**如何在 C# 中使用 数据注解**

数据注解 是一种可以应用到 类 或者 [类成员](https://so.csdn.net/so/search?q=%E7%B1%BB%E6%88%90%E5%91%98&spm=1001.2101.3001.7020)上用来指定类之间关系的一种 Attribute，它的应用场景比较多，可用来描述 UI 上如何进行数据展示，还可以用来做类属性的规则验证，这篇文章就来讨论为什么 注解 值得你去学习，以及如何在 .NET Core 中使用。

**使用 System.ComponentModel.DataAnnotations**

要想运行本篇的例子，需要将 System.ComponentModel.DataAnnotations 引用到项目中。

值得注意的是，Attribute 常用来给 class 或者 property 标注元数据信息，很显然 注解 Atrribute 也是 Attribute 的一种，它的分类大概有以下几种。

* Validation attribute

从名字上就能看出，主要用在 实体属性上，目的是用来管控 属性值 的有效性。

* Display attribute

用来指定数据如何在 UI 上展示。

* Modeling attribute

用来指定类之间的关系。

**理解 数据注解 类**

在 System.ComponentModel.Annotations 命名空间下有如下几个注解类。

* ConcurrencyCheck
* Key
* MaxLength
* Required
* StringLength
* Timestamp

**数据注解 案例展示**

现在新建一个 Author 类，代码如下：

public class Author

{

[Required(ErrorMessage = "{0} is required")]

[StringLength(50, MinimumLength = 3,

ErrorMessage = "First Name should be minimum 3 characters and a maximum of 50 characters")]

[DataType(DataType.Text)]

public string FirstName { get; set; }

[Required(ErrorMessage = "{0} is required")]

[StringLength(50, MinimumLength = 3,

ErrorMessage = "Last Name should be minimum 3 characters and a maximum of 50 characters")]

[DataType(DataType.Text)]

public string LastName { get; set; }

[DataType(DataType.PhoneNumber)]

[Phone]

public string PhoneNumber { get; set; }

[DataType(DataType.EmailAddress)]

[EmailAddress]

public string Email { get; set; }

}

下面的代码片段展示了如何给 Author 的属性赋值。

Author author = new Author();

author.FirstName = "Joydip";

author.LastName = "";

author.PhoneNumber = "1234567890";

author.Email = "joydipkanjilal@yahoo.com";

接下来在 Main 方法中去验证这些属性值是否符合验证规则。

ValidationContext context = new ValidationContext(author, null, null);

List<ValidationResult> validationResults = new List<ValidationResult>();

bool valid = Validator.TryValidateObject(author, context, validationResults, true);

if (!valid)

{

foreach (ValidationResult validationResult in validationResults)

{

Console.WriteLine("{0}", validationResult.ErrorMessage);

}

}

从上面代码可以看出，context 就是 验证规则的上下文，然后调用 Validator.TryValidateObject 去做规则验证，如果有属性不符合 数据注解 的规则，错误信息都会如实的记录到 List<ValidationResult> 集合中。

下面是完整的可供参考的代码。

class Program

{

static void Main(string[] args)

{

Author author = new Author();

author.FirstName = "Joydip";

author.LastName = "";

author.PhoneNumber = "1234567890";

author.Email = "joydipkanjilal@yahoo.com";

ValidationContext context = new ValidationContext(author, null, null);

List<ValidationResult> validationResults = new List<ValidationResult>();

bool valid = Validator.TryValidateObject(author, context, validationResults, true);

if (!valid)

{

foreach (ValidationResult validationResult in validationResults)

{

Console.WriteLine("{0}", validationResult.ErrorMessage);

}

}

Console.ReadLine();

}

}

public class Author

{

[Required(ErrorMessage = "{0} is required")]

[StringLength(50, MinimumLength = 3,

ErrorMessage = "First Name should be minimum 3 characters and a maximum of 50 characters")]

[DataType(DataType.Text)]

public string FirstName { get; set; }

[Required(ErrorMessage = "{0} is required")]

