Рефлексия

Юрий Литвинов
y.litvinov@spbu.ru

Введение

Рефлексия[1] (иногда весьма некрасиво переводимая как «отражение») — это языковой механизм, позволяющей программе прямо во время выполнения получать информацию о чисто компиляторных вещах, таких как типы объектов, информация об их методах, полях и т.д. и т.п. При этом рефлексия позволяет не просто получать эту информацию, но и пользоваться ей — создавать объекты типов, которые мы узнали с помощью рефлексии, вызывать методы у объектов, менять значения полей (даже private!) и т.п.

Разумеется, рефлексия требует поддержки со стороны компилятора и среды времени выполнения, чтобы сохранять после компиляции и предоставлять во время выполнения нужную информацию. Поэтому, например, в Си рефлексии нет вообще, а в C++ она весьма и весьма жалкая (RTTI), в Java она очень развита, но есть типичные для Java проблемы с генериками, в C# (точнее, в .NET) она хороша, в языках наподобие JavaScript вообще отлична (потому что там компилятора нет вообще, и программа имеет буквально всю информацию о себе во время выполнения).

Зачем рефлексия вообще нужна — без неё не обходится ни один инструмент, требующий какой-то работы с кодом. Например, тестовые системы используют рефлексию, чтобы найти все тестовые методы, запустить их и посмотреть, что получилось. Так что да, на самом деле мы используем рефлексию уже очень давно. Ещё применения:

- библиотеки сериализации — Newtonsoft JSON нам уже встречался, в этой лекции ещё будет про System.Xml.Serialization, все остальные библиотеки устроены примерно так же. Они используют рефлексию, чтобы обходить сериализуемый объект и извлекать информацию о типах его полей, или при десереализации, чтобы создать и обойти целевой объект, заполняя его поля значениями.

- Библиотеки для работы с базами данных, примерно для тех же целей — обойти классы из программы и отобразить их в таблицы из БД или наоборот. Про эту сферу применения рефлексии подробнее в следующей лекции.

- Анализаторы кода могут работать на уже скомпилированных библиотеках, рефлексией обходя код и ища типичные ошибки, либо считая разные метрики. Так может работать, например, FxCop, который теперь встроен в Visual Studio и называется там Run Code Analysis.

[1] Изложение в этой лекции ведётся в основном по книге J. Richter, «CLR via C#», которая, напоминаю, обязательна к прочтению для любого .NET-программиста

- Плагинные системы для приложений бывает удобно реализовывать на рефлексии — вы качаете откуда-то из интернета плагин, рефлексией узнаёте поддерживаемые им возможности, рефлексией же создаёте экземпляр плагина — и спокойно работаете с ним. Причём, плагины могут быть написаны задолго после самого приложения.

- Inversion-of-Control-контейнеры, например, `http://unitycontainer.org/`, с помощью рефлексии берут на себя всю работу по созданию и инициализации структуры объектов, тем самым позволяя не писать длинных var что-то-там = new Что-то-там в Main-е, не путаться с порядком инициализации и не думать о том, кому, кого и куда надо передавать, в конструктор или в свойство. Такие штуки очень распространены в веб-приложениях и вообще в enterprise-системах.

- И т.д. и т.п.

Однако рефлексия — это отнюдь не способ решения всех проблем, механизм расправы над компилятором за все унижения, пережитые на первом курсе или способ «круто» вызывать методы у класса. Рефлексия на самом деле — это очень нишевый механизм, прежде всего для реализации низкоуровневой функциональности, которую по-другому более-менее и не сделать, ориентированный на людей, которые знают, что делают. И её вообще не должно быть в реализации бизнес-логики приложения. Если вы чувствуете соблазн сделать что-то рефлексией, подумайте ещё раз.

Причин этому две. Основная — рефлексия позволяет обойти систему типов и ограничения семантики языка: вызывать private-методы, обращаться к полям классов, создавать объекты неведомых типов и даже генерить код на лету. Это только звучит здорово, а на деле превращает программу в мешанину нечитаемого кода, в котором все глупые ошибки, которые бы сразу поймал компилятор, доживают до времени выполнения. Вторая, менее важная — механизмы рефлексии работают небыстро, так что скорость работы программы от чрезмерного увлечения рефлексией может сильно пострадать.

Конкретно в .NET всё, что связано с рефлексией, живёт в пространстве имён System.Reflection и опирается на формат хранения байт-кода в .NET-машине. В .NET байт-код сохраняет всю информацию о типах, даже конкретные имена методов, классов, параметров и т.д. и т.п., разве что комментарии из кода удаляются. Даже информация о параметрах-типах генериков в .NET хранится и .NET-машина умеет отличать генерики и их инстанциации с разными параметрами-типами, и даже инстанцировать генерики, подставляя им фактические параметры-типы, прямо во время выполнения — в отличие от Java, где для реализации генериков был выбран подход стирания типов.

В .NET байт-код хранится в *сборках*, сборки же — самые крупные единицы кода, которыми оперируют механизмы рефлексии. Сборка — это то, во что собирается один проект на .NET-языках, то есть либо .dll, либо .exe. Более формально, сборка — это единица развёртывания программы, то есть один файл, который можно куда-то установить[2].

Сборки же хранят ту самую метаинформацию, доступ к которой можно получить во время выполнения через рефлексию — таблицы модулей, типов, методов, полей, параметров, свойств и событий. Модуль — довольно тайное понятие, модуль — это часть сборки, но Visual Studio умеет собирать сборки только из одного модуля, поэтому про модули в

[2] На самом деле, это неправда, бывают многофайловые сборки, но Visual Studio их делать не умеет. См. «CLR via C#», J. Richter, по поводу всех жутких подробностей того, как устроены сборки в .NET и что они на самом деле такое

.NET никто не знает. Всё остальное, думаю, понятно — разве что обратите внимание на тот факт, что таблицы устроены наподобие таблиц реляционных баз данных, то есть информация о типе лежит не в одну кучу, а есть отдельно таблица типов, таблица методов, на которую таблица типов ссылается, таблица параметров, на которую ссылается таблица методов и т.п. Рефлексия на самом деле просто грузит сборку в память и вычитывает информацию из этих таблиц.

