**BSTR用法详解**

链接：<https://blog.csdn.net/qq_21232339/article/details/51220592>

**BSTR 详解一 - BSTR 简介和内部结构**

**1          Why need BSTR**

COM 是一种跨编程语言的平台，需要提供语言无关的数据类型。多数编程语言有自己的字符串表示。

·                     C++ 字符串是以 0 结束的 ASCII 或 Unicode 字符数组

·                     Visual Basic 字符串是一个 ASCII 字符数组加上表示长度的前缀。

　　·                     Java 字符串是以 0 结束的 Unicode 字符数组。

　　需要定义一种通用的字符串类型，可以很容易的匹配到不同编程语言。 **在 C++ 中，就是 BSTR**。

**2          What is BSTR**

***2.1*      *BSTR 简介***

" **Basic STRing**" 的简称，微软在 COM/OLE 中定义的标准字符串数据类型。对于 C++ ， Windows 头文件 wtypes.h 中定义如下：

typedef wchar\_t WCHAR;

typedef WCHAR OLECHAR;

typedef OLECHAR \_\_RPC\_FAR \*BSTR;;

***2.2*      *BSTR 实现***

在 COM 中，字符用 16-bit OLECHAR 表示，这样使 COM 可以支持各种 code pages ，包括 Unicode 。对于 windows 系统，可以简单理解为 OLECHAR 使用的就是 Unicode 。 OLECHAR 串与单字节字符串很类似，是一个以 null 结尾的 buffer 。唯一的区别是每个字符占两个字节，而不是一个

 0 1 2 3 4 5 6 7 8 9 0 1

| H | E | L | L | O | \0|

 ^

 OLCHAR

Figure 1. Format of an OLECHAR string.

使用以 Null 结尾的简单字符串在 COM component 间传递不太方便。因此， **标准 BSTR 是一个有长度前缀和 null 结束符的 OLECHAR 数组。**BSTR 的前 4 字节是一个表示字符串长度的前缀。 BSTR 长度域的值是字符串的字节数，并且不包括 0 结束符。由于是 Unicode 串，所以字符数是字节数的一半。这种方式的优点是允许程序员在 BSTR 串中间嵌入 NULL 字符。但是， BSTR 的前四个字节表示长度，而 OLECHAR 数组的前四字节表示前两个字符。这种情况下，对于 C++ 程序，如何实现 BSTR 和 OLECHAR 的交换？答案是 COM 提供了两个 BSTR 分配用的 API ： SysAllocString / SysReallocString 。函数返回的指针指向 BSTR 的第一个字符，而不是 BSTR 在内存的第一个字节。

 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

0a000000 | H | E | L | L | O | \0|

         ^

         BSTR

Figure 2.  Format of a BSTR.

下面是 SysAllocString 和 SysFreeString 的伪代码。

BSTR SimpleSysAllocString( const OLECHAR \* sz)

{

    if ( sz == NULL) return NULL;

    BYTE\* buf = new BYTE[sizeof(INT32) + (wcslen(sz)+1)\*sizeof(OLECHAR) ];

    if(buf == NULL)

    {

        return NULL;

    }

    else

    {

        INT32 len = wcslen(sz) \* sizeof(OLECHAR);

        \*((INT32\*) buf) = len;

        wcscpy( (WCHAR\*)(buf+sizeof(INT32)), sz);

        return (BSTR)(buf+sizeof(INT32));

    }

}

VOID SimpleSysFreeString( BSTR bstr)

{

    if(bstr != NULL)

    {

       BYTE\* start = (BYTE\*)bstr - sizeof(INT32);

       delete []start;

    }

}

Trackback: http://tb.blog.csdn.net/TrackBack.[asp](http://biancheng.dnbcw.info/asp/)x?PostId=1486331

