

优化 ADOBE® FLASH® PLATFORM 的性能

法律声明

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

目录

第 1 章：简介

运行时代码执行基本原理	1
感知性能与实际性能	2
实现优化	2

第 2 章：节省内存

显示对象	4
原始类型	4
重用对象	5
释放内存	10
使用位图	11
滤镜和动态位图卸载	17
直接进行 mip 映射	18
使用 3D 效果	18
文本对象和内存	19
事件模型与回调	20

第 3 章：最大程度减小 CPU 使用量

针对 CPU 使用量的 Flash Player 10.1 增强功能	21
睡眠模式	23
冻结和解冻对象	24
激活和停用事件	27
鼠标交互	27
计时器与 ENTER_FRAME 事件	28
补间症状	30

第 4 章：ActionScript 3.0 性能

Vector 类和 Array 类	31
绘图 API	32
事件捕获和冒泡	33
处理像素	34
正则表达式	36
其他优化	36

第 5 章：呈现性能

重绘区域	41
后台内容	42
影片品质	43
Alpha 混合	45
应用程序帧速率	45

位图缓存	46
手动位图缓存	52
呈现文本对象	58
GPU	62
异步操作	64
透明窗口	65
矢量形状的平滑处理	66
 第 6 章：优化网络交互	
针对网络交互的增强功能	67
外部内容	68
输入 / 输出错误	70
Flash Remoting	71
不必要的网络操作	72
 第 7 章：处理媒体	
视频	73
StageVideo	73
音频	73
 第 8 章：SQL 数据库性能	
针对数据库性能的应用程序设计	74
数据库文件优化	76
不必要的数据库运行时处理	76
有效的 SQL 语法	77
SQL 语句性能	78
 第 9 章：基准测试和部署	
基准测试	79
部署	80

第 1 章：简介

Adobe® AIR® 和 Adobe® Flash® Player 应用程序在许多平台上运行，包括台式机、移动设备、平板电脑和电视设备。本文档通过代码示例和使用案例，为部署这些应用程序的开发人员概括介绍了最佳做法。主题包括：

- 节省内存
- 最大程度减小 CPU 使用量
- 提高 ActionScript 3.0 性能
- 加快呈现速度
- 优化网络交互
- 使用音频和视频
- 优化 SQL 数据库性能
- 基准测试和部署应用程序

其中的大多数优化在 AIR 运行时和 Flash Player 运行上适用于所有设备上的应用程序。文档中还讨论了针对特定设备的新增项和例外情况。

其中的一些优化侧重于在 Flash Player 10.1 和 AIR 2.5 中引用的功能。但是，其中的许多优化也适用于早期版本的 AIR 和 Flash Player。

运行时代码执行基本原理

了解如何改进应用程序性能的关键是了解 Flash Platform 运行时如何执行代码。运行时在一个循环中运行，其中某些操作是针对每个“帧”发生的。在这种情况下，帧只是由为应用程序指定的帧速率决定的一段时间。分配给每个帧的时间直接对应于帧速率。例如，如果指定帧速率为 30 帧 / 秒，则运行时会尝试使每个帧的执行时间为三十分之一秒。

可以在创作应用程序时指定初始帧速率。可以使用 Adobe® Flash® Builder™ 或 Flash Professional 中的设置来设置帧速率。还可以在代码中指定初始帧速率。可通过对根文档类应用 [SWF(frameRate="24")] 元数据标签，在纯 ActionScript 应用程序中设置帧速率。在 MXML 中，可在 Application 或 WindowedApplication 标签中设置 frameRate 属性。

每个帧循环包括两个阶段，分为三部分：事件、enterFrame 事件和呈现。

第一阶段包括两部分（事件和 enterFrame 事件），这两部分都可能会导致调用代码。在第一阶段的第一部分，发生并调度运行时事件。这些事件可以表示异步操作完成或异步操作的进度，例如来自网络加载数据的响应。还包括来自用户输入的事件。调度事件时，运行时在您已注册的侦听器中执行代码。如果没有发生事件，运行时等待完成此执行阶段，而不会执行任何动作。由于缺少活动，运行时永远不会提高帧速率。如果在执行周期的其他部分发生事件，运行时将这些事件排队并在下一个帧中进行调度。

第一阶段的第二部分是 enterFrame 事件。此事件与其他事件不同，因为每个帧始终只调度一次该事件。

调度所有事件后，帧循环的呈现阶段开始。此时，运行时将计算屏幕上所有可见元素的状态并将其绘制到屏幕上。然后，此进程重复，就像赛跑者围绕跑道奔跑。

注：对于包括 updateAfterEvent 属性的事件，可强制立即执行呈现操作，而不是等到呈现阶段。但是，如果 updateAfterEvent 频繁导致性能问题，应避免使用它。

最简单的方法是假设帧循环中的两个阶段需要相同的时间。在这种情况下，一半的时间运行帧循环事件处理函数和应用程序代码，另一半的时间发生呈现。但是，事实通常是不同的。有时，应用程序代码会通过增加运行时间并减少用于呈现的时间来占用帧中多一半的可用时间。在其他情况下，特别是对于滤镜和混合模式等复杂的可见内容，呈现需要一半以上的帧时间。由于各阶段需要的实际时间是灵活的，所以帧循环常称为“弹性跑道”。

如果帧循环的组合操作（代码执行和呈现）所需时间太长，运行时将无法保持帧速率。帧会进行扩展，所需时间超过其分配的时间，因此在触发下个帧之前会出现延迟。例如，如果一个帧循环需要的时间超过三十分之一秒，则运行时不能以每秒三十帧的速度更新屏幕。如果帧速率减慢，将影响体验效果。最乐观的情况是动画断断续续。如果情况更糟糕，应用程序将冻结，并且窗口一片空白。

有关 Flash Platform 运行时代码执行和呈现模型的详细信息，请参阅下列资源：

- [Flash Player Mental Model - The Elastic Racetrack](#)（文章作者：Ted Patrick）
- [Asynchronous ActionScript Execution](#)（文章作者：Trevor McCauley）
- 优化 Adobe AIR 的代码执行、内存和呈现（网址为 http://www.adobe.com/go/learn_fp_air_perf_tv_cn）（由 Sean Christmann 撰写的 MAX 会议演示文稿的视频）

感知性能与实际性能

应用程序性能的最终判定者是该应用程序的用户。开发人员可以根据某些操作的运行时间或创建的对象实例的数量来度量应用程序性能。但是，这些度量标准对最终用户来说不重要。有时，用户通过不同标准度量性能。例如，应用程序是否快速顺利地执行，是否快速响应输入？它是否会对系统的性能产生负面影响？自测下列感知性能测试的问题：

- 动画是流畅还是断断续续？
- 观看视频内容时是流畅还是断断续续？
- 音频剪辑是连续播放还是暂停再恢复播放？
- 在时间较长的操作期间，窗口是否会闪烁或变成空白？
- 键入时，文本输入保持同步还是有些延迟？
- 如果单击，会立即发生某些情况还是存在延迟？
- 应用程序运行时，CPU 风扇声音是否会变大？
- 在便携式计算机或移动设备上运行应用程序时，电池是否会很快耗尽？
- 此应用程序运行时，其他应用程序是否响应效果很差？

感知性能和实际性能之间的区别很重要。达到最佳感知性能的方式与获得绝对最快性能的方式并不始终相同。请确保应用程序不执行太多会使运行时无法频繁更新屏幕和收集用户输入的代码。在某些情况下，达到此平衡需要将程序任务拆分成几部分，以便运行时在各部分之间更新屏幕。（有关特定指导信息，请参阅第 41 页的“[呈现性能](#)”。）

此处介绍的技巧和技术旨在改进实际代码执行性能和用户感知性能的方法。

实现优化

有些性能改进不会为用户带来明显的改进。将性能优化集中在存在特定应用程序问题的领域很重要。有些性能优化是常规的好做法，任何情况下都可以遵守。对于其他优化，它们是否有帮助取决于您的应用程序的需要及其预期的用户。例如，如果您不使用任何动画、视频或图形滤镜和效果，您的应用程序的执行性能确实会更好。但是，使用 **Flash Platform** 构建应用程序的原因之一是媒体和图形功能允许表现力丰富的应用程序。考虑您所需的丰富级别是否与运行应用程序的计算机和设备的性能特性匹配良好。

常见的一个建议是“避免过早优化”。某些性能优化需要以难以读取或灵活性差的方式编写代码。此类代码经过优化后更难维护。对于这些优化，在选择优化代码之前，通常最好等待并确定代码的特定部分是否执行性能很差。

改进性能有时需要权衡。理想情况下，降低应用程序的内存使用量也会提高应用程序执行任务的速度。但是，这种理想类型的改进并非总能实现。例如，如果应用程序在操作期间冻结，解决方案通常需要将工作拆分以在多个帧中运行。由于正在拆分工作，很可能将需要更长的时间来完成整个进程。但是，如果应用程序继续响应输入并且不会冻结，用户可能不会注意到额外时间。

了解要优化的内容以及优化是否有用的关键是执行性能测试。第 79 页的“[基准测试和部署](#)”中介绍了用于测试性能的多种技术和技巧。

有关确定应用程序中适合优化的部分的详细信息，请参阅下列资源：

- AIR 应用程序的性能优化（网址为 http://www.adobe.com/go/learn_fp_goldman_tv_cn）（由 Oliver Goldman 撰写的 MAX 会议演示文稿的视频）
- Adobe AIR 应用程序的性能优化（网址为 http://www.adobe.com/go/learn_fp_air_perf_devnet_cn）（由 Oliver Goldman 根据他的演示文稿撰写的 Adobe Developer Connection 文章）

第 2 章：节省内存

节省内存对于应用程序（尤其台式机应用程序）开发一直非常重要。然而，内存的使用量对移动设备来说非常重要，因此有必要限制应用程序占用的内存量。

显示对象



选择适当的显示对象。

ActionScript 3.0 包含很多显示对象。要限制内存用量，最简单的优化技巧之一是使用合适类型的显示对象。对于非交互式简单形状，请使用 **Shape** 对象。对于不需要时间轴的交互式对象，请使用 **Sprite** 对象。对于使用时间轴的动画，请使用 **MovieClip** 对象。应始终为应用程序选择最有效的对象类型。

以下代码显示不同显示对象的内存使用量：

```
trace(getSize(new Shape()));  
// output: 236  
  
trace(getSize(new Sprite()));  
// output: 412  
  
trace(getSize(new MovieClip()));  
// output: 440
```

`getSize()` 方法显示对象在内存中占用的字节数。可以看到，如果不需要 **MovieClip** 对象的功能，使用多个 **MovieClip** 对象（而不是使用简单的 **Shape** 对象）会浪费内存。

原始类型



使用 `getSize()` 方法为代码设置基准确定任务的最有效对象。

所有原始类型（**String** 除外）在内存中都是占用 4 – 8 个字节。使用特定类型的基元无法优化内存：


```
// Primitive types
var a:Number;
trace(getSize(a));
// output: 8

var b:int;
trace(getSize(b));
// output: 4

var c:uint;
trace(getSize(c));
// output: 4

var d:Boolean;
trace(getSize(d));
// output: 4

var e:String;
trace(getSize(e));
// output: 4
```

如果没有为表示 64 位值的数字分配值，ActionScript 虚拟机 (AVM) 将为其分配 8 个字节。所有其他原始类型存储时均占用 4 个字节。

```
// Primitive types
var a:Number = 8;
trace(getSize(a));
// output: 4

a = Number.MAX_VALUE;
trace(getSize(a));
// output: 8
```

此行为不同于 String 类型。分配的存储量基于 String 的长度：

```
var name:String;
trace(getSize(name));
// output: 4

name = "";
trace(getSize(name));
// output: 24
```

```
name = "Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the
industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and
scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into
electronic typesetting, remaining essentially unchanged. It was popularized in the 1960s with the release
of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like
Aldus PageMaker including versions of Lorem Ipsum.";
trace(getSize(name));
// output: 1172
```

使用 `getSize()` 方法为代码设置基准确定任务的最有效对象。

重用对象



尽可能重复使用对象而不是重新创建对象。

优化内存的另一种简单方法是尽可能重复使用对象并避免重新创建对象。例如，在循环中，不要使用以下代码：

```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

var area:Rectangle;

for (var:int = 0; i < MAX_NUM; i++)
{
    // Do not use the following code
    area = new Rectangle(i,0,1,10);
    myBitmapData.fillRect(area,COLOR);
}
```

在各个循环迭代中重新创建 **Rectangle** 对象将使用更多内存且速度更慢，因为将在各个迭代中创建一个新对象。请使用以下方法：

```
const MAX_NUM:int = 18;
const COLOR:uint = 0xCCCCCC;

// Create the rectangle outside the loop
var area:Rectangle = new Rectangle(0,0,1,10);

for (var:int = 0; i < MAX_NUM; i++)
{
    area.x = i;
    myBitmapData.fillRect(area,COLOR);
}
```

上面的示例中使用的对象对内存影响相对较小。下面的示例说明通过重复使用 **BitmapData** 对象可以节省更多的内存。以下用于创建平铺效果的代码浪费了内存：

```
var myImage:BitmapData;
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i < MAX_NUM; i++)
{
    // Create a 20 x 20 pixel bitmap, non-transparent
    myImage = new BitmapData(20,20,false,0xF0D062);

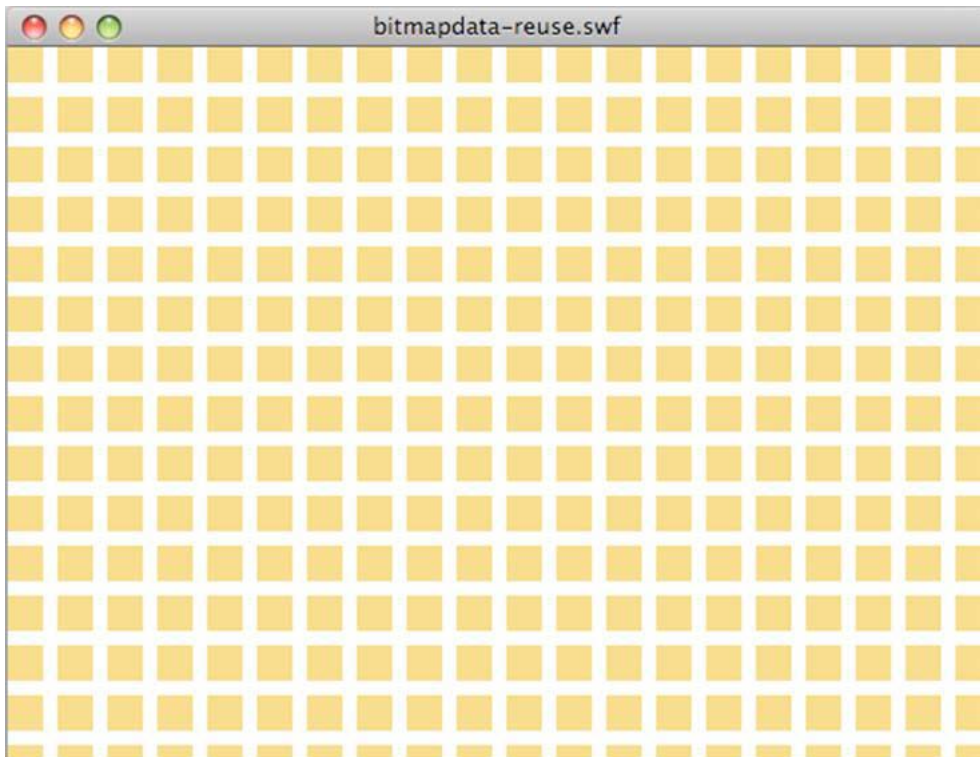
    // Create a container for each BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

注：使用正值时，将舍入值转换为整数比使用 **Math.floor()** 方法快得多。

以下图像显示位图平铺效果：



位图平铺效果

优化的版本创建了由多个 **Bitmap** 实例引用的单一 **BitmapData** 实例，并产生相同的效果：

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i < MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the display list
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);
}
```

此方法节省了大约 700 KB 的内存，这对于传统移动设备相当可观。使用 **Bitmap** 属性，不必更改原始 **BitmapData** 实例即可操纵每个位图容器：

```
// Create a single 20 x 20 pixel bitmap, non-transparent
var myImage:BitmapData = new BitmapData(20,20,false,0xF0D062);
var myContainer:Bitmap;
const MAX_NUM:int = 300;

for (var i:int = 0; i< MAX_NUM; i++)
{
    // Create a container referencing the BitmapData instance
    myContainer = new Bitmap(myImage);

    // Add it to the DisplayList
    addChild(myContainer);

    // Place each container
    myContainer.x = (myContainer.width + 8) * Math.round(i % 20);
    myContainer.y = (myContainer.height + 8) * int(i / 20);

    // Set a specific rotation, alpha, and depth
    myContainer.rotation = Math.random()*360;
    myContainer.alpha = Math.random();
    myContainer.scaleX = myContainer.scaleY = Math.random();
}
```

下图显示的是位图转换的效果:



位图转换效果

更多帮助主题

第 46 页的“[位图缓存](#)”

对象池



请尽可能使用对象池。

另一个重要优化称为对象池，涉及到不断重复使用对象。在初始化应用程序期间创建一定数量的对象并将其存储在一个池中，例如 **Array** 或 **Vector** 对象。对一个对象完成操作后，停用该对象以免它占用 CPU 资源，然后删除所有相互引用。然而，不要将引用设置为 **null**，这将使它符合垃圾回收条件。只需将该对象放回到池中，在需要新对象时可以对其进行检索。

重用对象可减少实例化对象的需求，而实例化对象成本很高。还可以减少垃圾回收器运行的机会，从而提高应用程序运行速度。以下代码演示对象池技术：

```
package
{
    import flash.display.Sprite;

    public final class SpritePool
    {
        private static var MAX_VALUE:uint;
        private static var GROWTH_VALUE:uint;
        private static var counter:uint;
        private static var pool:Vector.<Sprite>;
        private static var currentSprite:Sprite;

        public static function initialize( maxPoolSize:uint, growthValue:uint ):void
        {
            MAX_VALUE = maxPoolSize;
            GROWTH_VALUE = growthValue;
            counter = maxPoolSize;

            var i:uint = maxPoolSize;

            pool = new Vector.<Sprite>(MAX_VALUE);
            while( --i > -1 )
                pool[i] = new Sprite();
        }

        public static function getSprite():Sprite
        {
            if ( counter > 0 )
                return currentSprite = pool[--counter];

            var i:uint = GROWTH_VALUE;
            while( --i > -1 )
                pool.unshift( new Sprite() );
            counter = GROWTH_VALUE;
            return getSprite();
        }

        public static function disposeSprite(disposedSprite:Sprite):void
        {
            pool[counter++] = disposedSprite;
        }
    }
}
```

SpritePool 类在初始化应用程序时创建新的对象池。**getSprite()** 方法返回这些对象的实例，而 **disposeSprite()** 方法释放这些实例。代码允许池容量用尽时可以增长。还可以创建固定大小的池，这样当池容量用尽时将不会分配新对象。尽可能避免在循环中创建新对象。有关详细信息，请参见第 10 页的“[释放内存](#)”。以下代码使用 **SpritePool** 类检索新实例：

```
const MAX_SPRITES:uint = 100;
const GROWTH_VALUE:uint = MAX_SPRITES >> 1;
const MAX_NUM:uint = 10;

SpritePool.initialize ( MAX_SPRITES, GROWTH_VALUE );

var currentSprite:Sprite;
var container:Sprite = SpritePool.getSprite();

addChild ( container );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    for ( var j:int = 0; j< MAX_NUM; j++ )
    {
        currentSprite = SpritePool.getSprite();
        currentSprite.graphics.beginFill ( 0x990000 );
        currentSprite.graphics.drawCircle ( 10, 10, 10 );
        currentSprite.x = j * (currentSprite.width + 5);
        currentSprite.y = i * (currentSprite.width + 5);
        container.addChild ( currentSprite );
    }
}
```

以下代码在当单击鼠标时，将删除显示列表中的所有显示对象，并在以后的其他任务中重复使用这些对象：

```
stage.addEventListener ( MouseEvent.CLICK, removeDots );

function removeDots ( e:MouseEvent ):void
{
    while (container.numChildren > 0 )
        SpritePool.disposeSprite (container.removeChildAt(0) as Sprite );
}
```

注：池矢量始终引用 **Sprite** 对象。如果要从内存中完全删除对象，需要对 **SpritePool** 类使用 **dispose()** 方法，从而删除所有剩余引用。

释放内存



删除对对象的所有引用以确保触发垃圾回收。

在 **Flash Player** 的发行版中无法直接启动垃圾回收器。要确保将一个对象作为垃圾回收，请删除对该对象的所有引用。请记住，在 **ActionScript 1.0** 和 **2.0** 中使用的旧 **delete** 运算符在 **ActionScript 3.0** 中有不同的行为。它只能用于删除动态对象的动态属性。

注：在 **Adobe® AIR®** 和 **Flash Player** 的调试版中可以直接调用垃圾回收器。

例如，以下代码将 **Sprite** 引用设置为 **null**：

```
var mySprite:Sprite = new Sprite();

// Set the reference to null, so that the garbage collector removes
// it from memory
mySprite = null;
```

请记住，当对象设置为 **null** 时，不必将其从内存中删除。如果系统认为可用内存不是足够低，垃圾回收器可能不会运行。垃圾回收的执行时间不可预知。内存分配（而不是对象删除）会触发垃圾回收。当垃圾回收器运行时，它将查找尚未收集的对象的图形。垃圾回收器通过这些图形中查找相互引用但应用程序不再使用的对象，从而检测出处于非活动状态的对象。将删除通过这种方式检测到的处于非活动状态的对象。

在大型应用程序中，此进程会占用大量 CPU 并影响性能，还可导致应用程序运行速度显著降低。通过尽量重复使用对象，尝试限制使用垃圾回收。此外，尽可能地将引用设置为 **null**，以便垃圾回收器用较少处理时间来查找对象。将垃圾回收看作一项保护措施，并始终尽可能明确地管理对象生存期。

注：将对显示对象的引用设置为 **null** 不能确保冻结该对象。该对象在作为垃圾回收之前，仍将占用 CPU 周期。在将对对象的引用设置为 **null** 之前，先确保正确地停用对象。

可使用 Adobe AIR 和 Flash Player 的调试版中提供的 `System.gc()` 方法启动垃圾回收器。将此设置与 Adobe® Flash® Builder™ 捆绑还可以手动启动垃圾回收器。通过运行垃圾回收器，可以了解应用程序的响应方式以及是否已将对象从内存中正确删除。

注：如果将某个对象用作事件侦听器，则其他对象可以引用它。如果是这样，先使用 `removeEventListener()` 方法删除事件侦听器，然后再将引用设置为 **null**。

