Autodesk®
Scaleform®

Scaleform 最佳□化指南

本文档提供了使用 Scaleform 4.0 和更新的版本创建资源的最佳优化指南

作者： Matthew Doyle
版本： 3.0
最后修订： 2011 年 4 月 25 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform 联系方式：

文档	Scaleform 最佳实践指南
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA

网站	www.scaleform.com
邮箱	info@scaleform.com
电话	(301) 446-3200
传真	(301) 446-3199

目录

1 概要

本文档包括了使用 Autodesk® Scaleform® 4.0 和更新的版本开发 Adobe® Flash®内容，并不断完善的一个最佳优化条目清单。这些优化方法主要针对于改进游戏中 Flash 运行的内存和性能；同时也可以用于其他的应用情况。文档中的内容创建章节主要面向美工和设计师，而 ActionScript™ (AS)章节面向技术设计师和工程师。人物 (HUD) 开发章节为 HUD 创建的开发场景的概要描述。我们鼓励设计师和工程师都能够在使用 Flash 和 Scaleform 3.0 开发 HUD 之前阅读这些章节。

本文档提供了不同来源的多种类型信息汇编，如客户支持请求、开发者论坛帖子、各种 web 上的 Flash 和 AS 资源。Scaleform 可以在多种不同的平台上使用（从手机到先进的游戏机和电脑）并集成了众多不同的引擎，从而，不幸的是没有一个“万能”解决方案。每个游戏和项目需要一个不同的解决方案以优化内存使用和性能（例如，从高端桌上电脑移植到手持终端时可能需要对代码进行调整）。

我们尽量提供这些最佳优化方法的相关性能数据；然而，这些数据与你的应用项目中的结果不尽相同，因为还需要考虑很多变量。我们建议在 UI 开发过程中所有的用户界面（UI）应该进行彻底得测试，可选方案应该在不同的应用场景中提前进行着重测试。

Flash 设计师和开发者必须直观的编写代码和构建应用结构，有利于自身以及工作在相同项目的其他成员，这在具有多个资源的 FLA 文件中尤为重要。注意，Flash Studio CS5 引入一种基于 XML 的内容格式，取代二进制 .FLA 格式。这对于版本控制和更改列表跟踪非常有用。

通常单个 Flash 项目有多个设计师或开发者协同工作，当每个成员都按照统一标准来设定指导方针以使用 Flash、组织 FLA 文件以及编写 AS 代码。文档的章节中略述了 AS 代码编写的最佳优化方法和使用 Flash 创作工具创建丰富媒体内容的最佳实践。在任何时候、设计师或开发者、独立工作或团队协作都可以使用最佳优化方法。以下列出了一些学习和使用最佳优化方法而获益的原因：

- 当工作在 Flash 或 AS 文档：
    - 采用一致和有效的做法，有助于加速工作流程。使用已制定的编码约定可以更快得进行开发，当进一步编辑时更益于理解和回忆文档的组织结构。并且，代码在一个大项目的架构中更易于移植和重用。
- 当共享 FLA 或 AS 文件：
    - 其他成员编辑的文档可以快速查找和理解 AS 脚本代码，从而修改代码，查找和编辑资源。
- 当工作于应用软件：
    - 多个创作者可以共同工作于一个应用软件之上，较少的冲突、更高的效率。项目或者现场管理员可以管理并构建复杂项目或应用，控制极少的冲突或冗余。

- 当学习或教导 Flash 和 AS：
    - 学习如何使用最优方法创建应用并遵循编码惯例以减少重复学习特殊方法的必要。如果学生不断得学习 Flash 优化以及更佳的代码结构方法，他们可以更快得学习编程语言，少走很多弯路。

开发者在阅读这些最优方法时必定会发现更多并发展他们自身的好习惯。考虑到下列作为与 Flash 协作时的方法指南；开发者可能选择采用其中一些或全部建议。开发者也可以修改建议以适合他们自身工作。本章中的许多指导方针将帮助开发者利用一种一致的方法用在 Flash 和 AS 代码编写上面。

Flash 中的优化方法分类如下所示：

- 通过优化图形提高动画性能以快速重绘。
- 通过编写代码提高运算性能以加快执行速度。

如果代码运行缓慢，意味着开发者需要减少使用范围或者采用其他更加有效的方法解决问题。开发者应该确定并消除瓶颈，例如，优化图形或通过使用 ActionScript 脚本代码减少工作量。

通常性能表现可以被感觉到。如果开发者试图在单个帧中执行过多任务，Flash 没有足够时间来渲染场景，用户可以感觉到速度降低。如果开发者将执行任务分成更小的任务块，Flash 能够在指定帧速率内刷新场景，不会感觉到速度降低。

1.1 常用用法

- 减少场景中对象数量。一次增加一个对象观察何时以及多少性能将被降低。
- 避免使用标准 UI 组件（在 Flash 的组件面板中可以获取）。这些组件设计为运行在桌面电脑之上，并没有进行优化以在 Scaleform 3.3 上运行。利用 Scaleform 通用精简界面工具包(CLIK™)组件代替。

1.2 设置事项

- 减少对象层级深度。例如，如果有些对象不单独移动或旋转，则不需要将其转换成组。
- 一个大的用户界面通常最好打散为单独的 Flash 文件。例如，在一个 MMORPG 交易场所为一个单独的 Flash 界面，而 HUD 生命值为另外一个单独 Flash 界面。在 AS 端，不同文件可以通过 AS2 和 AS3 中的各种 SWF 加载 API 进行加载。这些函数能够在应用程序中在任何给定时间内导入界面所需内容并且在不需要时释放这些内容。另外一个功能为通过 C++ API 函数导入和释放不同的 SWF 文件（创建不同的 GFx::Movie 实例）。

- Scaleform 3.3 支持多线程导入和播放。如果有额外的 CPU 运算内核，后台导入多个 Flash 文件不会对性能产生负面影响。但是，开发者必须注意何时从 CD 或 DVD 导入多个文件，因为光盘搜索速度缓慢可能会影响文件导入速度。
- 避免使用情景，情景通常为较大的 SWF 文件。
- 在游戏中的实际 UI 内存消耗量可能为以下范围：
    - 简单游戏 HUD 覆盖：400K–1.5M
    - 动画开始/菜单界面以及多个界面：800K–4M
    - 完全采用 ActionScript 脚本代码并具有资源（Pacman）的简单游戏：700K–1.5M
    - 频繁变化的矢量动画资源：2M–10M+

