Autodesk®
Scaleform®

Scaleform 游戏通讯概述

本文件描述的是利用 Scaleform 3.1 及其更新版本在 C++、 Flash 和 ActionScript 之间的通讯机制。

作者： Mustafa Thamer
版本： 2.01
最后修订日期： 2010 年 9 月 21 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.3

如何与 Autodesk Scaleform 联系：

文件名称	Scaleform 游戏通讯概述
地址	美国格林贝尔特 MD 20770 常春藤路 6305 号 310 室 Scaleform 公司
网站	www.scaleform.com
电子邮件	info@scaleform.com
直线电话	(301) 446-3200
传真	(301) 446-3199

目 录

1 C++、 Flash 和 ActionScript 建立接口

Flash®的 ActionScript™ 脚本语言能够创建交互式视频内容。诸如点击按钮、达到某一帧或加载某一视频等事件都能动态改变视频内容，控制视频流，甚至会启动更多视频。 ActionScript 完全有能力在 Flash 中创建完整的迷你游戏。与大多数编程语言一样， ActionScript 支持变量和子程序，并包括代表项目或控制的对象。

程序与 Flash 之间的通讯需要用复杂的例子来说明。Autodesk Scaleform®支持 Flash 提供的标准机制，它能够让 ActionScript 传递事件，并将数据返回到 C++程序。它还提供了一个便捷的 C++ 接口，可直接操纵 ActionScript 变量、数组和对象，并直接调用 ActionScript 子程序。

在本文件中，我们讨论了 C++与 Flash 之间通讯的不同机制。下面列出了可用选项：

ActionScript → C++	**C++ → ActionScript**
FSCommand 简单，基于字符串的函数执行，没有返回值，不推荐	**GFx::Movie::Get/SetVariable** 在 ActionScript 获取数据，使用字符串路径
ExternalInterface 灵活的参数处理，支持返回值，推荐	**GFx::Movie::Invoke** 在 ActionScript 中调用函数，使用字符串路径
Direct Access API Uses 使用 GFx::Value 作为数据和函数获取对象的直接参数，性能高	

1.1 ActionScript 到 C++

Scaleform 为 C++程序从 ActionScript 接收事件提供了两种机制： *FSCommand* 和 *ExternalInterface*。这些机制是标准 ActionScript API 的一部分，有关这部分内容的更多信息可参阅 Flash 文件。FSCommand 和 ExternalInterface 均通过 Scaleform 注册了 C++ 事件处理器来接收事件通知。这些处理器按照 Scaleform 状态进行注册，因此能够在 GFx::Loader、 GFx::MovieDef 或 GFx::Movie 中进行注册，具体情况取决于函数要求。有关 GFx 状态的信息，请参阅 Scaleform documentation.

FSCommand 事件通过 ActionScript 的 'fscommand' 函数来触发。 fscommand 函数只能从 ActionScript 传递两个字符串参数，一个用于命令，另外一个用于单个数据参数。这些字符串的数值被 C++处理器接收，并作为两个常数字符串指针。 不幸的是，C++ fscommand 处理器 无法返回数值，因为 ActionScript fscommand 函数不具有返回值支持。

ExternalInterface 事件会在 ActionScript 调用 *flash.external.ExternalInterface.call* 函数时触发。这一接口能够传递 ActionScript 值的任意列表和命令名称。 C++ ExternalInterface 处理器接收一个常量字符指针用作命令名称和运行时所传递的与 ActionScript 值对应的 GFx::Value 参数数组。C++ ExternalInterface 处理器还能返回一个 GFx::Value，因为 ActionScript ExternalInterface.call 方法支持返回值。ActionScript ExternalInterface 类是外部 API 的一部分， 一个应用程序接口，在使用 Scaleform 的程序中，它能在 ActionScript 与 Flash 播放器容器之间实现直接通讯。

由于受到灵活性的限制， FSCommands 因 ExternalInterface 而变得过时，并且不再推荐使用。在这里进行描述是为了完整性和传统性支持的需要。