幸运的是，这样可以立即减少位图使用的内存量。例如，`BitmapData` 类包括一个 `dispose()` 方法。下面的示例将创建一个 1.8 MB 的 `BitmapData` 实例。当前使用的内存增大到 1.8 MB，`System.totalMemory` 属性返回一个更小的值：

```
trace(System.totalMemory / 1024);  
// output: 43100  
  
// Create a BitmapData instance  
var image:BitmapData = new BitmapData(800, 600);  
  
trace(System.totalMemory / 1024);  
// output: 44964
```

然后，手动将 `BitmapData` 从内存中删除并再次检查内存使用情况：

```
trace(System.totalMemory / 1024);  
// output: 43100  
  
// Create a BitmapData instance  
var image:BitmapData = new BitmapData(800, 600);  
  
trace(System.totalMemory / 1024);  
// output: 44964  
  
image.dispose();  
image = null;  
  
trace(System.totalMemory / 1024);  
// output: 43084
```

尽管 `dispose()` 方法可删除内存中的像素，但仍必须将引用设置为 **null** 才可完全释放内存。当不再需要 `BitmapData` 对象时，要始终调用 `dispose()` 方法并将引用设置为 **null**，以便立即释放内存。

注：Flash Player 10.1 和 AIR 1.5.2 在 `System` 类中引入了一个名为 `disposeXML()` 的新方法。借助此方法，通过将 XML 树作为参数传递，可立即使 XML 对象可用于垃圾回收。

更多帮助主题

第 24 页的“[冻结和解冻对象](#)”

使用位图

使用矢量（而不是位图）是节省内存的好方法。然而，使用矢量（特别是大量矢量），会显著增加对 CPU 或 GPU 资源的需求。使用位图是优化呈现的一个好方法，因为运行时在屏幕上绘制像素比呈现矢量内容需要的处理资源要少。

更多帮助主题
第 52 页的“[手动位图缓存](#)”

位图采样

为了更充分利用内存，当 Flash Player 检测到 16 位屏幕时，32 位不透明图像将缩小为 16 位图像。采样只占用一半内存资源，图像的呈现速度会更快。此功能只在适用于 Windows Mobile 的 Flash Player 10.1 中可用。

注：在 Flash Player 10.1 之前的版本中，在内存中创建的所有像素都以 32 位（4 个字节）存储。一个简单的 300 x 300 像素的徽标会占用 350 KB 内存 (300*300*4/1024)。使用此新功能，同样的不透明徽标仅占用 175 KB。如果徽标透明，则不会将采样缩小为 16 位，而在内存中保持原大小。此功能仅适用于嵌入位图或运行时加载的图像（PNG、GIF、JPG）。

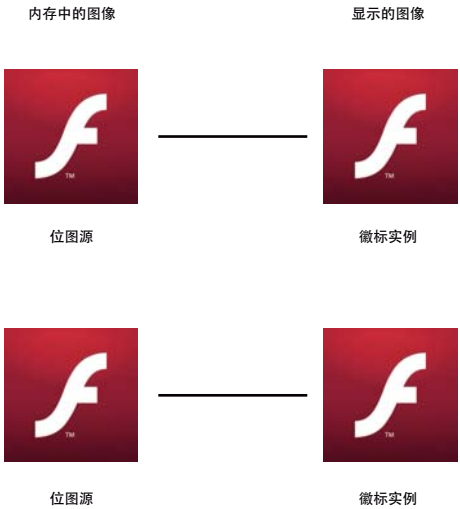
在移动设备上，可能难以区分图像是以 16 位呈现还是以 32 位呈现。对于只包含几种颜色的简单图像，检测不到区别。即使对于更复杂的图像，也很难检测到区别。但是，图像放大时可能褪色，因此从外观上看，16 位的变化趋势比 32 位明显。

BitmapData 单个引用

通过尽量重复使用实例来优化 BitmapData 类的使用，这一点非常重要。Flash Player 10.1 和 AIR 2.5 引入了一个适用于所有平台的新功能，称为 BitmapData 单个引用。从嵌入图像创建 BitmapData 实例时，将对所有 BitmapData 实例使用位图的同一版本。如果稍后修改位图，则在内存中为该位图提供其自身的唯一位图。嵌入的图像可来自库或 [Embed] 标签。

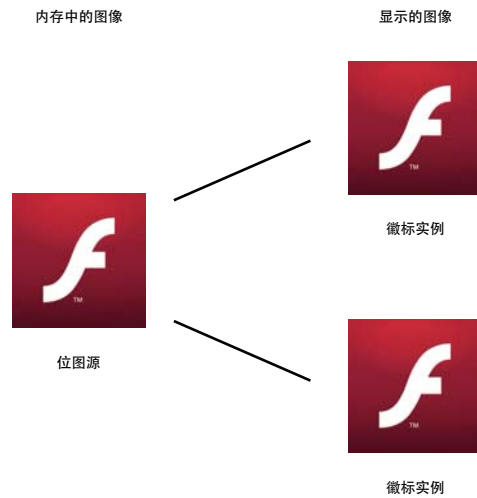
注：此新功能同样适用于当前内容，因为 Flash Player 10.1 和 AIR 2.5 会自动重复使用位图。

当实例化嵌入的图像时，将在内存中创建相关联的位图。在 Flash Player 10.1 和 AIR 2.5 之前的版本中，内存中为每个实例都提供了一个单独的位图，如下图所示：



运行 Flash Player 10.1 和 AIR 2.5 之前，内存中的位图

在 Flash Player 10.1 和 AIR 2.5 中，当创建同一图像的多个实例时，单个版本的位图将用于所有 BitmapData 实例。下图演示了此概念：



Flash Player 10.1 内存中的位图 和 AIR 2.5

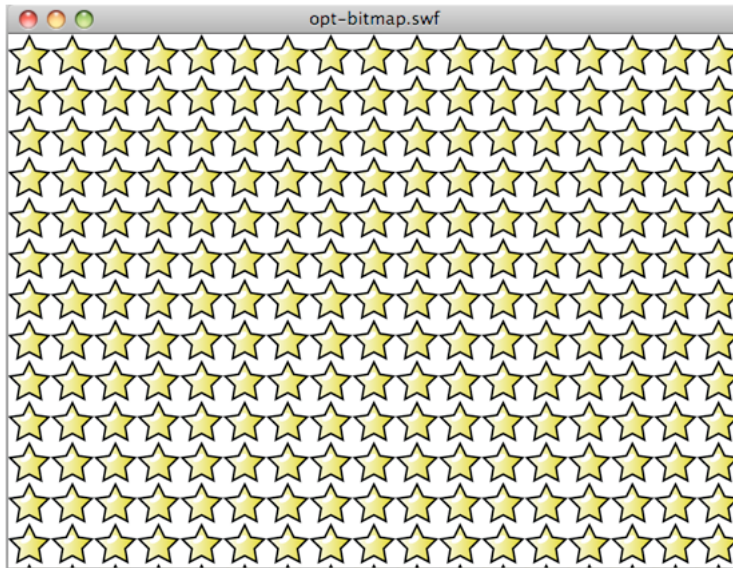
此方法大大降低了具有许多位图的应用程序的内存使用量。以下代码创建了 **Star** 符号的多个实例：

```
const MAX_NUM:int = 18;

var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}
```

下图显示该代码的效果：



用于创建多个符号实例的代码的效果

例如，使用 Flash Player 10 时，上面的动画使用大约 1008 KB 内存。使用 Flash Player 10.1 时，在台式机和移动设备上该动画只使用 4 KB。

以下代码修改一个 BitmapData 实例：

```
const MAX_NUM:int = 18;

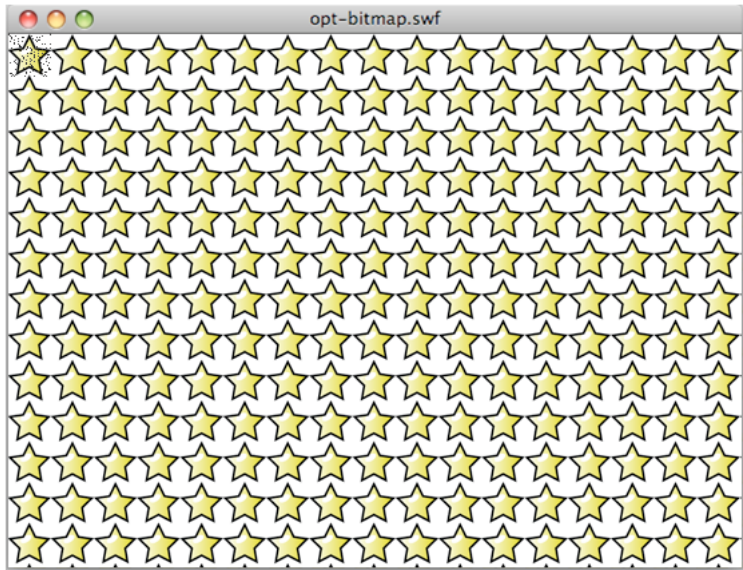
var star:BitmapData;
var bitmap:Bitmap;

for (var i:int = 0; i<MAX_NUM; i++)
{
    for (var j:int = 0; j<MAX_NUM; j++)
    {
        star = new Star(0,0);
        bitmap = new Bitmap(star);
        bitmap.x = j * star.width;
        bitmap.y = i * star.height;
        addChild(bitmap)
    }
}

var ref:Bitmap = getChildAt(0) as Bitmap;

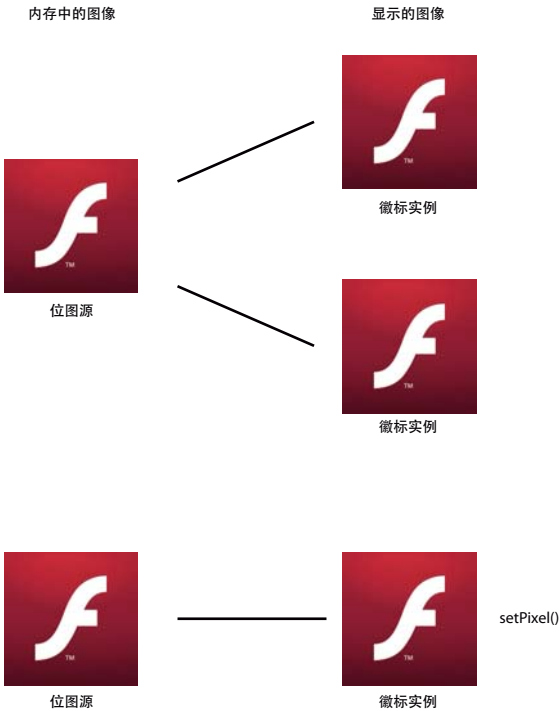
ref.bitmapData.pixelDissolve(ref.bitmapData, ref.bitmapData.rect, new
Point(0,0),Math.random()*200,Math.random()*200, 0x990000);
```

下图显示修改一个 Star 实例的效果：



修改一个实例的效果

在内部，运行时会在内存中自动分配和创建位图以处理像素修改。当调用的 `BitmapData` 类的方法导致像素修改时，将在内存中创建一个新实例，而不更新其他实例。下图演示了此概念：



修改一个位图后，内存中的效果

如果修改一个星星，则在内存中创建一个新的副本。在 `Flash Player 10.1` 和 `AIR 2.5` 上，生成的动画使用大约 `8 KB` 内存。
在上一示例中，各个位图可单独用于转换。要仅创建平铺效果，则 `beginBitmapFill()` 方法是最合适的方法：

```
var container:Sprite = new Sprite();

var source:BitmapData = new Star(0,0);

// Fill the surface with the source BitmapData
container.graphics.beginBitmapFill(source);
container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);

addChild(container);
```

此方法与仅创建一个 **BitmapData** 实例的效果相同。要连续旋转星形，请使用绕各个帧旋转的 **Matrix** 对象，而不是访问各个 **Star** 实例。将此 **Matrix** 对象传递给 **beginBitmapFill()** 方法：

```
var container:Sprite = new Sprite();

container.addEventListener(Event.ENTER_FRAME, rotate);

var source:BitmapData = new Star(0,0);
var matrix:Matrix = new Matrix();

addChild(container);

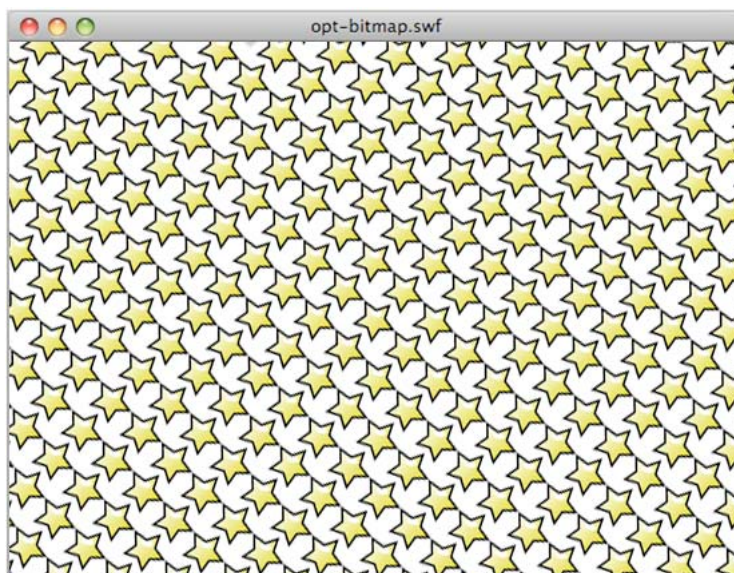
var angle:Number = .01;

function rotate(e:Event):void
{
    // Rotate the stars
    matrix.rotate(angle);

    // Clear the content
    container.graphics.clear();

    // Fill the surface with the source BitmapData
    container.graphics.beginBitmapFill(source,matrix,true,true);
    container.graphics.drawRect(0,0,stage.stageWidth,stage.stageHeight);
}
```

借助此技术，无需 **ActionScript** 循环即可创建这种效果。运行时在内部执行所有操作。下图显示转换星形的效果：



旋转星形的效果

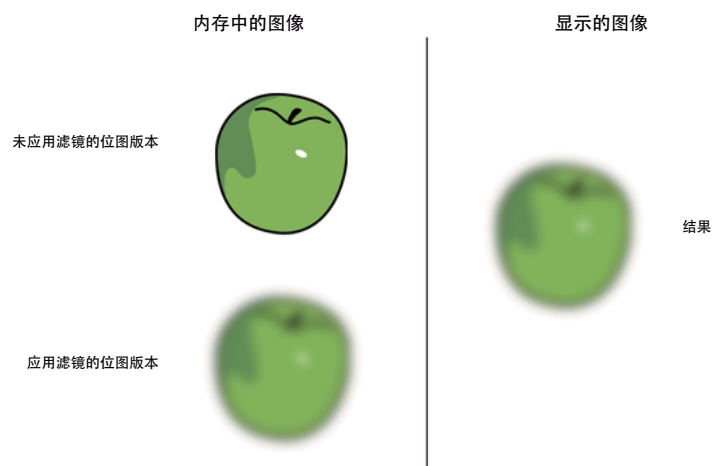
借助此方法，更新原始的源 **BitmapData** 对象时将自动更新其在舞台上的所有使用，这可能是一项功能强大的技术。然而，此方法不允许单独缩放各个星形，如前面的示例中所示。

注：当使用同一图像的多个实例时，将根据某个类是否与内存中的原始位图相关联进行绘图。如果不存在与位图相关联的类，则图像将作为 **Shape** 对象使用位图填充进行绘制。

滤镜和动态位图卸载

💡 避免使用滤镜，包括经过 **Pixel Bender** 处理的滤镜。

尝试尽量减少使用滤镜效果，包括通过 **Pixel Bender** 在移动设备中处理的滤镜。将滤镜应用于显示对象时，运行时将在内存中创建两个位图。其中每个位图的大小与显示对象相同。将第一个位图创建为显示对象的栅格化版本，然后用于生成应用滤镜的另一个位图：



应用滤镜时内存中的两个位图

当修改滤镜的某个属性时，内存中的两个位图都将更新以创建生成的位图。此过程涉及一些 **CPU** 处理，这两个位图可能会占用大量内存。

Flash Player 10.1 和 **AIR 2.5** 在所有平台上引入一种新的滤镜行为。如果滤镜在 **30** 秒内没有进行修改，或者将其隐藏或置于屏幕之外，将释放未过滤的位图占用的内存。

此功能可节省所有平台上的滤镜占用的一半内存。以应用模糊滤镜后的文本对象为例。在这种情况下，文本用于简单装饰，不会进行修改。**30** 秒后，将释放内存中未过滤的位图。如果文本隐藏 **30** 秒或置于屏幕之外，将会产生同样的效果。修改滤镜的某个属性后，将重新创建内存中未过滤的位图。此功能称为动态位图卸载。即使进行了这些优化，在使用滤镜时仍要谨慎小心；在对它们进行修改时，仍要求大量 **CPU** 或 **GPU** 处理。

最佳做法是，尽可能使用通过创作工具（例如 **Adobe® Photoshop®**）创建的位图来模拟滤镜。避免在 **ActionScript** 中使用运行时创建的动态位图。使用外部创作的位图可帮助运行时减少 **CPU** 或 **GPU** 负载，特别是当滤镜属性不随时间更改时。如果可能，在创作工具中创建位图所需的任何效果。然后，可以在运行时中显示该位图，而无需对它进行任何处理，这样速度要快得多。

直接进行 mip 映射



如果需要，使用 mip 映射功能缩放较大图像。

在所有平台上，Flash Player 10.1 和 AIR 2.5 中还提供了另一个新功能，该功能与 mipmap 处理有关。Flash Player 9 和 AIR 1.0 引入了 mipmap 处理功能，可改进缩小位图的品质和性能。

注：Mip 映射功能仅适用于动态加载的图像或嵌入位图。Mip 映射不适用于经过过滤或缓存的显示对象。仅当位图的宽度和高度为偶数时才可以进行 mip 映射。当位图的宽度或高度为奇数时，mip 映射将停止。例如，可将 250 x 250 图像通过 mip 映射缩小到 125 x 125，但无法对其进行进一步的 mip 映射。在这种情况下，至少其中一个尺寸是奇数。尺寸为 2 的若干次幂的位图的效果最佳，例如，256 x 256、512 x 512、1024 x 1024 等。

例如，假设加载了一个 1024 x 1024 图像，但开发人员想对该图像进行缩放以在库中创建一个缩略图。Mip 映射功能在使用中间采样版本的位图作为纹理缩放图像时可以正确呈现该图像。运行时的早期版本在内存中创建中间缩小版本的位图。如果加载了一个 1024 x 1024 图像并以 64 x 64 显示，则运行时早期版本创建的每个位图大小只有原来的一半。例如，在这种情况下将会创建 512 x 512、256 x 256、128 x 128 和 64 x 64 位图。

Flash Player 10.1 和 AIR 2.5 当前支持直接从原始源 mipmap 处理到所需的目标大小。在上一示例中，将仅创建 4 MB (1024 x 1024) 的原始位图和 16 KB (64 x 64) 的经过 mip 映射处理的位图。

Mip 映射逻辑同样适用于动态位图卸载功能。如果仅使用 64 x 64 的位图，则从内存中释放 4MB 的原始位图。如果必须重新创建 mip 映射，则将重新加载原始位图。另外，如果需要其他各种大小经过 mip 映射处理的位图，则使用位图的 mip 映射链来创建位图。例如，如果必须创建 1:8 位图，则会检查 1:4、1:2 和 1:1 位图以确定首先将哪个位图加载到内存中。如果找不到其他版本，则将从资源中加载 1:1 原始位图并使用该位图。

JPEG 解压缩程序可以使用自己的格式执行 mip 映射。通过直接进行 mip 映射，可将大型位图直接压缩为 mip 映射格式，而无需加载整个解压缩后的图像。生成 mip 映射的速度明显加快，并且不会为大型位图分配占用的内存然后再将其释放。JPEG 图像品质相当于常规 mip 映射技术。

注：尽量少用 mip 映射。尽管它可以改进缩小位图的品质，但它对带宽、内存和速度都有影响。在某些情况下，最好选择使用通过外部工具预缩放的位图版本，并将其导入到您的应用程序中。如果只需缩小位图，不要一开始就使用较大位图。

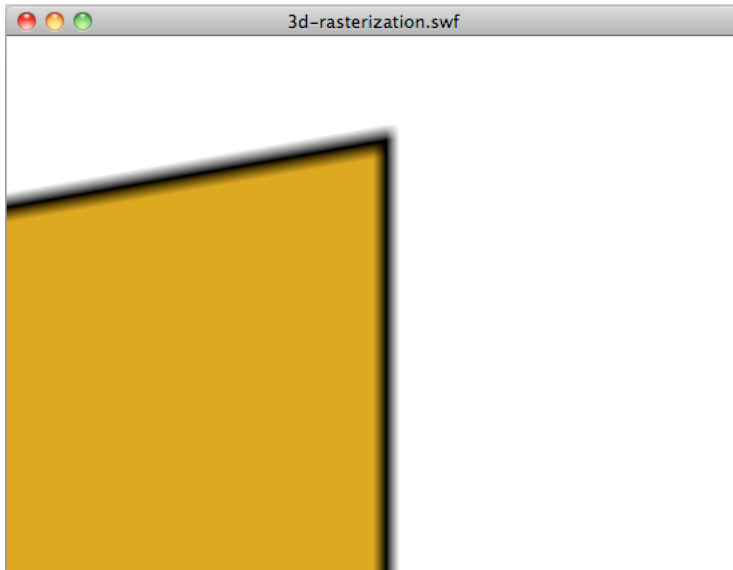
使用 3D 效果



考虑手动创建 3D 效果。

Flash Player 10 和 AIR 1.5 引入了一个 3D 引擎，允许您对显示对象应用透视转换。您可以使用 rotationX 和 rotationY 属性或 Graphics 类的 drawTriangles() 方法应用这些转换。您还可以使用 z 属性应用深度。请记住，每个经过透视转换的显示对象都将被栅格化为位图，因此需要更多内存空间。

下图演示了使用透视转换时由栅格化处理生成的消除锯齿：



由透视转换导致的消除锯齿

消除锯齿是将矢量内容动态栅格化为位图的结果。当您在台式机版本的 AIR 和 Flash Player 中以及在用于移动设备的 AIR 2.0.1 和 AIR 2.5 中使用 3D 效果时，会消除锯齿。但是，在移动设备的 Flash Player 上不会应用锯齿消除。

如果您可以手动创建 3D 效果，而无需依赖本机 API，则可以减少内存使用量。不过，Flash Player 10 和 AIR 1.5 中引入的新 3D 功能更便于进行纹理映射，因为 `drawTriangles()` 等方法可以本机处理纹理映射。

作为开发人员，应确定要创建的 3D 效果在通过本机 API 或手动处理后是否会提供更好的性能。请考虑 ActionScript 执行和呈现性能以及内存使用量。

在 AIR 2.0.1 和 AIR 2.5 移动应用程序中，您将 `renderMode` 应用程序属性设置为 GPU，则 GPU 会执行 3D 转换。但是，如果 `renderMode` 为 CPU，则由 CPU 而不是 GPU 执行 3D 转换。在 Flash Player 10.1 应用程序中，CPU 执行 3D 转换。

