ADOBE® AIR® HTML 开发人员指南

法律声明

有关法律声明，请参阅 http://help.adobe.com/zh_CN/legalnotices/index.html。

目录

第 1 章 : 关于 HTML 环境

HTML 环境概述 2

AIR 和 WebKit 4

第 2 章 : 在 AIR 中进行 HTML 和 JavaScript 编程

创建基于 HTML 的 AIR 应用程序 18

示例应用程序和安全启示 18

避免与安全相关的 JavaScript 错误 20

通过 JavaScript 访问 AIR API 类 24

关于 AIR 中的 URL 25

在 HTML 中嵌入 SWF 内容 26

在 HTML 页中使用 ActionScript 库 28

转换 Date 和 RegExp 对象 29

跨脚本访问不同安全沙箱中的内容 30

第 3 章 : 处理 AIR 中与 HTML 相关的事件

HTMLLoader 事件 34

AIR 类 - 事件处理与 HTML DOM 中其他事件处理的不同之处 34

Adobe AIR 事件对象 36

使用 JavaScript 处理运行时事件 37

第 4 章 : 为 AIR HTML 容器编写脚本

HTMLLoader 对象的显示属性 41

访问 HTML 历史记录列表 43

设置在加载 HTML 内容时使用的用户代理 44

设置用于 HTML 内容的字符编码 44

为 HTML 内容定义类似于浏览器的用户界面 45

第 5 章 : 使用矢量

矢量基础知识 56

创建矢量 57

向矢量中插入元素 57

检索值和删除矢量元素 58

Vector 对象的属性和方法 58

示例：使用需要矢量的 AIR API 58

第 6 章 : AIR 安全性

AIR 安全性基础知识 60

安装和更新 60

Adobe AIR 中的 HTML 安全性 63

通过脚本访问不同域中的内容 68
写入磁盘 69
安全使用不受信任的内容 70
开发人员的最佳安全做法 71
代码签名 72

第 7 章 : 使用 AIR 本机窗口
AIR 中的本机窗口的基础知识 73
创建窗口 78
管理窗口 84
侦听窗口事件 91
显示全屏窗口 92

第 8 章 : AIR 中的显示屏幕
AIR 中的显示屏幕的基础知识 95
枚举屏幕 96

第 9 章 : 使用菜单
菜单基础知识 99
创建本机菜单 (AIR) 103
关于 HTML 中的上下文菜单 (AIR) 105
显示弹出本机菜单 (AIR) 106
处理菜单事件 106
本机菜单示例：窗口和应用程序菜单 (AIR) 108
使用 MenuBuilder 框架 110

第 10 章 : AIR 中的任务栏图标
关于任务栏图标 122
停靠栏图标 122
系统任务栏图标 123
Window 任务栏图标和按钮 125

第 11 章 : 使用文件系统
使用 AIR 文件系统 API 126

第 12 章 : AIR 中的拖放
HTML 中的拖放 153
将数据拖出 HTML 元素 156
将数据拖入 HTML 元素 157
示例：覆盖默认的 HTML 拖入行为 157
在非应用程序 HTML 沙箱中处理文件放置 159
放置文件释放 160

第 13 章 : 复制和粘贴
复制粘贴基础知识 168
读取和写入系统剪贴板 168
AIR 中的 HTML 复制和粘贴 169
剪贴板数据格式 171

第 14 章 : 在 AIR 中使用本地 SQL 数据库
关于本地 SQL 数据库 176
创建和修改数据库 179
操作 SQL 数据库数据 182
使用同步和异步数据库操作 198
对 SQL 数据库使用加密 202
使用 SQL 数据库的策略 216

第 15 章 : 加密的本地存储区
将数据添加到加密本地存储区 219
访问加密的本地存储区中的数据 219
从加密的本地存储区中删除数据 220

第 16 章 : 使用字节数组
读取并写入 ByteArray 221
ByteArray 示例：读取 .zip 文件 226

第 17 章 : 在 AIR 中添加 PDF 内容
检测 PDF 功能 231
加载 PDF 内容 232
编写 PDF 内容的脚本 232
对 AIR 中的 PDF 内容的已知限制 234

第 18 章 : 处理声音
声音处理基础知识 235
了解声音体系结构 235
加载外部声音文件 236
处理嵌入的声音 238
处理声音流文件 239
处理动态生成的音频 239
播放声音 241
处理声音元数据 244
访问原始声音数据 245
捕获声音输入 248

第 19 章 : 客户端系统环境
客户端系统环境基础知识 251
使用 System 类 252
使用 Capabilities 类 252

第 20 章 : AIR 应用程序的调用和终止
应用程序调用 254
捕获命令行参数 255
用户登录时调用 AIR 应用程序 257
从浏览器调用 AIR 应用程序 258
应用程序终止 259

第 21 章 : 处理 AIR 运行时和操作系统信息
管理文件关联 261
获取运行时版本和修补级别 261
检测 AIR 功能 262
跟踪用户当前状态 262

第 22 章 : 套接字
TCP 套接字 263
UDP 套接字 (AIR) 268
IPv6 地址 269

第 23 章 : HTTP 通信
加载外部数据 271
Web 服务请求 277
在其他应用程序中打开 URL 282
向服务器发送 URL 284

第 24 章 : 与其他 Flash Player 和 AIR 实例通信
关于 LocalConnection 类 285
在两个应用程序之间发送消息 285
连接到不同域中的内容和 AIR 应用程序 286

第 25 章 : 针对 JavaScript 开发人员的 ActionScript 基础知识
ActionScript 和 JavaScript 之间的差异：概述 288
ActionScript 3.0 数据类型 289
ActionScript 3.0 类、包和命名空间 290
ActionScript 3.0 函数中的必需参数和默认值 291
ActionScript 3.0 事件侦听器 292

第 26 章 : 本地数据库中的 SQL 支持
支持的 SQL 语法 293
数据类型支持 310

第 27 章 : SQL 错误详细消息、 ID 和参数

第 1 章：关于 HTML 环境

Adobe AIR 1.0 和更高版本

AIR 和 Safari Web 浏览器均使用 WebKit (www.webkit.org) 分析、布局和呈现 HTML 和 JavaScript 内容。AIR 的内置主机类和对象为传统上与桌面应用程序关联的功能提供了一个 API。这些功能包括读取和写入文件以及管理窗口。 Adobe AIR 还继承了 Adobe® Flash® Player 的 API，其中包括声音和二进制套接字等功能。

重要说明: Adobe AIR 运行时的新版本可能包含 WebKit 的更新版本。AIR 新版本中的 WebKit 更新可能会对已部署的 AIR 应用程序造成意外更改。这些更改可能会影响应用程序中 HTML 内容的行为或外观。例如， WebKit 呈现中的改进或更正可能会更改应用程序用户界面中元素的布局。为此，我们强烈建议您在应用程序中提供一个更新机制。如果因 AIR 中包含的 WebKit 版本发生更改而需要更新应用程序， AIR 更新机制可提示用户安装应用程序的新版本。

下表列出了所使用的 WebKit 版本与 AIR 中使用的 WebKit 版本相同的 Safari Web 浏览器版本：

AIR 版本	Safari 版本
1.0	2.04
1.1	3.04
1.5	4.0 测试版
2.0	4.03
2.5	4.03
2.6	4.03
2.7	4.03
3	5.0.3

您始终可以通过检查由 HTMLLoader 对象返回的默认用户代理字符串来确定 WebKit 的已安装版本：

```
air.trace( window.htmlLoader.userAgent );
```

请记住，AIR 中使用的 WebKit 版本与开放源版本不同。AIR 中不支持某些功能，并且 AIR 版本可以包括在相应 WebKit 版本中尚不可用的安全性和错误修复功能。请参阅 第 13 页的“AIR 中不支持的 WebKit 功能 ”。

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镶边，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，请参阅第 42 页的“ 在 HTML 页中加载 SWF 或 PDF 内容时的注意事项 ” 和第 76 页的“ 窗口透明度 ”。

更多帮助主题

Webkit DOM 参考

Safari HTML 参考

Safari CSS 参考

www.webkit.org

HTML 环境概述

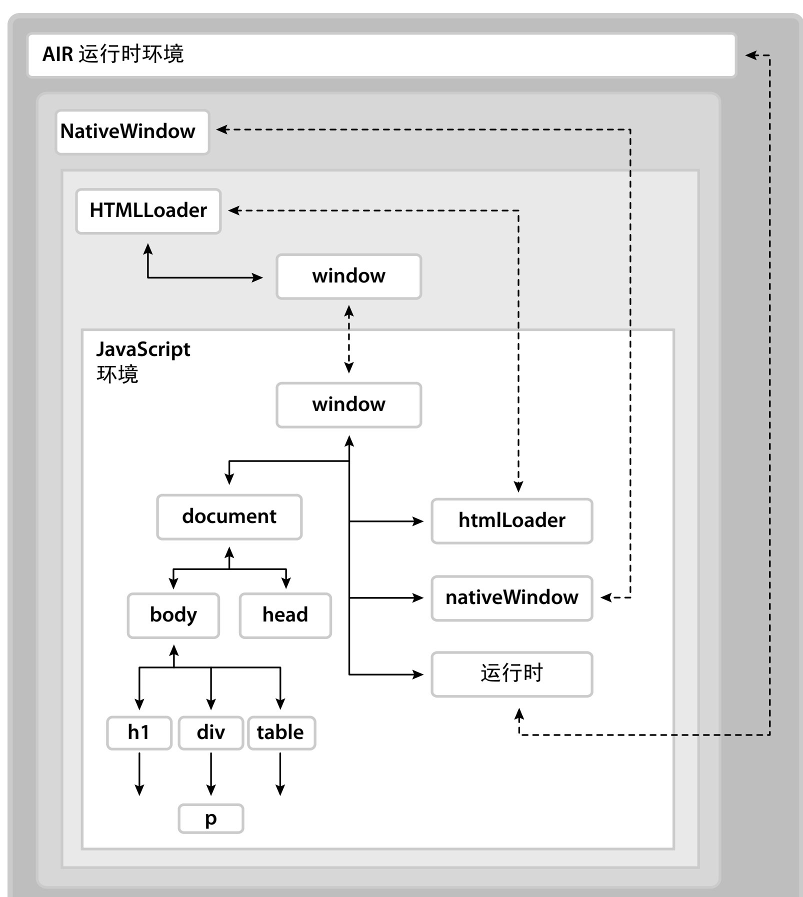
Adobe AIR 1.0 和更高版本

Adobe AIR 使用 HTML 渲染器、文档对象模型和 JavaScript 解释程序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 主机之间的关系

Adobe AIR 1.0 和更高版本

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 应用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，加载到 frame 中的文档可以使用 parent.runtime 访问其父级的 runtime 属性。）

关于安全性

Adobe AIR 1.0 和更高版本

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 63 页的 “Adobe AIR 中的 HTML 安全性 ” 以及 第 10 页的 “HTML frame 和 iframe 元素 ”。

关于插件和嵌入对象

Adobe AIR 1.0 和更高版本

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。

更多帮助主题

第 63 页的 “Adobe AIR 中的 HTML 安全性 ”

第 4 页的 “HTML 沙箱 ”

第 10 页的 “HTML frame 和 iframe 元素 ”

第 8 页的 “JavaScript Window 对象 ”

第 5 页的 “XMLHttpRequest 对象 ”

第 231 页的 “ 在 AIR 中添加 PDF 内容 ”

AIR 和 WebKit

Adobe AIR 1.0 和更高版本

Adobe AIR 中使用开放源代码 WebKit 引擎，该引擎也用于 Safari Web 浏览器。 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 的其他文档，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）。

AIR 中的 JavaScript

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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 20 页的 " 避免与安全相关的 JavaScript 错误 "。

HTML 沙箱

Adobe AIR 1.0 和更高版本

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，请参阅安全性沙箱。

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。只能将从应用程序目录 （app: URL 方案引用的安装目录）加载的内容放置到应用程序沙箱中。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明确公开的对象、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，您仍然必须避免明确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

Adobe AIR 1.0 和更高版本

完成加载页之后，将在应用程序沙箱中限制使用 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 65 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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"
</iframe>
```

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"
</iframe>
```

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"
</iframe>
```

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 10 页的 “HTML frame 和 iframe 元素 ” 以及 第 63 页的 “Adobe AIR 中的 HTML 安全性 ”。

Cookie

Adobe AIR 1.0 和更高版本

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，还提供了存储永久性数据的其他方法，包括 EncryptedLocalStore、SharedObject 和 FileStream 类。

Clipboard 对象

Adobe AIR 1.0 和更高版本

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
clearData(mimeType)	清除剪贴板数据。将 mimeType 参数设置为要清除的数据的 MIME 类型。
getData(mimeType)	获取剪贴板数据。该方法只能在 paste 事件的处理函数中调用。将 mimeType 参数设置为要返回的数据的 MIME 类型。
setData(mimeType, data)	将数据复制到剪贴板。将 mimeType 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 168 页的 “ 复制和粘贴 ” 以及 Using the Pasteboard from JavaScript（Apple 开发人员中心）。

拖放

Adobe AIR 1.0 和更高版本

进出 HTML 的拖放手势生成下列 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 dragstart 事件的处理函数中的 effectsAllowed 属性。
dropEffect	指定拖动目标支持哪些允许的拖动效果。设置 dragEnter 事件的处理函数中的 dropEffect 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 153 页的 “AIR 中的拖放 ” 和 Using the Drag-and-Drop from JavaScript （Apple 开发人员中心）。

innerHTML 和 outerHTML 属性

Adobe AIR 1.0 和更高版本

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，则不会添加事件侦听器。同样，不会计算嵌入的 <script> 标签。有关详细信息，请参阅 第 63 页的 “Adobe AIR 中的 HTML 安全性 ”。

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

Adobe AIR 1.0 和更高版本

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，在完成页加载之后调用 document.write() 或 writeln() 将清除当前页或打开一个新的空白页，这与在大多数 Web 浏览器中相同。

Document.designMode 属性

Adobe AIR 1.0 和更高版本

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 12 页的“HTML contentEditable 属性”。

unload 事件 （对于 body 和 frameset 对象）

Adobe AIR 1.0 和更高版本

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，以下 JavaScript 代码将在用户关闭应用程序时显示 ("Goodbye.") 消息：

```
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

Adobe AIR 1.0 和更高版本

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 24 页的 “ 使用 AIRAliases.js 文件 ”。

Window.nativeWindow 属性 nativeWindow 属性提供对基础本机 Window 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 的引用来访问 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 30 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，您不能定义作为字符串执行的代码。您必须使用函数引用。有关详细信息，请参阅第 22 页的“setTimeout() 和 setInterval()”。

Window.open() 函数 当在非应用程序沙箱中运行的代码调用 open() 方法时，该方法在调用时仅打开一个窗口，以作为用户交互（例如鼠标单击或按键）的结果。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 68 页的“调用 JavaScript window.open() 方法的限制”。

air.NativeApplication 对象

Adobe AIR 1.0 和更高版本

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。有关 NativeApplication 对象的详细信息，请参阅第 261 页的“处理 AIR 运行时和操作系统信息”。

JavaScript URL 方案

Adobe AIR 1.0 和更高版本

在应用程序沙箱内部阻止执行在 JavaScript URL 方案（如在 href="javascript:alert('Test')" 中）定义的代码。不引发错误。

AIR 中的 HTML

Adobe AIR 1.0 和更高版本

AIR 和 WebKit 定义了几个非标准 HTML 元素和属性，包括：

第 10 页的 "HTML frame 和 iframe 元素 "

第 11 页的 "HTML 元素事件处理函数 "

HTML frame 和 iframe 元素

Adobe AIR 1.0 和更高版本

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 在帧开始标签中包含 allowCrossDomainXHR="allowCrossDomainXHR"，以便允许该帧中的内容对任何远程域执行 XMLHttpRequest 操作。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 涉及严重的安全影响。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" 仍将启用跨域 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 www.example.com 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/"/>
```

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 dominitialize 事件建立沙箱桥，并将 child.html 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 30 页的 “ 跨脚本访问不同安全沙箱中的内容 ” 和 第 63 页的 “Adobe AIR 中的 HTML 安全性 ”。

HTML 元素事件处理函数

Adobe AIR 1.0 和更高版本

AIR 和 WebKit 中的 DOM 对象调度未包含在标准 DOM 事件模型中的某些事件。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	当拖动手势进入元素范围时调用。
ondragleave	当拖动手势离开元素范围时调用。
ondragover	当拖动手势位于元素范围内部时持续调用。
ondragstart	当拖动手势开始时调用。
ondrop	当在元素上释放拖动手势时调用。
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。
onselectstart	当开始选择时调用。

HTML contentEditable 属性

Adobe AIR 1.0 和更高版本

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 8 页的 “Document.designMode 属性 ”。

Data: URL

Adobe AIR 2 和更高版本

AIR 支持下列元素的 data: URL：

- img

- input type=”image”
- CSS 规则，允许使用图像 （例如，背景图像）

数据 URL 允许您将二进制图像数据作为 Base64 编码的字符串直接插入到 CSS 或 HTML 文档。以下示例使用 data: URL 作为重复的背景：

```
<html>
<head>
<style>
body {
background-
image:url('data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAGQAAABkCAMAAABHPGVmAAAAGXRFWHRTb2Z0d2FyZQBBZG9i
ZSBJbWFnZVJlYWR5ccllPAAAAAZQTFRF%2F6cA%2F%2F%2F%2Fgxp3lwAAAAJ0Uk5T%2FwDltzBKAAABF0lEQVR42uzZQQ7CMAxE0e%2F
7X5oNCyRocWzPiJbMBZ6qpIljE%2BnwklgKG7kwUjc2IkIaxkY0CPdEsCCasws6ShXBgmBBmEagpXQQLAgWBAuSY2gaKaWPYEGwIEwg0F
RmECwIFoQeQjJlhJWUEFazjFDJCkI5WYRWMgjtfEGYyQnCXD4jTCdm1zmngFpBFznwVNi5RPSbwbWnpYr%2BBHi%2FtCTfgPLEPL7jBct
AKBRptXJ8M%2BprIuZKu%2BUKcg4YK1PLz7kx4bSqHyPaT4d%2B28OCJJiRBo4FCQsSA0bziT3XubMgYUG6fc5fatmGBQkL0hoJ1IaZMi
QsSFiQ8vRscTjlQOI2iHZwtpHuf%2BJAYiOiJSkj8Z%2FIQ4ABANvXGLd3%2BZMrAAAAAElFTkSuQmCC');
background-repeat:repeat;
}
</style>
</head>
<body>
</body>
</html>
```

使用 data: URL 时，一定要注意多余的空格。例如，数据字符串必须作为单一、完整的行输入。否则，换行符将被视为数据的一部分，并且无法解码图像。

AIR 中的 CSS

Adobe AIR 1.0 和更高版本

WebKit 支持多个扩展 CSS 属性。其中，许多扩展名使用以下前缀：-webkit。请注意，其中某些扩展名本质上是实验性的，并可能从 WebKit 的将来版本中删除。有关 CSS 支持的 Webkit 和适用于 CSS 的 Webkit 扩展名的详细信息，请参阅 Safari CSS 参考。

AIR 中不支持的 WebKit 功能

Adobe AIR 1.0 和更高版本

AIR 不支持 WebKit 或 Safari 4 中提供的下列功能：

- 通过 window.postMessage 进行跨域消息传递 （AIR 提供自己的跨域通信 API）
- CSS 变量
- Web 开放字体格式 (WOFF) 和 SVG 字体。
- HTML 视频和音频标签
- 媒体设备查询
- 脱机应用程序缓存
- 打印 （AIR 提供自己的 PrintJob API）
- 拼写和语法检查
- SVG
- WAI-ARIA

- WebSocket（AIR 提供自己的 Socket API）
- Web workers
- WebKit SQL API（AIR 提供自己的 API）
- WebKit geolocation API（在受支持的设备中，AIR 提供自己的 geolocation API）
- WebKit 多文件上传 API
- WebKit 触摸事件（AIR 提供自己的触摸事件）
- 无线标记语言 (WML)

下面的列表包含 AIR 不支持的特定 JavaScript API、HTML 元素以及 CSS 属性和值：

不支持的 **JavaScript Window** 对象成员：
- applicationCache()
- console
- openDatabase()
- postMessage()
- document.print()

不支持的 **HTML** 标签：
- audio
- video

不支持的 **HTML** 属性：
- aria-*
- draggable
- formnovalidate
- list
- novalidate
- onbeforeload
- onhashchange
- onorientationchange
- onpagehide
- onpageshow
- onpopstate
- ontouchstart
- ontouchmove
- ontouchend
- ontouchcancel
- onwebkitbeginfullscreen
- onwebkitendfullscreen
- pattern

- required
- sandbox

不支持的 **JavaScript** 事件：
- beforeload
- hashchange
- orientationchange
- pagehide
- pageshow
- popstate
- touchstart
- touchmove
- touchend
- touchcancel
- webkitbeginfullscreen
- webkitendfullscreen

不支持的 **CSS** 属性：
- background-clip
- background-origin （使用 -webkit-background-origin）
- background-repeat-x
- background-repeat-y
- background-size （使用 -webkit-background-size）
- border-bottom-left-radius
- border-bottom-right-radius
- border-radius
- border-top-left-radius
- border-top-right-radius
- text-rendering
- -webkit-animation-play-state
- -webkit-background-clip
- -webkit-color-correction
- -webkit-font-smoothing

不支持的 **CSS** 值：
- 外观属性值：
  - media-volume-slider-container

  - media-volume-slider
  - media-volume-sliderthumb
  - outer-spin-button
- border-box （background-clip 和 background-origin）
- contain (background-size)
- content-box （background-clip 和 background-origin）
- cover (background-size)
- 列表属性值:
  - afar
  - amharic
  - amharic-abegede
  - cjk-earthly-branch
  - cjk-heavenly-stem
  - ethiopic
  - ethiopic-abegede
  - ethiopic-abegede-am-et
  - ethiopic-abegede-gez
  - ethiopic-abegede-ti-er
  - ethiopic-abegede-ti-et
  - ethiopic-halehame-aa-er
  - ethiopic-halehame-aa-et
  - ethiopic-halehame-am-et
  - ethiopic-halehame-gez
  - ethiopic-halehame-om-et
  - ethiopic-halehame-sid-et
  - ethiopic-halehame-so-et
  - ethiopic-halehame-ti-er
  - ethiopic-halehame-ti-et
  - ethiopic-halehame-tig
  - hangul
  - hangul-consonant
  - lower-norwegian
  - oromo
  - sidama
  - somali

  - tigre
  - tigrinya-er
  - tigrinya-er-abegede
  - tigrinya-et
  - tigrinya-et-abegede
  - upper-greek
  - upper-norwegian
- -wap-marquee （显示属性）

第 2 章 : 在 AIR 中进行 HTML 和 JavaScript 编程

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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，根页面在应用程序启动时将自动加载。

2 创建应用程序页面和代码。

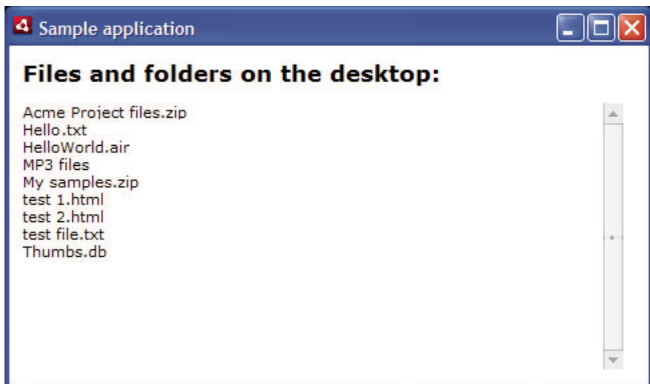
3 使用 ADL 实用程序或 Dreamweaver 测试应用程序。

4 使用 ADT 实用程序或 Dreamweaver 将应用程序打包为 AIR 安装文件。

示例应用程序和安全启示

Adobe AIR 1.0 和更高版本

以下 HTML 代码使用文件系统 API 来列出用户的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。

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 24 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。对所有 File 对象均定义了 getDirectoryListing() 方法，并返回 File 对象的数组。 File.desktopDirectory.getDirectoryListing() 方法返回表示用户桌面上的文件和目录的 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中的文件和目录，并将其名称追加到应用程序 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

Adobe AIR 1.0 和更高版本

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，来自其他源的内容不能访问 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）。

由于存在可通过调用 eval() 函数 （及相关 API）来利用的安全漏洞，因此，默认情况下限制使用这些方法。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。来自不同源的内容分别放置在不同的安全性分类中，称为沙箱 （请参阅安全沙箱）。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 30 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，则运行时将调度 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 20 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 63 页的 "Adobe AIR 中的 HTML 安全性 "。

避免与安全相关的 JavaScript 错误

Adobe AIR 1.0 和更高版本

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，则运行时将调度 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

Adobe AIR 1.0 和更高版本

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 65 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他沙箱

Adobe AIR 1.0 和更高版本

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 30 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他沙箱。如果该内容还必须访问 AIR API，则可以按照第 31 页的 " 设置沙箱桥接口 " 中的说明创建一个沙箱桥。

eval() 函数

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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，在下面的情况中，可以通过改写代码来避免使用 eval()。

将属性分配给对象

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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记号构建新对象，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()

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

javascript: URL

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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他沙箱。请参阅第 30 页的 " 将应用程序内容加载到非应用程序序沙箱 "。

document.write() 和 document.writeln()

Adobe AIR 1.0 和更高版本

如果页面 load 事件已处理完毕，则将忽略对 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

动态创建的脚本元素 （例如，使用 innerHTML 或 document.createElement() 方法创建的元素）将被忽略。

通过 JavaScript 访问 AIR API 类

Adobe AIR 1.0 和更高版本

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。runtime 对象的 flash 属性提供了对 Flash 包的访问。同样，runtime 对象的 flash.filesystem 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。HTML 开发人员可能对 Flash Player API 中的其他类感兴趣，将在《针对 HTML 开发人员的 Adobe AIR API 参考》中介绍这些类。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 290 页的 “ActionScript 3.0 类、包和命名空间 ” 以及第 288 页的 “ 针对 JavaScript 开发人员的 ActionScript 基础知识 ”。

使用 AIRAliases.js 文件

Adobe AIR 1.0 和更高版本

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 API 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

Adobe AIR 1.0 和更高版本

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 的其他任何地方，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。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。	`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 273 页的“URI 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 271 页的 “HTTP 通信 ”。

在 HTML 中嵌入 SWF 内容

Adobe AIR 1.0 和更高版本

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）。

如果在显示 HTML 和 SWF 内容的 NativeWindow 对象中启用了透明度，则用于嵌入内容的窗口模式 (wmode) 设置为 window 时， AIR 不会显示 SWF 内容。要在透明窗口的 HTML 页中显示 SWF 内容，请将 wmode 参数设置为 opaque 或 transparent。 window 是 wmode 的默认值，因此如果不指定值，则可能不会显示所需内容。

以下示例说明如何使用 HTML object 标签在 HTML 内容中显示 SWF 文件。应将 wmode 参数设置为 opaque 才可显示内容，即使基础 NativeWindow 对象是透明的。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>
    <param name="wmode" value="opaque"></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);
    var flash = createSWFObject(urlString, 'opaque', 650, 650);
    displayContainer.appendChild(flash);
}
function createSWFObject(urlString, wmodeString, 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);
    var wmodeParam = document.createElement("param");
    wmodeParam.setAttribute("name","wmode");
    wmodeParam.setAttribute("value",wmodeString);
    SWFObject.appendChild(wmodeParam);
    return SWFObject;
}
</script>
```

如果缩放或旋转 HTMLLoader 对象或者将 alpha 属性设置为 1.0 之外的任何值，则不会显示 SWF 内容。在早于 AIR 1.5.2 的版本中，无论将 wmode 值设置为何值，透明窗口中都不显示 SWF 内容。

注：当嵌入式 SWF 对象尝试加载外部资源 （如视频文件）时，如果 HTML 文件中未提供视频文件的绝对路径，可能无法正确呈现 SWF 内容。不过，嵌入式 SWF 对象可以使用相对路径加载外部图像文件。

下面的示例描述了如何通过嵌入到 HTML 内容中的 SWF 对象加载外部资源：

```
var imageLoader;

function showSWF(urlString, elementID){
    var displayContainer = document.getElementById(elementID);
    imageLoader = createSWFObject(urlString,650,650);
    displayContainer.appendChild(imageLoader);
}

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);

    var flashVars = document.createElement("param");
    flashVars.setAttribute("name","FlashVars");

    //Load the asset inside the SWF content.
    flashVars.setAttribute("value","imgPath=air.jpg");
    SWFObject.appendChild(flashVars);

    return SWFObject;
}
function loadImage()
{
  showSWF("ImageLoader.swf", "imageSpot");

}
```

在下面的 ActionScript 示例中，读取 HTML 文件传递的图像路径，并将该图像加载到舞台上:

```
package
{
	import flash.display.Sprite;
	import flash.display.LoaderInfo;
	import flash.display.StageScaleMode;
	import flash.display.StageAlign;
	import flash.display.Loader;
	import flash.net.URLRequest;

	public class ImageLoader extends Sprite
	{
		public function ImageLoader()
		{

			var flashvars = LoaderInfo(this.loaderInfo).parameters;

			if(flashvars.imgPath){
				var imageLoader = new Loader();
				var image = new URLRequest(flashvars.imgPath);
				imageLoader.load(image);
				addChild(imageLoader);
				imageLoader.x = 0;
				imageLoader.y = 0;
				stage.scaleMode=StageScaleMode.NO_SCALE;
				stage.align=StageAlign.TOP_LEFT;
			}
		}
	}
}
```

在 HTML 页中使用 ActionScript 库

Adobe AIR 1.0 和更高版本

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。

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。 acompc 实用程序是 Flex SDK 的一部分， Flex SDK 文档中对此有相关说明。

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

Adobe AIR 1.0 和更高版本

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，需将对 JavaScript 对象的引用（例如 window 或 document）传递给 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他对象）。

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 Window 对象作为一个参数。使用此 Window 引用，可访问页面中的其他对象，包括变量、函数和 DOM 元素，如以下定义所示：

```
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

Adobe AIR 1.0 和更高版本

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);
```

跨脚本访问不同安全沙箱中的内容

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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他安全沙箱。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他沙箱中，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，映射到其他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他沙箱 （可选）执行跨域 XMLHttpRequest。

有关详细信息，请参阅 第 10 页的 “HTML frame 和 iframe 元素 ”、第 63 页的 “Adobe AIR 中的 HTML 安全性 ” 和 第 5 页的 “XMLHttpRequest 对象 ”。

将应用程序内容加载到非应用程序沙箱

Adobe AIR 1.0 和更高版本

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/ 或某些无不利影响的其他值作为源域。

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，您不能访问位于 www.example.com/local/（或其子目录中）中的任何远程内容，因为 AIR 会将请求重新映射到本地应用程序目录：无论是页面导航、XMLHttpRequest 还是采用其他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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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他访问权利。子级内容可以调用 save 函数，如下所示：

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

应用程序内容向其他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

Adobe AIR 1.0 和更高版本

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，必须将侦听器添加到通过 window.open() 函数创建的新子级 window 对象：

```
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 3 章：处理 AIR 中与 HTML 相关的事件

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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	在为响应加载操作而创建 HTML DOM 后，紧接在 HTML 页面中的 onload 事件后调度。
htmlBoundsChanged	在 contentWidth 和 / 或 contentHeight 属性发生了变化时调度。
locationChange	在 HTMLLoader 的 location 属性发生了变化时调度。
locationChanging	由于用户导航、 JavaScript 调用或重定向，在 HTMLLoader 的位置发生变化前调度。当您调用 load()、loadString()、reload()、historyGo()、historyForward() 或 historyBack() 方法时未调度 locationChanging 事件。 调用所调度事件对象的 preventDefault() 方法会取消导航。 如果系统浏览器中打开某个链接，则不会调度 locationChanging 事件，因为 HTMLLoader 不会改变位置。
scroll	只要 HTML 引擎更改滚动位置，便会调度此事件。 Scroll 事件的引发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 window.scrollTo() 方法。在文本输入或文本区域中输入文本也可能会引发 scroll 事件。
uncaughtScriptException	当在 HTMLLoader 中发生 JavaScript 异常，并且在 JavaScript 代码中未捕获到该异常时调度。

AIR 类 - 事件处理与 HTML DOM 中其他事件处理的不同之处

Adobe AIR 1.0 和更高版本

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
```

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

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

不过，由于运行时对象不会出现在 DOM 中，因此您只能通过调用 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，以便当该事件发生时，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侦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

Adobe AIR 1.0 和更高版本

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，您必须调用已调度的事件对象的 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 MP3 文件中的足量数据后， Sound 对象会调度 id3 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

Adobe AIR 1.0 和更高版本

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 AIR 参考文档中出现的有关事件阶段的信息。

Adobe AIR 事件对象

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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() 侦听器函数以响应 myFileStream 调度的 complete 事件：

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

AIR Event 类定义许多类常量 （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 API 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，在添加事件侦听器时，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

Adobe AIR 1.0 和更高版本

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。您可以使用 preventDefault() 方法阻止或取消与少量事件关联的默认行为。有关详细信息，请参阅第 37 页的 “ 取消事件默认行为 ”。

了解 Event 类的方法

Adobe AIR 1.0 和更高版本

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 35 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

Adobe AIR 1.0 和更高版本

Event 类有两个实用程序方法。clone() 方法用于创建事件对象的副本。toString() 方法用于生成事件对象属性的字符串表示形式以及它们的值。

取消事件默认行为

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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，同时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

Adobe AIR 1.0 和更高版本

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，事件对象作为实参传递给 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() 方法包括三个其他参数，在针对 HTML 开发人员的 Adobe AIR API 参考中进行了介绍；这些参数为 useCapture、 priority 和 useWeakReference。

删除事件侦听器

Adobe AIR 1.0 和更高版本

可以使用 removeEventListener() 方法删除不再需要的事件侦听器。建议删除将不再使用的所有侦听器。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 页面中的事件侦听器

Adobe AIR 1.0 和更高版本

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侦听器。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 事件侦听器。如果发生页面导航（在 HTMLLoader 对象中），请在 window 对象的 unload 事件发生期间删除事件侦听器。

例如，下面的 JavaScript 代码删除 uncaughtScriptException 事件的事件侦听器：

```
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 (window.nativeWindow) 的 closing 事件。例如，下面的 JavaScript 代码删除 uncaughtScriptException 事件的事件侦听器：

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

为了防止发生此错误，您还可以在事件侦听器运行时将其删除（如果该事件只需要处理一次）。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 事件添加一个事件侦听器。在调用 complete 事件处理函数时，它会使用 removeEventListener() 函数删除它自己的事件侦听器：

```
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侦听器，则还会使系统垃圾回收器能回收与这些侦听器关联的任何内存空间。

检查有无现有的事件侦听器

Adobe AIR 1.0 和更高版本

hasEventListener() 方法用于检查某个对象是否存在事件侦听器。

没有侦听器的错误事件

Adobe AIR 1.0 和更高版本

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 4 章：为 AIR HTML 容器编写脚本

Adobe AIR 1.0 和更高版本

在 Adobe® AIR® 中，HTMLLoader 类用作 HTML 内容的容器。此类提供了许多属性和方法，用于控制 HTML 内容的行为和外观。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 HTMLHost 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 实现分配给其他 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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	控制对象及所其包含的所有内容的可见性。

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 的显示宽度和高度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，大多数其他类型的显示对象也存在此现象。

HTMLLoader 内容的透明度

Adobe AIR 1.0 和更高版本

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 对象下的其他显示对象在 HTML 内容的前景元素后是可见的。

如果 body 元素或 HTML 文档的任何其他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，并使用指定的背景颜色呈现。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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。

此外，运行时无法在透明窗口中显示 PDF 内容。如果将 object 或 embed 标签的 wmode 参数设置为 opaque 或 transparent，则运行时仅显示在 HTML 页中嵌入的 SWF 内容。因为 wmode 的默认值是 window，所以在透明窗口中不显示 SWF 内容，除非明确设置 wmode 参数。

注：在早于 AIR 1.5.2 的版本中，无论使用哪种 wmode 值，都不显示在 HTML 中嵌入的 SWF。

有关在 HTMLLoader 中加载这些类型的媒体的详细信息，请参阅第 26 页的 “ 在 HTML 中嵌入 SWF 内容 ” 和第 231 页的 “ 在 AIR 中添加 PDF 内容 ”。

高级显示属性

Adobe AIR 1.0 和更高版本

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他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
alpha	会降低 HTML 内容的易读性
filters	在 HTML 窗口中，外部效果沿窗口边缘剪裁
graphics	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）。paintsDefaultBackground 属性必须为 false，绘制的形状才可见。
opaqueBackground	不更改默认背景的颜色。paintsDefaultBackground 属性必须为 false，此颜色层才可见。
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。它们未在针对 HTML 开发人员的 Adobe AIR API 参考中列出，在 AIRAliases.js 文件中也没有别名。有关这些类的文档，请参阅用于 Adobe Flash Platform 的 ActionScript 3.0 参考。

访问 HTML 历史记录列表

Adobe AIR 1.0 和更高版本

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。

类成员	说明
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目作为 HTMLHistoryItem 类型的对象存储。 HTMLHistoryItem 类包含下列属性：

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

Adobe AIR 1.0 和更高版本

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。需在调用 load() 方法之前设置 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应用于所有 URLRequest 对象，不只是与 HTMLLoader 对象的 load() 方法一起使用的 URLRequest 对象。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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

JavaScript 提供了多个 API 来控制显示 HTML 内容的窗口。在 AIR 中，可以通过实现自定义 HTMLHost 类覆盖这些 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 28 页的 " 在 HTML 页中使用 ActionScript 库 "。

关于扩展 HTMLHost 类

Adobe AIR 1.0 和更高版本

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，不会启用所列的 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 对象分配给为基于 HTML 的 AIR 应用程序创建的初始窗口以及使用 JavaScript 的 window.open() 方法的默认实现创建的所有窗口。

示例：扩展 HTMLHost 类

Adobe AIR 1.0 和更高版本

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：

Flex 示例：

1 创建一个 HTMLHost 类的扩展类（子类）。

2 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.*;
    import flash.display.StageScaleMode;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;

    public class CustomHost extends HTMLHost
    {
        import flash.html.*;
        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;
        }
        override public function updateTitle(title:String):void
        {
            htmlLoader.stage.nativeWindow.title = title;
        }
    }
}
```

3 在包含 HTMLLoader 的代码（不是新建的 HTMLHost 子类的代码）中，创建新类的对象。将新对象分配给 HTMLLoader 的 htmlHost 属性。以下 Flex 代码使用上一步中定义的 CustomHost 类：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication
    xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    applicationComplete="init()">
    <mx:Script>
        <![CDATA[
            import flash.html.HTMLLoader;
            import CustomHost;
            private function init():void
            {
                var html:HTMLLoader = new HTMLLoader();
                html.width = container.width;
                html.height = container.height;
                var urlReq:URLRequest = new URLRequest("Test.html");
                html.htmlHost = new CustomHost();
                html.load(urlReq);
                container.addChild(html);
            }
        ]]>
    </mx:Script>
    <mx:UIComponent id="container" width="100%" height="100%"/>
</mx:WindowedApplication>
```

若要测试此处所述的代码，请将具有以下内容的 HTML 文件放在应用程序目录下：

```
<html>
    <head>
        <title>Test</title>
    </head>
    <script>
        function openWindow()
        {
            window.runtime.trace("in");
            document.title = "foo"
            window.open('Test.html');
            window.runtime.trace("out");
        }
    </script>
    <body>
        <a href="#" onclick="openWindow()">window.open('Test.html')</a>
    </body>
</html>
```

Flash Professional 示例：

1 为 AIR 创建一个 Flash 文件。将其文档类设置为 CustomHostExample，然后将文件另存为 CustomHostExample.fla。

2 创建一个名为 CustomHost.as 的 ActionScript 文件，该文件包含一个 HTMLHost 类的扩展类 （子类）。此类将覆盖新类的某些方法，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() 方法将在新窗口中打开 HTML 页，更改 window.document.title 属性（包括 HTML 页的 <title> 元素的设置）将设置该窗口的标题。

```
package
{
    import flash.display.StageScaleMode;
    import flash.display.NativeWindow;
    import flash.display.NativeWindowInitOptions;
    import flash.events.Event;
    import flash.events.NativeWindowBoundsEvent;
    import flash.geom.Rectangle;
    import flash.html.HTMLLoader;
    import flash.html.HTMLHost;
    import flash.html.HTMLWindowCreateOptions;
    import flash.text.TextField;

    public class CustomHost extends HTMLHost
    {
        public var statusField:TextField;

        public function CustomHost(defaultBehaviors:Boolean=true)
        {
            super(defaultBehaviors);
        }
        override public function windowClose():void
        {
            htmlLoader.stage.nativeWindow.close();
        }
        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;
        }
        override public function updateLocation(locationURL:String):void
        {
            trace(locationURL);
        }
        override public function set windowRect(value:Rectangle):void
        {
            htmlLoader.stage.nativeWindow.bounds = value;
```

```
        }
        override public function updateStatus(status:String):void
        {
            statusField.text = status;
            trace(status);
        }
        override public function updateTitle(title:String):void
        {
            htmlLoader.stage.nativeWindow.title = title + "- Example Application";
        }
        override public function windowBlur():void
        {
            htmlLoader.alpha = 0.5;
        }
        override public function windowFocus():void
        {
            htmlLoader.alpha = 1;
        }
    }
}
```