2 创建内容的内存和性能

当开发 Flash 内容时，很多考虑和优化需要遵循和实现以达到最佳性能。

2.1 绘图原型

绘画单元 (DP) 是 GFx 创建的网格对象，用来将 Flash 元素渲染到显示屏，例如，一个形状。在支持批处理（或实例）渲染的平台上，GFx 尝试将具有兼容属性的相同和相邻层上的形状组合到单个 DP 中。每个 DP 单独渲染，因而招致性能成本。每个绘图原型可以单独渲染，导致了大量消耗，在场景中使用的 DP 数量的增加导致显示性能呈线性降低；因此最好保持 DP 使用量越少越好。绘图原型的数量可以通过 Scaleform Player HUD 来判断，只需要点击快捷键(F2)，将弹出 AMP HUD 摘要界面。界面显示三角形计数、DP、内存使用和其他优化信息。

以下为一些可以帮助降低绘图原型数量的方法：

1. DP 只能包含具有相同属性的项目（具有不同纯色的形状除外）。例如，具有不同纹理或混合模式的项目永远不能合并到 DP 中。
2. 不同层的重叠对象不能合并到一个 DP 中，即使它们具有相同的属性。
3. 如果同时在一个形状中使用多个梯度填充，则这些梯度填充会增加 DP 的数量。
4. 具有纯色填充和没有笔画的矢量图形非常廉价。
5. 空的电影剪辑不需要 DP，不过，其中含有对象的电影剪辑则必须具有那些对象所确定的 DP 个数。

可以通过在 AMP 中启用该选项或按 (Ctrl+J) 来在 Scaleform 播放器中可视化批处理组和/或实例组。在此模式下，具有完全相同的颜色的项目以单个 DP 进行渲染。

2.2 动画剪辑

1. 与其隐藏动画剪辑，不如不再使用时从时间轴彻底删除，否则，在前进播放时占用处理时间。
2. 避免过多的动画剪辑嵌套，因为嵌套将影响性能。
3. 如果需要隐藏一个动画剪辑，则使用`_visible=false` ，比使用`_alpha=0`(in AS2) and `visible=false` rather than `alpha=0` (in AS3)更有优势。确保调用“stop()”函数停止隐藏动画剪辑中的动画。否则，隐藏的动画仍然发生动作并影响性能。

2.3 艺术品

2.3.1 位图与矢量图对比

Flash 内容能够通过矢量技术以及图像进行创建，Scaleform 可以无缝渲染矢量和位图。但是，每种类型都有其优势和劣势。何时使用矢量或位图的判断界限并不是十分清晰，通常由多个因素决定。本章将讨论矢量图和位图之间的差异以帮助创作者进行选择。

矢量图在进行缩放时可以维持平滑的图形，而位图在缩放时按矩形缩放或者按像素拉伸。与位图不同，矢量图需要更多的运算处理。尽管简单的纯色图形处理速度接近位图，具有大量三角形、图形和填充的复杂矢量图渲染开销较大。因此，大量使用矢量图有时将降低整体应用性能。

有些应用中最好使用位图，因为位图不需要太多处理时间来渲染矢量图，但是，相对于矢量图位图显著增加内存消耗。

2.3.2 矢量图

矢量图比其他格式图像更加简洁，因为矢量图使用数学运算（点、曲线、填充）在运行中实时渲染图像而不是如位图一样采用图像原型（像素）。但是，将矢量数据转换成最终的现实图像需要消耗时间，且在外观发生变化或图形缩放时必须重新执行。如果动画剪辑包含复杂图形且在每个帧都发生变化，动画将运行缓慢。

以下为帮助有效渲染矢量图的一些方法：

- 对负责矢量图转换为位图进行试验并测试性能影响。
- 在使用混合透明时注意一下几点
    - 纯色块比混合透明块需要更少运算开销，因为纯色块处理能够使用更多有效算法。
    - 避免使用透明化效果（alpha）。Flash 必须检查透明化形状之下的每个像素，显著降低了渲染速度。需要隐藏一个剪辑，设置_visible(AS2) or visible (AS3) 属性为 false，而不要将_alpha (AS2) or alpha (AS3)属性设置为 0。在_alpha/alpha 属性设置为 100 时图像渲染速度更快。设置动画剪辑的时间轴到一个空的帧（这样动画剪辑无可显示的内容）通常可以加快速度。有时 Flash 仍然试图渲染隐藏的剪辑；将_x 和_y 坐标移出到场景的可视范围也可以隐藏剪辑，也可以将_visible/visible 属性设置为 false，Flash 不再绘制该图像。
- 优化矢量图

    - 使用矢量图可以尽可能得简化图形消除冗余点。这将减少计算每个矢量图的播放器计算量。
    - 使用矢量原型包括曲线、矩形和直线。
    - Flash 绘图性能与每个帧绘制的点数有关。通过 Modify -> Shape submenu 来优化图形，然后选择 Smooth、Straighten 或 Optimize（根据讨论的图像决定）以减少需要绘制的点的数量。这将减少 Scaleform 矢量镶嵌码产生的网格数据。
- 拐角比曲线高效
    - 避免使用大量曲线和点的复杂矢量。
    - 拐角渲染数学算法比曲线简单，尽可能得坚持用扁平边缘，特别是非常小的矢量图形，曲线可以模仿该方法。
- 尽量少用梯度颜色填充和梯度块
- 避免图形轮廓描绘（打散）
    - 如果可以，任何时候都不要使用矢量图形块，因为这样增加渲染线条数量。
    - 矢量图像周边轮廓描绘影响性能。
    - 尽管填充只是一个外部形状渲染，轮廓绘制需要进行内部和外部渲染。这比在填充上绘制直线多两倍工作量。
- 减少 Flash 绘图 API 函数的使用，如果不必要的使用将导致性能超标。如果需要，可以使用绘图 API 函数绘制一次动画剪辑。渲染这样一个定制动画剪辑将不会产生性能影响。
- 限制蒙板的使用。蒙板下面的像素将仍然消耗渲染时间并对性能产生负面影响，甚至不进行任何绘制也是如此。多蒙板影响与使用的蒙板数相关联。注意在很多情况下设计师需使用蒙板达到的可视效果未必需要使用蒙板才能实现。在有些情况，通常使用蒙板从位图剪切一个图形，直接在 Flash Studio 中在图像上填充一个图形可以实现同样的效果。这为 Scaleform 的 EdgeAA 反锯齿效果提供了额外优势。
- 将多个非重叠对象转换为拥有相同的属性，尽可能避免生成额外的绘画单元（如上所述）。
- 在创建一个图形后，可以进行转换、旋转和混合，不需要额外内存开销。但是，引入新的较大图形或进行明显缩放将从镶嵌方格中消耗更多内存。
- 使 EdgeAA 有效，纯色图形比多个色阶/图像渲染速度更快。在一个图形中档链接色阶/图像时建议需要慎重处理，因为这将导致绘图原型快速增加。

