活动模板库介绍

在这一章中，我们将学习活动模板库(ATL，Active Template Library)，这是 Microsoft Visual C++ .NET 中包含的第二个应用程序框架(MFC 是第一个)。首先我们将快速回顾一下组件对象模型(COM)，并看一看用另一种方法来编写第 22 章中的 CSpaceship 对象，这也正好说明了可以有多种方法来编写一个 COM 类。(当您查看 ATL 的类的组成方法时，这将变得很重要。)接下去，我们将研究活动模板库，首先将重点放在 C++模板和纯粹的 C++ Smart 指针上，以及它们如何被用于 COM 开发之中。我们将讨论 ATL 的客户端程序设计，并且看一看 ATL 中的一些 Smart 指针。最后，我们将讨论 ATL 的服务器端程序设计，分别用经典的 ATL 和属性化的 ATL 来重新实现第 22 章的飞船示例对象，以便使您对 ATL 的结构有一个初步的认识。

25.1 再谈 COM

要理解 COM 程序设计，最重要的概念是，它是基于接口的。正如您在第 22 章中所看到的，您在使用基于接口的程序设计方法时并不需要真正的 COM，甚至不需要 Micro-soft 运行时库的支持。您需要的仅仅是一些基本训练。

回想一下第 22 章中的飞船示例程序。我们从一个名为 CSpaceship 的类开始，它实现了几个函数。经验丰富的 C++开发人员通常一坐在计算机前，就可以编写出像下面这样的一个类：

```
class CSpaceship {
    void Fly();
    int& GetPosition();
};
```

然而，当使用基于接口的开发技术时，这个过程有所不同。您现在不再直接编写出一个类，而是先定义一个接口，然后再实现这个接口。在第 22 章中，Fly 和 GetPosition 函数被移到一个名为 IMotion 的抽象基类中。

```
struct IMotion {
```

```
    virtual void Fly() = 0;
    virtual int& GetPosition() = 0;
};
```

然后，我们使 CSpaceship 类继承自 IMotion 接口，如下所示：

```
class CSpaceship : IMotion {
    void Fly();
    int& GetPosition();
};
```

请注意，这时 IMotion 接口与它的实现被分离开了。当您进行接口开发时，接口总是先出现。然后您就可以在接口之上进行工作，并确保它是完整的，但又不会过度膨胀。但是，一旦接口已经被发布(也就是说，一旦其他开发人员开始对它进行编码时)，接口就被冻结了，并且不能再改变了。

基于类的程序设计和基于接口的程序设计之间存在着微妙的差别，这种差别带来了一些额外的编程负担。但是，这正是理解 COM 的关键点之一。通过将 Fly 和 GetPosition 函数集中到一个接口中，您实际上制定了一个二进制原型特征。也就是说，通过预先定义接口，并利用接口与一个类进行对话，您就可以做到，让客户代码在与一个类对话时按照一种与语言无关的方式来进行。

将函数集中起来放到接口中，这本身也是很有用的。譬如说，您希望描述飞船以外的事物，例如飞机。当然，您一定也想像得到，飞机也具有 Fly 和 GetPosition 函数。接口程序设计方法提供了一种更为高级的多态性——在接口层次上的多态性，而不仅仅在单个函数的层次上。

接口与实现的分离是基于接口的程序设计方法的基础。而 COM 是以接口程序设计方法为中心的，它加强了接口与实现之间的区别。在 COM 中，客户代码与一个对象进行对话的唯一途径是通过接口。但仅仅将函数集中到接口中还不够。还需要一个要素——一种在运行时可以发现功能的机制。

25.1.1　核心接口：IUnknown

使 COM 与普通的接口程序设计方法区别开来的关键要素是下面这条规则：每个 COM 接口的头 3 个函数都是相同的。COM 中的核心接口 IUnknown 如下所示：

```
struct IUnknown {
    virtual HRESULT QueryInterface(REFIID riid, void** ppv) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

每一个 COM 接口都从这个接口派生(这就意味着，您所看到的每一个 COM 接口的前三个函数都是 QueryInerface、AddRef 和 Release)。为了将 IMotion 转变为一个 COM 接口，您可以将它从 IUnknown 派生：

```
struct IMotion : IUnknown {
    void Fly();
    int& GetPosition();
};
```

说明： 如果您希望这些接口在进程外也能工作，那么您必须使每一个函数都返回一个 HRESULT。在本章后面讨论到属性化 ATL 时您将会看到这一点。

AddRef 和 Release 值得再次提及，因为它们是 IUnknown 的一部分。AddRef 和 Release 允许一个对象控制它自己的生存期(如果它选择要这样做的话)。通常，客户应该像对待资源一样对待接口指针：客户获得接口，使用接口，当用完接口之后再将它们释放。通过 AddRef，对象就会知道有了新的引用；通过 Release 函数，对象知道它们已不再被引用。对象通常使用该信息来控制它们的生存期。例如，许多对象在它们的引用计数为零时将自己删除。

下面给出了一些使用该飞船的可能的客户代码：

```
void UseSpaceship() {
    IMotion* pMotion = NULL;

    pMotion = GetASpaceship(); // This is a member of the
                               //  hypothetical Spaceship
                               //  API. It's presumably an
                               //  entry point into some DLL.
                               //  Returns an IMotion* and
                               //  causes an implicit AddRef.
    If(pMotion) {
        pMotion->Fly();
        int i = pMotion->GetPosition();
        pMotion->Release(); // done with this instance of CSpaceship
    }
}
```

IUnknown 内部另一个(也是更重要的一个)函数是第一个成员函数：QueryInterface。QueryInterface 是 COM 在运行时发现功能的一种机制。如果有人给了您一个 COM 接口指针，它指向一个对象，但您不想使用该指针，那么您可以用这个指针向对象请求它的其他不同的接口。这种机制，以及接口一旦发布之后将保持不变的事实，结合起来使得基于 COM 的软件可以随着时间的发展而安全地演变。结果是，您可以在 COM 软件中加入功能，而不会打破那些依赖于该 COM 软件的老版本客户。而且，客户一旦知道了新的功能，它就可以有更多的获得新功能的识别办法。

例如，您只需加入一个名为 IVisual 的新接口，就可以在 CSpaceship 的实现中加入新的功能。加入这个接口是很有意义的，因为您可以让三维空间中的对象从视图中移进移出。您也可以让有些对象在三维空间中是不可见的(例如一个黑洞)。下面是 IVisual 接口：

```
struct IVisual : IUnknown {
```

```
    virtual void Display() = 0;
};
```

客户可以像这样使用 IVisual 接口：

```
void UseSpaceship() {
    IMotion* pMotion = NULL;

    pMotion = GetASpaceship(); // Implicit AddRef
    if(pMotion) {
        pMotion->Fly();
        int i = pMotion->GetPosition();

        IVisual* pVisual = NULL;
        PMotion->QueryInterface(IID_IVisual, (void**) &pVisual);
        // Implicit AddRef within QueryInterface

        if(pVisible) {
            pVisual->Display(); // uncloaking now
            pVisual->Release(); // done with this interface
        }
    }
    pMotion->Release(); // done with this instance of IMotion
}
```

注意，上面的代码非常小心地使用了接口指针：只有当正确地获得了接口的时候，它才使用这些接口；然后，当用完了这些接口之后，就将它们释放。这是在最低层次上对 COM 进行编程的原始方法——即首先获得一个接口指针，然后使用接口指针，用完之后再将它释放。

25.2　编写 COM 代码

正如您所看到的，编写 COM 客户代码并不是完全不同于编写常规的 C++代码。然而，客户与之打交道的 C++类是抽象基类。您不再像往常那样调用 new 操作符，而是显式地调用某些 API 函数来创建 COM 对象，并获得 COM 接口。并且，您不再是直接删除对象，而是按照 COM 接口的规则，通过调用 Release，以便与 AddRef 调用保持平衡。

那么，您该如何实现 COM 类并使它运行起来呢？在第 22 章中，您已经看到了如何用 MFC 来做到这一点。下面是另一个实现了 CSpaceship 的 COM 类的例子。该例子使用多继承方法来编写 COM 类。也就是说，该 C++类同时继承了几个接口，然后实现了所有函数(当然也包括 IUnknown 的函数)的并集。

```
struct CSpaceship : IMotion, IDisplay {
    ULONG m_cRef;
```

```
    int m_nPosition;

    CSpaceship() : m_cRef(0),
                   m_nPosition(0) {
    }

    HRESULT QueryInterface(REFIID riid,
                           void** ppv);
    ULONG AddRef() {
        return InterlockedIncrement(&m_cRef);
    }
    ULONG Release() {
        ULONG cRef = InterlockedIncrement(&m_cRef);
        if(cRef == 0){
            delete this;
            return 0;
        } else
            return m_cRef;
    }
// IMotion functions:
    void Fly() {
        // Do whatever it takes to fly here
    }
    int GetPosition() {
        return m_nPosition;
    }