3 创建另一个名为 CustomHostExample.as 的 ActionScript 文件，以包含应用程序的文档类。此类将创建一个 HTMLLoader 对象，并将其主机属性设置为上一步中定义的 CustomHost 类的一个实例：

```
package
{
    import flash.display.Sprite;
    import flash.html.HTMLLoader;
    import flash.net.URLRequest;
    import flash.text.TextField;

    public class CustomHostExample extends Sprite
    {
        function CustomHostExample():void
        {
            var html:HTMLLoader = new HTMLLoader();
            html.width = 550;
            html.height = 380;
            var host:CustomHost = new CustomHost();
            html.htmlHost = host;

            var urlReq:URLRequest = new URLRequest("Test.html");
            html.load(urlReq);

            addChild(html);

            var statusTxt:TextField = new TextField();
            statusTxt.y = 380;
            statusTxt.height = 20;
            statusTxt.width = 550;
            statusTxt.background = true;
            statusTxt.backgroundColor = 0xEEEEEEEE;
            addChild(statusTxt);

            host.statusField = statusTxt;
        }
    }
}
```

若要测试此处所述的代码，请将具有以下内容的 HTML 文件放在应用程序目录下：

```
<html>
    <head>
    <title>Test</title>
    <script>
    function openWindow()
    {
    document.title = "Test"
    window.open('Test.html');
    }
    </script>
    </head>
    <body bgColor="#EEEEEE">
    <a href="#" onclick="window.open('Test.html')">window.open('Test.html')</a>
    <br/><a href="#" onclick="window.document.location='http://www.adobe.com'">
    window.document.location = 'http://www.adobe.com'</a>
    <br/><a href="#" onclick="window.moveBy(6, 12)">moveBy(6, 12)</a>
    <br/><a href="#" onclick="window.close()">window.close()</a>
    <br/><a href="#" onclick="window.blur()">window.blur()</a>
    <br/><a href="#" onclick="window.focus()">window.focus()</a>
    <br/><a href="#" onclick="window.status = new Date().toString()">window.status=new
Date().toString()</a>
    </body>
</html>
```

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 acompc 组件编译器将该类编译为 SWF 文件。

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

注：acompc 编译器包含在 Flex 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 compc，组件编译器中提供了使用 acompc 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

Adobe AIR 1.0 和更高版本

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.moveBy()、 window.moveTo()、 window.resizeTo()、 window.resizeBy() 的 JavaScript 调用

Adobe AIR 1.0 和更高版本

覆盖 set windowRect() 方法以处理 HTML 内容范围的更改。当某页中的 JavaScript 调用 window.moveBy()、 window.moveTo()、 window.resizeTo() 或 window.resizeBy() 时，将调用 set windowRect() 方法。以下示例仅更新了桌面窗口范围：

```
override public function set windowRect(value:Rectangle):void
{
    htmlLoader.stage.nativeWindow.bounds = value;
}
```

处理对 window.open() 的 JavaScript 调用

Adobe AIR 1.0 和更高版本

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。

作为参数传递到 createWindow() 方法的对象是一个 HTMLWindowCreateOptions 对象。HTMLWindowCreateOptions 类包含相关属性，可报告在对 window.open() 的调用中， features 参数字符串中设置的值：

HTMLWindowCreateOptions 属性	在对 window.open() 的 JavaScript 调用中， features 字符串中的相应设置
fullscreen	fullscreen
height	height
locationBarVisible	location
menuBarVisible	menubar
resizeable	resizable
scrollBarsVisible	scrollbars
statusBarVisible	status
toolBarVisible	toolbar
width	width
x	left 或 screenX
y	top 或 screenY

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() 方法的其他参数由系统处理。 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

Adobe AIR 1.0 和更高版本

覆盖 windowClose() 以处理对 window.close() 方法的 JavaScript 调用。以下示例在调用 window.close() 方法时将关闭桌面窗口。

```
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（可能包含其他内容）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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。可以更改其他界面元素，如文本字段。

处理对 window.blur() 和 window.focus() 的 JavaScript 调用

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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	一个用于定义新窗口的位置和大小的 Rectangle 对象。

例如，以下代码使用 HTMLLoader.createRootWindow() 方法创建带有使用滚动条的 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() 创建的窗口将独立于打开的 HTML 窗口。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()，opener 和 parent 将引用打开的 HTML 窗口。

第 5 章：使用矢量

Adobe AIR 1.5 和更高版本

Vector 实例是 " 指定类型的数组 "，这表示 Vector 实例中的所有元素始终具有同一数据类型。某些 AIR API （如 NativeProcess 和 NetworkInfo）使用 Vector 作为属性或方法的数据类型。

在 Adobe AIR 中运行的 JavaScript 代码中， Vector 类是作为 air.Vector （在 AIRAliases.js 文件中）引用的。

矢量基础知识

Adobe AIR 1.5 和更高版本

在声明 Vector 变量或实例化 Vector 对象时，要显式指定 Vector 可以包含的对象的数据类型。指定的数据类型称为 Vector 的 " 基本类型 "。在运行时，会检查用于设置或检索 Vector 值的任何代码。如果要添加或检索的对象的数据类型与 Vector 的基本类型不匹配，则会发生错误。

除数据类型限制之外， Vector 类还具有一些其他限制，从而有别于 Array 类：

- Vector 是一种密集数组。即使某个 Array 对象在位置 1 到 6 没有值，该对象的索引 0 和 7 处也可以有值。但是， Vector 的每个索引位置都必须有值 （或为 null）。
- Vector 可以是固定长度。这表示 Vector 包含的元素数不能更改。
- 对 Vector 的元素的访问需要接受范围检查。绝对不能从大于最后一个元素索引 (length - 1) 的索引中读取值。绝对不能对超过当前最后一个索引一个以上位置的索引设置值 （也就是说，只能在现有索引或索引 [length] 处设置值）。

由于 Vector 具有这些限制，因此 Vector 相对于所有元素均为单个类的实例的 Array 实例有三个主要优点：

- 性能：使用 Vector 实例时的数组元素访问和迭代的速度比使用 Array 实例时的速度要快很多。
- 类型安全性：这类错误的例子包括将数据类型错误的值分配给 Vector 或从 Vector 中读取值时预期错误的数据类型。在运行时，当向 Vector 对象添加数据或从 Vector 对象读取数据时会检查数据类型。
- 可靠性：与 Array 相比，运行时范围检查 （或固定长度检查）大大提高了可靠性。

除了有一些限制和优点以外， Vector 类与 Array 类非常相似。 Vector 对象的属性和方法与 Array 的属性和方法类似 （通常完全相同）。对于大多数需要使用所有元素都具有相同数据类型的 Array 的情况， Vector 实例更为可取。

重要概念和术语

以下参考列表包含在对处理例程的数组和矢量进行编程时需要了解的重要术语：

数组访问 ([]) 运算符 一对中括号，其中含有唯一标识数组元素的索引或键。此语法用在矢量变量名称之后，以指定矢量的单个元素而不是整个矢量。

基本类型 允许 Vector 实例存储的对象的数据类型。

元素 矢量中的单个项目。

索引 用于标识索引数组中的单个元素的数字 " 地址 "。

T 本文档中使用的标准约定，用于表示 Vector 实例的基本类型（与具体的基本类型无关）。T 约定用于表示类名称，如 Type 参数说明中所示。（"T" 表示 " 类型 "，如同在 " 数据类型 " 中一样。）。

类型参数 与 Vector 类名称一起使用以指定 Vector 的基本类型 （它存储的对象的数据类型）的语法。该语法包括一个点 (.)，然后是由尖括号 (<>) 括起来的数据类型名称。放在一起后类似于：Vector.<T>。在本文档中，在类型参数中指定的类通常表示为 T。

Vector 一种数组类型，其所有元素都是同一数据类型的实例。

创建矢量

AIR 1.5 和更高版本

通过调用 air.Vector["<T>"]() 构造函数创建 Vector 实例。调用此构造函数时，即指定了 Vector 变量的基类型。可以使用 type 参数语法指定 Vector 的基本类型。在代码中，类型参数紧跟单词 Vector。类型参数包括左中括号，然后是字符串 （包含用尖括号 (<>) 括起来的基类名称），最后是右中括号。下面的示例显示此语法：

```
var v = new air.Vector["<String>"]();
```

在此示例中，变量 v 被声明为 String 对象的一个矢量。换句话说，它表示只能包含 String 实例的索引数组。

如果使用不带任何参数的 air.Vector["<T>"]() 构造函数，将创建一个空 Vector 实例。可以通过检查 Vector 的 length 属性来测试它是否为空。例如，下面的代码调用不带参数的 Vector["<T>"]() 构造函数：

```
var names = new air.Vector["<String>"]();
air.trace(names.length); // output: 0
```

如果您预先知道 Vector 最初需要多少元素，则可以预定义 Vector 中的元素数。若要使用特定数量的元素创建 Vector，请将元素数作为第一个参数 （length 参数）进行传递。因为 Vector 元素不能为空，所以会使用具有基本类型的实例填充这些元素。如果基本类型是允许使用 null 值的引用类型，则所有元素都包含 null。否则，所有元素都包含该类的默认值。例如，Number 变量不能为 null。因此，在下面的代码清单中，使用三个元素创建名为 ages 的 Vector，其中每个元素都包含默认 Number 值 NaN：

```
var ages = new air.Vector["<Number>"](3);
air.trace(ages); // output: NaN, NaN, NaN
```

使用 Vector["<T>"]() 构造函数还可以创建固定长度 Vector，方法是将 true 作为第二个参数 （fixed 参数）进行传递。在这种情况下，将使用指定的元素数创建 Vector，且元素数不可更改。但是请注意，仍然可以更改固定长度 Vector 的元素值。

如果创建 AIR 运行时对象 （在 window.runtime 对象中定义的类）的 Vector，请在调用 Vector 构造函数时引用该类的完全限定 ActionScript 3.0 名称。例如，下面的代码创建 File 对象的一个 Vector：

```
var files = new air.Vector["flash.filesystem.File"](3);
```

向矢量中插入元素

Adobe AIR 1.5 和更高版本

将元素添加到矢量的最基本方法是使用数组访问 ([]) 运算符：

```
songTitles[5] = "Happy Birthday";
```

如果 Vector 在该索引处还没有元素，则会创建该索引并将值存储在那里。

对于 Vector 对象，您只能向现有索引或下一个可用索引赋值。下一个可用索引对应于 Vector 对象的 length 属性。向 Vector 对象添加新元素的最安全方式是使用类似于下面的清单的代码：

```
myVector[myVector.length] = valueToAdd;
```

对于数组，可以使用三个 Vector 类方法 （push()、 unshift() 和 splice()）向矢量中插入元素。

注：如果某个 Vector 对象的 fixed 属性为 true，则不能更改该 Vector 中的元素数。如果尝试使用 push() 方法或其他方式向固定长度 Vector 添加新元素，则会发生错误。

检索值和删除矢量元素

Adobe AIR 1.5 和更高版本

检索矢量中的元素值的最简单方法是使用数组访问 ([]) 运算符。若要检索矢量元素的值，请在赋值语句的右侧使用矢量对象名称和索引编号：

```
var myFavoriteSong = songTitles[3];
```

可以尝试使用不存在元素的索引来检索矢量中的值。在这种情况下， Vector 会引发 RangeError 异常。

可以使用 Array 和 Vector 类的三种方法 （pop()、 shift() 和 splice()）删除元素。

```
var vegetables = new air.Vector["<String>"];
vegetables.push("spinach");
vegetables.push("green pepper");
vegetables.push("cilantro");
vegetables.push("onion");
var spliced = vegetables.splice(2, 2);
air.trace(spliced); // output: spinach,green pepper
```

可以使用 length 属性截断矢量。

如果将矢量的 length 属性的长度设置为小于矢量当前长度，将截断矢量。将删除索引编号大于 length 新值减 1 的位置处存储的所有元素。

注：如果某个 Vector 对象的 fixed 属性为 true，则不能更改该 Vector 中的元素数。如果尝试使用此处介绍的方法删除固定长度 Vector 中的元素或截断固定长度 Vector，则会发生错误。

Vector 对象的属性和方法

Adobe AIR 1.5 和更高版本

Array 对象的很多方法和属性都可用于 Vector 对象。例如，可以调用 reverse() 方法来更改 Vector 的元素的顺序。可以调用 sort() 方法对 Vector 的元素进行排序。但是， Vector 类不包含 sortOn() 方法。

有关受支持的属性和方法的详细信息，请参阅 《针对 HTML 开发人员的 Adobe AIR 语言参考》中的 Vector 类文档。

示例：使用需要矢量的 AIR API

Adobe AIR 1.5 和更高版本

有些 Adobe AIR 运行时类使用矢量作为属性或方法返回值。例如， NetworkInfo 类的 findInterfaces() 方法返回 NetworkInterface 对象的数组。 NativeProcessStartupInfo 类的 arguments 属性 是一个字符串矢量。

访问返回矢量对象的 AIR API

Adobe AIR 2.0 和更高版本

NetworkInfo 类的 findInterfaces() 方法返回 NetworkInterface 对象的数组。例如，下面的代码列出计算机网络接口：

```
var netInfo = air.NetworkInfo;
var interfaces = netInfo.findInterfaces();
for (i = 0; i < interfaces.length; i++)
{
    air.trace(interfaces[i].name];
    air.trace(" hardware address: ", interface.hardwareAddress);
}
```

可以循环访问 NetworkInfo 对象的矢量，就像循环访问数组一样。使用 for 循环和方括号可以访问矢量的索引元素。

NetworkInterface 对象的 interfaces 属性是 InterfaceAddress 对象的一个矢量。下面的代码扩展了前面的示例，添加了一个函数来枚举每个网络接口的接口地址：

```
var netInfo = air.NetworkInfo;
var interfaces = netInfo.findInterfaces();
for (i = 0; i < interfaces.length; i++)
{
    air.trace(interfaces[i].name];
    air.trace(" hardware address: ", interface.hardwareAddress);
    air.trace(" addresses: ", traceAddresses(i);
}

function traceAddresses(i)
{
    returnString = new String();
    for (j = 0; j < interfaces[i].addresses.length; j++)
        returnString += interfaces[i],addresses[j].address + " ";
    }
}
```

设置是矢量的 AIR API

Adobe AIR 2.0 和更高版本

NativeProcessStartupInfo 类的 arguments 属性 是一个字符串矢量。要设置此属性，请使用 air.Vector() 构造函数创建一个字符串矢量。可以使用 push() 方法向矢量添加字符串：

```
var arguments = new air.Vector["<String>"]();

arguments.push("test");
arguments.push("44");

var startupInfo = new air.NativeProcessStartupInfo();
startupInfo.arguments = arguments;
startupInfo.executable = File.applicationDirectory.resolvePath("myApplication.exe");

process = new air.NativeProcess();
process.start(startupInfo);
```

有关使用本机进程 API 的详细信息，请参阅网络和通信中的“ 与本机进程通信 ”。

第 6 章：AIR 安全性

Adobe AIR 1.0 和更高版本

AIR 安全性基础知识

Adobe AIR 1.0 和更高版本

AIR 应用程序运行的安全性限制与本机应用程序一样。一般来说，AIR 应用程序像本机应用程序一样，对操作系统功能有广泛的访问权限，例如读取和写入文件、启动应用程序、绘制到屏幕以及与网络通信。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

Adobe AIR 1.0 和更高版本

AIR 应用程序通过具有 air 扩展名的 AIR 安装程序文件分发，或者通过具有本机平台的文件格式和扩展名的本机安装程序分发。例如， Windows 的本机安装程序格式是 EXE 文件，对于 Android，则本机格式是 APK 文件。

当安装 Adobe AIR 后，打开 AIR 安装程序文件时，AIR 运行时会管理安装进程。当使用本机安装程序时，操作系统管理安装进程。

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

Adobe AIR 1.0 和更高版本

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。

运行时将安装到桌面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。

注：在 iOS 上，不单独安装 AIR 运行时；每个 AIR 应用程序都是自包含应用程序。

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。

无缝安装 （运行时和应用程序）

Adobe AIR 1.0 和更高版本

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单化的安装过程。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

Adobe AIR 1.0 和更高版本

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。

应用程序安装流程

Adobe AIR 1.0 和更高版本

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他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。与其他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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

AIR 应用程序具有写入用户硬盘驱动器上的任意位置的权限；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中，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> 为用户的 " 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> 为用户的 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> 为 /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 126 页的 " 使用文件系统 "。

更新 Adobe AIR

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 AIR 文件的远程位置。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。

Updater 类仅用于更新作为 AIR 文件分发的应用程序。作为本机应用程序分发的应用程序必须使用本机操作系统的更新组件（如果有）。

注：开发人员可以通过设置应用程序描述符文件的 versionNumber 属性指定应用程序的版本。

卸载 AIR 应用程序

Adobe AIR 1.0 和更高版本

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。然而，它不删除应用程序可能写入应用程序目录外的所有文件。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。

针对管理员的 Windows 注册表设置

Adobe AIR 1.0 和更高版本

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	指定允许安装不受信任的 AIR 应用程序 （没有可信证书的应用程序）。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 ”。

Adobe AIR 中的 HTML 安全性

Adobe AIR 1.0 和更高版本

本主题介绍 AIR HTML 安全体系结构，以及如何使用 iframe、帧与沙箱桥设置基于 HTML 的应用程序和安全地将 HTML 内容集成到基于 SWF 的应用程序。

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。

AIR SDK 为呈现 HTML 内容提供三个类。

HTMLLoader 类提供 JavaScript 代码和 AIR API 之间的紧密集成。

StageWebView 类是一个 HTML 呈现类，与主机 AIR 应用程序仅有非常有限的集成。StageWebView 类加载的内容从不放置在应用程序安全沙箱中，因而无法访问主机 AIR 应用程序中的数据或调用其中的函数。在桌面平台上， StageWebView 类根据 Webkit（HTMLLoader 类也使用它）使用内置 AIR HTML 引擎。在移动平台上，StageWebView 类使用操作系统提供的 HTML 控件。因此，在移动平台上， StageWebView 类与系统 Web 浏览器有着相同的安全性注意事项和漏洞。

TextField 类可以显示 HTML 文本字符串。 JavaScript 不可以执行，但文本可以包括链接和外部加载的图像。

有关详细信息，请参阅第 20 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

Adobe AIR 1.0 和更高版本

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 " 到其他页面时会完全刷新。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。在页面的 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帧中的内容。）有关应用程序沙箱中的 JavaScript 限制的完整列表，请参阅第 65 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他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

Adobe AIR 1.0 和更高版本

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他域中的内容：

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

Adobe AIR 1.0 和更高版本

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他访问权限。通常，应用程序内容向其他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

Adobe AIR 1.0 和更高版本

在 第 63 页的 "Adobe AIR 中的 HTML 安全性 " 主题的简介中已介绍过，运行时强制执行规则，并提供克服 HTML 和 JavaScript 中可能的安全漏洞的机制。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 20 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

Adobe AIR 1.0 和更高版本

对于应用程序安全沙箱中的 HTML 内容，加载代码后 （即在调度 body 元素的 onload 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，只能通过有限的方式使用 eval() 函数。以下规则适用于在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");
```

如果应用程序沙箱中允许使用代码，则动态生成的代码 （例如在调用 eval() 函数时生成的代码）将导致安全风险。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 script 标签的 src 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 setTimout() 函数，其中，第一个参数 （用于定义要异步运行的函数）为要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）。

对于其他 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 AIR 中的 JavaScript 限制和第 70 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/cn/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）

Adobe AIR 1.0 和更高版本

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 70 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

Adobe AIR 1.0 和更高版本

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 事件期间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 allowcrosscomainxhr 属性。如果将此属性设置为任何非空值，则会允许帧或 iframe 中的内容使用 JavaScript XMLHttpRequest 对象从域 （注意不是调用请求的代码的域）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 68 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）

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

如果通过调用 JavaScript window.open() 方法创建的窗口中显示了非应用程序安全沙箱中的内容，则窗口的标题将以主 （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而触发的事件。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() 方法设置为在延迟 （例如调用 setTimeout() 函数）后执行。

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。

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内容 （请参阅安全沙箱）只可使用 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

Adobe AIR 1.0 和更高版本

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 20 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

Adobe AIR 1.0 和更高版本

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，请参阅从字符串中加载 HTML 内容。

通过脚本访问不同域中的内容

Adobe AIR 1.0 和更高版本

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他内容 （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

Adobe AIR 1.0 和更高版本

通常，一个域中的内容无法调用其他域中的脚本。

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)

Adobe AIR 1.0 和更高版本

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 31 页的“ 设置沙箱桥接口 ”。

限制 API 公开

Adobe AIR 1.0 和更高版本

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 （读取任意文件的内容）易于受到滥用。最好公开未使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。

更多帮助主题

第 30 页的“ 跨脚本访问不同安全沙箱中的内容 ”

写入磁盘

Adobe AIR 1.0 和更高版本

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他数据，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 71 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
app:/	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。
app-storage:/	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。
file:///	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:/ 的文件将放在标准位置中：

- 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\<userName>\Application Data
- 在 Linux 中为 <appData>/<appID>/Local Store/，其中 <appData> 为 /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 71 页的“开发人员的最佳安全做法”。

安全使用不受信任的内容

Adobe AIR 1.0 和更高版本

未分配给应用程序沙箱的内容可以为应用程序提供其他脚本功能，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

Adobe AIR 1.0 和更高版本

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 68 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。运行时会阻止应用程序通过引发 SecurityError 异常，写入或修改使用 app:/ URL 方案的文件和目录。
- 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。其中包括使用 Loader.loadBytes() 方法和 JavaScript eval() 函数。

使用外部源确定路径的风险

Adobe AIR 1.0 和更高版本

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。信任最终由开发人员及其构建的网络连接进行保障，但加载外部数据本身就具有风险，不应在敏感操作中使用此输入。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

Adobe AIR 1.0 和更高版本

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 218 页的 “ 加密的本地存储区 ”。
- 除非网络源可信并且使用 HTTPS: 或传输层安全 (TLS) 协议进行传输，否则不要将未加密的用户凭据传输到网络源。
- 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

Adobe AIR 1.0 和更高版本

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，为应用程序使用明确的版本控制方案将使欺骗用户安装降级版本变得更加困难。

代码签名

Adobe AIR 1.0 和更高版本

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。可使用由外部证书颁发机构 (CA) 颁发的证书或您自己创建的自签名证书对 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、Flash Builder 或使用 adt 生成证书的其他应用程序来创建自签名的证书。自签名证书不保证安装的应用程序为正版，它只能用于测试即将公开发行的应用程序。

第 7 章：使用 AIR 本机窗口

Adobe AIR 1.0 和更高版本

使用 Adobe® AIR® 本机窗口 API 提供的类可创建和管理桌面窗口。

AIR 中的本机窗口的基础知识

Adobe AIR 1.0 和更高版本

有关在 AIR 中使用本机窗口的快速介绍和代码示例，请参阅 Adobe Developer Connection 中的以下快速入门文章：

- 自定义窗口的外观

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

Adobe AIR 1.0 和更高版本

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。在使用 ActionScript 和 Flash Professional 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。
- 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。
- 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。

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。

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，您可以使用其中简便易用的方法。

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。

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。

本机窗口类

Adobe AIR 1.0 和更高版本

本机窗口 API 包含以下类：

包	类
flash.display	• NativeWindow • NativeWindowInitOptions • NativeWindowDisplayState • NativeWindowResize • NativeWindowSystemChrome • NativeWindowType
flash.events	• NativeWindowBoundsEvent • NativeWindowDisplayStateEvent

本机窗口事件流

Adobe AIR 1.0 和更高版本

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他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 API 参考。

控制本机窗口样式和行为的属性

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

以下属性控制窗口的基本外观和行为：

- type
- systemChrome
- transparent
- owner

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

Adobe AIR 1.0 和更高版本

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	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	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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

若要允许窗口与桌面或其他窗口进行 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 1，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他显示对象、其他窗口和桌面。

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。然而，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。

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他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，透明窗口中不能显示 HTML 中的 SWF 和 PDF 内容。有关详细信息，请参阅第 42 页的“在 HTML 页中加载 SWF 或 PDF 内容时的注意事项”。

静态 NativeWindow.supportsTransparency 属性可报告窗口透明度是否可用。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。这种做法可以很好地应对万一此属性测试失败而需要回退的情况。例如，您可以向用户显示警告对话框，或显示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

Adobe AIR 1.0 和更高版本

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。

窗口所有权

一个窗口可以拥有 一个或多个其他窗口。这些拥有的窗口始终显示在主窗口的前面，随主窗口一起最小化和还原，并在关闭主窗口时关闭。窗口所有权无法转让给其他窗口，也无法删除窗口所有权。一个窗口只能归一个主窗口所有，但该窗口可拥有任意数量的其他窗口。

使用窗口所有权，可以更加轻松地管理工具调色板和对话框所使用的窗口。例如，如果在文档窗口中显示与之关联的 " 保存 " 对话框，使文档窗口拥有该对话框可使该对话框自动显示在文档窗口前面。

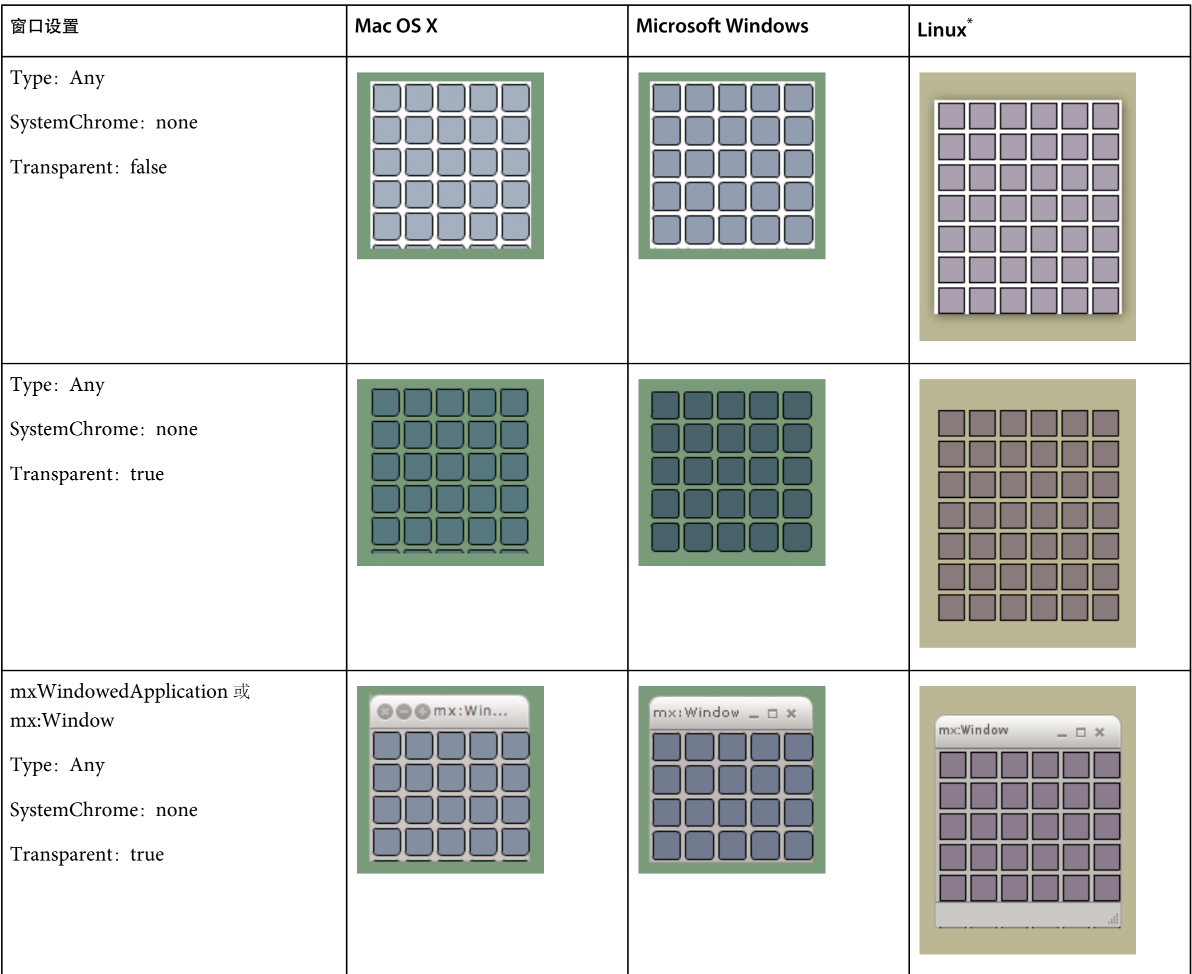
- NativeWindow.owner
- Christian Cantrell：AIR 2.6 中的专属窗口

可视窗口目录

Adobe AIR 1.0 和更高版本

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	Nor...	N...	Normal
Type：utility SystemChrome：standard Transparent：false	Utility	Utility	Utility



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

Adobe AIR 1.0 和更高版本

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他窗口。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。创建的窗口是一个 NativeWindow 对象，其显示列表中包含 HTMLLoader 对象。 HTMLLoader 对象解释并显示窗口的 HTML 内容和 JavaScript 内容。您可以使用 window.nativeWindow 属性从 JavaScript 访问基础 NativeWindow 对象的属性。（只有在 AIR 应用程序沙箱中运行的代码可以访问此属性。）

初始化窗口时（包括初始应用程序窗口），您应考虑在不可见状态下创建窗口、加载内容或执行任何图形更新，然后使该窗口可见。此过程可防止用户看到任何不和谐的可视更改。您可以指定应用程序的初始窗口应在不可见状态下创建，方法是在应用程序描述符中指定 <visible>false</visible> 标签（或通过完全忽略此标签，因为 false 是默认值）。默认情况下，新 NativeWindow 不可见。使用 HTMLLoader createRootWindow() 方法创建 HTML 窗口时，可以将 visible 参数设置为 false。调用 NativeWindow activate() 方法或将 visible 属性设置为 true 以使窗口可见。

指定窗口初始化属性

Adobe AIR 1.0 和更高版本

创建桌面窗口后，无法更改本机窗口的初始化属性。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
owner	null
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他窗口属性，例如 visible、 width 和 height，但可以随时更改这些属性。）

使用 NativeWindowInitOptions 类可设置应用程序创建的其他本机窗口和 HTML 窗口的属性。在创建窗口时，必须将指定窗口属性的 NativeWindowInitOptions 对象传递到 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 52 页的 “处理对 window.open() 的 JavaScript 调用”。

创建初始应用程序窗口

Adobe AIR 1.0 和更高版本

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>
```

注：此示例中引用的 AIRAliases.js 文件是一个脚本文件，它为常用内置 AIR 类定义方便的别名变量。该文件位于 AIR SDK 的 frameworks 目录内。

创建 NativeWindow

Adobe AIR 1.0 和更高版本

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);
```

在将 visible 属性设置为 true 或调用 activate() 方法后，才显示该窗口。

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

Adobe AIR 1.0 和更高版本

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，将创建具有系统镶边的窗口以在指定 URL 处显示内容。例如：

```
newWindow = window.open("xmpl.html", "logWindow", "height=600, width=400, top=10, left=10");
```

注：可以在 ActionScript 中扩展 HTMLHost 类以自定义用 JavaScript window.open() 函数创建的窗口。请参阅第 45 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 像素的系统镶边的简单类型窗口：

```
 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。

如果创建透明窗口，则不一定总能显示加载到该窗口的 HTML 中嵌入的 SWF 内容。对于用于引用 SWF 文件的 object 或 embed 标签，必须将它们的 wmode 参数设置为 opaque 或 transparent。 wmode 的默认值为 window，因此默认情况下，透明窗口中不显示 SWF 内容。无论设置哪种 wmode 值，透明窗口中都无法显示 PDF 内容。（在早于 AIR 1.5.2 的版本中，透明窗口中也无法显示 SWF 内容。）

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：可以从 JavaScript 和 ActionScript 中调用 createRootWindow() 函数。

创建 mx:Window

Adobe AIR 1.0 和更高版本

若要创建 mx:Window，可以将 mx:Window 用作根标签创建 MXML 文件，也可以直接调用 Window 类构造函数。

下例通过调用 Window 构造函数来创建和显示 mx:Window：

```
 var newWindow:Window = new Window();
newWindow.systemChrome = NativeWindowSystemChrome.NONE;
newWindow.transparent = true;
newWindow.title = "New Window";
newWindow.width = 200;
newWindow.height = 200;
newWindow.open(true);
```

向窗口添加内容

Adobe AIR 1.0 和更高版本

向 AIR 窗口添加内容的方式取决于窗口类型。例如，使用 MXML 和 HTML，您能够以声明方式定义窗口的基本内容。可以在应用程序 SWF 文件中嵌入资源，也可以从单独的应用程序文件中加载这些资源。Flex、Flash 和 HTML 内容都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 60 页的 “AIR 安全性 ”。

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) );
```

若要将 HTML 页加载到 Flex 应用程序中，可以使用 Flex HTML 组件。

如果窗口使用透明度 （即窗口的 transparent 属性为 true），则不会显示 HTML 文件中的 SWF 内容，除非将用于引用 SWF 文件的 object 或 embed 标签的 wmode 参数设置为 opaque 或 transparent。wmode 的默认值为 window，因此默认情况下，透明窗口中不显示 SWF 内容。无论使用哪种 wmode 值，透明窗口中都不显示 PDF 内容。

另外，如果缩放、旋转 HTMLLoader 控件或将 HTMLLoader alpha 属性设置为 1.0 之外的任何值，则不会显示 SWF 内容和 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，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 HTMLlLoader 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, similique sunt in culpa qui
officia deserunt mollitia animi, id est laborum et dolorum fuga.</p>
</body>
</html>
```

此示例简要介绍了 AIR 中跨越 JavaScript 和 ActionScript 之间边界的一些高级技术。如果您不熟悉如何使用 ActionScript 显示对象，请参阅 《Adobe ActionScript 3.0 开发人员指南》中的显示编程。

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，但可能不适用于其他内容。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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。

注：当使用 Flex 框架时，通常最好使用框架类来管理窗口行为。通过 mx:WindowedApplication 类和 mx:Window 类可访问大多数 NativeWindow 属性和方法。

获取 NativeWindow 实例

Adobe AIR 1.0 和更高版本

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。

由于 Flex mx:WindowedApplication 和 mx:Window 对象都是显示对象，因此使用 stage 属性即可轻松地在 MXML 文件中引用应用程序窗口，如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" applicationComplete="init();">
    <mx:Script>
        <![CDATA[
            import flash.display.NativeWindow;

            public function init():void{
                var appWindow:NativeWindow = this.stage.nativeWindow;
                //set window properties
                appWindow.visible = true;
            }
        ]]>
    </mx:Script>
</WindowedApplication
```

注： 在 Flex 框架将 WindowedApplication 或 Window 组件添加到窗口舞台后，该组件的 stage 属性为 null。此行为与 Flex Application 组件的行为一致，但并不意味着不能访问舞台或 WindowedApplication 和 Window 组件初始化周期中较早出现事件 （例如 creationComplete）的侦听器中的 NativeWindow 实例。在调度 applicationComplete 事件后，访问舞台和 NativeWindow 实例是安全的。

激活、显示和隐藏窗口

Adobe AIR 1.0 和更高版本

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他窗口的顺序。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。

更改窗口的可见性时，该窗口所拥有的所有窗口的可见性也将发生更改。例如，如果隐藏窗口，则该窗口所拥有的所有窗口也将隐藏。

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

Adobe AIR 1.0 和更高版本

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他窗口的前面或后面。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 来使该窗口位于其他窗口的前面。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他应用程序中窗口的前面。由于此行为会阻碍用户查看其他窗口，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他窗口的不便。
- 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。

如果窗口拥有其他窗口，则这些窗口始终按顺序显示在该窗口前面。如果对拥有其他窗口的某个窗口调用 orderToFront()，或者在该窗口上将 alwaysInFront 设置为 true，则所有者窗口将显示在其他窗口前面，该窗口所拥有的窗口随该窗口一起重新排序，且所拥有的窗口仍显示在所有者的前面。

在同一窗口所拥有的窗口中，对拥有的窗口调用窗口顺序方法可正常工作，但同时会更改整个拥有的窗口组相对于该组之外的其他窗口的排序顺序。例如，如果对某个拥有的窗口调用 orderToFront()，则该窗口、其所有者以及同一所有者所拥有的所有其他窗口都将移动到窗口显示列表的前面。

NativeWindow 类提供以下属性和方法来设置一个窗口相对于其他窗口的显示顺序：

成员	说明
alwaysInFront 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， false 都是最佳设置。将值从 false 更改为 true 会将窗口置于所有其他窗口的前面 （但不会激活该窗口）。将值从 true 更改为 false 会将窗口的顺序排在最上面组中其余窗口的后面，但仍位于其他窗口的前面。将窗口的该属性设置为其当前值不会更改窗口显示顺序。 alwaysInFront 设置对其他窗口所拥有的窗口没有任何影响。
orderToFront()	将窗口置于前面。
orderInFrontOf()	将窗口置于紧靠特定窗口前面。
orderToBack()	将窗口发送到其他窗口后面。
orderBehind()	将窗口发送到紧靠特定窗口后面。
activate()	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

Adobe AIR 1.0 和更高版本

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，但如果其他对象引用了此内容，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。

当所有者关闭时，该所有者所拥有的所有窗口也将同时关闭。所拥有的窗口不会调度关闭事件，因此无法阻止关闭这些窗口。已调度 close 事件。

允许取消窗口操作

Adobe AIR 1.0 和更高版本

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，则该窗口不会关闭。

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。若要在执行窗口更改前通知应用程序中的组件，则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。

例如，以下逻辑实现窗口关闭按钮的可取消事件处理函数：

```
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，由于其他原因，它还可以返回 false，因此最好明确使用 isDefaultPrevented() 方法来测试是否应取消更改。

最大化、最小化和还原窗口

Adobe AIR 1.0 和更高版本

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();
```

当所有者窗口最小化或还原时，所拥有的窗口也将最小化和还原。所拥有的窗口在最小化时不会调度任何事件，因为是对其所有者执行的最小化操作。

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 对象将调度 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 91 页的 “ 侦听窗口事件 ”。

示例：最小化、最大化、还原和关闭窗口

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 85 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 对象将调度 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 91 页的 " 侦听窗口事件 "。

移动窗口

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 事件时完成。

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 API 参考。

若要以编程方式移动窗口，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 对象将调度 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 91 页的“侦听窗口事件”。

示例：调整窗口大小和移动窗口

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 window close() 方法不会自动调度 closing 事件，而只调度 close 事件。但是，您可以构造 closing 事件对象并使用 window dispatchEvent() 方法来调度它。

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。

事件	说明
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

Adobe AIR 1.0 和更高版本

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）。

例如，以下 Flex 代码定义用于设置简单全屏端点的简单 AIR 应用程序：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical"
    applicationComplete="init()" backgroundColor="0x003030" focusRect="false">
    <mx:Script>
        <![CDATA[
            private function init():void
            {
                stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;
                focusManager.setFocus(terminal);
                terminal.text = "Welcome to the dumb terminal app. Press the ESC key to exit..\n";
                terminal.selectionBeginIndex = terminal.text.length;
                terminal.selectionEndIndex = terminal.text.length;
            }
        ]]>
    </mx:Script>
    <mx:TextArea
        id="terminal"
        height="100%" width="100%"
        scroll="false"
        backgroundColor="0x003030"
        color="0xCCFF00"
        fontFamily="Lucida Console"
        fontSize="44"/>
</mx:WindowedApplication>
```

以下用于 Flash 的 ActionScript 示例模拟简单全屏文本端点：

