

Autodesk® Scaleform®

Scaleform 遊戲通訊概述

本檔描述的是利用 Scaleform 3.1 及其更新版本在 C++、Flash 和 ActionScript 之間的通訊機制。

作者： Mustafa Thamer

版本： 2.01

最後修訂日期： 2010 年 9 月 27 日

Copyright Notice

Autodesk® Scaleform® 4.3

© 2013 Autodesk, Inc. All rights reserved. Except as otherwise permitted by Autodesk, Inc., this publication, or parts thereof, may not be reproduced in any form, by any method, for any purpose.

Certain materials included in this publication are reprinted with the permission of the copyright holder.

The following are registered trademarks or trademarks of Autodesk, Inc., and/or its subsidiaries and/or affiliates in the USA and other countries: 123D, 3ds Max, Algor, Alias, AliasStudio, ATC, AutoCAD, AutoCAD Learning Assistance, AutoCAD LT, AutoCAD Simulator, AutoCAD SQL Extension, AutoCAD SQL Interface, Autodesk, Autodesk 123D, Autodesk Homestyler, Autodesk Intent, Autodesk Inventor, Autodesk MapGuide, Autodesk Streamline, AutoLISP, AutoSketch, AutoSnap, AutoTrack, Backburner, Backdraft, Beast, Beast (design/logo), BIM 360, Built with ObjectARX (design/logo), Burn, Buzzsaw, CADmep, CAiCE, CAMduct, CFdesign, Civil 3D, Cleaner, Cleaner Central, ClearScale, Colour Warper, Combustion, Communication Specification, Constructware, Content Explorer, Creative Bridge, Dancing Baby (image), DesignCenter, Design Doctor, Designer's Toolkit, DesignKids, DesignProf, Design Server, DesignStudio, Design Web Format, Discreet, DWF, DWG, DWG (design/logo), DWG Extreme, DWG TrueConvert, DWG TrueView, DWGX, DXF, Ecotect, ESTmep, Evolver, Exposure, Extending the Design Team, FABmep, Face Robot, FBX, Fempro, Fire, Flame, Flare, Flint, FMDesktop, ForceEffect, Freewheel, GDX Driver, Glue, Green Building Studio, Heads-up Design, Heidi, Homestyler, HumanIK, i-drop, ImageModeler, iMOUT, Incinerator, Inferno, Instructables, Instructables (stylized robot design/logo), Inventor, Inventor LT, Kynapse, Kynogon, LandXplorer, Lustre, Map It, Build It, Use It, MatchMover, Maya, Mechanical Desktop, MIMI, Moldflow, Moldflow Plastics Advisers, Moldflow Plastics Insight, Moondust, MotionBuilder, Movimento, MPA, MPA (design/logo), MPI (design/logo), MPX, MPX (design/logo), Mudbox, Multi-Master Editing, Navisworks, ObjectARX, ObjectDBX, Opticore, Pipeplus, Pixlr, Pixlr-o-matic, PolarSnap, Powered with Autodesk Technology, Productstream, ProMaterials, RasterDWG, RealDWG, Real-time Roto, Recognize, Render Queue, Retimer, Reveal, Revit, Revit LT, RiverCAD, Robot, Scaleform, Scaleform GFx, Showcase, Show Me, ShowMotion, SketchBook, Smoke, Softimage, Socialcam, Sparks, SteeringWheels, Stitcher, Stone, StormNET, TinkerBox, ToolClip, Topobase, Toxik, TrustedDWG, T-Splines, U-Vis, ViewCube, Visual, Visual LISP, Vtour, WaterNetworks, Wire, Wiretap, WiretapCentral, XSI.

All other brand names, product names or trademarks belong to their respective holders.

Disclaimer

THIS PUBLICATION AND THE INFORMATION CONTAINED HEREIN IS MADE AVAILABLE BY AUTODESK, INC. "AS IS." AUTODESK, INC. DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE REGARDING THESE MATERIALS.

如何與 Autodesk Scaleform 聯繫：

檔案名稱	Scaleform 遊戲通訊概述
地址	美國格林貝爾特 MD 20770 常春藤路 6305 號 310 室
網站	Scaleform 公司 www.scaleform.com
電子郵件	info@scaleform.com
直線電話	(301) 446-3200
傳真	(301) 446-3199

目 錄

1 C++、Flash 和 ActionScript 建立介面.....	1
1.1 ActionScript 到 C++.....	2
1.1.1 FSCCommand 回調.....	2
1.1.2 外部介面 API.....	3
1.2 C++到 ActionScript.....	7
1.2.1 操縱 ActionScript 變數	7
1.2.2 執行 ActionScript 副程式.....	8
1.2.3 路徑	10
2 Direct Access API.....	12
2.1 物件與陣列.....	13
2.2 顯示物件	14
2.3 函數物件	14
2.4 Direct Access 公共介面	17
2.4.1 Object 支持	17
2.4.2 Array 支持	18
2.4.3 顯示物件支援.....	19
2.4.4 影片剪輯支援.....	19
2.4.5 文本欄位支援.....	20

1 C++、Flash 和 ActionScript 建立介面

Flash®的 ActionScript™ 腳本語言能夠創建互動式視頻內容。諸如點擊按鈕、達到某一幀或載入某一視頻等事件都能動態改變視頻內容，控制視頻流，甚至會啟動更多視頻。 ActionScript 完全有能力在 Flash 中創建完整的迷你遊戲。與大多數編程語言一樣， ActionScript 支援變數和副程式，並包括代表專案或控制的物件。

程式與 Flash 之間的通訊需要用複雜的例子來說明。Autodesk Scaleform®支援 Flash 提供的標準機制，它能夠讓 ActionScript 傳遞事件，並將資料返回到 C++ 程式。它還提供了一個便捷的 C++ 介面，可直接操縱 ActionScript 變數、陣列和物件，並直接調用 ActionScript 副程式。

