# 不安全代码

# C#中可以使用指针

.NET提供垃圾回收机制，已经把指针封装了，我们一般不使用指针。但是在某些特殊情况下，我们仍旧可以在C#代码中使用指针。

# 为什么要使用

1. 实时场景，通过指针来提高性能
2. 调用C++ dll的API

# 指针可以操作的区域

通过指针，既可以操作托管区，又可以操作非托管区。

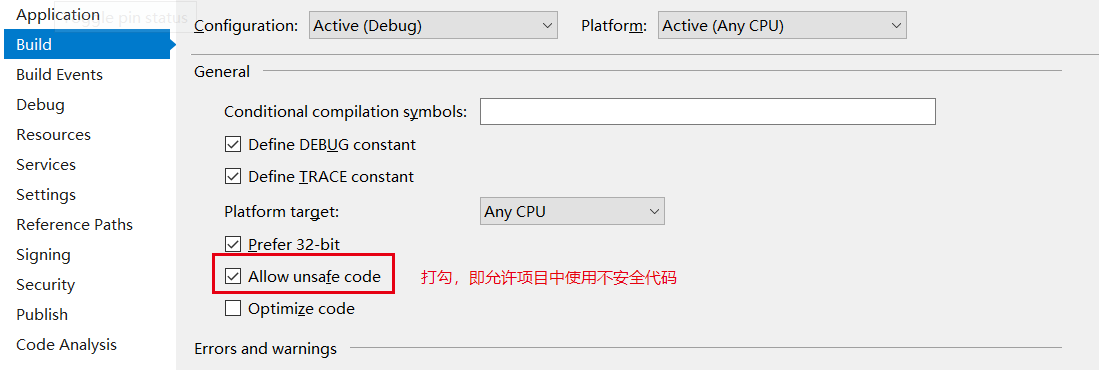
1. 在非托管堆分配内存，该区域不受GC控制，用完需要自己主动释放，否则内存泄漏。
2. 将在托管区的实例固定住，然后通过指针操作该对象所占的非托管内存。用完不要通过指针释放该区域，因为该区域位于托管堆，归GC管理。但是我们应该及时释放我们对实例的固定，因为不释放固定，该区域始终无法被垃圾回收。

# 不建议使用非安全代码

C#项目中，微软并不鼓励我们使用，因为CLR为我们提供了很多，但是指针对内存的操作，CLR无法保证感知的，通过指针我们可以破坏CLR管理的数据。

* 实时应用，采用指针来提高性能；
* 引用非.net DLL提供的如C＋＋编写的外部函数，需要指针来传递该函数；

但是有些特殊场景，我们只能通过指针直接操作内存，来提高程序的性能，那么需要打开项目的不安全模式，允许项目使用指针。打开方法如下图，打开后，我们的项目才能编译成功。



# unsafe用法

出现不安全代码的片段，必须被unsafe关键字包围。

unsafe修饰方法，那么方法的所有代码片段都被包围，我们可以在方法中随便写不安全代码。

public unsafe static int Add(int a, int\* b)

{

  return a + \*b;

}

unsafe修饰代码片段，那被修饰的代码片段允许出现不安全代码。

static void Main(string[] args)

{

   unsafe

   {

      int res, a;

      int\* b;

      a = 1;

      b = &a;

      res = Add(a,b);

    }

}

unsafe还可以标记类、结构，但不能标记局部变量。

# fixed

sizeof只能用于不安全代码块，且不能对类使用。

不能创建指向类的指针，这是因为垃圾收集器不包含指针的任何信息，只包含引用的信息，因此创建指向类的指针会使垃圾收集器不能正常工作。

int\* pX = &(myObject.X);这样的表达式是错误的，因为在垃圾收集的过程中，垃圾收集器会把myClass移动到内存的一个新单元，这样，pX会指向错误的存储单元。解决方法是使用fixed关键字。fixed(int\* pX=&(myObject.X))(//do something}。这样，垃圾收集器知道，在执行fixed块的代码是，不能移动myObject对象。

pa的地址也是被固定了的，所以不能对他进行赋值操作！

包含下面三个指针操作符之一就需要使用Unsafe  
关键字：

 \*

 &

 ->

为了减少内存碎片C#的自动垃圾回收机制会允许已经分配的内存在运行时进行位置调整，所以如果我们多次调用的话就可能  
导致指针指向其他的变量。比如\*pInt为指向一个变量的地址为1001，CLR在重新内存整理分配后该变量就存储在地址为5001的地方。而原来1001的地方可能会  
被分配其他变量，要解决这个问题我们就需要使用Fixed关键字。

