Несмотря на всю свою полезность, операторы as и is проверяют лишь совместимость

двух типов. Но зачастую требуется информация о самом типе. Для этой цели в

С# предусмотрен оператор typeof. Он извлекает объект класса System.Туре для заданного

типа. С помощью этого объекта можно определить характеристики конкретного

типа данных.

**Применение оператора typeof**

Ниже приведена общая форма оператора typeof:

typeof(тип)

где *тип* обозначает получаемый тип. Информация, описывающая тип, инкапсулируется

в возвращаемом объекте класса Туре.

Получив объект класса Туре для заданного типа, можно извлечь информацию

о нем, используя различные свойства, поля и методы, определенные в классе Туре.

Класс Туре довольно обширен и содержит немало членов, поэтому его рассмотрение

придется отложить до следующего раздела, посвященного рефлексии. Но в качестве

краткого введения в этот класс ниже приведена программа, в которой используются

три его свойства: ***FullName***, ***IsClass*** и ***IsAbstract***. Для получения полного имени

типа служит свойство FullName. Свойство IsClass возвращает логическое значение

true, если тип относится к классу. А свойство IsAbstract возвращает логическое значение

true, если класс является абстрактным.

(***glava17\_3***)

using System;

using System.IO;

class UseTypeOf

{

static void Main()

{

Type t = typeof(StreamReader);

Console.WriteLine(t.FullName);

if (t.IsClass) Console.WriteLine("Its a class.");

if (t.IsAbstract) Console.WriteLine("Class is abstract.");

else Console.WriteLine("It base class.");

}

}

В данной программе сначала извлекается объект класса Туре, описывающий тип

StreamReader. Затем выводится полное имя этого типа данных и определяется его

принадлежность к классу, а далее — к абстрактному или конкретному классу.

**Рефлексия**

Рефлексия — это средство, позволяющее получать сведения о типе данных. Термин

*рефлексия,* или отражение, происходит от принципа действия этого средства: объект

класса Туре отражает базовый тип, который он представляет. Для получения информации

о типе данных объекту класса Туре делаются запросы, а он возвращает (отражает)

обратно информацию, связанную с определяемым типом. Рефлексия является

эффективным механизмом, поскольку она позволяет выявлять и использовать возможности

типов данных, известные только во время выполнения.

Многие классы, поддерживающие рефлексию, входят в состав прикладного интерфейса

.NET Reflection API, относящегося к пространству имен System.Reflection.

Поэтому для применения рефлексии в код программы обычно вводится следующая

строка.

using System.Reflection;

**Класс System.Type - ядро подсистемы рефлексии**

Класс System.Туре составляет ядро подсистемы рефлексии, поскольку он инкапсулирует

тип данных. Он содержит многие свойства и методы, которыми можно

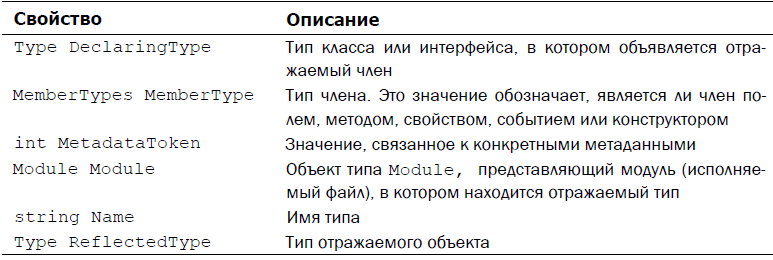
пользоваться для получения информации о типе данных во время выполнения.

Класс Туре является производным от абстрактного класса System.Reflection.

MemberInfo.

В классе MemberInfo определены приведенные ниже свойства, доступные только

для чтения.