[StringLength(50, MinimumLength = 3,

ErrorMessage = "Last Name should be minimum 3 characters and a maximum of 50 characters")]

[DataType(DataType.Text)]

public string LastName { get; set; }

[DataType(DataType.PhoneNumber)]

[Phone]

public string PhoneNumber { get; set; }

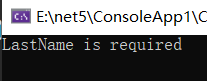
[DataType(DataType.EmailAddress)]

[EmailAddress]

public string Email { get; set; }

}

接下来把程序跑起来，你会在 Console 中看到如下错误信息。



**创建 自定义验证规则**

前面都是使用系统内置的，如果你有 自定义验证规则 的需求，那就要重写 ValidationAttribute 类中的 IsValid 方法，如下代码所示：

[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]

public class IsEmptyAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

var inputValue = value as string;

return !string.IsNullOrEmpty(inputValue);

}

}

然后将这个自定义的 IsEmptyAttribute 标记在 Author 类的 FirstName 和 LastName 属性上，如下代码所示：

[IsEmpty(ErrorMessage = "Should not be null or empty.")]

public string FirstName { get; set; }

[IsEmpty(ErrorMessage = "Should not be null or empty.")]

public string LastName { get; set; }

数据注解 最初是在 .NET 3.5 中作为 System.ComponentModel.DataAnnotations 的一部分被引入的，从那时起，它就越来越流行了，你可以直接在属性上加 数据注解 避免在项目各处中都有对 Author 实体类的验证逻辑，这种也太麻烦了，不是吗？

后面的文章中，我会带大家一起学习如何在 ASP.NET Core MVC 中使用 数据注解 去做 [model](https://so.csdn.net/so/search?q=model&spm=1001.2101.3001.7020) 的验证。

译文链接：<https://www.infoworld.com/article/3543302/how-to-use-data-annotations-in-csharp.html>

**更多高质量干货：参见我的 GitHub:**[**csharptranslate**](https://github.com/ctripxchuang/csharptranslate)

# 参考2

## [C#中的方括号[](特性、属性）](https://cloud.tencent.com/developer/tools/blog-entry?target=https%3A%2F%2Fwww.cnblogs.com%2Fyangxx-1990%2Fp%2F10919612.html&source=article&objectId=1765736)

C#中的方括号[](特性、属性）

约定：

1.”attribute” 和 ”attributes” 均不翻译

2.”property” 译为“属性”

3.msdn 中的原句不翻译

4.”program entity” 译为 ” 语言元素 ”

Attributes in C#

介绍

Attributes 是一种新的描述信息，我们既可以使用 attributes 来定义设计期信息（例如 帮助文件，文档的 URL ），还可以用 attributes 定义运行时信息（例如，使 XML 中的元素与类的成员字段关联起来）。我们也可以用 attributes 来创建一个“自描述”的组件。在这篇指南中我们将明白怎么创建属性并将其绑定至各种语言元素上，另外我们怎样在运行时环境下获取到 attributes 的一些信息。

定义

MSDN 中做如下定义 (ms-help://MS.MSDNQTR.2002APR.1033/csspec/html/vclrfcsh ARP spec\_17\_2.htm)

"An attribute is a piece of additional declarative information that is specified for a declaration."

使用预定义 Attributes

在 c# 中已有一小组预定义的 attributes ，在我们学习怎样创建自定义 attributes 前，先来了解下在我们的代码中使用那些预定义的 attributes.

代码语言：c#

1 using System;

2

3 public class AnyClass

4

5 {

6 [Obsolete( " Don't use Old method, use New method " , true )]

7

8 static void Old( ) { }

9

10 static void New( ) { }

11

12 public static void Main( )

13 {

14 Old( );

15 }

16 }

仔细看下该实例，在该实例中我们用到了 ”Obsolete”attribute ，它标记了一个不该再被使用的语言元素 （ 译者注：这里的元素为方法 ） ，该属性的第一个参数是 string 类型，它解释为什么该元素被荒弃，以及我们该使用什么元素来代替它。实际中，我们可以书写任何其它文本来代替这段文本。第二个参数是告诉编译器把依然使用这被标识的元素视为一种错误，这就意味着编译器会因此而产生一个警告。