2.3.3 位图

创建优化并改进的动画或图形第一步为在创建前规划工程，指定所要创建的对象的文件尺寸、内存使用和动画长度的目标数，完整测试整个开发过程确保全程跟踪。

除了前面描述的绘图原型，还有一个影响渲染性能的重要因素为整个表面区域的绘制。每次一个可视图形或图像放置到场景中，需要进行渲染，甚至在被其他图形覆盖时也是如此，消耗显卡的填充率。尽管当前显卡比 Flash 软件速度高出一个数量级，屏幕上较大的覆盖透明混合对象仍然能够极大降低性能，特别是应用在低级终端和老的硬件当中。鉴于此原因，简化重叠图形和位图将尤为重要，并明确隐藏模糊或剪切对象。

当隐藏对象，最好在 SWF 文件中将动画剪辑实例的`_visible/visible` 属性设置为 `false`，替代将 `_alpha (in AS2) or alpha (in AS3)`值设为 0。尽管 Scaleform 在`_alpha/alpha` 值为 0 时不会绘制对象，它们的子对象动画和 ActionScript 仍然消耗 CPU 处理开销。如果将实例可视化属性设置为 false，有可能可以减少 CPU 循环时间以及节省内存，可以使 SWF 文件更加平滑的动画并为应用程序提供更优的整体性能。代替释放和重复导入资源，设置`_visible/visible` 属性设置为 `false`，这可以大大减轻处理器负荷。以下为帮助进行高效位图渲染的一些指导方法：

- 创建所有纹理/位图的宽度和高度值使用 2 的整数次方。例如位图尺寸为 16x32、 256x128、1024x1024 和 512x32。
- 不要使用目标硬件不支持的经过压缩的图像（例如，JPEG 图像经过压缩，但在硬件中不受任何平台支持）。进行 JPEG 压缩在导入文件时需要解压缩时间。
- 使用具有纹理压缩开关的 gfxexport 工具来创建您的目标平台的硬件所支持的压缩图像（例如，DXT 压缩就受许多平台支持）。最终的 SWF 转换成为 GFX 格式，以通过纹理压缩来减少位图内存要求。与未压缩纹理相比，压缩纹理可提供巨大位图内存节约。许多压缩的纹理格式都使用有损压缩（包括 DXT），因此，要确保结果产生的位图的质量令人满意。使用 gfxexport 中的选项–qp 或 –qh 以获得最高等级的 DDS 纹理质量（注意这些选项可能需要很长时间来处理位图图像）。
- 尽量少用位图和/或小尺寸位图。
- 如果一个位图用来显示一个较大而简单的图形，使用矢量图重新创建该位图。这将产生具有 Edge AA 的更高质量，且可以节省内存。
- 根据需要可以用 ActionScript 脚本导入和释放较大位图。
- SWF/GFX 文件尺寸不应该用来判断其内存占用量。即使一个 SWF 文件很小，加载时它也会使用大量内存。例如，如果一个 SWF 文件包含一个 1024 x 1024 嵌入式 JPEG 图像，则图像解压后，运行时图像会使用 4 MB 内存。
- 追踪在 UI 中使用的图像文件数量和尺寸非常重要。计算所有的图像尺寸，加起来然后乘以 4（通常每个像素 需要四个字节）。注意 gfxexport 工具应该设置为–i DDS 选项以使用纹理压缩减少图像内存消耗。使用 AMP 检查位图内存消耗。
- 总之，在大多数情况下，位图来描述任何图形速度更快；但是，矢量图外观效果更好。最终决定应该参考系统填充速率、转换影音性能和 CPU 速率。最终，最实际的方法为在目标系统测试性能。

- 创建一个 gradient.swf 主文件，只包含色阶纹理并根据需要导入到 SWF 文件。使用 gfxexport 工具的-d0 开关导出 gradients.swf。该开关禁止压缩，应用到所有的 SWF 文件中的纹理当中。需要确保所有的纹理在该文件中，利用色阶，不受边界限制。
- 尽可能避免位图的透明通道。
- 在图像处理应用程序中优化位图，如在 Adobe Photoshop®，而不再 Flash 中。
- 如果位图需要透明化处理则使用 PNG 图像，视图减少表面区域绘制数量。
- 避免较大位图的重叠，那样将影响填充性能。
- 导入的位图图像尺寸应符合应用大小；不要导入图像然后在 Flash 中将其缩小，这样浪费文件大小以及运行内存。

2.3.4 动画

当添加动画到一个应用程序中，考虑 FLA 文件的帧速率。将影响最终 SWF 文件的性能。设置帧速率过高将导致性能问题，特别是当使用很多资源或 AS 脚本代码来创建动画，因为这些资源都与帧速率紧密相关。

然而，帧速率设置同样也影响动画播放的平滑度。例如，一个动画设置为每秒 12 帧（FPS），每秒在时间轴上播放 12 帧。如果文档帧速率设置为 24 FPS，则动画将比 12 FPS 表现得更加平滑。但是，一个动画在 24 FPS 帧速率下比 12 FPS 快两倍，所以同样数量帧的动画持续时间（单位为秒）为后者一半。因此，为了使 5 秒动画使用更高帧速率，需要添加更多的帧，这样增加了整个文件的大小。

注释：当使用一个 `onEnterFrame(in AS2)` or `Event.ENTER_FRAME (in AS3)` 事件句柄来创建脚本动画时，动画运行在文档的帧速率，类似于在时间轴上创建一个运行动作。
`onEnterFrame/Event.ENTER_FRAME` 事件句柄可以用 `setInterval` 进行替换。函数调用由特定的时间间隔决定，以毫秒为单位，不依赖于帧速率。如 `onEnterFrame/Event.ENTER_FRAME`，用 `setInterval` 调用函数越频繁，动画资源消耗越多。

尽可能使用最低的帧速率可以在运行时渲染一个平滑动画。这将降低处理器性能的影响程度。尽量不要使用超过 30 到 40 FPS 的帧速率；超过这个范围的帧速率将增加 CPU 开销且不能明显改进动画平滑度。在多数情况下，Flash UI 可以安全得设置为游戏目标帧速率的一半。

以下为有助于有效设计和创建动画的一些指导方法：

- 场景中的对象数量和移动速度影响整体性能。
- 如果在场景中拥有大量动画剪辑，且需要快速切换 on/off，则可以使用`_visible/visible = true/false` 设置来控制可视属性，而不需要使用附加/删除动画剪辑操作。
- 密切关注 tween 动画的使用

