Autodesk®
Scaleform®

記憶體系統介紹

本文介紹了 Scaleform 中記憶體系統的設置、優化和管理。

作者： Michael Antonov, Maxim Shemanarev
版本： 4.01
最後修訂 2011 年 2 月 17 號

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.2

Autodesk Scaleform 聯繫方式：

文檔	記憶體系統介紹
地址	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
網站	www.scaleform.com
郵箱	info@scaleform.com
電話	(301) 446-3200
傳真	(301) 446-3199

目錄

1 Scaleform 3.x 記憶體系統

Autodesk® Scaleform® 3.0 將記憶體分配轉變成為一種基於堆的策略,同時針對分配和堆引入詳細的記憶體報告。記憶體報告可以通過程式設計方式或者通過記憶體和性能分析器 (AMP) 工具獲得,從 Scaleform 3.2 起,該工具作為一個獨立的應用程式提供。在新的 Scaleform 3.3 版本中,為解決原來的堆實現效率低下問題,更新了記憶體介面;第 1.2 節將會介紹這些更改。

本文其餘部分將介紹 Scaleform 3.3 記憶體系統,詳細闡述以下幾個重要方面：

- 可重寫的記憶體分配介面。
- 對 MovieViews 和 Scaleform 子系統進行基於堆的記憶體管理。
- ActionScript 垃圾回收。
- 可用的記憶體報告功能。

1.1 關鍵的記憶體概念

本節高度概述關鍵的 Scaleform 記憶體管理概念、它們之間的相互作用以及對記憶體使用情況的影響。建議開發者在自訂分配器和/或分析 Scaleform 記憶體使用情況之前先理解這些概念。

1.1.1 可重寫的分配器(Override Allocator)

Scaleform 進行的所有外部記憶體分配都要通過一個中央介面,該介面是啟動時安裝在 GFx::System 上的,允許開發者插入自己的記憶體管理系統。可以按照以下多種不同的方法來自訂記憶體管理：

1. 開發者可以安裝自己的 Scaleform::SysAlloc 介面,實現該介面的 Alloc/Free/Realloc 函數,以便委派給記憶體分配器。如果應用程式採用一種集中分配器實現(如 dlmalloc),這種做法就非常適當。
2. 可以給 Scaleform 分配一個或多個固定大小的大區塊(在啟動時預先分配)。其它塊可以保留給像“暫停功能表”(Pause Menu) 這樣的臨時螢幕,並通過使用記憶體分配場地 (Memory Arena) 完全釋放。
3. 使用 Scaleform 提供的系統分配器實現(如 Scaleform::SysAllocWinAPI)可以充分利用硬體分頁 (hardware paging) 的優勢。

所選擇的方法取決於應用程式所採用的記憶體策略。案例 (2) 在為子系統分配有固定記憶體預算的主控台上十分常見。第 2 節“記憶體分配”中將詳述不同的實現方法。

1.1.2 記憶體堆(Memory Heap)

不管開發者在外部選用什麼記憶體管理方法,所有內部 Scaleform 分配都是先組織成堆,然後按檔和按用途進行細分的。開發者在查看 AMP 或 MemReport 輸出時很可能首先遇到這些問題。最常見的堆是：

- Global 堆 –包含所有共用的分配和配置物件。
- MovieData 堆 – 存儲從某個 SWF/GFx 檔載入的唯讀資料。
- MovieView 堆 – 代表單個 GFx::Movie ActionScript 沙箱 (sandbox),包含其時間軸和實例資料。此堆受制于内部垃圾回收。
- MeshCache –這裡分配棋盤格狀三角形資料。

通過從我們稱之為“頁面”的專用區塊來處理給定檔子系統的分配,從而使記憶體堆實現發生作用。這種方法有如下諸多優點：

- 減少外部系統分配次數,因而提高性能。
- 消除了在只有一個執行緒訪問的堆內進行的執行緒同步。
- 通過以一個單元釋放相關資料來減少外部碎片(external fragmentation)。
- 在記憶體報告上執行邏輯結構。

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 提供了一種基於 Scaleform::SysAlloc 的新的堆實現方法,更加積極地釋放記憶體,從而解決了這一難題。Scaleform 3.3 還使位於同一線程的 GFx::Movie 可以共用單個記憶體堆,因而可以減小部分填充的塊所佔有的記憶體。

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 資料結構內迴圈引用所導致的記憶體洩漏 (Memory Leak)。妥善回收對於 CLIK 以及任何涉及的 ActionScript 程式的運行至關重要。

在 Scaleform 中,垃圾回收包含在 GFx::Movie 物件內,該物件充當 AS 虛擬機器的一個執行沙箱。有了 Scaleform 3.3,就可以共用 Movie 堆,同時還可以共用和統一關聯的垃圾回收器 (Garbage Collector)。多數情況下,回收是在 Movie 堆由於 ActionScript 分配而變大時自動觸發的;不過,也可能直接觸發,或者指示 Scaleform 按基於幀的明確間隔進行回收。第 4 節“垃圾回收”將詳細介紹垃圾回收及其配置選項。