在一定程度上，Direct Access API 与 GFx::FunctionHandler 界面会使 FSCommands 和 ExternalInterface 变得过时。该界面允许将 ActionScript VM 中分配的 C++方法按照正常的函数直接进行回调。当 ActionScript 中的函数被调用时，它会反过来调用 C++回调。有关 GFx::FunctionHandler 的更多信息，请参阅 Direct Access API 部分内容。

1.1.1 FSCommand 回调

ActionScript fscommand 函数向主机应用程序传递命令和数据参数。下面是在 ActionScript 中的典型用法：

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

向 fscommand 传递的任何非字符串参数，例如布尔值或整数都将转换成字符串。

ActionScript fscommand 调用 会向 GFx FSCommand 处理器传递两个字符串。 应用程序会通过子类 GFx::FSCommandHandler 注册一个 fscommand 处理器，并为这一类注册一个示例作为 GFx::Loader、GFx::MovieDef 或单个 GFx::Movie 对象的共享状态。请牢记， 状态包之间的设置是为了做出如下委托：GFx::Loader -> GFx::MovieDef -> GFx::Movie。他们可在后续的任何示例中进行覆盖。这意味着，如果你想获得一个拥有自己的 GFx::RenderConfig （例如，对于单独的 EdgeAA 控制）或 GFx::Translator （以便能够翻译成不同的语言），你可在那个对象中进行设置，并且无论应应用什么状态，在委托链中都会存在一个高于该对象所用的状态，例如 GFx::Loader。如果在 GFx::Movie 中设置了一个命令处理器，它只能在那个视频示例中收到 FSCommand 调用的回调。 GFxPlayerTiny 示例说明了这一过程 (查找 “FxPlayerFSCommandHandler”)。

下面是 fscommand 处理器设置的例子。 该处理器源于 GFx::FSCommandHandler。

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

回调方法收到两个传递到 ActionScript 的 fscommand 中的字符串参数，同时作为这一具体视频示例调用 fscommand 的指针。 我们的自定义处理器只是简单地将每一个 fscommand 事件打印到调试控制台上。

接下来，在创建 GFx::Loader 对象后注册处理器：

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

通过 GFx::Loader 注册处理器会导致每一个 GFx::Movie 和 GFx::MovieDef 都能继承这一处理器。SetFSCommandHandler 可在某一视频示例中进行调用，以覆盖这一默认设置。

一般来说， C++ 事件处理器应当是无锁定的，并且会尽快返回调用程序。事件处理器通常只能在 Advance 或 Invoke calls 中进行调用。

1.1.2 外部接口 API

Flash ExternalInterface.调用方法与 fscommand 类似，但它是首选方法，因为它能提供更加灵活的参数处理，并且能够返回数值。注册一个 ExternalInterface 处理器与注册一个 fscommand 处理器类似。 这里举一个 ExternalInterface 处理器的例子，它能打印数字和字符串参数。

```
class OurExternalInterfaceHandler : public ExternalInterface
{
    public:
    virtual void Callback(MovieView* 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);
    }
};
```

一旦处理器开始执行，它的一个实例将会按照 GFx::Loader 进行如下登记：

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

现在，回调处理器可通过 ActionScript 中的 ExternalInterface 调用而触发。

1.1.2.1 FxDelegate

上述 ExternalInterface 处理器提供了非常基础的回调功能。在实际应用中， 每当 ExternalInterface 回调通过某一特定 methodName 被触发时，应用程序都可能会注册具体的调用函数。这需要一个更加先进的回调系统，它允许函数按照相应的 methodName 进行注册，然后使用哈希表（或类似的东西）进行查找，并利用相应的 GFx::Value 参数执行他们。我们在示例中提供了这样的系统，并被称为 FxDelegate (参见 *Apps\Samples\GameDelegate\FxGameDelegate.h*)。

FxDelegate 来自 GFx::ExternalInterface。 它拥有函数注册/注销处理器，允许应用程序安装自己的回调处理器，并通过 methodName 进行注册。

