C#调用 VC ++ 动态链接库的研究

李 伟
(西安邮电大学 自动化学院,陕西 西安 710121)

摘要:C#与 C ++ 混合编程是最常见的混合编程方式。用 C#调用 C ++ 编写的动态链接库存在很多需要解决的问题。对 C#调用 C ++ 编写的动态链接库的必要性及托管与非托管的概念进行了描述,并对 C#调用 C ++ 时面临的导入 DLL、数据类型转换、结构体参数的处理、指向结构体的指针及在 C#中用结构体模拟共用体等常见问题进行研究并给出必要的示例代码。

关键词:C#;C ++ ;动态链接库;指针;结构体
中图分类号:TP312 **文献标识码**:A **文章编号**:1000 - 8829(2013)05 - 0105 - 04

Study on C# Invoking the Dynamic Link Library Developed by VC ++

LI Wei
(School of Automation, Xi' an University of Posts & Telecommunications, Xi' an 710121, China)

Abstract: Combined C# with C ++ is the most normally used mixed programming method. There are many problems to be solved while use C# to invoke the dynamic link library(DLL) developed by C ++. The necessity of invoking C ++ developed dynamic link library by C# is described and the concept of managed and unmanaged code is explained. The problems such as import DLL, data type convert, struct parameter, pointer and union parameter etc. are researched and necessary example code is listed.

Key words: C#; C ++ ; DLL; pointer; struct

作为微软为 Visual Studio. net 框架量身定做的程序语言,C#拥有 C/C ++ 的强大功能及 VB 简单易用的特点,从而使 C#成为目前最为流行的开发语言之一。但是 C#也有一些不足,如对不少底层操作无能为力,只能通过调用 Win32 动态链接库(DLL,dynamic link library)或 C ++ 等编写的动态链接库进行底层操作。一般认为 C#的保密性不够强,它容易被反编译而得到部分源码,所以需要使用混合编程加强 C#的保密性,把 DLL 嵌入 C#程序并实现动态调用是比较理想的方法,因为可以把 DLL 文件先用某一算法进行加密甚至压缩后再作为资源文件添加到 C#程序中,在程序运行时才用某一算法进行解压解密后再进行加载,所以即使用反编译软件,也只能得到一个资源文件,且该资源文件是用一个复杂算法加密过的,不可能再次对资源中的内容进行反编译从而大大加强代码的保密性。由于历史及编程习惯的原因,以及 C ++ 在编写设备驱动程序方面固有的优势导致有很多库函数,特别是与硬件有关的驱动程序还是用 VC ++ 编写,并以动态链接库(.dll)的形式提供给 C#使用。

采用动态链接库具有很多优点,如可以使用较少资源,可以推广模块式体系结构,可以简化部署安装等。因此用 C#调用 C ++ 编写的动态链接库具有很广泛的应用场合。但是用 C#调用 C ++ 所写的动态链接库的方法与调用 C#本身所写的动态链接库的方法有很大区别,存在着诸如动态链接库的导入、C#与 C ++ 数据类型转换、结构体转换、指针的转换、共用体参数的构造以及调用习惯等问题。本文详细阐述了以上问题的解决方法,可作为此类应用的一个完整参考。

1 导入 DLL

1.1 托管 DLL 与非托管 DLL

用 C#调用托管 DLL 与调用非托管 DLL 的方法有很大不同,因此一定要明白托管与非托管的区别。C#是内存托管的,new 出来的对象 C#系统还会自动创建一个系统指针指向它,如果创建的指针都被销毁了,.net系统就会自动利用系统指针将空间销毁掉,这个就是垃圾回收的原理,new 出来的对象用完后可以不

收稿日期:2012 - 02 - 29
基金项目:西安邮电学院博士科研启动经费资助项目(110-1207)
作者简介:李伟(1977—),男,四川崇州人,讲师,博士,主要研究方向为微嵌入式系统、无线传感器网络、软件开发。

用再管它，系统会解决一切，当然也可以通过调用对象的 Dispose 方法来手动销毁。基于 Win32 平台开发的 DLL，ActiveX 组件一般是非托管的。比如用 VC ++ 开发出来的 DLL 就是非托管的，new 出来的对象需要自己手动销毁，否则会导致内存泄露。