1.1.4 記憶體報告和調試

Scaleform 記憶體系統能夠給分配標上 “stat ID” 標記,以描述所進行的分配的用途。然後可以按堆報告分配的記憶體,細分成 stat ID 類別,或者按 stat ID 報告,細分成堆。此記憶體報告功能主要是通過記憶體和性能分析器 (AMP) 公開的,從 Scaleform 3.2 開始,此 AMP 是作為一個獨立的分析應用程式提供的。有關 AMP 使用方面的詳細資訊,請參閱 AMP User Guide。也可以通過調用 Scaleform::MemoryHeap::MemReport 函數來以程式設計方式獲得字串格式的記憶體報告。

上文所述的 stat ID 標記是作為調試資料與實際的記憶體分配分開存儲的。調試資料也是為檢測記憶體洩漏而保留的。當 Scaleform 調試版關閉時,Visual Studio 輸出視窗或主控台就會生成一份記憶體洩漏報告。由於 Scaleform 沒有任何已知內部記憶體洩漏,檢測到的任何洩漏都很有可能是由於在 Scaleform 上引用計數使用不當造成的。額外的關聯調試資料不在 Shipping 配置中分配。

1.2 Scaleform 3.3 記憶體 API 更改

如前所述,Scaleform 3.3 包括兩個改進記憶體使用情況的重要更新：

- GFx::Movie 記憶體上下文共用特性使獨立電影視圖實例可以共用堆和垃圾回收,因而可以提高記憶體重複利用率。內部字串表也可以共用,因而可以進一步節約記憶體。第 4 節將詳細介紹此共用情況。
- SysAlloc 介面經過更新後更加“對 malloc 友好”,消除了對於外部分配的 4K 頁面對齊要求,並使堆優化為將大於 512 位元組的所有分配都直接路由到 SysAlloc。此方法導致更多珍貴的記憶體迅速返回到應用程式中,因而提高了效率和重複利用率。

請注意,原來的 Scaleform::SysAlloc 實現現已更名為 Scaleform::SysAllocPaged,而且仍然可用。重寫 SysAlloc 的開發者可以選擇更新其實現,或將基類更改為 SysAllocPaged。依賴 SysAllocStatic 和記憶體分配場地的使用者不受影響,因為那個類仍然可用。

2 記憶體分配

如第 1.1.1 節所述,Scaleform 使開發者可以通過選擇以下三種方法之一來自訂記憶體分配：

1. 重寫 SysAlloc 介面,將分配委派給應用程式記憶體系統，
2. 給 Scaleform 提供一個或多個固定大小的區塊,或者
3. 指示 Scaleform 直接从操作系统分配内存。

要想實現最佳記憶體效率,開發者應該選擇一種適合其應用程式的方法。下麵探討最常用的方法 (1) 與方法 (2) 之間的區別。

當開發者重寫 SysAlloc 時,他們將 Scaleform 分配委派給自己的記憶體系統。例如,當 Scaleform 需要使用記憶體來載入新 SWF 檔時,Scaleform 會通過調用 Scaleform::SysAlloc::Alloc 函數來進行此請求;當卸載內容時,它會調用 Scaleform::SysAlloc::Free。在此情況下,當所有系統(包括 Scaleform 在內)都使用單個共用全域分配器時,就可以實現盡可能最大的記憶體效率,因為已釋放的任何區塊均可供需要它的任何系統重複使用。任何區隔化 (compartmentalization) 分配方式均會降低整體共用效率,因而增大總體記憶體佔用大小。下麵的 2.1 節將提供重寫 SysAlloc 的詳細資訊。

全域分配方法的最大障礙是碎片,這就是許多主控台開發者更喜歡使用預先確定的固定記憶體佈局的原因。利用預定記憶體佈局,可以為各個系統分配固定大小的記憶體區域,而其中每個記憶體區域都是在啟動時或水準載入 (level loading) 時確定的。總之,這種方法就是用全域系統的效率換取一種更可預測的記憶體佈局。

如果某個應用程式使用這種固定記憶體方法,開發者就應對 Scaleform 也使用固定大小區塊分配器 – 第 2.2 節將介紹這種配置。為使這種情形更加切合實際,Scaleform 還允許創建輔助記憶體分配場地,或僅限有限時間期間(如顯示“暫停功能表”時)使用的記憶體區域。

2.1 關鍵記憶體區域分配

要更換外部 Scaleform 分配器,開發者可以遵循以下兩個步驟：

1. 創建自己的 SysAlloc 介面執行器，需定義 GetInfo、 Alloc 和 Free 方法。
2. 在 Scaleform 初始化過程中用 GFx::System 構造器提供一個該調度分配器的實例。

Scaleform 提供了一種既可直接使用也可作為參考的預設 Scaleform::SysAllocMalloc 實現,這種實現依賴于標準的 malloc/free 及其針對系統的替代方案(支援校準(alignment)）。

