3 ATL 中的对象

ATL 对 COM 的基本支持可以分为两个部分：对象和服务器。本章涉及 IUnknown 的实现类，并且重点在于线程和各种与 COM 实体身份有关的方面，例如，独立的还是被聚合的对象。下一章的重点是如何从 COM 服务器中暴露类。

3.1 回顾 COM 套间

COM 支持两种对象：多线程对象和单线程对象。多线程对象是那些能够自己完成同步工作的对象，而单线程对象则让 COM 来处理同步问题。虽然多线程对象可能有更高的并发程度，但是单线程对象更容易实现。

COM 判断一个对象是单线程的还是多线程的取决于其生存的套间。COM 套间是一个或者多个线程的组合，这些线程或者被标记成单线程，或者被标记成多线程。调用 CoInitializeEx 可以标记一个线程：

```
HRESULT CoInitializeEx(void* pvReserved, DWORD dwCoInit);
```

根据在线程中创建的对象是单线程的还是多线程的，dwCoInit 参数分别为 COINIT_APARTMENTTHREADED 或者 COINIT_MULTITHREADED。调用 CoInitialize 等同于用 COINIT_APARTMENTTHREADED 参数调用 CoInitializeEx。通过调用 CoInitializeEx，我们称一个线程*加入*了一个套间。

在单线程套间（STA ，single-threaded apartment）的情况下，COM 创建一个不可见的窗口把对当前套间中对象的请求排队。为了服务这些请求，线程必须使用一个 Windows 消息循环。例如：

```
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int) {
  CoInitialzeEx(0, COINIT_APARTMENTTHREADED); // or CoInitialize(0);
```

```
    ...
    MSG msg;
    while( GetMessage(&msg, 0, 0, 0) ) DispatchMessage(&msg);
    CoUninitialize();
}
```

由于每次只能服务一个请求，所以，对于在单线程套间中创建的对象的调用必须完成之后，才能服务下一个请求。这就是 COM 对单线程对象提供的同步机制。

另一方面，在多线程套间（MTA，multithreaded apartment）中创建的对象必须随时准备接收来自任意数目线程的调用。多线程对象负责同步自己的状态。

3.1.1 进程内服务器需要考虑的问题

虽然客户程序和可执行的服务器程序各自管理自己所有的线程，并且因此负责创建它们自己的套间，但是进程内服务器可没有这么奢华。当进程内服务器第一次被装载时，它将处于一个“已经加入了某个套间”的线程环境中。任何调用 CoGetClassObject 或者 CoCreateInstance 的线程必须已经加入了一个套间，否则调用会立即失败。[1] 那么，这是否意味着所有进程内服务器暴露的对象必须与创建它们的线程同是单线程套间或者多线程套间，从而才能使任意的客户不会发生问题呢？幸运的是，不是这样的。

在访问一个类对象(class object，有的书籍称之为类厂 —— 译注)的进程中，当 COM 注意到调用套间的线程模型与正在激活的类不兼容的时候，它将会从另一个兼容的套间环境中调用类对象，必要时甚至创建新的套间。在保持两个套间的同步和并发要求的同时，COM 会根据需要装载代理/存根，比如当一个接口指针在两个套间之间传递的时候。

为了通知 COM 提供适当的保护，我们在注册表中使用 InProcServer32 键（key）下的 ThreadingModel 名字值（named value）来标记一个类。通过使用.reg 文件格式，一个套间线程模型的进程内类会这样标记自己：

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{44EBF751-...}\InProcServer32]
@=bworsvr.dll
ThreadingModel=Apartment
```

ThreadingModel 名字值有以下可能的取值：

- (none)：如果在注册表中 ThreadingModel 没有出现，那么 COM 会假设它是一个

[1] COM 要求在任何“感兴趣”的 COM 调用之前，必须先调用 CoInitializeEx。

遗留下来的单线程（single-threaded）类，并且从进程的第一个 STA（必要时创建一个新的 STA）中访问类对象。这样无需同步对全局、静态或者实例数据的访问就能保护对象。

- Apartment：无需同步对实例数据的访问，就能保护对象；但是这个类的对象可以在不同的 STA 中被创建，所以静态和全局数据仍然需要同步。
- Free：这个值表示这个类的对象只能存在于 MTA 中。在 MTA 中的对象必须同步对实例的访问，全局和静态数据也是如此。
- Both：希望共享客户套间的对象可以把它们的类标记为 Both，这意味着没有不兼容的线程模型。当我们想避免初始的客户程序和对象之间的代理/存根对开销的时候，可以使用这个值。

3.1.2 实现 IUnknown

COM 对象有一个责任：实现 IUnknown 的方法。这些方法完成两项服务 —— 生命周期管理和运行时刻类型发现，如下：

```
interface IUnknown {
 // runtime type discovery
 HRESULT QueryInterface( [i n] REFIID riid,
                        [out, iid_is(riid)] void **ppv);

 // lifetime management
 ULONG AddRef();
 ULONG Release();
}
```

COM 允许每个对象（在某些限制下，详见第 5 章）选择自己的实现方法。典型的实现如下：

```
// Server lifetime management
extern void ServerLock();
extern void ServerUnlock();
class CPenguin : public IBird, public ISnappyDresser {
public:
 CPenguin() : m_cRef(O) { ServerLock(); }
 virtual ~CPenguin()    { ServerUnlockO; }
```

```
// IUnknown methods
STDMETHODIMP Querylnterface(REFIID riid, void **ppv) {
    if( riid == IID_IBird II riid == IID_IUnknown )
       *ppv = static_cast<IBi rd*>(this);
    else if( riid == IID_ISnappyDresser )
       *ppv = static_cast<ISnappyDresser*>(this);
    el se *ppv = O;

    if( *ppv ) {
       rei nterpret_cast<IUnknown*>(*ppv) ->AddRef();
       return S_OK;
   }
    return E_NOINTERFACE;
}

  STDMETHODIMP_(ULONG) AddRef()
  { return Interlockedlncrement(reinterpret_cast<LONG*>(&m_cRef)); }

  STDMETHODIMP_(ULONG) Release()
    ULONG 1 = InterlockedDecrement(reinterpret_cast<LONG*>(&m_cRef));
    if( 1 == 0 ) delete this;
    return I;
}

  // IBird and ISnappyDresser methods...
private:
  ULONG m_cRef;
};
```

这个 IUnknown **实现基于以下几个假设：**

- **对象位于堆**(heap)**上，因为它使用** delete **操作符来删除自己。进一步，对象的生命完全受“未完结的引用**(outstanding references)**”支配。当没有引用时，它清除自己。**
- **这个对象能够在多线程套间中运行，因为它使用线程安全的方式来操纵引用计数。当然为了使对象是完全线程安全的，其他方法也必须以线程安全的方式来实**

现。

- 对象是独立的，并且不能被聚合，因为它没有缓存指向外部控制对象(controlling outer)的引用，也不能把 IUnknown 的方法传递给外部控制对象。
- 对象使用多重继承来暴露接口。
- 对象的存在将使服务器一直运行。构造函数和析构函数分别用来锁定和解锁服务器。

虽然这是一些通用的假设，但是它们并不是唯一的可能。常见的变化包括下面几种：

- 一个对象可以是全局的，并且和服务器的生命一样长，这样的对象不需要引用计数，因为它们永远不删除自己。
- 对象可以不是线程安全的，因为它可能只生存于单线程套间中。
- 一个对象可以选择既支持聚合又支持独立使用，或者只支持聚合。
- 除了多重继承，对象还可以使用其他的技术来暴露接口，包括嵌套复合、tear-off 和聚合。
- 我们可能不想由于一个对象的存在而强迫服务器一直运行。对于全局对象经常是这样，因为它们的存在会禁止服务器卸载。

改变任何这些假设都将导致一个不同的 IUnknown 实现，虽然对象的其他部分好像并没有很大变化（线程安全当然是个例外）。因为这些 IUnknown 的实现细节（虽然很重要）往往会使用非常规则的形式，所以它们可以被封装到 C++的类中。说老实话，我们非常愿意使用别人经过测试的代码，而且以后不费很大功夫就能改变我们的想法。我们同时希望这个样板的代码很容易地与对象的真实行为分离开来，这样我们可以把精力集中在特定领域的实现上。ATL 设计的基本目标是提供这种功能和灵活性。

3.2　ATL 的层次

ATL 把对建立 COM 对象的支持分成几个层次，如图 3.1。这些层次是按照 ATL 为建立对象所暴露的服务来划分的。

1．CComXxxThreadModel 被 CComObjectRootEx 使用，它用来提供恰好足够线程安全（just-safe-thread-enough）的对象生命周期管理和对象锁定功能。

2．CComObjectRootBase 和 CComObjectRootEx 提供了用于实现 IUnknown 的辅助函数。

3．我们的类(即图中的 CYourClass 类 —— 译注)从 CComObjectRootEx 派生。我们的类也必须从希望实现的任何一个接口派生，同时还必须提供这些方法的实现。方法的实现可能由我们来提供，或者通过某个 ATL IXxxImpl 类提供。

4．CComObject 等类提供了 IUnknown 方法的真正实现，它的实现方式与我们对对象和服务器生命周期管理的要求一致。这最后的层次从我们的类派生得到。

我们对基类和最底层次派生类(most derived class，即指 CComObject<T> —— 译注)的选择将决定 IUnknown 方法的实现方式。如果我们的选择改变了，那么在编译时（或者运行时）使用不同的类将会改变 ATL 对于 IUnknown 的实现，而与对象的其他行为无关。下面的部分将探讨 ATL 的每个层次。

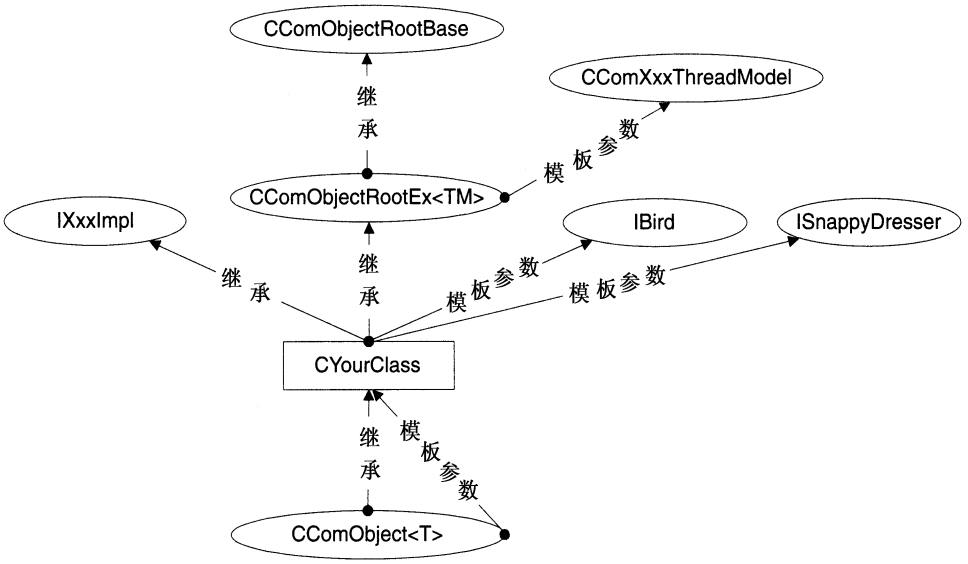

图 3.1　ATL 的层次

3.3　线程模型支持

3.3.1　恰好足够的线程安全

前面见到的 AddRef 和 Release 的线程安全实现对于我们的 COM 对象而言可能并不必要。例如，如果一个指定类的实例仅在单线程套间中运行，那么没有理由使用线程安全的 Win32 函数 InterlockedIncrement 和 InterlockedDecrement。对于单线程对象，下面的 AddRef 和 Release 实现更加有效。

```
class Penguin {
...
  ULONG AddRef()
 { return ++m_cRef; }
```

```
 ULONG Release() {
    ULONG 1 = --m_cRef;
    if( 1 == 0 ) delete this;
    return 1;
 }
...
};
```

虽然在单线程对象中使用线程安全的 Win32 **函数也能正常工作，但是不必要的线程安全带来了应该尽量避免的额外开销。由于这个原因，**ATL 提供了 3 个类 CComSingleThreadModel、CComMultiThreadModel **和** CComMultiSingleThreadModelNoCS。**这些类提供了两个静态成员函数** Increment **和** Decrement**，用于抽象出管理一个对象的生命周期计数器在单线程方式和多线程方式下的差异。下面是这些函数的两个版本（注意在** CComMultiThreadModel **和** CComMultiSingleThreadModelNoCS **中这两个函数的实现是一致的）。**

```
class CComSingleThreadmodel {
 static ULONG WINAPI Increment(LPLONG p) { return ++(*p); }
 static ULONG WINAPI Decrment(LPLONG p) { return –(*p); }
 ...
};

class CcomMultiThreadModel {
 static ULONG WINAPI Increment(LPLONG p)
   { return InterlockedIncrement(p); }
 static ULONG WINAPI Dectrement(LPLONG p)
   { return InterlockedDecrement(p); }
 ...
};

class CcomMultiThreadMode1NoCS {
 static ULONG WINAPI Increment(LPLONG p)
   { return InterlockedIncrement(p); }
 static ULONG WINAPI Decrement(LPLONG p)
   { return InterlockedDecrement(p); }
```

