学习使用
ACTIONSCRIPT® 3.0

法律声明

上次更新 2011/5/17

目录

第 1 章 : ActionScript 3.0 简介

关于 ActionScript

ActionScript 是 Adobe® Flash® Player 和 Adobe® AIR™ 运行时环境的编程语言。它在 Flash、 Flex 和 AIR 内容和应用程序中实现交互性、数据处理以及其他许多功能。

ActionScript 在 ActionScript 虚拟机 (AVM) 中执行，后者包含在 Flash Player 和 AIR 中。 ActionScript 代码通常由编译器转换为字节代码格式。字节代码是一种由计算机编写和识别的编程语言。编译器示例有：内置到 Adobe® Flash® Professional 中的编译器、内置到 Adobe® Flash® Builder™ 中的编译器，以及 Adobe® Flex™ SDK 中提供的编译器。字节代码嵌入在 Flash Player 和 AIR 执行的 SWF 文件中。

ActionScript 3.0 提供了可靠的编程模型，具备面向对象编程基本知识的开发人员都熟悉此模型。 ActionScript 3.0 相对于早期 ActionScript 版本改进的一些重要功能包括：

- 一个新增的 ActionScript 虚拟机，称为 AVM2，它使用全新的字节代码指令集，可使性能显著提高
- 一个更为先进的编译器代码库，可执行比早期编译器版本更深入的优化
- 一个扩展并改进的应用程序编程接口 (API)，拥有对对象的低级控制和真正意义上的面向对象的模型
- 一个基于 ECMAScript for XML (E4X) 规范 （ECMA-357 第 2 版）的 XML API。 E4X 是 ECMAScript 的一种语言扩展，它将 XML 添加为语言的本机数据类型。
- 一个基于文档对象模型 (DOM) 第 3 级事件规范的事件模型

ActionScript 3.0 的优点

ActionScript 3.0 的脚本编写功能优于 ActionScript 的早期版本。它旨在方便创建拥有大型数据集和面向对象的可重用代码库的高度复杂应用程序。在 Flash Player 中运行的内容不要求使用 ActionScript 3.0。但是，拥有它可以得到只能通过 AVM2（ActionScript 3.0 虚拟机）实现的性能改善。与旧的 ActionScript 代码相比， ActionScript 3.0 代码的执行速度可以快 10 倍。

早期版本的 ActionScript 虚拟机 AVM1 执行 ActionScript 1.0 和 ActionScript 2.0 代码。Flash Player 9 和 10 支持 AVM1 以实现后向兼容性。

ActionScript 3.0 中的新增功能

ActionScript 3.0 包含许多类似于 ActionScript 1.0 和 2.0 的类和功能。但是， ActionScript 3.0 在架构和概念上与早期的 ActionScript 版本不同。 ActionScript 3.0 中的改进包括新增的核心语言功能，以及能够更好地控制低级对象的改进 API。

核心语言功能

核心语言定义编程语言的基本构造块，例如语句、表达式、条件、循环和类型。 ActionScript 3.0 包含许多加快开发过程的功能。

运行时异常

ActionScript 3.0 报告的错误情形比早期的 ActionScript 版本多。运行时异常用于常见的错误情形，可改善调试体验并使您能够开发可以可靠地处理错误的应用程序。运行时错误可提供带有源文件和行号信息注释的堆栈跟踪，以帮助您快速定位错误。

运行时类型

在 ActionScript 3.0 中，类型信息在运行时保留。这些信息用于执行运行时类型检查，以改善系统的类型安全性。类型信息还可用于以本机形式表示变量，这样提高了性能，减少了内存使用量。经过比较，在 ActionScript 2.0 中，类型批注主要是一个开发人员辅助手段，所有值都在运行时以动态方式键入。

密封类

ActionScript 3.0 中引入了密封类的概念。密封类只能拥有在编译时定义的一组固定的属性和方法；不能添加其他属性和方法。由于不能在运行时更改类，使得编译时检查更严格，从而开发的程序更可靠。由于不要求每个对象实例都有一个内部哈希表，因此还提高了内存的使用率。还可以通过使用 dynamic 关键字来实现动态类。默认情况下， ActionScript 3.0 中的所有类都是密封的，但可以使用 dynamic 关键字将其声明为动态类。

闭包方法

ActionScript 3.0 使闭包方法可以自动记起它的原始对象实例。此功能对于事件处理非常有用。在 ActionScript 2.0 中，闭包方法无法记起它是从哪个对象实例提取的，所以调用闭包方法时会导致意外的行为。

ECMAScript for XML (E4X)

ActionScript 3.0 实现了 ECMAScript for XML (E4X)，后者最近被标准化为 ECMA-357。 E4X 提供一组用于操作 XML 的自然流畅的语言构造。与传统的 XML 分析 API 不同，使用 E4X 的 XML 就像该语言的本机数据类型一样执行。 E4X 通过大大减少所需代码的数量来简化操作 XML 的应用程序的开发。

若要查看 ECMA E4X 规范，请访问 www.ecma-international.org。

正则表达式

ActionScript 3.0 包括对正则表达式的固有支持，因此您可以快速搜索并操作字符串。由于在 ECMAScript (ECMA-262) 第 3 版语言规范中对正则表达式进行了定义，因此 ActionScript 3.0 实现了对正则表达式的支持。

命名空间

命名空间与用于控制声明 （public、 private、 protected）的可见性的传统访问说明符类似。它们的工作方式与名称由您指定的自定义访问说明符类似。命名空间使用统一资源标识符 (URI) 以避免冲突，而且在您使用 E4X 时还用于表示 XML 命名空间。

新基元类型

ActionScript 3.0 包含三种数值类型: Number、int 和 uint。Number 表示双精度浮点数。int 类型是一个带符号的 32 位整数，它使 ActionScript 代码可充分利用 CPU 的快速处理整数数学运算的能力。 int 类型对使用整数的循环计数器和变量都非常有用。 uint 类型是无符号的 32 位整数类型，可用于 RGB 颜色值、字节计数和其他方面。而 ActionScript 2.0 只包含 Number 一种数值类型。

API 功能

ActionScript 3.0 中的 API 包含许多可用于在低级别控制对象的类。语言体系结构的设计比早期版本更为直观。虽然有太多的类需要详细介绍，但是一些重要的区别更值得注意。

DOM3 事件模型

文档对象模型级别 3 事件模型 (DOM3) 提供了一种生成和处理事件消息的标准方式。这种事件模型的设计允许应用程序中的对象进行交互和通信、维持其状态以及响应更改。 ActionScript 3.0 事件模型的模式遵守万维网联合会 DOM 级别 3 事件规范。这种模型提供的机制比早期版本的 ActionScript 中提供的时间系统更清楚、更有效。

事件和错误事件都位于 flash.events 包中。 Flash Professional 组件和 Flex 框架使用的事件模型相同，因此整个 Flash Platform 中的事件系统是统一的。

显示列表 API

用于访问显示列表 （包含应用程序中所有可视元素的树）的 API 由使用可视基元的类组成。

Sprite 类是一个轻型构建基块，被设计为可视元素（如用户界面组件）的基类。Shape 类表示原始的矢量形状。可以使用 new 运算符很自然地实例化这些类，并可以随时动态重新指定其父类。

深度管理是自动进行的。提供了用于指定和管理对象的堆叠顺序的方法。

处理动态数据和内容

ActionScript 3.0 包含用于加载和处理 应用程序中的资源和数据的机制，这些机制在 API 中是直观的并且是一致的。Loader 类提供了一种加载 SWF 文件和图像资源的单一机制，并提供了一种访问已加载内容的详细信息的方式。 URLLoader 类提供了一种单独的机制，用于在数据驱动的应用程序中加载文本和二进制数据。 Socket 类提供了一种以任意格式从 / 向服务器套接字中读取 / 写入二进制数据的方式。

低级数据访问

多种 API 都提供对数据的低级访问。对于正在下载的数据而言，可使用 URLStream 类在下载数据的同时访问原始二进制数据。使用 ByteArray 类可优化二进制数据的读取、写入以及使用。使用 Sound API，可以通过 SoundChannel 类和 SoundMixer 类对声音进行精细控制。安全性 API 提供有关 SWF 文件或加载内容的安全权限的信息，使您能够更好地处理安全错误。

使用文本

ActionScript 3.0 包含一个用于所有与文本相关的 API 的 flash.text 包。TextLineMetrics 类为文本字段中的一行文本提供精确度量；该类取代了 ActionScript 2.0 中的 TextFormat.getTextExtent() 方法。TextField 类包含可以提供有关文本字段中一行文本或单个字符的特定信息的低级别方法。例如， getCharBoundaries() 方法返回一个表示字符边框的矩形。 getCharIndexAtPoint() 方法返回位于指定点的字符的索引。 getFirstCharInParagraph() 方法返回段落中第一个字符的索引。行级方法包括 getLineLength() （返回指定文本行中的字符数）和 getLineText() （返回指定行的文本）。 Font 类提供了一种管理 SWF 文件中的嵌入字体的方法。

为了对文本进行更低级别的控制， flash.text.engine 包中的类组成了 Flash 文本引擎。这组类提供对文本的低级控制，是针对创建文本框架和组件而设计的。

第 2 章 : ActionScript 快速入门

编程基础

ActionScript 是一种编程语言，因此，如果您先弄懂一些常规计算机编程概念，对您学习 ActionScript 会很有帮助。

计算机程序的用途

首先，对计算机程序的概念及其用途有一个概念性的认识是非常有用的。计算机程序主要包括两个方面：

- 程序是计算机执行的一系列指令或步骤。
- 每一步最终都涉及到对某一段信息或数据的处理。

通常认为，计算机程序只是您提供给计算机并让它逐步执行的指令列表。每个单独的指令都称为语句。在 ActionScript 中，每个语句都以分号结尾。

实质上，程序中指令所做的全部事情是操作存储在计算机内存中的一些数据位。举个简单的例子：指示计算机将两个数字相加，并将结果存储在内存中。再举个比较复杂的例子：假设在屏幕上绘制了一个矩形，您希望写个程序，将此矩形移动到屏幕上的其他位置。计算机会记住有关此矩形的某些信息：所处位置的 x、y 坐标，高度和宽度，颜色等等。这些信息位中的每一位都存储在计算机内存中的某个位置。程序要将此矩形移动到其他位置，可能会包含如下步骤“将 X 坐标更改为 200；将 Y 坐标更改为 150”。换句话说，程序将指定新的 X 和 Y 坐标值。计算机在后台根据此数据进行处理，将这些值实际应用到计算机屏幕上显示的图像上。不过，只要基本了解“移动屏幕上的矩形”这个过程仅涉及到更改计算机内存中的数据位就够了。

变量和常量

编程主要涉及更改计算机内存中的信息。因此，要有一种表示单条信息的方式，这在程序中很重要。变量 是一个名称，表示计算机内存中的值。当编写语句来处理操作值时，写入变量名来代替值。计算机在查看程序中的变量名时，都将查看内存并使用在内存中找到的值。例如，如果两个名为 value1 和 value2 的变量分别包含一个数字，则可以编写如下语句将这两个数字相加：

```
value1 + value2
```

当实际执行这些步骤时，计算机将查看每个变量中的值并将它们相加。

在 ActionScript 3.0 中，一个变量实际上包含三个不同部分：

- 变量的名称
- 可以存储在变量中的数据的类型
- 存储在计算机内存中的实际值

您已了解计算机如何使用名称作为值的占位符。数据类型也非常重要。当您在 ActionScript 中创建变量时，请指定此变量打算支持的特定数据类型。此后，程序的指令在此变量中仅可存储该类型的数据。您可以使用与值的数据类型相关联的特定特性处理值。在 ActionScript 中，若要创建一个变量 （称为声明 变量），应使用 var 语句：

```
var value1:Number;
```

此示例要求计算机创建名为 value1 的变量，此变量仅接受支持 Number 数据。（“Number” 是 ActionScript 中定义的特定数据类型。）您还可以立即在变量中存储一个值：

```
var value2:Number = 17;
```

Adobe Flash Professional

在 Flash Professional 中，还有另外一种变量声明方法。在将一个影片剪辑元件、按钮元件或文本字段放置在舞台上时，可以在属性检查器中为它指定一个实例名称。Flash Professional 在后台创建与实例同名的变量。可在 ActionScript 代码中使用该名称表示该舞台项。例如，假设舞台上有一个影片剪辑元件，并为其指定了实例名称 rocketShip。只要您在 ActionScript 代码中使用变量 rocketShip，实际上就是在操作该影片剪辑。

常量类似于变量。它是使用指定的数据类型表示计算机内存中的值的名称。不同之处在于，在 ActionScript 应用程序运行期间只能为常量赋值一次。一旦为某个常量赋值之后，该常量的值在整个应用程序运行期间都保持不变。声明常量的语法与声明变量的语法几乎相同。唯一的不同之处在于，需要使用关键字 const，而不是关键字 var：

```
const SALES_TAX_RATE:Number = 0.07;
```

如需定义在整个项目中多个位置使用且正常情况下不会更改的值，则常量非常有用。使用常量而不使用字面值能让代码更加便于理解。例如，我们看一下同一代码的两种版本。一个用 SALES_TAX_RATE 与价格相乘。另一个则用 0.07 与价格相乘。使用 SALES_TAX_RATE 常量的版本较易理解。另外，假设用常量定义的值确实需要更改。如果您使用常量在整个项目中表示特定值，可以在一处位置更改此值 （常量声明）。相反，如果您使用硬编码的字面值，则必须在各个位置更改此值。

数据类型

在 ActionScript 中，您可以将很多数据类型用作所创建的变量的数据类型。某些数据类型可以看作 “ 简单 ” 或 “ 基础 ” 数据类型：

- String：文本值，例如，一个名称或书中某一章的文字
- Numeric：对于 numeric 型数据， ActionScript 3.0 包含三种特定的数据类型：
  - Number：任何数值，包括有小数部分或没有小数部分的值
  - Int：一个整数 （不带小数部分的整数）
  - Uint：一个 “ 无符号 ” 整数，即不能为负数的整数
- Boolean：一个 true 或 false 值，例如开关是否开启或两个值是否相等

简单数据类型表示单条信息：例如，单个数字或单个文本序列。不过， ActionScript 中定义的大多数数据类型可能是复杂数据类型。它们表示单一容器中的一组值。例如，数据类型为 Date 的变量表示单一值 （某个时刻）。然而，该日期值以多个值表示：天、月、年、小时、分钟、秒，等等，这些值都为单独的数字。人们一般认为日期为单一值，您可以通过创建 Date 变量将日期视为单一值。不过，在计算机内部，计算机认为它是共同定义一个日期的一组值。

大部分内置数据类型以及程序员定义的数据类型都是复杂数据类型。您可能知道下面的一些复杂数据类型：

- MovieClip：影片剪辑元件
- TextField：动态文本字段或输入文本字段
- SimpleButton：按钮元件
- Date：有关时间中的某个片刻的信息 （日期和时间）

经常用作数据类型的同义词的两个词是类和对象。类只是数据类型的定义。它像一个适用于某数据类型的所有对象的模板，就好像说 “ 示例数据类型的所有变量都具有以下特性：A、 B 和 C”。另一方面，对象只是类的实际实例。例如，数据类型为 MovieClip 的变量可以被描述为 MovieClip 对象。下面几条陈述虽然表达的方式不同，但意思是相同的：

- 变量 myVariable 的数据类型是 Number。
- 变量 myVariable 是一个 Number 实例。
- 变量 myVariable 是一个 Number 对象。
- 变量 myVariable 是 Number 类的一个实例。

使用对象

ActionScript 是一种面向对象的编程语言。面向对象的编程只是一种编程方法。组织程序中代码的方法确实只有一种，即使用对象。

术语“计算机程序”早些时候定义为计算机执行的一系列步骤或指令。所以，从概念上讲，您可以认为计算机程序就是一个很长的指令列表。然而，在面向对象的编程中，程序指令分布在不同对象中。代码被编组为功能区块，因此相关的功能类型或相关的各条信息被编组到一个容器中。

Adobe Flash Professional

如果您已在 Flash Professional 中使用过元件，就应该已习惯使用对象了。假定您定义了一个影片剪辑元件（例如矩形绘制），并已在舞台上放置了其副本。从严格意义上来说，该影片剪辑元件也是 ActionScript 中的一个对象；即 MovieClip 类的一个实例。

您可以修改该影片剪辑的不同特征。选择该影片剪辑元件后，您可以在属性检查器中更改值（如元件的 X 坐标或宽度）。您还可以进行各种颜色调整，例如更改其 Alpha（透明度）或对其应用投影滤镜。还可以使用其他 Flash Professional 工具进行更多更改，例如，使用自由变换工具旋转矩形。 Flash Professional 中可用于修改影片剪辑元件的所有这些方法，在 ActionScript 中也提供。通过更改合在一起组成称为 MovieClip 对象的单一绑定的各种数据，可以在 ActionScript 中修改影片剪辑。

在 ActionScript 面向对象的编程中，任何类都可以包含三种类型的特性：

- 属性
- 方法
- 事件

这些元素用于管理程序使用的各种数据并决定执行哪些操作及执行顺序。

属性

属性表示某个对象中绑定在一起的若干数据块中的一个。例如， song 对象可以包含名为 artist 和 title 的属性； MovieClip 类具有 rotation、 X、 width 和 alpha 等属性。您可以像使用各变量那样使用属性。事实上，您可以简单地将属性视为包含于对象中的“子”变量。

以下是一些使用属性的 ActionScript 代码的示例。以下代码行将名为 square 的 MovieClip 移动到 100 个像素的 x 坐标处：

```
square.x = 100;
```

此代码使用 rotation 属性旋转 square MovieClip，以便与 triangle MovieClip 的旋转相匹配：

```
square.rotation = triangle.rotation;
```

此代码改变了 square MovieClip 的水平缩放，使其宽度变为之前的 1.5 倍：

```
square.scaleX = 1.5;
```

请注意上面几个示例的通用结构：将变量（square 和 triangle）用作对象的名称，后跟一个句点 (.) 和属性名（x、 rotation 和 scaleX）。句点称为点运算符，用于指示您要访问对象的某个子元素。整个结构“变量名 - 点 - 属性名”的使用类似于单个变量，作为计算机内存中的单个值的名称。

方法

方法是对象可执行的动作。例如，假设您在 Flash Professional 中的时间轴上创建了带有多个关键帧和动画的影片剪辑元件。影片剪辑可以播放、停止或根据命令将播放头移动到特定帧。

下面的代码指示名为 shortFilm 的 MovieClip 开始播放：

```
shortFilm.play();
```

下面的代码行使名为 shortFilm 的 MovieClip 停止播放 （播放头停在原地，就像暂停播放视频一样）：

```
shortFilm.stop();
```

下面的代码使名为 shortFilm 的 MovieClip 将其播放头移到第 1 帧，然后停止播放 （就像后退视频一样）：

```
shortFilm.gotoAndStop(1);
```

您可以通过依次写下对象名 （变量）、句点、方法名和小括号来访问方法，这与属性类似。小括号用于指示要调用方法 （即指示对象执行该操作）。有时，为了传递执行动作所必需的额外信息，也将值 （或变量）放入小括号中。这些值称为方法参数。例如，gotoAndStop() 方法需要关于转到哪一帧的信息，所以要求小括号中有一个参数。有些方法 （如 play() 和 stop()）自身的意义已非常明确，因此不需要额外信息。但书写时仍然带有小括号。

与属性 （和变量）不同的是，方法不能用作值占位符。然而，一些方法可以执行计算并返回可以像变量一样使用的结果。例如， Number 类的 toString() 方法将数值转换为文本表示形式：

```
var numericData:Number = 9;
var textData:String = numericData.toString();
```

例如，如果希望在屏幕上的文本字段中显示 Number 变量的值，应使用 toString() 方法。 TextField 类的 text 属性定义为 String，因此只能包含文本值。（文本 property 表示屏幕上显示的实际文本内容。）此行代码将变量 numericData 中的数值转换为文本。然后，使该值显示在屏幕上名为 calculatorDisplay 的 TextField 对象中：

```
calculatorDisplay.text = numericData.toString();
```

事件

计算机程序就是计算机分步执行的一系列指令。一些简单的计算机程序仅包括计算机要执行的几个步骤以及程序的结束点。然而， ActionScript 程序可以保持运行、等待用户输入或等待其他事件发生。事件是确定计算机执行哪些指令以及何时执行的机制。

本质上，事件就是所发生的、 ActionScript 能够识别并可响应的事情。许多事件与用户交互相关联，诸如用户单击某个按钮或按键盘上的某个键之类。还有其他类型的事件。例如，如果使用 ActionScript 加载外部图像，有一个事件可让您知道图像何时加载完毕。当 ActionScript 程序运行时，从概念上讲，它只是坐等某些事情发生。发生这些事情时，为这些事件指定的特定 ActionScript 代码将运行。

基本事件处理

用于指定为响应特定事件而执行的特定操作的技术称为事件处理。在编写执行事件处理的 ActionScript 代码时，您需要识别三个重要元素：

- 事件源：发生该事件的是哪个对象？例如，单击了哪个按钮，或哪个 Loader 对象正在加载图像？事件源也称为事件目标。之所以有此名称，是因为它是计算机确定事件的目标位置的对象 （也就是说，实际发生事件的位置）。
- 事件：将要发生什么事情，以及您希望响应什么事情？识别特定事件非常重要，因为许多对象都会触发多个事件。
- 响应：当事件发生时，您希望执行哪些步骤？

无论何时编写 ActionScript 代码来处理事件，都要求使用这三个元素。代码遵循以下基本结构 （粗体元素是占位符，您要根据自己的具体情况填充）：

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

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

此代码完成两项任务。首先，定义一个函数，这是指定为响应事件而要执行的动作的方法。其次，调用源对象的 addEventListener() 方法。调用 addEventListener() 实质上是为指定事件“订阅”此函数。该事件发生时，将执行函数的动作。让我们更详细地讨论其中每个部分。

函数是提供了一种方法，可以将若干个操作用类似快捷名称的单个名称组合在一起，通过这个单个名称执行这些操作。除了不必与特定类关联之外，函数与方法完全相同。（事实上，术语“方法”可定义为与特定类相关联的函数。）在创建用于事件处理的函数时，请选择函数名称 （本示例中名为 eventResponse）。还要指定一个参数 （本示例中名为 eventObject）。指定函数参数类似于声明变量，所以还必须指明参数的数据类型。（在本示例中，该参数的数据类型为 EventType。）

您要侦听的每种事件类型都有一个与其相关联的 ActionScript 类。为函数参数指定的数据类型始终是与您要响应的特定事件关联的类。例如，click 事件 （在用户使用鼠标单击某个项目时触发）与 MouseEvent 类相关联。若要为 click 事件编写侦听器函数，可使用数据类型为 MouseEvent 的参数定义侦听器函数。最后，在左大括号与右大括号之间 ({ ... }) 编写希望计算机在事件发生时执行的指令。

事件处理函数已编写。下一步要告知事件源对象 （发生该事件的对象，如按钮）您希望其在该事件发生时调用此函数。通过调用该事件源对象的 addEventListener() 方法向该对象注册您的函数 （所有包含事件的对象也都具有 addEventListener() 方法）。addEventListener() 方法有两个参数：

- 第一个参数是您希望响应的特定事件的名称。每个事件都与一个特定类相关联。每个事件类都为其中每一个事件定义了一个特殊的值，就像唯一名称。对第一个参数使用该值。
- 第二个参数是事件响应函数的名称。请注意，如果将函数名称作为参数进行传递，则在写入函数名称时不使用括号。

事件处理过程

下面分步描述了创建事件侦听器时执行的过程。在本例中，您将创建一个侦听器函数，在单击名为 myButton 的对象时调用该函数。

程序员实际编写的代码如下所示：

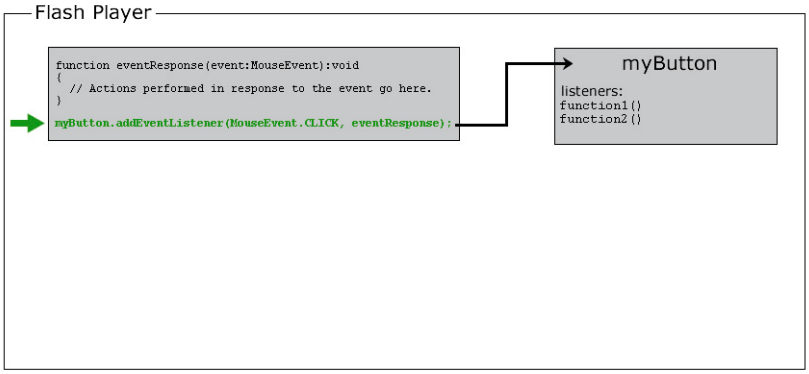
```
function eventResponse(event:MouseEvent):void
{
    // Actions performed in response to the event go here.
}

myButton.addEventListener(MouseEvent.CLICK, eventResponse);
```

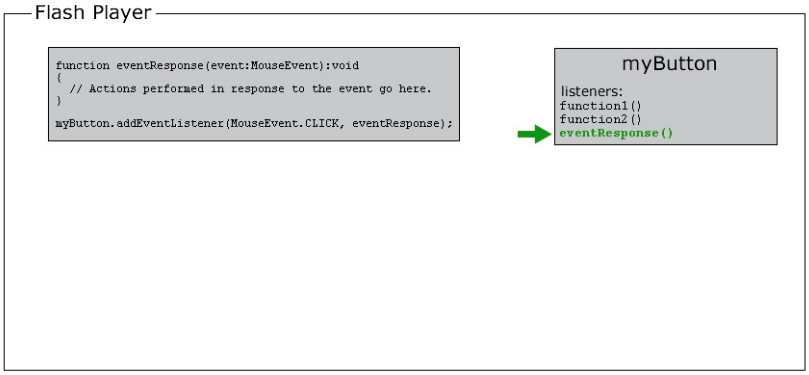
下面是此代码运行时的实际工作方式：

1 加载 SWF 文件时，计算机会注意到有一个名为 eventResponse() 的函数。

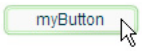
2 计算机随后运行该代码（具体地说，是指不在函数中的代码行）。在本例中，只有一行代码：针对事件源对象（名为 myButton）调用 addEventListener() 方法，并将 eventResponse 函数作为参数进行传递。



在内部，myButton 保留正在监听其各个事件的函数的列表。调用 addEventListener() 方法时，myButton 将 eventResponse() 函数存储到其事件侦听器列表中。

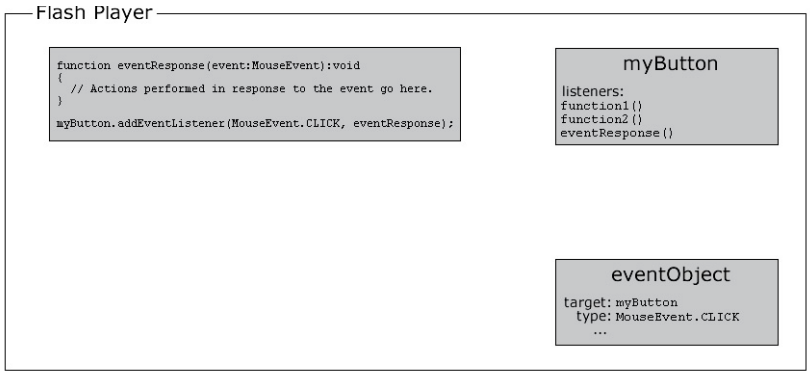


3 在某一时刻，用户单击 myButton 对象以触发其 click 事件（在代码中将其标识为 MouseEvent.CLICK）。

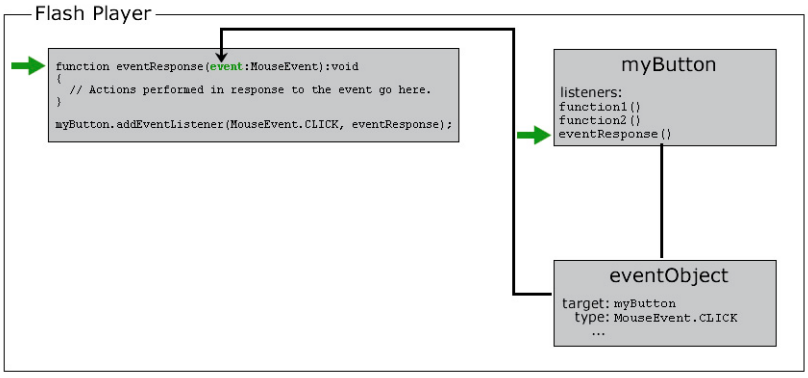


此时发生了以下事件：

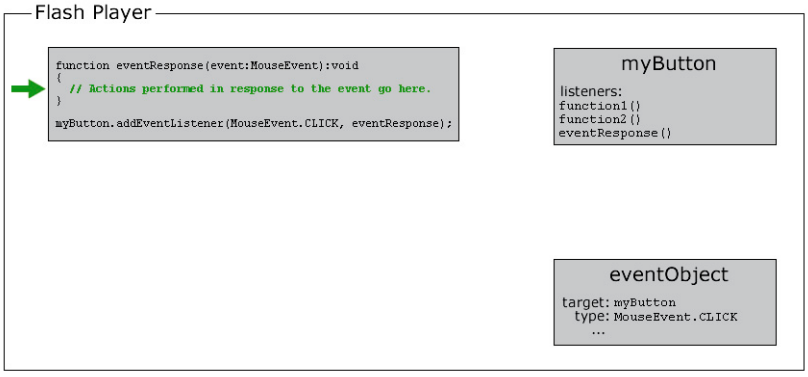
a 创建对象，该对象是与发生的事件（此示例中为 MouseEvent）相关联的类的实例。对很多事件而言，此对象是 Event 类的实例。对鼠标事件而言，它是 MouseEvent 实例。对其他事件而言，它是与该事件关联的类的实例。创建的此对象称为事件对象，包含有关所发生的事件的特定信息：事件类型、发生位置以及其他特定于事件的信息（如果适用）。



b 计算机随后查看 myButton 存储的事件侦听器的列表。它逐个查看这些函数，以调用每个函数并将事件对象作为参数传递给该函数。由于 eventResponse() 函数是 myButton 的侦听器之一，因此，计算机将在此过程中调用 eventResponse() 函数。



c 当调用 eventResponse() 函数时，将运行该函数中的代码，因此，将执行您指定的动作。



事件处理示例

以下是事件处理代码的更多实际示例。给出这些示例，旨在提供关于编写事件处理代码时可以使用的一些常见事件元素及可用变体的建议：

- 单击按钮开始播放当前的影片剪辑。在下面的示例中，playButton 是按钮的实例名称，而 this 是表示“当前对象”的特殊名称：

```
this.stop();

function playMovie(event:MouseEvent):void
{
    this.play();
}

playButton.addEventListener(MouseEvent.CLICK, playMovie);
```

- 检测文本字段中的键入操作。在下面的示例中，entryText 是一个输入文本字段，而 outputText 是一个动态文本字段：

```
function updateOutput(event:TextEvent):void
{
    var pressedKey:String = event.text;
    outputText.text = "You typed: " + pressedKey;
}

entryText.addEventListener(TextEvent.TEXT_INPUT, updateOutput);
```

- 单击按钮导航到一个 URL。在本例中，linkButton 是该按钮的实例名称：

```
function gotoAdobeSite(event:MouseEvent):void
{
    var adobeURL:URLRequest = new URLRequest("http://www.adobe.com/");
    navigateToURL(adobeURL);
}

linkButton.addEventListener(MouseEvent.CLICK, gotoAdobeSite);
```

创建对象实例

在 ActionScript 中使用某个对象之前，首先该对象必须存在。创建对象的步骤之一是声明变量；然而，声明变量仅仅是在计算机的内存中创建一个空位置。在尝试使用或操作变量之前，务必为变量指定实际值（即创建一个对象并将其存储在变量中）。创建对象的过程称为实例化对象。换句话说，是创建特定类的实例。

有一种创建对象实例的简单方法完全不涉及 ActionScript。在 Flash Professional 中，将一个影片剪辑元件、按钮元件或文本字段放置到舞台上并为其指定实例名称。 Flash Professional 将自动使用该实例名称声明变量、创建对象实例并将该对象存储到变量中。同样，在 Flex 中，您通过对 MXML 标签进行编码或将组件放置到 Flash Builder Design 模式的编辑器中，在 MXML 中创建组件。为该组件分配 ID 后，该 ID 就成为包含该组件实例的 ActionScript 变量的名称。