在本檔中，我們討論了 C++ 與 Flash 之間通訊的不同機制。下面列出了可用選項：

ActionScript → C++	C++ → ActionScript
FSCommand 簡單，基於字串的函數執行，沒有返回值，不推薦	GFx::Movie::Get/SetValue 在 ActionScript 獲取資料，使用字串路徑
ExternalInterface 靈活的參數處理，支援返回值，推薦	GFx::Movie::Invoke 在 ActionScript 中調用函數，使用字串路徑
Direct Access API Uses 使用 GFx::Value 作為資料和函數獲取物件的直接參數，性能高	

1.1 ActionScript 到 C++

Scaleform 為 C++ 程式從 ActionScript 接收事件提供了兩種機制：*FSCommand* 和 *ExternalInterface*。這些機制是標準 ActionScript API 的一部分，有關這部分內容的更多資訊可參閱 Flash 檔。*FSCommand* 和 *ExternalInterface* 均通過 Scaleform 註冊了 C++ 事件處理器來接收事件通知。這些處理器按照 Scaleform 狀態進行註冊，因此能夠在 *GFx::Loader*、*GFx::MovieDef* 或 *GFx::Movie* 中進行註冊，具體情況取決於函數要求。有關 Scaleform 狀態的資訊，請參閱 [Scaleform documentation](#).

FSCommand 事件通過 ActionScript 的 'fscommand' 函數來觸發。*fscommand* 函數只能從 ActionScript 傳遞兩個字串參數，一個用於命令，另外一個用於單個資料參數。這些字串的數值被 C++ 處理器接收，並作為兩個常數字串指標。不幸的是，C++ *fscommand* 處理器無法返回數值，因為 ActionScript *fscommand* 函數不具有返回值支援。

ExternalInterface 事件會在 ActionScript 調用 *flash.external.ExternalInterface.call* 函數時觸發。這一介面能夠傳遞 ActionScript 值的任意列表和命令名稱。C++ *ExternalInterface* 處理器接收一個常量字元指標用作命令名稱和運行時所傳遞的與 ActionScript 值對應的 *GFx::Value* 參數陣列。C++ *ExternalInterface* 處理器還能返回一個 *GFx::Value*，因為 ActionScript *ExternalInterface.call* 方法支援返回值。ActionScript *ExternalInterface* 類是外部 API 的一部分，一個應用程式介面，在使用 Scaleform 的程式中，它能在 ActionScript 與 Flash 播放器容器之間實現直接通訊。

由於受到靈活性的限制，*FSCommands* 因 *ExternalInterface* 而變得過時，並且不再推薦使用。在這裏進行描述是為了完整性和傳統性支援的需要。

在一定程度上，**Direct Access** API 與 *GFx::FunctionHandler* 介面會使 *FSCommands* 和 *ExternalInterface* 變得過時。該介面允許將 ActionScript VM 中分配的 C++ 方法按照正常的函數直接進行回調。當 ActionScript 中的函數被調用時，它會反過來調用 C++ 回調。有關 *GFx::FunctionHandler* 的更多資訊，請參閱 [Direct Access API](#) 部分內容。

1.1.1 FSCommand 回調

ActionScript *fscommand* 函數向主機應用程式傳遞命令和資料參數。下面是在 ActionScript 中的典型用法：

```
fscommand("setMode", "2");
```

向 *fscommand* 傳遞的任何非字串參數，例如布林值或整數都將轉換成字串。

ActionScript *fscommand* 調用 會向 *GFx FSCommand* 處理器傳遞兩個字串。應用程式會通過子類 *GFx::FSCommandHandler* 註冊一個 *fscommand* 處理器，並為這一類註冊一個示例作為 *GFx::Loader*、*GFx::MovieDef* 或單個 *GFx::Movie* 物件的共用狀態。請牢記，狀態包之間的設置是為了做出如下委託：

GFx::Loader -> GFx::MovieDef -> GFx::Movie。他們可在後續的任何示例中進行覆蓋。這意味著，如果你想獲得一個擁有自己的 GFx::RenderConfig（例如，對於單獨的 EdgeAA 控制）或 GFx::Translator（以便能夠翻譯成不同的語言），你可在那個物件中進行設置，並且無論應應用什麼狀態，在委託鏈中都會存在一個高於該物件所用的狀態，例如 GFx::Loader。如果在 GFx::Movie 中設置了一個命令處理器，它只能在那個視頻示例中收到 FSCommand 調用的回調。 GFxPlayerTiny 示例說明了這一過程 (查找“FxPlayerFSCommandHandler”)。

下面是 fscommand 處理器設置的例子。該處理器源於 GFx::FSCommandHandler。

```
class OurFSCommandHandler : public FSCommandHandler
{
public:
    virtual void Callback(Movie* pmovie, const char* pcommand,
                          const char* parg)
    {
        printf("FSCommand: %s, Args: %s", pcommand, parg);
    }
};
```

回調方法收到兩個傳遞到 ActionScript 的 fscommand 中的字串參數，同時作為這一具體視頻示例調用 fscommand 的指標。我們的自定義處理器只是簡單地將每一個 fscommand 事件列印到調試控制臺上。

接下來，在創建 GFx::Loader 物件後註冊處理器：

```
// Register our fscommand handler
Ptr<FSCommandHandler> pcommandHandler = *new OurFSCommandHandler;
gfxLoader->SetFSCommandHandler(pcommandHandler);
```

