：

参数	说明
resourceFileName	资源文件的文件名（例如“about.html”）。
templateArgs	可选。用于替换替换字符串中的编号标签的字符串数组。以调用 <code>templateArgs</code> 参数为 <code>["Raúl", "4"]</code> 且匹配的资源文件包含以下两行的函数为例： <pre><html> <body>Hello, {0}. You have {1} new messages.</body> </html></pre> 在此情况下，此函数将返回两行字符串： <pre><html> <body>Hello, Raúl. You have 4 new messages. </body> </html></pre>
locale	要使用的区域设置代码，例如“en_GB”。如果提供了区域设置但未找到匹配的文件，则此方法将不会继续在区域设置链中的其它区域设置中搜索。如果未指定区域设置代码，则此函数将返回区域设置链中第一个区域设置中的文本，而该区域设置链具有一个与 <code>resourceFileName</code> 相匹配的文件。

例如，以下代码使用 fr 区域设置的 `about.html` 文件的内容调用 `document.write()` 方法：

```
var aboutWin = window.open();
var aboutHtml = localizer.getFile("about.html", null, "fr");
aboutWin.document.close();
aboutWin.document.write(aboutHtml);
```

当 `getFile()` 方法在区域设置链中无法找到资源时，将调度 `fileNotFound` 事件。`air.Localizer.FILE_NOT_FOUND` 常量定义了字符串 “resourceNotFound”。`getFile()` 方法采用异步方式操作（并异步调度 `fileNotFound` 事件）。该事件具有两个属性：`fileName` 和 `locale`。`fileName` 属性为未找到的文件的名称。`locale` 属性是在其中未找到该资源的区域设置的名称。以下代码为此事件设置了一个事件侦听器：

```
air.Localizer.localizer.addEventListener(air.Localizer.FILE_NOT_FOUND, fnfHandler);
air.Localizer.localizer.getFile("missing.html", null, "fr");
function fnfHandler(event)
{
    alert(event.fileName + ": " + event.locale);
}
```

对日期、时间和货币进行本地化

应用程序为每个区域设置显示日期、时间和货币的方式均有很大不同。例如，美国标准表示日期的方式为月 / 日 / 年，而欧洲标准表示日期的方式为日 / 月 / 年。

您可以编写代码以设置日期、时间和货币的格式。例如，以下代码将 Date 对象转换为月 / 日 / 年格式或日 / 月 / 年格式。如果将 locale 变量（表示区域设置）设置为 "en_US"，则函数会返回“月 / 日 / 年”格式。该示例将 Date 对象转换为所有其它区域设置的日 / 月 / 年格式：

```
function convertDate(date)
{
    if (locale == "en_US")
    {
        return (date.getMonth() + 1) + "/" + date.getDate() + "/" + date.getFullYear();
    }
    else
    {
        return date.getDate() + "/" + (date.getMonth() + 1) + "/" + date.getFullYear();
    }
}
```

有些 Ajax 框架支持对日期和数字进行本地化。

索引

- 符号**
- : (冒号) 字符, 在 SQL 语句参数名称中 218
- ? (问号) 字符, 在未命名 SQL 参数中 218
- @ (at) 字符, 在 SQL 语句参数名称中 218
- & (and 符) 294
- 数字**
- 1024-RSA 33
- 2048-RSA 33
- A**
- AC_FL_RunContent() 函数 (在 default_badge.html 中) 307
- AC_RuntimeActiveContent.js 306
- acompc 编译器 54
- Acrobat 62, 248
- Action Message Format (AMF) 201, 203
- activate() 方法 (NativeWindow 类) 122, 126, 127
- active 事件 131
- activeWindow 属性 (NativeApplication 类) 126
- activity 事件 266
- addChild() 方法 (Stage 类) 124
- addChildAt() 方法 (Stage 类) 124
- Adobe Acrobat 开发人员中心 249
- Adobe AIR
 - 安装 89
 - 更新 89
 - 简介 6
 - 卸载 2
- Adobe ColdFusion 296
- Adobe 出版社书籍 9
- Adobe Dreamweaver 46
- Adobe Media Player 268
- Adobe Reader 62, 248
- Adobe 文档 9
- Adobe 支持网站 9
- AES-CBC 128 位加密 246
- Ajax
 - 安全 97
 - 在应用程序沙箱中受支持 97
- AIR 程序**
 - 卸载 91
- AIR Debug Launcher (ADL)
 - 退出和错误代码 23
- AIR Developer Tool (ADT)
 - 创建自签名证书 32
 - 打包 AIR 文件 24
 - 签名选项 29
- AIR DevTools (AIR) 文件 31
- AIR HTML/JavaScript 应用程序内部检查器 36
- AIR 开发人员证书 (AIR Developer Certificate) 313
- air 属性 (AIRAliases.js 文件) 52, 61
- AIR 文件
 - 打包 24
 - 签名 312
- AIR 应用程序
 - 安装 89, 305
 - 安装路径 105
 - 版本 105, 286, 318
 - 版权信息 106
 - 调用 278
 - 分发 305
 - 更新 89, 108, 318
 - 检测是否已安装 310
 - 浏览器调用 108
 - 启动 278
 - 设置 102, 104, 285
 - 图标 108
 - 退出 278
 - 文件类型关联 108, 279, 286
 - 现有 278
 - 运行 305, 311
- AIR 应用程序安装程序中支持的语言 106
- AIR 应用程序的版权信息 106
- AIR 运行时
 - 更新 89
 - 检测 286, 309
 - 卸载 2
 - 修补级别 103, 286
- AIRAliases.js 文件 52, 61
- AIRI 文件
 - 使用 AIR Developer Tool (ADT) 创建 31
- AIRIntrospector.js 文件 36
- AIRLocalizer.js 文件 336
- AIRSourceViewer.js 文件 331
- allowBrowserInvocation 元素 (应用程序描述符文件) 108, 278, 282
- allowCrossDomainXHR 属性 (frame 和 iframe 元素) 64, 68
- allowDomain() 方法 (LocalConnection 类) 303
- allowInsecureDomain() 方法 (LocalConnection 类) 303
- alwaysInFront 属性 (NativeWindow 类) 126, 127
- and 符 (&) 294
- app URL 方案 53, 57, 63, 99, 100, 123, 168, 249
- AppInstallDisabled (Windows 注册表设置) 91
- Apple 开发人员证书 (Apple Developer Certificate) 313
- application/x-www-form-urlencoded 293
- applicationDescriptor 属性 (NativeApplication 类) 285
- applicationStorageDirectory 属性 (File 类) 164
- ApplicationUpdater_UI.swf 文件 322
- ApplicationUpdater.swf 文件 322
- app-storage URL 方案 91, 99, 100, 168, 249
- app-support URL 方案 57
- arguments 属性
 - BrowserInvokeEvent 类 282
 - InvokeEvent 类 279
- asfunction 协议 93
- at (@) 字符, 在 SQL 语句参数名称中 218
- attach() 方法 (SQLConnection 类) 227
- autoExit 属性
 - NativeApplication 类 283
- AUTOINCREMENT 列 (SQL) 226