**BSTR 详解二 - 使用时机**

**3          When to use BSTR**

**只有在你不得不用的时候。**

**使用 BSTR 一般有以下几种情况：**

·                     COM interface 接口定义，并且不希望额外提供 custom marshaling 库（ MDIL 生成或开发人员自己订制），必须使用 BSTR 传递字符串。使用 C/C++ 类型的字符串在 COM DLL 传递字符串，表面上可以使用，但违背了 COM 的基本规则，并且给以后的扩展留下了隐患。例如，把一个 In-process COM Object( 简单说 COM DLL) 改成 out-of-process object （ COM EXE ）。理论上，客户端的代码应该不做任何改变。但如果是用了 C/C++ 字符串，又希望只使用系统的 automation mashaller （ Oleaut32.dll ），就会出错。

·                     如果可以提供 custom marshaling ，也推荐使用 BSTR 。

·                     客户要求接口必须使用 BSTR ，和客户讨论后，不能修改。

·                     使用的外部库的接口使用 BSTR

**不使用的情况：**

·                     不推荐在 IDL 结构体中定义 BSTR 成员，会给结构体的复制和释放带来麻烦。最好直接使用限定最大长度的 TCHAR 数组。如果确实需要传递变长字符串， BSTR 应该被定义成独立的参数或者使用独立的 get/set 接口。

·                     尽可能缩小的 BSTR 及相关类型的作用域范围。类的成员变量和函数参数不使用 BSTR 。局部变量要尽快释放类的内部不使用 BSTR 。代码处理逻辑中只在接口直接相关部分使用 BSTR 。接收到一个 BSTR 时，尽量立刻变成 C/C++ 的字符串副本进行处理。在需要传递 BSTR 参数前产生 BSTR ，用过立即释放。

**字符串相关类型的推荐选择顺序**

|  |  |  |
| --- | --- | --- |
| 优先级 | 类型 | 说明 |
| 最高 | stl::string/wstring | ·          功能最完善，可移植性最好。 |
|  | CString | ·          **如果编码规范限制使用 STL 的时候，推荐 CString 。**  ·          VC 6 的版本很不完善。 .Net 有明显改进，需要进一步研究。 |
|  | C/C++ basic type （ TCHAR\* / char\* / LPTSTR / LPCTSTR / TCHAR[] ） | ·          在结构体中，优先使用指定最大长度的字符数组。  ·          效率最好 |
|  | CComBSTR/ \_bstr\_t | ·          在必须使用 BSTR 时的优先选择。  ·          在 ATL （ COM component ）工程或者工程中必须使用 ATL 中，优先选择 CComBSTR 。一般 Exe/dll 如果 \_bstr\_t 能满足要求，优先使用 \_bstr\_t 。  ·          对于 VC6 ，使用 \_bstr\_t 一定要慎重，最好只用作简单临时变量保存调被调用函数的传入参数。因为 \_bstrt\_t 不能支持一些关键性操作，比如 Detach 。  ·          对于 VC++ .Net 推荐使用 \_bstr\_t ，它是 C++ 扩展，不需要额外包含 ATL 的文件。 |
| 最低 | BSTR | ·          COM 接口 |

**BSTR 详解三 - BSTR 使用注意事项**

**1          How to use BSTR**

***1.1*      *BSTR 分析***

BSTR 设计对于 C++ 程序员好坏参半。一方面， BSTR 可以被用于大多数需要 OLECHAR 数组作为参数的函数。另一方面，不能用熟悉的 C/C++ 函数进行对 BSTR 的分配、释放和处理，例如 malloc, free, new, delete, lstrcat, and lstrlen 等函数不能用于处理 BSTR 。就像对接口指针和类指针的处理不一样，对 BSTR 的处理和对 TCHAR\* 的处理也不一样。 BSTR 是一种 C 语言方式的类型定义方式，这种定义方式提高了 BSTR 在 C++ 的应用效率，但是也带来了很多的潜在风险，它使程序员失去了利用编译器检查潜在问题的机会。

***1.2*      *BSTR 使用基本规则***

·                     在对 BSTR 进行读取操作的时候，可以把 BSTR 看作 OLECHAR 数组。 BSTR 可以用于 const wchar\_t\*(LPCTSTR/ LPCWSTR/ cosnt TCHAR\*/ cosnt WCHAR\* in Unicode project) ，不能用于需要 wchar\_t\* (LPTSTR/ LPWSTR/ TCHAR\*/ WCHAR\* in Unicode project) 的地方。