    // IVisual functions:
    void Display() {
        // Uncloak
    }
};
```

25.2.1 使用多继承实现 COM 类

如果您习惯于阅读简单的 C++代码，那么前面的代码可能看起来会比较陌生。这是多继承机制的一种比较少见的形式，称为接口继承(interface inheritance)。大多数 C++开发人员对于实现继承(implementation inheritance)都比较熟悉，在实现继承中，派生类会继承基类的一切，包括具体的实现。所谓接口继承，是指派生类继承了基类的接口。前面的代码实际上为 CSpaceship 类加入了 2 个数据成员，每一个隐含的 vtable 都有一个 vptr。

当您使用多继承方法来实现接口的时候，每一个接口都共享了 CSpaceship 的 IUnk-nown 实现。这个共享也说明了 COM 中一个既深奥又很重要的概念，即 COM 身份(COM identity)。COM 身份的

基本思想是：IUnknown 是 COM 中的 void*。IUnknown 是一个保证不会与任何一个对象脱离的接口，您总是可以到达对象的 IUnknown 接口。COM 身份也表明了(在前面的示例中)，客户可以通过 CSpaceship 的 IMotion 接口来调用 QueryIn-terface，以获得 IVisual 接口。相反，客户也可以通过 CSpaceship 的 IVisual 接口来调用 QueryInterface，以获得 IMotion 接口。最后，客户可以通过 IUnknown 来调用 QueryInterface，以获得 IMotion 或 IVisual 接口，而且，客户也可以通过 IMotion 或者 IVisual 来调用 Query-Interface，以获得指向 IUnknown 的指针。要想进一步了解关于 COM 身份的知识，请参阅 Don Box 所著的 *Essential COM*(Addison-Wesley，1997)或者 Dale Rogerson 所著的 *Inside COM*(Microsoft Press，1997)。

您常常会看到一种用“棒棒糖”(lollipop)形式的图案来表示的 COM 类，它描述了一个 COM 类所实现的接口。在 22.2.3 小节中，您可以看到这样的棒棒糖图案的示例。

用多继承方法来实现 CSpaceship 可以自动满足 COM 身份规则。注意，所有对 Query-Interface、AddRef 和 Release 的调用都会到达 C++类中的相同位置上，而不论它们是通过哪个接口被调用的。

这或多或少是 COM 的本质。作为 COM 开发人员，您的任务是创建有用的服务，并通过 COM 接口将这些服务暴露出来。在最基本的层次上，这意味着，要将一些函数表连接起来，以便满足 COM 的身份规则。您已经看到了有两种办法可以做到这一点。(第 22 章演示了如何用嵌套类和 MFC 来做到这一点，而本章仅仅说明了如何在 C++中用多继承机制来编写 COM 类。)然而，除了接口编程，以及编写一些类来实现接口之外，COM 还有其他一些令人迷惑的方面。

25.3　COM 基础设施

当您明白了基于接口的程序设计方法的概念后，您要想让您的类与系统的其余部分协同工作，必定还要实现相当多的细节。这些细节往往掩盖了 COM 最基本的优美之处。

首先，COM 类需要有生存的空间，所以，您必须将它们包装在一个 EXE 或者 DLL 文件中。另外，您所编写的每一个 COM 类都需要一个伴随的类对象(经常被称为一个类厂)。一个 COM 服务器的类对象被暴露出来的方式因情况而异，具体取决于您如何包装 COM 类(在 DLL 或者 EXE 中)。您还必须考虑服务器的生存期。只有当确实有必要的时候，服务器才应该驻留在内存中；当不再需要时，它必须从内存中离开。为了做到这一点，服务器维护了全局的锁计数，它表明有多少个对象具有外部接口指针。最后，行为良好的服务器应该在注册表中插入必要的值，从而客户软件可以很容易地将它们激活。

在本书中，我们已经花了很多时间来讨论 MFC。正如您在第 22 章中看到的那样，MFC 负责处理大量与 COM 有关的细节问题。例如，CCmdTarget 有 IUnknown 的实现。MFC 甚至还创建了一些 C++类和宏来实现类对象(例如 COleObjectFactory、COleTemplate-Server、DECLARE_OLE_CREATE 和

IMPLEMENT_OLE_CREATE)，它们将大多数正确的注册表项放到了注册表中。MFC 有一个最容易实现的、也是最灵活的 IDispatch 版本——您所需要的是一个 CCmdTarget 对象和 Visual Studio .NET 环境(特别是 Add Property Wizard 和 Add Method Wizard)。而对 OLE 拖放这种您自己的事情，MFC 也为拖放协议提供了一个标准的实现。最后，MFC 也提供了一种很容易获取的方法来编写快速的、功能强大的 ActiveX 控件。(您也可以在 Microsoft Visual Basic 中编写 ActiveX 控件，但是，您不会有在 MFC 中那样的灵活性。)这些都是非常好的优点。然而，使用 MFC 也有缺点。

为了获得这些特性，您必须引入整个 MFC。这不见得是一个不好的做法，但是，当您决定使用 MFC 的时候，您必须要知道引入 MFC 所付出的代价。MFC 很大，它不得不这样，因为它是一个带有许多功能的 C++框架。

从前几章介绍的示例程序中您可以看出，实现 COM 类并使它们可以被客户使用，将需要编写大量的代码，而且从一个类实现到另一个类实现，这些代码都是相同的。对于您所碰到的每一个 COM 类，IUnknown 实现基本都是相同的——这些类之间的主要区别在于，每一个类所暴露的接口并不相同。

现在让我们来快速了解一下 COM 和 ATL 分别处于什么样的地位。

25.3.1 ActiveX、OLE 和 COM

COM 只是一系列高层应用程序集成技术的综合，其中包括 ActiveX 控件和 OLE 拖放等等。这些技术定义了基于 COM 接口进行通信的协议。您可能会选择自己来实现某些高层次的特性，例如拖放和控件。然而，让某一个应用程序框架来完成这些琐碎的工作可能会更有意义。当然，那正是 MFC 存在的原因。


说明： 要想了解如何用纯粹的 C++来实现高层次的特性，请参阅 Kraig Brockschmidt 所著的 *Inside OLE, 2nd Edition*(Microsoft Press，1995)。

25.3.2 ActiveX、MFC 和 COM

虽然单纯的 COM 本身非常有趣(只要看一看 COM 远程通信的原理，您就会感到惊讶)，但是，高层次的特性才是应用程序被市场接受的原因。MFC 是一个巨大的框架，适合于创建完整的基于 Windows 的应用程序。在 MFC 内部，您可以找到大量的实用工具类和一套数据管理/表现机制(文档-视图结构)，以及对拖放、自动化和 ActiveX 控件的支持。您可能不会想到要从头开发一个支持 OLE 拖放的应用程序，使用 MFC 就会方便很多。然而，如果您需要创建一个小型或者中等规模的 COM 服务，那么您也许应该远离 MFC，这样，您就可以不用包含 MFC 为高层次特性而维护的所有代码。

您可以用纯粹的 C++来创建 COM 组件，但是这样做的话，您将会花费大部分时间来编写样板代码(例如 IUnknown 和类对象)。用 MFC 来编写 COM 应用程序是在应用程序中加入高级特性比较容易的办法，但是，要用 MFC 来编写轻量级的 COM 类是很困难的。ATL 位于纯粹的 C++和 MFC 之间，它既不要求您输入样板代码，也不用引入 MFC 的整个结构，就可以实现基于 COM 的软件。ATL 主要是一组 C++模板，和其他一些为编写 COM 类而提供的支持。

25.4　ATL 简介

如果您看一看 ATL 的源代码，就会发现，ATL 是由一组头文件和 C++源代码文件组成的。其中的绝大多数都驻留在 ATLMFC\Include 目录中(该目录位于 Microsoft Visual Studio .NET 的安装目录中)。下面列出了一些 ATL 文件及其中的内容。

25.4.1　AtlBase.h

该文件包含：

- ATL 的函数类型定义(typedef)
- 结构和宏定义
- 管理 COM 接口指针的 Smart 指针
- 线程同步支持类
- CComBSTR、CComVariant 的定义，以及对线程和套间支持的定义

25.4.2　AtlCom.h

该文件包含：

- 支持类对象/类厂的模板类
- IUnknown 实现
- 对于 tear-off 接口的支持
- 类型信息管理和支持
- ATL 的 IDispatch 实现
- COM 枚举器模板
- 连接点支持

25.4.3 AtlConv.cpp 和 AtlConv.h

这两个源代码文件包含了对 Unicode 转换的支持。

25.4.4 AtlCtl.cpp 和 AtlCtl.h

这两个文件包含：

- 用于 ATL 的 IDispatch 客户支持和事件激发支持的源代码
- CComControlBase
- 对控件的 OLE 嵌入协议的支持
- 属性页支持

25.4.5 AtlIFace.idl 和 AtlIFace.h

AtlIFace.idl(它会产生 AtlIFace.h)包含了一个名为 IRegistrar 的专门用于 ATL 的接口。

25.4.6 AtlImpl.cpp

AtlImpl.cpp 实现了一些类，如 CComBSTR，这些类在 AtlBase.h 中声明。

25.4.7 AtlWin.cpp 和 AtlWin.h

这两个文件提供了关于窗口和用户界面的支持，包括：

- 一个消息映射机制
- 一个窗口类
- 对话框支持

25.4.8 StatReg.cpp 和 StatReg.h

ATL 提供了一个名为 Registrar 的 COM 组件，它可用来在注册表中加入适当的表项。实现这个组件的代码在 StatReg.h 和 StatReg.cpp 中。

现在让我们从 ATL 对客户端 COM 开发的支持开始来学习 ATL。

25.5　客户端 ATL 程序设计

ATL 主要有两个方面——客户端的支持和对象端的支持。到目前为止，大部分的支持都在对象端，因为 ATL 的主要需求是实现 ActiveX 控件。不过，ATL 提供的客户端支持也是很有用的，并且也很有趣。下面我们将看一看 ATL 客户端的支持。由于 C++模板是 ATL 的基础，所以我们将首先看一看 C++模板。

25.5.1　C++模板

理解 ATL 的关键在于理解 C++模板。虽然模板语法很让人恐惧，但模板的概念是非常简单的。C++模板有时被称为编译器特许的宏(complier-approved macro)，这是非常恰当的描述。考虑一下宏所完成的工作：当预处理器遇到一个宏时，它就查看这个宏，并将其展开为常规的 C++代码。但是，宏的问题在于，它们有时很容易出错，而且它们永远不是类型安全的。如果您在使用一个宏的时候传递了一个不正确的参数，那么编译器不会有任何提示，而您的程序却可能会崩溃。然而，模板就好像是类型安全的宏。当编译器遇到一个模板时，它会将模板展开，就像对宏所做的那样。但是，因为模板是类型安全的，所以编译器会在用户之前首先捕获任何与类型有关的错误。

利用模板来重用代码，这与您所习惯的传统 C++开发方式有所不同。利用模板编写出来的组件通过模板替换来重用代码，而不是从基类中继承功能来重用代码。所有来自模板的样板代码都被原封不动地粘贴到项目中。

一个使用模板的典型例子是动态数组。想像您需要一个存储整数的数组。您希望这个数组可以根据需要而增长，所以不再将它声明为固定大小的数组。因此，您将这个数组设计为一个 C++类。当您的同事得知您要开发这样的新类，她说她正需要这样的功能。但她要在数组中使用浮点数。在这样的情况下，您可以使用一个 C++模板，而不必重复完全相同的代码(除了使用不同的数据类型以外)。

这里有一个例子，说明了您可以如何用模板来解决上述问题。下面是一个用模板来实现的动态数组：

```
template <class T> class DynArray {
public:
    DynArray();
    ~DynArray(); // clean up and do memory management
    int Add(T Element); // adds an element and does
                        //  memory management
    void Remove(int nIndex) // remove element and
```

```
                        //  do memory management
    T GetAt(nIndex) const;
    int GetSize();
private:
    T* TArray;
    int m_nArraysize;
};

void UseDynArray() {
    DynArray<int> intArray;
    DynArray<float> floatArray;

    intArray.Add(4);
    floatArray.Add(5.0);

    intArray.Remove(0);
    floatArray.Remove(0);

    int x = intArray.GetAt(0);
    float f = floatArray.GetAt(0);
}
```

您可以想像得到，创建一些模板对于实现 COM 样板代码是非常有用的，而且模板是 ATL 用来提供 COM 支持的机制。前面的例子仅仅是模板的许多用法中的一种。模板不仅可用于将类型信息应用到特定种类的数据结构上，而且它们也可以被用来封装算法。当您进一步了解 ATL 的时候，就可以看到模板的这些用法。

25.5.2 Smart 指针

在模板的大多数通用用法中，其中之一便是 Smart 指针。传统的 C++书籍将 C++的内置指针称为“哑”(dumb)指针。这不是一个很好听的名字，但是普通的 C++指针除了指向功能外，并没有做其他的事情。通常是由客户来执行细节操作，例如指针初始化。

作为一个例子，让我们对两种使用 C++类的开发人员进行建模。首先，我们创建两个类 CVBDeveloper 和 CCPPDeveloper。

```
class CVBDeveloper {
public:
    CVBDeveloper() {
    }
    ~CVBDeveloper() {
        AfxMessageBox
            ("I used Visual Basic .NET, so I got home early.");
```

```
    }
    virtual void DoTheWork() {
        AfxMessageBox("Write them forms");
    }
};

class CCPPDeveloper {
public:
    CCPPDeveloper() {
    }
    ~CCPPDeveloper() {
        AfxMessageBox("Stay at work and fix those pointer problems");
    }
    virtual void DoTheWork() {
        AfxMessageBox("Hacking C++ code");
    }
};
```

Visual Basic 开发人员和 C++开发人员都具有进行性能优化的函数。现在，想像某个客户编写出这样的代码：

```
//UseDevelopers.cpp

void UseDevelopers() {
    CVBDeveloper* pVBDeveloper;

    // The VBDeveloper pointer needs
    //  to be initialized
    //  sometime. But what if
    //  you forget to initialize and later
    //  on do something like this:
    if(pVBDeveloper) {
        // Get ready for fireworks
        //  because pVBDeveloper is
        //  NOT NULL, it points
        //  to some random data.
        c->DoTheWork();
    }
}
```

这里，客户代码忘记将 pVBDeveloper 指针初始化为 NULL。(当然，在现实中这是不会发生的！)因为 pVBDeveloper 包含一个非 NULL 的值(该值实际上是当时栈上的随机值)，所以，针对指针的有效性测试成功了，但是实际上您期望它失败。客户将继续进行，它相信一切都正常。当然客户最终会崩溃，因为客户正在“调用到黑暗中”(谁知道 pVB-Developer 会指向哪里呢——就算是 Visual Basic 开发人员对此可能也一无所知)。实质上，我们需要某种机制来确保指针被正确地初始化。这正是 Smart

指针的方便之处。

现在，请想像第二种情形。您可能会希望在“开发人员类型”的类中加入一点额外的代码，它们可以执行某一种对所有开发人员来说都是共同的操作。例如，您可能希望所有的开发人员在开始编码之前，都先进行一些设计工作。以前面的 Visual Basic 开发人员和 C++开发人员为例，当客户调用 DoTheWork 时，开发人员将不得不在没有正确的设计的情况下进入编码阶段，当客户陷入困境时他可能会离开。您所期望做到的是在开发人员类中加入一个一般化的钩子，这样它们可以确保在开始编码之前设计工作已经完成。

C++中解决这些问题的方案是 Smart 指针。

25.5.3 让 C++指针具有智能

记住，Smart 指针是一个用于包装指针的 C++类。您可以把一个指针包装到一个类中(特别是包装到一个模板中)，这样可以确保某些特定的操作将会自动进行，而不用让这些模式化的普通操作等到客户来调用。此类操作的一个很好的例子是，确保指针被正确地初始化，从而避免发生由于随机赋值的指针而导致崩溃的情况。另一个很好的例子是，您可以确保在通过一个指针进行函数调用之前，样板代码首先被执行。

让我们为前面描述的开发人员模型创建一个 Smart 指针。考虑一个名为 SmartDeveloper 的模板类：

```
template<class T>
class SmartDeveloper {
    T* m_pDeveloper;

public:
    SmartDeveloper(T* pDeveloper) {
        ASSERT(pDeveloper != NULL);
        m_pDeveloper = pDeveloper;
    }
    ~SmartDeveloper() {
        AfxMessageBox("I'm smart so I'll get paid.");
    }
    SmartDeveloper &
      operator=(const SmartDeveloper& rDeveloper) {
        return *this;
    }
     T* operator->() const {
        AfxMessageBox("About to de-reference pointer. Make /
                    sure everything's okay. ");
        return m_pDeveloper;
```

```
    }
};
```

上面列出的 SmartDeveloper 模板包装了一个指针——可以是任何一个指针。因为 SmartDeveloper 类是基于模板的，所以，它可以提供一般化的功能，而不管与该类联系在一起的类型是什么。您可以将模板看作是编译器特许的宏：模板是指类或者函数的声明，并且这些类或者函数可以应用到任何数据类型上。

我们希望上面的 Smart 指针可以处理所有的开发人员类型，包括那些使用 Visual Basic、Visual C++、C#和 Delphi(以及其他语言)的开发人员。最上边的 template<class T>语句就可以实现这个想法。SmartDeveloper 模板包含一个指针 m_pDeveloper，它所指向的开发人员类型正是该类将被定义的那个类型。SmartDeveloper 构造函数的参数是一个指向该类型的指针，构造函数将该指针参数赋给 m_pDeveloper。注意，如果客户传递一个 NULL 参数来构造 SmartDeveloper 的话，那么构造函数将产生一个断言(assertion)。

除了包装一个指针以外，SmartDeveloper 也实现了几个操作符。最重要的一个是->操作符(成员选择操作符)。这个操作符是任何一个 Smart 指针类真正承担负荷的地方。正是因为重载了成员选择操作符，所以就把一个常规的类转变成了一个 Smart 指针。通常情况下，在一个常规的 C++哑指针上使用成员选择操作符，这就告诉编译器选择一个属于该指针所指的类或结构的成员。通过重载成员选择操作符，您为客户提供了一种方法，允许客户在每次调用一个方法时可以钩入或者调用某些样板代码。在 SmartDeveloper 例子中，灵巧的开发人员(smart develeper)将确保在工作之前其工作区域是有序的。(这个例子多多少少是编造出来的。例如，在现实环境中，您可能需要插入一个调试钩子。)

在一个类中加入->操作符将使得该类的行为与 C++的内置指针类似。为了使 Smart 指针在其他方面也能像内置的 C++指针一样工作，Smart 指针类必须要实现其他的一些标准操作符，例如解引用操作符和赋值操作符。

25.5.4　使用 Smart 指针

实际上，使用 Smart 指针与使用常规的内置 C++指针没有什么不同。让我们首先来看一看客户是如何使用普通的开发人员类的：

