Autodesk®
Scaleform®

Scaleform ゲームコミュニケーションの概要

このドキュメントは、Scaleform 3.1 以降のバージョンを使った、C++、Flash、および ActionScript 間でのコミュニケーションメカニズムについて説明しています。

作者　Mustafa Thamer、Prasad Silva

バージョン　2.01
最終更新日　2010 年 9 月 21 日

Autodesk®
GAMEWARE

Copyright Notice

Autodesk® Scaleform® 4.4

Disclaimer

Autodesk Scaleform の連絡先：

ドキュメント名	Scaleform ゲームコミュニケーションの概要
住所	Autodesk Scaleform Corporation 6305 Ivy Lane, Suite 310 Greenbelt, MD 20770, USA
ウェブサイト	www.scaleform.com
Email	info@scaleform.com
電話	+1 (301) 446-3200
Fax	+1 (301) 446-3199

目次

1 C++、Flash、ActionScript とのインタフェース

Flash®の ActionScript™はインタラクティブなムービーコンテンツを作成できるスクリプト言語です。ボタンのクリック、特定フレームへの到達、あるいはムービーのロードなどのイベントでコードを実行して、ムービーコンテンツの動的な変更、ムービーのフローコントロール、別のムービーの起動、といった処理が可能です。ActionScript はミニゲーム全体を Flash で作成するには十分に強力です。多くのプログラミング言語と同様に、ActionScript は、変数やサブルーチンをサポートするとともに、アイテムやコントロールを表現するオブジェクトで構成されます。

複雑なゲームを開発するにはアプリケーションと Flash コンテンツ間のコミュニケーションが必要です。Autodesk Scaleform®は、ActionScript と C++アプリケーションとの間でのイベント渡しやデータ戻しを実現する、Flash によって与えられる標準的なメカニズムをサポートしています。また、ActionScript 変数、アレイ、オブジェクトを直接操作できるほか、ActionScript サブルーチンを直接コールできる利便性の高い C++インタフェースも装備しています。

このドキュメントでは、C++と Flash 間のコミュニケーションに利用できる複数のメカニズムを説明します。利用できるオプションを以下に示します。

ActionScript → C++	**C++ → ActionScript**
FSCommand ストリングベースのシンプルな関数実行、戻り値なし、非推奨	**GFx::Movie::Get/SetVariable** ActionScript 内のデータアクセス、ストリングパスを使用
ExternalInterface 自由度の高い引数処理、戻り値あり、推奨	**GFx::Movie::Invoke** ActionScript 内の関数コール、ストリングパスを使用
Direct Access API データアクセスと関数アクセスのオブジェクトに対する直接リファレンスとして GFx::Value を使用、高性能。GFx::FunctionHandler を使用して ActionScript VM 内の直接関数コールバックを設定。	

1.1 ActionScript から C++

Scaleform は C++アプリケーションに対して、ActionScript からのイベントを受信する 2 種類のメカニズム、*FSCommand* と *ExternalInterface* を提供しています。これらメカニズムは、スタンダード ActionScript API の一部であり、詳細は Flash ドキュメントに記載されています。FSCommand と ExternalInterface の両方は、イベント通知を受信するために、C++イベントハンドラを Scaleform に登録します。これらハンドラは Scaleform ステートとして登録されるため、要求機能に応じて、GFx::Loader、GFx::MovieDef、あるいは GFx::Movie 上に登録することが可能です。Scaleform ステートの詳細は Scaleform ドキュメントを参照してください。

FSCommand イベントは ActionScript の「fscommand」関数でトリガーされます。fscommand 関数は、ActionScript から 2 個のストリング引数のみを渡すことができ、ひとつはコマンド用でひとつは単一データ引数用です。これらストリング値は、2 個の const char ポインタとして、C++ハンドラによって受信されます。ActionScript の fscommand 関数が戻り値をサポートしていないため、C++の fscommand ハンドラは残念ながら値を戻しません、

ActionScript が *flash.external.ExternalInterface.call* 関数をコールすると ExternalInterface イベントがトリガーされます。このインタフェースは ActionScript 値の任意のリストとコマンド名のみを渡します。C++ ExternalInterface ハンドラは、コマンド名の const char ポインタと、ランタイムから渡された ActionScript 値に対応する GFx::Value 引数のアレイを受信します。ActionScript の ExternalInterface.call メソッドが戻り値をサポートしているため、C++ ExternalInterface もランタイムに GFx::Value を戻します。ActionScript の ExternalInterface クラスは、ActionScript と Flash Player コンテナ（この場合では Scaleform を使ったアプリケーション）との正攻法的なコミュニケーションを実現するアプリケーションプログラミングインタフェースで、External API の一部です。

