2 ATL 智能类型

📖 String，BSTR，VARIANT 和接口指针

COM 除了 C 语言中的数字类型之外还有一些其他的数据类型。其中的三个是正文字符串（特别是 BSTR 形式）、VARIANT 数据类型和接口指针。ATL 针对这些数据类型提供了有用的类来封装它们以及它们的特性。

字符串引入了几种不同的字符集。COM 组件经常需要使用多种字符集并且偶尔需要从一种字符集转换到另一种。ATL 提供了一些字符串转换宏（从 MFC 源代码继承而来），例如：A2W、OLE2CT，等等。这些宏在必要时完成从一种字符集到另一种字符集的转换，不必要时什么也不做。

CComBSTR 类是一个智能字符串类。这个类根据 BSTR 字符串的语义正确地申请、复制和释放一个字符串。CComBSTR 的实例能够在大多数(但不是全部)使用 BSTR 的地方使用。

CComVariant 类是一个智能 VARIANT 类。这个类实现了 COM VARIANT 数据类型特殊的初始化、复制和清除语义。CComVariant 的实例能够在大多数(但不是全部)使用 VARIANT 的地方使用。

CComPtr 和 CComQIPtr 是智能指针类。智能指针类的对象，其表现类似于指针，明确地说是一个带有附加语义的指针。例如：对于智能接口指针，附加的语义是当智能指针离开作用域(包括像异常处理这种不寻常的情况)时，析构函数会释放接口指针。

2.1 字符串数据类型、转换宏和辅助函数

2.1.1 回顾正文数据类型

在 C++编程中处理正文数据类型相当痛苦。主要问题是，不只是一个正文数据类型，

而是有太多的正文数据类型。在这里我使用术语*正文数据类型*（*text data type*）指一般意义上的字符数组。通常，除了把字符数组当作正文字符串之外，不同的操作系统和编程语言会在字符数组之上引入其他的语义（例如，以 NUL 字符作为结尾或者增加一个长度前缀）。

当选择一个正文数据类型时，我们需要做出几个决定。第一，数组由什么类型的字符组成。当我们向操作系统传递字符串（例如一个文件名）时，有些操作系统要求我们使用 ANSI 字符。另一些操作系统更愿意我们使用 Unicode 字符，但是也能接受 ANSI 字符。其他操作系统要求我们使用 EBCDIC 字符。一些陌生的字符集也在使用，诸如多/双字节字符集（Multi/Double Byte Character Set，MBCS/DBCS），对它们的细节本书不作大量讨论。

第二，编写程序时我们使用什么字符集。源代码使用的字符集并不一定要与运行程序的操作系统首选的字符集相同。当然，使用相同的字符集会更方便。但是，程序和操作系统可以使用不同的字符集。我们必须“简单地”转换所有送往和来自操作系统的正文字符串。

第三，决定正文字符串的长度。一些语言（诸如 C 和 C++）和一些操作系统（诸如 Windows 9x/NT 和 UNIX）使用结束字符 NUL 来定义正文字符串的结束。其他语言更愿意使用一个明确的、指定正文字符串中的字符个数的长度前缀，诸如 Microsoft 的 Visual Basic 解释器、Microsoft 的 Java 虚拟机和 Pascal。

第四，在实际工作中，正文字符串带来的资源管理问题。正文字符串的长度一般是不同的，因此，为了有效地使用内存，正文字符串经常是动态分配的。当然，这意味着最后必须释放正文字符串。资源管理引出正文字符串所有者的概念。字符串的所有者，并且只有所有者，才释放字符串而且仅释放一次。当我们在位于异种计算机上的组件之间传递正文字符串时，所有权变得十分重要。

两个 COM 对象可以驻留在运行不同操作系统的不同计算机上，而且这两个操作系统偏爱不同字符集的正文字符串。例如，我们可以在 Visual Basic 中编写一个 COM 对象并且运行在 Window NT 操作系统上。我们可能会把一个正文字符串传递给另一个使用 C++编写的、运行在 IBM 主机上的 COM 对象。很显然，我们需要一些标准的正文数据类型，并且不同环境下所有的 COM 对象都能够理解这些类型。

COM 使用 OLECHAR 字符数据类型。一个 COM 正文字符串是一个以 NUL 字符作为结尾的 OLECHAR 字符数组，并且指向这样一个字符串的指针为一个 LPOLECHAR。[1] 作为一个原则，传递给 COM 接口方法的正文字符串参数必须是 LPOLECHAR 类型。如果一个方法并不改变这个字符串，那么这个参数的类型应该是 LPCOLECHAR 类型，也就是指向 OLECHAR 数组的常量指针。

通常情况下(但并不总是这样)，OLECHAR 类型与我们编写代码时使用的字符不会相

[1] 注意，一个操作系统（例如 Windows NT）上 OLECHAR 的实际字符数据类型可能不同于另一种不同的操作系统（例如 OS/390）上 OLECHAR 的字符数据类型。COM 远程基础设施会在列集(marshaling)和散集(unmarshaling)时完成任何必要的字符集转换。因此，COM 组件总是收到它所期望的 OLECHAR 格式的正文。

同。有时候(但并不总是这样)，OLECHAR 类型也不同于我们向操作系统传递的正文字符串。这意味着，有时候我们需要*根据环境*，把正文字符串从一种字符集转换到另一种，但有时并不需要。

不幸的是，编译选项的改变（例如，一个 Window NT Unicode build 或者一个 Window CE build）可能会改变这个环境，导致以前不需要转换字符串的代码现在却需要转换，或者相反。我们当然不希望每次改变一个编译选项就重写所有的字符串操作代码。为此，ATL 提供了一些字符串转换宏，这些宏把正文字符串从一种字符集转换到另一种，并且对请求转换的环境很敏感。

🕮 Windows 字符数据类型

现在，让我们特别关注 Windows 平台。基于 Windows 的 COM 组件一般混合使用四种正文数据类型。

- **Unicode**：Unicode 规范把字符表示为“宽字符（wide-character）”，即 16 位的多语言字符编码。Windows NT 操作系统内部使用 Unicode 字符集。当前全球计算机使用的所有字符都可以在 Unicode 中被唯一地表示出来，包括技术符号和用于出版的特殊字符。固定的字符大小(即 16 位)简化了使用国际字符集的编程工作。在 C/C++中，使用 wchar_t 数组来表示宽字符串；指向这样一个字符串的指针为 wchar_t*指针。
- **MBCS/DBCS**：多字节字符集(MBCS，Multi-Byte Character Set)是一种混合长度的字符集，其中有些字符由多个字节组成。Windows 9x 操作系统一般情况下使用 MBCS 来表示字符。DBCS（Double-Byte Character Set，双字节字符集）是多字节字符集的特殊类型。它包含的字符有些由一个字节组成，有些由两个字节组成来表示一个指定地区的符号，例如日文、中文和韩文。

在 C/C++中，我们把 MBCS/DBCS 字符串表示为 unsigned char 数组；指向这种字符串的指针是 unsigned char*指针。有时字符的长度是一个 unsigned char，有时比一个要多。这为字符串的处理增加了特殊的负担，特别是考虑字符串中的回退时。在 Visual C++中，MBCS 总是意味着 DBCS，它不支持超过两个字节的字符集。

- **ANSI**：我们可以只用八个比特(bit)来表示英语和很多西欧语言使用的所有字符。Windows 使用 MBCS 的退化情况来支持这些语言，被称为 Microsoft Windows ANSI 字符集，该字符集中不出现多字节字符。Microsoft Windows ANSI 字符集最初基于一个 ANSI 标准的草案，本质上是 ISO 8859/x 再加上其他一些字符。

ANSI 字符集映射字母和数字的方式与 ASCII 相同。但是 ANSI 不支持控制字符并且映射了许多符号，这些符号包括标准 ASCII 中没有出现的重音字母（accented letter）。所有的 Windows 字体都是在 ANSI 字符集上被定义的。为保持对称，它也被称为 SBCS（Single-Byte Character Set，单字节字符集）。

在 C/C++中，ANSI 字符串被表示为 char 数组 指向这样一个字符串的指针为一个 char*

指针。其字符永远是一个 char 的长度。缺省情况下，Visual C++中的 char 为 signed char。因为 MBCS 字符是无符号的而 ANSI 字符一般是有符号的，所以使用 ANSI 字符的表达式取值与使用 MBCS 字符的不同。

- TCHAR/_TCHAR：一个由 Microsoft 定义的通用正文数据类型，我们可以使用不同的编译选项分别把它们映射到 Unicode 字符、MBCS 字符或者 ANSI 字符。我们使用这种字符类型来编写通用的代码，这些代码可以针对上述三种字符集中的任何一种进行编译。这简化了适应国际市场的代码开发工作。C 运行库定义了_TCHAR 类型，Windows 操作系统定义了 TCHAR 类型。它们是相同的。

  注意： tchar.h 是一个由 Microsoft 定义的 C 运行库头文件，它定义了通用正文数据类型_TCHAR。ANSI 的 C/C++编译器要求实现者定义的名字以下划线作为前缀。当没有定义预编译符号__STDC__时（这是 Visual C++的缺省设置），表示不需要遵从 ANSI 标准。在这种情况下，如果 TCHAR 尚未定义，那么 tchar.h 头文件把 TCHAR 符号定义为通用正文数据类型的另一个别名。winnt.h 是一个 Microsoft 指定的 Win32 操作系统的头文件，它定义了通用正文数据类型 TCHAR。这个头文件是操作系统指定的，所以符号的名字不需要以下划线作为前缀。

Win32 API 和字符串

每个需要字符串的 Win32 API 有两个版本：一个需要 Unicode 形式的参数，另一个需要 MBCS 形式的参数。在使用非 MBCS 版本的 Windows 上，一个 API 的 MBCS 版本需要 ANSI 形式的参数。例如 API SetWindowText 并不真的存在。事实上是两个函数：接受 Unicode 字符串形式参数的 SetWindowTextW 和接受 MBCS/ANSI 字符串形式参数的 SetWindowTextA。

Windows NT 操作系统的内部只使用 Unicode 字符串。因此，当我们在 Windows NT 上调用 SetWindowsTextA，这个函数把指定的字符串翻译成 Unicode，然后再调用 SetWindowTextW。Windows 9x 操作系统不支持 Unicode。函数 SetWindowTextA 在 Windows 9x 操作系统上能够正常工作，但是 SetWindowTextW 则返回一个错误。

这使我们难于作出选择。我们可以编写一个性能最优的组件，使用 Unicode 字符串运行在 Windows NT 上，但是却不能运行在 Windows 9x 上。我们可以使用 MBCS/ANSI 字符串编写一个更加通用的组件，能够运行在两种操作系统上，但是在 Windows NT 上不是最优的。我们可以在编码时回避这个问题，直到编译时刻再决定支持什么样的字符集。

一个小小的编码约定再加上一些预编译技巧，使得我们编写代码时好像只有一个被称为 SetWindowText 的 API 函数，它要求一个 TCHAR 字符串参数。在编译时刻我们再指定组件的类型。例如，我们编写代码调用 SetWindowText，并且指定一个 TCHAR 的缓冲区。当编译 Unicode 组件时，我们实际上调用了 SetWindowTextW，并且参数的形式是一个 wchar_t 缓冲区。当编译 MBCS/ANSI 组件时，我们实际上调用了 SetWindowTextA，并且参数形式是一个 char 缓冲区。

当编写基于 Windows 的 COM 组件时，我们一般应该使用 TCHAR 字符类型来表示组

件内部使用的字符。此外，对所有与操作系统交互的字符也使用 TCHAR。相似地，我们应该使用 TEXT 或者__TEXT 宏来括起每个文字或者字符串。tchar.h 定义了功能等价的宏_T、__T 和_TEXT，这些宏都把字符或者字符串编译成通用正文字符或者文字。winnt.h 也定义了功能上等价的宏 TEXT 和__TEXT，它们与_T，__T，和_TEXT 的效果完全一样。这五种方法完成了完全相同的事情。本章中的例子使用的是__TEXT，因为它定义在 winnt.h 中。我更喜欢使用_T，因为它使我的源代码更短一些。

如果是在不知道操作系统的情况下进行编码，那么我们会倾向于包含 tchar.h 头文件，并使用_TCHAR 通用正文数据类型，因为这样稍稍降低了与 Windows 操作系统捆绑的紧密程度。但是，我们讨论的是“在编译时刻、为特定的 Windows 操作系统版本使用最优的正文处理方法建立组件”。这就是说我们应该使用在 winnt.h 中定义的 TCHAR。另外，TCHAR 不象_TCHAR 那么刺眼。它的录入次数较少。大多数代码已经隐式地包含了 winnt.h 头文件，但是我们必须显式地包含 tchar.h。由于使用 TCHAR 有各种各样充足的理由，所以这本书中的例子使用 TCHAR 作为通用正文数据类型。

这使得我们能够为不同的市场或者由于性能的原因编译特定版本的组件。这些类型和宏被定义在 winnt.h 头文件中。

当操作 TCHAR 字符的字符串时，我们也必须使用不同的字符串运行库函数集。我们熟悉的函数如 strlen、strcpy 和其他一些函数只能操作 char 字符。我们不太熟悉的函数如 wcslen、wcscpy 等等函数在 wchar_t 字符上工作。此外，我们完全陌生的函数如_mbslen、_mbscpy 等等函数可以操作多字节字符。因为 TCHAR 有时是 wchar_t，有时是 char 拥有(char-holding)的 ANSI 字符，还有的时候是 char 拥有的（名义上是 unsigned）的多字节字符，所以我们需要一个与 TCHAR 字符共同工作的等价的运行库函数集。

tchar.h 头文件为字符串处理函数定义了一些有用的通用正文映射函数。这些函数接受 TCHAR 参数，所以所有这些函数的名字都使用前缀_tcs（_t 字符集）。例如，_tcslen 等价于 C 运行库中的函数 strlen。_tcslen 函数接受 TCHAR 字符，而 strlen 接受 char 字符。