    - 避免过多条目同时使用 tween，减少 tween 数量并且/或者顺序动画，这样一个动画开始，另外一个动画结束。
    - 尽可能使用时间轴动画 tween 代替标准 Flash Tween 类因为对性能影响较小。
    - Scaleform 推荐使用 CLIK Tween 类（gfx.motion.Tween in AS2 or scaleform.clik.motion.Tween in AS3）代替标准 Flash Tween 类，因为前者更小、更快、更简洁。
- 保持较低帧速率，帧速率的不同往往不被注意到，帧速率越高，动画越平滑，但是增加对性能的影响。一个游戏运行在 60 帧每秒的帧速率不需要将 Flash 文件设置为 60 FPS。Flash 帧速率应该设置为满足必要视觉效果的最低帧速率。
- 透明和色阶消耗处理器资源应尽量少用。
- 精心设计焦点区域使其为动画，在屏幕中其它区域减少动画和特效。
- 在转换过程中终止背景动画（例如，微妙的背景效果）。
- 测试添加/删除动画元素衡量性能影响。
- 巧妙使用 tween 动画。在较慢的硬件上可以创建一个“外观滞后”效果。
- 避免使用形变动画，如将曲线变成矩形，因为形变动画需要消耗大量 CPU 资源。形变动画（morphing）消耗大量 CPU 资源是因为形变动画在每帧都需要重新计算；开销由图形的复杂程度决定（边缘数、曲线和交叉点）。在某些情景中能发挥作用，但是需要确保开销在可以接受的范围内；消耗为四三角形动画可以被接受。实质上，需要明白性能/内存的相互关系。显示常规图形导致镶嵌格子缓存中保存图形，因此能够在将来的播放帧中有效得显示出来。使用形变动画，改变了其相互关系，因为图形的任何变化为原来的栅格释放新的栅格建立。
- 最有效的动画为转换和旋转。最好避免使用缩放动画，因为缩放动画导致镶嵌栅格（具有明显的性能影响）产生的栅格占用更多内存。

2.3.5 文本和字体

- 文本轮廓字体尺寸应该小于字体缓存管理器 SlotHeight 或者 gfxexport（默认为 48 个像素）计划使用的尺寸。如果使用了一个更大的字体，则将使用矢量从而产生很多 DP 降低速度（每个矢量轮廓产生一个 DP）。
- 尽可能关闭文本区域边框和背景，因为这将节省一个绘图原型。
- 每帧中更新一个文本域内容严重影响性能，这可以简单得避免。改为，只当其内容确实改变或者在尽可能低的帧速率下改变文本域的值。例如，当更新一个显示时钟的计时器，不需要在 30 FPS 下每帧都进行更新。记录原来的值并在值发生改变时重新分配文本域的值。
- 不要使用链接到文本域的变量("TextField.variable"属性)。由于文本域在每帧都要将重复访问并比较变量，因此影响性能。
- 重新分配“htmlText”属性减少更新文本。解析 HTML 开销非常高昂。

- 使用 gfxexport 的`-fc`, `-fcl`, `-fcm` 选项来使字体更加紧凑节省字形轮廓占用的内存（特别在内嵌亚洲字体中）。见“字体概述”文档获得更多详细信息。
- 只插入需要的字形或者使用字体库机制用于本地化（也可以见“字体概述”文档获得更多详细信息）
- 使用最少数量必须的 TextField 对象，尽可能将多个条目合并成一个。单个文本域通常可以用一个 DP 进行渲染，甚至在使用不同颜色和字体时也是如此。
- 避免缩放文本域或使用较大字体；在超过一定尺寸的大小后，文本域将切换到矢量字形每个矢量字形需要一个绘图原型。如果需要剪切块（只有部分矢量沦落可视）则需要用到蒙板。蒙板速度缓慢且需要额外的绘图原型，字体裁剪的光栅不需要蒙板。
- 确保字形缓存有足够大空间存放所有（或多数）使用的字形。如果缓存尺寸不够，则某些字形可能会消失，或者频繁栅格化字形严重影响性能。
- 使用文本特效如模糊、阴影或滤镜需要使用更多字体缓存，同时也影响性能。尽可能少使用文本滤镜。
- 尽可能使用 DrawText API 函数代替单独动画。DrawText API 允许开发者通过编程方式用 C++语言绘制文本，使用与 Flash 字体和 Scaleform 用户界面中相同的文本。在屏幕移动物体上渲染指示牌名称或者为一个雷达上的条目标签，如果游戏不能使用 Flash UI 处理，则用 C++将更加有效。更多详细信息请见文档“DrawText API 参考”。

3 ActionScript 优化

ActionScript 不编译成本地机器码；而是转换成字节码，比解释语言运行速度要快但是没有编译的本地代码速度快。尽管 AS 脚本代码速度较慢，但是在大多数多媒体展现中，资源如图像、声音、视频 – 而不是代码 – 作为性能的瓶颈。

很多优化技术并不是专门针对 AS 脚本代码，只是众所周知的技术可以用在无优化编辑器且用任何语言编写的代码当中。例如，在循环中的条目在每次循环不发生改变时放到循环体的外部，可以使循环运行得更快。

3.1 通用 ActionsScript2 指南

以下优化可以增加 AS 执行速度。

- AS 脚本代码使用越少，文件性能越高。完成既定任务尽可能少的使用代码。AS 主要用户交互，而适合用来创建一个图形元素。如果你的代码包含了大量 `attachMovie` 调用，重新考虑 FLA 文件如何构造。
- 保持 AS 尽量简洁。
- 尽量少用脚本动画；时间轴动画通常具有更高性能。
- 避免使用大量字符串。
- 避免过多循环动画剪辑使用一个“if”语句以避免终止。
- 避免使用 `on()` 或 `onClipEvent()`事件句柄，使用 `onEnterFrame`、`onPress` 来代替。
- 在帧上最小化主要的 AS 逻辑代码，而在函数内部放置较大代码块。用 Flash AS 编译器编译生产的代码相对于在帧上直接运行或用老式句柄(`onClipEvent, on`)运行，速度更加快。但是在帧上使用简单逻辑代码也是可以接受的，如时间轴控制(`gotoAndPlay、play、stop` 等)以及其他非关键逻辑。
- 避免在包含多个动画的长时间轴电影剪接中使用“远距”`gotoAndPlay/gotoAndStop`。
    - 如果是前向 `gotoAndPlay/gotoAndStop`，目标帧距离当前帧越远，`gotoAndPlay/gotoAndStop` □□□控件就越□。因而，最贵的前向 `gotoAndPlay/gotoAndStop` 是从第一个帧到最后一个帧的部分。
    - 如果是后向 `gotoAndPlay/gotoAndStop`，目标帧距离时间轴的开头越远，时间轴控件就越贵。因而，最贵的后向 `gotoAndPlay/gotoAndStop` 是从最后一个帧到倒数第二个帧的部分。

