IT-Academy

ОБРАЗОВАТЕЛЬНЫЙ ЦЕНТР
ПРОГРАММИРОВАНИЯ И ВЫСОКИХ
ТЕХНОЛОГИЙ

2020г.

Рефлексия

AGENDA

- ✓ Метаданные типа
- ✓ Получение данных о типе
- ✓ Изменение данных с помощью рефлексии
- ✓ Создание экземпляров при помощи рефлексии

Рефлексия

Рефлексия – процесс инспектирования метаданных и скомпилированного кода во время выполнения приложения.

Рефлексия – процесс выявления типов во время выполнения приложения.

Каждая сборка содержит набор используемых классов, интерфейсов, а также их методов, свойств и т.д., из которых и складывается приложение. Рефлексия позволяет определить все эти составные элементы сборки.

Скомпилированный код в сборке включает практически все содержимое исходного кода. Некоторая информация утрачивается, например: имена локальных переменных, комментарии и директивы предпроцессора.

Рефлексия

Основной функционал рефлексии сосредоточен в пространстве имен System.Reflection.

Класс	Описание
Assembly	Класс, представляющий сборку и позволяющий манипулировать этой сборкой.
AssemblyName	Класс, хранящий информацию о сборке.
MemberInfo	Базовый абстрактный класс, определяющий общий функционал для классов **EventInfo**, **FieldInfo**, **MethodInfo** и **PropertyInfo**.
EventInfo	Класс, хранящий информацию об определенном событии.
FieldInfo	Класс, хранящий информацию об определенном поле типа.
MethodInfo	Класс, хранящий информацию об определенном методе.
PropertyInfo	Класс, хранящий информацию об определенном свойстве.
ConstructorInfo	Класс, хранящий информацию об определенном конструкторе.
ParameterInfo	Класс, хранящий информацию о параметре метода.

Метаданные типа

Метаданные типов содержат следующие данные:

- Имя, видимость, базовый класс и реализованные интерфейсы.
- Элементы типа (методы, поля, свойства, события, вложенные типы).

Экземпляр класса **System.Type** представляет метаданные для типа. Поскольку класс **Type** применяется очень широко, он находится в пространстве имен **System**, а не в **System.Reflection**.

Класс **System.Type** является абстрактным, поэтому операция получения типа должна на самом деле давать подкласс класса **Type**. Среда **CLR** использует внутренний подкласс сборки **mscorlib** по имени **RuntimeType**.

Метаданные типа

Класс **System.Type** представляет изучаемый тип, инкапсулируя всю информацию о нем. С помощью его свойств и методов можно получить эту информацию.

Метод	Описание
FindMembers()	Возвращает массив объектов **MemberInfo** данного типа.
GetConstructors()	Возвращает все конструкторы типа в виде набора объектов **ConstructorInfo**.
GetEvents()	Возвращает все события типа в виде массива объектов **EventInfo**.
GetFields()	Возвращает все поля типа в виде массива объектов **FieldInfo**.
GetInterfaces()	Возвращает все реализуемые данным типом интерфейсы в виде массива объектов **Type**.
GetMembers()	Возвращает все члены типа в виде массива объектов **MemberInfo**.
GetMethods()	Возвращает все методы типа в виде массива объектов **MethodInfo**.
GetProperties()	Возвращает все свойства типа в виде массива объектов **PropertyInfo**.

Рефлексия

Свойство	Описание
Name	Возвращает имя типа.
Assembly	Возвращает название сборки, где определен тип.
Namespace	Возвращает название пространства имен, где определен тип.
FullName	Возвращает полное имя типа.
IsArray	Возвращает **true**, если тип является массивом.
IsClass	Возвращает **true**, если тип представляет класс.
IsEnum	Возвращает **true**, если тип является перечислением.
IsInterface	Возвращает **true**, если тип представляет интерфейс.
AssemblyQualifiedName	Возвращает значение свойства **FullName** и полное имя сборки.

AssemblyQualifiedName – возвращает значение которое можно передавать методу **GetType()**, и он уникальным образом идентифицирует тип внутри стандартного контекста загрузки.

Метаданные типа

Тип имеет свойства **Namespace**, **Name** и **FullName**. В большинстве случаев **FullName** является объединением первых двух свойств:

```
Type t = typeof(System.Text.StringBuilder);
Console.WriteLine(t.Namespace); // System.Text
Console.WriteLine(t.Name); // StringBuilder
Console.WriteLine(t.FullName); // System.Text.StringBuilder
```

Из этого правила существуют два исключения: вложенные типы и закрытые обобщенные типы.

В случае вложенных типов содержащий тип присутствует только в **FullName**:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine (t.Namespace); // System
Console.WriteLine (t.Name); // SpecialFolder
Console.WriteLine (t.FullName);
// System.Environment+SpecialFolder
```

Метаданные типа

Имена обобщенных типов снабжаются суффиксами в виде символа ' , за которым следует количество параметров типа. Если обобщенный тип является несвязанным, это правило применяется и к **Name**, и к **FullName**:

```
Type t = typeof(Dictionary<, >); // Unbound (несвязанный)
Console.WriteLine(t.Name); // Dictionary'2
Console.WriteLine(t.FullName);
//System.Collections.Generic.Dictionary'2
```

Однако если обобщенный тип является закрытым, то свойство FullName приобретает важное дополнение: список всех параметров типа, для каждого из которых указывается полное имя, включающее сборку.