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

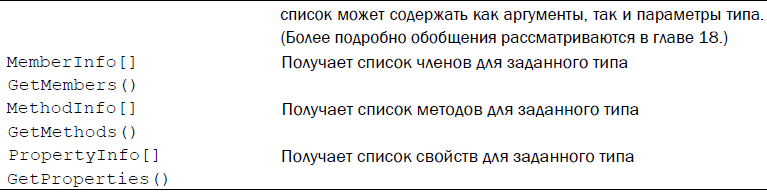
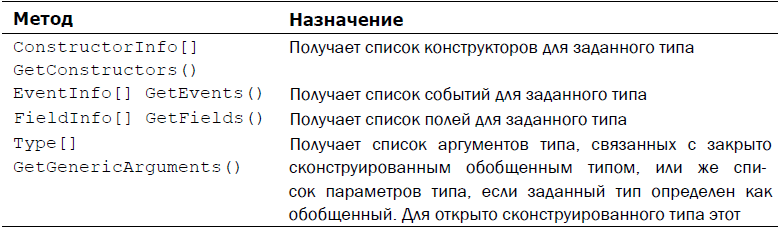
Version 4.0 внедрен метод GetCustomAttributesData(), возвращающий сведения

о специальных атрибутах. (Подробнее об атрибутах речь пойдет далее в этой главе.)

Класс Туре добавляет немало своих собственных методов и свойств к числу тех, что

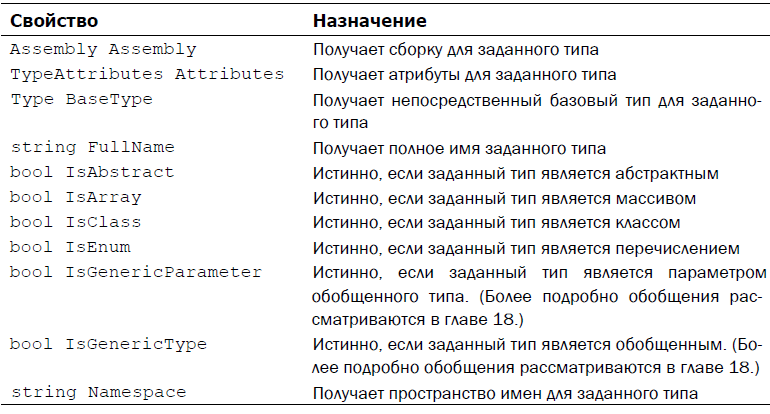
определены в классе MemberInfo. В качестве примера ниже перечислен ряд наиболее

часто используемых методов класса Туре.



Далее приведен ряд наиболее часто используемых свойств, доступных только для

чтения и определенных в классе Туре.



**Применение рефлексии**

С помощью методов и свойств класса Туре 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 поэтому не может быть

полностью рассмотрен в этой главе. Ведь для этого потребовалась бы целая книга! Но

прикладной интерфейс Reflection API имеет ясную логическую структуру, а следовательно,

уяснив одну его часть, нетрудно понять и все остальное. Принимая во внимание

это обстоятельство, в последующих разделах демонстрируются четыре основных

способа применения рефлексии: получение сведений о методах, вызов методов, конструирование

объектов и загрузка типов данных из сборок.

**Получение сведений о методах**

Имея в своем распоряжении объект класса Туре, можно получить список методов,

поддерживаемых отдельным типом данных, используя метод GetMethods().

MethodInfo[] 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 MemberInfо. 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 Туре.

Метод GetParameters() возвращает список параметров, связанных с анализируемым

методом. Ниже приведена его общая форма.

ParameterInfо[] 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 Туре.

В качестве примера ниже приведена программа, в которой рефлексия используется

для получения методов, поддерживаемых классом MyClass. В этой программе выводится

возвращаемый тип и имя каждого метода, а также имена и типы любых параметров,

которые может иметь каждый метод.

(***glava17\_4***)

using System.Reflection;

class MyClass

{

int x;

int y;

public MyClass(int i, int j)

{

x = i;

y = j;

}

public int Sum()

{

return x + y;

}

public bool IsBetween(int i)

{

if (x < i && i < y) return true;

else return false;

}

public void Set(int a, int b)

{

x = a;

y = b;

}

public void Set(double a, double b)

{

x = (int)a;

y = (int)b;

}

public void Show()

{

Console.WriteLine(" x: {0}, y: {1}", x, y);

}

}

class ReflectDemo

{

static void Main()

{

Type t = typeof(MyClass); //get object Type, with MyClass

Console.WriteLine("Analysing methods, defined in class " + t.Name);

Console.WriteLine();

Console.WriteLine("Supported methods.");

MethodInfo[] mi = t.GetMethods();

//show methods, defined in MyClass

foreach (MethodInfo m in mi)

{

//show return type and name of methods

Console.Write(" " + m.ReturnType.Name + " " + m.Name + "(");

//show parameters

ParameterInfo[] pi = m.GetParameters();