当 CPU 执行 3D 转换时，如果要对显示对象应用任何 3D 转换，则需要内存中有两个位图。一个位图用于源位图，另一个用于透视转换的版本。在这种情况下，3D 转换的工作原理与滤镜类似。因此，当 CPU 执行 3D 转换时尽量少用 3D 属性。

文本对象和内存



将 Adobe® Flash® 文本引擎用于只读文本；将 `TextField` 对象用于输入文本。

Flash Player 10 和 AIR 1.5 引入了一种强大的新文本引擎，即 Adobe Flash 文本引擎 (FTE)，可节省系统内存。但是，FTE 是一个低级 API，要求在其上使用 `flash.text.engine` 包中提供的其他 ActionScript 3.0 图层。

对于只读文本，最好使用 Flash 文本引擎，它占用较少的内存并提供更好的呈现效果。对于输入文本，最好使用 `TextField` 对象，因为在创建典型行为（例如，输入处理和自动换行）时需要的 ActionScript 代码较少。

更多帮助主题

第 58 页的“[呈现文本对象](#)”

事件模型与回调



考虑使用简单的回调代替事件模型。

ActionScript 3.0 事件模型基于对象调度的概念。事件模型是面向对象的，可以进行优化以重复使用代码。`dispatchEvent()` 方法循环访问侦听器列表，并对各个注册的对象调用事件处理函数方法。然而，事件模型的一个缺点是可能要在应用程序的生存期内创建许多对象。

假设您必须从时间轴调度一个事件来指示动画序列的末尾。要实现此通知，您可以从时间轴中的特定帧调度一个事件，如以下代码所示：

```
dispatchEvent( new Event ( Event.COMPLETE ) );
```

Document 类可以使用以下代码行侦听此事件：

```
addEventListener( Event.COMPLETE, onAnimationComplete );
```

虽然此方法是正确的，但使用本机事件模型与使用传统的回调函数相比，速度更慢且占用的内存更多。必须创建 **Event** 对象并为其分配内存，而这会降低性能。例如，当侦听 **Event.ENTER_FRAME** 事件时，将在各个帧上为事件处理函数创建一个新事件对象。在捕获和冒泡阶段（如果显示列表很复杂，此成本会很高），显示对象的性能可能会特别低。

第 3 章：最大程度减小 CPU 使用量

优化的另一个重要部分是 CPU 使用量。优化 CPU 处理可提高性能，从而延长移动设备上的电池寿命。

针对 CPU 使用量的 Flash Player 10.1 增强功能

Flash Player 10.1 引入了两个有助于减少 CPU 处理的新功能。这些功能包括：SWF 内容在屏幕外时暂停和恢复播放，限制页面上的 Flash Player 实例数。

暂停、节流和恢复

注：暂停、节流和恢复功能不适用于 Adobe® AIR® 应用程序。

为了优化 CPU 和电池使用量，Flash Player 10.1 引入了一个与非活动实例相关的新功能。此功能是当内容位于屏幕之外和在屏幕上时暂停和恢复 SWF 文件，从而允许您限制 CPU 使用量。借助此功能，Flash Player 通过删除在恢复播放内容时可能重新创建的任何对象，来释放尽可能多的内存。当全部内容都位于屏幕之外时，才将该内容视为位于屏幕之外。

以下两种情况会导致 SWF 内容位于屏幕之外：

- 用户滚动页面，导致 SWF 内容移动到屏幕之外。

在这种情况下，如果播放任何音频或视频，内容将继续播放，但呈现会停止。如果没有播放任何音频或视频，要确保播放或 ActionScript 执行不会暂停，请将 hasPriority HTML 参数设置为 true。但是，请记住，当内容在屏幕以外或隐藏时，无论 hasPriority HTML 参数为何值，SWF 内容呈现均将暂停。

- 在浏览器中打开了一个选项卡，导致 SWF 内容移动到背景中。

在这种情况下，无论 hasPriority HTML 标签为何值，SWF 内容将降速或节流到 2 和 8 fps 之间。除非 SWF 内容再次可见，否则音频和视频播放将停止并且不处理任何内容呈现。

对于运行在 Windows 和 Mac 台式机浏览器上的 Flash Player 11.2 和更高版本，可以在应用程序中使用 ThrottleEvent。Flash Player 暂停、节流或恢复播放时调度 ThrottleEvent。

此 ThrottleEvent 事件为广播事件，这意味着将由所有具有注册了此事件的侦听器的 EventDispatcher 对象调度此事件。有关广播事件的详细信息，请参阅 DisplayObject 类。

实例管理

注：实例管理功能不适用于 Adobe® AIR® 应用程序。



使用 hasPriority HTML 参数可延迟加载位于屏幕之外的 SWF 文件。

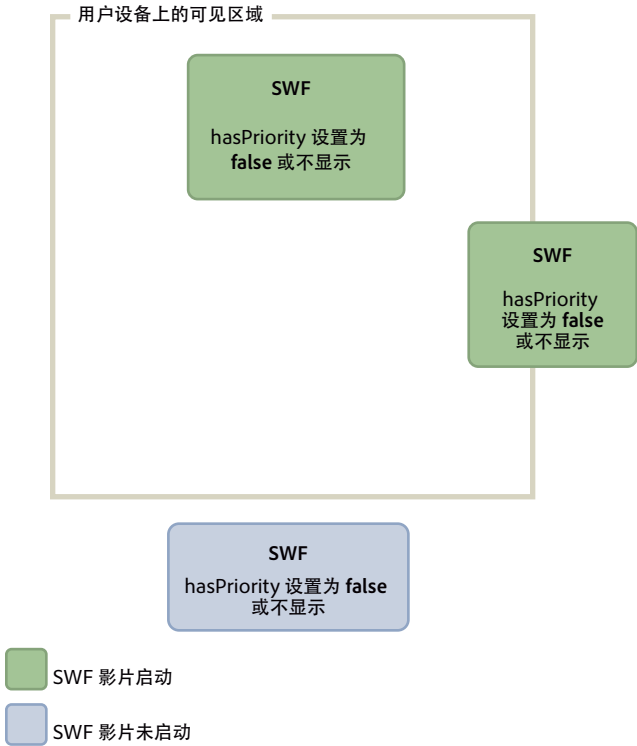
Flash Player 10.1 引入了一个名为 hasPriority 的新 HTML 参数：

```
<param name="hasPriority" value="true" />
```

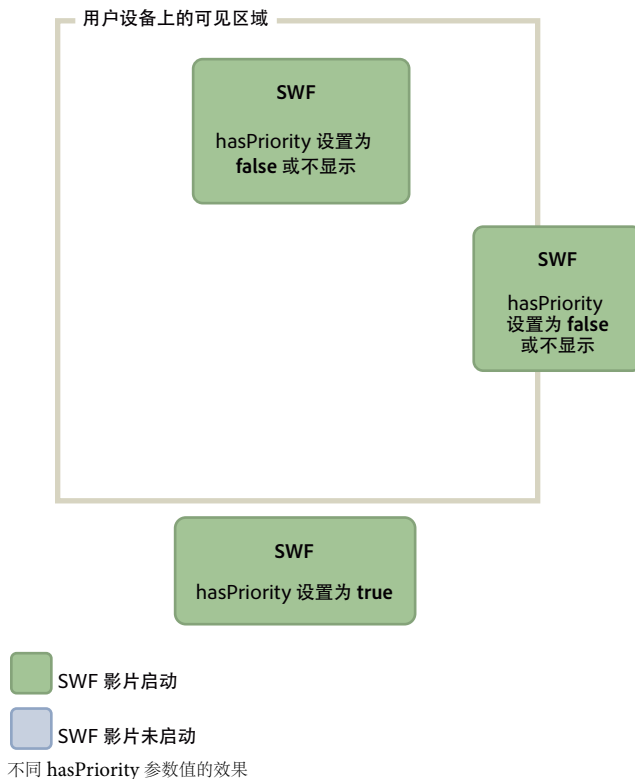
此功能限制在页面上启动的 Flash Player 实例的数量。限制实例的数量有助于节省 CPU 和电池资源。其目的是将特定优先级分配给 SWF 内容，使某些内容的优先级高于页面上的其他内容。请看一个简单的示例：一个用户正在浏览网站，而索引页面承载着三个不同的 SWF 文件。其中一个文件可见，一个在屏幕上部分可见，最后一个位于屏幕之外，需要滚动页面才能看到。前两个动画可以正常启动，但最后一个动画被延迟到其变为可见时启动。当 hasPriority 参数不存在或被设置为 false 时，这种情况是默认行为。为了确保能够启动 SWF 文件（甚至该文件位于屏幕之外），请将 hasPriority 参数设置为 true。然而，不管 hasPriority 参数的值是多少，始终暂停呈现用户看不到的 SWF 文件。

注：如果可用的 CPU 资源量降低，将不再自动启动 Flash Player 实例，即使将 `hasPriority` 参数设置为 `true` 也是如此。如果新实例是在加载此页后通过 JavaScript 创建的，这些实例将忽略 `hasPriority` 标志。如果网站管理员无法添加 `hasPriority` 标志，则会启动任何 1x1 像素或 0x0 像素内容，阻止辅助 SWF 文件延迟。然而，仍可通过单击启动 SWF 文件。此行为称为“单击播放”。

下图显示将 `hasPriority` 参数设置为不同值的效果：



不同 `hasPriority` 参数值的效果



睡眠模式

Flash Player 10.1 和 AIR 2.5 对移动设备引入了一个新功能，有助于节省 CPU 处理，从而提高电池寿命。此功能涉及到许多移动设备上存在的背景光。例如，如果运行移动应用程序的用户被中断并停止使用设备，运行时可检测背景光进入睡眠模式的时间。然后将帧速率降低到 4 帧 / 秒 (fps) 并暂停呈现。对于 AIR 应用程序，还会在应用程序进入背景状态时开始睡眠模式。

ActionScript 代码在睡眠模式下继续执行，这与将 Stage.frameRate 属性设置为 4 fps 类似。但是跳过呈现步骤，因此用户看不到该播放器正在以 4 fps 的速率运行。之所以将帧速率选择为 4 fps（而不是 0），是因为该速率允许所有连接保持打开状态（NetStream、Socket 和 NetConnection）。将帧速率切换到 0 fps 会断开打开的连接。之所以将刷新频率选择为 250 毫秒 (4 fps)，是因为许多设备制造商使用此帧频率作为其刷新频率。使用此值可以使运行时的帧频率与设备本身保持在同一范围。

注：当运行时处于睡眠模式时，Stage.frameRate 属性将返回原始 SWF 文件的帧速率，而不是 4 fps。

当背景光重新进入打开模式时，呈现将恢复。帧速率将恢复其原始值。以用户正在播放音乐的媒体播放器应用程序为例。如果屏幕进入睡眠模式，运送时将根据正在播放的内容类型做出响应。以下是对应的运行时行为的情况列表：

- 背景光进入睡眠模式并且正在播放非 A/V 内容：暂停呈现并将帧速率设置为 4 fps。
- 背景光进入睡眠模式并且正在播放 A/V 内容：运行时强制始终打开背景光，继续供用户使用。
- 背景光从睡眠模式进入打开模式：运行时将帧速率恢复为原始 SWF 文件帧速率设置并恢复呈现。
- Flash Player 在播放 A/V 内容时暂停：Flash Player 将背景光状态重置为默认的系统行为，因为不再播放 A/V。
- 移动设备在播放 A/V 内容时接收到电话呼叫：暂停呈现并将帧速率设置为 4 fps。
- 对移动设备禁用背景光睡眠模式：运行时正常运行。

当背景光进入睡眠模式时，呈现暂停且帧速率减慢。此功能可节省 CPU 处理，但就像在游戏应用程序中一样，不能依赖该功能进行实际暂停。

注：运行时进入或退出睡眠模式时，不会调度 **ActionScript** 事件。

冻结和解冻对象



使用 **REMOVED_FROM_STAGE** 和 **ADDED_TO_STAGE** 事件正确冻结和解冻对象。

要优化您的代码，请始终冻结和解冻您的对象。冻结和解冻对所有对象都很重要，但对显示对象尤其重要。即使显示对象不再位于显示列表中并正在等待作为垃圾回收，其代码仍然占用大量 CPU。例如，它们仍然在使用 **Event.ENTER_FRAME**。因此，使用 **Event.REMOVED_FROM_STAGE** 和 **Event.ADDED_TO_STAGE** 事件正确冻结和解冻对象非常关键。以下示例显示一个在舞台上播放的、与键盘交互的影片剪辑：

```
// Listen to keyboard events
stage.addEventListener(MouseEvent.CLICK, keyIsDown);
stage.addEventListener(MouseEvent.CLICK, keyIsUp);

// Create object to store key states
var keys:Dictionary = new Dictionary(true);

function keyIsDown(e:MouseEvent):void
{
    // Remember that the key was pressed
    keys[e.keyCode] = true;

    if (e.keyCode==Keyboard.LEFT || e.keyCode==Keyboard.RIGHT)
    {
        runningBoy.play();
    }
}

function keyIsUp(e:MouseEvent):void
{
    // Remember that the key was released
    keys[e.keyCode] = false;

    for each (var value:Boolean in keys)
        if ( value ) return;
    runningBoy.stop();
}

runningBoy.addEventListener(Event.ENTER_FRAME, handleMovement);
runningBoy.stop();

var currentState:Number = runningBoy.scaleX;
var speed:Number = 15;

function handleMovement(e:Event):void
{
    if (keys[Keyboard.RIGHT])
    {
        e.currentTarget.x += speed;
        e.currentTarget.scaleX = currentState;
    } else if (keys[Keyboard.LEFT])
    {
        e.currentTarget.x -= speed;
        e.currentTarget.scaleX = -currentState;
    }
}
```



与键盘交互的影片剪辑

单击“Remove”按钮后，将从显示列表中删除该影片剪辑：

```
// Show or remove running boy
showBtn.addEventListener(MouseEvent.CLICK,showIt);
removeBtn.addEventListener(MouseEvent.CLICK,removeIt);

function showIt(e:MouseEvent):void
{
    addChild(runningBoy);
}

function removeIt(e:MouseEvent):void
{
    if (contains(runningBoy)) removeChild(runningBoy);
}
```

即使将该影片剪辑从显示列表中删除，它仍会调度 `Event.ENTER_FRAME` 事件。影片剪辑仍在运行，但不会呈现。要正确处理这种情况，请侦听正确的事件并删除事件侦听器，以阻止执行占用大量 CPU 资源的代码：

```
// Listen to Event.ADDED_TO_STAGE and Event.REMOVED_FROM_STAGE
runningBoy.addEventListener(Event.ADDED_TO_STAGE,activate);
runningBoy.addEventListener(Event.REMOVED_FROM_STAGE,deactivate);

function activate(e:Event):void
{
    // Restart everything
    e.currentTarget.addEventListener(Event.ENTER_FRAME,handleMovement);
}

function deactivate(e:Event):void
{
    // Freeze the running boy - consumes fewer CPU resources when not shown
    e.currentTarget.removeEventListener(Event.ENTER_FRAME,handleMovement);
    e.currentTarget.stop();
}
```

按下“Show”按钮后，将重新启动影片剪辑，它将再次侦听 `Event.ENTER_FRAME` 事件且键盘将正确控制该影片剪辑。

注：如果将显示对象从显示列表中删除，则完成删除后将其引用设置为 **null** 不能确保该对象是冻结的。如果不运行垃圾回收器，则对象将继续占用内存和 CPU 处理，即使此对象不再显示。要确保对象占用的 CPU 处理最少，请确保在将其从显示列表中删除后完全将其冻结。

使用 **Flash Player 10** 和 **AIR 1.5** 启动时，还会发生以下行为。如果播放头遇到一个空帧，将自动冻结显示对象，即使没有实施任何冻结行为也是如此。

冻结的概念在使用 **Loader** 类加载远程内容时也很重要。在 **Flash Player 9** 和 **AIR 1.0** 中使用 **Loader** 类时，必须通过侦听由 **LoaderInfo** 对象调度的 **Event.UNLOAD** 事件来手动冻结内容。必须手动冻结每个对象，这是一项至关重要的任务。**Flash Player 10** 和 **AIR 1.5** 对 **Loader** 类引入了一个重要的新方法，称为 **unloadAndStop()**。通过此方法，可以卸载 SWF 文件、自动冻结加载的 SWF 文件中的每个对象并强制运行垃圾回收器。

以下代码将加载 SWF 文件，然后使用 **unload()** 方法将其卸载，这需要更多的处理操作和手动冻结：

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with no automatic object deactivation
    // All deactivation must be processed manually
    loader.unload();
}
```

最好使用 **unloadAndStop()** 方法，该方法以本机方式处理冻结并强制运行垃圾回收过程：

```
var loader:Loader = new Loader();

loader.load ( new URLRequest ( "content.swf" ) );

addChild ( loader );

stage.addEventListener ( MouseEvent.CLICK, unloadSWF );

function unloadSWF ( e:MouseEvent ):void
{
    // Unload the SWF file with automatic object deactivation
    // All deactivation is handled automatically
    loader.unloadAndStop();
}
```

调用 **unloadAndStop()** 方法时将执行下列操作：

- 停止声音。
- 将删除注册到 SWF 文件的主时间轴中的侦听器。
- 停止 **Timer** 对象。
- 释放硬件外围设备（如摄像头和麦克风）。
- 停止每个影片剪辑。
- 停止调度 **Event.ENTER_FRAME**、**Event.FRAME_CONSTRUCTED**、**Event.EXIT_FRAME**、**Event.ACTIVATE** 和 **Event.DEACTIVATE**。

激活和停用事件



使用 Event.ACTIVATE 和 Event.DEACTIVATE 事件检测后台是否处于非活动状态，并相应地优化应用程序。

有两个事件（Event.ACTIVATE 和 Event.DEACTIVATE）可以帮助您微调应用程序以使其尽量使用最少的 CPU 周期。这些事件允许您检测运行时获得或失去焦点的时间。因此可以对代码进行优化，以便对上下文更改做出反应。下列代码侦听这两种事件，并在应用程序失去焦点时动态地将帧频率更改为零。例如，动画可能在用户切换到另一个制表符或将应用程序放入后台时失去焦点：

```
var originalFrameRate:uint = stage.frameRate;
var standbyFrameRate:uint = 0;

stage.addEventListener ( Event.ACTIVATE, onActivate );
stage.addEventListener ( Event.DEACTIVATE, onDeactivate );

function onActivate ( e:Event ):void
{
    // restore original frame rate
    stage.frameRate = originalFrameRate;
}

function onDeactivate ( e:Event ):void
{
    // set frame rate to 0
    stage.frameRate = standbyFrameRate;
}
```

当应用程序再次获得焦点时，帧频率会重置为原始值。除了动态更改帧速率，您还可以考虑进行其他优化，如冻结对象或解冻对象。

activate 和 deactivate 事件能让您实现与有时在移动设备或上网本上提供的“暂停和恢复”功能类似的机制。

更多帮助主题

第 45 页的“[应用程序帧速率](#)”

第 24 页的“[冻结和解冻对象](#)”

鼠标交互



尽可能考虑禁用鼠标交互。

使用交互式对象（例如 MovieClip 或 Sprite 对象）时，运行时执行本机代码以检测和处理鼠标交互。当屏幕上显示许多交互式对象时，特别是当它们重叠时，检测鼠标交互可能会占用大量 CPU 资源。避免此处理的一种简便方法是对不需要任何鼠标交互的对象禁用鼠标交互。以下代码说明了 mouseEnabled 和 mouseChildren 属性的用法：

```
// Disable any mouse interaction with this InteractiveObject
myInteractiveObject.mouseEnabled = false;
const MAX_NUM:int = 10;

// Create a container for the InteractiveObjects
var container:Sprite = new Sprite();

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    // Add InteractiveObject to the container
    container.addChild( new Sprite() );
}

// Disable any mouse interaction on all the children
container.mouseChildren = false;
```

尽可能考虑禁用鼠标交互，这有助于您的应用程序使用较少的 CPU 处理，从而在移动设备上减少电池使用量。

计时器与 ENTER_FRAME 事件



根据内容是否为动画，选择计时器或 ENTER_FRAME 事件。

对于执行时间太长的非动画内容，优先选择计时器，而不是 Event.ENTER_FRAME 事件。

在 ActionScript 3.0 中，有两种方法可以特定的间隔调用函数。第一种方法是使用由显示对象 (DisplayObject) 调度的 Event.ENTER_FRAME 事件。第二种方法是使用计时器。ActionScript 开发人员通常使用 ENTER_FRAME 事件方法。需要对每个帧调度 ENTER_FRAME 事件。因此，调用函数的间隔与当前帧速率有关。可通过 Stage.frameRate 属性来查看帧速率。然而，在某些情况下，使用计时器比使用 ENTER_FRAME 事件更合适。例如，如果您没有使用动画，但又想以特定的间隔调用代码，则使用计时器可能是更好的选择。

计时器的行为与 ENTER_FRAME 事件的行为类似，但是调度事件无需考虑帧速率。通过此行为，可实现一些重要优化。以视频播放器应用程序为例。在这种情况下，由于仅移动应用程序控件，不需要使用高帧速率。

注：帧速率不影响视频，因为视频未嵌入时间轴中。视频是通过渐进式下载或流式下载动态加载的。

在此示例中，帧速率设置为一个较低的值 10 fps。计时器以每秒一次的速度更新控件。TimerEvent 对象中提供的 updateAfterEvent() 方法可以提供更高的更新速率。如果需要，此方法会在每次计时器调度事件时强制更新屏幕。以下代码演示了这一概念：

```
// Use a low frame rate for the application
stage.frameRate = 10;

// Choose one update per second
var updateInterval:int = 1000;
var myTimer:Timer = new Timer(updateInterval,0);

myTimer.start();
myTimer.addEventListener( TimerEvent.TIMER, updateControls );

function updateControls( e:TimerEvent ):void
{
    // Update controls here
    // Force the controls to be updated on screen
    e.updateAfterEvent();
}
```


调用 `updateAfterEvent()` 方法不会修改帧速率。它只强制运行时更新屏幕上已更改的内容。时间轴仍以 10 fps 的速度运行。请记住，在低性能设备上，或者事件处理函数包含要求进行大量处理的代码时，计时器和 `ENTER_FRAME` 事件并不完全精确。就像 SWF 文件帧速率一样，计时器的更新帧速率随情况的不同而不同。



将应用程序中 `Timer` 对象和注册的 `enterFrame` 处理函数的数量降至最少。

对于每个帧，运行时将为其显示列表中的每个显示对象调度一个 `enterFrame` 事件。尽管您可以使用多个显示对象为 `enterFrame` 事件注册侦听器，但这样做意味着将在每个帧上执行更多代码。或者，考虑使用一个集中的 `enterFrame` 处理函数，该函数执行要运行每个帧需要的所有代码。通过集中此类代码，更容易管理所有频繁运行的代码。

同样，如果使用的是 `Timer` 对象，将产生与从多个 `Timer` 对象创建和调度事件相关联的开销。如果您必须在不同的时间间隔触发不同的操作，以下提供了一些建议的替代方法：

- 根据其发生的频率，使用最少数量的 `Timer` 对象和组操作。

例如，将一个 `Timer` 对象用于频繁执行的操作，设置为每 100 毫秒触发一次。将另一个 `Timer` 对象用于频率较低的操作或后台操作，设置为每 2000 毫秒触发一次。

- 使用一个 `Timer` 对象，并以 `Timer` 对象的 `delay` 属性时间间隔的倍数触发操作。

例如，假设您希望某些操作每 100 毫秒发生一次，而其他操作每 200 毫秒发生一次。在这种情况下，请使用一个 `delay` 值为 100 毫秒的 `Timer` 对象。在 `timer` 事件处理函数中，添加一个条件语句，即仅每隔一次运行一次时间间隔为 200 毫秒的操作。以下示例对此技术进行了演示：