·                     如果有相应的 BSTR 处理函数，必须使用 BSTR 处理函数，不要使用普通字符串函数。特别是一个 BSTR 包含多个字符串 ( 也就是，包含多个 0 结束符 ) 的情况。在对 BSTR 进行修改（包括创建和释放时），必须使用 BSTR 的专用函数。主要要保证对字符长度前缀的正确修改。不要直接读取 BSTR 的长度域，应该使用 BSTR 处理函数计算长度。

|  |  |
| --- | --- |
| String Manipulation Functions | Descriptions |
| SysAllocString | Creates and initializes a string. |
| SysAllocStringByteLen | Creates a zero-terminated string of a specified length. |
| SysAllocStringLen | Creates a string of a specified length. |
| SysFreeString | Frees a previously created string. |
| SysReAllocString | Changes the size and value of a string. |
| SysReAllocStringLen | Changes the size of an existing string. |
| SysStringByteLen | Returns the length of a string in bytes. |
| SysStringLen | Returns the length of a string. |

·                     NULL 是 BSTR 的有效值。按照约定，它可以被看作含有 0 个字符的字符串。 BSTR 变量必须等于 NULL ，或者正确分配的 BSTR 指针。在改变 BSTR 变量的之前，必须释放原来指向的 BSTR 。不要把 BSTR 直接初始化成常量字符指针，例如， BSTR bs = L”” 。

·                     Automation 会 cache BSTR 使用的空间，以提高 SysAllocString/SysFreeString 的性能，会给测试发现问题带来困难。如果可能推荐在调试时使用 Compuware DevPartner 7.x 及更高版本的工具。

***1.3*      *BSTR 参数使用***

多数时候， BSTR 是被用于函数参数。关于 BSTR 参数的使用规则是 BSTR 类型的基础。只有熟练掌握，才能分析 warpper 类或转换函数的正确性。

**基本原则：在给 by-reference[in/out] 参数赋一个新的值前，被调用者负责释放。其他情况，都是调用者负责释放。**

**调用者使用 BSTR 的规则如下：**

·          释放被调用函数返回的 BSTR ，或者被调用函数通过 by-reference 返回的 BSTR 。

HRESULT IWebBrowser2::get\_StatusText( BSTR FAR\* pbstr );

//...

BSTR bstrStatus;

pBrowser->get\_StatusText( &bstrStatus );

// shows using the Win32 function

// to freee the memory for the string:

::SysFreeString( bstrStatus );

·          释放通过 by-value 方式传给其他函数的 BSTR.

//.h

HRESULT IWebBrowser2::put\_StatusText( BSTR bstr );

//.cpp

// shows using the Win32 function

// to allocate memory for the string:

BSTR bstrStatus = ::SysAllocString( L"Some text" );

if (bstrStatus == NULL)

   return E\_OUTOFMEMORY;

pBrowser->put\_StatusText( bstrStatus );

// Free the string:

::SysFreeString( bstrStatus );

//...

**被调用者按照如下规则处理 BSTR ：**

·          如果一个 BSTR 参数是 by-reference 方式，在给参数赋新值之前， Free 以前的值。如果没有给参数赋的新值，不要 Free 传入值。

void RefreshBSTR(BSTR& bs)

// bs is an [in/out] parameter. BSTR\* is the same

{

// using the bs here

Dosomething(bs);

// if (bs is about to be updated)

ASSERT(bs != NULL);

::Sys **Re**allocString(bs, \_T(“NEW STRING”));

// Sys **Re**allocString will call SysFreeString and

// SysAllocString in sequence

// If bs is only [out] parameter, SysAllocString

// should be called here.

}

·          不要 Free 通过 by-value 传入的 BSTR 。

void SetBSTR(BSTR bs)

// bs is an [in] parameter. BSTR\* is the same

{

// using the bs here

Dosomething(bs);

::SysFreeString(bs); //ERROR

}

·          不要 Free 返回给调用者的 BSTR .