当我们试图编译上面的上面的程序，我们会得到如下错误：

AnyClass.Old()' is obsolete: 'Don't use Old method, use New method'

开发自定义 Attributes

现在我们即将了解怎么开发自定义的 attributes 。这儿有个小小处方，有它我们就可以学会创建自定义的 attributes 。

在 C# 中，我们的 attribute 类都派生于 System.Attribute 类 ( A class that derives from the abstract class System.Attribute, whether directly or indirectly, is an attribute class. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration ) ，我们就这么行动吧。

代码语言：c#

using System;

public class HelpAttribute : Attribute

{

}

不管你是否相信我，就这样我们就已经创建了一个自定义 attribute 。现在就可以用它来装饰我们的类了，就像我们使用 obsolete attribute 一样。

代码语言：c#

[Help()]

public class AnyClass

{

}

注意：按惯例我们是用 ”Attribute“ 作为 attribute 类名的后缀，然而，当我们当我们把 attribute 绑定到某语言元素时，是不包含 “Attribute“ 后缀的。编译器首先在 System.Attribute 的继承类中查找该 attribute ，如果没有找到，编译器会把 “Attribute“ 追加到该 attribute 的名字后面，然后查找它。

但是迄今为止，该 attribute 没有任何用处。为了使它有点用处，让我们在它里面加点东西吧。

代码语言：c#

using System;

public class HelpAttribute : Attribute

{

public HelpAttribute(String Descrition\_in)

{

this .description = Description\_in;

}

protected String description;

public String Description

{

get

{

return this .description;

}

}

}

[Help( " this is a do-nothing class " )]

public class AnyClass

{

}

在上面的例子中，我们在 attribute 类中添加了一个属性，在最后一节中，我们将在运行时查询该属性。

定义或控制自定义 Attribute 的用法

AttributeUsage 类是另一预定义类 （ 译者注： attribute 类本身用这个 atrribute System.AttributeUsage 来标记 ） ，它将帮助我们控制我们自定义 attribute 的用法，这就是，我们能为自定义的 attribute 类定义 attributes 。

它描述了一个自定义 attribute 类能被怎样使用。

AttributeUsage 提供三个属性，我们能将它们放置到我们的自定义 attribute 类上， 第一个特性是：

ValidOn

通过这个属性，我们能指定我们的自定义 attribute 可以放置在哪些语言元素之上。这组我们能把自定义 attribute 类放置其上的语言元素被放在枚举器 AttributeTargets 中。我们可以使用 bitwise（ 译者注：这个词不知道怎么翻译好，但他的意思是可以这么用 ： [AttributeUsage ( ( AttributeTargets)4 , AllowMultiple = false , Inherited = false )], 4 代表就是 “ class ” 元素，其它诸如 1 代表“ assembly ”， 16383 代表“ all ”等等 ） 或者 ”.” 操做符绑定几个 AttributeTargets 值。 （译者注：默认值为 AttributeTargets.All )

AllowMultiple

该属性标识我们的自定义 attribte 能在同一语言元素上使用多次。 （ 译者注：该属性为 bool 类型，默认值为 false ，意思就是该自定义 attribute 在同一语言元素上只能使用一次 ）

Inherited

我们可以使用该属性来控制我们的自定义 attribute 类的继承规则。该属性标识我们的自定义 attribute 是否可以由派生类继承。（ （译者注：该属性为 bool 类型，默认值为 false ，意思是不能继承）

让我们来做点实际的东西吧，我们将把 AttributeUsage attribute 放置在我们的 help attribute 上并在它的帮助下，我们来控制 help attribute 的用法。

代码语言：c#

using System;

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false , Inherited = false )]

public class HelpAttribute : Attribute

{

public HelpAttribute(String Description\_in)

{

　　　　this .description = Description\_in;

}

protected String description;

public String Description

{

　　　　get

{

　　　　return this .description;

}

}

}

首先我们注意 AttributeTargets.Class . 它规定这个 help attribute 只能放置在语言元素 ”class” 之上。这就意味着，下面的代码将会产生一个错误。

AnyClass.cs: Attribute 'Help' is not valid on this declaration type.

