(85条消息) C#调用C++的DLL_c#调用c++dll_追烽少年x的博客-CSDN博客

本文来讲述一下C#中调用DLL的注意事项：
一.导出C++普通函数
新建C++动态连接库控制台程序，代码如下：
EncryptString.h

```
#pragma once
#ifndef _ENCRYPTSTRING_H_
#define _ENCRYPTSTRING_H_

#ifndef ENCRYPT_EXPORTS
#define ENCRYPT_EXPORTS _declspec(dllexport)
#else
#define ENCRYPT_EXPORTS _declspec(dllimport)
#endif // ENCRYPT_EXPORTS

// 方式1:
// 使用导出函数的命令 declspec(dllexport)导出函数
// 使用 extern "C" 告诉编译器按照C风格的形式导出函数
// extern "C"   ENCRYPT_EXPORTS void _stdcall GetMD5(const char* pSrcStr, char* pDesStr);

// 方式2:使用模块化定义文件:*.def 导出函数
void _stdcall GetMD5(const char* pSrcStr, char* pDesStr);

#endif // !_ENCRYPTSTRING_H_
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

实现代码：EncryptString.cpp如下：

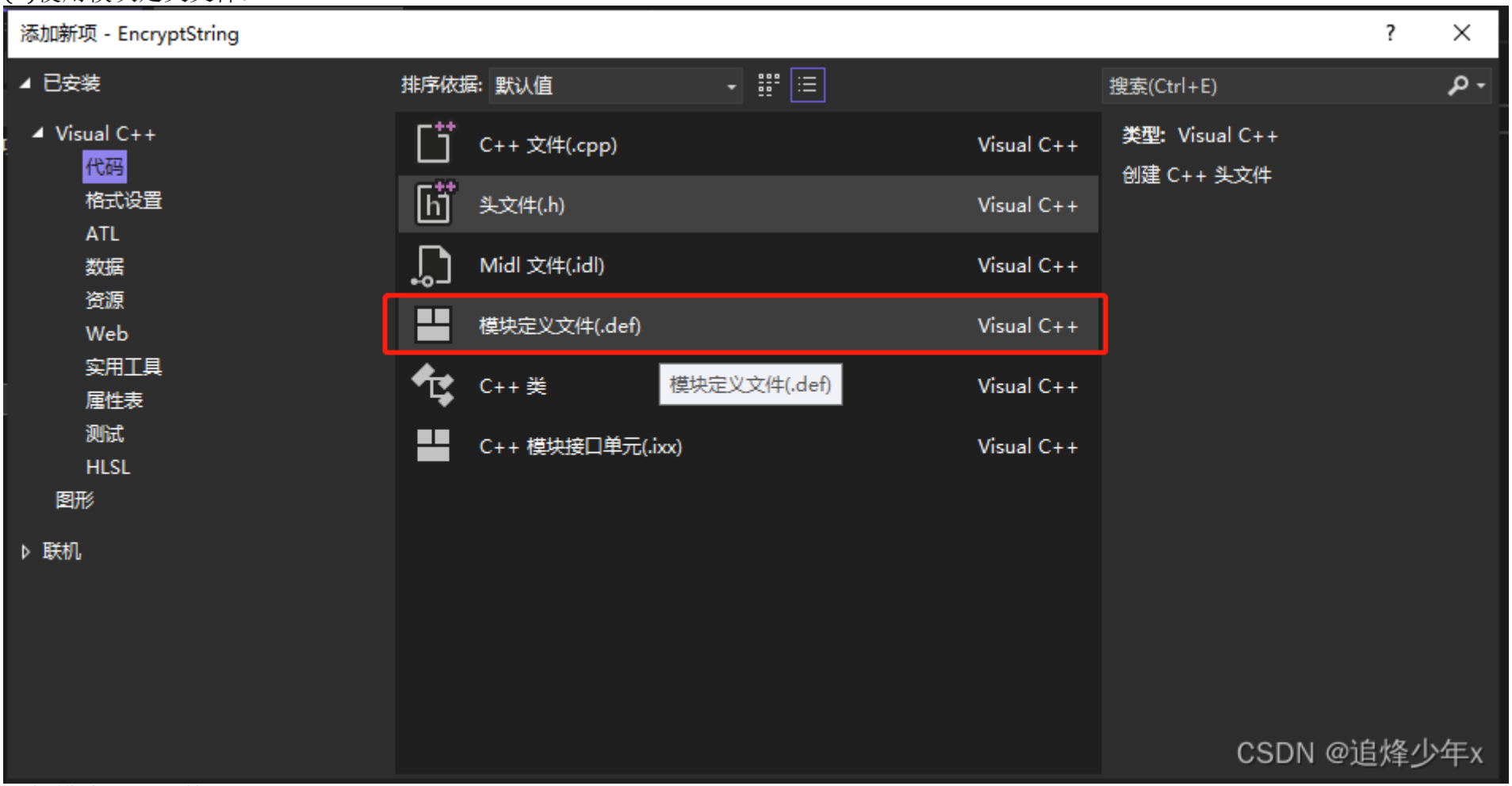
```
#include "pch.h"
#include "EncryptString.h"
#include <assert.h>
#include <cstdio>
#include <iostream>
#include <windows.h>
void _stdcall GetMD5(const char* pSrcStr, char* pDesStr)
{
        assert(pSrcStr != NULL && pDesStr != NULL);
        strcpy(pDesStr, pSrcStr);
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11

