这是封面

By [Largesky@hotmail.com](mailto:Largesky@hotmail.com)

# 目录

[目录 2](#_Toc379714423)

[一、 前言 4](#_Toc379714424)

[编写此书的目地 4](#_Toc379714425)

[本书适合的读者 4](#_Toc379714426)

[本书将涉及的内容 4](#_Toc379714427)

[一些术语的使用 4](#_Toc379714428)

[示例代码的下载 4](#_Toc379714429)

[欢迎反馈您的意见 4](#_Toc379714430)

[二、 基础概念 5](#_Toc379714431)

[32位与64位应用程序 5](#_Toc379714432)

[Unicode与ANSI 5](#_Toc379714433)

[方法调用约定 6](#_Toc379714434)

[三、 Platform Invoke 基础 6](#_Toc379714435)

[执行过程 6](#_Toc379714436)

[声明托管类原型 7](#_Toc379714437)

[声明托管函数原型 8](#_Toc379714438)

[调用声明的C#函数 9](#_Toc379714439)

[参数的输入与输出 10](#_Toc379714440)

[加载非托管 DLL流程 11](#_Toc379714441)

[函数查找规则 11](#_Toc379714442)

[四、 基础类型 12](#_Toc379714443)

[五、 字符串 13](#_Toc379714444)

[确认被调用函数使用的字符串编码 13](#_Toc379714445)

[C#中映射字符串编码的几种方式 13](#_Toc379714446)

[C#中接受C更改后的字符串结果 15](#_Toc379714447)

[单个字符串的使用char 15](#_Toc379714448)

[String指向的内容真的不可变吗 16](#_Toc379714449)

[六、 数组 17](#_Toc379714450)

[基础类型数组 17](#_Toc379714451)

[字符数组 17](#_Toc379714452)

[char[]与byte[]的陷阱 18](#_Toc379714453)

[结构数组 19](#_Toc379714454)

[七、 Class与Struct 20](#_Toc379714455)

[选择Class或Struct 20](#_Toc379714456)

[使用基础数据 Int Double作为成员变量 21](#_Toc379714457)

[使用Char字符作为成员变量 21](#_Toc379714458)

[使用字符数组 23](#_Toc379714459)

[使用字符指针 25](#_Toc379714460)

[使用内嵌结构 26](#_Toc379714461)

[使用内嵌结构数组 27](#_Toc379714462)

[使用指针 28](#_Toc379714463)

[使用指针数组 30](#_Toc379714464)

[八、 联合Union 33](#_Toc379714465)

[基础数据类型联合 33](#_Toc379714466)

[非基础数据类型联合 34](#_Toc379714467)

[结构中包含联合 36](#_Toc379714468)

[联合参数传递问题 38](#_Toc379714469)

[九、 指针 41](#_Toc379714470)

[一级指针 41](#_Toc379714471)

[二级指针 41](#_Toc379714472)

[指针作为函数返回值 44](#_Toc379714473)

[十、 回调函数 44](#_Toc379714474)

[回调函数实现机制 44](#_Toc379714475)

[声明回调函数的委托 44](#_Toc379714476)

[函数回调示例 44](#_Toc379714477)

[十一、 综合示例 44](#_Toc379714478)

[十二、 进阶 数据封送处理器 46](#_Toc379714479)

[结构一致型与非结构一致性 46](#_Toc379714480)

[参数传递的两种方式 复制与固定 48](#_Toc379714481)

[附录 常用类型信息 48](#_Toc379714482)

[DllImportAttribute 48](#_Toc379714483)

[InAttribute 49](#_Toc379714484)

[OutAttribute 49](#_Toc379714485)

[MarshalAsAttribute 49](#_Toc379714486)

[UnmanagedFPointerAttribute 50](#_Toc379714487)

[Marshal 50](#_Toc379714488)

[UnmanagedType枚举 51](#_Toc379714489)

[附录 托管与非托管的数据类型映射 53](#_Toc379714490)

[附录 常见问题 54](#_Toc379714491)

[为何不能加载我的DLL 54](#_Toc379714492)

[BadImageFormatException 55](#_Toc379714493)

[调试时提示我设置的回调委托正在被回收 55](#_Toc379714494)

[非托管代码调用托管回调函数后，提示栈异常 55](#_Toc379714495)

[非托管代码在对托管传入的参数进行赋值时出现 内存访问异常 55](#_Toc379714496)

[为什么OutAttribute对String无效 56](#_Toc379714497)

# 前言

## 编写此书的目地

## 本书适合的读者

虽然距我最后一次使用Platform Invoke技术已经过去了两年时间，但我仍然可以感受到当时的心情有多么的糟糕与沮丧。是的，当时我正忙着调用Win32 API设置系统服务启动类型。而那时我是一个正在大学的大四学生，会一点C#,也会一点C和C++（其实那时最熟悉的还是Java）。

虽然很少使用这种技术了，但是个人却在那之后，进行了不断的学习。后来在网络上也经常看到有人提问这方面的问题，他们通常是编写设备控制程序的程序员，如自动化测试，视频监控等。但是回答者通常很少，一是因为大多数的人并不了解这种技术。二是因为一个程序有错其原因真的是太多了，我敢肯定有时候真的是比路上的美女多个几倍。

所以我编写此书，给那些需要系统学习PI技术的人。当然你可以当作快速参考手册因为在高级示例里面有很多可以参考的实例。

本书适合那些需要使用Platform Invoke技术的人。而阅读者需要熟悉C#或者其实类似的xxx.net语言。如果不熟悉，建议先学习一门语言。当然熟悉C语言更佳。

## 本书将涉及的内容

## 一些术语的使用

## 示例代码的下载

## 欢迎反馈您的意见

人这一辈子，总是会做一些错事呀。由于知识水平与其它原因，本书难免出现错误。欢迎指正。当然有好意见也欢迎反馈。

你可以通过以下方式联系我：largesky@hotmail.com

# 基础概念

## 32位与64位应用程序

在CPU中有一个非常重要的概念：CPU位数。表示CPU内部一次性可以处理数据的位数。对于支持X86指令的CPU来说它的位数是32位。而对于支持AMD64指令的CPU来说就是64位。现在的CPU都同时支持X86与AMD64指令，并且可以工作这两种模式之上。

对于非托管应用程序（包括DLL，Exe等）而言，在编译好的指令中它只能是支持64位或者32位的，不能两者都支持。这是因为在AMD64指令环境下，一些数据的类型大小，指令与X86都不相同。

当然对于一般的应用来说，最直接的反映就是指针是64位，这使得一个应用程序可以使用内存理论上达到了2的64次方。

在Windows中，当创建一个进程时，系统就同时确定它是32位的，还是64位。32位的进程，只能加载32位的DLL。而64位的进程只能加载64位的DLL。

而对于托管.NET程序而言，它还多一种模式：Any CPU。这种方式编译的程序集，当系统是32位的它就以32进程序运行。当系统是64位的就以64进程运行。

表2-1：程序运行平台及其可以加载的DLL

|  |  |  |
| --- | --- | --- |
|  | 32位操作系统 | 64位操作系统 |
| Any cpu | 可以运行，只能加载32位DLL | 可以运行，只能加载64位DLL |
| 32位 | 可以运行，只能加载32位DLL | 可以运行，只能加载32位DLL |
| 64位 | 不能运行 | 可以运行，只能加载64位DLL |

对于.NET应用程序言,如果由于编译位数与运行的目标系统不匹配，可能导致加载相应的DLL失败，也就是抛出BadImageFormatException.

**建议：如果一个C#程序使用32的位的C代码DLL，则该程序就应该被编译成32的，如果使用了64的C代码DLL则该程序就应该被编译成64位的。如果C#程序没有使用其它本地的DLL则依据所依赖的其它的。NET库DLL决定，当然这时候你通常可以编译成anycpu。**

## Unicode与ANSI

Unicode用4个字节来编码所有的字符，也就是每一个字符在Unicode中对应一个唯一的值。

然而在实际使用中，人们发现常用的字符并不多，所以人们开发出其它表示方式来编码Unicode，从而节约储存空间。我们常用的有UTF-8，Unicode 16，UTF-32。例如一个汉字’中’在UTF -8中为三个字节，而在Unicode 16中为两个字节。

Windows操作系统自从Windows 2000开始其内核就采用的是Unicode16编码。.NET程序也是采用的Unicode16编码。Unicode16编码，并不能表示所有字符，但是能够表示大多数常用的字符，而且在表示非字母时比UTF-8节省空间。我想这也是Windows采用该标准的原因之一吧。

ANSI编码，准确的来说这不是一种编码。因为它的实际编码方式依据系统语言环境而定。例如在简体中文操作系统上ANSI的实际编码方式为:GB2312。而在英文操作系统上ANSI的编码方式为:ASCII。

**注：在.NET程序中，我们常说Unicode编码是指Unicode16 Big Endean编码。**

## 方法调用约定

函数调用约定描述了调用者如何传参数给被调用函数，以及调用完成后谁负责处理栈指针(ESP)。常用的调用约定如下：

Cdecl：调用参数从右至左依次入栈。由调用者完成栈清理。如C库中的printf

Stdcall：调用参数从右至左依次入栈。由函数自身完成栈清理。如Win32中的CreateFile

Fastcall：调用参数中最左边的两个不大于4(32位)或者8(64)的参数采用ECX和EDX传递，其余参数依次入栈。由函数本身完成栈清理。

**注意：在.NET中函数调用约定为Fastcall。在.NET中调用C语言DLL函数时，一定要注意函数调用约定。**

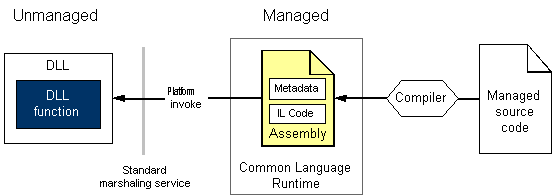
# Platform Invoke 基础

## 执行过程

Platform Invoke也通常简称为PInvoke或者PI。我们中文叫做：跨平台调用。它允许托管代码调用非托管代码。

在编译器编译PInvoke代码的时候，会生成特定的代码并与CLR完成相应的操作，这些细节通常不需要我们关心。

PI调用流程如下图：



从上图可以看出：托管代码通过PI调用非托管代码。CLR中有一个数据封送处理器，它会完成参数的输入与输出（如果指定了）。有关数据封送器请参见第十三章 进阶 数据封送处理器

## 声明托管类原型

当我们调用非托管函数时函数参数大多数时候只是基本的数据类型如INT,DWORD,BOOL等。但有时候它们可能更复杂，有可能是结构，数组，联合，指针等。

无论是非托管代码中的结构还是联合，在使用之前我们都要在托管代码中进行声明。

示例：

C：

typedef struct \_Dog

{

int age;

wchar\_t name[24];

}Dog;

C#：

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode, Pack = 8, Size = 52)]

public class Dog

{

[MarshalAs(UnmanagedType.I4)]

public int age = 123;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 24)]

public string name = "123".PadRight(24, ' ');

}

在上面的示例中我们在C语言中声明了一个Dog类型，它包含一个int类型的age字段与一个字符串数组可以包含24个wchar\_t字符(双字节字符)。

在对应的C#中我们也声明了一个Dog类（两者的名称可以不一样）。该类也包含一个age字段与一个name字段。下面让我们详细说明声明的各个部分：

StructLayout：用来指明一个类内部字段的分布类型。默认情况下C#编译器会采取最优的方式来分布类中字段。在此处我们使用LayoutKind.Sequential来表示：类中的字段应该按照我们编写的顺序排列，只有这样才能和C代码中的类结构正确对应。

CharSet：用来指明类的字段类型为string、char,char[]数组时，它应该映射到ANSI char还是Unicode wchar\_t。

Pack：结构的对齐位数，默认为8.

