利用 C++ ATL 技术实现反射机制

金百东，李文举
(辽宁师范大学 计算机与信息技术学院，辽宁 大连 116081)

摘 要：编制灵活的应用程序框架系统，反射机制是重要的实现手段。但由于C++本身没有成熟的反射技术，对此进行了深入研究并提出一种实现方法。首先论述了反射机制的作用；然后描述了ATL动态链接库实现反射机制的基本原理，完善了ATL IDL文件接口标识符定义，利用前绑定或后绑定技术实现反射机制，比较了这两种方法的不同之处，着重强调了Idispatch接口在实现反射机制中的作用，最后给出需要继续研究的问题。由于采用了动态链接库技术，方便了构件的维护和复用，有助于团队开发和分布式开发。

关键词：反射机制；前绑定；后绑定；分布式开发
中图分类号：TP311 **文献标识码**：A **文章编号**：1674-7720(2013)02-0004-03

Implementation of reflection by C++ ATL technique

Jin Baidong, Li Wenju
(School of Computer and Information Technology, Liaoning Normal University, Dalian 116081, China)

Abstract: Reflection technique is important to implement the smart program frame system. C++ has not mature reflection technique, so the topic has a depth search and bring out a implemention method. Firstly, the topic demonstrates the function of reflection. Secondly, descripts the basis theory to implement reflection by ATL DLL, that means improving and perfecting the ATL IDL file, and implementing the reflection by binding early or binding late technology. Also the difference between the two methods is compared, and emphasize the importance of Idispatch interface is emphasized. At last, the topic brings out the functions which will be continue to research. Because of DLL technique, it is convenient to maintain and reused of component, and it is helpful to group development and distributed development.

Key words: reflection; binding early; binding late; distributed development

众所周知，Java语言有成熟的反射机制[1-2]，主要提供了以下功能：(1) 在运行时判断任意一个对象所属的类；(2)在运行时构造任意一个类的对象；(3)在运行时判断任意一个类所具有的成员变量和方法；(4)在运行时调用任意一个对象的方法；(5)生成动态代理。由于在struts、hibernate、spring框架中大量采用了反射机制，因此反射机制在编制框架类应用程序中大量运用，其作用是巨大的。但是C++本身并没有现成的反射机制，因此，这也成为了C++研究领域中的热门话题，其中绝大多数人研究了C++下统一类(即所有类定义及实现都在一个工程中)的反射机制问题，但目前流行的开发方式是“主程序+动态链接库方式”。本文旨在对该问题下的C++反射机制加以研究。

为了便于说明问题，特以图1为例。定义父类接口IFruit，定义多态函数Draw()，两个类Apple、Pear实现了该接口，并重写了多态函数。本文以该示例为基础对C++动态链接库反射机制加以论述来实现。

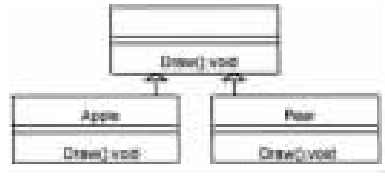

图1 示例基本类层次图

1 反射机制研究具体内容

由于反射机制主要应用于框架程序，而框架程序动态信息往往封装在配置文件中，配置文件中内容均是字符串，如封装了的类名信息等。因此，本文研究的主要内容是：(1)根据类名，动态产生该类的一个对象实例；(2)根据函数名及参数信息，动态执行该对象实例的相应函数[3-4]。

2 实现思想

2.1 动态链接库选择

本文选择的是ATL动态链接库。这是由于ATL本身提供了许多优秀的功能，例如：(1)AppWizard，它负责创建

初始的 ATL 工程；(2)Object Wizard(对象向导)，它为基本的 COM 组件创建代码；(3)对低级别的 COM 功能的内置式支持，如 IUnknown、类工厂和自注册功能；(4)支持 Microsoft 的接口定义语言，它提供了对自定义的虚函数表 Vtable 接口的调度支持，以及通过类型库进行自描述的功能；(5)支持 IDispatch 自动化接口及双向接口。

也就是说，运用 AppWizard 及 Object Wizard 产生的 ATL 动态链接库初始代码隐含了许多系统功能，可以直接或间接应用，而且代码质量都是专家级的，避免了人为的低层次的重复开发。

根据图 1 功能，需要分别产生两个 ATL 动态链接库工程，假设名称为 MyApple 及 MyPear，再分别插入 IFruit 接口，声明 Draw 函数；完善对应的实现类 Apple 及 Pear，实现具体的 Draw 函数功能。

2.2 完善 ATL IDL 接口标识副符定义

由于 ATL 一个功能类中可能实现多个接口，ATL 使用全局特有标识符 uuid 来标识类及接口。uuid 是一个独有的、128 bit、并且具有非常高可靠率的数值，它把一个独有的网络地址和一个非常精细的时间印签(100 ns)结合在一起。因此，一个类对应一个 uuid，所实现的各个接口对应各自的 uuid。