```
 ...
};
```

利用这些类，我们可以对我们的类进行参数化，[2] 以便得到一个恰好线程安全的 AddRef 和 Release 实现，像这样：

```
template <typename ThreadModel>
class Penguin {
...
 ULONG AddRef()
 { return ThreadModel: :Increment(&m_cRef); }

 ULONG Release() {
    ULONG I = ThreadModel::Decrement(&m_cRef);
    if( 1 == 0 ) delete this;
     return I;
 }
...
};
```

现在，根据对 CPenguin 类的要求，我们可以通过把线程模型作为模板的参数使得其恰好线程安全。

```
// Let's make a thread-safe CPenguin
CPenguin* pobj : new CPenguin<CComMultiThreadModel>;
```

3.3.2 实例数据的同步

当我们创建一个线程安全的对象时，仅仅保护对象的引用计数是不够的。我们还必须在多线程访问的情况下保护成员数据。一种流行的多线程数据访问保护方法是使用 Win32 临界区对象(Win32 critical section object)，如下：

```
template <typename ThreadModel>
```

[2] 关于 C++模板的介绍和它们在 ATL 中的使用，请参见附录 A

```
class CPenguin {
publ i c:
 CPenguin(): m_nWingspan(O) { ServerLockO;
      Ini ti al i zeCri ti cal Section (&m_cs); }
 ~CPenguin() { ServerUnlockO; DeleteCriticalSection(&m_cs); }

 // IBi rd
 STDMETHODIMP get_Wingspan(long* pnWingspan) {
     Lock(); // Lock out other threads during data write
    *pnWingSpan = m_nWingspan;
    Unlock();
     return S_OK;
 }

 STDMETHODIMP put_Wingspan(long nWingspan) {
     Lock(); // Lock out other threads during data read
    m_nWingspan = nWingspan;
     Unlock();
     return S_OK;
 }
...
private:
 long m_nWingspan;
 CRITICAL_SECTION m_cs;

 void Lock() { EnterCriticalSection(&m_cs); }
 void Unlock() { LeaveCriticalSection(&m_cs); }
};1
```

注意在读写任何成员数据之前，CPenguin**对象进入临界区，把来自其他线程的访问锁在外面。在一个线程读写期间，这种粗粒度的、对象级的锁定防止调度程序切换给其他可能破坏成员数据的线程。**

但是对象级的锁定不会带给我们希望的并发性能。如果一个对象只有一个临界区，那么当一个线程更新一个相关成员变量的时候，另一个线程增加或者减少引用计数的操作就可能会被阻塞。更高的并发性需要更多的临界区，这样当一个线程访问一个数据成员的时候，也允许第二个线程访问另外一个数据成员。这种细粒度的同步要小心使用，因为它经

常导致死锁。例如：

```
class CSneech : public Isneech
public:
...
 // Isneech
 STDMETHODIMP GoNorthO {
      EnterCriticalSection(&m_csl); // Enter csl...
      EnterCriticalSection(&m_cs2); // ...then enter cs2
      // Go north...
      LeaveCri ti cal Secti on (&re_cs2);
      LeaveCri ti cal Secti on (&m_cs 1);
}

 STDMETHODIMP GoSouthO {
      EnterCriticalSection(&m_cs2); // Enter cs2...
      EnterCriticalSection(&m_csi); // ...then enter csi
      // Go south...
      LeaveCri ti cal Secti on (&m_csl);
      LeaveC ri ti cal Secti on (&re_cs2);
 }
...
private:
 CRITICAL_SECTION  m_csl;
 CRITICAL_SECTION  m_cs2;
};
```

想像调度程序让一个线程执行 GoNorth 方法进入第一个临界区，然后切换到另一个线程执行 GoSouch 方法进入第二个临界区。如果这种情况发生了，没有一个线程能够进入另一个临界区，因而这两个线程都不能继续进行下去，它们死锁了。这种情况应该避免。[3]

无论我们决定使用对象级的锁定还是更细粒度的锁定，临界区是很方便的。ATL 提供了两个包装类来简化它们的使用：CComCriticalSection 和 CComAutoCriticalSection。

```
class CcomCriticalSection
```

[3] 想了解更多关于如何处理死锁的指导，请阅读 Mike Woodring 和 Aaron Coheo 的著作 Win32 Multithreaded Programming（O'Reilly&Associates, 1997）。

```
public:
  void Lock()     { EnterCriticalSection(&m_sec); }
  void Unlock()  { LeaveCriticalSection(&m_sec); }
  void Init()     { InitializeCriticalSection(&m_sec); }
   void Term()    { DeleteCriticalSection(&m_sec); }
  CRITICAL_SECTION m_sec;
};

class CComAutoCriticalSection {
public:
  void Lock()                 { EnterCriticalSection(&m_sec); }
  void Unlock()               { LeaveCriticalSection(&m_sec); }
  CComAutoCriticalSection()   { InitialzeCriticalSection(&m_sec); }
  ~CcomAutoCriticalSection()  { DeleteCriticalSection(&m_sec); }
  CRITICAL_SECTION m_sec;
};
```

注意 CComCriticalSection **没有构造函数和析构函数，但是它提供了** Init **和** Term **函数。这使得当需要一个全局或者静态的临界区，并且** C **运行库（**CRT**，**C Runtime Library**）不能自动地完成构造和析构工作时，**CComCriticalSection **也非常有用。另一方面，如果临界区对象被作为实例的数据成员来创建，并且这时** CRT **可用的话，那么** CcomAuto CriticalSection **更容易使用。例如，在** CPenguin **类中使用** CComAutoCriticalSection **可使代码简捷一些：**

```
templ ate <typename ThreadModel>
class CPenguin {
public:
  // IBird methods call Lock() and Unlock() as before..,
```

但是，注意我们的 CPenguin **仍然通过线程模型来参数化。在单线程的情况下保护数据成员毫无意义。在这种情况下，使用另外一个可以替代** CComCriticalSection **或者** CComAutoCriticalSection **的临界区类会更方便。为了这个目的** ATL **提供了** CcomFake CriticalSection **类。**

```
...
private:
```

```
    CComAutoCri ti cal Section m_cs;

    void Lock() { m_cs. LockO; }
    void Unlock() { m_cs. UnlockO; }
};
```

给出了 CComFakeCriticalSection**，我们可能会通过增加另一个模板参数来进一步参数化** CPenguin **类，但这是不必要的。根据使用的是单线程还是多线程，**ATL **线程模型类已经包含了映射到真正的或者虚拟的临界区的类型定义。**

```
class CcomFakeCriticalSection {
public:
    void Lock()    {}
    void Unlock()  {}
    void Init()    {}
    void Term()    {}
};
```

利用这些类型定义，使得 CPenguin **类对象的引用计数和粗粒度的对象同步都恰好是线程安全的：**

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

class CcomMultiThreadModel {
public:
   static ULONG WINAPI Increment(LPLONG p)
     { retrun InterlockedIncrement(p); }
  static ULONG WINAPI Decrement(LPLONG p)
     { return InterlockedDecrement(p);}
  typedef CcomAutoCriticalSection AutoCriticalSection;
```

```
  typedef CcomCriticalSection CriticalSection;
   typedef CcomMultiThreadModelNoCS tHREADmODELnOcs;
};

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

这种技术允许我们把恰好足够线程安全的操作提供给编译器。当线程模型是 CComSingleThreadModel **时，对** Increment **和** Decrement **的调用变成操作符**++**和**--**，**Lock **和** Unlock **调用变成空的内联函数**(inline functions)**。**

```
template <typename ThreadingModel>
class CPenguin {
public:
 // IBird methods as before...
...
private:
 Threadi ngModel:: AutoCri ti cal Section m_cs;

 void Lock() { m_cs,Lock(); }
 void Unlock() { m_cs.UnlockO; }
};
```

当线程模型是 CComMultiThreadModel **时，对** Increment **和** Decrement **的调用变成对** InterlockedIncrement **和** InterlockedDecrement **的调用，**Lock **和** Unlock **调用变成对** EnterCriticalSection **和** LeaveCriticalSection **的调用。**

表 3.1. 基于线程模型类的展开代码

	CComSingle-ThreadModel	CComMultiThreadModel	CComMulti-ThreadModelNoCS
TM::Increment	++	InterlockedIncrement	InterlockedIncrement
TM::Decrement	--	InterlockedDecrement	InterlockedDecrement
TM::AutoCriticalSection::Lock	(nothing)	EnterCriticalSction	(nothing)
TM::AutoCriticalSection::UnLock	(nothing)	LeaveCriticalSction	(nothing)

最后，当线程模型是 CComMultiThreadModelNoCS 时，对 Increment 和 Decrement 的调用是线程安全的，但是与 CComSingleThreadModel 一样，临界区是假的。CcomMulti ThreadModelNoCS 是为那些多线程对象设计的，这些对象不使用对象级的锁定而倾向于更精细粒度的方法。表 3.1 显示了根据我们使用的线程模型类的情况展开之后的代码。

3.3.3 服务器的缺省线程模型

基于 ATL 的服务器有一个概念 —— 什么是服务器的“缺省”线程模型。在没有直接指定线程模型的情况下使用这个缺省的线程模型。为了设置服务器的缺省线程模型，我们定义以下三个符号之一：_ATL_SINGLE_THREADED、_ATL_APARTMENT_THREADED 或者_ATL_FREE_THREADED。如果没有指定上述符号，那么 ATL 假设为_ATL_FREE _THREADED。但是 ATL COM Appwizard 会在生成的 stdafx.h 文件中定义_ATL_ APARTMENT_THREADED。ATL 使用这些符号来定义两个类型定义。

```
#i f defi ned (_ATLSINGLE_THREADED)
      typedef CComSingleThreadModel CComObjectThreadModel;
      typedef CComSi ngl eTh readModel CComG1 obal sTh readModel;
#el i f defi ned (_ATL_APARTMENT_THREADED)
      typedef CComSingleThreadModel CComObjectThreadModel;
      typedef CComMul tiTh readModel CComG1 obal sTh readModel,
#else // _ATL_FREE_THREADED
      typedef CComMultiThreadModel CComObjectThreadModel;
      typedef CComMul tiTh readModel CComG1 obal sTh readModel;
#endif
```

ATL 在内部使用 CComObjectThreadModel 来保护实例数据，使用 CcomGlobals ThreadModel 来保护全局和静态数据。因为这种用法很难覆盖所有的情况，所以我们应该确保在编译服务器的时候，ATL 使用了该服务器的所有类中最具保护性的线程模型进行编

译。在实际工作中，这意味着哪怕服务器中只有一个多线程类，我们也必须把向导生成的 _ATL_APARTMENT_THREADED 符号改为_ATL_FREE_THREADED。

3.4　IUnknown 的核心

3.4.1 独立服务器的引用计数

为了封装 Lock 和 Unlock 方法，以及恰好线程安全的引用计数，ATL 提供了 CcomObjectRootEx 基类，它按照期望的线程模型进行参数化。[4]

```
Template <class ThreadModel>
Class CComObjectRootEx : public CcomObjectRootBase {
Public:
   typedef ThreadModel _ThreadModel;
   typedef _TreadMdodel::AutoCriticalSection _CritSec;
   typedef CcomObjectLockT<_ThreadModdel> ObjectLock;

   ULONG InternalAddRef() {
      ATLASSERT(m_dwRef != -1L);
      Return _ThreadModel::Increment(&m_dwRef);
   }

   ULONG InternalRelease() {
      ATLASSERT(m_dwRef > 0);
      return _ThreadModel::Decrement(&m_dwRef);
   }
   void Lock()     { m_critsec.Lock(); }
   void Unlock()   { m_critesc.Unlock(); }
private:
   CritSec mcritsec;
};
```

[4] 作为一个优化，ATL 提供了 CComobjectRootEx<CComSingleThreadedModel>的一个特化版本，这个版本没有_CritSec 对象，侧面避开了编译器强制的最小对象尺寸要求。

当对象被独立创建（也就是说，不是聚合）的时候，ATL 类从 CComObjectRootEx 派生并且把对 AddRef 和 Release 的调用转移到 InternalAddRef 和 InternalRelease 方法。

在基类中的 Lock 和 Unlock 方法非常容易使用，我们可能会被误导编写出下面错误的代码：

```
class CPenguin : public CComObjectRootEx<CComMultiThreadModel>, ... {
    STDMETHODIMP get_Wingspan(long* pnWingspan) {
    Lock();
    if( !pnWingspan ) return E_POINTER; // Forgot to Unlock
    *pnWingSpan = m_nWingspan;
    Unlock();
    return S_OK;
 }
...
};
```

为了帮助我们避免这一类错误，CComObjectRootEx 提供了一个被称为 ObjectLock 的类型定义，它以 CComObjectLockT 为基础，而 CComObjectLockT 又以线程模型为模板参数，如下：

```
template <class ThreadModel>
class CComObjectLockT {
public:
 CComObjectLockT(CComObjectRootEx<ThreadModel>* p) {
   if (p) p->Lock();
   m_p = p;
}
~CComObjectLockT() {
   if (m_p) m_p->Unlock();
}
CComObjectRootEx<ThreadModel>* m_p;
};
```

CComObjectLockT 的实例会 Lock 传递给构造函数的对象并且在析构函数中 Unlock 它。ObjectLock 类型定义提供了一种便利的方法来编写无论返回的路径如何都会正确解锁的代码：