BSTR GetBSTR1()

{

BSTR bs = ::SysAllocString(\_T(“test”));

::SysFreeString(bs); //ERROR

return bs;

}

void GetBSTR2(BSTR\* pBs)

{

CComBSTR bs(\_T(“test”));

\*pBS = (BSTR) bs; //ERROR: pBS will be freed automatically

}

·          如果需要保存传入的 BSTR ，被调用着需要用 SysAllocString() 生成一个新的副本，并保存。输入的 BSTR 会被调用者释放。

void MyClass::SetBSTR(BSTR bs)

{

//BSTR m\_bs;

m\_bs = bs; //ERROR

m\_bs = ::SysReAllocString(bs);

}

·          如果需要返回一个已经存储的 BSTR ，返回 BSTR 的一个拷贝。调用者释放返回的 BSTR 拷贝。

void MyClass::GetBSTR(BSTR\* pbs)

{

//BSTR m\_bs;

\*pbs = m\_bs; //ERROR

\*pbs = ::SysAllocString(m\_bs);

}

**BSTR 详解四 - BSTR 包容类**

***1.1*      *Programming with CComBSTR***

**1.1.1       概述**

CComBSTR 是 ATL 提供的 BSTR 包装类，是 VC 6 中提供的最完善的 BSTR wrapper 。就像 MFC CString 提供了对 TCHAR 的封装， CComBSTR 提供了对 BSTR 的封装。 Table 1 CComBSTR Methods 列出了 CComBSTR 的主要方法。

**Table 1 CComBSTR Methods**

|  |  |
| --- | --- |
| CComBSTR Method | Description |
| CComBSTR | 多个版本的构造函数用来创建新的 BSTR 。可以使用的参数包括 LPCOLESTR, LPCSTR, CComBSTR 。 |
| ~CComBSTR, Empty | 释放内部封装的 BSTR. |
| Attach, Detach, Copy | Attach 把一个已经存在 BSTR 加入类中。 Detach 把劣种的 BSTR 剥离，以便在超出作用域的时候，析构函数不会释放 BSTR 。 Detach 用于把 CComBSTR 赋给 [out] 参数。  Copy 用于产生一个 BSTR 的副本。一般用于用于把 CComBSTR 内容赋给 [out] 参数。 |
| operator BSTR, operator& | 允许直接操作内部的 BSTR 。 operator BSTR 用于把 CComBSTR 传给 BSTR 输入 [in] 参数。 operator& 用于把 CComBSTR 传给 BSTR\* 类型输出 [out] 参数。 |
| operator=, operator+=, operator<, operator==, operator> | 重载运算符，用于赋值、字符串连接、简单比较。 |
| Append, AppendBSTR | 字符串连接 |
| Length | 计算字符串长度 |
| LoadString | 利用字符串资源初始化 BSTR 。 |
| ToLower, ToUpper | 字符串大小写转换。 |
| WriteToStream,ReadFromStream | 从 IStream 中读 / 写 BSTR 。 |

下面的伪代码展示了 CComBSTR 的典型用法：

 HRESULT CMyObject::MyMethod(IOtherObject\* pSomething)

{

    CComBSTR bstrText(L"Hello");

    bstrText += " again";                     // LPCSTR conversion

    bstrText.ToUpper();

    pSomething->Display(bstrText);            // [in] parameter

    MessageBoxW(0, bstrText, L"Test", MB\_OK); // Assumes Windows NT

}

对于熟悉 MFC 的程序员， CComBSTR 让人失望。很多 CString 提供的方便的特性 CComBSTR 都没有提供。重要的缺省列在了 Table 2  Notable CComBSTR Omissions 中。简而言之， CComBSTR 没有提供完整的字符串操作。它的主要用途是把 LPCTSTR 转换成 BSTR ，同时提供一个操作 BSTR 的类，使程序员可以不使用 COM SysXXXXString APIs 。如果需要使用复杂的字符串操作，可以使用 STL 提供的 wstring 类。