使用预处理器控制通用正文映射

两个预处理器符号和两个宏控制从 TCHAR 数据类型到应用程序使用的内在字符类型之间的映射。

- **UNICODE/_UNICODE**：Windows 操作系统 API 的头文件使用 UNICODE 预处理器符号。C/C++运行库的头文件使用_UNICODE 预处理器符号。一般来说，我们或者同时定义这两个符号或者都不定义。如果编译时定义了_UNICODE 符号，那么 tchar.h 会把所有的 TCHAR 字符映射成 wchar_t 字符。_T、__T 和_TEXT 宏会为每个文字和字符串加上前缀大写的 L(分别创建一个 Unicode 字符或者字符串)。如果编译时定义了符号 UNICODE，则 winnt.h 会把所有的 TCHAR 字符映射成 wchar_t 字符。TEXT 和__TEXT 宏会为每个字符和字符串加上前缀大写的 L（分别创建一个 Unicode 字符或者字符串)。

- **_MBCS：如果编译时定义了**_MBCS **符号，则所有的** TCHAR **字符会被映射成** char **字符，并且预处理器会去掉所有的**_T **和**_TEXT **宏的变种，不改变字符或者字符串（分别创建一个** MBCS **字符或者字符串）。**
- **若编译时未定义任何预处理器符号，则所有的** TCHAR **字符会被映射为** char **字符，并且预处理器会去掉所有的**_T **和**_TEXT **宏的变种，不改变字符或者字符串（分别创建一个** ANSI **字符或者字符串）。**

我们使用通用正文数据类型和函数来编写与通用正文兼容（generic-text-compatible**）的代码。下面的通用正文代码首先反转一个字符串中字符的顺序，然后又添加一个字符串：**

```
TCHAR *reversedString, *sourceString, *completeString;
reversedString = _tcsrev (sourceString);
completeString = _testat (reversedString, __TEXT("suffix"));
```

若未定义任何预处理器符号进行编译，则预处理器产生如下输出：

```
char *reversedString, *sourceString, *completeString;
reversedString = _strrev (sourceString);
completeString = strcat (reversedString, "suffix");
```

若定义了预处理器符号_UNICODE **之后进行编译，则预处理器产生如下输出：**

```
_wchar_t *reversedString, *sourceString, *completeString;
reversedString = _wcsrev (sourceString);
completeString = wcscat (reversedString, L"suffix");
```

若定义了预处理器符号_MBCS **之后进行编译，则预处理器产生如下输出：**

```
char *reversedString, *sourceString, *completeString;
reversedString = _mbsrev (sourceString);
completeString = _mbscat (reversedString, "suffix");
```

📖 COM 字符数据类型

COM **使用两种字符类型。**

- OLECHAR：**在编译源代码的目标操作系统上** COM **使用的字符类型。对于** Win32 **操作系统，这是** wchar_t **字符类型。**

注意：事实上，通过定义预处理器符号 OLE2ANSI**，我们可以把** Win32 OLECHAR **数据类型从**

缺省的 wchar_t（COM 内部使用的）改变为 char。这可以让我们假装 COM 使用的是 ANSI。MFC 曾经使用了这个特性。不要走向黑暗的一面，Luke（星球大战台词 —— 译者注）。使用这项“便利”特性的同时，也要求我们随后使用 OLE2ANSI thunking 库来链接组件。每当 API 和 COM 接口方法被调用时，这个库把所有 ANSI 字符串参数转换回 wchar_t。相应地，它把输出字符串从 wchar_t 转换成 char。在许多情况下这样做增加了不必要的开销。Microsoft 重写了这部分代码，不再使用 OLE2ANSI，而是在必要时使用字符串转换宏，从而 MFC 的性能约提高了 10%。

- 对于 Win16 操作系统，这是 char 字符类型。对于 MacOS，这是 char 字符类型。对于 Solaris OS，这是 wchar_t 字符类型。对于至今未知的操作系统，没有人知道是什么类型。让我们假设只是有一个叫做 OLECHAR 的抽象数据类型。COM 使用它。不要假设它映射到任何特定的底层数据类型。
- BSTR：某些 COM 组件使用的特殊数据类型。BSTR 是一个包含长度前缀的 OLECHAR 数组，它有许多特殊的语义。

现在让我们把事情变得复杂一些。我们希望编写能够在编译时刻选择字符类型的代码。为此，我们在内部必须严格地使用 TCHAR。我们也希望调用 COM 方法时传递给它同样的字符串。我们必须根据方法的原型传递给它 OLECHAR 字符串或者 BSTR 字符串。根据编译选项，组件使用字符串的格式可能正确也可能不正确。这是超级宏（supermacro）的工作。

2.1.2　ATL 字符串转换宏

ATL 提供了一些字符串转换宏，必要时它们在前面描述的各种字符类型之间进行转换。当编译选项使得源字符类型与目标字符类型相同时，这些宏并不执行转换，事实上什么也没干。

宏的名字使用了各种字符数据类型的缩写。

- T 表示指向 Win32 TCHAR 字符类型的指针 —— LPTSTR 参数。
- W 表示指向 Unicode wchar_t 字符类型的指针 —— LPWSTR 参数。
- A 表示指向 MBCS/ANSI char 字符类型的指针 —— LPSTR 参数。
- OLE 表示指向 COM OLECHAR 字符类型的指针 —— LPOLESTR 参数。
- C 表示 C/C++中的 const 修饰符。

所有宏的名字使用“<源类型缩写>2<目标类型缩写>”的形式；例如，A2W 宏把一个 LPSTR 转换成一个 LPWSTR。当宏的名字中出现 C 时，那么在它后面的类型缩写前面加上 const 修饰符；例如，宏 T2COLE 把 LPTSTR 转换成 LPCOLESTR。

宏的真正行为依赖于我们定义的预处理器符号（表 2.1）。表 2.2 列出了 ATL 字符串转换宏。

表 2.1 字符集预处理器符号

定义的预处理器符号	T 成为	OLE 成为
None	A	W
_UNICODE	W	W
OLE2ANSI	A	A
_UNICODE 和 OLE2ANSI	W	A

表 2.2 ATL 字符串转换宏

A2BSTR	A2W	OLE2CW	T2COLE	W2CA
A2COLE	A2WBSTR	OLE2T	T2CW	W2COLE
A2CT	OLE2A	OLE2W	T2OLE	W2CT
A2CW	OLE2BSTR	T2A	T2W	W2OLE
A2OLE	OLE2CA	T2BSTR	W2A	W2T
A2T	OLE2CT	T2CA	W2BSTR	

所有的宏接受一个指向源字符集中字符串的指针作为参数。每个宏的行为像一个函数调用，它们会返回一个指向目标字符集中字符串的指针。当源字符集和目标字符集相同时，宏只是把指定的参数作为目标返回。

当源字符集和目标字符集不同并且*目标类型不是 BSTR* 时，这些宏使用运行库函数 _alloca。_alloca 函数在栈上分配内存（也就是说，增长栈指针）。转换宏在栈上为目标字符串分配内存是为了当（使用转换宏的）函数返回时自动地回收内存。但是，了解内存发生的变化是非常重要的，这样我们就不会在循环中使用字符串转换宏。在一个循环中反复地调用这些转换宏会使栈变得越来越大。最后，我们会耗尽所有的栈空间。

转换宏使用了局部变量。我们必须在使用转换宏的函数开始处指定宏 USES_CONVERSION(只需一次)，以便分配这些变量。下面的代码把一个 TCHAR 字符串转换成 OLECHAR 字符串，并且调用一个 COM 方法，这个 COM 方法需要一个指向 OLECHAR 字符串的常量指针。我们不需要释放 OLECHAR 字符串，因为在函数返回时它会被自动释放。

```
STDMETHODIMP put_Name (/* [in] */ const OLECHAR* pName);
void SetName (LPTSTR lpsz)
{
    USES_CONVERSION;
...
    pObj->put_Name (T2COLE(lpsz));
```

```
}
```

当源字符集和目标字符集不同并且*目标类型是 BSTR* 时，这些宏使用 SysAllocString 和 SysAllocStringLen 函数来分配目标字符串。我们必须使用 SysFreeString 显式地释放这个 BSTR。下面的代码把一个 TCHAR 字符串转换成 BSTR，调用一个需要 BSTR 字符串的 COM 方法，然后释放该 BSTR。

```
void SetName (LPTSTR lpsz)
{
    USES_CONVERSION;
...
    BSTR bstr = T2BSTR(lpsz);
    pObj->put_Name (bstr);
     ::SysFreeString (bstr) ;
}
```

2.1.3　ATL 字符串辅助函数

有时我们想复制一个 OLECHAR 字符组成的字符串。我们碰巧知道 OLECHAR 字符在 Win32 操作系统上是宽字符。当编写一个 Win32 版本的组件时，我们可能会调用 Win32 操作系统中复制宽字符的函数 lstrcpyW。不幸的是，虽然支持 Unicode 的 Windows NT 实现了 lstrcpyW，但 Window 95 没有实现这个函数。因此，使用 lstrcpyW API 的组件在 Windows 95 上不能正常工作。

我们可以使用 ATL 字符串辅助函数 ocscpy 来复制一个 OLECHAR 字符串，而不是使用 lstrcpyW。它在 Windows NT 和 Windows 95 上都能正常工作。还有一个 ATL 字符串辅助函数 ocslen 返回 OLECHAR 字符串的长度。虽然 ocslen 替代的函数 lstrlenW 在两种操作系统上都能工作，但是为了对称起见我们也使用 ocslen。

```
OLECHAR* ocscpy(LPOLESTR dest, LPCOLESTR src);
Size_t ocslen(LPCOLESTR s);
```

类似地，Win32 操作系统函数 CharNextW 不能用于 Windows 95 操作系统，所以 ATL 提供了字符串辅助函数 CharNextO。这个函数把 OLECHAR*加上一个字符并且返回指向下一个字符的指针。当遇到 NUL 终止符时指针不再增加。

```
LPOLESTR CharNextO(LPCOLESTR 1p);
```

2.2 COM 字符串数据类型——BSTR

COM 是一个语言中立、硬件结构中立的模型。因此它需要一个语言中立、硬件结构中立的正文数据类型。不幸的是，没有一个正文数据类型普遍适用于所有的硬件和语言。有些平台使用由 8 位字符组成的 ASCII/ANSI 字符集。有些使用另外一种 8 位字符组成的 EBCDIC 字符集。有些使用由 8 位和 16 位字符混合组成的 MBCS（Multi-Byte Character Set）。有些使用由 16 位字符组成的 Unicode 字符集。还有其他一些不常用的字符集也在使用之中。

COM 定义了一种通用正文数据类型 —— OLECHAR，它被 COM 用来在特定的平台上表示正文数据。在大多数平台上，包括所有 32 位 Windows 在内，OLECHAR 数据类型被定义（typedef）成 wchar_t 数据类型。也就是说，在大多数平台上，COM 正文数据类型等价于 C/C++中的包含 Unicode 字符的宽字符数据类型。在某些平台上，例如 16 位的 Windows 操作系统和 Macintosh OS，OLECHAR 被定义成标准 C 中包含 ANSI 字符的 char 数据类型。一般来说，我们会把 COM 接口中使用的所有字符串参数定义成 OLECHAR*形式的参数。

COM 也定义了另一种正文数据类型叫做 BSTR。BSTR 是一个包含长度前缀的 OLECHAR 字符串。大多数解释环境出于性能考虑更喜欢包含长度前缀的字符串。例如，一个包含长度前缀的字符串不需要消耗时间扫描 NUL 终止符以决定字符串的长度。其实 NUL 字符结尾的字符串是一个语言相关的概念，最初仅适用于 C/C++语言。Microsoft Visual Basic 解释器、Microsoft Java 虚拟机和大多数脚本语言，诸如 VBScript 和 JScript，内部都使用 BSTR 来表示字符串。

因此，当我们向 C/C++组件定义的接口方法的参数传递一个字符串，或者从方法的参数接收一个字符串时，我们经常会用到 OLECHAR*数据类型。但是如果我们需要使用一个由其他语言定义的接口，那么字符串参数经常是 BSTR 数据类型。BSTR 数据类型有许多语义很少被文档提及，这使得 C++开发人员使用 BSTR 既单调乏味，又容易出错。

BSTR 有以下属性：

- BSTR 是一个指向包含长度前缀的 OLECHAR 字符数组的指针。
- BSTR 是一个指针数据类型。它指向数组的第一个字符。长度前缀以整数的形式恰好存储在数组的第一个字符之前。
- 字符数组以 NUL 字符作为结尾。
- 长度前缀以字节为单位而不是字符，并且不包括 NUL 终止符。
- 字符数组内部可以包含嵌入的 NUL 字符。
- 它必须使用 SysAllocString 和 SysFreeString 函数族来分配和释放。
- NULL BSTR 指针意味着一个空字符串。
- BSTR 没有引用计数；因此相同字符串内容的两个引用必须指向不同的 BSTR。换句话说，复制 BSTR 意味着制作字符串的一个拷贝，而不是简单地复制指针。

2.3　CComBSTR 类

CComBSTR 是一个 ATL 工具类，它是对 COM 字符串数据类型 —— BSTR 的一个有效封装。文件 atlbase.h 包含了 CComBSTR 类的定义。这个类维护的唯一状态是一个 BSTR 类型的公有成员变量 m_str。

```
//////////////////////////////////////////////////////
// CComBSTR

class CComBSTR
{
public:
   BSTR m_str;
...
};
```

2.3.1　构造函数和析构函数

CComBSTR 对象有 8 个构造函数。缺省的构造函数简单地把变量 m_str 初始化为 NULL，等价于一个表示空字符串的 BSTR。析构函数调用 SysFreeString 释放包含在 m_str 变量中的任何 BSTR。SysFreeString 的文档明确地说明当输入参数为 NULL 时函数只是简单地返回，所以在空对象上运行析构函数没有任何问题。

```
CComBSTR() { m_str = NULL; }
~CComBSTR() { ::SysFreeString(m_str);}
```

在本节的后面我们将会看到，CComBSTR 类提供了许多便利的函数。但是，迫使我们使用这个类最重要的原因之一是，析构函数会在适当的时候释放内部的 BSTR，所以我们不需要显式地释放 BSTR。当定位一个异常处理块时，在栈结构回收期间释放 BSTR 非常地方便。

从一个以 NUL 字符结尾的 OLECHAR 字符数组（或者是更广为人知的 LPCOLESTR）来初始化一个 CComBSTR 对象可能是使用最多的构造函数。

```
CcomBSTR(LPCOLESTR pSrc) { m_str = ::SysAllocString(pSrc);}
```

下面的代码调用了这个构造函数：

```
CComBSTR strl (OLESTR ("This is a string of OLECHARs")) ;
```