安全
 Ajax 框架 97
 asfunction 协议 93
 安装 (应用程序和运行时) 89
 CSS 94
 动态代码生成 96
 eval() 函数 96
 frame 94
 非应用程序沙箱 93
 HTML 96
 iframe 94
 img 标签 93
 加载内容 123
 降级攻击 101
 跨脚本访问 98
 浏览器调用功能 282
 沙箱 92
 沙箱桥 95, 98
 window.open() 98
 文本字段 93
 文件系统 99
 XMLHttpRequest 对象 97
 应用程序存储目录 91
 应用程序沙箱 92
 用户安装权限 89
 用户凭据 101
 最佳做法 100
 安全性
 对数据进行加密 246
 HTML 47, 62, 94
 JavaScript 57
 JavaScript 错误 48
 剪贴板 193
 跨域缓存 93
 沙箱 56, 62, 63, 286
 沙箱桥 57
 数据库 219
 XMLHttpRequest 69
 安装 AIR 应用程序 305

B
 big-endian 字节顺序 202, 297
 bitmaps 属性 (Icon 类) 158
 bounce() 方法 (Icon 类) 159
 browseForDirectory() 方法 (File 类) 166

browseForOpen() 方法 (File 类) 167
 browseForSave() 方法 (File 类) 167
 browserInvoke 事件 282, 312
 BrowserInvokeEvent 类 282
 bufferTime 属性 (SoundMixer 类) 256
B
 ByteArray 类
 bytesAvailable 属性 202
 compress() 方法 203
 length 属性 202
 position 属性 201
 readBytes() 方法 201
 readFloat() 方法 201
 readInt() 方法 201
 readObject() 方法 201
 readUTFBytes() 方法 201
 uncompress() 方法 203
 writeBytes() 方法 201
 writeFloat() 方法 201
 writeInt() 方法 201
 writeObject() 方法 201
 writeUTFBytes() 方法 201
 bytesAvailable 属性 (ByteArray 类) 202
 bytesLoaded 属性 (ProgressEvent 类) 255
 bytesTotal 属性 (ProgressEvent 类) 255
 版本, AIR 应用程序 286
 本地化 335
 本地数据库
 请参阅数据库
 本机菜单
 参阅菜单
 本机窗口
 请参阅窗口
 表 (数据库) 211
 创建 214
 标题栏图标 (Windows) 120

C
 cancelable 属性 (Event 类) 74
 Canvas 对象 64, 70
 Capabilities
 语言属性 336
 childSandboxBridge 属性
 Window 对象 95
 ChosenSecurity 证书 312, 313

clearData() 方法
 ClipboardData 对象 65
 DataTransfer 对象 66, 186
 client 属性 (LocalConnection 类) 302
 clientX 属性 (HTML 拖动事件) 186
 clientY 属性 (HTML 拖动事件) 186
 Clipboard 65
 Clipboard 类
 generalClipboard 属性 192
 setData() 方法 199
 setDataHandler() 方法 199
 clipboardData 属性 (HTML 复制和粘贴事件) 193, 194
 clipboardData 属性 (剪贴板事件) 66
 ClipboardFormats 类 197
 ClipboardTransferModes 类 198
 close 事件 131
 close() 方法
 NativeWindow 类 127
 close() 方法 (Sound 类) 260
 close() 方法 (window 对象) 116
 closing 事件 76, 127, 131, 283
 Command 键 140
 complete 事件 56, 73, 253
 compress() 方法 (ByteArray 类) 203
 CompressionAlgorithm 类 203
 computeSpectrum() 方法 (SoundMixer 类) 262
 connect() 方法 (XMLSocket 类) 298
 content 元素 (应用程序描述符文件) 107
 contenteditable 属性 (HTML) 189
 contentType 属性 (URLRequest 类) 293
 ContextMenu 类 140
 contextmenu 事件 142
 ContextMenuItem 类 140
 cookie 65
 copyTo() 方法 (File 类) 173
 copyToAsync() 方法 (File 类) 173
 CREATE TABLE 语句 (SQL) 214
 createDirectory() 方法 (File 类) 171
 createElement() 方法 (Document 对象) 51
 createRootWindow() 方法 (HTMLLoader 类) 122, 123
 createTempDirectory() 方法 (File 类) 171, 174

