**Сериализация объектов**

Понятие атрибутов.

В C# разрешается вводить в программу информацию декларативного характера в форме атрибута, с помощью которого определяются дополнительные сведения (метаданные), связанные с классом, структурой, методом и т.д. Например, в программе можно указать атрибут, определяющий тип кнопки, которую должен отображать конкретный класс. Атрибуты указываются в квадратных скобках перед тем элементом, к которому они применяются. Следовательно, атрибут не является членом класса, но обозначает дополнительную информацию, присоединяемую к элементу.

Одной из задач компилятора .NET является генерирование описаний метаданных для всех типов, которые были определены, и на которые имеется ссылка в текущем проекте. Помимо стандартных метаданных, которые помещаются в любую сборку, программисты в .NET могут включать в состав сборки дополнительные метаданные с использованием атрибутов. В сущности, атрибуты представляют собой не более чем просто аннотации, которые могут добавляться в код и применяться к какому-то конкретному типу (классу, интерфейсу, структуре и т.д.), члену (свойству, методу и т.д.), сборке или модулю.

Концепция аннотирования кода с применением атрибутов является далеко не новой. Еще в COM IDL (Interface Definition Language — язык описания интерфейсов) поставлялось множество предопределенных атрибутов, которые позволяли разработчикам описывать типы, содержащиеся внутри того или иного СОМ-сервера. Однако в СОМ атрибуты представляли собой немногим более чем просто набор ключевых слов. Когда требовалось создать специальный атрибут, разработчик в СОМ мог делать это, но затем он должен был ссылаться на этот атрибут в коде с помощью 128-битного числа (GUID-идентификатора), что, как минимум, довольно затрудняло дело.

В .NET атрибуты представляют собой типы классов, которые расширяют абстрактный базовый **класс System.Attribute**. В поставляемых в .NET пространствах имен доступно множество предопределенных атрибутов, которые полезно применять в своих приложениях. Более того, можно также создавать собственные атрибуты и тем самым дополнительно уточнять поведение своих типов, создавая для атрибута новый тип, унаследованный от Attribute.

Ниже перечислены некоторые из предопределенных атрибутов, предлагаемые в различных пространствах имен .NET:

**[CLSCompliant]**

Заставляет элемент, к которому применяется, отвечать требованиям CLS (Common Language Specification — общеязыковая спецификация). Вспомните, что типы, которые отвечают требованиям CLS, могут без проблем использоваться во всех языках программирования .NET

**[DllImport]**

Позволяет коду .NET отправлять вызовы в любую неуправляемую библиотеку кода на С или С++, в том числе и API-интерфейс лежащей в основе операционной системы. Обратите внимание, что при взаимодействии с программным обеспечением, работающим на базе СОМ, этот атрибут не применяется

**[Obsolete]**

Позволяет указать, что данный тип или член является устаревшим. Когда другие программисты попытаются использовать элемент с таким атрибутом, они получат соответствующее предупреждение от компилятора

**[Serializable]**

Позволяет указать, что класс или структура является "сериализируемой", т.е. способна сохранять свое текущее состояние в потоке

**[NonSerialized]**

Позволяет указать, что данное поле в классе или структуре не должно сохраняться в процессе сериализации

**[WebMethod]**

Позволяет указать, что метод может вызываться посредством HTTP-запросов, и CLR-среда должна преобразовывать его возвращаемое значение в формат XML

Важно уяснить, что при применении атрибутов в коде размещающиеся внутри них метаданные, по сути, остаются бесполезными до тех пор, пока не запрашиваются явным образом в каком-то другом компоненте программного обеспечения посредством рефлексии. Если этого не происходит, они спокойно игнорируются и не причиняют никакого вреда.

Потребители атрибутов

Как нетрудно догадаться, в составе NET Framework 4.0 SDK поставляется множество утилит, которые позволяют производить поиск разнообразных атрибутов. Даже сам компилятор C# (csc.ехе) запрограммирован так, что он проверяет наличие разных атрибутов во время компиляции. Например, сталкиваясь с атрибутом [CLSCompilant], он автоматически проверяет соответствующий элемент и удостоверяется в том, что в нем содержатся только отвечающие требованиям CLS инструкции, а при обнаружении элемента с атрибутом [Obsolete] отображает внутри окна ErrorList (Список ошибок) в Visual Studio 2010 соответствующее предупреждение.

Помимо утилит, предназначенных для использования во время разработки, многие методы в библиотеках базовых классов .NET тоже изначально запрограммированы так, чтобы распознавать определенные атрибуты посредством рефлексии. Например, чтобы информация о состоянии объекта сохранялась в файле, все, что потребуется делать — это просто добавить в класс или структуру в виде аннотации атрибут [Serializable]. Встретив этот атрибут, метод Serialize() из класса BinaryFormatter автоматически сохраняет соответствующий объект в файле в компактном двоичном формате.

CLR-среда в .NET тоже выполняет проверки на предмет наличия определенных атрибутов. Самым известным из них, пожалуй, является *атрибут [WebMethod]*, который применяется для создания веб-служб XML с помощью ASP.NET. Чтобы можно было получать доступ к методу посредством HTTP-запросов, а его возвращаемое значение автоматически преобразовывалось в формат XML, понадобится применить к этому методу атрибут [WebMethod], а обо всех остальных деталях будет заботиться CLR-среда. Помимо разработки веб-служб, атрибуты также играют важную роль в функционировании системы безопасности .NET, в Windows Communication Foundation, в обеспечении функциональной совместимости между СОМ и .NET, и во многом другом.

И, наконец, допускается разрабатывать приложения, способные распознавать специальные атрибуты, а также любые из тех, что поставляются в составе библиотек базовых классов .NET. Это позволяет, по сути, создавать набор "ключевых слов", понятных только определенному множеству сборок.

## Атрибут Serializable

Чтобы посмотреть, как применение атрибутов в C# выглядит на практике, создадим новое консольное приложение. Предположим, что требуется создать класс по имени Motorcycle (мотоцикл), который бы мог сохраняться в двоичном формате. Для этого достаточно применить к определению этого класса атрибут [Serializable]. Если в классе есть какое-то поле, которое не должно сохраняться, к нему должен быть применен атрибут [NonSerialized]:

// Этот класс может сохраняться на диске.

[Serializable]

public class Motorcycle

{

// Это поле сохраняться не будет

[NonSerialized]

float weightOfСurrentPassengers;

// Эти поля будут сериализованы.

bool hasRadioSystem;

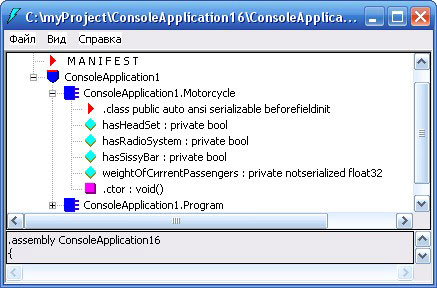
bool hasHeadSet;

bool hasSissyBar;

...

Действие любого атрибута распространяется только на элемент, который следует сразу же после него. Например, в классе Motorcycle сериализации не будет подвергнуто только поле weightOfCurrentPassengers. Все остальные поля будут обязательно сериализоваться, потому что класс снабжен атрибутом [Serializable].

После компиляции этого класса можно просмотреть дополнительные метаданные с помощью утилиты ildasm.ехе. При этом следует обратить внимание на то, что эти атрибуты будут сопровождаться маркером serializable (в строке с треугольником красного цвета внутри класса Motorcycle) и маркером notserialized (в строке, представляющей поле weightOfCurrentPassengers), как показано на рисунке:



К каждому элементу может применяться сразу несколько атрибутов.

Что такое сериализация?

Термин ***сериализация*** описывает процесс сохранения (и, возможно, передачи) состояния объекта в потоке (т.е. файловом потоке и потоке в памяти). Последовательность сохраняемых данных содержит всю информацию, необходимую для реконструкции (или десериализации) состояния объекта с целью последующего использования. Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями.

Во многих случаях сохранение данных приложения с использованием служб сериализации выливается в код меньшего объема, чем применение классов для чтения/записи из пространства имен System.IO.