2.1.1 實現自己的 SysAlloc 函數

實現自己的記憶體堆最簡單的方法爲拷貝 Scaleform SysAllocMalloc 並作相應修改以調用自己的記憶體分配器或堆。對 SysAllocMalloc 分配器代碼的修改見以下內容：

```
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，記憶體分配執行相對簡單。MySysAlloc 類從 SysAlloc 基本介面上派生而來並實現了三個虛擬函數：`GetInfo`、 `Alloc` 和 `Free`。儘管實現這些函數必須尊重校準,但 Scaleform 將一般只請求小規模校準,如不超過 16 位元組。oldSize 和 align 參數傳遞到 Free/Realloc 介面之中,以簡化實現;它們有助於(例如)作為一對 Alloc/Free 調用的包裝程式來實現 Realloc。

一旦創建了一個自己的分配器實例，在 Scaleform 初始化期間可以傳遞給 GFx::System 結構。

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

在 Scaleform 3.1/3.0 中，GFx::System 物件需要創建在其他 Scaleform 物件創建之前，在其他 Scaleform 物件釋放之後被釋放。這些步驟通常最好在分配初始化中完成，初始化中的函數調用 Scaleform 中使用的代碼；但是，也可以作爲其他分配物件的一部分，這些物件的生命周期長於 Scaleform 中的物件（GFx::System 不能全局聲明）。如果發現這種用法不方便，可以使用

GFx::System::Init()和 GFx::System::Destroy()靜態函數來代替，不需要創建物件。與 GFx::System 構造器類似，GFx::System::Init()擁有一個 SysAlloc 指標參數。

2.2 將固定內存块用于 Scaleform

如引言中所述,主控台開發者可以選擇提前保留一個或多個區塊,並將它們傳遞到 Scaleform,而不用重寫 SysAlloc。這是通過產生實體 GysAllocStatic 來完成的,如下所示：

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

上例中傳遞一個 6 百萬位元組的存儲塊給 Scaleform 用戶記憶體分配。當然，存儲塊在 `gfxSystem` 和 `blockAlloc` 物件消亡之前不能重用或釋放。不過,可以出於臨時目的(如為了顯示暫停功能表)添加額外的“可釋放”區塊。下麵探討這樣的塊,通過使用記憶體分配場地來為這些塊提供支援。

但是用靜態記憶體分配器時，開發者需要特別留意 `Scaleform` 使用的存儲量，確保沒有導入文件或創建動畫實例以至超出指定的存儲範圍。一旦 SysAllocStatic 失敗，將從 `Alloc` 執行函數返回 0，導致 Scaleform 運行失敗或崩潰。如有需要，開發者可以使用簡單的 SysAllocPaged 封裝物件來檢測這種危險情況。

2.2.1 記憶體實存塊

Scaleform 3.1 介紹了對記憶體實存塊的支援,在程式執行過程中的某個特定點上,能夠保證使用者規定的分配器區域全部被釋放。更具體地來說,記憶體實存塊所定義的記憶體區能夠載入 Scaleform 檔以及創建 GFx::Movies,因此一旦佔用這些區域的 Scaleform 檔被銷毀,這些地區將會全部得到釋放。一旦某一實存塊被銷毀,程式就能夠將所有記憶體重新用於其他非 Scaleform 資料。作為一種可能的使用情況,某一記憶體實存塊可被定義為載入遊戲中的一個“暫停功能表”介面,這將暫時佔用記憶體,但只要遊戲恢復運行狀態,它們將立即被釋放並可重新使用。

2.2.1.1 *背景*

一般情況下,大部分開發商不需要通過 Scaleform 將記憶體實存塊定義為重新使用記憶體佔用。當 Scaleform 分配記憶體時,它將從 GFx::System 所規定的 SysAlloc 物件中獲取;該記憶體將會被釋放,因為您卸載了資料並銷毀了 Scaleform 目標。然而,由於多種原因的存在,被釋放的記憶體模式並不總能與分配

相匹配。舉例來說,如果您調用 CreateMovie,使用它,然後釋放它,很可能出現這樣的情況,在創建調用過程中所分配的大部分區塊能夠被釋放,但其中的某一小部分可能會保持較長的一段時間。導致出現這種情況的原因有很多,包括碎片、動態資料結構、多執行緒、Scaleform 資源分享與緩存等。

在大多數情況下,一部分區塊暫時未被釋放並不是一個問題,因為在 Scaleform 的生命週期中,這些區塊不會累積並被重新使用。然而,如果您的遊戲引擎用作固定大小的記憶體緩衝區,並需要與 Scaleform 和其他遊戲引擎資料進行共用時,不可預測的分配性質倒是能夠造成一定的麻煩。在 SDK 的早期版本中,開發人員可以關閉所有的 Scaleform 以確保緩衝區記憶體能夠完全被釋放。然而,在使用 Scaleform 3.1 時,使用者能夠為這些緩衝區定義記憶體實存塊,並向特定的記憶體實存塊分配 SWF/GFX 檔。一旦這些檔被卸載,Scaleform::Memory::DestroyArena 就能夠被調用,並且所有的實存塊記憶體也能夠安全地重新使用。

2.2.1.2 *使用記憶體實存塊*

為了使用記憶體實存塊,開發人員需要完成以下幾步:

1. 通過調用Scaleform::Memory::CreateArena來創建一個實存塊,並為其賦予一個非零整數識別字(零意味著全球或預設實存塊,總是存在的)。為了進入實存塊,您需要提供一個SysAlloc埠;SysAllocStatic可在固定大小的記憶體緩衝區內使用。

   ```
   // 假定pbuf1代表一個10,000,000位元組的緩衝區。
   SysAllocStatic  sysAlloc(pbuf1, 10000000);
   Memory::CreateArena(1, &sysAlloc);
   ```

2. 調用 GFx::Loader::CreateMovie / GFx::MovieDef::CreateInstance,同時為實存塊設置使用 id。那些視頻物件所需的堆疊將在規定的實存塊內創建,因此所需要的大部分記憶體將會來自這裡。請注意,某些“共用的”全球記憶體仍會被分配,因此,您還需確保在全球範圍內擁有足夠的儲備。

   ```
   // 該視頻使用實存塊1
   Ptr<GFx::MovieDef> pMovieDef =
               *Loader.CreateMovie(filenameStr, Loader::LoadWaitFrame1, 1);
   ...
   //該示例使用實存塊1
   Ptr<GFx::Movie> pMovie =
            *pnewMovieDef->CreateInstance(MovieDef::MemoryParams(1), false);
   ```

3. 必要時使用合成的 GFx::MovieDef/GFx::Movie 物件。

2. 釋放處於實存塊內所有已創建的視頻和物件。如果GFx::ThreadedTaskManager用於執行緒後臺載入,那麼在釋放前應當對GFx::MovieDef::WaitForLoadFinish進行調用,因為它將強迫後臺執行緒終止載入並釋放他們的視頻引用。

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

or (if Ptr is not used):

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

3. 調用 DestroyArena。完成這一操作後,所有實存塊的記憶體將能夠安全地重複使用,直至 CreateArena 被再次調用為止。Scaleform::Memory::ArenaIsEmpty 函數可用於檢查記憶體實存塊是否為空以及是否準備撤銷。