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

{

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

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

}

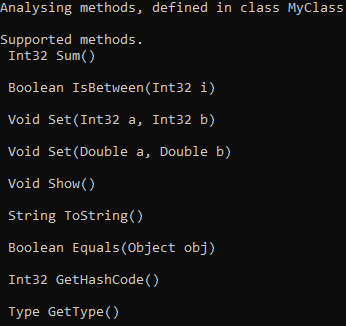
Console.WriteLine(")");

Console.WriteLine();

}

}

}



Как видите, помимо методов, определенных в классе MyClass, в данной программе

выводятся также методы, определенные в классе object, поскольку все типы данных в

C# наследуют от класса object. Кроме того, в качестве имен типов указываются имена

структуры .NET. Обратите также внимание на то, что метод Set() выводится дважды,

поскольку он перегружается. Один из его вариантов принимает аргументы типа int,

а другой — аргументы типа double.

Объект класса Туре, представляющий класс MyClass, создается в методе Main() в

следующей строке кода.

Type t = typeof(MyClass); //get object Type, with MyClass

Напомним, что оператор typeof возвращает объект класса Туре, представляющий

конкретный тип данных (в данном случае — класс MyClass).

С помощью переменной t и прикладного интерфейса Reflection API в данной

программе затем выводятся сведения о методах, поддерживаемых в классе MyClass.

Для этого в приведенной ниже строке кода сначала выводится список соответствующих

методов.

MethodInfo[] mi = t.GetMethods();

Затем в цикле foreach организуется обращение к элементам массива mi. На каждом

шаге этого цикла выводится возвращаемый тип, имя и параметры отдельного метода,

как показано в приведенном ниже фрагменте кода.

//show methods, defined in MyClass

foreach (MethodInfo m in mi)

{

//show return type and name of methods

Console.Write(" " + m.ReturnType.Name + " " + m.Name + "(");

//show parameters

ParameterInfo[] pi = m.GetParameters();

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

{

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

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

}

В этом фрагменте кода параметры, связанные с каждым методом, сначала создаются

с помощью метода GetParameters() и сохраняются в массиве pi. Затем в цикле

for происходит обращение к элементам массива pi и выводится тип и имя каждого

параметра. Самое главное, что все эти сведения создаются динамически во время

выполнения программы, не опираясь на предварительную осведомленность о классе

MyClass.

**Вторая форма метода GetMethods()**

Существует вторая форма метода GetMethods(), позволяющая указывать различные

флажки для отфильтровывания извлекаемых сведений о методах. Ниже приведена

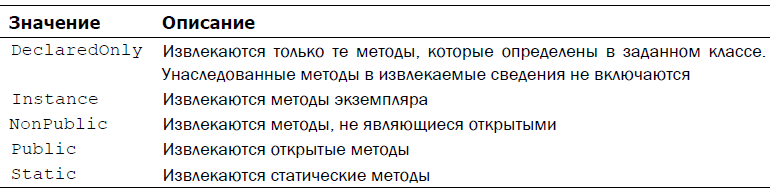
эта общая форма метода GetMethods().

MethodInfo[] 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.



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[] mi = t.GetMethods(BindingFlags.DeclaredOnly

| BindingFlags.Instance | BindingFlags.Public);

Как видите, теперь выводятся только те методы, которые явно определены в классе

MyClass.

**Вызов методов с помощью рефлексии**

Как только методы, поддерживаемые определенным типом данных, становятся известны,

их можно вызывать. Для этой цели служит метод Invoke(), входящий в состав

класса MethodInfo. Ниже приведена одна из форм этого метода:

object Invoke(object obj, object[] parameters)

где *obj* обозначает ссылку на объект, для которого вызывается метод. Для вызова статических

методов (static) в качестве параметра *obj* передается пустое значение (null).

Любые аргументы, которые должны быть переданы методу, указываются в массиве

*parameters.* Если же аргументы не нужны, то вместо массива *parameters* указывается

пустое значение (null). Кроме того, количество элементов массива *parameters*

должно точно соответствовать количеству передаваемых аргументов.

Для вызова конкретного метода достаточно вызвать метод Invoke() для экземпляра

объекта типа MethodInfo, получаемого при вызове метода GetMethods(). Эта

процедура демонстрируется в приведенном ниже примере программы.