自由度が限られている FSCommands は、ExternalInterface の登場により現在では古いものとなっているため、現在ではその使用は推奨していません。説明に漏れのないようにするためと従来機能のサポートを目的に記述しています。

また、FSCommands と ExternalInterface は、Direct Access API および GFx::FunctionHandler インタフェースの登場により現在では古いものとなっています。このインタフェースは、C++メソッドへの直接コールバックが通常関数として ActionScript VM 内部に割り当てられることを許可します。この関数が ActionScript 内で呼び出されると、次に C++コールバックを呼び出します。
GFx::FunctionHandler の詳細は Direct Access API セクションを参照してください。

1.1.1FSCommand コールバック

ActionScript の fscommand 関数はコマンドとデータ引数をホストアプリケーションに渡します。以下に ActionScript での一般的な使い方を示します。

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

boolean や integer など、fscommand に与えられるすべての非ストリング引数はストリングに変換されます。

ActionScript の fscommand コールは GFx FSCommand ハンドラに 2 個のストリングを渡します。アプリケーションは、GFx::FSCommandHandler をサブクラスにし、かつ、そのクラスのインスタンスを共有ステートとして登録することで、fscommand ハンドラを GFx::Loader か GFx::MovieDef のいずれか、または独立した GFx::Movie オブジェクトに登録します。ステートバッグ間の設定は GFx::Loader -> GFx::MovieDef -> GFx::Movie に示すような委任を意図しているということに留意してください。それらはいずれかの最新のインスタンスのひとつによって上書きされます。このことは、特定の GFx::Movie にそれ自身の GFx::RenderConfig（たとえば、別々の EdgeAA コントロール）または GFx::Translator（別の言語に翻訳されるように）を持たせたい場合に、オブジェクト上での設定が可能なほか、GFx::Loader など委任チェーン内で高位のオブジェクト上に適用されたステートよりも適用したステートが優先します。コマンドハンドラーが GFx::Movie 上に設定されている場合、ハンドラーはそのムービーインスタンス内で呼び出された FSCommand コールのみのコールバックを受信します。GFxPlayerTiny はこのプロセスのサンプルです（「FxPlayerFSCommandHandler」で検索してください）。

fscommand ハンドラセットアップの例を以下に示します。ハンドラは GFx::FSCommandHandler から取得しています。

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

Callback メソッドは、ActionScript 内の fscommand に渡される 2 個の引数と、fscommand で呼び出された特定のムービーインスタンスへのポインタを受信します。当社のカスタムハンドラーは、各 fscommand イベントをデバッグコンソールに単純に出力します。

次に、GFx::Loader オブジェクトを作成したあとでハンドラーを登録します。

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

GFx::Loader にハンドラーを登録すると、GFx::Movie および GFx::MovieDef はそれぞれ、このハンドラーを継承するようになります。このデフォルト設定を上書きするには、個々のムービーインスタンス上から SetFSCommandHandler をコールします。

一般に、C++イベントハンドラはノンブロッキングでなければならず、かつ、できるだけ速やかにコーラーに戻らなければなりません。イベントハンドラは通常、Advance コールまたは Invoke コール中にのみコールされます。

1.1.2外部インタフェース API

Flash の ExternalInterface.call メソッドは fscommand と似ていますが、より自由度の高い引数処理と戻り値を持つことから、こちらの使用を推奨しています。ExternalInterface ハンドラの登録は fscommand ハンドラの登録と同様です。以下は、数値とストリング変数をプリント出力する ExternalInterface ハンドラの一例です。

```
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);
    }
};
```

ハンドラを実装すると、以下に示すように、ハンドラのインスタンスを GFx::Loader に登録することができるようになります。

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

ここで、コールバックハンドラは、ActionScript 内の ExternalInterface コールによってトリガーされます。

1.1.2.1 FxDelegate

上記の ExternalInterface ハンドラは、きわめて基本的なコールバック機能を与えます。実際、ExternalInterface コールバックが特定の methodName でトリガーされるときに、アプリケーションはコールされる特定の関数を登録したい場合があります。このことは、対応する methodName で関数を登録可能で、次にハッシュテーブル（または類似のもの）を使って参照し対応する GFx::Value 引数を使って実行する、より先進的なコールバックシステムが必要であることを意味します。そのようなシステムは当社のサンプルに含まれており、FxDelegate と呼ばれます（Apps\Samples\GameDelegate\FxGameDelegate.h を参照）。