Посмотреть на то, как устроена сборка и какая метаинформация и как в ней лежит, можно с помощью утилиты ILDasm, которая поставляется с .NET. Однако лучше пользоваться dotPeek (https://www.jetbrains.com/decompiler/), он столь же бесплатный, что и ILDasm, но с более приятным пользовательским интерфейсом[3].

А ещё есть понятие AppDomain — это группа загруженных в память сборок, у которых в рамках одной .NET-машины есть своё адресное пространство и которые работают как одно приложение. В рамках одного процесса может быть несколько AppDomain-ов, и есть как минимум один. Код из разных AppDomain-ов не может ничего делать друг с другом (вызывать методы, получать доступ к общей памяти), то есть AppDomain-ы реально ведут себя как мини-процессы в рамках одного процесса (впрочем, есть механизм *маршаллинга*, который позволяет им общаться). Нужны AppDomain-ы для изоляции частей приложения и появились в основном для нужд веб-приложений. Есть веб-сервер, это один процесс, в котором запущена .NET-машина. Есть веб-приложения, это .dll-ки (и их зависимости, тоже .dll-ки), которые загружаются в веб-сервер как плагины. Их нельзя сделать разными процессами, потому что разные процессы не смогут слушать один сетевой порт, а запускать и останавливать процессы прямо при обработке сетевого запроса из веб-сервера слишком дорого, и нельзя, чтобы разные веб-приложения могли портить жизнь друг другу. Поэтому AppDomain-ы в .NET и появились. Про них тоже приходится помнить, потому что рефлексия работает в пределах AppDomain-а, и этот термин часто поминается в разных местах, связанных с рефлексией.

1. Сборки

Для манипуляции сборками в .NET есть класс Assembly, со статическими методами, позволяющими грузить сборки в разных ситуациях:

```
public class Assembly {
    public static Assembly Load(AssemblyName assemblyRef);
    public static Assembly Load(String assemblyString);
    public static Assembly Load(byte[] rawAssembly)
    public static Assembly LoadFrom(String path);
    public static Assembly ReflectionOnlyLoad(String assemblyString);
    public static Assembly ReflectionOnlyLoadFrom(String assemblyFile);
}
```

Чаще всего используются методы Load, принимающие имя сборки — они заставляют .NET-машину сначала поискать нужную сборку в GAC[4], затем в рабочей папке приложе-

[3] Тут я даже не могу сказать, что JetBrains не заплатила мне за рекламу

[4] Global Assembly Cache, то место, куда обычно ставятся системные библиотеки и сборки, которые мы хотим использовать глобально по всей системе (что нынче не в моде вообще)

ния и по путям, указанным в app.config. Перегрузка с rawAssembly позволяет загрузить сборку в виде просто массива байтов (например, скачанного по сети, без сохранения в файловую систему). LoadFrom применяет довольно странные правила по поиску сборки, которую надо загрузить, а переданный ему путь — скорее как подсказку, где искать сборку (подробности, как обычно, у Рихтера), зато принимает URL, сам качает сборку и грузит её (.NET отчасти потому и .NET, что проектировался для такого вот сетевого деплоя, но потом это как-то не прижилось). Методы ReflectionOnlyLoad грузят только метаинформацию из сборки, так что гарантированно не могут исполнить оттуда код, что хорошо и с точки зрения безопасности, и с точки зрения скорости работы.

Пример программной загрузки сборки:

```
var a = Assembly.LoadFrom(@"http://example.com/ExampleAssembly.dll");
```

При этом, когда мы грузим сборку, надо, чтобы такая сборка ещё не была загружена. При этом выгрузить сборку внезапно нельзя. Хотя, не так уж и внезапно — представьте себе, что было бы, если бы мы загрузили сборку, насоздавали разных объектов из этой сборки, а потом выгрузили сборку. На самом деле, выгружать можно целый AppDomain — поскольку .NET-машина гарантирует, что между AppDomain-ами не может быть прямых ссылок, выгрузка всех сборок в AppDomain-е и всех объектов, что в нём были, оставит приложение в консистентном состоянии, висячих ссылок гарантированно не будет. Кстати, если в сборках внезапно классы с одинаковыми именами, то более свежая сборка замещает более старую, так что если вы попытаетесь в свою программу загрузить её же сборки, это вам сделать дадут, но рефлексия будет работать, скажем так, необычно. Например, Type() у объекта внезапно может оказаться не равен typeof его класса.

1.1. Сильные и слабые имена сборок

Даже имена сборок в .NET не так просты. В .NET есть механизм защиты от неавторизованного изменения байт-кода, который упоминался в предыдущей лекции, в рассказе о сертификатах. Чтобы им воспользоваться, нужна пара из публичного и приватного ключей, так что этот механизм применяется не всегда: в ходе разработки и отладки сборки не подписываются. Подписанные сборки называются *сильно именованными* (Strongly Named Assemblies) и, соответственно, имеют *сильное имя*, неподписанные — *слабо именованными* (Weakly Named).

Сильно именованные сборки подписываются асимметричным ключом. Публичная часть ключа внедряется в саму сборку, дальше от сборки считается SHA-1-хеш, шифруется приватным ключом и тоже внедряется в сборку. При загрузке сборки (при запуске приложения или рефлексией) .NET-машина считает SHA-1-хеш и проверяет, что он совпал с подписанным. Если хеш не сошёлся, значит, в сборке были изменения после того, как её собрал её автор, и такую сборку грузить .NET не будет. Этот механизм позволяет проверить, что сборка именно такая, какой её собрал автор, но не позволяет проверить идентичность автора — любой может сгенерить себе пару ключей, подписывать ими сборки и публиковать их.