通過 GFx::Loader 註冊處理器會導致每一個 GFx::Movie 和 GFx::MovieDef 都能繼承這一處理器。

SetFSCommandHandler 可在某一視頻示例中進行調用，以覆蓋這一默認設置。

一般來說，C++ 事件處理器應當是無鎖定的，並且會儘快返回調用程式。事件處理器通常只能在 Advance 或 Invoke calls 中進行調用。

1.1.2 外部介面 API

Flash ExternalInterface 調用方法與 fscommand 類似，但它是首選方法，因為它能提供更加靈活的參數處理，並且能夠返回數值。註冊一個 ExternalInterface 處理器與註冊一個 fscommand 處理器類似。這裏舉一個 ExternalInterface 處理器的例子，它能列印數位和字串參數。

```

class OurExternalInterfaceHandler : public ExternalInterface
{
public:
    virtual void Callback(Movie* pmovieView, const char* methodName,
                          const Value* args, unsigned argCount)
    {
        Printf("ExternalInterface: %s, %d args: ", methodName, argCount);
        for(unsigned i = 0; i < argCount; i++)
        {
            switch(args[i].GetType())
            {
                case Value::VT_Number:
                    Printf("%3.3f", args[i].GetNumber());
                    break;
                case Value::VT_String:
                    Printf("%s", args[i].GetString());
                    break;
            }
            Printf("%s", (i == argCount - 1) ? "" : ", ");
        }
        Printf("\n");
    }

    // return a value of 100
    Value retValue;
    retValue.SetNumber(100);
    pmovieView->SetExternalInterfaceRetVal(retValue);
}
};

```

一旦處理器開始執行，它的一個實例將會按照 GFx::Loader 進行如下登記：

```

Ptr<ExternalInterface> pEIHandler = *new OurExternalInterfaceHandler;
gfxLoader.SetExternalInterface(pEIHandler);

```

現在，回調處理器可通過 ActionScript 中的 ExternalInterface 調用而觸發。

1.1.2.1 FxDelegate

上述 ExternalInterface 處理器提供了非常基礎的回調功能。在實際應用中，每當 ExternalInterface 回調通過某一特定 methodName 被觸發時，應用程式都可能會註冊具體的調用函數。這需要一個更加先進的回調系統，它允許函數按照相應的 methodName 進行註冊，然後使用哈希表（或類似的東西）進行查找，

並利用相應的 GFx::Value 參數執行他們。我們在示例中提供了這樣的系統，並被稱為 FxDelegate (參見 Apps\Samples\GameDelegate\FxGameDelegate.h)。

FxDelegate 來自 GFx::ExternalInterface。 它擁有函數註冊/登出處理器，允許應用程式安裝自己的回調處理器，並通過 methodName 進行註冊。

關於 FxDelegate 的用法示例，請參閱我們的 [HUD Kit](#)，它使用 FxDelegate 實施它的 minimap。這裏有一個來自 minimap 演示的程式編碼，它利用 FxDelegate 註冊了自己的回調函數：

```
// Minimap Begin
void FxPlayerApp::Accept(FxDelegateHandler::CallbackProcessor* cbreg)
{
    cbreg->Process("registerMiniMapView", FxPlayerApp::RegisterMiniMapView);

    cbreg->Process("enableSimulation", FxPlayerApp::EnableSimulation);
    cbreg->Process("changeMode", FxPlayerApp::ChangeMode);
    cbreg->Process("captureStats", FxPlayerApp::CaptureStats);

    cbreg->Process("loadSettings", FxPlayerApp::LoadSettings);
    cbreg->Process("numFriendliesChange", FxPlayerApp::NumFriendliesChange);
    cbreg->Process("numEnemiesChange", FxPlayerApp::NumEnemiesChange);
    cbreg->Process("numFlagsChange", FxPlayerApp::NumFlagsChange);
    cbreg->Process("numObjectivesChange", FxPlayerApp::NumObjectivesChange);
    cbreg->Process("numWaypointsChange", FxPlayerApp::NumWaypointsChange);
    cbreg->Process("playerMovementChange", FxPlayerApp::PlayerMovementChange);
    cbreg->Process("botMovementChange", FxPlayerApp::BotMovementChange);

    cbreg->Process("showingUI", FxPlayerApp::ShowingUI);
}
```

FxDelegate 具有全部功能，並作為一個更加複雜的回調處理和註冊系統示例。它應當成為用戶的出發點。我們鼓勵開發者去查看一下它的工作細節，並為了他們自己的需要而進行自定義和擴展。

1.1.2.2 使用 ActionScript 的外部介面

在 ActionScript 中，外部介面調用將通過下列代碼進行初始化：

```
import flash.external.*;
ExternalInterface.call("foo", arg);
```

這裏的“foo”是方法名稱，“arg”為基本類型的參數。你還可以規定 0 或更多參數，以逗號隔開。有關 ExternalInterface API 的更多資訊，請參閱 Flash 檔。

如果 ExternalInterface 沒有輸入上述代碼，ExternalInterface 調用必須完全合格：

```
flash.external.ExternalInterface.call("foo",arg)
```

1.1.2.3 ExternalInterface.addCallback

ExternalInterface.addCallback 函數通常會使用別名註冊一個 ActionScript 方法。這允許從 C++ 中調用註冊的方法，而無需路徑首碼。它支援在單一別名下註冊方法和調用內容。註冊別名可被 C++ GFx::Movie::Invoke() 方法進行調用。

示例：

AS Code:

```
import flash.external.*;

var txtField:TextField = this.createTextField("txtField",
                                              this.getNextHighestDepth(), 0, 0, 200, 50);

var aliasName:String = "setText";
var instance:Object = txtField;
var method:Function = SetText;
var wasSuccessful:Boolean = ExternalInterface.addCallback(aliasName, instance,
                                                          method);

function SetText()
{
    trace(this + ".SetText ");
    this.text = "INVOKED!";
}
```

現在，通過調用別名就可能調用 SetText ActionScript 函數將 "this" 在 txtField 中進行設置：

C++ Code:

```
pMovie->Invoke("setText", "");
```

這一結果將會跟蹤 "txtField.SetText"，並且將文本 "INVOKED!" 設置到 "txtField" 文本欄位中。有關使用 GFx::Movie::Invoke() 的更多資訊，請參閱第 1.2.2 部分內容。.