FxDelegate はそのような期待に応えるもので、GFx::ExternalInterface に由来します。FxDelegate は、アプリが methodName で登録されたアプリ自身のコールバックハンドラを委任としてインストールできるようにする、Register/Unregister Handlers 関数を備えています。

FxDelegate 使い方の例としては、minimap 実装に FxDelegate を活用した当社の HUD キットを参照してください。コールバック関数を FxDelegate に登録する minimap デモから、アプリケーションコードを示します。

```
// *** 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 は完全に機能的であり、より洗練されたコールバック処理と登録システムの一例を与えてくれます。FxDelegate はユーザーにとってのスターティングポイントであり、デベロッパはできるだけ、その動作の詳細と、必要に応じたカスタマイズ方法および拡張方法を理解するようにしてください。

1.1.2.2 ActionScript から External Interface を使用する

ActionScript では External Interface コールは以下のコードで開始します。

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

ここで、「foo」はメソッド名、「arg」は基本タイプの引数です。0 個または 1 個以上のパラメータをカンマで区切って指定することができます。ExternalInterface API の詳細は Flash ドキュメントを参照してください。

ExternalInterface が上記コードのようにインポートされない場合は、ExternalInterface コールは完全修飾しなければなりません。

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

1.1.2.3 ExternalInterface.addCallback

ExternalInterface.addCallback 関数は別名を使って ActionScript をグローバルに登録します。そのため、C++からパスプレフィクスなしで登録済みメソッドを読み出すことが可能です。この関数は、メソッドの登録と、単一の別名でのコンテクスト（「this」オブジェクト）のコールをサポートします。登録された別名は C++の GFx::Movie::Invoke()メソッドからコールできます。

例：

AS コード：

```
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!";
}
```

別名の呼び出しによって txtField に「this」を設定した状態で、SetText ActionScript 関数をコールすることが可能です。

C++コード:

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

このコードは、「txtField.SetText」をトレースし、テキスト「INVOKED!」を「txtField」テキストフィールドに設定します。GFx::Movie::Invoke()の使い方はセクション 1.2.2 を参照してください。

1.2 C++から ActionScript

前のセクションでは ActionScript から C++をコールする方法について説明しました。このセクションでは、C++プログラムから Flash ムービーへのやりとりを実現する Scaleform 関数を使って、逆方向のやりとりについて説明します。Scaleform は ActionScript 変数（シンプルタイプ、コンプレックスタイプ、アレイ）の直接的な取得および設定のほか、ActionScript サブルーチンの呼び出しを目的とする C++関数をサポートしています。

Direct Access API は、ActionScript へのアクセスおよび操作を C++から実現する利便性および効率の高いインタフェースです。このインタフェースの詳細については Direct Access API セクションを参照してください。

1.2.1ActionScript 変数の操作

Scaleform は ActionScript 変数を直接操作する GetVariable および SetVariable をサポートしています。性能が必要な場合は、変数とオブジェクト属性を変更するより効率的な Direct Access API のセクション 2 を参照してください。Set/GetVariable は性能の点において推奨しませんが、Direct Access API よりも使いやすい場合もあります。たとえば、アプリケーションのライフタイムを通じて単一の値を一回設定するためだけに、とくに深いネストレベルにターゲットがある場合、Direct Access コールの GFx::Value::SetMember を使う必要はありません。また、ActionScript へのリファレンスの取得は、通常は GetVariable を使います。GetVariable は GFx::Value を取得するために一回コールされ、その後の Direct Access 動作で使われます。

GetVariable と SetVariable を使って ActionScript カウンタをインクリメントするコード例を以下に示します。

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

GetVariableDouble は_root.counter 変数の値を C++の Double タイプに自動的に変換して返します。最初の時点では変数は存在せず、GetVariableDouble はゼロを返します。次の行でカウンタをインクリメントし、SetVariable を使って新しい値が_root.counter に格納されます。GFx::Movie のオンラインドキュメントに GetVariable and SetVariable のバリエーションを示します。

SetVariable には、割り当てが「sticky」であることを宣言する、タイプ GFx::Movie::SetVarType の 3 番目の引数がオプションとして用意されています。このオプションは、割り当てられる変数が Flash タイムライン上でまだ作成されていないときに有用です。たとえばムービーが 3 フレーム目になるまでテキストフィールド_root.mytextfield は作成されていないと想定してください。もし、ムービーが作成された直後である 1 フレーム目で SetVariable(“_root.mytextfield.text”, “testing”, SV_Normal)がコールされたとすると、割り当ては効果を及ぼしません。しかし、SV_Sticky（デフォルト値）を使ってコールが行われると、要求はキューイングされ、3 フレーム目で

_root.mytextfield.text 値が有効になった時点で適用されます。このような動作によって C++からムービーを簡単に初期化することができます。ただし一般に、SV_Normal のほうが SV_Sticky よりも効率的であり、可能な限り SV_ Normal を使うようにしてください。

データのアレイにアクセスするために、Scaleform では関数 SetVariableArray and GetVariableArray が用意されています。

SetVariableArray は指定範囲のアレイエレメントを指定タイプのデータアイテムによって設定します。アレイが存在しない場合は作成されます。アレイがすでに存在しているものの、十分なアイテム数で構成されていない場合は、適切にリサイズされます。ただし、アレイの現在のサイズ未満のエレメント数を設定すると、アレイのリサイズは起こりません。GFx::Movie::SetVariableArraySize はアレイのサイズを設定します。前のエレメント数よりも少ないエレメントを既存アレイに設定する場合に有用です。

SetVariableArray を使って AS 内のストリングアレイを設定するコードの一例を以下に示します。