在上面的代码中涉及到了如何导出C++函数,使用了两种方式：
(1) 使用_declspec(dllexport)声明函数为导出函数，因为C++支持函数重载，所以使用dumpbin -exports 查看导出函数时发生了

函数重载，编译器在编译代码的过程中会把函数的参数类型也加入到函数命名中，导致导出函数的名称发生了变化，这给代码调用带来了不便。为了避免函数发生重载，比较常用的做法是告诉编译器将函数导出为C风格的形式，因此在函数前面加上extern “C” _declspec(dllexport)。

(2)使用模块定义文件：



添加模块定义文件***.def,定义如下:

```
LIBRARY EncryptString.dll
EXPORTS
GetMD5
```

- 1
- 2

- 3

上面LIBRARY 后面的是导出的DLL的名称，EXPORTS下面的是导出函数的名称。
说明：在函数声明中有_stdcall，表示将函数声明为标准调用，C#默认采用的就是这种方式。所以在声明C++导出函数时最好声明_stdcall。
最后生成了EncryptString.dll。

C#调用EncryptString.dll步骤如下：
新建C#控制台应用程序如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;

namespace EncryptString
{
    internal class Program
    {
        // 1
        // 函数封装为const char*
        //[DllImport("EncryptString.dll",CallingConvention = CallingConvention.StdCall,
        //    CharSet = CharSet.Ansi,EntryPoint = "GetMD5")]
        //public static extern void GetMD5(string strSrc, StringBuilder strDes);

        // 2
        // 当函数输入参数为char*,函数调用将其"退化"为一个指针,读取内容知道\0为止
        // 那么C#封装时,可以通过考虑IntPtr
        [DllImport("EncryptString.dll", CallingConvention = CallingConvention.StdCall,
            CharSet = CharSet.Ansi, EntryPoint = "GetMD5")]
        public static extern void GetMD5(IntPtr strSrc, IntPtr strDes);
        static void Main(string[] args)
        {
            // 1
            //string str = "123456";
            //StringBuilder stringBuilder = new StringBuilder();
            //GetMD5(str, stringBuilder);
            //Console.WriteLine(stringBuilder.ToString());

            // 2
            string str = "123456";
            // 将托管区string复制到非托管区(ANSI编码)
```