- 使用短时间轴电影剪辑。gotoAndPlay/gotoAndStop 的成本在很大程度上取决于关键帧的数量和时间轴动画的复杂程度。因此，如果计划通过调用 gotoAndPlay/gotoAndStop 进行导航，就不要创建复杂的长时间轴。或者，将电影剪辑时间轴拆分成具有较短时间轴和较少 gotoAndPlay/gotoAndStop 调用的若干个独立的电影剪辑。
- 如果同时更新多个对象，则需要开发一个系统其中对象可以按组更新。在用 C++时可以使用 GFx::Movie::SetVariableArray 调用从 C++传递大量数据到 AS。该调用基于上载阵列可以通过单个调用更新多个对象。将多个调用组合成组通常比单独调用速度要快好几倍。
- 不要在一个帧内试图执行太多任务，Scaleform 可能没有足够时间渲染场景，用户会感到速度减慢。而需将大量任务分散为小的块，使 Scaleform 在给定的帧速率内进行刷新不至于感到速度缓慢。
- 不要过度使用对象类型
    - 数据类型注释须精确，因为这使编译器类型检查定位 bug，只有在没有其他方法时才使用对象类型描述。
- 避免使用 eval()函数或阵列访问操作，通常，设置一次本地索引将更加可取且有效。
- 在循环使用前将 Array.length 分配给一个变量作为循环条件，而不是用 myArr.length，例如：

  使用下列代码

```
var fontArr:Array = TextField.getFontList();
var arrayLen:Number = fontArr.length;
for (var i:Number = 0; i < arrayLen; i++) {
    trace(fontArr[i]);
}
```

  代替：

```
var fontArr:Array = TextField.getFontList();
for (var i:Number = 0; i < fontArr.length; i++) {
    trace(fontArr[i]);
}
```

- 精确管理实践。保持事件监听器阵列紧凑，使用条件判断调用前监听器是否存在（不为 null）。
- 在释放对象索引之前须调用 removeListener()从对象删除监听器。
- 最大限度地减小包名的级数以便于缩短启动时间。启动时，AS VM 必须创建对象链，每级一个对象。而且，在创建每级对象之前，AS 编译器添加一个 "if" 条件句，以检查是否已经创建那级对象。因此，对于包 "com.xxx.yyy.aaa.bbb"，VM 将创建对象 "com"、"xxx"、"yyy"、"aaa"、"bbb"，并且在每次创建之前，将会有一个检查对象是否存在的 "if" 运算代码。访问此类深度嵌套的对象/类/函数的速度也很慢，因为解析名称时需要分析每个级（解析 "com"，然后解析 "xxx"，然后解析 "xxx" 内的 "yyy"，以此类推）。为了避免此额外开销，有些 Flash 开发人员使用预处理软件在编译 SWF 之前缩短到达单级唯一标识符（如 c58923409876.functionName()）的路径。

- 如果一个应用程序包括多个 SWF 文件且使用相同的 AS 类，除了那些编译时选自 SWF 文件的类，这可以帮助减少运行内存需求。
- 如果时间轴上的关键帧中的 AS 脚本代码需要很长执行时间，可以考虑将代码分割到多个关键帧中去。
- 当发布最终 SWF 文件时，从代码中删除 trace()语句。完成此操作，在 Publish Settings 对话框中的 Flash 标签中选择 Omit Trace Actions 复选框，然后加以注释或直接删除。这是用来禁止实时调试跟踪语句的有效方法。
- 继承增加了方法调用的数量和内存的使用：一个类包括了所有需要的函数执行起来比从上级类继承功能的类执行起来更加高效。因此，在类的扩展和代码效率上需要有一个设计权衡。
- 当一个 SWF 文件导入另外一个包含一个自定义 AS 类（例如：`foo.bar.CustomClass` ）然后释放 SWF 文件，该类的定义仍然在内存中。为了节省内存，在释放 SWF 文件时需要删除自定义类。使用 `delete` 语句并制定类的完整名称，如下例所示：

```
delete foo.bar.CustomCLass
```

- 不是所有代码都在每个帧中都需要运行，对于非 100%有严格时间要求的项可以使用动态调度方法（每个帧中部分代码为可选）。
- 试图尽量少用 `onEnterFrames`。
- 预先计算数据表而不使用数据函数
    - 如果处理过多数学运算，可以将值预先计算出来并保存到一个变量数据（假设）中。从数据表获取这些值比 Scaleform 来处理更加有效。

3.1.1 循环

- 重点优化循环中的任何重复动作。
- 限制使用的循环数量和每个循环包含的代码量。
- 在不再需要时立即停止基于帧的循环。
- 避免多次调用一个循环内部的函数。
    - 最好在循环体内包含少量函数。

3.1.2 函数

- 尽可能避免函数深沉嵌套。
- 不要在函数内使用 `with` 语句。该操作将关闭优化功能。

3.1.3 变量/属性

- 避免参考不存在的变量、对象或函数。
- 尽可能使用“var”关键字。在函数内部使用“var”关键字特别重要，因为 ActionScript 编辑器优化了使用内部寄存器对本地边路按的访问，直接使用访问索引而不将数据放到哈希表中根据名字查找。
- 在本地变量足够情况下不要使用类变量或全局变量。
- 限制全局变量使用，因为在动画剪辑定义然后删除时不会进行垃圾回收。
- 删除变量或在不需要后设置为 null。这个操作用于垃圾回收，删除变量可以优化运行时内存使用，因为不需要的资源从 SWF 文件按中删除。最好是直接删除变量，比设置为 null 更加有效。
- 总是尽量直接访问属性，比使用 AS 获取和设置有效，后者需要更多开销。

3.2 一般 ActionScript 3 指导原则

下面的优化将会提高 AS3 执行速度。

3.2.1 严格数据类型确定

- 始终声明变量或类成员的数据类型。
- 努力避免声明对象 (Object) 类型的变量和类成员。对象是 AS3 中的最普通的数据类型。声明“对象”类型的变量与根本不声明类型相同。
- 努力避免使用数组 (Array) 类。而要使用矢量 (Vector) 类。数组是一种罕见的数据结构。除非需要在索引 4294967295（或接近）位置访问/设置值，否则不需要使用数组。
- 矢量允许您指定元素类型。类型 Vector<*> 意味着您可以在此矢量的实例中存储任何类型的数据。避免使用像 Vector<*> 这样的通用类型。而是将具体类型作为某个矢量的元素。例如，Vector<int>、Vector<String> 或 Vector<YourFavoriteClass> 占用的内存小得多，而且性能比 Vector<*> 好。
- 不要把对象用作散列表。而是使用类 **`flash.utils.Dictionary`**。
- 避免使用动态类（和动态属性）。此时在 GFx 或 Flash 中都不能优化对动态属性的访问。
- 避免使用/更改对象的原型。原型是 AS2 的组成部分。保留它的目的是保持在 AS3 中的兼容性，但在 AS3 中，同样的功能可以通过使用静态函数和成员来实现。