**Table 2  Notable CComBSTR Omissions**

|  |  |
| --- | --- |
| Features Not Included in CComBSTR | Explanation |
| LPCSTR extraction | CComBSTR 可以把一个单字节字符串转换成 BSTR ，但是没有提供反向转换的功能。 \_bstr\_t 提供了 LPCTSTR operator 。 |
| String manipulation (including Replace, Insert, Delete, Remove, Find, Mid, Left, Right, and so on) | CComBSTR 没有提供这些方法。如果需要，可以使用 STL 中的 wstring 。 |
| Language-sensitive collation | CComBSTR 提供的字符串比较 (<, >, ==) 按照是 byte-by-byte 方式进行的。没有提供语言相关的比较 (language-specific collation) 。如果需要可以使用 wstring. |

**1.1.2       CComBSTR 注意事项**

使用 CComBSTR 时需要考虑的问题。

·                      CComBSTR **初始化**

CComBSTR 提供了一个长度初始化函数， CComBSTR(int nSize) 。所以简单给 CComBSTR 初始化成 NULL 会发生意想不到的调用。

// CComBSTR(int nSize) is called 。

CComBSTR bstr1 = NULL;

CComBSTR bstr2(NULL);

// CComBSTR(LPCOLESTR pSrc) is called.

CComBSTR bstr3 = static\_cast< LPCOLESTR>(NULL);

CComBSTR bstr4(static\_cast< LPCOLESTR>(NULL));

上面的例子中， bstr1/bstr2 被初始化成长度为 0 的 BSTR ，也就是说 CComBSTR::m\_str 是有内容的。 bstr3/bstr4 的值被初始化成 NULL ，也就是说 CComBSTR::m\_str == 0 。这样， bstr1/bstr2 在被赋新的值前需要考虑是否需要释放其中的 BSTR 。

·                      **字符集转换**

尽管某些 CComBSTR 方法可以自动把 ANSI 字符串转换成 Unicode 。所有的接口返回的都是 Unicode 字符串。如果需要转回 ANSI ，可以使用 ATL 或 MFC 转换类，或者 Windows API 。如果使用文字串修改 CComBSTR ，使用宽字节字符串。可以减少不必要的转换。例如：

// Declare a CComBSTR object. Although the argument is ANSI,

// the constructor converts it into UNICODE.

CComBSTR bstrMyString( "Hello World" );

// Convert the string into an ANSI string

CW2CT szMyString( bstrMyString );

// Display the ANSI string

MessageBox( NULL, szMyString, \_T("String Test"), MB\_OK );

// The following converts the ANSI string to Unicode

CComBSTR bstr("Test");

// The following uses a Unicode string at compile time

CComBSTR bstr(L"Test");

·                      **变量作用域 (Scope)**

象所有设计完整的类一样， CComBSTR 会在离开作用域的时候释放资源。如果一个函数返回一个指向 CComBSTR 的指针，可能会带来问题：指针有可能指向已经被释放的内存。此时应该使用 Copy 或 Detach 方法。参考下面的例子。

HRESULT CMyObject::MyMethod3(/\*[out, retval]\*/ BSTR\* pbstr)

{

    CComBSTR bstrText(L"Hello");

    bstrText += " again";

    \*pbstr = bstrText;        // No! Call Detach instead!

}

通过复制语句 \*pbstr = bstrText ，被 bstrText 封装的 BSTR 的指针作为传出 [out] 参数传递。在 MyMethod3 return 时， bstrText 离开作用域， CComBSTR destructor 毁掉用 SysFreeString 释放这个 BSTR 。因此，调用者得到了一个指向已经被释放的内存的指针，可能导致意想不到的结果。因为 bstrText 即将超出作用域，所以必须使用 CComBSTR Copy 或 Detach 给 \*pbstr 赋值。 CComBSTR Copy 生成字符串的一格副本， Detach 简单的把 BSTR 移出包装类。这样，在 bstrText 离开作用域的时候就不会被释放。

HRESULT CMyObject::MyMethod4(/\*[out, retval]\*/ BSTR\* pbstr)

{

    CComBSTR bstrText(L"Hello");

    bstrText += L" again";

    //\*pbstr = bstrText.Copy();    // Better!

    \*pbstr = bstrText.Detach();    // Much better!