(***glava17\_5***)

using System.Reflection;

class MyClass

{

int x;

int y;

public MyClass(int i, int j)

{

x = i;

y = j;

}

public int Sum()

{

return x + y;

}

public bool IsBetween(int i)

{

if ((x < i) && (i < y)) return true;

else return false;

}

public void Set(int a, int b)

{

Console.Write("In method Set(int, int). ");

x = a;

y = b;

Show();

}

//overload Set method

public void Set(double a, double b)

{

Console.Write("In method Set(double, double). ");

x = (int)a;

y = (int)b;

Show();

}

public void Show()

{

Console.WriteLine("Value x: {0}, value y: {1}", x, y);

}

}

class InvokeMethDemo

{

static void Main()

{

Type t = typeof(MyClass);

MyClass reflectOb = new MyClass(10, 20);

int val;

Console.WriteLine("Call methods, defined in " + t.Name);

Console.WriteLine();

MethodInfo[] mi = t.GetMethods();

//call each method

foreach(MethodInfo m in mi)

{

//get parameters

ParameterInfo[] pi = m.GetParameters();

if(m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(int))

{

object[] args = new object[2];

args[0] = 9;

args[1] = 18;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(double))

{

object[] args = new object[2];

args[0] = 1.12;

args[1] = 23.4;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Sum") == 0)

{

val = (int)m.Invoke(reflectOb, null);

Console.WriteLine("Sum is: " + val);

}

else if (m.Name.CompareTo("IsBetween") == 0)

{

object[] args = new object[1];

args[0] = 14;

if ((bool)m.Invoke(reflectOb, args))

Console.WriteLine("Value 14 between x and y");

}

else if(m.Name.CompareTo("Show") == 0)

{

m.Invoke(reflectOb, null);

}

}

}

}

Рассмотрим подробнее порядок вызова методов. Сначала создается список методов.

Затем в цикле foreach извлекаются сведения об их параметрах. Далее каждый метод

вызывается с указанием соответствующего типа и числа аргументов в последовательном

ряде условных операторов if/else. Обратите особое внимание на перегрузку метода

Set() в приведенном ниже фрагменте кода.

if(m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(int))

{

object[] args = new object[2];

args[0] = 9;

args[1] = 18;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(double))

{

object[] args = new object[2];

args[0] = 1.12;

args[1] = 23.4;

m.Invoke(reflectOb, args);

}

Если имя метода — Set, то проверяется тип первого параметра, чтобы выявить

конкретный вариант этого метода. Так, если это метод Set(int, int), то его аргументы

загружаются в массив args. В противном случае используются аргументы типа

double.

**Получение конструкторов конкретного типа**

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 Туре. Ниже приведена одна из наиболее часто используемых форм этого метода.

ConstructorInfо[] GetConstructors()

Метод GetConstructors() возвращает массив объектов класса ConstructorInfо, описывающих конструкторы.

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.

В приведенном ниже примере программы рефлексия используется для создания

экземпляра объекта класса MyClass.

(***glava17\_6***)

class MyClass

{

int x;

int y;

public MyClass(int i)

{

Console.WriteLine("Constructor of MyClass (int).");

x = y = i;

}

public MyClass(int i, int j)

{

Console.WriteLine("Constructor of MyClass (int, int).");

x = i;

y = j;

Show();

}

public int Sum()

{

return x + y;

}

public bool IsBetween(int i)

{

if ((x < i) && (i < y)) return true;

else return false;

}

public void Set(int a, int b)

{

Console.Write("In method Set(int, int). ");

x = a;

y = b;

Show();

}

//overload Set method

public void Set(double a, double b)

{

Console.Write("In method Set(double, double). ");

x = (int)a;

y = (int)b;

Show();

}

public void Show()

{

Console.WriteLine("Value x: {0}, value y: {1}", x, y);

}

}

class InvokeConsDemo

{

static void Main()

{

Type t = typeof(MyClass);

int val;

ConstructorInfo[] ci = t.GetConstructors();

Console.WriteLine("Available constructors: ");

foreach (ConstructorInfo c in ci)

{

Console.Write(" " + t.Name + "(");

//show parametrs

ParameterInfo[] pi = c.GetParameters();

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

{

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

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

}

Console.Write(")");

Console.WriteLine();

}

//find constructor

int x;

for (x = 0; x < ci.Length; x++)

{

ParameterInfo[] pi = ci[x].GetParameters();

if (pi.Length == 2) break;