- createTempFile() 方法 (File 类) 174
 creationDate 属性 (File 类) 173
 creator 属性 (File 类) 173
 CSS
 AIR 扩展 71
 从 ActionScript 访问 HTML 样式 56
 Ctrl 键 140
 currentDirectory 属性 (InvokeEvent 类) 279
 customUpdateUI 元素 (应用程序描述符文件) 108, 278, 319
 菜单 137
 创建 141
 窗口 141, 145
 弹出 141, 143
 等效键 140
 分隔线 142
 结构 138, 139
 类型 137
 默认系统 138
 事件 144
 事件流 139, 143
 使用类 137
 停靠栏 138
 停靠栏项目 141
 系统任务栏图标 138, 141
 项目 139
 应用程序 141, 145
 子菜单 139, 142
 自定义 138
 菜单栏 139
 菜单命令的等效键 140
 菜单命令的快捷键 140
 菜单项 139
 创建 142
 data, 分配到 141
 等效键 140
 复制和粘贴 196
 快捷键 140
 选择 144
 已启用 141
 已选中 141
 助记键字符 140
 状态 141
 参数, 在 SQL 语句中 218
 插件 (采用 HTML) 62
 尺寸, 窗口 107
 创建目录 171
 窗口 115
 背景 118
 初始 116
 初始化 120
 创建 120, 125
 大小 107, 122
 调整大小 107, 117, 129
 非矩形 118
 关闭 117, 127, 283
 管理 125
 还原 117, 128
 活动 126, 127
 激活 122
 简单 117
 可见性 107
 类型 117
 普通窗口 117
 事件 131
 事件流 117
 实用程序窗口 117
 使用的类 116
 属性 107
 顺序 127
 透明度 107, 118
 外观 117
 位置 107
 舞台缩放模式 122
 系统镶边 118
 显示 126
 显示顺序 126
 镶边 118
 行为 117
 样式 117
 移动 117, 129, 134
 隐藏 126
 自定义镶边 118
 最大大小 122
 最大化 107, 117, 128
 最小大小 122
 最小化 107, 117, 126, 128
 窗口菜单 137, 145
 创建 141
 窗口大小 107
 窗口的背景 118
 窗口的外观 117
 窗口可见性 107
 窗口顺序 126
 窗口位置 107
 错误代码
 DRM 274

D
 data 属性
 NativeMenuItem 类 141
 data 属性 (URLRequest 类) 294
 dataFormat 属性 (URLLoader 类) 296
 DataTransfer 对象
 types 属性 189
 DataTransfer 对象 (HTML 拖放) 66, 186, 187, 188, 189
 Date 对象, 在 ActionScript 和 JavaScript 之间
 转换 55
 打包 AIR 文件
 AIR Developer Tool (ADT) 24
 deactivate 事件 131
 decode() 方法 (URLVariables 类) 294
 default_badge.html 306
 deflate 压缩 203
 DELETE 语句 (SQL) 226
 deleteDirectory() 方法 (File 类) 172
 deleteDirectoryAsync() 方法 (File 类) 172
 deleteFile() 方法 (File 类) 174
 deleteFileAsync() 方法 (File 类) 174
 description 元素 (应用程序描述符文件) 106
 descriptor-sample.xml 文件 102
 designMode 属性 (Document 对象) 67, 189
 desktopDirectory 属性 (File 类) 164
 Dictionary 类 52
 dispatchEvent() 方法 (NativeWindow 类) 117
 display() 方法 (NativeMenu 类) 143
 displaying 事件 139, 144, 145
 displayState 属性 (Stage 类) 132
 displayStateChange 事件 117, 131
 displayStateChanging 事件 117, 131
 Document 对象
 createElement() 方法 51
 designMode 属性 67, 189
 stylesheets 属性 56

writeln() 方法 66
 write() 方法 51, 66, 97
 writeln() 方法 51, 97
 documentRoot 属性 (frame 和 iframe 元素) 57, 62, 68, 94
 documentsDirectory 属性 (File 类) 164
 dominitialize 事件 69
 DPAPI (使加密数据与用户相关联) 246
 drag 事件 66, 186
 dragend 事件 66, 186
 dragenter 事件 66, 186
 dragleave 事件 66, 186
 dragover 事件 66, 186
 dragstart 事件 66, 186
 DRM 268
 凭据 274
 DRMAuthenticateEvent 类 268, 273
 DRMErrorEvent 类 268
 错误代码 274
 subErrorCode 属性 274
 DRMStatusEvent 类 268
 drop 事件 66, 186
 dropEffect 属性 (DataTransfer 对象) 66, 186, 187
 大小, 窗口 122
 打印 62
 代理图标
 Mac OS 120
 代码签名 101, 312
 弹出菜单 137, 143
 创建 141
 弹出式窗口 126
 登录, 启动 AIR 应用程序 281
 等效键
 复制和粘贴 197
 调试 36
 使用 ADL 23
 调用 AIR 应用程序 278
 调整窗口大小 107, 117, 129
 动态代码生成 96
 读取文件 175
 对 AIR 文件签名 24
 对窗口进行排序 127
 对象, 检查和调试 36
 对象文本 (在 JavaScript 中) 96

E
 effectAllowed 属性 (DataTransfer 对象) 66, 186, 187, 188
 encoding 属性 (File 类) 170
 EncryptedLocalStore 类 246
 Endian.BIG_ENDIAN 202, 297
 Endian.LITTLE_ENDIAN 202, 297
 enterFrame 事件 124
 error 事件 217
 eval() 函数 47, 48, 63, 93, 96
 Event 类 74
 execute() 方法 (SQLStatement 类) 217, 220, 225
 exists 属性 (File 类) 173
 exit() 方法
 NativeApplication 类 283
 exiting 事件 283
 二进制数据
 请参阅字节数组
F
 FDB (debugger) 23
 File 类 162, 163
 applicationStorageDirectory 属性 163
 browseForDirectory() 方法 166
 browseForOpen() 方法 167
 browseForSave() 方法 167
 copyTo() 方法 173
 copyToAsync() 方法 173
 createDirectory() 方法 171
 createTempDirectory() 方法 171, 174
 createTempFile() 方法 174
 creationDate 属性 173
 creator 属性 173
 deleteDirectory() 方法 172
 deleteDirectoryAsync() 方法 172
 deleteFile() 方法 174
 deleteFileAsync() 方法 174
 desktopDirectory 属性 163
 documentsDirectory 属性 163
 encoding 属性 170
 exists 属性 173
 getDirectoryListingAsync() 方法 171
 getRootDirectories() 163
 getRootDirectories() 方法 163
 isDirectory 属性 173
 lineEnding 属性 170
 load() 方法 181
 modificationDate 属性 173
 moveTo() 方法 173
 moveToAsync() 方法 173
 moveToTrash() 方法 174
 moveToTrashAsync() 方法 174
 name 属性 173
 nativePath 属性 163, 173
 parent 属性 173
 relativize() 方法 168
 resolvePath() 方法 163
 save() 方法 181
 separator 属性 170
 size 属性 173
 spaceAvailable 属性 170
 type 属性 173
 url 属性 163, 173
 userDirectory 属性 163
 引用本地数据库 214
 file URL 方案 53, 99, 168
 FileMode 类 162
 filename 元素 (应用程序描述符文件) 105
 FileReference 类
 load() 方法 181
 save() 方法 181
 FileStream 类 162
 fileTypes 元素 (应用程序描述符文件) 108, 286
 Flash Media Rights Management Server 268
 Flash Player 52, 63
 FlashVars 设置 (以便使用 badge.swf) 307
 FLV 视频, 加密 268
 FMRMS (Flash Media Rights Management Server) 268
 frame 94
 frame 元素 62, 64, 68
 发行商标识别符 285, 313
 发行商名称 312
 范例应用程序 2
 翻译应用程序 335
 非应用程序沙箱 45, 47, 48, 57, 62, 63, 93, 191
 分发 AIR 应用程序 305
 分隔线, 菜单 142
 丰富 Internet 应用程序 (RIA) 6