```
 // 如果記憶體在調用前沒有被正確釋放的話,則DestroyArena將會生效。
//   在釋放過程中它將會在"return *(int*)0;" 中崩潰。
 Memory::DestroyArena(1);
```

SysAlloc 目標現在能夠超出範圍或被銷毀，此後 “ pbuff1 ” 可被重新使用。記憶體實存塊能夠在必要時重新被創建。

2.3 將分配委派给 OS

開發者不用重寫分配器,而是可以選擇使用隨 Scaleform 提供的 OS-直接分配器之一。它們包括：

- *Saleform::SysAllocWinAPI* – 使用 VirtualAlloc/VirtualFree API 函數，具有良好的對齊和頁面調度功能，爲在微軟平臺上的最佳選擇；
- *Scaleform::SysAllocPS3* – 使用 sys_memory_allocate/sys_memory_free 調用，用於高效的頁面管理和 PS3 上的對齊功能。

分頁爲作業系統(OS)的功能以重映射實體記憶體到頁面內資料塊的虛擬位址空間。如果和使用智慧化的系統分配，分頁可以在大存儲塊中減少碎片，實現方法爲在小的自由存儲塊上劃分線性記憶體範圍，這些小的自由存儲塊分散在位址空間。我們的 WinAPI 和 PS3 SysAlloc 執行函數利用了映射和取消映射系統記憶體到粒度資料塊的優勢。當然，如果你之前已經保留了系統記憶體且尚未釋放給系統，則無法實現此動態“磁片碎片整理”。
此細微性遠比(例如) dlmalloc 使用的 2MB 塊效率高。

2.3.1 實現 SysAllocPaged

隨著 Scaleform 3.3 中引入一個更新的 SysAlloc 介面,原來的 SysAlloc 實現更名為 SysAllocPaged。Scaleform 中還有使用它的分配器實現,並且用於 SysAllocStatic 和特定 OS 的分配器實現。這些分配器實現包含在庫中,但如果不用就不會被連結。出於相容目的,可以實現 SysAllocPaged,而不是 SysAlloc。請看下文。

對 SysAllocPagedMalloc 分配器代碼的修改見以下內容：

```
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，記憶體分配執行相對簡單。MySysAlloc 類從 SysAlloc 基本介面上派生而來並實現了三個虛擬函數：`GetInfo`、 `Alloc` 和 `Free`。
另外還有一個 `ReallocInPlace` 函數用於優化，但在這裏尚未提到。