```
import flash.display.Sprite;
import flash.display.StageDisplayState;
import flash.text.TextField;
import flash.text.TextFormat;

public class FullScreenTerminalExample extends Sprite
{
    public function FullScreenTerminalExample():void
    {
        var terminal:TextField = new TextField();
        terminal.multiline = true;
        terminal.wordWrap = true;
        terminal.selectable = true;
        terminal.background = true;
        terminal.backgroundColor = 0x00333333;

        this.stage.displayState = StageDisplayState.FULL_SCREEN_INTERACTIVE;

        addChild(terminal);
        terminal.width = 550;
        terminal.height = 400;

        terminal.text = "Welcome to the dumb terminal application. Press the ESC key to exit.\n_";

        var tf:TextFormat = new TextFormat();
        tf.font = "Courier New";
        tf.color = 0x00CCFF00;
        tf.size = 12;
        terminal.setTextFormat(tf);

        terminal.setSelection(terminal.text.length - 1, terminal.text.length);
    }
}
```

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 8 章：AIR 中的显示屏幕

Adobe AIR 1.0 和更高版本

使用 Adobe® AIR® Screen 类访问关于连接到计算机或设备的显示屏幕的信息。

更多帮助主题

flash.display.Screen

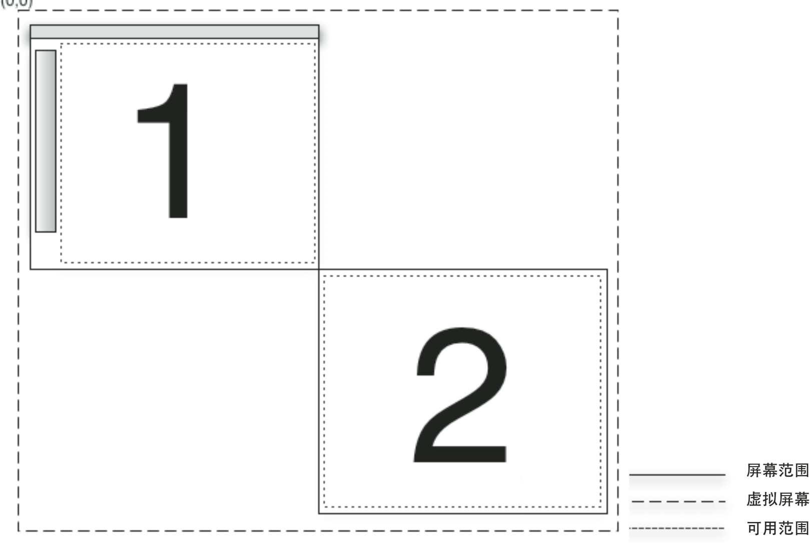
AIR 中的显示屏幕的基础知识

Adobe AIR 1.0 和更高版本

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) 左上角的坐标始终是 (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 API 参考。

枚举屏幕

Adobe AIR 1.0 和更高版本

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。数组的顺序不重要。
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。

不要保存 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。该示例调用 Screen.getScreensForRectangle()，传入窗口范围，以便识别窗口的当前屏幕。

```
<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 9 章：使用菜单

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

在 Adobe® AIR® 中，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。

菜单基础知识

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

有关在 AIR 应用程序中创建本机菜单的快速介绍和代码示例，请参阅 Adobe Developer Connection 上的以下快速入门文章：

- 向 AIR 应用程序添加本机菜单

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。

在 AIR 外，您可以使用上下文菜单类修改上下文菜单，当用户右键单击或按住 Cmd 键单击应用程序的中的对象时， Flash Player 会自动显示上下文菜单。（AIR 应用程序不会自动显示上下文菜单。）

菜单类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

菜单类包括：

包	类
flash.display	• NativeMenu • NativeMenuItem
flash.events	• Event

菜单类型

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

AIR 支持以下类型的菜单：

上下文菜单 右键单击或按住 Command 键单击 SWF 内容中的交互式对象或 HTML 内容中的文档元素时，作为响应，上下文菜单会打开。

在 Flash Player 运行时中，上下文菜单自动显示。您可以使用 ContextMenu 类和 ContextMenuItem 类向此菜单中添加您自己的命令。您也可以删除一些 （非全部）内置命令。

在 AIR 运行时中，可以使用 NativeMenu 类或 ContextMenu 类创建上下文菜单。在 AIR 中的 HTML 内容中，您可以使用 Webkit HTML 和 JavaScript API 向 HTML 元素添加上下文菜单。

应用程序菜单 （仅限 AIR） 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 （仅限 AIR） 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。

停靠栏和系统任务栏图标菜单 （仅限 AIR） 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项目的上方。 Windows 或 Linux 中没有标准菜单。

弹出菜单 （仅限 AIR） 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。通过调用任何 NativeMenu 对象的 display() 方法，可以使弹出菜单在窗口中的任何位置显示。

自定义菜单 本机菜单完全由操作系统调出，因此存在于 Flash 和 HTML 呈现模型以外。您可以不使用本机菜单，而总是使用 MXML、 ActionScript 或 JavaScript 创建自己的自定义非本机菜单 （仅 AIR）。这些菜单必须在应用程序内容中完全呈现。

默认菜单 （仅 **AIR**）

以下默认菜单由操作系统或内置 AIR 类提供：

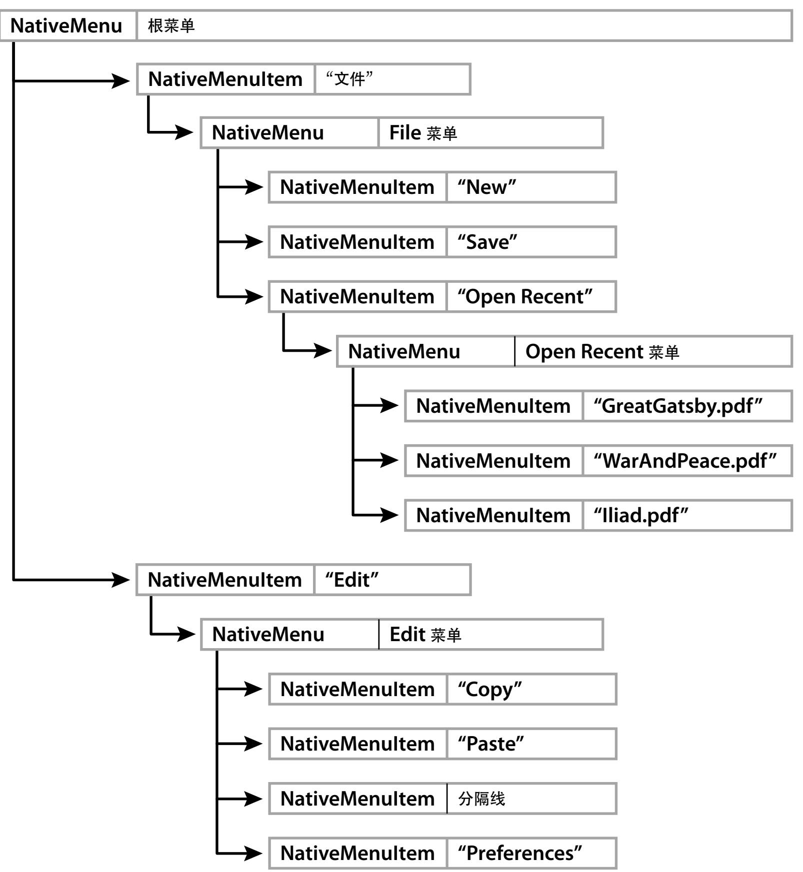
- Mac OS X 上的应用程序菜单
- Mac OS X 上的停靠栏图标菜单
- HTML 内容中的所选文本和图像的上下文菜单
- TextField 对象 （或扩展 TextField 的对象）中的所选文本的上下文菜单

本机菜单结构 (AIR)

Adobe AIR 1.0 和更高版本

本机菜单本质上是分层的。 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 108 页的“本机菜单示例：窗口和应用程序菜单 (AIR)”。

菜单的事件

Adobe AIR 1.0 和更高版本

NativeMenu 和 NativeMenuItem 对象均调度 preparing、 displaying 和 select 事件：

Preparing：每当对象即将开始用户交互时，该菜单及其菜单项会将 preparing 事件调度到任何注册的侦听器。交互包括打开菜单或使用键盘快捷键选择项目。

注：preparing 事件仅适用于 Adobe AIR 2.6 和更高版本。

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。

preparing 和 displaying 事件允许您在向用户显示菜单内容或项目外观之前对其进行更新。例如，在“打开最近的文件”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。

如果您删除了一个其键盘快捷键会触发 preparing 事件的菜单项，则将实际取消菜单交互，并且将不会调度 select 事件。

该事件的 target 和 currentTarget 属性均为在其上注册了侦听器的对象：菜单本身或其中的一个项目。

preparing 事件将在 displaying 事件之前调度。通常，您只会侦听这两个事件中的一个，而不会同时侦听二者。

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 ContextMenuItem 对象调度 menuItemSelect 和 menuSelect 事件以及 select、preparing 和 displaying 事件。

本机菜单命令的等效键 (AIR)

Adobe AIR 1.0 和更高版本

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，是 Control 键 (Keyboard.CONTROL)。这些默认键会自动添加到功能键组中。若要分配其他功能键，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 (AIR)

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 (AIR)

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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() 方法在给定索引处添加项目，否则项目以添加顺序出现在菜单中。

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：

设置应用程序菜单或窗口菜单
您的代码中应包含应用程序菜单 （受 Mac OS 支持）和窗口菜单 （受其他操作系统支持），这很重要

```
var root = new air.NativeMenu();
if (air.NativeApplication.supportsMenu)
{
    air.NativeApplication.nativeApplication.menu = root;
}
else if (NativeWindow.supportsMenu)
{
    nativeWindow.menu = root;
}
```

注： Mac OS 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。

Adobe Flex 提供了 FlexNativeMenu 类，用于方便地创建跨平台工作的菜单。如果您使用的是 Flex 框架，请使用 FlexNativeMenu 类，而不要使用 NativeMenu 类。

设置停靠栏图标菜单或系统托盘图标菜单

```
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。

以弹出方式显示菜单

```
root.display(window.nativeWindow.stage, x, y);
```

更多帮助主题

开发跨平台 AIR 应用程序

创建子菜单

Adobe AIR 1.0 和更高版本

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：

可以使用 addSubmenu() 方法在一个步骤中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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 (AIR)

Adobe AIR 1.0 和更高版本

在使用 HTMLLoader 对象显示的 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，可以侦听 contextmenu 事件并调用事件对象的 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。以下示例通过调用菜单的 display() 方法响应 HTML contextmenu 事件，来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>
```

显示弹出本机菜单 (AIR)

Adobe AIR 1.0 和更高版本

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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

将事件侦听器添加到菜单或个别项目来处理菜单事件。

Object	**调度的事件**
NativeMenu (AIR)	Event.PREPARING（Adobe AIR 2.6 和更高版本） Event.DISPLAYING Event.SELECT（从子项目和子菜单传播）
NativeMenuItem (AIR)	Event.PREPARING（Adobe AIR 2.6 和更高版本） Event.SELECT Event.DISPLAYING（从父菜单传播）

选择菜单事件

Adobe AIR 1.0 和更高版本

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.addEventListener(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.addEventListener(air.Event.SELECT,
                                                              colorChoice);
} else if (air.NativeWindow.supportsMenu){
    var windowMenu = new air.NativeMenu();
    window.nativeWindow.menu = windowMenu;
    windowMenu.addItem(colorMenuItem);
    windowMenu.addEventListener(air.Event.SELECT, colorChoice);
}

function colorChoice(event) {
    var menuItem = event.target;
    air.trace(menuItem.label + " has been selected");
}
```

如果正在使用 ContextMenuItem 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他信息，但不能上升到包含菜单。

显示菜单事件

Adobe AIR 1.0 和更高版本

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。还可以从 ContextMenu 对象侦听 menuSelect 事件。

在 AIR 2.6 和更高版本中，可以使用 preparing 事件来更新菜单，以响应显示菜单或使用键盘快捷键选择项目。

本机菜单示例：窗口和应用程序菜单 (AIR)

Adobe AIR 1.0 和更高版本

以下示例创建在第 100 页的 " 本机菜单结构 (AIR)" 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 (item.submenu == menu) {
                return item;
            }
        }
    }
    return null;
}
</script>
<title>AIR menus</title>
</head>
<body onload="MenuExample()"></body>
</html>
```

使用 MenuBuilder 框架

Adobe AIR 1.0 和更高版本

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 99 页的 " 菜单基础知识 "。

使用 MenuBuilder 框架创建菜单

Adobe AIR 1.0 和更高版本

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，就可以用其他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

Adobe AIR 1.0 和更高版本

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）。对于其他菜单类型，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 113 页的 " 定义 MenuBuilder 菜单结构 "。

2 加载菜单结构：调用合适的 Menu 类方法 （Menu.createFromXML() 或 Menu.createFromJSON()）以加载菜单结构文件，并将其转换为实际的菜单对象。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()，将该菜单显示为 DOM 元素的上下文菜单
- Menu.setAsIconMenu()，将该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，而不是在 onload 或其他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() 调用都必须在创建此 DOM 元素之后发生。最安全的方法是将包含菜单分配代码的 <script> 块放在位于 HTML 页末尾的结束标记 </body> 以内。

加载菜单结构

Adobe AIR 1.0 和更高版本

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()，可以加载和分析 XML 格式的菜单结构文件
- Menu.createFromJSON()，可以加载和分析 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 113 页的 " 定义 MenuBuilder 菜单结构 "。

创建应用程序菜单或窗口菜单

Adobe AIR 1.0 和更高版本

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他项目。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() 方法将菜单设置为执行调用时所在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上， setAsMenu() 调用将此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() 方法将 NativeMenu 对象指定为应用程序菜单时， NativeMenu 中的项目将插入到标准菜单结构中的 "Edit" （编辑）菜单和 "Window" （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

Adobe AIR 1.0 和更高版本

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() 方法将该菜单指定为 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。若要将 NativeMenu 指定为 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

Adobe AIR 1.0 和更高版本

除了应用程序窗口中的 DOM 元素的上下文菜单以外， Adobe AIR 应用程序还支持其他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之一，请首先使用 Menu.createFromXML() 或 Menu.createFromJSON() 方法创建 NativeMenu。然后，通过调用 Menu.setAsIconMenu() 方法，将 NativeMenu 指定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

Adobe AIR 1.0 和更高版本

使用 Menu.createFromXML() 或 Menu.createFromJSON() 方法创建 NativeMenu 对象时，XML 元素或对象的结构将定义所产生的菜单的结构。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

Adobe AIR 1.0 和更高版本

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。或者，如果该项目为子项目，则该菜单项会调度一个 preparing 事件，然后调度一个 displaying 事件，最后打开子菜单。
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，并调用在数据源的 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他 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，以包含由其他对象组成的数组。

菜单数据源属性 (attribute) 或属性 (property)

Adobe AIR 1.0 和更高版本

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。
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 118 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 框架将其他所有值、或没有 type 条目的元素或对象均视为正常菜单条目。

MenuBuilder 框架忽略其他所有对象属性 (property) 或 XML 属性 (attribute)。

示例：XML MenuBuilder 数据源

Adobe AIR 1.0 和更高版本

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 116 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="SubMenuItem D-1"/>
        <menuitem label="SubMenuItem D-2"/>
        <menuitem label="SubMenuItem 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

Adobe AIR 1.0 和更高版本

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 115 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: "SubMenuItem D-1"},
            {label: "SubMenuItem D-2"},
            {label: "SubMenuItem 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 102 页的 " 本机菜单命令的等效键 (AIR)"。

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

Adobe AIR 1.0 和更高版本

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 102 页的 “ 助记键 (AIR)”。

对于 MenuBuilder 框架，则指定菜单项助记键的最简单方式是在菜单项的 label 字段中包括下划线字符 (_)。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包括文本下划线字符，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

Adobe AIR 1.0 和更高版本

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 101 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。由 <ul> 和 <li> 元素所表示的项目列表的上下文菜单使用 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);
            }

            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>
    <menuitem label="File">
        <menuitem label="New" keyEquivalent="n" onSelect="fileMenuCommand"/>
        <menuitem label="Open" keyEquivalent="o" onSelect="fileMenuCommand"/>
        <menuitem label="Save" keyEquivalent="s" onSelect="fileMenuCommand"/>
        <menuitem label="Save As..." keyEquivalent="S" onSelect="fileMenuCommand"/>
        <menuitem label="Close" keyEquivalent="w" onSelect="fileMenuCommand"/>
    </menuitem>
    <menuitem label="Edit">
        <menuitem label="Cut" keyEquivalent="x" onSelect="editMenuCommand"/>
        <menuitem label="Copy" keyEquivalent="c" onSelect="editMenuCommand"/>
        <menuitem label="Paste" keyEquivalent="v" onSelect="editMenuCommand"/>
    </menuitem>
</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(", "), "]"].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 10 章 : AIR 中的任务栏图标

Adobe AIR 1.0 和更高版本

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。

更多帮助主题

flash.desktop.NativeApplication

flash.desktop.DockIcon

flash.desktop.SystemTrayIcon

关于任务栏图标

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 事件将传送到正在运行的应用程序实例。

系统任务栏图标

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。）

注：使用 Flex WindowedApplication 组件时，必须将 WindowedApplication 标记的 visible 属性设置为 false。该属性取代了应用程序描述符中的设置。

```
<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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 123 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

Adobe AIR 1.0 和更高版本

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。如果只是需要使初始窗口变得不可见，请向应用程序描述符文件的 <initalWindow> 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 11 章：使用文件系统

Flash Player 9 和更高版本，**Adobe AIR 1.0** 和更高版本

Adobe® AIR® 文件系统 API 提供了对主机的文件系统的完全访问权限。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。

更多帮助主题

flash.filesystem.File

flash.filesystem.FileStream

使用 AIR 文件系统 API

Adobe AIR 1.0 和更高版本

Adobe AIR 文件系统 API 包括以下类：

- 文件
- FileMode
- FileStream

借助该文件系统 API，您可以执行以下操作 （以及其他更多操作）：

- 复制、创建、删除和移动文件和目录
- 获取有关文件和目录的信息
- 读写文件

AIR 文件基础知识

Adobe AIR 1.0 和更高版本

有关在 AIR 中使用文件系统的快速介绍和代码示例，请参阅 Adobe Developer Connection 中的以下快速入门文章：

- 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

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，以便您可以打开该文件并读取或写入数据。

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() 和 File.getDirectoryListingAsync()
- File.moveTo() 和 File.moveToAsync()
- File.moveToTrash() 和 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。使用操作的异步版本时，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，可能会延迟重要的进程 （如显示对象呈现和动画）。

使用 AIR 中的 File 对象

Adobe AIR 1.0 和更高版本

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。 Adobe® Flash® Player 和 AIR 中提供的 FileReference 类表示指向文件的指针。 File 类添加了一些属性和方法，出于安全方面的考虑，在 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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 134 页的 " 支持的 AIR 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。此目录适用于存储动态应用程序资源和用户首选项。考虑在其他位置存储大量数据。
- File.applicationDirectory — 安装应用程序的目录 （还存储任何安装的资源）。在有些操作系统上，应用程序存储在一个软件包文件中而不是物理目录中。在这种情况下，可能无法使用本机路径访问内容。应用程序目录是只读的。
- File.desktopDirectory— 用户的桌面目录。如果平台不定义桌面目录，则使用文件系统上的另一个位置。
- File.documentsDirectory— 用户的文档目录。如果平台不定义文档目录，则使用文件系统上的另一个位置。
- File.userDirectory — 用户目录。如果平台不定义用户目录，则使用文件系统上的另一个位置。

注：当平台不定义桌面、文档或用户目录的标准位置时， File.documentsDirectory、 File.desktopDirectory 和 File.userDirectory 可以引用同一个目录。

这些属性在不同的操作系统上有不同的值。例如，对于用户的桌面目录， Mac 和 Windows 有各自不同的本机路径。然而， File.desktopDirectory 属性在每个平台上指向适当的目录路径。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他目录和文件时，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");
```

尽管可以使用 File 类来指向特定文件路径，但这种做法会指向无法跨平台工作的应用程序。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。

公用目录位置

平台	目录类型	典型的文件系统位置
Linux	应用程序	/opt/filename/share
	应用程序存储	/home/userName/.appdata/applicationID/Local Store
	桌面	/home/userName/Desktop
	文档	/home/userName/Documents
	临时	/tmp/FlashTmp.randomString
	用户	/home/userName
Mac	应用程序	/Applications/filename.app/Contents/Resources
	应用程序存储	/Users/userName/Library/Preferences/applicationID/Local Store
	桌面	/Users/userName/Desktop
	文档	/Users/userName/Documents
	临时	/private/var/folders/JY/randomString/TemporaryItems/FlashTmp
	用户	/Users/userName
Windows	应用程序	C:\Program Files\filename
	应用程序存储	C:\Documents and settings\userName\ApplicationData\applicationID\Local Store
	桌面	C:\Documents and settings\userName\Desktop
	文档	C:\Documents and settings\userName\My Documents
	临时	C:\Documents and settings\userName\Local Settings\Temp\randomString.tmp
	用户	C:\Documents and settings\userName

根据具体的操作系统和计算机配置，这些目录的实际本机路径会有所不同。此表中显示的路径是典型示例。您应该始终使用适当的静态 File 类属性引用这些目录，以便您的应用程序在任何平台上都能正常工作。在一个实际的 AIR 应用程序中，表中显示的 applicationID 和 filename 的值取自应用程序描述符。如果您在应用程序描述符中指定发布者 ID，则发布者 ID 在这些路径中会追加到应用程序 ID。 userName 的值是安装用户的帐户名称。

将 File 对象指向目录

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

您可以将 File 对象指向用户的主目录。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

Adobe AIR 1.0 和更高版本

您可以将 File 对象指向用户的文档目录。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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。您可以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");
```

应用程序存储目录位置通常基于用户名称和应用程序 ID。此处提供了下列文件系统位置以帮助您调试应用程序。您应该始终使用 File.applicationStorage 属性或 app-storage: URI 方案解析此目录中的文件：

- 在 Mac OS 中，位于：

  /Users/user name/Library/Preferences/applicationID/Local Store/

  例如：

  ```
  /Users/babbage/Library/Preferences/com.example.TestApp/Local Store
  ```

- 在 Windows 中，位于 Documents and Settings 目录下的以下位置：

  C:\Documents and Settings\user name\Application Data\applicationID\Local Store\

  例如：

  ```
  C:\Documents and Settings\babbage\Application Data\com.example.TestApp\Local Store
  ```

- 在 Linux 中位于：

  /home/user name/.appdata/applicationID/Local Store/

  例如：

  ```
  /home/babbage/.appdata/com.example.TestApp/Local Store
  ```

注：如果应用程序具有发行商 ID，则还可将该 ID 用作应用程序存储目录路径的一部分。

通过 File.applicationStorageDirectory 创建的 File 对象的 URL （和 url 属性）将使用 app-storage URL 方案 （请参阅第 134 页的 " 支持的 AIR 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

Adobe AIR 1.0 和更高版本

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 File.applicationDirectory 创建的 File 对象的 URL （和 url 属性）将使用 app URL 方案 （请参阅第 134 页的 “ 支持的 AIR 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

Adobe AIR 1.0 和更高版本

File.getRootDirectories() 方法列出所有根卷，如 Windows 计算机中的 C: 和已装好的卷。在 Mac OS 和 Linux 中，此方法始终返回计算机的唯一根目录 （“/” 目录）。 StorageVolumeInfo.getStorageVolumes() 方法提供有已关装的存储卷的更多详细信息 （请参阅第 142 页的 “ 使用存储卷 ”）。

指向明确的目录

Adobe AIR 1.0 和更高版本

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。您可以使用 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）导航到相对路径。

导航到相对路径

Adobe AIR 1.0 和更高版本

您可以使用 resolvePath() 方法获取相对于其他给定路径的路径。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 133 页的 “ 修改文件路径 ”。

让用户浏览以选择目录

Adobe AIR 1.0 和更高版本

File 类包括 browseForDirectory() 方法，它表示系统对话框，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，File 对象将调度一个 select 事件；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

Adobe AIR 1.0 和更高版本

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 255 页的 “ 捕获命令行参数 ”。

将 File 对象指向文件

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。您可以使用 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 133 页的 “ 修改文件路径 ”）导航到相对路径。

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;
```

您还可以将 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 URL 的 URI 编码版本 （例如，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";
```

您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 133 页的 “ 修改文件路径 ”。

枚举目录中的文件

Adobe AIR 1.0 和更高版本

您可以使用 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 138 页的 " 枚举目录 "。

让用户浏览以选择文件

Adobe AIR 1.0 和更高版本

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。当用户选择一个文件时（或者，对于 browseForSave() 选择一个目标路径时），browseForOpen() 和 browseForSave() 方法将调度 select 事件。对 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

Adobe AIR 1.0 和更高版本

通过调用 resolvePath() 方法或通过修改对象的 nativePath 或 url 属性，您还可以修改现有 File 对象的路径，如以下示例中所示（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 AIR URL 方案

Adobe AIR 1.0 和更高版本

在 AIR 中定义 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： `file:///c:/AIR Test/test.txt` URL 标准规定 file URL 采用 file://<host>/<path> 形式。作为一个特例， <host> 可以是空字符串，它被解释为 " 解释该 URL 的计算机 "。因此， file 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 application.xml 文件）的路径。例如，以下路径指向所安装应用程序的目录的 images 子目录： `app:/images`
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 settings 子目录中的 prefs.xml 文件： `app-storage:/settings/prefs.xml`

查找两个文件之间的相对路径

Adobe AIR 1.0 和更高版本

您可以使用 getRelativePath() 方法查找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 useDotDot 允许在结果中返回 .. 语法，以指示父目录：

```
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

Adobe AIR 1.0 和更高版本

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() 方法可以转换 nativePath 对象，以使用文件名或目录名的正确大写形式。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，您还可以使用 canonicalize() 方法将短文件名 （"8.3" 名称）转换为长文件名，如以下示例中所示：

```
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

Adobe AIR 1.0 和更高版本

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，以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

Adobe AIR 1.0 和更高版本

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 143 页的 " 读取和写入文件 "。

StorageVolumeInfo.getStorageVolumes() 方法提供有已关装的存储卷的更多详细信息 （请参阅第 142 页的 " 使用存储卷 "）。

使用默认系统应用程序打开文件

Adobe AIR 2 和更高版本

在 AIR 2 中，您可以使用操作系统注册的用来打开某文件的应用程序打开该文件。例如，AIR 应用程序可以使用注册的用来打开文档文件的应用程序打开一个文档文件。使用 File 对象的 openWithDefaultApplication() 方法打开该文件。例如，以下代码打开用户桌面上名为 test.doc 的文件，并且打开该文件所用的是与文档文件相对应的默认应用程序：

```
var file = air.File.deskopDirectory;
file = file.resolvePath("test.doc");
file.openWithDefaultApplication();
```

注：在 Linux 中，文件的 MIME 类型 （而不是文件扩展名）确定文件的默认应用程序。

以下代码使用户可以导航到一个 mp3 文件，并在用于播放 mp3 文件的默认应用程序中打开它：

```
var file = air.File.documentsDirectory;
var mp3Filter = new air.FileFilter("MP3 Files", "*.mp3");
file.browseForOpen("Open", [mp3Filter]);
file.addEventListener(Event.SELECT, fileSelected);
function fileSelected(event)
{
    file.openWithDefaultApplication();
}
```

无法对位于应用程序目录中的文件使用 openWithDefaultApplication() 方法。

AIR 会阻止您使用 openWithDefaultApplication() 方法打开某些文件。在 Windows 中， AIR 会阻止您打开某些文件类型的文件，例如 EXE 或 BAT。在 Mac OS 和 Linux 中，AIR 阻止您打开将在某些应用程序中启动的文件。(其中包括 Mac OS 中的 Terminal 和 AppletLauncher 以及 Linux 中的 csh、bash 或 ruby。) 尝试使用 openWithDefaultApplication() 方法打开其中一个文件将导致异常。有关阻止打开的文件类型的完整列表，请参阅 File.openWithDefaultApplication() 方法的语言参考条目。

注：对于使用本机安装程序 （一种扩展桌面应用程序）安装的 AIR 应用程序不存在这种限制。

获取文件系统信息

Adobe AIR 1.0 和更高版本

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
File.lineEnding	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。
File.separator	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。
File.systemCharset	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
Capabilities.hasIME	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。
Capabilities.language	指定运行播放器的系统的语言代码。
Capabilities.os	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，请看以下代码：

```
var separator:String;
if (Capablities.os.indexOf("Mac") > -1)
{
    separator = "/";
}
else
{
    separator = "\\";
}
```

此代码会导致 Linux 上出现问题。最好只使用 File.separator 属性。

使用目录

Adobe AIR 1.0 和更高版本

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 129 页的 “ 将 File 对象指向目录 ”。

创建目录

Adobe AIR 1.0 和更高版本

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， 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 143 页的 " 读取和写入文件的工作流程 "。

创建临时目录

Adobe AIR 1.0 和更高版本

File 类包括一个 createTempDirectory() 方法，该方法可在系统的临时目录文件夹中创建一个目录，如以下示例中所示：

```
var temp = air.File.createTempDirectory();
```

createTempDirectory() 方法会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临时目录，因为不会 在所有设备上自动删除临时目录。

枚举目录

Adobe AIR 1.0 和更高版本

您可以使用 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

Adobe AIR 1.0 和更高版本

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，现有目标目录中的所有文件和文件夹都将删除，并替换为源目录中的文件和文件夹 （即使在源目录中目标文件不存在）。

指定为 copyTo() 方法的 newLocation 参数的目录将指定所生成的目录的路径，它不 指定将包含所生成的目录的父 目录。

有关详细信息，请参阅第 140 页的 " 复制和移动文件 "。

删除目录内容

Adobe AIR 1.0 和更高版本

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 126 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，您可以使用这些方法将目录移到系统垃圾桶。有关详细信息，请参阅第 141 页的 " 将文件移到垃圾桶 "。

使用文件

Adobe AIR 1.0 和更高版本

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 60 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 261 页的 " 管理文件关联 "。

获取文件信息

Adobe AIR 1.0 和更高版本

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。）
downloaded	（AIR 2 和更高版本）指示是否已 （从 Internet）下载引用的文件或目录。属性仅在文件可以标记为已下载的操作系统上有意义： • Windows XP Service Pack 2 和更高版本，在 Windows Vista 上 • Mac OS 10.5 和更高版本
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 128 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 128 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 API 参考中的 File 类条目。

复制和移动文件

Adobe AIR 1.0 和更高版本

File 类包括两个用于复制文件或目录的方法：copyTo() 和 copyToAsync()。 File 类包括两个用于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 126 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 overwrite 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() 方法可以监视任务是否完成或错误条件，如以下代码中所示：

```
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, fileMoveIOErrorEventHandler);
original.moveToAsync(destination);

function fileMoveCompleteHandler(event){
    alert(event.target); // [object File]
}
function fileMoveIOErrorEventHandler(event) {
    alert("I/O Error.");
}
```

File 类还包括 File.moveToTrash() 和 File.moveToTrashAsync() 方法，它们将文件或目录移到系统垃圾桶。

删除文件

Adobe AIR 1.0 和更高版本

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 126 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，您可以使用这些方法将文件或目录移到系统垃圾桶。有关详细信息，请参阅第 141 页的 “ 将文件移到垃圾桶 ”。

将文件移到垃圾桶

Adobe AIR 1.0 和更高版本

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 126 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();
```

注：在不支持可恢复垃圾桶文件夹概念的操作系统上，会立即删除文件。

创建临时文件

Adobe AIR 1.0 和更高版本

File 类包括一个 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() 方法会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唯一的临时目录。

您可能需要在关闭应用程序前删除临时文件，因为不会 在所有设备上自动删除临时文件。

使用存储卷

Adobe AIR 2 和更高版本

在 AIR 2 中，安装或卸载大容量存储卷时，您可以进行检测。 StorageVolumeInfo 类定义单个 storageVolumeInfo 对象。StorageVolumeInfo.storageVolumeInfo 对象在存储卷安装之后调度 storageVolumeMount 事件。它在某个卷被卸载时调度 storageVolumeUnmount 事件。 StorageVolumeChangeEvent 类定义这些事件。

注：在现今的 Linux 发行版中， StorageVolumeInfo 对象仅对在特定位置装载的物理设备和网络驱动器调度 storageVolumeMount 和 storageVolumeUnmount 事件。

StorageVolumeChangeEvent 类的 storageVolume 属性是一个 StorageVolume 对象。 StorageVolume 类定义存储卷的基本属性：

- drive — Windows 上的卷驱动器号 （在其他操作系统上为 null）
- fileSystemType — 存储卷上的文件系统的类型 （如 “FAT”、 “NTFS”、 “HFS” 或 “UFS”）
- isRemoveable — 卷是可删除 （为 true）还是不可删除 （为 false）
- isWritable — 卷可写入 (true) 还是不可写入 (false)
- name — 卷的名称
- rootDirectory — 与卷的根目录对应的 File 对象

StorageVolumeChangeEvent 类还包括一个 rootDirectory 属性。rootDirectory 属性是一个引用已装载或已卸载的存储卷的根目录的 File 对象。

对于卸载的卷，未定义 StorageVolumeChangeEvent 对象的 storageVolume 属性 (null)。然而，您可以访问此事件的 rootDirectory 的属性。

以下代码在安装完存储卷之后输出该存储卷的名称和文件路径：

```
air.StorageVolumeInfo.storageVolumeInfo.addEventListener(air.StorageVolumeChangeEvent.STORAGE_VOLUME_MOUN
T, onVolumeMount);
function onVolumeMount(event)
{
    air.trace(event.storageVolume.name, event.rootDirectory.nativePath);
}
```

以下代码在卸载完存储卷之后输出该存储卷的文件路径：

```
air.StorageVolumeInfo.storageVolumeInfo.addEventListener(air.StorageVolumeChangeEvent.STORAGE_VOLUME_UNMO
UNT, onVolumeUnmount);
function onVolumeUnmount(event)
{
    air.trace(event.rootDirectory.nativePath);
}
```

StorageVolumeInfo.storageVolumeInfo 对象包含 getStorageVolumes() 方法。此方法返回与当前安装的存储卷对应的 StorageVolume 对象的矢量。以下代码显示如何列出所有安装的存储卷的名称和根目录：

```
var volumes = air.StorageVolumeInfo.storageVolumeInfo.getStorageVolumes();
for (i = 0; i < volumes.length; i++)
{
    air.trace(volumes[i].name, volumes[i].rootDirectory.nativePath);
}
```

注：在现今的 Linux 发行版中， getStorageVolumes() 方法返回与在特定位置安装的物理设备和网络驱动器相对应的对象。

File.getRootDirectories() 方法列出根目录 （请参阅第 131 页的 " 指向文件系统根目录 "）。然而， StorageVolume 对象 （由 StorageVolumeInfo.getStorageVolumes() 方法枚举）提供有关存储卷的详细信息。

可使用 StorageVolume 对象的 rootDirectory 属性的 spaceAvailable 属性查看存储卷上的可用空间。（请参阅第 136 页的 " 确定卷上的可用空间 "。）

更多帮助主题

StorageVolume

StorageVolumeInfo

读取和写入文件

Adobe AIR 1.0 和更高版本

FileStream 类允许 AIR 应用程序读取和写入文件系统。

读取和写入文件的工作流程

Adobe AIR 1.0 和更高版本

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。
File 对象表示您要使用的文件 （或您以后将创建的文件）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 132 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 File 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 145 页的 " 初始化 FileStream 对象以及打开和关闭文件 " 和 第 144 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各种情形下由 FileStream 对象调度的事件。这些情形包括：从文件读取数据时、遇到 I/O 错误时、要写入的全部数据量已写入时。

有关详细信息，请参阅第 148 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，您可以使用 readUTFBytes() 和 writeUTFBytes() 方法。如果您希望将数据作为字节数组处理，您可以使用 readByte()、 readBytes()、 writeByte() 和 writeBytes() 方法。有关详细信息，请参阅第 149 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 147 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 136 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。
调用 close() 方法可使文件对其他应用程序可用。

有关详细信息，请参阅第 145 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
- 构建文本文件编辑器
- 构建文本文件编辑器
- 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

使用 FileStream 对象

Adobe AIR 1.0 和更高版本

FileStream 类定义打开、读取和写入文件的方法。

FileStream 打开模式

Adobe AIR 1.0 和更高版本

FileStream 对象的 open() 和 openAsync() 方法都包括一个 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 （您可以将这些模式指定为 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，只有写入的字节会覆盖现有字节 （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

Adobe AIR 1.0 和更高版本

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 144 页的 “FileStream 打开模式 ”。

如果您使用 openAsync() 方法打开文件进行异步文件操作，请设置事件侦听器以处理异步事件：

```
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您使用 open() 方法还是 openAsync() 方法。有关详细信息，请参阅 第 126 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 147 页的 “ 读取缓冲区和 FileStream 对象的 bytesAvailable 属性 ”。

您可以调用 FileStream 对象的 close() 方法关闭关联文件，使其他应用程序可以使用该文件。

FileStream 对象的 position 属性

Adobe AIR 1.0 和更高版本

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：

- 当您明确设置 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)
{
    air.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，即使写入操作完成后 position 仍设置为该值 (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

Adobe AIR 1.0 和更高版本

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。当您打开文件后， FileStream 对象立即开始将数据读取到缓冲区中 （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），您始终可以设置 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），如以下代码中所示 （假定该文件包含至少 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 bytesAvalable 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 progress 事件指示的一系列异步缓冲区填充数据中，从 position 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

Adobe AIR 1.0 和更高版本

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，您可以调用 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

Adobe AIR 1.0 和更高版本

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，然后将它存储到字符串，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");
}
```

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”）。Adobe AIR 支持常见的字符集编码 （请参阅支持的字符集）。

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 字符集中字符的长度是可变的，请不要在响应 progress 事件的方法中使用 readUTFBytes()，这是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 151 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。 Adobe AIR、 Flash Player、 Flash Media Server 和 Flex Data Services 包括用于此格式数据的 API。

还有一些其他读取和写入方法 （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 151 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 和更高版本， **Adobe AIR 1.5** 和更高版本

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

Adobe AIR 1.0 和更高版本

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 149 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();
}
```

在将整个文件读入到读取缓冲区时 （当 FileStream 对象调度 complete 事件时），将调用 processXMLData() 方法。它调用 readUTFBytes() 方法以获取所读数据的字符串版本，然后它基于该字符串创建一个 XML 对象 prefsXML。

要查看演示这些功能的示例应用程序，请参阅从 XML 首选参数文件中读取和写入。

要查看演示这些功能的示例应用程序，请参阅从 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

Adobe AIR 1.0 和更高版本

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。

当访问文件以进行随机读取 / 写入访问时，将 FileMode.UPDATE 指定为 open() 或 openAsync() 方法的 fileMode 参数很重要。

```
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。

当读取数据后 （在调度 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 0 (String.fromCharCode(0)) 填充。

第 12 章：AIR 中的拖放

Adobe AIR 1.0 和更高版本

使用 Adobe® AIR™ 拖放 API 中的类可以支持在用户界面上执行拖放手势。此处所说的手势 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 手势。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 手势。

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

HTML 中的拖放

Adobe AIR 1.0 和更高版本

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 的应用程序中不经常使用，因此在针对 HTML 开发人员的 Adobe AIR API 参考中未提供。有关使用这些类的更多信息，请参阅 Adobe ActionScript 3.0 开发人员指南和用于 Adobe Flash Platform 的 ActionScript 3.0 参考。

默认的拖放行为

Adobe AIR 1.0 和更高版本

HTML 环境为涉及文本、图像和 URL 的拖放手势提供了默认行为。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 来控制任何 HTML 元素的选择方式。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 属性控制元素作为一个整体是否可以拖动。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 13 页的“AIR 中的 CSS”。

HTML 中的拖放事件

Adobe AIR 1.0 和更高版本

从中发起拖动操作的启动器元素调度的事件有：

事件	说明
dragstart	在用户启动拖动手势时调度。如有必要，此事件的处理函数可以通过调用事件对象的 preventDefault() 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 effectAllowed 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 dataTransfer 属性为拖动手势设置其他数据。
drag	在执行拖动手势期间持续调度
dragend	在用户释放鼠标按键终止拖动手势时调度。

由拖动目标调度的事件有：

事件	说明
dragover	当拖动手势仍然在元素边界内时持续调度。此事件的处理函数应设置 dataTransfer.dropEffect 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。
dragenter	在拖动手势进入元素的边界内时调度。 如果在 dragenter 事件处理函数中更改 dataTransfer 对象的任何属性，则这些更改将很快被下一个 dragover 事件覆盖。另一方面，在 dragenter 与第一个 dragover 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。
dragleave	在拖动手势离开元素边界时调度。
drop	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
effectAllowed	拖动源允许的效果。通常情况下， dragstart 事件的处理函数设置此值。请参阅 第 155 页的 “HTML 中的拖动效果 ”。
dropEffect	由目标或用户选择的效果。如果在 dragover 或 dragenter 事件处理函数中设置 dropEffect，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 dropEffect 设置匹配，则不允许放置，并显示不可用 光标。如果尚未设置 dropEffect 来响应最新的 dragover 或 dragenter 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。 最终的效果由为 dragend 调度的对象的 dropEffect 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 dropEffect 将设置为 none。
类型	一个数组，其中包含了 dataTransfer 对象中存在的每种数据格式的 MIME 类型字符串。
getData(mimeType)	以 mimeType 参数指定的格式获取数据。 只能为响应 drop 事件调用 getData() 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。可以通过对每种 MIME 类型调用 setData()，以多种格式添加数据。将清除由默认拖动行为放入 dataTransfer 对象的所有数据。 只能为响应 dragstart 事件调用 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。只能为响应 dragstart 事件，且只有当拖动整个 HTML 元素 （通过将 -webkit-user-drag CSS 样式设置为 element）时，才可以调用 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