It is valid on 'class' declarations only.

现在试着把它绑定到方法。

代码语言：c#

[Help( " this is a do-nothing class " )]

public class AnyClass

{

[Help( " this is a do-nothing method " )] // error

public void AnyMethod()

{

}

}

我们可以使用 AttributeTargets.All 来允许 Help attribute 可以放置在任何预定义的语言元素上，那些可能的语言元素如下 :

Assembly,

Module,

Class,

Struct,

Enum,

Constructor,

Method,

Property,

Field,

EVE nt,

Interface,

Parameter,

Delegate,

All = Assembly | Module | Class | Struct | Enum | Constructor | Method | Property | Field | EVE nt | Interface | Parameter | Delegate,

ClassMembers = Class | Struct | Enum | Constructor | Method | Property | Field | EVE nt | Delegate | Interface )

~ 现在考虑下 AllowMultiple = false . 这就规定该 attribute 不能在同一语言元素上放置多次 .

它产生了一个编译错误：

AnyClass.cs: Duplicate 'Help' attribute

Ok ！现在我们该讨论下最后那个属性了， ”Inherited”, 指出当把该 attribute 放置于一个基类之上，是否派生类也继承了该 attribute 。如果绑定至某个 attribute 类的 ”Inherited” 被设为 true, 那么该 attribute 就会被继承，然而如果绑定至某个 attribute 类的 ”Inherited” 被设为 false 或者没有定义，那么该 attribute 就不会被继承。

让我们假设有如下的类关系。

我们有四种可能的绑定 :

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false , Inherited = false )]

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true , Inherited = false) ]

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false , Inherited = true )]

[AttributeUsage(AttributeTargets.Class, AllowMultiple = true , Inherited = true) ]

第一种情况

如果我们查询（我们将在后面来了解如何在运行时来查询 attributes ）派生类中的 help attribute ，我们将不可能查询到因为 ”Inherited” 被设为了 false 。

第二种情况

第二种情况没有什么不同，因为其 ”Inherited” 也被设为了 false 。

第三种情况

为了解释第三种和第四种情况，让我们为派生类也绑定同一 attribute 。

代码语言：c#

[Help( " BaseClass " )]

public class Base

{

}

public class Derive : Base

{

}

现在我们查询相关的 help attribute ，我们将仅仅可以得到派生类的 attribute ，为什么这样是因为 help attribute 虽然允许被继承，但不能多次在同一语言元素上使用，所以基类中的 help attribute 被派生类的 help attribute 重写了。

第四种情况

在第四种情况中，当我们查询派生类的 help attribute 时，我们可以得到两个 attributes ，当然是因为 help attribute 既允许被继承，又允许在同一语言元素上多次使用的结果。

注意： AttributeUsage attribute 仅应用在那种是 System.Attribute 派生的 attriubte 类而且绑定值该 attriubte 类的 AllowMultiple 和 Inherited 均为 false 上才是有效的。

可选参数 vs. 命名参数

可选参数是 attribute 类构造函数的参数。它们是强制的，必须在每次在 attribute 绑定至某语言元素时提供一个值。而另一方面，命名参数倒是真正的可选参数，不是在 attribute 构造函数的参数。

为了更加详细的解释，让我们在 Help 类中添加另外的属性。

代码语言：c#

[Help( " BaseClass " )]

public class Base

{

}

[Help( " DeriveClass " )]

public class Derive : Base

{

}

当我们在 Class1 中查询 Help attribute 已经它的属性，我们将得到：

Help.Description : This is Class1

Help.Version :No Version is defined for this class

因为我们没有为 Version 这个属性定义任何任何值，所以在构造函数中设定的值被我们查询出来了。如果没有定义任何值，那么就会赋一个该类型的默认值（例如：如果是 int 型，默认值就是 0 ）。

