C 和 C# 混编

作者：sknown

原文链接：https://ld246.com/article/1526710054705

来源网站：链滴

字符串传递

MSDN上给出C/C++字符串类型与C#字符串类型的对应关系

Wtypes.h 中的非托管类型 **非托管C/C++语言类型**
管类名 **说明**

CHAR char System.Char
ANSI 修饰

LPSTR char* System.String 或 System.Text.Stri
gBuilder 用 ANSI 修饰

LPCSTR Const char* System.String 或 System.T
xt.StringBuilder 用 ANSI 修饰

LPWSTR wchar_t* System.String 或 System.Text
StringBuilder 用 Unicode 修饰

LPCWSTR Const wchar_t* System.String 或 Sys
em.Text.StringBuilder 用 Unicode 修饰

MSDN给出Marsha
As属性控制字符串封送行为:

枚举类型 **非托管格式说明**

UnmanagedType.AnsiBStr 长度前缀为双字节的
Unicode字符的COM样式的BSTR

UnmanagedType.LPStr 单字节、null空终止的
NSI 字符数组的指针。（默认值）

UnmanagedType.LPTStr null空终止与平台相关
字符数组的指针。

UnmanagedType.LPWStr null空终止与Unicod
的字符数组的指针。

UnmanagedType.TBStr 一个有长度前缀的与平
相关的 COM样式的BSTR。

void __stdcall PrintString(char * hello)

```
 public static extern void PrintStringByBytes(byte[] hello);
 public static extern void PrintStringByMarshal([MarshalAs(UnmanagedType.LPStr)]string hello
;
```

char * __stdcall GetStringReturn()

```
[DllImport("TestDll", EntryPoint = "GetStringReturn")]
public static extern IntPtr GetStringReturnByBytes();

[DllImport("TestDll", EntryPoint = "GetStringReturn")]
[return:MarshalAs(UnmanagedType.LPStr)]
```

```
public static extern string GetStringReturnByMarshal();

Console.WriteLine(Marshal.PtrToStringAnsi(GetStringReturnByBytes()));
```

封送字符串数组

```
C++:
int TestArrayOfStrings(char* ppStrArray[], int size);

C#:
[ DllImport( "test.dll" )]
public static extern int TestArrayOfStrings( [In, Out]   String[] ppStrArray, int size );

使用:
String[] strArray = { "one", "two", "three", "four", "five" };
int lenSum = LibWrap.TestArrayOfStrings( strArray, strArray.Length );
```

结构体传送

按顺序字节方式即可：[StructLayout(LayoutKind.Sequential)]

```
C:
  struct Lable1  {
        BYTELabFilterChan0[4][256];
        BYTELabFilterChan1[4][256];
  }

C#:
   [StructLayout(LayoutKind.Sequential)]
    public structByteStru    {
       [MarshalAs(UnmanagedType.ByValArray, SizeConst = 256)]
        public byte[]a;
    };


   [StructLayout(LayoutKind.Sequential)]
    public structLabel1    {
       [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
        publicByteStru[] LabFilterChan0 ;

       [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
        public ByteStru[] LabFilterChan1 ;
    };

C:
typedef struct _MYPERSON{
char* first;              //字符指针
} MYPERSON, *LP_MYPERSON;

C#:
[ StructLayout( LayoutKind.Sequential, CharSet=CharSet.Ansi )]
public struct MyPerson {
```

```
public String first;
}

C:
typedef struct _MYPERSON1{
    char first[20];        //字符数组
} MYPERSON1, *LP_MYPERSON1;

C#:
[ StructLayout( LayoutKind.Sequential, CharSet=CharSet.Ansi )]
public struct MyPerson1 {
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 20)]
    public String first;
}

C:
typedef struct _MYARRAYSTRUCT{
    bool flag;
    int vals[ 3 ];            //值类型数组
} MYARRAYSTRUCT;
C#:
public struct MyArrayStruct {
    public bool flag;
    [ MarshalAs( UnmanagedType.ByValArray, SizeConst=3 )]
    public int[] vals;
}
```

- 结构体声明必须保证：字段声明顺序、字段类型、字段在内存中的大小原来的一致！结构体名称，成员名称可以不同。

- 结构体中，char*与char[]在C#声明区别很大，前者直接对应string，后者(字符数组)很容易被初学误用char[]来对应，它还是要用string来对应，但还需要用[MarshalAs(UnmanagedType.ByValTStr, izeConst = 20)]来指明该字段的封送行为。

- 其他值类型的数组，直接用数组方式对应，但也需要用[MarshalAs(UnmanagedType.ByValArray, SizeConst=3)] 指明封送行为。

- 有直接结构体对应的结构体指针，建议直接用ref + 具体类型，而不采用IntPtr，省去一些不必要的换操作，

TestArrayInStruct、TestStructInStruct2、TestStructInStruct3都是如此。

结构体作为函数返回值

```
C++:
MYPERSON* TestReturnStruct();
void FreeStruct(MYPERSON* pStruct);

C#:
[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern IntPtr TestReturnStruct();        //注意对应的是IntPtr指针

[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern void FreeStruct(IntPtr pStruct);
```

使用:

```
IntPtr pStruct=TestReturnStruct();
MYPERSON person=(MYPERSON)Marshal.PtrToStructure(pStruct,typeof(MYPERSON));
```

```
//在非托管代码，大多用new/malloc分配内存，net无法正确释放，
//需要对应的调用释放内存的方法释放非托管内存
```

```
FreeStruct(pStruct);
```

