C# 9.0. Справочник

Дополнительные материалы

Содержание

26

Сериализация

Настоящая глава посвящена сериализации и десериализации — механизмам, с помощью которых объекты могут быть представлены в простой текстовой или двоичной форме. Если не отмечено особо, то все типы, упоминаемые в главе, находятся в следующих пространствах имен:

```
System.Runtime.Serialization
System.Xml.Serialization
System.Text.Json
```

Концепции сериализации

Сериализация — это действие по превращению находящегося в памяти объекта или *графа объектов* (набора объектов, ссылающихся друг на друга) в поток байтов, XML, JSON или похожее представление, которое можно сохранять или передавать. *Десериализация* работает в противоположном направлении, получая поток данных и восстанавливая его в объект или граф объектов в памяти.

Сериализация и десериализация обычно используются для решения следующих задач:

- передача объектов по сети или за границы приложения;
- сохранение представлений объектов внутри файла или базы данных.

Еще одно менее распространенное применение связано с глубоким копированием объектов. Механизмы сериализации на основе контрактов данных и XML могут также использоваться в качестве универсальных инструментов для загрузки и сохранения XML-файлов с известной структурой, а сериализатор JSON может делать то же самое для файлов JSON.

Сериализация и десериализация в .NET поддерживается как с точки зрения клиентов, желающих сериализировать и десериализировать объекты, так и с точки зрения типов, которым требуется определенный контроль над способом их сериализации.

Механизмы сериализации

В .NET 5+ и .NET Core доступны четыре механизма сериализации:

- `XmlSerializer` (XML);
- `JsonSerializer` (JSON);
- (в некоторой степени избыточный) сериализатор на основе контрактов данных (XML и JSON);
- двоичный сериализатор.

На заметку! Частота использования двоичного сериализатора постепенно сокращается из-за его уязвимостей в плане безопасности. За деталями обращайтесь по ссылке `https://aka.ms/binaryformatter`.

При выполнении сериализации в XML вы можете выбирать между сериализатором `XmlSerializer` и сериализатором на основе контрактов данных. Сериализатор `XmlSerializer` обеспечивает великолепную гибкость в том, как структурирована разметка XML, тогда как сериализатор на основе контрактов данных обладает возможностью предохранения разделяемых объектных ссылок.

При сериализации в JSON у вас тоже есть выбор. `JsonSerializer` предлагает более высокую производительность, а начиная с версии

.NET 5, имеется несколько причин отдать предпочтение сериализатору на основе контрактов данных. И если `JsonSerializer` не предоставляет необходимые вам средства, то еще одним вариантом является популярная сторонняя библиотека Json.NET.

Если вам нужно взаимодействовать с унаследованными веб-службами, базирующимися на SOAP, тогда сериализатор на основе контрактов данных будет наилучшим выбором.

А если вас не заботит формат, то механизм двоичной сериализации является наиболее мощным и простым в использовании. Однако его вывод нечитабелен для человека и менее устойчив к версиям, чем другие сериализаторы.

На заметку! С учетом того, что от двоичного сериализатора постепенно отказываются, в настоящее время наилучшим выбором считается `JsonSerializer`, если вам безразличен формат.

В табл. 26.1 приведены сравнительные оценки всех механизмов. Чем больше указано звездочек, тем выше (и лучше) оценка.

Обратите внимание, что для достижения хорошей производительности механизм сериализации XML требует повторного использования того же самого объекта `XmlSerializer`.

Таблица 26.1. Сравнение механизмов сериализации

Функциональная возможность	`XmlSerializer`	`JsonSerializer`	Сериализатор на основе контрактов данных	Двоичный сериализатор
Уровень автоматизации	****	*****	***	*****
Выходной формат	XML	JSON	XML или JSON	Двоичный
Привязка к типам	Слабая	Слабая	Слабая	Тесная
Устойчивость к версиям	*****	*****	*****	***
Возможность сериализации подтипов	указания подтипов	Нет	Требует указания подтипов	Да
Предохранение объектных ссылок	Нет	Начиная с .NET 5	В случае XML	Да
Возможность сериализации неоткрытых полей	Нет	Начиная с .NET 5	Да	Да
Пригодность к обмену сообщениями с возможностью взаимодействия	Да	Да	Да	Нет
Гибкость в выходном формате	****	***	**	–
Сжатие вывода	**	***	**	****
Производительность	От * до ***	****	***	***

Почему механизмов четыре?

Причина наличия четырех механизмов сериализации отчасти историческая. Изначально перед сериализацией в .NET Framework стояли две разные цели:

- сериализация графов объектов .NET с полной точностью типов и ссылок;
- возможность взаимодействия со стандартами обмена сообщениями XML и SOAP.

Первая цель достигалась двоичным сериализатором (который применялся инфраструктурой .NET Remoting), а вторая — сериализатором `XmlSerializer` (который использовался веб-службами ASMX).

С выходом в 2006 году инфраструктуры Windows Communication Foundation (WCF) понадобился новый механизм сериализации — *сериализатор на основе контрактов данных* — и была надежда, что он сумеет в значительной степени заменить два предшествующих механизма. Тем не менее, поскольку он был спроектирован с ориентацией в основном на средства, связанные с обменом сообщениями, он не смог полностью достичь данной цели и два предшествующих механизма остались востребованными.

`XmlSerializer`

Механизм сериализации XML может генерировать только данные XML и обладает меньшими возможностями, чем двоичный сериализатор и сериализатор на основе контрактов данных, в том, что касается сохранения и восстановления сложного графа объектов (не позволяет восстанавливать разделяемые объектные ссылки). Однако среди четырех механизмов он отличается наибольшей гибкостью в следовании произвольной структуре XML. Например, можно выбирать, во что должны сериализироваться свойства — в элементы или в атрибуты, и обрабатывать внешний элемент коллекции. Механизм сериализации XML также характеризуется великолепной устойчивостью к версиям. Сериализатор `XmlSerializer` применяется веб-службами ASMX.

`JsonSerializer`

Сериализатор JSON работает быстро и эффективно. Он также отличается хорошей устойчивостью к версиям и делает возможным использование специальных преобразователей, увеличивая тем самым гибкость. Сериализатор `JsonSerializer` применяется в ASP.NET Core 3, устраняя зависимость от библиотеки Json.NET, хотя если возникнет потребность в ее функциях, то возвратиться к Json.NET очень легко.

Начиная с .NET 5, сериализатор JSON способен предохранять объектные ссылки.

Сериализатор на основе контрактов данных

Сериализатор на основе контрактов данных поддерживает модель *контрактов данных*, которая помогает отвязать низкоуровневые детали типов, подлежащих сериализации, от структуры сериализированных данных. В результате обеспечивается высокая устойчивость к версиям, что означает возможность десериализации данных, которые были сериализированы из более ранней или более поздней версии типа. Можно даже десериализировать типы, которые были переименованы или перемещены в другую сборку.

Сериализатор на основе контрактов данных способен справиться с большинством графов объектов, хотя он требует большего внимания, чем двоичный сериализатор. При наличии свободы в выборе структуры XML он также может применяться в качестве универсального инструмента для чтения/записи XML-файлов. (Если необходимо хранить данные в атрибутах или иметь дело с XML-элементами, расположенными в произвольном порядке, то сериализатор на основе контрактов данных использовать нельзя.)

Двоичный сериализатор

Механизм двоичной сериализации прост в применении, хорошо автоматизирован и повсеместно поддерживается в рамках .NET 5+ и .NET Core 3 (и даже еще лучше в .NET Framework). Довольно часто единственный атрибут — это все, что требуется для того, чтобы сделать сложный тип полностью сериализируемым. Двоичный сериализатор также работает быстрее сериализатора на основе контрактов данных, когда требуется высокая точность типов.

Тем не менее, он тесно связывает внутреннюю структуру типа с форматом сериализированных данных, приводя в результате к низкой устойчивости к версиям (хотя он устойчив к добавлению простого поля). Механизм двоичной сериализации выпускает только двоичные данные; в .NET 5+ и .NET Core 3 он не может генерировать данные в формате XML или JSON. (В .NET Framework имеется форматер для обмена сообщениями на основе SOAP, который обеспечивает ограниченную поддержку XML).

Перехватчик `IXmlSerializable`

Для решения сложных задач сериализации XML можно реализовывать интерфейс `IXmlSerializable` и выполнять сериализацию самостоятельно с помощью классов `XmlReader` и `XmlWriter`. Интерфейс `IXmlSerializable` распознается как классом `XmlSerializer`, так и сериализатором на основе контрактов данных, поэтому его можно избирательно использовать для поддержки более сложных типов. Классы `XmlReader` и `XmlWriter` были подробно описаны в главе 11.

Форматеры

Вывод сериализатора на основе контрактов данных и двоичного сериализатора оформляется с помощью подключаемого *форматера*. Роль форматера одинакова в обоих механизмах сериализации, хотя для выполнения работы они используют совершенно разные классы.

Форматер приводит форму финального представления в соответствие с конкретной средой или контекстом сериализации. В .NET 5+ и .NET Core сериализатор на основе контрактов данных позволяет выбирать между форматерами XML и JSON, а в .NET Framework можно также отдавать предпочтение двоичному форматеру. Двоичный форматер предназначен для работы в контексте, где будут применяться произвольные потоки байтов — как правило, файл/поток или патентованный пакет обмена сообщениями. Двоичный вывод по размерам обычно меньше, чем XML или JSON.

Двоичный сериализатор в .NET 5+ и .NET Core предлагает только двоичный форматер (в .NET Framework существует также форматер SOAP для обмена сообщениями на основе XML).

Сравнение явной и неявной сериализации

Сериализация и десериализация могут быть инициированы двумя способами.

Первый способ предусматривает *явное* запрашивание сериализации и десериализации конкретного объекта. При явной сериализации или десериализации выбирается механизм и форматер.

Напротив, *неявная* сериализация запускается .NET, что происходит в следующих случаях:

- сериализатор рекурсивно сериализирует дочерний объект;
- используется средство, которое полагается на сериализацию, такое как Web API.

Средство Web API может работать с сериализацией XML или JSON.

Неявная сериализация менее распространена в .NET 5+ и .NET Core, чем в инфраструктуре .NET Framework, которая включает WCF (применяется сериализатор на основе контрактов данных), Remoting (неявно используется механизм двоичной сериализации) и ASMX Web Services (неявно применяется `XmlSerializer`).

Сериализатор XML

Класс `XmlSerializer` из пространства имен `System.Xml.Serialization` выполняет сериализацию и десериализацию на основе атрибутов в ваших классах.

Начало работы с сериализацией на основе атрибутов

Чтобы использовать класс `XmlSerializer`, необходимо создать его экземпляр и вызвать метод `Serialize` или `Deserialize` с экземпляром `Stream` и нужным объектом. В целях иллюстрации мы определим следующий класс:

```
public class Person
{
  public string Name;
  public int Age;
}
```

Приведенный ниже код сохраняет объект `Person` в файл XML и затем восстанавливает его:

```
Person p = new Person();
p.Name = "Stacey"; p.Age = 30;

var xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);

Console.WriteLine (p2.Name + " " + p2.Age);  // Stacey 30
```

Методы `Serialize` и `Deserialize` способны работать с объектом `Stream`, `XmlWriter`/`XmlReader` или `TextWriter`/`TextReader`. Вот результирующие данные XML:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Stacey</Name>
  <Age>30</Age>
</Person>
```

Класс XmlSerializer может сериализировать типы, вообще не содержащие атрибутов, такие как наш тип Person. По умолчанию XmlSerializer сериализирует все открытые поля и свойства в типе. Вы можете исключать члены, которые не хотите сериализировать, применяя атрибут XmlIgnore:

```
public class Person
{
  ...
  [XmlIgnore] public DateTime DateOfBirth;
}
```

При десериализации класс XmlSerializer полагается на конструктор без параметров, генерируя исключение, если он отсутствует. (В нашем примере Person имеет неявный конструктор без параметров.) Это также означает, что перед десериализацией выполняются инициализаторы полей:

```
public class Person
{
  public bool Valid = true;  // Выполняется перед десериализацией
}
```

Хотя класс XmlSerializer способен десериализировать почти любой тип, он распознает перечисленные далее типы и трактует их особым образом:

- примитивные типы, DateTime, TimeSpan, Guid и их версии, допускающие null;
- byte[] (преобразуется в данные Base-64);
- экземпляр XmlAttribute или XmlElement (чье содержимое внедряется в поток);
- любой тип, реализующий интерфейс IXmlSerializable;
- любой тип коллекции.

Десериализатор устойчив к версиям: он не сообщает об ошибке в случае отсутствия элементов или атрибутов либо присутствия избыточных данных.

Атрибуты, имена и пространства имен

По умолчанию поля и свойства сериализируются в элемент XML. Взамен можно затребовать использование атрибута XML:

```
[XmlAttribute] public int Age;
```

Управлять именем элемента или атрибута можно следующим образом:

```
public class Person
{
  [XmlElement ("FirstName")] public string Name;
  [XmlAttribute ("RoughAge")] public int Age;
}
```

Вот результат:

```
<Person RoughAge="30" ...>
  <FirstName>Stacey</FirstName>
</Person>
```

Стандартное пространство имен XML является пустым. Для указания пространства имен XML атрибуты [XmlElement] и [XmlAttribute] принимают аргумент Namespace. Можно также назначить имя и пространство имен самому типу с помощью атрибута [XmlRoot]:

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

В итоге элемент person получает имя Candidate; кроме того, элементу и его дочерним элементам назначается пространство имен.

Порядок следования элементов XML

Класс XmlSerializer записывает элементы в порядке, в котором они определены в классе. Изменить его можно за счет указания параметра Order в атрибуте XmlElement:

```
public class Person
{
  [XmlElement (Order = 2)] public string Name;
  [XmlElement (Order = 1)] public int Age;
}
```

В случае если параметр Order вообще применяется, тогда он должен использоваться повсюду. Десериализатор не заботится о порядке следования элементов — они могут появляться в любой последовательности и тип будет надлежащим образом десериализирован.

Подклассы и дочерние объекты

Создание подклассов для корневого типа

Предположим, что корневой тип имеет два подкласса:

```
public class Person { public string Name; }

public class Student : Person { }
public class Teacher : Person { }
```

и нужно написать многократно используемый метод для сериализации корневого типа:

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof (Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

Чтобы метод SerializePerson работал с типом Student или Teacher, класс XmlSerializer потребуется информировать о существовании подклассов. Для этого есть два способа. Первый способ предусматривает регистрацию каждого подкласса путем применения атрибута XmlInclude:

```
[XmlInclude (typeof (Student))]
[XmlInclude (typeof (Teacher))]
public class Person { public string Name; }
```

Второй способ предполагает указание каждого подтипа при конструировании XmlSerializer:

```
XmlSerializer xs = new XmlSerializer (typeof (Person),
                   new Type[] { typeof (Student), typeof (Teacher)
} );
```

В любом случае сериализатор отреагирует регистрацией подтипа в атрибуте type:

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Stacey</Name>
</Person>
```

Благодаря этому атрибуту десериализатор будет знать о необходимости создания экземпляра Student, а не Person.

На заметку! Управлять именем, которое появляется в XML-атрибуте type, можно за счет применения к подклассу атрибута [XmlType]:

```
[XmlType ("Candidate")]
public class Student : Person { }
```

Вот результат:

```
<Person xmlns:xsi="..."
        xsi:type="Candidate">
```

Сериализация дочерних объектов

Класс XmlSerializer автоматически проходит по объектным ссылкам, таким как поле HomeAddress в Person:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address();
}
public class Address { public string Street, PostCode; }
```

Ниже показана демонстрация:

```
Person p = new Person { Name = "Stacey" };
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

В результате сериализации получается следующие данные XML:

```
<Person ... >
  <Name>Stacey</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

На заметку! При наличии двух полей или свойств, которые ссылаются на один и тот же объект, этот объект сериализируется дважды. Если необходимо предохранить ссылочную эквивалентность, тогда должен использоваться другой механизм сериализации.

Создание подклассов для дочерних объектов

Предположим, что нужно сериализировать класс `Person`, который может ссылаться на подклассы класса `Address`:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address { }
public class AUAddress : Address { }

public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

В зависимости от того, как желательно структурировать разметку XML, существуют два способа продолжения. Если вы хотите, чтобы имя элемента всегда совпадало с именем поля или свойства с подтипом, указанным в атрибуте `type`:

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

тогда применяйте `[XmlInclude]` для регистрации каждого подкласса в классе `Address`:

```
[XmlInclude (typeof (AUAddress))]
[XmlInclude (typeof (USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

С другой стороны, если вы хотите, чтобы имя элемента отражало имя подтипа со следующим результатом:

```
<Person ...>
  ...
  <USAddress>
    ...
  </USAddress>
</Person>
```

то взамен укажите множество атрибутов `[XmlElement]` перед полем или свойством в родительском типе:

```
public class Person
{
  public string Name;
  [XmlElement ("Address", typeof (Address))]
  [XmlElement ("AUAddress", typeof (AUAddress))]
  [XmlElement ("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress();
}
```

Каждый `XmlElement` сопоставляет имя элемента с типом. В случае принятия такого подхода вам не придется применять атрибут `[XmlInclude]` в типе `Address` (хотя его присутствие не нарушит сериализацию).

На заметку! Если вы опускаете имя элемента в `[XmlElement]` (и указываете только тип), то используется стандартное имя типа (на которое влияет `[XmlType]`, но не `[XmlRoot]`).

Сериализация коллекций

Класс `XmlSerializer` распознает и сериализирует конкретные типы коллекций без какого-либо вмешательства:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}
public class Address { public string Street, PostCode; }
```

В результате сериализации получается показанная далее разметка XML:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    ...
  </Addresses>
</Person>
```

Атрибут `[XmlArray]` позволяет переименовывать *внешний* элемент (т.е. `Addresses`).

