X3 插件基础模块
使用说明书

编写人：张云贵

编写日期：2010-9-30

文档修订记录

版本编号	说明：如形成文件、变更内容和变更范围	日期	执行人	批准日期	批准人
V0.1	形成文件	2010-9-28	张云贵	YYYY-MM-DD	
V0.2	在附件中增加内容	2010-11-8	张云贵		

1. 重用模块概述

X3 插件基础模块是“X3 插件框架”的最底层独立模块，用于形成其他插件模块。X3 插件框架的设计目标是汇集各种常用的轻量级 C++插件通用模块，其插件既能灵活组合到各种系统，又能单独拆开使用。“X3”是开发代号，不是版本号。

X3 插件基础模块用于开发具有统一接口标准的 C++插件模块，使其具有 COM 组件的多种特点（接口与实现分离、一个实现类支持多个接口、引用计数管理、模块独立编译、透明部署、模块可替换等），同时简单易用、轻量化，这样开发人员就能简便快捷的开发出可重用、易于测试的规范模块。

本模块所提供的插件机制是 2008 年 3 月从飞腾创艺插件 SDK 中经过大量的简化而来，并结合常见的桌面应用开发的实际需要改编，经过了十几个系统的实际成功验证。

1.1.1. 插件原理

插件原理如图 1 所示，所有插件都是普通 DLL，通过在工程内包含辅助文件自动实现了统一的导出函数，通过该导出函数就能获取到在该插件内所实现的所有接口的信息及对象创建函数地址；主程序使用插件管理器（PluginManager）加载这些 DLL，插件管理器通过这个导出函数将各个类 ID、对象创建函数地址统一管理起来，从而在插件管理器的中介作用下让各个插件能相互使用各种接口函数。

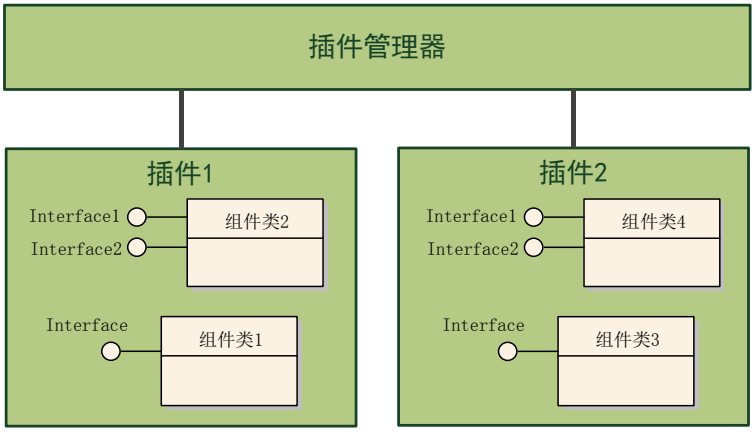

图 1

1.1.2. 本模块的特点

采用本模块提供的机制和代码文件来开发插件模块，具有下列主要特点：

- **接口定义简单灵活**

  采用普通的 C++接口，即由纯虚函数组成的结构体，不需要特殊的基类，不需要宏和 UUID 申明；同时可以使用 C++的各种变量类型，不受 COM 接口那样的约束。例如下面的接口 Ix_定义：

```
interface Ix_Example
{
    virtual void Foo() = 0;
    virtual void* GetData(std::vector<int>& items) = 0;
};
```

- **接口与实现分离**

  对外提供接口文件，在插件内部用类来实现一个或多个接口，不需要对外导出该类或暴露实现细节。这样还有一个好处是只有约定了接口就可以让多个模块并行开发，模块相互之间不存在编译依赖（不需要其他插件的 LIB 等文件），这可用于测试驱动开发模式。

- **多接口转换、引用计数管理**

  采用智能指针类来管理接口的引用计数及生命期，可从一个接口动态转换为另一个接口（内部采用 C++的 RTTI 机制动态转换），可以区分插件内部的接口引用和插件外部的接口引用。

- **模块透明部署**

  一个模块只需要使用其他模块的接口，不需要关心该接口是在哪个插件中实现的。可以根据需要将各个实现类进行合并或拆分，使其分布到不同插件中，而接口使用者不受影响。另外，插件部署于哪个目录也不影响插件接口的使用。