结构体数组作为输入输出参数

C++:

```
int TestArrayOfStructs2 (MYPERSON* pPersonArray, int size);
```

C#:

```
[ DllImport( "test.dll" )]
public static extern int TestArrayOfStructs2( [In, Out] MyPerson[] personArray, int size );
```

使用:

```
MyPerson[] persons = { new MyPerson( "Kim", "Akers" ), new MyPerson( "Adam", "Barr" )};
int namesSum = TestArrayOfStructs2( persons, persons.Length );
```

总结:

- 一般我们数组作为输入输出参数，需要显式加上[In,Out]属性，标识为输入输入参数。如果不写，认为In方向，CLR将不会回传修改后的内存值

结构体嵌套结构体

C++:

```
typedef struct _MYPERSON2{
    MYPERSON* person;
    int age;
} MYPERSON2, *LP_MYPERSON2;

typedef struct _MYPERSON3{
    MYPERSON person;
    int age;
} MYPERSON3;

int TestStructInStruct(MYPERSON2* pPerson2);
void TestStructInStruct3(MYPERSON3 person3);
```

C#:

```
[ StructLayout( LayoutKind.Sequential )]
public struct MyPerson2 {
    public IntPtr person;
    public int age;
}

[ StructLayout( LayoutKind.Sequential )]
```

```
public struct MyPerson3 {
    public MyPerson person;
    public int age;
}

[ DllImport( "test.dll" )]
public static extern int TestStructInStruct( ref MyPerson2 person2 );

[ DllImport( "test.dll" )]
public static extern int TestStructInStruct3( MyPerson3 person3 );
```

使用：

```
MyPerson personName;
personName.first = "Mark";
personName.last = "Lee";

MyPerson2 personAll;
personAll.age = 30;
IntPtr buffer = Marshal.AllocCoTaskMem( Marshal.SizeOf( personName ));

Marshal.StructureToPtr( personName, buffer, false );
personAll.person = buffer;

int res = TestStructInStruct( ref personAll );
MyPerson personRes = (MyPerson)Marshal.PtrToStructure( personAll.person, typeof( MyPers
n ));

Marshal.FreeCoTaskMem( buffer );

MyPerson3 person3 = new MyPerson3();
person3.person.first = "John";
person3.person.last = "Evens";
person3.age = 27;

TestStructInStruct3( person3 );
```

总结：

- 结构体嵌套，如果是实体成员，直接用结构体类型对应，如上面的MyPerson3；
- 如果是指针变量，则用IntPtr对应，如上面的MYPERSON2；
- 如果嵌套的是结构体数组，那么，出来办法以值类型数组方式对应，如MYARRAYSTRUCT，只不，类型为具体的结构体类型。这里不另外在举例。(还是给个例子)

C++:

```
typedef struct Student  {
    char name[20];
    int age;
    double scores[32];
}Student;
typedef struct Class  {
    int number;
    Student students[126];
}Class;

C#:
    [StructLayout(LayoutKind.Sequential)]
    public struct Student
    {
           [MarshalAs(UnmanagedType.ByValTStr,SizeConst=20)]
           public string name;
           public int age;
           [MarshalAs(UnmanagedType.ByValArray,SizeConst=32)]
           public double[] scores;
    }

    [StructLayout(LayoutKind.Sequential)]
    struct Class
    {
          public int number;
          [MarshalAs(UnmanagedType.ByValArray,SizeConst=126)]
          public Student[] students;
    }
```

函数调用

```
C:
int TestStructInStruct1(MYPERSON pPerson);
int TestStructInStruct2(MYPERSON* pPerson);
int TestStructInStruct3(MYPERSON1* pPerson);
void TestArrayInStruct( MYARRAYSTRUCT* pStruct );

C#:
[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern int TestStructInStruct( MyPerson person);

[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern int TestStructInStruct1(ref  MyPerson person);

[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern int TestStructInStruct2(ref  MyPerson1 person);

[ DllImport( "test.dll" ,CharSet = CharSet.Ansi)]
public static extern int TestArrayInStruct(ref MYARRAYSTRUCT person);
```

函数调用标准

```
    // Summary:
    //     Specifies the calling convention required to call methods implemented in
    //     unmanaged code.
    [Serializable]
    [ComVisible(true)]
    public enum CallingConvention
    {
        // Summary:
        //     This member is not actually a calling convention, but instead uses the default
        //     platform calling convention. For example, on Windows the default is System.Runtime.InteropServices.CallingConvention.StdCall
        //     and on Windows CE.NET it is System.Runtime.InteropServices.CallingConvention.Cdcl.
        Winapi = 1,
        //
        // Summary:
        //     The caller cleans the stack. This enables calling functions with varargs,
        //     which makes it appropriate to use for methods that accept a variable number
        //     of parameters, such as Printf.
        Cdecl = 2,
        //
        // Summary:
        //     The callee cleans the stack. This is the default convention for calling unmanaged
        //     functions with platform invoke.
        StdCall = 3,
        //
        // Summary:
        //     The first parameter is the this pointer and is stored in register ECX. Other
        //     parameters are pushed on the stack. This calling convention is used to call
        //     methods on classes exported from an unmanaged DLL.
        ThisCall = 4,
        //
        // Summary:
        //     This calling convention is not supported.
        FastCall = 5,
    }
```

原文链接：C 和 C# 混编