}

在这个例子中，从效率考虑，最好使用 Detach 而不是 Copy 。 Detach 不需要产生一个额外副本的开销。当 CComBSTR 必须在复制之后保持自己的内容的时候，例如 CComBSTR 是一个成员变量，必须使用 Copy 。

·                      **显式释放 CComBSTR 内容**

程序员可以在 CComBSTR 超出作用域范围前显示释放 CComBSTR 中的字符串。一旦释放了， CComBSTR 内容就无效了。 CComBSTR 提供了 operator BSTR ，所以代码中可以显示的释放其中的 BSTR 。

HRESULT CMyObject::MyMethod1()

{

CComBSTR bstrText(L"This is a test");

    ::SysFreeString(bstrText);

// The string will be freed a second time

// when the CComBSTR object goes out of scope,

// which is invalid.

// CComBSTR::Empty() should be used in order to

// explicitly free the BSTR

}

在这段代码中， bstrText 中的 BSTR 被释放了。但是， bstrText 仍然没有超出作用域，看起来仍然可以使用。当 bstrText 最终超出作用域的时候， SysFreeString 被第二次调用。为了防止这种意外，需要把 operator BSTR 从类中删除。但这样没有办法把它用于需要 BSTR 类型输入 [in] 参数的地方，会使 CComBSTR 几乎没有任何用处。

·                      **外部 CComBSTR 用作 [out] 参数**

把一个已经初始化好的 CComBSTR 的地址传给一个函数作为 [out] 参数会导致内存泄漏。当把 CComBSTR 用于 BSTR\* 类型的传出参数 [out] 时，必须首先调用 Empty 方法清空字符串的内容。

HRESULT CMyObject::MyMethod2(ISomething\* p)

{

    CComBSTR bstrText;

    bstrText = L"Some assignment";     // BSTR is allocated.

    bstrText.Empty();                  // Must call empty before

    pSomething->GetText(&bstrText);    // using as an [out] parameter.

    if(bstrText != L"Schaller")

        bstrText += "Hello";           // Convert from LPCSTR.

}

在把 CComBSTR 作为 [out] 参数传递前，调用 Empty 释必须的。因为按照 COM 标准中的 [out] 参数的使用规则 - 被调用方法不应该在覆盖 BSTR 的内容前调用 SysFreeString 。如果你忘记调用 Empty ，调用前 BSTR 的内容占用的资源就会泄漏。

对于相同的代码，如果参数类型是 [in, out] ，就不会有泄漏。因为函数会在复制之前， Free 原有的串。

·                      **用 CComBSTR 给 BSTR 变量赋值**

在下面的代码中， CStringTest 使用 CComBSTR 作为成员变量保存 BSTR 属性。

class CStringTest

{

    CComBSTR m\_bstrText;

// IStringTest

public:

    STDMETHOD(put\_Text)(/\*[in]\*/ BSTR newVal)

    {

        m\_bstrText = newVal;

        return S\_OK;

    }

    STDMETHOD(get\_Text)(/\*[out, retval]\*/ BSTR \*pVal)

    {

        \*pVal = m\_bstrText;    // Oops! Call m\_bstrText.Copy

                               // instead.

        return S\_OK;

    }

};

由于 m\_bstrText 在 get\_Text 结束没有超出作用域，你可能认为在 the \*pVal = m\_bstrText 赋值时，不需要调用 Copy 。这是不对的。按照 COM 规则，调用者负责释放传出 [out] 参数的内容。由于 \*pVal 指向了 m\_bstrText 封装的 BSTR ，而不是一个副本，调用者和 m\_bstrText 析构函数都会企图删除字符串。

·                      **循环中使用 CComBSTR Objects**

尽管 CComBSTR 可以分配 buffer 完成一些操作，例如： += operator 或 Append 。但是，不推荐在一个小循环内部使用 CComBSTR 完成字符串操作。这种情况下， CString 能提供更好的性能。

// This is not an efficient way

// to use a CComBSTR object.