关于 FxDelegate 的用法示例，请参阅我们的 HUD Kit ，它使用 FxDelegate 实施它的 minimap。这里有一个来自 minimap 演示的程序编码，它利用 FxDelegate 注册了自己的回调函数：

```
// 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，并作为一个更加复杂的回调处理和注册系统示例。它应当成为用户的出发点。我们鼓励开发者去查看一下它的工作细节，并为了他们自己的需要而进行自定义和扩展。

1.1.2.2 使用 ActionScript 的外部接口

在 ActionScript 中，外部接口调用将通过下列代码进行初始化：

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

这里的“foo” 是方法名称，“arg”为基本类型的参数。你还可以规定 0 或更多参数，以逗号隔开。有关 ExternalInterface API 的更多信息，请参阅 Flash 文件。

如果 ExternalInterface 没有输入上述代码，ExternalInterface 调用必须完全合格：

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

1.1.2.3 ExternalInterface.addCallback

ExternalInterface.addCallback 函数通常会使用别名注册一个 ActionScript 方法。这允许从 C++中调用注册的方法，而无需路径前缀。它支持在单一别名下注册方法和调用内容。注册别名可被 C++ GFx::Movie::Invoke()方法进行调用。

示例：

AS Code:

```
import flash.external.*;

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

varaliasName: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!";
}
```

现在，通过调用别名就可能调用 SetText ActionScript 函数将"this"在 txtField 中进行设置：

C++ Code:

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

这一结果将会跟踪 "txtField.SetText"，并且将文本 "INVOKED!"设置到 "txtField" 文本字段中。有关使用 GFx::Movie::Invoke()的更多信息，请参阅第 1.2.2 部分内容。.

1.2 C++到ActionScript

前面章节解释了 ActionScript 如何调用 C++。本节内容将描述如何利用能够在 C++与 Flash 视频之间进行通讯的 Scaleform 函数实现相反方向的通讯。 Scaleform 支持 C++ 函数直接获取和设置 ActionScript 变量（简单类型、复杂类型和数组），同时调用 ActionScript 子程序。

Direct Access API 为访问和操纵来自 C++的 ActionScript 变量提供了更加便捷和有效的界面。有关这一界面的更多信息，请参阅 Direct Access API 部分内容。

1.2.1 操纵 ActionScript 变量

Scaleform 支持 GetVariable and SetVariable, 它们能够直接操纵 ActionScript 变量。对于执行关键用法的示例，请参阅 Direct Access API 第 2 部分关于修改变量和对象属性的更有效方法的内容。 尽管出于性能原因不推荐使用 Set/GetVariable，但在某些情况下，它们比使用 Direct Access API 更便捷。 例如，没有必要使用 Direct Access call GFx::Value::SetMember 在该程序的生命期内设置一个单一数值，尤其是如果该目标处于深嵌套层面中时。还有一种情况，通常通过 GetVariable 从 ActionScript 对象中获取参数，调用一次就可获得一个 GFx::Value 参数，该参数随后将用于 Direct Access 操作。

下列示例说明了利用 GetVariable 和 SetVariable 使 ActionScript 计数器的计数开始递增：

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

GetVariableDouble 返回的 _root.counter 变量值将会自动转换为 C++ Double 类型。 最初，这一变量并不存在，因此 GetVariableDouble 的返回值为 0.计数器的计数会在下一行增加，新值通过 SetVariable 存储到 _root.counter 中。GFx::Movie 的在线文件列出了 GetVariable 与 SetVariable 之间的不同。

SetVariable 具有一个可选的第三参数，其类型为 GFx::Movie::SetVarType， 它能声明分配“粘性”。当所分配的变量还没有在 Flash 时间线中创建时，这个功能会很有用。例如，t 假定文本字段_root.mytextfield 在该视频的第三帧之前并没有创建，如果 SetVariable(“_root.mytextfield.text”, “testing”, SV_Normal)在第 1 帧中进行调用，当该视频刚刚被创建后，这种分配是无效的。然而，如果通过 SV_Sticky (默认值) 来调用，那么这一请求将进行排队，一旦_root.mytextfield.text 值在第 3 帧开始生效，它就可以执行了。这使得 C++中的视频初始化变得更加容易。请牢记，通常来说， SV_Normal 比 SV_Sticky 更有效，因此，在可能的情况下应当使用 SV_Normal。

为了获取数据数组， Scaleform 提供了函数 SetVariableArray 和 GetVariableArray。

SetVariableArray 将规定类型的数据项目设置到规定范围的数组元素中。如果该数组并不存在，它就会进行创建。如果数组已经存在，但包含的项目不够，它就会适当地扩容。然而， 当设置的一些元素小于当前数组的大小时，并不会改变数组的大小。GFx::Movie::SetVariableArraySize 设置数组的大小，当在现有数组中所设置的元素少于之前的元素时 ，这一函数是有用的。

下列示例说明了 SetVariableArray 在 AS 中设置字符串数组的用法：