服务器端脚本 296

复制和粘贴

等效键 197

HTML 65, 193

默认菜单项 (Mac OS) 196

延迟呈现 199

传输模式 198

复制目录 171

复制事件 194

复制文件 173

G

generalClipboard 属性 (Clipboard 类) 192

getApplicationVersion() 方法 (air.swf 文件)
310

getData() 方法

ClipboardData 对象 65

DataTransfer 对象 66, 189

HTML 复制和粘贴事件 194

getData() 方法 (属于 HTML 拖动事件的
dataTransfer 属性) 186

getDefaultApplication() 方法
(NativeApplication 类) 286

getDirectoryListing() 方法 (File 类) 171

getDirectoryListingAsync() 方法 (File 类) 171

getResult() 方法 (SQLStatement 类) 225

getScreensForRectangle() 方法 (Screen 类)
134

getStatus() 方法 (air.swf 文件) 309

GlobalSign 证书 312, 313

GZIP 格式 203

根卷 164

更新 AIR 应用程序 108, 318

更新 AIR 运行时或 AIR 应用程序时所需的权
限 89, 305, 311

更新框架 321

更新描述符文件 (更新框架) 324

更新配置文件 (更新框架) 324

工具栏 (Mac OS) 120

功能键, 菜单项 140

关闭窗口 117, 127, 283

关闭应用程序 283

关系数据库

请参阅数据库

光标, 拖放效果 187

H

hasEventListener() 方法 78

height 元素 (应用程序描述符文件) 107

Hello World 范例应用程序 46

hostContainer 属性 (PDF) 249

HTML

AIR 扩展 68

安全性 57, 62, 94

插件 62

窗口 122

打印 62

调试 36

叠加 SWF 内容 123

复制和粘贴 193

滚动 73

嵌入对象 62

沙箱 63

事件 73

拖放支持 185, 187

HTML DOM 和本机窗口 116

htmlBoundsChanged 事件 73

htmlDOMInitialize 事件 73

HTMLLoader 类

createRootWindow() 方法 122, 123

复制和粘贴 193

JavaScript 访问 61

loadString() 方法 98

paintsDefaultBackground 属性 118, 124

pdfCapability 属性 248

placeLoadStringContentInApplicationSan
dbox 属性 98

事件 73

htmlLoader 属性 (Window 对象) 61, 67, 121

htmlLoader 属性 (window 对象) 122

HTMLPDFCapability 类 248

HTTP 隧道 297

还原窗口 117, 128

函数 (JavaScript)

定义 96

构造函数 50

文本 96

函数构造函数 (采用 JavaScript) 63

活动 (用户), 检测 287

活动窗口 126

J

Java 加密体系结构 (JCA) 29

Java 套接字服务器 298

JavaScript

AIR 运行时和 60

AIR 支持 62

AIRAliases.js 文件 52, 61

安全性 57

避免安全错误 48

编程 46

错误 48, 77

错误事件 73

debugging 23

调试 36

访问 AIR API 51

PDF 249

事件, 处理 76

JavaScript 安全 96

javascript URL 方案 50, 68, 97

JavaSoft 开发人员证书 (JavaSoft Developer
Certificate) 314

Icon 类

bitmaps 属性 158

bounce() 方法 159

icon 属性 (NativeApplication 类) 158

icon 元素 (应用程序描述符文件) 108

id 元素 (NativeApplication 类) 285

id 元素 (应用程序描述符文件) 104

id3 事件 253, 261

id3 属性 (Sound 类) 261

ID3Info 类 252

IDInput 和 IDataOutput 接口 297

idleThreshold 属性 (NativeApplication 类)
287

iframe 元素 62, 64, 68, 94

激活窗口 122, 127

img 标签 (在 TextField 对象内容中) 93

Info.plist 文件 (Mac OS) 106

initialWindow 元素 (应用程序描述符文件
107, 116

innerHTML 属性 51, 66, 97

INSERT 语句 (SQL) 229

installApplication() 方法 (air.swf 文件) 310

installFolder 元素 (应用程序描述符文件) 106

INTEGER PRIMARY KEY 列 (SQL) 226

- invoke 事件 278
I
 InvokeEvent 类 109, 279
 arguments 属性 279
 currentDirectory 属性 279
 ioError 事件 253
 isBuffering 属性 (Sound 类) 256
 isDirectory 属性 (File 类) 173
 isHTTPS 属性 (BrowserInvokeEvent 类) 282
 JSON 63
 isSetAsDefaultApplication() 方法
 (NativeApplication 类) 286
 技术支持 9
 加密 268
 加密数据, 存储和检索 246
 加密数据的强绑定 246
 简单窗口 117
 剪切事件 194
 监视器
 请参阅屏幕
 剪贴板
 安全性 193
 数据格式 197, 198
 系统 192
 剪贴板事件 66
 降级攻击和安全 101