Например, предположим, что создано настольное приложение с графическим интерфейсом, в котором необходимо предоставить конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.п.). Для этого можно определить класс по имени UserPrefs и инкапсулировать в нем примерно два десятка полей данных. В случае применения типа System.IO.BinaryWriter придется вручную сохранять каждое поле объекта UserPrefs.

Аналогично, когда вам понадобится загрузить данные из файла обратно в память, придется использовать SystemIO.BinaryReader и, опять-таки, вручную читать каждое значение, чтобы реконструировать новый объект UserPrefs.

Сэкономить значительное время можно, снабдив класс UserPrefs атрибутом **[Serializable]**:

[Serializable]

public class UserPrefs

{

public string WindowColor;

public int FontSize;

}

После этого полное состояние объекта может быть сохранено с помощью всего нескольких строк кода. Пока не погружаясь в детали, взгляните на следующий метод Main():

static void Main(string[] args)

{

UserPrefs userData = new UserPrefs();

userData.WindowColor = "Yellow";

userData.FontSize = 50;

// BinaryFormatter сохраняет данные в двоичном формате. Чтобы получить доступ к BinaryFormatter, понадобится

// импортировать System.Runtime.Serialization.Formatters.Binary

BinaryFormatter binFormat = new BinaryFormatter();

// Сохранить объект в локальном файле.

using(Stream fStream = new FileStream("user.dat",

FileMode.Create, FileAccess.Write, FileShare.None))

{

binFormat.Serialize(fStream, userData);

}

}

Хотя сохранять объекты с помощью механизма сериализации объектов .NET довольно просто, процесс, происходящий при этом "за кулисами", достаточно сложен. Например, когда объект сохраняется в потоке, все ассоциированные с ним данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализуются. Поэтому, при попытке сериализовать производный класс в игру вступают также все данные по цепочке наследования. И как будет показано, набор взаимосвязанных объектов, участвующих в этом, представляется **графом объектов**.

Службы сериализации .NET также позволяют сохранять граф объектов в различных форматах. В предыдущем примере кода применялся тип BinaryFormatter, поэтому состояние объекта UserPrefs сохраняется в компактном двоичном формате. Граф объектов можно также сохранить в формате SOAP или XML, используя другие типы форматеров. Эти форматы полезны, когда необходимо гарантировать возможность передачи хранимых объектов между разными операционными системами, языками и архитектурами.

В WCF предлагается слегка отличающийся механизм для сериализации объектов в и из операций службы WCF в нем используются атрибуты **[DataContract]** и **[DataMember]**.

И, наконец, имейте в виду, что граф объектов может быть сохранен в любом типе, унаследованном от System.IO.Stream. В предыдущем примере объект UserPrefs был сохранен в локальном файле через тип FileStream. Однако если вместо этого понадобится сохранить объект в определенной области памяти, можно применить тип MemoryStream. Главное, чтобы последовательность данных корректно представляла состояние объектов в графе.

Отношения между объектами.

Прежде чем приступить к изучению основных паттернов также рассмотрим основные отношения между объектами, которые помогут нам понять связи между сущностями при их использовании в паттернах. Мы можем выделить несколько основных отношений: наследование, реализация, ассоциация, композиция и агрегация.

### Наследование

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение **IS A**, то есть "является". Например:

class User

{

    public int Id { get; set; }

    public string Name { get; set; }

}

class Manager : User

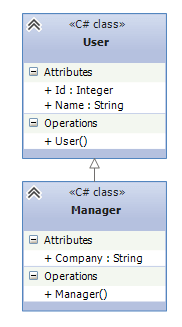
{

    public string Company{ get; set; }

}

В данном случае используется наследование, а объекты класса Manager также **являются** и объектами класса User.

С помощью диаграмм UML отношение между классами выражается в незакрашенной стрелочке от класса-наследника к классу-родителю:



### Реализация

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

public interface IMovable

{

    void Move();

}

public class Car : IMovable

{

    public void Move()

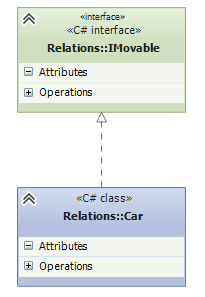
    {

        Console.WriteLine("Машина едет");

    }

}

С помощью диаграмм UML отношение реализации также выражается в незакрашенной стрелочке от класса к интерфейсу, только линия теперь пунктирная:



### Ассоциация

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

class Team

{

}

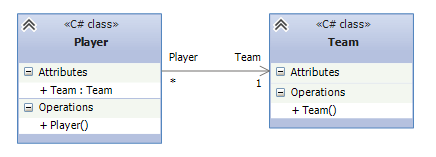
class Player

{

    public Team Team { get; set; }

}

Класс Player связан отношением ассоциации с класом Team. На схемах UML ассоциация обозначается в виде обычно стрелки:



Нередко при отношении ассоциации указывается кратность связей. В данном случае единица у Team и звездочка у Player на диаграмме отражает связь 1 ко многим. То есть одна команда будет соответствовать многим игрокам.

Агрегация и композиция являются частными случаями ассоциации.

### Композиция

Композиция определяет отношение **HAS A**, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя:

public class ElectricEngine

{ }

public class Car

{

    ElectricEngine engine;

    public Car()

    {

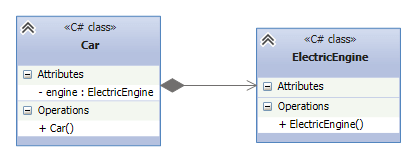
        engine = new ElectricEngine();

    }

}

При этом класс автомобиля полностью управляет жизненным циклом объекта двигателя. При уничтожении объекта автомобиля в области памяти вместе с ним будет уничтожен и объект двигателя. И в этом плане объект автомобиля является главным, а объект двигателя - зависимой.

На диаграммах UML отношение композиции проявляется в обычной стрелке от главной сущности к зависимой, при этом со стороны главной сущности, которая содержит, объект второй сущности, располагается закрашенный ромбик:



### Агрегация

От композиции следует отличать агрегацию. Она также предполагает отношение **HAS A**, но реализуется она иначе:

public abstract class Engine

{ }

public class Car

{

    Engine engine;

    public Car(Engine eng)

    {

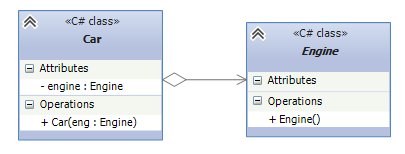
        engine = eng;

    }

}

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Отношение агрегации на диаграммах UML отображается также, как и отношение композиции, только теперь ромбик будет незакрашенным:



При проектировании отношений между классами надо учитывать некоторые общие рекомендации. В частности, вместо наследования следует предпочитать композицию. При наследовании весь функционал класса-наследника жестко определен на этапе компиляции. И во время выполнения программы мы не можем его динамически переопределить. А класс-наследник не всегда может переопределить код, который определен в родительском классе. Композиция же позволяет динамически определять поведение объекта во время выполнения, и поэтому является более гибкой.

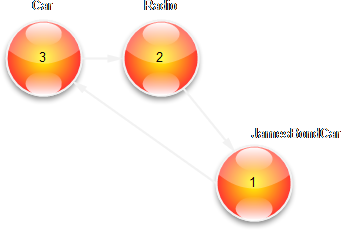
Вместо композиции следует предпочитать агрегацию, как более гибкий способ связи компонентов. В то же время не всегда агрегация уместна. Например, у нас есть класс человека, который содержит объект нервной системы. Понятно, что в реальности, по крайней мере на текущий момент, невозможно вовне определить нервную систему и внедрить ее в человека. То есть в данном случае человек будет главным компонентом, а нервная система - зависимым, подчиненным, и их создание и жизненный цикл будет происходить совместно, поэтому здесь лучше выбрать композицию.

Графы отношений объектов.

Как упоминалось ранее, когда объект сериализуется, среда CLR учитывает все связанные объекты, чтобы гарантировать корректное сохранение данных. Этот набор связанных объектов называется графом объектов. Графы объектов представляют простой способ документирования набора отношений между объектами, и эти отношения не обязательно отображаются на классические отношения ООП (вроде отношений "является" и "имеет"), хотя достаточно хорошо моделируют эту парадигму.