Атрибут `[XmlArrayItem]` позволяет переименовывать *внутренние* элементы (т.е. элементы `Address`).

Например, приведенный ниже класс:

```
public class Person
{
  public string Name;
  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

сериализируется следующим образом:

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

Атрибуты `XmlArray` и `XmlArrayItem` также позволяют указывать пространства имен XML. Чтобы сериализировать коллекцию без внешнего элемента, скажем:

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

необходимо добавить к полю или свойству типа коллекции атрибут `[XmlElement]`:

```
public class Person
{
  ...
  [XmlElement ("Address")]
  public List<Address> Addresses = new List<Address>();
}
```

Работа с элементами коллекции, являющимися подклассами

Правила создания подклассов элементов коллекций естественным образом вытекают из других правил создания подклассов. Чтобы закодировать элементы, являющиеся подклассами, с помощью атрибута `type`, например:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
    ...
```

нужно добавить атрибуты `[XmlInclude]` к базовому типу (`Address`), как делалось ранее. Прием работает независимо от того, подавляется сериализация внешнего элемента или нет.

Если необходимо именовать элементы, являющиеся подклассами, в соответствии с их типами, скажем:

```
<Person ... >
  <Name>...</Name>
  <!--Начало необязательного внешнего элемента-->
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <!--Конец необязательного внешнего элемента-->
</Person>
```

тогда для поля или свойства типа коллекции понадобится указать множество атрибутов `[XmlArrayItem]` или `[XmlElement]`. Если желательно *включить* внешний элемент типа коллекции, то следует указать множество атрибутов `[XmlArrayItem]`:

```
[XmlArrayItem ("Address", typeof (Address))]
[XmlArrayItem ("AUAddress", typeof (AUAddress))]
[XmlArrayItem ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

Если желательно *исключить* внешний элемент типа коллекции, тогда нужно указать множество атрибутов `[XmlElement]`:

```
[XmlElement ("Address", typeof (Address))]
[XmlElement ("AUAddress", typeof (AUAddress))]
[XmlElement ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

IXmlSerializable

Хотя сериализация XML на основе атрибутов является гибкой, ей присущи ограничения. Например, нельзя добавлять перехватчики сериализации и невозможно сериализировать неоткрытые члены. Кроме того, ее неудобно ис-

пользовать, если разметка XML может представлять один и тот же элемент или атрибут несколькими отличающимися способами. Что касается последней проблемы, то ситуацию можно в некоторой степени улучшить, передавая конструктору класса `XmlSerializer` объект `XmlAttributeOverrides`. Однако наступает момент, когда легче применить императивный подход. За работу подобного рода отвечает интерфейс `IXmlSerializable`:

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml (XmlReader reader);
  void WriteXml (XmlWriter writer);
}
```

Реализация интерфейса `IXmlSerializable` обеспечивает полный контроль над тем, как разметка XML читается и записывается.

На заметку! Класс коллекции, который реализует интерфейс `IXmlSerializable`, обходит правила для сериализации коллекций, принятые в классе `XmlSerializer`. Это может быть удобно, если необходимо сериализировать коллекцию с полезной нагрузкой — другими словами, с дополнительными полями или свойствами, которые иначе бы игнорировались.

Ниже описаны правила реализации `IXmlSerializable`:

- метод `ReadXml` должен читать внешний начальный элемент, далее содержимое и затем внешний конечный элемент;
- метод `WriteXml` должен записывать только содержимое.

Вот пример:

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

public class Address : IXmlSerializable
{
  public string Street, PostCode;
  public XmlSchema GetSchema() { return null; }
  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement();
    Street = reader.ReadElementContentAsString ("Street", "");
    PostCode = reader.ReadElementContentAsString ("PostCode", "");
    reader.ReadEndElement();
  }

  public void WriteXml (XmlWriter writer)
  {
    writer.WriteElementString ("Street", Street);
    writer.WriteElementString ("PostCode", PostCode);
  }
}
```

Сериализация и десериализация экземпляра Address через XmlSerializer автоматически инициирует вызовы методов WriteXml и ReadXml. Более того, если бы класс Person был определен примерно так:

```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

то метод WriteXml интерфейса IXmlSerializable вызывался бы выборочно для сериализации поля HomeAddress.

Классы XmlReader и XmlWriter были подробно описаны в главе 11. Вдобавок в разделе "Шаблоны для использования XmlReader/XmlWriter" главы 11 приведены примеры классов, готовых к реализации IXmlSerializable.

Сериализатор JSON

Класс JsonSerializer (из пространства имен System.Text.Json) использовать легко из-за простоты формата JSON. Корнем документа JSON является либо массив, либо объект. Под этим корнем находятся свойства, каждое из которых может быть объектом, массивом, строкой, числом, "true", "false" или "null". Сериализатор JSON отображает имена свойств класса прямо на имена свойств в JSON.

Начало работы

Имея следующее определение класса Person:

```
public class Person
{
  public string Name { get; set; }
}
```

мы можем сериализировать его в строку JSON, вызывая метод JsonSerializer.Serialize:

```
var p = new Person { Name = "Ian" };
string json = JsonSerializer.Serialize (p,
              new JsonSerializerOptions { WriteIndented = true });
```

А вот результат:

```
{
  Name: "Ian"
}
```

Десериализация выполняется вызовом метода JsonSerializer.Deserialize:

```
Person p2 = JsonSerializer.Deserialize<Person> (json);
```

Десериализация неизменяемых типов

Десериализатор способен заполнять типы со свойствами, допускающими только чтение, при условии, что имеется открытый конструктор с именами параметров, которые совпадают (без учета регистра) с десериализируемыми свойствами:

```
var p = new Person ("Joe", "Bloggs");
string json = JsonSerializer.Serialize (p);
Person p2 = JsonSerializer.Deserialize<Person> (json);

public class Person
{
  public string FirstName { get; }
  public string LastName { get; }

  public Person (string firstName, string lastName)
    => (FirstName, LastName) = (firstName, lastName);
}
```

Если в типе определено более одного конструктора с параметрами, тогда десериализатору потребуется указать, какой из них использовать, за счет применения к подходящему конструктору атрибута `[JsonConstructor]`.

Записи (появившиеся в C# 9) по умолчанию хорошо работают с десериализатором.

Сериализация дочерних объектов

Пусть класс `Person` определен так, что он имеет домашний и рабочий адреса (типа `Address`):

```
public class Address
{
  public string Street { get; set; }
  public string PostCode { get; set; }
}

public class Person
{
  public string Name { get; set; }
  public Address HomeAddress { get; set; }
  public Address WorkAddress { get; set; }
}
```

Мы можем сериализировать его экземпляр без дополнительной работы:

```
var home = new Address { Street = "1 Main St.", PostCode="11235" };
var work = new Address { Street = "4 Elm Ln.", PostCode="31415" };
var p = new Person { Name = "Ian", HomeAddress = home,
                     WorkAddress = work };

Console.WriteLine (JsonSerializer.Serialize (p,
            new JsonSerializerOptions { WriteIndented = true } ));
```

Встретив `HomeAddress` и `WorkAddress`, сериализатор создает объекты JSON:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "4 Elm Ln.",
    "PostCode": "31415"
  }
}
```

Тем не менее, обратите внимание на то, что произойдет, когда мы установим HomeAddress и WorkAddress в один и тот же экземпляр:

```
var p = new Person { Name = "Ian", HomeAddress = home,
                     WorkAddress = home };
```

Ниже показан вывод:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
}
```

В данных JSON отсутствует информация о том, что HomeAddress и WorkAddress первоначально указывали на тот же самый экземпляр. При десериализации будут созданы два экземпляра Address и присвоены соответствующим свойствам.

Предохранение объектных ссылок

Решить проблему можно (начиная с версии .NET 5), сообщив сериализатору о необходимости предохранения объектных ссылок с использованием следующего параметра:

```
new JsonSerializerOptions
{
  ReferenceHandler = ReferenceHandler.Preserve,
  WriteIndented = true
}
```

Теперь вывод будет примерно таким:

```
{
  "$id": "1",
  "Name": "Ian",
```

```
  "HomeAddress": {
    "$id": "2",
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "$ref": "2"
  }
}
```

При условии, что десериализатору будет предоставлен тот же параметр, данные десериализируются без дублирования объекта `Address`.

Предохранение объектных ссылок также позволяет классу `JsonSerializer` обрабатывать циклы в графе объектов.

Сериализация коллекций

Класс `JsonSerializer` сериализирует коллекции автоматически. Коллекции могут появляться в свойствах объекта, а также в самом корневом объекте. Мы можем проиллюстрировать последнюю ситуацию с применением классов `Person` и `Address`, которые были определены в начале предыдущего раздела:

```
var sara = new Person { Name = "Sara" };
var ian = new Person { Name = "Ian" };
Console.WriteLine (JsonSerializer.Serialize (new[] { sara, ian },
  new JsonSerializerOptions { WriteIndented = true }));
```

Вот результат:

```
[
  {
    "Name": "Sara"
  },
  {
    "Name": "Ian"
  }
]
```

Следующий код десериализирует данные JSON:

```
Person[] people = JsonSerializer.Deserialize<Person[]> (json);
```

Можно сериализировать коллекцию, содержащую по-разному типизированные объекты:

```
var sara = new Person { Name = "Sara" };
var addr = new Address { Street = "1 Main St.", PostCode = "11235" };
Console.WriteLine(JsonSerializer.Serialize (new object[] { sara, addr },
  new JsonSerializerOptions { WriteIndented = true }));
```

Будет получен следующий результат:

```
[
  {
    "Name": "Sara"
  },
```

```
  {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
]
```

Десериализация таких коллекций реализуется неуклюже, поскольку тип каждого элемента в данные JSON не записывается. В отношении десериализации `JsonElement[]` необходимо принять низкоуровневый подход и затем организовать перечисление каждого свойства:

```
var deserialized = JsonSerializer.Deserialize<JsonElement[]>(json);
foreach (var element in deserialized)
{
  foreach (var prop in element.EnumerateObject())
    Console.WriteLine ($"{prop.Name}: {prop.Value}");
  Console.WriteLine ("---");
}
```

Вывод выглядит так:

```
Name: Sara
---
Street: 1 Main St.
PostCode: 11235
```

Использование класса `JsonElement` демонстрировалось в разделе "`JsonDocument`" главы 11.

Управление сериализацией с помощью атрибутов

Управлять процессом сериализации можно с помощью атрибутов, определенных в пространстве имен `System.Text.Json.Serialization`.

`JsonIncludeAttribute` (начиная с версии .NET 5)

По умолчанию сериализатор/десериализатор игнорирует поля, если только к ним не применен атрибут `[JsonInclude]`:

```
class Person
{
  [JsonInclude]
  public string Phone;  // Это поле будет включено
}
```

Атрибут `[JsonInclude]` также можно применять к свойствам с неоткрытым средством доступа `set`, чтобы инструктировать десериализатор о его вызове через рефлексию:

```
class Person
{
  [JsonInclude]
  public string Phone { get; private set; }
}
```

JsonIgnoreAttribute

По умолчанию сериализатор JSON сериализирует все свойства, если только к ним не был применен атрибут `JsonIgnore`:

```
public class Person
{
  public string Name { get; set; }
  [JsonIgnore]
  public decimal NetWorth { get; set; }  // Не сериализируется
}
```

JsonPropertyNameAttribute

Если имя свойства JSON отличается от имени свойства C#, тогда можно создать отображение с помощью атрибута `[JsonPropertyName]`. Скажем, если именем свойства JSON является `"FullName"`, а именем свойства C# — `Name`, то вот как можно было бы создать отображение:

```
public class Person
{
  [JsonPropertyName("FullName")]
  public string Name { get; set; }
}
```

Вот результат сериализации:

```
{
  "FullName":"...",
}
```

JsonExtensionDataAttribute

Возьмем API-интерфейс для веб-приложений, который возвращает экземпляры класса `Person`, и клиент, потребляющий этот API-интерфейс. Оба продукта сопровождаются разными организациями. Если автор API-интерфейса добавляет новое свойство в класс `Person` (вроде `Age`), тогда клиент по-прежнему способен десериализировать данные JSON посредством старого класса `Person`, потому что он просто будет пропускать неизвестное свойство `Age`. Однако предположим, что клиент затем обновляет свой экземпляр `Person`, сериализирует его и отправляет обратно API-интерфейсу. В итоге первоначальное значение `Age` утрачивается.

В целях иллюстрации определим класс `Person` в API-интерфейсе для веб-приложений, как показано ниже:

```
public class Person  // версия 2
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Age { get; set; }  // Новое свойство
}
```

что приведет к генерации данных JSON следующего вида:

```
{
  "Id": 27182,
  "Name": "Sara",
  "Age": 35
}
```

В случае десериализации этих данных JSON в старую версию класса (без свойства `Age`) информацию о возрасте поместить некуда:

```
public class Person  // версия 1
{
  public int Id { get; set; }
  public string Name { get; set; }
}
```

Если позже сериализировать нашу версию и отправить ее обратно API-интерфейсу, то данные JSON не будут содержать свойство `Age`, и API-интерфейс будет интерпретировать свойство `Age` как имеющее нулевое значение (стандартное значение для целого типа).

Класс `JsonExtensionDataAttribute` решает описанную проблему, предоставляя механизм для сохранения всех нераспознанных свойств, так что их значения могут использоваться при повторной сериализации. Когда атрибут `JsonExtensionData` применяется к свойству типа `IDictionary<string,TValue>` (`TValue` должен быть `object` или `JsonElement`), сериализатор использует это свойство для сохранения нераспознанных свойств JSON и никакая информация не утрачивается:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  [JsonExtensionData]
  public IDictionary<string, JsonElement> Storage { get; set; } =
    new Dictionary<string, JsonElement>();
}
```

JsonConverterAttribute

Атрибут `JsonConverter` позволяет указывать тип, используемый для преобразования данных в формат JSON из из него. Мы обсудим его более подробно в следующем разделе.

Настройка преобразования данных

Предположим, что имеется потребность во взаимодействии с поставщиком API-интерфейса, который кодирует даты с применением формата отметок времени Unix (количество секунд, прошедших с момента 1/1/1970):

```
{
  "Id":27182,
  "Name":"Sara",
  "Born":464572800  // Количество секунд, прошедших с момента 1/1/1970
}
```

Необходимо десериализировать эти данные в класс, который использует класс DateTime из .NET:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public DateTime Born { get; set; }
}
```

Достичь цели можно за счет реализации специального преобразователя данных:

```
public class UnixTimestampConverter : JsonConverter<DateTime>
{
  public override DateTime Read (ref Utf8JsonReader reader, Type type,
                                 JsonSerializerOptions options)
  {
    if (reader.TryGetInt32(out int timestamp))
      return new DateTime (1970, 1, 1).AddSeconds (timestamp);
    throw new Exception ("Expected the timestamp as a number.");
  }
  public override void Write (Utf8JsonWriter writer, DateTime value,
                              JsonSerializerOptions options)
  {
    int timestamp =
       (int)(value - new DateTime(1970, 1, 1)).TotalSeconds;
    writer.WriteNumberValue(timestamp);
  }
}
```

Затем можно либо применить атрибут [JsonConverter] к свойствам, которые желательно преобразовать:

```
[JsonConverter(typeof(UnixTimestampConverter))]
public DateTime Born { get; set; }
```

либо если API-интерфейс согласованно представляет типы данных, тогда сделать преобразователь действующим по умолчанию:

```
JsonSerializerOptions opts = new JsonSerializerOptions();
opts.Converters.Add (new UnixTimestampConverter());
var sara = JsonSerializer.Deserialize<Person> (json, opts);
```

Последний код инструктирует сериализатор о том, что каждый раз, когда встречается значение DateTime, он должен использовать UnixTimestamp Converter.

Обработка значений null (начиная с версии .NET 5)

С целью оптимизации производительности при обнаружении значений null сериализатор и десериализатор обычно обходят специальные преобразователи. Если нужно, чтобы специальный преобразователь обрабатывал значения null, тогда придется переопределить свойство HandleNull следующим образом:

```
public class UnixTimestampConverter : JsonConverter<DateTime?>
{
  public override bool HandleNull => true;
  ...
```

Прием работает со значениями null ссылочных типов (скажем, строк) и с типами значений, допускающими null (как в рассмотренном примере).

Параметры сериализации JSON

Сериализатор принимает необязательный параметр JsonSerializationOptions, делая возможным дополнительный контроль над процессом сериализации и десериализации. В последующих подразделах описаны наиболее полезные параметры.

На заметку! Начиная с версии .NET 5, стандартные параметры сериализации отличаются в зависимости от того, разрабатываете вы веб-приложение или нет.

Ниже перечислены параметры, которые отличаются в зависимости от вида разрабатываемого приложения.

Параметр	Обычное значение	Стандартное значение для веб-приложения
PropertyNameCaseInsensitive	false	true
PropertyNamingPolicy	–	CamelCase
NumberHandling	Strict	AllowReadingFromString

Независимо от вида разрабатываемого приложения доступ к каждому набору стандартных параметров возможен через следующие статические свойства:

```
JsonSerializerDefaults.Default
JsonSerializerDefaults.Web
```

Любое из них можно клонировать, используя конструктор, который принимает еще один параметр JsonSerializationOptions:

```
var options = new JsonSerializerOptions (JsonSerializerDefaults.Web);
```

WriteIndented

В текущем разделе параметр WriteIndented устанавливается в true, чтобы сериализатор добавлял пробельные символы к данным JSON, улучшая их читабельность. Стандартным значением является false, что приводит к представлению всех данных в одной строке.

AllowTrailingCommas

Спецификация JSON требует, чтобы свойства и элементы массива отделялись друг от друга запятыми, но не разрешает использовать завершающие запятые:

```
{
  "Name":"Dylan",
  "LuckyNumbers": [10, 7, ],
  "Age":46,
}
```

Завершающие запятые после `7` и `46` по умолчанию не разрешены. Вот как их включить:

```
var commaTolerant = JsonSerializer.Deserialize<Person>
(brokenJson,
  new JsonSerializerOptions { AllowTrailingCommas = true });
```

ReadCommentHandling

По умолчанию десериализатор генерирует исключение, когда встречает комментарии (поскольку комментарии не входят в официальный стандарт JSON). Установка `ReadCommentHandling` в `JsonCommentHandling.Skip` инструктирует десериализатор о необходимости пропуска комментариев, так что следующие данные могут быть успешно разобраны:

```
{
  "Name":"Dylan"  // Комментарий
  /* Еще один комментарий */
}
```

PropertyNameCaseInsensitive

При сопоставлении имен свойств JSON и имен свойств C# десериализатор по умолчанию чувствителен к регистру. Это означает, что следующие входные данные:

```
{ "name":"Dylan" }
```

не смогут заполнить свойство `Name` в нашем классе `Person` (свойство JSON будет проигнорировано).

Проблему решает установка `PropertyNameCaseInsensitive` в `true`, что указывает десериализатору на необходимость выполнять сопоставление, нечувствительное к регистру (за счет небольшого снижения производительности):

```
var dylan = JsonSerializer.Deserialize<Person> (json,
  new JsonSerializerOptions { PropertyNameCaseInsensitive = true
});
```

Если во входных данных регистр используется предсказуемо, то еще одно решение предусматривает применение атрибута `JsonPropertyName` (рассматривался ранее) или параметра `PropertyNamingPolicy` (будет описан следующим).

На заметку! Начиная с версии .NET 5, стандартным значением для этого параметра является `true`, когда разрабатываются веб-приложения (и `false` для приложений других видов).

PropertyNamingPolicy

Для лучшей поддержки популярного соглашения об именовании свойств в "верблюжьем" стиле в версии .NET Core 3 введен параметр `PropertyNamingPolicy`. Он обеспечивает более высокую производительность, чем только что описанный параметр `PropertyNameCaseInsensitive` и применяется как к сериализации, так и к десериализации. Таким образом, код:

```
var dylan = new Person { Name = "Dylan" };
var json = JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

выдает следующие данные:

```
{"name": "Dylan"}
```

которые могут быть десериализированы тем же способом:

```
var dylan2 = JsonSerializer.Deserialize<Person> (json,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

Начиная с версии .NET 5, при написании веб-приложений параметр `CamelCase` принимается по умолчанию.

DictionaryKeyPolicy

С помощью параметра `DictionaryKeyPolicy` можно принудительно обеспечить сериализацию и десериализацию ключей словаря в "верблюжьем" стиле:

```
var dict = new Dictionary<string, string>
{
  { "BookName", "Nutshell" }
  { "BookVersion", "9.0" },
};
Console.WriteLine (JsonSerializer.Serialize (dict,
  new JsonSerializerOptions
  {
    WriteIndented = true,
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase
  }));
```

Вот что выведет показанный код:

```
{
  "bookName": "Nutshell"
  "bookVersion": "9.0",
}
```

Encoder

Стандартный кодировщик текста энергично отменяет символы, так что выходные данные могут появляться в документе HTML без дополнительной обработки:

```
string dylan = "<b>Dylan & Friends</b>";
Console.WriteLine (JsonSerializer.Serialize (dylan));
```

Вот вывод:

```
"\u003Cb\u003EDylan \u0026 Friends\u003C/b\u003E"
```

Предотвратить проблему подобного рода можно, изменив Encoder:

```
Console.WriteLine (JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions {
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
  }));
```

Результат выглядит следующим образом:

```
"<b>Dylan & Friends</b>"
```

UnsafeRelaxedJsonEscaping — это подкласс класса System.Text.Encodings.Web.JavaScriptEncoder. При необходимости можно реализовать собственный подкласс для обретения полного контроля над процессом кодировки.

IgnoreNullValues

По умолчанию свойства со значениями null включаются в выходные данные JSON, так что

```
var person = new Person { Name = null };
```

сериализируется в

```
{
  "Name": null
}
```

В случае установки параметра IgnoreNullValues в true свойства со значениями null полностью игнорируются:

```
Console.WriteLine (JsonSerializer.Serialize (person),
  new JsonSerializerOptions { IgnoreNullValues = true } ));
```

Вывод будет таким:

```
{}
```

IgnoreReadOnlyProperties

По умолчанию свойства, допускающие только чтение, сериализируются (но не десериализируются, потому что средство доступа set отсутствует). За счет установки параметра IgnoreReadOnlyProperties в true сериализатору можно сообщить о том, что свойства, допускающие только чтение, должны быть проигнорированы.

`NumberHandling` (начиная с версии .NET 5)

Числа в JSON обычно форматируются без кавычек. Тем не менее, не все средства записи JSON следуют такому соглашению, поэтому в .NET 5 был введен параметр для поддержки чтения и записи чисел в кавычках:

```
var options = new JsonSerializerOptions
{
  WriteIndented = true,
  NumberHandling = JsonNumberHandling.AllowReadingFromString |
                   JsonNumberHandling.WriteAsString
};
string json =
  JsonSerializer.Serialize (new Point { X = 2, Y = 3}, options);
Console.WriteLine (json);
var p2 =
 (Point) JsonSerializer.Deserialize (json, typeof (Point), options);
public class Point
{
  public int X { get; set; }
  public int Y { get; set; }
}
```

Вот вывод:

```
{
  "X": "2",
  "Y": "3"
}
```

При написании веб-приложений по умолчанию принимается `JsonNumberHandling.AllowReadingFromString`, т.е. числа можно безопасно читать независимо от того, находятся они в кавычках или нет.

Сериализатор на основе контрактов данных

Сериализатор на основе контрактов данных поддерживает модель *контрактов данных*, которая помогает отвязать низкоуровневые детали типов, подлежащих сериализации, от структуры сериализированных данных. В результате обеспечивается великолепная устойчивость к версиям, что означает возможность десериализации данных, которые были сериализированы из более ранней или более поздней версии типа. Можно даже десериализировать типы, которые были переименованы или перемещены в другую сборку.

Сериализатор на основе контрактов данных способен справиться с большинством графов объектов, хотя он требует большего внимания, чем двоичный сериализатор. При наличии свободы в выборе структуры XML он также может применяться в качестве универсального инструмента для чтения/записи XML-файлов. (Если необходимо хранить данные в атрибутах либо иметь дело с XML-элементами, расположенными в произвольном порядке, то сериализатор на основе контрактов данных использовать нельзя.)

Начало работы

Использование сериализатора на основе контрактов данных предусматривает выполнение следующих базовых шагов.

1. Решить, какой класс применять — `DataContractSerializer` или `DataContractJsonSerializer` (в .NET Framework также имеется `NetDataContractSerializer`).
2. Декорировать сериализируемые типы и члены с помощью атрибутов `[DataContract]` и `[DataMember]` соответственно.
3. Создать экземпляр сериализатора и вызвать его метод `WriteObject` или `ReadObject`.

В случае выбора `DataContractSerializer` также понадобится зарегистрировать "известные" типы (подтипы, которые тоже будут сериализироваться) и принять решение по поводу предохранения объектных ссылок.

Может еще возникнуть необходимость в специальном действии для обеспечения надлежащей сериализации коллекций.

На заметку! Типы для сериализатора на основе контрактов данных определены в пространстве имен `System.Runtime.Serialization` внутри сборки с таким же именем.

Выбор сериализатора

Доступны три сериализатора на основе контрактов данных.

- `DataContractSerializer`. Обеспечивает слабую привязку типов .NET к типам контрактов данных через XML.
- `DataContractJsonSerializer`. Обеспечивает слабую привязку типов .NET к типам контрактов данных через JSON.
- `NetDataContractSerializer`. Осуществляет тесную привязку типов .NET к типам контрактов данных (только .NET Framework).

Первые два класса требуют заблаговременной явной регистрации сериализируемых подтипов, чтобы иметь возможность сопоставления имени контракта данных, такого как `Person`, с корректным типом .NET. Классу `NetDataContractSerializer` подобная помощь не нужна, т.к. он записывает полные имена типов и сборок для сериализируемых типов, что довольно похоже на механизм двоичной сериализации:

```
<Person z:Type="SerialTest.Person" z:Assembly=
 "SerialTest,Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

При выполнении десериализации такие выходные данные полагаются на наличие специфического типа .NET в конкретном пространстве имен и сборке.

Если вы сохраняете граф объектов в "черный ящик", то можете выбрать любой из сериализаторов в зависимости от того, какие преимущества вам представляются важными. Если коммуникации производятся через WCF или выполняется чтение/запись XML-файла, тогда наиболее вероятно, что вам понадобится класс `DataContractSerializer`.

Все упомянутые темы будут подробно рассматриваться в последующих разделах.

Использование сериализаторов

Следующий шаг после выбора сериализатора — присоединение атрибутов к типам и членам, которые нужно сериализировать. Необходимо предпринять минимум следующие действия:

- добавить атрибут `[DataContract]` к каждому типу;
- добавить атрибут `[DataMember]` к каждому члену, который должен быть включен.

Ниже приведен пример:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

Таких атрибутов вполне достаточно для того, чтобы сделать тип *неявно* сериализируемым посредством механизма сериализации на основе контрактов данных.

Затем объект можно *явно* сериализировать и десериализировать, создавая экземпляр класса сериализатора и вызывая метод `WriteObject` или `ReadObject`:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));
using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);                    // Сериализировать
Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);          // Десериализировать
Console.WriteLine (p2.Name + " " + p2.Age);  // Stacey 30
```

Конструктор класса `DataContractSerializer` требует указания типа *корневого объекта* (типа объекта, который явно сериализируется). В отличие от него конструктор класса `NetDataContractSerializer` этого не требует:

```
var ns = new NetDataContractSerializer();
// В других отношениях класс NetDataContractSerializer
// используется точно так же, как DataContractSerializer.
...
```

Оба типа сериализаторов по умолчанию применяют форматер XML. При работе с классом `XmlWriter` можно запросить добавление в вывод отступов, улучшая читабельность:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings(){Indent = true};
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Ниже показан результат:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

Имя XML-элемента `<Person>` отражает *имя контракта данных*, которое по умолчанию представляет собой имя типа .NET. Такое поведение можно переопределить и явно указать имя контракта данных следующим образом:

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

Пространство имен XML отражает *пространство имен контракта данных*, которым по умолчанию является `http://schemas.datacontract.org/2004/07/`, а также пространство имен типа .NET. Поведение можно переопределить в аналогичной манере:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

На заметку! Указание имени и пространства имен разрывает связь между идентичностью контракта и именем типа .NET. Это позволяет гарантировать, что в случае проведения рефакторинга и изменения имени либо пространства имен типа сериализация не будет затронута.

Можно также переопределять имена данных-членов:

```
[DataContract (Name="Candidate",
               Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

Вот как будет выглядеть вывод:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

Атрибут `[DataMember]` поддерживает поля и свойства — открытые и закрытые. Тип данных поля или свойства может быть одним из перечисленных ниже:

- любой примитивный тип;
- `DateTime`, `TimeSpan`, `Guid`, `Uri` или перечисление;
- версии указанных выше типов, которые допускают значение `null`;
- `byte[]` (сериализируется в XML с использованием кодировки Base-64);
- любой "известный" тип, декорированный с помощью `DataContract`;
- любой тип, реализующий интерфейс `IEnumerable` (как показано ниже в разделе "Сериализация коллекций");
- любой тип, который снабжен атрибутом `[Serializable]` или реализует интерфейс `ISerializable` (как показано в разделе "Расширение контрактов данных" далее в главе);
- любой тип, реализующий интерфейс `IXmlSerializable`.

Указание двоичного форматера (только .NET Framework)

Двоичный форматер можно применять с классом `DataContractSerializer` или `NetDataContractSerializer`. Процесс аналогичен:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter
           w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader
           r = XmlDictionaryReader.CreateBinaryReader (s2,
                    XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

Объем вывода варьируется между немного меньшим, чем при получении из форматера XML, и существенно меньшим, если типы содержат крупные массивы.

Сериализация подклассов

Для поддержки сериализации подклассов с помощью NetDataContract Serializer никаких специальных усилий прикладывать не придется. Единственное требование заключается в том, что подклассы должны иметь атрибут DataContract. Сериализатор будет записывать полностью определенные имена действительных типов, которые он сериализирует, следующим образом:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=null">
```

Тем не менее, класс DataContractSerializer должен быть информирован обо всех подтипах, которые ему предстоит сериализировать или десериализировать. В целях иллюстрации предположим, что мы создаем подклассы Person, как показано ниже:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

и затем реализуем метод для клонирования экземпляра Person:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

который далее вызываем:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };
Person  p2 =           DeepClone (person);   // Нормально
Student s2 = (Student) DeepClone (student);  // Генерируется
                                             // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);  // Генерируется
                                             // SerializationException
```

Метод DeepClone работает, когда он вызывается с объектом Person, но приводит к генерации исключения при вызове с объектом Student или Teacher, поскольку десериализатор не имеет возможности узнать, в какой тип .NET (или сборку) должно быть преобразовано имя “Student” или “Teacher”. Это также способствует обеспечению безопасности в том, что предотвращает десериализацию непредвиденных типов.

Решение предусматривает указание всех разрешенных, или "известных", подтипов. Такое действие можно предпринять либо при конструировании экземпляра DataContractSerializer:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

либо в самом типе с помощью атрибута KnownType:

```
[DataContract, KnownType (typeof (Student)),
               KnownType (typeof (Teacher))]
public class Person
...
```

Ниже показано, как теперь будет выглядеть сериализированный объект Student:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
  ...
<Person>
```

Из-за того, что в качестве корневого типа указан Person, корневой элемент по-прежнему имеет такое имя. Действительный подкласс описан отдельно — в атрибуте type.

На заметку! Класс NetDataContractSerializer влияет на производительность при сериализации подтипов, причем с любым форматером. Ситуация выглядит так, будто бы, столкнувшись с подтипом, форматер должен остановиться и подумать некоторое время!

Производительность сериализации имеет значение на сервере приложений, который обрабатывает множество параллельных запросов.

Объектные ссылки

Ссылки на другие объекты тоже сериализируются. Рассмотрим следующие классы:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Ниже показан результат их сериализации в XML с использованием класса DataContractSerializer:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

На заметку! Написанный в предыдущем разделе метод DeepClone клонировал бы также и член HomeAddress — что отличает его от простого метода MemberwiseClone.

Если работа производится с классом DataContractSerializer, то при создании подклассов Address применимы те же самые правила, что и при создании подклассов корневого типа. Таким образом, если мы определим, например, класс USAddress:

```
[DataContract]
public class USAddress : Address { }
```

и присвоим его экземпляр объекту Person:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress =
      new USAddress { Street="Fawcett St", Postcode="02138" };
```

тогда объект p не сможет быть сериализирован. Решение предусматривает либо применение атрибута KnownType к Address:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

либо сообщение экземпляру DataContractSerializer о классе USAddress во время конструирования:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(Сообщать о классе Address нет необходимости, потому что он является объявленным типом члена HomeAddress.)

Предохранение объектных ссылок

Класс NetDataContractSerializer всегда предохраняет эквивалентность ссылок. Класс DataContractSerializer этого не делает без специального запроса.

Другими словами, если на один и тот же объект имеются ссылки в двух разных местах, то DataContractSerializer обычно записывает его дважды. Таким образом, если модифицировать предыдущий пример, чтобы класс Person также хранил рабочий адрес:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

и затем сериализировать его экземпляр, как показано ниже:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

тогда в XML можно будет увидеть, что те же самые детали адреса встречаются два раза:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

При последующей десериализации приведенных данных XML члены `WorkAddress` и `HomeAddress` будут разными объектами. Преимущество такой системы связано с тем, что она позволяет сохранить разметку XML простой и совместимой со стандартами. Недостатки системы включают большой размер данных XML, утерю ссылочной целостности и невозможность справляться с циклическими ссылками.

Затребовать ссылочную целостность можно за счет конструирования экземпляра `DataContractSerializer` следующим образом:

```
var settings = new DataContractSerializerSettings {
PreserveObjectReferences = true };
var ds = new DataContractSerializer (typeof (Person), settings);
```

Можно также указывать максимальное количество объектных ссылок, которые сериализатор должен отслеживать, устанавливая свойство `MaxItemsInObjectGraph` объекта `settings`. Если это количество превышено, тогда сериализатор генерирует исключение (препятствуя атаке типа отказа в обслуживании через злонамеренно сконструированный поток данных).

Вот как затем будут выглядеть данные XML для объекта `Person` с одинаковыми домашним и рабочим адресами:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
   xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
      z:Id="1">
  <Age>30</Age>
```

```
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

Платой будет сокращение возможностей взаимодействия (обратите внимание на патентованное пространство имен для атрибутов `Id` и `Ref`).

Устойчивость к версиям

Данные-члены можно добавлять и удалять, не нарушая прямой или обратной совместимости. По умолчанию десериализаторы на основе контрактов данных обладают следующими особенностями:

- пропускают данные, для которых в типе отсутствует атрибут `[DataMember]`;
- не сообщают, если в потоке сериализации отсутствуют данные, для которых в типе предусмотрен атрибут `[DataMember]`.

