Информатика

рефлексия

Сборка

- Сборка (***Assembly***) – результат компиляции исходного кода в .NET приложении
- Типы сборок:
  - Исполняемый файл *.exe
  - Библиотека классов *.dll

Составные части сборки

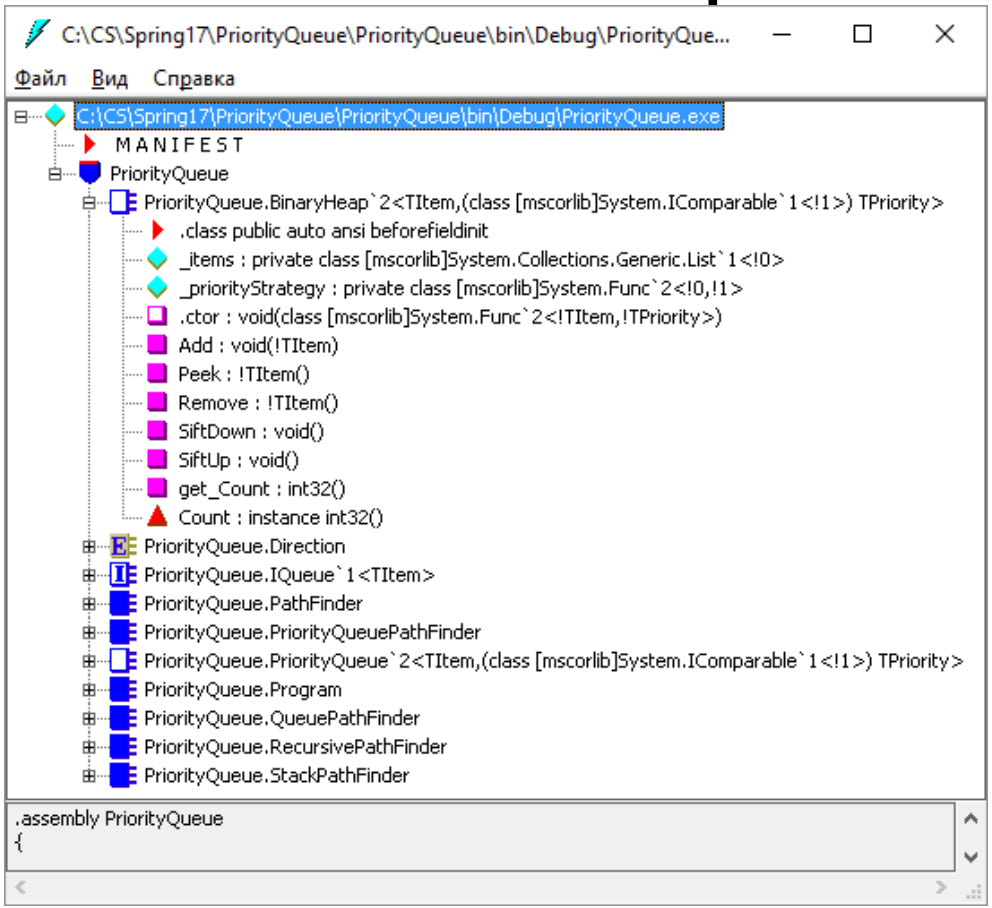
- Манифест – **Метаданные сборки**
- **Метаданные типов**
- CIL код
- Ресурсы

Ключевые технологии .NET

- **CIL и METAданные**
- Лежат в основе CLR
  - загрузки и выполнения программ
  - системы типов и управлении памятью
  - сборки мусора
  - системы проверки безопасности кода
  - обработки исключений
  - и работы др. систем

Визуальный анализ сборки

ILDASM



МЕТА

- Греческое слово **μετά**
- В эпистемологии (научное знание) означает «**о себе**»
- Что такое ***Метаданные - ?***

МЕТА

- Греческое слово **μετά**
- В эпистемологии (научное знание) означает «**о себе**»
- ***Метаданные – данные о данных***
  – описывающие другие данные