}

if (x == ci.Length)

{

Console.WriteLine("Constructor has not been found.");

return;

}

else Console.WriteLine("Found constructor with two parameters.\n");

//define object

object[] consargs = new object[2];

consargs[0] = 10;

consargs[1] = 20;

object reflectOb = ci[x].Invoke(consargs);

Console.WriteLine("\nCall methods for reflectOb object.");

Console.WriteLine();

MethodInfo[] mi = t.GetMethods();

foreach (MethodInfo m in mi)

{

//get parameters

ParameterInfo[] pi = m.GetParameters();

if (m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(int))

{

object[] args = new object[2];

args[0] = 9;

args[1] = 18;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(double))

{

object[] args = new object[2];

args[0] = 1.12;

args[1] = 23.4;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Sum") == 0)

{

val = (int)m.Invoke(reflectOb, null);

Console.WriteLine("Sum is: " + val);

}

else if (m.Name.CompareTo("IsBetween") == 0)

{

object[] args = new object[1];

args[0] = 14;

if ((bool)m.Invoke(reflectOb, args))

Console.WriteLine("Value 14 between x and y");

}

else if (m.Name.CompareTo("Show") == 0)

{

m.Invoke(reflectOb, null);

}

}

}

}

Сначала получается перечень открытых конструкторов в следующей строке кода.

ConstructorInfo[] ci = t.GetConstructors();

Затем для наглядности примера выводятся полученные конструкторы. После этого

осуществляется поиск по списку конструктора, принимающего два аргумента, как показано

в приведенном ниже фрагменте кода.

for (x = 0; x < ci.Length; x++)

{

ParameterInfo[] pi = ci[x].GetParameters();

if (pi.Length == 2) break;

}

Если такой конструктор найден, как в данном примере, то в следующем фрагменте

кода получается экземпляр объекта заданного типа.

//define object

object[] consargs = new object[2];

consargs[0] = 10;

consargs[1] = 20;

object reflectOb = ci[x].Invoke(consargs);

После вызова метода Invoke() переменная экземпляра reflectOb будет ссылаться

на объект типа MyClass. А далее в программе выполняются соответствующие методы

для экземпляра этого объекта.

Следует, однако, иметь в виду, что ради простоты в данном примере предполагается

наличие лишь одного конструктора с двумя аргументами типа int. Очевидно,

что в реальном коде придется дополнительно проверять соответствие типов каждого

параметра и аргумента.

**Получение типов данных из сборок**

В предыдущем примере все сведения о классе MyClass были получены с помощью

рефлексии, за исключением одного элемента: типа самого класса MyClass. Несмотря

на то что сведения о классе получались в предыдущем примере динамически, этот

пример опирался на тот факт, что имя типа MyClass было известно заранее и использовалось

в операторе typeof для получения объекта класса Туре, по отношению

к которому осуществлялось косвенное или непосредственное обращение к методам

рефлексии. В некоторых случаях такой подход может оказаться вполне пригодным, но

истинные преимущества рефлексии проявляются лишь тогда, когда доступные в программе

типы данных определяются динамически в результате анализа содержимого

других сборок.

Как следует из главы 16, 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.

Туре[] GetTypes()

Этот метод возвращает массив типов, содержащихся в сборке.

Для того чтобы продемонстрировать порядок обнаружения типов в сборке, потребуются

два исходных файла. Первый файл будет содержать ряд классов, обнаруживаемых

в коде из второго файла. Создадим сначала файл MyClasses.cs, содержащий

следующий код.

(***MyClasses.cs***)

class MyClass

{

int x;

int y;

public MyClass(int i)

{

Console.WriteLine("Constructor MyClass(int).");

x = y = i;

Show();

}

public MyClass(int i, int j)

{

Console.WriteLine("Constructor MyClass(int, int).");

x = i;

y = j;

Show();

}

public int sum()

{

return x + y;

}

public bool IsBetween(int i)

{

if ((x < i) && (i < y)) return true;

else return false;

}

public void Set(int a, int b)

{

Console.WriteLine("In method Set(int, int).");

x = a;

y = b;

Show();

}

//overload Set

public void Set(double a, double b)

{

Console.WriteLine("Constructor Set(double, double).");

x = (int)a;

x = (int)b;

Show();

}

public void Show()

