

THE EXPERT'S VOICE® IN NET

LINQ

ЯЗЫК ИНТЕГРИРОВАННЫХ
ЗАПРОСОВ В С# 2008

ДЛЯ ПРОФЕССИОНАЛОВ

*Изучите новейшую технологию
от Microsoft*



www.williamspublishing.com

Apress®

www.apress.com

Джозеф Раттц-мл.

PRO LINQ

LANGUAGE INTEGRATED QUERY IN C# 2008

Joseph C. Rattz, Jr.

Apress®

LINQ

язык интегрированных запросов в C# 2008 для профессионалов

Джозеф Раттц-мл.



Москва • Санкт-Петербург • Киев
2008

ББК 32.973.26-018.2.75

P25

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией С.Н. Тригуб

Перевод с английского Н.А. Мухина

Под редакцией Ю.Н. Артеменко

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

Ратц-мл., Джозеф С.

P25 LINQ: язык интегрированных запросов в C# 2008 для профессионалов. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 560 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1427-9 (рус.)

Книга одного из экспертов в области технологий .NET представляет собой учебное и справочное пособие для разработчиков .NET-приложений, использующих новую версию ASP.NET 3.5 и предложенную Microsoft технологию работы с данными под названием LINQ, которая является встроенной в язык C# 3.0. Предложенный автором практический подход к изложению материала позволяет оперативно изучить новейшие способы доступа к данным из разнообразных источников, в том числе SQL Server, и методы работы с XML на реальных примерах. Подробно рассматриваются все вопросы, связанные с LINQ, начиная с объектной модели, операций и API-интерфейсов LINQ to Objects, LINQ to XML, LINQ to DataSet, LINQ to SQL и LINQ to Entities, и заканчивая разрешением конфликтов параллельного доступа и работе с представлениями баз данных.

Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием и разработкой для .NET.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства APress, Berkeley, CA.

Authorized translation from the English language edition published by APress, Copyright © 2007 by Joseph C. Rattz, Jr.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008.

ISBN 978-5-8459-1427-9 (рус.)

ISBN 978-1-59-059789-7 (англ.)

© Издательский дом "Вильямс", 2008

© by Joseph C. Rattz, Jr., 2007

Оглавление

Часть I. LINQ: язык интегрированных запросов в C# 2008	17
Глава 1. Знакомство с LINQ	18
Глава 2. Расширения языка C# 3.0 для LINQ	32
Часть II. LINQ to Objects	61
Глава 3. Введение в LINQ to Objects	62
Глава 4. Отложенные операции	70
Глава 5. Не отложенные операции	133
Часть III. LINQ to XML	179
Глава 6. Введение в LINQ to XML	180
Глава 7. Интерфейс LINQ to XML API	186
Глава 8. Операции LINQ to XML	258
Глава 9. Дополнительные возможности XML	282
Часть IV. LINQ to DataSet	315
Глава 10. Операции LINQ to DataSet	316
Глава 11. Дополнительные возможности DataSet	346
Часть V. LINQ to SQL	353
Глава 12. Введение в LINQ to SQL	354
Глава 13. Советы и инструменты, связанные с LINQ to SQL	365
Глава 14. Операции для баз данных в LINQ to SQL	387
Глава 15. Сущностные классы LINQ to SQL	426
Глава 16. <code>DataContext</code>	468
Глава 17. Конфликты параллельного доступа	519
Глава 18. Дополнительные возможности SQL	536
Предметный указатель	547

Содержание

Об авторе	13
О техническом рецензенте	13
Благодарности	14
От издательства	16
Исходный код примеров	16
Часть I. LINQ: язык интегрированных запросов в C# 2008	17
Глава 1. Знакомство с LINQ	18
Смещение парадигмы	18
Запрос к XML	19
Запрос к базе данных SQL Server	20
Введение	21
LINQ для запросов данных	21
Компоненты	22
Как получить LINQ	23
LINQ — не только для запросов	23
Советы начинающим	26
Когда запутались, используйте ключевое слово var	26
Использование операций Cast или OfType для унаследованных коллекций	27
Отдавайте предпочтение операции OfType перед Cast	28
Не рассчитывайте на безошибочность запросов	28
Используйте преимущество отложенных запросов	30
Использование Log из DataContext	30
Используйте форум LINQ	31
Резюме	31
Глава 2. Расширения языка C# 3.0 для LINQ	32
Новые дополнения языка C# 3.0	32
Лямбда-выражения	32
Деревья выражений	37
Ключевое слово var, инициализация объектов и анонимные типы	38
Расширяющие методы	43
Частичные методы	47
Выражения запросов	49
Резюме	59
Часть II. LINQ to Objects	61
Глава 3. Введение в LINQ to Objects	62
Обзор LINQ to Objects	62
IEnumerable<T>, последовательности и стандартные операции запросов	63
Возврат IEnumerable<T>, Yielding и отложенные запросы	64
Делегаты Func	67
Алфавитный указатель стандартных операций запросов	68
Резюме	69

Глава 4. Отложенные операции	70
Необходимые пространства имен	70
Необходимые сборки	70
Общие классы	70
Разделение отложенных операций по их назначению	72
Ограничение	72
Проекция	74
Разбиение	82
Конкатенация	88
Упорядочивание	91
Соединение	106
Группировка	109
Множества	115
Преобразование	120
Элемент	126
Генерация	130
Резюме	132
Глава 5. Не отложенные операции	133
Необходимые пространства имен	133
Общие классы	133
Не отложенные операции по их назначению	136
Преобразование	136
Эквивалентность	147
Элемент	149
Квантификаторы	160
Агрегация	165
Резюме	177
Часть III. LINQ to XML	179
Глава 6. Введение в LINQ to XML	180
Введение	182
Обман W3C DOM XML API	183
Резюме	185
Глава 7. Интерфейс LINQ to XML API	186
Необходимые пространства имен	186
Существенные усовершенствования дизайна API	186
Конструирование деревьев XML было упрощено функциональным конструированием	187
Центральная роль элемента вместо документа	189
Имена, пространства имен и префиксы	190
Извлечение значения узла	193
Объектная модель LINQ to XML	195
Отложенное выполнение запросов, удаление узлов и “проблема Хэллоуина”	196
Создание XML	198
Создание элементов с помощью XElement	199
Создание атрибутов с помощью XAttribute	201
Создание комментариев с помощью XComment	202
Создание контейнеров с помощью XContainer	202

8 Содержание

Создание объявлений с помощью XDeclaration	203
Создание типов документов с помощью XDocumentType	203
Создание документов с помощью XDocument	205
Создание имен с помощью XName	205
Создание пространств имен с помощью XNamespace	206
Создание узлов с помощью XNode	206
Создание инструкций обработки с помощью XProcessingInstruction	206
Создание потоковых элементов с помощью XStreamingElement	208
Создание текста с помощью XText	209
Создание CData с помощью XCData	210
Вывод XML	210
Сохранение с помощью XDocument.Save()	210
Сохранение с помощью XElement.Save()	211
Ввод XML	212
Загрузка с помощью XDocument.Load()	212
Загрузка с помощью XElement.Load()	214
Разбор содержимого методами XDocument.Parse() или XElement.Parse()	214
Проход по XML	215
Свойства прохода	216
Вперед с помощью XNode.NextNode	216
Методы прохода	219
Модификация XML	231
Добавление узлов	231
Удаление узлов	235
Обновление узлов	238
XElement.SetValue() на дочерних объектах XElement	241
Атрибуты XML	242
Создание атрибута	242
Проход по атрибутам	242
Модификация атрибута	245
Аннотации XML	249
Добавление аннотаций с помощью XObject.AddAnnotation()	249
Обращение к аннотациям с помощью XObject.Annotation() или XObject.Annotations()	249
Удаление аннотаций с помощью XObject.RemoveAnnotations()	250
Пример аннотаций	250
События XML	253
XObject.Changing	253
XObject.Changed	253
Несколько примеров событий	254
Трюк, забава или неопределенность?	257
Резюме	257
Глава 8. Операции LINQ to XML	258
Введение в операции LINQ to XML	258
Ancestors	259
Прототипы	259
Примеры	259
AncestorsAndSelf	263
Прототипы	263
Примеры	263

Attributes	265
Прототипы	265
Примеры	265
DescendantNodes	267
Прототипы	267
Примеры	267
DescendantNodesAndSelf	268
Прототипы	268
Примеры	269
Descendants	270
Прототипы	270
Примеры	270
DescendantsAndSelf	272
Прототипы	272
Примеры	272
Elements	274
Прототипы	274
Примеры	275
InDocumentOrder	276
Прототипы	276
Примеры	276
Nodes	277
Прототипы	278
Примеры	278
Remove	279
Прототипы	279
Примеры	279
Резюме	281
Глава 9. Дополнительные возможности XML	282
Необходимые пространства имен	282
Запросы	283
Без спуска	283
Сложный запрос	285
Трансформации	290
Трансформации с использованием XSLT	291
Трансформация с использованием функционального конструирования	293
Советы	295
Проверка достоверности	300
Расширяющие методы	300
Прототипы	300
Получение схемы XML	301
Примеры	303
XPath	313
Прототипы	313
Примеры	313
Резюме	314

10 Содержание

Часть IV. LINQ to DataSet	315
Глава 10. Операции LINQ to DataSet	316
Необходимые сборки	317
Необходимые пространства имен	317
Общий код для примеров	317
Операции множеств DataRow	318
Distinct	319
Except	322
Intersect	324
Union	326
SequenceEqual	327
Операции над полями DataRow	329
Field<T>	332
SetField<T>	337
Операции DataTable	339
AsEnumerable	339
CopyToDataTable<DataRow>	340
Примеры	341
Резюме	345
Глава 11. Дополнительные возможности DataSet	346
Необходимые пространства имен	346
Типизированные DataSet	346
Собираем все вместе	348
Резюме	350
Часть V. LINQ to SQL	353
Глава 12. Введение в LINQ to SQL	354
Введение в LINQ to SQL	355
DataContext	357
Сущностные классы	357
Ассоциации	358
Обнаружение конфликтов параллельного доступа	359
Разрешение конфликтов параллельного доступа	359
Предварительные условия для запуска примеров	359
Получение соответствующей версии базы данных Northwind	359
Генерация сущностных классов Northwind	360
Генерация XML-файла отображения Northwind	361
Использование LINQ to SQL API	361
IQueryable<T>	361
Некоторые общие методы	362
GetStringFromDb()	362
ExecuteStatementInDb()	363
Резюме	364
Глава 13. Советы и инструменты, связанные с LINQ to SQL	365
Введение	365
Советы	366
Используйте свойство DataContext.Log	366

Используйте метод GetChangeSet()	367
Попробуйте использовать частичные классы или файлы отображения	367
Попробуйте использовать частичные методы	367
Инструменты	368
SQLMetal	368
Object Relational Designer	373
Совместное использование SQLMetal и O/R Designer	386
Резюме	386
Глава 14. Операции для баз данных в LINQ to SQL	387
Предварительные условия для запуска примеров	387
Некоторые общие методы	387
Использование программного интерфейса LINQ to SQL API	388
Стандартные операции для баз данных	388
Вставки	388
Запросы	392
Обновления	413
Удаления	416
Переопределение операторов модификации базы данных	419
Переопределение метода Insert	419
Переопределение метода Update	420
Переопределение метода Delete	420
Пример	420
Переопределение в Object Relational Designer	422
Сообщения	423
Трансляция SQL	423
Резюме	425
Глава 15. Сущностные классы LINQ to SQL	426
Предварительные условия для запуска примеров	426
Сущностные классы	426
Создание сущностных классов	426
XML-схема внешнего файла отображения	454
Сравнение проекций на сущностные и на несущностные классы	454
Расширение сущностных методов частичными методами	459
Импорт классов System.Data.Linq	461
EntitySet<T>	461
EntityRef<T>	462
Table<T>	464
IExecuteResult	464
ISingleResult<T>	465
IMultipleResults	466
Резюме	467
Глава 16. DataContext	468
Предварительные условия для запуска примеров	468
Некоторые общие методы	468
Использование LINQ to SQL API	468
Класс [Your]DataContext	468
Класс DataContext	469
Главные цели	471

12 Содержание

DataContext() и [Your]DataContext()	477
SubmitChanges()	488
DatabaseExists()	494
CreateDatabase()	494
DeleteDatabase()	496
CreateMethodCallQuery()	496
ExecuteQuery()	498
Translate()	500
ExecuteCommand()	501
ExecuteMethodCall()	503
GetCommand()	509
GetChangeSet()	510
GetTable()	512
Refresh()	513
Резюме	518
Глава 17. Конфликты параллельного доступа	519
Предварительные условия для запуска примеров	519
Некоторые общие методы	519
Использование LINQ to SQL API	519
Конфликты параллелизма	519
Оптимистический параллелизм	520
Пессимистический параллелизм	530
Альтернативный подход для средних звеньев и серверов	533
Резюме	535
Глава 18. Дополнительные возможности SQL	536
Предварительные условия для запуска примеров	536
Использование LINQ to SQL API	536
Использование LINQ to XML API	536
Представления базы данных	536
Наследование сущностных классов	538
Транзакции	543
Резюме	545
Предметный указатель	547

Об авторе

Джозеф Ратц-мл. (Joseph C. Rattz, Jr.) неосознанно начал свою карьеру в разработке программного обеспечения в 1990 г., когда друг попросил его помочь в написании текстового редактора ANSI под названием *ANSI Master* для компьютера Commodore Amiga. Вскоре за этим последовала игра в палача (*The Gallows* — “Виселица”). От этих программ, написанных на компилируемом Basic, он перешел к программированию на С, в поисках более высокой скорости и мощи. После этого Джо разрабатывал приложения для *JumpDisk* — журнал Amiga на дисках, а также для журнала *Amiga World*. Из-за того, что ему пришлось работать в маленьком городе в относительной изоляции, Джо изучил все неправильные способы написания кода. Это обучение происходило при попытках усовершенствовать его плохо написанные приложения, в процессе которых он осознал важность написания хорошо сопровождаемого кода. Впервые познакомившись с отладчиком уровня исходного кода, он влюбился в него с первого взгляда.

Двумя годами позже Джо получил свою первую работу в качестве разработчика программного обеспечения в Policy Management Systems Corporation, как программист начального уровня, разрабатывающий клиент-серверное приложение системы страхования для OS/2 и Presentation Manager. С годами он добавил к своему багажу знаний C++, Unix, Java, ASP, ASP.NET, C#, HTML, DHTML и XML, разрабатывая приложения для SCT, DocuCorp, IBM, Комитета по проведению олимпийских игр в Атланте, CheckFree, NCR, EDS, Delta Technology, Radiant Systems и Genuine Parts Company. Джо нравились творческие аспекты дизайна пользовательского интерфейса, и он осознал необходимость дисциплины при разработке программного обеспечения серверной стороны. Но когда у него была такая возможность, его любимым времятпрепровождением была отладка кода.

Сегодня Джо можно найти в Genuine Parts Company — родительской компании NAPA — в департаменте Automotive Parts Group Information Systems, где он трудится над своим детищем — Web-сайтом Storefront. Этот сайт обслуживает хранилища NAPA, предоставляя их счета и данные в сети систем AS/400.

Вы можете обратиться к Джо через его Web-сайт www.linqdev.com.

О техническом рецензенте

Фабио Клаудио Феррачати (Fabio Claudio Ferracchiat) — старший консультант и аналитик-разработчик, имеющий дело с технологиями Microsoft. Он работает на компанию Brain Force (www.brainforce.com) в ее итальянском подразделении (www.brainforce.it). Является сертифицированным разработчиком решений Microsoft для .NET, сертифицированным разработчиком приложений Microsoft для .NET, сертифицированным профессионалом Microsoft, а также плодовитым автором и техническим рецензентом. За последние десять лет он написал множество статей для итальянских и международных изданий, а также является соавтором более десятка книг на различные компьютерные темы. Вы можете ознакомиться с его блогом, посвященным LINQ на сайте www.ferracchiat.com.

Благодарности

Я уже несколько лет хотел написать книгу, но все не мог найти тему, о которой бы чувствовал себя вправе писать. Либо я знал материал хорошо, потому что он был старым и хорошо всем известным, не требующим новой книги, либо я не знал о нем ничего, и не чувствовал себя вправе о нем писать.

Когда я впервые столкнулся с LINQ в сентябре 2005 г., когда мне на глаза попался видеоролик на тему LINQ от Андерса Хейлсберга (Anders Hejlsberg), я сразу оценил революционный эффект, который окажет эта технология на разработку .NET. Но я не мог представить, что всего семью месяцами спустя начну писать книгу о LINQ. У меня не было намерения писать именно такую книгу, но написать книгу я хотел. Однажды в своем почтовом ящике я обнаружил электронное письмо на тему программирования, которое было озаглавлено May 2006 LINQ Community Technology Preview (Предварительный обзор технологии сообщества LINQ от мая 2006 г.). Я понял, что пока никто не может знать о LINQ больше меня в силу его новизны. Кроме разработчиков из Microsoft никто не обладал более чем несколькомесячным опытом в LINQ. Поэтому мне показалось, что наступил подходящий момент для написания книги, поэтому несмотря на то, что я не знал ничего о LINQ, кроме того, что увидел в этом видеоролике, я решился заключить контракт с Apress на написание книги о LINQ. О чем я думал?

Эта книга, которая начиналась, как девятимесячный проект, выросла до семнадцатимесячного проекта. Вы не можете посвятить 17 месяцев вашей жизни чему-то одному, не имея хорошей поддержки со стороны окружающих. Что касается меня, то я бы не смог это сделать без своей жены Вики (Vickey). Целых 17 месяцев она заботилась обо всей жизненной рутине и тянула на себе дом. Целых 17 месяцев она слышала от меня "Не могу, мне надо работать над книгой". Это долго. Я бы не справился с задачей без нее, да и не пытался бы.

Хочу также поблагодарить всех замечательных людей из Apress, которым пришлось со мной повозиться. Никто не сделал большего вклада, чем Трэйси Браун Коллинз (Tracy Brown Collins). Она помогала мне быть сосредоточенным и не отступать от графика. На протяжении четырех выпусков LINQ/Orcas она управляла всем и терпела мои панические реакции.

Следующим в списке Apress идет мой технический рецензент Фабио Клаудио Феррачати (Fabio Claudio Ferracchiat). Если это имя знакомо вам, это может быть связано с тем, что Фабио сам написал две книги о LINQ. Первоначально я не имел понятия о том, что мой технический рецензент был автором другой книги на эту тему. Если бы я знал о том, что за моей работой наблюдает другой автор книги о LINQ — это было бы плохой идеей. Но я уверен, что знания Фабио принесли пользу и моей книге.

Это — моя первая книга, и я определенно нуждался в наставнике, и, вероятно, не мог найти никого лучше на эту роль, чем Эвен Бэкингем (Ewan Buckingham). Опыт Эвена помогал облегчать стрессовые ситуации, с которыми я не знал, как справиться. Эвен также посоветовал использовать некоторые нестандартные приемы, такие как начать книгу прямо с кода. Мне нравится, что книга начинается с кода, и я надеюсь, что вам тоже.

Также не могу обойти вниманием моих редакторов Хизер Ланг (Heather Lang) и Дженифер Виппл (Jennifer Whipple). Хотя иногда это и раздражало, но все-таки когда есть некто, кто полирует и доводит до совершенства написанное тобой — это замечательно. Один урок, который я вынес из общения с ними, заключается в том, что я чрезвычайно злоупотребляю запятыми.

Хочу поблагодарить Кэти Стенси (Katie Stence) и ее команду. Глава 2 оказалась довольно трудной и потребовала значительных усилий от ее команды. Благодаря ей, трансляции кода представлены настолько хорошо.

Также хочу поблагодарить издательство Apress за предоставленную мне возможность написать книгу. Такая возможность выпадает не каждый день. Это было трудно, но результат того стоил.

Конечно, эта книга была бы невозможной, если бы не Microsoft, и не все ее талантливые и самоотверженные разработчики, которые тратили свое время, отвечая на запросы в форуме MSDN LINQ. Для начала, я хочу поблагодарить Мэтта Уоррена (Matt Warren), который отвечает на львиную долю всех запросов. Он также обратил внимание, что Visual Studio 2008 (вместе с LINQ) был запущен в мой день рождения, так разве я могу не поблагодарить его за это?

Также хочу сказать спасибо Кейт Фармер (Keith Farmer), Динеш Кулкарни (Dinesh Kulkarni), Мэдсу Торгерсену (Mads Torgersen) и Эрику Уайту (Eric White), которые предоставляли мне вопросы, информацию и периодическую поддержку. Уверен, что есть множество незнакомых мне сотрудников Microsoft, которые вложили свой труд в создание LINQ. Хочу сказать им спасибо. И, наконец, хочу поблагодарить Андерса Хейлслерга за его представление о том, откуда растет LINQ.

Эта книга также была бы невозможна без вдохновения и поддержки, которыми я обязан моему другу и автору Брюсу Буковиксу (Bruce Bukovics). Запомните это имя. Работая с Брюсом, всегда можно рассчитывать на то, что он поделится своими знаниями и мудростью.

Вряд ли эта книга могла появиться, если бы не мои родители, потому что мой отец убеждал меня в том, что я могу достичь любой цели, поставленной перед собой, и потому, что моя мама постоянно уверяла меня в моих способностях во время учебы в школе, даже когда ей приходилось заниматься со мной, когда я отставал. Могу сказать, что нет ничего важнее в образовании детей, чем участие и забота родителей.

Упомянув о роли родителей в моем образовании, было бы несправедливо не вспомнить некоторых исключительных учителей, которые изменили мою жизнь. Первая из них — Сюзанн Хадли (Susan Hadley) — моя учительница в шестом классе. Миссис Хадли доказала мне, что учеба может быть интересной, и это является прямым следствием того, что она была моей учительницей. Если вам нравятся шутки, которые встречаются в этой книге — это благодаря ей. Следующим в моем списке идет Руби Джонсон (Ruby Johnson). В то время как прочие учителя заботились лишь о том, чтобы я не отставал от других, она прилагала усилия к моему индивидуальному обучению, что выражалось в индивидуальных уроках, домашних заданиях и тестах. Это, наверно, удваивало ее рабочую нагрузку в классе. Далее идет Нэнси Купер (Nancy Cooper). Миссис Купер также доказала, что учеба не должна быть скучной. Она может быть интересной, захватывающей и не ограниченной рамками учебника.

Эта книга также была бы невозможной без участия многих замечательных моих коллег, с которыми мне довелось работать, и у которых я учился. Наиболее важными были Майк Фурнелл (Mike Furnell), Джеймс Ричардсон (James Richardson), Джон Проктор (John Proctor) и Брэд Радакер (Brad Radaker). Майк, мой первый технический руководитель, как-то спросил: какое у меня самое слабое место в языке C. Когда я сказал, что указатели, он посоветовал мне выполнить отладку нашей библиотеки связных списков на бумаге. Если вы прочтете 2000 строк библиотечного кода управления связными списками, то вы либо свихнетесь, либо изучите указатели как следует, причем одно другого не исключает. Джеймс Ричардсон, обладая всего шестимесячным опытом, продемонстрировал мне один из величайших трюков в программировании, которые я когда-либо видел. Казалось, он просто впитывает подобно губке программирование, языки и кон-

цепции. Он помог мне понять все, что я знаю об абстракции и повторном использовании кода. Джон Проктор обладает, наверное, самым острым программистским умом из всех, что я встречал. Сколько раз он звонил мне по утрам, чтобы спросить о технологии, о которой я, как правило, не имел понятия. Когда я перезванивал ему позже в тот же день, он уже к тому времени загружал из Internet какую-нибудь реализацию с открытым кодом, собирая ее, тестируя, внося изменения в код, разбирая по косточкам и писал собственную реализацию. Брэд Радакер служил постоянным источником информации. Часто он вспоминал годы спустя такие технические детали, которых я совершенно не ожидал. В любой день он мог объяснить мне, как работает мой код, лучше, чем я сам. Все эти ребята служили для меня постоянным источником знаний и вдохновения.

И, наконец, я хочу поблагодарить Эр-Джи Майкла (RJ Mical), который однажды вселил в меня уверенность в своих способностях. Благодаря ему, у меня появилась уверенность, что я правлюсь.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

Исходный код примеров

Исходный код примеров, рассмотренных в книге, доступен для загрузки на сайте издательства по адресу <http://www.williamspublishing.com>.

ЧАСТЬ I

LINQ: язык интегрированных запросов в C# 2008

В этой части...

Глава 1. Знакомство с LINQ

Глава 2. Расширения языка C# 3.0 для LINQ

ГЛАВА 1

Знакомство с LINQ

Листинг 1.1. “Hello LINQ”

```
using System;
using System.Linq;
string[] greetings = {"hello world", "hello LINQ", "hello Apress"};
var items =
    from s in greetings
    where s.EndsWith("LINQ")
    select s;
foreach (var item in items)
    Console.WriteLine(item);
```

На заметку! Код листинга 1.1 был добавлен в проект, созданный через шаблон консольного приложения в Visual Studio 2008. Если у вас этого еще нет, вы должны добавить директиву `using` для пространства имен `System.Linq`.

Запуск приведенного выше кода по нажатию `<Ctrl+F5>` выдаст следующий вывод в окно консоли:

```
hello LINQ
```

Смещение парадигмы

Вы почувствовали только что, как ваш мир сдвинулся с места? Как разработчик .NET, вы должны были это почувствовать. То, что показано в тривиальном примере программы из листинга 1.1, который вы только что запустили, выглядит как запрос на языке структурированных запросов (Structured Query Language — SQL) к массиву строк¹. Взгляните на конструкцию `where`. Она выглядит так, будто я использовал метод `EndsWith` объекта `string`, потому что так оно и есть. Вы можете спросить — а как насчет типа переменной `var`? Выполняет ли по-прежнему C# контроль типов? Ответ — да, он проверяет статически типы во время компиляции. Какое средство или средства C# позволяют все это? Ответ: Microsoft Language Integrated Query (язык интегрированных запросов Microsoft), иначе именуемый как *LINQ*.

¹ Стоит отметить, что порядок противоположен типичному SQL. К тому же добавлена часть “`s in`” запроса, которая представляет ссылку на набор элементов, содержащихся в источнике, которым в данном случае является массив строк “hello world”, “hello LINQ” и “hello Apress”.

Запрос к XML

В то время как пример из листинга 1.1 достаточно тривиален, пример листинга 1.2 уже может очертировать потенциальную мощь, которую вручает LINQ в руки разработчика .NET. Он демонстрирует легкость, с которой можно взаимодействовать и опрашивать данные в расширяемом языке разметки (Extensible Markup Language — XML) с помощью программного интерфейса LINQ to XML. Вы должны уделить особое внимание тому, как я конструирую из данных XML объект по имени books, с которым потом можно взаимодействовать программно.

Листинг 1.2. Простой запрос XML с использованием LINQ to XML

```
using System;
using System.Linq;
using System.Xml.Linq;
 XElement books = XElement.Parse(
 @"<books>
  <book>
    <title>Pro LINQ: Language Integrated Query in C# 2008</title>
    <author>Joe Rattz</author>
  </book>
  <book>
    <title>Pro WF: Windows Workflow in .NET 3.0</title>
    <author>Bruce Bukovics</author>
  </book>
  <book>
    <title>Pro C# 2005 and the .NET 2.0 Platform, Third Edition</title>
    <author>Andrew Troelsen</author>
  </book>
 </books>");

var titles =
 from book in books.Elements("book")
 where (string) book.Element("author") == "Joe Rattz"
 select book.Element("title");
foreach(var title in titles)
 Console.WriteLine(title.Value);
```

На заметку! Код в листинге 1.2 требует добавления к ссылкам проекта сборки System.Xml.Linq.dll, если это еще не сделано. Также обратите внимание, что я добавил директиву using для пространства имен System.Xml.Linq.

Запуск предыдущего кода нажатием <Ctrl+F5> выведет следующие данные в окно консоли:

```
Pro LINQ: Language Integrated Query in C# 2008
```

Вы обратили внимание, что я разобрал данные XML для помещения их в объект XElement? Я нигде не создавал XmlDocument. Среди преимуществ LINQ to XML — расширения, которые он привносит в XML API. Теперь вместо того, чтобы сосредотачивать все вокруг XmlDocument, как того требует W3C Document Object Model (DOM) XML API, LINQ to XML позволяет разработчику взаимодействовать на уровне элемента, используя класс XElement.

На заметку! В дополнение к средствам запросов, LINQ to XML предоставляет более мощный и простой способ использования интерфейса для работы с данными XML.

Опять-таки, обратите внимание, что я использовал некоторый SQL-подобный синтаксис для опроса данных XML, как если бы это была база данных.

Запрос к базе данных SQL Server

Следующий пример демонстрирует применение LINQ to SQL для опроса таблиц базы данных². В листинге 1.3 выполняется запрос к стандартной базе данных примеров Microsoft Northwind.

Листинг 1.3. Простой запрос XML с использованием LINQ to SQL

```
using System.Linq;
using System.Data.Linq;
using nwind;
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
    from c in db.Customers
    where c.City == "Rio de Janeiro"
    select c;
foreach (var cust in custs)
    Console.WriteLine("{0}", cust.CompanyName);
```

На заметку! Код в листинге 1.3 требует добавления сборки System.Data.Linq.dll к списку ссылок проекта, если это не сделано ранее. Также обратите внимание, что я добавил директиву using для пространства имен System.Data.Linq.

Вы можете видеть здесь, что я добавил директиву using для пространства имен nwind. Для этого примера вы должны использовать утилиту командной строки SQLMetal или Object Relational Designer, чтобы сгенерировать сущностные классы для целевой базы данных, которой в данном случае является база данных примеров Microsoft Northwind. Как это делается с SQLMetal — читайте в главе 12. Сгенерированные сущностные классы создаются в пространстве имен nwind, которое специфицировал при их генерации. Затем я добавил сгенерированный SQLMetal исходный модуль в проект, а также директиву using для пространства имен nwind.

На заметку! Вам может понадобиться изменить строку соединения, переданную конструктору Northwind в листинге 1.3, чтобы правильно установить соединение. В разделе “DataContext() и [Your]DataContext()” главы 16 описаны разных способах подключения к базе данных.

Запуск предыдущего кода нажатием <Ctrl+F5> выведет следующие данные в окно консоли:

```
Hanari Carnes
Que Delicia
Ricardo Adocicados
```

Этот пример демонстрирует опрос таблицы Customers из базы данных Northwind на предмет списка заказчиков из Рио-де-Жанейро (Rio de Janeiro). Хотя может показаться, что здесь не происходит ничего нового или особенного, чего нельзя было бы получить существующими средствами, все же на самом деле есть серьезные отличия. Важнее всего то, что этот запрос интегрирован в язык, а это значит, что я получаю поддержку уровня языка, включающую проверку синтаксис и IntelliSense. Ушли в прошлое те дни, когда запрос SQL записывался в строку, и невозможно было обнаружить ошибку до тех

² В настоящее время LINQ to SQL поддерживает только SQL Server.

пор. пока код не будет выполнен. Вы хотите сделать вашу конструкцию `where` зависящей от поля таблицы `Customers`, но не помните имени этого поля? IntelliSense покажет вам поля таблицы. Как только вы введете `c.` в предыдущем примере, IntelliSense отобразит все поля таблицы `Customers`.

Все предыдущие запросы используют синтаксис выражений запросов. Из главы 12 вы узнаете, что существует два синтаксиса запросов LINQ, один из которых — синтаксис выражений запросов. Конечно, вы можете всегда использовать синтаксис *стандартной точечной нотации*, который вы привыкли видеть в C#. Этот синтаксис предлагает нормальный шаблон вызовов `объект.метод()`, который вы обычно применяете.

Введение

По мере взросления платформы Microsoft .NET и поддерживаемых ею языков C# и VB, стало ясно, что одной из наиболее проблемных областей для разработчиков остается доступ к данным из разных источников. В частности, доступ к базе данных и манипуляции XML часто в лучшем случае запутаны, а в худшем — проблематичны.

Проблемы баз данных многочисленны. Во-первых, сложность представляет то, что вы не можете программно взаимодействовать с базой данных на уровне естественного языка. Это приводит к синтаксическим ошибкам, которые не обнаруживаются вплоть до момента запуска. Неправильные ссылки на поля базы данных также не обнаруживаются. Это может пагубно отразиться на программе, особенно если произойдет во время выполнения кода обработки ошибок. Нет ничего хуже, чем крах механизма обработки ошибок из-за синтаксически неверного кода, который никогда не тестировался. Иногда это неизбежно из-за непредсказуемого поведения ошибки. Наличие кода базы данных, который не проверяется во время компиляции, определенно может привести к этой проблеме.

Вторая проблема связана с неудобством, которое вызвано различными типами данных, используемыми определенным доменом данных, таким как разница между типами базы данных или типами данных XML и естественным языком, на котором написана программа. В частности, серьезно докучать могут времена и даты.

Анализ, итерация и манипуляция XML могут быть достаточно утомительными. Часто фрагмент XML — это все, что вам нужно, но из-за W3C DOM XML API объект `XmlDocument` должен быть создан только для того, чтобы выполнять различные операции над фрагментом XML.

Вместо того чтобы просто добавить больше классов и методов для постепенного восполнения этих недостатков, команда разработчиков Microsoft решила пойти на один шаг дальше в абстрагировании основ запросов данных из этих конкретных доменов данных. В результате появился LINQ. LINQ — это технология Microsoft, предназначенная для обеспечения механизма поддержки уровня языка для опроса данных всех типов. Эти типы включают массивы и коллекции в памяти, базы данных, документы XML и многое другое.

LINQ для запросов данных

Большей частью LINQ ориентирован на запросы — будь то запросы, возвращающие набор подходящих объектов, единственный объект или подмножество полей из объекта или множества объектов. В LINQ этот возвращенный набор объектов называется *последовательностью* (*sequence*). Большинство последовательностей LINQ имеют тип `IEnumerable<T>`, где `T` — тип данных объектов, находящихся в последовательности. Например, если у вас есть последовательность целых чисел, они должны храниться в переменной типа `IEnumerable<int>`. Вы увидите, что `IEnumerable<T>` буквально соответствует в LINQ. Очень многие методы LINQ возвращают `IEnumerable<T>`.

Компоненты

Поскольку LINQ настолько мощный, вы можете ожидать появления множества систем и продуктов, совместимых с LINQ. Почти любое хранилище данных может быть хорошим кандидатом на поддержку запросов LINQ. Сюда относятся базы данных, система Microsoft Active Directory, системный реестр, файловая система, файл Excel и т.д.

Microsoft уже идентифицировала компоненты, представленные в настоящем разделе, как необходимые для LINQ. Нет никаких сомнений в том, что их количество будет расти.

LINQ to Objects

LINQ to Objects — название, данное `IEnumerable<T>` API для стандартных операций запросов (Standard Query Operators). Именно LINQ to Objects позволяет вам выполнять запросы к массивам и находящимся в памяти коллекциям данных. Стандартные операции запросов — это статические методы класса `System.Linq.Enumerable`, которые вы используете для создания запросов LINQ to Objects.

LINQ to XML

LINQ to XML — название, присвоенное интерфейсу LINQ API, предназначенному для работы с XML. Этот интерфейс был ранее известен как `XLinq` в предварительных выпусках LINQ. Microsoft не только добавила необходимые библиотеки XML для работы с LINQ, но также восполнила недостатки стандартной модели XML DOM, тем самым существенно облегчив работу с XML. Прошли времена, когда нужно было создавать `XmlDocument` только для того, чтобы поработать с маленьким кусочком XML. Чтобы воспользоваться преимуществами LINQ to XML, вы должны иметь в своем проекте ссылку на сборку `System.Xml.Linq.dll` и директиву `using` следующего вида:

```
using System.Xml.Linq;
```

LINQ to DataSet

LINQ to DataSet — наименование, присвоенное программному интерфейсу LINQ API для `DataSet`. У многих разработчиков есть масса кода, полагающегося на `DataSet`. Те, кто не хотят отставать от новых веяний, но и не готовы переписывать свой код, благодаря этому интерфейсу могут воспользоваться всей мощью LINQ.

LINQ to SQL

LINQ to SQL — наименование, присвоенное программному интерфейсу `IQueryable<T>`, позволяющему запросам LINQ работать с базой данных Microsoft SQL Server. Этот интерфейс в ранних выпусках LINQ назывался `DLinq`.

Чтобы воспользоваться преимуществами LINQ to SQL, вы должны иметь в своем проекте ссылку на сборку `System.Data.Linq.dll`, а также следующую директиву `using` в исходном коде:

```
using System.Data.Linq;
```

LINQ to Entities

LINQ to Entities — альтернативный интерфейс LINQ API, используемый для обращения к базе данных. Он отделяет сущностную объектную модель от физической базы данных, вводя логическое отображение между ними двумя. С таким отделением возрастают мощь и гибкость, но также растет и сложность. Поскольку LINQ to Entities стоит в стороне от ядра каркаса LINQ, мы не описываем его в настоящей книге. Однако если

вы считаете, что вам нужна более высокая гибкость, чем обеспечиваемая LINQ to SQL, может быть, вам стоит рассмотреть эту альтернативу. В частности, если вам нужно ослабить связь между вашей сущностной объектной моделью и базой данных, если ваши сущностные объекты конструируются из нескольких таблиц или вам нужна большая гибкость в моделировании ваших сущностных объектов, то в этом случае LINQ to Entities может стать оптимальным выбором.

Как получить LINQ

Технически отдельного продукта LINQ, который нужно было бы получать отдельно, не существует. LINQ — это просто кодовое имя проекта средства запросов, добавленного в C# 3.0 и .NET Framework 3.5, которое впервые было представлено в очередной версии Visual Studio — Visual Studio 2008.

Чтобы получить актуальную информацию о LINQ и Visual Studio 2008, посетите <http://www.linqdev.com> и <http://apress.com/book/bookDisplay.html?bID=10241>.

LINQ — не только для запросов

Может показаться, что LINQ — это нечто, связанное только с запросами, поскольку расшифровывается, как язык интегрированных запросов (Language Integrated Query). Однако не думайте о нем лишь в этом контексте. Лично я предпочитаю воспринимать LINQ как механизм итерации данных (data iteration engine), но возможно в Microsoft не захотели называть технологию DIE (“умереть”).

Приходилось ли вам когда-нибудь вызывать метод, который возвращает обратно данные в некоторой структуре, которую вам затем приходилось конвертировать в еще одну структуру данных, прежде чем передать другому методу? Скажем, для примера, что вы вызываете метод A, и этот метод возвращает массив типа `string`, содержащий числовые значений в виде строк. Затем вам нужно вызвать метод B, но метод B требует массива целых чисел. Обычно вам приходится организовывать цикл для прохода по массиву строк и наполнения вновь сконструированного массива целых чисел. Какая досада! Позвольте мне указать на мощь Microsoft LINQ.

Предположим, что имеется массив строк, которые я принял от метода A, как показано в листинге 1.4.

Листинг 1.4. Преобразования массива строк в массив целых

```
string[] numbers = { "0042", "010", "9", "27" };
```

Для этого примера объявлен статический массив строк. Теперь перед вызовом метода B нужно преобразовать массив строк в массив целых чисел:

```
int[] nums = numbers.Select(s => Int32.Parse(s)).ToArray();
```

Вот и все. Что может быть проще? Даже если вы скажете “абракадабра”, это сэкономит вам всего 48 символов.

А вот код, необходимый для отображения результирующего массива целых чисел:

```
foreach(int num in nums)
    Console.WriteLine(num);
```

Вывод будет выглядеть так:

Я знаю, о чём вы подумали: может, я просто отсек ведущие пробелы? Но если я отсортирую результат, убедит ли вас это? Если бы это были по-прежнему строки, то 9 окажется в конце, а 10 — в начале. Листинг 1.5 содержит некоторый код, который выполняет преобразование и сортирует вывод.

Листинг 1.5. Преобразования массива строк в массив целых с последующей сортировкой

```
string[] numbers = { "0042", "010", "9", "27" };
int[] nums = numbers.Select(s => Int32.Parse(s)).OrderBy(s => s).ToArray();
foreach(int num in nums)
    Console.WriteLine(num);
```

И вот результат:

```
9
10
27
42
```

Не правда ли — гладко? Хорошо, скажете вы, все это прекрасно, но ведь это всего лишь простой пример. Давайте рассмотрим более сложный.

Скажем, у вас есть некоторый код, содержащий класс Employee. В этом классе Employee есть метод, возвращающий всех сотрудников. Также предположим, что у вас есть другой код, включающий класс Contact, и в этом классе — метод, публикующий контакты. Предположим, что вам нужно опубликовать всех сотрудников, как контакты.

Задача кажется достаточно простой, но здесь таится ловушка. Общий метод Employee, который извлекает всех сотрудников, возвращает их в списке ArrayList, хранящем объекты Employee, а метод Contact, публикующий контакты, требует массива объектов типа Contact. Ниже показан обычный для такого случая код.

```
namespace LINQDev.HR
{
    public class Employee
    {
        public int id;
        public string firstName;
        public string lastName;
        public static ArrayList GetEmployees()
        {
            // Конечно, реальный код должен был
            // бы выполнить запрос к базе данных.
            ArrayList al = new ArrayList();
            // Не правда ли, новое средство инициализации
            // объектов C# превращает это в пару пустяков?
            al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
            al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
            al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
            return(al);
        }
    }
}

namespace LINQDev.Common
{
    public class Contact
    {
        public int Id;
        public string Name;
```

```
public static void PublishContacts(Contact[] contacts)
{
    // Этот метод публикации просто пишет их в окно консоли.
    foreach(Contact c in contacts)
        Console.WriteLine("Contact Id: {0} Contact: {1}", c.Id, c.Name);
}
```

Как видите, класс Employee и метод GetEmployees находятся в одном пространстве имен — LINQDev.HR, и метод GetEmployees возвращает ArrayList. Метод PublishContacts находится в другом пространстве имен — LINQDev.Common, и требует передачи ему массива объектов Contact.

Ранее это всегда требовало итерации по ArrayList, который возвращен методом GetEmployees, и создания нового массива типа Contact для передачи методу PublishContacts. LINQ значительно облегчает задачу, как показано в листинге 1.6.

Листинг 1.6. Вызов обычного кода

```
ArrayList alEmployees = LINQDev.HR.Employee.GetEmployees();
LINQDev.Common.Contact[] contacts = alEmployees
    .Cast<LINQDev.HR.Employee>()
    .Select(e => new LINQDev.Common.Contact {
        Id = e.id,
        Name = string.Format("{0} {1}", e.firstName, e.lastName)
    })
    .ToArray<LINQDev.Common.Contact>();
LINQDev.Common.Contact.PublishContacts(contacts);
```

Чтобы преобразовать ArrayList объектов Employee в массив объектов Contact, я сначала привожу ArrayList объектов Employee к последовательности IEnumerable<Employee>, используя стандартную операцию запросов Cast. Это необходимо, потому что использован унаследованный класс коллекций ArrayList. Синтаксически говоря, в коллекции ArrayList хранятся объекты класса System.Object, а не объекты типа класса Employee. Поэтому я должен привести их к объектам Employee. Если бы метод GetEmployees возвращал обобщенную коллекцию List, необходимости в этом не было бы. Однако этот тип коллекции был недоступным, когда был написан унаследованный код.

Затем язываю операцию Select на возвращенной последовательности объектов Employee, и в лямбда-выражении — коде, переданном внутри вызова метода Select, — создаю и инициализирую экземпляр объекта Contact, используя для этого новые средства инициализации объектов C# 3.0, чтобы присвоить значения входного элемента Employee вновь сконструированному выходному элементу Contact. Лямбда-выражение (lambda expression) — новое средство C# 3.0, представляющее собой новое сокращение для специфирования анонимных методов, которое будет объяснено в главе 2. И, наконец, я преобразую последовательность вновь сконструированных объектов Contact в массив объектов Contact, применяя для этого операциюToArray, потому что этого требует метод PublishContacts. Разве не изящно? А вот результат:

```
Contact Id: 1 Contact: Joe Rattz
Contact Id: 2 Contact: William Gates
Contact Id: 3 Contact: Anders Hejlsberg
```

Как видите, LINQ может делать многое помимо запросов данных. По мере чтения глав нашей книги задумайтесь о дополнительных применениях средств, представленных LINQ.

Советы начинающим

Работая с LINQ при написании этой книги, я часто чувствовал себя запутанным, сбитым с толку и несообразительным. Несмотря на то что доступно много очень полезных ресурсов для разработчиков, которые желают изучать применение LINQ во всей его потенциальной мощи, все же я хочу привести несколько небольших советов, чтобы помочь вам начать. В некоторых отношениях эти советы кажутся более уместными в конце книги. В конце концов, я еще не объяснил некоторые упоминающиеся в них концепции. Но было бы, мягко говоря, некорректно заставлять вас сначала прочесть всю книгу лишь только для того, чтобы добраться до полезных советов в ее конце. С учетом всего сказанного, в данном разделе содержатся некоторые советы, которые, как я полагаю, вы сочтете полезными, даже не полностью понимая их или их контекст.

Когда запутались, используйте ключевое слово var

В то время как необходимо использовать ключевое слово `var` при захвате последовательности от анонимных классов в переменную, иногда это также удобный путь заставить код компилироваться, когда вы запутались со сложными обобщенными типами. Хотя я отдаю предпочтение разработчикам, которые точно знают, какого типа данные содержатся в последовательности — в том смысле, что для `IEnumerable<T>` вы должны знать, какой тип представляет `T` — иногда, особенно начиная работать с LINQ, это может вводить в заблуждение. Если вы обнаруживаете, что код не компилируется из-за определенного рода несоответствия типов данных, попробуйте заменить явно установленные типы переменных на указанные с применением ключевым словом `var`.

Например, предположим, что есть такой код:

```
// Этот код не компилируется.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<?> orders = db.Customers
    .Where(c => c.Country == "USA" && c.Region == "WA")
    .SelectMany(c => c.Orders);
```

Может быть неясно, каков тип данных у последовательности `IEnumerable`. Вы знаете, что это `IEnumerable` некоторого типа `T`, но что есть `T`? Удобный трюк состоит в присвоении результата запроса переменной, чей тип специфицирован ключевым словом `var`, нежели указывать тип переменной с необходимостью точного знания, что за типом является `T`. В листинге 1.7 показано как должен выглядеть такой код.

Листинг 1.7. Пример кода, использующего ключевое слово var

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var orders = db.Customers
    .Where(c => c.Country == "USA" && c.Region == "WA")
    .SelectMany(c => c.Orders);
Console.WriteLine(orders.GetType());
```

В этом примере обратите внимание на то, что теперь тип переменной `orders` указан с использованием ключевого слова `var`. Запуск этого кода даст следующий вывод:

```
System.Data.Linq.DataQuery`1[nwind.Order]
```

Здесь используется загадочный жаргон компилятора, но для нас интерес представляет часть `nwind.Order`. Теперь вы знаете, что тип данных полученной последовательности — `nwind.Order`.

Если вас смущает такой вывод, запуск примера в отладчике и просмотр переменной orders в окне Locals (Локальные) покажет вам такой тип данных orders:

```
System.Linq.IQueryable<nwind.Order> {System.Data.Linq.DataQuery<nwind.Order>}
```

Это проясняет, что вы имеете последовательность объектов nwind.Order. Технически вы получаете здесь IQueryable<nwind.Order>, но, если хотите, это можно присвоить IEnumerable<nwind.Order>, поскольку IQueryable<T> наследуется от IEnumerable<T>. Таким образом, вы можете переписать предыдущий код, плюс выполнить перечисление результатов, как показано в листинге 1.8.

Листинг 1.8. Пример кода из листинга 1.7, но с явными типами

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Order> orders = db.Customers
    .Where(c => c.Country == "USA" && c.Region == "WA")
    .SelectMany(c => c.Orders);
foreach(Order item in orders)
    Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

На заметку! Чтобы приведенный код работал, у вас должна быть директива using для пространства имен System.Collections.Generic в дополнение к пространству имен System.Linq, которого следует ожидать при работе с кодом LINQ.

Этот код должен дать следующий сокращенный результат:

```
3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
...
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets
```

Использование операций Cast или OfType для унаследованных коллекций

Вы обнаружите, что большинство стандартных операций запросов LINQ могут быть вызваны на коллекциях, реализующих интерфейс IEnumerable<T>. Ни одна из унаследованных коллекций C# из пространства имен System.Collection не реализует IEnumerable<T>. Поэтому возникает вопрос — как использовать LINQ с унаследованными коллекциями из вашего существующего кода?

Есть две стандартных операции запросов, специально предназначенных для этой цели — Cast и OfType. Обе они могут быть использованы для преобразования унаследованных коллекций в последовательности IEnumerable<T>. В листинге 1.9 показан пример.

Листинг 1.9. Преобразование унаследованной коллекции в IEnumerable<T> с применением операции Cast

```
// Построим унаследованную коллекцию.
ArrayList arrayList = new ArrayList();
// Конечно, можно было бы использовать здесь инициализацию коллекций,
// но это не работает с унаследованными коллекциями.
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");
IEnumerable<string> names = arrayList.Cast<string>().Where(n => n.Length < 7);
foreach(string name in names)
    Console.WriteLine(name);
```

В листинге 1.10 представлен пример использования операции OfType.

Листинг 1.10. Использование операции OfType

```
// Построим унаследованную коллекцию.
ArrayList arrayList = new ArrayList();
// Конечно, можно было бы использовать здесь инициализацию коллекций,
// но это не работает с унаследованными коллекциями.
arrayList.Add("Adams");
arrayList.Add("Arthur");
arrayList.Add("Buchanan");
IEnumerable<string> names = arrayList.OfType<string>().Where(n => n.Length < 7);
foreach(string name in names)
    Console.WriteLine(name);
```

Оба примера дают одинаковый результат:

```
Adams
Arthur
```

Разница между двумя операциями в том, что Cast пытается привести все элементы в коллекции к указанному типу, помещая их в выходную последовательность. Если в коллекции есть объект типа, который не может быть приведен к указанному, генерируется исключение. Операция OfType пытается поместить в выходную последовательность только те элементы, которые могут быть приведены к специфицированному типу.

Отдавайте предпочтение операции OfType перед Cast

Одной из наиболее важных причин добавления обобщений в C# была необходимость дать языку возможность создавать коллекции со статическим контролем типов. До появления обобщений приходилось создавать собственные специфические типы коллекций для каждого типа данных, которые нужно было в них хранить — не было никакой возможности гарантировать, чтобы каждый элемент, помещаемый в унаследованную коллекцию, был одного и того же корректного типа. Ничего в языке не мешало коду добавить объект TextBox в ArrayList, предназначенный для хранения только объектов Label.

С появлением обобщений в C# 2.0 разработчики получили в свои руки способ явно устанавливать, что коллекция может содержать только элементы определенного указанного типа. Хотя и операция OfType, и операция Cast могут работать с унаследованными коллекциями, Cast требует, чтобы каждый объект в коллекции был правильного типа, что было основным фундаментальным недостатком унаследованных коллекций, из-за которого были введены обобщения. При использовании операции Cast, если любой из объектов в коллекции не может быть приведен к указанному типу данных, генерируется исключение. Поэтому используйте операцию OfType. С ней в выходной последовательности IEnumerable<T> будут сохранены только объекты указанного типа, и никаких исключений генерироваться не будет. В лучшем случае все объекты будут правильного типа и все попадут в выходную последовательность. В худшем случае некоторые элементы будут пропущены, но в случае применения операции Cast они бы привели к исключению.

Не рассчитывайте на безошибочность запросов

В главе 3 речь пойдет о том, что запросы LINQ являются отложенными (deferred) и на самом деле не выполняются, когда вы инициируете их. Например, рассмотрим следующий фрагмент кода из листинга 1.1:

```

var items =
    from s in greetings
    where s.EndsWith("LINQ")
    select s;
foreach (var item in items)
    Console.WriteLine(item);

```

Хотя, кажется, что запрос выполняется при инициализации переменной `items`, на самом деле это не так. Поскольку операции `Where` и `Select` являются отложенными, запрос на самом деле не выполняется в этой точке. Запрос просто вызывается, объявляется, или определяется, но не выполняется. Все начинает происходить тогда, когда из него извлекается первый результат. Это обычно происходит тогда, когда выполняется перечисление результатов переменной запроса. В данном примере результат запроса не востребован до тех пор, пока не запустится оператор `foreach`. Именно в этой точке будет выполнен запрос. Таким образом, мы говорим, что запрос является *отложенным*.

Очень легко забыть о том, что многие операции запросов являются отложенными и не будут выполнены до тех пор, пока не начнется перечисление результатов. Это значит, что можно иметь неправильно написанный запрос, который генерирует исключение только тогда, когда начнется перечисление его результатов. Такое перечисление может начаться намного позже, так что можно легко забыть, что причиной неприятностей был неправильный запрос.

Рассмотрим код в листинге 1.11.

Листинг 1.11. Запрос с преднамеренным исключением, отложенным до перечисления

```

string[] strings = { "one", "two", null, "three" };
Console.WriteLine("Before Where() is called.");
IEnumerable<string> ieStrings = strings.Where(s => s.Length == 3);
Console.WriteLine("After Where() is called.");
foreach(string s in ieStrings)
{
    Console.WriteLine("Processing " + s);
}

```

Я знаю, что третий элемент в массиве строк — `null`, и я не могу вызвать `null.Length` без генерации исключения. Выполнение кода благополучно пройдет строку, где вызывается запрос. И все будет хорошо до тех пор, пока я не начну перечисление последовательности `ieStrings`, и не доберусь до третьего элемента, где возникнет исключение. Вот результат этого кода, подтверждающий мои слова:

```

Before Where() is called.
After Where() is called.
Processing one
Processing two

```

```

Unhandled Exception: System.NullReferenceException: Object reference not set to an
instance of an object.
~
```

Как видите, я вызвал операцию `Where` без исключения. Оно не появилось до тех пор, пока я не попытался в перечислении обратиться к третьему элементу последовательности, где и возникло исключение. Теперь представьте, что последовательность `ieStrings` передана функции, которая дальше выполняет перечисление последовательности — возможно, чтобы наполнить выпадающий список или какой-то другой элемент управления. Легко подумать, что исключение вызвано сбоем в этой функции, а не самим запросом LINQ.

Используйте преимущество отложенных запросов

В главе 3 я раскрою тему отложенных запросов более глубоко. Однако хочу отметить, что если запрос является отложенным, который в конечном итоге возвращает `IEnumerable<T>`, этот объект `IEnumerable<T>` может перечисляться снова и снова, получая последние данные из источника. Вам не нужно ни вызывать, ни, как отмечалось ранее, объявлять запрос заново.

В большинстве примеров кода этой книги вы увидите вызов запроса и возврат `IEnumerable<T>` для некоторого типа `T`, сохраняемый в переменной. Затем я обычно вызываю `foreach` на последовательности `IEnumerable<T>`. Это делается в целях демонстрации. Если код выполняется много раз, повторный вызов запроса — лишняя работа. Более оправданным может быть наличие метода инициализации запроса, который вызывается однажды в жизненном цикле контекста, и там конструировать все запросы. Затем вы можете выполнять перечисление конкретной последовательности, чтобы получить последнюю версию результатов запроса.

Использование Log из `DataContext`

При работе с LINQ to SQL не забудьте, что класс базы данных, генерируемый `SQLMetal`, унаследован от `System.Data.Linq.DataContext`. Это значит, что ваш сгенерированный класс `DataContext` имеет некоторую полезную встроенную функциональность, такую как объект `TextWriter` по имени `Log`.

Одной из прелестей объекта `Log` является то, что он выводит эквивалентный SQL-оператор запроса `IQueryable<T>` до подстановки параметров. Случалось ли вам сталкиваться с отказом кода в рабочей среде, который, как вам кажется, вызван данными? Не правда ли, было бы хорошо запустить запрос на базе данных, вводя его в `SQL Enterprise Manager` или `Query Analyzer`, чтобы увидеть в точности, какие данные он возвращает? Объект `Log` класса `DataContext` выведет для вас запрос SQL. Пример показан в листинге 1.12.

Листинг 1.12. Пример использования объекта `DataContext.Log`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
IQueryable<Order> orders = from c in db.Customers
                             from o in c.Orders
                             where c.Country == "USA" && c.Region == "WA"
                             select o;
foreach(Order item in orders)
    Console.WriteLine("{0} - {1} - {2}", item.OrderDate, item.OrderID, item.ShipName);
```

Этот код производит следующий вывод:

```
SELECT [t1].[OrderID], [t1].[CustomerID], [t1].[EmployeeID], [t1].[OrderDate],
[t1].[RequiredDate], [t1].[ShippedDate], [t1].[ShipVia], [t1].[Freight],
[t1].[ShipName], [t1].[ShipAddress], [t1].[ShipCity], [t1].[ShipRegion],
[t1].[ShipPostalCode], [t1].[ShipCountry]
FROM [dbo].[Customers] AS [t0], [dbo].[Orders] AS [t1]
WHERE ([t0].[Country] = @p0) AND ([t0].[Region] = @p1) AND ([t1].[CustomerID] =
[t0].[CustomerID])
-- @p0: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p1: Input String (Size = 2; Prec = 0; Scale = 0) [WA]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
3/21/1997 12:00:00 AM - 10482 - Lazy K Kountry Store
5/22/1997 12:00:00 AM - 10545 - Lazy K Kountry Store
6/19/1997 12:00:00 AM - 10574 - Trail's Head Gourmet Provisioners
```

```

6/23/1997 12:00:00 AM - 10577 - Trail's Head Gourmet Provisioners
1/8/1998 12:00:00 AM - 10822 - Trail's Head Gourmet Provisioners
7/31/1996 12:00:00 AM - 10269 - White Clover Markets
11/1/1996 12:00:00 AM - 10344 - White Clover Markets
3/10/1997 12:00:00 AM - 10469 - White Clover Markets
3/24/1997 12:00:00 AM - 10483 - White Clover Markets
4/11/1997 12:00:00 AM - 10504 - White Clover Markets
7/11/1997 12:00:00 AM - 10596 - White Clover Markets
10/6/1997 12:00:00 AM - 10693 - White Clover Markets
10/8/1997 12:00:00 AM - 10696 - White Clover Markets
10/30/1997 12:00:00 AM - 10723 - White Clover Markets
11/13/1997 12:00:00 AM - 10740 - White Clover Markets
1/30/1998 12:00:00 AM - 10861 - White Clover Markets
2/24/1998 12:00:00 AM - 10904 - White Clover Markets
4/17/1998 12:00:00 AM - 11032 - White Clover Markets
5/1/1998 12:00:00 AM - 11066 - White Clover Markets

```

Используйте форум LINQ

Несмотря на то что я привел наилучшие советы, которые мог придумать, все же более чем вероятно то, что периодически будете оказываться в тупике. Помните, что на MSDN.com существует форум, посвященный LINQ. Здесь вы можете найти ссылку на него: <http://www.linqdev.com>. Этот форум отслеживается разработчиками Microsoft, и вы можете почерпнуть массу полезных знаний.

Резюме

Я чувствую, что вам уже не терпится перейти к следующей главе, но сначала я хотел бы напомнить вам несколько вещей.

Во-первых, LINQ имеет шанс изменить способ, которым разработчики .NET выполняют запросы данных. Поставщики программного обеспечения более чем вероятно будут стараться приклеивать метки "LINQ-совместимый" к своим продуктам, как сейчас они поступают с XML.

Следует иметь в виду, что LINQ — это не просто новая библиотека, которую нужно добавить к вашему проекту. Это тотальный подход к опросу данных, объединяющий несколько компонентов в зависимости от типа опрашиваемого хранилища данных. В настоящее время вы можете применять LINQ для запросов к следующим источникам данных: коллекциям, находящимся в памяти, используя LINQ to Objects; к XML, используя LINQ to XML; DataSet — с помощью LINQ to DataSet, и базам данных SQL Server, используя LINQ to SQL.

Также помните, что LINQ предназначен не только для запросов. Работая над проектом примера, я использовал LINQ и обнаружил, что LINQ весьма полезен не только для опроса данных, но также для их преобразования в необходимый формат для представления в элементе управления WinForm.

И последнее (по порядку, но не по важности): я надеюсь, что вы не станете пренебречь советами, приведенными в этой главе. Ничего, если вы не понимаете некоторые из них. Они проясняются в процессе чтения книги. Просто вспоминайте о них, когда попадаете в сложную ситуацию.

Нет сомнений в том, что увидев некоторые примеры LINQ и советы этой главы, вы почувствуете себя озадаченными некоторыми деталями синтаксиса, который делает все это возможным. Если это так, не беспокойтесь, потому что в следующей главе я опишу расширения, которые Microsoft добавила в C# версии 3.0, позволившие достичь такого результата.

ГЛАВА 2

Расширения языка C# 3.0 для LINQ

В предыдущей главе предлагалось введение в LINQ. Было приведено несколько примеров, чтобы разжечь ваш аппетит, и выдано несколько предварительных советов. Возможно, вы были озадачены синтаксисом. Если да, это, вероятно потому, что язык C#, с которым вы столкнулись в этой главе, является новым и усовершенствованным. Причина обновления языка C# состояла в том, что C# 2.0 просто не имел достаточных средств для поддержки LINQ. В настоящей главе вы ознакомитесь с более мощной версией C# 3.0.

Новые дополнения языка C# 3.0

Чтобы обеспечить гладкую интеграцию LINQ с C#, в язык C# потребовалось внести существенные усовершенствования. Почти каждое заметное расширение, добавленное в язык C# версии 3.0, специально предназначено для поддержки LINQ. Хотя все эти средства ценные и сами по себе, на самом деле они являются частями общего вклада в LINQ, который делает C# 3.0 столь замечательным.

Чтобы действительно понять большую часть синтаксиса LINQ, необходимо разобраться в некоторых новых средствах языка C# 3.0, прежде чем приступить к работе с компонентами LINQ. В этой главе будут раскрыты следующие дополнения языка:

- лямбда-выражения;
- деревья выражений;
- ключевое слово var, инициализация объектов и коллекций и анонимные типы;
- расширяющие методы;
- частичные методы;
- выражения запросов.

В примерах этой главы я не показываю явно, какие сборки должны быть добавлены и какие пространства имен специфицированы директивами using из тех, что я упоминал в главе 1. Однако я укажу некоторые новые, но только в тех примерах, где они встречаются впервые.

Лямбда-выражения

В C# 3.0 Microsoft добавила так называемые лямбда-выражения (lambda expressions). Лямбда-выражения использовались в языках программирования вроде LISP задавна, а впервые были сформулированы в 1936 г. американским математиком Алонзо Черчем

(Alonzo Church). Эти выражения представляют сокращенный синтаксис для спецификации алгоритма.

Но, прежде чем обратиться непосредственно к лямбда-выражениям, давайте взглянем на эволюцию способов спецификации алгоритма как аргумента метода, поскольку именно в этом и состоит назначение лямбда-выражений.

Использование именованных методов

До появления C# 2.0, когда метод или переменная была типизирована так, что требовала delegate, разработчик должен был создавать именованный метод и передавать его имя туда, где требовался delegate.

В качестве примера рассмотрим следующую ситуацию. Предположим, что у нас есть два разработчика, один из которых — разработчик кода общего назначения, а другой — прикладной разработчик. Не обязательно, чтобы это были два разных разработчика, я просто хочу разграничить две разные роли. Разработчик кода общего назначения хочет создавать код общего применения, которые может быть использован многократно во всем проекте. Прикладной разработчик будет потреблять код общего назначения, чтобы создавать приложение. В этом случае разработчик общего кода желает создать метод для фильтрации массивов целых чисел, но с возможностью специфицировать алгоритм, применяемый для фильтрации. Для начала он должен объявить delegate. Этот делегат будет прототипирован для приема параметра int и возврата значения true, если данный int должен быть включен в отфильтрованный массив.

Итак, он создает служебный класс и добавляет делегат и метод фильтрации. Вот этот код общего назначения:

```
public class Common
{
    public delegate bool IntFilter(int i);
    public static int[] FilterArrayOfInts(int[] ints, IntFilter filter)
    {
        ArrayList aList = new ArrayList();
        foreach (int i in ints)
        {
            if (filter(i))
            {
                aList.Add(i);
            }
        }
        return ((int[])aList.ToArray(typeof(int)));
    }
}
```

Разработчик кода общего применения поместит и объявление delegate, и FilterArrayOfInts в общую библиотечную сборку — динамическую библиотеку DLL, чтобы его можно было использовать во многих приложениях.

Метод FilterArrayOfInts, приведенный выше, позволяет прикладному разработчику передавать ему массив целых чисел и delegate его метода фильтрации, получая обратно отфильтрованный массив.

Теперь представим, что прикладной разработчик хочет отфильтровать только нечетные числа. Вот его метод фильтрации, который объявлен в его прикладном коде:

```
public class Application
{
    public static bool IsOdd(int i)
    {
        return ((i & 1) == 1);
    }
}
```

На основе кода метода `FilterArrayOfInts` этот метод будет вызван для каждого значения `int` в массиве, переданном ему. Этот фильтр вернет `true`, если переданное значение `int` является нечетным. В листинге 2.1 демонстрируется пример использования метода `FilterArrayOfInts`, за которым представлен результат.

Листинг 2.1. Вызов метода фильтрации из общей библиотеки

```
using System.Collections;
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, Application.IsOdd);
foreach (int i in oddNums)
    Console.WriteLine(i);
```

Вот результат:

```
1
3
5
7
9
```

Обратите внимание, что для передачи `delegate` в виде второго параметра `FilterArrayOfInts` прикладной разработчик просто передает имя метода. Просто создав другой фильтр, он может фильтровать числа иначе. Он может иметь фильтр для четных чисел, простых чисел или отобранных в соответствии с любым нужным ему критерием. Делегаты позволяют создавать в высшей степени многократно используемый код.

Использование анонимных методов

Это все хорошо, но написание всех этих методов фильтрации или любых других методов `delegate` может оказаться довольно утомительным. Многие из этих методов будут использованы лишь однократно, и скучно создавать именованные методы для таких случаев. Начиная с C# 2.0, у разработчиков появилась возможность использовать анонимные методы, чтобы передавать встроенный код, подставляя его вместо `delegate`. Анонимные методы позволяют разработчику специфицировать код практически везде, где обычно должен передаваться делегат. Вместо создания метода `IsOdd` он может специфицировать код фильтрации прямо в точке, где обычно передается `delegate`. В листинге 2.2 показан тот же код, что в листинге 2.1, но с использованием анонимного метода.

Листинг 2.2. Вызов метода фильтрации из общей библиотеки

```
using System.Collections;
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, delegate(int i) { return ((i & 1) == 1); });
foreach (int i in oddNums)
    Console.WriteLine(i);
```

Совсем неплохо. Прикладной разработчик более не обязан где-либо объявлять метод. Это замечательно для кода логики фильтрации, вероятность многократного использования которого невелика. Как и ожидалось, вывод программы не отличается от предыдущего:

```
1
3
5
7
9
```

Применение анонимных методов имеет один недостаток. Оно довольно громоздко и трудно читаемо. Должен существовать более удобный способ написания кода метода.

Использование лямбда-выражений

Лямбда-выражения специфицированы как разделенный запятыми список параметров, за которым следует лямбда-операция, а за ней — выражение или блок операторов. Если параметров более одного, входные параметры заключаются в скобки. В C# лямбда-операция записывается как =>. Таким образом, лямбда-выражение в C# выглядит подобно следующему:

```
(параметр1, параметр2, ..., параметрN) => выражение
```

Или, когда требуется более высокая сложность, может быть использован блок операторов:

```
(параметр1, параметр2, ..., параметрN) =>
{
    оператор1;
    оператор2;
    ...
    операторN;
    return (тип_возврата_лямбда_выражения);
}
```

В этом примере возвращаемый тип данных в конце блока операторов должен соответствовать типу возврата, указанному делегатом. Вот пример лямбда-выражения:

```
x => x
```

Это лямбда-выражение может быть прочитано, как “x идет к x”, или, возможно, “ввод x возвращает x”. Это значит, что при входной переменной x выражение вернет x. Это выражение просто возвращает то, что оно получило. Поскольку здесь только один единственный параметр x, нет необходимости заключать его в скобки. Важно знать, что этот делегат диктует тип входного параметра x и тип возврата. Например, если delegate определен как принимающий string, но возвращающий bool, тогда x => x не может быть использовано, потому что если входной x будет иметь тип string, то возвращаемый x также должен относиться к типу string, но делегат специфицирован как возвращающий bool. Поэтому с delegate, определенным таким образом, часть выражения справа от лямбда-операции (=>) должна вычисляться для возврата bool, как показано в следующем примере:

```
x => x.Length > 0
```

Это лямбда-выражение может быть прочитано как “x идет в x.Length > 0”, или, возможно, “ввод x возвращает x.Length > 0”. Поскольку правая часть выражения вычисляется как bool, делегат должен быть специфицирован, что метод возвращает bool, иначе компилятор выдаст ошибку.

Следующее лямбда-выражение попытается вернуть длину входного аргумента. Значит, делегат должен специфицировать тип возврата int:

```
s => s.Length
```

Если лямбда-выражению передается несколько параметров, отделяйте их запятыми и помешайте в скобки, как здесь:

```
(x, y) => x == y
```

Сложные лямбда-выражения могут быть даже специфицированы блоком операторов, как показано ниже:

```
(x, y) =>
{
    if (x > y)
        return (x);
    else
        return (y);
}
```

Что еще важно помнить, так это то, что `delegate` определяет, какими должны быть типы входных параметров и каким — тип возврата. Поэтому убедитесь, что лямбда-выражение соответствует определению `delegate`.

Внимание! Убедитесь, что ваше лямбда-выражение рассчитано на прием входных типов, специфицированных определением делегата, и возвращает тип, определенный как возвращаемый тип делегата.

Чтобы освежить вашу память, приведем объявление `delegate`, определенное разработчиком кода общего назначения:

```
delegate bool IntFilter(int i);
```

Лямбда-выражение прикладного разработчика должно поддерживать `int`, переданный в параметре, и возвращать `bool`. Это может быть выведено из вызываемого им метода и целью метода фильтрации, но важно помнить, что `delegate` диктует это.

Поэтому предыдущий пример, но на этот раз с использованием лямбда-выражения, должен выглядеть, как показано в листинге 2.3.

Листинг 2.3. Вызов метода фильтрации с лямбда-выражением

```
int[] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] oddNums = Common.FilterArrayOfInts(nums, i => ((i & 1) == 1));
foreach (int i in oddNums)
    Console.WriteLine(i);
```

Вот какой удобный код! Это может показаться поначалу забавным, но как только вы привыкнете, то оцените читабельность и сопровождаемость такого кода. Как и требовалось, результат будет тем же, что и у предыдущих примеров:

```
1
3
5
7
9
```

Для сравнения покажем, как выглядят важнейшие строки примера для каждого подхода:

```
int[] oddNums = // использование именованного метода
    Common.FilterArrayOfInts(nums, Application.IsOdd);
int[] oddNums = // использование анонимного метода
    Common.FilterArrayOfInts(nums, delegate(int i){return((i & 1) == 1);});
int[] oddNums = // использование лямбда-выражения
    Common.FilterArrayOfInts(nums, i => ((i & 1) == 1));
```

Конечно, первая строка на самом деле короче, но не забудьте, что там еще есть именованный метод, объявленный где-то еще, в котором определено то, что он делает. Конечно, если логика фильтрации должна использоваться в нескольких местах, или, возможно, алгоритм чересчур сложен и должен быть поручен специальному разработчику, может быть, имеет больше смысла создать именованный метод, который будут применять другие разработчики.

Совет. Сложные или многократно используемые алгоритмы лучше могут быть реализованы именованными методами, чтобы их можно было использовать повторно любым разработчиком, не требуя от него полного его понимания.

Использовать ли именованные методы, анонимные методы или лямбда-выражения — выбор за разработчиком. Используйте то, что имеет смысл в каждой конкретной ситуации.

Преимуществами лямбда-выражений вы часто пользуетесь, передавая их в качестве аргументов ваших вызовов операций запросов LINQ. Поскольку каждый запрос LINQ, вероятнее всего, будет иметь уникальное или очень ограниченно используемое лямбда-выражение, это обеспечивает гибкость специфирования логики вашей операции, без необходимости постоянного создания методов почти для каждого запроса.

Деревья выражений

Дерево выражений — эффективное представление в форме дерева данных лямбда-выражения операций запросов. Эти представления деревьев выражений могут быть вычислены все сразу, так что единственный запрос может быть построен и выполнен на одном источнике данных, таком как база данных.

В большинстве примеров, рассмотренных до сих пор, операции выражений выполнялись в последовательной линейной манере. Рассмотрим следующий код:

```
int[] nums = new int[] { 6, 2, 7, 1, 9, 3 };
IEnumerable<int> numsLessThanFour = nums
    .Where(i => i < 4)
    .OrderBy(i => i);
```

Этот запрос содержит две операции — `Where` и `OrderBy`, — которые ожидают делегатов методов в качестве своих аргументов. При компиляции этого кода генерируется код промежуточного языка .NET (IL), идентичный анонимному методу вместо каждого лямбда-выражения операций запросов.

Когда выполняется этот запрос, сначала вызывается операция `Where`, за ней — операция `OrderBy`. Такое линейное выполнение операций кажется оправданным для данного примера, но стоит подумать о запросе к очень большому источнику данных, такому как база данных. Имело бы смысл для запроса SQL сначала обратиться к базе данных только с оператором `where`, чтобы изменить порядок последующих вызовов? Конечно, это не реально для запросов к базе данных, как потенциально и для других типов запросов. Именно здесь приходят на помощь деревья выражений. Поскольку дерево выражений допускает параллельное вычисление и выполнение всех операций в запросе, может быть произведен единственный общий запрос вместо отдельных запросов для каждой операции.

Итак, теперь есть две разные вещи, которые может генерировать компилятор для лямбда-выражения операции — IL-код и дерево выражений. Что определяет — будет ли лямбда-выражения операции компилироваться в IL или дерево выражений? Прототип операции определит, какое из этих двух действий предпримет компилятор. Если операция объявлена для приема делегата метода, будет сгенерирован IL-код. Если же операция объявлена для приема выражения метода делегата, будет создано дерево выражений.

В качестве примера давайте рассмотрим две разные реализации операции `Where`. Первая — стандартная операция запроса, присутствующая в программном интерфейсе LINQ to Objects API и определенная в классе `System.Linq.Enumerable`:

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Вторая реализация операции `Where` находится в `LINQ to SQL API`, и принадлежит классу `System.Linq.Queryable`:

```
public static IQueryable<T> Where<T>(
    this IQueryable<T> source,
    System.Linq.Expressions.Expression<Func<int, bool>> predicate);
```

Как видите, первая операция `Where` объявлена так, что принимает делегат метода, на что указывает делегат `Func`, и компилятор для этого лямбда-выражения операции сгенерирует IL-код. Делегат `Func` рассматривается в главе 3. А пока просто имейте в виду, что он определяет сигнатуру делегата, передаваемого в качестве аргумента предиката. Вторая операция `Where` объявлена для приема дерева выражений (`Expression`), поэтому здесь компилятор генерирует древовидное представление лямбда-выражения.

На заметку! Методы расширений на последовательностях `IEnumerable<T>` имеют IL-код, сгенерированный компилятором. Расширяющие методы на `IQueryable<T>` имеют сгенерированные компилятором деревья выражений.

Ключевое слово `var`, инициализация объектов и анонимные типы

Должен предупредить: почти невозможно говорить о ключевом слове `var` и выведении неявных типов без демонстрации инициализации объектов или анонимных типов. И точно также почти невозможно обсуждать инициализацию объектов или анонимные типы, не упоминая ключевого слова `var`. Все эти три расширения языка C# тесно между собой связаны.

Прежде чем описать в деталях каждое из этих средств языка — поскольку каждое из них описывает себя в терминах других — я представлю все три вместе. Рассмотрим следующий оператор:

```
Var1 mySpouse = new2 FirstName = "Vickey"3, LastName = "Rattz"3 ;
```

В этом примере я объявляю переменную по имени `mySpouse`, используя ключевое слово `var`. Ей присваивается значение анонимного типа, инициализированного с использованием новых средств инициализации объектов. Эта единственная строка кода использует преимущество ключевого слова `var`, анонимных типов и инициализации объектов.

¹Вы можете обнаружить в этой строке кода использование ключевого слова `var`, потому что оно явно указано. ²Вы можете видеть, что здесь есть анонимный тип, потому что я использовал операцию `new` без указания именованного класса. ³И вы можете видеть явную инициализацию анонимного объекта с использованием нового средства инициализации объектов.

Короче говоря, ключевое слово `var` позволяет вывести тип объекта на основе типа данных, которыми он инициализирован. Анонимные типы позволяют на лету создавать новые типы данных — классы. В соответствии со словом `анонимные`, эти новые типы данных не имеют названия. Вы не сможете нормально создавать анонимный тип данных, если не знаете, какие переменные-члены он содержит, и не можете знать, какие члены он содержит, если вам неизвестны их типы. И, наконец, вы не можете знать типы этих членов до тех пор, пока они не инициализированы. Средство инициализации объектов позволяет все это решить.

На основании этой строки кода компилятор создаст новый анонимный класс, содержащий два общедоступных члена типа `string`; первый будет назван `FirstName`, а второй — `LastName`.

Ключевое слово var и явно типизированной локальной переменной

С добавлением анонимных типов в C# стала очевидной новая проблема. Если создается переменная безымянного типа, каковым является анонимный тип, кому типу переменной ее можно присваивать? Рассмотрим в качестве примера следующий код:

```
// Этот код не компилируется.
??? unnamedTypeVar = new {firstArg = 1, secondArg = "Joe" };
```

Какой тип переменной нужно объявлять для `unnamedTypeVar`? Это проблема. В Microsoft решили ее, создав ключевое слово `var`. Это новое ключевое слово информирует компилятор о том, что он должен явно вывести тип переменной из инициализатора переменной. Это значит, что переменная, объявленная с ключевым словом `var` должна иметь инициализатор.

Если вы пропустите инициализатор, то получите ошибку компилятора. В листинге 2.4 показан код, который объявляет переменную ключевым словом `var`, но не инициализирует ее.

Листинг 2.4. Неправильное объявление переменной с использованием ключевого слова var

```
var name;
```

И здесь мы получаем ошибку компиляции:

```
Implicitly-typed local variables must be initialized
Неявно типизированная локальная переменная должна быть инициализирована
```

Поскольку эти переменные подвергаются статической проверке типов во время компиляции, инициализатор необходим, чтобы компилятор мог явно вывести тип из него. Попытка присвоить значение другого типа где-нибудь в коде вызовет ошибку компиляции. Например, рассмотрим код в листинге 2.5.

Листинг 2.5. Неправильное присваивание переменной, объявленной с ключевым словом var

```
var name = "Joe";      // Пока все хорошо.
name = 1;              // М-да...
Console.WriteLine(name);
```

Этот код не будет компилироваться, потому что тип переменной `name` явно выводится как `string`; однако я пытаюсь присвоить ей целочисленное значение 1. Вот такую ошибку компиляции вызывает этот код:

```
Cannot implicitly convert type 'int' to 'string'
Невозможно явно преобразовать тип 'int' в 'string'
```

Как видите, компилятор обязывает переменную сохранять первоначальный тип. Если вернуться к исходному примеру кода с присваиванием типа посредством ключевого слова `var`, то мой код с дополнительной строкой для отображения переменной будет выглядеть так, как показано в листинге 2.6.

Листинг 2.6. Присваивание анонимного типа переменной, объявленной с ключевым словом var

```
var unnamedTypeVar = new {firstArg = 1, secondArg = "Joe" };
Console.WriteLine(unnamedTypeVar.firstArg + ". " + unnamedTypeVar.secondArg);
```

Вот результат работы этого кода:

1. Joe

Как видите, используя ключевое слово `var`, вы получаете статический контроль типа, плюс гибкость поддержки анонимных типов. Это станет очень важно, когда речь пойдет об операциях проекции типов в остальной части книги.

В примерах, приведенных до сих пор, использование ключевого слова `var` было обязательным, поскольку альтернативы не существовало. Если вы присваиваете объект анонимного типа переменной, у вас нет никакой другой возможности, кроме как присвоить ее переменной, объявленной с ключевым словом `var`. Однако можно использовать `var` всегда при объявлении переменной, если только она правильно инициализируется. Я рекомендую воздержаться от злоупотребления этим из соображений сопровождаемости. Понятно, что разработчики всегда должны знать тип данных, с которыми они работают, но если действительный тип данных известен вам сейчас, то будет ли это так, когда вы вернетесь к этому коду шестью месяцами спустя? Как насчет других разработчиков, которые займутся сопровождением, когда перестанете этим заниматься?

Совет. В целях сопровождаемости кода избегайте использования ключевого слова `var` только потому, что это удобно. Применяйте его только там, где это необходимо, например, присваивая переменной объект анонимного типа.

Выражения инициализации объектов и коллекций

Из-за потребности в динамических типах данных, которые позволяют создавать анонимные типы, возникает необходимость изменения способов инициализации объектов и коллекций. Поскольку выражения теперь представлены в виде лямбда-выражений или дерева выражений, инициализация объектов и коллекций была упрощена, допуская инициализацию “на лету”.

Инициализация объектов

Инициализация объектов позволяет специфицировать значения инициализации для общедоступных членов и свойств класса во время создания экземпляра. Для примера рассмотрим следующий код:

```
public class Address
{
    public string address;
    public string city;
    public string state;
    public string postalCode;
}
```

До появления средства инициализации объектов, добавленного в C# 3.0, без специализированного конструктора вы должны были инициализировать объект типа `Address`, как показано в листинге 2.7.

Листинг 2.7. Создание экземпляра и инициализация класс старым способом

```
Address address = new Address();
address.address = "105 Elm Street";
address.city = "Atlanta";
address.state = "GA";
address.postalCode = "30339";
```

Это было бы очень неудобно в лямбда-выражениях. Предположим, что вам нужно запросить значения из источника данных и проектировать определенные члены на объект `Address` операцией `Select`:

```
// Этот код не компилируется.
IEnumerable<Address> addresses = somedatasource
    .Where(a => a.State = "GA")
    .Select(a => new Address(???)?);
```

У вас просто не было бы удобного способа инициализировать члены вновь сконструированного объекта Address. Но не беспокойтесь! На помощь пришла инициализация объектов. Теперь вы можете сказать, что можно было бы просто создать конструктор, который принимал бы все эти инициализирующие значения при создании экземпляра объекта. Да, это так — иногда. Но что за морока! И как поступить, если анонимный тип создается на лету? Не проще ли создавать экземпляры объектов, как показано в листинге 2.8?

Листинг 2.8. Создание экземпляра и инициализация класс новым забавным способом

```
Address address = new Address {
    address = "105 Elm Street",
    city = "Atlanta",
    state = "GA",
    postalCode = "30339"
};
```

Вы можете уйти от проблемы с помощью лямбда-выражения. К тому же напомню, что эти новые возможности инициализации объектов могут применяться где угодно, а не только в запросах LINQ.

При использовании инициализации объекта компилятор создает экземпляр объекта, используя конструктор класса без параметров, затем инициализирует именованные члены указанными значениями. Любые члены, которые не специфицированы, получают свои значения по умолчанию для их типов данных.

Инициализация коллекций

Но так как недостаточно одной только новой инициализации объектов, кто-то в Microsoft должен был сказать: “А как насчет коллекций?”. *Инициализация коллекций* позволяет специфицировать инициализирующие значения для коллекции — как вы делаете это для объектов — до тех пор, пока коллекция реализует интерфейс System.Collections.Generic.ICollection<T>. Это значит, что ни одна из унаследованных старых коллекций C#, присутствующих в пространстве имен System.Collection, не могут быть инициализированы подобным образом.

Для примера инициализации коллекции рассмотрим код в листинге 2.9.

Листинг 2.9. Пример инициализации коллекции

```
using System.Collections.Generic;
List<string> presidents = new List<string> { "Adams", "Arthur", "Buchanan" };
foreach(string president in presidents)
{
    Console.WriteLine(president);
}
```

Запустив этот пример нажатием <Ctrl+F5>, вы получите следующий результат:

```
Adams
Arthur
Buchanan
```

В дополнение к использованию инициализации коллекций с LINQ может быть также очень удобно создавать инициализированные коллекции в коде, где не присутствуют запросы LINQ.

Анонимные типы

Создание нового уровня API в языке для обобщенных запросов данных было затруднено в C# нехваткой возможности создания новых типов данных во время компиляции. Если мы хотим, чтобы запросы данных извлекали первоклассные элементы на уровне языка, для этого язык должен обладать возможностью создания первоклассных элементов данных уровня языка, которыми в C# являются классы. Поэтому спецификация языка C# 3.0 теперь включает возможность динамического создания новых безымянных классов и объектов этих классов. Классы такого рода известны как *анонимные типы*.

Анонимный тип не имеет имени и генерируется компилятором на основе инициализации создаваемого экземпляра объекта. Поскольку класс не имеет имени типа, любые переменные, присваиваемые объекту анонимного типа, должны иметь какой-то способ объявления. Именно в этом предназначение нового ключевого слова `var` в C# 3.0.

Анонимный тип незаменим при проектировании (projecting) новых типов данных с использованием операций `Select` или `SelectMany`. Без анонимных типов при вызове этих операций должны были всегда существовать предопределенные именованные классы. Было бы очень неудобно создавать именованные классы для каждого запроса.

В разделе этой главы, посвященной инициализации объектов, мы обсуждали следующий код создания экземпляра и инициализации объекта:

```
Address address = new Address {
    address = "105 Elm Street",
    city = "Atlanta",
    state = "GA",
    postalCode = "30339"
};
```

Если бы вместо использования именованного класса `Address` я захотел применить анонимный тип, мне нужно было бы просто пропустить имя класса. Однако тогда вы не можете сохранить вновь созданный объект в переменной типа `Address`, потому что он уже не был бы переменной типа `Address`. Он относился бы теперь к сгенерированному типу, известному только компилятору. Поэтому пришлось бы изменить и тип переменной `address` также. Опять-таки, для этого пригодится ключевое слово `var`, как продемонстрировано в листинге 2.10.

Листинг 2.10. Создание и инициализация экземпляра анонимного типа с использованием инициализации объектов

```
var address = new {
    address = "105 Elm Street",
    city = "Atlanta",
    state = "GA",
    postalCode = "30339"
};
Console.WriteLine("address = {0} : city = {1} : state = {2} : zip = {3}",
    address.address, address.city, address.state, address.postalCode);
Console.WriteLine("{0}", address.GetType().ToString());
```

Я добавил последний вызов метода `Console.WriteLine`, чтобы вы увидели внутреннее, сгенерированное компилятором имя анонимного класса. Вот результат этой программы:

```
address = 105 Elm Street : city = Atlanta : state = GA : zip = 30339
<>f__AnonymousType5`4[System.String,System.String,System.String,System.String]
```

Этот тип анонимного класса определено выглядит сгенерированным компилятором. Конечно, имя класса, сгенерированное вашим компилятором, может отличаться.

Расширяющие методы

Расширяющий метод (extension method) — это статический метод статического класса, который вы можете вызывать, как если бы он был методом экземпляра другого класса. Например, вы можете создать расширяющий метод по имени `ToDouble`, который будет статическим методом созданного вами статического класса по имени `StringConversion`, но вызываемый так, как будто бы он был методом объекта типа `String`.

Прежде чем я подробно объясню расширяющие методы, давайте сначала рассмотрим проблему, которая привела к их созданию, обсудив разницу между статическими методами (класса) и методами уровня экземпляра (объекта). Методы уровня экземпляра могут вызываться только на экземплярах класса, иначе известных, как объекты. Вы не можете вызвать метод уровня экземпляра на самом классе. Аналогично статические методы должны вызываться на классе, а не на экземплярах класса.

Сравнение методов экземпляра (объекта) и статических методов (класса)

Метод `ToUpper` класса `string` является примером метода уровня экземпляра. Вы не можете вызвать `ToUpper` на самом классе `string`; вы должны вызывать его на объекте `string`.

В коде листинга 2.11 я демонстрирую это, вызывая метод `ToUpper` на объекте по имени `name`.

Листинг 2.11. Вызов метода экземпляра на объекте

```
// Этот код компилируется.  
string name = "Joe";  
Console.WriteLine(name.ToUpper());
```

Этот код успешно компилируется и при запуске выдает следующий вывод:

JOE

Однако если я попытаюсь вызвать метод `ToUpper` на самом классе `string`, то получу ошибку компиляции, потому что метод `ToUpper` — это метод уровня экземпляра, а я пытаюсь вызвать его на классе вместо объекта. В листинге 2.12 показан пример такой попытки и ошибка компилятора, вызванная ею.

Листинг 2.12. Попытка вызвать метод экземпляра на классе

```
// Этот код не компилируется.  
string.ToUpper();
```

Попытка скомпилировать этот код приведет к следующей ошибке компилятора:

An object reference is required for the nonstatic field, method, or property
'string.ToUpper()'

Объектная ссылка требуется для нестатического поля, метода или свойства
'string.ToUpper()'

Этот пример кажется немного искусственным, поскольку он никак не может работать, так как я не даю здесь никакого строчного значения, которое нужно было бы конвертировать в верхний регистр. Любая попытка сделать это эквивалентна попытке вызвать вариант метода `ToUpper`, который не существует, потому что нет прототипа для `ToUpper`, чья сигнатура включает `string`.

Контрастом методу `ToUpper` может быть метод `Format` класса `string`. Этот метод определен как `static`. Это требует, чтобы метод `Format` вызывался на самом классе `string`, а не на объекте типа `string`. Сначала я попробую вызвать его на объекте в коде, приведенном в листинге 2.13.

Листинг 2.13. Попытка вызвать метод класса на объекте

```
string firstName = "Joe";
string lastName = "Rattz";
string name = firstName.Format("{0} {1}", firstName, lastName);
Console.WriteLine(name);
```

Этот код породит следующую ошибку компилятора:

`Member 'string.Format(string, object, object)' cannot be accessed with an instance reference; qualify it with a type name instead`

Член `'string.Format(string, object, object)'` не может быть доступным через ссылку экземпляра; квалифицируйте его с помощью имени типа

Однако если вместо этого я вызову метод `Format` на самом классе `string`, он скомпилируется и будет работать, как ожидалось, что и демонстрируется в листинге 2.14.

Листинг 2.14. Вызов метода `Format` на классе `string`

```
string firstName = "Joe";
string lastName = "Rattz";
string name = string.Format("{0} {1}", firstName, lastName);
Console.WriteLine(name);
```

Этот код выдаст следующий результат:

Joe Rattz

Иногда из частей сигнатуры помимо ключевого слова `static` очевидно, что метод должен быть методом уровня экземпляра. Например, рассмотрим метод `ToUpperc`. Он не принимает никаких аргументов, за исключением одной перегруженной версии, принимающей аргумент типа ссылки на объект `CultureInfo`. Поэтому если он не полагается на внутренние данные экземпляра `string`, какую строку он будет преобразовывать в верхний регистр?

Решение проблемы расширяющими методами

Так в чем же проблема — спросите вы. Для дискуссии представим, что вы — разработчик, отвечающий за проектирование нового способа запроса множества объектов. Скажем, вы решили создать метод `Where`, чтобы помочь в этом, в классе `where`. Как вы должны это сделать?

Нужно ли сделать операцию `Where` методом экземпляра? Если да, то в какой класс вы должны добавить метод `Where`? Вы хотите, чтобы он работал при опросе любой коллекции объектов. Нет подходящего класса, куда логично было бы добавить метод `Where`. Приняв такой подход, вы должны модифицировать массу разных классов, если хотите получить универсальное средство опроса данных.

Таким образом, вы поняли, что метод должен быть статическим — так в чем же проблема? Подумайте о вашем типичном запросе (SQL), и о том, как много конструкций `where` там часто бывает. И учтите соединения (`joins`), группировку и упорядочивание.

Предположим, что вы создали концепцию нового типа данных, последовательность обобщенных объектов данных, который мы назовем `Enumerable`. Имеет смысл, что метод `Where` понадобится для того, чтобы оперировать `Enumerable` (данных) и возвращать

другой отфильтрованный `Enumerable`. В добавок метод `Where` должен будет принимать аргумент, который позволит разработчику специфицировать точную логику, используемую для фильтрации записей данных из `Enumerable` и в него. Этот аргумент, который я назову *предикатом*, должен быть специфицирован как именованный метод, анонимный метод или лямбда-выражение.

Внимание! Следующие три примера кода в этом разделе являются гипотетическими и компилироваться не будут.

Поскольку метод `Where` требует входного `Enumerable` для фильтрации, и метод объявлен как `static`, этот входной `Enumerable` должен быть специфицирован как аргумент метода `Where`. Выглядеть он должен примерно так:

```
static Enumerable Enumerable.Where(Enumerable input, LambdaExpression predicate) {
    ...
}
```

Если пока проигнорировать семантику лямбда-выражения, то вызов метода `Where` должен выглядеть подобно показанному ниже:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable filteredEnumerable = Enumerable.Where(enumerable, lambdaExpression);
```

Выглядит не особенно страшно. Но что случится, если нам понадобится несколько конструкций `where`? Поскольку `Enumerable`, которым оперирует метод `Where`, должен быть аргументом метода, то в результате имеем необходимость выстраивания цепочки методов посредством включения их друг в друга. Всего три конструкции `where` изменят наш код следующим образом:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable =
    Enumerable.Where(Enumerable.Where(Enumerable.Where(enumerable, lX1), lX2), lX3);
```

И вы должны читать этот оператор, начиная изнутри и наружу. Такое нелегко прочитать быстро. Можете ли вы представить, как будет выглядеть действительно сложный запрос? Должен существовать какой-то другой, лучший путь.

Решение

Замечательным решением была бы возможность вызывать статический метод `Where` на каждом объекте `Enumerable`, а не на классе. Тогда не пришлось бы передавать каждый `Enumerable` в метод `Where`, потому что объект `Enumerable` имел бы доступ к своему собственному внутреннему `Enumerable`. Это изменило бы синтаксис запроса, представленного выше, на нечто вроде такого:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable = enumerable.Where(lX1).Where(lX2).Where(lX3);
```

Внимание! Предыдущий и следующий код являются гипотетическими и компилироваться не будут.

Его можно будет даже переписать следующим образом:

```
Enumerable enumerable = {"one", "two", "three"};
Enumerable finalEnumerable = enumerable
    .Where(lX1)
    .Where(lX2)
    .Where(lX3);
```

Это намного легче читать. Вы можете прочесть оператор слева направо, сверху вниз. Как видим, такой синтаксис очень легко понимать, как только вы разберетесь, что он

делает. Именно поэтому вы часто встретите запросы LINQ, написанные в этом формате, в документации по LINQ и в настоящей книге.

В конечном итоге все, что вам нужно — это возможность иметь статический метод, который можно вызывать на экземпляре класса. Именно для этого предназначены расширяющие методы, и именно это они позволяют делать. Эти методы были добавлены в C#, чтобы предоставить синтаксически элегантный способ вызова статических методов, не передавая им первый аргумент. Это позволяет вызывать расширяющие методы так, как будто они являются методами первого аргумента, что делает гораздо более читабельными цепочки вызовов расширяющих методов, чем в случае обычной передачи им первого аргумента. Расширяющие методы помогают LINQ, позволяя вызывать стандартные операции запросов на интерфейсе `IEnumerable<T>`.

На заметку! Расширяющие методы — это такие методы, которые, будучи статическими, могут быть вызваны на экземпляре (объекте) класса, а не на самом классе.

Объявление и вызов расширяющих методов

Спецификация первого аргумента метода модификатором — ключевым словом `this` делает такой метод расширяющим.

Расширяющий метод выглядит как метод экземпляра любого объекта с тем же типом, что и у его первого аргумента. Например, если первый аргумент расширяющего метода имеет тип `string`, то расширяющий метод будет выглядеть как метод экземпляра `string`, и может быть вызван на любом экземпляре `string`.

Также имейте в виду, что расширяющие методы могут быть объявлены только в статических классах.

Ниже приведен пример расширяющего метода.

```
namespace Netsplore.Utilities
{
    public static class StringConversions
    {
        public static double ToDouble(this string s)
        {
            return Double.Parse(s);
        }
        public static bool ToBool(this string s)
        {
            return Boolean.Parse(s);
        }
    }
}
```

Обратите внимание, что и класс, и каждый его метод объявлены как `static`. Теперь вы можете воспользоваться преимуществом этих расширяющих методов, вызывая статические методы на экземплярах объектов, как показано в листинге 2.15. Поскольку метод `ToDouble` статический, и его первый аргумент специфицирован с ключевым словом `this`, метод `ToDouble` является расширяющим.

Листинг 2.15. Вызов расширяющего метода

```
using Netsplore.Utilities;
double pi = "3.1415926535".ToDouble();
Console.WriteLine(pi);
```

Этот код дает следующий вывод:

3.1415926535

Важно, чтобы вы специфицировали директиву `using` для пространства имен `Netsplore.Utilities`, иначе компилятор не найдет расширяющие методы и вы получите ошибку компиляции, приведенную ниже:

```
'string' does not contain a definition for 'ToDouble' and no extension method  
'.ToDouble' accepting a first argument of type 'string' could be found (are you  
missing a using directive or an assembly reference?)
```

'string' не содержит определения для '.ToDouble' и не найдено ни одного
расширяющего метода

'ToString', принимаемого первый аргумент типа 'string' (возможно, вы пропустили
директиву `using` или ссылку на сборку?)

Как уже упоминалось ранее, попытка объявить расширяющий метод внутри не-
статического класса не допускается. Если вы сделаете это, то увидите такую ошибку
компилятора:

Extension methods must be defined in a non-generic static class

Расширяющие методы должны быть определены в необобщенном статическом классе

Преимущество расширяющих методов

Обычные методы экземпляров объектов имеют преимущество перед расширяющими
методами, когда их сигнатура соответствует сигнатуре вызова.

Расширяющие методы выглядят действительно полезной концепцией, особенно ко-
гда вы хотите иметь возможность расширить класс, который нельзя расширить, такой
как герметизированный (`sealed`) класс, либо один из классов, исходного кода которого у
вас нет. Все предыдущие примеры расширяющих методов эффективно добавляли мето-
ды в класс `string`. Без расширяющих методов вы не смогли бы этого сделать, поскольку
класс `string` герметизирован.

Частичные методы

Недавно добавленные в C# 3.0 частичные методы (`partial methods`) добавляют в C#
легковесный механизм обработки событий. Выбросьте из головы ваши предположения
о частичных методах, которые могли у вас появиться, исходя из их названия. У частич-
ных методов есть только одно общее с частичными классами — это то, что они могут
существовать только в частичном классе. Фактически, это правило номер 1 для час-
тичных методов.

Прежде чем я перейду к описанию всех правил, которым подчиняются частичные
методы, я объясню, что они собой представляют. Частичные методы — это методы, где
прототип или определение метода специфицировано в объявлении частичного клас-
са, а реализация метода не представлена в том же объявлении частичного класса.
Фактически, они могут не иметь никакой реализации ни в одном из объявлений одного
и того же частичного класса. И если реализации метода нет ни в одном из объявлений
частей этого класса, то компилятором не генерируется никакого IL-кода ни для объяв-
ления метода, ни для его вызова, ни для оценки аргументов, переданных методу. То есть
все обстоит так, как будто этот метод вообще никогда не существовал.

Некоторым не нравится термин “частичный метод”, потому что это нечто несоответ-
ствующее их поведению, если сравнить их с частичными классами. Возможно, модифи-
катор метода должен был называться `ghost` (привидение) вместо `partial`.

Пример частичного метода

Взглянем на частичный класс, содержащий определение частичного метода в файле
по имени `MyWidget.cs`:

Файл класса MyWidget

```
public partial class MyWidget
{
    partial void MyWidgetStart(int count);
    partial void MyWidgetEnd(int count);
    public MyWidget()
    {
        int count = 0;
        MyWidgetStart(++count);
        Console.WriteLine("In the constructor of MyWidget.");
        MyWidgetEnd(++count);
        Console.WriteLine("count = " + count);
    }
}
```

В приведенном объявлении класса MyWidget содержится частичный класс по имени MyWidget. Первые две строки кода являются определениями частичных методов. Я определил частичные методы по имени MyWidgetStart и MyWidgetEnd, каждый из которых принимает входной параметр типа int и возвращает void. Это еще одно правило: частичные методы должны возвращать void.

Следующая часть кода в классе MyWidget — это конструктор. Как видите, я объявил переменную типа int по имени count и инициализировал ее нулем. Затем я вызываю метод MyWidgetStart, вывожу сообщение на консоль, вызываю метод MyWidgetEnd и, наконец, вывожу значение переменной count на консоль. Обратите внимание, что я увеличиваю значение count всякий раз, передавая ее частичному методу. Я делаю это для того, чтобы доказать, что если никакой реализации частичного метода нет, его аргументы не обрабатываются.

В листинге 2.16 создается экземпляр объекта MyWidget.

Листинг 2.16. Создание экземпляра MyWidget

```
MyWidget myWidget = new MyWidget();
```

Взглянем на вывод этого примера, нажав <Ctrl+F5>:

```
In the constructor of MyWidget.
count = 0
```

Как видите, даже после того, как конструктор MyWidget дважды увеличил значение его переменной count, при отображении ее значения в конце конструктора оно все еще равно 0. Это объясняется тем, что код вычисления аргументов нереализованных частичных методов не генерируется компилятором. Никакого кода IL не было создано ни для одного из вызовов этих двух частичных методов.

Теперь добавим реализацию двух частичных методов:

Другое объявление MyWidget, содержащее реализации частичных методов

```
public partial class MyWidget
{
    partial void MyWidgetStart(int count)
    {
        Console.WriteLine("In MyWidgetStart(count is {0})", count);
    }
    partial void MyWidgetEnd(int count)
    {
        Console.WriteLine("In MyWidgetEnd(count is {0})", count);
    }
}
```

Добавив это объявление, снова запустите код из листинга 2.16, и вы увидите такой результат:

```
In MyWidgetStart(count is 1)
In the constructor of MyWidget.
In MyWidgetEnd(count is 2)
count = 2
```

Как видим, теперь не только вызваны реализации частичных методов, но и переданные аргументы также были вычислены. Это доказывает значение count в конце вывода.

В чем смысл частичных методов?

Итак, вы можете спросить: в чем смысл всего этого? Другие скажут: "Это похоже на использование наследования и виртуальных методов. Зачем засорять язык чем-то похожим?". Я отвечаю, что частичные методы более эффективны, если вы планируете разрешить многие потенциально нереализуемые трюки в коде. Они позволяют писать код с учетом того, что кто-то другой расширит его через парадигму частичного класса, при этом не нарушая производительности первоначальной реализации, если этого не случится.

Возможно, причиной появления частичных методов послужило то, что они позволяют генерировать код инструментами для сущностных классов LINQ to SQL. Чтобы сделать сгенерированные сущностные классы более полезными, к ним были добавлены частичные методы. Например, каждое отображенное свойство сгенерированного сущностного класса имеет частичный метод, который вызывается перед изменением свойства, а другой частичный метод вызывается после изменения свойства. Это позволяет вам добавить другой модуль, объявляющий тот же сущностный класс, реализовать эти частичные методы и получать извещения всякий раз, когда свойство собирается измениться, а также сразу после того, как оно изменится. Насколько это круто? А если вы не сделаете этого, код не станет больше или медленнее. Кто против этого?

Правила

Пока все выглядело достаточно забавно, но, к сожалению, существуют некоторые правила, которым подчиняются частичные методы. Ниже приведен их список.

- Частичные методы должны быть определены только в частичных классах.
- Частичные методы должны специфицировать модификатор `partial`.
- Частичные методы являются частными, но не должны специфицировать модификатор `private`, иначе компилятор выдаст ошибку.
- Частичные методы должны возвращать `void`.
- Частичные методы могут быть нереализованными.
- Частичные методы могут не иметь аргументов.

Все эти правила не так плохи. Учитывая выигрыш в гибкости генерируемых сущностных классов, плюс то, что мы можем делать с ними самостоятельно, я думаю, С# получил отличное новое средство.

Выражения запросов

Одним из удобств, предоставленных языком C#, является оператор `foreach`. Когда вы используете `foreach`, компилятор транслирует его в цикл с вызовами таких методов, как `GetEnumerator` и `MoveNext`. Простота оператора `foreach`, используемого для перечисления элементов массивов и коллекций, обеспечила ему широкую популярность и частое применение.

Одним из привлекательных для разработчиков средств LINQ является SQL-подобный синтаксис, доступный в LINQ-запросах. Первые несколько примеров LINQ в первой главе нашей книги использовали этот синтаксис. Синтаксис представлен посредством нового расширения языка C# 3.0, называемого выражениями запросов. Выражения запросов позволяют запросам LINQ принимать форму, подобную SQL, лишь с несколькими небольшими отличиями.

Чтобы выполнить запрос LINQ, не обязательно использовать выражения запросов. Альтернативой является применение стандартной точечной нотации C# с вызовом методов на объектах и классах. Во многих случаях я нахожу применение стандартной точечной нотации более предпочтительным, поскольку мне кажется, что она более наглядно демонстрирует, что происходит в действительности и когда. При записи запроса в стандартной точечной нотации не происходит никакой трансформации при компиляции. Именно поэтому многие примеры в этой книге не используют синтаксис выражений запросов, предпочитая стандартный синтаксис точечной нотации. Однако привлекательность синтаксиса выражений запросов бесспорна. Впечатление чего-то знакомого, которое он производит, когда вы формулируете свой первый запрос, может быть весьма привлекательно.

Чтобы получить представление о различиях между этими двумя синтаксисами, в листинге 2.17 показан один запрос в стандартном точечном синтаксисе.

Листинг 2.17. Запрос, использующий стандартный синтаксис точечной нотации

```
string[] names = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = names
    .Where(n => n.Length < 6)
    .Select(n => n);
foreach (string name in sequence)
{
    Console.WriteLine("{0}", name);
}
```

В листинге 2.18 представлен эквивалентный запрос, использующий синтаксис выражений запросов.

Листинг 2.18. Эквивалентный запрос, использующий синтаксис выражений запросов

```
string[] names = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = from n in names
    where n.Length < 6
    select n;
foreach (string name in sequence)
{
    Console.WriteLine("{0}", name);
}
```

Первое, что вы можете заметить в примере с выражением запроса — это то, что в отличие от SQL, оператор `from` предшествует `select`. Одной из причин этого была необходимость сужения контекста для IntelliSense. Без такой инверсии операторов, если в текстовом редакторе Visual Studio 2008 вы набираете "select", за которым следует пробел, то IntelliSense не будет иметь представления о том, какие переменные нужно отображать в его выпадающем списке. Контекст допустимых переменных в этой точке ничем не ограничен. Если же сначала специфицировать, откуда поступают данные, то IntelliSense получает контекст и может представить вам список переменных для выбора. Оба эти примера дают один и тот же результат:

```
Adams
Bush
Ford
Grant
Hayes
Nixon
Polk
Taft
Tyler
```

Важно отметить, что синтаксис выражений запросов транслирует только наиболее распространенные операции запросов: `Where`, `Select`, `SelectMany`, `Join`, `GroupJoin`, `GroupBy`, `OrderBy`, `ThenBy`, `OrderByDescending` и `ThenByDescending`.

Грамматика выражений запросов

Ваша выражения запросов должны подчиняться следующим правилам.

1. Выражение должно начинаться с конструкции `from`.
2. Остальная часть выражения может содержать ноль или более конструкций `from`, `let` или `where`. Конструкция `from` — это генератор, который объявляет одну или более переменных-перечислителей последовательности или соединения (`join`) нескольких последовательностей. Конструкция `let` представляет переменную и присваивает ей значение. Конструкция `where` фильтрует элементы из последовательности или соединения нескольких последовательностей.
3. Остальная часть выражения запроса может затем включать конструкцию `orderby`, содержащую одно или более полей сортировки с необязательным направлением упорядочивания. Направление может быть `ascending` или `descending`.
4. Затем в оставшейся части выражения может идти конструкция `select` или `group`.
5. Наконец, в оставшейся части выражения может следовать необязательная конструкция продолжения. Такой конструкцией может быть либо `into`, ноль или более конструкций `join`, или же другая повторяющаяся последовательность перечисленных элементов, начиная с пункта 2. Конструкция `into` направляет результаты запроса в воображаемую выходную последовательность, которая служит конструкцией `from` для последующих выражения запросов, начиная с конструкций из пункта 2.

За более строгим и лаконичным техническим описанием синтаксиса выражений запросов обращайтесь к следующей грамматической диаграмме, представленной Microsoft в документации MSDN LINQ.

выражение-запроса:
конструкция-`from` тело-запроса

52 Часть I. LINQ: язык интегрированных запросов в C# 2008

конструкция-from:

 from тип_{необязательно} идентификатор in выражение конструкции-join_{необязательно}

конструкции-join:

 конструкция-join

 конструкции-join конструкция-join

конструкция-join:

 join тип_{необязательно} идентификатор in выражение on идентификатор equals
 идентификатор

 join тип_{необязательно} идентификатор in выражение on идентификатор equals
 идентификатор into идентификатор

тело-запроса:

 конструкции-from-let-where_{необязательно} конструкция-orderby_{необязательно}

конструкция-select-или-group

 продолжение-запроса_{необязательно}

конструкции-from-let-where:

 конструкция-from-let-where

 конструкции-from-let-where конструкция-from-let-where

конструкция-from-let-where:

 конструкция-from

 конструкция-let

 конструкция-where

конструкция-let:

 let идентификатор = выражение

конструкция-where:

 where boolean-выражение

конструкция orderby:

 orderby упорядочения

упорядочения:

 упорядочение

 упорядочения , упорядочение

упорядочение:

 выражение направление-упорядочения_{необязательно}

направление-упорядочения:

 ascending

 descending

конструкция-select-or-group:

 конструкция-select

 конструкция-group

конструкция-select:

 select выражение

конструкция-group:

 group выражение by выражение

продолжение-запроса:

 into идентификатор конструкции-join_{необязательно} тело-запроса

Трансляция выражений запросов

Теперь предположим, что вы создали синтаксически корректное выражение запроса. Следующая сложность состоит в том, как компилятор транслирует это выражение запроса в код C#. Он должен транслировать ваше выражение в стандартную точечную нотацию C#, о которой говорилось в разделе, посвященном выражениям запросов. Но как это делается?

Чтобы транслировать выражение запроса, компилятор ищет в нем шаблоны кода (code patterns). Компилятор выполняет несколько шагов трансляции в определенном порядке, чтобы превратить выражение запроса в стандартную нотацию C#. Каждый из этих шагов ищет один или более взаимосвязанных шаблонов кода. Компилятор должен многократно транслировать все вхождения шаблонов кода для данного шага трансля-

ции, прежде чем перейти к следующему шагу. И на каждом шаге он исходит из предположения, что все шаблоны кода для всех предыдущих шагов уже транслированы.

Прозрачные идентификаторы

Некоторые трансляции вставляют переменные перечислений с прозрачными идентификаторами. На шаге трансляции, описанном в следующем разделе, прозрачный идентификатор обозначен звездочкой (*). Его не нужно путать с символом * — шаблоном полей выбора SQL. При трансляции выражения запроса иногда компилятором генерируются дополнительные перечисления, и прозрачные идентификаторы используются для прохода по ним. Прозрачные идентификаторы существуют только во время процесса трансляции, и как только выражение запроса полностью транслировано, никаких прозрачных идентификаторов в запросе не остается.

Шаги трансляции

Теперь я расскажу о шагах трансляции. Делая это, я использую буквы переменных, показанных в табл. 2.1, чтобы представить определенные части запроса.

Таблица 2.1. Переменные шагов трансляции

Переменная	Описание	Пример
c	Сгенерированная компилятором временная переменная	—
e	Переменная перечислителя	from e in customers
f	Выбранный элемент-поле или новый анонимный тип	from e in customers select f
g	Групповой элемент	from e in s group g by k
i	Воображаемая последовательность into	from e in s into i
k	Групповой или объединенный ключевой элемент	from e in s group g by k
l	Переменная, представленная let	from e in s let l = v
o	Упорядочивающий элемент	from e in s orderby o
s	Входная последовательность	from e in s
v	Значение, присвоенное переменной let	from e in s let l = v
w	Конструкция where	from e in s where w

Здесь необходимо высказать одно предупреждение. Описание шагов трансляции может показаться довольно сложным. Пусть это вас не расстраивает. Вам не нужно полностью понимать все шаги трансляции, чтобы писать запросы LINQ — по крайней мере, не больше того, что вам нужно знать о трансляции компилятором оператора `foreach`, чтобы использовать его. Я привожу их здесь для того, чтобы предоставить дополнительную информацию о трансляции на тот случай, если вдруг она вам понадобится, что случается редко, а может быть и никогда.

Шаги трансляции документированы как “code pattern \Leftrightarrow translation” (шаблон кода \Leftrightarrow трансляция). Как ни странно, несмотря на то, что я представляю шаги трансляции в порядке их выполнения компилятором, полагаю, что понять этот процесс проще, если изучать его в обратном порядке. Причина в том, что когда вы посмотрите на первый шаг трансляции — он выполняет только трансляцию первого шаблона кода, и у вас еще остается масса не транслированных шаблонов кода, которые нужно разобрать. По моему мнению, это оставляет место для кривотолков. Поскольку каждый шаг трансляции требует, чтобы шаблон кода предыдущего шага уже был транслирован, на момент полу-

54 Часть I. LINQ: язык интегрированных запросов в C# 2008

чения финального шага трансляции не остается никаких неясностей. И потому я считаю, что понять последний шаг легче, чем первый. К тому же, по моему мнению, проход по шагам трансляции в обратном порядке — наилучший способ понять весь процесс.

Итак, вот шаги трансляции, представленные в порядке, в котором компилятор выполняет их.

Конструкции `select` и `group` с конструкцией продолжения `into`

Если ваше выражение запроса содержит конструкцию продолжения `into`, выполняется следующая трансляция:

```
from ...1 into ...2
```

```
from i in  
►from ...1  
...2
```

Вот пример:

```
from c in customers  
group c by c.Country into g  
select new  
{ Country = g.Key,  
CustCount = g.Count() }
```

```
from g in  
from c in customers  
group c by c.Country  
►select new  
{ Country = g.Key,  
CustCount = g.Count() }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers.GroupBy(c => c.Country)  
.Select(g => new {Country = g.Key, CustCount = g.Count() })
```

Явные типы переменных перечислений

Если ваше выражение запроса содержит конструкцию `from`, которая явно специфицирует тип переменной перечисления, выполнится следующая трансляция:

```
from T e in s
```

```
►from e in s.Cast<t>()
```

Вот пример:

```
from Customer c in customers  
select c
```

```
►from c in customers.Cast<Customer>()
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers.Cast<Customer>()
```

Если ваше выражение запроса содержит конструкцию `join`, которая явно специфицирует тип переменной перечисления, то будет выполнена следующая трансляция:

```
from T e in s  
on k1 equals k2
```

```
►join e in s.Cast<T>()  
on k1 equals k2
```

Вот пример:

```
from c in customers  
join Order o in orders  
on c.CustomerID equals o.CustomersID  
select new { c.Name,  
o.OrderDate,  
o.Total }
```

```
from c in customers  
join o in orders.Cast<Order>()  
►on c.CustomerID equals o.CustomersID  
select new  
{ c.Name, o.OrderDate, o.Total }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers  
.Join(orders.Cast<Order>(),  
c => c.CustomerID,  
o => o.CustomerID,  
(c, o) => new { c.Name, o.OrderDate, o.Total })
```

Совет. Явно типизированные переменные перечислений необходимы, когда коллекция данных является одной из унаследованных коллекций данных, такой как `ArrayList`. Приведение, выполняемое при явной типизации переменной перечисления, преобразует унаследованную коллекцию в последовательность, реализующую `IEnumerable<T>` — так, что могут выполняться другие операции запросов.

Конструкции `join`

Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join` без продолжения `into` вслед за `select`, то имеет место следующая трансляция (`t` — временная переменная, сгенерированная компилятором):

```
from e1 in s1
join e2 in s2
on k1 equals k2
select f
          from t in s1
          .Join(s2,
    ➤      e1 => k1,
            e2 => k2,
            (e1, e2) => f)
select t
```

Вот пример:

```
from c in customers
join o in orders
on c.CustomerID equals o.CustomerID
select new { c.Name,
            o.OrderDate,
            o.Total }
          from t in customers
          .Join(orders,
    ➤      c => c.CustomerID,
            o => o.CustomerID,
            (c, o) => new
            { c.Name,
              o.OrderDate,
              o.Total })
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
.Join(orders,
    c => c.CustomerID,
    o => o.CustomerID,
    (c, o) => new { c.Name, o.OrderDate, o.Total })
```

Если выражение запроса содержит конструкцию `from`, за которой следует `join` с продолжением `into` вслед за `select`, имеет место следующая трансляция (`t` — временная переменная, сгенерированная компилятором):

```
from e1 in s1
join e2 in s2
on k1 equals k2
into i
select f
          from t in s1
          .GroupJoin(s2,
    ➤      e1 => k1,
            e2 => k2,
            (e1, e2) => f)
          select t
```

Вот пример:

```
from c in customers
join o in orders
on c.CustomerID equals o.CustomerID
into co
select new
{ c.Name, Sum = co.Sum(o => o.Total) }
          from t in customers
          .GroupJoin(orders,
    ➤      c => c.CustomerID,
            o => o.CustomerID,
            (c, o) => new
            { c.Name,
              Sum = co.Sum(
                o => co.Total)
            Select t }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
.GroupJoin(orders,
    c => c.CustomerID,
    o => o.CustomerID,
    (c, o) => new { c.Name, Sum = o.Sum(o = o.Total) })
```

Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join` без продолжения `into` вслед за чем-то, отличным от `select`, то имеет место следующая трансляция (* — прозрачный идентификатор):

<pre>from e₁ in s₁ join e₂ in s₂ on k₁ equals k₂ ... </pre>	<pre>from * in from e₁ in s₁ ➤join e₂ in s₂ on k₁ equals k₂ select new { e₁, e₂ }</pre>
---	---

Обратите внимание, что у вас есть шаблон кода, соответствующий первому шаблону кода на этом шаге трансляции. Точнее, у вас есть выражение запроса, содержащее конструкцию `from`, за которой следует конструкция `join`, без продолжения `into`, за которым следует `select`. Поэтому компилятор повторит этот шаг трансляции.

Если выражение запроса содержит конструкцию `from`, за которой следует конструкция `join`, с продолжением `into`, за которым следует нечто отличное от `select`, то имеет место следующая трансляция (* — прозрачный идентификатор):

<pre>from e₁ in s₁ join e₂ in s₂ on k₁ equals k₂ into i ... </pre>	<pre>From * in from e₁ in s₁ ➤join e₂ in s₂ on k₁ equals k₂ into i select new { e₁, i }</pre>
--	--

На этот раз обратите внимание на присутствие шаблона кода, соответствующего второму шаблону кода на этом шаге трансляции. Точнее, есть выражение запроса, содержащее конструкцию `from`, за которой следует конструкция `join`, с продолжением `into`, за которым идет `select`. Поэтому компилятор повторит этот шаг трансляции.

Конструкции `let` и `where`

Если выражение запроса содержит конструкцию `from`, за которой немедленно следует `let`, то имеет место следующая трансляция (* — прозрачный идентификатор):

<pre>from e in s let l = v</pre>	<pre>from * in ➤from e₁ in s₁ select new { e, l = v }</pre>
----------------------------------	---

Приведем пример (`t` — сгенерированный компилятором идентификатор, который невидим и недоступен никакому написанному вами коду):

<pre>from c on customers let cityStateZip = c.City + ", " + c.State + " " + c.Zip select new { c.Name, cityStateZip }</pre>	<pre>from * in from c in customers select new { c, cityStateZip = c.City + ", " + c.State + " " + c.Zip }</pre>
---	---

В результате последующих шагов трансляции, это в конечном итоге транслируется^в

```
customers
.Select(c => new { c, cityStateZip = c.City + ", " + c.State + " " + c.Zip })
.Select(t => new { t.c.Name, t.cityStateZip })
```

Если выражение запроса содержит конструкцию `from`, за которой немедленно следует конструкция `where`, имеет место следующая трансляция:

```
from e in s
where w
      from e in s
      ►.Where(e => w)
```

Вот пример:

```
from c in customers
where c.Country == "USA"
select new { c.Name, c.Country }
      from c in customers
      ►.Where(c => c.Country == "USA")
      select new { c.Name, c.Country }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
.Where(c => c.Country == "USA")
.Select(c => new { c.Name, c.Country })
```

Множественные генерирующие конструкции `from`

Если выражение запроса содержит две конструкции `from`, за которыми следуют `select`, происходит следующая трансляция:

```
from e1 in s1
from e2 in s2
select f
      from c in s1
      .SelectMany(e1 => from e2 in s2
      ►      select f)
      select c
```

Вот пример (`t` — временная переменная, сгенерированная компилятором):

```
from c in customers
from o in c.Orders
select new
{ c.Name, o.OrderID, o.OrderDate }
      from t in customers
      .SelectMany(c => from o in c.Orders
      ►      select new {
          c.Name,
          o.OrderID,
          o.OrderDate
        })
      Select t
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
.SelectMany(c => c.Orders.Select(o => new { c.Name, o.OrderID, o.OrderDate }))
```

Если выражение запроса содержит две конструкции `from`, за которыми следует что-то отличное от `select`, то происходит следующая трансляция (* — прозрачный идентификатор):

```
from e1 in s1
from e2 in s2
...
      from *
      from e1 in s1
      ►from e2 in s2
      select new { e1, e2 }
      ...
```

Вот пример (* — прозрачный идентификатор):

```
from c in customers
from o in c.Orders
orderby o.OrderDate descending
select new
{ c.Name, o.OrderID, o.OrderDate }
      from *
      from c in customers
      from o in c.Orders
      ►select new { c, o }
      orderby o.OrderDate descending
      select new
{ c.New, o.OrderID, o.OrderDate }
```

В дополнение к последующим шагам трансляции предыдущий транслированный код должен вызывать этот шаг трансляции снова, потому что после предыдущего первого шага вы имеете конструкцию `from`, за которой следует `from`, а за ней — конструкция

`select`, что является первым шаблоном кода, который данный шаг ищет для трансляции. Это пример шагов трансляции, которые иногда должны вызываться несколько раз, чтобы полностью транслировать все шаблоны кода, которые ищет любой конкретный шаг.