Каждый объект в графе получает уникальное числовое значение. Имейте в виду, что числа, назначенные объектам в графе, являются произвольными и не имеют никакого значения для внешнего мира. Как только каждому объекту присвоено числовое значение, граф объектов может записать все наборы зависимостей каждого объекта.

В качестве простого примера предположим, что создан набор классов, моделирующих автомобили. Существует базовый класс по имени Car, который "имеет" класс Radio. Другой класс по имени JamesBondCar расширяет базовый тип Car. На рисунке показан возможный граф объектов, который моделирует эти отношения:



При чтении графов объектов для описания соединяющих стрелок можно использовать выражение "зависит от" или "ссылается на". Таким образом, на рисунке видно, что класс Car ссылается на класс Radio (учитывая отношение "имеет"), JamesBondCar ссылается на Car (учитывая отношение "имеет"), как и на Radio (поскольку наследует эту защищенную переменную-член).

Конечно, среда CLR не рисует картинок в памяти для представления графа взаимосвязанных объектов. Вместо этого отношение, документированное в предыдущей диаграмме, представлено математической формулой, которая выглядит примерно так:

*[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]*

Если вы проанализируете эту формулу, то опять увидите, что объект 3 (Car) имеет зависимость от объекта 2 (Radio). Объект 2 (Radio) — это "одинокий волк", которому не нужен никто. И, наконец, объект 1 (JamesBondCar) имеет зависимость как от объекта 3, так и от объекта 2. В любом случае, при сериализации или десериализации JamesBondCar граф объектов гарантирует, что типы Radio и Car также будут участвовать в процессе.

Изящество процесса сериализации состоит в том, что граф, представляющий отношения между объектами, устанавливается автоматически, "за кулисами". Как будет показано далее, если необходимо вмешаться в конструирование графа объектов, это можно сделать посредством настройки процесса сериализации через атрибуты и интерфейсы.

Атрибуты для сериализации [Serializable] и [NonSerialized].

Чтобы сделать объект доступным для служб сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом [Serializable]. Если выясняется, что некоторый тип имеет члены-данные, которые не должны (или не могут) участвовать в схеме сериализации, можно пометить такие поля атрибутом [NonSerialized]. Это помогает сократить размер хранимых данных, при условии, что в сериализуемом классе есть переменные-члены, которые не следует "запоминать" (например, фиксированные значения, случайные значения, кратковременные данные и т.п.).

Для начала создадим новое консольное приложение. Добавим в него новый класс по имени Radio, помеченный атрибутом [Serializable], у которого исключается одна переменная-член (radioID), помеченная атрибутом [NonSerialized] и потому не сохраняемая в специфицированном потоке данных:

[Serializable]

public class Radio

{

public bool hasTweeters;

public bool hasSubWoofers;

public double[] stationPresets;

[NonSerialized]

public string radioID = "XF-552RF6";

}

Затем добавим два дополнительных типа, представляющих базовые классы JamesBondCar и Car (оба они также помечены атрибутом [Serializable]), и определим в них следующие поля данных:

[Serializable]

public class Car

{

public Radio theRadio = new Radio();

public bool isHatchBack;

}

[Serializable]

public class JamesBondClass : Car

{

public bool canFly;

public bool canSubmerge;

}

Атрибут [Serializable] не может наследоваться от родительского класса. Поэтому при наследовании типа, помеченного [Serializable], дочерний класс также должен быть помечен [Serializable] или же его нельзя будет сохранить в потоке. Фактически все объекты в графе объектов должны быть помечены атрибутом [Serializable]. Попытка сериализовать несериализуемый объект с использованием BinaryFormatter или SoapFormatter приводит к исключению SerializationException во время выполнения.

Обратите внимание, что в каждом из этих классов поля данных определены как public, это сделано для упрощения примера. Конечно, приватные данные, представленные общедоступными свойствами, были бы более предпочтительны с точки зрения ООП. Также для простоты в этих типах не определились никакие специальные конструкторы, и потому все неинициализированные поля данных получат ожидаемые значения по умолчанию.

Оставив в стороне принципы ООП, можно спросить: какого определения полей данных типа требуют различные форматеры, чтобы сериализовать их в поток? Ответ такой: в зависимости от обстоятельств. Если вы сохраняете состояние объекта, используя BinaryFormatter или SoapFormatter, то разницы никакой. Эти типы запрограммированы для сериализации всех сериализуемых полей типа, независимо от того, представлены они общедоступными полями, приватными полями или приватными полями с соответствующими общедоступными свойствами.

Однако вспомните, что если есть элементы данных, которые не должны сохраняться в графе объектов, можно выборочно пометить общедоступные или приватные поля атрибутом [NonSerialized], как сделано со строковыми полями в типе Radio.

Однако ситуация существенно меняется, если вы собираетесь применять тип XmlSerializer. Этот тип будет сериализовать только сериализуемые общедоступные поля данных или приватные поля, представленные общедоступными свойствами. Приватные данные, не представленные свойствами, просто игнорируются. Например, рассмотрим следующий сериализуемый тип Person:

[Serializable]

public class Person

{

// Общедоступное поле

public bool isAlive = true;

// Приватное поле

private int personAge = 21;

// Общедоступное свойство/приватные данные.

private string fName = string.Empty;

public string FirstName

{

get { return fName; }

set { fName = value; }

}

При обработке BinaryFormatter или SoapFormatter обнаружится, что поля isAlive, personAge и fName сохраняются в выбранном потоке. Однако XmlSerializer не сохранит значения personAge, поскольку эта часть приватных данных не инкапсулирована в свойство. Чтобы сохранять возраст персоны с помощью XmlSerializer, это поле понадобится определить как public или же инкапсулировать его в общедоступном свойстве.

Форматы сериализации.

*JSON vs XML*

[XML](http://www.script-coding.com/XML.html) (Extensible Markup Language) – расширяемый не зависящий от платформы язык разметки, который определяет набор правил для представления иерархических данных и унификации передаваемой информации формате, читаемом человеком и машиной. Язык описан в спецификации W3C XML 1.0 и ряде других спецификацией, с ней связанных.

[JSON](http://json.org/json-ru.html) (JavaScript Object Notation) — открытый формат стандарта, который использует человеко-читаемый текст для передачи объектов данных, состоящих из пары «атрибут-значение». Применяется, в основном, для передачи данных между сервером и веб-приложением, как альтернатива XML.

Так почему же стоит вопрос выбора языка разметки? Множество постов в сети описывают преимущества JSON над XML. Некоторые называют JSON «обезжиренной» версией XML из-за меньшего грамматического набора (т.е. не такой многословной). JSON также более точно соответствует структурам данных, используемых в современных языках программирования.

Как XML, так и JSON, используются при реализации [Ajax](http://www.ibm.com/developerworks/ru/library/wa-ajaxintro1/) подхода. Ajax описывает способность веб-страницы запрашивать новые данные после того, как она уже загружена в веб-браузер, обычно в ответ на действия пользователя на отображаемой веб-странице. Новые данные, как часть модели Ajax, обычно отражаются в пользовательском интерфейсе динамически, в момент получения этих данных от сервера. Примером реализации подхода может служить набор пользователем в строке поиска запроса, который отправляется клиентской частью программы на сервер, а в ответ из базы данных поступает список возможных вариантов. Список может быть отображен в виде ниспадающего меню под строкой поиска. Пользователь может остановить набор и выбрать непосредственно интересующую его строку. На начальных этапах в решениях Ajax в качестве формата обмена данными в основном использовался XML. Сейчас многие разработчики Ajax для обмена обновленными данными между клиентом и сервером используют JSON, т.к. он более быстрый и эффективный.