1 `GetInfo()` – 返回分配器記憶體對齊支援能力和粒度，在 Scaleform::SysAllocPaged::Info 結構成員中填入這些值。

2 `Alloc` – 分配指定大小記憶體。函數包含了 size 和 align 兩個參數，在 Alloc 執行時需要用到。傳遞的 align 值不能大於從 GetInfo 返回的 MaxAlign 值。由於在我們例子中已將 MaxAlign 設爲 1 位元組，我們可以忽略第二個參數 align。

3 `Free` – 釋放先前 Alloc 分配的記憶體。函數包含 size 和 align 參數，告知原來分配空間大小；在我們的空間釋放實現方式中無需此參數，可以忽略。

4 `ReallocInPlace` – 試圖重復分配記憶體而不轉移到不同的位置，如果不成功則返回 false。很多用戶不需要用到這個函數；請參考 Scaleform 參考資料 獲取詳細資訊。

可以看出，`Alloc` 和 `Free` 函數的都爲標準的操作，所以只有 `GetInfo` 函數比較特殊。
該函數報告了 Scaleform 中分配器的功能，所以可以影響 SysAllocPaged 執行記憶體對齊支援和需求，以及 Scaleform 性能和記憶體使用效率。`SysAllocPaged::Info` 結構包含以下四個值：

- `MinAlign` – 分配器在所有分配中應用的最小記憶體對齊空間。
- `MaxAlign` – 分配器可以支援的最大記憶體對齊空間。如果值爲 0，Scaleform 認爲支援任何對齊空間。如果值爲 1，則不支援任何對齊方式，在此情況下 Alloc 函數執行時可以忽略 align 參數。
- `Granularity` – 爲 Scaleform 分配推薦粒度；Scaleform 中對分配請求空間大小至少爲此大小的粒度。如果分配器如在 malloc 函數應用中不能友好對齊，我們推薦使用最小對齊空間爲 64K。
- `HasRealloc` – 一個布林標誌指定執行器是否支援 `ReallocInPlace`；在本例中返回。
- `SysDirectThreshold`-定義了全球的規模門檻(在不為空的情況下)。如果分配的規模大於或等於 SysDirectThreshold,它將對系統進行重新定向,忽略粒層。
- `MaxHeapGranularity` 在不為空的情況下限制了可能的最大堆細微性。在大多數情況下,MaxHeapGranularity 能夠減少常用部分價格的 alloc/free 操作(該操作將會使分配器變慢)對系統的記憶體佔用。MaxHeapGranularity 必須至少為 4096,並且是 4096 的整數倍。

從 `MaxAlign` 參數的描述中可以看出，如果分配器不支援記憶體對齊，則 SysAlloc 介面實現很容易，但是也需要利用記憶體對齊的優勢。在實際應用中，需要考慮三個方面：

1. *分配器執行不支援記憶體對齊，或處理無效*。如果爲這種情況，設置 `MaxAlign` 爲 1，使 Scaleform 內部完成所有需要的對齊工作。
2. *分配器能有效處理記憶體對齊*。如果爲這種情況，設置 `MaxAlign` 爲 0 或者能支援的最大記憶體對齊空間大小，執行 `Alloc` 函數正確處理記憶體對齊。
3. *分配器爲系統的介面*，頁面大小總是需指定和對齊。爲有效處理，設置 `MinAlign` 和 `MaxAlign` 爲系統頁大小值並將所有的分配空間值傳遞給作業系統。

一旦創建了一個自己的分配器實例，在 Scaleform 初始化期間可以傳遞給 GFx::System 結構。

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

2.4 存儲堆

前面已經提到,Scaleform 在稱為“堆”的記憶體池中維護記憶體。每個堆都是出於特定目的創建的,例如:為了保存從像網格緩存 (Mesh Cache) 這樣的特定檔子系統載入的資料。本節概述用來在 Scaleform 核心內管理堆記憶體的 API。

2.4.1 堆 API 函數

爲便於用戶設置，大多數記憶體堆管理的詳細內容隱藏在代碼內部。
開發者可以在所有的 Scaleform 物件上直接使用“new”操作符，將執行堆記憶體分配。

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

可以看到，操作符‘new’的使用沒有任何堆描述資訊，所以記憶體分配將發生在 Scaleform 全部堆內部。
因爲配置物件的數量很少，這個方法可以很好得工作。

全域堆是在初始化 GFx::System 期間創建的,並保持活動狀態,直到關閉 GFx::System。它始終使用記憶體分配場地 0。

然而，可以在內部使用記憶體堆控制記憶體碎片。例如，GFx::MeshCacheManager 類創建一個內部堆保存和約束所有柵格形狀資料。`GFx::Loader::CreateMovie()`創建一個堆保存從特定 SWF/GFX 文件導入的資料。`GFx::MovieDef::CreateInstance` 創建一個堆來保存時間軸和 ActionScript 運行狀態資訊。所有這些詳細資訊位於後臺，不應該對大多數用戶產生影響，至少在你計劃改變、擴展或調試 Scaleform 代碼之前不應改產生影響。