3.3 前进播放

如果前进播放执行时间太长，有以下六种优化方法：

1. 不要在每个帧都执行 AS 代码，避免 onEnterFrame/Event.ENTER_FRAME 句柄调用，该句柄在每个帧都需要调用代码。
2. 使用时间驱动编程条例，在发生改变时，通过外部调用改变文本域和 UI 状态值。
3. 在隐藏动画剪辑中停止动画(_visible 属性应该设置为 true；使用 stop() 函数停止动画)。这样可以从 Advance 列表中排除这样的动画剪辑。注意必须停止每个层级的单独动画剪辑，包括其子党员，甚至在上级党员动画剪辑已经停止的情况下也是如此。
4. 一种可供选择的做法为_global.noInvisibleAdvance(AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)扩展的使用。该扩展函数可以从 Advance 列表中排除隐藏视频剪辑分组，而不需要将其停止。如果扩展属性设置为"true"则隐藏视频剪辑不添加到 Advance 列表（包括其子对象）从而提高性能。注意这项技术不完全与 Flash 兼容。确保 Flash 文件不依赖于任何类型的隐藏动画帧处理。不要忘记通过使用该扩展函数（或其他函数）设置_global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3) 为 true 以打开 Scaleform 扩展属性。
5. 减少场景中的动画剪辑数量。限制不需要的动画剪辑嵌套，因为每个嵌套剪辑在前进中都需要一定数量的开销。
6. 减少时间轴动画、关键帧数量和形变动画。

3.4 onEnterFrame, Event.Enter_Frame

尽量少用 onEnterFrame(AS2) or Event.ENTER_FRAME (AS3)事件句柄，或采用最小安装在不需要时及时删除，而不要在所有时间都执行。具有大量 onEnterFrame/Event.ENTER_FRAME 句柄将明显降低性能。作为补充，可以使用 setInterval 和 setTimeout 函数。当使用 setInterval 时：

- 不要在句柄不再需要时忘记调用 clearInterval。
- setInterval 和 setTimeout 句柄在比 onEnterFrame/Event.ENTER_FRAME 句柄执行频繁时可能比 onEnterFrame/Event.ENTER_FRAME 句柄要慢。使用常数设定时间间隔来避免。

3.4.1 清理 onEnterFrame

使用 delete 操作删除 onEnterFrame 句柄：

```
delete this.onEnterFrame;
delete mc.onEnterFrame;
```

不要分配 null 或 undefined 到 onEnterFrame (例如 this.onEnterFrame = null;)，因为这个操作不能将 onEnterFrame 句柄完全删除。Scaleform 将试图解决这个句柄，因为名为 onEnterFrame 的成员函数仍然存在。

3.5 ActionScript 2 优化

3.5.1 onClipEvent 和 on Events

避免使用 onClipEvent()和 on()事件，而使用 onEnterFrame、 onPress 等。这样做的原因如下：

- 功能型事件句柄可在运行时安装和删除。
- 函数中的字节码比老式 onClipEvent 事件以及句柄具有更多优化属性。主要的优化为预先保存在缓存中的 this、 _global、arguments、super 等。并使用 256 个内部寄存器存放本地变量。该优化属性只用于函数。

在第一帧执行前需要安装 onLoad 函数类型句柄存在的唯一问题，可以使用无正式文档支持的事件句柄 onClipEvent(construct)来安装 onEnterFrame：

```
onClipEvent(construct)
{
      this.onLoad = function()
      {
            //函数体
      }
}
```

或者，使用 onClipEvent(load)并从中调用一个常规函数。该方法效率较低，因为将导致额外的函数调用开销。

3.5.2 Var 关键字

尽可能的使用 var 关键字。在函数内部该操作尤为重要，因为 AS 编译器使用内部寄存器优化本地变量访问，直接通过索引而不将数据放到哈希表中查找。使用 var 关键字可以使 AS 函数执行速度加倍。

未经优化代码：

```
var i = 1000;
countIt = function()
{
      num = 0;
      for(j=0; j<i; j++)
```

```
	{
		j++;
		num += Math.random();
	}
	displayNumber.text = num;
}
```

优化代码：

```
var i = 1000;
countIt = function()
{
	var num = 0;
	var ii = i;
	for(var j=0; j<ii; j++)
	{
		j++;
		num += Math.random();
	}
	displayNumber.text = num;
}
```

3.5.3 预存

将频繁访问的只读对象成员预存到本地变量（使用 var 关键字）。

未经优化代码：

```
function foo(var obj:Object)
{
	for (var i = 0; i < obj.size; i++)
	{
		obj.value[i] = obj.num1 * obj.num2;
	}
}
```

优化代码：

```
function foo(var obj:Object)
{
	var sz = obj.size;
	var n1 = obj.num1;
	var n2 = obj.num1;
	for (var i = 0; i < sz; i++)
```

```
      {
            obj.value[i] = n1*n2;
      }
}
```

预存可以用于其他情景更加有效，一些例子包括：

```
var floor = Math.floor
var ceil = Math.ceil
num = floor(x) - ceil(y);
```

```
var keyDown = Key.isDown;
var keyLeft = Key.LEFT;
if (keyDown(keyLeft))
{
      // 执行操作
}
```

3.5.4 预存长路径

避免重复使用长路径，如：

```
mc.ch1.hc3.djf3.jd9._x = 233;
mc.ch1.hc3.djf3._x = 455;
```

将文件路径预存到本地变量：

```
var djf3 = mc.ch1.hc3.djf3;
djf3._x = 455;

var jd9 = djf3.jd9;
jd9._x = 223;
```

3.5.5 复杂表达式

避免复杂、类 C 表达式，如下所示：

```
this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
_global.MyStrings[queue];
```

将这些表达式分割成更小单元，将交互数据保存在本地变量：

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
var slot_i = _global.slots[i];
_splqueue[slot_i].gosplash.text = _splstring;
```

如果在分割单元中有多个索引这个操作就特别重要，例如如下循环：

```
for(i=0; i<3; i++)
{
     this[_global.mynames[queue]][_global.slots[i]].gosplash.text =
                                                    _global.MyStrings[queue];
     this[_global.mynames[queue]][_global.slots[i]].gosplash2.text =
                                                    _global.MyStrings[queue];
     this[_global.mynames[queue]][_global.slots[i]].gosplash2.textColor =
                                                                0x000000;
}
```

之前循环的改进版本如下所示：

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
      var slot_i = _global.slots[i];
      _splqueue[slot_i].gosplash.text = _splstring;
      _splqueue[slot_i].gosplash2.text = splstring;
      _splqueue[slot_i].gosplash2.textColor = 0x000000;
}
```