不过，您并非总是希望以可视方式创建对象，而又不能创建非可视对象。可以通过其他若干种方法仅使用 ActionScript 创建对象实例。

借助几个 ActionScript 数据类型，可以使用文本表达式（直接写入 ActionScript 代码的值）创建实例。下面给出了一些示例：

- 文本数字值（直接输入数字）：

```
var someNumber:Number = 17.239;
var someNegativeInteger:int = -53;
var someUint:uint = 22;
```

- 文本字符串值（用双引号将本文引起来）：

```
var firstName:String = "George";
var soliloquy:String = "To be or not to be, that is the question...";
```

- 文本布尔值 （使用字面值 true 或 false）：

```
var niceWeather:Boolean = true;
var playingOutside:Boolean = false;
```

- 文本数组值 （在中括号中包含以逗号分隔的值列表）：

```
var seasons:Array = ["spring", "summer", "autumn", "winter"];
```

- 文本 XML 值 （直接输入 XML）：

```
var employee:XML = <employee>
        <firstName>Harold</firstName>
        <lastName>Webster</lastName>
    </employee>;
```

ActionScript 还为 Array、 RegExp、 Object 和 Function 数据类型定义了文本表达式。

为任一数据类型创建实例的最常用方法是使用带有类名称的 new 运算符，如下所示：

```
var raceCar:MovieClip = new MovieClip();
var birthday:Date = new Date(2006, 7, 9);
```

使用 new 运算符创建对象通常称为 “ 调用类的构造函数 ”。构造函数 是在创建类的实例的过程中调用的一种特殊方法。请注意，在使用此方法创建实例时，在类名称后加上小括号。有时要在小括号中指定参数值。调用方法时还需要完成另外两项操作。

甚至对于可使用文本表达式创建实例的数据类型，也可以使用 new 运算符来创建对象实例。例如，下面的两行代码执行相同的操作：

```
var someNumber:Number = 6.33;
var someNumber:Number = new Number(6.33);
```

熟悉使用 new ClassName() 创建对象的方法是非常重要的。许多 ActionScript 数据类型没有直观的表示形式。因此，无法通过将项目放置到 Flash Professional 舞台或 Flash Builder 的 MXML 编辑器的设计模式来创建。您只能使用 new 运算符在 ActionScript 中创建这些数据类型的实例。

Adobe Flash Professional

在 Flash Professional 中， new 运算符还可用于创建已在库中定义、但没有放到舞台上的影片剪辑元件的实例。

更多帮助主题

使用数组

使用正则表达式

使用 ActionScript 创建 MovieClip 对象

常用编程元素

还有其他几个构造基块，您可以使用它们创建 ActionScript 程序。

运算符

“ 运算符 ” 是用于执行计算的特殊符号 （有时候是词）。这些运算符主要用于数学运算，有时也用于值的比较。通常，运算符使用一个或多个值并 “ 计算出 ” 一个结果。例如：

- 加法运算符 (+) 将两个值相加，结果是一个数字：

```
var sum:Number = 23 + 32;
```

- 乘法运算符 (*) 将一个值与另一个值相乘，结果是一个数字：

```
var energy:Number = mass * speedOfLight * speedOfLight;
```

- 等于运算符 (==) 比较两个值以确定是否相等，结果是一个 true 或 false （布尔）值：

```
if (dayOfWeek == "Wednesday")
{
    takeOutTrash();
}
```

如上所示，等于运算符和其他“比较”运算符最常用于 if 语句中，用来确定是否执行某些指令。

注释

编写 ActionScript 时，通常希望为自己留些注释。例如，有时希望说明某些代码行如何工作或做出特定选择的原因。代码注释是一个工具，用于编写计算机在代码中忽略的文本。 ActionScript 包括两种注释：

- 单行注释：在一行中的任意位置放置两个斜杠来指定单行注释。计算机会忽略斜线后面直到该行结尾的所有内容：

```
// This is a comment; it's ignored by the computer.
var age:Number = 10; // Set the age to 10 by default.
```

- 多行注释：多行注释包括一个开始注释标记 (/*)、注释内容和一个结束注释标记 (*/)。计算机会忽略开始和结束注释标记之间的所有内容，而不考虑注释有多少行：

```
/*
This is a long description explaining what a particular
function is used for or explaining a section of code.

In any case, the computer ignores these lines.
*/
```

注释还常用于暂时“关闭”一行或多行代码。例如，您可以在试验某项工作的其他处理方式时使用注释。您还可以使用注释来尝试了解为什么有些 ActionScript 代码未达到预期工作效果。

流控制

在程序中，经常需要重复某些动作，仅执行某些动作而不执行其他动作，或根据某些条件执行替代动作等等。“流控制”就是用于控制执行哪些动作。 ActionScript 中提供了几种类型的流控制元素。

- 函数：函数类似于快捷方式，它们将一系列操作组合到单个名称下，并可用于执行计算。处理事件需要用到函数，但函数也可用作组合一系列指令的通用工具。
- 循环：使用循环结构，可指定计算机反复执行一组指令，直到达到设定的次数或某些条件发生改变为止。通常借助循环并使用一个其值在计算机每执行完一次循环后就改变的变量来处理几个相关项。
- 条件语句：条件语句用来指定一些仅在特定情况下执行的指令。还可用来提供在其他情况下使用的几组备用指令。最常见的一类条件语句是 if 语句。 if 语句检查该语句括号中的值或表达式。如果值为 true，则执行大括号中的代码行。否则，忽略这些代码行。例如：

```
if (age < 20)
{
    // show special teenager-targeted content
}
```

同时使用 if 语句与 else 语句可以指定在条件不为 true 时计算机执行的替代指令：

```
if (username == "admin")
{
    // do some administrator-only things, like showing extra options
}
else
{
    // do some non-administrator things
}
```

示例：动画公文包片段 (Flash Professional)

该示例旨在让您在第一时机看到如何将各段 ActionScript 合并为一个完整的应用程序。该动画包片段是一个有关如何利用现有线性动画以及添加某些次要交互式元素的示例。例如，您可以将为某客户创建的动画纳入联机包。要添加到动画中的交互行为包括两个查看者可以单击的按钮：一个用于启动动画，另一个用于导航到单独的 URL （例如包菜单或作者的主页）。

创建该片段的过程可以分为四个主要部分：

1 准备 FLA 文件以便添加 ActionScript 和交互式元素。

2 创建和添加按钮。

3 编写 ActionScript 代码。

4 测试应用程序。

准备添加交互

向动画添加交互式元素之前，创建一些用于添加新内容的位置将有助于创建 FLA 文件。此任务包括在舞台上创建可放置按钮的实际空间。此外，还包括在 FLA 文件中创建 “ 空间 ”，以分隔不同的项目。

要创建 **FLA** 以添加交互式元素，请执行下列操作：

1 创建一个包含简单动画 （如一个补间动画或补间形状）的 FLA 文件。如果已经创建了一个包含项目中正在展示的动画的 FLA 文件，则打开该文件，以新名称保存。

2 确定两个按钮在屏幕上的显示位置。一个按钮用于启动动画，另一个用于链接到作者的包或主页。如果需要，在舞台上为该新内容清除或添加一些空间。如果该动画还没有启动按钮，可以在第一个帧上创建一个启动屏幕。这时，您可能希望移动动画，使其从第二帧或后面的帧启动。

3 在时间轴中的其他图层上添加一个新图层，并将其命名为 **buttons**。这是将要在其中添加按钮的图层。

4 在 buttons 图层之上创建一个新图层，并将其命名为 **actions**。这是将要在其中向应用程序添加 ActionScript 代码的图层。

创建和添加按钮

接下来，将实际创建和放置构成交互式应用程序中心的按钮。

要创建按钮并将其添加到 **FLA**，请执行下列操作：

1 使用绘图工具在 buttons 图层上创建第一个按钮 （“play” 按钮）的可视外观，例如，绘制一个顶部带文本的水平椭圆。

2 使用 “ 选择 ” 工具选择单个按钮的所有图形部分。

3 在主菜单中，选择 “ 修改 ”>“ 转换为元件 ”。

4 在对话框中，选择 “ 按钮 ” 作为元件类型，为该元件赋予一个名称，然后单击 “ 确定 ”。

5 选择按钮之后，在 “ 属性 ” 检查器中为按钮赋予实例名称 **playButton**。

6 重复步骤 1 至 5，创建可将查看者带到作者主页的按钮。将该按钮命名为 **homeButton**。

编写代码

虽然此应用程序的 ActionScript 代码全都是在同一个位置输入的，但是该代码可分为三组功能。代码的这三组功能分别是：

- 一旦 SWF 文件开始加载 （当播放头进入第 1 帧时），就停止播放头。
- 侦听一个事件，该事件在用户单击播放按钮时开始播放 SWF 文件。
- 侦听一个事件，该事件在用户单击创作者主页按钮时将浏览器定向至相应的 URL。

要创建代码，使得在播放头进入第 **1** 帧时停止播放头，请执行下列操作：

1. 在 actions 图层的第 1 帧上选择关键帧。
2. 若要打开“动作”面板，请从主菜单中选择“窗口”>“动作”。
3. 在“脚本”窗格中，输入以下代码：

```
stop();
```

要编写代码，使得单击播放按钮时启动动画，请执行下列操作：

1. 在前面步骤中输入的代码的末尾添加两个空行。
2. 在脚本底部输入以下代码：

```
function startMovie(event:MouseEvent):void
{
    this.play();
}
```

此代码定义一个名为 startMovie() 的函数。调用 startMovie() 时，该函数会导致主时间轴开始播放。

3. 在上一步中添加的代码的下一行中，输入以下代码行：

```
playButton.addEventListener(MouseEvent.CLICK, startMovie);
```

此代码行将 startMovie() 函数注册为 playButton 的 click 事件的侦听器。也就是说，它使得只要单击名为 playButton 的按钮，就会调用 startMovie() 函数。

要编写代码，使得单击主页按钮时将浏览器定向至某一个 **URL**，请执行下列操作：

1. 在前面步骤中输入的代码的末尾添加两个空行。
2. 在脚本底部输入以下代码：

```
function gotoAuthorPage(event:MouseEvent):void
{
    var targetURL:URLRequest = new URLRequest("http://example.com/");
    navigateToURL(targetURL);
}
```

此代码定义一个名为 gotoAuthorPage() 的函数。此函数首先创建一个表示 URL http://example.com/ 的 URLRequest 实例。然后将该 URL 传递给 navigateToURL() 函数，使用户的浏览器打开此 URL。

3. 在上一步中添加的代码的下一行中，输入以下代码行：

```
homeButton.addEventListener(MouseEvent.CLICK, gotoAuthorPage);
```

此代码行将 gotoAuthorPage() 函数注册为 homeButton 的 click 事件的侦听器。也就是说，它使得只要单击名为 homeButton 的按钮，就会调用 gotoAuthorPage() 函数。

测试应用程序

应用程序的功能现已齐备。让我们测试一下应用程序，看看到底是不是这样。

测试应用程序：

1 从主菜单中，选择“控制”>“测试影片”。Flash Professional 将创建 SWF 文件，并在 Flash Player 窗口中打开该文件。

2 试用这两个按钮，确保它们按您预期的方式工作。

3 如果按钮不起作用，请检查下列事项：

- 这两个按钮是否具有不同的实例名？
- addEventListener() 方法调用使用的名称是否与按钮的实例名相同？
- addEventListener() 方法调用中使用的事件名称是否正确？
- 为各个函数指定的参数是否正确？（两种方法都需要一个数据类型为 MouseEvent 的参数。）

上述所有错误和大多数其他错误都会导致出现错误消息。选择“测试影片”命令时，或测试项目时单击此按钮，都会出现该错误消息。在“编译器错误”面板中查看编译器错误（第一次选择“测试影片”时出现的错误）。检查“输出”面板查看是否有在播放内容（如单击按钮）时发生的运行时错误。

使用 ActionScript 构建应用程序

编写 ActionScript 来构建应用程序的过程不仅仅需要了解使用的语法和类名称。大多数 Flash Platform 文档都涉及这两个主题（语法和使用 ActionScript 类）。然而，要构建 ActionScript 应用程序，您还需要了解以下信息：

- 可以使用哪些程序编写 ActionScript？
- 如何组织 ActionScript 代码？
- 如何在应用程序中加入 ActionScript 代码？
- 开发 ActionScript 应用程序需要按哪些步骤进行？

用于组织代码的选项

您可以使用 ActionScript 3.0 代码来实现任何目的，从简单的图形动画到复杂的客户端 - 服务器事务处理系统都可以通过它来实现。根据您要构建的应用程序的类型，使用其中一种或多种不同方式在项目中加入 ActionScript。

将代码存储在 **Flash Professional** 时间轴的帧中

在 Flash Professional 中，可以向时间轴中的任何帧添加 ActionScript 代码。该代码在影片播放期间播放头进入该帧时执行。

通过向帧中添加 ActionScript 代码，可以方便地向使用 Flash Professional 构建的应用程序添加行为。您可以将代码添加到主时间轴中的任何帧，或任何 MovieClip 元件的时间轴中的任何帧。但是，这种灵活性也有一定的代价。构建较大的应用程序时，这会容易导致无法跟踪哪些帧包含哪些脚本。这种复杂的结构会使应用程序随着时间的推移越来越难以维护。

许多开发人员将代码只放在时间轴的第一帧中，或放在 Flash 文档中的特定图层上，以简化在 Flash Professional 中组织其 ActionScript 代码的工作。将代码分隔开可以比较容易地在 Flash FLA 文件中查找和维护代码。然而，不能在另一个 Flash Professional 项目中使用同一代码，除非将代码复制粘贴到该新文件中。

为了将来更容易在其他 Flash Professional 项目中使用 ActionScript 代码，请将代码存储在外部 ActionScript 文件（扩展名为 .as extension 的文本文件）中。

在 **Flex MXML** 文件中嵌入代码

在 Flex 开发环境（如 Flash Builder）中，可以在 Flex MXML 文件的 <fx:Script> 标签中添加 ActionScript 代码。然而，这种方法会增加大项目的复杂性，使在另一个 Flex 项目中使用同一代码更困难。为了将来更容易在其他 Flex 项目中使用 ActionScript 代码，请将代码存储在外部 ActionScript 文件中。

注：可以为 <fx:Script> 标签指定一个源参数。通过使用源参数，您可以从外部文件中“导入”ActionScript 代码，就好像直接在 <fx:Script> 标签中键入该代码一样。但是，您使用的源文件不能定义自己的类，从而限制了其可重用性。

将代码存储在 **ActionScript** 文件中

如果您的项目中包括重要的 ActionScript 代码，则最好在单独的 ActionScript 源文件（扩展名为 .as 的文本文件）中组织这些代码。可以采用以下两种方式之一来设置 ActionScript 文件的结构，具体取决于您打算如何在应用程序中使用该文件。

- 非结构化 ActionScript 代码：编写 ActionScript 代码行（包括语句或函数定义），就好像它们是直接在时间轴脚本或 MXML 文件中输入的一样。

  使用 ActionScript 中的 include 语句或 Flex MXML 中的 <fx:Script> 标签，可以访问以此方式编写的 ActionScript。ActionScript include 语句告知编译器包含位于指定位置且属于脚本中给定范围内的外部 ActionScript 文件的内容。结果就像代码是在这些位置直接输入的一样。在 MXML 语言中，使用带源属性的 <fx:Script> 标签识别编译器在应用程序中该点加载的外部 ActionScript。例如，以下标签加载名为 Box.as 的外部 ActionScript 文件：

  ```
  <fx:Script source="Box.as" />
  ```

- ActionScript 类定义：定义一个 ActionScript 类，包含它的方法和属性。

  定义类后，通过创建类实例和使用其属性、方法和事件，您可以访问此类中的 ActionScript 代码。使用您自己的类与使用内置 ActionScript 类相同，都需要两个步骤：

  - 使用 import 语句来指定该类的全名，以便 ActionScript 编译器知道可以在哪里找到它。例如，若要使用 ActionScript 中的 MovieClip 类，则使用该类的全名（包括包和类）将其导入：

    ```
    import flash.display.MovieClip;
    ```

    或者，您也可以导入包含该 MovieClip 类的包，这与针对该包中的每个类编写单独的 import 语句是等效的：

    ```
    import flash.display.*;
    ```

    顶级类是该规则的唯一例外，要在代码中使用某个类，必须导入该类。这些类不是在包中定义的。

  - 编写专门使用该类名称的代码。例如，将该类的一个变量声明为其数据类型，然后创建该类的一个实例并将其存储到此变量中。在 ActionScript 代码中使用某个类，实际上也就通知了编译器加载该类的定义。例如，假设给定一个名为 Box 的外部类，此语句会创建 Box 类的一个实例：

    ```
    var smallBox:Box = new Box(10,20);
    ```

    编译器第一次遇到 Box 类的引用时，会搜索可用的源代码来查找 Box 类定义。

选择合适的工具

可以使用其中一种工具（或同时配合使用多种工具）来编写和编辑 ActionScript 代码。

Flash Builder

Adobe Flash Builder 是创建使用 Flex 框架的项目或主要包含 ActionScript 代码的项目的首要工具。Flash Builder 还包括功能齐全的 ActionScript 编辑器、可视布局和 MXML 编辑功能。它可用于创建 Flex 项目或纯 ActionScript 项目。Flex 具有以下几个优点：包含一组内容丰富的预置用户界面控件、灵活的动态布局控件以及用于使用远程数据并将外部数据链接到用户界面元素的内置机制。然而，由于要求使用更多的代码提供这些功能，因此，与不使用 Flex 的项目相比，使用 Flex 的项目的 SWF 文件比较大。

如果您正在创建使用 Flex 的功能齐全、数据驱动的富 Internet 应用程序，请使用 Flash Builder。您可以用它来编辑 ActionScript 代码、编辑 MXML 代码以及直观地展示应用程序，利用这一个工具即可完成上述所有任务。

对于许多构建大量使用 ActionScript 的项目的 Flash Professional 用户来说，可以使用 Flash Professional 来创建可视资源，使用 Flash Builder 作为 ActionScript 代码的编辑器。

Flash Professional

Flash Professional 除了具有图形和动画制作功能外，还包括用来处理 ActionScript 代码的工具。代码可以附加到 FLA 文件中的元素中，也可以附加到外部纯 ActionScript 文件的元素中。 Flash Professional 是包含重要动画或视频的项目的理想选择。当您想自己制作大部分图形资源时，这个工具很有用。使用 Flash Professional 开发 ActionScript 项目的另一个原因，是可以在同一应用程序中创建可视资源和编写代码。 Flash Professional 还包括预置的用户界面组件。您可以使用这些组件来获得更小的 SWF 文件大小，以及利用可视工具获得项目的外观。

Flash Professional 包括两个编写 ActionScript 代码的工具：

- “动作”面板：在 FLA 文件中工作时可用，该面板允许您编写附加到时间轴上的帧的 ActionScript 代码。
- “脚本”窗口：“脚本”窗口是用于处理 ActionScript (.as) 代码文件的专用文本编辑器。

第三方 ActionScript 编辑器

由于 ActionScript (.as) 文件存储为简单的文本文件，因此任何能够编辑纯文本文件的程序都可以用来编写 ActionScript 文件。除了 Adobe 的 ActionScript 产品之外，还有几个拥有特定于 ActionScript 的功能的第三方文本编辑程序。您可以使用任何文本编辑程序来编写 MXML 文件或 ActionScript 类。然后，您可以使用 Flex SDK 从这些文件创建应用程序。项目可以使用 Flex，也可以是纯 ActionScript 应用程序。此外，一些开发人员使用 Flash Builder 或第三方 ActionScript 编辑器来编写 ActionScript 类，结合使用 Flash Professional 来创建图片内容。

选择使用第三方 ActionScript 编辑器的理由如下：

- 您喜欢在单独的程序中编写 ActionScript 代码，在 Flash Professional 中设计可视元素。
- 将某个应用程序用于非 ActionScript 编程 （例如，创建 HTML 页或以其他编程语言构建应用程序）。您希望将该应用程序也用于 ActionScript 编码。
- 您希望使用 Flex SDK 而不用 Flash Professional 和 Flash Builder 来创建纯 ActionScript 项目或 Flex 项目。

有一些提供特定于 ActionScript 的支持的代码编辑器值得注意，其中包括：

- Adobe Dreamweaver® CS4
- ASDT
- FDT
- FlashDevelop
- PrimalScript
- SE|PY
- TextMate （包含 ActionScript 和 Flex 绑定）

ActionScript 开发过程

无论您的 ActionScript 项目大还是小，按一定过程来设计和开发应用程序会提高工作效率。下面几个步骤说明了构建使用 ActionScript 3.0 的应用程序的基本开发过程：

1 设计应用程序。

先以某种方式描述应用程序，然后再开始构建该应用程序。

2 编写 ActionScript 3.0 代码。

您可以使用 Flash Professional、 Flash Builder、 Dreamweaver 或文本编辑器来创建 ActionScript 代码。

3 创建 Flash 或 Flex 项目来运行代码。

在 Flash Professional 中，创建 FLA 文件、设置发布设置、向应用程序添加用户界面组件并引用 ActionScript 代码。在 Flex 中，定义应用程序、使用 MXML 添加用户界面组件并引用 ActionScript 代码。

4 发布和测试 ActionScript 应用程序。

测试应用程序包括在开发环境中运行应用程序和确保应用程序各方面都符合您的预期。

不必按顺序执行这些步骤，或者说，不必等待一个步骤彻底完成后再去执行另一步骤。例如，您可以先设计应用程序的一个屏幕 （步骤 1），然后创建图形、按钮等等 （步骤 3），最后再编写 ActionScript 代码 （步骤 2）并进行测试 （步骤 4）。您也可以先设计应用程序的一部分，然后再一次添加一个按钮或一个界面元素，并为每个按钮或界面元素编写 ActionScript，在生成后对之进行测试。记住开发过程的这四个步骤很有用。然而，在实际开发中，根据需要在各阶段进行前后调整更为有效。

创建自己的类

创建在项目中使用的类的过程可能令人望而生畏。但是，在创建类的过程中，更难的部分是设计类的方法、属性和事件。

类设计策略

面向对象的设计这一主题较为复杂；整个行业的人员都对此学科进行了大量的学术研究和专业实践。尽管如此，下面还是给出了几条建议以帮助您着手进行面向对象的编程。

1 请考虑一下该类的实例在应用程序中扮演的角色。通常，对象担任以下三种角色之一：

- 值对象：这些对象主要作为数据的容器，可能具有几个属性和为数不多的方法 （有时甚至没有方法）。它们通常是明确定义的项目的代码表示。例如，音乐播放器应用程序可以包含一个表示一首实际歌曲的 Song 类和一个表示一组概念歌曲的 Playlist 类。
- 显示对象：它们是实际显示在屏幕上的对象。例如，用户界面元素 （如下拉列表或状态显示）和图形元素 （如视频游戏中的角色）等就是显示对象。
- 应用程序结构：这些对象在应用程序执行的逻辑或处理方面扮演着广泛的支持角色。例如，您可以在生物学模拟中安排某个对象执行特定的计算。您可以安排一个对象负责同步音乐播放器应用程序中拨号控件和音量显示之间的值。另一个对象可安排来管理视频游戏中的规则。或者，您还可以安排一个类在绘图应用程序中加载保存的图片。

2 确定类所需的特定功能。不同类型的功能通常会成为类的方法。

3 如果打算将类用作值对象，请确定实例包含的数据。这些项是很好的候选属性。

4 由于类是专门为项目而设计的，因此最重要的是提供应用程序所需的功能。尝试自己独立回答以下问题：

- 应用程序存储、跟踪和处理哪些信息？回答此问题可以帮助您识别所需的任何值对象和属性。
- 应用程序执行哪些动作组合？例如，应用程序第一次加载时、单击特定按钮时以及停止播放影片时会出现什么情况？这些是很好的候选方法。如果“动作”包含更改单独的值，这些也可以是属性。
- 对于任何给定的动作，执行该动作需要哪些信息？这些信息将成为方法的参数。
- 随着应用程序开始工作，应用程序的其他部分需要了解类中的哪些内容会发生更改？这些是很好的候选事件。

5 是不是现有对象类似于您需要的对象，但是缺少一些您希望添加的其他功能？请考虑创建子类。（子类是在现有类的功能的基础上构建的类，它不定义自己的全部功能。）例如，若要创建一个在屏幕上是可视对象的类，则使用现有显示对象的行为作为您的类的基础。在这种情况下，显示对象 （如 MovieClip 或 Sprite）是基类，而您的类是该类的扩展。

编写类的代码

想明白类的设计，或者至少想明白它存储哪些信息、执行哪些动作后，实际编写类的语法非常简单。

下面是创建自己的 ActionScript 类的最基本步骤：

1 在 ActionScript 文本编辑器程序中打开一个新的文本文档。

2 输入 class 语句定义类的名称。要添加 class 语句，先输入单词 public class，然后输入类的名称。添加左大括号和右大括号，在其中包含类的内容 （方法和属性定义）。例如：

```
public class MyClass
{
}
```

单词 public 表示可以从任何其他代码中访问该类。有关其他选项，请参阅访问控制命名空间属性。

3 键入 package 语句来指示包含您的类的包的名称。语法形式为：单词 package，后面依次是完整包名称和左大括号与右大括号，括号中内容为 class 语句块。例如，将上一步中的代码更改为以下形式：

```
package mypackage
{
    public class MyClass
    {
    }
}
```

4 在类体中使用 var 语句定义该类中的每个属性；语法与用于声明变量的语法相同 （增加了 public 修饰符）。例如，在类定义的左大括号与右大括号之间添加下列行将创建名为 textProperty、 numericProperty 和 dateProperty 的属性：

```
public var textProperty:String = "some default value";
public var numericProperty:Number = 17;
public var dateProperty:Date;
```

5 使用与函数定义所用的相同语法来定义类中的每个方法。例如：

- 要创建 myMethod() 方法，应输入：

```
public function myMethod(param1:String, param2:Number):void
{
    // do something with parameters
}
```

- 要创建一个构造函数 （在创建类实例的过程中调用的特殊方法），应创建一个名称与类名称完全匹配的方法：

```
public function MyClass()
{
    // do stuff to set initial values for properties
    // and otherwise set up the object
    textVariable = "Hello there!";
    dateVariable = new Date(2001, 5, 11);
}
```

如果类中未包含构造函数方法，编译器会自动在您的类中创建一个空构造函数。（即不带任何参数和语句的构造函数。）

您可以定义更多类元素。这些元素相对复杂一些。

- 存取器 是方法与属性之间的一个特殊交点。在编写代码来定义类时，可以像编写方法一样来编写取值函数。您可以执行多个动作，而不是像在定义属性时那样只能读取值或赋值。但是，在创建类的实例时，可将取值函数视为属性，并使用名称来读取值或赋值。
- ActionScript 中的事件不是使用特定语法来定义的。您需要使用 EventDispatcher 类的功能在您的类中定义事件。

更多帮助主题

处理事件

示例：创建基本应用程序

ActionScript 3.0 可以在许多应用程序开发环境中使用，这些环境包括 Flash Professional、Flash Builder 工具或任意文本编辑器。

本示例将引导您完成使用 Flash Professional 或 Flash Builder 创建并增强一个简单的 ActionScript 3.0 应用程序的步骤。您构建的应用程序体现了一种在 Flash Professional 和 Flex 中使用外部 ActionScript 3.0 类文件的简单模式。

设计 ActionScript 应用程序

本示例 ActionScript 应用程序是一个标准 “Hello World” 应用程序，设计非常简单：

- 此应用程序名为 HelloWorld，
- 将显示一个包含 “Hello World!” 字样的文本字段。
- 此应用程序使用一个名为 Greeter 的面向对象的类。这种设计允许在 Flash Professional 或 Flex 项目中使用该类。
- 在本示例中，首先创建该应用程序的基本版本。然后添加功能，使用户可以输入用户名，使应用程序可以对照已知用户列表检查名称。

有了这一简明的定义之后，您可以开始构建该应用程序了。

创建 HelloWorld 项目和 Greeter 类

Hello World 应用程序的设计说明指出该应用程序的代码易于重用。为了达到这个目标，应用程序使用一个名为 Greeter 的面向对象的类。在您于 Flash Builder 或 Flash Professional 中创建的应用程序中使用该类。

要在 **Flex** 中创建 **HelloWorld** 项目和 **Greeter** 类，请执行以下操作：

1 在 Flash Builder 中，选择 “ 文件 ”>“ 新建 ”>“Flex 项目 ”，

2 键入 HelloWorld 作为项目名称。确保应用程序类型设置为 “Web (在 Adobe Flash Player 中运行)”，然后单击 “ 完成 ”。

Flash Builder 将创建项目，然后在包资源管理器中显示该项目。默认情况下，此项目中已有一个名为 HelloWorld.mxml 的文件，并且该文件已在编辑器中打开。

3 现在开始在 Flash Builder 中创建一个自定义 ActionScript 类文件。请选择 “ 文件 ”>“ 新建 ”>“ActionScript 类 ”。

4 在 “ 新建 ActionScript 类 ” 对话框的 “ 名称 ” 字段中，键入 **Greeter** 作为类名称，然后单击 “ 完成 ”。

将显示一个新的 ActionScript 编辑窗口。

继续完成在 Greeter 类中添加代码。

要在 **Flash Professional** 中创建 **Greeter** 类，请执行以下操作：

1 在 Flash Professional 中，选择 “ 文件 ”>“ 新建 ”。

2 在 “ 新建文档 ” 对话框中，选择 “ActionScript 文件 ”，然后单击 “ 确定 ”。

将显示一个新的 ActionScript 编辑窗口。

3 选择 “ 文件 ”>“ 保存 ”。选择一个文件夹以包含您的应用程序，将 ActionScript 文件命名为 **Greeter.as**，然后单击 “ 确定 ”。

继续完成在 Greeter 类中添加代码。

在 Greeter 类中添加代码

Greeter 类定义一个对象 Greeter，您将在 HelloWorld 应用程序中使用该对象。

要向 **Greeter** 类中添加代码，请执行以下操作：

1 在新文件中键入下列代码 （有些代码可能已添加）：

```
package
{
    public class Greeter
    {
        public function sayHello():String
        {
            var greeting:String;
            greeting = "Hello World!";
            return greeting;
        }
    }
}
```

Greeter 类包含一个 sayHello() 方法，该方法返回字符串 “Hello World!”。

2 选择 “ 文件 ”>“ 保存 ” 保存此 ActionScript 文件。

Greeter 类准备就绪，可在应用程序中使用。

创建使用 ActionScript 代码的应用程序

您构建的 Greeter 类定义一组自包含的软件功能，但它不表示完整的应用程序。为使用该类，创建一个 Flash Professional 文档或 Flex 项目。

该代码需要 Greeter 类的实例。下面说明如何在应用程序中使用 Greeter 类。

要使用 **Flash Professional** 创建 **ActionScript** 应用程序，请执行以下操作：

1 选择 “ 文件 ”>“ 新建 ”。

2 在 “ 新建文档 ” 对话框中，选择 “Flash 文件 (ActionScript 3.0)”，然后单击 “ 确定 ”。

将显示一个新的文档窗口。

3 选择 “ 文件 ”>“ 保存 ”。选择包含该 Greeter.as 类文件的同一文件夹，将 Flash 文档命名为 **HelloWorld.fla**，然后单击 “ 确定 ”。

4 在 Flash Professional 工具调板中，选择 “ 文本 ” 工具。在舞台上拖动来定义一个大约 300 像素宽、100 像素高的新文本字段。

5 在 “ 属性 ” 面板中，在仍然选中舞台中的文本字段时，将文本类型设置为 “ 动态文本 ”。键入 **mainText** 作为该文本字段的实例名称。

6 单击主时间轴的第 1 帧。选择 “ 窗口 ”>“ 动作 ” 来打开 “ 动作 ” 面板。

7 在 “ 动作 ” 面板中键入以下脚本：

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello();
```

8 保存该文件。

继续完成发布和测试 ActionScript 应用程序。

要使用 **Flash Builder** 创建 **ActionScript** 应用程序，请执行以下操作：

1 打开 HelloWorld.mxml 文件，添加代码来匹配下列列表：

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400"/>

</s:Application>
```