现在，查询 Class2 的结果是：

Help.Description : This is Class2

Help.Version : 1.0

我们不能为了可选参数而使用多个构造函数，应该用已命名参数来代替。我们之所以称它们为已命名的，是因为当我们在构造函数为它们提供值时，我们必须命名它们。例如，在第二个类中，我们如是定义 Help 。

[Help( "This is Class2" , Version = "1.0" )]

在 AttributeUsage 例子中 , 参数 ”ValidOn” 是可选参数，而 “Inherited“ 和 “AllowMultiple“ 是命名参数。

注意：为了在 attribute 的构造函数中设定命名参数的值，我们必须为相应的属性提供一个 set 方法否则会引起编译期错误：

'Version' : Named attribute argument can't be a read only property

现在，我们在 Class3 中查找 Help attribute 及其属性会发生什么呢？结果是跟上面提到的相同的编译期错误。

'Desciption' : Named attribute argument can't be a read only property

现在我们修改下 Help 类，为属性 ”Description” 加一个 set 方法。现在的输出就是：

Help.Description : This is do-nothing class

Help.Version : 2.0

在屏幕后面究竟发生了什么呢？首先带有可选参数的构造函数被调用，然后，每个命名参数的 set 方法被调用，在构造函数中赋给命名参数的值被 set 方法所覆写。

参数类型

一个 attribute 类的参数类型被限定在如下类型中：

bool ,

byte,

char ,

double ,

float ,

int ,

long ,

short ,

string

System.Type

object

An enum type, provided that it and any types in which it is nested are publicly accessible. A one-dimensional array involving any of the types listed above

Attributes 标记

假设，我们想把 Help attribute 绑定至元素 assembly 。第一个问题是我们要把 Help attribute 放在哪儿才能让编译器确定该 attribute 是绑定至整个 assembly 呢？考虑另一种情况，我们想把 attribute 绑定至一个方法的返回类型上，怎样才能让编译器确定我们是把 attribute 绑定至方法的返回类型上，而不是整个方法呢？

为了解决诸如此类的含糊问题，我们使用 attribute 标识符，有了它的帮助，我们就可以确切地申明我们把 attribute 绑定至哪一个语言元素。

例如 :

[ assembly: Help( "this a do-nothing assembly" )]

这个在 Help attribute 前的 assembly 标识符确切地告诉编译器，该 attribute 被绑定至整个 assembly 。可能的标识符有：

assembly

module

type

method

property

EVE nt

field

param

return

在运行时查询 Attributes

现在我们明白怎么创建 attribtes 和把它们绑定至语言元素。是时候来学习类的使用者该如何在运行时查询这信息。

为了查询一语言元素上绑定的 attributes ，我们必须使用反射。反射有能力在运行时发现类型信息。

我们可以使用 .NET Framework Reflection APIs 通过对整个 assembly 元数据的迭代，列举出 assembly 中所有已定义的类，类型，还有方法。

记住那旧的 Help attribute 和 AnyClass 类。

代码语言：c#

[AttributeUsage(AttributeTargets.Class, AllowMultiple = false ,

Inherited = false )]

public class HelpAttribute : Attribute

{

public HelpAttribute(String Description\_in)

{

this .description = Description\_in;

this .verion = " No Version is defined for this class " ;

}

protected String description;

public String Description

{

get

{

return this .description;

}

}

protected String version;

public String Version

{

get

{

return this .version;

}

// if we EVE r want our attribute user to set this property,

// we must specify set method for it

set

{

this .verion = value;

}

}

}

[Help( " This is Class1 " )]

public class Class1

{

}

[Help( " This is Class2 " , Version = " 1.0 " )]

public class Class2

{

}

[Help( " This is Class3 " , Version = " 2.0 " ,

Description = " This is do-nothing class " )]

public class Class3

{

}

我们将在接下来的两节中在我们的 Main 方法中加入 attribute 查询代码。

查询程序集的 Attributes

在接下来的代码中，我们先得到当前的进程名称，然后用 Assembly 类中的 LoadForm （）方法加载程序集，再有用 GetCustomAttributes （）方法得到被绑定至当前程序集的自定义 attributes ，接下来用 foreach 语句遍历所有 attributes 并试图把每个 attribute 转型为 Help attribute （即将转型的对象使用 as 关键字有一个优点，就是当转型不合法时，我们将不需担心会抛出异常，代之以空值（ null ）作为结果），接下来的一行就是检查转型是否有效，及是不是为空，跟着就显示 Help attribute 的“ Description ”属性。