1.2 C++ 到 ActionScript

前面章節解釋了 ActionScript 如何調用 C++。本節內容將描述如何利用能夠在 C++ 與 Flash 視頻之間進行通訊的 Scaleform 函數實現相反方向的通訊。Scaleform 支援 C++ 函數直接獲取和設置 ActionScript 變數（簡單類型、複雜類型和陣列），同時調用 ActionScript 副程式。

Direct Access API 為訪問和操縱來自 C++ 的 ActionScript 變數提供了更加便捷和有效的介面。有關這一介面的更多資訊，請參閱 [Direct Access API](#) 部分內容。

1.2.1 操縱 ActionScript 變數

Scaleform 支持 [GetVariable](#) 和 [SetVariable](#)，它們能夠直接操縱 ActionScript 變數。對於執行關鍵用法的示例，請參閱 Direct Access API 第 2 部分關於修改變數和物件屬性的更有效方法的內容。儘管出於性能原因不推薦使用 Set/GetVariable，但在某些情況下，它們比使用 Direct Access API 更便捷。例如，沒有必要使用 Direct Access call GFx::Value::SetMember 在該程式的生命期內設置一個單一數值，尤其是如果該目標處於深嵌套層面中時。還有一種情況，通常通過 GetVariable 從 ActionScript 物件中獲取參數，調用一次就可獲得一個 GFx::Value 參數，該參數隨後將用於 Direct Access 操作。

下列示例說明了利用 GetVariable 和 SetVariable 使 ActionScript 計數器的計數開始遞增：

```
int counter = (int)pHUDMovie ->GetVariableDouble("_root.counter");
counter++;
pHUDMovie->SetVariable("_root.counter", GFx::Value((double)counter));
```

GetVariableDouble 返回的 _root.counter 變數值將會自動轉換為 C++ Double 類型。最初，這一變數並不存在，因此 GetVariableDouble 的返回值為 0. 計數器的計數會在下一行增加，新值通過 SetVariable 存儲到 _root.counter 中。[GFx::Movie](#) 的線上檔列出了 GetVariable 與 SetVariable 之間的不同。

SetVariable 具有一個可選的第三參數，其類型為 GFx::Movie::SetVarType，它能聲明分配“粘性”。當所分配的變數還沒有在 Flash 時間線中創建時，這個功能會很有用。例如，t 假定文本欄位 _root.mytextfield 在該視頻的第三幀之前並沒有創建，如果 SetVariable("_root.mytextfield.text", "testing", SV_Normal) 在第 1 幀中進行調用，當該視頻剛剛被創建後，這種分配是無效的。然而，如果通過 SV_Sticky (預設值) 來調用，那麼這一請求將進行排隊，一旦 _root.mytextfield.text 值在第 3 幀開始生效，它就可以執行了。這使得 C++ 中的視頻初始化變得更加容易。請牢記，通常來說，SV_Normal 比 SV_Sticky 更有效，因此，在可能的情況下應當使用 SV_Normal。

為了獲取資料陣列，Scaleform 提供了函數 [SetVariableArray](#) 和 [GetVariableArray](#)。

`SetVariableArray` 將規定類型的資料項目設置到規定範圍的陣列元素中。如果該陣列並不存在，它就會進行創建。如果陣列已經存在，但包含的專案不夠，它就會適當地擴容。然而，當設置的一些元素小於當前陣列的大小時，並不會改變陣列的大小。`GFx::Movie::SetVariableArraySize` 設置陣列的大小，當在現有陣列中所設置的元素少於之前的元素時，這一函數是有用的。

下列示例說明了 `SetVariableArray` 在 AS 中設置字串陣列的用法：

```
int idxInASArray = 0;
const char* strarr[2];
strarr[0] = "This is the first string";
strarr[1] = "This is the second one";
pMovie->SetVariableArray(Movie::SA_String, "_root.strArray",
                           idxInASArray, strarr, 2);
```

下面是之前例子的另外一個變種，使用寬字串：

```
int idxInASArray = 0;
const wchar_t* strarr[2];
strarr[0] = L"This is the first string";
strarr[1] = L"This is the second one";
pMovie->SetVariableArray(Movie::SA_StringW, "_root.strArray",
                           idxInASArray, strarr, 2)
```

The `GetVariableArray` 方法將來自 AS 陣列的結果填充到了所提供的資料緩衝區中。該緩衝區必須具有足夠的空間來容納提出請求的項目數。

1.2.2 執行 ActionScript 副程式

除了修改 ActionScript 變數外，ActionScript 方法還能通過 [GFx::Movie::Invoke\(\)](#) 方法進行調用。這在處理更加複雜的進程、觸發動畫、改變當前幀、通過編程改變 UI 控制項當前狀態以及動態創建 UI 內容例如新按鈕或文本時是非常有用的。對於性能重要用法示例，請參閱 Direct Access API 中關於調用方法和控制動作流的更加有效的方法。

示例：

下列 ActionScript 函數可用於設置相對範圍內的滑塊位置。

假定在`_root` level 存在一一個“`mySlider`”物件。`SetSliderPos` 在 “`mySlider`” 物件中的定義如下：

AS Code:

```

this.SetSliderPos = function (pos)
{
    // Clamp the incoming position value
    if (pos < rangeMin) pos = rangeMin;
    if (pos > rangeMax) pos = rangeMax;
    gripClip._x = trackClip._width * ((pos - rangeMin) / (rangeMax - rangeMin));
    gripClip.gripPos = gripClip._x;
}