{

Console.WriteLine("Values x: {0}, y: {1}", x, y);

}

}

class AnotherClass

{

string msg;

public AnotherClass(string str)

{

msg = str;

}

public void Show()

{

Console.WriteLine(msg);

}

}

class Demo

{

static void Main()

{

Console.WriteLine("Its envelope.");

}

}

Этот файл содержит класс MyClass, неоднократно использовавшийся в предыдущих

примерах. Кроме того, в файл добавлены второй класс AnotherClass и третий

класс Demo. Следовательно, сборка, полученная из исходного кода, находящегося в

этом исходном файле, будет содержать три класса. Затем этот файл компилируется, и

из него формируется исполняемый файл MyClasses.ехе. Именно эта сборка и будет

опрашиваться программно.

Ниже приведена программа, в которой будут извлекаться сведения о файле сборки

MyClasses.ехе. Ее исходный текст составляет содержимое второго файла.

(***glava17\_7***)

class ReflectAssemblyDemo

{

static void Main()

{

int val;

//load MyClasses.exe

Assembly asm = Assembly.LoadFrom("myclasses.exe");//has to be in Debug folder!!!

//find types in MyClasses.exe

Type[] alltypes = asm.GetTypes();

foreach (Type temp in alltypes)

{

Console.WriteLine("Found: " + temp.Name);

}

Console.WriteLine();

//use first type, in this case MyClass

Type t = alltypes[0]; //use the first found class

Console.WriteLine("Used: " + t.Name);

//get info about constructor

ConstructorInfo[] ci = t.GetConstructors();

Console.WriteLine("Available constructors: ");

foreach (ConstructorInfo c in ci)

{

//show return type and name

Console.Write(" " + t.Name + "(");

//show parameters

ParameterInfo[] pi = c.GetParameters();

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

{

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

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

}

Console.WriteLine(")");

}

Console.WriteLine();

//find suitable constructor

int x;

for (x = 0; x < ci.Length; x++)

{

ParameterInfo[] pi = ci[x].GetParameters();

if (pi.Length == 2) break;

}

if (x == ci.Length)

{

Console.WriteLine("Needed constructor was not found.");

return;

}

else Console.WriteLine("Needed constructor with two params has found.\n");

//define object

object[] consargs = new object[2];

consargs[0] = 10;

consargs[1] = 20;

object reflectOb = ci[x].Invoke(consargs);

Console.WriteLine("\nCall method for reflectOb.");

Console.WriteLine();

MethodInfo[] mi = t.GetMethods();

//call each method

foreach (MethodInfo m in mi)

{

//get params

ParameterInfo[] pi = m.GetParameters();

if (m.Name.CompareTo("Set") == 0 &&

pi[0].ParameterType == typeof(int))

{

//this is method Set(int, int)

object[] args = new object[2];

args[0] = 9;

args[1] = 18;

m.Invoke(reflectOb, args);

}

else if (m.Name.CompareTo("Set") == 0 &&

pi[1].ParameterType == typeof(double))

{

//this is method Set(double, double)

object[] args = new object[2];

args[0] = 1.12;

args[1] = 23.4;

m.Invoke(reflectOb, args);

}

else if(m.Name.CompareTo("Sum") == 0)

{

val = (int)m.Invoke(reflectOb, null);

Console.WriteLine("The summ is: " + val);

}

else if(m.Name.CompareTo("IsBetween")==0)

{

object[] args = new object[1];

args[0] = 14;

if((bool) m.Invoke(reflectOb, args))

Console.WriteLine("Value 14 is between x and y");

}

else if (m.Name.CompareTo("Show") == 0)

{

m.Invoke(reflectOb, null);

}

}

}

}

Как следует из результата выполнения приведенной выше программы, обнаружены

все три класса, содержащиеся в файле сборки MyClasses.ехе. Первым среди них

обнаружен класс MyClass, который затем был использован для получения экземпляра

объекта и вызова соответствующих методов.

Отдельные типы обнаруживаются в сборке MyClasses.ехе с помощью приведенной

ниже последовательности кода, находящегося в самом начале метода Маin().

//load MyClasses.exe

Assembly asm = Assembly.LoadFrom("myclasses.exe");//has to be in Debug folder!!!