```
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);
```

下記は上の例のバリエーションで、ワイドストリングを使ったコードの一例です。

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

GetVariableArray メソッドは、AS アレイの結果を使って、与えれたデータバッファをフィルします。バッファは要求されたアイテム数を保持できる十分な大きさがなければなりません。

1.2.2ActionScript サブルーチンの実行

ActionScript 変数を変更したことに加えて、GFx::Movie::Invoke()メソッドを使って ActionScript メソッドの呼び出しが可能になりました。この機能は、アニメーションのトリガー、現在のフレームの変更、UI コントロールステートのプログラム的な変更、新ボタンまたはテキストなど UI コンテンツの動的な生成など、より複雑な処理の実行に効果的です。性能が必要な場合は、より効率的な呼び出し方法とアクションフローのコントロールについて、Direct Access API セクションを参照してください。

例：
射程範囲からの相対位置としてスライダーグリップ位置を設定する ActionScript を以下に示します。

_root レベル内に「mySlider」オブジェクトが存在すると仮定します。SetSliderPos は「mySlider」オブジェクト内部で次のように定義されます。

AS コード：

```
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;
}
```

ここで「 gripClip」は、「mySlider」内のネストされたムービークリップのひとつで、pos は rangeMin/Max の範囲内で常に有効です。

ユーザーアプリケーションからは Invoke 関数は次のように使います。

C++コード：

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

Invoke は、メソッドが実際に呼び出されたときは true を返し、そうでなければ false を返します。

ActionScript 関数を呼び出すときには、その ActionScript がロードされていることを確認する必要があります。Invoke での共通的な誤りのひとつが利用できない ActionScript ルーチンをコールしてしまうことで、Scaleform ログにエラーが記録されます。ActionScript ルーチンは、そのルーチンに関連付けられるフレームの再生が完了するるまで、あるいは、そのルーチンに関連付けられるネストオブジェクトのロードが完了するまで、利用可能にはなりません。1 フレーム目中のすべての ActionScript コードは、GFx::Movie::Advance への最初のコールの時点、または initFirstFrame を true にセットした状態で GFx::MovieDef::CreateInstance がコールされた時点で、即座に利用可能になります。

Invoke とともに、GFx::Movie::IsAvailable()メソッドは、一般に AS 関数が呼び出し前に存在していることを確認するために使います。

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

このコード例では Invoke の「printf」スタイルを使用しています。この場合、Invoke は、フォーマットストリングで記述される可変長引数リストとして、引数を取り込みます。関数の他のバージョンは GFx::Value を使って非ストリング引数を効率的に処理します。たとえば、

```
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 は、可変長引数リストへのポインタをアプリケーションに与えるために va_list 引数を取り込む以外は、Invoke と同じです。Invoke と InvokeArgs の関係は、printf と vprintf の関係に似ています。

1.2.3パス

ユーザーアプリケーション内から Flash エレメントをアクセスする場合、一般にオブジェクトの参照には完全修飾パスが使われます。

以下は完全修飾パスに依存した GFx::Movie 関数です。

```
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, ...)
```

ネストオブジェクトパスを正しく解決するために、すべての親ムービークリップはユニークなインスタンス名を持っていなければなりません。ネストオブジェクト名はドット「.」で区切り、以下のように大文字と小文字を区別します。