```
var timer:Timer = new Timer(100);
timer.addEventListener(TimerEvent.Timer, timerHandler);
timer.start();

var offCycle:Boolean = true;

function timerHandler(event:TimerEvent):void
{
    // Do things that happen every 100 ms

    if (!offCycle)
    {
        // Do things that happen every 200 ms
    }

    offCycle = !offCycle;
}
```



停止未使用的 `Timer` 对象。

如果 `Timer` 对象的 `timer` 事件处理函数仅在特定的条件下执行操作，则当不符合这些条件时调用 `Timer` 对象的 `stop()` 方法。



在 `enterFrame` 事件或 `Timer` 处理函数中，尽量减少对可导致重绘屏幕的显示对象外观的更改。

对于每个帧，呈现阶段都将重绘在该帧期间更改的舞台部分。如果重绘区域很大，或者很小但包含大量或复杂的显示对象，则运行时需要更多时间才能呈现。要测试需要重绘的量，请使用 Flash Player 调试版或 AIR 中的“显示重绘区域”功能。

有关提高重复操作的性能的详细信息，请参阅以下文章：

- [Writing well-behaved, efficient, AIR applications](#)（Arno Gourdol 编写的文章和示例应用程序）

更多帮助主题

第 55 页的“[隔离行为](#)”

补间症状



要节省 CPU 电量，请限制补间的使用，这可以节省 CPU 处理、内存并延长电池寿命。

在台式机上生成 **Flash** 内容的设计人员和开发人员容易在其应用程序中使用许多补间动画。在性能较低的移动设备上生成内容时，尝试尽量减少使用补间动画。限制补间动画的使用有助于提高内容在低级设备上的运行速度。

第 4 章 : ActionScript 3.0 性能

Vector 类和 Array 类



尽可能使用 Vector 类而不是 Array 类。

Vector 类的读写访问速度比 Array 类快。

一个简单的基准就可说明 Vector 类与 Array 类相比的优势所在。以下代码显示 Array 类的基准：

```
var coordinates:Array = new Array();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 107
```

以下代码显示 Vector 类的基准：

```
var coordinates:Vector.<Number> = new Vector.<Number>();
var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 72
```

通过为矢量分配特定长度并将其长度设为固定值，可进一步优化此示例：

```
// Specify a fixed length and initialize its length
var coordinates:Vector.<Number> = new Vector.<Number>(300000, true);

var started:Number = getTimer();

for (var i:int = 0; i< 300000; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 48
```

如果不提前指定矢量的大小，则矢量大小将在矢量用完空间后增加。每次矢量大小增加时，都将分配一个新的内存块。矢量的当前内容会复制到新内存块中。这种额外的分配和复制数据会降低性能。以上代码通过指定矢量的原始大小针对性能进行了优化。但是，代码没有针对可维护性进行优化。为了还能够提高可维护性，将重用的值存储在常量中：

```
// Store the reused value to maintain code easily
const MAX_NUM:int = 300000;

var coordinates:Vector.<Number> = new Vector.<Number>(MAX_NUM, true);
var started:Number = getTimer();

for (var i:int = 0; i< MAX_NUM; i++)
{
    coordinates[i] = Math.random()*1024;
}

trace(getTimer() - started);
// output: 47
```

尽可能尝试使用 **Vector** 对象 API，因为它们的运行速度可能更快。

绘图 API



使用绘图 API 可加快代码执行。

Flash Player 10 和 AIR 1.5 提供了一个新的绘图 API，使用它可获得更好的代码执行性能。此新的 API 不提供呈现性能改进，但是可以大大减少必须编写的代码行数。代码行越少，**ActionScript** 执行性能越好。

此新绘图 API 包含下列方法：

- `drawPath()`
- `drawGraphicsData()`
- `drawTriangles()`

注：本讨论不会重点介绍 `drawTriangles()` 方法，此方法与 3D 有关。但是，此方法可以改进 **ActionScript** 性能，因为它可以处理本机纹理映射。

以下代码为每个正在绘制的代码行明确地调用相应的方法：

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.moveTo ( coords[0], coords[1] );
container.graphics.lineTo ( coords[2], coords[3] );
container.graphics.lineTo ( coords[4], coords[5] );
container.graphics.lineTo ( coords[6], coords[7] );
container.graphics.lineTo ( coords[8], coords[9] );

addChild( container );
```

以下代码的运行速度比上一示例快，因为它执行的代码行比较少。路径越复杂，使用 `drawPath()` 方法获得的性能越高：

```
var container:Shape = new Shape();
container.graphics.beginFill(0x442299);

var commands:Vector.<int> = Vector.<int>([1,2,2,2,2]);
var coords:Vector.<Number> = Vector.<Number>([132, 20, 46, 254, 244, 100, 20, 98, 218, 254]);

container.graphics.drawPath(commands, coords);

addChild( container );
```

drawGraphicsData() 方法可提供类似的性能改进。

事件捕获和冒泡



使用事件捕获和冒泡可以最大程度地减少事件处理函数。

ActionScript 3.0 中的事件模型引入了事件捕获和事件冒泡的概念。利用事件冒泡可帮助您优化 ActionScript 代码执行时间。您可以在一个对象（而不是多个对象）上注册事件处理函数以提高性能。

例如，想象创建这样一种游戏，在该游戏中用户必须以最快的速度单击苹果以将其销毁。该游戏将删除屏幕上各个被击中的苹果并为用户增加分数。要侦听由各个苹果调度的 MouseEvent.CLICK 事件，您可能会编写以下代码：

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    addChild ( currentApple );

    // Listen to the MouseEvent.CLICK event
    currentApple.addEventListener ( MouseEvent.CLICK, onAppleClick );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.currentTarget as InteractiveObject;
    currentAppleClicked.removeEventListener(MouseEvent.CLICK, onAppleClick );
    removeChild ( currentAppleClicked );
}
```

此代码对各个 Apple 实例调用 addEventListener() 方法。此外，它还会在用户单击每个苹果时使用 removeEventListener() 方法删除对应的侦听器。然而，ActionScript 3.0 中的事件模型为某些事件提供了一个捕获和冒泡阶段，允许您侦听来自父 InteractiveObject 的这些事件。因此，可以优化以上代码并在最大程度上减少对 addEventListener() 和 removeEventListener() 方法的调用次数。以下代码使用捕获阶段侦听来自父对象的事件：

```
const MAX_NUM:int = 10;
var sceneWidth:int = stage.stageWidth;
var sceneHeight:int = stage.stageHeight;
var currentApple:InteractiveObject;
var currentAppleClicked:InteractiveObject;
var container:Sprite = new Sprite();

addChild ( container );

// Listen to the MouseEvent.CLICK on the apple's parent
// Passing true as third parameter catches the event during its capture phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, true );

for ( var i:int = 0; i< MAX_NUM; i++ )
{
    currentApple = new Apple();
    currentApple.x = Math.random()*sceneWidth;
    currentApple.y = Math.random()*sceneHeight;
    container.addChild ( currentApple );
}

function onAppleClick ( e:MouseEvent ):void
{
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

上述代码不仅经过了简化，而且在很大程度进行了优化，只对父容器调用了一次 `addEventListener()` 方法。侦听器不再注册到 `Apple` 实例，因此不需要在单击苹果时将其删除。可通过停止事件传播（此操作将阻止继续传播事件）来进一步优化 `onAppleClick()` 处理函数：

```
function onAppleClick ( e:MouseEvent ):void
{
    e.stopPropagation();
    currentAppleClicked = e.target as InteractiveObject;
    container.removeChild ( currentAppleClicked );
}
```

冒泡阶段也可用于捕捉事件，方法是将 `false` 作为第三个参数传递到 `addEventListener()` 方法：

```
// Listen to the MouseEvent.CLICK on apple's parent
// Passing false as third parameter catches the event during its bubbling phase
container.addEventListener ( MouseEvent.CLICK, onAppleClick, false );
```

捕获阶段参数的默认值是 `false`，因此您可以将其忽略：

```
container.addEventListener ( MouseEvent.CLICK, onAppleClick );
```

处理像素



使用 `setVector()` 方法绘制像素。

当绘制像素时，使用 `BitmapData` 类的相应方法即可进行一些简单优化。快速绘制像素的一种方式是使用 `setVector()` 方法：

```
// Image dimensions
var width:int = 200;
var height:int = 200;
var total:int = width*height;

// Pixel colors Vector
var pixels:Vector.<uint> = new Vector.<uint>(total, true);

for ( var i:int = 0; i< total; i++ )
{
    // Store the color of each pixel
    pixels[i] = Math.random()*0xFFFFFFFF;
}

// Create a non-transparent BitmapData object
var myImage:BitmapData = new BitmapData ( width, height, false );
var imageContainer:Bitmap = new Bitmap ( myImage );

// Paint the pixels
myImage.setVector ( myImage.rect, pixels );
addChild ( imageContainer );
```

如果使用的是较慢的方法，如 `setPixel()` 或 `setPixel32()`，请使用 `lock()` 和 `unlock()` 方法加快运行速度。在以下代码中，使用了 `lock()` 和 `unlock()` 方法来改进性能：

```
var buffer:BitmapData = new BitmapData(200,200,true,0xFFFFFFFF);
var bitmapContainer:Bitmap = new Bitmap(buffer);
var positionX:int;
var positionY:int;

// Lock update
buffer.lock();
var starting:Number=getTimer();

for (var i:int = 0; i<2000000; i++)
{
    // Random positions
    positionX = Math.random()*200;
    positionY = Math.random()*200;
    // 40% transparent pixels
    buffer.setPixel32( positionX, positionY, 0x66990000 );
}

// Unlock update
buffer.unlock();
addChild( bitmapContainer );


trace( getTimer () - starting );
// output : 670
```

`BitmapData` 类的 `lock()` 方法可以锁定图像，并防止引用该图像的对象在 `BitmapData` 对象更改时进行更新。例如，如果 `Bitmap` 对象引用 `BitmapData` 对象，则可以锁定 `BitmapData` 对象，对其更改后再解锁。在 `BitmapData` 对象解锁之前，`Bitmap` 对象不会更改。要提高性能，请在对 `setPixel()` 或 `setPixel32()` 方法进行多次调用之前和之后使用此方法及 `unlock()` 方法。调用 `lock()` 和 `unlock()` 可防止屏幕进行不必要的更新。


注：如果处理的是位图（而不是显示列表）中的像素（双缓冲），有时该技术不会提高性能。如果位图对象没有引用位图缓冲区，则使用 `lock()` 和 `unlock()` 不会提高性能。Flash Player 检测到未引用缓冲区，并且位图不会呈现在屏幕上。

遍历像素的方法（例如 `getPixel()`、`getPixel32()`、`setPixel()` 和 `setPixel32()`）可能速度很慢，特别是在移动设备上。如果可能，请使用在一次调用中检索所有像素的方法。要读取像素，请使用 `getVector()` 方法，它比 `getPixels()` 方法速度快。此外，请记住，尽可能使用依赖于 `Vector` 对象的 API，因为它们的运行速度可能更快。

正则表达式

 使用 `String` 类方法（如 `indexOf()`、`substr()` 或 `substring()`）代替正则表达式来实现基本字符串查找和提取。

可以使用正则表达式执行的某些操作也可以使用 `String` 类方法来完成。例如，要查找一个字符串是否包含另一个字符串，您可以使用 `String.indexOf()` 方法或使用正则表达式。但是，当 `String` 类方法可用时，它的运行速度比等效的正则表达式快且不需要创建其他对象。

 如果需要对元素进行分组，但不需要在结果中隔离该组的内容，请使用非捕获组（“`(?:xxxx)`”）代替正则表达式中的（“`(xxxx)`”）组。


通常，在复杂性属于中等的正则表达式中，将表达式各部分组合到一起。例如，在以下正则表达式模式中，括住文本“`ab`”的括号创建了一个组。因此，“`+`”限定符应用于该组，而不是单个字符：

```
/(ab)+/
```

默认情况下，“捕获”每个组的内容。作为执行正则表达式的结果的一部分，您可以在您的模式下获取各个组的内容。捕获这些组结果需要较长的时间并且需要占用更多内存，因为要创建将包含组结果的对象。作为一种替代方法，您可以使用非捕获组语法，即在左括号后包含一个问号和冒号。该语法指定这些字符作为一个组，但不会被捕获以获得结果：


```
/(?:ab)+/
```

与使用标准组语法相比，使用非捕获组语法速度更快且占用的内存更少。

 如果某正则表达式执行性能很差，考虑使用其他正则表达式模式。

有时，可使用多个正则表达式模式进行测试或标识相同的文本模式。出于各种不同的原因，某些模式的执行速度比其他备用方法快。如果您确定是正则表达式导致代码的运行速度没有达到必需的速度，请想出可获得相同结果的替代正则表达式模式。测试这些替代模式，以确定哪个运行速度最快。

其他优化

 对于 `TextField` 对象，请使用 `appendText()` 方法，而不要使用 `+=` 运算符。

当使用 `TextField` 类的 `text` 属性时，请使用 `appendText()` 方法，而不是 `+=` 运算符。使用 `appendText()` 方法可改进性能。

例如，以下代码使用 `+=` 运算符，循环需要 1120 毫秒才能完成：

```
addChild ( myTextField );

myTextField.autoSize = TextFieldAutoSize.LEFT;
var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.text += "ActionScript 3";
}

trace( getTimer() - started );
// output : 1120
```

在以下示例中，`+=` 运算符被替换为 `appendText()` 方法：


```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.appendText ( "ActionScript 3" );
}

trace( getTimer() - started );
// output : 847
```

现在，代码只需 847 毫秒即可完成。



请尽可能更新循环外的文本字段。

通过简单的技术即可进一步优化此代码。在每个循环中更新文本字段会使用很多内部处理。通过仅连接一个字符串并将其分配给循环外的文本字段，可大大减少运行代码所需的时间。此代码现在仅需要 2 毫秒即可完成：

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.text;

for (var i:int = 0; i< 1500; i++ )
{
    content += "ActionScript 3";
}

myTextField.text = content;

trace( getTimer() - started );
// output : 2
```

处理 HTML 文本时，前一种方法速度太慢，以致在某些情况下可能会在 Flash Player 中引发 Timeout 异常。例如，如果基础硬件速度太慢，则可能会引发异常。

注：Adobe® AIR® 不会引发此异常。

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();

for (var i:int = 0; i< 1500; i++ )
{
    myTextField.htmlText += "ActionScript <b>2</b>";
}

trace( getTimer() - started );
```

通过将此值分配给循环外的字符串，此代码将仅需要 29 毫秒即可完成：

```
var myTextField:TextField = new TextField();
addChild ( myTextField );
myTextField.autoSize = TextFieldAutoSize.LEFT;

var started:Number = getTimer();
var content:String = myTextField.htmlText;

for (var i:int = 0; i< 1500; i++ )
{
    content += "<b>ActionScript<b> 3";
}

myTextField.htmlText = content;

trace ( getTimer() - started );
// output : 29
```

注: 在 Flash Player 10.1 和 AIR 2.5 中, `String` 类已进行了改进, 所以字符串使用的内存较少。



尽可能避免使用中括号运算符。

使用中括号运算符可能会降低性能。将您的引用存储在本地变量中可避免使用该运算符。以下代码示例演示了使用中括号运算符的效率很低:

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();

for ( i = 0; i< lng; i++ )
{
    arraySprite[i].x = Math.random()*stage.stageWidth;
    arraySprite[i].y = Math.random()*stage.stageHeight;
    arraySprite[i].alpha = Math.random();
    arraySprite[i].rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 16
```

以下优化的版本减少了对中括号运算符的使用:

```
var lng:int = 5000;
var arraySprite:Vector.<Sprite> = new Vector.<Sprite>(lng, true);
var i:int;

for ( i = 0; i< lng; i++ )
{
    arraySprite[i] = new Sprite();
}

var started:Number = getTimer();
var currentSprite:Sprite;

for ( i = 0; i< lng; i++ )
{
    currentSprite = arraySprite[i];
    currentSprite.x = Math.random()*stage.stageWidth;
    currentSprite.y = Math.random()*stage.stageHeight;
    currentSprite.alpha = Math.random();
    currentSprite.rotation = Math.random()*360;
}

trace( getTimer() - started );
// output : 9
```



尽可能使用内联代码以减少代码中函数的调用次数。

调用函数成本很高。尝试通过移动内联代码来减少函数的调用次数。移动内联代码是优化代码以获得良好性能的一种好方法。但请注意，使用内联代码可能会使代码很难重复使用，还会增加 SWF 文件的大小。一些函数调用（如 **Math** 类方法）很容易以内联的方式移动。以下代码使用 **Math.abs()** 方法计算绝对值：

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = Math.abs ( currentValue );
}

trace( getTimer() - started );
// output : 70
```

由 **Math.abs()** 执行的计算可以手动完成并以内联方式移动：

```
const MAX_NUM:int = 500000;
var arrayValues:Vector.<Number>=new Vector.<Number>(MAX_NUM,true);
var i:int;

for (i = 0; i< MAX_NUM; i++)
{
    arrayValues[i] = Math.random()-Math.random();
}

var started:Number = getTimer();
var currentValue:Number;

for (i = 0; i< MAX_NUM; i++)
{
    currentValue = arrayValues[i];
    arrayValues[i] = currentValue > 0 ? currentValue : -currentValue;
}

trace( getTimer() - started );
// output : 15
```

以内联方式移动函数调用可使代码运行速度提高四倍以上。这种方法适用于多种情况，但要注意它对可重用性和可维护性的影响。

注：代码大小对播放器的整体执行情况有很大影响。如果应用程序包含大量 **ActionScript** 代码，则虚拟机将花费大量时间验证代码和 **JIT** 编译。属性查找的速度可能较慢，原因在于继承层次较深且内部缓存很容易失败。要降低代码大小，请避免使用 **Adobe® Flex®** 框架、**TLF** 框架库或任何大型第三方 **ActionScript** 库。



避免计算循环中的语句。

不计算循环中的语句也可实现优化。以下代码遍历数组，但未进行优化，因为在每次遍历时都需要计算数组长度：

```
for (var i:int = 0; i< myArray.length; i++)
{
}
```

最好存储该值并重复使用：

```
var lng:int = myArray.length;

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



对 **while** 循环使用相反的顺序。

以相反顺序进行 **while** 循环的速度比正向循环快：

```
var i:int = myArray.length;

while (--i > -1)
{
}
```

这些技巧提供了几种优化 **ActionScript** 的方法，说明了单行代码如何影响性能和内存。还可能存在着许多其他的 **ActionScript** 优化。有关详细信息，请参阅以下链接：<http://www.rozengain.com/blog/2007/05/01/some-actionscript-30-optimizations/>。

第 5 章：呈现性能

重绘区域

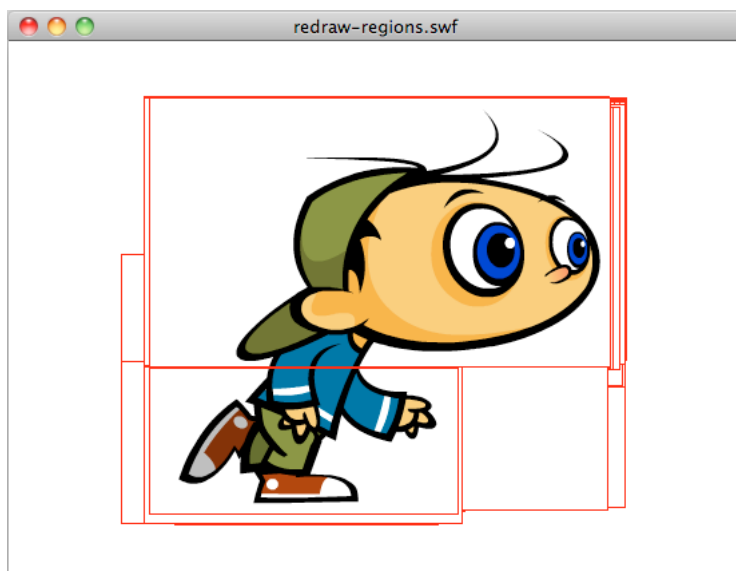


构建您的项目时始终使用重绘区域选项。

要改进呈现，务必在构建您的项目时使用重绘区域选项。使用此选项可以查看 **Flash Player** 正在呈现和处理的区域。通过在 **Flash Player** 的调试版本的上下文菜单中选择“显示重绘区域”可以启用此选项。

注：“显示重绘区域”选项在 **Adobe AIR** 或 **Flash Player** 发行版中不提供。（在 **Adobe AIR** 中，上下文菜单仅在台式机应用程序中提供，但没有内置项或标准项，如“显示重绘区域”。）

下图演示了此选项通过时间轴上的简单动画 **MovieClip** 启用：



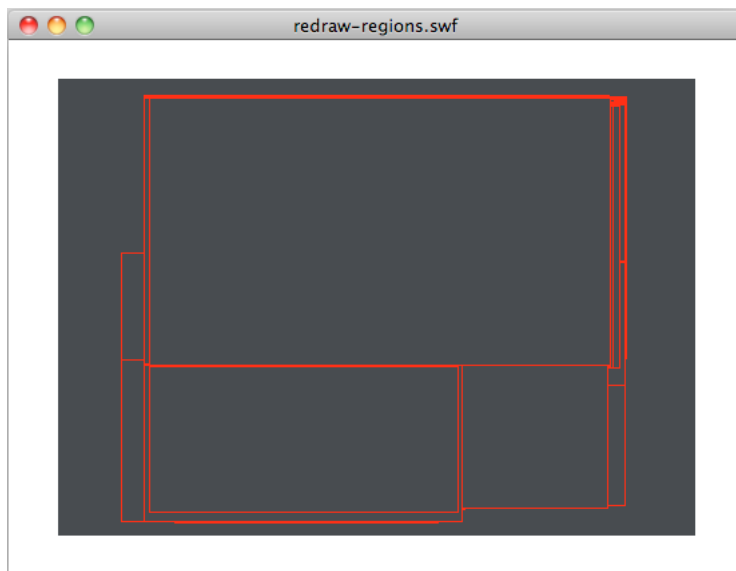
启用重绘区域选项

您还可以使用 `flash.profiler.showRedrawRegions()` 方法以编程方式启用此选项：