这个构造函数不停地复制字符直至遇到 NUL 终止符。当我们想少复制一些字符，例如只取字符串的前缀部分，或者我们想复制内嵌 NUL 字符的字符串时，我们必须明确地指定复制字符的个数。在这种情况下，可以使用下面的构造函数：

```
CComBSTR(int nSize, LPCOLESTR sz) { m_str = ::SysAllocStringLen
  (sz, nSize); }
```

这个构造函数创建一个由 nSize 指定大小的 BSTR，从 sz 复制包括任何内嵌 NUL 字符在内的指定数量的字符，然后加上一个 NUL 终止符作为后缀。当 sz 为 NULL 时，SysAllocStringLen 跳过复制步骤，只是创建指定大小的尚未初始化的 BSTR。下面的代码调用了这个构造函数：

```
CComBSTR str2 (16, OLESTR ("This is a string of OLECHARs"));
// str2 contains "This is a string"

CComBSTR str3 (64, (LPCOLESTR) NULL);
// Allocates an uninitialized BSTR with room for 64 characters

CComBSTR str4 (64);
// Allocates an uninitialized BSTR with room for 64 characters
```

在前面的代码中，CComBSTR 类为 str3 例子提供了一个特殊的构造函数，它不需要我们提供 NULL 参数。前面的 str4 例子展示了它的用法。这个构造函数是：

```
CComBSTR(itn nSize) {m_str = ::SysAllocStringLen(NULL,nSize);}
```

关于 BSTR 一个奇怪的语义特点是，NULL 指针是一个空 BSTR 字符串的合法值。例如，Visual Basic 认为 NULL BSTR 等同于指向空字符串的指针，也就是第一个字符是终止符并且长度为 0 的字符串。或者用符号表示，Visual Basic 认为当 p 是一个设置为 NULL 的 BSTR 时，表达式 IF "" = p 的值为“真”。

不幸的是，并不是所有处理 BSTR 的 API 都认识到这个事实。特别是 SysStringLen 函数的文档明确地说明参数必须非空。这意味着为了正确地获得 BSTR 的长度（按照文档），我们必须如下编写代码：

```
UINT length = bstrlnput ? SysStringLen (bstrlnput) : 0 ;
```

如果一个 BSTR 已经被封装进 CComBSTR 对象之中，那么我们可以简单地调用成员函数 Length 来作同样的事情：

```
unsigned int Length() const
  { return (m_str ==NULL) ?0: SysStringLen(m_str);}
```

在 Windows NT 4.0 的 Service Pack 3 上进行的测试表明，事实上 SysStringLen 并不真的在乎参数是否为 NULL。当参数不为 NULL 时，SysStringLen 返回正确的长度。当参数为 NULL 时，SysStringLen 返回 0。但是在 Windows NT 的早期版本，或者其他支持 COM 的操作系统上（Solaris、OS/390，等等），这不见得是对的。在其他操作系统上有可能是以文档描述的方式实现了 SysStringLen。

我们也可以使用下面的复制构造函数(copy constructor)来创建和初始化一个 CComBSTR 对象，这个对象等价于一个已经被初始化的 CComBSTR 对象。

```
CComBSTR(const CComBSTR& src { m_str = src.Copy(); }
```

在下面的代码中，创建 str5 变量时调用了上面的复制构造函数，以便初始化对象：

```
CComBSTR strl (OLESTR("This is a string of OLECHARs")) ;
CComBSTR str5 = strl ;
```

注意上面的复制构造函数调用了源 CComBSTR 对象的 Copy 方法。Copy 方法制作了自己字符串的一份拷贝并且返回这个新的 BSTR。因为 Copy 方法利用现有 BSTR 的长度来申请新的 BSTR，并且复制指定长度的字符串内容，所以 Copy 方法能够正确地复制内嵌 NUL 字符的 BSTR。

```
BSTR Copy() const
  { return ::SysAllocStringLen(m_str,::SysStringLen(m_str)); }
```

有两个构造函数用 LPCSTR 字符串来初始化 CComBSTR。只有一个参数的构造函数需要 NUL 字符结尾的 LPCSTR 字符串。带两个参数的构造函数允许我们指定 LPCSTR 字符串的长度。这两个构造函数在功能上等价于前面讨论的两个接受 LPCOLESTR 参数的构造函数。这两个构造函数接受 ANSI 字符，并且创建一个包含等价字符串（以 OLECHAR 字符的形式）的 BSTR。

```
#ifndef OLE2ANSI
CComBSTR(LPCSTR pSrc) { m_str = A2WBSTR(pSrc);}
CcomBSTR(int nSize, LPCSTR sz) { m_str = A2WBSTR(sz, nSize);}
#endif
```

只有当编译时编译器符号 OLE2ANSI 没有被定义的情况下，这两个构造函数才会出现。当我们定义了编译器符号 OLE2ANSI 时，LPCOLESTR 数据类型等同于 LPCSTR 数据类型，所以这两个构造函数已经被定义了。

最后一个构造函数不成对。它接受一个 GUID 参数并且生成一个包含 GUID 字符串表示的字符串。

```
CComBSTR(REFGUID src) { ... }
```

在建立组件注册字符串时，这个构造函数非常有用。有些场合下我们需要把 GUID 的字符串表示写入注册表。有些代码如下使用这个构造函数：

```
// Define a GUID as a binary constant
static const GUID GUlD_Sample = { Ox8a44ellO, Oxf134, Oxlldl,
      { 0x96, Oxbl, OxBA, OxDB, OxAD, OxBA, OxDB, OxAD } };
// Convert the binary GUID to its string representation
CComBSTR str6 (GUID_Sample) ;
// str6 contains "{8A44EllO-F134-11dl-96B1-BADBADBADBAD}"
```

2.3.2 初始化

CComBSTR 类定义了三个赋值操作符。第一个使用另一个 CComBSTR 对象来初始化当前 CComBSTR 对象。第二个使用一个 LPCOLESTR 指针来初始化一个 CComBSTR 对象。第三个使用 LPCSTR 指针来初始化当前对象。下面的 operator=()方法根据另一个 CComBSTR 对象初始化当前 CComBSTR 对象。

```
CComBSTR& operator=(const CComBSTR& src)
{
   if (m_str != src.m_str) {
      if (m_str)
           ::SysFreeString(m_str);
      m_str = src.Copy();
```

```
    }
    return *this;
}
```

注意这个赋值操作符使用 Copy **方法制作了指定** CComBSTR **实例的一份精确拷贝。下面的代码调用了这个操作符：**

```
CComBSTR strl (OLESTR("This is a string of OLECHARs"));
CComBSTR str7 ;

str7 = strl; // str7 contains "This is a string of OLECHARs"
str7 = str7; // This is a NOP, Assignment operator detects this case
```

第二个 operator=()**方法使用指向** NUL **字符结尾的** LPCOLESTR **指针来初始化一个** CComBSTR **对象。**

```
CComBSTR& operator=(LPCOLESTR pSrc)
{
    ::SysFreeString(m_str);
    m_str = ::SysAllocString(pSrc);
    return *this;
}
```

注意赋值操作符使用 SysAllocString**函数来申请指定的** LPCOLESTR**参数的一份** BSTR **拷贝。下面的代码调用了这个操作符：**

```
CComBSTR str8 ;

str8 : OLESTR ("This is a string of OLECHARs");
```

当处理内嵌 NUL **字符的字符串时，很容易误用赋值操作符。例如，下面的代码演示了如何使用和误用这个方法。**

```
CComBSTR str9 ;
str9 = OLESTR ("This works as expected");
```

```
// BSTR bstrlnput contains "This is part one\Oand here's part two"
CComBSTR strlO ;
strlO = bstrlnput; // strlO now contains "This is part one"!!!
```

第三个 operator=()**方法使用一个指向** NUL **字符结尾字符串的** LPCSTR **指针来初始化一个** CComBSTR **对象。操作符把输入的** ANSI **字符串转换成一个** UNICODE **字符串，然后创建一个包含** UNICODE **字符串的** BSTR**。同样地，当我们定义了（不推荐）预处理器符号** OLE2ANSI **时，**LPCSTR **和** LPCOLESTR **数据类型是等价的，所以这个方法已经被定义，因而不能重定义。**

```
#ifndef OLEZANSI
CComBSTR& operator=(LPCSTR pSrc)
{
   ::SysFreeString(m_str);
   m_str = Azwbstr(pSrc);
   return *this;
}
#endif
```

最后两个初始化方法是两个被称为 LoadString **的重载方法，。**

```
bool LoadString(HINSTANCE hInst, UINT nID);
bool LoadString(UINT nID);
```

第一个方法从 hInst **指定的模块中（使用实例句柄）装载指定的字符串资源** nID**。第二个方法使用全局变量**_pModule **从当前模块中装载指定的字符串资源** nID**。**

全局变量_pModule **指向这个模块的(从** CComModule **派生的)实例变量。当调用** CComModule::Init **方法时我们初始化这个变量。我们一般在** DllMain**（对于** DLL**）或者** WinMain**（对于** EXE**）中调用** CComModule::Init**。注意，即使是在没有全局** CComModule **对象的模块中，其他的** CComBSTR **方法也会工作得很好。但是，使用** LoasString**（**UINT nID**)** **方法不仅要求我们拥有一个**CComModule **全局实例，而且要求这个实例必须已经被初始化。您问我我是怎么知道**...

2.3.3 CComBSTR **操作符**

有四个方法供我们以不同的方式访问 CComBSTR **类封装在内部的** BSTR **字符串。操**

作符 BSTR()方法使得我们在需要“裸”BSTR 指针时可以使用 CComBSTR 对象。当显式地或者隐式地把一个 CComBSTR 对象强制转换成一个 BSTR 对象的时候，我们会调用这个方法。

```
Operator BSTR() const { return m_str;}
```

当我们把 CComBSTR 对象当作参数传递给一个需要 BSTR 的函数时，经常会隐式地调用这个操作符。下面的代码说明了这一点：

```
HRESULT put_Name (/* [in] */ BSTR pNewValue) ;

CComBSTR bstrName = OLESTR ("Frodo Baggins");
pObj->put_Name (bstrName); // Implicit cast to BSTR
```

当我们取一个 CComBSTR 对象的地址的时候，operator&()方法返回内部变量 m_str 的地址。取一个 CComBSTR 对象的地址时要小心。因为 operator&()方法返回的是内部变量 m_str 的地址，我们有可能在改写这个内部变量之前没有释放原来的字符串。这将导致内存泄漏。

```
SBTR* operator&() { return &m_str; {
```

当作为方法调用的输出接收一个 BSTR 指针时，这个方法相当有用。我们可以把返回的 BSTR 直接存储在 CComBSTR 对象中，这样这个对象将会管理字符串的生命周期，如下所示：

```
HRESULT get_Name (/* [out] */ BSTR* pName);

CComBSTR bstrName ;
get_Name (&bstrName); // bstrName empty so no memory leak
```

CopyTo 方法产生 CComBSTR 封装的字符串的一个副本，并且把副本的指针拷贝到指定位置。我们需要显式地调用 SysFreeString 来释放返回的 BSTR。

```
HRESULT CopyTo(BSTR* pbstr) { ... }
```

当我们需要把一个现有的 BSTR 属性的拷贝返回给调用者时，这个方法非常方便。例

如：

```
STDMETHODIMP SomeClass::get_Name (/* [out] */ BSTR* pName)
{
// Name is maintained in variable m_strName of type CComBSTR
 return m_strName.CopyTo (pName);
}
```

Detach 方法返回 CComBSTR 对象包含的 BSTR。它清空了对象，所以析构函数不会释放内部的 BSTR。我们需要显式地调用 SysFreeString 来释放返回的 BSTR。

```
BSTR Detach() {BSTR s = m_str; m_STR = NULL; return s;
```

如果在 CComBSTR 对象中有一个字符串，我们想把这个字符串返回给调用者并且不再需要保留它，那么我们应该使用这个方法。在这种情况下，使用 CopyTo 方法效率较低，因为我们必须要复制字符串，返回这个拷贝，然后丢弃原来的字符串。使用 Detach 直接返回原来的字符串，如下：

```
STDMETHODIMP SomeClass: :get_Label (/* [out] */ BSTR* pName)
{
 CComBSTR strLabel;
 // Generate the returned string in strLabel here
 *pName = strLabel.Detach ();
 return S_OK;
}
```

Attach 方法完成相反的操作。它接受一个 BSTR 并且把它附到一个空的 CComBSTR 对象上。BSTR 的所有权现在属于 CComBSTR 对象并且对象的析构函数最终会释放这个字符串。注意我们不应该把一个 BSTR 附到一个非空的 CComBSTR 对象上。如果这样做，调试（debug）版本在运行时会报告一个断言（assertion）。但是正式（release）版本将会悄无声息地产生内存泄漏。

```
void Attach(BSTR src) { ATLASSERT(m_str == NULL); m_str = src; }
```

Attach 方法要小心使用。我们必须拥有将被附到 CComBSTR 对象的 BSTR 的所有权，因为最后对象将试图释放这个 BSTR。例如，下面的代码是错误的：

```
STDMETHODIMP SomeClass::put_Name (/* [in] */ BSTR bstrName)
{
 // Name is maintained in variable m_strName of type CComBSTR
 m_strName. Empty();
 m_strName.Attach (bstrName); // Wrong! We don't own bstrName
 return E_BONEHEAD;
}
```

当我们放弃对一个 BSTR **的所有权，并且希望一个** CComBSTR **对象来管理字符串的生命周期时，我们常常会使用** Attach**。**

```
STDMETHODIMP SomeClass::get_Name (/* [out] */ BSTR* pName);
...
BSTR bstrName;
pObj->get_Name (&bstrName); // We own and must free the raw BSTR

CComBSTR strName;
strName.Attach(bstrName); // Attach raw BSTR to the object
```

当 BSTR **欲附着的目标** CComBSTR **对象已经包含了一个** BSTR **的时候，首先调用** Empty **方法。**Empty **方法释放任何内部的** BSTR **并且把成员变量** m_str **置为** NULL**。函数** SysFreeString **的文档明确地说明当输入参数为** NULL **时函数只是简单地返回，所以对空对象调用** Empty **也没有问题。**

```
void Empty() { ::SysFreeString(m_str);m_str = NULL; }
```