Вместо пропуска нераспознанных данных десериализатор можно инструктировать о необходимости сохранять нераспознанные данные-члены в "черном ящике" и воспроизводить их позже, когда тип будет сериализироваться повторно. Это позволяет корректно обходиться с данными, которые были сериализированы более поздней версией типа. Для активизации такой возможности необходимо реализовать интерфейс `IExtensibleDataObject`. Указанный интерфейс на самом деле можно было назвать поставщиком "черного ящика" (скажем, "IBlackBoxProvider"). Он требует реализации единственного свойства, предназначенного для получения/установки "черного ящика":

```
[DataContract] public class Person : IExtensibleDataObject {
  [DataMember] public string Name;
  [DataMember] public int Age;

  ExtensionDataObject IExtensibleDataObject.ExtensionData
   { get; set; }
}
```

Обязательные члены

Если член является жизненно важным для типа, то с помощью аргумента `IsRequired` можно затребовать его обязательное присутствие:

```
[DataMember (IsRequired=true)] public int ID;
```

Если такой член отсутствует, тогда при десериализации сгенерируется исключение.

Упорядочение членов

Сериализаторы на основе контрактов данных крайне придирчивы к порядку следования членов. На самом деле десериализаторы *пропускают любые члены, которые считаются неупорядоченными.*

При сериализации члены записываются в описанном далее порядке.

1. Порядок от базового класса к подклассу.
2. Порядок от низких значений `Order` к высоким значениям `Order` (для данных-членов с установленным аргументом `Order`).
3. Алфавитный порядок (с использованием *ординального* сравнения строк).

Таким образом, в предшествующих примерах `Age` будет находиться перед `Name`. В следующем примере `Name` находится перед `Age`:

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

При наличии у класса `Person` базового класса сначала сериализировались бы все данные-члены базового класса.

Основная причина для указания порядка — соответствие определенной схеме XML. Порядок следования XML-элементов совпадает с порядком следования данных-членов.

Если возможность взаимодействия с чем-то другим не нужна, то простейший подход заключается в том, чтобы *не* указывать значения `Order` для членов и полагаться чисто на упорядочение по алфавиту. Тогда в случае добавления и удаления членов расхождение между сериализацией и десериализацией никогда не возникнет. Единственная ситуация, когда произойдет нестыковка — перемещение члена между базовым классом и подклассом.

Пустые значения и `null`

Существуют два способа обработки члена с пустым значением или `null`.

1. Явно записать пустое значение или `null` (стандартный способ).
2. Не помещать член в выходные данные сериализации.

В XML явное значение `null` выглядит так:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Записывание членов с пустыми значениями или `null` может приводить к бесполезному расходу пространства, особенно в случае типов с множеством полей или свойств, которые обычно остаются пустыми. Более важно то, что может возникнуть необходимость следовать XML-схеме, которая ожидает

применения необязательных элементов (скажем, minOccurs="0"), а не значений nil.

Проинструктировать сериализатор о том, что он не должен выдавать данные-члены для пустых значений или null, можно следующим образом:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

Член Name будет пропущен, если его значение равно null, а член Age — если его значением является 0 (стандартное значение для типа int). В случае объявления Age с типом int, допускающим null, данный член будет пропускаться тогда (и только тогда), когда его значением окажется null.

На заметку! Десериализатор на основе контрактов данных при восстановлении объекта пропускает конструкторы и инициализаторы полей типа. Это позволяет пропускать данные-члены, как было описано выше, не разрушая поля, которым присвоены нестандартные значения, из-за выполнения инициализатора или конструктора. В целях иллюстрации предположим, что в качестве стандартного значения для Age в Person выбрано 30:

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

А теперь представим, что мы создаем объект Person, явно переустанавливаем его поле Age из 30 в 0 и затем сериализируем его. Выходные данные не будут включать Age, поскольку 0 — стандартное значение для типа int. В итоге при десериализации поле Age будет проигнорировано и останется со своим стандартным значением, которое по счастливой случайности равно 0, учитывая, что инициализаторы полей и конструкторы были пропущены.

Контракты данных и коллекции

Сериализаторы на основе контрактов данных могут сохранять и повторно заполнять любую перечислимую коллекцию. Например, пусть класс Person определен со списком List<> адресов:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Ниже показан результат сериализации объекта Person с двумя адресами:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Обратите внимание, что сериализатор не кодирует информацию о конкретном *mune* коллекции, которую он сериализирует. Если бы поле `Addresses` имело тип `Address[]`, то вывод был бы идентичным. Это позволяет изменять тип коллекции между сериализацией и десериализацией, не приводя к ошибке.

Тем не менее, иногда требуется, чтобы коллекция была более специфичного типа, чем указанный. Самым крайним примером является вариант с интерфейсами:

```
[DataMember] public IList<Address> Addresses;
```

Данный член сериализируется корректно (как и ранее), но во время десериализации возникнет проблема. У десериализатора нет никакой возможности узнать, объект какого конкретного типа должен быть создан, так что он выбирает простейший вариант — массив. Десериализатор придерживается такой стратегии, даже когда вы инициализируете поле с другим конкретным типом:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Вспомните, что десериализатор пропускает инициализаторы полей.) Обойти проблему можно, сделав член закрытым полем и добавив открытое свойство для доступа к нему:

```
[DataMember (Name="Addresses")] List<Address> _addresses;
public IList<Address> Addresses { get { return _addresses; } }
```

Вполне вероятно, что в нетривиальных приложениях вы будете в любом случае использовать свойства в подобной манере. Единственный необычный момент здесь касается пометки в качестве члена данных закрытого поля, а не открытого свойства.

Элементы коллекции, являющиеся подклассами

Сериализатор прозрачно обрабатывает элементы коллекции, являющиеся подклассами. Допустимые подтипы должны объявляться точно так же, как если бы они применялись в любом другом месте:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}

public class USAddress : Address { }
```

Добавление `USAddress` в список адресов `Person` приводит к генерации XML следующего вида:

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Настройка имен коллекции и элементов

В случае создания подкласса от самого класса коллекции XML-имя, которое используется для описания каждого элемента, можно настраивать, присоединяя атрибут `CollectionDataContract`:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Вот результат:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

Атрибут `CollectionDataContract` также позволяет указывать аргументы `Namespace` и `Name`. Последний аргумент не применяется, когда коллекция сериализирована как свойство другого объекта (вроде того, что было в приведенном примере), но используется в случае, если коллекция сериализируется как корневой объект.

Атрибут `CollectionDataContract` можно также применять для управления сериализацией словарей:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                         ValueName="Number")]
```

```
public class PhoneNumberList : Dictionary <string, string> { }
[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Вот как это форматируется:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Расширение контрактов данных

В настоящем разделе будет показано, как можно расширять функциональные возможности сериализатора на основе контрактов данных с помощью ловушек сериализации, атрибута `[Serializable]` и интерфейса `IXmlSerializable`.

Ловушки сериализации и десериализации

Можно затребовать, чтобы до или после сериализации выполнялся специальный метод, пометив его одним из перечисленных ниже атрибутов.

- `[OnSerializing]`. Указывает метод для вызова *до* сериализации.
- `[OnSerialized]`. Указывает метод для вызова *после* сериализации.

Аналогичные атрибуты поддерживаются и для десериализации.

- `[OnDeserializing]`. Указывает метод для вызова *до* десериализации.
- `[OnDeserialized]`. Указывает метод для вызова *после* десериализации.

Специальный метод должен иметь единственный параметр типа `StreamingContext`. Данный параметр обязателен для обеспечения согласованности с механизмом двоичной сериализации; сериализатор на основе контрактов данных его не использует.

Атрибуты `[OnSerializing]` и `[OnDeserialized]` пригодны для обработки членов, которые выходят за рамки возможностей механизма сериализации на основе контрактов данных, таких как коллекция, несущая дополнительную полезную нагрузку или не реализующая стандартные интерфейсы. Ниже демонстрируется базовый подход:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;

  [DataMember (Name="Addresses")]
  SerializationFriendlyType _serializationFriendlyAddresses;

  [OnSerializing]
  void PrepareForSerialization (StreamingContext sc)
  {
    // Копировать Addresses в _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // Копировать _serializationFriendlyAddresses в Addresses
    // ...
  }
}
```

Метод `[OnSerializing]` может также применяться для условной сериализации полей:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Вспомните, что десериализаторы на основе контрактов данных пропускают инициализаторы полей и конструкторы. Метод `[OnDeserializing]` действует в качестве псевдоконструктора для десериализации, и он удобен для инициализации полей, исключенных из сериализации:

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

Если бы не метод Init, то поле _editable в десериализированном экземпляре Test содержало бы значение false, несмотря на две попытки сделать его равным true.

Методы, декорированные упомянутыми четырьмя атрибутами, могут быть закрытыми. Если должны участвовать подтипы, то они могут определять собственные методы с теми же самыми атрибутами, и такие методы также будут выполняться.

Возможность взаимодействия с помощью [Serializable]

Сериализатор на основе контрактов данных может также сериализировать типы, помеченные с помощью атрибутов и интерфейсов механизма двоичной сериализации. Такая возможность важна, поскольку поддержка механизма двоичной сериализации широко использовалась в коде, написанном до выхода версии .NET Framework 3.0, включая и саму платформу .NET!

На заметку! Пометить тип как сериализируемый для механизма двоичной сериализации можно посредством:

- атрибута [Serializable];
- реализации интерфейса ISerializable.

Возможность двоичного взаимодействия полезна при сериализации существующих типов, а также новых типов, которые нуждаются в поддержке обоих механизмов. Она также предоставляет еще одно средство расширения возможностей сериализатора на основе контрактов данных, потому что интерфейс ISerializable механизма двоичной сериализации является более гибким, чем атрибуты контрактов данных. К сожалению, сериализатор на основе контрактов данных неэффективен в том, как он форматирует данные, добавленные через ISerializable.

В типе, для которого желательно извлечь лучшее из двух технологий, нельзя определять атрибуты для обоих механизмов. Иначе образовалась бы проблема для таких типов, как string и DateTime, которые по историческим причинам не могут быть отделены от атрибутов механизма двоичной сериализации. Сериализатор на основе контрактов данных обходит указанную проблему за счет фильтрации базовых типов и их обработки специальным образом. Ко всем другим типам, помеченным для двоичной сериализации, сериализатор на основе контрактов данных применяет правила, похожие на те, которые использовал бы механизм двоичной сериализации. Это означает, что он учитывает атрибуты вроде NonSerialized или обращается к интерфейсу ISerializable в случае его реализации. Он не *переходит* на сам механизм двоичной сериализации, гарантируя тем самым, что вывод форматируется в таком же стиле, как если бы применялись атрибуты контрактов данных.

На заметку! Типы, предназначенные для сериализации с помощью двоичного механизма, ожидают предохранения объектных ссылок. Активизировать его можно через класс DataContractSerializer (или за счет использования класса NetDataContractSerializer).

Правила для регистрации известных типов также применяются к объектам и подобъектам, которые сериализированы посредством механизма двоичной сериализации.

В следующем примере демонстрируется класс с членом [Serializable]:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}

[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Вот результат его сериализации:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Если бы класс Address реализовывал интерфейс ISerializable, то результат оказался бы не настолько эффективно сформатированным:

```
<MailingAddress>
  <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">str</Street>

  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">pcode</Postcode>
</MailingAddress>
```

Возможность взаимодействия с помощью IXmlSerializable

Ограничение сериализатора на основе контрактов данных заключается в том, что он предоставляет лишь небольшой контроль над структурой XML. На самом деле это может быть выгодно для приложений WCF, т.к. упрощается согласование инфраструктуры со стандартными протоколами обмена сообщениями.

Если необходим точный контроль над данными XML, тогда можно реализовать интерфейс IXmlSerializable и затем использовать классы XmlReader и XmlWriter для чтения и записи разметки XML вручную. Сериализатор на основе контрактов данных позволяет поступать так только с типами, для которых подобный уровень контроля является обязательным.

Двоичный сериализатор

Механизм двоичной сериализации сохраняет и восстанавливает объекты с полной точностью типов и ссылок и может использоваться для решения таких задач, как сохранение и восстановление объектов с диска. Двоичный сериализатор хорошо автоматизирован и способен обрабатывать сложные графы объектов с минимальным вмешательством. Однако он не доступен в приложениях Магазина Microsoft.

Сделать тип поддерживающим двоичную сериализацию можно двумя путями. Первый из них основан на атрибутах, а второй предусматривает реализацию интерфейса `ISerializable`. Добавление атрибутов проще, но реализация `ISerializable` обеспечивает более высокую гибкость. Интерфейс `ISerializable` обычно реализуется для достижения следующих целей:

- динамическое управление тем, что сериализируется;
- удобство создания подклассов сериализируемого типа другими потребителями.

На заметку! С использованием двоичного сериализатора связаны значительные проблемы в плане безопасности, которые обсуждаются по ссылке `https://aka.ms/binaryformatter`.

Начало работы

Тип делается сериализируемым за счет применения единственного атрибута:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

Атрибут `[Serializable]` инструктирует сериализатор о необходимости включения всех полей в данном типе, что касается как закрытых, так и открытых полей (но не свойств). Каждое поле само должно допускать сериализацию, иначе сгенерируется исключение. Примитивные типы .NET, такие как `string` и `int`, поддерживают сериализацию (подобно многим другим типам .NET).

На заметку! Атрибут `Serializable` не наследуется, а потому подклассы не будут автоматически сериализируемыми, если их не пометить указанным атрибутом.

Чтобы сериализировать экземпляр `Person`, необходимо создать экземпляр `BinaryFormatter` и вызвать метод `Serialize`.

На заметку! В .NET Framework также предлагается тип `SoapFormatter`, который можно использовать для генерации выходных данных XML, совместимых с SOAP. По сравнению с `BinaryFormatter` он менее функционален и не поддерживает ни обобщенные типы, ни фильтрацию посторонних данных, которая требуется для сериализации, устойчивой к версиям.

Следующий код выполняет сериализацию объекта `Person` с помощью `BinaryFormatter`:

```
Person p = new Person() { Name = "George", Age = 25 };
IFormatter formatter = new BinaryFormatter();
using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

Все данные, необходимые для воссоздания объекта `Person`, записываются в файл `serialized.bin`. Метод `Deserialize` восстанавливает объект:

```
using (FileStream s = File.OpenRead ("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p.Age);     // George 25
}
```

На заметку! При воссоздании объектов десериализатор пропускает все конструкторы. "За кулисами" для выполнения такой работы он вызывает метод `FormatterServices.GetUninitializedObject`. Его можно вызывать напрямую для реализации черновых шаблонов проектирования.

Сериализированные данные включают полные сведения о типе и сборке, поэтому если попытаться привести результат десериализации к совпадающему типу `Person` из другой сборки, то возникнет ошибка. Десериализатор восстанавливает объектные ссылки полностью в их исходном состоянии. Это касается и коллекций, которые трактуются просто как сериализируемые объекты, похожие на любые другие (все коллекции, определенные в пространствах имен `System.Collections.*`, помечены как сериализируемые).

На заметку! Механизм двоичной сериализации способен обрабатывать крупные и сложные графы объектов, не требуя специальной поддержки (кроме обеспечения возможности сериализации всех участвующих членов). Единственный момент, к которому следует относиться осторожно — тот факт, что производительность сериализатора снижается пропорционально количеству ссылок в графе объектов. В итоге может возникнуть проблема на сервере Remoting, который должен обрабатывать много параллельных запросов.

Атрибуты двоичной сериализации

[NonSerialized]

По умолчанию сериализируются *все* поля. Поля, которые сериализировать нежелательно, вроде тех, что используются для временных расчетов или для хранения файловых и оконных дескрипторов, потребуется пометить с помощью атрибута `[NonSerialized]`:

```
[Serializable] public sealed class Person
{
  public string Name;
  [NonSerialized] public int Age;
}
```

Атрибут `[NonSerialized]` инструктирует сериализатор о том, что член `Age` должен быть проигнорирован.

На заметку! Несериализированные члены при десериализации всегда получают пустое значение или `null`, даже если инициализаторы полей или конструкторы устанавливают их по-другому.

[OnDeserializing]

Метод, помеченный атрибутом `[OnDeserializing]`, запускается непосредственно перед десериализацией и действует как своего рода конструктор. Это может быть важно, поскольку двоичный десериализатор пропускает все нормальные конструкторы и инициализаторы полей.

В следующем примере определяется поле по имени `Valid`, которое с помощью атрибута `[NonSerialized]` исключается из сериализации:

```
public sealed class Person
{
  public string Name;
  [NonSerialized] public bool Valid = true;

  public Person() => Valid = true;
}
```

Десериализированный объект `Person` никогда не будет иметь значение `true` в своем поле `Valid` — невзирая на установку `Valid` в `true` конструктором и инициализатором поля. Решить проблему можно, написав специальный "конструктор" десериализации:

```
[OnDeserializing]
void OnDeserializing (StreamingContext context) => Valid = true;
```

[OnDeserialized]

Метод, помеченный атрибутом `[OnDeserialized]`, запускается сразу после десериализации. Это может быть удобно для обновления вычисляемых полей и в сочетании с атрибутом `[OnSerializing]`, который рассматривается далее.

[OnSerializing] и [OnSerialized]

С помощью атрибутов `[OnSerializing]` и `[OnSerialized]` помечаются методы, подлежащие выполнению до и после сериализации.

Атрибут `[OnSerializing]` удобен для заполнения поля, которое используется только для сериализации. В целях иллюстрации предположим, что следующий класс нужно сделать сериализируемым:

```
class Foo
{
  public XDocument Xml;
}
```

Сложность в том, что тип `XDocument` (из пространства имен `System.Xml.Linq`) сам не является сериализируемым. Решить проблему можно, применив атрибут `[NonSerialized]` к полю `Xml` и затем определив метод `[OnSerializing]`, который записывает содержимое `XDocument` в строковое поле (поддающееся сериализации):

```
[Serializable]
class Foo
{
  [NonSerialized]
  public XDocument Xml;

  string _xmlString;  // Используется только для сериализации

  [OnSerializing]
  void OnSerializing (StreamingContext context)
    => _xmlString = Xml.ToString();
}
```

Финальным шагом будет воссоздание `XDocument` при десериализации, для чего добавляется метод `[OnDeserialized]`:

```
[OnDeserialized]
void OnDeserialized (StreamingContext context)
  => Xml = XDocument.Parse (_xmlString);
```

[OptionalField] и поддержка версий

Добавление или удаление полей не нарушает совместимость с уже сериализированными данными: десериализатор пропускает данные, для которых не было найдено соответствующее поле. При добавлении поля можно применить показанный ниже атрибут как напоминание о том, что оно может отсутствовать в данных, сериализированных более старой версией программного обеспечения:

```
[Serializable] public sealed class Person
{
  public string Name;
  [OptionalField (VersionAdded = 2)] public DateTime DateOfBirth;
}
```

Атрибут служит в качестве документации и не оказывает влияния на семантику сериализации.

На заметку! Если надежность поддержки версий важна, тогда избегайте переименования полей и ретроспективного добавления атрибута `NonSerialized`. Никогда не изменяйте типы полей.

Двоичная сериализация с помощью `ISerializable`

Реализация интерфейса `ISerializable` предоставляет типу полный контроль над прохождением его двоичной сериализации и десериализации.

Вот как выглядит определение интерфейса `ISerializable`:

```
public interface ISerializable
{
  void GetObjectData (SerializationInfo info,
                      StreamingContext context);
}
```

Метод `GetObjectData` запускается при сериализации; его работа заключается в заполнении объекта `SerializationInfo` (словарь пар "имя/значение") данными из всех полей, которые необходимо сериализировать. Ниже демонстрируется, как можно реализовать метод `GetObjectData`, сериализирующий два поля с именами `Name` и `DateOfBirth`:

```
public virtual void GetObjectData (SerializationInfo info,
                                   StreamingContext context)
 {
   info.AddValue ("Name", Name);
   info.AddValue ("DateOfBirth", DateOfBirth);
 }
```

В приведенном примере было решено именовать каждый элемент согласно полю, которому он соответствует. Поступать так вовсе не обязательно; может применяться любое имя при условии, что при десериализации используется точно такое же имя. Сами значения могут иметь любой сериализируемый тип; при необходимости будет выполняться рекурсивная сериализация. В словаре разрешено хранить значения `null`.

На заметку! Неплохо объявить метод `GetObjectData` как `virtual` — если только класс не является `sealed`. Это позволит подклассам расширять сериализацию без необходимости в повторной реализации интерфейса `ISerializable`.

Класс `SerializationInfo` также содержит свойства, которые можно применять для управления типом и сборкой, куда экземпляр должен быть десериализирован.

В дополнение к реализации интерфейса `ISerializable` тип, управляющий собственной сериализацией, должен предоставить конструктор десериализации, который принимает такие же два параметра, как и конструктор `GetObjectData`. Конструктор может быть объявлен с любым уровнем доступа — исполняющая среда все равно его найдет. Обычно он объявляется как `protected`, так что подклассы могут его вызывать.

В следующем примере мы объявляем классы `Player` и `Team`, соблюдая принципы неизменяемости (все предназначено только для чтения). Но из-за того, что неизменяемые коллекции не являются сериализируемыми, необходимо получить контроль над процессом сериализации, реализуя интерфейс `ISerializable`:

```
[Serializable] public class Player
{
  public readonly string Name;
  public Player (string name) => Name = name;
}
[Serializable] public class Team : ISerializable
{
  public readonly string Name;
  public readonly ImmutableList<Player> Players;  // Не является
                                                  // сериализируемым!
  public Team (string name, params Player[] players)
  {
    Name = name;
    Players = players.ToImmutableList();
  }
  // Сериализировать объект:
  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    // Преобразовать Players в обычный сериализируемый массив:
    si.AddValue ("PlayerData", Players.ToArray());
  }
  // Десериализировать объект:
  protected Team (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    // Десериализировать Players в массив в соответствии
    // с сериализацией:
    Player[] p = (Player[])si.GetValue("PlayerData",typeof (Player[]));
    // Создать из массива экземпляр неизменяемого списка:
    Players = p.ToImmutableList();
  }
}
```

(Задачу можно было бы также решить с использованием атрибутов `[OnSerializing]` и `[OnDeserialized]`, которые обсуждались ранее.)

Для широко используемых типов в классе `SerializationInfo` есть типизированные методы `Get*`, такие как `GetString`, предназначенные для упрощения реализации конструкторов десериализации. В случае указания имени, для которого данные не существуют, генерируется исключение. Чаще всего это происходит, когда имеется несовпадение версий в коде, выполняющем сериализацию и десериализацию. Например, вы добавили дополнительное поле и не подумали о последствиях, которые вызовет десериализация старого экземпляра.

Чтобы обойти такую проблему, можно поступить следующим образом:

- добавить обработку исключений к коду, который извлекает член, добавленный в последней версии;
- реализовать собственную систему нумерации версий, например:

```
public string MyNewField;
public virtual void GetObjectData (SerializationInfo si,
                                   StreamingContext sc)
{
  si.AddValue ("_version", 2);
  si.AddValue ("MyNewField", MyNewField);
  ...
}
protected Team (SerializationInfo si, StreamingContext sc)
{
  int version = si.GetInt32 ("_version");
  if (version >= 2) MyNewField = si.GetString ("MyNewField");
  ...
}
```

Создание подклассов из сериализируемых классов

В предшествующих примерах классы, полагающиеся на атрибуты для сериализации, запечатывались с помощью `sealed`. Чтобы понять причину, рассмотрим следующую иерархию классов:

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}
[Serializable] public sealed class Student : Person
{
  public string Course;
}
```

В этом примере классы `Person` и `Student` являются сериализируемыми, и оба они задействуют стандартное поведение сериализации исполняющей среды, т.к. ни один из них не реализует интерфейс `ISerializable`.

Теперь предположим, что разработчик класса `Person` по какой-то причине решил реализовать интерфейс `ISerializable` и предоставить конструктор десериализации, чтобы управлять сериализацией `Person`. Новая версия `Person` может иметь такой вид:

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;
```

```
  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("Age", Age);
  }
  protected Person (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    Age = si.GetInt32 ("Age");
  }
  public Person() {}
}
```

Несмотря на возможность работы с экземплярами `Person`, внесенное изменение нарушает сериализацию экземпляров `Student`. Сериализация экземпляра `Student` выглядит успешной, однако поле `Course` в типе `Student` не сохраняется в потоке, поскольку реализации метода `ISerializable.GetObjectData` в классе `Person` ничего не известно о членах типа, производного от `Student`. Вдобавок десериализация экземпляров `Student` генерирует исключение, потому что исполняющая среда ищет (и безуспешно) конструктор десериализации в `Student`.

Решение изложенной проблемы предусматривает реализацию интерфейса `ISerializable` с самого начала для сериализируемых классов, которые являются открытыми и незапечатанными. (Для классов `internal` это не настолько важно, т.к. подклассы при необходимости можно легко модифицировать позже.)

Если мы начинаем с написания класса `Person`, как было в предыдущем примере, тогда класс `Student` должен быть реализован следующим образом:

```
[Serializable]
public class Student : Person
{
  public string Course;
  public override void GetObjectData (SerializationInfo si,
                                      StreamingContext sc)
  {
    base.GetObjectData (si, sc);
    si.AddValue ("Course", Course);
  }
  protected Student (SerializationInfo si, StreamingContext sc)
    : base (si, sc)
  {
    Course = si.GetString ("Course");
  }
  public Student() {}
}
```

27

Компилятор Roslyn

Сам компилятор C#, известный под названием Roslyn, написан полностью на языке C# и доступен в виде набора модульных библиотек. Ссылаясь на такие библиотеки, функциональность компилятора можно эксплуатировать многими способами помимо компиляции исходного кода в сборку. Например, можно писать инструменты статического анализа и рефакторинга кода, редакторы с подсветкой синтаксиса и автозавершением кода, а также подключаемые модули для Visual Studio, которые воспринимают код C#.

Загрузить библиотеки Roslyn можно с помощью диспетчера NuGet, причем предусмотрены пакеты как для C#, так и для Visual Basic. Поскольку оба языка опираются на определенную архитектуру, существуют общие зависимости. Идентификатором пакета NuGet для библиотек компилятора C# является `Microsoft.CodeAnalysis.CSharp`.

Веб-сайт GitHub для Roslyn (`https://github.com/dotnet/roslyn`) содержит также документацию, примеры и пошаговые демонстрации анализа кода и рефакторинга.

Архитектура Roslyn

Архитектура Roslyn разделяет компиляцию на три фазы.

- Разбор кода в синтаксические деревья (*синтаксический* уровень).
- Привязка идентификаторов к символам (*семантический* уровень).
- Выпуск кода на промежуточном языке (Intermediate Language — IL).

На первой фазе *синтаксический анализатор* читает код C# и выдает *синтаксические деревья*. Синтаксическое дерево — это объектная модель документа (Document Object Model — DOM), которая описывает исходный код в древовидной структуре.

На второй фазе происходит *статическое связывание* C#. Компилятор читает ссылки на сборки и выясняет, например, что "Console" относится к `System.Console` из `System.Console.dll`. Частью такого процесса также являются распознавание перегруженных версий и выведение типов.

На третьей фазе производится выходная сборка. Если вы планируете использовать Roslyn для анализа или рефакторинга кода, то не будете иметь дело с функциональностью третьей фазы.

Редактор Visual Studio применяет выходные данные синтаксического уровня для выделения цветом ключевых слов, строк, комментариев и запрещенного кода (соответственно, синим, красным, зеленым и серым цветами), а выходные данные семантического уровня — для выделения цветом распознанных имен типов (бирюзовым цветом).

Рабочие области

В этой главе мы описываем компилятор и открываемые им функциональные средства. Полезно помнить о наличии дополнительных "уровней" над компилятором, которые называются *рабочими областями* и *средствами*.

Уровень рабочих областей поставляется в NuGet-пакете `Microsoft.CodeAnalysis.CSharp.Workspaces` и предоставляет API-интерфейсы для работы с решениями, проектами и документами. Уровень средств поставляется в NuGet-пакете `Microsoft.CodeAnalysis.CSharp.Features` и включает многочисленные API-интерфейсы для анализа и рефакторинга кода.

Поддержка сценариев

С помощью NuGet-пакета `Microsoft.CodeAnalysis.CSharp.Scripting` можно писать код вроде показанного ниже:

```
int result = (int) await CSharpScript.EvaluateAsync ("1 + 2");
```

"За кулисами" API-интерфейс сценариев компилирует `"1 + 2"` в программу, которую затем выполняет, так что работа со сценариями менее эффективна, чем решение, описанное в главе 19 (см. раздел "Взаимодействие с динамическими языками").

Синтаксические деревья

Синтаксическое дерево — это DOM-модель для исходного кода. Следует отметить, что API-интерфейс синтаксических деревьев полностью отделен от API-интерфейса `System.Linq.Expressions`, обсуждаемого в разделе "Деревья выражений" главы 8, хотя концептуальные сходства имеются. Оба API-интерфейса могут представлять выражения C# в DOM-модели; однако синтаксическое дерево Roslyn обладает перечисленными ниже уникальными характеристиками.

- Оно способно представлять все конструкции языка C#, а не только выражения.
- Оно может включать комментарии, пробельные символы и другую дополнительную синтаксическую информацию, а также достоверно переключаться обратно на первоначальный исходный код.
- Оно поступает с методом `ParseText`, который разбирает исходный код в синтаксическое дерево.

С другой стороны, API-интерфейс `System.Linq.Expressions` обладает следующими уникальными характеристиками.

- Он встроен в .NET, и компилятор C# сам запрограммирован на выпуск типов `System.Linq.Expression` в случае обнаружения лямбда-выражения с преобразованием в тип `Expression<T>` при присваивании.
- Он имеет быстрый и легковесный метод `Compile`, который выпускает делегат. По контрасту семантический уровень, компилирующий синтаксические деревья Roslyn, предлагает только тяжеловесный вариант компиляции полной программы в сборку.

Общей чертой обоих API-интерфейсов является неизменяемость синтаксических деревьев, так что ни один из их элементов не может быть модифицирован после создания. Это значит, что приложения вроде Visual Studio и LINQPad при каждом нажатии вами клавиши в редакторе должны создавать новое синтаксическое дерево для обновления служб подсветки синтаксиса и автозавершения кода. Данный процесс не настолько дорогостоящий, как может показаться, т.к. новое синтаксическое дерево в состоянии повторно использовать большинство элементов из старого синтаксического дерева (как показано в разделе "Трансформация синтаксического дерева" далее в главе). К тому же знание того, что объект не может изменяться, упрощает работу с API-интерфейсом. Неизменяемость также делает возможным более легкое и быстрое распараллеливание, поскольку многопоточный код может безопасно получать доступ ко всем частям синтаксического дерева без блокировок.

Структура `SyntaxTree`

Структура `SyntaxTree` содержит три главных элемента.

- **Узлы.** (Абстрактный класс `SyntaxNode`.) Представляет такие конструкции C#, как выражения, операторы и объявления методов. Узлы всегда имеют, по крайней мере, один дочерний элемент, а потому узел в дереве никогда не может быть листовым. В качестве дочерних элементов узлы могут иметь узлы и лексемы.
- **Лексемы.** (Структура `SyntaxToken`.) Представляет идентификаторы, ключевые слова, операции и знаки пунктуации, которые являются частью исходного кода. Единственный вид дочерних элементов, которые могут иметь лексемы — это ведущая и замыкающая дополнительная синтаксическая информация. Родительским элементом лексемы всегда будет узел.
- **Дополнительная синтаксическая информация.** (Структура `SyntaxTrivia`.) Под дополнительной синтаксической информацией (trivia) понимаются пробельные символы, комментарии, директивы препроцессора и код, который остается неактивным вследствие условной компиляции. Дополнительная синтаксическая информация всегда ассоциирована с лексемой, которая находится непосредственно слева или справа, и доступна через свойства `TrailingTrivia` и `LeadingTrivia` данной лексемы соответственно.

На рис. 27.1 показана структура следующего кода, где узлы изображаются черным, лексемы — серым, а дополнительная синтаксическая информация — белым цветом:

```
Console.WriteLine ("Hello");
```

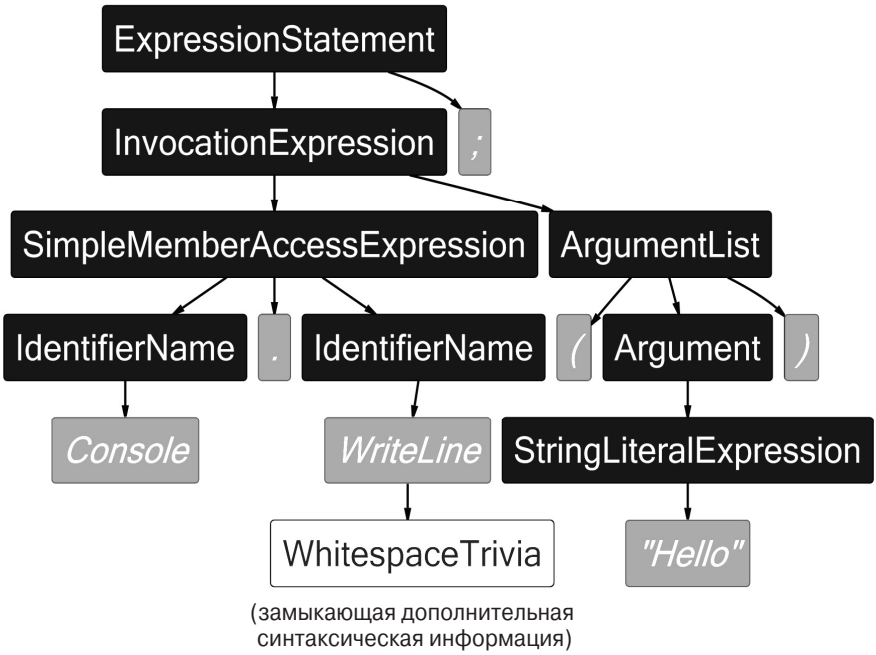

Рис. 27.1. *Синтаксическое дерево*

Класс `SyntaxNode` является абстрактным, а для каждого синтаксического элемента в языке C# предусмотрен его специальный подкласс, такой как `VariableDeclarationSyntax` или `TryStatementSyntax`.

Поскольку `SyntaxToken` и `SyntaxTrivia` — структуры, то все разновидности лексем и дополнительной синтаксической информации представляются с помощью единственного типа. Для различения видов лексем или дополнительной синтаксической информации должно применяться свойство `RawKind` или расширяющий метод `Kind` (как объясняется в следующем разделе).

На заметку! Лучший способ исследования синтаксического дерева предполагает использование визуализатора. В среде Visual Studio имеется загружаемый визуализатор для применения с ее отладчиком, а LINQPad располагает встроенным визуализатором. Инструмент LINQPad автоматически отображает визуализатор для кода в текстовом редакторе, когда вы щелкаете на кнопке Tree (Дерево) в окне вывода. Кроме того, вы можете запросить у LINQPad отображение визуализатора для синтаксического дерева, созданного вами программно, путем вызова метода `DumpSyntaxTree` на объекте дерева (или метода `DumpSyntaxNode` на объекте узла).

Понятие типов узлов

Для отражения результата синтаксического разбора были спроектированы подклассы класса `SyntaxNode`, которые не замечают семантическую информацию о типе/символе, получаемую на более позднем этапе связывания. Например, рассмотрим результат разбора следующего кода:

```
using System;

class Foo : SomeBaseClass
{
  void Test() { Console.WriteLine(); }
}
```

Можно было бы ожидать, что вызов `Console.WriteLine` должен быть представлен классом по имени `MethodCallExpressionSyntax`, но такого класса не существует. Взамен вызов представляется с помощью объекта `InvocationExpressionSyntax`, под которым есть объект `SimpleMemberAccessExpression`. Причина в том, что анализатор не осведомлен о типах, поэтому он не знает, что `Console` является типом, а `WriteLine` — методом. Существует много других возможностей: `Console` могло бы быть свойством класса `SomeBaseClass` или `WriteLine` могло бы быть событием, полем либо свойством какого-то типа делегата. Из синтаксиса можно узнать лишь то, что выполняется доступ к члену (*идентификатор.идентификатор*), за которым следует разновидность *вызова* с нулевым количеством аргументов.

Общие свойства и методы

Узлы, лексемы и дополнительная синтаксическая информация имеют несколько важных общих свойств и методов.

- **Свойство `SyntaxTree`.** Возвращает синтаксическое дерево, к которому принадлежит объект.
- **Свойство `Span`.** Возвращает позицию объекта в исходном коде (см. раздел “Нахождение дочернего элемента по его смещению” далее в главе).
- **Расширяющий метод `Kind`.** Возвращает член перечисления `SyntaxKind`, классифицирующего узел, лексему или дополнительную синтаксическую информацию посредством нескольких сотен значений (например, `IntKeyword`, `CommaToken` и `WhitespaceTrivia`). Одно и то же перечисление `SyntaxKind` охватывает узлы, лексемы или дополнительную синтаксическую информацию.
- **Метод `ToString`.** Возвращает текст (исходный код) для узла, лексемы или дополнительной синтаксической информации. В случае лексем его эквивалентом является свойство `Text`.
- **Метод `GetDiagnostics`.** Возвращает ошибки или предупреждения, сгенерированные во время разбора.

- **Метод `IsEquivalentTo`.** Возвращает `true`, если объект идентичен другому экземпляру узла, лексемы или дополнительной синтаксической информации. Отличия в пробельных символах существенны (чтобы проигнорировать пробельные символы, перед сравнением вызовите метод `NormalizeWhitespace`).

На заметку! Узлы и лексемы также располагают свойством `FullSpan` и методом `ToFullString`. Они принимают во внимание дополнительную синтаксическую информацию, тогда как `Span` и `ToString` — нет.

Расширяющий метод `Kind` — это сокращение для приведения свойства `RawKind`, имеющего тип `int`, к типу `Microsoft.CodeAnalysis.CSharp.SyntaxKind`. Причина, по которой не было просто определено свойство `Kind` типа `SyntaxKind`, связана с тем, что типы лексем и дополнительной синтаксической информации также используются в синтаксических деревьях Visual Basic, имеющих другой тип перечисления для `SyntaxKind`.

Получение синтаксического дерева

Статический метод `ParseText` класса `CSharpSyntaxTree` производит разбор кода C# в объект `SyntaxTree`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");

Console.WriteLine (tree.ToString());

tree.DumpSyntaxTree();    // Отображает визуализатор
                          // синтаксических деревьев в LINQPad
```

Чтобы выполнить такой код в проекте Visual Studio, установите NuGet-пакет `Microsoft.CodeAnalysis.CSharp` и импортируйте следующие пространства имен:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
```

Методу `ParseText` можно дополнительно передавать объект `CSharpParseOptions`, задающий версию языка C#, символы препроцессора и член перечисления `DocumentationMode`, который указывает на то, должны ли быть подвергнуты разбору XML-комментарии (см. раздел “Структурированная дополнительная синтаксическая информация” далее в главе). Также есть возможность указать значение перечисления `SourceCodeKind`. Выбор значения `Script` заставляет анализатор вместо требования полной программы принимать одиночное выражение или оператор (операторы), что поддерживается в Roslyn версии 2 и выше.

Еще один способ получения синтаксического дерева предусматривает вызов метода `CSharpSyntaxTree.Create` с передачей ему объектного графа узлов и лексем. Мы покажем, как создавать такие объекты, в разделе “Трансформация синтаксического дерева” далее в главе.

После разбора дерева можно получить ошибки и предупреждения, вызвав метод GetDiagnostics. (Упомянутый метод можно также вызывать для отдельного узла или лексемы.)

На заметку! Если разбор привел к непредвиденным ошибкам, тогда структура дерева может оказаться не той, которая ожидалась. По этой причине полезно вызывать метод GetDiagnostics, прежде чем двигаться дальше.

Полезная особенность дерева с ошибками заключается в том, что оно будет переключаться обратно на первоначальный текст (с теми же самыми ошибками). В таких случаях анализатор делает все возможное, чтобы предоставить синтаксическое дерево, которое пригодно для семантического уровня, при необходимости создавая "фантомные узлы". Такой подход позволяет инструментам вроде автозавершения кода работать с неполным кодом. (Выяснить, является ли узел фантомным, можно с помощью свойства IsMissing.)

Вызов метода GetDiagnostics на синтаксическом дереве, созданном в предыдущем разделе, указывает на отсутствие ошибок, несмотря на вызов Console.WriteLine без импортирования пространства имен System. Это хороший пример сравнения синтаксического и семантического разборов: программа синтаксически корректна и ошибка не проявится до тех пор, пока мы не соберем все вместе, добавим ссылки на сборки и запросим *семантическую модель*, где происходит связывание.

Обход и поиск в дереве

Структура SyntaxTree действует как оболочка для древовидной структуры. Она имеет ссылку на единственный корневой узел, который можно получить с помощью вызова метода GetRoot:

```
var tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");

SyntaxNode root = tree.GetRoot();
```

Корневой узел программы C# представлен объектом CompilationUnitSyntax:

```
Console.WriteLine (root.GetType().Name); // CompilationUnitSyntax
```

Обход дочерних элементов

Класс SyntaxNode предлагает дружественные к LINQ методы для обхода своих дочерних узлов и лексем. Вот простейшие из них:

```
IEnumerable<SyntaxNode> ChildNodes()
IEnumerable<SyntaxToken> ChildTokens()
```

Из предыдущего примера следует, что наш корневой узел имеет единственный дочерний узел типа ClassDeclarationSyntax:

```
var cds = (ClassDeclarationSyntax) root.ChildNodes().Single();
```

Мы можем выполнить перечисление членов объекта `cds` либо через его метод `ChildNodes`, либо посредством свойства `Members` класса `ClassDeclarationSyntax`:

```
foreach (MemberDeclarationSyntax member in cds.Members)
  Console.WriteLine (member.ToString());
```

с показанным ниже результатом:

```
static void Main() => Console.WriteLine (""Hello"");
```

Существуют также методы `Descendant*`, которые рекурсивно спускаются к дочерним элементам. Реализовать перечисление лексем, составляющих нашу программу, можно следующим образом:

```
foreach (var token in root.DescendantTokens())
  Console.WriteLine ($"{token.Kind(),-30} {token.Text}");
```

Вот результат:

```
ClassKeyword                  class
IdentifierToken               Test
OpenBraceToken                {
StaticKeyword                 static
VoidKeyword                   void
IdentifierToken               Main
OpenParenToken                (
CloseParenToken               )
EqualsGreaterThanToken        =>
IdentifierToken               Console
DotToken                      .
IdentifierToken               WriteLine
OpenParenToken                (
StringLiteralToken            "Hello"
CloseParenToken               )
SemicolonToken                ;
CloseBraceToken               }
EndOfFileToken
```

Обратите внимание на отсутствие пробельных символов в результате. Замена обращения `token.Text` вызовом метода `token.ToFullString` привела бы к получению пробельных символов (и любой другой дополнительной синтаксической информации).

В приведенном ниже коде метод `DescendantNodes` используется при нахождении местоположения узла для нашего объявления метода:

```
var ourMethod = root.DescendantNodes()
           .First (m => m.Kind() == SyntaxKind.MethodDeclaration);
```

Или по-другому:

```
var ourMethod = root.DescendantNodes()
                    .OfType<MethodDeclarationSyntax>()
                    .Single();
```

В последнем примере переменная ourMethod имеет тип MethodDeclarationSyntax, который открывает доступ к полезным свойствам, специфичным для объявлений методов. Скажем, если бы в примере содержалось более одного определения метода, и нужно было найти только метод с именем Main, то можно было бы поступить так:

```
var mainMethod = root.DescendantNodes()
                     .OfType<MethodDeclarationSyntax>()
                     .Single (m => m.Identifier.Text == "Main");
```

Identifier — это свойство класса MethodDeclarationSyntax, которое возвращает лексему, соответствующую идентификатору метода (т.е. его имени). Получить тот же самый результат можно и с бо́льшими усилиями:

```
root.DescendantNodes().First (m =>
  m.Kind() == SyntaxKind.MethodDeclaration &&
  m.ChildTokens().Any (t =>
    t.Kind() == SyntaxKind.IdentifierToken && t.Text == "Main"));
```

В классе SyntaxNode также определены методы GetFirstToken и GetLastToken, которые являются эквивалентами вызовов DescendantTokens().First() и DescendantTokens().Last().

На заметку! Вызов GetLastToken быстрее вызова DescendantTokens().Last(), т.к. он возвращает прямую ссылку, а не производит перечисление по всем потомкам.

Поскольку узлы могут содержать как дочерние узлы, так и лексемы, относительный порядок следования которых имеет значение, то существуют также методы для их перечисления вместе:

```
ChildSyntaxList ChildNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokensAndSelf()
```

(Класс ChildSyntaxList реализует интерфейс IEnumerable<SyntaxNodeOrToken>, одновременно открывая доступ к свойству Count и индексатору для обращения к элементам по позициям.)

Обходить дополнительную синтаксическую информацию можно напрямую из узла с помощью методов GetLeadingTrivia, GetTrailingTrivia и DescendantTrivia. Однако чаще всего вы будете производить доступ к дополнительной синтаксической информации через лексему, к которой она присоединена, посредством свойств LeadingTrivia и TrailingTrivia лексемы. Или же при преобразовании в текст вы могли бы применять метод ToFullString, который включает дополнительную синтаксическую информацию в результат.

Обход родительских элементов

Узлы и лексемы имеют свойство Parent типа SyntaxNode.

Для структуры SyntaxTrivia “родительским элементом” является лексема, доступная через свойство Token.

Узлы также располагают методами, которые поднимаются вверх по дереву; их имена снабжены префиксом `Ancestor`.

Нахождение дочернего элемента по его смещению

Все узлы, лексемы и дополнительная синтаксическая информация имеют свойство `Span` типа `TextSpan` для указания смещений начала и конца в исходном коде. В узлах и лексемах также есть свойство `FullSpan`, которое включает ведущую и замыкающую дополнительную синтаксическую информацию (тогда как свойство `Span` ее не содержит). Тем не менее, свойство `Span` узла включает дочерние узлы и лексемы.

Работа со структурой `TextSpan`

Структура `TextSpan` имеет целочисленные свойства `Start`, `Length` и `End`, которые указывают символьные смещения в исходном коде. В ней также определены такие методы, как `Overlap`, `OverlapsWith`, `Intersection` и `IntersectsWith`. Разница между перекрытием и пересечением сводится к одному символу: два промежутка *перекрываются*, если один начинается до окончания другого (<), тогда как они *пересекаются*, когда просто соприкасаются (<=).

Класс `SyntaxTree` открывает доступ к методу `GetLineSpan`, который преобразует структуру `TextSpan` в строковое и символьное смещение. Метод `GetLineSpan` игнорирует результаты действия любых директив `#line`, присутствующих в исходном коде. Есть также метод `GetMappedLineSpan`, который принимает во внимание упомянутые директивы.

С помощью методов `FindNode`, `FindToken` и `FindTrivia` класса `SyntaxNode` объект-потомок можно искать по позиции. Эти методы возвращают объект-потомок с наименьшим промежутком, который полностью содержит промежуток, указанный при вызове. Существует также метод `ChildThatContainsPosition`, производящий поиск и узлов-потомков, и лексем.

Если поиск дает в результате два узла с идентичными промежутками (обычно дочерний и внучатый узлы), то метод `FindNode` возвратит внешний (родительский) узел. Такое поведение можно изменить, передав методу `getInnermostNodeForTie` значение `true` в дополнительном аргументе.

Методы `Find*` также принимают необязательный параметр `findInsideTrivia` типа `bool`. В случае передачи `true` методы выполняют поиск узлов или лексем также и внутри *структурированной дополнительной синтаксической информации* (см. раздел “Дополнительная синтаксическая информация” далее в главе).

`CSharpSyntaxWalker`

Еще один способ обхода дерева предполагает создание подкласса класса `CSharpSyntaxWalker` с переопределением одного или более из его сотен виртуальных методов.

Следующий класс подсчитывает количество операторов `if`:

```
class IfCounter : CSharpSyntaxWalker
{
  public int IfCount { get; private set; }
  public override void VisitIfStatement (IfStatementSyntax node)
  {
    IfCount++;
    // Вызвать базовый метод, если нужно спуститься
    // к дочерним элементам.
    base.VisitIfStatement (node);
  }
}
```

Вот как его задействовать:

```
var ifCounter = new IfCounter ();
ifCounter.Visit (root);
Console.WriteLine ($"I found {ifCounter.IfCount} if statements");
```

Результат эквивалентен выполнению такого кода:

```
root.DescendantNodes().OfType<IfStatementSyntax>().Count()
```

Написание средства обхода синтаксиса может оказаться легче, чем использование методов `Descendant*` в более сложных случаях, когда необходимо переопределять множество методов (отчасти потому, что C# не обладает возможностью сопоставления по образцу, присущей языку F#).

По умолчанию класс `CSharpSyntaxWalker` посещает только узлы. Чтобы посетить лексемы или дополнительную синтаксическую информацию, потребуется вызвать базовый конструктор со значением перечисления `SyntaxWalkerDepth`, которое указывает желаемую глубину (узел→лексема→дополнительная синтаксическая информация). Затем можно переопределять методы `VisitToken` и `VisitTrivia`:

```
class WhiteWalker : CSharpSyntaxWalker     // Посчитывает
                                           // пробельные символы
{
  public int SpaceCount { get; private set; }
  public WhiteWalker() : base (SyntaxWalkerDepth.Trivia) { }
  public override void VisitTrivia (SyntaxTrivia trivia)
  {
    SpaceCount += trivia.ToString().Count (char.IsWhiteSpace);
    base.VisitTrivia (trivia);
  }
}
```

Если удалить вызов базового конструктора из конструктора `WhiteWalker`, то метод `VisitTrivia` не запустится.

Дополнительная синтаксическая информация

Дополнительная синтаксическая информация предназначена для кода, который после разбора компилятор может почти полностью игнорировать в смысле построения выходной сборки. Сюда входят пробельные символы, комментарии, XML-документация, директивы препроцессора и код, являющийся неактивным из-за условной компиляции.

Обязательные пробельные символы в коде также рассматриваются как дополнительная синтаксическая информация. Хотя она жизненно важна для разбора, после производства синтаксического дерева необходимость в ней отпадает (во всяком случае, со стороны компилятора). Дополнительная синтаксическая информация по-прежнему важна для переключения обратно на первоначальный исходный код.

Дополнительная синтаксическая информация принадлежит лексеме, с которой она соседствует. По соглашению анализатор помещает пробельные символы и комментарии, следующие за лексемой, в конец строки (внутрь замыкающей дополнительной синтаксической информации лексемы). Все, что находится после этого, анализатор трактует как ведущую дополнительную синтаксическую информацию для следующей лексемы. (Существуют исключения для позиций в самом начале и в конце файла.) Если вы создаете лексемы программно (см. раздел "Трансформация синтаксического дерева" далее в главе), то можете помещать пробельные символы в любое место (или вообще не поступать так, если не собираетесь выполнять преобразование обратно в исходный код):

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static /*комментарий*/ void Main() {}
}");

SyntaxNode root = tree.GetRoot();

// Найти лексему ключевого слова static:
var method = root.DescendantTokens().Single (t =>
  t.Kind() == SyntaxKind.StaticKeyword);

// Вывести дополнительную синтаксическую информацию вокруг
// лексемы ключевого слова static:
foreach (SyntaxTrivia t in method.LeadingTrivia)
  Console.WriteLine (new { Kind = "Leading " + t.Kind(),t.Span.Length });

foreach (SyntaxTrivia t in method.TrailingTrivia)
  Console.WriteLine (new { Kind = "Trailing " + t.Kind(),
                                   t.Span.Length });
```

Ниже показан вывод:

```
{ Kind = Leading WhitespaceTrivia, Length = 1 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
{ Kind = Trailing MultiLineCommentTrivia, Length = 11 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
```

Директивы препроцессора

Может показаться странным, что директивы препроцессора считаются дополнительной синтаксической информацией, учитывая значительное воздействие на вывод, которое оказывают некоторые директивы (в частности, директивы условной компиляции).

Причина в том, что директивы препроцессора семантически обрабатываются самим анализатором, т.е. выполнение предварительной обработки — задача анализатора. После этого не остается ничего такого, что компилятор должен явно принимать во внимание (кроме `#pragma`). В целях иллюстрации давайте посмотрим, как анализатор обрабатывает директивы условной компиляции:

```
#define FOO

#if FOO
  Console.WriteLine ("FOO is defined");   // Символ FOO определен
#else
  Console.WriteLine("FOO is not defined"); // Символ FOO не определен
#endif
```

Прочитав директиву `#if FOO`, анализатор знает, что символ `FOO` определен, поэтому следующая за ней строка разбирается обычным образом (как узлы и лексемы), в то время как строка кода, находящаяся за директивой `#else`, разбирается в `DisabledTextTrivia`.

На заметку! При вызове метода `CSharpSyntaxTree.Parse` можно предоставить дополнительные символы препроцессора, сконструировав и передав данному методу экземпляр `CSharpParseOptions`.

Следовательно, благодаря условной компиляции это именно тот текст, который может быть проигнорирован и находится в конце дополнительной синтаксической информации (т.е. неактивный код и сами директивы условной компиляции).

