ГЛАВА 6

Введение в LINQ to XML

Итак, вы хотите стать "героем XML"? Готовы ли вы к тому, чтобы выдержать шквал камней и стрел? В листинге 6.1 показан код, который в конечном итоге создает тривиальную иерархию XML, используя оригинальный программный интерфейс Microsoft объектной модели документов — XML Document Object Model (DOM) API, основанный на W3C DOM XML API, который демонстрирует, насколько болезненной может быть данная модель.

Листинг 6.1. Простой пример XML

```
using System.Xml;

// Объявляю некоторые переменные, которые будут использоваться повторно.
XmlElement xmlBookParticipant;
XmlAttribute xmlParticipantType;
XmlElement xmlFirstName;
XmlElement xmlLastName;

// Сначала я должен построить документ XML.
XmlDocument xmlDoc = new XmlDocument();

// Создаю корневой элемент и добавляю его в документ.
XmlElement xmlBookParticipants = xmlDoc.CreateElement("BookParticipants");
xmlDoc.AppendChild(xmlBookParticipants);

// Создаю список участников и добавляю в список несколько участников подготовки книги.
xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
xmlParticipantType = xmlDoc.CreateAttribute("type");
xmlParticipantType.InnerText = "Author";
xmlBookParticipant.Attributes.Append(xmlParticipantType);
xmlFirstName = xmlDoc.CreateElement("FirstName");
xmlFirstName.InnerText = "Joe";
xmlBookParticipant.AppendChild(xmlFirstName);
xmlLastName = xmlDoc.CreateElement("LastName");
xmlLastName.InnerText = "Rattz";
xmlBookParticipant.AppendChild(xmlLastName);
xmlBookParticipants.AppendChild(xmlBookParticipant);

// Создаю еще одного участника и добавляю в список участников подготовки книги.
xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
xmlParticipantType = xmlDoc.CreateAttribute("type");
xmlParticipantType.InnerText = "Editor";
xmlBookParticipant.Attributes.Append(xmlParticipantType);
xmlFirstName = xmlDoc.CreateElement("FirstName");
xmlFirstName.InnerText = "Ewan";
```

```
xmlBookParticipant.AppendChild(xmlFirstName);
xmlLastName = xmlDoc.CreateElement("LastName");
xmlLastName.InnerText = "Buckingham";
xmlBookParticipant.AppendChild(xmlLastName);
xmlBookParticipants.AppendChild(xmlBookParticipant);

// Теперь найду авторов и отображу их имена и фамилиии.
XmlNodeList authorsList =
  xmlDoc.SelectNodes("BookParticipants/BookParticipant[@type=\"Author\"]");
foreach (XmlNode node in authorsList)
{
  XmlNode firstName = node.SelectSingleNode("FirstName");
  XmlNode lastName = node.SelectSingleNode("LastName");
  Console.WriteLine("{0} {1}", firstName, lastName);
}
```

Последняя строка кода с вызовом метода `WriteLine` выделена полужирным, потому что я собираюсь ее изменить. Весь этот код строит следующую иерархию XML и пытается отобразить имя каждого участника книги:

Желаемая структура XML

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Писать, разбирать и сопровождать такой код — сущий кошмар. Он чрезвычайно многословен. Просто взглянув на него, невозможно получить никакого представления о том, как должна выглядеть результирующая структура XML. Отчасти то, что делает его таким громоздким — тот факт, что вы не можете создать элемент, инициализировать его и прикрепить в иерархии в одном операторе. Вместо этого каждый элемент сначала необходимо создать, затем вызовом его внутреннего метода `InnerTest` установить нужное значение и, наконец, добавить к некоторому уже существующему узлу в документе XML. Это должно быть сделано с каждым элементом и атрибутом. Все это порождает огромный объем кода. Вдобавок XML-документ должен сначала быть создан, потому что без этого вы не можете даже создать элемент. Очень часто вам не нужен действительный документ XML, а нужен только небольшой его фрагмент вроде приведенного выше.