2.3.4　使用 CComBSTR 连接字符串

把指定的字符串连接到 CComBSTR **对象的方法有** 6 **个：四个** Append **重载方法，一个** AppendBSTR **方法和** operator+=()**操作符。**

```
HRESULT Append(PCOLESTR lpsz,int nLen);
HRESULT Append(LPCOLESTR lpsz);
#ifndef OLE2ANSI
HRESULT Append(LPCSTR);
```

```
#endif

HRESULT Append(const CComBSTR& bstrSrc);
CComBSTR& operator+=(const CComBSTR& bstrSrc);

HRESULT AppendBSTR(BSTR p);
```

Append(LPCOLESTR lpsz, int nLen)**方法计算当前字符串长度和** nLen **的总和，并且申请一个正确大小的空** BSTR**。它把原来的字符串复制到新的** BSTR**，然后把** lpsz **字符串的前** nLen **个字符串连接到新** BSTR **的尾部。最后，释放原来的字符串代之以新的** BSTR**。**

```
CComBSTR strSentence = OLESTR("Now is ");
strSentence.Append(OLESTR("the time of day is 03:00 PM"), 9);
// strSentence contains "Now is the time"
```

剩下 3 **个重载版本的** Append **函数都使用第一个方法来完成真正的工作。区别仅在于方法获得字符串和长度的方式。**Append(LPCOLESTR lpsz)**函数连接一个** NUL **字符结尾的** OLECHAR **字符串。**Append(LPCSTR lpsz)**方法连接一个** NUL **字符结尾的** ANSI **字符串。**Append(const CComBSTR& bstrSrc)**方法连接另一个** CComBSTR**。为了方便表示和语句构造上的便利，**operator+=()**方法也把指定的** CComBSTR **连接到当前字符串。**

```
CComBSTR strll (OLESTR("for all good men "));
strSentence.Append(strll); // calls Append(const CComBSTR& bstrSrc);
// strSentence contains "Now is the time for all good men"

strSentence.Append(OLESTR("to come ")); // calls Append (LPCOLESTR
                                                              1 psz);
// strSentence contains "Now is the time for all good men to come"

strSentence.Append("to the aid "); // calls Append (LPCSTR lpsz);
// strSentence contains
// "Now is the time for all good men to come to the aid"

CComBSTR str12 (OLESTR("of their country"));
StrSentence += str12; // calls operator+=()
// "Now is the time for all good men to come to the aid of their
```

```
// country"
```

当使用 BSTR 参数调用 Append 时，我们事实上调用的是 Append(LPCOLESTR lpsz) 方法，因为对编译器来说 BSTR 参数就是 OLECHAR*参数。因此这个方法从 BSTR 添加字符直至遇到第一个 NUL 字符。当添加可能嵌入了 NUL 字符的 BSTR 中的内容时，我们必须明确地调用 AppendBSTR 方法。按照下面的指示我们就不会犯错误：

- 当参数是 BSTR 时，使用 AppendBSTR 方法来添加整个 BSTR，不管它是否包含内嵌的 NUL 字符。
- 当参数是 LPCOLESTR 或者 LPCSTR 时，使用 Append 方法来添加 NUL 字符结尾的字符串。
- 函数重载就这些...

2.3.5　字符大小写转换

两个大小写转换方法——ToLower 和 ToUpper，分别把内部的字符串转换成小写和大写。这两个方法都使用了 ATL 字符串转换宏 OLE2T，这个宏在必要时把指向 OLECHAR 字符的指针转换成 TCHAR 字符串。Win32 函数 CharLower 和 CharUpper 用来转换 TCHAR 字符串的大小写。最后这两个方法使用 ATL 字符串转换宏 T2BSTR 来把经过大小写转换的字符串转换回 BSTR。当所有工作完成后，新的字符串替换原来的字符串成为 CComBSTR 对象的内容。

```
 USES_CONVERSION;
 if (m_str != NULL) {
    LPTSTR psz = CharLower(OLE2T(m_str));
    if (psz == NULL) return E_OUTOFMEMORY;
    BSTR b = T2BSTR(psz);
    // if (psz == NULL) return E_OUTOFMEMORY; // ATL 3.0 BUG!
    if (b == NULL)  return E_OUTOFMEMORY;
  SysFreeString(m_str);
  M_str = b;
 }
 retrun S_OK;
}
```

注意不管实际的字符串有多长，从 OLECHAR 字符到 TCHAR 字符的转换终止于第一个内嵌的 NUL 字符。由这些方法产生的、大小写转换后的 BSTR 在源字符串中第一个内

嵌 NUL 字符的位置被截断。同时，因为当本地语言代码页（local code page）不包含源 UNICODE 字符的等价字符时这个字符无法转换，所以转换可能是有损失的。

2.3.6 CComBSTR 比较操作符

最简单的比较操作符是 operator!()。当 CComBSTR 对象为空时它返回 true,否则返回 false。

```
Bool operator!() const { return (m_str == NULL);}
```

operator<()和 operator==()方法各有两个重载版本，因为两个方法中的代码几乎一样，所以我们只讨论 operator<()方法。这些讨论同样适用于 operator==()方法。

所有这些比较操作符在一定程度上都有一些值得争议的地方。基本的问题是，它们不支持对两个包含内嵌 NUL 字符的字符串的 CComBSTR 对象进行比较。这些比较函数使用了 ANSI C 的运行库函数 wcscmp 进行比较。wcscmp 接受 NUL 字符结尾的字符串。因此，在第一次碰到 NUL 字符时它就停止了比较。

另外，wcscmp 函数使用遵从“C 运行时刻本地语言设置（runtime locale setting）”的比较规则比较字符串。缺省情况是“C”本地语言设置 —— 仅比较 52 个不带重音的字母字符。要想使用操作系统的本地设置来比较字符串，我们必须像下面那样调用 setlocale：

```
setlocale (LC_ALL, "");
```

在 operator<()方法的第一个重载版本中，操作符用于比较一个给定的 BSTR 参数。函数原型表明了（因此也隐含了）参与比较的是 BSTR，而不是简单的 OLECHAR*。因此，我曾经认为它能够处理内嵌的 NUL 字符。但是因为它使用了函数 wcscmp，所以它做不到。

```
Bool operator<(BSTR bstrSrc) const
{
   if (bstrSrc == NULL && m_str == UNLL) return false;
   if (bstrSrc != NULL && m_str != NULL) retrun wcscmp;
     (m_str, bstrSrc) <0;
   retrun m_str == NULL ;
}
```

在 operator<()方法的第二个重载版本中，操作符用于比较一个给定的 LPCSTR 参数。LPCSTR 与内部包含宽字符的 BSTR 字符串的字符类型不同。因此在执行比较之前，这个

方法使用 ATL 辅助宏 A2W 把 ANSI 字符串转换成宽字符。

```
Bool operator<(LPCSTR pszSrc) const
{
  if (pszSrc == NULL && m_str == UNLL) return false;
  USES_CONVERSION;
  if (pszSrc != NULL && m_str != NULL) retrun wcscmp(m_str, AZW
    (pszSrc)) <0;
   retrun m_str == NULL ;
}
```

很奇怪，再没有其他比较操作符了。CComBSTR 类没有定义 operator>()或者 operator!=() 方法。虽然它们不是必须的，但是有了它们，使用起来会更方便，而且它们的实现也很简单。

2.3.7　CComBSTR 对永久性的支持

CComBSTR 类最后两个方法从流中读出一个 BSTR 或者把一个 BSTR 写入流中。WriteToStream 方法把表示 BSTR 长度（以字节为单位）的 ULONG 计数值写入流中。紧跟着这个数值之后写入 BSTR 字符。注意这个方法没有用写入数据时的字节顺序来标记流。因此与通常情况的流数据一样，CComObject 对象以硬件结构相关的格式把字符串写到流中。

```
HRESULT WriteToStream(IStream* pStream)
{
   ATLASSERT(pStream != NULL);
   ULONG cb;
   ULONG cbStrLen = m_str ? SysStringByteLen(m_str)+sizeof
     (OLECHAR) : 0;
   HRESULT hr = pStream->Write((void*) &chStrLden, sizeof (cbStrLen),
     &cb);
   if (FAILED(hr)) return hr;
   return cbStrLen ? pStream->Write((void*) m_str, cbStrLen, &cb);
     S_OK;
}
```

ReadFromStream 方法从指定流中读出一个 ULONG 计数值（字节数），申请一个正确大小的 BSTR，然后把字符直接读入 BSTR 字符串。当调用 ReadFromStream 时 CComBSTR 对象必须是空的，否则，调试版本下我们会收到一个断言(assertion)，而正式版本则会悄无声息地产生内存泄漏。

```
HRESULT ResdFromStream(IStream* pStream)
{
   ATLASSERT(pStream != NULL);
   ATLASSERT(m_str == NULL); // should be empty
   ULONG cbStrLen = 0;
   HRESULT hr = pStream->Read((void*) &cbStrLen, sizeof (cbStrLen),
    NULL);
   if ((hr == S_OK) && (cbStrLen != 0)) {
        //subtract size for terminating NULL that we wrote out
        //since SysAllocStringByteLen overallocates for the NULL
       m_str = SysAllocStringByteLen(NULL, cbStrLen-sizeof(OLECHAR));
        if (m_str == NULL) hr = E_OUTOFMEMORY;
        else hr = pStream->Read((void*) m_str, cbStrLen, NULL);
   }
   if (hr == S_FALSE) hr = E_FAIL;
   return hr;
}
```

2.3.8 关于 BSTR 的注意点、内嵌 NUL 字符的字符串

编译器认为 BSTR 类型和 OLECHAR*是相同的。事实上符号 BSTR 就是 OLECHAR*的一个 typedef。例如，在 wtype.h 中：

```
typedef /* [wire_marshal] */ OLECHAR __RPC_FAR *BSTR;
```

这简直使人伤透脑筋。一个任意的 BSTR 不是一个 OLECHAR*，并且任意的 OLECHAR*也不是 BSTR。常常误导我们作出错误判断的原因是 把 BSTR 当作 OLECHAR*使用能够正常工作，这种事情经常发生。

```
STDMETHODIMP SomeClass::put_Name (LPCOLESTR pName) ;
```

```
BSTR bstrlnput = . . .
pObj->put_Name (bstrlnput) ; // This works just fine.., usually
SysFreeString (bstrlnput) ;
```

在上面的例子中，bstrInput 参数按照定义是一个 BSTR，能够在字符串中包含内嵌的 NUL 字符。接受 LPCOLESTR（NUL 字符结尾的字符串）参数的 put_Name 方法，可能只保留第一个内嵌 NUL 字符之前的字符。换句话说，它会把字符串截短。

在需要[out] OLECHAR*参数的地方，我们也不能使用 BSTR。例如：

```
STDMETHODIMP SomeClass::get_Name (/* [out] */ OLECHAR** ppName)
{
 BSTR bstrOutput =.      // Produce BSTR string to return
 *ppName = bstrOutput ; // This compiles just fine
 return S_OK ;       // but leaks memory as caller doesn't release BSTR
}
```

相对地，在需要 BSTR 参数的地方我们也不能使用 OLECHAR*。即使碰巧可以工作，也是一个潜在的错误。例如，下面的代码是错误的：

```
STDMETHODIMP SomeClass::put_Name (BSTR bstrName) ;

pObj->put_Name (OLECHAR("This is not a BSTR!")) ; // Wrong! Wrong!
```

put_Name 方法会调用 SysStringLen 来获得 BSTR 的长度，当试图从字符串前面的整数获取 BSTR 的长度时它将会非常吃惊，事实上表示长度的整数并不存在。如果 put_Name 是一个远程方法（即运行在调用进程之外）事情会变得更遭。在这种情况下，列集(marshaling)代码将会调用 SysStringLen 来获得要放入 ORPC 请求包的字符个数。通常这是一个巨大的数值（在这个例子中是文字字符串前面的四个字节），并且在试图复制字符串时经常会导致一个 GPF(一般保护错——译注)。

因为编译器不知道 BSTR 和 OLECHAR*的区别，所以很容易调用错误的 CComBSTR 方法来处理包含内嵌 NUL 字符的 BSTR。下面的讨论告诉我们应该为这些 BSTR 使用哪些方法。

为了构造一个 CComBSTR，我们必须指定字符串的长度：

```
BSTR bstrlnput =
```

```
 SysAllocStringLen (OLESTR ("This is part one\Oand here's part two"),
    36);

CComBSTR str8 (bstrInput) ; // Wrong! Unexpected behavior here
                      // Note: str8 contains only "This is part one"
CComBSTR str9 (::SysStringLen (bstrInput), bstrInput); // Correct!
// str9 contains "This is part one\Oand here's part two"
```

把一个包含内嵌 NUL **字符的** BSTR **赋值给一个** CComBSTR **对象永远无法正常工作。例如：**

```
// BSTR bstrlnput contains "This is part one\Oand here's part two"
CComBSTR strl0 ;
strlO = bstrlnput; // Wrong! Unexpected behavior here
                // strl0 now contains "This is part one"!!!
```

完成 BSTR **赋值工作最简单的方法是使用** Empty **和** AppendBSTR **方法：**

```
strlO.Empty ();              // Insure object is initially empty
strlO.AppendBSTR (bstrlnput); // This works!
```

使用 ToLower **和** ToUpper **方法要小心。经过大小写转换的字符串在第一个内嵌** NUL **字符处被截断。同样地，比较操作符 —— operator<()和** operator==()**，仅比较第一个内嵌** NUL **字符之前的字符。**

实际上，虽然 BSTR **可能潜在地包含** 0 **个或者多个内嵌的** NUL **字符，但是大多数情况下都不包含。这意味着在大多数情况下，我们看不到由于错误的** BSTR **用法而导致的错误。**

2.4 智能 VARIANT 类 ComVariant

2.4.1 回顾 COM VARIANT 数据类型

有时在使用 COM **的时候，我们希望在对方法所要求的数据类型一无所知的情况下向它传递参数。为了使方法能够解释接收到的参数，调用者必须指定数据的格式和相应的值。**

另外，我们调用的方法返回的结果根据不同的环境可能是不同的数据类型。有时返回一个字符串，有时是一个 long，甚至是一个接口指针。这要求方法返回的数据必须能够自我描述格式。对于每个传送的值，必须发送两个域：一个指定数据类型的编码和一个使用指定数据类型表示的值。显然，为了达到这个目的，发送者和接收者必须就可能的格式集合达成一致。