此 Flex 项目包括四个 MXML 标签：

- 一个 <s:Application> 标签，该标签定义 Application 容器
- 一个 <s:layout> 标签，定义 Application 标签的布局样式 （垂直布局）
- 包含一些 ActionScript 代码的 <fx:Script> 标签
- 一个 <s:TextArea> 标签，该标签定义用以向用户显示文本消息的一个字段

<fx:Script> 标签中的代码，定义一个在加载应用程序时调用的 initApp() 方法。 initApp() 方法将 mainTxt TextArea 的文本值设置为刚刚编写的自定义类 Greeter 的 sayHello() 方法所返回的字符串 “Hello World!”。

2 选择 “ 文件 ”>“ 保存 ” 以保存应用程序。

继续完成发布和测试 ActionScript 应用程序。

发布和测试 ActionScript 应用程序

软件开发是一个重复的过程。编写一些代码，尝试编译这些代码，然后编辑代码，直到能够完全编译这些代码为止。运行已编译的应用程序，测试该应用程序是否符合设计预期。如果不符合，则再次编辑代码，直到符合设计预期。 Flash Professional 和 Flash Builder 开发环境提供了多种发布、测试和调试应用程序的方式。

下面是在每种环境中测试 HelloWorld 应用程序的基本步骤。

要使用 **Flash Professional** 发布和测试 **ActionScript** 应用程序，请执行以下操作：

1 发布您的应用程序并观察编译错误。在 Flash Professional 中，选择 “ 控制 ”>“ 测试影片 ”，编译您的 ActionScript 代码并运行 HelloWorld 应用程序。

2 如果测试应用程序时 “ 输出 ” 窗口中显示任何错误或警告，则在 HelloWorld.fla 或 HelloWorld.as 文件中修复这些错误。然后尝试再次测试该应用程序。

3 如果没有编译错误，则会显示一个显示 Hello World 应用程序的 Flash Player 窗口。

您已经创建了一个简单但完整的面向对象的应用程序，该应用程序使用 ActionScript 3.0。继续改进 HelloWorld 应用程序。

要使用 **Flash Builder** 发布和测试 **ActionScript** 应用程序，请执行以下操作：

1 选择“运行”>“运行 HelloWorld”。

2 HelloWorld 应用程序启动。

- 如果测试应用程序时“输出”窗口中显示任何错误或警告，则在 HelloWorld.mxml 或 Greeter.as 文件中修复这些错误。然后尝试再次测试该应用程序。
- 如果没有任何编译错误，则会打开一个显示 Hello World 应用程序的浏览器窗口。应当显示文本“Hello World!”。

您已经创建了一个简单但完整的面向对象的应用程序，该应用程序使用 ActionScript 3.0。继续完成改进 HelloWorld 应用程序。

改进 HelloWorld 应用程序

若要使该应用程序更有趣，现在可让应用程序要求用户输入用户名并对照预定义的名称列表来验证该用户名。

首先，更新 Greeter 类以添加新功能。然后更新应用程序以使用新功能。

要更新 **Greeter.as** 文件，请执行以下操作：

1 打开 Greeter.as 文件。

2 将文件的内容更改为如下内容 （新行和更改的行以粗体显示）：

```
package
{
    public class Greeter
    {
        /**
         * Defines the names that receive a proper greeting.
         */
        public static var validNames:Array = ["Sammy", "Frank", "Dean"];

        /**
         * Builds a greeting string using the given name.
         */
        public function sayHello(userName:String = ""):String
        {
            var greeting:String;
            if (userName == "")
            {
                greeting = "Hello. Please type your user name, and then press "
                           + "the Enter key.";
            }
            else if (validName(userName))
            {
                greeting = "Hello, " + userName + ".";
            }
            else
            {
                greeting = "Sorry " + userName + ", you are not on the list.";
```

```
            }
            return greeting;
        }

        /**
         * Checks whether a name is in the validNames list.
         */
        public static function validName(inputName:String = ""):Boolean
        {
            if (validNames.indexOf(inputName) > -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

现在， Greeter 类拥有许多新功能:

- validNames 数组列出了有效的用户名。在加载 Greeter 类时该数组将初始化为包含三个名称的列表。
- sayHello() 方法现在接受一个用户名并根据一些条件来更改问候语。如果 userName 是一个空字符串 ("")，则 greeting 属性将设置为提示用户输入用户名的语句。如果用户名有效，则问候语会是 “Hello, userName"。最后，如果这两个条件都不满足，则 greeting 变量设置为 "Sorry userName, you are not on the list"。
- 如果在 validNames 数组中找到 inputName，则 validName() 方法将返回 true ；否则，返回 false。语句 validNames.indexOf(inputName) 对照 inputName 字符串检查 validNames 数组中的每个字符串。 Array.indexOf() 方法返回数组中某个对象的第一个实例的索引位置。如果在数组中找不到对象，则返回值 -1。

接下来，编辑引用此 ActionScript 类的应用程序文件。

要使用 **Flash Professional** 修改应用程序，请执行以下操作:

1. 打开 HelloWorld.fla 文件。
2. 修改第 1 帧中的脚本，将一个空字符串 ("") 传递到 Greeter 类的 sayHello() 方法:

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");
```

3. 从工具调板中选择文本工具。在舞台上新建两个文本字段。将这两个文本字段直接并排放在现有的 mainText 文本字段下面。
4. 第一个新文本字段是标签，在其中键入文本 **User Name:**。
5. 选择另一个新文本字段，并在 “ 属性 ” 检查器中选择 Input Text 作为该文本字段的类型。选择 “ 单行 ” 作为 “ 行类型 ”。键入 **textIn** 作为实例名。
6. 单击主时间轴的第 1 帧。
7. 在 “ 动作 ” 面板中，在现有脚本的末尾添加以下行:

```
mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

新代码添加以下功能：

- 前两行只定义两个文本字段的边框。
- 输入文本字段 （例如 textIn 字段）具有一组可以调度的事件。使用 addEventListener() 方法可以定义一个函数，该函数在发生某一类型的事件时运行。在本例中，该事件指的是按键盘上的某个键。
- keyPressed() 自定义函数会检查按下的键是否为 Enter 键。如果是，则该函数调用 myGreeter 对象的 sayHello() 方法，同时传递 textIn 文本字段中的文本作为参数。该方法基于传入的值返回字符串 “greeting”。返回的字符串随后分配给 mainText 文本字段的 text 属性。

第 1 帧的完整脚本如下所示：

```
var myGreeter:Greeter = new Greeter();
mainText.text = myGreeter.sayHello("");

mainText.border = true;
textIn.border = true;

textIn.addEventListener(KeyboardEvent.KEY_DOWN, keyPressed);

function keyPressed(event:KeyboardEvent):void
{
    if (event.keyCode == Keyboard.ENTER)
    {
        mainText.text = myGreeter.sayHello(textIn.text);
    }
}
```

8 保存该文件。

9 选择 “ 控制 ”>“ 测试影片 ” 以运行此应用程序。

运行该应用程序时，它会提示您输入用户名。如果用户名有效 （Sammy、 Frank 或 Dean），应用程序则会显示 “hello” 确认消息。

要使用 **Flash Builder** 修改应用程序，请执行以下操作：

1 打开 HelloWorld.mxml 文件。

2 接下来，修改 <mx:TextArea> 标签，告诉用户它只用于显示。将背景颜色更改为浅灰色，将 editable 属性设为 false：

```
    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false" />
```

3 现在，将以下各行添加到与 <s:TextArea> 结束标签紧邻的后面。这些行将创建一个 TextInput 组件，用户利用此组件可以输入用户名值：

```
    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>
```

enter 属性定义用户在 userNameTxt 字段中按下 Enter 键会发生什么情况。在本示例中，代码会将字段中的文本传递到 Greeter.sayHello() 方法。 mainTxt 字段中的问候语会随之更改。

HelloWorld.mxml 文件如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:s="library://ns.adobe.com/flex/spark"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    minWidth="1024"
    minHeight="768""
    creationComplete="initApp()">

    <fx:Script>
        <![CDATA[
            private var myGreeter:Greeter = new Greeter();

            public function initApp():void
            {
                // says hello at the start, and asks for the user's name
                mainTxt.text = myGreeter.sayHello();
            }
        ]]>
    </fx:Script>

    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <s:TextArea id="mainTxt" width="400" backgroundColor="#DDDDDD" editable="false"/>

    <s:HGroup width="400">
        <mx:Label text="User Name:"/>
        <s:TextInput id="userNameTxt" width="100%" enter="mainTxt.text =
myGreeter.sayHello(userNameTxt.text);" />
    </s:HGroup>

</s:Application>
```

4 保存经过编辑的 HelloWorld.mxml 文件。选择 “ 运行 ” >“ 运行 HelloWorld” 来运行应用程序。

运行该应用程序时，应用程序会提示您输入用户名。如果用户名有效 （Sammy、 Frank 或 Dean），应用程序会显示 “Hello, userName” 确认消息。

第 3 章 : ActionScript 语言及语法

ActionScript 3.0 既包含 ActionScript 核心语言，又包含 Adobe Flash Platform 应用程序编程接口 (API)。核心语言是定义语言语法以及顶级数据类型的 ActionScript 部分。 ActionScript 3.0 提供对 Adobe Flash Platform 运行时 （Adobe Flash Player 和 Adobe AIR）的编程访问。

语言概述

对象是 ActionScript 3.0 语言的核心 — 它们是 ActionScript 3.0 语言的基本构造块。您所声明的每个变量、所编写的每个函数以及所创建的每个类实例都是一个对象。可以将 ActionScript 3.0 程序视为一组执行任务、响应事件以及相互通信的对象。

熟悉 Java 或 C++ 中面向对象的编程 (OOP) 的程序员可能会将对象视为包含以下两类成员的模块：存储在成员变量或属性中的数据，以及可通过方法访问的行为。 ActionScript 3.0 采用一种相似但稍有不同的方式定义对象。在 ActionScript 3.0 中，对象只是属性的集合。这些属性是一些容器，除了保存数据，还保存函数或其他对象。以这种方式附加到对象的函数称为方法。

尽管具有 Java 或 C++ 背景的程序员可能会觉得 ActionScript 3.0 中的定义有些奇怪，但实际上，用 ActionScript 3.0 类定义对象类型与在 Java 或 C++ 中定义类的方式非常相似。在讨论 ActionScript 对象模型和其他高级主题时，了解这两种对象定义之间的区别很重要，但在其他大多数情况下，术语属性表示类成员变量 （与 “ 方法 ” 对应）。例如，《用于 Adobe Flash Platform 的 ActionScript 3.0 参考》使用术语属性 表示变量或 getter-setter 属性。使用术语方法表示作为类的一部分的函数。

ActionScript 中的类与 Java 或 C++ 中的类之间有一个细小的差别，那就是 ActionScript 中的类不仅仅是抽象实体。ActionScript 类由存储类的属性和方法的类对象表示。这样便可以使用可能不为 Java 和 C++ 程序员所熟悉的方法，例如，在类或包的顶级包括语句或可执行代码。

ActionScript 类与 Java 或 C++ 类之间还有一个区别，那就是每个 ActionScript 类都有一个原型对象。在早期版本的 ActionScript 中，原型对象链接成原型链，共同作为整个类继承层次的基础。但是，在 ActionScript 3.0 中，原型对象在继承系统中仅扮演很小的角色。但是，原型对象仍非常有用，如果您希望在类的所有实例中共享某个属性及其值，可以使用原型对象来代替静态属性和方法。

过去，高级 ActionScript 程序员可以用特殊的内置语言元素来直接操作原型链。现在，由于 ActionScript 语言为基于类的编程接口提供了更成熟的实现，因此其中的许多特殊语言元素 （如 __proto__ 和 __resolve）不再是该语言的一部分。而且，内部继承机制的优化预先排除了对继承机制的直接访问，从而大大改善了性能。

对象和类

在 ActionScript 3.0 中，每个对象都是由类定义的。可将类视为某一类对象的模板或蓝图。类定义中可以包括变量和常量以及方法，前者用于保存数据值，后者是封装绑定到类的行为的函数。存储在属性中的值可以是基元值，也可以是其他对象。基元值是指数字、字符串或布尔值。

ActionScript 中包含许多属于核心语言的内置类。其中的某些内置类（如 Number、Boolean 和 String）表示 ActionScript 中可用的基元值。其他类 （如 Array、 Math 和 XML 类）定义更加复杂的对象。

所有的类 （无论是内置类还是用户定义的类）都是从 Object 类派生的。以前在 ActionScript 方面有经验的程序员一定要注意到，Object 数据类型不再是默认的数据类型，尽管其他所有类仍从它派生。在 ActionScript 2.0 中，下面的两行代码等效，因为缺乏类型注释意味着变量为 Object 类型：

```
var someObj:Object;
var someObj;
```

但是， ActionScript 3.0 引入了无类型变量这一概念，这一类变量可通过以下两种方法来指定：

```
var someObj:*;
var someObj;
```

无类型变量与 Object 类型的变量不同。二者的主要区别在于无类型变量可以保存特殊值 undefined，而 Object 类型的变量则不能保存该值。

您可以使用 class 关键字来定义自己的类。在方法声明中，可通过以下三种方法来声明类属性：用 const 关键字定义常量，用 var 关键字定义变量，用 get 和 set 属性定义 getter 和 setter 属性。可以用 function 关键字来声明方法。

可使用 new 运算符来创建类的实例。下面的示例创建 Date 类的一个名为 myBirthday 的实例。

```
var myBirthday:Date = new Date();
```

包和命名空间

包和命名空间是两个相关的概念。使用包，可以通过有利于共享代码并尽可能减少命名冲突的方式将多个类定义捆绑在一起。使用命名空间，可以控制标识符 （如属性名和方法名）的可见性。无论命名空间位于包的内部还是外部，都可以应用于代码。包可用于组织类文件，命名空间可用于管理各个属性和方法的可见性。

包

在 ActionScript 3.0 中，包是用命名空间实现的，但包和命名空间并不同义。在声明包时，可以隐式创建一个特殊类型的命名空间并保证它在编译时是已知的。显式创建的命名空间在编译时不必是已知的。

下面的示例使用 package 指令来创建一个包含一个类的简单包：

```
package samples
{
    public class SampleCode
    {
        public var sampleGreeting:String;
        public function sampleFunction()
        {
            trace(sampleGreeting + " from sampleFunction()");
        }
    }
}
```

在本例中，该类的名称是 SampleCode。由于该类位于 samples 包中，因此编译器在编译时会自动将其类名称限定为完全限定名称：samples.SampleCode。编译器还限定任何属性或方法的名称，因此 sampleGreeting 和 sampleFunction() 分别变成 samples.SampleCode.sampleGreeting 和 samples.SampleCode.sampleFunction()。

许多开发人员 （尤其是那些具有 Java 编程背景的人）可能会选择只将类放在包的顶级。但是， ActionScript 3.0 不但支持将类放在包的顶级，而且还支持将变量、函数甚至语句放在包的顶级。此功能的一个高级用法是，在包的顶级定义一个命名空间，使之对于该包中的所有类均可用。但是，请注意，在包的顶级只允许使用两个访问说明符：public 和 internal。 Java 允许将嵌套类声明为私有，而 ActionScript 3.0 则不同，它既不支持嵌套类也不支持私有类。

但是，在其他许多方面， ActionScript 3.0 中的包与 Java 编程语言中的包非常相似。从上一个示例可看出，完全限定的包引用使用点运算符 (.) 来表示，这与 Java 相同。可以用包将代码组织成直观的分层结构，以供其他程序员使用。这样，您就可以将自己所创建的包与他人共享，还可以在自己的代码中使用他人创建的包，从而推动了代码共享。

使用包还有助于确保所使用的标识符名称是唯一的，而且不与其他标识符名称冲突。事实上，有些人认为这才是包的主要优点。例如，假设两个希望相互共享代码的编程人员各创建了一个名为 SampleCode 的类。如果没有包，这样就会造成名称冲突，唯一的解决方法就是重命名其中的一个类。但是，使用包，就可以将其中的一个 （最好是两个）类放在具有唯一名称的包中，从而轻松地避免了名称冲突。

您还可以在包名称中嵌入点来创建嵌套包，这样就可以创建包的分层结构。ActionScript 3.0 提供的 flash.display 包就是一个很好的示例。 flash.display 包嵌套在 flash 包中。

大部分 ActionScript 3.0 都划分到 flash 包中。例如，flash.display 包中包含显示列表 API，flash.events 包中包含新的事件模型。

创建包

ActionScript 3.0 在包、类和源文件的组织方式上具有很大的灵活性。早期的 ActionScript 版本只允许每个源文件有一个类，而且要求源文件的名称与类名称匹配。 ActionScript 3.0 允许在一个源文件中包括多个类，但是，每个文件中只有一个类可供该文件外部的代码使用。换言之，每个文件中只有一个类可以在包声明中进行声明。您必须在包定义的外部声明其他任何类，以使这些类对于该源文件外部的代码不可见。在包定义内部声明的类的名称必须与源文件的名称匹配。

ActionScript 3.0 在包的声明方式上也具有更大的灵活性。在早期的 ActionScript 版本中，包只是表示用于放置源文件的目录，您不必用 package 语句来声明包，而是在类声明中将包名称包括在完全限定的类名称中。在 ActionScript 3.0 中，尽管包仍表示目录，但是它现在不只包含类。在 ActionScript 3.0 中，使用 package 语句来声明包，这意味着您也可以在包的顶级声明变量、函数和命名空间，甚至还可以在包的顶级包括可执行语句。如果在包的顶级声明变量、函数或命名空间，则在顶级只能使用 public 和 internal 属性，并且每个文件中只能有一个包级声明使用 public 属性 （无论该声明是类声明、变量声明、函数声明还是命名空间声明）。

包的作用是组织代码并防止名称冲突。您不应将包的概念与类继承这一不相关的概念混淆。位于同一个包中的两个类具有共同的命名空间，但在其他任何方面，它们不必相关。同样，在语义方面，嵌套包可以与其父包无关。

导入包

如果您希望使用位于某个包内部的特定类，则必须导入该包或该类。这与 ActionScript 2.0 不同，在 ActionScript 2.0 中，类的导入是可选的。

例如，想想以前列举过的 SampleCode 类示例。如果该类位于名为 samples 的包中，那么，在使用 SampleCode 类之前，您必须使用下列导入语句之一：

```
import samples.*;
```

或

```
import samples.SampleCode;
```

通常，import 语句越具体越好。如果您只打算使用 samples 包中的 SampleCode 类，则应只导入 SampleCode 类，而不应导入该类所属的整个包。导入整个包可能会导致意外的名称冲突。

还必须将定义包或类的源代码放在类路径内部。类路径是用户定义的本地目录路径列表，它决定编译器在何处搜索导入的包和类。类路径有时称为生成路径或源路径。

在正确地导入类或包之后，可以使用类的完全限定名称 (samples.SampleCode)，也可以只使用类名称本身 (SampleCode)。

当同名的类、方法或属性会导致代码不明确时，完全限定的名称非常有用，但是，如果将它用于所有的标识符，则会使代码变得难以管理。例如，在实例化 SampleCode 类的实例时，使用完全限定的名称会导致代码冗长：

```
var mySample:samples.SampleCode = new samples.SampleCode();
```

包的嵌套级别越高，代码的可读性越差。如果您确信不明确的标识符不会导致问题，就可以通过使用简单的标识符来提高代码的可读性。例如，如果在实例化 SampleCode 类的新实例时仅使用类标识符，代码就会比较简洁：

```
var mySample:SampleCode = new SampleCode();
```

如果您尝试使用标识符名称，而不先导入相应的包或类，则编译器无法找到类定义。另一方面，即便您导入了包或类，只要尝试定义的名称与所导入的名称冲突，也会产生错误。

创建包时，该包的所有成员的默认访问说明符是 internal，这意味着，默认情况下，包成员仅对其所在包的其他成员可见。如果您希望某个类对包外部的代码可用，则必须将该类声明为 public。例如，下面的包包含 SampleCode 和 CodeFormatter 两个类：

```
// SampleCode.as file
package samples
{
    public class SampleCode {}
}

// CodeFormatter.as file
package samples
{
    class CodeFormatter {}
}
```

SampleCode 类在包的外部可见，因为该类声明为 public 类。但是，CodeFormatter 类仅在 samples 包的内部可见。如果您尝试访问位于 samples 包外部的 CodeFormatter 类，将产生错误，如下面的示例所示：

```
import samples.SampleCode;
import samples.CodeFormatter;
var mySample:SampleCode = new SampleCode(); // okay, public class
var myFormatter:CodeFormatter = new CodeFormatter(); // error
```

如果您希望这两个类在包外部都可用，就必须将它们都声明为 public。不能将 public 属性应用于包声明。

完全限定的名称可用来解决在使用包时可能发生的名称冲突。如果您导入两个包，但它们用同一个标识符来定义类，就可能会发生名称冲突。例如，请考虑下面的包，该包也有一个名为 SampleCode 的类：

```
package langref.samples
{
    public class SampleCode {}
}
```

如果按如下方式导入两个类，则使用 SampleCode 类时会发生名称冲突：

```
import samples.SampleCode;
import langref.samples.SampleCode;
var mySample:SampleCode = new SampleCode(); // name conflict
```

编译器无法确定要使用哪个 SampleCode 类。要解决此冲突，必须使用每个类的完全限定名称，如下所示：

```
var sample1:samples.SampleCode = new samples.SampleCode();
var sample2:langref.samples.SampleCode = new langref.samples.SampleCode();
```

注：具有 C++ 背景的程序员经常会将 import 语句与 #include 混淆。#include 指令在 C++ 中是必需的，因为 C++ 编译器一次处理一个文件，而且除非显式包括了头文件，否则不会在其他文件中查找类定义。ActionScript 3.0 有一个 include 指令，但是它的作用不是为了导入类和包。要在 ActionScript 3.0 中导入类或包，必须使用 import 语句，并将包含该包的源文件放在类路径中。

命名空间

通过命名空间可以控制所创建的属性和方法的可见性。请将 public、 private、 protected 和 internal 访问控制说明符视为内置的命名空间。如果这些预定义的访问控制说明符无法满足您的要求，您可以创建自己的命名空间。

如果您熟悉 XML 命名空间，那么，您对这里讨论的大部分内容不会感到陌生，但是 ActionScript 实现的语法和细节与 XML 的稍有不同。即使您以前从未使用过命名空间，也没有关系，因为命名空间概念本身很简单，但是其实现涉及一些您需要了解的特定术语。

要了解命名空间的工作方式，有必要先了解属性或方法的名称总是包含两部分：标识符和命名空间。标识符通常被视为名称。例如，下面的类定义中的标识符是 sampleGreeting 和 sampleFunction()：

```
class SampleCode
{
    var sampleGreeting:String;
    function sampleFunction () {
        trace(sampleGreeting + " from sampleFunction()");
    }
}
```

只要定义不以命名空间属性开头，其名称就会用默认 internal 命名空间限定，这意味着，这些定义仅对同一个包中的调用方可见。如果编译器设置为严格模式，则编译器会发出一个警告，指明 internal 命名空间将应用于没有命名空间属性的任何标识符。为了确保标识符可在任何位置使用，您必须在标识符名称的前面明确加上 public 属性。在上面的示例代码中， sampleGreeting 和 sampleFunction() 都有一个命名空间值 internal。

使用命名空间时，应遵循以下三个基本步骤。第一步，必须使用 namespace 关键字来定义命名空间。例如，下面的代码定义 version1 命名空间：

```
namespace version1;
```

第二步，在属性或方法声明中，使用命名空间 （而非访问控制说明符）来应用命名空间。下面的示例将一个名为 myFunction() 的函数放在 version1 命名空间中：

```
version1 function myFunction() {}
```

第三步，在应用了该命名空间后，可以使用 use 指令进行引用，也可以使用命名空间来限定标识符的名称。下面的示例通过 use 指令来引用 myFunction() 函数：

```
use namespace version1;
myFunction();
```

还可以使用限定名称引用 myFunction() 函数，如下面的示例所示：

```
version1::myFunction();
```

定义命名空间

命名空间中包含一个名为统一资源标识符 (URI) 的值，该值有时称为命名空间名称。使用 URI 可确保命名空间定义的唯一性。

可通过使用以下两种方法之一来声明命名空间定义，以创建命名空间：像定义 XML 命名空间那样使用显式 URI 定义命名空间；省略 URI。下面的示例说明如何使用 URI 来定义命名空间：

```
namespace flash_proxy = "http://www.adobe.com/flash/proxy";
```

URI 用作该命名空间的唯一标识字符串。如果您省略 URI （如下面的示例所示），编译器将创建一个唯一的内部标识字符串来代替 URI。您对于这个内部标识字符串不具有访问权限。

```
namespace flash_proxy;
```

在定义了命名空间 （具有 URI 或没有 URI）后，就不能在同一个作用域内重新定义该命名空间。如果尝试定义的命名空间以前在同一个作用域内定义过，则将生成编译器错误。

如果在某个包或类中定义了一个命名空间，则该命名空间可能对于此包或类外部的代码不可见，除非使用了相应的访问控制说明符。例如，下面的代码显示了在 flash.utils 包中定义的 flash_proxy 命名空间。下面的示例中没有使用访问控制说明符，这意味着 flash_proxy 命名空间仅对于 flash.utils 包内部的代码可见，而对于该包外部的任何代码都不可见：

```
package flash.utils
{
    namespace flash_proxy;
}
```

下面的代码使用 public 属性使 flash_proxy 命名空间对该包外部的代码可见：

```
package flash.utils
{
    public namespace flash_proxy;
}
```

应用命名空间

应用命名空间意味着在命名空间中放置定义。可以放在命名空间中的定义包括函数、变量和常量 （不能将类放在自定义命名空间中）。

例如，请考虑一个使用 public 访问控制命名空间声明的函数。在函数的定义中使用 public 属性会将该函数放在 public 命名空间中，从而使该函数对于所有的代码都可用。在定义命名空间之后，可以按照与使用 public 属性相同的方式来使用所定义的命名空间，该定义对于可以引用您的自定义命名空间的代码可用。例如，如果您定义一个名为 example1 的命名空间，则可以添加一个名为 myFunction() 的方法并将 example1 用作属性，如下面的示例所示：

```
namespace example1;
class someClass
{
    example1 myFunction() {}
}
```

如果在声明 myFunction() 方法时将 example1 命名空间用作属性，则意味着该方法属于 example1 命名空间。

在应用命名空间时，应切记以下几点：

- 对于每个声明只能应用一个命名空间。
- 不能一次将同一个命名空间属性应用于多个定义。换言之，如果您希望将自己的命名空间应用于 10 个不同的函数，则必须将该命名空间作为属性分别添加到这 10 个函数的定义中。
- 如果您应用了命名空间，则不能同时指定访问控制说明符，因为命名空间和访问控制说明符是互斥的。换言之，如果应用了命名空间，就不能将函数或属性声明为 public、 private、 protected 或 internal。

引用命名空间

在使用借助于任何访问控制命名空间 （如 public、 private、 protected 和 internal）声明的方法或属性时，无需显式引用命名空间。这是因为对于这些特殊命名空间的访问由上下文控制。例如，放在 private 命名空间中的定义会自动对于同一个类中的代码可用。但是，对于您所定义的命名空间，并不存在这样的上下文相关性。要使用已经添加到某个自定义命名空间中的方法或属性，必须引用该命名空间。

可以用 use namespace 指令来引用命名空间，也可以通过名称限定符 (::) 来使用命名空间限定名称。用 use namespace 指令引用命名空间会 “ 打开 ” 该命名空间，以便将该命名空间应用于任何未限定的标识符。例如，如果您已经定义了 example1 命名空间，则可以通过使用 use namespace example1 来访问该命名空间中的名称：

```
use namespace example1;
myFunction();
```

一次可以打开多个命名空间。在使用 use namespace 打开了某个命名空间之后，它会在打开它的整个代码块中保持打开状态。不能显式关闭命名空间。

但是，如果同时打开多个命名空间则会增加发生名称冲突的可能性。如果您不愿意打开命名空间，则可以用命名空间和名称限定符来限定方法或属性名，从而避免使用 use namespace 指令。例如，下面的代码说明如何用 example1 命名空间来限定 myFunction() 名称：

```
example1::myFunction();
```

使用命名空间

在包含在 ActionScript 3.0 中的 flash.utils.Proxy 类中，可以找到用来防止名称冲突的命名空间的实际示例。Proxy 类取代了 ActionScript 2.0 中的 Object.__resolve 属性，可用来截获对未定义的属性或方法的引用，以免发生错误。为避免名称冲突，Proxy 类的所有方法都位于 flash_proxy 命名空间中。

为了更好地了解 flash_proxy 命名空间的使用方法，您需要了解如何使用 Proxy 类。 Proxy 类的功能仅对于继承它的类可用。换言之，如果您要对某个对象使用 Proxy 类的方法，则该对象的类定义必须扩展 Proxy 类。例如，如果您希望截获对未定义的方法的调用，则应扩展 Proxy 类，然后覆盖 Proxy 类的 callProperty() 方法。

前面已讲到，实现命名空间的过程通常分为三步，即定义、应用然后引用命名空间。但是，由于您从不显式调用 Proxy 类的任何方法，因此 flash_proxy 命名空间只会被定义和应用，而不会被引用。 ActionScript 3.0 定义 flash_proxy 命名空间并在 Proxy 类中应用它。在您的代码中，只需要将 flash_proxy 命名空间应用于扩展 Proxy 类的类。

flash_proxy 命名空间按照与下面类似的方法在 flash.utils 包中定义：

```
package flash.utils
{
    public namespace flash_proxy;
}
```

该命名空间将应用于 Proxy 类的方法，如下面摘自 Proxy 类的代码所示：

```
public class Proxy
{
    flash_proxy function callProperty(name:*, ... rest):*
    flash_proxy function deleteProperty(name:*):Boolean
    ...
}
```

如下面的代码所示，您必须先导入 Proxy 类和 flash_proxy 命名空间。随后必须声明自己的类，以便它对 Proxy 类进行扩展（如果是在严格模式下进行编译，则还必须添加 dynamic 属性）。在覆盖 callProperty() 方法时，必须使用 flash_proxy 命名空间。

```
package
{
    import flash.utils.Proxy;
    import flash.utils.flash_proxy;

    dynamic class MyProxy extends Proxy
    {
        flash_proxy override function callProperty(name:*, ...rest):*
        {
            trace("method call intercepted: " + name);
        }
    }
}
```

如果您创建 MyProxy 类的一个实例，并调用一个未定义的方法 （如在下面的示例中调用的 testing() 方法），则 Proxy 对象会截获对该方法的调用，并执行覆盖后的 callProperty() 方法内部的语句 （在本例中为一个简单的 trace() 语句）。

```
var mySample:MyProxy = new MyProxy();
mySample.testing(); // method call intercepted: testing
```

将 Proxy 类的方法放在 flash_proxy 命名空间内部有两个好处。第一个好处是，在扩展 Proxy 类的任何类的公共接口中，拥有单独的命名空间可提高代码的可读性。（在 Proxy 类中大约有 12 个可以覆盖的方法，所有这些方法都不能直接调用。将所有这些方法都放在公共命名空间中可能会引起混淆。）第二个好处是，当 Proxy 子类中包含名称与 Proxy 类方法的名称匹配的实例方法时，使用 flash_proxy 命名空间可避免名称冲突。例如，您可能希望将自己的某个方法命名为 callProperty()。下面的代码是可接受的，因为您所用的 callProperty() 方法位于另一个命名空间中：

```
dynamic class MyProxy extends Proxy
{
    public function callProperty() {}
    flash_proxy override function callProperty(name:*, ...rest):*
    {
        trace("method call intercepted: " + name);
    }
}
```

当您希望以一种无法由四个访问控制说明符（public、private、internal 和 protected）实现的方式提供对方法或属性的访问时，命名空间也可能会非常有用。例如，您可能有几个分散在多个包中的实用程序方法。您希望这些方法对于您的所有包均可用，但是不希望这些方法成为公共方法。为此，您可以创建一个命名空间，并将它用作您自己的特殊访问控制说明符。

下面的示例使用用户定义的命名空间将两个位于不同包中的函数组合在一起。通过将这两个函数组合到同一个命名空间中，可以通过一条 use namespace 语句使这两个函数对于某个类或某个包可见。

