

A photograph of a sailboat on the water, viewed from the deck. The boat's white hull and yellow sail are visible against a backdrop of green mountains under a clear sky.

反射和特性

siki 微信: devsiki QQ:804632564

什么是元数据，什么是反射

1. 程序是用来处理数据的，文本和特性都是数据，而我们程序本身（类的定义和BLC中的类）这些也是数据。
2. 有关程序及其类型的数据被称为元数据(metadata)，它们保存在程序的程序集中。
3. 程序在运行时，可以查看其它程序集或其本身的元数据。一个运行的程序查看本身的元数据或者其他程序集的元数据的行为叫做反射。

下面我们来学习如何使用Type类来反射数据，以及如何使用特性来给类型添加元数据。

Type位于System.Reflection命名空间下



Type类

预定义类型(int long 和string等), BCL中的类型(Console, IEnumerable等)和程序员自定义类型(MyClass, MyDel等)。每种类型都有自己的成员和特性。

BCL声明了一个叫做Type的抽象类, 它被设计用来包含类型的特性。使用这个类的对象能让我们获取程序使用的类型的信息。

由于 Type是抽象类, 因此不能利用它去实例化对象。关于Type的重要事项如下:

对于程序中用到的每一个类型, CLR都会创建一个包含这个类型信息的Type类型的对象。

程序中用到的每一个类型都会关联到独立的Type类的对象。

不管创建的类型有多少个示例, 只有一个Type对象会关联到所有这些实例。

System.Type类部分成员

成员	成员类型	描述
Name	属性	返回类型的名字
Namespace	属性	返回包含类型声明的命名空间
Assembly	属性	返回声明类型的程序集。
GetFields	方法	返回类型的字段列表
GetProperties	方法	返回类型的属性列表
GetMethods	方法	返回类型的方法列表



获取Type对象

获取Type对象有两种方式

1, Type t = myInstance.GetType(); //通过类的实例来获取Type对象

在object类有一个GetType的方法，返回Type对象，因为所有类都是从object继承的，所以我们可以在任何类型上使用GetType()来获取它的Type对象

2, Type t = typeof(ClassName); //直接通过typeof运算符和类名获取Type对象

获取里面的属性

```
FieldInfo[] fi = t.GetFields();
foreach(FieldInfo f in fi) {
    Console.WriteLine(f.Name + " ");
}
```



Assembly类

Assembly类在System.Reflection命名空间中定义，它允许访问给定程序集的元数据，它也包含了可以加载和执行程序集。



如何加载程序集？

1, Assembly assembly1 = Assembly.Load("SomeAssembly");根据程序集的名字加载程序集，它会在本地目录和全局程序集缓存目录查找符合名字的程序集。

2, Assembly assembly2 = Assembly.LoadFrom(@"c:\xx\xx\xx\SomeAssembly.dll");//这里的参数是程序集的完整路径名，它不会在其他位置搜索。

Assembly对象的使用

1, 获取程序集的全名 string name = assembly1.FullName;

2, 遍历程序集中定义的类型 Type[] types = theAssembly.GetTypes();

```
foreach(Type definedType in types){  
    //  
}
```

3, 遍历程序集中定义的所有特性(稍后介绍)

```
Attribute[] definedAttributes = Attribute.GetCustomAttributes(someAssembly);
```