K
 Keyboard 类 140
 KeyChain (使加密数据与用户相关联) 246
 keyEquivalent 属性 (NativeMenuItem 类)
 140
 keyEquivalentModifiers 属性
 (NativeMenuItem 类) 140
 keystore 29, 32
 开始菜单 (Windows) 107
 可伸缩的矢量图形 (SVG) 62
 空闲时间 (用户) 287
 跨脚本访问 56, 98
 跨域缓存安全性 93
 扩展名 (文件), 与 AIR 应用程序关联 108, 279,
 286

L
 label 属性 (NativeMenuItem 类) 196
 lastInsertRowID 属性 (SQLResult 类) 225

M
 lastUserInput 属性 (NativeApplication 类)
 287
 length 属性 (ByteArray 类) 202
 lineEnding 属性 (File 类) 170
 listRootDirectories() 方法 (File 类) 164
 little-endian 字节顺序 202, 297
 垃圾桶 (删除文件) 174
 load 事件 48, 51, 62, 63, 121
 load() 方法 (FileReference 类) 181
 load() 方法 (Sound 类) 255
 load() 方法 (URLLoader 类) 294
 Loader 类 123
 loadString() 方法 (HTMLLoader 类) 98
 LocalConnection 类 306, 312
 allowInsecureDomain() 方法 303
 client 属性 302
 connectionName 参数 303
 send() 方法 302
 locationChange 事件 73
 连接到数据库 216
 列 (数据库) 211
 临时目录 171
 临时文件 174
 浏览
 选择目录 166
 选择文件 167
 浏览器调用功能 108, 282
 路径 (文件和目录) 167
 路径, 相对 168
 路径分隔符 (文件系统) 167

N
 Mac OS
 代理图标 120
 工具栏 120
 mainScreen 属性 (Screen 类) 134
 maximizable 元素 (应用程序描述符文件) 107
 maximize() 方法 (NativeWindow 类) 128
 maxSize 元素 (应用程序描述符文件) 107
 menuItemSelect 事件 140
 menuSelect 事件 140
 messageHandler 属性 (PDF) 249
 method 属性 (URLRequest 类) 294
 Microphone 类 252

Microsoft Windows
 标题栏图标 120
 Microsoft 验证码数字 ID (Microsoft
 Authenticode Digital ID) 313
 Microsoft 验证码证书 (Microsoft
 Authenticode Certificate) 314
MIME 类型
 HTML 复制和粘贴 65, 197
 HTML 拖放 187
 minimizable 元素 (应用程序描述符文件) 107
 minimize() 方法 (NativeWindow 类) 128
 minimumPatchLevel 属性 (应用程序描述符文
 件) 103
 minSize 元素 (应用程序描述符文件) 107
 mnemonicIndex 属性
 NativeMenuItem 类 140
 modificationDate 属性 (File 类) 173
 move 事件 117, 131
 moveTo() 方法
 File 类 173
 Window 对象 116
 moveToAsync() 方法 (File 类) 173
 moveToTrash() 方法 (File 类) 174
 moveToTrashAsync() 方法 (File 类) 174
 moving 事件 131
 mouseDown 事件 129
 MP3 文件 253, 261
 My Documents 目录 (Windows) 164
 麦克风
 访问 265
 检测活动 266
 传送到本地扬声器 265
 冒号 (:) 字符, 在 SQL 语句参数名称中 218
 枚举目录 171
 枚举屏幕 134
 密码
 加密媒体内容的设置 268
 命令, 菜单
 参阅菜单项
 命令行参数, 捕获 279
 命名参数 (在 SQL 语句中) 218
 目录 164, 171
 创建 171
 复制 171
 枚举 171
 删除 172, 174

- 移动 171
引用 164
应用程序调用 279
目录选择器对话框 166
- N**
- name 属性 (File 类) 173
name 元素 (应用程序描述符文件) 105
NativeApplication 类 68
 activeWindow 属性 126
 addEventListener() 方法 279
 applicationDescriptor 属性 285
 autoExit 属性 283
 exit() 方法 283
 getDefaultValue() 方法 286
 icon 属性 158
 id 属性 285
 idleThreshold 属性 287
 isSetAsDefaultApplication() 方法 286
 lastUserInput 属性 287
 publisherID 属性 285, 313
 removeAsDefaultApplication() 方法 286
 runtimePatchLevel 属性 286
 runtimeVersion 属性 286
 setAsDefaultApplication() 方法 108
 startAtLogin 属性 281
 supportsDockIcon 属性 158
 supportsMenu 属性 145
 supportsSystemTrayIcon 属性 158
NativeApplication.setAsDefaultApplication() 方法 286
NativeBoundsEvent 类 131
NativeMenu 类 139, 143
NativeMenuItem 类 139
 data 属性 141
 keyEquivalent 属性 140
 keyEquivalentModifiers 属性 140
 label 属性 196
 mnemonicIndex 属性 140
 submenu 属性 139
nativePath 属性 (File 类) 164, 173
NativeWindow 类 115
 activate 方法 126
 activate() 方法 122, 127
 addEventListener() 方法 131
- alwaysInFront 属性 126, 127
close() 方法 127
dispatchEvent() 方法 117
构造函数 122
JavaScript 访问 61
maximize() 方法 128
minimize() 方法 128
orderBehind() 方法 127
orderInBackOf() 方法 127
orderInFrontOf() 方法 127
orderToBack() 方法 127
orderToFront() 方法 127
restore() 方法 128
stage 属性 124
startMove() 方法 129
startResize() 方法 129
systemChrome 属性 117
systemMaxSize 属性 122
systemMinSize 属性 122
事件 131
实例化 125
transparent 属性 117, 118
type 属性 117
visible 属性 122, 126
- nativeWindow 属性
 Stage 类 126
 Window 对象 61, 67
nativeWindow 属性 (window 对象) 116, 121, 122
NativeWindowDisplayStateEvent 事件 131
NativeWindowInitOptions 类 121, 122
NetStream 类
 preloadEmbeddedMetadata() 方法 269
 resetDRMVouchers() 方法 272
 setDRMAuthenticationCredentials() 方法 268, 272
Netstream 类
 加密内容, 播放 268
NSHumanReadableCopyright 字段 (Mac OS) 106
内部检查器 (AIR HTML/JavaScript 应用程序内部检查器) 36
内存中数据库 214
- O**
- OID 列名 (SQL) 226
onclick 处理函数 51
ondominitialize 属性 69
onload 处理函数 96
onmouseover 处理函数 51
open 事件 253
open() 方法
 SQLConnection 类 213
 Window 对象 68, 98, 122
open() 方法 (SQLConnection 类) 214
openAsync() 方法 (SQLConnection 类) 213, 214, 216
opener 属性 (window 对象) 122
orderBehind() 方法 (NativeWindow 类) 127
orderInBackOf() 方法 (NativeWindow 类) 127
orderInFrontOf() 方法 (NativeWindow 类) 127
orderToBack() 方法 (NativeWindow 类) 127
orderToFront() 方法 (NativeWindow 类) 127
outerHTML 属性 66
- P**
- P12 文件 313
paintsDefaultBackground 属性
 (HTMLLoader 类) 118, 124
pan 属性 (SoundTransform 类) 260
parameters 属性 (SQLStatement 类) 217, 218
parent 属性 (File 类) 173
parent 属性 (window 对象) 122
parentSandboxBridge 属性
 Window 对象 68, 95
parentSandboxBridge 属性 (Window 对象) 58
PDF
 支持 62, 248
PDF 内容
 JavaScript 通信 249
 加载 249
 添加到 AIR 应用程序 248
 已知限制 250
pdfCapability 属性 (HTMLLoader 类) 248
PFX 文件 313
placeLoadStringContentInApplicationSandbox 属性 (HTMLLoader 类) 98