本示例使用四个文件来说明此方法。所有的文件都必须位于您的类路径中。第一个文件 (myInternal.as) 用来定义 myInternal 命名空间。由于该文件位于名为 example 的包中，因此您必须将该文件放在名为 example 的文件夹中。该命名空间标记为 public，因此可以导入到其他包中。

```
// myInternal.as in folder example
package example
{
    public namespace myInternal = "http://www.adobe.com/2006/actionscript/examples";
}
```

第二个文件 (Utility.as) 和第三个文件 (Helper.as) 定义的类中包含应可供其他包使用的方法。Utility 类位于 example.alpha 包中，这意味着该文件应放在 example 文件夹下的 alpha 子文件夹中。Helper 类位于 example.beta 包中，这意味着该文件应放在 example 文件夹下的 beta 子文件夹中。这两个包（example.alpha 和 example.beta）在使用命名空间之前必须先导入它。

```
// Utility.as in the example/alpha folder
package example.alpha
{
    import example.myInternal;

    public class Utility
    {
        private static var _taskCounter:int = 0;

        public static function someTask()
        {
            _taskCounter++;
        }

        myInternal static function get taskCounter():int
        {
            return _taskCounter;
        }
    }
}
```

```
// Helper.as in the example/beta folder
package example.beta
{
    import example.myInternal;

    public class Helper
    {
        private static var _timeStamp:Date;

        public static function someTask()
        {
            _timeStamp = new Date();
        }

        myInternal static function get lastCalled():Date
        {
            return _timeStamp;
        }
    }
}
```

第四个文件 (NamespaceUseCase.as) 是主应用程序类，应是 example 文件夹的同级。在 Flash Professional 中，此类用作 FLA 的文档类。 NamespaceUseCase 类还导入 myInternal 命名空间，并使用该命名空间调用两个位于其他包中的静态方法。在本示例中，使用静态方法的目的仅在于简化代码。在 myInternal 命名空间中既可以放置静态方法也可以放置实例方法。

```
// NamespaceUseCase.as
package
{
    import flash.display.MovieClip;
    import example.myInternal; // import namespace
    import example.alpha.Utility;// import Utility class
    import example.beta.Helper;// import Helper class

    public class NamespaceUseCase extends MovieClip
    {
        public function NamespaceUseCase()
        {
            use namespace myInternal;

            Utility.someTask();
            Utility.someTask();
            trace(Utility.taskCounter); // 2

            Helper.someTask();
            trace(Helper.lastCalled); // [time someTask() was last called]
        }
    }
}
```

变量

变量可用来存储程序中使用的值。要声明变量，必须将 var 语句和变量名结合使用。在 ActionScript 3.0 中，总是需要使用 var 语句。例如，下面的 ActionScript 行声明一个名为 i 的变量：

```
var i;
```

如果在声明变量时省略了 var 语句，则在严格模式下会出现编译器错误，在标准模式下会出现运行时错误。例如，如果以前未定义变量 i，则下面的代码行会产生错误：

```
i; // error if i was not previously defined
```

要将变量与一个数据类型相关联，则必须在声明变量时进行此操作。声明变量时不指定变量的类型是合法的，但在严格模式下，这样做会产生编译器警告。可通过在变量名后面追加一个后跟变量类型的冒号 (:) 来指定变量类型。例如，下面的代码声明一个 int 类型的变量 i:

```
var i:int;
```

可以使用赋值运算符 (=) 为变量赋值。例如，下面的代码声明一个变量 i 并将值 20 赋给它:

```
var i:int;
i = 20;
```

您可能会发现在声明变量的同时为变量赋值可能更加方便，如下面的示例所示:

```
var i:int = 20;
```

通常，在声明变量的同时为变量赋值的方法不仅在赋予基元值 （如整数和字符串）时很常用，而且在创建数组或实例化类的实例时也很常用。下面的示例显示了一个使用一行代码声明和赋值的数组。

```
var numArray:Array = ["zero", "one", "two"];
```

可以使用 new 运算符来创建类的实例。下面的示例创建一个名为 CustomClass 的实例，并向名为 customItem 的变量赋予对该实例的引用:

```
var customItem:CustomClass = new CustomClass();
```

如果要声明多个变量，则可以使用逗号运算符 (,) 来分隔变量，从而在一行代码中声明所有这些变量。例如，下面的代码在一行代码中声明 3 个变量:

```
var a:int, b:int, c:int;
```

也可以在同一行代码中为其中的每个变量赋值。例如，下面的代码声明 3 个变量 （a、 b 和 c）并为每个变量赋值:

```
var a:int = 10, b:int = 20, c:int = 30;
```

尽管您可以使用逗号运算符来将各个变量的声明组合到一条语句中，但是这样可能会降低代码的可读性。

了解变量的作用域

变量的作用域是指可在其中通过词汇引用来访问变量的代码区域。全局变量是指在代码的所有区域中定义的变量，而局部变量是指仅在代码的某个部分定义的变量。在 ActionScript 3.0 中，始终为变量分配声明它们的函数或类的作用域。全局变量是在任何函数或类定义的外部定义的变量。例如，下面的代码通过在任何函数的外部声明一个名为 strGlobal 的全局变量来创建该变量。从该示例可看出，全局变量在函数定义的内部和外部均可用。

```
var strGlobal:String = "Global";
function scopeTest()
{
    trace(strGlobal); // Global
}
scopeTest();
trace(strGlobal); // Global
```

可以通过在函数定义内部声明变量来将它声明为局部变量。可定义局部变量的最小代码区域就是函数定义。在函数内部声明的局部变量仅存在于该函数中。例如，如果在名为 localScope() 的函数中声明一个名为 str2 的变量，该变量在该函数外部是不可用的。

```
function localScope()
{
    var strLocal:String = "local";
}
localScope();
trace(strLocal); // error because strLocal is not defined globally
```

如果用于局部变量的变量名已经被声明为全局变量，那么，当局部变量在作用域内时，局部定义会隐藏（或遮蔽）全局定义。全局变量在该函数外部仍然存在。例如，下面的代码创建一个名为 str1 的全局字符串变量，然后在 scopeTest() 函数内部创建一个同名的局部变量。该函数中的 trace 语句输出该变量的局部值，而函数外部的 trace 语句则输出该变量的全局值。

```
var str1:String = "Global";
function scopeTest ()
{
    var str1:String = "Local";
    trace(str1); // Local
}
scopeTest();
trace(str1); // Global
```

与 C++ 和 Java 中的变量不同的是， ActionScript 变量没有块级作用域。代码块是指左大括号 ({) 与右大括号 (}) 之间的任何语句组。在某些编程语言（如 C++ 和 Java）中，在代码块内部声明的变量在代码块外部不可用。对于作用域的这一限制称为块级作用域， ActionScript 中不存在这样的限制，如果您在某个代码块中声明一个变量，那么，该变量不仅在该代码块中可用，在该代码块所属函数的其他任何部分也都可用。例如，下面的函数包含在不同的块作用域中定义的变量。所有的变量均在整个函数中可用。

```
function blockTest (testArray:Array)
{
    var numElements:int = testArray.length;
    if (numElements > 0)
    {
        var elemStr:String = "Element #";
        for (var i:int = 0; i < numElements; i++)
        {
            var valueStr:String = i + ": " + testArray[i];
            trace(elemStr + valueStr);
        }
        trace(elemStr, valueStr, i); // all still defined
    }
    trace(elemStr, valueStr, i); // all defined if numElements > 0
}

blockTest(["Earth", "Moon", "Sun"]);
```

有趣的是，如果缺乏块级作用域，那么，只要在函数结束之前对变量进行声明，就可以在声明变量之前读写它。这是因为存在一种名为提升的方法，该方法表示编译器会将所有的变量声明移到函数的顶部。例如，下面的代码会进行编译，即使先有 num 变量的初始 trace() 函数后声明 num 变量也是如此：

```
trace(num); // NaN
var num:Number = 10;
trace(num); // 10
```

但是，编译器将不会提升任何赋值语句。这就说明了为什么 num 的初始 trace() 会生成 NaN （而非某个数字）， NaN 是 Number 数据类型变量的默认值。这意味着您甚至可以在声明变量之前为变量赋值，如下面的示例所示：

```
num = 5;
trace(num); // 5
var num:Number = 10;
trace(num); // 10
```

默认值

默认值是设置变量的值之前变量中包含的值。首次设置变量的值实际上就是初始化变量。如果您声明了一个变量，但是没有设置它的值，则该变量便处于未初始化状态。未初始化的变量的值取决于它的数据类型。下表说明了变量的默认值，并按数据类型对这些值进行组织：

数据类型	默认值
Boolean	false
int	0
Number	NaN
Object	null
String	null
uint	0
未声明 （与类型注释 * 等效）	undefined
其他所有类 （包括用户定义的类）。	null

对于 Number 类型的变量，默认值是 NaN（而非某个数字），NaN 是一个由 IEEE-754 标准定义的特殊值，它表示非数字的某个值。

如果您声明某个变量，但是未声明它的数据类型，则应用默认数据类型 *，这实际上表示该变量是无类型变量。如果您还没有用值初始化无类型变量，则该变量的默认值是 undefined。

对于 Boolean、Number、int 和 uint 以外的数据类型，所有未初始化变量的默认值都是 null。这适用于由 ActionScript 3.0 定义的所有类以及您创建的所有自定义类。

对于 Boolean、Number、int 或 uint 类型的变量，null 不是有效值。如果您尝试将值 null 赋予这样的变量，则该值会转换为该数据类型的默认值。对于 Object 类型的变量，可以赋予 null 值。如果您尝试将值 undefined 赋予 Object 类型的变量，则该值会转换为 null。

对于 Number 类型的变量，有一个名为 isNaN() 的特殊的顶级函数，如果变量不是数字，则该函数会返回布尔值 true，否则会返回 false。

数据类型

数据类型用于定义一组值。例如， Boolean 数据类型所定义的一组值中仅包含两个值：true 和 false。除了 Boolean 数据类型外，ActionScript 3.0 还定义了其他几个常用的数据类型，如 String、Number 和 Array。您可以使用类或接口来自定义一组值，从而定义自己的数据类型。 ActionScript 3.0 中的所有值均是对象，无论这些值是基元值还是复杂值。

基元值是一个属于下列数据类型之一的值：Boolean、 int、 Number、 String 和 uint。使用基元值的速度通常比使用复杂值的速度快，因为 ActionScript 按照一种尽可能优化内存和提高速度的特殊方式来存储基元值。

注：关注技术细节的读者会发现， ActionScript 在内部将基元值作为不可改变的对象进行存储。这意味着按引用传递与按值传递同样有效。这可以减少内存的使用量并提高执行速度，因为引用通常比值本身小得多。

复杂值是指基元值以外的值。定义复杂值的集合的数据类型包括：Array、 Date、 Error、 Function、 RegExp、 XML 和 XMLList。

许多编程语言都区分基元值及其包装对象。例如， Java 中有一个 int 基元值和一个包装它的 java.lang.Integer 类。 Java 基元值不是对象，但它们的包装是对象，这使得基元值对于某些运算非常有用，而包装对象则更适合于其他运算。在 ActionScript 3.0 中，出于实用的目的，不对基元值及其包装对象加以区分。所有的值 （甚至基元值）都是对象。运行时将这些基元类型视为特例，它们的行为与对象相似，但是不需要创建对象所涉及的正常开销。这意味着下面的两行代码是等效的：

```
var someInt:int = 3;
var someInt:int = new int(3);
```

上面列出的所有基元数据类型和复杂数据类型都是由 ActionScript 3.0 核心类定义的。通过 ActionScript 3.0 核心类，可以使用字面值 （而非 new 运算符）创建对象。例如，可以使用字面值或 Array 类的构造函数来创建数组，如下所示：

```
var someArray:Array = [1, 2, 3]; // literal value
var someArray:Array = new Array(1,2,3); // Array constructor
```

类型检查

类型检查可以在编译时或运行时执行。静态类型语言 （如 C++ 和 Java）在编译时执行类型检查。动态类型语言 （如 Smalltalk 和 Python）在运行时执行类型检查。作为动态类型的语言，ActionScript 3.0 在运行时执行类型检查，同时也支持在称为严格模式的特殊编译器模式下在编译时执行类型检查。在严格模式下，类型检查既发生在编译时也发生在运行时，但是在标准模式下，类型检查仅发生在运行时。

在构造代码时，动态类型的语言带来了极大的灵活性，但代价是在运行时可能出现类型错误。静态类型的语言在编译时报告类型错误，但代价是要求类型信息在编译时是已知的。

编译时类型检查

在较大的项目中通常建议使用编译时类型检查，因为随着项目变大，相对于尽早捕获类型错误，数据类型的灵活性通常会变得不那么重要。这就是为什么将 Flash Professional 和 Flash Builder 中的 ActionScript 编译器默认设置为在严格模式下运行的原因。

Adobe Flash Builder

可以通过 “ 项目属性 ” 对话框中的 ActionScript 编译器设置在 Flash Builder 中禁用严格模式。

为了提供编译时类型检查，编译器需要知道代码中的变量或表达式的数据类型信息。为了显式声明变量的数据类型，请在变量名后面添加后跟数据类型的冒号运算符 (:) 作为其后缀。要将数据类型与参数相关联，应使用后跟数据类型的冒号运算符。例如，下面的代码向 xParam 参数中添加数据类型信息，并用显式数据类型声明变量 myParam：

```
function runtimeTest(xParam:String)
{
    trace(xParam);
}
var myParam:String = "hello";
runtimeTest(myParam);
```

在严格模式下， ActionScript 编译器将类型不匹配报告为编译器错误。例如，下面的代码声明一个 Object 类型的函数参数 xParam，但是之后又尝试向该参数赋予 String 类型和 Number 类型的值。这在严格模式下会产生编译器错误。

```
function dynamicTest(xParam:Object)
{
    if (xParam is String)
    {
        var myStr:String = xParam; // compiler error in strict mode
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam; // compiler error in strict mode
        trace("Number: " + myNum);
    }
}
```

但是，即使在严格模式下，也可以选择不在赋值语句右侧指定类型，从而退出编译时类型检查。可以通过省略类型注释或使用特殊的星号 (*) 类型注释，来将变量或表达式标记为无类型。例如，如果对上一个示例中的 xParam 参数进行修改，使其不再有类型注释，则代码会在严格模式下进行编译：

```
function dynamicTest(xParam)
{
    if (xParam is String)
    {
        var myStr:String = xParam;
        trace("String: " + myStr);
    }
    else if (xParam is Number)
    {
        var myNum:Number = xParam;
        trace("Number: " + myNum);
    }
}
dynamicTest(100)
dynamicTest("one hundred");
```

运行时类型检查

在 ActionScript 3.0 中，无论是在严格模式下还是在标准模式下编译，在运行时都将进行类型检查。请考虑以下情形：将值 3 作为一个参数传递给需要数组的函数。在严格模式下，编译器将生成一个错误，因为值 3 与 Array 数据类型不兼容。如果您禁用严格模式而在标准模式下运行，则编译器不会指出类型不匹配，但在运行时进行类型检查时会产生运行时错误。

下面的示例演示一个名为 typeTest() 的函数，该函数需要一个 Array 参数，但传递的值却是 3。在标准模式下这会产生运行时错误，因为值 3 不是参数声明的数据类型 (Array) 的成员。

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum:Number = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0 standard mode
```

还可能会出现如下情形：即使在严格模式下运行，也可能会获得运行时类型错误。如果您使用严格模式，但是通过使用无类型变量而退出了编译时类型检查，就可能会出现上述情形。当您使用无类型变量时，并不会消除类型检查，而只是将其延迟到运行时执行。例如，如果上一个示例中的 myNum 变量没有已声明的数据类型，则编译器无法检测到类型不匹配，但代码会生成运行时错误，因为它会将 myNum 的运行时值 （赋值语句将其设为 3）与 xParam 的类型 （设置为 Array 数据类型）进行比较。

```
function typeTest(xParam:Array)
{
    trace(xParam);
}
var myNum = 3;
typeTest(myNum);
// run-time error in ActionScript 3.0
```

与编译时类型检查相比，运行时类型检查还允许更灵活地使用继承。标准模式会将类型检查延迟到运行时执行，从而使您可以引用子类的属性，即使您上传也是如此。当您使用基类来声明类实例的类型，但是使用子类来实例化类实例时，就会发生上传。例如，您可以创建一个名为 ClassBase 的可扩展类 （具有 final 属性的类不能扩展）：

```
class ClassBase
{
}
```

随后，您可以创建一个名为 ClassExtender 的 ClassBase 子类，该子类具有一个名为 someString 的属性，如下所示：

```
class ClassExtender extends ClassBase
{
    var someString:String;
}
```

通过使用这两个类，可以创建一个使用 ClassBase 数据类型进行声明、但使用 ClassExtender 构造函数进行实例化的类实例。上传被视为安全操作，这是因为基类不包含子类中没有的任何属性或方法。

```
var myClass:ClassBase = new ClassExtender();
```

但是，子类中则包含其基类中没有的属性或方法。例如，ClassExtender 类中包含 someString 属性，该属性在 ClassBase 类中不存在。在 ActionScript 3.0 标准模式下，可以使用 myClass 实例来引用此属性，而不会生成编译时错误，如下面的示例所示：

```
var myClass:ClassBase = new ClassExtender();
myClass.someString = "hello";
// no error in ActionScript 3.0 standard mode
```

is 运算符

is 运算符可用来测试变量或表达式是否为给定数据类型的成员。在早期的 ActionScript 版本中，此功能由 instanceof 运算符提供。但在 ActionScript 3.0 中，不应使用 instanceof 运算符来测试变量或表达式是否为数据类型的成员。对于手动类型检查，应用 is 运算符来代替 instanceof 运算符，因为表达式 x instanceof y 只是在 x 的原型链中检查 y 是否存在（在 ActionScript 3.0 中，原型链不能全面地描述继承层次）。

is 运算符检查正确的继承层次，不但可以用来检查对象是否为特定类的实例，而且还可以检查对象是否是用来实现特定接口的类的实例。下面的示例创建 Sprite 类的一个名为 mySprite 的实例，并使用 is 运算符来测试 mySprite 是否为 Sprite 和 DisplayObject 类的实例，以及它是否实现 IEventDispatcher 接口：

```
var mySprite:Sprite = new Sprite();
trace(mySprite is Sprite); // true
trace(mySprite is DisplayObject);// true
trace(mySprite is IEventDispatcher); // true
```

is 运算符检查继承层次，并正确地报告 mySprite 与 Sprite 和 DisplayObject 类兼容（Sprite 类是 DisplayObject 类的子类）。 is 运算符还检查 mySprite 是否是从实现 IEventDispatcher 接口的任意类继承的。由于 Sprite 类是从实现 IEventDispatcher 接口的 EventDispatcher 类继承的，因此 is 运算符会正确地报告 mySprite 也实现该接口。

下面的示例说明与上一个示例相同的测试，但使用的是 instanceof 运算符，而不是 is 运算符。 instanceof 运算符正确地识别出 mySprite 是 Sprite 或 DisplayObject 的实例，但是，当该运算符用于测试 mySprite 是否实现 IEventDispatcher 接口时，返回的却是 false。

```
trace(mySprite instanceof Sprite); // true
trace(mySprite instanceof DisplayObject);// true
trace(mySprite instanceof IEventDispatcher); // false
```

as 运算符

as 运算符也可用来检查表达式是否为给定数据类型的成员。但是，与 is 运算符不同的是，as 运算符不返回布尔值。as 运算符返回表达式的值 （代替 true）或 null （代替 false）。下面的示例演示在检查 Sprite 实例是否为 DisplayObject、IEventDispatcher 和 Number 数据类型的成员这样的简单情况下，使用 as 运算符替代 is 运算符的结果。

```
var mySprite:Sprite = new Sprite();
trace(mySprite as Sprite); // [object Sprite]
trace(mySprite as DisplayObject); // [object Sprite]
trace(mySprite as IEventDispatcher); // [object Sprite]
trace(mySprite as Number);           // null
```

在使用 as 运算符时，右侧的操作数必须是数据类型。如果尝试使用表达式 （而非数据类型）作为右侧的操作数，将会产生错误。

动态类

动态类定义在运行时可通过添加或更改属性和方法来改变的对象。非动态类 （如 String 类）是密封类。您不能在运行时向密封类中添加属性或方法。

在声明类时，可以通过使用 dynamic 属性来创建动态类。例如，下面的代码创建一个名为 Protean 的动态类：

```
dynamic class Protean
{
    private var privateGreeting:String = "hi";
    public var publicGreeting:String = "hello";
    function Protean()
    {
        trace("Protean instance created");
    }
}
```

如果要在以后实例化 Protean 类的实例，可以在类定义的外部向其添加属性或方法。例如，下面的代码创建 Protean 类的一个实例，并向该实例中添加两个名称分别为 aString 和 aNumber 的属性：

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
trace(myProtean.aString, myProtean.aNumber); // testing 3
```

添加到动态类实例中的属性是运行时实体，因此会在运行时完成所有类型检查。不能向以这种方式添加的属性中添加类型注释。

您还可以定义一个函数并将该函数附加到 myProtean 实例的某个属性，从而向 myProtean 实例中添加方法。下面的代码将 trace 语句移到一个名为 traceProtean() 的方法中：

```
var myProtean:Protean = new Protean();
myProtean.aString = "testing";
myProtean.aNumber = 3;
myProtean.traceProtean = function ()
{
    trace(this.aString, this.aNumber);
};
myProtean.traceProtean(); // testing 3
```

但是，以这种方式创建的方法对于 Protean 类的任何私有属性或方法都不具有访问权限。而且，即使对 Protean 类的公共属性或方法的引用也必须用 this 关键字或类名进行限定。下面的示例说明了 traceProtean() 方法，该方法尝试访问 Protean 类的私有变量和公共变量。

```
myProtean.traceProtean = function ()
{
    trace(myProtean.privateGreeting); // undefined
    trace(myProtean.publicGreeting); // hello
};
myProtean.traceProtean();
```

数据类型说明

基元数据类型包括 Boolean、int、Null、Number、String、uint 和 void。ActionScript 核心类还定义下列复杂数据类型：Object、 Array、 Date、 Error、 Function、 RegExp、 XML 和 XMLList。

Boolean 数据类型

Boolean 数据类型包含两个值：true 和 false。对于 Boolean 类型的变量，其他任何值都是无效的。已经声明但尚未初始化的布尔变量的默认值是 false。

int 数据类型

int 数据类型在内部存储为 32 位整数，包含的整数集从

-2,147,483,648 (-2^{31}) 至 2,147,483,647 ($2^{31} - 1$) 之间 （两端包含在内）。早期的 ActionScript 版本仅提供 Number 数据类型，该数据类型既可用于整数又可用于浮点数。在 ActionScript 3.0 中，现在可以访问 32 位带符号整数和无符号整数的低位机器类型。如果您的变量不需要浮点数，那么，使用 int 数据类型来代替 Number 数据类型会更快更高效。

对于小于 int 的最小值或大于 int 的最大值的整数值，应使用 Number 数据类型。 Number 数据类型可以处理 -9,007,199,254,740,992 和 9,007,199,254,740,992 （53 位整数值）之间的值。 int 数据类型的变量的默认值是 0。

Null 数据类型

Null 数据类型仅包含一个值：null。这是 String 数据类型和用来定义复杂数据类型的所有类 （包括 Object 类）的默认值。其他基元数据类型 （如 Boolean、 Number、 int 和 uint）均不包含 null 值。在运行时，如果尝试将 null 分配给 Boolean、Number、 int 或 uint 类型的变量，则值 null 会转换为相应的默认值。不能将 Null 数据类型用作类型注释。

Number 数据类型

在 ActionScript 3.0 中， Number 数据类型可以表示整数、无符号整数和浮点数。但是，为了尽可能提高性能，应将 Number 数据类型仅用于浮点数，或者用于 int 和 uint 类型可以存储的、大于 32 位的整数值。要存储浮点数，数字中应包括一个小数点。如果您省略小数点，数字将存储为整数。

Number 数据类型使用由 IEEE 二进制浮点算术标准 (IEEE-754) 指定的 64 位双精度格式。此标准规定如何使用 64 个可用位来存储浮点数。其中的 1 位用来指定数字是正数还是负数。 11 位用于指数，以 2 为底进行存储。其余的 52 位用于存储有效位数 （又称为尾数），有效位数是 2 的 N 次幂， N 即前面所提到的指数。

可以将 Number 数据类型的所有位都用于有效位数，也可以将 Number 数据类型的某些位用于存储指数，后者可存储的浮点数比前者大得多。例如，如果 Number 数据类型使用全部 64 位存储有效位数，则可以存储的最大数字为 $2^{65} - 1$。如果使用 11 位存储指数，则 Number 数据类型可以存储的最大有效数字为 2^{1023}。

Number 类型可以表示的最大值和最小值存储在 Number 类的名为 Number.MAX_VALUE 和 Number.MIN_VALUE 的静态属性中。

```
Number.MAX_VALUE == 1.79769313486231e+308
Number.MIN_VALUE == 4.940656458412467e-324
```

尽管这个数字范围很大，但代价是此范围的精度有所降低。 Number 数据类型使用 52 位来存储有效位数，因此，那些要求用 52 位以上的位数才能精确表示的数字 （如分数 1/3）将只是近似值。如果应用程序要求小数达到绝对精度，则需要使用实现小数浮点算术 （而非二进制浮点算术）的软件。

如果用 Number 数据类型来存储整数值，则仅使用 52 位有效位数。 Number 数据类型使用这 52 位和一个特殊的隐藏位来表示从 -9,007,199,254,740,992 (-2^{53}) 到 9,007,199,254,740,992 (2^{53}) 的整数。

NaN 值不仅用作 Number 类型的变量的默认值，还用作应返回数字却没有返回数字的任何运算的结果。例如，如果您尝试计算负数的平方根，结果会是 NaN。其他特殊的 Number 值包括正无穷大和负无穷大。

注：在被 0 除时，如果被除数也是 0，则结果只有一个，那就是 NaN。在被 0 除时，如果被除数是正数，则结果为 “ 正无穷大 ”；如果被除数是负数，则结果为 “ 负无穷大 ”。

String 数据类型

String 数据类型表示一个 16 位字符的序列。字符串在内部存储为 Unicode 字符，并使用 UTF-16 格式。字符串是不可改变的值，就像在 Java 编程语言中一样。对字符串值的操作返回字符串的一个新的实例。用字符串数据类型声明的变量的默认值是 null。值 null 与空字符串 ("") 不同。值 null 表示变量中未存储任何值，而空字符串表示变量中有一个 String 值，但其中不包含任何字符。

uint 数据类型

uint 数据类型在内部存储为 32 位无符号整数，包含的整数集介于 0 和 4,294,967,295 (2^{32} - 1) 之间 （包括 0 和 4,294,967,295）。uint 数据类型可用于要求非负整数的特殊情形。例如，必须使用 uint 数据类型来表示像素颜色值，因为 int 数据类型有一个内部符号位，该符号位并不适合处理颜色值。对于大于 uint 的最大值的整数值，应使用 Number 数据类型，该数据类型可以处理 53 位整数值。 uint 数据类型的变量的默认值是 0。

void 数据类型

void 数据类型仅包含一个值：undefined。在早期的 ActionScript 版本中， undefined 是 Object 类实例的默认值。在 ActionScript 3.0 中，Object 实例的默认值是 null。如果您尝试将值 undefined 赋予 Object 类的实例，则该值会转换为 null。您只能为无类型变量赋予 undefined 这一值。无类型变量是指缺乏类型注释或者使用星号 (*) 作为类型注释的变量。您可以将 void 只用作返回类型注释。

Object 数据类型

Object 数据类型是由 Object 类定义的。Object 类用作 ActionScript 中的所有类定义的基类。ActionScript 3.0 中的 Object 数据类型与早期版本中的 Object 数据类型存在以下三方面的区别：第一， Object 数据类型不再是指定给没有类型注释的变量的默认数据类型。第二， Object 数据类型不再包括 undefined 这一值，该值以前是 Object 实例的默认值。第三，在 ActionScript 3.0 中， Object 类实例的默认值是 null。

在早期的 ActionScript 版本中，会自动为没有类型注释的变量赋予 Object 数据类型。ActionScript 3.0 现在包括真正无类型变量这一概念，因此不再为没有类型注释的变量赋予 Object 数据类型。没有类型注释的变量现在被视为无类型变量。如果您希望向代码的读者清楚地表明您是故意将变量保留为无类型，可以使用星号 (*) 表示类型注释，这与省略类型注释等效。下面的示例演示两条等效的语句，两者都声明一个无类型变量 x：

```
var x
var x:*
```

只有无类型变量才能保存值 undefined。如果您尝试将值 undefined 赋给具有数据类型的变量，则运行时会将值 undefined 转换为该数据类型的默认值。对于 Object 数据类型的实例，默认值为 null，这表示如果您尝试将 undefined 分配给 Object 实例，该值会转换为 null。

类型转换

在将某个值转换为其他数据类型的值时，就说发生了类型转换。类型转换可以是隐式的，也可以是显式的。隐式转换也称为强制转换，有时在运行时执行。例如，如果将值 2 赋给 Boolean 数据类型的变量，则值 2 会先转换为布尔值 true，然后再将其赋给该变量。显式转换又称为转换，在代码指示编译器将一个数据类型的变量视为属于另一个数据类型时发生。在涉及基元值时，转换功能将一个数据类型的值实际转换为另一个数据类型的值。要将对象转换为另一类型，请用小括号括起对象名并在它前面加上新类型的名称。例如，下面的代码提取一个布尔值并将它转换为一个整数：

```
var myBoolean:Boolean = true;
var myINT:int = int(myBoolean);
trace(myINT); // 1
```

隐式转换

在运行时，隐式转换会发生在许多上下文中：

- 在赋值语句中
- 在将值作为函数的参数传递时
- 在从函数中返回值时
- 在使用某些运算符 （如加法运算符 (+)）的表达式中

  对于用户定义的类型，当要转换的值是目标类 （或者派生自目标类的类）的实例时，隐式转换会成功。如果隐式转换不成功，就会出现错误。例如，下面的代码中包含成功的隐式转换和不成功的隐式转换：

```
class A {}
class B extends A {}

var objA:A = new A();
var objB:B = new B();
var arr:Array = new Array();

objA = objB; // Conversion succeeds.
objB = arr; // Conversion fails.
```

对于基元类型而言，隐式转换是通过调用内部转换算法来处理的，该算法与显式转换函数所调用的算法相同。

显式转换

在严格模式下进行编译时，使用显式转换会非常有用，因为您有时可能不希望因类型不匹配而生成编译时错误。当您知道强制功能会在运行时正确转换您的值时，可能就属于这种情况。例如，在使用从表单接收的数据时，您可能希望依赖强制功能将某些字符串值转换为数值。下面的代码会生成编译时错误，即使代码在标准模式下能够正确运行也是如此：

```
var quantityField:String = "3";
var quantity:int = quantityField; // compile time error in strict mode
```

如果您希望继续使用严格模式，但是希望将字符串转换为整数，则可以使用显式转换，如下所示：

```
var quantityField:String = "3";
var quantity:int = int(quantityField); // Explicit conversion succeeds.
```

转换为 int、 uint 和 Number

您可以将任何数据类型转换为以下三种数字类型之一：int、 uint 和 Number。如果由于某些原因不能转换数值，则会将默认值 0 赋给 int 和 uint 数据类型，将默认值 NaN 赋给 Number 数据类型。如果将布尔值转换为数字，则 true 变成值 1，false 变成值 0。

```
var myBoolean:Boolean = true;
var myUINT:uint = uint(myBoolean);
var myINT:int = int(myBoolean);
var myNum:Number = Number(myBoolean);
trace(myUINT, myINT, myNum); // 1 1 1
myBoolean = false;
myUINT = uint(myBoolean);
myINT = int(myBoolean);
myNum = Number(myBoolean);
trace(myUINT, myINT, myNum); // 0 0 0
```

仅包含数字的字符串值可以成功地转换为数字类型之一。看上去像负数的字符串或者表示十六进制值的字符串 （例如， 0x1A）也可以转换为数字类型。转换过程中会忽略字符串值中的前导或尾随空白字符。还可以使用 Number() 来转换看上去像浮点数的字符串。如果包含小数点，则会导致 uint() 和 int() 返回一个整数，小数点和它后面的字符被截断。例如，下面的字符串值可以转换为数字：

```
trace(uint("5")); // 5
trace(uint("-5")); // 4294967291. It wraps around from MAX_VALUE
trace(uint(" 27 ")); // 27
trace(uint("3.7")); // 3
trace(int("3.7")); // 3
trace(int("0x1A")); // 26
trace(Number("3.7")); // 3.7
```