И, наконец, просто посмотрите, насколько много строк кода понадобилось, чтобы сгенерировать XML столь небольшого объема.

Теперь давайте взглянем на потрясающий вывод. Я просто нажму <Ctrl+F5>:

```
System.Xml.XmlElement System.Xml.XmlElement
```

Ух, ты! Похоже, я не получил действительного текста узлов `FirstName` и `LastName` в этом цикле `foreach`. Я модифицирую этот метод `Console.WriteLine`, чтобы получить эти данные:

```
Console.WriteLine("{0} {1}", firstName.ToString(), lastName.ToString());
```

Теперь готовьтесь! Абракадабра, <Ctrl+F5>:

```
System.Xml.XmlElement System.Xml.XmlElement
```

Какой удар...

Я вспоминаю свое первое знакомство с XML. В 1999 г. я выполнял работу по контракту для подразделения информационных технологий крупной корпорации. Я был включен в группу общей архитектуры, которая искала решение для протоколирования в проекте HP/UX. Мне хотелось использовать что-то готовое для ведения журнальных файлов, что избавило бы меня от необходимости разработки собственного механизма. Конечно, я хотел, чтобы мои журналы хранились в обычных текстовых файлах, и для их обработки можно было применить арсенал команд оболочки, имевшийся в моем распоряжении. Вместо этого мне порекомендовали решение протоколирования, которое хранило журнальные сообщения в виде XML. Мне показали один из протокольных файлов. Ох!

"Как мне порезать это все[1]?" — спросил я. Мне сказали, чтобы я не беспокоился; у них есть приложение с графическим интерфейсом пользователя, специально предназначенное для чтения этих файлов. Все это казалось шуткой. К сожалению, проект закрыли прежде, чем дело дошло до собственно протоколирования.

В 2000 г. в другой компании мне навязали "использование промышленного стандарта XML", несмотря на тот факт, что в индустрии не существовало стандартной схемы, зафиксированной в XML. Но наш менеджмент решил, что мы должны стать первыми, даже без партнера, который будет потребителем этого. Стандарт ли это, если вы одни только его и используете?

Из моего опыта работы с XML мне стало ясно, что XML — технология, которой придется придерживаться. Многие компании что-то бубнили о формате данных, который было бы легко использовать совместно. На бумаге XML выглядел хорошо, хотя его API был далеко не дружественным. Каждый журнал, посвященный высоким технологиям, пел дифирамбы XML. Исполнители читали их, руководители технологических служб верили им, а директора — требовали применения. XML был обречен на успех.

Независимо от нашего сопротивления, XML без сомнения, стал стандартом обмена данными. И как сказал один из моих лучших друзей, пытаясь найти комплимент для XML, — он хорошо сжимает.

Поэтому следующий раз, когда вы захотите привлечь внимание юной леди, прошепчите ей что-нибудь на тему пространств имен, узлов или атрибутов. Она окажется в ваших руках:

```
<PuttyInYourHands>True</PuttyInYourHands>
```

Введение

Microsoft могла бы ограничиться тем, что предоставила бы нам интерфейс LINQ XML API, который позволял бы лишь выполнять запросы LINQ, и все. К счастью для разработчиков XML, они прошли немного дальше. В дополнение к предоставлению поддержки XML запросов LINQ, Microsoft компенсировала многие недостатки стандартного DOM XML API. После нескольких лет мучений с W3C DOM XML API большинству разработчиков стало ясно, что многие задачи оказывается, не так просты, как должны были быть. Имея дело с небольшими фрагментами XML, используя W3C DOM, всегда приходится создавать целый документ XML, даже когда нужно создать всего несколько элементов. Случалось ли вам строить строки, выглядящие как XML вместо применения DOM API, просто потому что это было проще? Уверен, что да.

---

[1] Намек на стандартную команду оболочки `cut` (вырезать), которая позволяет вырезать поля из строки текстового файла на основе либо позиций символа в строке, либо общего разделителя полей, такого как знак табуляции, пробел или запятая.