Adobe AIR 1.0 和更高版本

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，请参阅第 159 页的 “ 在非应用程序 HTML 沙箱中处理文件放置 ”。

HTML 中的拖动效果

Adobe AIR 1.0 和更高版本

拖动手势的启动器可以通过在 dragstart 事件的处理函数中设置 dataTransfer.effectAllowed 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"	使用指回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。

拖动手势的目标可以设置 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

Adobe AIR 1.0 和更高版本

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，以便将您自己的数据放入拖动手势。

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，则请将此属性设置为 "copyLink"。

设置拖动的数据

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

使用 dataTransfer 属性在 dragstart 事件的处理函数中为拖动手势添加数据。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

Adobe AIR 1.0 和更高版本

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，支持页面上的替代拖入行为。

允许拖入

Adobe AIR 1.0 和更高版本

若要处理拖入手势，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

Adobe AIR 1.0 和更高版本

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手势。

示例：覆盖默认的 HTML 拖入行为

Adobe AIR 1.0 和更高版本

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手势的关键步骤，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.air.bitmap") > -1){
            imageCell.appendChild(event.dataTransfer.getData("image/x-vnd.adobe.air.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 <a href="http://www.adobe.com">URL.</a></li>
           <li><img src="icons/AIRApp_16.png" alt="An image"/></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> </td><td> </td><td> </td><td> </td><td> </td></tr>
       </table>
       </div>
</div>
</body>
</html>
```

在非应用程序 HTML 沙箱中处理文件放置

Adobe AIR 1.0 和更高版本

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和数据类型不同。例如，HTML 事件 dataTransfer 属性的用途与 ActionScript 事件 clipboard 属性相同。有关使用这些类的更多信息，请参阅 Adobe ActionScript 3.0 开发人员指南和用于 Adobe Flash Platform 的 ActionScript 3.0 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 事件处理函数中调用 Event 对象的 preventDefault() 方法来提供有效的放置目标。否则，放置事件绝对不会发生。

```
<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>
```

放置文件释放

Adobe AIR 2 和更高版本

文件释放是一种拖放剪贴板格式，这种格式允许用户将尚不存在的文件拖出 AIR 应用程序外。例如，使用文件释放，您的应用程序使用户可将代理图标拖动到桌面文件夹中。代理图标表示 URL 上已知并可用的文件或一些数据。在用户放置图标后，运行时将下载数据并将文件写入放置位置。

可使用 AIR 应用程序中的 URLFilePromise 类来拖放 URL 上可访问的文件。在 aircore 库中， URLFilePromise 实现作为 AIR 2 SDK 的一部分提供。使用包含在 SDK frameworks/libs/air 目录中的 aircore.swc 或 aircore.swf 文件。

或者，您可以使用 IFilePromise 接口 （在运行时 flash.desktop 包中定义）实现自己的文件承诺逻辑。

文件释放在概念上类似于在剪贴板上使用数据处理函数的延迟呈现。在拖放文件时使用文件释放而不使用延迟呈现。当生成或下载数据时，使用延迟呈现技术可能会导致拖动手势出现不需要的暂停。使用延迟呈现执行复制粘贴操作 （文件释放不支持此操作）。

使用文件释放时的限制

与您可以放在拖放剪贴板中的其他数据格式相比，文件释放有以下限制：

- 文件释放只能拖出 AIR 应用程序；而不能拖入 AIR 应用程序中。
- 并非所有操作系统都不支持文件释放。使用 Clipboard.supportsFilePromise 属性测试主机系统是否支持文件释放。在不支持文件释放的系统中，您应提供替代机制，以便下载或生成文件数据。
- 文件释放不能与复制粘贴剪贴板 (Clipboard.generalClipboard) 一起使用。

更多帮助主题

flash.desktop.IFilePromise

air.desktop.URLFilePromise

放置远程文件

Adobe AIR 2 和更高版本

使用 URLFilePromise 类创建表示 URL 上可用文件或数据的文件释放对象。使用 FILE_PROMISE_LIST 剪贴板格式将一个或多个文件释放对象添加到剪贴板。在以下示例中，从 http://www.example.com/foo.txt 下载一个单独的文件，并作为 bar.txt 保存到放置位置。（远程和本地文件名不必匹配。）

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script src="aircore.swf" type="application/x-shockwave-flash"></script>
<script language="javascript">
    function init(){
        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
    }

    function dragStartHandler(event){
        event.preventDefault();
        startDrag();
    }
    function startDrag()
    {
        var filePromise = new air.URLFilePromise(); //defined in aircore.swf
        filePromise.request = new air.URLRequest("http://example.com/foo.txt");
        filePromise.relativePath = "bar.txt";
        var fileList = new Array( filePromise );
        var clipboard = new air.Clipboard();
        clipboard.setData( air.ClipboardFormats.FILE_PROMISE_LIST_FORMAT, fileList );
        air.NativeDragManager.doDrag( window.htmlLoader, clipboard );
    }
</script>
</head>
<body onLoad="init()">
    <p id="source" style="-webkit-user-drag:element; -webkit-user-select:none;">
        Drag to file system
    </p>
</body>
</html>
```

通过将多个文件释放对象添加到分配给剪贴板的数组，您可以允许用户一次拖动多个文件。您还可以在 relativePath 属性中指定子目录，以便将操作中包括的部分文件或所有文件放置到子文件夹 （相对于放置位置）中。

以下示例演示如何启动包括多个文件释放的拖动操作。在此示例中，HTML 页面 article.html 作为文件释放与两个链接的图像文件一起放置在剪贴板上。这些图像被复制到 images 子文件夹中，以便保持相对链接。

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script src="aircore.swf" type="application/x-shockwave-flash"></script>
<script language="javascript">
    function init(){
        var source = document.getElementById('source');
        source.addEventListener("dragstart", dragStartHandler);
    }

    function dragStartHandler(event){
        event.preventDefault();
        startDrag();
    }
    function startDrag()
    {
        var filePromise = new air.URLFilePromise();
        filePromise.request = new air.URLRequest("http://example.com/article.html");
        filePromise.relativePath = "article.html";

        var image1Promise = new air.URLFilePromise();
        image1Promise.request = new air.URLRequest("http://example.com/images/img_1.jpg");
        image1Promise.relativePath = "images/img_1.html";
        var image2Promise = new air.URLFilePromise();
        image2Promise.request = new air.URLRequest("http://example.com/images/img_2.jpg");
        image2Promise.relativePath = "images/img_2.jpg";

        //Put the promise objects onto the clipboard inside an array
        var fileList = new Array( filePromise, image1Promise, image2Promise );
        var clipboard = new air.Clipboard();
        clipboard.setData( air.ClipboardFormats.FILE_PROMISE_LIST_FORMAT, fileList );
        air.NativeDragManager.doDrag( window.htmlLoader, clipboard );
    }
</script>
</head>
<body onLoad="init()">
    <p id="source" style="-webkit-user-drag:element; -webkit-user-select:none;">
        Drag to file system
    </p>
</body>
</html>
```

实现 IFilePromise 接口

Adobe AIR 2 和更高版本

若要为不能使用 URLFilePromise 对象访问的资源提供文件释放，可以在自定义类中实现 IFilePromise 接口。一旦放置文件释放后， IFilePromise 接口就会定义由 AIR 运行时访问要写入到文件中的数据所用的方法和属性。

注：由于 JavaScript 语言不支持接口的实现，您只能使用 ActionScript 实现自己的文件承诺逻辑。当然，您可以将包含 ActionScript 类的 SWF 文件导入到使用 <script> 标签的 HTML 页面并以 JavaScript 代码访问这些类。

IFilePromise 实现将为该文件释放提供数据的另一个对象传递到 AIR 运行时。此对象必须实现 AIR 运行时用于读取数据的 IDataInput 接口。例如，可实现 IFilePromise 的 URLFilePromise 类使用 URLStream 对象作为数据提供程序。

AIR 可以同步读取数据，也可以异步读取数据。 IFilePromise 实现通过返回 isAsync 属性中的相应值来报告支持哪种访问模式。如果提供异步数据访问，则数据提供程序对象必须实现 IEventDispatcher 接口并调度必要的事件，例如 open、 progress 和 complete。

可使用自定义类或以下内置类之一作为文件释放的数据提供程序：

- ByteArray （同步）
- FileStream （同步或异步）
- Socket （异步）
- URLStream （异步）

要实现 IFilePromise 接口，必须为下列函数和属性提供代码：

- open():IDataInput — 从读取的已释放文件的数据返回数据提供程序对象。该对象必须实现 IDataInput 接口。如果异步提供数据，对象还必须实现 IEventDispatcher 接口并调度必要的事件 （请参阅第 164 页的 " 在文件释放中使用异步数据提供程序 "）。
- get relativePath():String — 提供已创建文件的路径，包括文件名。此路径被解析为相对于由拖放操作中用户选择的放置位置。要确保此路径使用适用于主机操作系统的适当分隔符，请在指定包含目录的路径时使用 File.separator 常量。您可以添加 setter 函数或使用构造函数参数来允许在运行时设置路径。
- get isAsync():Boolean — 通知 AIR 运行时数据提供程序对象是异步提供数据还是同步提供数据。
- close():void — 当完全读取数据时由运行时调用 （否则将出现阻止进一步读取的错误）。可使用此函数清除资源。
- reportError(e:ErrorEvent):void — 在读取数据出错时由运行时调用。

在涉及文件释放的拖放操作期间，所有 IFilePromise 方法都由运行时调用。通常，应用程序逻辑不应直接调用上述任何方法。

在文件释放中使用同步数据提供程序

Adobe AIR 2 和更高版本

实现 IFilePromise 接口最简单的方法是使用同步数据提供程序对象，例如 ByteArray 或同步 FileStream。在下面的示例中，创建一个 ByteArray 对象，用数据填充它，并在调用 open() 方法时返回。

```
package
{
    import flash.desktop.IFilePromise;
    import flash.events.ErrorEvent;
    import flash.utils.ByteArray;
    import flash.utils.IDataInput;

    public class SynchronousFilePromise implements IFilePromise
    {
        private const fileSize:int = 5000; //size of file data
        private var filePath:String = "SynchronousFile.txt";

        public function get relativePath():String
        {
            return filePath;
        }

        public function get isAsync():Boolean
        {
            return false;
        }

        public function open():IDataInput
        {
```

```
            var fileContents:ByteArray = new ByteArray();

            //Create some arbitrary data for the file
            for( var i:int = 0; i < fileSize; i++ )
            {
                fileContents.writeUTFBytes( 'S' );
            }

            //Important: the ByteArray is read from the current position
            fileContents.position = 0;
            return fileContents;
        }

        public function close():void
        {
            //Nothing needs to be closed in this case.
        }

        public function reportError(e:ErrorEvent):void
        {
            trace("Something went wrong: " + e.errorID + " - " + e.type + ", " + e.text );
        }
    }
}
```

实际上，同步文件释放的效用很受限。如果数据量很小，您可以在临时目录中轻松创建一个文件，将普通文件列表数组添加到拖放剪贴板。另一方面，如果数据量很大或生成数据的计算成本很高，则长的同步进程是必需的。长的同步进程会在明显较长的时间内阻止 UI 更新，使应用程序看起来像无响应一样。若要避免此问题，您可以创建一个由计时器驱动的异步数据提供程序。

在文件释放中使用异步数据提供程序

Adobe AIR 2 和更高版本

使用异步数据提供程序对象时， IFilePromise isAsync 属性必须为 true，并且由 open() 方法返回的对象必须实现 IEventDispatcher 接口。运行时侦听多个替代事件，以便可以使用不同的内置对象作为数据提供程序。例如，progress 事件由 FileStream 和 URLStream 对象调度，而 socketData 事件由 Socket 对象调度。运行时从所有这些对象侦听适当的事件。

下列事件推动从数据提供程序对象读取数据的进程：

- Event.OPEN — 通知运行时数据源已准备好。
- ProgressEvent.PROGRESS — 通知运行时数据可用。运行时将从数据提供程序对象读取可用的数据量。
- ProgressEvent.SOCKET_DATA — 通知运行时数据可用。 socketData 事件由基于 Socket 的对象调度。对于其他对象类型，您应调度 progress 事件。（运行时同时侦听这两个事件以检测何时可以读取数据。）
- Event.COMPLETE — 通知运行时已读取所有数据。
- Event.CLOSE — 通知运行时已读取所有数据。（运行时同时侦听 close 和 complete 来实现此目的。）
- IOErrorEvent.IOERROR — 通知运行时读取数据时出错。运行时终止文件创建并调用 IFilePromise close() 方法。
- SecurityErrorEvent.SECURITY_ERROR — 通知运行时出现安全性错误。运行时终止文件创建并调用 IFilePromise close() 方法。
- HTTPStatusEvent.HTTP_STATUS — 运行时同时使用它和 httpResponseStatus，以确保可用数据表示所需内容，而不会显示错误消息 （例如 404 页面）。基于 HTTP 协议的对象应调度此事件。
- HTTPStatusEvent.HTTP_RESPONSE_STATUS — 运行时同时使用它和 httpStatus，以确保可用数据表示所需内容。基于 HTTP 协议的对象应调度此事件。

数据提供程序应按照以下顺序调度这些事件：

1 open 事件

2 progress 或 socketData 事件

3 complete 或 close 事件

注：内建对象 （FileStream、 Socket 和 URLStream）自动调度相应的事件。

以下示例使用自定义异步数据提供程序创建一个文件释放。数据提供程序类扩展了 ByteArray （以实现 IDataInput 支持）并实现 IEventDispatcher 接口。在每个 timer 事件发生时，对象会生成大量数据，并调度 progress 事件以通知运行时这些数据可用。当生成的数据足够多时，对象调度 complete 事件。

```
package
{
import flash.events.Event;
import flash.events.EventDispatcher;
import flash.events.IEventDispatcher;
import flash.events.ProgressEvent;
import flash.events.TimerEvent;
import flash.utils.ByteArray;
import flash.utils.Timer;

[Event(name="open", type="flash.events.Event.OPEN")]
[Event(name="complete",  type="flash.events.Event.COMPLETE")]
[Event(name="progress", type="flash.events.ProgressEvent")]
[Event(name="ioError", type="flash.events.IOErrorEvent")]
[Event(name="securityError", type="flash.events.SecurityErrorEvent")]
public class AsyncDataProvider extends ByteArray implements IEventDispatcher
{
    private var dispatcher:EventDispatcher = new EventDispatcher();
    public var fileSize:int = 0; //The number of characters in the file
    private const chunkSize:int = 1000; //Amount of data written per event
    private var dispatchDataTimer:Timer = new Timer( 100 );
    private var opened:Boolean = false;

    public function AsyncDataProvider()
    {
        super();
        dispatchDataTimer.addEventListener( TimerEvent.TIMER, generateData );
    }

    public function begin():void{
        dispatchDataTimer.start();
    }

    public function end():void
    {
        dispatchDataTimer.stop();
    }
    private function generateData( event:Event ):void
    {
        if( !opened )
        {
            var open:Event = new Event( Event.OPEN );
            dispatchEvent( open );
            opened = true;
        }
        else if( position + chunkSize < fileSize )
        {
            for( var i:int = 0; i <= chunkSize; i++ )
            {
```

```
                writeUTFBytes( 'A' );
            }
            //Set position back to the start of the new data
            this.position -= chunkSize;
            var progress:ProgressEvent =
                new ProgressEvent( ProgressEvent.PROGRESS, false, false, bytesAvailable, bytesAvailable +
chunkSize);
            dispatchEvent( progress )
        }
        else
        {
            var complete:Event = new Event( Event.COMPLETE );
            dispatchEvent( complete );
        }
    }
     //IEventDispatcher implementation
    public function addEventListener(type:String, listener:Function, useCapture:Boolean=false,
priority:int=0, useWeakReference:Boolean=false):void
    {
        dispatcher.addEventListener( type, listener, useCapture, priority, useWeakReference );
    }

    public function removeEventListener(type:String, listener:Function, useCapture:Boolean=false):void
    {
        dispatcher.removeEventListener( type, listener, useCapture );
    }

    public function dispatchEvent(event:Event):Boolean
    {
        return dispatcher.dispatchEvent( event );
    }

    public function hasEventListener(type:String):Boolean
    {
        return dispatcher.hasEventListener( type );
    }

    public function willTrigger(type:String):Boolean
    {
        return dispatcher.willTrigger( type );
    }
}
}
```

注：由于示例中的 AsyncDataProvider 类扩展了 ByteArray，所以它无法同时扩展 EventDispatcher。为实现 IEventDispatcher 接口，该类使用内部 EventDispatcher 对象并将 IEventDispatcher 方法调用转发给该内部对象。您也可以扩展 EventDispatcher 并实现 IDataInput （或同时实现两个接口）。

异步 IFilePromise 实现与同步实现几乎是相同的。主要区别是 isAsync 返回 true，而 open() 方法返回异步数据对象：

```
package
{
    import flash.desktop.IFilePromise;
    import flash.events.ErrorEvent;
    import flash.events.EventDispatcher;
    import flash.utils.IDataInput;

    public class AsynchronousFilePromise extends EventDispatcher implements IFilePromise
    {
        private var fileGenerator:AsyncDataProvider;
        private const fileSize:int = 5000; //size of file data
        private var filePath:String = "AsynchronousFile.txt";

        public function get relativePath():String
        {
            return filePath;
        }

        public function get isAsync():Boolean
        {
            return true;
        }

        public function open():IDataInput
        {
            fileGenerator = new AsyncDataProvider();
            fileGenerator.fileSize = fileSize;
            fileGenerator.begin();
            return fileGenerator;
        }

        public function close():void
        {
            fileGenerator.end();
        }

        public function reportError(e:ErrorEvent):void
        {
            trace("Something went wrong: " + e.errorID + " - " + e.type + ", " + e.text );
        }
    }
}
```

第 13 章：复制和粘贴

Flash Player 10 和更高版本，**Adobe AIR 1.0** 和更高版本

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。与 Adobe® Flash® Player 或 Adobe® AIR® 中运行的应用程序进行相互数据传输时，可使用的数据格式包括：

- 文本
- HTML 格式的文本
- RTF 数据
- 序列化对象
- 对象引用 （仅在源应用程序内有效）
- 位图 （仅限 AIR）
- 文件 （仅限 AIR）
- URL 字符串 （仅限 AIR）

复制粘贴基础知识

Flash Player 10 和更高版本，**Adobe AIR 1.0** 和更高版本

复制和粘贴 API 包含以下类。

包	类
flash.desktop	• Clipboard • 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。

HTMLLoader 类 （在 AIR 中）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。

读取和写入系统剪贴板

Flash Player 10 和更高版本，**Adobe AIR 1.0** 和更高版本

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：

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

if(Clipboard.generalClipboard.hasFormat(ClipboardFormats.TEXT_FORMAT)){
    var text:String = Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT);
}
```

注 在 Flash Player 中运行的内容或在 AIR 的非应用程序沙箱中运行的内容只能在 paste 事件的事件处理函数中调用 getData() 方法。换句话说，只有在 AIR 应用程序沙箱中运行的代码才能在 paste 事件处理函数的外部调用 getData() 方法。

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。

```
import flash.desktop.Clipboard;
import flash.desktop.ClipboardFormats;

var textToCopy:String = "Copy to clipboard.";
Clipboard.generalClipboard.clear();
Clipboard.generalClipboard.setData(ClipboardFormats.TEXT_FORMAT, textToCopy, false);
```

注：在 Flash Player 中运行的内容或在 AIR 的非应用程序沙箱中运行的内容只能在用户事件 （如键盘或鼠标事件，或者 copy 或 cut 事件）的事件处理函数中调用 setData() 方法。换句话说，只有在 AIR 应用程序沙箱中运行的代码才能在用户事件处理函数的外部调用 setData() 方法。

AIR 中的 HTML 复制和粘贴

Adobe AIR 1.0 和更高版本

Adobe AIR 中的 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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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，并传递要复制的数据和 MIME 类型：

```
 function customCopy(event){
    event.clipboardData.setData("text/plain", "A copied string.");
}
```

若要访问被粘贴的数据，您可以使用 clipboardData 对象的 getData() 方法，并传递数据格式的 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>
```

剪贴板数据格式

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

剪贴板格式描述了 Clipboard 对象中放置的数据。Flash Player 或 AIR 可在 ActionScript 数据类型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基于 ActionScript 的应用程序内或这些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 Sprite 的 Clipboard 对象可以包括在同一应用程序中使用的引用格式、由在 Flash Player 或 AIR 中运行的另一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，还可能具有延迟呈示功能以对 PNG 文件进行编码，以便将 Sprite 的表示形式复制或拖动到文件系统。

标准数据格式

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

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	（仅限 AIR）位图格式数据与 ActionScript BitmapData 类之间的转换。
FILE_LIST_FORMAT	（仅限 AIR）文件列表格式数据与 ActionScript File 对象数组之间的转换。
URL_FORMAT	（仅限 AIR） URL 格式数据与 ActionScript String 类之间的转换。

因响应 HTML 内容 （在 AIR 应用程序中承载的）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 paste 事件而被调度，则无法从该事件对象的 clipboardData 属性中获得 RTF 格式数据。

自定义数据格式

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

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。参考只在同一应用程序中有效。序列化对象可以在应用程序之间传输，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() 方法时将可序列化 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

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

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。共有四种传输模式，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_PREFFERED	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_PREFFERED	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

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

将对象写入剪贴板时，可将任何不以保留前缀 air: 或 flash: 开头的字符串用于格式 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：

```
public function createClipboardObject(object:Object):Clipboard{
    var transfer:Clipboard = Clipboard.generalClipboard;
    transfer.setData("object", object, true);
}
```

```
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 CLONE_ONLY 或 CLONE_PREFFERED 传输模式。

```
var transfer:Object = clipboard.getData("object", ClipboardTransferMode.CLONE_ONLY);
```

```
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 ORIGINAL_ONLY 或 ORIGINAL_PREFFERED 传输模式：

```
var transferredObject:Object =
    clipboard.getData("object", ClipboardTransferMode.ORIGINAL_ONLY);
```

```
var transferredObject =
    clipboard.getData("object", air.ClipboardTransferMode.ORIGINAL_ONLY);
```

仅当 Clipboard 对象是源自当前应用程序中时，引用才有效。当引用可用时，可使用 ORIGINAL_PREFFERED 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

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

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。

呈现函数是通过使用 setDataHandler() 方法添加到 Clipboard 对象中的。该函数必须返回相应格式的数据。例如，如果您调用 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，则该数据的优先级高于其延迟版本 （不会调用呈示函数）。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示函数。

注：在 Mac OS X 上，延迟呈示仅适用于自定义数据格式。使用标准数据格式时，会直接调用呈示函数。

使用延迟呈现函数粘贴文本

Flash Player 10 和更高版本， **Adobe AIR 1.0** 和更高版本

下面的示例说明了如何实现延迟呈现函数。

当用户按下 “Copy” 按钮时，应用程序将清除系统剪贴板以确保不会留下上次剪贴板操作的数据。然后， setDataHandler() 方法将 renderData() 函数设置为剪贴板渲染器。

当用户从目标文本字段的上下文菜单中选择 “ 粘贴 ” 命令时，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() 函数将返回源文本中的文本，然后将其分配给目标文本。

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。）

Flash 示例

```
package {
    import flash.desktop.Clipboard;
    import flash.desktop.ClipboardFormats;
    import flash.desktop.ClipboardTransferMode;
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.text.TextFormat;
    import flash.text.TextFieldType;
    import flash.events.MouseEvent;
    import flash.events.Event;
    public class DeferredRenderingExample extends Sprite
    {
        private var sourceTextField:TextField;
        private var destination:TextField;
        private var copyText:TextField;
        public function DeferredRenderingExample():void
        {
            sourceTextField = createTextField(10, 10, 380, 90);
            sourceTextField.text = "Neque porro quisquam est qui dolorem "
                + "ipsum quia dolor sit amet, consectetur, adipisci velit.";

            copyText = createTextField(10, 110, 35, 20);
            copyText.htmlText = "<a href='#'>Copy</a>";
            copyText.addEventListener(MouseEvent.CLICK, onCopy);

            destination = createTextField(10, 145, 380, 90);
```

```
            destination.addEventListener(Event.PASTE, onPaste);
        }
        private function createTextField(x:Number, y:Number, width:Number,
                        height:Number):TextField
        {
            var newTxt:TextField = new TextField();
            newTxt.x = x;
            newTxt.y = y;
            newTxt.height = height;
            newTxt.width = width;
            newTxt.border = true;
            newTxt.multiline = true;
            newTxt.wordWrap = true;
            newTxt.type = TextFieldType.INPUT;
            addChild(newTxt);
            return newTxt;
        }
        public function onCopy(event:MouseEvent):void
        {
            Clipboard.generalClipboard.clear();
            Clipboard.generalClipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT,
                        renderData);
        }
        public function onPaste(event:Event):void
        {
            sourceTextField.text =
            Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT).toString;
        }
        public function renderData():String
        {
            trace("Rendering data");
            var sourceStr:String = sourceTextField.text;
            if (sourceTextField.selectionEndIndex >
                    sourceTextField.selectionBeginIndex)
            {
                return sourceStr.substring(sourceTextField.selectionBeginIndex,
                                    sourceTextField.selectionEndIndex);
            }
            else
            {
                return sourceStr;
            }
        }
    }
}
```

Flex 示例

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute" width="326" height="330"
applicationComplete="init()">
    <mx:Script>
    <![CDATA[
    import flash.desktop.Clipboard;
    import flash.desktop.ClipboardFormats;

    public function init():void
    {
        destination.addEventListener("paste", doPaste);
    }

    public function doCopy():void
    {
        Clipboard.generalClipboard.clear();
        Clipboard.generalClipboard.setDataHandler(ClipboardFormats.TEXT_FORMAT, renderData);
    }
    public function doPaste(event:Event):void
    {
        destination.text = Clipboard.generalClipboard.getData(ClipboardFormats.TEXT_FORMAT).toString;
    }

    public function renderData():String{
        trace("Rendering data");
        return source.text;
    }
    ]]>
    </mx:Script>
    <mx:Label x="10" y="10" text="Source"/>
    <mx:TextArea id="source" x="10" y="36" width="300" height="100">
        <mx:text>Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
velit.</mx:text>
    </mx:TextArea>
    <mx:Label x="10" y="181" text="Destination"/>
    <mx:TextArea id="destination"  x="12" y="207" width="300" height="100"/>
    <mx:Button click="doCopy();" x="91" y="156" label="Copy"/>
</mx:Application>
```

第 14 章：在 AIR 中使用本地 SQL 数据库

Adobe AIR 1.0 和更高版本

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他类型的数据。

关于本地 SQL 数据库

Adobe AIR 1.0 和更高版本

有关使用 SQL 数据库的快速介绍和代码示例，请参阅 Adobe Developer Connection 中的以下快速入门文章：

- 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

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

Adobe AIR 1.0 和更高版本

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）。

更多帮助主题

Christophe Coenraets：AIR for Android 上的 Employee 目录

Raymond Camden：jQuery 和 AIR – 从网页到应用程序

关于 AIR 数据库和数据库文件

Adobe AIR 1.0 和更高版本

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由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

Adobe AIR 1.0 和更高版本

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，其中一个项目与其他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他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

Adobe AIR 1.0 和更高版本

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 语言的特定说明，请参阅第 293 页的 " 本地数据库中的 SQL 支持 "。

关于 SQL 数据库类

Adobe AIR 1.0 和更高版本

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
air.SQLConnection	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。
air.SQLStatement	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。
air.SQLResult	提供一种通过执行语句获取信息或结果的方法，如执行 SELECT 语句后的结果行、受 UPDATE 或 DELETE 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
air.SQLSchemaResult	充当通过调用 SQLConnection.loadSchema() 方法生成的数据库架构结果的容器。
air.SQLTableSchema	提供描述数据库中单个表的信息。
air.SQLViewSchema	提供描述数据库中单个视图的信息。
air.SQLIndexSchema	提供描述数据库中表或视图的单个列的信息。
air.SQLTriggerSchema	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
air.SQLMode	定义一组常量，它们表示 SQLConnection.open() 和 SQLConnection.openAsync() 方法的 openMode 参数的可能值。
air.SQLColumnNameStyle	定义一组常量，它们表示 SQLConnection.columnNameStyle 属性的可能值。
air.SQLTransactionLockType	定义一组常量，它们表示 SQLConnection.begin() 方法的 option 参数的可能值。
air.SQLCollationType	定义一组常数，它们表示 SQLColumnSchema() 构造函数的 SQLColumnSchema.defaultCollationType 属性和 defaultCollationType 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 SQLConnection 或 SQLStatement 实例调度的事件。每个操作都具有一个在 SQLEvent 类中定义的关联事件类型常数。
air.SQLErrorEvent	定义 SQLConnection 或 SQLStatement 实例在其任何操作导致错误时调度的事件。
air.SQLUpdateEvent	定义因执行 INSERT、 UPDATE 或 DELETE SQL 语句而导致其连接数据库之一中的表数据更改时 SQLConnection 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 SQLError 类的 operation 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

Adobe AIR 1.0 和更高版本

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，与其他代码相比，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，它们以与所有其他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，同时异步执行其他操作或语句。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()，则连接在同步执行模式下操作；如果调用 SQLConnection.openAsync()，则连接在异步执行模式下操作。使用 open() 或 openAsync() 将 SQLConnection 实例连接到数据库后，除非先关闭再重新打开到数据库的连接，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 198 页的 “ 使用同步和异步数据库操作 ”。

创建和修改数据库

Adobe AIR 1.0 和更高版本

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。

更多帮助主题

使用 Flex：更新现有 AIR 数据库

创建数据库

Adobe AIR 1.0 和更高版本

要创建数据库文件，需要首先创建 SQLConnection 实例。调用其 open() 方法在同步执行模式下打开它，或者调用其 openAsync() 方法在异步执行模式下打开它。open() 和 openAsync() 方法用于打开到数据库的连接。如果传递的 File 实例引用 reference 参数（第一个参数）的不存在的文件位置，则 open() 或 openAsync() 方法将在该文件位置创建一个数据库文件，并打开到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，而不是在磁盘上创建数据库文件。

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。在本例中，数据库文件保存在第 130 页的“指向应用程序存储目录”中，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);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.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.applicationStorageDirectory）和 resolvePath() 方法（如上一示例所示）。有关详细信息，请参阅第 128 页的“File 对象的路径”。

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();

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.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

Adobe AIR 1.0 和更高版本

在数据库中创建表包括使用与执行 SELECT、 INSERT 等 SQL 语句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 184 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 的 SQLConnection 实例，并且该实例已经实例化并连接到数据库。

```
// 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 的 SQLConnection 实例，并且该实例已经实例化并连接到数据库。

```
// 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

Adobe AIR 1.0 和更高版本

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 179 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他代码。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 198 页的 “ 使用同步和异步数据库操作 ”。

连接到数据库

Adobe AIR 1.0 和更高版本

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，以便知道 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”，位于用户的第 130 页的“指向应用程序存储目录”下。

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

conn.addEventListener(air.SQLEvent.OPEN, openHandler);
conn.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.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”，位于用户的第 130 页的“指向应用程序存储目录”下。

```
// Include AIRAliases.js to use air.* shortcuts

var conn = new air.SQLConnection();

// The database file is in the application storage directory
var folder = air.File.applicationStorageDirectory;
var dbFile = folder.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，第二个参数是常数 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，则会打开该文件，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

Adobe AIR 1.0 和更高版本

单个 SQL 语句 （查询或命令）在运行时中表示为 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，如果在异步执行模式下使用 Responder 而不是事件侦听器，则将 Responder 实例传递到 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 187 页的 " 从数据库检索数据 "

第 192 页的 " 插入数据 "

第 195 页的 " 更改或删除数据 "

在语句中使用参数

Adobe AIR 1.0 和更高版本

使用 SQL 语句参数，您可以创建可重用的 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。所以，使用参数可提供性能优势，并且可以更轻松地进行应用程序编码。

了解语句参数

Adobe AIR 1.0 和更高版本

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。要在 SQL 语句中使用参数，请像通常一样创建 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，所以需要使用以下语法设置特殊值：

```
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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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，运行时会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，请参阅第 310 页的 “ 数据类型支持 ”。

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，系统将参数值明确视为替代值，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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

要从数据库检索现有数据，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，请参阅第 293 页的 “ 本地数据库中的 SQL 支持 ”。

以下示例演示如何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";

selectStmt.addEventListener(air.SQLEvent.RESULT, resultHandler);
selectStmt.addEventListener(air.SQLErrorEvent.ERROR, errorHandler);

selectStmt.execute();

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.
}
```

以下示例演示如何使用同步执行模式执行 SELECT 语句从名为 “ 产品 ” 的表检索数据:

```
// 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";

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.
}
```

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 Responder 对象作为参数传递给 execute() 方法，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

Adobe AIR 1.0 和更高版本

SELECT 语句完成执行后，下一步是访问已检索的数据。通过调用 SQLStatement 对象的 getResult() 方法，从执行 SELECT 语句中检索结果数据：

```
var result = selectStatement.getResult();
```

getResult() 方法返回 SQLResult 对象。 SQLResult 对象的 data 属性是一个包含 SELECT 语句的结果的数组：

```
var numResults = result.data.length;
for (var i = 0; i < numResults; i++)
{
    // row is an Object representing one row of result data
    var row = result.data[i];
}
```

SELECT 结果集中的每行数据成为包含在 data 数组中的 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，使用名为 itemId、 itemName 和 price 的属性创建 Object 实例。这些属性包含来自其相应列的值。

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 employees 的表的所有行中检索包含 firstName 和 lastName 列值的行。此示例使用异步执行模式。执行完成时，调用 selectResult() 方法，使用 SQLStatement.getResult() 访问生成的数据行，使用 trace() 方法显示它们。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 employees 的表的所有行中检索包含 firstName 和 lastName 列值的行。使用 SQLStatement.getResult() 访问生成的数据行，并使用 trace() 方法显示它们。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

Adobe AIR 1.0 和更高版本

默认情况下，由 SELECT 语句返回的每行都创建为 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

Adobe AIR 1.0 和更高版本

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。

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，请指定 prefetch 参数值，并只检索由此值指定的行数：

```
// 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。得到的 SQLResult 实例的 data 属性包含数据行，并且其 complete 属性指示是否存在要检索的其他结果行。要检索其他结果行，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() 方法返回后续的一组结果行时， SQLStatement 都会调度 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。

有关详细信息，请参阅 SQLStatement.execute() 方法 （prefetch 参数描述）和 SQLStatement.next() 方法的描述。

插入数据

Adobe AIR 1.0 和更高版本

向数据库添加数据包括执行 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

Adobe AIR 1.0 和更高版本

要向数据库中的表添加数据，请创建并执行其文本为 SQL INSERT 语句的 SQLStatement 实例。

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 的 SQLConnection 实例，并且该实例已经实例化并连接到数据库。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 的 SQLConnection 实例，并且该实例已经实例化并连接到数据库。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

Adobe AIR 1.0 和更高版本

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。新插入的行的主键可使用与语句执行相关联的 SQLResult 对象进行检索。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。如果您使用的是事件侦听器或同步执行模式，则可通过调用 SQLStatement 对象的 getResult() 方法来访问 SQLResult 实例。或者，如果您使用的是异步执行模式并将 Responder 实例传递给 execute() 调用，则 SQLResult 实例将作为参数传递给结果处理函数。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，则数据库将在后台为行生成整数行标识符值。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。

由于存在这些规则，因此，如果您希望获得明确定义的主键列，并且其值通过 SQLResult.lastInsertRowID 属性调用 INSERT 命令后获得，则该列必须定义为 INTEGER PRIMARY KEY 列。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 语句中都是可用的。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，就像明确声明的 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，请参阅第 293 页的 " 本地数据库中的 SQL 支持 "。

更改或删除数据

Adobe AIR 1.0 和更高版本

执行其他数据处理操作的过程和用于执行 SQL SELECT 或 INSERT 语句的过程相同，具体内容在第 184 页的 " 使用 SQL 语句 " 中进行了介绍。仅需使用 SQLStatement 实例的 text 属性中的不同 SQL 语句进行替换即可：

- 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。

有关这些语句的说明，请参阅第 293 页的 " 本地数据库中的 SQL 支持 "。

使用多个数据库

Adobe AIR 1.0 和更高版本

使用 SQLConnection.attach() 方法，在已具有打开的数据库的 SQLConnection 实例上打开到其他数据库的连接。在 attach() 方法调用中，使用 name 参数为附加的数据库提供一个名称。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

Adobe AIR 1.0 和更高版本

通常，数据库错误处理与其他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

Adobe AIR 1.0 和更高版本

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他位置。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他位置，然后改用该位置。

语法错误

Adobe AIR 1.0 和更高版本

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为容易。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

Adobe AIR 1.0 和更高版本

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他列的值不同）。

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 197 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。

明确指定列名 可以编写 INSERT 语句而不明确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。通过明确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他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，约束错误比其他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，请参阅第 293 页的 “ 本地数据库中的 SQL 支持 ” 中的 “ON CONFLICT （冲突算法） ” 部分。

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，并要求用户选择其他名称。另一种可能的响应是，允许用户查看有关同名的其他产品的信息。

使用数据库数据类型

Adobe AIR 1.0 和更高版本

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 语句中明确声明列关联。

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。

有关可用列关联类型以及如何在 SQL 语句中使用数据类型的详细信息，请参阅第 310 页的 “ 数据类型支持 ”。

使用同步和异步数据库操作

Adobe AIR 1.0 和更高版本

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他对象。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，同时异步执行其他操作或语句。指定当您打开到数据库的连接时，是异步还是同步操作 SQLConnection。如果调用 SQLConnection.open()，则连接在同步执行模式下操作；如果调用 SQLConnection.openAsync()，则连接在异步执行模式下操作。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

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。当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();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.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

Adobe AIR 1.0 和更高版本

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他行中，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();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.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();
```

在同步执行时处理错误

Adobe AIR 1.0 和更高版本

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括在一组 try..catch..finally 代码块中。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。以下示例演示如何使用 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();

// The database file is in the application storage directory
var folder = File.applicationStorageDirectory;
var dbFile = folder.resolvePath("DBSample.db");

// open the database
conn.open(dbFile, air.SQLMode.UPDATE);

// start a transaction
conn.begin();

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

Adobe AIR 1.0 和更高版本

使用异步执行模式的一个常见问题就是，假设您当前正在对同一个数据库连接执行某个 SQLStatement 实例，则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 语句使用单独的 SQLStatement 实例，则可以在其他 SQLStatement 实例仍执行的同时调用 SQLStatement 的 execute() 方法，且不会导致错误。

在内部，当您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，则在第一个操作完成之前，无法将该语句添加到队列中 （换句话说，无法调用其 execute() 方法）。这是因为调用第二个语句的 execute() 方法后，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 事件的侦听器。将调用 SQLStatement.execute() 方法的代码放置在该侦听器方法中。

对 SQL 数据库使用加密

Adobe AIR 1.5 和更高版本

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

Adobe AIR 1.5 和更高版本

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 176 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。特别是，它可影响您的应用程序为数据库创建、获取及存储加密密钥的方式。有关这些注意事项的详细信息，请参阅第 206 页的 " 对数据库使用加密的注意事项 "。

除了加密数据库，加密本地存储区是用于保存私有敏感数据的另一种机制。使用加密本地存储区，可使用字符串密钥存储单一 ByteArray 值。只有存储该值的 AIR 应用程序可访问它，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的详细信息，请参阅第 218 页的 " 加密的本地存储区 "。

创建加密数据库

Adobe AIR 1.5 和更高版本

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 205 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 180 页的 " 创建数据库 " 中所述。首先创建一个表示数据库连接的 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）提供值。

有效的 encryptionKey 参数值为正好包含 16 个字节的 ByteArray 对象。

下列示例演示创建加密数据库。为简洁起见，在这些示例中，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。

```
var conn = new air.SQLConnection();

var encryptionKey = new air.ByteArray();
encryptionKey.writeUTFBytes("Some16ByteString"); // This technique is not secure!