```
void UseDevelopers() {
    CVBDeveloper VBDeveloper;
    CCPPDeveloper CPPDeveloper;
    VBDeveloper.DoTheWork();
    CPPDeveloper.DoTheWork();
}
```

这里丝毫没有令人惊讶之处——执行这一段代码只不过是让 Visual Basic 开发人员和 C++开发人

员进来并做他们的工作。但您想用的是灵巧开发人员(smart developer)——他们在真正开始编程之前，确保首先完成设计。下面的代码将 Visual Basic 开发人员和 C++开发人员对象包装到 Smart 指针类中：

```
void UseSmartDevelopers {
    CVBDeveloper VBDeveloper;
    CCPPDeveloper CPPDeveloper;

    SmartDeveloper<CVBDeveloper> smartVBDeveloper(&VBDeveloper);
    SmartDeveloper<CCPPDeveloper> smartCPPDeveloper(&CPPDeveloper);

    smartVBDeveloper->DoTheWork();
    smartCPPDeveloper->DoTheWork();
}
```

在上面的例子中，客户并没有引入老的开发人员来做工作(前一个例子是这样做的)，而是请灵巧开发人员来做工作。灵巧开发人员将在编码之前自动准备好系统设计。

25.5.5 Smart 指针和 COM

尽管上一个例子是我们为了使事情更加有趣而编造的，但是在现实世界中 Smart 指针确实很有用。其中一个应用是使客户端 COM 程序设计更加简单。

Smart 指针经常被用来实现引用计数。因为引用计数是一个非常通用的操作，所以，将客户端引用计数的管理工作放到一个 Smart 指针中是非常有意义的。

因为您现在对 COM 已经很熟悉了，所以您一定很清楚 COM 对象对外界仅暴露出接口。对于 C++客户而言，所谓接口只不过是一些纯抽象基类，而 C++客户对待接口或多或少就像对待普通的 C++对象一样。然而，正如在前面的章节中您所见到的，COM 对象与常规 C++对象有所不同。COM 对象位于二进制层次上。正因为如此，所以我们可以用与语言无关的方法来创建和销毁 COM 对象。客户需要通过调用 API 函数来创建 COM 对象。大多数 COM 对象都使用一个引用计数来决定什么时候该从内存中将自己删除。一旦创建了一个 COM 对象，客户对象就可以通过引用多个接口(它们属于同一个 COM 对象)，用多种方式来引用这个 COM 对象。另外，几个不同的客户也可以与同一个 COM 对象进行对话。在这些情况下，只要一个 COM 对象被客户引用了，那么它就必须保持活跃的状态。大多数 COM 对象在不再被任何客户引用之后，就会将自己销毁。COM 对象用引用计数来实现这种自我销毁的功能。

为了支持这种引用计数方案，COM 定义了两条规则，以便从客户端来管理 COM 接口。第一条规则是，若要为一个 COM 接口新建一份拷贝，就应该使这个对象的引用计数增加 1。第二条规则是，当客户用完了接口指针时，就应该将它们释放。引用计数是 COM 中比较难以正确处理的一个方面，特别是从客户端的角度来看。跟踪 COM 接口的引用计数是 Smart 指针的理想用途。

例如，Smart 指针的构造函数可能带一个活跃的接口指针作为一个参数，并且将内部的指针设置为该接口指针。然后，析构函数可能会调用该接口指针的 Release 函数以释放接口，从而当 Smart 指针被删除或者离开作用域时，该接口指针将会被自动释放。另外，Smart 指针可以用来管理那些被拷贝的 COM 接口。

例如，假定您创建了一个 COM 对象，并且拥有它的一个接口指针。现在您需要对这个接口指针做一份拷贝(可能要将它作为一个输出参数传递出去)。在原始 COM 的层次上，您必须执行几个步骤。首先，您必须释放老的接口指针。然后，您需要把老的指针拷贝给新的指针。最后，您必须在新拷贝的接口指针上调用 AddRef。不管您使用的是什么样的接口，这些步骤都是必需的，这就使得这个处理过程很适合用样板代码来实现。为了在 Smart 指针类中实现此处理过程，您所需要做的就是重载赋值操作符。然后，客户就能够将老的指针赋值给新的指针。Smart 指针完成了管理接口指针的所有工作，从而减轻了客户的负担。

25.5.6　ATL 的 Smart 指针

ATL 对客户端 COM 开发的支持大部分位于两个 ATL Smart 指针之中：CComPtr 和 CComQIPtr。CComPtr 是一个基本的 Smart 指针，它包装了 COM 接口指针。CComQIPtr 又增加了一点智能，它将一个 GUID(用作接口 ID)与一个 Smart 指针联系起来。CComPtr 又将它的大部分功能提炼出来，放到一个名为 CComPtrBase 的类中。下面我们首先从 CComPtrBase 开始介绍。

25.5.7　CComPtrBase 类

CComPtrBase 为那些使用基于 COM 的内存函数的 Smart 指针类提供了一个基类。下面是 CComPtrBase 类：

```
template <class T>
class CComPtrBase
{
protected:
    CComPtrBase() throw()
    {
        p = NULL;
    }
    CComPtrBase(int nNull) throw()
    {
        ATLASSERT(nNull == 0);
        (void)nNull;
        p = NULL;
    }
```

```
    CComPtrBase(T* lp) throw()
    {
        p = lp;
        if (p != NULL)
            p->AddRef();
    }
public:
    typedef T _PtrClass;
    ~CComPtrBase() throw()
    {
        if (p)
            p->Release();
    }
    operator T*() const throw()
    {
        return p;
    }
    T& operator*() const throw()
    {
        ATLASSERT(p!=NULL);
        return *p;
    }
    //The assert on operator& usually indicates a bug.  If this is really
    //what is needed, however, take the address of the p member explicitly.
    T** operator&() throw()
    {
        ATLASSERT(p==NULL);
        return &p;
    }
    _NoAddRefReleaseOnCComPtr<T>* operator->() const throw()
    {
        ATLASSERT(p!=NULL);
        return (_NoAddRefReleaseOnCComPtr<T>*)p;
    }
    bool operator!() const throw()
    {
        return (p == NULL);
    }
    bool operator<(T* pT) const throw()
    {
        return p < pT;
    }
    bool operator==(T* pT) const throw()
    {
        return p == pT;
```

```
}
// Release the interface and set to NULL
void Release() throw()
{
    T* pTemp = p;
    if (pTemp)
    {
        p = NULL;
        pTemp->Release();
    }
}
// Compare two objects for equivalence
bool IsEqualObject(IUnknown* pOther) throw()
{
    if (p == pOther)
        return true;

    if (p == NULL || pOther == NULL)
        return false; // One is NULL the other is not

    CComPtr<IUnknown> punk1;
    CComPtr<IUnknown> punk2;
    p->QueryInterface(__uuidof(IUnknown), (void**)&punk1);
    pOther->QueryInterface(__uuidof(IUnknown), (void**)&punk2);
    return punk1 == punk2;
}
// Attach to an existing interface (does not AddRef)
void Attach(T* p2) throw()
{
    if (p)
        p->Release();
    p = p2;
}
// Detach the interface (does not Release)
T* Detach() throw()
{
    T* pt = p;
    p = NULL;
    return pt;
}
HRESULT CopyTo(T** ppT) throw()
{
    ATLASSERT(ppT != NULL);
    if (ppT == NULL)
        return E_POINTER;
```

```
        *ppT = p;
        if (p)
            p->AddRef();
        return S_OK;
    }
    HRESULT SetSite(IUnknown* punkParent) throw()
    {
        return AtlSetChildSite(p, punkParent);
    }
    HRESULT Advise(IUnknown* pUnk, const IID& iid, LPDWORD pdw) throw()
    {
        return AtlAdvise(p, pUnk, iid, pdw);
    }
    HRESULT CoCreateInstance(REFCLSID rclsid,
                             LPUNKNOWN pUnkOuter = NULL,
                             DWORD dwClsContext = CLSCTX_ALL) throw()
    {
        ATLASSERT(p == NULL);
        return ::CoCreateInstance(rclsid, pUnkOuter, dwClsContext,
                                  __uuidof(T), (void**)&p);
    }
    HRESULT CoCreateInstance(LPCOLESTR szProgID,
                             LPUNKNOWN pUnkOuter = NULL,
                             DWORD dwClsContext = CLSCTX_ALL) throw()
    {
        CLSID clsid;
        HRESULT hr = CLSIDFromProgID(szProgID, &clsid);
        ATLASSERT(p == NULL);
        if (SUCCEEDED(hr))
            hr = ::CoCreateInstance(clsid, pUnkOuter, dwClsContext,
                                    __uuidof(T), (void**)&p);
        return hr;
    }
    template <class Q>
    HRESULT QueryInterface(Q** pp) const throw()
    {
        ATLASSERT(pp != NULL);
        return p->QueryInterface(__uuidof(Q), (void**)pp);
    }
    T* p;
};
```

CComPtrBase 是一个非常基础的 Smart 指针。请注意类型为 T(这是由模板参数引入的类型)的数据成员 p。CComPtrBase 的构造函数在该指针上执行 AddRef，而析构函数则释放指针——这并不值得惊奇。CComPtrBase 也具有为包装一个 COM 接口所需要的全部操作符。在此需要特别说明的是赋值

操作符。它所做的赋值是针对底层原始指针的重新赋值。赋值操作符调用了一个名为 AtlComPtrAssign 的函数：

```
ATLINLINE ATLAPI_(IUnknown*) AtlComPtrAssign(IUnknown** pp,
                                             IUnknown* lp)
{
    if (lp != NULL)
        lp->AddRef();
    if (*pp)
        (*pp)->Release();
    *pp = lp;
    return lp;
}
```

AtlComPtrAssign 进行了一个盲(blind)指针赋值操作，它首先在要赋值的指针上调用 AddRef，然后在被赋值的指针上调用 Release。稍后您将会看到该函数的另一个版本(它调用了 QueryInterface 函数)。

CComPtrBase 的主要功能是，它可以在某种程度上帮助程序员管理一个指针的引用计数。类层次中的下一个是 CComPtr——这是您在实际应用程序中需要用到的一个类。

25.5.8　CComPtr 类

因为 CComPtr 是从 CComPtrBase 派生的，所以它包含了 CComPtrBase 的所有接口指针管理功能。CComPtr 不仅可以帮助您管理 AddRef 和 Release 操作，还可以管理代码的布局。通过展示一些代码将有助于说明 CComPtr 的用途。假定客户代码需要 3 个接口指针才能完成它的工作，如下所示：

```
void GetLottaPointers(LPUNKNOWN pUnk){
    HRESULT hr;
    LPPERSIST pPersist;
    LPDISPATCH pDispatch;
    LPDATAOBJECT pDataObject;
    hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
    if(SUCCEEDED(hr)) {
        hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *)
                                  &pDispatch);
        if(SUCCEEDED(hr)) {
            hr = pUnk->QueryInterface(IID_IDataObject,
                                      (LPVOID *) &pDataObject);
            if(SUCCEEDED(hr)) {
                DoIt(pPersist, pDispatch, pDataObject);
                pDataObject->Release();
            }
```

```
            pDispatch->Release();
        }
        pPersist->Release();
    }
}
```

您可能会使用很有争议的 goto 语句(您的同事可能会因此而笑话您)，以便使代码看起来更为整齐和清晰，就像这样：

```
void GetLottaPointers(LPUNKNOWN pUnk){
    HRESULT hr;
    LPPERSIST pPersist;
    LPDISPATCH pDispatch;
    LPDATAOBJECT pDataObject;

    hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&pPersist);
    if(FAILED(hr)) goto cleanup;

    hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &pDispatch);
    if(FAILED(hr)) goto cleanup;

    hr = pUnk->QueryInterface(IID_IDataObject,
                          (LPVOID *) &pDataObject);
    if(FAILED(hr)) goto cleanup;
    DoIt(pPersist, pDispatch, pDataObject);

cleanup:
    if (pDataObject) pDataObject->Release();
    if (pDispatch) pDispatch->Release();
    if (pPersist) pPersist->Release();
}
```

但这可能并不是您所希望的好的解决方案。而通过使用 CComPtr，则同样的代码不仅看起来更为整齐，而且也更加容易阅读，如下所示：

```
void GetLottaPointers(LPUNKNOWN pUnk){
    HRESULT hr;
    CComPtr<IUnknown> persist;
    CComPtr<IUnknown> dispatch;
    CComPtr<IUnknown> dataobject;

    hr = pUnk->QueryInterface(IID_IPersist, (LPVOID *)&persist);
    if(FAILED(hr)) return;

    hr = pUnk->QueryInterface(IID_IDispatch, (LPVOID *) &dispatch);
    if(FAILED(hr)) return;
```

```
    hr = pUnk->QueryInterface(IID_IDataObject,
                          (LPVOID *) &dataobject);
    if(FAILED(hr)) return;

    DoIt(pPersist, pDispatch, pDataObject);

    // Destructors call release...
}
```

现在，您可能会感到很奇怪，为什么 CComPtr 没有包装 QueryInterface。毕竟，QueryInterface 是使用引用计数的一个重要场所。要想为 Smart 指针增加 QueryInterface 支持，这需要通过某种途径将 GUID 与 Smart 指针联系起来。CComPtr 是在 ATL 的第一版中引入的。Microsoft 并没有破坏已有的代码基础，而是引入了 CComPtr 的加强版本，其类名为 CComQIPtr。

25.5.9　CComQIPtr 类

下面是 CComQIPtr 的定义：