以下の例では、インスタンス名「clip2」のムービークリップが、メインステージ上にある「clip1」という名前のムービー中にネストされていると想定しています。

Main Stage -> clip1 -> clip2
clip1 はメインステージ内またはメインステージ上
clip2 は clip1 内

有効なパスは、
```
"clip1.clip2"
"_root.clip1.clip2"
"_level0.clip1.clip2"
```

無効なパスは、
```
"Clip1.clip2"  <- 大文字と小文字が異なる。「Clip1」は小文字にする。
"clip2"        <-  親「Clip1」がない。パス名の誤り。
```

"_root" と"_level0" の名称は ActionScript 2 にはオプションで、これらは特定のベースレベルのルックアップを強制的に行うために使用できます。しかし、ActionScript 3 ではルックアップはステージレベルで起こりますので、これらは必要です。後方互換性を持たせるために、ActionScript 3 には次の名称でルートにアクセスできるようにエイリアスを備えています。_root、_level0、root、level0。別の SWF ファイルをレベルにロードする場合、_root は現在のレベルのベースを参照しますが、上記のシンタックスを使用して_levelN を指定すると特定のレベルを選択できます。

備考：
– Flash Studio の Test Movie（Ctrl-Enter）環境では、Ctrl-Alt-V（変数）を押下することで、オブジェクトへのパスと変数を確認することができます

– Scene 1 リンクから始まる Flash Studio 内のタイムラインシーケンスは、ターゲットパスを指示しません。オブジェクトパス内で実際には使用されていない一部のエレメントがリストされます。Ctrl-Alt-V（変数）ポップアップを使って、実際の有効パスを確認してください。

– loadMovie コマンドを使って複数ムービー中に複数ムービーをロードする場合、オブジェクトのレベル変化に起因して問題が発生する場合があります。_level0 に存在するオブジェクトは、_level1 または_level2（どの Level にロードするかに依存）になるか、対象の MovieClip 内になります。他のレイヤー上の Object を参照するには、単純に_levelN.objectName または _levelN.objectPath.objectName を使用します。ここで N はレベル番号です。

このセクションの多くの項目は以下のチュートリアルから抜き出したものです。一読を推奨します。

1. Paths to Objects and Variables 、作成 Jesse Stratford
2. Advanced Pathing 、作成 Jesse Stratford

2 Direct Access API

Scaleform 3.1 以後のバージョンに搭載された GFx::Value Direct Access サポートによって、AS ランタイムとのゲームコミュニケーションが大きく改善されました。Action Script コミュニケーション API が簡単なものから高度なものへと拡張された結果、複雑なオブジェクト（およびそれらオブジェクトに含まれる個々のメンバー）を効率的に設定およびクエリすることが可能になっています。たとえば、UI とゲームとの間で、ネストされたヘテロジニアスなデータ構造を、双方向でやりとりすることができます。Direct Access API の使い方の例は HUD Kit を参照してください。minimap 上の大量の Flash オブジェクトを更新したときに得られる性能の改善効果が示されています。

Direct Access API で GFx::Values は、シンプルな ActionScript タイプに加えて、Object、Array、および Display Object への各リファレンスを格納できるようになりました。Display Objects は、Object タイプの特別な場合で、ステージ上の複数エンティティ（MovieClip、Button、TextFields）に対応します。GFx::Value クラス API は、それら値およびメンバーの設定および取得を行う関数と、アレイを取り扱う関数を一通り備えています。GFx::Movie クラス API にはオブジェクトとアレイを作成するメソッドが追加されています。その詳細は GFx::Value リファレンスを参照してください。

Direct Access API によって、アプリケーションは ActionScript 内のオブジェクトに（GFx::Value タイプの）C++変数を直接バインドすることが可能になりました。バインドまたはリファレンスが完了すると、変数を使って ActionScript オブジェクトを簡単かつ効率的に変更できるようになります。この変更の前は、ユーザーは GFx::Movie とストリングパスを指定して AS オブジェクトをアクセスしなければなりませんでした。この方法はパスの構文解析と AS オブジェクトの検索が必要で、性能的に課題がありました。直接アクセスオブジェクトを使うと、構文解析と検索に要していたオーバーヘッドが排除されます。

GFx::Movie レベルでのみ過去に利用できた多くの動作は、GFx::Value タイプによって表される ActionScript（AS）オブジェクトに直接適用できるようになりました。これは読みやすく効率的なコードの開発に結びつくでしょう。たとえば、ルートにある「foo」と呼ばれるオブジェクト上のメソッドをコールするには、Movie の Invoke コールを使うのではなく、オブジェクトへのリファレンスを取得して Invoke を直接コールします。