// Create an encrypted database in asynchronous mode
conn.openAsync(dbFile, air.SQLMode.CREATE, null, false, 1024, encryptionKey);

// Create an encrypted database in synchronous mode
conn.open(dbFile, air.SQLMode.CREATE, false, 1024, encryptionKey);
```

有关介绍生成加密密钥的建议方式的示例，请参阅第 206 页的 " 示例：生成和使用加密密钥 "。

连接到加密数据库

Adobe AIR 1.5 和更高版本

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 182 页的 " 连接到数据库 " 中有更详细的说明。使用 open() 方法在同步执行模式下打开连接，或者使用 openAsync() 方法在异步执行模式下打开连接。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.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.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 206 页的“示例：生成和使用加密密钥”。

更改数据库的加密密钥

Adobe AIR 1.5 和更高版本

数据库加密后，可以在以后更改数据库的加密密钥。要更改数据库的加密密钥，请首先创建一个 SQLConnection 实例并调用其 open() 或 openAsync() 方法，从而打开到数据库的连接。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() 操作会同步运行。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

Adobe AIR 1.5 和更高版本

第 202 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 206 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 206 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

Adobe AIR 1.5 和更高版本

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 208 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

Adobe AIR 1.5 和更高版本

不需要了解 EncryptionKeyGenerator 类的工作方式的详情即可在您的应用程序中使用它。如果您有兴趣详细了解此类是如何为数据库生成加密密钥的，请参阅第 214 页的“了解 EncryptionKeyGenerator 类”。

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，添加与 SWF 文件相链接的 <script> 块，以此导入 SWF 代码库。第 28 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，用来表示 EncryptionKeyGenerator 类：

```
var ekg;
if (window.runtime)
{
    if (!ekg) ekg = {};
    ekg.EncryptionKeyGenerator = window.runtime.com.adobe.air.crypto.EncryptionKeyGenerator;
}
```

4 在创建数据库或打开数据库连接的代码位置之前，添加一段代码，以通过调用 EncryptionKeyGenerator() 构造函数来创建 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() 方法有一个必需的参数，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，方法是将它作为第二个 getEncryptionKey() 参数进行传递，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 203 页的 “ 创建加密数据库 ” 和第 203 页的 “ 连接到加密数据库 ” 中所述。

在此示例中，应用程序设计为以异步执行模式打开数据库。代码设置适当的事件侦听器，调用 openAsync() 方法，并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 208 页的 “ 用于生成和使用加密密钥的完整示例代码 ”。

用于生成和使用加密密钥的完整示例代码

Adobe AIR 1.5 和更高版本

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。

Flex 示例

应用程序 MXML 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：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();">
    <mx:Script>
        <![CDATA[
            import com.adobe.air.crypto.EncryptionKeyGenerator;

            private const dbFileName:String = "encryptedDatabase.db";

            private var dbFile:File;
            private var createNewDB:Boolean = true;
            private var conn:SQLConnection;

            // ------- Event handling -------

            private function init():void
            {
                conn = new SQLConnection();
                dbFile = File.applicationStorageDirectory.resolvePath(dbFileName);
                if (dbFile.exists)
                {
                    createNewDB = false;
                    instructions.text = "Enter your database password to open the encrypted database.";
                    openButton.label = "Open Database";
                }
            }

            private function openConnection():void
            {
                var password:String = passwordInput.text;

                var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();

                if (password == null || password.length <= 0)
                {
                    statusMsg.text = "Please specify a password.";
                    return;
                }

                if (!keyGenerator.validateStrongPassword(password))
                {
                    statusMsg.text = "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.";
                    return;
                }

                passwordInput.text = "";
                passwordInput.enabled = false;
                openButton.enabled = false;

                var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(password);

                conn.addEventListener(SQLEvent.OPEN, openHandler);
                conn.addEventListener(SQLErrorEvent.ERROR, openError);

                conn.openAsync(dbFile, SQLMode.CREATE, null, false, 1024, encryptionKey);
            }

            private function openHandler(event:SQLEvent):void
            {
                conn.removeEventListener(SQLEvent.OPEN, openHandler);
                conn.removeEventListener(SQLErrorEvent.ERROR, openError);
```

```
                statusMsg.setStyle("color", 0x009900);
                if (createNewDB)
                {
                    statusMsg.text = "The encrypted database was created successfully.";
                }
                else
                {
                    statusMsg.text = "The encrypted database was opened successfully.";
                }
            }

            private function openError(event:SQLErrorEvent):void
            {
                conn.removeEventListener(SQLEvent.OPEN, openHandler);
                conn.removeEventListener(SQLErrorEvent.ERROR, openError);

                if (!createNewDB && event.error.errorID ==
EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
                {
                    statusMsg.text = "Incorrect password!";
                }
                else
                {
                    statusMsg.text = "Error creating or opening database.";
                }
            }
        ]]>
    </mx:Script>
    <mx:Text id="instructions" text="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." width="75%"
height="65"/>
    <mx:HBox>
        <mx:TextInput id="passwordInput" displayAsPassword="true"/>
        <mx:Button id="openButton" label="Create Database" click="openConnection();"/>
    </mx:HBox>
    <mx:Text id="statusMsg" color="#990000" width="75%"/>
</mx:WindowedApplication>
```

Flash Professional 示例

应用程序 FLA 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。FLA 文件有以下四个放置在舞台上的组件：

实例名称	组件类型	说明
instructions	标签	包含提供给用户的说明
passwordInput	文本输入	用户从中输入密码的输入字段
openButton	按钮	输入密码后用户单击的按钮
statusMsg	标签	显示状态 （成功或失败）消息

在主时间轴的第 1 帧上的关键帧上定义此应用程序的代码。以下是应用程序的代码：

```
import com.adobe.air.crypto.EncryptionKeyGenerator;

const dbFileName:String = "encryptedDatabase.db";

var dbFile:File;
var createNewDB:Boolean = true;
var conn:SQLConnection;

init();

// ------- Event handling -------

function init():void
{
    passwordInput.displayAsPassword = true;
    openButton.addEventListener(MouseEvent.CLICK, openConnection);
    statusMsg.setStyle("textFormat", new TextFormat(null, null, 0x990000));

    conn = new SQLConnection();
    dbFile = File.applicationStorageDirectory.resolvePath(dbFileName);

    if (dbFile.exists)
    {
        createNewDB = false;
        instructions.text = "Enter your database password to open the encrypted database.";
        openButton.label = "Open Database";
    }
    else
    {
        instructions.text = "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.";
        openButton.label = "Create Database";
    }
}

function openConnection(event:MouseEvent):void
{
    var keyGenerator:EncryptionKeyGenerator = new EncryptionKeyGenerator();

    var password:String = passwordInput.text;

    if (password == null || password.length <= 0)
    {
        statusMsg.text = "Please specify a password.";
        return;
    }

    if (!keyGenerator.validateStrongPassword(password))
    {
        statusMsg.text = "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.";
        return;
    }

    passwordInput.text = "";
    passwordInput.enabled = false;
    openButton.enabled = false;

    var encryptionKey:ByteArray = keyGenerator.getEncryptionKey(password);

    conn.addEventListener(SQLEvent.OPEN, openHandler);
    conn.addEventListener(SQLErrorEvent.ERROR, openError);
```

```
    conn.openAsync(dbFile, SQLMode.CREATE, null, false, 1024, encryptionKey);
}

function openHandler(event:SQLEvent):void
{
    conn.removeEventListener(SQLEvent.OPEN, openHandler);
    conn.removeEventListener(SQLErrorEvent.ERROR, openError);

    statusMsg.setStyle("textFormat", new TextFormat(null, null, 0x009900));
    if (createNewDB)
    {
        statusMsg.text = "The encrypted database was created successfully.";
    }
    else
    {
        statusMsg.text = "The encrypted database was opened successfully.";
    }
}

function openError(event:SQLErrorEvent):void
{
    conn.removeEventListener(SQLEvent.OPEN, openHandler);
    conn.removeEventListener(SQLErrorEvent.ERROR, openError);

    if (!createNewDB && event.error.errorID == EncryptionKeyGenerator.ENCRYPTED_DB_PASSWORD_ERROR_ID)
    {
        statusMsg.text = "Incorrect password!";
    }
    else
    {
        statusMsg.text = "Error creating or opening database.";
    }
}
```

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";
                }
            }

            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>
```

了解 EncryptionKeyGenerator 类

Adobe AIR 1.5 和更高版本

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 207 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。可以下载包括源代码和文档的 as3corelib 包。还可以在项目站点查看源代码，或进行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

Adobe AIR 1.5 和更高版本

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。

应用程序的代码创建 EncryptionKeyGenerator 实例并调用其 getEncryptionKey() 方法，将用户输入的密码作为参数 （此示例中为 password 变量）传递：

```
var keyGenerator = new ekg.EncryptionKeyGenerator();
var encryptionKey = keyGenerator.getEncryptionKey(password);
```

调用 getEncryptionKey() 方法后， EncryptionKeyGenerator 类所采取的第一步是检查用户输入的密码，以确保其符合密码强度的要求。EncryptionKeyGenerator 类要求密码长度为 8 - 32 个字符。密码必须包含大小写字母的混合形式，而且至少包括一个数字或符号字符。

在内部，getEncryptionKey() 方法调用 EncryptionKeyGenerator 类的 validateStrongPassword() 方法，并且密码无效时会引发异常。 validateStrongPassword() 方法是一个公共方法，这样应用程序代码无需调用 getEncryptionKey() 方法即可进行密码检查，从而避免导致错误。

将密码扩展到 256 位

Adobe AIR 1.5 和更高版本

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

Adobe AIR 1.5 和更高版本

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

Adobe AIR 1.5 和更高版本

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

Adobe AIR 1.5 和更高版本

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

Adobe AIR 1.5 和更高版本

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，或使用某些其他方式来选择位。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。

使用 SQL 数据库的策略

Adobe AIR 1.0 和更高版本

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

Adobe AIR 1.0 和更高版本

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。用户第一次运行该应用程序时，会将原始数据库文件复制到用户的第 130 页的 " 指向应用程序存储目录 "（或其他位置），并在应用程序中使用该数据库。

使用本地 SQL 数据库的最佳做法

Adobe AIR 1.0 和更高版本

下面列出了一组建议的方法，在使用本地 SQL 数据库时，可以通过这些方法提高应用程序的性能、安全性和易维护性。

预创建数据库连接

Adobe AIR 1.0 和更高版本

尽管应用程序在首次加载时不执行任何语句，但是在运行语句时，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() 方法可以避免延迟。请参阅第 182 页的 " 连接到数据库 "。

重用数据库连接

Adobe AIR 1.0 和更高版本

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 182 页的 " 连接到数据库 "。

推荐使用异步执行模式

Adobe AIR 1.0 和更高版本

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 198 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 191 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 201 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

Adobe AIR 1.0 和更高版本

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。

使用语句参数

Adobe AIR 1.0 和更高版本

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 185 页的 " 在语句中使用参数 "。

第 15 章：加密的本地存储区

Adobe® AIR® 运行时为安装在用户计算机的每个 AIR 应用程序都提供了一个永久加密的本地存储 (ELS)。此功能可保存和检索存储在用户的本地硬盘驱动器中的加密数据（其他用户无法轻松解密）。为每个 AIR 应用程序使用一个单独的加密本地存储区，每个 AIR 应用程序为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 202 页的 “ 对 SQL 数据库使用加密 ”。

您可能想使用加密本地存储区来缓存必须保护的信息，如用于获取 Web 服务的登录凭据。 ELS 适合存储不得向其他用户公开的信息。但是，使用同一用户帐户运行的其他进程仍可访问它存储的数据。因此，它不适用于合保护秘密应用程序数据，例如 DRM 或加密密钥。

在桌面平台上，通过在 Windows 中使用 DPAPI，在 Mac OS 和 iOS 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。

在 Android 上， EncryptedLocalStorage 类存储的数据未加密。而该数据由操作系统提供的用户级别的安全性进行保护。 Android 操作系统为每个应用程序分配一个单独的用户 ID。应用程序只能访问自己的文件和在公共位置创建的文件（如移动存储卡）。注意，在 Android 的 “ 根 ” 设备上，使用根权限运行的应用程序可以访问其他应用程序的文件。因此，在根设备上，加密的本地存储不提供与非根设备上级别一样高的数据保护。

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。

如果更新 AIR 应用程序，则更新后的版本仍能够访问加密本地存储区中的任何现有数据，以下情况除外：

- 使用 stronglyBound 参数添加的项目设置为 true
- 现有和更新版本发布的时间都早于 AIR 1.5.3，并且更新使用迁移签名进行签名。

加密本地存储区的限制

加密本地存储区中的数据由用户操作系统帐户凭据进行保护。除非可以用该用户的身份进行登录，否则其他实体无法访问存储区中的数据。但是，已通过身份验证的用户运行的其他应用程序仍可访问这些数据。

由于用户必须经过身份验证才能使这些攻击生效，所以用户的隐私数据仍然受到保护（除非用户的帐户本身已被泄漏）。但是，应用程序希望对用户保密的数据（如用于授权或数字版权管理的密钥）是不安全的。因此， ELS 不是存储此类信息的适当位置。它只适合存储用户的隐私数据，如密码。

ELS 中的数据可能由于各种原因而丢失。例如，用户可能会卸载应用程序并删除加密的文件。或者，发行商 ID 可能由于更新而发生更改。因此，应将 ELS 用作私有缓存，而不是永久性数据存储。

stronglyBound 参数已弃用，不应将其设置为 true。此参数设置为 true 后，不会对数据进行任何额外保护。同时，即使发行商 ID 保持不变，应用程序每次更新后都会丢失对数据的访问。

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。

使用 ELS 的最佳做法包括：

- 使用 ELS 存储例如密码等敏感用户数据（将 stronglyBound 设置为 false）
- 不使用 ELS 存储应用程序机密（如 DRM 密钥或授权令牌）。
- 为应用程序提供在 ELS 数据丢失的情况下重新创建 ELS 中存储的数据的方法。例如，在必要时，通过提示用户重新输入帐户凭据来实现此操作。
- 不要使用 stronglyBound 参数。
- 如果 stronglyBound 确实设置为 true，则在更新期间不要迁移存储的项目。而应在更新后重新创建数据。

- 仅存储较少数量的数据。对于大量数据，请使用加密的 AIR SQL 数据库。

更多帮助主题

flash.data.EncryptedLocalStore

将数据添加到加密本地存储区

使用 EncryptedLocalStore 类的 setItem() 静态方法将数据存储在本地存储区中。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);
```

setItem() 方法的第三个参数 （即 stronglyBound 参数）是可选参数。如果此参数设置为 true，则加密本地存储区会将存储的项目绑定到存储 AIR 应用程序的数字签名和位：

```
var str = "Bob";
var bytes = new air.ByteArray();
bytes.writeUTFBytes(str);
air.EncryptedLocalStore.setItem("firstName", bytes, false);
```

对于将 stronglyBound 设置为 true 后存储的项目，在以后调用 getItem() 时，仅当调用方 AIR 应用程序与存储方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。忽略在移动设备上将 stronglyBound 设置为 true ；始终将该参数视为 false。

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。应用程序的位可以更改 （需由同一发行商对这些位进行签名），但不需要与存储数据的应用程序中的位完全相同。如果更新后的应用程序与原始应用程序的发行商 ID 相同，则可继续访问这些数据。

注：实际上，将 stronglyBound 设置为 true 不会增加任何额外数据保护。“ 恶意 ” 用户仍可更改应用程序，从而访问存储在 ELS 中的项目。而且，无论将 stronglyBound 设置为 true 还是 false，保护数据免受外部非用户威胁的强度都是一样的。出于以上原因，建议不要将 stronglyBound 设置为 true。

访问加密的本地存储区中的数据

Adobe AIR 1.0 和更高版本

您可以使用 EncryptedLocalStore.getItem() 方法从加密的本地存储区中检索值，如下例所示：

```
var storedValue = air.EncryptedLocalStore.getItem("firstName");
air.trace(storedValue.readUTFBytes(storedValue.length)); // "foo"
```

从加密的本地存储区中删除数据

Adobe AIR 1.0 和更高版本

您可以使用 EncryptedLocalStore.removeItem() 方法删除加密的本地存储区中的值，如下例所示：

```
air.EncryptedLocalStore.removeItem("firstName");
```

您可以通过调用 EncryptedLocalStore.reset() 方法清除加密的本地存储区中的所有数据，如下例所示：

```
air.EncryptedLocalStore.reset();
```

第 16 章 : 使用字节数组

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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() 方法可以将具有序列化 Action Message Format (AMF) 格式的对象写入 ByteArray，而利用 readObject() 方法则可以从 ByteArray 中将序列化对象读取到原始数据类型的变量中。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。

更多帮助主题

flash.utils.ByteArray

flash.utils.IExternalizable

Action Message Format 规范

读取并写入 ByteArray

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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。此数组结构允许代码调用相应的 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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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, consectetuer 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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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。该属性可接受的值为 "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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

利用 compress() 方法可以根据指定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，字节数组的长度将设置为新的长度并且 position 属性将设置为结尾。

CompressionAlgorithm 类 (AIR) 定义了可用来指定压缩算法的常量。 ByteArray 类支持 deflate （仅限 AIR）和 zlib 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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

readObject() 和 writeObject() 方法可从 ByteArray 中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。AMF 0 通过引用支持发送复杂对象并允许端点还原对象关系。AMF 3 通过以下方式对 AMF 0 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 中引入的新的数据类型。ByteArray.objectEcoding 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 中，随后将该 ByteArray 写入桌面上的 order 文件中。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 中读取序列化 AMF 格式的对象并将其存储到指定类型的对象中。以下示例将桌面中的 order 文件读入 ByteArray (inBytes) 中并调用 readObject() 以将其存储在 orderXML 中，然后将其转换为 XML 对象文档 myXML，并显示两个项和价格元素的值。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

Adobe AIR 1.0 和更高版本

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
一般用途位标记	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 MXML 文档为应用程序的 Flex 版本定义用户界面：

```
<?xml version="1.0" encoding="utf-8"?>
<mx:WindowedApplication xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
creationComplete="init();">
    <mx:Script>
        <![CDATA[
            // The application code goes here
        ]]>
    </mx:Script>
    <mx:Form>
        <mx:FormItem label="Output">
            <mx:TextArea id="taFiles" width="320" height="150"/>
        </mx:FormItem>
    </mx:Form>
</mx:WindowedApplication>
```

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 flNameLength;
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();
```

在 Flex 中，程序代码从 init() 方法开始，该方法将作为根 mx:WindowedApplication 标签的 creationComplete 处理函数调用。

程序代码通过 init() 方法启动，该方法将作为 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 bytes 的 endian 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 = air.Endian.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为十六进制值 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
flNameLength = bytes.readShort();// store file name
offset += flNameLength; // 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(flNameLength); // 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
if (compSize == 0) continue;
zStream.readBytes(bytes, 0, compSize);
```

接下来，示例将对压缩的文件进行解压缩并调用 outfile() 函数将文件写入输出文件流。它将向 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
```

右括号指示 while 循环、 init() 方法以及应用程序代码结束，不过 outFile() 方法除外。执行过程再次返回至 while 循环的开始处并继续处理 .zip 文件中其余的字节：提取另一个文件，如果最后一个文件已处理完毕，则终止该 .zip 文件处理。当所有文件均经过处理后，示例将 output 变量的内容写入 div 元素 taFiles 以便在屏幕上显示文件信息。

```
    } // end of while loop

    document.getElementById("taFiles").innerHTML = output;
} // end of init() method
```

outfile() 函数将以 WRITE 模式打开桌面上的输出文件，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 17 章：在 AIR 中添加 PDF 内容

Adobe AIR 1.0 和更高版本

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 AIR 应用程序中的 PDF 文件显示。对 Reader 工具栏界面 （例如控件的位置、定位和可见性）的修改仍然存在于对 AIR 应用程序和浏览器中的 PDF 文件的后续查看中。

重要说明：要在 AIR 中呈现 PDF 内容，用户必须安装 Adobe Reader 或 Adobe® Acrobat® 版本 8.1 或更高版本。

检测 PDF 功能

Adobe AIR 1.0 和更高版本

如果用户没有 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
HTMLPDFCapability.STATUS_OK	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 HTMLLoader 对象中。
HTMLPDFCapability.ERROR_INSTALLED_READER_NOT_FOUND	未检测到任何 Adobe Reader 版本。 HTMLLoader 对象无法显示 PDF 内容。
HTMLPDFCapability.ERROR_INSTALLED_READER_TOO_OLD	已检测到 Adobe Reader，但版本太旧。 HTMLLoader 对象无法显示 PDF 内容。
HTMLPDFCapability.ERROR_PREFERRED_READER_TOO_OLD	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。HTMLLoader 对象无法显示 PDF 内容。

在 Windows 上，如果用户的系统上运行的是 Adobe Acrobat 或 Adobe Reader 版本 7.x 或更高版本，即使安装了支持加载 PDF 的最新版本，也会使用当前运行的版本。在这种情况下，如果 pdfCapability 属性的值是 HTMLPDFCapability.STATUS_OK，则当 AIR 应用程序尝试加载 PDF 内容时，较旧版本的 Acrobat 或 Reader 会显示警报 （并且 AIR 应用程序中不会引发异常）。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。

以下代码检测用户能否在 AIR 应用程序中显示 PDF 内容。如果用户无法显示 PDF，代码会跟踪对应于 HTMLPDFCapability 错误对象的错误代码：

```
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

Adobe AIR 1.0 和更高版本

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 273 页的 “URI 方案 ”。

编写 PDF 内容的脚本

Adobe AIR 1.0 和更高版本

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 （当然还包括其他功能）：

- 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 开发人员联盟：http://www.adobe.com/devnet/acrobat/javascript.html。

HTML-PDF 通信基础知识

Adobe AIR 1.0 和更高版本

HTML 页中的 JavaScript 可以通过调用表示 PDF 内容的 DOM 对象的 postMessage() 方法来向 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 hostContainer 对象添加 messageHandler 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 1.0 和更高版本

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 应用程序中的其他显示对象不同。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。
- 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。更改这些属性会使 PDF 内容不可见，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， PDF 内容也会不可见。
- 只有在将包含 PDF 内容的 NativeWindow 对象的舞台对象（window.nativeWindow.stage 属性）的 scaleMode 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 18 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。

更多帮助主题

flash.media.Sound

flash.media.Microphone

flash.events.SampleDataEvent

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：

- 可将外部声音文件 （例如 mp3 文件）加载到此应用程序。
- 可以将声音信息嵌入 SWF 文件，加载它 （使用 <script src="[swfFile].swf" type="application/x-shockwave-flash"/>）并播放它。
- 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。

虽然存在多种可用来对数字音频进行编码的声音文件格式，但 AIR 仅支持以 mp3 格式存储的声音文件。它不能直接加载或播放 WAV 或 AIFF 等其他格式的声音文件。

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可以是应用程序同时播放的多个声音中的一个。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。

也可以使用几个其他运行时类，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 235 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)。

了解声音体系结构

应用程序可以从以下五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

- 使用 sampleData 事件处理程序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 等其他格式的声音文件。（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他声音输入设备。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、采样率以及其他特性。

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。要加载新的声音资源，应用程序需要创建另一 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 Sound 对象可以调度的事件：

事件	说明
open (air.Event.OPEN)	在声音加载操作之前最后一刻进行调度。
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

声音文件可能很大并需要很长时间进行加载，在通过 Internet 加载声音文件时尤为如此。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。调用 Sound.load() 方法并从声音文件接收第一批数据之后，将发生 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

  将声音文件嵌入应用程序的 SWF 文件中的具体方法随 SWF 内容开发环境的不同而不同。有关在 SWF 文件中嵌入媒体的信息，请参阅 SWF 内容开发环境文档

  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。包含 play() 方法，如上一示例所示。

处理声音流文件

如果在声音文件或视频文件的数据加载过程中播放该文件，则称为 “ 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 bufferTime 值，应用程序可以覆盖单个声音的全局 SoundMixer.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，它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它。

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);
    }
}
```

当您调用 Sound.play() 时，该应用程序将开始调用事件处理函数，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他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() 传递这些字节：

```
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，启用的其他 Sound 方法仅包括 Sound.extract() 和 Sound.play()。调用任何其他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，指定快速且连续地将声音重复播放固定的次数。

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，应用程序可以设置其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，仅当在开始播放之前完全加载了声音数据时，此代码才会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，并在其计算中使用该值。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() 函数作为其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。此代码对 SoundChannel 对象使用事件侦听器，以在完成播放时侦听发生的 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。

在播放整个声音后，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 对象的左右两个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。

以下代码示例使用音量值 0.6 和声相值 -1 创建了一个 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 SoundTransform 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 248 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 （声音在左右两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。

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他数据。

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

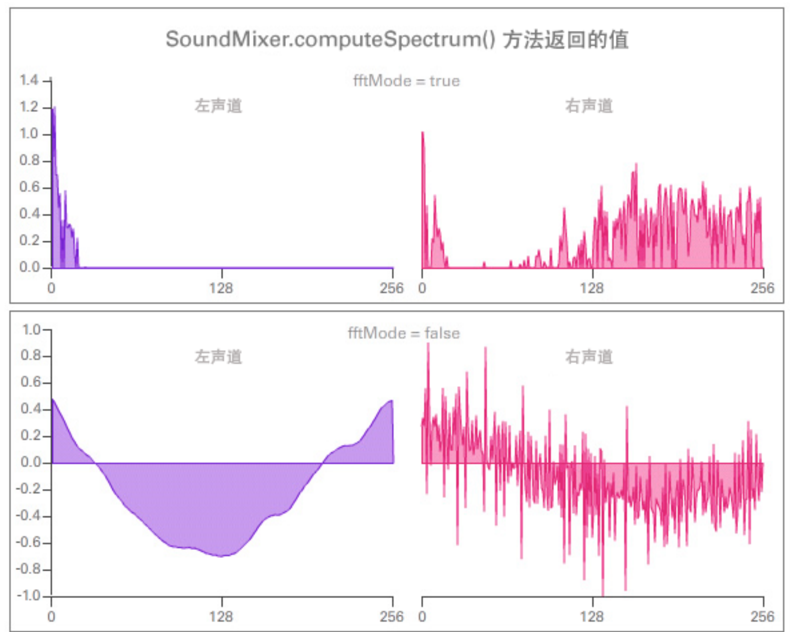
通过使用 SoundMixer.computeSpectrum() 方法，应用程序可以读取当前所播放的波形的原始声音数据。如果当前播放多个 SoundChannel 对象， SoundMixer.computeSpectrum() 方法将显示混合在一起的每个 SoundChannel 对象的组合声音数据。

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。频谱显示按声音频率 （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。

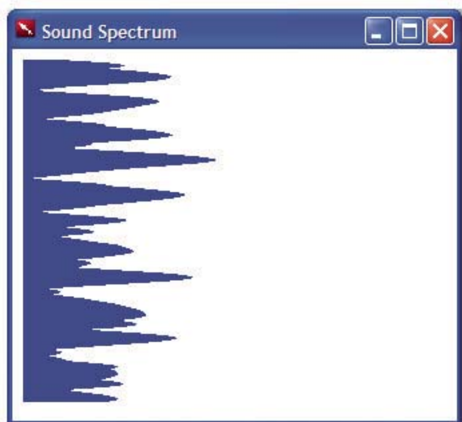
下图比较了将 FFTMode 参数设置为 true 和 false 时从 computeSpectrum() 方法返回的数据。此图所用的声音在左声道中包含很大的低音；而在右声道中包含击鼓声。



computeSpectrum() 方法也可以返回已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，采样率就减小一半，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() 方法来显示定期绘制动画的声音波形图：



```
<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他声音输入设备。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他声音输入设备。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 默认值是 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 上提供的在线 Flash Media Server 文档。

第 19 章：客户端系统环境

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

此讨论阐述了如何与用户的系统进行交互。具体介绍了如何确定支持哪些功能，以及如何通过用户安装的输入法编辑器 (IME) （如果有）构建多语言应用程序。还将介绍应用程序域的典型用法。

更多帮助主题

flash.system.System

flash.system.Capabilities

客户端系统环境基础知识

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

在构建更高级的应用程序时，您可能需要了解有关用户操作系统的详细信息并访问用户操作系统的功能。 flash.system 包包含一个类集合，使您可以访问系统级功能，例如：

- 确定应用程序和安全域代码的执行位置
- 确定用户的 Flash 运行时 （如 Flash® Player 或 Adobe® AIR™）实例的功能，如屏幕大小 （分辨率），以及某种功能 （如 mp3 音频）是否可用
- 使用 IME 建立多语言站点
- 与 Flash 运行时的容器 （可能是 HTML 页面或容器应用程序）交互。
- 将信息保存到用户的剪贴板中

flash.system 包还包括 IMEConversionMode 和 SecurityPanel 类。这两个类分别包含与 IME 和 Security 类一起使用的静态常量。

重要概念和术语
以下参考列表包含重要术语：

操作系统 计算机上运行的主要程序 （所有其他应用程序均在其中运行），如 Microsoft Windows、 Mac OS X 或 Linux®。

剪贴板 用于保存复制或剪切的文本或项目的操作系统容器，可从中将项目粘贴到应用程序中。

应用程序域 用于将不同 SWF 文件中使用的类分开的机制，以便在 SWF 文件包含具有相同名称的不同类时，这些类不会彼此覆盖。

IME （输入方法编辑器） 用于使用标准键盘输入复杂字符或符号的程序 （或操作系统工具）。

客户端系统 在编程术语中，客户端是在个人计算机上运行且由单个用户使用的应用程序部分 （或整个应用程序）。“ 客户端系统 ” 是指用户计算机上的基础操作系统。

使用 System 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

可以通过 System 类包含的一些方法和属性与用户的操作系统进行交互，并检索运行时当前的内存使用情况。还可以使用 System 类的方法和属性来侦听 imeComposition 事件、指示运行时使用用户的当前代码页加载外部文本文件或作为 Unicode 进行加载或设置用户剪贴板的内容。

在运行时获取有关用户系统的数据

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

通过检查 System.totalMemory 属性，可以确定运行时当前使用的内存量 （以字节为单位）。该属性可让您监视内存使用情况，并根据内存级别的更改方式优化应用程序。例如，如果特定视觉效果导致内存使用量大幅增加，您可能需要考虑修改此效果或将其完全消除。

System.ime 属性是对当前安装的输入法编辑器 (IME) 的引用。该属性允许使用 addEventListener() 方法来侦听 imeComposition 事件 (flash.events.IMEEvent.IME_COMPOSITION)。

System 类中的第三个属性是 useCodePage。当 useCodePage 设置为 true 时，运行时使用操作系统的传统代码页加载外部文本文件。如果将此属性设置为 false，就是告诉运行时按 Unicode 解释外部文件。

如果将 System.useCodePage 设置为 true，请记住，操作系统的传统代码页必须包括在外部文本文件中使用的字符，这样才能显示文本。例如，如果您加载了一个包含中文字符的外部文本文件，则这些字符不能显示在使用英文 Windows 代码页的系统上，因为该代码页不包括中文字符。

要确保所有平台上的用户都能查看您的应用程序中使用的外部文本文件，应使所有外部文本文件采用 Unicode 编码，并将 System.useCodePage 设置保留为默认设置 false。这样，运行时将按 Unicode 解释文本。

使用 Capabilities 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

开发人员可通过 Capabilities 类来确定正在运行应用程序的环境。使用 Capabilities 类的多种属性，您可以了解用户的系统分辨率，用户的系统是否支持辅助软件、用户操作系统的语言，以及当前安装的 Flash 运行时版本。

通过检查 Capabilities 类的属性，可以自定义应用程序，使其以最佳方式与特定用户环境配合工作。例如，通过检查 Capabilities.screenResolutionX 和 Capabilities.screenResolutionY 属性，可以确定用户系统所使用的显示分辨率，并确定最合适的视频大小。或者，在尝试加载外部 mp3 文件之前，可以检查 Capabilities.hasMP3 属性以查看用户系统是否支持 mp3 播放。

以下代码使用正则表达式分析客户端正在使用的 Flash 运行时版本：

```
var versionString = air.Capabilities.version;
var pattern = /^(\w*) (\d*),(\d*),(\d*),(\d*)$/;
var result = pattern.exec(versionString);
if (result != null)
{
    air.trace("input: " + result.input);
    air.trace("platform: " + result[1]);
    air.trace("majorVersion: " + result[2]);
    air.trace("minorVersion: " + result[3]);
    air.trace("buildNumber: " + result[4]);
    air.trace("internalBuildNumber: " + result[5]);
}
else
{
    air.trace("Unable to match RegExp.");
}
```

第 20 章：AIR 应用程序的调用和终止

Adobe AIR 1.0 和更高版本

本节讨论几种对已安装的 Adobe® AIR® 应用程序进行调用的方法，以及关闭运行中的应用程序的选项和注意事项。

注：NativeApplication、InvokeEvent 和 BrowserInvokeEvent 对象只可用于 AIR 应用程序沙箱中运行的 SWF 内容。在 Flash Player 运行时、浏览器、独立播放器（放映文件）或应用程序沙箱之外的 AIR 应用程序中运行的 SWF 内容无法访问这些类。

有关调用和终止 AIR 应用程序的快速介绍和代码示例，请参阅 Adobe Developer Connection 上的以下快速入门文章：

- 启动选项
- 启动选项

更多帮助主题

air.NativeApplication

flash.events.InvokeEvent

flash.events.BrowserInvokeEvent

应用程序调用

Adobe AIR 1.0 和更高版本

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正在处理应用程序自我更新后开始更新 AIR 应用程序（方法是将 <customUpdateUI>true</customUpdateUI> 声明包含在应用程序描述符文件中）。
- 访问承载了将调用 com.adobe.air.AIR launchApplication() 方法（该方法可为 AIR 应用程序指定识别信息）的 Flash 标志或应用程序的网页。（要使浏览器调用成功，应用程序描述符还必须包含 <allowBrowserInvocation>true</allowBrowserInvocation> 声明。）

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 对象将仅调度一个 invoke 事件。

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。在对 addEventListener() 的调用返回后不久，将以短时间间隔一次调度一个排队的 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，

```
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

Adobe AIR 1.0 和更高版本

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 InvokeEvent 对象中进行传送的。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。

除非用双引号引起来，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	Array
tick tock	{tick,tock}
tick "tick tock"	{tick,tick tock}
"tick" “tock”	{tick,tock}
\"tick\" \"tock\"	{"tick","tock"}

InvokeEvent 对象的 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。

使用 currentDirectory File 对象的 resolve() 方法可访问该文件：

```
 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

Adobe AIR 1.0 和更高版本

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>
```

用户登录时调用 AIR 应用程序

Adobe AIR 1.0 和更高版本

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。对于任何其他调用路径， 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。

从浏览器调用 AIR 应用程序

Adobe AIR 1.0 和更高版本

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>
```

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。在对 addEventListener() 的调用返回后将调度这些事件，但并不一定会在注册后可能接收到的其他 BrowserInvokeEvent 事件之前调度。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
arguments	要传递给应用程序的参数 （字符串）数组。
isHTTPS	浏览器中的内容是否使用 https URL 方案。如果是，则为 (true，否则为 (false)。
isUserEvent	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 true ； AIR 需要与浏览器调用功能有关的用户事件。
sandboxType	浏览器中内容的沙箱类型。定义的有效值与可用于 Security.sandboxType 属性的值相同，可以是以下值之一： • Security.APPLICATION — 内容位于应用程序安全沙箱中。 • Security.LOCAL_TRUSTED — 内容位于只能与本地文件系统内容交互的安全沙箱中。 • Security.LOCAL_WITH_FILE — 内容位于只能与本地文件系统内容交互的安全沙箱中。 • Security.LOCAL_WITH_NETWORK — 内容位于只能与远程内容交互的安全沙箱中。 • Security.REMOTE — 内容位于远程 （网络）域中。
securityDomain	浏览器中的内容的安全域，如 "www.adobe.com" 或 "www.example.org"。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。如果从浏览器传递的将是远程路径，则该应用程序应确保路径不使用 file:// 协议替代远程协议。

应用程序终止

Adobe AIR 1.0 和更高版本

终止应用程序最快的方法是调用 NativeApplication exit() 方法。在应用程序没有要保存的数据或没有要清理的外部资源时，此方法非常适用。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 并不通过系统镶边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。例如，如果您拥有带有工具调色板的文档窗口，则可能需要运用此方法。如果系统关闭了调色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镶边、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，然后再调用应用程序代码中的 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 NativeApplication.nativeApplication 对象的应用程序 exiting 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）。

第 21 章：处理 AIR 运行时和操作系统信息

Adobe AIR 1.0 和更高版本

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。

更多帮助主题

flash.desktop.NativeApplication

管理文件关联

Adobe AIR 1.0 和更高版本

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。要接管与其他应用程序的关联，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他应用程序，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
isSetAsDefaultApplication()	如果 AIR 应用程序当前与指定的文件类型关联，则返回 true。
setAsDefaultApplication()	在 AIR 应用程序与文件类型的打开操作之间建立关联。
removeAsDefaultApplication()	删除 AIR 应用程序与文件类型之间的关联。
getDefaultApplication()	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。

获取运行时版本和修补级别

Adobe AIR 1.0 和更高版本

NativeApplication 对象包含一个 runtimeVersion 属性，该属性表示在其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

Adobe AIR 1.0 和更高版本

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 60 页的“AIR 安全性”。

跟踪用户当前状态

Adobe AIR 1.0 和更高版本

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 userPresent 事件之间，只调度一个 userIdle 事件。

第 22 章 : 套接字

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

套接字是在两个计算机进程之间建立的一种网络连接类型。通常，这些进程在两台连接到相同 Internet 协议 (IP) 网络的不同计算机上运行。然而，连接的进程可以在使用特定 " 本地主机 "IP 地址的同一台计算机上运行。

Adobe Flash Player 支持客户端传输控制协议 (TCP) 套接字。 Flash Player 应用程序可以作为套接字服务器连接到其他进程，但是不能接受来自其他进程的传入连接请求。换句话说， Flash Player 应用程序可以连接到 TCP 服务器，但不能用作 TCP 服务器。

Flash Player API 也包含 XMLSocket 类。XMLSocket 类使用 Flash Player 特定的协议，通过该协议，您可以与识别该协议的服务器交换 XML 消息。 ActionScript 1 中引入了 XMLSocket 类，该类现在仍然受支持以提供向后兼容性。通常，除非连接到特别创建以与 Flash XMLSocket 进行通信的服务器，否则 Socket 类应该用于新应用程序。

Adobe AIR 添加了多个用于基于套接字的网络编程的附加类。借助 ServerSocket 类，AIR 应用程序可以用作 TCP 套接字服务器，并可以连接到要求 SSL 或 TLS 安全的套接字服务器。AIR 应用程序还可以使用 DatagramSocket 类发送和接收通用数据报协议 (UDP) 消息。

TCP 套接字

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

通过传输控制协议 (TCP)，可通过永久网络连接交换消息。 TCP 可确保发送的任何消息都以正确的顺序到达，出现重大网络问题时除外。TCP 连接要求具有 " 客户端 " 和 " 服务器 "。Flash Player 可以创建客户端套接字。此外，Adobe AIR 可以创建服务器套接字。

下列 ActionScript API 提供了 TCP 连接：

- 套接字 — 允许客户端应用程序连接到服务器。 Socket 类无法侦听传入连接。
- SecureSocket (AIR) — 允许客户端应用程序连接到受信任服务器并进行加密通信。
- ServerSocket (AIR) — 允许应用程序侦听传入连接并用作服务器。
- XMLSocket — 允许客户端应用程序连接到 XMLSocket 服务器。

二进制客户端套接字

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

二进制套接字连接与 XML 套接字类似，不同的是客户端和服务器不局限于交换 XML 消息。该连接可以将数据作为二进制信息传输。因此，您可以连接到更加丰富的服务，包括邮件服务器 （POP3、 SMTP 和 IMAP）和新闻服务器 (NNTP)。

Socket 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

使用 Socket 类，您可以建立套接字连接以及读取和编写原始二进制数据。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，您可以编写与多个不同的 Internet 协议 （例如 POP3、 SMTP、 IMAP 和 NNTP）进行交互的代码。这种交互进而又使您的应用程序可以连接到邮件服务器和新闻服务器。

Flash Player 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。little-endian 字节顺序很常用，因为 Intel® x86 体系结构使用该字节顺序。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。默认情况下， IDataInput 和 IDataOutput 接口执行的所有操作和实现这些接口的类 （ByteArray、Socket 和 URLStream）都以 big-endian 格式编码；即，最高有效字节位于前面。选择的这种默认字节顺序与 Java 和正式网络字节顺序匹配。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 Endian.BIG_ENDIAN 或 Endian.LITTLE_ENDIAN。

Socket 类继承了由 IDataInput 和 IDataOutput 接口定义的所有方法（位于 flash.utils 包中）。必须使用这些方法写入和读取套接字。

有关详细信息，请参阅：

- Socket
- IDataInput
- IDataOutput
- socketData 事件

安全客户端套接字 (AIR)

Adobe AIR 2 和更高版本

您可以使用 SecureSocket 类连接到使用第 4 版安全套接字层 (SSLv4) 或第 1 版传输层安全性 (TLSv1) 的套接字服务器。安全套接字提供了三大优势：服务器身份验证、数据完整性和消息机密性。运行时使用服务器证书及其在用户信任存储区中授权机构颁发的根证书或中级证书的相关性对服务器进行身份验证。运行时依靠 SSL 和 TLS 协议实现所使用的加密算法提供数据完整性和消息机密性。

当您使用 SecureSocket 对象连接到服务器时，运行时将使用证书信任存储区验证服务器证书。在 Windows 和 Mac 上，操作系统提供信任存储区。在 Linux 中，运行时自行提供信任存储区。

如果服务器证书无效或不可信，运行时将调度 ioError 事件。您可以检查 SecureSocket 对象的 serverCertificateStatus 属性来确定验证失败的原因。没有为与不具备有效和可信证书的服务器进行通信提供任何设置。

CertificateStatus 类用于定义表示可能的验证结果的字符串常量：

- 过期 — 已超过证书过期日期。
- 无效 — 多种原因可导致证书无效。例如，证书可能已更改、损坏，或者是错误类型的证书。
- 无效链 — 证书的服务器链中有一个或多个证书无效。
- 主体不匹配 — 服务器的主机名与证书公用名不匹配。换句话说，服务器使用的是错误的证书。
- 吊销 — 证书颁发机构已吊销该证书。
- 受信任 — 证书有效且受信任。 SecureSocket 对象仅可以连接到使用有效的受信任证书的服务器。
- 未知 — SecureSocket 对象尚未验证证书。serverCertificateStatus 属性在您调用 connect() 和调度 connect 或 ioError 事件之前具有此状态值。
- 不受信任的签名者 — 证书不能 " 链接 " 到客户端计算机的信任存储区中的受信任根证书。

与 SecureSocket 对象通信要求服务器使用安全协议，并包含有效的受信任证书。在其他方面，使用 SecureSocket 对象与使用 Socket 对象是相同的。

并不是所有平台都支持 SecureSocket 对象。使用 SecureSocket 类 isSupported 属性测试运行时是否支持在当前客户端计算机上使用 SecureSocket 对象。

有关详细信息，请参阅：

- SecureSocket
- CertificateStatus
- IDataInput
- IDataOutput
- socketData 事件

XML 套接字

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

通过 XML 套接字，可以创建与远程服务器的连接，且该服务器在明确关闭之前始终保持打开状态。您可以在服务器和客户端之间交换字符串数据，如 XML。使用 XML 套接字服务器的优点之一是客户端不需要明确请求数据。服务器无需等待请求即可发送数据，并且可以将数据发送到每个已连接的客户端。

在应用程序沙箱外的 Flash Player 和 Adobe AIR 内容中， XML 套接字连接要求在目标服务器上提供套接字策略文件。有关详细信息，请参阅网站控制 （策略文件）和连接到套接字。

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 Flash Media Server （支持 RTMP）。

对于应用程序安全沙箱外的 Flash Player 或 AIR 应用程序中的内容使用 XMLSocket 对象连接到服务器的方式及位置，规定了下列限制：

- 对于应用程序安全沙箱外部的内容，XMLSocket.connect() 方法只能连接到端口号大于或等于 1024 的 TCP 端口。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的同一域中的计算机。（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他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 60 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。

当调用 XMLSocket.connect() 方法时，应用程序会打开到服务器的 TCP/IP 连接并使该连接保持打开状态，直到出现下列情况之一：

- XMLSocket 类的 XMLSocket.close() 方法被调用。
- 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）。

使用 XMLSocket 类连接到服务器

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

要创建套接字连接，必须创建服务器端应用程序以等待套接字连接请求，并将响应发送到 Flash Player 或 AIR 应用程序。此类服务器端应用程序可使用 AIR 或其他编程语言 （如 Java、 Python 或 Perl）编写。要使用 XMLSocket 类，服务器计算机必须运行可识别 XMLSocket 类使用的简单协议的守护程序：

- 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。

服务器套接字

Adobe AIR 2 和更高版本

使用 ServerSocket 类可以允许其他进程使用传输控制协议 (TCP) 套接字连接到您的应用程序。可以在本地计算机或另一台网络连接的计算机上运行连接进程。当 ServerSocket 对象收到连接请求时，会调度 connect 事件。该事件调度的 ServerSocketConnectEvent 对象包含 Socket 对象。您可以使用此 Socket 对象与其他进程进行后续通信。

要侦听传入的套接字连接，请执行以下操作：

1 创建一个 ServerSocket 对象并将其绑定到本地端口
2 为 connect 事件添加事件侦听器
3 调用 listen() 方法
4 响应 connect 事件，它为每个传入连接提供 Socket 对象

ServerSocket 对象在您调用 close() 方法之前会继续侦听新连接。

以下代码示例说明了如何创建套接字服务器应用程序。该示例侦听端口 8087 上的传入连接。收到连接时，此示例会将消息（字符串 “Connected”）发送给客户端套接字。此后，服务器会将任何收到的消息回显给客户端。

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
        var serverSocket;
        var clientSockets = new Array();
        function startServer()
        {
            try
            {
                // Create the server socket
                serverSocket = new air.ServerSocket();

                // Add the event listener
                serverSocket.addEventListener( air.Event.CONNECT, connectHandler );
                serverSocket.addEventListener( air.Event.CLOSE, onClose );

                // Bind to local port 8087
                serverSocket.bind( 8087, "127.0.0.1" );

                // Listen for connections
                serverSocket.listen();
                air.trace( "Listening on " + serverSocket.localPort );
            }
            catch( e )
```

```
            {
                air.trace( e );
            }
        }
        function connectHandler( event )
        {
            //The socket is provided by the event object
            var socket = event.socket;
            clientSockets.push( socket );

            socket.addEventListener( air.ProgressEvent.SOCKET_DATA, socketDataHandler);
            socket.addEventListener( air.Event.CLOSE, onClientClose );
            socket.addEventListener( air.IOErrorEvent.IO_ERROR, onIOError );

            //Send a connect message
            socket.writeUTFBytes("Connected.");
            socket.flush();

            air.trace( "Sending connect message" );
        }

        function socketDataHandler( event )
        {
            var socket = event.target

            //Read the message from the socket
            var message = socket.readUTFBytes( socket.bytesAvailable );
            air.trace( "Received: " + message);
            // Echo the received message back to the sender
            message = "Echo -- " + message;
            socket.writeUTFBytes( message );
            socket.flush();
            air.trace( "Sending: " + message );
        }

        function onClientClose( event )
        {
            air.trace( "Connection to client closed." );
            //Should also remove from clientSockets array...
        }
        function onIOError( errorEvent )
        {
            air.trace( "IOError: " + errorEvent.text );
        }
        function onClose( event )
        {
            air.trace( "Server socket closed by OS." );
        }
</script>
</head>
<body onload="startServer()">
</body>
</html>
```

