C# Windows 文件系统的操作 C# 反射（Reflection）

C# 特性（Attribute）

特性（Attribute）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。您可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在它所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net 框架提供了两种类型的特性：*预定义*特性和*自定义*特性。

规定特性（Attribute）

规定特性（Attribute）的语法如下：

```
[attribute(positional_parameters, name_parameter = value, ...)]
element
```

特性（Attribute）的名称和值是在方括号内规定的，放置在它所应用的元素之前。positional_parameters 规定必需的信息，name_parameter 规定可选的信息。

预定义特性（Attribute）

.Net 框架提供了三种预定义特性：

- AttributeUsage
- Conditional
- Obsolete

AttributeUsage

预定义特性 **AttributeUsage** 描述了如何使用一个自定义特性类。它规定了特性可应用到的项目的类型。

规定该特性的语法如下：

```
[AttributeUsage(
   validon,
   AllowMultiple=allowmultiple,
   Inherited=inherited
)]
```

其中：

- 参数 validon 规定特性可被放置的语言元素。它是枚举器 *AttributeTargets* 的值的组合。默认值是 *AttributeTargets.All*。
- 参数 *allowmultiple*（可选的）为该特性的 *AllowMultiple* 属性（property）提供一个布尔值。如果为 true，则该特性是多用的。默认值是 false（单用的）。

- 参数 *inherited*（可选的）为该特性的 *Inherited* 属性（property）提供一个布尔值。如果为 true，则该特性可被派生类继承。默认值是 false（不被继承）。

例如：

```
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]
```

Conditional

这个预定义特性标记了一个条件方法，其执行依赖于指定的预处理标识符。

它会引起方法调用的条件编译，取决于指定的值，比如 **Debug** 或 **Trace**。例如，当调试代码时显示变量的值。

规定该特性的语法如下：

```
[Conditional(
   conditionalSymbol
)]
```

例如：

```
[Conditional("DEBUG")]
```

下面的实例演示了该特性：

实例

```
#define DEBUG
using System;
using System.Diagnostics;
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}
class Test
{
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
    static void function2()
```

```
    {
        Myclass.Message("In Function 2.");
    }
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main function
In Function 1
In Function 2
```

Obsolete

这个预定义特性标记了不应被使用的程序实体。它可以让您通知编译器丢弃某个特定的目标元素。例如，当一个新方法被用在一个类中，但是您仍然想要保持类中的旧方法，您可以通过显示一个应该使用新方法，而不是旧方法的消息，来把它标记为 obsolete（过时的）。

规定该特性的语法如下：

```
[Obsolete(
   message
)]
[Obsolete(
   message,
   iserror
)]
```

其中：

- 参数 *message*，是一个字符串，描述项目为什么过时的原因以及该替代使用什么。
- 参数 *iserror*，是一个布尔值。如果该值为 true，编译器应把该项目的使用当作一个错误。默认值是 false（编译器生成一个警告）。

下面的实例演示了该特性：

实例

```
using System;
public class MyClass
{
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
   static void OldMethod()
   {
      Console.WriteLine("It is the old method");
```

```
    }
    static void NewMethod()
    {
       Console.WriteLine("It is the new method");
    }
    public static void Main()
    {
       OldMethod();
    }
}
```

当您尝试编译该程序时，编译器会给出一个错误消息说明：

```
  Don't use OldMethod, use NewMethod instead
```

创建自定义特性（Attribute）

.Net 框架允许创建自定义特性，用于存储声明性的信息，且可在运行时被检索。该信息根据设计标准和应用程序需要，可与任何目标元素相关。

创建并使用自定义特性包含四个步骤：

- 声明自定义特性
- 构建自定义特性
- 在目标程序元素上应用自定义特性
- 通过反射访问特性

最后一个步骤包含编写一个简单的程序来读取元数据以便查找各种符号。元数据是用于描述其他数据的数据和信息。该程序应使用反射来在运行时访问特性。我们将在下一章详细讨论这点。

声明自定义特性

一个新的自定义特性应派生自 **System.Attribute** 类。例如：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