如前面提到，存儲堆由 Scaleform::MemoryHeap 類表示，伴隨爲特定 Scaleform 子系統或資料設置產生的實例。每個子存儲堆使用 Scaleform::MemoryHeap::CreateHeap 在全局堆上創建調用 `Scaleform::MemoryHeap::Release` 銷毀。在堆的生命周期內，可以調用 `Scaleform::MemoryHeap::Alloc` 函數分配記憶體和 `Scaleform::MemoryHeap::Free` 函數釋放記憶

體。當堆被銷毀後，所有的內部存儲被釋放。鑒於堆的空間容量，這些記憶體通常可以通過 SysAlloc 介面立即釋放給遊戲應用程式。

爲確保記憶體中正確的堆分配，Scaleform 中的大多數分配使用特別的宏包含一個 Scaleform::MemoryHeap 指標。這些宏包括：

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

巨集的使用確保正確的字元行和檔案名在調試編譯時用戶分配記憶體，‘id’號用來標記分配用途，這些識別字在 Scaleform AMP 記憶體分析系統中分成組。

設定記憶體分配巨集，一個新的 GFx::SpriteDef 物件可以由一個自定義堆名‘pHeap’創建，調用形式如下所示：

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

這裏，GFx::SpriteDef 爲在 Scaleform 中使用的一個內部類。由於 Scaleform 設計了大多數物件，通常在指標物件 Ptr<>中描述，當這些指標超出範圍，調用內部釋放函數進行銷毀。對於不包括在內的物件，如從 Scaleform::NewOverrideBase 繼承來的物件，可直接使用‘delete’操作符。沒有必要用指定堆來釋放記憶體，delete 操作和 SF_FREE（位址）宏可以自動找出堆。

2.4.1 自動堆

儘管記憶體分配巨集提供了所有必要的功能來支援堆，但是有時用起來需要更多消耗，因爲巨集需要一個堆指標傳遞給程式。在用來聚集容器物件需要記憶體分配時這些指標傳遞將變得非常麻煩，如 Scaleform::String 或 Scaleform::Array<>。見下面實例代碼聲明一個類並在堆中創建一個實例：

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

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

儘管 GFx::Sprite 物件本身創建於指定堆，包含了一個字串和一個陣列物件需要被動態分配。如果沒有特殊考慮，這些物件將在全局堆中結束分配 - 這種情況可能性最大，但不是理想中的情況。爲了解決這個問題，一個堆指標理論上可以被傳遞到 GFx::Sprite 物件結構體然後進入字串和陣列結構體。這些參數傳遞將使編程變得更爲困難，潛在需要很多物件，如陣列和字串來維持一個指標，該指標指向堆和其他地方消耗的記憶體。

爲了使該情況下的變成更加容易，Scaleform 內核使用了特殊的“自動堆”分配宏：

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

這些宏的行爲與同屬的 SF_HEAP_ALLOC 和 SF_HEAP_NEW 非常類似，但是有一個明顯的區別：他們擁有一個指標指向一個記憶體分配而不是指向堆。當被調用時，這些宏將自動辨別建立在所提供的記憶體位址上的堆並從相同的堆中分配記憶體。見以下例子：

```
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);
```

在本例中我們從指定堆創建一個緩衝記憶體以及在相同堆上創建一個 GFx::Sprite 物件作爲緩衝記憶體。本例中特有的方法爲不將‘pbuffer’指標傳遞到 SF_HEAP_AUTO_NEW；而傳遞一個緩存分配範圍內的地址。Scaleform 存儲堆系統，具有辨識一個存儲堆的功能，存儲堆可以爲一個分配記憶體上的任何位址（這樣做更加高效）。這種獨特的方法爲解決堆傳遞問題的關鍵，這些問題在本章開頭已有描述。配置了自動堆辨識，我們可以創建容器從正確位置的堆中自動分配記憶體，也即我們可以重寫前面提到的 GFx::Sprite 類，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();
```

這裏，我們已將位於相同本地堆中 Scaleform::String 和 Scaleform::Array<>類替換爲相等的“本地堆” Scaleform::StringLH 和 Scaleform::ArrayLH<>。這些本地堆容器從包含相同物件的堆分配記憶體，使用了前面提到的“自動堆”技術，這個過程不需要參數傳遞，參數傳遞在其他地方是必須的。你可以發現這些和類似的容器在 Scaleform 內核中到處被使用，必須確保記憶體在正確的堆中結束。

2.4.1 堆跟蹤程式

使用者可以通過啟用 GFxConfig.h 中定義的 SF_MEMORY_TRACE_ALL 來跟蹤基本記憶體分配情況。通過調用 Scaleform::Memory 類中的 SetTracer 方法,可以跟蹤 CreateHeap、DestroyHeap、Alloc、Free 等記憶體操作。

下麵提供在 SysAllocWinAPI 的情況下如何使用跟蹤程式的一個示例代碼：