有关详细信息，请参阅：

- ServerSocket
- ServerSocketConnectEvent
- Socket

UDP 套接字 (AIR)

Adobe AIR 2 和更高版本

通用数据报协议 (UDP) 提供了一种通过无状态网络连接交换消息的方法。UDP 无法确保消息按顺序传送，甚至无法确保消息的传送。使用 UDP，操作系统的网络代码通常在封送、跟踪和确认消息上将花费更少的时间。因此，通常 UDP 消息到达目标应用程序的延迟比 TCP 消息到达目标应用程序的延迟要短。

在必须发送实时信息 （例如游戏中的位置更新或音频聊天应用程序中的声音数据包）时， UDP 套接字通信很有用。在此类应用程序中，丢失一些数据是可以接受的，并且低传输延迟比保证及时到达更重要。对于几乎所有其他目的， TCP 套接字是更好的选择。

AIR 应用程序可以使用 DatagramSocket 和 DatagramSocketDataEvent 类发送和接收 UDP 消息。要发送或接收 UDP 消息，请执行以下操作：

1 创建一个 DatagramSocket 对象

2 为 data 事件添加事件侦听器

3 使用 bind() 方法将套接字绑定到本地 IP 地址和端口

4 通过调用 send() 方法发送消息，传递目标计算机的 IP 地址和端口

5 通过响应 data 事件接收消息。为此事件调度的 DatagramSocketDataEvent 对象包含一个 ByteArray 对象，该对象中包含消息数据。

以下代码示例说明应用程序如何发送和接收 UDP 消息。此示例将包含字符串 “Hello” 的单一消息发送到目标计算机。它还跟踪接收的任何消息内容。

```
<html>
<head>
<script src="AIRAliases.js"></script>
<script language="javascript">
    var datagramSocket;

    //The IP and port for this computer
    var localIP = "192.168.0.1";
    var localPort = 55555;

    //The IP and port for the target computer
    var targetIP = "192.168.0.2";
    var targetPort = 55555;

    function createDatagramSocket()
    {
        //Create the socket
        datagramSocket = new air.DatagramSocket();
        datagramSocket.addEventListener( air.DatagramSocketDataEvent.DATA, dataReceived );

        //Bind the socket to the local network interface and port
        datagramSocket.bind( localPort, localIP );

        //Listen for incoming datagrams
        datagramSocket.receive();
```

```
        //Create a message in a ByteArray
        var data = new air.ByteArray();
        data.writeUTFBytes("Hello.");

        //Send the datagram message
        datagramSocket.send( data, 0, 0, targetIP, targetPort);
    }

    function dataReceived( event )
    {
        //Read the data from the datagram
        air.trace("Received from " + event.srcAddress + ":" + event.srcPort + "> " +
            event.data.readUTFBytes( event.data.bytesAvailable ) );
    }
</script>
</head>
<body onload="createDatagramSocket()">
</body>
</html>
```

使用 UDP 套接字时，请记住下列注意事项：

- 单一数据包不能大于网络接口或发送方和接收方之间任何网络节点的最大传输单位 (MTU) 中最小的那一个。传递到 send() 方法的 ByteArray 对象中的所有数据都将作为单一数据包发送。（在 TCP 中，较大的消息被分为几个单独的包。）
- 发送方和目标之间不存在信号交换。如果目标不存在或指定端口上没有活动侦听器，则会丢弃消息且不会报错。
- 使用 connect() 方法时，将忽略从其他源发送的消息。 UDP 连接仅提供方便的数据包过滤。这并不意味着目标地址和端口上必须存在有效的侦听进程。
- UDP 流量可以造成网络拥塞。如果发生网络拥塞，网络管理员可能需要实现服务质量控制。（TCP 有内置的流量控制来减少网络拥塞的影响。）

有关详细信息，请参阅：

- DatagramSocket
- DatagramSocketDataEvent
- ByteArray

IPv6 地址

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

Flash Player 9.0.115.0 及更高版本支持 IPv6（Internet 协议版本 6）。IPv6 是支持 128 位地址的 Internet 协议版本（它是支持 32 位地址的早期 IPv4 协议的改进版本）。您可能需要在网络接口中激活 IPv6。有关详细信息，请参阅承载数据的操作系统的帮助。

如果承载系统中支持 IPv6，您可以在用中括号 ([]) 括起的 URL 中指定数字 IPv6 文本地址，如下所示：

```
[2001:db8:ccc3:ffff:0:444d:555e:666f]
```

Flash Player 根据以下规则返回 IPv6 字面值：

- Flash Player 返回长形式的 IPv6 地址字符串。
- IP 值没有双冒号缩写词。
- 十六进制数字全小写。

- IPv6 地址包含在中括号 ([]) 中。
- 每个四重地址都输出为 0 到 4 个十六进制数字 （省略前导零）。
- 内容全为零的四重地址输出为单个零 （而不是双冒号），下表所列例外情况除外。

Flash Player 返回的 IPv6 值具有以下例外：

- 未指定的 IPv6 地址 （内容全为零）输出为 [::]。
- 环回或本地主机 IPv6 地址输出为 [::1]。
- IPv4 映射 （转换为 IPv6）地址输出为 [::ffff:a.b.c.d]，其中 a.b.c.d 为典型的 IPv4 点分十进制值。
- IPv4 兼容地址输出为 [::a.b.c.d]，其中 a.b.c.d 为典型的 IPv4 点分十进制值。

第 23 章：HTTP 通信

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

Adobe® AIR® 和 Adobe® Flash® Player 应用程序可以与基于 HTTP 的服务器通信，以便加载数据、图像、视频和交换消息。

本主题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，不介绍在 Web 浏览器中发挥作用的 HTML 和 JavaScript 本身固有的所有网络和通信功能 （例如，有关如何使用 XMLHttpRequest 类的详细信息）。

更多帮助主题

flash.net.URLLoader

flash.net.URLStream

flash.net.URLRequest

flash.net.URLRequestDefaults

flash.net.URLRequestHeader

flash.net.URLRequestMethod

flash.net.URLVariables

加载外部数据

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

AIR 运行时包括从外部源加载数据的机制。这些源可以提供静态内容 （例如文本文件）或动态内容 （例如 Web 脚本生成的内容）。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 URLRequest 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

加载外部数据的许多 API 使用 URLRequest 类来定义所需网络请求的属性。

URLRequest 属性

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

您可以在任何安全沙箱中设置 URLRequest 对象的下列属性：

属性	说明
contentType	使用 URL 请求发送的任何数据的 MIME 内容类型。如果未设置 contentType，则值将作为 application/x-www-form-urlencoded 发送。
data	一个对象，它包含将随 URL 请求一起传输的数据。
digest	唯一标识要存储为 （或从其检索） Adobe® Flash® Player 缓存的已签名 Adobe 平台组件的字符串。
method	HTTP 请求方法，例如 GET 或 POST。（AIR 应用程序安全域中运行的内容可将除 "GET" 或 "POST" 之外的字符串指定为 method 属性。允许任何 HTTP 动词， "GET" 为默认方法。请参阅第 60 页的 “AIR 安全性 ”。）
requestHeaders	要追加到 HTTP 请求的 HTTP 请求标头的数组。请注意，在 Flash Player 以及在应用程序安全沙箱外运行的 AIR 内容中，设置一些标题的权限时会受到限制。
url	指定要请求的 URL。

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 AIR 应用程序沙箱支持此属性。
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 AIR 应用程序沙箱支持设置此属性。
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 AIR 应用程序沙箱支持设置此属性。默认为对请求进行身份验证 — 如果服务器要求提供凭据，则可能会显示身份验证对话框。您还可以使用 URLRequestDefaults 类设置用户名和密码 — 请参阅第 272 页的 “ 设置 URLRequest 默认值 （仅 AIR)”。
cacheResponse	指定是否为此请求缓存响应数据。仅 AIR 应用程序沙箱支持设置此属性。默认值为缓存响应 (true)。
useCache	指定在此 URLRequest 获取数据之前是否应查询本地缓存。仅 AIR 应用程序沙箱支持设置此属性。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。

设置 URLRequest 默认值 （仅 AIR)

Adobe AIR 1.0 和更高版本

借助 URLRequestDefaults 类，您可以定义 URLRequest 对象的应用程序特定默认设置例如，以下代码设置 manageCookies 和 useCache 属性的默认值。所有新 URLRequest 对象将使用这些属性的指定值，而不是常规默认值：

```
air.URLRequestDefaults.manageCookies = false;
air.URLRequestDefaults.useCache = false;
```

注：URLRequestDefaults 类仅针对 Adobe AIR 中运行的内容进行定义。 Flash Player 中不支持该类。

URLRequestDefaults 类包含 setLoginCredentialsForHost() 方法，使用该方法可指定要为特定主机使用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 （例如 "www.example.com"）或域加端口号 （例如 "www.example.com:80"）。请注意， "example.com"、 "www.example.com" 和 "sales.example.com" 均视为唯一的主机。

只有在服务器要求提供凭据时才会使用这些凭据。如果用户已通过身份验证 （例如，使用身份验证对话框），则调用 setLoginCredentialsForHost() 方法不会更改通过身份验证的用户。

以下代码设置用于发送到 www.example.com 的请求的默认用户名和密码：

```
air.URLRequestDefaults.setLoginCredentialsForHost("www.example.com", "Ada", "love1816$X");
```

URLRequestDefaults 设置仅适用于当前应用程序域，只有一个例外。传递到 setLoginCredentialsForHost() 方法的凭据用于 AIR 应用程序中任何应用程序域所发出的请求。

有关详细信息，请参阅针对 HTML 开发人员的 Adobe AIR API 参考中的 URLRequestDefaults 类。

URI 方案

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

标准 URI 方案 （例如下列方案）可以用于任何安全沙箱所发出的请求：

http: 和 **https:**

将这些用于标准 Internet URL （与在 Web 浏览器的使用方式相同）。

文件：

使用 file: 指定位于本地文件系统中的文件的 URL。例如：

```
file:///c:/AIR Test/test.txt
```

在 AIR 中，您还可以在定义在应用程序安全性沙箱中运行的内容的 URL 时使用下列方案：

app:

使用 app: 指定相对于安装的应用程序的根目录的路径。例如，以下路径指向应用程序安装目录的 resources 子目录：

```
app:/resources
```

使用 AIR Debug Launcher (ADL) 启动 AIR 应用程序时，应用程序目录是包含应用程序描述符文件的目录。

使用 File.applicationDirectory 创建的 File 对象的 URL （和 url 属性）使用 app URI 方案，如下所示：

```
var dir = air.File.applicationDirectory;
dir = dir.resolvePath("assets");
air.trace(dir.url); // app:/assets
```

app-storage:

使用 app-storage: 指定相对于应用程序的数据存储目录的路径。 AIR 为安装的每个应用程序 （和用户）都创建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
```

使用 File.applicationStorageDirectory 创建的 File 对象的 URL （和 url 属性）使用 app-storage URI 方案，如下所示：

```
var prefsFile = air.File.applicationStorageDirectory;
prefsFile = prefsFile.resolvePath("prefs.xml");
air.trace(prefsFile.url); // app-storage:/prefs.xml
```

mailto:

在传递给 navigateToURL() 函数的 URLRequest 对象中可以使用 mailto 方案。请参阅第 282 页的 " 在其他应用程序中打开 URL"。

您可以使用 URLRequest 对象 （使用这些 URI 方案中的任何一个方案）定义许多不同对象 （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。

但是，您仅可以使用应用程序安全沙箱中的内容中的 AIR 特定 URI 方案 （app: 和 app-storage:）。有关详细信息，请参阅第 60 页的 "AIR 安全性 "。

设置 URL 变量

当可以直接向 URL 字符串添加变量时，可以更轻松地使用 URLVariables 类来定义请求所需要的任何变量。

可以使用三个方法向 URLVariables 对象添加参数:

- 在 URLVariables 构造函数中
- 使用 URLVariables.decode() 方法
- 作为 URLVariables 对象自身的动态属性

以下示例演示了全部三个方法以及如何将变量分配到 URLRequest 对象:

```
var urlVar = new air.URLVariables( "one=1&two=2" );
urlVar.decode("amp=" + air.encodeURIComponent( "&" ) );
urlVar.three = 3;
urlVar.amp2 = "&&";
air.trace(urlVar.toString()); //amp=%26&amp2=%26%26&one=1&two=2&three=3

var urlRequest = new air.URLRequest( "http://www.example.com/test.cfm" );
urlRequest.data = urlVar;
```

当在 URLVariables 构造函数或在 URLVariables.decode() 方法中定义变量时，请确保 URL 编码在 URI 字符串中包含特定含义的字符。例如，当在参数名称或值中使用 & 符时，必须通过将其从 & 更改为 %26 来编码该 & 符，因为 & 符用作分隔符。顶级 encodeURIComponent() 函数可以用于此目的。

使用 URLLoader 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

借助 URLLoader 类，您可以向服务器发送请求并访问返回的信息。您也可以使用 URLLoader 类来访问允许访问本地文件的上下文中 （例如 Flash Player 只能与本地文件系统内容交互的沙箱和 AIR 应用程序沙箱）的本地文件系统上的文件。URLLoader 类以文本、二进制数据或 URL 编码变量形式从 URL 下载数据。 URLLoader 类调度的事件包括 complete、httpStatus、 ioError、 open、 progress 和 securityError 等。

URLLoader 类为 XMLHttpRequest 类提供了替代项。您可以使用两种类中任意一种通过 HTTP 请求下载数据。

下载完成之前，下载的数据不可用。可通过侦听要调度的 progress 事件监视下载进度 （已加载字节数和总字节数）。但是，如果文件加载足够快，可能不调度 progress 事件。成功下载文件后，将调度 complete 事件。通过设置 URLLoader dataFormat 属性，您可以以文本、原始二进制数据或 URLVariables 对象格式接收数据。

URLLoader.load() 方法 （并且可能 URLLoader 类的构造函数）使用单个参数， request，该参数是 URLRequest 对象。URLRequest 对象包含所有用于单一 HTTP 请求的信息，例如目标 URL、请求方法 （GET 或 POST）、其他标题信息和 MIME 类型。

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 = (new DOMParser()).parseFromString( "<time>" + secondsUTC + "</time>", "application/xml" );
var request = new air.URLRequest("http://www.example.com/time.cfm");
request.contentType = "text/xml";
request.data = dataXML;
request.method = air.URLRequestMethod.POST;
var loader = new air.URLLoader();
loader.load(request);
```

上一个代码片断创建包含要发送到服务器的 XML 包的名为 dataXML 的 XML 文档。此示例将 URLRequest contentType 属性设置为 "text/xml" 并将 XML 文档分配给 URLRequest data 属性。最后，示例创建 URLLoader 对象并通过使用 load() 方法将请求发送到远程脚本。

使用 URLStream 类

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

数据到达时， URLStream 类将提供对下载的数据的访问。并且 URLStream 类还允许在完成下载前关闭流。下载的数据以原始二进制数据的形式提供。

当从 URLStream 对象读取数据时，使用 bytesAvailable 属性确定在读取数据之前是否提供了足够的数据。当您尝试阅读的数据比可用数据多时，将引发 EOFError 异常。

httpResponseStatus 事件 (AIR)

URLStream 类在传送任何响应数据之前将调度 httpResponseStatus 事件。httpResponseStatus 事件（由 HTTPStatusEvent 类表示）包括 responseURL 属性 （该属性是返回响应的 URL）和 responseHeaders 属性 （该属性是表示响应返回的响应标题的 URLRequestHeader 对象的数组）。

从外部文档加载数据

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

构建动态应用程序时，从外部文件或从服务器端脚本加载数据会很有用。这样，您不必编辑或重新编译应用程序，即可生成动态应用程序。例如，如果您要构建一个 " 每日提示 " 应用程序，就可以编写一个服务器端脚本，该脚本每天从数据库检索一次随机提示并将其保存到文本文件。然后，应用程序可以加载静态文本文件的内容，而不必每次查询数据库。

下面的片断创建 URLRequest 和 URLLoader 对象，用于加载外部文本文件 params.txt 的内容：

```
var request = new air.URLRequest("params.txt");
var loader = new air.URLLoader();
loader.load(request);
```

默认情况下，如果您未定义请求方法，则 Flash Player 和 Adobe AIR 会使用 HTTP GET 方法加载内容。要使用 POST 方法发送请求，可使用静态常量 URLRequestMethod.POST 将 request.method 属性设置为 POST，如以下代码所示：

```
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。

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更困难。

加载数据后，将调度 complete 事件，随后就可以在 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 = URLLoader(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，可使用顶级函数 （如 parseInt()、 parseFloat() 和 Number()）将这些值转换为数值。

无需将远程文件的内容作为字符串加载和新建 URLVariables 对象，您可以将 URLLoader.dataFormat 属性设置为在 URLLoaderDataFormat 类中找到的静态属性之一。 URLLoader.dataFormat 属性有以下三个可能的值：

- URLLoaderDataFormat.BINARY — URLLoader.data 属性包含 ByteArray 对象中存储的二进制数据。
- URLLoaderDataFormat.TEXT — URLLoader.data 属性包含 String 对象中的文本。
- URLLoaderDataFormat.VARIABLES — URLLoader.data 属性包含 URLVariables 对象中存储的 URL 编码的变量。

下面的代码演示了如何通过将 URLLoader.dataFormat 属性设置为 URLLoaderDataFormat.VARIABLES，从而自动将加载的数据分析为 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 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 实例和 URLLoader 实例，然后使用它们下载远程 XML 文档。文件完全下载后，将调度 complete 事件， trace() 函数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

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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端脚本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。

下面的片断创建一个名为 variables 的 URLVariables 对象，该对象创建称为 name 的新变量。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，此方法用于启动该请求。

```
var variables = new air.URLVariables("name=Franklin");
var request = new air.URLRequest();
request.url = "http://www.[yourdomain].com/greeting.cfm";
request.method = air.URLRequestMethod.POST;
request.data = variables;
var loader = new air.URLLoader();
loader.dataFormat = URLLoaderDataFormat.VARIABLES;
loader.addEventListener(Event.COMPLETE, completeHandler);
try
{
    loader.load(request);
}
catch (error)
{
    air.trace("Unable to load URL");
}

function completeHandler(event)
{
    air.trace(event.target.data.welcomeMessage);
}
```

下面的代码包含上面的示例中使用的 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>
```

Web 服务请求

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

存在各种基于 HTTP 的 Web 服务。主要类型包括：

- REST
- XML-RPC
- SOAP

要在 ActionScript 3 中使用 Web 服务，应创建一个 URLRequest 对象，使用 URL 变量或 XML 文档构建 Web 服务调用，然后使用 URLLoader 对象将调用发送到服务。 Flex 框架包含诸多类，其中几个类使借助 Web 服务更加轻松，尤其非常适用于访问复杂的 SOAP 服务。从 Flash Professional CS3 开始，您就可以将 Flex 类用于在 Flash Professional 和 Flash Builder 中开发的应用程序。

在基于 HTML 的 AIR 应用程序中，您可以使用 URLRequest 和 URLLoader 类或 JavaScript XMLHttpRequest 类。如果需要，您还可以创建 SWF 库，将 Flex 框架的 Web 服务组件公开到您的 JavaScript 代码。

当应用程序在浏览器中运行时，您使用的 Web 服务只能位于调用 SWF 所在的 Internet 域，除非承载 Web 服务的服务器也承载允许从其他域访问的跨域策略文件。当跨域策略文件不可用时，通常使用通过自己的服务器代理请求这项技术。 Adobe Blaze DS 和 Adobe LiveCycle 支持 Web 服务代理。

在 AIR 应用程序中，当应用程序安全沙箱调用 Web 服务时不需要跨域策略文件。远程域从不为 AIR 应用程序内容提供服务，因此该域不能加入跨域策略阻止的攻击类型。在基于 HTML 的 AIR 应用程序中，应用程序安全沙箱中的内容可以生成跨域 XMLHttpRequest。您可以允许其他安全沙箱中的内容生成跨域 XMLHttpRequest，只要该内容加载到 iframe 中。

更多帮助主题

Adobe BlazeDS

Adobe LiveCycle ES2

REST 体系结构

XML-RPC

SOAP 协议

REST 样式 Web 服务请求

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

REST 样式 Web 服务使用 HTTP 方法动词指定基本动作，并使用 URL 变量指定动作详细信息。例如，请求获得某个项目的数据时可以使用 GET 动词和 URL 变量指定方法名称和项目 ID。生成的 URL 字符串可能是：

```
http://service.example.com/?method=getItem&id=d3452
```

要使用 ActionScript 访问 REST 样式 Web 服务，您可以使用 URLRequest、URLVariables 和 URLLoader 类。在 AIR 应用程序内的 JavaScript 代码中，您还可以使用 XMLHttpRequest。

在 ActionScript 中编程 REST 样式 Web 服务调用，通常包括下列步骤：

1 创建 URLRequest 对象。

2 针对请求对象设置服务 URL 和 HTTP 方法动词。

3 创建 URLVariables 对象。

4 将服务调用参数设置为变量对象的动态属性。

5 将变量对象分配给请求对象的数据属性。

6 使用 URLLoader 对象将调用发送到服务。

7 处理由 URLLoader 调度的 complete 事件，指示服务调用已完成。还应该侦听可由 URLLoader 对象调度的多个错误事件。

例如，请考虑一种用于公开测试方法的 Web 服务，该方法将调用参数回显给请求者。可使用以下 ActionScript 代码调用此服务：

```
function restServiceCall()
{
    //Create the HTTP request object
    var request = new air.URLRequest( "http://service.example.com/" );
    request.method = air.URLRequestMethod.GET;

    //Add the URL variables
    var variables = new air.URLVariables();
    variables.method = "test.echo";
    variables.api_key = "123456ABC";
    variables.message = "Able was I, ere I saw Elba.";
    request.data = variables;

    //Initiate the transaction
    window.requestor = new air.URLLoader();
    requestor.addEventListener( air.Event.COMPLETE, httpRequestComplete );
    requestor.addEventListener( air.IOErrorEvent.IOERROR, httpRequestError );
    requestor.addEventListener( air.SecurityErrorEvent.SECURITY_ERROR, httpRequestError );
    requestor.load( request );
}
function httpRequestComplete( event )
{
    air.trace( event.target.data );
}

function httpRequestError( error ){
    air.trace( "An error occured: " + error.message );
}
```

在 AIR 应用程序内的 JavaScript 中，您可以使用 XMLHttpRequest 对象提出相同请求：

```
<html>
<head><title>RESTful web service request</title>
<script type="text/javascript">

function makeRequest()
{
    var requestDisplay = document.getElementById( "request" );
    var resultDisplay  = document.getElementById( "result" );

    //Create a conveninece object to hold the call properties
    var request = {};
    request.URL = "http://service.example.com/";
    request.method = "test.echo";
    request.HTTPmethod = "GET";
    request.parameters = {};
    request.parameters.api_key = "ABCDEF123";
    request.parameters.message = "Able was I ere I saw Elba.";
    var requestURL = makeURL( request );
    xmlhttp = new XMLHttpRequest();
    xmlhttp.open( request.HTTPmethod, requestURL, true);
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4) {
            resultDisplay.innerHTML = xmlhttp.responseText;
        }
    }
    xmlhttp.send(null);

    requestDisplay.innerHTML = requestURL;
}
```

```
//Convert the request object into a properly formatted URL
function makeURL( request )
{
    var url = request.URL + "?method=" + escape( request.method );
    for( var property in request.parameters )
    {
        url += "&" + property + "=" + escape( request.parameters[property] );
    }

    return url;
}
</script>
</head>
<body onload="makeRequest()">
<h1>Request:</h1>
<div id="request"></div>
<h1>Result:</h1>
<div id="result"></div>
</body>
</html>
```

XML-RPC Web 服务请求

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

XML-RPC Web 服务将其调用参数看作 XML 文档而不是一组 URL 变量。要使用 XML-RPC Web 服务执行事务，请创建格式正确的 XML 消息并使用 HTTP POST 方法将其发送到 Web 服务。此外，您应该为请求设置 Content-Type 标题，以便服务器将请求数据看作 XML。

在以下示例中，使用 DOM 方法创建 XML-RPC 消息和 XMLHttpRequest 来执行 Web 服务事务：

```
<html>
<head>
<title>XML-RPC web service request</title>
<script type="text/javascript">

function makeRequest()
{
    var requestDisplay = document.getElementById( "request" );
    var resultDisplay  = document.getElementById( "result" );

    var request = {};
    request.URL = "http://services.example.com/xmlrpc/";
    request.method = "test.echo";
    request.HTTPmethod = "POST";
    request.parameters = {};
    request.parameters.api_key = "123456ABC";
    request.parameters.message = "Able was I ere I saw Elba.";
    var requestMessage = formatXMLRPC( request );

    xmlhttp = new XMLHttpRequest();
    xmlhttp.open( request.HTTPmethod, request.URL, true);
    xmlhttp.onreadystatechange = function() {
        if (xmlhttp.readyState == 4) {
            resultDisplay.innerText = xmlhttp.responseText;
        }
    }
    xmlhttp.send( requestMessage );

    requestDisplay.innerText = xmlToString( requestMessage.documentElement );
}
```

```
//Formats a request as XML-RPC document
function formatXMLRPC( request )
{
    var xmldoc = document.implementation.createDocument( "", "", null );
    var root = xmldoc.createElement( "methodCall" );
    xmldoc.appendChild( root );
    var methodName = xmldoc.createElement( "methodName" );
    var methodString = xmldoc.createTextNode( request.method );
    methodName.appendChild( methodString );

    root.appendChild( methodName );

        var params = xmldoc.createElement( "params" );
        root.appendChild( params );

        var param = xmldoc.createElement( "param" );
        params.appendChild( param );
        var value = xmldoc.createElement( "value" );
        param.appendChild( value );
        var struct = xmldoc.createElement( "struct" );
        value.appendChild( struct );

        for( var property in request.parameters )
        {
            var member = xmldoc.createElement( "member" );
            struct.appendChild( member );

            var name = xmldoc.createElement( "name" );
            var paramName = xmldoc.createTextNode( property );
            name.appendChild( paramName )
            member.appendChild( name );

            var value = xmldoc.createElement( "value" );
            var type = xmldoc.createElement( "string" );
            value.appendChild( type );
            var paramValue = xmldoc.createTextNode( request.parameters[property] );
            type.appendChild( paramValue )
            member.appendChild( value );
        }
    return xmldoc;
}

//Returns a string representation of an XML node
function xmlToString( rootNode, indent )
{
    if( indent == null ) indent = "";
    var result = indent + "<" + rootNode.tagName + ">\n";
    for( var i = 0; i < rootNode.childNodes.length; i++)
    {
        if(rootNode.childNodes.item( i ).nodeType == Node.TEXT_NODE )
        {
            result += indent + "    " + rootNode.childNodes.item( i ).textContent + "\n";
        }
    }
    if( rootNode.childElementCount > 0 )
    {
        result += xmlToString( rootNode.firstElementChild, indent + "    " );
    }
    if( rootNode.nextElementSibling )
```

```
    {
        result += indent + "</" + rootNode.tagName + ">\n";
        result += xmlToString( rootNode.nextElementSibling, indent );
    }
    else
    {
        result += indent +"</" + rootNode.tagName + ">\n";
    }
    return result;
}

</script>
</head>
<body onload="makeRequest()">
<h1>Request:</h1>
<pre id="request"></pre>
<h1>Result:</h1>
<pre id="result"></pre>
</body>
</html>
```

在其他应用程序中打开 URL

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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 271 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，下面的代码打开 www.adobe.com 网页：

```
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 方法的 URLRequest 对象（其 method 属性设置为 URLRequestMethod.POST）视为使用 GET 方法。

使用 navigateToURL() 函数时，根据调用 navigateToURL() 函数的代码的安全沙箱，决定是否允许 URI 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，当在 AIR 中使用这些 API 时禁止使用某些 URI 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 60 页的 "AIR 安全性 "。）

应用程序沙箱 （仅限 **AIR**）
任何 URI 方案均可用于 AIR 应用程序沙箱中运行的内容所启动的 URL。应用程序必须经过注册才能处理 URI 方案，否则该请求不起任何作用。许多计算机和设备上支持以下方案：

- http:
- https:
- file:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序
- sms: — AIR 将 sms: 请求定向到默认的短信应用程序。 URL 格式必须符合运行应用程序的系统约定。例如，在 Android 上， URI 方案必须小写。

  ```
  navigateToURL( new URLRequest( "sms:+15555550101") );
  ```

- tel: — AIR 将 tel: 请求定向到默认的电话拨号应用程序。URL 格式必须符合运行应用程序的系统约定。例如，在 Android 上， URI 方案必须小写。

```
navigateToURL( new URLRequest( "tel:5555555555") );
```

- market: — AIR 将 market: 请求定向到通常在 Android 设备上支持的 Market 应用程序。

```
navigateToURL( new URLRequest( "market://search?q=Adobe Flash") );
navigateToURL( new URLRequest( "market://search?q=pname:com.adobe.flashplayer") );
```

如果操作系统允许，应用程序可以定义和注册自定义 URI 方案。您可以使用该方案创建 URL，以便从 AIR 启动该应用程序。

远程沙箱

允许以下方案。使用这些方案的方法与在 Web 浏览器中的用法相同。

- http:
- https:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序

所有其他 URL 方案已禁止。

只能与本地文件系统内容交互的沙箱

允许以下方案。使用这些方案的方法与在 Web 浏览器中的用法相同。

- file:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序

所有其他 URL 方案已禁止。

只能与远程内容交互的沙箱

允许以下方案。使用这些方案的方法与在 Web 浏览器中的用法相同。

- http:
- https:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序

所有其他 URL 方案已禁止。

受信任的本地沙箱

允许以下方案。使用这些方案的方法与在 Web 浏览器中的用法相同。

- file:
- http:
- https:
- mailto: — AIR 将这些请求指向注册的系统邮件应用程序

所有其他 URL 方案已禁止。

向服务器发送 URL

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

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，即 request，该参数是一个 URLRequest 对象 （请参阅第 271 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 274 页的 " 使用 URLLoader 类 "。

第 24 章：与其他 Flash Player 和 AIR 实例通信

Flash Player 9 和更高版本，**Adobe AIR 1.0** 和更高版本

LocalConnection 类支持在 Adobe® AIR® 应用程序之间以及在浏览器中运行的 SWF 内容之间进行通信。您也可以使用 LocalConnection 类在 AIR 应用程序与在浏览器中运行的 SWF 内容之间进行通信。使用 LocalConnection 类，您可以构建能在 Flash Player 和 AIR 实例之间共享数据的通用应用程序。

关于 LocalConnection 类

Flash Player 9 和更高版本，**Adobe AIR 1.0** 和更高版本

LocalConnection 对象仅可在运行于同一台客户端计算机上的 AIR 应用程序和 SWF 文件之间通信。但是，应用程序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，您就不必担心安全问题了。但如果您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，请参阅用于 Adobe Flash Platform 的 ActionScript 3.0 参考中列出的 send() 方法的 connectionName 参数和 LocalConnection 类中的 allowDomain() 和 domain 条目的介绍。

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。

isPerUser 属性

将 isPerUser 属性添加到 Flash Player (10.0.32) 和 AIR (1.5.2) 中，旨在解决多个用户登录到 Mac 计算机时发生的冲突。在其他操作系统上，将忽略此属性，因为本地连接始终限制为单个用户。在新代码中应该将 isPerUser 属性设置为 true。但是，目前的默认值为 false 以实现向后兼容。在以后的运行时版本中此默认值可能会更改。

在两个应用程序之间发送消息

Flash Player 9 和更高版本，**Adobe AIR 1.0** 和更高版本

可使用 LocalConnection 类在不同的 AIR 应用程序之间以及在浏览器中运行的不同 Adobe® Flash® Player (SWF) 应用程序之间进行通信。还可使用此 LocalConnection 类在 AIR 应用程序和在浏览器中运行的 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，负责接受从其他应用程序传入的 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。当其他应用程序调用此 LocalConnection 实例中的方法时，运行时会在 clientObject 对象中查找此方法。

如果已存在具有指定名称的连接，则会引发 Argument Error 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要求您首先在发送 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名称来调用 LocalConnection.send() 方法。例如，要向您早期创建的 LocalConnection 对象发送 doQuit 方法，可使用以下代码：

```
sendingConnection.send("conn1", "doQuit");
```

此代码使用连接名称 conn1 连接到现有 LocalConnection 对象，并调用远程应用程序中的 doMessage() 方法。如果想要将参数发送到远程应用程序，可以在 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 AIR 应用程序

Flash Player 9 和更高版本， **Adobe AIR 1.0** 和更高版本

若要只允许从特定域进行通信，可以调用 LocalConnection 类的 allowDomain() 或 allowInsecureDomain() 方法，并传递包含允许访问此 LocalConnection 对象的一个或多个域的列表，以便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() 方法尝试从调用代码没有访问权限的安全沙箱与应用程序进行通信，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他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() 来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。

Adobe AIR

要与在 AIR 应用程序安全沙箱中运行的内容（随 AIR 应用程序一起安装的内容）通信，必须将可标识 AIR 应用程序的超级域用作连接名称的前缀。超级域字符串以 app# 开头，然后依次是应用程序 ID、点 (.) 字符和发行商 ID （如果已定义）。例如，在 ID 为 com.example.air.MyApp 且没有发行商 ID 的应用程序中， connectionName 参数中使用的正确超级域为 "app#com.example.air.MyApp"。因此，如果基础连接名称为 "appConnection"，则应在 connectionName 参数中使用的整个字符串为 "app#com.example.air.MyApp:appConnection"。如果应用程序包含发行商 ID，则必须将该 ID 也包含在超级域字符串中，即 "app#com.example.air.MyApp.B146A943FBD637B68C334022D304CEA226D129B4.1"。

当您允许其他 AIR 应用程序通过本地连接与您的应用程序通信时，必须调用 LocalConnection 对象的 allowDomain() 来传入本地连接域名。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 和发行商 ID。例如，如果发送方 AIR 应用程序的应用程序 ID 为 com.example.air.FriendlyApp 且发行商 ID 为 214649436BD677B62C33D02233043EA236D13934.1，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.214649436BD677B62C33D02233043EA236D13934.1。（自 AIR 1.5.3 起，并不是所有 AIR 应用程序都包含发行商 ID。）

第 25 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 API 参考中提供了这些在基于 HTML 的应用程序的 JavaScript 代码中非常有用的类的文档。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 开发人员指南
- 用于 Adobe Flash Platform 的 ActionScript 3.0 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）。但是，当前有关这些类 （未包含在 针对 HTML 开发人员的 Adobe AIR API 参考中）的 API 文档使用 ActionScript 语法描述它们。换句话说，有关运行时的某些高级功能，请参阅 用于 Adobe Flash Platform 的 ActionScript 3.0 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赋予了 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：

- 对于指定了类型的对象，不仅可以在运行时检查数据类型，而且还可以在编译时检查数据类型 （只要您使用的是严格模式）。在编译时进行类型检查有助于识别错误。（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 290 页的 “ActionScript 3.0 自定义类 ”。

void 数据类型

void 数据类型将用作函数的返回值，实际上该函数不返回任何值。（不包含 return 语句的函数不会返回值。）

* 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）。但是，运行时还包含标准 JavaScript 中未找到的类。这些附加的类具有从播放丰富媒体 （例如声音）到与套接字交互的不同用途。

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 290 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() 方法在 public 命名空间中。使用 public 命名空间可从该类以外的对象和类访问方法和属性。

ActionScript 3.0 包

包提供了用来组织 ActionScript 3.0 类的方法。例如，许多与使用计算机上的文件和目录相关的类均包含在 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();
```