1.2 C#调用托管 DLL

C#调用托管的 DLL 很简单，只要在“解决方案资源管理器”中需要调用 DLL 的项目下用鼠标右击“引用”，选择“添加引用”，选择已列出的 DLL 或通过浏览来选择 DLL 文件，然后使用 using 导入相关的命名空间，即可像使用自己开发的类一样使用托管 DLL 中的类和函数。

1.3 C#调用非托管 DLL

C#调用非托管 DLL，首先应将需要导入的 DLL 文件放在程序当前目录或系统定义的查询路径中，即系统环境变量中 Path 所设置的路径下。在程序中引入 System. Runtime. InteropServices 命名空间，并应在 C# 语言源程序中声明外部方法，其基本形式为：

```
[DllImport("DLL 文件名.dll")]
修饰符 extern　返回变量类型　方法名称(参数列表)
```

其中，“DLL 文件名”为包含定义外部方法的库文件名；修饰符为除了 abstract 以外的在声明方法时可以使用的修饰符，常见的如 public、private 等；返回变量类型、方法名称、参数列表等都要与 DLL 文件中定义的返回变量类型、方法名及参数列表相一致。DllImport 只能放在方法声明上。若一个 DLL 中包含多个方法，每个方法都需要在 C#源文件中用 DllImport 声明，此时可用 DllImport 的 EntryPoint 属性来设置需调用的方法名。

以下代码为 System. Runtime. InteropServices 名字空间中关于 DllImport 属性的声明：

```
namespace System. Runtime. InteropServices
{
  [AttributeUsage(AttributeTargets. Method)]
  public class DllImportAttribute:System. Attribute
  {
    public DllImportAttribute(string dllName) {...}
    public CallingConvention CallingConvention;
    public CharSet CharSet;
    public string EntryPoint;
    public bool ExactSpelling;
    public bool PreserveSig;
    public bool SetLastError;
    public string Value { get {...} }
  }
}
```

CallingConvention：指示入口点的调用约定，如：CallingConvention = CallingConvention. Winapi；或 CallingConvention = CallingConvention. Cdecl；等。调用时的 CallingConvention 要与开发 DLL 时的 CallingConvention 相对应。

CharSet：指示用在入口点中的字符集，一般选 CharSet = CharSet. Auto，如果有中文就要用 Unicode。

EntryPoint：要调用的方法名。该方法必须存在于所调用的 DLL 中。

ExactSpelling：指示 EntryPoint 是否必须与指示的入口点的拼写完全匹配，如：ExactSpelling = false。

PreserveSig：指示方法的签名应当被保留还是被转换，如：PreserveSig = true。

SetLastError：指示方法是否保留 Win32 的上一个错误，如：SetLastError = true。

2 C#与 C ++ 数据类型转换

用 DllImport 时，要保证声明的参数列表与 DLL 中的参数列表完全一致。为此需要将 C ++ 用的数据类型与 C#中的数据类型进行对应。如表 1 所示。

表 1　C#与 C ++ 数据类型对应关系表

C ++ 类型	C#类型	备注
WORD	ushort	16 位无符号数
DWORD	uint	32 位无符号数
CHAR	char	字符
ulong	uint	32 位无符号整数
int	int	32 位整数
DWORDLONG	long	64 位长整数
UCHAR	int/byte	
Int *	ref int	整型指针
HANDLE	int	句柄，32 位整数
UCHAR *	String/IntPtr	
char *	String	
Void *	IntPtr	空指针
HWND	IntPtr	窗口句柄，指针类型
LPSTR	String	指向字符的 32 位指针
LPCTSTR	String	指向常字符的 32 位指针
long	int	32 位整数
COLORREF	uint	
HDC	int	设备描述表句柄
HINSTANCE	int	实例句柄
HWM	int	窗口句柄
HPARAM	int	32 位消息参数
LPARAM	int	32 位消息参数
WPARAM	int	32 位消息参数

3 C#中的结构体及指向结构体的指针

C ++ 提供的 DLL 中函数的参数很多是用结构体或指向结构体的指针传递的。用 C#调用这样的参数

时,需要重新定义与 C ++ 结构体对应的 C#中的结构体,一定要注意结构体中成员变量的长度要与 C ++ 中结构体的成员变量所占字节数一致。否则,DLL 调用会失败。

3.1 C#中利用结构体数组来充当结构体指针进行参数传递

设 C ++ 中结构体声明如下:

```
typedef struct{
    unsigned char Port;
    unsigned long Id;
    unsigned char Ctrl;
    unsigned char pData[8];
}HSCAN_MSG;
```