```
            IntPtr pStrSrc = Marshal.StringToHGlobalAnsi(str);
            // 在非托管区动态分配内存
            IntPtr pStrDes = Marshal.AllocHGlobal(128);
            // 写入0
            Marshal.WriteByte(pStrDes,0);

            GetMD5(pStrSrc,pStrDes);
            // 获取字符串(将非托管区内存复制到托管区并赋值给string)
            string strDes = Marshal.PtrToStringAnsi(pStrDes);
            Console.WriteLine(strDes);
            // 释放非托管区内存
            Marshal.FreeHGlobal(pStrSrc);
            Marshal.FreeHGlobal(pStrDes);
            Console.ReadKey();
        }
    }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21

- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51

上述代码中，导入外部DLL需要声明using System.Runtime.InteropServices;具体作用描述可以参考微软官方文档：System.Runtime.InteropServices描述
方式1说明： 使用DllImport[]来导入函数,指定编码方式为ansi（win32 C++中char*对应的编码方式是ansi*），*C++中char*与C#中的string对应。这里有一条原则：*char 作为C++函数的参数在函数内不发生变化，比如声明为const char* 在C#中对应string类型。如果char* 本身作为返回值，即会在函数内部发生变化，则对应C#的StringBulider 类型。**

方式2说明：当函数输入参数为字符串char* 时，调用函数将其退化为一个指针，读取内容知道\0为止，因此可以考虑通过C#的IntPtr来封装函数。在实际代码中，更加推荐使用这种方式，具体代码如上。

二，导出C++结构体
C++中的结构体是复杂的类型，比较难以处理。下面讲一下如何处理在C#中处理C++结构体.
新建C++动态链接库工程StructDLL如下：
StrcutDLL.h

```
#pragma once
#define DLL_API extern "C" _declspec(dllexport)
#pragma pack(1)
typedef struct
{
        char name[64];
        int age;
        bool male;
        char address[128];
}PERSON;
#pragma pack()

DLL_API char* _stdcall GetName(PERSON* pInfo);
DLL_API int _stdcall GetAge(PERSON* pInfo);
DLL_API bool _stdcall GetMale(PERSON* pInfo);
DLL_API char* _stdcall GetAddress(PERSON* pInfo);
DLL_API void _stdcall ClonePerson(PERSON* pSrcInfo, PERSON* pDesInfo);
```

- 15
- 16
- 17

具体实现代码如下：StructDLL.cpp

```
include"pch.h"
#include "StructDLL.h"
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* _stdcall GetName(PERSON* pInfo)
{
        return pInfo->name;
}
int _stdcall GetAge(PERSON* pInfo)
{
        return pInfo->age;
}
bool _stdcall GetMale(PERSON* pInfo)
{
        return pInfo->male;
}
char* _stdcall GetAddress(PERSON* pInfo)
{
        return pInfo->address;
}
void _stdcall ClonePerson(PERSON* pSrcInfo, PERSON* pDesInfo)
{
        assert(pSrcInfo!=NULL && pDesInfo!= NULL);
        sprintf_s(pDesInfo->address, pSrcInfo->address,128);
        sprintf_s(pDesInfo->name, pSrcInfo->name, 64);
        pDesInfo->age = pSrcInfo->age;
        pDesInfo->male = pSrcInfo->male;
}
```

- 1
- 2
- 3
- 4
- 5

- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31

说明上面的每一个函数的函数参数都是一个结构体。
下面来看C#如何封装，新建C#的工程代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;

namespace StrcutDLL
```

```
{
    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi, Pack = 1)]
    public struct Person
    {
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 64)]
        public string strName;
        public int nAge;
        public byte bMale;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
        public string strAddress;
    }

    internal class Program
    {
        // 方式1,使用ref的方式封装接口
        //[DllImport("StructDLL.dll", EntryPoint = "GetName",
        //CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        //public static extern IntPtr GetName( Person pInfo);

        //[DllImport("StructDLL.dll", EntryPoint = "GetAge",
        //CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        //public static extern int GetAge(ref Person pInfo);