在上面的代码中，我们已经声明了一个名为 *DeBugInfo* 的自定义特性。

构建自定义特性

让我们构建一个名为 *DeBugInfo* 的自定义特性，该特性将存储调试程序获得的信息。它存储下面的信息：

- bug 的代码编号
- 辨认该 bug 的开发人员名字
- 最后一次审查该代码的日期
- 一个存储了开发人员标记的字符串消息

我们的 *DeBugInfo* 类将带有三个用于存储前三个信息的私有属性（property）和一个用于存储消息的公有属性（property）。所以 bug 编号、开发人员名字和审查日期将是 DeBugInfo 类的必需的定位（ positional）参数，消息将是一个可选的命名（named）参数。

每个特性必须至少有一个构造函数。必需的定位（ positional）参数应通过构造函数传递。下面的代码演示了 *DeBugInfo* 类：

实例

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
  private int bugNo;
  private string developer;
  private string lastReview;
  public string message;

  public DeBugInfo(int bg, string dev, string d)
  {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
  }

  public int BugNo
  {
      get
      {
          return bugNo;
      }
  }
  public string Developer
  {
      get
      {
          return developer;
      }
  }
  public string LastReview
```

```
  {
      get
      {
          return lastReview;
      }
  }
  public string Message
  {
      get
      {
          return message;
      }
      set
      {
          message = value;
      }
  }
}
```

应用自定义特性

通过把特性放置在紧接着它的目标之前，来应用该特性：

实例

```
[DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
[DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
class Rectangle
{
  // 成员变量
  protected double length;
  protected double width;
  public Rectangle(double l, double w)
  {
      length = l;
      width = w;
  }
  [DeBugInfo(55, "Zara Ali", "19/10/2012",
  Message = "Return type mismatch")]
  public double GetArea()
  {
      return length * width;
  }
  [DeBugInfo(56, "Zara Ali", "19/10/2012")]
  public void Display()
  {
      Console.WriteLine("Length: {0}", length);
      Console.WriteLine("Width: {0}", width);
      Console.WriteLine("Area: {0}", GetArea());
  }
}
```

在下一章中，我们将使用 Reflection 类对象来检索这些信息。

C# Windows 文件系统的操作 C# 反射（Reflection）

2 篇笔记

写笔记

C# 中利用 Conditional 定义条件方法

利用 Conditional 属性，程序员可以定义条件方法。Conditional 属性通过测试条件编译符号来确定适用的条件。当运行到一个条件方法调用时，是否执行该调用，要根据出现该调用时是否已定义了此符号来确定。如果定义了此符号，则执行该调用；否则省略该调用（包括对调用的参数的计算）。使用Conditional是封闭#if和#endif内部方法的替代方法，它更整洁，更别致、减少了出错的机会。

条件方法要受到以下限制：

- 条件方法必须是类声明或结构声明中的方法。如果在接口声明中的方法上指定Conditional属性，将出现编译时错误。
- 条件方法必须具有返回类型。
- 不能用override修饰符标记条件方法。但是，可以用virtual修饰符标记条件方法。此类方法的重写方法隐含为有条件的方法，而且不能用Conditional属性显式标记。
- 条件方法不能是接口方法的实现。否则将发生编译时错误。
- 如果条件方法用在“委托创建表达式”中，也会发生编译时错误

这里需要注意的是：如果创建一个没有定义任何条件的方法，那么默认只要调用就总是会执行此方法，如果你想通过条件来判断执行，那么该方法上必须至少包含一个conditional特性所定义的条件，它才会响应你定义的条件

GPF 1年前 (2018-01-18)

1、创建一个自定义特性：

```
// 描述如何使用一个自定义特性 SomethingAttribute
[AttributeUsage(AttributeTargets.All, AllowMultiple = true, Inherited = true)]

//********自定义特性SomethingAttribute**************//
public class SomethingAttribute : Attribute    {
    private string name; // 名字
    private string data; // 日期
    public string Name {
        get { return name; }
        set { name = value; }
    }
    public string Data    {
        get { return data; }
        set { data = value; }
    }
    public SomethingAttribute(string name)    {
        this.name = name;
        this.name = name;
    }
}
```

2、实例化自定义

```
[Something("Amy", Data = "2018-06-18")]
[Something("Jack", Data = "2018-06-18")]
class Test{}
```

3、获取自定义特性的中的变量

```
Type t = typeof(Test);
var something = t.GetCustomAttributes(typeof(SomethingAttribute),true);
foreach(SomethingAttribute each in something)
{
    Console.WriteLine("Name:{0}", each.Name);
    Console.WriteLine("Data:{0}",each.Data);
}
```

RightQ 9个月前 (06-18)