```
Console.WriteLine (typeof (Dictionary<int,string>).FullName);
// ВЫВОД:
System.Collections.Generic.Dictionary'2[[System.Int32, …],
[System.String, …]]
```

Получение данных о типе

Получить экземпляр **System.Type** можно путем вызова метода **GetType()** на любом объекте или с помощью операции **typeof** языка **C#**:

```
Type t1 = DateTime.Now.GetType();
// Экземпляр Type, полученный во время выполнения
Type t2 = typeof(DateTime);
// Экземпляр Type, полученный на этапе компиляции
```

Важно понимать, что:

- **typeof** – это оператор для получения типа, известного во время компиляции.
- **GetType()** – это метод, который вы вызываете для отдельных объектов, чтобы получить тип времени выполнения объекта.

```
object obj = "hello";

Type t1 = typeof(object); // ==> object
Type t2 = obj.GetType(); // ==> string!
```

Изменение данных с помощью рефлексии

Использование рефлексии также позволяет динамически получить и установить значение поля объекта по имени. Имея объект **MethodInfo**, **PropertyInfo** или **FieldInfo**, к нему можно динамически обращаться либо извлекать/устанавливать его значение. Это называется динамическим связыванием или поздним связыванием, т.к. выбор вызываемого члена производится во время выполнения, а не на этапе компиляции. Для этого используются методы **GetValue()** и **SetValue()**.

Использование рефлексии позволяет изменять даже скрытые поля объекта.

```
Assembly a = Assembly.Load(@"Libra");
Type t = a.GetType("booleanField");
FieldInfo field = t.GetField("enabled",
BindingFlags.NonPublic);
field.SetValue(a, false);
Console.WriteLine(field.GetValue(a));
```

Создание экземпляров

Сильные стороны рефлексии проявляются наиболее заметно лишь в том случае, если объект создается динамически во время выполнения. И для этого необходимо получить сначала список конструкторов, а затем экземпляр объекта заданного типа, вызвав один из этих конструкторов. Такой механизм позволяет получать во время выполнения экземпляр объекта любого типа, даже не указывая его имя в операторе объявления.

Динамически создать объект из его типа можно двумя путями:

- Вызвать статический метод **Activator.CreateInstance().**
- Вызвать метод **Invoke()** на объекте **ConstructorInfo**, который получен в результате вызова метода **GetConstructor()** на экземпляре **Type**.

Атрибуты

Атрибуты в **.NET** представляют специальные инструменты, которые позволяют встраивать в сборку дополнительные метаданные. Атрибуты могут применяться как ко всему типу (классу, интерфейсу и т.д.), так и к отдельным его частям (методу, свойству и т.д.). Основу атрибутов составляет класс **System.Attribute**, от которого образованы все остальные классы атрибутов.

В **.NET** имеется множество различных классов атрибутов. Например, при сериализации в различные форматы используются атрибуты **[Serializable]** и **[NonSerialized]**. С помощью рефлексии стандартные классы **.NET** получают использованные атрибуты и производят определенные действия. Например, атрибут **[Serializable]** указывает классу **BinaryFormatter**, что объекты с данным атрибутом можно сохранять в бинарный файл. В то ж время пока к классу с атрибутом не применена рефлексия, атрибут не размещается в памяти, и никакого влияния на данный класс не оказывает.

Атрибуты

С помощью атрибута **AttributeUsage** можно ограничить типы, к которым будет применяться атрибут. Например, мы хотим, чтобы определенный атрибут мог применяться только к классам.

```
[AttributeUsage(AttributeTargets.Class)]
public class RoleInfoAttribute : System.Attribute
{
// Блок кода
}
```

С помощью атрибута **AttributeUsage** можно ограничить типы, к которым будет применяться атрибут. Например, мы хотим, чтобы определенный атрибут мог применяться только к классам.

Атрибуты

Ограничение задает перечисление **AttributeTargets**, которое может принимать еще ряд значений:

- **All**: используется всеми типами;
- **Assembly**: атрибут применяется к сборке;
- **Constructor**: атрибут применяется к конструктору;
- **Delegate**: атрибут применяется к делегату;
- **Enum**: применяется к перечислению;
- **Event**: атрибут применяется к событию;
- **Field**: применяется к полю типа;
- **Interface**: атрибут применяется к интерфейсу;
- **Method**: применяется к методу;
- **Property**: применяется к свойству;
- **Struct**: применяется к структуре.

С помощью логической операции ИЛИ можно комбинировать эти значения. Например, атрибут может применяться к классам и структурам: **[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)].**

Задание

Возьмите файл Car.dll и сделайте так, чтобы я смог вызвать метод CarMethod() со строкой и увидел эту строку в консоли. DLL содержит:

```
namespace Car
{
    public class MyCar
    {
        private int age;

        private void CarMethod(string value)
        {
            if (age < 5)
            {
                throw new Exception("Hea");
            }

            Console.WriteLine(value);
        }
    }
}
```

Задание

Написать свой атрибут, который будет задавать валидационные правила для класса Car. Он должен смотеть, чтобы цвет и возраст машины соответствовал условиям (задайте сами).
Т е создали машину и проверили, что она валидная по условиям из атрибута.