对于包含非数字字符的字符串值，在用 int() 或 uint() 转换时，将返回 0 ；在用 Number() 转换时，将返回 NaN。转换过程中会忽略前导和尾随空白，但是，如果字符串中包含将两个数字隔开的空白，则将返回 0 或 NaN。

```
trace(uint("5a")); // 0
trace(uint("ten")); // 0
trace(uint("17 63")); // 0
```

在 ActionScript 3.0 中，Number() 函数不再支持八进制数或基数为 8 的数。如果您向 ActionScript 2.0 Number() 函数提供的字符串中包含前导 0，则该数字将被视为八进制数，并将转换为等效的十进制数。对于 ActionScript 3.0 中的 Number() 函数，情况并非如此，因为 ActionScript 3.0 会忽略前导 0。例如，在使用不同的 ActionScript 版本进行编译时，下面的代码会生成不同的输出结果：

```
trace(Number("044"));
// ActionScript 3.0 44
// ActionScript 2.0 36
```

将一种数值类型的值赋给另一种数值类型的变量时，转换并不是必需的。即使在严格模式下，数值类型也会隐式转换为其他数值类型。这意味着，在某些情况下，在超出类型的范围时，可能会生成意外的值。下面的几个示例全部是在严格模式下进行编译的，但是某些示例会生成意外的值：

```
var myUInt:uint = -3; // Assign int/Number value to uint variable
trace(myUInt); // 4294967293

var myNum:Number = sampleUINT; // Assign int/uint value to Number variable
trace(myNum) // 4294967293

var myInt:int = uint.MAX_VALUE + 1; // Assign Number value to uint variable
trace(myInt); // 0

myInt = int.MAX_VALUE + 1; // Assign uint/Number value to int variable
trace(myInt); // -2147483648
```

下表概述了将其他数据类型转换为 Number、 int 或 uint 数据类型的结果。

数据类型或值	转换为 Number、 int 或 uint 时的结果
Boolean	如果值为 true，则结果为 1 ；否则为 0。
Date	Date 对象的内部表示形式，即从 1970 年 1 月 1 日午夜 （通用时间）以来所经过的毫秒数。
null	0
Object	如果实例为 null 并转换为 Number，则结果为 NaN ；否则为 0。
String	如果字符串可以转换为某个数字，则为数字，否则，如果转换为 Number，则为 NaN ；如果转换为 int 或 uint，则为 0。
undefined	如果转换为 Number，则结果为 NaN ；如果转换为 int 或 uint，则结果为 0。

转换为 **Boolean**

在从任何数值数据类型 （uint、 int 和 Number）转换为 Boolean 时，如果数值为 0，则结果为 false ；否则为 true。对于 Number 数据类型，如果值为 NaN，则结果也为 false。下面的示例说明在转换 -1、 0 和 1 等数字时的结果：

```
var myNum:Number;
for (myNum = -1; myNum<2; myNum++)
{
    trace("Boolean(" + myNum +") is " + Boolean(myNum));
}
```

本示例的输出结果演示在这三个数字中，只有 0 返回 false 值：

```
Boolean(-1) is true
Boolean(0) is false
Boolean(1) is true
```

在将字符串值转换为 Boolean 数据类型时，如果字符串为 null 或空字符串 (""), 则会返回 false。否则，将返回 true。

```
var str1:String; // Uninitialized string is null.
trace(Boolean(str1)); // false

var str2:String = ""; // empty string
trace(Boolean(str2)); // false

var str3:String = " "; // white space only
trace(Boolean(str3)); // true
```

在将 Object 类的实例转换为 Boolean 数据类型时，如果该实例为 null，则将返回 false；否则将返回 true：

```
var myObj:Object; // Uninitialized object is null.
trace(Boolean(myObj)); // false

myObj = new Object(); // instantiate
trace(Boolean(myObj)); // true
```

在严格模式下，系统会对布尔变量进行特殊处理，因为您不必转换即可向布尔变量赋予任何数据类型的值。即使在严格模式下，也可以将所有的数据类型隐式强制为 Boolean 数据类型。换言之，与几乎其他所有数据类型不同，转换为 Boolean 数据类型不是避免在严格模式下出错所必需的。下面的几个示例全部是在严格模式下编译的，它们在运行时按照预期的方式工作：

```
var myObj:Object = new Object(); // instantiate
var bool:Boolean = myObj;
trace(bool); // true
bool = "random string";
trace(bool); // true
bool = new Array();
trace(bool); // true
bool = NaN;
trace(bool); // false
```

下表概述了在从其他数据类型转换为 Boolean 数据类型时的结果：

数据类型或值	转换为 Boolean 数据类型时的结果
String	如果值为 null 或空字符串 ("")，则结果为 false；否则为 true。
null	false
Number、int 或 uint	如果值为 NaN 或 0，则结果为 false；否则为 true。
Object	如果实例为 null，则结果为 false；否则为 true。

转换为 **String**

从任何数值数据类型转换为 String 数据类型时，都会返回数字的字符串表示形式。在将布尔值转换为 String 数据类型时，如果值为 true，则返回字符串 "true"；如果值为 false，则返回字符串 "false"。

在从 Object 类的实例转换为 String 数据类型时，如果该实例为 null，则返回字符串 "null"。否则，从 Object 类转换为 String 类型会返回字符串 "[object Object]"。

从 Array 类的实例转换为 String 会返回一个字符串，其中包含所有数组元素的逗号分隔列表。例如，在下面的示例中，在转换为 String 数据类型时，将返回一个包含数组中的全部三个元素的字符串：

```
var myArray:Array = ["primary", "secondary", "tertiary"];
trace(String(myArray)); // primary,secondary,tertiary
```

在从 Date 类的实例转换为 String 时，会返回该实例所包含日期的字符串表示形式。例如，下面的示例返回 Date 类实例的字符串表示形式 （输出结果显示的是太平洋夏令时）：

```
var myDate:Date = new Date(2005,6,1);
trace(String(myDate)); // Fri Jul 1 00:00:00 GMT-0700 2005
```

下表概述了在将其他数据类型转换为 String 数据类型时的结果：

数据类型或值	转换为 String 数据类型时的结果
Array	由所有数组元素组成的字符串。
Boolean	“true” 或 “false”
Date	Date 对象的字符串表示形式。
null	"null"
Number、 int 或 uint	数字的字符串表示形式。
Object	如果实例为 null，则结果为 "null" ；否则为 "[object Object]"。

语法

语言的语法定义了一组在编写可执行代码时必须遵循的规则。

区分大小写

ActionScript 3.0 是一种区分大小写的语言。只是大小写不同的标识符会被视为不同。例如，下面的代码创建两个不同的变量：

```
var num1:int;
var Num1:int;
```

点语法

点运算符 (.) 提供对对象的属性和方法的访问。使用点语法，可以使用后跟点运算符和属性名或方法名的实例名来引用类的属性或方法。以下面的类定义为例：

```
class DotExample
{
    public var prop1:String;
    public function method1():void {}
}
```

借助于点语法，可以使用在如下代码中创建的实例名来访问 prop1 属性和 method1() 方法：

```
var myDotEx:DotExample = new DotExample();
myDotEx.prop1 = "hello";
myDotEx.method1();
```

定义包时，可以使用点语法。可以使用点运算符来引用嵌套包。例如， EventDispatcher 类位于一个名为 events 的包中，该包嵌套在名为 flash 的包中。可以使用下面的表达式来引用 events 包：

```
flash.events
```

还可以使用此表达式来引用 EventDispatcher 类：

```
flash.events.EventDispatcher
```

斜杠语法

ActionScript 3.0 不支持斜杠语法。在早期的 ActionScript 版本中，斜杠语法用于指示影片剪辑或变量的路径。

字面值

文本是直接出现在代码中的值。下面的例子都是文本：

```
17
"hello"
-3
9.4
null
undefined
true
false
```

字面值还可以组合起来构成复合字面值。数组文本括在中括号字符 ([]) 中，各数组元素之间用逗号隔开。

数组文本可用于初始化一个数组。下例显示了使用数组文本进行初始化的两个数组。您可以使用 new 语句将复合字面值作为参数传递给 Array 类构造函数，但是，您也可以在实例化下面的 ActionScript 核心类的实例时直接赋予字面值：Object、Array、 String、 Number、 int、 uint、 XML、 XMLList 和 Boolean。

```
// Use new statement.
var myStrings:Array = new Array(["alpha", "beta", "gamma"]);
var myNums:Array = new Array([1,2,3,5,8]);

// Assign literal directly.
var myStrings:Array = ["alpha", "beta", "gamma"];
var myNums:Array = [1,2,3,5,8];
```

字面值还可用来初始化通用对象。通用对象是对象类的一个实例。Object 字面值括在大括号 ({}) 中，各对象属性之间用逗号隔开。每个属性都用冒号字符 (:) 进行声明，冒号用于分隔属性名和属性值。

可以使用 new 语句创建一个通用对象并将该对象的字面值作为参数传递给 Object 类构造函数，也可以在声明实例时直接将对象字面值赋给实例。下面的示例演示两个替代方法，用于创建一个新的通用对象，并使用三个值分别设置为 1、 2 和 3 的属性（propA、 propB 和 propC）初始化该对象：

```
// Use new statement and add properties.
var myObject:Object = new Object();
myObject.propA = 1;
myObject.propB = 2;
myObject.propC = 3;

// Assign literal directly.
var myObject:Object = {propA:1, propB:2, propC:3};
```

更多帮助主题

使用字符串

使用正则表达式

初始化 XML 变量

分号

可以使用分号字符 (;) 来终止语句。如果您省略分号字符，则编译器假设一行代码代表一条语句。由于很多程序员都习惯使用分号来表示语句结束，因此，如果您坚持使用分号来终止语句，则代码会更易于阅读。

使用分号终止语句可以在一行中放置多个语句，但是这样会使代码变得难以阅读。

小括号

在 ActionScript 3.0 中，可以通过三种方式使用小括号 (())。第一，可以使用小括号来更改表达式中的运算顺序。组合到小括号中的运算总是最先执行。例如，小括号可用来改变如下代码中的运算顺序：

```
trace(2 + 3 * 4); // 14
trace((2 + 3) * 4); // 20
```

第二，可以结合使用小括号和逗号运算符 (,) 来计算一系列表达式并返回最后一个表达式的结果，如下面的示例所示：

```
var a:int = 2;
var b:int = 3;
trace((a++, b++, a+b)); // 7
```

第三，可以使用小括号来向函数或方法传递一个或多个参数，如下面的示例所示，此示例向 trace() 函数传递一个字符串值：

```
trace("hello"); // hello
```

注释

ActionScript 3.0 代码支持两种类型的注释：单行注释和多行注释。这些注释机制与 C++ 和 Java 中的注释机制类似。编译器将忽略标记为注释的文本。

单行注释以两个正斜杠字符 (//) 开头并持续到该行的末尾。例如，下面的代码包含一个单行注释：

```
var someNumber:Number = 3; // a single line comment
```

多行注释以一个正斜杠和一个星号 (/*) 开头，以一个星号和一个正斜杠 (*/) 结尾。

```
/* This is multiline comment that can span
more than one line of code. */
```

关键字和保留字

保留字是一些单词，因为这些单词是保留给 ActionScript 使用的，所以不能在代码中将它们用作标识符。保留字包括词汇关键字，编译器将词汇关键字从程序的命名空间中移除。如果您将词汇关键字用作标识符，则编译器会报告错误。下表列出了 ActionScript 3.0 词汇关键字：

as	break	case	catch
class	const	continue	default
delete	do	else	extends
false	finally	for	function
if	implements	import	in
instanceof	interface	internal	is
native	new	null	package
private	protected	public	return
super	switch	this	throw
to	true	try	typeof
use	var	void	while
with			

有一小组称为句法关键字的关键字，这些关键字可用作标识符，但是在某些上下文中具有特殊的含义。下表列出了 ActionScript 3.0 句法关键字：

each	get	set	namespace
include	dynamic	final	native
override	static		

还有几个有时称为供将来使用的保留字的标识符。这些标识符不是为 ActionScript 3.0 保留的，但是其中的一些可能会被采用 ActionScript 3.0 的软件视为关键字。可以在您的代码中使用其中的许多标识符，但是 Adobe 建议您不要使用这些标识符，因为它们可能在未来版本的语言中作为关键字。

abstract	boolean	byte	cast
char	debugger	double	enum
export	float	goto	intrinsic
long	prototype	short	synchronized
throws	to	transient	type
virtual	volatile		

常量

ActionScript 3.0 支持 const 语句，该语句可用来创建常量。常量是指具有无法改变的固定值的属性。只能为常量赋值一次，而且必须在最接近常量声明的位置赋值。例如，如果将常量声明为类的成员，则只能在声明过程中或者在类构造函数中为常量赋值。

下面的代码声明两个常量。第一个常量 MINIMUM 是在声明语句中赋值的。第二个常量 MAXIMUM 是在构造函数中赋值的。请注意，此示例仅在标准模式下进行编译，因为严格模式只允许在初始化时对常量进行赋值。

```
class A
{
    public const MINIMUM:int = 0;
    public const MAXIMUM:int;

    public function A()
    {
        MAXIMUM = 10;
    }
}

var a:A = new A();
trace(a.MINIMUM); // 0
trace(a.MAXIMUM); // 10
```

如果您尝试以其他任何方法向常量赋予初始值，则会出现错误。例如，如果您尝试在类的外部设置 MAXIMUM 的初始值，则会出现运行时错误。

```
class A
{
public const MINIMUM:int = 0;
public const MAXIMUM:int;
}

var a:A = new A();
a["MAXIMUM"] = 10; // run-time error
```

ActionScript 3.0 定义了各种各样的常量供您使用。按照惯例，ActionScript 中的常量全部使用大写字母，各个单词之间用下划线字符 (_) 分隔。例如， MouseEvent 类定义将此命名惯例用于其常量，其中每个常量都表示一个与鼠标输入有关的事件：

```
package flash.events
{
    public class MouseEvent extends Event
    {
    public static const CLICK:String = "click";
    public static const DOUBLE_CLICK:String = "doubleClick";
    public static const MOUSE_DOWN:String = "mouseDown";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...
    }
}
```

运算符

运算符是一种特殊的函数，它们具有一个或多个操作数并返回相应的值。操作数是运算符用作输入的值 （通常为字面值、变量或表达式）。例如，在下面的代码中，将加法运算符 (+) 和乘法运算符 (*) 与三个字面值操作数 （2、 3 和 4）结合使用来返回一个值。赋值运算符 (=) 随后使用此值将返回值 14 赋给变量 sumNumber。

```
var sumNumber:uint = 2 + 3 * 4; // uint = 14
```

运算符可以是一元、二元或三元的。一元运算符采用 1 个操作数。例如，递增运算符 (++) 就是一元运算符，因为它只有一个操作数。二元运算符采用 2 个操作数。例如，除法运算符 (/) 有 2 个操作数。三元运算符采用 3 个操作数。例如，条件运算符 (?:) 采用 3 个操作数。

有些运算符是重载的，这意味着其行为因传递给它们的操作数的类型或数量而异。例如，加法运算符 (+) 就是一个重载运算符，其行为因操作数的数据类型而异。如果两个操作数都是数字，则加法运算符会返回这些值的和。如果两个操作数都是字符串，则加法运算符会返回这两个操作数连接后的结果。下面的示例代码说明运算符的行为如何因操作数而异：

```
trace(5 + 5); // 10
trace("5" + "5"); // 55
```

运算符的行为还可能因所提供的操作数的数量而异。减法运算符 (-) 既是一元运算符又是二元运算符。对于减法运算符，如果只提供一个操作数，则该运算符会对操作数求反并返回结果；如果提供两个操作数，则减法运算符返回这两个操作数的差。下面的示例说明首先将减法运算符用作一元运算符，然后再将其用作二元运算符。

```
trace(-3); // -3
trace(7 - 2); // 5
```

运算符的优先级和结合律

运算符的优先级和结合律决定了处理运算符的顺序。虽然对于熟悉算术的人来说，编译器先处理乘法运算符 (*) 后处理加法运算符 (+) 似乎是自然而然的事情，但实际上编译器要求显式指定先处理哪些运算符。此类指令统称为运算符优先级。ActionScript 定义了一个默认的运算符优先级，可以使用小括号运算符 (()) 来改变它。例如，下面的代码改变上一个示例中的默认优先级，以强制编译器先处理加法运算符，然后再处理乘法运算符：

```
var sumNumber:uint = (2 + 3) * 4; // uint == 20
```

您可能会遇到这样的情况：同一个表达式中出现两个或更多个具有相同的优先级的运算符。在这些情况下，编译器使用结合律的规则确定首先处理哪个运算符。除了赋值运算符之外，所有二进制运算符中都是左结合的，也就是说，先处理左边的运算符然后再处理右边的运算符。赋值运算符和条件运算符 (?:) 都是右结合的，也就是说先处理右边的运算符然后再处理左边的运算符。

例如，考虑小于运算符 (<) 和大于运算符 (>)，这两个运算符具有相同的优先级。如果将这两个运算符用于同一个表达式中，因为这两个运算符都是左结合的，所以首先处理左边的运算符。也就是说，以下两个语句将生成相同的输出结果：

```
trace(3 > 2 < 1); // false
trace((3 > 2) < 1); // false
```

大于运算符会首先处理，这会生成值 true，因为操作数 3 大于操作数 2。随后，将值 true 与操作数 1 一起传递给小于运算符。下面的代码表示此中间状态：

```
trace((true) < 1);
```

小于运算符将值 true 转换为数值 1，然后将该数值与第二个操作数 1 进行比较，这会返回值 false （因为值 1 不小于 1）。

```
trace(1 < 1); // false
```

您可以用括号运算符来改变默认的左结合律。您可以通过用小括号括起小于运算符及其操作数来命令编译器先处理小于运算符。下面的示例使用与上一个示例相同的数，但是因为使用了小括号运算符，所以生成不同的输出结果：

```
trace(3 > (2 < 1)); // true
```

小于运算符会首先处理，这会生成值 false，因为操作数 2 不小于操作数 1。值 false 随后将与操作数 3 一起传递给大于运算符。下面的代码表示此中间状态：

```
trace(3 > (false));
```

大于运算符将值 false 转换为数值 0，然后将该数值与另一个操作数 3 进行比较，这会返回 true （因为 3 大于 0）。

```
trace(3 > 0); // true
```

下表按优先级递减的顺序列出了 ActionScript 3.0 中的运算符。在该表中，每一行中包含的运算符优先级相同。表中每一行运算符的优先级都高于出现在它下面的行中的运算符。

组合	运算符
主要	[] {x:y} () f(x) new x.y x[y] <></> @ :: ..
后缀	x++ x--
一元	++x --x + - ~ ! 删除 typeof 无效
乘性	* / %
加性	+ -
按位移位	<< >> >>>
关系	< > <= >= 如在 instanceof 中为
等于	== != === !==
按位 AND	&
按位 XOR	^
按位 OR	|
逻辑 AND	&&
逻辑 OR	||
条件	?:
赋值	= *= /= %= += -= <<= >>= >>>= &= ^= |=
逗号	,

主要运算符

主要运算符包括那些用来创建 Array 和 Object 字面值、对表达式进行分组、调用函数、实例化类实例以及访问属性的运算符。

下表列出了所有主要运算符，它们具有相同的优先级。属于 E4X 规范的运算符用 (E4X) 来表示。

运算符	执行的运算
[]	初始化数组
{x:y}	初始化对象
()	对表达式进行分组
f(x)	调用函数
new	调用构造函数
x.y x[y]	访问属性
<></>	初始化 XMLList 对象 (E4X)
@	访问属性 (E4X)
::	限定名称 (E4X)
..	访问子级 XML 元素 (E4X)

后缀运算符

后缀运算符只有一个操作数，它递增或递减该操作数的值。虽然这些运算符是一元运算符，但是它们有别于其他一元运算符，被单独划归到了一个类别，因为它们具有更高的优先级和特殊的行为。在将后缀运算符用作较长表达式的一部分时，先返回表达式的值后处理后缀运算符。例如，下面的代码演示了如何在值递增之前返回表达式 xNum++ 的值。

```
var xNum:Number = 0;
trace(xNum++); // 0
trace(xNum); // 1
```

下表列出了所有的后缀运算符，它们具有相同的优先级：

运算符	执行的运算
++	递增 （后缀）
--	递减 （后缀）

一元运算符

一元运算符只有一个操作数。这一组中的递增运算符 (++) 和递减运算符 (--) 是前缀运算符，这意味着它们在表达式中出现在操作数的前面。前缀运算符与它们对应的后缀运算符不同，因为递增或递减操作是在返回整个表达式的值之前完成的。例如，下面的代码演示如何在递增值之后返回表达式 ++xNum 的值：

```
var xNum:Number = 0;
trace(++xNum); // 1
trace(xNum); // 1
```

下表列出了所有的一元运算符，它们具有相同的优先级：

运算符	执行的运算
++	递增 （前缀）
--	递减 （前缀）
+	一元 +
-	一元 - （非）

运算符	执行的运算
!	逻辑 NOT
~	按位 NOT
delete	删除属性
typeof	返回类型信息
void	返回未定义值

乘性运算符

乘性运算符对两个操作数执行乘、除或求模计算。

下表列出了所有的乘性运算符，它们具有相同的优先级：

运算符	执行的运算
*	乘法
/	除法
%	求模

加性运算符

加性运算符对两个操作数执行加法或减法计算。下表列出了所有加性运算符，它们具有相同的优先级：

运算符	执行的运算
+	加法
-	减法

按位移位运算符

按位移位运算符涉及两个操作数，它将第一个操作数的各位按第二个操作数指定的位数移位。下表列出了所有按位移位运算符，它们具有相同的优先级：

运算符	执行的运算
<<	按位左移位
>>	按位右移位
>>>	按位无符号右移位

关系运算符

关系运算符涉及两个操作数，它比较两个操作数的值，然后返回一个布尔值。下表列出了所有关系运算符，它们具有相同的优先级：

运算符	执行的运算
<	小于
>	大于
<=	小于或等于
>=	大于或等于
as	检查数据类型
in	检查对象属性
instanceof	检查原型链
is	检查数据类型

等于运算符

等于运算符涉及两个操作数，它比较两个操作数的值，然后返回一个布尔值。下表列出了所有等于运算符，它们具有相同的优先级：

运算符	执行的运算
==	等于
!=	不等于
===	全等
!==	不全等

按位逻辑运算符

按位逻辑运算符涉及两个操作数，执行位级别的逻辑运算。按位逻辑运算符具有不同的优先级；下表按优先级递减的顺序列出了按位逻辑运算符：

运算符	执行的运算
&	按位 AND
^	按位 XOR
|	按位 OR

逻辑运算符

逻辑运算符涉及两个操作数，这类运算符返回布尔结果。逻辑运算符具有不同的优先级；下表按优先级递减的顺序列出了逻辑运算符：

运算符	执行的运算
&&	逻辑 AND
||	逻辑 OR

条件运算符

条件运算符是一个三元运算符，也就是说它涉及三个操作数。条件运算符是应用 if..else 条件语句的一种简化方式。

运算符	执行的运算
?:	条件

赋值运算符

赋值运算符涉及两个操作数，它根据一个操作数的值对另一个操作数进行赋值。下表列出了所有赋值运算符，它们具有相同的优先级：

运算符	执行的运算
=	赋值
*=	乘法赋值
/=	除法赋值
%=	求模赋值
+=	加法赋值
-=	减法赋值
<<=	按位左移位赋值
>>=	按位右移位赋值
>>>=	按位无符号右移位赋值
&=	按位 AND 赋值
^=	按位 XOR 赋值
|=	按位 OR 赋值

条件语句

ActionScript 3.0 提供了三个可用来控制程序流的基本条件语句。

if..else

使用 if..else 条件语句可以测试一个条件，如果该条件存在，则执行一个代码块，如果该条件不存在，则执行替代代码块。例如，下面的代码测试 x 的值是否超过 20，如果是，则生成一个 trace() 函数，如果不是则生成另一个 trace() 函数：

```
if (x > 20)
{
    trace("x is > 20");
}
else
{
    trace("x is <= 20");
}
```

如果您不想执行替代代码块，则可以仅使用 if 语句，而不用 else 语句。

if..else if

可以使用 if..else if 条件语句测试多个条件。例如，下面的代码不仅测试 x 的值是否超过 20，而且还测试 x 的值是否为负数：

```
if (x > 20)
{
    trace("x is > 20");
}
else if (x < 0)
{
    trace("x is negative");
}
```

如果 if 或 else 语句后面只有一条语句，则无需用大括号括起该语句。例如，下面的代码不使用大括号：

```
if (x > 0)
    trace("x is positive");
else if (x < 0)
    trace("x is negative");
else
    trace("x is 0");
```

但是， Adobe 建议您始终使用大括号，因为以后如果在缺少大括号的条件语句中添加语句，可能会出现无法预期的行为。例如，在下面的代码中，无论条件的计算结果是否为 true， positiveNums 的值总是按 1 递增：

```
var x:int;
var positiveNums:int = 0;

if (x > 0)
    trace("x is positive");
    positiveNums++;

trace(positiveNums); // 1
```

switch

如果多个执行路径依赖于同一个条件表达式，则 switch 语句非常有用。该语句的功能与一长段 if..else if 系列语句类似，但是更易于阅读。 switch 语句不是对条件进行测试以获得布尔值，而是对表达式进行求值并使用计算结果来确定要执行的代码块。代码块以 case 语句开头，以 break 语句结尾。例如，下面的 switch 语句基于由 Date.getDay() 方法返回的日期值输出星期日期：

```
var someDate:Date = new Date();
var dayNum:uint = someDate.getDay();
switch(dayNum)
{
    case 0:
        trace("Sunday");
        break;
    case 1:
        trace("Monday");
        break;
    case 2:
        trace("Tuesday");
        break;
    case 3:
        trace("Wednesday");
        break;
    case 4:
        trace("Thursday");
        break;
    case 5:
        trace("Friday");
        break;
    case 6:
        trace("Saturday");
        break;
    default:
        trace("Out of range");
        break;
}
```

循环

循环语句允许您使用一系列值或变量来反复执行一个特定的代码块。Adobe 建议您始终用大括号 ({}) 括起代码块。尽管您可以在代码块中只包含一条语句时省略大括号，但是就像在介绍条件语言时所提到的那样，不建议您这样做，原因也相同：这样会增加无意中将以后添加的语句从代码块中排除的可能性。如果您以后添加一条语句，并希望将它包括在代码块中，但忘了加必要的大括号，则该语句不会在循环过程中执行。

for

使用 for 循环可以循环访问某个变量以获得特定范围的值。必须在 for 语句中提供 3 个表达式：一个设置了初始值的变量，一个用于确定循环何时结束的条件语句，以及一个在每次循环中都更改变量值的表达式。例如，下面的代码循环 5 次。变量 i 的值从 0 开始到 4 结束，输出结果是从 0 到 4 的五个数字，每个数字各占一行。

```
var i:int;
for (i = 0; i < 5; i++)
{
    trace(i);
}
```

for..in

for..in 循环访问对象属性或数组元素。例如，可以使用 for..in 循环来循环访问通用对象的属性 （不按任何特定的顺序来保存对象的属性，因此属性可能以看似随机的顺序出现）：

```
var myObj:Object = {x:20, y:30};
for (var i:String in myObj)
{
    trace(i + ": " + myObj[i]);
}
// output:
// x: 20
// y: 30
```

还可以循环访问数组中的元素：

```
var myArray:Array = ["one", "two", "three"];
for (var i:String in myArray)
{
    trace(myArray[i]);
}
// output:
// one
// two
// three
```

如果对象是密封类 （包括内置类和用户定义的类）的实例，则不能循环访问该对象的属性。您只能循环访问动态类的属性。即使是动态类的实例，您也只能循环访问动态添加的属性。

for each..in

for each..in 循环用于循环访问集合中的项，这些项可以是 XML 或 XMLList 对象中的标签、对象属性保存的值或数组元素。如下面这段摘录的代码所示，可以使用 for each..in 循环来循环访问通用对象的属性，但是与 for..in 循环不同的是， for each..in 循环中的迭代变量包含属性所保存的值，而不包含属性的名称：

```
var myObj:Object = {x:20, y:30};
for each (var num in myObj)
{
    trace(num);
}
// output:
// 20
// 30
```

您可以循环访问 XML 或 XMLList 对象，如下面的示例所示：

```
var myXML:XML = <users>
    <fname>Jane</fname>
    <fname>Susan</fname>
    <fname>John</fname>
</users>;

for each (var item in myXML.fname)
{
    trace(item);
}
/* output
Jane
Susan
John
*/
```

还可以循环访问数组中的元素，如下面的示例所示：

```
var myArray:Array = ["one", "two", "three"];
for each (var item in myArray)
{
    trace(item);
}
// output:
// one
// two
// three
```

如果对象是密封类的实例，则您将无法循环访问该对象的属性。即使对于动态类的实例，也无法循环访问任何固定属性 （即，作为类定义的一部分定义的属性）。

while

while 循环与 if 语句相似，只要条件为 true，就会反复执行。例如，下面的代码与 for 循环示例生成的输出结果相同：

```
var i:int = 0;
while (i < 5)
{
    trace(i);
    i++;
}
```

使用 while 循环（而非 for 循环）的一个缺点是，编写 while 循环更容易导致无限循环。如果遗漏递增计数器变量的表达式，则 for 循环示例代码将无法编译；而 while 循环示例代码将能够编译。若没有用来递增 i 的表达式，循环将成为无限循环。

do..while

do..while 循环是一种 while 循环，保证至少执行一次代码块，这是因为在执行代码块后才会检查条件。下面的代码显示了 do..while 循环的一个简单示例，该示例在条件不满足时也会生成输出结果：

```
var i:int = 5;
do
{
    trace(i);
    i++;
} while (i < 5);
// output: 5
```

函数

函数是执行特定任务并可以在程序中重用的代码块。 ActionScript 3.0 中有两种函数类型：方法和函数闭包。将函数称为方法还是函数闭包取决于定义函数的上下文。如果您将函数定义为类定义的一部分或者将它附加到对象的实例，则该函数称为方法。如果您以其他任何方式定义函数，则该函数称为函数闭包。

函数在 ActionScript 中始终扮演着极为重要的角色。例如，在 ActionScript 1.0 中，不存在 class 关键字，因此“类”由构造函数定义。尽管之后的 ActionScript 版本中已经添加了 class 关键字，但是，如果您想充分利用该语言所提供的功能，深入了解函数仍然十分重要。对于希望 ActionScript 函数的行为与 C++ 或 Java 等语言中的函数的行为相似的程序员来说，这可能是一个挑战。尽管基本的函数定义和调用对有经验的程序员来说应不是什么问题，但是仍需要对 ActionScript 函数的一些更高级功能进行解释。

函数的基本概念

调用函数

可通过使用后跟小括号运算符 (()) 的函数标识符来调用函数。要发送给函数的任何函数参数都括在小括号中。例如， trace() 函数在 ActionScript 3.0 中是一个顶级函数：

```
trace("Use trace to help debug your script");
```

如果要调用没有参数的函数，则必须使用一对空的小括号。例如，可以使用没有参数的 Math.random() 方法来生成一个随机数：

```
var randomNum:Number = Math.random();
```

定义您自己的函数

在 ActionScript 3.0 中可通过两种方法来定义函数：使用函数语句和使用函数表达式。您可以根据自己的编程风格 （偏于静态还是偏于动态）来选择相应的方法。如果您倾向于采用静态或严格模式的编程，则应使用函数语句来定义函数。如果您有特定的需求，需要用函数表达式来定义函数，则应这样做。函数表达式更多地用在动态编程或标准模式编程中。

函数语句

函数语句是在严格模式下定义函数的首选方法。函数语句以 function 关键字开头，后跟：

- 函数名
- 用小括号括起来的逗号分隔参数列表
- 用大括号括起来的函数体 — 即在调用函数时要执行的 ActionScript 代码

例如，下面的代码创建一个定义一个参数的函数，然后将字符串 “hello” 用作参数值来调用该函数：

```
function traceParameter(aParam:String)
{
    trace(aParam);
}

traceParameter("hello"); // hello
```