```
#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 記憶體報告

儘管 Scaleform 3.0 原來引入詳細的記憶體報告,作為 Scaleform Player AMP HUD 的組成部分,但現在已經將分析功能遷移到一個獨立的 AMP 應用程式之中,該應用程式可遠端連接到 PC 和主控台 Scaleform 應用程式。有關 AMP 使用方面的詳細資訊,請參閱 AMP User Guide。

儘管為了使 Scaleform Player 更加輕便,已從其中刪除了 HUD UI,但仍然可以生成記憶體報告並通過按 Ctrl + F5 鍵發送到主控台。此報告是 Scaleform::MemoryHeap::MemReport 函數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 生成一份內存報告，可能的格式是 XML，寫到一個提供的字符串緩衝區。
然後可以將此字串寫到輸出主控台或用於調試的螢幕。MemReportType 參數可以是下列情況之一：

- MemoryHeap::MemReportBrief – Scaleform 系統所消耗總記憶體的摘要,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 – 此記憶體類型為按 Ctrl-F6 時 Scaleform Player 輸出的記憶體類型。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 此外,每個堆段內都有已分配記憶體的明細資訊(按 stat ID)。下麵顯示一個按堆給出的資訊的示例：

```
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 – 此記憶體類型生成按檔顯示的消耗的記憶體。該報告按 stat ID 將每個 SWF 檔的 MovieData、MovieDef 和所有 MovieView 的值合計在一起。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 -此內存類型是 AMP 在內存選項卡中顯示的內存類型，而且是最詳細的報告，顯示每個堆使用多大內存、多大內存用於調試信息以及 Scaleform 已申請但尚未使用的內存大小。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
```

上面已經提到，当使用 SF_MEMORY_ENABLE_DEBUG_INFO 编译 Scaleform 时,stat ID 標記描述上述分配的用途。下麵簡述其中一些 stat ID 類別：

- MovieDef –代表已載入的檔資料。完成檔載入後此記憶體不應繼續增大。
- MovieView –包含 GFxMovieView 實例資料。如果 ActionScript 正在分配許多物件,或者目前存在其他物件的許多電影剪輯,此類別的大小就會增加。
- CharDefs –各個電影剪輯或其他 Flash 物件的定義。
- ShapeData –複雜形狀和字體的向量表示。
- Tags –動畫關鍵幀資料。
- ActionOps – ActionScript 位元組碼。
- MeshCache –包含向量細分單元 (Vector Tesselator) 和 EdgeAA 生成的緩存形狀網格資料。此類別隨新的形狀被細化而不斷增大,最終達到一個極限。
- FontCache –包含緩存的字體資料。

4 垃圾回收

Scaleform 版本 2.2 及早期版本使用一個簡單的索引計數機制來處理 ActionScript 物件。大多數情況下這已夠用。但是，在兩個或更多物件互相關聯，ActionScript 允許你創建稱之爲“circular references”或“cycle references”物件。這個迴圈引用造成了記憶體泄漏影響系統性能。常見下面一個例子的代碼，將產生泄漏，除非其中有一個物件索引被明確分離開。

Code:

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

理論上，可以重寫這些代碼以避免迴圈引用或者使用一個清理函數將物件從迴圈引用分離出來。在多數情況下，清理函數不能起到作用，如果使用了 ActionScript 類和元件，反而問題會變得更加糟糕。常見情況下單獨使用會導致記憶體泄漏，使用標準 Flash UI 元件也是如此。

爲解決迴圈引用的問題，Scaleform 3.0 中引入了一個高度優化的引用計數器 - 基於清除機制稱爲“垃圾回收”。Scaleform 垃圾回收與通常的垃圾回收機制類似；但是，對其進行了優化用於引用計數。如果用戶的 ActionScript 不創建“迴圈引用”，該機制作爲常規應用計數系統。在迴圈引用被創建的情況下，才執行回收函數並在需要時釋放這些物件。對於大多數 Flash 文件性能影響很小。

默認情況下，垃圾回收在 Scaleform 3.1/3.0 中爲有效。當使用了垃圾回收機制，所有先前迴圈引用導致的的記憶體泄漏將被消除。如果不需要，垃圾回收功能可以禁止。但是，只有擁有源代碼的用戶有此許可權，因爲需要重新編譯 Scaleform。

如要關閉垃圾回收機制，必須開打文件 *Include/GFxConfig.h* 取消 GFX_AS_ENABLE_GC 巨集（默認情況下爲注釋掉的）：

```
// 禁止垃圾回收
#define GFX_AS_ENABLE_GC
```

在取消宏後，必須完全重新編譯 Scaleform。

4.1 配置可用於垃圾回收的記憶體堆

Scaleform 3.x 允許配置記憶體堆支援垃圾回收機制。Scaleform 3.3 和更高版本都允許進行堆共用, 因而可以在多個 MovieView 之間實現垃圾回收器共用。