//find types in MyClasses.exe

Type[] alltypes = asm.GetTypes();

foreach (Type temp in alltypes)

{

Console.WriteLine("Found: " + temp.Name);

}

Console.WriteLine();

Этой последовательностью кода можно пользоваться всякий раз, когда требуется

динамически загружать и опрашивать сборку.

Но сборка совсем не обязательно должна быть исполняемым файлом с расширением

.ехе. Сборки могут быть также в файлах динамически компонуемых библиотек

(DLL) с расширением .dll.

**Полностью автоматизированное обнаружение типов**

Прежде чем завершить рассмотрение рефлексии, обратимся к еще одному поучительному

примеру. Несмотря на то что в программе из предыдущего примера класс

MyClass был полноценно использован без явного указания на его имя в программе,

этот пример все же опирается на предварительную осведомленность о содержимом

класса MyClass. Так, в программе были заранее известны имена методов Set и Sum из

этого класса. Но с помощью рефлексии можно воспользоваться типом данных, ничего

не зная о нем заранее. С этой целью придется извлечь все сведения, необходимые

для конструирования объекта и формирования вызовов соответствующих методов. Такой

подход может оказаться пригодным, например, при создании инструментального

средства визуального проектирования, поскольку он позволяет использовать типы

данных, имеющиеся в системе.

Рассмотрим следующий пример, демонстрирующий полностью автоматизированное

обнаружение типов. В этом примере сначала загружается сборка MyClasses.ехе,

затем конструируется объект класса MyClass и далее вызываются все методы, объявленные

в классе MyClass, причем о них ничего заранее неизвестно.

(***glava17\_8***)

class ReflectAssemblyDemo

{

static void Main()

{

int val;

Assembly asm = Assembly.LoadFrom("myclasses.exe");

Type[] alltypes = asm.GetTypes();

Type t = alltypes[0];//use first found class

Console.WriteLine("Used: " + t.Name);

ConstructorInfo[] ci = t.GetConstructors();

//use first found constructor

ParameterInfo[] cpi = ci[0].GetParameters();

object reflectOb;

if (cpi.Length > 0)

{

object[] consargs = new object[cpi.Length];

//initialize arguments

for (int n = 0; n < cpi.Length; n++)

consargs[n] = 10 + n \* 20;

//make an object

reflectOb = ci[0].Invoke(consargs);

}

else

reflectOb = ci[0].Invoke(null);

Console.WriteLine("\nCall method for reflectOb.");

Console.WriteLine();

//ignore inherent methods

MethodInfo[] mi = t.GetMethods(BindingFlags.DeclaredOnly | BindingFlags.Instance

| BindingFlags.Public);

//call each method

foreach (MethodInfo m in mi)

{

Console.WriteLine("Calling method {0}", m.Name);

//get parameters

ParameterInfo[] pi = m.GetParameters();

//use methods

switch (pi.Length)

{

case 0: //no arguments

if (m.ReturnType == typeof(int))

{

val = (int)m.Invoke(reflectOb, null);

Console.WriteLine("Result: " + val);

}

else if (m.ReturnType == typeof(void))

{

m.Invoke(reflectOb, null);

}

break;

case 1: //one argument

if (pi[0].ParameterType == typeof(int))

{

object[] args = new object[1];

args[0] = 14;

if ((bool)m.Invoke(reflectOb, args))

Console.WriteLine("Value 14 between x and y");

else

Console.WriteLine("Value 14 is not between x and y");

}

break;

case 2: //two arguments

if ((pi[0].ParameterType == typeof(int)) &&

(pi[1].ParameterType == typeof(int)))

{

object[] args = new object[2];

args[0] = 9;

args[1] = 18;

m.Invoke(reflectOb, args);

}

else if ((pi[0].ParameterType == typeof(double)) &&

(pi[1].ParameterType == typeof(double)))

{

object[] args = new object[2];

args[0] = 1.12;

args[1] = 23.4;

m.Invoke(reflectOb, args);

}

break;

}

Console.WriteLine();

}

}

}

Эта программа работает довольно просто, но все же требует некоторых пояснений.

Во-первых, получаются и используются только те методы, которые явно объявлены

в классе MyClass. Для этой цели служит форма BindingFlags метода GetMethods(),

чтобы воспрепятствовать вызову методов, наследуемых от объекта. И во-вторых, количество

параметров и возвращаемый тип каждого метода получаются динамически,

а затем определяются и проверяются в операторе switch. На основании этой информации

формируется вызов каждого метода.