```
customers
    .SelectMany(c => c.Orders.Select(o => new { c, o }))
    .OrderByDescending(t => t.o.OrderDate)
    .Select(t => new { t.c.Name, t.o.OrderID, t.o.OrderDate })
```

Конструкции orderby

Если направление упорядочивания — `ascending`, происходит следующая трансляция:

```
from e in s
orderby o1, o2
    from e in s
    ►.OrderBy(e => o2).ThenBy(e => o2)
```

Вот пример:

```
from c in customers
orderby c.Country, c.Name
select new { c.Country, c.Name }
```

```
from c in customers
►.OrderBy(e => c.Country)
    .ThenBy(c.Name)
    select new { c.Country, c.Name }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
    .OrderBy(c => c.Country)
    .ThenBy(c.Name)
    .Select(c => new { c.Country, c.Name })
```

Если направление любого упорядочивания — `descending`, происходит трансляция в операции `OrderByDescending` или `ThenByDescending`. Следующий пример — такой же, как и предыдущий, за исключением того, что имена запрашиваются в убывающем порядке:

```
from c in customers
orderby c.Country, c.Name descending
select new { c.Country, c.Name }
```

```
from c in customers
►.OrderBy(e => c.Country)
    .ThenByDescending(c.Name)
    select new { c.Country, c.Name }
```

В результате последующих шагов трансляции, это в конечном итоге транслируется в

```
customers
    .OrderBy(c => c.Country)
    .ThenByDescending(c.Name)
    .Select(c => new { c.Country, c.Name })
```

Конструкции select

В выражении запроса, если выбранный элемент — тот же идентификатор, что и переменная-перечислитель последовательности, в том смысле, что вы выбираете целый элемент, хранящийся в последовательности, — происходит следующая трансляция:

```
from e in s
select f
    ► s
```

Вот пример:

```
from c in customers
select c
    ► customers
```

Если выбираемый элемент — не тот же идентификатор, что и переменная-перечислитель последовательности, в том смысле, что вы выбираете что-то отличное от всего элемента, хранящегося в последовательности, такое как член элемента или анонимный тип, сконструированный из нескольких членов элемента, — происходит следующая трансляция:

```
from e in s
select c
```

➤ s.Select(e => f)

Вот пример:

```
from c in customers
select c.Name
```

➤ customers.Select(c => c.Name)

Конструкции group

В выражении запроса, если групповой элемент — это тот же идентификатор, что и перечислитель последовательности, в том смысле, что вы группируете весь элемент, хранящийся в последовательности, — происходит такая трансляция:

```
from e in s
group g by k
```

➤ s.GroupBy(e => k)

Вот пример:

```
from c in customers
group c by c.Country
```

➤ customers.GroupBy(c => c.Country)

Если групповой элемент — не тот же идентификатор, что и перечислитель последовательности, в том смысле, что вы группируете нечто отличное от всего элемента, хранящегося в последовательности, — происходит следующая трансляция:

```
from e in s
group g by k
```

➤ s.GroupBy(e => k, e=> g)

И вот пример:

```
from c in customers
group new { c.Country, c.Name } by c.Country
```

```
customers
    .GroupBy(c => c.Country,
              C => new {
                  c.Country,
                  c.Name
              })
```

В этой точке все шаги трансляции завершены, и выражение запроса должно быть полностью транслировано в синтаксис стандартной точечной нотации.

Резюме

Как видите, команда разработчиков Microsoft C# хорошо потрудилась, добавляя расширения в язык C#. Все расширения C#, которые мы обсудили в этой главе, были предназначены специально для LINQ. Но даже без LINQ новые средства C# дают массу преимуществ.

Новые выражения инициализации объектов и коллекций — просто подарок. Установка статических данных, данных примеров или тестовых данных стала много легче, чем раньше, существенно сократив количество строк кода, необходимых для генерации данных. Это средство, в сочетании с новым ключевым словом var и анонимными типами, существенно упростило создание данных и типов данных “на лету”.

Расширяющие методы обеспечили возможность добавлять функциональность к объектам, таким как герметизированные классы или классы, исходного кода которых у вас нет, что было совершенно невозможно ранее.

Лямбда-выражения позволили сократить спецификации функциональности. Устранив потребность в анонимных методах, они добавили целый арсенал способов специфицировать простую функциональность, и лично мне нравится их лаконичный синтаксис. Хотя поначалу они могут показаться вам непривычными, я думаю, со временем и накоплением опыта, ваше отношение к ним улучшится.

Деревья выражений предоставляют независимым поставщикам, которые хотят добавить поддержку LINQ своим собственным хранилищам данных, возможность обеспечить первоклассную производительность.

Частичные методы представили очень легковесный механизм обработки событий. Microsoft использует их в своих инструментах генерации сущностных классов для LINQ to SQL, так что вы можете легко вмешиваться в работу сущностных классов в нужные моменты времени.

И, наконец, выражения запросов вызывают горячее одобрение, когда вы впервые видите запрос LINQ, что заставляет вас немедленно ими воспользоваться. Ничто так не облегчает задачу разработчику, изучающему новую технологию, как ее сходство с хорошо знакомой и зарекомендовавшей себя технологией. Придав запросам LINQ сходство с запросами SQL, Microsoft существенно облегчила их изучение.

Хотя все эти расширения языка сами по себе являются замечательными средствами, все вместе они формируют фундамент LINQ. Я верю, что LINQ станет следующим SQL или ООП, и большинство разработчиков .NET захотят упомянуть его в своем резюме. Знаю, что так и будет.

Теперь, познакомив вас с тем, что собой представляет LINQ, какой синтаксис и новые средства C# ему требуются, пришло время заняться практикой, и пусть вас это не пугает. Следующим шагом будет изучение выполнения запросов LINQ к находящимся в памяти коллекциям, таким как массивы, ArrayList и все новые обобщенные коллекции C# 2.0. Во второй части книги вы найдете комплект функций, дополняющих ваши запросы. Этот комплект LINQ известен под названием LINQ to Objects.

ЧАСТЬ II

LINQ to Objects

В этой части...

Глава 3. Введение в LINQ to Objects

Глава 4. Отложенные операции

Глава 5. Не отложенные операции

ГЛАВА 3

Введение в LINQ to Objects

Листинг 3.1. Простой запрос LINQ to Objects

```
string[] presidents = {  
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",  
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",  
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",  
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",  
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",  
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};  
string president = presidents.Where(p => p.StartsWith("Lin")).First();  
Console.WriteLine(president);
```

На заметку! Этот код был добавлен в консольное приложение Visual Studio 2008.

Листинг 3.1 демонстрирует практически все, что нужно знать о LINQ to SQL — выполнение SQL-подобных запросов к коллекциям и массивам, находящимся в памяти. Запустив пример нажатием <Ctrl+F5>, можно получить следующий результат:

Lincoln

Обзор LINQ to Objects

Отчасти то, что делает LINQ настолько мощным и удобным в применении, заключается в его тесной интеграции с языком C#. Вместо того, чтобы иметь дело с полностью новым набором средств в форме классов, которые должны быть использованы для получения преимуществ LINQ, вы можете применять те же самые коллекции¹ и массивы, к которым привыкли, адаптированные к вашим существующим классам. Это значит, что вы можете получить все преимущества запросов LINQ с минимальными модификациями существующего кода или вообще без них. Функциональность LINQ to Objects обеспечивается интерфейсом `IEnumerable<T>`, последовательностями и стандартными операциями запросов.

Например, если у вас есть массив целых чисел, и его нужно отсортировать, вы можете выполнить запрос LINQ для упорядочивания результатов — почти так же, как если

¹ Коллекция должна реализовывать `IEnumerable<T>` или `IEnumerable`, чтобы ее можно было опрашивать с помощью LINQ

бы это был запрос SQL. Может быть, у вас есть `ArrayList` объектов `Customer`, и нужно найти определенный объект `Customer`. В этом случае LINQ to Object — наилучший выбор.

Я знаю, что многие будут склонны использовать главы части "LINQ to Objects" в качестве руководства. Хотя я предпринял существенные усилия, чтобы сделать их полезными в этом отношении, разработчик получит больше, прочитав их полностью от начала до конца. Многие концепции, применимые к одной операции, также применимы и к другим. Я старался сделать каждый раздел, посвященный отдельной операции, как можно более независимым и самодостаточным, но при прочтении всего от начала до конца создается контекст, который будет утерян, если вы станете читать об одной операции и пропускать другие.

IEnumerable<T>, последовательности и стандартные операции запросов

`IEnumerable<T>` — это интерфейс, реализуемый всеми обобщенными классами коллекций C#, как и массивами. Этот интерфейс представляет перечисление элементов коллекций.

Последовательность — логический термин для коллекции, реализующей интерфейс `IEnumerable<T>`. Если у вас есть переменная типа `IEnumerable<T>`, то вы можете сказать, что у вас есть последовательность элементов типа `T`. Например, если имеется `IEnumerable` элементов `string`, записанная как `IEnumerable<string>`, значит, у вас есть последовательность строк.

На заметку! Любая переменная, объявленная как `IEnumerable<T>` для типа `T` рассматривается как последовательность типа `T`.

Большинство стандартных операций запросов представляют собой расширяющие методы в статическом классе `System.Linq.Enumerable` и прототипированы `IEnumerable<T>` в качестве первого аргумента. Поскольку они являются расширяющими методами, предпочтительно вызывать их на переменной типа `IEnumerable<T>`, что позволяет синтаксис расширяющих методов — вместо передачи переменной типа `IEnumerable<T>` в виде первого аргумента.

Методы стандартных операций запросов класса `System.Linq.Enumerable`, которые не являются расширяющими методами — это просто статические методы, которые должны быть вызваны на классе `System.Linq.Enumerable`. Комбинация этих методов стандартных операций запросов дают вам возможность выполнять сложные запросы данных на последовательности `IEnumerable<T>`.

Унаследованные (старые) коллекции, которые не являются обобщенными, существовавшие до C# 2.0, поддерживают интерфейс `IEnumerable`, а не `IEnumerable<T>`. Это значит, что вы не можете непосредственно вызывать эти расширяющие методы с первым аргументом типа `IEnumerable<T>` на унаследованных коллекциях. Однако вы можете выполнять запросы LINQ на унаследованных коллекциях, вызывая стандартную операцию запроса `Cast` или `OfType` на унаследованной коллекции, чтобы произвести последовательность, реализующую `IEnumerable<T>`, что открывает вам доступ кному арсеналу стандартных операций запросов.

На заметку! Используйте операции `Cast` или `OfType`, чтобы выполнять запросы LINQ на унаследованных, необобщенных коллекциях C#.

Чтобы получить доступ к стандартным операциям запросов, добавьте директиву `using System.Linq;` к вашему коду, если она еще там не присутствует. Вам не нужно добавлять ссылку на сборку, потому что необходимый код содержится в сборке `System.Core.dll`, которая автоматически добавляется к вашему проекту средой Visual Studio 2008.

Возврат `IEnumerable<T>`, Yielding и отложенные запросы

Важно помнить, что хотя многие из стандартных операций запросов прототипированы на возврат `IEnumerable<T>`, и мы воспринимаем `IEnumerable<T>` как последовательность, на самом деле операции не возвращают последовательность в момент их вызова. Вместо этого операции возвращают объект, который при перечислении порождает (*yield*) очередной элемент последовательности. Во время перечисления возвращенного объекта запрос выполняется, и порожденный элемент помещается в выходную последовательность. Таким образом, выполнение запроса отложено.

Если вам не знаком термин *yield*, скажу, что я ссылаюсь на ключевое слово C# 2.0 `yield`, которое было добавлено к языку C# для облегчения написания перечислителей.

Например, рассмотрим код, представленный в листинге 3.2.

Листинг 3.2. Тривиальный пример запроса

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Where(p => p.StartsWith("A"));
foreach(string item in items)
    Console.WriteLine(item);
```

Запрос, использующий операцию `Where`, на самом деле не выполняется, когда выполняется строка, содержащая запрос. Вместо этого возвращается объект. И только во время перечисления элементов возвращенного объекта этот запрос `Where` выполняется. Это значит, что ошибка, возникающая в самом запросе, может быть не обнаружена до тех пор, пока не начнется перечисление.

На заметку! Ошибки запроса могут быть не обнаружены до тех пор, пока не начнется перечисление выходной последовательности.

Результат предыдущего запроса выглядит так:

```
Adams
Arthur
```

Запрос выполняется, как ожидалось. Однако я намеренно внес ошибку. Следующий код пытается проиндексировать пятый символ каждого имени президента. Когда перечисление достигает элемента, чья длина меньше пяти символов, возникает исключение. Помните, однако, что исключение не происходит до тех пор, пока не начинается перечисление выходной последовательности. В листинге 3.3 показан пример кода.

Листинг 3.3. Тривиальный пример запроса с внутренней ошибкой

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Where(s => Char.ToLower(s[4]));
Console.WriteLine("After the query.");
foreach (string item in items)
    Console.WriteLine(item);

```

Этот код успешно компилируется, но при запуске дает такой результат:

```

After the query.
Adams
Arthur
Buchanan
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the
bounds of the array.
...

```

Обратите внимание на вывод After the query (после запроса). Его нет до тех пор, пока в перечислении не дойдет очередь до четвертого элемента — Bush, где возникает исключение. Урок, который можно отсюда извлечь, состоит в том, факт успешной компиляции запроса и кажущееся отсутствие проблем при его выполнении еще не говорит о том, что он свободен от ошибок.

Вдобавок, поскольку запросы такого рода, возвращающие `IEnumerable<T>`, являются отложенными, вы можете вызывать код определения запроса однажды, а затем использовать его многократно, перечисляя его результаты несколько раз. Если вы будете так поступать, то при каждом перечислении результатов вы получите разные результаты в случае изменения данных. В листинге 3.4 показан пример отложенного запроса, где результат не кэшируется и может изменяться от одного перечисления к другому.

Листинг 3.4. Пример, демонстрирующий изменение результатов запроса между перечислениями

```

// Создать массив целых чисел.
int[] intArray = new int[] { 1, 2, 3 };
IEnumerable<int> ints = intArray.Select(i => i);
// Отобразить результаты.
foreach(int i in ints)
    Console.WriteLine(i);
// Изменить элемент в источнике данных.
intArray[0] = 5;
Console.WriteLine("-----");
// Снова отобразить результат.
foreach(int i in ints)
    Console.WriteLine(i);

```

Чтобы лучше прояснить то, что здесь происходит, я внесу большую техничность в описание. Когда язываю операцию `Select`, возвращается объект, хранящийся в переменной `ints` типа, реализующего интерфейс `IEnumerable<int>`. В этой точке запрос в действительности еще не выполняется, но хранится в объекте по имени `ints`. Другими словами, поскольку запрос еще не выполнен, последовательность целых чисел

еще не существует, но этот объект `ints` знает, как получить последовательность, выполнив запрос, присвоенный ему, которым в этом случае является операция `Select`.

Когда я первый раз вызываю оператор `foreach` на `ints`, объект `ints` выполняет запрос и получает последовательность по одному элементу за раз.

Затем я изменяю один элемент в исходном массиве целых чисел. Затем снова вызываю оператор `foreach`. Это заставляет `ints` снова выполнить запрос. Поскольку я изменил элемент в исходном массиве, и запрос выполнен снова, потому что заново запущено перечисление `ints`, на этот раз возвращается измененный элемент.

Вызванный мною запрос вернул объект, реализующий `IEnumerable<int>`. Однако в большинстве случаев при обсуждении LINQ в этой главе, как и в других дискуссиях за рамками нашей книги, обычно говорится, что запрос вернул последовательность целых чисел. Логически это верно, и в конечном итоге именно так оно и есть. Но важно, чтобы вы понимали технический аспект вопроса — что именно происходит.

Вот результат запуска этого кода:

```
1
2
3
-----
5
2
3
```

Обратите внимание, что несмотря на то, что я вызвал запрос только однажды, результаты двух перечислений отличаются. Это еще одно доказательство того, что запрос является отложенным. Если бы это было не так, то результаты двух перечислений совпадали бы. Это могло быть как преимуществом, так и недостатком. Если вы не хотите, чтобы подобное случилось, используйте одну из операций преобразования, которые возвращают не `IEnumerable<T>`, так что запрос получается не отложенным, а `ToArray`, `ToList`, `ToDictionary` или `ToLookup`, создавая различные структуры данных с кэшированными результатами, которые не изменяются с изменением источника данных.

В листинге 3.5 показан тот же код, что и в предыдущем примере, но запрос возвращает не `IEnumerable<T>`, а `List<int>` — посредством вызова операции `ToList`.

Листинг 3.5. Возврат `List`, так что запрос исполняется немедленно, и результаты кэшируются

```
// Создать массив целых чисел.
int[] intArray = new int[] { 1, 2, 3 };
List<int> ints = intArray.Select(i => i).ToList();
// Отобразить результаты.
foreach(int i in ints)
    Console.WriteLine(i);
// Изменить элемент в исходном массиве.
intArray[0] = 5;
Console.WriteLine("-----");
// Снова отобразить результаты.
foreach(int i in ints)
    Console.WriteLine(i);
```

Вот результат:

```
1
2
3
-----
1
2
3
```

Обратите внимание, что результаты, полученные от двух перечислений, одинаковы. Это потому, что метод `ToList` не является отложенным, и запрос на самом деле выполняется там, где он вызван.

Возвращаясь к технической дискуссии об отличии между этим примером и листингом 3.4, где операция `Select` отложена, скажу, что `ToList` из листинга 3.5 таковой не является. Когда операция `ToList` вызывается в операторе запроса, она немедленно перечисляет объект, возвращенный оператором `Select`, в результате чего весь запрос перестает быть отложенным.

Делегаты Func

Несколько из стандартных операций запросов прототипированы на прием делегата `Func` в качестве аргумента. Это предотвращает явное объявление типов делегатов. Ниже приведены объявления делегата `Func`.

```
public delegate TR Func<TR>();
public delegate TR Func<T0, TR>(T0 a0);
public delegate TR Func<T0, T1, TR>(T0 a0, T1 a1);
public delegate TR Func<T0, T1, T2, TR>(T0 a0, T1 a1, T2 a2);
public delegate TR Func<T0, T1, T2, T3, TR>(T0 a0, T1 a1, T2 a2, T3 a3);
```

В каждом объявлении `TR` ссылается на возвращаемый тип данных. Обратите внимание, что тип возвращаемого аргумента `TR` присутствует в конце шаблона параметра типа каждой перегрузки делегата `Func`. Другие параметры типа — `T0`, `T1`, `T2` и `T3` — ссылаются на входные параметры, переданные методу. Существует множество объявлений, потому что некоторые стандартные операции запросов имеют аргументы-делегаты, требующие больше параметров, чем другие. Взглянув на объявления, вы можете увидеть, что ни одна из стандартных операций запросов не имеет аргумента-делегата, требующего более четырех входных параметров.

Давайте взглянем на один из прототипов операции `Where`:

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Аргумент-предикат специфицирован как `Func<T, bool>`. Отсюда вы можете видеть, что метод-предикат или лямбда-выражение должны принимать один аргумент — параметр `T`, и возвращать `bool`. Вы знаете это потому, что вам известен тип возврата, специфицированный в конце списка параметров шаблона.

Листинг 3.6. Пример использования одного из объявлений делегата Func

```
// Создать массив целых чисел.
int[] ints = new int[] { 1, 2, 3, 4, 5, 6 };
// Объявление нашего делегата.
Func<int, bool> GreaterThanTwo = i => i > 2;
// Выполнить запрос ... но не совсем - не забывайте об отложенных запросах!
IEnumerable<int> intsGreaterThanTwo = ints.Where(GreaterThanTwo);
// Отобразить результаты.
foreach(int i in intsGreaterThanTwo)
    Console.WriteLine(i);
```

Этот код вернет следующие результаты:

```
3
4
5
6
```

Алфавитный указатель стандартных операций запросов

В табл. 3.1 приведен упорядоченный по алфавиту список стандартных операций запросов. Поскольку эти операции будут разнесены по главам в соответствии с тем, являются ли они отложенными или нет, данная таблица поможет вам найти каждую операцию в остальных главах части "LINQ to Objects".

Таблица 3.1. Алфавитный указатель стандартных операций запросов

Операция	Назначение	Отложенный?
Aggregate	Агрегация	
All	Квантификатор	
Any	Квантификатор	
AsEnumerable	Преобразование	Да
Average	Агрегация	
Cast	Преобразование	Да
Concat	Конкатенация	Да
Contains	Квантификатор	
Count	Агрегация	
DefaultEmpty	Элемент	Да
Distinct	Множество	Да
ElementAt	Элемент	
ElementAtOrDefault	Элемент	
Empty	Генерация	Да
Except	Множество	Да
First	Элемент	
FirstOrDefault	Элемент	
GroupBy	Группировка	Да
GroupJoin	Соединение	Да
Interest	Множество	Да
Join	Соединение	Да
Last	Элемент	
LastOrDefault	Элемент	
LongCount	Агрегация	
Max	Агрегация	
Min	Агрегация	
OfType	Преобразование	Да
OrderBy	Упорядочивание	Да

Окончание табл. 3.1

Операция	Назначение	Отложенный?
OrderByDescending	Упорядочивание	Да
Range	Генерация	Да
Repeat	Генерация	Да
Reverse	Упорядочивание	Да
Select	Проекция	Да
SelectMany	Проекция	Да
SequenceEqual	Эквивалентность	
Single	Элемент	
SingleOrDefault	Элемент	
Skip	Разбиение	Да
SkipWhile	Разбиение	Да
Sum	Агрегация	
Take	Разбиение	Да
TakeWhile	Разбиение	Да
ThenBy	Упорядочивание	Да
ThenByDescending	Упорядочивание	Да
ToArray	Преобразование	
ToDictionary	Преобразование	
ToList	Преобразование	
ToLookup	Преобразование	
Union	Множество	Да
Where	Ограничение	Да

Резюме

В этой главе я представил понятие *последовательности*, и ее технический тип данных — `IEnumerable<T>`. Если вы чувствуете себя некомфортно с чем-то в этой терминологии, уверяю вас, что со временем она станет вашей второй натурой. Просто воспринимайте `IEnumerable<T>` как последовательность объектов, на которых вы собираетесь вызывать методы, чтобы выполнять с ними какие-то действия.

Однако наиболее существенной вещью, о которой шла речь в этой главе, является важность отложенного выполнения запросов. Отложенное выполнение может работать как на вас, так и против вас. Глубокое понимание этого механизма и его важности — ключ к успеху. Неспроста я разнес стандартные операции запросов по отдельным главам, основываясь на этой характеристике. Отложенные операции рассматриваются в главе 4, а не отложенные — в главе 5.

Заложив здесь в ваше сознание понятие отложенных запросов, в следующей главе я начну углубленное рассмотрение отложенных операций.

ГЛАВА 4

Отложенные операции

В предыдущей главе рассказывалось о том, что такое последовательности, о типах данных, представляющих их, а также об их влиянии на отложенное выполнение запросов. Из-за важности понимания отложенных запросов я разнес описание отложенных и не отложенных операций по разным главам, чтобы подчеркнуть, какие из стандартных операций запросов отложены, а какие — нет.

В этой главе мы поговорим об отложенных операциях. Отложенную операцию легко выделить, поскольку она должна возвращать тип `IEnumerable<T>` или `IOrderedEnumerable<T>`. Каждая из этих отложенных операций будет отнесена к определенной категории по ее назначению.

Чтобы кодировать и выполнять примеры этой главы, вам следует помнить об использовании директивы `using` со всеми необходимыми пространствами имен, ссылках на все необходимые сборки, а также на общий код, разделяемый примерами.

Необходимые пространства имен

В примерах этой главы будут использоваться пространства имен `System.Linq`, `System.Collections`, `System.Collections.Generic` и `System.Data.Linq`. Поэтому вы должны добавить следующие директивы `using` к вашему коду, если они еще там не представлены:

```
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using System.Data.Linq;
```

В дополнение к этим пространствам имен, если вы выгрузите сопровождающий код, то увидите, что я также добавил директиву `using` для пространства имен `System.Diagnostics`. Это не будет необходимо, если вы будете набирать примеры из этой главы. Это понадобится только в сопровождающем коде — из-за некоторого добавленного мною рабочего кода.

Необходимые сборки

В дополнение к типичным сборкам вам понадобится установить ссылку на сборку `System.Data.Linq.dll`.

Общие классы

Нескольким примерам этой главы понадобятся классы — для всесторонней демонстрации поведения операций. Список классов, которые будут использованы в более чем одном примере, следует ниже.

Класс Employee представляет сотрудника. Для удобства он содержит статические методы, возвращающие ArrayList или массивы сотрудников.

Разделяемый класс Employee

```
public class Employee
{
    public int id;
    public string firstName;
    public string lastName;

    public static ArrayList GetEmployeesArrayList()
    {
        ArrayList al = new ArrayList();
        al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
        al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
        al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
        al.Add(new Employee { id = 4, firstName = "David", lastName = "Lightman" });
        al.Add(new Employee { id = 101, firstName = "Kevin", lastName = "Flynn" });
        return (al);
    }

    public static Employee[] GetEmployeesArray()
    {
        return ((Employee[])GetEmployeesArrayList().ToArray());
    }
}
```

Класс EmployeeOptionEntry представляет назначение определенных опций определенному сотруднику. Для удобства он содержит статический метод, возвращающий список назначенных опций.

Разделяемый класс EmployeeOptionEntry

```
public class EmployeeOptionEntry
{
    public int id;
    public long optionsCount;
    public DateTime dateAwarded;
    public static EmployeeOptionEntry[] GetEmployeeOptionEntries()
    {
        EmployeeOptionEntry[] empOptions = new EmployeeOptionEntry[]
        {
            new EmployeeOptionEntry {
                id = 1,
                optionsCount = 2,
                dateAwarded = DateTime.Parse("1999/12/31") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("1992/06/30") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("1994/01/01") },
            new EmployeeOptionEntry {
                id = 3,
                optionsCount = 5000,
                dateAwarded = DateTime.Parse("1997/09/30") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("2003/04/01") },
        };
    }
}
```

```

        new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
        new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
        new EmployeeOptionEntry {
            id = 4,
            optionsCount = 1500,
            dateAwarded = DateTime.Parse("1997/12/31") },
        new EmployeeOptionEntry {
            id = 101,
            optionsCount = 2,
            dateAwarded = DateTime.Parse("1998/12/31") }
    };
    return (empOptions);
}
}

```

Разделение отложенных операций по их назначению

Отложенные стандартные операции запросов в данном разделе организованы по их предназначению.

Ограничение

Операции ограничения (restriction) используются для включения или исключения элементов из входной последовательности.

Where

Операция **Where** используется для фильтрации элементов в последовательность.

Прототипы

Операция **Where** имеет два прототипа, описанных ниже.

Первый прототип Where

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Этот прототип **Where** принимает входную последовательность, производит делегат метода и возвращает объект, который при перечислении проходит по входной последовательности, порождая элементы, для которых делегат метода-предиката возвращает `true`.

Поскольку это расширяющий метод, мы на самом деле не передаем входную последовательность, пока вызываем операцию **Where**, используя синтаксис метода экземпляра.

На заметку! Благодаря расширяющим методам нет необходимости передавать первый аргумент стандартной операции запроса, чей первый аргумент помечен модификатором — ключевым словом `this`, пока мы вызываем операцию на объекте того же типа, что и у первого аргумента.

При вызове `Where` вы передаете делегат метода-предиката. Этот метод-предикат должен принимать тип `T` в качестве ввода, причем `T` — тип элементов, содержащихся во входной последовательности, и возвращать `bool`. Операция `Where` вызовет ваш метод-предикат для каждого элемента входной последовательности и передаст ему этот элемент. Если ваш метод-предикат вернет `true`, то `Where` вставит этот элемент в выходную последовательность `Where`. Если ваш метод-предикат вернет `false`, `Where` этого не сделает.

Второй прототип `Where`

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

Второй прототип `Where` идентичен первому, но с тем отличием, что он указывает на то, что ваш делегат метода-предиката принимает дополнительный целочисленный аргумент. Этот аргумент будет номером-индексом элемента из входной последовательности.

Нумерация индекса начинается с нуля, поэтому индексом первого элемента будет 0. Последний элемент имеет номер, соответствующий количеству элементов в последовательности минус единица.

На заметку! Помните, что нумерация индексов начинается с нуля.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

В листинге 4.1 представлен пример вызова первого прототипа.

Листинг 4.1. Пример вызова первого прототипа `Where`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Where(p => p.StartsWith("J"));
foreach (string s in sequence)
    Console.WriteLine("{0}", s);
```

В приведенном примере ограничение последовательности с использованием первого прототипа операции `Where` выполняется просто вызовом метода `Where` на последовательности, с передачей лямбда-выражения, возвращающего `bool`, что указывает на то, должен ли элемент включаться в выходную последовательность. В этом примере я возвращаю только элементы, начинающиеся со строки "J". Этот код при нажатии `<Ctrl+F5>` выдаст следующий результат:

```
Jackson
Jefferson
Johnson
```

Обратите внимание, что я передал метод-предикат через лямбда-выражение.

В листинге 4.2 показан код, вызывающий второй прототип операции `Where`. Обратите внимание, что эта версия даже не использует самого элемента `p`, а использует только индекс `i`. Этот включит каждый второй элемент — с нечетным индексом — в выходную последовательность.

Листинг 4.2. Пример вызова второго прототипа `Where`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Where((p, i) => (i & 1) == 1);
foreach (string s in sequence)
    Console.WriteLine("{0}", s);
```

Нажатие `<Ctrl+F5>` выдаст следующий результат:

```
Arthur
Bush
Cleveland
Coolidge
Fillmore
Garfield
Harding
Hayes
Jackson
Johnson
Lincoln
McKinley
Nixon
Polk
Roosevelt
Taylor
Tyler
Washington
```

Проекция

Операции проекции возвращают последовательность элементов, сгенерированных выбором элементов или даже созданием новых экземпляров элементов, которые содержат части элементов из входной последовательности.

Тип данных элементов выходной последовательности может отличаться от типа элементов входной последовательности.

Select

Операция `Select` используется для создания выходной последовательности одного типа элементов из входной последовательности элементов другого типа. Нет необходимости, чтобы эти типы совпадали.

Прототипы

Есть два прототипа этой операции, которые описаны ниже.

Первый прототип Select

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

Этот прототип Select принимает входную последовательность и делегат метода-селектора в качестве входных параметров, а возвращает объект, который при перечислении проходит по входной последовательности и порождает последовательность элементов типа S. Как упоминалось ранее, T и S могут быть как одного типа, так и разных.

При вызове Select вы передаете делегат метода-селектора через аргумент selector. Ваш метод-селектор должен принимать тип T в качестве входного, где T — тип элементов, содержащихся во входной последовательности, и возвращать элемент типа S. Операция Select вызовет ваш метод-селектор для каждого элемента входной последовательности, передав ему этот элемент. Ваш метод-селектор выберет часть входного элемента, который его интересует, создаст новый элемент — возможно, иного типа (даже анонимного) — и вернет его.

Второй прототип Select

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, int, S> selector);
```

В этом прототипе операции Select методу-селектору передается дополнительный целочисленный параметр. Это будет индекс, начинающийся с нуля, входного элемента во входной последовательности.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

Пример вызова первого прототипа показан в листинге 4.3.

Листинг 4.3. Пример вызова первого прототипа Select

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> sequence = presidents.Select(p => p.Length);
foreach (int item in nameLengths)
    Console.WriteLine(item);
```

Обратите внимание, что метод-селектор передан через лямбда-выражение. В данном случае лямбда-выражение вернет длину каждого элемента из входной последовательности. Также отметьте, что хотя тип входных элементов — строка, тип выходного — целое число.

Нажатие <Ctrl+F5> выдаст следующий результат:

```
5
6
8
4
6
```

```

9
7
8
10
8
4
8
5
7
8
5
6
7
7
7
7
8
6
5
4
6
9
4
6
6
5
9
10
6

```

Это простой пример, потому что я не генерирую никакого класса. Чтобы увидеть лучшую демонстрацию первого прототипа, обратитесь к листингу 4.4.

Листинг 4.4. Другой пример вызова первого прототипа Select

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select(p => new { p, p.Length });
foreach (var item in nameObjs)
    Console.WriteLine(item);

```

Обратите внимание, что лямбда-выражение создает экземпляр нового анонимного типа. Компилятор динамически генерирует анонимный тип, который будет содержать `string p` и `int p.Length`, и мой метод-селектор вернет этот вновь созданный объект.

У меня нет имени типа, по которому можно было бы на него сослаться. Поэтому я не могу присвоить выходную последовательность `Select` экземпляру `IEnumerable<int>` определенного известного типа, как делал это в первом примере, где присваивал ее переменной типа `IEnumerable<int>`, представляющей выходную последовательность. Поэтому я присваиваю выходную последовательность переменной, специфицированной с ключевым словом `var`.

На заметку! Операции проекции, чьи методы-селекторы создают анонимные типы для возврата, должны присваивать свою выходную последовательность переменной, тип которой специфицирован ключевым словом var.

При запуске примера нажатием <Ctrl+F5> этот код производит следующий вывод:

```
{ p = Adams, Length = 5 }
{ p = Arthur, Length = 6 }
{ p = Buchanan, Length = 8 }
{ p = Bush, Length = 4 }
{ p = Carter, Length = 6 }
{ p = Cleveland, Length = 9 }
{ p = Clinton, Length = 7 }
{ p = Coolidge, Length = 8 }
{ p = Eisenhower, Length = 10 }
{ p = Fillmore, Length = 8 }
{ p = Ford, Length = 4 }
{ p = Garfield, Length = 8 }
{ p = Grant, Length = 5 }
{ p = Harding, Length = 7 }
{ p = Harrison, Length = 8 }
{ p = Hayes, Length = 5 }
{ p = Hoover, Length = 6 }
{ p = Jackson, Length = 7 }
{ p = Jefferson, Length = 9 }
{ p = Johnson, Length = 7 }
{ p = Kennedy, Length = 7 }
{ p = Lincoln, Length = 7 }
{ p = Madison, Length = 7 }
{ p = McKinley, Length = 8 }
{ p = Monroe, Length = 6 }
{ p = Nixon, Length = 5 }
{ p = Pierce, Length = 6 }
{ p = Polk, Length = 4 }
{ p = Reagan, Length = 6 }
{ p = Roosevelt, Length = 9 }
{ p = Taft, Length = 4 }
{ p = Taylor, Length = 6 }
{ p = Truman, Length = 6 }
{ p = Tyler, Length = 5 }
{ p = Van Buren, Length = 9 }
{ p = Washington, Length = 10 }
{ p = Wilson, Length = 6 }
```

С этим кодом связана одна проблема: я не могу управлять именами членов динамически генерированного анонимного класса. Однако, благодаря новому средству инициализации объектов C# 3.0, можно написать лямбда-выражение и специфицировать имена членов анонимного класса, как показано в листинге 4.5.

Листинг 4.5. Третий пример вызова первого прототипа Select

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select(p => new { LastName = p, Length = p.Length });
foreach (var item in nameObjs)
    Console.WriteLine("{0} is {1} characters long.", item.LastName, item.Length);
```

Обратите внимание, что я специфицировал имя для каждого члена в лямбда-выражении и затем обратился к каждому члену по имени в вызове метода `Console.WriteLine`. Вот результат запуска этого кода:

```
Adams is 5 characters long.
Arthur is 6 characters long.
Buchanan is 8 characters long.
Bush is 4 characters long.
Carter is 6 characters long.
Cleveland is 9 characters long.
Clinton is 7 characters long.
Coolidge is 8 characters long.
Eisenhower is 10 characters long.
Fillmore is 8 characters long.
Ford is 4 characters long.
Garfield is 8 characters long.
Grant is 5 characters long.
Harding is 7 characters long.
Harrison is 8 characters long.
Hayes is 5 characters long.
Hoover is 6 characters long.
Jackson is 7 characters long.
Jefferson is 9 characters long.
Johnson is 7 characters long.
Kennedy is 7 characters long.
Lincoln is 7 characters long.
Madison is 7 characters long.
McKinley is 8 characters long.
Monroe is 6 characters long.
Nixon is 5 characters long.
Pierce is 6 characters long.
Polk is 4 characters long.
Reagan is 6 characters long.
Roosevelt is 9 characters long.
Taft is 4 characters long.
Taylor is 6 characters long.
Truman is 6 characters long.
Tyler is 5 characters long.
Van Buren is 9 characters long.
Washington is 10 characters long.
Wilson is 6 characters long.
```

Для примера второго прототипа я добавлю индекс, который передается моему методу-селектору, в тип элемента выходной последовательности, как показано в листинге 4.6.

Листинг 4.6. Пример вызова второго прототипа `Select`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
var nameObjs = presidents.Select((p, i) => new { Index = i, LastName = p });
foreach (var item in nameObjs)
    Console.WriteLine("{0}. {1}", item.Index + 1, item.LastName);
```

Этот пример выведет номер индекса плюс единица, за которым выведет имя. Код произведет следующий сокращенный результат:

```

1. Adams
2. Arthur
3. Buchanan
4. Bush
5. Carter
...
34. Tyler
35. Van Buren
36. Washington
37. Wilson

```

SelectMany

Операция `SelectMany` используется для создания выходной последовательности — проекции “один ко многим” из входной последовательности. В то время как операция `Select` возвращает один выходной элемент для каждого входного элемента, `SelectMany` вернет ноль или более выходных элементов для каждого входного.

Прототипы

Существуют два прототипа этой операции, которые рассматриваются ниже.

Первый прототип SelectMany

```
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<S>> selector);
```

Этот прототип операции получает входную последовательность элементов типа `T` и делегат метода-селектора, а возвращает объект, который при перечислении проходит по входной последовательности, получая каждый элемент индивидуально из входной последовательности и передавая его в метод-селектор. Последний затем возвращает объект, который, будучи перечислимым, порождает ноль или более элементов типа `S` в промежуточной выходной последовательности. Операция `SelectMany` вернет конкатенированную выходную последовательность при каждом вызове вашего метода-селектора.

Второй прототип SelectMany

```
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, int, IEnumerable<S>> selector);
```

Этот прототип ведет себя так же, как и первый, за исключением того, что вашему методу-селектору дополнительно передается индекс, начинающийся с нуля, каждого элемента входной последовательности.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

В листинге 4.7 показан пример вызова первого прототипа.

Листинг 4.7. Пример вызова первого прототипа SelectMany

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
```

```

    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEquatable<char> chars = presidents.SelectMany(p => p.ToArray());
foreach (char ch in chars)
    Console.WriteLine(ch);

```

В предыдущем примере метод-селектор принимает на входе строку и, вызвав метод `ToCharArray` на этой строке, возвращает массив символов, который становится выходной последовательностью типа `char`.

Таким образом, для одного элемента входной последовательности, которым в данном случае является экземпляр `string`, мой метод-селектор возвращает последовательность символов. Для каждой входной строки выводится последовательность `char`. Операция `SelectMany` соединяет все эти последовательности символов в единую последовательность, которую и возвращает. Вывод предыдущего кода выглядит так:

```

A
d
a
m
s
A
r
t
h
u
r
B
u
c
h
a
n
a
n
B
u
s
h
...
W
a
s
h
i
n
g
t
o
n
W
i
l
s
o
n

```

Это достаточно простой запрос, но не слишком демонстративный в смысле типичного применения. В следующем примере я использую общие классы Employee и EmployeeOptionEntry.

Я вызову операцию SelectMany на массиве элементов Employee, и для каждого элемента Employee в массиве мой делегат метода-селектора вернет ноль или более элементов созданного мной анонимного класса, содержащего id и optionsCount из массива элементов EmployeeOptionEntry для объекта Employee. Взглянем на код в листинге 4.9, который делает все это.

Листинг 4.8. Более сложный пример вызова первого прототипа SelectMany

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.
GetEmployeeOptionEntries();
var employeeOptions = employees
    .SelectMany(e => empOptions
        .Where(eo => eo.id == e.id)
        .Select(eo => new {
            id = eo.id,
            optionsCount = eo.optionsCount }));
foreach (var item in employeeOptions)
    Console.WriteLine(item);
```

В этом примере каждый сотрудник в массиве Employee передается в лямбда-выражение, которое передано операции SelectMany. Это лямбда-выражение затем извлекает каждый элемент EmployeeOptionEntry, чей id соответствует id текущего сотрудника, переданного ему посредством операции Where. Это эффективно связывает массив Employee с массивом EmployeeOptionEntry по их членам id. Операция Select лямбда-выражения затем создает анонимный объект, содержащий члены id и optionsCount для каждой соответствующей записи в массиве EmployeeOptionEntry. Это значит, что лямбда-выражением возвращает последовательность из нуля или более анонимных объектов для каждого переданного сотрудника. Это дает в результате последовательность или последовательности, которые операция SelectMany может соединить вместе.

Предыдущий код даст такой вывод:

```
{ id = 1, optionsCount = 2 }
{ id = 2, optionsCount = 10000 }
{ id = 2, optionsCount = 10000 }
{ id = 2, optionsCount = 10000 }
{ id = 3, optionsCount = 5000 }
{ id = 3, optionsCount = 7500 }
{ id = 3, optionsCount = 7500 }
{ id = 4, optionsCount = 1500 }
{ id = 101, optionsCount = 2 }
```

Хотя и немного надуманно, пример из листинга 4.9 показывает вызов второго прототипа SelectMany.

Листинг 4.9. Пример вызова второго прототипа SelectMany

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
IEnumerable<char> chars = presidents
    .SelectMany((p, i) => i < 5 ? p.ToArray() : new char[] { });
foreach (char ch in chars)
    Console.WriteLine(ch);
```

Лямбда-выражение, которое я здесь представил, проверяет входной индекс и выводит массив символов из входной строки только в том случае, если значение индекса меньше пяти. Это значит, что я получу символы первых пяти входных строк, что и подтверждает следующий вывод:

```
A  
d  
a  
m  
s  
A  
r  
t  
h  
u  
r  
B  
u  
c  
h  
a  
n  
a  
n  
B  
u  
s  
h  
C  
a  
r  
t  
e  
r
```

Имейте в виду, что это лямбда-выражение не особо эффективно, особенно, если входных элементов много. Дело в том, что лямбда-выражение вызывается для каждого входного элемента. Я просто возвращаю пустой массив после обработки первых пяти входных элементов. Для повышения производительности я предпочитаю операцию `Take`, которую опишу в следующем разделе этой главы. Операция `SelectMany` также полезна для соединения множества последовательностей в одну. Читайте об этом в разделе, посвященном операции `Concat`, далее в этой главе.

Разбиение

Операции разбиения (partitioning) позволяют вернуть выходную последовательность, которая является подмножеством входной последовательности.

Take

Операция `Take` возвращает указанное количество элементов из входной последовательности, начиная с ее начала.

Прототипы

Операция Take имеет один прототип, описанный ниже.

Прототип Take

```
public static IEnumerable<T> Take<T>(
    this IEnumerable<T> source,
    int count);
```

Этот прототип специфицирует, что Take принимает входную последовательность и целое число count, задающее количество элементов, которые нужно вернуть, и возвращает объект, который при перечислении порождает первые count элементов из входной последовательности.

Если значение count больше, чем количество элементов во входной последовательности, тогда каждый элемент из нее попадает в выходную последовательность.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

В листинге 4.10 показан пример вызова первого прототипа.

Листинг 4.10. Пример вызова единственного прототипа Take

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Take(5);
foreach (string item in items)
    Console.WriteLine(item);
```

Этот код вернет первые пять элементов из массива presidents. Полученный результат выглядит следующим образом:

```
Adams
Arthur
Buchanan
Bush
Carter
```

В листинге 4.9 приведен код, о котором говорилось, что он был бы более эффективным при условии использования операции Take вместо индекса, передаваемого лямбда-выражению. Листинг 4.11 демонстрирует эквивалентный код с применением операции Take. Он дает точно такой же результат, что и код из листинга 4.9, но при этом намного эффективнее.

Листинг 4.11. Другой пример вызова прототипа Take

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
```

```

"Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
"Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson");
IEnumerable<string> items = presidents.Take(5).SelectMany(s => s.ToArray());
foreach (char ch in chars)
    Console.WriteLine(ch);

```

Подобно примеру SelectMany, использующему второй прототип из листинга 4.9, предыдущий код возвращает следующие результаты:

```

A
d
a
m
s
A
r
t
h
u
r
B
u
c
h
a
n
a
n
B
u
s
h
C
a
r
t
e
r

```

Разница между этим примером кода и примером из листинга 4.9 в том, что этот принимает только первые пять элементов из входной последовательности, и только они передаются в качестве входной последовательности в SelectMany. Другой пример кода, из листинга 4.9, передает все элементы SelectMany; он просто возвращает пустой массив для всех входных элементов кроме первых пяти.

TakeWhile

Операция TakeWhile возвращает элементы из входной последовательности, пока истинно некоторое условие, начиная с начала последовательности. Остальные входные элементы пропускаются.

Прототипы

Операция TakeWhile имеет два прототипа, описанные ниже.

Первый прототип TakeWhile

```

public static IEnumerable<T> TakeWhile<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);

```

Эта операция `TakeWhile` принимает входную последовательность и делегат метода-предиката, а возвращает объект перечисление по которому порождает элементы до тех пор, пока метод-предикат не вернет `false`. Метод-предикат принимает элементы по одному из входной последовательности и возвращает признак того, должен элемент включаться в выходную последовательность или нет. Если да, обработка входных элементов продолжается. Как только метод-предикат вернет `false`, никакие последующие входные элементы не обрабатываются.

Второй прототип TakeWhile

```
public static IEnumerable<T> TakeWhile<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

Этот прототип подобен первому, за исключением того, что метод-предикат получает вдобавок начинающийся с нуля индекс элемента из входной последовательности.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

В листинге 4.12 приведен пример вызова первого прототипа.

Листинг 4.12. Пример вызова первого прототипа TakeWhile

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.TakeWhile(s => s.Length < 10);
foreach (string item in items)
    Console.WriteLine(item);
```

В приведенном коде я хотел извлекать входные элементы до тех пор, пока их длина не превышает 10 символов.

Вот результат:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
```

`Eisenhower` (Эйзенхауэр) — имя, которое заставило операцию `TakeWhile` прекратить обработку входных элементов. Теперь я представлю пример второго прототипа операции `TakeWhile` в листинге 4.13.

Листинг 4.13. Пример вызова второго прототипа TakeWhile

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
```

```

    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents
    .TakeWhile((s, i) => s.Length < 10 && i < 5);
foreach (string item in items)
    Console.WriteLine(item);

```

Этот пример прекратит выполнение, когда входной элемент превысит 9 символов в длину, или когда будет достигнут шестой элемент — в зависимости от того, что произойдет раньше. Вот результат:

```

Adams
Arthur
Buchanan
Bush
Carter

```

В этом случае обработка остановилась по достижении шестого элемента.

Skip

Операция Skip пропускает указанное количество элементов из входной последовательности, начиная с ее начала, и выводит остальные.

Прототипы

Операция Skip имеет один прототип, описанный ниже.

Прототип Skip

```

public static IEnumerable<T> Skip<T>(
    this IEnumerable<T> source,
    int count);

```

Операция Skip получает входную последовательность и целое число по имени count, задающее количество входных элементов, которое должно быть пропущено, и возвращает объект, который при перечислении пропускает первые count элементов, и выводит все последующие элементы.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

В листинге 4.14 показан пример вызова операции Skip.

Листинг 4.14. Пример вызова единственного прототипа Skip

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Skip(1);
foreach (string item in items)
    Console.WriteLine(item);

```

В данном примере я хотел пропустить первый элемент. Обратите внимание, что в следующем выводе действительно пропущен первый входной элемент "Adams":

```
Arthur
Buchanan
Bush
...
Van Buren
Washington
Wilson
```

SkipWhile

Операция SkipWhile обрабатывает входную последовательность, пропуская элементы до тех пор, пока условие истинно, а затем выводит остальные в выходную последовательность.

Прототипы

У операции SkipWhile есть два прототипа, описанных ниже.

Первый прототип SkipWhile

```
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Операция SkipWhile принимает входную последовательность и делегат метода-предиката, и возвращает объект, который при перечислении пропускает элементы до тех пор, пока метод-предикат возвращает true. Как только метод-предикат вернет false, операция SkipWhile начинает вывод всех прочих элементов. Метод-предикат принимает элементы входной последовательности по одному, и возвращает признак того, должен ли элемент быть пропущен из входной последовательности.

SkipWhile имеет второй прототип, который выглядит следующим образом.

Второй прототип SkipWhile

```
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> source,
    Func<T, int, bool> predicate);
```

Этот прототип подобен первому во всем, за исключением дополнительного параметра — начинающегося с нуля индекса элемента из входной последовательности.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

В листинге 4.15 показан пример вызова первого прототипа SkipWhile.

Листинг 4.15. Пример вызова первого прототипа SkipWhile

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
IEnumerable<string> items = presidents.SkipWhile(s => s.StartsWith("A"));
foreach (string item in items)
    Console.WriteLine(item);
```

В этом примере я говорю методу SkipWhile пропускать элементы до тех пор, пока они начинаются с буквы "A". Все остальные элементы будут выведены в выходную последовательность. Ниже приведен результат предыдущего запроса:

```
Buchanan
Bush
Carter
...
Van Buren
Washington
Wilson
```

Теперь я попробую второй прототип SkipWhile, показанный в листинге 4.16.

Листинг 4.16. Пример вызова второго прототипа SkipWhile

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents
    .SkipWhile((s, i) => s.Length > 4 && i < 10);
foreach (string item in items)
    Console.WriteLine(item);
```

В данном примере я пропускаю входные элементы до тех пор, пока их длина не превышает четыре символа, или пока не будет достигнут десятый элемент. Остальные элементы выводятся в выходную последовательность. И вот результат:

```
Bush
Carter
Cleveland
...
Van Buren
Washington
Wilson
```

В данном случае я прекратил пропускать элементы, как только встретил "Bush", поскольку это имя было не длиннее четырех символов, хотя его индекс — всего 3.

Конкатенация

Операция конкатенации позволяет объединить несколько однотипных входных последовательностей в одну выходную.

Concat

Операция Concat соединяет две входные последовательности одного и порождает одну выходящую последовательность.

Прототипы

У операции Concat есть два прототипа, описанных ниже.

Первый прототип Concat

```
public static IEnumerable<T> SkipWhile<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

В этом прототипе две последовательности одного типа T являются входными — first и second. Возвращается объект, который при перечислении проходит по первой последовательности, выводя каждый ее элемент в выходную последовательность, за которой идет перечисление второй входной последовательности, выводящее каждый ее элемент туда же — в выходную последовательность.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

Листинг 4.17 демонстрирует пример использования операции Concat, а также операций Take и Skip.

Листинг 4.17. Пример вызова единственного прототипа Concat

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Take(5).Concat(presidents.Skip(5));
foreach (string item in items)
    Console.WriteLine(item);
```

Этот код берет пять первых членов из входной последовательности presidents, и соединяет со всеми, кроме первых пяти входных элементов из последовательности presidents. В результате должна получиться последовательность, по содержимому идентичная последовательности presidents, следующего вида:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
```

```
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Альтернативная техника соединения состоит в вызове операции `SelectMany` на массиве последовательностей, как показано в листинге 4.18.

Листинг 4.18. Пример выполнения конкатенации, альтернативного применению операции `Concat`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = new[] {
    presidents.Take(5),
    presidents.Skip(5)
}
.SelectMany(s => s);
foreach (string item in items)
    Console.WriteLine(item);
```

В данном примере я создаю экземпляр массива, состоящего из двух последовательностей: одной, созданной вызовом операции `Take` на входной последовательности, и другой — созданной вызовом операции `Skip` на входной последовательности. Обратите внимание, что это подобно предыдущему примеру во всем, за исключением того, что я вызываю операцию `SelectMany` на массиве последовательностей. К тому же, в то время как операция `Concat` позволяет только двум последовательностям соединяться вместе, поскольку эта техника допускает массив последовательностей, может оказаться удобнее, когда есть потребность в объединении вместе более двух последовательностей.

Совет. Когда нужно объединить более двух последовательностей вместе, рассмотрите возможность использования подхода на основе `SelectMany`.

Конечно, ничто из этого не имеет значения, если вы не получаете того же результата, что и при вызове операции `Concat`. Конечно, это не проблема, поскольку результат тот же:

```
Adams
Arthur
Buchanan
Bush
```

```
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Упорядочивание

Операции упорядочивания позволяют выстраивать входные последовательности в определенном порядке. Важно отметить, что и `OrderBy`, и `OrderByDescending` требуют входной последовательности типа `IEnumerable<T>`, и возвращают последовательность типа `IOrderedEnumerable<T>`. Вы не можете передать `IOrderedEnumerable<T>` в качестве входной последовательности операциям `OrderBy` и `OrderByDescending`. Это значит, что вы не можете передавать возвращенную последовательности ни из `OrderBy`, ни из `OrderByDescending` в последующие вызовы операции `OrderBy` или `OrderByDescending`.

Если вам нужно больше упорядочивания, чем это возможно сделать за один вызов операции `OrderBy` или `OrderByDescending`, вы должны последовательно вызывать операции `ThenBy` или `ThenByDescending`, потому что они принимают в качестве входной последовательности `IOrderedEnumerable<T>` и возвращают в качестве выходной `IOrderedEnumerable<T>` последовательности.

Например, следующая последовательность вызовов недопустима:

```
inputSequence.OrderBy(s => s.LastName).OrderBy(s => s.FirstName)...
```

Вместо нее вы должны использовать такую:

```
inputSequence.OrderBy(s => s.LastName).ThenBy(s => s.FirstName)...
```

OrderBy

Операция OrderBy позволяет упорядочить входную последовательность на основе метода keySelector, который возвращает ключевое значение для каждого входного элемента, и упорядоченная выходная последовательность IOrderedEnumerable<T> порождается в порядке возрастания на основе значений возвращенных ключей.

Сортировка, выполненная операцией OrderBy, определена как *нестабильная*. Это значит, что она не предохраняет входной порядок элементов. Если два входных элемента поступают в операцию OrderBy в определенном порядке, и ключевые значения этих двух элементов совпадают, их расположение в выходной последовательности может остаться прежним или поменяться, причем ни то, ни другое не гарантировано. Даже если может показаться, то все стабильно, но поскольку порядок определен как нестабильный, вы всегда должны исходить из этого. Это значит, что вы никогда не должны полагаться на порядок элементов, выходящих из операций OrderBy или OrderByDescending, для любого поля кроме специфицированного в вызове метода. Сохранение любого порядка, присутствующего в последовательности, переданной любой из этих операций, не может быть гарантировано.

Внимание! Сортировка, выполняемая OrderBy и OrderByDescending, нестабильна.

Прототипы

Операция OrderBy имеет два прототипа, описанных ниже.

Первый прототип OrderBy

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector)
    where
        K : IComparable<K>;
```

В этом прототипе OrderBy входная последовательность source передается в операцию OrderBy наряду с делегатом метода keySelector, и возвращается объект, который при перечислении проходит входную коллекцию source, собирая все элементы и передавая каждый из них методу keySelector, тем самым извлекая каждый ключ и упорядочивая последовательность на основе этих ключей.

Методу keySelector передается входной элемент типа T и возвращается поле внутри элемента, которое используется в качестве значения ключа типа K этого входного элемента. Типы K и T могут быть одинаковыми или разными. Тип значения, возвращенного методом keySelector, должен реализовывать интерфейс IComparable.

OrderBy имеет и второй прототип, который выглядит следующим образом.

Второй прототип OrderBy

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
```

Этот прототип такой же, как первый, за исключением того, что он позволяет передавать объект-компаратор. Если используется эта версия операции OrderBy, то нет необходимости в том, чтобы тип K реализовывал интерфейс IComparable.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

Листинг 4.19 демонстрирует пример вызова первого прототипа.

Листинг 4.19. Пример вызова первого прототипа OrderBy

```
string[] presidents = {  
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",  
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",  
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",  
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",  
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",  
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};  
IEnumerable<string> items = presidents.OrderBy(s => s.Length);  
foreach (string item in items)  
    Console.WriteLine(item);
```

Этот пример упорядочивает список президентов по длине их имен. Результат выглядит так:

```
Bush  
Ford  
Polk  
Taft  
Adams  
Grant  
Hayes  
Nixon  
Tyler  
Arthur  
Carter  
Hoover  
Monroe  
Pierce  
Reagan  
Taylor  
Truman  
Wilson  
Clinton  
Harding  
Jackson  
Johnson  
Kennedy  
Lincoln  
Madison  
Buchanan  
Coolidge  
Fillmore  
Garfield  
Harrison  
McKinley  
Cleveland  
Jefferson  
Roosevelt  
Van Buren  
Eisenhower  
Washington
```

Теперь я испытую пример второго прототипа, используя свой собственный компаратор. Прежде чем пояснить код, стоит взглянуть на интерфейс IComparer.

Интерфейс IComparer<T>

```
interface IComparer<T> {
    int Compare(T x, T y);
}
```

Интерфейс `IComparer` требует от меня реализовать единственный метод по имени `Compare`. Этот метод будет принимать два аргумента одного и того же типа `T` и возвращать значение `int` меньше нуля, если первый аргумент меньше второго, ноль — если аргументы эквивалентны, и значение больше нуля — если второй аргумент больше первого. Обратите внимание, насколько в этом интерфейсе и прототипе пригодились обобщения, добавленные в C# 2.0.

В этом примере для ясности я не использую никакого компаратора по умолчанию. Я создал класс, реализующий интерфейс `IComparer`, который упорядочивает элементы исходя из частоты гласных букв.

Моя реализация интерфейса `IComparer` для примера вызова второго прототипа `OrderBy`

```
public class MyVowelToConsonantRatioComarer : IComparer<string>
{
    public int Compare(string s1, string s2)
    {
        int vCount1 = 0;
        int cCount1 = 0;
        int vCount2 = 0;
        int cCount2 = 0;
        GetVowelConsonantCount(s1, ref vCount1, ref cCount1);
        GetVowelConsonantCount(s2, ref vCount2, ref cCount2);
        double dRatio1 = (double)vCount1 / (double)cCount1;
        double dRatio2 = (double)vCount2 / (double)cCount2;
        if(dRatio1 < dRatio2)
            return(-1);
        else if (dRatio1 > dRatio2)
            return(1);
        else
            return(0);
    }
    // Это общедоступный метод, так что мой код, использующий
    // данный компаратор, может получить нужные значения.
    public void GetVowelConsonantCount(string s,
                                         ref int vowelCount,
                                         ref int consonantCount)
    {
        // ВНИМАНИЕ: этот код предназначен только для демонстрационных целей.
        // Код всегда трактует 'у' и 'Y' как гласные, что с
        // лингвистической точки зрения, может быть, и неверно.
        string vowels = "AEIOUY";

        // Инициализация счетчиков.
        vowelCount = 0;
        consonantCount = 0;
        // Преобразование в верхний регистр,
        // чтобы избежать зависимости от регистра.
        string sUpper = s.ToUpper();
        foreach(char ch in sUpper)
        {
            if(vowels.IndexOf(ch) < 0)
                consonantCount++;
        }
    }
}
```

```

        else
            vowelCount++;
    }
    return;
}
}

```

Этот класс содержит два метода — Compare и GetVowelConstantCount. Метод Compare требуется интерфейсом IComparer. Метод GetVowelConstantCount существует потому, что он нужен для внутреннего использования в методе Compare, чтобы получить количество гласных и согласных в данной входной строке. Также мне нужна была возможность вызывать ту же логику извне метода Compare, чтобы можно было получать значения для отображения при проходе циклом по упорядоченной последовательности.

Логика того, что мой компаратор делает, не столь важна. Весьма маловероятно, что вам когда-либо понадобится определять частоту гласных в строке, и еще менее вероятно, что придется сравнивать две строки на основе этой частоты. Важнее то, как я создал класс, реализующий интерфейс IComparer, реализуя метод Compare. Вы можете видеть здесь полезную реализацию метода Compare в виде блока if/else в конце этого метода. Как видите, в этом блоке кода я возвращаю -1, 1 или 0, тем самым следуя контракту интерфейса IComparer.

А теперь я вызову код, показанный в листинге 4.20.

Листинг 4.20. Пример вызова первого прототипа OrderBy

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
    .OrderBy(s => s, myComp);
foreach (string item in namesByVToCRatio)
{
    int vCount = 0;
    int cCount = 0;
    myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
    double dRatio = (double)vCount / (double)cCount;
    Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}

```

В приведенном примере вы можете видеть, как я создаю экземпляр моего компаратора перед вызовом операции OrderBy. Я мог бы создать его экземпляр в вызове метода OrderBy, но затем не смог бы ссылаться на него, когда хотел вызвать его в цикле foreach. Вот результат работы этого кода:

```

Grant - 0.25 - 1:4
Bush - 0.3333333333333333 - 1:3
Ford - 0.3333333333333333 - 1:3
Polk - 0.3333333333333333 - 1:3
Taft - 0.3333333333333333 - 1:3
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5

```

Johnson - 0.4 - 2:5
 Lincoln - 0.4 - 2:5
 Washington - 0.428571428571429 - 3:7
 Arthur - 0.5 - 2:4
 Carter - 0.5 - 2:4
 Cleveland - 0.5 - 3:6
 Jefferson - 0.5 - 3:6
 Truman - 0.5 - 2:4
 Van Buren - 0.5 - 3:6
 Wilson - 0.5 - 2:4
 Buchanan - 0.6 - 3:5
 Fillmore - 0.6 - 3:5
 Garfield - 0.6 - 3:5
 Harrison - 0.6 - 3:5
 McKinley - 0.6 - 3:5
 Adams - 0.666666666666667 - 2:3
 Nixon - 0.666666666666667 - 2:3
 Tyler - 0.666666666666667 - 2:3
 Kennedy - 0.75 - 3:4
 Madison - 0.75 - 3:4
 Roosevelt - 0.8 - 4:5
 Coolidge - 1 - 4:4
 Eisenhower - 1 - 5:5
 Hoover - 1 - 3:3
 Monroe - 1 - 3:3
 Pierce - 1 - 3:3
 Reagan - 1 - 3:3
 Taylor - 1 - 3:3
 Hayes - 1.5 - 3:2

Как видите, президенты, в чьих именах отношение гласных к согласным меньше, следуют первыми.

OrderByDescending

Эта операции прототипирована и ведет себя так же, как и операция OrderBy, но с тем отличием, что упорядочивает в обратном порядке.

Прототипы

Эта операция имеет два прототипа, описанных ниже.

Первый прототип OrderByDescending

```
public static IOrderedEnumerable<T> OrderByDescending<T, K>(
  this IEnumerable<T> source,
  Func<T, K> keySelector)
where
  K : IComparable<K>;
```

Этот прототип операции OrderByDescending ведет себя так, как его эквивалентный прототип OrderBy, за исключением того, что выполняет обратное упорядочивание.

Внимание! Сортировка, выполняемая OrderBy и OrderByDescending, нестабильна.

OrderByDescending имеет и второй прототип, который выглядит следующим образом.

Второй прототип OrderByDescending

```
public static IOrderedEnumerable<T> OrderByDescending<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
```

Этот прототип — тот же, что и первый, за исключением того, что позволяет передавать объект-компаратор. Если используется эта версия операции OrderByDescending, то типу K не обязательно реализовывать интерфейс IComparable.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

В примере первого прототипа, показанном в листинге 4.21, я упорядочу список президентов в обратном порядке по их именам.

Листинг 4.21. Пример вызова первого прототипа OrderByDescending

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.OrderByDescending(s => s);
foreach (string item in items)
    Console.WriteLine(item);
```

Как видите, имена президентов расположены в порядке, противоположном алфавитному:

```
Wilson
Washington
Van Buren
Tyler
Truman
Taylor
Taft
Roosevelt
Reagan
Polk
Pierce
Nixon
Monroe
McKinley
Madison
Lincoln
Kennedy
Johnson
Jefferson
Jackson
Hoover
Hayes
Harrison
Harding
```

```

Grant
Garfield
Ford
Fillmore
Eisenhower
Coolidge
Clinton
Cleveland
Carter
Bush
Buchanan
Arthur
Adams

```

Теперь я испытую пример второго прототипа `OrderByDescending`. Для этого я использую тот же пример, что и для второго прототипа операции `OrderBy`, за исключением того, что вместо вызова `OrderBy` будет выполнен вызов `OrderByDescending`. Я использую тот же компаратор `MyVowelToConsonantRatioComparer`, что и в том примере. Код показан в листинге 4.22.

Листинг 4.22. Пример вызова второго прототипа `OrderByDescending`

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
    .OrderByDescending((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
    int vCount = 0;
    int cCount = 0;
    myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
    double dRatio = (double)vCount / (double)cCount;
    Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}

```

Этот пример работает так же, как эквивалентный пример `OrderBy`. Вот его результат:

```

Hayes - 1.5 - 3:2
Coolidge - 1 - 4:4
Eisenhower - 1 - 5:5
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Roosevelt - 0.8 - 4:5
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Adams - 0.66666666666667 - 2:3
Nixon - 0.66666666666667 - 2:3
Tyler - 0.66666666666667 - 2:3
Buchanan - 0.6 - 3:5

```

```

Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Truman - 0.5 - 2:4
Van Buren - 0.5 - 3:6
Wilson - 0.5 - 2:4
Washington - 0.428571428571429 - 3:7
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Bush - 0.3333333333333333 - 1:3
Ford - 0.3333333333333333 - 1:3
Polk - 0.3333333333333333 - 1:3
Taft - 0.3333333333333333 - 1:3
Grant - 0.25 - 1:4

```

Результаты те же, что и у эквивалентного примера OrderBy, только в обратном порядке. Теперь президенты перечислены в порядке убывания отношения гласных к согласным в их именах.

ThenBy

Операция ThenBy позволяет упорядочивать входную последовательность типа IOrderedEnumerable<T> на основе метода keySelector, который вернет значение ключа, и при этом порождается упорядоченная последовательность типа IOrderedEnumerable<T>.

На заметку! И ThenBy, и ThenByDescending принимают другой тип входных последовательностей, не такой как большинство операций отложенных запросов LINQ to Objects. Они принимают IOrderedEnumerable<T> в качестве входной последовательности. Это значит, что операция OrderBy или OrderByDescending должна быть вызвана первой, чтобы создать IOrderedEnumerable, на котором вы сможете затем вызвать операцию ThenBy или ThenByDescending.

Сортировка, выполняемая операцией ThenBy, *стабильна*. Это значит, что она предохраняет входной порядок элементов с эквивалентными ключами. Поэтому, если два входных элемента поступили в операцию ThenBy в определенном порядке, и ключевое значение обоих элементов одинаково, то порядок тех же выходных элементов гарантированно сохранится.

На заметку! В отличие от OrderBy и OrderByDescending, операции ThenBy и ThenByDescending выполняют стабильную сортировку.

Прототипы

Операция ThenBy имеет два прототипа, описанные ниже.

Первый прототип ThenBy

```
public static IOrderedEnumerable<T> ThenBy<T, K>(
    this IOrderedEnumerable<T> source,
    Func<T, K> keySelector)
where
    K : IComparable<K>;
```

В этом прототипе операции ThenBy упорядоченная входная последовательность типа `IOrderedEnumerable<T>` передается операции ThenBy наряду с делегатом метода `keySelector`. Метод `keySelector` принимает элемент типа `T` и возвращает поле внутри элемента, которое используется в качестве ключевого значения с типом `K` для входного элемента. Типы `T` и `K` могут быть как одинаковыми, так и различными. Значение, возвращенное методом `keySelector`, должно реализовать интерфейс `IComparable`. Операция ThenBy упорядочит входную последовательность в порядке возрастания на основе возвращенных ключей.

Есть и второй прототип.

Второй прототип ThenBy

```
public static IOrderedEnumerable<T> ThenBy<T, K>(
    this IOrderedEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);
```

Этот прототип — такой же, как и первый, за исключением того, что позволяет передать ему объект-компаратор. Если используется эта версия операции ThenBy, то типу `K` не обязательно реализовывать интерфейс `IComparable`.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

В листинге 4.23 показан пример первого прототипа.

Листинг 4.23. Пример вызова первого прототипа ThenBy

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.OrderBy(s => s.Length).ThenBy(s => s);
foreach (string item in items)
    Console.WriteLine(item);
```

Этот пример сначала упорядочивает элементы по их длине, в данном случае — длине имени президента. Затем упорядочивает по самому элементу. В результате получаем список имён, отсортированный по длине — от коротких к длинным, а затем — по алфавиту. Вот доказательство:

```
Bush
Ford
Polk
Taft
```

```

Adams
Grant
Hayes
Nixon
Tyler
Arthur
Carter
Hoover
Monroe
Pierce
Reagan
Taylor
Truman
Wilson
Clinton
Harding
Jackson
Johnson
Kennedy
Lincoln
Madison
Buchanan
Coolidge
Fillmore
Garfield
Harrison
McKinley
Cleveland
Jefferson
Roosevelt
Van Buren
Eisenhower
Washington

```

Для примера второго прототипа ThenBy я вновь использую свой объект-компаратор MyVowelToConsonantRatioComparer, который представил в примере второго прототипа OrderBy. Однако чтобы вызвать ThenBy, сначала я должен вызвать либо OrderBy, либо OrderByDescending. Для данного примера я вызову OrderBy и выстрою список по количеству символов в имени. Таким образом, имена будут упорядочены по возрастанию числа букв в них, а затем — внутри каждой группы с одинаковой длиной — по частоте гласных. Этот пример показан в листинге 4.24.

Листинг 4.24. Пример вызова второго прототипа ThenBy

```

string[] presidents =
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
    .OrderBy(n => n.Length)
    .ThenBy((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
    int vCount = 0;
    int cCount = 0;
}

```

```

myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
double dRatio = (double)vCount / (double)cCount;
Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}

```

Этот код выдаст следующий результат:

```

Bush - 0.3333333333333333 - 1:3
Ford - 0.3333333333333333 - 1:3
Polk - 0.3333333333333333 - 1:3
Taft - 0.3333333333333333 - 1:3
Grant - 0.25 - 1:4
Adams - 0.6666666666666667 - 2:3
Nixon - 0.6666666666666667 - 2:3
Tyler - 0.6666666666666667 - 2:3
Hayes - 1.5 - 3:2
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Truman - 0.5 - 2:4
Wilson - 0.5 - 2:4
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Coolidge - 1 - 4:4
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Van Buren - 0.5 - 3:6
Roosevelt - 0.8 - 4:5
Washington - 0.428571428571429 - 3:7
Eisenhower - 1 - 5:5

```

Как и следовало ожидать, имена сначала упорядочены по их длине, а затем — по частоте гласных.

ThenByDescending

Эта операция прототипирована и ведет себя подобно операции ThenBy, за исключением того, что выстраивает элементы в обратном порядке.

Прототипы

Операция имеет два прототипа, описанных ниже.

Первый прототип ThenByDescending

```

public static IOrderedEnumerable<T> ThenByDescending<T, K>(
    this IOrderedEnumerable<T> source,

```

```

    Func<T, K> keySelector)
    where
        K : IComparable<K>;

```

Данный прототип операции ведет себя точно так же, как и первый прототип операции ThenBy, за исключением того, что упорядочивает в обратном порядке.

ThenByDescending имеет и второй прототип, который выглядит так:

Второй прототип ThenByDescending

```

public static IOrderedEnumerable<T> ThenByDescending<T, K>(
    this IOrderedEnumerable<T> source,
    Func<T, K> keySelector,
    IComparer<K> comparer);

```

Этот прототип — такой же, как первый, кроме того, что принимает объект-компаратор. Если используется эта версия ThenByDescending, то нет необходимости в реализации интерфейса IComparable типов K.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

Для примера применения первого прототипа операции ThenByDescending я использую тот же базовый подход, что и в примере вызова первого прототипа операции ThenBy, за исключением того, что он вызовет ThenByDescending вместо ThenBy. В листинге 4.25 показан код.

Листинг 4.25. Пример вызова первого прототипа ThenByDescending

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items =
    presidents.OrderBy(s => s.Length).ThenByDescending(s => s);
foreach (string item in items)
    Console.WriteLine(item);

```

Этот код порождает вывод, где имена в пределах одной группы по длине сортируются по алфавиту в обратном порядке тому, который обеспечивала операция ThenBy:

```

Taft
Polk
Ford
Bush
Tyler
Nixon
Hayes
Grant
Adams
Wilson
Truman
Taylor
Reagan

```

```
Pierce
Monroe
Hoover
Carter
Arthur
Madison
Lincoln
Kennedy
Johnson
Jackson
Harding
Clinton
McKinley
Harrison
Garfield
Fillmore
Coolidge
Buchanan
Van Buren
Roosevelt
Jefferson
Cleveland
Washington
Eisenhower
```

Для примера вызова второго прототипа операции ThenByDescending, который показан в листинге 4.26, я использую тот же пример, что и для второго прототипа операции ThenBy, но вместо ThenBy вызову ThenByDescending.

Листинг 4.26. Пример вызова второго прототипа ThenByDescending

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
MyVowelToConsonantRatioComparer myComp = new MyVowelToConsonantRatioComparer();
IEnumerable<string> namesByVToCRatio = presidents
    .OrderBy(n => n.Length)
    .ThenByDescending((s => s), myComp);
foreach (string item in namesByVToCRatio)
{
    int vCount = 0;
    int cCount = 0;
    myComp.GetVowelConsonantCount(item, ref vCount, ref cCount);
    double dRatio = (double)vCount / (double)cCount;

    Console.WriteLine(item + " - " + dRatio + " - " + vCount + ":" + cCount);
}
```

Этот код породит следующий результат:

```
Bush - 0.3333333333333333 - 1:3
Ford - 0.3333333333333333 - 1:3
Polk - 0.3333333333333333 - 1:3
Taft - 0.3333333333333333 - 1:3
Hayes - 1.5 - 3:2
Adams - 0.6666666666666667 - 2:3
```

Nixon - 0.666666666666667 - 2:3
Tyler - 0.666666666666667 - 2:3
Grant - 0.25 - 1:4
Hoover - 1 - 3:3
Monroe - 1 - 3:3
Pierce - 1 - 3:3
Reagan - 1 - 3:3
Taylor - 1 - 3:3
Arthur - 0.5 - 2:4
Carter - 0.5 - 2:4
Truman - 0.5 - 2:4
Wilson - 0.5 - 2:4
Kennedy - 0.75 - 3:4
Madison - 0.75 - 3:4
Clinton - 0.4 - 2:5
Harding - 0.4 - 2:5
Jackson - 0.4 - 2:5
Johnson - 0.4 - 2:5
Lincoln - 0.4 - 2:5
Coolidge - 1 - 4:4
Buchanan - 0.6 - 3:5
Fillmore - 0.6 - 3:5
Garfield - 0.6 - 3:5
Harrison - 0.6 - 3:5
McKinley - 0.6 - 3:5
Roosevelt - 0.8 - 4:5
Cleveland - 0.5 - 3:6
Jefferson - 0.5 - 3:6
Van Buren - 0.5 - 3:6
Eisenhower - 1 - 5:5
Washington - 0.428571428571429 - 3:7

Как я и хотел, имена упорядочены сначала по длине, а потом — в обратном порядке по частоте гласных.

Reverse

Эта операция выводит последовательность того же типа, что и входная, но в обратном порядке.

Прототипы

Единственный прототип операции описан ниже.

Прототип Reverse

```
public static IEnumerable<T> Reverse<T>(
    this IEnumerable<T> source);
```

Эта операция возвращает объект, который при перечислении перебирает элементы входной последовательности и выдает их в выходную последовательность в обратном порядке.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

Пример вызова прототипа операции `Reverse` представлен в листинге 4.27.

Листинг 4.27. Пример вызова прототипа Reverse

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> items = presidents.Reverse();
foreach (string item in items)
    Console.WriteLine(item);

```

Если это работает правильно, я должен увидеть президентов в порядке, обратном массиву presidents. Вот результат работы этого кода:

```

Wilson
Washington
Van Buren
...
Bush
Buchanan
Arthur
Adams

```

Соединение

Операции соединения (*join*) связывают вместе несколько последовательностей.

Join

Операция Join выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей, извлеченных из каждого элемента этих последовательностей.

Прототипы

Операция Join имеет один прототип, описанный ниже.

Прототип Join

```

public static IEnumerable<V> Join<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, U, V> resultSelector);

```

Обратите внимание, что первый аргумент метода назван outer. Поскольку это расширяющий метод, последовательностью, на которой вызвана операция Join, будет последовательность outer.

Операция Join возвращает объект, который при перечислении сначала проходит последовательность inner с элементами типа U, вызывая метод innerKeySelector по одному разу для каждого элемента и сохраняя элемент, на который ссылается его ключ, в хеш-таблице. Затем он проходит последовательность outer с элементами типа T. По мере того, как возвращаемый объект перечисляет каждый объект последовательности outer, он вызывает метод outerKeySelector для получения ключа и извлекает соответствующий элемент последовательности inner из хеш-таблицы, используя этот ключ. Для каждой соответствующей пары элементов из последовательности outer и

`inner` возвращаемый объект вызывает метод `resultSelector`, передавая ему и элемент `outer`, и соответствующий ему элемент `inner`. Метод `resultSelector` вернет экземпляр объекта типа `V`, который возвращаемый объект поместить в выходную последовательность типа `V`.

Порядок элементов последовательности `outer` предохраняется, как и порядок элементов `inner` в пределах каждого элемента `outer`.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Для того чтобы продемонстрировать пример применения этой операции, вместо массива `presidents`, используемого в большинстве примеров, я обращусь к двум общим классам, определенным в начале этой главы — `Employee` и `EmployeeOptionEntry`.

Приведу пример вызова операции `Join` с использованием этих классов. Я построил код листинга 4.28 немного иначе, чем обычно, чтобы сделать каждый аргумент `Join` более читабельным.

Листинг 4.28. Пример кода, вызывающего операцию `Join`

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
    .Join(
        empOptions,           // последовательность inner
        e => e.id,           // outerKeySelector
        o => o.id,           // innerKeySelector
        (e, o) => new        // resultSelector
    {
        id = e.id,
        name = string.Format("{0} {1}", e.firstName, e.lastName),
        options = o.optionsCount
    });
foreach (var item in employeeOptions)
    Console.WriteLine(item);
```

В приведенном коде сначала я получаю пару массивов для соединения, используя два общих класса. Поскольку я вызываю операцию `Join` на массиве `employees`, он становится внешней последовательностью, а `empOptions` — внутренней последовательностью. Вот результаты операции `Join`:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что `resultSelector` создает анонимный класс в качестве типа элемента результирующей выходной последовательности. Вы можете убедиться в том, что это анонимный класс, поскольку при вызове `new` не указано имя класса. Поскольку тип анонимный, выходную последовательность необходимо сохранить в переменной,

чей тип специфицирован с использованием ключевого слова `var`. Вы не можете специфицировать его как `IEnumerable<T>`, поскольку нет именованного типа, который послужил бы параметром для объявления `IEnumerable`.

Совет. Когда последняя вызванная операция возвращает анонимный тип, вы должны использовать ключевое слово `var` для сохранения последовательности.

GroupJoin

Операция `GroupJoin` выполняет групповое соединение двух последовательностей на основе ключей, извлеченных из каждого элемента последовательностей.

Операция `GroupJoin` работает очень похоже на `Join`, за исключением того, что `Join` передает один элемент внешней последовательности с одним соответствующим элементом внутренней последовательности методу `resultSelector`. Это значит, что множество элементов внутренней последовательности соответствующих одному элементу внешней последовательности приведут в результате к множеству вызовов `resultSelector` для этого элемента внешней последовательности. С операцией `GroupJoin` все соответствующие элементы внутренней последовательности для определенного элемента внешней последовательности передаются в `resultSelector` как последовательность этого типа элемента, в результате чего метод `resultSelector` вызывается только по одному разу для каждого элемента внешней последовательности.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип GroupJoin

```
public static IEnumerable<V> GroupJoin<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, IEnumerable<U>, V> resultSelector);
```

Обратите внимание, что первый аргумент метода назван `outer`. Поскольку это расширяющий метод, последовательность, на которой вызывается операция `GroupJoin`, является внешней (`outer`).

Операция `GroupJoin` возвращает объект, который при перечислении сначала проходит по последовательности `inner` элементов типа `U`, вызывая метод `innerKeySelector` по одному разу для каждого элемента и сохраняя элемент, на который ссылается ключ, в хеш-таблице. Затем возвращенный объект выполняет перечисление последовательности `outer` элементов типа `T`. По мере того, как возвращаемый объект перечисляет каждый элемент последовательности `outer`, он вызывает метод `outerKeySelector` для получения его ключа и извлекает соответствующий элемент последовательности `inner` из хеш-таблицы по этому ключу. Для каждого элемента последовательности `outer` возвращаемый объект вызывает метод `resultSelector`, передавая ему элемент `outer` и последовательность соответствующих элементов `inner`, так что `resultSelector` может вернуть экземпляр объекта типа `V`, куда возвращаемый объект поместит выходную последовательность типа `V`.

Порядок элементов последовательности `outer` предохраняется, как и порядок элементов `inner` в пределах каждого элемента `outer`.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Для примера применения `GroupJoin` я применяю те же классы `Employee` и `EmployeeOptionEntry`, которые я использовал в примере `Join`. Мой код примера, который приведен в листинге 4.29, соединит сотрудников с опциями и вычислит сумму опций для каждого сотрудника, используя операцию `GroupJoin`.

Листинг 4.29. Пример кода, вызывающего операцию `GroupJoin`

```
Employee[] employees = Employee.GetEmployeesArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
    .GroupJoin(
        empOptions,
        e => e.id,
        o => o.id,
        (e, os) => new
    {
        id = e.id,
        name = string.Format("{0} {1}", e.firstName, e.lastName),
        options = os.Sum(o => o.optionsCount)
    });
foreach (var item in employeeOptions)
    Console.WriteLine(item);
```

Приведенный код почти идентичен примеру с операцией `Join`. Однако если вы просмотрите второй аргумент лямбда-выражения, переданного в качестве метода `resultSelector`, то заметите, что я назвал входной аргумент `o` в примере `Join`, но в этом примере назвал его `os`. Это потому, что в примере `Join` в этом аргументе передавался единственный объект опций — `o`, но в примере `GroupJoin` передается последовательность объектов опций сотрудника — `os`. Затем в последний член моего экземпляра анонимного объекта — `optionsCount`, устанавливается сумма последовательности объектов опций сотрудника с использованием операции `Sum`, которую я опишу в следующей главе (поскольку это не отложенная операция запроса). А пока вам достаточно понимать, что операция `Sum` имеет возможность вычислять сумму каждого элемента или члена каждого элемента из входной последовательности.

Этот код выдаст следующий результат:

```
{ id = 1, name = Joe Rattz, options = 2 }
{id = 2, name = William Gates, options = 30000 }
{id = 3, name = Anders Hejlsberg, options = 20000 }
{id = 4, name = David Lightman, options = 1500 }
{id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что в этом результате одна запись для каждого сотрудника содержит сумму всех записей об опциях данного сотрудника. Отличие от примера операции `Join` состоит в том, что там были отдельные записи для каждой из записей опции сотрудника.

Группировка

Группирующие операции помогают объединять вместе элементы последовательности по общему ключу.

GroupBy

Операция GroupBy используется для группировки элементов входной последовательности.

Прототипы

Все прототипы операции GroupBy возвращают последовательность элементов IGrouping<K, T>. IGrouping<K, T> — это интерфейс, определенный, как описано ниже.

Интерфейс IGrouping<K, T>

```
public interface IGrouping<K, T> : IEnumerable<T>
{
    K.Key { get; }
}
```

Таким образом, IGrouping — это последовательность типа T с ключом типа K. Существуют четыре прототипа, описанные ниже.

Первый прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
```

Этот прототип операции GroupBy возвращает объект, который при перечислении перебирает входную последовательность source, вызывая метод keySelector собирает каждый элемент с его ключом и порождает последовательность экземпляров IGrouping<K, E>, где каждый элемент IGrouping<K, E> представляет собой последовательность элементов с одинаковым значением ключа. Значе ключа сравниваются с использованием компаратора эквивалентности по умолчанию EqualityComparerDefault. Говоря иначе, возвращаемое значение метода GroupBy это последовательность объектов IGrouping, каждый из которых содержит ключ и последовательность элементов из входной последовательности, имеющих тот же ключ.

Порядок экземпляров IGrouping будет тем же, что и вхождения ключей в последовательности source, и каждый элемент в последовательности IGrouping будет расположен в том порядке, в котором элементы находились в последовательности source.

Второй прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
```

Этот прототип операции GroupBy такой же, как и первый, за исключением того, что вместо использования компаратора эквивалентности EqualityComparerDefault, вы указываете собственный.

Третий прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector);
```

Этот прототип операции GroupBy подобен первому, за исключением того, что вместо помещения всего элемента из входной последовательности в выходную последовательность IGrouping целиком вы можете специфицировать, какая часть входного элемента должна попасть в выход, используя для этого elementSelector.

Четвертый прототип GroupBy

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

Этот прототип операции GroupBy является комбинацией второго и третьего, так что вы можете специфицировать компаратор в аргументе comparer и выводить элементы типа, отличного от входных элементов, используя аргумент elementSelector.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null.

Примеры

Для примера применения первого прототипа GroupBy я использую общий класс EmployeeOptionEntry. В этом примере (листинг 4.30) я группирую записи EmployeeOptionEntry по id и отображаю их.

Листинг 4.30. Пример вызова первого прототипа GroupBy

```
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.
GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, EmployeeOptionEntry>> outerSequence =
    empOptions.GroupBy(o => o.id);
// Сначала перечисление по внешней последовательности IGroupings.
foreach (IGrouping<int, EmployeeOptionEntry> keyGroupSequence in outerSequence)
{
    Console.WriteLine("Option records for employee: " + keyGroupSequence.Key);
    // Теперь перечисление по группированной последовательности
    // элементов EmployeeOptionEntry.
    foreach (EmployeeOptionEntry element in keyGroupSequence)
        Console.WriteLine("id={0} : optionsCount={1} : dateAwarded={2:d}",
            element.id, element.optionsCount, element.dateAwarded);
}
```

Обратите внимание в приведенном коде, что я выполняю перечисление по внешней последовательности по имени outerSequence, где каждый элемент — это объект, реализующий IGrouping, который содержит ключ, и последовательность элементов EmployeeOptionEntry, имеющих одинаковые ключи.

Вот результат:

```
Option records for employee: 1
id=1 : optionsCount=2 : dateAwarded=12/31/1999
Option records for employee: 2
id=2 : optionsCount=10000 : dateAwarded=6/30/1992
id=2 : optionsCount=10000 : dateAwarded=1/1/1994
id=2 : optionsCount=10000 : dateAwarded=4/1/2003
Option records for employee: 3
id=3 : optionsCount=5000 : dateAwarded=9/30/1997
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
Option records for employee: 4
id=4 : optionsCount=1500 : dateAwarded=12/31/1997
Option records for employee: 101
id=101 : optionsCount=2 : dateAwarded=12/31/1999
```

Для примера второго прототипа GroupBy, давайте предположим, что каждый сотрудник, чей id меньше 100, является основателем компании. Те, у кого id равен 100 и больше, основателями не являются. Моя задача — вывести все записи опций, сгруппированные по статусу сотрудника. Все опции основателей будут сгруппированы вместе, а отдельно от них — опции сотрудников — не основателей.

Теперь мне понадобится компаратор эквивалентности, который сможет выполнить это ключевое сравнение для меня. Мой компаратор эквивалентности должен реализовать интерфейс `IEqualityComparer`. Прежде чем рассматривать сам компаратор, давайте сначала бросим взгляд на интерфейс.

Интерфейс `IEqualityComparer<T>`

```
interface IEqualityComparer<T> {
    bool Equals(T x, T y);
    int GetHashCode(T x);
}
```

Этот интерфейс требует реализации двух методов — `Equals` и `GetHashCode`. Метод `Equals` принимает два объекта одного типа `T` и возвращает `true`, если два объекта считаются эквивалентными, и `false` — в противном случае. Метод `GetHashCode` принимает единственный объект и возвращает хеш-код типа `int` этого объекта.

Хеш-код — это числовое значение, обычно вычисляемое математически на основе некоторой части данных объекта, известной как ключ, в целях уникальной идентификации объекта. Функция вычисляемого хеш-кода состоит в том, чтобы служить индексом в некоторой структуре данных для хранения объекта и последующего его нахождения. Поскольку допускается, что множество ключей производят один и тот же хеш-код, что делает его не уникальным, также есть необходимость в определении эквивалентности двух ключей. В этом предназначение метода `Equals`.

Вот мой класс, реализующий интерфейс `IEqualityComparer`.

Класс, реализующий интерфейс `IEqualityComparer` для второго примера применения `GroupBy`

```
public class MyFounderNumberComparer : IEqualityComparer<int>
{
    public bool Equals(int x, int y)
    {
        return (isFounder(x) == isFounder(y));
    }
    public int GetHashCode(int i)
    {
        int f = 1;
        int nf = 100;
        return (isFounder(i) ? f.GetHashCode() : nf.GetHashCode());
    }
    public bool isFounder(int id)
    {
        return (id < 100);
    }
}
```

В дополнение к методам, которые требует интерфейс, я добавил метод `isFounder` для определения того, является ли сотрудник основателем компании на основании приведенного выше критерия. Это сделает код немного понятнее. Я сделал этот метод общедоступным, чтобы его можно было вызывать извне интерфейса, что вы и увидите в моем примере.

Мой компаратор эквивалентности рассматривает любой целочисленный идентификатор сотрудника, который меньше 100, как признак основателя компании, и если два идентификатора указывают на то, что оба являются основателями, или оба не являются основателями, так что все основатели попадают в одну группу, а прочие — в другую.

Пример кода MyGroup приведен в листинге 4.31.

Листинг 4.31. Пример вызова второго прототипа GroupBy

```
MyFounderNumberComparer comp = new MyFounderNumberComparer();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, EmployeeOptionEntry>> opts = empOptions
    .GroupBy(o => o.id, comp);
// Сначала перечисление по последовательности IGroupings.
foreach (IGrouping<int, EmployeeOptionEntry> keyGroup in opts)
{
    Console.WriteLine("Option records for: " +
        (comp.isFounder(keyGroup.Key) ? "founder" : "non-founder"));

    // Теперь — перечисление группированной последовательности
    // элементов EmployeeOptionEntry.
    foreach (EmployeeOptionEntry element in keyGroup)
        Console.WriteLine("id={0} : optionsCount={1} : dateAwarded={(2:d)}",
            element.id, element.optionsCount, element.dateAwarded);
}
```

В этом примере я заблаговременно создаю экземпляр компаратора эквивалентности — в противоположность тому, чтобы делать это в вызове метода GroupBy, так что могу использовать его для вызова метода isFounder в цикле foreach.

Вот результат работы этого кода:

```
Option records for: founder
id=1 : optionsCount=2 : dateAwarded=12/31/1999
id=2 : optionsCount=10000 : dateAwarded=6/30/1992
id=2 : optionsCount=10000 : dateAwarded=1/1/1994
id=3 : optionsCount=5000 : dateAwarded=9/30/1997
id=2 : optionsCount=10000 : dateAwarded=4/1/2003
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
id=3 : optionsCount=7500 : dateAwarded=9/30/1998
id=4 : optionsCount=1500 : dateAwarded=12/31/1997
Option records for: non-founder
id=101 : optionsCount=2 : dateAwarded=12/31/1998
```

Как видите, все записи опций сотрудников, чьи id меньше 100, сгруппированы с основателями. Иначе они группируются с прочими (не основателями).

Для примера третьего прототипа GroupBy мы предположим, что нас интересуют даты назначения опций для каждого сотрудника. Этот код будет очень похож на пример первого прототипа.

Таким образом, в листинге 4.32 вместо возврата последовательности сгруппированных объектов EmployeeOptionEntry я буду группировать даты.

Листинг 4.32. Пример вызова третьего прототипа GroupBy

```
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.
GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, DateTime>> opts = empOptions
    .GroupBy(o => o.id, e => e.dateAwarded);
// Сначала перечисление по последовательности IGroupings.
foreach (IGrouping<int, DateTime> keyGroup in opts)
```

```

    Console.WriteLine("Option records for employee: " + keyGroup.Key);
    // Теперь – перечисление группированной последовательности элементов DateTime.
    foreach (DateTime date in keyGroup)
        Console.WriteLine(date.ToShortDateString());
}

```

Обратите внимание, что в вызове операции GroupBy второй аргумент element Selector просто возвращает член dateAwarded. Поскольку я возвращаю DateTime, мой IGrouping теперь служит для типа DateTime вместо EmployeeOptionEntry.

Как и можно было ожидать, теперь я имею даты назначений опций, сгруппированные по сотруднику:

```

Option records for employee: 1
12/31/1999
Option records for employee: 2
6/30/1992
1/1/1994
4/1/2003
Option records for employee: 3
9/30/1997
9/30/1998
9/30/1998
Option records for employee: 4
12/31/1997
Option records for employee: 101
12/31/1998

```

Для того чтобы продемонстрировать четвертый и последний прототип, мне нужно использовать метод elementSelector и объект comparer, поэтому я применяю комбинацию примеров для второго и третьего прототипов. Я хочу группировать даты назначения опций по тому, были они назначены основателям или нет, если основателями считаются сотрудники с id меньше 100. Код представлен в листинге 4.33.

Листинг 4.33. Пример вызова четвертого прототипа GroupBy

```

MyFounderNumberComparer comp = new MyFounderNumberComparer();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
IEnumerable<IGrouping<int, DateTime>> opts = empOptions
    .GroupBy(o => o.id, o => o.dateAwarded, comp);
// Сначала перечисление по последовательности IGroupings.
foreach (IGrouping<int, DateTime> keyGroup in opts)
{
    Console.WriteLine("Option records for: " +
        (comp.isFounder(keyGroup.Key) ? "founder" : "non-founder"));
    // Теперь – перечисление группированной последовательности
    // элементов EmployeeOptionEntry.
    foreach (DateTime date in keyGroup)
        Console.WriteLine(date.ToShortDateString());
}

```

В выводе мы должны увидеть просто даты, сгруппированные по основателям и прочим сотрудникам:

```

Option records for: founder
12/31/1999
6/30/1992
1/1/1994
9/30/1997

```

```

4/1/2003
9/30/1998
9/30/1998
12/31/1997
Option records for: non-founder
12/31/1998

```

Множества

Операции множеств используются для выполнения математических операций с множествами на последовательностях.

Совет. Свойства операций множеств, описанные в этой главе, не работают правильно с `DataSet`. В случае `DataSet` используйте прототипы, описанные в главе 10.

Distinct

Операция `Distinct` удаляет дублированные элементы из входной последовательности.

Прототипы

У операции `Distinct` есть один прототип, описанный ниже.

Прототип `Distinct`

```

public static IEnumerable<T> Distinct<T>(
    this IEnumerable<T> source);

```

Эта операция возвращает объект, перечисляющий элементы входной последовательности `source` и порождающий последовательность, в которой каждый элемент не эквивалентен предыдущему. Эквивалентность элементов определяется методами `GetHashCode` и `Equals`. Случайно ли я только что сказал, где и как используются методы `GetHashCode` и `Equals`?

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

В этом примере я сначала отображаю количество элементов массива `presidents`, затем соединяю массив `presidents` с самим собой, отображаю число элементов результирующей соединенной последовательности, затем вызываю операцию `Distinct` на этой объединенной последовательности и, наконец, отображаю количество элементов последовательности, полученной от `Distinct`, которая должна совпасть с начальным массивом `presidents`.

Чтобы определить длину двух сгенерированных последовательностей, я использую стандартную операцию запроса `Count`. Поскольку это не отложенная операция, я не стану описывать его в этой главе. О нем, однако, будет рассказано в следующей главе. А пока просто знайте, что он возвращает счетчик элементов в последовательности, на которой он вызван.

Код примера приведен в листинге 4.34.

Листинг 4.34. Пример вызова операции `Distinct`

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
}

```

```

    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
    // Отобразить счетчик элементов массива presidents.
    // Console.WriteLine("presidents count: " + presidents.Count());
    // Соединить presidents с самим собой. Теперь каждый элемент
    // представлен в последовательности дважды.
    IEnumerable<string> presidentsWithDuplicates = presidents.Concat(presidents);
    // Отобразить счетчик элементов объединенной последовательности.
    // Console.WriteLine("presidentsWithDuplicates count: " + presidentsWithDuplicates.Count());
    // Исключить дубликаты и отобразить счетчик.
    IEnumerable<string> presidentsDistinct = presidentsWithDuplicates.Distinct();
    // Console.WriteLine("presidentsDistinct count: " + presidentsDistinct.Count());

```

Если это работает, как я ожидаю, то количество элементов в последовательности presidentsDistinct будет эквивалентно количеству элементов в последовательности presidents. Говорит ли результат об успехе?

```

presidents count: 37
presidentsWithDuplicates count: 74
presidentsDistinct count: 37

```

Конечно, да!

Union

Операция Union возвращает последовательность объединения множеств из двух исходных последовательностей.

Прототипы

У этой операции имеется один прототип, описанный ниже.

Прототип Union

```

public static IEnumerable<T> Union<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);

```

Эта операция возвращает объект, который сначала перечисляет элементы последовательности по имени first, порождая последовательность, в которой каждый элемент не эквивалентен предыдущему, затем перечисляет вторую входную последовательность second, опять-таки, порождая последовательность без повторений. Эквивалентность элементов определяется методами GetHashCode и Equals.

Исключения

Исключение ArgumentNullException генерируется, если любой аргумент равен null.

Примеры

Чтобы продемонстрировать разницу между операцией Union и операцией Concat, описанной выше, в примере, представленном в листинге 4.35, я создам последовательности first и second из моего массива presidents, что приведет к тому, что пятый элемент будет дублирован в обеих последовательностях. Затем я отобразжу счетчик массива presidents, а также последовательностей first и second, наряду со счетчиком склеенной (concatenated) и объединенной (union) последовательности.

Листинг 4.35. Пример вызова операции Union

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> first = presidents.Take(5);
IEnumerable<string> second = presidents.Skip(4);
// Поскольку я пропустил 4 элемента, пятый элемент
// должен присутствовать в обеих последовательностях.
IEnumerable<string> concat = first.Concat<string>(second);
IEnumerable<string> union = first.Union<string>(second);
Console.WriteLine("The count of the presidents array is: " + presidents.
Count());
Console.WriteLine("The count of the first sequence is: " + first.Count());
Console.WriteLine("The count of the second sequence is: " + second.Count());
Console.WriteLine("The count of the concat sequence is: " + concat.Count());
Console.WriteLine("The count of the union sequence is: " + union.Count());

```

Если это отработает правильно, то последовательность concat должна будет иметь на один элемент больше, чем массив presidents. Последовательность union должна содержать то же количество элементов, что и массив presidents. Доказательство — в результате выполнения кода:

```

The count of the presidents array is: 37
The count of the first sequence is: 5
The count of the second sequence is: 33
The count of the concat sequence is: 38
The count of the union sequence is: 37

```

Порядок!

Intersect

Операция Intersect возвращает пересечение множеств двух последовательностей.

Прототипы

Операция Intersect имеет один прототип, описанный ниже.

Прототип Intersect

```

public static IEnumerable<T> Intersect<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);

```

Эта операция возвращает объект, который сначала перечисляет элементы последовательности по имени first, выбирая оттуда каждый элемент, который не эквивалентен предыдущему выбранному элементу. Затем он перечисляет вторую входную последовательность, помечая любой элемент, имеющийся в обеих последовательностях для включения в выходную последовательность. Затем осуществляется проход по помеченным элементам, с помещением их в выходную последовательность в том порядке, в котором они были собраны. Эквивалентность элементов определяется с помощью методов GetHashCode и Equals.

Исключения

Исключение `ArgumentNullException` генерируется, если любой аргумент равен `null`.

Примеры

Чтобы продемонстрировать применение операции `Intersect`, в листинге 4.36 я использую операции `Take` и `Skip` для генерации двух последовательностей и получения некоторого их перекрытия, как в примере с `Union`, где я намеренно дублировал пятый элемент. Когда я вызываю операцию `Intersect` на этих двух сгенерированных последовательностях, в возвращаемой последовательности `intersect` должен оказаться только дублированный пятый элемент. Я отобразжу счетчики элементов массива `presidents` и всех последовательностей. И, наконец, пройдусь по последовательности `intersect`, отображая каждый элемент. В ней должен оказаться только один пятый элемент массива `presidents`.

Листинг 4.36. Пример вызова операции `Intersect`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
IEnumerable<string> first = presidents.Take(5);
IEnumerable<string> second = presidents.Skip(4);
// Поскольку я пропустил 4 элемента, пятый элемент
// должен присутствовать в обеих последовательностях.
IEnumerable<string> intersect = first.Intersect(second);
Console.WriteLine("The count of the presidents array is: " + presidents.Count());
Console.WriteLine("The count of the first sequence is: " + first.Count());
Console.WriteLine("The count of the second sequence is: " + second.Count());
Console.WriteLine("The count of the intersect sequence is: " + intersect.Count());
// Просто ради интереса я отобразжу последовательность intersection,
// которая должна состоять только из одного пятого элемента.
foreach (string name in intersect)
    Console.WriteLine(name);
```

Если это будет работать, как должно, я должен получить последовательность `intersect`, которая состоит только из одного элемента, содержащего дублированный пятый элемент массива `presidents`, а именно — "Carter":

```
The count of the presidents array is: 37
The count of the first sequence is: 5
The count of the second sequence is: 33
The count of the intersect sequence is: 1
Carter
```

LINQ работает! Насколько часто вам приходилось выполнять операции над множествами элементов из двух коллекций? Было ли это трудно? Благодаря LINQ, эти дни в прошлом.

Except

Операция `Except` возвращает последовательность, содержащую все элементы первой последовательности, которых нет во второй последовательности.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип Except

```
public static IEnumerable<T> Except<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

Эта операция возвращает объект, который при перечислении просматривает первую последовательность `first`, накапливая элементы, не эквивалентные предыдущему накопленному элементу. Затем он перечисляет входную последовательность `second`, удаляя из коллекции, собранной при проходе `first` те элементы, которые принадлежат обеим последовательностям. Затем выполняется перечисление оставшихся элементов с выводом их в том порядке, в котором они были набраны. Эквивалентность одного элемента другому определяется с использованием их методов `GetHashCode` и `Equals`.

Исключения

Исключение `ArgumentNullException` генерируется, если любой аргумент равен `null`.

Примеры

В этом примере я использую массив `presidents`, который уже применялся в большинстве примеров. Представьте себе сценарий, когда у вас есть первичный источник данных — массив `presidents`, с вхождениями, которые вы хотите подвергнуть какой-то обработке. По мере завершения обработки каждого вхождения, вы хотите добавлять его в коллекцию обработанных — так, чтобы если начать обработку снова, то можно было бы использовать операцию `Except` для производства последовательности исключенных элементов данных из первичной последовательности, чтобы не обрабатывать их повторно.

Для этого примера, представленного в листинге 4.37, я представил, что уже обработал первые четыре вхождений. Чтобы получить последовательность, содержащую первые четыре элемента массива `presidents`, я просто выполню на нем операцию `Take`.

Листинг 4.37. Пример вызова операции Intersect

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
// Сначала сгенерировать последовательность обработанных.
IEnumerable<string> processed = presidents.Take(4);
IEnumerable<string> exceptions = presidents.Except(processed);
foreach (string name in exceptions)
    Console.WriteLine(name);
```

В этом примере полученный результат будет содержать имена из массива `presidents`, расположенные после четвертого элемента — "Bush":

```
Carter
Cleveland
Clinton
Coolidge
```

```
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Работает, как следовало ожидать.

Преобразование

Операции преобразования представляют простой и удобный способ преобразования последовательностей в другие типы коллекций.

Cast

Операция Cast используется для приведения каждого элемента входной последовательности в выходную последовательность указанного типа.

Прототипы

Операция Cast имеет один прототип, описанный ниже.

Прототип Cast

```
public static IEnumerable<T> Cast<T>(
    this IEnumerable source);
```

Первое, что вы должны отметить в операции Cast — это то, что его первый аргумент по имени source имеет тип IEnumerable, а не IEnumerable<T>, в то время как большинство отложенных стандартных операций запросов принимают первый аргумент типа IEnumerable<T>. Это связано с тем, что операция Cast предназначена для вызова на классах, реализующих интерфейс IEnumerable, а не IEnumerable<T>. В частности, мы говорим об унаследованных коллекциях C#, разработанных до появления обобщений в C# 2.0.

Вы можете вызывать операцию Cast на унаследованных коллекциях C# до тех пор, пока они реализуют IEnumerable, и при этом будет создана выходная последователь-

ность `IEnumerable<T>`. Поскольку большинство стандартных операций запросов работают на последовательностях типа `IEnumerable<T>`, вы должны вызывать какой-то метод вроде этого, или, возможно, операцию `OfType`, о которой я расскажу ниже, чтобы преобразовать унаследованную коллекцию в последовательность, на которой можно вызывать стандартные операции запросов.

Эта операция вернет объект, который при перечислении проходит по исходной коллекции данных, преобразуя каждый элемент к типу `T`. Если элемент не может быть преобразован к типу `T`, генерируется исключение. Из-за этого данная операция должна вызываться только когда точно известно, что каждый элемент входной последовательности может быть преобразован к типу `T`.

Совет. Пытаясь выполнять запросы LINQ на унаследованных коллекциях, не забудьте вызывать `Cast` или `OfType` на такой коллекции, чтобы создать последовательность `IEnumerable<T>`, на которой можно вызывать затем стандартные операции запросов.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`. Исключение `InvalidOperationException` генерируется, если элемент входной коллекции не может быть приведен к типу `T`.

Примеры

Для этого примера я использую метод `GetEmployeesArrayList` моего общего класса `Employee`, чтобы вернуть унаследованный, необобщенный `ArrayList`.

В листинге 4.38 содержится код, демонстрирующий, как тип данных элементов `ArrayList` приводится к элементам последовательности `IEnumerable<T>`.

Листинг 4.38. Код, преобразующий `ArrayList` в `IEnumerable<T>`, который может быть использован с типичными стандартными операциями запросов

```
ArrayList employees = Employee.GetEmployeesArrayList();
Console.WriteLine("The data type of employees is " + employees.GetType());
var seq = employees.Cast<Employee>();
Console.WriteLine("The data type of seq is " + seq.GetType());
var emps = seq.OrderBy(e => e.lastName);
foreach (Employee emp in emps)
    Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Сначала язываю метод `GetEmployeesArrayList` для возврата `ArrayList` объектов `Employee` и затем отображаю тип данных переменной `employees`. Далее я преобразую этот `ArrayList` в последовательность `IEnumerable<T>`, вызывая для этого операцию `Cast`, после чего отображаю тип данных возвращенной последовательности. И, наконец, я перечисляю элементы возвращенной последовательности, чтобы доказать, что все это на самом деле работает.

Вот вывод этого кода:

```
The data type of employees is System.Collections.ArrayList
The data type of seq is
System.Linq.Enumerable+<CastIterator>d__b0`1[LINQChapter4.Employee]
Kevin Flynn
William Gates
Anders Hejlsberg
David Lightman
Joe Rattz
```

Вы можете видеть, что типом данных переменной `employees` является `ArrayList`. Немного сложнее определить тип данных `seq`. Мы определенно можем видеть, что он отличается и выглядит как последовательность. Также мы можем видеть присутствие слова `CastIterator` в его типе. Заметили ли вы, что когда я говорил об отложенных операциях, которые не возвращают немедленно выходной последовательности, а возвращают объект, перечисление по которому должно породить элементы выходной последовательности? Тип данных переменной `seq`, отображенный в предыдущем примере, указывает именно на объект такого рода. Однако это — деталь реализации, которая может измениться.

Внимание! Операция `Cast` попытается привести каждый элемент входной последовательности к указанному типу. Если любой из этих элементов не может быть приведен к указанному типу, будет сгенерировано исключение `InvalidOperationException`. Если существует вероятность присутствия разнотипных элементов в исходной коллекции, применяйте вместо `Cast` операцию `OfType`.

OfType

Операция `OfType` используется для построения выходной последовательности, содержащей только те элементы, которые могут быть успешно преобразованы к специфицированному типу.

Прототипы

Эта операция имеет один прототип, описанный ниже.

Прототип `OfType`

```
public static IEnumerable<T> OfType<T>(
    this IEnumerable source);
```

Первое, что вы должны заметить в операции `OfType` — это то, что его первый аргумент по имени `source`, как и у операции `Cast`, имеет тип `IEnumerable`, а не `IEnumerable<T>`. Большинство первых аргументов отложенных стандартных операций запросов имеют тип `IEnumerable<T>`. Это объясняется тем, что операция `OfType` предназначена для вызова на классах, реализующих интерфейс `IEnumerable`, а не `IEnumerable<T>`. В частности, мы говорим обо всех унаследованных коллекциях C#, разработанных до появления C# и обобщений.

Итак, вы можете вызывать операцию `OfType` на унаследованной коллекции C# до тех пор, пока она реализует `IEnumerable`, и при этом будет создана выходная последовательность `IEnumerable<T>`. Поскольку большинство стандартных операций запросов работает только на последовательностях типа `IEnumerable<T>`, вы должны вызывать некоторый метод, подобный этому, или, возможно, операцию `Cast`, чтобы преобразовать унаследованную коллекцию в последовательность, на которой могут быть вызваны стандартные операции запросов. Это важно, когда вы пытаетесь использовать стандартные операции запросов на унаследованных коллекциях.

Операция `OfType` вернет объект, который при перечислении проходит по входной последовательности, порождая только те элементы, чей тип преобразуется к указанному типу `T`.

Операция `OfType` отличается от `Cast` тем, что `Cast` пытается привести каждый элемент входной последовательности к типу `T` и вставить его в выходную последовательность. Если приведение не удается, генерируется исключение. Операция `OfType` пытается вставить входной элемент только в том случае, если он может привести его к типу `T`. Технически для элемента должно быть истинно `e is T` для того, чтобы он был вставлен в выходную последовательность.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

Для примера в листинге 4.39 я создаю `ArrayList`, содержащий объекты двух моих общих классов — `Employee` и `EmployeeOptionEntry`. Наполнив `ArrayList` объектами обоих классов, сначала я вызываю операцию `Cast`, чтобы показать, как он терпит неудачу в данной ситуации. Затем выполняю вызов операции `OfType`, демонстрируя на что она способна в той же ситуации.

Листинг 4.39. Пример кода вызова операций Cast и OfType

```
ArrayList al = new ArrayList();
al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
al.Add(new EmployeeOptionEntry { id = 1, optionsCount = 0 });
al.Add(new EmployeeOptionEntry { id = 2, optionsCount = 999999999999 });
al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
al.Add(new EmployeeOptionEntry { id = 3, optionsCount = 848475745 });
var items = al.Cast<Employee>();
Console.WriteLine("Attempting to use the Cast operator ...");
try
{
    foreach (Employee item in items)
        Console.WriteLine("{0} {1} {2}", item.id, item.firstName, item.lastName);
}
catch (Exception ex)
{
    Console.WriteLine("{0}{1}", ex.Message, System.Environment.NewLine);
}
Console.WriteLine("Attempting to use the OfType operator ...");
var items2 = al.OfType<Employee>();
foreach (Employee item in items2)
    Console.WriteLine("{0} {1} {2}", item.id, item.firstName, item.lastName);
```

Создав и наполнив `ArrayList` элементами, язываю операцию `Cast`. Следующий шаг — попытаться перечислить его. Этот шаг необходим, поскольку операция `Cast` является отложенной. Если никогда не перечислять результаты этого запроса, он никогда не будет выполнен, и проблема не обнаружится. Обратите внимание, что я заключил цикл `foreach`, который перечисляет результаты запроса, в блок `try/catch`. В данном случае это необходимо, поскольку я знаю, что возникнет исключение по причине наличия объектов двух разных типов. Затем язываю операцию `OfType` и выполняю перечисление с отображением его результатов. Обратите внимание, что этот цикл `foreach` я решил не заключать в блок `try/catch`. Разумеется, в реальном рабочем коде вы не должны игнорировать защиту, которую предоставляет блок `try/catch`.

Вот результат этого запроса:

```
Attempting to use the Cast operator ...
1 Joe Rattz
2 William Gates
Unable to cast object of type 'LINQChapter4.EmployeeOptionEntry' to type
'LINQChapter4.Employee'.
Attempting to use the OfType operator ...
1 Joe Rattz
2 William Gates
3 Anders Hejlsberg
```

Обратите внимание, что мне не удалось полностью выполнить перечисление результата запроса операции `Cast`, без генерации исключения. Но это удалось при опросе результата операции `OfType`, но в данном случае только элементы типа `Employee` были включены в выходную последовательность.

Совет. Если вы пытаетесь преобразовать необобщенную коллекцию, такую как один из унаследованных классов коллекций, в тип `IEnumerable<T>`, который может быть использован со стандартными операциями запросов, применяйте операцию `OfType` вместо `Cast`, если существует возможность того, что входная коллекция содержит объекты разных типов.

AsEnumerable

Операция `AsEnumerable` просто заставляет входную последовательность типа `IEnumerable<T>` быть возвращенной, как тип `IEnumerable<T>`.

Прототипы

Операция `AsEnumerable` имеет один прототип, описанный ниже:

Прототип AsEnumerable

```
public static IEnumerable<T> AsEnumerable<T>(
    this IEnumerable<T> source);
```

Приведенный прототип объявляет, что операция `AsEnumerable` работает над входным `IEnumerable<T>` по имени `source` и возвращает ту же последовательность, типизированную как `IEnumerable<T>`. Он служит ни чему иному, кроме как изменению типа выходной последовательности во время компиляции.

Это может показаться излишним, поскольку операция уже вызывается на типе `IEnumerable<T>`. Вы можете спросить: "Зачем может понадобиться конвертировать последовательность `IEnumerable<T>` в последовательность типа `IEnumerable<T>`?". Хороший вопрос!

Стандартные операции запросов объявлены для работы с нормальными последовательностями LINQ to Objects — коллекциями, реализующими интерфейс `IEnumerable<T>`. Однако другие специфичные для предметной области коллекции, такие как обращающиеся к базе данных, могут реализовывать свои собственные типы последовательностей со своими операциями. Обычно при вызове операции запроса на коллекции одного из таких типов должны вызываться операции, специфичные для данного типа. Операция `AsEnumerable` позволяет привести входную коллекцию к нормальному типу последовательности `IEnumerable<T>`, позволяя вызывать на ней методы стандартных операций запросов.