以上代码还能够进一步优化，尽可能取消指向相同数组元素的多个索引。将本地变量中的既定变量预先存放在缓存中：

```
var _splqueue = this[_global.mynames[queue]];
var _splstring = _global.MyStrings[queue];
for (var i=0; i<3; i++)
{
      var slot_i = _global.slots[i];
      var elem = _splqueue[slot_i];
      elem.gosplash.text = _splstring;
      var gspl2 = elem.gosplash2;
      gspl2.text = splstring;
      gspl2.textColor = 0x000000;
}
```

4 HUD 开发

本章概要介绍 Scaleform 推荐的创作和人物显示(HUD)的最佳优化方法。下面列出的执行绝不是必须的；但是可以作为指导可以帮助开发者用 Scaleform 创作 HUD 时实现更好的性能和内存优化。

我们的建议按照复杂性和执行时间长度列出在递增列表中。如果多次创建一个 HUD，我们建议使用 4.1 小节中的方法并深入到列表反复直到创建最终版本。这是一种快速原型化可以在游戏中操作的 HUD 元素的方法。使用我们的推荐优化方法对 HUD 和资源进行提炼和优化。

请牢记如果开发一个极其复杂、具有高性能多个层的 HUD 并且需要很少内存的项目，使用 C++将非常高效，也许是最佳选择。

4.1 多 SWF 动画视图

总之，该类型的 HUD 对于开发和复用都很快捷。完全是艺术驱动并且在很短的周期内实现更好的效果以及更佳的图形描绘，这在优化之间进行原型化和重用设计是最佳选择，但是将增加内存使用。单个动画试图创建新的播放器实例增加大概 80K 的内存空间。需要在更快的 HUD 和单独的动画控制之间进行权衡，单独的动画控制将使用更多内存。将内存和性能因素与该动态、多个图层系统的易用性对比进行考虑。

使用多个 SWF 动画试图能够提供以下优势：

1. *可以在不同的线程调用 Advance*：一个多线程 HUD 界面允许在不同线程上执行或播放每个 Flash 动画（不是渲染而是整个播放过程，例如，时间轴、动画、AS 执行、Flash 处理）。这使得可以进行独立的前进播放控制；将 Flash 打散成多个动画可以使开发者控制特定元素的停止和前景，或者在不同时间用不同线程调用 Advance。HUD 的某些元素能够以更高的速率进行播放，或者开发者能够实际上在一个静态 HUD 元素上停止调用 Advance，直到游戏中发生一个事件需要执行一个动作为止。
   注释：Scaleform 允许在不同线程上调用不同 GFx::Movie 对象的 Advance；但是，由于每个动画实例不是线程安全显示，不能在没有同步的情况下调用不同线程。在使用 Advance 时同步还需要进行输入和调用。
2. *利用纹理渲染缓存*：调用渲染 HUD 元素和纹理并保存在缓存中，只在需要时进行更新。这将最少使用绘图原型，但是将占用更多内存，因为需要一个纹理缓存需要和 HUD 元素一样的内存空间。可以提高性能，增加内存使用量。只在元素极少更新并且非常简单的情况下可以考虑使用该方法。

4.2 单个动画试图包含多个SWF

本方法包含了多个 Flash 文件通过 AS `loadMovie` 命令导入到单个全屏动画视图中。该方法的优势包括更高效的内存使用和略微显示性能的提升。

我们推荐在导入多个 Flash 文件到单个动画试图时遵循以下事项：

1. 仔细将对象分组可以分批隐藏并将标记设为`_visible = false(AS2) or visible = false (AS3)`，同时将`_global. noInvisibleAdvance (AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)`设置为 true 以减少前进播放处理开销。**以下可能为创建 HUD 时可以进行的最重要的操作：**将可以分批隐藏的对象分组并添加一个主干来进行管理。使用`_visible/visible = false` 在对象隐藏时停止处理（注意：这不是用来控制可视状态而是在对象已经隐藏的情况下停止 Advance 调用）。通过隐藏特定对象的组，在那些 Flash 文件内部的元素不会调用执行逻辑。这在部分 HUD 需要隐藏和显示的地方特别有用（例如，暂停菜单、地图、身体状况栏）。我们也建议当需要隐藏/显示整个 HUD 时，开发者应该完全停止那个 Advance。不要忘记将`_global.gfxExtensions(AS2) or scaleform.gfx.Extensions.enabled (AS3)`设置为 `true` 开启 Scaleform 扩展属性。
2. 通过 `SetVariableArray` 和一个调用将应用程序中的多个变量分组更新。这在一个元素拥有多个不同组件时可以起到作用，一个高度复杂（例如，具有移动元素的地图），将更新分组采用单个调用对比地图上的每个图标进行单独调用。调用 `SetVariableArray` 来传递一个数组数据（例如，每个地图元素的新位置）然后单独调用以处理并使用数组数据，将条目都集中到单个函数执行。但是，若只有少量数据需要更新，不要使用这种方法，因为这样可能需要降低性能。
3. 当在创建 HUD 元素是明智得使用 `onEnterFrame (AS2)` or `Event.ENTER_FRAME (AS3)`，如果在 HUD 中有多个元素具有 `onEnterFrame/Event.ENTER_FRAME`，每次开发者调用 Advance 将被执行，甚至在特殊元素不被改变也是如此。
4. 保持动画帧速度为游戏帧速度的一半，只在需要时调用 AS。一个通常的游戏编程范例为在游戏引擎的每个帧都进行调用。这在使用 Flash 时明显不是理想方法。开发者需要避免每帧都调用 AS 脚本代码以减少内存使用并增加性能。例如，如果游戏运行速率为 30-40 FPS，只需以 15-20 FPS 的速率更新动画。但是，帧速率太慢动画调用不够频繁可能导致 HUD 动画可能会延迟、跳跃以及不够平滑。
5. 尽量缩短动画时间轴，因为延长的时间轴动画需要使用更多内存。但是，这必须小心处理，因为过短的时间轴将导致画面跳跃。

4.3 单个动画视图

使用这种方法可以非常有效得处理 Flash 复杂多元素界面（例如，雷达界面）。需要仔细对比 Flash 和 C++如何用来渲染元素。多个 Flash 图层导致独立的绘图原型，降低性能。确保充分利用 4.2 小节描述的指导方法另外还有以下方法：