Несколько ключевых недостатков W3C DOM XML API были преодолены. Была создана новая модель документов. И в результате появился намного более простой и элегантный метод создания деревьев XML. Непомерно раздутый код, вроде приведенного в листинге 6.1, с появлением LINQ стал достоянием прошлого. Создание полного дерева XML с помощью единственного оператора стало реальностью, благодаря функциональному конструированию. Функциональное конструирование — термин, используемый для описания возможности создания полной иерархии XML в единственном операторе. Уже одно это заставляет ценить LINQ to XML на вес золота.

Конечно, это не стало бы частью LINQ, если бы новый XML API не поддерживал запросы LINQ. Именно для этого было добавлено несколько специфичных для XML операций запросов, реализованных в виде расширяющих методов. Комбинация этих новых XML-специфичных операций со стандартными операциями запросов LINQ to Objects, которые мы обсуждали в части 2 этой книги, создает мощное элегантное решение для нахождения любых нужных данных в дереве XML.

LINQ не только поддерживает все это, но комбинируя запрос с функциональным конструированием, вы можете получить трансформации XML. LINQ to XML чрезвычайно гибок.

Обман W3C DOM XML API

Хорошо, вы работаете со своим проектом и знаете, что некоторые конкретные данные должны быть сохранены в виде XML. В моем случае я разрабатывал класс общего назначения для протоколирования, который позволял бы мне отслеживать все, что делает пользователь внутри моего приложения ASP.NET. Я разработал журнальный файл для двух целей. Во-первых, я хотел, чтобы у меня было средство для обнаружения злоупотреблений системой, если таковые случатся. Во-вторых, что более важно — чтобы мое Web-приложение извещало меня по электронной почте в случае возникновения исключений. Меня всегда расстраивало то, что пользователи, у которых возникали исключения, никогда не помнили, что они делали непосредственно перед этим. Они никогда не могли вспомнить никаких подробностей того, что привело их к ошибке.

Поэтому мне нужно было что-то такое, что отслеживало все их движения, по крайней мере, на стороне сервера. Любое действие пользователя, такое как запуск запроса или отправка заказа, должно рассматриваться как событие. В моей базе данных предусмотрены поля, которые фиксируют пользователя, дату, время и тип события, а также все прочие поля общего назначения, которые могут понадобиться. Однако недостаточно просто знать, что они запросили какой-то счет; я должен знать, каковы были параметры поиска. Если отправлен заказ, нужно знать каким был идентификатор партии и сколько единиц товара заказано. По сути, мне нужны все данные, которые позволят выполнить точно ту же операцию, чтобы воспроизвести условия, повлекшие за собой исключения. Каждый тип события имеет свои данные параметров. Я точно не хотел заводить отдельную таблицу для каждого типа событий, и уверен, что не хочу, чтобы код просмотра событий вынужден был обращаться к миллиону различных таблиц, чтобы воспроизвести действия пользователя. Мне нужна одна таблица, способная вместить всю необходимую информацию, чтобы при ее просмотре я мог увидеть каждое действие (событие), выполненное пользователем. Таким образом, я столкнулся с мнением, что все, что мне нужно — это строка данных XML, хранящая в базе информацию о параметрах событий.

Могло не быть схемы, определяющей то, как выглядит этот XML, потому что это зависит от нужд конкретного события. Если событием является запрос счета за некоторый диапазон дат, он может выглядеть так:

```
<StartDate>10/2/2006</StartDate>
<EndDate>10/9/2006</EndDate>
<IncludePaid>False</IncludePaid>
```

Если же это подтверждение заказа, оно может выглядеть иначе:

```
<PartId>4754611903</PartId>
<Quantity>12</Quantity>
<DistributionCenter>Atlanta<DistributionCenter>
<ShippingCode>USPS First Class<ShippingCode>
```

Я зафиксировал поля, которые мне понадобятся для того, чтобы воспроизвести событие. Поскольку данные варьируются в зависимости от типа события, это исключает проверку достоверности XML, так что уже есть преимущество перед использованием XML DOM API.