```

"gripClip" 是 "mySlider" 中的一個嵌套視頻剪輯，並且 pos 總是位於 rangeMin/Max 之中。

在用戶的程式中，調用函數的用法如下：

C++ Code:

```

Value result;
bool bInvoked = pMovie->Invoke("_root.mySlider.SetSliderPos", &result, "%d",
                                newPos);

```

如果該方法確實被調用，則 `Invoke` 返回值為真，否則為假。

在調用一個 ActionScript 函數時，你必須確保它已經載入。在調用時出現的一個常見錯誤是所調用的 ActionScript 程式是不可用的，在這種情況下，錯誤資訊會列印在 Scaleform log 中。只有當相關的幀已經播放或者相關的嵌套物件已經載入，ActionScript 程式才是可用的。只要對 `GFx::Movie::Advance` 進行了第一次調用，或者如果 `GFx::MovieDef::CreateInstance` 被 `initFirstFrame` 調用的設置為真，則在第 1 幀中的所有 ActionScript 代碼都是可用的。

在調用時，通常會使用 [GFx::Movie::IsAvailable\(\)](#) 方法，以確保 AS 在被調用前就已經存在了：

```

Value result;
if (pMovie->IsAvailable("parentPath.mySlider.SetSliderPos"))
    pMovie->Invoke("parentPath.mySlider.SetSliderPos", &result, "%d", newPos);

```

該示例中使用了'printf'格式的調用。在這個例子中，`Invoke` 將這些參數變為一個用格式字串描述的可變參數列表。其他版本的函數使用 [GFx::Value](#) 來有效處理非字串參數。例如：

```

Value args[3], result;
args[0].SetNumber(i);
args[1].SetString("test");
args[2].SetNumber(3.5);
pMovie->Invoke("path.to.methodName", &result, args, 3);

```

`InvokeArgs` 與 `Invoke` 是相同的，只是它採用 `va_list` 參數為程式提供一個紙箱可變參數列表的指標。`Invoke` 與 `InvokeArgs` 之間的關係類似於 `printf` 與 `vprintf` 之間的關係。

1.2.3 路徑

當在用戶程式內對 Flash 元素進行評估時，通常利用全路徑來查找物件。

下列 GFx::Movie 函數將依靠全路徑：

```
Movie::IsAvailable(path)
Movie::SetVariable(path, value)
Movie::SetVariableDouble(path, value)
Movie::SetVariableArray(path, index, data, count)
Movie::SetVariableArraySize(path, count)
Movie::GetVariable(value, path)
Movie::GetVariableDouble( path)
Movie::GetVariableArray(path, index, data, count)
Movie::GetVariableArraySize(path)
Movie::Invoke(pathToMethodName, result, argList, ...)
```

為了正確解決嵌套物件的路徑，所有父電影剪輯必須具有獨一無二的實例名稱。嵌套對象名稱以點"."隔開，並區分大小寫，如下所示。

在下面的示例中，考慮一個示例名稱為"clip2"的影片剪輯嵌入到另一個名稱為"clip1"的影片剪輯中，並位於主要平臺上。

Main Stage -> clip1 -> clip2

1. clip1 位於主要平臺內/上
2. clip2 位於 clip1 內

有效的路徑為：

```
"clip1.clip2"
"_root.clip1.clip2"
"_level0.clip1.clip2"
```

無效的路徑為：

```
"Clip1.clip2" <- 字母不匹配 - "Clip1" 必須用小寫字母
"clip2"      <- 丢失了父"clip1." - 路徑名稱
```

"_root"和"_level0"名稱在 ActionScript 2 中是可選的、並可用來強迫進行特定基礎級查找。不過，在 ActionScript 3 中它們是必需的，因為預設情況下查找將在期級發生。對於向後相容性，我們在 ActionScript 3 中提供了別名、以便於訪問帶有下列名稱的根:_root、_level0、root、level0。將不同的 SWF 檔載入到各個級別之中時、_root 將引用當前級別的基礎、而指定 _levelN (使用上面顯示的語法) 允許您選擇一個具體的級別。

注釋：

- 你可以在 Flash Studio 的 Test Movie (Ctrl-Enter) 環境下檢查物件和變數的路徑，方法是按 (**Ctrl-Alt-V**) (變數)。
- 中的時間線順序（從場景 1 鏈結開始）不會指定目標路徑。在此列出的某些特定元素並不真的用於物件路徑中。使用 (**Ctrl-Alt-V**) (變數) 彈出視窗來監測實際的有效路徑。
- 當向影片中載入影片時，利用 `loadMovie` 命令，你可能會遇到因物件層級變化而帶來的問題。原來位於 `_level0` 層級的物件現在位於 `_level1` 或 `_level2` (取決於你載入的層級)，或者位於目標影片剪輯內部。為了在其他層中引用物件，可簡單使用：`_levelN.objectName` 或 `_levelN.objectPath.objectName`，這裏的 N 表示層級編號。

本部分中的許多元素都採用了下列兩個路徑教程，我們推薦閱讀：

1. [Paths to Objects and Variables](#) 作者 Jesse Stratford
2. [Advanced Pathing](#) 作者 Jesse Stratford

2 Direct Access API

包含在 Scaleform 3.1 中的 Direct Access 值支持是在 AS 運行時進行遊戲通訊的主要改進。ActionScript 通訊 API 已經做出了簡單類型之外的擴展，因此可對複雜物件（以及這些物件中的單個元素）進行設置和有效查詢。例如，嵌套和異構資料結構現在能夠在遊戲 UI 進行前後傳遞。有關使用 Direct Access 的進一步示例，請參閱 HUD Kit 中的 [Minimap Demo](#)，它說明了在升級 minimap 中的大量 Flash 物件時所獲得的性能提升。