```
// Enable Show Redraw Regions
// Blue color is used to show redrawn regions
flash.profiler.showRedrawRegions ( true, 0x0000FF );
```

在 **Adobe AIR** 应用程序中，此方法是启用重绘区域选项的唯一方式。

使用重绘区域确定优化机会。请记住，虽然未显示某些显示对象，但这些对象仍会占用 **CPU** 周期，因为它们仍在呈现。下图演示了此概念。黑色的矢量形状包括动画运行的字符。此图像显示显示对象尚未从显示列表中删除并仍在呈现。这将浪费 **CPU** 周期：



重绘区域

要改进性能，请将隐藏的运行字符的 **visible** 属性设置为 **false** 或将其从显示列表中完全删除。您还应该停止其时间轴。这些步骤可确保显示对象被冻结并使用最小的 CPU 电量。

请记住，在整个开发周期中使用重绘区域选项。使用此选项不会在项目结尾时出现重绘区域多余和优化区域丢失的情况。

更多帮助主题

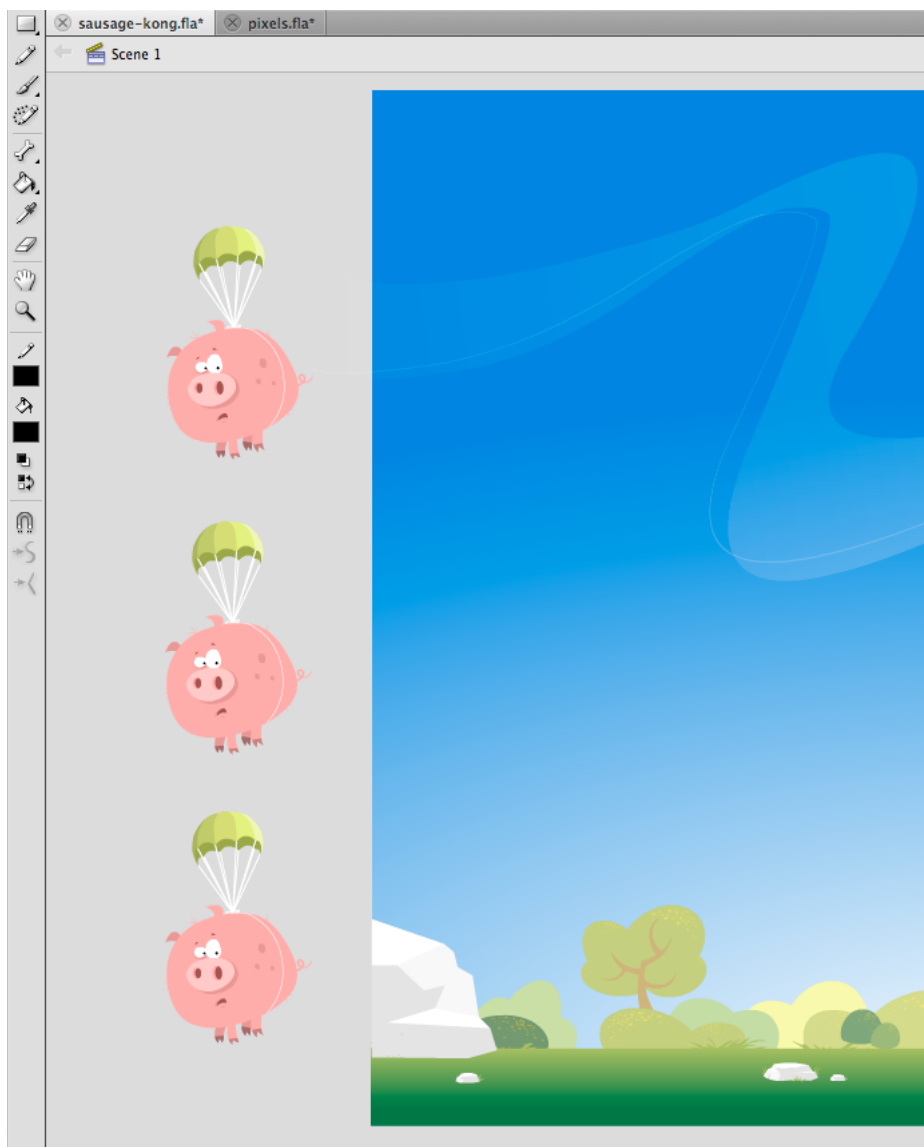
第 24 页的“[冻结和解冻对象](#)”

后台内容



避免将内容放在后台。而是在需要时将对象放在显示列表中。

如果可能，尽量不要将图形内容放在后台。设计人员和开发人员通常会将元素放在后台，以便在应用程序的生命周期中重复使用资源。下图说明这一常见技巧：



后台内容

即使后台元素不显示在屏幕上，也不呈现出来，它们仍存在于显示列表中。运行时继续在这些元素上运行内部测试，以确保它们仍位于后台，而且用户不与它们交互。因此，要尽可能避免将对象放在后台，而应该将它们从显示列表中删除。

影片品质



使用适当的舞台品质设置可提高呈现性能。

当为带有小型屏幕的移动设备（例如电话）开发内容时，图像品质没有开发台式机应用程序时那么重要。适当设置舞台品质可以提高呈现性能。

舞台品质可以使用以下设置：

- **StageQuality.LOW**：使播放速度优先于外观，并且不使用消除锯齿功能。在用于台式机或电视的 **Adobe AIR** 中不支持此设置。
- **StageQuality.MEDIUM**：应用一些消除锯齿功能，但并不会平滑缩放位图。此设置是移动设备上 **AIR** 的默认值，但在用于台式机的 **AIR** 或 **AIR for TV** 中不支持。
- **StageQuality.HIGH**：（台式机的默认值）使外观优先于播放速度，并始终应用消除锯齿功能。如果 **SWF** 文件不包含动画，则会对缩放位图进行平滑处理；如果 **SWF** 文件包含动画，则不会对缩放位图进行平滑处理。
- **StageQuality.BEST**：提供最佳的显示品质，而不考虑播放速度。所有输出都已消除锯齿，而且始终对缩放位图进行平滑处理。

使用 **StageQuality.MEDIUM** 通常就可以为移动设备上的应用程序提供足够高的品质，但在某些情况下，使用 **StageQuality.LOW** 也可以提供足够高的品质。**Flash Player 8** 及更高版本可以精确呈现消除锯齿的文本，甚至将舞台品质设置为 **LOW** 也可以达到这种效果。

注：在某些移动设备上，即使将品质设置为 **HIGH**，仍会使用 **MEDIUM** 在 **Flash Player** 应用程序中获得更好的性能。但是，由于移动设备屏幕通常具有更高的 **dpi**，所以将品质设置为 **HIGH** 通常也不会产生明显的区别。（不同的设备 **dpi** 可能也有所不同。）

在下图中，影片品质设置为 **MEDIUM**，且文本呈现设置为“动画消除锯齿”：



中等舞台品质，且文本呈现设置为“动画消除锯齿”

因为没有使用相应的文本呈现设置，所以舞台品质设置会影响文本品质。

运行时允许您将文本呈现设置为“可读性消除锯齿”。无论您使用哪种舞台品质设置，此设置都可以保持完美的（消除锯齿）文本品质：



低舞台品质，且文本呈现设置为“可读性消除锯齿”

通过将文本呈现设置为“位图文本”（未消除锯齿）可获得相同的呈现品质：



Here is some sample text

低舞台品质，且文本呈现设置为“位图文本”（未消除锯齿）

最后两个示例表明，无论您使用哪种舞台品质设置，都可以获得高品质文本。Flash Player 8 及更高版本中都提供了该功能，并可在移动设备上使用。请记住，在某些设备上，Flash Player 10.1 可自动切换到 StageQuality.MEDIUM 以提高性能。

Alpha 混合



尽可能避免使用 alpha 属性。

使用 alpha 属性时避免使用要求 alpha 混合的效果，例如淡化效果。当显示对象使用 alpha 值混合处理时，运行时必须将每个堆叠显示对象的颜色值与背景色混合起来，以确定最终颜色。因此，alpha 值混合处理可能比绘制不透明颜色占用更多的处理器资源。这种额外的计算可能影响慢速设备上的性能。尽可能避免使用 alpha 属性。

更多帮助主题

第 46 页的“[位图缓存](#)”

第 58 页的“[呈现文本对象](#)”

应用程序帧速率




通常，使用尽可能低的帧速率可以提高性能。

应用程序的帧速率可确定每个“应用程序代码和呈现”循环的可用时间，如在第 1 页的“[运行时代码执行基本原理](#)”中所述。帧速率越高，创建的动画越流畅。但是，如果没有动画或其他可视更改，则通常没有必要使用高帧速率。相比较低的帧速率，较高的帧速率会耗用更多的 CPU 资源和电池能量。


以下是一些关于适合应用程序的默认帧速率的常规指导方针：

- 如果使用的是 Flex 框架，将起始帧速率保持为默认值。
- 如果应用程序包括动画，适当的帧速率至少是 20 帧 / 秒。通常不必超过 30 帧 / 秒。
- 如果应用程序不包括动画，12 帧 / 秒的帧速率可能就足够了。

“尽可能低的帧速率”可能因应用程序的当前活动而异。有关详细信息，请参阅下一个技巧“[动态更改应用程序的帧速率](#)”。

 如果视频是应用程序中唯一的动态内容，请使用低帧速率。

运行时以本机帧速率播放加载的视频内容，而与应用程序的帧速率无关。如果应用程序不包含任何动画或其他快速更改的可视内容，使用低帧速率不会降低用户界面的体验。

 动态更改应用程序的帧速率。

可以在项目或编译器设置中定义应用程序的初始帧速率，但是帧速率不会固定在该值。可以通过设置 `Stage.frameRate` 属性（或 `Flex` 中的 `WindowedApplication.frameRate` 属性）来更改帧速率。

根据应用程序的当前需要更改帧速率。例如，在应用程序不执行任何动画时，请降低帧速率。在要开始动画转换时提高帧速率。同样，如果应用程序正在后台运行（已失去焦点），您通常可以进一步降低帧速率。用户可能关注其他应用程序或任务。

以下是一些常规指导，用于开始确定适用于不同类型活动的合适帧速率：

- 如果使用的是 **Flex** 框架，将起始帧速率保持为默认值。
- 播放动画时，将帧速率设置为至少 20 帧 / 秒。通常不必超过 30 帧 / 秒。
- 不播放动画时，12 帧 / 秒的帧速率可能就足够了。
- 系统以本机帧速率播放加载的视频，而与应用程序帧速率无关。如果视频是应用程序中唯一的移动内容，12 帧 / 秒的帧速率可能就足够了。
- 如果应用程序没有输入焦点，5 帧 / 秒的帧速率可能就足够了。
- 当 AIR 应用程序不可见时，2 帧 / 秒或更低的帧速率可能就足够了。例如，如果应用程序最小化，则应用此准则。在台式机设备上，如果本机窗口的 `visible` 属性设置为 `false`，也应用此准则。

对于 **Flex** 中的内置应用程序，`spark.components.WindowedApplication` 类已经内置了用于动态更改应用程序的帧速率的支持功能。当应用程序处于非活动状态时，`backgroundFrameRate` 属性将定义应用程序的帧速率。默认值为 1，该值将使用 **Spark** 框架构建的应用程序的帧速率更改为 1 帧 / 秒。您可以通过设置 `backgroundFrameRate` 属性来更改后台帧速率。您可以将此属性设置为其他值，或将其设置为 -1 以关闭自动帧速率限制。

有关动态更改应用程序的帧速率的详细信息，请参阅下列文章：

- [Reducing CPU usage in Adobe AIR](#)（Adobe 开发人员中心文章和示例代码，作者：Jonnie Hallman）
- [Writing well-behaved, efficient, AIR applications](#)（Arno Gourdol 编写的文章和示例应用程序）


Grant Skinner 创建了一个帧速率节流器类。当应用程序处于后台时，您可以在应用程序中使用该类自动降低帧速率。有关更多信息以及下载 `FramerateThrottler` 类的源代码，请参阅 Grant 的文章 `Idle CPU Usage in Adobe AIR and Flash Player`，网址是 http://gskinner.com/blog/archives/2009/05/idle_cpu_usage.html。

自适应帧速率

在编译 SWF 文件时，应为影片设置特定的帧速率。在 CPU 速率较低的受限环境中，帧速率有时会在播放过程中降低。为了保留用户可以接受的帧速率，运行时会跳过某些帧的呈现。跳过某些帧的呈现可以防止帧速率降低到可接受的值以下。

注：在这种情况下，运行时不会跳过某些帧，只是跳过某些帧中内容的呈现。仍然会执行代码并更新显示列表，但不会将更新显示在屏幕上。无法指定阈值 `fps` 值来指示在运行时无法保持帧速率的稳定性时要跳过的帧数。

位图缓存

 在适当的时候，对复杂的矢量内容使用位图缓存功能。

使用位图缓存功能可实现良好的优化。此功能缓存矢量对象，将其作为位图在内部呈现，并使用该位图进行呈现。此结果会显著提高呈现的性能，但需要占用大量内存。针对复杂的矢量内容使用位图缓存功能，如复杂渐变或文本。

为包含复杂的矢量图形（例如文本或渐变）的动画对象打开位图缓存可提高性能。但是，如果在显示对象（如播放其时间轴的视频剪辑）中启用了位图缓存，您将获得相反的效果。在各个帧上，运行时必须更新缓存的位图，然后在屏幕上重绘该位图，这一过程需要许多 CPU 周期。仅当缓存的位图可以一次生成，且随后无需更新即可使用时，才适合使用位图缓存功能。

为 **Sprite** 对象打开位图缓存后，移动该对象不会使运行时重新生成缓存的位图。更改对象的 **x** 和 **y** 属性不会导致重新生成。然而，任何试图旋转、缩放对象或更改其 **alpha** 值的行为都将导致运行时重新生成缓存的位图，从而降低性能。

注：AIR 和 **Packager for iPhone** 中提供的 **DisplayObject.cacheAsBitmapMatrix** 属性没有此限制。通过使用 **cacheAsBitmapMatrix** 属性，您可以旋转、缩放、倾斜显示对象以及更改其 **Alpha** 值，而不触发重新生成位图。

缓存位图占用的内存大于常规影片剪辑实例。例如，如果舞台上的影片剪辑为 250 x 250 像素，缓存它可能会使用 250 KB 内存，而未缓存它只需 1 KB。

以下示例使用一个包含苹果图像的 **Sprite** 对象。以下类将附加到苹果符号：

```
package org.bytearray.bitmap
{
    import flash.display.Sprite;
    import flash.events.Event;

    public class Apple extends Sprite
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function Apple ()
        {
            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener (Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

以上代码使用的是 **Sprite** 类而不是 **MovieClip** 类，因为并非每个苹果都需要时间轴。为了获得最佳性能，请尽可能使用最轻型的对象。接下来，将使用以下代码实例化此类：

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(MouseEvent.CLICK,createApples);
stage.addEventListener(KeyboardEvent.KEY_DOWN,cacheApples);

const MAX_NUM:int = 100;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

function createApples(e:MouseEvent):void
{
    for (var i:int = 0; i< MAX_NUM; i++)
    {
        apple = new Apple();

        holder.addChild(apple);
    }
}

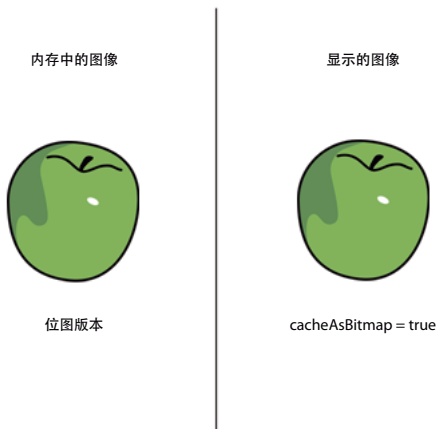
function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holder.numChildren;

        for (var i:int = 0; i < lng; i++)
        {
            apple = holder.getChildAt (i) as Apple;

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

当用户单击鼠标时，将创建苹果，而不需要进行缓存。当用户按下 **C** 键（键代码为 67），苹果矢量将作为位图缓存并显示在屏幕上。当 **CPU** 较慢时，此技术可显著提高台式机和移动设备上的呈现性能。

但是，虽然使用位图缓存功能可以提高呈现性能，但也会快速占用大量内存。缓存对象后，其表面将捕获为透明位图并存储在内存中，如下图所示：

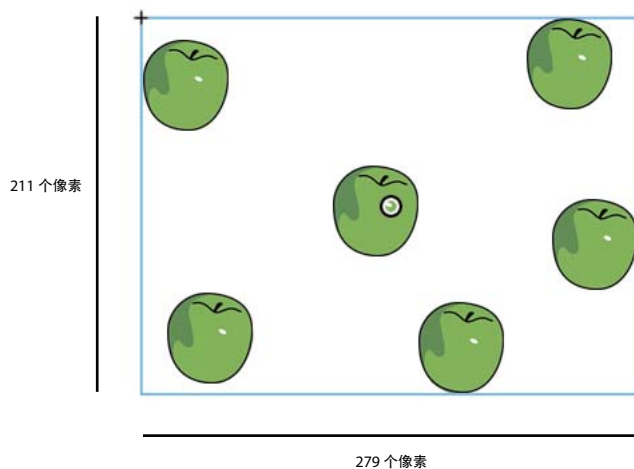


对象及其存储在内存中的表面位图

Flash Player 10.1 和 AIR 2.5 优化内存使用的方法与第 17 页的“[滤镜和动态位图卸载](#)”中介绍的方法相同。如果隐藏缓存的显示对象或将其置于屏幕之外，则在一段时间未使用后释放其位图占用的内存。

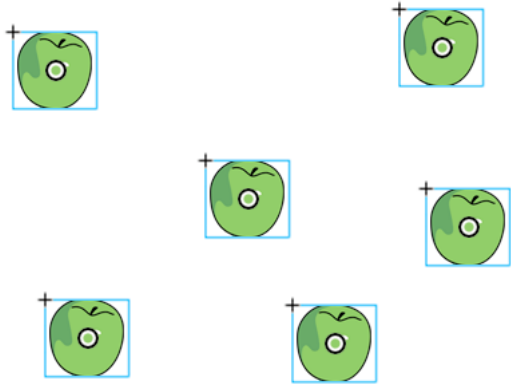
注：如果将显示对象的 `opaqueBackground` 属性设置为特定颜色，运行时会将显示对象视为不透明。当运行时与 `cacheAsBitmap` 属性一起使用时，它将在内存中创建一个非透明的 32 位位图。将 `alpha` 通道设置为 `0xFF` 可提高性能，因为在屏幕上绘制位图时不要求透明度。避免使用 `alpha` 混合可提高呈现速度。如果当前屏幕深度限制为 16 位，则内存中的位图会存储为 16 位图像。使用 `opaqueBackground` 属性不会隐式激活位图缓存。

要节省内存，请使用 `cacheAsBitmap` 属性，并对每个显示对象而不是容器激活该属性。对容器激活位图缓存会导致最终位图占用更多内存，从而创建一个尺寸为 211 x 279 像素的透明位图。此图像大约占用 229 KB 内存：



对容器激活位图缓存

此外，通过缓存容器，如果任何苹果开始在帧上移动时，您将面临在内存中更新整个位图的风险。对各个实例激活位图缓存会导致在内存中缓存 6 个 7KB 的表面，总共仅使用 42 KB 内存：



对实例激活位图缓存

访问显示列表中的各个苹果实例并调用 `getChildAt()` 方法，会将引用存储在 `Vector` 对象中以便于访问：

```
import org.bytearray.bitmap.Apple;

stage.addEventListener(KeyboardEvent.KEY_DOWN, cacheApples);

const MAX_NUM:int = 200;
var apple:Apple;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<Apple> = new Vector.<Apple>(MAX_NUM, true);

for (var i:int = 0; i < MAX_NUM; i++)
{
    apple = new Apple();

    holder.addChild(apple);

    holderVector[i] = apple;
}

function cacheApples(e:KeyboardEvent):void
{
    if (e.keyCode == 67)
    {
        var lng:int = holderVector.length

        for (var i:int = 0; i < lng; i++)
        {
            apple = holderVector[i];

            apple.cacheAsBitmap = Boolean(!apple.cacheAsBitmap);
        }
    }
}
```

请记住，如果不在各个帧上旋转、缩放或更改缓存的内容，则位图缓存可提高呈现效果。但是，对于 X 和 Y 轴上的平移之外的任何转换，则不会提高呈现性能。在这些情况下，每次在显示对象上发生转换时，Flash Player 都将更新缓存的位图副本。更新缓存副本会导致占用大量 CPU、降低性能以及消耗大量电池电量。同样，AIR 或 Packager for iPhone 中的 `cacheAsBitmapMatrix` 属性也没有此限制。

以下代码会更改移动方法中的 `alpha` 值，这会更改每个帧中苹果的不透明度：

```
private function handleMovement(e:Event):void
{
    alpha = Math.random();
    x -= (x - destinationX)*.5;
    y -= (y - destinationY)*.5;

    if (Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
        initPos();
}
```

使用位图缓存会导致性能降低。更改 `alpha` 值会强制运行时在修改 `alpha` 值时更新内存中的缓存位图。

滤镜依赖每次缓存的影片剪辑的播放头移动时更新的位图。因此，使用滤镜会自动将 `cacheAsBitmap` 属性设置为 `true`。下图演示了一个动画影片剪辑：



动画影片剪辑

请避免在动画内容中使用滤镜，因为这可能导致性能问题。在下图中，设计人员添加了一个投影滤镜：



使用投影滤镜的动画影片剪辑

因此，如果播放影片剪辑内部的时间轴，则必须重新生成位图。如果以简单的 `x` 或 `y` 转换以外的任何方式修改内容，则也必须重新生成位图。每个帧都会强制运行时重绘位图，这需要更多的 CPU 资源、导致性能非常低，并缩短电池寿命。

Paul Trani 提供了以下培训视频，演示了如何使用 Flash Professional 和 ActionScript 来优化使用位图的图形：

- [优化图形](#)
- [使用 ActionScript 优化图形](#)

AIR 中的缓存位图转换矩阵



当在移动 AIR 应用程序中使用缓存位图时，设置 `cacheAsBitmapMatrix` 属性。

在 AIR 移动配置文件中，您可以向显示对象的 `cacheAsBitmapMatrix` 属性分配一个矩阵对象。设置此属性后，您可以在不重新生成缓存位图的情况下对对象应用任何二维转换。您还可以在不重新生成缓存位图的情况下更改 `alpha` 属性。另外，`cacheAsBitmap` 属性必须设置为 `true`，对象不得设置任何 3D 属性。

设置 `cacheAsBitmapMatrix` 属性会生成缓存位图，即使显示对象不在屏幕上、在视图中隐藏或其 `visible` 属性设置为 `false` 也是如此。使用一个包含其他转换的矩阵对象重置 `cacheAsBitmapMatrix` 属性也会重新生成缓存位图。