        //[DllImport("StructDLL.dll", EntryPoint = "GetMale",
        //CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        //public static extern byte GetMale(ref Person pInfo);

        //[DllImport("StructDLL.dll", EntryPoint = "GetAddress",
        //CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        //public static extern IntPtr GetAddress(ref Person pInfo);

        //[DllImport("StructDLL.dll", EntryPoint = "ClonePerson",
        //CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        //public static extern IntPtr ClonePerson(ref Person pSrcInfo, ref Person pDesInfo);


        // 方式2: 建议当调用结构体类型的变量时,采用IntPtr来处理
        // 全部使用IntPtr来封装
        [DllImport("StructDLL.dll", EntryPoint = "GetName",
       CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern IntPtr GetName( IntPtr pInfo);

        [DllImport("StructDLL.dll", EntryPoint = "GetAge",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern int GetAge(IntPtr pInfo);
```

```
[DllImport("StructDLL.dll", EntryPoint = "GetMale",
CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
public static extern byte GetMale(IntPtr pInfo);

[DllImport("StructDLL.dll", EntryPoint = "GetAddress",
CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
public static extern IntPtr GetAddress(IntPtr pInfo);

[DllImport("StructDLL.dll", EntryPoint = "ClonePerson",
CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
public static extern IntPtr ClonePerson(IntPtr pSrcInfo, IntPtr pDesInfo);

static void Main(string[] args)
{
    Person p1 = new Person
    {
        strName = "Kikay",
        nAge=18,
        bMale=0,
        strAddress="China"
    };
    Person p2 = new Person();

    // 在非托管区中分配内存,并复制结构体给该内存
    IntPtr pP1 = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Person)));
    Marshal.WriteByte(pP1, 0);
    Marshal.StructureToPtr(p1, pP1, true);

    IntPtr pP2 = Marshal.AllocHGlobal(Marshal.SizeOf(typeof(Person)));
    Marshal.WriteByte(pP2, 0);
    Marshal.StructureToPtr(p2, pP2, true);

    IntPtr pName = GetName(pP1);
    // 将非托管区内存复制到托管区,并且赋值给strName;
    string strName = Marshal.PtrToStringAnsi(pName);
    int nAge = GetAge(pP1);
    byte bMale = GetMale(pP1);

    IntPtr pAddress = GetAddress(pP1);
    string strAddress = Marshal.PtrToStringAnsi(pAddress);
    Console.WriteLine("{0},{1},{2},{3}", strName,nAge, bMale, strAddress);

    ClonePerson(pP1, pP2);
    // 将非托管区的内存复制到托管区,并且转换为结构体
```

```
            Person p3 = (Person)Marshal.PtrToStructure(pP2,typeof(Person));

            Marshal.FreeHGlobal(pP1);
            Marshal.FreeHGlobal(pP2);

            Console.ReadKey();
        }
    }
}
```

- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64

- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97
- 98
- 99
- 100

- 101
- 102
- 103
- 104
- 105
- 106
- 107

在上述代码中，首先声明StrcutLayout属性控制结构体成员的布局，大家可以参考该说明：StrcutLayout说明
接下来声明结构体，其中的MarshalAs属性说明：指示如何在托管代码和非托管代码之间封送数据。更详细的说明可以参考微软的文档MarshalAs说明或者MarshalAs的简单总结
接下来导入DLL中函数，结构体是以传值方式传递，类才是以传地址方式传递。所以考虑使用ref 的方式实现结构体的封装。如上述代码中的方式1，
方式1说明：上述代码最终只有GetAge()和GetMale()两个函数可以调用成功。具体原因可以参考该文章C#调用Win32C++那些事
本文也装载自该文章。
上述代码的情况是如果以ref 来传递结构体指针,对于字符串这样的字段，可能会出现乱码等异常情况。干脆全部换成IntPtr算了。如方式2代码所示：
建议：**当调用结构体类型的变量时，采用IntPtr的方式来处理。**

三.导出C++类
C++类是比结构体还要复杂的自定义类型，下面讲述如何导出C++类。
新建工程如下：
MyMath.h