статья по метаданным и не только

Метаданные общедоступны

- Метаданные доступны любым программным компонентам и инструментам разработки
- При компиляции по метаданным проверяются зависимости сборок и соответствие используемых типов
- Среда разработки извлекает информацию о типах, предоставляет справочную информацию (IntelliSense)

Метаданные расширяемы

- Можно не только работать с предопределёнными метаданными, но и расширять их
- С помощью ***атрибутов*** можно снабдить программные элементы дополнительной информацией

Атрибуты

- Определяют метаданные
  - **сборки** (в манифесте сборки)
  - **типов** (классов, интерфейсов…)
  - **частей типов** (методов, свойств…)

статья об атрибутах на RSDN

System.Attribute

- ***Атрибуты – обычные классы***
- Все классы атрибутов – ***производные от System.Attribute***

Атрибуты сборки

- Файл AssemblyInfo.cs в проекте

```
// Управление общими сведениями о сборке осуществляется с помощью
// набора атрибутов. Измените значения этих атрибутов, чтобы изменить
// сведения, связанные со сборкой.
[assembly: AssemblyTitle("LearningLinq")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("LearningLinq")]
[assembly: AssemblyCopyright("Copyright ©  2017")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
...
```

Предопределённые атрибуты

- `[Serializable]` – возможность класса или структуры сохранять своё текущее состояние (в потоке)
- `[Obsolete]`– помечает тип или член как устаревший
- `[TestClass][TestMethod]` – помечают элементы модульного тестирования
  Microsoft.VisualStudio.TestTools.UnitTesting

Пользовательские атрибуты

```
public class RoleAttribute
  : Attribute
{
    public string RoleName
    {get;set;}
    public int RoleId {get;set;}
    public RoleAttribute(){}
    public RoleAttribute
    (string name)
    {
        RoleName = name;
    }
}
```

```
  [Role("customer")]
  public class User
  {
      public string Name {get;set;}
      public int Age { get; set; }
      public User(string n, int a)
      {
          Name = n;
          Age = a;
      }
  }
```

I need to go deeper

- Атрибуты можно добавлять к классам атрибутов?
- Есть стандартный кейс:

ограничение применения атрибута

Ограничение применения атрибута

- С помощью атрибута AttributeUsage можно ограничить типы, к которым будет применяться пользовательский атрибут
  - Ограничения задаются перечислением AttributeTargets
  - Можно комбинировать значения с помощью |

```
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Struct)]
public class RoleAttribute : Attribute
{ ... }
```

Что такое метаданные - понятно

- Мы знаем что такое метаданные
- Знаем о метаданных сборок и типов
- Знаем о добавлении информации в метаданные с помощью аннотаций
- ***А как с этими метаданными работать?***

Как получить доступ к метаданным?

- Сборка во время исполнения имеет доступ к собственным метаданным
  - может ***взглянуть на себя, внутрь себя, самоисследоваться***
- Отсюда название технологии – **рефлексия**
  - Дж. Локк (1632-1704) – «наблюдение, которому ум подвергает свою деятельность»

Рефлексия

- Работа с метаданными ***во время исполнения приложения***
- Основной функционал для рефлексии
  - Пространство имён **System.Reflection**
  - Класс **System.Type**

Классы System.Reflection

- Assembly, AssemblyName
- MemberInfo – базовый абстрактный класс
- EventInfo, FieldInfo, MethodInfo, PropertyInfo, ConstructorInfo
- Module
- ParamererInfo

System.Type

- Класс для получения информации о типе и его членах
- Можно получить тип
  - Динамически с помощью **GetType()** у экземпляра
  - Динамически с помощью поиска типа по имени **Type.GetType(“TypeName”)**
  - Статически с помощью **typeof(…)**

Пример получения типа

```
User user = new User("Tom", 30);
Type myType1 = user.GetType();

Type myType2 = Type.GetType("TestConsole.User", false, true);
//TypeName, ThrowOnError, IgnoreCase

Type myType3 = Type.GetType
("TestConsole.User, MyLibrary", false, true);
//Тип из другой сборки
```

Исследование типов

- **Is** Свойства для получение деталей типа
- **Get** Методы для получения информации о членах
- **FindMembers** – поиск членов на основе критериев