C ++ 中的函数声明为

```
extern "C" int _stdcall HSCAN_SendCANMessage(unsigned char nDevice, unsigned char nPort, HSCAN_MSG * msg, int nLength);
```

C ++ 中的调用为:

```
HSCAN_MSG msg[100];
HSCAN_SendCANMessage(m_nDevice, m_nPort, msg, nFrames);
```

可见,函数中指向结构体的指针是通过一个结构体数组传递过去的。可以采用类似的方式在 C#中调用该函数。

首先,要在 C#中声明与 C ++ 中结构体对应的结构体。C#中结构体声明时,一定要注意每个成员所占字节数要与 C ++ 中每个成员所占字节数一致。当成员为数组等类型时,要标明数组长度。

```
[StructLayout(LayoutKind.Sequential)]
  public struct  HSCAN_MSG
{
    //指明 Port 的类型和长度为 1 个字节
    [mARSHALaS(UnmanagedType.U1)]
    public byte Port;
    //指明 Id 的类型和长度为 4 个字节
    [mARSHALaS(UnmanagedType.U4)]
    public uint Id;
    [mARSHALaS(UnmanagedType.U1)]
    public byte nCtrl;
    //指明 pData 为 8 字节数组
    [mARSHALaS(UnmanagedType.ByValArray, SizeConst=8)]
    public byte[] pData;
}
```

然后,在 C#中用 DllImport 引入 DLL 中定义的函数:

```
[DllImport("DllTest.dll")]
public static extern int HSCAN_SendCANMessage(byte nDevice,byte nPort,HSCAN_MSG[] pMsg,int nLength);
```

最后,可在代码中调用函数:

```
//定义数组,数组的大小可根据实际情况设置
HSCAN_MSG[]  msg = new HSCAN_MSG[10];
//用 for 循环将数组中的成员初始化为一个结构体
for(int yy = 0;yy < msg.Length;yy ++)
  msg[yy] = new HSCAN_MSG();
//用结构体数组充当指针调用函数
HSCAN_SendCANMessage(0x0,0x0,msg,10);
```

3.2 C#中使用 IntPtr 充当结构体指针进行参数传递

有时候,C#调用 DLL 函数时不能使用结构体数组充当结构体指针来传递参数,此时,可以使用 C#中的指针 IntPtr 替代结构体指针。下例为采用 InPtr 替代结构体数组的方法,C ++ 结构体及函数原型与 3.1 节中的例子相同。本例中,采用 IntPtr 作为指向结构体的指针进行参数调用,由于在调用完后,有可能该指针所指的值已经变化,所以调用完后对 IntPtr 所指值进行读取。

```
//定义有 10 个元素的结构体数组
HSCAN_MSG[]  msg1 = new HSCAN_MSG[10];
//用 for 循环对结构体数组初始化
for(int i = 0;i < msg1.Length;i ++)
{
    msg1[i] = new  HSCAN_MSG();
    msg1[i].pData = new byte[8];
}
//定义指针数组
IntPtr[] ptArray = new IntPtr[1];
//分配内存
ptrArray[0] = Marshal.AllocHGlobal(Marshal.SizeOf(HSCAN_MSG) * 10);
//定义指针
IntPtr pt = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(HSCAN_MSG)));
//使指针指向 ptArray
Marshal.Copy(ptArray,0,pt,1);
//调用函数,可能 pt 所指的值通过调用已改变。
int count = HSCAN_ReadCANMessage(0x0,0,pt,10);
String  dataRead + = "/r/n" + "读取数据";
//用 for 循环读取函数调用后 pt 所指的值
for(int j = 0;j < 10;j ++)
{
  msg1[j] = (HSCAN_MSG)Marshal.PtrToStructure((IntPtr)((UInt32)pt + j * Marshal.SizeOf(typeof(HSCAN_MSG))),typeof(HSCAN_MSG));
  dataRead + = Convert.ToByte(msg1[j].pData[0]).ToString() + "|" + Convert.ToByte(msg1[j].pData[1]).ToString() + …
}
```

4 C ++ 中的引用类型在 C#中的实现

C ++ 中常用引用来从一个函数调用中输出参数。

如下例中，void swap(int　&a, int &b)。在该例中，参数 a、b 是引用类型。函数调用后，参数 a、b 可能会发生改变。从而能够把函数的返回值带到主调程序中。在 C#中，如果想通过参数得到函数运行的输出结果，在函数调用时必须加 ref 关键字。即上述函数在 C#中调用时，可以这样写：

```
[DllImport("DllTest.dll")]
public　static　extern　void swap(ref int a,ref int b);
int a=0,b=0;
swap(ref a,ref b);
```