函数表达式

声明函数的第二种方法就是结合使用赋值语句和函数表达式，函数表达式有时也称为函数字面值或匿名函数。这是一种较为繁杂的方法，在早期的 ActionScript 版本中广为使用。

带有函数表达式的赋值语句以 var 关键字开头，后跟：

- 函数名
- 冒号运算符 (:)
- 指示数据类型的 Function 类
- 赋值运算符 (=)
- function 关键字
- 用小括号括起来的逗号分隔参数列表
- 用大括号括起来的函数体 — 即在调用函数时要执行的 ActionScript 代码

例如，下面的代码使用函数表达式来声明 traceParameter 函数：

```
var traceParameter:Function = function (aParam:String)
{
    trace(aParam);
};
traceParameter("hello"); // hello
```

请注意，就像在函数语句中一样，在上面的代码中也没有指定函数名。函数表达式和函数语句的另一个重要区别是，函数表达式是表达式，而不是语句。这意味着函数表达式不能独立存在，而函数语句则可以。函数表达式只能用作语句 （通常是赋值语句）的一部分。下面的示例显示了一个赋予数组元素的函数表达式：

```
var traceArray:Array = new Array();
traceArray[0] = function (aParam:String)
{
    trace(aParam);
};
traceArray[0]("hello");
```

在函数语句和函数表达式之间进行选择

原则上，除非在特殊情况下要求使用表达式，否则应使用函数语句。函数语句较为简洁，而且与函数表达式相比，更有助于保持严格模式和标准模式的一致性。

函数语句比包含函数表达式的赋值语句更便于阅读。与函数表达式相比，函数语句使代码更为简洁而且不容易造成混淆，因为函数表达式既需要 var 关键字又需要 function 关键字。

函数语句更有助于保持严格模式和标准模式的一致性，因为在这两种编译器模式下，均可借助点语法来调用使用函数语句声明的方法。但这对于用函数表达式声明的方法却不一定成立。例如，下面的代码定义了一个具有下面两个方法的 Example 类：methodExpression() （用函数表达式声明）和 methodStatement() （用函数语句声明）。在严格模式下，不能使用点语法来调用 methodExpression() 方法。

```
class Example
{
var methodExpression = function() {}
function methodStatement() {}
}

var myEx:Example = new Example();
myEx.methodExpression(); // error in strict mode; okay in standard mode
myEx.methodStatement(); // okay in strict and standard modes
```

一般认为，函数表达式更适合于关注运行时行为或动态行为的编程。如果您喜欢使用严格模式，但是还需要调用使用函数表达式声明的方法，则可以使用这两种方法中的任一方法。首先，可以使用中括号 ([]) 代替点运算符 (.) 来调用该方法。下面的方法调用在严格模式和标准模式下都能够成功执行：

```
myExample["methodLiteral"]();
```

第二，您可以将整个类声明为动态类。尽管这样您就可以使用点运算符来调用方法，但缺点是，该类的所有实例在严格模式下都将丢失一些功能。例如，如果您尝试访问动态类实例的未定义属性，则编译器不生成错误。

在某些情况下，函数表达式非常有用。函数表达式的一个常见用法就是用于那些使用一次后便丢弃的函数。另一个用法就是向原型属性附加函数，这个用法不太常见。有关详细信息，请参阅原型对象。

函数语句与函数表达式之间有两个细微的区别，在选择要使用的方法时，应考虑这两个区别。第一个区别体现在内存管理和垃圾回收方面，因为函数表达式不像对象那样独立存在。换言之，当您将某个函数表达式分配给另一个对象 （如数组元素或对象属性）时，就会在代码中创建对该函数表达式的唯一引用。如果该函数表达式所附加到的数组或对象脱离作用域或由于其他原因不再可用，您将无法再访问该函数表达式。如果删除该数组或对象，该函数表达式所使用的内存就会符合垃圾回收条件，这意味着内存符合回收条件并可重新用于其他用途。

下面的示例说明对于函数表达式，一旦删除该表达式所赋予的属性，该函数就不再可用。 Test 类是动态的，这意味着您可以添加一个名为 functionExp 的属性来保存函数表达式。functionExp() 函数可以用点运算符来调用，但是一旦删除了 functionExp 属性，就无法再访问该函数。

```
dynamic class Test {}
var myTest:Test = new Test();

// function expression
myTest.functionExp = function () { trace("Function expression") };
myTest.functionExp(); // Function expression
delete myTest.functionExp;
myTest.functionExp(); // error
```

另一方面，如果该函数最初是用函数语句定义的，那么，该函数将以对象的形式独立存在，即使在您删除它所附加到的属性之后，该函数仍将存在。delete 运算符仅适用于对象的属性，因此，即使是用于删除 stateFunc() 函数本身的调用也不工作。

```
dynamic class Test {}
var myTest:Test = new Test();

// function statement
function stateFunc() { trace("Function statement") }
myTest.statement = stateFunc;
myTest.statement(); // Function statement
delete myTest.statement;
delete stateFunc; // no effect
stateFunc();// Function statement
myTest.statement(); // error
```

函数语句与函数表达式之间的第二个区别是，函数语句存在于定义它们的整个作用域 （包括出现在该函数语句前面的语句）内。与之相反，函数表达式只是为后续的语句定义的。例如，下面的代码能够在定义 scopeTest() 函数之前成功调用该函数：

```
statementTest(); // statementTest

function statementTest():void
{
    trace("statementTest");
}
```

函数表达式只有在定义之后才可用，因此，下面的代码会生成运行时错误：

```
expressionTest(); // run-time error

var expressionTest:Function = function ()
{
    trace("expressionTest");
}
```

从函数中返回值

要从函数中返回值，请使用后跟要返回的表达式或字面值的 return 语句。例如，下面的代码返回一个表示参数的表达式：

```
function doubleNum(baseNum:int):int
{
    return (baseNum * 2);
}
```

请注意， return 语句会终止该函数，因此，不会执行位于 return 语句下面的任何语句，如下所示：

```
function doubleNum(baseNum:int):int {
    return (baseNum * 2);
    trace("after return"); // This trace statement will not be executed.
}
```

在严格模式下，如果您选择指定返回类型，则必须返回相应类型的值。例如，下面的代码在严格模式下会生成错误，因为它们不返回有效值：

```
function doubleNum(baseNum:int):int
{
    trace("after return");
}
```

嵌套函数

您可以嵌套函数，这意味着函数可以在其他函数内部声明。除非将对嵌套函数的引用传递给外部代码，否则嵌套函数将仅在其父函数内可用。例如，下面的代码在 getNameAndVersion() 函数内部声明两个嵌套函数：

```
function getNameAndVersion():String
{
    function getVersion():String
    {
        return "10";
    }
    function getProductName():String
    {
        return "Flash Player";
    }
    return (getProductName() + " " + getVersion());
}
trace(getNameAndVersion()); // Flash Player 10
```

在将嵌套函数传递给外部代码时，它们将作为函数闭包传递，这意味着嵌套函数保留在定义该函数时处于作用域内的任何定义。有关详细信息，请参阅函数作用域。

函数参数

ActionScript 3.0 为函数参数提供了一些功能，这些功能对于那些刚接触 ActionScript 语言的程序员来说可能是很陌生的。尽管大多数程序员都熟悉按值或按引用传递参数这一概念，但是很多人可能对 arguments 对象和 ... (rest) 参数感到很陌生。

按值或按引用传递参数

在许多编程语言中，一定要了解按值传递参数与按引用传递参数之间的区别；这种区别会影响代码的设计方式。

按值传递意味着将参数的值复制到局部变量中以便在函数内使用。按引用传递意味着将只传递对参数的引用，而不传递实际值。这种方式的传递不会创建实际参数的任何副本，而是会创建一个对变量的引用并将它作为参数传递，并且会将它赋给局部变量以便在函数内部使用。局部变量是对函数外部的变量的引用，它使您能够更改初始变量的值。

在 ActionScript 3.0 中，所有的参数均按引用传递，因为所有的值都存储为对象。但是，属于基元数据类型 （包括 Boolean、Number、 int、 uint 和 String）的对象具有一些特殊运算符，这使它们可以像按值传递一样工作。例如，下面的代码创建一个名为 passPrimitives() 的函数，该函数定义了两个类型均为 int、名称分别为 xParam 和 yParam 的参数。这些参数与在 passPrimitives() 函数体内声明的局部变量类似。当使用 xValue 和 yValue 参数调用函数时， xParam 和 yParam 参数将用对 int 对象 （由 xValue 和 yValue 表示）的引用进行初始化。因为参数是基元值，所以它们像按值传递一样工作。尽管 xParam 和 yParam 最初仅包含对 xValue 和 yValue 对象的引用，但是，对函数体内的变量的任何更改都会导致在内存中生成这些值的新副本。

```
function passPrimitives(xParam:int, yParam:int):void
{
    xParam++;
    yParam++;
    trace(xParam, yParam);
}

var xValue:int = 10;
var yValue:int = 15;
trace(xValue, yValue);// 10 15
passPrimitives(xValue, yValue); // 11 16
trace(xValue, yValue);// 10 15
```

在 passPrimitives() 函数内部，xParam 和 yParam 的值递增，但这不会影响 xValue 和 yValue 的值，如上一条 trace 语句所示。即使参数的命名与 xValue 和 yValue 变量的命名完全相同也是如此，因为函数内部的 xValue 和 yValue 将指向内存中的新位置，这些位置不同于函数外部同名的变量所在的位置。

其他所有对象 （即不属于基元数据类型的对象）始终按引用传递，这样您就可以更改初始变量的值。例如，下面的代码创建一个名为 objVar 的对象，该对象具有两个属性：x 和 y。该对象作为参数传递给 passByRef() 函数。因为该对象不是基元类型，所以它不但按引用传递，而且还保持一个引用。这意味着对函数内部的参数的更改会影响到函数外部的对象属性。

```
function passByRef(objParam:Object):void
{
    objParam.x++;
    objParam.y++;
    trace(objParam.x, objParam.y);
}
var objVar:Object = {x:10, y:15};
trace(objVar.x, objVar.y); // 10 15
passByRef(objVar); // 11 16
trace(objVar.x, objVar.y); // 11 16
```

objParam 参数与全局 objVar 变量引用相同的对象。正如在本示例的 trace 语句中所看到的一样，对 objParam 对象的 x 和 y 属性所做的更改将反映在 objVar 对象中。

默认参数值

在 ActionScript 3.0 中，可以为函数声明默认参数值。如果在调用具有默认参数值的函数时省略了具有默认值的参数，那么，将使用在函数定义中为该参数指定的值。所有具有默认值的参数都必须放在参数列表的末尾。指定为默认值的值必须是编译时常量。如果某个参数存在默认值，其效果会使该参数成为可选参数。没有默认值的参数被视为必需参数。

例如，下面的代码创建一个具有三个参数的函数，其中的两个参数具有默认值。当仅用一个参数调用该函数时，将使用这些参数的默认值。

```
function defaultValues(x:int, y:int = 3, z:int = 5):void
{
    trace(x, y, z);
}
defaultValues(1); // 1 3 5
```

arguments 对象

在将参数传递给某个函数时，可以使用 arguments 对象来访问有关传递给该函数的参数的信息。 arguments 对象的一些重要方面包括：

- arguments 对象是一个数组，其中包括传递给函数的所有参数。
- arguments.length 属性报告传递给函数的参数数量。
- arguments.callee 属性提供对函数本身的引用，这对于递归调用函数表达式很有用。

  注：如果将任何参数命名为 arguments，或者使用 ...(rest) 参数，则 arguments 对象不可用。

  如果在函数体中引用了 arguments 对象，则在 ActionScript 3.0 中，函数调用中所包括的参数的数量可以大于在函数定义中所定义的参数数量，但是，如果参数的数量与必需参数 （以及可选情况下的任何可选参数）的数量不匹配，则在严格模式下会生成编译器错误。您可以使用 arguments 对象的数组样式来访问传递给函数的任何参数，而无需考虑是否在函数定义中定义了该参数。下面的示例 （仅在标准模式下进行编译）使用 arguments 数组及 arguments.length 属性来跟踪传递给 traceArgArray() 函数的所有参数：

```
function traceArgArray(x:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

arguments.callee 属性通常用在匿名函数中以创建递归。您可以使用它来提高代码的灵活性。如果递归函数的名称在开发周期内的不同阶段会发生改变，而且您使用的是 arguments.callee（而非函数名），则不必花费精力在函数体内更改递归调用。下面的函数表达式中使用 arguments.callee 属性来支持递归：

```
var factorial:Function = function (x:uint)
{
    if(x == 0)
    {
        return 1;
    }
    else
    {
        return (x * arguments.callee(x - 1));
    }
}

trace(factorial(5)); // 120
```

如果在函数声明中使用 ... (rest) 参数，则不能使用 arguments 对象，而必须使用为参数声明的参数名来访问参数。

还应避免将 “arguments” 字符串用作参数名，因为该字符串会遮蔽 arguments 对象。例如，如果重写 traceArgArray() 函数，以便添加 arguments 参数，那么，函数体内对 arguments 的引用所引用的将是该参数，而不是 arguments 对象。下面的代码不生成输出结果：

```
function traceArgArray(x:int, arguments:int):void
{
    for (var i:uint = 0; i < arguments.length; i++)
    {
        trace(arguments[i]);
    }
}

traceArgArray(1, 2, 3);

// no output
```

在早期的 ActionScript 版本中，arguments 对象还包含一个名为 caller 的属性，该属性是对调用了当前函数的函数的引用。ActionScript 3.0 中没有 caller 属性，但是，如果您需要引用调用函数，则可以更改调用函数，以使其传递一个额外的参数来引用其本身。

... (rest) 参数

ActionScript 3.0 中引入了一个称为 ... (rest) 参数的新参数声明。此参数可用来指定一个数组参数以接受任意多个以逗号分隔的参数。参数可以拥有保留字以外的任意名称。此参数声明必须是最后一个指定的参数。使用此参数会使 arguments 对象不可用。尽管 ... (rest) 参数提供了与 arguments 数组和 arguments.length 属性相同的功能，但是未提供与由 arguments.callee 提供的功能类似的功能。使用 ...(rest) 参数之前，应确保不需要使用 arguments.callee。

下面的示例使用 ... (rest) 参数 （而不是 arguments 对象）来重写 traceArgArray() 函数：

```
function traceArgArray(... args):void
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 1
// 2
// 3
```

... (rest) 参数还可与其他参数一起使用，前提是 ... (rest) 参数是列出的最后一个参数。下面的示例修改 traceArgArray() 函数，以便其第一个参数 x 是 int 类型，第二个参数使用 ... (rest) 参数。输出结果会忽略第一个值，因为第一个参数不再属于由 ...(rest) 参数创建的数组。

```
function traceArgArray(x: int, ... args)
{
    for (var i:uint = 0; i < args.length; i++)
    {
        trace(args[i]);
    }
}

traceArgArray(1, 2, 3);

// output:
// 2
// 3
```

函数作为对象

ActionScript 3.0 中的函数是对象。当您创建函数时，就是在创建对象，该对象不仅可以作为参数传递给另一个函数，而且还可以有附加的属性和方法。

作为参数传递给另一个函数的函数是按引用 （而不是按值）传递的。在将某个函数作为参数传递时，只能使用标识符，而不能使用在调用方法时所用的小括号运算符。例如，下面的代码将名为 clickListener() 的函数作为参数传递给 addEventListener() 方法：

```
addEventListener(MouseEvent.CLICK, clickListener);
```

尽管刚接触 ActionScript 的程序员可能对此感觉有些奇怪，但是，函数确实可以像其他任何对象那样具有属性和方法。实际上，每个函数都有一个名为 length 的只读属性，它用来存储为该函数定义的参数数量。该属性与 arguments.length 属性不同，后者报告发送给函数的参数数量。回想一下，在 ActionScript 中，发送给函数的参数数量可以超过为该函数定义的参数数量。下面的示例说明了这两个属性之间的区别，此示例仅在标准模式下编译，因为严格模式要求所传递的参数数量与所定义的参数数量完全相同：

```
// Compiles only in standard mode
function traceLength(x:uint, y:uint):void
{
    trace("arguments received: " + arguments.length);
    trace("arguments expected: " + traceLength.length);
}

traceLength(3, 5, 7, 11);
/* output:
arguments received: 4
arguments expected: 2 */
```

在标准模式下，可以通过在函数体外部定义函数属性来定义您自己的函数属性。函数属性可以用作准静态属性，用来保存与该函数有关的变量的状态。例如，您可能希望跟踪对特定函数的调用次数。如果您正在编写游戏，并且希望跟踪用户使用特定命令的次数，则这样的功能会非常有用，尽管您也可以使用静态类属性来实现此目的。下面的示例 （该示例仅在标准模式下进行编译，因为严格模式不允许向函数添加动态属性）在函数声明外部创建一个函数属性，并在每次调用该函数时递增此属性：

```
// Compiles only in standard mode
var someFunction:Function = function ():void
{
    someFunction.counter++;
}

someFunction.counter = 0;

someFunction();
someFunction();
trace(someFunction.counter); // 2
```

函数作用域

函数的作用域不但决定了可以在程序中的什么位置调用函数，而且还决定了函数可以访问哪些定义。适用于变量标识符的作用域规则同样也适用于函数标识符。在全局作用域中声明的函数在整个代码中都可用。例如， ActionScript 3.0 包含可在代码中的任意位置使用的全局函数，如 isNaN() 和 parseInt()。嵌套函数 （即在另一个函数中声明的函数）可以在声明它的函数中的任意位置上使用。

作用域链

无论何时开始执行函数，都会创建许多对象和属性。首先，会创建一个称为激活对象的特殊对象，该对象用于存储在函数体内声明的参数以及任何局部变量或函数。由于激活对象属于内部机制，因此您无法直接访问它。接着，会创建一个作用域链，其中包含由运行时检查标识符声明的对象的有序列表。所执行的每个函数都有一个存储在内部属性中的作用域链。对于嵌套函数，作用域链始于其自己的激活对象，后跟其父函数的激活对象。作用域链以这种方式延伸，直到到达全局对象。全局对象是在 ActionScript 程序开始时创建的，其中包含所有的全局变量和函数。

函数闭包

函数闭包是一个对象，其中包含函数的快照及其词汇环境。函数的词汇环境包括函数作用域链中的所有变量、属性、方法和对象以及它们的值。无论何时在对象或类之外的位置执行函数，都会创建函数闭包。函数闭包保留定义它们的作用域，这样，在将函数作为参数或返回值传递给另一个作用域时，会产生有趣的结果。

例如，下面的代码创建两个函数：foo() （返回一个用来计算矩形面积的嵌套函数 rectArea()）和 bar() （调用 foo() 并将返回的函数闭包存储在名为 myProduct 的变量中）。即使 bar() 函数定义了自己的局部变量 x （值为 2），当调用函数闭包 myProduct() 时，该函数闭包仍保留在函数 foo() 中定义的变量 x （值为 40）。因此， bar() 函数会返回值 160，而不是 8。

```
function foo():Function
{
    var x:int = 40;
    function rectArea(y:int):int // function closure defined
    {
        return x * y
    }
    return rectArea;
}
function bar():void
{
    var x:int = 2;
    var y:int = 4;
    var myProduct:Function = foo();
    trace(myProduct(4)); // function closure called
}
bar(); // 160
```

方法的行为与函数闭包类似，因为方法也保留有关创建它们的词汇环境的信息。当方法提取自它的实例 （这会创建绑定方法）时，此特征尤为突出。函数闭包与绑定方法之间的主要区别在于，绑定方法中 this 关键字的值始终引用它最初附加到的实例，而函数闭包中 this 关键字的值可以改变。

第 4 章 : ActionScript 中面向对象的编程

面向对象的编程简介

面向对象的编程 (OOP) 通过将程序中的代码分组到对象中来组织代码。从这个意义上说，术语对象表示包含信息 （数据值）和功能在内的单个元素。当您使用面向对象的方式组织程序时，就将特定信息与通用功能或与该信息相关的动作组合起来。例如，您可以将唱片标题、曲目标题或艺术家名称等音乐信息与 “ 向播放列表添加曲目 ” 或 “ 播放此艺术家演唱的所有歌曲 ” 等功能组合起来。这些内容将合并为一个项目，即一个对象 （例如，“ 唱片 ” 或 “ 音轨 ”）。将值和函数绑定到一起有几个优点。一个主要的优点是您只需要使用一个变量而不是多个变量。此外，这样可将相关功能放在一起。最后一点，通过合并信息和功能，您可以按照更符合实际情况的方式构建程序。

类

类是对象的抽象表示形式。类用来存储有关对象可保存的数据类型及对象可表现的行为的信息。如果编写的小脚本中只包含几个彼此交互的对象，使用这种抽象类的作用并不一定明显。然而，随着程序的作用域不断扩大，必须管理的对象数量也在增加。在这种情况下，通过类可以更好地控制创建对象的方式以及对象彼此交互的方式。

早在 ActionScript 1.0 中，ActionScript 程序员就能使用 Function 对象创建类似类的构造函数。在 ActionScript 2.0 中，通过使用 class 和 extends 等关键字，正式添加了对类的支持。 ActionScript 3.0 不仅继续支持 ActionScript 2.0 中引入的关键字，而且还添加了新功能。例如，ActionScript 3.0 包含对 protected 和 internal 属性的增强的访问控制。它还通过使用 final 和 override 关键字，提供了更好的继承控制。

对于使用过 Java、C++ 或 C# 等编程语言创建类的开发人员，ActionScript 提供了熟悉的体验。ActionScript 共享许多相同的关键字和属性名称，例如 class、 extends 和 public。

注：在 Adobe ActionScript 文档中，术语 “ 属性 ” 表示对象或类的任何成员，包括变量、常量和方法。此外，虽然术语 “ 类 ” 和 “ 静态 ” 经常互换使用，但在本章中这两个术语是不同的。例如，“ 类属性 ” 指的是类的所有成员，而不仅是静态成员。

类定义

ActionScript 3.0 类定义使用的语法与 ActionScript 2.0 类定义使用的语法相似。正确的类定义语法要求 class 关键字后跟类名。类体要放在大括号 ({}) 内，且放在类名后面。例如，下面的代码创建名为 Shape 的类，其中包含名为 visible 的变量：

```
public class Shape
{
    var visible:Boolean = true;
}
```

对于包中的类定义，有一项重要的语法更改。在 ActionScript 2.0 中，如果类在包中，则在类声明中必须包含包名称。在 ActionScript 3.0 中，引入了 package 语句，包名称必须包含在包声明中，而不是包含在类声明中。例如，以下类声明说明如何在 ActionScript 2.0 和 ActionScript 3.0 中定义 BitmapData 类 （该类是 flash.display 包的一部分）：

```
// ActionScript 2.0
class flash.display.BitmapData {}

// ActionScript 3.0
package flash.display
{
    public class BitmapData {}
}
```

类属性

在 ActionScript 3.0 中，可使用以下四个属性之一来修改类定义：

属性	定义
dynamic	允许在运行时向实例添加属性。
final	不得由其他类扩展。
internal（默认）	对当前包内的引用可见。
public	对所有位置的引用可见。

使用 internal 以外的每个属性时，显式包含该属性才能实现相关的行为。例如，如果定义类时未包含 dynamic 属性，则不能在运行时向类实例中添加属性。通过在类定义的开始处放置属性，可显式地分配属性，如下面的代码所示：

```
dynamic class Shape {}
```

请注意，列表中未包含名为 abstract 的属性。 ActionScript 3.0 不支持抽象类。另请注意，该列表也未包含名为 private 和 protected 的属性。这些属性只在类定义中有意义，但不可以应用于类本身。如果不希望某个类在包以外公开可见，请将该类放在包中，并用 internal 属性标记该类。或者，可以省略 internal 和 public 这两个属性，编译器会自动为您添加 internal 属性。还可以将类定义为仅在定义此类的源文件内可见。将类置于源文件的底部，包定义的右大括号下方。

类体

类体用大括号括起来。类体定义类的变量、常量和方法。以下示例显示 ActionScript 3.0 中对 Accessibility 类的声明：

```
public final class Accessibility
{
    public static function get active():Boolean;
    public static function updateProperties():void;
}
```

还可以在类体中定义命名空间。下面的示例说明如何在类体中定义命名空间，以及如何在该类中将命名空间用作方法的属性：

```
public class SampleClass
{
    public namespace sampleNamespace;
    sampleNamespace function doSomething():void;
}
```

ActionScript 3.0 不但允许在类体中包括定义，而且还允许包括语句。位于类体内但在方法定义之外的语句，实际执行次数严格为一次。这在第一次遇到类定义并创建关联的类对象时执行。下面的示例包括一个对 hello() 外部函数的调用和一个 trace 语句，该语句在定义类时输出确认消息：

```
function hello():String
{
    trace("hola");
}
class SampleClass
{
    hello();
    trace("class created");
}
// output when class is created
hola
class created
```

在 ActionScript 3.0 中，允许在同一类体中定义同名的静态属性和实例属性。例如，下面的代码声明一个名为 message 的静态变量和一个同名的实例变量：

```
class StaticTest
{
    static var message:String = "static variable";
    var message:String = "instance variable";
}
// In your script
var myST:StaticTest = new StaticTest();
trace(StaticTest.message); // output: static variable
trace(myST.message); // output: instance variable
```

类属性的属性

讨论 ActionScript 对象模型时，术语属性指可以作为类成员的任何成员，包括变量、常量和方法。不过，在 《用于 Adobe Flash Platform 的 Adobe ActionScript 3.0 参考》中，该术语的使用范围比较严格。在该上下文中，属性这一术语仅包含作为变量或通过 getter 或 setter 方法定义的类成员。ActionScript 3.0 中提供了一组可以与类的任何属性一起使用的属性。下表列出了这组属性。

属性	定义
internal （默认）	对同一包中的引用可见。
private	对同一类中的引用可见。
protected	对同一类及派生类中的引用可见。
public	对所有位置的引用可见。
static	指定某一属性属于该类，而不属于该类的实例。
UserDefinedNamespace （用户定义的命名空间）	用户定义的自定义命名空间名。

访问控制命名空间属性

ActionScript 3.0 提供了四个特殊的属性来控制对在类中定义的属性的访问：public、 private、 protected 和 internal。

public 属性使某一属性在脚本的任何位置可见。例如，要使某个方法可用于包外部的代码，必须使用 public 属性声明该方法。这适用于任何属性，不管属性是使用 var、 const 还是 function 关键字声明的。

private 属性使某一属性只对属性的定义类中的调用方可见。这一行为与 ActionScript 2.0 中的 private 属性有所不同，后者允许子类访问超类中的私有属性。另一处明显的行为变化是必须执行运行时访问。在 ActionScript 2.0 中，private 关键字只在编译时禁止访问，运行时很容易避开它。在 ActionScript 3.0 中，这种情况不复存在。标记为 private 的属性在编译时和运行时都不可用。

例如，下面的代码创建一个带一个私有变量的名为 PrivateExample 的简单类，然后尝试从该类的外部访问该私有变量。

```
class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // ActionScript 2.0 allows access, but in ActionScript 3.0, this is a run-time
error.
```

在 ActionScript 3.0 中使用严格模式时，尝试使用点运算符 (myExample.privVar) 访问私有属性会导致编译时错误。否则，会在运行时报告错误，就像使用属性访问运算符 (myExample["privVar"]) 时一样。

下表汇总了试图访问属于密封 （非动态）类的 private 属性的结果：

	严格模式	标准模式
点运算符 (.)	编译时错误	运行时错误
中括号运算符 ([])	运行时错误	运行时错误

在使用 dynamic 属性声明的类中尝试访问私有变量不会导致运行时错误。但是，变量不可见，因此返回值 undefined。但是，如果在严格模式下使用点运算符，则会发生编译时错误。下面的示例与上一个示例相同，只是 PrivateExample 类被声明为动态类：

```
dynamic class PrivateExample
{
    private var privVar:String = "private variable";
}

var myExample:PrivateExample = new PrivateExample();
trace(myExample.privVar);// compile-time error in strict mode
trace(myExample["privVar"]); // output: undefined
```

当类外部的代码尝试访问 private 属性时，动态类通常会返回值 undefined，而不是生成错误。下表说明了只有在严格模式下使用点运算符访问 private 属性时才会生成错误：

	严格模式	标准模式
点运算符 (.)	编译时错误	undefined
中括号运算符 ([])	undefined	undefined

protected 属性是 ActionScript 3.0 中的新增属性，可使某一属性对于其所属类或子类中的调用方可见。换句话说，protected 属性在其所属类中可用，或者对继承层次中该类下面的类可用。无论子类在同一包中还是在不同包中，这一点都适用。

对熟悉 ActionScript 2.0 的人来说，本功能类似于 ActionScript 2.0 中的 private 属性。ActionScript 3.0 中的 protected 属性也类似于 Java 中的 protected 属性。不同之处在于 Java 版本还允许访问同一包中的调用方。如果存在子类需要的变量或方法，但要对继承链外部的代码隐藏该变量或方法，此时 protected 属性会很有用。

internal 属性是 ActionScript 3.0 的新增属性，可使某一属性对所在包中的调用方可见。该属性是包中代码的默认属性，它适用于任何没有以下任意属性的属性：

- public
- private
- protected
- 用户定义的命名空间

internal 属性与 Java 中的默认访问控制相似，不过，在 Java 中该访问级别没有明确的名称，只能通过省略所有其他访问修饰符来实现。 ActionScript 3.0 中提供的 internal 属性旨在为您提供一个明确表达自己意图的选项，以使属性仅对所在包中的调用方可见。

static 属性

static 属性可以与用 var、const 或 function 关键字声明的那些属性一起使用，使用该属性可将属性附加到类而不是类的实例。类外部的代码必须使用类名 （而不是使用实例名）调用静态属性。

静态属性不由子类继承，但是这些属性是子类作用域链中的一部分。这意味着在子类体中，不必引用在其中定义静态变量或方法的类，就可以使用静态变量或方法。

用户定义的命名空间属性

作为预定义访问控制属性的替代方法，您可以创建自定义命名空间以用作属性。每个定义只能使用一个命名空间属性，而且不能将命名空间属性与任何访问控制属性 （public、 private、 protected 和 internal）组合使用。

变量

可以使用 var 或 const 关键字声明变量。在脚本的整个执行过程中，使用 var 关键字声明的变量可多次更改其变量值。使用 const 关键字声明的变量称为常量，只能赋值一次。尝试给已初始化的常量分配新值，将生成错误。

静态变量

静态变量是使用 static 关键字和 var 或 const 语句共同声明的。静态变量附加到类而不是类的实例，对于存储和共享应用于对象的整个类的信息非常有用。例如，当要保存类实例化的总次数或者要存储允许的最大类实例数，使用静态变量比较合适。

下面的示例创建一个 totalCount 变量 （用于跟踪类实例化数）和一个 MAX_NUM 常量 （用于存储最大实例化数）。totalCount 和 MAX_NUM 这两个变量是静态变量，因为它们包含的值应用于整个类，而不是某个特定实例。

```
class StaticVars
{
    public static var totalCount:int = 0;
    public static const MAX_NUM:uint = 16;
}
```

StaticVars 类及其任何子类外部的代码只能通过该类本身来引用 totalCount 和 MAX_NUM 属性。例如，以下代码有效：

```
trace(StaticVars.totalCount); // output: 0
trace(StaticVars.MAX_NUM); // output: 16
```

不能通过类实例访问静态变量，因此以下代码会返回错误：

```
var myStaticVars:StaticVars = new StaticVars();
trace(myStaticVars.totalCount); // error
trace(myStaticVars.MAX_NUM); // error
```

必须在声明常量的同时初始化使用 static 和 const 关键字声明的变量，就像 StaticVars 类初始化 MAX_NUM 那样。您不能为构造函数或实例方法中的 MAX_NUM 赋值。以下代码会生成错误，因为它不是初始化静态常量的有效方法：