Size：该类的字节大小。此处为52。

在public int age=123;的上一行，我们使用了[MarshalAs(UnmanagedType.I4)]标记，表示：该字段与C中的4字节int对应。

在public string name = "123".PadRight(24, ' ');的上一行，我们使用了[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 24)]标记，表示该字段映射到C语言中的字符串数组，数组的长度为24，由于CharSet为Unicode，所以每一个元素类型为wchar\_t。

MarshalAs：用来标记一个类字段的映射类型。

实际上为了简化编程，C#中有许多的默认值，对于以上的声明我们可以简化成以下声明：

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]

public class Dog

{

public int age = 123;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 24)]

public string name = "123".PadRight(24, ' ');

}

如上所示，是简化后的类型。

StructLayout：中的Pack,Size均不赋值，在程序运行的时候将由CLR自动确定。

Public int age=123;不用MarshalAs标记，系统自动映射成C中的int。

*有关系统类型的默认映射请参见：附录 托管与非托管的数据类型映射。*

## 声明托管函数原型

在声明使用的类原型后（如果都是基础类型则不需要），接下来就需要声明C#中的函数原型。

示例：

C：

BOOL Ch3\_ModifyDog(Dog \*dog);

C#:

public class Chapter3\_Native

{

const string DLLName = "Native.dll";

[DllImport(DLLName, EntryPoint = "Ch3\_ModifyDog", CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Unicode, SetLastError = true)]

[return: MarshalAs(UnmanagedType.Bool)]

[HandleProcessCorruptedStateExceptions]

public static extern bool Ch3\_ModifyDog([MarshalAs(UnmanagedType.LPStruct), In, Out] Dog dog);

}

现在让我们来分析以上的声明：

在C#中一切对象皆为类，每一个方法都不能独立存在，必须属于一个类，所以我们必须首先声明一个类，在这里我把它叫做：Chapter3\_Native。

const string DLLNAME=”Native.dll”：在类中声明了一个字符串常量，用来指示DLL名称路径。在下面你将看到会使用该值。

在一个函数上标明DllImport同时指定static extern表示该函数并不是由.NET代码实现的，而是指向一个非托管代码中的某个函数。

DllImport各个属性值说明：

* DLLNAME：表示我们调用的函数所在的DLL名称或者全路径。注意：我通常会使用这种方式进行声明，特别是当函数很多的时候。因为到改变DLL储存路径时，这种方式会更方便。
* EntryPoint：表示C DLL中的函数名称。
* CallingConvention：表示我们调用的API函数调用约定方式（请参见第二部分 基础概念中的 函数调用约定）。AbsInt采用的是cdecl方式。
* CharSet：表示调用函数的字符串环境。
* SetLastError：在函数调用返回之前，由CLR自动调用GetLastError()，并设置到托管环境中，该值可以通过Marshal.GetLastWin32Error()获得。

[return: MarshalAs(UnmanagedType.Bool)]：用来指示函数的返回值信息，MarshalAs(UnmanagedType.Bool)表示C语言中的函数其返回值是int。此时CLR将自动将其值转换成C#中一个字节的bool值。其规则为 false=0,true=!0。

[HandleProcessCorruptedStateExceptions]：表示应用程序可以捕捉C语言中发生的异常如常见的ACCESS\_VALIDATION，如果不指明该属性，则C#中的tray catch不能捕捉异常，程序将直接退出。

[MarshalAs(UnmanagedType.LPStruct), In, Out] Dog dog：声明函数的参数：

* MarshalAs(UnmanagedType.LPStruct)：表示参数dog应该被转换成一个C语言风格的指针。
* In：参数作为输入。
* Out：参数作为输出。

关于参数的输入与输出请参见本章后面的：参数的输入与输出节。

## 调用声明的C#函数

到目前为此，我们学习了如何声明相应的类结构与函数原型（如果你仍有不清楚请不要急，继续向后看，并可以回过来头看本节）。下面让我们使用它：

C#:

public static void Test\_Ch3\_ModifyDog()

{

Dog dog = new Dog();

if (Chapter3\_Native.Ch3\_ModifyDog(dog) == false)

{

throw new Exception("Test\_Ch3\_ModifyDog fail");

}

}

很简单不是吗？就像调用C#编写的函数一样，生成了一个Dog实例，将调用C中的代码进行修改，通过调试观察上述代码是正确的。

## 参数的输入与输出

到目前为此，我们已经学习了基础的使用。下面让我们讨论一下参数的输入与输出细节。

参数的输入只有两种方式：值传递与指针（引用）传递。在C语言中除了数组外的任何类型都可能以通常上述两种方式传递。而在C#中，只有值类型才能使用值传递，引用类型只能使用指针（引用 ）传递。

示例1：

C:

void Ch3\_ModifyDog1(int iValue, double \* dValue, Dog \* pDogValue);

C#:

[DllImport(DLLName, CallingConvention = CallingConvention.Cdecl)]

public static extern void Ch3\_ModifyDog1(int iValue, ref double dValue, Dog dogValue);

示例2：

C:

void Ch3\_ModifyDog2(int iValue, double \* dValue, Dog dogValue);

C#:

[DllImport(DLLName, CallingConvention = CallingConvention.Cdecl)]

public static extern void Ch3\_ModifyDog2(int iValue, ref double dValue, [MarshalAs(UnmanagedType.Struct), In, Out] Dog dogValue);

在上面的两中示例中：

参数int iValue都采用是的值传递。

参数double dValue采用的是指针传递。

示例1中的Dog \*pDogValue采用的是指针传递，而示例2中的采用的是值传递。注意：示例2中比示例中多一个标记[MarshalAs(UnmanagedType.Struct), In, Out]，这表示参数dog将采用值传递，也就是说dog的数据会被自动复制到C函数的栈上。

总结输入与输出的特性：

* 基础数据类型：如果是值传递不用任何标记，如果是指针传递则采用ref关键字（或者out关键字）标记。同时应该注意这种方式只能实现一级指针，如果需要实现多级指针请参考第九章 指针。
* Struct：默认采用指针传递，但是不会接收C语言中修改（有特殊情况）。标记[Out]可以接收C语言中修改。采用MarshalAs(UnmanagedType.LPStruct)声明可以强制为值传递。
* Class：默认采用指针传递，同时会接收C语言中的修改（不用标记[Out]属性）。采用MarshalAs(UnmanagedType.Struct)声明可以强制为值传递。

编程建议：

在C#中总是声明类而不是结构，因为类可以有无参的构造函数，而结构不行。类具有自动接收C语言修改后的数据的特性，而结构需要用[Out]标记。

***注意：当一个结构包含一个结构数组时，被包含的结构必须采用Struct声明。***

## 加载非托管 DLL流程

在CLR能够调用相应的C函数之前，它首先要做就是加载指定的DLL文件。C#应用程序在启动的时候并不会立即加载Platform Invoke所以引用的DLL，而是在第一次使用的时候进行加载。

CLR在加载该DLL的时候采用标准的非托管DLL加载顺序：

1：如果Dll名称为绝对路径，例如以D:开头的，则直接加载相应的文件，如果找不到则失败。

2：如果Dll以相对路径或者名称，例如NativeDlls\Native.dll则根据应用程序当前目录加上相对路径。如应用程序当前目录为D:\MyWork则，DLL的搜索路径为D:\MyWork\NativeDlls\Native.dll。如果找不到则进行下一步。

3: 在进程的应用程序目录查找，如果找不到则进行下一步。

4：根据Dll名称到SystemDirectory目录查找，如果找不到则进行下一步。

5：根据Dll名称到Windows Directory目录查找，如果找不到则进行下一步。

6：在应用程序当前的环境变量path中所指定的路径进行搜索。

7：加载该DLL所依赖的所有的DLL,搜索顺序为3,4,5,6,7。

*注意：在加载某个DLL时，会自动加载其依赖的Dll，其搜索顺序同上。如果在加载某一个DLL时，其依赖的DLL未成功加载，也会报DLL无法正常加载。*

*注意：加载非托管代码的DLL与加载托管代码的DLL行为是有差别的。*

*注意：由于32与64的原因，有可能也无法正常DLL，有关该详情请参考 基础概念 中的32位与64位应用程序。*

关于Windows加载顺序可以参考：<http://msdn.microsoft.com/en-us/library/windows/desktop/ms682586(v=vs.85).aspx>

## 函数查找规则

我们已经知道了如果加载DLL，当加载DLL后，CLR将从其中查找符合的函数名称，并取得它的地址进行调用。

以Windows中的函数CreateFile为例，事实上CreateFile为一个宏定义，当编译条件为Unicode时，它真正编译调用的是CreateFileW函数。而当编译条件为ANSI时，它真正编译调用的是CreateFileA函数。那么CLR该如何调用到正确的C语言中的函数了？

CLR在查找函数名称时，主要受以下几个方面的影响：

* EntryPoint：指明被调用非托管函数名称或者序号，如果没有指定该值，则默认为函数名称与C#代码声明的函数名称一样。
* ExactSpelling：如果为true表示，不根据charset来查找函数，使用声明的函数名称。
* CharSet：控制函数名称重整。如果为Unicode则会查找带W的函数，否则查找带A的函数。

图3-1是函数名称查找的流程图：

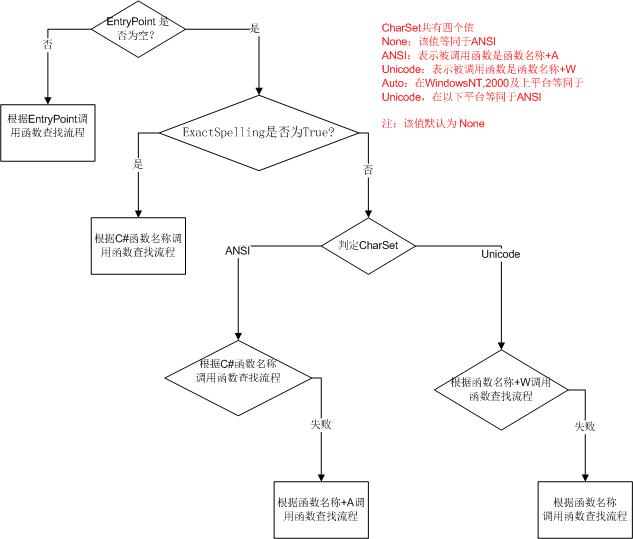


图3-1 函数查找流程

# 基础类型

在前面上节我们学习了，如何声明PInvoke函数原型以及基础的数据类型映射。在接下来的章节中，让我们开始进行基础的应用。

示例：

C:

BOOL Ch4\_ModifyBasic(int iValue,double \*dValue,DWORD \*dwValue);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch4\_ModifyBasic(int iValue, ref double dValue, ref uint dwValue);

对于基础类型来说，我们在传递参数时并不需要做太多的事情。因为CLR已经知道一切，并帮助我们完成了这些工作。但是你仍应该清楚的知道：基础数据类型的指针应该采用ref或者out关键字。

而Windows中的DWORD类型实际为unsigned long，为无符号32位整数，在C#中与之对应的就是uint。

有关于基础数据类型的多级指针，请参见第九章指针。

# 字符串

在学习了基础的数据类型传输入后，我们来学习程序中最常使用的字符串。字符串表示一串连续的字符。但是在托管与非托管中有以下两个不同点：

1：在托管程序中，字符串包含其 字符串长度信息。而非托管中不包含长度信息。

2：在托管程序中，字符串始终是Unicode的。而在非托管中它即可以是Unicode的，也可以是ANSI的。

3：C#中String的编码规则总是可以适用于Char。

## 确认被调用函数使用的字符串编码

现在回想一下，你所知道的字符串编码方式。我记的有Unicode16,ASCII，UTF8,UTF32,GB2312,GBK等等。

但是在Windows中，我们最常用的编码方式是ANSI(在简体中文中系统中被映射成GB2312)与Unicode(Unicode16)。

那么如何确定一个函数的字符串编码方式了？

通常来说有一下几种方式：

* 阅读C(C++)函数的声明。
* 对于CreateFile这类函数需要知道编译时的字符串环境，因为它会真正编译成CreateFileW或者CreateFileA。这可能需要阅读相应的文档或者向开发人员求助。
* 如果没有文档，也没有开发人员，则可以阅读汇编代码从而推断其字符串编码方式。

在确定时应该始终记住以下表格：

|  |  |
| --- | --- |
| C语言类型 | 编码类型 |
| char,CHAR | ANSI |
| wchar\_t WCHAR | Unicode |

## C#中映射字符串编码的几种方式

在正确的确定了函数字符串各个参数的编码方式后，就可以编写C#对应函数原型。

在C#中控制字符串编码类型映射的方式有两种：

* 在class或者struct中标记的CharSet值。
* 在函数声明DllImport中标记的 CharSet值。
* 参数声明标记的MarshalAs值。

关于在class或者struct中标记使用CharSet值，将在第七章学习，这里我们主要学习后两种方式。

示例：

C:

int Ch5PrintA(const char \* message,const char \* message1);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]

public static extern int Ch5PrintA(string message, string message1);

在C语言的函数Ch5PrintA中声明了两个参数，根据上一节的判断方式，我们知道它们都是ANSI编码。

在函数的声明中DllImport中标记CharSet=CharSet.Ansi。标记该值后它将起两个作用：

1：控制函数名称查找（请参见第三章 函数查找规则）。

2：控制函数参数中字符串参数的编码映射规则。

在此是CharSet是Ansi：也就是说在CLR调用C语言的函数时，它将自动在在非托管内存中创建两个缓存，并自动的将message与message1从Unicode转换成Ansi函数，然后调用方法，在方法返回后清理所创建的两个缓存。

示例：

C:

int Ch5PrintW(const wchar\_t \* message,const wchar\_t \* message1);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Unicode)]

public static extern int Ch5PrintW(string message, string message1);