Директива `#line` обрабатывается аналогично в том смысле, что она читается и интерпретируется анализатором. Собранная информация применяется при вызове метода `GetMappedLineSpan` на синтаксическом дереве.

Директива `#region` семантически пуста: единственная роль анализатора заключается в проверке того, что директивам `#region` соответствуют директивы `#endregion`. Директивы `#error` и `#warning` также обрабатываются анализатором и приводят к генерации ошибок и предупреждений, которые можно просмотреть, вызвав метод `GetDiagnostics` на дереве или узле.

Исследование содержимого директив препроцессора может быть в равной степени полезным и для целей, выходящих за рамки производства выходной сборки (скажем, подсветка синтаксиса). Это облегчается посредством *структурированной дополнительной синтаксической информации*.

Структурированная дополнительная синтаксическая информация

Существуют два вида дополнительной синтаксической информации.

- **Неструктурированная дополнительная синтаксическая информация.** Комментарии, пробельные символы и код, который неактивен из-за условной компиляции.
- **Структурированная дополнительная синтаксическая информация.** Директивы препроцессора и XML-документация

Неструктурированная дополнительная синтаксическая информация трактуется исключительно как текст. С другой стороны, структурированная дополнительная синтаксическая информация располагает также и содержимым, которое разбирается в миниатюрное синтаксическое дерево.

Свойство `HasStructure` структуры `SyntaxTrivia` указывает, присутствует ли структурированная дополнительная синтаксическая информация, а метод `GetStructure` возвращает корневой узел для миниатюрного синтаксического дерева:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");

// В LINQPad:
tree.DumpSyntaxTree();    // LINQPad отображает структурированную
      // дополнительную синтаксическую информацию в визуализаторе.

SyntaxNode root = tree.GetRoot();

var trivia = root.DescendantTrivia().First();
Console.WriteLine (trivia.HasStructure); // True
Console.WriteLine (trivia.GetStructure().Kind());
                                          // DefineDirectiveTrivia
```

В случае директив препроцессора можно переходить непосредственно к структурированной дополнительной синтаксической информации, вызывая метод `GetFirstDirective` класса `SyntaxNode`. Имеется также свойство `ContainsDirectives`, предназначенное для указания на наличие синтаксической информации препроцессора:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");
SyntaxNode root = tree.GetRoot();

Console.WriteLine (root.ContainsDirectives);      // True

// directive - это корневой узел структурированной
// дополнительной синтаксической информации:
var directive = root.GetFirstDirective();
Console.WriteLine (directive.Kind());     // DefineDirectiveTrivia
Console.WriteLine (directive.ToString()); // #define FOO

// Если директив было больше, тогда мы можем получить
// их следующим образом:
Console.WriteLine (directive.GetNextDirective());  // (null)
```

После получения узла дополнительной синтаксической информации мы можем привести его к специфичному типу и обращаться к свойствам, как поступали бы с любым другим узлом:

```
var hashDefine = (DefineDirectiveTriviaSyntax) root.
GetFirstDirective();
Console.WriteLine (hashDefine.Name.Text);        // FOO
```

На заметку! Все узлы, лексемы и дополнительная синтаксическая информация имеют свойство `IsPartOfStructuredTrivia`, которое указывает на то, является ли данный объект частью дерева структурированной дополнительной синтаксической информации (т.е. происходит от объекта дополнительной синтаксической информации).

Трансформация синтаксического дерева

С помощью набора методов (большинство из которых представляют собой расширяющие методы) со следующими префиксами можно "модифицировать" узлы, лексемы и дополнительную синтаксическую информацию:

```
Add*
Insert*
Remove*
Replace*
With*
Without*
```

Поскольку синтаксические деревья являются неизменяемыми, все эти методы возвращают новый объект с желаемыми модификациями, оставляя исходный объект незатронутым.

Обработка изменений в исходном коде

Если вы строите, скажем, редактор кода C#, то должны обновлять синтаксическое дерево на основе изменений, внесенных в исходный код. Класс `SyntaxTree` имеет метод `WithChangedText`, выполняющий именно такую работу: он частично разбирает исходный код заново, базируясь на модификациях, которые описаны с помощью экземпляра класса `SourceText` (из пространства имен `Microsoft.CodeAnalysis.Text`).

Чтобы создать экземпляр класса `SourceText`, вызовите его статический метод `From`, передав ему завершенный исходный код. Затем для создания синтаксического дерева можно использовать приведенный ниже код:

```
SourceText sourceText = SourceText.From ("class Program {}");
var tree = CSharpSyntaxTree.ParseText (sourceText);
```

В качестве альтернативы экземпляр класса `SourceText` можно получить для существующего дерева, вызвав его метод `GetText`.

Теперь `sourceText` можно "обновить" с помощью метода `Replace` или `WithChanges`. Например, вот как заменить первые пять символов (`class`) символами `struct`:

```
var newSource = sourceText.Replace (0, 5, "struct");
```

Наконец, можно вызвать метод `WithChangedText` на дереве для его обновления:

```
var newTree = tree.WithChangedText (newSource);
Console.WriteLine (newTree.ToString());  // struct Program {}
```

Создание новых узлов, лексем и дополнительной синтаксической информации с помощью класса `SyntaxFactory`

Статические методы класса `SyntaxFactory` позволяют программно создавать узлы, лексемы и дополнительную синтаксическую информацию, которую можно применять для "трансформации" существующих синтаксических деревьев или для построения новых деревьев с нуля.

Самой трудной частью этого процесса является выяснение того, узлы и лексемы какого вида нужно создавать. Решение предусматривает предварительный разбор желаемого примера кода с целью исследования результатов в визуализаторе синтаксиса. В качестве примера представим, что необходимо создать синтаксический узел для следующего кода:

```
using System.Text;
```

Визуализировать синтаксическое дерево для этого кода в LINQPad можно так:

```
CSharpSyntaxTree.ParseText ("using System.Text;").DumpSyntaxTree();
```

(Мы можем провести разбор кода `using  System.Text;` без ошибок, потому что он допустим как завершенная, хоть и функционально пустая программа. Большинство других фрагментов кода потребуется помещать внутрь определения метода и/или типа, чтобы их разбор стал возможным.)

Результат имеет показанную ниже структуру, в которой нас интересует второй узел (т.е. `UsingDirective` и его потомки):

```
Kind                                  Token Text
==================================    ==========
CompilationUnit (node)
  UsingDirective (node)
    UsingKeyword (token)              using
      WhitespaceTrivia (trailing)
    QualifiedName (node)
      IdentifierName (node)
        IdentifierToken (token)       System
      DotToken (token)                .
      IdentifierName (node)
        IdentifierToken (token)       Text
    SemiColonToken (token)            ;
  EndOfFileToken (token)
```

Начав изнутри, мы обнаруживаем два узла `IdentifierName`, родительским узлом которых является `QualifiedName`, что можно создать следующим образом:

```
QualifiedNameSyntax qualifiedName = SyntaxFactory.QualifiedName (
  SyntaxFactory.IdentifierName ("System"),
  SyntaxFactory.IdentifierName ("Text"));
```

Мы используем перегруженную версию метода `QualifiedName`, принимающую два идентификатора. Она вставляет лексему точки автоматически.

Теперь узел необходимо поместить внутрь `UsingDirective`:

```
UsingDirectiveSyntax usingDirective =
  SyntaxFactory.UsingDirective (qualifiedName);
```

Поскольку мы не указываем лексемы для ключевого слова using или завершающей точки с запятой, такие лексемы создаются и добавляются автоматически. Тем не менее, автоматически создаваемые лексемы не включают пробельные символы. Это не препятствует компиляции, но преобразование дерева в строку даст в результате синтаксически некорректный код:

```
Console.WriteLine(usingDirective.ToFullString()); //usingSystem.Text;
```

Устранить проблему можно вызовом метода NormalizeWhitespace на узле (или одном из его предшественников); такое действие автоматически добавляет дополнительную синтаксическую информацию для пробельных символов (обеспечивая корректность и читабельность). Или же в целях большего контроля пробельные символы можно добавить явно:

```
usingDirective = usingDirective.WithUsingKeyword (
  usingDirective.UsingKeyword.WithTrailingTrivia (
    SyntaxFactory.Whitespace (" ")));
Console.WriteLine(usingDirective.ToFullString());//using System.Text;
```

Для краткости мы задействуем существующую лексему UsingKeyword узла, к которому добавляем замыкающую дополнительную синтаксическую информацию. Мы могли бы создать эквивалентную лексему с бо́льшими усилиями, вызвав SyntaxFactory.Token(SyntaxKind.UsingKeyword).

Финальный шаг заключается в добавлении узла UsingDirective к существующему или новому синтаксическому дереву (или, выражаясь точнее, к корневому узлу дерева). Для этого мы приводим корневой узел существующего дерева к типу CompilationUnitSyntax и вызываем метод AddUsings. Затем мы можем создать новое дерево из трансформированной единицы компиляции:

```
var existingTree = CSharpSyntaxTree.ParseText ("class Program {}");
var existingUnit = (CompilationUnitSyntax) existingTree.GetRoot();
var unitWithUsing = existingUnit.AddUsings (usingDirective);
var treeWithUsing = CSharpSyntaxTree.Create (
  unitWithUsing.NormalizeWhitespace());
```

На заметку! Помните, что все части синтаксического дерева являются неизменяемыми. Вызов AddUsings возвращает новый узел, оставляя исходный узел незатронутым. Игнорирование возвращаемого значения — путь к ошибке!

Так как мы вызвали метод NormalizeWhitespace на единице компиляции, вызов ToString на дереве выдаст синтаксически корректный и читабельный код. В качестве альтернативы мы могли бы добавить к usingDirective явную дополнительную синтаксическую информацию для новой строки:

```
.WithTrailingTrivia (SyntaxFactory.EndOfLine("\r\n\r\n"))
```

Создание единицы компиляции и синтаксического дерева с нуля представляет собой похожий процесс. Простейший подход состоит в том, чтобы начать с пустой единицы компиляции и вызвать на ней метод AddUsings, как мы поступали ранее:

```
var unit = SyntaxFactory.CompilationUnit().AddUsings (usingDirective);
```

Добавить к такой единице компиляции определения типов можно путем их создания в аналогичной манере и вызова метода AddMembers:

```
// Создать простое пустое определение класса:
unit = unit.AddMembers (SyntaxFactory.ClassDeclaration ("Program"));
```

Наконец, дерево можно создать:

```
var tree = CSharpSyntaxTree.Create (unit.NormalizeWhitespace());
Console.WriteLine (tree.ToString());
```

Вот вывод:

```
using System.Text;
class Program{}
```

CSharpSyntaxRewriter

Для более сложных трансформаций синтаксического дерева можно создавать подклассы класса CSharpSyntaxRewriter.

Класс CSharpSyntaxRewriter похож на CSharpSyntaxWalker, который рассматривался ранее (см. раздел "CSharpSyntaxWalker"), за исключением того, что каждый метод Visit* принимает и возвращает синтаксический узел. За счет возвращения сущности, отличающейся от переданной, синтаксическое дерево можно "переписывать".

Например, следующий класс изменяет имена объявлений методов, представляя их символами верхнего регистра:

```
class MyRewriter : CSharpSyntaxRewriter
{
  public override SyntaxNode VisitMethodDeclaration
    (MethodDeclarationSyntax node)
  {
    //"Заменить" идентификатор метода его версией в верхнем регистре:
    return node.WithIdentifier (
      SyntaxFactory.Identifier (
        node.Identifier.LeadingTrivia,
          // Сохранить старую дополнительную синтаксическую информацию
        node.Identifier.Text.ToUpperInvariant(),
        node.Identifier.TrailingTrivia));
         // Сохранить старую дополнительную синтаксическую информацию
  }
}
```

Теперь с классом MyRewriter можно работать:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() { Test(); }
  static void Test() {       }
}");
var rewriter = new MyRewriter();
var newRoot = rewriter.Visit (tree.GetRoot());
Console.WriteLine (newRoot.ToFullString());
```

Ниже показан вывод:

```
class Program
{
  static void MAIN() { Test(); }
  static void TEST() {        }
}
```

Обратите внимание, что вызов `Test` в главном методе не был переименован, потому что мы посещали только *объявления* членов, игнорируя *обращения*. Однако для надежного переименования обращений мы должны быть в состоянии определять, относятся ли вызовы метода `Main` или `Test` к типу `Program`, а не к какому-то другому типу. В такой ситуации одного лишь синтаксического дерева недостаточно; нам также нужна *семантическая модель*.

Объекты компиляции и семантические модели

Объект компиляции заключает в себе синтаксические деревья, ссылки и параметры компиляции. Он служит двум целям:

- разрешить компиляцию в библиотеку либо исполняемый файл (фаза выпуска);
- открыть доступ к семантической модели, которая предлагает информацию о символах (полученную от связывания).

Семантическая модель необходима для реализации таких функциональных средств, как переименование символов или предоставление списков автозавершения кода в редакторе.

Создание объекта компиляции

Независимо от того, заинтересованы вы в выдаче запросов к семантической модели или в проведении полной компиляции, первым шагом будет создание экземпляра `CSharpCompilation` с передачей конструктору (простого) имени сборки, подлежащей созданию:

```
var compilation = CSharpCompilation.Create ("test");
```

Простое имя сборки важно, даже если вы не планируете выпускать сборку, поскольку оно формирует часть удостоверения типов внутри объекта компиляции.

По умолчанию предполагается, что вы хотите создать библиотеку. Другой вид вывода (оконный исполняемый файл, консольный исполняемый файл и т.д.) можно указать следующим образом:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication));
```

Конструктор класса `CSharpCompilationOptions` имеет более десятка необязательных параметров для опций, которые можно передавать компилятору. Например, чтобы включить оптимизации компилятора и назначить сборке строгое имя, нужно поступить так:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication,
    optimizationLevel:OptimizationLevel.Release));
```

Затем мы будем добавлять синтаксические деревья. Каждое синтаксическое дерево соответствует "файлу", который должен быть включен в объект компиляции:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (""Hello"");
}");
compilation = compilation.AddSyntaxTrees (tree);
```

В заключение понадобится сослаться на сборки времени выполнения .NET. Поскольку сложно в точности определить, какое сочетание сборок потребуется, легче всего сослаться на все сборки. Следующий код возвращает все сборки времени выполнения .NET (плюс любые сборки, на которые имеет ссылки вызывающее приложение):

```
string trustedAssemblies = (string)AppContext.GetData
                      ("TRUSTED_PLATFORM_ASSEMBLIES");
string[] trustedAssemblyPaths =
       trustedAssemblies.Split(Path.PathSeparator);
```

На заметку! Обратите внимание, что код возвращает сборки времени выполнения, которые специфичны для текущей платформы и версии .NET. Если вы планируете использовать Roslyn для компиляции библиотек, которые будут корректно работать под управлением разных платформ и версий .NET, тогда взамен должны применять ссылочные сборки. Ссылочные сборки доступны в NuGet-пакете `Microsoft.NETCore.app.ref` (для .NET Core), `Microsoft.AspNetCore.App.ref` (для ASP.NET Core) и `Microsoft.WindowsDesktop.app.ref` (для Windows Forms/WPF).

Затем ссылки можно добавить к процессу компиляции:

```
var references = trustedAssemblyPaths.Select
  (path => MetadataReference.CreateFromFile (path));
compilation = compilation.AddReferences (references);
```

Вызов метода `MetadataReference.CreateFromFile` приводит к чтению содержимого сборки в память, но без применения обычной рефлексии. Взамен метод использует высокопроизводительное переносимое средство чтения сборок (`System.Reflection.Metadata`), которое устраняет необходимость в создании объекта `Assembly`. (Создание объекта `Assembly` может оказаться медленным, в результате чего файл сборки будет блокироваться до тех пор, пока процесс не завершится.)

На заметку! Экземпляр реализации `PortableExecutableReference`, получаемый из метода `MetadataReference.CreateFromFile`, может иметь значительный отпечаток в памяти, так что будьте осторожны и удерживайте

только ссылки, которые нужны. Кроме того, если вы обнаружили, что неоднократно создаете ссылки на одну и ту же сборку, то полезно обдумать применение кеша (идеальным будет кеш, хранящий слабые ссылки).

Все действия можно выполнить за один шаг, вызвав перегруженную версию метода `CSharpCompilation.Create`, которая принимает синтаксические деревья, ссылки и параметры. Или же можно использовать текучий синтаксис в единственном выражении:

```
var compilation = CSharpCompilation.Create ("...")
  .WithOptions (...)
  .AddSyntaxTrees (...)
  .AddReferences (...);
```

Диагностика

Компиляция может генерировать ошибки и предупреждения, даже если в синтаксических деревьях отсутствуют ошибки. Примером может быть недостающее импортирование пространства имен, опечатка при ссылке на имя типа или члена и неудавшееся выведение параметра типа. Для получения ошибок и предупреждений понадобится вызвать метод `GetDiagnostics` на объекте компиляции. В результат будут включены также и любые синтаксические ошибки.

Выпуск сборки

Создание выходной сборки сводится просто к вызову метода `Emit`:

```
EmitResult result = compilation.Emit (@"c:\temp\test.exe");
Console.WriteLine (result.Success);
```

Если значением `result.Success` является `false`, тогда `EmitResult` имеет также свойство `Diagnostics`, предназначенное для указания на ошибки, которые произошли во время выпуска (сюда входит и диагностика из предшествующих этапов). Если метод `Emit` терпит неудачу из-за ошибки файлового ввода-вывода, то вместо передачи кодов ошибок он сгенерирует исключение.

В .NET 5+ и .NET Core расширение `.dll` должно указываться даже для консольных или оконных приложений. Чтобы выполнить приложение, потребуется запустить `dotnet.exe` с путем к сборке `.dll`.

Метод `Emit` также позволяет указывать путь к файлу `.pdb` (для отладочной информации) и путь к файлу XML-документации.