CComBSTR bstrMyString;

while (bstrMyString.Length()<1000)

{

   bstrMyString.Append(L"\*");

}

***1.2*      *\_bstr\_t Class***

\_bstr\_t 是微软 C++ COM 扩展的一部分。 \_bstr\_t 封装了 BSTR 数据类型。 \_bstr\_t 通过 SysAllocString and SysFreeString 等 BSTR APIs 管理资源的分配和释放。 \_bstr\_t 提供了内部引用计数来减少额外负担。

|  |  |  |
| --- | --- | --- |
| Construction | Version |  |
| \_bstr\_t |  | Constructs a \_bstr\_t object. |
| Operations |  |  |
| Assign |  | Copies a BSTR into the BSTR wrapped by a \_bstr\_t. |
| *Attach* | *VC 7* | *Links a \_bstr\_t wrapper to a BSTR.* |
| copy |  | Constructs a copy of the encapsulated BSTR. |
| *Detach* | *VC 7* | *Returns the BSTR wrapped by a \_bstr\_t and detaches the BSTR from the \_bstr\_t.* |
| *GetAddress* | *VC 7* | *Points to the BSTR wrapped by a \_bstr\_t.* |
| *GetBSTR* | *VC 7* | *Points to the beginning of the BSTR wrapped by the \_bstr\_t.* |
| length |  | Returns the number of characters in the \_bstr\_t. |
| Operators |  |  |
| operator = |  | Assigns a new value to an existing \_bstr\_t object. |
| operator += |  | Appends characters to the end of the \_bstr\_t object. |
| operator + |  | Concatenates two strings. |
| operator ! |  | Checks if the encapsulated BSTR is a NULL string. |
| operator ==, !=, <, >, <=, >= |  | Compares two \_bstr\_t objects. |
| operator wchar\_t\* | char\* |  | Extract the pointers to the encapsulated Unicode or multibyte BSTR object. |
|  |  |  |

VC6 中 \_bstr\_t 缺少了几个重要的方法： **Attach/Detach/GetAddress/GetBSTR**，所以比 CComBSTR 简单，使得 \_bstr\_t 的应用场合非常有限。而且， \_bstr\_t 使用了引用计数在不同的对象间共享 BSTR ，内部实现比 CComBSTR 复杂。使用注意事项可以参考 CComBSTR 的类似函数。

建议只用于下面的情况：

·                      **BSTR 的作用域管理**

解决 BSTR 变量超出作用域范围的自动回收。 (1) 构造简单的 BSTR 对象，对 BSTR 进行基本字符串操作，作为输入 [in] 参数传递给被调用者。

{

\_bstr\_t bs1(L"first ");

    bs1 += L"second ";

SetBs(bs1); // void SetBs(BSTR bs)

}

(2) 作为 BSTR 的 wrapper ，解决 [out] 参数 BSTR 的生命周期之后的回收问题。

HRESULT BetterMethod()

{

    BSTR val = NULL;

    GetBs(&val); //void GetBs(/\* [out] \*/ BSTR\*)

\_bstr\_t bsVal(val, **false**);

    // false is IMPORTANT. Other constructor could

          // store the BSTR, too. But you must free the

          // BSTR later.

}

HRESULT GoodMethod()

{

    BSTR val = NULL;

    GetBs(&val); //void GetBs(/\* [out] \*/ BSTR\*)

// All the function create a copy of BSTR.

// But you must free the BSTR immediately.

        \_bstr\_t bsVal2(val);

        \_bstr\_t bsVal3;

        bsVal3 = val;