仔细观察一下上面的例子，你会以现所有的参数都是使用的Unicode编码，而在C#中，我仅仅是将CharSet标记设置为Unicode。

一切就是这样简单，难道不是吗？在前面的两个示例中，我们稍做修改函数就可以支持ANSI与Unicode两个版本。感谢CLR团队，他们为我们做了很多基础的工作。

如果你足够细心，你会发现上两个示例中函数的参数都是一致的。一致的意思是：它们要么都是ANSI，要么都是Unicode。那么如果一个函数同时带有多种编码方式的参数了，如下：

示例：

C:

int Ch5PrintAW(const char \* message,const wchar\_t \*message1);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern int Ch5PrintAW([MarshalAs(UnmanagedType.LPStr)] string message, [MarshalAs(UnmanagedType.LPWStr)] string message1);

Ch5PrintAW正如函数中AW所表示的那样，它接受ANSI与Unicode参数。这使事情变得复杂。幸好我们可以使用MarshalAs分别标记每一个参数。

MarshalAs：用来标记参数，返回值，类（结构）中的字段成员的数据映射方式。

在此处我们将string message映射成LPStr，也就是C中的ANSI字符串。

而将string message1映射成LPWStr，也就是C中的Unicode字符串。

我们没有在DllImport标记CharSet，因为这并不影响。

在CLR调用C语言方法时，采用以下以规则确定函数参数中字符串的编码映射。

1：如果参数标记有MarshalAs则使用该标记的值进行映射。

2：如果没有标记MarshalAs，则使用DllImport中的CharSet值进行映射。

*建议：为了编程方便，对那些参数类型一致的函数采用DllImport中的CharSet值进行映射控制，而不是在每一个参数中编写。*

## C#中接受C更改后的字符串结果

前面的示例告诉我们如果控制字符串参数的传入，接下来我们将学习如果接受C语言更改后的结果。

示例：

C:

int Ch5Login(const wchar\_t \*name,const wchar\_t \*pwd,wchar\_t \*error);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Unicode)]

public static extern int Ch5Login(string name, string pwd, StringBuilder error);

C# 调用：

string name = "123";

string pwd = "123";

StringBuilder error = new StringBuilder(128);

int ret = Chapter5Native.Ch5Login(name, pwd, error);

函数Ch5Login共有三个参数，前两个为只读的参数。而最一个error表示要接收的错误消息，也就是说函数会填充该缓存，以指示相应的错误消息。

在C#中string指向的内容一经生成，变不可被更改。所以不能使用string类型来接受C语言所做修改。而只能使用StringBuilder。在函数调用完成后，你可以简单的调用StringBuilder的ToString()方法从而得到一个string类型的实例。

**注意：[Out]标记对string类型无效，会被CLR自动忽略。StringBuilder默认带有[In]，[Out]属性，所以上面的示例我们没有使用[Out]。**

在这个示例中我并没有编写ANSI版本，读者可以自动编写相应的版本以进行验证。

## 单个字符串的使用char

对于单个字符来说，它即有基础数据类型的特性，又有string类型的特性。

示例：

C:

BOOL Ch5ModifyChar(wchar\_t \*chr);

C#：

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Unicode)]

public static extern bool Ch5ModifyChar(ref char chr);

C#调用：

char c = 'B';

bool ret = Chapter5Native.Ch5ModifyChar(ref c);

采用ref关键字，将使用CLR传递c变量在线程栈上的地址（使用ref时不用[Out]）。C函数修改完成后，返回将可以看到c变量的内容已经被修改成了’A’。

CharSet.Unicode表示char chr将被转换成Unicode，而不是Ansi。

## String指向的内容真的不可变吗

我们在上一节中学到，要接受更改时需要使用StringBuilder是因为它是可以更改的，而string是不可以更改的。

那么string指向的内容真的不可以更改吗？答案是不可以更改，也可以更改。

对于C#语言来说，你没有方法去更改，但是对于C语言来说是可以更改的。

示例：

C:

BOOL Ch5ModifyString(wchar\_t \*str,int size);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Unicode)]

public static extern bool Ch5ModifyString(string str, int size);

C#调用：

string str1 = "Helloworld";

string str2 = "Helloworld";

bool ret = Chapter5Native.Ch5ModifyString(str1, str1.Length);

如上所示C方法Ch5ModifyString接受一个字符串指针与指示它的长度（在函数内部我将所有的字符串填充为’A’）。

在C#中我们指出函数的参数为Unicode。对于string str我们没有做任何标记，因为我们已经说过标记[In][Out]对于string类型是无效的。

如果你对上一节中的内容使用StringBuilder来接受C语言的更改已经根深蒂固的话，那么你会觉得结果是不科学的。

在C#中调用方法Chapter5Native.Ch5ModifyString(str1, str1.Length);后，你可以观察到str1与str2指向的内容都变成了: "AAAAAAAAAA"。下面我来解释你的疑惑：

1：string类型不是不可以接受更改吗？

CLR为了优化调用C语言方法的效率，在传递参数时会采取一定的优化。其中之一就是当C语言中的参数是wchar\_t \*时，也就是Unicode是时，CLR会直接传递string在托管内存中的地址到C语言代码中。函数调用完成后，将不做任何处理。C语言中的代码拿到了地址，这对于C语言来说几乎什么都可以干了。在这里它将所有的内容修改成了’A’。

2：为什么修改了str1，str2的内容也跟着变了？

在CLR对于string有一个优化，也就是：所有字符串内容相应的string实例，都指向的是同一个地址。也就是str1和str2由于内容相同，他们指向的是同一个地址，当修改其中一个的时候，也就修改了另一个。

在本节所讲的东西，只是一个“小聪明“，大多数时候是不适用的，你应该总是使用StringBuiler来接受C语言的更改。

# 数组

在非托管与非托管中，数组都表示一块连续的内存区域。数组的起始地址实际上也就是首元素的地址。

**在数据使用的时候，如果要接受C代码的更改，需要加上标记[Out]。**

**在调用有数组作为参数的函数时，数组一定要合理的初始化。**

## 基础类型数组

对于基础数据类型的数组而言，其使用十分简单。

示例：

C:

BOOL Ch6ModifyArrayInt(int values[5]);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch6ModifyArrayInt([In, Out]int[] values);

C#调用：

var values = new int[5];

for (int i = 0; i < values.Length; i++)

{

values[i] = 123;

}

bool ret = Chapter6Native.Ch6ModifyArrayInt(values);

在上面的示例中，我们编写一个C修改int数组的代码。参数int[] values使用[In,Out]属性标记，表示参数需要传入，并且修改后的结果需要传回到C#中。

在调用的时候生成一了个int型数组，里面包含5个int元素。你可以根据该示例将扩展到byte,long,double,float等其它基础数据类型。

## 字符数组

事实上在C语言中，当我们讨论字符数组的时候，我们就是在讨论字符串，字符指针，内存块等一列的东西。但是在C#中，字符数组char[]与字符串(string)是完成不同的两个东西，虽然它们具有很多相似的地方，例如：内存布局，都是在某一个地方连续的储存数据。

示例：

C:

BOOL Ch6ModifyArrayChar(char values[5]);

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl, CharSet = CharSet.Ansi)]

public static extern bool Ch6ModifyArrayChar([In, Out] char[] values);

C#调用：

var values = new char[5];

for (int i = 0; i < values.Length; i++)

{

values[i] = 'B';

}

bool ret = Chapter6Native.Ch6ModifyArrayChar(values);

在上面的示例中，我们创建一个包含5个char的数组，并全部初始化成了’B’，然后调用Ch6ModifyArrayChar方法，该方法会将所有字符修改成为’A’。

属性标记中的Out表示，当调用C函数后，需要接受更改后的值。

## char[]与byte[]的陷阱

我见过一段类似下面的代码，

C:

Int \_\_stdcall GetDeviceData(char buf[1024]);。

C#:

[DllImport(“xxx.dll”,CharSet=CharSet.Ansi)]

int GetDeviceData([In,Out]char[] buf);.

C#调用:

Char[] buf=new char[1024];

Int ret=Native.GetDeviceData(buf);

Byte[] buf1= System.Text.Encoding.Unicode.GetBytes(buf);

//处理后续buf1

在上面的代码中，函数调用返回值是正确的，buf里面也包含有东西。但是处理buf1的时候他告诉我，数据不正确。长度比预期的长了1倍。数据由间隔的00组成。

在上面的代码中，一共有两个缺陷：

1：函数声明，这个没有全错。但不是最佳的方式。因为后来通过阅读文档知道，函数事实上需要的是一块仅是一块内存。但是由于C中没有byte类型，很多不规范的API都采用char\*或者 char[]。这使得该编程人员采用char[]作为参数。

2：System.Text.Encoding.Unicode.GetBytes(buf);是绝对不正确的。为什么了？由于函数返回的事实上0x00到0xFF之间的数据。但是由于CLR的作用，将每一个char(byte)被转换成了2个字节的unicode char。Unicode.GetBytes的时候数据。错误就这样发生了。

C语言中的指针是如此的强大与灵活以至于它们通常也是Bug的根源。如果一个函数需要仅仅是一块内存，那么它们绝对不应该使用char\*或者char []，而应该使用void \*。当调用C函数的编程人员看到void \*时，立即应该知道它应该对应的是byte[]，而不是char[]。而当看到char\*或者char[]则应该多长个心眼，搞清楚再做。

经过修改后代码：

C:

Int \_\_stdcall GetDeviceData(void \* buf, int size);。

C#:

[DllImport(“xxx.dll”,CharSet=CharSet.Ansi)]

int GetDeviceData([In,Out]byte[] buf);.

C#调用:

byte[] buf=new byte[1024];

Int ret=Native.GetDeviceData(buf,buf.Length);

//处理后续buf

## 结构数组

结构数组与基础类型的数组并没有太大的不同。

示例：

C:

typedef struct \_Ch6Dog

{

int age;

wchar\_t name[20];

}Ch6Dog;

BOOL Ch6ModifyArrayCh6Dog(Ch6Dog values[5]);

C#:

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]

public struct Ch6Dog

{

public int age;

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 20)]

public string name;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch6ModifyArrayCh6Dog([In, Out]Ch6Dog[] values);

C#调用：

var values = new Ch6Dog[5];

for (int i = 0; i < values.Length; i++)

{

values[i] = new Ch6Dog

{

age = 123,

name = "狗".PadRight(20, (char)0)

};

}

bool ret = Chapter6Native.Ch6ModifyArrayCh6Dog(values);

在上面的示例中，在C#中声明了Ch6Dog与C中的struct \_Ch6Dog 对应。C#中函数声明了一个参数Ch6Dog[] values来表示数组，采用[In,Out]标记，说明需要接受C函数的修改的结果(可以试试去掉Out观察调用后值是否更改)。当然在调用之前初始化了相应的数据。

关于结构的用法，我们将在第七章学习。

# Class与Struct

## 选择Class或Struct

如果你了解PInvoke，那么你一定知道：网上对使用class还是struct来声明C语言中对应的结构一直存在争议。

首先来说，class能用的地方struct都能用。Struct能用的地方class不一定能用。但是我还是推荐你使用class来声明与C中对应的结构。因为它有以几个方便的地方：

* 传递参数时，默认采用指针传递，可以提高性能。而结构默认是按值传递(标记ref或者out)可以按地址传递)。
* 可以包含无参构造函数，也可以编写字段初始化代码。而这些在结构中是不允许的。而且我们总是需要初始化相关的字段，为其他们分配内存。这一点所带来的方便，是让你值得使用class的。

当然在某些情况下你可能不得不使用struct：

* 当函数中的参数是结构数组时，如：void foo(Dog dog[5]);。此时使用class声明CLR不能正确的自动映射。
* 参数传递时采用的是值传递时，则只能使用结构，因为类不能映射成按值传递。
* 当一个结构中的字段成员是结构数组时，被包含的结构在C#中应当被声明成struct。如下面所示，则\_InStruct在C#中就必须声明为struct

Struct \_InStruct

{

Int age;

Wchar\_t name[20];

}

Struct \_MyStruct

{

Int cout;

\_InStruct InData[5];

}

## 使用基础数据 Int Double作为成员变量

首先我们使用基础数据类型作为成员变量，来看一个实例。

C:

typedef struct \_Ch7Basic

{

int iValue;

double dValue;

BOOL bWinValue;

bool bValue;

}Ch7Basic;

BOOL Ch7ModifyCh7Basic(Ch7Basic \*value);

C#:

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]

public class Ch7Basic

{

public int iValue = 123;

public double dValue = 123.123;

[MarshalAs(UnmanagedType.Bool)]

public bool bWinValue = false;

[MarshalAs(UnmanagedType.I1)]

public bool bValue = false;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7Basic(Ch7Basic value);

