Autodesk®
Scaleform®

内存系统介绍

本文介绍了 Scaleform 中内存系统的设置、优化和管理。

作者： Michael Antonov, Maxim Shemanarev
版本： 4.01
最后修订 2011 年 2 月 17 号

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform 联系方式：

文档	内存系统介绍
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 Scaleform 3.x 内存系统

Autodesk® Scaleform® 3.0 将内存分配转变成为一种基于堆的策略，同时针对分配和堆引入详细的内存报告。内存报告可以通过编程方式或者通过内存和性能分析器 (AMP) 工具获得，从 Scaleform 3.2 起，该工具作为一个独立的应用程序提供。在新的 Scaleform 3.3 版本中，为解决原来的堆实现效率低下问题，更新了内存接口；第 1.2 节将会介绍这些更改。

本文其余部分将介绍 Scaleform 3.3 内存系统，详细阐述以下几个重要方面：

- 可重写的内存分配接口。
- 对 MovieViews 和 Scaleform 子系统进行基于堆的内存管理。
- ActionScript 垃圾回收。
- 可用的内存报告功能。

1.1 关键的内存概念

本节高度概述关键的 Scaleform 内存管理概念、它们之间的相互作用以及对内存使用情况的影响。建议开发者在自定义分配器和/或分析 Scaleform 内存使用情况之前先理解这些概念。

1.1.1 可重写的分配器 (Override Allocator)

Scaleform 进行的所有外部内存分配都要通过一个中央接口，该接口是启动时安装在 GFx::System 上的，允许开发者插入自己的内存管理系统。可以按照以下多种不同的方法来自定义内存管理：

1. 开发者可以安装自己的 Scaleform::SysAlloc 接口，实现该接口的 Alloc/Free/Realloc 函数，以便委派给内存分配器。如果应用程序采用一种集中分配器实现（如 dlmalloc），这种做法就非常适当。
2. 可以给 Scaleform 分配一个或多个固定大小的大内存块（在启动时预先分配）。其它块可以保留给像“暂停菜单”(Pause Menu) 这样的临时屏幕，并通过使用内存分配场地 (Memory Arena) 完全释放。
3. 使用 Scaleform 提供的系统分配器实现（如 Scaleform::SysAllocWinAPI）可以充分利用硬件分页 (Hardware Paging) 的优势。

所选择的方法取决于应用程序所采用的内存策略。案例 (2) 在为子系统分配有固定内存预算的控制台上十分常见。第 2 节“内存分配”中将详述不同的实现方法。

1.1.2 内存堆 (Memory Heap)

不管开发者在外部选用什么内存管理方法，所有内部 Scaleform 分配都是先组织成堆，然后按文件和按用途进行细分的。开发者在查看 AMP 或 MemReport 输出时很可能首先遇到这些问题。最常见的堆是：

- Global 堆 – 包含所有共享的分配和配置对象。
- MovieData 堆 – 存储从某个 SWF/GFx 文件加载的只读数据。
- MovieView 堆 – 代表单个 MovieView ActionScript 沙箱 (sandbox)，包含其时间轴和实例数据。此堆受制于内部垃圾回收。
- MeshCache – 这里分配棋盘格状三角形数据。

通过从我们称之为“页面”的专用内存块来处理给定文件子系统的分配，从而使内存堆实现发生作用。这种方法有如下诸多优点：

- 减少外部系统分配次数，因而提高性能。
- 消除了在只有一个线程访问的堆内进行的线程同步。
- 通过以一个单元释放相关数据来减少外部碎片 (external fragmentation)。
- 在内存报告上执行逻辑结构。

不过，堆也有一个严重缺陷 – 受制于内部碎片 (internal fragmentation)。当没有将堆内释放的小内存块释放到应用程序其余部分时，就会发生内部碎片，因为这些内存块的所有关联页面都不空闲，导致实际上该堆占有“闲置的”内存。

Scaleform 3.3 提供了一种基于 GSysAlloc 的新的堆实现方法，更加积极地释放内存，从而解决了这一难题。Scaleform 4.0 还使位于同一线程的 GFx::Movie 可以共享单个内存堆，因而可以减小部分填充的块所占有的内存。

1.1.3 垃圾回收 (Garbage Collection)

Scaleform 3.0 引入 ActionScript (AS) 垃圾回收功能，消除了 AS 数据结构内循环引用所导致的内存泄漏 (memory leak) 。妥善回收对于 CLIK 以及任何涉及的 ActionScript 程序的运行至关重要。

在 Scaleform 中，垃圾回收包含在 GFx::Movie 对象内，该对象充当 AS 虚拟机的一个执行沙箱。有了 Scaleform 3.3，就可以共享 Movie 堆，同时还可以共享和统一关联的垃圾回收器 (garbage collector)。多数情况下，回收是在 Movie 堆由于 ActionScript 分配而变大时自动触发的；不过，也可能直接触发，或者指示 Scaleform 按基于帧的明确间隔进行回收。第 4 节“垃圾回收”将详细介绍垃圾回收及其配置选项。

1.1.4 内存报告和调试

Scaleform 内存系统能够给分配标上 “stat ID” 标记，以描述所进行的分配的用途。然后可以按堆报告分配的内存，细分成 stat ID 类别，或者按 stat ID 报告，细分成堆。此内存报告功能主要是通过内存和性能分析器 (AMP) 公开的，从 Scaleform 3.2 开始，此 AMP 是作为一个独立的分析应用程序提供的。有关 AMP 使用方面的详细信息，请参阅 AMP User Guide。也可以通过调用 Scaleform::MemoryHeap::MemReport 函数来以编程方式获得字符串格式的内存报告。

上文所述的 stat ID 标记是作为调试数据与实际的内存分配分开存储的。调试数据也是为检测内存泄漏而保留的。当 Scaleform 调试版关闭时，Visual Studio 输出窗口或控制台就会生成一份内存泄漏报告。由于 Scaleform 没有任何已知内部内存泄漏，检测到的任何泄漏都很有可能是由于在 Scaleform 上引用计数使用不当造成的。额外的关联调试数据不在 Shipping 配置中分配。