```
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个变种，使用宽字符串：

```
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 方法将来自 AS 数组的结果填充到了所提供的数据缓冲区中。该缓冲区必须具有足够的空间来容纳提出请求的项目数。

1.2.2 执行 ActionScript 子程序

除了修改 ActionScript 变量外， ActionScript 方法还能通过 GFx::Movie::Invoke() 方法进行调用。这在处理更加复杂的进程、触发动画、改变当前帧、 通过编程改变 UI 控件当前状态以及动态创建 UI 内容例如新按钮或文本时是非常有用的。对于性能重要用法示例，请参阅 Direct Access API 中关于调用方法和控制动作流的更加有效的方法。

示例：
下列 ActionScript 函数可用于设置相对范围内的滑块位置。

假定在_root level 存在一个“mySlider”对象。SetSliderPos 在 "mySlider" 对象中的定义如下：

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"中的一个嵌套视频剪辑，并且 pos 总是位于 rangeMin/Max 之中。

在用户的程序中，调用函数的用法如下：

C++ Code:

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

如果该方法确实被调用，则 Invoke 返回值为真，否则为假。

在调用一个 ActionScript 函数时，你必须确保它已经加载。在调用时出现的一个常见错误是所调用的 ActionScript 程序是不可用的，在这种情况下，错误信息会打印在 Scaleform log 中。 只有当相关的帧已经播放或者相关的嵌套对象已经加载，ActionScript 程序才是可用的。只要对 GFx::Movie::Advance 进行了第一次调用，或者如果 GFx::MovieDef::CreateInstance 被 initFirstFrame 调用的设置为真，则在第 1 帧中的所有 ActionScript 代码都是可用的。

在调用时，通常会使用 GFx::Movie::IsAvailable() 方法，以确保 AS 在被调用前就已经存在了：

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

该示例中使用了'printf'格式的调用。在这个例子中， Invoke 将这些参数变为一个用格式字符串描述的可变参数列表。其他版本的函数使用 GFx::Value 来有效处理非字符串参数。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 与 Invoke 是相同的，只是它采用 va_list 参数为程序提供一个纸箱可变参数列表的指针。Invoke 与 InvokeArgs 之间的关系类似于 printf 与 vprintf 之间的关系。

1.2.3 路径

当在用户程序内对 Flash 元素进行评估时，通常利用全路径来查找对象。

下列 GFx::Movie 函数将依靠全路径：

```
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, ...)
```

为了正确解决嵌套对象的路径，所有父电影剪辑必须具有独一无二的实例名称。嵌套对象名称以点“.”隔开，并区分大小写，如下所示。

在下面的示例中，考虑一个示例名称为"clip2"的影片剪辑嵌入到另一个名称为"clip1"的影片剪辑中，并位于主要平台上。

Main Stage -> clip1 -> clip2

1. clip1 位于主要平台内/上
2. clip2 位于 clip1 内

有效的路径为：

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

无效的路径为：