什么是特性？

特性(attribute)是一种允许我们向程序的程序集增加元数据的语言结构。它是用于保存程序结构信息的某种特殊类型的类。

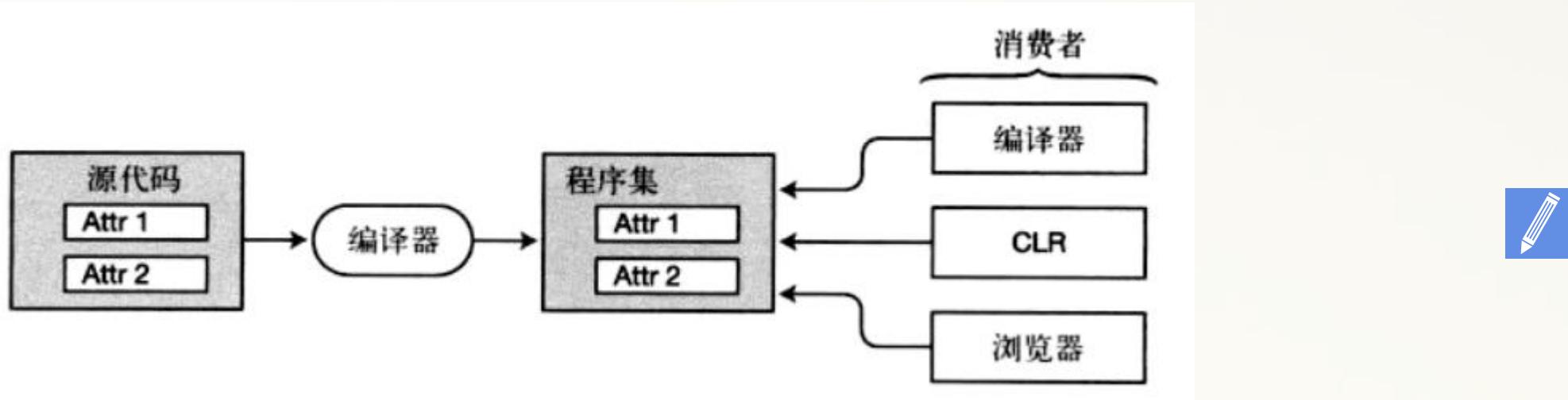
将应用了特性的程序结构叫做目标

设计用来获取和使用元数据的程序（对象浏览器）叫做特性的消费者

.NET预定了很多特性，我们也可以声明自定义特性



创建和使用特性



我们在源代码中将特性应用于程序结构；

编译器获取源代码并且从特性产生元数据，然后把元数据放到程序集中；

消费者程序可以获取特性的元数据以及程序中其他组件的元数据。注意，编译器同时生产和消费特性。

关于特性的命名规范，特性名使用Pascal命名法（首字母大写），并且以Attribute后缀结尾，当为目标应用特性时，我们可以不使用后缀。例如对于SerializableAttribute和MyAttributeAttribute这两个特性，我们把他们应用到结构是可以使用Serializable和MyAttribute。

应用特性

先看看如何使用特性。特性的目的是告诉编译器把程序结构的某组元数据嵌入程序集。我们可以通过把特性应用到结构来实现。

1. 在结构前放置特性片段来应用特性；
2. 特性片段被方括号包围，特性片段包括特性名和特性的参数列表；
3. 应用了特性的结构成为特性装饰。



案例1

```
[Serializable] //特性  
public class MyClass{  
    // ...  
}
```

案例2

```
[MyAttribute("Simple class", "Version 3.57")] //带有参数的特性  
public class MyClass{  
    //...  
}
```

Obsolete特性->.NET预定义特性

一个程序可能在其生命周期中经历多次发布，而且很可能延续多年。在程序生命周期的后半部分，程序员经常需要编写类似功能的新方法替换老方法。处于多种原因，你可能不再使用哪些调用过时的旧方法的老代码。而只想用新编写的代码调用新方法。旧的方法不能删除，因为有些旧代码也使用的旧方法，那么如何提示程序员使用新代码呢？可以使用Obsolete特性将程序结构标注为过期的，并且在代码编译时，显示有用的警告信息。

```
class Program{
    [Obsolete("Use method SuperPrintOut")]           //将特性应用到方法
    static void PrintOut(string str){
        Console.WriteLine(str);
    }
    [Obsolete("Use method SuperPrintOut", true)]//这个特性的第二个参数表示是否应该标记为错误，而不仅仅是警告。
    static void PrintOut(string str){
        Console.WriteLine(str);
    }
    static void Main(string[] args){
        PrintOut("Start of Main");
    }
}
```