1.2 Scaleform 3.3 内存 API 更改

如前所述，Scaleform 3.3 包括两个改进内存使用情况的重要更新：

- GFx::Movie 内存上下文共享特性使独立电影视图实例可以共享堆和垃圾回收，因而可以提高内存重复利用率。内部字符串表也可以共享，因而可以进一步节约内存。第 4 节将详细介绍此共享情况。
- SysAlloc 接口经过更新后更加“对 malloc 友好”，消除了对于外部分配的 4K 页面对齐要求，并使堆优化为将大于 512 字节的所有分配都直接路由到 SysAlloc。此方法导致更多珍贵的内存迅速返回到应用程序中，因而提高了效率和重复利用率。

请注意，原来的 Scaleform::SysAlloc 实现现已更名为 Scaleform::SysAllocPaged，而且仍然可用。重写 SysAlloc 的开发者可以选择更新其实现，或将基类更改为 SysAllocPaged。依赖 SysAllocStatic 和内存分配场地的用户不受影响，因为那个类仍然可用。

2 内存分配

如第 1.1.1 节所述，Scaleform 使开发者可以通过选择以下三种方法之一来自定义内存分配：

1. 重写 SysAlloc 接口，将分配委派给应用程序内存系统，
2. 给 Scaleform 提供一个或多个固定大小的内存块，或者
3. 指示 Scaleform 直接从操作系统分配内存。

要想实现最佳内存效率，开发者应该选择一种适合其应用程序的方法。下面探讨最常用的方法 (1) 与方法 (2) 之间的区别。

当开发者重写 SysAlloc 时，他们将 Scaleform 分配委派给自己的内存系统。例如，当 Scaleform 需要使用内存来加载新 SWF 文件时，Scaleform 会通过调用 Scaleform::SysAlloc::Alloc 函数来进行此请求；当卸载内容时，它会调用 Scaleform::SysAlloc::Free。在此情况下，当所有系统（包括 Scaleform 在内）都使用单个共享全局分配器时，就可以实现尽可能最大的内存效率，因为已释放的任何内存块均可供需要它的任何系统重复使用。任何区隔化 (compartmentalization) 分配方式均会降低整体共享效率，因而增大总体内存占用大小。下面的 2.1 节将提供重写 SysAlloc 的详细信息。

全局分配方法的最大障碍是碎片，这就是许多控制台开发者更喜欢使用预先确定的固定内存布局的原因。利用预定内存布局，可以为各个系统分配固定大小的内存区域，而其中每个内存区域都是在启动时或水平加载 (level loading) 时确定的。总之，这种方法就是用全局系统的效率换取一种更可预测的内存布局。

如果某个应用程序使用这种固定内存方法，开发者就应对 Scaleform 也使用固定大小内存块分配器 – 第 2.2 节将介绍这种配置。为使这种情形更加切合实际，Scaleform 还允许创建辅助内存分配场地，或仅限有限时间期间（如显示“暂停菜单”时）使用的内存区域。

2.1 关键内存区域分配

要更换外部 Scaleform 分配器，开发者可以遵循以下两个步骤：

1. 创建自己的 SysAlloc 接口执行器，需定义 GetInfo、 Alloc 和 Free 方法。
2. 在 Scaleform 初始化过程中用 GFx::System 构造器提供一个该调度分配器的实例。

Scaleform 提供了一种既可直接使用也可作为参考的默认 Scaleform::SysAllocMalloc 实现，这种实现依赖于标准的 malloc/free 及其针对系统的替代方案（支持校准 (alignment) ）。

2.1.1 实现自己的 SysAlloc 函数

实现自己的内存堆最简单的方法为拷贝 Scaleform SysAllocMalloc 并作相应修改以调用自己的内存分配器或堆。对 SysAllocMalloc 分配器代码的修改见以下内容：

```
class MySysAlloc : public SysAlloc
{
  public:
    virtual void* Alloc(UPInt size, UPInt align)
    {
        return _aligned_malloc(size, align);
    }

    virtual void  Free(void* ptr, UPInt size, UPInt align)
    {
        SF_UNUSED2(size, align);
        _aligned_free(ptr);
        return true;
    }
    virtual void* Realloc(void* oldPtr, UPInt oldSize,
                          UPInt newSize, UPInt align)
    {
        SF_UNUSED(oldSize);
        return _aligned_realloc(oldPtr, newSize, align);
    }
};
```

可以看出，内存分配执行相对简单。MySysAlloc 类从 SysAlloc 基本接口上派生而来并实现了三个虚拟函数：`GetInfo`、`Alloc` 和 `Free`。尽管实现这些函数必须尊重校准，但 Scaleform 将一般只请求小规模校准，如不超过 16 字节。oldSize 和 align 参数传递到 Free/Realloc 接口之中，以简化实现；它们有助于（例如）作为一对 Alloc/Free 调用的包装程序来实现 Realloc。

一旦创建了一个自己的分配器实例，在 Scaleform 初始化期间可以传递给 GFx::System 结构。

```
MySysAlloc myAlloc;
GFx::System gfxSystem(&myAlloc);
```

在 Scaleform 3.0 中，GFx::System 对象需要创建在其他 Scaleform 对象创建之前，在其他 Scaleform 对象释放之后被释放。这些步骤通常最好在分配初始化中完成，初始化中的函数调用 Scaleform 中使用的代码；但是，也可以作为其他分配对象的一部分，这些对象的生命周期长于 Scaleform 中的对象（GFx::System 不能全局声明）。如果发现这种用法不方便，可以使用 GFx::System::Init()和

GFx::System::Destroy()静态函数来代替，不需要创建对象。与 GFx::System 构造器类似，GFx::System::Init()拥有一个 SysAlloc 指针参数。