COM 指定了这样一个可能的格式集合（枚举类型 VARTYPE），并且指定了包含格式和相应值的结构（VARIANT 结构）。VARIANT 结构的定义如下：

```
typedef struct FARSTRUCT tagVARIANT VARIANT;
typedef struct FARSTRUCT tagVARIANT VARIANTRG;

typedef struct tagVariant {
  VARTYPE vt;
  unsigned short wReserved1;
  unsigned short wReserved2;
  unsigned short wReserved3;
  union {
    unsigned char         bVal;       // VT_UI1
    short                 iVal;       // VT_I2
    long                  lVal;       // VT_I4
    float                 fltVal;     // VT_R4
    double                dblVal;     // VT_R8
    VARIANT_BOOL          boolVal;    // VT_BOOL
    SCODE                 scode;      // VT_ERROR
    CY                    cyVal;      // VT_CY
    DATE                  date;       // VT_DATE
    BSTR                  bstrVal;    // VT_BSTR
    IUnknown         FAR* punkVal;    // VT_UNKNOWN
    IDispatch        FAR* pdispVal;   // VT_DISPATCH
    SAFEARRAY        FAR* parray;     // VT_ARRAY|*
    unsigned char    FAR* pbVal;      // VT_BYREF|VT_UI1
    short            FAR* piVal;      // VT_BYREF|VT_I2
    long             FAR* plVal;      // VT_BYREF|VT_I4
    float            FAR* pfltVal;    // VT_BYREF|VT_R4
    double           FAR* pdblVal;    // VT_BYREF|VT48
    VARIANT_BOOL     FAR* pboolVal;   // VT_BYREF|VTBOOL
```

```
    SCODE             FAR* pscode;       // VT_BYREF|VTERROR
    CY                FAR* pcyVal;       // VT_BYREF|VTCY
    DATE              FAR* pdate;        // VT_BYREF|BSTR
    BSTR              FAR* pbstrVal;     // VT_BYREF|UNKNOWN
    IUnknown FAR*     FAR* ppunkVal;     // VT_BYREF|DISPATCH
    IDispatch FAR*    FAR* ppunkVal;     // VT_BYREF|DISPATCH
    SAFEARRAY FAR*    FAR* pparray;       // VT_ARRAY|*
    VARIANT           FAR* pvarVal;      // VT_BYREF|VT_VARIANT
    Void              FAR* byref;        // Generic ByRef
  };
};
```

我们通过把值存进联合中的一个域，然后把相应的类型编码存入 VARIANT **结构的** vt **成员来初始化** VARIANT **结构。对于** C++**开发人员来说，**VARIANT **结构有一些使用起来单调乏味并且容易出错的语义。**

- VARIANT **在使用之前必须调用** VariantInit **函数进行初始化。或者，我们可以把类型和与类型关联的值域初始化为合法的状态。**
- **必须通过调用** VariantCopy **函数来复制一个** VARIANT**。这个函数根据在** VARIANT **中存储的数据类型正确地完成浅层的或者深层的复制工作。**
- **必须调用** VariantClear **来释放** VARIANT**。这个函数根据** VARIANT **中存储的数据类型正确地完成浅层的或者深层的释放工作。例如，当释放一个包含** BSTR **的** SAFEARRAY **的** VARIANT **时，**VariantClear **首先释放数组中每个** BSTR**，然后再释放数组本身。**
- VARIANT **可以有选择地表示最多一层的间接性，这可以通过在类型码中设置** VT_ARRAY **位来完成。我们可以调用** VariantCopyInd **以去除** VARIANT **中这一层间接性。**
- **调用** VariantChangeType[Ex] **可以试图改变** VARIANT **的数据类型。**

有这么多特殊的语义，把这些细节封装到一个可重用的类中会非常有益。ATL **提供了这样一个类：**CComVarient**。**

2.4.2 CComVariant 类

CComVariant **类是一个** ATL **工具类，用于封装** COM **的自我描述数据类型——**VARIANT**。**atlbase.h **文件包含了** CComVariant **类的定义。**CComVariant **类维护的唯一状态是一个** VARIANT **结构的实例，它通过继承** tagVARIANT **结构而得到这个实例。这意味着** CComVariant **实例就是一个** VARIANT **结构，所以我们可以把** CComVariant **实例传递给任**

何需要 VARIANT **结构的函数。**

```
class CComVariant:public tagVARIANT
{
. . .
} ;
```

当需要把一个 VARIANT 参数传递给一个 COM 方法时，我们经常使用 CComVariant 实例。下面的代码把一个字符串参数传递给一个需要 VARIANT 的方法。代码中使用了 CComVariant 类，所以它不需要处理 BSTR 混乱的语义。

```
STDMETHODIMP put_Name(/* [in]* const VARIANT* name);

HRESULT SetName (LPTSTR pszName)
{
     // Initializes the VARIANT structure
    // Allocates a BSTR copy of pszName
    // Sets the VARIANT to the BSTR
    CComVariant v (pszName);

    // Pass the raw VARIANT to the method
    Return pObj->put_Name(&v);

    // Destructor clears v, freeing the BSTR
}
```

2.4.3　构造和析构函数

CComVariant 对象有 16 个构造函数。缺省的构造函数简单地把 VARIANT 初始化为 EMPTY，EMPTY 表示 VARIANT 不包含任何信息。析构函数调用 Clear 成员函数来释放 VARIANT 拥有的任何潜在的资源。

```
CComVariant() { vt = VT_EMPTY; }
~CComVariant() { Clear (); }
```

另外 15 个构造函数根据构造函数的参数类型来适当地初始化 VARIANT 结构。

许多构造函数只是简单地把 VARIANT 的 vt 成员赋值为表示构造函数参数类型的值，并且把参数值存入联合的对应成员中。

```
CComVariant(int nSrc)         { vt = VT_I4;  lVal = nSrc; }
CComVariant(BYTE nSrc)        { vt = VT_UI1; BVal = nSrc; }
CComVariant(short nSrc)       { vt = VT_I2;  iVal = nSrc; }
CComVariant(float fltSrc)     { vt = VT_R4;  fltVal = fltSrc; }
```

```
CComVariant(double dblSrc)    { vt = VT_R8;  dblVal = dblSrc; }
CComVariant(CY cySrc)         { vt = VT_CY;  cyVal.Hi = cySrc.Hi ;
                                             cyVal.Lo = cySrc.Lo ; }
```

有几个构造函数要复杂一些。对于编译器来说，SCODE 看起来像一个 long。因此，指定一个 SCODE 或者 long 初始值的情况下构造一个 CComVariant，应该调用接受 long 的构造函数。为了允许我们区别这两种情况，构造函数有一个可选的参数来指定放入 VARIANT 的 long 是作为 long 还是 SCODE。当我们指定的 variant 类型不是 VT_I4 或者 VT_ERROR 时，这个构造函数在调试版本下会报告一个断言(assertion)。

Windows 16 位的 COM 把 HRESULT（指向结果码的句柄）定义成包含 SCODE（status code，状态码）的数据类型。因此，我们偶尔见到遗留的旧代码认为两个数据类型是不同的。事实上，有一些已经被废弃的宏能够把 SCODE 转换成 HRESULT，以及从 HRESULT 中提取出 SCODE。但是在 32 位 COM 应用程序中 SCODE 和 HRESULT 数据类型是一样的。VARIANT 数据结构包含有一个 SCODE 域而不是 HRESULT 域，因为那是最初的声明方式。

```
CComVariant(long nSrc, VARTYPE vtSrc = VT_I4);
```

接受一个 bool 初始值的构造函数把 VARIANT 的内容设置为 VARIANT_TRUE 或者 VARIANT_FALSE，而不是指定的 bool 值。一个逻辑值 TRUE，在 VARIANT 中必须表示成 VARIANT_TRUE（16 位值-1），逻辑值 FALSE 是 VARIANT_FALSE（16 位的 0）。C++ 语言定义 bool 数据类型是一个 8 位的值（0 或者 1）。该构造函数提供了在两种布尔（Boolean）值表示之间的转换功能。

```
CComVariant(bool bSrc)
     { vt = VT_BOOL; boolVal = bSrc ? VARIANT_TPUE : VARIANT_FALSE ; }
```

有两个构造函数接受一个接口指针作为初始值，并且生成一个 CComVariant 实例，这个实例包含了一个经过 AddRef 之后的接口指针的拷贝。第一个构造函数接受 IDispatch* 形式的参数。第二个接受 IUnknown*形式的参数。

```
CComVariant(IDispatch* pSrc);
CComVariant(IUnknown* pSrc);
```

有两个构造函数允许我们使用另一个 VARIANT 结构或者 CComVariant 实例来初始化当前 CComVariant 实例。

```
CComVariant(const VARIANT& varSrc) { VT = VT_EMPTY; InternalCopy
                                    (&varSrc); }
CComVariant(const CComVariant& varSrc); { Same as above }
```

它们的实现完全相同，并且它们的实现会导致微妙的副作用，所以让我们看看这两个构造函数都用到的辅助函数 InternalCopy。

```
void InternalCopy(const VARIANT* pSrc)
{
    HRESULT hr = Copy(pSrc);
    if (FAILED(hr)) { vt = VT_ERROR; scode =hr; }
}
```

注意 InternalCopy 函数是如何把指定的 VARIANT 复制到被构造的实例中的；当复制失败时，InternalCopy 把 CComVariant 实例初始化成包含一个错误码（VT_ERROR）。实际的错误码是用来进行复制的 Copy 方法的返回值。在认识到“构造函数除了抛出异常之外没有其他返回错误的方法”之前，我们会觉得这看起来是一个很奇怪的方法。ATL 不要求一定支持异常处理，所以这个构造函数即使当 Copy 方法失败时也必须初始化实例。

我曾经有一个 CComVarient 实例，这个实例中好像总是有一个 VT_ERROR 码，甚至我认为不应该的时候也是这样。问题原来是这样的：我使用了一个驻留在栈上的未经初始化的 VARIANT 结构来构造 CComVariant 实例。观察如下的代码：

```
void func ()
{                       // The following code is incorrect
    VARIANT v;           // Uninitialized stack garbage in vt member
    CComVariant vv (v); // Indeterminate State - with luck, VT_ERROR
}
```

三个构造函数接受一个字符串初始值，并且产生一个包含 BSTR 的 CComVariant 实例。第一个构造函数接受 BSTR 形式的参数，并且创建一个包含指定 BSTR 的一份拷贝的 CComVarient。第二个构造函数接受 LPCOLESTR 形式的参数，并且创建一个包含指定 OLECHAR 字符串的 BSTR 拷贝的 CComVarient。第三个构造函数接受 LPCSTR 类型的参数，并且创建一个包含 BSTR 的 CComVarient，这个 BSTR 是由指定的 ANSI 字符串转换成 OLECHAR 字符串的拷贝。如果定义了符号 OLE2ANSI，那么 LPCOLECHAR 类型与 LPCSTR 类型是相同的。在这种情况下，ATL 不能再定义这第三个构造函数。这三个构造函数也有失败的可能。在三个构造函数中，当构造函数不能为新 BSTR 申请到空间时，它

会把 CComVariant 实例初始化为 VT_ERROR，并且 SCODE 为 E_OUTOFMEMORY。

```
CComVariant(BSTR bstrSrc);
CComVariant(LPCOLESTR lpszSrc);
#ifndef OLE2ANSI
CComVariant(LPCSTR lpszSrc);
#endif
```

接受 BSTR 形式的参数和接受 LPCOLECHAR 形式的参数的构造函数是分离的，这多少有点奇怪。因为本章早些时候已经强调过这个令人伤脑筋的 typedef，两种数据类型的唯一区别是 const 类型修饰符。也就是说，BSTR 是 OLECHAR*，而 LPCOLESTR 是 const OLECHAR*。这意味着实际调用的构造函数经常不是我们想调用的。

```
void func ()

{
 BSTR bstrInput = ::SysAllocString (OLESTR ("This is a BSTR string")) ;

 CComVariant vl (bstrInput); // calls CComVariant (BSTR)

 // calls CComVariant (BSTR)
 CComVariant v2 (OLESTR("This is an OLECHAR string")) ;

 OLECHAR* ps = OLESTR("This is another OLECHAR string") ;
 CComVariant v3 (ps); // calls CComVariant (BSTR)

 const OLECHAR* pcs = OLESTR("This is another OLECHAR string") ;
 CComVariant v4 (pcs); // Only this calls CComVariant (LPCOLESTR)

 ::SysFreeString (bstrInput) ;
}
```

CComVariant(BSTR)构造函数也不能正确地处理内嵌 NUL 字符的 BSTR。它仅使用 BSTR 中第一个 NUL 字符之前的字符来产生一个初始化的 CComVariant。总的来说，这个构造函数提供的功能并不比构造函数 CComVariant(LPCOLESTR)多。

2.4.4　初始化

CComVariant 类定义了 15 个赋值操作符。所有这些操作符

- 清除 VARIANT 的当前内容。
- 把 VARIANT 结构的成员 vt 设置为表示赋值操作符参数类型的值。
- 把参数的值存入联合的适当成员。

```
CComVariant& operator=(int nSrc);          // Creates VT_I4 variant
CComVariant& operator=(BYTE nSrc);         // Creates VT_UI1 variant
CComVariant& operator=(short nSrc);        // Creates VT_U2 variant
CComVariant& operator=(long nSrc);         // Creates VT_I4 variant
CComVariant& operator=(float fltSrc);      // Creates VT_R4 variant
CComVariant& operator=(double dblSrc);     // Creates VT_R8 variant
CComVariant& operator=(CY cySrc);          // Creates VT_CY variant
```

其余的 operator=方法还有其他的语义。和对应的构造函数一样，接受 bool 初始值的赋值操作符把 VARIANT 的内容设置为 VARIANT_TRUE 或 VARIANT_FALSE，而不是指定的 bool 值。

```
CComVariant& operator=(bool bSrc);
```

两个赋值操作符接受接口指针，并且生成一个包含接口指针的一份拷贝的 Ccom Variant 实例，它是经过 AddRef 之后的接口指针。其中一个接受 IDispatch*类型的参数。另一个接受 IUnknown*类型的参数。

```
CComVariant& operator=(IDispatch* pSrc);
CComVariant& operator=(IUnknown* pSrc);
```

有两个赋值操作符允许我们通过另一个 VARIANT 结构或者 CComVariant 实例来初始化当前的 CComVariant 实例。它们都使用了前面描述的 InernalCopy 方法来产生输入参数的一份拷贝。因此这些赋值操作符也有可能失败，并且产生一个初始化成 VT_ERROR 类型的实例

```
CComVariant& operator=(const CComVariant& varSrc);
CComVariant& operator=(const VARIANT& varSrc);
```