Conditional特性

Conditional特性允许我们包括或取消特定方法的所有调用。为方法声明应用Conditional特性并把编译符作为参数来使用。

定义方法的CIL代码本身总是会包含在程序集中，只是调用代码会被插入或忽略。

```
#define DoTrace
```



```
class Program{
[Conditional("DoTrace")]
static void TraceMessage(string str) {
    Console.WriteLine(str);
}
static void Main() {
    TraceMessage("Start of Main");
    Console.WriteLine("Doing work in Main.")
    TraceMessage("End of Main");
}
```

调用者信息特性

调用者信息特性可以访问文件路径，代码行数，调用成员的名称等源代码信息。

1. 这个三个特性名称为CallerFilePath, CallerLineNumber和CallerMemberName
2. 这些特性只能用于方法中的可选参数

```
public static void PrintOut(string message, [CallerFilePath] string filename="", [CallerLineNumber] int lineNumber  
= 0, [CallerMemberName]string callingMember="") {  
    Console.WriteLine("Message:" + message);  
    Console.WriteLine("Line :" + lineNumber);  
    Console.WriteLine("Called from:" + callingMember);  
    Console.WriteLine("Message :" + message);  
}
```



DebuggerStepThrough特性

我们在单步调试代码的时候，常常希望调试器不要进入某些方法。我们只想执行该方法，然后继续调试下一行。DebuggerStepThrough特性告诉调试器在执行目标代码时不要进入该方法调试。有些方法小并且毫无疑问是正确的，在调试时对其反复单步调试只能徒增烦恼。要小心使用该特性，不要排除了可能出现bug的代码。

该特性位于System.Diagnostics命名空间下

该特性可用于类，结构，构造方法，方法或访问器

```
class Program{
    int _x=1;
    int X{
        get{return _x;}
        [DebuggerStepThrough]
        set{
            _x=_x*2;
            _x+=value;
        }
    }
    public int Y{get;set;}
    [DebuggerStepThrough]
    void IncrementFields(){
        X++;
        Y++;
    }
}
```



其他预定义特性

特性	意义
CLSCompliant	声明可公开的成员应该被编译器检查是否符合CLS。兼容的程序集可以被任何.NET兼容的语言使用
Serializable	声明结构可以被序列化
NonSerialized	声明结构不可以被序列化
DLLImport	声明是非托管代码实现的
WebMethod	声明方法应该被作为XML Web服务的一部分暴露
AttributeUsage	声明特性能应用到什么类型的程序结构。将这个特性应用到特性声明上



多个特性

我们可以为单个结构应用多个特性。有下面两种添加方式

独立的特性片段相互叠在一起

```
[Serializable]
```

```
[MyAttribute("Simple class", "Version 3.57")]
```

单个特性片段，特性之间使用逗号间隔

```
[Serializable, MyAttribute("Simple class", "Version 3.57")]
```



特性目标

我们可以将特性应用到字段和属性等程序结构上。我们还可以显示的标注特性，从而将它应用到特殊的目标结构。要使用显示目标，在特性片段的开始处放置目标类型，后面跟冒号。例如，如下的代码用特性装饰方法，并且还把特性应用到返回值上。

```
[return:MyAttribute("This value ...", "Version2.3")]
```

```
[method:MyAttribute("Print....", "Version 3.6")]
```

```
public long ReturnSettings()
```

```
...
```

c#定义了10个标准的特性目标。

```
event    field    method   param   property return   type    typevar  assembly module
```

其中type覆盖了类，结构，委托，枚举和接口。

typevar指定使用泛型结构的类型参数。



全局特性

我们可以通过使用assembly和module目标名称来使用显式目标说明符把特性设置在程序集或模块级别。程序集级别的特性必须放置在任何命名空间之外，并且通常放置在AssemblyInfo.cs文件中。AssemblyInfo.cs文件通常包含有关公司，产品以及版权信息的元数据。



```
[assembly: AssemblyTitle("ClassLibrary1")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("ClassLibrary1")]
[assembly: AssemblyCopyright("Copyright © 2015")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

自定义特性

应用特性的语法和之前见过的其他语法很不相同。你可能会觉得特性跟结构是完全不同的类型，其实不是，特性只是某个特殊结构的类。所有的特性类都派生自System.Attribute。



声明自定义特性

声明一个特性类和声明其他类一样。有下面的注意事项

声明一个派生自System.Attribute的类

给它起一个以后缀Attribute结尾的名字

(安全起见，一般我们声明一个sealed的特性类)

特性类声明如下：