```
template <class T, const IID* piid = &__uuidof(T)>
class CComQIPtr : public CComPtr<T>
{
public:
    CComQIPtr() throw()
    {
    }
    CComQIPtr(T* lp) throw() :
        CComPtr<T>(lp)
    {
    }
    CComQIPtr(const CComQIPtr<T,piid>& lp) throw() :
        CComPtr<T>(lp.p)
    {
    }
    CComQIPtr(IUnknown* lp) throw()
    {
        if (lp != NULL)
            lp->QueryInterface(*piid, (void **)&p);
    }
    T* operator=(T* lp) throw()
    {
        return static_cast<T*>(AtlComPtrAssign((IUnknown**)&p, lp));
    }
    T* operator=(const CComQIPtr<T,piid>& lp) throw()
```

```
    {
        return static_cast<T*>(AtlComPtrAssign((IUnknown**)&p, lp.p));
    }
    T* operator=(IUnknown* lp) throw()
    {
        return static_cast<T*>(AtlComQIPtrAssign((IUnknown**)&p,
                              lp, *piid));
    }
};

//Specialization to make it work
template<>
class CComQIPtr<IUnknown, &IID_IUnknown> : public CComPtr<IUnknown>
{
public:
    CComQIPtr() throw()
    {
    }
    CComQIPtr(IUnknown* lp) throw()
    {
        //Actually do a QI to get identity
        if (lp != NULL)
            lp->QueryInterface(__uuidof(IUnknown), (void **)&p);
    }
    CComQIPtr(const CComQIPtr<IUnknown,&IID_IUnknown>& lp) throw() :
        CComPtr<IUnknown>(lp.p)
    {
    }
    IUnknown* operator=(IUnknown* lp) throw()
    {
        //Actually do a QI to get identity
        return AtlComQIPtrAssign((IUnknown**)&p, lp,
                                 __uuidof(IUnknown));
    }
    IUnknown* operator=(const CComQIPtr<IUnknown,&IID_IUnknown>& lp)
        throw()
    {
        return AtlComPtrAssign((IUnknown**)&p, lp.p);
    }
};
```

CComQIPtr 与 CComPtr 的区别在于第二个模板参数 piid——即接口的 GUID。这个 Smart 指针有几个构造函数：一个默认构造函数、一个拷贝构造函数、一个带一未指定类型的接口指针的构造函数，以及一个接受 IUnknown 接口作为参数的构造函数。对于最后一个构造函数，请注意，如果开发人员创建了这种类型的一个对象，并用一个老的 IUnk-nown 指针对它进行初始化，则 CComQIPtr 将会用

GUID 模板参数来调用 QueryInterface。同时也要注意，对 IUnknown 指针的赋值调用了 AtlComQIPtrAssign 来进行。正如您所想像的那样，由于使用了 GUID 模板参数，所以，实际上 AtlComQIPtrAssign 执行了一个 QueryInterface。

使用 CComQIPtr

下面的代码是某些 COM 客户使用 CComQIPtr 的一种可能的做法：

```
void GetLottaPointers(ISomeInterface* pSomeInterface){
    HRESULT hr;
    CComQIPtr<IPersist, &IID_IPersist> persist;
    CComQIPtr<IDispatch, &IID_IDispatch> dispatch;
    CComPtr<IDataObject, &IID_IDataObject> dataobject;

    dispatch = pSomeInterface;   // implicit QI
    persist = pSomeInterface;    // implicit QI
    dataobject = pSomeInterface; // implicit QI

    DoIt(persist, dispatch, dataobject); // send to a function
                                         //  that needs IPersist*,
                                         //  IDispatch*, and
                                         //  IDataObject*

    // Destructors call release...
}
```

当您期望使用 Java 风格或者 Visual Basic 风格的类型转换时，CComQIPtr 是非常有用的。注意，上面列出的代码并不需要任何对 QueryInterface 或者 Release 的调用。这些调用都是自动进行的。

25.5.10　ATL Smart 指针的问题

在某些地方 Smart 指针是非常便利的(例如在 CComPtr 例子中，我们通过 Smart 指针消除了 Goto 语句)。不幸的是，C++ Smart 指针并不是程序员期望用来解决引用计数和指针管理问题的万能药。Smart 指针只是将问题转移到了另一个不同的层次上。

如果您原来的代码没有使用 Smart 指针，现在要将它转换为使用 ATL Smart 指针，在这种情形下您必须要谨慎对待 Smart 指针。问题在于，ATL Smart 指针没有隐藏 AddRef 和 Release 调用。这意味着，您必须要理解 Smart 指针是如何工作的，而不仅仅是关心如何调用 AddRef 和 Release。

例如，假设有下面这样的代码：

```
void UseAnInterface(){
    IDispatch* pDispatch = NULL;
```

```
    HRESULT hr = GetTheObject(&pDispatch);
    if(SUCCEEDED(hr)) {
        DWORD dwTICount;
        pDispatch->GetTypeInfoCount(&dwTICount);
         pDispatch->Release();
    }
}
```

如果任意地将上面这段代码进行转换以使用 Smart 指针，如下所示：

```
void UseAnInterface() {
    CComPtr<IDispatch> dispatch = NULL;

    HRESULT hr = GetTheObject(&dispatch);
    if(SUCCEEDED(hr)) {
        DWORD dwTICount;
        dispatch->GetTypeInfoCount(&dwTICount);
        dispatch->Release();
    }
}
```

那么因为 CComPtr 和 CComQIPtr 没有隐藏对 AddRef 和 Release 的调用，所以，在以上这段盲目转换的代码中，当通过 Smart 指针 dispatch 来调用 Release 函数时，就会出现问题。IDispatch 接口执行它自己的 Release，所以这段代码调用 Release 两次：第一次通过显式调用 dispatch->Release()；第二次是在函数的结束括号处被隐式调用的。

另外，ATL 的 Smart 指针包含隐式的类型转换操作符，通过这些类型转换操作符，您可以将 Smart 指针赋值给原始的指针。在这种情况下，实际上，这已经使引用计数变得很混乱了。

总体而言，尽管 Smart 指针使客户端 COM 开发的某些方面变得更为方便了，但它们不是十分安全的。如果您希望安全地使用 Smart 指针，那么您仍然必须要了解一些关于 Smart 指针是如何工作的知识。

25.6 服务器端 ATL 程序设计

尽管 ATL 中有相当一部分是针对客户端开发的支持(例如 Smart 指针和 BSTR 包装类等)，但是，ATL 之所以存在的主要原因还在于支持 COM 服务器。接下来，您将会看到 ATL 的总体概况，以便理解 ATL 中各个部分是如何配合工作的。然后，我们将用 ATL 重新实现飞船的示例，以便学习 ATL Object Wizard，从而感觉一下怎样利用 ATL 来编写 COM 类。

25.6.1　ATL 和 COM 类

作为一个 COM 类的开发者，您的任务是将函数表与函数的实现包装起来，并保证 QueryInterface、AddRef 和 Release 如预期般地工作。如何做到这一点，这是您自己的事情。至于用户，他们并不关心您到底使用了什么样的实现方法。到现在为止，您已经看到了两种基本的方法——利用接口多继承的原始 C++方法，以及利用宏与嵌套类的 MFC 方法。ATL 实现 COM 类的方法与这两种方法都有所不同。

首先将原始的 C++方法与 MFC 方法做一比较。在前面章节曾经看到过，在使用原始 C++来开发 COM 类的方法中，需要将一个 C++类从多个(至少一个)COM 接口中继承过来，然后再为这个 C++类编写所有的代码。这时，您必须用手工方式来加入任何额外的特性(例如支持 IDispatch 或者 COM 聚合)。而用 MFC 来建立 COM 类，则涉及到使用大量的宏(这些宏定义了嵌套类，并且每一个嵌套类实现一个接口)。MFC 支持 IDispatch 和 COM 聚合——您无需做太多的工作就可以让这些特性起作用。然而，您很难在一个 COM 类上添加一些新的接口而又不用做很多的工作。(正如您在第 22 章中所看到的，MFC 的 COM 支持使用了一些长长的宏。)

用 ATL 来实现 COM 类，则要求将一个 C++类从几个模板类继承过来。然而，通过 ATL 中的类模板，Microsoft 已经完成了所有为实现 IUnknown 而需要的工作。

让我们现在开始创建一个针对飞船例子的 COM 类。同以前一样，首先在 Visual C++ .NET 的 File 菜单中选择 New Project。在 New Project 对话框中(如图 25.1 所示)，从 Visual C++ Projects 夹中选择 ATL Project。为项目取一个有意义的名字，例如 ATLSpace-ShipSvr，然后单击 OK。ATL Project Wizard 就会启动起来。

图 25.1　在 New Project 对话框中选择 ATL Project Wizard

25.6.2 ATL Project 选项

在 ATL Project Wizard 的 Application Settings 页面中(如图 25.2 所示)，您可以选择项目的服务器类型。该向导允许您选择创建一个动态链接库(Dynamic-Link Library)、一个可执行文件(Executable)或者一个服务(Service)。如果您选择了 DLL 选项，并且没有选中 Attributed 选项，那么，将代理/存根代码合并到 DLL 的选项，以及在 ATL 项目中包含 MFC 的选项都将会被激活。同时还有一个支持 COM+ 1.0 的选项。

图 25.2 ATL Project Wizard 的 Application Settings 页面

如果您选择了 DLL 作为服务器类型，那么向导将产生所有使服务器 DLL 满足 COM 环境所必要的组成部分。在这些组成部分中包括下列知名的 COM 函数：DllGetClassOb-ject、DllCanUnloadNow、DllRegisterServer 和 DllUnregisterServer。此外还将包含一套针对 DLL 的服务器生存期管理机制。

如果您认为您的 DLL 有可能会在一个代理进程中作为进程外组件来运行，那么可选择 Allow Merging Of Proxy/Stub Code 选项，这样就可以把您的所有组件打包到一个二进制文件中。(传统上，代理/存根代码被作为一个单独的 DLL 来发布。)采用这种方法，您只需发布一个 DLL 就可以了。如果您绝对确定要在 DLL 中包含 MFC，那么您可以毫不犹豫地选择 Support MFC 选项。对 MFC 的支持将导致在 StdAfx.h 文件中包含 AfxWin.h 和 AfxDisp.h，并且会将该项目链接到当前版本的 MFC 导入库上。虽然使用 MFC 非常方便，而且用起来几乎会上瘾，但是，您一定要清楚在包含 MFC 之后所带来的依赖性。您也可以选择 Support COM+ 1.0，以增加对 COM+ 1.0 运行时服务的支持。

如果您选择了要产生 Executable EXE 服务器，则 ATL Project Wizard 将产生出可被编译为 EXE 文件的代码。EXE 通过 CoRegisterClassObject 和 CoRevokeClassObject，将类对象正确地注册到操作系统中。该项目也将会插入正确的代码来管理可执行服务器的生存期。最后，如果您选择了

Service(EXE)选项，则向导将会加入必要的针对服务的代码。

用 ATL Project Wizard 来编写一个轻量级的 COM 服务器，这将会生成一个项目文件，通过该项目文件您可以编译该项目。该项目文件将项目中的所有源代码联系在一起，并且为每一个文件维护了正确的编译指令。

> **属性化的 ATL 和经典的 ATL**[1]
>
> 我曾经提到了 ATL Project Wizard 的 Application Settings 页面上的 Attributed 选项。Attribute(属性)是 Visual C++ .NET 的一个新特性，它的设计目标是简化 COM 编程和.NET 公共语言运行时库的开发。使用属性就好像是在您的源代码中加入脚注一样。通过在源代码中加入属性，您可以给编译器一些指令，让它与提供者 DLL(provider DLL)一起协作，以便在所生成的目标文件中插入代码或者修改代码。这些属性可以帮助 Visual C++ .NET 创建 IDL 文件、接口、类型库和其他的 COM 元素。Visual C++ .NET 的向导和 Properties 窗口都支持属性。
>
> 如果您对 IDL 比较熟悉的话，您一定很容易理解属性。在 IDL 中可以找得到的大多数独立声明都变成了属性，它们可以直接出现在源代码中，而不是在 IDL 代码中。
>
> C++很早以前就已经面世了——甚至在 Windows 成为一个流行的开发平台之前。您在前面已经看到过 COM 了，其实 C++并不是生成 DLL 和组件的最佳方案——特别是因为该语言中内置了许多复杂的元素。这也正是为什么 COM 存在的原因。在许多方面，COM 充分利用了 C++将虚函数表映射到函数实现的优势，从而使得 C++ DLL 可以直接发布。Visual C++ .NET 引入了属性则又往前走了一大步。
>
> 属性编程扩展了 C++，同时也没有打破 C++语言的传统结构。属性允许您通过一些提供者 DLL 来为语言加入功能。属性的首要目标是简化 COM 组件的编写。您可以将属性应用到大多数的 C++语言结构上，包括类、数据成员和成员函数。
>
> 在本章后面我们将讨论经典的 ATL 编程和属性化的 ATL 编程。

25.6.3　创建一个经典的 ATL COM 类

一旦您创建了一个 COM 服务器，您就可以在该 COM 服务器中加入 COM 类了。幸运的是，利用 ATL Simple Object Wizard 可以很容易地做到这一点，如图 25.3 所示。为了启动该向导，请从 Project

---

1 译者注：这里的属性指的是 Attribute，它描述了该属性所依附的主体的特性。它与自动化技术中的属性(property)没有必然的关系。请不要混淆这两个概念。在后文可能引起混淆的地方，我都将标出相应的英文单词。

菜单中选择 Add Class，然后从列出的模板中选择 ATL Simple Object。

> 说明 如果要创建一个经典的 COM DLL，那么请注意，一定不要选中 ATL Project Wizard 的 Application Settings 页面上的 Attributed 复选框。

利用 ATL Simple Object Wizard 来产生一个新对象，将导致在项目中加入一个 C++源文件和一个头文件，在这两个文件中包含了新类的定义和实现。另外，该向导也会在 IDL 代码中加入一个接口。尽管该向导负责管理 IDL 文件的骨架，但是，如果您想要编写出有效的 COM 接口(很快您就会看到)，那么您仍需要在一定程度上理解 IDL。

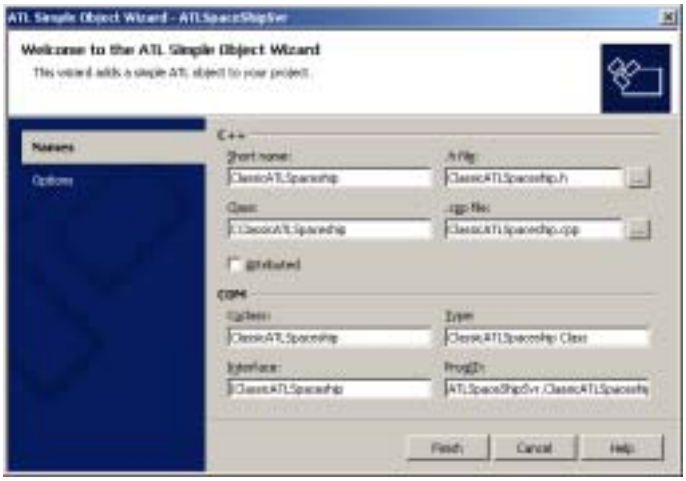

图 25.3 利用 ATL Simple Object Wizard 在项目中插入一个新的 ATL COM 类

通过 ATL Simple Object Wizard 的 Options 页面，您可以为 COM 类选择线程模型，并且指定该接口是一个双接口(以 IDispatch 为基接口)或是一个自定义接口。同时您还可以选择该 COM 类将如何支持聚合。并且，通过该向导，您可以很容易地在 COM 类中包含 ISupportErrorInfo 接口和连接点，也可以加入 Internet Explorer 宿主(hosting)支持。最后，对于指定了线程模型为 Both 或者 Neutral 的对象，您可以选择聚合到自由线程列集器(free-threaded marshaler)。

25.6.4 套间和线程

要想掌握 COM，您必须理解，COM 是以抽象的概念为中心的——它向客户隐藏尽可能多的信息。COM 向客户隐藏的信息之一是：该 COM 类是否是线程安全的。客户应该可以像正常情况下那样使用一个对象，而不用担心一个对象是否已经正确地将对它的访问进行了序列化——即，是否正确地保护了对它内部数据的访问。COM 定义了套间(apartment)的概念，以提供这种抽象。

一个套间定义了一个执行环境，或者线程，它可以容纳多个接口指针。通过调用 CoInitialize 函

数族中的一个函数，线程就可以进入到一个套间中。(CoInitialize 函数族包括 CoInitialize、CoInitializeEx 和 OleInitialize。)然后，COM 要求对一个接口指针的所有方法调用都必须在初始化该指针的套间内执行(换句话说，必须从调用了 CoCreateInstance 的同一个线程来执行)。COM 定义了两种类型的套间——单线程套间和多线程套间。单线程套间只能容纳一个线程，而多线程套间可以容纳多个线程。一个进程只能有一个多线程套间，但可以有多个单线程套间。一个套间可以容纳任何数量的 COM 对象。

单线程套间可以保证，凡是在该套间内创建的 COM 对象，它们的方法调用被将序列化，这是通过一个远程层(remoting layer)来做到的。而在多线程套间内创建的 COM 对象没有这样的性质。有一种办法可以帮助您记住这两种套间之间的区别，您可以这样来想：在多线程套间中实例化一个 COM 对象，就好像将一段数据放在全局作用域范围内，多个线程可以同时访问这段数据；而在单线程套间内实例化一个 COM 对象，就好像将数据放在只有一个线程可访问的作用域范围内。基本的一点是，期望生存在多线程套间内的 COM 类最好是线程安全的，而仅仅生存在自己的套间内的 COM 类，则无需担心对它们的数据的并发访问。

如果一个 COM 对象生存在与客户不同的进程空间中，那么它的方法调用将通过远程层，被自动序列化。然而，生存在 DLL 中的 COM 对象可能应该提供它自己的内部保护(例如使用临界区)，而不是让远程层来保护它。一个 COM 类通过注册表设置来对外公布它的线程安全性。这个命名值位于注册表的 HKEY_CLASSES_ROOT 中的 CLSID 下，如下所示：