应用到 `cacheAsBitmapMatrix` 属性上的矩阵转换会应用到显示对象上，因为它已呈现到位图缓存中。因此，如果转换中包含一个 `2x` 缩放，位图呈现就会是矢量呈现大小的两倍。渲染器将反向转换应用于缓存位图，以便最终显示外观相同。您可以将缓存位图缩小，以减少内存的使用，但这样做有可能导致呈现失真。您还可以在某些情况下放大位图，以提高呈现质量，但这样会增加内存的使用。总之，使用不应用转换的单位矩阵可避免外观发生变更，如下例中所示：

```
displayObject.cacheAsBitmap = true;  
displayObject.cacheAsBitmapMatrix = new Matrix();
```

一旦设置了 `cacheAsBitmapMatrix` 属性，您就可以缩放、倾斜、旋转和平移对象，而不触发重新生成位图。

还可以在 0 和 1 的范围内更改 `alpha` 值。如果使用颜色转换，通过 `transform.colorTransform` 属性更改 `alpha` 值，转换对象中使用的 `alpha` 必须位于 0 和 255 之间。以任何其他方式更改颜色转换都会重新生成缓存位图。

在针对移动设备创建的内容中，只要将 `cacheAsBitmap` 设置为 `true`，就必须设置 `cacheAsBitmapMatrix` 属性。但是，请考虑以下潜在的缺陷。旋转对象后，无论是缩放的还是倾斜的，最终呈现会显示出相比正常矢量呈现而言所表现的位图缩放或锯齿失真。

手动位图缓存



使用 `BitmapData` 类创建自定义位图缓存行为。

以下示例重用了显示对象的单一栅格化位图版本并引用了同一个 `BitmapData` 对象。当缩放每个显示对象时，不会更新和重绘内存中的原始 `BitmapData` 对象。此方法可节省 CPU 资源并加快应用程序的运行速度。缩放显示对象时，将拉伸其中包含的位图。

以下是更新后的 `BitmapApple` 类：


```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.events.Event;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super(buffer);

            addEventListener(Event.ADDED_TO_STAGE,activation);
            addEventListener(Event.REMOVED_FROM_STAGE,deactivation);
        }

        private function activation(e:Event):void
        {
            initPos();
            addEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function deactivation(e:Event):void
        {
            removeEventListener(Event.ENTER_FRAME,handleMovement);
        }

        private function initPos():void
        {
            destinationX = Math.random()*(stage.stageWidth - (width>>1));
            destinationY = Math.random()*(stage.stageHeight - (height>>1));
        }

        private function handleMovement(e:Event):void
        {
            alpha = Math.random();

            x -= (x - destinationX)*.5;
            y -= (y - destinationY)*.5;

            if ( Math.abs(x - destinationX) < 1 && Math.abs(y - destinationY) < 1)
                initPos();
        }
    }
}
```

仍在各个帧上修改 **alpha** 值。以下代码将原始源缓冲区传递给各个 **BitmapApple** 实例：

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x, -bounds.y);

var buffer:BitmapData = new BitmapData(source.width+1, source.height+1, true, 0);
buffer.draw(source, mat);

var bitmapApple:BitmapApple;

for (var i:int = 0; i < MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}
```

此技术仅占用少量内存，因为只使用内存中单个缓存的位图并由所有 **BitmapApple** 实例共享。此外，不管对 **BitmapApple** 实例做任何修改（例如 **alpha**、旋转或缩放），原始源位图永远不会更新。使用此技术可防止性能降低。

要获得平滑的最终位图，请将 **smoothing** 属性设置为 **true**：

```
public function BitmapApple(buffer:BitmapData)
{
    super (buffer);

    smoothing = true;

    addEventListener(Event.ADDED_TO_STAGE, activation);
    addEventListener(Event.REMOVED_FROM_STAGE, deactivation);
}
```

调整舞台品质也可以提高性能。将舞台品质设置为 **HIGH**，进行栅格化处理，然后切换到 **LOW**：

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild ( holder );

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds ( source );

var mat:Matrix = new Matrix();
mat.translate ( -bounds.x, -bounds.y );

var buffer:BitmapData = new BitmapData ( source.width+1, source.height+1, true, 0 );

stage.quality = StageQuality.HIGH;

buffer.draw ( source, mat );

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i< MAX_NUM; i++ )
{
    bitmapApple = new BitmapApple( buffer );

    holderVector[i] = bitmapApple;

    holder.addChild ( bitmapApple );
}
```

在将矢量绘制为位图前后切换舞台品质是一项功能强大的技术，可以在屏幕上获得消除锯齿的内容。无论最终舞台品质如何，此技术都有效。例如，您可以通过使用消除锯齿文本，甚至通过将舞台品质设置为 **LOW** 来获取消除锯齿的位图。使用 **cacheAsBitmap** 属性时不能实现此技术。在这种情况下，将舞台品质设置为 **LOW** 会更新矢量品质，从而更新内存中的位图表面，并更新最终品质。

隔离行为



尽可能隔离事件，例如单个处理函数中的 **Event.ENTER_FRAME** 事件。

通过在单个处理函数的 **Apple** 类中隔离 **Event.ENTER_FRAME** 事件，可进一步优化此代码。此技术可节省 **CPU** 资源。以下示例介绍了一种不同的方法，在该方法中，**BitmapApple** 类不再处理移动行为：

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;

    public class BitmapApple extends Bitmap
    {
        private var destinationX:Number;
        private var destinationY:Number;

        public function BitmapApple(buffer:BitmapData)
        {
            super (buffer);

            smoothing = true;
        }
    }
}
```

以下代码对苹果进行了实例化，并在单个处理函数中处理其移动行为：

```
import org.bytearray.bitmap.BitmapApple;

const MAX_NUM:int = 100;
var holder:Sprite = new Sprite();

addChild(holder);

var holderVector:Vector.<BitmapApple> = new Vector.<BitmapApple>(MAX_NUM, true);
var source:AppleSource = new AppleSource();
var bounds:Object = source.getBounds(source);

var mat:Matrix = new Matrix();
mat.translate(-bounds.x, -bounds.y);

stage.quality = StageQuality.BEST;

var buffer:BitmapData = new BitmapData(source.width+1, source.height+1, true, 0);
buffer.draw(source, mat);

stage.quality = StageQuality.LOW;

var bitmapApple:BitmapApple;

for (var i:int = 0; i < MAX_NUM; i++)
{
    bitmapApple = new BitmapApple(buffer);

    bitmapApple.destinationX = Math.random()*stage.stageWidth;
    bitmapApple.destinationY = Math.random()*stage.stageHeight;

    holderVector[i] = bitmapApple;

    holder.addChild(bitmapApple);
}

stage.addEventListener(Event.ENTER_FRAME, onFrame);
```

```
var lng:int = holderVector.length

function onFrame(e:Event):void
{
    for (var i:int = 0; i < lng; i++)
    {
        bitmapApple = holderVector[i];
        bitmapApple.alpha = Math.random();

        bitmapApple.x -= (bitmapApple.x - bitmapApple.destinationX) *.5;
        bitmapApple.y -= (bitmapApple.y - bitmapApple.destinationY) *.5;

        if (Math.abs(bitmapApple.x - bitmapApple.destinationX) < 1 &&
            Math.abs(bitmapApple.y - bitmapApple.destinationY) < 1)
        {
            bitmapApple.destinationX = Math.random()*stage.stageWidth;
            bitmapApple.destinationY = Math.random()*stage.stageHeight;
        }
    }
}
```

结果是单个 `Event.ENTER_FRAME` 事件即可处理移动，而不是使用 200 个处理函数来移动每个苹果。可轻松地暂停整个动画，这在游戏中非常有用。

例如，一个简单游戏可以使用以下处理函数：

```
stage.addEventListener(Event.ENTER_FRAME, updateGame);
function updateGame (e:Event):void
{
    gameEngine.update();
}
```

下一步是使苹果与鼠标或键盘交互，这要求修改 `BitmapApple` 类。

```
package org.bytearray.bitmap
{
    import flash.display.Bitmap;
    import flash.display.BitmapData;
    import flash.display.Sprite;

    public class BitmapApple extends Sprite
    {
        public var destinationX:Number;
        public var destinationY:Number;
        private var container:Sprite;
        private var containerBitmap:Bitmap;

        public function BitmapApple(buffer:BitmapData)
        {
            container = new Sprite();
            containerBitmap = new Bitmap(buffer);
            containerBitmap.smoothing = true;
            container.addChild(containerBitmap);
            addChild(container);
        }
    }
}
```

结果是 `BitmapApple` 实例与传统 `Sprite` 对象一样是交互式的。然而，这些实例链接到单个位图，在转换显示对象时不会重新采样。

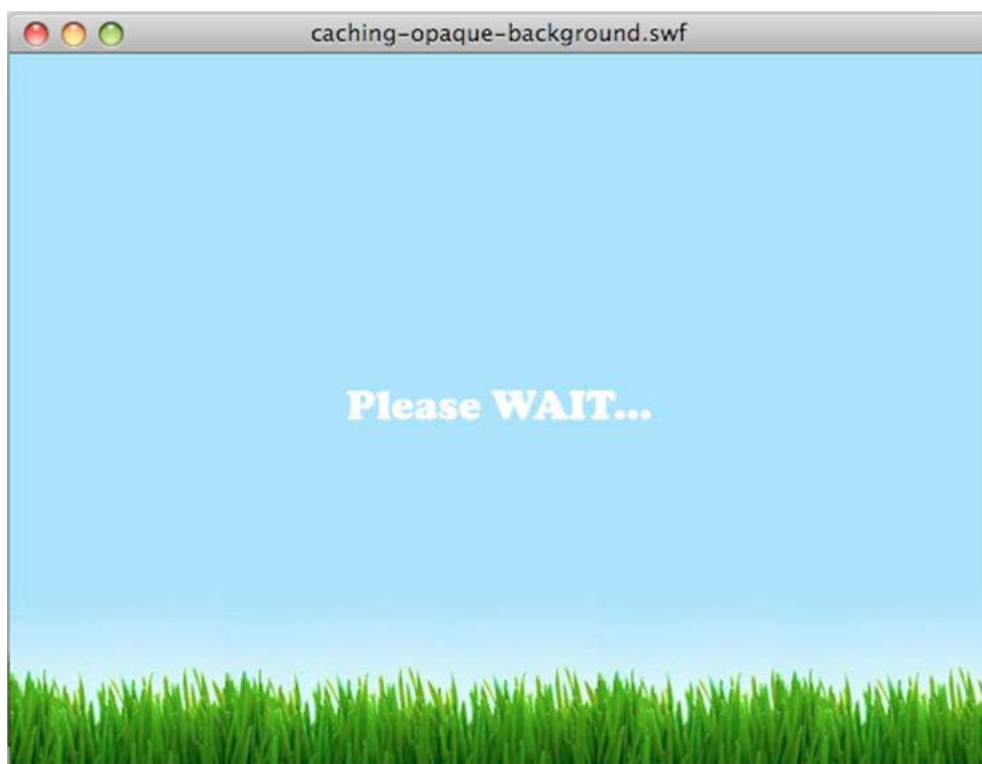
呈现文本对象

💡 使用位图缓存功能和 `opaqueBackground` 属性可提高文本呈现性能。

Flash 文本引擎提供一些很出色的优化。但是，需要使用许多类才能显示单个文本行。为此，使用 `TextLine` 类创建可编辑的文本字段需要大量内存和许多行 `ActionScript` 代码。`TextLine` 类最适合静态文本和不可编辑的文本，它们能够以更快的速度呈现并占用更少的内存。

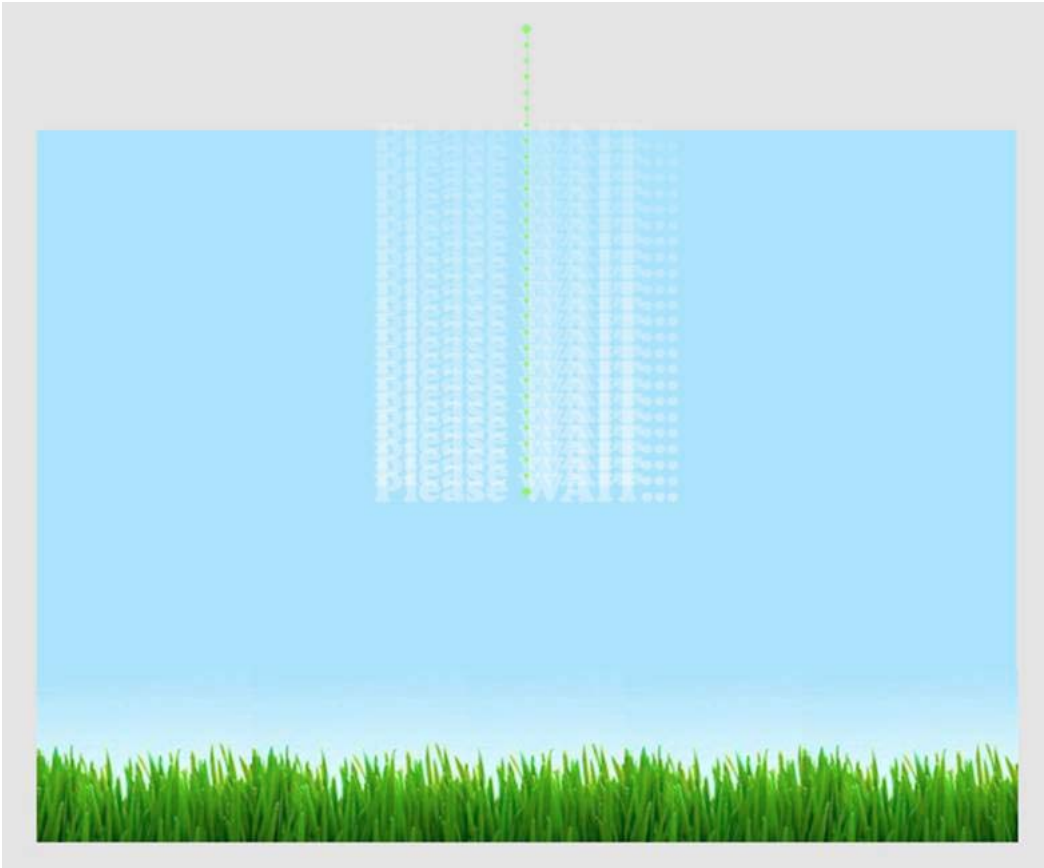
借助位图缓存功能，您可以将矢量内容缓存为位图来提高呈现性能。此功能对于复杂矢量内容以及需要进行处理才能呈现的文本内容都很有用。

以下示例显示了如何使用位图缓存功能和 `opaqueBackground` 属性来提高呈现性能。下图演示了在用户等待加载内容时显示的典型“欢迎”屏幕：



“欢迎”屏幕

下图演示了以编程方式应用到 `TextField` 对象的缓动。文本从场景的顶部缓动到场景的中心：



文本缓动

以下代码创建缓动。**preloader** 变量存储当前目标对象以在最大程度上减少属性查找，这可能会降低性能：

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

var destX:Number=stage.stageWidth/2;
var destY:Number=stage.stageHeight/2;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(preloader.y-destY)<1)
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
}
```

可以在此处以内联方式移动 **Math.abs()** 函数，以减少函数的调用次数并获得更多的性能改进。最佳做法是使用 **destX** 属性和 **destY** 属性的整数类型，以便拥有固定点值。通过使用整数类型，您可以获得完美的像素贴紧，而无需使用较慢的方法（如 **Math.ceil()** 或 **Math.round()**）手动对值进行四舍五入。此代码不会将坐标四舍五入为整数，因为不断对值进行四舍五入会使对象不能平滑移动。此对象可能不会平滑移动，因为坐标已采用各个帧上最接近的四舍五入的整数。然而，此技术在设置显示对象的最终位置时可能非常有用。不要使用以下代码：

```
// Do not use this code
var destX:Number = Math.round ( stage.stageWidth / 2 );
var destY:Number = Math.round ( stage.stageHeight / 2 );
```

以下代码的执行速度要快得多:

```
var destX:int = stage.stageWidth / 2;
var destY:int = stage.stageHeight / 2;
```

上一代码可通过使用移位运算符分解值来进一步优化:

```
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
```

通过位图缓存功能, 运行时可以更轻松地使用动态位图呈现对象。在当前示例中, 缓存了包含 **TextField** 对象的影片剪辑:

```
wait_mc.cacheAsBitmap = true;
```

提高性能的另一种方式是删除 **alpha** 透明度。在绘制透明位图图像时, **Alpha** 透明度增加了运行时的负担, 如前面的代码所示。您可以使用 **opaqueBackground** 属性指定背景颜色来避免此情况。

当使用 **opaqueBackground** 属性时, 在内存中创建的位图表面仍使用 32 位。然而, **alpha** 偏移量已设置为 255 且未使用透明度。因此, 在使用位图缓存功能时, **opaqueBackground** 属性不会减少内存使用, 但会提高呈现性能。以下代码包含所有优化:

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;

// Set the background to the color of the scene background
wait_mc.opaqueBackground = 0x8AD6FD;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;

function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
        e.currentTarget.removeEventListener ( Event.ENTER_FRAME, movePosition );
}
```

现在动画已经过优化, 还通过删除透明度优化了位图缓存。在移动设备上, 请考虑以下情况: 当使用位图缓存功能时, 在不同动画状态期间将舞台品质切换为 **LOW** 和 **HIGH**:


```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );
wait_mc.cacheAsBitmap = true;
wait_mc.opaqueBackground = 0x8AD6FD;

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth>>1;
var destY:int = stage.stageHeight>>1;
var preloader:DisplayObject;

function movePosition( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if (Math.abs(e.currentTarget.y-destY)<1)
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener( Event.ENTER_FRAME, movePosition );
    }
}
```

但是，在这种情况下，更改舞台品质会强制运行时重新生成 **TextField** 对象的位图表面以与当前舞台品质匹配。因此，最好不要在使用位图缓存功能时更改舞台品质。

此处可能已经使用了手动位图缓存方法。要模拟 **opaqueBackground** 属性，可将影片剪辑绘制为非透明的 **BitmapData** 对象，这样不会强制运行时重新生成位图表面。

此技术适用于不随时间更改的内容。但是，如果文本字段的内容可以更改，请考虑使用其他策略。例如，假设一个文本字段以某个百分比不断进行更新来表示应用程序的加载量。如果该文本字段或其包含的显示对象缓存为位图，则每次内容更改时必须生成其表面。在此处无法使用手动位图缓存，因为显示对象内容不断更改。此常量更改将强制手动调用 **BitmapData.draw()** 方法以更新缓存的位图。

请记住，在 **Flash Player 8**（和 **AIR 1.0**）及更高版本中，无论 **Stage** 品质值如何，呈现设置为“可读性消除锯齿”的文本字段会始终保持完全消除锯齿状态。此方法占用的内存较少，但是需要更多的 **CPU** 处理，并且其呈现速度比位图缓存功能稍慢。

以下代码使用了此方法：

```
wait_mc.addEventListener( Event.ENTER_FRAME, movePosition );

// Switch to low quality
stage.quality = StageQuality.LOW;
var destX:int = stage.stageWidth >> 1;
var destY:int = stage.stageHeight >> 1;
var preloader:DisplayObject;
function movePosition ( e:Event ):void
{
    preloader = e.currentTarget as DisplayObject;

    preloader.x -= ( preloader.x - destX ) * .1;
    preloader.y -= ( preloader.y - destY ) * .1;