…

User

```
[Role("customer")]
public class User
{
    public string Name {get;set;}
    public int Age { get; set; }
    public User(string n, int a)
    {
        Name = n;
        Age = a;
    }
}
```

B Main:

```
Type myType = Type.GetType
 ("TestConsole.User", false, true);

bool check = myType.IsClass;
MethodInfo[] methods =
    myType.GetMethods();

foreach (MemberInfo mi in
myType.GetMembers())
    Console.WriteLine
    (mi.DeclaringType + " " +
     mi.MemberType + " " +
     mi.Name);
```

Результат работы GetMembers

```
[Role("customer")]
public class User
{
    public string Name {get;set;}
    public int Age { get; set; }
    public User(string n, int a)
    {
        Name = n;
        Age = a;
    }
}
```

```
TestConsole.User Method get_Name
TestConsole.User Method set_Name
TestConsole.User Method get_Age
TestConsole.User Method set_Age
System.Object Method ToString
System.Object Method Equals
System.Object Method GetHashCode
System.Object Method GetType
TestConsole.User Constructor .ct
TestConsole.User Property Name
TestConsole.User Property Age
```

А что с generic типами?

- У класса Type есть специальные свойства и методы для работы с обобщенными типами
  - Можно *конструировать тип* по обобщенному
  - Можно *узнавать обобщенный тип*
  - Можно *узнавать generic аргумент*
  - …

Метаданные обобщённых типов

```
Console.WriteLine(typeof(Dictionary<,>));
var type = typeof(List<>);
var closed = type.MakeGenericType(typeof(int));
Console.WriteLine(closed);
Console.WriteLine(closed.GetGenericTypeDefinition());
Console.WriteLine(closed.GetGenericArguments()[0]);
```

System.Collections.Generic.Dictionary`2[TKey,TValue]
System.Collections.Generic.List`1[System.Int32]
System.Collections.Generic.List`1[T]
System.Int32

Ещё один пример

```
Type nullable = typeof(bool?);

Console.WriteLine(
    nullable.IsGenericType &&
    nullable.GetGenericTypeDefinition() ==
        typeof(Nullable<>)); // True

Console.WriteLine(nullable);
//System.Nullable`1[System.Boolean]
```

Показать то, что скрыто

- Можно получать информацию не только о публичных членах
- Можно учитывать или игнорировать инкапсуляцию
- Перечисление BindingFlags
  DeclaredOnly, Instance, NonPublic, Public, Static

```
MethodInfo[] methods = myType.GetMethods
(BindingFlags.DeclaredOnly
    | BindingFlags.Instance
    | BindingFlags.NonPublic
    |BindingFlags.Public
);
```

Получение всех типов в сборке

```
using System;
using System.Reflection;
namespace App
{
    class Program
    {
        static void Main()
        {
            foreach (Type type in Assembly.GetExecutingAssembly().GetTypes())
                Console.WriteLine(type.Name);
            Console.ReadLine();
        }
    }
}
```

Динамическая загрузка сборок

- Можно динамически загружать сборки
  - из каталога приложения (private сборки)
  - Global Assembly Cache (разделяемые)
  - из файла на диске

➢ и анализировать не только собственные метаданные исполняемой сборки

Исследование типов в сборке

```
using System; using System.Reflection;
class Program
{
    static void Main(string[] args){
            Assembly asm = Assembly.LoadFrom("TestConsole.exe");
            Console.WriteLine(asm.FullName);
            // получаем все типы из сборки TestConsole.exe
            Type[] types = asm.GetTypes();
            foreach (Type t in types)
            {
                Console.WriteLine(t.Name);
            }
}}
```

Возвращаясь к атрибутам

- Нам нужно анализировать атрибуты сборок и типов
  - в том числе пользовательских
- Поскольку информация об атрибутах записывается в метаданные, никаких проблем нет
- Работаем с ними так же, как и с информацией о членах

Исследование аттрибутов

- Есть методы расширения **GetCustomAttribute(s)**

```
Type t = typeof(TestConsole.User);
object[] attrs = t.GetCustomAttributes(false);
foreach (RoleAttribute roleAttr in attrs)
{
    Console.WriteLine
        (roleAttr.RoleName+" "+roleAttr.RoleId);
}
```

Атрибуты, рефлексия и память

- **Атрибуты – экземпляры** производных от System.Attribute классов
- Под них выделяется память
- Но выделяется только при работе рефлексии

Информация о типах и экземплярах

- Мы можем получать всю информацию о типе
  - по его экземпляру
  - по имени типа из текущей сборки
  - по имени, даже если тип не известен в момент компиляции
- Мы можем получать информацию об экземплярах и их членах

А что, если...