play() 方法 (Sound 类) 258
 position 属性 (ByteArray 类) 201
 position 属性 (SoundChannel 类) 258, 259
 postMessage() 方法 (PDF 对象) 250
 preloadEmbeddedMetadata() 方法
 (NetStream 类) 269
 preventDefault() 方法 74
 print() 方法 (Window 对象) 62
 Program Files 目录 (Windows) 305
 programMenuFolder 元素 (应用程序描述符文件)
) 107
 progress 事件 253
 publisherID 属性 (NativeApplication 类) 285,
 313
 publisherid 文件 285
 凭据
 针对 DRM 加密内容 274
 屏幕 133
 窗口, 移动 134
 枚举 134
 主 134
 凭证, 与 DRM 加密的内容一起使用 268
 普通窗口 117

Q
 启动 (系统), 启动 AIR 应用程序 281
 启动 AIR 应用程序 278
 签名
 迁移 31, 315
 嵌入对象 (采用 HTML) 62
 迁移签名 31, 315
 清除目录 172
 区域设置, 为应用程序选择 336
 全屏窗口 132

R
 readBytes() 方法 (ByteArray 类) 201
 readFloat() 方法 (ByteArray 类) 201
 readInt() 方法 (ByteArray 类) 201
 readObject() 方法 (ByteArray 类) 201
 readUTFBytes() 方法 (ByteArray 类) 201
 RegExp 对象, 在 ActionScript 和 JavaScript 之间转换 55
 relativize() 方法 (File 类) 168
 removeAsDefaultApplication() 方法
 (NativeApplication 类) 286
 removeEventListener() 方法 77
 resetDRMVouchers() 方法 (NetStream 类)
) 272
 resizable 元素 (应用程序描述符文件) 107
 resize 事件 117, 131
 resizing 事件 131
 resolvePath() 方法 (File 类) 164
 Responder 类 217, 225
 restore() 方法 (NativeWindow 类) 128
 result 事件 217
 ROWID 列名 (SQL) 226
 ROWID 列名 (SQL) 226
 runtime 属性 (Window 对象) 52, 61, 67, 121,
 122
 runtimePatchLevel 属性 (NativeApplication
 类) 286
 runtimeVersion 属性 (NativeApplication
 类) 286
 任务栏图标 126, 158

S
 sandboxRoot 属性
 frame 94
 iframe 94
 sandboxRoot 属性 (frame 和 iframe 元素) 57,
 62, 64, 68
 sandboxType 属性
 BrowserInvokeEvent 类 282
 Security 类 286
 save() 方法 (FileReference 类) 181
 scaleMode 属性
 Stage 类 129
 Screen 类 133
 getScreenForRectangle() 方法 134
 mainScreen 属性 134
 screens 属性 134
 screens 属性 (Screen 类) 134
 screenX 属性 (HTML 拖动事件) 186
 screenY 属性 (HTML 拖动事件) 186
 script 标签 36, 51, 52, 54, 63, 66, 255
 src 属性 97
 scroll 事件 73
 Security 类
 sandboxType 属性 286
 securityDomain 属性 (BrowserInvokeEvent
 类) 282
 select 事件 139, 144, 145
 SELECT 语句 (SQL) 219, 229
 send() 方法 (LocalConnection 类) 302
 separator 属性 (File 类) 170
 setAsDefaultApplication() 方法
 (NativeApplication 类) 108, 286
 setData() 方法
 ClipboardData 对象 65
 Clipboard 方法 199
 DataTransfer 对象 66, 186, 188
 setDataHandler() 方法 (Clipboard 类) 199
 setDragImage() 方法 (属于 HTML 拖动事件的
 dataTransfer 属性) 186
 setDRMAuthenticationCredentials() 方法
 (NetStream 类) 268, 272
 setInterval() 函数 50, 68, 97
 setTimeout() 函数 50, 68, 97
 Shift 键 140
 size 属性 (File 类) 173
 Socket 类 297
 Sound 类 252, 258
 SoundChannel 类 252, 260
 soundComplete 事件 259
 SoundLoaderContext 类 252
 SoundMixer 类 252, 256
 SoundTransform 类 252
 soundTransform 属性 (SoundChannel
 类) 260
 Source Viewer 331
 spaceAvailable 属性 (File 类) 170
 SQL
 AUTOINCREMENT 列 226
 CREATE TABLE 语句 214
 DELETE 语句 226
 关于 212
 INSERT 语句 229
 INTEGER PRIMARY KEY 列 226
 命名参数 (在语句中) 218
 OID 列名 226
 ROWID 列名 226
 ROWID 列名 226
 SELECT 语句 219, 229
 数据类型指定 219, 229
 UPDATE 语句 226
 未命名参数 (在语句中) 218