Например, когда я буду говорить о LINQ to SQL в остальной части книги, вы увидите, что LINQ to SQL в действительности использует свой собственный тип последовательности — `IQueryable<T>`, и реализует собственные операции. Когда вы вызываете метод `Where` на последовательности типа `IQueryable<T>`, то вызывается метод `Where` из LINQ to SQL, а не стандартная операция запроса `Where` из LINQ to Objects. Если вы попытаетесь вызвать одну из стандартных операций запросов, то получите исключение, если только не окажется одноименной операции LINQ to SQL. Операцией `AsEnumerable` вы можете выполнить приведение последовательность `IQueryable<T>` к последовательности `IEnumerable<T>`, тем самым позволив вызывать на ней стандартные операции запросов. Это бывает очень удобно, когда нужно контролировать, в каком API вызывается операция.

Исключения

Исключения нет.

Пример

Чтобы лучше понять эту операцию, необходима ситуация, когда реализована специфичная для предметной области операция. Для этого мне понадобится пример LINQ to SQL. Я начну с первого примера LINQ to SQL, приведенного в главе 1. Чтобы напомнить вам, приведу этот пример еще раз.

Повтор листинга 1.3 (для удобства)

```
using System;
using System.Linq;
using System.Data.Linq;
using nwind;
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
    from c in db.Customers
    where c.City == "Rio de Janeiro"
    select c;
foreach (var cust in custs)
    Console.WriteLine("{0}", cust.CompanyName);
```

Вот результат запуска этого примера:

```
Hanari Carnes
Que Delicia
Ricardo Adocicados
```

Чтобы пример работал, вы должны добавить в проект ссылку на сборку System.Data.Linq.dll, добавить директиву using для пространства имен nwind, а также сгенерированные классы, о которых будет сказано в главах, посвященных LINQ to SQL. Вдобавок вам может понадобиться подкорректировать строку соединения.

Предположим, что по какой-то причине мне нужно изменить порядок записей, поступивших из базы данных. Я не волнуюсь, поскольку знаю, что есть операция Reverse, описанная ранее в данной главе. В листинге 4.40 показан модифицированный предыдущий пример для вызова операции Reverse.

Листинг 4.40. Вызов операции Reverse

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
    (from c in db.Customers
    where c.City == "Rio de Janeiro"
    select c)
    .Reverse();
foreach (var cust in custs)
    Console.WriteLine("{0}", cust.CompanyName);
```

Выглядит довольно просто. Как видите, единственное изменение состоит в том, что я добавил вызов метода Reverse. Код нормально компилируется. А вот результат его запуска:

Unhandled Exception: System.NotSupportedException: The query operator 'Reverse' is not supported.

Необработанное исключение: System.NotSupportedException: Операция запроса 'Reverse' не поддерживается.

...

Все казалось так просто, что же случилось? А случилось то, что для интерфейса `IQueryable<T>` нет метода `Reverse`, потому и было сгенерировано исключение. Я должен использовать метод `AsEnumerable` для преобразования последовательности типа `IQueryable<T>` в последовательность типа `IEnumerable<T>`, чтобы при вызове метода `Reverse` был вызван метод `IEnumerable<T>.Reverse()`. Соответствующим образом модифицированный код представлен в листинге 4.41.

Листинг 4.41. Вызов операции `AsEnumerable` перед вызовом `Reverse`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs =
    (from c in db.Customers
     where c.City == "Rio de Janeiro"
     select c)
    .AsEnumerable()
    .Reverse();
foreach (var cust in custs)
    Console.WriteLine("{0}", cust.CompanyName);
```

Теперь я сначалазываю метод `AsEnumerable`, затем операцию `Reverse`, так что будет вызван операция `Reverse` из LINQ to Objects. И вот результат:

```
Ricardo Adocicados
Que Delicia
Hanari Carnes
```

Результирующий список представляет элементы из начального примера в обратном порядке, значит, все работает, как надо.

Элемент

Операции элементов позволяют вам извлекать элементы из входной последовательности.

`DefaultIfEmpty`

Операция `DefaultIfEmpty` возвращает последовательность, содержащую элемент по умолчанию, если входная последовательность пуста.

Прототипы

У операции `DefaultIfEmpty` есть два прототипа, которые описаны ниже.

Первый прототип `DefaultIfEmpty`

```
public static IEnumerable<T> DefaultIfEmpty<T>(
    this IEnumerable<T> source);
```

Этот прототип операции `DefaultIfEmpty` возвращает объект, который при перечислении входной последовательности `source` порождает каждый ее элемент, если только последовательность не окажется пустой — тогда возвращается последовательность из одного элемента `default(T)`. Для ссылочных и допускающих `null` типов значением по умолчанию является `null`.

Обратите внимание, что в отличие от всех прочих операций типа элементов, `DefaultIfEmpty` возвращает последовательность типа `IEnumerable<T>` вместо самого типа `T`. Существуют еще дополнительные операции типа элементов, но они не включены в эту главу, поскольку не являются отложенными.

Второй прототип позволяет специфицировать значение по умолчанию.

Второй прототип DefaultIfEmpty

```
public static IEnumerable<T> DefaultIfEmpty<T>(
    this IEnumerable<T> source,
    T defaultValue);
```

Эта операция полезна для всех других операций, которые генерируют исключения в случае пустой входной последовательности. В добавок эта операция удобна в сочетании с операцией GroupJoin для производства левосторонних внешних соединений (left outer joins).

Исключения

Исключение ArgumentNullException генерируется, если аргумент source равен null.

Примеры

В листинге 4.42 показан пример прототипа DefaultEmpty с пустой последовательностью. В данном примере не применяется операция DefaultEmpty, чтобы посмотреть, что происходит. Будет производиться поиск в массиве presidents имени Jones, возвращаться первый элемент, и если он не будет равен null, выдаваться сообщение.

**Листинг 4.42. Первый пример вызова первого прототипа DefaultEmpty
без использования DefaultEmpty**

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string jones = presidents.Where(n => n.Equals("Jones")).First();
if (jones != null)
    Console.WriteLine("Jones was found");
else
    Console.WriteLine("Jones was not found");
```

Вот результат:

Unhandled Exception: System.InvalidOperationException: Sequence contains no elements
Необработанное исключение: System.InvalidOperationException: Последовательность
не содержит элементов

...

В приведенном коде запрос не нашел никакого элемента, эквивалентного Jones, поэтому пустая последовательность была передана операции First. Операции First не нравятся пустые последовательности, поэтому она генерирует исключение.

Теперь, в листинге 4.43 я вызову тот же код, но на этот раз вставлю операцию DefaultEmpty между операциями Where и First. Таким образом, вместо пустой последовательности операции First будет передана последовательность, содержащая элемент null.

**Листинг 4.43. Второй пример вызова первого прототипа DefaultEmpty
с использованием DefaultEmpty**

```
string[] presidents = {"Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

128 Часть II. LINQ to Objects

```
string jones = presidents.Where(n => n.Equals("Jones")).DefaultIfEmpty().First();
if (jones != null)
    Console.WriteLine("Jones was found");
else
    Console.WriteLine("Jones was not found");
```

Теперь результат выглядит так:

```
Jones was not found.
```

Для примера второго прототипа я позволил специфицировать значение по умолчанию для пустой последовательности, как показано в листинге 4.44.

Листинг 4.44. Пример вызова второго прототипа DefaultEmpty

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name =
    presidents.Where(n => n.Equals("Jones")).DefaultEmpty("Missing").First();
Console.WriteLine(name);
```

Результат будет таким:

```
Missing
```

Далее, для последней группы примеров я выполню левое внешнее соединение, используя как операцию GroupJoin, так и DefaultEmpty. Также я использую два моих общих класса — Employee и EmployeeOptionEntry. В листинге 4.45 представлен пример без применения операции DefaultEmpty.

Листинг 4.45. Пример без использования операции DefaultEmpty

```
ArrayList employeesAL = Employee.GetEmployeesArrayList();
// Добавить нового сотрудника, не имеющего записей EmployeeOptionEntry.
employeesAL.Add(new Employee {
    id = 102,
    firstName = "Michael",
    lastName = "Bolton" });
Employee[] employees = employeesAL.Cast<Employee>().ToArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
    .GroupJoin(
        empOptions,
        e => e.id,
        o => o.id,
        (e, os) => os
            .Select(o => new
            {
                id = e.id,
                name = string.Format("{0} {1}", e.firstName, e.lastName),
                options = o != null ? o.optionsCount : 0
            }));
.SelectMany(r => r);
foreach (var item in employeeOptions)
    Console.WriteLine(item);
```

Есть три момента, которые я хочу отметить в этом примере. Во-первых, он очень похож на пример, который я представил в примере использования операции GroupJoin, когда речь шла о нем. Во-вторых, поскольку мой общий класс EmployeeOptionEntry уже имел соответствующий объект для каждого сотрудника в общем классе Employee, я получаю ArrayList сотрудников и добавляю в него нового сотрудника по имени Michael Bolton, так что он будет иметь одного сотрудника, для которого нет соответствующих объектов EmployeeOptionEntry. В-третьих, я не вызываю операцию DefaultEmpty в этом примере. Результаты запроса таковы:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
```

Обратите внимание, что поскольку для сотрудника по имени Michael Bolton не было соответствующих объектов в массиве EmployeeOptionEntry, я могу предоставить соответствующую запись по умолчанию, как показано в листинге 4.46.

Листинг 4.46. Пример с использованием операции DefaultEmpty

```
ArrayList employeesAL = Employee.GetEmployeesArrayList();
// Добавить нового сотрудника, не имеющего записей EmployeeOptionEntry.
employeesAL.Add(new Employee {
    id = 102,
    firstName = "Michael",
    lastName = "Bolton" });

Employee[] employees = employeesAL.Cast<Employee>().ToArray();
EmployeeOptionEntry[] empOptions = EmployeeOptionEntry.GetEmployeeOptionEntries();
var employeeOptions = employees
    .GroupJoin(
        empOptions,
        e => e.id,
        o => o.id,
        (e, os) => os
            .DefaultIfEmpty()
            .Select(o => new
            {
                id = e.id,
                name = string.Format("{0} {1}", e.firstName, e.lastName),
                options = o != null ? o.optionsCount : 0
            }));
    .SelectMany(r => r);
foreach (var item in employeeOptions)
    Console.WriteLine(item);
```

В приведенном примере я также добавляю объект сотрудника Michael Bolton, без соответствующих объектов EmployeeOptionEntry. Но теперь я вызываю операцию DefaultEmpty. Вот результат моего левого внешнего соединения:

```
{ id = 1, name = Joe Rattz, options = 2 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 2, name = William Gates, options = 10000 }
{ id = 3, name = Anders Hejlsberg, options = 5000 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
{ id = 3, name = Anders Hejlsberg, options = 7500 }
```

```
{ id = 4, name = David Lightman, options = 1500 }
{ id = 101, name = Kevin Flynn, options = 2 }
{ id = 102, name = Michael Bolton, options = 0 }
```

Как видите, теперь я вижу запись Michael Bolton, даже несмотря на отсутствие соответствующих объектов EmployeeOptionEntry. Из результатов вы можете видеть, что Michael Bolton не получил никаких опций. В самом деле, неудивительно, что он оказался в конце списка.

Генерация

Операции генерации помогают в генерации последовательностей.

Range

Операция Range генерирует последовательность целых чисел.

Прототипы

У операции Range есть один прототип, описанный ниже.

Прототип Range

```
public static IEnumerable<int> Range(
    int start,
    int count);
```

Последовательность целых чисел будет сгенерирована, начиная со значения, переданного, как start и длительностью count.

Обратите внимание, что это не расширяющий метод, и это один из нескольких стандартных операций запросов, которые не расширяют `IEnumerable<T>`.

На заметку! Range — не расширяющий метод. Это статический метод класса `System.Linq.Enumerable`.

Исключения

Исключение `ArgumentOutOfRangeException` генерируется, если count меньше нуля, или start плюс count минус один больше `int.MaxValue`.

Примеры

Листинг 4.47. Пример вызова операции Range

```
IEnumerable<int> ints = Enumerable.Range(1, 10);
foreach(int i in ints)
    Console.WriteLine(i);
```

Опять-таки, я хочу подчеркнуть, что я не вызываю операцию Range на последовательности. Это статический метод класса `System.Linq.Enumerable`. И вывод доказывает отсутствие каких-либо сюрпризов:

```
1
2
3
4
5
6
7
8
9
10
```

Repeat

Операция **Repeat** генерирует последовательность, повторяя указанный элемент заданное количество раз.

Прототипы

У операции **Repeat** есть один прототип, описанный ниже.

Прототип Repeat

```
public static IEnumerable<int> Repeat(
    T element,
    int count);
```

Этот прототип возвращает объект, который, будучи перечисленным, породит `count` экземпляров элемента `T`.

Обратите внимание, что он не является расширяющим методом, а является одним из нескольких стандартных операций запросов, которые не расширяют `IEnumerable<T>`.

На заметку! `Repeat` — не расширяющий метод. Это статический метод класса `System.Linq.Enumerable`.

Исключения

Исключение `ArgumentOutOfRangeException` генерируется, если `count` меньше нуля.

Примеры

В листинге 4.48 генерируется последовательность из десяти элементов, каждый из которых является числом 2.

Листинг 4.48. Возврат последовательности из десяти целых чисел, каждое из которых равно 2

```
IEnumerable<int> ints = Enumerable.Repeat(2, 10);
foreach(int i in ints)
    Console.WriteLine(i);
```

Вот результат работы этого примера:

```
2
2
2
2
2
2
2
2
2
2
```

Empty

Операция **Empty** генерирует пустую последовательность заданного типа.

Прототипы

Операция **Empty** имеет один прототип, представленный ниже.

Прототип Empty

```
public static IEnumerable<T> Empty<T>();
```

Этот прототип возвращает объект, при перечислении возвращающий последовательность из нуля элементов типа T.

Обратите внимание, что это не расширяющий метод, а одна из немногих стандартных операций запросов, которая не расширяет `IEnumerable<T>`.

На заметку! `Empty` — не расширяющий метод. Это статический метод класса `System.Linq.Enumerable`.

Исключения

Исключений нет.

Примеры

В листинге 4.49 генерируется пустая последовательность типа `string` с использованием операции `Empty` и отображается счетчик `Count` для сгенерированной последовательности, который должен быть равен нулю, поскольку последовательность пустая.

Листинг 4.49. Пример возврата пустой последовательность строк

```
IEnumerator<string> strings = Enumerable.Empty<string>();
foreach(string s in strings)
    Console.WriteLine(s);
Console.WriteLine(strings.Count());
```

Вывод этого кода:

0

Поскольку последовательность пуста, нечего отображать в цикле `foreach`, поэтому я добавил отображение значения счетчика элементов в последовательности.

Резюме

Я знаю, что это был очень поспешный тур по отложенным стандартным операциям запросов. Я попытался предоставить примеры применения почти каждого прототипа каждой отложенной операции, вместо того, чтобы ограничиться простейшими. Мне всегда не нравится, когда в книгах отображается простейшая форма вызова метода, а более сложные предлагаются исследовать самостоятельно. Надеюсь, мне удалось просто продемонстрировать даже самые сложные прототипы.

Вдобавок я полагаю, что разбив стандартные операции запросов на отложенные и нет, я правильно подчеркнул значение этого обстоятельства для ваших запросов.

Хотя в данной главе описано великое множество стандартных операций запросов, в следующей главе я дополню представление LINQ to Objects рассмотрением не отложенных стандартных операций запросов.

ГЛАВА 5

Не отложенные операции

В предыдущей главе рассказывалось об отложенных стандартных операциях запросов. Их легко отличить по тому, что они возвращают либо `IEnumerable<T>`, либо `OrderesSequence<T>`. Однако отложенные операции — это только половина из всех стандартных операций запросов. Для полноты картины нужно также описать не отложенные (*nondeferred*) операции. Их легко отличить по типу возврата, отличающемуся от `IEnumerable<T>`, и `OrderesSequence<T>`. Эти операции в настоящей главе разбиты по их назначению.

Для того чтобы закодировать и выполнить примеры настоящей главы, вам нужно будет удостовериться в наличии директив `using` для всех необходимых пространств имен. Вы должны также иметь некоторый общий код, разделяемый всеми примерами.

Необходимые пространства имен

Примеры этой главы будут использовать пространства имен `System.Linq`, `System.Collections` и `System.Collections.Generic`. Таким образом, вы должны добавить следующие директивы `using` в ваш код, если они еще там отсутствуют:

```
using System.Linq;
using System.Collections;
using System.Collections.Generic;
```

В дополнение к этим пространствам имен, если вы загрузите сопровождающий код, то увидите, что я также добавил директиву `using` для пространства имен `System.Diagnostics`. Это не обязательно, если вы будете самостоятельно набирать примеры из этой главы. Это важно только для сопровождающего кода, поскольку там это необходимо для некоторых домашних заготовок.

Общие классы

Несколько примеров этой главы требуют классов для того, чтобы полностью продемонстрировать поведение операций. В этом разделе я опишу четыре класса, которые будут применяться в более чем одном примере, начиная с класса `Employee`.

Класс `Employee` предназначен для того, чтобы представлять сотрудников. Для удобства он содержит статические методы для возврата `ArrayList` или массива сотрудников.

Разделяемый класс Employee

```
public class Employee
{
    public int id;
    public string firstName;
    public string lastName;
    public static ArrayList GetEmployeesArrayList()
    {
        ArrayList al = new ArrayList();
        al.Add(new Employee { id = 1, firstName = "Joe", lastName = "Rattz" });
        al.Add(new Employee { id = 2, firstName = "William", lastName = "Gates" });
        al.Add(new Employee { id = 3, firstName = "Anders", lastName = "Hejlsberg" });
        al.Add(new Employee { id = 4, firstName = "David", lastName = "Lightman" });
        al.Add(new Employee { id = 101, firstName = "Kevin", lastName = "Flynn" });
        return (al);
    }
    public static Employee[] GetEmployeesArray()
    {
        return ((Employee[])GetEmployeesArrayList().ToArray());
    }
}
```

Класс EmployeeOptionEntry представляет назначение опций определенному сотруднику. Для удобства он содержит статический метод, возвращающий массив назначенных опций.

Разделяемый класс EmployeeOptionEntry

```
public class EmployeeOptionEntry
{
    public int id;
    public long optionsCount;
    public DateTime dateAwarded;
    public static EmployeeOptionEntry[] GetEmployeeOptionEntries()
    {
        EmployeeOptionEntry[] empOptions = new EmployeeOptionEntry[]
        {
            new EmployeeOptionEntry {
                id = 1,
                optionsCount = 2,
                dateAwarded = DateTime.Parse("1999/12/31") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("1992/06/30") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("1994/01/01") },
            new EmployeeOptionEntry {
                id = 3,
                optionsCount = 5000,
                dateAwarded = DateTime.Parse("1997/09/30") },
            new EmployeeOptionEntry {
                id = 2,
                optionsCount = 10000,
                dateAwarded = DateTime.Parse("2003/04/01") },
            new EmployeeOptionEntry {
                id = 3,
                optionsCount = 7500,
                dateAwarded = DateTime.Parse("1998/09/30") }
        };
    }
}
```

```

        new EmployeeOptionEntry {
            id = 3,
            optionsCount = 7500,
            dateAwarded = DateTime.Parse("1998/09/30") },
        new EmployeeOptionEntry {
            id = 4,
            optionsCount = 1500,
            dateAwarded = DateTime.Parse("1997/12/31") },
        new EmployeeOptionEntry {
            id = 101,
            optionsCount = 2,
            dateAwarded = DateTime.Parse("1998/12/31") }
    );
    return (empOptions);
}
}

```

Несколько операций будут принимать классы, реализующие интерфейс `IEqualityComparer<T>` в целях сравнения элементов для определения их эквивалентности. Это полезно для тех случаев, когда два значения не в точности эквивалентны, но вы хотите трактовать их как эквивалентные. Например, вы можете игнорировать регистр при сравнении двух строк. Однако для такой ситуации в .NET Framework уже есть соответствующий класс для сравнения эквивалентности.

Поскольку я детально описал интерфейс `IEqualityComparer<T>` в предыдущей главе, не стану повторяться здесь.

Для моих примеров понадобится класс сравнения эквивалентности, которые знает, как сравнивать числа в строковом формате. Так, например, строки "17" и "00017" должны рассматриваться как эквивалентные. Ниже показан класс `MyStringifiedNumberComparer`, который делает это.

Разделяемый класс `MyStringifiedNumberComparer`

```

public class MyStringifiedNumberComparer : IEqualityComparer<string>
{
    public bool Equals(string x, string y)
    {
        return Int32.Parse(x) == Int32.Parse(y);
    }
    public int GetHashCode(string obj)
    {
        return Int32.Parse(obj).ToString().GetHashCode();
    }
}

```

Обратите внимание, что эта реализация интерфейса `IEqualityComparer` будет работать только с переменными типа `string`, но для данного примера этого достаточно. В основном, для всех сравнений я просто преобразую все значения из `string` в `Int32`. Таким образом, "002" превращается в целочисленное значение 2, так что ведущие нули никак не влияют на ключевое значение.

Для некоторых примеров этой главы мне понадобится класс, который сможет хранить записи с не уникальными ключами. Для этой цели я создал класс `Actor`, приведенный ниже. Я использую его член `birthYear` в качестве ключа специально для этой цели.

Разделяемый класс `Actor`

```

public class Actor
{
    public int birthYear;
}

```

```

public string firstName;
public string lastName;
public static Actor[] GetActors()
{
    Actor[] actors = new Actor[] { firstName = "Keanu", lastName = "Reeves" },
        new Actor { birthYear = 1964, firstName = "Owen", lastName = "Wilson" },
        new Actor { birthYear = 1968, firstName = "James", lastName = "Spader" },
        new Actor { birthYear = 1960, firstName = "Sandra", lastName = "Bullock" },
        new Actor { birthYear = 1964, firstName = "Sandra", lastName = "Bullock" },
    };
    return (actors);
}
}

```

Не отложенные операции по их назначению

В этом разделе описаны не отложенные стандартные операции запросов, разделенные по их предназначению.

Преобразование

Следующие операции преобразования предоставляют простой и удобный способ преобразования последовательностей в другие типы коллекций.

ToArray

Операция `ToArray` создает массив типа `T` из входной последовательности типа `T`.

Прототипы

У этой операции один прототип, описанный ниже.

Прототип `ToArray`

```
public static T[] ToArray<T>(
    this IEnumerable<T> source);
```

Эта операция берет входную последовательность `source` с элементами типа `T` и возвращает массив элементов типа `T`.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

Для примера, демонстрирующего операцию `ToArray`, мне понадобится последовательность типа `IEnumerable<T>`. Я создам последовательность этого типа вызовом операции `OfType`, которая была описана в предыдущей главе, из массива. Имея такую последовательность, я смогу вызывать операцию `ToArray` для создания массива, как показано в листинге 5.1.

Листинг 5.1. Пример вызова операции `ToArray`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
string[] names = presidents.OfType<string>().ToArray();
foreach (string name in names)
    Console.WriteLine(name);
```

Сначала я преобразую массив `presidents` в последовательность типа `IEnumerable<string>`, используя операцию `OfType`. Затем преобразую эту последовательность в массив с помощью операции `ToArray`. Поскольку `ToArray` — не отложенная операция, запрос выполняется немедленно, еще до его перечисления.

Запуск приведенного кода даст такой результат:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Pierce
Polk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson
```

Технически код этого примера несколько избыточен. Массив `presidents` — это уже последовательность, поскольку в C# 3.0 массивы реализуют интерфейс `IEnumerable<T>`. Поэтому я мог бы пропустить вызов операции `OfType` и просто вызвать операцию `ToArray` на массиве `presidents`. Однако я полагаю, было бы не очень впечатляюще преобразовывать массив в массив.

Эта операция часто бывает удобной для кэширования последовательности, чтобы она не могла измениться перед тем, как вы начнете ее перечисление. К тому же, поскольку эта операция не является отложенной и выполняется немедленно, множество перечислений, созданных на одном массиве, всегда будут видеть одни и те же данные.

ToList

Операция `ToList` создает `List` типа `T` из входной последовательности типа `T`.

Прототипы

У этой операции один прототип, описанный ниже.

Прототип `ToList`

```
public static List<T> ToList<T>(
    this IEnumerable<T> source);
```

Данная операция принимает последовательность по имени `source` элементов типа `T` и возвращает `List` элементов типа `T`.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

Листинг 5.2 демонстрирует применение операции `ToList`.

Листинг 5.2. Пример вызова операции `ToList`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
List<string> names = presidents.ToList();
foreach (string name in names)
    Console.WriteLine(name);
```

В приведенном коде я использую массив из предыдущего примера. Но в отличие от предыдущего примера я не вызываю операцию `OfType` для создания промежуточной последовательности `IEnumerable<T>`, поскольку кажется, достаточно преобразовать массив `presidents` в список `List<string>`.

Вот результат:

```
Adams
Arthur
Buchanan
Bush
Carter
Cleveland
Clinton
Coolidge
Eisenhower
Fillmore
Ford
Garfield
Grant
Harding
Harrison
Hayes
Hoover
Jackson
```

```

Jefferson
Johnson
Kennedy
Lincoln
Madison
McKinley
Monroe
Nixon
Pierce
Folk
Reagan
Roosevelt
Taft
Taylor
Truman
Tyler
Van Buren
Washington
Wilson

```

Эта операция часто бывает полезной для кэширования последовательности, чтобы она не могла измениться перед тем, как вы ее перечислите. Также, поскольку эта операция не является отложенной и выполняется немедленно, множество перечислений на созданном `List<T>` всегда видят одинаковые данные.

ToDictionary

Операция `ToDictionary` создает `Dictionary` типа `<K, T>`, или, возможно, `<K, E>`, если прототип имеет аргумент `elementSelector`, из входной последовательности типа `T`, где `K` — тип ключа, а `T` — тип хранимых значений. Или же, если `Dictionary` имеет тип `<K, E>`, то типом хранимых значений будет `E`, отличающийся от типа элементов в последовательности — `T`.

На заметку! Если вы не знакомы с классом `Dictionary` коллекций C#, скажу, что он позволяет хранить элементы, которые можно извлекать по ключу. Каждый ключ должен быть униканен, и только один элемент может быть сохранен для одного ключа. Вы индексируете элементы в `Dictionary` по ключу, чтобы извлекать эти элементы по этому ключу.

Прототипы

У этой операции имеется четыре прототипа, описанные ниже.

Первый прототип операции `ToDictionary`

```

public static Dictionary<K, T> ToDictionary<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);

```

В этом прототипе создается словарь `Dictionary` типа `<K, T>` и возвращаются при перечислении входной последовательности по имени `source`. Делегат метода `keySelector` вызывается для извлечения значения ключа из каждого входного элемента, и этот ключ становится ключом элемента в `Dictionary`. Эта версия операции дает в результате элементы в `Dictionary` того же типа, что и элементы входной последовательности.

Поскольку данный прототип не предусматривает спецификацию объекта `IEqualityComparer<K>`, эта версия `ToDictionary` по умолчанию использует объект для проверки эквивалентности `EqualityComparer<K>.Default`.

Второй прототип `ToDictionary` подобен первому, за исключением того, что он позволяет специфицировать объект проверки эквивалентности `IEqualityComparer<K>`. Ниже показан второй прототип.

Второй прототип операции `ToDictionary`

```
public static Dictionary<K, T> ToDictionary<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
```

Этот прототип предоставляет возможность специфицировать объект проверки эквивалентности `IEqualityComparer<K>`. Данный объект используется для сравнения элементов по значению ключа. Поэтому, если вы добавляете или обращаетесь к элементу в `Dictionary`, он использует этот `comparer` для сравнения указанного вами ключа с ключами, содержащимися в `Dictionary`, чтобы определить их соответствие.

Реализация по умолчанию интерфейса `IEqualityComparer<K>` представлена `EqualityComparer.Default`. Однако если вы собираетесь использовать класс проверки эквивалентности по умолчанию, нет причин специфицировать параметр `comparer`, потому что предыдущий прототип, где `comparer` не специфицирован, использует эту установку по умолчанию. Класс `StringComparer` реализует в себе несколько классов проверки эквивалентности, включая игнорирующий регистр символов. Таким образом, ключи "Joe" и "joe" оцениваются как эквивалентные.

Третий прототип `ToDictionary` подобен первому, за исключением того, что позволяет специфицировать селектор элемента, так что тип данных элементов, хранимых в `Dictionary`, может отличаться от типа элементов входной последовательности.

Третий прототип `ToDictionary`

```
public static Dictionary<K, E> ToDictionary<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector);
```

Через аргумент `elementSelector` вы можете специфицировать делегат метода, возвращающий часть входного элемента — или вновь созданный объект совершенно другого типа, — который вы хотите сохранить в `Dictionary`.

Четвертый прототип операции `ToDictionary` даст вам лучшее от всех предыдущих. Это комбинация второго и третьего прототипов, а это означает, что вы можете специфицировать объекты `elementSelector` и `comparer`.

Четвертый прототип `ToDictionary`

```
public static Dictionary<K, E> ToDictionary<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

Этот прототип специфицирует объекты `elementSelector` и `comparer`.

Исключения

Исключение `ArgumentNullException` генерируется, если аргументы `source`, `keySelector` или `elementSelector` равны `null`, или же ключ, возвращенный `keySelector`, равен `null`.

`ArgumentException` генерируется, если `keySelector` возвращает одинаковый ключ для двух элементов.

Примеры

В этом примере вместо типичного массива `presidents`, применяемого выше, я использую мой общий класс `Employee`. Я собираюсь создать словарь типа `Dictionary<int, Employee>`, где ключ типа `int` — это член `id` класса `Employee`, и сам объект `Employee` представляет хранящийся элемент.

В листинге 5.3 представлен пример вызова операции `ToDictionary` с применением класса `Employee`.

Листинг 5.3. Пример вызова первого прототипа `ToDictionary`

```
Dictionary<int, Employee> eDictionary =
    Employee.GetEmployeesArray().ToDictionary(k => k.id);
Employee e = eDictionary[2];
Console.WriteLine("Employee whose id == 2 is {0} {1}", e.firstName, e.lastName);
```

Я объявляю свой `Dictionary` с целочисленным типом ключа, поскольку в качестве ключа использую поле `Employee.id`. Поскольку прототип операции `ToDictionary` позволяет мне хранить только входной элемент целиком, которым является объект `Employee`, типом элемента `Dictionary` также является `Employee`. `Dictionary<int, Employee>` затем позволяет искать сотрудников по их идентификатору `id`, используя производительность и удобство класса `Dictionary`. Вот результат работы приведенного выше кода:

```
Employee whose id == 2 is William Gates
```

Для примера второго прототипа, поскольку его предназначение — позволить специфицировать объект, проверяющий эквивалентность типа `IEqualityComparer<T>`, мне понадобится ситуация, в которой будет полезен класс, проверяющий эквивалентность. Это ситуация, когда ключи, которые literally могут быть неэквивалентны, должны трактоваться такими посредством моего класса проверки эквивалентности. Я использую для этой цели числовое значение в строковом формате в качестве ключа, такое как "1". Поскольку иногда числовые значения в строковом формате имеют ведущие нули, также возможно, что ключ для одного и того же элемента данных может оказаться "1" или "01", или даже "00001". Поскольку эти строковые значения не эквивалентны, мне понадобится класс проверки эквивалентности, который будет знать, что все эти значения следует трактовать эквивалентными.

Сначала, однако, мне понадобится класс с ключом типа `string`. Для этого я проведу небольшую модификацию моего общего класса `Employee`, который уже использовал выше. Я создам следующий класс `Employee2`, во всем идентичный `Employee`, за исключением того, что типом члена `id` будет `string` вместо `int`.

Класс для примера применения второго прототипа операции `ToDictionary`

```
public class Employee2
{
    public string id;
    public string firstName;
    public string lastName;
    public static ArrayList GetEmployeesArrayList()
    {
        ArrayList al = new ArrayList();
        al.Add(new Employee2 { id = "1", firstName = "Joe", lastName = "Rattz" });
        al.Add(new Employee2 { id = "2", firstName = "William", lastName = "Gates" });
        al.Add(new Employee2 { id = "3", firstName = "Anders", lastName = "Hejlsberg" });
        al.Add(new Employee2 { id = "4", firstName = "David", lastName = "Lightman" });
        al.Add(new Employee2 { id = "101", firstName = "Kevin", lastName = "Flynn" });
        return (al);
    }
}
```

```

public static Employee2[] GetEmployeesArray()
{
    return ((Employee2[])GetEmployeesArrayList()).ToArray(typeof(Employee2));
}
}

```

Я изменил тип ключа на `string`, чтобы продемонстрировать, как может быть использован класс, проверяющий эквивалентность ключей, даже в случае, когда они лiteralно не эквивалентны. В этом примере, поскольку мои ключи теперь `string`, я использую мой общий класс `MyStringifiedNumberComparer`, который будет знать, что ключ "02" эквивалентен ключу "2".

Теперь давайте взглянем на некоторый код, использующий класс `Employee2` и мою реализацию `IEqualityComparer`, представленный в листинге 5.4.

Листинг 5.4. Пример вызова второго прототипа `ToDictionary`

```

Dictionary<string, Employee2> eDictionary = Employee2.GetEmployeesArray()
    .ToDictionary(k => k.id, new MyStringifiedNumberComparer());
Employee2 e = eDictionary["2"];
Console.WriteLine("Employee whose id == \"2\" : {0} {1}",
    e.firstName, e.lastName);
e = eDictionary["000002"];
Console.WriteLine("Employee whose id == \"000002\" : {0} {1}",
    e.firstName, e.lastName);

```

В данном примере я пытаюсь обратиться к элементам в `Dictionary` с значениями ключей "2" и "000002". Если мой класс, проверяющий эквивалентность, работает правильно, я должен получить в обоих случаях из `Dictionary` один и тот же элемент. Вот результат:

```

Employee whose id == "2" : William Gates
Employee whose id == "000002" : William Gates

```

Как видите, я получил один и тот же элемент из `Dictionary`, независимо от того, какой строковый ключ был использован для доступа — до тех пор, пока каждое переданное строковое значение представляет одно и то же целое число.

Третий прототип позволяет сохранять в словаре элемент, чей тип отличается от типа элементов входной последовательности. Для примера третьего прототипа я использую тот же класс `Employee`, который применялся в коде примера первого прототипа `ToDictionary`. Листинг 5.5 содержит код примера вызова третьего прототипа `ToDictionary`.

Листинг 5.5. Пример вызова третьего прототипа `ToDictionary`

```

Dictionary<int, string> eDictionary = Employee.GetEmployeesArray()
    .ToDictionary(k => k.id,
        i => string.Format("{0} {1}", // elementSelector
            i.firstName, i.lastName));
string name = eDictionary[2];
Console.WriteLine("Employee whose id == 2 is {0}", name);

```

В данном коде я представляю лямбда-выражение, соединяющее `firstName` и `lastName` в одну строку. Эта объединенная строка становится значением, хранимым в `Dictionary`. Таким образом, хотя тип входного элемента — `Employee`, тип данных элемента, хранимого в словаре — `string`. Вот результаты этого запроса:

```

Employee whose id == 2 is William Gates

```

Чтобы продемонстрировать прототип `ToDictionary`, я использую мои классы `Employee2` и `MyStringifiedNumberComparer`. Листинг 5.6 содержит код примера.

Листинг 5.6. Пример вызова четвертого прототипа `ToDictionary`

```
Dictionary<string, string> eDictionary = Employee2.GetEmployeesArray()
    .ToDictionary(k => k.id, // keySelector
        i => string.Format("{0} {1}", // elementSelector
            i.firstName, i.lastName),
        new MyStringifiedNumberComparer()); // comparer
string name = eDictionary["2"];
Console.WriteLine("Employee whose id == \"2\" : {0}", name);
name = eDictionary["000002"];
Console.WriteLine("Employee whose id == \"000002\" : {0}", name);
```

В приведенном коде я представил `elementSelector`, который специфицирует единственную строку в качестве значения, сохраняемого в `Dictionary`, а также пользовательский объект проверки эквивалентности. В результате я могу использовать "2" или "000002" для извлечения элемента из `Dictionary`, благодаря моему классу проверки эквивалентности, и то, что я теперь получаю из `Dictionary` — это просто строка, в которой содержатся имя и фамилия сотрудника, соединенные вместе. Вот результат:

```
Employee whose id == "2" : William Gates
Employee whose id == "000002" : William Gates
```

Как видите, обращение по индексу к `Dictionary` со значением ключа "2" или "000002" извлекает один и тот же элемент.

ToLookup

Операция `ToLookup` создает `Lookup` типа `<K, T>`, или, возможно, `<K, E>`, из входной последовательности типа `T`, где `K` — тип ключа, а `T` — тип хранимых значений. Или же, если `Lookup` типа `<K, T>`, то типом хранимых значений может быть `E`, который отличается от типа элементов входной последовательности `T`.

Хотя все прототипы операции `ToLookup` создают `Lookup`, возвращают они объект, реализующий интерфейс `ILookup`. В этом разделе я обычно буду ссылаться на объект, реализующий интерфейс `ILookup`, как `Lookup`.

На заметку! Если вы незнакомы с классом `Lookup` коллекций C#, скажу, что он позволяет хранить элементы, которые могут быть извлечены по ключу. Каждый ключ должен быть уникальным, и множество элементов может быть сохранено с одним ключом. Обращение по индексу к `Lookup` с применением ключа извлекает последовательность сохраненных с этим ключом элементов.

Прототипы

У операции `ToLookup` есть четыре прототипа, описанные ниже.

Первый прототип операции `ToLookup`

```
public static ILookup<K, T> ToLookup<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
```

В этом прототипе создается `Lookup` типа `<K, T>` и возвращается перечислением входной последовательности `source`. Делегат метода `keySelector` вызывается для извлечения ключевого значения из каждого входного элемента, и этот ключ является

ключом элемента в Lookup. Эта версия операции сохраняет в Lookup значения того же типа, что и элементы входной последовательности.

Поскольку прототип предотвращает спецификацию объекта проверки эквивалентности IEqualityComparer<K>, данная версия Lookup по умолчанию использует в качестве такого объекта экземпляр класса EqualityComparer<K>.Default.

Второй прототип ToLookup подобен первому, за исключением того, что предоставляет возможность специфицировать объект проверки эквивалентности IEqualityComparer<K>. Вот как выглядит второй прототип.

Второй прототип операции ToLookup

```
public static ILookup<K, T> ToLookup<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    IEqualityComparer<K> comparer);
```

Прототип предоставляет возможность специфицировать объект comparer типа IEqualityComparer<K> для проверки эквивалентности ключей. Поэтому, когда вы добавляете или обращаетесь к элементу в Lookup, он использует этот объект comparer для сравнения специфицированного вами ключа с ключами, уже находящимися в Lookup, чтобы определить их соответствие.

Реализация по умолчанию интерфейса IEqualityComparer<K> представлена EqualityComparer.Default. Однако если вы собираетесь использовать класс проверки эквивалентности по умолчанию, то специфицировать объект, проверяющий эквивалентность, незачем, потому что предыдущий прототип применяет его в любом случае. Класс StringComparer реализует некоторые классы проверки эквивалентности, такие как, например, игнорирующий различия в регистре. Таким образом, ключи "Joe" и "joe" трактуются как один и тот же ключ.

Третий прототип ToLookup подобен первому, за исключением того, что он позволяет специфицировать селектор элемента, чтобы тип данных значения, сохраненного в Lookup, мог отличаться от типа элементов входной последовательности. Третий прототип представлен ниже.

Третий прототип операции ToLookup

```
public static ILookup<K, E> ToLookup<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector);
```

В аргументе elementSelector вы можете специфицировать делегат метода, возвращающий часть входного элемента, либо вновь созданный объект совершенно другого типа, который вы хотите сохранить в Lookup.

Четвертый прототип операции ToLookup концентрирует в себе лучшее из двух предыдущих. Это комбинация второго и третьего прототипов, что означает, что вы можете специфицировать elementSelector и объект проверки эквивалентности comparer. Вот четвертый прототип.

Четвертый прототип операции ToLookup

```
public static ILookup<K, E> ToLookup<T, K, E>(
    this IEnumerable<T> source,
    Func<T, K> keySelector,
    Func<T, E> elementSelector,
    IEqualityComparer<K> comparer);
```

Этот прототип позволяет специфицировать и elementSelector, и comparer.

Исключения

Исключение `ArgumentNullException` генерируется, если аргументы `source`, `keySelector` или `elementSelector` равны `null`, либо же ключ, возвращенный `keySelector`, равен `null`.

Примеры

В этом примере прототипа `ToLookup` вместо типичного массива `presidents`, который я постоянно применяю, необходим класс с элементами, содержащими члены, которые могут быть использованы в качестве ключей, но не являются уникальными. Для этой цели я использую мой общий класс `Actor`.

Листинг 5.7 содержит пример вызова операции `ToLookup` с использованием класса `Actor`.

Листинг 5.7. Пример вызова первого прототипа `ToLookup`

```
Ilookup<int, Actor> lookup = Actor.GetActors().ToLookup(k => k.birthYear);
// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
Ienumerable<Actor> actors = lookup[1964];
foreach (var actor in actors)
    Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);
```

Во-первых, я создаю `Lookup`, используя член `Actor.birthYear` в качестве ключа к `Lookup`. Затем обращаюсь к нему по индексу, используя ключ 1964. Затем перечисляю возвращенные значения. И вот результат:

```
Keanu Reeves
Sandra Bullock
```

Похоже, я получил множественный результат. В конце концов, я и не ожидал, что он будет единственным. Поэтому хорошо, что я преобразовал входную последовательность в `Lookup` вместо `Dictionary`, потому что имеется несколько элементов с одинаковым ключом.

Для примера, демонстрирующего второй прототип `ToLookup`, я внесу небольшую модификацию в мой общий класс `Actor`. Я создам класс `Actor2`, во всем идентичный `Actor`, за исключением того, что член `birthYear` теперь имеет тип `string` вместо `int`.

Класс для примера применения второго прототипа операции `ToLookup`

```
public class Actor2
{
    public string birthYear;
    public string firstName;
    public string lastName;
    public static Actor2[] GetActors()
    {
        Actor2[] actors = new Actor2[]
        {
            new Actor2 { birthYear = "1964", firstName = "Keanu", lastName = "Reeves" },
            new Actor2 { birthYear = "1968", firstName = "Owen", lastName = "Wilson" },
            new Actor2 { birthYear = "1960", firstName = "James", lastName = "Spader" },
            // Первая дата, для которой отсутствует проблема Y10K!
            new Actor2 { birthYear = "01964", firstName = "Sandra",
                lastName = "Bullock" },
        };
        return(actors);
    }
}
```

Обратите внимание, что в этом классе я изменил тип члена birthYear на string. Теперь я буду вызывать операцию ToLookup, как показано в листинге 5.8.

Листинг 5.8. Пример вызова второго прототипа ToLookup

```
ILookup<string, Actor2> lookup = Actor2.GetActors()
    .ToLookup(k => k.birthYear, new MyStringifiedNumberComparer());
// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<Actor2> actors = lookup["0001964"];
foreach (var actor in actors)
    Console.WriteLine("{0} {1}", actor.firstName, actor.lastName);
```

Я использую тот же объект проверки эквивалентности, что и в примере Dictionary. В этом случае я преобразую входную последовательность в Lookup, предоставляя объект проверки эквивалентности, потому что знаю, что ключ, хранимый в виде string, может иногда содержать ведущие нули. Этот мой объект знает, как с этим справиться. И вот результат:

Keanu Reeves
Sandra Bullock

Обратите внимание, что я пытаюсь извлечь все элементы со значением ключа "0001964". И получаю элементы с ключами "1964" и "01964". Значит, мой объект проверки эквивалентности работает.

Для третьего прототипа операции ToLookup я применяю тот же класс Actor, что использовал в примере кода первого прототипа для ToLookup. В листинге 5.9 представлен код примера, вызывающий третий прототип ToLookup.

Листинг 5.9. Пример вызова третьего прототипа ToLookup

```
ILookup<int, string> lookup = Actor.GetActors()
    .ToLookup(k => k.birthYear,
        a => string.Format("{0} {1}", a.firstName, a.lastName));
// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<string> actors = lookup[1964];
foreach (var actor in actors)
    Console.WriteLine("{0}", actor);
```

В моем elementSelector я просто соединяю в одну строку члены firstName и lastName. И вот результат:

Keanu Reeves
Sandra Bullock

Использование варианта операции ToLookup с elementSelector позволяет мне сохранять в Lookup данные другого типа, отличного от типа данных элемента входной последовательности.

Для примера четвертого прототипа ToLookup я использую класс Actor2 и мой общий класс MyStringifiedNumberComparer. Код примера представлен в листинге 5.10.

Листинг 5.10. Пример вызова четвертого прототипа ToLookup

```
ILookup<string, string> lookup = Actor2.GetActors()
    .ToLookup(k => k.birthYear,
        a => string.Format("{0} {1}", a.firstName, a.lastName),
        new MyStringifiedNumberComparer());
// Посмотрим, можно ли найти кого-то, рожденного в 1964 г.
IEnumerable<string> actors = lookup["0001964"];
foreach (var actor in actors)
    Console.WriteLine("{0}", actor);
```

Вот результат:

```
Keanu Reeves
Sandra Bullock
```

Как видите, я обращаюсь к `Lookup` по индексу, используя значение ключа, отличное от любого из действительных значений ключей извлеченных значений, поэтому можно сделать вывод, что мой объект проверки эквивалентности работает. И вместо того, чтобы сохранять весь объект `Actor2`, я просто сохраняю интересующую меня строку.

Эквивалентность

Следующие операции эквивалентности используются для проверки эквивалентности последовательностей.

`SequenceEqual`

Операция `SequenceEqual` определяет, эквивалентны ли две входные последовательности.

Прототипы

У операции `SequenceEqual` два прототипа, описанные ниже.

Первый прототип `SequenceEqual`

```
public static bool SequenceEqual<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

Эта операция перечисляет каждую входную последовательность параллельно, сравнивая элементы посредством метода `System.Object.Equals`. Если элементы эквивалентны, и последовательности содержат одинаковое количество элементов, операция возвращает `true`. Иначе она возвращает `false`.

Второй прототип операции работает так же, как и первый, за исключением того, что принимает объект `IEqualityComparer<T>`, которые может быть использован для проверки эквивалентности.

Второй прототип `SequenceEqual`

```
public static bool SequenceEqual<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Пример приведен в листинге 5.11.

Листинг 5.11. Пример вызова первого прототипа операции `SequenceEqual`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
```

```
bool eq = presidents.SequenceEqual(presidents);
Console.WriteLine(eq);
```

И вот результат:

True

Смотрится бедненько, не так ли? Хорошо, я немного усложню пример в листинге 5.12.

Листинг 5.12. Другой пример вызова первого прототипа операции SequenceEqual

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool eq = presidents.SequenceEqual(presidents.Take(presidents.Count()));
Console.WriteLine(eq);
```

В приведенном коде я использовал операцию Take, чтобы выбрать только первые N элементов из массива presidents, а затем сравнить эту выходную последовательность с оригинальным массивом presidents.

Итак, если в приведенном выше коде я возьму все элементы массива presidents, указав количество элементов через presidents.Count(), то должен буду получить всю выходную последовательность целиком. Как и следовало ожидать, вот результат:

True

Хорошо, все работает, как и должно. Теперь я возьму все элементы кроме последнего, вычтя единицу из presidents.Count(), как показано в листинге 5.13.

Листинг 5.13. Еще один пример вызова первого прототипа операции SequenceEqual

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool eq = presidents.SequenceEqual(presidents.Take(presidents.Count() - 1));
Console.WriteLine(eq);
```

Теперь результат должен быть false, потому что две последовательности имеют разную длину. Во второй из них недостает последнего элемента:

False

Все идет хорошо. Но просто из любопытства давайте продолжим. Если вспомнить описание операций Take и Skip из предыдущей главы, там было сказано, что соединенные вместе надлежащим образом, они должны породить исходную последовательность. Теперь я это попробую. Я использую операции Take, Skip, Concat и SequenceEqual, чтобы доказать это утверждение, как показано в листинге 5.14.

Листинг 5.14. Более сложный пример вызова первого прототипа операции SequenceEqual

```

string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool eq =
    presidents.SequenceEqual(presidents.Take(5).Concat(presidents.Skip(5)));
Console.WriteLine(eq);

```

В данном примере я беру первые пять элементов исходной последовательности, вызывая операцию Take. Затем соединяю входную последовательность, начиная с шестого элемента с помощью операций Skip и Concat. И, наконец, я определяю, эквивалентны ли эта объединенная последовательность исходной последовательности, вызывая для этой операции SequenceEqual. Как вы думаете? Давайте посмотрим:

True

Отлично, все работает! Для примера использования второго прототипа я создаю два массива типа string, где каждый элемент представляет собой число в строковой форме. Элементы двух массивов будут такими, что при преобразовании в числа окажутся эквивалентными. В этом примере, показанном в листинге 5.15, я использую мой общий класс MyStringifiedNumberComparer.

Листинг 5.15. Пример вызова второго прототипа операции SequenceEqual

```

string[] stringifiedNums1 = {
    "001", "49", "017", "0080", "00027", "2" };
string[] stringifiedNums2 = {
    "1", "0049", "17", "080", "27", "02" };
bool eq = stringifiedNums1.SequenceEqual(stringifiedNums2,
                                         new MyStringifiedNumberComparer());
Console.WriteLine(eq);

```

В этом примере, если вы проверите два массива, то увидите, что если преобразовать каждый элемент обоих массивов в целое число, а затем сравнить соответствующие числа, то эти два массива должны будут считаться эквивалентными. Посмотрим, подтверждает ли это результат:

True

Элемент

Следующие операции элементов позволяют вам извлекать отдельные элементы из входной последовательности.

First

Операция First возвращает первый элемент последовательности или первый элемент последовательности, соответствующий предикату — в зависимости от использованного прототипа.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип First

```
public static T First<T>(
    this IEnumerable<T> source);
```

При использовании этого прототипа операции `First` выполняется перечисление входной последовательности `source` и возвращается ее первый элемент.

Второй прототип операции `First` позволяет передать ему предикат.

Второй прототип First

```
public static T First<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Эта версия операции `First` возвращает первый найденный им элемент, для которого `predicate` вернул `true`. Если ни один из элементов не заставил `predicate` вернуть `true`, то операция `First` генерирует исключение `InvalidOperationException`.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`. `InvalidOperationException` генерируется в случае пустой последовательности `source`, либо когда `predicate` ни разу не вернул `true`.

Примеры

Пример первого прототипа `First` приведен в листинге 5.16.

Листинг 5.16. Пример кода, вызывающего первый прототип First

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.First();
Console.WriteLine(name);
```

Вот результат:

Adams

Вы можете спросить: чем отличается эта операция от вызова операции `Take` с параметром 1? Отличие в том, что `Take` возвращает последовательность элементов, даже если она состоит всего из одного элемента. Операция `First` всегда возвращает точно один элемент либо генерирует исключение, если возвращать нечего. Листинг 5.17 демонстрирует пример использования второго прототипа операции `First`.

Листинг 5.17. Пример кода, вызывающего второй прототип First

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.First(p => p.StartsWith("H"));
Console.WriteLine(name);
```

Он должен вернуть первый элемент из входной последовательности, начинающийся с строки "H". Вот результат:

Harding

Напомню, что если любой из прототипов операции `First` не находит элемента, который нужно возвратить, генерируется исключение `InvalidOperationException`. Чтобы избежать этого, используйте операцию `FirstOrDefault`.

FirstOrDefault

Операция `FirstOrDefault` подобна `First` во всем, кроме поведения, когда элемент не найден.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип `FirstOrDefault`

```
public static T FirstOrDefault<T>(
    this IEnumerable<T> source);
```

Эта версия прототипа `FirstOrDefault` возвращает первый элемент, найденный во входной последовательности. Если последовательность пуста, возвращается `default(T)`. Для ссылочных и допускающих `null` типов значением по умолчанию является `null`.

Второй прототип операции `FirstOrDefault` позволяет передать `predicate`, определяющий, какой элемент следует возвратить.

Второй прототип `FirstOrDefault`

```
public static T FirstOrDefault<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Листинг 5.18 содержит пример первого прототипа `FirstOrDefault`, где не найдено никаких элементов. Для того чтобы получить это, мне понадобилась пустая последовательность. Я получил ее от `Take(0)`.

Листинг 5.18. Вызов первого прототипа `FirstOrDefault`, когда элемент не найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Take(0).FirstOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Вот результат:

NULL

В листинге 5.19 представлен тот же пример без вызова `Take(0)`, так что элемент найден.

Листинг 5.19. Вызов первого прототипа FirstOrDefault, когда элемент найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

И, наконец, результат работы кода, когда элемент найден:

Adams

Для второго прототипа FirstOrDefault я укажу, что мне нужен первый элемент, который начинается со строки "B", как показано в листинге 5.20.

Листинг 5.20. Вызов второго прототипа FirstOrDefault, когда элемент найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault(p => p.StartsWith("B"));
Console.WriteLine(name == null ? "NULL" : name);
```

И вот результат:

Buchanan

Теперь я попробую это с predicate, который не найдет соответствия, как показано в листинге 5.21.

Листинг 5.21. Вызов второго прототипа FirstOrDefault, когда элемент не найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.FirstOrDefault(p => p.StartsWith("Z"));
Console.WriteLine(name == null ? "NULL" : name);
```

Поскольку в массиве presidents нет имени, начинающегося с "Z", результат будет таким:

NULL

Last

Операция Last возвращает последний элемент последовательности или последний элемент, соответствующий предикату — в зависимости от того, какой прототип использован.

Прототипы

Первый прототип Last

```
public static T Last<T>(
    this IEnumerable<T> source);
```

При использовании этого прототипа операция Last перечисляет входную последовательность по имени source и возвращает ее последний элемент.

Второй прототип Last позволяет передать predicate и выглядит, как описано ниже.

Второй прототип Last

```
public static T Last<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Эта версия операции Last возвращает последний из найденных элементов, для которых predicate вернет true.

Примеры

Листинг 5.22 содержит пример первого прототипа Last.

Листинг 5.22. Пример вызова первого прототипа Last

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Last();
Console.WriteLine(name);
```

Вот его результат:

Wilson

Операция Last всегда возвращает точно один элемент или же генерирует исключение, если возвращать нечего.

В листинге 5.23 представлен код примера применения второго прототипа Last.

Листинг 5.23. Вызов второго прототипа Last

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Last(p => p.StartsWith("H"));
Console.WriteLine(name);
```

Это должно вернуть последний элемент входной последовательности, начинающийся на строку "H". Вот результат запуска этого примера:

Hoover

Напомню, что если любому из двух прототипов операции `Last` возвращать `null`, он генерирует исключение `InvalidOperationException`. Чтобы избежать этого, используйте операцию `LastOrDefault`.

LastOrDefault

Операция `LastOrDefault` подобна `Last` во всем, за исключением поведения в случае, когда элемент не найден.

Прототипы

Первый прототип `LastOrDefault`

```
public static T LastOrDefault<T>(
    this IEnumerable<T> source);
```

Эта версия прототипа `LastOrDefault` возвращает последний элемент, найденный во входной последовательности. Если последовательность пуста, возвращается `default(T)`. Для ссылочных и допускающих `null` типов значением по умолчанию является `null`.

Второй прототип операции `LastOrDefault` позволяет передать ему `predicate`, определяющий, какой элемент должен быть возвращен.

Второй прототип `LastOrDefault`

```
public static T LastOrDefault<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Листинг 5.24 содержит пример первого прототипа `LastOrDefault`, где не найде никаких элементов. Для того чтобы получить это, мне понадобилась пустая последовательность. Для ее получения я вызвал `Take(0)`.

Листинг 5.24. Вызов прототипа `LastOrDefault`, когда элемент не найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.Take(0).LastOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Вот результат:

NULL

Листинг 5.25 демонстрирует тот же пример, но без `Take(0)`, поэтому элемент он находит.

Листинг 5.25. Вызов прототипа LastOrDefault, когда элемент найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.LastOrDefault();
Console.WriteLine(name == null ? "NULL" : name);
```

Вот результат:

Wilson

Для второго прототипа операции LastOrDefault, показанного в листинге 5.26, я указал, что мне нужен последний элемент, начинающийся со строки "B".

Листинг 5.26. Вызов второго прототипа LastOrDefault, когда элемент найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.LastOrDefault(p => p.StartsWith("B"));
Console.WriteLine(name == null ? "NULL" : name);
```

Результат будет таким:

Bush

Теперь я попробую указать predicate, который не найдет соответствия, как показано в листинге 5.27.

Листинг 5.27. Вызов второго прототипа LastOrDefault, когда элемент не найден

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string name = presidents.LastOrDefault(p => p.StartsWith("Z"));
Console.WriteLine(name == null ? "NULL" : name);
```

Поскольку ни одно имя в массиве presidents не начинается с "Z", результат вполне ожидаемый:

NULL

Single

Операция Single возвращает единственный элемент последовательности или единственный элемент последовательности, соответствующий предикату — в зависимости от используемого прототипа.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип Single

```
public static T Single<T> (
    this IEnumerable<T> source);
```

Используя этот прототип, операция Single перечисляет входную последовательность по имени source и возвращает единственный ее элемент.

Второй прототип Single принимает predicate и выглядит, как описано ниже.

Второй прототип Single

```
public static T Single<T> (
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Эта версия операции Single возвращает единственный найденный элемент, для которого predicate возвращает true. Если ни один из элементов не заставляет predicate вернуть true, то операция Single генерирует исключение InvalidOperationException.

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null. InvalidOperationException генерируется в случае пустой последовательности source, либо когда predicate ни разу не вернул true, или же нашел более одного элемента, для которого вернул true.

Примеры

В листинге 5.28 содержится пример применения первого прототипа Single, использующий общий класс Employee.

Листинг 5.28. Пример кода, вызывающего первый прототип Single

```
Employee emp = Employee.GetEmployeesArray()
    .Where(e => e.id == 3).Single();
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

В данном примере вместо ожидания получения последовательности от запроса мне нужна просто ссылка на определенного сотрудника. Операция Single очень полезна в такой ситуации — до тех пор, пока вы можете гарантировать, что в переданной ему последовательности есть только один элемент. В данном случае, поскольку я вызвал операцию Where, указав уникальный ключ, я в безопасности. И вот результат:

Anders Hejlsberg

Листинг 5.29 содержит пример кода, использующего второй прототип операции Single.

Листинг 5.29. Пример кода, вызывающего второй прототип Single

```
Employee emp = Employee.GetEmployeesArray()
    .Single(e => e.id == 3);
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Данный код функционально эквивалентен предыдущему примеру. Вместо вызова операции Where для того, чтобы гарантировать наличие единственного элемента в последовательности, я могу выполнить ту же операцию фильтрования последовательности

в самой операции `Single`. Она должна вернуть только один элемент из входной последовательности, чей `id` равен 3. Смотрим результат:

Anders Hejlsberg

Напомню: если любой прототип операции `Single` не находит элемента для возврата, генерируется исключение `InvalidOperationException`. Избежать этого позволяет операция `SingleOrDefault`.

`SingleOrDefault`

Операция `SingleOrDefault` подобна `Single`, но отличается поведением в случае, когда элемент не найден.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип `SingleOrDefault`

```
public static T SingleOrDefault<T>(
    this IEnumerable<T> source);
```

Эта версия прототипа возвращает единственный элемент, найденный во входной последовательности. Если последовательность пуста, возвращается `default(T)`. Для ссылочных и допускающих `null` типов значением по умолчанию является `null`. Если найдено более одного элемента, генерируется исключение `InvalidOperationException`.

Второй прототип операции `SingleOrDefault` позволяет вам передать `predicate`, определяющий, какой элемент должен возвращаться.

Второй прототип `SingleOrDefault`

```
public static T SingleOrDefault<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`. `InvalidOperationException` генерируется в случае, когда `predicate` встречает более одного элемента, для которого возвращает `true`.

Примеры

Листинг 5.30 содержит пример вызова первого прототипа `SingleOrDefault`, когда элемент не найден. Для этого мне понадобилась пустая последовательность. Я воспользовалась операцией `Where` и указу в качестве условия фильтра несуществующее значение `500`.

Листинг 5.30. Вызов первого прототипа `SingleOrDefault`, когда элемент не найден

```
Employee emp = Employee.GetEmployeesArray()
    .Where(e => e.id == 5).SingleOrDefault();
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Я запросил сотрудника, чей `id` равен 5, поскольку знаю, что такого нет, поэтому будет возвращена пустая последовательность. В отличие от операции `Single`, `SingleOrDefault` нормально обрабатывает пустые последовательности. Вот результат работы примера:

Листинг 5.31 демонстрирует тот же пример, где единственный элемент найден. Я использую операцию Where, чтобы получить последовательность, содержащую единственный элемент.

Листинг 5.31. Вызов первого прототипа SingleOrDefault, когда элемент найден

```
Employee emp = Employee.GetEmployeesArray()
    .Where(e => e.id == 4).SingleOrDefault();
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

На этот раз я указал существующий id. И вот результат работы кода, когда элемент найден:

David Lightman

Как видите, сотрудник найден. Для второго прототипа SingleOrDefault, показанного в листинге 5.32, я специфицирую id, о котором точно знаю, что он существует, но вместо операции Where встраиваю фильтр в сам вызов операции SingleOrDefault.

Листинг 5.32. Вызов второго прототипа SingleOrDefault, когда элемент найден

```
Employee emp = Employee.GetEmployeesArray()
    .SingleOrDefault(e => e.id == 4);
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Этот пример функционально эквивалентен предыдущему, но с тем отличием, что вместо фильтрации элементов посредством операции Where я делаю это, передав predicate операции SingleOrDefault. И вот результат:

David Lightman

Теперь я попробую то же самое, но с предикатом, который не найдет соответствия, как показано в листинге 5.33.

Листинг 5.33. Вызов второго прототипа SingleOrDefault, когда элемент не найден

```
Employee emp = Employee.GetEmployeesArray()
    .SingleOrDefault(e => e.id == 5);
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Поскольку нет элемента с id равным 5, никакого элемента не найдено. Об этом свидетельствует результат:

NULL

Когда элементов не найдено во входной последовательности, операция SingleOrDefault изящно обрабатывает эту ситуацию вместо генерации исключений.

ElementAt

Операция ElementAt возвращает элемент из исходной последовательности по указанному индексу.

Прототипы

У этой операции один прототип, описанный ниже.

Прототип ElementAt

```
public static T ElementAt<T>(
    this IEnumerable<T> source,
    int index);
```

Если последовательность реализует `IList<T>`, то интерфейс `IList` используется для извлечения индексированного элемента непосредственно. Если же последовательность не реализует `IList<T>`, то последовательность перечисляется до тех пор, пока не будет достигнутый указанный индексом элемент. Исключение `ArgumentOutOfRangeException` генерируется, если индекс меньше нуля либо больше или равен числу элементов в последовательности.

Напоминание! В C# индексы начинаются с нуля. Это значит, что индекс первого элемента равен нулю. Индекс последнего элемента равен счетчику элементов последовательности минус единица.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`. `ArgumentOutOfRangeException` генерируется, если индекс меньше нуля или больше или равен количеству элементов в последовательности.

Примеры

Листинг 5.34 содержит пример вызова единственного прототипа операции `ElementAt`.

Листинг 5.34. Вызов операции ElementAt

```
Employee emp = Employee.GetEmployeesArray()
    .ElementAt(3);
Console.WriteLine("{0} {1}", emp.firstName, emp.lastName);
```

Я указал, что мне нужен элемент, чей индекс равен 3, т.е. четвертый по счету. И вот результат этого запроса:

David Lightman

ElementAtOrDefault

Операция `ElementAtOrDefault` возвращает элемент из исходной последовательности, имеющий указанный индекс местоположения.

Прототипы

У этой операции один прототип, описанный ниже.

Прототип ElementAtOrDefault

```
public static T ElementAtOrDefault<T>(
    this IEnumerable<T> source,
    int index);
```

Если последовательность реализует `IList<T>`, то интерфейс `IList` используется для извлечения индексированного элемента непосредственно. Если же последовательность не реализует `IList<T>`, то последовательность перечисляется до тех пор, пока не будет достигнутый указанный индексом элемент. Если заданный индекс меньше либо больше или равен количеству элементов в последовательности, возвращает-

ся `default(T)`. Для ссылочных и допускающих `null` типов возвращается значение по умолчанию `null`. Это поведение отличает его от операции `ElementAt`.

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен `null`.

Примеры

Листинг 5.35 содержит пример вызова операции `ElementAtOrDefault` с корректным индексом.

Листинг 5.35. Вызов операции `ElementAtOrDefault` с корректным индексом

```
Employee emp = Employee.GetEmployeesArray()
    .ElementAtOrDefault(3);
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Вот результат запроса:

David Lightman

Как и ожидалось, извлекается элемент с индексом 3. Теперь я попробую запустить запрос с неправильным индексом, используя код, представленный в листинге 5.36.

Листинг 5.36. Вызов операции `ElementAtOrDefault` с неверным индексом

```
Employee emp = Employee.GetEmployeesArray()
    .ElementAtOrDefault(5);
Console.WriteLine(emp == null ? "NULL" :
    string.Format("{0} {1}", emp.firstName, emp.lastName));
```

Поскольку в последовательности нет элемента с индексом 5, результат будет таким:

NULL

Квантификаторы

Следующие операции-квантификаторы позволяют выполнять операции типа квантификации над входной последовательностью.

Any

Операция `Any` возвращает `true`, если любой из элементов входной последовательности отвечает условию.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип `Any`

```
public static bool Any<T>(
    this IEnumerable<T> source);
```

Этот прототип операции `Any` вернет `true`, если входная последовательность `source` содержит любые элементы. Второй прототип операции `Any` перечисляет входную последовательность и возвращает `true`, если хотя бы один элемент из входной последовательности заставит вызов делегата метода `predicate` вернуть `true`. Перечисление входной последовательности `source` прекращается, как только `predicate` возвращает `true`.

Второй прототип Any

```
public static bool Any<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен null.

Примеры

Сначала я воспроизведу ситуацию с пустой последовательностью, как показано в листинге 5.37. Я использую операцию `Empty`, описанную в предыдущей главе.

Листинг 5.37. Первый прототип Any при отсутствии элементов во входной последовательности

```
bool any = Enumerable.Empty<string>().Any();
Console.WriteLine(any);
```

Результат выполнения этого кода:

False

Затем я попробую тот же прототип, но на этот раз с непустой входной последовательностью, как показано в листинге 5.38.

Листинг 5.38. Первый прототип Any при наличии элементов во входной последовательности

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any();
Console.WriteLine(any);
```

Вот результат запуска этого кода:

True

Для следующего примера я использую второй прототип — сначала без элементов, соответствующих `predicate`, как показано в листинге 5.39.

Листинг 5.39. Второй прототип Any, когда ни для одного элемента предикат не возвращает True

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any(s => s.StartsWith("Z"));
Console.WriteLine(any);
```

Я указал, что мне нужны президенты с именами, начинающимися со строки "Z". Поскольку таких нет, будет возвращена пустая последовательность, что заставит операцию `Any` вернуть `false`. Результат, как и ожидалось:

False

И, наконец, я попробую пример второго прототипа с `predicate`, который должен вернуть `true`, по крайней мере, для одного элемента, как показано в листинге 5.40.

Листинг 5.40. Второй прототип `Any`, когда минимум для одного элемента предикат возвращает `True`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool any = presidents.Any(s => s.StartsWith("A"));
Console.WriteLine(any);
```

Наконец, мы имеем результат:

`True`

All

Операция `All` возвращает `true`, если каждый элемент входной последовательности отвечает условию.

Прототипы

У операции `All` один прототип, описанный ниже.

Прототип `All`

```
public static bool All<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Операция `All` перечисляет входную последовательность `source` и возвращает `true`, только если `predicate` возвращает `true` для каждого элемента последовательности. Как только `predicate` вернет `false`, перечисление прекращается.

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

В листинге 5.41 я начну с `predicate`, о котором знаю, что для некоторых элементов он вернет `false`.

Листинг 5.41. Прототип `All`, когда не все элементы заставляют предикат вернуть `True`

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool all = presidents.All(s => s.Length > 5);
Console.WriteLine(all);
```

Поскольку известно, что не все имена президентов в массиве имеют длину более пяти символов, я знаю, что predicate вернет `false` для некоторых элементов. Вот результат:

`False`

Теперь я воспроизведу случай, когда каждый элемент последовательности заставит predicate вернуть `true`, как показано в листинге 5.42.

Листинг 5.42. Прототип All, когда все элементы заставляют предикат вернуть True

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool all = presidents.All(s => s.Length > 3);
Console.WriteLine(all);
```

Поскольку я знаю, что каждое из имен президентов состоит не менее чем из четырех символов, операция `All` должна вернуть `true`. Что мы и видим в выводе:

`True`

Contains

Операция `Contains` возвращает `true`, если любой элемент входной последовательности соответствует указанному значению.

Прототипы

У этой операции имеются два прототипа, описанные ниже.

Первый прототип Contains

```
public static bool Contains<T>(
    this IEnumerable<T> source,
    T value);
```

Этот прототип операции `Contains` сначала проверяет входную последовательность на предмет реализации ею интерфейса `ICollection<T>`, и если она его реализует, то вызывается метод `Contains` реализации последовательности. Если последовательность не реализует интерфейс `ICollection<T>`, входная последовательность `source` перечисляется с проверкой соответствия каждого элемента специфицированному значению. Как только находится элемент, который ему не соответствует, перечисление прекращается.

Указанное значение сравнивается с каждым элементом посредством класса проверки эквивалентности по умолчанию `EqualityComparer<K>.Default`.

Второй прототип подобен первому, за исключением возможности спецификации объекта `IEqualityComparer<T>`. Если используется этот прототип, каждый элемент в последовательности сравнивается с переданным значением посредством переданного объекта проверки эквивалентности.

Второй прототип Contains

```
public static bool Contains<T>(
    this IEnumerable<T> source,
    T value,
    IEqualityComparer<T> comparer);
```

Исключения

Исключение `ArgumentNullException` генерируется, если аргумент `source` равен null.

Примеры

Для демонстрации первого прототипа я начну со значения, о котором известно, что оно отсутствует в моей входной последовательности, как показано в листинге 5.43.

Листинг 5.43. Прототип `Contains`, когда ни один элемент не соответствует специфицированному значению

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool contains = presidents.Contains("Rattz");
Console.WriteLine(contains);
```

Поскольку в массиве нет элемента со значением "Rattz", переменная `contains` должна быть `false`. Вот вывод:

`False`

В листинге 5.44 мне известно, что элемент соответствует моему специфицированному значению.

Листинг 5.44. Прототип `Contains`, когда элемент соответствует специфицированному значению

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
bool contains = presidents.Contains("Hayes");
Console.WriteLine(contains);
```

Поскольку есть элемент со значением "Hayes", переменная `contains` должна быть `true`. И вот результат:

`True`

Для демонстрации примера использования второго прототипа `Contains` я использую мой общий класс `MyStringifiedNumberComparer`. Я проверю массив чисел в строковом формате на предмет наличия числа, технически не эквивалентного ни одному элементу массива, но поскольку я применяю собственный класс проверки эквивалентности, соответствующий элемент будет найден. В листинге 5.45 показан пример.

Листинг 5.45. Второй прототип `Contains`, когда элемент соответствует специфицированному значению

```
string[] stringifiedNums = {
    "001", "49", "017", "0080", "00027", "2" };
bool contains = stringifiedNums.Contains("0000002",
    new MyStringifiedNumberComparer());
Console.WriteLine(contains);
```

Поскольку я ищу элемент со значением "0000002" и используется мой объект проверки эквивалентности, который преобразует это строковое значение наряду со всеми элементами последовательности в целое число, прежде чем выполнить сравнение, и потому что моя последовательность содержит элемент "2", переменная `contains` должна быть равна `true`. Взглянем на результат:

```
True
```

Теперь я попробую тот же пример, но на этот раз запросив элемент, который, как известно, не существует. Код представлен в листинге 5.46.

Листинг 5.46. Второй прототип `Contains`, когда элемент не соответствует специфицированному значению

```
string[] stringifiedNums = {
    "001", "49", "017", "0080", "00027", "2" };
bool contains = stringifiedNums.Contains("0000271",
                                         new MyStringifiedNumberComparer());
Console.WriteLine(contains);
```

Поскольку я знаю, что ни один из элементов не может быть преобразован в целое число, равное 271, я выполняю поиск в массиве значения "000271". И вот результат:

```
False
```

Агрегация

Последующие операции агрегирования позволяют выполнять агрегатные операции над элементами входной последовательности.

Count

Операция `Count` возвращает количество элементов во входной последовательности.

Прототипы

У этой операции имеются два прототипа, описанные ниже.

Первый прототип `Count`

```
public static int Count<T>(
    this IEnumerable<T> source);
```

Этот прототип операции `Count` возвращает общее количество элементов во входной последовательности, проверяя сначала, реализует ли она интерфейс `ICollection<T>`, и если да, получая счетчик последовательности через реализацию этого интерфейса. Если же входная последовательность `source` не реализует интерфейс `ICollection<T>`, `Count` перечисляет всю эту последовательность, подсчитывая количество элементов.

Второй прототип операции `Count` перечисляет входную последовательность `source` и подсчитывает все элементы, которые заставляют делегат метода `predicate` вернуть `true`.

Второй прототип `Count`

```
public static int Count<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Листинг 5.47 начинается с использования первого прототипа. Сколько элементов содержится в последовательности presidents?

Листинг 5.47. Первый прототип Count

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
int count = presidents.Count();
Console.WriteLine(count);
```

Вот результат:

37

Теперь я попробую второй прототип, как показано в листинге 5.48. На этот раз подсчитываю количество президентов, чье имя начинается с "J".

Листинг 5.48. Второй прототип Count

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
int count = presidents.Count(s => s.StartsWith("J"));
Console.WriteLine(count);
```

Результат работы этого кода:

3

А как быть, если количество элементов превысит значение `int.MaxValue`? Для этого предусмотрена операция `LongCount`.

LongCount

Операция `LongCount` возвращает количество элементов входной последовательности как значение типа `long`.

Прототипы

У этой операции два прототипа, описанные ниже.

Первый прототип LongCount

```
public static long LongCount<T>(
    this IEnumerable<T> source);
```

Этот прототип операции `LongCount` возвращает общее количество элементов в входной последовательности, перечисляя всю эту последовательность и подсчитывая число элементов.

Второй прототип операции `LongCount` перечисляет входную последовательность `source` и подсчитывает все элементы, которые заставляют делегат метода `predicate` вернуть `true`.

Второй прототип `LongCount`

```
public static int LongCount<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`.

Примеры

Начну с примера первого прототипа, показанного в листинге 5.49. Я мог бы просто повторить те же два примера, что сопровождали описание операции `Count`, изменив соответствующие места на тип `long`, но это было бы не слишком наглядно. Поскольку было бы нереально описать достаточно длинную статическую последовательность, для которой понадобился бы `LongCount`, я использую стандартный оператор запроса, чтобы сгенерировать ее. К сожалению, генерирующие операции, описанные в предыдущей главе, позволяют специфицировать количество элементов в значении `int`. Придется соединить вместе пару таких сгенерированных последовательностей для получения достаточного количества элементов, чтобы потребовалась операция `LongCount`.

Листинг 5.49. Первый прототип `LongCount`

```
long count = Enumerable.Range(0, int.MaxValue).
    Concat(Enumerable.Range(0, int.MaxValue)).LongCount();
Console.WriteLine(count);
```

Как видите, я сгенерировал две последовательности, используя операцию `Range`, описанную в предыдущей главе, и соединил их вместе с помощью операции `Concat`, также описанной в предыдущей главе.

Внимание! На выполнение этого примера потребуется немало времени. На моей машине с процессором P4 и 1 Гбайт памяти для этого потребовалось две с половиной минуты.

Если вы запустите тот же пример, используя операцию `Count`, то получите исключение. Теперь я попробую пример вызова второго прототипа. Для этого примера я использую тот же базовый подход, что и для предыдущего, за исключением того, что специфицирую `predicate`, который возвратит `true` только для целых чисел больше 1 и меньше 4. То есть — для элементов 2 и 3. Поскольку у меня две последовательности с одинаковыми значениями, я должен получить в результате значение счетчика 4, как показано в листинге 5.50.

Листинг 5.50. Пример вызова второго прототипа `LongCount`

```
long count = Enumerable.Range(0, int.MaxValue).
    Concat(Enumerable.Range(0, int.MaxValue)).LongCount(n => n > 1 && n < 4);
Console.WriteLine(count);
```

Этот код почти такой же, как и код предыдущего примера, за исключением того, что дополнительно я специфицировал `predicate`. И на его выполнение потребуется даже больше времени, чем для предыдущего.

Результат работы этого кода выглядит так:

4

Sum

Операция **Sum** возвращает сумму числовых значений, содержащихся в элементах последовательности.

Прототипы

У операции имеются два прототипа, описанные ниже.

Первый прототип Sum

```
public static Numeric Sum(
    this IEnumerable<Numeric> source);
```

Тип **Numeric** должен быть одним из **int**, **long**, **double** или **decimal**, либо одним из их допускающих **null** эквивалентов: **int?**, **long?**, **double?** или **decimal?**.

Первый прототип операции **Sum** возвращает сумму всех элементов входной последовательности **source**.

Пустая последовательность приведет к возврату нуля. Операция **Sum** не включает значения **null** в результат для числовых типов, допускающих **null**.

Второй прототип **Sum** ведет себя так же, как и первый, но суммирует только те значения входной последовательности, которые выбраны делегатом метода **selector**.

Второй прототип Sum

```
public static Numeric Sum<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

Исключения

Исключение **ArgumentNullException** генерируется, если любой из аргументов равен **null**. Исключение **OverflowException** генерируется, если сумма оказывается слишком большой, чтобы уместиться в тип **Numeric**, если тип **Numeric** отличается от **decimal** или **decimal?**. Если же **Numeric** — именно **decimal** или **decimal?**, возвращается положительная или отрицательная бесконечность.

Примеры

Я начну с примера применения первого прототипа, который показан в листинге 5.51. Сначала я генерирую последовательность целых чисел, используя операцию **Range**, а затем использую операцию **Sum** для суммирования их.

Листинг 5.51. Пример вызова первого прототипа Sum

```
IEnumerable<int> ints = Enumerable.Range(1, 10);
foreach (int i in ints)
    Console.WriteLine(i);
Console.WriteLine("<!--");
int sum = ints.Sum();
Console.WriteLine(sum);</pre>


---



```

Результат выглядит так:

1
2
3

```

4
5
6
7
8
9
10
-- 
55

```

Теперь я попробую второй прототип, как показано в листинге 5.52. Для этого примера я использую мой общий класс `EmployeeOptionEntry` и просуммирую опции для всех сотрудников.

Листинг 5.52. Пример вызова второго прототипа `Sum`

```

IEnumerable<EmployeeOptionEntry> options =
    EmployeeOptionEntry.GetEmployeeOptionEntries();
long optionsSum = options.Sum(o => o.optionsCount);
Console.WriteLine("The sum of the employee options is: {0}", optionsSum);

```

Вместо того чтобы пытаться суммировать весь элемент, что не имело бы смысла в данном примере, поскольку речь идет об объекте сотрудника, я могу использовать элемент `selector` второго прототипа, чтобы извлечь только те члены, которые меня интересуют — в данном случае член `optionsCount`. Результат этого кода выглядит так:

The sum of the employee options is: 51504

Min

Операция `Min` возвращает минимальное значение входной последовательности.

Прототипы

Эта операция имеет четыре прототипа, описанные ниже.

Первый прототип `Min`

```

public static Numeric Min(
    this IEnumerable<Numeric> source);

```

Тип `Numeric` должен быть одним из `int`, `long`, `double` или `decimal`, либо одним из их допускающих `null` эквивалентов: `int?`, `long?`, `double?` или `decimal?`.

Первый прототип операции `Min` возвращает элемент с минимальным числовым значением из входной последовательности `source`. Если тип элемента реализует интерфейс `IComparable<T>`, этот интерфейс будет использован для сравнения элементов. Если же элемент не реализует интерфейс `IComparable<T>`, будет использован необобщенный интерфейс `IComparable`.

Пустая последовательность либо последовательность, состоящая только из значений `null`, вернет `null`.

Второй прототип операции `Min` ведет себя подобно первому, за исключением того, что он предназначен для нечисловых типов.

Второй прототип `Min`

```

public static T Min<T>(
    this IEnumerable<T> source);

```

Третий прототип предназначен для значений `Numeric` и подобен первому, но с возможностью спецификации делегата метода `selector`, который позволяет сравнивать

члены каждого элемента входной последовательности в процессе поиска минимального значения и возвращать это минимальное значение.

Третий прототип Min

```
public static T Min<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

Четвертый прототип предназначен для нечисловых типов и подобен второму, но с возможностью спецификации делегата метода `selector`, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска минимального значения и возвращать это минимальное значение.

Четвертый прототип Min

```
public static S Min<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`. Исключение `InvalidOperationException` генерируется, если последовательность `source` пуста для *Numeric*-версий прототипов, если `T` не допускает значений `null`, такой как `int`, `long`, `double` или `decimal`. Если типы допускают `null`, подобные `int?`, `long?`, `double?` или `decimal?`, то вместо этого операция возвращает `null`.

Примеры

В примере использования прототипа `Min`, показанном в листинге 5.53, я объявляю массив целых чисел и возвращаю минимальное из них.

Листинг 5.53. Пример вызова первого прототипа Min

```
int[] myInts = new int[] { 974, 2, 7, 1374, 27, 54 };
int minInt = myInts.Min();
Console.WriteLine(minInt);
```

Это очень тривиальный пример, и его результат выглядит так:

2

Для примера использования второго прототипа, показанного в листинге 5.54, я просто вызываю операцию `Min` на моем стандартном массиве `presidents`. Он должен вернуть элемент с минимальным значением в алфавитном порядке.

Листинг 5.54. Пример вызова второго прототипа Min

```
string[] presidents = {
    "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string minName = presidents.Min();
Console.WriteLine(minName);
```

Это пример выдает следующий результат:

Adams

Хотя это тот же вывод, что и при вызове операции `First`, но такое совпадение объясняется лишь тем, что массив `presidents` упорядочен по алфавиту. Если бы его элементы были неупорядочены либо располагались в каком-то другом порядке, то результат все равно был бы `Adams`.

Для примера третьего прототипа операции `Min` я использую мой общий класс `Actor`, чтобы найти самую раннюю дату рождения актера посредством вызова `Min` на годах рождения.

Листинг 5.55 содержит соответствующий код с вызовом операции `Min`.

Листинг 5.55. Пример вызова третьего прототипа `Min`

```
int oldestActorAge = Actor.GetActors().Min(a => a.birthYear);
Console.WriteLine(oldestActorAge);
```

Запустив этот пример, мы обнаружим, что год рождения актера, перенесшего максимальное число пластических операций (я имею в виду, наиболее старшего из всех) будет таким:

1960

Для примера использования четвертого прототипа `Min`, показанного в листинге 5.56, я получу фамилию актера, которая находится первой в алфавитном порядке, снова используя мой класс `Actor`.

Листинг 5.56. Пример вызова четвертого прототипа `Min`

```
string firstAlphabetically = Actor.GetActors().Min(a => a.lastName);
Console.WriteLine(firstAlphabetically);
```

И наш "Оскар" достается:

Bullock

Max

Операция `Max` возвращает максимальное значение из входной последовательности.

Прототипы

У этой операции имеются четыре прототипа, описанные ниже.

Первый прототип `Max`

```
public static Numeric Max(
    this IEnumerable<Numeric> source);
```

Тип `Numeric` должен быть одним из `int`, `long`, `double` или `decimal`, либо одним из их допускающих `null` эквивалентов: `int?`, `long?`, `double?` или `decimal?`.

Первый прототип операции `Max` возвращает элемент с максимальным числовым значением из входной последовательности `source`. Если тип элемента реализует интерфейс `IComparable<T>`, этот интерфейс будет использован для сравнения элементов. Если же элемент не реализует интерфейс `IComparable<T>`, будет использован необобщенный интерфейс `IComparable`.

Пустая последовательность либо последовательность, состоящая только из значений `null`, вернет `null`.

Второй прототип операции `Max` ведет себя подобно первому, за исключением того, что он предназначен для нечисловых типов.

Второй прототип Max

```
public static T Max<T>(
    this IEnumerable<T> source);
```

Третий прототип предназначен для значений *Numeric* и подобен первому, но с возможностью специфицирования делегата метода *selector*, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска максимального значения и возвращать это максимальное значение.

Третий прототип Max

```
public static T Max<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

Четвертый прототип предназначен для нечисловых типов и подобен второму, но с возможностью специфицирования делегата метода *selector*, который позволяет сравнивать члены каждого элемента входной последовательности в процессе поиска максимального значения и возвращать это максимальное значение.

Четвертый прототип Max

```
public static S Max<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

Исключения

Исключение *ArgumentNullException* генерируется, если любой из аргументов равен *null*. Исключение *InvalidOperationException* генерируется, если последовательность *source* пуста для *Numeric*-версий прототипов, если *T* не допускает значений *null*, такой как *int*, *long*, *double* или *decimal*. Если типы допускают *null*, подобные *int?*, *long?*, *double?* или *decimal?*, то вместо этого операция возвращает *null*.

Примеры

Для примера прототипа *Max*, показанного в листинге 5.57, я объявляю массив целых чисел и возвращаю максимальное из них.

Листинг 5.57. Пример вызова первого прототипа Max

```
int[] myInts = new int[] { 974, 2, 7, 1374, 27, 54 };
int maxInt = myInts.Max();
Console.WriteLine(maxInt);
```

Результат выглядит так:

1374

Для примера второго прототипа, показанного в листинге 5.58, я просто вызову операцию *Max* на моем стандартном массиве *presidents*.

Листинг 5.58. Пример вызова второго прототипа Max

```
string[] presidents = {"Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland",
    "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield",
    "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",
    "Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",
    "Monroe", "Nixon", "Pierce", "Polk", "Reagan", "Roosevelt", "Taft",
    "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};
string maxName = presidents.Max();
Console.WriteLine(maxName);
```

Это даст следующий результат:

Wilson

Опять-таки, как я уже упоминал, описывая эквивалентный пример операции `Min`, хотя этот пример выдает тот же результат, что выдал бы и операция `Last`, это только потому, что массив `presidents` упорядочен по алфавиту.

Для примера третьего прототипа операции `Max` я использую мой общий класс `Actor`, чтобы найти самый последний год рождения актера, вызвав `Max` на годах рождения.

Код, вызывающий этот вариант операции `Max`, представлен в листинге 5.59.

Листинг 5.59. Пример вызова третьего прототипа `Max`

```
int oldestActorAge = Actor.GetActors().Max(a => a.birthYear);
Console.WriteLine(oldestActorAge);
```

Запустив пример, узнаем год рождения самого молодого из моего актерского класса:

1968

Для примера четвертого прототипа `Max`, показанного в листинге 5.60, я получу фамилию актера, которая идет последней в алфавитном порядке с использованием того же класса `Actor`.

Листинг 5.60. Пример вызова четвертого прототипа `Max`

```
string firstAlphabetically = Actor.GetActors().Max(a => a.lastName);
Console.WriteLine(firstAlphabetically);
```

Результат будет таким:

Wilson

Вот это да — я получил тот же результат, что и во втором примере, где использовал массив имен президентов США. От этого совпадения бросает в дрожь!

Average

Операция `Average` возвращает среднее арифметическое числовых значений элементов входной последовательности.

Прототипы

У этой операции имеются два прототипа, описанные ниже.

Первый прототип `Average`

```
public static Result Average(
    this IEnumerable<Numeric> source);
```

Тип `Numeric` должен быть одним из `int`, `long`, `double` или `decimal`, либо одним из их допускающих `null` эквивалентов: `int?`, `long?`, `double?` или `decimal?`. Если тип `Numeric` — `int` или `long`, то тип `Result` будет `double`. Если же тип `Numeric` — `int?` или `long?`, то тип `Result` будет `double?`. В противном случае тип `Result` совпадет с типом `Numeric`.

Первый прототип операции `Average` перечисляет входную последовательность `source` элементов типа `Numeric`, вычисляя их среднее значение.

Второй прототип операции `Average` перечисляет входную последовательность `source` элементов и определяет среднее значение членов элементов, возвращаемых функцией `selector` для каждого элемента входной последовательности.

Второй прототип Average

```
public static Result Average<T>(
    this IEnumerable<T> source,
    Func<T, Numeric> selector);
```

Исключения

Исключение ArgumentNullException генерируется, если любой из аргументов равен null. Исключение OverflowException генерируется, если сумма усредняемых значений превышает емкость long для типов Numeric — int, int?, long и long?.

Примеры

Начну с примера первого прототипа, показанного в листинге 5.61. Для этого примера я использовал операцию Range, чтобы создать последовательность целых чисел, а затем подсчитать их среднее значение.

Листинг 5.61. Пример вызова первого прототипа Average

```
IEnumerable<int> intSequence = Enumerable.Range(1, 10);
Console.WriteLine("Here is my sequence of integers:");
foreach (int i in intSequence)
    Console.WriteLine(i);
double average = intSequence.Average();
Console.WriteLine("Here is the average: {0}", average);
```

Результат этого примера:

```
Here is my sequence of integers:
1
2
3
4
5
6
7
8
9
10
Here is the average: 5.5
```

Теперь я попробую воспользоваться вторым прототипом, который обращается к членам элементов. Для этого примера, показанного в листинге 5.62, я использую мой общий класс EmployeeOptionEntry.

Листинг 5.62. Пример вызова второго прототипа Average

```
IEnumerable<EmployeeOptionEntry> options =
EmployeeOptionEntry.GetEmployeeOptionEntries();
Console.WriteLine("Here are the employee ids and their options:");
foreach (EmployeeOptionEntry eo in options)
    Console.WriteLine("Employee id: {0}, Options: {1}", eo.id, eo.optionsCount);
// Теперь получим среднее значений опций.
double optionAverage = options.Average(o => o.optionsCount);
Console.WriteLine("The average of the employee options is: {0}", optionAverage);
```

Сначала я извлекаю объекты EmployeeOptionEntry. Затем выполняю перечисление последовательности объектов и отображаю каждый из них. В конце вычисляю среднее значение и показываю его. Результат работы этого кода выглядит следующим образом:

```

Here are the employee ids and their options:
Employee id: 1, Options: 2
Employee id: 2, Options: 10000
Employee id: 2, Options: 10000
Employee id: 3, Options: 5000
Employee id: 2, Options: 10000
Employee id: 3, Options: 7500
Employee id: 3, Options: 7500
Employee id: 4, Options: 1500
Employee id: 101, Options: 2
The average of the employee options is: 5722.6666666667

```

Aggregate

Операция `Aggregate` выполняет специфицированную пользователем функцию на каждом элементе входной последовательности, передавая значение, возвращенное этой функцией для предыдущего элемента, и возвращая ее значение для последнего элемента.

Прототипы

У этой операции имеются два прототипа, описанные ниже.

Первый прототип Aggregate

```

public static T Aggregate<T>(
    this IEnumerable<T> source,
    Func<T, T, T> func);

```

В этой версии прототипа операция `Aggregate` перечисляет каждый элемент входной последовательности `source`, вызывая делегат метода `func` на каждом из них, передавая значение, возвращенное им же на предыдущем элементе, и, наконец, помещая значение, возвращенное `func`, во внутренний аккумулятор, который затем передается на обработку следующего элемента. Первый элемент сам передается в качестве входного значения делегату метода `func`.

Второй прототип операции `Aggregate` ведет себя так же, как и первая версия, за исключением того, что принимает также начальное значение, служащее входным при первоначальном вызове делегата метода `func`, вместо самого первого элемента.

Второй прототип Aggregate

```

public static U Aggregate<T, U>(
    this IEnumerable<T> source,
    U seed,
    Func<U, T, U> func);

```

Исключения

Исключение `ArgumentNullException` генерируется, если любой из аргументов равен `null`. Исключение `InvalidOperationException` генерируется, если входная последовательность `source` пуста, только для первого прототипа `Aggregate`, когда не указывается никакого начального значения.

Примеры

Я начну с примера первого прототипа, показанного в листинге 5.63. В этом примере я вычисляю факториал числа 5. Факториал — это произведение всех положительных целых чисел, меньших или равных заданному. Итак, 5! будет равно $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$. Похоже, что я могу использовать операции `Range` и `Aggregate` для этого вычисления.

Листинг 5.63. Пример вызова первого прототипа Aggregate

```

int N = 5;
IEnumerable<int> intSequence = Enumerable.Range(1, N);
// Выведу выходную последовательность, чтобы все могли ее увидеть.
foreach (int item in intSequence)
    Console.WriteLine(item);
// Теперь вычислю факториал и покажу его.
// av == aggregated value, e == element
int agg = intSequence.Aggregate((av, e) => av * e);
Console.WriteLine("{0}! = {1}", N, agg);

```

В приведенном коде я генерирую последовательность, содержащую целые числа от 1 до 5, используя для этого операцию Range. После отображения всех элементов сгенерированной последовательности я вызываю операцию Aggregate, передав ей лямбда-выражение, умножающее переданное агрегатное значение на сам переданный элемент. Результат выглядит так:

```

1
2
3
4
5
5! = 120

```

Внимание! Вы должны быть осторожны, используя эту версию операции Aggregate, чтобы первый элемент не был обработан дважды, поскольку он передается в качестве входного для первого элемента. В предыдущем примере первый вызов лямбда-выражения func должен получить 1 и 1. Поскольку я только перемножаю два этих значения, и оба они равны единице, никакого вредного побочного эффекта нет. Но если бы мне нужно было складывать два значения, то в итоговую сумму первый элемент был бы включен дважды.

Для примера второго прототипа, показанного в листинге 5.64, я написал собственную версию операции Sum.

Листинг 5.64. Пример вызова второго прототипа Aggregate

```

IEnumerable<int> intSequence = Enumerable.Range(1, 10);
// Просто выведу последовательность, чтобы все ее увидели.
foreach (int item in intSequence)
    Console.WriteLine(item);
Console.WriteLine("--");
// Теперь вычисли сумму и покажу ее.
int sum = intSequence.Aggregate(0, (s, i) => s + i);
Console.WriteLine(sum);

```

Обратите внимание, что я передал 0 в качестве начального значения при вызове операции Aggregate. И вот результат:

```

1
2
3
4
5
6
7
8
9

```

```
10
--  
55
```

Как видите, я получил в точности тот же результат, что и при вызове `Sum` в листинге 5.51.

Резюме

Ох, просто голова закружилась! Надеюсь, я растерял не слишком много читателей к этому моменту... Я знаю, что большая часть этой и предыдущей глав получилась немножко суховатой, но здесь содержатся сущности, составляющие LINQ. Надеюсь, я раскрыл каждую операцию, которая вам может понадобиться. Большая часть эффективности LINQ зависит от вашего знания операций и понимания того, что они делают. Даже если вы не запомните все вариации каждой операции, просто знание об их существовании и о том, что они делают, может здорово пригодиться.

Надеюсь, что мой обзор `LINQ to Objects` и стандартных операций запросов позволил вам убедиться в мощи и удобстве LINQ, когда речь идет об опросе данных любого типа, хранящихся коллекциях, которые расположены в памяти.

Предоставляя около 50 операций, `LINQ to Objects` без сомнения позволит сделать ваш код опроса данных более согласованным, надежным и легким в написании.

Я еще раз хочу подчеркнуть, что большинство стандартных операций запросов работают на коллекциях, реализующих интерфейс `IEnumerable<T>`, что исключает унаследованные старые коллекции C#. Я точно знаю, что некоторые читатели не обратят внимание на этот факт и будут разочарованы, поскольку им покажется, что их унаследованный код с `ArrayList` не поддается обработке запросами нового типа. Если вы относитесь к их числу, прочтите еще раз об операциях `Cast` и `OfType`.

Теперь, когда, как я надеюсь, вы достигли определенной степени понимания `LINQ to Objects` и того, что LINQ может сделать для вас, самое время изучить применение LINQ для запросов и генерации XML. Эта функциональность называется `LINQ to XML`, и не случайно именно так называется следующая часть книги.



ЧАСТЬ III

LINQ to XML

В этой части...

Глава 6. Введение в LINQ to XML

Глава 7. Интерфейс LINQ to XML API

Глава 8. Операции LINQ to XML

Глава 9. Дополнительные возможности XML

ГЛАВА 6

Введение в LINQ to XML

И так, 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 отобразжу их имена и фамилии.
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 `InnerText` 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! Абраcadабра, `<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 все¹?”— спросил я. Мне сказали, чтобы я не беспокоился: у них есть приложение с графическим интерфейсом пользователя, специально предназначенное для чтения этих файлов. Все этоказалось 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 вели 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 поле из строки текстового файла на основе либо позиций символов 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 позволяют 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 следующего 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. Яшел 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, предназначеннное 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.

ГЛАВА 7

Интерфейс LINQ to XML API

В предыдущей главе я продемонстрировал создание документа XML с использованием программного интерфейса W3C DOM XML API, и вы увидели, насколько неуклюжим может быть этот API. Также я показал вам некоторые приемы, которые предназначены облегчить неудобства, вызванные им.

Также я приоткрыл вам маленький секрет LINQ, который состоит в том, что LINQ касается не только запросов данных, но также и XML. Я сообщил вам, что на горизонте появился новый XML API, и это — LINQ to XML API.

Итак, существует лучший или, по крайней мере, более простой способ конструировать, проходить, манипулировать и опрашивать XML, который называется LINQ to XML. В этой главе я покажу, как конструировать, проходить, манипулировать и опрашивать документы XML, используя LINQ to XML API, а также как выполнять поиски в объекте XML.

Для примеров этой главы я создам консольное приложение. Однако прежде чем вы сможете воспользоваться этим новым API-интерфейсом, вам нужно добавить в ваш проект ссылку на сборку System.Xml.Linq, если вы еще этого не сделали.

Необходимые пространства имен

Примеры этой главы используют пространства имен System.Linq, System.Xml.Linq и System.Collections.Generic. Поэтому вы должны добавить директивы using для этих пространств имен в ваш код, если не сделали этого раньше:

```
using System.Linq;
using System.Xml.Linq;
using System.Collections.Generic;
```

В дополнение к этим пространствам имен, если вы загрузите сопровождающий код то увидите там, что я также добавил директиву using для пространства имен System.Diagnostics. Это не обязательно, если вы собираетесь набирать самостоятельно примеры этой главы. Это нужно лишь для загружаемого сопровождающего кода.

Существенные усовершенствования дизайна API

После нескольких лет существования с W3C XML DOM API от Microsoft специалистами этой корпорации было выделено несколько ключевых областей, в которых проявляется неудобство, сложность или слабость оригинального API. Чтобы побороть эти проблемы, были выделены следующие точки приложения усилий:

- конструирование деревьев XML;
- центральная роль документа;
- пространства имен и префиксы;
- извлечение значений узлов.

Каждый из этих проблемных доменов представлял собой препятствия в работе с XML. Они не только вызывали "разбухание" кода работы с XML, часто непреднамеренно затеняя этот код, но преодолеть их было необходимо для того, чтобы обеспечить гладкую работу LINQ с XML. Например, если вы хотите использовать проекцию, чтобы вернуть XML из запроса LINQ, проблемой была невозможность создания экземпляра элемента операцией new. Такое ограничение существующего XML API должно было быть преодолено, чтобы обеспечить практическую работу LINQ с XML. Давайте рассмотрим каждую из перечисленных проблемных областей, а также способ их разрешения в новом программном интерфейсе LINQ to XML API.

Конструирование деревьев XML было упрощено функциональным конструированием

При чтении кода первого примера в предыдущей главе (в листинге 6.1) сразу становится ясно, что, глядя на этот код создания дерева XML, очень сложно определить схему XML. После создания документа XML мы должны создать некоторого типа узел XML, такой как элемент, установить его значение и добавить к родительскому элементу. Однако каждый из этих трех шагов должен быть выполнен отдельно с использованием W3C DOM API. Это приводит к неясной схеме и объемному коду. Этот API не предусматривает поддержки создания элемента или любого типа узла в определенном месте дерева XML по отношению к его родителю и его инициализации за одну операцию.

LINQ to XML API не только предоставляет ту же возможность создания дерева XML, что и W3C DOM API, но также предлагает новую технику, называемую *функциональным конструированием*, для создания дерева XML. Функциональное конструирование позволяет схеме диктовать то, как конструируются объекты XML и инициализируются их значения, и все это — одновременно, в единственном операторе. API достигает этого за счет предоставления конструкторов новых XML-объектов, которые принимают в качестве параметров как отдельные объекты, так и их множества, специфицируя их значения. Тип добавляемого объекта или объектов определяет то, где именно в схеме они располагаются. Общий шаблон выглядит так:

```
XMLOBJECT o =
    new XMLOBJECT(OBJECTNAME,
                  XMLOBJECT1,
                  XMLOBJECT2,
                  ...
                  XMLOBJECTN);
```

На заметку! Приведенный фрагмент является просто псевдокодом, предназначенным для иллюстрации шаблона. Ни один из классов, присутствующих в нем, не существует на самом деле; они лишь представляют некоторый концептуальный абстрактный класс XML.

Если вы добавляете к элементу, реализованному классом XElement, атрибут XML, который реализован классом XAttribute из LINQ to XML, этот атрибут становится атрибутом этого элемента. Например, если в предыдущем псевдокоде XMLOBJECT1 добавляется ко вновь созданному XMLOBJECT по имени o, и o является XElement, а XMLOBJECT1 есть XAttribute, то XMLOBJECT1 становится атрибутом XElement o.

Если вы добавляете XElement к XElement, то добавляемый XElement становится дочерним элементом того, к которому он добавлен. Поэтому, например, если XMLOBJECT1 — элемент и о — элемент, то XMLOBJECT1 становится дочерним элементом о.

При создании экземпляра объекта XMLOBJECT, как показано в приведенном псевдокоде, мы можем специфицировать его содержимое, указав от 1 до N объектов XMLOBJECT. Как вы узнаете далее в разделе “Создание текста с помощью XText”, можно даже специфицировать его содержимое, включающее строку, поскольку строка будет автоматически преобразована для вас в XMLOBJECT.

Все это совершенно логично и составляет суть функционального конструирования. Пример приведен в листинге 7.1.

Листинг 7.1. Использование функционального конструирования для создания схемы XML

```
XElement xBookParticipant =
    new XElement("BookParticipant",
        new XElement("FirstName", "Joe"),
        new XElement("LastName", "Rattz"));
Console.WriteLine(xBookParticipant.ToString());
```

Обратите внимание, что при конструировании элемента по имени BookParticipant я передал два объекта XElement в качестве его значения, каждый из которых стал его дочерним элементом. Также обратите внимание, что при конструировании элементов FirstName и LastName вместо спецификации дочерних объектов, как это было сделано при конструировании элемента BookParticipant, я указал просто текстовые значения элементов. И вот результат работы этого кода:

```
<BookParticipant>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
</BookParticipant>
```

Обратите внимание, насколько проще теперь инициализировать в коде схему XML. Также отметьте, насколько менее многословным стал код, чем код первого примера из листинга 6.1 предыдущей главы. Код LINQ to XML API полностью заменяет код из листинга 6.1, создающий дерево XML, будучи при этом существенно короче, как доказывает листинг 7.2.

Листинг 7.2. Создание того же дерева XML, что и в листинге 6.1, намного более коротким кодом

```
XElement xBookParticipants =
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham")));
Console.WriteLine(xBookParticipants.ToString());
```

Как видите, здесь намного меньше кода пришлось создать, и намного меньше впоследствии сопровождать. К тому же схему намного легче понять, просто прочитав этот код.

И вот вывод:

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

Есть еще одно преимущество нового API, которое наглядно проявляется в результате этого примера. Обратите внимание, что вывод сформатирован так, что он выглядит как дерево XML. Если я выведу дерево XML, созданное в листинге 6.1, оно будет выглядеть следующим образом:

```
<BookParticipants><BookParticipant type="Author"><FirstName>Joe</FirstName>...
```

Что вам легче читать? В следующей главе, когда речь пойдет о выполнении запросов, производящих вывод XML, вы убедитесь в незаменимости функционального конструирования.

Центральная роль элемента вместо документа

В оригинальном W3C DOM API вы не могли просто создать элемент XML — `Xmlelement`; вы должны были иметь документ XML — `Xmldocument`, из которого создавать его. Если бы вы попытались создать экземпляр `Xmlelement` вроде следующего:

```
Xmlelement xmlBookParticipant = new Xmlelement("BookParticipant");
```

то получили бы ошибку компиляции:

```
'System.Xml.Xmlelement.Xmlelement(string, string, string, System.Xml.Xmldocument)'  
is inaccessible due to its protection level  
'System.Xml.Xmlelement.Xmlelement(string, string, string, System.Xml.Xmldocument)'  
не доступен из-за своего уровня защиты
```

С помощью W3C DOM API вы могли создать `Xmlelement` только вызовом метода `Xmldocument` по имени `CreateElement`, как показано ниже:

```
Xmldocument xmlDoc = new Xmldocument();  
Xmlelement xmlBookParticipant = xmlDoc.CreateElement("BookParticipant");
```

Этот код компилируется успешно. Но часто это неудобно — обязательно создавать документ XML, когда вы просто хотите создать элемент XML. Новый LINQ-оснащенный XML API позволяет вам создать экземпляр элемента без создания документа XML:

```
XElement xeBookParticipant = new XElement("BookParticipant");
```

XML-элементы — не единственный тип узлов, подчиняющихся этому ограничению W3C DOM. Атрибуты, комментарии, разделы CData, инструкции обработки и ссылки на сущности — все это должно было создаваться из документа XML. К счастью, LINQ to XML API дает возможность непосредственно создавать каждый из этих объектов “на лету”.

Конечно, ничто не мешает вам создавать XML-документы с помощью нового API. Например, вы можете создать XML-документ и добавить в него элемент `BookParticipants`, а в последний — `BookParticipant`, как показано в листинге 7.3.

Листинг 7.3. Использование LINQ to XML для создания документа XML и добавления к нему некоторой структуры

```
XDocument xDocument =
    new XDocument(
        new XElement("BookParticipants",
            new XElement("BookParticipant",
                new XAttribute("type", "Author"),
                new XElement("FirstName", "Joe"),
                new XElement("LastName", "Rattz"))));
Console.WriteLine(xDocument.ToString());
```

Нажатие <Ctrl+F5> приведет к следующему результату:

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

XML, произведенный предыдущим кодом, очень похож на XML, который был создан в листинге 6.1, за исключением того, что я добавил только один BookParticipant вместо двух. Однако этот код намного более читабелен, благодаря новым средствам функционального конструирования. К тому же, глядя на этот код, можно легко проследить схему. Теперь, когда документы XML не обязательны, я могу пропустить документ XML и получить тот же результат, как показано в листинге 7.4.

Листинг 7.4. Пример, аналогичный предыдущему, но без документа XML

```
XElement xElement =
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")));
Console.WriteLine(xElement.ToString());
```

Запуск этого кода даст в точности тот же результат, что и предыдущий пример:

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

В дополнение к созданию деревьев XML без документов XML вы можете делать большинство тех же вещей, которые требуют документов, таких как чтение XML из файла и сохранение его в файл.

Имена, пространства имен и префиксы

Чтобы исключить путаницу, связанную с именами, пространствами имен и префиксами пространств имен, последние опущены; т.е. изъяты из API. С помощью LINQ to XML API префиксы пространства имен разворачиваются на вводе и возвращаются в выводе. Внутри они не существуют.

Пространство имен используется в XML с целью уникальной идентификации схемы XML для некоторой части дерева XML. URI используются в качестве пространств имен XML, потому что они уже уникальны для каждой организации. В некоторых из моих примеров кода я создавал дерево XML следующим образом:

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

Любой код, обрабатывающий эти данные XML, будет написан в предположении, что узел BookParticipants может содержать множество узлов BookParticipant, каждый из которых имеет атрибут type, а также узлы FirstName и LastName. Но что, если коду также придется обрабатывать XML из другого источника, в котором также окажется узел BookParticipants, но с отличающейся внутренней схемой? Пространство имен известит код о том, как должна выглядеть схема, тем самым позволяя ему соответствующим образом обработать XML.

В XML каждый элемент должен иметь имя. Когда элемент создается, если его имя специфицировано в конструкторе, оно неявно преобразуется из string в объект XName. Объект XName состоит из пространства имен — объекта XNamespace — и своего локального имени, того, которое вы указали. Поэтому, например, вы можете создать элемент BookParticipants следующим образом:

```
XElement xBookParticipants = new XElement("BookParticipants");
```

Когда вы создаете элемент, объект XName получает пустое пространство имен и локальное имя BookParticipants. Если вы станете отлаживать эту строку кода и просмотрите переменную xBookParticipants в окне слежения, то увидите, что ее член Name установлен в {BookParticipants}. Если же вы развернете член Name, то увидите, что он содержит член LocalName, установленный в BookParticipants, и член по имени Namespace, который будет пустым — {}. В данном случае пространства имен нет.

Чтобы специфицировать пространство имен, вам нужно просто создать объект XNamespace и предварить им локальное имя следующим образом:

```
XNamespace nameSpace = "http://www.linqdev.com";
 XElement xBookParticipants = new XElement(nameSpace + "BookParticipants");
```

После этого, если вы станете просматривать элемент xBookParticipants в окне слежения отладчика, то увидите, что Name установлено в {{http://www.linqdev.com} BookParticipants}. Разворачивание члена Name покажет, что LocalName будет по-прежнему BookParticipants, а член Namespace установлен в {http://www.linqdev.com}.

Не обязательно в действительности использовать объект XNamespace для того, чтобы специфицировать пространство имен. Я мог бы задать его в жестко закодированном строковом литерале, как показано ниже:

```
XElement xBookParticipants = new XElement("{http://www.linqdev.com}" +
  "BookParticipants");
```

Обратите внимание, что я заключил пространство имен в фигурные скобки. Это указывает конструктору XElement на тот факт, что данная часть означает пространство имен. Если вы вновь просмотрите член Name объекта BookParticipants в окне слежения, то увидите, что член Name и встроенные в него члены LocalName и Namespace установлены идентично тем значениям, установленным в предыдущем примере, где я использовал объект XNamespace для создания элемента.

Имейте в виду, что при установке пространства имен простого указания URI вашей компании или домена организации может быть недостаточно для того, чтобы гарантировать уникальность. Это гарантирует лишь отсутствие коллизий с любой другой существующей организацией, которая соблюдает установленные соглашения названий пространств имен. Однако внутри организации могут случиться коллизии с любым другим подразделением, если в пространстве имен вы не укажете ничего помимо URI организации. И здесь весьма пригодится ваше знание организационной структуры вашего предприятия — его подразделений, департаментов и так далее... Лучше всего будет, если ваше пространство имен будет раскрывать весь путь до определенного уровня, находящегося под вашим контролем. Например, если вы работаете в LINQDev.com, и создаете схему для департамента кадров, которая будет содержать информацию пенсионного плана, ваше пространство имен может выглядеть так:

```
XNamespace nameSpace = "http://www.linqdev.com/humanresources/pension";
```

Поэтому для окончательного примера использования пространств имен я модифицирую код из листинга 7.2, включив в него пространство имен, как показано в листинге 7.5.

Листинг 7.5. Модифицированная версия листинга 7.2 с указанным пространством имен

```
XNamespace nameSpace = "http://www.linqdev.com";
 XElement xBookParticipants =
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham")));
    Console.WriteLine(xBookParticipants.ToString());
```

Нажатие <Ctrl+F5> приведет к выдаче следующего результата:

```
<BookParticipants xmlns="http://www.linqdev.com">
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Теперь любой код может прочесть это и узнать, что схема должна соответствовать той, то предоставлена LINQDev.com.

Чтобы иметь контроль над префиксами пространств имен, используйте атрибут XAttribute для создания префикса, как в листинге 7.6.

Листинг 7.6. Спецификация префикса пространства имен

```
XNamespace nameSpace = "http://www.linqdev.com";
 XElement xBookParticipants =
    new XElement(nameSpace + "BookParticipants",
        new XAttribute(XNamespace.Xmlns + "lingdev", nameSpace),
        new XElement(nameSpace + "BookParticipant"));
    Console.WriteLine(xBookParticipants.ToString());
```

В приведенном коде я специфицирую в качестве префикса пространства имен linqdev, и использую объект XAttribute для включения в схему спецификации префикса. Вот вывод этого кода:

```
<linqdev:BookParticipants xmlns:linqdev="http://www.linqdev.com">
  <linqdev:BookParticipant />
</linqdev:BookParticipants>
```

Извлечение значения узла

Если вы прочли первый пример кода из предыдущей главы, приведенный в листинге 6.1, и посмеялись над результатом (я надеюсь на это), то без сомнений, вам приходилось сталкиваться с той же проблемой, что помешала мне сразу получить результат, полученный позже, т.е. действительное значение узла. Если бы у меня не было опыта достаточно длительной работы с кодом XML DOM, я неизбежно бы столкнулся с ошибкой подобного рода (как в листинге 6.1), почти всегда забывая о необходимости дополнительного шага для получения значения узла.

LINQ to XML API решает эту проблему очень изящно. Во-первых, метод ToString элемента выводит саму строку XML, а не тип объекта, как это делается в W3C DOM API. Это очень удобно, когда вам нужен определенный фрагмент XML из некоторой точки дерева, к тому же имеет намного больше смысла, чем вывод типа объекта. Листинг 7.7 демонстрирует пример.

Листинг 7.7. Вызов метода ToString на элементе, производящий дерево XML

```
XElement name = new XElement("Name", "Joe");
Console.WriteLine(name.ToString());
```

Нажатие <Ctrl+F5> выдаст следующий результат:

```
<Name>Joe</Name>
```

Уже лучше. Но подождите, можно добиться еще лучшего результата. Конечно, дочерние узлы включены в вывод, и поскольку метод WriteLine не имеет явной перегрузки, принимающей XElement, он вызывает для вас метод ToString, показанный в листинге 7.8.

Листинг 7.8. Неявный вызов ToString методом Console.WriteLine для производства дерева XML

```
XElement name = new XElement("Person",
  new XElement("FirstName", "Joe"),
  new XElement("LastName", "Rattz"));
Console.WriteLine(name);
И вот результат:
<Person>
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</Person>
```

Что еще более важно — если вы приведете узел к типу данных, к которому может быть преобразовано его значение, то будет выведено само значение. Листинг 7.9 демонстрирует другой пример, но я также вывожу узел, приведенный к string.

Листинг 7.9. Приведение элемента к его типу данных выводит значение

```
XElement name = new XElement("Name", "Joe");
Console.WriteLine(name);
Console.WriteLine((string)name);
```

Вот результат этого кода:

```
<Name>Joe</Name>
Joe
```

Не правда ли, гладко? Но сколько за это придется платить? Существуют операции приведения для `string`, `int`, `int?`, `uint`, `uint?`, `long`, `long?`, `ulong`, `ulong?`, `bool`, `bool?`, `float`, `float?`, `double`, `double?`, `decimal`, `decimal?`, `TimeSpan`, `TimeSpan?`, `DateTime`, `DateTime?`, `GUID` и `GUID?`.

Листинг 7.10 демонстрирует пример нескольких других типов узлов.

Листинг 7.10. Другие типы узлов, извлеченные через приведение к типу значения узла

```
XElement count = new XElement("Count", 12);
Console.WriteLine(count);
Console.WriteLine((int)count);
XElement smoker = new XElement("Smoker", false);
Console.WriteLine(smoker);
Console.WriteLine((bool)smoker);
XElement pi = new XElement("Pi", 3.1415926535);
Console.WriteLine(pi);
Console.WriteLine((double)pi);
```

И вот, пожалуйста:

```
<Count>12</Count>
12
<Smoker>false</Smoker>
False
<Pi>3.1415926535</Pi>
3.1415926535
```

Это выглядит очень простым и интуитивно понятным. Похоже на то, что применение LINQ to XML API вместо W3C DOM API оставит в прошлом ошибки вроде той, что проявляется в листинге 6.1 из предыдущей главы.

В то время как все эти примеры упрощают получение значения элемента, все они представляют собой случаи приведения элемента к тому типу данных, который изначально имело значение этого элемента. Это не обязательно. Все, что нужно — чтобы значение элемента могло быть преобразовано к специфицированному типу данных. В листинге 7.11 показан пример, где начальный тип данных — `string`, но я получаю его значение как `bool`.

Листинг 7.11. Приведение узла к типу данных, отличающемуся от оригинального типа его значения

```
XElement smoker = new XElement("Smoker", "true");
Console.WriteLine(smoker);
Console.WriteLine((bool)smoker);
```

Поскольку я специфицировал значение элемента как `"true"`, и поскольку строка `"true"` может быть успешно преобразована в `bool`, этот код работает:

```
<Smoker>true</Smoker>
True
```

К сожалению то, как именно значения преобразуются, не специфицировано, но на самом деле для этой цели используются методы класса `System.Xml.XmlConvert`. Листинг 7.12 демонстрирует, как это происходит при приведении к `bool`.

Листинг 7.12. Приведение к bool вызывает метод System.Xml.XmlConvert.ToBoolean

```

try
{
    XElement smoker = new XElement("Smoker", "Tue");
    Console.WriteLine(smoker);
    Console.WriteLine((bool)smoker);
}
catch (Exception ex)
{
    Console.WriteLine(ex);
}

```

Обратите внимание, что я специально допустил опечатку в слове "True", чтобы инициировать исключение, которое покажет, где именно происходило приведение. Повезет ли мне? Нажмем <Ctrl+F5>, чтобы проверить.

```

<Smoker>Tue</Smoker>
System.FormatException: The string 'tue' is not a valid Boolean value.
at System.Xml.XmlConvert.ToBoolean(String s)
System.FormatException: Стока 'tue' не является допустимым булевским значением.
в System.Xml.XmlConvert.ToBoolean(String s)
...

```

Как видите, исключение было сгенерировано в методе `System.Xml.XmlConvert.ToBoolean`.

Объектная модель LINQ to XML

С новым программным интерфейсом LINQ to XML пришла новая объектная модель, содержащая множество новых классов, находящихся в пространстве имен `System.Xml.Linq`. Один из них — статический класс, в котором находятся расширяющие методы LINQ to XML — `Extensions`; еще два класса сравнения — `XNodeDocumentOrderComparer` и `XNodeEqualityComparer`; остальные используются для построения деревьев XML. Эти остальные классы показаны на диаграмме, представленной на рис. 7.1.