```
[HKCR\CLSID\{some GUID ...}\InprocServer32]
@="C:\SomeServer.DLL"
ThreadingModel=<thread model>
```

ThreadingModel 可以是下面五个值之一：Single、Both、Free、Apartment 或者 Neutral，或者也可以是空的。ATL 支持所有当前的线程模型。下面简要地说明了每一个值所代表的含义：

- Single 或者空，表示这个类仅在主线程(客户创建的第一个单线程套间中的线程)内运行。
- Both，表示这个类是线程安全的，它既可以在单线程套间中运行，也可以在多线程套间中运行。该值告诉 COM，使用与客户同样类型的套间。
- Free，表示这个类是线程安全的。该值告诉 COM，该类的对象必须位于多线程套间内。
- Apartment，表示这个类不是线程安全的，它必须生存在自己的单线程套间内。
- Neutral，表示这个类可以生存在线程中立的套间(thread-neutral apartment)中。它遵循与多线程类同样的规则，但是它可以在任何一个线程中运行。

当您在 ATL Simple Object Wizard 中选择一个线程模型时，该向导将根据您的选择，在类中插入不同的代码。例如，如果您选择了 Apartment 模型，则 Object Wizard 将使您的类从 CComObjectRootEx 派生，并包含 CComSingleThreadModel 作为参数，如下所示：

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
```

```
    public CComCoClass<CClassicATLSpaceship,
                       &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
                         &IID_IClassicATLSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{
……
};
```

CComSingleThreadModel 模板参数为 IUnknown 引入了更加有效的标准递增和递减操作(因为对该类的访问是自动被序列化的)。另外，ATL Simple Object Wizard 也将使该类在注册表中插入正确的线程模型值。如果您在该向导中选择了 Single 选项，那么该类将使用 CComSingleThreadModel，但是注册表中的 ThreadingModel 值保持为空。

选择 Both 或者 Free 选项将使该类使用 CComMultiThreadModel 模板参数，它将采用 Win32 中线程安全的递增和递减操作 InterlockedIncrement 和 InterlockedDecrement。例如，一个自由线程类的定义看起来像这样：

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CClassicATLSpaceship,
                       &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
                         &IID_IClassicATLSpaceship,
                         &LIBID_SPACESHIPSVRLib>
{
……
};
```

选择 Both 线程模型将导致在注册表中插入 Both 作为 ThreadingModel 值。选择 Free 将导致在注册表中插入 Free 作为 ThreadingModel 值。

25.6.5 连接点和 ISupportErrorInfo

在 COM 类中加入连接点是很容易的。选择 Connection Points 复选框将使该类从 IConnectionPointImpl 派生。该选项也在类中加入了一个空的连接映射表。在一个类中加入连接点(例如一个事件集)简单到只需执行以下四步：

1. 在 IDL 文件中定义回调接口。
2. 利用 ATL 代理生成器创建一个代理。
3. 在 COM 类中加入代理类。
4. 在连接点映射表中加入连接点。

ATL 也包含了对 ISupportErrorInfo 的支持。ISupportErrorInfo 接口可以确保错误信息正确地沿着调用链向上传播。那些使用错误处理接口的 OLE 自动化对象必须实现 ISup-portErrorInfo。在 ATL Simple Object Wizard 中选择 Support ISupportErrorInfo 将使得该 ATL 类从 ISupporErrorInfoImpl 派生。

25.6.6　自由线程列集器

您可以选择 Free Threaded Marshaler 选项，以便将 COM 自由线程列集器(COM free-threaded marshaler)聚合到您的类中。前面提到过，该选项仅适用于那些指定了 Both 或 Neutral 作为线程模型的对象。由此而产生的类将在它的 FinalConstruct 函数中调用 CoCreateFreeThreadedMarshaler，以便聚合 COM 的自由线程列集器。自由线程列集器允许线程安全的对象绕过标准列集过程，所谓标准列集过程是指接口方法在跨套间被调用时所实施的处理过程。这样带来的好处是，一个套间中的线程在访问另一个套间中的接口方法时就好像它们位于同一个套间中一样。

这将会大大加速跨套间的调用。自由线程列集器通过实现 IMarshal 接口来做到这一点。当客户向对象请求一个接口时，远程层调用 QueryInterface，请求 IMarshal 接口。如果该对象实现了 IMarshal(在这里的情形中，对象实现了 IMarshal，因为 ATL Simple Object Wizard 也在该类的接口中加入了一个条目，以处理对于 IMarshal 的 QueryInterface 请求)，并且列集请求正在进行之中，那么，自由线程列集器实际上将该指针拷贝到列集数据包中。通过这种方式，客户将会接收到一个实际指向对象的指针。客户可以直接与对象进行对话，而不必通过代理和存根。当然，如果您选择了 Free Threaded Marshaler 选项，那么，对象中的所有数据最好都是线程安全的。因此，如果您选择了这个选项，请一定要非常谨慎。

25.6.7　用经典的 ATL 来实现飞船类

我们将利用 ATL Simple Object Wizard 所提供的默认设置来创建飞船类。例如，飞船类将有一个双接口，所以客户可以从诸如 Web 页面上的 JScript 这样的环境中对它进行访问。另外，飞船类将是一个套间模型对象，这意味着 COM 将会管理大多数的并发问题。唯一需要您在 ATL Simple Object Wizard 中提供的信息是一个恰当的名字。在 Names 页面上的 Short Name 文本框中输入一个值，例如 **ClassicAtlSpaceship**。

现在，您不需要再设置任何其他的选项。例如，您不需要设置 Connection Points 选项，因为我们将在下一章中讨论连接。您总是可以通过手工方式在以后加入连接点。

下面是向导产生的类定义：

```
// CClassicATLSpaceship

class ATL_NO_VTABLE CClassicATLSpaceship :
```

```
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship,
                    &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
        &IID_IClassicATLSpaceship,
        &LIBID_ATLSpaceShipSvrLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
……
};
```

尽管 ATL 包含了相当多的面向 COM 的 C++类，但是，在上面飞船类的基类列表中列出的这些类，已经足够让您了解 ATL 是如何工作的了。

绝大多数普通的 ATL COM 对象都从三个基类中派生：CComObjectRoot、CComCo-Class、IDispatch。CComObjectRoot 实现了 IUnknown，并管理类的身份。这意味着，CCom-ObjectRoot 实现了 AddRef 和 Release，并钩住了 ATL 的 QueryInterface 机制。CComCoClass 管理 COM 类的类对象和某些一般化的错误报告。在上面的类定义中，CComCoClass 加入了类对象，它知道如何创建 CClassicATLSpaceship 对象。最后，ATL Simple Object Wizard 产生的代码包含了一个 IDispatch 的实现，该实现建立在编译 IDL 所产生的类型库基础之上。默认的 IDispatch 是以一个双接口(一个 IDispatch 接口，其后跟着一些在 IDL 中定义的函数)为基础的。

正如您所看到的，使用 ATL 来实现 COM 类与使用纯 C++是不同的。ATL 的设计思想不同于您在开发常规 C++类时使用的方法。对于经典的 ATL，项目中最重要的部分是接口，而接口是在 IDL 中被描述的。通过在 IDL 代码中将函数加到接口中，您可以自动把函数加入到实现该接口的具体类中。这些函数之所以会被自动加入到类中，是因为这样的项目在被建立的时候，已经被设置成：编译 IDL 文件以产生一个 C++头文件，该 C++头文件中包含了这些函数。在接口中加入了函数之后，剩下您所要做的事情就是在 C++类中实现这些函数。IDL 文件也提供了一个类型库，因而 COM 类就可以用它来实现 IDispatch。然而，ATL 不仅对于实现轻量级的 COM 服务和对象很有用，同时，ATL 也是一种用来创建 ActiveX 控件的新方法，在下一章中您将会看到这种方法。

25.6.8 基本的 ATL 体系结构

如果您用 ATL 做过一些开发试验，您就会看到它如何简化了实现 COM 类的过程。工具支持非常棒——利用 Visual C++ .NET 来开发 COM 类几乎与创建 MFC 程序一样容易。您只需用 ATL Project Wizard 来创建一个服务器程序，并用 ATL Simple Object Wizard 来创建新的 ATL 类即可。如同 MFC 的情形一样，您可以使用 Class View 来为接口加入新的函数定义。然后，您只要简单地在 Class View 所产生的 C++代码中填入函数即可。ATL Project Wizard 所产生的代码包含了所有为实现您的类而需要的代码，包括 IUnknown 的一个实现、一个用作 COM 类宿主的服务器模块，以及一个实现了

IClassFactory 接口的类对象。

就像刚刚描述的那样来编写 COM 对象当然比其他大多数方法要方便得多。但是，当您使用 ATL Project Wizard 来产生代码时，到底发生了什么呢？如果您想超越 ATL Project Wizard 和 Class View 所提供的支持，希望扩展您的 ATL COM 类和服务器模块，那么，您就需要理解经典的 ATL 是如何工作的，这是非常重要的。例如，ATL 提供了高级的接口技术支持，如 tear-off 接口。不幸的是，Visual C++ .NET 中并不存在针对 tear-off 接口的向导选项。尽管 ATL 支持 tear-off 接口，但您必须手工做一些工作才能实现 tear-off 接口。在这种情况下，理解 ATL 实现 IUnknown 的方法将是非常有用的。

现在我们来更加细致地看一看 CClassicATLSpaceship 类。下面是完整的定义：