Сильное имя сборки состоит из имени файла (без расширения), в котором сборка находится, версии (в формате Major Version.Minor Version.Release.Build), культуры сборки (под какую она локаль), и PublicKeyToken — это короткий хеш её публичного

ключа, нужный для того, чтобы однозначно идентифицировать сборку в любом случае. PublicKeyToken никаким криптографическим целям не служит, .NET-машина всегда проверяет настоящий ключ при загрузке сборки. Нужен он для того, чтобы если два разработчика выложили сборку MyCoolAssembly 1.0.0.0, .NET мог эти сборки различить. Вот пример сильного имени:

```
"MyTypes, Version=1.0.8123.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

Только сборки с сильными именами могут быть установлены в GAC (инструментом GACUtil, поставляющимся с .NET-ом, подробности, как обычно, в Рихтере и в документации, благо GAC не очень нужен нынче). При этом сильные сборки могут ссылаться (в References) только на сильные сборки, что логично — если ссылаться на слабую сборку, её может подменить хакер, и тогда сколько мы свою сборку ни будем подписывать, всё равно будет исполняться нехороший код. Все сборки из стандартной библиотеки и из NuGet — сборки с сильными именами.

Все сборки, которые собираются без ключей и подписи (то есть по сути все, что мы использовали до этого и будем использовать дальше) — это сборки со слабыми именами, они могут ссылаться на любые сборки и идентифицируются только своим именем файла. Соответственно, они не могут быть в GAC, и ищутся только в рабочей папке приложения — их уникальность гарантирует сама файловая система.

Вот пример того, как загрузить сборку с сильным именем и распечатать имена всех типов, которые в ней находятся:

```
using System;
using System.Reflection;

public static class Program {
    public static void Main() {
        string dataAssembly = "System.Data, version=4.0.0.0, "
                + "culture=neutral, PublicKeyToken=b77a5c561934e089";
        LoadAssemblyAndShowPublicTypes(dataAssembly);
    }

    private static void LoadAssemblyAndShowPublicTypes(string assemblyId) {
        var a = Assembly.Load(assemblyId);
        foreach (Type t in a.ExportedTypes) {
            Console.WriteLine(t.FullName);
        }
    }
}
```

2. Типы

Если мы загрузили сборку не в Reflection-Only-режиме, мы можем создавать объекты типов, которые в этой сборке объявлены. Причём, аж тремя способами:

- System.Activator.CreateInstance — можно передавать тип (объект типа Type) или строку с именем типа. Версии со строкой возвращают System.Runtime.Remoting.ObjectHandle (часть машинерии по работе между AppDomain-ами), и надо вызвать метод Unwrap() для того, чтобы добраться до реального объекта. В любом случае либо вызывается конструктор без параметров, либо можно передать массив Object-ов, которые и будут параметрами конструктора (при этом найдётся наиболее подходящий, по типам времени выполнения параметров).
- System.Activator.CreateInstanceFrom — принимает имя сборки и имя типа как строки. Сначала он вызывает LoadFrom для сборки, а затем уже создаёт экземпляр типа. Тоже вызывает конструктор, обойти вызов конструктора рефлексией можно, но сложно — все обычные методы создания объекта конструктор вызывают.
- System.Reflection.ConstructorInfo.Invoke — это, в отличие от двух предыдущих, не статичческий метод, он требует объекта ConstructorInfo, который можно раздобыть из Type. Это просто вызов конструктора (несколько дольше писать, чем предыдущие варианты).

При этом учтите, что рефлексия ничего не знает о синонимах типов и ключевых словах конкретных языков программирования. Например, int надо создавать как System.Int32. Например,

```
var zero = Activator.CreateInstance("mscorlib.dll", "System.Int32").Unwrap();
```

Это весьма извращённый метод сделать то же, что `int zero = 0;`. Созданный объект будет объектом-значением и будет размещаться на стеке, как обычно. Вообще, все примитивные типы и многие другие живут в mscorlib.dll.

С генериками всё сложнее. Поскольку генерикам нужны параметры-типы, создавать объект можно только если все параметры-типы переданы (то есть генерик суть *закрытый тип*). Однако получать информацию можно и о типах, у которых не все параметры-типы подставлены (то есть об *открытых типах*). А можно и выполнить подстановку параметров-типов программно, и тогда уже получить возможность создать объект. Например, вот так можно инстанцировать генерик и создать объект:

```
using System;
using System.Reflection;

internal sealed class Dictionary<TKey, TValue> { }

public static class Program {
    public static void Main() {
        Type openType = typeof(Dictionary<,>);
        Type closedType = openType.MakeGenericType(
                typeof(String), typeof(Int32));
        Object o = Activator.CreateInstance(closedType);
        Console.WriteLine(o.GetType());
    }
}
```

`typeof(Dictionary<,>);` вернёт нам объект типа Type, соответствующий открытому типу. MakeGenericType принимает объекты типа Type и возвращает другой Type, открытый или закрытый в зависимости от того, все ли параметры-типы были заполнены. В нашем случае параметра-типа было два, они оба заполнены фактическими параметрами-типами, так что получаем закрытый тип и можем его инстанцировать методом Activator.CreateInstance. После этого `o.GetType()` вернёт нам информацию уже о конкретном типе генерика.

2.1. Пример, как сделать свою плагинную систему

Одно из хороших применений рефлексии — реализация плагинной системы, позволяющей динамически расширять возможности приложения, даже во время работы. Например, вы пишете какой-нибудь редактор и хотите поддерживать разные форматы файлов — необходимый для работы с ними код можно поставлять в виде плагинов, которые грузятся при старте системы и могут быть скачаны из интернета (в том числе и средствами самого редактора) и подключены в любой момент (так, например, устроена Visual Studio Code, да и сама Visual Studio).

Сделать это можно в три этапа:

- Сначала продумать интерфейс, по которому система будет взаимодействовать с плагинами. Это, пожалуй, самая сложная часть, поскольку именно этот интерфейс и определит возможности плагинной системы. Если он окажется слишком общим, плагины писать будет очень неудобно, если слишком узким — плагины будут не очень богаты функциональностью. Потребуется описать не только методы, но и структуры данных, которыми система будет общаться с плагином. Менять раз созданный и опубликованный интерфейс, по которому уже начали писать плагины (особенно сторонние разработчики), может быть очень болезненно. Сам интерфейс и все нужные ему структуры данных оформляются в отдельную сборку (даже если там нет никакого содержательного кода).

- Дальше требуется разработать «ядро» плагинной системы — отдельную сборку, которая будет ссылаться на сборку с интерфейсом плагина, и отвечать за загрузку плагинов, их инициализацию и общение с ними всего остального кода системы.

- После этого можно реализовать набор плагинов, которые тоже будут ссылаться на сборку с интерфейсом плагина (поэтому нам и потребовалось вынести его в отдельную сборку, чтобы плагины не зависели от всей системы). Плагины как раз и будут реализовывать этот интерфейс.

Вот мини-пример интерфейса плагина (в отдельной сборке, обратите внимание — да, это будет целый проект в Visual Studio или Rider, где всего один файл и вот эти семь строчек):