- **模块可替换、可扩展**

可根据需要替换某个插件，只有该插件实现了相同的接口，即使内部功能不相同，这样就实现了插件可替换、按需组合。通过在新的插件中支持更多的接口，可扩展更多的功能。可以在新插件中局部替换原有插件的某些接口或部分函数，实现重用和扩展。

- **线程安全性**

  本插件机制所提供的内部实现文件考虑了线程安全性，允许多线程访问而不冲突，同时采用的是轻量级的锁定机制（计数原子锁定），运行开销很小。

- **跨版本**

  允许不同版本的 VC++开发的插件相互调用对方的接口，虽然实际中一般不需要这样做。由于没有采用 VC++特殊的编译指令，因此容易移植到其他开发平台下。

2. 重用模块使用说明

2.1. 编译运行环境

本插件机制采用 C++实现，用到了 C++的 RTTI 机制和少量 Windows API 函数，没有使用 MFC、ATL、STL，没有使用 LIB 文件，外部依赖文件少，没有使用 VC++特殊编译指令。

编译环境为 Visual C++ 6.0/2003/2005/2008/2010，其他 C++开发平台下待测试。

运行环境为 Windows 2000 及以后的操作系统，Windows 98 需要安装 UNICODE 支持包。

2.2. 使用方法--开发插件

2.2.1. 定义接口

在 H 文件中定义接口（即纯虚函数组成的结构体），在一个 H 文件中可定义一个或多个接口。例如下面的 Ix_Example.h：

```
#pragma once

interface Ix_Example
{
    virtual void Foo() = 0;
    virtual void* GetData(std::vector<int>& items) = 0;
};
```

2.2.2. 定义类 UID

一个组件类如果要让外界模块能创建对象实例，需要指定组件类的唯一标识信息，一般采用 GUID 串来标识组件类。由于普通字符串与接口指针都是指针，为了用强类型作区分，

采用 XCLSID 辅助类来封装类 UID，定义类 UID 常量，供创建对象时之间使用该常量。

既可以在单独的 H 文件中定义类 UID 常量，也可以和接口定义同在一个 H 文件中。

下面的例子定义了一个类 UID 常量 CLSID_Example：

```
const XCLSID CLSID_Example("86347b32-64e4-490c-b273-ec7e010f244e");
```

注：XCLSID 辅助类是在 Ix_Object.h 中定义的，包含 XComPtr.h 后就可使用该类。

2.2.3. 实现接口

实现接口没有特殊之处，在插件内部按传统方式来实现一个或多个接口，具体就是定义一个类，从接口派生并实现其接口函数。

下面的例子是在 Cx_Example.h 中定义了 Cx_Example 类，实现了两个接口。

```
#pragma once

#include <Ix_Example.h>

class Cx_Example
    : public Ix_Example
    , public Ix_Example2
{
protected:
    Cx_Example();
    ~Cx_Example();

protected:
    // Ix_Example
    virtual void Foo();

    // Ix_Example2
    virtual void Foo2();
};
```

上面的例子中有两个地方值得留意（不是必须要这样，但推荐这样）：（1）构造函数和析构函数是保护类型，表示不允许直接实例化对象，也不允许直接删除销毁对象，通过智能指针类 Cx_Interface 或 Cx_Ptr 来自动实例化和销毁对象；（2）所实现的接口函数申明为保护类型，表示不允许直接调用该对象实例的函数，当然也可以申明为私有类型，表示不允许派生实现类调用其函数。

除了直接从接口继承的实现方式外（称之为“接口继承”），还可以从已有实现类继承，从而继承了对应的接口实现，这称之为“实现继承”，可以在其基础上实现更多接口或者重载部分函数。例如下面的 Cx_Example2 继承了 Cx_Example 的所有接口的实现内容：

（1）Cx_Example.h 文件：

```
#pragma once

#include <Ix_Example.h>

class Cx_Example : public Ix_Example
{
protected:
    Cx_Example();
    ~Cx_Example();

protected:
    virtual void Foo();
};
```

（2）Cx_Example2.h 文件：

```
#pragma once

#include "Cx_Example.h"

class Cx_Example2
    : public Cx_Example
    , public Ix_Example2
{
protected:
    Cx_Example2();
    ~Cx_Example2();

protected:
    virtual void Foo2();
};
```