语句 216
 语句中的参数 218
SQLConnection 类
 attach() 方法 227
 open 方法 214
 open() 方法 213
 openAsync() 方法 213, 214, 216
sqlConnection 属性 (SQLStatement 类) 217
SQLError 类 217
SQLErrorEvent 类 217
SQLite 数据库支持 210
 请参阅数据库
SQLMode 类 216
SQLResult 类 225
SQLStatement 类 216
 execute 方法 217
 execute() 方法 220, 225
 getResult() 方法 225
 parameters 对象 217
 parameters 属性 218
 sqlConnection 属性 217
 text 属性 217, 218, 220, 226
Stage 类
 addChild() 方法 124
 addChildAt() 方法 124
 displayState 属性 132
 nativeWindow 属性 126
 scaleMode 属性 122, 129
stage 属性
 NativeWindow 类 124
StageDisplayState 类 132
StageScaleMode 类 122, 129
startAtLogin 属性 (NativeApplication 类) 281
startMove() 方法 (NativeWindow 类) 129
startResize() 方法 (NativeWindow 类) 129
status 事件 266
StatusEvent 类 268
stop() 方法 (SoundChannel 类) 258
styleSheets 属性 (Document 对象) 56
subErrorID 属性 (DRMErrorEvent 类) 274
submenu 属性
 NativeMenuItem 类 139
SWF 内容
 采用 HTML 62
 在 HTML 上方叠加 123
SWF 文件
 嵌入的声音 255
 通过 script 标签加载 54
 与 AIR 应用程序通信 302
 Sun Java 签名数字 ID (Sun Java Signing Digital ID) 313
 supportsDockIcon 属性 (NativeApplication 类) 158
 supportsMenu 属性 (NativeApplication 类) 145
 supportsSystemTrayIcon 属性 (NativeApplication 类) 158
 systemChrome 属性 (NativeWindow 类) 117
 systemMaxSize 属性 (NativeWindow 类) 122
 systemMinSize 属性 (NativeWindow 类) 122
 沙箱 56, 63, 92, 286
 沙箱桥 45, 48, 57, 62, 63, 95, 98
 删除目录 172, 174
 删除文件 174
 上下文菜单 137
 HTML 142
 声音
 播放 258
 从 SWF 文件加载 255
 基础知识 252
 加载 MP3 文件 253
 加载进度 254
 类 252
 流 256
 事件 253
 数据 262
 向服务器发送和从中接收 267
 事件
 AIR 运行时 73
 本机窗口 117
 菜单 139, 143
 处理函数 76
 HTML 73
 流 74
 默认行为 74
 NativeWindow 类 131
 侦听器 76
 时间戳 314
 视频内容加密 268
 实用程序窗口 117
 受信任的本地沙箱 63, 92
数据
 发送到服务器 296
 加载外部 293
 数据格式, 剪贴板 197
 数据加密 246
数据库
 安全性 219
 表 211, 214
 创建 214
 错误 227
 多个, 使用 227
 更改数据 226
 关于 211
 检索数据 219
 结构 211
 连接 216
 列 211
 内存中 214
 删除数据 226
 数据类型指定 219, 229
 同步模式 213
 文件 211
 行 211
 行标识符 226
 性能 219
 异步模式 213
 用途 211
 主键 225, 226
 字段 211
 数据类型, 数据库 229
 数据验证, 应用程序调用 282
 数字签名 24, 29, 312
 数字权限管理 268
 私钥 29
T
text 属性 (SQLStatement 类) 217, 218, 220, 226
TextField 类
 复制和粘贴 193
 img 标签 93
 Thawte 证书 312, 313
title 元素 (应用程序描述符文件) 107
transparent 属性 (NativeWindow 类) 117, 118

transparent 元素 (应用程序描述符文件) 107

type 属性 (Event 类) 74

type 属性 (File 类) 173

type 属性 (NativeWindow 类) 117

types 属性

- DataTransfer 对象 66
- HTML 复制和粘贴事件 194
- HTML 拖动事件 186
- types** 属性 (DataTransfer 对象) 189

套接字服务器 298

套接字连接 297

停靠栏菜单 138

停靠栏图标 158

- 菜单 141
- 窗口最小化 126
- 回弹 159
- 支持 158

同步编程

- 数据库 213, 216, 229
- 文件系统 162
- XMLHttpRequests 51

透明窗口 107, 118

图标

- 动画 158
- 任务栏 126, 158
- 删除 158
- 停靠栏 158
- 图像 158
- 系统任务栏 158
- 应用程序 108

退出 AIR 应用程序 278

拖出动作 185

拖放

- 到非应用程序沙箱内容 (采用 HTML 格式) 191
- 动作 185
- 光标效果 187

HTML 66

HTML 中的默认行为 185

HTML 中的事件 186

相关类 185

传输格式 185

拖入动作 185

W

Web 浏览器

- 安装 AIR 应用程序 310
- 从中启动 AIR 应用程序 282
- 检测 AIR 运行时 309
- 检测是否已安装 AIR 应用程序 310
- 启动 AIR 应用程序 311

WebKit 62, 71

- webkit-border-horizontal-spacing CSS 属性 71
- webkit-border-vertical-spacing CSS 属性 71
- webkit-line-break CSS 属性 71
- webkit-margin-bottom-collapse CSS 属性 71
- webkit-margin-collapse CSS 属性 71
- webkit-margin-start CSS 属性 71
- webkit-margin-top-collapse CSS 属性 72
- webkit-nbsp-mode CSS 属性 72
- webkit-padding-start CSS 属性 72
- webkit-rtl-ordering CSS 属性 72
- webkit-text-fill-color CSS 属性 72
- webkit-text-security CSS 属性 72
- webkit-user-drag CSS 属性 72, 186, 188
- webkit-user-modify CSS 属性 72
- webkit-user-select CSS 属性 72, 186, 188