```
namespace MyCoolSystem.SDK
{
    public interface IAddIn
    {
        string DoSomething(int x);
```

```
    }
}
```

А вот пара примеров реализации этого интерфейса мини-плагинами (тоже в отдельной сборке, а по-хорошему даже в отдельных сборках, по одной для каждого плагина):

```
using MyCoolSystem.SDK;

public sealed class AddInA : IAddIn
{
    public String DoSomething(int x)
    {
        return "AddInA: " + x.ToString();
    }
}

public sealed class AddInB : IAddIn
{
    public String DoSomething(int x)
    {
        return "AddInB: " + (x * 2).ToString();
    }
}
```

И, наконец, содержательный код — ядро системы:

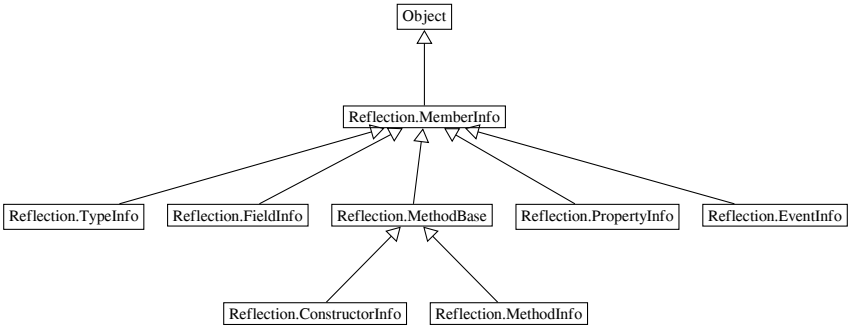
```
public static class Program
{
    public static void Main()
    {
        string addInDir = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);
        var addInAssemblies = Directory.EnumerateFiles(addInDir, "*.dll");
        var addInTypes =
            addInAssemblies.Select(Assembly.Load)
                .SelectMany(a => a.ExportedTypes)
                .Where(t => t.IsClass
                        && typeof(IAddIn).GetTypeInfo().IsAssignableFrom(t.GetTypeInfo()));

        foreach (Type t in addInTypes)
        {
            var addIn = (IAddIn)Activator.CreateInstance(t);
            Console.WriteLine(addIn.DoSomething(5));
        }
    }
}
```

`Assembly.GetEntryAssembly().Location` вернёт нам папку, из которой было запущено наше приложение, там мы и будем искать плагины.

`Directory.EnumerateFiles(addInDir, "*.dll");` вернёт нам всё .dll-ки, что есть в этой папке (может быть плохой идеей, если само наше приложение собрано как .dll). Дальше с помощью LINQ для каждого имени сборки из списка файлов применяем Assembly.Load, что вернёт нам список объектов типа Assembly. Дальше из каждой сборки мы достаём её ExportedTypes и сваливаем их всех в один список методом SelectMany. Дальше мы бежим по этому списку и выбираем из него только классы и только такие, которые совместимы по присваиванию (IsAssignableFrom) с интерфейсом IAddIn (то есть реализуют IAddIn, на самом деле). Ну и дельше в цикле for мы уже создаём экземпляры плагинов, кастаем их к типу интерфейса и вызываем их методы.

3. Информация о членах класса

Более подроббную информацию о методах, полях, свойствах и т.д. можно получить с помощью классов-наследников Reflection.MemberInfo:



- TypeInfo — это хранилище информации о типе, доступной рефлексии. Объект типа TypeInfo можно получить по Type. Type — это объект, представляющий тип, в нём хранятся статические поля, ссылки на код методов и т.п., TypeInfo — это объект, представляющий информацию о типе. Они взаимосвязаны, но всё-таки разные вещи. Кстати, чтобы ещё больше всё запутать, TypeInfo наследуется от Type.
- FieldInfo — информация о полях класса (имя, тип, модификаторы видимости и т.д.);
- ConstructorInfo — информация о конструкторах (параметры, модификаторы видимости и т.д.);
- MethodInfo — информация о методах (по сути то же, что информация о конструкторах, но ещё имя метода);
- PropertyInfo и EventInfo — информация о свойствах и событиях соответственно.

Например, вот так можно распечатать информацию обо всех полях и методах всех классов всех сборок, загруженных сейчас .NET-машиной (в AppDomain, на самом деле, но обычно AppDomain один):