```
Class CPenguin : public CComObjectRootEx<CComMultiThreadModel>, ... {
 STDMETHODIMP get_Wingspan(long* pnWingspan) {
    ObjectLock lock(this);
    if( !pnWingspan ) return F_POINTER; // Stack unwind invokes Unlock
    *pnWingSpan = m_nWingspan;
    return S_OK;
 }
...
};
```

3.4.2　表驱动的 QueryInterface

CComObjetRootEx 除了为独立的 COM 对象提供了恰好线程安全的 AddRef 和 Release 方法实现之外，它还通过它的基类 CComObjectRootBase 为 QueryInterface 提供了一个静态的表驱动实现，叫做 InernalQueryInterface。

```
static HRESULT WINAPI
CComObjectRootBase::InternalQueryInterface(
  Void *                    pThis,
  Const _ ATL_INTMAP_ENTRY* pEntries,
  REFIID                    iid,
  Void**                    ppvObject);
```

这个静态成员函数使用 pThis 参数提供的指向对象的 this 指针，并且对于被请求的接口，用一个指向适当 vptr（virtual function table pointer）的指针来填充参数 ppvObject。它使用 pEntries 参数来做到这一点，pEntries 参数是一个以 0 结尾的_ATL_INTMAP_ENTRY 结构的数组。

```
struct _ATL_INTMAP_ENTRY {
  const IID*           piid;
  DWORD                dw;
  _ATL_CREATORARGFUNC*  pFunc;
};
```

COM 对象暴露的每个接口都会成为接口映射表的一个入口，接口映射表是类中一个

静态的_ATL_INTMAP_ENTRY 结构的数组。每个入口由一个接口标识符（IID）、一个函数指针和表示成 DWORD 的函数参数组成。这样提供了一种灵活的、可扩展的机制来实现 QueryInterface，这种机制支持多重继承、聚合、tear-off、嵌套复合、调试、chaining 以及其他任何 C++程序员正在使用的、古怪的 COM 实体身份技巧。[5] 但是，大多数接口通过使用多重继承来实现，所以我们并不经常需要这么多的灵活性。例如，考虑 CPenguin 类的实例的一个可能的对象布局，如图 3.2：

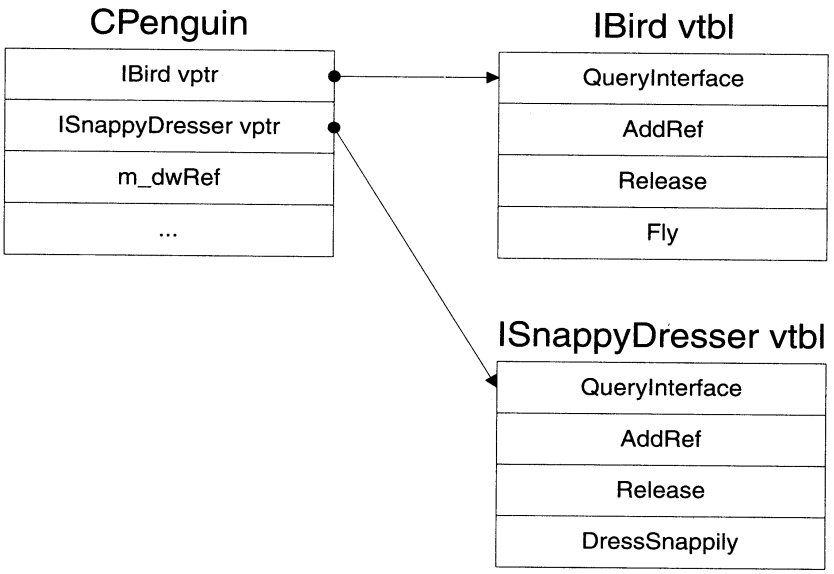

图 3.2. 包含从 vptr 到 vtbl 的 CPenguin 对象布局

```
class Cpenguin : public Ibird, public IsnappyDresser {...};
```

使用多重继承的类的 QueryInterface 的典型实现由一组 if 声明和 static_cast 操作符组成，目的是通过一些固定的偏移量来调整 this 指针，以便指向适当的 vptr。因为这些偏移量在编译时刻就已经知道，所以一个“用来匹配接口标识符到偏移量”的表将会为运行时刻调整 this 指针提供适当的数据结构。为了支持这种常见的情形，如果 pFunc 成员是特殊的值 _ATL_SIMPLEMAPENTRY，那么 InternalQueryInterface 函数把 _ATL_INTMAP_ENTRY 当作一个简单的“IID/偏移量”对。

```
#define ATLSIMPLEMAPENTRY ((_ATL_CREATORARGFUNC*)1)
```

为了能够使用 InternalQueryInterface 函数，每个实现都要生成一个静态接口映射表。为了方便填充这个数据结构和提供其他内部使用的方法，ATL 提供了下面的宏（还有其他

[5] 所有关于这些接口映射表的用法将在第 5 章讲述。

的宏将在第 5 章介绍）：

```
#define BEGIN_COM_MAP(class)...
#define COM_INTERFACE_ENTRY(itf)...
#define END_COM_MAP()...
```

例如，我们的 CPenguin 类会这样声明它的接口映射：

```
    class CPenguin :
 publ i c CComObj ectRootEx<CComMul ti Th readModel >,
     public IBird,
     public ISnappyDresser {
    ...
    public:
    BEGIN_COM_MAP (CPengu i n)
     COM_INTERFACE_ENTRY (IBi rd)
     COM_INTERFACE_ENTRY (ISnappyDresser)
    END_COM_MAPO
    ...
    };
```

上面省略形式可以扩展为下面的内容：

```
    class CPenguin :
     publ i c CComObjectRootEx<CComMul tiThreadModel>,
     public IBird,
     public ISnappyDresser {
    ...
    publ i c:
       IUnknown* GetUnknownO {
        ATLASSERT(_GetEntries() [O].pFunc == _ATL_SIMPLEMAPENTRY);
         return (IUnknown*)C(int)this+_GetEntriesO->dw); }
     }
     HRESULT _InternalQuerylnterface(REFIIO i id, void** ppvObject) {
         return InternalQuerylnterface(this, _GetEntries(), iid, ppvObject);
     }
```

```
const static _ATL_INTMAP_ENTRY* WINAPI _GetEntriesO {
    static const _ATL_INTMAP_ENTRY _entries[] = {
      { &IID_IBird,          O, _ATL_SIMPLEMAPENTRY },
      { &IID_ISnappyOresser, 4, _ATL_SIMPLEMAPENTRY },
     *{ o, o, o }*
  };
   return _entries;
}
...
};
```

图 3.3 展示了在内存中接口映射与 CPenguin 类的实例的关系：

值得一提的还有 BEGIN_COM_MAP 提供的 GetUnknown 成员函数。虽然它在 ATL 内部使用，但是当我们把 this 指针传递给一个需要 IUnknown*的函数时它也很有用。由于我们的类可能从多个接口派生，并且每个接口都从 IUnknown 派生，所以把自己的 this 指针作为 IUnknown*进行传递对编译器来说是有二义性的。例如：

```
HRESULT Fl ylnAnAi rpl ane (IUnknown* punkPassenger);

// Penguin.cpp
STDMETHODIMP CPenguin: :Fly() {
 return FlylnAnAirplane(this); // ambiguous
}
```

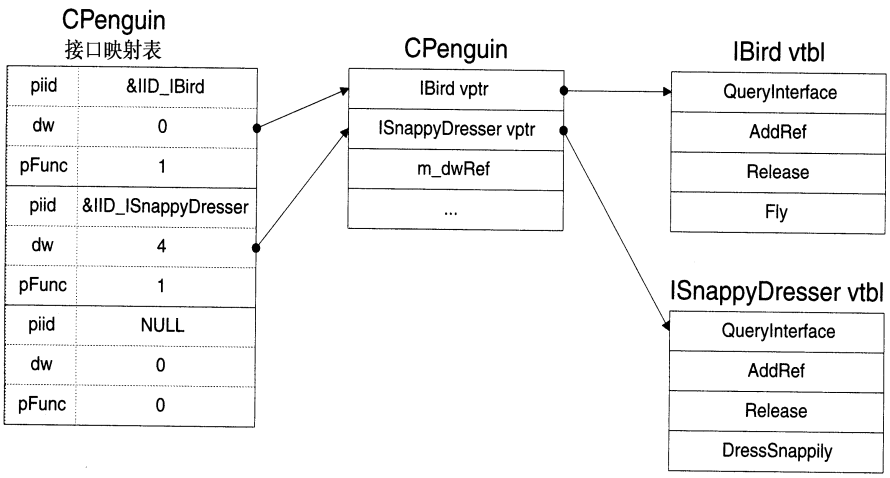

图 3.3　CPenguin 接口映射表、CPenguin 对象和 vtbl

在这种情况下，GetUnknown 会帮助我们：

```
STDMETHODIMP CPenguin: :Fly() {
 return FlylnAnAi rplane(this->GetUnknown()); // unambiguous
}
```

后面我们将会看到，GetUnknown 是通过给出接口映射表的第一个入口实现的。

3.4.3　支持聚合：受控的内部对象

到现在为止，我们已经讨论了独立 COM 对象的 IUnknown 实现。但是，当我们的对象作为受控的内部对象(controlled inner)参与聚合时，我们的工作不能仅仅为自己考虑，而要为其他对象的想法和期望考虑。一个受控的内部对象通过盲目地把所有对 IUnknown 公开实现的调用都转移给外部控制对象的实现来做到这一点。外部控制对象的实现是由 IClassFactory::CreateInstance 方法的 pUnkOuter 参数提供的 。当基于 ATL 的 COM 对象作为受控的内部对象使用时，它简单地把所有对 IUnknown 方法的调用传递给 CComObjectRootBase 提供的 OuterQueryInterface、OuterAddRef 和 OuterRelease 函数，然后再传递到外部控制对象。这里给出 CComObjectRootBase 的有关函数：

```
class CComObjectRootBase {
public:
 CComObjectRootBase() { m_dwRef = OL; }
 ...
 ULONG OuterAddRef();
   return m_pOuterUnknown->AddRef();
}
ULONG OuterRelease() {
   return m_pOuterUnknown->Release();
}
HRESULT OuterQueryInterface(REFIID iid, void ** ppvObject) {
   return
}
...
union {
   long    m_dwRef;
   IUnknown* m_pOuterUnknown;
```

```
 };
};
```

注意 CComObjectRootBase **中对象的引用计数和指向外部控制对象的指针组成一个联合。这也就是说一个对象或者维护自己的引用计数或者被聚合，但是不能两者皆是。这个推论是不正确的。当一个对象被聚合的时候，它必须同时维护一个引用计数和指向外部控制对象的指针。我将在后面讨论，在这种情况下，**ATL **使用一个** CComObjectBase **实例保持** m_pUnkOuter**，然后再从** CComObjectBase **派生保持对象的引用计数。**

虽然利用基类 CComObjectRootEx **的方法可以在我们的类中直接实现** IUnknown **的方法，但是大多数** ATL **类没有这样做。相反把** IUnknown **方法真正的实现留给了从我们的类派生出来的类，例如** CComObject**。在讲完我们的类的责任之后我会讨论这一点。**

3.5 我们的类

因为 ATL **在** CComObjectRootEx **类中提供了** IUnknown **的行为，并且在** CComObject **类（和其他类似的类）中提供了真正的实现，所以我们的类的工作相当轻松：从接口派生并且实现它们的方法。除了确保接口映射表列出了所有我们实现的接口以外，我们可以把大部分** IUnknown **的实现留给** ATL**，并且把精力集中在自定义的功能上。这毕竟是** ATL **首要的目标。**

3.5.1 ATL 的实现类

许多标准的接口有着共同的实现。ATL **提供了许多标准接口的实现类；例如** IPersistStreamInitImpl**、**IConnectionPointContainerImpl **和** IViewObjectExImpl **分别实现了** IPersistStreamInit**、**IConnectionPointContainer**和** IViewObjectEx **接口。这些接口中有些功能非常相似，许多对象都可能实现了这些功能，例如永久性、事件或者枚举器。有些是用于特殊目的而且只与特殊的结构相关，例如控制**(controls)**、支持** Internet **的组件或者** Microsoft **管理控制台扩展（**Microsoft Management Console extensions**）。大多数通用目的接口的实现将在第** 6**、**7 **和** 8 **章中讨论。与控制框架相关的接口实现将在第** 10 **和** 11 **章中讨论。在这里我们讨论一个通用目的接口实现：**IDispatchImp1**。**

3.5.2 脚本支持

为了在脚本环境中访问 COM **对象的功能，**COM **对象必须实现** IDispatch**，如下：**

```
interface IDispatch : IUnknown {
 HRESULT GetTypeInfoCount([out] UINT * pctinfo);

 HRESULT GetTypeInfo([in] UINI iTInfo,
                     [in] LCID lcid,
                     [out] ItypeInfo ** ppTInfo);

 HRESULT Invoke([in] DISPID dispIdMember,
                [in] REFIID riid,
                [in] LCID lcid,
                [in] WORD wFlags,
                [in, out] DISPPARAMS * pDispParams,
                [out] VARIANT * pVarResult,
                [out] EXCEPINFO * pExcepInfo,
                [out] UINT * puArgEfrr);
}
```

IDispatch 最重要的方法是 GetIDsOfNames 和 Invoke。想像下面一行脚本代码：