根据图 1 产生的工程 MyApple、MyPear，它们产生的类标识符 uuid 一定是不同的。由于它们都实现了 IFruit 接口，希望 IFruit 接口对应的 uuid 相同，但默认情况下它们是不同的，因此，最好把它们改成相同的，只须修改相应的接口定义 IDL 文件即可。下面截取了 MyPear.idl 关键部分代码，如果以 MyApple.idl 为基准，则只须把接口标识符对应内容修改为 MyApple.idl 文件中的内容即可。这样，就形成了完善的 ATL 多态组件。

MyPear.idl 关键部分代码如下：

接口标识符：

```
[
    object,
    uuid(908F5798-4D39-4389-A426-43A23F3F5772),
    dual,
    helpstring("IFruit Interface"),
    pointer_default(unique)
]
interface IFruit : IDispatch
{
    [id(1), helpstring("method Draw")] HRESULT Draw();
};
```

类标识符：

```
[
    uuid(82B621A0-6224-45D9-9ECA-B83B835B63A6),
    helpstring("Pear Class")
]
coclass Pear
{
    [default] interface IFruit;
};
```

2.3 动态加载组件

客户端可通过前绑定或后绑定技术加载并执行 ATL 动态链接库组件。下面以图 1 组件为例加以说明。

(1)前绑定加载并执行组件。主要代码如下所示：

```
IID IID_IFruit={0x908F5798,0x4D39,0x4389,
    {0xA4,0x26,0x43,0xA2,0x3F,0x3F,0x57,0x72}};
                                        //接口 uuid
CLSID CLSID_Apple={0x59CA74CB,0x7AC2,0x4F88,{0x92,0xFF,
0x5B,0x91,0xA6,0xC0,0xC2,0x88}};        //Apple uuid
CLSID CLSID_Pear= {0x82B621A0,0x6224,0x45D9,{0x9E,0xCA,
0xB8,0x3B,0x83,0x5B,0x63,0xA6}};        //Pear uuid
interface IFruit : public IDispatch     //定义多态接口 IFruit
{
public:
    virtual HRESULT STDMETHODCALLTYPE Draw()=0;
};
void main( )
{
    CoInitialize(NULL);                 //初始化 COM 库
    IFruit *p = NULL;
    CoCreateInstance(CLSID_Apple,NULL,
        CLSCTX_INPROC_SERVER,
        IID_IFruit, (LPVOID *)&p);
    p->Draw();
    p->Release();
    CoUninitialize();                   //卸载 COM 库
}
```

可知若实现前绑定技术，客户端需完成以下工作：①从组件 IDL 文件中提取类标识符uuid 及接口标识符 uuid 放在客户端。对图 1 组件来说，由于在 2.2 节中已经使各个 IFruit 接口的 uuid 值相同，因此提取出了两个类标识符 CLSID_Apple、CLSID_Pear，以及一个接口标识符 IID_IFruit；②客户端定义多态接口 Ifruit；③通过 CoCreateInstance 加载所需组件并执行相应函数。该函数的第 1 个参数是类标识符 uuid，第 4 个参数是接口标识符 uuid。其基本功能是产生相应组件的一个对象实例，这其实是实现 C++反射机制的核心所在。若没有该函数，则一定要编制一个组件工厂类，在众多组件中根据条件选择产生相应的组件对象。由于编制的工厂类不是系统内嵌部分，稳定性、完善性都存在许多未知的因素。而现在这一切都被 CoCreateInstance 函数隐藏了，它属于系统内嵌的部分，编程者无须考虑它的完备性，只须调用系统提供的接口即可。

CoCreateInstance 是通过类标识符 CLSID_xxx 来创建组建对象的，该标识符不好记忆，但在该组件向注册表注册时，同时注册了一个 ProgID 字符串，它是对 CLSID_xxx 的文字说明，是一一映射关系。因此，在产生 ATL 组件时添好 ProgID 字符串，之后，在程序中通过 ProgID 字符串即

可获得 CLSID_xxx，如下述代码所示，运用 CLSIDFromProgID 函数可完成所需信息的转换。也就是说，可通过反射机制所要求的“类名”来加载相应组件。

```
CLSID clsid;
CLSIDFromProgID(L"Apple",&clsid);
```