2.2 将固定内存块用于 Scaleform

如引言中所述，控制台开发者可以选择提前保留一个或多个内存块，并将它们传递到 Scaleform，而不用重写 SysAlloc。这是通过实例化 SysAllocStatic 来完成的，如下所示：

```
void*             pmemChunk = …;
SysAllocStatic  blockAlloc(pmemChunk, 6*1024*1024);
GFx::System        gfxSystem(&blockAlloc);
…
```

上例中传递一个 6 兆字节的存储块给 Scaleform 用户内存分配。当然，存储块在 `gfxSystem` 和 `blockAlloc` 对象消亡之前不能重用或释放。不过，可以出于临时目的（如为了显示暂停菜单）添加额外的“可释放”内存块。下面探讨这样的块，通过使用内存分配场地来为这些块提供支持。

但是用静态内存分配器时，开发者需要特别留意 `Scaleform` 使用的存储量，确保没有导入文件或创建动画实例以至超出指定的存储范围。一旦 SysAllocStatic 失败，将从 `Alloc` 执行函数返回 0，导致 Scaleform 运行失败或崩溃。如有需要，开发者可以使用简单的 SysAllocPaged 封装对象来检测这种危险情况。

2.2.1 内存实存块

Scaleform 3.1 介绍了对内存实存块的支持，在程序执行过程中的某个特定点上，能够保证用户规定的分配器区域全部被释放。更具体地来说，内存实存块所定义的内存区能够加载 Scaleform 文件以及创建 GFx::Movie，因此一旦占用这些区域的 Scaleform 文件被销毁，这些地区将会全部得到释放。一旦某一实存块被销毁，程序就能够将所有内存重新用于其他非 Scaleform 数据。作为一种可能的使用情况，某一内存实存块可被定义为加载游戏中的一个“暂停菜单”界面，这将暂时占用内存，但只要游戏恢复运行状态，它们将立即被释放并可重新使用。

2.2.1.1 背景

一般情况下，大部分开发商不需要通过 Scaleform 将内存实存块定义为重新使用内存占用。当 Scaleform 分配内存时，它将从 GFx::System 所规定的 SysAlloc 对象中获取；该内存将会被释放，因为您卸载了数据并销毁了 Scaleform 目标。然而，由于多种原因的存在，被释放的内存模式并不总能与分

配相匹配。举例来说，如果您调用 CreateMovie，使用它，然后释放它，很可能出现这样的情况，在创建调用过程中所分配的大部分内存块能够被释放，但其中的某一小部分可能会保持较长的一段时间。导致出现这种情况的原因有很多，包括碎片、动态数据结构、多线程、Scaleform 资源共享与缓存等。

个问题，因为在 Scaleform 的生命周期中，这些内存块不会累积并被重新使用。然而，如果您的游戏引擎用作固定大小的内存缓冲区，并需要与 Scaleform 和其他游戏引擎数据进行共享时，不可预测的分配性质倒是能够造成一定的麻烦。在 SDK 的早期版本中，开发人员可以关闭所有的 Scaleform 以确保缓冲区内存能够完全被释放。然而，在使用 Scaleform 3.1 时，用户能够为这些缓冲区定义内存实存块，并向特定的内存实存块分配 SWF/GFX 文件。一旦这些文件被卸载，Scaleform::Memory::DestroyArena 就能够被调用，并且所有的实存块内存也能够安全地重新使用。

2.2.1.2 使用内存实存块

为了使用内存实存块，开发人员需要完成以下几步：

1. 通过调用 Scaleform::Memory::CreateArena 来创建一个实存块，并为其赋予一个非零整数标识符（零意味着全球或默认实存块，总是存在的）。为了进入实存块，您需要提供一个 SysAlloc 端口；SysAllocStatic 可在固定大小的内存缓冲区内使用。

   ```
   // 假定pbuf1代表一个10,000,000字节的缓冲区。
   SysAllocStatic  sysAlloc(pbuf1, 10000000);
   Memory::CreateArena(1, &sysAlloc);
   ```

2. 调用 GFx::Loader::CreateMovie / GFx::MovieDef::CreateInstance，同时为实存块设置使用 id。那些视频对象所需的堆栈将在规定的实存块内创建，因此所需要的大部分内存将会来自这里。请注意，某些“共享的”全球内存仍会被分配，因此，您还需确保在全球范围内拥有足够的储备。

   ```
   // 该视频使用实存块1
   Ptr<GFx::MovieDef> pMovieDef =
               *Loader.CreateMovie(filenameStr, Loader::LoadWaitFrame1, 1);
   ...
   // 该示例使用实存块1
   Ptr<GFx::Movie> pMovie =
            *pnewMovieDef->CreateInstance(MovieDef::MemoryParams(1), false);
   ```

3. 必要时使用合成的 GFx::MovieDef/GFx::Movie 对象。

4. 释放处于实存块内所有已创建的视频和对象。如果 GFx::ThreadedTaskManager 用于线程后台加载，那么在释放前应当对 GFx::MovieDef::WaitForLoadFinish 进行调用，因为它将强迫后台线程终止加载并释放他们的视频引用。

```
pMovieDef->WaitForLoadFinish(true);
pMovie    = 0;
pMovieDef = 0;
```

or (if Scaleform::Ptr is not used):

```
pMovie->Release();
pMovieDef->Release();
```

5. 调用 DestroyArena。完成这一操作后，所有实存块的内存将能够安全地重复使用，直至 CreateArena 被再次调用为止。Memory::ArenaIsEmpty 函数可用于检查内存实存块是否为空以及是否准备撤销。

```
// 如果内存在调用前没有被正确释放的话，则DestroyArena将会生效。
//  在释放过程中它将会在"return *(int*)0;"中崩溃。
Memory::DestroyArena(1);
```