Выдача запросов к семантической модели

Вызов метода `GetSemanticModel` на объекте компиляции возвращает *семантическую модель* для синтаксического дерева:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (123);
}");
```

```
var references =
     ((string)AppContext.GetData("TRUSTED_PLATFORM_ASSEMBLIES"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));
var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);
SemanticModel model = compilation.GetSemanticModel (tree);
```

(Причина, по которой необходимо указывать дерево, связана с тем, что объект компиляции может содержать множество деревьев.)

Можно было бы предположить, что семантическая модель похожа на синтаксическое дерево, но с бо́льшим количеством свойств и методов и более детализированной структурой. Это не так; нет никакой всеобъемлющей DOM-модели, которая бы ассоциировалась с семантической моделью. Взамен вы располагаете набором методов, вызываемых для получения семантической информации о конкретной позиции или узле в синтаксическом дереве.

Вывод из сказанного — вы не можете "исследовать" семантическую модель подобно тому, как обошлись бы с синтаксическим деревом, и работа с ней довольно похожа на игру "20 вопросов": проблема заключается в том, чтобы отыскать правильные вопросы. Существует около 50 обычных и расширяющих методов; в настоящем разделе мы раскроем ряд наиболее распространенных методов, в частности, те, которые демонстрируют принципы использования семантической модели.

Продолжая предыдущий пример, мы могли бы запросить информацию о символах для идентификатора `WriteLine` следующим образом:

```
var writeLineNode = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "WriteLine").Parent;
SymbolInfo symbolInfo = model.GetSymbolInfo (writeLineNode);
Console.WriteLine (symbolInfo.Symbol);
      // System.Console.WriteLine(int)
```

Структура `SymbolInfo` — это оболочка для символов, нюансы которой мы вскоре обсудим. Начнем с символов.

Символы

В синтаксическом дереве имена, подобные"System", "Console" и "WriteLine", разбираются как *идентификаторы* (узел `IdentifierNameSyntax`). Идентификаторы мало что значат, и синтаксический анализатор не предпринимает никаких действий для их "понимания", а только проводит различие между ними и контекстными ключевыми словами.

Семантическая модель также способна трансформировать идентификаторы в *символы*, которые имеют информацию о типе (выходные данные фазы *связывания*).

Все символы реализуют интерфейс `ISymbol`, хотя для каждого вида символов предусмотрен более специфичный интерфейс. В нашем примере "System", "Console" и "WriteLine" отображаются на символы следующих типов:

```
"System"      INamespaceSymbol
"Console"     INamedTypeSymbol
"WriteLine"   IMethodSymbol
```

Некоторые типы символов вроде `IMethodSymbol` имеют концептуальный аналог в пространстве имен `System.Reflection` (`MethodInfo` в данном случае); тогда как ряд других типов символов, таких как `INamespaceSymbol`, аналогов не имеют. Это объясняется тем, что система типов Roslyn существует для оказания помощи компилятору, в то время как система типов `Reflection` предназначена для содействия среде CLR (после того, как исходный код исчезает).

Тем не менее, работа с типами `ISymbol` во многом подобна применению API-интерфейса рефлексии, который был описан в главе 18. Давайте расширим предыдущий пример:

```
ISymbol symbol = model.GetSymbolInfo (writeLineNode).Symbol;

Console.WriteLine (symbol.Name);                  // WriteLine
Console.WriteLine (symbol.Kind);                  // Method
Console.WriteLine (symbol.IsStatic);              // True
Console.WriteLine (symbol.ContainingType.Name);   // Console

var method = (IMethodSymbol) symbol;
Console.WriteLine (method.ReturnType.ToString()); // void
```

Вывод в последней строке кода иллюстрирует тонкое отличие от API-интерфейса рефлексии. Обратите внимание на то, что слово "void" представлено в нижнем регистре; это является спецификацией C# (API-интерфейс рефлексии безразличен к языкам). Подобным же образом вызов метода `ToString` на типе `INamedTypeSymbol` для `System.Int32` возвращает "int". Есть кое-что еще, чего не удастся добиться с помощью API-интерфейса рефлексии:

```
Console.WriteLine (symbol.Language);  // C#
```

На заметку! Благодаря наличию API-интерфейса синтаксических деревьев классы, представляющие синтаксические узлы, отличаются для языков C# и Visual Basic (хотя они совместно используют абстрактный базовый тип `SyntaxNode`). Это имеет смысл, т.к. языки обладают разной лексической структурой. По контрасту `ISymbol` и производные от него интерфейсы разделяются между C# и Visual Basic. Однако их внутренние конкретные реализации *специфичны* для каждого языка, а вывод, получаемый из их методов, отражает отличия, характерные для того или иного языка.

Можно также выяснить, откуда поступил символ:

```
var location = symbol.Locations.First();
Console.WriteLine (location.Kind);                // MetadataFile
Console.WriteLine (location.MetadataModule
               == compilation.References.Single() // True
```

Если символ был определен в принадлежащем нам исходном коде (т.е. в синтаксическом дереве), тогда свойство `SourceTree` возвратит такое дерево, а свойство `SourceSpan` — его местоположение в дереве:

```
Console.WriteLine (location.SourceTree == null);  // True
Console.WriteLine (location.SourceSpan);          // [0..0)
```

Частичный тип может иметь множество определений и соответственно множество местоположений.

Приведенный ниже запрос возвращает все перегруженные версии метода `WriteLine`:

```
symbol.ContainingType.GetMembers ("WriteLine").OfType<IMethodSymbol>()
```

Можно также вызывать метод `ToDisplayParts` на символе. Он возвратит коллекцию "частей", которые образуют полное имя; в данном случае `System.Console.WriteLine(int)` включает четыре символа, отделяемые друг от друга знаками пунктуации.

SymbolInfo

Если вы реализуете средство автозавершения кода для какого-то редактора, то вам понадобится получать символы для кода, который пока еще не завершен или является некорректным. Например, взгляните на следующий незавершенный код:

```
System.Console.Writeline(
```

Поскольку метод `WriteLine` перегружен, сопоставление с единственной реализацией `ISymbol` невозможно. Взамен необходимо предложить пользователю варианты на выбор. Для такой цели в семантической модели имеется метод `GetSymbolInfo`, возвращающий структуру `ISymbolInfo`, которая содержит показанные далее свойства:

```
ISymbol Symbol
ImmutableArray<ISymbol> CandidateSymbols
CandidateReason CandidateReason
```

В случае ошибки или неоднозначности свойство `Symbol` возвращает `null`, а свойство `CandidateSymbols` — коллекцию с наилучшими совпадениями. Свойство `CandidateReason` возвращает перечисление, сообщающее о том, что именно пошло не так.

На заметку! Чтобы получить информацию об ошибках и предупреждениях для раздела кода, можно также вызвать метод `GetDiagnostics` на семантической модели, указав ему структуру `TextSpan`. Вызов `GetDiagnostics` без аргументов эквивалентен вызову того же самого метода на объекте `CSharpCompilation`.

Доступность символов

В интерфейсе `ISymbol` имеется свойство `DeclaredAccessibility`, которое указывает, является ли символ открытым, защищенным, внутренним и т.д. Тем не менее, для определения доступности заданного символа в конкретной точке исходного кода этого недостаточно. Скажем, локальные переменные обладают лексически ограниченной областью видимости, а защищенные члены класса доступны в местах исходного кода, относящихся к внутренностям это-

го класса или его производных классов. Справиться с задачей поможет метод IsAccessible класса SemanticModel:

```
bool canAccess = model.IsAccessible (42, someSymbol);
```

Здесь метод IsAccessible возвращает true, если someSymbol может быть доступен по смещению 42 в исходном коде.

Объявленные символы

Вызов метода GetSymbolInfo на объявлении типа или члена не приводит к получению каких-либо символов. Например, предположим, что требуется символ для метода Main:

```
var mainMethod = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "Main").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (mainMethod);
Console.WriteLine (symbolInfo.Symbol == null);           // True
Console.WriteLine (symbolInfo.CandidateSymbols.Length);  // 0
```

На заметку! Прием применим не только к объявлениям типов/членов, но и к любому узлу, где *вводится* новый символ, а не *потребляется* существующий.

Чтобы получить символ, необходимо взамен вызвать метод GetDeclaredSymbol:

```
ISymbol symbol = model.GetDeclaredSymbol (mainMethod);
```

В отличие от GetSymbolInfo метод GetDeclaredSymbol либо успешно завершается, либо нет. (Если он терпит неудачу, то из-за того, что не может найти допустимый узел объявления.)

В качестве еще одного примера рассмотрим метод Main следующего вида:

```
static void Main()
{
  int xyz = 123;
}
```

Тип xyz можно выяснить так:

```
SyntaxNode variableDecl = tree.GetRoot().DescendantTokens().
Single (
  t => t.Text == "xyz").Parent;

var local = (ILocalSymbol) model.GetDeclaredSymbol (variableDecl);
Console.WriteLine (local.Type.ToString());   // int
Console.WriteLine (local.Type.BaseType.ToString());
                                             // System.ValueType
```

TypeInfo

Иногда нужна информация о типе выражения или литерала, для которого явные символы отсутствуют. Взгляните на следующий код:

```
var now = System.DateTime.Now;
System.Console.WriteLine (now - now);
```

Чтобы определить тип выражения `now - now`, мы вызываем метод `GetTypeInfo` на семантической модели:

```
SyntaxNode binaryExpr = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "-").Parent;
TypeInfo typeInfo = model.GetTypeInfo (binaryExpr);
```

Структура `TypeInfo` имеет два свойства, `Type` и `ConvertedType`. Последнее указывает тип после любых неявных преобразований:

```
Console.WriteLine (typeInfo.Type);           // System.TimeSpan
Console.WriteLine (typeInfo.ConvertedType);  // object
```

Поскольку метод `Console.WriteLine` перегружен для приема типа `object`, но не `TimeSpan`, произошло неявное преобразование в `object`, которое было подтверждено свойством `typeInfo.ConvertedType`.

Поиск символов

Мощной возможностью семантической модели является средство запрашивания всех символов, находящихся в области видимости, для конкретного места исходного кода. Результатом будет основа для формирования списков IntelliSense, когда пользователю нужен перечень доступных символов.

Чтобы получить такой список, необходимо просто вызвать метод `LookupSymbols`, передав ему желаемое смещение в исходном коде. Ниже представлен завершенный пример:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main()
  {
    int x = 123, y = 234;
  }
}");
var references =
     ((string)AppContext.GetData ("TRUSTED_PLATFORM_ASSEMBLIES"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));
var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);
SemanticModel model = compilation.GetSemanticModel (tree);
// Найти доступные символы в начале 6-й строки:
int index = tree.GetText().Lines[5].Start;
foreach (ISymbol symbol in model.LookupSymbols (index))
  Console.WriteLine (symbol.ToString());
```

Вот результат:

```
y
x
Program.Main()
object.ToString()
object.Equals(object)
object.Equals(object, object)
object.ReferenceEquals(object, object)
object.GetHashCode()
object.GetType()
object.~Object()
object.MemberwiseClone()
Program
Microsoft
System
Windows
```

(В случае импортирования пространства имен `System` мы увидим сотни дополнительных символов для типов, определенных в этом пространстве имен.)

Пример: переименование символа

Чтобы продемонстрировать рассмотренные средства, давайте напишем метод переименования символа, который надежно работает в большинстве сценариев использования. В частности:

- символ может быть типом, членом, локальной переменной, переменной диапазона или переменной цикла;
- можно указывать символ либо в месте его применения, либо в точке его объявления;
- в случае класса или структуры будут переименованы статические конструкторы и конструкторы экземпляров;
- в случае класса будет переименован финализатор (деструктор).

Для краткости мы опустим некоторые проверки, такие как выяснение, не используется ли уже новое имя и не относится ли символ к краевому случаю, когда переименование потерпит неудачу. Наш метод будет принимать во внимание только одиночное синтаксическое дерево, поэтому он получит следующую сигнатуру:

```
public SyntaxTree RenameSymbol (SemanticModel model,
                                SyntaxToken token,
                                string newName)
```

Одним очевидным способом реализации является создание подкласса класса `CSharpSyntaxRewriter`. Однако более элегантный и гибкий подход заключается в том, чтобы заставить метод `RenameSymbol` вызывать какой-то низкоуровневый метод, который возвращает текстовые промежутки, подлежащие переименованию:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
```

Это позволяет редактору вызывать метод GetRenameSpans напрямую и применять только изменения (внутри транзакции отмены), избегая утери состояния редактора, что в противном случае может привести к замене всего текста.

В результате метод RenameSymbol становится относительно простой оболочкой вокруг GetRenameSpans. Мы можем использовать метод WithChanges класса SourceText для применения последовательности изменений в тексте:

```
public SyntaxTree RenameSymbol (SemanticModel model,
                                SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans = GetRenameSpans (model, token);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (span => new TextChange (span, newName))
               .OrderBy (tc => tc));

  return model.SyntaxTree.WithChangedText (newSourceText);
}
```

Метод WithChanges генерирует исключение, если изменения не были упорядочены; именно потому мы вызываем метод OrderBy.

Теперь мы должны написать метод GetRenameSpans. Первым делом необходимо найти символ, соответствующий лексеме, которую мы хотим переименовать. Лексема может быть частью либо объявления, либо употребления, так что мы сначала вызываем метод GetSymbolInfo, и если результатом окажется null, тогда вызываем метод GetDeclaredSymbol:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol = model.GetSymbolInfo (node).Symbol
             ?? model.GetDeclaredSymbol (node);
  if (symbol == null) return null; // Символ для переименования
                                   // отсутствует.
```

Далее потребуется найти определения символа. Их можно получить из свойства Locations символа. (Учет множества местоположений обеспечивает надежную работу в сценарии с частичными классами и методами, хотя для того, чтобы данный пример стал пригодным в случае частичных классов, его следовало бы расширить работой с множеством синтаксических деревьев.)

```
var definitions =
  from location in symbol.Locations
  where location.SourceTree == node.SyntaxTree
  select location.SourceSpan;
```

Теперь необходимо найти случаи употребления символа. Мы начинаем с поиска лексем-потомков, имена которых совпадают с именем символа, т.к. это быстрый способ отсеять большинство лексем. Затем мы можем вызвать метод GetSymbolInfo на родительском узле лексемы и увидеть, соответствует ли он символу, который нужно переименовать:

```
var usages =
  from t in model.SyntaxTree.GetRoot().DescendantTokens()
  where t.Text == symbol.Name
  let s = model.GetSymbolInfo (t.Parent).Symbol
  where s == symbol
  select t.Span;
```

На заметку! Операции, относящиеся к связыванию, такие как запрос информации о символе, имеют тенденцию выполняться медленнее операций, которые работают только с текстом или синтаксическими деревьями. Причина в том, что процесс связывания может требовать поиска типов в сборках, применения правил выведения типов и проверки расширяющих методов.

Если символ представляет собой что-то, отличающееся от именованного типа (является локальной переменной, переменной диапазона и т.п.), то работа сделана и можно возвращать определения вместе со случаями употребления:

```
if (symbol.Kind != SymbolKind.NamedType)
  return definitions.Concat (usages);
```

Если символ — именованный тип, тогда понадобится переименовать его конструкторы и деструктор при условии их наличия. Для этого мы организуем перечисление узлов-потомков в поиске объявлений типов с именами, совпадающими с именем типа, который подлежит переименованию. Затем мы получаем его *объявленный* символ, и если он совпадает с тем, который переименовывается, то мы находим его методы конструкторов и деструктора, возвращая промежутки текста его идентификаторов, когда они существуют:

```
  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                    .OfType<TypeDeclarationSyntax>
()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

Ниже показан завершенный код наряду с примерами его использования:

```
void Demo()
{
  var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static Program() {}
  public Program() {}
```

```
  static void Main()
  {
    Program p = new Program();
    p.Foo();
  }
  void Foo() => Bar();
  void Bar() => Foo();
}
");
var references =
 ((string)AppContext.GetData ("TRUSTED_PLATFORM_ASSEMBLIES"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));
var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);
var model = compilation.GetSemanticModel (tree);
var tokens = tree.GetRoot().DescendantTokens();
  // Переименовать класс Program в Program2:
  SyntaxToken program = tokens.First (t => t.Text == "Program");
  Console.WriteLine
      (RenameSymbol (model, program, "Program2").ToString());
  // Переименовать метод Foo в Foo2:
  SyntaxToken foo = tokens.Last (t => t.Text == "Foo");
  Console.WriteLine (RenameSymbol (model, foo, "Foo2").ToString());
  // Переименовать локальную переменную p в p2:
  SyntaxToken p = tokens.Last (t => t.Text == "p");
  Console.WriteLine (RenameSymbol (model, p, "p2").ToString());
}
public SyntaxTree RenameSymbol (SemanticModel model,
                                SyntaxToken token, string newName)
{
  IEnumerable<TextSpan> renameSpans =
    GetRenameSpans (model, token).OrderBy (s => s);
  SourceText newSourceText =
      model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (s => new TextChange (s, newName)));
  return model.SyntaxTree.WithChangedText (newSourceText);
}
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;
  ISymbol symbol =
    model.GetSymbolInfo (node).Symbol ??
    model.GetDeclaredSymbol (node);
```

```
  if (symbol == null) return null;  // Нет символов для переименования
  var definitions =
    from location in symbol.Locations
    where location.SourceTree == node.SyntaxTree
    select location.SourceSpan;
  var usages =
    from t in model.SyntaxTree.GetRoot().DescendantTokens ()
    where t.Text == symbol.Name
    let s = model.GetSymbolInfo (t.Parent).Symbol
    where s == symbol
    select t.Span;
  if (symbol.Kind != SymbolKind.NamedType)
    return definitions.Concat (usages);
  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                  .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;
  return definitions.Concat (usages).Concat (structors);
}
```