    if ( Math.abs ( preloader.y - destY ) < 1 )
    {
        // Switch back to high quality
        stage.quality = StageQuality.HIGH;
        preloader.removeEventListener ( Event.ENTER_FRAME, movePosition );
    }
}
```

不建议对处于运动状态的文本应用此选项（可读性消除锯齿）。当缩放文本时，此选项将导致文本尝试保持对齐，从而产生移动效果。但是，如果显示对象的内容不断更改并且您需要缩放文本，则可通过将品质设置为 **LOW** 以在移动应用程序中提高性能。当运动完成后，请将品质切换回 **HIGH**。

GPU

Flash Player 应用程序中的 GPU 呈现

Flash Player 10.1 一项重要的新功能是它可以使用 GPU 在移动设备上呈现图形内容。以前，只能通过 CPU 呈现图形。使用 GPU 可以优化滤镜、位图、视频和文本的呈现。请记住，GPU 呈现并不始终像软件呈现一样精确。使用硬件渲染器时，内容可能稍微显得“矮胖”。此外，Flash Player 10.1 还有一个限制，即阻止在屏幕上呈现 **Pixel Bender** 效果。在使用硬件加速时，这些效果可能呈现为黑色的正方形。

尽管 Flash Player 10 具有 GPU 加速功能，但该 GPU 不用于计算图形。它仅用于将所有图形发送到屏幕。在 Flash Player 10.1 中，GPU 也用于计算图形，从而大大加快了呈现速度。此外，这样做还减少了 CPU 工作负载，这对资源有限的设备（如移动设备）很有用。

在移动设备上运行内容时将自动设置 GPU 模式以实现最佳性能。尽管不必再将 `wmode` 设置为 `gpu` 以获得 GPU 呈现，但将 `wmode` 设置为 `opaque` 或 `transparent` 将禁用 GPU 加速。

注：台式机上的 Flash Player 仍然使用 CPU 来进行软件呈现。之所以使用软件呈现，是因为驱动程序在台式机上很有大区别，且驱动程序会强调呈现差异。台式机和某些移动设备之间还存在呈现差异。

移动设备 AIR 应用程序中的 GPU 呈现

通过在应用程序描述符中加入 `<renderMode>gpu</renderMode>`，在 AIR 应用程序中启用硬件图形加速。不能在运行时更改呈现模式。在台式计算机上，会忽略 `renderMode` 设置；当前不支持 GPU 图形加速。

GPU 呈现模式限制

当在 AIR 2.5 中使用 GPU 呈现模式时，存在下列限制：

- 如果 GPU 不能呈现某个对象，该对象就完全不显示。CPU 呈现没有回退功能。
- 不支持下列混合模式：图层、`alpha`、擦除、叠加、强光、变亮、变暗。
- 不支持滤镜。
- 不支持 **PixelBender**。
- 许多 GPU 单位的最大纹理大小为 **1024x1024**。在 **ActionScript** 中，它会在所有转换之后转变为显示对象的最大最终呈现大小。
- Adobe 不建议在播放视频的 AIR 应用程序中使用 GPU 呈现模式。
- 在 GPU 呈现模式中，文本字段并非总是在虚拟键盘打开时移动到可见位置。要确保用户输入文本时文本字段可见，请执行以下操作之一。当文本字段获得焦点时，将其放在屏幕上半部分或将其移到屏幕上半部分。
- GPU 呈现模式在某些设备上被禁用，因为在这些设备上该模式无法可靠地工作。有关最新信息，请参阅 AIR 开发人员发行说明。

GPU 呈现模式最佳实践

下列方法可以使 GPU 呈现加速：

- 限制舞台上可见项目的数量。每个项目都需要花费一些时间来呈现其周围的其他项目并与它们合成。当您不再需要显示显示对象时，可将其 `visible` 属性设置为 `false`。不要简单地将其移出舞台、将其隐藏在另一个对象后面或将其 `alpha` 属性设置为 `0`。如果彻底不再需要显示对象，使用 `removeChild()` 将其从舞台删除。
- 重复使用对象，而不是创建和销毁它们。
- 使位图的大小接近但小于 2^n 乘 2^m 位。尺寸不必正好为 2 的整数次幂，但是应该接近 2 的整数次幂，而不需要更大。例如， 31 乘 15 像素的图像比 33 乘 17 像素的图像的呈现速度更快。（ 31 和 15 刚刚小于 2 的整数次幂： 32 和 16 。）
- 如果可能，在调用 `Graphic.beginBitmapFill()` 方法时将 `repeat` 参数设置为 `false`。
- 不要过分夸张。将背景色作为背景。不要将较大形状进行叠放。每个必须绘制的像素都需要成本。
- 避免使用带有狭长突起、自相交的边缘或边缘周围存在许多细节的形状。呈现这些形状需要的时间比边缘平滑的显示对象长。
- 限制显示对象的大小。
- 针对不经常更新图形的线索对象启用 `cacheAsBitMap` 和 `cacheAsBitmapMatrix`。
- 避免使用 `ActionScript` 绘图 API（`Graphics` 类）创建图形。而是尽可能在创作时以静态方式创建这些对象。
- 先将位图资产缩放到最终大小，然后再将其导入。

手机版 AIR 2.0.3 中的 GPU 呈现模式

GPU 呈现在利用 `Packager for iPhone` 创建的手机版 AIR 应用程序中更受限制。GPU 只对位图、实体形状和设置了 `cacheAsBitmap` 属性的显示对象有效。此外，对于设置了 `cacheAsBitmap` 和 `cacheAsBitmapMatrix` 的对象，GPU 可有效地呈现旋转或缩放的对象。GPU 还用于其他显示对象的串联中，这样通常会导致呈现性能变差。

优化 GPU 呈现性能的提示

尽管 GPU 呈现可以大大改进 SWF 内容的性能，不过内容的设计也发挥着非常重要的作用。请注意，以前在软件呈现中作用良好的设置有时在 GPU 呈现下可能不能正常发挥作用。以下提示可以帮助您在不牺牲软件呈现性能的情况下，实现良好的 GPU 呈现性能。

注：支持硬件呈现的移动设备通常会从 Web 访问 SWF 内容。因此，最好的办法是在创建 SWF 内容时就考虑到这些提示，这样可以确保在所有的屏幕范围内实现最佳的体验。

- 避免在 HTML 嵌入参数中使用 `wmode=transparent` 或 `wmode=opaque`。这些模式可能会导致性能下降。还可能会导致在软件和硬件呈现中音频 - 视频同步的微小损失。另外，当这些模式生效时，许多平台不支持 GPU 呈现，会严重减弱性能。
- 请仅使用正常和 Alpha 混合模式。避免使用其他混合模式，特别是层混合模式。当在 GPU 下呈现时，并非所有混合模式均可以忠实地再现。
- 当 GPU 呈现矢量图形时，它会在绘制图形前将图形分为若干小三角形形状的网格。这个过程称为镶嵌。镶嵌会带来小的性能损失，当形状的复杂程度增加时，这个成本也会随之增加。若要尽量减少性能影响，请避免使用 `morph` 形状（GPU 会在每个帧上呈现镶嵌）。
- 避免自相交曲线、极细的曲线区域（例如非常细的月牙）以及沿着形状的边缘具有非常复杂的细节。这些形状比较复杂，GPU 很难将其镶嵌成三角形网格。要理解其中的原因，以这样两个矢量为例：一个 500×500 正方形，一个 100×10 月牙。GPU 可以轻松地呈现大的正方形，因为它正好可以分成两个三角形。然而，可能需要很多个三角形才能描述出月牙的曲线。因此，呈现形状更为复杂，尽管它牵涉的像素很少。
- 避免按比例进行大的改动，因为这种变动同样会导致 GPU 再次镶嵌图形。
- 尽量避免过度绘制。过度绘制会分层放置多个图形元素，各个图形之间会彼此遮蔽。使用软件呈现器，每个像素仅绘制一次。因此，对于软件呈现，应用程序不会导致性能下降，不论在那个像素位置有多少图形元素彼此覆盖。相反，硬件呈现器

会为每个元素绘制像素，不论其他元素是否遮蔽该区域。如果两个矩形彼此重叠，则硬件呈现器会绘制重叠区域两次，而软件呈现器仅绘制该区域一次。

因此，在使用软件呈现器的台式机上，您通常不会发现到过度绘制的影响。不过，许多重叠的形状可能会在使用 GPU 呈现的设备上带来不利的影响。最佳做法是从显示列表中删除对象，而不是隐藏这些对象。

- 避免使用大的填充矩形作为背景。而是改为设置舞台的背景色。
- 尽可能避免位图重复的默认位图填充模式。改为使用位图锁定模式以实现更好的性能。

异步操作



推荐使用操作的异步版本，而不是同步版本（如果适用）。

在代码通知同步操作运行时即立刻开始运行，而且代码会等待此操作完成后才会继续执行其他操作。因此，它们在帧循环的应用程序代码阶段运行。如果同步操作需要的时间太长，它将扩展帧循环的大小，这可能会导致显示冻结或不连贯。

当代码执行异步操作时，不必立即运行。代码和当前执行线程中的其他应用程序代码会继续执行。然后，运行时在尝试阻止呈现问题的同时会尽快执行此操作。在某些情况下，执行操作会在后台发生，且根本不会作为运行时帧循环的一部分运行。最后，当操作完成时，运行时会调度一个事件，且代码可以侦听该事件以执行进一步操作。

安排异步操作并将其拆分为几部分以避免出现呈现问题。因此，使响应应用程序使用操作的异步版本要容易得多。有关详细信息，请参阅第 2 页的“[感知性能与实际性能](#)”。

但是，异步运行操作会产生一些开销。异步操作的实际执行时间可能更长，尤其是那些短时间完成的操作。

在运行时中，许多操作本身就是同步操作或异步操作，因此您无法选择如何执行这些操作。然而，在 Adobe AIR 中，有三种类型的操作可供您选择是采用同步执行还是异步执行：

- **File 和 FileStream 类操作**

File 类的许多操作既可以同步执行，也可以异步执行。例如，复制或删除文件或目录以及列出目录的内容的方法都采用异步版本。这些方法包含添加到异步版本名称中的后缀“**Async**”。例如，要异步删除一个文件，请调用 **File.deleteFileAsync()** 方法，而不是 **File.deleteFile()** 方法。

当使用 **FileStream** 对象读取或写入文件时，打开 **FileStream** 对象的方式决定是否异步执行操作。使用 **FileStream.openAsync()** 方法执行异步操作。采用异步方式执行数据写入。数据读取分区块完成，因此每次只有部分数据可用。相比而言，在同步模式下，**FileStream** 对象读取整个文件，然后才继续执行代码。

- **本地 SQL 数据库操作**

使用本地 SQL 数据库时，通过 **SQLConnection** 对象执行的所有操作可以在同步模式下执行，也可以在异步模式下执行。要指定采用异步方式执行操作，请使用 **SQLConnection.openAsync()** 方法代替 **SQLConnection.open()** 方法打开到数据库的连接。当数据库操作采用异步方式运行时，将在后台执行。数据库引擎根本不在运行时帧循环中运行，因此数据库操作不太可能导致呈现问题。

有关提高本地 SQL 数据库性能的其他策略，请参阅第 74 页的“[SQL 数据库性能](#)”。

- **Pixel Bender 独立着色器**

ShaderJob 类允许您通过 **Pixel Bender** 着色器运行图像或数据集以及访问原始结果数据。默认情况下，当调用 **ShaderJob.start()** 方法时，着色器以异步方式执行。执行发生在后台，不使用运行时帧循环。要强制 **ShaderJob** 对象以异步方式执行（不建议这样做），请将值 **true** 传递给 **start()** 方法的第一个参数。

除用于异步运行代码的这些内置机制之外，您还可以构建自己的代码来异步运行而不是同步运行。如果您正在编写的代码用来执行可能会长期运行的任务，您可以构建自己的代码，以便分部分执行。将您的代码拆分为几部分允许运行时在代码执行块之间执行其呈现操作，从而尽可能减少呈现问题。

下面列出了几种用于拆分代码的技术。所有这些技术的主旨是编写的代码在任何时候仅执行其部分工作。可以跟踪代码执行的操作以及停止运行的位置。使用某种机制（例如 **Timer** 对象）反复检查工作是否仍在进行并分区块执行其他工作，直到完成此工作。

建立了几个模式可用于以此种方式构建代码以拆分工作。下列文章和代码库介绍了这些模式，并提供了代码以帮助您在应用程序中实施这些模式：

- [Asynchronous ActionScript Execution](#)（由 Trevor McCauley 撰写的文章，其中包括更多背景详细信息以及多个实施示例）
- [Parsing & Rendering Lots of Data in Flash Player](#)（由 Jesse Warden 撰写的文章，描述了“构建者模式”和“绿色线程”两种方法详细背景信息和示例）
- [Green Threads](#)（文章作者：Drew Cummins，其中使用示例源代码介绍了“绿色线程”技术）
- [greenthreads](#)（Charlie Hubbard 介绍的开放源代码库，用于在 ActionScript 中实施“绿色线程”。有关更多信息，请参阅 [greenthreads Quick Start](#)。）
- ActionScript 3 中的线程，网址为 http://www.adobe.com/go/learn_fp_as3_threads_cn（文章作者：Alex Harui，其中包括实施“伪线程”技术的示例）

透明窗口



在 AIR 台式机应用程序中，考虑使用不透明的矩形应用程序窗口而不是透明窗口。

要为 AIR 台式机应用程序的初始窗口使用不透明窗口，请在应用程序描述符 XML 文件中设置以下值：

```
<initialWindow>
    <transparent>false</transparent>
</initialWindow>
```

对于由应用程序代码创建的窗口，则创建一个 **NativeWindowInitOptions** 对象，将其 **transparent** 属性设置为 **false**（默认值）。创建 **NativeWindow** 对象时，将其传递给 **NativeWindow** 构造函数：

```
// NativeWindow: flash.display.NativeWindow class

var initOptions:NativeWindowInitOptions = new NativeWindowInitOptions();
initOptions.transparent = false;
var win:NativeWindow = new NativeWindow(initOptions);
```

对于 **Flex Window** 组件，确保在调用 **Window** 对象的 **open()** 方法前，该组件的透明属性已设置为 **false**（默认值）。

```
// Flex window component: spark.components.Window class

var win:Window = new Window();
win.transparent = false;
win.open();
```

透明窗口可能在该应用程序窗口后显示用户台式机或其他应用程序窗口的一部分。因此，运行时使用更多的资源来呈现透明窗口。矩形非透明窗口，无论它使用操作系统镶边还是自定义镶边，都不会有相同的呈现负荷。

仅当很有必要通过应用程序窗口显示非矩形或背景内容时才使用透明窗口。

矢量形状的平滑处理

💡 对形状进行平滑处理以改进呈现性能。

与呈现位图不同，呈现矢量内容（尤其是包含很多控制点的渐变和复杂路径）需要进行很多计算。设计者或开发人员应确保对形状进行充分优化。下图显示的是具有很多控制点的未简化路径：



未优化的路径

使用 **Flash Professional** 中的“平滑”工具可以删除多余的控制点。**Adobe® Illustrator®** 提供了一个等效工具，可以在“文档信息”面板中查看控制点和路径的总数。

进行平滑处理会删除多余的控制点，减小 **SWF** 文件的最终大小，并改进呈现性能。下图显示的是经过平滑处理的相同路径：



经过优化的路径

只要不过度简化路径，这种优化不会改变任何可见内容。但是，简化复杂路径可以大大提高最终应用程序的平均帧速率。

第 6 章：优化网络交互

针对网络交互的增强功能

Flash Player 10.1 和 AIR 2.5 针对所有平台上的网络优化引入了一组新功能，包括循环缓冲和智能搜索。

循环缓冲

在移动设备上加载媒体内容时，可能会遇到在台式机上几乎从不会发生的问题。例如，您很有可能遇到磁盘空间或内存用尽的情况。加载视频时，Flash Player 10.1 和 AIR 2.5 的台式机版本会下载整个 FLV 文件（或 MP4 文件）并将其缓存到硬盘驱动器上。然后运行时从该缓存文件播放视频。很少出现磁盘空间用尽的情况。如果发生这种情况，台式机运行时将停止播放视频。

移动设备可能更容易用尽磁盘空间。如果设备的磁盘空间用尽，运行时不会像在台式机运行时中那样停止播放。运行时将再次从缓存文件开头写入来开始重复使用缓存文件。用户可以继续观看视频。用户无法在已经重新写入的视频区域搜索，文件开头除外。默认情况下不会启动循环缓冲。循环缓冲可以在播放期间启动，如果影片大于磁盘空间或 RAM，还可以在播放开始时启动。运行时要求 RAM 至少为 4 MB、磁盘空间至少为 20 MB 才能使用循环缓冲。

注：如果设备有足够的磁盘空间，则运行时的移动设备版本与在台式机上的行为相同。请记住，如果设备没有磁盘或磁盘已满，RAM 中的缓冲区会用作回退。可在编译时设置缓存文件和 RAM 缓冲区的大小限制。一些 MP4 文件的结构要求下载整个文件才能开始播放。如果磁盘空间不足且无法播放 MP4 文件，运行时将检测这些文件并阻止下载。最好根本不要请求下载这些文件。

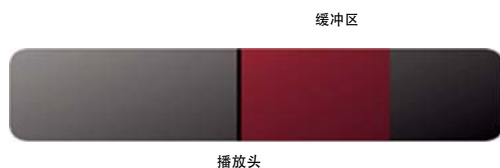
作为开发人员，请记住，只能在缓存流的边界内进行搜索。如果偏移量超出范围，`NetStream.seek()` 有时会失败，此时将调度 `NetStream.Seek.InvalidTime` 事件。

智能搜索

注：智能搜索功能要求使用 Adobe® Flash® Media Server 3.5.3。

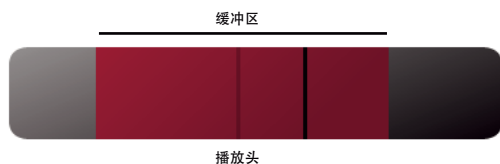
Flash Player 10.1 和 AIR 2.5 引入了一种称为智能搜索的新行为，可以在播放视频流时改进用户体验。如果用户在缓冲区边界内搜索目标，运行时将重复使用该缓冲区以提供即时搜索。在运行时的早期版本中，不会重用缓冲区。例如，如果用户从流服务器播放视频并且将缓冲时间设置为 20 秒 (`NetStream.bufferTime`)，该用户尝试提前 10 秒进行搜索，则运行时将丢弃所有缓冲数据，而不是重用这 10 秒加载的数据。此行为强制运行时更加频繁地从服务器请求新数据，导致在连接速度慢时播放性能欠佳。

下图演示了缓冲区在运行时的早期版本中的行为方式。`bufferTime` 属性指定提前预加载的秒数，以便断开连接时可以使用缓冲区而无需停止视频：

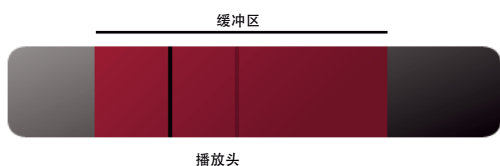


智能搜索功能之前的缓冲区行为

通过智能搜索功能，当用户对视频进行前进或后退播放时，运行时现在使用缓冲区来提供向前或向后搜索。下图演示了该新行为：



通过智能搜索功能进行向前搜索



通过智能搜索功能进行向后搜索

当用户向前或向后搜索时，智能搜索会重用缓冲区，因此播放体验更快更顺畅。这一新行为的优点之一是可以节省视频发布者的带宽。但是，如果搜索在缓冲区限制范围之外进行，则发生标准行为，运行时会从服务器请求新数据。

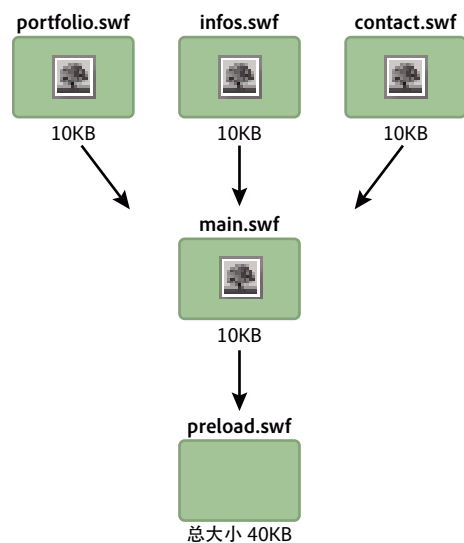
注：此行为不适用于渐进式视频下载。

要启用智能搜索，请将 `NetStream.inBufferSeek` 设置为 `true`。

外部内容

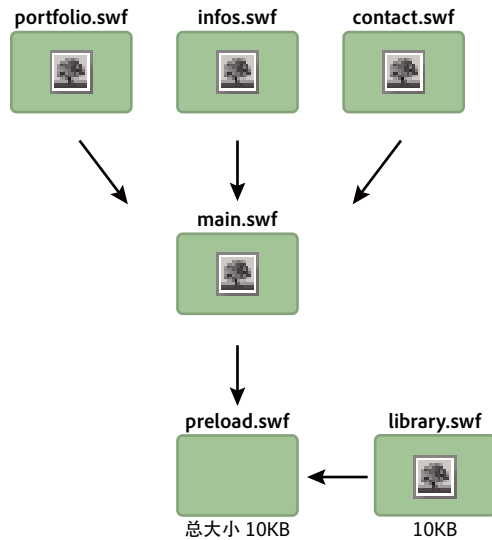
💡 将应用程序分为多个 SWF 文件。

移动设备可能具有受限的网络访问权限。要快速加载内容，可将应用程序分为多个 SWF 文件。尝试重用整个应用程序的代码逻辑和资源。例如，考虑将一个应用程序分为多个 SWF 文件，如下图所示：



分为多个 SWF 文件的应用程序

在此示例中，每个 SWF 文件都包含同一位图的其自己的副本。使用运行时共享库可以避免这种重复，如下图所示：



使用运行时共享库

借助此技术，可以加载运行时共享库，以使位图对其他 SWF 文件可用。ApplicationDomain 类存储已加载的所有类定义，并通过 `getDefinition()` 方法使它们在运行时可用。

运行时共享库也可以包含所有代码逻辑。整个应用程序可在运行时更新而无需重新编译。以下代码加载运行时共享库，并在运行时提取 SWF 文件中包含的定义。此技术可用于字体、位图、声音或任何 ActionScript 类：

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load(new URLRequest("library.swf") );
var classDefinition:String = "Logo";

function loadingComplete(e:Event):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the loaded SWF file application domain
    var appDomain:ApplicationDomain = objectLoaderInfo.applicationDomain;

    // Check whether the definition is available
    if ( appDomain.hasDefinition(classDefinition) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate logo
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " + classDefinition + " is not available.");
}
```

通过在正在加载的 SWF 文件的应用程序域中加载类定义，可以更容易获取定义：

```
// Create a Loader object
var loader:Loader = new Loader();

// Listen to the Event.COMPLETE event
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, loadingComplete );

// Load the SWF file
loader.load ( new URLRequest ("rsl.swf"), new LoaderContext ( false, ApplicationDomain.currentDomain) );
var classDefinition:String = "Logo";

function loadingComplete ( e:Event ):void
{
    var objectLoaderInfo:LoaderInfo = LoaderInfo ( e.target );

    // Get a reference to the current SWF file application domain
    var appDomain:ApplicationDomain = ApplicationDomain.currentDomain;

    // Check whether the definition is available
    if (appDomain.hasDefinition( classDefinition ) )
    {
        // Extract definition
        var importLogo:Class = Class ( appDomain.getDefinition(classDefinition) );

        // Instantiate it
        var instanceLogo:BitmapData = new importLogo(0,0);

        // Add it to the display list
        addChild ( new Bitmap ( instanceLogo ) );
    } else trace ("The class definition " + classDefinition + " is not available.");
}
```

现在，通过对当前应用程序域调用 `getDefinition()` 方法，可以使用加载的 SWF 文件中提供的类。您还可以通过调用 `getDefinitionByName()` 方法访问这些类。因为此技术仅加载一次字体和大型资源，所以节省了带宽。资源不会以任何其他 SWF 文件的形式导出。唯一的限制是应用程序必须经过测试并通过 `loader.swf` 文件运行。此文件首先加载资源，然后加载组成该应用程序的不同 SWF 文件。

输入 / 输出错误



针对 IO 错误提供事件处理函数和错误消息。

与连接到高速 Internet 的台式机的网络相比，移动设备上的网络可靠性比较低。在移动设备上访问外部内容有两个限制：可用性和速度。因此，请确保使用轻型资源，并为每个 `IO_ERROR` 事件添加处理函数以便向用户提供反馈。

例如，假设一个用户正在使用移动设备浏览您的网站，突然在两个地铁站之间失去了网络连接。失去连接时正在加载一个动态资源。在台式机上，您可以使用空事件侦听器阻止显示运行时错误，因为这种情况几乎从不会发生。然而，在移动设备上，您必须处理这种情况，而不仅仅是使用简单的空侦听器。

以下代码不响应 IO 错误。不要在其显示时使用它：

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener( Event.COMPLETE, onComplete );
addChild( loader );
loader.load( new URLRequest ( "asset.swf" ) );

function onComplete( e:Event ):void
{
    var loader:Loader = e.currentTarget.loader;
    loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
    loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}
```

建议处理此类失败并向用户提供错误消息。以下代码可以正确处理此失败：