剩下的三个赋值操作符接受一个字符串初始值，并且产生一个包含 BSTR 的

CComVariant 实例。第一个赋值操作符接受 BSTR 形式的参数，并且创建一个包含指定 BSTR 拷贝的 CComVarient。第二个赋值操作符接受 LPCOLESTR 形式的参数，并且创建一个包含指定 OLECHAR 字符串的 BSTR 拷贝的 CComVarient。第三个赋值操作符接受 LPCSTR 类型的参数并且创建一个包含 BSTR 的 CComVariant，这个 BSTR 是由指定 ANSI 字符串经过转换而得到的 OLECHAR 字符串的拷贝。

```
CComVariant& operator=(BSTR bstrSrc);
CComVariant& operator=(LPCOLESTR 1pszSrc);
#ifndef OLEZANSI
CComVariant& operator=(LPCSTR 1pszSrc);
#endif
```

前面关于“使用字符串初始值的构造函数”的说明对于“使用字符串初始值的赋值操作符”同样适用。事实上，构造函数使用赋值操作符来完成它们的初始化。

- 这些赋值操作符可能“失败”，产生一个被初始化为 VT_ERROR 类型的 CComVariant 实例。
- operator=(BSTR)方法和 operator=(LPCOLESTR)方法并没有实质的不同。
- operator=(BSTR)方法需要一个 NUL 字符结尾的字符串。因此它仅复制第一个内嵌 NUL 字符之前的字符。

2.4.5 CComVariant 操作

认识到 VARIANT 是一个必须被正确管理的资源是非常重要的。就象堆上的内存必须被申请和释放，VARIANT 也必须被初始化和清除。就像“一个内存块的所有权必须被明确地管理起来，从而它仅被释放一次”一样，VARIANT 内容的所有权也必须要被明确地管理起来，从而它仅被清除一次。有四个方法可让我们控制 CComVariant 实例拥有的任何资源。

Clear 方法通过调用 VariantClear 函数释放实例可能包含的任何资源。对于包含诸如 long 或者类似数值的实例，这个方法什么也不做。但是对于包含 BSTR 的实例，这个方法将释放字符串。对于包含接口指针的实例，这个方法将释放接口指针。对于包含 SAFEARRAY 的实例，这个方法将释放数组的每个元素，然后释放 SAFEARRAY 本身。

```
HRESULT Clear() { return ::VariantClear(this); }
```

当不再需要一个 CComVariant 实例包含的资源时，我们应该调用 Clear 方法来释放它。析构函数会自动完成此事。所以，如果在用完了实例资源之后，实例将很快离开作用域，

那么让析构函数负责清除工作。但是，如果 CComVariant 是一个全局或者静态变量，那么它将长时间停留在作用域内。这种情况下 Clear 方法非常有效。

Copy 方法生成指定 VARIANT 的唯一拷贝。Copy 方法产生的 CComVariant 实例的生命周期与被复制的 VARIANT 的生命周期无关。

```
HRESULT Copy(const VARIANT* pSrc)
      { return ::VariantCopy(this,const_cast<VARIANT*>(pSrc)); }
```

Copy 方法经常被用来复制一个作为[in]参数接收到的 VARIANT。调用者提供一个[in]参数是把资源借给我们。当我们希望拥有这个参数的时间超过方法调用的范围时，我们需要复制 VARIANT。

```
STDMETHODIMP SomeClass::put_Option (/* [in] */ const VARIANT* pOption)
{
    // Option saved in member m_Option of type CComVariant
    return m_varOption,Copy (pOption) ;
}
```

当我们想把 CComVariant 实例中资源的所有权转移给一个 VARIANT 结构的时候，我们应该使用 Detach 方法。它清除目标 VARIANT 结构，执行 memcpy 把 CComVariant 实例复制到指定的 VARIANT 结构，并且把实例设为 VT_EMPTY。注意，这项技术避免了额外的内存分配和 AddRef/Release 调用。

```
HRESULT Detach(VARIANT* pDest);
```

不要随便使用 Detach 方法来更新一个[out]VARIANT 参数！一个[out]参数在进入方法时是未经初始化的。Detach 方法在改写指定的 VARIANT 之前首先清除它。清除一个包含随机信息的 VARIANT 将导致任意不确定的行为。

```
STDMETHODIMP SomeClass::get_Option (/* [out] */ VARIANT* pOption)
{
    CComVariant varOption ;
    ...   Initialize the variant wi th the output data
    // Wrong! The following code can generate an exception, corrupt
    // your heap, and give at least seven years' bad luck!
    return varOption.Detach (pOption);
```

```
}
```

在把 CComVariant **实例中的资源分离（**detach**）到一个**[out]VARIANT **参数之前，一定要保证输出参数已经被初始化。**

```
// Special care taken to initialize [out] VARIANT
::Variantlnit (pOption) ;     Jori     pOption->vt = VT_EMPTY ;
return varOption. Detach (pOption); // Now we can Detach safely.
```

当我们想把 VARIANT **结构的资源的所有权从结构交给一个** CComVariant **实例时，我们可以使用** Attach **方法。它清除当前的实例，执行** memcpy **把** VARIANT **结构复制到当前实例中，并且把指定的** VARIANT **置为** VT_EMPTY**。注意，这项技术避免了额外的内存分配和** AddRef/Release **调用。**

```
HRESULT Attach(VARIANT* pSrc);
```

客户代码可以使用 Attach **方法接受“作为**[out]**参数接收到的** VARIANT **的所有权”。提供了**[out]**参数的函数把资源的所有权交给调用者。**

```
STDMETHODIMP SomeClass::get_Option (/* [out] */ VARIANT* pOption);
void VerboseGetOption () {
VARIANT v;
pObj->get_Option (&v) ;

CComVariant cv;
cv,Attach (&v);   // Destructor now releases the VARIANT
}
```

我们可以使用下面更有效、但同时也是更危险的另一种编码方法：

```
void FragileGetOption() {
   CComVariant v;          // This is fragile code!!
   pObj->get_Option (&v) ; // Directly update the contained VARIANT
                           // Destructor now releases the VARIANT
}
```

注意在这种情况下，get_Option **方法改写了** CComVariant **实例包含的** VARIANT **结构。**

因为方法需要一个[out]参数，所以 get_Option 方法不会释放包含在参数中的任何资源。在上面的例子中，由于实例刚刚被构造，所以被改写时是空的。但是，下面的代码则导致一个内存泄漏。

```
void LeakyGetOptionO {
   CComVariant v (OLESTR ("This string leaks!")) ;
   pObj->get_Option (&v) ;// Directly updates the contained VARIANT
                        // Destructor now releases the VARIANT

}
```

当把 CComVariant 实例作为[out]参数传给一个需要 VARIANT 的方法时，如果实例有可能不为空，那么我们必须首先清除这个实例。

```
void NiceGetOption() {
   CComVariant v (OLESTR ("This string doesn't leak!")) ;
   ...
   v, Clear 0;
   pObj->get_Option (&v) ;// Directly updates the contained VARIANT
                        // Destructor now releases the VARIANT
}
```

ChangeType 方法把一个 CComVariant 实例转换成 vtNew 参数指定的新类型。如果我们指定了第二个参数，则 ChangeType 把它当作转换的源。否则，ChangeType 使用 CComVariant 实例作为转换的源并且就地完成转换。

```
HRESULT ChangeType(VARTYPE vtNew, const VARIANT* pSrc = NULL);
```

ChangeType 在基本类型之间进行转换（包括数字到字符串和字符串到数字的强制转换）。对于引用类型，ChangeType 通过找到被引用的值，把一个包含引用类型的源（就是说设置了 VT_BYREF 位）强制转换成一个值。对于对象类型，ChangeType 总是通过获取对象的 Value 属性(即 DISPID 为 DISPID_VALUE 的自动化属性)，把一个对象引用强制转换为一个值。

2.4.6　CComVariant 比较操作符

operator==()方法比较一个 CComVariant 实例是否与指定的 VARIANT 结构相等。

```
bool operator==(const VARIANT& varSrc) const;
bool operator!=(cosnt VARIANT& varSrc) const;
```

当两个操作数具有不同的类型时，操作符返回 false。对于基本的类型，操作符比较两个值来决定是否相等。操作符比较两个 BSTR 操作数时先比较它们的长度。当两个 BSTR 的长度相等时，操作符再比较指定长度内的所有字符。这意味着与 CComBSTR 类不同，CComVariant 的 operator==()方法能够正确地比较两个内嵌 NUL 字符的 BSTR。

接口指针的比较有点怪异并且是否有用值得怀疑。操作符直接比较接口指针的二进制数值。这就是说比较操作并不能决定两个接口指针是否引用同一个对象。

当客户查询指定的接口指针时（除了查询 IUnknown 接口以外），COM 允许对象每次给出一个不同的二进制数值。因此两个包含 IDispatch 接口指针的 VARIANT，尽管这两个接口指针引用同一个对象，但是比较的结果可能是不相等的。甚至两个包含 IUnknown 接口指针的 VARIANT，虽然这两个接口指针引用同一个对象，比较的结果也可能不相等。以 VT_UNKNOWN 类型放入 VARIANT 的接口指针并不要求是严格的 IUnknown（调用 QueryInterface（IID_IUnknown，...）返回的结果）。VT_UNKNOWN 类型的 VARIANT 可以容纳任何强制转换成 IUnknown 的接口指针。

operator!=()方法返回 operator==()方法的相反值，所以上面的说明全部适用于 operator!=()。

operator<() 方法和 operator>()方法使用 Variant Math API 函数 VarCmp 分别完成各自的比较工作。[2]

```
bool operator<(const VARIANT& varSrc) const;
bool operator>(cosnt VARIANT& varSrc) const;
```

2.4.7 CComVariant 对永久性的支持

CComVariant 类的最后两个方法从流中读取一个 VARIANT 或者把一个 VARIANT 写入流中。

```
HRESULT WriteToStream(IStream* pStream);
```

[2] 有许多操作系统函数可以处理 VARIANT。函数 VarAbs、VarAdd、VarAnd、VarCat、VarCmp、VarDiv、VarEqv、VarFix、VarIdiv、VarImp、VarInt、VarMod、VarMul、VarNeg、VarNot、VarOr、VarPow、VarRound、VarSub 和 VarXor 一起组成了 Variant Math API。我所找到的关于这些函数的唯一文档是定义在 oleauto.h 中的函数原型。

```
HRESULT ReadFromStream(IStream* pStream);
```

WriteToStream 方法把类型码 vt 写入流中。对于简单类型，例如 VT_I4、VT_R8 和其他类似的数值，紧跟着类型码后写入的是 VARIANT 的值。对于接口指针，当指针为 NULL 时 WriteToStream 把 GUID CLSID_NULL 写入流中。当接口指针不为 NULL 时，WriteToStream 查询引用对象的 IPersistStream 接口。如果对象支持这个接口，那么 WriteToStream 调用 COM 的 OleSaveToStream 函数把对象存入流中。当接口指针不为 NULL 并且对象不支持 IPersistStream 接口时，WriteToStream 失败。

- 注意：在 ATL3.0 中，WriteToStream 不使用 IPersistStreamInit 接口，虽然许多对象偏爱实现这个接口更甚于 IPersistStream。

对于复杂类型，包括 VT_BSTR、所有的引用类型和所有的数组，WriteToStream 在必要时把值转换成 BSTR 并且使用 CComBSTR:: WriteToStream 把字符串写入流中。

ReadFromStream 方法完成相反的操作。首先，它清除当前 CComVariant 实例。然后从流中读出 Variant 类型码。对于简单类型，例如 VT_I4、VT_R8 和其他类似的数值，它从流中读出 VARIANT 的值。对于接口指针，它调用 COM 的 OleLoadFromStream 函数从流中读出对象，请求适当的 IUnknown 或者 IDispatch 接口。当 OleLoadFromStream 返回 REGDB_E_CLASSNOTREG（通常是由于读出了 CLSID_NULL）时，ReadFromStream 默默地返回 S_OK 状态。

对所有其他类型，包括 VT_BSTR、所有的引用类型和所有的数组，ReadFromStream 调用 CComBSTR:: ReadFromStream 从流中读出先前写入的字符串。然后这个方法把字符串强制转换回原来的类型。

2.5　CComPtr、CComQIPtr、和 CComDispatchDriver 智能指针类

2.5.1　回顾智能指针

智能指针是一个行为与指针类似的对象。也就是说，在许多通常使用指针的场合我们也可以使用智能指针类的实例。但是，使用智能指针和使用裸指针相比有一些优点。例如，一个智能指针类可以：

- 在类的析构函数执行时，释放封装在内部的接口指针。
- 在栈上分配的智能接口指针，在异常处理期间会自动地释放它的接口指针，这样降低了编写显式异常处理代码的要求。

- 在赋值操作期间，在改写被封装的接口指针之前先释放它。
- 在赋值操作期间，对接收到的接口指针调用 AddRef。
- 通过便利的机制，提供了各种构造函数来初始化一个新的智能指针。
- 可以在许多(但不是全部)通常使用裸接口指针的场合下使用。

ATL 提供了三个智能指针类：CComPtr、CComQIPtr 和 CComDispatchDriver。

CComPtr 类是一个智能 COM 接口指针类。我们可以为指定的接口指针类型量体裁衣地创建实例。例如，下面代码中的第一行创建了一个智能 IUnknown 接口指针。第二行创建了自定义接口 INamedObject 的智能接口指针。

```
CComPtr<IUnknown>        punk;
CComPtr<InamedObject>    pno;
```

CComQIPtr 类是一个更智能的 COM 接口指针类，能够完成 CComPtr 的所有功能，以及更多其他的功能。当我们把一个与智能指针不同类型的接口指针赋值给 CComQIPtr 实例时，这个类会对输入的接口指针调用 QueryInterface。

```
CComPtr<IUnknown>            punk = /* Init to some IUnknown* */ ;
CComQIPtr<INamedObject>      pno = punk;
   // Calls punk->QI (IID_INamedObject, ...)
```

CComDispatchDriver 类是智能 IDispatch 接口指针。我们利用这个类的实例通过 IDispatch 接口来获取和设置对象的属性。

```
CComVariant            v;
CComDispatchDriver     pdisp : /* Init to object's IDispatch* */ ;

HRESULT hr = pdisp->GetProperty (DISPID_COUNT, &v); // Get the
   Count property
```