```
using System;
using System.Reflection;

public static class Program
{
    public static void Main()
    {
        Assembly[] assemblies = AppDomain.CurrentDomain.GetAssemblies();
        foreach (Assembly a in assemblies)
        {
            Console.WriteLine($"Assembly: {a}");
            foreach (Type t in a.ExportedTypes)
            {
                Console.WriteLine($"  Type: {t}");
                foreach (MemberInfo mi in t.GetTypeInfo().DeclaredMembers)
                {
                    var typeName = mi switch
                    {
                        FieldInfo _ => "FieldInfo",
                        MethodInfo _ => "MethodInfo",
                        ConstructorInfo _ => "ConstructorInfo",
                        _ => ""
                    };
                    Console.WriteLine($"    {typeName}: {mi}");
                }
            }
        }
    }
}
```

Какие полезные свойства есть у любого подвида MemberInfo:

- Name (string) — имя члена класса (поля, метода и т.д.);
- DeclaringType (Type) — тип, в котором данный член класса объявлен;
- Module (Module) — модуль, в котором он объявлен;
- CustomAttributes (IEnumerable<CustomAttributeData>) — коллекция *атрибутов*, соответствующих этому члену класса. Про то, как объявлять и использовать свои атрибуты, чуть-чуть позже, но все ими пользовались и так. Например, когда писали юнит-тесты — система модульного тестирования ищет методы у классов, помеченные атрибутом Test, и рефлексией запускает их.

Однако всё, что мы рассматривали до этого, позволяет лишь получить информацию о членах. На самом деле основная польза рефлексии в том, что она позволяет и манипулировать объектами, причём делать с ними то, что даже среда времени выполнения делать

не позволит. Например, можно менять значения полей: у FieldInfo и PropertyInfo есть методы GetValue и SetValue. Причём в большинстве случаев так можно получить доступ и к private-полям. Тут вы, наверное, подумаете, что это же отличный способ писать юнит-тесты — выполняете действие и рефлексией ковыряетесь в тестируемом объекте, проверяя результаты. Не делайте этого! Это, возможно, будет работать, но при малейшем изменении тестируемого класса тесты начнут падать в рантайме.

Для вызова методов есть уже знакомый нам метод Invoke, работающий не только для ConstructorInfo, но и для MethodInfo. И методы подписывания/отписывания для событий: AddEventHandler и RemoveEventHandler у EventInfo.

Например, создадим объект какого-то пользовательского типа с помощью конструктора с одним параметром и вызовем его метод:

```
using System;
using System.Reflection;
using System.Linq;

internal sealed class SomeType
{
    public SomeType(int test) { }
    private int DoSomething(int x) => x * 2;
}

public static class Program
{
    public static void Main()
    {
        Type t = typeof(SomeType);
        Type ctorArgument = Type.GetType("System.Int32");
        ConstructorInfo ctor = t.GetTypeInfo().DeclaredConstructors.First(
                c => c.GetParameters()[0].ParameterType == ctorArgument);
        Object[] args = { 12 };
        Object obj = ctor.Invoke(args);
        MethodInfo mi = obj.GetType().GetTypeInfo().GetDeclaredMethod("DoSomething");
        int result = (int)mi.Invoke(obj, new object[]{3});
        Console.WriteLine($"result = {result}");
    }
}
```

DeclaredConstructors возвращает нам список всех конструкторов, объявленных в нашем классе, среди них мы находим тот, что имеет тип System.Int32, формируем ему массив аргументов args (из одного аргумента, но всё равно массив), и вызываем его Invoke. Получаем объект obj с типом времени выполнения SomeType. Дальше находим его метод DoSomething и вызваем, передавая число 3 как параметр. Приводим возвращаемое им значение к int (Invoke, как обычно, вернёт Object) и печатаем на экран.

Всё это делает в точности то же самое, что и код

```
using System;
```

```
internal sealed class SomeType
{
    public SomeType(int test) { }
    private int DoSomething(int x) => x * 2;
}

public static class Program
{
    public static void Main()
    {
        var someType = new SomeType(12);
        int result = someType.DoSomething(3);
        Console.WriteLine($"result = {result}");
    }
}
```

Возможности рефлексии как-то не очень впечатляют в этом примере (специально чтобы её не рекламировать, рефлексия всё-таки «чёрная магия»). Но надо учесть, что SomeType вполне может быть объявлен в другой сборке и недоступен во время компиляции, так что второй подход было бы не реализовать никак, а первый (с рефлексией) прекрасно будет работать.

4. Атрибуты

Атрибуты — это по сути механизм, позволяющий добавлять к коду произвольную информацию, которая доступна рефлексии и даже во время компиляции. Информацию эту можно понимать как пометки на абстрактном синтаксическом дереве вашей программы. Пример атрибута, используемого самим компилятором — `[Obsolete]`, который заставит его сгенерировать предупреждение, если вы используете помеченный им класс или метод, либо атрибуты сборки, типа AssemblyVersion, которые генеривались в файл Properties.cs в старом .NET Framework. Или `[CallerMemberName]`, атрибут, который можно нацепить на строковый параметр метода, и он будет инициализироваться именем метода, который нас вызвал. Примеры атрибутов, доступных внешним инструментам после компиляции — Test, TestData, Before, After и т.д.

К одному классу или методу может быть применено сразу несколько атрибутов (например, TestData обычно много). Вообще, атрибуты могут быть применены к сборке, типу, полю, методу, параметру метода, возвращаемому значению, свойству, событию, параметру-типу. Атрибуты могут иметь параметры (как, опять-таки, подсказывает опыт использования библиотек модульного тестирования).

И, страшная правда про атрибуты — они на самом деле объекты, экземпляры классов-наследников класса System.Attribute. И когда мы пишем `[Test]`, на самом деле во время компиляции вызывается конструктор класса TestAttribute, создаётся объект, объект сериализуется и то, что получилось, кладётся в таблицу с метаинформацией сборки. Так что атрибуты могут иметь конструктор с парамерами, свойства, теоретически даже методы, но

без идей, зачем они могут быть нужны, их ведь никак кроме как рефлексией и не вызвать. Суффикс Attribute по соглашению должен быть у всех классов-атрибутов, но компилятор разрешает при использовании атрибута его не указывать.

Вот так можно объявить собственный атрибут[5]:

```
public enum Animal
{
    Dog = 1,
    Cat,
    Bird,
}

public class AnimalTypeAttribute : Attribute
{
    public AnimalTypeAttribute(Animal pet)
    {
        this.Pet = pet;
    }

    public Animal Pet { get; set; }
}
```

На что обратить внимание:

- атрибут должен наследоваться от System.Attribute;
- атрибут должен иметь public-конструктор, с параметрами или без;
- атрибут может иметь как свойства только значения элементарных типов, перечислений, строк, массивов, типа Type.

Как можно «навесить» на код наш объявленный только что атрибут:

```
class AnimalTypeTestClass
{
    [AnimalType(Animal.Dog)]
    public void DogMethod() { }

    [AnimalType(Animal.Cat)]
    public void CatMethod() { }

    [AnimalType(Animal.Bird)]
    public void BirdMethod() { }
}
```

И как можно потом, после компиляции этого кода, достать атрибуты рефлексией:

[5] Пример из MSDN

```
static void Main(string[] args)
{
    var testClass = new AnimalTypeTestClass();
    Type type = testClass.GetType();

    foreach (MethodInfo mInfo in type.GetMethods())
    {
        foreach (Attribute attr in Attribute.GetCustomAttributes(mInfo))
        {
            // Check for the AnimalType attribute.
            if (attr.GetType() == typeof(AnimalTypeAttribute))
                Console.WriteLine(
                    "Method {0} has a pet {1} attribute.",
                    mInfo.Name, ((AnimalTypeAttribute)attr).Pet);
        }
    }
}
```

Attribute.GetCustomAttributes, применённый к MethodInfo, возвращает нам список атрибутов, «навешенных» на данный метод. Пройдясь по этому списку (который у нас будет состоять всего из одного атрибута для каждого метода), мы можем проверить тип атрибута через `attr.GetType() == typeof(AnimalTypeAttribute)`, но тут есть одна ловушка, в которую часто попадаются даже опытные программисты — чтобы это сравнение сработало, надо, чтобы тип атрибута на методе был тем же самым типом, что и тип атрибута, который вернёт typeof. Если мы анализируем сборку, которую мы сами же загрузили рефлексией, она может ссылаться не на те атрибуты (например, из сборки другой версии), и, хоть имена у них будут совпадать, это будут разные объекты Type. То же может произойти, если мы рефлексией загрузили сами себя — нове атрибуты перекроют старые, хоть это одни и те же атрибуты, и объекты Type для них будут другие.

А ещё можно сказать компилятору, что ваш атрибут применим только конкретно к полю, конкретно к методу, конкретно к параметру или к какой-то комбинации этих сущностей (чтобы не навешивать атрибут Test на параметры-типы генерика, например). Это делается при объявлении атрибута с помощью атрибута AttributeUsage, например, так:

```
namespace System
{
    [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
    public class FlagsAttribute : System.Attribute
    {
        public FlagsAttribute()
        {
        }
    }
}
```

Тут написано, что атрибут FlagsAttribute применим только к enum-ам и не наследуется потомками. Был бы он Inherited = true, все наследники получали бы атрибут тоже, даже ес-

ли бы явно его не прописывали (кстати, по умолчанию он true). Да, атрибуты используются при объявлении атрибутов, чего такого.

5. Пример: библиотеки сериализации

Пример библиотек, активно применяющих рефлексию — это библиотеки сериализации, с которыми мы сталкивались на лекции про работу с веб-сервисами. Сериализация вообще — это перевод объекта в массив байт, пригодный для дальнейшего сохранения или передачи. Это нетривиальная задача, потому что объекты часто ссылаются на другие объекты по их адресам в памяти, которые оказываются бесполезны в другом адресном пространстве (например, при загрузке объекта в память с диска), так что все адреса при сериализации приходится заменять на некоторые идентификаторы, а при десериализации заменять обратно. А поскольку между объектами бывают круговые зависимости, при сериализации надо уметь аккуратно обходить граф объектов в самом общем виде.

Объекты часто сериализуются в документы в человекочитаемом формате, что облегчает отладку и редактирование того, что получилось. Самые популярные ныне форматы — это JSON и XML. Бывают и бинарные форматы сериализации, которые не читаемы человеком, зато гораздо компактнее — например, Google Protobuf, который активно используется в веб-сервисах от Google.

5.1. System.Xml.Serialization

Сериализация в XML есть в стандартной библиотеке .NET, все соответствующие классы находятся в пространстве имён System.Xml.Serialization, причём пользоваться ей очень просто, так что она хорошо подходит как быстрое решение, если нет причин поступать иначе. Рассмотрим для примера данные, типичные для картинки в векторном графическом редакторе:

```
public class Point
{
    public Point() { }
    public Point(int x, int y) { this.X = x; this.Y = y; }
    public int X { get; set; }
    public int Y { get; set; }
}

public class SomeData
{
    public int Radius { get; set; }
    public Point Center { get; set; }
    public List<Point> Points { get; } = new List<Point>();
}
```

Сериализовать в XML их можно следующим образом:

```
var data = new SomeData { Radius = 10, Center = new Point(5, 5)};
data.Points.Add(new Point(1, 1));
```

```
data.Points.Add(new Point(2, 2));

var serializer = new XmlSerializer(typeof(SomeData));
using (var textWriter = new StringWriter())
{
    serializer.Serialize(textWriter, data);
    Console.WriteLine(textWriter.ToString());
}
```

Конструктор XmlSerializer принимает объект типа Type, который библиотека будет использовать, чтобы получать информацию о полях и свойствах сериализуемого объекта, а метод Serialize — сам объект и поток (или TextWriter), куда нужно вывести данные (то есть сериализовывать можно прямо в файл, сеть и т.д., не держа сериализуемые данные в памяти). В нашем примере получится вот что:

```
<?xml version="1.0" encoding="utf-16"?>
<SomeData xmlns:xsi="..." xmlns:xsd="...">
    <Radius>10</Radius>
    <Center>
        <X>5</X>
        <Y>5</Y>
    </Center>
    <Points>
        <Point>
            <X>1</X>
            <Y>1</Y>
        </Point>
        <Point>
            <X>2</X>
            <Y>2</Y>
        </Point>
    </Points>
</SomeData>
```

Видно, что свойствва стали XML-ными тэгами, а список — тэгом с несколькими дочерними тэгами одного типа. Десериализовать это можно методом Deserialize у того же XmlSerializer.

5.2. Newtonsoft.Json

А вот как сериализовать те же данные в JSON, с помощью библиотеки Newtonsoft.Json, одной из самых используемых библиотек под .NET вообще.