Direct Access 通過 GFx::Values 來說明，它包括簡單的 ActionScript 類型與物件、陣列和 Display Objects。Display Objects 是物件類型中的一種具體情況，並且與平臺上的實體相對應（MovieClips、Buttons、TextFields）。GFx::Value 類 API 具有全套功能，能夠設置和獲取他們的數值和成員，並且能夠處理陣列。更多資訊請參閱 [GFx::Value](#) 參考資料。

Direct Access API 允許程式把 C++ 變數 (GFx::Value 類型)直接綁定到 ActionScript 的物件中。一旦出現這種綁定或引用，該變數能夠很方便地進行使用，並有效修改 ActionScript 物件。在作出這一變化前，用戶已經通過 GFx::Movie 與規定字串路徑等方式獲取了 AS 物件。這種方法在解析路徑和尋找 AS 物件時會產生性能損失。对于直接出去(Direct Access)来说，在每次调用时出现的这种昂贵的解析和查询已经被删除。

之前只能在 GFx::Movie 層面可用的許多操作現在能够直接用于 ActionScript (AS)對象，由 [GFx::Value](#) 型來描绘。這使得代碼變得更加清晰和有效。例如，當調用根目錄下的一個名為'foo' 的物件方法時，我們可以獲得該物件的參數，並直接調用 Invoke ，而不是使用影片的 Invoke 調用。

1. GFx::Movie Invoke 方法 (不太有效):

```
Value ret;
Value args[N];
pMovie->Invoke( "_root.foo.method" , &ret, args, N );
```

2. Direct Access Invoke 方法 (相對較好):

```
(Assumes 'foo' holds a reference to an AS object at "_root.foo")
Value ret;
Value args[N];
bool = foo.Invoke( "method" , &ret, args, N );
```

關於 Invoke，對 AS 物件的檢查、獲取和設置調用現在都能通過 Direct Access API 利用 GFx::Value::HasMember、GetMember 和 SetMember 來實現。

利用 GetMember 對 MovieClip (存儲在 GFx::Value)中的一個文本欄位進行升級的例子：

```

Value tf;
MovieClip.GetMember("textField", &tf);
tf.SetText("hello");

```

在上述例子中，我們首先獲取 MovieClip 物件中的 textField 成員，然後使用函數 SetText 對其文本值進行升級。

2.1 物件與陣列

Direct Access API 所支持的另外一個重要能力是創建一個 AS 物件或層級物件。通過這種方式，物件的層級深度和結構都能夠被 C++ 創建和管理。例如，可使用下面的代碼片段來創建兩個具有層級結構的物件，並將其中的一個作為另一個的孩子：

```

Movie* pMovie = ... ;
Value parent, child;
pMovie->CreateObject(&parent);
pMovie->CreateObject(&child);
parent.SetMember("child", child);

```

[CreateObject](#) 創建了一個 ActionScript 物件實例。如果某一類的特定類型中的某一實例要求的話，它還接受可選的、完全限定的傳遞類名。例如：

```

Value mat, params[6];
// (為矩陣數據設置 params[0..6] ) ...
pMovie->CreateObject(&mat, "flash.geom.Matrix", params, 6);

```

考慮既使用物件又實用陣列的更加複雜的情況。下列代碼創建一個帶有複雜物件元素的陣列：

```

// (假定 pMovie 是一個 GFx::Movie* ):
Value owner, childArr, childObj;
pMovie->CreateArray(&owner); // 創建父矩陣
pMovie->CreateArray(&childArr); // 創建子矩陣
pMovie->CreateObject(&childObj); // 創建子對象
bool = owner.SetElement(0, childArr); // 將 parent[0] 設置為子矩陣
bool = owner.SetElement(1, childObj); // 將 parent[1] 設置為子對象
...
bool = foo.SetMember("owner", owner); // 稍後，將物件 foo 中的 'owner' 變數設置為父物件

```

函數 [GFx::Value::VisitMembers\(\)](#) 可用於遍曆對象的公共成員。該函數利用 ObjectVisitor 類中的一個實例，它擁有一個簡單的 Visit 回調（必須被覆蓋）。該函數僅對物件類型有效（包括陣列和 DisplayObject）。

請注意，你不能使用 *VisitMembers* 對某一類的實例進行完全自省。方法不勝枚舉，因為他們生活在原型中。為了獲得成員與屬性的可見性，可使用 ActionScript 中的 *ASSetPropFlags* 方法：

```
e.g.: _global.ASSetPropFlags(MyClass.prototype, ["someFunc", "__get__number",
                                                 "test"], 6, 1);
```

Properties (getter/setter) 無法通過 *VisitMembers* 枚舉，甚至通過 *ASSetPropFlags*。然而，他們可通過 Direct Access 介面來獲取，並返回 *VT_Object*。他們的值總是 "[屬性]"。函數以 *VT_Object* (可以枚舉的話) 形式返回。他們的值總是 "[類型 函數]"。

2.2 顯示物件

Direct Access API 列出了 *Object* 類型的一個特別實例，稱為 *DisplayObject*。它與平臺實體相對應，例如 *MovieClips*、*Buttons*、*TextFields*。所提供的自定義的 [DisplayInfo](#) API 能夠獲取他們的顯示屬性。利用 *DisplayInfo* API，你能輕鬆設置 *DisplayObject* 的屬性，例如透明度、旋轉、可見性、位置、偏移和大小。儘管這些顯示屬性還能通過 *SetMember* 函數來設置，但 *SetDisplayInfo* 調用在完成這項工作時是最快的，因為它能直接修改物件的顯示屬性。請注意，這兩種方法都比調用 *Gfx::Movie::SetVariable* 快，因為後者不是直接操作目標物件。