- для полного счастья научиться ещё и создавать экземпляры по данным о типах, а потом ещё вызывать их методы...

Позднее связывание?

- Нужна возможность создания экземпляра определённого типа с последующим вызовом его методов без жесткой привязки к конкретному методу во время компиляции

Почему бы и нет

- В этом нам поможет класс System.Activator, содержащий метод CreateInstance
  - несколько перегрузок

```
Assembly asm = Assembly.LoadFrom("MyApp.exe");

Type t = asm.GetType("MyApp.Program", true, true);

// создаем экземпляр класса Program
object obj = Activator.CreateInstance(t);
```

- Осталось научиться вызывать метод

Вызвать метод у объекта?

- Если известен интерфейс – легко!
- С помощью as безопасно преобразуем полученный объект к интерфейсной ссылке и вызываем метод

А если интерфейс неизвестен?

- Нужно динамически находить метод и его вызывать
- С помощью метода Invoke у объекта MemberInfo или InvokeMember через тип Type

код примера

Волшебство

```
Assembly asm = Assembly.LoadFrom("TestConsole.exe");

Type t = asm.GetType("Program", true, true);

// создаем экземпляр класса Program
object obj = Activator.CreateInstance(t);

// получаем метод GetResult
MethodInfo method = t.GetMethod("GetResult");

// вызываем метод, передаем ему значения для параметров и получаем результат
object result = method.Invoke(obj, new object[] { 6, 100, 3 });
Console.WriteLine(result);
```

Доступ к свойству

- Аналогичен вызову метода
  - var type = horse.GetType();
  - var property = type.GetProperty(“Name”);
  - var value= property.GetValue(horse,null);

Пример рефлексии

```
Type t = Type.GetType(Console.ReadLine());

Type t2 = Type.GetType(Console.ReadLine());

string methodName = Console.ReadLine();

MethodInfo m = t.GetMethod(methodName);

object o = Activator.CreateInstance(t);

object o2 = Activator.CreateInstance(t2);

Console.WriteLine(m.Invoke(o, new[] { o2 }));
```

Работает, если подать

System.Collections.ArrayList

System.Int32

Add

Итого

- Мы можем динамически загружать сборки и работать с типами, не известными на этапе компиляции
- А что, если мы ещё и код методов налету будем генерировать и потом вызывать сгенерированные методы?

Hello world!

```
var dynMeth = new DynamicMethod("Foo", // Название метода
    null, // Возвращаемый тип
    null, // Типы аргументов
    typeof(Program)); // Родительский тип
ILGenerator gen = dynMeth.GetILGenerator();
gen.EmitWriteLine("Hello world");
gen.Emit(OpCodes.Ret);
dynMeth.Invoke(null, null); // Hello world
```

изощрённый динамический Hello World!

Верх изощрения

- Генетическое программирование
  - эволюционные алгоритмы, использующие методику естественного отбора
  - генерируются программы, выбираются лучшие варианты, скрещиваются и т.д.

ссылка на статью в msdn

Рефлексия по рефлексии

- Рефлексия – мощный инструмент для создания максимально гибких приложений
- Рефлексия – дорогая. Не лучшее решение для требовательных к производительности приложений
- Если можно обойтись без неё, лучше так и поступить
- Проверка типов на этапе компиляции – надёжнее

По рефлексии рефлексия

- Питает джедая Сила. Но бойся темной стороны…
- …Если раз ступишь на темную тропу, навсегда она твою судьбу определит.

Вопросы?

e-mail: marchenko@it.kfu.ru