Как только типы сконфигурированы для участия в схеме сериализации .NET с применением необходимых атрибутов, следующий шаг состоит в выборе формата (двоичного, SOAP или XML) для сохранения состояния объектов. Перечисленные возможности представлены следующими классами:

* **BinaryFormatter**
* **SoapFormatter**
* **XmlSerializer**

Тип BinaryFormatter сериализует состояние объекта в поток, используя компактный двоичный формат. Этот тип определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, которое входит в сборку mscorlib.dll. Таким образом, чтобы получить доступ к этому типу, необходимо указать следующую директиву using:

// Получить доступ к BinaryFormatter в mscorlib.dll

using System.Runtime.Serialization.Formatters.Binary;

Тип SoapFormatter сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб). Этот тип определен в пространстве имен System.Runtime.Serialization.Formatters.Soap, находящемся в отдельной сборке. Поэтому для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на System.Runtime.Serialization.Formatters.Soap.dll, используя диалоговое окно Add Reference (Добавить ссылку) в Visual Studio 2010 и затем указать следующую директиву using:

// Необходима ссылка на System.Runtime.Serialization.Formatters.Soap.dll!

using System.Runtime.Serialization.Formatters.Soap;

И, наконец, для сохранения дерева объектов в документе XML имеется тип XmlSerializer. Чтобы использовать этот тип, нужно указать директиву using для пространства имен System.Xml.Serialization и установить ссылку на сборку System.Xml.dll. К счастью, шаблоны проектов Visual Studio 2010 автоматически ссылаются на System.Xml.dll, так что достаточно просто указать соответствующее пространство имен:

// Определено внутри System.Xml.dll

using System.Xml.Serialization;

Независимо от того, какой форматер выбран, имейте в виду, каждый из них наследуется непосредственно от System.Object, так что они не разделяют общего набора членов от какого-то базового класса сериализации. Однако типы BinaryFormatter и SoapFormatter поддерживают общие члены через реализацию интерфейсов IFormatter и IRemotingFormatter (как ни странно, XmlSerializer не реализует ни одного из них).

В System.Runtime.Serialization.IFormatter определены основные методы **Serialize()** и **Deserialize()**, которые выполняют черновую работу по перемещению графов объектов в определенный поток и обратно. Помимо этих членов в IFormatter определено несколько свойств, используемых "за кулисами" реализующим типом:

public interface IFormatter

{

SerializationBinder Binder { get; set; }

StreamingContext Context { get; set; }

ISurrogateSelector SurrogateSelector { get; set; }

object Deserialize(Stream serializationStream);

void Serialize(Stream serializationStream, object graph);

}

Интерфейс System.Runtime.Remoting.Messaging.IRemotingFormatter (который внутри полагается на уровень удаленного взаимодействия .NET Remoting) перегружает члены Serialize() и Deserialize() в манере, более подходящей для распределенного сохранения. Обратите внимание, что интерфейс IRemotingFormatter унаследован от более общего интерфейса IFormatter:

public interface IRemotingFormatter : IFormatter

{

object Deserialize(Stream serializationStream, HeaderHandler handler);

void Serialize(Stream serializationStream, object graph,

Header[] headers);

}

Хотя взаимодействовать с этими интерфейсами не понадобится в большинстве сценариев сериализации, вспомните, что полиморфизм на базе интерфейсов позволяет подставлять экземпляры BinaryFormatter или SoapFormatter там, где ожидается IFormatter.

## Точность типов среди форматеров

Наиболее очевидное отличие между тремя форматерами связано с тем, как граф объектов сохраняется в потоке (двоичном, SOAP или XML). Следует знать также о некоторых более тонких отличиях, в частности — каким образом форматеры добиваются ***точности типов (type fidelity)***.

Когда используется тип BinaryFormatter, он сохраняет не только данные полей объектов из графа, но также полное квалифицированное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер общедоступного ключа и культура). Эти дополнительные элементы данных делают BinaryFormatter идеальным выбором, когда необходимо передавать объекты по значению (т.е. полные копии) между границами машин для использования в .NET-приложениях.

Форматер SoapFormatter сохраняет трассировки сборок-источников за счет использования пространства имен XML. Например, вспомните тип Person, определенный в предыдущей статье. Если понадобится сохранить этот тип в сообщении SOAP, вы обнаружите, что открывающий элемент Person квалифицирован сгенерированным параметром xmlns. Взгляните на следующее частичное определение, обратив особое внимание на пространство имен XML под названием a1:

<a1:Person id="ref-1" xmlns:a1=

"http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20

Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">

<isAlive>true</isAlive>

<personAge>21</personAge>

<fName id="ref-3"></fName>

</a1:Person>

Однако XmlSerializer не пытается предохранить точную информацию о типе, и потому не записывает его полного квалифицированного имени или сборки, в которой он определен. Хотя на первый взгляд это может показаться ограничением, причина состоит в открытой природе представления данных XML. Ниже показано возможное XML-представление типа Person:

<?xml version="1.0"?>

<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:xsd="http://www.w3.org/2 001/XMLSchema">

<isAlive>true</isAlive>

<PersonAge>2l</PersonAge>

<FirstName />

</Person>

Если необходимо сохранить состояние объекта так, чтобы его можно было использовать в любой операционной системе (Windows ХР, Mac OS X и различных дистрибутивах Linux), на любой платформе приложений (.NET, Java Enterprise Edition, COM и т.п.) или в любом языке программирования, придерживаться полной точности типов не следует, поскольку нельзя рассчитывать, что все возможные адресаты смогут понять специфичные для .NET типы данных. Учитывая это, SoapFormatter и XmlSerializer являются идеальным выбором, когда требуется гарантировать как можно более широкое распространение объектов.

# BinaryFormatter и SoapFormatter

Чтобы проиллюстрировать, насколько просто сохранить экземпляр JamesBondCar (пример из предыдущих статей) в физическом файле, воспользуемся типом BinaryFormatter. Двумя ключевыми методами типа BinaryFormatter, о которых следует знать, являются Serialize() и Deserialize():

* Serialize() сохраняет граф объектов в указанный поток в виде последовательности байтов;
* Deserialize() преобразует сохраненную последовательность байт в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить этот экземпляр в файле \*.dat. Первая задача — создание самого файла \*.dat. Для этого можно создать экземпляр типа System.IO.FileStream. Затем следует создать экземпляр BinaryFormatter и передать ему FileStream и граф объектов для сохранения. Взгляните на следующий метод Main():

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace Serialize

{

class Program

{

static void Main()

{

JamesBondClass jbc = new JamesBondClass();

jbc.canFly = true;

jbc.canSubmerge = false;

jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };

jbc.theRadio.hasTweeters = true;

// Сохранить объект в указанном файле в двоичном формате

SaveBinaryFormat(jbc, "carData.dat");

Console.ReadLine();

}

static void SaveBinaryFormat(object objGraph, string fileName)

{

BinaryFormatter binFormat = new BinaryFormatter();

using (Stream fStream = new FileStream(fileName, FileMode.Create, FileAccess.Write, FileShare.None))

{

binFormat.Serialize(fStream, objGraph);

}

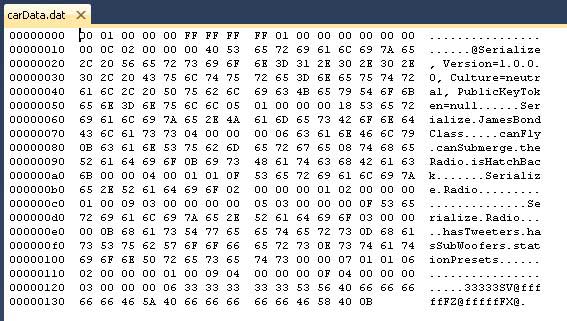
Console.WriteLine("--> Сохранение объекта в Binary format");

}

}

метод BinaryFormatter.Serialize() — это член, отвечающий за составление графа объектов и передачу последовательности байт в некоторый объект унаследованного от Stream типа. В данном случае поток представляет физический файл. Однако можно также сериализовать объекты в любой тип-наследник Stream, представляющий область памяти, сетевой поток и т.п.