SysAlloc目标现在能够超出范围或被销毁，此后“pbuff1”可被重新使用。内存实存块能够在必要时重新被创建。

2.3 将分配委派给 OS

开发者不用重写分配器，而是可以选择使用随 Scaleform 提供的 OS-直接分配器之一。它们包括：

- *Scaleform::SysAllocWinAPI* – 使用 VirtualAlloc/VirtualFree API 函数，具有良好的对齐和页面调度功能，为在微软平台上的最佳选择；
- *Scaleform::SysAllocPS3* – 使用 sys_memory_allocate/sys_memory_free 调用，用于高效的页面管理和 PS3 上的对齐功能。

分页为操作系统(OS)的功能以重映射物理内存到页面内数据块的虚拟地址空间。如果和使用智能化的系统分配，分页可以在大存储块中减少碎片，实现方法为在小的自由存储块上划分线性内存范围，这些小的自由存储块分散在地址空间。
我们的 Scaleform WinAPI 和 PS3 SysAlloc 执行函数利用了映射和取消映射系统内存到粒度数据块的优势。当然，如果你之前已经保留了系统内存且尚未释放给系统，则无法实现此动态“磁盘碎片整理”。此粒度远比（例如） dlmalloc 使用的 2MB 块效率高。

2.3.1 实现 SysAllocPaged

随着 Scaleform 3.3 中引入一个更新的 SysAlloc 接口，原来的 SysAlloc 实现更名为 SysAllocPaged。Scaleform 中还有使用它的分配器实现，并且用于 SysAllocStatic 和特定 OS 的分配器实现。这些分配器实现包含在库中，但如果不用就不会被链接。出于兼容目的，可以实现 SysAllocPaged，而不是 SysAlloc。请看下文。

对 SysAllocMalloc 分配器代码的修改见以下内容：

```
class MySysAlloc : public SysAllocPaged
{
  public:
    virtual void  GetInfo(Info* i) const
    {
        i->MinAlign     = 1;
        i->MaxAlign     = 1;
        i->Granularity  = 128*1024;
        i->HasRealloc   = false;
    }

    virtual void* Alloc(UPInt size, UPInt align)
```

```
        {
            // Ignore 'align' since reported MaxAlign is 1.
            return malloc(size);
        }

        virtual bool  Free(void* ptr, UPInt size, UPInt align)
        {
            // free() doesn't need size or alignment of the memory block, but
            // you can use it in your implementation if it makes things easier.
            free(ptr);
            return true;
        }
    };
```

可以看出，内存分配执行相对简单。MySysAlloc 类从 SysAllocPaged 基本接口上派生而来并实现了三个虚拟函数：`GetInfo`、 `Alloc` 和 `Free`。另外还有一个 `ReallocInPlace` 函数用于优化，但在这里尚未提到。

1 `GetInfo()` – 返回分配器内存对齐支持能力和粒度，在 Scaleform::SysAllocPaged::Info 结构成员中填入这些值。

2 `Alloc` – 分配指定大小内存。函数包含了 size 和 align 两个参数，在 Alloc 执行时需要用到。传递的 align 值不能大于从 GetInfo 返回的 MaxAlign 值。由于在我们例子中已将 MaxAlign 设为 1 字节，我们可以忽略第二个参数 align。

3 `Free` – 释放先前 Alloc 分配的内存。函数包含 size 和 align 参数，告知原来分配空间大小；在我们的空间释放实现方式中无需此参数，可以忽略。

4 `ReallocInPlace` – 试图重复分配内存而不转移到不同的位置，如果不成功则返回 false。很多用户不需要用到这个函数；请参考 Scaleform 参考资料 获取详细信息。

可以看出，`Alloc` 和 `Free` 函数的都为标准的操作，所以只有 `GetInfo` 函数比较特殊。
该函数报告了 Scaleform 中分配器的功能，所以可以影响 SysAllocPaged 执行内存对齐支持和需求，以及 Scaleform 性能和内存使用效率。`SysAllocPaged::Info` 结构包含以下四个值：

- `MinAlign` – 分配器在所有分配中应用的最小内存对齐空间。
- `MaxAlign` – 分配器可以支持的最大内存对齐空间。如果值为 0，Scaleform 认为支持任何对齐空间。如果值为 1，则不支持任何对齐方式，在此情况下 Alloc 函数执行时可以忽略 align 参数。
- `Granularity` – 为 Scaleform 分配推荐粒度；Scaleform 中对分配请求空间大小至少为此大小的粒度。如果分配器如在 malloc 函数应用中不能友好对齐，我们推荐使用最小对齐空间为 64K。
- `HasRealloc` – 一个布尔标志指定执行器是否支持 `ReallocInPlace`；在本例中返回。

- `SysDirectThreshold` 定义了全球的规模门槛（在不为空的情况下）。如果分配的规模大于或等于 SysDirectThreshold，它将对系统进行重新定向，忽略粒层。
- `MaxHeapGranularity` 在不为空的情况下限制了可能的最大堆粒度。在大多数情况下，`MaxHeapGranularity` 能够减少常用部分价格的 alloc/free 操作（该操作将会使分配器变慢）对系统的内存占用。MaxHeapGranularity 必须至少为 4096，并且是 4096 的整数倍。

从 `MaxAlign` 参数的描述中可以看出，如果分配器不支持内存对齐，则 SysAlloc 接口实现很容易，但是也需要利用内存对齐的优势。在实际应用中，需要考虑三个方面：

1. *分配器执行不支持内存对齐，或处理无效*。如果为这种情况，设置 `MaxAlign` 为 1，使 Scaleform 内部完成所有需要的对齐工作。
2. *分配器能有效处理内存对齐*。如果为这种情况，设置 `MaxAlign` 为 0 或者能支持的最大内存对齐空间大小，执行 `Alloc` 函数正确处理内存对齐。
3. *分配器为系统的接口*，页面大小总是需指定和对齐。为有效处理，设置 `MinAlign` 和 `MaxAlign` 为系统页大小值并将所有的分配空间值传递给操作系统。