```
// CClassicATLSpaceship
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship,
                    &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
        &IID_IClassicATLSpaceship,
        &LIBID_ATLSpaceShipSvrLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
    CClassicATLSpaceship()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_CLASSICATLSPACESHIP)

BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }
    void FinalRelease()
    {
    }
public:
};
```

```
OBJECT_ENTRY_AUTO(__uuidof(ClassicATLSpaceship), CClassicATLSpaceship)
```

虽然这是一段普通的 C++源代码，但它与常规的实现 COM 对象的 C++源代码在几个方面有所不同。例如，许多 COM 类实现都直接从 COM 接口进行派生，但是，这里的 COM 类是从几个模板派生的。另外，该 C++类使用了几个看起来很古怪的宏。如果您查看它们的代码，您将会看到 ATL 的 IUnknown 实现，以及一些令人感兴趣的话题，比如管理 vtable 膨胀的一项技术，以及不同于平常的模板用法。让我们从向导产生的宏代码中的第一个符号 ATL_NO_VTABLE 来开始讨论吧。

25.6.9 管理 Vtable 膨胀

用 C++中的纯虚基类来表达 COM 接口是非常容易的。利用多继承机制来编写 COM 类(还有其他编写 COM 类的方法)仅仅是在基类列表中加入 COM 接口基类，并且实现所有这些成员函数的联合。当然，这意味着，在 COM 服务器的内存中将包含较多的 vtable 开销，因为 COM 类所实现的每一个接口都会带来 vtable 开销。如果您只有少量的接口，并且 C++类层次不是很深，那么这不是一个大问题。然而，用这种方法来实现接口确实增加了开销，而且随着接口数目和层次深度的增加，这种开销也会随之增加。ATL 提供了一种方法来减少由大量虚函数所引入的开销。ATL 定义了下面的符号：

```
#define ATL_NO_VTABLE  __declspec(novtable)
```

使用 ATL_NO_VTABLE 可以防止对象的 vtable 在构造函数中被初始化，因而使链接器消除了 vtable 和该类的 vtable 所指向的全部函数。这样消除了之后，可以在某种程度上减小 COM 服务器的尺寸，只要最终的派生类不使用上面所示的__declspec(novtable)即可。对于派生层次较深的情况，您将会注意到尺寸上的差别。然而，请注意一条告诫：对于任何使用了该 declspec 的类的对象，在构造函数中调用虚函数是不安全的，因为此时 vptr 还没有被初始化。

在前面的类声明中，第二行说明了 CClassicATLSpaceship 是从 CComObjectRootEx 派生的。这正是获得 ATL 的 IUnknown 实现的地方。

25.6.10 ATL 的 IUnknown：CComObjectRootEx

虽然 CComObjectRootEx 并不是在 ATL 层次结构的最顶端，但它离顶端非常接近。在 ATL 中，COM 对象的实际基类是一个名为 CComObjectRootBase 的类。(这两个类的定义都在 AtlCom.h 中)。看一看 CComObjectRootBase 的代码，您将会看到在一个 C++ COM 类中所期望出现的代码。CComObjectRootBase 包含了一个用来实现引用计数的名为 m_dwRef 的 DWORD 成员。您还将看到 OuterQueryInterface、OuterAddRef 和 OuterRe-lease，它们用来支持 COM 聚合和 tear-off 接口。看一看 CComObjectRootEx，您将看到 InternalAddRef、InternalRelease 和 InternalQueryInterface，它们执行常规的引用计数功能，以及支持对象身份同一性的 QueryInterface 机制。

注意，CClassicATLSpaceship 的定义表明该类是从 CComObjectRootEx 派生的，而 CComObjectRootEx 是一个参数化的模板类。下面的代码清单显示了 CComObjectRootEx 的定义：

```
template <class ThreadModel>
class CComObjectRootEx : public CComObjectRootBase
{
public:
    typedef ThreadModel _ThreadModel;
    typedef _ThreadModel::AutoCriticalSection _CritSec;
    typedef CComObjectLockT<_ThreadModel> ObjectLock;

    ULONG InternalAddRef()
    {
        ATLASSERT(m_dwRef != -1L);
        return _ThreadModel::Increment(&m_dwRef);
    }
    ULONG InternalRelease()
    {
#ifdef _DEBUG
        LONG nRef = _ThreadModel::Decrement(&m_dwRef);
        if (nRef < -(LONG_MAX / 2))
        {
            ATLASSERT(0 && _T("Release called on a pointer "
                              "that has already been released"));
        }
        return nRef;
#else
        return _ThreadModel::Decrement(&m_dwRef);
#endif
    }

    void Lock() {m_critsec.Lock();}
    void Unlock() {m_critsec.Unlock();}
private:
    _CritSec m_critsec;
};
```

CComObjectRootEx 是一个模板类，根据模板参数中传入的线程模型类的种类的不同，它的类型也随之不同。实际上，ATL 支持几种线程模型：单线程套间、多线程套间和自由线程。ATL 包含了三个预处理器符号，可作为一个项目的不同默认线程模型：_ATL_SINGLE_THREADED、_ATL_APARTMENT_THREADED 和_ATL_FREE_THREA-DED。

在 Stdafx.h 中定义预处理器符号_ATL_SINGLE_THREADED 将改变默认线程模型，改为只支持一个 STA 线程。这个选项对于那些并不创建任何额外线程的进程外服务器是非常有用的。因为这样

的服务器仅仅支持一个线程，所以 ATL 的全局状态可以不需要用临界区来保护，因此服务器的效率更高。缺点是，该服务器将只支持一个线程。定义预处理器符号_ATL_APARTMENT_THREAEDED 将使默认的线程模型改为支持多个 STA 线程。这对于套间模型的进程内服务器(即注册表的 ThreadingModel=Apartment 值的服务器)是非常有用的。因为使用该线程模型的服务器可能支持多个线程，所以 ATL 利用临界区来保护它的全局状态。最后，定义了_ATL_FREE_THREADED 预处理器符号则可以创建出可兼容于任何线程环境的服务器。也就是说，ATL 使用临界区来保护它的状态，并且服务器中的每一个对象都将有自己的临界区来维护数据的安全性。

这些预处理器符号仅仅决定哪个线程类将作为模板参数插入到 CComObjectRootEx 中。ATL 提供了三个线程模型类。对上面列出的三种环境中的 COM 类，通过这三个类，它们都可以获得有效的线程安全行为的支持。这三个类是 CComMultiThreadModelNoCS、CComMultiThreadModel 和 CComSimpleThreadModel。下面的代码展示了 ATL 中的这三个线程模型类：

```
class CComMultiThreadModelNoCS
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw()
        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p) throw()
        {return InterlockedDecrement(p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
};

class CComMultiThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw()
        {return InterlockedIncrement(p);}
    static ULONG WINAPI Decrement(LPLONG p) throw()
        {return InterlockedDecrement(p);}
    typedef CComAutoCriticalSection AutoCriticalSection;
    typedef CComCriticalSection CriticalSection;
    typedef CComMultiThreadModelNoCS ThreadModelNoCS;
};

class CComSingleThreadModel
{
public:
    static ULONG WINAPI Increment(LPLONG p) throw() {return ++(*p);}
    static ULONG WINAPI Decrement(LPLONG p) throw() {return --(*p);}
    typedef CComFakeCriticalSection AutoCriticalSection;
```

```
    typedef CComFakeCriticalSection CriticalSection;
    typedef CComSingleThreadModel ThreadModelNoCS;
};
```

注意，每一个类都导出了两个静态函数——Increment 和 Decrement，以及针对临界区的各种别名。

CComMultiThreadModel 和 CComMultiThreadModelNoCS 都通过线程安全的 Win32 函数 InterlockedIncrement 和 InterlockedDecrement 来实现 Increment 和 Decrement。CCom-SingleThreadModel 则利用更加常规的++和--操作符来实现。

除了实现递增和递减的方法有所不同以外，这三个线程模型管理临界区的方式也不相同。ATL 封装了两个临界区——CComCriticalSection(这是对 Win32 临界区 API 的简单封装)和 CcomAutoCritical Section (与 CComCriticalSection 相同，但是增加了对临界区的自动初始化和清除操作)。ATL 也定义了一个“假的”临界区类，它具有与其他临界区类相同的原型特征，但不做任何事情。正如从类定义中可以看到的，CComMultiThreadModel 使用真正的临界区，而 CComMultiThreadModelNoCS 和 CComSingleThreadModel 使用假的 no-op(空操作)临界区。

所以，现在最小化的 ATL 类定义就会有更多的意义。每当您定义一个 COM 类的时候，CcomObject RootEx 就会使用一个线程模型类。CClassicATLSpaceship 类是用 CCom-SingleThreadModel 类定义的，所以它为递增和递减使用 CComSingleThreadModel 的方法，并使用假的空操作临界区。因而 CClassicATLSpaceship 可以使用最为有效的行为方式，因为它不需要考虑保护数据。然而，您不能仅仅满足于这一种模型。例如，如果您想使 CClassicATLSpaceship 对于任何线程环境都是安全的，那么，您只需重新定义 CClassic-ATLSpaceship，用 CComMultiThreadModel 作为模板参数从 CComObjectRootEx 派生即可。AddRef 和 Release 调用将自动被映射到正确的 Increment 和 Decrement 函数上。

25.6.11　ATL 和 QueryInterface

看起来好像 ATL 套用了 MFC 中实现 QueryInterface 的方法，因为 ATL 也像 MFC 的 QueryInterface 版本一样使用了一个查询表。看一下 CClassicATLSpaceship 定义的中间部分，您会看到一个被称为接口映射表的宏结构。ATL 的接口映射表构成了它的 QueryInter-face 机制。

客户利用 QueryInterface 可以任意地增加它与对象之间的连接。也就是说，当客户需要一个新的接口的时候，它将会通过一个已有的接口来调用 QueryInterface。然后，对象将会检查被请求接口的名字，并且将它与该对象所实现的所有接口进行比较。如果对象实现了这个接口，那么它将该接口传回给客户。否则的话，QueryInterface 将返回一个错误值，表明没有找到接口。

传统的 QueryInterface 实现通常是由一组长长的 if-then 语句组成的。例如，对于多继承方式的 COM 类，一个标准的 QueryInterface 实现可能看起来像这样：

```
class CClassicATLSpaceship: public IDispatch,
                                IClassicATLSpaceship {
   HRESULT QueryInterface(RIID riid,
                          void** ppv) {
      if(riid == IID_IDispatch)
         *ppv = (IDispatch*) this;
      else if(riid == IID_IClassicATLSpaceship ||
             riid == IID_IUnknown)
         *ppv = (IClassicATLSpaceship *) this;
      else {
         *ppv = 0;
         return E_NOINTERFACE;
      }

      ((IUnknown*)(*ppv))->AddRef();
      return NOERROR;
   }
   // AddRef, Release, and other functions
};
```

很快您将会看到，ATL 利用一个查询表取代了这种传统的 if-then 语句。

ATL 的查询表从一个名为 BEGIN_COM_MAP 的宏开始。下面的代码清单给出了 BEGIN_COM_MAP 的完整定义：

```
#define BEGIN_COM_MAP(x) public: \
   typedef x _ComMapClass; \
   static HRESULT WINAPI _Cache(void* pv, \
      REFIID iid, void** ppvObject, \
      DWORD_PTR dw) throw() \
   { \
      _ComMapClass* p = (_ComMapClass*)pv; \
      p->Lock(); \
      HRESULT hRes = \
         ATL::CComObjectRootBase::_Cache(pv, iid, ppvObject, dw); \
      p->Unlock(); \
      return hRes; \
   } \
   IUnknown* _GetRawUnknown() throw() \
   { ATLASSERT(_GetEntries()[0].pFunc == _ATL_SIMPLEMAPENTRY); \
      return (IUnknown*)((INT_PTR)this+_GetEntries()->dw); } \
   _ATL_DECLARE_GET_UNKNOWN(x) \
   HRESULT _InternalQueryInterface(REFIID iid, \
      void** ppvObject) throw() \
   { return InternalQueryInterface(this, \
      _GetEntries(), iid, ppvObject); } \
```

```
    const static ATL::_ATL_INTMAP_ENTRY* WINAPI _GetEntries() \
        throw() { \
    static const ATL::_ATL_INTMAP_ENTRY _entries[] = \
        { DEBUG_QI_ENTRY(x)
```

每一个使用 ATL 来实现 IUnknown 的类都指定了一个接口映射表，以提供给 InternalQueryInterface。ATL 的接口映射表由一个数据结构组成，该结构中包含了“接口 ID(GUID)/DWORD/函数指针”三元组。下面的代码显示了名为_ATL_INTMAP_ENTRY 的类型，其中包含了该三元组。

```
struct _ATL_INTMAP_ENTRY
{
    const IID* piid;       // the interface id (IID)
    DWORD_PTR dw;
    _ATL_CREATORARGFUNC* pFunc; //NULL:end, 1:offset, n:ptr
};
```

第一个成员是接口 ID(一个 GUID)，第二个成员表明了当该接口被请求时应当采取什么样的动作。有三种方法可用来解释第三个成员。如果 pFunc 等于常量_ATL_SIMPLE-MAPENTRY(值为 1)，则 dw 是对象内部的一个偏移量；如果 pFunc 是非空的，但不等于 1，则 pFunc 表示一个函数，当该接口被请求时，该函数将被调用；如果 pFunc 是 NULL，则 dw 表明了 QueryInterface 查询表的尾部。

注意，CClassicATLSpaceship 使用了 COM_INTERFACE_ENTRY。这是对于常规接口的映射表项。下面是该宏的定义：