После выполнения программы можно просмотреть содержимое файла carData.dat, представляющее этот экземпляр JamesBondCar, в папке bin\Debug текущего проекта. На рисунке показан этот файл, открытый в Visual Studio:



Теперь предположим, что необходимо прочитать сохраненный объект JamesBondCar из двоичного файла обратно в объектную переменную. После открытия файла carData.dat (методом File.OpenRead()) просто вызовите метод Deserialize() класса BinaryFormatter. Имейте в виду, что Deserialize() возвращает объект общего типа System.Object, так что понадобится применить явное приведение, как показано ниже:

static void LoadFromBinaryFile(string fileName)

{

BinaryFormatter binFormat = new BinaryFormatter();

using (Stream fStream = File.OpenRead(fileName))

{

JamesBondClass carFromDisk =

(JamesBondClass)binFormat.Deserialize(fStream);

Console.WriteLine(carFromDisk.canFly);

}

}

Обратите внимание, что при вызове Deserialize() ему передается тип-наследник Stream, представляющий местоположение сохраненного графа объектов. Приведя возвращенный объект к правильному типу, вы получаете объект в том состоянии, в каком он был на момент сохранения.

Следующий форматер, которым мы воспользуемся, будет SoapFormatter, сериализующий данные в подходящем конверте SOAP. Протокол ***SOAP (Simple Object Access Protocol — простой протокол доступа к объектам)*** определяет стандартный процесс вызова методов в независящей от платформы и операционной системы манере.

Предполагая, что ссылка на сборку System.Runtime.Serialization.Formatters.Soap.dll установлена, а пространство имен System.Runtime.Serialization.Formatters.Soap импортировано, для сохранения и извлечения JamesBondCar в виде сообщения SOAP можно просто заменить в предыдущем примере все вхождения BinaryFormatter на SoapFormatter. Ниже показан новый метод класса Program, который сериализует объект в локальный файл:

static void SoapWriteFile(object objGraph, string fileName)

{

SoapFormatter soapFormatter = new SoapFormatter();

using (Stream fStream = new FileStream(fileName,

FileMode.Create, FileAccess.Write, FileShare.None))

{

soapFormatter.Serialize(fStream, objGraph);

}

Console.WriteLine("--> Сохранение объекта в SOAP-формат");

}



# XmlSerializer

В дополнение к двоичному форматеру и форматеру SOAP сборка System.Xml.dll предлагает третий класс форматера — System.Xml.Serialization.XmlSerializer — который может использоваться для сохранения общедоступного состояния заданного объекта в виде чистой XML-разметки, в противоположность данным XML внутри сообщения SOAP. Работа с этим типом несколько отличается от работы с типами SoapFormatter или BinaryFormatter. Рассмотрим следующий код (предполагается, что было импортировано пространство имен System.Xml.Serialization):

static void SaveInXmlFormat(object objGraph, string fileName)

{

XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondClass),

new Type[] { typeof(Radio), typeof(Car) });

using (Stream fStream = new FileStream(fileName,

FileMode.Create, FileAccess.Write, FileShare.None))

{

xmlFormat.Serialize(fStream, objGraph);

}

Console.WriteLine("--> Сохранение объекта в XML-формат");

}

Ключевое отличие состоит в том, что тип XmlSerializer требует указания информации о типе, представляющей класс, который необходимо сериализовать. В сгенерированном файле XML находятся показанные ниже данные XML:

<?xml version="1.0"?>

<JamesBondClass xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<theRadio>

<hasTweeters>true</hasTweeters>

<hasSubWoofers>false</hasSubWoofers>

<stationPresets>

<double>89.3</double>

<double>105.1</double>

<double>97.1</double>

</stationPresets>

<radioID>XF-552RF6</radioID>

</theRadio>

<isHatchBack>false</isHatchBack>

<canFly>true</canFly>

<canSubmerge>false</canSubmerge>

</JamesBondClass>

Класс XmlSerializer требует, чтобы все сериализованные типы в графе объектов поддерживали конструктор по умолчанию (поэтому не забудьте его добавить, если определяли специальные конструкторы). Если этого не сделать, во время выполнения сгенерируется исключение InvalidOperationException.

Если у вас есть опыт в технологиях XML, то вы хорошо знаете, насколько важно удостоверяться, что данные внутри документа XML отвечают набору правил, которые устанавливают действительность данных. Понятие "действительного" документа XML не имеет отношения к синтаксической правильности элементов XML (вроде того, что все открывающие элементы должны иметь соответствующие закрывающие элементы).

Действительные документы — это те, что отвечают согласованным правилам форматирования (например, поле X должно быть выражено как атрибут, но не как подэлемент), которые обычно определены схемой XML или в **файле определения типа документа (Document-Type Definition — DTD)**.

По умолчанию класс XmlSerializer сериализует все общедоступные поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса XmlSerializer, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен System.Xml.Serialization. Ниже документированы некоторые (но не все) атрибуты, которые влияют на кодирование данных XML в потоке:

**[XmlAttribute]**

Поле или свойство будет сериализовано как атрибут XML (а не как подэлемент)

**[XmlElement]**

Поле или свойство будет сериализовано как элемент XML с указанным именем

**[XmlEnum]**

Этот атрибут предоставляет имя элемента, являющееся членом перечисления

**[XmlRoot]**

Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и название элемента)

**[XmlText]**

Свойство или поле должно быть сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)

**[XmlType]**

Этот атрибут предоставляет имя и пространство имен типа XML

Если необходимо указать специальное пространство имен XML, которое квалифицирует JamesBondCar и закодирует значения canFly и canSubmerge в виде атрибутов XML, модифицируем определение класса JamesBondCar следующим образом:

[XmlRoot(Namespace="http://professorweb.ru")]

public class JamesBondClass : Car

{

[XmlAttribute]

public bool canFly;

[XmlAttribute]

public bool canSubmerge;

}

Создание пользовательского формата сериализации. Интерфейс ISerializable.

Объекты, которые помечены атрибутом [Serializable], имеют опцию реализации интерфейса ISerializable. Реализация этого интерфейса позволяет вмешаться в процесс сериализации и выполнить необходимое форматирование данных до и после сериализации.

После выхода версии .NET 2.0 предпочтительный способ настройки процесса сериализации начал предусматривать использование атрибутов сериализации. Тем не менее, знание интерфейса ISerializable важно для сопровождения систем, которые уже существуют. Интерфейс ISerializable довольно прост, учитывая, что в нем определен единственный метод **GetObjectData()**.

Метод GetObjectData() вызывается автоматически заданным форматером во время процесса сериализации. Реализация этого метода заполняет входной параметр SerializationInfo последовательностью пар "имя/значение", которые (обычно) отображают данные полей сохраняемого объекта. В SerializationInfo определены многочисленные вариации перегруженного метода AddValue(), а также небольшой набор свойств, которые позволяют устанавливать и получать имя типа, определять сборку и счетчик членов.

Типы, реализующие интерфейс ISerializable, также должны определять специальный конструктор со следующей сигнатурой:

// Необходимо предусмотреть специальный конструктор с такой сигнатурой,

// чтобы позволить исполняющей среде устанавливать состояние объекта.

[Serializable]

class SomeClass : ISerializable

{

protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}

}

Обратите внимание, что видимость конструктора указана как protected. Это приемлемо, учитывая, что форматер будет иметь доступ к этому члену независимо от его видимости. Такие специальные конструкторы обычно делают protected (или private), тем самым гарантируя, что небрежный пользователь объекта никогда не создаст объект подобным образом. Как видите, первым параметром конструктора является экземпляр типа SerializationInfo.

Второй параметр этого специального конструктора имеет тип **StreamingContext** и содержит информацию относительно источника или места назначения передаваемых данных. Наиболее информативным членом StreamingContext является свойство State, представляющее значение из перечисления StreamingContextStates.

Значения этого перечисления представляют базовую композицию текущего потока. Если только не планируется реализовать какие-то низкоуровневые службы удаленного взаимодействия, то иметь дело с этим перечислением непосредственно требуется редко

В следующем примере кода показано использование интерфейса [ISerializable](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable?view=netcore-3.1) для определения пользовательского поведения сериализации для класса.

using System;

using System.Text;

using System.IO;

// Add references to Soap and Binary formatters.

using System.Runtime.Serialization.Formatters.Binary;

using System.Runtime.Serialization.Formatters.Soap ;

using System.Runtime.Serialization;

[Serializable]

public class MyItemType : ISerializable

{

public MyItemType()

{

// Empty constructor required to compile.