一旦创建了一个自己的分配器实例，在 Scaleform 初始化期间可以传递给 GFx::System 结构。

```
MySysAlloc myAlloc;
GFx::System gfxSystem(&myAlloc);
```

2.4 存储堆

前面已经提到，Scaleform 在称为“堆”的内存池中维护内存。每个堆都是出于特定目的创建的，例如：为了保存从像网格缓存 (Mesh Cache) 这样的特定文件子系统加载的数据。本节概述用来在 Scaleform 核心内管理堆内存的 API。

2.4.1 堆 API 函数

为便于用户设置，大多数内存堆管理的详细内容隐藏在代码内部。开发者可以在所有的 Scaleform 对象上直接使用“new”操作符，将执行堆内存分配。

```
Ptr<GFx::FileOpener> opener = *new GFx::FileOpener();
loader.SetFileOpener(opener);
```

可以看到，操作符‘new’的使用没有任何堆描述信息，所以内存分配将发生在 Scaleform 全部堆内部。

因为配置对象的数量很少，这个方法可以很好得工作。全局堆是在初始化 GFx::System 期间创建的，并保持活动状态，直到关闭 GFx::System。它始终使用内存分配场地 0。

然而，可以在内部使用内存堆控制内存碎片。例如，GFx::MeshCacheManager 类创建一个内部堆保存和约束所有栅格形状数据。`GFx::Loader::CreateMovie()`创建一个堆保存从特定 SWF/GFX 文件导入的数据。`GFx::MovieDef::CreateInstance` 创建一个堆来保存时间轴和 ActionScript 运行状态信息。所有这些详细信息位于后台，不应该对大多数用户产生影响，至少在你计划改变、扩展或调试 Scaleform 代码之前不应改产生影响。

如前面提到，存储堆由 Scaleform::MemoryHeap 类表示，伴随为特定 Scaleform 子系统或数据设置产生的实例。 每个子存储堆使用 Scaleform::MemoryHeap::CreateHeap 在全局堆上创建调用 Scaleform::MemoryHeap::Release 销毁。在堆的生命周期内，可以调用 Scaleform::MemoryHeap::Alloc 函数分配内存和 Scaleform::MemoryHeap::Free 函数释放内存。当堆被销毁后，所有的内部存储被释放。鉴于堆的空间容量，这些内存通常可以通过 SysAlloc 接口立即释放给游戏应用程序。

为确保内存中正确的堆分配，Scaleform 中的大多数分配使用特别的宏包含一个 Scaleform::MemoryHeap 指针。这些宏包括：

- SF_HEAP_ALLOC(heap, size, id)
- SF_HEAP_MEMALIGN(heap, size, alignment, id)
- SF_HEAP_NEW(heap) ClassName
- SF_FREE(ptr)

宏的使用确保正确的字符行和文件名在调试编译时用户分配内存，‘id’号用来标记分配用途，这些标识符在 Scaleform AMP 内存分析系统中分成组。

设定内存分配宏，一个新的 GFx::SpriteDef 对象可以由一个自定义堆名‘pHeap’创建，调用形式如下所示：

```
Ptr<GFx::SpriteDef> def = *SF_HEAP_NEW(pHeap) GFx::SpriteDef(pdataDef);
```

这里，GFx::SpriteDef 为在 Scaleform 中使用的一个内部类。由于 Scaleform 设计了大多数对象，通常在指针对象 Ptr<>中描述，当这些指针超出范围，调用内部释放函数进行销毁。对于不包括在内的对象，如从 Scaleform::NewOverrideBase 继承来的对象，可直接使用‘delete’操作符。没有必要用指定堆来释放内存，delete 操作和 SF_FREE（地址）宏可以自动找出堆。

2.4.1 自动堆

尽管内存分配宏提供了所有必要的功能来支持堆，但是有时用起来需要更多消耗，因为宏需要一个堆指针传递给程序。在用来聚集容器对象需要内存分配时这些指针传递将变得非常麻烦，如 Scaleform::String 或 Scaleform::Array<>。见下面实例代码声明一个类并在堆中创建一个实例：

```
class Sprite : public RefCountBase<Sprite>
{
    String                  Name;
    Array<Ptr<Sprite> > Children;

    Sprite(String& name, …) { … }
};
Ptr<Sprite> sprite = *SF_HEAP_NEW(pHeap) Sprite(name);
sprite->DoSomething();
```

尽管 GFx::Sprite 对象本身创建于指定堆，包含了一个字符串和一个数组对象需要被动态分配。如果没有特殊考虑，这些对象将在全局堆中结束分配 - 这种情况可能性最大，但不是理想中的情况。为了解决这个问题，一个堆指针理论上可以被传递到 GFx::Sprite 对象结构体然后进入字符串和数组结构体。这些参数传递将使编程变得更为困难，潜在需要很多对象，如数组和字符串来维持一个指针，该指针指向堆和其他地方消耗的内存。

为了使该情况下的变成更加容易，Scaleform 内核使用了特殊的“自动堆”分配宏：

- SF_HEAP_AUTO_ALLOC(ptr, size)
- SF_HEAP_AUTO_NEW(ptr) ClassName

这些宏的行为与同属的 SF_HEAP_ALLOC 和 SF_HEAP_NEW 非常类似，但是有一个明显的区别：他们拥有一个指针指向一个内存分配而不是指向堆。当被调用时，这些宏将自动辨别建立在所提供的内存地址上的堆并从相同的堆中分配内存。见以下例子：

```
MemoryHeap*    pheap = …;
UByte *         pbuffer = (UByte*)SF_HEAP_ALLOC(pheap, 100,
GFx::StatMV_ActionScript_Mem);
Ptr<GFx::Sprite> sprite = *SF_HEAP_AUTO_NEW(pbuffer + 5) GFx::Sprite();
…
sprite->DoSomething();
...
// Release sprite and free buffer memory.
sprite = 0;
SF_FREE(pbuffer);
```