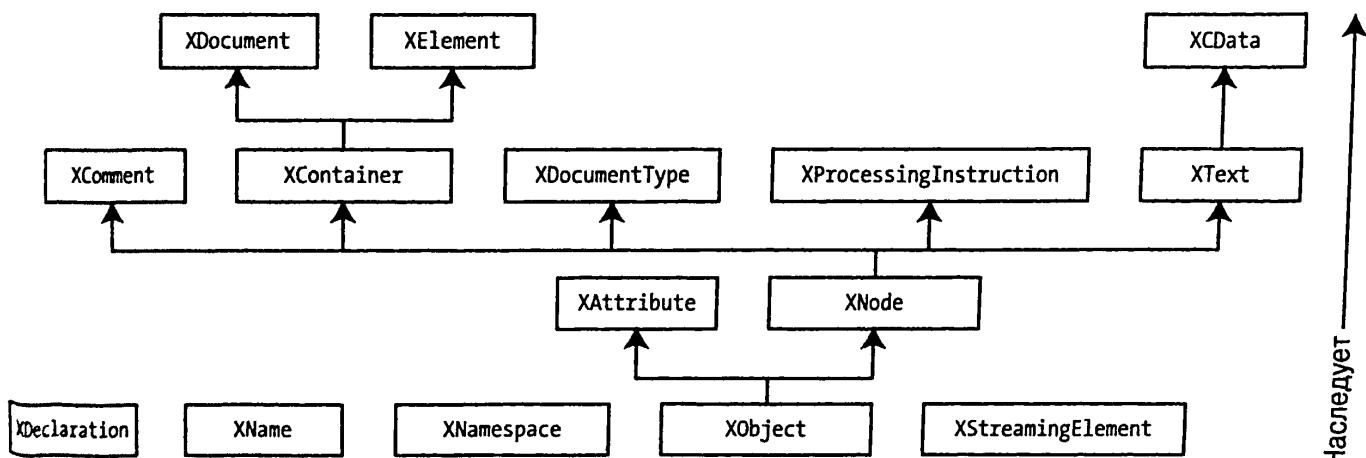


Рис. 7.1. Объектная модель LINQ to XML

Здесь следует отметить некоторые интересные вещи.

1. Три из этих классов — абстрактные: `XObject`, `XContainer` и `XNode`, так что вы никогда не будете конструировать их объектов.

2. Атрибут `XAttribute` не наследуется от узла `XNode`. Фактически, это вообще не узел, а совершенно другого типа класс, который на самом деле представляет пару “ключ-значение”.
3. Потоковые элементы `XStreamingElement` не имеют отношения наследования с элементом `XElement`.
4. Классы `XDocument` и `XElement` — единственные классы, имеющие узлы, унаследованные от `XNode`.

Все эти классы вы используете для построения ваших деревьев XML. Что более интересно — то, что вы используете класс `XElement`, поскольку, как я уже упоминал, в LINQ to XML API центральную роль играет элемент, в противоположность документу в W3C XML DOM.

Отложенное выполнение запросов, удаление узлов и “проблема Хэллоуина”

Этот раздел служит предупреждением, чтобы вы опасались некоторых подводных камней. Первый из них — *отложенное выполнение запросов*. Никогда не забывайте, что многие из операций LINQ откладывают выполнение запроса до тех пор, пока это не станет абсолютно необходимо, и это может вызвать потенциальные побочные эффекты.

Другая проблема — так называемая “*проблема Хэллоуина*” (Halloween problem), которая получила свое название потому, что впервые всплыла в дискуссии внутри небольшой группы экспертов именно в день этого праздника. К проблемам этого типа можно отнести почти любую проблему, которая проявляется при изменении тех данных в процессе итерации, которые затрагивают эту итерацию. Впервые она была обнаружена инженерами баз данных при работе над оптимизатором базы данных. Они столкнулись с ней, когда тестировали запрос, изменяющий значение столбца базы данных, который разрабатываемый оптимизатор использовал в качестве индекса. Их тестовый запрос должен был извлечь запись на основе индекса, созданного на одном из столбцов таблицы, и этот запрос должен был изменить значение этого столбца. Поскольку затронутый столбец определяет индексацию записи, эта запись затем появлялась дальше в списке записей, извлеченная тем же запросом и подвергаясь повторной обработке. Это приводило к бесконечному циклу, потому что при всяком извлечении ее из набора записей она обновлялась и перемещалась далее в наборе записей, где вновь извлекалась и обрабатывалась, и так бесконечно.

Вы сами наверняка сталкивались с “*проблемой Хэллоуина*”, даже не зная, что она так называется. Приходилось ли вам когда-либо работать с некоторого рода коллекций, выполняя итерацию по ней и удаляя из нее элемент, что приводило к прерыванию итерации или ее неправильному поведению? Я сам недавно сталкивался с этим, работая с крупным комплектом серверных элементов управления ASP.NET. Этот комплект включал серверный элемент `DataGrid`, и мне нужно было выборочно удалить из него записи. Выполняя итерацию по записям от начала до конца, я удалял нужные записи, но при этом терял текущую точку итерации. В результате оставались записи, которых должны были быть удалены, и удалялись те, которые не должны были. Я вызвал службу поддержки, и они предложили мне выполнять итерацию в обратном порядке. Это решило проблему.

С LINQ to XML вы, скорее всего, столкнетесь с этой проблемой при удалении узлов из дерева XML, хотя это может произойти в других случаях, так что вам следует иметь это в виду при кодировании. Рассмотрим пример, представленный в листинге 7.13.

Листинг 7.13. Намеренный вызов “проблемы Хэллоуина”

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
foreach ( XElement element in elements)
{
    Console.WriteLine("Removing {0} = {1} ...", element.Name, element.Value);
    element.Remove();
}
Console.WriteLine(xDocument);

```

В предыдущем коде я сначала строю документ XML. Затем я создаю последовательность элементов BookParticipant. По этой последовательности я буду выполнять перечисление, удаляя элементы. Затем я отображаю каждый элемент в последовательности, так что вы можете убедиться, что у меня действительно есть два элемента BookParticipant. Далее я вновь выполняю перечисление последовательности, отображая сообщение об удалении элемента, и удаляю элемент BookParticipant. Затем я отображаю результирующий документ XML.

Если “проблема Хэллоуина” не проявится, вы должны увидеть сообщение “Removing...” (“Удаление...”) дважды; и когда в конце будет отображен XML-документ, вы должны иметь пустой элемент BookParticipants. Вот результат:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Removing BookParticipant = JoeRattz ...
<BookParticipants>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>

```

Как и ожидалось, в последовательности есть два элемента BookParticipant, подлежащих удалению. Вы можете видеть, что первый из них — Joe Rattz — был удален. Однако не похоже, чтобы второй элемент был удален, и когда я отображаю результирующий документ XML, второй элемент BookParticipant все еще в нем присутствует. Перечисление ведет себя неправильно; налицо “проблема Хэллоуина”. Имейте в виду, что “проблема Хэллоуина” не всегда проявляется одинаково. Иногда перечисление может прекратиться раньше, чем должно; иногда оно генерирует исключения. Поведение варьируется в зависимости от того, что именно произошло.

Я знаю, что вы недоумеваете — каково же решение? Решение для данного случая заключается в кэшировании элементов и перечислении по кэшю вместо обычной тех-

ники перечисления, которая полагается на внутренние указатели, повреждаемые в процессе удаления или модификации элементов. Для данного примера я буду кэшировать последовательность элементов, используя стандартные операции запросов, специально предназначенные для кэширования, чтобы предотвратить проблемы с отложенным выполнением запроса. Я использую операцию `ToArray`. В листинге 7.14 показан тот же код, что и раньше, за исключением того, что я вызываю операцию `ToArray` и перечисляю его результат.

Листинг 7.14. Предотвращение “проблемы Хэллоуина”

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
foreach ( XElement element in elements.ToArray())
{
    Console.WriteLine("Removing {0} = {1} ...", element.Name, element.Value);
    element.Remove();
}
Console.WriteLine(xDocument);
```

Этот код идентичен предыдущему, за исключением того, что я вызываю операцию `ToArray` в финальном перечислении, где удаляю элементы. И вот результат:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Removing BookParticipant = JoeRattz ...
Removing BookParticipant = EwanBuckingham ...
<BookParticipants />
```

Обратите внимание, что на этот раз я получил два сообщения, информирующих об удалении элемента `BookParticipant`. Кроме того, когда я отображаю XML-документ после удаления, я получаю пустой элемент `BookParticipants`, потому что все его дочерние элементы удалены. “Проблема Хэллоуина” решена!

Создание XML

Как я уже упоминал, функциональное конструирование, предлагаемое `LINQ to XML API`, позволяет очень легко создавать дерево XML, особенно если сравнить с `W3C DOM API`. Рассмотрим теперь создание каждого из основных классов в `LINQ to XML API`.

Поскольку новый API ориентирован на элементы, и именно их вы будете создавать большей частью, сначала я опишу создание элементов с помощью класса `XElement`. Затем опишу остальные классы XML в алфавитном порядке.

Создание элементов с помощью XElement

Для начала вы должны помнить, что в этом новом API класс `XElement` — один из наиболее часто используемых. Рассмотрим создание экземпляра `XElement`. Класс `XElement` имеет несколько конструкторов, но мы рассмотрим два из них:

```
XElement(XName name, object content);
XElement(XName name, params object[] content);
```

Первый конструктор представляет простейший случай, когда элемент имеет текстовое значение и не имеет дочерних узлов. Этот случай показан в листинге 7.15.

Листинг 7.15. Создание элемента с использованием первого прототипа

```
XElement firstName = new XElement("FirstName", "Joe");
Console.WriteLine((string)firstName);
```

Первый аргумент конструктора — объект `XName`. Как упоминалось ранее, объект `XName` создается неявным преобразованием входного параметра `string` в `XName`. Второй аргумент — единственный объект, представляющий содержимое элемента. В данном случае содержимое — `string` со значением `"Joe"`. API “на лету” преобразует этот строковый литерал `"Joe"` в объект `XText`. Обратите внимание, что я использую преимущество новых средств извлечения значения узла, чтобы получить значение из переменной элемента `firstName`. То есть я привожу элемент к типу его значения, которым в данном случае является `string`. Таким образом, будет извлечено значение элемента `firstName`. И вот результат:

Joe

Тип данных единственного объекта содержимого очень гибок. Это тип данных объекта содержимого, который контролирует его отношение к элементу, к которому он добавлен. В табл. 7.1 описаны все допустимые типы данных объектов содержимого, и то, как они обрабатываются.

Помните, что даже несмотря на то, что значение элемента может быть сохранено как `string`, как это возможно для любого прочего типа (*any remaining type*)¹, такого как целое число, благодаря новым средствам извлечения значения вы можете получить его в оригинальном типе. Поэтому, например, если при создании объекта `XElement` вы специфицируете целое (`int`) в качестве объекта содержимого приведением узла к целому (`int`), то получите значение, преобразованное для вас в целое. Пока вы выполняете приведение к одному из типов, для которых предусмотрена операция приведения, и до тех пор, пока значение элемента может быть преобразовано в тип, к которому выполняется приведение, такая операция представляет простой способ получения значения элемента.

Второй конструктор `XElement`, показанный выше, подобен первому, за исключением того, что вы можете предоставить множество объектов для содержимого. Именно это делает функциональное конструирование столь мощным. Вам нужно лишь изучить листинги 7.1 и 7.2, чтобы увидеть примеры использования второго конструктора, где множество объектов содержимого передается конструктору `XElement`.

¹Этот термин объясняется в табл. 7.1.

Таблица 7.1. Таблица поведения LINQ to XML при вставке дочернего объекта в родительский

Тип данных объекта содержимого	Способ обработки
string	Объект <code>string</code> или строковый литерал автоматически преобразуется в объект <code>XText</code> и с этого момента обрабатывается как <code>XText</code> .
<code>XText</code>	Этот объект может иметь значение либо <code>string</code> , либо <code>XText</code> . Добавляется как дочерний узел элемента, но трактуется как текстовое содержимое элемента.
<code>XCData</code>	Этот объект может иметь значение либо <code>string</code> , либо <code>XCData</code> . Добавляется как дочерний узел элемента, но трактуется как <code>CData</code> -содержимое элемента.
<code>XElement</code>	Этот объект добавляется как дочерний элемент.
<code>XAttribute</code>	Этот объект добавляется как атрибут.
<code>XProcessingInstruction</code>	Этот объект добавляется как дочернее содержимое.
<code>XComment</code>	Этот объект добавляется как дочернее содержимое.
<code>IEnumerable</code>	Этот объект перечисляется, и обработка типов составляющих его объектов выполняется рекурсивно.
<code>null</code>	Этот объект игнорируется. Вы можете удивиться, зачем вообще может понадобиться передавать <code>null</code> конструктору элемента, но оказывается, это очень полезно для трансформаций XML.
Любой прочий тип	Вызывается метод <code>ToString</code> , и результирующее значение трактуется как содержимое <code>string</code> .

Ранее я упоминал, что функциональное конструирование очень полезно для запросов LINQ, производящих XML. В качестве примера я создам стандартное дерево XML `BookParticipants`, которое уже использовал, но вместо жесткого кодирования значений элементов в виде строковых литералов, я извлеку данные из источника, опрашиваемого LINQ. В данном случае таким источником будет массив.

Во-первых, мне нужен класс, в котором будут храниться данные. Кроме того, поскольку у меня есть типы `BookParticipants`, я создам `enum` для разных типов, как показано ниже:

enum и класс для следующего примера

```
enum ParticipantTypes
{
    Author = 0,
    Editor
}
class BookParticipant
{
    public string FirstName;
    public string LastName;
    public ParticipantTypes ParticipantType;
}
```

Теперь я построю массив типа BookParticipant и сгенерирую дерево XML, используя запрос LINQ для извлечения данных из массива, как показано в листинге 7.16.

Листинг 7.16. Генерация дерева XML запросом LINQ

```
BookParticipant[] bookParticipants = new[] {
    new BookParticipant {FirstName = "Joe", LastName = "Rattz",
        ParticipantType = ParticipantTypes.Author},
    new BookParticipant {FirstName = "Ewan", LastName = "Buckingham",
        ParticipantType = ParticipantTypes.Editor}
};

 XElement xBookParticipants =
    new XElement("BookParticipants",
        bookParticipants.Select(p =>
            new XElement("BookParticipant",
                new XAttribute("type", p.ParticipantType),
                new XElement("FirstName", p.FirstName),
                new XElement("LastName", p.LastName))));

Console.WriteLine(xBookParticipants);
```

В приведенном коде я создаю массив объектов BookParticipant по имени bookParticipants. Затем код запрашивает значения из массива bookParticipants, используя операцию Select, и генерирует для каждого элемента BookParticipant, используя члены элемента массива. Так выглядит дерево XML, сгенерированное предыдущим кодом:

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

Представьте, как это сделать в W3C XML DOM API. Вообще-то представлять даже и не надо; вы можете просто взглянуть на листинг 6.1, потому что код создает точно такое же дерево XML.

Создание атрибутов с помощью XAttribute

В отличие от W3C XML DOM API, атрибуты не наследуются от узлов. Атрибут, реализованный в LINQ to XML с классом XAttribute, является парой “имя-значение”, хранящейся в коллекции объектов XAttribute, относящихся к объекту XElement.

Я могу создать атрибут и “на лету” добавить его к элементу, используя функциональное конструирование, как показано в листинге 7.17.

Листинг 7.17. Создание атрибута с функциональным конструированием

```
XElement xBookParticipant = new XElement("BookParticipant",
    new XAttribute("type", "Author"));

Console.WriteLine(xBookParticipant);
```

Запуск этого кода выдаст следующий результат:

```
'BookParticipant type="Author" />
```

Иногда, однако, вы не можете создать атрибут одновременно с конструированием его элемента. В таком случае вы должны создать его экземпляр и затем добавить к элементу, как показано в листинге 7.18.

Листинг 7.18. Создание атрибута и добавление его к элементу

```
XElement xBookParticipant = new XElement("BookParticipant");
XAttribute xAttribute = new XAttribute("type", "Author");
xBookParticipant.Add(xAttribute);
Console.WriteLine(xBookParticipant);
```

Результат идентичен:

```
<BookParticipant type="Author" />
```

Опять-таки, обратите внимание на гибкость метода `XElement.Add`. Он принимает любой объект, применяя те же правила для содержимого элемента, что и при создании экземпляра `XElement`. Замечательно!

Создание комментариев с помощью `XComment`

Создание комментариев с LINQ to XML тривиально. Комментарии XML реализуются в LINQ to XML классом `XComment`. Вы можете создать комментарий и добавить его к элементу “на лету”, используя функциональное конструирование, как показано в листинге 7.19.

Листинг 7.19. Создание комментария функциональным конструированием

```
XElement xBookParticipant = new XElement("BookParticipant",
    new XComment("This person is retired."));
Console.WriteLine(xBookParticipant);
```

Запуск этого кода выдаст следующий результат:

```
<BookParticipant>
  <!--This person is retired.-->
</BookParticipant>
```

Иногда, однако, вы не сможете создать комментарий одновременно с конструированием элемента. В таком случае вы должны сначала создать его экземпляр, а затем добавить его к элементу, как показано в листинге 7.20.

Листинг 7.20. Создание комментария и добавление его к элементу

```
XElement xBookParticipant = new XElement("BookParticipant");
XComment xComment = new XComment("This person is retired.");
xBookParticipant.Add(xComment);
Console.WriteLine(xBookParticipant);
```

Результат идентичен предыдущему:

```
<BookParticipant>
  <!--This person is retired.-->
</BookParticipant>
```

Создание контейнеров с помощью `XContainer`

Поскольку `XContainer` — абстрактный класс, вы не можете создавать его экземпляры. Вместо этого вы должны создавать экземпляр одного из его подклассов — `XDocument` или `XElement`. Концептуально `XContainer` — это класс, унаследованный от `XNode`, который может содержать другие классы-наследники `XNode`.

Создание объявлений с помощью XDeclaration

В программном интерфейсе LINQ to XML API создание объявлений также просто. Объявления XML реализованы здесь классом XDeclaration.

В отличие от большинства других классов LINQ to XML API, объявления должны добавляться к документу XML, а не к элементу. Помните, насколько гибкий конструктор у класса XElement? Любой класс, не спроектированный специально для этого, должен иметь собственный метод ToString, и выведенный им текст должен быть добавлен к элементу в виде текстового содержимого. Поэтому вы можете по невнимательности добавить к элементу объявление, используя класс XDeclaration. Однако это не даст вам ожидаемого результата.

Внимание! В то время как объявления XML применяются к документу XML в целом и должны быть добавлены к нему, объект XElement также благополучно принимает добавляемый к нему объект XDeclaration. Однако это не даст результата, которого вы ожидаете.

Можно создать объявление и добавить ее к документу XML “на лету”, используя функциональное конструирование, как в листинге 7.21.

Листинг 7.21. Создание объявления функциональным конструированием

```
XDocument xDocument = new XDocument(new XDeclaration("1.0", "UTF-8", "yes"),
                                     new XElement("BookParticipant"));
Console.WriteLine(xDocument);
```

Этот код производит следующий результат:

```
<BookParticipant />
```

Вы заметили, что в выводе недостает объявлению? Все правильно: метод ToString пропускает ее. Однако если вы отлаживаете код и заглянете в документ в окне слежения, то обнаружите, что объявление на месте.

Тем не менее, иногда вы не можете создать объявление одновременно с конструируемым документом. Тогда вы должны создать его экземпляр, а затем установить его в свойство Declaration документа, как показано в листинге 7.22.

Листинг 7.22. Создание объявления и установка в свойство Declaration документа

```
XDocument xDocument = new XDocument(new XElement("BookParticipant"));
XDeclaration xDeclaration = new XDeclaration("1.0", "UTF-8", "yes");
xDocument.Declaration = xDeclaration;
Console.WriteLine(xDocument);
```

Этот код производит следующий результат:

```
<BookParticipant />
```

Опять-таки, обратите внимание, что объявление не попадает в вывод при вызове метода ToString документа. Но, как и с предыдущим примером, если вы станете отлаживать код и просмотрите документ, то обнаружите там объявление.

Создание типов документов с помощью XDocumentType

Программный интерфейс LINQ to XML API делает операцию создания типов документов совершенно безболезненной операцией. Типы XML-документов реализованы LINQ to XML API классом XDocumentType.

В отличие от большинства других классов в LINQ to XML API, типы документов предназначены для добавления к документам XML, а не к элементам. Помните, насколько гибок конструктор класса XElement? Любой класс, который не был специально спроектирован для обработки, должен иметь метод `ToString`, и полученный текст будет добавлен к элементу в качестве текстового содержимого. Поэтому вы можете нечаянно добавить тип документа к элементу, используя класс XDocumentType, но это не даст вам ожидаемого результата.

Внимание! Хотя типы документов XML применимы к документу XML в целом, и должны добавляться к документу XML, объект XElement благополучно принимает добавляемый к нему объект XDocumentType. Однако это не дает результата, которого можно было ожидать.

Можно создавать тип документа и добавлять его к документу XML "на лету", используя функциональное конструирование, как показано в листинге 7.23.

Листинг 7.23. Создание типа документа функциональным конструированием

```
XDocument xDocument = new XDocument(new XDocumentType("BookParticipants",
    null,
    "BookParticipants.dtd",
    null),
    new XElement("BookParticipant"));
Console.WriteLine(xDocument);
```

Этот код выдаст следующий результат:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipant />
```

Иногда, однако, вы не можете создать тип документа одновременно с конструированием этого документа. Тогда вы должны сначала создать его экземпляр, а затем добавить к документу, как в листинге 7.24.

Листинг 7.24. Создание типа документа и добавление его к документу

```
XDocument xDocument = new XDocument();
XDocumentType documentType =
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null);
xDocument.Add(documentType, new XElement("BookParticipants"));
Console.WriteLine(xDocument);
```

Вот результат работы этого кода:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipants />
```

Обратите внимание, что в предыдущем коде я не добавил ни одного элемента перед добавлением типа документа. Если вы добавите тип документа после добавления любого элемента, то получите следующее исключение:

Unhandled Exception: System.InvalidOperationException: This operation would create an incorrectly structured document.

Необработанное исключение: System.InvalidOperationException: Эта операция создаст некорректно структурированный документ.

...

Поэтому, если вы собираетесь специфицировать тип документа после создания его экземпляра, убедитесь, что не указали ни одного элемента при создании экземпляра документа, используя функциональное конструирование, или не добавили ни одного элемента перед добавлением типа документа.

Создание документов с помощью XDocument

Я повторял это уже так много раз, что вам, должно быть, надоело слушать, но скажу вновь: с LINQ to XML не обязательно создавать XML-документ только для того, чтобы создать дерево XML или его фрагмент. Однако при необходимости создание XML-документа с LINQ to XML столь же тривиально. XML-документы реализованы в LINQ to XML классом XDocument. Пример приведен в листинге 7.25.

Листинг 7.25. Простой пример создания XML-документа с помощью XDocument

```
XDocument xDocument = new XDocument();
Console.WriteLine(xDocument);
```

Этот код не производит никакого вывода, потому что созданный документ XML пуст. Приведенный пример может быть, чересчур тривиален, поэтому я создам документ с классами LINQ to XML, которые специально предназначены для добавления к объекту XDocument, как показано в листинге 7.26.

Листинг 7.26. Несколько более сложный пример создания XML-документа с помощью XDocument

```
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    new XElement("BookParticipants"));
Console.WriteLine(xDocument);
```

И инструкция обработки (processing instruction), и элемент также могут быть добавлены к элементам, но я хотел создать XML-документ с некоторым "мясом". Также я добавил инструкцию обработки, так что вы можете увидеть ее в действии.

Результат этого кода выглядит так:

```
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants />
```

Возможно, вы заметили отсутствие объявления. Как это упоминалось в связи с примером создания объявлений, метод ToString документа пропускает объявление при выводе. Однако если вы попробуете отладить код и заглянете внутрь документа, то увидите, что объявление на месте.

Создание имен с помощью XName

Как я упоминал ранее в этой главе, с LINQ to XML у вас нет необходимости непосредственно создавать имена с объектами XName. Фактически класс XName не имеет общедоступных конструкторов, так что нет способа создавать его экземпляры. Объект XName может быть создан для вас из строки с необязательным пространством имен автоматически при создании объекта XName.

Объект XName состоит из LocalName — строки — и пространства имен, которое представлено в XNamespace.

Листинг 7.27 содержит код примера вызова конструктора XElement, принимающего XName в качестве аргумента.

Листинг 7.27. Пример кода, где для вас создается объект XName

```
XElement xBookParticipant = new XElement("BookParticipant");
Console.WriteLine(xBookParticipant);
```

В приведенном примере я создаю экземпляр объекта XElement, передав ему имя элемента в виде строки, так что объект XName создается с LocalName, равным BookParticipant, и присваивается свойству Name объекта XElement. В данном случае никакого пространства имен не указывается, так что объект XName не имеет пространства имен.

Нажав <Ctrl+F5>, получим следующий результат:

```
<BookParticipant />
```

Я могу специфицировать пространство имен в коде, показанном в листинге 7.28.

Листинг 7.28. Пример кода, где для вас создается объект XName и специфицировано пространство имен

```
XNamespace ns = "http://www.linqdev.com/Books";
 XElement xBookParticipant = new XElement(ns + "BookParticipant");
 Console.WriteLine(xBookParticipant);
```

Этот код выдаст следующий XML:

```
<BookParticipant xmlns="http://www.linqdev.com/Books" />
```

За дополнительной информацией о создании имен с помощью LINQ to XML API обращайтесь в раздел “Имена, пространства имен и префиксы”, приведенный ранее в этой главе.

Создание пространств имен с помощью XNamespace

В LINQ to XML API пространства имен реализованы классом XNamespace. Пример создания и использования пространства имен приведен выше в листинге 7.28. Он демонстрирует создание пространства имен с помощью класса XNamespace.

За дополнительной информацией о создании пространств имен с помощью LINQ to XML API обращайтесь в раздел “Имена, пространства имен и префиксы”, приведенный ранее в этой главе.

Создание узлов с помощью XNode

Поскольку XNode — абстрактный класс, вы не можете создавать его экземпляры. Вместо этого вы должны создавать экземпляры одного из его подклассов: XComment, XContainer, XDocumentType, XProcessingInstruction или XText. Концептуально XNode — это любой класс, который функционирует как узел в дереве XML.

Создание инструкций обработки с помощью XProcessingInstruction

Инструкции обработки никогда ранее не было так легко создавать, как в API-интерфейсе LINQ to XML. Здесь они реализуются классом XProcessingInstruction.

Вы можете создать инструкции обработки на уровне документа или элемента. Листинг 7.29 демонстрирует пример их создания “на лету” в обоих случаях с помощью функционального конструирования.

Листинг 7.29. Создание инструкции обработки на уровне документа и элемента

```
XDocument xDocument = new XDocument(
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    new XElement("BookParticipants",
        new XElement("BookParticipant",
```

```

new XProcessingInstruction("ParticipantDeleter", "delete"),
new XElement("FirstName", "Joe"),
new XElement("LastName", "Rattz"))));
Console.WriteLine(xDocument);

```

В приведенном коде я добавил инструкции обработки как к документу, так и к элементу BookParticipant. Прежде чем отобразить результат, я хочу обратить ваше внимание на то, как хорошо здесь проходит функциональное конструирование. Очень легко создать это дерево XML с двумя инструкциями обработки. По сравнению с этим самый первый пример из предыдущей главы, приведенный в листинге 6.1, опять доказывает, насколько LINQ to XML API упрощает ваш код. И, наконец, вот результат:

```

<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant>
    <?ParticipantDeleter delete?>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

```

Теперь, я полагаю, вы уже можете представить себе код добавления инструкции обработки после конструирования, поскольку это очень похоже на добавление любого другого узла, описанного ранее. Поэтому, чтобы не утомлять вас рутиной, в листинге 7.30 представлен существенно более сложный пример создания и добавления инструкции обработки постфактум.

Листинг 7.30. Более сложный пример добавления инструкций обработки после конструирования документа и элемента

```

XDocument xDocument =
  new XDocument(new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
XProcessingInstruction xPI1 = new XProcessingInstruction("BookCataloger",
  "out-of-print");
xDocument.AddFirst(xPI1);
XProcessingInstruction xPI2 = new XProcessingInstruction("ParticipantDeleter",
  "delete");
 XElement outOfPrintParticipant = xDocument
  .Element("BookParticipants")
  .Elements("BookParticipant")
  .Where(e => ((string)(( XElement)e).Element("FirstName")) == "Joe"
    && ((string)(( XElement)e).Element("LastName")) == "Rattz")
  .Single< XElement>();
outOfPrintParticipant.AddFirst(xPI2);
Console.WriteLine(xDocument);

```

Стоит сделать несколько замечаний относительно кода этого примера. Во-первых, я создал документ и его дерево XML, используя функциональное конструирование. После конструирования документа и дерева я добавил к документу инструкцию обработки. Однако здесь я использовал метод XElement.AddFirst для создания первого дочернего узла документа, в противоположность методу XElement.Add, который просто добавил бы его в конец списка дочерних узлов документа, что для любой инструкции обработки было бы слишком поздно.

Вдобавок, чтобы добавить инструкцию обработки к одному из элементов, мне нужна ссылка на него. Я должен был сконструировать объект XElement и сохранить ссылку на него, но тут я подумал, что стоит начать давать подсказки относительно будущих возможностей запросов. Вы можете видеть, что я выполняю достаточно сложный запрос, где получаю из документа элемент BookParticipants, используя метод Element, который опишу позднее в разделе "Проход по XML", а затем получаю последовательность объектов XElement по имени BookParticipant, где элемент FirstName элемента BookParticipant равен "Joe", а LastName равен "Rattz". Обратите внимание, что я использую новые средства извлечения узлов из LINQ to XML API, о которых говорил ранее, чтобы получить значения узла FirstName и LastName, приводя их к типу string.

И, наконец, операция Where возвращает IEnumerable<T>, а мне нужен непосредственно объект XElement. Поэтому я вспомнил, что в моем описании отложенных стандартных операций запросов LINQ to Object в главе 5 присутствует одна операция, которая возвращает элемент из последовательности, в предположении, что он в ней единственный, и эта операция называется Single. Получив от этого запроса ссылку на правильный объект XElement, совсем легко добавить к нему инструкцию обработки и отобразить результаты. И если говорить о результатах, вот они:

```
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant>
    <?ParticipantDeleter delete?>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Создание потоковых элементов с помощью XStreamingElement

Помните ли вы из части 2 нашей книги, которая называлась "LINQ to Objects", что многие из стандартных операций запросов в действительности откладывают свою работу до того момента, когда не начнется перечисление возвращаемых ими данных? Если я вызываю некоторую операцию, которая фактически откладывает свое выполнение, и хочу спроектировать вывод моего запроса в виде XML, у меня возникает дилемма. С одной стороны, я хочу воспользоваться преимуществом отложенной природы операции, поскольку нет необходимости выполнять лишнюю работу до тех пор, пока в ней не возникнет необходимости. Но с другой стороны, мой вызов LINQ to XML API заставит запрос выполнятся немедленно.

В листинге 7.31 обратите внимание, что даже несмотря на то, что я изменил четвертый элемент массива names при выводе значения моего объекта XElement, дерево XML содержит его оригинальное значение. Это связано с тем, что элемент xNames был полностью создан перед изменением элемента массива names.

Листинг 7.31. Немедленное выполнение конструирования дерева XML

```
string[] names = { "John", "Paul", "George", "Pete" };
XElement xNames = new XElement("Beatles",
    from n in names
    select new XElement("Name", n));
names[3] = "Ringo";
Console.WriteLine(xNames);
```

Перед обсуждением результатов этого кода я хочу обратить ваше внимание, насколько он замечателен. Заметьте, что я создаю элемент, чье имя — Beatles, и чье содержимое составляет последовательность объектов XElement, элемент которых назван Name. Этот код производит следующее дерево XML:

```
<Beatles>
  <Name>John</Name>
  <Name>Paul</Name>
  <Name>George</Name>
  <Name>Pete</Name>
</Beatles>
```

Впечатляюще. Каждый объект XElement из последовательности становится дочерним элементом. Чем это хорошо? Как я упоминал, несмотря на то, что я изменил names[3] на "Ringo" перед выводом XML, последний элемент по-прежнему содержит имя Pete — исходное значение. Это потому, что последовательность names должна быть полностью перечислена, чтобы сконструировать объект XElement, что требует немедленного выполнения запроса.

Если я действительно хочу, чтобы конструирование дерева XML было отложено, мне нужен какой-то другой способ сделать это, и именно для этого предназначены потоковые (streaming) элементы. В LINQ to XML потоковые элементы реализованы классом XStreamingElement.

Листинг 7.32 демонстрирует тот же пример, но на этот раз я использую объекты XStreamingElement вместо XElement.

Листинг 7.32. Демонстрация отложенного выполнения конструирования дерева XML посредством использования класса XStreamingElement

```
string[] names = { "John", "Paul", "George", "Pete" };
XStreamingElement xNames =
  new XStreamingElement("Beatles",
    from n in names
    select new XStreamingElement("Name", n));
names[3] = "Ringo";
Console.WriteLine(xNames);
```

Если это работает, как я описал, то теперь значение Name последнего узла должно быть Ringo, а не Pete. И вот доказательство:

```
<Beatles>
  <Name>John</Name>
  <Name>Paul</Name>
  <Name>George</Name>
  <Name>Ringo</Name>
</Beatles>
```

Прошу прощения, Pete, кажется, тебя вновь заменили.

Создание текста с помощью XText

Создание элемента с текстовым значением — чрезвычайно простая задача. В листинге 7.33 приведен код, делающий это.

Листинг 7.33. Создание элемента и присвоения строки в качестве его значения

```
XElement xFirstName = new XElement("FirstName", "Joe");
Console.WriteLine(xFirstName);
```

Все совершенно очевидно, и нет никаких сюрпризов. Запустив этот код нажатием <Ctrl+F5>, получим следующий результат:

```
<FirstName>Joe</FirstName>
```

Однако здесь скрыт тот факт, что строка "Joe" преобразуется в объект XText, и именно этот объект добавляется к объекту XElement. Фактически просмотр объекта xFirstName в отладчике покажет, что он содержит единственный узел — объект XText, чье значение — "Joe". Поскольку все это делается для вас автоматически, в большинстве случаев вам не понадобится непосредственно конструировать текстовый объект.

Тем не менее, если такая необходимость возникнет, вы можете создать текстовый объект, создав экземпляр объекта XText, как показано в листинге 7.34.

Листинг 7.34. Создание текстового узла и передача его в качестве значения созданному элементу

```
XText xName = new XText("Joe");
 XElement xFirstName = new XElement("FirstName", xName);
 Console.WriteLine(xFirstName);
```

Этот код производит в точности такой же результат, что и предыдущий пример, и если мы просмотрим внутреннее состояние объекта xFirstName, оно также будет идентично тому, что и в предыдущем примере:

```
<FirstName>Joe</FirstName>
```

Создание CData с помощью XCData

Создать элемента со значением CData также очень просто. Пример приведен в листинге 7.35.

Листинг 7.35. Создание узла XCData и передача его в качестве значения созданного элемента

```
XElement xErrorMessage = new XElement("HTMLMessage",
    new XCData("<H1>Invalid user id or password.</H1>"));
Console.WriteLine(xErrorMessage);
```

Этот код дает следующий вывод:

```
<HTMLMessage><! [CDATA[<H1>Invalid user id or password.</H1> ]]></HTMLMessage>
```

Как видите, LINQ to XML API облегчает работу с CData.

ВЫВОД XML

Конечно, создание, модификация и удаление данных XML ничего не стоит, если вы не можете сохранить изменения. В этом разделе мы опишем несколько способов вывода вашего XML.

Сохранение с помощью XDocument.Save()

Вы можете сохранить документ XML, используя любой из нескольких методов XDocument.Save. Вот список его прототипов:

```
void XDocument.Save(string filename);
void XDocument.Save(TextWriter textWriter);
void XDocument.Save(XmlWriter writer);
void XDocument.Save(string filename, SaveOptions options);
void XDocument.Save(TextWriter textWriter, SaveOptions options);
```

Листинг 7.36 содержит пример, где я сохраняю документ XML в файле, находящемся в папке моего проекта.

Листинг 7.36. Сохранение документа методом XDocument.Save

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
xDocument.Save("bookparticipants.xml");
```

Обратите внимание, что язываю метод Save на *объекте* типа XDocument. Это потому, что методы Save являются методами экземпляра. Методы Load, о которых вы прочтете ниже в разделе “Ввод XML”, являются статическими методами, и должны вызываться на классе XDocument или XElement.

Ниже приведено содержимое сгенерированного файла bookparticipants.xml при просмотре его в текстовом редакторе, подобном Блокноту (Notepad):

```
<?xml version="1.0" encoding="utf-8"?>
<BookParticipants>
    <BookParticipant type="Author" experience="first-time" language="English">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
</BookParticipants>
```

Этот вывод документа XML легко читать, поскольку вызванная версия метода Save форматирует вывод. То есть, если я вызову версию метода Save, принимающую строку имени файла и аргумент SaveOptions, передав в нем значение SaveOptions.None, то получу тот же результат, что и раньше. Но если я вызову метод Save следующим образом:

```
xDocument.Save("bookparticipants.xml", SaveOptions.DisableFormatting);
```

то результат в файле будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?><BookParticipants><BookParticipant type="Author" experience="first-time" language="English"><FirstName>Joe</FirstName><LastName>Rattz</LastName></BookParticipant></BookParticipants>
```

Это одна непрерывная строка текста. Однако вы должны просматривать результат именно в текстовом редакторе, потому что браузер все красиво сформатирует для вас.

Конечно, вы можете использовать любой другой доступный метод для вывода вашего документа — выбор за вами.

Сохранение с помощью XElement.Save()

Я уже много раз говорил, что в LINQ to XML API создавать документ XML не обязательно. Также не обязательно это делать для сохранения вашего XML-файла. Класс XElement также имеет несколько методов Save, предназначенных для этой цели:

```
void XElement.Save(string filename);
void XElement.Save(TextWriter textWriter);
void XElement.Save(XmlWriter writer);
void XElement.Save(string filename, SaveOptions options);
void XElement.Save(TextWriter textWriter, SaveOptions options);
```

Листинг 7.37 содержит пример, очень похожий на предыдущий, за исключением того, что в нем не создается документ XML.

Листинг 7.37. Сохранение элемента методом XElement.Save

```
XElement bookParticipants =
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")));
bookParticipants.Save("bookparticipants.xml");
```

Сохраненный XML выглядит идентично предыдущему примеру, где я в действительности имел XML-документ:

```
<?xml version="1.0" encoding="utf-8"?>
<BookParticipants>
    <BookParticipant type="Author" experience="first-time" language="English">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
</BookParticipants>
```

Ввод XML

Создание и сохранение XML в файле не имело бы смысла, если бы не было способа загрузить его обратно в дерево XML. Опишем некоторые примеры чтения XML.

Загрузка с помощью XDocument.Load()

Теперь, когда вы знаете, как сохранять документы и фрагменты XML, вы, наверно, захотите посмотреть, как их загружать обратно. Это можно сделать, используя любой из нескольких доступных методов. Вот их список:

```
static XDocument XDocument.Load(string uri);
static XDocument XDocument.Load(TextReader textReader);
static XDocument XDocument.Load(XmlReader reader);
static XDocument XDocument.Load(string uri, LoadOptions options);
static XDocument XDocument.Load(TextReader textReader, LoadOptions options);
static XDocument XDocument.Load(XmlReader reader, LoadOptions options);
```

Вы можете заметить, насколько эти методы симметричны методам XDocument.Save. Однако есть пара отличий, на которые следует обратить внимание. Во-первых, метод Save должен вызываться на объектах типа XDocument или XElement, потому что Save — метод экземпляра. Однако метод Load — статический, поэтому вы должны вызывать его на самом классе XDocument. Во-вторых, методы Save принимают параметр типа string, в котором должно быть передано имя файла, в то время как методы Load могут принимать в параметре string строку URI.

Вдобавок метод Load допускает при загрузке документа XML специфицировать параметр типа LoadOptions. Значения enum LoadOptions перечислены в табл. 7.2.

Таблица 7.2. Перечисление LoadOptions

Опция	Описание
LoadOptions.None	Применяйте эту опцию, чтобы указать, что никакие опции загрузки не применяются.
LoadOptions.PreserveWhitespace	Используйте эту опцию, чтобы предохранить пробелы и пустые строки в исходном XML.
LoadOptions.SetLineInfo	Используйте эту опцию, чтобы иметь возможность получать строку и позицию любого объекта, унаследованного от XObject, посредством интерфейса IXmlLineInfo.
LoadOptions.SetBaseUri	Используйте эту опцию, чтобы получать базовый URI любого объекта-наследника XObject.

Эти опции можно комбинировать с помощью битовой операции “ИЛИ” (`|`). Однако некоторые опции не работают в ряде контекстов. Например, при создании элемента или документа передачей строки никакой информации о позициях и номерах строк недоступно, как и недоступен базовый URI. Также нет базового URI при создании документа с помощью XmlReader.

В листинге 7.38 показан пример, где я загружаю документ XML, созданный в предыдущем примере — листинге 7.37.

Листинг 7.38. Загрузка документа методом XDocument.Load

```
XDocument xDocument = XDocument.Load("bookparticipants.xml",
    LoadOptions.SetBaseUri | LoadOptions.SetLineInfo);
Console.WriteLine(xDocument);
 XElement firstName = xDocument.Descendants("FirstName").First();
Console.WriteLine("FirstName Line:{0} - Position:{1}",
    ((IXmlLineInfo)firstName).LineNumber,
    ((IXmlLineInfo)firstName).LinePosition);
Console.WriteLine("FirstName Base URI:{0}", firstName.BaseUri);
```

На заметку! Вы должны либо добавить директиву `using System.Xml;`, если ее еще нет, либо специфицировать пространство имен, ссылаясь на интерфейс `IXmlLineInfo` в вашем коде. В противном случае тип `IXmlLineInfo` не будет найден компилятором.

Этот код загружает тот же файл XML, что я создал в предыдущем примере. После того, как я загружаю и отображаю документ, я получаю ссылку на элемент `FirstName` и отображаю строку и позицию этого элемента в исходном XML-документе. Затем отображаю базовый URI элемента.

Вот результат:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
FirstName Line:4 - Position:6
FirstName Base URI:file:///C:/Documents and Settings/.../Projects/LINQChapter7/
LINQChapter7/bin/Debug/bookparticipants.xml
```

Этот вывод выглядит так, как и следовало ожидать, но с одним возможным исключением. Во-первых, действительно, XML-документ выглядит хорошо. Я вижу номер строки и позиции элемента FirstName, но номер строки вызывает недоумение. Показано, что элемент находится в четвертой строке, но в отображенном документе на самом деле элемент FirstName расположен в третьей строке. В чем же дело? Если вы рассмотрите загруженный XML-документ, то увидите, что он начинается с объявления документа, которая пропущена в выводе:

```
<?xml version="1.0" encoding="utf-8"?>
```

Именно поэтому номер строки, в которой расположен элемент FirstName, отображается как четвертый.

Загрузка с помощью XElement.Load()

Точно так же, как можно сохранять и XDocument, и XElement, можно загружать тот или другой. Загрузка содержимого элемента почти идентична загрузке документа. Вот доступные для этого методы:

```
static XElement XElement.Load(string uri);
static XElement XElement.LoadTextReader(TextReader textReader);
static XElement XElement.Load(XmlReader reader);
static XElement XElement.Load(string uri, LoadOptions options);
static XElement XElement.Load(TextReader textReader, LoadOptions options);
static XElement XElement.Load(XmlReader reader, LoadOptions options);
```

Эти методы являются статическими, как и методы XDocument.Save, поэтому должны вызываться из класса XElement напрямую. Листинг 7.39 содержит пример загрузки того же файла XML, который я сохранил методом XElement.Save в листинге 7.37.

Листинг 7.39. Загрузка элемента методом XElement.Load

```
XElement xElement = XElement.Load("bookparticipants.xml");
Console.WriteLine(xElement);
```

Как следовало ожидать, вывод выглядит так:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Как и XDocument.Load, метод XElement.Load также имеет перегрузки, принимающие параметр LoadOptions. Его описание читайте выше, в разделе “Загрузка с помощью XDocument.Load()” настоящей главы.

Разбор содержимого методами XDocument.Parse() или XElement.Parse()

Сколько раз вам приходилось передавать в своих программах содержимое XML в виде строки, лишь иногда нуждаясь в выполнении некоторой серьезной работы XML? Получение данных из переменной типа string в переменную типа документа XML всегда представляет определенную трудность. Отныне можете об этом не беспокоиться. Одно из моих любимых средств LINQ to XML API — метод Parse.

Оба класса — и XDocument, и XElement — имеют статический метод Parse для разбора строк XML. Полагаю, вы уже достаточно подготовлены к тому, что если вы можете

выполнять разбор строки с классом `XDocument`, то также можете делать это с классом `XElement` и наоборот. И поскольку весь программный интерфейс LINQ to XML построен вокруг элементов, на этот раз я представлю только пример, ориентированный на элементы:

В разделе “Сохранение с помощью `XDocument.Save()`” ранее в этой главе я показал вывод метода `Save`, если специфицирован параметр `LoadOptions` со значением `DisableFormatting`. Результат — единая строка XML. Пример из листинга 7.40 я начну со строки XML (после защиты внутренних кавычек), разберу ее в элемент и выведу этот XML-элемент на экран.

Листинг 7.40. Разбор строки XML в элемент

```
string xml = "<?xml version=\"1.0\" encoding=\"utf-8\"?><BookParticipants>" +
    "<BookParticipant type=\"Author\" experience=\"first-time\" language=" +
    "\"English\"><FirstName>Joe</FirstName><LastName>Rattz</LastName>" +
    "</BookParticipant></BookParticipants>";
XElement xElement = XElement.Parse(xml);
Console.WriteLine(xElement);
```

Результат выглядит следующим образом:

```
<BookParticipants>
  <BookParticipant type="Author" experience="first-time" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Здорово, не правда ли? Помните старый способ, когда нужно было создавать документ с использованием класса `XmlDocument` из W3C XML DOM? Благодаря исключению центральной роли документа, вы можете превратить строку XML в реальное дерево XML в одно мгновение — простым вызовом метода.

Проход по XML

Проход по XML выполняется с помощью 4 свойств и 11 методов. В данном разделе я попытаюсь в основном использовать один и тот же пример кода для каждого свойства и метода, за исключением изменения единственного аргумента в единственной строке, когда это возможно. Пример в листинге 7.41 строит полный XML-документ.

Листинг 7.41. Базовый пример для всех последующих примеров

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
```

```

    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
Console.WriteLine(xDocument);

```

Для начала отметьте, что я сохраняю ссылку на первый сконструированный элемент BookParticipant. Я делаю это, чтобы иметь базовый элемент, с которого можно было бы продолжать проход. Хотя в данном примере я не использую переменную firstParticipant, она понадобится в последующих примерах. Следующее, что нужно отметить — это аргумент метода Console.WriteLine. В данном случае я вывожу сам документ. В последующих примерах я буду изменять этот аргумент, чтобы продемонстрировать, как проходить по дереву XML. Ниже представлен вывод предыдущего примера:

```

<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

```

Свойства прохода

Начнем дискуссию с первичных свойств прохода. Когда специфицированы направления (вверх, вниз и т.д.), они указываются относительно элемента, на котором вызван метод. В последующих примерах я сохраняю ссылку на первый элемент BookParticipant, который и служит базовым элементом для прохода.

Вперед с помощью XNode.NextNode

Проход вперед по дереву XML выполняется со свойством NextNode. Пример приведен в листинге 7.42.

Листинг 7.42. Проход вперед от объекта XElement через свойство NextNode

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке я сохраняю
  // ссылку на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
  Console.WriteLine(firstParticipant.NextNode);

```

Поскольку базовым служит первый элемент BookParticipant — firstParticipant, проход вперед должен привести меня ко второму элементу BookParticipant. И вот результат:

```
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
```

Результат доказывает, что я заслужил вознаграждение. Поверите ли вы мне, если я скажу, что если бы я обратился к свойству PreviousNode элемента, то оно бы вернуло мне null, поскольку этот узел — первый в списке родительского узла? Если вас это не впечатлило, я могу опять вернуться к первому узлу. То есть я сделаю шаг вперед по свойству NextNode и затем — обратно с PreviousNode, что приведет меня туда, откуда я начал. Если вы когда-либо слышали выражение “шаг вперед и два назад”, то всего одно дополнительное обращение к свойству PreviousNode позволит вам сделать это. LINQ допускает это. Пример представлен в листинге 7.43.

Листинг 7.43. Проход назад от объекта XElement через свойство PreviousNode

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.NextNode.PreviousNode);
```

Если это работает, как следовало ожидать, то я должен получить XML первого элемента BookParticipant:

```
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

LINQ to XML действительно превращает проход по дереву XML в пару пустяков.

Вверх к документу с помощью XObject.Document

Получить XML-документ из объекта XElement можно, просто обратившись к свойству Document элемента. Обратите внимание на изменение в вызове метода Console.WriteLine, как показано в листинге 7.44.

Листинг 7.44. Обращение к документу XML от объекта XElement через свойство Document

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
```

```

new XDeclaration("1.0", "UTF-8", "yes"),
new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
new XProcessingInstruction("BookCataloger", "out-of-print"),
// Обратите внимание, что в следующей строке я сохраняю ссылку на первый элемент BookParticipant.
new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
        new XAttribute("type", "Author"),
        new XElement("FirstName", "Joe"),
        new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.Document);

```

Это даст нам тот же документ, что и в выводе примера из листинга 7.41, что и подтверждает вывод:

```

<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<?BookCataloger out-of-print?>
<BookParticipants>
    <BookParticipant type="Author">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>

```

Вверх с помощью `XObject.Parent`

Когда нужно подняться по дереву на один уровень выше, не удивительно, что в этом вам поможет свойство `Parent`. Изменение узла, переданного методу `WriteLine`, как показано в листинге 7.45, изменит вывод (как вы увидите ниже).

Листинг 7.45. Проход вверх от объекта `XElement` через свойство `Parent`

```

 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.Parent);

```

Выход соответствующим образом изменится:

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

Однако не давайте ввести себя в заблуждение. Это не весь документ. Здесь отсутствует описание типа документа и инструкция обработки.

Методы прохода

Чтобы продемонстрировать методы прохода, поскольку они возвращают последовательности из множества узлов, мне придется заменить единственный вызов метода `Console.WriteLine` циклом `foreach`, чтобы вывести потенциально многочисленные узлы. Это приведет нас к прежнему вызову `Console.WriteLine`, который выглядит в основном так:

```
foreach(XNode node in firstParticipant.Nodes())
{
  Console.WriteLine(node);
}
```

Единственное, что будет изменяться от примера к примеру — вызываемый метод на узле `firstParticipant` в цикле `foreach`.

Вниз с помощью `XContainer.Nodes()`

Нет, я не выражаю своего презрения к узлам и не выражаю своего восхищения ими, как после восхождения на высокую гору, откуда открывается чудесный вид. Я просто описываю направление прохода по дереву.

Спуск по дереву XML выполняется простым вызовом метода `Nodes`, который возвращает последовательность объектов `XNode` данного объекта. Если вы что-то пропустили в предыдущих главах, напомню, что последовательность — это `IEnumerable<T>`, т.е. `IEnumerable` определенного типа. Пример приведен в листинге 7.46.

Листинг 7.46. Проход вниз от объекта `XElement` через свойство `Nodes`

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
  new XDeclaration("1.0", "UTF-8", "yes"),
  new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
  new XProcessingInstruction("BookCataloger", "out-of-print"),
  // Обратите внимание, что в следующей строке я сохраняю
  // ссылку на первый элемент BookParticipant.
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
```

```

foreach(XNode node in firstParticipant.Nodes())
{
    Console.WriteLine(node);
}

```

Вот вывод:

```

<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>

```

Не забудьте, что этот метод возвращает все дочерние узлы, а не только элементы. Поэтому любой узел из списка дочерних узлов первого участника будет включен. Сюда могут относиться комментарии (XComment), текст (XText), инструкции обработки (XProcessingInstruction), типы документа (XDocumentType) или элементы (XElement). Также обратите внимание, что сюда не входят атрибуты, потому что атрибут — не узел.

Чтобы представить лучший пример использования метода Nodes, давайте взглянем на код в листинге 7.47. Он подобен базовому примеру, но с некоторыми дополнительными узлами.

Листинг 7.47. Проход вниз от объекта XElement через свойство Nodes с дополнительными типами узлов

```

 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes())
{
    Console.WriteLine(node);
}

```

Этот пример отличается от предыдущего в том, что здесь к первому элементу BookParticipant добавлен комментарий и инструкция обработки. Нажатие <Ctrl+F5> отобразит следующее:

```

<!--This is a new author.-->
<?AuthorHandler new?>
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>

```

Теперь мы видим комментарий и инструкцию обработки. Но что, если вам нужны только узлы определенного типа, например, только элементы? Помните операцию OfType из главы 4? Я могу использовать эту операцию для возврата только тех узлов,

которые относятся к определенному типу, такому как XElement. Используя тот же базовый код, что и в листинге 7.47, для возврата только элементов, я просто изменю строку foreach, как показано в листинге 7.48.

Листинг 7.48. Использование операции OfType для возврата только элементов

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes().OfType< XElement>())
{
    Console.WriteLine(node);
}
```

Как видите, объекты XComment и XProcessingInstruction по-прежнему создаются. Но поскольку я вызвал операцию OfType, код выдает следующий результат:

```
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Видите, насколько умно работают новые средства языка C# с LINQ? Разве не здорово, что мы можем использовать эту стандартную операцию запроса для такого ограничения последовательности узлов XML? Таким образом, если вы захотите получить только комментарии из первого элемента BookParticipant, можно ли использовать операцию OfType для этого? Конечно можно! И код будет выглядеть подобно показанному в листинге 7.49.

Листинг 7.49. Использование операции OfType для возврата только комментариев

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
```

```

        new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Nodes().OfType<XComment>())
{
    Console.WriteLine(node);
}

```

Вот вывод:

```
<!--This is a new author.-->
```

Но сможете ли вы использовать операцию OfType для извлечения только атрибутов? Нет, не сможете. Это хитрый вопрос. Напомню, что в отличие от W3C XML DOM API, в LINQ to XML API атрибуты не являются узлами дерева XML. Они представляют собой последовательность пар "имя-значение", привязанных к элементу. Чтобы получить атрибуты узла BookParticipant, придется изменить код, как показано в листинге 7.50.

Листинг 7.50. Использование операции OfType для возврата только комментариев

```

 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach(XAttribute attr in firstParticipant.Attributes())
{
    Console.WriteLine(attr);
}

```

Обратите внимание, что мне пришлось изменить не только свойство или метод первого элемента BookParticipant, к которому я обращаюсь, но также изменить переменную перечисления, сделав ее типа XAttribute, потому что XAttribute не наследуется от XNode. И вот результат:

```
type="Author"
```

Вниз с помощью XContainer.Elements()

Поскольку LINQ to XML API настолько сосредоточен на элементах, и именно с ними мы работаем больше всего, в Microsoft предусмотрели быстрый способ получения только элементов из всех дочерних узлов посредством метода Elements. Это эквивалент вызова метода OfType< XElement > на последовательности, возвращенной методом Nodes.

Листинг 7.51 демонстрирует пример, логически эквивалентный примеру из листинга 7.48.

Листинг 7.51. Обращение к дочерним элементам с использованием метода Elements

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach(XNode node in firstParticipant.Elements())
{
    Console.WriteLine(node);
}

```

Этот код выдает в точности тот же результат, что и код из листинга 7.48:

```
<FirstName>Joe</FirstName>
<LastName>Rattz</LastName>
```

Метод Elements также имеет перегруженную версию, которая позволяет вам передавать имя искомого элемента, как показано в листинге 7.52.

Листинг 7.52. Обращение к именованным дочерним элементам с использованием метода Elements

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));

```

```

foreach(XNode node in firstParticipant.Elements("FirstName"))
{
    Console.WriteLine(node);
}

```

Этот код выдаст вот что:

```
<FirstName>Joe</FirstName>
```

Вниз с помощью XContainer.Element()

Вы можете получить дочерний элемент, соответствующий указанному имени, используя метод Element. Вместо возврата последовательности, требующей затем применения цикла foreach, возвращается единственный элемент, как показано в листинге 7.53.

Листинг 7.53. Обращение к первому дочернему элементу с указанным именем

```

 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine(firstParticipant.Element("FirstName"));

```

Этот код даст следующий вывод:

```
<FirstName>Joe</FirstName>
```

Рекурсивно вверх с помощью XNode.Ancestors()

В то время как вы можете получить единственный родительский элемент из свойства узла Parent, всю последовательность элементов-предков можно получить методом Ancestors. Его отличие состоит в том, что он рекурсивно проходит вверх по дереву XML вместо того, чтобы остановиться одним уровнем выше, и возвращает только элементы а не все узлы.

Чтобы сделать пример более наглядным, я добавлю некоторые дочерние узлы к элементу FirstName первого участника. Также вместо перечисления предков первого элемента BookParticipant я использую метод Element для спуска двумя уровнями ниже вновь добавленному элементу NickName. Это предоставит больше предков для большей наглядности. Необходимый код показан в листинге 7.54.

Листинг 7.54. Проход вверх от объекта XElement методом Ancestors

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XText("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName",
                new XText("Joe"),
                new XElement("NickName", "Joey")),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach (XElement element in firstParticipant.
    Element("FirstName").Element("NickName").Ancestors())
{
    Console.WriteLine(element.Name);
}

```

И вновь обратите внимание, что я добавил некоторые дочерние узлы к элементу FirstName первого участника. Это добавит элементу FirstName первого участника содержимое, включающее объект XText, эквивалентный строке "Joe", и дочерний элемент NickName. Я извлекаю элемент NickName элемента FirstName первого участника, чтобы получить всех предков. Вдобавок отметьте, что я использую переменную типа XElement вместо типа XNode для перечисления последовательности, возвращенной методом Ancestors. Таким образом, я могу добраться до свойства Name элемента. Вместо отображения XML элемента, как я делал это в предыдущих примерах, я лишь отображаю имя каждого элемента в последовательности предков. Я делаю так, чтобы избежать путаницы при отображении XML каждого предка, потому что каждый из них включал бы предыдущий, что затруднило бы восприятие результата. С учетом всего сказанного, вот что я имею:

```

 FirstName
 BookParticipant
 BookParticipants

```

Как и ожидалось, код рекурсивно поднимается по дереву XML.

Рекурсивно вверх с помощью XElement.AncestorsAndSelf()

Этот метод работает точно так же, как и Ancestors, но с тем отличием, что в возвращенную последовательность предков он включает сам элемент, на котором вызван. В листинге 7.55 представлен тот же пример, что и предыдущий, но с вызовом метода AncestorsAndSelf.

Листинг 7.55. Проход вверх от объекта XElement методом AncestorsAndSelf

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName",
                new XText("Joe")),
            new XElement("NickName", "Joey")),
        new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach (XElement element in firstParticipant.
    Element("FirstName").Element("NickName").AncestorsAndSelf())
{
    Console.WriteLine(element.Name);
}

```

Результат должен быть таким же, как и при вызове метода Ancestors, за исключением того, что я также должен видеть имя элемента NickName в начале вывода:

```

' NickName
 FirstName
 BookParticipant
 BookParticipants

```

Рекурсивно вниз с помощью XContainer.Descendants()

В дополнение к рекурсивному проходу вверх вы можете также рекурсивно проходить вниз методом Descendants. Этот метод возвращает только элементы. Существует эквивалентный метод DescendantNodes для возврата всех узлов-потомков. Листинг 7.56 представляет тот же код, что и предыдущий, но с вызовом метода Descendants на элементе первого участника.

Листинг 7.56. Проход вниз от объекта XElement методом Descendants

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),

```

```

new XAttribute("type", "Author"),
new XElement("FirstName",
    new XText("Joe"),
    new XElement("NickName", "Joey")),
new XElement("LastName", "Rattz"),
new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
foreach (XElement element in firstParticipant.Descendants())
{
    Console.WriteLine(element.Name);
}

```

Результат выглядит так:

```

FirstName
NickName
LastName

```

Как видите, пример выполняет проход до конца по каждой ветви дерева XML.

Рекурсивно вниз с помощью XElement.DescendantsAndSelf()

Точно так же, как метод Ancestors имеет вариацию AncestorsAndSelf, у метода Descendants есть свой подобный аналог. Метод DescendantsAndSelf работает подобно Descendants, но с тем отличием, что включает в возвращенную последовательность сам элемент, на котором он вызван. В листинге 7.57 представлен пример, подобный предыдущему, но с вызовом DescendantsAndSelf вместо Descendants.

Листинг 7.57. Проход вниз от объекта XElement методом DescendantsAndSelf

```

 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XComment("This is a new author."),
            new XProcessingInstruction("AuthorHandler", "new"),
            new XAttribute("type", "Author"),
            new XElement("FirstName",
                new XText("Joe"),
                new XElement("NickName", "Joey")),
            new XElement("LastName", "Rattz"),
            new XElement("BookParticipant",
                new XAttribute("type", "Editor"),
                new XElement("FirstName", "Ewan"),
                new XElement("LastName", "Buckingham"))));
    foreach (XElement element in firstParticipant.DescendantsAndSelf())
    {
        Console.WriteLine(element.Name);
    }

```

Будет ли вывод включать также имя элемента firstParticipant? Давайте посмотрим:

```
BookParticipant
FirstName
NickName
LastName
```

Конечно, будет.

Вперед с помощью XNode.NodesAfterSelf()

Для этого примера в дополнение к изменению вызова `foreach` я добавил пару комментариев к элементу `BookParticipants`, чтобы сделать более наглядной разницу между извлечением узлов и элементов, поскольку `XComment` — узел, а не элемент. Листинг 7.58 содержит код этого примера.

Листинг 7.58. Проход вперед от текущего узла с использованием метода NodesAfterSelf

```
 XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants",
        new XComment("Begin Of List"), firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham")),
        new XComment("End Of List")));
foreach (XNode node in firstParticipant.NodesAfterSelf())
{
    Console.WriteLine(node);
}
```

Обратите внимание, что я добавил два комментария по соседству с двумя элементами `BookParticipant`. Эта модификация конструируемого документа XML будет выполнена для примеров `NodesAfterSelf`, `ElementsAfterSelf`, `NodesBeforeSelf` и `ElementsBeforeSelf`.

Это вызовет перечисление всех соседних узлов первого узла `BookParticipant`. Вот результат:

```
<BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
</BookParticipant>
<!--End Of List-->
```

Как видите, последний комментарий включен в вывод потому, что он является узлом. Не позволяйте этому выводу ввести вас в заблуждение. Метод `NodesAfterSelf` возвращает только два узла: элемент `BookParticipant`, чей атрибут `type` равен `Editor`, и комментарий `End Of List`. Остальные узлы — `FirstName` и `LastName` — отображаются просто потому, что вызывается метод `ToString` узла `BookParticipant`.

Имейте в виду, что этот метод возвращает узлы, а не элементы. Если вы хотите ограничить тип возвращаемых узлов, то для этого можете применить операцию TypeOf, которую я демонстрировал в предыдущих примерах. Но если вас интересуют элементы, то для этого есть метод ElementsAfterSelf.

Вперед с помощью XNode.ElementsAfterSelf()

Этот пример использует некоторые модификации документа XML, проведенные в листинге 7.58 и касающиеся добавления двух комментариев.

Чтобы получить последовательность только соседних элементов после указанного узла, вызывается метод ElementsAfterSelf, как показано в листинге 7.59.

Листинг 7.59. Проход вперед от текущего узла с использованием метода ElementsAfterSelf

```
XElement firstParticipant;
// Полный документ - со всеми мелочами.
XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants",
        new XComment("Begin Of List"), firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham")),
        new XComment("End Of List")));
foreach (XNode node in firstParticipant.ElementsAfterSelf())
{
    Console.WriteLine(node);
}
```

Код примера с этими модификациями выдаст следующий результат:

```
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
```

Обратите внимание, что на этот раз комментарий исключен, поскольку не является элементом. Опять-таки, элементы FirstName и LastName отображаются лишь потому, что являются содержимым элемента BookParticipant, который был извлечен, и потому что на этом элементе вызван метод ToString.

Назад с помощью XNode.NodesBeforeSelf()

Этот пример использует те же модификации в XML-документе, что проведены в листинге 7.58, касающиеся добавления двух комментариев.

Этот метод работает подобно NodesAfterSelf, за исключением того, что извлекает соседние узлы, находящиеся перед текущим. В коде примера, поскольку начальной ссылкой на документ является первый узел BookParticipant, я получаю ссылку на второй узел BookParticipant, используя свойство NextNode первого узла BookParticipant, так что остаются узлы для возврата, как показано в листинге 7.60.

Листинг 7.60. Проход назад от текущего узла

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),
    new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
    new XProcessingInstruction("BookCataloger", "out-of-print"),
    // Обратите внимание, что в следующей строке я сохраняю
    // ссылку на первый элемент BookParticipant.
    new XElement("BookParticipants",
        new XComment("Begin Of List"), firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham")),
        new XComment("End Of List")));
foreach (XNode node in firstParticipant.NextNode.NodesBeforeSelf())
{
    Console.WriteLine(node);
}

```

Эта модификация должна проявиться в возврате первого узла BookParticipant и первого комментария. Вот результат:

```

<!--Begin Of List-->
<BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
</BookParticipant>

```

Интересно! Я ожидал возврата двухузлов — комментария и первого BookParticipant в обратном порядке. Я ожидал, что метод начнет со ссылаемого узла и построит последовательность через свойство PreviousNode. Возможно, так и случилось, но затем была вызвана операция Reverse или InDocumentOrder. Операцию InDocumentOrder я опишу в следующей главе. Опять-таки, не позволяйте узлам FirstName и LastName запутать вас. Метод NodesBeforeSelf не вернул их. Они присутствуют здесь лишь потому, что методом Console.WriteLine был вызван метод ToString на первом узле BookParticipant, и поэтому они были отображены.

Назад с помощью XNode.ElementsBeforeSelf()

Этот пример использует те же модификации документа XML, что и в листинге 7.58, касающиеся добавления двух комментариев.

Подобно тому, как у метода NodesAfterSelf есть компаньон — метод ElementsAfterSelf для возврата только элементов, так и у метода NodesBeforeSelf имеется дополняющий его метод ElementsBeforeSelf, который возвращает только соседние элементы, предшествующие ссылаемому узлу, как показано в листинге 7.61.

Листинг 7.61. Проход назад от текущего узла

```

 XElement firstParticipant;
 // Полный документ - со всеми мелочами.
 XDocument xDocument = new XDocument(
    new XDeclaration("1.0", "UTF-8", "yes"),

```

```

new XDocumentType("BookParticipants", null, "BookParticipants.dtd", null),
new XProcessingInstruction("BookCataloger", "out-of-print"),
// Обратите внимание, что в следующей строке я сохраняю
// ссылку на первый элемент BookParticipant.
new XElement("BookParticipants",
    new XComment("Begin Of List"), firstParticipant =
    new XElement("BookParticipant",
        new XAttribute("type", "Author"),
        new XElement("FirstName", "Joe"),
        new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham")),
    new XComment("End Of List")));
foreach (XNode node in firstParticipant.NextNode.ElementsBeforeSelf())
{
    Console.WriteLine(node);
}

```

Обратите внимание, что я опять получаю ссылку на второй узел BookParticipant через свойство NextNode. Войдет ли комментарий в вывод?

```

<BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
</BookParticipant>

```

Конечно, нет, потому что он не является элементом.

Модификация XML

Модификация данных XML теперь стала проще, чем когда-либо, благодаря LINQ to XML. Оперируя всего горстью методов, вы можете выполнять любые модификации, которые только пожелаете. Будь то добавление, изменение или удаление узлов или элементов — всегда найдется метод, который может выполнить работу.

Как я повторяю вновь и вновь, в LINQ to XML большую часть времени вы будете работать с объектами XElement. Поэтому большинство приведенных примеров связано с обработкой элементов. Классы LINQ to XML, унаследованные от XNode, рассматриваются первыми, а за ними следует раздел, посвященный атрибутам.

Добавление узлов

В этом разделе, посвященном добавлению новых узлов к дереву XML, я начну с базового примера кода, представленного в листинге 7.62.

Листинг 7.62. Базовый пример с единственным участником подготовки книги

```

// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine(xDocument);

```

Этот код производит дерево XML с единственным участником подготовки книги. Вот его вывод:

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

Описание различных методов добавления узлов я начну с этого базового кода.

На заметку! Хотя все последующие примеры добавляют элементы, та же техника работает со всеми классами LINQ to XML, унаследованными от класса XNode.

В дополнение к следующим способам добавления узлов не забудьте заглянуть в раздел “`XElement.SetElementValue()` на дочерних объектах `XElement`” далее в этой главе.

`XContainer.Add()` (`AddLast`)

Метод, который вы будете использовать наиболее часто для добавления элементов в дерево XML — это `Add`. Он добавляет узел в конец списка дочерних узлов по отношению к указанному узлу. Пример приведен в листинге 7.63.

Листинг 7.63. Добавление узла в конец списка дочерних узлов методом `Add`

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").Add(
  new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham")));
Console.WriteLine(xDocument);
```

В приведенном коде вы можете видеть, что я начал с базового кода и добавил элемент `BookParticipant` к элементу документа `BookParticipants`. Как видите, я использую здесь метод документа `Element`, чтобы получить элемент `BookParticipants`, и затем добавляю элемент к списку его дочерних элементов, используя для этого метод `Add`. Это добавляет новый элемент к списку дочерних:

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

Метод Add добавляет вновь сконструированный элемент BookParticipant в конец списка дочерних элементов по отношению к BookParticipants. Как видите, метод Add столь же гибок, как и конструктор XElement, и следует тем же правилам относительно его аргументов, применимым при функциональном конструировании.

XContainer.AddFirst()

Чтобы добавить узел в начало списка дочерних узлов, применяйте метод AddFirst. Используя тот же код, что и раньше, за исключением вызова метода AddFirst, получим код, приведенный в листинге 7.64.

Листинг 7.64. Добавление узла в начало списка дочерних узлов методом AddFirst

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").AddFirst(
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham")));
Console.WriteLine(xDocument);
```

Как и можно было ожидать, вновь добавленный элемент BookParticipant появится в голове списка дочерних элементов BookParticipants:

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

Можно ли себе представить более простой способ манипуляций XML? Думаю, нет.

XNode.AddBeforeSelf()

Чтобы вставить узел в определенное место списка дочерних узлов, получите ссылку либо на предшествующий узел, либо на узел, непосредственно следующий за местом вставки, и вызовите метод AddBeforeSelf или AddAfterSelf.

Я использую дерево XML, произведенное примером метода Add из листинга 7.63, в качестве начальной точки, и вставлю новый узел между двумя уже существующими элементами BookParticipant, как показано в листинге 7.65.

Листинг 7.65. Добавление узла в список дочерних узлов методом AddBeforeSelf

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
```

```

        new XElement("FirstName", "Joe"),
        new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").Add(
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham")));
xDocument.Element("BookParticipants").
Elements("BookParticipant").
Where(e => ((string)e.Element("FirstName")) == "Ewan").
Single< XElement>().AddBeforeSelf(
    new XElement("BookParticipant",
        new XAttribute("type", "Technical Reviewer"),
        new XElement("FirstName", "Fabio"),
        new XElement("LastName", "Ferracchiati")));
Console.WriteLine(xDocument);

```

Для напоминания о стандартных операциях запросов из части 2 этой книги, "LINQ to Objects", и для интеграции того, о чем вы узнали в этой главе, я решил найти элемент BookParticiant, перед которым хочу вставить новый, используя для этого арсенала операций LINQ. Обратите внимание, что я использую метод Element для углубления в документ, чтобы выбрать элемент BookParticipants. Затем я выбираю дочерние элементы BookParticipants по имени BookParticipant, у которых есть дочерний элемент по имени FirstName со значением "Ewan". Поскольку я знаю, что этому критерию отвечает только один элемент BookParticipant, и потому, что мне нужен объект типа XElement, на котором можно вызвать метод AddBeforeSelf, я вызываю операцию Single, чтобы получить объект XElement BookParticipant. Это дает мне ссылку на элемент BookParticipant, перед которым я хочу вставить новый XElement.

Также обратите внимание, что в вызове операции Where я выполняю приведение элемента FirstName к string, чтобы использовать средство извлечения значения узла для получения значения элемента FirstName, чтобы проверить его эквивалентность с "Ewan".

Получив ссылку на правильный элемент BookParticipant, я просто вызываю метод AddBeforeSelf, и — пожалуйста:

```

<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Technical Reviewer">
    <FirstName>Fabio</FirstName>
    <LastName>Ferracchiati</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

```

Как я и хотел, новый BookParticipant вставлен перед элементом BookParticipant, у которого значение элемента FirstName равно "Ewan".

XNode.AddAfterSelf()

После всех моих усилий, предпринятых для получения ссылки на второй элемент BookParticipant, предпринятых в предыдущем примере, пример из листинга 7.66 м

жет разочаровать. Я просто получу ссылку на первый элемент BookParticipant, используя метод Element, и добавлю новый элемент BookParticipant сразу после него, применив для этого метод AddAfterSelf.

Листинг 7.66. Добавление узла в определенное место списка дочерних узлов методом AddAfterSelf

```
// Документ с одним участником подготовки книги.
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
xDocument.Element("BookParticipants").Add(
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham")));
xDocument.Element("BookParticipants").
Element("BookParticipant").AddAfterSelf(
    new XElement("BookParticipant",
        new XAttribute("type", "Technical Reviewer"),
        new XElement("FirstName", "Fabio"),
        new XElement("LastName", "Ferracchiat")));
Console.WriteLine(xDocument);
```

После предыдущего этот пример покажется тривиальным:

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

Удаление узлов

Удаление узлов выполняется одним из двух методов: Remove или RemoveAll. В дополнение к чтению о последующих способах удаления узлов не забудьте заглянуть в раздел " XElement.SetValue () на дочерних объектах XElement" далее в этой главе.

XNode.Remove ()

Метод Remove удаляет из дерева XML любой узел, а также все его дочерние узлы и атрибуты. В первом примере я конструирую дерево XML и сохраняю ссылку на первый элемент, описывающий участника подготовки книги, как делал это в некоторых предыдущих примерах. Затем я отображаю дерево XML после конструирования, но перед удалением узлов. Затем удаляю первый элемент, описывающий участника книги, и отображаю полученное в результате дерево XML, как показано в листинге 7.67.

Листинг 7.67. Удаление определенного узла методом Remove

```
// Это я использую для сохранения ссылки на один из элементов дерева XML.
 XElement firstParticipant;
 Console.WriteLine(System.Environment.NewLine + "Before node deletion");
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
 Console.WriteLine(xDocument);
 firstParticipant.Remove();
 Console.WriteLine(System.Environment.NewLine + "After node deletion");
 Console.WriteLine(xDocument);
```

Если все пойдет, как запланировано, я должен получить дерево XML сначала с первым элементом, описывающим участника, а затем без него:

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

Как видите, первый элемент BookParticipant исчез после удаления.

IEnumerable<T>.Remove()

В предыдущем случае я вызывал метод Remove на единственном объекте XElement. Я могу также вызвать Remove на последовательности (IEnumerable<T>). В листинге 7.68 представлен пример, где я использую метод Descendants документа, чтобы рекурсивно пройти вниз по всему дереву XML, возвратив только те элементы, чье имя — FirstName, используя для этого операцию Where. Затем я вызываю метод Remove на результирующей последовательности.

Листинг 7.68. Удаление последовательности узлов методом Remove

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
```

```

    new XElement("LastName", "Rattz"),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
xDocument.Descendants().Where(e => e.Name == "FirstName").Remove();
Console.WriteLine(xDocument);

```

Мне нравится этот пример, потому что здесь я могу связать вместе все элементы LINQ. Я использую метод `XDocument.Descendants` для получения всех дочерних узлов, возвращенных в последовательности, а затем вызываю стандартную операцию запроса `Where` для фильтрации только тех узлов, что отвечают критерию поиска, которым в данном случае является имя элемента `FirstName`. Он возвращает последовательность, на которой я затем вызываю метод `Remove`. Блеск! И вот результаты:

```

<BookParticipants>
  <BookParticipant type="Author">
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

```

Обратите внимание, что все узлы `FirstName` исчезли.

`XElement.RemoveAll()`

Иногда вы можете пожелать удалить содержимое элемента, но не сам элемент. Именно для этого предназначен метод `RemoveAll`. Пример приведен в листинге 7.69.

Листинг 7.69. Удаление содержимого узла методом `RemoveAll`

```

XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine(System.Environment.NewLine + "Before removing the content.");
Console.WriteLine(xDocument);
xDocument.Element("BookParticipants").RemoveAll();
Console.WriteLine(System.Environment.NewLine + "After removing the content.");
Console.WriteLine(xDocument);

```

Здесь я сначала отображаю документ перед удалением содержимого узла `BookParticipants`. Затем удаляю содержимое узла `BookParticipant` и снова отображаю документ. Вот как выглядит результат:

```

Before removing the content.
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>

```

```

<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
</BookParticipants>
After removing the content.
<BookParticipants />

```

Обновление узлов

Несколько подклассов XNode — вроде XElement, XText и XComment — имеют свойство Value, которое может быть обновлено непосредственно. Другие, такие как XDocumentType и XProcessingInstruction, имеют свои специфичные свойства, которые также могут обновляться. У элементов в дополнение к модификации свойства Value можно изменять их значение вызовом методов XElement.SetElementValue или XContainer.ReplaceAll, описанных далее в этой главе.

XElement.Value на объектах XElement, XText.Value на объектах XText и XComment.Value на объектах XComment

Каждый из этих подклассов XNode имеет свойство Value, которое может быть установлено для обновления значения узла. Листинг 7.70 демонстрирует их все.

Листинг 7.70. Обновление значения узла

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine("Before updating nodes:");
Console.WriteLine(xDocument);
// Теперь обновим элемент, комментарий и текстовый узел.
firstParticipant.Element("FirstName").Value = "Joey";
firstParticipant.Nodes().OfType<XComment>().Single().Value =
    "Author of Pro LINQ: Language Integrated Query in C# 2008.";
(XElement)firstParticipant.Element("FirstName").NextNode
    .Nodes().OfType<XText>().Single().Value = "Rattz, Jr.";
Console.WriteLine("After updating nodes:");
Console.WriteLine(xDocument);

```

В этом примере я сначала обновляю элемент FirstName, используя свойство Value, затем — комментарий, используя его свойство Value, и, наконец — элемент LastName, обращаясь к его значению через свойство Value объекта XText. Обратите внимание на гибкость, которую предлагает LINQ to XML для получения ссылок на разные объекты, которые я хочу обновить. Только имейте в виду, что вообще-то мне незачем было обращаться к значению элемента LastName, получая объект XText из его дочерних узлов. Я сделал это исключительно в демонстрационных целях. А иначе мне достаточно было напрямую обратиться к его свойству Value. Вот результат:

```

Before updating nodes:
<BookParticipants>
  <BookParticipant type="Author">
    <!--This is a new author.-->
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

```

```

After updating nodes:
<BookParticipants>
  <BookParticipant type="Author">
    <!--Author of Pro LINQ: Language Integrated Query in C# 2008.-->
    <FirstName>Joey</FirstName>
    <LastName>Rattz, Jr.</LastName>
  </BookParticipant>
</BookParticipants>

```

Как видите, все значения узлов обновлены.

XDocumentType.Name, XDocumentType.PublicId, XDocumentType.SystemId и XDocumentType.InternalSubset на объектах XDocumentType

Чтобы обновить узел типа документа, класс XDocumentType представляет четыре свойства для обновления его значений. Листинг 7.71 содержит код примера, демонстрирующего их.

Листинг 7.71. Обновление типа документа

```

// Это я использую для хранения ссылки на DocumentType для дальнейшего обращения.
XDocumentType docType;
XDocument xDocument = new XDocument(
  docType = new XDocumentType("BookParticipants", null,
    "BookParticipants.dtd", null),
  new XElement("BookParticipants"));
Console.WriteLine("Before updating document type:");
Console.WriteLine(xDocument);
docType.Name = "MyBookParticipants";
docType.SystemId = "http://www.somewhere.com/DTDs/MyBookParticipants.DTD";
docType.PublicId = "-//DTDs//TEXT Book Participants//EN";
Console.WriteLine("After updating document type:");
Console.WriteLine(xDocument);

```

Вот результат работы этого кода:

```

Before updating document type:
<!DOCTYPE BookParticipants SYSTEM "BookParticipants.dtd">
<BookParticipants />
After updating document type:
<!DOCTYPE MyBookParticipants PUBLIC "-//DTDs//TEXT Book Participants//EN"
"http://www.somewhere.com/DTDs/MyBookParticipants.DTD">
<BookParticipants />

```

XProcessingInstruction.Target на объектах

XProcessingInstruction и XProcessingInstruction.Data на объектах XProcessingInstruction

Чтобы обновить значение инструкции обработки, просто модифицируйте свойства Target и Data объекта XProcessingInstruction. Пример можно найти в листинге 7.72.

Листинг 7.72. Обновление инструкции обработки

```

// Это я использую для хранения ссылки для дальнейшего обращения.
XProcessingInstruction procInst;
XDocument xDocument = new XDocument(
  new XElement("BookParticipants"),
  procInst = new XProcessingInstruction("BookCataloger", "out-of-print"));

```

```

Console.WriteLine("Before updating processing instruction:");
Console.WriteLine(xDocument);
procInst.Target = "BookParticipantContactManager";
procInst.Data = "update";
Console.WriteLine("After updating processing instruction:");
Console.WriteLine(xDocument);

```

Взглянем на вывод:

```

Before updating processing instruction:
<BookParticipants />
<?BookCataloger out-of-print?>
After updating processing instruction:
<BookParticipants />
<?BookParticipantContactManager update?>

```

XElement.ReplaceAll()

Метод ReplaceAll удобен для замены целого поддерева XML, начинающегося с элемента. Вы можете передать простое значение, такое как новая строка или значение числового типа; или же, поскольку есть перегруженная версия метода, принимающая множество объектов через ключевое слово params — заменить целое поддерево. Метод ReplaceAll также заменяет атрибуты. Листинг 7.73 содержит код примера.

Листинг 7.73. Применение ReplaceAll для замены всего поддерева элемента

```

// Это я использую для сохранения ссылки на один из элементов дерева XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(System.Environment.NewLine + "Before updating elements:");
 Console.WriteLine(xDocument);
 firstParticipant.ReplaceAll(
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"));
 Console.WriteLine(System.Environment.NewLine + "After updating elements:");
 Console.WriteLine(xDocument);

```

Обратите внимание, что, заменяя содержимое методом ReplaceAll, я опустил спецификацию атрибута. Как можно было ожидать, содержимое заменено:

```

Before updating elements:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After updating elements:
<BookParticipants>
  <BookParticipant>
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>

```

Обратите внимание, что атрибут типа BookParticipant пропал. Интересно, что атрибуты не являются дочерними узлами элемента. Однако, тем не менее, метод ReplaceAll заменяет их.

XElement.SetValue() на дочерних объектах XElement

Не позволяйте этому методу с простым именем ввести вас в заблуждение. Он обладает способностью добавлять, изменять и удалять элементы. Более того, он выполняет эти операции на дочерних элементах того элемента, на котором вы его вызываете. Иначе говоря, вы вызываете метод SetValue на родительском элементе, чтобы затронуть его содержимое, т.е. дочерние элементы.

При вызове метода SetValue вы передаете ему имя дочернего элемента, значение которого вы хотите установить. Если дочерний элемент найден по имени, его значение обновляется, если только переданное значение не равно null. Если передано значение null, этот найденный дочерний элемент удаляется. Если же элемент с указанным именем не найден, он будет добавлен с переданным значением. Замечательный метод!

Также метод SetValue затрагивает только первый дочерний элемент с указанным именем, который он находит. Все последующие элементы с тем же именем не затрагиваются — будь то изменение значения или удаление элемента по причине передачи значения null. Листинг 7.74 демонстрирует все применения этого метода: обновление, добавление и удаление.

Листинг 7.74. Применение XElement.SetValue для обновления, добавления и удаления дочерних элементов

```
// Это я использую для сохранения ссылки на один из элементов дерева XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Before updating elements:");
Console.WriteLine(xDocument);
// Во-первых, я использую XElement.SetValue для обновления значения элемента.
// Поскольку есть элемент по имени FirstName, его значение будет изменено на Joseph.
firstParticipant.SetValue("FirstName", "Joseph");
// Во-вторых, я использую XElement.SetValue для добавления элемента.
// Поскольку элемента по имени MiddleInitial нет, он будет добавлен.
firstParticipant.SetValue("MiddleInitial", "C");
// В-третьих, я использую XElement.SetValue для удаления элемента.
// Установка значения элемента в null удалит его.
firstParticipant.SetValue("LastName", null);
Console.WriteLine(System.Environment.NewLine + "After updating elements:");
Console.WriteLine(xDocument);
```

Как видите, сначала я вызываю метод SetValue на дочернем элементе firstParticipant по имени FirstName. Поскольку элемент с таким именем существует, его значение будет обновлено. Затем я вызываю метод SetValue на дочернем элементе firstParticipant по имени MiddleInitial. Поскольку элемента с та-

ким именем нет, он будет добавлен. И, наконец, я вызываю метод `SetElementValue` из дочернем элементе `firstParticipant` по имени `LastName`, передавая `null`. Поскольку передан `null`, элемент `LastName` будет удален. Смотрите, какую гибкость обеспечивает метод `SetElementValue!` Уверен, вам не терпится увидеть результаты:

```
Before updating elements:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After updating elements:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joseph</FirstName>
    <MiddleInitial>C</MiddleInitial>
  </BookParticipant>
</BookParticipants>
```

Неплохо, правда? Значение элемента обновлено, добавлен элемент `MiddleInitial` и удален элемент `LastName`.

Внимание! Исходя из того, что вызов метода `SetElementValue` со значением `null` удаляет узел, не следует думать, что установка вручную значения элемента в `null` — это то же самое, что удаление его средствами LINQ to XML API. Это просто поведение метода `SetElementValue`. Если вы попытаетесь установить значение элемента в `null`, используя свойство `Value`, будет сгенерировано исключение.

Атрибуты XML

Как я ранее упоминал, в API-интерфейсе LINQ to XML атрибуты реализованы классом `XAttribute`, и в отличие от W3C XML DOM API, они не унаследованы от узла. Поэтому они не имеют отношений наследования с элементами. Однако в LINQ to XML API работать с ними так же просто, как и с элементами. Давайте посмотрим, как это делается.

Создание атрибута

Атрибуты создаются точно так же, как и элементы, и большинство других классов LINQ to XML. Эта тема рассматривалась ранее в разделе “Создание атрибутов с помощью `XAttribute`” настоящей главы.

Проход по атрибутам

Можно организовать проход по атрибутам, используя свойства `XElement.FirstAttribute`, `XElement.LastAttribute`, `XAttribute.NextAttribute` и `XAttribute.PreviousAttribute`,³ также методы `XElement.Attribute` и `XElement.Attributes`. Все они описаны в ⁴ следующих разделах.

Вперед с помощью `XElement.FirstAttribute`

Вы можете получить доступ к атрибутам элемента, обратившись к его первому атрибуту через свойство элемента `FirstAttribute`. Пример представлен в листинге 7.75.

Листинг 7.75. Обращение к первому атрибуту элемента через свойство FirstAttribute

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(firstParticipant.FirstAttribute);
```

Вывод этого кода:

type="Author"

Вперед с помощью XAttribute.NextAttribute

Чтобы выполнить проход вперед по атрибутам элемента, обратитесь к свойству атрибута NextAttribute. Пример представлен в листинге 7.76.

Листинг 7.76. Обращение к следующему атрибуту элемента через свойство NextAttribute

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(firstParticipant.FirstAttribute.NextAttribute);
```

Обратите внимание, что я использую свойство FirstAttribute, чтобы получить ссылку на первый атрибут, а затем на нем — к ссылке NextAttribute. И вот результат:

experience="first-time"

Если свойство атрибута NextAttribute равно null, значит, текущий атрибут у элемента является последним.

Назад с помощью XAttribute.PreviousAttribute

Чтобы выполнить обратный проход по атрибутам элемента, обращайтесь к свойству атрибута PreviousAttribute. Пример представлен в листинге 7.77.

Листинг 7.77. Обращение к предыдущему атрибуту элемента через свойство PreviousAttribute

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(firstParticipant.FirstAttribute.NextAttribute.PreviousAttribute);
```

Обратите внимание, что я выстраиваю в цепочку `FirstAttribute` и `NextAttribute`, чтобы получить ссылку на второй атрибут, от которого выполняется шаг назад. Это должно привести меня обратно к первому атрибуту. И вот результат:

```
type="Author"
```

Так и есть! Если свойство `PreviousAttribute` равно `null`, значит, данный атрибут — первый у элемента.

Назад с помощью XElement. LastAttribute

Для получения доступа к самому последнему атрибуту элемента, чтобы затем выполнить обратный проход по атрибутам, используйте свойство `LastAttribute`, как показано в листинге 7.78.

Листинг 7.78. Обращение к последнему атрибуту элемента через свойство LastAttribute

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(firstParticipant.LastAttribute);
```

Это должно вывести атрибут `language`. Посмотрим:

```
language="English"
```

Превосходно! Я еще ни разу не употреблял этого слова раньше. Пришлось воспользоваться средством проверки орфографии, чтобы написать его правильно.

XElement.Attribute()

Этот метод принимает имя атрибута и возвращает *первый* атрибут с указанным именем, если таковой имеется. Пример показан в листинге 7.79.

Листинг 7.79. Обращение к атрибуту методом Attribute

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
 Console.WriteLine(firstParticipant.Attribute("type").Value);
```

Я использую здесь метод `Attribute`, чтобы вернуть ссылку на атрибут `type`. Затем я отображаю значение атрибута, используя свойство `Value`. Если все идет, как ожидается, вывод должен быть таким:

```
Author
```

Напомню, однако, что вместо получения значения атрибута через свойство `Value` мог бы просто привести атрибут к типу `string`.

XElement.Attributes()

Можно получить доступ сразу ко всем атрибутам элемента через метод `Attributes`. Этот метод возвращает последовательность объектов `XAttribute`. В листинге 7.80 показан пример.

Листинг 7.80. Обращение ко всем атрибутам элемента методом `Attributes`

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
foreach(XAttribute attr in firstParticipant.Attributes())
{
    Console.WriteLine(attr);
}
```

Вывод будет таким:

```
type="Author"
experience="first-time"
```

Модификация атрибута

Есть несколько методов и свойств, которые могут быть использованы для модификации атрибутов. В этом разделе я опишу их.

Добавление атрибутов

Как я уже указывал, существует фундаментальное отличие в способах обработки атрибутов между интерфейсами W3C XML DOM API и LINQ to XML API. В W3C API атрибут — это дочерний узел по отношению к узлу, атрибутом которого он является. В LINQ to XML API атрибуты не являются дочерними узлами такого узла. Вместо этого атрибуты представляют собой пары “имя-значение”, доступ к которым осуществляется через метод `Attributes` элемента либо через свойство `FirstAttribute`. Об этом важно помнить.

Однако работа с атрибутами очень похожа на работу с элементами. Методы и свойства атрибутов очень похожи на методы и свойства, предназначенные для работы с элементами.

Чтобы добавить атрибут к элементу можно использовать следующие методы:

- `XElement.Add()`
- `XElement.AddFirst()`
- `XElement.AddBeforeThis()`
- `XElement.AddAfterThis()`

В примерах, представляющих каждый из этих методов в разделе “Добавление узлов”, приведенных ранее в этой главе, атрибуты также добавлялись. Обратитесь к этим примерам, чтобы увидеть, как добавляются атрибуты. В добавок загляните в раздел, посвященный методу `XElement.SetAttributeValue`, далее в этой главе.

Удаление атрибутов

Удаление атрибутов может выполняться либо методом `XAttribute.Remove()`, либо `IEnumerable<T>.Remove` — в зависимости от того, нужно вам удалить единственный атрибут или последовательность атрибутов.

В дополнения к описанным ниже способам удаления атрибутов загляните в раздел " XElement.SetValue()" далее в этой главе.

`XElement.Remove()`

Подобно классу `XNode`, класс `XAttribute` также имеет метод `Remove`. Пример его использования представлен в листинге 7.81.

Листинг 7.81. Удаление атрибута

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Before removing attribute:");
Console.WriteLine(xDocument);
firstParticipant.Attribute("type").Remove();
Console.WriteLine(System.Environment.NewLine + "After removing attribute:");
Console.WriteLine(xDocument);
```

Как видите, я использую метод `Attribute`, чтобы получить ссылку на атрибут, который хочу удалить, а затем вызываю метод `Remove` на нем. Ниже представлен результат работы этого примера:

```
Before removing attribute:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After removing attribute:
<BookParticipants>
  <BookParticipant>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Обратите внимание, что атрибут `type` исчез.

`IEnumerable<T>.Remove()`

Точно так же, как метод `IEnumerable<T>.Remove` позволяет удалять последовательность узлов, с его помощью можно удалить и все атрибуты элемента, как показано в листинге 7.82.

Листинг 7.82. Удаление всех атрибутов элемента

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
XElement firstParticipant;
```

```

xDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Before removing attributes:");
Console.WriteLine(xDocument);
firstParticipant.Attributes().Remove();
Console.WriteLine(System.Environment.NewLine + "After removing attributes:");
Console.WriteLine(xDocument);

```

В приведенном примере я вызываю метод Attributes, возвращающий последовательность атрибутов элемента, на котором вызван этот метод, а затем вызываю метод Remove на этой возвращенной последовательности, чтобы удалить их все. Это выглядит таким простым и интуитивным, что я, наверно зря трачу ваше время, объясняя все это. Вот результат:

```

Before removing attributes:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After removing attributes:
<BookParticipants>
  <BookParticipant>
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

```

Атрибуты исчезли, словно по волшебству.

Обновление атрибутов

Чтобы обновить значение атрибута, используйте свойство XAttribute.Value.

На заметку! В дополнение к использованию свойства XAttribute.Value для обновления атрибутов не забудьте заглянуть в раздел "XElement.SetAttributeValue()" далее в этой главе.

Обновление значения атрибута легко осуществляется через его свойство Value. Пример приведен в листинге 7.83.

Листинг 7.83. Изменение значения атрибута

```

// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine +
    "Before changing attribute's value:");
Console.WriteLine(xDocument);

```

```

firstParticipant.Attribute("experience").Value = "beginner";
Console.WriteLine(System.Environment.NewLine + "After changing attribute's value:");
Console.WriteLine(xDocument);

```

Обратите внимание, что я использую метод `Attribute` для получения ссылки на атрибут `experience`. Результат работы этого кода выглядит так:

```

Before changing attribute's value:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After changing attribute's value:
<BookParticipants>
  <BookParticipant type="Author" experience="beginner">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>

```

Как видите, значение атрибута `experience` изменилось с `"first-time"` на `"beginner"`.

`XElement.SetAttributeValue()`

Чтобы сохранить симметрию с методами элементов, можно ожидать, что метод `SetAttributeValue` должен быть столь же мощным, что и метод `SetElementValue`. Так оно и есть. Метод `XElement.SetAttributeValue()` обладает способностью добавлять, удалять и обновлять атрибут.

Передача этому методу имени несуществующего атрибута приводит к его добавлению. Передача имени существующего атрибута со значением, отличным от `null`, вызывает обновление значения указанного атрибута. Передача имени существующего атрибута и значения `null` вызовет удаление соответствующего атрибута. Пример всех трех случаев приведен в листинге 7.84.

Листинг 7.84. Применение `SetAttributeValue` для добавления, удаления и обновления атрибутов

```

// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
 XDocument xDocument = new XDocument(
  new XElement("BookParticipants", firstParticipant =
   new XElement("BookParticipant",
    new XAttribute("type", "Author"),
    new XAttribute("experience", "first-time"),
    new XElement("FirstName", "Joe"),
    new XElement("LastName", "Rattz"))));
Console.WriteLine(System.Environment.NewLine + "Before changing the attributes:");
Console.WriteLine(xDocument);

// Этот вызов обновит значение атрибута type,
// потому что атрибут по имени "type" существует.
firstParticipant.SetAttributeValue("type", "beginner");

// Этот вызов добавит атрибут, потому что атрибут в указанным именем не существует.
firstParticipant.SetAttributeValue("language", "English");

// Этот вызов удалит атрибут, потому что атрибут с указанным
// именем существует, а в качестве значения передано null.
firstParticipant.SetAttributeValue("experience", null);

Console.WriteLine(System.Environment.NewLine + "After changing the attributes:");
Console.WriteLine(xDocument);

```

Как видите, в этом примере сначала я обновляю значение существующего атрибута, затем добавляю новый атрибут, и, наконец, удаляю существующий атрибут, передав значение null. Вот результат:

```
before changing the attributes:
<BookParticipants>
  <BookParticipant type="Author" experience="first-time">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
After changing the attributes:
<BookParticipants>
  <BookParticipant type="beginner" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
</BookParticipants>
```

Аннотации XML

Программный интерфейс LINQ to XML API предоставляет возможность ассоциировать объект пользовательских данных с любым классом-наследником XObject через аннотации. Это позволяет разработчику приложений назначать данные любого типа элементу, документу или любому другому объекту, чей класс наследуется от XObject. Объект может быть дополнительными ключами для данных элемента; он может быть объектом, который разберет содержимого элемента и сохранит его в себе в ином виде, либо чем-то другим, что вы захотите.

Добавление аннотаций с помощью XObject.AddAnnotation()

Добавление аннотаций осуществляется методом XObject.AddAnnotation. Вот его прототип:

```
void XObject.AddAnnotation(object annotation);
```

Обращение к аннотациям с помощью XObject. Annotation() или XObject.Annotations()

Обращение к аннотациям осуществляется с использованием методов XObject.Annotation или XObject.Annotations. Ниже показаны их прототипы:

```
object XObject.Annotation(Type type);
† XObject.Annotation<T>();
IEnumerable<object> XObject.Annotations(Type type);
IEnumerable<T> XObject.Annotations<T>();
```

Внимание! При извлечении аннотаций вы должны передавать действительный тип объекта, а не базовый класс или интерфейс. В противном случае аннотация не будет найдена.

Удаление аннотаций с помощью XObject.RemoveAnnotations()

Удаление аннотаций осуществляется методом `XObject.RemoveAnnotations`. У него есть два прототипа:

```
void XObject.RemoveAnnotations(Type type);
void XObject.RemoveAnnotations<T>();
```

Пример аннотаций

Для демонстрации аннотаций я создам один пример, который добавит, изъяет и удалит аннотации. В этом примере я использую мое привычное XML-дерево `BookParticipants`. Мне нужен способ ассоциировать обработчик каждого `BookParticipant` на основе атрибута `type`. В данном примере обработчик будет просто отображать элемент в специфичном для атрибута `type` формате: один формат для авторов и другой — для редакторов.

Сначала мне понадобится пара классов — по одному для авторов и редакторов:

```
public class AuthorHandler
{
    public void Display(XElement element)
    {
        Console.WriteLine("AUTHOR BIO");
        Console.WriteLine("-----");
        Console.WriteLine("Name: {0} {1}",
            (string)element.Element("FirstName"),
            (string)element.Element("LastName"));
        Console.WriteLine("Language: {0}", (string)element.Attribute("language"));
        Console.WriteLine("Experience: {0}", (string)element.Attribute("experience"));
        Console.WriteLine("======" + System.Environment.NewLine);
    }
}
public class EditorHandler
{
    public void Display(XElement element)
    {
        Console.WriteLine("EDITOR BIO");
        Console.WriteLine("-----");
        Console.WriteLine("Name: {0}, (string)element.Element("FirstName"));
        Console.WriteLine(" {0}, (string)element.Element("LastName"));
        Console.WriteLine("======" + System.Environment.NewLine);
    }
}
```

Здесь нет ничего особенного. Мне нужно два обработчика, ведущих себя по-разному. В этом случае они будут отображать данные элемента в немного отличающемся формате. Конечно, они не обязательно должны только отображать данные. Они могут делать все, что вам нужно. Или же аннотации могут вообще не быть обработчиками. Они могут быть просто какими-то ассоциированными данными. Но в данном примере это обработчики.

Поскольку этот пример сложнее, чем обычно, я разделил части кода пояснениями, как показано в листинге 7.85.

Листинг 7.85. Добавление, извлечение и удаление аннотаций

```
// Я использую это для хранения ссылки на один из элементов в дереве XML.
 XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("experience", "first-time"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
// Отобразить документ для ссылки.
Console.WriteLine(xDocument + System.Environment.NewLine);
```

Все, что делается в этой части — это построение типичного документа XML, который будет использован далее, а также его отображение. В следующей части кода я выполню перечисление участников книги и для каждого создам экземпляр обработчика на основе значения атрибута type, и добавлю аннотацию к элементу для соответствующего обработчика.

```
// Добавлю некоторые аннотации на основе значения атрибута type.
foreach(XElement e in xDocument.Element("BookParticipants").Elements())
{
    if((string)e.Attribute("type") == "Author")
    {
        AuthorHandler aHandler = new AuthorHandler();
        e.AddAnnotation(aHandler);
    }
    else if((string)e.Attribute("type") == "Editor")
    {
        EditorHandler eHandler = new EditorHandler();
        e.AddAnnotation(eHandler);
    }
}
```

Теперь каждый элемент BookParticipant имеет добавленный к нему обработчик в виде аннотации, зависящий от его атрибута type. Теперь, когда каждый элемент имеет добавленный через аннотации обработчик, я выполню перечисление элементов, вызывая обработчик, извлеченный его из аннотации элемента:

```
AuthorHandler aHandler2;
EditorHandler eHandler2;
foreach(XElement e in xDocument.Element("BookParticipants").Elements())
{
    if((string)e.Attribute("type") == "Author")
    {
        aHandler2 = e.GetAnnotation<AuthorHandler>();
        if(aHandler2 != null)
        {
            aHandler2.Display(e);
        }
    }
    else if((string)e.Attribute("type") == "Editor")
    {
```

```

        eHandler2 = e.GetAnnotation<EditorHandler>();
        if(eHandler2 != null)
        {
            eHandler2.Display(e);
        }
    }
}

```

В этой точке отображающий обработчик будет вызван для каждого элемента. Конкретный вызываемый обработчик зависит от атрибута type. Далее я просто удаляю аннотации у каждого элемента:

```

foreach(XElement e in xDocument.Element("BookParticipants").Elements())
{
    if((string)e.Attribute("type") == "Author")
    {
        e.RemoveAnnotation<AuthorHandler>();
    }
    else if((string)e.Attribute("type") == "Editor")
    {
        e.RemoveAnnotation<EditorHandler>();
    }
}

```

Это порядочный кусок кода примера, но он состоит всего из четырех основных разделов. В первом разделе я строю документ XML и отображаю его. До сих пор вы видели это много раз. Во втором разделе я выполняю перечисление элементов BookParticipants, и на основе их атрибута type добавляю к каждому обработчик. В третьем разделе я вновь выполняю перечисление элементов BookParticipants, и на основе атрибута type извлекаю обработчик и вызываю метод Display объекта обработчика. В четвертом разделе я опять перечисляю элементы BookParticipants, удаляя аннотации.

Обратите внимание, что при обращении к атрибутам я привожу их к string, чтобы получить их значение.

Что нужно помнить — так это то, что эти аннотации могут быть любыми объектами данных, которые вы хотите ассоциировать с элементом.

И, наконец, вот результат:

```

<BookParticipants>
    <BookParticipant type="Author" experience="first-time" language="English">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
AUTHOR BIO
-----
Name:      Joe Rattz
Language:   English
Experience: first-time
=====
EDITOR BIO
-----
Name: Ewan
Buckingham
=====
```

Что еще важно отметить относительно результата — это то, что разные обработчики вызываются в зависимости от атрибута `type` с использованием аннотаций. Конечно объекты, которые вы добавляете в виде аннотаций, могут служить любой цели, а не обязательно быть какими-то обработчиками.

События XML

Программный интерфейс LINQ to XML API дает вам возможность зарегистрировать обработчик события, чтобы вы могли получать извещения всякий раз, когда объект, унаследованный от `XObject`, собирается модифицироваться либо уже модифицирован.

Первое, что вам нужно знать при регистрации обработчика события для объекта — событие будет инициировано на объекте, когда этот объект либо любой его производный объект изменится. Это значит, что если вы регистрируете обработчик события на документе или корневом элементе, то всякое изменение в дереве вызовет зарегистрированный вами метод. Поэтому не выдвигайте никаких предположений относительно типа данных объекта, который вызвал событие. Когда вызывается зарегистрированный вами метод, инициировавший событие объект будет передан в качестве отправителя события, и типом его данных будет `object`. Будьте очень осторожны, выполняя его приведения или обращаясь к его свойствам, либо вызывая его методы. Может оказаться, что его тип не тот, что вы ожидали. Я продемонстрирую это в листинге 7.86, где `object` на самом деле оказывается объектом `XText`, когда я ожидаю, что он относится к типу `XElement`.

И, наконец, имейте в виду, что конструирование XML не инициирует никаких событий. Да и как оно могло бы это делать? До конструирования никаких обработчиков событий еще не зарегистрировано. Только модификация или удаление уже существующего XML может инициировать возникновение события, и только в случае, если обработчик события зарегистрирован.

`XObject.Changing`

Это событие возникает, когда объект, унаследованный от `XObject`, собирается измениться, но перед действительным его изменением. Вы регистрируете обработчик события, добавляя объект типа `EventHandler` к событию `Changing` объекта, как показано ниже:

```
myobject.Changing += new
EventHandler<XObjectEventArgs>(MyHandler);
```

где делегат вашего метода должен соответствовать такой сигнатуре:

```
void MyHandler(object sender, XObjectEventArgs cea)
```

Объект `sender` — тот, что собирается изменяться, который и инициировал возникновение события. Аргументы события изменения, `cea`, содержат свойство по имени `ObjectChange` типа `XObjectChange`, указывающее тип изменения, которое должно произойти: `XObjectChange.Add`, `XObjectChange.Name`, `XObjectChange.Remove` или `XObjectChange.Value`.

`XObject.Changed`

Это событие инициируется после того, как объект-наследник `XObject` был изменен. Вы регистрируете обработчик события, добавляя объект типа `EventHandler` к событию `Changed` объекта, как показано ниже:

```
myobject.Changed += new EventHandler<XObjectEventArgs>(MyHandler);
```

где ваш делегат метода должен соответствовать следующей сигнатуре:

```
void MyHandler(object sender, XObjectEventArgs cea)
```

Объект `sender` — изменившийся объект, который и вызвал возникновение события. Аргумент события `change` по имени `cea` содержит свойство по имени `ObjectChange`, типа `XObjectChange`, указывающее на тип произошедшего изменения: `XObjectChange.Add`, `XObjectChange.Name`, `XObjectChange.Remove` или `XObjectChange.Value`.

Несколько примеров событий

Чтобы увидеть все, что необходимо для обработки событий `XObject`, понадобится пример. Однако прежде чем я покажу необходимый код, мне понадобятся некоторые обработчики событий, приведенные ниже.

Этот метод будет зарегистрирован для обработки события изменения элемента

```
public static void MyChangingEventHandler(object sender, XObjectEventArgs cea)
{
    Console.WriteLine("Type of object changing: {0}, Type of change: {1}",
        sender.GetType().Name, cea.ObjectChange);
}
```

Я зарегистрирую приведенный метод в качестве обработчика события, которое происходит, когда элемент изменен.

Ранее я упоминал, что событие инициируется, когда изменяется любой из объектов-потомков зарегистрированного объекта. Чтобы лучше продемонстрировать это, у меня есть также один дополнительный метод, который я зарегистрирую для вызова, когда объект изменен. Его единственное назначение — сделать более наглядным возникновение события `Changed`, независимо от того, изменен ли сам объект, или его потомок находящийся несколькими уровнями ниже. Этот метод представлен ниже.

Этот метод будет зарегистрирован для обработки события изменения XML-документа

```
public static void DocumentChangedHandler(object sender, XObjectEventArgs cea)
{
    Console.WriteLine("Doc: Type of object changed: {0}, Type of change: {1}{2}",
        sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine);
}
```

Единственное существенное изменение между методом `DocumentChangedHandler` и методом `MyChangedEventHandler` заключается в том, что метод `DocumentChangedHandler` начинает экранный вывод с префикса `"Doc:"`, чтобы указать на то, что вызван метод-обработчик для события `Changed` документа, а не обработчик того же события элемента.

Теперь взглянем на пример кода, показанный в листинге 7.86.

Листинг 7.86. Обработка события `XObject`

```
XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
```

Пока ничего нового. Как это делалось это много раз, здесь я создаю документ XML, используя функциональное конструирование, а затем отображаю его. Обратите также внимание, что, как и в большинстве предыдущих примеров, я сохранил ссылку на первый элемент BookParticipant. Это тот элемент, для событий которого я регистрирую обработчик.

```
firstParticipant.Changing += new
EventHandler<XObjectEventArgs>(MyChangingEventHandler);
firstParticipant.Changed += new
EventHandler<XObjectEventArgs>(MyChangedEventHandler);
xDocument.Changed += new
EventHandler<XObjectEventArgs>(DocumentChangedHandler);
```

Теперь зарегистрировано, что первый элемент BookParticipant должен принимать события Changing и Changed. Вдобавок я зарегистрировал обработчик для получения документом события Changed. Я сделал это для того, чтобы продемонстрировать, что вы получаете событие, даже когда изменяется объект-потомок, а не сам объект, для которого зарегистрирован обработчик. Теперь проведем изменение:

```
firstParticipant.Element("FirstName").Value = "Seph";
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
```

Все, что я сделал — изменил значение подэлемента FirstName элемента BookParticipant. Затем я отображаю результирующий XML-документ. Посмотрим на результат:

```
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
Type of object changing: XText, Type of change: Remove
Type of object changed: XText, Type of change: Remove
Doc: Type of object changed: XText, Type of change: Remove
Type of object changing: XText, Type of change: Add
Type of object changed: XText, Type of change: Add
Doc: Type of object changed: XText, Type of change: Add
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Seph</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Вы можете видеть документ в начале и конце вывода результатов, и значение элемента FirstName изменилось, как следовало ожидать. Что вас здесь должно заинтересовать — так это вывод, вызванный возникновением события, который находится между двумя выводами документа XML. Обратите внимание, что тип изменяемого объекта — XText. Вы ожидали этого? Я — нет. Я ожидал увидеть тип XElement. Устанавливая значение элемента в виде строкового литерала, легко забыть, что при этом автоматически для вас создается объект типа XText.

Если посмотреть на вывод события, становится немного яснее, что именно происходит, когда вы изменяете значение элемента. Вы можете видеть, что сначала значение XText, подлежащее изменению, сначала должно быть удалено, и оно удаляется. Затем также вы видите инициализацию события Changed документа. Отсюда видно, что поток событий распространяется вверх.

Затем вы видите ту же последовательность событий, но на этот раз объект XText добавляется. Таким образом, теперь вы знаете, что при изменении строкового значения элемента объект XText удаляется, а затем добавляется обратно.

В предыдущем примере я использовал именованные методы, но это не значит, что вы обязаны поступать так же. Я мог бы с тем же успехом применять анонимные методы или даже лямбда-выражения. В листинге 7.87 показан тот же пример, но на этот раз вместо регистрации уже реализованных методов обработчиков используются лямбда-выражения, чтобы "на лету" определить код, вызываемый событиями.

Листинг 7.87. Обработка события XObject с использованием лямбда-выражений

```

 XElement firstParticipant;
XDocument xDocument = new XDocument(
    new XElement("BookParticipants", firstParticipant =
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);
firstParticipant.Changing += new EventHandler<XObjectEventArgs>(
    (object sender, XObjectEventArgs cea) =>
    Console.WriteLine("Type of object changing: {0}, Type of change: {1}",
        sender.GetType().Name, cea.ObjectChange));
firstParticipant.Changed += (object sender, XObjectEventArgs cea) =>
    Console.WriteLine("Type of object changed: {0}, Type of change: {1}",
        sender.GetType().Name, cea.ObjectChange);
xDocument.Changed += (object sender, XObjectEventArgs cea) =>
    Console.WriteLine("Doc: Type of object changed: {0}, Type of change: {1}{2}",
        sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine);
xDocument.Changed += new XObjectEventHandler((sender, cea) =>
    Console.WriteLine("Doc: Type of object changed: {0}, Type of change: {1}{2}",
        sender.GetType().Name, cea.ObjectChange, System.Environment.NewLine));
firstParticipant.Element("FirstName").Value = "Seph";
Console.WriteLine("{0}{1}", xDocument, System.Environment.NewLine);

```

Теперь код совершенно самодостаточен и не зависит от ранее написанных методов обработчиков. Взглянем на результаты:

```

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

```

```
Type of object changing: XText, Type of change: Remove
Type of object changed: XText, Type of change: Remove
Doc: Type of object changed: XText, Type of change: Remove
Type of object changing: XText, Type of change: Add
Type of object changed: XText, Type of change: Add
Doc: Type of object changed: XText, Type of change: Add
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Seph</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Вывод мне кажется таким же, как и в предыдущем случае. Взглянув на этот пример, можете ли вы не восхититься лямбда-выражениями? Я не раз наблюдал первое впечатление разработчиков от LINQ. Многим нравятся разные аспекты, но общее впечатление состоит в том, что многим лямбда-выражения не по душе. Возможно, это потому, что это очень новое и необычное средство. Но когда вы видите подобный пример, что здесь может не понравиться? Думаю, вы согласитесь.

Трюк, забава или неопределенность?

Помните ли вы “проблему Хэллоуина”, о которой я говорил ранее в этой главе? Пожалуйста, избегайте соблазна в своих методах-обработчиках вносить изменения в область дерева XML, содержащую объект, для которого инициировано текущее событие. Это приведет к непредсказуемому эффекту для вашего дерева XML и возникающих событий.

Резюме

В настоящей главе я рассказал, как использовать LINQ to XML для создания, модификации и прохода по XML-документам, а также о том, как выполнять запросы LINQ на отдельном XML-объекте. Вы, вероятно, заметили, что новый API для создания и модификации данных XML — не роскошь, а необходимость для выполнения запросов LINQ. Вы не сможете хорошо проектировать данные на структуру XML, если не сможете создавать “на лету” элементы XML, инициализировать их значения и поместить в дерево XML в одном операторе. Программный интерфейс W3C DOM XML API полностью лишен гибкости, необходимой для выполнения запроса LINQ. И это, к счастью для нас, привело к появлению совершенно нового XML API.

Хотя эта глава была полезна для демонстрации выполнения базовых запросов LINQ на данных XML, вы не могли не заметить серьезных ограничений, присущих запросам LINQ. То есть все запросы, что я выполнял здесь, имели дело с единственным объектом XML, таким как элемент. Я запрашивал предков элемента и потомков элемента. А что, если вам нужно выполнить запрос LINQ на последовательности элементов, такой как все потомки или все предки элементов, которые, возможно, являются потомками одного элемента? Для этого вам понадобятся дополнительный набор операций XML. В следующей главе я опишу новые операции LINQ to XML, добавленные специально для этих целей.

ГЛАВА 8

Операции LINQ to XML

К этому моменту я достаточно углубился в LINQ to XML, и вы, наверно, задаете вопрос: "Когда же мы перейдем к запросам?". Если это так, я отвечу: "Раскройте глаза, ребята! Вы уже видели их". На протяжении предыдущей главы я постоянно выполнял запросы LINQ to XML — возвращали они просто все дочерние элементы одного элемента или получали всех его предков. Помните метод `XContainer.Elements?` Можете вспомнить любой пример, где я вызывал его? Если да, значит, вы уже видели запрос LINQ to XML. Из-за очевидной гладкости интеграции запросов LINQ в язык, иногда можно и не заметить, что вы выполняете запрос.

Поскольку многие из методов классов, о которых я рассказал до сих пор, возвращают последовательность объектов классов XML, т.е. `IEnumerable<T>`, где `T` — один из классов программного интерфейса LINQ to XML, вы можете вызывать стандартные операции запросов на возвращенных последовательностях, обеспечивая им еще большую мощь и гибкость.

Так что есть способы получить последовательность XML-объектов из единственного объекта XML, таких как предки или потомки любого данного элемента. Но чего не хватает — так это способов выполнения операций LINQ to XML на каждом объекте этих последовательностей. Например, не существует простого пути получения последовательности элементов и выполнения другой XML-специфичной операции на каждом элементе последовательности, такой как возврат дочерних элементов для каждого элемента последовательности. Другими словами, вы можете получить последовательность дочерних элементов, вызвав метод `Elements` элемента, но не можете получить последовательность дочерних элементов дочернего элемента. Это потому, что метод `Elements` должен быть вызван на контейнере `XContainer`, таком как `XElement` или `XDocument`, но его нельзя вызвать на последовательности объектов `XContainer`. И здесь на помощь приходят операции LINQ to XML.

Введение в операции LINQ to XML

Интерфейс LINQ to XML API расширяет стандартные операции запросов LINQ to Objects специфичными для XML операциями. Эти XML-операции являются расширяющими методами, которые определены в классе `System.Xml.Linq.Extensions`, представляющим собой ни что иное, как класс-контейнер расширяющих методов.

Каждая из этих операций XML вызывается на последовательности некоторого типа данных LINQ to XML, и выполняет некоторые действия на каждом вхождении этой последовательности, такие как возврат всех предков или потомков данного вхождения.

Почти любая операция XML в этой главе имеет эквивалентный метод, описанный в предыдущей главе. Отличие состоит в том, что метод, описанный в предыдущей главе, вызывается на единственном объекте, а операция из настоящей главы вызывает

на последовательности объектов. Например, в предыдущей главе я рассказал о методе `XContainer.Elements`. Его прототип выглядит следующим образом:

```
IEnumerable< XElement > XContainer.Elements()
```

В этой главе я расскажу об операции `Extensions.Elements`, чей прототип выглядит так:

```
IEnumerable< XElement > Elements< T > (this IEnumerable< T > source) where T : XContainer
```

Между этими двумя методами есть существенная разница. Первый прототип вызывается на единственном объекте, унаследованном от `XContainer`, в то время как второй прототип вызывается на последовательности объектов, каждый из которых должен наследоваться от `XContainer`. Не забывайте о разнице.

В этой главе, чтобы отличить методы, описанные в предыдущей главе от расширяющих методов, описанных здесь, я обычно буду называть расширяющие методы *операциями*.

Итак, рассмотрим каждую из этих операций.

Ancestors

Операция `Ancestors` может быть вызвана на последовательности узлов и возвращать последовательность, содержащую элементы-предки каждого исходного узла.

Прототипы

Операция `Ancestors` имеет два прототипа.