下列代碼利用 *SetDisplayInfo* 方法來改變 movieclip 實例的位置和旋轉角度：

```
// 假定 MovieClip 是一個 Gfx::Value*
Value::DisplayInfo info;
PointF pt(100,100);
info.setRotation(90);
info.setPosition(pt.x, pt.y);
MovieClip.SetDisplayInfo(info);
```

2.3 函數物件

ActionScript2 虛擬機 (AS2 VM)中的函數與基本物件是相似的。這些函數物件可以是分配的部分，它們能夠按照使用情況提供額外的 內省資訊。利用 *Gfx::Movie::CreateFunction* 方法，開發者能夠創建帶有 C++ 回調物件的函數物件。通過 *CreateFunction* 返回的函數物件可作為虛擬機中任何物件的一部分進行分配。但這一 AS 函數被調用時，C++回調也依次被調用。這種能力能夠讓開發者從虛擬機向他們自己的回調處理程式中註冊直接回調，而無需在上面作出額外委派（通過 *fscommand* 或 *ExternalInterface*）。

下列示例創建了一個自定義回調，並將其分配給 AS 物件的某一部分：

```

Value obj;
pmovie->GetVariable(&obj, "_root.obj");

class MyFunc : public FunctionHandler
{
public:
    virtual void Call(const Params& params)
    {
        // 回調邏輯/處理
    }
};

Ptr<MyFunc> customFunc = *SF_HEAP_NEW(Memory::GetGlobalHeap()) MyFunc();
Value func;
pmovie->CreateFunction(&func, customFunc);
obj.SetMember("func", func);

```

這一方法可在 ActionScript (在 _root timeline) 中進行調用：

```
obj.func(param1, param2, param3);
```

傳遞到 Call() 方法的 Params 結構將包含下列各項：

- pRetVal：指向一個 GFx::Value 的指標，該 GFx::Value 將被作為返回值回傳給調用程式。如果回傳一個複雜物件，則將此指標傳遞到 pMovie->CreateObject() or CreateArray() 以創建容器。對於原始類型(數位、布林值等)，請調用適當的設值函數。
- pMovie：指向調用了該函數的電影的指標。
- pHsThis：調用程式上下文。如果調用上下文為不存在或無效，可能未定義。
- ArgCount：傳遞到回呼函數的參數的數量。
- pArgs：代表傳遞到回呼函數的參數的 GFx::Values 陣列。使用 [] 運算子訪問各個參數。例如：
GFx::Value firstArg = params.pArgs[0];
- pArgsWithThisRef：參數陣列加上預先掛起的調用上下文。這用來連結其他函數物件(有關注入自訂行為的示例，請參見下文)。
- pUserData：註冊函數物件時的自訂資料集。此資料對於將回呼函數以自訂方式委派給不同的處理常式方法非常有用。

函數物件通常能夠覆蓋虛擬機中的現有函數，也會在現有函數體的開始或末尾加入自定義行為。函數物件還能註冊靜態方法或帶有類定義的示例方法。通過擴展還能定義一個自定義類。下列示例創建了一個能在虛擬機中實現的示例類：

```

Value networkProto, networkCtorFn, networkConnectFn;
class NetworkClass : public FunctionHandler
{
public:
    enum Method

```

```

{
    METHOD_Ctor,
    METHOD_Connet,
};

virtual ~NetworkClass() {}
virtual void Call(const Params& params)
{
    int method = int(params.pUserData);
    switch (method)
    {
        case METHOD_Ctor:
            // 自定義邏輯
            break;
        case METHOD_Connet:
            // 自定義邏輯
            break;
    }
}
};

Ptr<NetworkClass> networkClassDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                         NetworkClass();

// 創建構造函數
pmovie->CreateFunction(&networkCtorFn, networkClassDef,
                        (void*)NetworkClass::METHOD_Ctor);

// 創建原型物件
pmovie->CreateObject(&networkProto);
// 在構造函數中對原型進行設置
networkCtorFn.SetMember("prototype", networkProto);
// 為“連接”創建原型方法
pmovie->CreateFunction(&networkConnectFn, networkClassDef,
                        (void*)NetworkClass::METHOD_Connet);
networkProto.SetMember("connect", networkConnectFn);
// 註冊帶有_global的構造函數
pmovie->SetVariable("_global.Network", networkCtorFn);

```

現在，可以在虛擬機中實現此類：

```
var netObj:Object = new Network(param1, param2);
```

植入行為主要針對那些具備虛擬機專業知識的開發者，同時為了在虛擬機物件的生命期內進行管理。下列示例說明了行為植入：

```
Value origFuncReal;
obj.GetMember("funcReal", &origFuncReal);
```

```

class FuncRealIntercept : public FunctionHandler
{
    Value OrigFunc;
public:
    FuncRealIntercept(Value origFunc) : OrigFunc(origFunc) {}
    virtual ~FuncRealIntercept() {}
    virtual void Call(const Params& params)
    {
        // 擷取邏輯 (開始)
        OrigFunc.Invoke("call", params.pRetVal, params.pArgsWithThisRef,
                        params.ArgCount + 1);
        // 擷取邏輯 (結束)
    }
};

Value funcRealIntercept;
Ptr<FuncRealIntercept> funcRealDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                         FuncRealIntercept(origFuncReal);
pmovie->CreateFunction(&funcRealIntercept, funcRealDef);
obj.SetMember("funcReal", funcRealIntercept);

```

說明：`GFx::Value` 在自定義函數環境物件中的壽命必須由開發者來維護，因為它為虛擬機物件保留了參考。在.swf (GFx::Movie)失效前，需要開發者對該參考非常清楚。這一要求與所有 `GFx::Values` 相關的壽命管理要求（保留虛擬機複雜物件的參考）是一致的。