在本例中我们从指定堆创建一个缓冲存储器以及在相同堆上创建一个 GFx::Sprite 对象作为缓冲存储器。本例中特有的方法为不将‘pbuffer’指针传递到 SF_HEAP_AUTO_NEW；而传递一个缓存分配范围内的地址。Scaleform 存储堆系统，具有辨识一个存储堆的功能，存储堆可以为一个分配存储器上的任何地址（这样做更加高效）。这种独特的方法为解决堆传递问题的关键，这些问题在本章开头已有描述。配置了自动堆辨识，我们可以创建容器从正确位置的堆中自动分配内存，也即我们可以重写前面提到的 GFx::Sprite 类，如下所示：

```
class Sprite : public RefCountBase<Sprite>
{
    StringLH                    Name;
    ArrayLH<Ptr<Sprite> > Children;

    Sprite(String& name, …) { … }
};
Ptr<Sprite> sprite = *SF_HEAP_NEW(pHeap) Sprite(name);
sprite->DoSomething();
```

这里，我们已将位于相同本地堆中 Scaleform::String 和 Scaleform::Array<>类替换为相等的“本地堆” Scaleform::StringLH 和 Scaleform::ArrayLH<>。这些本地堆容器从包含相同对象的堆分配内存，使用了前面提到的“自动堆”技术，这个过程不需要参数传递，参数传递在其他地方是必须的。你可以发现这些和类似的容器在 Scaleform 内核中到处被使用，必须确保内存在正确的堆中结束。

2.4.2 堆跟踪程序

用户可以通过启用 GFxConfig.h 中定义的 SF_MEMORY_TRACE_ALL 来跟踪基本内存分配情况。通过调用 Scaleform::Memory 类中的 SetTracer 方法，可以跟踪 CreateHeap、DestroyHeap、Alloc、Free 等内存操作。

下面提供在 SysAllocWinAPI 的情况下如何使用跟踪程序的一个示例代码：

```
#include "Kernel/HeapPT/HeapPT_SysAllocWinAPI.h"

class MyTracer : public Scaleform::MemoryHeap::HeapTracer
{
 public:
    virtual void OnCreateHeap(const MemoryHeap* heap)
    {
        //...
    }

    virtual void OnDestroyHeap(const MemoryHeap* heap)
    {
        //...
```

```
    }

    virtual void OnAlloc(const MemoryHeap* heap, UPInt size, UPInt align,
                                 unsigned sid, const void* ptr)
    {
        //...
    }

    virtual void OnRealloc(const MemoryHeap* heap, const void* oldPtr,
                               UPInt  newSize, const void* newPtr)
    {
        //...
    }

    virtual void OnFree(const MemoryHeap* heap, const void* ptr)
    {
        //...
    }
};

static MyTracer tracer;
static SysAllocWinAPI sysAlloc;
static GFx::System* gfxSystem;


class MySystem : public Scaleform::System
{
 public:

    MySystem() : Scaleform::System(&sysAlloc)
    {
        Scaleform::Memory::GetGlobalHeap()->SetTracer(&tracer);
    }
    ~MySystem() { }
};


SF_PLATFORM_SYSTEM_APP(FxPlayer, MySystem, FxPlayerApp)
```

3 内存报告

尽管 Scaleform 3.0 原来引入详细的内存报告，作为 Scaleform Player AMP HUD 的组成部分，但现在已经将分析功能迁移到一个独立的 AMP 应用程序之中，该应用程序可远程连接到 PC 和控制台 Scaleform 应用程序。有关 AMP 使用方面的详细信息，请参阅 AMP User Guide。

尽管为了使 Scaleform Player 更加轻便，已从其中删除了 HUD UI，但仍然可以生成内存报告并通过按 Ctrl + F5 键发送到控制台。此报告是 Scaleform::MemoryHeap::MemReport 函数生成的可能的格式之一：

```
void MemReport (class StringBuffer& buffer, MemReportType detailed,
                    bool xmlFormat = false);
void MemReport (struct MemItem* rootItem, MemReportType detailed);
```

MemReport 生成一份内存报告，可能的格式是 XML，写到一个提供的字符串缓冲区。
然后可以将此字符串写到输出控制台或用于调试的屏幕。MemReportType 参数可以是下列情况之一：

- MemoryHeap::MemReportBrief – Scaleform 系统所消耗总内存的摘要，如下例所示：

```
Memory 4,680K / 4,737K
    Image                                       1,052
    Sound                                       136
    Movie View                                  1,739,784
    Movie Data                                  300,156
```

- MemoryHeap::MemReportFull –此内存类型为按 Ctrl-F6 时 Scaleform Player 输出的内存类型。它包括一个系统内存摘要以及具体堆内存摘要，而且，如果定义了 SF_MEMORY_ENABLE_DEBUG_INFO，它还包括，每个堆段内都有已分配内存的明细信息（按 stat ID）。下面显示一个按堆给出的信息的示例：

```
Memory 4,651K / 4,707K
    System Summary
        System Memory FootPrint                 4,832,440
        System Memory Used Space                4,775,684
        Debug Heaps Footprint                   12,884
        Debug Heaps Used Space                  5,392
    Summary
        Image                                   1,052
        Sound                                   136
        Movie View                              1,715,128
```

```
    Movie Data                                  300,156
[Heap] Global                                   4,848,864
    Heap Summary
        Total Footprint                         4,848,864
        Local Footprint                         2,427,860
        Child Footprint                         2,421,004
        Child Heaps                             4
        Local Used Space                        2,417,196
        DebugInfo                               120,832
    Memory
        MovieDef
            Sounds                              8
            ActionOps                           80,476
            MD_Other                            200
        MovieDef                                36
        MovieView
            MV_Other                            32
        General                                 91,486
        Image                                   92
        Sound                                   136
        String                                  31,425
        Debug Memory
            StatBag                             16,384
        Renderer
            Buffers                             2,097,152
            RenderBatch                         5,696
            Primitive                           1,512
            Fill                                900
            Mesh                                4,884
            MeshBatch                           2,200
            Context                             15,184
            NodeData                            15,160
            TreeCache                           13,220
            TextureManager                      2,336
            MatrixPool                          4,096
            MatrixPoolHandles                   2,032
            Text                                1,232
        Renderer                                13,488
    Allocations Count                           1,822
```