fixed 语句禁止垃圾回收器重定位可移动的变量。

unsafe在C#程序中的使用场合：

* fixed其实是在非安全代码下用到才会有意义的。因为他负责对指针所指向的“动态分配内存的对象或对象中的成员”简称“对象”进行锁定（之所以要锁 定，是因为即使在非安全代码模式下，这些对象一样的使用CLR的代码托管，这样的话，很可能造成对象的地址变动，就是因为CLR对资源重新分配的不确定 性，假使指针在未完成对对象的操作时，对象地址变动，那么指针指向的地址就会出现混乱，很可能造成内存泄漏甚至系统崩溃。如果使用fixed进行锁定，那 么只有在执行完fixed模块后，指针所指向的对象才能被移动）。
* a) 垃圾回收机制，维护引用信息不维护指针信息
* b) 引用类型的实例化对象在生存期内由垃圾回收机制处理，可能移动内存
* c) 当一个类的实例化对象中含有值类型时，定义指向这些值类型的指针编译报error，因为这些内嵌在引用类型中的值类型实例会随着引用实例化对象的内存移动而移动，所以指针值在不知情的情况下会发生变化，fixed关键字做的工作便是让这样的类实例化对象（不是类类型本身，而是一个实例化对象）
* example 2：
* 总的来说.net 调用c++方式就几种，要么传地址指针要么传对应结构类型
* 在传递那种方式都是要分配足够大小的空间(有些需要等大小的空间）托管到非托管代码中调用，或者直接再托管内存中分配对应大小类型直接传递。
* 在传递结构体的时候有自带格式，其中Sequential是顺序排列，还有个精确位置排列适用于共联体；pack值是指：类或结构的数据字段在内存中的对齐方式。用默认的不填写则按该结构体内最大字节数作为排列空间的参考大小（一般用默认即可）。
* [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi, Pack = 0)]
* 在传递共联体的时候就用到[StructLayoutAttribute(LayoutKind.Explicit)]的Explicit精确定位，当然要配套使用 [FieldOffsetAttribute(0)]这个属性，用来存放数据在内存中的相对位置。
* ————————————————
* 版权声明：本文为CSDN博主「Theflybird」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。
* 原文链接：https://blog.csdn.net/theflybird/java/article/details/51943567

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | using System;  using System.Collections.Generic;  using System.Text;  using System.Runtime.InteropServices;  namespace ConsoleDll  {　　　//例子1 |

* unsafe void SquarePtrParam(int\* p)  
  　　　　{
* \*p \*= \*p;
* }
* public unsafe void run()
* {
* int[] arr = new int[] { 1, 2, 3, 4, 5 };
* fixed (int\* p = &arr[0]) //此处将锁住arr[0]，使得在fixed操作块内，arr[0]不会被CLR移动
* {
* for (int i = 0, n = arr.Length; i < n; i++)
* {
* SquarePtrParam(p + i);
* Console.WriteLine(p[i]);
* }
* }
* }

|  |  |
| --- | --- |
| 1 | //例子2 |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public static int Add(int[] a,int b)          {              unsafe              {                  fixed (int\* pa = a)//此处将锁住a，使得在fixed操作块内，a不会被CLR移动                  {                      return \*pa + b;                  }              }            }          static void Main(string[] args)          {              int[] a = new int[1];              a[0] = 1;              int b = 2;              int res = Add(a,b);          }      }  } |

* **上述代码需注意的是：pa的地址也是被固定了的，所以不能对他进行赋值操作。**

### ****二、C#可以定义为指针的类型有****

sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool，  
struct(结构体)，结构体中只能包括非托管类型。

### ****三、C#指针操作符****

|  |  |
| --- | --- |
| 操作符 | 说明 |
| \* | 取值运算符 |
| & | 取址运算符 |
| -> | 通过指针处理结构体中的数据（获取或赋值） |
| ++与– | 指针增、减操作 |
| fixed | 用户暂时固定托管代码中引用类型的位置。 |
| Stackallc | 分配内存 |

MSDN对fixed的解释是：**fixed 语句设置指向托管变量的指针，并在执行该语句期间"固定"此变量。**这样就可以防止变量的重定位。

unsafe

{

    int[] array = new int[10];

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

    {

        array[i] = i;

    }

    fixed (int\* p = array)

    {

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

        {

            System.Console.WriteLine(p[i]);

        }

    }

}

fixed (int\* p = array)中，p的定义不能写在别处，而且fixed关键字也只能在unsafe区域里使用。

**：指针指向的内存一定要固定。凡是C#里的引用类型(一切类型的数组都是引用类型)都是分配在托管堆上的，都不固定。有两种方法强制固定，一种是用stackalloc分配在栈上，另一种是用fixed分配在堆上。**

C#指针类型不能是引用类型，不能是含有引用类型的自定义类型或复合类型，只能是预定义类型

    string\* pMessage    //错误C#指针定义

    struct ServiceStatus  
    {  
      int State;  
      string Description;  // 引用类型