```
penguin.wingspan = 102
```

这将会变成两个 IDispatch 上的调用。第一个是 GetIDsOfNames，用来查询对象是否支持属性 wingspan。如果答案是“是”，那么对 IDispath 的第二个调用是 Invoke。这个调用将会包含一个标识符（被称作 DISPID），它唯一标识了客户感兴趣的属性或者方法的名称（由 GetIDsOfNames 得到）、执行操作的类型（调用方法、设置属性或者获取属性）、一个参数列表和存放结果的地方（如果有的话）。然后，对象的 Invoke 实现必须解释脚本客户发出的请求。这一般包括：解开参数列表（作为 VARIANT 结构的数组被传递过来）、把它们转换成适当的类型（如果可能的话）、把它们压栈，然后调用一些其他处理真正数据类型（而不是 VARIANT）的方法。

在理论上，对象的实现可以采取任意个有趣、动态的步骤来分析和解释客户的请求。实际上，大多数对象把请求转给一个帮手，它的工作是建立栈和调用对象实现的接口的一个方法来完成真正的工作。帮手使用通常与服务器捆绑在一起的类型库中的类型信息。COM 类型库掌握恰好足够的信息，使得一个 TypeInfo 对象的实例（也就是说，一个实现了 ITypeInfo 的对象）能够完成这项服务。用于实现 IDispatch 的 TypeInfo 对象通常建立在一个双接口的基础上，在 IDL 中的定义如下：

```
[ dual, uuid(44EBF74E-116D-11D2-9828-OO600823CFFB) ]
interface IPenguin : IDispatch {
 [propput] HRESULT Wingspan([in] long nWingspan);
 [propget] HRESULT Wingspan(lout, retval] long* pnWingspan);
          HRESULT Fly();
}
```

用一个 TypeInfo **对象作为帮手使得一个对象可以这样（粗体字代码表示不同实现的差异）来实现** IDispatch**：**

```
class CPenguin :

 public CComObjectRootEx<CComSingleThreadModel>,

 public IBird,
 public ISnappyDresser,
 public IPenguin {
public:
 CPenguin() : m_pTypelnfo(O) {
    IID*       plID   = &IID_IPenguin;
    GUID*     pLIBID : &LIBID_BIRDSERVERLib;
    WORD      wMajor = 1;
    WORD      wMinor = O;
    ITypeLib* ptl= O;
    HRESULT hr = LoadRegTypeLib(*pLIBID, wMajor, wMinor, O, &ptl);
    if(SUCCEEDED(hr) ) {
       hr = ptl->GetTypeInfoOfGuid(*pIID, &m_pTypeInfo);
       ptl ->Rel ease();
   }
 }
virtual ~CPenguinO {

    i f( m_pTypeInfo ) m_pTypeInfo->Release(),
 }

BEGIN_COM_MAP (CPengui n)
```

```
 COM_INTERFACE_ENTRY (IBi rd)
 COM_INTERFACE_ENTRY(ISnappyD resse r)
 COM_INTERFACE_ENTRY (IDi spatch)
 COM_INTERFACE_ENTRY (I Pengui n)
END_COM_MAP ()

 // IDispatch methods
 STDMETHODIMP GetTypeInfoCount(UINT *pctinfo) {
     return (*pctinfo = 1), S_OK;
 }

  STDMETHODIMP GetTypeInfo(UINT ctinfo, LCID lcid, ITypeInfo **ppti) {
     if( ctinfo != 0 ) return (*ppti = 0), DISP_E_BADINDEX;
     return (*ppti = m_pTypeInfo)->AddRefO, S_OK;
 }

  STDMETHODIMP GetIDsOfNames (REFIID ri i d,
                                          OLECHAR ** rgszNames,
                                          UINT cNames,
                                          L/ID 1 ci d,
                                          DISPID *rgdispid) {
     return m_pTypeInfo->GetIDsOfNames(rgszNames, cNames, rgdispid);
}

  STDMETHODIMP Invoke(DISPID di spidMember,
                                 REFIID riid,
                                 LCID 1 cid,
                                 WORD wFlags,
                                 DISPPARAMS *pdi spparams,
                                 VARIANT *pvarResul t,
                                 EXCEPINFO *pexcepi nfo,
                                 UINT *puArgErr) {
      return m_pTypeI nfo->Invoke (stati c_cast<IPengui n*> (thi s),
                                              di spidMember, wFlags,
                                              pdispparams, pvarResult,
                                              pexcepinfo, puArgErr);
```

```
}
// IBird, ISnappyDresser, and IPenguin methods...

private:
 ITypelnfo*  m_pTypelnfo;
};
```

因为这个实现非常典型（它只随双接口类型、接口标识符、类型库标识符和主副版本号变化），所以使用模板类实现起来非常容易。ATL **中** IDispatch **的参数化实现是** IDispatchImpl**。**

```
Template <class T,
          const IDD* piid,
          const GUID* plibid = &CComModule::m_libid,
          WOWD wMajor = 1,
          WOWD wMinor = 0,
          class tihclass = CcomTypeInfoHolder>
class ATL_NO_VTBL IDispatchImpl : public T {...};
```

有了 IDispatchImpl**，**IPenguin **的实现简单了不少：**

```
class CPenguin :
 public CComObjectRootEx<CComMultiThreadModel>,
 public IBird,
 public ISnappyDresser,
 public IDispatchImpl<IPenguin, &IID_IPenguin> {
public:
BEGIN_COM_MAP(CPenguin)
 COM_INTERFACE_ENTRY(IBird)
 COM_INTERFACE_ENTRY(ISnappyDresser)
 COM_INTERFACE_ENTRY(IDispatch)
 COM_INTERFACE_ENTRY(IPenguin)
END_COM_MAPO
// IBird, ISnappyDresser and IPenguin methods...
};
```

3.5.3　支持多个双接口

虽然我不希望，但是这个问题还是会被提出来：“怎样在我的 COM 对象中支持多个双接口？”我的回答是：“你为什么想那样做？”

问题是，我们熟悉的脚本环境需要一个对象实现 IDispatch，而不是支持 QueryInterface 的对象。所以，虽然使用 ATL 可以实现多个双接口，但是我们必须选择哪个实现是“缺省”的，也就是当客户请求 IDispatch 时得到的那个。例如，我们不使用一个“把对象*所有*功能都暴露给脚本客户”的 IPenguin 接口，而是决定使所有接口都成为双接口。

```
[ dual, uuid(...) ] interface IBird : IDispatch {...}
[ dual, uuid(...) ] interface ISnappyDresser : IDispatch { ... };
```

我们可以使用 ATL 的 IDispatchImpl 来实现这两个双接口：

```
class CPenguin :
    publ i c CComObj ectRootEx<CComSi ngl eTh readModel >,
    public IDispatchlmpl<IBird, &IID_IBi rd>,
    public IDispatchlmpl<ISnappyDresser, &IID_ISnappyDresser> {
public:
BEGIN_COM_MAP (CPengui n)
 COM_INTERFACE_ENTRY (I Bi rd)
 COM_INTERFACE_ENTRY(ISnappyD res se r)
 COM_INTERFACE_ENTRY(IDispatch) // ambiguous
END_COM_MAP ()
...
};
```

但是，当我们这样填写接口映射表时，编译器将陷入混乱。记住，宏 COM_INTERFACE_ENTRY 本质上就是对所涉及到的接口的 static_cast。因为有两个不同的接口从 IDispatch 派生，所以编译器不能确定我们想转换到哪个接口。为了解决这个难题，ATL 提供了另一个宏：

```
#define COM_INTERFACE_ENTRY2(itf,branch)
```

这个宏让我们告诉编译器哪一个分支紧跟着 IDispatch 基类的继承层次。使用这个宏允许我们选择缺省的 IDispatch 接口：

```
class CPenguin :
     publ i c CComObj ectRootEx<CComSi ngl eTh readModel >,
     public IDispatchlmpl<IBird, &IID_IBird>,
     public IDispatchlmpl<ISnappyDresser, &IID_ISnappyDresser> {
publ i c:
BEGIN_COM_MAP(CPenguin)
 COM_INTERFACE_ENTRY(IBird)
 COM_INTERFACE_ENTRY(ISnappyDresser)
COM_INTERFACE_ENTRY2(IDispatch, IBird) // Compiles (unfortunately)
END_COM_MAP()
...
};
```

下面是我的反对意见。虽然 ATL **和编译器共同促成了这种用法，但是并不意味着它是一个好方法。*没有正当的理由在一个单独的实现中支持多个双接口*。任何支持** QueryInterface **的客户不需要使用** GetIDsOfNames **或者** Invoke**。只要自定义接口能够满足客户对于参数类型的要求，客户就会非常乐于使用自定义接口。另一方面，不支持** QueryInterface **的脚本客户仅能获得缺省双接口的方法和属性。例如，下面的代码不会正常工作：**

```
// Since IBird is the default, its operations are available
penguin,fly
// Since ISnappyDresser is not the default, its operations aren't
avai I able
penguin.straightenTie // runtime error
```

所以，我给诸位如下的忠告：不要把可重用的、多态的 COM **接口设计成双接口。相反，如果想支持脚本客户，那么像我们第一次定义** IPenguin **那样，定义一个单独的双接口来暴露类的所有功能。这样还有一个附加的好处，这意味着我们只须定义一个支持脚本客户的接口，而不是强迫所有的接口都支持脚本客户。**

3.6 CComObject 等

考虑下面的 C++**类：**

```
class CPenguin :
 public CComObjectRootEx<CComMultiThreadModel>,
 public IBird,
 public ISnappyDresser {
 public:
 BEGIN_COM_MAP(CPenguin)
 COM_INTERFACE_ENTRY (I Bi rd)
 COM_INTERFACE_ENTRY (ISnappyDresse r)
 END_COM_MAP()
// IBird and ISnappyDresser methods...
// IUnknown methods not implemented here
};
```

因为这个类没有实现 IUnknown **的方法，所以下面的代码在编译时会失败：**

```
STDMETHODIMP
CPenguinCO::Createlnstance(IUnknown* pUnkOuter,
                        REFIID ri,d, void** ppv) {
...
   CPenguin* pobj = new CPenguin; // IUnknown not implemented
   ...
}
```

有了 CComObjectRootBase**，我们可以很容易地实现** IUnknown **的方法。例如：**

```
// Server lifetime management
extern void ServerLockO;
extern void ServerUnlock();

class CPenguin :
 public CComObjectRootEx<CComMultiThreadModel>,
 public IBird,
 public ISnappyDresser {
public:
 CPenguinO  { ServerLockO; }
```

```
 ~CPenguin() { ServerUnlock(); }
BEGIN_COM_MAP(CPenguin)
 COM_INTERFACE_ENTRY(IBird)
 COM_INTERFACE_ENTRY(ISnappyDresser)
END_COM_MAPO
// IBird and ISnappyDresser methods...
// IUnknown methods for standalone, heap-based objects
 STDMETHODIMP Querylnterface(REFIID riid, void** ppv)
 { return _InternalQueryInterface(riid, ppv); }
STDMETHODIMP_(ULONG) AddRefO
 { return InternalAddRefO; }

 STDMETHODIMP_(ULONG) Releasei) {
    ULONG I = InternalReleaseO;
    ifc 1 == 0 ) delete this;
    return cRef;
 }
};
```

不幸的是，虽然这个实现使用了基类的行为，但是它对对象的生命周期和实体身份作了固定的假设。例如，这个类的实例不能作为聚合体被创建。就像能把有关线程安全的决定封装进基类一样，我们也希望封装有关生命周期和实体身份的决定。但与线程安全的决定不同（这份决定是基于每个类的）的是，生命周期和实体身份的决定是基于每个实例的。因此把生命周期和实体身份的行为封装到类中意味着要从我们的类派生。

3.6.1 独立激活

为了封装我刚刚展示的、独立的、基于堆的对象的 IUnknown 实现，ATL 提供了 CComObject，下面展示的内容有一点简略。

```
template <class Base>
class CcomObject : public Base {
public:
 typedef Base _BaseClass;

CComObject(void* = NULL)
```

```
{ _Module.Lock(); } // Keeps server loaded

~CComObject() }
   m_dwRef = 1L;
   FinalRelease();
   _Module.Unlock(); // Allows server to unlload
}
STDMETHOD(QueryInterface)(REFIID iid, void ** ppvObject)
{ return _InternalQueryInterface(iid, ppvObject); }

STDMETHOD_(ULONG, AddRef)()
{ retrun InternalAddRef(); }

STDMETHOD_(ULONG, Release)() {
   ULONG 1 = InternalRelease();
   if (1 == 0) delete this;
   return 1;
}

 template <class Q>
 HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp)
 { return QuryIntrface(__uuidof(Q), (void**)pp); }

 static HRESULT WINAPI CreateInstance(CComObject<Base>** pp);
};
```

注意，CComObject **使用了一个被称作** Base **的模板参数。它是** CComObject **的派生基类，**CComObject **从中获得** CComObjectRootEx **的功能和我们的对象所包含的自定义功能。虽然** CPenguin **实现不包含** IUnknown **的方法实现，但是编译器会乐于接受按下面的方式使用** CComObject**(后面我会讲述为什么在创建基于** ATL **的** COM **对象时不能直接使用** new**)：**

```
STDMETHODIMP
CPenguinCO::CreateInstance(IUnknown* pUnkOuter,
                           REFIID riid, void** ppv) {
 *ppv = 0;
 if( pUnkOuter ) return CLASS_E_NOAGGREGATION;
```