```
#define offsetofclass(base, derived) \
    ((DWORD_PTR) \
    (static_cast<base*>((derived*)_ATL_PACKING))-_ATL_PACKING)

#define COM_INTERFACE_ENTRY(x) \
    {&_ATL_IIDOF(x), \
    offsetofclass(x, _ComMapClass), \
    _ATL_SIMPLEMAPENTRY}
```

COM_INTERFACE_ENTRY 用接口的 GUID 来填充_ATL_INTMAP_ENTRY 结构。另外，请注意 offsetofclass 如何将 this 指针转换到正确的接口上，并用这个值来填充 dw 成员。最后，COM_INTERFACE_ENTRY 用_ATL_SIMPLEMAPENTRY 来填充最后的域，以表明这里的 dw 指向类内部的一个偏移量。

例如，当预处理器完成预处理工作之后，CClassicATLSpaceship 的接口映射表看起来会像这样：

```
const static _ATL_INTMAP_ENTRY* __stdcall _GetEntries() {
    static const _ATL_INTMAP_ENTRY _entries[] = {
        {&IID_IClassicATLSpaceship,
         ((DWORD)(static_cast<IClassicATLSpaceship*>
            ((_ComMapClass*)8))-8),
```

```
        ((_ATL_CREATORARGFUNC*)1)},
        {&IID_IDispatch,
         ((DWORD)(static_cast<IDispatch*>((_ComMapClass*)8))-8),
         ((_ATL_CREATORARGFUNC*)1)},
         {0, 0, 0}
    };
    return _entries;
}
```

现在，CClassicATLSpaceship 类支持两个接口——IClassicATLSpaceship 和 IDispatch，所以，在映射表中仅有两项。

CComObjecRootEx 的 InternalQueryInterface 实现使用_GetEntries 函数作为第二个参数。CComObjecRootEx::InternalQueryInterface 使用了一个名为 AtlInternalQueryInterface 的全局 ATL 函数，用它在映射表中查询接口。AtlInternalQueryInterface 只是简单地遍历这张表，以尽可能找到客户所请求的接口。

除了 COM_INTERFACE_ENTRY 之外，ATL 还包含了 16 个其他的宏，用于实现从 tear-off 接口到 COM 聚合等的各种复合技术。现在，您将会看到如何来增强 IClassicATL-Spaceship 接口，并增加另外两个接口：IMotion 和 IVisual。您还将学习一项名为双接口的重要的 COM 技术。

25.6.12 让飞船动起来

现在，您已经获得了一些 ATL 代码，您能用它们来做什么呢？因为这是 COM，所以，最开始的工作地点应该是在 IDL 文件中。再次说明，如果您是一个有经验的 C++开发人员，那么这是软件开发过程中的一个新阶段，您以前可能没有经历过。请记住，在今天，软件的分布和集成正变得越来越重要。您可能曾经成功地编写了许多 C++类，并将它们都集成到一个项目中，这样做之所以能成功是因为您(作为一个开发人员)了解整个项目的编码情况。但是，组件技术(比如 COM)改变了这种做法。开发人员不再需要了解整个项目的代码结构。您通常只有一个组件——您并没有该组件的源代码。您唯一知道该如何与一个组件进行通话的方法是通过它所暴露的接口。

请记住,现代软件开发人员需要使用许多不同的工具,并不仅仅是 C++。您可能接触过 Visual Basic 开发人员、Delphi 开发人员和 C 开发人员。COM 的职责在于将这些边界缝合起来，从而使得当这些由各种组件创建的软件片断结合到一起的时候，它们可以平滑地集成起来并协同工作。另外，远程的分布式软件(可能是同一台机器上的进程外软件，甚至也可能是不同机器上的软件)需要某种跨进程的通信标准。这正是 IDL 之所以存在的原因。下面是由 ATL 向导创建的默认 IDL 文件，其中定义了新的飞船类：

```
import "oaidl.idl";
import "ocidl.idl";
```

```
[
    object,
    uuid(45896187-46FF-4A07-A9DC-557377380535),
    dual,
    nonextensible,
    helpstring("IClassicATLSpaceship Interface"),
    pointer_default(unique)
]
interface IClassicATLSpaceship : IDispatch{
};
[
    uuid(F5FD4043-22AE-470D-8C43-1AC904D2E8E0),
    version(1.0),
    helpstring("ATLSpaceShipSvr 1.0 Type Library")
]
library ATLSpaceShipSvrLib
{
    importlib("stdole2.tlb");
    [
        uuid(E485E21E-A23C-413F-A93B-909318565113),
        helpstring("ClassicATLSpaceship Class")
    ]
    coclass ClassicATLSpaceship
    {
        [default] interface IClassicATLSpaceship;
    };
};
```

这里涉及到的关键概念是，IDL 是一门纯声明性的语言。IDL 语言定义了其他客户将如何与对象进行对话。记住，您最终将通过 MIDL 编译器来运行这段代码，以获得一个纯抽象基类(对于 C++客户是非常有用的)和一个类型库(对于 Visual Basic 和 Java 客户以及其他客户是非常有用的)。如果您能够理解普通的 C 代码，那么您将会很容易地理解 IDL。您可以把 IDL 看作是带有脚注的 C。IDL 的语法规定了属性(attribute)总是出现在语言实体的说明之前。例如，属性出现在接口声明、库声明和方法参数之前。

如果您看一下 IDL 文件，您就会注意到在开始处引入了 Oaidl.idl 和 Ocidl.idl。之所以引入这些文件，在某种程度上这与您在 C 或者 C++文件中包含 Windows.h 是一样的。这些 IDL 文件包含了所有基本的 COM 基础设施的定义(其中包括 IUnknown 和 IDispatch 的定义)。

在 import 语句后面，跟着一个左方括号“ [”。在 IDL 中，方括号总是将属性括起来。在这个 IDL 文件中，第一个描述的元素是 IClassicATLSpaceship 接口。然而，在描述该接口之前，您必须对它赋予某些属性。例如，它需要一个名字(一个 GUID)，而且您需要告诉 MIDL 编译器，该接口是面向 COM

的，而不是被用于标准的 RPC(remote procedure call，远过程调用)，而且这是一个双接口。(稍后将进一步介绍双接口。)接下来是实际的接口内容。注意，它看起来非常像一个普通的 C 结构。

一旦在 IDL 中说明了一个接口，则将这些信息集中到一个类型库中也是很有用的，这是由 IDL 文件的下一部分来完成的。注意，类型库部分也是以一个方括号作为开始的，这表明它的属性紧随其后。如同通常情形一样，在 COM 中类型库也是一个独立的“ 事物 ”，所以它需要一个名字(GUID)。library 语句告诉 MIDL 编译器，这个库包含了一个名为 ClassicATLSpaceship 的 COM 类，并且该类的客户可以获得 IClassicATLSpaceship 接口。

25.6.13 在接口中增加方法

现在，IClassicATLSpaceship 接口还是空荡荡的。看起来它可能会用到一两个方法。让我们来增加一个方法。当我们为 MFC COM 类加上了自动化属性时，我们就可以使用 Class View 了。对于 ATL，我们同样可以这样做。同时也请注意，CClassicATLSpaceship 是从 IClassicATLSpaceship 派生出来的。当然，这里的 IClassicATLSpaceship 是一个 COM 接口。在 Class View 中的 IClassicATLSpaceship 上双击，将在编辑器窗口中打开 IDL 文件，并直接定位到其中特定的部分。

这时，您可以开始在 IDL 文件中输入该 COM 接口。如果您用这种方法(直接添加到 IDL 文件中)来添加函数和方法，那么您必须要访问 ClassicATLSpaceship.h 和 ClassicATL-Spaceship.cpp，并且手工插入这些方法。要想在接口中加入方法，一种更有效的做法是通过 Class View 使用 Add Method Wizard(如图 25.4 所示)。为了利用 Class View 来编辑 IDL 文件，只需在 Class View 中的接口名上面单击右键。您将会在快捷菜单中看到 Add Method 和 Add Property 命令。让我们来加入一个名为 CallStarFleet 的方法。

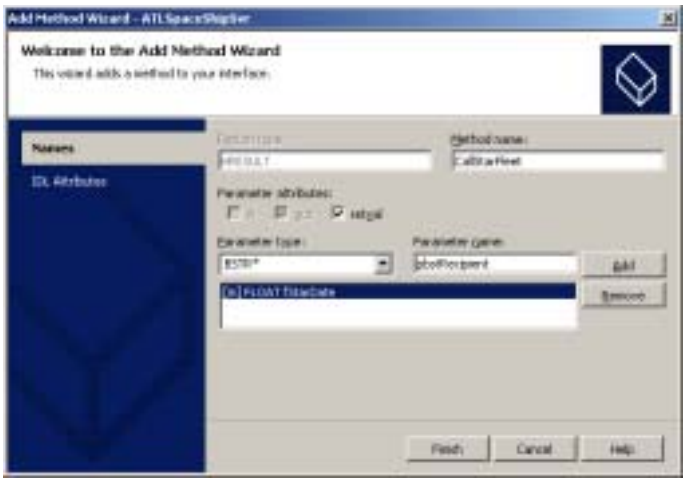

图 25.4 在一个接口中加入一个方法

为了加入一个方法，您只需在 Method Name 文本框内输入方法的名字。然后在 Parameter Name 和 Parameter Type 文本框中输入方法的参数。这里将有助于您更好地理解 IDL。

记住，IDL 的目的是提供关于“方法如何被调用”的完整而又明确的信息。在标准 C++中，您可能经常能躲过诸如未定义数组大小这样的歧义，因为调用者和被调用者共享相同的堆栈框架——栈中总是有许多可用的空间。现在，这些方法调用可能要通过底层的线路连接，所以，很重要的一点是，您要确切地告诉远程层，当它遇到一个 COM 接口时它应该会得到什么样的数据。这是通过在方法参数前面加上一些属性来实现的(多个方括号)。

图 25.4 所示的方法调用在它的列表中有两个参数——一个浮点数(fStarDate)指明了恒星时间，一个 BSTR 参数指明了谁将会收到通讯信息。注意，该方法定义清楚地说明了参数的方向。fStartDate 被传递进方法调用，这由[in]属性来指定的。标识接收方的 BSTR 被作为一个指针传递回来。[out]属性表明了参数的方向是从对象返回到客户。[retval]属性表明了您可以在某种支持这种特性的高级语言中将该方法的结果赋值给一个变量。

25.6.14　双接口

在第 23 章中，您已经看到过 IDispatch 接口了。IDispatch 使得向诸如 JScript 这样的环境(在二进制层次上)暴露功能成为可能，在这样的环境中，程序员是不可能直接产生 vtable 结构的。为了使 IDispatch 可以工作，客户在能够调用 Invoke 之前，必须要先通过一些机关。客户首先要获得该调用的 DISPID。然后客户必须设置 VARIANT 参数。在对象一方，对象必须解析出所有这些 VARIANT 参数，并确定它们是正确的，再将它们放入某种栈帧中，然后执行真正的函数调用。您可以想像，所有这些工作是复杂的，而且也非常耗时。

如果您正在编写一个 COM 对象，并且您又期望某些客户会使用脚本语言，而其他的客户使用像 C++这样的语言，那么，此时您将不得不做出选择。您要么包含 IDispatch 接口，要么将所有使用脚本语言的客户排除在外。如果您只是提供了 IDispatch，那么这将使得从 C++中访问您的对象变得非常不方便。当然，您可以提供两种访问途径，即或者通过 IDispatch，或者通过一个自定义接口，但是，那样会涉及到大量的事务工作。双接口的诞生正是为了处理这样的问题。

双接口非常简单，它是在 IDispatch 末尾再粘贴上一些函数。例如，下面描述的 IMotion 接口是一个有效的双接口：

```
interface IMotion : public IDispatch {
    virtual HRESULT Fly() = 0;
    virtual HRESULT GetPosition() = 0;
};
```

因为 IMotion 是从 IDispatch 派生的，所以，IMotion 的前 7 个函数是 IDispatch 的那些函数。如

果客户只能理解 IDispatch(例如 JScirpt)，那么它可以把双接口看成 IDispatch 的另一个版本，并且将 DISPID 提供给 Invoke 函数，以期望调用后面的某一个函数。如果客户能够理解 vtable 风格的自定义接口，那么它可以看到整个接口，并忽略掉中间的 4 个函数(即 IDispatch 的后 4 个函数)，将注意力集中在前 3 个函数(IUnknown)和最后 22个函数(这两个函数代表了该接口的核心功能)上。图 25.5 显示了 IMotion 的 vtable 布局情况。

大多数原始的 C++实现都会载入类型库，并委托 ITypeInfo 来执行 Invoke 和 GetIDs-OfNames 的繁杂任务。要想了解期间的工作过程，请参阅 Kraig Brockschmidt 所著的 *Inside OLE, 2nd Edition*(Microsoft Press，1995)或者 Dale Rogerson 所著的 *Inside COM* (Microsoft Press，1997)。

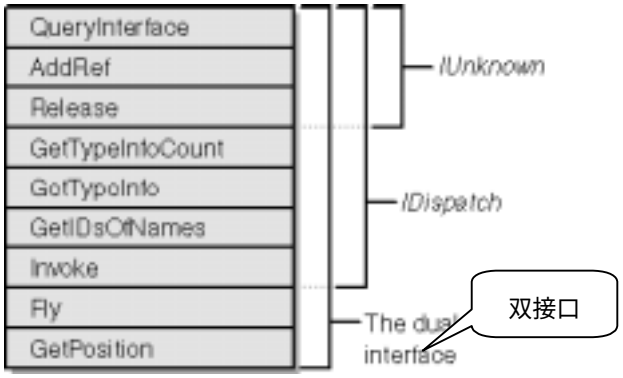

图 25.5　双接口的布局结构

25.6.15　ATL 和 IDispatch

ATL 的 IDispatch 实现被委托给了类型库。ATL 的 IDispatch 实现位于 IDispatchImpl 类中。对于希望实现双接口的对象，需要在基类列表中包含 IDispatchImpl，如下所示：

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship, &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship, &IID_IClassicATLSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IVisual, &IID_IVisual,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>
{
```