在此处， File() 方法是与具有相同名称 (File) 的类相对应的构造函数。

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()。二者均以 JavaScript 变量的属性的形式提供，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，其中的一个参数 n 是必需参数：

```
function cube(n:Number):Number {
    return n*n*n;
}
```

以下 ActionScript 3.0 代码定义了一个要求 n 参数的函数。它还包含可选的 p 参数，其默认值为 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。将 ActionScript 对象事件作为函数参数传递给事件侦听器。此事件对象的使用与 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，Sound 对象将调度下列任一事件：

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 接口是用于定义类可实现的一系列方法的数据类型。）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。以下代码对此进行了说明。（此代码假定 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 26 章：本地数据库中的 SQL 支持

Adobe AIR 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，支持具有许多标准 SQL 功能的本地 SQL 数据库。运行时未指定在文件系统上存储数据库数据的方式或位置。每个数据库都完全存储在单个文件中。开发人员可指定数据库文件在文件系统中的存储位置，单个 AIR 应用程序可访问一个或多个单独的数据库（即单独的数据库文件）。本文档概述了 SQL 语法和支持 Adobe AIR 本地 SQL 数据库的数据类型。本文档并不用作综合的 SQL 参考，而仅介绍有关 Adobe AIR 支持的 SQL 方言的详细信息。运行时支持大多数符合 SQL-92 标准的 SQL 方言。由于可以通过众多的参考资料、网站、书籍和培训材料来学习 SQL，因此本文档并不用作综合的 SQL 参考或教程。相反，本文档特别侧重于 AIR 支持的 SQL 语法，以及 SQL-92 和支持的 SQL 方言之间的差异。

SQL 语句定义约定

在本文档的语句定义中，使用了以下约定：

- 文本大小写
  - UPPER CASE — 文本 SQL 关键字以全大写形式书写。
  - lower case — 占位符项或子句名称以全小写形式书写。
- 定义字符
  - ::= - 指示一个子句或语句定义。
- 分组和替代字符
  - | - 管道字符在备选选项之间使用，可读作 “ 或 ”。
  - [] - 中括号中的项是可选项；括号中可包含单个项或一组替代项。
  - () - 括住一组替代项（一组由管道字符分隔的项）的圆括号指定一组必需项，即作为单个必需项的可能值的一组项。
- 数量表示符
  - + - 圆括号中的项后面的加号字符指示前面的项可出现 1 次或更多次。
  - * - 中括号中的项后面的星号字符指示前面的（在括号中的）项可出现 0 次或更多次
- 文本字符
  - * - 在列名中或函数名称后面的圆括号之间使用的星号，表示文本星号字符，而不是 “0 个或更多个 ” 数量表示符。
  - . 句点字符表示文本句点。
  - , - 逗号字符表示文本逗号。
  - () - 括住单个子句或项的一对圆括号指示圆括号是必需的文本圆括号字符。
  - 其他字符 - 除非另行说明，否则其他字符均表示相应的文本字符。

支持的 SQL 语法

本部分介绍 Adobe AIR SQL 数据库引擎支持的 SQL 语法。下面所列的各项分别用于说明不同的语句和子句类型、表达式、内置函数和运算符。本部分包含以下主题：

- 常规 SQL 语法
- 数据操作语句（SELECT、 INSERT、 UPDATE 和 DELETE）
- 数据定义语句（用于表、索引、视图和触发器的 CREATE、 ALTER 和 DROP 语句）

- 特殊的语句和子句
- 内置函数 （聚合函数、标量函数和日期 / 时间格式函数）
- 运算符
- 参数
- 不支持的 SQL 功能
- 其他 SQL 功能

常规 SQL 语法

除了各种语句和表达式的特定语法外，以下是 SQL 语法的一般规则：

区分大小写 SQL 语句 （包括对象名称）不区分大小写。但是， SQL 语句经常以大写形式的 SQL 关键字编写，本文档使用了该约定。尽管 SQL 语法不区分大小写，但是 SQL 中的文本值区分大小写，而且比较和排序操作可区分大小写，如为列或操作定义的排序规则序列所指定的。有关详细信息，请参阅 COLLATE。

空白 必须使用空白字符 （如空格、制表符、换行符等）来分隔 SQL 语句中的各个单词。但是，单词和符号之间的空白是可选的。 SQL 语句中空白字符的类型和数量并不重要。可使用空白 （如缩进和换行符）来设置 SQL 语句的格式以便于阅读，而不影响语句的含义。

数据操作语句

数据操作语句是最常用的 SQL 语句。这些语句用于从数据库表检索、添加、修改和删除数据。支持以下数据操作语句：SELECT、 INSERT、 UPDATE 和 DELETE。

SELECT

SELECT 语句用于查询数据库。 SELECT 的结果是零行或多行数据，其中每行都具有固定的列数。结果中的列数由 result 列名称或 SELECT 和可选 FROM 关键字之间的表达式列表指定。

```
sql-statement   ::=  SELECT [ALL | DISTINCT] result
                     [FROM table-list]
                     [WHERE expr]
                     [GROUP BY expr-list]
                     [HAVING expr]
                     [compound-op select-statement]*
                     [ORDER BY sort-expr-list]
                     [LIMIT integer [( OFFSET | , ) integer]]
result          ::=  result-column [, result-column]*
result-column   ::=  * | table-name . * | expr [[AS] string]
table-list      ::=  table [ join-op table join-args ]*
table           ::=  table-name [AS alias] |
                     ( select ) [AS alias]
join-op         ::=  , | [NATURAL] [LEFT | RIGHT | FULL] [OUTER | INNER | CROSS] JOIN
join-args       ::=  [ON expr] [USING ( id-list )]
compound-op     ::=  UNION | UNION ALL | INTERSECT | EXCEPT
sort-expr-list  ::=  expr [sort-order] [, expr [sort-order]]*
sort-order      ::=  [COLLATE collation-name] [ASC | DESC]
collation-name  ::=  BINARY | NOCASE
```

任意的表达式都可用作结果。如果结果表达式是 *，则以所有表的所有列替换该表达式。如果表达式是表名后跟 .*，则结果是该表中的所有列。

DISTINCT 关键字可导致返回结果行的子集，其中每个结果行都不同。各 NULL 值不被视为彼此不同。默认行为是返回所有结果行，这可通过关键字 ALL 明确指定。

对在 FROM 关键字后指定的一个或多个表执行查询。如果多个表名由逗号分隔，则查询将使用各个表的交叉联接。 JOIN 语法还可用于指定如何联接表。支持的唯一一个外部联接类型是 LEFT OUTER JOIN。join-args 中的 ON 子句表达式必须解析为布尔值。括号中的子查询可用作 FROM 子句中的表。可省略整个 FROM 子句，在此情况下结果是由 result 表达式列表的值组成的单个行。

WHERE 子句用于限制查询所检索的行数。WHERE 子句表达式必须解析为布尔值。WHERE 子句筛选是在任何分组之前执行的，因此 WHERE 子句表达式不能包括聚合函数。

GROUP BY 子句导致将结果的一行或多行合并到输出的单个行中。当结果包含聚合函数时， GROUP BY 子句尤其有用。GROUP BY 子句中的表达式不必是出现在 SELECT 表达式列表中的表达式。

HAVING 子句与 WHERE 类似，因为它限制语句返回的行数。但是， HAVING 子句在发生由 GROUP BY 子句指定的任何分组后应用。因此，HAVING 表达式可能引用包括聚合函数的值。不要求 HAVING 子句表达式出现在 SELECT 列表中。与 WHERE 表达式一样， HAVING 表达式必须解析为布尔值。

ORDER BY 子句导致对输出行进行排序。ORDER BY 子句的 sort-expr-list 参数是用作排序关键字的表达式列表。对于简单的 SELECT，这些表达式不必是结果的一部分，但是在复合 SELECT （使用 compound-op 运算符之一的 SELECT）中，每个排序表达式都必须与结果列之一完全匹配。每个排序表达式可能 （可选）后跟 sort-order 子句，该子句包含 COLLATE 关键字以及用于对文本进行排序的排序规则函数的名称和 / 或用于指定排序顺序 （升序或降序）的关键字 ASC 或 DESC。可省略排序顺序，在此情况下将使用默认值 （升序）。有关 COLLATE 子句和排序规则函数的定义，请参阅 COLLATE。

LIMIT 子句为结果中返回的行数设定上限。负的 LIMIT 指示无上限。LIMIT 后面的可选 OFFSET 指定要在结果集开头跳过的行数。在复合 SELECT 查询中，LIMIT 子句可能仅出现在最终 SELECT 语句之后，并且限制应用于整个查询。请注意，如果在 LIMIT 子句中使用 OFFSET 关键字，则限制是第一个整数，偏移量是第二个整数。如果使用逗号而不是 OFFSET 关键字，则偏移量是第一个数，限制是第二个数。此表面上的矛盾是有意的 — 这最大限度提高了与早期 SQL 数据库系统的兼容性。

复合 SELECT 是由运算符 UNION、UNION ALL、INTERSECT 或 EXCEPT 之一所连接的两个或更多个简单 SELECT 语句构成的。在复合 SELECT 中，所有用于构成的 SELECT 语句都必须指定相同数量的结果列。在最终 SELECT 语句之后只能有一个 ORDER BY 子句（如果指定了单个 LIMIT 子句，还要在这类子句之前）。UNION 和 UNION ALL 运算符将前面和后面的 SELECT 语句的结果组合到单个表中。两者的差异在于，在 UNION 中，所有结果行都不重复，但是在 UNION ALL 中，则可能存在重复行。 INTERSECT 运算符求出前面和后面的 SELECT 语句的结果的交集。在删除后面的 SELECT 的结果后，EXCEPT 求出前面的 SELECT 的结果。将三个或更多 SELECT 语句连接为复合语句时，它们从第一个到最后一个进行组合。

有关允许的表达式的定义，请参阅表达式。

从 AIR 2.5 开始，为了将 BLOB 数据转换为 ActionScript ByteArray 对象而进行读取时，支持 SQL CAST 运算符。例如，以下代码读取不以 AMF 格式存储的原始数据并将该数据存储在 ByteArray 对象中：

```
stmt.text = "SELECT CAST(data AS ByteArray) AS data FROM pictures;";
stmt.execute();
var result:SQLResult = stmt.getResult();
var bytes:ByteArray = result.data[0].data;
```

INSERT

INSERT 语句有两种基本形式，它用于用数据填充表。

```
sql-statement  ::=  INSERT [OR conflict-algorithm] INTO [database-name.] table-name [(column-list)] VALUES
(value-list) |
                   INSERT [OR conflict-algorithm] INTO [database-name.] table-name [(column-list)] select-
statement
                    REPLACE INTO [database-name.] table-name [(column-list)] VALUES (value-list) |
                    REPLACE INTO [database-name.] table-name [(column-list)] select-statement
```

第一种形式 （使用 VALUES 关键字）在现有表中创建单个新行。如果未指定 column-list，则值的数量必须等于表中的列数。如果指定了 column-list，则值的数量必须与指定列的数量匹配。将用创建表时定义的默认值填充未出现在列的列表中的表列；如果未定义默认值，则使用 NULL 填充。

INSERT 语句的第二种形式从 SELECT 语句提取其数据。如果未指定 column-list，则 SELECT 的结果中的列数必须与表中的列数完全匹配，或者它必须与在 column-list 中指定的列数完全匹配。在表中为 SELECT 结果的每一行创建一个新项。SELECT 可能是简单的或复合的。有关允许的 SELECT 语句的定义，请参阅 SELECT。

可选的 conflict-algorithm 允许指定要在此命令过程中使用的替代约束冲突解决算法。有关冲突算法的解释和定义，请参阅第 301 页的 " 特殊的语句和子句 "。

该语句的两种 REPLACE INTO 形式等效于将标准的 INSERT [OR conflict-algorithm] 形式与 REPLACE 冲突算法一起使用 （即 INSERT OR REPLACE... 形式）。

该语句的两种 REPLACE INTO 形式等效于将标准的 INSERT [OR conflict-algorithm] 形式与 REPLACE 冲突算法一起使用 （即 INSERT OR REPLACE... 形式）。

UPDATE

update 命令可更改表中的现有记录。

```
sql-statement  ::=  UPDATE [database-name.] table-name SET column1=value1, column2=value2,... [WHERE expr]
```

该命令由 UPDATE 关键字后跟要更新记录的表的名称组成。在 SET 关键字之后，采用逗号分隔的列表形式提供列名称以及要将该列更改为的值。 WHERE 子句表达式提供要更改其记录的一个或多个行。

DELETE

delete 命令用于从表中删除记录。

```
sql-statement  ::=  DELETE FROM [database-name.] table-name [WHERE expr]
```

该命令由 DELETE FROM 关键字后跟要从其删除记录的表的名称组成。

如果没有 WHERE 子句，则删除表的所有行。如果提供了 WHERE 子句，则仅删除与表达式匹配的那些行。 WHERE 子句表达式必须解析为布尔值。有关允许的表达式的定义，请参阅表达式。

数据定义语句

数据定义语句用于创建、修改和删除数据库对象，如表、视图、索引和触发器。支持以下数据定义语句：

- 表：
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- 索引：
  - CREATE INDEX
  - DROP INDEX
- 视图：
  - CREATE VIEWS
  - DROP VIEWS
- 触发器：
  - CREATE TRIGGERS
  - DROP TRIGGERS

CREATE TABLE

CREATE TABLE 语句由关键字 CREATE TABLE 后跟新表的名称以及列定义和约束的列表（在括号中）组成。表名可以是标识符或字符串。

```
sql-statement       ::=  CREATE [TEMP | TEMPORARY] TABLE [IF NOT EXISTS] [database-name.] table-name
                         ( column-def [, column-def]* [, constraint]* )
sql-statement       ::=  CREATE [TEMP | TEMPORARY] TABLE [database-name.] table-name AS select-statement
column-def          ::=  name [type] [[CONSTRAINT name] column-constraint]*
type                ::=  typename | typename ( number ) | typename ( number , number )
column-constraint   ::=  NOT NULL [ conflict-clause ] |
                         PRIMARY KEY [sort-order] [ conflict-clause ] [AUTOINCREMENT] |
                         UNIQUE [conflict-clause] |
                         CHECK ( expr ) |
                         DEFAULT default-value |
                         COLLATE collation-name
constraint          ::=  PRIMARY KEY ( column-list ) [conflict-clause] |
                         UNIQUE ( column-list ) [conflict-clause] |
                         CHECK ( expr )
conflict-clause     ::=  ON CONFLICT conflict-algorithm
conflict-algorithm  ::=  ROLLBACK | ABORT | FAIL | IGNORE | REPLACE
default-value       ::=  NULL | string | number | CURRENT_TIME | CURRENT_DATE | CURRENT_TIMESTAMP
sort-order          ::=  ASC | DESC
collation-name      ::=  BINARY | NOCASE
column-list         ::=  column-name [, column-name]*
```

每个列定义都是列的名称后跟该列的数据类型，然后是一个或多个可选的列约束。列的数据类型限制可在该列中存储的数据。如果尝试在具有不同数据类型的列中存储某个值，则运行时会将该值转换为相应的类型 （如果可能），或者引发错误。有关其他信息，请参阅数据类型支持部分。

NOT NULL 列约束指示列不能包含 NULL 值。

UNIQUE 约束导致在指定的一个或多个列上创建索引。此索引必须包含唯一键 — 没有任何两行可以包含重复值或者指定的一个或多个列的值的组合。CREATE TABLE 语句可具有多个 UNIQUE 约束（包括列定义中有 UNIQUE 约束的多个列）和 / 或多个表级 UNIQUE 约束。

CHECK 约束定义计算结果必须为 true 的表达式，以便插入或更新行的数据。 CHECK 表达式必须解析为布尔值。

列定义中的 COLLATE 子句指定在比较列的文本项时要使用的文本排序规则函数。默认情况下，使用 BINARY 排序规则函数。有关 COLLATE 子句和排序规则函数的详细信息，请参阅 COLLATE。

DEFAULT 约束指定执行 INSERT 时要使用的默认值。该值可能是 NULL、字符串常量或数字。默认值还可能是与大小写无关的特殊关键字 CURRENT_TIME、CURRENT_DATE 或 CURRENT_TIMESTAMP 之一。如果该值是 NULL、字符串常量或数字，则只要 INSERT 语句不指定列的值，则按字面将它插入到列中。如果该值是 CURRENT_TIME、CURRENT_DATE 或 CURRENT_TIMESTAMP，则将当前的 UTC 日期和 / 或时间插入到列中。对于 CURRENT_TIME，格式为 HH:MM:SS。对于 CURRENT_DATE，格式为 YYYY-MM-DD。CURRENT_TIMESTAMP 的格式为 YYYY-MM-DD HH:MM:SS。

指定 PRIMARY KEY 通常仅在对应的一个或多个列上创建 UNIQUE 索引。但是，如果 PRIMARY KEY 约束在具有数据类型 INTEGER （或它的一个同义词，如 int）的单个列上，则数据库会将该列用作表的实际主键。这意味着该列只能保存唯一整数值。（注意：在许多 SQLite 实现中，只有列类型 INTEGER 会导致列用作内部主键，但在 Adobe AIR 中， INTEGER 的同义词，如 int，也会指定该行为。）

如果表没有 INTEGER PRIMARY KEY 列，则在插入行时将自动生成整数键。使用特殊名称 ROWID、 OID 或 _ROWID_ 之一，始终可以访问行的主键。可使用这些名称，而不管它是明确声明的 INTEGER PRIMARY KEY 还是内部生成的值。但是，如果表具有显式 INTEGER PRIMARY KEY，则结果数据中的列名称是实际列名称，而不是特殊名称。

INTEGER PRIMARY KEY 列还可包括关键字 AUTOINCREMENT。如果使用 AUTOINCREMENT 关键字，则在执行未指定列的显式值的 INSERT 语句时，数据库会在 INTEGER PRIMARY KEY 列中自动生成并插入按顺序递增的整数键。

CREATE TABLE 语句中只能有一个 PRIMARY KEY 约束。它可以是一个列的定义的一部分或一个单表级 PRIMARY KEY 约束。主键列隐式为 NOT NULL。

许多约束后面的可选 conflict-clause 允许为该约束指定替代的默认约束冲突解决算法。默认值为 ABORT。同一表中的不同约束可能具有不同的默认冲突解决算法。如果 INSERT 或 UPDATE 语句指定不同的冲突解决算法，则使用该算法，替代在 CREATE TABLE 语句中指定的算法。请参阅第 301 页的“特殊的语句和子句”的 ON CONFLICT 部分，了解更多信息。

其他约束（如 FOREIGN KEY 约束）不会导致错误，但运行时会忽略它们。

如果在 CREATE 和 TABLE 之间出现 TEMP 或 TEMPORARY 关键字，则创建的表仅在同一数据库连接（SQLConnection 实例）内才是可见的。关闭数据库连接时会自动删除它。在临时表上创建的任何索引也是临时的。临时的表和索引存储在与主数据库文件不同的单独文件中。

如果指定可选的 database-name 前缀，则在指定的数据库（通过使用指定的数据库名称调用 attach() 方法连接到 SQLConnection 实例的数据库）中创建表。除非 database-name 前缀是 temp，否则同时指定 database-name 前缀和 TEMP 关键字是错误的。如果未指定数据库名称，而且不存在 TEMP 关键字，则在主数据库（使用 open() 或 openAsync() 方法连接到 SQLConnection 实例的数据库）中创建表。

对表中的列数或约束数没有任何限制。对行中的数据量也没有任何限制。

CREATE TABLE AS 形式将表定义为查询的结果集。表列的名称是结果中列的名称。

如果存在可选的 IF NOT EXISTS 子句，而且同名的其他表已存在，则数据库将忽略 CREATE TABLE 命令。

可使用 DROP TABLE 语句删除表，并可使用 ALTER TABLE 语句进行有限的更改。

ALTER TABLE

ALTER TABLE 命令允许用户重命名现有表或向其添加新列。无法从表中删除列。

```
sql-statement ::= ALTER TABLE [database-name.] table-name alteration
alteration    ::= RENAME TO new-table-name
alteration    ::= ADD [COLUMN] column-def
```

RENAME TO 语法用于将由 [database-name.] table-name 标识的表重命名为 new-table-name。此命令不能用于在附加的数据库之间移动表，而只能用于重命名同一数据库中的表。

如果要重命名的表具有触发器或索引，则在重命名表后，这些触发器或索引仍然附加到表。但是，如果存在任何视图定义或由引用要重命名的表的触发器执行的语句，则不会自动修改它们以使用新表名。如果重命名的表具有关联的视图或触发器，则必须手动删除再重新创建使用新表名的触发器或视图定义。

ADD [COLUMN] 语法用于向现有表添加新列。新列始终追加到现有列的列表的结尾。 column-def 子句可采用 CREATE TABLE 语句中允许的任何形式，但有以下限制：

- 列不能具有 PRIMARY KEY 或 UNIQUE 约束。
- 列不能具有默认值 CURRENT_TIME、 CURRENT_DATE 或 CURRENT_TIMESTAMP。
- 如果指定了 NOT NULL 约束，则列必须具有除 NULL 之外的默认值。

ALTER TABLE 语句的执行时间不受表中数据量的影响。

DROP TABLE

DROP TABLE 语句删除使用 CREATE TABLE 语句添加的表。具有指定 table-name 的表就是所删除的表。它将从数据库和磁盘文件中完全删除。无法恢复表。与表关联的所有索引也将被删除。

```
sql-statement  ::=  DROP TABLE [IF EXISTS] [database-name.] table-name
```

默认情况下， DROP TABLE 语句不减小数据库文件的大小。将保留数据库中的空白空间，并在后续 INSERT 操作中使用。若要删除数据库中的可用空间，请使用 SQLConnection.clean() 方法。如果在最初创建数据库时 autoClean 参数设置为 true，则将自动释放空间。

可选的 IF EXISTS 子句抑制表不存在时通常会导致的错误。

CREATE INDEX

CREATE INDEX 命令包含关键字 CREATE INDEX，后跟新索引的名称、关键字 ON、之前创建的要编制索引的表的名称，以及表中其值用于索引键的列的名称的带括号列表。

```
sql-statement  ::=  CREATE [UNIQUE] INDEX [IF NOT EXISTS] [database-name.] index-name
                    ON table-name ( column-name [, column-name]* )
column-name    ::=  name [COLLATE collation-name] [ASC | DESC]
```

每个列名都可后跟 ASC 或 DESC 关键字以指示排序顺序，但运行时会忽略指定的排序顺序。排序始终按升序执行。

每个列名后面的 COLLATE 子句定义用于该列中文本值的排序规则序列。默认的排序规则序列是在 CREATE TABLE 语句中为该列定义的排序规则序列。如果未指定排序规则序列，则使用 BINARY 排序规则序列。有关 COLLATE 子句和排序规则函数的定义，请参阅 COLLATE。

对于可附加到单个表的索引数没有任何限制。对于索引中的列数也没有限制。

DROP INDEX

drop index 语句删除使用 CREATE INDEX 语句添加的语句。指定的索引将从数据库文件中完全删除。恢复索引的唯一方法是，重新输入相应的 CREATE INDEX 命令。

```
sql-statement ::= DROP INDEX [IF EXISTS] [database-name.] index-name
```

默认情况下， DROP INDEX 语句不减小数据库文件的大小。将保留数据库中的空白空间，并在后续 INSERT 操作中使用。若要删除数据库中的可用空间，请使用 SQLConnection.clean() 方法。如果在最初创建数据库时 autoClean 参数设置为 true，则将自动释放空间。

CREATE VIEW

CREATE VIEW 命令为预定义的 SELECT 语句分配一个名称。之后，此新名称可用于其他 SELECT 语句的 FROM 子句中来代替表名。视图通常用于简化查询，方法是将一组复杂的 （和频繁使用的）数据组合到可在其他操作中使用的结构中。

```
sql-statement ::= CREATE [TEMP | TEMPORARY] VIEW [IF NOT EXISTS] [database-name.] view-name AS select-
statement
```

如果在 CREATE 和 VIEW 之间出现 TEMP 或 TEMPORARY 关键字，则创建的视图仅对已打开数据库的 SQLConnection 实例是可见的，并且在关闭数据库时会自动删除该视图。

如果指定了 [database-name]，则使用指定的 name 参数在指定的数据库 （使用 attach() 方法连接到 SQLConnection 实例的数据库）中创建视图。除非 [database-name] 是 temp，否则同时指定 [database-name] 和 TEMP 关键字是错误的。如果未指定数据库名称，而且不存在 TEMP 关键字，则在主数据库（使用 open() 或 openAsync() 方法连接到 SQLConnection 实例的数据库）中创建视图。

视图是只读的。除非至少定义一个关联类型 （INSTEAD OF DELETE、 INSTEAD OF INSERT、 INSTEAD OF UPDATE）的触发器，否则不能对视图使用 DELETE、INSERT 或 UPDATE 语句。有关为视图创建触发器的信息，请参阅 CREATE TRIGGER。

使用 DROP VIEW 语句可从数据库中删除视图。

DROP VIEW

DROP VIEW 语句删除由 CREATE VIEW 语句创建的视图。

```
sql-statement ::= DROP VIEW [IF EXISTS] view-name
```

指定的 view-name 是要删除的视图的名称。将从数据库中删除它，但不会修改基础表中的数据。

CREATE TRIGGER

create trigger 语句用于向数据库架构添加触发器。触发器是在发生指定的数据库事件 (database-event) 时自动执行的数据库操作 (trigger-action)。

```
sql-statement   ::=  CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] [database-name.] trigger-name
                     [BEFORE | AFTER] database-event
                     ON table-name
                     trigger-action
sql-statement   ::=  CREATE [TEMP | TEMPORARY] TRIGGER [IF NOT EXISTS] [database-name.] trigger-name
                     INSTEAD OF database-event
                     ON view-name
                     trigger-action
database-event  ::=  DELETE |
                     INSERT |
                     UPDATE |
                     UPDATE OF column-list
trigger-action  ::=  [FOR EACH ROW] [WHEN expr]
                     BEGIN
                       trigger-step ;
                       [ trigger-step ; ]*
                     END
trigger-step    ::=  update-statement |
                     insert-statement |
                     delete-statement |
                     select-statement
column-list     ::=  column-name [, column-name]*
```

触发器被指定为只要出现以下条件就会激发：发生特定数据库表的 DELETE、 INSERT 或 UPDATE，或者更新了表的一个或多个指定列的 UPDATE。除非使用了 TEMP 或 TEMPORARY 关键字，否则触发器是永久性的。在这种情况下，当关闭 SQLConnection 实例的主数据库连接时，将删除触发器。如果未指定时间 （BEFORE 或 AFTER），则触发器默认为 BEFORE。

仅支持 FOR EACH ROW 触发器，因此 FOR EACH ROW 文本是可选的。对于 FOR EACH ROW 触发器，如果 WHEN 子句表达式的计算结果为 true，则对导致触发器激发的语句所插入、更新或删除的每个数据库行执行 trigger-step 语句。

如果提供了 WHEN 子句，则仅对 WHEN 子句为 true 的行执行指定为触发器步骤的 SQL 语句。如果未提供 WHEN 子句，则对所有行执行 SQL 语句。

在触发器体 （trigger-action 子句）中，受影响表的更改之前值和更改之后值使用特殊的表名 OLD 和 NEW 提供。 OLD 和 NEW 表的结构与创建触发器的表的结构匹配。 OLD 表包含由触发语句修改或删除的任何行，处于其在触发语句操作之前的状态。 NEW 表包含由触发语句修改或创建的任何行，处于其在触发语句操作之后的状态。 WHEN 子句和 trigger-step 语句都可访问使用 NEW.column-name 和 OLD.column-name 形式的引用插入、删除或更新的行的值，其中 column-name 是与触发器关联的表中的列的名称。 OLD 和 NEW 表引用的可用性取决于触发器所处理的 database-event 类型：

- INSERT – NEW 引用有效
- UPDATE – NEW 和 OLD 引用有效
- DELETE – OLD 引用有效

指定的时间（BEFORE、AFTER 或 INSTEAD OF）确定何时执行与插入、修改或删除关联行有关的 trigger-step 语句。可将 ON CONFLICT 子句指定为 trigger-step 中 UPDATE 或 INSERT 语句的一部分。但是，如果将 ON CONFLICT 子句指定为导致触发器激发的语句的一部分，则改用该冲突处理策略。

除了表触发器外，还可在视图上创建 INSTEAD OF 触发器。如果在某个视图上定义了一个或多个 INSTEAD OF INSERT、INSTEAD OF DELETE 或 INSTEAD OF UPDATE 触发器，则在该视图上执行关联类型的语句（INSERT、DELETE 或 UPDATE）不会被视为错误。在这种情况下，在视图上执行 INSERT、DELETE 或 UPDATE 会导致关联触发器激发。由于触发器是 INSTEAD OF 触发器，因此导致触发器激发的语句不会修改视图的基础表。但是，触发器可用于对基础表执行修改操作。

在具有 INTEGER PRIMARY KEY 列的表上创建触发器时，请牢记一个重要问题。如果 BEFORE 触发器修改要由导致触发器激发的语句更新的行的 INTEGER PRIMARY KEY 列，则不会发生更新。解决方法是创建具有 PRIMARY KEY 列而不是 INTEGER PRIMARY KEY 列的表。

可使用 DROP TRIGGER 语句删除触发器。删除表或视图时，也会自动删除与该表或视图关联的所有触发器。

RAISE() 函数

特殊的 SQL 函数 RAISE() 可用于触发器的 trigger-step 语句中。此函数的语法如下：

```
raise-function  ::=  RAISE ( ABORT, error-message ) |
                     RAISE ( FAIL, error-message ) |
                     RAISE ( ROLLBACK, error-message ) |
                     RAISE ( IGNORE )
```

在触发器执行期间调用前三种形式之一时，会执行指定的 ON CONFLICT 处理操作 （ABORT、 FAIL 或 ROLLBACK），而且当前语句的执行结束。 ROLLBACK 被认为是语句执行失败，因此执行其 execute() 方法的 SQLStatement 实例将调度 error (SQLErrorEvent.ERROR) 事件。被调度的事件对象的 error 属性中的 SQLError 对象将其 details 属性设置为在 RAISE() 函数中指定的 error-message。

调用 RAISE(IGNORE) 时，将放弃当前触发器的剩余部分、导致触发器执行的语句和执行的任何后续触发器。不回滚数据库更改。如果导致触发器执行的语句本身是触发器的一部分，则在下一步开始时该触发器程序将继续执行。有关冲突解决算法的详细信息，请参阅 ON CONFLICT （冲突算法）部分。

DROP TRIGGER

DROP TRIGGER 语句删除由 CREATE TRIGGER 语句创建的触发器。

```
sql-statement  ::=  DROP TRIGGER [IF EXISTS] [database-name.] trigger-name
```

将从数据库中删除触发器。请注意，在删除其关联表时，会自动删除触发器。

特殊的语句和子句

本部分介绍几个对运行时提供的 SQL 进行扩展的子句，以及可在许多语句、注释和表达式中使用的两个语言元素。

COLLATE

COLLATE 子句在 SELECT、CREATE TABLE 和 CREATE INDEX 语句中使用，指定对值进行比较或排序时使用的比较算法。

```
sql-statement   ::=  COLLATE collation-name
collation-name  ::=  BINARY | NOCASE
```

列的默认排序规则类型是 BINARY。将 BINARY 排序规则用于 TEXT 存储类的值时，通过比较内存中表示值的字节来执行二进制排序规则，而不考虑文本编码。

NOCASE 排序规则序列仅应用于 TEXT 存储类的值。在使用时， NOCASE 排序规则执行不区分大小写的比较。

排序规则序列不用于 NULL、 BLOB、 INTEGER 或 REAL 类型的存储类。

若要将除 BINARY 之外的排序规则类型用于列，必须将 COLLATE 子句指定为 CREATE TABLE 语句中列定义的一部分。每当对两个 TEXT 值进行比较时，都将按照以下规则使用排序规则序列来确定比较的结果：

- 对于二进制比较运算符 （=、 <、 >、 <= 和 >=），如果任一操作数为列，则列的默认排序规则类型确定用于比较的排序规则序列。如果两个操作数都是列，则左操作数的排序规则类型确定所用的排序规则序列。如果操作数都不是列，则使用 BINARY 排序规则序列。
- BETWEEN...AND 运算符等效于使用两个包含 >= 和 <= 运算符的表达式。例如，表达式 x BETWEEN y AND z 等效于 x >= y AND x <= z。因此， BETWEEN...AND 运算符按照上述规则来确定排序规则序列。
- IN 运算符的行为与 = 运算符类似，用于确定要使用的排序规则序列。例如，如果 x 是列，则用于表达式 x IN (y, z) 的排序规则序列是 x 的默认排序规则类型。否则，使用 BINARY 排序规则。
- 可为属于 SELECT 语句的 ORDER BY 子句明确分配要用于排序操作的排序规则序列。在这种情况下，始终使用显式排序规则序列。否则，如果由 ORDER BY 子句排序的表达式是一个列，则使用列的默认排序规则类型确定排序顺序。如果表达式不是一个列，则使用 BINARY 排序规则序列。

EXPLAIN

EXPLAIN 命令修饰符是 SQL 的非标准扩展。

```
sql-statement  ::=  EXPLAIN sql-statement
```

如果 EXPLAIN 关键字出现在任何其他 SQL 语句之前，则结果报告它用于执行命令的虚拟机指令序列，而不是实际执行命令，就像不存在 EXPLAIN 关键字一样。 EXPLAIN 功能是一种高级功能，允许开发人员更改 SQL 语句文本以尝试优化性能或调试看起来工作不正常的语句。

ON CONFLICT （冲突算法）

ON CONFLICT 子句不是单独的 SQL 命令。它是可出现在许多其他 SQL 命令中的非标准子句。

```
conflict-clause     ::=  ON CONFLICT conflict-algorithm
conflict-clause     ::=  OR conflict-algorithm
conflict-algorithm  ::=  ROLLBACK |
                         ABORT |
                         FAIL |
                         IGNORE |
                         REPLACE
```

ON CONFLICT 子句的第一种形式 （使用关键字 ON CONFLICT）用于 CREATE TABLE 语句中。对于 INSERT 或 UPDATE 语句，使用第二种形式（将 ON CONFLICT 替换为 OR）以使语法看起来更自然。例如，语句变为 INSERT OR IGNORE，而不再是 INSERT ON CONFLICT IGNORE。虽然关键字是不同的，但是子句的含义在任一形式中都是相同的。

ON CONFLICT 子句指定用于解决约束冲突的算法。五种算法为 ROLLBACK、 ABORT、 FAIL、 IGNORE 和 REPLACE。默认算法为 ABORT。以下是对这五种冲突算法的说明：

ROLLBACK 出现约束冲突时，会立即发生 ROLLBACK，结束当前的事务。命令将中止， SQLStatement 实例调度 error 事件。如果没有事务处于活动状态 （在每个命令上创建的隐含事务除外），则此算法的作用与 ABORT 相同。

ABORT 出现约束冲突时，命令将撤消它之前可能已进行的任何更改， SQLStatement 实例会调度 error 事件。不执行 ROLLBACK，因此将保留以前命令在事务中进行的更改。 ABORT 是默认行为。

FAIL 出现约束冲突时，命令将中止， SQLStatement 会调度 error 事件。但是，将保留而不撤消在遇到约束违反之前语句对数据库进行的任何更改。例如，如果 UPDATE 语句在它尝试更新的第 100 行上遇到约束冲突，则保留对前 99 行的更改，但不会更改第 100 行和之后的行。

IGNORE 出现约束冲突时，不插入或更改包含约束冲突的一行。除了忽略此行外，命令通常会继续正常执行。通常会继续正常地插入或更新包含约束违反的行之前和之后的其他行。不返回错误。

REPLACE 出现 UNIQUE 约束冲突时，在插入或更新当前行之前，删除导致约束冲突的预先存在的行。因此，插入或更新会始终进行，而且命令通常会继续正常执行。不返回错误。如果出现 NOT NULL 约束冲突，则将 NULL 值替换为该列的默认值。如果列没有默认值，则使用 ABORT 算法。如果出现 CHECK 约束冲突，则使用 IGNORE 算法。此冲突解决策略删除行以便满足约束时，它不会调用这些行上的删除触发器。

在 INSERT 或 UPDATE 语句的 OR 子句中指定的算法将覆盖在 CREATE TABLE 语句中指定的任何算法。如果未在 CREATE TABLE 语句或者执行 INSERT 或 UPDATE 语句中指定算法，则将使用 ABORT 算法。

REINDEX

REINDEX 命令用于删除并重新创建一个或多个索引。在排序规则序列的定义更改时，此命令很有用。

```
sql-statement  ::=  REINDEX collation-name
sql-statement  ::=  REINDEX [database-name .] ( table-name | index-name )
```

在第一种形式中，将重新创建使用指定排序规则序列的所有附加数据库中的所有索引。在第二种形式中，指定 table-name 时，将重新生成与表关联的所有索引。如果提供了 index-name，则仅删除并重新创建指定的索引。

COMMENTS

注释不是 SQL 命令，但它们可出现在 SQL 查询中。运行时将它们视为空白。它们可从能找到空白的任何位置开始，包括跨多行的表达式的内部。

```
comment             ::=  single-line-comment |
                         block-comment
single-line-comment ::=  -- single-line
block-comment       ::=  /* multiple-lines or block [*/]
```

单行注释由两个短划线指示。单行注释仅扩展到当前行的结尾。

块注释可跨任何数目的行，或者嵌入到单行中。如果没有终止分隔符，则块注释可扩展到输入的结尾。此情况并不被视为错误。新的 SQL 语句可在块注释结束后的行上开始。块注释可嵌入到能出现空白的任何位置中，包括表达式内部和其他 SQL 语句的中间。块注释不嵌套。忽略块注释内的单行注释。

EXPRESSIONS

表达式是其他 SQL 块中的子命令。以下介绍 SQL 语句中表达式的有效语法：

```
expr            ::=  expr binary-op expr |
                     expr [NOT] like-op expr [ESCAPE expr] |
                     unary-op expr |
                     ( expr ) |
                     column-name |
                     table-name.column-name |
                     database-name.table-name.column-name |
                     literal-value |
                     parameter |
                     function-name( expr-list | * ) |
                     expr ISNULL |
                     expr NOTNULL |
                     expr [NOT] BETWEEN expr AND expr |
                     expr [NOT] IN ( value-list ) |
                     expr [NOT] IN ( select-statement ) |
                     expr [NOT] IN [database-name.] table-name |
                     [EXISTS] ( select-statement ) |
                     CASE [expr] ( WHEN expr THEN expr )+ [ELSE expr] END |
                     CAST ( expr AS type ) |
                     expr COLLATE collation-name
like-op         ::=  LIKE | GLOB
binary-op       ::=  see Operators
unary-op        ::=  see Operators
parameter       ::=  :param-name | @param-name | ?
value-list      ::=  literal-value [, literal-value]*
literal-value   ::=  literal-string | literal-number | literal-boolean | literal-blob | literal-null
literal-string  ::=  'string value'
literal-number  ::=  integer | number
literal-boolean  ::=  true | false
literal-blob  ::=  X'string of hexadecimal data'
literal-null  ::=  NULL
```

表达式是值和可解析为单个值的运算符的任何组合。根据表达式解析为布尔值（true 或 false）还是解析为非布尔值，可将表达式分为两种常规类型。

在几种常见情况下，包括在 WHERE 子句、HAVING 子句、JOIN 子句中的 ON 表达式以及 CHECK 表达式中，表达式必须解析为布尔值。以下类型的表达式满足此条件：

- ISNULL
- NOTNULL
- IN ()
- EXISTS ()

- LIKE
- GLOB
- 某些函数
- 某些运算符 （特别是比较运算符）

字面值
文本数字值是以整数或浮点数形式书写的。支持科学记数法。 . （句点）字符始终用作小数点。

字符串文本是通过用单引号 ' 将字符串括起来指示的。若要在字符串中加入单引号，请在一行中连续放置两个单引号，例如 ''。

布尔文本由值 true 或 false 指示。文本布尔值用于 Boolean 列数据类型。

BLOB 文本是包含十六进制数据且前面有单个 x 或 X 字符的字符串文本，例如 X'53514697465'。

字面值也可以是标记 NULL。

列名
列名可以是在 CREATE TABLE 语句中定义的任何名称或以下特殊标识符之一：ROWID、 OID 或 _ROWID_。这些特殊标识符都描述与每个表的每一行关联的唯一随机整数键 （"行键"）。如果 CREATE TABLE 语句未定义同名的真正列，则特殊标识符仅引用行键。行键充当只读列。行键可在能使用常规列的任何位置使用，但您不能在 UPDATE 或 INSERT 语句中更改行键的值。 SELECT * FROM table 语句在其结果集中不包括行键。