```
    // Read on for why not to use new like this!
    CComObject<CPenguin>* pobj = new CComObject<CPenguin>;
    if( pobj ) {
    pobj->AddRef();
    HRESULT hr = pobj->QueryInterface(riid, ppv);
    pobj->Release();
       return hr;
}
    return E_OUTOFMEMORY;
   }
```

除了对 FinalRelease **和静态成员函数** CreateInstance **的调用（都将在本章的“** ATL **创建者”部分讲述），**CComObject **提供了另外一个值得注意的内容 —— **QueryInterface **成员函数模板。**[6]

```
template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp)
{ return QueryInterface(__uuidof(Q), (void**)pp); }
```

这个成员函数模板使用了 Visual C++（VC++）**编译器的新功能 —— 用全局唯一标识符（**UUID **，**universally unique identifier**）来标识一个类型。这个功能从** VC++ 5.0 **后开始提供，并且使用说明性编译指示符（**declspec **，**declarative specifier**）的形式，例如：**

```
struct __declspec(uuid("00000000-0000-0000-C000-000000000046"))IUnknown
  {...};
```

这些 declspec **是** Microsoft IDL **编译器的输出，既可以用于标准接口，也可以用于自定义接口。我们可以使用**__uuidof **操作符来得到一个类型的** UUID**，下面的语法是允许的：**

```
void TryToFly(IUnknown* punk) {
 IBird* pbird = 0;
 if( SUCCEEDED(punk->Querylnterface(__uuidof(pbi vd), (void**)&pbi rd) )
{
    pbi rd->FlyO;
    pbi rd->Rel ease();
```

```
 }
}
```

有了基于 CComObject **的对象的引用，使用** CComObject **提供的** QueryInterface **成员函数模板带来一点语句结构上的便利。例如：**

```
vold TryToFly (CComObj ect<CPenguin>* pPenguin) {
 TBird* pbird = O;
 if(SUCCEEDED(pPenguin->Querylnterface(&pbird) ) {
    pbi rd->FlyO;
    pbi rd->Rel ease();
 }
}
```

3.6.2　被聚合对象的激活

注意前面展示的 CPenguin **类对象的实现通过检查非零的** pUnkOuter **和返回** CLASS_E_NOAGGREGATION **来禁止聚合。如果我们也想支持聚合或者只想支持聚合，我们需要另一个类来实现被聚合实例的传递调用行为。为此，**ATL **提供了** CComAggObject**。**

CComAggObject **实现了作为受控内部对象的主要服务，也就是说，提供了** IUnknown **的两个实现。一个实现把调用转发到外部控制对象，包括它的生命周期和实体身份服务。另一个被外部控制对象用于私有用途，用来真正维护内部的生命周期和接口查询。为了得到** IUnknown **的两个实现，**CComAggObject **从** CComObjectRootEx **继承了两次：一次直接继承、另一次通过包容我们从** CComContainedObject **派生的类的实例间接地继承，如下：**

```
template <class contained>
class CComAggaObject :
 public IUnknown,
 public CComObjectRootEx<contained::_ThreadModel::ThreadModelNoCS> {
public:
 typedef contained _BaseClass;
 CcomAggOject(void* pv) : m_contained(pv)
 { _Module.Lock(); }
```

[6] **附录** A **有成员函数模板的简单描述。**

```
 ~CComAggObject() {
   m_dwRef = 1L;
   FinalRelease();
   _Module.Unlock();
}

STDMETHOD(QueryInterface)(REFIID iid, void ** ppvObject) {
  HRESULT hRes = S_OK;
  if (InlineIsEqualUnknown(iid)) {
    if (ppvObject == NULL) return E_pointer;
    *ppvObject = (void*)(IUnknown*)this;
    AddRef();
  }
  else
    hRes = m_contained._InternalQuaryInterface(iid, ppvObject);
  return hRes;
}

STDMETHOD_(ULONG, AddRef)()
{ return InternalAddRef(); }

STDMETHOD_(ULONG, Release)() {
  ULONG 1 = InternalRelease();
  if (1 == 0) delete this;
  return 1;
}

  template <class Q>
  HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp)
  { return QueryInterface(__uuidof(Q), (void**)pp); }

  static HRESULT WINAPI CreateInstance(LPUNKNOWN pUnkOuter,
    CcomAggObject<contained>** pp);

  CcomContainedObject<contained> m_contained;
```

```
};
```

我们可以看到 CComAggObject 不是从我们的类（作为模板参数传递）派生的，而是直接从 CComObjectRootEx 派生。它对 QueryInterface 的实现依赖于我们的类建立起来的接口映射表，但是它的 AddRef 和 Release 的实现依赖于 CComObjectRootBase 的第二个实例，CComAggObject 之所以得到这个实例是因为它继承了 CComObjectRootEx。这第二个 CComObjectRootBase 实例使用联合中的 m_dwRef 成员。

CComObjectRootBase 的第一个实例管理联合中的 pOuterUnknown 成员，它是 CComAggObject 通过把 CComContainedObject 作为 m_contained 成员（并且 Ccom ContainedObject 又继承了我们的类)而得到的。CComContainedObject 通过把调用委托给 m_pOuterUnknown 实现了 QueryInterface、AddRef 和 Release，而 m_pOuterUnknown 是在构造函数中被传递进来的，CComContainedObject 的代码如下：

```
Template <class Base>
Class CcomContainedObject : public Base {
Public:
   Typedef Bae _bBaseClass;
   CcomContainedObject(void* pv)  { m_pOuterUnknown = (IUnknown*)pv; }

 STDMETHOD(QueryInterface)(REFIID iid, void ** ppvObject) {
   HRESULT hr = OuterQueryIntrface(iid,ppvObject);
   if (FAILED(hr) && CetRawUnknown() != mpOuterUnknown)
    hr = _InternalQueryInterface(iid, ppvObject);
   return hr;
}

STDMETHOD_(ULONG,AddRef)()
{ return OuterAddRef(); }

STDMETHOD_(ULONG, Release)()
{ return OuterRelease(); }

template <class Q>
HRESULT STDMETHODCALLTYPE QueryInterface(Q** pp)
{ return QueryInterface(__uuidof(Q), (void**)pp); }
```

```
IUnknown* GetControllingUnknown()
{ return m_pOuterUnknown; }
};
```

3.6.3 作为受控内部对象

利用 CComAggObject **和它的两个** IUnknown **实现，我们的** CPenguin **类对象的实现在不修改源代码的情况下既可以支持独立的服务器也可以支持聚合激活：**

```
STDMETHODIMP
CPenguinCO::Createlnstance(IUnknown* pUnkOuter,
                        REFIID riid, void** ppv) {
 *ppv = O;
 if( pUnkOuter ) {
      CComAggObject<CPenguin>* pobj =
      new CComAggObj ect<CPengui n> (pUnkOuter);
...
}
el se {
    CComObject<CPenguin>* pobj = new CComObject<CPenguin>;
    ...
 }
}
```

这种用法提供了最有效的运行时刻决策。当对象是独立的时，它付出的代价是一个引用计数和一个 IUnknown **实现。当它被聚合时，它付出的代价是一个引用计数、一个指向外部控制对象的指针，以及两个** IUnknown **实现。但是我们付出的额外代价是一套多余的** vtbl(**虚表**)**。由于既使用** CComAggObject<CPenguin>**又使用** CComObject<CPenguin>**，因此我们创建了两个类和两套** vtbl**。**

如果只有少量的实例或者几乎所有的实例都是被聚合的，则我们可能希望一个类既可以处理聚合激活又可以处理独立激活，从而去掉一个 vtbl**。我们可以通过使用** CComPloyObject **替代** CComObject **和** CComAggObject **来做到这一点，像这样：**

```
STDMETHODIMP
CPenguinCO::Createlnstance(IUnknown* pUnkOuter,
                        REFIID riid, void** ppv) {
```

```
      *ppv = 0;
      CComPolyObject<CPenguin>* pobj :
         new CComPolyObject<CPenguin>(pUnkOuter);
     ...
}
```

CComPloyObject 和 CComAggObject 除了构造函数以外完全相同，如果 pUnkOuter 是 0，它会使用 IUnknown 的第二个实现作为第一个 IUnknown 实现转发调用的外部控制对象，如下：

```
class CComPolyObject :
   public IUnknown,
   public CComObjectRootEx<contained::_ThreadModel::ThreadModelNoCS> {
public:
   ...
   CComPolyObject(void* pv) : m_contained(pv ? pv : this) {...}
   ...
};
```

使用 CComPloyObject 节省了一套 vtbl，所以模块的尺寸较小，但是在作为独立对象的情况下付出的代价是一个多余的 IUnknown 实现和一个指向这个实现的多余指针。

3.6.4　其他激活技术

除了独立操作以外，CComObject 还对对象的分配方式（堆）和“对象的存在是否使服务器一直处于装载状态”（是）作了假设。为了其他需要，ATL 提供了另外 4 个类：CComObjectCached、CComObjectNoLock、CComObjectGlobal 和 CComObjectStack，在我们的实现层次中它们是最终被派生出来的类。

📖　CComObjectCached

CComObjectCached 对象基于下面的假设实现引用计数 —— 我们将创建一个实例并且在整个服务器生命周期内拥有它，一旦被请求就提交它的引用。虽然对象的生命周期仍然按照边界为 0 进行管理，但是为了避免被缓冲起来的实例创建之后服务器一直运行，所以使服务器保持运行状态的边界是引用计数为 1，像这样：

```
Template <class Base>
class CcomObjectCached : public Base {
public:

 ...
 STDMETHOD_(ULONG, AddRef)() {
   m_csCached.Lock();
   ULONG 1 = InternalAddRef();
   if (m_dwRef == 2)_Module.Lock();
   m_csCached.Unlock();
   return 1;
 }
 STDMETHOD_(ULONG, Release)() {
   m_csCached.Lock();
   InternalRelease();
   ULONG 1 = m_dwRef;
   m_csCached.Unlock();
   if (1 == 0) delete this;
   else if (l == 1) _Module.Unlock();
   return l;
}
 ...
 CComGlobalsThreadModel::AutoCriticalSection m_csCached;
};
```

缓冲对象对于进程内的类对象非常有用：

```
static CComObjectCached<CPenguinCO>* g_pPenguinCO = 0;

BOOL WINAPI DllMain(HINSTANCE, DWORD dwReason, void*) {
   switch( dwReason ) {
   case DLL_PROCESS_ATTACH:
      g_pPenguinCO = new CComObjectCached<CPenguinCO>;

      // 1st ref. doesn't keep server alive
      if( g_pPenguinCO ) g_pPenguinCO->AddRef();
```

```
   break;

   case DLL_PROCESS_DETACH:
       if( g_pPenguinCO ) g_pPenguinCO->ReleaseO;
   break;
   }
}
    return TRUE;
}

STDAPI DllGetClassObject(REFCLSlD clsid, REFIID riid, void** ppv) {
   // Subsequent references do keep server alive
    if( clsid == CLSID_Penguin && g_pPenguinCO )
       return g_pPenguinCO->Querylnterface(riid, ppv);
    return CLASS_E_CLASSNOTAVAILABLE;
}
```

📖 CComObjectNoLock

有时我们不希望对象引用的存在使服务器一直存活。例如，进程外服务器的类对象被 ole32.dll **维护的一个表缓冲了多次(一次不够)。由于这个原因，**COM **自己使用** IClassFactory **的** LockServer **方法来管理“一个类对象的生命周期如何影响它的进程外服务器的生命周期”。为了这种用法，**ATL **提供了** CComObjectNoLock**，它的实现不影响服务器的生命周期。**

```
Template <class Base>
class CcomObjectNoLock : public Base {
public:
   ...
   STDMETHOD_(ULONG, AddRef)()
   { return InternalAddRef(); }

STDMETHOD_(ULONG, Release)() {
   Ulong 1 = InternalRelease();
   if (1 == 0) delete this;
   return 1;
 }
```

```
 ...
};
```

无锁对象对于进程外的类对象非常有用。

```
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int){
   Colnitialize(O);

   CComObjectNoLock<CPenguinCO>* pPenguinCO =
      new CComObjectNoLock<CPenguinCO>;
   if( !pPenguinCO ) return E_OUTOFMEMORY;
   pPengui nCO->AddRef();

   DWORD   dwReg;
   HRESULT hr;

  // Reference(s) cached by ole32.dll
  // won't keep server from shutting down
   hr = CoRegisterClassObject(CLSID_Penguin, pPenguinCO,  ....  &dwReg);
   if(SUCCEEDED(hr) ) {
      MSG msg; while( GetMessage(&msg, O, O, O) ) DispatchMessage(&msg);
       CoRevokeC1 assObj ect (dwReg);
       pPengui nCO->Rel ease();
 }

   CoUninitialize();
   return hr;
}
```

🕮 CComObjectGlobal

就像有时“对象的存在或者未完结的引用并不使服务器保持存活”会提供方便一样，有时“使一个对象的生命周期等同于服务器的生命周期”也会提供便利。例如，一个全局或者静态对象在服务器装载时构造一次，并且在 WinMain 或 DllMain 完成之前一直不析构。显然，仅仅是全局对象的存在并不能使服务器保持运行，或者服务器永远不能关闭。但是，我们希望在全局对象有引用时服务器能够保持运行。为了这个目的，我们使用

CComObjectGlobal。