```
"Clip1.clip2" <- 字母不匹配 - "Clip1" 必须用小写字母
"clip2"       <- 丢失了父"clip1." - 路径名称
```

"_root"和 "_level0" 名称在 ActionScript 2 中是可选的，并可用来强迫进行特定基础级查找。不过，在 ActionScript 3 中它们是必需的，因为默认情况下查找将在 stage 级发生。对于向后兼容性，我们在 ActionScript 3 中提供了别名，以便于访问带有下列名称的根：_root、_level0、root、level0。将不同的 SWF 文件加载到各个级别之中时，_root 将引用当前级别的基础，而指定 _levelN（使用上面显示的语法）允许您选择一个具体的级别。

注释：

- 你可以在 Flash Studio 的 Test Movie (Ctrl-Enter)环境下检查对象和变量的路径，方法是按 (**Ctrl-Alt-V)** (变量)。

- 中的时间线顺序（从场景 1 链接开始）不会指定目标路径。在此列出的某些特定元素并不真的用于对象路径中。使用 (Ctrl-Alt-V) (变量) 弹出窗口来监测实际的有效路径。

- 当向影片中加载影片时，利用 loadMovie 命令，你可能会遇到因对象层级变化而带来的问题。原来位于 _level0 层级的对象现在位于 _level1 或 _level2 (取决于你加载的层级)，或者位于目标影片剪辑内部。为了在其他层中引用对象，可简单使用： `_levelN.objectName` 或 `_levelN.objectPath.objectName`，这里的 N 表示层级编号。

本部分中的许多元素都采用了下列两个路径教程，我们推荐阅读：

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 运行时进行游戏通讯的主要改进。 ActionScript 通讯 API 已经做出了简单类型之外的扩展，因此可对复杂对象（以及这些对象中的单个元素）进行设置和有效查询。例如，嵌套和异构数据结构现在能够在游戏 UI 进行前后传递。 有关使用 Complex Objects 的进一步示例，请参阅 HUD Kit 中的 Minimap Demo ，它说明了在升级 minimap 中的大量 Flash 对象时所获得的性能提升。

Direct Access 通过 GFx::Values 来说明，它包括简单的 ActionScript 类型与对象、数组和 Display Objects。Display Objects 是对象类型中的一种具体情况，并且与平台上的实体相对应（MovieClips、Buttons、TextFields）。 GFx::Value 类 API 具有全套功能，能够设置和获取他们的数值和成员，并且能够处理数组。更多信息请参阅 GFx::Value 参考资料。

Direct Access API 允许程序把 C++ 变量 (GFx::Value 类型)直接绑定到 ActionScript 的对象中。 一旦出现这种绑定或引用，该变量能够很方便地进行使用，并有效修改 ActionScript 对象。在作出这一变化前，用户已经通过 GFx::Movie 与规定字符串路径等方式获取了 AS 对象。这种方法在解析路径和寻找 AS 对象时会产生性能损失。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 层面可用的许多操作现在能够直接用于 ActionScript (AS)对象, 由 GFx::Value 类型来描绘。这使得代码变得更加清晰和有效。例如， 当调用根目录下的一个名为‘foo’ 的对象方法时，我们可以获得该对象的参数，并直接调用 Invoke ，而不是使用影片的 Invoke 调用。

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 (相对较好):

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

关于 Invoke，对 AS 对象的检查、获取和设置调用现在都能通过 Direct Access API 利用 GFx::Value::HasMember、 GetMember 和 SetMember.来实现。

利用 GetMember 对 MovieClip (存储在 GFx::Value)中的一个文本字段进行升级的例子：

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

在上述例子中，我们首先获取 MovieClip 对象中的 textfield 成员，然后使用函数 SetText 对其文本值进行升级。

2.1 对象与数组

Direct Access API 所支持的另外一个重要能力是创建一个 AS 对象或层级对象。通过这种方式，对象的层级深度和结构都能够被 C++创建和管理。例如，可使用下面的代码片段来创建两个具有层级结构的对象，并将其中的一个作为另一个的孩子：

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

CreateObject 创建了一个 ActionScript 对象实例。如果某一类的特定类型中的某一实例要求的话，它还接受可选的、 完全限定的传递类名。例如：