1. GFx::Movie Invoke を使う方法（あまり効率的ではない）

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

2. Direct Access Invoke を使う方法（より効率的）

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

Invoke に加えて、AS オブジェクトの値のチェック、ゲット、およびセットへのコールは、GFx::Value::HasMember、GetMember、および SetMember を使用して Direct Access API から行えるようになりました。

movieClip 上のテキストフィールド（GFx::Value 内に格納）を更新する GetMember の使用法の一例を示します。

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

上記の例では、movieClip オブジェクトの textfield メンバーを最初に取得し、次に関数 SetText を使ってそのテキスト値を更新しています。

2.1 オブジェクトとアレイ

Direct Access API でサポートされたもうひとつの重要な機能が、AS オブジェクトまたはオブジェクト階層の作成機能です。この方法を使って、任意深さと構造を持ったオブジェクトの作成と管理が C++から行えます。たとえば、次のコード snippet を使うと、片方が一方の子オブジェクトとなる階層構造の 2 個のオブジェクトが作成できます。

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

CreateObject は ActionScript オブジェクトのインスタンスを作成します。また、特定クラスタイプのインタンスが必要な場合、CreateObject にはオプションの完全修飾クラス名が渡されます。たとえば、

```
Value mat, params[6];
// (set params[0..6] to matrix data) …
pMovie->CreateObject(&mat, "flash.geom.Matrix", params, 6);
```

Object と Array の両方を使うより複雑な状況を考えてみます。コンプレックスオブジェクトエレメントを持つアレイを作成するコードを以下に示します。

```
// (Assuming pMovie is a GFx::Movie*):
Value owner, childArr, childObj;
pMovie->CreateArray(&owner);           // create parent array
pMovie->CreateArray(&childArr);        // create child array
pMovie->CreateObject(&childObj);       // create child object
bool = owner.SetElement(0, childArr);  //set parent[0] to child array
bool = owner.SetElement(1, childObj); // set parent[1]to child object
```

```
…
bool = foo.SetMember("owner", owner); // later, set the 'owner' variable in
                                      // object foo, to the parent object
```

関数 GFx::Value::VisitMembers()はオブジェクトのパブリックメンバーのトラバースに使用します。関数は、上書きされるべき単純な Visit コールバックを有する ObjectVisitor クラスのインスタンスを取り込みます。この関数はオブジェクトタイプ（Array と DisplayObject を含む）に対してのみ有効です。クラスインスタンスを完全に調べるために *VisitMembers* を使うことはできませんので注意してください。メソッドはプロトタイプ内で動作するため枚挙可能（enumerable）ではありません。メンバーの可視性と属性を強制するには、ActionScript 内の ASSetPropFlags メソッドを使います。

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

属性（getter/setter）は、ASSetPropFlags を使ったとしても、VisitMembers 経由で enumerable にはなりません。ただし、これらは Direct Access インタフェース経由でアクセス可能で、VT_Object として返ります。関数も VT_Object として返ります（その関数が ASSetPropFlags 経由で enumerable になった場合）。

2.2 ディスプレイオブジェクト

Direct Access API は、DisplayObject と呼ぶ Object タイプの特殊なケースをサポートしていて、MovieClip、Button、および TextField のようにステージ上の複数のエンティティに対応します。それらの表示属性にアクセスするために、カスタム DisplayInfo API が提供されます。DisplayInfo API を使用すると、アルファ、回転、可視性、ポジション、オフセット、倍率など、DisplayObject 上の属性を簡単に設定することが可能です。これらディスプレイ属性は SetMember 関数を使って設定することもできますが、オブジェクトのディスプレイ属性を直接変更する SetDisplayInfo コールのほうが高速です。これら両方のメソッドは、対象オブジェクトを直接操作しない GFx::Movie::SetVariable をコールするよりも高速です。

SetDisplayInfo メソッドを使って movieclip インスタンスのポジションと向きを変更するコードを以下に示します。

```
// Assumes movieClip is a GFx::Value*
Value::DisplayInfo info;
PointF pt(100,100);
info.SetRotation(90);
info.SetPosition(pt.x, pt.y);
movieClip.SetDisplayInfo(info);
```

2.3 関数オブジェクト

ActionScript2 Virtual Machine（AS2 VM）の関数は基本オブジェクトと同様です。これら関数オブジェクトはメンバーに割り当てることが可能であり、使用条件に応じてより詳しい内部情報を与えることがあります。GFx::Movie::CreateFunction メソッドを使うと、C++コールバックオブジェクトをラップする関数オブジェクトを作成することができます。CreateFunction が返す関数オブジェクトは、VM 内の任意のオブジェクトのメンバーとして割り当てることができます。この AS 関数が呼び出されたとき、C++コールバックが次に呼び出されます。この動作によって、VM からの直接コールバックを、委任のオーバーヘッドなしで、デベロッパ自身のコールバックハンドラーに登録することができます（fscommand または ExternalInterface を経由）。