```
template <class Basc>
class CComObjectGlobal : public Base {
public:
  ...
  STDMETHOD_(ULONG, AddRef)()  { return _Module.Lock(); }
  STDMETHOD_(ULONG, Release)() { return _Module.Unlock(); }
  ...
};
```

在实现进程内类对象时，全局对象可以替代缓冲对象使用，而且全局和静态对象也可以使用它们：

```
// No references yet, so server not forced to stay alive
static CComObjectGlobal<CPenguinCO> g_penguinCO;

STDAPI DllGetClassObjectCREFCLSID clsid, REFIID riid, void** ppv) {
   // All references keep the server alive
   if( clsid == CLSID_Penguin )
     return g_penguinCO.Querylnterface(riid, ppv);
   return CLASS_E_CLASSNOTAVAILABLE;
}
```

注意全局对象比缓冲对象更容易使用，因为我们依赖运行库来正确地构造和析构对象。自动构造和析构全局和静态对象需要链接 CRT，但是这正是建立基于 ATL 的 COM 服务器经常需要回避的地方（像第 4 章讨论的那样）。

🕮 CComObjectStack

与使用全局和静态对象不同，我们可能发现迫切地需要在栈上分配一个 COM 对象。ATL 提供的 CComObjectStack 支持这一技术。

```
template <class Base>
class CcomObjectStack : public Base {
public:
 ...
```

```
STDMETHOD_(ULONG, AddRef)()
{ ATLASSERT(FALSE); return 0; }

STDMETHOD_(ULONG, Release)()
{ ATLASSERT(FALSE); return 0; }

STDMETHOD(QueryInterface)(REFIID idd, void** ppvObject)
{ ATLASSERT(FALSE); return E_NOINTERFACE; }
...
};
```

根据上面的实现代码，可以清楚地看到我们根本不是在使用 COM。**虽然** Ccom ObjectStack **会让编译器顺利编译，但是我们仍然不能使用** IUnknown **的任何方法，这意味着我们不能向外传递一个栈上对象的接口引用。这是正确的，因为如同引用栈上的任何其他事物一样，一旦栈失效了，引用就指向了垃圾数据。**ATL **对** CComObjectStack **实现的优点在于它会在运行时（对于调试版本）警告我们 —— “你们正在干坏事”。例如：**

```
void DoABadThing(IBird** ppbird) {
 CComObjectStack<CPengui n> pengui n;
 penguin. Fly();           // Using IBird method is OK
 penguin. StraightenTie(); // Using ISnappyDresser method also OK

 // This will trigger an assert at runtime
 penguin. QueryInterface (IID_IBi rd, (vol d**) ppbi rd);
}
```

📖 一个来自 A 列、两个来自 B 列

表 3.2 **显示了** ATL **提供的各种实体身份和生命周期选项。**

表 3.2. ATL **的实体身份和生命周期选项**

类	独立/被聚合	堆/栈	对象存在使服务器存活	未完结引用使服务器存活
CComObject	独立	堆	是	是
CComAggObject	聚合	堆	是	是

续 表

类	独立/被聚合	堆/栈	对象存在使服务器存活	未完结引用使服务器存活
CComPolyObject	独立或被聚合	堆	是	是
CComObjectCached	独立	堆	否	第二个引用
CComObjectNoLock	独立	堆	否	否
CComObjectGlobal	独立	数据段	否	是
CComObjectStack	独立	栈	否	否

3.7　ATL 创建者

3.7.1　多步骤构造

我已经提到（在第 4 章中我们会了解得更多），ATL 服务器并不经常链接 CRT。但是离开 CRT 的生活是痛苦的。其中包括：如果没有 CRT，我们就不能获得 C++的异常。在下面特定的情况下我们几乎无事可做。

```
// CPenguin constructor
CPenguin: :CPenguinO {
 HRESULT hr = CoCreatelnstance(CLSID_EarthAtmosphere, O, CLSCTX_ALL,
                          liD_IAi r, (void**)&m_pAi r);
 if(FAILED(hr) ) {
     return hr; // Can't return an error from actor
    throw hr;  // Can't throw an error without the CRT
    // This won't help much
    OutputDebugString(__T("Help! Can't bre...\n"));
 }
}
```

OutputDebugString 不会通知客户正在创建的对象没有获得生存所需要的资源；也就是说，没有办法把失败的结果返回给客户。这好像是不公平的，因为创建对象的 IClassFactory::CreateInstance 方法能够返回一个 HRESULT。问题是需要有一种方法能把来自实例的失败信息交给类对象，使得它可以返回给客户。按照习惯，ATL 类提供了一个被

称为 FinalConstruct 的公有成员函数来参与多步骤的构造过程。

```
HRESULT FinalConstruct();
```

CComObjectRootBase 为 FinalConstruct 成员函数提供了一个空的实现，这样所有的 ATL 对象都有一个 FinalConstruct。因为 FinalConstruct 返回一个 HRESULT，所以现在我们有了一个明确的途径来获得任何值得注意的构造结果。

```
HRESULT CPenguin:: FinalConstruct() {
return CoCreatelnstance(CLSID_EarthAtmosphere, O, CLSCTX_ALL,
                    IID_IAi r, (void**)&m_pAi r);
}

STDMETHODIMP
CPengui nCO:: CreateInstance (IUnknown* pUnkOute r,
                          REFIID riid, void** ppv) {
   *ppv = O;
   if( !pUnkOuter ) {
      CComObject<CPenguin>* pobj = new CComObject<CPenguin>;
      if( !pobj ) return E_OUTOFMEMORY;
      HRESULT hr : pobj->FinalConstructO;
      if(SUCCEEDED(hr) ) ...
      return h r;
  }
  ...
}
```

但是我们还需要考虑一些事情。注意当 CreateInstance 调用 FinalConstruct 时，并没有增加对象的引用计数。这导致了一个问题：如果在 FinalConstruct 执行期间，对象把对自己的一个引用给了另一个对象会发生什么事情？如果您认为这种情况并不常见，那么请回忆 IClassFactory 中方法 CreateInstance 的 pUnkOuter 参数。即使没有聚合，也可能会碰上这个问题。想像下面这段虽然有点人为、但却完全合法的代码：

```
// CPenguin implementation
HRESULT CPenguin' 'FinalConstruct() {
   HRESULT hr,
   hr = CoCreateInstance(CLSID_EarthAtmosphere, O, CLSCTX_ALL,
```

```
                        IID_TAi r, (void**)&m_pAi r);
    if(SUCCEEDED(hr) ) {
       // Pass reference to object with reference count of O
       hr = m_pAi r->CheckSuitabil ity(GetUnknownO);
   }
    return hr;
}

// CEarthAtmosphere implementation in separate server
STDMETHODIMP CEarthAtmosphere' 'CheckSuitabi 1 i ty(ZUnknown* punk) {
    IBreatheO2* pbo2 = O;
    HRESULT     hr = E_FATL;

    // CPenguin's lifetime increased to I via QI
    hr = punk->QueryInterfaceCIID_IBreatheO2, (void**)&pbo2);
    ifc  SUCCEEDED(hr)  )   {
       pbo2->ReleaseO; // During this call, lifetime decreases to O and
                       // destruction sequence begins...
   }
    return (SUCCEEDED(hr) ? S_OK : E_FAIL);
}
```

为了避免过早析构的问题，我们需要在调用 FinalConstruct **之前人为地增加对象的引用计数，然后在调用之后减小引用计数，像这样：**

```
STDMETHODIMP
CPenguinCO::Createlnstance(IUnknown* pUnkOuter,
                           REFIID riid, void** ppv) {
    *ppv = O;
    if( !pUnkOuter ) {
      CComObject<CPenguin>* pobj = new CComObject<CPenguin>;
       if(FAILED(hr) ) return E_OUTOFMEMORY;

      // Protect object from premature destruction
       pobj ->Internal AddRef 0;
       hr = pobj->Fi nalConstruct();
```

```
        pobj->Internal Rel ease();

      if(SUCCEEDED(hr) ) ...
      return hr;
   }
   ...
  }
```

3.7.2 恰好足够安全的引用计数

并不是所有对象的引用计数都需要按照我前面描述的方式进行管理。事实上，对于不需要这种保护的多线程对象，对 InterlockedIncrement **和** InterlockedDecrement **的多余调用代表了不必要的开销。最后，**CComObjectRootBase **提供一对函数来保证用恰好安全的引用计数方式来括起对** FinalConstruct **的调用：**

```
STDMETHODIMP
CPenguinCO: :Createlnstance(IUnknown* pUnkOuter,
                        REFIID riid, void** ppv) {
   *ppv = O;
   if( !pUnkOuter ) {
      CComObject<CPenguin>* pobj = new CComObject<CPenguin>;
      if(FAILED(hr) ) return E_OUTOFMEMORY;

      // Protect object from premature destruction (maybe)
      pobj ->Internal Fi nal Const ructAddRef 0;
      hr = pobj->FinalConstruct();
      pobj ->Internal Fi nal Const ructRel ease();

     if(SUCCEEDED(hr) ) ...
     return h r;
  }
  ...
  }
```

缺省情况下，InternalFinalConstructAddRef **和** InternalFinalConstructRelease **在发行版本（**release-build**）中并没有运行开销。**

```
class CComObjectRootBase{
public:
   ...
   void InternalFinalConstructAddRef() {}
   void InternalFinalConstructRelease() { ATLASSERT(m_dwRef == 0); }
   ...
};
```

为了改变 InternalFinalConstructAddRef 和 InternalFinalConstructRelease 的实现以便提供引用计数的安全性，ATL 提供了以下的宏：

```
#define DECLARE_PROTECT_FINAL_CONSTRUCT() \
   void InteralFinalConstructAddRef() { InternalAddRef(); } \
   void InteralFinalConstructRelease() { InternalRelease(); }
```

DECLARE_PROTECT_FINAL_CONSTRUCT 宏按照需要为每个类启用引用计数安全性。我们的 CPenguin 会这样使用：

```
class Cpenguin : ... {
public:
   HRESULT FinalConstruct();
   DECLARE_PROTECT_FINAL_CONSTRUCT()
   ...
};
```

我个人的观点是：DECLARE_PROTECT_FINAL_CONSTRUCT 是一个 ATL 过于优化的地方。使用它不仅需要大量 COM 和 ATL 的内部知识，而且需要大量在 FinalConstruct 方法中创建对象的实现知识。因为我们经常不具备那些知识，所以唯一安全的方法是一直使用 DECLARE_PROTECT_FINAL_CONSTRUCT。并且由于规则过于复杂，我认为大多数人会忘记它。因此，我给出一个简单一些的规则：

无论哪里出现 FinalConstruct *成员函数，必须也出现一个* DECLARE_PROTECT_FINAL_CONSTRUCT *宏的实例。*

幸运的是，向导在产生一个新类时会生成 DECLARE_PROTECT_FINAL_CONSTRUCT，这样我们的 FinalConstruct 代码缺省情况下就是安全的。如果不需要，我们

可以删除它。[7]

多步骤构造的另一个原因

假设一个 C++类希望在构造过程中调用虚的成员函数，而另一个 C++类重载了这个函数：

```
class Base {
public:
 Base() { Init(); }
 virtual void Init() {}
};

class Derived : public Base {
public:
 virtual void Init() {}
};
```

因为作为构造序列的一部分而调用虚成员函数是非常罕见的，所以没有多少人知道在 Base 的构造函数中调用的 Init 并不是 Derived::Init 而是 Base::Init。虽然这可能违反直觉，但是这样做有充足的理由：在派生类被正确地构造出来之前，调用派生类的虚成员函数并没有意义。但是，一直要到基类被构造之后，派生类才会被正确地构造。为了保证在构造期间只能调用已经被正确构造的类的函数，C++编译器建立了两个 vtbl，一个为 Base，另一个为 Derived。随后 C++运行库会在构造序列期间调整 vptr 使它指向适当的 vtbl。

虽然这都是 C++的标准，但是并不直观，特别是由于它很少被用到（或者正是由于不直观所以很少使用）。因为很少使用它，所以 Microsoft 在 Visual C++5.0 中引入了 _declspec(novtbl)来禁止构造期间对 vptr 的调整。当基类是一个抽象基类时，这经常导致编译器产生了 vtbl，但是没有使用它，因此链接器把它们从最终的映象中删除。

当一个类使用 ATL_NO_VTBL 宏声明时，ATL 也使用了这项优化：

```
#ifdef ATL_DISABLE_NO_VTBL
#define ATI_NO_VTBL
#else
#define ATL_NO_VTBL __declspec(novtb1)
#endif
```

[7] 正如我的朋友 Tim Ewald 一直说的那样，“删除代码总是比添加代码容易。”

如果没有定义_ATL_DISABLE_NO_VTBL，一个使用_ATL_NO_VTBL 定义的类会用__declspec（novtbl）调整它的构造行为。

```
Class ATL_NO_VTBL Cpenguin ... {};
```

虽然这是一个好的并且是真正的优化，但是使用它的类不能在构造函数中调用虚成员函数。[8] 不要在构造函数中调用虚成员函数，而是在 FinalConstruct 中调用它们，当最终派生类的构造函数完成，并且编译器把 vptr 调整为正确的值之后，ATL 再调用这个函数。

关于__declspec(novtbl)还有最后一点值得一提，就是：因为在构造时关闭了对 vtbl 的调整，所以在析构时也关闭了对 vtbl 的调整。因此，在析构函数中也要避免使用虚成员函数；相对应地，在对象的 FinalRelease 成员函数中调用它们。

📖 FinalRelease

在我们的 ATL 对象的最后一个接口引用被释放之后、但是在析构函数被调用之前，ATL 会调用对象的 FinalRelease 函数。