Первый прототип `Ancestors`

```
public static IEnumerable< XElement > Ancestors< T > (
    this IEnumerable< T > source
) where T : XNode
```

Эта версия операции может быть вызвана на последовательности узлов или объектов, унаследованных от `XNode`. Она возвращает последовательность элементов, содержащую элементы-предки каждого узла исходной последовательности `source`.

Второй прототип `Ancestors`

```
public static IEnumerable< XElement > Ancestors< T > (
    this IEnumerable< T > source,
    XName name
) where T : XNode
```

Эта версия подобна первой, за исключением того, что ей передается имя, и только элементы-предки, соответствующие указанному имени, возвращаются в выходной последовательности.

Примеры

В листинге 8.1 представлен пример вызова первого прототипа `Ancestors`.

Листинг 8.1. Пример вызова первого прототипа `Ancestors`

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
```

```

    new XElement("LastName", "Rattz"),
    new XElement("BookParticipant",
        new XAttribute("type", "Editor"),
        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим элементы-предки для каждого исходного элемента.
foreach ( XElement element in elements.Ancestors())
{
    Console.WriteLine("Ancestor element: {0}", element.Name);
}

```

В приведенном примере я сначала создаю документ XML. Затем генерирую последовательность элементов FirstName. Напомню, что этот метод Ancestors вызывается на последовательности узлов, а не отдельном узле, поэтому мне нужна последовательность на которой можно будет вызвать его. Поскольку я хочу иметь возможность отобразить имена узлов в целях идентификации, в действительности я строю последовательность элементов, потому что элементы имеют имена, а узлы — нет. Затем я выполняю перечисление по последовательности, отображая исходные элементы — просто чтобы вы могли увидеть исходную последовательность. Затем я перечисляю элементы, возвращенные методом Ancestors, и отображаю их. Результат выглядит так:

```

Source element: FirstName : value = Joe
Source element: FirstName : value = Ewan
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: BookParticipant
Ancestor element: BookParticipants

```

Как видите, он отображает два элемента исходной последовательности — два первых элемента FirstName. Затем он отображает предков каждого из этих двух элементов.

Таким образом, используя операцию Ancestors, я могу извлекать все элементы предков для каждого узла в последовательности. В данном случае моя последовательность — это последовательность элементов, но это нормально, потому что элемент идет следующий от узла. Напомню, что не нужно путать операцию Ancestors, вызванную последовательности узлов, который я продемонстрировал здесь, с методом Ancestors, описанным в предыдущей главе.

В этом виде пример не столь впечатляющ, как мог бы быть, поскольку я расширил код в демонстрационных целях. Например, я хотел захватить последовательность элементов FirstName, поскольку хотел отобразить их наглядно в выводе. То есть оператор содержит вызов метода Descendants, и последующий блок foreach предназначен для этой цели. Затем во втором цикле foreach язываю операцию Ancestors и отображаю каждый элемент-предок. В действительности, во втором цикле foreach я мог бы вызвать на каждом элементе последовательности элементов FirstName метод Ancestors из предыдущей главы, не прибегая к методу Ancestors, который здесь я демонстрирую. В листинге 8.2 представлен пример, демонстрирующий то, что я мог бы сделать, чтобы получить тот же результат, не используя операцию Ancestors.

Листинг 8.2. Получение того же результата, что и в листинге 8.1, но без вызова операции Ancestors

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала отображу исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
foreach ( XElement element in elements)
{
    // Вызвать метод Ancestors на каждом элементе.
    foreach ( XElement e in element.Ancestors())
        // Теперь я отображу элементы-предки для каждого исходного элемента.
        Console.WriteLine("Ancestor element: {0}", e.Name);
}
```

Разница между этим примером и предыдущим состоит в том, что вместо вызова операции Ancestors на последовательности elements в цикле foreach я просто прохожу циклом по каждому элементу в последовательности и вызываю метод Ancestors на нем. Этот код выдает точно такой же вывод:

```
Source element: FirstName : value = Joe
Source element: FirstName : value = Ewan
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: BookParticipant
Ancestor element: BookParticipants
```

Однако, благодаря операции Ancestor и лаконичности LINQ, этот запрос можно комбинировать в единственный более краткий оператор, как показано в листинге 8.3.

Листинг 8.3. Более краткий пример вызова первого прототипа Ancestors

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
foreach ( XElement element in
    xDocument.Element("BookParticipants").Descendants("FirstName").Ancestors())
{
    Console.WriteLine("Ancestor element: {0}", element.Name);
}
```

В этом примере я перехожу прямо к делу и вызываю операцию `Ancestors` на по следовательности элементов, возвращенных методом `Descendants`. Поэтому метод `Descendants` возвращает последовательность элементов, а операция `Ancestors` вернет последовательность элементов, содержащих всех предков каждого элемента в последовательности, на которой она вызвана.

Поскольку этот код призван быть более кратким, он не отображает элементов `FirstName`, как это делали предыдущие два примера. Однако элементы-предки должны быть теми же самыми. Давайте в этом убедимся:

```
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: BookParticipant
Ancestor element: BookParticipants
```

Так и есть! В коде реального приложения вы, вероятно, выберете более компактный запрос, вроде только что продемонстрированного. Однако в этой главе для наглядности примеры будут более многословными, как в листинге 8.1.

Чтобы показать второй прототип `Ancestors`, я использую тот же базовый код, что и в листинге 8.1, но изменяю вызов операции `Ancestors` таким образом, что она включит параметр `BookParticipant`, чтобы я получил только элементы с соответствующим именем. Этот код представлен в листинге 8.4.

Листинг 8.4. Вызов второго прототипа `Ancestors`

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала я покажу исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь я покажу родительские элементы каждого исходного элемента.
foreach ( XElement element in elements.Ancestors("BookParticipant"))
{
    Console.WriteLine("Ancestor element: {0}", element.Name);
}
```

Теперь результаты должны только включать элементы `BookParticipant` и, конечно, исходные элементы, но два элемента `BookParticipants`, которые были показаны в примере с первым прототипом, теперь должны исчезнуть:

```
Source element: FirstName : value = Joe
Source element: FirstName : value = Ewan
Ancestor element: BookParticipant
Ancestor element: BookParticipant
```

Так и произошло.

AncestorsAndSelf

Операция `AncestorsAndSelf` может быть вызвана на последовательности элементов и возвращает последовательность, которая включает элементы-предков каждого исходного элемента, а также его самого. Эта операция подобна операции `Ancestors` за исключением того факта, что он может быть вызван только на элементах, а не на узлах, и также включает каждый исходный элемент в возвращенную последовательность элементов-предков.

Прототипы

Операция `AncestorsAndSelf` имеет два прототипа.

Первый прототип `AncestorsAndSelf`

```
public static IEnumerable< XElement> AncestorsAndSelf (
    this IEnumerable< XElement> source
)
```

Эта версия операции может быть вызвана на последовательности элементов и возвращает последовательность элементов, включающих каждый исходный элемент и всех его предков.

Второй прототип `AncestorsAndSelf`

```
public static IEnumerable< XElement> AncestorsAndSelf (
    this IEnumerable< XElement> source,
    XName name
)
```

Эта версия подобна первой, за исключением дополнительного параметра — имени, и возвращает в выходной последовательности только те элементы и его предков, которые соответствуют указанному имени.

Примеры

Для демонстрации первого прототипа `AncestorsAndSelf` я использую тот же базовый пример, что использовал и для первого прототипа `Ancestors`, только на этот раз вызову операцию `AncestorsAndSelf` вместо `Ancestors`, как показано в листинге 8.5.

Листинг 8.5. Пример вызова первого прототипа `AncestorsAndSelf`

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
```

```
// Теперь отобразим элементы-предки для каждого исходного элемента.
foreach ( XElement element in elements.AncestorsAndSelf())
{
    Console.WriteLine("Ancestor element: {0}", element.Name);
}
```

Как и с первым прототипом Ancestors, сначала я создаю документ XML. Затем генерирую последовательность элементов FirstName. Напомню, что этот метод AncestorsAndSelf вызывается на последовательности элементов, а не на отдельном элементе, так что мне понадобилась последовательность, на которой его вызвать. Затем я перечисляю элементы этой последовательности, отображая исходные элементы. А после этого я перечисляю элементы, возвращенные методом AncestorsAndSelf, и отображаю их.

Если все работает, как ожидалось, то результат должен быть таким же, как и в примере прототипа Ancestors, за исключением того, что элементы FirstName также будут включены в вывод. И вот результат:

```
Source element: FirstName : value = Joe
Source element: FirstName : value = Ewan
Source element: FirstName
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: FirstName
Ancestor element: BookParticipant
Ancestor element: BookParticipants
Ancestor element: BookParticipants
```

Для демонстрации второго прототипа AncestorsAndSelf я использую тот же базовый пример, что и в демонстрации второго прототипа Ancestors, конечно, с заменой вызова Ancestors на AncestorsAndSelf, как показано в листинге 8.6.

Листинг 8.6. Вызов второго прототипа AncestorsAndSelf

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Descendants("FirstName");
// Сначала я покажу исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь покажу родительские элементы каждого исходного элемента.
foreach ( XElement element in elements.AncestorsAndSelf("BookParticipant"))
{
    Console.WriteLine("Ancestor element: {0}", element.Name);
}
```

Теперь я должен получить только элементы по имени BookParticipant. Вот результат:

```
Source element: FirstName : value = Joe
Source element: FirstName : value = Ewan
Ancestor element: BookParticipant
Ancestor element: BookParticipant
```

Обратите внимание, что отображенный вывод метода `AncestorsAndSelf` теперь содержит только элементы `BookParticipant`, потому что только они соответствуют переданному имени. В этом случае я даже не получил исходные элементы, поскольку их имена не соответствуют. Так что функция работает, как определено.

Назовите меня сумасшедшим, но этот прототип операции кажется мне довольно бесполезным. Сколько уровней одноименных элементов вы собираетесь иметь в дереве XML? Если вы не ответите "минимум два", то как этот метод сможет когда-либо вернуть сами исходные элементы вместе с предками? Мне это кажется маловероятным. Да, я знаю: мне тоже нравятся симметричные API.

Attributes

Операция `Attributes` может быть вызвана на последовательности элементов и возвращает последовательность, содержащую атрибуты каждого исходного элемента.

Прототипы

Операция `Attributes` имеет два прототипа.

Первый прототип `Attributes`

```
public static IEnumerable<XAttribute> Attributes (
    this IEnumerable< XElement > source
)
```

Эта версия операции может быть вызвана на последовательности элементов и возвращает последовательность атрибутов, содержащую все атрибуты каждого из исходных элементов.

Второй прототип `Attributes`

```
public static IEnumerable<XAttribute> Attributes (
    this IEnumerable< XElement > source,
    XName name
)
```

Эта версия операции подобна первой, за исключением того, что в выходной последовательности возвращаются только атрибуты, соответствующие указанному имени.

Примеры

Для демонстрации первого прототипа `Attributes` я построю то же дерево XML, что и в предыдущих примерах. Однако последовательность исходных элементов, которую я сгенерирую, будет немного отличаться, потому что мне нужна последовательность элементов с атрибутами. Поэтому я сгенерирую последовательность элементов `BookParticipant` и обработаю ее, как показано в листинге 8.7.

Листинг 8.7. Вызов первого прототипа `Attributes`

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
```

```

        new XElement("LastName", "Rattz"),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим атрибуты каждого элемента.
foreach ( XAttribute attribute in elements.Attributes())
{
    Console.WriteLine("Attribute: {0} : value = {1}",
        attribute.Name, attribute.Value);
}

```

Получив последовательность элементов BookParticipant, я отображаю исходную последовательность. Затем вызываю операцию Attributes на исходной последовательности и отображаю атрибуты в последовательности, возвращенной операцией Attributes. И вот результат:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Attribute: type : value = Author
Attribute: type : value = Editor

```

Как видите, атрибуты извлечены. Для демонстрации второго прототипа Attributes я применяю тот же базовый пример, но на этот раз специфицирую имя, которому должны соответствовать атрибуты, возвращенные операцией Attributes, как показано в листинге 8.8.

Листинг 8.8. Вызов первого прототипа Attributes

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим атрибуты каждого элемента.
foreach ( XAttribute attribute in elements.Attributes("type"))
{
    Console.WriteLine("Attribute: {0} : value = {1}",
        attribute.Name, attribute.Value);
}

```

В приведенном коде я указал, что атрибуты должны соответствовать имени type. Поэтому этот пример должен выдать тот же вывод, что и предыдущий. По нажатию <Ctrl+F5> увидим следующее:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Attribute: type : value = Author
Attribute: type : value = Editor
```

Результат ожидаемый. Если бы я указал имя Type с заглавной первой буквой, то эти два атрибута не были бы отображены, потому что операция Attributes не вернула бы их из входной последовательности. Это демонстрирует случай, когда соответствия имени нет, а также тот факт, что сопоставление имени чувствительно к регистру, что не удивительно, учитывая зависимость от регистра XML.

DescendantNodes

Операция DescendantNodes может быть вызвана на последовательности элементов и возвращает последовательность, содержащую узлы-наследники каждого элемента в документе.

Прототипы

Операция DescendantNodes имеет один прототип.

Единственный прототип DescendantNodes

```
public static IEnumerable<XNode> DescendantNodes<T> (
    this IEnumerable<T> source
) where T : XContainer
```

Эта версия может быть вызвана на последовательности элементов или документов и возвращает последовательность, содержащую каждый из узлов-потомков исходного элемента или документа.

Отличие от метода XContainer.DescendantNodes заключается в том, что этот метод вызывается на последовательности элементов или документов, а не на одном элементе или документе.

Примеры

Для примера я построю то же дерево XML, что и в предыдущем примере, за исключением того, что также добавлю комментарий к первому элементу BookParticipant. Это нужно для того, чтобы как минимум, один возвращенный узел не был элементом. При построении моей исходной последовательности элементов я хочу, чтобы некоторые из них имели потомков, поэтому построю ее из элементов BookParticipant, поскольку у них есть некоторые потомки, как показано в листинге 8.9.

Листинг 8.9. Вызов единственного прототипа DescendantNodes

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
```

```

        new XElement("FirstName", "Ewan"),
        new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
    element.Name, element.Value);
}
// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach ( XNode node in elements.D DescendantNodes())
{
    Console.WriteLine("Descendant node: {0}", node);
}

```

Как это принято в примерах настоящего раздела, я строю мое дерево XML и исходную последовательность из элементов. В данном случае исходная последовательность содержит элементы BookParticipant. Затем язываю операцию DescendantNodes на исходной последовательности и отображаю результаты:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant node: <!--This is a new author.-->
Descendant node: < FirstName>Joe</ FirstName>
Descendant node: Joe
Descendant node: < LastName>Rattz</ LastName>
Descendant node: Rattz
Descendant node: < FirstName>Ewan</ FirstName>
Descendant node: Ewan
Descendant node: < LastName>Buckingham</ LastName>
Descendant node: Buckingham

```

Обратите внимание, что в результате я получаю не только элементы-потомки, но также и мой комментарий. Также обратите внимание, что для каждого элемента в документе XML я получаю по два узла. Например, есть узел, чье значение выглядит как "< FirstName>Joe</ FirstName>", а также узел со значением "Joe". Первый узел — элемент FirstName. Второй узел — XText для этого элемента. Наверное, вы заметили об этих автоматически создаваемых объектах XText. Тем не менее, они есть.

DescendantNodesAndSelf

Операция DescendantNodesAndSelf может быть вызвана на последовательности элементов и возвращает последовательность, содержащую сам каждый исходный элемент и его узлы-потомки.

Прототипы

Операция DescendantNodesAndSelf имеет один прототип.

Единственный прототип DescendantNodesAndSelf

```

public static IEnumerable< XNode> DescendantNodesAndSelf (
    this IEnumerable< XElement> source
)

```

Эта версия вызывается на последовательности элементов и возвращает последовательность узлов, содержащую каждый сам исходный элемент и все его узлы-потомки.

Примеры

Для демонстрации этой операции я применяю тот же пример, что использовался для демонстрации операции DescendantNodes, просто заменив операцию на DescendantNodesAndSelf, как показано в листинге 8.10.

Листинг 8.10. Вызов единственного прототипа DescendantNodes

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach ( XNode node in elements.DescendantNodesAndSelf())
{
    Console.WriteLine("Descendant node: {0}", node);
}

```

Вопрос в том, будет ли вывод таким же, как и у примера DescendantNodes, только с включением исходных элементов? Можете убедиться в этом:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant node: <BookParticipant type="Author">
    <!--This is a new author.-->
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
</BookParticipant>
Descendant node: <!--This is a new author.-->
Descendant node: <FirstName>Joe</FirstName>
Descendant node: Joe
Descendant node: <LastName>Rattz</LastName>
Descendant node: Rattz
Descendant node: <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
</BookParticipant>
Descendant node: <FirstName>Ewan</FirstName>
Descendant node: Ewan
Descendant node: <LastName>Buckingham</LastName>
Descendant node: Buckingham

```

Я не только получил сами элементы BookParticipant и их потомки, но также получил один узел, не являющийся элементом — комментарий. В этом состоит отличие между операциями DescendantNodesAndSelf и описанной ниже DescendantNodes, которая пропускает узлы, не являющиеся элементами.

Descendants

Операция Descendants может быть вызвана на последовательности элементов или документов и возвращает последовательность элементов, содержащую все потомки каждого исходного элемента или документа.

Прототипы

Операция Descendants имеет два прототипа.

Первый прототип Descendants

```
public static IEnumerable< XElement> Descendants< T> (
    this IEnumerable< T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность документов, содержащую потомков каждого исходного элемента или документа.

Отличие этой операции от метода XContainer.Descendants в том, что этот метод вызывается на последовательности элементов или документов, а не на отдельном элементе или документе.

Второй прототип Descendants

```
public static IEnumerable< XElement> Descendants< T> (
    this IEnumerable< T> source,
    XName name
) where T : XContainer
```

Эта версия во всем подобна первой, за исключением того, что в выходной последовательности возвращаются только те элементы, которые соответствуют указанному имени.

Примеры

Для демонстрации первого прототипа я использую в основном тот же пример, что и для операции DescendantNodes, вызывая вместо него операцию Descendants. Вызов должен быть таким же, только без узлов, не являющихся элементами. То есть вы увидите комментариев в выводе. Код представлен в листинге 8.11.

Листинг 8.11. Вызов первого прототипа Descendants

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
```

```

// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach ( XElement element in elements.Descendants())
{
    Console.WriteLine("Descendant element: {0}", element);
}

```

Этот пример такой же, как предыдущий, но с тем отличием, что здесь мы увидим только элементы-потомки двух элементов BookParticipant. Результат работы этого примера выглядит так:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant element: <FirstName>Joe</FirstName>
Descendant element: <LastName>Rattz</LastName>
Descendant element: <FirstName>Ewan</FirstName>
Descendant element: <LastName>Buckingham</LastName>

```

Сравнивая эти результаты с примером операции DescendantNodes, я замечаю некоторые отличия, которых не ожидал. Конечно, потомки помечены как элементы, а не узлы, и комментарий отсутствует, но при этом отсутствуют также узлы-потомки, такие как Joe и Rattz. Ну конечно, эти узлы также не являются элементами. Это объекты XText. Программный интерфейс LINQ to XML API обрабатывает текстовые узлы настолько гладко, что о них легко забыть.

Для демонстрации второго прототипа я использую тот же код, что и в первом примере, за исключением дополнительного указания имени, которому должны соответствовать элементы-потомки, чтобы второй прототип операции Descendant вернул их, как показано в листинге 8.12.

Листинг 8.12. Вызов второго прототипа Descendants

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим узлы-потомки каждого исходного элемента.
foreach ( XNode node in elements.Descendants("LastName"))
{
    Console.WriteLine("Descendant node: {0}", node);
}

```

Результат этого примера выглядит следующим образом:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant element: <LastName>Rattz</LastName>
Descendant element: <LastName>Buckingham</LastName>
```

Как можно было ожидать, возвращены только элементы LastName.

DescendantsAndSelf

Операция DescendantsAndSelf может быть вызвана на последовательности элементов и возвращает последовательность, содержащую каждый исходный элемент и его потомков.

Прототипы

Операция DescendantsAndSelf имеет два прототипа.

Первый прототип DescendantsAndSelf

```
public static IEnumerable<XElement> DescendantsAndSelf (
    this IEnumerable<XElement> source
)
```

Эта версия вызывается на последовательности элементов и возвращает последовательность элементов, содержащую каждый исходный элемент и его потомков.

Второй прототип DescendantsAndSelf

```
public static IEnumerable<XElement> DescendantsAndSelf (
    this IEnumerable<XElement> source,
    XName name
)
```

Эта версия подобна первой, но принимает дополнительный параметр, указывающий имя, которому должны соответствовать возвращаемые элементы.

Примеры

Для демонстрации я использую тот же код, что и в примере применения первого прототипа операции Descendants, заменив его на операцию DescendantsAndSelf, как показано в листинге 8.13.

Листинг 8.13. Вызов первого прототипа DescendantsAndSelf

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable<XElement> elements =
xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
```

```

foreach (XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.DescendantsAndSelf())
{
    Console.WriteLine("Descendant element: {0}", element);
}

```

Запустив этот пример, вы должны увидеть все исходные элементы и их элементы-потомки. Результат запуска этого примера показан ниже.

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant element: <BookParticipant type="Author">
    <!--This is a new author.-->
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
</BookParticipant>
Descendant element: <FirstName>Joe</FirstName>
Descendant element: <LastName>Rattz</LastName>
Descendant element: <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
</BookParticipant>
Descendant element: <FirstName>Ewan</FirstName>
Descendant element: <LastName>Buckingham</LastName>

```

Таким образом, вывод тот же самый, что и у операции Descendants, за исключением того, что он также включает сами исходные элементы — элементы BookParticipant. Пусть вас не введет в заблуждение наличие комментария в результатах. Это не потому, что он был возвращен операцией DescendantsAndSelf, а потому, что я отображаю элемент BookParticipant, который был возвращен операцией.

Для демонстрации второго прототипа DescendantsAndSelf я использую тот же пример, что и для первого прототипа, только с добавлением имени, которому должны соответствовать возвращаемые элементы, как показано в листинге 8.14.

Листинг 8.14. Вызов второго прототипа DescendantsAndSelf

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach (XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}

```

```
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach (XElement element in elements.DescendantsAndSelf("LastName"))
{
    Console.WriteLine("Descendant element: {0}", element);
}
```

Результат этого примера выглядит так:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Descendant element: <LastName>Rattz</LastName>
Descendant element: <LastName>Buckingham</LastName>
```

Результат включает только элементы-потомки, соответствующие указанному имени. Здесь не слишком очевидно, что я вызвал операцию DescendantsAndSelf, а не Descendants, поскольку исходные элементы не возвращены, так как их имя не соответствует указанному.

Опять-таки, как и со всеми операциями, которые возвращают элементы из множества уровней дерева XML, принимающих аргумент — имя, которому должны соответствовать возвращаемые элементы, вряд ли вам понадобятся версии AndSelf операций, потому что маловероятно, что у вас будет много уровней одноименных операций.

Elements

Операция Elements может быть вызвана на последовательности элементов или документов и возвращает последовательность элементов, содержащих дочерние элементы каждого исходного элемента или документа.

Эта операция отличается от операции Descendants, поскольку операция Elements возвращает только непосредственные дочерние элементы каждого из элементов входной последовательности, в то время как операция Descendants рекурсивно возвращает все дочерние элементы до достижения конца каждого дерева.

Прототипы

Операция Elements имеет два прототипа.

Первый прототип Elements

```
public static IEnumerable< XElement> Elements< T> (
    this IEnumerable< T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность элементов, содержащую дочерние элементы каждого исходного элемента.

Операция отличается от метода XContainer.Elements в том, что этот метод вызывается на последовательности элементов или документов, а не на единственном элементе или документе.

Второй прототип Elements

```
public static IEnumerable< XElement> Elements< T> (
    this IEnumerable< T> source,
    XName name
) where T : XContainer
```

Эта версия подобна первой, за исключением дополнительного параметра, заставляющего операцию извлекать в выходную последовательность только те элементы, имя которых соответствует его значению.

Примеры

Дальнейшее вы можете предположить. Для демонстрации первого прототипа я использую тот же базовый пример, что и для операции DescendantsAndSelf, за исключением того, что вместо нее я вызову операцию Elements, как показано в листинге 8.15.

Листинг 8.15. Вызов первого прототипа Elements

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach ( XElement element in elements.Elements())
{
    Console.WriteLine("Descendant element: {0}", element);
}
```

Как и в предыдущих примерах, я строю дерево XML, получая последовательность исходных элементов, отображаю каждый элемент, извлекаю последовательность дочерних элементов каждого исходного элемента и отображаю дочерние элементы:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Child element: <FirstName>Joe</FirstName>
Child element: <LastName>Rattz</LastName>
Child element: <FirstName>Ewan</FirstName>
Child element: <LastName>Buckingham</LastName>
```

Этот пример возвращает все дочерние элементы. Чтобы извлечь лишь элементы, соответствующие указанному имени, я использую второй прототип операции Elements, как показано в листинге 8.16.

Листинг 8.16. Вызов второго прототипа Elements

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"))),
```

```

new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала отобразим исходные элементы.
foreach ( XElement element in elements)
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь отобразим элементы-потомки каждого исходного элемента.
foreach ( XElement element in elements.Elements("LastName"))
{
    Console.WriteLine("Descendant element: {0}", element);
}

```

Теперь я должен получить только те элементы, что соответствуют имени LastName:

```

Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
Child element: <LastName>Rattz</LastName>
Child element: <LastName>Buckingham</LastName>

```

Все работает, как и следовало ожидать.

InDocumentOrder

Операция InDocumentOrder может быть вызвана на последовательности узлов и возвращает последовательность, дочерние узлы каждого исходного узла в порядке документа.

Прототипы

Операция InDocumentOrder имеет один прототип.

Единственный прототип InDocumentOrder

```

public static IEnumerable< T> InDocumentOrder< T> (
    this IEnumerable< T> source
) where T : XNode

```

Эта версия вызывается на последовательности указанного типа, элементы которой должны быть узлами или типами-наследниками узлов, и возвращает последовательность некоторого типа, содержащую дочерние узлы исходных узлов в порядке документа.

Примеры

Эта довольно странная операция. Для данного примера мне понадобится последовательность узлов. Поскольку мне нужно видеть в дополнение к элементам некоторые узлы, которые не являются элементами, я построю последовательность узлов дочерних по отношению к элементам BookParticipant. Это нужно потому, что один из них имеет комментарий, который является узлом, но не элементом. Исходный код представлен в листинге 8.17.

Листинг 8.17. Вызов единственного прототипа InDocumentOrder

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant").
    Nodes().Reverse();
// Сначала я отображу все исходные узлы.
foreach (XNode node in nodes)
{
    Console.WriteLine("Source node: {0}", node);
}
// Теперь я отображу дочерние узлы каждого исходного узла.
foreach (XNode node in nodes.InDocumentOrder())
{
    Console.WriteLine("Ordered node: {0}", node);
}

```

Как видно в этом коде, я строю дерево XML. Когда я извлекаю мою исходную последовательность, то получаю дочерние узлы элемента BookParticipant, вызывая операцию `Nodes`, а затем вызывая стандартную операцию запроса `Reverse`. Если вы вспомните из части 2 настоящей книги, посвященной LINQ to Objects, операция `Reverse` возвращает последовательность входных элементов, выстроенную в обратном порядке. Теперь у меня есть последовательность узлов, расположенных не в исходном порядке. Я предпринимаю этот дополнительный шаг по изменению порядка, чтобы при вызове операции `InDocumentOrder` было обнаружено отличие. Затем я отображаю исходные узлы в неупорядоченном виде, вызываю операцию `InDocumentOrder` и показываю результат, который выглядит так:

```

Source node: <LastName>Buckingham</LastName>
Source node: <FirstName>Ewan</FirstName>
Source node: <LastName>Rattz</LastName>
Source node: <FirstName>Joe</FirstName>
Source node: <!--This is a new author.-->
Ordered node: <!--This is a new author.-->
Ordered node: <FirstName>Joe</FirstName>
Ordered node: <LastName>Rattz</LastName>
Ordered node: <FirstName>Ewan</FirstName>
Ordered node: <LastName>Buckingham</LastName>

```

Как вы можете видеть, исходные узлы расположены в порядке, противоположном тому, в котором я построил их, а упорядоченные узлы оказываются опять в своем исходном порядке. Отлично, хотя и странно.

Nodes

Операция `Nodes` может быть вызвана на последовательности элементов или документов и возвращает последовательность узлов, содержащих дочерние узлы каждого исходного элемента или документа.

Эта операция отличается от операции DescendantNodes тем, что Nodes возвращает только непосредственных потомков — дочерние элементы каждого элемента входной последовательности, в то время как операция DescendantNodes рекурсивно возвращает все дочерние узлы, пока не достигнет конца дерева.

Прототипы

Операция Nodes имеет один прототип.

Единственный прототип Nodes

```
public static IEnumerable<XNode> Nodes<T> (
    this IEnumerable<T> source
) where T : XContainer
```

Эта версия вызывается на последовательности элементов или документов и возвращает последовательность узлов, содержащих дочерние узлы всех исходных элементов.

В отличие от метода XContainer.Nodes этот метод вызывается на последовательности элементов или документов, а не на отдельном элементе или документе.

Примеры

Для этого примера я построю типичное дерево XML и исходную последовательность элементов BookParticipant. Затем отобразу каждый из них, после чего верну дочерние узлы каждого исходного элемента и отобразу их, как показано в листинге 8.18.

Листинг 8.18. Вызов единственного прототипа Nodes

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement > elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
// Сначала я отобразу все исходные узлы.
foreach ( XElement element in elements )
{
    Console.WriteLine("Source element: {0} : value = {1}",
        element.Name, element.Value);
}
// Теперь я отобразу дочерние узлы каждого исходного элемента.
foreach ( XNode node in elements.Nodes() )
{
    Console.WriteLine("Child node: {0}", node);
}
```

Поскольку эта операция возвращает дочерние узлы, а не элементы, выходная последовательность будет включать в результате комментарий из первого элемента BookParticipant:

```
Source element: BookParticipant : value = JoeRattz
Source element: BookParticipant : value = EwanBuckingham
```

```

Child node: <!--This is a new author.-->
Child node: <FirstName>Joe</FirstName>
Child node: <LastName>Rattz</LastName>
Child node: <FirstName>Ewan</FirstName>
Child node: <LastName>Buckingham</LastName>

```

Результат отображает все дочерние узлы каждого исходного элемента. Обратите внимание, что поскольку извлекаются только непосредственные дочерние узлы, я не получил узлы XText, являющиеся дочерними по отношению к каждому элементу FirstName и LastName, как это было в примере операции DescendantNodes.

Remove

Операция Remove может быть вызвана на последовательности узлов или атрибутов для их удаления. Этот метод кэширует копию узлов или атрибутов в List для того, чтобы избежать "проблемы Хэллоуина", описанной в предыдущей главе.

Прототипы

Операция Remove имеет два прототипа.

Первый прототип Remove

```

public static void Remove (
    this IEnumerable<XAttribute> source
)

```

Эта версия вызывается на последовательности атрибутов и удаляет все атрибуты в исходной последовательности.

Второй прототип Remove

```

public static void Remove<T> (
    this IEnumerable<T> source
) where T : XNode

```

Эта версия вызывается на последовательности указанного типа, элементы которой должны быть узлами или типом, производным от узлов, и удаляет все узлы входной последовательности.

Примеры

Поскольку первый прототип предназначен для удаления атрибутов, мне понадобилась последовательность атрибутов. Поэтому я построю стандартное дерево XML и извлеку последовательность атрибутов элементов BookParticipant. Я отобразу каждый исходный атрибут, после чего вызову операцию Remove на последовательности исходных атрибутов. Затем, просто для того, чтобы доказать, что он работает, я отобразу весь XML-документ с удаленными атрибутами, как показано в листинге 8.19.

Листинг 8.19. Вызов первого прототипа Remove

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),

```

```

    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> attributes =
    xDocument.Element("BookParticipants").Elements("BookParticipant").Attributes();
// Сначала я отобразу все исходные атрибуты.
foreach (XAttribute attribute in attributes)
{
    Console.WriteLine("Source attribute: {0} : value = {1}",
        attribute.Name, attribute.Value);
}
attributes.Remove();
// Теперь я отобразу XML-документ.
Console.WriteLine(xDocument);

```

Будет это работать? Давайте посмотрим:

```

Source attribute: type : value = Author
Source attribute: type : value = Editor
<BookParticipants>
    <BookParticipant>
        <!--This is a new author.-->
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant>
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>

```

Пока все хорошо. Теперь я попробую применить второй прототип. Для этого примера вместо получения последовательности узлов и удаления их я продемонстрирую нечто более интересное. Я получу последовательность комментариев определенных элементов и удалю только их, как показано в листинге 8.20.

Листинг 8.20. Вызов второго прототипа Remove

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> comments =
    xDocument.Element("BookParticipants").Elements("BookParticipant").
        Nodes().OfType< XComment>();
// Сначала я отобразу все исходные комментарии.
foreach (XComment comment in comments)
{
    Console.WriteLine("Source comment: {0}", comment);
}
comments.Remove();
// Теперь я отобразу XML-документ.
Console.WriteLine(xDocument);

```

В данном примере при построении моей исходной последовательности я извлекаю дочерние узлы каждого элемента BookParticipant. Я мог бы просто вызвать операцию Remove на этой последовательности, и тогда все дочерние узлы каждого элемента BookParticipant исчезли бы. Но вместо этого я вызываю стандартную операцию запроса OfType. Если вы помните из части 2 этой книги, упомянутая операция вернет только те объекты входной последовательности, которые соответствуют указанному типу. Вызывая операцию OfType и специфицируя тип XComment, я получаю последовательность, состоящую только из комментариев. Затем я вызываю на этих комментариях метод Remove. В результате исходный документ будет очищен от всех комментариев:

```
Source comment: <!--This is a new author.-->
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
```

Это работает весьма неплохо. Взгляните, насколько удобна операция OfType, и как она интегрируется в запрос LINQ to XML. Похоже, она может оказаться весьма полезной.

Резюме

В предыдущей главе я описал программный интерфейс LINQ to XML API, позволяющий создавать, модифицировать, сохранять и загружать деревья XML. Обратите внимание, что я сказал деревья, а не документы, потому что в LINQ to XML документы не обязательны. В этой главе я продемонстрировал, как запросить один узел или элемент на предмет узлов и элементов, иерархически связанных с ним. В этой главе я описал, каким образом то же самое делается с последовательностями узлов или элементов с применением операций LINQ to XML. Я надеюсь, что достаточно прояснил выполнение элементарных запросов по деревьям XML, используя LINQ to XML. Уверен, что этот новый программный интерфейс XML API докажет свою способность удобно запрашивать данные XML. В частности, возможность комбинирования стандартных операций запросов с операциями LINQ to XML позволяет строить довольно элегантные и мощные запросы.

К данному моменту я рассказал почти все, что нужно знать о построении блоков, необходимых для выполнения запросов LINQ to XML. В следующей главе я представлю несколько более сложные запросы и расскажу об остальных предметах первой необходимости XML, таких как проверка достоверности и трансформация.

ГЛАВА 9

Дополнительные возможности XML

В предыдущих двух главах я продемонстрировал, как создавать, модифицировать и проходить по данным XML с использованием программного интерфейса LINQ to XML API. Также я показал строительные блоки, из которых можно создавать мощные запросы XML. Надеюсь, теперь вы согласитесь, что LINQ to XML покроет 90% ваших потребностей, но как насчет оставшихся 10%? Посмотрим, можно ли еще повысить этот процент. Если Microsoft добавила возможности проверки достоверности, трансформации схем и запросов XPath, какой процент ваших случаев теперь можно покрыть?

Хотя я описал новый программный интерфейс LINQ to XML API, а также способы выполнения самых основных запросов с его помощью, все же мне еще осталось продемонстрировать несколько более сложные запросы, потребность в которых может возникнуть на практике. В этой главе я приведу ряд примеров, которые, как можно надеяться, сделают задачу опроса данных XML с LINQ to XML тривиальной, включая применение синтаксиса выражений запросов для тех из вас, кто его предпочитает.

Вдобавок описание нового программного интерфейса LINQ to XML API будет неполным без нескольких дополнительных возможностей, таких как трансформация и проверка достоверности. В этой главе я опишу эти дополнения к LINQ to XML, а также приведу и другую полезную информацию.

В частности, я расскажу, как выполнять трансформации с XSLT и без него, а также продемонстрирую, как сверить документ XML со схемой, и даже представлю пример выполнения запроса в стиле XPath.

Необходимые пространства имен

Примеры этой главы будут использовать пространства имен `System.Xml`, `System.Xml.Schema`, `System.Xml.Xsl` и `System.Xml.XPath`, в дополнение к обычным пространствам имен LINQ to XML — `System.Linq` и `System.Xml.Linq`. Поэтому вам понадобится добавить следующие директивы `using`, если вы не сделали этого раньше:

```
using System.Linq;
using System.Xml;
using System.Xml.Linq;
using System.Xml.Schema;
using System.Xml.XPath;
using System.Xml.Xsl;
```

Запросы

В предыдущих главах, посвященных LINQ to XML, я продемонстрировал основные принципы, необходимые для выполнения запросов XML с использованием LINQ to XML. Однако большая часть примеров была специально спроектирована для демонстрации операций или свойства. В данном разделе я хочу представить некоторые примеры, которые более ориентированы на решение какой-то проблемы.

Без спуска

В предыдущих главах, во многих примерах для получения ссылки на определенный элемент, выполнялся ступенчатый спуск вниз по иерархии, посредством рекурсивного вызова операций `Element` и `Elements` до тех пор, пока не будет достигнут нужный элемент.

Например, многие примеры содержали примерно такие строки кода:

```
IEnumerable< XElement > elements =
    xDocument.Element("BookParticipants").Elements("BookParticipant");
```

В этом операторе я начинаю с уровня документа, затем получаю его дочерний элемент по имени `BookParticipants`, после чего — его дочерний элемент по имени `BookParticipant`. Однако вовсе не обязательно спускаться по иерархии вниз таким вот способом. Вместо этого я мог бы просто написать код, подобный представленному в листинге 9.1.

Листинг 9.1. Получение элементов без иерархического спуска

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement > elements = xDocument.Descendants("BookParticipant");
foreach ( XElement element in elements)
{
    Console.WriteLine("Element: {0} : value = {1}",
        element.Name, element.Value);
}
```

В этом примере я получаю все элементы-потомки документа по имени `BookParticipant`. Поскольку я не обращаюсь к определенной ветви дерева XML, мне необходимо знать схему, поскольку я могу получить элементы не из той ветви, которая мне нужна. Однако во многих случаях, включая данный, этот подход работает хорошо.

Вот результат:

```
Element: BookParticipant : value = JoeRattz
Element: BookParticipant : value = EwanBuckingham
```

Однако может быть, мне не нужны все элементы `BookParticipant`; возможно, мне необходимо как-то ограничить возвращаемые элементы? В листинге 9.2 представлен пример, возвращающий только элементы, у которых значение дочернего элемента `FirstName` равно `"Ewan"`.

Листинг 9.2. Получение ограниченных элементов без иерархического спуска

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements = xDocument
    .Descendants("BookParticipant")
    .Where(e => ((string)e.Element("FirstName")) == "Ewan");
foreach ( XElement element in elements)
{
    Console.WriteLine("Element: {0} : value = {1}",
        element.Name, element.Value);
}

```

На этот раз я добавил вызов операции `Where`. Обратите внимание, что я привожу элемент `FirstName` к типу `string`, чтобы получить его значение для сравнения с `"Ewan"`. И вот результат:

```
Element: BookParticipant : value = EwanBuckingham
```

Конечно, иногда вам нужно контролировать порядок. На этот раз, чтобы былоозвращено более одного элемента, и порядок имел значение, я изменил лямбда-выражение операции `Where`, чтобы были возвращены оба элемента. Чтобы сделать пример интереснее, я выполню запрос по атрибуту `type`, и попробую переписать его в синтаксисе выражений запросов, как показано в листинге 9.3.

Листинг 9.3. Получение ограниченных элементов без иерархического спуска с упорядочиванием и использованием синтаксиса выражений запросов

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
IEnumerable< XElement> elements =
    from e in xDocument.Descendants("BookParticipant")
    where ((string)e.Attribute("type")) != "Illustrator"
    orderby ((string)e.Element("LastName"))
    select e;
foreach ( XElement element in elements)
{
    Console.WriteLine("Element: {0} : value = {1}",
        element.Name, element.Value);
}

```

В этом примере я по-прежнему запрашиваю элементы BookParticipant документа, но только те, чей атрибут отличается от Illustrator. В данном случае это будут все элементы BookParticipant.

Затем я упорядочиваю их по значению элемента LastName. Опять-таки, обратите внимание, что я выполняю приведение и атрибута type, и элемента LastName для получения их значений. Вот результат:

```
Element: BookParticipant : value = EwanBuckingham
Element: BookParticipant : value = JoeRattz
```

Сложный запрос

До сих пор все примеры запросов были тривиальными, поэтому прежде чем закрыть эту тему, я хотел бы привести пример более сложного запроса. Для этого примера я использую те же данные, которые рекомендованы W3C специально для тестирования предцедентов использования запроса XML.

Пример в листинге 9.4 содержит данные из трех разных документов XML. В моем коде примера я создаю каждый документ, разбирая текстовое представление каждого из рекомендованных W3C документов XML. Поскольку этот пример сложен, я буду вставлять объяснения по мере продвижения.

Первый шаг — создание документов из XML.

Листинг 9.4. Сложный запрос, объединяющий три документа, в синтаксисе выражений запросов

```
XDocument users = XDocument.Parse(
 @"<users>
  <user_tuple>
   <userid>U01</userid>
   <name>Tom Jones</name>
   <rating>B</rating>
  </user_tuple>
  <user_tuple>
   <userid>U02</userid>
   <name>Mary Doe</name>
   <rating>A</rating>
  </user_tuple>
  <user_tuple>
   <userid>U03</userid>
   <name>Dee Linquent</name>
   <rating>D</rating>
  </user_tuple>
  <user_tuple>
   <userid>U04</userid>
   <name>Roger Smith</name>
   <rating>C</rating>
  </user_tuple>
  <user_tuple>
   <userid>U05</userid>
   <name>Jack Sprat</name>
   <rating>B</rating>
  </user_tuple>
  <user_tuple>
   <userid>U06</userid>
   <name>Rip Van Winkle</name>
   <rating>B</rating>
  </user_tuple>
 </users>");
```

```
XDocument items = XDocument.Parse(
    @"<items>
        <item_tuple>
            <itemno>1001</itemno>
            <description>Red Bicycle</description>
            <offered_by>U01</offered_by>
            <start_date>1999-01-05</start_date>
            <end_date>1999-01-20</end_date>
            <reserve_price>40</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1002</itemno>
            <description>Motorcycle</description>
            <offered_by>U02</offered_by>
            <start_date>1999-02-11</start_date>
            <end_date>1999-03-15</end_date>
            <reserve_price>500</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1003</itemno>
            <description>Old Bicycle</description>
            <offered_by>U02</offered_by>
            <start_date>1999-01-10</start_date>
            <end_date>1999-02-20</end_date>
            <reserve_price>25</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1004</itemno>
            <description>Tricycle</description>
            <offered_by>U01</offered_by>
            <start_date>1999-02-25</start_date>
            <end_date>1999-03-08</end_date>
            <reserve_price>15</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1005</itemno>
            <description>Tennis Racket</description>
            <offered_by>U03</offered_by>
            <start_date>1999-03-19</start_date>
            <end_date>1999-04-30</end_date>
            <reserve_price>20</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1006</itemno>
            <description>Helicopter</description>
            <offered_by>U03</offered_by>
            <start_date>1999-05-05</start_date>
            <end_date>1999-05-25</end_date>
            <reserve_price>50000</reserve_price>
        </item_tuple>
        <item_tuple>
            <itemno>1007</itemno>
            <description>Racing Bicycle</description>
            <offered_by>U04</offered_by>
            <start_date>1999-01-20</start_date>
            <end_date>1999-02-20</end_date>
            <reserve_price>200</reserve_price>
        </item_tuple>
        <item_tuple>
```

```
<itemno>1008</itemno>
<description>Broken Bicycle</description>
<offered_by>U01</offered_by>
<start_date>1999-02-05</start_date>
<end_date>1999-03-06</end_date>
<reserve_price>25</reserve_price>
</item_tuple>
</items">;
XDocument bids = XDocument.Parse(
@"<bids>
<bid_tuple>
<userid>U02</userid>
<itemno>1001</itemno>
<bid>35</bid>
<bid_date>1999-01-07</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U04</userid>
<itemno>1001</itemno>
<bid>40</bid>
<bid_date>1999-01-08</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U02</userid>
<itemno>1001</itemno>
<bid>45</bid>
<bid_date>1999-01-11</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U04</userid>
<itemno>1001</itemno>
<bid>50</bid>
<bid_date>1999-01-13</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U02</userid>
<itemno>1001</itemno>
<bid>55</bid>
<bid_date>1999-01-15</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U01</userid>
<itemno>1002</itemno>
<bid>400</bid>
<bid_date>1999-02-14</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U02</userid>
<itemno>1002</itemno>
<bid>600</bid>
<bid_date>1999-02-16</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U03</userid>
<itemno>1002</itemno>
<bid>800</bid>
<bid_date>1999-02-17</bid_date>
</bid_tuple>
<bid_tuple>
```

```

<userid>U04</userid>
<itemno>1002</itemno>
<bid>1000</bid>
<bid_date>1999-02-25</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U02</userid>
<itemno>1002</itemno>
<bid>1200</bid>
<bid_date>1999-03-02</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U04</userid>
<itemno>1003</itemno>
<bid>15</bid>
<bid_date>1999-01-22</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U05</userid>
<itemno>1003</itemno>
<bid>20</bid>
<bid_date>1999-02-03</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U01</userid>
<itemno>1004</itemno>
<bid>40</bid>
<bid_date>1999-03-05</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U03</userid>
<itemno>1007</itemno>
<bid>175</bid>
<bid_date>1999-01-25</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U05</userid>
<itemno>1007</itemno>
<bid>200</bid>
<bid_date>1999-02-08</bid_date>
</bid_tuple>
<bid_tuple>
<userid>U04</userid>
<itemno>1007</itemno>
<bid>225</bid>
<bid_date>1999-02-12</bid_date>
</bid_tuple>
</bids>");
```

Этот пример данных в основном предназначен для представления на сайте Internet-аукциона. Я просто создал три документа XML, вызывая метод `XDocument.Parse` на строковом представлении данных XML. Здесь имеются документы для пользователей товаров и предложенных цен.

Я хочу, чтобы мой запрос производил список всех цен, превышающих \$50. В результатах я хочу видеть дату и цену, наряду с пользователем, предложившим ее, а также наименование товара и его описание.

Вот мой запрос:

```

var biddata = from b in bids.Descendants("bid_tuple")
    where ((double)b.Element("bid")) > 50
    join u in users.Descendants("user_tuple")
    on ((string)b.Element("userid")) equals
        ((string)u.Element("userid"))
    join i in items.Descendants("item_tuple")
    on ((string)b.Element("itemno")) equals
        ((string)i.Element("itemno"))
    select new { Item = ((string)b.Element("itemno")),
        Description = ((string)i.Element("description")),
        User = ((string)u.Element("name")),
        Date = ((string)b.Element("bid_date")),
        Price = ((double)b.Element("bid")) };

```

Как видите, запрос сложный. Первый шаг состоит в том, что я запрашиваю потомков по имени `bid_tuple` в документе `bids`, используя метод `Descendants`. Затем выполняю конструкцию `where` для элементов, имеющих дочерние элементы по имени `bid`, чье значение превышает 50. Таким образом, я извлекаю предложения цены, превышающие \$50. Это может показаться немного необычным, что я выполняю `where` в запросе так скоро. Я действительно должен был сделать это в запросе много ниже — непосредственно перед вызовом конструкции `select`. Однако это означало бы, что мне пришлось бы извлечь и выполнить соединение всех записей из XML-документов пользователей и товаров, для которых предложения цены не превышают \$50, что излишне. Фильтруя результатирующий набор как можно раньше, я сокращаю рабочую нагрузку на остальную часть запроса, тем самым повышая производительность.

Отфильтровав рабочий набор по предложениям цен, превышающим \$50, я выполняю соединение (`join`) этих предложений с XML-документом пользователей через одинаковый в двух наборах элемент `userid`, чтобы получить имя пользователя. В этот момент у меня есть предложения и связанные с ними пользователи, предложившие цену свыше \$50.

Затем я соединяю результат с XML-документом наименований товара через одинаковый элемент `itemno`, чтобы получить описание товара. Таким образом, у меня есть соединение предложенной цены, пользователей и товаров.

Отметьте опять, что мне приходится выполнять приведение всех элементов к интересующему меня типу данных, чтобы получить стоимость товара. Особенно интересен тот факт, что я получаю предложенную цену, приводя элемент `bid` к типу `double`. Даже несмотря на то, что входное значение предложенной цены является строкой, поскольку оно может быть успешно преобразовано в `double`, я имею возможность выполнить приведение к `double`, чтобы получить значение цены в числовом виде двойной точности. Здорово, не правда ли?

Следующий шаг — просто выбор анонимного класса, содержащего всю информацию из дочерних элементов соединенных элементов, которые меня интересуют.

Следующий шаг — отображение заголовка:

```

Console.WriteLine("{0,-12} {1,-12} {2,-6} {3,-14} {4,10}",
    "Date",
    "User",
    "Item",
    "Description",
    "Price");
Console.WriteLine("=====");

```

Здесь нет ничего особенного. Все, что осталось — выполнить перечисление результатирующей последовательности и отобразить каждое предложение цены:

```

foreach (var bd in biddata)
{
    Console.WriteLine("{0,-12} {1,-12} {2,-6} {3,-14} {4,10:C}",
        bd.Date,
        bd.User,
        bd.Item,
        bd.Description,
        bd.Price);
}

```

Эта часть тривиальна. В действительности все тривиально, кроме самого запроса. Готовы ли вы увидеть результат? Я — да:

Date	User	Item	Description	Price
1999-01-15	Mary Doe	1001	Red Bicycle	\$55.00
1999-02-14	Tom Jones	1002	Motorcycle	\$400.00
1999-02-16	Mary Doe	1002	Motorcycle	\$600.00
1999-02-17	Dee Linquent	1002	Motorcycle	\$800.00
1999-02-25	Roger Smith	1002	Motorcycle	\$1,000.00
1999-03-02	Mary Doe	1002	Motorcycle	\$1,200.00
1999-01-25	Dee Linquent	1007	Racing Bicycle	\$175.00
1999-02-08	Jack Sprat	1007	Racing Bicycle	\$200.00
1999-02-12	Roger Smith	1007	Racing Bicycle	\$225.00

Вы должны признать, что это впечатляет, разве нет? Я просто соединил три XML-документа в одном запросе.

Определенно, теперь вы увидели мощь LINQ to XML. Хотите узнать, почему LINQ to XML — моя любимая часть LINQ? И сколько вы готовы заплатить? Но минуточку, есть еще кое-что!

Трансформации

С помощью LINQ to XML вы можете выполнять трансформации XML, используя для этого два совершенно разных подхода. Первый подход заключается в применении XSLT через классы-мосты — `XmlReader` и `XmlWriter`. Второй подход предусматривает использование LINQ to XML для выполнения трансформаций посредством функционального конструирования целевого документа XML и встраивания запроса LINQ to XML в некоторый документ XML.

Применение XSLT обладает тем преимуществом, что это — стандартная технология XML. Существует инструментарий, который помогает в написании, отладке и тестировании трансформаций XSLT. Вдобавок, поскольку он уже существует, вы можете иметь готовые XSLT-документы и использовать их в новом коде с применением LINQ to XML. Существует целый мир доступных XSLT-документов, из которых вы можете выбрать нужные. К тому же, использование XSLT для ваших трансформаций наиболее الدينично. В отличие от применения подхода на основе функционального конструирования LINQ to XML, вам не нужно перекомпилировать код для изменения трансформации. Простое изменение документа XSLT позволяет вам модифицировать трансформации в любое время выполнения. И, наконец, XSLT — широко известная технология, есть множество знающих ее разработчиков, которые могут помочь вам. Что касается подхода на основе функционального конструирования, пока, на начальном этапе существования LINQ to XML, еще не так.

Применение подхода с функциональным конструированием не потребует больших затрат. Оно позволит выполнять трансформации XML, не зная ничего о том, что такое LINQ to XML. Поэтому если вы не знакомы с XSLT, а ваши потребности в трансформации

мации скромны, этот подход может оказаться для вас более подходящим. К тому же, хотя функциональное конструирование менее удобно, чем простая модификация документа XSLT, необходимость перекомпиляции кода для модификации трансформаций может добавить безопасности. Никто не сможет извне вмешаться и модифицировать вашу трансформацию. Таким образом, для тех случаев, когда вы думаете, что нарушаете ограничения, используя закон Сарбейнса-Оксли¹ в качестве оправдания своего бездействия, упирайте на тот факт, что невозможно просто изменить трансформацию без изменения кода. Или же, если вы работаете в области здравоохранения, и не считаете, что можете избежать ответственности за еще одно нарушение HIPAA, то трансформация функциональным конструированием может послужить вам оправданием, если вас обвинят в нерасторопности при удовлетворении потребностей заказчика.

Трансформации с использованием XSLT

Чтобы выполнить трансформацию с использованием XSLT, вы должны обратиться к классам-мостам XmlWriter и XmlReader, экземпляры которых получите от методов класса XDocument — CreateWriter и CreateReader соответственно.

Поскольку пример, показанный в листинге 9.5, требует некоторых пояснений, я приведу их по мере продвижения по его коду. Для начала я специфицирую таблицу стилей трансформации.

Листинг 9.5. Трансформация документа XML с использованием XSLT

```
string xsl =
@"<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:template match='//BookParticipants'>
<html>
<body>
<h1>Book Participants</h1>
<table>
<tr align='left'>
<th>Role</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
<xsl:apply-templates></xsl:apply-templates>
</table>
</body>
</html>
</xsl:template>
<xsl:template match='BookParticipant'>
<tr>
<td><xsl:value-of select='@type' /></td>
<td><xsl:value-of select='FirstName' /></td>
<td><xsl:value-of select='LastName' /></td>
</tr>
</xsl:template>
</xsl:stylesheet>";
```

¹Закон Сарбейнса-Оксли, принятый в 2002 г. (Sarbanes-Oxley Act of 2002) в США, касается дополнений в закон "О фондовых биржах" 1934 г. В соответствии с ним, в обязанность компаний, акции которых котируются на фондовых биржах, регулируемых Комиссией по ценным бумагам и биржам, входит ведение финансовой отчетности согласно общепринятым принципам бухгалтерского учета.

Здесь нет ничего особо потрясающего. Я просто специфицирую некоторый XSL для создания некоторого HTML для отображения моего типичного XML с участниками издания книги в виде таблицы HTML. Затем я создам мой документ XML с участниками книги:

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
```

Это просто мой обычный XML. А теперь о том, где начинается "магия". Мне нужно создать новый XDocument для трансформированной версии. Затем из этого документа я создам XmlWriter, создам экземпляр объекта XslCompiledTransform, загружу объект трансформации таблицей стилей трансформации и выполню трансформацию моего входного XML-документа в выходной XmlWriter:

```
XDocument transformedDoc = new XDocument();
using (XmlWriter writer = transformedDoc.CreateWriter())
{
    XslCompiledTransform transform = new XslCompiledTransform();
    transform.Load(XmlReader.Create(new StringReader(xsl)));
    transform.Transform(xDocument.CreateReader(), writer);
}
Console.WriteLine(transformedDoc);
```

Конечно, после всего этого я отобразжу трансформированную версию документа. Видите, я использую оба класса-моста — XmlWriter и XmlReader — для выполнения трансформации. И вот результат:

```
<html>
<body>
    <h1>Book Participants</h1>
    <table>
        <tr align="left">
            <th>Role</th>
            <th>First Name</th>
            <th>Last Name</th>
        </tr>
        <tr>
            <td>Author</td>
            <td>Joe</td>
            <td>Rattz</td>
        </tr>
        <tr>
            <td>Editor</td>
            <td>Ewan</td>
            <td>Buckingham</td>
        </tr>
    </table>
</body>
</html>
```

Трансформация с использованием функционального конструирования

Хотя LINQ to XML поддерживает трансформацию XSLT, существуют очень эффективные способы выполнения трансформаций на основе самого программного интерфейса LINQ to XML API. Логически рассуждая, трансформация может быть настолько проста, как комбинирование функционально сконструированного дерева XML с встроенным запросом XML.

Совет. Комбинируйте функциональное конструирование со встроенным запросом XML LINQ для выполнения трансформации.

Я объясню XML-трансформацию на примере. Во многих примерах, приведенных в главах, посвященных LINQ to XML, я работал со следующим деревом XML:

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

Давайте представим, что мне нужно трансформировать это дерево XML в следующее:

```
<MediaParticipants type="book">
  <Participant Role="Author" Name="Joe Rattz" />
  <Participant Role="Editor" Name="Ewan Buckingham" />
</MediaParticipants>
```

Чтобы осуществить эту трансформацию, я использую функциональное конструирование со встроенным запросом. С таким подходом вы, по сути, функционально конструируете новый документ, соответствующий желаемой выходной древовидной структуре XML, получая необходимые для этого данные из оригинального исходного документа XML с помощью запроса LINQ to XML. Желаемая выходная древовидная структура определяет ваше функциональное конструирование и логику запроса.

Поскольку эта задача несколько сложнее многих предыдущих примеров LINQ to XML, я объясню все шаг за шагом. Код представлен в листинге 9.6.

Листинг 9.6. Трансформация документа XML

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the original XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

Приведенный код просто создает оригинальный исходный XML-документ, который буду трансформировать, и отображает его. Затем мне нужно построить новый документ и корневой элемент:

```
XDocument xTransDocument = new XDocument(
    new XElement("MediaParticipants",
```

Напомню, что желаемая структура выходного XML-дерева определяет мое функциональное конструирование. В этой точке у меня имеется документ и корневой элемент MediaParticipants. Теперь мне нужно добавить атрибут type к корневому элементу.

```
    new XAttribute("type", "book"),
```

Атрибут type и его значение не существует в исходном документе XML. Он должен быть жестко закодирован или наоборот — сконфигурирован, в моей программной логике, что безопасно, поскольку я уже знаю, что этот код предназначен для описания книг; в противном случае он не должен вызываться.

Теперь у меня есть атрибут type элемента MediaParticipants. Затем мне нужно сгенерировать элемент Participant для каждого элемента BookParticipant исходного XML. Чтобы сделать это, я запрошу в оригинальном XML-документе элементы BookParticipant:

```
xDocument.Element("BookParticipants")
    .Elements("BookParticipant")
```

Теперь у меня есть возвращенная последовательность элементов BookParticipant. Далее мне нужно сгенерировать элемент Participant для каждого элемента BookParticipant и заполнить его атрибуты. Для этого я использую проекцию через операцию Select, чтобы сконструировать элементы Participant:

```
.Select(e => new XElement("Participant",
```

Далее я сконструирую два атрибута — Role и Name — для элемента Participant, получив их значения из элемента BookParticipant:

```
    new XAttribute("Role", (string)e.Attribute("type")),
    new XAttribute("Name", (string)e.Element("FirstName") + " " +
        (string)e.Element("LastName"))));
```

И, наконец, я отобразжу трансформированный XML-документ:

```
Console.WriteLine("Here is the transformed XML document:");
Console.WriteLine(xTransDocument);
```

Посмотрим, получу ли я в результате то, что мне нужно:

Here is the original XML document:

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

Here is the transformed XML document:

```
<MediaParticipants type="book">
    <Participant Role="Author" Name="Joe Rattz" />
    <Participant Role="Editor" Name="Ewan Buckingham" />
</MediaParticipants>
```

Ура! Все прошло блестяще! Я получил точно тот вывод, который хотел. Неплохо, учитывая, что не использовалось ничего помимо LINQ to XML.

Советы

Считаю необходимым дать несколько советов относительно выполнения трансформаций XML посредством LINQ to XML API. Может быть, вы и не нуждаетесь в них, но нет причин им не следовать.

Упрощайте сложные задачи вспомогательными методами

Не существует требования, чтобы каждый кусочек кода, необходимый для выполнения трансформации или запроса находился в самом коде трансформации. Можно создавать вспомогательные методы, которые возьмут на себя наиболее сложную часть рутинной работы.

Ниже приведен пример кода, демонстрирующего, как можно создать вспомогательный метод для разбиения на части наиболее сложной задачи.

Вспомогательный метод для трансформации документа XML

```
static IEnumerable< XElement > Helper()
{
    XElement[] elements = new XElement[] {
        new XElement("Element", "A"),
        new XElement("Element", "B")};
    return(elements);
}
```

В листинге 9.7 я начинаю с конструирования дерева XML. В самом вызове конструктора создается корневой узел по имени `RootElement`. Чтобы создать дочерние узлы, он вызывает вспомогательный метод по имени `Helper`. Не важно, что именно делает этот вспомогательный метод; важно лишь то, что он помогает мне построить некоторую часть моего дерева XML, и что вызов этого метода может быть встроен в функциональное конструирование дерева XML.

Листинг 9.7. Использование вспомогательного метода для трансформации документа XML

```
XElement xElement = new XElement("RootElement", Helper());
Console.WriteLine(xElement);
```

Вот результирующий код:

```
<RootElement>
  <Element>A</Element>
  <Element>B</Element>
</RootElement>
```

Помните, как я говорил в главе 7, что конструктор `XElement` знает, как справиться с `IEnumerable< T >`, что является типом возврата метода `myHelper`. Здорово, правда?

Подавляйте конструирование узла с null

Может случиться, что по той или иной причине вы захотите подавить конструирование некоторых узлов. Возможно, в источнике информации недостает некоторых данных, что вызовет у вас желание пропустить элемент при создании дерева, или же данные окажутся такими, что вы предпочтете пропустить их.

В разделе “Создание элементов с помощью `XElement`” главы 7, когда я описывал конструктор `XElement`, я упоминал, что вы можете передать `null` в качестве значения

объекта для содержимого элемента, и это может быть удобно при выполнении трансформаций. Подавление конструирования узлов — именно тот случай.

Для примера я сначала построю последовательность элементов. Затем начну конструирование нового дерева XML на основе этой последовательности. Однако если значение входного элемента — "A", то мне не нужно создавать выходной элемент для такого входного. Я передам значение null, чтобы это произошло. Код приведен в листинге 9.8.

Листинг 9.8. Подавление конструирования узла с помощью null

```
IEnumerable< XElement> elements =
    new XElement[] {
        new XElement("Element", "A"),
        new XElement("Element", "B")};
XElement xElement = new XElement("RootElement",
    elements.Select(e => (string)e != "A" ? new XElement(e.Name, (string)e) : null));
Console.WriteLine(xElement);
```

Как видите, в этом коде я строю входную последовательность элементов. Затем я конструирую корневой элемент и выполняю перечисление входной последовательности. После этого, применяя операцию Select, до тех пор, пока значений входного элемента не равно "A", я конструирую объект XElement, используя для этого входной элемент. Если же значение входного элемента эквивалентно "A", возвращаю null. Конструктор элемента XElement знает, как обращаться с null: он игнорирует его. В результате любой элемент, чье значение эквивалентно "A", исключается из выходного дерева XML. Вы можете видеть, что я использую новое средство извлечения значения узла, выполняя приведение элемента e к string в лямбда-выражении операции Select.

Вот результат:

```
<RootElement>
  <Element>B</Element>
</RootElement>
```

Обратите внимание, что элемент "A" отсутствует. Конечно, есть и другие способы реализовать ту же логику, не используя null. Например, я мог бы использовать операцию where для того, чтобы отфильтровать элементы со значением "A". Но я хотел продемонстрировать здесь эффект от применения null на очень простом примере.

Есть другие пути использования той же концепции. Возможно, мне нужно генерировать некоторый XML, который позволял бы мне иметь пустой элемент в некоторых экземплярах, существование которых нежелательно. Рассмотрим код из листинга 9.9.

Листинг 9.9. Пример генерации пустого элемента

```
IEnumerable< XElement> elements =
    new XElement[] {
        new XElement("BookParticipant",
            new XElement("Name", "Joe Rattz"),
            new XElement("Book", "Pro LINQ: Language Integrated Query in C# 2008")),
        new XElement("BookParticipant",
            new XElement("Name", "John Q. Public"))};
XElement xElement =
    new XElement("BookParticipants",
        elements.Select(e =>
            new XElement(e.Name,
                new XElement(e.Element("Name").Name, e.Element("Name").Value),
                new XElement("Books", e.Elements("Book")))));
Console.WriteLine(xElement);
```

В первом операторе приведенного кода я генерирую последовательность элементов BookParticipant, состоящую из двух элементов. Обратите внимание, что некоторые из элементов BookParticipant имеют дочерние элементы Book, такие как BookParticipant, у которого дочерний элемент Name имеет значение "Joe Rattz", а некоторые не имеют элементов Book — вроде BookParticipant, у которого дочерний элемент Name имеет значение "John Q. Public".

Во втором операторе я строю дерево XML, используя полученную последовательность элементов. В дереве XML я создаю элемент с тем же именем, что и в исходной последовательности, которым будет BookParticipant. Затем я создаю его дочерний элемент name, а затем — список Book для каждого участника. И вот вывод этого кода:

```
<BookParticipants>
  <BookParticipant>
    <Name>Joe Rattz</Name>
    <Books>
      <Book>Pro LINQ: Language Integrated Query in C# 2008</Book>
    </Books>
  </BookParticipant>
  <BookParticipant>
    <Name>John Q. Public</Name>
    <Books />
  </BookParticipant>
</BookParticipants>
```

XML получился таким, как и следовало ожидать на основе приведенного кода, но заметьте, что элемент Books второго элемента BookParticipant пуст. Что, если вам не нужен пустой элемент Books, если элементов Books нет? Вы могли бы использовать null, чтобы подавить элементы Books, с корректной операцией. В листинге 9.10 я внес небольшое изменение в код, производящий XML.

Листинг 9.10. Пример, предотвращающий пустой элемент

```
IEnumerable< XElement> elements =
  new XElement[] {
    new XElement("BookParticipant",
      new XElement("Name", "Joe Rattz"),
      new XElement("Book", "Pro LINQ: Language Integrated Query in C# 2008")),
    new XElement("BookParticipant",
      new XElement("Name", "John Q. Public"))};
XElement xElement =
  new XElement("BookParticipants",
    elements.Select(e =>
      new XElement(e.Name,
        new XElement(e.Element("Name").Name, e.Element("Name").Value),
        e.Elements("Book").Any() ?
          new XElement("Books", e.Elements("Book")) : null)));
Console.WriteLine(xElement);
```

Существенные изменения в этом коде выделены полужирным. Вместо простого создания элемента Books и спецификации всех существующих элементов Book в качестве его содержимого я использую стандартную операцию Any в сочетании с тернарной операцией (if ? then : else) для создания элемента Books только в случае наличия элементов Book. Если не существует элементов Book, то тернарная операция возвратит null, и конструктор XElement просто проигнорирует null, тем самым исключив создание элемента Books. Это может быть очень удобно. Вот результат после модификации:

```
<BookParticipants>
  <BookParticipant>
    <Name>Joe Rattz</Name>
    <Books>
      <Book>Pro LINQ: Language Integrated Query in C# 2008</Book>
    </Books>
  </BookParticipant>
  <BookParticipant>
    <Name>John Q. Public</Name>
  </BookParticipant>
</BookParticipants>
```

Как видите, теперь второй элемент BookParticipant уже не содержит пустого элемента Books, как это было в предыдущем примере.

Обработка множества соседних узлов одного уровня

Иногда при выполнении XML-трансформации вы знаете точно, сколько выходных элементов каждого типа вам нужно. Но что произойдет, если будет несколько известных элементов, наряду с переменным количеством повторяющихся элементов, и все на одном уровне дерева для каждого вхождения в исходном XML? Скажем, у меня есть показанный ниже XML.

Как я хочу, чтобы выглядел мой входной XML

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

Что если я хочу выровнять эту структуру, чтобы корневой узел BookParticipants содержал только повторяющиеся наборы элементов FirstName, LastName и Nickname, вместо того, чтобы помещать их в дочерний элемент BookParticipant? Я хочу, чтобы целевой XML выглядел следующим образом.

```
<BookParticipants>
  <!-- BookParticipant -->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
  <Nickname>Joey</Nickname>
  <Nickname>Null Pointer</Nickname>
  <!-- BookParticipant -->
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipants>
```

Комментарии не обязательны, но они могут облегчить людям понимание того, что они видят. К тому же без них, если вы взглянете вниз списка, где опять встречается FirstName, то может показаться, что есть BookParticipant по имени Ewan Rattz, когда на самом деле такого нет.

Поскольку этот пример более сложен, я поясню его. Взгляните на пример в листинге 9.11, выполняющий эту трансформацию.

Листинг 9.11. Обработка множества равноправных узлов в плоской структуре

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz"),
            new XElement("Nickname", "Joey"),
            new XElement("Nickname", "Null Pointer")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the original XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

В этой точке я построил дерево XML и отобразил его. Оно соответствует XML, специфицированному в качестве исходного ранее. Теперь мне нужно трансформировать исходный XML:

```
XDocument xTransDocument = new XDocument(
    new XElement("BookParticipants",
        xDocument.Element("BookParticipants")
            .Elements("BookParticipant"))
```

Здесь возникает сложность. Я собираюсь использовать проекцию посредством операции Select, чтобы создать объект, в котором будет содержаться комментарий, имя, фамилия и любые прозвища. Но какого типа объект должен я создать? Я мог бы создать элемент и сделать комментарий, имя и остальное его дочерними элементами, но это добавит в XML новый уровень. Поэтому я должен создать нечто такое, что не приведет к добавлению нового уровня к дереву XML. Массив объектов подошел бы для этого, поскольку в C# 3.0 массив реализует `IEnumerable<T>`, что позволяет работать с массивом как с последовательностью. Как вы помните, надеюсь, из главы 7, когда `IEnumerable` передается в качестве содержимого конструктору `XElement`, то выполняется перечисление последовательности, и каждый объект этой последовательности применяется к конструируемому элементу. Я использую инициализацию коллекции C# 3.0 для наполнения этого массива комментарием, именем, фамилией и прозвищами:

```
.Select(e => new object[] {
    new XComment(" BookParticipant "),
    new XElement("FirstName", (string)e.Element("FirstName")),
    new XElement("LastName", (string)e.Element("LastName")),
    e.Elements("Nickname"))));
Console.WriteLine("Here is the transformed XML document:");
Console.WriteLine(xTransDocument);
```

В этой точке я спроектировал массив, содержащий комментарий, элемент `FirstName`, элемент `LastName` и столько элементов `NickName`, сколько есть в исходном XML. И, наконец, я отображаю трансформированный документ XML.

Этот пример действительно довольно сложен. Обратите внимание, что мой массив объектов включает объект `XComment`, два объекта `XElement` и `IEnumerable< XElement >`. Проектируя вновь созданный массив как возвращаемое значение операции `Select`, возвращается последовательность из `object[]`, `IEnumerable< object[] >`, как содержимое вновь сконструированного элемента `BookParticipants`.

В этом случае каждый объект этой последовательности является массивом объектов, состоящим из комментария, элементов `FirstName` и `LastName`, а также последо-

вательности элементов NickName. Поскольку, как я упомянул, массив объектов не добавляет уровень к дереву XML, он добавляет свои элементы непосредственно в элемент BookParticipants.

Это может показаться запутанным, так что лучше взглянем на результат:

Here is the original XML document:

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

Here is the transformed XML document:

```
<BookParticipants>
  <!-- BookParticipant -->
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
  <Nickname>Joey</Nickname>
  <Nickname>Null Pointer</Nickname>
  <!-- BookParticipant -->
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipants>
```

Трансформированный XML в точности отвечает спецификации. Браво! Действительно искусственная часть этого примера заключена в проекции массива объектов, не-XML класса, для создания равноправных элементов XML без добавления нового уровня XML в дерево.

Проверка достоверности

Интерфейс XML API был бы неполным без возможности проверки достоверности XML. Поэтому LINQ to XML обладает такой способностью — проверять XML-документ на соответствие схеме XML.

Расширяющие методы

LINQ to XML удовлетворяет потребность в проверке достоверности, создавая статический класс System.Xml.Schema.Extensions, содержащий методы проверки достоверности. Эти методы проверки достоверности реализованы в виде расширяющих методов.

Прототипы

Ниже приведен список некоторых доступных методов проверки достоверности из класса System.Xml.Schema.Extensions:

```
void Extensions.Validate(this XDocument source, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler)
void Extensions.Validate(this XDocument source, XmlSchemaSet schemas,
  ValidationEventHandler validationEventHandler, bool addSchemaInfo)
```

```

void Extensions.Validate(this XElement source,
    XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
    ValidationEventHandler validationEventHandler)
void Extensions.Validate(this XElement source,
    XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
    ValidationEventHandler validationEventHandler, bool addSchemaInfo)
void Extensions.Validate(this XAttribute source,
    XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
    ValidationEventHandler validationEventHandler)
void Extensions.Validate(this XAttribute source,
    XmlSchemaObject partialValidationType, XmlSchemaSet schemas,
    ValidationEventHandler validationEventHandler, bool addSchemaInfo)

```

Для каждого типа объектов, на которых может быть вызван метод, предусмотрено по два прототипа. Эти типы — `XDocument`, `XElement` и `XAttribute`. Второй прототип для каждого типа объекта просто добавляет аргумент `bool` для указания того, должна ли добавляться информация схемы к объектам `XElement` и `XAttribute` после проверки достоверности. Первый метод для каждого из типов объектов, который не имеет аргумента `bool`, работает так же, как второй, когда в аргументе `addSchemaInfo` ему передается `false`. В этом случае после проверки достоверности в объекты LINQ to XML никакой информации схемы не добавляется.

Чтобы получить информацию схемы для объектов `XElement` или `XAttribute`, вызывайте метод `GetSchemaInfo` на объекте. Если информация схемы не добавлена, либо в результате применения первого прототипа, либо второго с передачей `false` в аргументе `addSchemaInfo`, то метод `GetSchemaInfo` вернет `null`. В противном случае он вернет объект, реализующий `IXmlSchemaInfo`. Этот объект будет содержать свойство по имени `SchemaElement`, которое вернет объект `XmlSchemaAttribute`, если предположить, что элемент или атрибут корректен. Эти объекты могут использоваться для получения дополнительной информации о схеме.

Важно отметить, что информация о схеме не доступна во время проверки достоверности, а только после ее завершения. Это значит, что вы не можете получить информацию о схеме в вашем обработчике события проверки достоверности. Вызов метода `GetSchemaInfo` вернет `null` в вашем обработчике события проверки достоверности. Это также означает, что проверка достоверности должна завершиться, и что вы не должны генерировать исключений в вашем обработчике события проверки достоверности.

Совет. Информация о схеме не доступна во время проверки достоверности, а только после нее. Вызов `GetSchemaInfo` в вашем обработчике события проверки достоверности вернет `null`.

Обратите внимание, что прототипы метода `Validate` для элементов и атрибутов требуют передачи `XmlSchemaObject` в качестве одного из аргументов. Это значит, что у вас уже должен быть проверен документ, в котором они находятся.

И, наконец, если вы передадите `null` в аргументе `ValidationEventHandler`, то в случае ошибки проверки достоверности будет сгенерировано исключение типа `XmlSchemaValidationException`. Это будет простейшим подходом в проверке достоверности XML-документа.

Получение схемы XML

Если вы заинтересованы в проверке достоверности вашего XML-документа, то не плохо бы вам иметь или знать, как получить файл схемы XSD. В случае если этого нет, я продемонстрирую, как .NET Framework может помочь вам в этом. Рассмотрим пример в листинге 9.12.

Листинг 9.12. Создание схемы XSD извлечением ее из документа XML

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
xDocument.Save("bookparticipants.xml");
XmlSchemaInference infer = new XmlSchemaInference();
XmlSchemaSet schemaSet =
    infer.InferSchema(new XmlTextReader("bookparticipants.xml"));
XmlWriter w = XmlWriter.Create("bookparticipants.xsd");
foreach (XmlSchema schema in schemaSet.Schemas())
{
    schema.Write(w);
}
w.Close();
XDocument newDocument = XDocument.Load("bookparticipants.xsd");
Console.WriteLine("Here is the schema:");
Console.WriteLine("{0}{1}{1}", newDocument, System.Environment.NewLine);

```

В приведенном коде я сначала создаю обычный XML-документ, который уже использовался во многих примерах, и отображаю его для просмотра. Затем я сохраняю документ XML на диске. После этого создаю экземпляр объекта XmlSchemaInference и создаю XmlSchemaSet вызовом метода InferSchema на объекте XmlSchemaInference. Я создаю писатель и выполняю перечислении набора схем, записывая каждую в файл bookparticipants.xsd. И, наконец, загружаю сгенерированный файл схемы XSD и отображаю его. Результат выглядит, как показано ниже.

```

Here is the source XML document:
<BookParticipants>
    <BookParticipant type="Author">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
Here is the schema:
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="BookParticipants">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="BookParticipant">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="FirstName" type="xs:string" />
                            <xs:element name="LastName" type="xs:string" />
                        </xs:sequence>

```

```

<xss:attribute name="type" type="xs:string" use="required" />
</xss:complexType>
</xss:element>
</xss:sequence>
</xss:complexType>
</xss:element>
</xss:schema>

```

Получить схему подобным образом совсем не трудно. Далее я использую сгенерированный файл XSD схемы по имени в примерах проверки достоверности. Также вы должны заметить, что я применяю в данном примере класс XmlSchemaSet, который также будет использован в примерах проверки достоверности.

Примеры

В первом примере я продемонстрирую простейшие средства проверки достоверности документа XML — именно такой подход предпочитает большинство разработчиков. Чтобы сделать это, я просто специфицирую null в качестве аргумента ValidationEventHandler, как показано в листинге 9.13.

Листинг 9.13. Проверка достоверности документа XML с обработкой события проверки по умолчанию

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("MiddleInitial", "C"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
    xDocument.Validate(schemaSet, null);
    Console.WriteLine("Document validated successfully.");
}
catch (XmlSchemaValidationException ex)
{
    Console.WriteLine("Exception occurred: {0}", ex.Message);
    Console.WriteLine("Document validated unsuccessfully.");
}

```

В этом примере я конструирую обычный документ XML, за исключением добавления элемента MiddleInitial, чтобы намеренно сделать документ некорректным. Я использую схему, полученную в предыдущем примере. Обратите внимание, что в аргументе ValidationEventHandler метода Validate я передаю null. Это значит, что если возникнет ошибка проверки достоверности, будет автоматически сгенерировано исключение XmlSchemaValidationException. И вот результат:

```

Here is the source XML document:
<BookParticipants>

```

```

<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <MiddleInitial>C</MiddleInitial>
  <LastName>Rattz</LastName>
</BookParticipant>
<BookParticipant type="Editor">
  <FirstName>Ewan</FirstName>
  <LastName>Buckingham</LastName>
</BookParticipant>
</BookParticipants>
Exception occurred: The element 'BookParticipant' has invalid child element
'MiddleInitial'. List of possible elements expected: 'LastName'.
Document validated unsuccessfully.
Возникло исключение: Элемент 'BookParticipant' имеет недопустимый дочерний элемент
'MiddleInitial'. Список ожидаемых допустимых элементов: 'LastName'.
Проверка достоверности документа завершена неудачей.

```

Работает как заклинание. Все очень просто и совсем неплохо.

В следующем примере я выполню проверку достоверности моего типичного документа XML — того, который был использован для генерации схемы — на предмет соответствия его этой схеме. Конечно, поскольку схема получена на основе этого самого XML-документа, она должна работать. Однако для этого примера мне понадобится метод ValidationEventHandler. Рассмотрим тот, что я собираюсь использовать.

Мой обработчик ValidationEventHandler

```

static void MyValidationEventHandler(object o, ValidationEventArgs vea)
{
    Console.WriteLine("A validation error occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine(vea.Message);
    throw (new Exception(vea.Message));
}

```

В этом обработчике я не делаю ничего помимо вывода сообщения о проблеме и генерации исключения. Конечно, то, как будет выполнена обработка, полностью зависит от моего обработчика. Нет необходимости генерировать исключение. Я мог бы организовать более изящную обработку ошибок проверки достоверности — возможно, игнорируя все или определенные ошибки.

Рассмотрим пример использования этого обработчика, приведенный в листинге 9.14.

Листинг 9.14. Успешная проверка достоверности документа XML по схеме XSD

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
    xDocument.Validate(schemaSet, MyValidationEventHandler);
}

```

```

        Console.WriteLine("Document validated successfully.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception occurred: {0}", ex.Message);
        Console.WriteLine("Document validated unsuccessfully.");
    }
}

```

В этом примере я создаю мой типичный XML-документ и отображаю его на консоли. Затем я создаю экземпляр объекта XmlSchemaSet и добавляю созданный ранее файл схемы с помощью метода Add. Затем я просто вызываю расширяющий метод Validate на XML-документе, передавая ему набор схемы и мой метод-обработчик события проверки достоверности. Обратите внимание, что я поместил вызов метода Validate в блок try/catch в целях безопасности. Взглянем на результат:

Here is the source XML document:

```

<BookParticipants>
    <BookParticipant type="Author">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
Document validated successfully.

```

Как видите, документ XML успешно прошел проверку достоверности. Теперь рассмотрим пример, приведенный в листинге 9.15, с неправильным документом.

Листинг 9.15. Неудачная проверка достоверности документа XML по схеме XSD

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XAttribute("language", "English"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
    xDocument.Validate(schemaSet, MyValidationEventHandler);
    Console.WriteLine("Document validated successfully.");
}
catch (XmlSchemaValidationException ex)
{
    Console.WriteLine("Exception occurred: {0}", ex.Message);
    Console.WriteLine("Document validated unsuccessfully.");
}

```

Этот код идентичен коду предыдущего примера, за исключением того, что я добавил дополнительный атрибут — language. Поскольку схема не специфицирует этот атрибут, документ XML некорректен. И вот результат:

```
Here is the source XML document:
<BookParticipants>
  <BookParticipant type="Author" language="English">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
A validation error occurred processing object type XAttribute.
The 'language' attribute is not declared.
Exception occurred: The 'language' attribute is not declared.
Document validated unsuccessfully.
Ошибка проверки достоверности возникла при обработке объектного типа XAttribute.
Атрибут 'language' не объявлен.
Возникло исключение: Атрибут 'language' не объявлен.
Проверка достоверности документа завершена неудачей.
```

Как видите, документ XML не прошел проверку достоверности успешно. В двух предыдущих примерах я создавал именованный метод MyValidationEventHandler для обработки проверки достоверности. Напомню, что в C# 2.0 появились анонимные методы, а в C# 3.0 — лямбда-выражения. Листинг 9.16 демонстрирует тот же пример, что и предыдущий, за исключением того, что вместо именованного метода для ValidationEventHandler применяется лямбда-выражение.

Листинг 9.16. Неудачная проверка достоверности документа XML по схеме XSD с использованием лямбда-выражения

```
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XAttribute("language", "English"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz")),
    new XElement("BookParticipant",
      new XAttribute("type", "Editor"),
      new XElement("FirstName", "Ewan"),
      new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
try
{
  xDocument.Validate(schemaSet, (o, vea) =>
  {
    Console.WriteLine(
      "A validation error occurred processing object type {0}.",
      o.GetType().Name);
    Console.WriteLine(vea.Message);
    throw (new Exception(vea.Message));
  });
  Console.WriteLine("Document validated successfully.");
}
```

```

catch (XmlSchemaValidationException ex)
{
    Console.WriteLine("Exception occurred: {0}", ex.Message);
    Console.WriteLine("Document validated unsuccessfully.");
}

```

Посмотрите: целый метод специфицирован в виде лямбда-выражения. Разве не вещь? И вот результат:

```

Here is the source XML document:
<BookParticipants>
    <BookParticipant type="Author" language="English">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
A validation error occurred processing object type XAttribute.
The 'language' attribute is not declared.
Exception occurred: The 'language' attribute is not declared.
Document validated unsuccessfully.
Ошибка проверки достоверности возникла при обработке объектного типа XAttribute.
Атрибут 'language' не объявлен.
Возникло исключение: Атрибут 'language' не объявлен.
Проверка достоверности документа завершена неудачей.

```

Теперь я попробую пример, специфицирующий добавление информации схемы, как показано в листинге 9.17.

Листинг 9.17. Неудачная проверка достоверности документа XML по схеме XSD с использованием лямбда-выражения

```

XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("MiddleName", "Carson"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add(null, "bookparticipants.xsd");
xDocument.Validate(schemaSet, (o, vea) =>
{
    Console.WriteLine("An exception occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine("{0}{1}", vea.Message, System.Environment.NewLine);
},
    true);
foreach(XElement element in xDocument.Descendants())
{

```

```

Console.WriteLine("Element {0} is {1}", element.Name,
    element.GetSchemaInfo().Validity);
XmlSchemaElement se = element.GetSchemaInfo().SchemaElement;
if (se != null)
{
    Console.WriteLine(
        "Schema element {0} must have MinOccurs = {1} and MaxOccurs = {2}{3}",
        se.Name, se.MinOccurs, se.MaxOccurs, System.Environment.NewLine);
}
else
{
    // Неверный элемент не имеет SchemaElement.
    Console.WriteLine();
}
}

```

Этот пример начинается так же, как и предыдущий. Он создает документ XML. На этот раз, однако, я добавил в первый BookParticipant дополнительный элемент MiddleName. Это неправильно, потому что не специфицировано в схеме, по которой выполняется проверка достоверности. В отличие от предыдущего примера я не генерирую исключения в моем коде обработчика события проверки достоверности. Как вы, возможно, помните, я упоминал ранее, что проверка достоверности должна завершиться, чтобы иметь добавленную информацию схемы, поэтому ваш обработчик не должен генерировать исключения. Поэтому я также удалил блок try/catch.

После завершения проверки достоверности я выполняю перечисление по всем элементам документа и отображаю информацию об их правильности. Вдобавок я получаю объект SchemaElement из добавленной информации схемы. Обратите внимание, что я проверяю значение SchemaElement на неравенство null, поскольку если элемент неправильный, то свойство SchemaElement будет равно null. В конечном итоге элемент может быть неправильным потому, что его нет в схеме, и тогда откуда взять для него информацию схемы? То же касается свойства SchemaAttribute для неверных атрибутов. Получив объект SchemaElement, я отображаю его свойства Name, MinOccurs и MaxOccurs. Вот результат:

```

Here is the source XML document:
<BookParticipants>
    <BookParticipant type="Author">
        <FirstName>Joe</FirstName>
        <MiddleName>Carson</MiddleName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
An exception occurred processing object type XElement.
The element 'BookParticipant' has invalid child element 'MiddleName'. List of
possible elements expected: 'LastName'.
Element BookParticipants is Invalid
Schema element BookParticipants must have MinOccurs = 1 and MaxOccurs = 1
Element BookParticipant is Invalid
Schema element BookParticipant must have MinOccurs = 1 and MaxOccurs =
79228162514264337593543950335
Element FirstName is Valid
Schema element FirstName must have MinOccurs = 1 and MaxOccurs = 1

```

```

Element MiddleName is Invalid
Element LastName is NotKnown
Element BookParticipant is Valid
Schema element BookParticipant must have MinOccurs = 1 and MaxOccurs =
79228162514264337593543950335
Element FirstName is Valid
Schema element FirstName must have MinOccurs = 1 and MaxOccurs = 1
Element LastName is Valid
Schema element LastName must have MinOccurs = 1 and MaxOccurs = 1

```

В этом выводе нет никаких сюрпризов. Обратите внимание, что свойство MaxOccurs для элемента BookParticipant содержит очень большое число. Это потому, что в схеме атрибут maxOccurs специфицирован как "unbounded" (не ограничен).

И для последних двух примеров проверки достоверности я использую один из прототипов метода Validate, применимый к проверке достоверности элементов. Первое, что вы заметите здесь, это то, что он имеет аргумент, требующий передачи XmlSchemaObject. Это значит, что документ должен уже быть проверен. Это выглядит странным. То есть получается сценарий, когда нужно повторно выполнять проверку достоверности части дерева XML.

Для целей этого сценария предположим, что я загрузил документ XML и проверил его вначале. Затем я позволил пользователю обновить данные об одном из участников работы над книгой, и теперь мне нужно обновить XML-документ, чтобы отразить эти изменения, поэтому я хочу снова выполнить проверку достоверности части дерева XML — после обновлений. И здесь приходят на помощь прототипы метода Validate для элементов и атрибутов. Поскольку этот пример, показанный в листинге 9.18, более сложен, чем некоторые предыдущие, приведу некоторые пояснения. Во-первых, для небольшого разнообразия, и еще потому, что мне нужна расширенная схема для облегчения редактирования дерева XML, я определяю ее программно вместо загрузки из файла, как это делалось в предыдущих примерах.

Листинг 9.18. Успешная проверка достоверности элемента XML

```

string schema =
@"
<?xml version='1.0' encoding='utf-8'?>
<xsschema attributeFormDefault='unqualified'
elementFormDefault='qualified'
xmlns:xss='http://www.w3.org/2001/XMLSchema'>
<xselement name='BookParticipants'>
<xsccomplexType>
<xsssequence>
<xselement maxOccurs='unbounded' name='BookParticipant'>
<xsccomplexType>
<xsssequence>
<xselement name='FirstName' type='xss:string' />
<xselement minOccurs='0' name='MiddleInitial'
    type='xss:string' />
<xselement name='LastName' type='xss:string' />
</xsssequence>
<xseattribute name='type' type='xss:string' use='required' />
</xsccomplexType>
</xselement>
</xsssequence>
</xsccomplexType>
</xselement>
</xsschema>";
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add("", XmlReader.Create(new StringReader(schema)));

```

В приведенном коде я просто скопировал схему из ранее использованного файла. Я нашел все двойные кавычки и заменил их одиночными. Также я добавил элемент MiddleInitial между элементами FirstName и LastName. Обратите внимание, что я специфицировал атрибут minOccurs равным 0, т.е. данный элемент необязателен. Затем я создал набор схем из этой схемы. Теперь самое время создать документ XML:

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
```

Здесь нет ничего нового. Я просто создал тот же документ, как делаю это обычно в примерах, и отобразил его. Теперь выполню проверку достоверности документа:

```
bool valid = true;
xDocument.Validate(schemaSet, (o, vea) =>
{
    Console.WriteLine("An exception occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
}, true);
Console.WriteLine("Document validated {0}.{1}",
    valid ? "successfully" : "unsuccessfully",
    System.Environment.NewLine);
```

Обратите внимание, что я выполняю проверку достоверности немного иначе, чем в предыдущих примерах. Я инициализирую bool значением true, тем самым утверждая, что документ корректен. Внутри обработчика проверки достоверности я устанавливаю его в false. Таким образом, если возникнет ошибка проверки достоверности, valid будет установлено в false. Затем я проверяю значение valid после проверки достоверности, чтобы определить, корректен ли документ, и отображаю сведения о его корректности. В данном примере в этой точке документ корректен.

Теперь предположим, что я позволил пользователю отредактировать информацию о любом определенном участнике работы над книгой. Пользователь отредактировал информацию об участнике по имени "Joe". Поэтому я получаю ссылку на этот элемент, обновляю ее и заново выполняю проверку достоверности после обновления:

```
XElement bookParticipant = xDocument.Descendants("BookParticipant").
    Where(e => ((string)e.Element("FirstName")).Equals("Joe")).First();
bookParticipant.Element("FirstName").
    AddAfterSelf(new XElement("MiddleInitial", "C"));
valid = true;
bookParticipant.Validate(bookParticipant.GetSchemaInfo().SchemaElement, schemaSet,
    (o, vea) =>
{
    Console.WriteLine("An exception occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
}, true);
```

```
Console.WriteLine("Element validated {0}. {1}",
    valid ? "successfully" : "unsuccessfully",
    System.Environment.NewLine);
```

Как видите, я инициализирую `valid` значением `true`, и вызываю метод `Validate` — на этот раз на элементе `bookParticipant` вместо целого документа. Внутри обработчика события проверки достоверности я устанавливаю `valid` в `false`. После проверки достоверности элемента участника книги я отображаю его корректность. Вот результат:

```
Here is the source XML document:
<BookParticipants>
  <BookParticipant type="Author">
    <FirstName>Joe</FirstName>
    <LastName>Rattz</LastName>
  </BookParticipant>
  <BookParticipant type="Editor">
    <FirstName>Ewan</FirstName>
    <LastName>Buckingham</LastName>
  </BookParticipant>
</BookParticipants>
Document validated successfully.
Element validated successfully.
```

Как видите, элемент успешно прошел проверку достоверности. И для последнего примера у меня припасен тот же код, но на этот раз при обновлении элемента `BookParticipont` я создам элемент `MiddleName` вместо `MiddleInitial`, что неправильно. Этот код представлен в листинге 9.19.

Листинг 9.19. Неудачная проверка достоверности элемента XML

```
string schema =
@"
<?xml version='1.0' encoding='utf-8'?>
<xss:schema attributeFormDefault='unqualified'
elementFormDefault='qualified'
xmlns:xss='http://www.w3.org/2001/XMLSchema'>
  <xss:element name='BookParticipants'>
    <xss:complexType>
      <xss:sequence>
        <xss:element maxOccurs='unbounded' name='BookParticipant'>
          <xss:complexType>
            <xss:sequence>
              <xss:element name='FirstName' type='xss:string' />
              <xss:element minOccurs='0' name='MiddleInitial' type='xss:string' />
              <xss:element name='LastName' type='xss:string' />
            </xss:sequence>
            <xss:attribute name='type' type='xss:string' use='required' />
          </xss:complexType>
        </xss:element>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
</xss:schema>";
XmlSchemaSet schemaSet = new XmlSchemaSet();
schemaSet.Add("", XmlReader.Create(new StringReader(schema)));
XDocument xDocument = new XDocument(
  new XElement("BookParticipants",
    new XElement("BookParticipant",
      new XAttribute("type", "Author"),
      new XElement("FirstName", "Joe"),
      new XElement("LastName", "Rattz"))),
```

```

new XElement("BookParticipant",
    new XAttribute("type", "Editor"),
    new XElement("FirstName", "Ewan"),
    new XElement("LastName", "Buckingham"))));
Console.WriteLine("Here is the source XML document:");
Console.WriteLine("{0}{1}{1}", xDocument, System.Environment.NewLine);
bool valid = true;
xDocument.Validate(schemaSet, (o, vea) =>
{
    Console.WriteLine("An exception occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
}, true);
Console.WriteLine("Document validated {0}.{1}",
    valid ? "successfully" : "unsuccessfully",
    System.Environment.NewLine);
 XElement bookParticipant = xDocument.Descendants("BookParticipant").
    Where(e => ((string)e.Element("FirstName")).Equals("Joe")).First();
bookParticipant.Element("FirstName").
    AddAfterSelf(new XElement("MiddleName", "Carson"));
valid = true;
bookParticipant.Validate(bookParticipant.GetSchemaInfo().SchemaElement, schemaSet,
(o, vea) =>
{
    Console.WriteLine("An exception occurred processing object type {0}.",
        o.GetType().Name);
    Console.WriteLine(vea.Message);
    valid = false;
}, true);
Console.WriteLine("Element validated {0}.{1}",
    valid ? "successfully" : "unsuccessfully",
    System.Environment.NewLine);

```

Этот код идентичен коду предыдущего примера, за исключением добавления элемента MiddleInitial. Я специально добавил вместо него неправильный элемент MiddleName. И вот результат:

```

Here is the source XML document:
<BookParticipants>
    <BookParticipant type="Author">
        <FirstName>Joe</FirstName>
        <LastName>Rattz</LastName>
    </BookParticipant>
    <BookParticipant type="Editor">
        <FirstName>Ewan</FirstName>
        <LastName>Buckingham</LastName>
    </BookParticipant>
</BookParticipants>
Document validated successfully.
An exception occurred processing object type XElement.
The element 'BookParticipant' has invalid child element 'MiddleName'. List of
possible elements expected: 'MiddleInitial, LastName'.
Element validated unsuccessfully.

```

Как видите, новый элемент некорректен. Теперь этот пример может показаться не сколько наивным, поскольку я сказал — представьте, что пользователь редактирует документ. Ни один разработчик в здравом уме и трезвой памяти не станет создавать

интерфейс, который позволит пользователю вносить некорректные изменения в процесс редактирования. Но предположим, что пользователем является некоторый другой процесс, обрабатывающий XML-документ. Возможно, вы передали XML-документ чьей-то еще программе, чтобы она выполнила некоторые изменения, и вы знаете, что от нее можно ожидать всякого. В таком случае повторная проверка достоверности совершенно правдана. Вы знаете, что никому доверять нельзя.

XPath

Если вы привыкли использовать XPath, то можете также получить в свое распоряжение некоторые возможности запросов XPath, благодаря классу `System.Xml.XPath.Extensions` из пространства имен `System.Xml.XPath`. Этот класс добавляет возможности поиска XPath через расширяющие методы.

Прототипы

Ниже приведен список некоторых из прототипов методов, доступных в классе `System.Xml.XPath.Extensions`.

```
XPathNavigator Extensions.CreateNavigator(this XNode node);
XPathNavigator Extensions.CreateNavigator(this XNode node, XmlNameTable nameTable);
object Extensions.XPathEvaluate(this XNode node, string expression);
object Extensions.XPathEvaluate(this XNode node, string expression,
    IXmlNamespaceResolver resolver);
 XElement Extensions.XPathSelectElement(this XNode node, string expression);
 XElement Extensions.XPathSelectElement(this XNode node, string expression,
    IXmlNamespaceResolver resolver);
IEnumerable<XElement> Extensions.XPathSelectElements(this XNode node,
    string expression);
IEnumerable<XElement> Extensions.XPathSelectElements(this XNode node,
    string expression, IXmlNamespaceResolver resolver);
```

Примеры

Используя расширяющие методы, можно опросить документ LINQ to XML с применением поисковых выражений. Пример приведен в листинге 9.20.

Листинг 9.20. Опрос XML с синтаксисом XPath

```
XDocument xDocument = new XDocument(
    new XElement("BookParticipants",
        new XElement("BookParticipant",
            new XAttribute("type", "Author"),
            new XElement("FirstName", "Joe"),
            new XElement("LastName", "Rattz")),
        new XElement("BookParticipant",
            new XAttribute("type", "Editor"),
            new XElement("FirstName", "Ewan"),
            new XElement("LastName", "Buckingham"))));
Element bookParticipant = xDocument.XPathSelectElement(
    "//BookParticipants/BookParticipant[FirstName='Joe']");
Console.WriteLine(bookParticipant);
```

Как видите, я создал мой обычный документ XML. На этот раз, однако, я не отобразил его. Я вызвал метод `XPathSelectElement` на документе и представил выражение поиска XPath для нахождения элемента `BookParticipant`, значение чьего элемента `FirstName` равно "Joe". Ниже показан результат:

```
<BookParticipant type="Author">
  <FirstName>Joe</FirstName>
  <LastName>Rattz</LastName>
</BookParticipant>
```

Используя расширяющие методы XPath, вы можете получить ссылку на объект `System.Xml.XPath.XPathNavigator` для осуществления навигации по вашему документу XML, выполнения запроса XPath для возврата элемента или последовательности элементов либо вычисления выражений запросов XPath.

Резюме

К этому моменту, если вы приступили к чтению данной главы, не обладая никакими знаниями XML, могу себе представить вашу растерянность. Если же у вас было базовое понимание XML, но вы ничего не знали о LINQ to XML, я надеюсь, мне удалось сделать его для вас более понятным. Мощь и гибкость программного интерфейса LINQ to XML API оказывает возбуждающее действие.

В процессе написания этой главы и создания примеров, я не раз ловил себя на том, что попадаю в состояние “XML-эйфории” — состояния, лишенного внутреннего желания избегать “реального” XML, постоянно борясь с соблазном воспользоваться в своей ежедневной работе преимуществом простоты, предоставляемой LINQ to XML, невзирая на тот факт, что мой проект не может его использовать, поскольку этот программный интерфейс еще официально не издан. Как много раз я думал, как здорово было бы использовать функциональное конструирование, чтобы “подстегнуть” кусок XML, и возвращаться к реальности ситуации, заставляющей меня использовать мою запасную библиотеку XML и метод `String.Format`.

Не порицайте меня за это. Как я уже упоминал, мне случилось побывать на семинаре Microsoft, где докладчик демонстрировал код построения XML аналогичным образом.

Написав так много примеров для этой главы и предыдущих глав, посвященных LINQ to XML, я даже не могу выразить, с каким нетерпением я жду возможности применения LINQ to XML в реальном рабочем коде. Дело в том, что в LINQ to XML, поскольку работа с XML в нем в основном основана на элементах, а не на документах, в сочетании с возможностью функционального конструирования, создание XML совершенно безболезненно. Оно может даже доставлять удовольствие. Добавьте к легкости создания документов интуитивные средства навигации и модификации, и работа с XML станет доставлять вам радость, по сравнению с альтернативными технологиями.

Облегчение работы с XML, основанное на мощном и гибком языке запросов, делает LINQ to XML моей любимой частью всего LINQ. Если до сих пор вас пугала работа с XML, я думаю, вы найдете LINQ to XML довольно приятным.

ЧАСТЬ IV

LINQ to DataSet

В этой части...

Глава 10. Операции LINQ to DataSet

Глава 11. Дополнительные возможности DataSet

ГЛАВА 10

Операции LINQ to DataSet

Поскольку я еще не описывал LINQ to SQL, позвольте мне упомянуть сейчас, что для использования LINQ to SQL с определенной базой данных для нее должны быть сгенерированы и скомпилированы классы исходного кода, либо же должен быть создан файл отображения. Это значит, что выполнение запросов LINQ с LINQ to SQL на неизвестной до момента выполнения базе данных попросту невозможно. К тому же LINQ to SQL работает только с Microsoft SQL Server. Что же делать разработчику?

Операции LINQ to DataSet позволяют разработчику выполнять запросы LINQ на DataSet, и поскольку DataSet может быть получен с использованием нормальных запросов ADO.NET SQL, интерфейс LINQ to DataSet позволяет выполнять запросы LINQ на любой базе данных, которая может быть опрошена с помощью ADO.NET. Это представляет намного более динамичный интерфейс запросов баз данных, чем LINQ to SQL.

Вы спросите: при каких обстоятельствах может случиться так, что вам будет неизвестна база данных до момента выполнения программы? Это верно, что в типичных приложениях база данных известна на этапе разработки, и потому нет необходимости в LINQ to DataSet. Но как насчет приложений типа утилит баз данных? Например, рассмотрим такое приложение, как диспетчер SQL Server Enterprise Manager. Ему не известно, какие базы данных будут инсталлированы на сервере до момента его запуска. Приложение Enterprise Manager позволяет вам проверить, какие базы инсталлированы на сервере и какие таблицы есть в каждой из них. Не существует способа для разработчика Enterprise Manager сгенерировать классы LINQ to SQL во время компиляции именно для вашей базы данных. И здесь на помощь приходит LINQ to DataSet.

Хотя эта часть книги названа "LINQ to DataSet", вы найдете здесь операции, относящиеся к объектам DataTable, DataRow и DataColumn. Не удивляйтесь, если вы нечасто встретите здесь ссылки на объекты DataSet. Понятно, что в реальных обстоятельствах ваши объекты DataTable почти всегда происходят от объектов DataSet. Однако для независимости от баз данных, краткости и ясности в большинстве приведенных примерах я намеренно создаю простые объекты DataTable программно, а не извлекаю их из базы данных. Операции LINQ to DataSet состоят из нескольких специальных операций из множества сборок и пространств имен, которые позволяют разработчику делать описанные ниже вещи.

- Выполнять операции множеств на последовательностях объектов DataRow.
- Извлекать и устанавливать значения DataColumn.
- Получать стандартные последовательности LINQ `IEnumerable<T>` из DataTable, так что можно вызывать стандартные операции запросов.
- Копировать измененные последовательности из объектов DataRow в DataTable.

В дополнение к этим операциям LINQ to DataSet, как только вы вызовете операцию `AsEnumerable`, то сможете вызывать стандартные операции запросов из LINQ to Objects на возвращенных последовательностях объектов `DataRow`, достигая большей мощи и гибкости.

Необходимые сборки

Для примеров настоящей главы вам понадобится добавить в проект ссылки на динамические библиотеки сборок `System.Data.dll` и `System.Data.DataSetExtensions`, если это не было сделано ранее.

Необходимые пространства имен

Чтобы использовать операции LINQ to DataSet, добавьте директивы `using` в начало вашего кода для пространств имен `System.Linq` и `System.Data`, если не сделали этого раньше:

```
using System.Data;
using System.Linq;
```

Это позволит вашему коду находить операции LINQ to DataSet.

Общий код для примеров

Почти каждый пример настоящей главы потребует объекта `DataTable`, чтобы выполнять запросы LINQ to DataSet. В реальном рабочем коде вы обычно будете получать объекты `DataTable`, опрашивая базу данных. Однако для некоторых примеров я представлю ситуации, когда условий данных из типичной базы будет недостаточно. Например, мне понадобятся дублированные записи для демонстрации метода `Distinct`. Вместо того чтобы “прыгать через обруч”, пытаясь манипулировать базой данных для получения нужной информации, я просто буду программно создавать `DataTable`, содержащий определенные данные, которые мне понадобятся для примера. Это также избавит вас от необходимости иметь специальную базу данных для тестирования большинства этих примеров.

Поскольку я не стану опрашивать базу данных для получения объектов `DataTable`, и для того, чтобы облегчить создание этих объектов, я стану генерировать их из массива объектов предопределенных классов. В качестве предопределенного класса я использую приведенный ниже класс `Student`.

Простой класс с двумя общедоступными членами

```
class Student
{
    public int Id;
    public string Name;
}
```

Вам нужно просто представить, что я опрашиваю таблицу по имени `Students`, в которой каждая запись представляет студента, и эта таблица состоит из двух столбцов: `Id` и `Name`.

Чтобы облегчить создание базы данных `DataTable`, а также предотвратить неясности в существенных деталях каждого примера, я использую общий метод для преобразования массива объектов `Student` в объект `DataTable`. Это позволит легко варьировать данные от примера к примеру. Вот этот общий метод:

Преобразование массива объектов Student в DataTable

```
static DataTable GetDataTable(Student[] students)
{
    DataTable table = new DataTable();
    table.Columns.Add("Id", typeof(Int32));
    table.Columns.Add("Name", typeof(string));
    foreach (Student student in students)
    {
        table.Rows.Add(student.Id, student.Name);
    }
    return (table);
}
```

В этом методе нет ничего сложного. Я просто создаю экземпляр объекта DataTable, два столбца и добавляю строку для каждого элемента в переданный массив students.

Для многих примеров применения операций LINQ to DataSet мне понадобится отображать DataTable, чтобы показать результаты работы кода. Хотя реальные данные в DataTable будут варьироваться, код, необходимый для отображения заголовка объекта DataTable останется неизменным. Вместо многократного повторения этого кода во всех примерах я создам следующий метод и стану вызывать его в каждом примере, где нужно будет вывести заголовок DataTable.

Метод OutputDataTableHeader

```
static void OutputDataTableHeader(DataTable dt, int columnWidth)
{
    string format = string.Format("{0}0,-{1}{2}", "{", columnWidth, "}");
    // Отображение заголовков столбцов.
    foreach(DataColumn column in dt.Columns)
    {
        Console.Write(format, column.ColumnName);
    }
    Console.WriteLine();
    foreach(DataColumn column in dt.Columns)
    {
        for(int i = 0; i < columnWidth; i++)
        {
            Console.Write("=");
        }
    }
    Console.WriteLine();
}
```

Назначение этого метода — вывести заголовок DataTable в табличной форме.

Операции множеств DataRow

Как вы, вероятно, помните, в программном интерфейсе LINQ to Object API есть группа стандартных операций запросов, предназначенных для выполнения операций множеств над последовательностями объектов. Я имею в виду операции Distinct, Except, Intersect, Union и SequenceEqual. Каждая из этих операций выполняет одну из операций множеств над двумя последовательностями.

Каждой из этих операций для выполнения соответствующей операции с множествами необходим способ определения эквивалентности двух элементов множеств. Они осуществляют сравнение элементов, вызывая методы GetHashCode и Equals над элементами. Для DataRow это означает сравнение ссылок, что является нежелательным по-

вседением. Это приведет к некорректному определению эквивалентности элементов, что заставит операции возвращать некорректные результаты. Из-за этого каждая из таких операций имеет дополнительный прототип, который я пропустил в главах, посвященных LINQ to Objects. Этот дополнительный прототип позволяет представлять в качестве аргумента объект `IEqualityComparer`. Удобно, что объект-компаратор по умолчанию — `System.Data.DataRowComparer.Default` — представлен нам специально для этих версий операций. Этот класс-компаратор находится в пространстве имен `System.Data` в сборке `System.Data.Entity.dll`. Этот компаратор определяет эквивалентность элементов, сравнивая количество столбцов и статический тип данных каждого из них, а также используя интерфейс `IComparable` на динамическом типе данных столбца, если тип реализует этот интерфейс. В противном случае он вызывает статический метод `Equals` класса `System.Object`.

Каждый из этих дополнительных прототипов операций определен в статическом классе `System.Linq.Enumerable`, наряду со всеми остальными прототипами этих операций.

В настоящем разделе я представлю несколько примеров, чтобы проиллюстрировать неправильный, и, что более важно — правильный способ выполнения сравнений последовательностей при работе с объектами `DataSet`.

Distinct

Операция `Distinct` исключает дублированные строки из последовательности объектов. Он возвращает объект, который при перечислении исходной последовательности возвращает последовательность объектов с исключеными дублированными строками. Обычно эта операция определяет дублирование строк, вызывая методы `GetHashCode` и `Equals` типа данных каждого из элементов. Однако для объектов типа `DataRow` это даст неверный результат.

Поскольку я собираюсь вызывать дополнительный прототип и предоставлять объект-компаратор `System.Data.DataRowComparer.Default`, эквивалентность элементов будет определена правильно. С ним дублирование строк будет определяться сравнением объектов `DataRow`, с использованием количества столбцов в строке и статического типа данных каждого столбца, а затем использованием интерфейса `IComparable` на каждом столбце, если его динамический тип данных реализует интерфейс `IComparable`, либо вызовом статического метода `Equals` на `System.Object`, если это не так.

Прототипы

Операция `Distinct` имеет один прототип, описанный ниже.

Прототип `Distinct`

```
public static IEnumerable<T> Distinct<T> (
    this IEnumerable<T> source,
    IEqualityComparer<T> comparer);
```

Примеры

В первом примере я создаю `DataTable` из массива объектов `Student`, используя общий метод `GetDataTable`, и массив, содержащий в себе дубликат. Запись, чей `Id` равен 1, в массиве дублируется. Затем я отображаю `DataTable`. Это доказывает, что запись встречается в `DataTable` дважды. Затем я удаляю любые дублированные строки, вызывая операцию `Distinct`, и снова отображаю `DataTable`, показывая, что дублированные строки удалены. Код представлен в листинге 10.1.

Листинг 10.1. Операция Distinct с компаратором эквивалентности

```

Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 6, Name = "Ulyses Hutchens" },
    new Student { Id = 19, Name = "Bob Tanko" },
    new Student { Id = 45, Name = "Erin Doutensal" },
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 12, Name = "Bob Mapplethorpe" },
    new Student { Id = 17, Name = "Anthony Adams" },
    new Student { Id = 32, Name = "Dignan Stephens" }
};
DataTable dt = GetDataTable(students);
Console.WriteLine("{0}Before calling Distinct(){0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in dt.Rows)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
IEnumerable<DataRow> distinct =
    dt.AsEnumerable().Distinct(DataRowComparer.Default);
Console.WriteLine("{0}After calling Distinct(){0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in distinct)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}

```

Обратите внимание, что я использую операцию AsEnumerable для получения последовательности объектов DataRow из DataTable, потому что именно на ней я должен вызывать операцию Distinct. Также заметьте, что в массиве students запись с Id, равным 1, повторяется.

Несомненно, вы заметите, что язываю метод по имени Field на объекте DataRow. Пока вам достаточно понимать, что это вспомогательный метод, который делает получение значения объекта DataColumn из DataRow более удобным. Позднее, в разделе “Операции над полями DataRow” настоящей главы я расскажу об операции Field<T> более подробно.

Вот результат:

Before calling Distinct()	
Id	Name
1	Joe Rattz
6	Ulyses Hutchens
19	Bob Tanko
45	Erin Doutensal
1	Joe Rattz
12	Bob Mapplethorpe
17	Anthony Adams
32	Dignan Stephens

```
After calling Distinct()
Id      Name
=====
1      Joe Rattz
6      Ulyses Hutchens
19     Bob Tanko
45     Erin Doutensal
12     Bob Mapplethorpe
17     Anthony Adams
32     Dignan Stephens
```

Обратите внимание, что в результатах, перед тем, как я вызвал операцию `Distinct`, запись со значением `Id` равным 1, повторялась, а после его вызова второе ее появление было удалено. Во втором примере я собираюсь продемонстрировать результат вызова операции `Distinct`, но без указания объекта-компаратора. Код представлен в листинге 10.2.

Листинг 10.2. Операция `Distinct` без компаратора эквивалентности

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 6, Name = "Ulyses Hutchens" },
    new Student { Id = 19, Name = "Bob Tanko" },
    new Student { Id = 45, Name = "Erin Doutensal" },
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 12, Name = "Bob Mapplethorpe" },
    new Student { Id = 17, Name = "Anthony Adams" },
    new Student { Id = 32, Name = "Dignan Stephens" }
};
DataTable dt = GetDataTable(students);
Console.WriteLine("{0}Before calling Distinct(){0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in dt.Rows)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
IEnumerable<DataRow> distinct = dt.AsEnumerable().Distinct();
Console.WriteLine("{0}After calling Distinct(){0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt, 15);
foreach (DataRow dataRow in distinct)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
```

Единственное отличие этого кода от предыдущего примера в том, что здесь при вызове операции `Distinct` не указан компаратор эквивалентности. Удалит ли он дублированную строку? Давайте посмотрим:

```
Before calling Distinct()
Id      Name
=====
1      Joe Rattz
6      Ulyses Hutchens
19     Bob Tanko
```

```

45      Erin Doutensal
1       Joe Rattz
12     Bob Mapplethorpe
17     Anthony Adams
32     Dignan Stephens

After calling Distinct()
Id      Name
=====
1       Joe Rattz
6       Ulyses Hutchens
19     Bob Tanko
45     Erin Doutensal
1       Joe Rattz
12     Bob Mapplethorpe
17     Anthony Adams
32     Dignan Stephens

```

Нет, дублированная строка не удалена! Как видите, эти два примера сравнивают строки по-разному.

Except

Операция Except предоставляет последовательность объектов DataRow, которые есть в первой последовательности объектов DataRow, но отсутствуют во второй последовательности объектов DataRow. Операция возвращает объект, который при перечислении проходит по первой последовательности объектов DataRow, собирая уникальные элементы, затем проходит по второй последовательности объектов DataRow, исключая те из них, которые обнаружены в первой последовательности. И, наконец, она выдает оставшиеся элементы в том порядке, в котором они были собраны.

Чтобы определить уникальность элементов в одной последовательности, а также то, что один элемент из первой последовательности не эквивалентен элементу во второй последовательности, операция должна иметь возможность определять их эквивалентность. Обычно эта операция определяет эквивалентность, вызывая методы GetHashCode и Equals типа данных каждого элемента. Однако для объектов типа DataRow это может дать некорректный результат.

Поскольку я собираюсь вызывать дополнительный прототип и предоставить объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. С этим объектом строки считаются дублированными в результате сравнения объектов DataRow по количеству столбцов в строке и статическому типу данных каждого столбца, а затем — использования интерфейса IComparable на каждом столбце, если его динамический тип данных реализует интерфейс IComparable, или же вызовом статического метода Equals класса System.Object — если не реализует.

Прототипы

Операция Except имеет один прототип, описанный ниже.

Прототип Except

```

public static IEnumerable<T> Except<T> (
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);

```

Примеры

В этом примере я вызову операцию Except дважды. Первый раз я передам объект-компаратор System.Data.DataRowComparer.Default, так что результаты первого

запроса с операцией Except будет корректным. При втором вызове операции Except я не передам объект-компаратор. Это приведет к некорректному результату запроса. В листинге 10.3 показан код.

Листинг 10.3. Операция Except с компаратором эквивалентности и без

```

Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};
Student[] students2 = {
    new Student { Id = 5, Name = "Abe Henry" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 29, Name = "Future Man" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
    seq1.Except(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Results of Except() with comparer{0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
except = seq1.Except(seq2);
Console.WriteLine("{0}Results of Except() without comparer{0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}

```

Об этом примере также много не скажешь. По сути, я создаю два объекта DataTable, которые наполняются данными из массивов Student. Я создаю последовательность из каждого объекта DataTable, вызывая метод AsEnumerable. Затем вызываю операцию Except, передавая ему объект-компаратор System.Data.DataRowComparer.Default. Второй раззываю его без этого объекта.

Нажав <Ctrl+F5>, посмотрим на результат работы этого кода:

Results of Except() with comparer	
Id	Name
1	Joe Rattz
13	Stacy Sinclair

```
Results of Except() without comparer.
Id      Name
=====
1       Joe Rattz
7       Anthony Adams
13      Stacy Sinclair
72      Dignan Stephens
```

Как видите, операция Except, вызванная с объектом-компаратором System.Data.DataRowComparer.Default, способна правильно определить эквивалентность элементов в двух последовательностях; в то время как операция Except, вызванная без объекта-компаратора, не может идентифицировать эквивалентность элементов двух последовательностей, что является нежелательным поведением для этой операции.

Intersect

Операция Intersect порождает последовательность объектов DataRow, которая представляет собой пересечение двух последовательностей объектов DataRow. Он возвращает объект, который при перечислении проходит по первой последовательности, собирая уникальные элементы, а затем проходит по второй последовательности, отмечая элементы, присутствующие в обеих последовательностях. И, наконец, она выдает отмеченные элементы в порядке, в котором они были собраны.

Чтобы определить уникальность элементов одной последовательности, а также эквивалентность элемента одной последовательности элементу другой последовательности, операция должна уметь определять эквивалентность элементов. Обычно эта операция определяет эквивалентность элементов вызовом методов GetHashCode и Equals типа данных каждого элемента. Однако для объектов типа DataRow это может дать неправильный результат.

Поскольку я собираюсь вызывать дополнительный прототип и предоставлять объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. При этом строка будет считаться дублированной после сравнения объектов DataRow по количеству столбцов в строке и статическому типу данных каждого столбца, затем по использованию интерфейса IComparable на каждом столбце, если ее динамический тип данных реализует интерфейс IComparable, или по вызову статического метода Equals на System.Object, если он этот интерфейс не реализует.

Прототипы

Операция Intersect имеет один прототип, описанный ниже.

Прототип Intersect

```
public static IEnumerable<T> Intersect<T> (
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);
```

Примеры

В этом примере я использую тот же базовый код, который применялся в примере Except, на этот раз заменив вызов операции Except на Intersect. В листинге 10.4 показан код примера.

Листинг 10.4. Операция Intersect с компаратором эквивалентности и без

```

Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

Student[] students2 = {
    new Student { Id = 5, Name = "Abe Henry" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 29, Name = "Future Man" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
    seq1. Intersect(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Results of Intersect() with comparer{0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
except = seq1. Intersect(seq2);
Console.WriteLine("{0}Results of Intersect() without comparer{0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}

```

Здесь нет ничего нового. Я создаю пару объектов DataTable из двух массивов Student и получаю последовательности из них. Затем я вызываю операцию Intersect сначала с объектом-компаратором, а потом без него. Результаты отображаю после каждого вызова Intersect. Нажав <Ctrl+F5>, посмотрим на результат:

```

Results of Intersect() with comparer
Id      Name
=====
7       Anthony Adams
72      Dignan Stephens

Results of Intersect() without comparer
Id      Name
=====

```

Как видите, операция Intersect с компаратором способна правильно определить эквивалентность элементов из двух последовательностей, в то время как операция Intersect без компаратора не идентифицирует никакие элементы двух последовательностей как эквивалентные; такое поведение явно нежелательно для данной операции.

Union

Операция Union производит последовательность объектов DataRow, которая представляет собой объединение двух последовательностей объектов DataRow. Он возвращает объект, который при перечислении проходит по первой последовательности объектов DataRow, затем по второй последовательности объектов DataRow, выводя любой элемент, который еще не был выведен ранее.

Для определения того факта, что элемент уже был выведен, операция должна иметь возможность определять эквивалентность двух элементов. Обычно эта операция определяет эквивалентность элементов посредством вызова методов GetHashCode и Equals типа данных элементов. Однако для объектов типа DataRow это может привести к некорректному результату.

Поскольку я собираюсь вызывать дополнительный прототип и передавать объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. При этом строка будет считаться дублированной после сравнения объектов DataRow по количеству столбцов в строке и статическому типу данных каждого столбца, затем по использованию интерфейса IComparable на каждом столбце, если ее динамический тип данных реализует интерфейс IComparable, или по вызову статического метода Equals на System.Object, если он этот интерфейс не реализует.

Прототипы

Операция Union имеет один прототип, описанный ниже.

Прототип Union

```
public static IEnumerable<T> Union<T> (
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);
```

Примеры

В этом примере я использую тот же базовый код, что и в примере Intersect, за исключением того, что заменяю вызов операции Intersect на Union. Код представлен в листинге 10.5.

Листинг 10.5. Операция Union с компаратором эквивалентности и без

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

Student[] students2 = {
    new Student { Id = 5, Name = "Abe Henry" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 29, Name = "Future Man" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students2);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
IEnumerable<DataRow> except =
    seq1.Union(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("{0}Results of Intersect() with comparer{0}",
```

```

System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}
except = seq1.Union(seq2);
Console.WriteLine("{0}Results of Union() without comparer{0}",
    System.Environment.NewLine);
OutputDataTableHeader(dt1, 15);
foreach (DataRow dataRow in except)
{
    Console.WriteLine("{0,-15}{1,-15}",
        dataRow.Field<int>(0),
        dataRow.Field<string>(1));
}

```

И здесь нет ничего нового. Я создаю пару объектов `DataTable` из двух массивов `Student` и получаю последовательности из них. Затем я вызываю операцию `Intersect` сначала с объектом-компаратором, потом без него. Результаты отображаю после каждого вызова `Intersect`. Нажав `<Ctrl+F5>`, посмотрим на результат:

```

Results of Union() with comparer
Id      Name
=====
1       Joe Rattz
7       Anthony Adams
13      Stacy Sinclair
72      Dignan Stephens
5       Abe Henry
29      Future Man

Results of Union() without comparer
Id      Name
=====
1       Joe Rattz
7       Anthony Adams
13      Stacy Sinclair
72      Dignan Stephens
5       Abe Henry
7       Anthony Adams
29      Future Man
72      Dignan Stephens

```

Обратите внимание, что результат операции `Union` с объектом-компаратором корректен, а результат операции `Union` без объекта-компаратора — нет.