```
#pragma once
#ifndef  _MYMATH_H_
#define  _MYMATH_H_

template<class T>
class MyMath
{
public:
        MyMath();
        ~MyMath();
        // 加法
        int Add(const T& a,const T& b);
        // 减法
        int Substract(const T& a, const T& b);
        // 排序
        void Sort(T* pArr,const int& nSize);
        // 输出运算结果
```

```
        char* ToString();
private:
        char m_Info[32];
};

#define TEMPLATE_DLL
#include "MyMath.cpp"

#endif // ! _MYMATH_H_
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26

模板类的具体的实现方法:

MyMath.cpp

```
#include"pch.h"
#ifdef TEMPLATE_DLL
#include"MyMath.h"
#include<iostream>
#include<vector>
#include<cstdio>
#include<cstring>
#include<algorithm>

template<class T>
class MyAscCompare
{
public:
        bool operator()(T& t1, T& t2)
        {
                return t1 < t2;
        }
};

template<class T>
MyMath<T>::MyMath()
{
        memset(m_Info,0,sizeof(m_Info));
}

template<class T>
MyMath<T>::~MyMath()
{

}

// 加法
template<class T>
int MyMath<T>::Add(const T& a, const T& b)
{
        memset(m_Info, 0, sizeof(m_Info));
        sprintf_s(m_Info,"加法运算",32);
        return a + b;
}
// 减法
template<class T>
int MyMath<T>::Substract(const T& a, const T& b)
{
```

```
        memset(m_Info, 0, sizeof(m_Info));
        sprintf_s(m_Info, "减法运算", 32);
        return a - b;
}
// 排序
template<class T>
void MyMath<T>::Sort(T* pArr, const int& nSize)
{
        typename std::vector<T> v;
        for (int i = 0; i < nSize; i++)
        {
                v.push_back(pArr[i]);
        }
        std::sort(v.begin(),v.end(), MyAscCompare<T>());
        for (int i = 0; i < nSize; i++)
        {
                pArr[i] = v[i];
        }
        memset(m_Info, 0, sizeof(m_Info));
        sprintf_s(m_Info, "升序排序运算", 32);
}
// 输出运算结果
template<class T>
char* MyMath<T>::ToString()
{
        return m_Info;
}
#endif
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13

- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49

- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71

程序说明：上面的程序定义了一个模板类：MyMath。
下面我们使用这个模板类,添加头文件如下：
ClassDLL.h

```
#pragma once
#ifndef CLASSDLL_H_
#define CLASSDLL_H_
#include "MyMath.h"
#define DLL_API extern "C" __declspec(dllexport)
DLL_API MyMath<int>* _stdcall InitMyMath();
// 加法
DLL_API int _stdcall Add(MyMath<int>* pMath, const int& a, const int& b);
// 减法
DLL_API int _stdcall Substract(MyMath<int>* pMath, const int& a, const int& b);
// 数组升序
DLL_API void _stdcall SortArray(MyMath<int>* pMath, int* pArr, const int& nSize);
```

```
// 输出
DLL_API char* _stdcall ToString(MyMath<int>* pMath);
// 释放
DLL_API void _stdcall CloseMath(MyMath<int>* pMath);
#endif // CLASSDLL_H_
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17