**SysFreeString(val);**

}

·                      **使用范围**

完成简单的 BSTR 字符串连接、比较等操作。

**BSTR 详解五 - BSTR 与其它字符串类型转换**

**1          类型转换**

常用字符串件的类型转换。

|  |  |  |
| --- | --- | --- |
| From | To | Sample |
| 字符串常量 | BSTR | **Right:**  BSTR bs = ::SysAllocString(\_T("Test string"));  …  ::SysFreeString();  **Wrong:**  BSTR bs = \_T("Test string"); //ERROR |
| LPWSTR /  LPCWSTR /  WCHAR\* /  wchar\_t | BSTR | **Right:**  LPCTSTR sz1 = \_T("Test String");  BSTR bs = ::SysAllocString(sz1);  …  ::SysFreeString();  **Wrong:**  LPTSTR sz1 = \_T("Test String");  BSTR bs = sz1; //ERROR |
| BSTR | LPCWSTR /  const WCHAR \* /  const wchar\_t \* | **Right:**  BSTR bs = ...; //  ...  LPCTSTR sz = static\_cast<LPCTSTR>bs;  ...  ::SysFreeString(bs);  //Never use sz after this line  **Wrong:**  BSTR bs = ...; //  ...  LPCTSTR sz = bs;  ...  ::SysFreeString(bs);  //Never use sz after this line  \_tcslen(sz); //ERROR |
| BSTR | LPWSTR /  WCHAR\* /  wchar\_t\* | **Right:**  BSTR bs = ...; //  //...  UINT len = ::SysStringLen(bs);  // Do not modify the BSTR content by  // C/C++ string functions  LPTSTR sz = new TCHAR[len+1];  \_tcsncpy(sz, bs, len);  ::SysFreeString(bs);  delete []sz;  **Wrong:**  BSTR bs = ...; //  //...  // Do not modify the BSTR content by  // C/C++ string functions  LPTSTR sz = bs; //Error |
| CString | BSTR | **Right:**  CString str1 = ...;  BSTR bs = str1.AllocSysString();  SomeMethod(bs);  // void SomeMethod([in]BSTR)  ::SysFreeString(bs);  CComBSTR bs1(static\_cast<LPCTSTR>(str1));  SomeMethod(static\_cast<BSTR> (bs1) );  // void SomeMethod([in] BSTR )  \_bstr\_t bs2( static\_cast<LPCTSTR>(str1));  SomeMethod(static\_cast<BSTR> (bs2) );  **Wrong:**  CString str1 = ...;  SomeMethod(str1.AllocSysString());  // No one will releasee the return BSTR of  // str1.AllocSysString() |
| BSTR | CString | **Right:**  BSTR bs = SysAllocString(\_T(“Test”));  CString str1(bs);  CString str2;  Str2 = bs;  SysFreeString(bs); // Never forget this line |
| char\* / LPSTR / LPCSTR | BSTR | **Right:**  *Solution 1 ：*  char str[MAX\_STR\_LEN] = "ANSI string";  WCHAR wstr[MAX\_WSTR\_LEN];  // Convert ANSI to Unicode  MultiByteToWideChar( CP\_ACP, 0, str,          strlen(str)+1, wstr,       sizeof(wstr)/sizeof(wstr[0]) );  BSTR bs1 = ::SysAllocString(wstr);  CString cs = str;  BSTR bs2 = cs. AllocSysString()  *Solution 2 ：*  char str[MAX\_STR\_LEN] = "ANSI string";  \_bstr\_t bs1(str);  CComBSTR bs2(str);  **Wrong:**  char \*str = "ANSI string";  BSTR bstr1 = SysAllocString(              (const OLECHAR\*) str); |
| BSTR | char\* / LPSTR / LPCSTR | **Right:**  *Solution 1 ：*  char str[MAX\_STR\_LEN];  BSTR bs = ::SysAllocString(L"Test");  // Convert ANSI to Unicode  WideCharToMultiByte( CP\_ACP, 0,     (LPCWSTR)bs, -1,     str, MAX\_STR\_LEN, NULL, NULL );  ::SysFreeString(bs);  *Solution 2 ：*  BSTR bs = ::SysAllocString(L"Test");  \_bstr\_t bs1(bs, false);  const char\* str = static\_cast <const char\*> bs1;  **Wrong:**  BSTR bstr1 = SysAllocString(L”ANSI string");  char \*str = (char\*) bstr1; |

IMPORTANT: 上面所有的例子都是按照 UNICODE 应用程序设计的。并且不考虑 BSTR 中包含多个字串的情况，也就是 BSTR 只在结束的位置有一个 0 结束符。对于 MBCS/ANSI 程序，可以参考上面的例子。主要区别是对于现在的 COM 版本 OLECHAR 是 wchar\_t ，但是 TCHAR  对于 UNICODE 程序才是 wchar\_t 。