```
// !! Error to initialize static constant this way
class StaticVars2
{
    public static const UNIQUESORT:uint;
    function initializeStatic():void
    {
        UNIQUESORT = 16;
    }
}
```

实例变量

实例变量包括使用 var 和 const 关键字但未使用 static 关键字声明的属性。实例变量附加到类实例而不是整个类，对于存储特定于实例的值很有用。例如， Array 类有一个名为 length 的实例属性，用来存储 Array 类的特定实例保存的数组元素的个数。

不能覆盖子类中声明为 var 或 const 的实例变量。但是，通过覆盖 getter 和 setter 方法，可以实现类似于覆盖变量的功能。

方法

方法是类定义中的函数。创建类的一个实例后，该实例就会捆绑一个方法。与在类外部声明的函数不同，不能将方法与附加方法的实例分开使用。

方法是使用 function 关键字定义的。至于任一类属性，可以将任一类属性应用到方法，这些属性包括 private、 protected、 public、 internal、 static，或者自定义命名空间。您可以使用函数语句，例如：

```
public function sampleFunction():String {}
```

也可以使用分配了函数表达式的变量，如下所示：

```
public var sampleFunction:Function = function () {}
```

多数情况下都使用函数语句，而不是函数表达式，原因如下：

- 函数语句更为简洁易读。
- 函数语句允许使用 override 和 final 关键字。
- 函数语句在标识符 （函数名）与方法体代码之间创建了更强的绑定。由于可以使用赋值语句更改变量值，因此可随时断开变量与其函数表达式之间的连接。虽然可通过使用 const （而不是 var）声明变量来解决这个问题，但这种方法并不是最好的做法。这会使代码难以阅读，还会阻止使用 override 和 final 关键字。

必须使用函数表达式的一种情况是：选择将函数附加到原型对象时。

构造函数方法

构造函数方法有时称为构造函数，是与在其中定义函数的类共享同一名称的函数。只要使用 new 关键字创建了类实例，就会执行构造函数方法中包括的所有代码。例如，以下代码定义名为 Example 的简单类，该类包含名为 status 的属性。 status 变量的初始值是在构造函数中设置的。

```
class Example
{
    public var status:String;
    public function Example()
    {
        status = "initialized";
    }
}

var myExample:Example = new Example();
trace(myExample.status); // output: initialized
```

构造函数方法只能是公共方法，但可以选择性地使用 public 属性。不能对构造函数使用任何其他访问控制说明符 （包括使用 private、 protected 或 internal）。也不能对函数构造方法使用用户定义的命名空间。

构造函数可以使用 super() 语句显式地调用其直接超类的构造函数。如果未显式调用超类构造函数，编译器会在构造函数体中的第一个语句前自动插入一个调用。还可以使用 super 前缀作为对超类的引用来调用超类的方法。如果决定在同一构造函数中使用 super() 和 super，务必先调用 super()。否则， super 引用的行为会与预期不符。另外， super() 构造函数也应在 throw 或 return 语句之前调用。

下面的示例说明如果在调用 super() 构造函数之前尝试使用 super 引用，将会发生什么情况。新类 ExampleEx 扩展了 Example 类。 ExampleEx 构造函数尝试访问在其超类中定义的状态变量，但访问是在调用 super() 之前进行的。 ExampleEx 构造函数中的 trace() 语句生成了 null 值，原因是 status 变量在 super() 构造函数执行之前不可用。

```
class ExampleEx extends Example
{
    public function ExampleEx()
    {
        trace(super.status);
        super();
    }
}

var mySample:ExampleEx = new ExampleEx(); // output: null
```

虽然在构造函数中使用 return 语句是合法的，但是不允许返回值。换句话说， return 语句不得有相关的表达式或值。因此，不允许构造函数方法返回值，这意味着不可以指定任何返回值。

如果没有在类中定义构造函数方法，编译器会自动为您创建一个空构造函数。如果某个类扩展了另一个类，编译器会在所生成的构造函数中包括 super() 调用。

静态方法

静态方法也叫做类方法，它们是使用 static 关键字声明的方法。静态方法附加到类而不是类的实例，因此在封装对单个实例的状态以外的内容有影响的功能时，静态方法很有用。由于静态方法附加到整个类，所以只能通过类访问静态方法，而不能通过类实例访问。

静态方法为封装所提供的功能不仅仅在影响类实例状态的方面。换句话说，如果方法提供的功能对类实例的值没有直接的影响，该方法应是静态方法。例如， Date 类具有名为 parse() 的静态方法，它接收字符串并将其转换为数字。该方法就是静态方法，因为它并不影响类的单个实例。而 parse() 方法使用表示日期值的字符串，分析该字符串，然后使用与 Date 对象的内部表示形式兼容的格式返回一个数字。此方法不是实例方法，因为将该方法应用到 Date 类的实例并没有任何意义。

请将静态 parse() 方法与 Date 类的一个实例方法 （如 getMonth()）相比较。 getMonth() 方法是一个实例方法，因为它通过检索 Date 实例的特定组件 （即 month），对实例值直接执行操作。

由于静态方法不绑定到单个实例，因此不能在静态方法体中使用关键字 this 或 super。 this 和 super 这两个引用只在实例方法上下文中有意义。

与其他基于类的编程语言不同， ActionScript 3.0 中的静态方法不是继承的。

实例方法

实例方法指的是不使用 static 关键字声明的方法。实例方法附加到类实例而不是整个类，在实现对类的各个实例有影响的功能时，实例方法很有用。例如， Array 类包含名为 sort() 的实例方法，该实例方法直接对 Array 实例执行操作。

在实例方法体中，静态变量和实例变量都在作用域中，这表示使用一个简单的标识符可以引用同一类中定义的变量。例如，以下类 CustomArray 扩展了 Array 类。CustomArray 类定义一个名为 arrayCountTotal 的静态变量（用于跟踪类实例总数）、一个名为 arrayNumber 实例变量 （用于跟踪创建实例的顺序）和一个名为 getPosition() 的实例方法 （用于返回这两个变量的值）。

```
public class CustomArray extends Array
{
    public static var arrayCountTotal:int = 0;
    public var arrayNumber:int;

    public function CustomArray()
    {
        arrayNumber = ++arrayCountTotal;
    }

    public function getArrayPosition():String
    {
         return ("Array " + arrayNumber + " of " + arrayCountTotal);
    }
}
```

虽然类外部的代码必须使用 CustomArray.arrayCountTotal 通过类对象来访问 arrayCountTotal 静态变量，但是位于 getPosition() 方法体中的代码可以直接引用静态 arrayCountTotal 变量。即使对于超类中的静态变量，这一点也适用。虽然在 ActionScript 3.0 中不继承静态属性，但是超类的静态属性在作用域中。例如， Array 类有几个静态变量，其中一个是名为 DESCENDING 的常量。位于 Array 子类中的代码可以使用一个简单的标识符来访问静态常量 DESCENDING：

```
public class CustomArray extends Array
{
    public function testStatic():void
    {
        trace(DESCENDING); // output: 2
    }
}
```

实例方法体中的 this 引用的值是对方法所附加实例的引用。下面的代码说明 this 引用指向包含方法的实例：

```
class ThisTest
{
    function thisValue():ThisTest
    {
        return this;
    }
}

var myTest:ThisTest = new ThisTest();
trace(myTest.thisValue() == myTest); // output: true
```

使用关键字 override 和 final 可以控制实例方法的继承。可以使用 override 属性重新定义继承的方法，以及使用 final 属性禁止子类覆盖方法。

get 和 set 存取器方法

get 和 set 存取器函数还分别称为 getter 和 setter，可以使用这些函数为创建的类提供易于使用的编程接口，并遵循信息隐藏和封装的编程原则。使用 get 和 set 函数可保持类的私有类属性，但允许类用户访问这些属性，就像他们在访问类变量而不是调用类方法。

这种方法的好处是，可避免出现具有不实用名称的传统存取器函数，如 getPropertyName() 和 setPropertyName()。 getter 和 setter 的另一个好处是，使用它们可避免允许进行读写访问的每个属性有两个面向公共的函数。

下面的示例类名为 GetSet，其中包含名为 publicAccess() 的 get 和 set 存取器函数，用于提供对名为 privateProperty 的私有变量的访问：

```
class GetSet
{
    private var privateProperty:String;

    public function get publicAccess():String
    {
        return privateProperty;
    }

    public function set publicAccess(setValue:String):void
    {
        privateProperty = setValue;
    }
}
```

如果尝试直接访问属性 privateProperty，将发生错误，如下所示：

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.privateProperty); // error occurs
```

GetSet 类的用户所使用的对象是显示为名为 publicAccess 的属性，但实际上这是对名为 privateProperty 的 private 属性执行的一对 get 和 set 取值函数。下面的示例将实例化 GetSet 类，然后使用名为 publicAccess 的公共存取器设置 privateProperty 的值：

```
var myGetSet:GetSet = new GetSet();
trace(myGetSet.publicAccess); // output: null
myGetSet.publicAccess = "hello";
trace(myGetSet.publicAccess); // output: hello
```

使用 getter 和 setter 函数还可以覆盖从超类继承来的属性，这是使用常规类成员变量时不能做到的。在子类中不能覆盖使用 var 关键字声明的类成员变量。但是，使用 getter 和 setter 函数创建的属性没有此限制。可以对从超类继承的 getter 和 setter 函数使用 override 属性。

绑定方法

绑定方法有时也叫做闭包方法，就是从它的实例提取的方法。作为参数传递给函数的方法或作为值从函数返回的方法都是绑定方法。在 ActionScript 3.0 中，新增的绑定方法类似于闭包函数，其中保留了词汇环境，即使从其实例中提取出来也是如此。绑定方法与闭包函数之间的主要不同差别是，绑定函数的 this 引用保留到实现方法的实例的链接或绑定。换句话说，绑定方法中的 this 引用总是指向实现方法的原始对象。对于闭包函数， this 引用是通用的，这意味着调用函数时，该引用指向与函数关联的任何对象。

如果使用 this 关键字，了解绑定方法就很重要。重新调用 this 关键字可提供对方法父对象的引用。大多数 ActionScript 编程人员都希望 this 关键字总是表示包含方法定义的对象或类。但是，如果不使用方法绑定，并不是总是做到这样。例如，在以前版本的 ActionScript 中，this 引用并不总是引用实现方法的实例。从 ActionScript 2.0 的实例中提取方法后，不但 this 引用不绑定到原始实例，而且实例类的成员变量和方法也不可用。在 ActionScript 3.0 中不存在这样的问题，这是因为将方法当作参数传递时会自动创建绑定方法。绑定方法用于确保 this 关键字总是引用在其中定义了方法的对象或类。

下面的代码定义了名为 ThisTest 的类，该类包含一个名为 foo() 的方法 （该方法定义绑定方法）和一个名为 bar() 的方法 （该方法返回绑定方法）。类外部的代码创建 ThisTest 类的实例，然后调用 bar() 方法，最后将返回值存储在名为 myFunc 的变量中。

```
class ThisTest
{
    private var num:Number = 3;
    function foo():void // bound method defined
    {
        trace("foo's this: " + this);
        trace("num: " + num);
    }
    function bar():Function
    {
        return foo; // bound method returned
    }
}

var myTest:ThisTest = new ThisTest();
var myFunc:Function = myTest.bar();
trace(this); // output: [object global]
myFunc();
/* output:
foo's this: [object ThisTest]
output: num: 3 */
```

代码的最后两行表明：虽然前一行中的 this 引用指向全局对象，但绑定方法 foo() 中的 this 引用仍然指向 ThisTest 类的实例。另外，存储在 myFunc 变量中的绑定方法仍然可以访问 ThisTest 类的成员变量。如果以上代码在 ActionScript 2.0 中运行，this 引用会匹配，但 num 变量将为 undefined。

绑定方法最值得注意的一种情况是使用事件处理函数，因为 addEventListener() 方法要求将函数或方法作为参数来传递。

类的枚举

“ 枚举 ” 是您创建的一些自定义数据类型，用于封装一小组值。 ActionScript 3.0 并不支持具体的枚举工具，这与 C++ 使用 enum 关键字或 Java 使用 Enumeration 接口不一样。不过，您可以使用类或静态常量创建枚举。例如， ActionScript 3.0 中的 PrintJob 类使用名为 PrintJobOrientation 的枚举来存储 "landscape" 和 "portrait" 值，如下面的代码所示：

```
public final class PrintJobOrientation
{
    public static const LANDSCAPE:String = "landscape";
    public static const PORTRAIT:String = "portrait";
}
```

按照惯例，枚举类是使用 final 属性声明的，因为不需要扩展该类。该类仅包括静态成员，这表示不创建该类的实例。而是直接通过类对象来访问枚举值，如以下代码摘录中所示：

```
var pj:PrintJob = new PrintJob();
if(pj.start())
{
    if (pj.orientation == PrintJobOrientation.PORTRAIT)
    {
        ...
    }
    ...
}
```

ActionScript 3.0 中的所有枚举类都只包含 String、 int 或 uint 类型的变量。使用枚举而不使用文本字符串或数字值的好处是，使用枚举更易于发现字面错误。如果枚举名输入错误， ActionScript 编译器会生成一个错误。如果使用字面值，存在拼写错误或使用了错误数字时，编译器并不会报错。在上一个示例中，如果枚举常量的名称不正确，编译器会生成错误，如以下代码摘录中所示：

```
    if (pj.orientation == PrintJobOrientation.PORTRAI) // compiler error
```

但是，如果拼错了字符串字面值，编译器并不生成错误，如下所示：

```
    if (pj.orientation == "portrai") // no compiler error
```

创建枚举的第二种方法还包括使用枚举的静态属性创建单独的类。这种方法的不同之处在于每一个静态属性都包含一个类实例，而不是字符串或整数值。例如，以下代码为一星期中的各天创建了一个枚举类：

```
public final class Day
{
    public static const MONDAY:Day = new Day();
    public static const TUESDAY:Day = new Day();
    public static const WEDNESDAY:Day = new Day();
    public static const THURSDAY:Day = new Day();
    public static const FRIDAY:Day = new Day();
    public static const SATURDAY:Day = new Day();
    public static const SUNDAY:Day = new Day();
}
```

ActionScript 3.0 并不使用这种方法，但是许多开发人员都使用，他们更喜欢使用这种方法提供的改进类型检查功能。例如，返回枚举值的方法可将返回值限定为枚举数据类型。以下代码不但显示了返回星期中各天的函数，还显示了将枚举类型用作类型注释的函数调用：

```
function getDay():Day
{
    var date:Date = new Date();
    var retDay:Day;
    switch (date.day)
    {
        case 0:
            retDay = Day.MONDAY;
            break;
        case 1:
            retDay = Day.TUESDAY;
            break;
        case 2:
            retDay = Day.WEDNESDAY;
            break;
        case 3:
            retDay = Day.THURSDAY;
            break;
        case 4:
            retDay = Day.FRIDAY;
            break;
        case 5:
            retDay = Day.SATURDAY;
            break;
        case 6:
            retDay = Day.SUNDAY;
            break;
    }
    return retDay;
}

var dayOfWeek:Day = getDay();
```

您还可以增强 Day 类的功能，使其将一个整数与一星期中的各天关联，并提供一个 toString() 方法来返回各天的字符串表示形式。

嵌入资源类

ActionScript 3.0 使用称为嵌入资源类 的特殊类来表示嵌入的资源。嵌入资源 指的编译时包括在 SWF 文件中的资源，如声音、图像或字体。嵌入资源而不是动态加载资源可以确保资源在运行时可用，但代价是增加了 SWF 文件的大小。

在 Flash Professional 中使用嵌入的资源类

要嵌入资源，首先将该资源放入 FLA 文件的库中。接着，使用资源的链接属性，提供资源的嵌入资源类的名称。如果无法在类路径中找到具有该名称的类，则将自动生成一个类。然后，可以创建嵌入资源类的实例，并使用任何由该类定义或继承的属性和方法。例如，以下代码可用于播放链接到名为 PianoMusic 的嵌入资源类的嵌入声音：

```
var piano:PianoMusic = new PianoMusic();
var sndChannel:SoundChannel = piano.play();
```

还可使用 [Embed] 元数据标签在 Flash Professional 项目中嵌入资源，如下面所述。如果在代码中使用 [Embed] 元数据标签，Flash Professional 将使用 Flex 编译器而非 Flash Professional 编译器编译您的项目。

利用 Flex 编译器使用嵌入资源类

如果您正在使用 Flex 编译器编译代码，可使用 [Embed] 元数据标签将资源嵌入到 ActionScript 代码中。将资源放置在项目生成路径中的主资源文件夹或其他文件夹中。当 Flex 编译器遇到 Embed 元数据标签时，会创建嵌入资源类。通过紧接 [Embed] 元数据标签之后声明的数据类型 Class 的变量，可以访问该资源类。例如，下面的代码嵌入名为 sound1.mp3 的声音，使用名为 soundCls 的变量来存储对该声音的关联嵌入资源类的引用。然后，该示例创建嵌入资源类的实例，并对该实例调用 play() 方法：

```
package
{
    import flash.display.Sprite;
    import flash.media.SoundChannel;
    import mx.core.SoundAsset;

    public class SoundAssetExample extends Sprite
    {
        [Embed(source="sound1.mp3")]
        public var soundCls:Class;

        public function SoundAssetExample()
        {
            var mySound:SoundAsset = new soundCls() as SoundAsset;
            var sndChannel:SoundChannel = mySound.play();
        }
    }
}
```

Adobe Flash Builder

若要在 Flash Builder ActionScript 项目中使用 [Embed] 元数据标签，可从 Flex 框架导入所有必需的类。例如，若要嵌入声音，可导入 mx.core.SoundAsset 类。若要使用 Flex 框架，可将文件 framework.swc 包含在 ActionScript 生成路径中。这样会增加 SWF 文件的大小。

Adobe Flex

另外，在 Flex 中，通过在 MXML 标签定义中使用 @Embed() 指令也可以嵌入资源。

接口

接口是方法声明的集合，以使不相关的对象能够彼此通信。例如， ActionScript 3.0 定义了 IEventDispatcher 接口，其中包含的方法声明可供类用于处理事件对象。 IEventDispatcher 接口建立了标准方法，供对象相互传递事件对象。以下代码显示 IEventDispatcher 接口的定义：

```
public interface IEventDispatcher
{
    function addEventListener(type:String, listener:Function,
            useCapture:Boolean=false, priority:int=0,
            useWeakReference:Boolean = false):void;
    function removeEventListener(type:String, listener:Function,
            useCapture:Boolean=false):void;
    function dispatchEvent(event:Event):Boolean;
    function hasEventListener(type:String):Boolean;
    function willTrigger(type:String):Boolean;
}
```

接口的基础是方法的接口与方法的实现之间的区别。方法的接口包括调用该方法必需的所有信息，包括方法名、所有参数和返回类型。方法的实现不仅包括接口信息，而且还包括执行方法的行为的可执行语句。接口定义只包含方法接口，实现接口的所有类负责定义方法实现。

在 ActionScript 3.0 中， EventDispatcher 类通过定义所有 IEventDispatcher 接口方法并在每个方法中添加方法体来实现 IEventDispatcher 接口。以下代码摘录自 EventDispatcher 类定义：

```
public class EventDispatcher implements IEventDispatcher
{
    function dispatchEvent(event:Event):Boolean
    {
        /* implementation statements */
    }

    ...
}
```

IEventDispatcher 接口用作一个协议， EventDispatcher 实例通过该协议处理事件对象，然后将事件对象传递到也实现了 IEventDispatcher 接口的其他对象。

另一种描述接口的方法是：接口定义了数据类型，就像类一样。因此，接口可以用作类型注释，也像类一样。作为数据类型，接口还可以与需要指定数据类型的运算符一起使用，如 is 和 as 运算符。但是与类不同的是，接口不可以实例化。这个区别使很多程序员认为接口是抽象的数据类型，认为类是具体的数据类型。

定义接口

接口定义的结构类似于类定义的结构，只是接口只能包含方法但不能包含方法体。接口不能包含变量或常量，但是可以包含 getter 和 setter。要定义接口，请使用 interface 关键字。例如，接口 IExternalizable 是 ActionScript 3.0 中 flash.utils 包的一部分。 IExternalizable 接口定义一个用于对对象进行序列化 （即将对象转换为适合在设备上存储或通过网络传输的格式）的协议。

```
public interface IExternalizable
{
    function writeExternal(output:IDataOutput):void;
    function readExternal(input:IDataInput):void;
}
```

IExternalizable 接口是使用 public 访问控制修饰符声明的。只能使用 public 和 internal 访问控制说明符来修改接口定义。接口定义中的方法声明不能包含任何访问控制说明符。

ActionScript 3.0 遵循一种约定，其中接口名以大写字母 I 开头，不过您可以使用任何合法的标识符作为接口名。接口定义经常位于包的顶级。接口定义不能放在类定义或另一个接口定义中。

接口可扩展一个或多个其他接口。例如，下面的接口 IExample 扩展了 IExternalizable 接口：

```
public interface IExample extends IExternalizable
{
    function extra():void;
}
```

实现 IExample 接口的所有类不但必须包括 extra() 方法的实现，还要包括从 IExternalizable 接口继承的 writeExternal() 和 readExternal() 方法的实现。

在类中实现接口

类是唯一可实现接口的 ActionScript 3.0 语言元素。在类声明中使用 implements 关键字可实现一个或多个接口。下面的示例定义两个接口 IAlpha 和 IBeta 以及实现这两个接口的类 Alpha：

```
interface IAlpha
{
    function foo(str:String):String;
}

interface IBeta
{
    function bar():void;
}

class Alpha implements IAlpha, IBeta
{
    public function foo(param:String):String {}
    public function bar():void {}
}
```

在实现接口的类中，实现的方法必须：

- 使用 public 访问控制标识符。
- 使用与接口方法相同的名称。
- 拥有相同数量的参数，每一个参数的数据类型都要与接口方法参数的数据类型相匹配。
- 使用相同的返回类型。

```
public function foo(param:String):String {}
```

不过在命名所实现方法的参数时，您有一定的灵活性。虽然实现的方法的参数数和每个参数的数据类型必须与接口方法的参数数和数据类型相匹配，但参数名不需要匹配。例如，在上一个示例中，将 Alpha.foo() 方法的参数命名为 param：

但是，将 IAlpha.foo() 接口方法中的参数命名为 str：

```
function foo(str:String):String;
```

另外，使用默认参数值也具有一定的灵活性。接口定义可以包含使用默认参数值的函数声明。实现这种函数声明的方法必须采用默认参数值，默认参数值是与接口定义中指定的值具有相同数据类型的一个成员，但是实际值不一定匹配。例如，以下代码定义的接口包含一个使用默认参数值 3 的方法：

```
interface IGamma
{
    function doSomething(param:int = 3):void;
}
```

以下类定义实现 IGamma 接口，但使用不同的默认参数值：

```
class Gamma implements IGamma
{
    public function doSomething(param:int = 4):void {}
}
```

提供这种灵活性的原因是，实现接口的规则的设计目的是确保数据类型兼容性，因此不必要求采用相同的参数名和默认参数名，就能实现目标。

继承

继承是指一种代码重用的形式，允许程序员基于现有类开发新类。现有类通常称为基类或超类，而新类称为子类。继承的主要优势是，允许重复使用基类中的代码，但不修改现有代码。此外，继承不要求改变其他类与基类交互的方式。不必修改可能已经过彻底测试或可能已被使用的现有类，使用继承可将该类视为一个集成模块，可使用其他属性或方法对它进行扩展。因此，您使用 extends 关键字指明类从另一类继承。

通过继承还可以在代码中利用多态。有一种方法在应用于不同数据类型时会有不同行为，多态就是对这样的方法应用一个方法名的能力。名为 Shape 的基类就是一个简单的示例，该类有名为 Circle 和 Square 的两个子类。Shape 类定义了名为 area() 的方法，该方法返回形状的面积。如果已实现多态，则可以对 Circle 和 Square 类型的对象调用 area() 方法，然后执行正确的计算。使用继承能实现多态，实现的方式是允许子类继承和重新定义或覆盖 基类中的方法。在下面的示例中，由 Circle 和 Square 两个类重新定义了 area() 方法：

```
class Shape
{
    public function area():Number
    {
        return NaN;
    }
}

class Circle extends Shape
{
    private var radius:Number = 1;
    override public function area():Number
    {
        return (Math.PI * (radius * radius));
    }
}

class Square extends Shape
{
    private var side:Number = 1;
    override public function area():Number
    {
        return (side * side);
    }
}

var cir:Circle = new Circle();
trace(cir.area()); // output: 3.141592653589793
var sq:Square = new Square();
trace(sq.area()); // output: 1
```

因为每个类定义一个数据类型，所以使用继承会在基类和扩展基类的类之间创建一个特殊关系。子类保证拥有其基类的所有属性，这意味着子类的实例总是可以替换基类的实例。例如，如果方法定义了 Shape 类型的参数 (parameter)，由于 Circle 扩展了 Shape，因此 Circle 类型的参数 (argument) 是合法的，如下所示：

```
function draw(shapeToDraw:Shape) {}

var myCircle:Circle = new Circle();
draw(myCircle);
```

实例属性和继承

对于实例属性，无论是使用 function、 var，还是 const 关键字定义，只要在基类中未使用 private 属性进行声明，就可以由子类继承。例如， ActionScript 3.0 中的 Event 类具有许多子类，它们继承了所有事件对象共有的属性。

对于某些类型的事件， Event 类包含了定义事件所需的所有属性。这些类型的事件不需要 Event 类中定义的实例属性以外的实例属性。 complete 事件和 connect 事件就是这样的事件，前者在成功加载数据时发生，后者在建立网络连接时发生。

下面的示例是从 Event 类中摘录的，显示由子类继承的某些属性和方法。由于继承了属性，因此任何子类的实例都可以访问这些属性。

```
public class Event
{
    public function get type():String;
    public function get bubbles():Boolean;
    ...

    public function stopPropagation():void {}
    public function stopImmediatePropagation():void {}
    public function preventDefault():void {}
    public function isDefaultPrevented():Boolean {}
    ...
}
```

其他类型的事件需要 Event 类中没有提供的特有属性。这些事件是使用 Event 类的子类定义的，所以可向 Event 类中定义的属性添加新属性。 MouseEvent 类就是这样的一个子类，它可添加与鼠标移动或鼠标单击相关的事件的特有属性，如 mouseMove 和 click 事件。下面的示例是从 MouseEvent 类中摘录的，它说明了在子类中存在但在基类中不存在的属性的定义：

```
public class MouseEvent extends Event
{
    public static const CLICK:String= "click";
    public static const MOUSE_MOVE:String = "mouseMove";
    ...

    public function get stageX():Number {}
    public function get stageY():Number {}
    ...
}
```

访问控制说明符和继承

如果某一属性是用 public 关键字声明的，则该属性对任何位置的代码可见。这表示 public 关键字与 private、 protected 和 internal 关键字不同，它对属性继承没有任何限制。

如果属性是使用 private 关键字声明的，该属性只在定义该属性的类中可见，这表示它不能由任何子类继承。此行为与以前版本的 ActionScript 不同，在这些版本中， private 关键字的行为更类似于 ActionScript 3.0 中的 protected 关键字。

protected 关键字指出某一属性不仅在定义该属性的类中可见，而且还在所有子类中可见。与 Java 编程语言中的 protected 关键字不一样，ActionScript 3.0 中的 protected 关键字并不使属性对同一包中的所有其他类可见。在 ActionScript 3.0 中，只有子类可以访问使用 protected 关键字声明的属性。此外， protected 属性对子类可见，不管子类和基类是在同一包中，还是在不同包中。

要限制某一属性在定义该属性的包中的可见性，请使用 internal 关键字或者不使用任何访问控制说明符。未指定访问控制说明符时，应用的默认访问控制说明符是 internal 访问控制说明符。标记为 internal 的属性仅由位于同一包中的子类继承。

可以使用下面的示例来查看每一个访问控制说明符如何影响跨越包边界的继承。以下代码定义了一个名为 AccessControl 的主应用程序类和两个名为 Base 的 Extender 的其他类。Base 类在名为 foo 的包中，Extender 类是 Base 类的子类，它在名为 bar 的包中。 AccessControl 类只导入 Extender 类，并创建一个 Extender 实例，该实例试图访问在 Base 类中定义的名为 str 的变量。 str 变量将声明为 public 以使代码能够进行编译和运行，如以下摘录中所示：

```
// Base.as in a folder named foo
package foo
{
    public class Base
    {
        public var str:String = "hello"; // change public on this line
    }
}

// Extender.as in a folder named bar
package bar
{
    import foo.Base;
    public class Extender extends Base
    {
        public function getString():String {
            return str;
        }
    }
}

// main application class in file named AccessControl.as
package
{
    import flash.display.MovieClip;
    import bar.Extender;
    public class AccessControl extends MovieClip
    {
        public function AccessControl()
        {
            var myExt:Extender = new Extender();
            trace(myExt.str);// error if str is not public
            trace(myExt.getString()); // error if str is private or internal
        }
    }
}
```

要查看其他访问控制说明符如何影响先前示例的编译和执行，请在 AccessControl 类中删除或注释掉以下行，然后将 str 变量的访问控制说明符更改为 private、 protected 或 internal：

```
trace(myExt.str);// error if str is not public
```

不允许覆盖变量

将继承使用 var 或 const 关键字声明的属性，但不能对其进行覆盖。覆盖某一属性就表示在子类中重新定义该属性。唯一可覆盖的属性类型是 get 和 set 取值函数 （使用 function 关键字声明的属性）。虽然不能覆盖实例变量，但是通过为实例变量创建 getter 和 setter 方法并覆盖这些方法，可实现类似的功能。

覆盖方法

覆盖方法表示重新定义已继承方法的行为。静态方法不能继承，也不能覆盖。但是，实例方法可由子类继承，也可覆盖，只要符合以下两个条件：

- 实例方法在基类中不是使用 final 关键字声明的。当 final 关键字与实例方法一起使用时，该关键字指明程序员的设计目的是要禁止子类覆盖方法。
- 实例方法在基类中不是使用 private 访问控制说明符声明的。如果某个方法在基类中标记为 private，则在子类中定义同名方法时不需要使用 override 关键字，因为基类方法在子类中不可见。

  要覆盖符合这些条件的实例方法，子类中的方法定义必须使用 override 关键字，且必须在以下几个方面与方法的超类版本相匹配：

- 覆盖方法必须与基类方法具有相同级别的访问控制。标记为内部的方法与没有访问控制说明符的方法具有相同级别的访问控制。
- 覆盖方法必须与基类方法具有相同的参数数。
- 覆盖方法参数必须与基类方法参数具有相同的数据类型注释。
- 覆盖方法必须与基类方法具有相同的返回类型。

但是，覆盖方法中的参数名不必与基类中的参数名相匹配，只要参数数和每个参数的数据类相匹配即可。

super 语句

覆盖方法时，程序员经常希望在要覆盖的超类方法的行为上添加行为，而不是完全替换该行为。这需要通过某种机制来允许子类中的方法调用它本身的超类版本。 super 语句就提供了这样一种机制，其中包含对直接超类的引用。下面的示例定义了名为 Base 的类 （其中包含名为 thanks() 的方法），还包含名为 Extender 的 Base 类的子类 （用于覆盖 thanks() 方法）。Extender.thanks() 方法使用 super 语句调用 Base.thanks()。

```
package {
    import flash.display.MovieClip;
    public class SuperExample extends MovieClip
    {
        public function SuperExample()
        {
            var myExt:Extender = new Extender()
            trace(myExt.thanks()); // output: Mahalo nui loa
        }
    }
}

class Base {
    public function thanks():String
    {
        return "Mahalo";
    }
}

class Extender extends Base
{
    override public function thanks():String
    {
        return super.thanks() + " nui loa";
    }
}
```

覆盖 **getter** 和 **setter**

虽然不能覆盖超类中定义的变量，但是可以覆盖 getter 和 setter。例如，下面的代码用于覆盖在 ActionScript 3.0 的 MovieClip 类中定义的名为 currentLabel 的 getter:

```
package
{
    import flash.display.MovieClip;
    public class OverrideExample extends MovieClip
    {
        public function OverrideExample()
        {
            trace(currentLabel)
        }
        override public function get currentLabel():String
        {
            var str:String = "Override: ";
            str += super.currentLabel;
            return str;
        }
    }
}
```

OverrideExample 类构造函数中的 trace() 语句的输出是 Override: null，这说明该示例能够覆盖继承的 currentLabel 属性。

不继承静态属性

静态属性不由子类继承。这意味着不能通过子类的实例访问静态属性。只能通过定义静态属性的类对象来访问静态属性。例如，以下代码定义了名为 Base 的基类和扩展名为 Extender 的 Base 子类。Base 类中定义了名为 test 的静态变量。以下摘录中编写的代码在严格模式下不会进行编译，在标准模式下会生成运行时错误。

```
package {
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// error
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base { }
```