在实现类的函数实现文件中没有特殊之处，不需要包含特殊的外部文件或使用任何宏。

2.2.4. 在插件中登记实现类

插件中的实现类一般不直接用于实例化对象，是通过智能指针类 Cx_Interface 或 Cx_Ptr 来实例化对象，需要在插件内登记该插件中有哪些可供实例化的类、实现类对应的类 UID、是否为单实例类。例如下面的 Module.cpp：

```
#include "stdafx.h"
#include <XModuleMacro.h>      // XDEFINE_CLASSMAP_ENTRY等宏
#include <XModuleImpl.h>       // 包含后自动实现插件内部机制
```

```
#include "Cx_Example.h"        // 包含实现类的定义
#include "Cx_ExampleTool.h"

// 登记有哪些可供实例化的类、实现类对应的类UID、是否为单实例类
XBEGIN_DEFINE_MODULE()
    XDEFINE_CLASSMAP_ENTRY(CLSID_Example, Cx_Example)
    XDEFINE_CLASSMAP_ENTRY_Singleton(CLSID_ExamTool, Cx_ExampleTool)
XEND_DEFINE_MODULE()
```

其中包含 XModuleImpl.h 用于自动实现插件内部机制，插件机制如何实现对插件开发者来说不用关心，宏 XDEFINE_CLASSMAP_ENTRY 用于登记普通类及其类 UID，可实例化出多个对象，XDEFINE_CLASSMAP_ENTRY_Singleton 用于登记单实例类及其类 UID，在一个进程中不论创建多少次都会得到同一个对象实例。

如果一个插件中没有实现任何接口，只需要使用接口，则去掉类登记语句行就行，例如：

```
#include "stdafx.h"
#include <XModuleMacro.h>
#include <XModuleImpl.h>

XBEGIN_DEFINE_MODULE()
XEND_DEFINE_MODULE()
```

对于以 MFC 扩展动态库或 Win32 动态库类型创建插件工程的情况，由于 DLL 入口函数基本上都相同，为了避免在多个插件中重复出现这样类似的 DLL 入口函数代码，可以将上面例子中的 XEND_DEFINE_MODULE() 换为下面两个宏之一：

```
XEND_DEFINE_MODULE_MFCEXTDLL()
XEND_DEFINE_MODULE_WIN32DLL()
```

2.2.5. 插件初始化和退出函数（可选）

如果希望在插件被加载后能进行一些额外操作，例如向管理器注册响应函数、观察者，可以在该插件中实现一个导出函数 InitializePlugin，该函数由插件管理器调用。插件管理器加载了一批插件后会再依次调用各个插件的 InitializePlugin 函数（如果实现了该函数的话）。InitializePlugin 函数返回 true 表示成功，返回 false 时插件管理器就会认为该插件应卸载掉。

```
extern "C" __declspec(dllexport) bool InitializePlugin()
{
    Cx_Interface<Ix_SomeManager> pIFManager(CLSID_SomeManager);
    if (pIFManager)
    {
        s_pObserver = new MyObserver();
        pIFManager->Register(s_pObserver);
    }
```

```
    return true;
}
```

如果希望在插件被卸载前能进行一些额外操作，例如向管理器注销响应函数、观察者，或者销毁某些资源，就可以在该插件中实现一个导出函数 UninitializePlugin，该函数由插件管理器调用。插件管理器先依次调用各个插件的 UninitializePlugin 函数（如果实现了该函数的话），然后再卸载这些插件。UninitializePlugin 函数无返回值，例如：

```
extern "C" __declspec(dllexport) void UninitializePlugin()
{
    if (s_pObserver)
    {
        Cx_Interface<Ix_SomeManager> pIFManager(CLSID_SomeManager);
        if (pIFManager)
        {
            pIFManager->Unregister(s_pObserver);
        }
        delete s_pObserver;
        s_pObserver = NULL;
    }
}
```

2.2.6. 插件工程的说明

对于 VC++，支持各种类型的工程，例如可在 MFC 应用程序、MFC 常规动态库、MFC 扩展动态库、Win32 动态库工程中实现插件功能，在 ActiveX 控件、ATL COM 控件、Win32 控制台程序、MFC 应用程序、各种动态库工程中使用插件接口。

要开发一个插件，通常可使用两种类型的工程：MFC 扩展动态库、Win32 动态库。采用 MFC 扩展动态库相对于 MFC 常规动态库的好处是不需要频繁的切换 MFC 模块状态。