结构Ch7Basic十分简单，它几个基本的数据字段，我们在C#中声明的时候与之一一对应。但我还是要注意以下几点：

* Win32中的BOOL是4个字节，而C#中的bool是一个字节，所以需要使用[MarshalAs(UnmanagedType.Bool)]来标记。
* C++中的bool与C#中的bool类型一样都是一个字节，所以需要使用[MarshalAs(UnmanagedType.I1)]来标记。

## 使用Char字符作为成员变量

当使用字符时，情况就变得有点复杂。因为字符在非托管代码中有两中表现方式：char与wchar\_t。

在使用char时，又有几种情况：一种是全部为char或者全部为wchar\_t，另一种是包含两种的。

示例：

C:

typedef struct \_Ch7CharA

{

char acValue1;

char acValue2;

}Ch7CharA;

BOOL Ch7ModifyCh7CharA(Ch7CharA \*value);

C#:

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]

public class Ch7CharA

{

public char acValue1 = 'B';

public char acValue2 = 'B';

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7CharA(Ch7CharA value);

C#调用：

var value = new Ch7CharA();

bool ret = Ch7Native.Ch7ModifyCh7CharA(value);

在上面的示例中，我们遇到的是第一种情况，所有的字符都是ansi的（我希望你能够将此种情况自动应用到Unicode char）。在类的StructLayout中有一个属性CharSet我们标记为Ansi表示该结构所有与字符，字符串相关的变量都应该被映射成char或者char\*。

示例：

C:

typedef struct \_Ch7CharAW

{

wchar\_t wcValue;

char acValue;

}Ch7CharAW;

BOOL Ch7ModifyCh7CharAW(Ch7CharAW \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7CharAW

{

[MarshalAs(UnmanagedType.U2)]

public char wcValue = '你';

[MarshalAs(UnmanagedType.U1)]

public char acValue = 'B';

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7CharAW(Ch7CharAW value);

C#调用:

var value = new Ch7CharAW();

bool ret = Ch7Native.Ch7ModifyCh7CharAW(value);

在结构Ch7StringAW中结构稍微变得有点复杂，它包含一个ansi char与unicode wchar\_t。此时我们不能在类的StructLayout上标记CharSet来解决问题，因为这个标记是针对整个类的。

事实上在C中wchar\_t可以表示为2个字节的无符号整数unsigned ushort，而char则是1个字节的无符号整数。

所以：我们在wcValue字段上使用[MarshalAs(UnmanagedType.U2)]来表示该字段应该被映射成成C中的2字节无符号整数。而acValue字段上使用[MarshalAs(UnmanagedType.U1)]表示该字段应该被映射成C中的1个字节无符号整数。

幸运的是这一切CLR都是支持的。

## 使用字符数组

在类中的字符串数组变得有一点复杂，当然也分两种情况：一种情况是所有的字符编码格式相同，另一个包含混合编码格式。

示例：

C:

typedef struct \_Ch7StringA

{

char asValue1[20];

char asValue2[20];

}Ch7StringA;

BOOL Ch7ModifyCh7StringA(Ch7StringA \*value);

C#:

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]

public class Ch7StringA

{

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 20)]

public string asValue1 = "BBBB".PadRight(20, (char)0);

[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 20)]

public string asValue2 = "BBBB".PadRight(20, (char)0);

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7StringA(Ch7StringA value);

C#调用：

var value = new Ch7StringA();

bool ret = Ch7Native.Ch7ModifyCh7StringA(value);

在上面的示例中，我们仍然以类中只包含一种编码类型这种简单的方式来开始讲解。

类Ch7StringA中包含两个均为ansi char数组的字段asValue1与asValue2，并且长度均为20。

标记[MarshalAs(UnmanagedType.ByValTStr, SizeConst = 20)]:

* ByValTStr表示相应的字段应该被映射成：结构中内嵌的字符数组。
* SizeConst表示数组的长度为20。
* StructLayout中的CharSet=CharSet.Ansi表示凡是使用上述标记中的每一个字段中的数组元素增均为Ansi。

一切就这样简单，如果一个结构中所有的字符串数组的编码方式都一样，那么按照上面的试，一切就可以运行了。

当一个结构中同时包含多种编码方式时，由于CLR的一些规则，事情就变得有点复杂。

示例：

C:

typedef struct \_Ch7StringAW

{

wchar\_t wsValue[20];

char asValue[20];

}Ch7StringAW;

BOOL Ch7ModifyCh7StringAW(Ch7StringAW \*value);

C#:

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]

public struct Ch7StringAW

{

[MarshalAs(UnmanagedType.ByValArray, ArraySubType = UnmanagedType.U2, SizeConst = 20)]

public char[] wsValue;

[MarshalAs(UnmanagedType.ByValArray, ArraySubType = UnmanagedType.U1, SizeConst = 20)]

public char[] asValue;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7StringAW(ref Ch7StringAW value);

C#调用：

var value = new Ch7StringAW

{

asValue = "BBBB".PadRight(20, (char)0).ToArray(),

wsValue = "你你你你".PadRight(20, (char)0).ToArray(),

};

bool ret = Ch7Native.Ch7ModifyCh7StringAW(ref value);

在结构中Ch7StringAW中同时一个为Ansi字符编码的asValue与一个为Unicode编码的wsValue。而在C#中我们采用的是char[]来做映射的原型。

那么此处为什么不能使用string了？

原因：string只能映射成指针或者字符串数组。并且其对应的编码方式只受StructLayout中的CharSet控制。

所以：当一个类中同时包含两种编码方式时，如果我们使用string并声明为字符串数组，它么只能被映射成ansi或者unicode。很显然如果我们将StructLayout中的 CharSet标记为Unicode那么字段asValue将映射出错，而将StructLayout中的CharSet标记为Ansi那么字段wsValue将映射出错。所以我们只能使用char[]来完成该项工作。

事实上采用char[]看上去才是最正统的方式，但是string往往更给我们带来更多的便利。所以在能用string的情况下，我们通常使用string。

任何数组在映射成结构中的内嵌数组时，均使用UnmanagedType.ByValArray,作为标记，同时使用ArraySubType来声明每一个数组元素的类型，而SizeConst指明了整个数组中的元素个数。

在此处，对于ansi 而言，其每个元素类型为1字节的无符号数:U1。而对于unicode而言其每个元素类型为2字节的无符号数:U2。

SizeConst=20说明每一个数组中的元素个数均为20个。

**建议：通常在一个结构或者类中要么都使用ANSI字符串，或者Unicode字符串。**

## 使用字符指针

在结构中使用字符指针时，远比使用字符数组简单，因为CLR支持对指针的多种传递方式。

示例：

C:

typedef struct \_Ch7StringPtr

{

wchar\_t \* pwsValue;

char \* pasValue;

}Ch7StringPtr;

BOOL Ch7ModifyCh7StringPtr(Ch7StringPtr \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7StringPtr

{

[MarshalAs(UnmanagedType.LPWStr)]

public string pwsName;

[MarshalAs(UnmanagedType.LPStr)]

public string pasName;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7StringPtr([In, Out]Ch7StringPtr value);

C#调用：

var value = new Ch7StringPtr();

bool ret = Ch7Native.Ch7ModifyCh7StringPtr(value);

对于结构Ch7StringPtr来说，它包含一个指针Unicode字符串的指针pwsValue的指针与一个Ansi字符串的指针。

在C#中我们采用string来对应两个变量，但是需要注意的是pwsValue被标记成LPWStr：表示该成员被映射成Unicode指针也就是wchar\_t\*，而pasValue被标记成LPStr：表示该成员被映射成Ansi指针也就是char\*。

无论是char还是string总是需要被标记相应的映射类型以便正确映射。

## 使用内嵌结构

在结构中使用结构是一种常见的形式。在这种情况下结构所有的内容都做为另一个为另一个类的一部分。

示例：

C:

typedef struct \_Ch7Struct

{

Ch7StringAW ch7StringAW;

}Ch7Struct;

BOOL Ch7ModifyCh7Struct(Ch7Struct \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7Struct

{

[MarshalAs(UnmanagedType.Struct)]

public Ch7StringAW ch7StringAW;

public Ch7Struct()

{

this.ch7StringAW = new Ch7StringAW

{

asValue = "我和你".PadRight(20, (char)0).ToArray(),

wsValue = "我和你".PadRight(20, (char)0).ToArray(),

};

}

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7Struct([In, Out]Ch7Struct value);

C#调用：

var value = new Ch7Struct();

bool ret = Ch7Native.Ch7ModifyCh7Struct(value);

在示例中Ch7Struct包含一个结构体成员ch7StringAW(该类在前面章节中的 使用字符数组中声明)。如果你在前面章节足够仔细，你会发现只有Ch7StringAW是被声明成Struct的，而其它的都是class。这也是我们在本章开头所提到的那样：如果一个结构被包含在另一个结构中或者该结构的数组形式被包含在另一个结构中，则建议将该结构声明成struct，而不是class。

在字段ch7StringAW的声明中标记得有[MarshalAs(UnmanagedType.Struct)]，表示该成员被映射成内嵌的结构，事实上这也是struct类型的默认方式。可以不用标记。

在类Ch7Struct构造函数中，我们初始化的内部成员ch7StringAW。因为Ch7StringAW声明结构，我们不能声明不带参数的构造函数，这使得我们必须在外部初始化。

## 使用内嵌结构数组

如果你还记得前面所学习的如何使用字符串数组作为结构成员变量，那么你一定不会对内嵌数组陌生。事实上字符串数组也是一种类型的数组，但有点特殊，因为系统允许我们使用string类型来做映射，从而简化编程。在本节中，我们将学习如何使用基础数据类型作为结构中的内嵌数组。

示例：

C:

typedef struct \_Ch7StructArray

{

Ch7StringAW ch7StringAWArray[4];

int iArray[4];

}Ch7StructArray;

BOOL Ch7ModifyCh7StructArray(Ch7StructArray \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7StructArray

{

[MarshalAs(UnmanagedType.ByValArray, ArraySubType = UnmanagedType.Struct, SizeConst = 4)]

public Ch7StringAW[] ch7StringAWArray;

[MarshalAs(UnmanagedType.ByValArray, ArraySubType = UnmanagedType.I4, SizeConst = 4)]

public int[] iArray;

public Ch7StructArray()

{

this.ch7StringAWArray = new Ch7StringAW[4];

for (int i = 0; i < this.ch7StringAWArray.Length; i++)

{

this.ch7StringAWArray[i] = new Ch7StringAW

{

asValue = "BBBB".PadRight(20, (char)0).ToArray(),

wsValue = "你你你你".PadRight(20, (char)0).ToArray(),

};

}

this.iArray = new int[] { 123, 123, 123, 123 };

}

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7StructArray([In, Out]Ch7StructArray value);

C#调用：

var value = new Ch7StructArray();

bool ret = Ch7Native.Ch7ModifyCh7StructArray(value);

在结构Ch7StructArray中我们声明了两个数组字段分别为ch7StringAWArray与iArray。在使用字符数组时我们学习过，标记数组的方式ByValArray。

对于ch7StringAWArray来说其ArraySubType为Struct，对于iArray来说其ArraySubType为I4(4字节有符号整数)。

SizeConst都为4。

这里我再提醒一句：对于内嵌的结构与内嵌的结构数组，被内嵌的类型应当声明成struct（如这里的Ch7StringAW一样），以避免不能正确映射。

## 使用指针

在字符指针中我们可以使用LPWStr也可以使用LPStr来完成映射工作，但是对于其它类型的结构没有那么幸运CLR不支持自动将其映射成指针形式。也就是说我们需要做更多的工作。

示例：

C:

typedef struct \_Ch7Pointer

{

Ch7StringAW \*pch7StringAWPtr;

int \*iValuePtr;

}Ch7Pointer;

BOOL Ch7ModifyCh7Pointer(Ch7Pointer \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7Pointer

{

public IntPtr ch7StringAWPtr = IntPtr.Zero;

public IntPtr iValuePtr = IntPtr.Zero;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7Pointer([In, Out]Ch7Pointer value);

C#调用：

//create objects to use

var value = new Ch7Pointer();

var ch7StringAW = new Ch7StringAW

{

asValue = "BBBB".ToArray(),

wsValue = "你你你你".ToArray(),

};

int iValue = 123;

//alloc memory from process heap

value.ch7StringAWPtr = Marshal.AllocHGlobal(Marshal.SizeOf(ch7StringAW));

value.iValuePtr = Marshal.AllocHGlobal(Marshal.SizeOf(iValue));

//copy managed object memory to native heap

Marshal.StructureToPtr(ch7StringAW, value.ch7StringAWPtr, false);

Marshal.StructureToPtr(iValue, value.iValuePtr, false);

//call method

if (Ch7Native.Ch7ModifyCh7Pointer(value))