    }

    ServiceStatus\* pStatus   //错误指针定义

3. C#中指针赋值

    在C#里，引用类型也叫Movable类型，由于垃圾回收器和内存布局优化，在堆上分配的内存是不固定的，会被挪动。所不

    能将堆上分配的地址赋值给C#指针， 如果能将堆上地址固定，就没有问题了，那么如何固定引用类型？C#中有一个关键字

    fixed可以用在此处。

    byte\* pData = &bytes[0];  // 编译错误，数组在C#里是引用类型，属于movable

    byte[] bytes = new byte[24];  
    fixed (byte\* pData = &bytes)   //正确，使用fixed固定  
    {  
       // ...  
    }

    当然，使用fixed不是没有代价的，它会使内存出现碎片。

# 封装一级指针

C++的参数如果是一维指针，如double\* tag，double tag[] (Struct tag[] 或 Struct\* tag). 那么C#的封装参数形式有三种：ref double tag， out double tag，double[ ] tag. 如果tag是一个传入参数变量，供C++函数读取使用，那么就用ref double tag。如果tag是一个传出参数变量，供C++函数写入传出结果，那么就用out double tag。如果tag是个数组，供C++写入或读取，则采用tag[ ]. 针对数组参数，C++内部不分配任何内存，只是拿到外部分配好的内存的指针，读写操纵内存。C#分配的内存大小最好不小于C++操纵的范围。

DMC\_API short \_\_stdcall dmc\_get\_release\_version(WORD CardNo,char \*ReleaseVersion);//读取发布版本号

[DllImport("LTDMC.dll", EntryPoint = "dmc\_get\_release\_version", CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]

public static extern short dmc\_get\_release\_version(ushort ConnectNo, byte[] ReleaseVersion);//读取发布版本号

DMC\_API short \_\_stdcall nmc\_download\_configfile(WORD CardNo,WORD PortNum,const char \*FileName);//总线ENI配置文件

[DllImport("LTDMC.dll", EntryPoint = "nmc\_download\_configfile", CharSet = CharSet.Ansi, CallingConvention = CallingConvention.StdCall)]

public static extern short nmc\_download\_configfile(UInt16 CardNo, UInt16 PortNum, String FileName);//总线ENI配置文件

# 拿到一个C#实例的Intptr

public sealed class PinnedObject : IDisposable

{

private GCHandle \_Handle;

private bool \_disposed;

public IntPtr Pointer

{

get { return \_Handle.AddrOfPinnedObject(); }

}

public PinnedObject(object target)

{

\_Handle = GCHandle.Alloc(target, GCHandleType.Pinned);

}

public void Dispose()

{

CleanUp(true);

}

private void CleanUp(bool disposing)

{

if (!\_disposed)

{

\_disposed = true;

\_Handle.Free();

\_Handle = new GCHandle();

if(disposing)

GC.SuppressFinalize(this);

}

}

~PinnedObject()

{

CleanUp(false);

}

}

**typedef** struct **{**

INT majorNumber**;**

INT minorNumber**;**

INT revisionNumber**;**

INT buildNumber**;**

**}** CL3IF\_VERSION\_INFO**;**

[StructLayout(LayoutKind.Sequential)]

public struct CL3IF\_VERSION\_INFO

{

public int majorNumber;

public int minorNumber;

public int revisionNumber;

public int buildNumber;

};

**typedef** struct **{**

BYTE ipAddress**[**4**];**

WORD portNo**;**

BYTE reserved**[**2**];**

**}** CL3IF\_ETHERNET\_SETTING**;**

[StructLayout(LayoutKind.Sequential)]

public struct CL3IF\_ETHERNET\_SETTING

{

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

public byte[] ipAddress;

public ushort portNo;

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]

public byte[] reserved;

};

枚举

**typedef** enum **{**

CL3IF\_DEVICETYPE\_INVALID **=** 0x0000**,**

CL3IF\_DEVICETYPE\_CONTROLLER **=** 0x0001**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT1 **=** 0x0011**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT2 **=** 0x0012**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT3 **=** 0x0013**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT4 **=** 0x0014**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT5 **=** 0x0015**,**