2.5.2 CComPtr 和 CComQIPtr 类

CComPtr 和 CComQIPtr 类，除了初始化和赋值部分之外，其他部分是非常相似的。所以除非我明确声明，否则下面关于 CComPtr 类的全部说明同样适用于 CComQIPtr 类。

atlbase.h 文件包含这两个类的定义。每个类维护的唯一状态是一个公有成员变量 ——T* p。

```
template <class T>    template <class T,const IID* piid =
                     &_uuidof(T)>
class  CCOmPtr        class CComQIPtr
{                    {
public:              public:
     T* p;           T* p;
...                  ...
} ;                  };
```

第一个模板参数指定了智能接口指针的类型。CComQIPtr **类的第二个模板参数指定了智能指针的接口** ID**。缺省情况下，它是与类的第一个参数相关联的全局唯一标识符（**GUID **，**globally unique identifier**）。这里有一些使用这些智能指针类的例子。中间三个例子是等价的。**

```
CComPtr<IUnknown> punk;    // Smart IUnknown*
CComPtr<INamedObject> pno; // Smart INamedObject*

CComQIPtr<INamedObject> pno;
CComQIPtr<INamedObject, &_uuidof(INamedObject)> pno;
CComQIPtr<INamedObject, &IID_INamedObject> pno;

CComQIPtr<IDispatch, &IID_ISomeDual> pdisp;
```

构造函数和析构函数

CComPtr **对象可以通过一个适当类型的接口指针进行初始化。也就是说，**CComPtr<IFoo>**对象能够使用** IFoo***或者另一个** CComPtr<IFoo>**对象来初始化。使用其他类型会产生一个编译错误。缺省的构造函数把内部的接口指针初始化为** NULL**。其他构造函数把内部的接口指针初始化为指定的接口指针。如果指定的值不是** NULL**，则构造函数会调用** AddRef **方法。析构函数对非** NULL **的接口指针调用** Release **方法。**

```
CComPtr()                       { p = NULL; }
CComPtr(T* lp)                   { if ((p = lp) != NULL) p-> AddRef(); }
CComPtr(const CComPtr<T>& lp)
   { if ((p = lp.p) != NULL) p-> AddRef(); }
```

```
~CComPtr()                          { if (p) p->Release(); }
```

CComQIPtr 对象能够使用任意类型的接口指针进行初始化。当初始值与智能指针的类型相同时，这个构造函数与 CComPtr 类一样，只是简单地对输入的接口指针调用 AddRef。但是，当指定不同的类型来调用下面的构造函数时，构造函数会在输入接口指针中查询适当的接口：

```
CComQIPtr(IUnknown* lp)
  { p= NULL; if (lp != NULL) lp->QueryInterface(*piid, (void **)&p); }
```

构造函数永远不会失败。然而 QueryInterface 调用可能会不成功。当不能获得被请求的接口时，CComQIPtr 类把内部的指针置为 NULL。因此，我们必须像下面的代码那样测试对象是否已经被初始化：

```
void func (IUnknown* punk)
{
    CComQIPtr<INamedObject> pno (punk);
    if (pno) {
           // Can call SomeMethod because the QI worked
         pno->SomeMethod ();
   }
}
```

我们可以通过检查是否为 NULL 指针，从而知道查询是否失败，但是我们还是不知道为什么会失败。构造函数没有保存失败的 QueryInterface 调用返回的 HRESULT。

初始值

CComPtr 类定义了两个赋值操作符，CComQIPtr 类定义了同样的两个赋值操作符，再加上第三个。所有这些赋值操作符：

- 当前接口指针非 NULL 时 Release 它。
- 当源接口指针非 NULL 时 AddRef 它。
- 把源接口指针保存为当前接口指针。

CComQIPtr 增加的第三个赋值操作符在非 NULL 的源接口指针上查询适当的接口，并且保存查询的结果接口指针。当 QueryInterface 调用失败时，我们收到一个 NULL 指针。与等价的构造函数一样，它没有提供查询失败时的 HRESULT。

```
// CComPtr assignment operators
T* operator=(T* lp);
T* operator=(const CComPtr<T>& lp);
```

```
// CComQIPtr assignment ops.
T* operator=(T* lp);
T* operator=
  (const CComQIPtr<T>& lp);
T* operator=(IUnknown* lp);
```

我们一般使用 CComQIPtr **的赋值操作符来完成一个** QueryInterface **调用。紧跟着赋值操作之后的是** NULL **指针测试，如下：**

```
CComQIPtr<IExpectedlnterface> m_object;  // Member variable holding
   object
STDMETHODIMP put_Object (IUnknown* punk)
{               // Releases current object, if any, and
  m_object = punk; // queries for the expected interface
  if (!m_object)
      return E_UNEXPECTED;
  return S_OK;
}
```

📖 对象实例化方法

智能接口指针类提供了一个被称为 CoCreateInstance **的重载方法，可以用它来实例化一个对象并且获得对象的一个接口指针。这个方法有两种形式。第一种要求实例化类的类标识符（**CLSID**）。第二种要求实例化类的程序标识符（**ProgID**）。这两个重载函数都接受用于实例化的可选参数，分别用于指定作为外部控制的** unknown **和类的环境。外部控制的** unknown **参数缺省为** NULL **——这是一般情况，表示没有聚合。类环境参数缺省为** CLSCTX_ALL**，表示任何可用的服务器都可以为请求提供服务。**

```
HRESULT CoCreateInstance (REFCLSID rclsid,
                          LPUNKNOWN pUnkOuter = NULL,
                          DWORD dwClsContext = CLSCTX_ALL
{
   ATLASSERT(p == NULL);
   return ::CoCreateInstance(rclsid, pUnkOuter, dwClontext,
                             __uuidof(T), (void**)*p);
}
```

```
HRESULT CoCoreateInstance (LPCOLESTR szProgID,
                           LPUNKNOWN pUnkOuter = NULL,
                           DWORD dwClsContext = CLSCTX_ALL);
```

注意前面代码中第一个 CoCreateInstance **方法是如何创建指定类的实例的。它把方法的参数传给了** COM API CoCreateInstance**，并且请求的初始接口是智能指针类支持的接口（这是表达式**__uuidof(T)**的用途）。第二个** CoCreateInstance **重载方法把输入的** ProgID **转换成** CLSID**，然后使用与第一个方法相同的方式创建实例。**

因此，下面的代码是等价的（虽然我个人认为使用智能指针的代码更容易理解一些）。第一个实例化请求明确地使用了 COM API CoCreateInstance**。第二个使用了智能指针的** CoCreateInstance **方法。**

```
ISpeaker* pSpeaker;
HRESULT hr =
::CoCreatelnstance (__uuidof (Demagogue), NULL, CLSCTX_ALL,

                    __uuidof (ISpeaker, (void**) &pSpeaker);
...   Use the interface
pSpeaker->Release 0 ;

CComPtr<ISpeaker> pSpeaker;
HRESULT hr = pSpeaker.CoCreatelnstance (__uuidof (Demagogue));
...  Use the interface. It releases when pSpeaker leaves scope
```

📖 CComPtr 和 CComQIPtr 的操作

因为智能接口指针的行为应该与原始接口指针尽可能保持一致，所以 CComPtr **类定义了一些操作符使得** CComPtr **对象的行为像一个指针。例如，当使用** operator*()**来解除对指针的引用时，我们希望得到指针所指的内容。所以，解除智能接口指针的引用应该得到内部接口指针所指的内容。它是这样做的：**

```
T& operator*() const { ATLASSERT(p!=NULL); return *p; }
```

- **注意，如果我们的组件是调试版本，那么当试图解除** NULL **智能接口指针的引用时，**operator*()**方法会友善地报告一个断言**(assertion)**。当然，我一直认为一般保护错**(GPF，General Protection Fault)**消息框等价于一个断言。但是，**ATLASSERT **宏**

会产生更多对程序员有利的、关于错误位置的提示。

为了保持指针的样子，取一个智能指针对象的地址（也就是调用 operator&()）应该返回内部裸指针的地址。注意这里的问题并不是实际返回的二进制数值。智能指针的状态仅包含一个裸接口指针。因此，智能指针和裸接口指针占据相同大小的存储空间。智能指针对象的地址和内部成员变量的地址是同一个二进制数。

如果没有重载 CComPtr<T>::operator&()，那么取一个实例的地址应该返回 CcomPtr <T>*。为了使一个智能指针类保持与 T*类型相同的指针语义，类的 operator&()方法必须返回 T**。

```
T** operatur&() { ATLASSERT(p==NULL)
```

注意当对一个非 NULL 智能接口指针取地址时这个操作符将进行断言(assert)，这是由于我们可能会解除对返回地址的引用,并且在正确地释放接口指针之前改写内部数据成员。这个断言用来保护指针的语义并且防止我们意外地破坏指针。但是，这个行为使我们不能用智能接口指针作为一个[in,out]的函数参数。

```
STDMETHODIMP SomeClass::UpdateObject (/* [in, out] */ IExpected**
   ppExpected);

CComPtr<IExpected> pE = /* Initialize to some value */ ;

pobj->UpdateObject (&pE); // Asserts in debug build: pE is non-NULL
```

如果我们确实想用这种方式使用智能接口指针，那么可以取成员变量的地址：

```
pobj->UpdateObject (&pE.p);
```

CComPtr 和 CComQIPtr 的资源管理操作。一个智能接口指针代表一个资源，虽然它试图正确地管理自己，但是，有时我们想显式地管理资源。例如，我们必须在调用 CoUninitialize 方法之前释放所有的接口指针。这意味着如果申请的对象是全局的或者静态的（或者是 main()中的局部变量），那么我们不能等到 CComPtr 的析构函数来释放接口指针。全局和静态变量的析构函数在主函数退出之后才执行，此时 CoUninitialize 早已完成。（如果这些析构函数能够运行的话。为了保证全局和静态变量的构造函数和析构函数能够执行，我们必须链接 C++运行库。ATL 本身并不使用 C/C++运行库；因此在缺省情况下，ATL 组件并不链接它。）

我们可以通过把智能指针赋值为 NULL 来释放内部的接口指针。也可以更明确地调用

Release 方法。

```
int main O
{
    HRESULT hr = CoInitialize (NULL);
    If (FAILED (hr)) return -1;  // Something is seriously wrong

    CComPtr<IUnknown> punk = /* Initialize to some object */ ;
    ...
    punk. Release O;          // Must Release before CoUninitialize!

    CoUninitialize(),
}
```

注意这段代码*调用了智能指针对象的* CComPtr<T>::Release ***方法*，因为它使用了点操作符来引用对象。*它并不象我们期望的那样，直接调用内部接口指针的*** IUnknown::Release ***方法*。智能指针的** CComPtr<T>::Release **调用内部接口指针的** IUnknown::Release **方法并且把内部的接口指针置为** NULL**。这样可以防止析构函数再一次释放接口指针。下面是智能指针的** Release **方法：**

```
template <class T> void CComPtr<T>::Release()
{
    IUnknown* pTemp = p;
    if (pTemp) { p = NULL; pTemp->Release(); }
}
```

为什么 CComPtr<T>::Release **不是简单地使用成员变量** p **调用** IUnknown::Release **方法呢？原因并不显然。相反，它把接口指针成员变量复制到一个临时变量中，并把成员变量置为** NULL**，然后使用临时变量释放接口指针。这个方法避免了智能指针保存的接口被释放两次。**

例如，假设智能指针是类 A **的成员变量，并且智能指针拥有对象** B **的一个引用。我们调用智能指针的.Release 方法。智能指针释放了对对象** B **的引用。对象** B **反过来保存了包含智能指针的类** A **的一个实例的引用。对象** B **决定释放对类** A **实例的引用。类** A **实例决定析构，析构过程中调用智能指针成员变量的析构函数。这个析构函数检测到接口指针非**

NULL，这样它会又一次释放接口。[3]

在 ATL3.0 版本发布之前，下面的代码可以成功地通过编译，并且将会两次释放接口指针。注意箭头操作符的用法。

```
Punk->Release ();              // Wrong! Wrong! Wrong!
```

在那些 ATL 的版本中，箭头操作符返回内部的接口指针。因此上面那行代码实际上调用了 IUnknown::Release 函数，而不是我们期望的 CComPtr<T>::Release 方法。这使得接口指针成员函数仍然为非 NULL，所以最终析构函数会第二次释放接口指针。

这是一个不容易被发现的错误。智能指针鼓励我们把它的实例当作接口指针。但是在这种特殊的情况下，我们不能使用箭头操作符（如果它确实是一个指针的话，我们可以使用），必须使用点操作符，因为事实上它是一个对象。更糟的是，当我们犯错误时编译器不能告诉我们。

在 ATL 的 3.0 版本中情况有了变化。注意现在箭头操作符的定义是返回一个 _NoAddRefReleaseOnCComPtr<T>* 值。

```
_NoAddRefReleaseOnCComPtr<T>* operator->() const
{
   ATLASSERT(p!=NULL); return (_NoAddRefleaseOnCComPtr<T>*)p;
}
```

这是一个简单的模板类，它唯一的目的是使得 AddRef 和 Release 不可访问。

```
template <calss T>
class _NoAddRefReleaseOnCComPtr : public T
{
   private:
        STDMETHOD_(ULONG, AddRef)()=0;
        STDMETHOD_(ULONG, Release)()=0;
};
```

模板类 _NoAddRefReleaseOnCComPtr<T> 从被返回的接口派生。因此，它继承了该接口的所有方法。然后它重载了 AddRef 和 Release 方法，使得它们成为私有的和纯虚的。现在当我们使用箭头操作符调用这两个方法时，将会得到如下的编译错误：

```
error C2248: 'Release' : cannot access private member declared in
```

[3] 谢谢 Jim Springfield 指出了这一点。

```
cl ass 'ATL' '_NoAddRefRel easeOnCComPtr<T>'
```

CopyTo 方法。CopyTo 方法产生接口指针的一份经过 AddRef 之后的拷贝，并且把它放到指定的位置。因此，CopyTo 方法生成的接口指针的生命周期与被复制的智能指针无关。

```
HRESULT CopyTo(T** ppT)
{
    ATLASSERT(ppT != NULL);
    if (ppT==NULL) return E_POINTER;
    *ppT = p;
    if (p) p->AddRef();
    return S_OK;
}
```