(2) 后绑定加载并执行组件。主要代码如下：

```
IID IID_IFruit={0x908F5798,0x4D39,0x4389,{0xA4,0x26,
0x43,0xA2,0x3F,0x3F,0x57,0x72}};          //接口 uuid
CLSID CLSID_Apple={0x59CA74CB,0x7AC2,0x4F88,{0x92,
0xFF,0x5B,0x91,0xA6,0xC0,0xC2,0x88}};     //Apple uuid
CLSID CLSID_Pear={0x82B621A0,0x6224,0x45D9, {0x9E,
0xCA,0xB8,0x3B,0x83,0x5B,0x63,0xA6}};     //Pear uuid
void main( )
{
    CoInitialize(NULL);                   //初始化 COM 库
IDispatch *p = NULL;
CoCreateInstance(CLSID_Apple, NULL,
    CLSCTX_INPROC_SERVER,
    IID_IDispatch, (LPVOID *)&p);
LPOLESTR lpOleStr = L"Draw";
DISPID dispid;
DISPPARAMS disp = {NULL, NULL, 0, 0};
VARIANTARG vaResult;
VariantInit(&vaResult);
p->GetIDsOfNames(IID_NULL, &lpOleStr, 1,
   LOCALE_USER_DEFAULT, &dispid);
p->Invoke(dispid,IID_NULL, LOCALE_USER_DEFAULT,
    DISPATCH_METHOD, &disp, NULL, 0, NULL);
p->Release();
CoUninitialize();                         //卸载 COM 库
}
```

可知若实现后绑定技术，客户端需完成以下工作：①从组件 IDL 文件中提取类标识符uuid 及接口标识符 uuid 放在客户端，同前绑定；②无需定义 IFruit 多态接口，只需通过 CoCreateInstance 获得系统固有的 IDispatch 接口即可，该函数的第 1 个参数仍然是类标识符对应的uuid，第 4 个参数是 ATL 固有的 IDispatch 接口标识符 IID_IDispatch，而不是 IID_IFruit；③创建所调用函数需要的函数参数结构体 DISPPARAMS，示例中 Draw 函数是无参的，所以 DISPPARAM 结构体值非常简单；④定义返回值参数，如示例中的 VARIANTARG vaResult；⑤通过系统函数 GetIDsOfNames 获得 Draw 函数的分配号；⑥通过系统函数 Invoke 隐式地启动 Draw 函数。

同前绑定相比，由于不需要在客户端定义多态接口，所以称之为后绑定。后绑定技术主要应用了 IDispatch 自动化技术，其原理如图 2 所示。

当客户端获得一个有效 IDispatch 指针后，就可以操作 IDispatch 虚函数表中的 7 个函数。如可通过 Invoke 函数，同时根据组件功能函数在派发表中的函数号(函数号可在 IDL 文件中看出)，即可间接调用组件中相应函数。因此如果一个组件中有多个函数，可以用相同的 Invoke 方法来调用。与 Java 反射 Method 类中的 invoke 方法非常相似。

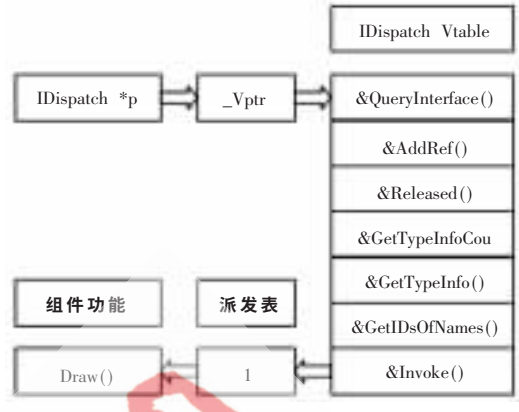

图 2 IDispatch Vtable 以及派发表

同前绑定相比，后绑定技术需要在运行时做大量的工作，所以其运行速度稍慢，但它的灵活性也是最高的。如果服务器接口发生变化，则客户机程序不用重新编译就可以执行服务器新增的功能[5]。

本文论述了利用 ATL 动态链接库实现反射机制的基本原理，主要优点有以下两方面内容：

(1)利用类、接口标识符及 CoCreateInstance 函数，可方便产生组件的对象实例，省去了人为工厂类的编制。

(2)利用 IDispatch 自动化接口，使用统一方法 Invoke 屏蔽了组件接口函数的差异，可编制更加灵活的基于反射技术的框架程序。

当然，利用 IDispatch 接口的 7 个系统函数能否开发出类似 Java 的 Class、Method、Field、Constructor 等反射类，这些都是值得继续探讨的问题。

参考文献

[1] 祝连鹏,王虎,赵学臣.基于 SOA 反射工厂的软件体系架构[J].计算机技术与发展,2011,21(7):125-128.

[2] 谷玉奎,曹宝香,袁玉珠.基于 SOA 的分布式构件库管理模型[J]. 计算机技术与发展,2008,18(04):101-103.

[3] 蒲海红,侯秀平,赵云峰.构件集成算法的研究[J].计算机技术与发展,2009,19(5):75-78.

[4] 雷宁宁,薛锦云,刘超. 基于构件开发与传统面向对象开发之比较[J].计算机技术与发展,2007,17(8):88-91.

[5] 罗巨波,吴可嘉,叶鹏,等. 基于反射机制的软件体系结构重用方法及工具[J]. 计算机工程,2009,35(14):90-92.

(收稿日期：2012-10-18)

作者简介：

金百东，男，1969 年生，硕士，讲师，主要研究方向：系统分析与应用。

李文举，男，1964 年生，博士，教授，主要研究方向：系统分析与应用。