VeriSign 证书 314

Verisign 证书 312, 313

version 元素 (应用程序描述符文件) 105

width 元素 (应用程序描述符文件) 107

Window 对象

- childSandboxBridge 属性 95
- close() 方法 116
- htmlLoader 对象 61
- htmlLoader 属性 67, 121, 122
- moveTo() 方法 116
- nativeWindow 对象 61
- nativeWindow 属性 67, 116, 121, 122
- open 方法 68
- open() 方法 98, 122
- opener 属性 122
- parent 属性 122
- parentSandboxBridge 属性 58, 68, 95
- print() 方法 62
- runtime 属性 52, 61, 67, 93, 97, 121, 122

Window 类 115

WindowedApplication 类 115

Windows 注册表设置 91

visible 属性

- NativeWindow 类 122, 126

visible 元素 (应用程序描述符文件) 107

uncaughtScriptException 事件 73

uncompress() 方法 (ByteArray 类) 203

unload 事件 67

UntrustedAppInstallDisabled (Windows 注册表设置) 91

UPDATE 语句 (SQL) 226

update() 方法 (Updater 类) 318

UpdateDisabled (Windows 注册表设置) 91

Updater 类 318

- write() 方法 (Document 对象) 51, 66
- writeBytes() 方法 (ByteArray 类) 201
- writeFloat() 方法 (ByteArray 类) 201
- writeInt() 方法 (ByteArray 类) 201
- writeln() 方法 (Document 对象) 51, 66
- writeObject() 方法 (ByteArray 类) 201
- writeUTFBytes() 方法 (ByteArray 类) 201

URL 53

- 拖放支持 185, 187

URL 编码 294

URL 方案 168

url 属性

- File 类 164, 173

url 属性 (File 类) 164

URLLoader 类

- dataFormat 属性 296
- 构造函数 294
- 关于 293
- load() 方法 294

URLLoaderDataFormat 类 296

URLRequest 类 294

- contentType 属性 293
- data 属性 294
- method 属性 294

URLRequestMethod 类 294

URLStream 类 64

URLVariables 类 293

- decode() 方法 294

userDirectory 属性 (File 类) 164

userIdle 事件 287

userPresent 事件 287

外部数据, 加载 293

网络
 概念和术语 290
 关于 290
 网络字节顺序 297
 未命名参数 (在 SQL 语句中) 218
 位图
 拖放支持 185, 187
 位图图像, 为图标设置 158
 未知发行商名称 (在 AIR 应用程序的安装程序中)
) 312
 文本
 拖放支持 185, 187
 文档, 相关 9
 文档目录 164
 问号 (?) 字符, 在未命名 SQL 参数中 218
 文件
 读取 175
 复制 173
 删除 174
 数据库 211
 拖放支持 185
 写入 175
 移动 173
 引用 166
 文件 API 162
 文件类型关联 108, 279, 286
 文件列表
 拖放支持 187
 文件系统
 安全 99
 文件系统 API 162
 文件选择器对话框 167

X
 x 元素 (应用程序描述符文件) 107
 XML
 类 52
 套接字服务器 298
 XML 命名空间 (应用程序描述符文件) 103
 XMLHttpRequest 对象 45, 51, 63, 68, 97
 XMLList 类 52
 xmlns (应用程序描述符文件) 103
 XMLSocket 类 297, 298
 系统登录, 启动 AIR 应用程序 281
 系统任务栏图标 138, 141
 支持 158

系统镶边 118
 HTML 窗口 122
 显示窗口 126
 显示器
 请参阅屏幕
 显示顺序, 窗口 126, 127
 现有 AIR 应用程序 278
 相对路径 (文件之间) 168
 写入文件 175
 卸载
 AIR 应用程序 91
 AIR 运行时 2
 行 (数据库) 211, 225
 修补级别
 AIR 运行时 286
 修补级别, AIR 运行时 103
 序列化对象
 复制和粘贴支持 192

Y
 y 元素 (应用程序描述符文件) 107
 压缩数据 203
 延迟呈现 (复制和粘贴) 199
 扬声器和麦克风 265
 样式表, HTML
 在 ActionScript 中操作 56
 要求
 PDF 呈现 248
 异步编程
 数据库 213, 216, 229
 文件系统 162
 XMLHttpRequest 51
 移动窗口 117, 129
 移动目录 171
 移动文件 173
 已启用菜单项 141
 已选中菜单项 141
 隐藏窗口 126
 音频
 请参阅声音
 应用程序
 请参阅 AIR 应用程序
 应用程序 ID 104
 应用程序菜单 137, 145
 创建 141
 应用程序存储目录 53, 91, 164, 168

应用程序描述符文件 102
 读取 285
 应用程序目录 164
 应用程序沙箱 36, 47, 48, 51, 57, 62, 63, 92, 286
 用户活动, 检测 287
 用户名
 加密媒体内容的设置 268
 用户凭据和安全 101
 语句, SQL 216
 远程沙箱 63, 92
 源代码, 查看 331
 运行 AIR 应用程序 305, 311

Z
 ZIP 文件格式 206
 ZLIB 压缩 203
 在应用程序之间通信 302
 证书
 ADT 命令行选项 29
 颁发机构 (CA) 101
 代码签名 101
 对 AIR 文件进行签名 312
 格式 313
 更改 31, 315
 过期 314
 链 316
 迁移 31, 315
 证书颁发机构 (CA) 312
 证书吊销列表 (CRL) 316
 证书实行声明 (CPS) 316
 只能与本地文件系统内容交互的沙箱 63, 92
 只能与远程内容交互的沙箱 92
 注册文件类型 286
 助记键字符
 菜单项 140
 主键
 菜单项 140
 数据库 225
 主目录 164
 主屏幕 134
 桌面窗口
 请参阅窗口
 桌面目录 164
 子菜单 139, 142
 自定义更新用户界面 319

自定义镶边 118
自动启动 (登录时启动 AIR 应用程序) 281
字段 (数据库) 211
字节数组
 大小 202
 位置 201
 字节顺序 202
字节顺序 202, 297
自签名证书 32, 101, 312
最大化窗口 107, 117, 128
最小化窗口 107, 117, 126, 128