```
var data = new SomeData { Radius = 10, Center = new Point(5, 5)};
data.Points.Add(new Point(1, 1));
data.Points.Add(new Point(2, 2));
```

```
var jsonSerializer = new JsonSerializer() { Formatting = Formatting.Indented };
using (var textWriter = new StringWriter())
{
    jsonSerializer.Serialize(textWriter, data);
    Console.WriteLine(textWriter.ToString());
}
```

Как видим, принципиально это то же самое, разве что JsonSerializer-у не надо передавать тип сериализуемого объекта (он поймёт его сам по параметру Serialize — десериализация тут выполняется другим классом, так что заранее информация о типе JsonSerializer не нужна). Formatting.Indented просит поставить отступы в документе, иначе всё сериализуется в одну строку (что хорошо для передачи по сети, но плохо для чтения человеком). Вот что получится:

```
{
  "Radius": 10,
  "Center": {
    "X": 5,
    "Y": 5
  },
  "Points": [
    {
      "X": 1,
      "Y": 1
    },
    {
      "X": 2,
      "Y": 2
    }
  ]
}
```

Как видим, JSON-представление гораздо компактней, поэтому и более популярно нынче, чем XML.

Однако дьявол кроется в деталях. Пока у нас объекты образуют дерево, всё сериализуется хорошо, но давайте усложним задачу:

```
var center = new Point(5, 5);
var data = new SomeData { Radius = 10, Center = center };
data.Points.Add(new Point(1, 1));
data.Points.Add(center);
```

Теперь у нас один и тот же объект используется в двух местах структуры объектов, и если его сериализовать кодом выше, то библиотека сериализует его как два *разных* Point-а. И при десериализации получатся два разных объекта. Чтобы таких дел не было, надо просто попросить библиотеку сохранить ссылки на объекты:

```
var jsonSerializer = new JsonSerializer() {
    Formatting = Formatting.Indented,
    PreserveReferencesHandling = PreserveReferencesHandling.All
};
using (var textWriter = new StringWriter())
{
    jsonSerializer.Serialize(textWriter, data);
    Console.WriteLine(textWriter.ToString());
}
```

Получится вот такое:

```
{
  "$id": "1",
  "Radius": 10,
  "Center": {
    "$id": "2",
    "X": 5,
    "Y": 5
  },
  "Points": [
    {
      "$id": "3",
      "X": 1,
      "Y": 1
    },
    {
      "$ref": "2"
    }
  ]
}
```

$id — это уникальный в рамках документа идентификатор объекта, он генерится для всех сериализуемых объектов вообще. $ref — это, как нетрудно догадаться, ссылка на объект, который встречался ранее. Итого, когда библиотека первый раз посещает какой-то объект, она его сериализует честно и кладёт в таблицу. Когда посещает его второй раз, достаёт его id-шник из таблицы и делает ref. Прямо всегда использовать PreserveReferencesHandling не стоит, потому что это увеличивает размер документа, лучше продумывать структуру сериализуемых данных так, чтобы этого не требовалось. Нет ничего плохого в том, чтобы сериализовать не прямо объекты с бизнес-логикой, а сначала конвертить их в удобное для сериализации промежуточное представление (больше работы, но может быть лучше результат; а может и нет).

Десериализовать структуру объектов можно вот так:

```
var deserializedData =
    JsonConvert.DeserializeObject<SomeData>(serializedData);
Console.WriteLine(deserializedData.Radius);
```

Тут уже надо передать DeserializeObject тип десериализуемого объекта (как в System.XML.Serialization), но тут тип передаётся как параметр-тип в генерик, а не как объект типа Type. Кажется, что так даже удобнее.

Библиотеки сериализации также активно используют и атрибуты для управления тем, что во что должно сериализоваться, и должно ли вообще. Например, вот так можно управлять именами полей в сериализованном документе[6]:

```
public class Videogame
{
    [JsonProperty("name")]
    public string Name { get; set; }

    [JsonProperty("release_date")]
    public DateTime ReleaseDate { get; set; }
}
```

Это необходимо, когда приходится работать с JSON-документом, где поля называются не так, как хочет ваш стайлгайд от свойств в C#-коде.

А вот как управлять тем, что должно быть сериализуемо, а что нет:

```
[JsonObject(MemberSerialization.OptIn)]
public class File
{
    // Это поле не сериализуется,
    // потому что у него нет JsonPropertyAttribute
    public Guid Id { get; set; }

    [JsonProperty]
    public string Name { get; set; }

    [JsonProperty]
    public int Size { get; set; }
}
```

Тут мы атрибутом `JsonObject(MemberSerialization.OptIn)` говорим, что хотим сериализовать только те поля, которые мы явно сказали сериализовать, и атрибутом JsonProperty без параметров говорим, какие именно поля сериализовать. Id, допустим, мы хотим генерить всё время разный, так что его решили не сериализовать, ну он в сериализованном документе и не появится.

6. Ключевое слово dynamic

Теперь немного относящихся к рефлексии тайных знаний, которыми никогда-никогда не надо пользоваться. Первое такое знание — это возможность попросить компилятор игнорировать тип времени компиляции для объекта, используя так называемую «утиную

[6] Пример, разумеется, из документации на https://www.newtonsoft.com

типизацию» (Duck typing) — если оно ходит как утка и крякает как утка, то оно и есть утка. Пример:

```
using System;

internal static class DynamicDemo
{
    public static void Main()
    {
        dynamic value;
        for (int demo = 0; demo < 2; demo++)
        {
            value = (demo == 0) ? (dynamic)5 : (dynamic)"A";
            value = value + value;
            M(value);
        }
    }

    private static void M(int n) { Console.WriteLine("M(int): " + n); }
    private static void M(string s) { Console.WriteLine("M(string): " + s); }
}
```

Тут пееменная value объявляется с ключевым словом dynamic, которое говорит, что в value, как в JavaScript или Python, может быть значение любого типа. Ну и в цикле мы сначала кладём в value число, затем строку. Магия рефлексии делает так, что, во-первых, перегруженные операторы вызываются от типа времени выполнения переменной, то есть на первой итерации в value положится 10, на второй — “AA”, хоть это и совершенно разные операции. Более того, будет вызвана правильная перегрузка метода M.

Казалось бы, очень удобно — давайте все переменные объявлять как dynamic и никогда не получать ошибок компиляции. Но это **очень плохая** идея. Вы как бы говорите, что давайте выкинем всю систему типов языка, на разработку которой и её поддержки в компиляторе потратили кучу денег и сил, и будем надеяться, что у нас все типы во время выполнения будут такими, как надо. Если нет, программа во время выполнения упадёт, и компилятор ничем не сможет даже подсказать. Ну и, хоть это и не очень принципиально, работают dynamic-переменные не очень быстро.

Обратите внимание, dynamic и var — это совершенно разные вещи. var говрит, что тип времени компиляции у переменной есть, и компилятору надо самому его вывести из контекста, а затем и проверить. dynamic говорит, что типа времени компиляции у переменной нет и никаких проверок делать не надо. Вообще, dynamic создавалась для облегчения взаимодействия с языками, где утиная типизация — единственный доступный вид типизации, например, JavaScript. Если вы из C#-кода вызовете JS-код и он вам что-то вернёт, вы и правда не имеете идей, что за тип у результата. Можно с ним работать как с Object, но как с dynamic с ним работать гораздо удобнее (например, применять оператор сложения можно почти ко всему). Во всех других случаях использовать dynamic не стоит.

7. Генерация кода «на лету»

Ну и последний кусок «чёрной магии» на сегодня — это мрачный класс ILGenerator из запретного пространства имён System.Reflection.Emit. Это штука, которая позволяет относительно удобно генерировать байт-код .NET-машины (тот самый CIL) прямо в процессе выполнения программы. А благодаря наличию перегрузки Assembly.Load, принимающей поток байт, его можно тут же загрузить и исполнить. А теперь представим себе, что сгенерённый нами код сам генерит и грузит похожий код, и умеет слегка изменяться от поколения к поколению. И что он убежал в интернет...

Идея полиморфного кода (то есть кода, который может менять сам себя в процессе работы) была популярна годах в 60-х 20-го века, даже у Кнута в книге его машина использовала самомодифицирующийся код для реализации части операций (например, редактировала адрес возврата из функции при вызове). Однако люди быстро осознали, что отлаживать такой код невозможно, поэтому нынче полиморфный код используется только в исключительных случаях (например, в обфускаторах, где «отлаживать невозможно» как раз желаемое свойство). Но System.Reflection.Emit и класс ILGenerator вполне применимы на практике — особенно если вы пишете свой компилятор с .NET-машиной в качестве рантайма. А поскольку на матмехе любят компиляторы, и .NET — вполне достойный выбор в качестве бэкенда и среды исполнения, то смотрите:

```
public static void Main() {
    AssemblyName assemblyName = new AssemblyName {Name = "HelloEmit"};
    AppDomain appDomain = AppDomain.CurrentDomain;
    AssemblyBuilder assemblyBuilder = appDomain.DefineDynamicAssembly(
        assemblyName, AssemblyBuilderAccess.Save);
    ModuleBuilder moduleBuilder =
        assemblyBuilder.DefineDynamicModule(assemblyName.Name, "Hello.exe");
    TypeBuilder typeBuilder = moduleBuilder.DefineType("Test.MainClass",
        TypeAttributes.Public | TypeAttributes.Class);
    MethodBuilder methodBuilder = typeBuilder.DefineMethod("Main",
        MethodAttributes.Public | MethodAttributes.Static,
        typeof(int), new[] { typeof(string[]) });

    ILGenerator ilGenerator = methodBuilder.GetILGenerator();
    ilGenerator.Emit(OpCodes.Ldstr, "Hello, World!");
    ilGenerator.Emit(OpCodes.Call,
        typeof(Console).GetMethod("WriteLine", new[] { typeof(string) }));
    ilGenerator.Emit(OpCodes.Ldc_I4_0);
    ilGenerator.Emit(OpCodes.Ret);

    typeBuilder.CreateType();
    assemblyBuilder.SetEntryPoint(methodBuilder, PEFileKinds.ConsoleApplication);
    assemblyBuilder.Save("Hello.exe");
}
```

Это пример генерации вполне работоспособной сборки, в которой только главный класс с Main, печатающим «Hello, World!» (ну а что, `int zero = 0;` по-умному мы

уже писали). Сначала создаётся объект-сборка с именем HelloEmit, затем создаётся AssemblyBuilder, из которого мы получаем ModuleBuilder, который будет строить модуль Hello.exe (паттерн «Строитель»). В этом модуле мы определяем TypeBuilder, который строит класс Test.MainClass, в нём MethodBuilder для строительства метода Main (обратите внимание, мы передаём все модификаторы при создании билдера), из этого билдера берём ILGenerator и им печатаем байт-код.

Собственно, байт-код грузит на стек строку “Hello, World!” (все же помнят, что .NET внутри — это стековая машина, все вычисления там — это операции над вершиной стека). Дальше генерируется вызов метода WriteLine у типа Console, дальше на тек грузится число 0 (это код возврата) и генерится инструкция Ret — return, которая и заканчивает программу.

Дальше мы методом CreateType получаем тип с помощью всех этих билдеров, комадой SetEntryPoint говорим, что точка входа в сборку — это метод Main, а тип приложения — консольное, и сохраняем всё в файл Hello.exe. Потом этот файл можно запустить и он даже будет работать.