Этот инструмент отслеживания событий стал первоклассным средством поддержки, значительно упростившим идентификацию и нахождение ошибок. В качестве побочного эффекта было довольно забавно звонить пользователю на следующий день и говорить, что ошибка, которую они видели, пытаясь получить счет номер 3847329 в предыдущий день, уже исправлена. Паранойя, которую вселяло в пользователя знание того, что я всегда точно знаю, что он делал, часто сама по себе оправдывала применение следящего кода.

Те из вас, кто уже знаком с XML, могут посмотреть на эти схемы и сказать: "Эй, а ведь они неправильно сформированы! Здесь нет корневого узла". Да, это правда, и представляет проблему, если вы используете W3C DOM API. Однако я не использую W3C DOM API для производства XML; я использую другой XML API. И вы, вероятно, должны также использовать его. Он называется `String.Format` XML API, и его применение выглядит примерно так:

```
string xmlData =
  string.Format(
    "<StartDate>{0}</StartDate><EndDate>{1}</EndDate><IncPaid>{2}</IncPaid>",
    Date.ToShortDateString(),
    endDate.ToShortDateString(),
    includePaid.ToString());
```

Да, я знаю, что это плохой способ создания XML-данных. Да, он подвержен ошибкам. Конечно, легко представить допустить опечатку или изменить регистр (`EndDate` вместо `endDate`, например), таким образом некорректно закрыв дескриптор. Я зашел так далеко, что создал метод, принимающий список параметров — имен элементов и их данных. Потому мой код выглядит скорее так:

```
string xmlData =
  XMLHelper(
    "StartDate", startDate.ToShortDateString(),
    "EndDate", endDate.ToShortDateString(),
    "IncPaid", includePaid.ToString());
```

Этот метод `XMLHelper` также создаст для меня корневой узел. Но, опять-таки, это не намного лучше. Вы видите, что я никак не кодирую свои данные в этом вызове. Потребовалось некоторое время, прежде чем я осознал, что передаваемые данные лучше все-таки кодировать.

Хотя применение метод `String.Format` или любой другой техники, отличной от XML DOM API, — плохая замена DOM, все же существующий API несет с собой слишком много хлопот, когда приходится иметь дело лишь с небольшим фрагментом XML, как было у меня в описанном случае.

Если вы думаете, что я одинок в своем подходе к созданию XML, должен сказать, что мне случилось недавно побывать на семинаре Microsoft, где докладчик представлял код построения строки XML с использованием обычной конкатенации. Если бы был лучший способ! Если бы был доступен LINQ!

Резюме

Когда кто-либо произносит термин LINQ, то первое впечатление, которое возникает у большинства разработчиков — это нечто, предназначенное для выполнения запросов данных. Более того, им свойственно при этом исключать из области своего внимания другие источники информации, помимо баз данных. LINQ to XML напоминает вам, что LINQ предназначен также и для обработки XML, а не только для запросов к XML.

В этой главе я продемонстрировал неудобства обращения с XML, свойственные W3C DOM XML API, а также некоторые традиционные способы избежать этих неудобств. В следующей главе я опишу программный интерфейс LINQ to XML API. Используя этот API, я продемонстрирую, как можно создавать иерархии XML лишь небольшой частью того объема кода, который понадобился бы для этого в W3C DOM XML API. Просто чтобы заинтриговать вас, я скажу вам сейчас, что в следующей главе создам ту же иерархию XML, что и в листинге 6.1, используя для этого LINQ to XML, и вместо 29 строк кода, которые понадобились в листинге 6.1, уложусь всего в 10.

Надеюсь, закончив читать следующие две главы, вы согласитесь с тем, что LINQ произвел революцию в манипуляциях XML, как и в запросах базы данных.