具体的实现代码如下：
ClassDLL.cpp

```
#include"pch.h"
#include <limits.h>
#include"ClassDLL.h"
MyMath<int>* _stdcall InitMyMath()
{
        MyMath<int>* pMath = new MyMath<int>;
        return pMath;
}
int _stdcall Add(MyMath<int>* pMath, const int& a, const int& b)
{
        if (pMath != NULL)
        {
                return pMath->Add(a,b);
```

```
        }
        return INT_MIN;
}
int _stdcall Substract(MyMath<int>* pMath, const int& a,const int& b)
{
        if (pMath != NULL)
        {
                return pMath->Substract(a,b);
        }
        return INT_MIN;
}
void _stdcall SortArray(MyMath<int>* pMath, int* pArr, const int& nSize)
{
        if (pMath != NULL)
        {
                return pMath->Sort(pArr, nSize);
        }
}
char* _stdcall ToString(MyMath<int>* pMath)
{
        if (pMath != NULL)
        {
                return pMath->ToString();
        }
}
void _stdcall CloseMath(MyMath<int>* pMath)
{
        if (pMath != NULL)
        {
                delete pMath;
                pMath = NULL;
        }
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45

- 46

下面我们在C#工程中调用该MyMath.DLL.
新建C#工程如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.InteropServices;

namespace ClassDLL
{
    internal class Program
    {
        [DllImport("MyMath.dll", EntryPoint = "InitMyMath",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern IntPtr InitMyMath();

        [DllImport("MyMath.dll", EntryPoint = "Add",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern int Add(IntPtr ptr,ref int a,ref int b );

        [DllImport("MyMath.dll", EntryPoint = "Substract",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern int Substract(IntPtr ptr, ref int a, ref int b);
        [DllImport("MyMath.dll", EntryPoint = "SortArray",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern IntPtr SortArray(IntPtr ptr, IntPtr pArr,ref int nSize);

        [DllImport("MyMath.dll", EntryPoint = "ToString",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern IntPtr ToString(IntPtr ptr);

        [DllImport("MyMath.dll", EntryPoint = "CloseMath",
        CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]
        public static extern void CloseMath(IntPtr ptr);

        static void Main(string[] args)
        {
            IntPtr ptr = InitMyMath();

            int a = 1;
```

```
            int b = 101;
            IntPtr pRes;
            // 加法运算
            int nAdd = Add(ptr,ref a,ref b);
            pRes = ToString(ptr);
            Console.WriteLine(Marshal.PtrToStringAnsi(pRes) + "(,结果" + nAdd.ToString() + ")" );
            // 减法运算
            int nSub = Substract(ptr, ref a, ref b);
            pRes = ToString(ptr);
            Console.WriteLine(Marshal.PtrToStringAnsi(pRes) + "(,结果" + nSub.ToString() + ")");
            // 排序
            int[] arr = {0,2,3,6,4,1,9,7,8,5};
            int nLen = arr.Length;
            int nSize = Marshal.SizeOf(arr[0]) * nLen;
            IntPtr pArr = Marshal.AllocHGlobal(nSize);
            Marshal.Copy(arr,0, pArr, nLen);
            SortArray(ptr, pArr,ref nLen);

            // 还原数组
            int[] nSorted = new int[nLen];
            Marshal.Copy(pArr, nSorted,0,nLen);
            StringBuilder strSorted = new StringBuilder();
            for (int i = 0; i < nLen; i++)
            {
                strSorted.Append(nSorted[i].ToString());
                if (i != nLen - 1)
                {
                    strSorted.Append(",");
                }
            }
            pRes = ToString(ptr);
            Console.WriteLine(Marshal.PtrToStringAnsi(pRes) + "(,结果" + strSorted.ToString() + ")");
            Marshal.FreeHGlobal(pArr);

            CloseMath(ptr);

            Console.ReadKey();
        }
    }
}
```

- 1
- 2
- 3

- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39

- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75

- 76
- 77
- 78
- 79

上面的代码中使用了Marshal类，该类的主要作用是：提供了一个集合，这些方法用于分配非托管内存，复制非托管内存块，将托管类型转化为非托管类型。同时还提供了在与非托管代码交互时的其他杂项方法。更详细的细节可以参考微软官方文档
Marshal类的说明
上述工程的代码，可以从这里下载：Gitee