```
void FinalRelease();
```

FinalRelease 用于调用虚成员函数和释放另一个有反向指针（指向当前对象）的对象接口。因为另一个对象可能希望在关闭序列期间查询一个接口，所以防止对象析构两次与防止在 FinalConstruct 中过早的析构一样重要。即使 FinalRelease 成员函数在对象的引用计数减到 0（这是对象被销毁的原因）之后被调用，FinalRelease 的调用者也会手工地把引用计数置为 1 来避免两次删除。FinalRelease 的调用者将会是最终派生类的析构函数。例如：

```
CComObject::~CComObject() {
   m_dwRef = 1L;
   FinalRelease();
   _Module.Unlock();
}
```

[8] 严格地说，编译器会静态地绑定在构造函数或者析构函数中的虚调用，但是如果一个静态绑定的函数调用一个动态绑定的函数，仍然有很大的麻烦。

3.7.3 ATL 创建者

因为管理多步骤构造过程的额外步骤很容易忘记，所以 ATL 在一些被称为创建者（creator）的 C++类中封装了这个算法，其中每一个创建者类都可以完成正确的多步骤构造过程。事实上，每个创建者类仅仅是一种包装单个静态成员函数 CreateInstance 的方法：

```
static HRESULT WINAPI CreateInstance(void* pv,
                                     PEFIID riid, LPVOID* ppv);
```

后面我会讲到，创建者类的名字被用于与这个类相关的类型定义中。

📖 CComCreator

CComCreator **是一个既可以创建独立实例又可以创建聚合实例的创建者，被创建的 C++类是它的模板参数。例如，**CComObject<CPenguin>**。**CComCreator **的声明如下：**

```
Template <class T1> class CcomCreator {
Public:
 Static HRESULT WINAPI CreateInstance(void* pv,
                                      REFIID riid, LPVOID* ppv) {
   ATLASSERT(*ppv == NULL);
   HRESULT hRes = E_OUTOFMEMORY;
   T1* p= NULL;
   ATLTRY(p = new T1(pv))
   if (p != NULL) {
     p->SetVoid(pv);
     p->InternalFinalConstructAddRef();
     hRes = p->FinalConstruct();
     p->InternalFinalConstructRelease();
     if (hRes == S_OK) hRes = p->QueryInterface(riid, ppv);
     if (hRes != S_OK) delete p;
   }
   return hRes;
 }
};
```

使用 CComCreator 大大简化了我们的类对象的实现：

```
STDMETHODIMP
CPenguinCO::CreateInstance(IUnknown* pUnkOuter,
                           REFIID riid, void** ppv) {
   typedef CComCreator< CComPolyObject<CPenguin> > PenguinPolyCreator;
   return PenguinPolyCreator::CreateInstance(pUnkOuter, riid, ppv);
}
```

注意上面代码中使用类型定义来定义新的创建者类型。如果需要在服务器的其他位置创建 CPenguin**，我们必须重新建立类型定义。例如：**

```
STDMETHODIMP CAviary::CreatePenguin(IBird** ppbird) {
   typedef CComCreator< CComObject<CPenguin> > PenguinCreator;
   return PenguinCreator::CreateInstance(O, IID_IBird, (void**)ppbird);
}
```

像这样在被创建的类之外定义创建者有两个问题。第一，它复制了类型定义的代码。第二，更重要的是，它剥夺了我们选择 CPenguin **类是否支持聚合的权利，因为类型定义已经作出了决定。按照** ATL **的惯例 —— 减少代码和让类的设计者作出独立或者聚合激活的决定，我们把类型定义放入类的声明中并且给它一个众所周知的名字**_CreatorClass**：**

```
class CPenguin : ... {
public:
...
   typedef CComCreator< CComPolyObject<CPenguin> > _CreatorClass;
};
```

使用创建者类型定义，创建实例和获得初始接口的代码实际上比使用操作符 new **和** QueryInterface **的代码要少：**

```
STDMETHODIMP CAviary::CreatePenguin(IBird** ppbird) {
    return CPenguin::_CreatorClass::CreateInstance(O,
                                                   IID_IBird,
                                                   (void**)ppbird;
}
```

在第 4 章中，我会讲到另一个我们经常需要继承的基类：CComCoClass**。例如：**

```
class CPenguin : ..., public CComCoClass<CPenguin, &CLSlD_Penguin>, ...
  {...};
```

CComCoClass **提供了两个静态成员函数，都被称作** CreateInstance**，它们是这样使用类的创建者的：**

```
Temlate <class T, const CLSID* pclsid = &CLSID_NULL>
Class CComCoClass {
Public:
  ...
  template <class Q>
  static HRESULT CreateInstance(IUnknown* punkOuter, Q** pp)
  {
    return T::_CreatorClass::CreateInstance(punkOuter, __uuidof(Q),
                                            (void**) pp);
  }

  template <class Q>
  static HRESULT CreateInstance(Q** pp)
  {
    return T::_CreatorClass::CreateInstance(NULL, __uuidof(Q),
                                            (void**) pp);
 }
};
```

这更加简化了我们的创建代码：

```
STDMETHODIMP CAviary::CreatePenguin(IBird** ppbird) {
   return CPenguin: :Createlnstance(ppbi rd);
}
```

🕮 CComCreator2

因为 CComPolyObject **在独立情况下的开销问题，所以我们可能需要使用** CComObject

和 CComAggObject 而不是 CComPolyObject 来同时支持独立激活和聚合激活。为了避免把“根据 pUnkOuter 是否为 NULL 来选择适当的创建者”的逻辑放回类对象的 CreateInstance 方法中， ATL 提供了 CComCreator2：

```
template <class Ti, class T2> class CComCreator2 {
public:
   static HRESULT WINAPI Createlnstance(void* pv,
                                        REFIID riid, LPVOID* ppv) {
      ATLASSERT(*ppv == NULL);
      return (pv == NULL) ? TI: :Createlnstance(NULL, riid, ppv)
                          : T2::Createlnstance(pv, riid, ppv);
  }
};
```

注意 CComCreator2 包含两个其他的创建者类型作为模板参数。CComCreator2 所做的仅仅是检查 pUnkOuter 是否为空，并且把调用传递给其他两个创建者之一。所以，如果我们使用 CComObject 和 CComAggObject 而不是 CComPolyObject，那么可以这样做：

```
class CPenguin : ... {
publ i c:
   ...
   typedef CComCreator2< CComCreator< CComObject<CPenguin> >,
                         CComCreator< CComAggObject<CPenguin> > >
              _CreatorC1 ass;

};
```

当然，这种方法的美妙之处在于，因为所有的创建者具有同一个函数 —— CreateInstance，并且通过同名的类型定义暴露给外界 —— _CreatorClass，所以如果类的设计者改变了创建 CPenguin 的想法，那么创建 CPenguin 的服务器代码不需要作任何改变。

📖 CComFailCreator

我们可能想把创建方式改为仅支持独立激活或者仅支持聚合激活，而不是两种方式都支持。为了做到这一点，我们需要一个能返回错误码的特殊创建者以代替传递给 CComCreator2 的两个创建者模板参数之一。这就是 CComFailCreator 的目的：

```
template <HRESULT hr> class CComFailCreator {
public:
 static HRESULT WINAPI CreateInstance(void*, REFIID, LPVOID*)
    { return hr; }
};
```

如果我们只支持独立激活，那么可以这样使用 CComFailCreator：

Ifwe'd like standalone activation only, we can use CComFai 1 C reato r like this:

```
cqass CPenguin : ... {
public:
 ...
 typedef CComCreator2< CComCreator< CComObject<CPenguin> >,
                       CComFailCreator<CLASS_E_NOAGGREGATION> >
             _Creato rC1 ass;

 };
```

如果我们只支持聚合激活，那么可以这样使用 CComFailCreator：

```
If we'd like aggregate activation only, we can use CComFai1Creator like this:

class CPenguin : ... {
public:
...
typedef CComCreator2< CComFailCreator<E_FATL>,
                      CComCreator< CComAggObject<CPenguin> > >
             _CreatorC1 ass;
};
```

📖 方便的宏

为了方便使用，ATL **提供了下面的宏来代替每个类中手工指定的**_CreatorClass **类型定**

义。

```
#define DECLARE_POLY_AGGREGATABLE(x) public:\
  typedef CComCreator< CComPolyObject< x >> _CreatorClass;

#define DECLARE_AGGREGATABLE(x) public:\
  typedef CComCreator2< CComCreator< CComObject< x < <,
                          CComCreator< CComAggObject< x > > >
          _CreatorClass;

#define DECLARE_NOT_AGGREGATABLE(x) public:\
  typedef CComCreator2< CComCreator< CComObject< x > >,
                       CComFailCreator<CLASS_E_NOAGGREGATION> >
          _CreatorClass;

#define DECLARE_ONLY_AGGREGATABLE(x) public:\
  typedef CComCreator2< CComFailCreator<E_FAIL>,
                         CComCreator< CComAggObject< x > > >
          _CreatorClass;
```

通过使用这些宏，我们可以像下面那样把 CPenguin 声明为既可以被独立激活又可以被聚合激活：

```
class CPenguin: ... {
public:
   ....
   DECLARE_AGGREGATABLE(CPenguin)
};
```

表 3.3 总结了创建者用来从我们的类派生出来的类。

表 3.3.　　创建者类型定义宏

宏	独　立	聚　合
DECLARE_AGGREGATABLE	CComObject	CComAggObject
DECLARE_NOT_AGGREGATABLE	CComObject	N/A
DECLARE_ONLY_AGGREGATABLE	N/A	CComAggObject
DECLARE_POLY_AGGREGATABLE	CComPolyObject	CComPolyObject

3.7.4 私有的初始化

因为创建者遵循基于 ATL 的对象所使用的多步骤构造过程，所以它们是很方便的。但是创建者仅仅返回一个接口指针，而不是指向实现类的指针（也就是说，是 IBird*而不是 CPenguin*）。若一个类暴露的公有成员函数或者成员数据不能通过 COM 接口访问，这可能是个问题。作为从前的 C 程序员，我们的第一反应可能是简单地把作为结果的接口指针强制转换成期望的类型：

```
STDMETHODIMP
CAviary::CreatePenguin(BSTR bstrName, long nWingspan, IBird** ppbird) {
 HRESULT hr;
 hr = CPenguin::_CreatorClass::CreateInstance(O,
                                        IID_IBird,
                                        (void**)ppbird);
 if(SUCCEEDED(hr) ) {
    CPenguin* pPenguin = (CPenguin*)(*ppbird); // Resist this instinct!
     pPenguin->Init(bstrName, nWingspan);
 }
  return hr;
}
```

不幸的是，因为 QueryInterface 允许同一 COM 实体的接口在多个 C++对象上实现，或者甚至在多个 COM 对象基础上实现，所以在许多情况下强制转换并不能正常工作。相反，我们应该使用 CComObject、CComAggObject 和 CComPolyObject 的静态成员函数 CreateInstance：

```
static HRESULT WINAPI
CComObject::CreateInstance(CComObject<Base>** pp);

static HRESULT WINAPI
CComAggObject::CreateInstance(IUnknown* puo,
                              CComAggObject<contained>** pp);

static HRESULT WINAPI
CComPolyObject::CreateInstance(Iunknwn* puo,
                               CComPolyObject<contained>** pp);
```

这些静态成员函数并没有使创建者离开 CComObject、CComAggObject 和 CcomPoly Object，但是它们每个都完成了必要的附加工作(也要求调用对象的 FinalConstruct 成员函数)。使用它们的原因是：它们中的每一个都返回一个指向最外层派生类的指针。例如：

```
STDMETHODIMP
CAviary::CreatePenguin(BSTR bstrName, long nWingspan, IBird** ppbird) {
   HRESULT hr;
   CComObject<CPenguin>* pPenguin = 0;
   hr = CComObject<CPenguin>: :CreateInstance(&pPenguin);
   if(SUCCEEDED(hr) ) {
      pPengui n->AddRef();
      pPenguin->Init(bstrName, nWingspan);
      hr = pPenguin->QueryInterface(IID_IBird, (void**)ppbird);
      pPengui n->Rel ease();
  }
   return hr,
}
```

在这种方式下用于创建的类依赖于激活的方式。对于独立激活，使用 CComObject::CreateInstance。对于聚合激活，使用 CComAggObject::CreateInstance。对于既是独立激活又是聚合激活的情形，并且每个实例还要节省一套 vtbl 的开销，应该使用 CComPoly Object::CreateInstance。

3.7.5　栈上的多步骤构造

当创建一个基于 ATL 的 COM 对象的实例时，我们应该使用一个创建者（或者 CComObject 和其他类的静态成员函数 CreateInstance）而不是 C++的操作符 new。但是，如果有一个全局或静态对象，或者一个在栈上分配的对象时，因为我们没有调用 new，所以不能使用创建者。像我们前面所讨论的，ATL 提供了两个用来在堆以外的位置创建实例的类：CComObjectGlobal 和 CComObjectStack。但是，这些类不要求我们手工调用 FinalConstruct（和 FinalRelease），它们会在自己的构造函数和析构函数中完成适当的初始化和关闭工作，像下面 CComObjectGlobal 展示的那样：

```
template <class Base>
class CComObjectGlobal : public Base {
public:
```

```
  typedef Base _BaseClass;
  CComObjectGlobal(void* = NUll)
  { m_hResFinalConstruct = FinalConstruct(); }