SequenceEqual

Операция `SequenceEqual` сравнивает две последовательности объектов `DataRow`, чтобы определить, эквивалентны ли они. Он выполняет перечисление двух исходных последовательностей, сравнивая соответствующие объекты `DataRow`. Если две исходные последовательности имеют одинаковое количество записей, и все соответствующие объекты `DataRow` в них эквивалентны, возвращается `true`. В противном случае две последовательности не эквивалентны, и возвращается `false`.

Эта операция должна уметь определять эквивалентность двух элементов. Обычно эта операция определяет эквивалентность элементов вызовом методов GetHashCode и Equals типа данных элементов. Однако для объектов типа DataRow это может привести к некорректному результату.

Поскольку я собираюсь вызывать дополнительный прототип и передавать объект-компаратор System.Data.DataRowComparer.Default, эквивалентность элементов будет определена правильно. При этом строка будет считаться дублированной после сравнения объектов DataRow по количеству столбцов в строке и статическому типу данных каждого столбца, затем по использованию интерфейса IComparable на каждом столбце, если ее динамический тип данных реализует интерфейс IComparable, или по вызову статического метода Equals на System.Object, если он этот интерфейс не реализует.

Прототипы

Операция SequenceEqual имеет один прототип, который я опишу.

Прототип SequenceEqual

```
public static bool SequenceEqual<T> (
    this IEnumerable<T> first,
    IEnumerable<T> second,
    IEqualityComparer<T> comparer);
```

Примеры

В этом примере применения операции SequenceEqual я строю две идентичные последовательности объектов DataRow и сравниваю их сначала операцией SequenceEqual с передачей объекта-компаратора, за чем следует сравнение той же операцией, но без объекта-компаратора. Из-за этой разницы в вызове операция SequenceEqual с компаратором возвращает true, указывая на эквивалентность двух последовательностей, в то время как вызов той же операции без компаратора сообщает, что последовательности не эквивалентны. В листинге 10.6 показан код.

Листинг 10.6. Операция SequenceEqual с компаратором эквивалентности и без

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable(students);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
bool equal = seq1.SequenceEqual(seq2, System.Data.DataRowComparer.Default);
Console.WriteLine("SequenceEqual() with comparer : {0}", equal);
equal = seq1.SequenceEqual(seq2);
Console.WriteLine("SequenceEqual() without comparer : {0}", equal);
```

Здесь не о чем говорить, за исключением того, что первый вызов должен сообщать, что две последовательности эквиваленты, в то время как второй — что нет. Результат в точности соответствует ожиданиям:

```
SequenceEqual() with comparer : True
SequenceEqual() without comparer : False
```

Операции над полями DataRow

В дополнение к специальному для `DataRow` классу компаратора для операций множеств есть потребность в некоторых специфичных для `DataRow` операциях. Эти операции определены в сборке `System.Data.DataSetExtensions.dll`, в статическом классе `System.Data.DataRowExtensions`.

Вы, конечно, заметили, что почти во всех приведенных до сих пор примерах я использовал операцию `Field<T>` для извлечения значения объекта `DataColumn` из `DataRow`. У этой операции два предназначения: корректная проверка эквивалентности и обработка значения `null`.

С объектами `DataRow` мы имеем проблему. Их значения `DataColumn` не сравниваются правильно на предмет эквивалентности, когда обращение происходит через индексатор объекта `DataRow`, если столбец имеет тип значения. Причина в том, что поскольку тип данных столбца может быть любым, индексатор возвращает объект типа `System.Object`. Это позволяет индексатору вернуть целое число, строку или любой другой необходимый тип для этого столбца. И это значит, что если тип столбца — `int`, то это тип значения, который должен быть упакован в объект типа `Object`. Подобная упаковка в Microsoft .NET Framework так и называется — **упаковка (boxing)**. Извлечение типа значения обратно из объекта известно как **распаковка (unboxing)**. Именно в упаковке и кроется проблема.

Взглянем на некоторый простой код. Для начала возьмем пример сравнения целочисленного литерала с другим целочисленным литералом, имеющим то же значение, как показано в листинге 10.7.

Листинг 10.7. Сравнение 3 и 3

```
Console.WriteLine("(3 == 3) is {0}.", (3 == 3));
```

Результат этого кода выглядит так:

`(3 == 3) is True.`

Здесь нет абсолютно никакого сюрприза. Но что случится, когда целое число упаковывается? Рассмотрим код из листинга 10.8 и посмотрим на результат его работы.

Листинг 10.8. Сравнение значения 3, приведенного к `Object` с другим значением 3, приведенным к `Object`

```
Console.WriteLine("((Object)3 == (Object)3) is {0}.", ((Object)3 == (Object)3));
```

И вот результат:

`((Object)3 == (Object)3) is False.`

Что же произошло? Дело в том, что с приведением целочисленного литерала 3 к объекту типа `Object` было создано два объекта, и сравнивались ссылки (адреса) двух объектов, которые не эквивалентны. Когда вы обращаетесь к объектам `DataColumn` с использованием индексатора объекта `DataRow`, если любой из столбцов имеет тип значения, то значения таких столбцов упаковываются и не сравниваются правильно на предмет эквивалентности.

Чтобы продемонстрировать это, я создам более сложный пример, который действительно использует объекты `DataColumn`. В примере у меня есть два массива — каждый своего отличающегося типа класса. Один — тот же базовый массив студентов, что я использовал ранее. Второй — массив назначений на курсы с внешними ключами, указываемыми на массив студентов. Вот класс `StudentClass`:

Простой класс с двумя общедоступными свойствами

```
class StudentClass
{
    public int Id;
    public string Class;
}
```

Имея два разных типа классов, мне нужен метод для преобразования этого массива в объект типа DataTable. Вот этот метод:

```
static DataTable GetDataTable2(StudentClass[] studentClasses)
{
    DataTable table = new DataTable();
    table.Columns.Add("Id", typeof(Int32));
    table.Columns.Add("Class", typeof(string));
    foreach (StudentClass studentClass in studentClasses)
    {
        table.Rows.Add(studentClass.Id, studentClass.Class);
    }
    return (table);
}
```

Этот метод — не что иное, как копия существующего метода GetTableData, который был модифицирован для работы с массивами объектов StudentClass. Очевидно, если вы собираетесь работать с массивами в реальном рабочем коде, вам понадобится нечто более абстрактное, чем создание метода для каждого типа класса, для которого вам нужен объект DataTable. Возможно, обобщенный расширяющий метод был бы подходящим подходом. Но как я упоминал в начале моих примеров, вы обычно будете выполнять запросы LINQ to DataSet на данных, взятых из базы; а не из массивов, так что об этом можно не беспокоиться.

Для примера я построю последовательность объектов DataRow из каждого массива и попытаюсь соединить их по общему столбцу Id, который получу посредством индексации в DataRow с именем столбца, которым является Id. В листинге 10.9 показан этот код.

Листинг 10.9. Соединение двух столбцов типа значения посредством индексации в DataRow

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

StudentClass[] classDesignations = {
    new StudentClass { Id = 1, Class = "Sophmore" },
    new StudentClass { Id = 7, Class = "Freshman" },
    new StudentClass { Id = 13, Class = "Graduate" },
    new StudentClass { Id = 72, Class = "Senior" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
string anthonyClass = (from s in seq1
                        where s.Field<string>("Name") == "Anthony Adams"
                        from c in seq2
                        where c["Id"] == s["Id"]
                        select (string)c["Class"]).SingleOrDefault<string>();
Console.WriteLine("Anthony's Class is: {0}",
    anthonyClass != null ? anthonyClass : "null");
```

В этом запросе нужно отметить пару моментов. Во-первых, обратите внимание на выделенную полужирным строку. Здесь я индексирую объект DataRow для получения значений столбцов. Поскольку тип данных значений столбцов — строка, они упаковываются, что означает потенциальную проблему определения эквивалентности. Вдобавок вы можете видеть, что я использую в этом примере операцию `Field<T>` при сравнении поля `Name` со значением "Anthony Adams". Пока не обращайте на это внимания. Просто примите к сведению, что я вызываю операцию `Field<T>` для преодоления проблемы с упаковкой поля `Name`, которое использую наряду с полем `Id`. Также отметьте, что этот запрос комбинирует синтаксис выражений со стандартным синтаксисом точечной нотации. Как видите, я выполняю также соединение двух объектов `DataTable`. Запустим код и посмотрим результат:

```
Anthony's Class is: null
```

Строка `anthonyClass` равна `null`. Это потому, что соединение не может найти в `seq2` записи с эквивалентным значением поля `Id`. Причина в упаковке поля `Id` при извлечении с использованием индексатора `DataRow`. Теперь вы можете обработать распаковку самостоятельно, изменив строку

```
where c["Id"] == s["Id"]
```

на

```
where (int)c["Id"] == (int)s["Id"]
```

Листинг 10.10 содержит полный пример с замененной строкой.

Листинг 10.10. Использование приведения для корректной проверки эквивалентности

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

StudentClass[] classDesignations = {
    new StudentClass { Id = 1, Class = "Sophomore" },
    new StudentClass { Id = 7, Class = "Freshman" },
    new StudentClass { Id = 13, Class = "Graduate" },
    new StudentClass { Id = 72, Class = "Senior" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
string anthonyClass = (from s in seq1
    where s.Field<string>("Name") == "Anthony Adams"
    from c in seq2
    where (int)c["Id"] == (int)s["Id"]
    select (string)c["Class"]).SingleOrDefault<string>();

Console.WriteLine("Anthony's Class is: {0}",
    anthonyClass != null ? anthonyClass : "null");
```

Запустив этот код, вы получите следующий результат:

```
Anthony's Class is: Freshman
```

Это решает проблему упаковки. Однако остается еще одна. Когда вы попытаетесь извлечь значение столбца, используя индексатор объекта `DataRow`, помните, что значение столбца возвращается как объект типа `Object`. Поэтому для того, чтобы сравнить

его с любым значением или присвоить переменной, необходимо выполнить приведение к другому типу данных, как я сделал это, приведя его к int. Поскольку объекты DataSet используют значение DBNull.Value в качестве значения столбца null, то если значение столбца — DBNull.Value, то приведение его к другому типу данных вызовет генерацию исключения.

К счастью, LINQ to DataSet заставил исчезнуть обе эти проблемы — сравнение упакованных значений и обработка null — благодаря операциям Field<T> и SetField<T>. В листинге 10.11 показан предыдущий пример с использованием операции Field<T>.

Листинг 10.11. Использование операции Field

```

Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};
StudentClass[] classDesignations = {
    new StudentClass { Id = 1, Class = "Sophomore" },
    new StudentClass { Id = 7, Class = "Freshman" },
    new StudentClass { Id = 13, Class = "Graduate" },
    new StudentClass { Id = 72, Class = "Senior" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataTable dt2 = GetDataTable2(classDesignations);
IEnumerable<DataRow> seq2 = dt2.AsEnumerable();
string anthonyClass = (from s in seq1
                        where s.Field<string>("Name") == "Anthony Adams"
                        from c in seq2
                        where c.Field<int>("Id") == s.Field<int>("Id")
                        select (string)c["Class"]).SingleOrDefault<string>();
Console.WriteLine("Anthony's Class is: {0}",
    anthonyClass != null ? anthonyClass : "null");

```

Этот код такой же, как и в предыдущем примере, за исключением вызова операции Field<T> вместо приведения поля к типу int. И вот результат:

Anthony's Class is: Freshman

Field<T>

Как только что было показано в листинге 10.11, операция Field<T> позволяет получить значение столбца из объекта DataRow и устраниет проблемы приведения DBNull.Value и сравнения упакованных значений, описанные мной выше.

Прототипы

Операция Field имеет шесть прототипов, описанных ниже.

Первый прототип возвращает значение столбца для DataColumn и специфицированной версии.

Первый прототип Field

```

public static T Field (
    this DataRow first,
    System.Data.DataColumn column,
    System.Data.DataRowVersion version);

```

Второй прототип возвращает значение столбца с указанным именем и версией.

Второй прототип Field

```
public static T Field (
    this DataRow first,
    string columnName,
    System.Data.DataRowVersion version);
```

Третий прототип возвращает значение указанного по порядку столбца указанной ячейки.

Третий прототип Field

```
public static T Field (
    this DataRow first,
    int ordinal,
    System.Data.DataRowVersion version);
```

Четвертый прототип возвращает текущее значение столбца, только для указанного DataColumn.

Четвертый прототип Field

```
public static T Field (
    this DataRow first,
    System.Data.DataColumn column);
```

Пятый прототип возвращает текущее значение только столбца с указанным именем.

Пятый прототип Field

```
public static T Field (
    this DataRow first,
    string columnName);
```

Шестой прототип возвращает текущее значение только для столбца с указанным порядковым номером.

Шестой прототип Field

```
public static T Field (
    this DataRow first,
    int ordinal);
```

Как вы, возможно, заметили, первые три прототипа позволяют вам специфицировать, какие DataRowVersion значений объектов DataColumn вы хотите извлечь.

Примеры

До сих пор вы уже не раз сталкивались с разнообразными вызовами операции Field<T>. Но только теперь вы можете увидеть все прототипы в действии. В листинге 10.12 показан тривиальный пример каждого из них.

Листинг 10.12. Пример вызова каждого прототипа операции Field

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
```

```

int id;
// Применение прототипа 1.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>(dt1.Columns[0], DataRowVersion.Current));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 1 is: {0}", id);
// Применение прототипа 2.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>("Id", DataRowVersion.Current));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 2 is: {0}", id);
// Применение прототипа 3.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>(0, DataRowVersion.Current));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 3 is: {0}", id);
// Применение прототипа 4.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>(dt1.Columns[0]));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 4 is: {0}", id);
// Применение прототипа 5.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>("Id"));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 5 is: {0}", id);
// Применение прототипа 6.
id = (from s in seq1
       where s.Field<string>("Name") == "Anthony Adams"
       select s.Field<int>(0));
Single<int>();
Console.WriteLine("Anthony's Id retrieved with prototype 6 is: {0}", id);

```

Здесь нет ничего существенного. Я объявляю массив студентов и создаю из него объект DataTable, как и в большинстве других примеров. Затем также получаю последовательность объектов DataRow. После чего один за другим использую прототипы операции Field<T> для получения поля по имени Id. Обратите внимание, что в каждом запросе поля Id я также применяю операцию Field<T>, в части Where запроса. И вот результат:

```

Anthony's Id retrieved with prototype 1 is: 7
Anthony's Id retrieved with prototype 2 is: 7
Anthony's Id retrieved with prototype 3 is: 7
Anthony's Id retrieved with prototype 4 is: 7
Anthony's Id retrieved with prototype 5 is: 7
Anthony's Id retrieved with prototype 6 is: 7

```

Прежде чем перейти к операции SetField<T>, я хочу представить пример, демонстрирующий один из прототипов, который позволяет специфицировать DataRowVersion значения объекта DataColumn, подлежащего извлечению. Чтобы привести этот пример мне нужно модифицировать значение одного из объектов DataColumn, используя для этого операцию SetField<T>. Так как мы еще пока не обсуждали операцию SetField<T>, пока не вдавайтесь в его подробности. Я расскажу о нем в следующем разделе.

Также, поскольку целью этой главы является объяснение операций LINQ to DataSet, а не детальное обсуждение работы класса DataSet, я лишь кратко коснусь пары дополнительных методов DataSet, которые вызываются в этом примере. Код приведен в листинге 10.13.

Листинг 10.13. Прототип операции Field с указанным DataRowVersion

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};
DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
DataRow row = (from s in seq1
               where s.Field<string>("Name") == "Anthony Adams"
               select s).Single<DataRow>();
row.AcceptChanges();
row.SetField("Name", "George Oscar Bluth");
Console.WriteLine("Original value = {0} : Current value = {1}",
    row.Field<string>("Name", DataRowVersion.Original),
    row.Field<string>("Name", DataRowVersion.Current));
row.AcceptChanges();
Console.WriteLine("Original value = {0} : Current value = {1}",
    row.Field<string>("Name", DataRowVersion.Original),
    row.Field<string>("Name", DataRowVersion.Current));
```

В этом примере я получаю последовательность из массива студентов, как обычно это делаю. Затем я запрашиваю единственный объект DataRow, в котором могу провести некоторые изменения. Первое, что представляет интерес — метод AcceptChanges, вызванный после получения объекта DataRow. Я вызываю этот метод, чтобы заставить объект DataRow принять текущее значение каждого объекта DataColumn внутри оригинальной версии. Без этого не было бы оригинальной версии значений объектов DataColumn, и простая попытка обратиться к оригинальной версии поля вызвала бы генерацию исключения. Таким образом, объект DataRow готов начать отслеживать изменения значений объектов DataColumn. Мне нужно это для того, чтобы можно было получить разные версии DataRowVersion значений DataColumn объекта DataRow.

Вызвав метод AcceptChanges первый раз, я устанавливаю поле посредством операции SetField. Затем отображаю исходную и текущую версии значения DataColumn Name на консоли. В этой точке оригинальная версия должна быть "Anthony Adams", а текущая — "George Oscar Bluth". Это позволяет вам увидеть разные версии, которые вы можете получить из объекта DataRow.

После этого, чтобы стало еще интереснее, я второй раз вызываю метод AcceptChanges и снова отображаю оригинальную и текущую версии значения объекта DataColumn. На этот раз и оригинальная, и текущая версии должны совпадать и содержать строку "George Oscar Bluth", потому что я указал объекту DataRow принять изменения в качестве текущей версии. Посмотрим результат:

```
Original value = Anthony Adams : Current value = George Oscar Bluth
Original value = George Oscar Bluth : Current value = George Oscar Bluth
```

Все работает. Помните, однако, что без первоначального вызова метода AcceptChanges я мог бы изменять значение объекта DataColumn хоть целый день, и никакой оригинальной версии не было бы.

Также вы можете вспомнить, как я упоминал о том, что одним из дополнительных преимуществ операции `Field<T>` является ее способность справляться с ситуацией, когда поля содержат значение `null`. Взглянем на пример в листинге 10.14, где имя студента имеет значение `null`, но я не использую операцию `Field<T>`.

Листинг 10.14. Пример без операции `Field`, когда присутствует `null`

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = null },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
string name = seq1.Where(student => student.Field<int>("Id") == 7)
    .Select(student => (string)student["Name"])
    .Single();
Console.WriteLine("Student's name is '{0}'", name);
```

Это чрезвычайно простой пример. Обратите внимание, что я устанавливаю значение члена `Name` записи `Student` с `Id`, равным 7, в `null`. Также отметьте, что вместо использования операции `Field<T>` я просто обращаюсь по индексу к значению внутри `DataRow` и выполняю приведение его к типу `string`. Посмотрим на результат:

```
Unhandled Exception: System.InvalidCastException: Unable to cast object of type 'System.DBNull' to type 'System.String'.
Необработанное исключение: System.InvalidCastException: Невозможно привести объект типа 'System.DBNull' к типу 'System.String'.
...
...
```

Так что же произошло? А случилось то, что значение объекта `DataColumn` равно `DBNull`, и вы не можете привести его к `string`. Существуют довольно громоздкие решения, которые позволили бы избежать этой сложности, но операция `Field<T>` придумана специально, чтобы упростить их для вас. Рассмотрим такой же пример, но на этот раз с применением операции `Field<T>` для получения значения объекта `DataColumn`. Код представлен в листинге 10.15.

Листинг 10.15. Пример с операцией `Field`, когда присутствует значение `null`

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = null },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
string name = seq1.Where(student => student.Field<int>("Id") == 7)
    .Select(student => student.Field<string>("Name"))
    .Single();
Console.WriteLine("Student's name is '{0}'", name);
```

Все то же самое, но теперь я использовал операцию `Field<T>` вместо приведения к `string`. Посмотрим, что получилось:

```
Student's name is ''
```

С этим гораздо легче иметь дело.

SetField<T>

Как и в случае извлечения объектов DataColumn, null неблагоприятно влияет и на установку объектов DataColumn. Чтобы помочь справиться с этой проблемой, была создана операция SetField<T>. Она применяется в ситуации, когда устанавливается значение объекта DataColumn из допускающего null типа данных, чье текущее значение — null.

Прототипы

Операция SetField<T> имеет три прототипа, описанных ниже.

Первый прототип позволяет устанавливать текущее значение столбца для указанного объекта DataColumn.

Первый прототип SetField<T>

```
public static void SetField (
    this DataRow first,
    System.Data.DataColumn column,
    T value);
```

Второй прототип позволяет вам устанавливать текущее значение столбца с указанным именем.

Второй прототип SetField<T>

```
public static void SetField (
    this DataRow first,
    string columnName,
    T value);
```

Третий прототип позволяет вам устанавливать текущее значение столбца с указанным порядковым номером.

Третий прототип SetField<T>

```
public static void SetField (
    this DataRow first,
    int ordinal,
    T value);
```

Примеры

В качестве примера применения операции SetField<T>, показанного в листинге 10.16, сначала я отображаю последовательность объектов DataRow, содержащих студентов. Затем я запрашиваю одного из студентов по имени из последовательности объектов DataRow и изменяю его имя, используя операцию SetField<T>. Затем я снова отображаю последовательность объектов DataRow после проведенных изменений. Далее все повторяется с каждым прототипом.

Листинг 10.16. Пример с применением каждого прототипа операции SetField

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
IEnumerable<DataRow> seq1 = dt1.AsEnumerable();
Console.WriteLine("{0}Results before calling any prototype:",
    System.Environment.NewLine);
```

```

foreach (DataRow dataRow in seq1)
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
                      dataRow.Field<string>("Name"));
}
// Применение прототипа 1.
(from s in seq1
 where s.Field<string>("Name") == "Anthony Adams"
 select s).Single<DataRow>().SetField(dt1.Columns[1], "George Oscar Bluth");
Console.WriteLine("{0}Results after calling prototype 1:",
                  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
                      dataRow.Field<string>("Name"));
}
// Применение прототипа 2.
(from s in seq1
 where s.Field<string>("Name") == "George Oscar Bluth"
 select s).Single<DataRow>().SetField("Name", "Michael Bluth");
Console.WriteLine("{0}Results after calling prototype 2:",
                  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
                      dataRow.Field<string>("Name"));
}
// Применение прототипа 3.
(from s in seq1
 where s.Field<string>("Name") == "Michael Bluth"
 select s).Single<DataRow>().SetField("Name", "Tony Wonder");
Console.WriteLine("{0}Results after calling prototype 3:",
                  System.Environment.NewLine);
foreach (DataRow dataRow in seq1)
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
                      dataRow.Field<string>("Name"));
}

```

Этот код не так плох, как кажется. После того, как я получил последовательность студентов и отобразил их, идет блок кода, повторяющийся три раза — по одному для каждого прототипа. Каждый блок содержит запрос LINQ, извлекающий поле и обновляющий значение, отображающий строку заголовка на консоли, а затем — каждую строку в последовательности, чтобы показать изменение, проведенное в поле.

В этом примере следует отметить две вещи. В каждом запросе LINQ, где запрашивается DataRow по полю Name, я смешиваю синтаксис выражений запросов со стандартным синтаксисом точечной нотации. Также я использую операцию Flat<T> для нахождения записи, которую собираюсь установить операцией SetField<T>. После получения последовательности объектов DataRow студентов я работаю с ними всеми прототипами SetField<T> — один за другим. На протяжении примера я запрашиваю ранее измененный элемент по значению и снова изменяю его. Например, для первого прототипа я просто запрашиваю элемент, чье поле Name равно "Anthony Adams", и устанавливаю его в "George Oscar Bluth". Для второго прототипа я запрашиваю поле со значением "George Oscar Bluth" в Name, и изменяю его на что-то другое, что заслужу в следующем прототипе. Конечно, после обновления значения каждого элемента

я отображаю последовательность на консоли, чтобы вы убедились, что значение этого элемента действительно было обновлено.

Одной из изюминок этого примера является то, что я запрашиваю элемент и обновляю его значение в единственном операторе. Это настолько мощно, что может показаться какой-то иллюзией, но как вы увидите далее, никакой магии здесь нет. Вот результат:

Results before calling any prototype:

```
Student Id = 1 is Joe Rattz
Student Id = 7 is Anthony Adams
Student Id = 13 is Stacy Sinclair
Student Id = 72 is Dignan Stephens
Results after calling prototype 1:
Student Id = 1 is Joe Rattz
Student Id = 7 is George Oscar Bluth
Student Id = 13 is Stacy Sinclair
Student Id = 72 is Dignan Stephens
Results after calling prototype 2:
Student Id = 1 is Joe Rattz
Student Id = 7 is Michael Bluth
Student Id = 13 is Stacy Sinclair
Student Id = 72 is Dignan Stephens
Results after calling prototype 3:
Student Id = 1 is Joe Rattz
Student Id = 7 is Tony Wonder
Student Id = 13 is Stacy Sinclair
Student Id = 72 is Dignan Stephens
```

Как видите, поле Name соответствующего элемента обновляется каждый раз.

Операции DataTable

В дополнение к специфичным для DataRow операциям в классе DataRowExtensions предусмотрены также некоторые операции, специфичные для DataTable. Эти операции определены в сборке System.Data.Entity.dll, в статическом классе System.Data.DataTableExtensions.

AsEnumerable

Вы, вероятно, удивитесь, встретив здесь операцию AsEnumerable. Фактически вы еще больше удивитесь, узнав, что предусмотрен специальный AsEnumerable для класса DataTable, возвращающего последовательность объектов DataRow. Если это так, я могу быть доволен, поскольку это означает, что вас не удивляет на протяжении всей этой главы, почему я не упоминал ее до сих пор. А между тем, я вызывал ее почти в каждом примере.

Да, если вы посмотрите на статический класс System.Data.DataTableExtensions, то увидите, что там есть операция AsEnumerable. Назначение этой операции — возвращать последовательность типа `IEnumerable<DataRow>` из объекта DataTable.

Прототипы

Операция AsEnumerable имеет один прототип, описанный ниже.

Прототип AsEnumerable

```
public static IEnumerable<DataRow> AsEnumerable (
    this DataTable source
);
```

Эта операция, будучи вызванной на объекте `DataTable`, возвращает последовательность объектов `DataRow`. Это обычно первый шаг при выполнении запроса LINQ to DataSet на `DataTable` объекта `DataSet`. Вызывая эту операцию, вы можете получить последовательность `IEnumerable<T>`, где `T` является `DataRow`, что позволяет вам вызывать множество операций LINQ, которые могут быть вызваны на последовательности типа `IEnumerable<T>`.

Примеры

В этой главе нет недостатка в примерах. Поскольку вызов операции `AsEnumerable` – первый шаг при выполнении запроса LINQ to DataSet, почти каждый пример этой главы вызывает операцию `AsEnumerable`. Поэтому нет необходимости представлять здесь еще один.

`CopyToDataTable<DataRow>`

Теперь, когда вы знаете, как запрашивать и модифицировать значение `DataColumn` объекта `DataRow`, вы можете быть заинтересованы в помещении этой последовательности модифицированных объектов `DataRow` в `DataTable`. Операция `CopyToDataTable` предназначена именно для этой цели.

Прототипы

Операция `CopyToDataTable` имеет два прототипа, описанных ниже.

Первый прототип вызывается на `IEnumerable<DataRow>` и возвращает `DataTable`. Он используется для создания нового объекта `DataTable` из последовательности объектов `DataRow`.

Первый прототип `CopyToDataTable`

```
public static DataTable CopyToDataTable<T> (
    this IEnumerable<T> source
) where T : DataRow;
```

Первый прототип автоматически устанавливает оригинальные версии для каждого поля, не требуя от вас вызова метода `AcceptChanges`.

Второй прототип вызывается на `IEnumerable<DataRow>` исходного `DataTable`, чтобы обновить уже существующий целевой объект `DataTable`, на основе специфицированного значения `LoadOptions`.

Второй прототип `CopyToDataTable`

```
public static void CopyToDataTable<T> (
    this IEnumerable<T> source,
    DataTable table,
    LoadOption options
) where T : DataRow;
```

Значение переданного аргумента `LoadOptions` информирует операцию о том, что должны быть изменены только оригинальные значения столбцов, либо текущие значения, либо те и другие. Это полезно при управлении изменениями `DataTable`. Для `LoadOption` доступны описанные ниже значения.

- `OverwriteChanges`: в каждом столбце будут обновлены и оригинальное значение и текущее.
- `PreserveChanges`: обновляется только оригинальное значение каждого столбца.
- `Upsert`: обновляется только текущее значение каждого столбца.

Этот аргумент LoadOption, однако, создает небольшую проблему. Обратите внимание, что описание каждого возможного значения ссылается на обновление значений столбца. Это, конечно же, означает также обновление столбцов записи, уже находящейся в целевом DataTable. Как же операция CopyToDataTable узнает, какая именно запись в целевом DataTable соответствует записи в исходном DataTable? Другими словами, когда он пытается копировать запись из исходного DataTable в целевой, и получает параметр LoadOption, каким образом он узнает, следует ли просто добавить запись из исходного DataTable или же обновить уже существующую запись целевого DataTable? Ответ — никак, если только ему ничего не известно о полях первичных ключей в DataTable.

Поэтому для правильной работы этого прототипа операции CopyDataTable целевой объект DataTable должен иметь соответствующую информацию о полях первичного ключа. Если не указывать первичные ключи, то этот прототип добавит все записи из исходного объекта DataTable в целевой DataTable.

Это одно дополнительное усложнение, о котором следует помнить при работе с данным прототипом операции. Поскольку, используя этот прототип, вы, возможно, заинтересованы в сравнении оригинальной и текущей версий значений полей, не забудьте, что с этим прототипом операции CopyDataTable поле не имеет оригинальной версии, если только не был вызван метод AcceptChanges. Попытка обратиться к оригинальной версии, когда она не существует, приведет к генерации исключения. Однако вы можете вызвать метод HasVersion на каждом объекте DataRow, прежде чем пытаться получить доступ к оригинальной версии, чтобы узнать, существует ли она и предотвратить этот тип исключений.

Примеры

В качестве примера первого прототипа операции CopyToDataTable я просто модифицирую поле в DataTable, создам новый DataTable из модифицированного посредством вызова операции CopyToDataTable и затем отобразжу содержимое нового объекта DataTable. Код представлен в листинге 10.17.

Листинг 10.17. Вызов первого прототипа операции CopyToDataTable

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
Console.WriteLine("Original DataTable:");
foreach (DataRow dataRow in dt1.AsEnumerable())
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
        dataRow.Field<string>("Name"));
}

(from s in dt1.AsEnumerable()
    where s.Field<string>("Name") == "Anthony Adams"
    select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
Console.WriteLine("{0}New DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
    Console.WriteLine("Student Id = {0} is {1}", dataRow.Field<int>("Id"),
        dataRow.Field<string>("Name"));
}
```

Как я уже говорил, сначала я создаю DataTable из массива студентов, как обычно делал и в предыдущих примерах. Затем я отображаю содержимое этого DataTable на консоли. После этого модифицирую поле Name одного из объектов DataRow. Затем создаю новый DataTable вызовом операции CopyToDataTable. И, наконец, отображаю содержимое вновь созданного DataTable.

Готовы ли вы к решающему удару? Получите!

```
Original DataTable:  
Student Id = 1 is Joe Rattz  
Student Id = 7 is Anthony Adams  
Student Id = 13 is Stacy Sinclair  
Student Id = 72 is Dignan Stephens  
New DataTable:  
Student Id = 1 is Joe Rattz  
Student Id = 7 is George Oscar Bluth  
Student Id = 13 is Stacy Sinclair  
Student Id = 72 is Dignan Stephens
```

Как видите, я не только имею данные в новом DataTable, но эти данные в модифицированной версии, как и следовало ожидать.

В следующем примере я хочу продемонстрировать второй прототип операции CopyToDataTable. Как вы, возможно, помните, я уже упоминал, что для правильной работы аргумента LoadOption должны быть определены первичные ключи в целевом DataSet. В данном примере я не стану их указывать, чтобы вы увидели поведение. Поскольку данный пример несколько сложнее, приведу необходимые пояснения. Код представлен в листинге 10.18.

Листинг 10.18. Вызов второго прототипа операции CopyToDataTable, когда первичные ключи не установлены

```
Student[] students = {  
    new Student { Id = 1, Name = "Joe Rattz" },  
    new Student { Id = 7, Name = "Anthony Adams" },  
    new Student { Id = 13, Name = "Stacy Sinclair" },  
    new Student { Id = 72, Name = "Dignan Stephens" }  
};  
DataTable dt1 = GetDataTable(students);  
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
```

Пока ничего нового. Я создал исходный DataTable из массива students. Целевой DataTable я создам вызовом операции CopyToDataTable на исходном DataTable. Обратите внимание, что поскольку я вызвал первый прототип операции CopyToDataTable, мне незачем вызывать метод AcceptChanges на целевом DataTable. Это важно отметить, потому что в следующем сегменте кода я ссылаюсь на оригинальную версию поля Name. Если бы не тот факт, что первый прототип операции CopyToDataTable устанавливает для вас оригинальные версии полей, в этой ситуации было бы сгенерировано исключение.

```
Console.WriteLine("Before upserting DataTable:");  
foreach (DataRow dataRow in newTable.AsEnumerable())  
{  
    Console.WriteLine("Student Id = {0} : original {1} : current {2}",  
        dataRow.Field<int>("Id"),  
        dataRow.Field<string>("Name", DataRowVersion.Original),  
        dataRow.Field<string>("Name", DataRowVersion.Current));  
}
```

Здесь нет ничего особенного за исключением ссылки на оригинальную версию поля `Name` в записи и отсутствия генерации исключения при этом, поскольку этот прототип `CopyToDataTable` устанавливает оригинальную версию полей.

```
(from s in dt1.AsEnumerable()
where s.Field<string>("Name") == "Anthony Adams"
select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
dt1.AsEnumerable().CopyToDataTable(newTable, LoadOption.Upsert);
```

Это наиболее впечатляющий сегмент кода в данном примере. Обратите внимание, что я изменяю значение поля `Name` для одной записи в исходном `DataTable`, используя операцию `SetField<T>`. Затем я вызываю операцию `CopyToDataTable`, указывая, что должно произойти копирование типа `LoadOption.Upsert`, имея в виду обновление только текущей версии. Это вызывает проблему, поскольку из-за того, что я вызвал второй прототип операции `CopyToDataTable`, который не устанавливает оригинальных версий записей, вставляемых в базу данных, а также не вызвал метод `AcceptChanges`, то если я попытаюсь обратиться к оригинальным версиям вставленных записей, будет сгенерировано исключение. Мне придется применить метод `HasVersion`, чтобы предотвратить это в случае вставки любой записи. Поскольку я не специфицировал никаких первичных ключей, я знаю, что все записи исходной таблицы будут вставлены в целевую таблицу.

```
Console.WriteLine("{0}After upserting DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
    Console.WriteLine("Student Id = {0} : original {1} : current {2}",
        dataRow.Field<int>("Id"),
        dataRow.HasVersion(DataRowVersion.Original) ?
            dataRow.Field<string>("Name", DataRowVersion.Original) : "-does not exist-",
            dataRow.Field<string>("Name", DataRowVersion.Current));
}
```

В этом сегменте кода я просто отображаю содержимое `DataTable` на консоли. Теперь в этом примере интересно то, что поскольку я не специфицирую первичных ключей для целевой таблицы, то при копировании ни одна запись не будет рассматриваться как ранее существующая, поэтому все скопированные записи из исходного объекта `DataTable` будут добавлены в целевой.

Также обратите внимание, что я обращаюсь к исходной версии данных поля, только если метод `HasVersion` возвращает `true`, указывая на наличие оригинальной версии. Вот результат:

```
Before upserting DataTable:
Student Id = 1 : original Joe Rattz : current Joe Rattz
Student Id = 7 : original Anthony Adams : current Anthony Adams
Student Id = 13 : original Stacy Sinclair : current Stacy Sinclair
Student Id = 72 : original Dignan Stephens : current Dignan Stephens

After upserting DataTable:
Student Id = 1 : original Joe Rattz : current Joe Rattz
Student Id = 7 : original Anthony Adams : current Anthony Adams
Student Id = 13 : original Stacy Sinclair : current Stacy Sinclair
Student Id = 72 : original Dignan Stephens : current Dignan Stephens
Student Id = 1 : original -does not exist- : current Joe Rattz
Student Id = 7 : original -does not exist- : current George Oscar Bluth
Student Id = 13 : original -does not exist- : current Stacy Sinclair
Student Id = 72 : original -does not exist- : current Dignan Stephens
```

Отметьте, что несколько записей теперь дублированы, поскольку я не указал первичных ключей в целевом DataTable. Даже только что обновленная запись теперь встречается дважды в DataTable.

Может вызвать удивление, зачем возиться с вызовом метода HasVersion, если можно было просто вызвать метод AcceptChanges? Вы могли бы сделать это, но в этом случае все текущие значения полей стали бы оригиналными версиями значений, и невозможно было бы узнать, какие записи были изменены. Для этих примеров мне нужны были значения оригинальной версии, отличающиеся от значений текущей версии при изменении записи. Решение проблемы предыдущего примера заключается в спецификации первичных ключей целевого объекта DataTable. В листинге 10.19 представлен пример, аналогичный предыдущему, но на этот раз я специфицировал первичные ключи.

Листинг 10.19. Вызов второго прототипа операции CopyToDataTable, когда первичные ключи установлены

```
Student[] students = {
    new Student { Id = 1, Name = "Joe Rattz" },
    new Student { Id = 7, Name = "Anthony Adams" },
    new Student { Id = 13, Name = "Stacy Sinclair" },
    new Student { Id = 72, Name = "Dignan Stephens" }
};

DataTable dt1 = GetDataTable(students);
DataTable newTable = dt1.AsEnumerable().CopyToDataTable();
newTable.PrimaryKey = new DataColumn[] { newTable.Columns[0] };
Console.WriteLine("Before upserting DataTable:");
foreach (DataRow dataRow in newTable.AsEnumerable())
{
    Console.WriteLine("Student Id = {0} : original {1} : current {2}",
        dataRow.Field<int>"("Id"),
        dataRow.Field<string>"("Name", DataRowVersion.Original),
        dataRow.Field<string>"("Name", DataRowVersion.Current));
}
(from s in dt1.AsEnumerable()
where s.Field<string>"("Name") == "Anthony Adams"
select s).Single<DataRow>().SetField("Name", "George Oscar Bluth");
dt1.AsEnumerable().CopyToDataTable(newTable, LoadOption.Upsert);
Console.WriteLine("{0}After upserting DataTable:", System.Environment.NewLine);
foreach (DataRow dataRow in newTable.AsEnumerable())
{
    Console.WriteLine("Student Id = {0} : original {1} : current {2}",
        dataRow.Field<int>"("Id"),
        dataRow.HasVersion(DataRowVersion.Original) ?
            dataRow.Field<string>"("Name", DataRowVersion.Original) : "-does not exist-",
        dataRow.Field<string>"("Name", DataRowVersion.Current));
}
```

Единственное различие между этим примером и предыдущим в том, что я добавил строку, устанавливающую первичный ключ в DataTable по имени newTable. И вот результат:

```
Before upserting DataTable:
Student Id = 1 : original Joe Rattz : current Joe Rattz
Student Id = 7 : original Anthony Adams : current Anthony Adams
Student Id = 13 : original Stacy Sinclair : current Stacy Sinclair
Student Id = 72 : original Dignan Stephens : current Dignan Stephens
After upserting DataTable:
Student Id = 1 : original Joe Rattz : current Joe Rattz
```

```
Student Id = 7 : original Anthony Adams : current George Oscar Bluth
Student Id = 13 : original Stacy Sinclair : current Stacy Sinclair
Student Id = 72 : original Dignan Stephens : current Dignan Stephens
```

Вот это уже больше похоже на то, что нужно. Обратите внимание, что студент с Id, равным 7, у которого было имя Anthony Adams, теперь имеет имя George Oscar Bluth. Именно этого я и добивался.

Резюме

В этой главе я показал, как следует использовать операции `IEnumerable` для выполнения операций с множествами над объектами `DataRow`, а также как получать и устанавливать значения полей посредством операции `Field<T>` и `SetField<T>`. Кроме того, я продемонстрировал, что может пойти не так, если вы не используете специфичные прототипы операций с множествами для `DataRow`. Комбинируя стандартные операции запросов `LINQ to Objects` со специфичными операциями `DataSet`, можно строить мощные запросы `LINQ` к объектам `DataSet`.

В следующей главе я дополню часть, посвященную `LINQ to DataSet`, рассказом о том, как опрашивать типизированные объекты `DataSet` с помощью `LINQ`, а также приведу пример реального запроса `LINQ to DataSet` к базе данных.

ГЛАВА 11

Дополнительные возможности DataSet

В предыдущей главе я привел многочисленные примеры опроса объектов DataTable, которые в реальных приложениях естественным образом поступают от типичных DataSet. Однако для простоты я создавал объекты DataTable программно, используя объявление статического массива. Однако запросы DataSet — это нечто большее, чем просто создание объектов DataTable из статически объявленных массивов.

К тому же примеры из предыдущей главы выполнялись на не типизированных DataSet. Иногда у вас может возникнуть необходимость выполнения запроса к типизированному DataSet. LINQ to DataSet может и это.

В этой главе я опишу, как решается эта задача, и покажу, как выполняется большая часть LINQ to DataSet. Начнем с обсуждения запросов к типизированным DataSet посредством LINQ to DataSet. Затем, поскольку я упомянул о том, что чаще приходится опрашивать DataSet, чем программно создавать объекты DataTable, будет приведен реальный пример запроса базы данных с LINQ to DataSet.

Необходимые пространства имен

Примеры из этой главы ссылаются на классы из пространств имен System.Data, System.Data.SqlClient и System.Linq. Если следующих директив еще нет в вашем коде, вы должны их добавить:

```
using System.Data;
using System.Data.SqlClient;
using System.Linq;
```

БИЛЛИ ХАРП
Дж. Р. Г.

Типизированные DataSet

Типизированные DataSet могут быть опрошены с использованием LINQ, как это возможно и с не типизированными. Однако типизированные DataSet сделают код ваших запросов LINQ проще и более читабельным. При запросе к типизированному DataSet, поскольку существует класс для DataSet, вы можете обратиться к именам таблиц и столбцов, используя свойства типизированного класса DataSet, вместо индексации коллекции Tables или применения операций Field<T> и SetField<T>.

Поэтому вместо такого обращения к таблице объекта DataSet по имени Students:

```
DataTable Students = dataSet.Tables["Students"];
```

вы можете обратиться так:

```
DataTable Students = dataSet.Students;
```

Вместо такого получения значения поля:

```
dataRow.Field<string>("Name")
```

вы можете получить его следующим образом:

```
dataRow.Name
```

Это без сомнения делает код более читабельным и упрощает его сопровождение.

Прежде чем показать вам пример, мне нужно создать типизированный DataSet. Для этого нужно предпринять описанные ниже шаги.

1. Щелкнуть правой кнопкой мыши на вашем проекте в окне Solution Explorer.
2. Выбрать команду Add⇒New Item (Добавить⇒Новый элемент) в контекстном меню.
3. Развернуть дерево Categories (Категории) в диалоговом окне Add New Item (Добавить новый элемент). Выбрать в дереве узел Data (Данные). Выбрать шаблон DataSet в списке Data Templates (Шаблоны данных). Отредактировать имя файла DataSet, изменив его на StudentsDataSet.xsd, и щелкнуть на кнопке Add (Добавить).
4. После этого вы должны увидеть дизайнер DataSet Designer. Поместите указатель мыши в панель инструментов и перетащите DataTable на поверхность DataSet Designer.
5. Выполнить щелчок правой кнопкой мыши на панели заголовка, добавленном DataTable, и выбрать команду Properties (Свойства) из контекстного меню.
6. Отредактировать Name объекта DataTable, присвоив ему значение Students в окне Properties (Свойства).
7. Снова выполнить щелчок правой кнопкой мыши на DataTable и выбрать команду Add⇒Column (Добавить⇒Столбец) из контекстного меню.
8. Отредактировать Name вновь добавленного объекта в Id и изменить DataType на System.Int32.
9. Снова выполнить щелчок правой кнопкой мыши на DataTable и выбрать команду Add⇒Column из контекстного меню.
10. Отредактировать вновь Name добавленной DataColumn в Name.
11. Сохранить файл.

Таким образом, я создал типизированный DataSet по имени StudentsDataSet. Этот типизированный DataSet содержит DataTable по имени Students с двумя столбцами данных типа DataColumn — один по имени Id типа Int32 и один по имени Name типа string. Я могу использовать этот типизированный DataSet для выполнения запросов LINQ, и поскольку DataSet типизирован, могу обращаться к полям DataRow как к членам класса. Давайте рассмотрим пример.

Имея типизированный DataSet, я могу выполнять на нем запросы LINQ, как показано в листинге 11.1.

Листинг 11.1. Пример запроса к типизированному DataSet

```
StudentsDataSet studentsDataSet = new StudentsDataSet();
studentsDataSet.Students.AddStudentsRow(1, "Joe Rattz");
studentsDataSet.Students.AddStudentsRow(7, "Anthony Adams");
studentsDataSet.Students.AddStudentsRow(13, "Stacy Sinclair");
studentsDataSet.Students.AddStudentsRow(72, "Dignan Stephens");
string name =
    studentsDataSet.Students.Where(student => student.Id == 7).Single().Name;
Console.WriteLine(name);
```

В этом примере я создаю экземпляр объекта `StudentsDataSet` и добавляю четыре записи, каждая из которых была элементом массива в большинстве примеров предыдущей главы. Как правило, в рабочем коде вам не придется этого делать, потому что, скорее всего, вы будете получать эту информацию из базы данных.

Как только типизированный `DataSet` наполнен данными, я выполняю запрос к нему. Обратите внимание, что я обращаюсь к `Students DataTable` как к свойству объекта `StudentsDataSet`. Также заметьте, что в лямбда-выражении операции `Where` я напрямую обращаюсь к свойству `Id` элемента, который должен быть `DataRow` — вместо вызова свойства `Field` этого `DataRow`. Я могу делать это, потому что `DataSet` типизирован. Также отметьте, что когда я получаю единственный объект `DataRow` вызовом операции `Single`, я могу напрямую обратиться к его свойству `Name`, опять-таки, потому, что `DataSet` типизирован. Результат представлен ниже:

Anthony Adams

Разве не здорово? Типизированные `DataSet` делают работу с `DataSet` такой же простой, как работа с нормальными объектами классов и их свойствами.

Собираем все вместе

Я хотел, чтобы примеры из предыдущей главы обеспечили возможность легкого изучения запросов в программном интерфейсе LINQ to DataSet API. Мне хотелось, чтобы время, которое вы потратите на работу с примерами, в основном было потрачено на овладение LINQ. Я не хотел, чтобы вы отвлекались на обращение к базе данных или на обеспечение корректности строки подключения. Но прежде чем мы покинем эту главу, я хотел бы представить более полный пример — такой, который в действительности получает `DataSet` из базы данных, потому что, скорее всего, именно так вы будете получать их в коде реальных приложений.

Нужно признать, что создание примера разумных размеров, который получит данные из базы и использует LINQ to DataSet API, создаст впечатление надуманного. В конце концов, я собираюсь выполнить запрос SQL к данным из базы с помощью ADO.NET, чтобы получить `DataSet`, затем снова выполнить пирамидку и запросить эти данные с помощью LINQ to DataSet, и все это в нескольких строках кода. В реальной жизни кто-нибудь может спросить: почему бы просто не изменить запрос SQL таким образом, чтобы получить все, что вам нужно, с первого раза? Я отвечу ему: дерзайте! Чего я хотел добиться здесь, так это сценария, объясняющего, как избежать глупостей.

В моем сценарии я буду работать с компанией под названием Northwind. Моя компания уже имеет приложение, запрашивающее в базе данных список заказов. Это конкретное приложение выполняет разнообразный анализ того, какие сотрудники продали какие наименования товара каким заказчикам, и в какие страны были осуществлены поставки в соответствии с заказами. Так что приложение уже загружает сотрудников, заказчиков и страны поставок для всех заказов в `DataSet`. Моя задача — выполнить один или более анализов на основе этих уже запрошенных данных. Мне нужно произвести уникальный список каждого сотрудника, который продал что-либо любой компании по всем заказам, поставляемым в Германию.

В данном примере я создаю экземпляр `SqlDataAdapter`, за которым стоит `DataSet`, изываю метод `Fill` этого объекта `SqlDataAdapter` для наполнения `DataSet`. В настоящем сценарии это уже сделано, потому что это делает существующее приложение. Так что готовый объект `DataSet` должен быть передан моему коду. Но поскольку я не имею дела с полноценным приложением, я просто сделаю это в примере. После получения объекта `DataSet` с результатами запроса SQL все, что мне останется — это вы-

полнить запрос LINQ to DataSet и отобразить полученный результат. Код представлен в листинге 11.2.

Листинг 11.2. Собираем все вместе

```

string connectionString =
    @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated
    Security=SSPI;";
SqlDataAdapter dataAdapter = new SqlDataAdapter(
    @"SELECT O.EmployeeID, E.FirstName + ' ' + E.LastName as EmployeeName,
        O.CustomerID, C.CompanyName, O.ShipCountry
    FROM Orders O
    JOIN Employees E on O.EmployeeID = E.EmployeeID
    JOIN Customers C on O.CustomerID = C.CustomerID",
    connectionString);
DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet, "EmpCustShip");
// Весь предшествующий код унаследован.
var ordersQuery = dataSet.Tables["EmpCustShip"].AsEnumerable()
    .Where(r => r.Field<string>("ShipCountry").Equals("Germany"))
    .Distinct(System.Data.DataRowComparer.Default)
    .OrderBy(r => r.Field<string>("EmployeeName"))
    .ThenBy(r => r.Field<string>("CompanyName"));
foreach(var dataRow in ordersQuery)
{
    Console.WriteLine("{0,-20} {1,-20}", dataRow.Field<string>("EmployeeName"),
        dataRow.Field<string>("CompanyName"));
}

```

Как видите, я подключаюсь к базе данных Northwind. При необходимости вы можете настроить строку подключения под существующие условия.

Обратите внимание, что в предыдущем запросе я использовал операции AsEnumerable, Distinct и Field<T>, о которых рассказал в предыдущей главе, а также операции Where, OrderBy и ThenBy из LINQ to Objects API, чтобы составить такой запрос, как мне нужно. Вы должны оценить красоту этого решения — как все это работает вместе. Если запрос будет работать так, как мне надо, то я должен получить список всех сотрудников, осуществлявших продажу по заказу каждой компании, поставки по которому выполнялись в Германию, в алфавитном порядке имен сотрудников и названий компаний, без дублированных строк. Вот результат:

```

Andrew Fuller Die Wandernde Kuh
Andrew Fuller Königlich Essen
Andrew Fuller Lehmanns Marktstand
Andrew Fuller Morgenstern Gesundkost
Andrew Fuller Ottilies Käseladen
Andrew Fuller QUICK-Stop
Andrew Fuller Toms Spezialitäten
Anne Dodsworth Blauer See Delikatessen
Anne Dodsworth Königlich Essen
Anne Dodsworth Lehmanns Marktstand
Anne Dodsworth QUICK-Stop
...
Steven Buchanan Frankenversand
Steven Buchanan Morgenstern Gesundkost
Steven Buchanan QUICK-Stop

```

Обратите внимание, что для каждого сотрудника в левой части строки ни одна компания не повторяется в правой части. Это важно, потому что демонстрирует

необходимость операций работы с множествами LINQ to DataSet API. Для пробы измените вызов операции `Distinct` так, чтобы не был указан компаратор по умолчанию `DataRowComparer.Default`, и вы сразу получите дублированные строки.

Для того чтобы увидеть другой пример использования синтаксиса выражений, в листинге 11.3 представлен тот же пример, но с вышеупомянутым синтаксисом.

Листинг 11.3. Собираем все вместе с синтаксисом выражений запросов

```

string connectionString =
    @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";
SqlDataAdapter dataAdapter = new SqlDataAdapter(
    @"SELECT O.EmployeeID, E.FirstName + ' ' + E.LastName as EmployeeName,
        O.CustomerID, C.CompanyName, O.ShipCountry
    FROM Orders O
    JOIN Employees E on O.EmployeeID = E.EmployeeID
    JOIN Customers C on O.CustomerID = C.CustomerID",
    connectionString);
DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet, "EmpCustShip");
// Весь предшествующий код унаследован.
var ordersQuery = (from r in dataSet.Tables["EmpCustShip"].AsEnumerable()
                    where r.Field<string>("ShipCountry").Equals("Germany")
                    orderby r.Field<string>("EmployeeName"),
                            r.Field<string>("CompanyName")
                    select r)
    .Distinct(System.Data.DataRowComparer.Default);
foreach (var dataRow in ordersQuery)
{
    Console.WriteLine("{0,-20} {1,-20}", dataRow.Field<string>("EmployeeName"),
        dataRow.Field<string>("CompanyName"));
}

```

Теперь запрос использует синтаксис выражения запроса. Хотя моей целью было сделать запрос функционально эквивалентным предыдущему, я не мог этого добиться. Обратите внимание, что операция `Distinct` вызывается теперь в самом конце запроса. Напомню, что компилятор не может транслировать все операции запроса, специфицированного в синтаксисе выражений запросов, а только наиболее часто используемые. В этом случае он не знает, как транслировать операцию `Distinct`. Из-за этого я не могу сделать этот вызов в синтаксисе выражений запросов частью запроса. Как видите, я вызвал его в конце запроса. И полученный результат будет тем же, что и раньше.

Однако между запросами в листингах 11.3 и 11.2 существует разница в производительности. В листинге 11.2 операция `Distinct` вызывается сразу после операции `Where`, поэтому дублированные записи исключаются из результирующего набора до его упорядочивания. В листинге 11.3 операция `Distinct` не вызывается до самого конца, так что дублированные записи присутствуют на этапе упорядочивания результата. Это значит, что сначала записи будут отсортированы, а затем исключены при вызове операции `Distinct`. Это излишняя работа, однако, неизбежная, если вы предпочитаете применить синтаксис выражений запросов.

Резюме

Как сказано в этой главе, вы можете опрашивать не только обычные `DataSet` посредством LINQ to `DataSet`, но также и типизированные `DataSet`. Типизированные `DataSet` облегчают сопровождение вашего кода, повышая его читабельность, а LINQ to `DataSet`

позволяет очень легко и просто выполнять запросы к ним. Также я продемонстрировал более реалистичный запрос LINQ to DataSet, обращающийся к базе данных Northwind.

Программный интерфейс LINQ to DataSet API открывает доступ LINQ к еще одной предметной области. Учитывая наличие большого объема кода, использующего DataSet, LINQ to DataSet имеет широкое поле приложения, поскольку позволяет легко модифицировать унаследованный код .NET, тем как никогда облегчая запросы данных из DataSet.

Одно преимущество, которое имеется у LINQ to DataSet API перед LINQ to SQL API, заключается в том, для выполнения запросов LINQ to DataSet не требуется генерации и предварительной компиляции какого-то кода классов, связанных с базой данных. Это делает LINQ to DataSet более динамичным и в большей степени подходящим для построения утилит работы с базами, когда конкретная база данных неизвестна до момента выполнения.

Благодаря применению операции AsEnumerable для создания последовательностей из объектов DataTable, стало возможным применение стандартных операций запросов LINQ to Objects, что добавило мощи к арсеналу возможностей запросов.

В LINQ to DataSet были добавлены операции для работы с ключевыми классами DataSet: DataTable, DataRow и DataColumn. Следует не забывать о проблеме, которая вызвала необходимость в новых, ориентированных на множества, прототипах операций Distinct, Union, Intersect, Except и SequenceEqual, а именно — проблеме сравнения DataRow на эквивалентность. Поэтому при работе с объектами DataSet, DataTable и DataRow всегда предпочтите применение ориентированных на множества прототипов операций Distinct, Union, Intersect, Except и SequenceEqual, когда специфицирован объект проверки эквивалентности, а не тех версий прототипов, которые не предусматривают указание такого объекта.

И, наконец, при получении значения столбца используйте операции Field<T> и SetField<T> для исключения проблем сравнения со значениями null.

Одно обстоятельство стало очевидным для меня при работе с интерфейсом LINQ to DataSet API. Я совершенно недооценивал мощь и удобство DataSet. Эти объекты предоставляют широкие возможности как кэшированные хранилища реляционных данных. И хотя их возможности поиска ограничены, благодаря LINQ to DataSet API все ограничения устраняются. Теперь вы можете использовать LINQ для опроса данных из DataSet, что значительно облегчает разработку кода.



ЧАСТЬ V

LINQ to SQL

В этой части...

Глава 12. Введение в LINQ to SQL

Глава 13. Советы и инструменты, связанные с LINQ to SQL

Глава 14. Операции для баз данных в LINQ to SQL

Глава 15. Сущностные классы LINQ to SQL

Глава 16. `DataContext`

Глава 17. Конфликты параллельного доступа

Глава 18. Дополнительные возможности SQL

ГЛАВА 12

Введение в LINQ to SQL

Листинг 12.1. Простой пример обновления столбца ContactName таблицы Customer из базы данных Northwind

```
// Создать DataContext.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
// Извлечь заказчика LAZYK.
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
// Обновить контактное имя.
cust.ContactName = "Ned Plimpton";
try
{
    // Сохранить изменения.
    db.SubmitChanges();
}
// Обнаружить конфликты совместного доступа.
catch (ChangeConflictException)
{
    // Разрешить конфликты.
    db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
}
```

На заметку! Этот пример требует генерации сущностных классов, о чем я расскажу далее в этой главе.

В листинге 12.1 я использовал LINQ to SQL для запроса записи, чье поле CustomerID имеет значение "LAZYK", из таблицы Customers базы данных Northwind, и возврата объекта Customer, представляющего эту запись. Затем я обновил свойство ContactName объекта Customer и сохранил изменение в базе данных Northwind посредством вызова метода SubmitChanges. Здесь не так много кода, но он вдобавок обнаруживает конфликты параллельного доступа и разрешает их в случае возникновения.

Нажав <Ctrl+F5>, запустите код из листинга 12.1. Здесь нет консольного вывода, но если вы проверите вашу базу данных, то обнаружите, что ContactName для заказчика LAZYK теперь содержит значение "Ned Plimpton".

На заметку! Этот пример вносит изменения в базу данных, без последующего восстановления.

Исходное значение ContactName для заказчика LAZYK — "John Steel". Вы должны вернуть ему это значение, чтобы правильно работали последующие примеры. Это можно сделать вручную, или просто изменив пример кода так, чтобы он восстанавливал старое значение, и запустив пример снова. Эта книга использует расширенную версию базы данных Northwind. За подробностями обращайтесь в раздел этой главы, озаглавленный "Получение соответствующей версии базы данных Northwind".

Введение в LINQ to SQL

К данному моменту я уже говорил об использовании LINQ с находящимися в памяти коллекциями данных и массивами, XML и DataSet. Теперь я перейду к тому, что многим кажется наиболее веской причиной для использования LINQ — LINQ to SQL. Я так думаю потому, что когда я заглянул в форум MSDN, посвященный LINQ, то обнаружил, что большая часть дискуссий там посвящена LINQ to SQL. Думаю, многие разработчики недооценивают важность LINQ как языка запросов общего назначения и всего многообразия способов его применения. Надеюсь, я убедил вас в этом на протяжении предыдущих глав.

LINQ to SQL — это программный интерфейс (API) для работы с базами данных SQL Server. В современном мире, где господствуют объектно-ориентированные языки программирования, существует несоответствие между языком программирования и реляционной базой данных. При написании приложения мы моделируем классы как представления объектов реального мира, таких как заказчики, счета, политики и полеты. Нам нужен способ сохранения этих объектов, чтобы при перезапуске приложения все эти объекты с их данными не были потеряны. Однако большинство баз данных промышленного масштаба остаются реляционными и хранят свою информацию в виде записей в таблицах, а не в виде объектов. Пользовательский класс может содержать несколько адресов и номеров телефонов, хранящихся в коллекциях, представленных в виде дочерних свойств класса заказчика; при сохранении такая информация, скорее всего, будет храниться во множестве таблиц, таких как таблица заказчиков, таблица адресов и таблица телефонов.

Вдобавок типы данных, поддерживаемые языком приложения, отличаются от типов базы данных. Разработчикам приходится писать собственный код, который знает, как загрузить объект заказчика из всех соответствующих таблиц, а также как сохранить такой объект во все таблицы, обрабатывая необходимые преобразования данных между языком приложения и базой данных. Это утомительный, и часто подверженный ошибкам процесс. Наличие этих проблем объектно-реляционного отображения (object-relational mapping — ORM), часто называемых объектно-реляционной потерей соответствия (object-relational impedance mismatch), в течение ряда лет привело к появлению широкого разнообразия готовых программных ORM-решений. LINQ to SQL — это реализация ORM начального уровня от Microsoft на основе LINQ, предназначенная для работы с SQL Server.

Обратите внимание, что я сказал "для работы с SQL Server". LINQ to SQL предназначен исключительно для SQL Server. Однако можно надеяться, что другие поставщики баз данных уже работают либо собираются работать над собственными реализациями программных интерфейсов LINQ. Лично я хотел бы увидеть LINQ to DB2 API, и я уверен, что многие другие ждут появления LINQ to Oracle, LINQ to MySQL, LINQ to Sybase и т.п.

На заметку! LINQ to SQL работает только с SQL Server или SQL Express. Чтобы использовать LINQ с другими базами данных, потребуются дополнительные программные интерфейсы LINQ API, которые должны быть разработаны поставщиками баз данных. До тех пор в качестве альтернативы рассмотрите применение LINQ to DataSet.

Вы, вероятно, заметили, что я сказал, что LINQ to SQL — реализация ORM начального уровня. Если вы найдете ее недостаточно мощной или гибкой, чтобы отвечать вашим требованиям, попробуйте исследовать LINQ to Entities. Я не стану описывать LINQ to Entities в этой книге, но замечу, что этот интерфейс обещает быть более мощным и гибким, чем LINQ to SQL. Однако имейте в виду, что повышенная мощность влечет за себя дополнительную сложность. К тому же LINQ to Entities — пока еще не столь зрелый интерфейс, как LINQ to SQL.

Большинство инструментов ORM пытаются абстрагировать физическую базу данных в виде бизнес-объектов. С такой абстракцией обычно теряется возможность выполнения запросов SQL, составляющих значительную часть абстракции реляционных баз данных. Именно это отличает LINQ to SQL от большинства его аналогов. Мы не только получаем удобство в виде бизнес-объектов, которые отображаются на базу данных, но также получаем полноценный язык запросов, подобный хорошо знакомому SQL.

Совет. LINQ to SQL — инструмент ORM начального уровня, позволяющий выполнять мощные SQL-запросы.

В дополнение к предоставлению возможностей запросов LINQ, до те пор, пока ваш запрос возвращает сущностные объекты LINQ to SQL, в противоположность одиночным полям, именованным не сущностным классам или анонимным классам, LINQ to SQL также предлагает средства отслеживания изменений и обновлений базы данных, построенных на основе оптимистической модели обнаружения и разрешения конфликтов параллельного доступа, а также транзакционной целостности.

В листинге 12.1 я сначала должен был создать экземпляр класса Northwind. Этот класс наследуется от класса `DataContext`, который я рассмотрю более подробно в главе 16. А пока воспринимайте его как “усовершенствованное” соединение с базой данных. Он также обрабатывает обновление базы данных, как вы сможете увидеть позднее, когда я вызову на нем метод `SubmitChanges`. Далее я извлекаю одного заказчика из базы данных Northwind в объект `Customer`. Этот объект `Customer` является экземпляром сущностного класса `Customer`, который может быть либо написан вручную, либо сгенерирован. В данном случае класс `Customer` был сгенерирован для меня утилитой `SQLMetal`, как и класс `Northwind`. После извлечения заказчика я обновил одно из свойств объекта `Customer` по имени `ContactName` и вызвал метод `SubmitChanges` для сохранения модифицированного имени контакта в базе данных. Обратите внимание, что я окружил этот вызов блоком `try/catch`, и специально перехватываю исключение `ChangeConflictException`. Это нужно для обработки конфликтов параллельного доступа, которые я детально опишу в главе 17.

Прежде чем вы сможете запустить этот или любой другой пример из этой главы, вам понадобится создать сущностные классы для базы данных `Northwind`. Руководство по созданию необходимых сущностных классов читайте в разделе “Предварительные условия для запуска примеров” настоящей главы.

LINQ to SQL — сложная тема, и любые примеры требуют участия многих элементов LINQ to SQL. В первом примере в начале этой главы я использовал класс, производный от `DataContext`, каковым является класс `Northwind`; кроме того, я использовал сущностный класс — `Customer`, обнаружение и разрешение конфликтов параллельного доступа, а также обновление базы данных через метод `SubmitChanges`. Я не могу одновременно объяснить все эти концепции. Поэтому я должен предоставить вам какой-то фундамент для понимания каждого из перечисленных компонентов, прежде чем начать, чтобы вы имели базовое понимание основ LINQ to SQL. Не сомневайтесь, что я раскрою все эти концепции во всех утомительных деталях позднее, в последующих главах, посвященных LINQ to SQL.

DataContext

`DataContext` — это класс, устанавливающий соединение с базой данных. Он также предоставляет несколько служб, обеспечивающих отслеживание идентичности, отслеживание изменений и обработку этих изменений. Каждую из этих служб я опишу более детально в главе 16. А пока просто знайте, что `DataContext` — это класс, соединяющий с базой данных, отслеживающий то, что мы изменяем, и обновляющий базу данных при вызове метода `SubmitChanges`.

В LINQ to SQL принято использовать класс, производный от `DataContext`. Имя производного класса обычно совпадает с именем базы данных, на которую он отображается. Я часто буду ссылаться на этот производный класс в главах о LINQ to SQL, как на `[Your]DataContext`, поскольку его имя зависит от базы данных, для которой он создан.

В моих примерах производный от `DataContext` класс будет называться `Northwind`, потому что он был сгенерирован инструментом командной строки `SQLMetal`, а он имеет сгенерированные классы-наследники `DataContext` по имени базы данных, для которой он сгенерирован.

Этот производный от `DataContext` класс, `[Your]DataContext`, обычно будет иметь общедоступное свойство `Table<T>` для каждой таблицы базы данных, которую вы отображаете на базу данных, где `T` — тип сущностного класса, экземпляр которого создается для каждой извлеченной записи из конкретной таблицы базы данных. Тип данных `Table<T>` представляет собой специализированную коллекцию. Например, поскольку в базе `Northwind` присутствует таблица `Customers`, мой класс `Northwind`, унаследованный от класса `DataContext`, будет иметь `Table<Customers>` по имени `Customers`. Это значит, что я могу обращаться к записям в таблице `Customers` базы данных, непосредственно обращаясь к свойству `Customers` типа `Table<Customers>` в моем классе `Northwind`. Пример этого вы можете видеть в первом примере настоящей главы — листинге 12.1, где я кодирую это как `db.Customers`. Этот код запрашивает запись в таблице `Customers` базы данных `Northwind`.

Сущностные классы

LINQ to SQL подразумевает использование сущностных классов, причем каждый из них обычно отображается на единственную таблицу базы данных. Однако, используя наследуемое отображение сущностного класса, при определенных условиях можно отобразить целую иерархию классов на единственную таблицу. Подробнее об этом вы можете прочесть в главе 18. Таким образом, мы имеем сущностные классы, отображенные на таблицы базы данных, и свойства сущностных классов, отображенные на столбцы этих таблиц. Это отображение “класс-таблица” и “свойство-столбец” являются сутью LINQ to SQL.

На заметку! Суть LINQ to SQL заключается в отображении сущностных классов на таблицы базы данных и свойств сущностных классов на столбцы таблиц базы данных.

Такое отображение может возникать непосредственно в исходных файлах класса посредством оснащения их соответствующими атрибутами, или же может быть специфицировано во внешнем XML-файле отображения. При использовании такого внешнего файла специфичная для LINQ to SQL информация может храниться отдельно от исходного кода. Это может быть очень удобно, если у вас нет исходного кода, либо если вы хотите хранить код отдельно от LINQ to SQL. Для большинства примеров в главах об LINQ to SQL я использую сущностные классы, сгенерированные инструментом команд-

ной строки SQLMetal. SQLMetal генерирует сущностные классы с информацией об отражении LINQ to SQL, встроенной непосредственно в генерируемые исходные модули. Эта информация представлена в форме атрибутов и их свойств.

Вы сможете обнаружить наличие сущностных классов в моих примерах, когда увидите классы или объекты, имеющие форму единственного числа имени таблицы базы данных Northwind. Например, в листинге 12.1 я использую класс по имени `Customer`. Поскольку `Customer` — форма единственного числа от `Customers`, а в базе данных Northwind есть таблица по имени `Customers`, это указывает вам на то, что `Customer` — сущностный класс для таблицы `Customers` из базы данных Northwind.

Инструмент командной строки SQLMetal имеет опцию `/pluralize`, которая вызывает именование сущностного класса в форме единственного числа имени таблицы базы данных. Если бы я не специфицировал опцию `/pluralize` при генерации сущностных классов, то мой класс назывался бы `Customers` вместо `Customer`, потому что таково имя таблицы — `Customers`. Я упомянул об этом, чтобы вы не путались, читая другие источники по LINQ to SQL. В зависимости от того, как автор запускал SQLMetal и какие опции при этом указывал, сущностный класс может быть назван во множественном или единственном числе.

Ассоциации

Ассоциация — это термин, используемый для назначения первичного ключа для отношений внешнего ключа между двумя сущностными классами. В отношении “один к многим” результатом ассоциации является то, что родительский класс — тот, что содержит первичный ключ — включает коллекцию дочерних классов, т.е. классов, имеющих внешний ключ. Эта коллекция сохраняется в приватной переменной-члене типа `EntitySet<T>`, где `T` будет типом дочернего сущностного класса.

Например, в сущностном классе `Customer`, сгенерированном инструментом командной строки SQLMetal для базы данных Northwind, есть приватный член типа `EntitySet<Order>`, названный `_Orders`, который содержит все объекты `Order` для определенного объекта `Customer`:

```
private EntitySet<Order> _Orders;
```

SQLMetal также сгенерировал общедоступное свойство по имени `Orders`, используемое для доступа к приватной коллекции `_Orders`.

На другом конце отношения — в классе, содержащем внешний ключ, — находится ссылка на родительский класс, поскольку это отношение “многие к одному”. Эта ссылка сохраняется в приватной переменной-члене типа `EntityRef<T>`, где `T` — родительский класс (здесь термин “родительский” относится к ассоциации, а не наследованию).

В сгенерированных сущностных классах Northwind сущностный класс `Order` содержит приватную переменную-член типа `EntityRef<Customer>` по имени `_Customer`:

```
private EntityRef<Customer> _Customer;
```

И снова SQLMetal сгенерировал общедоступное свойство по имени `Customer` для доступа к ссылке на родителя.

Ассоциация, первичные и внешние ключи, а также направление отношения задаются атрибутами и свойствами атрибутов в исходном модуле сгенерированных сущностных классов.

Преимущество ассоциации состоит в возможности доступа к дочерним объектам родительских и, таким образом, к записям базы данных, с той же легкостью, как к любому свойству родительского объекта. Аналогично доступ к родительскому объекту из дочернего заключается в обращении к соответствующему свойству дочернего объекта.

Обнаружение конфликтов параллельного доступа

Одной из ценных служб, предоставляемых вам `DataContext`, является обработка изменений. Когда вы пытаетесь обновить базу данных вызовом метода `SubmitChanges` класса `DataContext`, он автоматически выполняет оптимистическое обнаружение конфликтов параллелизма.

Если конфликт обнаружен, генерируется исключение `ChangeConflictException`. Всякий раз, когда вы вызываете метод `SubmitChanges`, то должны окружать его блоком `try/catch` и перехватывать исключение `ChangeConflictException`. Это — правильный способ обнаружения конфликтов параллельного доступа.

Пример этого вы могли видеть в листинге 12.1. Подробное описание деталей обнаружения и разрешения конфликтов параллелизма приведено в главе 17. Многие примеры в этой и последующих главах, посвященных LINQ to SQL, не выполняют обнаружения конфликтов параллельного доступа или их разрешения; я поступил так только в целях краткости и ясности. В вашем реальном коде вы всегда должны это делать.

Разрешение конфликтов параллельного доступа

Как только конфликт параллелизма обнаружен, следующий шаг — его разрешение. Это можно сделать несколькими способами. В листинге 12.1 я пошел простейшим путем, вызвав метод `ResolveAll` коллекции `ChangeConflicts` унаследованного класса `DataContext`, когда перехвачено исключение `ChangeConflictException`.

Опять-таки, во многие примеры, приведенные в главах по LINQ to SQL, я не включаю кода для обнаружения и разрешения конфликтов, но вы всегда должны это делать в вашем реальном коде.

Как я упоминал в предыдущем разделе, подробности разрешения конфликтов параллелизма будут детально описаны в главе 17.

Предварительные условия для запуска примеров

Поскольку почти все примеры в этой и последующих главах, посвященных LINQ to SQL, используют расширенную базу данных примеров Northwind, нам понадобятся сущностные классы и файлы отображения для базы данных Northwind.

Получение соответствующей версии базы данных Northwind

К сожалению, в стандартной базе данных Northwind недостает нескольких вещей, которые понадобятся мне для полноценной демонстрации LINQ to SQL, таких как табличных и скалярных функций. Поэтому вместо использования стандартной базы данных Northwind я применяю расширенную версию, поставляемую Microsoft для демонстрации возможностей LINQ.

Вы можете получить расширенную версию базы Northwind на Web-странице, посвященной этой книге, на сайте издательства:

<http://www.williamspublishing.com>

Также она доступна на Web-сайте LINQDev.com. Загляните в раздел, озаглавленный "Obtain the Northwind Database":

<http://www.linqdev.com>

При загрузке с LINQDev.com удостоверьтесь, что загружаете именно расширенную версию.

Чтобы запустить все примеры из глав, посвященных LINQ to SQL, вам понадобится именно расширенная версия базы данных Northwind.

Генерация сущностных классов Northwind

Поскольку я еще не рассказал в деталях, как генерируют сущностные классы, мне придется вкратце описать этот процесс, не вдаваясь в подробности. Однако детали будут представлены в главе 13.

Чтобы сгенерировать сущностные классы, у вас должна быть расширенная версия базы данных Northwind, как я говорил в предыдущем разделе.

Откройте окно командной строки Visual Studio. Чтобы сделать это, загляните в меню Microsoft Visual Studio 2008 и его подменю по имени Visual Studio Tools (Инструменты Visual Studio), найдите там команду под названием Visual Studio 2008 Command Prompt (Командная строка Visual Studio 2008) и выберите ее. Как только откроется окно командной строки, смените текущий каталог на тот, в котором вы собираетесь создать сущностные классы и внешние файлы отображения. Я перешел в корневой каталог моего диска C:

```
cd \
```

Если вы собираетесь генерировать ваши сущностные классы, используя файлы базы данных Northwind, не подключая к ним базу данных, используйте следующую команду:

```
sqlmetal /namespace:nwind /code:Northwind.cs /pluralize /functions /sprocs /views
<path to Northwind MDF file>
```

Внимание! Уделите особое внимание имени файла MDF и регистру его символов, указывая его в командной строке. Имя и регистр генерируемого производного от DataContext класса будет соответствовать имени файла, переданному в командной строке, а не физическому имени самого файла. Если вы отступите от точного написания имени класса Northwind, то ни один пример не станет работать без модификации. Поэтому важно передать имя файла базы данных, как [путь]\Northwind.mdf, а не northwind.mdf, NorthWind.mdf или как-нибудь еще.

Чтобы создать сущностные классы из файла по имени Northwind.mdf, введите следующую команду:

```
sqlmetal /namespace:nwind /code:Northwind.cs /pluralize /functions /sprocs /views
"C:\Northwind.mdf"
```

Запуск этой команды вызовет создание модуля сущностного класса по имени Northwind.cs в вашем текущем каталоге.

Если вы собираетесь генерировать ваши сущностные классы из базы данных Northwind, которая уже подключена к SQL Server, используйте следующую команду:

```
sqlmetal /server:<server> /user:<user> /password:<password> /database:Northwind
/namespace:nwind /code:Northwind.cs /pluralize /functions /sprocs /views
```

Чтобы создать сущностные классы из подключенной базы данных по имени Northwind, введите следующую команду:

```
sqlmetal /server:.\\SQLEXPRESS /database:Northwind /namespace:nwind
/code:Northwind.cs /pluralize /functions /sprocs /views
```

На заметку! В зависимости от окружения вам может понадобиться специфицировать пользователя опцией /user: [имя_пользователя] и пароль — опцией /password: [пароль] в командной строке из предыдущего примера. За подробностями обращайтесь к разделу "SQLMetal" главы 13.

Команда, введенная с использованием любого из этих подходов, сообщает SQLMetal о том, что он должен сгенерировать файл исходного кода по имени Northwind.cs в текущем каталоге. Я опишу все опции этой программы в следующей главе. Скопируйте сгенерированный файл Northwind.cs в проект, добавив его в качестве существующего элемента.

Вы можете теперь применять LINQ to SQL на базе данных Northwind, используя сущностные классы, которые содержатся в файле Northwind.cs.

Совет. Будьте осторожны, внося изменения в исходный файл сгенерированных сущностных классов. Позднее у вас может возникнуть необходимость в их перегенерации, что вызовет потерю ваших изменений. Вы можете добавить бизнес-логику, добавив методы к сущностным классам. Вместо модификации сгенерированного файла рассмотрите возможность воспользоваться преимуществами частичных классов C# 2.0, чтобы хранить добавленные свойства и методы в отдельном исходном модуле.

Генерация XML-файла отображения Northwind

Также мне понадобится сгенерировать файл отображения (mapping file), который будет использован некоторыми примерами. Опять же, для этого я применяю SQLMetal. В той же командной строке и с тем же текущим каталогом выполните следующую команду:

```
sqlmetal /map:northwindmap.xml "C:\Northwind.mdf" /pluralize /functions /sprocs
/views /namespace:nwind
```

Обратите особое внимание на регистр символов, указывая MDF-файл. Это генерирует файл по имени northwindmap.xml в текущем каталоге.

На заметку! Эта команда выдаст код на экран наряду с генерацией XML-файла отображения, так что не удивляйтесь, увидев код на экране.

Использование LINQ to SQL API

Для того чтобы использовать LINQ to SQL API, вам нужно добавить сборку System.Data.Linq.dll к вашему проекту, если вы не сделали этого ранее. Также нужно добавить к вашему исходному коду директивы using для пространств имен System.Linq и System.Data.Linq:

```
using System.Data.Linq;
using System.Linq;
```

В добавок для примеров вам нужно будет добавить директиву using для пространства имен, в котором находятся сгенерированные сущностные классы:

```
using nwind;
```

IQueryable<T>

Вы увидите, что во многих примерах LINQ to SQL в этой главе и последующих главах LINQ to SQL я работаю с последовательностями типа IQueryable<T>, где T — тип сущностного класса.

Это тип последовательностей, которые обычно возвращаются запросами LINQ to SQL. Они часто работают подобно последовательности IEnumerable<T>, и это не случайно.

Интерфейс `IQueryable<T>` реализует интерфейс `IEnumerable<T>`. Вот его определение:

```
interface IQueryable<T> : IEnumerable<T>, IQueryable
```

Благодаря такому наследованию, вы можете трактовать последовательность `IQueryable<T>`, как последовательность `IEnumerable<T>`.

Некоторые общие методы

Как вы вскоре убедитесь, многие примеры из глав LINQ to SQL очень быстро усложняются. Демонстрация конфликтов параллельного доступа требует внесения изменений в базу данных, внешне по отношению к LINQ to SQL. Иногда приходится извлекать данные извне LINQ to SQL. Чтобы подчеркнуть код LINQ to SQL и исключить как можно больше тривиальных деталей, в то же время сохраняя реальную работоспособность примеров, я создал некоторые общие методы, предназначенные для использования во многих примерах.

Не забудьте добавить эти общие методы в исходные модули при тестировании примеров из глав, посвященных LINQ to SQL.

`GetStringFromDb()`

Общий метод, который не раз пригодится — это метод для получения простой строки из базы данных с использованием стандартного ADO.NET. Это позволит просматривать то, что в действительности имеется в базе данных, чтобы сравнить с тем, что показывает LINQ to SQL.

`GetStringFromDb(): метод для извлечения строки с использованием ADO.NET`

```
static private string GetStringFromDb(
    System.Data.SqlClient.SqlConnection sqlConnection, string sqlQuery)
{
    if (sqlConnection.State != System.Data.ConnectionState.Open)
    {
        sqlConnection.Open();
    }
    System.Data.SqlClient.SqlCommand sqlCommand =
        new System.Data.SqlClient.SqlCommand(sqlQuery, sqlConnection);
    System.Data.SqlClient.SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
    string result = null;
    try
    {
        if (!sqlDataReader.Read())
        {
            throw (new Exception(
                String.Format("Unexpected exception executing query [{0}].", sqlQuery)));
        }
        else
        {
            if (!sqlDataReader.IsDBNull(0))
            {
                result = sqlDataReader.GetString(0);
            }
        }
    }
    finally
    {
```

```

// Всегда вызывать Close по окончании чтения.
sqlDataReader.Close();
}
return (result);
}

```

При вызове метода `GetStringFromDb` ему передается `SqlConnection` и строка, содержащая запрос SQL. Метод проверяет, открыто ли соединение, и если нет, то открывает его.

После этого создается `SqlCommand` передачей конструктору запроса и соединения. Затем от вызова метода `ExecuteReader` класса `SqlCommand` получается `SqlDataReader`. Экземпляр `SqlDataReader` выполняет чтение вызовом метода `Read`, и если данные прочитаны и возвращенное значение первого столбца не равно `null`, это значение извлекается методом `GetString`.

И, наконец, `SqlDataReader` закрывается, и значение первого столбца возвращается вызвавшему методу.

ExecuteStatementInDb()

Иногда мне понадобится выполнять операторы SQL, не являющиеся запросами, вроде вставки, обновления и удаления в ADO.NET, чтобы модифицировать состояние базы данных вне LINQ to SQL. Для этой цели я создал метод `ExecuteStatementInDb()`.

ExecuteStatementInDb: метод для выполнения операторов вставки, обновления и удаления в ADO.NET

```

static private void ExecuteStatementInDb(string cmd)
{
    string connection =
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;";
    System.Data.SqlClient.SqlConnection sqlConn =
        new System.Data.SqlClient.SqlConnection(connection);
    System.Data.SqlClient.SqlCommand sqlComm =
        new System.Data.SqlClient.SqlCommand(cmd);

    sqlComm.Connection = sqlConn;
    try
    {
        sqlConn.Open();
        Console.WriteLine("Executing SQL statement against database with ADO.NET ...");
        sqlComm.ExecuteNonQuery();
        Console.WriteLine("Database updated.");
    }
    finally
    {
        // Закрыть соединение.
        sqlComm.Connection.Close();
    }
}

```

При вызове метода `ExecuteStatementInDb` ему передается строка, содержащая команду SQL. Создается `SqlConnection`, за которым следует `SqlCommand`. Экземпляр `SqlConnection` присваивается `SqlCommand`. Затем `SqlConnection` открывается и команда SQL выполняется вызовом метода `ExecuteNonQuery` объекта `SqlCommand`. И, наконец, `SqlConnection` закрывается.

Резюме

В этой главе я представил LINQ to SQL и его наиболее базовые понятия, такие как объекты `DataContext`, сущностные классы, ассоциации, а также обнаружение и разрешение конфликтов параллельного доступа.

Также я показал, как генерировать сущностные классы и внешние файлы отображения для расширенной базы данных Northwind. Эти сущностные классы будут интенсивно использоваться во всех примерах LINQ to SQL.

И, наконец, я привел пару общих методов, на которые полагаются многие из примеров в последующих главах, посвященных LINQ to SQL.

Следующий шаг состоит в том, чтобы вооружить вас некоторыми советами и показать, как использовать необходимые инструменты для применения LINQ to SQL, и именно этому будет посвящена следующая глава.

ГЛАВА 13

Советы и инструменты, связанные с LINQ to SQL

В предыдущей главе я представил программный интерфейс LINQ to SQL вместе с большей частью его терминологии. Я показал, как генерировать сущностные классы, которые потребуются большинству примеров в главах, посвященных LINQ to SQL. Также я предоставил некоторые общие методы, которые будут использованы в примерах этих глав.

В этой главе я предложу ряд примеров, которые, как я надеюсь, вы сочтете полезными для работы с LINQ to SQL. Также я опишу некоторые инструменты, которые делают использование LINQ to SQL столь приятным.

Введение

Сейчас самое время напомнить вам, что прежде чем запускать примеры этой главы, вы должны удовлетворить некоторым предварительным условиям. Первым делом, вы должны иметь расширенную базу данных Northwind и уже сгенерированные сущностные классы для нее. Вернитесь к разделу "Предварительные условия для запуска примеров" главы 12, чтобы убедиться, что у вас есть соответствующая база данных и сгенерированные для нее сущностные классы.

В этой главе, поскольку я продемонстрирую код, использующий сущностные классы, сгенерированные как SQLMetal, так и Object Relational Designer, я не стану специфицировать в примерах директиву `using` для пространства имен `nwind`. Вместо этого я явно укажу пространство имен, когда это будет необходимо для классов `nwind`. Это необходимо в настоящей главе, чтобы контролировать, на какой сущностный класс `Customer` выполняется ссылка в каждом конкретном примере. Поскольку по умолчанию Object Relational Designer генерирует пространство имен такое же, как и ваш проект, и поскольку примеры уже будут существовать в пространстве имен вашего проекта, вам не нужно будет специфицировать пространство имен для сгенерированных дизайнером сущностных классов, но это понадобится для сущностных классов, сгенерированных SQLMetal.

На заметку! В отличие от большинства глав, посвященных LINQ to SQL, в примерах данной главы не указывайте директиву `using` для пространства имен `nwind`.

Советы

Приступая к главам по LINQ to SQL и придерживаясь моего стиля, я начну с "фальстарта" и предоставлю вам некоторые советы, содержащие информацию, которую нужно обсудить. Если этот раздел принесет вам какую-то пользу, значит, я старался не зря! Я хотел бы, чтобы вы ознакомились с этими советами до того, как они понадобятся вам, а не после того, как вы набьете себе шишек в процессе самостоятельной работы.

Используйте свойство `DataContext.Log`

Теперь самое время напомнить о некоторых специфичных для LINQ to SQL советах, которые я приводил в главе 1. Один из них был озаглавлен "Использование Log из `DataContext`", и в нем говорилось об использовании свойства `Log` объекта `DataContext` для отображения транслированного запроса SQL. Это может быть очень полезно не только в целях отладки, но и для анализа производительности. Вы можете обнаружить, что ваши запросы LINQ to SQL транслируются в очень неэффективные запросы SQL. Или же вы можете обнаружить, что из-за отложенной загрузки ассоциированных сущностных классов вам приходится выполнять больше SQL-запросов, чем необходимо. Свойство `DataContext.Log` предоставит вам такую информацию.

Чтобы использовать преимущества этого средства, вы просто присваиваете свойству `DataContext.Log` объект `System.IO.TextWriter`, такой как `Console.Out`.

Пример приведен в листинге 13.1.

Листинг 13.1. Пример использования свойства `DataContext.Log`

```
nwind.Northwind db =
    new nwind.Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var custs = from c in db.Customers
            where c.Region == "WA"
            select new { Id = c.CustomerID, Name = c.ContactName };
foreach (var cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.Id, cust.Name);
}
```

Поскольку в этой главе я буду демонстрировать сущностные классы, сгенерированные и SQLMetal, и Object Relational Designer, у меня будет два класса `Customer`. Как я упоминал ранее, директиву `using` я не включаю в примеры, так что такие сущностные классы, как `Customer`, не будут неоднозначными. Поэтому я должен специфицировать пространство имён `nwind` для класса `Northwind` в листинге 13.1, поскольку использую в этом примере сущностный класс, сгенерированный SQLMetal.

Как видите, в листинге 13.1 я просто присваиваю `Console.Out` свойству `Log` моего объекта `DataContext` по имени `Northwind`. Вот результат работы примера из листинга 13.1:

```
SELECT [t0].[CustomerID], [t0].[ContactName]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Region] = @p0
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [WA]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
LAZYK - John Steel
TRAIH - Helvetius Nagy
WHITC - Karl Jablonski
```

Это позволяет в точности увидеть, как выглядит сгенерированный SQL-запрос. Обратите внимание, что сгенерированный оператор SQL — это не просто форматированная строка, он использует параметры. Поэтому, применяя LINQ to SQL, мы автоматически получаем защиту от атак внедрением SQL.

Внимание! Если вы видите в вашем результате, что имя, ассоциированное с заказчиком LAZYK — Ned Plimpton вместо John Steel, как показано в предыдущем примере, то вы, вероятно, запустили пример из листинга 12.1, не восстановив затем данные, как рекомендовалось. Вы можете исправить это перед тем, как будут затронуты все будущие примеры.

В последующих главах я продемонстрирую использование этого средства протоколирования для обнаружения потенциальных проблем, связанных с производительностью.

Используйте метод `GetChangeSet()`

Вы можете использовать метод `GetChangeSet()` объекта `DataContext` для получения всех сущностных объектов, содержащих изменения, которые должны быть сохранены в базе данных при вызове `SubmitChanges`. Это удобно для протоколирования и отладки. Этот метод также документирован в главе 16.

Попробуйте использовать частичные классы или файлы отображения

Без сомнения, одной из главных трудностей использования любого инструмента ORM является управление изменениями в базе данных. Если вы держите всю логику бизнес-классов и логику LINQ to SQL в одних и тех же модулях, то этим вы создаете себе проблемы сопровождения при изменениях базы данных. Рассмотрите возможность применения частичных классов, добавляя бизнес-логику в модуль, отдельный от модулей сгенерированных сущностных классов. Используя частичные классы для хранения ваших атрибутов базы данных LINQ to SQL отдельно от бизнес-логики, вы минимизируете необходимость в добавлении кода к любому сгенерированному сущностному классу.

Альтернативно вы можете разделить бизнес-классы и отображение сущностей LINQ to SQL, используя внешний XML-файл отображения. Речь идет об XML-файле, который отображает бизнес-объекты на базу данных, не полагаясь на атрибуты LINQ to SQL. Вы можете прочесть больше о файлах отображения в разделе “XML-схема внешнего файла отображения” главы 15, а также в разделе, посвященном конструктору `DataContext`, в главе 16.

Попробуйте использовать частичные методы

Частичные методы были добавлены в набор языковых средств C# относительно недавно, но это не значит, что вы должны их игнорировать. Частичные методы — это легковесные события, которые позволяют вам перехватывать определенные события, происходящие в сущностных классах. Вся прелест частичных методов в том, что если вы не реализуете тело такого метода, то нет никаких накладных расходов, и никакой код не порождается компилятором для их вызова.

В разделе “Вызов соответствующих частичных методов” главы 15 мы поговорим об их применении в сущностных классах.

Инструменты

Точно так же, как я заранее хотел дать вам несколько советов — до того, как они действительно вам понадобятся — стоит заранее упомянуть о некоторых инструментах, которые в дальнейшем могут облегчить вам жизнь. Опять-таки, вполне возможно, что я расскажу вам о них до того, как их применение станет оправданным для вас, но думаю, вам стоит знать о них и о том, как они могут облегчить и ускорить вашу адаптацию к LINQ to SQL.

SQLMetal

Хотя мне еще предстоит описать разные способы создания сущностных классов, необходимых для использования LINQ to SQL с базой данных, вы должны знать, что простейший путь генерации сущностных классов для всей базы данных, если у вас еще нет бизнес-классов, заключается в применении программы SQLMetal. Вы можете найти ее в своем каталоге %windir%\Microsoft.NET\Framework\v3.5. SQLMetal позволяет специфицировать базу данных, и он сгенерирует все необходимое для сущностных классов LINQ to SQL. SQLMetal — инструмент командной строки, не имеющий графического интерфейса пользователя.

Чтобы увидеть доступные опции программы SQLMetal, откройте окно командной строки Visual Studio. Для этого откройте в меню Microsoft Visual Studio 2008 подменю Visual Studio Tools (Инструменты Visual Studio), найдите в нем команду Visual Studio 2008 Command Prompt (Командная строка Visual Studio 2008) и выберите ее.

Как только окно командной строки откроется, введите в нем sqlmetal и нажмите <Enter>:

```
sqlmetal
```

Эта команда приведет к отображению шаблона вызова программы со всеми опциями:

```
Microsoft (R) Database Mapping Generator 2008 Beta 2 version 1.00.20706
for Microsoft (R) .NET Framework version 3.5
Copyright (C) Microsoft Corporation. All rights reserved.

SqlMetal [options] [<input file>]
    Generates code and mapping for the LINQ to SQL component of the .NET framework.
    SqlMetal can:
        - Generate source code and mapping attributes or a mapping file from a database.
        - Generate an intermediate dbml file for customization from the database.
        - Generate code and mapping attributes or mapping file from a dbml file.

Options:
    /server:<name>          Database server name.
    /database:<name>          Database catalog on server.
    /user:<name>              Login user ID (default: use Windows Authentication).
    /password:<password>      Login password (default: use Windows Authentication).
    /conn:<connection string> Database connection string. Cannot be used with
                                /server, /database, /user or /password options.
    /timeout:<seconds>        Timeout value to use when SqlMetal accesses the
                                database (default: 0 which means infinite).

    /views                   Extract database views.
    /functions                Extract database functions.
    /sprocs                   Extract stored procedures.
    /dbml[:file]              Output as dbml. Cannot be used with /map option.
    /code[:file]               Output as source code. Cannot be used with /dbml option.
    /map[:file]                Generate mapping file, not attributes. Cannot be used
                                with /dbml option.
```

/language:<language>	Language for source code: VB or C# (default: derived from extension on code file name).
/namespace:<name>	Namespace of generated code (default: no namespace).
/context:<type>	Name of data context class (default: derived from database name).
/entitybase:<type>	Base class of entity classes in the generated code (default: entities have no base class).
/pluralize	Automatically pluralize or singularize class and member names using English language rules.
/serialization:<option>	Generate serializable classes: None or Unidirectional (default: None).
/provider:<type>	Provider type: SQLCompact, SQL2000, or SQL2005. (default: provider is determined at run time).
<input file>	May be a SqlExpress mdf file, a SqlCE sdf file, or a dbml intermediate file.

Create code from SqlServer:

```
SqlMetal /server:myserver /database:northwind /code:nwind.cs /namespace:nwind
```

Generate intermediate dbml file from SqlServer:

```
SqlMetal /server:myserver /database:northwind /dbml:northwind.dbml
/namespace:nwind
```

Generate code with external mapping from dbml:

```
SqlMetal /code:nwind.cs /map:nwind.map northwind.dbml
```

Generate dbml from a SqlCE sdf file:

```
SqlMetal /dbml:northwind.dbml northwind.sdf
```

Generate dbml from SqlExpress local server:

```
SqlMetal /server:.\\sqlexpress /database:northwind /dbml:northwind.dbml
```

Generate dbml by using a connection string in the command line:

```
SqlMetal /conn:"server='myserver'; database='northwind'" /dbml:northwind.dbml
```

Как видите, здесь даже приводится несколько примеров. Большая часть этих опций самоочевидна, остальные поясняются в табл. 13.1.

Таблица 13.1. Опции командной строки SQLMetal

Опция/Пример	Описание
/server:<имя> /server:.\\SQLEXPRESS	Эта опция позволяет специфицировать имя сервера базы данных, к которому нужно подключиться. Если ее пропустить, SQLMetal принимает по умолчанию localhost/sqlexpress. Чтобы заставить SQLMetal сгенерировать сущностные классы из файла MDF, пропустите эту опцию вместе с опцией /database и специфицируйте полное путевое имя MDF-файла в конце команды.
/database:<имя> /database:Northwind	Это имя базы данных на указанном сервере, для которой генерируются сущностные классы. Чтобы заставить SQLMetal сгенерировать сущностные классы из файла MDF, пропустите эту опцию вместе с опцией /server и специфицируйте полное путевое имя MDF-файла в конце команды.
/user:<имя> /user:sa	Это пользовательская учетная запись, применяемая для регистрации в указанной базе данных при подключении для создания сущностных классов.
/password:<пароль> /password:1590597893	Это пароль, используемый указанной пользовательской учетной записью для регистрации в указанной базе данных при подключении для создания сущностных классов.

Опция/Пример	Описание
/conn:<строка подключения> /conn:"Data Source=.\SQLEXPRESS; InitialCatalog=Northwind; IntegratedSecurity=SSPI;"	Это строка подключения к базе данных. Вы можете использовать ее вместо указания опций /server, /database, /user и /password.
/timeout:<секунды> /timeout:120	Эта опция позволяет специфицировать значение таймаута в секундах для использования с командами базы данных. Пропуск этой опции аналогичен указанию SQLMetal значения по умолчанию — 0, что означает отсутствие таймаута. В настоящее время эта опция не поддерживается и никаких изменений в сгенерированном коде не вызывает. Вы можете опробовать эту опцию, чтобы узнать, работает ли она в вашей инсталляции Visual Studio 2008. Если эта опция не работает, подумайте об установке свойства CommandTimeout класса DataContext, или для еще более тонкого контроля вызовите метод DataContext.GetCommand, чтобы установить таймаут для конкретного запроса. Пример такого подхода приведен в листинге 16.29 главы 16.
/views /views	Указывайте эту опцию для генерации SQLMetal необходимых свойств Table<'T> и сущностных классов для поддержки специфированных представлений (views) базы данных.
/functions /functions	Указывайте эту опцию для генерации SQLMetal методов для вызова специфированных пользовательских функций базы данных.
/sprocs /sprocs	Указывайте эту опцию для генерации SQLMetal методов для вызова специфированных хранимых процедур базы данных.
/dbml[:файл] /dbml:Northwind.dbml	Эта опция указывает имя промежуточного DBML-файла. Целью генерации этого файла является контроль имён сгенерированных сущностных классов и их свойств. С этой опцией вы можете сгенерировать промежуточный DBML-файл, отредактировать его и затем создать модуль исходного кода вызовом SQLMetal на промежуточном файле DBML с указанием опции /code. Альтернативно вы можете загрузить промежуточный DBML-файл, созданный с этой опцией в Object Relational Designer, отредактировать его в дизайнере и позволить дизайнеру сгенерировать необходимый исходный код. Эта опция не может быть использована с опцией /map.
/code[:file] /code:Northwind.cs	Это имя файла, который создает SQLMetal, содержащего класс, производный от DataContext и сущностные классы на указанном языке программирования. Эта опция не может использоваться вместе с /dbml. Интересно, что если вы специфицируете обе опции — и /code, и /map — в одном вызове SQLMetal, то получите код, сгенерированный баз атрибутов LINQ to SQL. Конечно, вы должны использовать сгенерированное отображение (map) вместе с генерированным кодом, чтобы иметь возможность применять LINQ to SQL.

Окончание табл. 13.1

Опция/Пример	Описание
/map[:файл] /map:northwindmap.xml	Эта опция указывает, что SQLMetal должен генерировать внешний XML-файл отображения, в противоположность модулю исходного кода, специфицированного опцией /code. Этот внешний XML-файл отображения может затем быть загружен при создании экземпляра DataContext. Это позволяет использовать LINQ to SQL без какого-либо действительного скомпилированного вместе с вашим исходного кода LINQ to SQL. Интересно, что если вы специфицируете обе опции — и /code, и /map — в одном вызове SQLMetal, то получите код, сгенерированный без атрибутов LINQ to SQL. Конечно, вы должны использовать сгенерированное отображение вместе со сгенерированным кодом, чтобы иметь возможность применять LINQ to SQL.
/language:<язык> language:C#	Эта опция определяет, для какого языка программирования должен генерироваться код SQLMetal. Допустимые опции в настоящее время: csharp, C# и VB.
/namespace:<имя> /namespace:nwind	Эта опция диктует пространство имен, в котором будут находиться сгенерированный класс-наследник DataContext и сущностные классы.
/context:<тип> /context:Northwind	Указывает имя сгенерированного класса-наследника DataContext. Если данная опция опущена, имя класса будет совпадать с именем базы данных, для которой генерируется код.
/entitybase:<тип> /entitybase: MyEntityClassBase	Указывает имя класса, используемого в качестве базового для генерируемых SQLMetal сущностных классов. Если данная опция опущена, сгенерированные сущностные классы не наследуются ни от какого другого класса.
/pluralize /pluralize	Эта опция заставляет SQLMetal сохранять множественное число в именах таблиц, но единственное — в именах классов, отображенных на эти таблицы. Таким образом, для таблицы базы данных по имени Customers сгенерированный сущностный класс будет назван Customer (в единственном числе), а Table<Customer> будет сгенерирован с именем Customers (во множественном числе). Таким образом, объект Customer хранится в таблице Customers. С грамматической точки зрения это звучит правильно. Без указания этой опции сущностный класс будет назван Customers (во множественном числе), и Table<Customers> будет назван Customers (во множественном числе). Это значит, что объект Customers буде находиться в таблице Customers. С точки зрения грамматики звучит неверно.
/serialization:<опция> /serialization:None	Эта опция специфицирует, должен ли SQLMetal генерировать атрибуты сериализации для классов. Возможные варианты значений — None и Unidirectional. Если эта опция не специфицирована, по умолчанию SQLMetal принимает None.
/provider:<тип> /provider:SQL2005	Данная опция используется для спецификации класса-поставщика базы данных. Допустимые значения: SQLCompact, SQL2000 и SQL2005. Все эти классы поставщиков находятся в пространстве имен System.Data.Linq.SqlClient, и пространство имен должно быть специфицировано для этой опции.

Обратите внимание, что опции /dbml, /code и /map могут быть специфицированы без указания имени файла. Если имя файла не указано, то сгенерированный код XML выводится на консоль.

Сравнение XML-файла отображения и промежуточного файла DBML

Одним из запутанных аспектов использования SQLMetal является то, что он позволяет указать два разных типа выходных XML-файлов. Один создается указанием опции /map, а второй — опцией /dbml.

Отличие между этими двумя файлами состоит в том, что опция /map создает внешний XML-файл отображения, предназначенный для загрузки при создании экземпляра DataContext. Опция /map — альтернатива генерации или написанию вручную исходного модуля, содержащего атрибуты LINQ to SQL, которые вы компилируете. С таким подходом ваш исходный код не включает никакого специфичного для базы данных кода LINQ to SQL, подлежащего компиляции и компоновке вместе с ним. Это допускает в некоторой степени динамическое обращение к базе данных, поскольку нет необходимости ни в каком предварительно генерированном и компилированном коде. Я называю это “в некоторой степени динамическим обращением”, потому что ваш код должен знать имена таблиц и полей; в противном случае он просто не будет знать, что запрашивать. Внешний XML-файл отображения инструктирует LINQ to SQL о том, какие есть таблицы, столбцы и хранимые процедуры, с которыми он может взаимодействовать, и на какие классы, свойства и методы они должны быть отображены.

Опция /dbml создает промежуточный файл DBML (XML) для обеспечения возможности редактирования имен впоследствии генерируемых сущностных классов и их свойств. Затем вы должны генерировать модуль исходного кода, запустив SQLMetal снова — на этот раз на файле DBML, а не на базе данных, и указав опцию /code. Или же вы можете загрузить промежуточный DBML-файл в Object Relational Designer, отредактировать его там и позволить дизайнеру сгенерировать исходный код необходимых сущностных классов.

Другая причина того, что SQLMetal может генерировать два типа XML файлов — XML-файла отображения и промежуточного DBML-файла — состоит в путанице из-за их сходства. Так что не удивляйтесь, когда увидите, насколько они похожи. Схема XML-файла отображения обсуждается в главе 15.

Работа с промежуточным файлом DBML

Как я сказал, назначение промежуточного файла DBML — позволить вам вмешаться в процесс между извлечением схемы базы данных и генерацией сущностных классов, с тем, чтобы контролировать имена генерируемых сущностных классов и их имен. Поэтому, если у вас нет такой необходимости, то вам и незачем генерировать промежуточный файл DBML. Пока продолжим, исходя из того, что такая необходимость есть.

Предполагая, что у вас есть расширенная база данных Northwind, присоединенная к вашему серверу базы данных SQL Server, вот как вы можете создать промежуточный файл DBML:

```
sqlmetal /server:.\\SQLEXPRESS /database:Northwind /pluralize /sprocs /functions
/views /dbml:Northwind.dbml
```

На заметку! Указание опций /server и /database при запуске SQLMetal требует, чтобы расширенная база данных Northwind была присоединена к SQL Server.

Вдобавок вам может понадобиться специфицировать соответствующие опции /user и /password, чтобы SQLMetal мог подключиться к базе данных.

Или же, если вы предпочитаете, можно сгенерировать промежуточный файл DBML из файла MDF:

```
sqlmetal /pluralize /procs /functions /views /dbml:Northwind.dbml
"C:\Northwind.mdf"
```

Но заметку! Генерация промежуточного файла DBML из файла MDF может потребовать присоединения этого MDF-файла к SQL Server с именем C:\NORTHWIND.MDF или подобным. Для корректной работы примеров вы должны переименовать базу данных в "Northwind" внутри SQL Server Enterprise Manager или SQL Management Studio.

Любой из двух описанных подходов должен привести к генерации идентичных промежуточных DBML-файлов. Я специфицировал только те опции, которые существенны для чтения базы данных и производства DBML-файла. Такие опции, как /language и /code, имеют значение только при создании модуля исходного кода.

Отредактировав промежуточный XML-файл, вы можете сгенерировать модуль исходного кода следующим образом:

```
sqlmetal /namespace:nwind /code:Northwind.cs Northwind.dbml
```

Опции, которые я специфицировал при запуске SQLMetal, важны при генерации исходного кода.

Схема промежуточного файла DBML

Если вы решили пройти путь создания промежуточного файла DBML, так, чтобы можно было отредактировать его и затем сгенерировать на его основе ваши существенные классы, вам нужно знать схему промежуточного файла, а также что означают его имена элементов и атрибутов.

Поскольку схема — субъект изменений, проконсультируйтесь с документацией Microsoft относительно схемы промежуточного файла DBML, чтобы получить наиболее свежее определение схемы и пояснения к ней. Разобравшись в схеме, вы можете вручную редактировать промежуточный DBML-файл для управления именами генерируемых SQLMetal на основе этого промежуточного файла сущностных классов и их свойств.

Или, что еще лучше — вы можете загрузить сгенерированный DBML-файл в Visual Studio Object Relational Designer и отредактировать его там. Это даст вам графический интерфейс пользователя для манипуляции вашей объектно-реляционной моделью, а также избавит от необходимости знания и понимания схемы. В следующем разделе я опишу, как редактировать объектно-реляционную модель.

Object Relational Designer

В дополнение к инструменту SQLMetal существует также графический инструмент пользователя для генерации сущностных классов, который работает в среде Visual Studio. Этот инструмент называется Object Relational Designer, но вы чаще встретите ссылки на него, как на LINQ to SQL Designer, или O/R Designer, или даже DLinq Designer. SQLMetal предназначен для генерации сущностных классов для всех таблиц базы данных, несмотря на тот факт, что у вас есть возможность выборочной генерации промежуточного DBML-файла, его модификации и генерации на его основе сущностных классов. Более того, SQLMetal — утилита командной строки. А для более разборчивого подхода с графическим интерфейсом пользователя Object Relational Designer — то, что надо. В этой главе я буду называть его просто "дизайнером".

Дизайнер предоставляет в распоряжение разработчика средства моделирования сущностных классов методом перетаскивания. Вам не нужно беспокоиться: дизайнер делает за вас большую часть трудной работы. Ваше дело — выбрать таблицы базы дан-

ных, которые хотите моделировать, если они вам доступны, и отредактировать имена сущностных классов и их свойств. Конечно, у вас есть выбор выполнить в дизайнере все моделирование вручную, если вам необходим полный контроль.

Создание вашего файла классов LINQ to SQL

Первый шаг в использовании дизайнера — создание файла классов LINQ to SQL посредством щелчка правой кнопкой мыши на вашем проекте и выбора команды Add⇒New Item (Добавить⇒Новый элемент) из контекстного меню. Сделав это, вы увидите диалоговое окно Add New Item (Добавить новый элемент). Из списка инсталлированных шаблонов выберите LINQ to SQL Classes (Классы LINQ to SQL). Отредактируйте имя по своему усмотрению. Имя моделируемой базы данных — обычно подходящий выбор для имени файла классов LINQ to SQL. Расширение файла классов LINQ to SQL выглядит как .dbml. Для данного примера я использую в качестве имени файла Northwind.dbml.

Внимание! Если вы создаете файл по имени Northwind.dbml в ранее созданном проекте примеров этой книги, будьте осторожны, чтобы не вызвать конфликт между сгенерированным дизайнером кодом и уже существующим вашим кодом.

Щелкните на кнопке Add (Добавить), выбрав имя для файла. После этого вы увидите пустое окно. Это — ваша поверхность проектирования. На рис. 13.1 показано, как она выглядит.

Если вы щелкнете на этой поверхности и заглянете в окно Properties (Свойства), то увидите там свойство по имени Name. Значением свойства Name будет имя сгенерированного класса-наследника DataContext. Поскольку я назвал мой файл классов LINQ to SQL Northwind.dbml, по умолчанию значением моего свойства Name будет NorthwindDataContext, что вполне подходит.

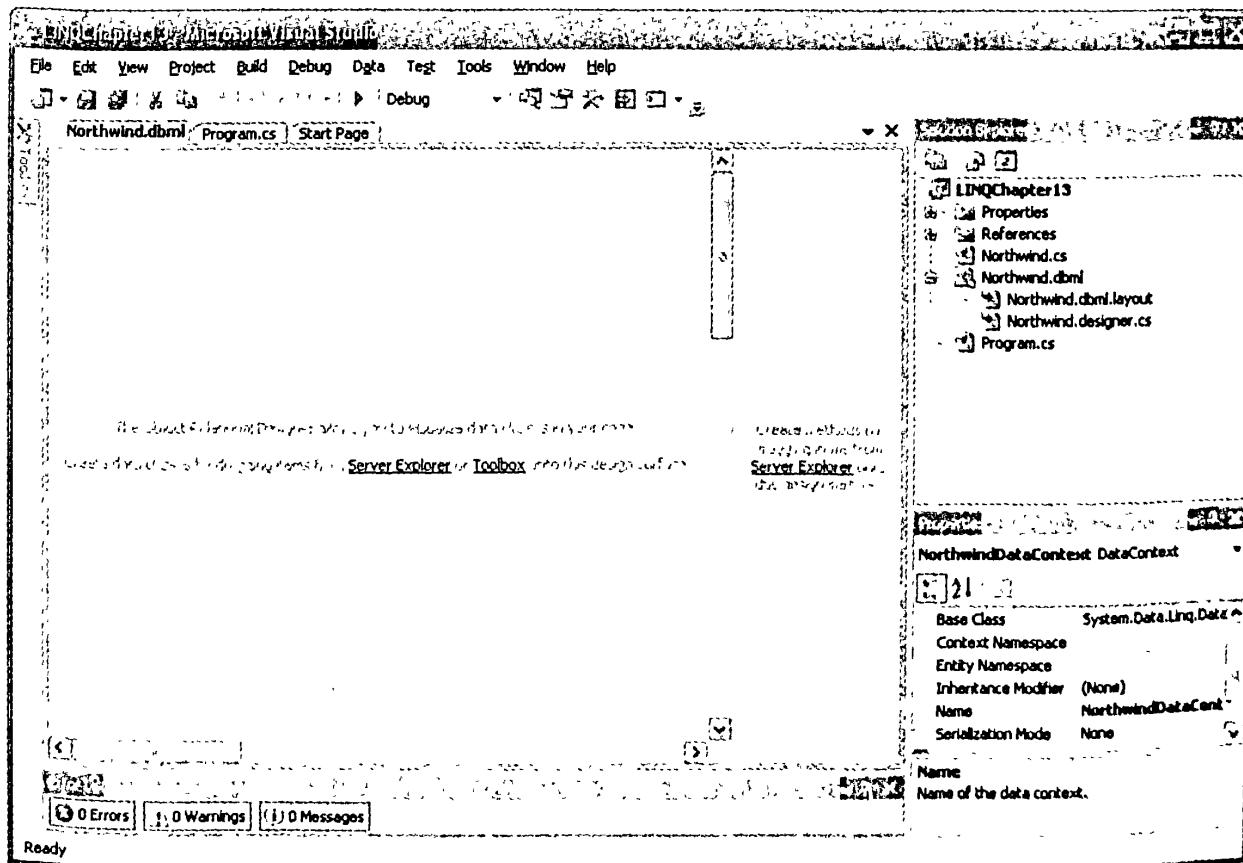


Рис. 13.1. Поверхность проектирования Object Relational Designer

Если вы заглянете в Solution Explorer, то увидите там прямо под файлом Northwind.cs еще один файл, названный Northwind.designer.cs. Если вы откроете его, то увидите, что пока он содержит очень мало кода. По сути, он будет содержать конструкторы для нового класса-наследника DataContext, который называется NorthwindDataContext.

Подключение DataContext к базе данных

Следующий шаг — добавление в окне Server Explorer соединения с соответствующим сервером, содержащим базу данных Northwind, если его еще там нет.

Совет. Если вы не видите окна Server Explorer, выберите команду Server Explorer из меню View (Вид) внутри Visual Studio.

Чтобы добавить соединение к базе данных, выполните щелчок правой кнопкой мыши на узле Data Connections (Соединения с данными) в окне Server Explorer и выберите команду меню Add Connection (Добавить соединение) для открытия диалогового окна Add Connection (Добавить соединение), показанного на рис. 13.2. Поле ввода Data source (Источник данных) по умолчанию будет содержать Microsoft SQL Server (SqlClient), что вам и надо.

Конфигурируйте соответствующие настройки вашей базы данных Northwind в диалоговом окне Add Connection. Вы можете щелкнуть на кнопке Test Connection (Проверить соединение), чтобы удостовериться, что соединение работает корректно.

Имея правильно конфигурированное соединение, щелкните на кнопку OK. Теперь у вас должен появиться узел, представляющий соединение с базой данных Northwind — прямо под узлом Data Connections в окне Server Explorer. Теперь вы имеете доступ к базе данных Northwind в проводнике.

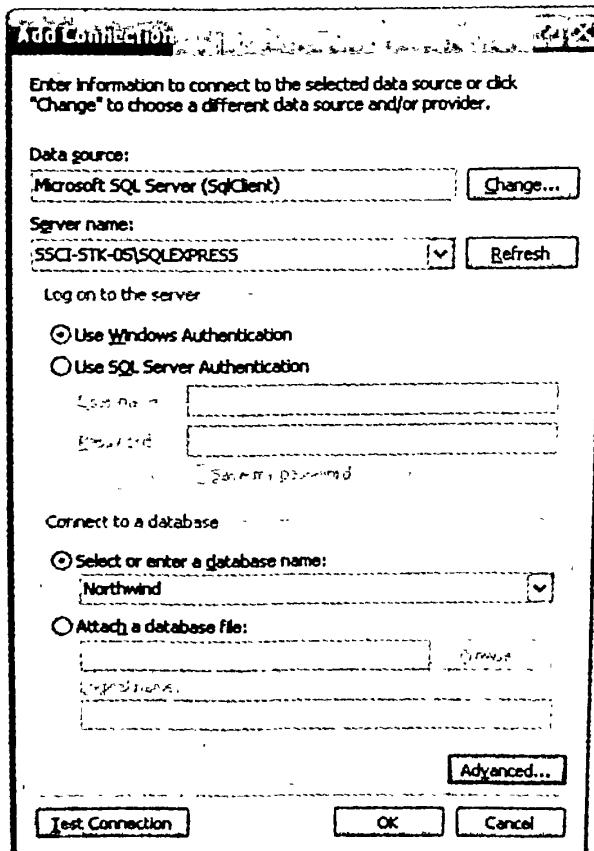


Рис. 13.2. Диалоговое окно Add Connection

Прежде чем двигаться дальше, убедитесь, что вы видите файл Northwind.dbml в редакторе Visual Studio.

Добавление сущностного класса

Найдите вашу базу данных Northwind в списке Data Connections в окне Server Explorer. Разверните узел Tables (Таблицы) и вы увидите список таблиц базы данных Northwind. Сущностные классы создаются перетаскиванием таблиц из списка Tables в окно Server Explorer на поверхность проектирования.

Перетащите таблицу Customers из Server Explorer на поверхность проектирования. Тем самым вы указываете дизайнеру создать сущностный класс для таблицы Customers по имени Customer. После этого ваша поверхность проектирования должна выглядеть, как показано на рис. 13.3.

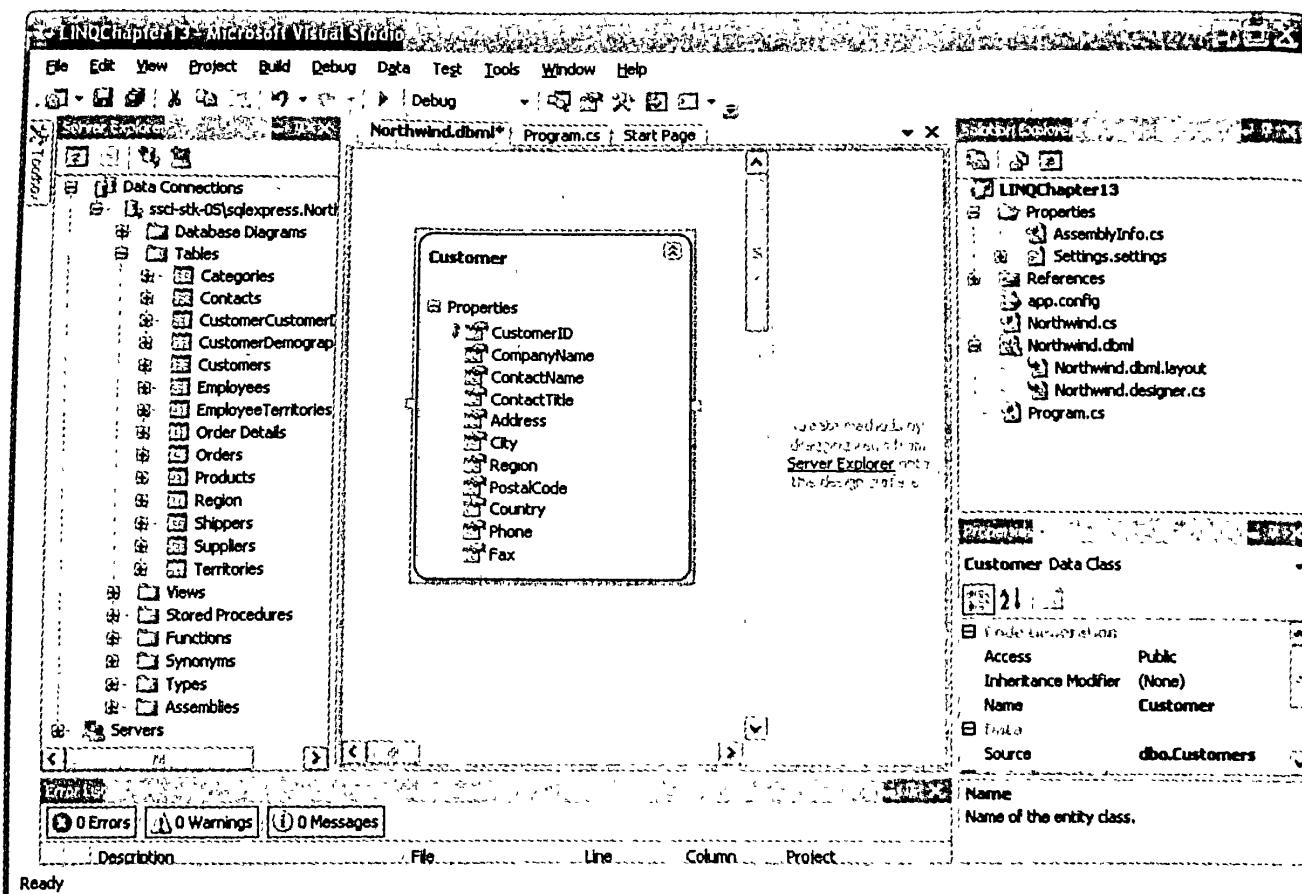


Рис. 13.3. Внешний вид дизайнера после добавления таблицы Customers на поверхность проектирования

Вам может понадобиться изменить размеры некоторых панелей, чтобы ясно все видеть. Перетаскивая таблицу Customers по поверхности проектирования, вы добавляете исходный код сущностного класса Customer в файл Northwind.designer.cs. После сборки проекта вы сможете начать использовать сущностный класс Customer для доступа и обновления данных в базе Northwind. Вот как все просто!

Однако прежде чем строить проект и писать код, использующий генерированные сущностные классы, я хочу создать еще несколько дополнительных вещей, необходимых для использования всех преимуществ LINQ to SQL. Теперь из Server Explorer перетащите на поверхность проектирования таблицу Orders. Вы можете подвигать ее по поверхности, чтобы найти подходящее местоположение. Тем самым вы указываете дизайнеру создать сущностный класс для таблицы Orders по имени Order. После этого поверхность дизайнера будет выглядеть, как показано на рис. 13.4.

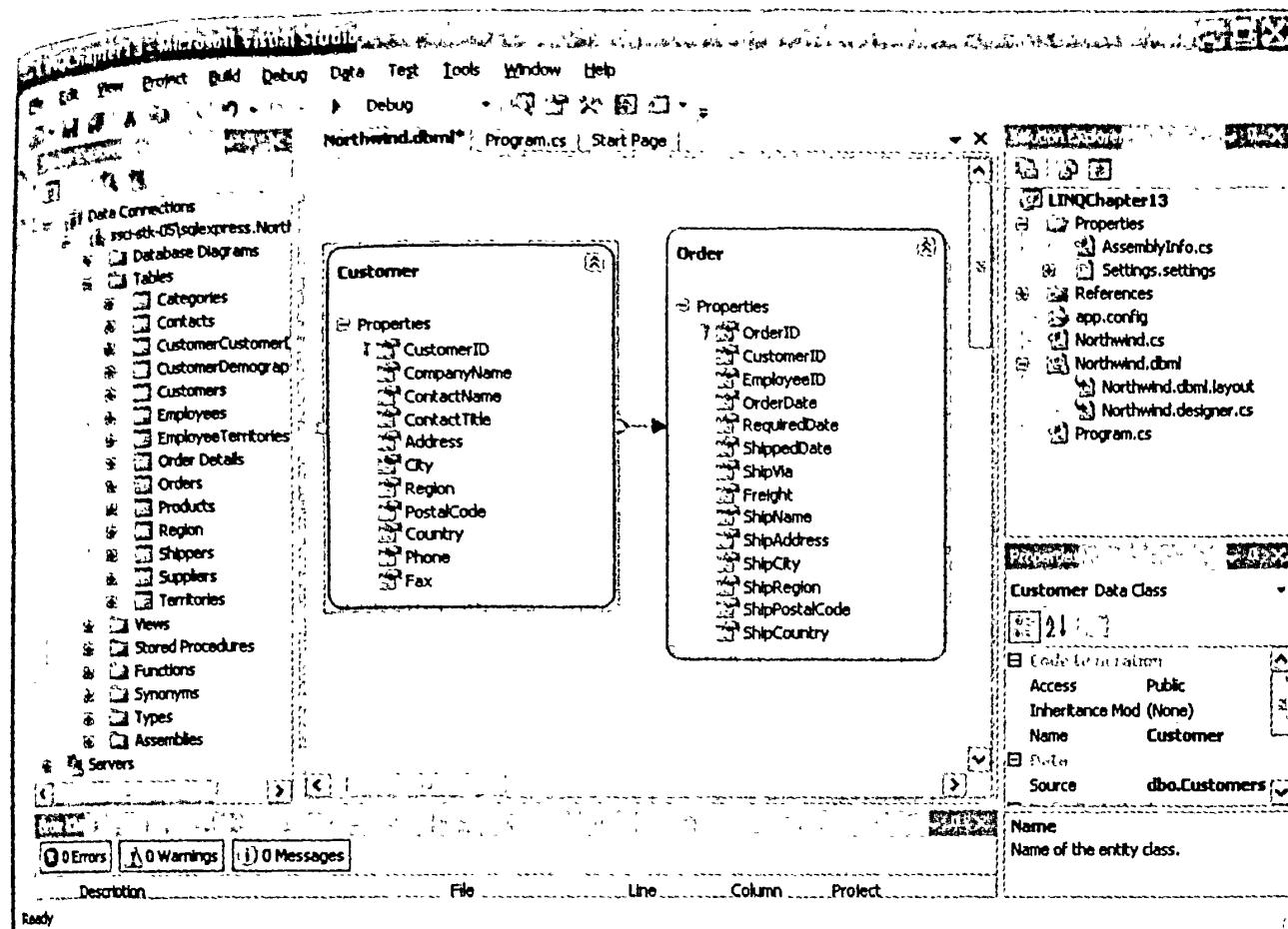


Рис. 13.4. Дизайнер после перетаскивания таблица Orders на поверхность проектирования

Вы можете заметить, что на рис. 13.4 справа уже нет панели, которая была видна на предыдущих экранных снимках дизайнера. Это окно — панель Methods (Методы). Я закрыл эту панель ее правым щелчком мыши на поверхности и выбором команды Hide Methods Pane (Скрыть панель методов) из контекстного меню. Чтобы вновь открыть панель Methods, щелкните правой кнопкой мыши на поверхности и выберите команду Show Methods Pane (Показать панель методов) из контекстного меню. Я оставил панель Methods закрытой, чтобы было видно больше поверхности проектирования.

Если вы посмотрите на поверхность, то увидите там пунктирную линию, соединяющую класс Customer с классом Order. Эта пунктирная линия представляет отношение, известное в LINQ to SQL как ассоциация между таблицами Customers и Orders, заданное ограничением внешнего ключа FK_Orders_Customers, которое существует в базе данных Northwind. Эта линия указывает на то, что дизайнер также создаст необходимую ассоциацию в сущностных классах для поддержки отношений между этими двумя сущностными классами. Наличие такой ассоциации позволит получать ссылку на коллекцию заказов определенного заказчика через свойство объекта Customer, а также получать ссылку на заказчика, разместившего конкретный заказ, через свойство объекта Order.

Если вам не нужна сгенерированная ассоциация, вы можете выбрать пунктирную линию, представляющую ее, и удалить ее нажатием <Delete> или выполнив щелчок правой кнопкой мыши и выбрав команду Delete (Удалить) из контекстного меню.

Использование сгенерированных дизайнером сущностных классов

Теперь вы готовы к использованию сущностных классов, сгенерированных для вас дизайнером. Листинг 13.2 содержит пример запроса в базе данных Northwind списка заказчиков из Лондона.

Листинг 13.2. Пример использования сгенерированных дизайнером сущностных классов

```
NorthwindDataContext db = new NorthwindDataContext();
IQueryable<Customer> custs = from c in db.Customers
                               where c.City == "London"
                               select c;
foreach(Customer c in custs)
{
    Console.WriteLine("{0} has {1} orders.", c.CompanyName, c.Orders.Count);
}
```

Это похоже на мой другой пример с исключением. Обратите внимание, что я не специфицировал никакой информации о соединении при создании экземпляра объекта `NorthwindDataContext`. Дело в том, что дизайнер сгенерировал мой класс `NorthwindDataContext` с конструктором без параметров, который получает информацию соединения из файла настроек проекта по имени `app.config`. Он был даже настолько любезен, что установил для меня соответствующее значение в файле настроек. Ниже можно видеть, как выглядит сгенерированный конструктор без параметров.

Сгенерированный дизайнером конструктор наследника `DataContext`

```
public NorthwindDataContext() :
    base(global::LINQChapter13.Properties.Settings.Default.NorthwindConnectionString,
        mappingSource)
{
    OnCreated();
}
```

Внимание! Если вы загрузите код примеров, сопровождающий эту книгу, не забудьте обновить установку `connectionString` в файле `app.config`. В частности, источник данных будет содержать имя машины, которое, скорее всего, не совпадет с именем вашей машины.

Обратите внимание в предыдущем коде, что я имею возможность доступа к заказам полученного заказчика через свойство `Orders` объекта `Customer`. Это возможно, благодаря ассоциации, автоматически созданной для меня дизайнером. Здорово, не правда ли? А вот результат работы кода из листинга 13.2:

```
Around the Horn has 13 orders.
B's Beverages has 10 orders.
Consolidated Holdings has 3 orders.
Eastern Connection has 8 orders.
North/South has 3 orders.
Seven Seas Imports has 9 orders.
```

Редактирование модели сущностных классов

Естественно, вы можете получить некоторую власть над именами сущностных классов, их свойствами, именами этих свойств и свойствами свойств. Эй, Microsoft, вы не могли сделать систему именования еще более запутанной? Действительно ли вы хотите называть члены классов “свойствами”, зная, что в Visual Studio этим термином обозначаются настройки?

Гибкость и легкость использования сущностных классов для контроля имен и их свойств — это то, что делает дизайнер столь привлекательным. Здесь все можно сделать методами указания, щелчка и перетаскивания!

Редактирование имени сущностного класса

Вы можете редактировать имя сущностного класса, выполнив двойной щелчок на имени на поверхности проектирования либо выбором сущностного класса на поверхности и редактированием свойства Name в окне Properties (Свойства).

Редактирование свойства сущностного класса (настройки сущностного класса)

Вы можете редактировать свойства как настройки сущностного класса, выбрав нужный класс в панели и редактируя соответствующие его свойства в окне Properties. Вы имеете возможность редактировать имя таблицы базы данных, в которой эти сущности хранятся, вставлять, обновлять и удалять переопределенные методы, а также прочие свойства.

Редактирование свойства сущностного класса (члена сущностного класса)

Вы можете редактировать имя свойства сущностного класса как члена сущностного класса, выполнив тройной щелчок на имени свойства на поверхности проектирования. Я также не догадывался о существовании такой вещи, как тройной щелчок, но она существует и действует. Или же вы можете выбрать свойство сущностного класса на поверхности и отредактировать его свойство Name в окне Properties.

Редактирование свойств свойства сущностного класса (настройки)

Вы можете редактировать свойства свойств сущностного класса, выбрав свойство на поверхности проектирования и редактируя соответствующее свойство в окне Properties. Там вы найдете все свойства, соответствующие свойствам атрибутов сущностного класса, такие как Name, UpdateCheck, для атрибута Column сущностного класса. Атрибуты сущностного класса мы обсудим в главе 15.

Добавление объектов к модели сущностных классов

Перетащить сущностный класс на поверхность достаточно просто — до тех пор, пока у вас есть соответствующая таблица базы данных в Server Explorer. Но бывают случаи, когда вы лишены такого удобства. Возможно, вы определили сущностный класс сначала, и планируете генерировать базу данных, вызвав метод CreateDatabase на объекте DataContext. Или, возможно, вы собираетесь воспользоваться преимуществом наследования сущностного класса, для которого нет соответствующей таблицы, куда он отображается.

Добавление нового сущностного класса

Один из способов добавления новых классов к вашей модели сущностных классов заключается в перетаскивании таблиц базы данных из вашего окна Server Explorer, как я сделал в предыдущем разделе. Другой способ создания сущностного класса состоит в перетаскивании на поверхность проектирования объекта Object Relational Designer Class из панели инструментов Visual Studio Toolbox. Отредактируйте имя и установите свойства сущностного класса, как было описано в предыдущем разделе.

Добавление новых свойств (членов) сущностного класса

Вы можете добавлять новые свойства (члены) сущностного класса, выполняя щелчок правой кнопкой мыши на сущностном классе в дизайнере и выбирая команду Properties (Свойства) в контекстном меню Add (Добавить). Как только свойство добавлено к сущностному классу, следуйте инструкциям по редактированию свойств свойства сущностного класса, приведенным в разделе “Редактирование свойств свойства сущностного класса (настройки)”.

Добавление новой ассоциации

Вместо использования механизма перетаскивания для создания ассоциации, как вы поступали при добавлении нового сущностного класса из панели инструментов Visual Studio Toolbox, ассоциацию вы создаете щелчком на объекте Association (Ассоциация) в панели инструментов, за которым следует щелчок на сущностном классе, задающем сторону “один” в отношении “один ко многим”, а за ним — щелчок на дочернем сущностном классе, задающем сторону “многие” в том же отношении. Каждый из этих классов должен иметь соответствующее свойство перед тем, как вы добавите ассоциацию, чтобы можно было отобразить первичный ключ на стороне “один” на внешний ключ на стороне “многие”. Выбрав второй класс (стороны “многие”) ассоциации, вы откроете диалоговое окно Association Editor (Редактор ассоциации), позволяющее отобразить свойство первого класса на соответствующее свойство второго класса.

Надлежащим образом отобразив свойства и закрыв окно Association Editor, вы увидите пунктирную линию, соединяющую родительский и дочерний сущностные классы.

Выберите ассоциацию щелчком на пунктирной линии и установите соответствующие свойства в окне Properties. За дополнительной информацией о свойствах ассоциации обратитесь к описанию атрибута Association и его свойств в главе 15.

Добавление нового наследования

Вы также можете использовать Object Relational Designer для моделирования отношения наследования. Добавление отношения наследования работает так же, как добавление новой ассоциации. Выберите объект Inheritance (Наследование) в панели инструментов Visual Studio Toolbox и щелкните на сущностном классе, который будет наследником, а затем — сущностный класс, который будет базовым. Убедитесь, что установлены все соответствующие свойства сущностного класса, как определено атрибутами InheritanceMapping и Column сущностного класса, которые я опишу в главе 15.

Добавление хранимых процедур и пользовательских функций

Чтобы заставить дизайнер сгенерировать код, необходимый для вызова хранимых процедур и определяемых пользователем функций, перетащите хранимую процедуру или пользовательскую функцию из Server Explorer в панель Methods (Методы) дизайнера. Я продемонстрирую это в следующем разделе.

Переопределение методов вставки, обновления и удаления

В главе 14 я опишу переопределение методов вставки, обновления и удаления, используемых LINQ to SQL при проведении изменений в объекте сущностного класса. Вы можете переопределять методы по умолчанию, добавляя специфические методы в сущностный класс. Приняв такой подход, не забудьте использовать частичные классы, чтобы не модифицировать никакого сгенерированного кода. В главе 14 я покажу, как это делается.

Однако переопределение методов вставки, обновления и удаления также легко выполняется в дизайнере. Предположим, что у вас есть хранимая процедура по имени InsertCustomer, которая будет вставлять запись о новом заказчике в таблицу Customers базы данных Northwind. Ниже показана хранимая процедура, которую я для этого использую.

Хранимая процедура InsertCustomer

```
CREATE PROCEDURE dbo.InsertCustomer
(
    @CustomerID nchar(5),
    @CompanyName nvarchar(40),
```

```

@ContactName nvarchar(30),
@ContactTitle nvarchar(30),
@Address nvarchar(60),
@City nvarchar(15),
@Region nvarchar(15),
@PostalCode nvarchar(10),
@Country nvarchar(15),
@Phone nvarchar(24),
@Fax nvarchar(24)
)
AS
INSERT INTO Customers
(
CustomerID,
CompanyName,
ContactName,
ContactTitle,
Address,
City,
Region,
PostalCode,
Country,
Phone,
Fax
)
VALUES
(
@CustomerID,
@CompanyName,
@ContactName,
@ContactTitle,
@Address,
@City,
@Region,
@PostalCode,
@Country,
@Phone,
@Fax
)

```

На заметку! Хранимая процедура `InsertCustomer` не является частью расширенной базы данных Northwind. Я добавил ее вручную в целях демонстрации.

Чтобы переопределить метод вставки сущностного класса `Customer`, сначала убедитесь, что у вас видима панель `Methods`. Если нет, выполните щелчок правой кнопкой мыши на панели дизайнера и выберите команду `Show Methods Pane` (Показать панель Методов) из контекстного меню. Затем откройте окно `Server Explorer` в `Visual Studio`, если оно еще не было открыто. Найдите и разверните узел `Stored Procedures` (Хранимые процедуры) в узле соответствующей базы данных. Ваша среда `Visual Studio` должна выглядеть подобно показанной на рис. 13.5.

Найдя хранимую процедуру, просто перетащите ее в панель `Methods`, которая представляет собой окно справа от модели сущностных классов. На рис. 13.6 показано окно `Visual Studio` после того, как я перетащил хранимую процедуру `InsertCustomer` в панель `Methods`.

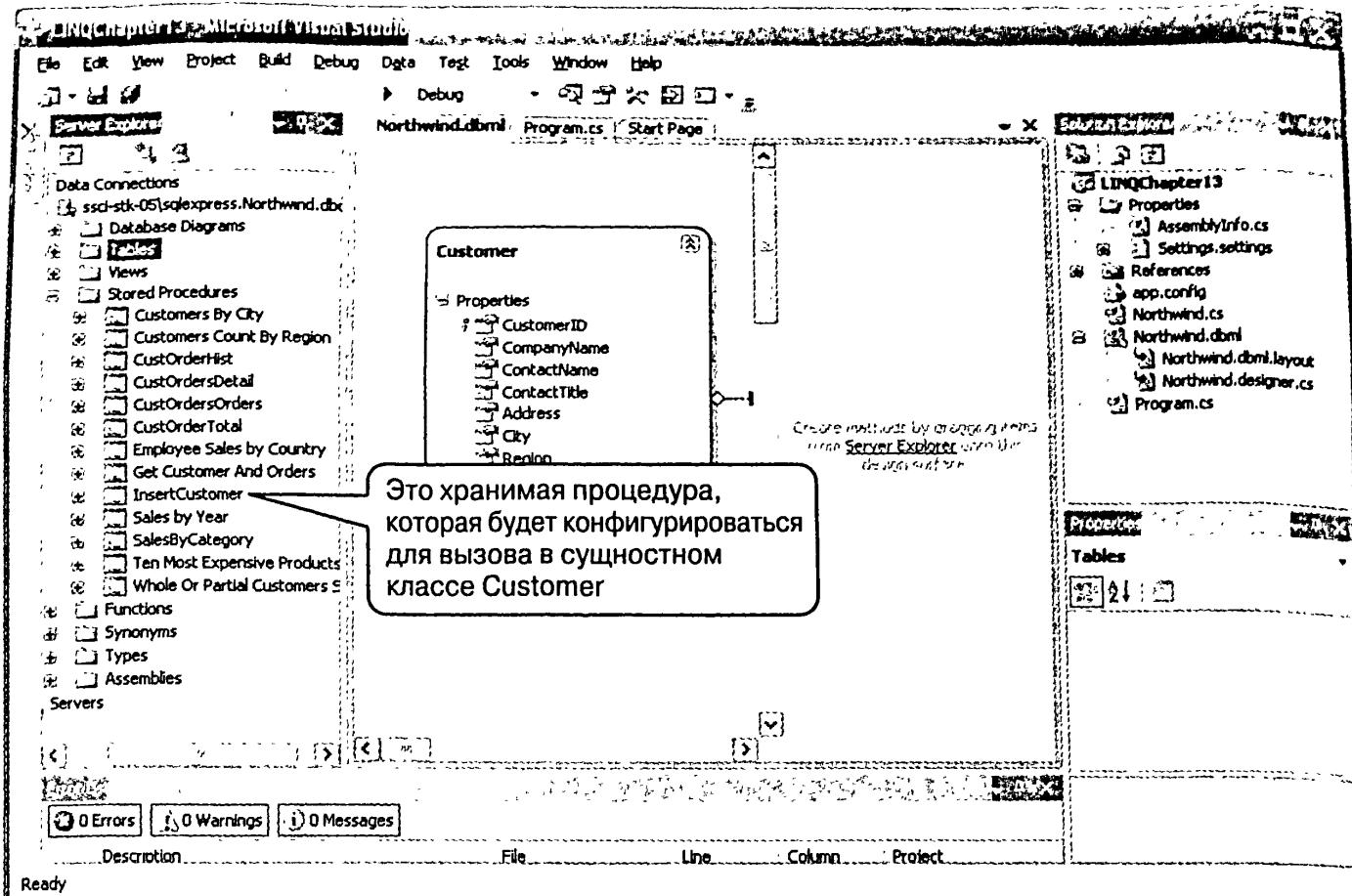


Рис. 13.5. Нахождение хранимой процедуры

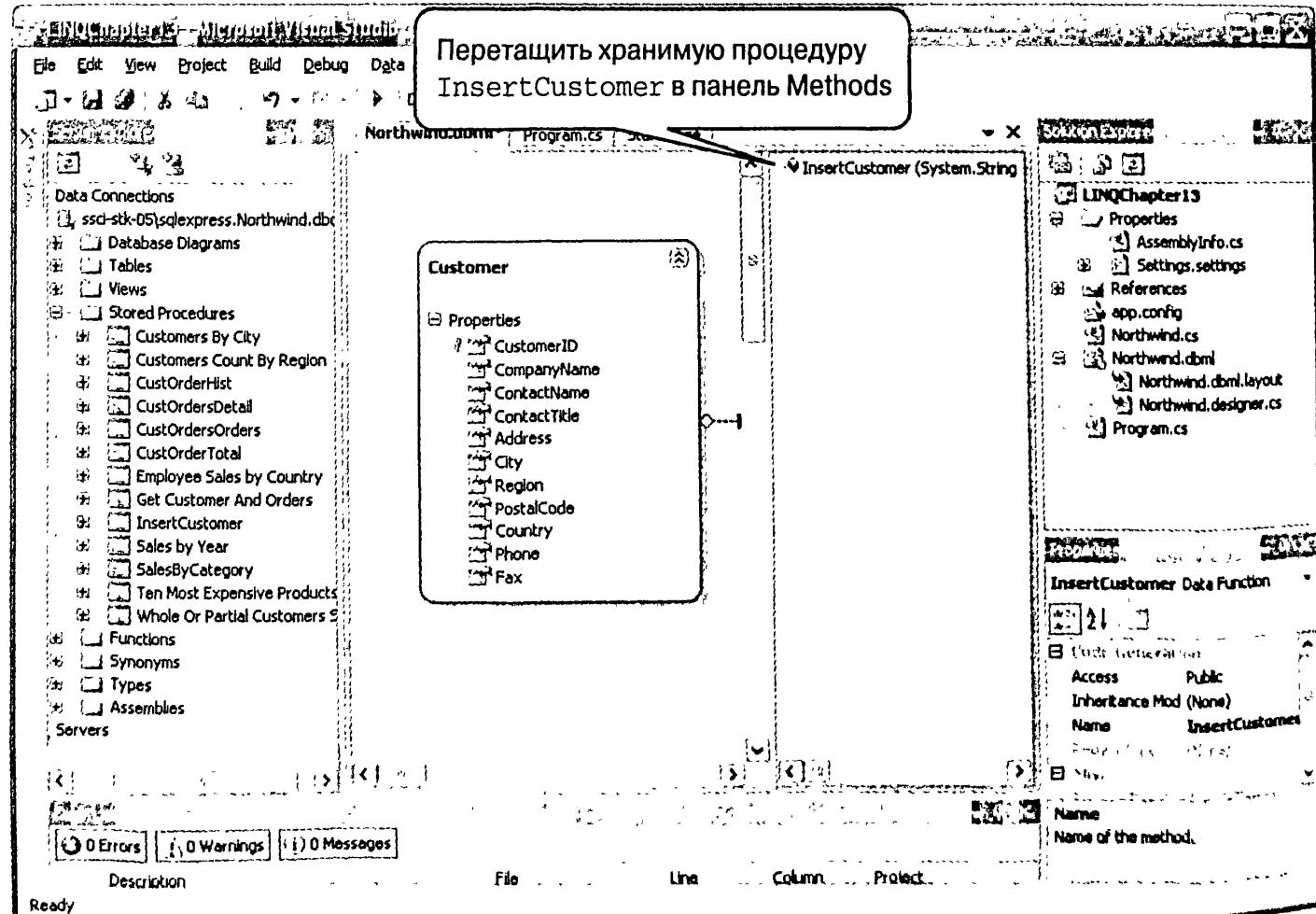


Рис. 13.6. Перетаскивание хранимой процедуры в панель Methods

Перетаскивание хранимой процедуры из окна Server Explorer в панель Methods — способ, которым вы инструктируете дизайнер о необходимости сгенерировать код, требуемый для вызова хранимой процедуры из LINQ to SQL. Точно так же вы инструктируете дизайнера о необходимости генерации кода вызова пользовательской функции.

Обеспечение доступа к хранимой процедуре из LINQ to SQL — первый шаг к выполнению операций вставки, обновления или удаления посредством вызова хранимой процедуры вместо обычного метода по умолчанию. Следующий шаг — переопределение одной из этих операций для вызова доступной теперь хранимой процедуры.

Теперь, когда хранимая процедура `InsertCustomer` находится в панели Methods, выберите класс `Customer` на поверхности проектирования и просмотрите окно свойств Properties класса `Customer`. Теперь вы увидите список Default Methods (Методы по умолчанию). Выберите метод `Insert`, щелкнув на нем. Вы получите кнопку выбора с многоточием (...), как показано на рис. 13.7.

Теперь просто щелкните на кнопке с многоточием для отображения диалогового окна Configure Behavior (Конфигурировать поведение). Выберите переключатель Customize (Настроить), затем выберите хранимую процедуру `InsertCustomer` из раскрывающегося списка. Отобразите Method Arguments (Аргументы метода) слева на соответствующие Customer Class Properties (Свойства пользовательского класса) справа, как показано на рис. 13.8.

Как это ни удивительно, после этого все мои аргументы метода отображены по умолчанию на соответствующие свойства класса. Отлично!

Как только вы отобразите все аргументы метода, щелкните на кнопке OK. После этого вы готовы к вставке записей `Customer` посредством вызова хранимой процедуры `InsertCustomer`. В листинге 13.3 я создам нового заказчика, используя хранимую процедуру `InsertCustomer`.

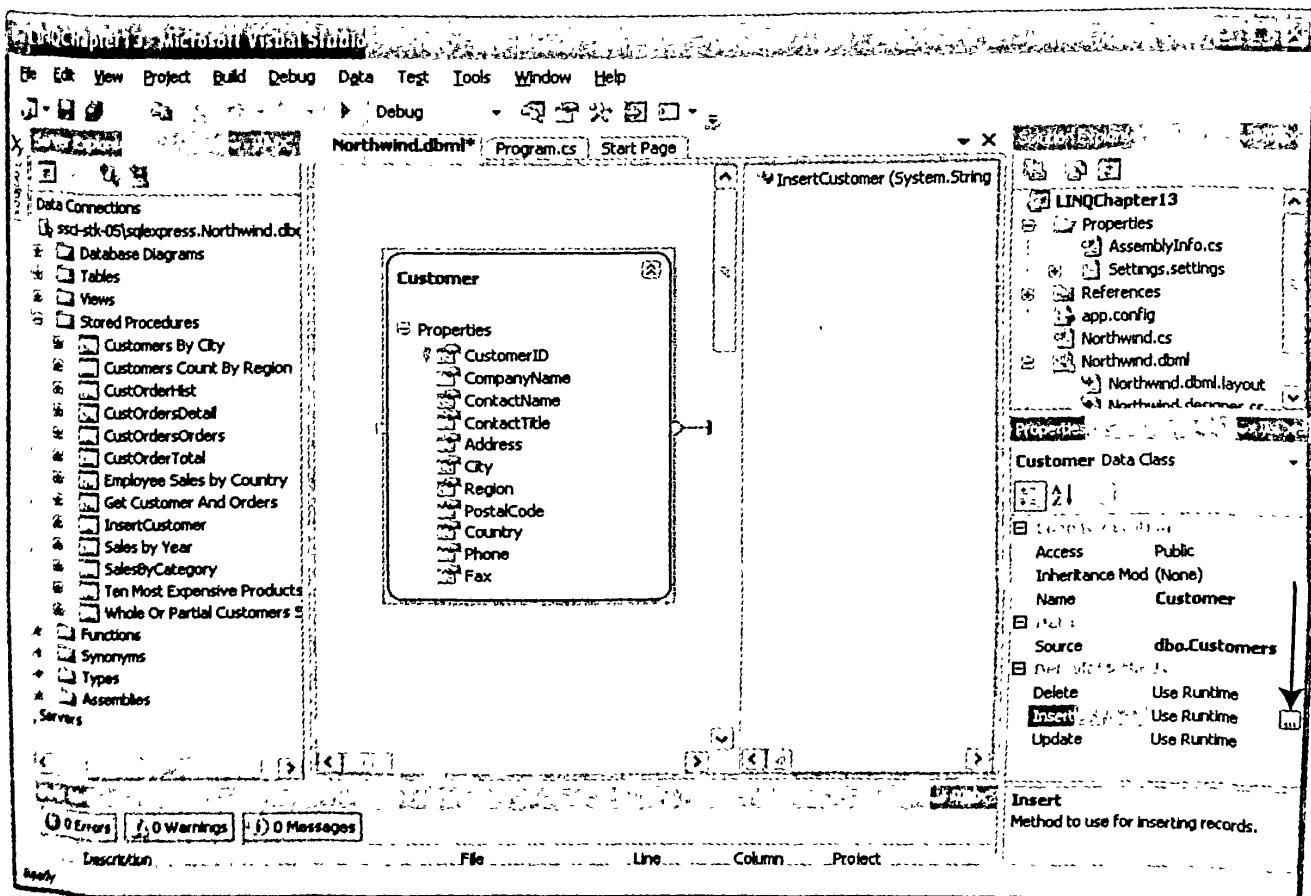


Рис. 13.7. Выбор метода Insert в категории Default Methods окна Properties

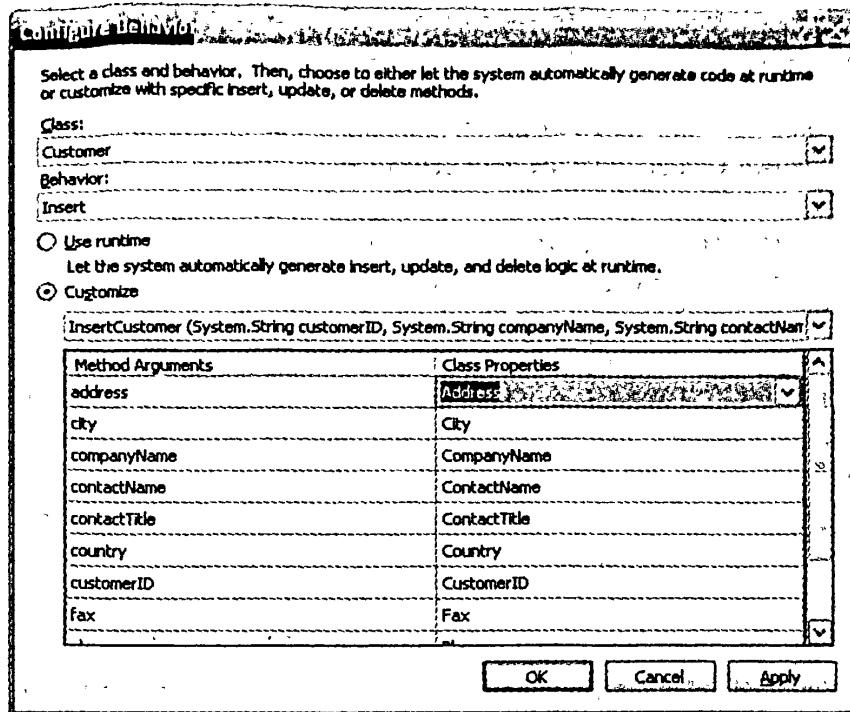


Рис. 13.8. Отображение аргументов метода на свойства класса

Листинг 13.3. Создание записи Customer с переопределенным методом вставки по умолчанию

```

NorthwindDataContext db = new NorthwindDataContext();
db.Log = Console.Out;
Customer cust =
    new Customer
{
    CustomerID = "EWICH",
    CompanyName = "Every 'Wich Way",
    ContactName = "Vickey Rattz",
    ContactTitle = "Owner",
    Address = "105 Chip Morrow Dr.",
    City = "Alligator Point",
    Region = "FL",
    PostalCode = "32346",
    Country = "USA",
    Phone = "(800) EAT-WICH",
    Fax = "(800) FAX-WICH"
};
db.Customers.InsertOnSubmit(cust);
db.SubmitChanges();
Customer customer = db.Customers.Where(c => c.CustomerID == "EWICH").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
// Восстановление базы данных.
db.Customers.DeleteOnSubmit(cust);
db.SubmitChanges();

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод `InsertOnSubmit`, вызванный в предыдущем коде, назывался `Add`, а метод `DeleteOnSubmit` — `Remove`.

Обратите внимание, что я не указываю никакого пространства имен для класса `Customer`; т.е. я использую класс `Customer`, находящийся в пространстве имен проекта — сгенерированный дизайнером класс `Customer`.

В листинге 13.3 нет ничего особенного. Я просто создаю экземпляр `DataContext`, которым в данном случае является генерированный дизайнером `NorthwindDataContext`. Затем я создаю новый объект `Customer` и вставляю его в свойство `Customers Table<T>`. Затем вызываю метод `SubmitChanges` для сохранения нового заказчика в базе данных. Затем запрашиваю этого заказчика из базы данных и отображаю его на консоли — просто чтобы доказать, что запись действительно была вставлена в таблицу базы. И последнее, что я делаю — удаляю только что вставленного заказчика вызовом метода `DeleteOnSubmit`, и фиксирую изменение в базе данных вызовом метода `SubmitChanges`, так что база данных остается в том же состоянии, что и была изначально, и последующие примеры будут работать правильно, а этот пример может быть выполнен много-кратно. Рассмотрим вывод программы из листинга 13.3:

```

EXEC @RETURN_VALUE = [dbo].[InsertCustomer] @CustomerID = @p0, @CompanyName = @p1,
@ContactName = @p2, @ContactTitle = @p3, @Address = @p4, @City = @p5, @Region = @p6,
@PostalCode = @p7, @Country = @p8, @Phone = @p9, @Fax = @p10
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- @p1: Input String (Size = 15; Prec = 0; Scale = 0) [Every 'Wich Way]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Vickey Rattz]
-- @p3: Input String (Size = 5; Prec = 0; Scale = 0) [Owner]
-- @p4: Input String (Size = 19; Prec = 0; Scale = 0) [105 Chip Morrow Dr.]
-- @p5: Input String (Size = 15; Prec = 0; Scale = 0) [Alligator Point]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [FL]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [32346]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(800) EAT-WICH]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(800) FAX-WICH]
-- @RETURN_VALUE: Output Int32 (Size = 0; Prec = 0; Scale = 0) []
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
SELECT TOP 1 [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Every 'Wich Way - Vickey Rattz
DELETE FROM [dbo].[Customers] WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1)
AND ([ContactName] = @p2) AND ([ContactTitle] = @p3) AND ([Address] = @p4) AND
([City] = @p5) AND ([Region] = @p6) AND ([PostalCode] = @p7) AND ([Country] = @p8)
AND ([Phone] = @p9) AND ([Fax] = @p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [EWICH]
-- @p1: Input String (Size = 15; Prec = 0; Scale = 0) [Every 'Wich Way]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Vickey Rattz]
-- @p3: Input String (Size = 5; Prec = 0; Scale = 0) [Owner]
-- @p4: Input String (Size = 19; Prec = 0; Scale = 0) [105 Chip Morrow Dr.]
-- @p5: Input String (Size = 15; Prec = 0; Scale = 0) [Alligator Point]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [FL]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [32346]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(800) EAT-WICH]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(800) FAX-WICH]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1

```

Хотя это не так просто сразу обнаружить в этом выводе, SQL-оператор `insert` здесь не был создан. Вместо него вызывалась хранимая процедура `InsertCustomer`. Дизайнер очень облегчает задачу переопределения методов вставки, обновления и удаления для существующего класса.

Совместное использование SQLMetal и O/R Designer

Поскольку формат промежуточного DBML-файла SQLMetal совпадает со схемой XML формата Object Relational Designer, вполне возможно использовать их совместно.

Например, вы можете сгенерировать промежуточный файл DBML для базы данных, используя для этого SQLMetal, а затем загрузить его в O/R Designer, чтобы подправить имена сущностных классов или их свойств по своему желанию. Такой подход предоставляет простой способ генерации сущностных классов для всей базы данных, оставляя возможность простой модификации того, что вам нужно.

Другой пример, где такая взаимозаменяемость может оказаться удобной — возможность переопределения операций вставки, обновления и удаления, которые предназначены для проведения сущностным классом изменений в базе данных. Вы можете генерировать промежуточный файл DBML для вашей базы данных посредством SQLMetal, но затем загружать его в дизайнер и модифицировать методы вставки, обновления и удаления, как было описано в разделе настоящей главы, посвященном Object Relational Designer.

Резюме

В соответствии с принятым мною стилем, большая часть информации этой главы может показаться преждевременной, поскольку нам еще только предстоит поговорить о сущностных классах и классе `DataContext`. Однако я не мог бы с чистой совестью позволить вам двигаться дальше, если бы не дал некоторых советов и не описал инструментов, доступных для разработки LINQ to SQL. Воспринимайте эти советы как фундамент для лучшего понимания того, что будет изложено далее.

Помните, что у вас есть два инструмента для моделирования сущностных классов. Первый — SQLMetal — является инструментом командной строки, больше подходящим для генерации сущностных классов для всей базы данных в целом. Второй — Object Relational Designer, часто называемый LINQ to SQL Designer — инструмент с графическим интерфейсом пользователя, позволяющий выполнять моделирование сущностных классов методом перетаскивания, который запускается в среде Visual Studio. Он больше подходит для итеративной и новой разработки. Но, как я уже отмечал, эти два инструмента вполне могут работать совместно. Лучше всего начать с SQLMetal для генерации ваших сущностных классов для всей вашей базы данных, а затем сопровождать их с помощью Object Relational Designer.

Теперь, вооруженные некоторыми знаниями относительно инструментов LINQ to SQL, вы можете перейти к созданию собственных сущностных классов. В главе 14 я покажу, как выполняются наиболее распространенные операции с базой данных, с которыми вам придется регулярно иметь дело.

ГЛАВА 14

Операции для баз данных в LINQ to SQL

В этой главе я опишу и продемонстрирую выполнение типичных операций для баз данных в LINQ to SQL. В частности, я расскажу, как выполняются:

- вставки;
- запросы;
- обновления;
- удаления.

После рассказа о стандартных операциях для баз данных я продемонстрирую, как можно переопределять методы вставки, обновления и удаления по умолчанию, используемые сущностными классами для сохранения изменений в базе данных.

И последняя тема, которую я раскрою здесь — автоматическая трансляция запросов LINQ to SQL, включая моменты, на которые следует особо обращать внимание при написании запросов.

Для того чтобы говорить о стандартных операциях для баз данных, мне придется обращаться к `DataContext` и сущностным классам. Я знаю, что пока еще не предоставил вам достаточно подробной информации о работе сущностных классов и `DataContext`, но она будет приведена в последующих главах. Сущностные классы мы обсудим в главе 15, а `DataContext` — в главе 16. А пока просто запомните, что `DataContext` управляет соединением с базой данных, а также объектами сущностных классов. Объект сущностного класса представляет определенную запись базы данных в форме объекта.

Предварительные условия для запуска примеров

Чтобы запустить примеры этой главы, вам понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите еще раз и выполните требования раздела “Предварительные условия для запуска примеров” из главы 12.

Некоторые общие методы

Дополнительно для запуска примеров этой главы вам понадобятся некоторые общие методы, которые используются примерами. Прочтите и выполните требования раздела “Некоторые общие методы” главы 12.

Использование программного интерфейса LINQ to SQL API

Чтобы запускать примеры этой главы, вам нужно будет добавить необходимые ссылки и директивы `using` к вашему проекту. Прочтите и выполните то, что сказано в разделе “Использование LINQ to SQL API” из главы 12.

Стандартные операции для баз данных

Хотя детали выполнения запросов LINQ to SQL я изложу в последующих главах, здесь я хочу дать вам некоторые поверхностные сведения о том, как выполнять рутинные операции с базой данных, без излишнего усложнения. Эти примеры предназначены только для демонстрации базовых концепций. И как таковые, они не будут включать в себя проверку ошибок и обработку исключений.

Например, поскольку многие из базовых операций, о которых пойдет речь, вносят изменения в базу данных, при этом следовало бы позаботиться об обнаружении и разрешении конфликтов параллельного доступа. Но в целях простоты данные примеры не будут следовать этим принципам. Однако в главе 17 мы рассмотрим детально обнаружение и разрешение таких конфликтов.

Вставки

Создания экземпляра сущностного класса, такого как `Customer`, недостаточно для выполнения вставки записи в базу данных. Сущностный объект должен быть либо вставлен в коллекцию таблицы `Table<T>`, где `T` — тип сущностного класса, хранящегося в таблице, либо должен быть добавлен в `EntitySet<T>` на сущностном объекте, отслеживаемом `DataContext`, где `T` — тип сущностного класса.

Первый шаг при вставке записи в базу данных — создание `DataContext`. Это первый шаг для любого запроса LINQ to SQL. Второй шаг — создание экземпляра сущностного объекта из сущностного класса. Третий — вставка этого сущностного объекта в соответствующую коллекцию таблицы. И четвертый — вызов метода `SubmitChanges` на объекте `DataContext`.

Листинг 14.1 содержит пример вставки записи в базу данных.

Листинг 14.1. Вставка записи посредством вставки сущностного объекта в `Table<T>`

```
// 1. Создание DataContext.
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// 2. Создание экземпляра сущностного объекта.
Customer cust =
    new Customer
{
    CustomerID = "LAWN",
    CompanyName = "Lawn Wranglers",
    ContactName = "Mr. Abe Henry",
    ContactTitle = "Owner",
    Address = "1017 Maple Leaf Way",
    City = "Ft. Worth",
    Region = "TX",
    PostalCode = "76104",
    Country = "USA",
    Phone = "(800) MOW-LAWN",
    Fax = "(800) MOW-LAWO"
};
// 3. Добавление сущностного объекта в таблицу Customers.
db.Customers.InsertOnSubmit(cust);
```

```

// 4. Вызов метода SubmitChanges.
db.SubmitChanges();
// 5. Опрос записи.
Customer customer = db.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
// Эта часть кода просто восстанавливает базы данных,
// чтобы можно было запустить пример снова.
Console.WriteLine("Deleting the added customer LAWN.");
db.Customers.DeleteOnSubmit(cust);
db.SubmitChanges();

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод InsertOnSubmit, вызванный в приведенном коде, назывался Add, а метод DeleteOnSubmit — Remove.

Об этом примере говорить особо нечего. Во-первых, я создаю экземпляр объекта Northwind, чтобы иметь объект DataContext на базе данных Northwind. Во-вторых, я создаю экземпляр объекта Customer и наполняю его, используя новое средство инициализации объектов C# 3.0. В-третьих, вставляю созданный экземпляр Customer в таблицу Customers типа Table<Customer>, в класс Northwind DataContext. В-четвертых, вызываю метод SubmitChanges для сохранения вновь созданного объекта Customer в базе данных. В-пятых, запрашиваю обратно из базы данных только что вставленную запись о заказчике, чтобы доказать, что она на самом деле была вставлена.

На заметку! Если вы запустите этот пример, то новая запись будет временно добавлена в таблицу Customers базы данных Northwind для заказчика LAWN. Обратите внимание, что после того, как вновь добавленная запись запрошена и отображена, она удаляется. Я делаю это, чтобы пример можно было запускать многократно и вновь добавленные записи не влияли на последующие примеры. Если любой пример, модифицирующий базу данных, по каким-то причинам не сможет успешно завершиться, вы должны вручную вернуть базу в ее исходное состояние.

Так выглядит результат запуска листинга 14.1:

```

Lawn Wranglers - Mr. Abe Henry
Deleting the added customer LAWN.

```

Как видите, вставленная запись успешно найдена в базе данных.

Альтернативно, чтобы вставить запись в базу данных, мы можем добавить новый экземпляр сущностного класса в уже существующий сущностный объект, отслеживаемый объектом DataContext, как показано в листинге 14.2.

Листинг 14.2. Вставка записи в базу данных Northwind добавлением к EntitySet<T>

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
// Используется для запроса записи.
DateTime now = DateTime.Now;
Order order = new Order
{
    CustomerID = cust.CustomerID,
    EmployeeID = 4,
    OrderDate = now,
    RequiredDate = DateTime.Now.AddDays(7),
    ShipVia = 3,
    Freight = new Decimal(24.66),
}

```

```

ShipName = cust.CompanyName,
ShipAddress = cust.Address,
ShipCity = cust.City,
ShipRegion = cust.Region,
ShipPostalCode = cust.PostalCode,
ShipCountry = cust.Country
};
cust.Orders.Add(order);
db.SubmitChanges();
IEnumerable<Order> orders =
    db.Orders.Where(o => o.CustomerID == "LONEP" && o.OrderDate.Value == now);
foreach (Order o in orders)
{
    Console.WriteLine("{0} {1}", o.OrderDate, o.ShipName);
}
// Эта часть кода просто восстанавливает базы данных,
// чтобы можно было запустить пример снова.
db.Orders.DeleteOnSubmit(order);
db.SubmitChanges();

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод `DeleteOnSubmit` из предыдущего кода назывался `Remove`. Вы можете недоумевать, почему в листинге 14.1 я вызвал метод `InsertOnSubmit`, а в листинге 14.2 — метод `Add`. Такое расхождение вызвано тем фактом, что эти методы вызываются на двух разных типах. В листинге 14.1 метод `InsertOnSubmit` вызван на объекте типа `Table<T>`, а в листинге 14.2 метод `Add` вызван на объекте типа `EntitySet<T>`.

В листинге 14.2 я создал `Northwind` `DataContext`, извлек заказчика и добавил вновь сконструированный сущностный объект к свойству `Orders EntitySet<Order>` сущностного объекта `Customer`. Затем я запросил новую запись и отобразил ее на консоли.

На заметку! Опять-таки, обратите внимание, что в конце примера я удаляю вновь вставленную в таблицу `Orders` запись методом `DeleteOnSubmit`. Если код примера не выполнится до конца, вы должны вручную удалить эту запись, чтобы восстановить базу данных для последующих примеров.

В случае данного примера не похоже, чтобы он делал что-то существенно отличающееся от кода в листинге 14.1. Там вставляемый объект, которым был `Customer`, добавлялся в переменную типа `Table<Customer>`. В листинге 14.2 вставляемый объект, которым является `Order`, добавляется в переменную типа `EntitySet<Order>`.

И вот результат работы листинга 14.2:

9/2/2007 6:02:16 PM Lonesome Pine Restaurant

Вывод подтверждает, что запись о заказе действительно была вставлена в базу данных.

Вставка прикрепленных сущностных объектов

Одной из прелестей вставки записей является то, что `DataContext` обнаруживает любые зависимые ассоциированные объекты сущностных классов, которые прикреплены к данному объекту, так что они также сохраняются при вызове метода `SubmitChanges`. Под зависимыми я подразумеваю объекты сущностных классов, содержащие внешний ключ, который указывает на вставляемый объект сущностного класса. Пример приведен в листинге 14.3.

Листинг 14.3. Добавление присоединенных записей

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust =
    new Customer {
        CustomerID = "LAWN",
        CompanyName = "Lawn Wranglers",
        ContactName = "Mr. Abe Henry",
        ContactTitle = "Owner",
        Address = "1017 Maple Leaf Way",
        City = "Ft. Worth",
        Region = "TX",
        PostalCode = "76104",
        Country = "USA",
        Phone = "(800) MOW-LAWN",
        Fax = "(800) MOW-LAWO",
        Orders = {
            new Order {
                CustomerID = "LAWN",
                EmployeeID = 4,
                OrderDate = DateTime.Now,
                RequiredDate = DateTime.Now.AddDays(7),
                ShipVia = 3,
                Freight = new Decimal(24.66),
                ShipName = "Lawn Wranglers",
                ShipAddress = "1017 Maple Leaf Way",
                ShipCity = "Ft. Worth",
                ShipRegion = "TX",
                ShipPostalCode = "76104",
                ShipCountry = "USA"
            }
        }
    };
db.Customers.InsertOnSubmit(cust);
db.SubmitChanges();
Customer customer = db.Customers.Where(c => c.CustomerID == "LAWN").First();
Console.WriteLine("{0} - {1}", customer.CompanyName, customer.ContactName);
foreach (Order order in customer.Orders)
{
    Console.WriteLine("{0} - {1}", order.CustomerID, order.OrderDate);
}
// Эта часть кода просто восстанавливает базы данных,
// чтобы можно было запустить пример снова.
db.Orders.DeleteOnSubmit(cust.Orders.First());
db.Customers.DeleteOnSubmit(cust);
db.SubmitChanges();

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод `InsertOnSubmit`, вызванный в приведенном коде, назывался `Add`, а метод `DeleteOnSubmit` — `Remove`.

В листинге 14.3 я создал новый объект `Customer` с прикрепленной к нему коллекцией `Orders`, содержащей один вновь созданный экземпляр `Order`. Несмотря на то что я вставляю только объект `cust` типа `Customer` в таблицу `Customers`, и явно не вставляю заказов в таблицу `Orders`, но так как новый `Order` присоединен к новому `Customer`, этот новый `Order` также будет сохранен в базе данных при вызове метода `SubmitChanges`.

Относительно этого примера нужно сделать одно замечание. Обратите внимание, что в коде очистки в конце листинга 14.3 я вызываю метод `DeleteOnSubmit` как на новом объекте `Order`, так и на `Customer`. В данном случае я удаляю только первый `Order`, но поскольку объект `Customer` — новый, мне точно известно, что у него был только один `Order`. Необходимость в ручном удалении заказов вызвана тем, что несмотря на то, что при вставке родительского объекта в базу данных все прикрепленные к нему объекты вставляются автоматически, при удалении такого не происходит. Если бы я не удалил заказы вручную, при попытке удалить заказчика сгенерировалось бы исключение. Более подробно мы поговорим об этом позже, в разделе “Удаления” настоящей главы.

Нажмем `<Ctrl+F5>` и посмотрим на вывод кода из листинга 14.3:

```
Lawn Wranglers - Mr. Abe Henry
LAWN - 9/2/2007 6:05:07 PM
```

Из этого вывода видно, что новый `Customer` действительно был вставлен в базу данных, хотя и временно — из-за кода восстановления базы данных в конце примера.

Запросы

Выполнение запросов LINQ to SQL похоже на выполнение любого другого запроса LINQ, но с несколькими исключениями. Опишу их очень кратко.

Чтобы выполнить запрос LINQ to SQL, сначала мне нужно создать `DataContext`. Затем я могу выполнить запрос таблицы из этого `DataContext`, как показано в листинге 14.4.

Листинг 14.4. Выполнение простого запроса LINQ to SQL на базе данных Northwind

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
Customer cust = (from c in db.Customers
                 where c.CustomerID == "LONEP"
                 select c).Single<Customer>();
```

Когда выполняется этот код, заказчик, чей `CustomerID` равен “`LONEP`”, извлекается в переменную `cust`. Однако вы должны иметь в виду, что, как упоминалось в главе 5, стандартная операция запроса `Single` генерирует исключение, если последовательность, на которой он вызван, не содержит подходящих элементов. Поэтому в данном случае вы должны были точно знать, что заказчик “`LONEP`” существует. В действительности стандартная операция запроса `SingleOrDefault` обеспечивает лучшую защиту на случай, если не найдется записи, соответствующей конструкции `where`.

В этом примере необходимо отметить еще пару моментов. Во-первых, обратите внимание, что запрос использует синтаксис C# при сравнении `CustomerID` с “`LONEP`”. Об этом говорит применение двойных кавычек вместо одинарных, как того требует синтаксис SQL. К тому же используется операция проверки эквивалентности C# `==` вместо операции проверки эквивалентности SQL `=`. Это демонстрирует тот факт, что запрос на самом деле интегрирован в язык — в конце концов, это следует из самого названия LINQ — язык интегрированных запросов. Во-вторых, обратите внимание, что в этом запросе я смешиваю синтаксис выражений запроса со стандартным синтаксисом точечной нотации. Часть, представленная в синтаксисе выражений запросов, заключена в скобки, а операция `Single` вызывается с использованием стандартной точечной нотации.

Теперь вопрос вам. Уже не раз на протяжении этой книги я говорил об отложенном выполнении запросов. Вопрос состоит в том, вызовет ли запуск приведенного кода

немедленное выполнение запроса? Обдумывая ответ, не забудьте об отложенном выполнении запросов. Ответ: да; стандартная операция запроса Single вызовет немедленное выполнение запроса. Если бы я исключил вызов этой операции и немедленно вернул результат запроса минус вызов операции Single, то запрос не был бы выполнен немедленно.

Код в листинге 14.4 не выдает никакого экранного вывода, поэтому просто для того, чтобы убедиться, что код действительно извлечет соответствующего заказчика, в листинге 14.5 приведен тот же код, но с добавлением вывода на консоль, отображающего извлеченную запись о заказчике.

Листинг 14.5. Выполнение того же запроса, но с выводом на консоль

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
```

Ниже показан вывод листинга 14.5:

Lonesome Pine Restaurant - Fran Wilson

Отступления от нормы

Ранее я упоминал, что запросы LINQ to SQL подобны обычным запросам LINQ, но с некоторыми исключениями. Давайте обсудим эти исключения.

Запросы LINQ to SQL возвращают IQueryable<T>

В то время как запросы LINQ, выполненные на массивах и коллекциях, возвращают последовательности IEnumerable<T>, запросы LINQ to SQL, запрашивающие такую последовательность, возвращают последовательность типа IQueryable<T>. Листинг 14.6 содержит пример запроса, возвращающего последовательность типа IQueryable<T>.

Листинг 14.6. Простой запрос LINQ to SQL, возвращающий последовательность IQueryable<T>

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                                where c.City == "London"
                                select c;
foreach(Customer cust in custs)
{
    Console.WriteLine("Customer: {0}", cust.CompanyName);
}
```

Как видите, типом возврата этого запроса является IQueryable<Customer>. Вот результат выполнения кода из листинга 14.6:

```
Customer: Around the Horn
Customer: B's Beverages
Customer: Consolidated Holdings
Customer: Eastern Connection
Customer: North/South
Customer: Seven Seas Imports
```

Однако, как я указывал в главе 12, поскольку IQueryable<T> реализует IEnumerable<T>, обычно вы можете трактовать последовательность типа IQueryable<T>, как если бы это была последовательность IEnumerable<T>. Если у вас при этом возникнут проблемы — не забудьте об операции AsEnumerable.

Запросы LINQ to SQL выполняются на объектах Table<T>

В то время как большинство обычных запросов LINQ выполняются на массивах и коллекциях, реализующих интерфейсы `IEnumerable<T>` или `IEnumerable`, запрос LINQ to SQL выполняется на классах, реализующих интерфейс `IQueryable<T>`, таком как `Table<T>`.

Это значит, что запросам LINQ to SQL доступны дополнительные операции запросов, наряду со стандартными операциями запросов, поскольку `IQueryable<T>` реализует `IEnumerable<T>`.

Запросы LINQ to SQL транслируются в SQL

Как я говорил в главе 2, поскольку запросы LINQ to SQL возвращают последовательность типа `IQueryable<T>`, они не компилируются в код промежуточного языка .NET, как это делают обычные запросы LINQ. Вместо этого они преобразуются в деревья выражений, что позволяет им вычисляться как единое целое и транслироваться в соответствующие и оптимальные конструкции SQL. Прочтите раздел, озаглавленный “Трансляция SQL” в конце настоящей главы, чтобы узнать больше о трансляции SQL, которая происходит в запросах LINQ to SQL.

Запросы LINQ to SQL выполняются в базе данных

В отличие от запросов LINQ, которые выполняются в памяти локальной машины, запросы LINQ to SQL транслируются в вызовы SQL, которые в действительности выполняются в базе данных. Из-за этого происходит некоторое расхождение, такое как способ обработки проекции, которая не может в действительности случаться в базе данных, поскольку база ничего не знает о сущностных классах, как и о любых других классах.

К тому же, поскольку запрос на самом деле выполняется в базе данных, и база не имеет доступа к коду вашего приложения, то, что вы можете сделать в запросе, должно транслироваться, что ограничивает его некоторым образом, в зависимости от возможностей транслятора. Вы не можете просто встроить вызов написанного вами метода в лямбда-выражение и ожидать, что SQL Server догадается, что нужно делать с вызовом. Из-за этого неплохо бы знать, что может транслироваться, во что оно может быть транслировано, и что случится, когда трансляция невозможна.

Ассоциации

Опрос ассоциированного класса в LINQ to SQL не сложнее простого обращения к переменной-члену сущностного класса. Это потому, что ассоциированный класс является переменной-членом связанного сущностного класса или хранится в коллекции объектов сущностных классов, причем коллекция является переменной-членом связанного сущностного класса. Если ассоциированный класс представляет стороны “многих” в отношении “один ко многим”, то объекты класса “многих” будут храниться в коллекции, тип которой — `EntitySet<T>`, а `T` — тип сущностного класса стороны “многих”. Эта коллекция будет членом класса стороны “одного”, а ссылка на объект класса “одного” станет членом класса стороны “многих”.

Например, рассмотрим случай сущностных классов `Customer` и `Order`, которые были сгенерированы для базы данных Northwind. Заказчик может иметь множество заказов, но заказ принадлежит только одному заказчику. В данном примере класс `Customer` находится на стороне “одного” отношения “один ко многим” между сущностными классами `Customer` и `Order`. Класс `Order` находится на стороне “многих” того же отношения “один ко многим”. Поэтому заказы, принадлежащие объекту `Customer`, могут быть доступны через переменную-член, обычно именованную `Orders`, типа `EntitySet<Order>` в классе `Customer`. Заказчик в объекте `Order` доступен через переменную-член, обычно именуемую `Customer`, типа `EntityRef<Customer>` в классе `Order` (рис. 14.1).

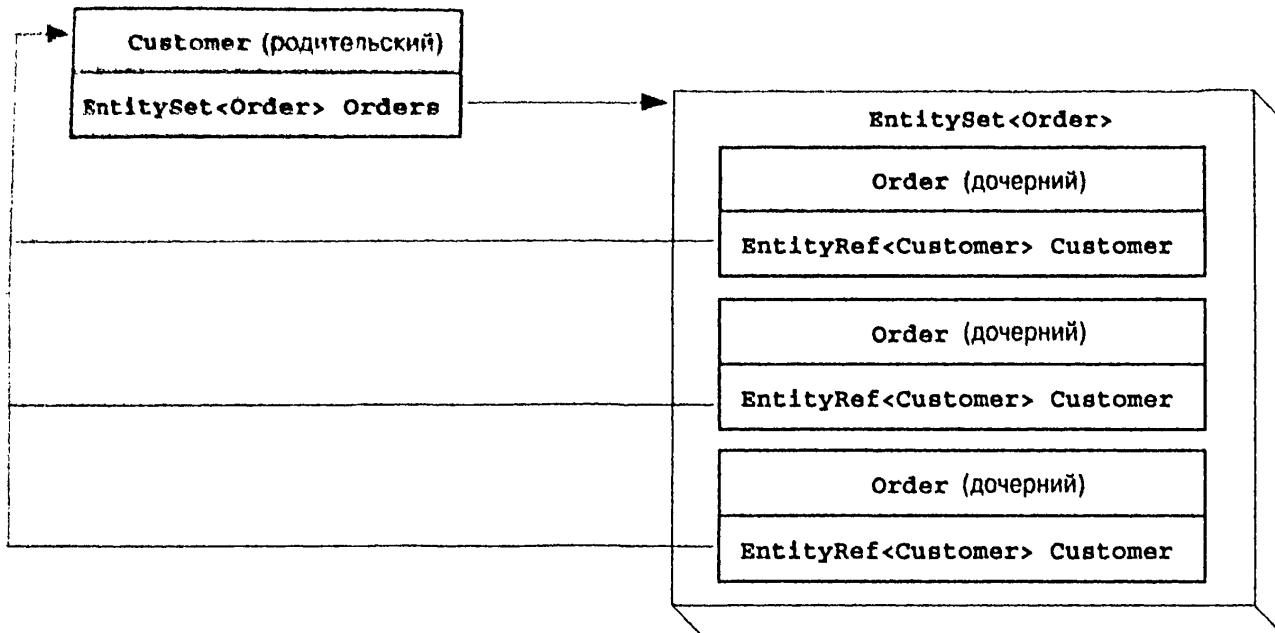


Рис. 14.1. Отношение ассоциации между родительским и дочерним сущностными классами

Если у вас вызывает затруднение определение того, какой конец отношения хранится в каком типе переменных, просто вспомните, что ребенок имеет одного родителя, а потому хранит его в единственной ссылке. Таким образом, дочерний объект хранит ассоциированного с ним родителя в переменной типа `EntityRef<T>`. Поскольку родитель может иметь много детей, он должен хранить ссылки на них в коллекции. Таким образом, родитель хранит ссылки на детей в переменной типа `EntitySet<T>`.

Классы ассоциированы посредством спецификации атрибута `Association` в свойстве класса, который хранит ссылку на ассоциированный класс в определении сущностного класса. Поскольку и родительский, и дочерний объекты имеют свойства класса, ссылающиеся друг на друга, атрибут `Association` специфицирован как в родительском, так и в дочернем сущностном классе. В главе 15 я вернусь к углубленному рассмотрению атрибута `Association`.

Листинг 14.7 содержит пример, в котором я запрашиваю определенных заказчиков и отображаю их вместе со всеми их заказами.

Листинг 14.7. Использование ассоциации для доступа к связанным данным

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                                where c.Country == "UK" &&
                                      c.City == "London"
                                orderby c.CustomerID
                                select c;
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
    foreach (Order order in cust.Orders)
    {
        Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
    }
}
```

Как видите, я перечисляю каждого заказчика, отображаю его, выполняя перечисление всех его заказов и также отображаю их. В запросе я даже никак не указываю, что мне нужны заказы. Вот сокращенный результат работы листинга 14.7:

```

Around the Horn - Thomas Hardy
 10355 11/15/1996 12:00:00 AM
 10383 12/16/1996 12:00:00 AM
 10453 2/21/1997 12:00:00 AM
 10558 6/4/1997 12:00:00 AM
 10707 10/16/1997 12:00:00 AM
 10741 11/14/1997 12:00:00 AM
 10743 11/17/1997 12:00:00 AM
 10768 12/8/1997 12:00:00 AM
 10793 12/24/1997 12:00:00 AM
 10864 2/2/1998 12:00:00 AM
 10920 3/3/1998 12:00:00 AM
 10953 3/16/1998 12:00:00 AM
 11016 4/10/1998 12:00:00 AM

...
Consolidated Holdings - Elizabeth Brown
 10435 2/4/1997 12:00:00 AM
 10462 3/3/1997 12:00:00 AM
 10848 1/23/1998 12:00:00 AM

...

```

Вам может показаться, что все замечательно. Заказы на месте, хотя я нигде даже не запрашивал их явно. У вас, однако, может возникнуть вопрос, эффективен ли такой запрос, если я нигде не обращаюсь к заказам данного заказчика?

Ответ — нет. Причина в том, что заказы в действительности не извлекаются до тех пор, пока на них никто не ссылается. Если бы мой код не обращался к свойству Orders заказчика, они бы никогда и не извлекались. Это известно под названием *отложенной загрузки*, которую не следует путать с отложенным выполнением запроса, о котором мы уже говорили.

Отложенная загрузка

Отложенная загрузка — термин, используемый для описания типа загрузки, при которой записи на самом деле не загружаются из базы данных до тех пор, пока это не становится абсолютно необходимо, что происходит при первом обращении к ним; таким образом, загрузка записей откладывается.

В листинге 14.7, если бы я ни разу не ссылался на переменную-член Orders, то заказы никогда бы не извлекались из базы данных. Для большинства ситуаций отложенная загрузка — вещь хорошая. Она предотвращает выполнение ненужных запросов и экономит пропускную способность сети за счет исключения пересылки ненужных данных.

Однако проблемы могут возникнуть. Листинг 14.8 демонстрирует такой же код, как и в листинге 14.7, за исключением того, что на этот раз я включил средство протоколирования объектом `DataContext.Log`, чтобы выявить проблему.

Листинг 14.8. Пример, демонстрирующий отложенную загрузку

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                                where c.Country == "UK" &&
                                      c.City == "London"
                                orderby c.CustomerID
                                select c;

// Включить протоколирование.
db.Log = Console.Out;
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
}

```

```

foreach (Order order in cust.Orders)
{
    Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
}

```

Запустив этот пример нажатием <Ctrl+F5>, я получу несколько усеченный вывод:

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [AROUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
    10355 11/15/1996 12:00:00 AM
    10383 12/16/1996 12:00:00 AM
    10453 2/21/1997 12:00:00 AM
    10558 6/4/1997 12:00:00 AM
    10707 10/16/1997 12:00:00 AM
    10741 11/14/1997 12:00:00 AM
    10743 11/17/1997 12:00:00 AM
    10768 12/8/1997 12:00:00 AM
    10793 12/24/1997 12:00:00 AM
    10864 2/2/1998 12:00:00 AM
    10920 3/3/1998 12:00:00 AM
    10953 3/16/1998 12:00:00 AM
    11016 4/10/1998 12:00:00 AM
B's Beverages - Victoria Ashworth
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
    10289 8/26/1996 12:00:00 AM
    10471 3/11/1997 12:00:00 AM
    10484 3/24/1997 12:00:00 AM
    10538 5/15/1997 12:00:00 AM
    10539 5/16/1997 12:00:00 AM
    10578 6/24/1997 12:00:00 AM
    10599 7/15/1997 12:00:00 AM
    10943 3/11/1998 12:00:00 AM
    10947 3/13/1998 12:00:00 AM
    11023 4/14/1998 12:00:00 AM
Consolidated Holdings - Elizabeth Brown
...

```

Здесь я выделил SQL-запросы, чтобы отличить их от выходных данных заказчика и заказов. В первом SQL-запросе вы можете видеть, что там опрашиваются только заказчики, и ничто не обращается к таблице заказов. Затем вы можете видеть, что отображается имя компании и имя контакта для первой компании, а затем идет вывод второго SQL-запроса. Во втором SQL-запросе вы можете видеть, что идет опрос таблицы Orders с определенным значением CustomerID в конструкции where. Поэтому запрос генерируется и выполняется только для определенного заказчика, который отображен на консоли. Далее вы видите список заказов для последнего выведенного заказчика, за которым идет следующий заказчик. После этого появляется другой SQL-запрос заказов определенного заказчика.

Как видите, для извлечения заказов каждого заказчика выполняется отдельный запрос. Заказы не запрашиваются, а потому и не загружаются — до тех пор, пока не выполняется обращение к переменной Orders EntityRef<T> во втором цикле foreach, который следует немедленно после отображения на консоли информации о заказчике. Так как заказы не извлекаются до тех пор, пока к ним не будет выполнено обращение, их загрузка откладывается.

Поскольку для каждого заказчика генерируется и выполняется отдельный запрос, потенциально в базу данных приходит множество SQL-запросов. Это может привести к проблемам с производительностью.

Поскольку код написан мною, я знаю, что собираюсь обращаться к заказам извлеченных заказчиков. В данном случае можно было бы добиться лучшей производительности, если бы удалось извлекать заказы вместе с извлечением заказчиков. То, что мне нужно — это немедленная загрузка.

Немедленная загрузка с помощью класса DataLoadOptions

В то время как отложенная загрузка является поведением по умолчанию для ассоциированных классов, можно выполнять и немедленную загрузку. Немедленная загрузка заставляет ассоциированные классы загружаться до того, как к ним выполнится обращение. Это может дать выигрыш в производительности. Вы можете использовать операцию LoadWidth<T> класса DataLoadOptions для того, чтобы заставить DataContext немедленно загрузить ассоциированный класс, специфицированный в лямбда-выражении операции LoadWidth<T>. При использовании операции LoadWidth<T>, когда происходит действительное выполнение запроса, то при этом извлекается не только первичный класс, но также и ассоциированный с ним класс.

В листинге 14.9 я использую тот же базовый код примера, что и в листинге 14.8, но на этот раз после создания экземпляра объекта DataLoadOptions, я вызову операцию LoadWith<T> этого объекта DataLoadOptions, передав ей Orders как класс, подлежащий немедленной загрузке вместе с объектом Customer, и присвою объект DataLoadOptions экземпляру Northwind DataContext. Кроме того, чтобы исключить любые сомнения относительного того, что ассоциированные объекты — заказы загружаются до того, как к ним произойдет обращение, я пропущу код, выполняющий перечисление заказов текущего заказчика, чтобы не было никаких ссылок на них.

Листинг 14.9. Пример, демонстрирующий немедленную загрузку с использованием класса DataLoadOptions

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
db.LoadOptions = dlo;
IQueryable<Customer> custs = from c in db.Customers
                               where c.Country == "UK" &
                               c.City == "London"
```

```

        orderby c.CustomerID
        select c;
// Включить протоколирование.
db.Log = Console.Out;
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
}

```

Единственное отличие этого листинга от листинга 14.8 состоит в создании экземпляра объекта `DataLoadOptions`, вызове операции `LoadWith<T>`, присваивании объекта `DataLoadOptions` `Northwind` `DataContext` и исключении ссылок на заказы каждого заказчика. В вызове операции `LoadWith<T>` я инструктирую `DataLoadOptions` немедленно загружать `Orders` при загрузке объекта `Customer`. Теперь посмотрим на вывод программы из листинга 14.9.

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax], [t1].[OrderID], [t1].[CustomerID] AS
[CustomerID2], [t1].[EmployeeID], [t1].[OrderDate], [t1].[RequiredDate],
[t1].[ShippedDate], [t1].[ShipVia], [t1].[Freight], [t1].[ShipName],
[t1].[ShipAddress], [t1].[ShipCity], [t1].[ShipRegion], [t1].[ShipPostalCode],
[t1].[ShipCountry], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t2]
    WHERE [t2].[CustomerID] = [t0].[CustomerID]
) AS [count]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[Orders] AS [t1] ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID], [t1].[OrderID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - Thomas Hardy
B's Beverages - Victoria Ashworth
Consolidated Holdings - Elizabeth Brown
Eastern Connection - Ann Devon
North/South - Simon Crowther
Seven Seas Imports - Hari Kumar

```

Как и в выводе кода из листинга 14.8, в этом выводе я выделил полужирным запросы SQL. В действительности я не заинтересован в выводе заказчиков; я хотел увидеть запросы SQL, которые были выполнены.

Как видите, был выполнен единственный запрос SQL для извлечения заказчиков, отвечающих условию моей конструкции `where`. Вы можете также видеть, что, несмотря на тот факт, что я нигде не обращаюсь к заказам данного заказчика, этот единственный запрос соединяет каждого заказчика с его заказами. Поскольку заказы загружаются до обращения к ним, их загрузка не откладывается и выполняется немедленно. Вместо множества запросов SQL, количеством которых равно одному (запрос заказчиков) плюс число извлеченных заказчиков (для заказов каждого заказчика), здесь мы имеем дело с одним SQL-запросом. Если заказчиков достаточно много, разница в производительности будет огромной.

Используя класс `DataLoadOptions`, вы не ограничены немедленной загрузкой объектов единственного ассоциированного класса или единственным иерархическим уровнем ассоциированных классов. Однако немедленная загрузка более одного ассоциированного класса отражается на ее производительности.

Когда немедленная загрузка не такая уж и немедленная

Когда классы не загружаются до того, как к ним будет выполнено обращение, говорят, что их загрузка отложена. Если же они загружаются до обращения, то их загрузку называют немедленной. Однако иногда немедленная загрузка является не совсем немедленной.

На примере кода из листинга 14.9 мы видели, что, специфицируя ассоциированный класс как аргумент метода `LoadWith<T>` класса `DataLoadOptions`, мы можем заставить загружать заказы вместе с заказчиками. Это не только не является отложенной загрузкой, поскольку происходит до первого обращения к ассоциированным объектам, но также эту загрузку можно было назвать действительно немедленной. Если мы вызовем метод `LoadWith<T>` много раз, чтобы немедленно загружать несколько классов, только один из этих классов будет соединен в запросе с исходным сущностным классом, а другие будут загружаться при ссылке на этот исходный сущностный класс. Когда такое случается, поскольку ассоциированные классы, не соединенные с оригинальным сущностным классом, все же загружаются до первой ссылки на них, они все равно считаются немедленно загружаемыми, хотя для их получения выполняются отдельные запросы по мере обращения к объектам исходного сущностного класса. Таким образом, хотя их загрузка также считается немедленной, на самом деле она не происходит столь немедленно, как в случае их объединения с исходным классом в едином запросе.

Решение о том, какой ассоциированный класс должен быть соединен в запросе с каким, чтобы загружаться до первого обращения, принимается LINQ to SQL. Это решение оптимизировано на основе общих принципов, применимых к вашей модели классов, т.е. это не имеет отношения к оптимизации, выполняемой базой данных. Оно соединяет самую нижнюю ассоциацию в иерархии немедленно загружаемых классов. Все станет понятнее, когда я перейду к разделу, посвященному немедленной загрузке иерархии ассоциированных классов.

Чтобы лучше понять это поведение, я рассмотрю это для каждого подхода, где немедленно загружается более одной ассоциации. Эти два подхода следующие: загрузка множества ассоциированных классов одного оригинального сущностного класса либо загрузка иерархии ассоциированных классов.

Немедленная загрузка множества ассоциированных классов

С помощью класса `DataLoadOptions` можно запросить немедленную загрузку множества классов, ассоциированных с одним сущностным классом.

Обратите внимание, что в листинге 14.9 сгенерированный запрос SQL не ссылается на ассоциированную с заказчиком демографию. Если бы я ссылался на демографию извлекаемых заказчиков, то выполнялись бы дополнительные конструкции SQL для каждого заказчика, для которого были запрошены демографические данные.

В листинге 14.10 я потребую через `DataLoadOptions` немедленно загрузить демографические данные заказчика наряду с его заказами.

Листинг 14.10. Немедленная загрузка множества `EntityType`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
dlo.LoadWith<Customer>(c => c.CustomerCustomerDemos);
db.LoadOptions = dlo;
IQueryable<Customer> custs = from c in db.Customers
    where c.Country == "UK" &&
        c.City == "London"
    orderby c.CustomerID
    select c;
```

```
// Включить протоколирование.
db.Log = Console.Out;
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
}
```

В листинге 14.10 я не только специфицирую немедленную загрузку заказов, но также и немедленную загрузку демографических данных. Поэтому любая загрузка этих ассоциированных классов произойдет немедленно, а не будет отложена. В действительности я не так заинтересован в возвращаемых данных, как в выполняемых конструкциях SQL. Рассмотрим вывод кода из листинга 14.10.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax], [t1].[CustomerID] AS [CustomerID2],
[t1].[CustomerTypeID], (
    SELECT COUNT(*)
    FROM [dbo].[CustomerCustomerDemo] AS [t2]
    WHERE [t2].[CustomerID] = [t0].[CustomerID]
) AS [count]
FROM [dbo].[Customers] AS [t0]
LEFT OUTER JOIN [dbo].[CustomerCustomerDemo] AS [t1] ON [t1].[CustomerID] =
[t0].[CustomerID]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID], [t1].[CustomerTypeID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @x1
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [AROUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry]
FROM [dbo].[Orders] AS [t0]
WHERE [t0].[CustomerID] = @x1
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
B's Beverages - Victoria Ashworth
...
```

Как можно видеть в сгенерированных SQL-запросах, демография заказчика соединена с заказчиками во время запроса, но для загрузки заказов каждого заказчика генерируется отдельный запрос SQL. Этот отдельный запрос заказов выполняется тогда, когда выполняется обращение к каждому заказчику, т.е. в операторе foreach. Обратите внимание, что в выводе запросов заказчика выведен перед информацией о заказчике, отображаемой на консоли.

Поскольку ни демография, ни заказы не имеют ссылок в коде, кроме как при вызове метода LoadWith<T>, поскольку они действительно загружаются, эта загрузка не

является отложенной, а потому является немедленной. Однако определенно возникает ощущение, что демография заказчика несколько в больше степени немедленная, чем его заказы.

Немедленная загрузка иерархии ассоциированных классов

В предыдущем разделе я описал, как заставить множество ассоциированных сущностных классов загружаться немедленно. Здесь я опишу, как заставить немедленно загружаться иерархию ассоциированных сущностных классов. Чтобы продемонстрировать это, в листинге 14.11 я запрошу не только немедленную загрузку заказов, но также и всех деталей этих заказов.

Листинг 14.11. Немедленная загрузка иерархии сущностных классов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
DataLoadOptions dlo = new DataLoadOptions();
dlo.LoadWith<Customer>(c => c.Orders);
dlo.LoadWith<Order>(o => o.OrderDetails);
db.LoadOptions = dlo;
IQueryable<Customer> custs = from c in db.Customers
                                where c.Country == "UK" &&
                                      c.City == "London"
                                orderby c.CustomerID
                                select c;
// Включить протоколирование.
db.Log = Console.Out;
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
    foreach (Order order in cust.Orders)
    {
        Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
    }
}
```

Обратите внимание, что в листинге 14.11 я немедленно загружаю заказы заказчика, и для каждого заказа — его детали. Вывод кода из листинга 14.11 показан ниже.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry], [t1].[OrderID] AS [OrderID2],
[t1].[ProductID], [t1].[UnitPrice], [t1].[Quantity], [t1].[Discount], (
    SELECT COUNT(*)
    FROM [dbo].[Order Details] AS [t2]
    WHERE [t2].[OrderID] = [t0].[OrderID]
) AS [count]
FROM [dbo].[Orders] AS [t0]
LEFT OUTER JOIN [dbo].[Order Details] AS [t1] ON [t1].[OrderID] = [t0].[OrderID]
WHERE [t0].[CustomerID] = @x1
```

```

ORDER BY [t0].[OrderID], [t1].[ProductID]
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [APOUT]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - Thomas Hardy
SELECT [t0].[OrderID], [t0].[CustomerID], [t0].[EmployeeID], [t0].[OrderDate],
[t0].[RequiredDate], [t0].[ShippedDate], [t0].[ShipVia], [t0].[Freight],
[t0].[ShipName], [t0].[ShipAddress], [t0].[ShipCity], [t0].[ShipRegion],
[t0].[ShipPostalCode], [t0].[ShipCountry], [t1].[OrderID] AS [OrderID2],
[t1].[ProductID], [t1].[UnitPrice], [t1].[Quantity], [t1].[Discount], (
    SELECT COUNT(*)
    FROM [dbo].[Order Details] AS [t2]
    WHERE [t2].[OrderID] = [t0].[OrderID]
) AS [count]
FROM [dbo].[Orders] AS [t0]
LEFT OUTER JOIN [dbo].[Order Details] AS [t1] ON [t1].[OrderID] = [t0].[OrderID]
WHERE [t0].[CustomerID] = @x1
ORDER BY [t0].[OrderID], [t1].[ProductID]
-- @x1: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [BSBEV]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
B's Beverages - Victoria Ashworth
...

```

Опять-таки, здесь меня не интересуют извлеченные данные, а интересуют выполняющиеся запросы SQL. Обратите внимание, что на этот раз запрос заказчиков не соединяется ни с заказами, ни с их деталями. Вместо этого при ссылке на каждого заказчика выполняется дополнительный запрос SQL, который соединяет заказы и их детали. Поскольку ни то, ни другое не имеет ссылок в коде, они загружаются до ссылок и потому рассматриваются как загружаемые немедленно.

На этом примере вы можете видеть, что LINQ to SQL выполняет единственное соединение ассоциации нижнего уровня иерархии немедленно загружаемых классов, как я упоминал ранее.

Фильтрация и упорядочивание

Говоря о классе `DataLoadOptions`, я хочу, чтобы вы помнили о методе `AssociateWith`, который может быть использован как для фильтрации ассоциированных дочерних объектов, так и для их упорядочивания.

В листинге 14.8 я извлек несколько заказчиков и перечислил их, отображая информацию о них и их заказах. Вы можете видеть, что в результате даты заказов упорядочены по возрастанию. Чтобы продемонстрировать, как метод `AssociateWith` может быть использован и для фильтрации, и для упорядочивания ассоциированных классов, в листинге 14.12 я сделаю и то, и другое.

Листинг 14.12. Использование класса `DataLoadOptions` для фильтрации и упорядочивания

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
DataLoadOptions dlo = new DataLoadOptions();
dlo.AssociateWith<Customer>(c => from o in c.Orders
                                where o.OrderID < 10700
                                orderby o.OrderDate descending
                                select o);

db.LoadOptions = dlo;
IQueryable<Customer> custs = from c in db.Customers
                                where c.Country == "UK" &&
                                c.City == "London"
                                orderby c.CustomerID
                                select c;

```

```

foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CompanyName, cust.ContactName);
    foreach (Order order in cust.Orders)
    {
        Console.WriteLine(" {0} {1}", order.OrderID, order.OrderDate);
    }
}

```

Обратите внимание, что в листинге 14.12 я встроил в запрос лямбда-выражение, переданное методу AssociateWith. В этом запросе я фильтрую все записи, в которых OrderID меньше 10700, и сортирую их в порядке убывания OrderDate. Рассмотрим результат запуска листинга 14.12.

```

Around the Horn - Thomas Hardy
10558 6/4/1997 12:00:00 AM
10453 2/21/1997 12:00:00 AM
10383 12/16/1996 12:00:00 AM
10355 11/15/1996 12:00:00 AM

B's Beverages - Victoria Ashworth
10599 7/15/1997 12:00:00 AM
10578 6/24/1997 12:00:00 AM
10539 5/16/1997 12:00:00 AM
10538 5/15/1997 12:00:00 AM
10484 3/24/1997 12:00:00 AM
10471 3/11/1997 12:00:00 AM
10289 8/26/1996 12:00:00 AM

Consolidated Holdings - Elizabeth Brown
10462 3/3/1997 12:00:00 AM
10435 2/4/1997 12:00:00 AM

Eastern Connection - Ann Devon
10532 5/9/1997 12:00:00 AM
10400 1/1/1997 12:00:00 AM
10364 11/26/1996 12:00:00 AM

North/South - Simon Crowther
10517 4/24/1997 12:00:00 AM

Seven Seas Imports - Hari Kumar
10547 5/23/1997 12:00:00 AM
10523 5/1/1997 12:00:00 AM
10472 3/12/1997 12:00:00 AM
10388 12/19/1996 12:00:00 AM
10377 12/9/1996 12:00:00 AM
10359 11/21/1996 12:00:00 AM

```

Как видите, возвращены только заказы с OrderID меньше 10700, и возвращены они в порядке убывания дат.

Случайные соединения

Одним из преимуществ ассоциаций является то, что они, по сути, выполняют соединения автоматически. Когда мы запрашиваем заказчиков из базы данных Northwind, каждый заказчик имеет коллекцию заказов, доступных через свойство Orders объекта Customer. Таким образом, извлечение заказов для заказчиков выполняется автоматически. Обычно вам пришлось бы выполнять соединение для получения такого поведения. Обратное также верно. Когда мы извлекаем заказы, класс Order имеет свойство Customer, ссылающееся на соответствующего заказчика.

Когда происходит автоматическое соединение, то на самом деле это, как сказал бы покойный художник Боб Росс, просто маленькая счастливая случайность. Соединение

случается потому, что у нас есть объект, скажем, дочерний объект, имеющий отношение с другим объектом, например, родительским, и мы ожидаем от него возможности доступа через ссылку на исходный дочерний объект.

Например, при работе с XML, когда у нас есть ссылка на узел, мы ожидаем получения ссылки на его родительской узел через переменную-член дочернего узла, ссылющуюся на родителя. Мы не готовы к тому, что придется выполнять запрос по всей структуре XML, передавая дочерний узел в качестве ключа поиска. К тому же, когда у нас есть ссылка на узел, мы ожидаем возможности доступа к его дочерним узлам со ссылками на него самого.

Поэтому, хотя автоматическое соединение, несомненно, удобно, реализация может по-своему обращаться с природой отношений, и наши ожидания относительно того, как они должны себя вести, определяют стремление к автоматическому установлению соединений. Поэтому соединения являются в определенной степени случайными.

Соединения

Мы только что обсудили тот факт, что многие отношения в базе данных специфицированы как ассоциации, и что мы можем получить доступ к ассоциированным объектам, просто обращаясь к членам класса. Однако отображаться подобным образом могут только те отношения, которые определены с использованием внешних ключей. Поскольку не каждый тип отношения определен с применением внешних ключей, иногда вам придется явно соединять таблицы.

Внутренние соединения

Внутреннее соединение по эквивалентности можно выполнить с помощью операции `join`. Как это принято при внутреннем соединении, любые записи во внешнем результирующем наборе исключаются, если не существуют связанные с ними записи во внутреннем результирующем наборе. Пример приведен в листинге 14.13.

Листинг 14.13. Выполнение внутреннего соединения

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var entities = from s in db.Suppliers
    join c in db.Customers on s.City equals c.City
    select new
    {
        SupplierName = s.CompanyName,
        CustomerName = c.CompanyName,
        City = c.City
    };
foreach (var e in entities)
{
    Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

В листинге 14.13 выполняется внутреннее соединение поставщиков и потребителей (заказчиков). Если запись о заказчике из того же города, что и у поставщика, не существует, то запись о поставщике будет исключена из результирующего набора. Ниже показан результат работы кода из листинга 14.13.

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
```

Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
 Sao Paulo: Refrescos Americanas LTDA - Família Arquibaldo
 Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
 Sao Paulo: Refrescos Americanas LTDA - Tradição Hipermercados
 Berlin: Heli Süßwaren GmbH & Co. KG - Alfred Futterkiste
 Paris: Aux joyeux ecclésiastiques - Paris spécialités
 Paris: Aux joyeux ecclésiastiques - Spécialités du monde
 Montréal: Ma Maison - Mère Paillarde

Как видите, несмотря на тот факт, что некоторым поставщикам в выводе соответствует по несколько заказчиков, некоторых поставщиков вообще нет в списке. Это потому, что не найдено поставщиков в тех городах, откуда эти пропущенные заказчики. Если нужно видеть поставщиков независимо от того, есть ли в том же городе заказчики или нет, то нам понадобится внешне соединение.

Внешние соединения

В главе 4 я рассказывал о стандартной операции запроса DefaultIfEmpty и упомянул, что он может быть использован для выполнения соединений. В листинге 14.14 я использую конструкцию `into` для направления соответствующих результатов `join` во временную последовательность, на которой затем вызову операции `DefaultIfEmpty`. Таким образом, если запись пропущена в соединенном результате, то будет подставлено значение по умолчанию. Я использую средство протоколирования `DataContext`, чтобы можно было увидеть сгенерированный оператор SQL.

Листинг 14.14. Выполнение внешнего соединения

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var entities =
  from s in db.Suppliers
  join c in db.Customers on s.City equals c.City into temp
  from t in temp.DefaultIfEmpty()
  select new
  {
    SupplierName = s.CompanyName,
    CustomerName = t.CompanyName,
    City = s.City
  };
foreach (var e in entities)
{
  Console.WriteLine("{0}: {1} - {2}", e.City, e.SupplierName, e.CustomerName);
}
```

Обратите внимание, что в конструкции `join` в листинге 14.14 я направил результаты соединения во временную последовательность по имени `temp`. Имя временной последовательности может быть любым, какое вам нравится, до тех пор, пока оно не конфликтует ни с одним другим именем или ключевым словом. Затем я выполняю последующий запрос результатов на последовательности `temp`, передавая его операции `DefaultEmpty`. Даже несмотря на то, что я еще описал его, отмечу, что операция `DefaultEmpty`, вызванная в листинге 14.14, — это не та же самая операция, о которой шла речь в главе 4. Как я вскоре объясню, запросы LINQ to SQL транслируются в операторы SQL, и эти операторы SQL выполняются базой данных. SQL Server не имеет никакой возможности вызвать стандартную операцию запроса `DefaultEmpty`. Вместо этого данный вызов операции транслируется в соответствующий оператор SQL. Вот почему мне понадобилось включить средство протоколирования `DataContext`.

К тому же заметьте, что я обращаюсь к наименованию города из таблицы Suppliers, а не из коллекции temp. Я поступаю так потому, что знаю, что всегда должна быть запись о поставщике, но для поставщиков, не имеющих соответствующих заказчиков, не будет значения города в присоединенном результате из коллекции temp. Это отличается от предыдущего примера внутреннего соединения, где я получал город из присоединенной таблицы. В том примере не имело значения, из какой таблицы брать город, потому что если соответствующего заказчика не было найдено, то не было бы и соответствующей результирующей записи, поскольку соединение было внутренним.

Взглянем на результат работы листинга 14.14.

```
SELECT [t0].[CompanyName], [t1].[CompanyName] AS [value], [t0].[City]
FROM [dbo].[Suppliers] AS [t0]
LEFT OUTER JOIN [dbo].[Customers] AS [t1] ON [t0].[City] = [t1].[City]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
New Orleans: New Orleans Cajun Delights -
Ann Arbor: Grandma Kelly's Homestead -
Tokyo: Tokyo Traders -
Oviedo: Cooperativa de Quesos 'Las Cabras' -
Osaka: Mayumi's -
Melbourne: Pavlova, Ltd. -
Manchester: Specialty Biscuits, Ltd. -
Göteborg: PB Knäckebröd AB -
Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
Sao Paulo: Refrescos Americanas LTDA - Tradição Hipermercados
Berlin: Heli Süßwaren GmbH & Co. KG - Alfreds Futterkiste
Frankfurt: Plutzer Lebensmittelgroßmärkte AG -
Cuxhaven: Nord-Ost-Fisch Handelsgesellschaft mbH -
Ravenna: Formaggi Fortini s.r.l. -
Sandvika: Norske Meierier -
Bend: Bigfoot Breweries -
Stockholm: Svensk Sjöföda AB -
Paris: Aux joyeux ecclésiastiques - Paris spécialités
Paris: Aux joyeux ecclésiastiques - Spécialités du monde
Boston: New England Seafood Cannery -
Singapore: Leka Trading -
Lyngby: Lyngbysild -
Zaandam: Zaanse Snoepfabriek -
Lappeenranta: Karkki Oy -
Sydney: G'day, Mate -
Montréal: Ma Maison - Mère Paillarde
Salerno: Pasta Buttini s.r.l. -
Montceau: Escargots Nouveaux -
Annecy: Gai pâturage -
Sté-Hyacinthe: Forêts d'érables -
```

Как видим из вывода листинга 14.14, я получаю как минимум одну запись для каждого поставщика, причем видно, что некоторые поставщики не имеют соответствующего заказчика, поскольку выполнено внешнее соединение. Но если у вас возникнут какие-то сомнения, вы можете взглянуть на генерированный оператор SQL, который явно указывает на выполнение внешнего соединения.

Упрощать или не упрощать

В примере из листингов 14.13 и 14.14 я проектирую результаты моего запроса на плоскую структуру. Под этим я понимаю объект анонимного класса, в котором каждое запрошенное поле становится членом этого класса. В противоположность этому можно было бы вместо создания единственного анонимного класса, содержащего все нужные мне поля, я мог бы создать анонимный класс, состоящий из объекта `Supplier` и соответствующего ему объекта `Customer`. В данном случае был бы верхний уровень анонимного класса, и нижний уровень, состоящий из объекта `Supplier` и либо соответствующего ему объекта `Customer`, либо объекта по умолчанию, представленного операцией `DefaultIfEmpty`, который был бы равен `null`.

Если принять “плоский” подход, как я сделал в предыдущих примерах, поскольку спроектированный выходной класс не был существенным классом, я не мог выполнять обновления выходных объектов через объект `DataContext`, который управляет сохранением изменений в базе данных. Это хорошо для данных, которые не подвергаются изменениям. Однако иногда вам может понадобиться возможность изменения извлеченных объектов с сохранением изменений через `DataContext`. Более подробно я раскрою эту тему в главе 16. А пока давайте взглянем на листинг 14.15, который содержит “не плоский” пример.

Листинг 14.15. Возврат не плоского результата, допускающего сохранение изменений через `DataContext`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var entities = from s in db.Suppliers
               join c in db.Customers on s.City equals c.City into temp
               from t in temp.DefaultIfEmpty()
               select new { s, t };
foreach (var e in entities)
{
    Console.WriteLine("{0}: {1} - {2}", e.s.City,
                      e.s.CompanyName,
                      e.t != null ? e.t.CompanyName : "");
}
```

В листинге 14.15 вместо возврата результатов запроса в плоский анонимный объект, состоящий из всех нужных полей, я возвращаю результаты запроса в анонимном объекте, состоящем из объекта `Supplier` и, потенциально, из объекта `Customer`. Также обратите внимание, что в вызове метода `Console.WriteLine` я учитываю, что временный результат может быть `null`, если нет соответствующего объекта `Customer`. Давайте взглянем на результаты выполнения кода из листинга 14.15.

```
London: Exotic Liquids - Around the Horn
London: Exotic Liquids - B's Beverages
London: Exotic Liquids - Consolidated Holdings
London: Exotic Liquids - Eastern Connection
London: Exotic Liquids - North/South
London: Exotic Liquids - Seven Seas Imports
New Orleans: New Orleans Cajun Delights -
Ann Arbor: Grandma Kelly's Homestead -
Tokyo: Tokyo Traders -
Oviedo: Cooperativa de Quesos 'Las Cabras' -
Osaka: Mayumi's -
Melbourne: Pavlova, Ltd. -
Manchester: Specialty Biscuits, Ltd. -
Göteborg: PB Knäckebröd AB -
```

Sao Paulo: Refrescos Americanas LTDA - Comércio Mineiro
 Sao Paulo: Refrescos Americanas LTDA - Familia Arquibaldo
 Sao Paulo: Refrescos Americanas LTDA - Queen Cozinha
 Sao Paulo: Refrescos Americanas LTDA - Tradição Hipermercados
 Berlin: Heli Süßwaren GmbH & Co. KG - Alfreds Futterkiste
 Frankfurt: Plutzer Lebensmittelgroßmärkte AG -
 Cuxhaven: Nord-Ost-Fisch Handelsgesellschaft mbH -
 Ravenna: Formaggi Fortini s.r.l. -
 Sandvika: Norske Meierier -
 Bend: Bigfoot Breweries -
 Stockholm: Svensk Sjöföda AB -
 Paris: Aux joyeux ecclésiastiques - Paris spécialités
 Paris: Aux joyeux ecclésiastiques - Spécialités du monde
 Boston: New England Seafood Cannery -
 Singapore: Leka Trading -
 Lyngby: Lyngbysild -
 Zaandam: Zaanse Snoepfabriek -
 Lappeenranta: Karkki Oy -
 Sydney: G'day, Mate -
 Montréal: Ma Maison - Mère Paillarde
 Salerno: Pasta Buttini s.r.l. -
 Montceau: Escargots Nouveaux -
 Annecy: Gai pâturage -
 Ste-Hyacinthe: Forêts d'éables -

В выводе кода из листинга 14.15 вы можете видеть, что некоторые поставщики не имеют соответствующих им заказчиков в своих городах. В отличие от последовательности анонимных объектов, возвращенных запросом в листинге 14.14, анонимные объекты, возвращенные запросом из листинга 14.15, содержат сущностные объекты типа `Supplier` и `Customer`. Поскольку это сущностные объекты, я могу воспользоваться преимуществом служб, предоставленных `DataContext` для управления изменениями в них и сохранения их в базе данных.

Отложенное выполнение запроса

Я знаю, что вы, наверно, уже раз десять прочитали мои объяснения относительно отложенного выполнения запросов. Но из-за присущей мне паранойи я все время опасаюсь, что вы можете пропустить некоторую важную часть предыдущей главы. И в данном случае я боюсь, что вы могли пропустить объяснение отложенного выполнения запросов.

Отложенное выполнение запросов объясняется тем фактом, что запрос LINQ любого типа — будь то запрос LINQ to SQL, запрос LINQ to XML или запрос LINQ to Object — может и не выполняться в точке его определения. Возьмем, к примеру, следующий запрос:

```
IQueryable<Customer> custs = from c in db.Customers
                                where c.Country == "UK"
                                select c;
```

Запрос базы данных на самом деле не выполняется при выполнении этого оператора; он просто определяется и присваивается переменной `custs`. Запрос не будет выполнен до тех пор, пока не начнется перечисление последовательности `custs`. Это влечет за собой ряд последствий.

Последствия отложенного выполнения запросов

Одним из последствий отложенного выполнения запросов является то, что ваш запрос может содержать ошибки, которые вызовут исключения, но только во время действительного выполнения запросов, а не во время его определения. Это может здорово изгуглять, когда вы проходите отладчиком через оператор запроса, и все идет хорошо,

а затем, много дальше в коде генерируется исключение при перечислении последовательности запроса. Или, возможно, вы вызываете другую операцию на последовательности запроса, что дает в результате новую последовательность, которая немедленно перечисляется.

Другим последствием является то, что поскольку запрос SQL выполняется при перечислении последовательности запроса, многократное ее перечисление ведет к многократному выполнению запроса SQL. Это определенно может плохо отразиться на производительности. Чтобы предотвратить это, нужно вызвать на последовательности одну из стандартных операций запросов преобразования — `ToArrayList<T>`, `ToDictionary<T, K>` или `ToLookup<T, K>`. Каждая из этих операций преобразует последовательность, на которой она вызвана, в структуру данных специфицированного типа, что в результате приведет к кэшированию результата. Вы можете многократно выполнять перечисление этой новой структуры данных, не вызывая тем самым повторного выполнения SQL-запроса с потенциальной вероятностью изменения результатов.

Использование преимуществ отложенного выполнения запросов

Одним из преимуществ отложенного выполнения запросов является повышение производительности и в то же время возможность повторного использования ранее определенных запросов. Поскольку запрос выполняется каждый раз, когда выполняется перечисление возвращаемой им последовательности, вы можете определить его один раз, а перечислять снова и снова, как только того потребует ситуация. И если поток кода идет по некоторому пути, который не требует просмотра результатов запроса посредством его перечисления, то производительность увеличивается за счет того, что запрос не выполняется.

Другая выгода от отложенного выполнения запросов заключается в том, что поскольку запрос на самом деле не выполняется при его определении, можно добавлять дополнительные операции программно по мере необходимости. Представьте приложение, которое позволяет пользователю запрашивать информацию о заказчиках. Также представьте, что пользователь может фильтровать запрошенный список заказчиков. Нарисуйте в воображении один из возможных вариантов интерфейса с фильтрацией, который имеет выпадающий список для каждого столбца таблицы заказчиков. Допустим, есть выпадающий список для столбца City и другой — для столбца Country. Каждый выпадающий список содержит все города и все страны из всех записей Customer в базе данных. В вершине каждого списка есть опция [ALL], принятая по умолчанию для соответствующего ему столбца данных. Если пользователь не изменяет состояния любого из этих выпадающих списков, никакой дополнительной конструкции `where` не добавляется к запросу для соответствующего столбца. Листинг 14.16 содержит пример программного построения запроса для такого интерфейса.

Листинг 14.16. Программное построение запроса

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Включить протоколирование.
db.Log = Console.Out;
// Представьте, что значения, приведенные ниже, не закодированы жестко,
// а вместо этого получаются из выбранных значений некоторых выпадающих списков.
string dropdownListCityValue = "Cowes";
string dropdownListCountryValue = "UK";
IQueryable<Customer> custs = (from c in db.Customers
                                select c);
if (!dropdownListCityValue.Equals("[ALL]"))
{
    custs = from c in custs
            where c.City == dropdownListCityValue
```

```

    select c;
}

if (!dropdownCountryValue.Equals("[ALL]"))
{
    custs = from c in custs
        where c.Country == dropdownCountryValue
        select c;
}

foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1} - {2}", cust.CompanyName, cust.City, cust.Country);
}

```

В листинге 14.16 я эмулировал получение выбранного пользователем города и страны из выпадающих списков, и только если они не установлены в "[ALL]", то я добавляю к запросу дополнительную операцию `where`. Поскольку запрос на самом деле не выполняется до того, как начнется перечисление возвращенной им последовательности, я могу строить его по частям.

Взглянем на результат работы кода из листинга 14.16:

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[City] = @p1)
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- @p1: Input String (Size = 5; Prec = 0; Scale = 0) [Cowes]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Island Trading - Cowes - UK

```

Обратите внимание, что поскольку я указал, что выбран город Cowes и страна UK, то и получил записи о заказчиках из Cowes, United Kingdom. Также обратите внимание, что здесь выполняется единственный оператор SQL. Поскольку выполнение запроса отложено до момента, когда в нем возникнет необходимость, я могу продолжать добавлять условия к запросу, чтобы дополнительно ограничить его, или упорядочить результат, без каких-либо затрат на множественные запросы SQL.

Вы можете видеть, что оба критерия фильтрации — город и страна — появляются в условии `where` выполняющегося оператора SQL.

Для другого теста — в листинге 14.17 — я изменю значение переменной `dropdownListCityValue` на "[ALL]" и посмотрю, как будет выглядеть выполняющийся оператор запроса SQL и каковы будут полученные результаты. Поскольку специфицирован город по умолчанию "[ALL]", запрос SQL не должен даже ограничивать результирующий набор никаким городом.

Листинг 14.17. Программное построение другого запроса

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Включить протоколирование.
db.Log = Console.Out;
// Представьте, что значения, приведенные ниже, не закодированы жестко,
// а вместо этого получаются из выбранных значений выпадающих списков.
string dropdownListCityValue = "[ALL]";
string dropdownListCountryValue = "UK";
IQueryable<Customer> custs = (from c in db.Customers
                                select c);

```

```

if (!dropdownListCityValue.Equals("[ALL]"))
{
    custs = from c in custs
            where c.City == dropdownListCityValue
            select c;
}
if (!dropdownListCountryValue.Equals("[ALL]"))
{
    custs = from c in custs
            where c.Country == dropdownListCountryValue
            select c;
}
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1} - {2}", cust.CompanyName, cust.City, cust.Country);
}

```

Ниже показан вывод кода из листинга 14.17:

```

SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Country] = @p0
-- @p0: Input String (Size = 2; Prec = 0; Scale = 0) [UK]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Around the Horn - London - UK
B's Beverages - London - UK
Consolidated Holdings - London - UK
Eastern Connection - London - UK
Island Trading - Cowes - UK
North/South - London - UK
Seven Seas Imports - London - UK

```

Вы можете видеть, что часть `where` оператора запроса SQL более не специфицирует город, что нам и требовалось. Также вы можете видеть, что в выводе результата теперь присутствуют заказчики из разных городов Соединенного Королевства.

Конечно, вы всегда можете добавить вызов стандартных операций запросов `ToArrayList<T>`, `ToDictionary<T, K>` или `ToLookup<T, K>`, чтобы принудительно выполнить запрос тогда, когда вы этого хотите.

Получение SQL-конструкции IN с помощью операции Contains

Одной из возможностей операций SQL, которых недоставало ранним воплощением LINQ to SQL, была способность выполнять SQL-конструкцию `IN` — вроде той, что представлена в следующем SQL-запросе:

Запрос SQL с конструкцией IN

```

SELECT *
FROM Customers
WHERE (City IN ('London', 'Madrid'))

```

Для решения этой проблемы была добавлена операция `Contains`. Эта операция, однако, используется иначе, чем может показаться на первый взгляд. На мой взгляд, она работает как раз наоборот по сравнению с тем, чего я ожидал от нее, как от реализации SQL-конструкции `IN`. Я ожидал, что она позволит сказать, что некоторый член сущностного класса должен быть в (`IN`) некотором наборе значений. Вместо этого она раб-

тает в обратной манере. Взглянем на листинг 14.18, где демонстрируется применение операции Contains.

Листинг 14.18. Применение операции Contains

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
string[] cities = { "London", "Madrid" };
IQueryable<Customer> custs = db.Customers.Where(c => cities.Contains(c.City));
foreach (Customer cust in custs)
{
    Console.WriteLine("{0} - {1}", cust.CustomerID, cust.City);
}
```

Как видно в листинге 14.18, вместо написания запроса так, что город заказчика должен был быть в некотором наборе значений, вы записываете запрос так, что некоторый набор значений содержит город заказчика. В случае листинга 14.18 я создал массив городов по имени cities. В моем запросе затем я вызываю операцию Contains на массиве cities и передаю ей город заказчика. Если массив cities содержит город заказчика, будет возвращено значение true в операции Where, что вызовет включение объекта Customer в выходную последовательность.

Ниже показан вывод кода из листинга 14.18.

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] IN (@p0, @p1)
-- @p0: Input String (Size = 6; Prec = 0; Scale = 0) [London]
-- @p1: Input String (Size = 6; Prec = 0; Scale = 0) [Madrid]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
AROUT - London
BOLID - Madrid
BSBEV - London
CONSH - London
EASTC - London
FISSA - Madrid
NORTS - London
ROMEY - Madrid
SEVES - London
```

Взглянув на сгенерированный оператор SQL, вы можете видеть, что операция Contains была транслирована в SQL-конструкцию IN.

Обновления

Проведение обновлений в базе данных посредством LINQ to SQL столь же просто, как изменение свойств объекта, вызов метода SubmitChanges объекта DataContext и последующая обработка возможных конфликтов параллельного доступа. Пусть вопрос обработки конфликтов параллельного доступа вас не пугает — существует несколько вариантов обработки подобных конфликтов, и ни один из них не является слишком болезненным. Тему обнаружения и обработки конфликтов параллелизма я рассмотрю в деталях в главе 17.

Конечно, все это так просто только в том случае, если вы правильно написали сущностные классы, корректно отображаемые на базу данных, выдерживая согласованность графа. За дополнительной информацией об отображении сущностных классов на базу данных обращайтесь в раздел под названием “Атрибуты сущностных классов и свой-

ства атрибутов" главы 15. Информацию о согласованности графа вы найдете в разделе "Согласованность графа" той же главы. Однако SQLMetal и Object Relational Designer выполняют всю необходимую работу за вас.

Простой пример проведения обновления в базе данных приведен в листинге 12.1 главы 12.

Обновление ассоциированных классов

В соответствии с дизайном LINQ to SQL позволяет вам обновлять любую сторону ассоциированных классов, чтобы удалить отношение между ними. Вы можете обновить ссылку родительского объекта на один из его дочерних объектов либо обновить ссылку дочернего объекта на родителя. Очевидно, что ссылки на каждом конце отношения должны обновляться, но при этом вам нужно только обновить одну или другую сторону.

О поддержке согласованности вашей объектной модели при обновлении одной из сторон заботится не LINQ to SQL; за это отвечает сущностный класс. Прочтите раздел "Согласованность графа" главы 15, чтобы получить больше сведений о том, как это должно быть реализовано.

Однако SQLMetal и Object Relational Designer обрабатывают это для вас, если вы получите им создать ваши сущностные классы.

Обновление ссылки на родителя у дочернего объекта

Поскольку мы можем обновить любую из сторон отношения, выберем вариант с обновлением ссылки на родителя дочернего объекта. Поэтому в качестве примера давайте посмотрим, как я мог бы изменить сотрудника, который получает кредит для заказа в базе данных Northwind, рассмотрев листинг 14.19. Поскольку этот пример более сложен, чем многие другие, я снабжу его последовательными пояснениями.

Листинг 14.19. Изменение отношения присвоением нового родителя

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Order order = (from o in db.Orders
               where o.EmployeeID == 5
               orderby o.OrderDate descending
               select o).First<Order>();
// Сохранить текущего сотрудника, чтобы восстановить его в конце.
Employee origEmployee = order.Employee;
```

В приведенном коде после получение DataContext я запрашиваю самый последний заказ сотрудника с EmployeeID равным 5, опрашивая заказы этого лица в порядке убывания дат и вызывая операцию First. Это даст мне самый свежий заказ. Затем, сохранив ссылку на оригинального сотрудника, которому был поручен этот заказ, я смогу восстановить его в конце примера. Я сохраняю эту ссылку в переменной по имени origEmployee.

```
Console.WriteLine("Before changing the employee.");
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
    .order.OrderID, order.OrderDate, order.Employee.EmployeeID);
```

Затем я отображаю строку на консоли, чтобы уведомить, что я еще не изменил сотрудника для извлеченного заказа, за которой следует идентификатор заказа и сотрудника. Мы должны увидеть, что заказ принят сотрудником с идентификатором 5, поскольку именно его заказ был запрошен.

```
Employee emp = (from e in db.Employees
                 where e.EmployeeID == 9
                 select e).Single<Employee>();
```

```
// Теперь я назначу нового сотрудника на заказ.
order.Employee = emp;
db.SubmitChanges();
```

Затем я запрашиваю некоторого другого сотрудника — того, у которого EmployeeID равно 9, чтобы назначить его ответственным за ранее извлеченный заказ. После этого сохраняю изменение вызовом метода SubmitChanges.

Теперь, чтобы доказать, что изменения действительно были проведены на обоих концах, я могу просто показать вам назначенного на данный заказ сотрудника, хотя это может показаться не совсем убедительным, поскольку установка свойства Employee заказа еще не доказывает изменения этого отношения на стороне заказчика. Было бы более наглядно найти только что измененный заказ в коллекции заказов нового сотрудника, что я и делаю.

```
Order order2 = (from o in emp.Orders
    where o.OrderID == order.OrderID
    select o).First<Order>();
```

В приведенном коде я запрашиваю измененный мною заказ по его OrderID в коллекции Orders нового заказчика. Если он будет найден, это докажет, что отношение между сотрудником и заказом было обновлено на обоих его концах.

```
Console.WriteLine("{0}After changing the employee.", System.Environment.NewLine);
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
    order2.OrderID, order2.OrderDate, order2.Employee.EmployeeID);
```

В приведенном коде я просто отображаю на консоли сообщение, говорящее о том, что ниже показан заказ после проведенного изменения — передачи его новому сотруднику emp. Затем я отображаю заказ. Теперь можно видеть, что EmployeeID сотрудника, которому он передан, равен 9. Ранее он был равен 5.

```
// Теперь мне нужно отменить изменения, чтобы пример можно было запускать многократно.
order.Employee = origEmployee;
db.SubmitChanges();
```

Последние две строки кода просто восстанавливают базу данных в том состоянии, которое она имела до этого, чтобы пример можно было запускать повторно.

Теперь посмотрим вывод кода из листинга 14.19.

```
Before changing the employee.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 5
After changing the employee.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 9
```

Как видите, сотрудником, которому был назначен заказ перед проведением изменения, был тот, у которого EmployeeID равен 5. После замены назначенного на заказ сотрудника его EmployeeID равен 9. Это доказывает, что заказ был изменен в отношении и со стороны сотрудника.

В данном примере я обновил ссылку на родителя в дочернем объекте, где дочерним выступает заказ, а родителем — сотрудник. Но есть и другой подход, посредством которого можно было получить тот же результат. Я мог бы обновить ссылку на дочерний объект у объекта-родителя.

Обновление ссылки на дочерний объект в родительском объекте

Другой подход к изменению отношения между двумя объектами заключается в удалении дочернего объекта из коллекции EntitySet<T> родительского объекта и добавлении его в коллекцию EntitySet<T> другого родительского объекта. В листинге 14.20 я удалю заказ из коллекции заказов сотрудника. Поскольку этот пример проще,

чем пример из листинга 14.19, я опущу пояснения, но существенные отличия выделю полужирным.

Листинг 14.20. Изменение отношения удалением и добавлением дочернего объекта в EntitySet родителя

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Order order = (from o in db.Orders
               where o.EmployeeID == 5
               orderby o.OrderDate descending
               select o).First<Order>();

// Сохранить текущего сотрудника, чтобы восстановить его в конце.
Employee origEmployee = order.Employee;
Console.WriteLine("Before changing the employee.");
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
    order.OrderID, order.OrderDate, order.Employee.EmployeeID);

Employee emp = (from e in db.Employees
                where e.EmployeeID == 9
                select e).Single<Employee>();

// Удалить заказ из Orders исходного сотрудника.
origEmployee.Orders.Remove(order);

// Теперь добавить его к заказам нового сотрудника.
emp.Orders.Add(order);
db.SubmitChanges();
Console.WriteLine("{0}After changing the employee.", System.Environment.NewLine);
Console.WriteLine("OrderID = {0} : OrderDate = {1} : EmployeeID = {2}",
    order.OrderID, order.OrderDate, order.Employee.EmployeeID);

// Теперь мне нужно отменить изменения, чтобы пример можно было запускать многократно.
order.Employee = origEmployee;
db.SubmitChanges();
```

В листинге 14.20 я извлекаю самый последний заказ для сотрудника с EmployeeID, равным 5, и сохраняю его в переменной `origEmployee`, чтобы восстановить в конце примера. Затем я отображаю заказ перед сменой сотрудника. После этого извлекаю сотрудника с EmployeeID, равным 9, и сохраняю ссылку на него в переменной `emp`. До этого момента код совпадает с листингом 14.19.