- MemoryHeap:: MemReportFileSummary –此内存类型生成按文件显示的消耗的内存。该报告按 stat ID 将每个 SWF 文件的 MovieData、MovieDef 和所有 MovieView 的值合计在一起。其格式如下例所示：

```
Movie File 3DGenerator_AS3.swf
    Memory                                      1,956,832
```

```
        MovieDef                                152,664
            CharDefs                            39,848
            ShapeData                           1,716
            Tags                                73,656
            MD_Other                            37,444
        MovieView                               196,249
            MovieClip                           120
            ActionScript                        181,917
            MV_Other                            14,212
        General                                 1,596,712
        Image                                   960
        String                                  2,783
        Renderer                                7,464
            Text                                7,464
```

- MemoryHeap::MemReportHeapDetailed – 此内存类型是 AMP 在内存选项卡中显示的内存类型，而且是最详细的报告，显示每个堆使用多大内存、多大内存用于调试信息以及 Scaleform 已申请但尚未使用的内存大小。例如：

```
Total Footprint                             4,858,148
 Used Space                                 4,793,972
     Global Heap                            2,405,492
         MovieDef                           80,720
             Sounds                         8
             ActionOps                      80,476
             MD_Other                       200
         MovieView                          32
             MV_Other                       32
         General                            92,566
         Image                              92
         Sound                              136
         String                             31,276
         Debug Memory                       8,192
             StatBag                        8,192
         Renderer                           2,182,960
             Buffers                        2,097,152
             RenderBatch                    5,696
             Primitive                      1,512
             Fill                           900
             Mesh                           4,884
             MeshBatch                      2,200
             Context                        15,184
             NodeData                       15,016
             TreeCache                      13,136
             TextureManager                 2,336
             MatrixPool                     8,192
             MatrixPoolHandles              2,032
             Text                           1,232
```

```
    Movie Data Heaps                        300,156
        MovieData "3DGenerator_AS3.swf"     300,156
            MovieDef                        152,664
                CharDefs                    39,848
                ShapeData                   1,716
                Tags                        73,656
                MD_Other                    37,444
            General                         141,332
            Image                           960
            String                          2,312
    Movie View Heaps                        1,727,936
        MovieView "3DGenerator_AS3.swf"     1,727,936
            MovieView                       195,977
                MovieClip                   120
                ActionScript                181,645
                MV_Other                    14,212
            General                         1,467,668
            String                          471
            Renderer                        7,464
                Text                        7,464
    Other Heaps                             360,580
        _FMOD_Heap                          360,580
            General                         359,944
Debug Data                                  12,884
Unused Space                                51,292
    Global                                  51,292
        _FMOD_Heap                          4,384
        MovieData "3DGenerator_AS3.swf"     5,356
        MovieView "3DGenerator_AS3.swf"     33,032
```

上面已经提到，当使用 SF_MEMORY_ENABLE_DEBUG_INFO 编译 Scaleform 时，stat ID 标记描述上述分配的用途。下面简述其中一些 stat ID 类别：

- MovieDef – 代表已加载的文件数据。完成文件加载后此内存不应继续增大。
- MovieView – 包含 GFx::Movie 实例数据。如果 ActionScript 正在分配许多对象，或者目前存在其他对象的许多电影剪辑，此类别的大小就会增加。
- CharDefs – 各个电影剪辑或其他 Flash 对象的定义。
- ShapeData – 复杂形状和字体的矢量表示。
- Tags – 动画关键帧数据。
- ActionOps – ActionScript 字节码。
- MeshCache – 包含矢量细分单元 (vector tesselator) 和 EdgeAA 生成的缓存形状网格数据。此类别随新的形状被细化而不断增大，最终达到一个极限。

- FontCache – 包含缓存的字体数据。

4 垃圾回收

Scaleform 版本 2.2 及早期版本使用一个简单的索引计数机制来处理 ActionScript 对象。大多数情况下这已够用。但是，在两个或更多对象互相关联，ActionScript 允许你创建称之为“circular references”或“cycle references”对象。这个循环引用造成了内存泄漏影响系统性能。常见下面一个例子的代码，将产生泄漏，除非其中有一个对象索引被明确分离开。

Code:

```
var o1 = new Object;
var o2 = new Object;
o1.a = o2;
o2.a = o1;
```

理论上，可以重写这些代码以避免循环引用或者使用一个清理函数将对象从循环引用分离出来。在多数情况下，清理函数不能起到作用，如果使用了 ActionScript 类和组件，反而问题会变得更加糟糕。常见情况下单独使用会导致内存泄漏，使用标准 Flash UI 组件也是如此。

为解决循环引用的问题，Scaleform 3.0 中引入了一个高度优化的引用计数器 - 基于清除机制称为“垃圾回收”。Scaleform 垃圾回收与通常的垃圾回收机制类似；但是，对其进行了优化用于引用计数。如果用户的 ActionScript 不创建“循环引用”，该机制作为常规应用计数系统。在循环引用被创建的情况下，才执行回收函数并在需要时释放这些对象。对于大多数 Flash 文件性能影响很小。

默认情况下，垃圾回收在 Scaleform 3.x 中为有效。当使用了垃圾回收机制，所有先前循环引用导致的的内存泄漏将被消除。如果不需要，垃圾回收功能可以禁止。但是，只有拥有源代码的用户有此权限，因为需要重新编译 Scaleform。

如要关闭垃圾回收机制，必须开打文件 *Include/GFxConfig.h* 取消 GFX_AS_ENABLE_GC 宏（默认情况下为注释掉的）：

```
// Enable garbage collection
#define GFX_AS_ENABLE_GC
```