我们经常使用 CopyTo 方法把一个智能指针复制到[out]参数中。一个[out]接口指针必须被返回指针的代码进行 AddRef。

```
STDMETHODIMP SomeClass::get_Object (/* [out] */ IExpected** ppExpected)
{
    // Interface saved in member m_object of type CComPtr<IExpected>
    return m_object.CopyTo (ppExpected) ; // Correctly AddRefs pointer
}
```

观察下面的代码。它所做的可能不是我们所希望的，而且是错误的。

```
STDMETHODIMP SomeClass:;get_Object (/* [out] */ IExpected** ppExpected)
{
    // Interface saved in member m_object of type CComPtr<IExpected>
    *pExpected : m_object ;  // Wrong! Does not AddRef pointer!
}
```

类型转换操作符。当把智能指针赋给一个裸指针时，我们隐式地调用了 operator T() 方法。换句话说，我们把智能指针强制转换为底层的类型。注意 operator T()并不对返回的指针进行 AddRef。

```
operator T*() const { return (T) p; }
```

因为在下面的情况下，我们不希望 AddRef。

```
STDMETHODIMP SomeClass::put_Object (/* [in] */ IExpected* pExpected);

// Interface saved in member m_object of type CComPtr<IExpected>
pObj->put_Object (m_object); // Correctly does not AddRef pointer!
```

Detach 和 Attach 方法。当我们想把一个 CComPtr 实例中接口指针的所有权交给一个等价的裸指针时，应该使用 Detach 方法。它返回内部的接口指针并且把智能指针置为 NULL，从而保证析构函数不会释放接口。调用 Detach 的客户负责释放接口。

```
T* Detach() { T* pt = p; p= NULL; retrn pt;
```

当我们把不再需要的接口指针返回给调用者时，我们经常会使用 Detach 方法。它并不“向调用者提供经过 AddRef 之后的接口拷贝，然后立即释放保存起来的接口指针”，而是简单地把引用传递给调用者，从而避免了多余的 AddRef/Release 调用。是的，这确实是一个小小的优化，但是非常简单。

```
STDMETHODIMP SomeClass::get_Object (/* [out] */ IExpected** ppExpected)

{
    CComPtr<IExpected> pobj = /* Initialize the smart pointer */ ;
    *ppExpected = pobj.Detach(); // Destructor no longer Releases
    return S_OK;
}
```

当我们想把一个裸指针的所有权交给一个智能指针时，应该使用 Attach 方法。它释放被智能指针保存的接口指针，然后把智能指针设置为裸指针。注意，这项技术同样避免了多余的 AddRef/Release 调用，并且是一个有效的小优化。

```
void Attach(T* p2) { if (p) p->Release(); p = P2; }
```

作为[out]参数接收到的裸指针，客户可以使用 Attach 方法来获取其所有权。提供[out]参数的函数把接口指针的所有权交给调用者。

```
STDMETHODIMP SomeClass::get_Object (/* [out] */ IExpected** ppObject);
```

```
void VerboseGetOption () {
    IExpected* p;
    pObj->get_Object (&p) ;

    CComPtr<IExpected> pE;
    pE.Attach (p);   // Destructor now releases the interface pointer
    // Let the exceptions fall where they may now!l!
    CallSomeFunctionWhichThrowsExceptions();
}
```

***其他智能指针方法。*智能指针类还提供了有效的、用于查询新接口的简捷语法：**QueryInteface **方法。它仅接受一个参数 —— 目标接口类型变量的地址。**

```
template <class Q>
HRESULT QueryInterface(Q** pp) const
{
   ATLASSERT(pp != NULL && *pp == NULL);
   return p->QueryInterface(__uuidof(Q), (void**)pp);
}
```

使用这个方法减少了犯以下常见错误的机会：查询一个接口（例如：IID_IBar**）但是为返回值指定不同类型的指针（例如：**IFoo***）。*

```
CComPtr<IFoo> pfoo = /* Initialize to some IFoo */
IBar* pbar;

// We specify an IBar variable so the method queries for IID_Ibar
HRESULT hr = pfoo,QueryInterface(&pbar);
```

使用 IsEqualObject **方法来判断两个接口指针是否引用同一个对象。**

```
bool IsEqualObject(IUnknown* pOther);
```

它完成对 COM **实体身份的同一性测试：查询每个接口的** IUnknown **并且比较查询的结果。当查询** IUnknown **接口时，一个** COM **对象总是返回相同的指针值。**IsEqualObject

方法对 COM **身份同一性测试有所扩展。它认为两个** NULL **接口指针是等同的对象。**

```
bool SameObjects(IUnknown* punkl, IUnknown* punk2)
{
    CComPtr<IUnknown> p (punkl);
     return p. IsEqualObject (punk2);
}

IUnknown* punkl = NULL;
IUnknown* punk2 = NULL;
ATLASSERT (SameObjects(punkl, punk2); // true
```

SetSite **方法把内部指针引用的对象与一个** site **对象（通过参数** punkParent **指定）联系起来。智能指针必须指向一个实现了** IObjectWithSite **接口的对象。**

```
HRESULT SetSite(IUnknown* punkParent);
```

Advise **方法把一个连接点接收器对象与智能接口指针引用的对象（事件源对象）联系起来。第一个参数是接收器接口，第二个是接收器接口的** ID**。第三个参数是输出参数，**Advise **方法通过这个参数返回一个标记值来唯一标识这个连接。**

```
HRESULT Advise(IUnknown* pUnk,const IID&  iid, LPDWORD pdw);

CComPtr<ISource> ps /* Initialized via some mechanism */ ;
ISomeSink* psink = /* Initialized via some mechanism */ ;
DWORD dwCookie;

ps->Advise (psink, __uuidof(ISomeSink), &dwCookie);
```

没有用来终止连接的智能指针方法 Unadvise**，因为指针不需要** Unadvise**。为了终止连接，我们只需要** cookie**、接收器接口标识符（**IID**）和一个事件源的引用。**

2.5.3　CComPtr 比较操作符

智能指针有三个操作符提供比较操作。当内部接口指针为 NULL **时，**operator!()**方法返回** true**。当比较操作数等于内部接口指针时，**operator==()**方法返回** true**。**operator<()**方法一**

般来说没什么用，因为它比较两个接口指针的二进制数值。但是，为了使这些类实例的 STL 集合能够正常工作，它需要这些比较操作符。

```
bool operator!() const       { return (p == NULL); }
bool operator<(T* pT) const  { return p < pT; }
bool operator==(T* pT) const { return p == pT) }
```

- 注意：当使用智能指针时，我们必须为比较操作改变代码风格。有些程序员喜欢在编写比较相等的代码时，把直接量(如常量)放在表达式的左边。ATL 智能接口指针在这种方式下不能正常工作。例如：

```
CComPtr<IFoo> pFoo;

if (!pFoo)          // Tests for pFoo.p == NULL using operator!
if (pFoo == NULL)    // Tests for pFoo.p == NULL using operator==
if (NULL == pFoo)    // Does not compile!
```

2.5.4 CComDispatchDriver 类

在 ATL 3.0 中，CComDispatchDriver 或多或少是 CComQIPtr<IDispatch>的特例，但是它做得更好。首先，它是一个独立的类。它并不是真的从 CComQIPtr 派生，因此 CcomDispatchDriver 类完全独立地实现了 CComQIPtr 智能指针的所有功能。遗憾的是，有些对 CComQIPtr 的改进并没有移植到这个类中。在 ATL 的下一个版本中，CComDispatchDriver 被 typedef 为 CComQIPtr<IDispatch>的特化版本。

注意 CComDispatchDriver 类的状态仅由一个成员变量 —— IDispatch* p 组成。

```
Class CcomDiaspatchDriver
{
public:
     IDispatch* p;
```

这个类包含了典型的智能指针方法，所以我只讨论一些与 CComPtr 和 CComQIPtr 类有重要区别的方法。

奇怪的实现习惯

我们可以使用 IDispatch 指针和 IUnknown 指针来初始化 CComDispatchDriver 的实例。在后者的情况下，构造函数会从指定的指针中查询 IDispatch 接口。与 CComPtr 和 CComQIPtr 类不同，我们不能使用一个 CComDispatchDriver 实例来初始化另一个实例。

```
CComDispatchDriver();
CComDispatchDriver(IDispatch* lp);
CComDispatchDriver(IUnknown* lp);
```

注意 operator->()方法返回 IDispatch*。

```
IdISPATCH* OPERATOR-> {ATLASSERT(p!= NULL); return p; }
```

它应该象其他智能指针类一样，返回一个_NoAddRefReleaseOnCComPtr<IDispatch>指针。这意味着我们仍然可以使用箭头（->）操作符方法，并且在没有清除成员变量 p 的情况下调用内部接口指针的 Release。稍后析构函数会再次释放这个接口指针。

还有一个不太重要的区别，它的 operator!()方法不像其他智能指针类返回 bool 类型，而是返回 BOOL 类型。BOOL 类型在 Windows 中是 int 的 typedef（在 Win32 系统上是一个 32 位的值），一般的语义是非 0 表示 TRUE，0 表示 FALSE。bool 值是 C++固有的类型，它是一个取值为 true 和 false 的字节值。

```
BOOL operator!() {return (p == UNLL) ? TRUE : FALSE:}
```

属性访问器及其变种

有几个方法使得用对象的 IDispatch 接口获取和设置对象的属性简单了很多。首先我们根据属性名字的字符串，通过调用 GetIDOfName 方法能够获得属性的 DISPID。

```
HRESULT GetIDOfName(LPCOLESTR lpsz, DISPID* pdispid);
```

一旦拥有了属性的 DISPID，我们就可以利用 GetProperty 和 PutProperty 方法来获取和设置属性的值。我们指定属性的 DISPID 来获取或者设置属性，并且在一个 VARIANT 结构中发送或者接收新值。

```
HRESULT GetProperty(DISPID dwDispID, VARIANT* pVar);
HRESULT putProperty(DISPID dwDispID, VARIANT* pVar);
```

我们可以跳过第一步，通过命名良好的 GetPropertyByName 和 PutPropertyByName 方法仅给出属性名字来获取和设置属性。

```
HRESULT GetProperty(LPCOLESTR lpsz, VARIANT* pVar);
HRESULT putProperty(LPCOLESTR lpsz, VARIANT* pVar);
```

方法调用的辅助函数

使用 IDispath::Invoke 方法来调用一个对象的方法非常痛苦。我们必须把所有的参数装进 VARIANT 结构，建立这些 VARIANT 的一个数组，并且把方法的名字翻译成 DISPID。这一切并不困难，但乏味之极并且容易出错。

CComDispatchDriver 类有一些为“使用 IDispatch 调用一个对象的方法时经常出现的情况”而定制的方法。它们有四种基本的变化：

- 通过 DISPID 或者名字调用一个方法，不传递参数。
- 通过 DISPID 或者名字调用一个方法，传递一个参数。
- 通过 DISPID 或者名字调用一个方法，传递两个参数。
- 通过 DISPID 或者名字调用一个方法，传递一个包含 N 个参数的数组。

每种变化都需要被调用方法的 DISPID 或者名字、参数和一个可选的返回值。

```
HRESULT Invoke0(DISPID dispid, VARIANT* pvarRet = NULL)
HRESULT Invoke0(LPCOLESTR lpszName, VARIANT* pvarRet = NULL)
HRESULT Invoke1(DISPID dispid, VARIANT* pvarParam1,
                VARIANT* pvarRet = NULL)
HRESULT Invoke1(LPCOLESTR 1pszName,
        VARIANT* pvarParam1, VARIANT* pvarRet = NULL)
HRESULT Invoke2(DISPID dispid,
         VARIANT* pvarParam1, VARIANT* pvarParam2,
         VARIANT* pvarRet = NULL)
HRESULT Invoke2(LPCOLESTR lpszName,
         VARIANT* pvarParam1, VARIANT* pvarParam2,
         VARIANT* pvarRet = NULL)
HRESULT InvokeN(DISPID dispid,
         VARIANT* pvarParams, int nParams, VARIANT* pvarRet = NULL)
HRESULT InvokeN(LPCOLESTR lpszName,
         VARIANT* pvarParams, int nParams, VARIANT* pvarRet = NULL)
```

最后还有两个静态成员函数：GetProperty 和 PutProperty。我们可以利用这些方法通过 DISPID 来获得和设置一个属性，而不必首先把 IDispatch 指针附着到一个 CComDispatchDriver 对象上。

```
static HRESULT GetProperty(IDispatch* pDisp, DISPID dwDispID,
                           VARIANT* pVar);
static HRESULT PutProperty(IDispatch* pDisp, DISPIN dwDispID,
                           VARIANT* pVar);
```

这里有一个例子：

```
HRESULT GetCount (IDispatch* pdisp, long* pCount)
{
    *pCount = O;
    const int DISPID_COUNT = 1;

    CComVariant v;
    CComDispatchDriver::GetProperty (pdisp, DISPID_COUNT, &v);

     HRESULT hr = v. ChangeType (VT I4);
     If (SUCCEEDED (hr))
          *pCount = V_I4(&v) ;
     return hr;
}
```

2.6　总结

ATL 为管理 COM 程序员经常使用的数据类型提供了丰富的类的集合。字符串转换宏提供了各种正文类型之间的有效转换，但是我们必须小心不要在循环中使用。

当使用 BSTR 字符串类型时，我们必须格外小心，因为它有许多特殊的语义。ATL CComBSTR 类为我们管理了许多特殊的语义并且非常有用。但是，这个类并不能弥补这样的事实：对 C++编译器而言，OLECHAR*和 BSTR 是同一种类型。我们要小心使用 BSTR 类型，因为编译器对许多错误并不会发出警告。

CComVariant 类实际上提供了 CComBSTR 类同样的优点，但只是针对 VARIANT 结构

而已。如果我们正在使用 VARIANT 结构，或者迟早要使用 VARIANT 结构，那么我们应该使用 ATL 的 CComVariant 智能 VARIANT 类。我们产生的资源泄漏问题会少得多。

CComPtr、CComQIPtr 和 CComDispatchDriver 智能指针类减轻了(并不是完全解决了)接口指针所需要的资源管理问题。这些类有许多有用的方法，这些方法有助于我们编写更多的应用程序代码，同时处理更少的底层资源管理的细节。当接口指针和能够抛出异常的代码一起使用时，您会发现智能指针最为有用。