カスタムコールバックの作成と、AS オブジェクトのメンバーへの割り当てを行うコードの一例を示します。

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

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

このメソッドは ActionScript 内で呼び出せます（_root timeline）。

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

Call()メソッドに渡される Param 構造は、以下の内容で構成されます。

- pRetVal: 戻り値としてコーラーに返されるであろう GFx::Value へのポインタ。コンプレックスオブジェクトを返す場合、このポインタを pMovie->CreateObject()か CreateArray()に渡してコンテナを作成します。プリミティブタイプの場合（Number、Boolean など）、適切なセッターをコールしてください。
- pMovie: 関数を呼び出したムービーへのポインタ。
- pThis: コーラーコンテクスト。コールしたコンテクストが存在しないか無効の場合は不定になることがあります。
- ArgCount: コールバックに渡された引数の個数。
- pArgs: 関数コールバックに渡された引数を表現する GFx::Values のアレイ。[ ] 演算子を使って個々の引数にアクセスします。
  例： `Value firstArg = params.pArgs[0];`

- pArgsWithThisRef: 引数のアレイと、あらかじめペンディングされたコーリングコンテクスト。他の関数オブジェクトをチェーンするときに使います（カスタムビヘイビアをインジェクションするには以下の例を参照）。
- pUserData: 関数オブジェクトが登録されたときのカスタムデータ。このデータは、別のハンドラーメソッドへのコールバックのカスタム委任に有用です。

関数オブジェクトは、VM 内の既存関数の上書きに使えるとともに、既存関数本体の開始時点または終了時点でのカスタムビヘイビアのインジェクションにも使えます。また関数オブジェクトは、クラス定義を伴った、レジスタスタティックメソッドまたはレジスタインスタンスメソッドになり得ます。そのためカスタムクラスの定義が拡張されます。以下は VM 内でインスタンス化可能なサンプルクラスを作成するコード例です。

```
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:
            // Custom logic
            break;
            case METHOD_Connet:
            // Custom logic
            break;
        }
    }
};
Ptr<NetworkClass> networkClassDef = *SF_HEAP_NEW(Memory::GetGlobalHeap())
                                                    NetworkClass();
// Create the constructor function
pmovie->CreateFunction(&networkCtorFn, networkClassDef,
                       (void*)NetworkClass::METHOD_Ctor);
// Create the prototype object
pmovie->CreateObject(&networkProto);
// Set the prototype on the constructor function
networkCtorFn.SetMember("prototype", networkProto);
```

```
// Create the prototype method for 'connect'
pmovie->CreateFunction(&networkConnectFn, networkClassDef,
                       (void*)NetworkClass::METHOD_Connet);
networkProto.SetMember("connect", networkConnectFn);
// Register the constructor function with _global
pmovie->SetVariable("_global.Network", networkCtorFn);
```

クラスは VM 内でインスタンス化されます。

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

ビヘイビアインジェクションは、VM の専門知識を有するとともに、VM オブジェクトのライフタイム管理を行うデベロッパー向けです。以下にビヘイビアインジェクションの例を示します。

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

備考：カスタム関数コンテクストオブジェクト内部で保持されている GFx::Value のライフタイムは、VM オブジェクトへのリファレンスを保持しているため、デベロッパによって維持されなければなりません。デベロッパは swf（GFx::Movie）が停止する前に、リファレンスをクリアしなければなりません。この要件は、VM から得られるコンプレックスオブジェクトへのリファレンスを保持するすべての GFx::Values に関連するライフタイム管理要件と一貫性があります。

2.4 Direct Access Public インタフェース

ここでは Direct Access API 内のパブリックメンバー関数を説明します。最新情報は GFx::Value のオンラインドキュメントを参照してください。

2.4.1オブジェクトのサポート

説明	**パブリックメソッド** （"_root.foo" 上のオブジェクトへのリファレンスを 'foo' が保持していると仮定）
オブジェクト内にメンバーが存在するか確認する	`bool has = foo.HasMember("bar");`
オブジェクトから値を読み出す	`Value bar;` `bool = foo.GetMember("bar", &bar);`
オブジェクトの値を設定する	`Value val;` `bool = foo.SetMember("bar", val);`

オブジェクトのメソッドをコールする	`Value ret;`
	`Value args[N];`
	`bool = foo.Invoke("method", args, N, &ret);` *or*
	`foo.Invoke("method", args, N);`