不论采用哪种类型的工程来开发插件，除了设置必要的头文件包含路径外，所做的额外改动工作如下：

（1） 在 StdAfx.h 中包含 XComPtr.h 文件，以便使用智能指针类 Cx_Interface；

（2） 在一个 CPP 文件（例如 Module.cpp）中包含 XModuleMacro.h 和 XModuleImpl.h 文件，使用 XBEGIN_DEFINE_MODULE 等宏来登记可能的实现类；

为了快速创建新的插件工程，规范工程的各种目录等配置属性，推荐使用本模块提供的模板工程 MFCExtTempl 或 Win32DllTempl。以 MFCExtTempl 为例，将 MFCExtTempl 文件夹复制一份出来，修改文件夹和其中的文件名为你的工程名，并使用 UltraEdit 等文本编辑工具将该文件夹下的所有文件中的 MFCExtTempl 全部替换为你的工程名。

2.3. 使用方法--使用插件

2.3.1. 使用接口

使用智能指针类 Cx_Interface<接口名> 来使用接口，调用其接口函数；如果在传递对象时（例如定义函数参数时）不想包含特定的接口文件时，可以使用智能指针类 Cx_Ptr。通过智能指针类自动管理对象的引用计数和生命期。

下面举例说明接口调用和接口转换：

```
#include <XComPtr.h>    // Cx_Interface，一般是在StdAfx.h中包含
#include <Ix_Example.h>

// 返回值使用Cx_Ptr而不是Cx_Interface，可避免在函数定义中包含接口文件
Cx_Ptr MyFunc1()
{
    // 使用类ID和接口类型创建对象
    Cx_Interface<Ix_Example> pIFExample(CLSID_Example);
    if (pIFExample)
    {
        pIFExample->Foo();    // 调用接口函数
    }

    // 从一个接口转换为其他接口
    Cx_Interface<Ix_Example2> pIFExample2(pIFExample);
    if (pIFExample2.IsNotNull())
    {
        pIFExample2->Foo2();
    }

    MyFunc2(Cx_Ptr(pIFExample));
    return Cx_Ptr(pIFExample);
}

// 形参使用Cx_Ptr而不是Cx_Interface，可避免在函数定义中包含接口文件
void MyFunc2(const Cx_Ptr& obj)
{
    // Cx_Interface与Cx_Ptr互转，转为特定接口
    Cx_Interface<Ix_Example> pIFExample(obj);
    if (pIFExample)
    {
        pIFExample->Foo();
    }
}
```

2.3.2. 工程说明

对于 VC++，可在 MFC 应用程序、ActiveX 控件、ATL COM 控件、Win32 控制台程序、MFC 动态库、Win32 动态库等各种工程中使用插件接口。

在插件工程中可以直接使用插件接口，不论该接口是在哪个插件中实现的。

对于不需要实现插件接口而想使用接口的工程，例如 ActiveX 控件工程，可以在工程的一个 CPP 文件（通常是 StdAfx.cpp）中包含 **XComCreator.h** 文件，这样 Cx_Interface 就可以正常创建接口对象了；不需要使用 XBEGIN_DEFINE_MODULE 等宏和插件实现所需要的几个 H 文件，不需要上面介绍的 Module.cpp。具体例子见附件的 SimpleUse 和 SimpleApp。

2.4. 使用方法—加载插件

下面要描述的方法中涉及到 CPluginManager 和 Ix_PluginLoader，其函数说明在 H 文件中有详细的文档注释说明，也可以查看附件 Help.rar 中的帮助文档。

2.4.1. 在主程序中批量加载插件

这里描述的情况适用于 ActiveX 控件、ATL COM 控件要加载插件的情况。

具体例子可参考附件的 SimpleApp，在 StdAfx.cpp 中包含 XComCreator.h 文件，然后使用 PluginManager.h 加载多个插件。如果在主程序中实现了插件接口，包含了 XModuleImpl.h 文件，则不包含 XComCreator.h 文件。

加载插件的方法为先加载核心插件，然后加载某个目录下的所有插件，例如：

```
static CPluginManager s_loader;

BOOL MyApp::InitInstance()
{
    CWinApp::InitInstance();

    AfxOleInit();     // 初始化OLE和COM

    // 加载核心插件和插件管理器
    if (s_loader.LoadCorePlugins(L"Plugins"))
    {
        // 加载Plugins目录下的所有插件
        s_loader.GetPluginLoader()->LoadPlugins(L"Plugins");
        s_loader.GetPluginLoader()->InitializePlugins();
    }

    return TRUE;
}
```

```
int CWinApp::ExitInstance()
{
    s_loader.Unload();          // 卸载所有插件
    return CWinApp::ExitInstance();
}
```