{

//copy native heap to managed object memory

ch7StringAW = (Ch7StringAW)Marshal.PtrToStructure(value.ch7StringAWPtr, typeof(Ch7StringAW));

iValue = (int)Marshal.PtrToStructure(value.iValuePtr, typeof(int));

}

//do not forgot free native heap memory

Marshal.FreeHGlobal(value.ch7StringAWPtr);

Marshal.FreeHGlobal(value.iValuePtr);

value.ch7StringAWPtr = IntPtr.Zero;

value.iValuePtr = IntPtr.Zero;

由于CLR不具备自动映射指针的功能（也许将来会增加），所以我们必须声明IntPtr原型与C中的指针对应。

声明是十分简单的，但是使用的时候有些复杂，但总是来说包括以下几个步骤：

1：创建C#对象，并复制。

2：使用Marshal类的方法或者其它自定义的类（如调用C中的malloc）从非托管内存中分配相应长度的内存。

3：使用Marshal方法将第一步创建的对象复制到第二步的分配的内存中。

4：调用相应的方法。

5：检查函数返回值，并根据需要将非托管内存中被修改后的数据复制回创建的C#对象中。

6：释放第二步在非托管内存中分配的内存，并清空相应指针。

## 使用指针数组

使用指针数组的基础步骤与使用单个指针基本一致。唯一不同的是在内存分配的时候需要操作多次，当然也可以一次分配完成。前者会分配多块内存，这些内存可能不连续在一起，这种方式性能较低，单兼容性好。后者分配一块连续的内存，性能较好，但兼容性差一些。当然这些都取决于实际的情况。

示例：

C:

typedef struct \_Ch7PointerArray

{

Ch7StringAW \*pch7StringAWPtrArray[4];

int \*iValuePtrArray[4];

}Ch7PointerArray;

BOOL Ch7ModifyCh7PointerArray(Ch7PointerArray \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch7PointerArray

{

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]

public IntPtr[] pchStringAWPtrArray = new IntPtr[4];

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]

public IntPtr[] iValuePtrArray = new IntPtr[4];

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch7ModifyCh7PointerArray(Ch7PointerArray value);

C#调用：

//create objects to use

var value = new Ch7PointerArray();

int[] iValues = new int[] { 123, 123, 123, 123 };

Ch7StringAW[] ch7StringAWs = new Ch7StringAW[4];

for (int i = 0; i < ch7StringAWs.Length; i++)

{

ch7StringAWs[i] = new Ch7StringAW

{

asValue = "BBBB".ToArray(),

wsValue = "你你你你".ToArray(),

};

}

//alloc memory from process heap

for (int i = 0; i < 4; i++)

{

value.iValuePtrArray[i] = Marshal.AllocHGlobal(sizeof(int));

value.pchStringAWPtrArray[i] = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Ch7StringAW)));

}

//copy managed object memory to native heap

for (int i = 0; i < 4; i++)

{

Marshal.StructureToPtr(iValues[i], value.iValuePtrArray[i], false);

Marshal.StructureToPtr(ch7StringAWs[i], value.pchStringAWPtrArray[i], false);

}

if (Ch7Native.Ch7ModifyCh7PointerArray(value))

{

for (int i = 0; i < 4; i++)

{

ch7StringAWs[i] = (Ch7StringAW)Marshal.PtrToStructure(value.pchStringAWPtrArray[i], typeof(Ch7StringAW));

iValues[i] = (int)Marshal.PtrToStructure(value.iValuePtrArray[i], typeof(int));

}

}

for (int i = 0; i < 4; i++)

{

Marshal.FreeHGlobal(value.iValuePtrArray[i]);

Marshal.FreeHGlobal(value.pchStringAWPtrArray[i]);

value.iValuePtrArray[i] = IntPtr.Zero;

value.pchStringAWPtrArray[i] = IntPtr.Zero;

}

}

示例中的结构Ch7StructPtrArray包含两个指针数组。在C#中声明了与之对应的IntPtr[]数组来与之对应。数组中的每一个元素都是IntPtr类型。C#中的类默认已经为每一个数组分配好4个元素长度。

C#调用2：

//create objects to use

var value = new Ch7PointerArray();

int[] iValues = new int[] { 123, 123, 123, 123 };

Ch7StringAW[] ch7StringAWs = new Ch7StringAW[4];

for (int i = 0; i < ch7StringAWs.Length; i++)

{

ch7StringAWs[i] = new Ch7StringAW

{

asValue = "BBBB".ToArray(),

wsValue = "你你你你".ToArray(),

};

}

//alloc block memory from process heap

IntPtr ptrIValues = Marshal.AllocHGlobal(4 \* 4);

IntPtr ptrCh7StringAW = Marshal.AllocHGlobal(4 \* Marshal.SizeOf(typeof(Ch7StringAW)));

//cal each object address

for (int i = 0; i < 4; i++)

{

value.iValuePtrArray[i] = ptrIValues + i \* 4;

value.pchStringAWPtrArray[i] = ptrCh7StringAW + i \* Marshal.SizeOf(typeof(Ch7StringAW));

}

//copy managed object memory to native heap

for (int i = 0; i < 4; i++)

{

Marshal.StructureToPtr(iValues[i], value.iValuePtrArray[i], false);

Marshal.StructureToPtr(ch7StringAWs[i], value.pchStringAWPtrArray[i], false);

}

if (Ch7Native.Ch7ModifyCh7PointerArray(value))

{

for (int i = 0; i < 4; i++)

{

ch7StringAWs[i] = (Ch7StringAW)Marshal.PtrToStructure(value.pchStringAWPtrArray[i], typeof(Ch7StringAW));

iValues[i] = (int)Marshal.PtrToStructure(value.iValuePtrArray[i], typeof(int));

}

}

Marshal.FreeHGlobal(ptrIValues);

Marshal.FreeHGlobal(ptrCh7StringAW);

for (int i = 0; i < 4; i++)

{

value.iValuePtrArray[i] = IntPtr.Zero;

value.pchStringAWPtrArray[i] = IntPtr.Zero;

}

ptrCh7StringAW = IntPtr.Zero;

ptrCh7StringAW = IntPtr.Zero;

在第二种调用方式中，我们调用Marshal.AllocHGlobal(4 \* 4)一次性分配内存。这表示一个4个元素，每一个元素int 4个字节。

这两种方式各有其好处，但我还是推荐你使用第一种方式。虽然它性能较低，但对于现在计算机来说大多数时候可以忽略，但是它更清晰、易懂。特别是如果C代码有错，造成访问越界的错误更容易被发现。

# 联合Union

联合这种数据类型，我个人认为是C语言中十分失败的数据类型，虽然很多时候它能发挥作用，并简化编程。

联合在内存中各个成员变量共享一块内存，按照给定方式排列，对齐。

## 基础数据类型联合

对于基础数据类型来说，联合并没有什么神秘的地方，我们只需要像平时编程一样。

示例：

C:

typedef enum \_Ch8DataType

{

INT\_TYPE,

DOUBLE\_TYPE,

CH8SAMPLE\_TYPE

}Ch8DataType;

typedef union \_Ch8Sample

{

int iValue;

double dValue;

}Ch8Sample;

BOOL Ch8ModifyCh8Sample(Ch8Sample \*value, Ch8DataType ch8DataType);

C#:

public enum Ch8DataType : int

{

INT,

DOUBLE,

CH8STRUCT,

}

[StructLayout(LayoutKind.Explicit)]

public class Ch8Sample

{

[FieldOffset(0)]

public int iValue = 123;

[FieldOffset(0)]

public double dValue = 123.123;

}

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8Sample([In, Out]Ch8Sample value, Ch8DataType ch8DataType);

C#调用：

bool ret = false;

var value = new Ch8Sample();

value.iValue = 123;

ret = Ch8Native.Ch8ModifyCh8Sample(value, Ch8DataType.INT);

value.dValue = 123.123;

ret = Ch8Native.Ch8ModifyCh8Sample(value, Ch8DataType.DOUBLE);

在上面的示例代码中，我们首先声明了一个数据类型的枚举Ch8DataType，该类型在后的示例中都将使用。请注意这点。

联合Ch8Sample顾名思义，是较为简单的类型，它内部只包含有基础类型，一个Int数据和一个double数据。

在声明联合对应的结构时，一定要注意使用:

[StructLayout(LayoutKind.Explicit)]该标记表示：类或者结构中的字段成员将按照特定的序列与起始地址排列。元素的起始地址由标记FieldOffset决定，在Ch8Sample中，由于只有两个，所以其地址我们与C代码对应从0开始。其内存结构如下图：

## 非基础数据类型联合

首先我们需要清楚什么是非基础数据类型？在这里我指的是那些除去int,byte,double,float,int64,char等基础类型。由于一些特殊原因（本书不会深入解释）CLR不能将非基础数据类型与基础数据类型排址在相同的偏移地址。这就给我们带了相应的麻烦。

示例：

C:

typedef union \_Ch8Complex

{

int IValue;

Ch8Sample ch8SampleValue;

}Ch8Complex;

BOOL Ch8ModifyCh8Complex(Ch8Complex \*value, Ch8DataType ch8DataType);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch8ComplexCh8Sample

{

[MarshalAs(UnmanagedType.Struct)]

public Ch8Sample ch8SampleValue = new Ch8Sample();

}

[StructLayout(LayoutKind.Sequential)]

public class Ch8ComplexINT

{

public int iValue = 123;

}

[DllImport(DLLNAME, EntryPoint = "Ch8ModifyCh8Complex", CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8ComplexCh8Sample([In, Out]Ch8ComplexCh8Sample value, Ch8DataType ch8DataType);

[DllImport(DLLNAME, EntryPoint = "Ch8ModifyCh8Complex", CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8ComplexINT([In, Out]Ch8ComplexINT value, Ch8DataType ch8DataType);

C#调用：

bool ret = false;

var value = new Ch8ComplexCh8Sample();

value.ch8SampleValue.dValue = 123.123;

ret = Ch8Native.Ch8ModifyCh8ComplexCh8Sample(value, Ch8DataType.CH8STRUCT);

bool ret = false;

var value = new Ch8ComplexINT();

value.iValue = 123;

ret = Ch8Native.Ch8ModifyCh8ComplexINT(value, Ch8DataType.INT);

在上面的示例中Ch8Complex包含一个int与一个Ch8Sample字段，并都在起始地址0处。在C语言中，这没有任何问题，编译器知道如何生成正确的代码并访问，操纵它们。但是在C#中的就不一样，这一点前面已经讲过：CLR不能将非基础数据类型与基础数据类型排址在相同的偏移地址。

所以我们将联合所有可能用到情况分开来，这是一个笨方法不是嘛？答案：是的，但这是我所知道唯一的方法。（如果你发现的更简单的方法，请告诉我）。

对于Int iValue的情况我们编写对应的类，并且为其编写方法Ch8ModifyCh8ComplexINT。

对于ch8SampleValue的情况我们编写对应的类Ch8ComplexCh8Sample，并且为其编写方法Ch8ModifyCh8ComplexCh8Sample。

请注意两个类，在类的标记中我们使用的是LayoutKind. Sequential，而不是LayoutKind. Explicit。这是因为这两个类都很简单，它们没有其它的字段了，使用Sequential与使用Explicit都可以，不会有任何的不同。但是你如果有更多的字段则你可能需要使用Explicit并使用FieldOffset指定偏移地址。

由于在C#中不能存在两个名称相同，参数也相同的函数。所以我们分别为操纵类Ch8ComplexINT与Ch8ComplexCh8Sample编写了一个函数。但是请注意这两个函数的EntryPoint = "Ch8ModifyCh8Complex"，这表示这两个函数真正调用的C函数是Ch8ModifyCh8Complex而不是Ch8ComplexINT或者Ch8ComplexCh8Sample，事实上这两个函数在C代码也根本不存在。

现在让我们来总结一下基础步骤：

1：针对每一种可能的情况编写相应的类（当然有可能多种情况使用的类可以相同）。

2：针对每一个在第一步编写的类，编写一个到多个函数。对于由C中某一个函数声明的函数，这经些函数的EntryPoint总是相同。

3：生成类，并进行调用。

## 结构中包含联合

在联合的使用中形式中，最常见的其实是在结构中。结构中通常包含一个字段来用指明它所包含的数据类型，程序根据该字段进行后续操作。

示例：

C:

typedef struct \_Ch8StructWithUnion

{

Ch8DataType ch8DataType;

union \_InneralUnion

{

int iValue;

Ch8Sample ch8SampleValue;

}InneralUnion;

}Ch8StructWithUnion;

BOOL Ch8ModifyCh8StructWithUnion(Ch8StructWithUnion \*value);

C#:

[StructLayout(LayoutKind.Sequential)]

public class Ch8StructWithUnionINT

{

[MarshalAs(UnmanagedType.I4)]

private Ch8DataType ch8DataType = Ch8DataType.INT;