访问静态变量 test 的唯一方式是通过类对象，如以下代码所示：

```
Base.test;
```

但允许使用与静态属性相同的名称定义实例属性。可以在与静态属性相同的类中或在子类中定义这样的实例属性。例如，以上示例中的 Base 类可以具有一个名为 test 的实例属性。在以下代码中，由于 Extender 类继承了实例属性，因此代码能够进行编译和执行。如果 test 实例变量的定义移到 （不是复制） Extender 类，该代码也会编译并执行。

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
            trace(myExt.test);// output: instance
        }
    }
}

class Base
{
    public static var test:String = "static";
    public var test:String = "instance";
}

class Extender extends Base {}
```

静态属性和作用域链

虽然并不继承静态属性，但是静态属性在定义它们的类或该类的任何子类的作用域链中。同样，可以认为静态属性在定义它们的类和任何子类的作用域 中。这意味着在定义静态属性的类体及该类的任何子类中可直接访问静态属性。

下面的示例对上一个示例中定义的类进行了修改，以说明 Base 类中定义的 test 静态变量在 Extender 类的作用域中。换句话说，Extender 类可以访问 test 静态变量，而不必用定义 test 的类名作为变量的前缀。

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base {
    public static var test:String = "static";
}

class Extender extends Base
{
    public function Extender()
    {
        trace(test); // output: static
    }

}
```

如果使用与同类或超类中的静态属性相同的名称定义实例属性，则实例属性在作用域链中的优先级比较高。因此认为实例属性遮蔽了静态属性，这意味着会使用实例属性的值，而不使用静态属性的值。例如，以下代码显示如果 Extender 类定义名为 test 的实例变量， trace() 语句将使用实例变量的值，而不使用静态变量的值:

```
package
{
    import flash.display.MovieClip;
    public class StaticExample extends MovieClip
    {
        public function StaticExample()
        {
            var myExt:Extender = new Extender();
        }
    }
}

class Base
{
    public static var test:String = "static";
}

class Extender extends Base
{
    public var test:String = "instance";
    public function Extender()
    {
        trace(test); // output: instance
    }

}
```

高级主题

ActionScript OOP 支持的历史

由于 ActionScript 3.0 是在以前版本的 ActionScript 基础上构建的，了解 ActionScript 对象模型的发展过程可能有所帮助。ActionScript 最初作为早期版本的 Flash Professional 的简单编写脚本机制。后来，编程人员开始使用 ActionScript 构建更加复杂的应用程序。为了迎合这些程序员的需要，每个后续版本都添加了一些语言功能以帮助创建复杂的应用程序。

ActionScript 1.0

ActionScript 1.0 是在 Flash Player 6 和更早版本中使用的语言版本。即使在这个早期开发阶段，ActionScript 对象模型也是建立在基础数据类型对象的概念的基础上。 ActionScript 对象是由一组属性 构成的复合数据类型。讨论对象模型时，术语属性 包括附加到对象的所有内容，如变量、函数或方法。

尽管第一代 ActionScript 不支持使用 class 关键字定义类，但是可以使用称为原型对象的特殊对象来定义类。 Java 和 C++ 等基于类的语言中使用 class 关键字创建要实例化为具体对象的抽象类定义，而 ActionScript 1.0 等基于原型的语言则将现有对象用作其他对象的模型 （或原型）。基于类的语言中的对象可能指向作为其模板的类，而基于原型的语言中的对象则指向作为其模板的另一个对象 （即其原型）。

要在 ActionScript 1.0 中创建类，可以为该类定义一个构造函数。在 ActionScript 中，函数不只是抽象定义，还是实际对象。您创建的构造函数用作该类实例的原型对象。以下代码创建一个名为 Shape 的类，还定义一个名为 visible 的属性，该属性默认情况下设置为 true：

```
// base class
function Shape() {}
// Create a property named visible.
Shape.prototype.visible = true;
```

此构造函数定义可以使用 new 运算符实例化的 Shape 类，如下所示：

```
myShape = new Shape();
```

就像 Shape() 构造函数对象用作 Shape 类实例的原型一样，它还可以用作 Shape 的子类 （即扩展 Shape 类的其他类）的原型。

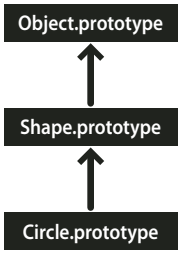
创建作为 Shape 类的子类的类的过程分两个步骤。首先，通过定义类的构造函数创建该类，如下所示：

```
// child class
function Circle(id, radius)
{
this.id = id;
this.radius = radius;
}
```

然后，使用 new 运算符将 Shape 类声明为 Circle 类的原型。默认情况下，创建的所有类都使用 Object 类作为其原型，这意味着 Circle.prototype 当前包含一个通用对象 （Object 类的实例）。要指定 Circle 的原型是 Shape 而不是 Object，请使用以下代码更改 Circle.prototype 的值，使其包含 Shape 对象而不是通用对象。

```
// Make Circle a subclass of Shape.
Circle.prototype = new Shape();
```

Shape 类和 Circle 类现在通过通常所说的原型链 的继承关系联系在一起。下图说明了原型链中的关系：



每个原型链末端的基类是 Object 类。 Object 类包含一个名为 Object.prototype 的静态属性，该属性指向在 ActionScript 1.0 中创建的所有对象的基原型对象。该示例原型链中的下一个对象是 Shape 对象。这是因为从不显式设置 Shape.prototype 属性，所以它仍然包含通用对象 （Object 类的实例）。此链中的最后一个链环是 Circle 类，该类链接到其原型 Shape 类（Circle.prototype 属性包含 Shape 对象）。

如果您创建 Circle 类的实例 （如以下示例所示），该实例会继承 Circle 类的原型链：

```
// Create an instance of the Circle class.
myCircle = new Circle();
```

回想一下，该示例包括一个名为 visible 的属性作为 Shape 类的成员。在本示例中， visible 属性并不作为 myCircle 对象的一部分，只是 Shape 对象的一个成员，但以下代码行的输出却为 true：

```
trace(myCircle.visible); // output: true
```

运行时能够沿着原型链检查 myCircle 对象是否继承了 visible 属性。执行此代码时，运行时首先在 myCircle 对象的属性中搜索名为 visible 的属性，但未发现这样的属性。运行时接下来在下一个 Circle.prototype 对象中查找，仍未发现名为 visible 的属性。运行时继续检查原型链，最终发现了在 Shape.prototype 对象上定义的 visible 属性，并输出该属性的值。

为简单起见，省略了原型链的许多细节和复杂性。目标只是提供足够信息来帮助您了解 ActionScript 3.0 对象模型。

ActionScript 2.0

在 ActionScript 2.0 中引入了 class、extends、public 和 private 等新关键字，通过使用这些关键字，您可以按 Java 和 C++ 等基于类的语言用户所熟悉的方式来定义类。ActionScript 1.0 和 ActionScript 2.0 中的基础继承机制是相同的，了解这一点很重要。 ActionScript 2.0 只是增加了定义类的新语法。在该语言的两个版本中，原型链的作用方式是一样的。

ActionScript 2.0 中引入了新语法 （如以下摘录中所示），可允许以多数程序员认为更直观的方式定义类：

```
// base class
class Shape
{
var visible:Boolean = true;
}
```

注意， ActionScript 2.0 还引入了用于编译时类型检查的类型注释。使用类型注释，可以将上个示例中的 visible 属性声明为应只包含布尔值。新 extends 关键字还简化了创建子类的过程。在下面的示例中，通过使用 extends 关键字，可以一步完成在 ActionScript 1.0 中需要分两步完成的过程：

```
// child class
class Circle extends Shape
{
    var id:Number;
    var radius:Number;
    function Circle(id, radius)
    {
        this.id = id;
        this.radius = radius;
        }
}
```

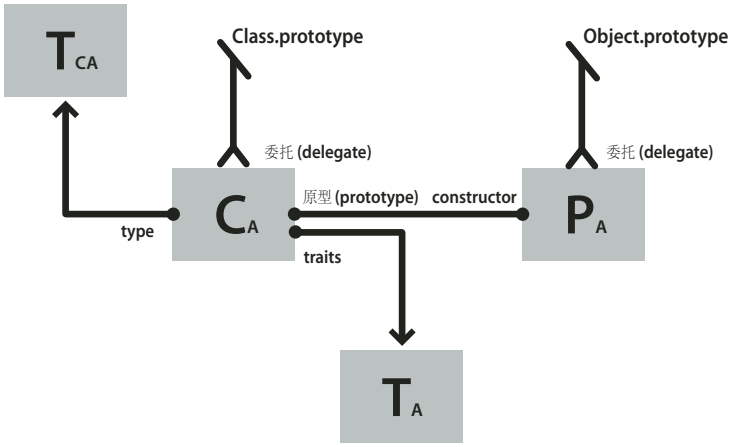
构造函数现在声明为类定义的一部分，还必须显式声明类属性 id 和 radius。

ActionScript 2.0 中还增加了对接口定义的支持，以使您能够使用为对象间通信正式定义的协议来进一步改进面向对象的程序。

ActionScript 3.0 类对象

常见的面向对象的编程范例多数常与 Java 和 C++ 相关联，这种范例使用类定义对象的类型。采用这种范例的编程语言也趋向使用类来构造类定义的数据类型的实例。 ActionScript 使用类是为了实现以上两个目的，但其根本还是一种基于原型的语言，并带有有趣的特征。 ActionScript 为每一类定义创建了特殊的类对象，允许共享行为和状态。但是，对多数 ActionScript 程序员而言，这个特点可能与实际编码没有什么牵连。 ActionScript 3.0 的设计目的是，不使用 （甚至是不必了解）这些特殊类对象，就可创建复杂的面向对象的 ActionScript 应用程序。

下图显示一个类对象的结构，该类对象表示使用语句 class A {} 定义的名为 A 的简单类：



图中的每个矩形表示一个对象。图中的每一个对象都有下标字符 A，这表示该对象属于类 A。类对象 (CA) 包含对许多其他重要对象的引用。实例 traits 对象 (TA) 用于存储在类定义中定义的实例属性。类 traits 对象 (TCA) 表示类的内部类型，用于存储该类定义的静态属性 （下标字符 C 代表 “ 类 ”）。原型对象 (PA) 始终指的是最初通过 constructor 属性附加到的类对象。

traits 对象

traits 对象是 ActionScript 3.0 中的新增对象，它是为了提高性能而实现的。在以前版本的 ActionScript 中，名称查找是一个耗时的过程，因为 Flash Player 要搜索原型链。在 ActionScript 3.0 中，名称查找更有效、耗时更少，因为可以将继承属性从超类复制到子类的 traits 对象。

编程代码不能直接访问 traits 对象，但是性能和内存使用情况的改善可反映它的存在。traits 对象给 AVM2 提供了关于类的布局和内容的详细信息。借助这些信息，AVM2 可显著减少执行时间，因为它可以经常生成直接机器指令来直接访问属性或直接调用方法，而省去了查找名称所耗费的时间。

由于使用了 traits 对象，与以前版本中 ActionScript 类似对象相比，该版本中对象占用内存的时间明显减少。例如，如果某个类已密封 （即，该类未声明为 dynamic），则该类实例不需要动态添加属性的哈希表，只保留一个到 traits 对象的指针和该类中定义的固定属性的某些位置。因此，如果对象在 ActionScript 2.0 中需要占用 100 个字节的内存，在 ActionScript 3.0 中只需要占用 20 个字节的内存。

注：traits 对象是内部实现详细信息，不保证在将来版本的 ActionScript 中此对象不更改，甚至消失。

原型对象

每个 ActionScript 类对象都有一个名为 prototype 的属性，它表示对类的原型对象的引用。ActionScript 根本上是基于原型的语言，原型对象是旧内容。有关详细信息，请参阅 ActionScript OOP 支持的历史。

prototype 属性是只读属性，这表示不能将其修改为指向其他对象。这不同于以前版本 ActionScript 中的类 prototype 属性，在以前版本中可以重新分配 prototype，使它指向其他类。虽然 prototype 属性是只读属性，但是它所引用的原型对象不是只读的。换句话说，可以向原型对象添加新属性。向原型对象添加的属性可在类的所有实例中共享。

原型链是以前版本的 ActionScript 的唯一继承机制，在 ActionScript 3.0 中，原型链只具有辅助作用。主要的继承机制 （固定属性）是由 traits 对象在内部处理的。固定属性指的是定义为类定义的一部分的变量或方法。固定属性继承也叫做类继承，因为它是与 class、 extends 和 override 等关键字相关的继承机制。

原型链提供了另一种继承机制，该机制的动态性比固定属性继承的更强。既可以将属性作为类定义的一部分，也可以在运行时通过类对象的 prototype 属性向类的原型对象中添加属性。但是，请注意，如果将编译器设置为严格模式，则不能访问添加到原型对象中的属性，除非使用 dynamic 关键字声明类。

Object 类就是这样类的示例，它的原型对象附加了若干属性。 Object 类的 toString() 和 valueOf() 方法实际上是一些函数，它们分配给 Object 类原型对象的属性。以下是一个示例，说明这些方法的声明理论上是怎样的 （实际实现时会因实现详细信息而稍有不同）：

```
public dynamic class Object
{
    prototype.toString = function()
    {
        // statements
    };
    prototype.valueOf = function()
    {
        // statements
    };
}
```

如前所述，可以将属性附加到类定义外部的类原型对象。例如，也可以在 Object 类定义外部定义 toString() 方法，如下所示：

```
Object.prototype.toString = function()
{
    // statements
};
```

但是，原型继承与固定属性继承不一样，如果要重新定义子类中的方法，原型继承不需要 override 关键字。例如 . 如果要重新定义 Object 类的子类中的 valueOf() 方法，您有以下三种选择。第一，可以在类定义中的子类原型对象上定义 valueOf() 方法。以下代码创建一个名为 Foo 的 Object 子类，并将 Foo 原型对象的 valueOf() 方法重新定义为类定义的一部分。因为每个类都是从 Object 继承的，所以不需要使用 extends 关键字。

```
dynamic class Foo
{
    prototype.valueOf = function()
    {
        return "Instance of Foo";
    };
}
```

第二，可以在类定义外部对 Foo 原型对象定义 valueOf() 方法，如以下代码中所示：

```
Foo.prototype.valueOf = function()
{
    return "Instance of Foo";
};
```

第三，可以将名为 valueOf() 的固定属性定义为 Foo 类的一部分。这种方法与其他混合了固定属性继承与原型继承的方法有所不同。要重新定义 valueOf() 的 Foo 的任何子类必须使用 override 关键字。以下代码显示 valueOf() 定义为 Foo 中的固定属性：

```
class Foo
{
    function valueOf():String
    {
        return "Instance of Foo";
    }
}
```

AS3 命名空间

由于存在两种继承机制，即固定属性继承和原型继承，所以涉及到核心类的属性和方法时，就存在两种机制的兼容性问题。如果与 ActionScript 所基于的 ECMAScript 语言规范兼容，则要求使用原型继承，这意味着核心类的属性和方法是在该类的原型对象上定义的。另一方面，如果与 ActionScript 3.0 兼容，则要求使用固定属性继承，这意味着核心类的属性和方法是使用 const、 var 和 function 关键字在类定义中定义的。此外，如果使用固定属性而不是原型属性，将显著提升运行时性能。

在 ActionScript 3.0 中，通过同时将原型继承和固定属性继承用于核心类，解决了这个问题。每一个核心类都包含两组属性和方法。一组是在原型对象上定义的，用于与 ECMAScript 规范兼容，另一组使用固定属性定义和 AS3 命名空间定义，以便与 ActionScript 3.0 兼容。

AS3 命名空间提供了一种约定机制，用来在两组属性和方法之间做出选择。如果不使用 AS3 命名空间，核心类的实例会继承在核心类的原型对象上定义的属性和方法。如果决定使用 AS3 命名空间，核心类的实例会继承 AS3 版本，因为固定属性的优先级始终高于原型属性。换句话说，只要固定属性可用，则始终使用固定属性，而不使用同名的原型属性。

通过用 AS3 命名空间限定属性或方法，可以选择使用 AS3 命名空间版本的属性或方法。例如，下面的代码使用 AS3 版本的 Array.pop() 方法：

```
var nums:Array = new Array(1, 2, 3);
nums.AS3::pop();
trace(nums); // output: 1,2
```

或者，也可以使用 use namespace 指令打开代码块中所有定义的 AS3 命名空间。例如，以下代码使用 use namespace 指令为 pop() 和 push() 方法打开 AS3 命名空间：

```
use namespace AS3;

var nums:Array = new Array(1, 2, 3);
nums.pop();
nums.push(5);
trace(nums) // output: 1,2,5
```

ActionScript 3.0 还为每组属性提供了编译器选项，以便将 AS3 命名空间应用于整个程序。 -as3 编译器选项表示 AS3 命名空间，-es 编译器选项表示原型继承选项（es 代表 ECMAScript）。要打开整个程序的 AS3 命名空间，请将 -as3 编译器选项设置为 true，将 -es 编译器选项设置为 false。要使用原型版本，请将编译器选项设置为相反值。 Flash Builder 和 Flash Professional 的默认编译器设置是 -as3 = true， -es = false。

如果计划扩展任何核心类并覆盖任何方法，应了解 AS3 命名空间对声明覆盖方法的方式有什么影响。如果要使用 AS3 命名空间，覆盖核心类方法的任何方法都必须使用 AS3 命名空间以及 override 属性。如果不打算使用 AS3 命名空间且要重新定义子类中的核心类方法，则不应使用 AS3 命名空间或 override 关键字。

示例：GeometricShapes

GeometricShapes 范例应用程序说明了如何使用 ActionScript 3.0 来应用很多面向对象的概念和功能，其中包括：

- 定义类
- 扩展类
- 多态和 override 关键字
- 定义、扩展和实现接口

示例中还包括一个用于创建类实例的“工厂方法”，说明如何将返回值声明为接口的实例，以及通过一般方法使用返回的对象。

若要获取此范例的应用程序文件，请参阅 www.adobe.com/go/learn_programmingAS3samples_flash_cn。在 Samples/GeometricShapes 文件夹下可找到 GeometricShapes 应用程序文件。该应用程序包含以下文件：

文件	说明
GeometricShapes.mxml 或 GeometricShapes.fla	Flash 或 Flex 中的主应用程序文件（分别为 FLA 和 MXML）。
com/example/programmingas3/geometricshapes/IGeometricShape.as	由所有 GeometricShapes 应用程序类实现的基接口定义方法。
com/example/programmingas3/geometricshapes/IPolygon.as	由有多条边的 GeometricShapes 应用程序类实现的接口定义方法。
com/example/programmingas3/geometricshapes/RegularPolygon.as	一种几何形状，这种几何形状的边长相等，并且这些边围绕形状中心对称分布。
com/example/programmingas3/geometricshapes/Circle.as	一种用于定义圆的几何形状。
com/example/programmingas3/geometricshapes/EquilateralTriangle.as	RegularPolygon 的子类，用于定义所有边长相等的三角形。
com/example/programmingas3/geometricshapes/Square.as	RegularPolygon 的子类，用于定义所有四条边相等的矩形。
com/example/programmingas3/geometricshapes/GeometricShapeFactory.as	包含工厂方法的一个类，用于创建给定了形状类型和尺寸的形状。

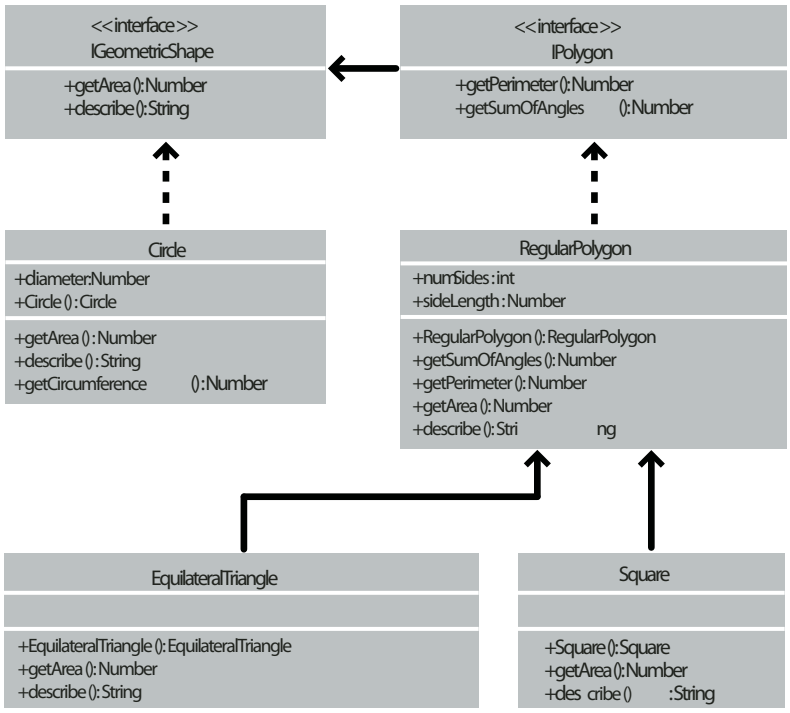
定义 GeometricShapes 类

使用 GeometricShapes 应用程序可以允许用户指定几何形状的类型和尺寸。随后，应用程序通过形状描述、其面积和周长进行响应。

应用程序的用户界面显得很零碎，其中包括一些可用于选择形状类型、设置尺寸及显示描述的控件。这个应用程序最引人注意的部分是程序的内部，即在类和接口本身的结构中。

这个应用程序用于处理几何形状，但是它并不以图形化方式显示这些形状。

在下图中使用统一建模语言 (UML) 表示法显示此示例中用于定义几何形状的类和接口：



GeometricShapes 示例类

定义接口的通用行为

这个 GeometricShapes 应用程序处理三种形状：圆、正方形和等边三角形。 GeometricShapes 类结构以非常简单的接口 IGeometricShape 作为开始，它列出通用于所有这三种形状的方法。

```
package com.example.programmingas3.geometricshapes
{
    public interface IGeometricShape
    {
        function getArea():Number;
        function describe():String;
    }
}
```

该接口定义两个方法：getArea() 方法和 describe() 方法，前者计算并返回形状的面积，后者汇编形状属性的文本描述。

每个形状的周长也是我们希望知道的。但是，圆的周边长度叫做周长，它的计算方式是独有的，所以其行为异于三角形或正方形的行为。不过，三角形、正方形和其他多边形之间仍然有很多类似之处，所以为它们定义一个新接口类 IPolygon 还是很有意义的。 IPolygon 接口也相当简单，如下所示：

```
package com.example.programmingas3.geometricshapes
{
    public interface IPolygon extends IGeometricShape
    {
        function getPerimeter():Number;
        function getSumOfAngles():Number;
    }
}
```

这个接口定义了所有多边形的两个通用方法：getPerimeter() 方法和 getSumOfAngles() 方法，前者用于计算所有边相加后的总长，后者用于将所有内角相加起来。

IPolygon 接口扩展了 IGeometricShape 接口，这意味着实现 IPolygon 接口的所有类必须声明所有四个方法，即来自 IGeometricShape 接口的两个方法和来自 IPolygon 接口的两个方法。

定义形状类

对每种形状的方法有所了解后，就可以定义形状类本身了。就需要实现的方法数而言，最简单的形状是 Circle 类，如下所示：

```
package com.example.programmingas3.geometricshapes
{
    public class Circle implements IGeometricShape
    {
        public var diameter:Number;

        public function Circle(diam:Number = 100):void
        {
            this.diameter = diam;
        }

        public function getArea():Number
        {
            // The formula is Pi * radius * radius.
            var radius:Number = diameter / 2;
            return Math.PI * radius * radius;
        }

        public function getCircumference():Number
        {
            // The formula is Pi * diameter.
            return Math.PI * diameter;
        }

        public function describe():String
        {
            var desc:String = "This shape is a Circle.\n";
            desc += "Its diameter is " + diameter + " pixels.\n";
            desc += "Its area is " + getArea() + ".\n";
            desc += "Its circumference is " + getCircumference() + ".\n";
            return desc;
        }
    }
}
```

Circle 类用于实现 IGeometricShape 接口，因此，它必须为 getArea() 方法和 describe() 方法提供代码。此外，它还定义 getCircumference() 方法，该方法对 Circle 类是唯一的。另外， Circle 类还声明属性 diameter，其他多边形类没有该属性。

其他两种形状，即正方形和等边三角形的某些其他方面是共同的：它们各自的边长相等，各自都有可用于计算周长和内角总和的通用公式。实际上，这些通用公式还适用于将来定义的任何其他正多边形。

RegularPolygon 类是 Square 和 EquilateralTriangle 两个类的超类。使用超类可在一个位置定义通用方法，所以不必在每个子类中单独定义方法。以下是 RegularPolygon 类的代码：

```
package com.example.programmingas3.geometricshapes
{
    public class RegularPolygon implements IPolygon
    {
        public var numSides:int;
        public var sideLength:Number;

        public function RegularPolygon(len:Number = 100, sides:int = 3):void
        {
            this.sideLength = len;
            this.numSides = sides;
        }

        public function getArea():Number
        {
            // This method should be overridden in subclasses.
            return 0;
        }

        public function getPerimeter():Number
        {
            return sideLength * numSides;
        }

        public function getSumOfAngles():Number
        {
            if (numSides >= 3)
            {
                return ((numSides - 2) * 180);
            }
            else
            {
                return 0;
            }
        }

        public function describe():String
        {
            var desc:String = "Each side is " + sideLength + " pixels long.\n";
            desc += "Its area is " + getArea() + " pixels square.\n";
            desc += "Its perimeter is " + getPerimeter() + " pixels long.\n";
            desc += "The sum of all interior angles in this shape is " + getSumOfAngles() + " degrees.\n";
            return desc;
        }
    }
}
```

第一， RegularPolygon 类用于声明所有正多边形的两个通用属性：每边的长度 （sideLength 属性）和边数 （numSides 属性）。

RegularPolygon 类实现 IPolygon 接口，还声明 IPolygon 接口的所有四个方法。它使用通用公式来实现 getPerimeter() 和 getSumOfAngles() 两种方法。

因为用于 getArea() 方法的公式因形状的不同而有所不同，所以该方法的基类版本不能包括可由子类方法继承的通用逻辑，而是只返回默认值 0 以指明未计算面积。要正确计算每个形状的面积， RegularPolygon 类的子类本身必须覆盖 getArea() 方法本身。

EquilateralTriangle 类的以下代码说明如何覆盖 getArea() 方法：

```
package com.example.programmingas3.geometricshapes
{
    public class EquilateralTriangle extends RegularPolygon
    {
        public function EquilateralTriangle(len:Number = 100):void
        {
            super(len, 3);
        }

        public override function getArea():Number
        {
        // The formula is ((sideLength squared) * (square root of 3)) / 4.
        return ( (this.sideLength * this.sideLength) * Math.sqrt(3) ) / 4;
        }

        public override function describe():String
        {
                /* starts with the name of the shape, then delegates the rest
                of the description work to the RegularPolygon superclass */
        var desc:String = "This shape is an equilateral Triangle.\n";
        desc += super.describe();
        return desc;
        }
    }
}
```

override 关键字指示 EquilateralTriangle.getArea() 方法有意覆盖 RegularPolygon 超类中的 getArea() 方法。当调用 EquilateralTriangle.getArea() 方法时，该方法使用先前代码中的公式计算面积，从不执行 RegularPolygon.getArea() 方法中的代码。

不同的是， EquilateralTriangle 类并不定义它自己版本的 getPerimeter() 方法。当调用 EquilateralTriangle.getPerimeter() 方法时，调用会沿继承链找到 RegularPolygon 超类的 getPerimeter() 方法，并执行其中的代码。

EquilateralTriangle() 构造函数使用 super() 语句显式地调用其超类的 RegularPolygon() 构造函数。如果这两个构造函数使用同一组参数，则可以完全省略 EquilateralTriangle() 构造函数，而执行 RegularPolygon() 构造函数。但是， RegularPolygon() 构造函数需要额外的参数，即 numSides。因此 EquilateralTriangle() 构造函数调用 super(len,3)，该语句传递 len 输入参数和值 3，以表示三角形有三个边。

describe() 方法还使用 super() 语句，但使用方式不同。它使用该语句调用 describe() 方法的 RegularPolygon 超类版本。EquilateralTriangle.describe() 方法先将 desc 字符串变量设置为有关形状类型的语句。然后调用 super.describe() 来获取 RegularPolygon.describe() 方法的结果，之后将结果追加到 desc 字符串。

此处不详细讨论 Square 类，但它类似于 EquilateralTriangle 类，提供构造函数及构造函数自己的 getArea() 和 describe() 方法的实现。

多态和工厂方法

可以采用许多有趣的方法正确使用接口的一组类及继承。例如，到目前为止讨论的所有形状类不是用于实现 IGeometricShape 接口，就是用于扩展执行的超类。因此，如果将一个变量定义为 IGeometricShape 实例，您不必知道它实际是 Circle 类的实例还是 Square 类的实例，就可以调用它的 describe() 方法。

以下代码说明这是如何实现的：

```
var myShape:IGeometricShape = new Circle(100);
trace(myShape.describe());
```

当调用 myShape.describe() 时，它执行 Circle.describe() 方法，因为即使将变量定义为 IGeometricShape 接口的实例， Circle 也是它的基础类。

这个示例说明了运行时多态的原则：完全相同的方法调用会导致执行不同的代码，这取决于要调用了其方法的对象的类。

GeometricShapes 应用程序会使用设计模式的简化版本 （叫做“工厂方法”）应用这种基于接口的多态原则。术语工厂方法指的是一个函数，该函数返回一个对象，其基本数据类型或内容可能会因上下文而不同。

以下显示的 GeometricShapeFactory 类定义一个名为 createShape() 的工厂方法：

```
package com.example.programmingas3.geometricshapes
{
    public class GeometricShapeFactory
    {
        public static var currentShape:IGeometricShape;

        public static function createShape(shapeName:String,
                                           len:Number):IGeometricShape
        {
            switch (shapeName)
            {
                case "Triangle":
                    return new EquilateralTriangle(len);

                case "Square":
                    return new Square(len);

                case "Circle":
                    return new Circle(len);
            }
            return null;
        }

        public static function describeShape(shapeType:String, shapeSize:Number):String
        {
            GeometricShapeFactory.currentShape =
                GeometricShapeFactory.createShape(shapeType, shapeSize);
            return GeometricShapeFactory.currentShape.describe();
        }
    }
}
```

createShape() 工厂方法允许形状子类构造函数定义所创建的实例的详细信息，同时返回新对象作为 IGeometricShape 实例，以便应用程序可以采用更通用的方式处理这些对象。

前面示例中的 describeShape() 方法说明应用程序如何使用工厂方法来获取对更具体对象的一般引用。该应用程序可通过以下方式获取新创建的 Circle 对象的描述：

```
GeometricShapeFactory.describeShape("Circle", 100);
```

然后，describeShape() 方法使用相同的参数调用 createShape() 工厂方法，以将新的 Circle 对象存储在名为 currentShape 的静态变量 （该变量的类型是 IGeometricShape 对象）中。接下来，对 currentShape 调用 describe() 方法，该调用将被自动解析以执行 Circle.describe() 方法，从而返回圆的详细描述。

增强范例应用程序

增强或更改应用程序后，接口和继承的实际作用会非常明显。

假定要在这个示例应用程序中添加新形状，即一个五边形。您需要创建一个 Pentagon 类，以扩展 RegularPolygon 类并定义它自己版本的 getArea() 和 describe() 方法。然后，在应用程序用户界面的组合框中添加一个新 Pentagon 选项。就是这样，Pentagon 类将通过继承来自动获取 RegularPolygon 类的 getPerimeter() 方法和 getSumOfAngles() 方法的功能。由于该类是

从实现 IGeometricShape 接口的类继承的，因此 Pentagon 实例也可以视为 IGeometricShape 实例。这意味着，要添加新形状类型，不需要更改 GeometricShapeFactory 类中的任何方法的方法签名（因而也不需要更改使用 GeometricShapeFactory 类的任何代码）。

您可能希望实践一下，将 Pentagon 类添加到 Geometric Shapes 示例，以了解向应用程序中添加新功能时，使用接口和继承将如何减轻工作量。