SELECT 语句
SELECT 语句可在表达式中作为 IN 运算符的右操作数，作为纯量 （单个结果值）或者作为 EXISTS 运算符的操作数。用作纯量或 IN 运算符的操作数时， SELECT 在其结果中只能具有单个列。允许使用复合 SELECT 语句 （通过诸如 UNION 或 EXCEPT 之类的关键字连接）。在使用 EXISTS 运算符时，忽略 SELECT 的结果集中的列。如果存在一个或多个行，则表达式返回 TRUE ；如果结果集为空，则返回 FALSE。如果 SELECT 表达式中没有项引用包含查询中的值，则该表达式将在任何其他处理之前计算一次，并在必要时重用结果。如果 SELECT 表达式不包含外部查询 （称为相关子查询）中的变量，则 SELECT 在每次需要时都重新进行计算。

当 SELECT 是 IN 运算符的右操作数时，如果左操作数的结果等于 SELECT 语句结果集中的任一值，则 IN 运算符返回 TRUE。可在 IN 运算符前面加上 NOT 关键字以反转测试的意义。

如果 SELECT 出现在表达式内但不是 IN 运算符的右操作数，则 SELECT 结果的第一行将成为表达式中使用的值。如果 SELECT 生成多个结果行，则忽略第一行后的所有行。如果 SELECT 未生成行，则 SELECT 的值为 NULL。

CAST 表达式
CAST 表达式将指定的值的数据类型更改为提供的类型。指定的类型可以是对 CREATE TABLE 语句的列定义中的类型有效的任何非空类型名称。有关详细信息，请参阅数据类型支持。

其他表达式元素
以下部分介绍可在表达式中使用的其他 SQL 元素：

- 内置函数：聚合函数，标量函数，日期和时间格式函数
- 运算符
- 参数

内置函数

内置函数分为以下三种主要类别：

- 聚合函数

- 标量函数
- 日期和时间格式函数

除了这些函数外，还有一个用于提供执行触发器时的错误通知的特殊函数 RAISE()。此函数只能在 CREATE TRIGGER 语句体使用。有关 RAISE() 函数的信息，请参阅 CREATE TRIGGER > RAISE()。

与 SQL 中的所有关键字一样，函数名称不区分大小写。

聚合函数

聚合函数对多行中的值执行操作。这些函数主要在 SELECT 语句中与 GROUP BY 子句一起使用。

AVG(X)	返回一个组中所有非 NULL X 的平均值。看起来不像数字的字符串和 BLOB 值被解释为 0。AVG() 的结果始终是浮点值，即使所有输入都是整数。
COUNT(X) COUNT(*)	第一种形式返回组中不是 NULL 的 X 次数的计数。第二种形式 （使用 * 参数）返回组中的总行数。
MAX(X)	返回组中所有值的最大值。通常的排序顺序用于确定最大值。
MIN(X)	返回组中所有值的最小非 NULL 值。通常的排序顺序用于确定最小值。如果组中的所有值都是 NULL，则返回 NULL。
SUM(X) TOTAL(X)	返回组中所有非 NULL 值的数字和。如果所有值都是 NULL，则 SUM() 返回 NULL，TOTAL() 返回 0.0。TOTAL() 的结果始终是浮点值。如果所有非 NULL 输入都是整数，则 SUM() 的结果是整数值。如果 SUM() 的任一输入不是整数且不是 NULL，则 SUM() 返回浮点值。此值可能是真实和的近似值。

在采用单个参数的上述任一聚合函数中，可在该参数前面加上关键字 DISTINCT。在这种情况下，在将重复的元素传递到聚合函数之前，会对其进行筛选。例如，函数调用 COUNT(DISTINCT x) 返回列 X 的非重复值数目，而不是列 x 中非 NULL 值的总数。

标量函数

标量函数一次对一行上的值进行运算。

ABS(X)	返回参数 X 的绝对值。
COALESCE(X, Y, ...)	返回第一个非 NULL 参数的副本。如果所有参数都为 NULL，则返回 NULL。必须至少有两个参数。
GLOB(X, Y)	此函数用于实现 X GLOB Y 语法。
IFNULL(X, Y)	返回第一个非 NULL 参数的副本。如果两个参数都为 NULL，则返回 NULL。此函数的行为与 COALESCE() 相同。
HEX(X)	该参数被解释为 BLOB 存储类型的值。结果是该值内容的十六进制呈现。
LAST_INSERT_ROWID()	返回通过当前的 SQLConnection 插入到数据库的最后一行的行标识符 （生成的主键）。此值与 SQLConnection.lastInsertRowID 属性返回的值相同。
LENGTH(X)	返回 X 的字符串长度 （以字符计）。
LIKE(X, Y [, Z])	此函数用于实现 SQL 的 X LIKE Y [ESCAPE Z] 语法。如果存在可选的 ESCAPE 子句，则使用三个参数调用该函数。否则，仅使用两个参数调用它。
LOWER(X)	返回字符串 X 的副本，其中所有字符都已转换为小写形式。
LTRIM(X) LTRIM(X, Y)	返回通过删除 X 左侧的空格而形成的字符串。如果指定了 Y 参数，则函数将从 X 的左侧删除 Y 中的任何字符。
MAX(X, Y, ...)	返回具有最大值的参数。除了数字外，参数还可能是字符串。最大值由定义的排序顺序确定。请注意，当 MAX() 具有 2 个或更多参数时，它是简单函数，而它只有一个参数时则是聚合函数。
MIN(X, Y, ...)	返回具有最小值的参数。除了数字外，参数还可能是字符串。最小值由定义的排序顺序确定。请注意， MIN() 在具有 2 个或更多参数时是简单函数，在只有一个参数时则是聚合函数。
NULLIF(X, Y)	如果参数是不同的，则返回第一个参数，否则返回 NULL。
QUOTE(X)	此例程返回一个字符串，该字符串是其适合于包含到另一 SQL 语句中的参数的值。字符串括在单引号中，并根据需要对内部引号转义。 BLOB 存储类作为十六进制文本进行编码。在编写触发器以实现撤消 / 重做功能时，此函数是很有用的。
RANDOM(*)	返回一个介于 -9223372036854775808 和 9223372036854775807 之间的伪随机整数。此随机值不是强加密的。
RANDOMBLOB(N)	返回一个包含伪随机字节的 N 字节 BLOB。 N 应该是一个正整数。此随机值不是强加密的。如果 N 的值为负，则返回单个字节。

ROUND(X) ROUND(X, Y)	将数值 X 舍入到小数点右侧的 Y 位。如果省略 Y 参数，则使用 0。
RTRIM(X) RTRIM(X, Y)	返回通过删除 X 右侧的空格而形成的字符串。如果指定了 Y 参数，则函数将从 X 的右侧删除 Y 中的任何字符。
SUBSTR(X, Y, Z)	返回输入字符串 X 中以第 Y 个字符开头且长度为 Z 个字符的子字符串。 X 的最左侧字符是索引位置 1。如果 Y 为负，则通过从右 （而不是从左）计数来查找子字符串的第一个字符。
TRIM(X) TRIM(X, Y)	返回通过删除 X 右侧的空格而形成的字符串。如果指定了 Y 参数，则函数将从 X 的右侧删除 Y 中的任何字符。
TYPEOF(X)	返回表达式 X 的类型。可能的返回值有 "null"、 "integer"、 "real"、 "text" 和 "blob"。有关数据类型的详细信息，请参阅数据类型支持。
UPPER(X)	返回已转换为全大写字母的输入字符串 X 的副本。
ZEROBLOB(N)	返回包含 0x00 的 N 字节的 BLOB。

日期和时间格式函数

日期和时间格式函数是一组用于创建带格式的日期和时间数据的标量函数。请注意，这些函数对字符串值和数字值进行运算，并返回字符串值和数字值。这些函数并不预定用于 DATE 数据类型。如果对其声明数据类型为 DATE 的列中的数据使用这些函数，则它们不会像预期的那样工作。

DATE(T, ...)	DATE() 函数返回一个包含日期的字符串，日期格式为 YYYY-MM-DD。第一个参数 (T) 指定在时间格式下找到的格式的时间字符串。可在时间字符串后指定任何数量的修饰符。可在修饰符下找到这些修饰符。
TIME(T, ...)	TIME() 函数返回一个包含时间的字符串，时间格式为 HH:MM:SS。第一个参数 (T) 指定在时间格式下找到的格式的时间字符串。可在时间字符串后指定任何数量的修饰符。可在修饰符下找到这些修饰符。
DATETIME(T, ...)	DATETIME() 函数返回一个包含日期和时间的字符串，日期和时间格式为 YYYY-MM-DD HH:MM:SS。第一个参数 (T) 指定在时间格式下找到的格式的时间字符串。可在时间字符串后指定任何数量的修饰符。可在修饰符下找到这些修饰符。
JULIANDAY(T, ...)	JULIANDAY() 函数返回一个数字，指示自格林尼治标准时间公元前 4714 年 11 月 24 日中午至所提供日期的天数。第一个参数 (T) 指定在时间格式下找到的格式的时间字符串。可在时间字符串后指定任何数量的修饰符。可在修饰符下找到这些修饰符。
STRFTIME(F, T, ...)	STRFTIME() 例程返回根据指定为第一个参数 F 的格式字符串设置格式的日期。格式字符串支持以下替换： %d - 一月中的某天 %f - 带小数的秒 SS.SSS %H - 小时 00-24 %j - 一年中的某天 001-366 %J - 儒略历日数 %m - 月份 01-12 %M - 分钟 00-59 %s - 自 1970-01-01 以来的秒数 %S - 秒 00-59 %w - 一周中的某天 0-6 （星期日 = 0） %W - 一年中的某周 00-53 %Y - 年份 0000-9999 %% - % 第二个参数 (T) 指定在时间格式下找到的格式的时间字符串。可在时间字符串后指定任何数量的修饰符。可在修饰符下找到这些修饰符。

时间格式

时间字符串可采用以下任一格式：

YYYY-MM-DD	2007-06-15
YYYY-MM-DD HH:MM	2007-06-15 07:30
YYYY-MM-DD HH:MM:SS	2007-06-15 07:30:59
YYYY-MM-DD HH:MM:SS.SSS	2007-06-15 07:30:59.152
YYYY-MM-DDTHH:MM	2007-06-15T07:30
YYYY-MM-DDTHH:MM:SS	2007-06-15T07:30:59
YYYY-MM-DDTHH:MM:SS.SSS	2007-06-15T07:30:59.152
HH:MM	07:30 （日期是 2000-01-01）
HH:MM:SS	7:30:59 （日期是 2000-01-01）
HH:MM:SS.SSS	07:30:59:152 （日期是 2000-01-01）
目前	当前的日期和时间 （按通用协调时间）。
DDDD.DDDD	儒略历日数是一个浮点数。

这些格式中的字符 T 是分隔日期和时间的文本字符 “T”。仅包括时间的格式假定日期是 2001-01-01。

修饰符

时间字符串可后跟用于更改日期或更改日期解释的零个或更多个修饰符。可用的修饰符如下：

NNN 天	要添加到时间的天数。
NNN 小时	要添加到时间的小时数。
NNN 分钟	要添加到时间的分钟数。
NNN.NNNN 秒	要添加到时间的秒数和毫秒数。
NNN 月	要添加到时间的月数。
NNN 年	要添加到时间的年数。
一月的开始	将时间后移到一月的开始。
一年的开始	将时间后移到一年的开始。
一天的开始	将时间后移到一天的开始。
工作日 N	将时间前移到指定的工作日。（0 = 星期日， 1 = 星期一，依此类推）。
本地时间	将日期转换为本地时间。
utc	将日期转换为通用协调时间。

运算符

SQL 支持大量的运算符 （其中包括大多数编程语言中存在的常见运算符）以及 SQL 独有的几个运算符。

常见运算符

在 SQL 块中允许以下二元运算符，它们按从最高到最低的优先顺序列出：

```
*     /     %
+     -
<<  >>  &  |
<  >=    >  >=
=     ==    !=    <> IN
AND
OR
```

支持的一元前缀运算符是：

```
 !     ~     NOT
```

可以认为 COLLATE 运算符是一元后缀运算符。 COLLATE 运算符具有最高的优先顺序。它始终比任何一元前缀运算符或任何二元运算符绑定得更紧密。

请注意，等号和不等号运算符各有两个变体。等号可以是 = 或 ==。不等号运算符可以是 != 或 <>。

|| 运算符是字符串串联运算符 — 它将其操作数的两个字符串联接在一起。

运算符 % 输出其左操作数以其右操作数为模的余数。

任何二元运算符的结果都是一个数字值，但给出字符串结果的 || 串联运算符除外。

SQL 运算符
LIKE

LIKE 运算符进行模式匹配比较。

```
expr     ::=  (column-name | expr) LIKE pattern
pattern  ::=  '[ string | % | _ ]'
```

LIKE 运算符右侧的操作数包含模式，而左侧的操作数包含对模式进行匹配的字符串。模式中的百分比符号 (%) 是一个通配符 — 它与字符串中零个或更多个字符的任何序列匹配。模式中的下划线 (_) 与字符串中的任何单个字符匹配。任何其他字符与自身匹配，或与其小写 / 大写形式匹配 （即匹配是以不区分大小写的方式执行的）。（注意：数据库引擎仅识别 7 位拉丁字符的大写 / 小写形式。因此，对于 8 位 iso8859 字符或 UTF-8 字符， LIKE 运算符区分大小写。例如，表达式 'a' LIKE 'A' 为 TRUE，但是 'æ' LIKE 'Æ' 为 FALSE）。可使用 SQLConnection.caseSensitiveLike 属性更改拉丁字符的区分大小写。

如果存在可选的 ESCAPE 子句，则 ESCAPE 关键字后面的表达式的计算结果必须是一个包含单个字符的字符串。可在 LIKE 模式中使用此字符，与文本百分比或下划线字符进行匹配。转义符后跟百分比符号、下划线或转义符本身分别与字符串中的文本百分比符号、下划线或转义符匹配。

GLOB

GLOB 运算符与 LIKE 类似，但是对其通配符使用 Unix 文件名替换语法。与 LIKE 不同， GLOB 区分大小写。

IN

IN 运算符计算其左操作数是否等于其右操作数 （括号中的一组值）中的值之一。

```
in-expr         ::=  expr [NOT] IN ( value-list ) |
                     expr [NOT] IN ( select-statement ) |
                     expr [NOT] IN [database-name.] table-name
value-list      ::=  literal-value [, literal-value]*
```

右操作数可以是一组逗号分隔的字面值，也可以是 SELECT 语句的结果。有关将 SELECT 语句用作 IN 操作符的右侧操作数的说明和限制，请参阅表达式中的 SELECT 语句。

BETWEEN...AND

BETWEEN...AND 运算符等效于使用两个包含 >= 和 <= 运算符的表达式。例如，表达式 x BETWEEN y AND z 等效于 x >= y AND x <= z。

NOT

NOT 运算符是一个求反运算符。可在 GLOB、 LIKE 和 IN 运算符前面加上 NOT 关键字来反转测试的意义 （换句话说，检查一个值是否与指示的模式不匹配）。

参数

参数在表达式中指定占位符，用于在运行时通过将值分配给 SQLStatement.parameters 关联数组来填充的字面值。参数可采用以下三种形式：

?	问号指示一个索引参数。根据参数在语句中的顺序，为其分配数字 （从零开始的）索引值。
:AAAA	冒号后跟标识符名称保存名为 AAAA 的命名参数的位置。命名参数也是根据它们在 SQL 语句中的顺序编号的。若要避免混淆，最好避免混合使用命名参数和编号参数。
@AAAA	“at 符号 ” 等效于冒号。

不支持的 SQL 功能

以下是在 Adobe AIR 中不支持的标准 SQL 元素的列表：

FOREIGN KEY 约束 分析但不强制执行 FOREIGN KEY 约束。

触发器 不支持 FOR EACH STATEMENT 触发器（所有触发器都必须为 FOR EACH ROW）。在表上不支持 INSTEAD OF 触发器 （仅在视图上允许 INSTEAD OF 触发器）。不支持递归触发器 （触发自身的触发器）。

ALTER TABLE 仅支持 ALTER TABLE 命令的 RENAME TABLE 和 ADD COLUMN 变体。其他种类的 ALTER TABLE 操作 （如 DROP COLUMN、 ALTER COLUMN、 ADD CONSTRAINT 等）将被忽略。

嵌套事务 仅允许单个活动事务。

RIGHT 和 FULL OUTER JOIN 不支持 RIGHT OUTER JOIN 或 FULL OUTER JOIN。

可更新的 VIEW 视图是只读的。不能对视图执行 DELETE、INSERT 或 UPDATE 语句。支持在尝试对视图执行 DELETE、INSERT 或 UPDATE 时激发的 INSTEAD OF 触发器，可使用它更新触发器体中的支持表。

GRANT 和 REVOKE 数据库是一个普通的磁盘文件；可应用的访问权限只有基础操作系统的常规文件访问权限。未实现通常在客户端 / 服务器 RDBMS 上找到的 GRANT 和 REVOKE 命令。

在一些 SQLite 实现中支持以下 SQL 元素和 SQLite 功能，但是在 Adobe AIR 中不支持它们。此功能的大部分可通过 SQLConnection 类的方法获取：

与事务相关的 SQL 元素 （BEGIN、 END、 COMMIT、 ROLLBACK） 通过 SQLConnection 类的与事务相关的方法 SQLConnection.begin()、 SQLConnection.commit() 和 SQLConnection.rollback() 可使用此功能。

ANALYZE 此功能可通过 SQLConnection.analyze() 方法获取。

ATTACH 此功能可通过 SQLConnection.attach() 方法获取。

COPY 不支持此语句。

CREATE VIRTUAL TABLE 不支持此语句。

DETACH 此功能可通过 SQLConnection.detach() 方法获取。

PRAGMA 不支持此语句。

VACUUM 此功能可通过 SQLConnection.compact() 方法获取。

无法访问系统表 系统表 （包括 sqlite_master 及其他具有 “sqlite_” 前缀的表）在 SQL 语句中不可用。运行时包括一个架构 API，用于提供面向对象的方法来访问架构数据。有关详细信息，请参阅 SQLConnection.loadSchema() 方法。

正则表达式函数 （MATCH() 和 REGEX()） 这些函数在 SQL 语句中不可用。

以下功能在许多 SQLite 实现和 Adobe AIR 之间是不同的：

索引语句参数 在许多实现中，索引语句参数从 1 开始。但是，在 Adobe AIR 中，索引语句参数是从零开始的 （即，为第一个参数指定索引 0，为第二个参数指定索引 1，依此类推）。

INTEGER PRIMARY KEY 列定义 在许多实现中，只有确切定义为 INTEGER PRIMARY KEY 的列才用作表的实际主键列。在这些实现中，使用通常为 INTEGER 的同义词 （例如 int）的其他数据类型不会导致将列用作内部主键。但是，在 Adobe AIR 中，int 数据类型（和其他 INTEGER 同义词）将视为完全等同于 INTEGER。因此，定义为 int PRIMARY KEY 的列用作表的内部主键。有关更多信息，请参阅 CREATE TABLE 和列关联部分。

其他 SQL 功能

默认情况下 SQLite 不支持以下列关联类型，但 Adobe AIR 却支持 （请注意，与 SQL 中的所有关键字一样，这些数据类型名称不区分大小写）：

Boolean 对应于 Boolean 类。

Date 对应于 Date 类。

int 对应于 int 类 （等效于 INTEGER 列关联）。

Number 对应于 Number 类 （等效于 REAL 列关联）。

Object 对应于 Object 类或可使用 AMF3 序列化和反序列化的任何子类。（这包括大多数类 （其中包括自定义类），但是不包括某些类 （其中包括显示对象以及将显示对象作为属性包括的对象）。）

String 对应于 String 类 （等效于 TEXT 列关联）。

XML 对应于 ActionScript (E4X) XML 类。

XMLList 对应于 ActionScript (E4X) XMLList 类。

在 SQLite 中默认情况下不支持以下字面值，但是在 Adobe AIR 中支持它们：

true 用于表示文本布尔值 true，以处理 BOOLEAN 列。

false 用于表示文本布尔值 false，以处理 BOOLEAN 列。

数据类型支持

与大多数 SQL 数据库不同， Adobe AIR SQL 数据库引擎不要求或强制表列包含某种类型的值。相反，运行时使用两个概念（存储类和列关联）来控制数据类型。本部分介绍存储类和列关联，以及如何在各种情况下解决数据类型差异：

- 第 310 页的 " 存储类 "
- 第 311 页的 " 列关联 "
- 第 313 页的 " 数据类型与比较运算符 "
- 第 313 页的 " 数据类型与数学运算符 "
- 第 313 页的 " 数据类型与排序 "
- 第 313 页的 " 数据类型与分组 "
- 第 313 页的 " 数据类型与复合 SELECT 语句 "

存储类

存储类表示用于在数据库中存储值的实际数据类型。下列存储类由数据库使用：

NULL 值为 NULL 值。

INTEGER 值为带符号的整数。

REAL 值为浮点数值。

TEXT 值为文本字符串 （限制为 256 MB）。

BLOB 值为二进制大型对象 (BLOB)；换句话说，为原始二进制数据 （限制为 256 MB）。

对于作为嵌入在 SQL 语句中的文本或使用参数绑定到准备的 SQL 语句的值提供给数据库的所有值，都在执行 SQL 语句之前为其分配存储类。

如果属于 SQL 语句的文本括在单引号或双引号中，则为其分配存储类 TEXT；如果文本被指定为没有小数点或指数的不带引号数字，则分配 INTEGER；如果文本是带有小数点或指数的不带引号数字，则分配 REAL；如果值为 NULL，则分配 NULL。具有存储类 BLOB 的文本是使用 X'ABCD' 表示法指定的。有关详细信息，请参阅表达式中的字面值。

对于作为使用 SQLStatement.parameters 关联数组的参数提供的值，为其分配与绑定的本机数据类型最紧密匹配的存储类。例如，int 值作为 INTEGER 存储类绑定，为 Number 值分配 REAL 存储类，为 String 值分配 TEXT 存储类，为 ByteArray 对象分配 BLOB 存储类。

列关联

列的关联是存储在该列中的数据的建议类型。一个值存储在列中（通过 INSERT 或 UPDATE 语句）时，运行时尝试将该值从其数据类型转换为指定的关联。例如，如果将 Date 值（ActionScript 或 JavaScript Date 实例）插入到一个其关联为 TEXT 的列中，则 Date 值在存储于数据库中之前将转换为 String 表示形式（等效于调用对象的 toString() 方法）。如果该值无法转换为指定的关联，则出现错误，且不执行操作。使用 SELECT 语句从数据库中检索值时，它作为对应于关联的类的实例返回，而不管它在被存储时是否已从不同数据类型转换。

如果一个列接受 NULL 值，则 ActionScript 或 JavaScript 值 null 可用作参数值在该列中存储 NULL。当 NULL 存储类值在 SELECT 语句中检索时，它始终作为 ActionScript 或 JavaScript 值 null 返回，而不管列的关联如何。如果一个列接受 NULL 值，则在尝试将值转换成不可为 Null 的类型（如 Number 或 Boolean）之前，始终检查从该列检索的值以确定它们是否为 null。

为数据库中的每个列分配以下类型关联之一：

- TEXT（或 String）
- NUMERIC
- INTEGER（或 int）
- REAL（或 Number）
- Boolean
- Date
- XML
- XMLLIST
- Object
- NONE

TEXT（或 **String**）

具有 TEXT 或 String 关联的列使用存储类 NULL、TEXT 或 BLOB 存储所有数据。如果将数字值插入到具有 TEXT 关联的列中，则在存储它之前将它转换为文本形式。

NUMERIC

具有 NUMERIC 关联的列包含使用存储类 NULL、 REAL 或 INTEGER 的值。将文本数据插入到 NUMERIC 列中时，在存储它之前，会尝试将它转换为整数或实数。如果转换成功，则使用 INTEGER 或 REAL 存储类存储值（例如，值 '10.05' 在被存储之前转换为 REAL 存储类）。如果无法执行转换，则出现错误。不会尝试转换 NULL 值。从 NUMERIC 列检索的值作为该值适合的最具体数字类型的实例返回。换句话说，如果该值是一个正整数或 0，则它作为 uint 实例返回。如果它是一个负整数，则它作为 int 实例返回。最后，如果它具有浮点部分（它不是一个整数），则它作为 Number 实例返回。

INTEGER（或 **int**）

使用 INTEGER 关联的列的行为方式与具有 NUMERIC 关联的列相同，但有一处不同。如果要存储的值是一个没有浮点部分的实数值（如 Number 实例），或者该值是可转换为没有浮点部分的实数值的文本值，则将它转换为整数并使用 INTEGER 存储类存储它。如果尝试存储具有浮点部分的实数值，则出现错误。

REAL（或 **Number**）

具有 REAL 或 NUMBER 关联的列的行为与具有 NUMERIC 关联的列类似，但是它将整数值强制为浮点表示形式。 REAL 列中的值始终作为 Number 实例从数据库返回。

Boolean

具有 Boolean 关联的列存储 true 或 false 值。 Boolean 列接受作为 ActionScript 或 JavaScript Boolean 实例的值。如果代码尝试存储字符串值，则将长度大于零的字符串视为 true，将空字符串视为 false。如果代码尝试存储数字数据，则任何非零值作为 true 存储，而 0 作为 false 存储。使用 SELECT 语句检索 Boolean 值时，它作为 Boolean 实例返回。非 NULL 值是使用 INTEGER 存储类存储的 （0 表示 false， 1 表示 true），并在检索数据时转换为 Boolean 对象。

Date

具有 Date 关联的列存储日期和时间值。Date 列用于接受作为 ActionScript 或 JavaScript Date 实例的值。如果尝试在 Date 列中存储 String 值，则运行时会尝试将该 String 值转换为罗马儒略历日期。如果转换失败，则出现错误。如果代码尝试存储 Number、int 或 uint 值，则不会尝试验证数据，而假定它是有效的罗马儒略历日期值。使用 SELECT 语句检索的 Date 值将自动转换为 Date 实例。使用 REAL 存储类将 Date 值存储为罗马儒略历日期值，因此排序和比较操作可以如预期的那样进行。

XML 或 XMLList

使用 XML 或 XMLList 关联的列存储 XML 结构。当代码尝试使用 SQLStatement 参数在 XML 列中存储数据时，运行时会尝试使用 ActionScript XML() 或 XMLList() 函数转换和验证值。如果该值无法转换为有效的 XML，则出现错误。如果存储数据的尝试使用 SQL 文本字面值（例如 INSERT INTO (col1) VALUES ('Invalid XML (no closing tag)')），则不分析或验证该值 — 而是假定它的格式正确。如果存储了无效值，则在检索它时，它作为空的 XML 对象返回。 XML 和 XMLList 数据是使用 TEXT 存储类或 NULL 存储类存储的。

Object

具有 Object 关联的列存储 ActionScript 或 JavaScript 复杂对象，其中包括 Object 类实例以及 Object 子类的实例 （如 Array 实例），甚至还包括自定义类实例。 Object 列数据以 AMF3 格式进行序列化，并使用 BLOB 存储类进行存储。在检索一个值时，它从 AMF3 进行反序列化，并在被存储时作为类的实例返回。请注意，某些 ActionScript 类 （尤其是显示对象）无法反序列化为其原始数据类型的实例。在存储自定义类实例之前，必须使用 flash.net.registerClassAlias() 方法 （或者在 Flex 中，通过向类声明添加 [RemoteObject] 元数据）为该类注册一个别名。此外，在检索该数据之前，必须为类注册相同的别名。无法正确进行反序列化的任何数据 （因为类本身无法进行反序列化，或者因为缺少类别名或类别名不匹配），在被存储时作为具有对应于原始实例的属性和值的匿名对象 （Object 类实例）返回。

NONE

对于具有关联 NONE 的列，使用任何存储类都是等效的。它不尝试在插入数据之前将其转换。

确定关联

列的类型关联由在 CREATE TABLE 语句中声明的列类型确定。在确定类型时，可应用以下规则 （不区分大小写）：

- 如果列的数据类型包含字符串 “CHAR”、“CLOB”、“STRI” 或 “TEXT” 中的任一个，则该列具有 TEXT/String 关联。请注意，类型 VARCHAR 包含字符串 “CHAR”，因此为其分配 TEXT 关联。
- 如果列的数据类型包含字符串 “BLOB”，或者未指定数据类型，则该列具有关联 NONE。
- 如果列的数据类型包含字符串 “XMLL”，则该列具有 XMLList 关联。
- 如果数据类型是字符串 “XML”，则列具有 XML 关联。
- 如果数据类型包含字符串 “OBJE”，则列具有 Object 关联。
- 如果数据类型包含字符串 “BOOL”，则列具有 Boolean 关联。
- 如果数据类型包含字符串 “DATE”，则列具有 Date 关联。
- 如果数据类型包含字符串 “INT” （包括 “UINT”），则为其分配 INTEGER/int 关联。
- 如果列的数据类型包含字符串 “REAL”、 “NUMB”、 “FLOA” 或 “DOUB” 中的任一个，则该列具有 REAL/Number 关联。
- 否则，关联为 NUMERIC。
- 如果表使用 CREATE TABLE t AS SELECT... 语句创建的，则所有列都未指定数据类型，将为它们分配关联 NONE。

数据类型与比较运算符

支持以下二元比较运算符 =、<、<=、>= 和 !=，以及测试集合成员身份的运算符 IN 和三元比较运算符 BETWEEN。有关这些运算符的详细信息，请参阅 " 运算符 "。

比较的结果取决于所比较的两个值的存储类。在比较两个值时，可应用以下规则：

- 将具有存储类 NULL 的值视为小于其他任何值 （包括具有存储类 NULL 的其他值）。
- INTEGER 或 REAL 值小于任何 TEXT 或 BLOB 值。对 INTEGER 或 REAL 与其他 INTEGER 或 REAL 进行比较时，将执行数字比较。
- TEXT 值小于 BLOB 值。对两个 TEXT 值进行比较时，将执行二元比较。
- 对两个 BLOB 值进行比较时，结果始终是使用二元比较确定的。

始终将三元运算符 BETWEEN 重新转换为等效的二元表达式。例如，a BETWEEN b AND c 将重新转换为 a >= b AND a <= c，即使这意味着在计算表达式所需的每个比较中将不同的关联应用于 a。

a IN (SELECT b ....) 类型的表达式由先前为二元比较列举的三个规则处理，即以与 a = b 类似的方式。例如，如果 b 是一个列值，a 是一个表达式，则在进行任何比较之前，将 b 的关联应用于 a。将表达式 a IN (x, y, z) 重新转换为 a = +x OR a = +y OR a = +z。IN 运算符右侧的值（此示例中为 x、y 和 z 值）被视为表达式，即使它们碰巧是列值。如果 IN 左侧的值是一个列，则使用该列的关联。如果该值是一个表达式，则不发生转换。

执行比较的方式也会受到使用 COLLATE 子句的影响。有关详细信息，请参阅 COLLATE。

数据类型与数学运算符

对于每个支持的数学运算符（*、/、%、+ 和 -），在计算表达式之前，会将数字关联应用于每个操作数。如果任一操作数无法成功转换为 NUMERIC 存储类，则表达式的计算结果将为 NULL。

如果使用了串联运算符 ||，则在计算表达式之前将每个操作数转换为 TEXT 存储类。如果任一操作数无法转换为 TEXT 存储类，则表达式的结果为 NULL。在两种情况下可能发生无法转换值的此问题：操作数的值为 NULL，或者它是包含非 TEXT 存储类的 BLOB。

数据类型与排序

通过 ORDER BY 子句对值排序时，具有存储类 NULL 的值排在最前面。它们后跟按数字顺序散布的 INTEGER 和 REAL 值，再后跟按二进制顺序或基于指定排序规则（BINARY 或 NOCASE）的 TEXT 值。最后是按二进制顺序的 BLOB 值。在排序之前，不发生存储类转换。

数据类型与分组

使用 GROUP BY 子句对值分组时，将具有不同存储类的值视为非重复。例外是 INTEGER 和 REAL 值，如果它们在数值上相等，则认为它们相等。 GROUP BY 子句不会导致将关联应用于任何值。

数据类型与复合 SELECT 语句

复合 SELECT 运算符 UNION、 INTERSECT 和 EXCEPT 执行值之间的隐式比较。在执行这些比较之前，可能会将关联应用于每个值。相同的关联 （如果有）将应用于在复合 SELECT 结果集的单个列中可能返回的所有值。所应用的关联是在该位置中具有列值 （而不是其他某种表达式）的第一个构成性 SELECT 语句所返回的列的关联。如果对于给定的复合 SELECT 列，构成性 SELECT 语句都不返回列值，则在对该列中的值进行比较之前，不将关联应用于这些值。

第 27 章 : SQL 错误详细消息、 ID 和参数

SQLError 类表示在使用 Adobe AIR 本地 SQL 数据库时可能出现的各种错误。对于任何给定的异常，SQLError 实例都具有一个包含中文错误消息的 details 属性。此外，每条错误消息都具有关联的唯一标识符，该标识符在 SQLError 对象的 detailID 属性中提供。使用 detailID 属性，应用程序可标识特定的 details 错误消息。应用程序可以为最终用户提供用其所在地区的语言表示的替换文本。可在错误消息字符串的合适位置替换 detailArguments 数组的参数值。这对于要直接向使用特定区域设置的最终用户显示错误的 details 属性错误消息的应用程序非常有用。

下表列出了 detailID 值和关联的中文错误消息文本。消息中的占位符文本指示在运行时替换 detailArguments 值的位置。此列表可用作对在 SQL 数据库操作中可能出现的错误消息进行本地化的源。

SQLError detailID	中文错误详细消息和参数
1001	连接已关闭。
1102	必须打开数据库才能执行此操作。
1003	在参数属性中找到了 %s [,| 和 %s] 参数名称，但在指定的 SQL 中未找到。
1004	参数计数不匹配。在指定的 SQL 中找到了 %d 个值，但在参数属性中设置了 %d 个值。应有 %s [,| 和 %s] 的值。
1005	无法打开自动压缩。
1006	无法设置 pageSize 值。
1007	未找到名为 “%s”、类型为 “%s” 的架构对象 （在数据库 “%s” 中）。
1008	未找到名为 “%s” 的架构对象 （在数据库 “%s” 中）。
1009	未找到类型为 “%s” 的架构对象 （在数据库 “%s” 中）。
1010	在数据库 “%s” 中未找到架构对象。
2001	分析器堆栈溢出
2002	函数 “%s” 上的参数过多
2003	“%s” 附近：语法错误
2004	已有另一个表或索引采用此名称：“%s”
2005	在 SQL 中不允许 PRAGMA。
2006	不是可写入目录。
2007	联接类型未知或不受支持：“%s %s %s”
2008	当前不支持 RIGHT OUTER JOIN 和 FULL OUTER JOIN。
2009	NATURAL 联接可能没有 ON 或 USING 子句。
2010	在同一联接中不能同时具有 ON 和 USING 子句。
2011	无法使用列 “%s” 进行联接 – 并非两个表中都有此列。
2012	作为表达式一部分的 SELECT 仅允许单个结果。
2013	没有这样的表：“[%s.]%s”
2014	未指定表。
2015	结果集中的列过多 |“%s” 上的列过多。
2016	%s ORDER|GROUP BY 项数超出范围 – 应介于 1 和 %d 之间
2017	ORDER BY 子句中的项过多。
2018	%s ORDER BY 项数超出范围 – 应介于 1 和 %d 之间。
2019	%r ORDER BY 项与结果集中的任何列都不匹配。
2020	ORDER BY 子句应在 “%s” 之后而不是之前。
2021	LIMIT 子句应在 “%s” 之后而不是之前。
2022	“%s” 左右 SELECT 的结果列数量不同。
2023	HAVING 前需要有 GROUP BY 子句。
2024	在 GROUP BY 子句中不允许聚合函数。
2025	聚合中的 DISTINCT 必须后跟一个表达式。
2026	复合 SELECT 中的项过多。
2027	ORDER|GROUP BY 子句中的项过多
2028	临时触发器可能没有限定名
2030	触发器 “%s” 已存在
2032	无法在视图上创建 BEFORE|AFTER 触发器：“%s”。

2033 无法在表上创建 INSTEAD OF 触发器：“%s”。
2034 没有这样的触发器：“%s”
2035 不支持递归触发器 （“%s”）。
2036 没有这样的列：%s[。 %s[.%s]]
2037 SQL 不允许 VACUUM。
2043 表 “%s”：索引函数返回的计划无效。
2044 一个联接中最多只能有 %d 个表。
2046 无法添加 PRIMARY KEY 列。
2047 无法添加 UNIQUE 列。
2048 无法添加具有默认值 NULL 的 NOT NULL 列。
2049 无法添加具有非常量默认值的列。
2050 无法向视图添加列。
2051 SQL 不允许 ANALYZE。
2052 名称无效：“%s”
2053 SQL 不允许 ATTACH。
2054 %s“%s” 不能引用数据库 “%s” 中的对象
2055 禁止访问 “[%s.]%s.%s”。
2056 未经授权。
2058 没有这样的视图：“[%s.]%s”
2060 临时表的名称必须是非限定的。
2061 表 “%s” 已存在。
2062 已存在此名称的索引：“%s”
2064 列名重复：“%s”
2065 表 “%s” 有多个主键。
2066 仅在 INTEGER PRIMARY KEY 上允许 AUTOINCREMENT
2067 没有这样的排序规则序列：“%s”
2068 视图中不允许有参数。
2069 视图 “%s” 是循环定义的。
2070 不能删除表 “%s”。
2071 使用 DROP VIEW 删除视图 “%s”
2072 使用 DROP TABLE 删除表 “%s”
2073 “%s” 上的外键应该仅引用表 “%s” 的一列。
2074 外键中的列数与引用表中的列数不匹配。
2075 外键定义中的列 “%s” 未知。
2076 不能编制表 “%s” 的索引。
2077 不能编制视图的索引。
2080 指定的 ON CONFLICT 子句相冲突。
2081 没有这样的索引：“%s”
2082 不能删除与 UNIQUE 或 PRIMARY KEY 约束关联的索引。
2083 SQL 不允许 BEGIN。
2084 SQL 不允许 COMMIT。
2085 SQL 不允许 ROLLBACK。
2086 无法打开临时数据库文件以存储临时表。
2087 无法标识要重新编制索引的对象。
2088 不能修改表 “%s”。
2089 “%s” 是视图，无法进行修改。
2090 变量编号必须介于 ?0 和 ?%d< 之间
2092 误用了别名聚合 “%s”
2093 列名不明确：“[%s.[%s.]]%s”
2094 没有这样的函数：“%s”
2095 函数 “%s” 的参数数量不正确
2096 在 CHECK 约束中禁止使用子查询。
2097 在 CHECK 约束中禁止使用参数。
2098 表达式树过大 （最大深度为 %d）
2099 RAISE() 只能在触发器程序中使用
2100 表 “%s” 具有 %d 个列，但提供了 %d 个值
2101 数据库架构已锁定：“%s”

2102	语句过长。
2103	由于存在活动的语句，无法删除 / 修改排序规则序列
2104	附加的数据库过多 — 最多 %d 个
2105	无法在事务内附加数据库。
2106	数据库 “%s” 已在使用中。
2108	附加的数据库必须使用与主数据库相同的文本编码。
2200	内存不足。
2201	无法打开数据库。
2202	无法在事务内分离数据库。
2203	无法分离数据库：“%s”
2204	数据库 “%s” 已锁定。
2205	无法获取数据库上的读取锁定。
2206	[列 | 列]“%s”[,“%s”] 不是 [唯一的 | 不是] 唯一的。
2207	数据库架构格式不正确。
2208	不支持的文件格式。
2209	标记无法识别：“%s”
2300	无法将文本值转换为数值。
2301	无法将字符串值转换为日期。
2302	无法将浮点值转换为整数而不丢失数据。
2303	无法回滚事务 - SQL 语句正在执行。
2304	无法提交事务 - SQL 语句正在执行。
2305	数据库表已锁定：“%s”
2306	只读表。
2307	字符串或 blob 过大。
2309	无法打开索引列以写入。
2400	无法打开类型为 %s 的值。
2401	没有这样的 rowid：%s
2402	此对象名称被保留供内部使用：“%s”
2403	不能更改视图 “%s”。
2404	列 “%s” 的默认值并不固定。
2405	无权使用函数 “%s”
2406	误用了聚合函数 “%s”
2407	误用了聚合：“%s”
2408	没有这样的数据库：“%s”
2409	表 “%s” 没有名为 “%s” 的列
2501	没有这样的模块：“%s”
2508	没有这样的保存点：“%s”
2510	无法回滚 - 没有事务处于活动状态。
2511	无法提交 - 没有事务处于活动状态。