  ~CComObjectGlobal()
  { FinalRelease(); }
  ...
  HRESULT m_hResFinalConstruct;
};
```

因为构造函数没有返回值，所以如果对 FinalConstruct 的结果感兴趣的话，我们必须检查公有成员变量 m_hResFinalConstruct 中缓存的结果。

3.8 调试

ATL 提供了几个帮助调试的工具，包括普通的包装类和产生调试输出的分类包装类、产生断言(assertion)的宏和用于一个接口一个接口地追踪 QueryInterface、AddRef 和 Release 的调试输出。当然，在正式版本(release build)中所有这些调试工具将不起作用，以便产生尽可能短、尽可能快的二进制映象。

3.8.1 产生断言

最好的调试技术可能是断言(assertion)。断言允许我们在代码中作出假设，并且在违反这些假设的时候立即告知我们。虽然 ATL 并不真正支持断言，但是它提供了一个 ATLASSERT 宏。其实，它是 Microsoft CRT 宏_ASSERTE 的另一个名字。

```
#ifndef ATLASSERT
#define ATLASSERT(expr) _ASSERTE(expr)
#endif
```

3.8.2 灵活的调试输出

OutputDebugString 是 Win32 下一个等价于 printf 的方便工具，但是它只接受一个字符串参数。我们需要的是输出到调试输出而不是标准输出的 printf，这是 AtlTrace 的目的：

```
void AtlTrace(LPCTSTR lpszFormat, ...);
```

但是，我们并不直接使用 AtlTrace**，相反，**ATL **提供了** ATLTRACE**。这个宏根据**_DEBUG **符号是否被定义，或者扩展为对** AtlTrace **的调用或者扩展为空。下面是它的典型用法：**

```
HRESULT CPenguin:: FinalConstruct() {
```

```
    ATLTRACE( T("%d+%d= %d\n"), 2, 2, 2+2);
}
```

如果我们想使调试输出有更多选择，ATL **提供了第二个跟踪函数** AtlTrace2**，它也有自己的宏 ——** ATLTRACE2**。**

```
void AtlTrac2(DWORD category, UINT level, LPCTSTR 1pszFormat, ...);
```

除了字符串格式和变量参数，AtlTrace2 **还使用一个跟踪类别和跟踪级别。跟踪类别是下面中的一个或者多个：**

```
enum atlTraceFlags {
  // Application-defined categories
  atlTraceUser            = 0x00000001,
  atlTraceUser2           = 0x00000002,
  atlTraceUser3           = 0x00000004,
  atlTraceUser4           = 0x00000008,

  // ATL-defined categories
  atlTraceGeneral         = 0x00000002,
  atlTraceCOM             = 0x00000040,
  atlTraceQI              = 0x00000080,
  atlTraceRegistrar       = 0x00000100,
  atlTraceRefcount        = 0x00000200,
  atlTraceWindowing       = 0x00000400,
  atlTraceControls        = 0x00000800,
  atlTraceHosting         = 0x00001000,
  atlTraceDBClient        = 0x00002000,
  atlTraceDBProvider      = 0x00004000,
  atlTraceSnapin          = 0x00008000,
  atlTraceNotImpl         = 0x00010000,
};
```

跟踪级别是严重程度的度量，0 **表示最严重。**ATL **本身仅使用** 0 **和** 2**，文档建议我们使用的值在** 0 **到** 4 **之间，但是我们可以一直使用到** 4,294,967,295**（虽然可能过于细致而未必有用）。**AtlTrace2 **把（通过参数输入的）类别和级别与服务器范围内的** ATL_TRACE_

CATEGORY 和 ATL_TRACE_LEVEL 定义进行匹配：

```
void AtlTrace2(DWORD category, UINT level, LPCTSER lpszFormat, ...) {
   if (category & ATL_TRACE_CAREGORY && level <= ATL_TRACE_LEVEL) {
     ... // ouput code removed for brevity
   }
}
```

如果 ATL 发现 ATL_TRACE_CATEGORY 和 ATL_TRACE_LEVEL 没有定义，就会使用它自己的缺省值：

```
#ifnedf ATL_TRACE_CATEGORY
#define ATL_REACE_CATEGORY 0xFFFFFFFF     // Output all categories
#endif

#ifndef ATL_TRACE_LEVEL
#define ATL_TRACE_LEVEL 0                 // Output most severe only
#endif
```

典型的用法允许细致地控制调试输出：

```
STDMETHODIMP CPenguin:: Fly() {
   ATLTRACE2 (atl TraceUser,    2, _T("IBi rd:: Fl y\n"));
   ATLTRACE2(atlTraceUser,  42, _T("Hmmm... Penguins can't fly...\n"));
   ATLTRACE2 (atl TraceNotImpl,0, _T("IBi rd:: Fly not implemented !\n"));
    return E_NOTIMPL;
}
```

事实上，因为 ATL 非常频繁地使用类别 atlTraceNotImpl，所以它甚至有一个特殊的宏来表示：

```
#define ATLTRACENOTIMPL(funcname) \
  ATLTRACE2(atlTraceNotImp1, 2, _T("ATL: %s not implemented.\n"), \
    Funcname); \
  Return E_NOTIMPL
 STDMETHOD(SetMoniker)(DWORD, Imoniker*) {
```

```
  ATLTRACENOTIMPL(_T("IOleObjectImpl::SetMoniker"));
}
```

许多 OLE **接口的实现使用了这个宏：**

```
STDMETHOD(SetMoniker)(DWORD, IMoniker*) {
  ATLTRACENOTIMPL(_T("IOleObjectImp::SetMoniker"));
}
```

当我们关注的内容是客户和对象中标准接口实现之间的交互时，我们需要在编译之前把 ATL_TRACE_LEVEL **定义为** 2**。**ATL **的接口实现类大量地使用了这个级别来跟踪对单个的方法调用。**

3.8.3　跟踪对 QueryInterface 的调用

ATL **对** QueryInterface **的实现是一个特别好的调试工具。如果在编译前定义了** _ATL_DEBUG_QI **符号，我们的对象就会输出它们的类名、被查询的接口（如果有的话给出名字** [9]**）以及查询是成功还是失败。这在对客户的接口要求进行逆向工程时非常有用。例如，这里是一个** Internet Explorer（IE）4 **容纳的控制的** _ATL_DEBUG_QI **输出的例子。**

```
CComClassFactory – IUnknown
CComClassFactory – IclassFactory
CPenguin – IUnknown
CPentuin – IOleControl
CPentuin – IClientSecurity – failed
CPenguin – IQuickActivate - failed
CPenguin – IOleObject
CPenguin – IOleobject
CPenguin – IPersistPropertyBag2 – failed
CPenguin – IPersistPropertyBag – failed
CPenguin – IPersistStremInit
CPenguin – IID_IViewObjectEx
CPenguin – IConnectionPointContainer – failed
CPenguin – IActiveScipt – failed
```

[9] **远程接口的接口名是注册表中** HKEY_CLASSES_ROOT\Interface\{IID} **关键字的缺省值。**

```
CPenguin – {6D5140D3-7436-11CE-8934-00AA006009FA} – failed
CPenguin – IOleControl
CPenguin – IOleCommandTarget – failed
CPenguin – IDispatchEx – failed
CPenguin – IDispatch
CPenguin – IOleObject
CPenguin – IOleObject
CPenguin – IRunnableObject – failed
CPenguin – IOeControl
CPenguin – IOleObject
CPenguin – IOleInPlaceObject
CPenguin – IID_IOleInPlaceObjectWindowless
CPenguin – IOleInPlaceActiveObject
CPenguin – IOleControl
CPenguin – IClientSecurity – failed
CPenguin – IClientSecurity - failed
```

3.8.4 跟踪对 AddRef 和 Release 的调用

比 QueryInterface 还有用的调试工具是 AddRef 和 Release。ATL 3.0 引入了一个相当精细的方法来跟踪对单个接口的 AddRef 和 Release 调用。称之为“精细”的原因是，每个基于 ATL 的 C++类只有一个 AddRef 和 Release 实现，它们在最外层的派生类(例如 CComObject)中实现。为了克服这种局限，当定义了_ATL_INTERFACE_DEBUG 的时候，ATL 把通过 QueryInterface 得到的每个新接口封装到另一个实现了某个单独接口的 C++对象中。[10] 每个 *thunk 对象*(夹层对象，即指 ATL 提供的封装对象 —— 译注)记录了真正的接口指针、接口名字和实现接口的类名。除了对象的引用计数之外，thunk 对象还保持了一个有关接口指针的引用计数，thunk 对象的 AddRef 和 Release 实现管理这个引用计数。对 AddRef 和 Release 的任何调用，每个 thunk 对象都准确地知道哪个接口正在被使用，并且把引用计数信息输出到调试输出。例如，下面的控制与 IE4 之间的交互过程和前面的相同，但是使用_ATL_DEBUG_INTERFACE 而不是_ATL_DEBUG_QI：

```
1> CComClassFactory - IUnknown
1> CComClassFactory - IClassFactory
```

[10] 像第 5 章描述的那样，ATL 确实一直是通过为每个对象的 IUnknown*分配同一个 thunk 来遵守 COM 同一性规则。

```
1> CPenguin - IUnknown
1> CPenbuin - IOleControl
0> CPenguin - IOleControl
1> CPenguin - IOleObject
1> CPenguin - IOleObject
0> CPenguin - IOleObject
0> CPenguin - IOleObject
1> CPenguin - IPersistStreamInit
0> CPenguin – IPersistStreamInit
1> CPenguin – IDD_IViewObjectEx
1> CPenguin - IOleControl
0> CPenguin - IOleControl
1> CPenguin - IDispatch
1> CPenguin - IOleObject
1> CPenguin - IOleObject
0> CPenguin - IOleObject
0> CPenguin - IOleObject
1> CPenguin - IOleControl
0> CPenguin - IOleControl
0> CComClassFactory - IClassFactory
1> CPenguin - IOleObject
1> CPenguin - IOleInPlaceObject
1> CPenguin – IID_IOleInPlaceObjectWindowless
1> CPenguin – IOleInPlaceActiveObject
0> CPenguin - IOleInPlaceActiveObject
0> CPenguin - IOleInPlaceObject
0> CPenguin - IOleObject
1> CPenguin - IOleControl
0> CPenguin - IOleControl
```

ATL 维护了一个 thunk 对象列表。这个列表在服务器卸载时用来检测内存泄漏，也就是那些未被客户释放的接口。当使用 _ATL_DEBUG_INTERFACE 时，我们可以在调试输出中检查字符串 INTERFACE LEAK，如果找到的话，则表示有人没有正确地管理接口引用。例如：

```
INTERFACE LEAK: PefCount = 1, MaxRefCount = 1, {Allocation = 4}
```

```
CPenguin – IsnappyDresser
```

这个通知中最有用的部分是 Allocation **数。通过在服务器启动时设置** CComModule **的** m_nIndexBreakAt **成员可以获得被泄漏的接口，此后我们可以利用它进行跟踪。例如：**

```
extern "C"
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID) {
   if (dwReason == DLL_PROCESS_ATTACH) {
   _Module. Init(ObjectMap, hlnstance, &LIBID_PISVRLib);
   DisableThreadLibraryCalls(hlnstance);

   // Track down interface leak
   _Module.m_nlndexBreakAt = 4;
}
else if (dwReason == DLL_PROCESS_DETACH)
   _Module.Term();
return TRUE;     // ok
}
```

当接口 thunk **被分配时，**_Module **会调用** DebugBreak**，把控制权交给调试器并且允许我们检查调用栈以及堵上泄漏。**

📖 _ATL_DEBUG_REFCOUNT

ATL 3.0 **之前的版本使用**_ATL_DEBUG_REFCOUNT **符号仅能跟踪** ATL **的** IXxxImpl **类的引用计数。虽然为了向后兼容** ATL **仍然支持**_ATL_DEBUG_REFCOUNT**，但是因为**_ATL_INTERFACE_DEBUG **要普遍得多，所以它已经替代了**_ATL_DEBUG_REFCOUNT**。**

```
#ifdef _ATL_DEBUG_REFCOUNT
#ifndef _ATL_DEBUG_INTERFACES
#define _ATL_DEBUG_INTERFACES
#endif
#endif
```

3.9　总结

ATL 提供了一种分层实现 IUnknown 的方法。CComXxxThreadModel 类表示的最上层，为寄生在 STA 和 MTA 中的对象所要求的同步提供了辅助函数和类型定义。第二层是 CComObjectRootEx，它利用线程模型类来支持恰好线程安全的 AddRef 和 Release 实现以及对象级的锁定。它的基类 —— CComObjectRootBase，利用我们的类提供的接口映射表提供了一个表驱动的 QueryInterface 实现。我们的类从 CComObjectRootEx 和任意数量的接口派生，并且提供了接口成员函数的实现。最后一层由 CComObject 或者其他类似的类提供，它们根据对象的生命周期和实体身份的要求提供 QueryInterface、AddRef 和 Release 的实现。

为了允许每个类都可以定义自己的生命周期和实体身份要求，每个类定义自己的 _CreatorClass 来定义适当的创建者。创建者负责正确地创建基于 ATL 的类的实例，并且代替 C++操作符 new 的使用。

最后，ATL 提供了许多调试工具用来调试我们的对象，包括追踪、接口使用和泄漏追踪。