```
public sealed class MyAttributeAttribute : System.Attribute{
```

...

特性类的公共成员可以是

字段

属性

构造函数



构造函数

特性类的构造函数的声明跟普通类一样，如果不写系统会提供一个默认的，可以进行重载

构造函数的调用

[MyAttribute("a value")] 调用特性类的带有一个字符串的构造函数



[MyAttribute("Version 2.3", "Mackel")] //调用特性类带有两个字符串的构造函数

构造函数的实参，必须是在编译期间能确定值的常量表达式

如果调用的是无参的构造函数，那么后面的（）可以不写

构造函数中的位置参数和命名参数

```
public sealed class MyAttributeAttribute: System.Attribute {  
    public string Description;  
    public string Ver;  
    public string Reviewer;  
    public MyAttributeAttribute(string desc) {  
        Description = desc;  
    }  
}  
  
//位置参数（按照构造函数中参数的位置）  
//命名参数，按照属性的名字进行赋值  
[MyAttribute("An excellent class", Reviewer = "Amy McArthur", Ver="3.12.3")]
```



限定特性的使用

有一个很重要的预定义特性可以用来应用到自定义特性上，那就是AttributeUsage特性，我们可以使用它来限制特性使用在某个目标类型上。

例如，如果我们希望自定义特性MyAttribute只能应用到方法上，那么可以用如下形式使用 AttributeUsage：

```
[AttributeUsage(AttributeTarget.Method)]  
public sealed class MyAttributeAttribute : System.Attribute{  
...}
```

AttributeUsage有是三个重要的公共属性如下：

ValidOn 保存特性能应用到的目标类型的列表，构造函数的第一个参数就是AttributeTarget
类型的枚举值

Inherited 一个布尔值，它指示特性是否会被特性类所继承 默认为true

AllowMultiple 是否可以多个特性的实例应用到一个目标上 默认为false

AttributeTarget枚举的成员

All Assembly Class Constructor Delegate Enum Event Field
GenericParameter Interface Method Module Parameter Property
ReturnValue Struct



多个参数之间使用按位或|运算符来组合

AttributeTarget.Method | AttributeTarget.Constructor

自定义特性一般遵守的规范

特性类应该表示目标结构的一些状态

如果特性需要某些字段，可以通过包含具有位置参数的构造函数来收集数据，可选字段可以采用命名参数按需初始化

除了属性之外，不要实现公共方法和其他函数成员

为了更安全，把特性类声明为**sealed**

在特性声明中使用**AttributeUsage**来指定特性目标组



案例

```
[AttributeUsage(AttributeTarget.Class)]  
public sealed class ReviewCommentAttribute: System.Attribute {  
    public string Description {get; set;}  
    public string VersionNumber {get; set;}  
    public string ReviewerID {get; set;}  
    public ReviewerCommentAttribute(string desc, string ver) {  
        Description = desc;  
        VersionNumber = ver;  
    }  
}
```



访问特性(消费特性)

1, 我们可以使用IsDefined方法来，通过Type对象可以调用这个方法，来判断这个类上是否应用了某个特性

```
bool isDefined = t.IsDefined( typeof(AttributeClass), false )
```

第一个参数是传递的需要检查的特性的Type对象

第二个参数是一个bool类型的，表示是否搜索AttributeClass的继承树

2, object[] attArray = t.GetCustomAttributes(false);

它返回的是一个object的数组，我们必须将它强制转换成相应类型的特性类型

bool的参数指定了它是否搜索继承树来查找特性

调用这个方法后，每一个与目标相关联的特性的实例就会被创建





关注公众号

发布最新的视频，文章和教学资源

THANK

@siki 微信: devsiki QQ:804632564