代码语言：c#

using System;

using System.Reflection;

using System.Diagnostics;

// attaching Help attribute to entire assembly

[assembly : Help( " This Assembly demonstrates custom attributes

creation and their run - time query. " )]

// our custom attribute class

public class HelpAttribute : Attribute

{

public HelpAttribute(String Description\_in)

{

//

// TODO: Add constructor logic here

this .description = Description\_in;

//

}

protected String description;

public String Description

{

get

{

return this .deescription;

}

}

}

// attaching Help attribute to our AnyClass

[HelpString( " This is a do-nothing Class. " )]

public class AnyClass

{

// attaching Help attribute to our AnyMethod

[Help( " This is a do-nothing Method. " )]

public void AnyMethod()

{

}

// attaching Help attribute to our AnyInt Field

[Help( " This is any Integer. " )]

public int AnyInt;

}

class QueryApp

{

public static void Main()

{

}

}

程序输出如下：

Description of QueryAttribute.exe:

This Assembly demonstrates custom attributes creation and

their run-time query.

Press any key to continue

查询类、方法、类成员的 Attributes

下面的代码中，我们惟一不熟悉的就是 Main （）方法中的第一行。

Type type = typeof (AnyClass);

它用 typeof 操作符得到了一个与我们 AnyClass 类相关联的 Type 型对象。剩下的查询类 attributes 代码就与上面的例子是相似的，应该不要解释了吧（我是这么想的）。

为查询方法和类成员的 attributes, 首先我们得到所有在类中存在的方法和成员，然后我们查询与它们相关的所有 attributes ，这就跟我们查询类 attributes 一样的方式。

代码语言：c#

class QueryApp

{

public static void Main()

{

HelpAttribute HelpAttr;

// Querying Assembly Attributes

String assemblyName;

Process p = Process.GetCurrentProcess();

assemblyName = p.ProcessName + " .exe " ;

Assembly a = Assembly.LoadFrom(assemblyName);

foreach (Attribute attr in a.GetCustomAttributes( true ))

{

HelpAttr = attr as HelpAttribute;

if ( null != HelpAttr)

{

Console.WriteLine( " Description of {0}:\n{1} " ,

assemblyName,HelpAttr.Description);

}

}

}

}

The output of the following program is.

Description of AnyClass:

This is a do-nothing Class.

Description of AnyMethod:

This is a do-nothing Method.

Description of AnyInt:

This is any Integer.

Press any key to continue

代码语言：c#

class QueryApp

{

public static void Main()

{

Type type = typeof (AnyClass);

HelpAttribute HelpAttr;

// Querying Class Attributes

foreach (Attribute attr in type.GetCustomAttributes( true ))

{

HelpAttr = attr as HelpAttribute;

if ( null != HelpAttr)

{

Console.WriteLine( " Description of AnyClass:\n{0} " ,

HelpAttr.Description);

}

}

// Querying Class-Method Attributes

foreach (MethodInfo method in type.GetMethods())

{

foreach (Attribute attr in method.GetCustomAttributes( true ))

{

HelpAttr = attr as HelpAttribute;

if ( null != HelpAttr)

{

Console.WriteLine( " Description of {0}:\n{1} " ,

method.Name,

HelpAttr.Description);

}

}

}

// Querying Class-Field (only public) Attributes

foreach (FieldInfo field in type.GetFields())

{

foreach (Attribute attr in field.GetCustomAttributes( true ))

{

HelpAttr = attr as HelpAttribute;

if ( null != HelpAttr)

{

Console.WriteLine( " Description of {0}:\n{1} " ,

field.Name,HelpAttr.Description);

}

}

}

}

}

代码语言：c#

[Help( " this is a do-nothing class " )]