要為給定 MovieDef 創建一個新堆,可以使用如下函數：

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

MemoryParams 結構體爲 GFx::MovieDef 類中的一個嵌套類，如下所示：

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

- Desc – 爲動畫所創建堆的描述符。可以指定堆對齊方式（MinAlign）、粒度(Granularity)、限制範圍(Limit)和標誌(Flags)。如果指定了堆的範圍，則 Scaleform 將堆的使用限制在這個範圍之內。
- HeapLimitMultiplier – 一個多點堆限制乘法器（0...1），用來確定超出限定範圍後堆的增加方式。見下面內容獲取詳細資訊。
- MaxCollectionRoots – 執行垃圾回收程式前 roots 數。該值爲最大 roots 的初始值；在執行中可以增加。見下面內容獲取詳細資訊。
- FramesBetweenCollections – 執行完成一定數量的幀後強制調用回收函數，甚至尚爲達到空間上限。在當前最大 roots 值較高，更好得用於執行回收功能，以減少回收發生時的消耗。見下面內容獲取詳細資訊。

同樣地,可用兩個步驟來創建一個 Movie View 堆：

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

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

MemoryContext 物件封裝 MovieView 堆以及其他特定堆的物件,如垃圾回收器。這通過記憶體上下文創建 MovieView 及其堆的第二種方法增強了指定 AMP 顯示的堆名的靈活性,無論是否需要執行緒安全,以及是否將該堆標記為調試並因此被排除在 AMP 報告之外。更重要的是,它使同一線程上的多個 MovieView 可以共用單個堆、垃圾回收器、字串管理器以及文本分配器,從而減少了額外開銷。

`MemoryParams::Desc` 被用來指定記憶體堆的常規屬性，這些堆用於指定記憶體分配的實例（例如，ActionScript 分配）爲控制堆的範圍可以設置兩個參數：`Desc.Limit` 和 `HeapLimitMultiplier`。

堆已經初始化爲預設範圍爲 `128K`(所以稱之爲動態分配）。當超過這個範圍，將調用一個特殊的內部控制碼。這個控制碼可以從邏輯上進行判斷：釋放空間或者擴展堆。採用何種方式需要根據 Boehm-Demers-Weiser (BDW)演算法垃圾回收器和記憶體分配器來做出判斷。

BDW 演算法如下（僞代碼）：

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

步驟“collect”包括了 ActionScript 垃圾回收器標識和其他記憶體釋放，如內部緩存刷新。

`HeapLimitMultiplier` 的預設值爲 `0.25`。因此，由於最終的記憶體佔用大於當前堆的 25%（HeapLimitMultiplier 預設值），則 `Scaleform` 將執行記憶體釋放。否則，將擴展限制範圍 25%加上請求增加的堆空間。

如果用戶指定了 `Desc.Limit`，然後上述演算法根據指定的限制範圍按照相同的方式執行。如果堆限制範圍超出 `Desc.Limit` 設定的值，則調用回收函數，無需考慮最終回收的分配空間量。動態堆限制範圍將設置爲堆的空間加上所需的分配請求範圍（該情況下，如果釋放的空間少於分配所需）。

第二種控制垃圾回收行爲的方法爲指定 `MaxCollectionRoots/FramesBetweenCollections`。`MaxCollectionRoots` 指定了 roots 數量，超過該數量則調用垃圾回收。這裏的“root”意思爲任何 ActionScript 物件可能與其他 ActionScript 物件產生迴圈引用的 ActionScript 物件。通常，一個 ActionScript 物件一旦與另外的 ActionScript 物件（意爲 ActionScript 引用該物件，例如“obj.member = 1”涉及到物件“obj”）相關聯則添加到 roots 陣列。基本上，我們將每個涉及的 ActionScript 物件作爲 root，因此，`MaxCollectionRoots` 指定當前與 ActionScript 協同工作的物件大概數量。因此，當涉及物件數量超過 `MaxCollectionRoots` 指定值時將調用垃圾回收。默認情況下，該參數設置爲 `0` 以關閉這項機制。

`FramesBetweenCollections` 指定了幀數量，播完這些幀之後強制調用垃圾回收函數。這裏的條目“frame”表示一個 Flash 幀。因此，如果一個 SWF 文件幀速率爲 30 fps，則一秒鐘播放 30 幀，2 秒種

爲 60 幀，依次類推。這個值用來避免在 Advance 操作中降低性能，當垃圾回收需要遍曆許多物件時導致性能下降。本例中，這個值可以設置爲如 1800，垃圾回收每 60 秒鐘調用一次，幀速率爲 30 fps。`FramesBetweenCollections` 參數可以避免當 `Desc.Limit` 未被設置情況下 ActionScript 記憶體堆過多增長。默認情況下，該參數設置爲 0，表示關閉此項功能。

4.2 GFx::Movie:: ForceCollectGarbage

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

本方法可以用來讓應用程式執行強制垃圾回收程式。如果垃圾回收功能爲 off 不產生任何效果。