1. 使用游戏引擎在接口内部绘制元素，而使用 Flash 来绘制边框和帧，用 Scaleform 处理文本。当一个 HUD 具有多个快速变化的元素时，Flash 可以用来绘制静态元素并用 C++渲染频繁改变的项。
2. 使用 C++进行元素定位而 Flash 进行绘制。雷达屏幕为一个很好的例子；创建一个 Flash 设置的点阵列，用 Render::Renderer 进行管理，使用 RenderString 标识进行元素标记。使用 C++引擎在渲染前重新正确定位元素。这将避免有些 AS 更新过度，但这十分复杂且需要额外编程。

4.4 单个动画视图 (Advanced)

使用该方法更加高级并且消耗更多时间，但是能够节省一些内存。在 HUD 创建和完善之前我们不建议使用这种方法。除了使用 4.3 小节描述的方法，还可以使用以下技术：

1. 只在 HUD 中图像改变时调用 Advance，或者用一个调用处理所有更新。该方法可以用于具有一个 HUD 动画更新的背景图层的单个动画，以及更多复杂 HUD 界面（例如，那些包含文本和进度条）在顶层没有动画并没有调用 Advance。身体状况条为一个很好的例子，通常不为动画；在没有发生变化时不需要调用 Advance。（例如，播放第一帧，停止，直到特性改变时候调用显示和 Advance/invoke）。这里进行的渲染更加有效，需要较少的 CPU 开销，但是管理复杂。
2. 利用自定义静态缓存管理用于顶点数据，并使用 Render::Renderer。该方法与 C++紧密相关，需要技术深厚的 C++图形程序员的参与。在 Render::Renderer 中，使用不同（自定义）视频内存矢量数据存储，并使用其他部分的 Scaleform 系统以使用静态缓存代替动态缓存来管理 HUD 元素。
3. 使用线程渲染和 Render::Renderer，这也许是最复杂的方法；需要重写渲染器并在 HUD 上调用一个独立的 Advance，由游戏引擎进行渲染。复杂层次的权衡可以获得更多潜在性能。

   注释：Scaleform–Unreal® Engine 3 集成环境中存在线程渲染，但是仍然需要大量编程来实现该方法。

4.5 不用 Flash 创建自定义 HUD

这个过程最为复杂且消耗大量时间。最终，HUD 应该纯粹用 C++和位图处理。Scaleform 和 Flash 可以在 HUD 创建和重用过程中始终发挥作用，但是在将 Flash 界面转变成位图形成最终版本后删除。在这一点上，Scaleform 不应执行任何播放操作或者将内存用户 HUD 元素，除非 Scaleform Player 在内存中产生事件。

根据所能承担，可以结合自定义调整的 C++渲染器用于性能要求较高的 HUD 用 Scaleform 进行其他的渲染操作。可以从外部自定义渲染器获益的区域为具有详细条目的最小地图和详细目录/状态界面；这些区域可以通过 DP 批处理和多个条目的有效更新进行优化，有些任务用 Scaleform 自动操作比较困难。其他的 HUD 元素如边框、面板和动画弹出框可以使用 Scaleform 处理，只有在这碰到编程瓶颈时进行别的方法替换。

不管上面说明，我们建议开发者继续使用 Scaleform 字体/文本引擎，特别是因为 Scaleform 包含一个 DrawText API 函数包可以使开发者使用 C++进行编程绘制文本，并且使用 Scaleform 用户界面中相同的 Flash 字体和文本系统。这样无需拥有两个单独字体系统可以节省内存：一个用户 HUD 另外一个用户其余的菜单系统。更多关于字体/文本引擎的信息，以及字体和文本的最佳优化方法，请参考"字体和文本"章节的 FAQ 和 "字体概述"/"DrawText API 参考" 文档。

总之，当在 HUD 上进行创建和重用时请注意遵循以下条例：

- 少用动画剪辑，只在必须时才使用嵌套条目。
- 尽可能不要使用蒙板，最多只能用一到两次。更多信息请参考“图像渲染和特效” 章节的 FAQ。
- 禁止 PC 和 Wii™（其他控制器已某人禁止）上的鼠标和键盘：
    - `GFx::Movie::SetMouseCursorCount(0);`
    - 如果禁止，不要进行输入。

- 将动画剪辑分组进行现实/隐藏，在 HUD 面板将 `noInvisibleAdvance (AS2) or scaleform.gfx.Extensions.noInvisibleAdvance (AS3)` 设置为 `_visible(AS2) or visible (AS3)= false`。
- 只在条目发生变化时进行调用。如果有两个或更多条目总是同时改变（相同的帧）则将条目变化集中到一个调用进行批量处理。不要对不常改变的帧使用这种处理方法（每帧）。
- 确保位图和色阶使用的优化。更多信息请参考 2.1 节或"艺术和资源"章节中的 FAQ。

5 常用优化提示

5.1 Flash 时间轴

时间轴上的帧和图层为 Flash 创作环境的两个重要部分。这些区域展示了资源存放位置和确定文档工作。时间轴和库如何设置和使用将影响整个 FLA 文件和总体可用性以及性能。

- 尽量少用基于帧的循环。基于帧的动画独立于应用程序帧速率，而基于时间模型则与 FPS 相关。
- 在不需要时立即停止基于帧的循环。
- 允许多个帧的复杂代码块分布。
- 与具有大量帧的 tween 动画时间轴类似，具有大量代码的脚本也需要大量处理器开销。
- 评估内容来判断动画/交互是否易于通过时间轴实现或者使用 AS 脚本简化进行模块化处理。
- 避免使用默认图层名字（如 Layer 1、Layer 2），因为在复杂文件中不易于记忆或资源的定位。

5.2 常用性能优化

- 结合转换可以实现更好的性能，例如，嵌套三种转换，手工计算一个矩阵。
- 如果速度变慢，检查内存泄漏。确保去除不再需要的东西。
- 在创作时，避免过多 trace()语句或动态更新文本域，因为这将影响性能。尽量不要更新太过频繁（例如，只有当发生改变时进行更新）。
- 如果可以，在时间轴图层顶部放置包括 AS 的图层以及一个包含帧标签的图层。例如，通常较好的惯例为将包含 AS 脚本的图层命名为 *actions*。
- 不要将帧脚本放在不同的图层；而将所有的脚本集中在一个图层。这将简化 AS 代码的管理并提高性能，消除了多个 AS 执行代码传递的开销。

6 附录

ActionScript 2.0 最佳优化
http://www.adobe.com/devnet/flash/articles/as_bestpractices.html

Flash 8 最佳优化
http://www.adobe.com/devnet/flash/articles/flash8_bestpractices.html

Flash ActionScript 2.0 学习向导
http://www.adobe.com/devnet/flash/articles/actionscript_guide.html