}

// The value to serialize.

private string myProperty\_value;

public string MyProperty

{

get { return myProperty\_value; }

set { myProperty\_value = value; }

}

// Implement this method to serialize data. The method is called

// on serialization.

public void GetObjectData(SerializationInfo info, StreamingContext context)

{

// Use the AddValue method to specify serialized values.

info.AddValue("props", myProperty\_value, typeof(string));

}

// The special constructor is used to deserialize values.

public MyItemType(SerializationInfo info, StreamingContext context)

{

// Reset the property value using the GetValue method.

myProperty\_value = (string) info.GetValue("props", typeof(string));

}

}

// This is a console application.

public static class Test

{

static void Main()

{

// This is the name of the file holding the data. You can use any file extension you like.

string fileName = "dataStuff.myData";

// Use a BinaryFormatter or SoapFormatter.

IFormatter formatter = new BinaryFormatter();

//IFormatter formatter = new SoapFormatter();

Test.SerializeItem(fileName, formatter); // Serialize an instance of the class.

Test.DeserializeItem(fileName, formatter); // Deserialize the instance.

Console.WriteLine("Done");

Console.ReadLine();

}

public static void SerializeItem(string fileName, IFormatter formatter)

{

// Create an instance of the type and serialize it.

MyItemType t = new MyItemType();

t.MyProperty = "Hello World";

FileStream s = new FileStream(fileName , FileMode.Create);

formatter.Serialize(s, t);

s.Close();

}

public static void DeserializeItem(string fileName, IFormatter formatter)

{

FileStream s = new FileStream(fileName, FileMode.Open);

MyItemType t = (MyItemType)formatter.Deserialize(s);

Console.WriteLine(t.MyProperty);

}

}

[Formatter](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.formatter?view=netcore-3.1) вызывает [GetObjectData](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable.getobjectdata?view=netcore-3.1) во время сериализации и заполняет переданный [SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=netcore-3.1) всеми данными, необходимыми для представления объекта. [Formatter](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.formatter?view=netcore-3.1) создает [SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=netcore-3.1) с типом объекта в графе. Объекты, которым необходимо отправлять собственные прокси-серверы, могут использовать методы [FullTypeName](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo.fulltypename?view=netcore-3.1) и [AssemblyName](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo.assemblyname?view=netcore-3.1) в [SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=netcore-3.1) для изменения передаваемых данных.

В случае наследования класса можно сериализовать класс, производный от базового класса, реализующего [ISerializable](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable?view=netcore-3.1). В этом случае производный класс должен вызвать реализацию базового класса [GetObjectData](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable.getobjectdata?view=netcore-3.1) в своей реализации [GetObjectData](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable.getobjectdata?view=netcore-3.1). В противном случае данные из базового класса не будут сериализованы.

Интерфейс [ISerializable](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable?view=netcore-3.1) подразумевает конструктор с конструктором сигнатуры ([SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=netcore-3.1) сведения, [StreamingContext](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.streamingcontext?view=netcore-3.1) контекст). Во время десериализации текущий конструктор вызывается только после того, как данные в [SerializationInfo](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.serializationinfo?view=netcore-3.1) десериализованы модулем форматирования. В общем случае этот конструктор должен быть защищен, если класс не запечатан.

Порядок, в котором выполняется десериализация объектов, не может быть гарантирован. Например, если один тип ссылается на тип, который еще не был десериализован, возникнет исключение. При создании типов с такими зависимостями можно обойти эту проблему, реализовав интерфейс IDeserializationCallback и метод OnDeserialization.

Архитектура сериализации обрабатывает типы объектов, которые расширяются [MarshalByRefObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.marshalbyrefobject?view=netcore-3.1) такими же, как типы, расширяющие [Object](https://docs.microsoft.com/ru-ru/dotnet/api/system.object?view=netcore-3.1). Эти типы можно пометить [SerializableAttribute](https://docs.microsoft.com/ru-ru/dotnet/api/system.serializableattribute?view=netcore-3.1) и реализовать интерфейс [ISerializable](https://docs.microsoft.com/ru-ru/dotnet/api/system.runtime.serialization.iserializable?view=netcore-3.1) как любой другой тип объекта. Их состояние объекта будет захвачено и сохранено в потоке.

Рефлексия

В мире .NET ***рефлексией (reflection)*** называется процесс обнаружения типов во время выполнения. С применением служб рефлексии те же самые метаданные, которые отображает утилита ildasm.exe, можно получать программно в виде удобной объектной модели. Например, рефлексия позволяет извлечь список всех типов, которые содержатся внутри определенной сборки \*.dll или \*.ехе (или даже внутри файла \*.netmodule если речь идет о многофайловой сборке), в том числе методов, полей, свойств и событий, определенных в каждом из них. Можно также динамически обнаруживать набор интерфейсов, которые поддерживаются данным типом, параметров, которые принимает данный метод, и других деталей подобного рода (таких как имена базовых классов, информация о пространствах имен, данные манифеста и т.д.).

Многие классы, поддерживающие рефлексию, входят в состав прикладного интерфейса .NET Reflection API, относящегося к пространству имен **System.Reflection**. Как и в любом другом пространстве имен, в System.Reflection (которое поставляется в составе сборки mscorlib.dll) содержится набор взаимосвязанных типов. Ниже описаны некоторые наиболее важные из этих типов:

**Assembly**

В этом абстрактном классе содержатся статические методы, которые позволяют загружать сборку, исследовать ее и производить с ней различные манипуляции

**AssemblyName**

Этот класс позволяет выяснить различные детали, связанные с идентификацией сборки (номер версии, информация о культуре и т.д.)

**EventInfо**

В этом абстрактном классе хранится информация о заданном событии

**FieldInfo**

В этом абстрактном классе хранится информация о заданном поле

**MemberInfo**

Этот абстрактный базовый класс определяет общее поведение для типов EventInfo, FieldInfo, MethodInfo и PropertyInfo

**MethodInfo**

В этом абстрактном классе содержится информация по заданному методу

**Module**

Этот абстрактный класс позволяет получить доступ к определенному модулю внутри многофайловой сборки

**ParameterInfо**

В этом классе хранится информация по заданному параметру

**PropertyInfо**

В этом абстрактном классе хранится информация по заданному свойству

Чтобы понять, каким образом использовать пространство имен System.Reflection для программного чтения метаданных .NET, необходимо сначала ознакомиться с классом System.Type.

Класс System.Type

Класс System.Type составляет ядро подсистемы рефлексии, поскольку он инкапсулирует тип данных. Он содержит многие свойства и методы, которыми можно пользоваться для получения информации о типе данных во время выполнения. Класс Type является производным от абстрактного класса *System.Reflection.MemberInfo*.

В классе MemberInfo определены приведенные ниже свойства, доступные только для чтения:

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Type DeclaringType | Тип класса или интерфейса, в котором объявляется отражаемый член |
| MemberType | Тип члена. Это значение обозначает, является ли член полем, методом, свойством, событием или конструктором |
| int MetadataToken | Значение, связанное к конкретными метаданными |
| Module | Объект типа Module, представляющий модуль (исполняемый файл), в котором находится отражаемый тип |
| string Name | Имя типа |
| Type ReflectedType | Тип отражаемого объекта |

Следует иметь в виду, что свойство MemberType возвращает тип MemberTypes — перечисление, в котором определяются значения, обозначающие различные типы членов. К их числу относятся следующие:

MemberTypes.Constructor

MemberTypes.Method

MemberTypes.Field

MemberTypes.Event

MemberTypes.Property

Следовательно, тип члена можно определить, проверив свойство MemberType. Так, если свойство MemberType имеет значение MemberTypes.Method, то проверяемый член является методом.

В класс MemberInfo входят два абстрактных метода: GetCustomAttributes () и IsDefined(). Оба метода связаны с атрибутами. Первый из них получает список специальных атрибутов, имеющих отношение к вызывающему объекту, а второй устанавливает, определен ли атрибут для вызывающего метода. В версию .NET Framework Version 4.0 внедрен метод GetCustomAttributesData(), возвращающий сведения о специальных атрибутах.