/// <summary>

/// In c code InneralUnion is 8 byte alignment so append here 4 byte

/// </summary>

private int alignment = -1;

public int iValue = 123;

}

[StructLayout(LayoutKind.Sequential)]

public class Ch8StructWithUnionCh8Sample

{

[MarshalAs(UnmanagedType.I4)]

private Ch8DataType ch8DataType = Ch8DataType.CH8SAMPLE;

[MarshalAs(UnmanagedType.Struct)]

public Ch8Sample ch8SampleValue = new Ch8Sample();

}

[DllImport(DLLNAME, EntryPoint = "Ch8ModifyCh8StructWithUnion", CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8StructWithUnionINT([In, Out]Ch8StructWithUnionINT value);

[DllImport(DLLNAME, EntryPoint = "Ch8ModifyCh8StructWithUnion", CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8StructWithUnionCh8Sample([In, Out]Ch8StructWithUnionCh8Sample value);

结构Ch8StructWithUnion包含一个字段ch8DataType用来指明联合字段InneralUnion中所包含的数据类型。由于该联合有两种可能，所以我们编写了两个类：

Ch8StructWithUnionINT对应的方法为Ch8Modify Ch8StructWithUnionINT。

Ch8StructWithUnionCh8Sample对应的方法为Ch8ModifyCh8StructWithUnionCh8Sample。与前一节的示例一样两个方法的EntryPoint值都为：Ch8ModifyCh8StructWithUnion。

类Ch8StructWithUnionINT与Ch8StructWithUnionCh8Sample中的字段ch8DataType我们都声明为private这是希望外部代码不可以更改该值，因为它们的值在这种情况下是确定的。

最后需要说明的是在类Ch8StructWithUnionINT中，还有一个字段：private int alignment = -1;该字段为保留字段外部程序不能访问。那为什么要声明这个字段了？

正如该字段的名称一样，它用于对齐。在C代码中联合中长度最大的类型是第二个：Ch8Sample。而Ch8Sample又是一个int与double联合组成。也就是说Ch8Sample对齐的地址为以8开始的位置。如图所示：

在C#中如果不手动添加该对齐字段，则iValue字段将被映射到图中的错误位置。

而类Ch8StructWithUnionCh8Sample由于第二个字段是类CLR已经知道如何自动对齐了。

**注：不论是在C中还是在C#结构成员默认对齐都是8字节，该值可能由于不同的编译条件不同而不同。所以在C#调用C时一定要清楚编译C代码时使用的对齐字节数。**

## 联合参数传递问题

在前面的示例代码中，你可能已经注意到一个事实：C代码中的结构体大小总是等于或者大于C#中的结构体大小。因为C#中的每一个结构体只是为了适应一个方面，所以它们只包含一部分数据。Ch8StructWithUnionINT类比Ch8StructWithUnion结构小，而Ch8StructWithUnionCh8Sample 的大小与Ch8StructWithUnion相同。

既然传入的结构体大小不一样，那么为什么前面的示例代码又可以正常运行了？原因有以下两点：

代码的正确性：C代码可以保证，在操作某一种数据类型下绝对不会操作其它的数据，也就是说永远不会发生访问违规这种错误。这使得对于程序来说其它的数据好像根本不存在一样。例如在数据类型为INT的时候，永远不会去操作Ch8Sample数据。

参数采用的是指针传递。在采用指针传递的时候，真正传递的是的4个字节的地址（64位为8个字节）。这样不论函数是什么调用约定都可以正确的入栈与出栈。

接下来让我们看一下两个示例。

示例：

C:

BOOL Ch8ModifyCh8StructWithUnionByValue(Ch8StructWithUnion value);

C#:

[StructLayout(LayoutKind.Sequential)]

public struct Ch8StructWithUnionINT2

{

[MarshalAs(UnmanagedType.I4)]

public Ch8DataType ch8DataType;

/// <summary>

/// In c code InneralUnion is 8 byte alignment so append here 4 byte

/// </summary>

private int reserved;

public int iValue;

}

[DllImport(DLLNAME, EntryPoint = "Ch8ModifyCh8StructWithUnionByValue", CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch8ModifyCh8StructWithUnionByValue([MarshalAs(UnmanagedType.Struct)]Ch8StructWithUnionINT2 value);

C#调用：

Ch8StructWithUnionINT2 value = new Ch8StructWithUnionINT2

{

iValue = 123,

ch8DataType = Ch8DataType.INT

};

bool ret = Ch8Native.Ch8ModifyCh8StructWithUnionByValue(value);

上面的函数与见过的Ch8ModifyCh8StructWithUnion并没有什么不同，唯一不同的是参数由指针传递变成了值传递。由于要进行值传递，我增加了Ch8StructWithUnionINT2与它Ch8StructWithUnionINT也基本一样，最大的区别是从class变成了struct，因为只有struct才能在传递时映射成值传递的结构形式。CLR在调用Ch8ModifyCh8StructWithUnionByValue的时候，会将参数全部复制一份到C代码的栈上。当然由于是值传递调用，在C#代码中看不到相应的修改结果。

运行上面的示例你会发现一切都是正确的。但是你会发现我们传递的Ch8StructWithUnionINT2的大小比C中的Ch8StructWithUnion结构要小。也就是说栈上的数据并没有实际的多（相对于通常的C函数调用C函数所传递的参数来说）。但是它真的能正常运行。

其原因就在于：函数的调用方式采用的是默认的C语言调用约定cdel。CLR在调用的时候将参数入栈，在调用完成后CLR将参数出栈。所以说在函数的调用的前后栈是平衡的，不会出现异常。而且我们的代码也保证了在处理某种数据类型时，不会访问其它的的数据，也就不会出现内存访问违规。这两点就保证了程序的正常运行。

那么前面的示例是否可以应用到其它的情况了？如stdcall,fastcall等。答案是：如果压入栈的参数长度与函数声明的长度一样则可以。否则将会失败。因为栈会不平衡。

示例：

C:

BOOL \_\_stdcall Ch8ModifyCh8StructWithUnionByValueStd(Ch8StructWithUnion value);

C#:

[DllImport(DLLNAME, EntryPoint = "\_Ch8ModifyCh8StructWithUnionByValueStd@16", CallingConvention = CallingConvention.StdCall)]

public static extern bool Ch8ModifyCh8StructWithUnionByValueStd([MarshalAs(UnmanagedType.Struct)] Ch8StructWithUnionINT2 value);

C#调用：

Ch8StructWithUnionINT2 value = new Ch8StructWithUnionINT2

{

iValue = 123,

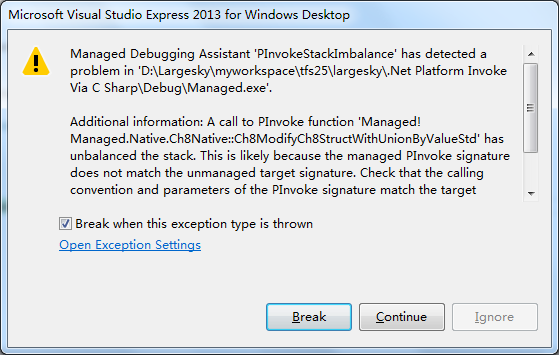
ch8DataType = Ch8DataType.INT

};

bool ret = Ch8Native.Ch8ModifyCh8StructWithUnionByValueStd(value);

函数Ch8ModifyCh8StructWithUnionByValueStd对比Ch8ModifyCh8StructWithUnionByValue来说除了函数名称不一样外，重要的一点就是前者的调用约定方式为:stdcall。

什么是stdcall？stdcall最重要的一点就是由函数自身完成栈参数清理（请参见第三章 方法调用约定）。

在VS2013中运行上面的代码，你会得到如下的提示：

上图的大致意思说是：检测到栈不平衡。

因为在调用的时候CLR根据Ch8StructWithUnionINT2压入栈的参数长度为12个字节，而函数Ch8ModifyCh8StructWithUnionByValueStd内部清理的栈参数为16个字节（也就是Ch8StructWithUnion的大小）。VS检测到了这种错，并进行了相应的提示。

为了避免这种错误我们可以有几个种方式：

1：使用int,byte等对结构进行补足，但是有一点记住补足的字段通常是private的，程序从来不使用。

对Ch8StructWithUnionINT2补足后：

[StructLayout(LayoutKind.Sequential)]

public struct Ch8StructWithUnionINT2

{

[MarshalAs(UnmanagedType.I4)]

public Ch8DataType ch8DataType;

/// <summary>

/// In c code InneralUnion is 8 byte alignment so append here 4 byte

/// </summary>

private int reserved;

public int iValue;

private int reserved1;

}

在这里我们增加了一个字段reserved1，使得结构的长度为16个字节，从而达到需要的长度。

2：使用指针传递联合的参数。

3：函数调用使用cdel约定。

# 指针

指针在一定程序上又可以称为地址，它里面的值是某一个地址，该地址可以指向程序中的任意地址（当然实际能不能用不一定）。指针通常指针间接来访问真正的数据。

在C#中如果希望传递一个变量的指针使用ref。但是C#中不能表示二级或者更多级指针。

## 一级指针

对于基础数据类型而言在参数前面加上ref就表示参数传递的指针。

对于class或者struct而言它们在传递时默认就采用的是指针传递，如果在其前面加上ref则表示C语言中的二级指针。

由于在前面的大量章节我们都已经描述了相应的如ref int ref double class等用法，所以我不会在此处编写示例。

## 二级指针

二级指针：我们常说的指向指针的指针。对于class与struct来说，标记ref就可以表示二级指针，但是对于基础数据类型来说我们还需要做更多的工作。

示例：

C:

BOOL Ch9AllocInt(int \*\*value)

{

if (value == 0)

{

return FALSE;

}

\*value = (int \*)malloc(sizeof(int));

if (\*value != 0)

{

\*\*value = 1234;

}

return \*value != 0;

}

void Ch9FreeMemory(void \*value)

{

free(value);

}

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch9AllocInt(ref IntPtr value);

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern void Ch9FreeMemory(IntPtr ptr);

C#调用：

IntPtr value = IntPtr.Zero;

bool ret = Ch9Native.Ch9AllocInt(ref value);

if (ret)

{

int iValue = (int)Marshal.PtrToStructure(value, typeof(int));

}

Ch9Native.Ch9FreeMemory(value);

由于C#中不能直接表示基础数据类型的二级指针，所以我们在C#中声明中参数采用ref IntPtr value，表示指向指针的指针。在调用的方法时同样加上ref标记，调用完成后value的值指向就是我们要数据。使用PtrToStructure将数据非托管内存复制到托管内存iValue里，它的值为1234。最后使用Ch9FreeMemory释放所申请的内存。

**注意：在编程中通常内存由谁申请，就由谁负责释放。所以这里我们在C代码中提供了一个方法Ch9FreeMemory，它负责释放内存。该方法在以后都会用到。**

关于基础数据类型的二级指针就讲到这里，我希望你能自动将该知识应用到其它的类型如double，char，bool等变量中。下面我们来看一下struct和class的二级指针。

示例：

C:

BOOL Ch9AllocCh9Dog(Ch9Dog \*\*value)

{

if (value == 0)

{

return FALSE;

}

\*value = (Ch9Dog \*)malloc(sizeof(Ch9Dog));

if (\*value != 0)

{

(\*value)->iValue = 1234;

wcscpy\_s((\*value)->wsValue, 4, L"我我我");

}

return \*value != 0;

}

C#:

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch9AllocCh9Dog(ref IntPtr value);

C#调用：

IntPtr value = IntPtr.Zero;

if (Ch9Native.Ch9AllocCh9Dog(ref value))

{

Ch9Dog dog = (Ch9Dog)Marshal.PtrToStructure(value, typeof(Ch9Dog));

}

Ch9Native.Ch9FreeMemory(value);

仔细观察，这种方式与处理基础数据类型的基本一样。但是我们还有一种方式：

C：

BOOL Ch9CoTaskMemAllocCh9Dog(Ch9Dog \*\*value)

{

if (value == 0)

{

return FALSE;

}

\*value = (Ch9Dog \*)CoTaskMemAlloc(sizeof(Ch9Dog));

if (\*value != 0)

{

(\*value)->iValue = 1234;

wcscpy\_s((\*value)->wsValue, 4, L"我我我");

}

return \*value != 0;

}

C#：

[DllImport(DLLNAME, CallingConvention = CallingConvention.Cdecl)]

public static extern bool Ch9CoTaskMemAllocCh9Dog(ref Ch9Dog value);

C#调用：

Ch9Dog dog = new Ch9Dog();

bool ret = Ch9Native.Ch9CoTaskMemAllocCh9Dog(ref dog);