在取消宏后，必须完全重新编译 Scaleform。

4.1 配置可用于垃圾回收的内存堆

Scaleform 3.x 允许配置内存堆支持垃圾回收机制。Scaleform 3.3 和更高版本都允许进行堆共享， 因而可以在多个 MovieView 之间实现垃圾回收器共享。

要为给定 MovieDef 创建一个新堆，可以使用如下函数：

```
GFx::Movie* GFx::MovieDef::CreateInstance(const MemoryParams& memParams,
                                          bool initFirstFrame = true)
```

MemoryParams 结构体为 GFx::MovieDef 类中的一个嵌套类，如下所示：

```
struct MemoryParams
{
      MemoryHeap::HeapDesc   Desc;
      float                  HeapLimitMultiplier;
      unsigned               MaxCollectionRoots;
      unsigned               FramesBetweenCollections;
};
```

- Desc – 为动画所创建堆的描述符。可以指定堆对齐方式（MinAlign）、粒度(Granularity)、限制范围(Limit)和标志(Flags)。如果指定了堆的范围，则 Scaleform 将堆的使用限制在这个范围之内。
- HeapLimitMultiplier – 一个多点堆限制乘法器（0...1），用来确定超出限定范围后堆的增加方式。见下面内容获取详细信息。
- MaxCollectionRoots – 执行垃圾回收程序前 roots 数。该值为最大 roots 的初始值；在执行中可以增加。见下面内容获取详细信息。
- FramesBetweenCollections – 执行完成一定数量的帧后强制调用回收函数，甚至尚为达到空间上限。在当前最大 roots 值较高，更好得用于执行回收功能，以减少回收发生时的消耗。见下面内容获取详细信息。

同样地，可用两个步骤来创建一个 Movie View 堆：

```
MemoryContext* MovieDef::CreateMemoryContext(const char* heapName,
                                             const MemoryParams& memParams,
                                             bool debugHeap )

Movie* MovieDef::CreateInstance(MemoryContext* memContext,
                                bool initFirstFrame = true)
```

MemoryContext 对象封装 Movie View 堆以及其他特定堆的对象，如垃圾回收器。这通过内存上下文创建 Movie View 及其堆的第二种方法增强了指定 AMP 显示的堆名的灵活性，无论是否需要线程安全，以及是否将该堆标记为调试并因此被排除在 AMP 报告之外。更重要的是，它使同一线程上的多个 Movie View 可以共享单个堆、垃圾回收器、字符串管理器以及文本分配器，从而减少了额外开销。

MemoryParams::Desc 被用来指定内存堆的常规属性，这些堆用于指定内存分配的实例（例如，ActionScript 分配）为控制堆的范围可以设置两个参数：Desc.Limit 和 HeapLimitMultiplier。堆已经初始化为预设范围为 128K(所以称之为动态分配）。当超过这个范围，将调用一个特殊的内部句柄。这个句柄可以从逻辑上进行判断：释放空间或者扩展堆。采用何种方式需要根据 Boehm-Demers-Weiser (BDW)算法垃圾回收器和内存分配器来做出判断。

BDW 算法如下（伪代码）：

```
if (allocs since collect >= heap footprint * HeapLimitMultiplier)
    collect
else
    expand(heap footprint + overlimit + heap footprint *
                                                    HeapLimitMultiplier)
```

步骤“collect”包括了 ActionScript 垃圾回收器标识和其他内存释放，如内部缓存刷新。

HeapLimitMultiplier 的默认值为 0.25。因此，由于最终的内存占用大于当前堆的 25%（HeapLimitMultiplier 默认值），则 Scaleform 将执行内存释放。否则，将扩展限制范围 25%加上请求增加的堆空间。

如果用户指定了 Desc.Limit，然后上述算法根据指定的限制范围按照相同的方式执行。如果堆限制范围超出 Desc.Limit 设定的值，则调用回收函数，无需考虑最终回收的分配空间量。动态堆限制范围将设置为堆的空间加上所需的分配请求范围（该情况下，如果释放的空间少于分配所需）。

第二种控制垃圾回收行为的方法为指定 MaxCollectionRoots/FramesBetweenCollections。MaxCollectionRoots/ FramesBetweenCollections. MaxCollectionRoots指定了roots数量，超过该数量则调用垃圾回收。这里的“root”意思为任何ActionScript对象可能与其他ActionScript对象产生循环引用的ActionScript对象。通常，一个ActionScript对象一旦与另外的ActionScript对象（意为ActionScript引用该对象，例如“obj.member = 1”涉及到对象“obj”）相关联则添加到roots数组。基本上，我们将每个涉及的ActionScript对象作为root，因此，MaxCollectionRoots指定当前与ActionScript协同工作的对象大概数量。因此，当涉及对象数量超过MaxCollectionRoots指定值时将调用垃圾回收。默认情况下，该参数设置为0以关闭这项机制。

FramesBetweenCollections 指定了帧数量，播完这些帧之后强制调用垃圾回收函数。这里的条目“frame”表示一个 Flash 帧。因此，如果一个 SWF 文件帧速率为 30 fps，则一秒钟播放 30 帧，2 秒种

为 60 帧，依次类推。这个值用来避免在 Advance 操作中降低性能，当垃圾回收需要遍历许多对象时导致性能下降。本例中，这个值可以设置为如 1800，垃圾回收每 60 秒钟调用一次，帧速率为 30 fps。`FramesBetweenCollections` 参数可以避免当 `Desc.Limit` 未被设置情况下 ActionScript 内存堆过多增长。默认情况下，该参数设置为 0，表示关闭此项功能。

4.2 GFx::Movie:: ForceCollectGarbage

```
virtual void  ForceCollectGarbage() = 0;
```

本方法可以用来让应用程序执行强制垃圾回收程序。如果垃圾回收功能为 off 不产生任何效果。