Класс System.Type имеет набор членов, которые могут применяться для изучения метаданных типа, и большинство из которых возвращает типы из пространства имен System.Reflection. Например, член Type.GetMethods() возвращает массив объектов типа MethodInfo, член Type.GetFields() — массив объектов типа FieldInfo. и т.д. Ниже приведен частичный список членов, которые поддерживает System.Type

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| IsAbstract  IsArray  IsClass  IsCOMObject  IsEnum  IsGenericTypeDefinition  IsGeneriсParameter  Islnterface  IsPrimitive  IsNestedPrivate  IsNestedPublic  IsSealed  IsValueType | Эти свойства позволяют выяснять ряд основных деталей об интересующем типе (например, является ли он абстрактной сущностью, массивом, вложенным классом и т.д.) |
| GetConstructors()  GetEvents()  GetFields()  GetInterfaces()  GetMembers()  GetMethods()  GetNestedTypes()  GetProperties() | Эти методы позволяют получать массив представляющих интерес элементов (интерфейсов, методов, свойств и т.д.). Каждый из этих методов возвращает соответствующий массив (например, GetFields() возвращает массив FieldInfo, GetMethods () — массив MethodInfo и тд.). Следует иметь в виду, что каждый из них имеет также форму единственного числа (например, GetMethod(), GetProperty() и т.п.), которая позволяет извлекать только один элемент по имени, а не целый массив подобных элементов |
| FindMembers() | Этот метод возвращает массив объектов типа MemberInfo на основе указанных критериев поиска |
| GetType() | Этот статический метод возвращает экземпляр Type, обладающий указанным строковым именем |
| InvokeMember() | Этот метод позволяет выполнять "позднее связывание" для заданного элемента |

# Рефлексия методов

С помощью методов и свойств класса Type можно получить подробные сведения о типе данных во время выполнения программы. Это довольно эффективное средство. Ведь получив сведения о типе данных, можно сразу же вызвать его конструкторы и методы или воспользоваться его свойствами. Следовательно, рефлексия позволяет использовать код, который не был доступен во время компиляции.

Прикладной **интерфейс Reflection API** весьма обширен и имеет ясную логическую структуру, поэтому, уяснив одну его часть, нетрудно понять и все остальное.

Чтобы проиллюстрировать базовый процесс рефлексии (и оценить пользу от System.Type), создадим новое консольное приложение. Это приложение будет отображать детали методов, свойств, полей и поддерживаемых интерфейсов (а также другие интересные элементы данных) для любого из типов, содержащихся как в самом приложении, так и в сборке mscorlib.dll (доступ к которой все приложения .NET получают автоматически). После создания соответствующего проекта первым делом необходимо импортировать пространство имен System.Reflection:

// Для выполнения рефлексии должно импортироваться это пространство имен

using System.Reflection;

Далее потребуется модифицировать класс Program, определив в нем ряд статических методов так, чтобы каждый из них принимал единственный параметр System.Type и возвращал void.

Имея в своем распоряжении объект класса Type, можно получить список методов, поддерживаемых отдельным типом данных, используя метод GetMethods(). Ниже приведена одна из форм, подходящих для этой цели:

Methodlnfo[] GetMethods()

Этот метод возвращает массив объектов класса MethodInfo, которые описывают методы, поддерживаемые вызывающим типом. Класс MethodInfo находится в пространстве имен System.Reflection.

Класс MethodInfo является производным от абстрактного класса MethodBase, который в свою очередь наследует от класса MemberInfo. Это дает возможность пользоваться всеми свойствами и методами, определенными в этих трех классах. Например, для получения имени метода служит свойство Name. Особый интерес вызывают два члена класса MethodInfo: ReturnType и GetParameters().

Возвращаемый тип метода находится в доступном только для чтения свойстве ReturnType, которое является объектом класса Type. Метод GetParameters() возвращает список параметров, связанных с анализируемым методом. Ниже приведена его общая форма:

ParameterInfo[] GetParameters();

Сведения о параметрах содержатся в объекте класса ParameterInfо. В классе ParameterInfо определено немало свойств и методов, описывающих параметры. Особое значение имеют два свойства: Name — представляет собой строку, содержащую имя параметра, a ParameterType — описывает тип параметра, который инкапсулирован в объекте класса Type.

Давайте рассмотрим пример:

using System;

using System.Reflection;

namespace Reflect

{

// Тестовый класс, содержащий некоторые конструкции

class MyTestClass

{

double d, f;

public MyTestClass(double d, double f)

{

this.d = d;

this.f = f;

}

public double Sum()

{

return d + f;

}

public void Info()

{

Console.WriteLine(@"d = {0}

f = {1}",d,f);

}

public void Set(int a, int b)

{

d = (double)a;

f = (double)b;

}

public void Set(double a, double b)

{

d = a;

f = b;

}

public override string ToString()

{

return "MyTestClass";

}

}

// В данном классе определены методы использующие рефлексию

class Reflect

{

// Данный метод выводит информацию о содержащихся в классе методах

public static void MethodReflectInfo<T> (T obj) where T: class

{

Type t = typeof(T);

// Получаем коллекцию методов

MethodInfo[] MArr = t.GetMethods();

Console.WriteLine("\*\*\* Список методов класса {0} \*\*\*\n",obj.ToString());

// Вывести методы

foreach (MethodInfo m in MArr)

{

Console.Write(" --> "+m.ReturnType.Name + " \t" + m.Name + "(");

// Вывести параметры методов

ParameterInfo[] p = m.GetParameters();

for (int i = 0; i < p.Length; i++)

{

Console.Write(p[i].ParameterType.Name + " " + p[i].Name);

if (i + 1 < p.Length) Console.Write(", ");

}

Console.Write(")\n");

}

}

}

class Program

{

static void Main()

{

MyTestClass mtc = new MyTestClass(12.0,3.5);

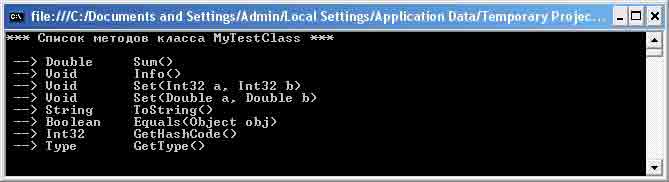
Reflect.MethodReflectInfo<MyTestClass>(mtc);

Console.ReadLine();

}

}

}



## Вторая форма метода GetMethods()

Существует вторая форма метода GetMethods(), позволяющая указывать различные флажки для отфильтровывания извлекаемых сведений о методах. Ниже приведена эта общая форма метода GetMethods():

Methodlnfo[] GetMethods(BindingFlags флажки)

В этом варианте создаются только те методы, которые соответствуют указанным критериям. BindingFlags представляет собой перечисление. Ниже перечислен ряд наиболее часто используемых его значений:

**DeclaredOnly**

Извлекаются только те методы, которые определены в заданном классе. Унаследованные методы в извлекаемые сведения не включаются

**Instance**

Извлекаются методы экземпляра

**Nonpublic**

Извлекаются методы, не являющиеся открытыми

**Public**

Извлекаются открытые методы

**Static**

Извлекаются статические методы

Два или несколько флажков можно объединить с помощью логической операции ИЛИ. Но как минимум флажок Instance или Static следует указывать вместе с флажком Public или NonPublic. В противном случае не будут извлечены сведения ни об одном из методов.

Форма BindingFlags метода GetMethods() чаще всего применяется для получения списка методов, определенных в классе, без дополнительного извлечения наследуемых методов. Это особенно удобно в тех случаях, когда требуется исключить получение сведений о методах, определяемых в классе конкретного объекта. В качестве примера попробуем выполнить следующую замену в вызове метода GetMethods() из предыдущей программы:

MethodInfo[] MArr = t.GetMethods(BindingFlags.DeclaredOnly |

BindingFlags.Instance | BindingFlags.Public);



Теперь выводятся только те методы, которые явно определены в классе MyTestClass.