该段代码的运行结果是，调用函数后，变量 a、b 的值可能已被函数改变，不再保持原值。用该方式可以得到函数的输出结果。

5　共用体(union)在 C#中的实现

共用体是 C ++ 中常用的一种数据类型，共用体中的数据成员在内存中是互相重叠存储的。每个数据成员都从相同的内存地址开始，分配给共用体的存储区数量是其中最大的数据成员所需的内存字节数，同一时刻只有一个成员可以被赋值，最后赋的值是真正有效的值。C ++ 中常用共用体进行不同的数据类型之间的转换，比如给 char 类型数据赋值，再以 long 类型读出来等。共用体是 C ++ 中常用的数据结构，在 C ++ 编写的 DLL 中，有些函数的参数是共用体类型。为了在 C#中调用这样的函数，必须在 C#中构建这样的数据类型。C#中不提供 union 关键字。只能用结构体来模拟共用体。

设 C ++ 中的共用体定义如下：

```
union　TokenValue{
    char　cVal;
    int　　iVal;
    double dval;
}
```

在 C#中为采用结构体模拟共用体类似的内存布局，需要结合使用 StructLayoutAttribute 特性、LayoutKind 枚举和 FieldOffsetAttribute 特性。它们都位于 System. Runtime. InteropServices 命名空间中。以下是 C#中模拟上述 TokenValue 的代码：

```
[StructLayout(LayoutKind.Explicit,Size=8)]
struct TokenValue
{
  [FieldOffset(0)]
  public char　cval;
  [FieldOffset(0)]
  public int　ival;
  [FieldOffset(0)]
  public double _dval;
}
```

共用体中的的每个数据成员都从相同的内存地址开始，通过把[FieldOffset(0)]应用到 TokenValue 的每一个成员，就指定了这些成员都处于同一起始位置。当然，需要事先告诉 . net 这些成员的内存布局，把 LayoutKind. Explicit 枚举传递给 StructLayoutAttribute 特性的构造函数，并应用到 TokenValue，. net 就不会再干涉该 struct 的成员在内存中的布局了。另外，显式地把 TokenValue 的大小设置为 8 字节，当然，这样做是可选的。这样定义好的 TokenValue 就可以用来作为参数调用 C ++ 写的 DLL 了。

在用 C#模拟 C ++ 的共用体时，需要注意的一点是不要在一个共用体中同时使用值类型和引用类型。如上例中，char、int、double 都是值类型，同时使用不会出问题。但如果在其中同时使用引用类型，比如增加一个 string 类型，当给共用体赋值为 0 时，string 对应的值就为 null，在使用时会导致不必要的错误出现。因此，在 C#中模拟共用体时，数据类型最好都为值类型，才能确保不会出现运行时错误。

6　结束语

本文对 C#调用 C ++ 编写的动态链接库的方法进行了描述。对于托管代码与非托管代码的概念，以及对 C#调用 C ++ 时面临的导入 DLL、数据类型转换、结构体参数的处理、指向结构体的指针及在 C#中用结构体模拟共用体等常见问题进行了研究，并给出了必要的示例代码。本文可作为 C#与 C ++ 混合编程的一个有用参考。

参考文献：

[1] Watson K, Nagel C. C#入门经典[M]. 齐立波，黄静，译. 3 版. 北京：清华大学出版社，2006.

[2] Nagel C, Evjen B, Glynn J. C#高级编程[M]. 李敏波，译. 4 版. 北京：清华大学出版社，2006.

[3] Hejlsberg A, Wiltamuth S, Golde P. C#编程语言详解[M]. 张晓坤，谭立平，车树良，译. 北京：电子工业出版社，2004.

[4] C#与 C ++ 数据类型比较及结构体转换[EB/OL]. http://blog. csdn. net/ruanruoshi/article/details/5675137, 2010.

[5] C#调用 C ++ 写的 dll 文件[EB/OL]. http://www. cnblogs. com/virusswb/archive/2008/06/02/1212358. html, 2008.

[6] Freeman E, Freeman E, Sierra K, et al. 深入浅出设计模式(影印版)[M]. 南京：东南大学出版社，2005.