2.4.2. 在单元测试中加载插件

具体例子可参考附件的 SimpleApp，在 StdAfx.cpp 中包含 XComCreator.h 文件，然后使用 PluginManager.h 加载多个插件。和主程序批量加载插件方法的不同点在于一般不批量加载所有插件，而是指定要加载哪几个具体的插件，例如：

```
int main()
{
    CPluginManager loader;

    if (loader.LoadPluginManager(L"../Plugins"))
    {
        loader.GetPluginLoader()->LoadPluginFiles(L"../Plugins",
            L" MyTest1.plugin, MyTest2.plugin");
        loader.GetPluginLoader()->InitializePlugins();

        Test();

        loader.Unload();
    }

    return 0;
}
```

2.4.3. 不使用插件管理器的简单加载方法

该方法不需要插件管理器，具有最小依赖性，但只能加载一个插件，具体例子见附件的 SimpleUse。具体是在 StdAfx.cpp 中先定义 USE_ONE_PLUGIN 然后包含 XComCreator.h 文件；直接加载插件 DLL 文件，并赋值给 g_hPluginDll 全局变量，用完再手动卸载插件。

3. 应用范例

在上面的使用方法描述中列举了比较详细的应用范例，更多范例见附件和开源网站。

PluginCore.rar 中包含本模块所提供的文件和几个简单示例工程，Help.rar 中为接口说明帮助文档。下面简单说明 PluginCore.rar 内文件的用途，这些文件中有详细文档说明。

pkg_Core\Interface	插件框架内核接口目录，所有工程都需要
Ix_Object.h	基本接口定义，所有工程都需要
XComPtr.h	智能指针类，所有工程都需要
Ix_ObjectFactory.h	供 XModuleImpl.h 或 XComCreator.h 使用
pkg_Core\Interface\Module	插件实现时所需要的文件
Ix_Module.h	获取自身插件的信息
XModuleMacro.h	插件中实现类登记所用的宏定义
XModuleImpl.h	插件机制实现文件
pkg_Core\Interface\PluginManager	供加载插件使用
XComCreator.h	供不实现插件而仅使用插件接口时包含
Ix_PluginLoader.h	插件加载和卸载
PluginManager.h	封装插件管理器接口、插件加载和卸载
pkg_Example	样例工程和模板工程

PluginCore.rar 还有下列不属于本模块、不在本文档中说明的常用接口文件：

pkg_Core\Interface\Log	插件日志输出接口、错误处理宏
pkg_Core\Interface\UtilFunc	常用的实用函数
pkg_Core\Interface\Xml	XML 读写封装接口
pkg_Core\Interface\ChangeObserver	改变通知 Observer 基本接口文件
pkg_Utility\Interface	常用的实用技术接口文件

4. 测试情况说明

本模块经过了较严格的单元测试（使用的是 CppUnit，有较多测试用例），测试都已通过（测试用例未作为本文档的附件）。从 2008 年 3 月到现在，历经三次较大的版本改进和完善，经过了十几个系统的实际成功验证。

下面是 X3 插件框架的发展和部分应用情况（本模块是 X3 插件框架的最底层独立模块）：

1) 2008.3，从 FantArt SDK 中提炼出来，应用于新产品 NewsLabel；
2) 2008.5，开发新系统 NewsCompose，完善了 1.0 插件机制；
3) 2008.9，开发新系统 ResGather，对插件机制进行了较大的改善和优化；
4) 2009.8，开发新系统 MagazineCompose，成功快速应用；
5) 2009.10，在已有系统 BumaLabel 5.2 中应用了插件框架；
6) 2009.11，插件框架第二版设计和测试，在部门内研讨；
7) 2010.2，开发新系统 TextbookCustomize，新框架 2.0 的成功快速应用；

8) 2010.4，开发新系统 NewsGather，扩充了 2.0 插件框架；

9) 2010.9，开发新系统 CebxExtractor；

10) 2010.9，XMLEditor 3.0 预研开发，框架改版为 3.0，进行了较多优化和较多测试；

11) 2010.11，重构 NewsCompose，应用 3.0 框架；

12) 2010.12，开发资源库后台服务配置工具，使用 2.0 插件框架。

注：目前的 3.0 版与 2.0 相比，插件开发者不受影响，改进完善了内部机制。