# Рефлексия конструкторов, полей и свойств

Мы рассмотрели пример использования рефлексии методов, таким же образом можно получить информацию о полях, свойствах и интерфейсах класса с помощью методов GetField() и GetInterfaces. Давайте модифицируем программу из предыдущей статьи и добавим обобщенные методы в класс Reflect, отображающие информацию о полях, свойствах и интерфейсах, добавив реализацию интерфейса в класс MyTestClass:

...

interface IInfoClass

{

double Sum();

void Info();

void Set(double d1, double d2);

}

// Тестовый класс, содержащий некоторые конструкции

class MyTestClass : IInfoClass

{

...

// В данном классе определены методы использующие рефлексию

class Reflect

{

// Информация о полях и реализуемых интерфейсах

public static void FieldInterfaceInfo<T>(T obj) where T : class

{

Type t = typeof(T);

Console.WriteLine("\n\*\*\* Реализуемые интерфейсы \*\*\*\n");

var im = t.GetInterfaces();

foreach (Type tp in im)

Console.WriteLine("--> "+tp.Name);

Console.WriteLine("\n\*\*\* Поля и свойства \*\*\*\n");

FieldInfo[] fieldNames = t.GetFields();

foreach (FieldInfo fil in fieldNames)

Console.Write("--> "+fil.ReflectedType.Name + " " + fil.Name + "\n");

}

// Данный метод выводит информацию о содержащихся в классе методах

public static void MethodReflectInfo<T> (T obj) where T: class

{

Type t = typeof(T);

// Получаем коллекцию методов

MethodInfo[] MArr = t.GetMethods(BindingFlags.DeclaredOnly | BindingFlags.Instance | BindingFlags.Public);

Console.WriteLine("\*\*\* Список методов класса {0} \*\*\*\n",obj.ToString());

// Вывести методы

foreach (MethodInfo m in MArr)

{

Console.Write(" --> "+m.ReturnType.Name + " \t" + m.Name + "(");

// Вывести параметры методов

ParameterInfo[] p = m.GetParameters();

for (int i = 0; i < p.Length; i++)

{

Console.Write(p[i].ParameterType.Name + " " + p[i].Name);

if (i + 1 < p.Length) Console.Write(", ");

}

Console.Write(")\n");

}

}

}

class Program

{

static void Main()

{

MyTestClass mtc = new MyTestClass(12.0,3.5);

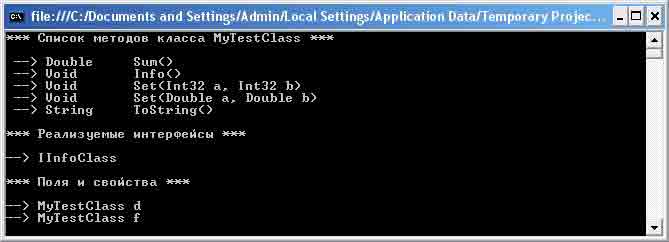
Reflect.MethodReflectInfo<MyTestClass>(mtc);

Reflect.FieldInterfaceInfo<MyTestClass>(mtc);

Console.ReadLine();

}

}



Следует иметь в виду, что большинство из "получающих", т.е. get-методов в System.Type (GetMethods(), GetInterfaces() и т.д.), имеют перегруженные версии, принимающие значения из перечисления BindingFlags. Это позволяет более точно указать, поиск чего должен производиться (например, только статических членов, только общедоступных членов, включая приватные члены, и т.д.).

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.

Конструкторы конкретного типа получаются при вызове метода GetConstructors() для объекта класса Type. Ниже приведена одна из наиболее часто используемых форм этого метода:

*ConstructorInfо[] GetConstructors()*

Метод GetConstructors() возвращает массив объектов класса ConstructorInfо, описывающих конструкторы. Класс ConstructorInfo является производным от абстрактного класса MethodBase, который в свою очередь наследует от класса MemberInfo. В нем также определен ряд собственных методов. К их числу относится интересующий нас метод GetConstructors(), возвращающий список параметров, связанных с конструктором. Этот метод действует таким же образом, как и упоминавшийся ранее метод GetParameters(), определенный в классе MethodInfo.

Как только будет обнаружен подходящий конструктор, для создания объекта вызывается метод Invoke(), определенный в классе ConstructorInfo. Ниже приведена одна из форм этого метода:

*object Invoke(object[] parameters)*

Любые аргументы, которые требуется передать методу, указываются в массиве parameters. Если же аргументы не нужны, то вместо массива parameters указывается пустое значение (null). Но в любом случае количество элементов массива parameters должно совпадать с количеством передаваемых аргументов, а типы аргументов — с типами параметров. Метод Invoke() возвращает ссылку на сконструированный объект.

# Рефлексия обобщенных типов

При вызове Type.GetType() для получения описаний метаданных обобщенных типов должен обязательно применяться специальный синтаксис в виде символа обратной одинарной кавычки (') со следующим за ним числовым значением, которое представляет количество параметров, поддерживаемое данным типом. Например, чтобы отобразить описание метаданных обобщенного типа System.Collections.Generic.List<T>, приложению потребуется передать следующую строку:

System.Collections.Generic.List '1

Здесь используется числовое значение 1, поскольку List<T> имеет только один параметр. Для применения рефлексии в отношении типа Dictionary<TKey, TValue>, однако, пришлось бы указать значение 2:

System.Collections.Generic.Dictionary '2

Сборка несет в себе сведения о типах классов, структур и прочих элементов данных, которые в ней содержатся. Прикладной интерфейс Reflection API позволяет загрузить сборку, извлечь сведения о ней и получить экземпляры объектов любых открыто доступных в ней типов. Используя этот механизм, программа может выявлять свою среду и использовать те функциональные возможности, которые могут оказаться доступными без явного их определения во время компиляции. Это очень эффективный и привлекательный принцип. Представьте себе, например, программу, которая выполняет роль "браузера типов", отображая типы данных, доступные в системе, или же инструментальное средство разработки, позволяющее визуально составлять программы из различных типов данных, поддерживаемых в системе. А поскольку все сведения о типах могут быть извлечены и проверены, то ограничений на применение рефлексии практически не существует.

Для получения сведений о сборке сначала необходимо создать объект **класса Assembly**. В классе Assembly открытый конструктор не определяется. Вместо этого объект класса Assembly получается в результате вызова одного из его методов. Так, для загрузки сборки по заданному ее имени служит метод LoadFrom(). Ниже приведена его соответствующая форма:

static Assembly LoadFrom(string файл\_сборки)

где файл\_сборки-обозначает конкретное имя файла сборки.

Как только будет получен объект класса Assembly, появится возможность обнаружить определенные в нем типы данных, вызвав для него метод GetTypes() в приведенной ниже общей форме:

Type[] GetTypes()

Этот метод возвращает массив типов, содержащихся в сборке.

Для того чтобы продемонстрировать порядок обнаружения типов в сборке, потребуются два исходных файла. Первый файл возьмем из первого раздела данного руководства, где мы создали библиотеку классов [fontinfo.dll](https://professorweb.ru/my/csharp/assembly/level1/1_8.php). Вы можете использовать какую то свою библиотеку классов.

using System;

using System.Reflection;

namespace Reflect

{

class Program

{

static void Main()

{

// Загружаем библиотеку классов

Assembly asm = Assembly.LoadFrom("C:\\myProject\\FontInfo\\fontinfo\\fontinfo.dll");

// Находим типы содержащиеся в fontinfo.dll

Type[] alltypes = asm.GetTypes();

Console.WriteLine("\*\*\* Найденые типы \*\*\*\n");

foreach (Type t in alltypes)

Console.WriteLine("-> "+t.Name);

Console.WriteLine();

// Выбираем последний класс

Type temp = alltypes[alltypes.Length-1];

Console.WriteLine("Используем класс "+temp.Name + "\n");

// Отображаем сведения о конструкторах

Console.WriteLine("\*\*\* Конструкторы \*\*\*\n");

ConstructorInfo[] ci = temp.GetConstructors();

foreach (ConstructorInfo c in ci)

{

Console.Write("-> " + c.Name + "(");

// Выводим параметры

ParameterInfo[] p = c.GetParameters();

for (int i = 0; i < p.Length; i++)

{

Console.Write(p[i].ParameterType.Name + " " + p[i].Name);

if (i + 1 < p.Length) Console.Write(", ");

}

Console.Write(")\n\n");

}

Console.ReadLine();

}

}

}