与上面的示例比较，我们发现有几点不同：

* 第一个在分配内存的使用的是c函数malloc，而第二个使用的是CoTaskMemAlloc。关于这两个函数你可以简单的认为它们在进程中不同的堆中分配内存。

## 指针作为函数返回值

# 回调函数

## 回调函数实现机制

## 声明回调函数的委托

## 函数回调示例

# 综合示例

假设，你正在负责一套视频监控的软件。并且你的老板告诉你，市场部人职员告诉他，客户对市场部的人员说：

客户：“Hi xx，我们需要一套功能强大，操作简单，界面漂亮的视频监控软件，而且我们可能会有很多种不同类型的视频设备”。

市场部人员：”好的，凭借我们公司多年的行业技术积累，我有理由相信我们可以一个月内开发出一套全功能的视频监控系统“。

不得不承认，我每次听到这种对话的时候，心都痛得哭不出来了。但是作为一个程序员，你还是默默的打开了盗版的VS2010，准备开工。

通过前期了解，你知道每一个视频设备生产商，都会提供相应的设备控制组件，他们通常是Win32形式的，当然也有COM的，但是他们不提供协议，所以你不得不使用PInvoke来完成这项任务。

由于需要支持多种设备，你决定将不同厂家提供了DLL放在主程序下各自的目录，毕竟全部放在一起，实在是个糟糕的主意。你的目录可能是这样的

App.exe

Device

Company1

Company1.dll

Protocol.dll

Company2

Company2.dll

其中Company1.dll为你使用的接口DLL，而Protocol则是Company1.dll内部使用的DLL。

现在你可以进行如下设计：

public interface INVR

{

bool Open();

}

class Company1NVRNative

{

const string DLLNAME = "Device\\Company1\\Company1.dll";

[DllImport(DLLNAME, CallingConvention = CallingConvention.StdCall)]

internal static extern bool OpenDevice([MarshalAs(UnmanagedType.LPStr)] string deviceURL, ref IntPtr hDevice);

}

public class Company1NVR : INVR

{

private IntPtr hDevice;

public string URL { get; set; }

public bool Open()

{

return Company1NVRNative.OpenDevice(this.URL, ref this.hDevice);

}

}

好吧，我想信上面的代码是大多数要都会这样写的。然后主程序启动的时候，会读取所有设备，并创建相应的类型，并使得他们工作。

然而代码真正的会像你预期那样正常工作吗？答案是：不一定。

对的，不一定，这取决于很多因素。但是在这里我会提出一点可能，你代码会出现无法

Company1.dll的异常。那么你一定很好奇，我不是在DLLNAME中指定要加载的DLL吗？难到是路径不对？不，都是对的，DLLNAME指定了正确的路径，而且在加载的Company1.dll的时候，也是可以找到该文件的，那么这又是为什么了？

答案是：系统无法为Company1.dll加载其所需要Protocol.dll。

很奇怪是吧。系统能够，加载所需要的Company1.dll是因为，它根据程序目录和Device\\Company1\\Company1.dll能够，找到该文件。但是Company1.dll引用了Protocol.dll，只有一个名称信息，系统就只能根据前需要节所提到的加载流程进行加载。很显然，在尝试所有的方法后，无法加载Protocol.dll，所以系统认为加载Company1.dll失败，抛出一个加载Company1.dll失败的异常。

好吧，我相信我已经很好的解释了错误的原因。接下来让我们看一下，如何解决该问题：为了解决这个问题，你需要清楚DLL加载流程，我是清楚的所以我可以给出以下几种方法：

1：Company1文件夹路径设置到系统的环境变量中。这样系统根据环境变量就可以找到该DLL。

2：程序在初始化的时候，设置自身的环境变量。这样系统也可以根据环境变量找到该DLL。

附：

程序启动时设置自身环境变量的代码：

class Program

{

static string[] GetSubDirs(string dir)

{

List<string> dirs = new List<string>();

dirs.Add(dir);

foreach (string s in System.IO.Directory.GetDirectories(dir))

{

dirs.Add(s);

dirs.AddRange(GetSubDirs(s));

}

return dirs.ToArray();

}

public static void Main(string[] args)

{

string path = Environment.GetEnvironmentVariable("path");

string deviceDir = System.IO.Path.Combine(new FileInfo(typeof(Program).Assembly.Location).DirectoryName, "Device");

string[] subDirs = GetSubDirs(deviceDir);

string newPath = subDirs.Aggregate((sum, current) => sum + ";" + current) + path;

Environment.SetEnvironmentVariable("path", newPath);

}

}

# 进阶 数据封送处理器

## 结构一致型与非结构一致性

在托管与非托管程序中，语意上一样的数据类型可能具有不同的内存分布形式。

结构一致型：对于那些在托管与非托管代码中所在内存大小，排列序列一致的类型，我们称之为结构一致型。

在.NET中提供的所有基础类型：System.Byte，System.SByte，System.Int16, System.Uint16, System.Int32, System.UInt32, System.Int64, System.UInt64, System.IntPtr, System.UIntPtr, System.Single, System.Double等均为结构一致性。

同时对于以下两种类型也为结构一致型：

任何包含结构一致型的一维数组。但是任何包含结构一致性的一维数组的类型不是结构一致型。

为什么包含结构一致性的一维数组的类是非结构一致性的了？因为在.NET中数组除了包含缓冲区，还包含得有相应的长度等信息。在非托管中，在计算相对数组里的缓冲区偏移地址的时候就是错的。

任何只包含结构一致型类型的类或结构均为结构一致型。

例如

Int 在非托管中也为32位，也内存排列序列一致。

int[] 在数据封送处理中将直接传递数组的缓冲区指针，也为结构一致型.

[StructLayout(LayoutKind.Sequential)]

class BlittableClass

{

public int Age;

public int Height;

}

类BlittableClass由于只包含两个结构一致性类型int所以也为结构一致性类型。

非结构致型：对于那些在托管与非托管代码中所暂用内存大小，排列序列可能不致的我们称之非结构一致型。

在.NET平台中提供的基础中类型中System.Char，System.Boolean，System.String(该类并不是基础类型，但很多时候会在基础类型中进行讨论)均为非结构一致性类型。

System.Char之所为非结构一致型，是其因为其在非托管中可以转换为1,2字节。

System.Boolean之所以为非结构一致型，是其因为其在非托管内存中可以转换为1,2,4字节。

System.String之所以为非结构一致型，是因为其在非托管内存中可以转换为ANSI或者Unicode字符串。

[StructLayout(LayoutKind.Sequential)]

class NonBlittableClass

{

public int Age;

[MarshalAs(UnmanagedType.ByValArray, ArraySubType = UnmanagedType.I4, SizeConst = 20)]

public int[] ChildrenAges;

}

对于类NonBlitableClass来说，它是非结构一致性的。因为它包含一个成员变量ChildrenAge。记住任何包含数组的类或者结构都是非结构一致型的。

## 参数传递的两种方式 复制与固定

对于任何结构一致性的类型，当采用指针作为传入参数时，数据封送处理器将直接固定托管内存，以防止在非托管代码使用内存期间CLR移动此内存。然后将指针传递给到非托管代码的栈上，以便进行使用。此时非托管代码通过指针对数据进行任何的操作都将直接作用到托管内存上。也就是说，参数此时不使用OutAttibute也可以看到更改。

而对于任何非结构一致性的类型，不管是任何值传输，还是指针传输数据封送处理器都会先在非托管内存中开辟固定长度的内存，然后将传入参数进行封装复制，在非托管代码调用完成后，释放申请的内存。如果指定的有OutAttribute则将非托管代码修改后的值复制到托管内存中以实现参数的双向传输。

数据封送处理器Marshaling Service完成了将.NET参数进行处理后传入非托管代码，在函数调用完成后将需要输出的参数进行复制到托管代码中。

如有以下声明：

Native:

extern "C" NATIVE\_API BOOL GetItemHeight(int index,int \*height);

Managed:

[DllImport("Native.dll", EntryPoint = "GetItemHeight", CallingConvention = CallingConvention.Cdecl)]

[return : MarshalAs(UnmanagedType.Bool)]

public static extern bool GetItemHeight(int index, ref int height);

在本例中，数据封送处理器，将参数height的内存地址，index值依次入栈。在调用完成后将C API的返回值BOOL(int)型，转换为一个C#中的bool值。

# 附录 常用类型信息

## DllImportAttribute

该类主要用来声明非托管函数。

|  |  |
| --- | --- |
| 属性或者字段名称 | 作用 |
| Value | 获取包含入口点的 DLL 文件的名称。 |
| BestFitMapping | 将 Unicode 字符转换为 ANSI 字符时，启用或禁用最佳映射行为。 |
| CallingConvention | 指示入口点的调用约定。 |
| CharSet | 指示如何向方法封送字符串参数，并控制名称重整。 |
| EntryPoint | 指示要调用的 DLL 入口点的名称或序号。 |
| ExactSpelling | 控制 DllImportAttribute.CharSet 字段是否使公共语言运行时在非托管 DLL 中搜索入口点名称，而不使用指定的入口点名称。 |
| SetLastError | 指示被调用方在从特性化方法返回之前是否调用 SetLastError Win32 API 函数。 |
| ThrowOnUnmappableChar | 启用或禁用在遇到已被转换为 ANSI“?”字符的无法映射的 Unicode 字符时引发异常。 |

## InAttribute

指示应将数据从调用方封送到被调用方，而不返回到调用方。

## OutAttribute

指示应将数据从被调用方封送回调用方。

## MarshalAsAttribute

指示如何在托管代码和非托管代码之间封送数据。

|  |  |
| --- | --- |
| 属性或者字段名称 | 作用 |
| Value | 获取 UnmanagedType 值，数据将被作为该值封送。 |
| ArraySubType | 指定非托管 UnmanagedType.LPArray 或 UnmanagedType.ByValArray 的元素类型。 |
| IidParameterIndex | 指定 COM 使用的非托管 iid\_is 特性的参数索引。 |
| MarshalCookie | 向自定义封送拆收器提供附加信息。 |
| MarshalType | 指定自定义封送拆收器的完全限定名。 |
| MarshalTypeRef | 将 MarshalAsAttribute.MarshalType 作为类型实现。 |
| SafeArraySubType | 指示 UnmanagedType.SafeArray 的元素类型。 |
| SafeArrayUserDefinedSubType | 指示用户定义的 UnmanagedType.SafeArray 元素类型。 |
| SizeConst | 指示固定长度数组中的元素数，或要导入的字符串中的字符（不是字节）数。 |
| SizeParamIndex | 指示从零开始的参数，该参数包含数组元素的计数，与 COM 中的 size\_is 类似。 |

## UnmanagedFPointerAttribute

控制作为非托管函数指针传入或传出非托管代码的委托签名的封送行为.

|  |  |
| --- | --- |
| 属性或者字段名称 | 作用 |
| CallingConvention | 获取调用约定的值。 |
| BestFitMapping | 将 Unicode 字符转换为 ANSI 字符时，启用或禁用最佳映射行为。 |
| CharSet | 指示如何向方法封送字符串参数。 |
| SetLastError | 指示被调用方在从特性化方法返回之前是否调用 SetLastError Win32 API 函数。 |
| ThrowOnUnmappableChar | 启用或禁用在遇到已被转换为 ANSI“?”字符的无法映射的 Unicode 字符时引发异常。 |
|  |  |

## Marshal

提供了一个方法集，这些方法用于分配非托管内存、复制非托管内存块、将托管类型转换为非托管类型，此外还提供了在与非托管代码交互时使用的其他杂项方法。

由于该类的方法属性过多，在此只列出部分

|  |  |
| --- | --- |
| 方法名称 | 作用 |
| AllocHGlobal(Int32) | 通过使用指定的字节数，从进程的非托管内存中分配内存。 |
| Copy(Byte[], Int32, IntPtr, Int32) | 将一维的托管 8 位无符号整数数组中的数据复制到非托管内存指针。 |
| Copy(IntPtr, Byte[], Int32, Int32) | 将数据从非托管内存指针复制到托管 8 位无符号整数数组。 |
| FreeHGlobal | 释放以前从进程的非托管内存中分配的内存。 |
| GetLastWin32Error | 返回由上一个非托管函数返回的错误代码，该函数是使用设置了 DllImportAttribute.SetLastError 标志的平台调用来调用的。 |
| PtrToStringAnsi(IntPtr) | 将非托管 ANSI 字符串中第一个 null 字符之前的所有字符复制到托管 String，并将每个 ANSI 字符扩展为 Unicode 字符。 |
| PtrToStringUni(IntPtr) | 分配托管 String，并从非托管 Unicode 字符串向其复制第一个空字符之前的所有字符。 |
| PtrToStructure(IntPtr, Object) | 将数据从非托管内存块封送到托管对象。 |
| PtrToStructure(IntPtr, Type) | 将数据从非托管内存块封送到新分配的指定类型的托管对象。 |
| StringToHGlobalUni | 向非托管内存复制托管 String 的内容。 |
| StructureToPtr | 将数据从托管对象封送到非托管内存块。 |