2.4 Direct Access 公共介面

這裏介紹的是 Direct Access API 中的公共成員函數。最新資訊請參閱 [GFx::Value](#) 線上文件。

2.4.1 Object 支持

描述	公共方法 (假定 'foo' 保存著 Object 在 "_root.foo" 中的一個參數)
檢查在物件中是否存在成員	bool has = foo.HasMember("bar");
在某一物件中檢索某一數值	Value bar; bool = foo.GetMember("bar", &bar);
在某一物件中設置一個數值	Value val; bool = foo.SetMember("bar", val);
調用某一物件中一個方法	Value ret; Value args[N]; bool = foo.Invoke("method", args, N, &ret); or foo.Invoke("method", args, N);

創建一個物件	<pre>Value obj; pMovie->CreateObject(&obj); ... bool = foo.SetMember("bar", obj);</pre>
創建一個具有層次結構的物件	<pre>Value owner, child; pMovie->CreateObject(&owner); pMovie->CreateObject(&child); bool = owner.SetMember("child", child); ... bool = foo.SetMember("owner", owner);</pre>
對某一物件中的成員進行迭代	<pre>Value::ObjectVisitor v; foo.VisitMembers(&v);</pre>
從某一對象中刪除某一成員	<pre>bool = foo.DeleteMember("bar");</pre>

2.4.2 Array 支持

公共方法	
描述	(假定 'bar' 存儲了 Array 在 "_root.foo.bar" 中的一個參數，並且 'foo' 保存了 Object 中的一個參數)
確定陣列大小	<code>unsigned sz = bar.GetArraySize();</code>
在陣列中對某一元素進行檢索	<code>Value val; bool = bar.GetElement(idx, &val);</code>
對陣列中的某一元素進行設置	<code>Value val; bool = bar.SetElement(idx, val);</code>
重新確定陣列大小	<code>bool = bar.SetArraySize(N);</code>
創建一個陣列	<pre>Value arr; pMovie->CreateArray(&arr); ... bool = foo.SetMember("bar", arr);</pre>
創建將其他 Complex Object 作為元素的一個陣列	<pre>Value owner, childArr, childObj; pMovie->CreateArray(&owner); pMovie->CreateArray(&childArr); pMovie->CreateObject(&childObj); bool = owner.SetElement(0, childArr); bool = owner.SetElement(1, childObj); ... bool = foo.SetMember("owner", owner);</pre>
對陣列的某一範圍內的元素進行迭代	<p>列舉</p> <pre>for (UPInt i=0; i < N; i++) { ... bool = bar.GetElement(i+idx, &val) ... }</pre> <p>使用訪問者模式：</p> <pre>Value::ArrayVisitor v;</pre>

	<pre>bar.VisitElements(v, idx, N); bool = bar.ClearElements();</pre>
清空陣列	<pre>PushBack GValue val; bool = bar.PushBack(val);</pre>
堆疊操作	<pre>PopBack Value val; bool = bar.PopBack(&val); or void bar.PopBack();</pre>
	<p>單個元素</p> <pre>bool = bar.RemoveElement(idx);</pre> <p>系列元素</p> <pre>bool = bar.RemoveElements(idx, N);</pre>
從陣列中刪除元素	

2.4.3 顯示物件支援

描述	公共方法
獲得當前顯示資訊	(假設 'foo' 在"_root.foo.bar"中保留了顯示物件(MovieClip, TextField, Button)的參考。)
設置當前顯示資訊	<pre>Value::DisplayInfo info; bool = foo.GetDisplayInfo(&info);</pre>
獲得當前顯示矩陣	<pre>Value::DisplayInfo info; ... bool = foo.SetDisplayInfo(info);</pre>
設置當前顯示矩陣	<pre>Matrix2_3 mat; bool = foo.GetDisplayMatrix(&mat);</pre>
獲得當前顏色轉換	<pre>Matrix2_3 mat; ... bool = foo.SetDisplayMatrix(mat);</pre>
設置當前顏色轉換	<pre>Render::Cxform cxform; bool = foo.GetColorTransform(&cxform);</pre>
	<pre>Render::Cxform cxform; ... bool = fooSetColorTransform(cxform);</pre>

2.4.4 影片剪輯支援

描述	公共方法
----	------

	(假定 'foo' 在 "_root.foo" 中保留了影片剪輯的參考。)
為影片剪輯添加識別字號	<pre>Value newInstance; bool = foo.AttachMovie(&newInstance, "SymbolName", "instanceName");</pre>
創建一個空的影片剪輯作為該 影片剪輯的孩子	<pre>Value emptyInstance; bool = foo.CreateEmptyMovieClip(&emptyInstance, "instanceName");</pre>
按名稱播放一個關鍵幀	<pre>bool = foo.GotoAndPlay("myframe");</pre>
按序號播放一個關鍵幀	<pre>bool = foo.GotoAndPlay(3);</pre>
按名稱停止一個關鍵幀	<pre>bool = foo.GotoAndStop("myframe");</pre>
按序號停止一個關鍵幀	<pre>bool = foo.GotoAndStop(3);</pre>

2.4.5 文本欄位支援

描述	公共方法 (假設 'foo' 在 "_root.foo" 中保留了一個文本欄位參考。)
獲得原始文本欄位文本	<pre>Value text; bool = foo.GetText(&text);</pre>
設置原始文本欄位文本	<pre>Value text; ... bool = bar.SetText(text);</pre>
獲得 HTML 文本	<pre>Value htmlText; bool = bar.GetTextHTML(&htmlText);</pre>
設置 HTML 文本	<pre>Value htmlText; ... bool = bar.SetTextHTML(htmlText);</pre>