[Help( " it contains a do-nothing method " )]

public class AnyClass

{

[Help( " this is a do-nothing method " )] // error

public void AnyMethod()

{

}

}

第三篇 托管

本文转自https://www.cnblogs.com/yangxx-1990/p/10919612.html

System.Runtime.InteropServices浅见

System.Runtime.InteropServices提供了相应的类或者方法来支持托管/非托管模块间的互相调用。

System.Runtime.InteropServices中几个比较重要的类:

DllImportAttribute : 该类提供对非托管动态链接库进行引用的方法，并告诉我们的编译器该程序的静态入口点是非托管的动态连接库，它的静态属性提供了对非托管动态链接库进行调用所必需的信息，作为最基本的要求，该类应该定义提供调用的非托管动态链接库的名称。成员详细信息

StructLayoutAttribute: 该类使得用户可以控制类或结构的数据字段的物理布局。

[StructLayout(LayoutKind.Explicit, Size=16, CharSet=CharSet.Ansi)]

public class MySystemTime

{

[FieldOffset(0)]public ushort wYear;

[FieldOffset(2)]public ushort wMonth;

[FieldOffset(4)]public ushort wDayOfWeek;

[FieldOffset(6)]public ushort wDay;

[FieldOffset(8)]public ushort wHour;

[FieldOffset(10)]public ushort wMinute;

[FieldOffset(12)]public ushort wSecond;

[FieldOffset(14)]public ushor wMilliseconds;

}

MarshalAsAttribute : 指示如何在托管代码和非托管代码之间封送数据。下面是MSDN给出的示例代码:

[C#]

//Applied to a parameter.

public void M1 ([MarshalAs(UnmanagedType.LPWStr)]String msg);

//Applied to a field within a class.

class MsgText {

[MarshalAs(UnmanagedType.LPWStr)] Public String msg;

}

//Applied to a return value.

[return: MarshalAs(UnmanagedType.LPWStr)]

public String GetMessage();

一个将三个类综合运用的实例:调用kernel32.dll中的非托管方法"GetSystemTime"将系统时间返回给定制的类MySystemTime并执行输出.

using System;

using System.Collections.Generic;

using System.Text;

using System.Runtime.InteropServices;

namespace DllImportTest

{

/// <summary>

/// 定义一个用于接收非托管API方法返回值的类

/// StructLayout定义了该类的各个成员在物理上的排列方式

/// </summary>

[StructLayout(LayoutKind.Explicit, Size = 16, CharSet = CharSet.Ansi)]

public class MySystemTime

{

[FieldOffset(0)]

public ushort wYear;

[FieldOffset(2)]

public ushort wMonth;

[FieldOffset(4)]

public ushort wDayOfWeek;

[FieldOffset(6)]

public ushort wDay;

[FieldOffset(8)]

public ushort wHour;

[FieldOffset(10)]

public ushort wMinute;

[FieldOffset(12)]

public ushort wSecond;

[FieldOffset(14)]

public ushort wMilliseconds;

}

/// <summary>

/// 用LibWrapper的静态方法来调用非托管API方法"GetSystemTime"

/// </summary>

class LibWrapper

{

[DllImport("kernel32.dll", EntryPoint = "GetSystemTime")]

//如果定义的方法名称与要进行封装的非托管API方法不同则需要在DLLImport中指定入口点.

public static extern void gettime([MarshalAs(UnmanagedType.LPStruct)]MySystemTime st);

}

class TestApplication

{

public static void Main()

{

try

{

MySystemTime sysTime = new MySystemTime();

//LibWrapper.GetSystemTime(sysTime);

LibWrapper.gettime(sysTime);

Console.WriteLine("The System time is {0}/{1}/{2} {3}:{4}:{5}", sysTime.wDay,

sysTime.wMonth, sysTime.wYear, sysTime.wHour, sysTime.wMinute, sysTime.wSecond);

}

catch (TypeLoadException e)

{

Console.WriteLine("TypeLoadException : " + e.Message);

}

catch (Exception e)

{

Console.WriteLine("Exception : " + e.Message);

}

}

}

}