## UnmanagedType枚举

|  |  |
| --- | --- |
| 名称 | 描述 |
| Bool | 4 字节布尔值（true != 0、false = 0）。这是 Win32 BOOL 类型。 |
| I1 | 1 字节带符号整数。可使用此成员将布尔值转换为 1 字节、C 样式的 bool（true = 1、false = 0）。 |
| U1 | 1 字节无符号整数。 |
| I2 | 2 字节带符号整数。 |
| U2 | 2 字节无符号整数。 |
| I4 | 4 字节带符号整数。 |
| U4 | 4 字节无符号整数。 |
| I8 | 8 字节带符号整数。 |
| U8 | 8 字节无符号整数。 |
| R4 | 4 字节浮点数。 |
| R8 | 8 字节浮点数。 |
| Currency | 在 System.Decimal 上使用，以将十进制数值作为 COM 货币类型而不是 Decimal 封送。 |
| BStr | 长度前缀为双字节的 Unicode 字符串。可以在 String 数据类型上使用此成员（它是 COM 中的默认字符串）。 |
| LPStr | 单字节、空终止的 ANSI 字符串。可在 System.String 或 System.Text.StringBuilder 数据类型上使用此成员。 |
| LPWStr | 一个 2 字节、空终止的 Unicode 字符串。  请注意，如果非托管字符串不是使用非托管的 CoTaskMemAlloc 函数创建的，则不能在此非托管字符串中使用 LPWStr 值。 |
| LPTStr | 与平台相关的字符串：在 Windows 98 上为 ANSI，在 Windows NT 和 Windows XP 上为 Unicode。该值仅对平台调用受支持，而对 COM 互操作 则不受支持，原因是不支持导出 LPTStr 类型的字符串。 |
| ByValTStr | 用于在结构中出现的内联定长字符数组。与 ByValTStr 一起使用的字符类型由应用于包含结构的 System.Runtime.InteropServices.StructLayoutAttribute 的 System.Runtime.InteropServices.CharSet 参数确定。应始终使用 MarshalAsAttribute.SizeConst 字段来指示数组的大小。  .NET Framework 的 ByValTStr 类型的行为类似于结构中的 C 样式、固定大小的字符串（例如，char s[5]）。 |
| IUnknown | COMIUnknown 指针。可以在 Object 数据类型上使用此成员。 |
| IDispatch | 一个 COM IDispatch 指针（在 Microsoft Visual Basic 6.0 中为 Object）。 |
| Struct | 一个用于封送托管格式化类和值类型的 VARIANT。 |
| Interface | COM 接口指针。从类元数据获得接口的 Guid。如果将此成员应用于类，则可以使用该成员指定确切的接口类型或默认的接口类型。当应用于 Object 数据类型时，此成员将产生 UnmanagedType.IUnknown 行为。 |
| SafeArray | SafeArray 是自我描述的数组，它带有关联数组数据的类型、秩和界限。可将此成员与 MarshalAsAttribute.SafeArraySubType 字段一起使用，以重写默认元素类型。 |
| ByValArray | 当 MarshalAsAttribute.Value 设置为 ByValArray 时，必须设置 SizeConst 以指示数组中的元素数。当需要区分字符串类型时，ArraySubType 字段可以选择包含数组元素的 UnmanagedType。此 UnmanagedType 只可用于作为结构中的字段的数组。 |
| SysInt | 与平台相关的带符号整数。在 32 位 Windows 上为 4 字节，在 64 位 Windows 上为 8 字节。 |
| SysUInt | 与平台相关的无符号整数。在 32 位 Windows 上为 4 字节，在 64 位 Windows 上为 8 字节。 |
| VBByRefStr | 允许 Visual Basic 2005 在非托管代码中更改字符串，并将结果在托管代码中反映出来。该值仅对平台调用受支持。 |
| TBStr | 一个有长度前缀的与平台相关的 char 字符串。在 Windows 98 上为 ANSI，在 Windows NT 上为 Unicode。很少用到这个类似于 BSTR 的成员。 |
| VariantBool | 2 字节、OLE 定义的 VARIANT\_BOOL 类型（true = -1、false = 0）。 |
| FPtr | 一个可用作 C 样式函数指针的整数。可将此成员用于 Delegate 数据类型或从 Delegate 继承的类型。 |
| AsAny | 一个动态类型，将在运行时确定对象的类型，并将该对象作为所确定的类型进行封送处理。仅对平台调用方法有效。 |
| LPArray | 指向 C 样式数组的第一个元素的指针。当从托管到非托管进行封送处理时，该数组的长度由托管数组的长度确定。当从非托管到托管进行封送处理时，将根据 MarshalAsAttribute.SizeConst 和 MarshalAsAttribute.SizeParamIndex 字段确定该数组的长度，当需要区分字符串类型时，还可以后跟数组中元素的非托管类型。 |
| LPStruct | 一个指针，它指向用于封送托管格式化类的 C 样式结构。仅对平台调用方法有效。 |
| CustomMarshaler | 当与 MarshalAsAttribute.MarshalType 或 MarshalAsAttribute.MarshalTypeRef 一起使用时，指定自定义封送拆收器类。MarshalAsAttribute.MarshalCookie 字段可用于将附加信息传递给自定义封送拆收器。可以在任何引用类型上使用此成员。 |
| Error | 此与 I4 或 U4 关联的本机类型将导致参数作为导出类型库中的 HRESULT 导出。 |

# 附录 托管与非托管的数据类型映射

表4-1 托管与非托管之前的数据类型映射关系

|  |  |  |  |
| --- | --- | --- | --- |
| **Unmanaged type in Wtypes.h** | **Unmanaged C language type** | **Managed class name** | **Description** |
| **HANDLE** | **void\*** | [System.IntPtr](http://msdn.microsoft.com/en-us/library/system.intptr(v=vs.100).aspx) | 32 bits on 32-bit Windows operating systems, 64 bits on 64-bit Windows operating systems. |
| **BYTE** | **unsigned char** | [System.Byte](http://msdn.microsoft.com/en-us/library/system.byte(v=vs.100).aspx) | 8 bits |
| **SHORT** | **short** | [System.Int16](http://msdn.microsoft.com/en-us/library/system.int16(v=vs.100).aspx) | 16 bits |
| **WORD** | **unsigned short** | [System.UInt16](http://msdn.microsoft.com/en-us/library/system.uint16(v=vs.100).aspx) | 16 bits |
| **INT** | **int** | [System.Int32](http://msdn.microsoft.com/en-us/library/system.int32(v=vs.100).aspx) | 32 bits |
| **UINT** | **unsigned int** | [System.UInt32](http://msdn.microsoft.com/en-us/library/system.uint32(v=vs.100).aspx) | 32 bits |
| **LONG** | **long** | [System.Int32](http://msdn.microsoft.com/en-us/library/system.int32(v=vs.100).aspx) | 32 bits |
| **BOOL** | **long** | [System.Int32](http://msdn.microsoft.com/en-us/library/system.sbyte(v=vs.100).aspx) | 32 bits |
| **DWORD** | **unsigned long** | [System.UInt32](http://msdn.microsoft.com/en-us/library/system.uint32(v=vs.100).aspx) | 32 bits |
| **ULONG** | **unsigned long** | [System.UInt32](http://msdn.microsoft.com/en-us/library/system.uint32(v=vs.100).aspx) | 32 bits |
| **CHAR** | **char** | [System.Char](http://msdn.microsoft.com/en-us/library/system.char(v=vs.100).aspx) | Decorate with ANSI. |
| **WCHAR** | **wchar\_t** | [System.Char](http://msdn.microsoft.com/en-us/library/system.char(v=vs.100).aspx) | Decorate with Unicode. |
| **LPSTR** | **char\*** | [System.String](http://msdn.microsoft.com/en-us/library/system.string(v=vs.100).aspx) or [System.Text.StringBuilder](http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.100).aspx) | Decorate with ANSI. |
| **LPCSTR** | **Const char\*** | [System.String](http://msdn.microsoft.com/en-us/library/system.string(v=vs.100).aspx) or [System.Text.StringBuilder](http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.100).aspx) | Decorate with ANSI. |
| **LPWSTR** | **wchar\_t\*** | [System.String](http://msdn.microsoft.com/en-us/library/system.string(v=vs.100).aspx) or [System.Text.StringBuilder](http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.100).aspx) | Decorate with Unicode. |
| **LPCWSTR** | **Const wchar\_t\*** | [System.String](http://msdn.microsoft.com/en-us/library/system.string(v=vs.100).aspx) or [System.Text.StringBuilder](http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=vs.100).aspx) | Decorate with Unicode. |
| **FLOAT** | **Float** | [System.Single](http://msdn.microsoft.com/en-us/library/system.single(v=vs.100).aspx) | 32 bits |
| **DOUBLE** | **Double** | [System.Double](http://msdn.microsoft.com/en-us/library/system.double(v=vs.100).aspx) | 64 bits |

在表4-1中，我们列出了基础的数据类型及其对应的关系。你所可能需要反复查看此表，以记住他们。当然你也可以在需要时进行查看。

# 附录 常见问题

## 为何不能加载我的DLL

将这个问题列在最前面，是因为这是我见过网络上问得最多的问题。如果要仔细的讲述这个问题，我相信没有个几天是说不清楚的。所以在这里我们简单的说明。

系统何时提示不能加载我的DLL？当系统需要使用一个DLL的时候它会尝试加载（请参考第三章 加载非托管DLL流程）。如果说某一个DLL文件找不到或者某一个DLL执行DLLMain函数失败的时候系统就会报该异常。由于第一种原因引起的问题是最常见的，因为我们C程序通常会使用第三方组件，或者会使用VC++的运行库，如果目标计算机没有安装这种程序就会导致程序提示无法加载我们的DLL。

当然有时候有时候你可能有这些DLL，但是你发现还是会失败。这时通常可能是由于程序的环境变量不正确的，导致加载的时候无法找到相应的DLL。事实上一个最偷赖的方式将你所需要DLL复制到System32目录或者Wow\system32目录下或者全部复制到应用程序的根目录下。但这真的是一个很丑陋的方式。

当发生这种问题时候，我也推荐你使用一个工具：Procmon.exe工具，跟踪查看程序的操作细节从而分析程序是在加载那个DLL失败。

## BadImageFormatException

当发生这种错误的时候，系统已经可以正确的定位所需要的DLL，但是该DLL的与程序当前的位数不一样。例如一个C#程序使用了一个32位的DLL但是编译的时候被设置成:anycpu，那么这个程序在32的系统上运行时候不会出现问题，但是当在64位系统中运行时候就会报该异常。因为anycpu的程序由实际运行的系统决定，如果运行64位系统上则进程是64的，但是它尝试加载的DLL是32位的，导致程序执行该异常。

## 调试时提示我设置的回调委托正在被回收

在C#中的委托的实例也存在于托管内存中，所以它也会被CLR进行垃圾回收。

假如你的C代码中需要一个回调函数指针，并且有一个后台线程会不断的调用该回调以通知上层代码。但是你将回调的委托实例声明在函数局部或者类、结构的成员变量则该变量就有可以会被回调。在使用VS调试的时候，它可以检测到这种错误，并进行相应的提示。

对于这个问题一个较好的选择是:像第10章中的那样将委托实例声明类的表态变量，这样可以保证直到应用域被结束该变量一直存在，也就不存在在C代码需要使用它的时候会回收的问题。

当然委托的实例也是可以声明为类的成员，但是你必须保证在C代码需回调该委托的时候，这个委托没有CLR进行垃圾回收。

## 非托管代码调用托管回调函数后，提示栈异常

C代码在声明相应的回调函数时会同时声明其调用约定，是的回调函数也是有调用约定的。在VC++中默认是的cdecl，而大数Win32 API及其回调都是stdcall。所以出现该问题时请确定标记在委托UnmanagedFPointerAttribute中的CallingConvention是否与C代码中声明的调用约定一样。

## 非托管代码在对托管传入的参数进行赋值时出现 内存访问异常

总的来说，这是真的是一个很复杂的问题。我也不告诉你会为什么，但是你可以从以下几个方面来检查代码：

* 是否传入了空指针
* 指针所指向的是否可以更改。
* 如果是类或者结构则结构成员是否对齐，如果没有对齐会导致访问到错误的数据，从而导致该问题。
* 程序内部本身是否有错误。

## 为什么OutAttribute对String无效

在.NET中String对象的实例被认为是不可以更改的。所以在CLR进行PInvoke的时候会忽略掉OutAttribute标记，也就是说如果CLR检测到参数类型是String类型则不会修改后的内存复制到托管内存。