```
Value mat, params[6];
// (为矩阵数据设置 params[0..6] ) …
pMovie->CreateObject(&mat, "flash.geom.Matrix", params, 6);
```

考虑既使用对象又实用数组的更加复杂的情况。下列代码创建一个带有复杂对象元素的数组：

```
// (假定 pMovie 是一个 GFx::Movie*):
Value owner, childArr, childObj;
pMovie->CreateArray(&owner);            // 创建父矩阵
pMovie->CreateArray(&childArr);         // 创建子矩阵
pMovie->CreateObject(&childObj);        // 创建子对象
bool = owner.SetElement(0, childArr);  //将 parent[0]设置为子矩阵
bool = owner.SetElement(1, childObj); // 将 parent[1]设置为子对象
…
bool = foo.SetMember("owner", owner); // 稍后，将对象 foo 中的'owner'变量设置为父对象
```

函数 GFx::Value::VisitMembers() 可用于遍历对象的公共成员。该函数利用 ObjectVisitor 类中的一个实例，它拥有一个简单的 Visit 回调（必须被覆盖）。该函数仅对对象类型有效（包括数组和 DisplayObject）。

请注意，你不能使用 *VisitMembers* 对某一类的实例进行完全自省。方法不胜枚举，因为他们生活在原型中。为了获得成员与属性的可见性，可使用 ActionScript 中的 ASSetPropFlags 方法：

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

Properties (getter/setter) 无法通过 VisitMembers 枚举，甚至通过 ASSetPropFlags。然而，他们可通过 Direct Access 接口来获取，并返回 VT_Object。他们的值总是 "[属性]"。函数以 VT_Object (可以枚举的话) 形式返回。 他们的值总是 "[类型 函数]"。

2.2 显示对象

Direct Access API 列出了 Object 类型的一个特别实例，称为 DisplayObject。它与平台实体相对应，例如 MovieClips、Buttons、TextFields。 所提供的自定义的 DisplayInfo API 能够获取他们的显示属性。 利用 DisplayInfo API，你能轻松设置 DisplayObject 的属性，例如透明度、 旋转、 可见性、位置、偏移和大小。尽管这些显示属性还能通过 SetMember 函数来设置，但 SetDisplayInfo 调用在完成这项工作时是最快的，因为它能直接修改对象的显示属性。请注意，这两种方法都比调用 GFx::Movie::SetVariable 快，因为后者不是直接操作目标对象。

下列代码利用 SetDisplayInfo 方法来改变 movieclip 示例的位置和旋转角度：

```
// 假定 MovieClip 是一个 GFx::Value*
Value::DisplayInfo info;
PointF pt(100,100);
info.SetRotation(90);
info.SetPosition(pt.x, pt.y);
MovieClip.SetDisplayInfo(info);
```

2.3 函数对象

ActionScript2 虚拟机 (AS2 VM)中的函数与基本对象是相似的。这些函数对象可以是分配的部分，它们能够按照使用情况提供额外的 内省信息。利用 GFx::Movie::CreateFunction 方法，开发者能够创建带有 C++ 回调对象的函数对象。通过 CreateFunction 返回的函数对象可作为虚拟机中任何对象的一部分进行分配。但这一 AS 函数被调用时，C++回调也依次被调用。这种能力能够让开发者从虚拟机向他们自己的回调处理程序中注册直接回调，而无需在上面作出额外委派（通过 fscommand 或 ExternalInterface）。

下列示例创建了一个自定义回调，并将其分配给 AS 对象的某一部分：

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

class MyFunc : public FunctionHandler
{
    public:
        virtual void Call(const Params& params)
        {
              // 回调逻辑/处理
        }
};
Ptr<MyFunc> customFunc = *SF_HEAP_NEW(Memory::GetGlobalHeap()) MyFunc();
Value func;
pmovie->CreateFunction(&func, customFunc);
obj.SetMember("func", func);
```