オブジェクトを作成する	`Value obj;`
	`pMovie->CreateObject(&obj);`
	…
	`bool = foo.SetMember("bar", obj);`

オブジェクト階層を作成する	`Value owner, child;`
	`pMovie->CreateObject(&owner);`
	`pMovie->CreateObject(&child);`
	`bool = owner.SetMember("child", child);`
	…
	`bool = foo.SetMember("owner", owner);`

オブジェクトのメンバー同士で反復する	`Value::ObjectVisitor v;` `foo.VisitMembers(&v);`
オブジェクトからメンバーを削除する	`bool = foo.DeleteMember("bar");`

2.4.2アレイのサポート

説明	**パブリックメソッド** （"_root.foo.bar" 上のアレイへのリファレンスを 'bar' が保持し、 オブジェクトへのリファレンスを 'foo' が保持していると仮定）
アレイサイズを決定する	`unsigned sz = bar.GetArraySize();`

アレイからエレメントを読み出す	`Value val;` `bool = bar.GetElement(idx, &val);`
アレイのエレメントを設定する	`Value val;` `bool = bar.SetElement(idx, val);`
アレイをリサイズする	`bool = bar.SetArraySize(N);`

アレイを作成する	`Value arr;`
	`pMovie->CreateArray(&arr);`
	`…`
	`bool = foo.SetMember("bar", arr);`

別のコンプレックスオブジェクトをエレメントとして含むアレイを作成する	`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);`

アレイ内のエレメントの範囲を対象に反復する	`Enumeration`
	`for (UPInt i=0; i <N; i++)`
	`{ … bool = bar.GetElement(i+idx, &val) … }`
	
	`Using a visitor pattern:`
	`Value::ArrayVisitor v;`
	`bar.VisitElements(v, idx, N);`

アレイをクリアする	`bool = bar.ClearElements();`

スタック操作	`PushBack`
	`Value val;`
	`bool = bar.PushBack(val);`
	
	`PopBack`
	`Value val;`
	`bool = bar.PopBack(&val);` *or* `void bar.PopBack();`

アレイからエレメントを削除する	`Single element`
	`bool = bar.RemoveElement(idx);`
	
	`Series of elements`
	`bool = bar.RemoveElements(idx, N);`

2.4.3ディスプレイオブジェクトのサポート

説明	**パブリックメソッド** （"_root.foo.bar" 上のディスプレイオブジェクト（MovieClip、TextField、Button）へのリファレンスを 'foo' が保持していると仮定）

現在のディスプレイ情報を取得する	`Value::DisplayInfo info;` `bool = foo.GetDisplayInfo(&info);`

現在のディスプレイ情報を設定する	`Value::DisplayInfo info;`
	`…`
	`bool = foo.SetDisplayInfo(info);`

現在のディスプレイマトリックスを取得する	`Matrix2_3 mat;` `bool = foo.GetDisplayMatrix(&mat);`

現在のディスプレイマトリックスを設定する	`Matrix2_3 mat;`
	`…`
	`bool = foo.SetDisplayMatrix(mat);`

現在の色変換を取得する	`Render::Cxform cxform;` `bool = foo.GetColorTransform(&cxform);`

現在の色変換を設定する	`Render::Cxform cxform;`
	`…`
	`bool = foo.SetColorTransform(cxform);`

2.4.4MovieClip のサポート

説明	**パブリックメソッド** （"_root.foo" 上の MovieClip へのリファレンスを 'foo' が保持していると仮定）
MovieClip にシンボルインスタンスを添付する	`Value newInstance;` `bool = foo.AttachMovie(&newInstance, "SymbolName", "instanceName");`
MovieClip の子として空の MovieClip を作成する	`Value emptyInstance;` `bool = foo.CreateEmptyMovieClip(&emptyInstance, "instanceName");`
名前で keyframe を再生する	`bool = foo.GotoAndPlay("myframe");`
番号で keyframe を再生する	`bool = foo.GotoAndPlay(3);`
名前で keyframe を停止する	`bool = foo.GotoAndStop("myframe");`
番号で keyframe を停止する	`bool = foo.GotoAndStop(3);`

2.4.5Textfield のサポート

説明	**パブリックメソッド** （"_root.foo" 上の Textfield へのリファレンスを 'foo' が保持していると仮定）
raw textField テキストを取得する	`Value text;` `bool = foo.GetText(&text);`

raw textField テキストを設定する	`Value text;`
	`…`
	`bool = bar.SetText(text);`

HTML テキストを取得する	Value htmlText; bool = bar.GetTextHTML (&htmlText);
HTML テキストを設定する	Value htmlText; … bool = bar.SetTextHTML(htmlText);