```
var loader:Loader = new Loader();
loader.contentLoaderInfo.addEventListener ( Event.COMPLETE, onComplete );
loader.contentLoaderInfo.addEventListener ( IOErrorEvent.IO_ERROR, onIOError );
addChild ( loader );
loader.load ( new URLRequest ( "asset.swf" ) );

function onComplete ( e:Event ):void
{
    var loader:Loader = e.currentTarget.loader;
    loader.x = ( stage.stageWidth - e.currentTarget.width ) >> 1;
    loader.y = ( stage.stageHeight - e.currentTarget.height ) >> 1;
}

function onIOError ( e:IOErrorEvent ):void
{
    // Show a message explaining the situation and try to reload the asset.
    // If it fails again, ask the user to retry when the connection will be restored
}
```

最佳做法是谨记为用户提供一种再次加载内容的方式。此行为可以在 onIOError() 处理函数中实现。

Flash Remoting

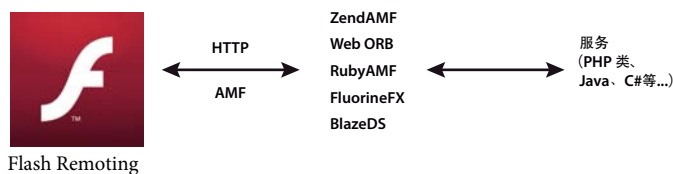


使用 Flash Remoting 和 AMF 实现优化的客户端 - 服务器数据通信。

您可以使用 XML 将远程内容加载到 SWF 文件中。但是，XML 是运行时加载和解析的纯文本。XML 最适用于加载内容有限的应用程序。如果您正在开发加载大量内容的应用程序，请考虑使用 Flash Remoting 技术和 Action Message Format (AMF)。

AMF 是用于在服务器和运行时之间共享数据的二进制格式。使用 AMF 将减少数据的大小并提高传输速度。由于 AMF 是运行时的一种本机格式，因此将 AMF 数据发送到运行时可避免序列化和反序列化在客户端上占用大量内存。远程网关可以处理这些任务。将 ActionScript 数据类型发送到服务器时，远程网关会在服务器端为您处理序列化。该网关还会向您发送相应的数据类型。此数据类型是在服务器上创建的一个类，该类用于公开一组从运行时调用的方法。Flash Remoting 网关包括 ZendAMF、FluorineFX、WebORB 和 BlazeDS，它是 Adobe 提供了一种正式的开放源 Java Flash Remoting 网关。

下图说明了 Flash Remoting 的概念：



以下示例使用 `NetConnection` 类连接到 Flash Remoting 网关：

```
// Create the NetConnection object
var connection:NetConnection = new NetConnection ();

// Connect to a Flash Remoting gateway
connection.connect ("http://www.yourserver.com/remoting-service/gateway.php");

// Asynchronous handlers for incoming data and errors
function success ( incomingData:* ):void
{
    trace( incomingData );
}

function error ( error:* ):void
{
    trace( "Error occurred" );
}

// Create an object that handles the mapping to success and error handlers
var serverResult:Responder = new Responder (success, error);

// Call the remote method
connection.call ("org.yourserver.HelloWorld.sayHello", serverResult, "Hello there ?");
```

连接到远程网关很简单。但是，使用 Adobe® Flex® SDK 中包含的 `RemoteObject` 类可使用 Flash Remoting 更简单。

注：外部 SWC 文件（如 Flex 框架中的某个此类文件）可在 Adobe® Flash® Professional 项目内使用。通过使用 SWC 文件，您可以使用 `RemoteObject` 类及其相关性，而无需使用其余 Flex SDK。如有必要，高级开发人员甚至可以通过原始 `Socket` 类直接与远程网关通信。

不必要的网络操作



加载资源后将本地缓存，而不是在每次需要时从网络进行加载。

如果应用程序加载诸如媒体或数据等资源，则通过将它们保存到本地设备来进行缓存。对于很少更改的资源，考虑间隔一段时间更新缓存。例如，应用程序可能每天检查一次图像文件是否有新版本或每两个小时检查一次是否有最新数据。

您可以根据资源的类型和性质通过多种方式缓存资源：

- 媒体资源（例如图像和视频）：使用 `File` 和 `FileStream` 类将文件保存到文件系统
- 单个数据值或小型数据集：使用 `SharedObject` 类将值另存为本地共享对象
- 较大数据集：将数据保存到本地数据库或对数据序列化并将其保存到文件中

对于缓存数据值，[open-source AS3CoreLib project](#) 包含一个 `ResourceCache` 类，可为您执行加载和缓存操作。

第 7 章：处理媒体

视频

有关移动设备上的视频性能优化的信息，请参阅 [Adobe Developer Connection](#) 网站上的[为移动设备优化 Web 内容](#)。

请特别注意以下小节：

- 在移动设备上播放视频
- 代码示例

这些小节包含有关为移动设备开发视频播放器的信息，例如：

- 视频编码准则
- 最佳做法
- 如何配置视频播放器的性能
- 参考视频播放器实现

StageVideo

使用 `StageVideo` 类充分利用硬件加速来呈现视频。

有关使用 `StageVideo` 对象的信息，请参阅 [ActionScript 3.0 开发人员指南](#)中的[使用 StageVideo 类来实现硬件加速呈现](#)。

音频

使用 `Flash Player 9.0.115.0` 和 `AIR 1.0` 启动时，运行时可以播放 AAC 文件（AAC Main、AAC LC 和 SBR）。使用 AAC 文件代替 mp3 文件可实现简单优化。在比特率相同的情况下，AAC 格式的文件比 MP3 格式的文件品质更好、文件大小更小。减少文件大小可以节省带宽，这对无法提供高速 Internet 连接的移动设备来说非常重要。

硬件音频解码

与视频解码类似，音频解码要求高 CPU 周期，并可通过利用设备上提供的硬件来进行优化。`Flash Player 10.1` 和 `AIR 2.5` 在解码 AAC 文件（LC、HE/SBR 配置文件）或 MP3 文件（不支持 PCM）时，可以检测和使用硬件音频驱动程序来提高性能。CPU 使用量显著减少，导致电池使用量减少并使 CPU 可用于其他操作。

注：使用 AAC 格式时，由于大多数设备上缺少硬件支持，所以设备上不支持 AAC Main 配置文件。

硬件音频解码对用户和开发人员公开。运行时开始播放音频流后，它将首先检查硬件，这与播放视频时相同。如果硬件驱动程序可用且支持该音频格式，将开始进行硬件音频解码。但是，即使可通过硬件处理传入的 AAC 或 MP3 流解码，有时硬件也无法处理所有效果。例如，基于硬件限制，有时硬件无法处理音频混合和重新采样。

第 8 章 : SQL 数据库性能

针对数据库性能的应用程序设计



不要在执行 `SQLStatement` 对象后更改其 `text` 属性。针对每个 SQL 语句使用一个 `SQLStatement` 实例并使用语句参数提供不同值。

在执行任何 SQL 语句之前，运行时准备（编译）该语句，确定在内部执行的步骤并执行语句。在以前未执行的 `SQLStatement` 实例上调用 `SQLStatement.execute()` 时，在执行该实例之前将自动准备语句。在对 `execute()` 方法的后续调用中，只要 `SQLStatement.text` 属性未更改，仍将准备语句。因此，它的执行速度更快。

为了充分发挥重用语句的优势，如果在语句执行之间值发生了更改，请使用语句参数自定义语句。（语句参数是使用 `SQLStatement.parameters` 关联数组属性指定的。）与更改 `SQLStatement` 实例的 `text` 属性不同，如果更改语句参数的值，则不要求运行时再次准备语句。

重用 `SQLStatement` 实例时，当准备 `SQLStatement` 实例后，应用程序必须存储对它的引用。要保存对该实例的引用，请将变量声明为类范围的变量而不是函数范围的变量。构建应用程序是使 `SQLStatement` 成为类范围变量的一种好方法，这样 SQL 语句将包装在单个类中。也可以将在组合中执行的一组语句包装在单个类中。（此技术称为使用 **Command** 设计模式。）通过将实例定义为类的成员变量，只要包装类的实例存在于应用程序中，它们就会一直存在。至少您可以在函数外只定义一个包含 `SQLStatement` 实例的变量，这样，该实例将保留于内存中。例如，将 `SQLStatement` 实例声明为 `ActionScript` 类中的成员变量，或声明为 `JavaScript` 文件中的非函数变量。然后可以设置语句的参数值，并在要实际运行查询时调用其 `execute()` 方法。



使用数据库索引可以提高执行数据比较和排序的速度。

创建列索引时，数据库将存储该列数据的副本。该副本按数字或字母顺序排序。通过排序，数据库可以快速与值匹配（例如，当使用等于运算符时）并使用 `ORDER BY` 子句对结果数据进行排序。

数据库索引会不断更新，因此该表中的数据更改操作（插入或更新）的速度有些降低。但是，提高数据检索速度很重要。为了实现这种性能平衡，请不要只对每个表中的每列进行索引。而应使用策略定义您的索引。使用下列准则可以计划索引策略：

- 对连接表、`WHERE` 子句或 `ORDER BY` 子句中使用的列编制索引
- 如果这些列经常一起使用，请在单个索引中对其编制索引
- 对于包含您检索的、按字母顺序排序的文本数据的列，请为索引指定 `COLLATE NOCASE` 排序规则



考虑在应用程序空闲时间期间预编译 SQL 语句。


第一次执行 SQL 语句时，速度较慢，因为要由数据库引擎准备（编译）SQL 文本。由于准备和执行语句可能要求过高，因此一种策略将预加载初始数据，然后在后台执行其他语句：

- 1 首先加载应用程序需要的数据。
- 2 当应用程序的初始启动操作完成后或在应用程序的其他“空闲”时间内，执行其他语句。

例如，假设应用程序根本不访问数据库来显示其初始屏幕。在这种情况下，请等待该屏幕显示，然后再打开数据库连接。最后，创建 `SQLStatement` 实例并执行可以执行的任何操作。

或者，假设应用程序在启动时就立即显示某些数据，如特定查询的结果。在该情况下，继续操作并执行该查询的 `SQLStatement` 实例。加载并显示初始数据后，为其他数据库操作创建 `SQLStatement` 实例，如有可能，则执行稍后需要的其他语句。

实际上，如果重用 `SQLStatement` 实例，则准备该语句需要的额外时间只是一次性成本。对总体性能可能不会有很大影响。

 在一个事务中组合多个 SQL 数据更改操作。

假设要执行涉及添加或更改数据的许多 SQL 语句（INSERT 或 UPDATE 语句）。通过在显式事务内执行所有语句，可以大大提高性能。如果不显式开始事务，则每个语句都运行在它自己的自动创建的事务中。每个事务（每个语句）完成执行后，运行时会将生成的数据写入磁盘上的数据库文件。

另一方面，应考虑显式创建事务并在该事务的上下文中执行语句时将发生的情况。运行时在内存中进行所有更改，然后在提交事务时将所有更改一次写入数据库文件。将数据写入磁盘通常是操作中最耗时的部分。因此，一次性写入磁盘而不是对每个 SQL 语句写入一次可以大大提高性能。

 使用 `SQLStatement` 类的 `execute()` 方法（带有 `prefetch` 参数）和 `next()` 方法处理各部分中的大量 SELECT 查询结果。

假设要执行一个检索大型结果集的 SQL 语句。然后，应用程序将循环处理每行数据。例如，它将设置数据的格式或从其中创建对象。处理这些数据需要很长时间，这可能导致呈现问题，例如屏幕冻结或没有响应。如在第 64 页的“[异步操作](#)”中所述，一种解决方案会将工作分为几个区块来执行。SQL 数据库 API 简化了拆分数据处理操作。

`SQLStatement` 类的 `execute()` 方法包含一个可选参数 `prefetch`（第一个参数）。如果您提供一个值，它指定在执行完成时数据库返回的最大结果行数：

```
dbStatement.addEventListener(SQLEvent.RESULT, resultHandler);  
dbStatement.execute(100); // 100 rows maximum returned in the first set
```

返回第一个结果数据集后，您可以调用 `next()` 方法继续执行该语句并检索另一组结果行。与 `execute()` 方法类似，`next()` 方法接受 `prefetch` 参数以指定要返回的最大行数：

```
// This method is called when the execute() or next() method completes  
function resultHandler(event:SQLEvent):void  
{  
    var result:SQLResult = dbStatement.getResult();  
    if (result != null)  
    {  
        var numRows:int = result.data.length;  
        for (var i:int = 0; i < numRows; i++)  
        {  
            // Process the result data  
        }  
  
        if (!result.complete)  
        {  
            dbStatement.next(100);  
        }  
    }  
}
```

您可以继续调用 `next()` 方法，直到加载完所有数据。如前面的列表表中所示，您可以确定何时加载完所有数据。检查每次 `execute()` 或 `next()` 方法完成时创建的 `SQLResult` 对象的 `complete` 属性。

注：使用 `prefetch` 参数和 `next()` 方法拆分结果数据的处理。不要使用此参数和方法将查询结果限制为其部分结果集。如果您只希望在语句的结果集中检索行的子集，请使用 SELECT 语句的 `LIMIT` 子句。如果结果集很大，仍可以使用 `prefetch` 参数和 `next()` 方法拆分结果的处理。

 考虑使用共用一个数据库的多个异步 `SQLConnection` 对象来同时执行多个语句。

当使用 `openAsync()` 方法将 `SQLConnection` 对象连接到数据库后，该对象在后台（而不是在主运行时执行线程中）运行。此外，每个 `SQLConnection` 都在自己的后台线程中运行。通过使用多个 `SQLConnection` 对象，您可以同时有效地运行多个 SQL 语句。


此方法也有潜在的缺点。最重要的是，每个其他 **SQLStatement** 对象都需要占用额外的内存。此外，同时执行还将导致处理器的任务增多，特别是对于只有一个 CPU 或 CPU 核心的计算机。由于存在这些问题，因此不建议对移动设备使用此方法。

还有一个问题是可能丢失重用 **SQLStatement** 对象的潜在优势，因为 **SQLStatement** 对象链接到单个 **SQLConnection** 对象。因此，如果 **SQLStatement** 对象的关联 **SQLConnection** 对象正在使用中，则无法重用该对象。


如果选择使用连接到一个数据库的多个 **SQLConnection** 对象，请记住每个对象在自己的事务中执行其语句。确保考虑到在更改数据的任何代码中的各个事务，例如添加、修改或删除数据。

Paul Robertson 创建了一个开放源代码库，可以帮助您结合使用多个 **SQLConnection** 对象的优点并在最大程度上减少潜在缺点。此库使用 **SQLConnection** 对象池并管理相关联的 **SQLStatement** 对象。这样，就可以确保重用 **SQLStatement** 对象，还可以使用多个 **SQLConnection** 对象同时执行多个语句。有关详细信息以及要下载该库，请访问 <http://probertson.com/projects/air-sqlite/>。

数据库文件优化


 避免数据库架构更改。

如有可能，在向数据库的表中添加数据后，避免更改数据库的架构（表结构）。通常，数据库文件是使用该文件开头的表定义构建的。打开到数据库的连接时，运行时将加载这些定义。向数据库表添加数据时，会将该数据添加到文件中，并放置在表定义数据之后。但是，如果进行架构更改，则新的表定义数据将与数据库文件中的表数据相混合。例如，向表中添加列或者添加新表会导致数据类型混合。如果表定义数据不全部位于数据库文件的开头，则打开到数据库的连接需要较长的时间。因为运行时从文件的不同部分读取表定义数据需要较长的时间，所以打开连接较慢。

 在架构更改后，使用 **SQLConnection.compact()** 方法优化数据库。

如果必须进行架构更改，则可以在完成更改后调用 **SQLConnection.compact()** 方法。此操作将重新结建数据库文件，以便表定义数据全部位于文件的开头。但是，**compact()** 操作可能需要大量的时间，尤其是数据库文件日益增长时。

不必要的数据库运行时处理

 在 SQL 语句中使用完全限定的表名称（包括数据库名称）。

始终在语句中显式指定数据库名称和每个表名称。（如果是主数据库，请使用“main”）。例如，以下代码包括一个显式数据库名称 **main**：

```
SELECT employeeId  
FROM main.employees
```

如果显式指定数据库名称，运行时就不必检查每个数据库才能找到匹配表。这样做还消除了使运行时选择错误数据库的可能性。即使 **SQLConnection** 仅连接到单个数据库，也要遵循此规则。在后台，**SQLConnection** 还连接到可通过 SQL 语句访问的临时数据库。

 在 SQL INSERT 和 SELECT 语句中使用显式列名称。

下列示例介绍如何使用显式列名称：


```
INSERT INTO main.employees (firstName, lastName, salary)
VALUES ("Bob", "Jones", 2000)
```

```
SELECT employeeId, lastName, firstName, salary
FROM main.employees
```

将上述示例与下列示例进行比较。避免使用这种类型的代码：

```
-- bad because column names aren't specified
INSERT INTO main.employees
VALUES ("Bob", "Jones", 2000)
```

```
-- bad because it uses a wildcard
SELECT *
FROM main.employees
```

如果没有显式列名称，运行时必须执行额外操作才能找到列名称。如果 **SELECT** 语句使用通配符（而不是显式列），将导致运行时检索额外数据。此额外数据需要额外处理并将创建不需要的额外对象实例。



除非将该表与其本身进行比较，否则避免在一个语句中多次连接同一个表。

随着 **SQL** 语句不断增大，您会无意中将数据库表多次连接到查询。通常，使用该表一次即可实现相同结果。如果在一个查询中使用一个或多个视图，很有可能多次连接同一表。例如，您可能将表连接到查询，也可能连接到一个包含该表中的数据的视图。这两种操作都可能导致多次连接。

有效的 SQL 语法



使用 **JOIN**（位于 **FROM** 子句中）可以在查询中包含表，而不是在 **WHERE** 子句中包含子查询。即使您只需要使用表数据进行过滤而不是生成结果集，该技巧也同样适用。

在 **FROM** 子句中连接多个表执行操作的效果比在 **WHERE** 子句中使用子查询的效果要好。



避免使用无法利用索引的 **SQL** 语句。这些语句包括使用子查询中的聚合函数、子查询中的 **UNION** 语句或包含 **UNION** 语句的 **ORDER BY** 子句。

索引可大大加快处理 **SELECT** 查询的速度。然而，某些 **SQL** 语法阻止数据库使用索引，从而强制它使用实际数据进行搜索或执行排序操作。



考虑避免使用 **LIKE** 运算符，特别是带有前导通配符的运算符，如 **LIKE('%XXXX%')** 所示。

由于 **LIKE** 操作支持使用通配符搜索，因此它的执行速度比使用完全匹配比较的速度要慢。具体而言，如果使用通配符开始字符串搜索，则数据库根本无法使用索引进行搜索，而必须搜索表中每行的完整文本。




考虑避免使用 **IN** 运算符。如果预先知道可能的值，可使用 **AND** 或 **OR** 写入 **IN** 操作以加快执行速度。

在下列两个语句中，第二个的执行速度稍快。之所以速度较快，是因为它使用了与 **OR** 组合在一起的简单等于表达式，而没有使用 **IN()** 或 **NOT IN()** 语句：


```
-- Slower
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary IN (2000, 2500)

-- Faster
SELECT lastName, firstName, salary
FROM main.employees
WHERE salary = 2000
      OR salary = 2500
```

 考虑使用 SQL 语句的替代形式以提高性能。

如上面的示例所示，编写 SQL 语句的方法也影响数据库性能。通常，有多种方法可以编写用于检索特定结果集的 SQL SELECT 语句。在某些情况下，一种方法的运行速度明显比其他方法要快。除了上面的建议，您还可以从有关 SQL 语言的专用资源中了解有关不同的 SQL 语句及其性能的更多信息。

SQL 语句性能

 直接比较替代 SQL 语句以确定哪个速度更快。

比较多个版本的 SQL 语句的性能的最佳方法是直接使用您的数据库和数据对它们进行测试。

下列开发工具提供了运行 SQL 语句时的执行次数。使用它们比较替代版本的语句的速度：

- [Run!](#) (Paul Robertson 提供的 AIR SQL 查询创作工具和测试工具)
- [Lita](#) (David Deraedt 提供的 SQLite 管理工具)

第 9 章：基准测试和部署

基准测试

有许多工具可用于基准测试应用程序。您可以使用由 Flash 社区成员开发的 Stats 类和 PerformanceTest 类。还可以使用 Adobe® Flash® Builder™ 中的探查器和 FlexPMD 工具。

Stats 类

要在运行时使用运行时的发行版（而不是外部工具）设置您的代码，您可以使用由 Flash 社区中的 doob 先生开发的 Stats 类。您可以在该网址下载 Stats 类：<https://github.com/mrdoob/Hi-ReS-Stats>。

Stats 类允许您跟踪以下内容：

- 每秒呈现的帧数（数字越高越好）。
- 呈现帧使用的毫秒数（数字越低越好）。
- 代码占用的内存量。如果代码在各个帧上占用的内存量增加，则您的应用程序可能存在内存泄露。调查可能存在的内存泄漏非常重要。
- 应用程序占用的最大内存量。

下载完成后，Stats 类可用于以下压缩代码：

```
import net.hires.debug.*;
addChild( new Stats() );
```

通过在 Adobe® Flash® Professional 或 Flash Builder 中使用条件编译，您可以启用 Stats 对象：

```
CONFIG::DEBUG
{
    import net.hires.debug.*;
    addChild( new Stats() );
}
```

通过切换 DEBUG 常量的值，您可以启用或禁用 Stats 对象的编译。可以使用这种方法替换您不想在应用程序中编译的任何代码逻辑。

PerformanceTest 类

为了设置 ActionScript 代码执行，Grant Skinner 开发了一种可集成到单元测试工作流程中的工具。将自定义类传递给 PerformanceTest 类，这将对您的代码执行一系列测试。通过 PerformanceTest 类，您可以轻松地地为不同的方法设置基准。可在以下地址下载 PerformanceTest 类：http://www.gskinner.com/blog/archives/2009/04/as3_performance.html。

Flash Builder 探查器

Flash Builder 附带一个探查器，该探查器允许您使用高级详细信息为代码设置基准。

注：使用调试版 Flash Player 访问该探查器，否则将出现错误消息。

概要分析器还可用于 Adobe Flash Professional 中生成的内容。要执行该操作，请将编译的 SWF 文件从 ActionScript 或 Flex 项目加载到 Flash Builder，然后可对其运行该探查器。有关概要分析器的详细信息，请参阅[使用 Flash Builder 4](#)中的“概要分析 Flex 应用程序”。

FlexPMD

Adobe Technical Services 发布了一款称为 FlexPMD 的工具, 该工具允许您审核 ActionScript 3.0 代码的品质。FlexPMD 是一种 ActionScript 工具, 类似于 JavaPMD。FlexPMD 通过审核 ActionScript 3.0 或 Flex 源目录来提高代码品质。它可以检测出品质欠佳的代码编写方法, 例如未使用的代码、过度复杂的代码、过长的代码和使用不正确的 Flex 组件生命周期。

FlexPMD 是一个 Adobe 开放源项目, 可在以下地址获得:

<http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD>。Eclipse 插件也可以在以下地址获得:

<http://opensource.adobe.com/wiki/display/flexpmd/FlexPMD+Eclipse+plugin>。

FlexPMD 简化了审核代码的过程, 而且更容易确保代码清晰且已经过优化。FlexPMD 的实际功能在于其可扩展性。作为开发人员, 您可以创建自己的规则集来审核任何代码。例如, 您可以创建一组规则, 用于检测大量使用的滤镜或任何其他您想要捕捉的品质欠佳的代码编写方法。

部署

在 Flash Builder 中导出应用程序的最终版本时, 确保导出该应用程序的发行版非常重要。导出发行版将删除 SWF 文件中包含的调试信息。删除调试信息将减小 SWF 文件的大小并有助于加快应用程序的运行速度。

要导出项目的发行版, 请使用 Flash Builder 中的“项目”面板和“导出发行版”选项。

注: 在 Flash Professional 中编译项目时, 不需要选择发行版或调试版。编译的 SWF 文件默认为发行版。