这一方法可在 ActionScript (在 _root timeline)中进行调用：

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

传递到 Call() 方法的 Params 结构将包含下列各项：

- `pRetVal`：指向一个 GFx::Value 的指针，该 GFx::Value 将被作为返回值回传给调用程序。如果回传一个复杂对象，则将此指针传递到 pMovie->CreateObject() or CreateArray() 以创建容器。对于原始类型（数字、布尔值等），请调用适当的设值函数。
- `pMovie`：指向调用了该函数的电影的指针。
- `pThis`：调用程序上下文。如果调用上下文为不存在或无效，可能未定义。
- `ArgCount`：传递到回调函数的参数的数量。
- `pArgs`：代表传递到回调函数的参数的 GFx::Values 数组。使用 [] 运算符访问各个参数。
  例如： `GFx::Value firstArg = params.pArgs[0];`
- `pArgsWithThisRef`：参数数组加上预先挂起的调用上下文。这用来链接其他函数对象（有关注入自定义行为的示例，请参见下文）。
- `pUserData`：注册函数对象时的自定义数据集。此数据对于将回调函数以自定义方式委派给不同的处理程序方法非常有用。

函数对象通常能够覆盖虚拟机中的现有函数，也会在现有函数体的开始或末尾加入自定义行为。函数对象还能注册静态方法或带有类定义的示例方法。通过扩展还能定义一个自定义类。下列示例创建了一个能在虚拟机中实现的示例类：

```
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:
                // 自定义逻辑
                break;
            case METHOD_Connet:
                // 自定义逻辑
                break;
            }
        }
    };
    Ptr<NetworkClass> networkClassDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                                        NetworkClass();
    // 创建构造函数
    pmovie->CreateFunction(&networkCtorFn, networkClassDef,
                           (void*)NetworkClass::METHOD_Ctor);
    // 创建原型对象
    pmovie->CreateObject(&networkProto);
    //在构造函数中对原型进行设置
    networkCtorFn.SetMember("prototype", networkProto);
    // 为“连接”创建原型方法
    pmovie->CreateFunction(&networkConnectFn, networkClassDef,
                           (void*)NetworkClass::METHOD_Connet);
    networkProto.SetMember("connect", networkConnectFn);
    // 注册带有_global的构造函数
    pmovie->SetVariable("_global.Network", networkCtorFn);
```

现在，可以在虚拟机中实现此类：

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

植入行为主要针对那些具备虚拟机专业知识的开发者，同时为了在虚拟机对象的生命期内进行管理。下列示例说明了行为植入：

```
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)
        {
              // 拦截逻辑（开始）
              OrigFunc.Invoke("call", params.pRetVal, params.pArgsWithThisRef,
                              params.ArgCount + 1);
              // 拦截逻辑（结束）
        }
};
Value funcRealIntercept;
Ptr<FuncRealIntercept> funcRealDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                                FuncRealIntercept(origFuncReal);
pmovie->CreateFunction(&funcRealIntercept, funcRealDef);
obj.SetMember("funcReal", funcRealIntercept);
```

说明：GFx::Value 在自定义函数环境对象中的寿命必须由开发者来维护，因为它为虚拟机对象保留了参考。在.swf (GFx::Movie)失效前，需要开发者对该参考非常清楚。这一要求与所有 GFx::Values 相关的寿命管理要求（保留虚拟机复杂对象的参考）是一致的。

2.4 Direct Access 公共接口

这里介绍的是 Direct Access API 中的公共成员函数。最新信息请参阅 GFx::Value 在线文件。

2.4.1 Object 支持

描述	**公共方法** (假定 'foo' 保存着 Object 在 "_root.foo"中的一个参数)
检查在对象中是否存在成员	`bool has = foo.HasMember("bar");`
在某一对象中检索某一数值	`Value bar;` `bool = foo.GetMember("bar", &bar);`
在某一对象中设置一个数值	`Value val;` `bool = foo.SetMember("bar", val);`

调用某一对象中一个方法	`Value ret;`
	`Value args[N];`
	`bool = foo.Invoke("method", args, N, &ret);` *or*
	`foo.Invoke("method", args, N);`

创建一个对象	Value obj;
	pMovie->CreateObject(&obj);
	…
	bool = foo.SetMember("bar", obj);