CL3IF\_DEVICETYPE\_OPTICALUNIT6 **=** 0x0016**,**

CL3IF\_DEVICETYPE\_EXUNIT1 **=** 0x0041**,** // Extensional unit 1

CL3IF\_DEVICETYPE\_EXUNIT2 **=** 0x0042 // Extensional unit 2

**}** CL3IF\_DEVICETYPE**;**

public enum CL3IF\_DEVICETYPE

{

CL3IF\_DEVICETYPE\_INVALID = 0x0000,

CL3IF\_DEVICETYPE\_CONTROLLER = 0x0001,

CL3IF\_DEVICETYPE\_OPTICALUNIT1 = 0x0011,

CL3IF\_DEVICETYPE\_OPTICALUNIT2 = 0x0012,

CL3IF\_DEVICETYPE\_OPTICALUNIT3 = 0x0013,

CL3IF\_DEVICETYPE\_OPTICALUNIT4 = 0x0014,

CL3IF\_DEVICETYPE\_OPTICALUNIT5 = 0x0015,

CL3IF\_DEVICETYPE\_OPTICALUNIT6 = 0x0016,

CL3IF\_DEVICETYPE\_EXUNIT1 = 0x0041, // Extensional Unit 1

CL3IF\_DEVICETYPE\_EXUNIT2 = 0x0042 // Extensional Unit 2

};

# 验证

1. 基本类型，如，char， int，double，ushort，byte等，可处理到二级指针，如int, int\*, int\*\*
2. 结构体，可处理到一级指针，如struct\*
3. 结构体内可含有一级指针，二级指针暂时处理不了，结构体内可含有结构体。

# 示例一

## C++结构体

typedef struct {

BYTE ipAddress[4];

WORD portNo;

BYTE reserved[2];

} CL3IF\_ETHERNET\_SETTING;

## C#结构体

[StructLayout(LayoutKind.Sequential)]

public struct CL3IF\_ETHERNET\_SETTING

{

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

public byte[] ipAddress;

public ushort portNo;

[MarshalAs(UnmanagedType.ByValArray, SizeConst = 2)]

public byte[] reserved;

};

## C++函数

CL3\_IF\_API LONG WINAPI CL3IF\_OpenEthernetCommunication(LONG deviceId, CL3IF\_ETHERNET\_SETTING\* ethernetSetting, DWORD timeout);

## C#函数

[DllImport(DllName)]

internal static extern int CL3IF\_OpenEthernetCommunication(int deviceId, ref CL3IF\_ETHERNET\_SETTING ethernetSetting, uint timeout);

## 传参过程

CL3IF\_ETHERNET\_SETTING ethernetSetting = new CL3IF\_ETHERNET\_SETTING();

ethernetSetting.ipAddress = new byte[4];

ethernetSetting.reserved = new byte[2];

int returnCode = CLP030.CL3IF\_OpenEthernetCommunication(0, ref ethernetSetting, 2000);

总结：用C++ dll封装API时候，如果是数组，就写 int[]形式，如果是一个变量的指针，就写 int \*

用C#封装时，如果看到int[]，就写int[],传进去一个数组，如果看到int \*，就创建一个int变量，写 ref int 形式。

## 总结

1. C#的封装形式和C++形式相同，但是C++的数组不需要new就已经有了内存，而C#需要new出内存，为结构体内的数组一一分配内存，才能传入参数。
2. 基元类型不需要分配内存，因为C#的基元类型是结构体，内存分配在栈上。
3. C++的参数如果是一维指针，如double\* tag，double tag[] (Struct tag[] 或 Struct\* tag). 那么C#的封装参数形式有三种：ref double tag， out double tag，double[ ] tag. 如果tag是一个传入参数变量，供C++函数读取使用，那么就用ref double tag。如果tag是一个传出参数变量，供C++函数写入传出结果，那么就用out double tag。如果tag是个数组，供C++写入或读取，则采用tag[ ]. 针对数组参数，C++内部不分配任何内存，只是拿到外部分配好的内存的指针，读写操纵内存。C#分配的内存大小最好不小于C++操纵的范围。
4. C++结构体中含有数组，一般是int[2] arr, C#结构体中是int[ ] arr, SizeConst = 2，数组的大小放到外面了。
5. 往封装后的C#函数中传递参数的时候，要保证每一个变量都有了内存，而不是只有个引用，没有对照的堆中内存。

# 示例二

[StructLayout(LayoutKind.Sequential)]

public struct CL3IF\_VERSION\_INFO

{

public int majorNumber;

public int minorNumber;

public int revisionNumber;

public int buildNumber;

};

# 示例三

结构体中含有结构体变量和结构体数组。

[StructLayout(LayoutKind.Sequential)]

public struct CL3IF\_MEASUREMENT\_DATA

{

public CL3IF\_ADD\_INFO addInfo;

public CL3IF\_OUTMEASUREMENT\_DATA[] outMeasurementData;

};

## C++函数

Void Fun(my\_struct \* ms);