2 译者注：原文此处为 3，疑有误。

```
……
};
```

除了在基类列表中包含 IDispatchImpl 模板类之外，在接口映射表中还要包含针对双接口的表项，以及针对 IDispatch 的表项，这样 QueryInterface 才能正常工作：

```
BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

正如您所看到的，IDispatchImpl 模板类实参包含了双接口自身、接口的 GUID，以及代表类型库的 GUID(该类型库中包含了有关该接口的所有信息)。除了这些模板实参以外，IDispatchImpl 类还有一些可选参数，上面的代码中没有给出这些参数。模板参数列表中也包含了类型库的主要和次要版本信息。最后一个模板参数是一个用于管理类型信息的类。ATL 提供了一个名为 CComTypeInfoHolder 的默认类。

在大多数原始的 C++ IDispatch 实现中，通常在类的构造函数中调用 LoadTypeLib 和 ITypeLib∷GetTypeInfoOfGuid，并在该类的生存期内保留此 ITypeInfo 指针。ATL 的实现与这种做法略有不同，它通过使用 CComTypeInfoHolder 类来帮助管理 ITypeInfo 指针。CComTypeInfoHolder 在一个数据成员中维护了一个 ITypeInfo 指针，并包装了两个关键的、与 IDispatch 相关的函数：GetIDsOfNames 和 Invoke。

客户用 IID_IClassicATLSpaceship 来调用 QueryInterface，就可以获得双接口。(客户也可以通过调用 QueryInterface 来请求 IDispatch 接口，从而得到这个接口)。如果客户调用该接口上的 CallStarFleet，则客户将直接访问这些函数(就像对其他 COM 接口一样)。

当客户调用 IDispatch∷Invoke 时，该调用将进入到 IDispatchImpl 的 Invoke 函数内部，就如您所期望的那样。在那里，IDispatchImpl∷Invoke 委托 CComTypeInfoHolder 类来执行该调用，即委托给 CComTypeInfoHolder 的 Invoke 函数。CComTypeInfoHolder 类并不立即调用 LoadTypeLib，而是直到真正调用 Invoke 或者 GetIDsOfNames 的时候才调用 LoadTypeLib。CComTypeInfoHolder 有一个名为 GetTI 的成员函数，它(利用作为模板参数传递进来的 GUID 和任何的主要/次要版本号)向注册表询问类型信息。然后，CCom-TypeInfoHolder 调用 ITypeLib∷GetTypeInfo 来获得关于该接口的信息。到了这个时候，CComTypeInfoHolder 再委托给该类型信息指针。IDispatchImpl 以同样的方式来实现 IDispatch∷GetIDsOfNames。

25.6.16　IMotion 和 IVisual 接口

为了使该 COM 类与其他版本(第 22 章中介绍的原始 C++版本和 MFC 版本)相同，您必须将 IMotion 和 IVisual 接口加入到该项目中，进而再加入到该类中。不幸的是，Visual Studio .NET 并没有提供一

个向导来将一个接口加入到项目中。为了做到这一点，您可以使用 ATL Simple Object Wizard 来加入一个简单对象。或者您也可以手工输入这些接口。请打开 IDL 文件，并将光标定位在靠近顶部的地方(在#import 语句之后，library 语句之前的某个位置上)，然后，按照下面段落所描述的那样输入接口定义。

一旦您熟悉了 IDL 的用法之后，当您描述一个接口时，您的第一直觉应该是插入一个方括号。记住，IDL 中独立的项是有属性的。对于一个接口，其中最重要的一个属性是名字，或者 GUID。另外，接口至少应该具有 object 属性，以便告诉 MIDL 编译器此时正在处理 COM 接口(而不是常规的 RPC 接口)。您也可以指定这些接口为双接口。接口属性中的关键字“ dual ”表明了这是双接口，并且会插入特定的注册表项，以便使通用列集机制(universal marshaling)起作用。当用右方括号结束了属性描述之后，interface 关键字将紧接着开始描述该接口。

我们将使 IMotion 成为一个双接口，使 IVisual 成为一个普通的自定义接口，以说明这两种不同类型的接口是如何被连接到 CSpaceship 类上的。下面是 IDL 中描述的 IMotion 和 IVisual 接口：

```
[
    object,
    uuid(692D03A4-C689-11CE-B337-88EA36DE9E4E),
    dual,
    helpstring("IMotion interface")
]
interface IMotion : IDispatch
{
    HRESULT Fly();
    HRESULT GetPosition([out,retval]long* nPosition);
};

[
    object,
    uuid(692D03A5-C689-11CE-B337-88EA36DE9E4E),
    helpstring("IVisual interface")
]
interface IVisual : IUnknown
{
    HRESULT Display();
};
```

一旦这两个接口在 IDL 中被描述了之后，您可以再次通过 MIDL 编译器运行 IDL。MIDL 编译器将产生一个新的 Spaceshipsvr.h，其中包含了 IMotion 和 IVisual 的纯抽象基类。

现在，您需要在 CSpaceship 类中加入这些接口。这里分两步。第一步是创建该 COM 类实体的接口部分。让我们首先从 IMotion 接口开始。在 CSpaceship 中加入 IMotion 接口很容易的。您只需利用 IDispatchImpl 模板来提供一个双接口的实现即可，如下所示：

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship,
                       &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
                         &IID_IClassicATLSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>
{
……
};
```

第二步涉及到扩充接口映射表，以便使客户能够获得 IMotion 接口。然而，在单个 COM 类中实现两个双接口将会引出一个有趣的问题。当一个客户通过调用 QueryInterface 来请求 IMotion 时，客户会很确定地获得 IMotion 接口。然而，当客户调用 QueryInterface 以请求 IDispatch 接口时，客户将获得哪一个 IDispatch 版本呢？是 IClassicATLSpaceship 的分发接口，还是 IMotion 的分发接口？

25.6.17　多个双接口

记住，所有的双接口都是以 IDispatch 的 7 个函数作为开始的。当客户用 IID_IDispatch 来调用 QueryInterface 时，问题就出现了。作为 COM 类的开发人员，您需要选择将哪一个 IDispatch 版本传递出去。

接口映射表正是为 QueryInterface 指定 IID_IDispatch 的地方。ATL 有一个特殊的宏来处理这种双接口的情形。首先，请考虑目前我们所得到的 CClassicATLSpaceship 接口映射表：

```
BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

当客户调用 QueryInterface 时，ATL 搜索该表，试图在表中找到与请求的 IID 相匹配的表项。上面的接口映射表支持两个接口：IClassicATLSpaceship 和 IDispatch。如果您希望在 CClassicATLSpaceship 类中加入另一个双接口，那么您需要另一个不同的宏。

宏 COM_INTERFACE_ENTRY2 能够在基于 ATL 的 COM 类中处理多个分发接口。为了使 QueryInterface 能够正确地工作，您所要做的只是确定一下：当客户请求 IDispatch 时，它应该获得哪一个 IDispatch 版本，如下所示：

```
BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
```

```
    COM_INTERFACE_ENTRY2(IDispatch, IClassicATLSpaceship)
END_COM_MAP()
```

在这种情况下，若客户请求 IDispatch，那么它将获得一个指向 IClassicATLSpaceship 的指针(它的前 7 个函数包含了 IDispatch 的函数)。

在基于 ATL 的 COM 类上添加非双接口更加简单。您只需在基类列表中加入该接口即可，如下所示：

```
class ATL_NO_VTABLE CClassicATLSpaceship :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CClassicATLSpaceship,
                       &CLSID_ClassicATLSpaceship>,
    public IDispatchImpl<IClassicATLSpaceship,
                         &IID_IClassicATLSpaceship,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl<IMotion, &IID_IMotion,
                         &LIBID_SPACESHIPSVRLib>,
    public IDispatchImpl(IVisual, &IID_IVisual,
                         &LIBID_SPACESHIPSVRLib>
{
 ......
};
```

然后，在接口映射表中加入一个表项，如下所示：

```
BEGIN_COM_MAP(CClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IMotion)
    COM_INTERFACE_ENTRY2(IDispatch, IClassicATLSpaceship)
    COM_INTERFACE_ENTRY(IVisual)
END_COM_MAP()
```

现在，您有了一个可以工作的 COM 服务器，它能够将自己注册到注册表中，能够以 COM 组件的形式参与到组件软件中。但事实上，利用 Visual C++ .NET 还有另外一种方法可以实现 COM 服务器。这就是属性化程序设计(attributed programming)。

25.7 属性化程序设计

您可以不用通过 C++模板以编程方式来加入 COM 支持，而是采用一种更具声明性的方法：属性化程序设计。经典的 ATL 程序设计涉及到通过模板类和接口映射表宏来获得 IUnknown 支持，与此不同的是，属性化程序设计非常简单，只需直接在源代码中声明一个 COM 类即可。

现在让我们用属性化程序设计的方法来创建一个同样的飞船服务器。首先，创建一个新的 ATL

项目，但这一次请在 ATL Project Wizard 的 Application Settings 页面上选中 Attributed 复选框。然后用 ATL Simple Object Wizard 加入一个属性化的类。将该类命名为 CAttributedATLSpaceship。当您设置该类的选项时，您将会注意到，其中的选项都是相同的。也就是说，您的类可以是 Apartment Threaded、Free Threaded、Both 或 Neutral。您也可以选择 ISupportErrorInfo 支持，或者允许连接点支持。

然后，当您查看由向导生成的代码时，您将会发现，这次生成的代码与经典的 ATL 代码有显著的不同。下面是您所得到的代码：

```
// IAttributedATLSpaceShip
[
    object,
    uuid("4B8685BD-00F1-4D38-AFC1-3012C786480D"),
    dual,    helpstring("IAttributedATLSpaceShip Interface"),
    pointer_default(unique)
]
__interface IAttributedATLSpaceShip : IDispatch
{
};
// CAttributedATLSpaceShip
[
    coclass,
    threading("apartment"),
    vi_progid("AttributedATLSpaceShipSvr.AttributedATL"),
    progid("AttributedATLSpaceShipSvr.AttributedA.1"),
    version(1.0),
    uuid("CE07EBA4-0858-4A81-AD1C-C12710B4A1A2"),
    helpstring("AttributedATLSpaceShip Class")
]
class ATL_NO_VTABLE CAttributedATLSpaceShip :
    public IAttributedATLSpaceShip
{
public:
    CAttributedATLSpaceShip()
    {
    }
    DECLARE_PROTECT_FINAL_CONSTRUCT()
    HRESULT FinalConstruct()
    {
        return S_OK;
    }
    void FinalRelease()
    {
    }
public:
};
```

在经典 ATL 中以模板形式被引入的所有 COM 支持，现在都通过提供者 DLL(provider DLL)被引入到 ATL 服务器中。在文件顶部用方括号括起来的属性告诉编译器：在 CAttri-butedATLSpaceship 类中加入 COM 基础设施。这比记住像 CComObjectRootEx 和 CComCo-Class 这样的类，以及 BEGIN_COM_MAP 这样的宏要容易得多。

要想进一步开发 COM 类也会容易得多。例如，假设您希望在该类中加入 IMotion 和 IVisible 接口。在属性化的 ATL 中，您只需直接将接口放到 ATL 源代码中即可，如下所示：

```
[
    object,
    uuid("692D03A4-C689-11CE-B337-88EA36DE9E4E"),
    dual,
    helpstring("IMotion interface")
]
__interface IMotion : IDispatch
{
    HRESULT Fly();
    HRESULT GetPosition([out,retval]long* nPosition);
};
[
    object,
    uuid("692D03A5-C689-11CE-B337-88EA36DE9E4E"),
    helpstring("IVisual interface")
]
__interface IVisual : IUnknown
{
    HRESULT Display();
};
// More code
```

在__interface 关键字前面的属性描述了该接口是一个 COM 接口——确切地说，是一个双接口(对于 IMotion 接口而言)。一旦这些接口已经在源代码中被描述过了，如果您想在一个类中实现这些接口，那么，您只需在 Class View 中右键单击类名，选择 Add，然后选择 Implement Interface。您可以从已被注册的类型库中，或者从源代码内列出的接口(IMotion 和 IVisible)中选择适当的接口。Visual Studio .NET 会为您生成相应的函数，您只需填充这些函数即可。

这样得到的 DLL 是一个功能齐备的 COM DLL，它有所有预期的入口点：DLLMain、DllGetClassObject、DllCanUnloadnow、DllRegisterServer 和 DllUnregisterServer。

文件名：	ch25.doc
目录：	C:\WINDOWS\Desktop\desk
模板：	C:\WINDOWS\Application Data\Microsoft\Templates\技术内幕.dot
题目：	第 25 章 活动模板库介绍
主题：	
作者：	liyj
关键词：	
备注：	
创建日期：	2004-6-1 14:55
更改编号：	14
上次保存日期：	2004-6-2 11:41
上次保存者：	zyp
总共编辑时间：	24 分钟
上次打印时间：	2004-6-2 12:13
打印最终结果	
页数：	51
字数：	8,595 （约）
字符数：	48,994 （约）