创建一个具有层次结构的对象	Value owner, child;
	pMovie->CreateObject(&owner);
	pMovie->CreateObject(&child);
	bool = owner.SetMember("child", child);
	…
	bool = foo.SetMember("owner", owner);

对某一对象中的成员进行迭代	Value::ObjectVisitor v; foo.VisitMembers(&v);
从某一对象中删除某一成员	bool = foo.DeleteMember("bar");

2.4.2 Array 支持

描述	**公共方法** (假定 'bar'存储了 Array 在 "_root.foo.bar"中的一个参数，并且'foo' 保存了 Object 中的一个参数)
确定数组大小	unsigned sz = bar.GetArraySize();
在数组中对某一元素进行检索	Value val; bool = bar.GetElement(idx, &val);
对数组中的某一元素进行设置	Value val; bool = bar.SetElement(idx, val);
重新确定数组大小	bool = bar.SetArraySize(N);

创建一个数组	Value arr;
	pMovie->CreateArray(&arr);
	…
	bool = foo.SetMember("bar", arr);

创建将其他 Complex Object 作为元素的一个数组	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);

对数组的某一范围内的元素进行迭代	列举
	for (UPInt i=0; i <N; i++)
	{ … bool = bar.GetElement(i+idx, &val) … }
	
	使用访问者模式:
	Value::ArrayVisitor v;
	bar.VisitElements(v, idx, N);

清空数组	`bool = bar.ClearElements();`

堆栈操作	`PushBack`
	`Value val;`
	`bool = bar.PushBack(val);`
	
	`PopBack`
	`Value val;`
	`bool = bar.PopBack(&val);` *or* `void bar.PopBack();`

从数组中删除元素	单个元素
	`bool = bar.RemoveElement(idx);`
	
	系列元素
	`bool = bar.RemoveElements(idx, N);`

2.4.3 显示对象支持

描述	**公共方法** (假设 'foo' 在"_root.foo.bar"中保留了显示对象(MovieClip, TextField, Button)的参考。)
获得当前显示信息	`Value::DisplayInfo info;` `bool = foo.GetDisplayInfo(&info);`

设置当前显示信息	`Value::DisplayInfo info;`
	…
	`bool = foo.SetDisplayInfo(info);`

获得当前显示矩阵	`Matrix2_3 mat;` `bool = foo.GetDisplayMatrix(&mat);`

设置当前显示矩阵	`Matrix2_3 mat;`
	…
	`bool = foo.SetDisplayMatrix(mat);`

获得当前颜色转换	`Render::Cxform cxform;` `bool = foo.GetColorTransform(&cxform);`

设置当前颜色转换	`Render::Cxform cxform;`
	…
	`bool = foo.SetColorTransform(cxform);`

2.4.4 影片剪辑支持

描述	**公共方法** (假定 'foo'在"_root.foo"中保留了影片剪辑的参考。)
为影片剪辑添加标识符号	`Value newInstance;`

	`bool = foo.AttachMovie(&newInstance, "SymbolName", "instanceName");`
创建一个空的影片剪辑作为该影片剪辑的孩子	`Value emptyInstance;` `bool = foo.CreateEmptyMovieClip(&emptyInstance, "instanceName");`
按名称播放一个关键帧	`bool = foo.GotoAndPlay("myframe");`
按序号播放一个关键帧	`bool = foo.GotoAndPlay(3);`
按名称停止一个关键帧	`bool = foo.GotoAndStop("myframe");`
按序号停止一个关键帧	`bool = foo.GotoAndStop(3);`

2.4.5 文本字段支持

描述	**公共方法** (假设 'foo' 在"_root.foo"中保留了一个文本字段参考。)
获得原始文本字段文本	`Value text;` `bool = foo.GetText(&text);`

设置原始文本字段文本	`Value text;`
	`…`
	`bool = bar.SetText(text);`

获得 HTML 文本	`Value htmlText;` `bool = bar.GetTextHTML (&htmlText);`

设置 HTML 文本	`Value htmlText;`
	`…`
	`bool = bar.SetTextHTML(htmlText);`