Затем я удаляю заказ из коллекции заказов исходного сотрудника и добавляю его в коллекцию заказов нового сотрудника. После чего вызываю метод `SubmitChanges`, чтобы сохранить изменения в базе данных. Далее после проведенных изменений я отображаю заказ на консоли. И, наконец, восстанавливая заказ в его исходном состоянии, чтобы можно было повторно запустить пример.

Взглянем на результат работы листинга 14.20.

```
Before changing the employee.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 5
After changing the employee.
OrderID = 11043 : OrderDate = 4/22/1998 12:00:00 AM : EmployeeID = 9
```

Удаления

Чтобы удалить запись из базы данных посредством LINQ to SQL, вы должны удалить существенный объект из `Table<T>`, членом которой он является, вызвав метод `DeleteOnSubmit` объекта `Table<T>`. Затем, конечно, нужно вызвать метод `SubmitChanges`. Пример приведен в листинге 14.21.

Внимание! В отличие от прочих примеров этой главы, этот пример не восстанавливает в конце базу данных в исходное состояние. Это потому, что одна из участвующих таблиц содержит столбец идентичности, и не так просто программно восстановить данные в состояние, предшествовавшее запуску примера. Поэтому прежде чем запустить пример, убедитесь, что у вас есть резервная копия базы данных, из которой вы сможете ее восстановить. Если вы выгрузили и упаковали расширенную версию базы данных Northwind, то после запуска этого примера можете просто отключить базу данных Northwind, заново извлечь файлы базы данных и снова подключить ее к серверу базы.

Листинг 14.21. Удаление записи посредством удаления ее из Table<T>

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Получить заказчика для удаления.
Customer customer = (from c in db.Customers
    where c.CompanyName == "Alfreds Futterkiste"
    select c).Single<Customer>();
db.OrderDetails.DeleteAllOnSubmit(
    customer.Orders.SelectMany(o => o.OrderDetails));
db.Orders.DeleteAllOnSubmit(customer.Orders);
db.Customers.DeleteOnSubmit(customer);
db.SubmitChanges();
Customer customer2 = (from c in db.Customers
    where c.CompanyName == "Alfreds Futterkiste"
    select c).SingleOrDefault<Customer>();
Console.WriteLine("Customer {0} found.", customer2 != null ? "is" : "is not");
```

Назад! В Visual Studio 2008 Beta 2 и более ранних выпусках метод DeleteOnSubmit из приведенного выше кода назывался Remove, а метод DeleteAllOnSubmit — RemoveAll.

Этот пример достаточно прямолинеен, но не лишен интересных аспектов. Во-первых, поскольку таблица Orders содержит внешний ключ, ссылающийся на таблицу Customers, вы не можете удалить заказчика без предварительного удаления всех его заказов. И поскольку таблица OrderDetails содержит внешний ключ, ссылающийся на таблицу Orders, вы не можете удалить заказ, не удалив его деталей. Таким образом, чтобы удалить заказчика, сначала я должен удалить детали всех его заказов, затем сами его заказы, и только потом — удалить заказчика.

Удаление всех заказов не представляет труда, благодаря операции DeleteAllOnSubmit, который может удалить последовательность заказов, но удаление деталей каждого заказа несколько сложнее. Конечно, я мог бы выполнить перечисление последовательности заказов, вызывая операцию DeleteAllOnSubmit на каждой последовательности деталей заказов, но это было бы утомительно. Вместо этого язываю операцию SelectMany, чтобы получить последовательность последовательностей деталей заказов и получить единую совокупную последовательность деталей заказов, которую затем передаю операции DeleteAllOnSubmit. Действительно, мощь LINQ просто поражает.

После удаления деталей заказов, самих заказов и их заказчика я просто вызываю метод SubmitChanges. Чтобы доказать, что заказчика больше нет, я запрашиваю его снова и показываю сообщение на консоли.

Посмотрим на вывод кода из листинга 14.21.

```
Customer is not found.
```

Вывод не слишком впечатляет, но он доказывает, что заказчика больше нет. Хотя в листинге 14.21 была демонстрация того, что для удаления сущностного объекта

вы должны удалить его из соответствующей коллекции `Table<T>`, этот пример также наглядно демонстрирует действие операции `SelectMany`.

На заметку! Напомню, что этот пример не восстанавливает базу данных в ее исходное состояние, так что вы должны восстановить ее вручную.

Удаление присоединенных сущностных объектов

В отличие от случая с автоматической вставкой в базу данных прикрепленных ассоциированных зависимых сущностных объектов посредством `DataContext`, как это было описано в листинге 14.3, когда вставлялся их родительский объект, в случае удаления родительского объекта зависимые объекты автоматически не удаляются. Под зависимыми я понимаю сущностные объекты, имеющие внешний ключ. Вы видели демонстрацию процесса удаления в листинге 14.21, где мне пришлось удалять записи `OrderDetails` перед записями `Orders`, а записи `Orders` — перед записью `Customers`.

Поэтому, например, с базой данных Northwind, если вы попытаетесь удалить заказ, то его детали не будут удалены автоматически. Попытка удаления заказа приведет к нарушению ограничения внешнего ключа. Поэтому перед тем, как удалить сущностный объект, вы должны удалить все прикрепленные к нему ассоциированные дочерние объекты.

В качестве примера этого процесса рассмотрите листинги 14.21 и 14.3. В каждом из них мне пришлось удалить ассоциированный прикрепленный сущностный объект до того, как удалять его родительский объект.

Удаление отношений

Чтобы удалить отношение между двумя сущностными объектами в LINQ to SQL, вы просто назначаете ссылке сущностного объекта, указывающей на связанный объект, значение `null`. Присваивая ссылке `null`, вы тем самым удаляете отношение данного сущностного объекта с сущностью этого типа. Однако удаление отношения присвоением `null` на самом деле не удаляет самой записи. Помните, что для действительного удаления записи соответствующий ей сущностный объект должен быть удален из соответствующей `Table<T>`. Листинг 14.22 содержит пример удаления отношения.

Листинг 14.22. Удаление отношения между двумя сущностными объектами

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Получить заказ для удаления отношения.
Order order = (from o in db.Orders
               where o.OrderID == 11043
               select o).Single<Order>();
// Сохранить оригинального заказчика, чтобы можно было восстановить обратно.
Customer c = order.Customer;
Console.WriteLine("Orders before deleting the relationship:");
foreach (Order ord in c.Orders)
{
    Console.WriteLine("OrderID = {0}", ord.OrderID);
}
// Удалить отношение с заказчиком.
order.Customer = null;
db.SubmitChanges();
Console.WriteLine("{0}Orders after deleting the relationship:",
    System.Environment.NewLine);
foreach (Order ord in c.Orders)
{
    Console.WriteLine("OrderID = {0}", ord.OrderID);
}
```

```
// Восстановить базу данных обратно в исходное состояние.
order.Customer = c;
db.SubmitChanges();
```

В листинге 14.22 я запрашиваю определенный заказ — тот, у которого OrderID равен 11043. Затем я сохраняю ссылку на заказчика этого заказа, чтобы восстановить его в конце примера. Далее я отображаю все заказы этого заказчика на консоли и присваиваю ссылке заказа на заказчика значение null, после чего вызываю метод `SubmitChanges` для сохранения изменений в базе данных. Затем я снова отображаю все заказы данного заказчика, и на этот раз в списке заказ с OrderID, равным 11043, отсутствует. Посмотрим на вывод кода из листинга 14.22.

`Orders before deleting the relationship:`

```
OrderID = 10738
OrderID = 10907
OrderID = 10964
OrderID = 11043
```

`Orders after deleting the relationship:`

```
OrderID = 10738
OrderID = 10907
OrderID = 10964
```

Как видите, как только я удалил отношение между заказчиком и заказом с OrderID, равным 11043, он исчез из коллекции заказов данного заказчика.

Переопределение операторов модификации базы данных

Если вы думали, что применение LINQ to SQL в вашем окружении невозможно, может быть, из-за требований обязательного применения хранимых процедур для проведения всех модификаций базы данных, то вам будет интересно узнать, что действительный код, вызываемый для проведения обновлений, включая вставки и удаления, может быть переопределен.

Переопределение кода, вызываемого для вставки, обновления и удаления столь же прост, как определение соответственно именованного частичного метода с соответствующей сигнатурой. Выполнив такое переопределение, вы заставляете процессор изменений `DataContext` вызывать вашу реализацию частичного метода для обновления, вставки и удаления записей базы данных. Это еще одно из преимуществ частичных методов, использованное Microsoft. Вы получаете возможность вмешаться в код без каких-либо накладных расходов, если вы ею не воспользуетесь.

Следует помнить, однако, что если вы принимаете этот подход, то берете на себя ответственность за обнаружение конфликтов параллельного доступа. Внимательно прочтите главу 17, прежде, чем брать на себя такую ответственность.

Определяя эти методы, вы задаете имя частичного метода и тип параметра существенного типа, которые инструктируют `DataContext` для их вызова в надлежащий момент. Давайте рассмотрим прототипы методов, которые вы должны определить для переопределения операций вставки, обновления и удаления.

Переопределение метода `Insert`

Вы можете переопределить метод, вызываемый для вставки записи в базу данных. Реализуя частичный метод со следующим прототипом:

```
partial void Insert[EntityClassName] (T instance)
```

где [EntityClassName] — имя сущностного класса, чей метод вставки переопределяется, а T — тип сущностного класса.

Вот пример переопределяемого прототипа метода вставки для сущностного класса Snipper:

```
partial void InsertShipper(Shipper instance)
```

Переопределение метода Update

Вы можете переопределить метод, вызываемый для обновления записи в базе данных, реализуя частичный метод со следующим прототипом:

```
partial void Update[EntityClassName](T instance)
```

где [EntityClassName] — имя сущностного класса, чей метод вставки переопределяется, а T — тип сущностного класса.

Вот пример переопределяемого прототипа метода обновления для сущностного класса Snipper:

```
partial void UpdateShipper(Shipper instance)
```

Переопределение метода Delete

Вы можете переопределить метод, вызываемый для удаления записи в базе данных, реализуя частичный метод со следующим прототипом:

```
partial void Delete[EntityClassName](T instance)
```

где [EntityClassName] — имя сущностного класса, чей метод вставки переопределяется, а T — тип сущностного класса.

Вот пример переопределяемого прототипа метода удаления для сущностного класса Snipper:

```
partial void DeleteShipper(Shipper instance)
```

Пример

Для примера, демонстрирующего методы вставки, обновления и удаления, вместо модификации моего сгенерированного файла сущностных классов я создам новый файл, специально для моих переопределенных частичных методов, чтобы в случае, если мне придется повторно генерировать файл сущностных классов, я не потерял мои переопределения частичных методов. Называться этот файл будет NorthwindExtended.cs. Выглядеть он будет так:

Мой файл NorthwindExtended.cs с переопределенными методами обновления базы данных

```
using System;
using System.Data.Linq;
namespace nwind
{
    public partial class Northwind : DataContext
    {
        partial void InsertShipper(Shipper instance)
        {
            Console.WriteLine("Insert override method was called for shipper {0}.",
                instance.CompanyName);
        }
    }
}
```

```

partial void UpdateShipper(Shipper instance)
{
    Console.WriteLine("Update override method was called for shipper {0}.",
        instance.CompanyName);
}
partial void DeleteShipper(Shipper instance)
{
    Console.WriteLine("Delete override method was called for shipper {0}.",
        instance.CompanyName);
}
}
}
}

```

На заметку! Вы должны добавить файл, содержащий частичные определения класса, в свой проект Visual Studio.

Первое, что нужно отметить относительно переопределенного кода — это тот факт, что переопределенные методы являются частичными методами, определенными на уровне `DataContext`. Они не определены в существенном классе, к которому относятся.

Как видите, мои переопределенные методы не делают ничего помимо того, что извещают меня о том, что они были вызваны. Во многих ситуациях переопределение нужно для вызова хранимых процедур, хотя это оставлено на усмотрении разработчика.

Теперь давайте взглянем на листинг 14.23, который содержит код, вызывающий мои переопределенные методы.

Листинг 14.23. Пример с переопределенными методами `Update`, `Insert` и `Delete`

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Shipper ship = (from s in db.Shippers
                where s.ShipperID == 1
                select s).Single<Shipper>();
ship.CompanyName = "Jiffy Shipping";
Shipper newShip =
    new Shipper
    {
        ShipperID = 4,
        CompanyName = "Vickey Rattz Shipping",
        Phone = "(800) SHIP-NOW"
    };
db.Shippers.InsertOnSubmit(newShip);
Shipper deletedShip = (from s in db.Shippers
                where s.ShipperID == 3
                select s).Single<Shipper>();
db.Shippers.DeleteOnSubmit(deletedShip);
db.SubmitChanges();

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод `InsertOnSubmit`, вы званный в предыдущем коде, назывался `Add`, а метод `DeleteOnSubmit` — `Remove`.

В листинге 14.23 я сначала извлекаю грузоотправителя, у которого `ShipperID` равен 1, и обновляю поле. Затем я вставляю другого грузоотправителя — `Vickey Rattz Shipping`, и еще одного удаляю — того, у которого `ShipperID` равно 3. Конечно, поскольку вызываются мои переопределенные методы, которые только выдают сообщение на консоль, ни одно из изменений не сохраняется в базе данных. Вот как выглядит результат работы листинга 14.23:

```
Update override method was called for shipper Jiffy Shipping.
Insert override method was called for shipper Vickey Rattz Shipping.
Delete override method was called for shipper Federal Shipping.
```

Из этого результата вы можете видеть, что был вызван каждый из моих переопределенных методов. Теперь возникает вопрос: что если вы хотите переопределить методы вставки, обновления и удаления, не отменяя поведения по умолчанию?

Поскольку необходимый код вступил бы в конфликт с моими частичными методами из предыдущего примера, я не стану приводить работающий пример этого, но объясню, как это сделать. В ваших реализациях частичных методов для вставки, обновления и удаления вы просто вызываете метод `DataContext.ExecuteDynamicInsert`, `DataContext.ExecuteDynamicUpdate` или `DataContext.ExecuteDynamicDelete` соответственно, чтобы получить поведение по умолчанию.

Например, если бы я захотел в моем предыдущем примере сначала вызвать протоколирование сообщений, а затем нормальный код LINQ to SQL, чтобы сохранить изменения в базе данных, я должен был изменить реализацию моего частичного метода следующим образом.

Переопределение методов Insert, Update и Delete, плюс поведение по умолчанию

```
namespace nwind
{
    public partial class Northwind : DataContext
    {
        partial void InsertShipper(Shipper instance)
        {
            Console.WriteLine("Insert override method was called for shipper {0}.",
                instance.CompanyName);
            this.ExecuteDynamicInsert(instance);
        }
        partial void UpdateShipper(Shipper instance)
        {
            Console.WriteLine("Update override method was called for shipper {0}.",
                instance.CompanyName);
            this.ExecuteDynamicUpdate(instance);
        }
        partial void DeleteShipper(Shipper instance)
        {
            Console.WriteLine("Delete override method was called for shipper {0}.",
                instance.CompanyName);
            this.ExecuteDynamicDelete(instance);
        }
    }
}
```

Обратите внимание, что в каждом из частичных методов язываю соответствующий метод `ExecuteDynamicInsert`, `ExecuteDynamicUpdate` или `ExecuteDynamicDelete`. Теперь я могу расширить поведение при вызове существенного класса, могу модифицировать его или даже создать оболочку для существующего поведения по умолчанию. LINQ to SQL весьма гибок.

Переопределение в Object Relational Designer

Не забывайте, что, как говорилось в главе 13, вы можете переопределить методы вставки, обновления и удаления, используя Object Relational Designer.

Соображения

Следует помнить, что когда вы переопределяете методы обновления, вставки и удаления, то тем самым берете на себя ответственность за обнаружение конфликтов параллельного доступа. Это значит, что вы должны хорошо разбираться в том, как работает текущая реализация обнаружения таких конфликтов. Например, Microsoft реализовала ее так, что требуется спецификация всех существенных полей, вовлеченных в обновление, в условии `where` оператора обновления. Затем логика проверяет, сколько записей было обновлено оператором. Если количество обновленных записей равно нулю, значит, имеет место конфликт параллельного доступа. Вы должны следовать аналогичному шаблону, и в случае обнаружения конфликта параллелизма генерировать исключение `ChangeConflictException`. Прочтите главу 17, прежде чем переопределять эти методы.

Трансляция SQL

При написании запросов LINQ to SQL вы, вероятно, заметили, что при специфировании выражений, таких как в конструкции `where`, используется "родной" язык программирования, а не SQL. В конце концов, это часть предназначения LINQ — языковая интеграция. В настоящей книге эти выражения пишутся на C#. Если вы этого не заметили, вам должно быть стыдно.

Например, в листинге 14.2 мой запрос выглядит так, как показано ниже.

Пример запроса LINQ to SQL

```
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
```

Обратите внимание, что выражение в конструкции `where` записано в синтаксисе C#, а не в синтаксисе SQL, который бы выглядел так:

Пример неправильного запроса Invalid LINQ to SQL

```
Customer cust = (from c in db.Customers
                  where c.CustomerID = 'LONEP'
                  select c).Single<Customer>();
```

Вместо операции эквивалентности C# (`==`) в SQL применяется операция эквивалентности (`=`). Строковый литерал SQL заключается не в двойные (" "), а в одинарные (' ') кавычки. Одной из целей LINQ является разрешение разработчикам программировать на своем родном языке программирования. Помните, что LINQ обозначает язык интегрированных запросов. Однако, поскольку база данных не выполняет выражений C#, эти ваши выражения должны быть транслированы в корректный SQL. Поэтому ваши запросы должны быть транслированы в SQL.

И это сразу означает, что то, что вы можете делать, имеет свои ограничения. Хотя в основном трансляция работает достаточно хорошо. Вместо того чтобы дублировать Руководство типа справочной системы MSDN для описания этого процесса трансляции с указанием того, что вы можете и чего не можете транслировать, я хотел бы показать, чего можно ожидать в тех случаях, когда ваш запрос LINQ to SQL не может быть транслирован.

Первым делом, имейте в виду, что код может скомпилироваться. Не расслабляйтесь, если ваш запрос успешно компилируется. Некорректная трансляция может не проявить себя до тех пор, пока запрос в действительности не будет выполнен. Из-за отложенного выполнения запроса это также означает, что строка кода, определяющая запрос, может выполниться успешно. И только когда дойдет дело до реального извлечения результата, неправильная трансляция заявит о себе в виде исключения подобного следующему:

Unhandled Exception: System.NotSupportedException: Method 'TrimEnd' has no supported translation to SQL.

Необработанное исключение: System.NotSupportedException: Метод 'TrimEnd' не имеет поддерживаемой трансляции в SQL.

...

Это совершенно ясное сообщение об ошибке. Давайте рассмотрим код из листинга 14.24, который генерирует это исключение.

Листинг 14.24. Запрос LINQ to SQL, который не может транслироваться

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                               where c.CustomerID.TrimEnd('K') == "LAZY"
                               select c;
foreach (Customer c in custs)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Обратите внимание, что метод TrimEnd, генерирующий исключение трансляции, вызывается на поле базы данных, а не на моем локальном строковом литерале. В листинге 14.25 я перенесу вызов метода TrimEnd в другую сторону равенства и посмотрим, что это даст.

Листинг 14.25. Запрос LINQ to SQL, который может транслироваться

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                               where c.CustomerID == "LAZY".TrimEnd('K')
                               select c;
foreach (Customer c in custs)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Вывод листинга 14.25 выглядит так:

Как видите, здесь нет никакого вывода. Но так и должно быть, поскольку это правильный вывод запроса, и никакого исключения трансляции SQL здесь нет.

Таким образом, вызов неподдерживаемого метода на столбце базы данных приводит к исключению, в то время как при вызове того же метода на переданном параметре — все в порядке. Это имеет смысл. LINQ to SQL не имеет никаких проблем с вызовом метода TrimEnd на нашем параметре, потому что может сделать это до привязки параметра к запросу, что и происходит в среде нашего процесса. Вызов метода TrimEnd на столбце базы данных должен был бы выполняться в базе данных, а это значит, что вместо вызова метода в среде нашего процесса этот вызов должен был быть транслирован в оператор SQL, передаваемый базе данных и выполняемый там. А так как метод TrimEnd не поддерживается в трансляции SQL, генерируется исключение.

Следует иметь в виду одну вещь, если вам нужно вызывать неподдерживаемый метод на столбце базы данных, возможно, вместо этого вы можете вызвать метод, имеющий противоположный эффект, но вызвать его на параметре. Скажем, к примеру, что вы хотите вызвать метод ToUpper на столбце базы данных, а он не поддерживается; возможно, вы можете вместо этого вызвать метод ToLower на параметре. Однако в этом случае метод ToUpper все-таки поддерживается, так что пример не совсем хороший. Также вы

должны убедиться, что вызываемый вами метод дает прямо противоположный эффект. В данном случае столбец базы данных может содержать строку в смешанном регистре, так что вызов `ToLower` не будет иметь в точности противоположного эффекта. Если столбец вашей базы данных содержит значение "Smith", а ваш параметр — "SMITH", и вы проверяете эквивалентность, то вызов метода `ToUpper` на столбце базы данных будет работать и даст вам соответствие. Однако если бы метод `ToUpper` не поддерживался, попытка вывернуть эту логику наизнанку вызовом метода `ToLower` на параметре не дала бы соответствия.

Вы можете спросить: как узнать, что метод `TrimEnd` не поддерживается в трансляции SQL. Поскольку природа того, какие примитивные типы и методы поддерживаются, динамична и подвержена изменениям, попытка документировать их все не входит в мои намерения. Относительно трансляции существует масса ограничений и исключений. Я надеюсь, что совершенствование трансляции SQL является областью приложения усилий со стороны Microsoft. Чтобы узнать, что поддерживается, а что нет, загляните в раздел документации MSDN, озаглавленный ".NET Framework Function Translation" для LINQ to SQL. Однако, как видно из предыдущих примеров, очень легко заметить, если метод не поддерживается.

Резюме

Я знаю, что эта глава выглядит как молниеносный тур по стандартным операциям для баз данных с использованием LINQ to SQL. Надеюсь, мне удалось сохранить примеры достаточно простыми, чтобы позволить вам сосредоточиться на основных шагах, необходимых для выполнения вставок, обновлений и удалений в базе данных. Также я указал на отличия в работе запросов LINQ to SQL и LINQ to Objects.

Имейте в виду, что любой код LINQ to SQL, который изменяет базу данных, должен обнаруживать и разрешать конфликты параллельного доступа. Однако для ясности ни в одном из примеров настоящей главы этого не делалось. Тема конфликтов параллельного доступа будет посвящена целиком глава 17, так что не беспокойтесь об этом.

В дополнение к пониманию того, как выполняются основные операции на сущностных объектах, важно также понимать, как это затрагивает сущностные объекты, ассоциированные с данным объектом. Помните, что когда вы вставляете сущностный объект в базу данных, все прикрепленные к нему объекты будут вставлены автоматически. Однако это не касается удалений. Чтобы удалить родительский сущностный объект в отношении ассоциации, вы должны сначала удалить дочерние сущностные объекты, чтобы предотвратить генерацию исключения.

Далее я продемонстрировал, как можно переопределять методы по умолчанию, которые сгенерированы для модификации записей базы данных, соответствующих вашему сущностному объекту. Такая возможность позволяет разработчику контролировать проведение изменений в базе данных, например, выполняя их посредством хранимых процедур.

И, наконец, я раскрыл вам тот факт, что запросы LINQ to SQL должны транслироваться в операторы SQL. Важно никогда не забывать об этой трансляции, и о том, что она накладывает некоторые ограничения на то, что вы можете сделать.

До настоящего момента в этой книге я уже не раз упоминал сущностные классы, не слишком углубляясь в них. Теперь пришло время обратить на них более пристальное внимание. Поэтому в следующей главе 15 я планирую рассмотреть эту тему полностью.

ГЛАВА 15

Сущностные классы LINQ to SQL

В предыдущих главах, посвященных LINQ to SQL, я много раз упоминал сущностные классы, но все еще не привел их четкого определения и описания. Здесь я исправлю это упущение.

В настоящей главе я дам определение сущностных классов, а также опишу различные способы их создания. Также я информирую вас о сложностях, с которыми вам придется иметь дело, если вы решите создавать свои собственные сущностные классы.

Но, прежде чем мы приступим к самому интересному, нужно сказать о некоторых предварительных условиях, которые следует выполнить, чтобы запускать примеры этой главы.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры настоящей главы, необходимо получить расширенную версию базы данных Northwind и сущностные классы, сгенерированные для нее. Прочтите и выполните инструкции, приведенные в разделе “Предварительные условия для запуска примеров” главы 12.

Сущностные классы

Классы, которые отображаются на базу данных SQL Server с использованием LINQ to SQL, известны как *сущностные классы* (entity classes). Созданный экземпляр сущностного класса — это сущность данного типа, и я буду называть ее *сущностным объектом* (entity object). Сущностные классы — это обычные классы C# с дополнительно специфицированными атрибутами LINQ to SQL. Вместо добавления атрибутов сущностные классы могут быть созданы с применением XML-файла отображения при создании экземпляра объекта `DataContext`. Эти атрибуты или вхождения файла отображения диктуют то, как сущностные классы могут отображаться на базу данных SQL Server с использованием LINQ to SQL.

Применяя сущностные классы, мы можем опрашивать и обновлять базу данных с использованием LINQ to SQL.

Создание сущностных классов

Сущностные классы — это базовые строительные блоки, участвующие в выполнении запросов LINQ to SQL. Для того чтобы начать использовать LINQ to SQL, необходимы сущностные классы. Есть два способа их получения: вы можете сгенерировать их, как

было показано в главах 12 и 13, или же вы можете написать их вручную. К тому же нет причин, запрещающих комбинировать оба способа.

Если у вас еще нет бизнес-классов для сущностей, хранящихся в базе данных, генерация сущностных классов — наверное, лучший подход. Если вы уже имеете объектную модель, тогда написание сущностных классов вручную более предпочтительно.

Если вы начинаете проект с нуля, я бы порекомендовал рассмотреть сначала моделирование базы данных и сгенерировать сущностные классы из этой базы данных. Это позволит вам применять сгенерированные сущностные классы, что облегчит задачу правильного их написания, что не всегда тривиально.

Генерация сущностных классов

До сих пор в этой книге я демонстрировал только генерацию сущностных классов, представляющую собой простейший способ их получения. В главе 12 я продемонстрировал, как сгенерировать сущностные классы из базы данных Northwind, чтобы вы могли пробовать запускать примеры LINQ to SQL, приведенные в этой книге. В главе 13 я описал детально, как можно генерировать сущностные классы с использованием либо инструмента командной строки по имени SQLMetal, либо инструмента с графическим интерфейсом пользователя под названием Object Relational Designer.

SQLMetal очень прост в использовании, но не предусматривает никаких опций для управления именованием сгенерированных сущностных классов помимо производства промежуточного XML-файла, его редактирования и последующей генерации сущностных классов на его основе. К тому же он генерирует сущностные классы для всех таблиц указанной базы данных, для всех полей этих таблиц, избавляя от трудоемкой процедуры, описанной ранее. Это дает вам минимум контроля над именами ваших сущностных классов и их свойств. Object Relational Designer потребует больше времени на создания полной объектной модели базы данных, но позволит вам точно специфицировать таблицы и поля, для которых вы хотите сгенерировать сущностные классы, а также позволит указать имена этих классов и их свойств. Мы уже обсуждали SQLMetal и Object Relational Designer в главе 13, так что обращайтесь в эту главу за дополнительными сведениями по использованию обоих инструментов.

Если вы генерируете сущностные классы для базы данных, никто не заставляет вас взаимодействовать с каждым сущностным классом каждой таблицы этой базы. Вы можете использовать сущностный класс для одной таблицы, и не использовать для другой. Вдобавок ничто не запрещает вам добавлять бизнес-функциональность к сгенерированным сущностным классам. Например, класс Customer был сгенерирован SQLMetal в главе 12. Нет причин, по которым было бы нельзя добавлять к классу Customer собственные бизнес-методы или непостоянные члены класса. Однако если вы делаете это, убедитесь, что ваши модификации не касаются сгенерированной части кода сущностного класса. Вместо этого создайте другой модуль класса Customer и воспользуйтесь преимуществом того факта, что сущностные классы генерируются как частичные классы. Частичные классы — замечательное приобретение C#, которое, как никогда ранее, облегчает разделение функциональности по разным модулям. Таким образом, если вам по какой-то причине придется повторно сгенерировать сущностный класс, вы не потеряете добавленных вами членов или методов.

Написание сущностных классов вручную

Написание сущностных классов вручную — более трудный подход. Он требует глубокого понимания атрибутов LINQ to SQL или внешней схемы отображения. Однако в то же время написание сущностных классов вручную — замечательный способ действительно изучить LINQ to SQL.

Где ручное написание сущностных классов действительно оправдано — так это в случае, когда у вас уже есть какие-то бизнес-классы. У вас может быть существующее приложение с уже реализованной собственной объектной моделью. В этом случае было бы не слишком выгодно генерировать сущностные классы из базы данных, поскольку у вас уже есть объектная модель, используемая приложением.

Решение состоит в добавлении необходимых атрибутов в вашу объектную модель или создание файла отображения. Благодаря гибкости LINQ to SQL нет необходимости, чтобы имена ваших классов соответствовали именам таблиц, в которых они сохраняются, а имена свойств — соответствовали именам столбцов этих таблиц. Это значит, что ранее реализованные классы могут быть модифицированы для хранения в базе данных SQL Server.

Чтобы создавать сущностные классы вручную с использованием атрибутов, вам нужно добавить соответствующие атрибуты к вашим классам — будь то существующие бизнес-классы либо новые классы — которые должны стать сущностными. В разделе настоящей главы “Атрибуты сущностных классов и свойства атрибутов” вы найдете описание доступных атрибутов и свойств.

Чтобы создавать сущностные классы с использованием внешнего файла отображения, вам понадобится создать XML-файл, который отвечает схеме, описанной ниже в разделе “XML-схема внешнего файла отображения” настоящей главы. Имея такой внешний файл отображения, вы используете соответствующий конструктор `DataContext` для создания объекта `DataContext`, который загрузит файл отображения. У этого класса есть два конструктора, позволяющие специфицировать внешний файл отображения.

Дополнительные обязанности сущностных классов

К сожалению, при написании сущностных классов вручную недостаточно только понимать атрибуты и свойства этих атрибутов. Вы также должны знать о некоторых дополнительных обязанностях сущностных классов.

Например, при написании сущностных классов вручную следует помнить об уведомлениях, связанных с изменением, и способе их реализации. Вы должны также обеспечивать целостность графа между вашими родительскими и дочерними классами.

Эти дополнительные обязанности учитываются при генерации классов инструментальными средствами `SQLMetal` и `Object Relational Design`, но если вы создаете собственный сущностный класс, то должны самостоятельно добавить весь необходимый код.

Уведомления об изменениях

В главе 16 мы поговорим об отслеживании изменений. Выясняется, что отслеживание изменений не может быть реализовано элегантно и эффективно без участия самих сущностных классов. Если ваши сущностные классы сгенерированы `SQLMetal` или `Object Relational Design`, то вы можете расслабиться, поскольку эти инструменты позаботятся обо всем, реализуя код, участвующий в уведомлениях об изменениях, при генерации ваших сущностных классов. Но если вы пишете собственные сущностные классы, вам следует понимать механизм уведомления об изменениях и самостоятельно реализовывать код, участвующий в этом процессе.

Сущностные классы имеют выбор — участвовать в уведомлениях об изменениях или нет. Если они не участвуют, то `DataContext` обеспечивает это, сохраняя две копии каждого сущностного объекта; одну для исходных значений и другую — для текущих. Он создает копии при первоначальном извлечении сущности из базы данных в начале отслеживания изменений. Однако вы можете сделать этот механизм более эффективным, если реализуете вашим сущностным классом два интерфейса уведомления об изменениях — `System.ComponentModel.INotifyPropertyChanging` и `System.ComponentModel.INotifyPropertyChanged`. На протяжении глав, посвященных LINQ

to SQL, я буду часто ссылаться на код, сгенерированный SQLMetal, дабы продемонстрировать вам квинтэссенцию способа обработки уведомления об изменении. Чтобы реализовать интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`, я должен сделать четыре вещи.

Сначала я должен указать, что мой сущностный класс реализует интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{ ... }
```

Поскольку сущностный класс реализует эти два интерфейса, `DataContext` будет известно, что нужно зарегистрировать два обработчика для двух событий, о которых я расскажу в нескольких абзацах.

Вы можете также заметить, что в приведенном коде специфицирован атрибут `Table`. В этом разделе я буду показывать связанные с контекстом атрибуты, но пока не стану объяснять их, поскольку это будет сделано в следующей главе. А пока просто представьте, что их нет.

Во-вторых, мне нужно объявлять `private static` класс типа `PropertyChangingEventArgs` и передать его конструктору `String.Empty`.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    private static PropertyChangingEventArgs emptyChangingEventArgs =
        new PropertyChangingEventArgs(String.Empty);
    ...
}
```

Объект `emptyChangingEventArgs` будет передан одному из ранее упомянутых обработчиков событий, когда возникнет соответствующее событие.

В-третьих, мне нужно добавить к сущностному классу два члена `public event` — один типа `System.ComponentModel.PropertyChangingEventHandler` по имени `PropertyChanging`, и еще один — типа `System.ComponentModel.PropertyChangedEventHandler` по имени `PropertyChanged`.

Из сгенерированного сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    private static PropertyChangingEventArgs emptyChangingEventArgs =
        new PropertyChangingEventArgs(String.Empty);
    ...
    ...
    public event PropertyChangingEventHandler PropertyChanging;
    public event PropertyChangedEventHandler PropertyChanged;
}
```

Когда объект `DataContext` инициирует отслеживание изменений сущностного объекта, я должен возбудить событие `PropertyChanging` непосредственно перед изменением свойства, а событие `PropertyChanged` — сразу после изменения свойства.

Хотя реализовывать возбуждение событий подобным образом и не обязательно, но для согласованности SQLMetal генерирует для вас методы `SendPropertyChanging` и `SendPropertyChanged`.

Из сгенерированного сущностного класса Customer

```

protected virtual void SendPropertyChanging()
{
    if ((this.PropertyChanging != null))
    {
        this.PropertyChanging(this, emptyChangingEventArgs);
    }
}
protected virtual void SendPropertyChanged(String propertyName)
{
    if ((this.PropertyChanged != null))
    {
        this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

Обратите внимание, что при возбуждении события `PropertyChanged` создается новый объект `PropertyChangedEventArgs`, получающий имя конкретного свойства, которое подвергается изменению. Это позволяет объекту `DataContext` знать точно, какое свойство изменилось. Поэтому при вызове метода `SendPropertyChanging` он инициирует событие `PropertyChanging`, которое приводит к вызову зарегистрированного объектом `DataContext` обработчика. То же самое происходит с методом `SendPropertyChanged` и событием `PropertyChanged`.

Конечно, вы можете встроить похожую логику в ваш код вместо создания повторно используемых методов, но это более утомительно, к тому же увеличивает объем сопровождаемого кода.

Затем в методе `set` каждого свойства я должен вызывать два метода `SendPropertyChanging` и `SendPropertyChanged` непосредственно перед изменением свойства и после него.

Из сгенерированного сущностного класса Customer

```

[Column(Storage="_ContactName", DbType="NVarChar(30)")]
public string ContactName
{
    get
    {
        return this._ContactName;
    }
    set
    {
        if ((this._ContactName != value))
        {
            this.OnContactNameChanging(value);
            this.SendPropertyChanging();
            this._ContactName = value;
            this.SendPropertyChanged("ContactName");
            this.OnContactNameChanged();
        }
    }
}

```

Опять-таки, обратите внимание, что при вызове метода `SendPropertyChanged` передается имя свойства, в данном случае — `ContactName`. Как только вызывается метод `SendPropertyChanged`, объект `DataContext` узнает, что свойство сущностного объекта `ContactName` было изменено.

Также я должен отслеживать возникновение соответствующих событий в методах set свойств, представляющих ассоциации. Поэтому на стороне “многие” ассоциации “один ко многим” мне нужно показанный ниже следующий код, выделенный полужирным.

Из класса Order, поскольку Customer не имеет свойств EntityRef<T>

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
    ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer
{
    get
    {
        return this._Customer.Entity;
    }
    set
    {
        Customer previousValue = this._Customer.Entity;
        if ((previousValue != value)
            || (this._Customer.HasLoadedOrAssignedValue == false))
        {
            this.SendPropertyChanging();
            if ((previousValue != null))
            {
                this._Customer.Entity = null;
                previousValue.Orders.Remove(this);
            }
            this._Customer.Entity = value;
            if ((value != null))
            {
                value.Orders.Add(this);
                this._CustomerID = value.CustomerID;
            }
            else
            {
                this._CustomerID = default(string);
            }
            this.SendPropertyChanged("Customer");
        }
    }
}
```

А на стороне “один” ассоциации “один ко многим” мне понадобится следующий код, выделенный полужирным:

Из сгенерированного сущностного класса Customer

```
public Customer()
{
    ...
    this._Orders =
        new EntitySet<Order>(new Action<Order>(this.attach_Orders),
                               new Action<Order>(this.detach_Orders));
}
...
private void attach_Orders(Order entity)
{
    this.SendPropertyChanging();
    entity.Customer = this;
    this.SendPropertyChanged("Orders");
}
```

```

private void detach_Orders(Order entity)
{
    this.SendPropertyChanging();
    entity.Customer = null;
    this.SendPropertyChanged("Orders");
}

```

Если вы не знакомы с обобщенным делегатом Action, использованным в приведенном коде, скажу, что он находится в пространстве имен System, и был добавлен в .NET Framework в версии 3.0. Предыдущий код создает экземпляр объекта Action для сущностного класса Order и передает ему делегат метода attach_Orders. LINQ to SQL использует этот делегат позднее, чтобы присвоить объекту Order соответствующий Customer. Аналогично создается экземпляр другого объекта-делегата Action, которому передается делегат метода detach_Orders. Этот делегат LINQ to SQL использует позднее для удаления присвоенного Customer из Order.

Реализуя подобным образом уведомление об изменении, мы можем сделать слежение за изменениями более эффективным. Теперь объект DataContext знает, когда и какие свойства сущностного класса изменяются.

Когда мы вызываем метод SubmitChanges, объект DataContext просто забывает исходные значения своих свойств, и этими исходными значениями свойств становятся текущие их значения, после чего начинается отслеживание изменений. Метод SubmitChanges детально описан в главе 16.

Конечно, как я ранее упоминал, если вы позволяете создавать ваши сущностные классы SQLMetal или Object Relational Designer, то вы избавлены от всех этих сложностей, потому что они генерируют весь необходимый рутинный код. И только если вы пишете сущностные классы вручную, вам следует сосредоточиться на реализации уведомлений об изменении.

Согласованность графа

В математике, когда узлы соединяются вместе, полученная в результате сеть называется *графом*. В некотором отношении сеть, представляющая соединения, которые образованы классами, ссылающимися на другие классы, также считается графом. Когда у вас есть два сущностных класса, участвующие в отношении — в том смысле, что между ними создается Association, — то поскольку каждый имеет ссылку на другого, в результате получаем граф.

Когда вы модифицируете отношение между двумя сущностными объектами, такими как Customer и Order, ссылка на каждую сторону отношения должна корректно обновляться, чтобы каждый сущностный объект правильно ссылался или более не ссылался на другой. Это верно как для создания отношений, так и для их удаления. Хотя LINQ to SQL требует, чтобы программист, пишущий код, который использует сущностные классы, модифицировал только одну сторону отношения, иногда приходится обрабатывать обновление другой стороны, и LINQ to SQL не делает этого для нас.

За обработку обновления другой стороны отношения отвечает сущностный класс. Если вы позволяете SQLMetal или Object Relational Designer генерировать ваши сущностные классы, вы свободны, поскольку они делают это для вас. Но когда вы создаете свои собственные сущностные классы, вам придется реализовать весь необходимый код самостоятельно.

При гарантированном правильном обновлении каждой стороны отношения граф остается согласованным. Без этого целостность графа может нарушиться и наступит хаос. Customer может быть связан с Order, а Order окажется связан с другим Customer, или вообще не связан ни с каким. Это сделает невозможной навигацию и является совершенно недопустимым.

К счастью, Microsoft предлагает шаблон, который мы можем использовать для того, чтобы гарантировать, что сущностные классы корректно реализуют согласованность графов. Взглянем на соответствующую реализацию, сгенерированную SQLMetal на базе данных Northwind.

Из генерированного сущностного класса Customer

```
public Customer()
{
    ...
    this._Orders =
        new EntitySet<Order>(new Action<Order>(this.attach_Orders),
        new Action<Order>(this.detach_Orders));
}
...
private void attach_Orders(Order entity)
{
    this.SendPropertyChanging();
    entity.Customer = this;
    this.SendPropertyChanged("Orders");
}
private void detach_Orders(Order entity)
{
    this.SendPropertyChanging();
    entity.Customer = null;
    this.SendPropertyChanged("Orders");
}
```

В данном примере класс Customer будет родительским, или стороной “один” в отношении “один ко многим”. Класс Order будет дочерним, представляющим сторону “многие” того же отношения.

В приведенном коде можно увидеть, что в конструкторе родительского класса Customer, когда инициализируется член EntitySet<T> для нашей коллекции дочерних объектов _Orders, конструктору передаются два объекта-делегата Action<T>.

Первый объект Action<T> получает делегат метода обратного вызова, который обработает присваивание текущего объекта Customer, на который ссылается ключевое слово this, такого как Customer объекта Order, который будет передан в метод обратного вызова. В приведенном выше коде метод обратного вызова, на который я ссылаюсь — это метод attach_Orders.

Второй параметр конструктора EntitySet<T> — объект-делегат Action<T>, который несет в себе делегат метода обратного вызова, который обработает удаление присваивания Customer переданного объекта Order. В приведенном выше коде метод обратного вызова, на который я ссылаюсь — это метод detach_Orders.

Даже несмотря на то, что приведенный выше код находится в родительском классе Customer, присваивание объекта дочернего класса Order на самом деле обрабатывается свойством Customer объекта Order. Вы можете видеть это как в методе attach_Orders, так и в detach_Orders; все, что они в действительности делают — это изменение свойства Customer объекта Order. Вы можете видеть, что свойство entity.Customer устанавливается в this и null, соответственно, чтобы прикрепить текущий Customer и отсоединить прикрепленный в данный момент Customer. В методах set и get дочернего класса Order происходит вся работа по поддержке согласованности графа. Таким образом, мы делегируем реальную работу дочернему классу. В родительском классе больше ничего делать не нужно.

Однако обратите внимание, что перед тем, как выполнить работу в методах `attach_Orders` и `detach_Orders`, инициируются уведомления об изменении вызовом методов `SendPropertyChanging` и `SendPropertyChanged`.

Теперь посмотрим, что нужно сделать в дочернем классе отношения "родительский-дочерний" для поддержки согласованности графа.

Из сгенерированного сущностного класса Order

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
    ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer
{
    get
    {
        return this._Customer.Entity;
    }
    set
    {
        Customer previousValue = this._Customer.Entity;
        if (((previousValue != value)
            || (this._Customer.HasLoadedOrAssignedValue == false)))
        {
            this.SendPropertyChanging();
            if ((previousValue != null))
            {
                this._Customer.Entity = null;
                previousValue.Orders.Remove(this);
            }
            this._Customer.Entity = value;
            if ((value != null))
            {
                value.Orders.Add(this);
                this._CustomerID = value.CustomerID;
            }
            else
            {
                this._CustomerID = default(string);
            }
            this.SendPropertyChanged("Customer");
        }
    }
}
```

В приведенном коде мы рассматриваем только метод `set` свойства `Customer` — именно потому, что родительская сторона отношения возлагает на него ответственность на поддержание согласованности графа.

Поскольку этот метод довольно сложный, я представлю код вместе с его описанием.

```
set
{
    Customer previousValue = this._Customer.Entity;
```

Вы можете видеть в первой строке кода метода `set` сохранение оригинального `Customer`, присвоенного `Order`, в переменную `previousValue`. Пусть вас не смущает тот факт, что код ссылается на `this._Customer.Entity`. Напомню, что переменная-член `_Customer` — это на самом деле объект типа `EntityRef<Customer>`, а не `Customer`. Чтобы получить объект `Customer`, код должен обратиться к свойству `Entity` объекта `EntityRef<T>`. Поскольку здесь `EntityRef<T>` предназначен для хранения `Customer`,

типов Entity будет Customer; никакого приведения не требуется. Я обожаю обобщения, появившиеся в C#, а вы?

```
if (((previousValue != value)
    || (this._Customer.HasLoadedOrAssignedValue == false)))
{
```

Далее приведенная строка кода проверяет, не присвоен ли уже этот Customer объекту Order, сравнивая переданный параметр value с имеющейся в Order ссылкой на Customer, поскольку в этом случае ничего делать не надо, если только этот Customer не был еще загружен или присвоен. Это не только логически оправдано, но, учитывая рекурсивную природу работы этого кода, эта строка становится чрезвычайно важной, поскольку позволяет остановить рекурсию.

```
this.SendPropertyChanging();
```

В этой строке кода вызывается метод SendPropertyChanging для инициации события уведомления об изменении.

```
if ((previousValue != null))
{
```

Далее код определяет, не присвоен ли уже родительский объект Customer данному дочернему объекту Order, сравнивая previousValue с null.

Если Customer назначен Order, в том смысле, что переменная previousValue, представляющая присвоенный Customer, не равна null, то код должен установить свойство Entity объекта Customer EntityRef<T> в null в следующей строке.

```
this._Customer.Entity = null;
```

Свойство Entity устанавливается здесь в null, чтобы остановить рекурсию, которая запускается в следующей строке кода. Поскольку свойство Entity свойства Customer объекта Order теперь равно null и не ссылается на действительный объект Customer, в то время как свойство Orders объекта Customer все еще содержит этот Order в коллекции, в данный момент времени граф находится в несогласованном состоянии.

В следующей строке кода вызывается метод Remove на свойстве Orders объекта Customer, и текущий Order передается в качестве удаляемого Order.

```
previousValue.Orders.Remove(this);
}
```

Вызов метода Remove инициирует вызов метода detach_Orders класса Customer и передает ему подлежащий удалению Order. В методе detach_Orders свойству Customer переданного объекта Order присваивается null. Чтобы освежить вашу память, еще раз покажу, как выглядит метод detach_Orders.

Это код отдельного метода, приведенный здесь для вашего удобства

```
private void detach_Orders(Order entity)
{
    this.SendPropertyChanging();
    entity.Customer = null;
    this.SendPropertyChanged("Orders");
}
```

Когда вызывается метод detach_Orders, свойство Customer переданного Order получает значение null. Это приводит к вызову метода set свойства Customer объекта Order, а этот метод вызывает код, обращающемся, в свою очередь, к методу detach_Orders, так что метод, запустивший этот процесс удаления Order, вызывается рекурсивно, и значение null передается в качестве value в метод set. Поток выполнения "переходит", находясь в рекурсивном вызове метода set свойства Customer.

Метод detach_Orders инициирует рекурсивный вызов метода set

```

set
{
    Customer previousValue = this._Customer.Entity;
    if (((previousValue != value)
        || (this._Customer.HasLoadedOrAssignedValue == false)))
    {

```

В четвертой строке метода set проверяется переданное значение value, и если оно эквивалентно свойству Entity текущего назначенного свойства Customer, этот рекурсивный вызов метода set возвращает управление, ничего не делая. Поскольку в предыдущей строке кода первого не рекурсивного вызова метода set свойству Entity свойства Customer присваивается значение null, и поскольку null был передан как value в метод detach_Orders, они на самом деле эквивалентны: существует рекурсивный вызов метода set, который ничего не делает, и тогда поток управления возвращается обратно — к первому вызову метода set. Именно это я имел в виду в предыдущем абзаце, говоря, что свойство Entity устанавливается в null, чтобы остановить рекурсию. Теперь сделаем небольшой перерыв, пока я приму немногого аспирина.

Итак, как только рекурсивный вызов метода set возвратил управление, поток возвращается к последней строке начального вызова метода set, о котором я говорил.

Строка кода, повторяющаяся из предыдущего фрагмента, для вашего удобства

```

    previousValue.Orders.Remove(this);
}

```

Как только метод Orders.Remove завершает работу, свойство Orders объекта Customer более не содержит ссылки на этот Order, и потому наш график вновь находится в согласованном состоянии.

Очевидно, что если вы планируете писать ваши собственные сущностные классы, вам стоит потратить некоторое время на их отладку. Просто поставьте точки останова в методах detach_Orders и set и наблюдайте, что происходит.

Затем в свойство Entity объекта Customer объекта Order устанавливается новый объект Customer, который был передан методу set в параметре value.

```
this._Customer.Entity = value;
```

В конце концов, это метод set свойства Customer. Мы пытались назначить Order новому Customer. И, опять-таки, в этой точке Order содержит ссылку на вновь назначенный Customer, а вновь назначенный Customer не имеет ссылки на Order, так что график отныне не согласован.

Далее код проверяет, не равен ли null назначаемый Order Customer, потому что если нет, то вновь назначенный Customer следует назначить Order.

```

if ((value != null))
{

```

Если переданный в value объект Customer не равен null, нужно добавить текущий Order в коллекцию объектов Orders переданного Customer.

```
value.Orders.Add(this);
```

Когда в этой строке Order добавляется в коллекцию Orders переданного объекта Customer, то вызывается делегат, переданный методу обратного вызова в конструкторе EntitySet<T> объекта Customer. Поэтому результатом присваивания является вызов метода attach_Orders объекта Customer.

Это, в свою очередь, установит свойство Customer текущего объекта Order в переданный Customer, в результате чего опять вызывается метод set свойства Customer

объекта Order. Код рекурсивно входит в метод set, как делал это ранее. Однако все-таки двумя операторами кода перед предыдущим оператором и перед началом рекурсии свойство Entity свойства Customer объекта Order было установлено в новый Customer, который передан методу set методом attach_Orders. Опять-таки, код метода set вызывается рекурсивно, и в конечном итоге вызывается следующая строка кода:

Следующая строка кода — из другого вызова метода set

```
if ((previousValue != value)
    || (this._Customer.HasLoadedOrAssignedValue == false)))
```

Поскольку текущий объект Customer объекта Order, который теперь хранится в previousValue, и параметр value — одно и то же, метод set возвращает управление, ничего более не делая, в результате чего рекурсия прекращается.

В следующей строке кода член CustomerID текущего объекта Order устанавливается в CustomerID нового объекта Customer.

```
this._CustomerID = value.CustomerID;
}
```

Если вновь назначаемый Customer равен null, то этот код просто устанавливает член CustomerID текущего объекта Order в значение по умолчанию для его типа данных — в этом случае типа string.

```
else
{
    this._CustomerID = default(string);
}
```

Если бы член CustomerID был типа int, код установил бы его в default(int).

В самой последней строке кода вызывается метод SendPropertyChanged и ему передается имя изменяемого свойства, чтобы инициировать событие уведомления об изменении.

```
this.SendPropertyChanged("Customer");
}
```

Этот шаблон действителен для отношения “один ко многим”. Для отношения “один к одному” каждая сторона должны быть реализована, как была реализована дочерняя в этом примере, но с парой изменений. Поскольку в отношении “один к одному” нет логического родителя или ребенка, представим, что отношение между заказчиками и заказами строится по схеме “один к одному”. Это даст мне имя для использования в качестве ссылки на каждый класс, поскольку родитель и ребенок более не применимы.

Если вы пишете сущностный класс вручную, и отношение между классом Customer и классом Order — “один к одному”, тогда каждый из этих классов будет содержать свойство типа EntityRef<T>, где T — тип противоположного класса. Класс Customer будет содержать EntityRef<Order>, а класс Order — EntityRef<Customer>. Поскольку ни один из классов не содержит EntitySet<T>, нет вызовов методов Add и Remove, которые присутствуют в ситуации с отношениями “один ко многим”, описанной ранее.

Таким образом, если предположить, что между заказами и заказчиками существует отношение “один к одному”, то метод set свойства Customer объекта Order должен выглядеть в основном так же, как и ранее, за исключением случая, когда мы удаляем назначение текущего Order исходному Customer. Поскольку оригинальный Customer имеет единственный Order, мы не удаляем текущий Order из коллекции объектов Order; мы просто присвоим null свойству Customer объекта Order.

Таким образом, вместо такой строки кода:

```
previousValue.Orders.Remove(this);
```

у нас будет следующая строка кода:

```
previousValue.Order = null;
```

Аналогично при назначении текущего Order новому Customer, поскольку он имеет единственный Order, вместо вызова метода Add на коллекции объектов Order мы просто устанавливаем свойство Customer нового объекта Order в текущий Order.

Поэтому вместо следующей строки кода:

```
value.Orders.Add(this);
```

у нас будет такая строка:

```
value.Order = this;
```

Как видите, обработка согласованности графа нетривиальна, и в ней легко запутаться. К счастью, существуют два инструмента, которые обо всем позаботятся за вас. Их названия — SQLMetal и Object Relational Designer. В том, что касается поддержки согласованности графа и правильной реализации уведомлений об изменении они стоят своего веса в металле. Возможно, инструмент командной строки должен был бы называться SQLGold, но я думаю, что "металлическая" часть его названия происходит от метаязыка (metalanguage).

Вызов соответствующих частичных методов

Когда Microsoft добавила частичные методы для облегчения расширения генерированного кода, такого как код сущностных классов, она тем самым возложила на вас немного больше ответственности, если вы собираетесь самостоятельно реализовывать сущностные классы.

Существуют несколько частичных методов, которые вы должны объявлять в ваших вручную написанных сущностных классах:

```
partial void OnLoaded();
partial void OnValidate(ChangeAction action);
partial void OnCreated();
partial void On[Property]Changing(int value);
partial void On[Property]Changed();
```

У вас должна быть пара методов On[Property]Changing и On[Property]Changed для каждого свойства сущностного класса.

Что касается методов OnLoaded и OnValidate, вам не нужно добавлять их вызовы где-либо в коде сущностных классов; вместо вас их будет вызывать DataContext.

Вы должны добавить код вызова метода OnCreate внутри конструктора вашего сущностного класса, как показано ниже.

Вызов частичного метода OnCreated

```
public Customer()
{
    OnCreated();
    ...
}
```

Затем для каждого свойства сущностного класса вы должны добавить вызов методов On[Property]Changing и On[Property]Changed непосредственно перед и после изменения этого свойства, как показано ниже.

Вызов из метода set свойства сущностного класса методов On[Property]Changing и On[Property]Changed

```

public string CompanyName
{
    get
    {
        return this._CompanyName;
    }
    set
    {
        if ((this._CompanyName != value))
        {
            this.OnCompanyNameChanging(value);
            this.SendPropertyChanging();
            this._CompanyName = value;
            this.SendPropertyChanged("CompanyName");
            this.OnCompanyNameChanged();
        }
    }
}

```

Обратите внимание, что метод On[Property]Changing вызывается перед вызовом метода SendPropertyChanging, а метод On[Property]Changed — после метода SendPropertyChanged.

Объявляя и вызывая эти частичные методы, вы предоставляете другим разработчикам возможность легкого расширения без потери производительности, если они такой возможностью не воспользуются. В этом прелесть частичных методов.

Сложности, связанные с EntityRef<T>

В то время как типом данных для ассоциированного класса является EntityRef<T>, общедоступное свойство для этого приватного члена класса должно возвращать тип сущностного класса, а не EntityRef<T>.

Взглянем на то, как SQLMetal генерирует свойство для члена EntityRef<T>.

Общедоступное свойство для члена класса, возвращающее

тип сущностного класса вместо EntityRef<T>

```

[Table(Name="dbo.Orders")]
public partial class Order : INotifyPropertyChanging, INotifyPropertyChanged
{
    ...
    private EntityRef<Customer> _Customer;
    ...
    [Association(Name="FK_Orders_Customers", Storage="_Customer",
        ThisKey="CustomerID", IsForeignKey=true)]
    public Customer Customer
    {
        get
        {
            return this._Customer.Entity;
        }
        set
        {
            ...
        }
    }
}

```

Как видите, даже несмотря на то, что приватный член класса имеет тип EntityRef<Customer>, свойство Customer возвращает тип Customer, а не EntityRef<Customer>. Это важно, потому что любая ссылка в запросе на тип EntityRef<T> не будет транслироваться в SQL.

Сложности, связанные с EntitySet<T>

В то время как общедоступные свойства для приватных членов класса типа EntityRef<T> должны возвращать тип T вместо EntityRef<T>, того же нельзя сказать об общедоступных свойствах для приватных членов класса типа EntitySet<T>. Взглянем на код, сгенерированный SQLMetal для члена приватного класса типа EntitySet<T>.

Приватный член класса EntitySet<T> и его свойство

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    ...
    private EntitySet<Order> _Orders;
    ...
    [Association(Name="FK_Orders_Customers", Storage="_Orders", OtherKey="CustomerID",
        DeleteRule="NO ACTION")]
    public EntitySet<Order> Orders
    {
        get
        {
            return this._Orders;
        }
        set
        {
            this._Orders.Assign(value);
        }
    }
    ...
}
```

Как видите, свойство возвращает тип EntitySet<Order> — тот же, который имеет и приватный член класса. Поскольку EntitySet<T> реализует интерфейс ICollection<T>, вам, возможно, стоило бы сделать так, чтобы свойство возвращало тип ICollection<T>, если вы хотите скрыть детали реализации.

Другая сложность, о которой нужно помнить при написании ваших собственных сущностных классов, заключается в том, что когда вы пишете общедоступный метод set для свойства EntitySet<T>, то должны использовать метод Assign вместо простого присваивания переданного значения члену класса EntitySet<T>. Это позволит сущностному объекту продолжать применять оригинальную коллекцию ассоциированных сущностных объектов, поскольку коллекция может уже отслеживаться службой слежения за изменениями объекта DataContext.

Если вы еще раз взглянете на предыдущий пример кода, то можете увидеть, что вместо присваивания переменной-члену this._Orders значения value он вызывает метод Assign.

Атрибуты сущностных классов и свойства атрибутов

Сущностные классы определяются атрибутами и свойствами атрибутов, которые отображают такой класс на таблицу базы данных, а его свойства — на столбцы таблицы базы данных. Атрибуты определяют наличие отображения, а свойства атрибутов специфицируют, как оно осуществляется. Например, есть атрибут Table, определяющий,

тот класс отображается на таблицу, но есть также его свойство Name, указывающее имя таблицы базы данных, на которую отображается класс.

Нет лучшего способа разобраться в атрибутах, их свойствах и в том, как они работают, чем изучить атрибуты, сгенерированные экспертами. Поэтому проанализируем сущностный объект Customer, сгенерированный SQLMetal.

Ниже приведена часть сущностного класса Customer.

Часть сгенерированного SQLMetal сущностного класса Customer

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
    ...
    [Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL", CanBeNull=false,
        IsPrimaryKey=true)]
    public string CustomerID
    {
        get
        {
            return this._CustomerID;
        }
        set
        {
            if ((this._CustomerID != value))
            {
                this.OnCustomerIDChanging(value);
                this.SendPropertyChanged();
                this._CustomerID = value;
                this.SendPropertyChanged("CustomerID");
                this.OnCustomerIDChanged();
            }
        }
    }
    ...
    [Association(Name="FK_Orders_Customers", Storage="_Orders", OtherKey="CustomerID",
        DeleteRule="NO ACTION")]
    public EntitySet<Order> Orders
    {
        get
        {
            return this._Orders;
        }
        set
        {
            this._Orders.Assign(value);
        }
    }
    ...
}
```

Для краткости я опустил все части сущностного класса за исключением тех, что содержат атрибуты LINQ to SQL. Также я опустил все излишние атрибуты.

А вот часть, содержащая хранимую процедуру и определенную пользователем функцию.

Часть, содержащая хранимую процедуру и определенную пользователем функцию

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
```

```

[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    ...
}

[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
    ...
}

```

В предыдущих фрагментах кода атрибуты выделены полужирным. Я привел эти фрагменты кода для обеспечения контекста для дискуссии об атрибутах.

Database

Атрибут `Database` специфицирует для класса-наследника `DataContext` имя по умолчанию отображенной базы данных, если это имя не специфицировано в строке подключения при создании экземпляра `DataContext`. Если атрибут `Database` не специфицирован и база данных не указана в строке подключения, то имя класса-наследника `DataContext` предполагается совпадающим с именем базы данных, к которой выполняется подключение.

Таким образом, для ясности: порядок приоритетов в определении имени базы данных, от высшего к низшему, выглядит следующим образом.

1. Информация подключения, представленная при создании экземпляра `DataContext`.
2. Имя базы данных, специфицированное в атрибуте `Database`.
3. Имя класса-наследника `DataContext`.

Ниже показана существенная часть сгенерированного `SQLMetal` класса-наследника `DataContext` по имени `Northwind`.

Из класса Northwind, сгенерированного SQLMetal

```
public partial class Northwind : System.Data.Linq.DataContext
{
```

Как видите, атрибут `Database` не специфицирован в сгенерированном классе `Northwind`, производном от класса `DataContext`. Поскольку класс был сгенерирован Microsoft, я могу предположить, что это сделано намеренно. Если вы собираетесь специфицировать атрибут `Database` и хотите, чтобы по умолчанию использовалась база данных по имени `NorthwindTest`, код должен выглядеть следующим образом:

Атрибут `Database`

```
[Database(Name="NorthwindTest")]
public partial class Northwind : System.Data.Linq.DataContext
{
```

Я не вижу причин избегать атрибута `Database`. Возможно, это потому, что если вы специфицируете базу данных в строке соединения, она переопределит имя класса `DataContext` и атрибут `Database`. Возможно, разработчики из Microsoft думали, что если вы не специфицируете имя базы данных в строке соединения, то будет использовано имя производного от `DataContext` класса, что удовлетворительно.

Я подумал об этом и пришел к заключению, что лично мне не нравится идея подключения сгенерированного класса-наследника `DataContext` к базе данных по умолчанию. Меня коробит от мысли, что придется запустить приложение, возможно, случайно, которое не было однозначно сконфигурировано, и допустить его подключение к базе данных по умолчанию. Это представляет собой возможность для очень болезненной ошибки, которая рано или поздно случится. Фактически я могу согласиться лишь с указанием атрибута `Database` с намеренно нелепым именем, чтобы предотвратить подключение к базе данных по умолчанию. Возможно, это должно выглядеть примерно так, как показано ниже.

Класс-наследник `DataContext`, допускающий в высшей степени маловероятное подключение к базе данных по умолчанию

```
[Database(Name="goopeygobezileywag")]
public partial class Northwind : System.Data.Linq.DataContext
{
```

И тогда я не увижу подключения ни к какой базе данных, если только не специфицирую ее в строке соединения, передаваемой `DataContext` во время создания его экземпляра.

Name (string)

Свойство `Name` атрибута `Database` — это строка, специфицирующая имя базы данных, с которой нужно устанавливать соединение, если имя базы не указано явно в информации подключения при создании экземпляра класса-наследника `DataContext`. Если свойство атрибута `Name` не специфицировано, а также не указано имя базы данных в информации подключения, тогда имя класса-наследника `DataContext` служит именем базы данных, к которой нужно подключиться.

Table

Атрибут `Table` специфицирует — в какой таблице базы данных будут сохраняться экземпляры сущностного класса. Имя сущностного класса не обязательно должно совпадать с именем таблицы. Вот часть сущностного класса, содержащая этот атрибут.

Атрибут `Table`

```
[Table(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging, INotifyPropertyChanged
{
```

Обратите внимание, что атрибут `Table` специфицирует имя таблицы базы данных в свойстве `Name`. Если имя сущностного класса совпадает с именем таблицы базы данных, свойство атрибута `Name` может быть опущено, так как имя класса станет по умолчанию именем таблицы, на которую он отображается.

В данном примере, поскольку я специфицировал опцию множественного числа при использовании `SQLMetal` для генерации моих сущностных классов `Northwind`, имя таблицы базы данных — `Customers` — преобразуется в форму единственного числа — `Customer`, которая и является именем класса. Поскольку имя класса не совпадает с именем таблицы базы данных, атрибут `Name` должен быть специфицирован.

Name (string)

Свойство `Name` атрибута `Table` — строка, специфицирующая наименование таблицы, на которую отображается данный сущностный класс. Если свойство атрибута `Name` не специфицировано, то имя сущностного класса по умолчанию будет отображено на однокомпонентную с ним таблицу базы данных.

Column

Атрибут `Column` указывает на то, что свойство сущностного класса отображается на поле базы данных. Вот иллюстрирующая его часть сущностного класса.

Атрибут `Column`

```
Private string _CustomerID;
...
[Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL", CanBeNull=false,
    IsPrimaryKey=true)]
public string CustomerID
{
```

В данном примере, поскольку специфицирован атрибут `Storage`, LINQ to SQL сможет непосредственно обращаться к переменной-члену `_CustomerID`, минуя общедоступное средство доступа `CustomerID`. Если свойство атрибута `Storage` не специфицировано, будет использовано средство доступа (accessor). Это может быть удобно для обхода специальной логики, которая может быть защищена в общедоступные средства доступа.

Вы можете видеть, что тип базы данных для этого поля указан в атрибуте `DbType`, как `NCHAR` длиной в пять символов. Поскольку атрибут `CanBeNull` специфицирован со значением `false`, значение этого поля в базе данных не может быть `NULL`, а поскольку специфицирован атрибут `IsPrimaryKey` со значением `true`, это поле представляет собой в таблице столбец идентичности.

Необязательно каждое свойство сущностного класса отображать на базу данных. У вас могут быть свойства времени выполнения, которые вы не собираетесь сохранять в базе данных, и это вполне нормально. Для таких свойств вы просто не специфицируете атрибут `Column`.

Также у вас могут быть хранимые столбцы, доступные только для чтения. Это обеспечивается отображением столбца и указанием свойства атрибута `Storage` на приватную переменную-член, без реализации метода `set` для свойства класса. `DataContext` может иметь доступ к приватному члену, но поскольку у данного свойства сущностного класса нет метода `set`, никто не может его изменять.

AutoSync (AutoSync enum)

Свойство атрибута `AutoSync` — это перечислимый тип (`enum`) `AutoSync`, которое инструктирует исполняющую систему о необходимости извлекать значение отображаемого столбца сразу после операции вставки или обновления записи в базе данных. Предоставляю вам право догадаться о том, какое именно значение используется по умолчанию. Не догадались? Согласно документации Microsoft, поведение по умолчанию — `Never`.

Установка атрибута свойства переопределяется, когда либо `IsDbGenerated`, либо `IsVersion` установлены в `true`.

CanBeNull (bool)

Булевское свойство атрибута `CanBeNull` специфицирует допустимость значения `NULL` для отображаемого столбца базы данных. По умолчанию это свойство атрибута равно `true`.

DbType (string)

Свойство атрибута `DbType` представляет собой строку, специфицирующую тип столбца базы данных, на который отображается данное свойство сущностного класса. Если свойство `DbType` не специфицировано, то тип столбца будет выведен из типа данных свойства сущностного класса. Это свойство атрибута используется для определения столбца только в случае вызова метода `CreateDatabase`.

expression (string)

Свойство атрибута `Expression` представляет собой строку, определяющую вычисляемый столбец базы данных. Он используется только в случае вызова метода `CreateDatabase`. По умолчанию имеет значение `String.Empty`.

IsDbGenerated (bool)

Булевское свойство атрибута `IsDbGenerated` указывает на то, что столбец таблицы базы данных, на который отображается свойство класса, генерируется автоматически базой данных. Если первичный ключ специфицирован со свойством атрибута `IsDbGenerated`, равным `true`, то свойство атрибута `DbType` должно быть установлено в `IDENTITY`.

Свойство класса, у которого значение свойства атрибута `IsDbGenerated` установлено в `true`, немедленно синхронизируется после того, как запись вставлена в базу данных, независимо от установки атрибута `AutoSync`, и синхронизированное значение свойства класса будет видимым в свойстве класса после успешного завершения метода `SubmitChanges`. Значение этого атрибута по умолчанию — `false`.

IsDiscriminator (bool)

Свойство `IsDiscriminator` атрибута хранит булевское значение, указывающее, что отображаемое свойство сущностного класса — это такое свойство, которое хранит значение дискриминатора для наследования сущностного класса. По умолчанию это свойство атрибута равно `false`. Прочтите раздел об атрибуте `InheritanceMapping` далее в этой главе, а также раздел “Наследование сущностных классов” главы 18, чтобы получит больше информации на эту тему.

IsPrimaryKey (bool)

Свойство `IsPrimaryKey` атрибута хранит булевское значение, указывающее на то, что столбец базы данных, на который отображается данное свойство класса, является частью первичного ключа. В данном случае все отображенные столбцы базы являются частями составного первичного ключа. Чтобы сущностный объект был обновляемым, по крайней мере, одно свойство сущностного класса должно иметь свойство атрибута `IsPrimaryKey`, установленное в `true`. В противном случае сущностный объект, отображенный на эту таблицу, будет доступен только для чтения. По умолчанию это свойство атрибута равно `false`.

IsVersion (bool)

Свойство `IsVersion` атрибута хранит булевское значение, специфицирующее, что отображенный столбец базы данных является либо номером версии, либо временной меткой, которая представляет информацию о версии для записи. Специфицируя свойство атрибута `IsVersion`, и устанавливая его значение в `true`, отображенный столбец базы данных будет инкрементирован, если это номер версии, и обновлен, если это временная метка, при всяком обновлении записи таблицы базы данных.

Свойство класса, чей атрибут `IsVersion` установлен в `true`, будет немедленно синхронизирован после вставки записи либо после ее обновления, независимо от установки атрибута `AutoSync`, и синхронизированное значение свойства класса будет видимо в свойстве класса немедленно после успешного завершения метода `SubmitChanges`. По умолчанию это свойство атрибута равно `false`.

Name (string)

Свойство `Name` атрибута `Column` — строка, специфицирующая наименование столбца таблицы, на который отображается это свойство класса. Если свойство `Name` атри-

бута не специфицировано, то имя свойства класса будет отображено по умолчанию на одноименный столбец базы данных.

Storage (string)

Свойство Storage атрибута — это строка, специфицирующая приватную переменную-член, в которой хранится значение свойства сущностного класса. Это позволяет LINQ to SQL миновать общедоступное средство доступа свойства и содержащуюся в нем бизнес-логику, и позволяет получить непосредственный доступ к приватной переменной-члену. Если свойство Storage атрибута не специфицировано, то по умолчанию будет использовано общедоступное средство доступа свойства.

UpdateCheck (UpdateCheck enum)

Свойство атрибута UpdateCheck — это перечислимый тип (enum) UpdateCheck, которое контролирует поведение определения оптимистического обнаружения параллелизма для свойства класса и отображаемого на него столбца базы данных, если ни одно из отображаемых свойств сущностного класса не имеет свойства атрибута IsVersion, установленного в true. Три допустимых значения этого свойства: UpdateCheck.Always, UpdateCheck.WhenChanged и UpdateCheck.Never. Если ни одно из свойств сущностного класса не имеет свойства атрибута IsVersion, установленного в true, то значение свойства UpdateCheck атрибута по умолчанию будет Always. Вы можете прочесть в главе 17 дополнительную информацию об этом свойстве атрибута и его эффекте.

Association

Атрибут Association используется для определения отношений между двумя таблицами, таких как отношение первичного и внешнего ключа. В этом контексте сущность, чья отображенная таблица имеет первичный ключ, является родителем, а сущность, чья отображенная таблица содержит внешний ключ, — ребенком. Приведем важные части двух сущностных классов, содержащих ассоциацию.

Ассоциация от родительского (Customer) сущностного класса

```
[Association(Name="FK_Orders_Customers", Storage="_Orders",
OtherKey="CustomerID",
DeleteRule="NO ACTION")]
public EntitySet<Order> Orders
{
```

Ассоциация от дочернего (Order) сущностного класса

```
[Association(Name="FK_Orders_Customers", Storage="_Customer",
ThisKey="CustomerID",
IsForeignKey=true)]
public Customer Customer
{
```

Для этой дискуссии об атрибуте Association и его свойствах я использую сущностный класс Customer в качестве примера родителя, и сущностный класс Order в качестве примера дочернего элемента. Таким образом, я представлю важные атрибуты Association, которые существуют в обоих сущностных классах — Customer и Order.

Если говорить о свойствах атрибута Association, то некоторые его свойства относятся к классу, в котором существует атрибут Association, а другие свойства — к другому ассоциированному сущностному классу. В этом контексте класс, в котором существует атрибут Association, можно назвать исходным классом, а другой ассоциированный сущностный класс — целевым классом. Поэтому, если я говорю о свойствах атрибута Association, специфицированного в сущностном классе Customer, то сущно-

стый класс `Customer` является исходным классом, а сущностный класс `Order` — целевым классом. Если я говорю о свойствах атрибута `Association`, специфицированных в классе `Order`, то исходным классом выступает `Order`, а целевым — `Customer`.

Атрибут `Association` определяет, что исходный сущностный класс `Customer` имеет отношение с целевым сущностным классом `Order`.

В предшествующих примерах свойство `Name` атрибута специфицировано для указания имени отношения. Значение этого свойства атрибута соответствует имени ограничения внешнего ключа в базе данных и будет использовано для создания этого ограничения при вызове метода `CreateDatabase`. Свойство `Storage` атрибута также специфицировано. Оно позволяет LINQ to SQL обойти общедоступное средство доступа (`accessor`), чтобы получить доступ к значению свойства сущностного класса.

В варианте ассоциации первичного ключа к внешнему, сущностный класс, являющийся родителем в отношении, будет хранить ссылку на дочерний сущностный класс в коллекции `EntitySet<T>`, поскольку детей может быть много. Сущностный класс — ребенок будет хранить ссылку на родительский сущностный класс в `EntityRef<T>`, поскольку он только один. Прочтите разделы, озаглавленные “`EntitySet<T>`” и “`EntityRef<T>`” далее в этой главе, а также раздел “Немедленная загрузка с помощью класса `DataLoadOptions`” главы 14, чтобы получить больше информации об ассоциациях и их характеристиках.

DeleteOnNull (bool)

Свойство `DeleteOnNull` атрибута принимает булевское значение, которое, будучи установленным в `true`, указывает, что сущностный объект с дочерней стороны ассоциации должен быть удален в случае, если ссылка на него у родителя устанавливается в `null`.

Значение этого свойства атрибута определяется `SQLMetal`, если в базе данных специфицировано правило удаления "CASCADE" для ограничения внешнего ключа и столбец внешнего ключа не допускает `null`.

DeleteRule (string)

Свойство `DeleteRule` атрибута принимает строку, которая специфицирует правило удаления (Delete Rule) для ограничения внешнего ключа. Оно используется только LINQ to SQL, когда создается ограничение в базе данных методом `CreateDatabase`.

Допустимые значения: "NO ACTION", "CASCADE", "SET NULL" и "SET DEFAULT". Определение каждого ищите в документации SQL Server.

HasForeignKey (bool)

Свойство `HasForeignKey` атрибута хранит булевское значение, которое, будучи установленным в `true`, говорит о том, что сущностный класс исходной стороны ассоциации содержит внешний ключ; т.е. является дочерней стороной этой ассоциации. По умолчанию это свойство атрибута установлено в `false`.

В примерах атрибута `Association`, показанных ранее для сущностных классов `Customer` и `Order`, поскольку данный атрибут, указанный в сущностном классе `Order`, содержит свойство `HasForeignKey` атрибута, установленное в `true`, класс `Order` является дочерним в данном отношении.

IsUnicode (bool)

Свойство `IsUnicode` атрибута хранит булевское значение, которое, будучи установленным в `true`, специфицирует наличие ограничения уникальности на внешнем ключе, т.е. задает отношение "один к одному" между двумя сущностными классами. По умолчанию это свойство атрибута установлено в `false`.

Name (string)

Свойство Name атрибута Association — это строка, указывающая имя ограничения внешнего ключа. Она используется для создания ограничения внешнего ключа при вызове метода CreateDatabase. Также оно будет использовано для различия нескольких отношений между одними и теми же сущностями. В этом случае, если обе стороны отношения специфицируют имя, оно должно быть одинаковым.

Если у вас нет нескольких отношений между одними и теми же сущностными классами, и вы не вызываете метод CreateDatabase, это свойство атрибута не обязательно. Для этого свойства значение по умолчанию не предусмотрено.

OtherKey (string)

Свойство OtherKey атрибута Association — строка, хранящая разделенный запятыми список всех свойств целевого сущностного класса, составляющих ключ, будь то внешний или первичный, в зависимости от того, какую сторону отношения занимает целевая сущность. Если это свойство атрибута не специфицировано, то члены первичного ключа целевого сущностного класса используются по умолчанию.

Важно понимать, что атрибут Association, специфицированный на каждой стороне отношения ассоциации — Customer и Order — указывает, где находятся ключи каждой из сторон. Атрибут Association, специфицированный в сущностном классе Customer, говорит о том, какие свойства сущностных классов Customer и Order составляют ключ для отношения.

Иногда может показаться, что каждая сторона всегда специфицирует местоположения ключа обеих сторон. Поскольку обычно на родительской стороне отношения используется первичный ключ таблицы, свойство ThisKey атрибута не должно быть специфицировано, поскольку первичный ключ используется по умолчанию. А на дочерней стороне свойство OtherKey атрибута не обязано быть специфицировано, так как по умолчанию применяется первичный ключ родителя. Таким образом, часто можно видеть свойство OtherKey специфицированным только на родительской стороне, а свойство ThisKey атрибута — только на дочерней стороне. Но благодаря значениям по умолчанию, и родитель, и ребенок знают ключи на обеих сторонах.

Storage (string)

Свойство Storage атрибута Association — это строка, специфицирующая приватную переменную-член, в которой хранится значение свойства сущностного класса. Это позволяет LINQ to SQL миновать общедоступное средство доступа и непосредственно обратиться к приватной переменной-члену. Это позволяет пропустить любую бизнес-логику из средств доступа. Если свойство Storage атрибута не специфицировано, то по умолчанию используются общедоступные средства доступа данного свойства.

В Microsoft рекомендуют, чтобы оба члена отношения ассоциации были свойствами сущностного класса с отдельными переменными-членами для хранения данных со специфицированными свойствами Storage атрибута отношения.

ThisKey (string)

Свойство ThisKey атрибута представляет собой строку, содержащую разделенный запятыми список всех свойств сущностного класса, составляющих ключ, будь то внешний или первичный, в зависимости от того, какую сторону отношения представляет исходная сущность, что определяется свойством IsForeignKey того же атрибута. Если свойство ThisKey атрибута не специфицировано, то члены первичного ключа исходного сущностного класса используются по умолчанию.

Поскольку ранее показанный пример атрибута Association для сущностного класса Order специфицирует свойство IsForeignKey, мы знаем, что сущностный класс

Customer представляет родительскую сторону отношения — ту, что содержит первичный ключ. Поскольку атрибут *Association* не указывает свойства *ThisKey*, мы знаем, что значение первичного ключа таблицы *Customer* становится внешним ключом в ассоциированной таблице *Orders*.

Так как атрибут *Association*, показанный ранее для сущностного класса *Order*, специфицирует атрибут *IsForeignKey* со значением *true*, мы знаем, что таблица *Orders* будет стороной ассоциации, содержащей внешний ключ. И, поскольку атрибут *Association* специфицирует значение свойства *ThisKey* этого атрибута, равное *CustomerID*, мы знаем, что столбец *CustomerID* таблицы *Orders* будет местом хранения внешнего ключа.

Важно понимать, что атрибут *Association*, специфицированный на каждой стороне отношения ассоциации — *Customer* и *Order* — указывает местонахождение ключей обеих сторон. Атрибут *Association*, специфицированный в сущностном классе *Customer*, указывает, какие свойства этого класса и класса *Orders* составляют ключ для отношения. Аналогично атрибут *Association*, специфицированный в классе *Orders*, указывает, какие свойства этого класса и класса *Customer* составляют ключ отношения.

Иногда может показаться, что каждая сторона всегда специфицирует местоположения ключа обеих сторон. Поскольку обычно на родительской стороне отношения используется первичный ключ таблицы, свойство *ThisKey* атрибута не обязано быть специфицировано, поскольку первичный ключ применяется по умолчанию. А на дочерней стороне свойство *OtherKey* атрибута не обязано быть специфицированным, так как по умолчанию используется первичный ключ родителя. Таким образом, часто можно видеть свойство *OtherKey* специфицированным только на родительской стороне, а свойство *ThisKey* атрибута — специфицированным только на дочерней стороне. Но благодаря значениям по умолчанию, и родитель, и ребенок знают ключи на обеих сторонах.

Function

Атрибут *Function* определяет, что метод класса вызывает хранимую процедуру либо пользовательскую функцию со скалярным или табличным значением возврата. Приведем важную часть класса, производного от *DataContext*, связанную с хранимой процедурой:

Атрибут Function, отображающий метод на хранимую процедуру в базе данных Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    ...
}
```

Отсюда мы видим, что есть метод по имени *GetCustomerAndOrders*, который вызывает хранимую процедуру по имени “*Get Customer And Orders*”. Мы знаем, что этот метод отображается на хранимую процедуру, а не определяемую пользователем функцию, поскольку свойство *IsComposable* атрибута не специфицировано и потому по умолчанию равно *false*, что отображает метод на хранимую процедуру. Мы также можем видеть, что он возвращает множественные формы результатов, потому что специфицировано для атрибута *ResultTypes*.

Написать ваш класс-наследник *DataContext*, чтобы он мог вызывать хранимую процедуру — не такая тривиальная задача, как отображение сущностного класса на табли-

цу. В дополнение к соответствующим атрибутам вы должны также вызывать соответствующую версию метода `ExecuteMethodCall` класса `DataContext`. Об этом методе вы узнаете из главы 16.

Конечно, как правило, это необходимо только при написании вашего собственного класса-наследника `DataContext`, поскольку `SQLMetal` и `Object Relational Designer` делают это за вас.

Существенная часть класса-наследника `DataContext`, ссылающаяся на определенную пользователем функцию, представлена ниже.

Атрибут `Function`, отображающий метод на определенную пользователем функцию в базе данных Northwind

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
    ...
}
```

Отсюда мы видим, что есть метод по имени `MinUnitPriceByCategory`, который будет вызывать определенную пользователем функцию по имени "MinUnitPriceByCategory". Мы знаем, что метод отображается именно на определенную пользователем функцию, а не хранимую процедуру, поскольку свойство `IsComposable` атрибута установлено в `true`. Также мы можем видеть по атрибуту `return`, что определенная пользователем функция вернет значение типа `Money`.

Написание вашего наследника класса `DataContext` так, чтобы он мог вызывать определяемые пользователем функции, не настолько просто, как отображение существенного класса на таблицу. В дополнение к соответствующим атрибутам вы должны также вызывать `ExecuteMethodCall` класса `DataContext` для функций, возвращающих скалярные значения, или же метод `CreateMethodCallQuery` для функций, возвращающих табличные значения. Вы также прочтете об этих методах в главе 16.

Конечно, как правило, это необходимо при написании вашего собственного класса-наследника `DataContext`, потому что `SQLMetal` и `Object Relational Designer` делают это за вас.

`IsComposable (bool)`

Свойство `IsComposable` атрибута содержит булевское значение, указывающее на то, что отображающий метод вызывает хранимую процедуру или определенную пользователем функцию. Если значение `IsComposable` равно `true`, метод отображается на определяемую пользователем функцию. Если значение `IsComposable` равно `false`, метод отображается на хранимую процедуру. Это свойство атрибута по умолчанию имеет значение `false`, если не специфицировано, т.е. метод, отображаемый атрибутом `Function`, по умолчанию отображается на хранимую процедуру, если свойство `IsComposable` атрибута не специфицировано.

`Name (string)`

Свойство `Name` атрибута содержит строку, специфицирующую действительное имя хранимой процедуры или определяемой пользователем функции в базе данных. Если свойство `Name` атрибута не специфицировано, то имя хранимой процедуры или пользовательской функции предполагается совпадающим с именем метода.

return

Атрибут `return` используется для указания возвращаемого типа данных из хранимой процедуры или определяемой пользователем функции. Обычно содержит атрибут `parameter`.

Атрибут `return` для класса Northwind

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal> MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
    ...
}
```

В приведенном коде видно, что определенная пользователем функция вернет значение типа `Money`, на что указывает атрибут `return` и свойство `DbType` встроенного атрибута `Parameter`.

ResultType

Атрибут `ResultType` отображает тип данных, возвращенный хранимой процедурой, на класс .NET, в котором сохраняются возвращенные данные. Хранимые процедуры, возвращающие множественные формы, специфицируют несколько атрибутов `ResultType` в соответствующем порядке.

Атрибуты `ResultType` класса Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    ...
}
```

В приведенном коде видно, что хранимая процедура, на которую отображается этот метод, сначала возвратит форму типа `GetCustomerAndOrdersResult1`, за которой следует форма типа `GetCustomerAndOrdersResult2`.

Parameter

Атрибут `Parameter` отображает параметр метода на параметр хранимой процедуры или пользовательской функции базы данных. Ниже приведена существенная часть кода класса-наследника `DataContext`.

Атрибут `Parameter` из класса Northwind

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    ...
}
```

Отсюда видно, что метод `GetCustomerAndOrders`, отображаемый на хранимую процедуру базы данных по имени "Get Customer And Orders", передает хранимой процедуре параметр типа `NChar(5)`.

DbType (string)

Свойство DbType атрибута — строка, специфицирующая тип данных базы и модификаторы параметра хранимой процедуры или определенной пользователем функции.

Name (string)

Свойство Name атрибута содержит строку, специфицирующую действительное имя параметра хранимой процедуры или определенной пользователем функции. Если свойство Name атрибута не специфицировано, то имя хранимой процедуры или пользовательской функции предполагается совпадающим с именем параметра метода.

InheritanceMapping

Атрибут InheritanceMapping используется для отображения кода дискриминатора (определителя) на базовый класс или подкласс базового класса. Код дискриминатора — это значение столбца сущностного класса, специфицированного в качестве дискриминатора, который определен как свойство сущностного класса, чье свойство IsDiscriminator атрибута установлено в true.

Например, рассмотрим следующий атрибут InheritanceMapping.

Атрибут InheritanceMapping

```
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
```

Приведенный атрибут InheritanceMapping определяет, что если запись базы данных имеет значение "G" в столбце дискриминатора, т.е. ее код дискриминатора — "G", то нужно создать запись, как объект Shape с использованием класса Shape. Поскольку свойство IsDefault атрибута установлено в true, если код дискриминатора записи не соответствует ни одному значению Code атрибута InheritanceMapping, эта запись будет создана в виде объекта класса Shape.

Чтобы использовать отображение наследования, при объявлении сущностного класса одному из его свойств назначается свойство IsDiscriminator атрибута Column, установленное в true. Это значит, что значение этого столбца будет определять по дискриминатору, какого класса экземпляр — базового или одного из его подклассов — хранит запись таблицы. Атрибут InheritanceMapping специфицирован на базовом классе для каждого из подклассов, а также на самом базовом классе. Из всех этих атрибутов InheritanceMapping один и только один должен иметь свойство IsDefault со значением true. Также эта запись таблицы базы данных, дискриминатор которой не соответствует ни одному из кодов дискриминатора, специфицированных в атрибутах InheritanceMapping, может стать экземпляром этого класса. Наверно, чаще всего атрибут InheritanceMapping специфицируют как атрибут InheritanceMapping по умолчанию.

Опять-таки, все атрибуты InheritanceMapping, специфицированные на одном из базовых классов, лишь ассоциируют код дискриминатора для базового класса или одного из его подклассов.

Поскольку база данных Northwind не содержит никаких таблиц, используемых таким образом, я приведу три класса-примера.

Некоторые примеры классов, демонстрирующие отображение наследования

```
[Table]
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
[InheritanceMapping(Code = "S", Type = typeof(Square))]
[InheritanceMapping(Code = "R", Type = typeof(Rectangle))]
public class Shape
{
```

```
[Column(IsPrimaryKey = true, IsDbGenerated = true,
    DbType = "Int NOT NULL IDENTITY")]
public int Id;

[Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
public string ShapeCode;
[Column(DbType = "Int")]
public int StartingX;
[Column(DbType = "Int")]
public int StartingY;
}

public class Square : Shape
{
    [Column(DbType = "Int")]
    public int Width;
}

public class Rectangle : Square
{
    [Column(DbType = "Int")]
    public int Length;
}
```

Здесь мы можем видеть, что я отобразил класс Shape на таблицу, и поскольку не специфицировал свойство атрибута Name, класс Shape будет отображен по умолчанию на таблицу по имени Shape.

Затем вы видите три атрибута InheritanceMapping. Первый определяет, что если значение столбца дискриминатора записи таблицы базы данных Shape равно "G", то из записи должен быть создан экземпляр объекта Shape с использованием класса Shape. Для моих целей я выбрал "G" в качестве обозначения *generic* (обобщения), в том смысле, что это обобщенная неизвестная фигура (shape). Поскольку в классе Shape есть свойство ShapeCode, которое является дискриминатором, в том смысле, что оно имеет свойство IsDiscriminator атрибута, установленное в true, если запись имеет значение ShapeCode, равное "G", из этой записи будет создан объект Shape.

Также вы можете видеть, что первый атрибут InheritanceMapping имеет свойство IsDefault атрибута, установленное в true, так что если значение столбца ShapeCode записи Shape равно "S", то из этой записи будет создан объект Square.

Третий атрибут InheritanceMapping ассоциирует код дискриминатора "R" с классом Rectangle. Поэтому, если запись в таблице Shape базы данных имеет ShapeCode, равное "R", то запись станет экземпляром объекта Rectangle.

Любая запись с ShapeCode, отличающимся от того, что специфицирован, превратится в экземпляр объекта Shape, потому что Shape — класс по умолчанию, как указано в свойстве IsDefault атрибута.

На заметку! Отображение наследования обсуждается вместе с примерами в главе 18.

Code (object)

Свойство Code атрибута специфицирует код дискриминатора для отображения на специфицированный класс, который будет специфицирован свойством Type атрибута.

IsDefault (bool)

Свойство атрибута IsDefault содержит булевское значение, которое специфицирует, какой атрибут InheritanceMapping должен использоваться, если столбец дискриминатора записи таблицы базы данных не соответствует ни одному коду дискриминатора, специфицированному в любом из атрибутов InheritanceMapping.

Type (Type)

Свойство Type атрибута специфицирует тип класса для создания экземпляра, представляющего запись, когда столбец дискриминатора соответствует отображеному коду дискриминатора.

Совместимость типов данных

Некоторые атрибуты сущностного класса имеют свойство атрибута DbType, в котором вы можете специфицировать тип данных столбца таблицы. Это свойство атрибута используется только при создании базы данных методом CreateDatabase. Поскольку отображение между типами данных .NET и типами данных SQL Server не строится по схеме “один к одному”, вам нужно специфицировать свойство DbType атрибута, если вы планируете вызывать метод CreateDatabase.

Поскольку типы данных исполняющей системы .NET Common Language Runtime (CLR), используемые в вашем коде LINQ, не совпадают с типами данных, применяемыми базой данных, вам следует обратиться к документации MSDN, касающейся отображению типов между SQL и CLR (LINQ to SQL). В этой документации есть матрица, которая определяет поведение при преобразовании между типа данных CLR и типа данных SQL. Если вам трудно найти эту информацию, загляните в раздел “Additional LINQ to SQL Resources” моего Web-сайта www.linqdev.com. Там вы найдете ссылку на документацию MSDN.

Следует иметь в виду, что некоторые преобразования типов данных не поддерживаются, а другие могут привести к потере данных, в зависимости от типов, участвующих в преобразовании и его направления.

Однако я думаю, что в большинстве случаев вас удовлетворит преобразование, и оно не составит проблемы. При написании примеров для глав, посвященных LINQ to SQL, я никогда не сталкивался с проблемами, вызванными преобразованием типов данных. Конечно, нужно придерживаться здравого смысла. Если вы попытаетесь отображать очевидно несовместимые типы, такие как числовые типы данных .NET на символьный тип данных SQL, то неизбежно столкнетесь с некоторыми проблемами.

XML-схема внешнего файла отображения

Как я говорил в разделе главы 13, посвященном SQLMetal, вы не только можете отображать сущностные классы на базу данных, но также можете использовать внешний XML-файл отображения. О том, как применять внешний XML-файл отображения вы узнаете при описании конструкторов для класса DataContext в главе 16.

Как говорилось в главе 13, простейший способ получить внешний XML-файл отображения заключается в вызове программы SQLMetal с указанием опции /map — в этом случае он будет сгенерирован для вас. Однако если вы намерены создавать файл отображения вручную, вам следует знать его схему.

Обратитесь к документации MSDN за схемой внешнего отображения, озаглавленной “External Mapping Reference (LINQ to SQL)”. Если вы затрудняетесь в поиске этой информации, загляните в раздел “Additional LINQ to SQL Resources” моего Web-сайта www.linqdev.com. Там вы найдете ссылку на документацию MSDN.

Сравнение проекций на сущностные и на несущностные классы

При выполнении запросов LINQ to SQL у вас есть два выбора для проекции возвращенных результатов. Вы можете спроектировать результаты на сущностный класс, или же спроектировать их на несущностный класс, который может быть именованным или

анонимным. Между проецированием на сущностный и несущностный класс имеется существенное отличие.

При проецировании на сущностный класс этот сущностный класс использует преимущества службы отслеживания идентичности объектов `DataContext`, слежения за изменениями и обработки изменений. Это значит, что вы не можете изменять несущностные классы и сохранять их посредством LINQ to SQL. Это оправдано, поскольку такой класс не будет иметь необходимых атрибутов или файла отображения класса на базу данных. И с другой стороны, если он имеет атрибуты или файл отображения, то он по определению является сущностным классом.

Ниже приведен пример запроса, проецируемого на сущностный класс.

Проектирование на сущностный класс представляет службы `DataContext`

```
IEnumerable<Customer> custs = from c in db.Customers
                                select c;
```

После этого запроса я мог бы вносить изменения в любой из сущностных объектов `Customer` в последовательности `custs`, и сохранять их посредством вызова метода `SubmitChanges`.

Далее показан пример запроса, проецируемого на несущностный класс.

Проектирование на несущностный класс не обеспечивает доступа к службам `DataContext`

```
var custs = from c in db.Customers
            select new { Id = c.CustomerID, Name = c.ContactName };
```

Выполняя проектирование результата на анонимный класс, я не смогу сохранить никакие проведенные мною изменения в каждый объект последовательности `custs` вызовом метода `SubmitChanges`.

Я упоминал о том, что есть одно специализированное исключение, имеющее отношение к получению преимуществ отслеживания идентичности и обработки изменений при проектировании несущностных классов. Это исключение случается, когда проецируемый класс содержит члены типа сущностных классов. Пример приведен в листинге 15.1.

Листинг 15.1. Проектирование на несущностный класс, содержащий сущностные классы

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var cusorders = from o in db.Orders
                where o.Customer.CustomerID == "CONSH"
                orderby o.ShippedDate descending
                select new { Customer = o.Customer, Order = o };

// Захватить первый заказ.
Order firstOrder = cusorders.First().Order;
// Теперь сохраним страну поставки первого заказа, чтобы можно
// было восстановить позже.
string shipCountry = firstOrder.ShipCountry;
Console.WriteLine("Order is originally shipping to {0}", shipCountry);
// Теперь изменим страну поставки с Великобритании на США.
firstOrder.ShipCountry = "USA";
db.SubmitChanges();

// Запрос для проверки того, была ли страна на самом деле изменена.
string country = (from o in db.Orders
                    where o.Customer.CustomerID == "CONSH"
                    orderby o.ShippedDate descending
                    select o.ShipCountry).FirstOrDefault<string>();
Console.WriteLine("Order is now shipping to {0}", country);
```

```
// Восстановить состояние базы данных, чтобы можно было пример запустить снова.
firstOrder.ShipCountry = shipCountry;
db.SubmitChanges();
```

В листинге 15.1 я запрашиваю заказы, поступившие от заказчика "CONSH". Возвращенные заказы я проецирую на анонимный тип, содержащий `Customer` и каждый `Order`. Сам анонимный класс не принимает служб `DataContext`, таких как отслеживание идентичности, отслеживание и обработка изменений, но его компоненты типа `Customer` и `Order` делают это, будучи сущностными классами. Затем я выполняю другой запрос на результатах предыдущего, чтобы получить первый `Order`. После этого сохраняю копию оригинальной `ShipCountry` объекта `Order`, чтобы можно было восстановить его в конце примера, и отображаю этот оригинальной `ShipCountry` на экране. Затем заменяю `ShipCountry` в `Order` и сохраняю изменение вызовом метода `SubmitChanges`. Затем снова запрашиваю `ShipCountry` для этого заказа из базы данных и отображаю его, просто чтобы доказать, что в базе данных действительно произошло изменение. Это доказывает правильность работы метода `SubmitChanges`, а также тот факт, что компоненты сущностного класса моего анонимного типа получают в свое распоряжение службы объекта `DataContext`. Далее я сбрасываю `ShipCountry` обратно в исходное значение и сохраняю его, чтобы пример можно было запустить опять, и никакие последующие примеры не пострадали. Вот результат запуска программы из листинга 15.1:

```
Order is originally shipping to UK
Order is now shipping to USA
```

Листинг 15.1 содержит пример, в котором я проецирую результаты запроса на тип несущностного класса, но поскольку он включает в себя объекты сущностного класса, я могу воспользоваться преимуществами отслеживания идентичности, отслеживания и обработки изменений, которые предоставлены классом `DataContext`.

Есть одно интересное замечание относительно приведенного выше кода. Вы заметите, что запрос, получающий ссылку на первый `Order`, выделен полужирным. Я сделал это, чтобы привлечь ваше внимание. Обратите внимание, что я вызываю операцию `First` перед выбором части интересующего меня элемента последовательности — члена `Order`. Я делаю это для повышения производительности, потому что, чем быстрее вы можете сузить поле поиска, тем выше производительность.

Предпочтайте инициализацию объектов параметризации конструкторов при проектировании

Вы вольны проецировать результат на классы до окончания запроса для последующих операций запроса, но делая это, предпочтайте инициализацию объектов параметризованным конструктором. Чтобы понять — почему, заглянем в листинг 15.2, который использует инициализацию объектов при проектировании.

Листинг 15.2. Проектирование с использованием инициализации объектов

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new { Name = c.ContactName, Phone = c.Phone } into co
               orderby co.Name
               select co;
foreach (var contact in contacts)
{
    Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}
```

Обратите внимание, что в листинге 15.2 я проецирую результат на анонимный класс и использую инициализацию объекта для наполнения создаваемых анонимных объектов. Посмотрим на вывод листинга 15.2:

```
SELECT [t0].[ContactName] AS [Name], [t0].[Phone]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input String (Size = 12; Prec = 0; Scale = 0) [Buenos Aires]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Patricia Simpson - (1) 135-5555
Sergio Gutiérrez - (1) 123-5555
Yvonne Moncada - (1) 135-5333
```

Меня даже не интересует вывод результатов запроса. В действительности я хочу увидеть сгенерированный запрос SQL. Поэтому вы спросите: “Зачем тогда нужен цикл foreach?”. Дело в том, что из-за того, что выполнение запроса отложено, без foreach запрос на самом деле вообще бы не выполнился.

Существенная для нашей дискуссии часть запроса LINQ to SQL заключена в конструкциях `select` и `orderby`. В моем запросе LINQ to SQL я инструктирую его о необходимости создания члена по имени `Name` в анонимном классе, который заполняется полем `ContactName` из таблицы `Customers`. Затем я указываю запросу отсортировать результат по члену `Name` анонимного объекта, на который выполняется проекция. Объект `DataContext` содержит всю информацию, переданную ему. Инициализация объекта эффективно отображает исходное поле `ContactName` из класса `Customer` на целевое поле `Name` анонимного класса, и объект `DataContext` отвечает за это отображение. На основании этой информации он способен понять, что я на самом деле сортирую `Customers` по полю `ContactName`, поэтому он может сгенерировать запрос SQL, который делает это. Если вы взглянете на сгенерированный SQL-запрос, то убедитесь, что именно это он и делает.

Теперь давайте посмотрим, что происходит, когда я проецирую результат на именованный класс с использованием параметризованного конструирования. Первым делом, мне нужен именованный класс. Я использую приведенный ниже.

Именованный класс, используемый в листинге 15.3

```
class CustomerContact
{
    public string Name;
    public string Phone;
    public CustomerContact(string name, string phone)
    {
        Name = name;
        Phone = phone;
    }
}
```

Обратите внимание, что здесь есть единственный конструктор, принимающий два параметра — `name` и `phone`. Теперь взглянем на тот же код, что и в листинге 15.2, но с тем отличием, что в листинге 15.3 код будет модифицирован для проецирования на класс `CustomerContact` с использованием параметризованного конструктора.

Листинг 15.3. Проецирование с использованием параметризованного конструирования

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
db.Log = Console.Out;
```

```

var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new CustomerContact(c.ContactName, c.Phone) into co
               orderby co.Name
               select co;
foreach (var contact in contacts)
{
    Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}

```

Опять-таки, я обращаю ваше внимание на конструкции `select` и `orderby`. Как вы можете видеть в листинге 15.3, вместо проектирования на анонимный класс я здесь я выполняю проекцию на класс `CustomerContact`. И вместо инициализации созданных объектов я применяю параметризованный конструктор. Этот код компилируется успешно, но что произойдет при запуске примера? А произойдет следующее исключение:

`Unhandled Exception: System.InvalidOperationException: Binding error : Member 'LINQChapter15.CustomerContact.Name' is not a mapped member of 'LINQChapter15.CustomerContact'.`

Необработанное исключение: System.InvalidOperationException: Ошибка связывания : Член 'LINQChapter15.CustomerContact.Name' не является отображенным членом 'LINQChapter15.CustomerContact'.

...

Так что же произошло? Взглянув на предшествующий запрос LINQ to SQL, спросите себя: “Откуда `DataContext` знает, какое поле класса `Customer` отображается на член `CustomerContact.Name`, по которому я пытаюсь выполнить упорядочивание?”. В листинге 15.2, поскольку я передал это в именах полей анонимного класса, было известно, что исходным полем класса `Customer` было `ContactName`, а целевым полем анонимного класса — поле `Name`. В листинге 15.3 это отображение не происходит в запросе LINQ to SQL, а происходит в конструкторе класса `CustomerContact`, о чем `DataContext` не имеет понятия. Поэтому он не знает, по какому полю исходного класса `Customer` следует выполнить упорядочивание при генерации оператора SQL. Отсюда и проблемы.

Однако вполне безопасно использовать параметризованное конструирование до тех пор, пока ничего в запросе проекции не ссылается на члены именованного класса, как показано в листинге 15.4.

Листинг 15.4. Проектирование с использованием параметризованного конструирования без ссылок на члены

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = from c in db.Customers
               where c.City == "Buenos Aires"
               select new CustomerContact(c.ContactName, c.Phone);
foreach (var contact in contacts)
{
    Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}

```

В листинге 15.4, поскольку я использую синтаксис выражений запросов, и поскольку этот синтаксис требует, чтобы запрос заканчивался конструкцией `select`, я вправе использовать параметризованное конструирование с последней конструкцией `select` запроса. Я вправе делать это, потому что ничего не следует после конструкции `select`, содержащей вызов параметризованного конструктора, который ссылается на именованные члены класса. Вот результат работы программы из листинга 15.4:

```

SELECT [t0].[ContactName], [t0].[Phone]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] = @p0
-- @p0: Input String (Size = 12; Prec = 0; Scale = 0) [Buenos Aires]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
Patricia Simpson - (1) 135-5555
Yvonne Moncada - (1) 135-5333
Sergio Gutiérrez - (1) 123-5555

```

Однако, поскольку использование стандартного точечного синтаксиса не требует, чтобы запрос оканчивался конструкцией `select`, небезопасно предполагать, что запрос будет работать просто потому, что проекция на именованный класс с применением параметризованного конструирования происходит в последней проекции. Листинг 15.5 содержит пример, использующий стандартный точечный синтаксис с последней проекцией, которая применяет параметризованное конструирование, но так как последующая часть запроса ссылается на члены именованного класса, запрос генерирует исключение.

Листинг 15.5. Проектирование с использованием параметризованного конструирования, ссылающееся на члены

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
var contacts = db.Customers.Where(c => c.City == "Buenos Aires").
    Select(c => new CustomerContact(c.ContactName, c.Phone)).
    OrderBy(c => c.Name);
foreach (var contact in contacts)
{
    Console.WriteLine("{0} - {1}", contact.Name, contact.Phone);
}

```

Запрос в листинге 15.5 очень похож на запрос из листинга 15.4 за исключением того, что здесь я использую стандартный синтаксис точечной нотации вместо синтаксиса выражения запроса, а также перемещаю операцию `OrderBy` в конец запроса. Я применяю параметризованное конструирование в финальной проекции, но это не работает, потому что операция `OrderBy` ссылается на член именованного класса. Вот результат работы кода из листинга 15.5:

```

Unhandled Exception: System.InvalidOperationException: Binding error : Member
'LINQChapter15.CustomerContact.Name' is not a mapped member of
'LINQChapter15.CustomerContact'.
Необработанное исключение: System.InvalidOperationException: Ошибка связывания :
Член 'LINQChapter15.CustomerContact.Name' не является отображенным членом
'LINQChapter15.CustomerContact'.
...

```

Расширение сущностных методов частичными методами

Одной из двух проблем, на которые жаловались разработчики, имевшие дело с ранними адаптациями LINQ, была невозможность узнать, что происходит внутри сущностного класса. В ранний период развития LINQ у разработчиков не было возможности узнать, когда свойство объекта сущностного класса претерпевает изменение, или когда создается сам объект сущностного класса, кроме как посредством модификации скомпилированного кода этого сущностного класса, чего, как мы знаем, делать ни в коем

случае не стоит. Любая модификация генерированного кода сущностного класса будет утеряна при следующей перегенерации, что неприемлемо. К счастью, инженеры из Microsoft к этим жалобам прислушались.

В главе 2 я говорил вам о частичных методах, и вот мы добрались до того случая, когда такие методы чрезвычайно полезны. В Microsoft определили, когда в течение жизненного цикла сущностного класса разработчики наиболее вероятно заинтересованы в получении уведомлений, и добавили вызовы частичных методов.

Ниже представлен список поддерживаемых частичных методов.

Поддерживаемые частичные методы для сущностного класса

```
partial void OnLoaded();
partial void OnValidate(ChangeAction action);
partial void OnCreated();
partial void On[Property]Changing([Type] value);
partial void On[Property]Changed();
```

Последние два метода из этого списка получают имя, заменяя часть "[Property]" именем свойства, а часть "[Type]" — типом данных этого свойства. Чтобы продемонстрировать некоторые частичные методы, поддерживаемые сущностными классами, я добавлю следующий код к сущностному классу Contact.

Дополнительное объявление для класса Contact для реализации некоторых частичных методов

```
namespace nwind
{
    public partial class Contact
    {
        partial void OnLoaded()
        {
            Console.WriteLine("OnLoaded() called.");
        }
        partial void OnCreated()
        {
            Console.WriteLine("OnCreated() called.");
        }
        partial void OnCompanyNameChanging(string value)
        {
            Console.WriteLine("OnCompanyNameChanging() called.");
        }
        partial void OnCompanyNameChanged()
        {
            Console.WriteLine("OnCompanyNameChanged() called.");
        }
    }
}
```

Обратите внимание, что я специфицировал пространство имен nwind. Это необходимо, поскольку пространство имен для моего объявления класса должно совпадать с пространством имен расширяемого класса. Поскольку я специфицировал пространство имен nwind при генерации моего сущностного класса в SQLMetal, я также должен объявить мой частичный класс Contact в том же пространстве имен nwind. В вашем реальном рабочем коде вы, вероятно, решите создать отдельный модуль, в который поместите это объявление частичного класса.

Заметьте, что я представил реализации для методов OnLoaded, OnCreated, OnCompanyNameChanging и OnCompanyNameChanged. Однако все, что они делают — это

отображение сообщения на консоли. Конечно, в своих реализациях вы можете, делать то, что хотите.

Теперь давайте взглянем на некоторый код, демонстрирующий частичные методы. В листинге 15.6 я запрашиваю запись Contact из базы данных и изменяю его свойство CompanyName.

Листинг 15.6. Запрос класса с реализованными частичными методами

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Contact contact = db.Contacts.Where(c => c.ContactID == 11).SingleOrDefault();
Console.WriteLine("CompanyName = {0}", contact.CompanyName);
contact.CompanyName = "Joe's House of Booze";
Console.WriteLine("CompanyName = {0}", contact.CompanyName);
```

В приведенном коде нет ничего особенного, за исключением того факта, что я реализовал некоторые частичные методы, поддерживаемые сущностными классами. Сначала я запрашиваю контакт и отображаю название его компании на консоли. Затем изменяю название компании и снова отображаю ее на консоли. Нажмем <Ctrl+F5> и посмотрим на вывод:

```
OnCreated() called.
OnLoaded() called.
CompanyName = B's Beverages
OnCompanyNameChanging() called.
OnCreated() called.
OnCompanyNameChanged() called.
CompanyName = Joe's House of Booze
```

Как видите, был вызван метод OnCreate, после чего — метод OnLoaded. В этой точке из базы данных была извлечена запись и загружена в сущностный объект Contact. Вы можете видеть, как наименование компании было выведено на консоль. Затем был вызван метод OnCompanyNameChanging, после которого, как ни удивительно, произошел еще один вызов OnCreate. Очевидно, что DataContext создает еще один сущностный объект Contact как часть его процедуры отслеживания изменений. Далее вызывается метод OnCompanyNameChanged, за которым — вывод нового имени компании на консоль.

Это демонстрирует, как можно расширить сущностные классы, используя частичные методы без модификации генерированного кода.

Импорт классов System.Data.Linq

Существует группа классов в пространстве имен System.Data.Linq, которые вы будете регулярно применять при использовании LINQ to SQL. В следующем разделе представлен краткий обзор этих классов, их свойств и места в схеме LINQ to SQL.

EntitySet<T>

Сущностный класс на стороне “один” в отношении “один ко многим” хранит ассоциированные с ним сущностные классы стороны “многие” в своем члене класса типа EntitySet<T>, где T — тип ассоциированного сущностного класса.

Поскольку в базе данных Northwind отношение между заказами и заказчиками построено по схеме “один ко многим”, в классе Customer соответствующие ему Orders хранятся в EntitySet<Order>.

```
private EntitySet<Order> _Orders;
```

Класс `EntitySet<T>` представляет собой специальную коллекцию, используемую LINQ to SQL. Он реализует интерфейс `IEnumerable<T>`, который позволяет выполнять на нем запросы LINQ. Кроме того, он реализует интерфейс `ICollection<T>`.

EntityRef<T>

Сущностный класс на стороне “многие” отношения “один ко многим” хранит ассоциированный с ним класс стороны “один” в члене типа `EntityRef<T>`, где `T` — тип ассоциированного сущностного класса.

Поскольку в базе данных Northwind отношение между заказами и заказчиками построено по схеме “один ко многим”, класс `Customer` сохраняется в классе `Order` в члене типа `EntityRef<Customer>`.

```
private EntityRef<Customer> _Customer;
```

Entity

Когда мы ссылаемся на ассоциированный сущностный класс, представляющий сторону “один” в отношении “один ко многим”, то склонны думать, что соответствующая переменная-член имеет тот же тип, что и сущностный класс. Например, когда мы ссылаемся на `Customer` объекта `Order`, то думаем, что объект `Customer` хранится в члене класса `Order` типа `Customer`. Однако на самом деле следует помнить, что `Customer` хранится в `EntityRef<Customer>`. Когда вам нужно обратиться к действительной ссылке на объект `Customer` из члена типа `EntityRef<Customer>`, это нужно делать через член объекта `EntityRef<Customer>` по имени `Entity`.

Бывают случаи, когда важно осознавать этот факт — например, при написании ваших собственных сущностных классов. Если вы взглянете на класс `Order`, сгенерированный SQLMetal, то заметите, что общедоступные методы `get` и `set` свойства `Customer` используют свойство `Entity` объекта `EntityRef<Customer>`, чтобы обращаться к `Customer`.

Общедоступное свойство, использующее свойство `EntityRef<T>.Entity` для доступа к действительному сущностному объекту

```
private EntityRef<Customer> _Customer;
...
public Customer Customer
{
    get
    {
        return this._Customer.Entity;
    }
    set
    {
        Customer previousValue = this._Customer.Entity;
        ...
    }
}
```

HasLoadedOrAssignedValue

Это булевское свойство позволяет узнать, присвоено ли свойству класса, хранящемуся в `EntityRef<T>`, значение, или же оно было загружено в него.

Обычно оно используется в методах `set` для ссылок на сторону “один” в ассоциации “один ко многим”, чтобы предотвратить несогласованность между свойством класса

са, хранящим идентификаторы стороны "один", и EntityRef<T>, хранящим ссылку на этого "одного".

Например, давайте посмотрим на методы set для свойств CustomerID и Customer сущностного класса Order.

Метод set для свойства CustomerID

```
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            if (this._Customer.HasLoadedOrAssignedValue)
            {
                throw new System.Data.Linq.ForeignKeyReferenceAlreadyHasValueException();
            }
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
    }
}
```

Обратите внимание в методе set для свойства CustomerID, что если EntityRef<T>, хранящий Customer, имеет свойство HasLoadedOrAssignedValue, равное true, то генерируется исключение. Это предохраняет разработчика от изменения CustomerID сущностного объекта Order, если этот Order уже имеет присвоенную ему сущность Customer. Благодаря такой защите, даже при желании невозможно достичь согласованности между CustomerID и Customer сущностного объекта Order.

В отличие от этого, в методе set для свойства Customer ссылка Customer может быть присвоена, только если свойство HasLoadedOrAssignedValue установлено в false.

Метод set для свойства Customer

```
public Customer Customer
{
    get
    {
        return this._Customer.Entity;
    }
    set
    {
        Customer previousValue = this._Customer.Entity;
        if (((previousValue != value)
            || (this._Customer.HasLoadedOrAssignedValue == false)))
        {
            this.SendPropertyChanging();
            if ((previousValue != null))
            {
                this._Customer.Entity = null;
                previousValue.Orders.Remove(this);
            }
        }
    }
}
```

```
this._Customer.Entity = value;
if ((value != null))
{
    value.Orders.Add(this);
    this._CustomerID = value.CustomerID;
}
else
{
    this._CustomerID = default(string);
}
this.SendPropertyChanged("Customer");
}
```

Проверка значения свойства HasLoadedOrAssignedValue в каждом методе set предохраняет разработчика от внесения несогласованности между ссылками CustomerID и Customer.

Table<T>

Этот тип данных LINQ to SQL использует для взаимодействия с таблицей или представлением в базе данных SQL Server. Обычно производный от `DataContext` класс, который в главах по LINQ to SQL часто назывался `[Your]DataContext`, будет иметь общедоступное свойство типа `Table<T>`, где `T` — существенный класс, для каждой таблицы базы данных в наследнике `DataContext`. Он должен выглядеть примерно так, как показано ниже.

Свойство Table<T> для таблицы Customers базы данных

```
public System.Data.Linq.Table<Customer> Customers
{
    get
    {
        return this.GetTable<Customer>();
    }
}
```

Table<T> реализует интерфейс IQueryble<T>, который сам реализует IEnumerable<T>. Это значит, что вы можете выполнять на нем запросы LINQ to SQL. Это — начальный источник данных для большинства запросов LINQ to SQL.

IExecuteResult

Когда методом ExecuteMethodCall вызывается хранимая процедура или определяемая пользователем функция, то результаты возвращаются в объекте, реализующем интерфейс IExecuteResult, как показано ниже.

Метод ExecuteMethodCall возвращает IExecuteResult

```
IExecuteResult result = this.ExecuteMethodCall(...);
```

Интерфейс `IExecuteResult` предоставляет одно свойство по имени `ReturnValue` и один метод по имени `GetParametrValue` для доступа к возвращенному значению и выходному параметру соответственно.

returnValue

Все результаты хранимой процедуры, кроме выходных параметров и результатов определяемых пользователем функций, возвращаются через переменную `IExecuteResult.ReturnValue`.

Чтобы получить доступ к возвращенному значению хранимой процедуры или функции со скалярным типом возврата, вы обращаетесь к члену `ReturnValue` возвращенного объекта. Ваш код должен выглядеть примерно так, как показано ниже.

Обращение к возвращенному значению хранимой процедуры, возвращающей целое число

```
IExecuteResult result = this.ExecuteMethodCall(...);
int resultCode = (int)(result.ReturnValue);
```

В главе 16 мы поговорим о методе `ExecuteMethodCall` и рассмотрим пример, возвращающий целое, которое возвращено хранимой процедурой.

Если хранимая процедура возвращает данные помимо ее возвращенного значения, то переменная `ReturnValue` будет реализовывать интерфейс `ISingleResult<T>` или `IMultipleResults`, который больше подходит, в зависимости от того, сколько форм данных возвращается из хранимой процедуры.

GetParameterValue

Чтобы получить доступ к выходным параметрам хранимой процедуры, вы вызываете метод `GetParameterValue` на возвращенном объекте, передавая ему начинаящийся с нуля индекс параметра, значение которого вы хотите получить. Если предположить, что хранимая процедура возвращает `CompanyName` в третьем параметре, ваш код может выглядеть примерно так, как показано ниже.

Доступ к возвращенным параметрам хранимой процедуры

```
IExecuteResult result = this.ExecuteMethodCall(..., param1, param2, companyName);
string CompanyName = (string)(result.GetParameterValue(2));
```

В главе 16 мы рассмотрим метод `ExecuteMethodCall` вместе с примерами доступа к выходным параметрам хранимой процедуры.

ISingleResult<T>

Когда хранимая процедура возвращает свои результаты в единственной форме, то они возвращаются в объекте, реализующем интерфейс `ISingleResult<T>`, где `T` — сущностный класс. Этот возвращенный объект, реализующий `ISingleResult<T>` — переменная `IExecuteResult.ReturnValue`. Ваш код должен выглядеть подобно приведенному ниже.

Обращение к возвращенным результатам, когда они имеют одну форму

```
IExecuteResult result = this.ExecuteMethodCall(...);
ISingleResult<CustOrdersOrdersResult> results =
    (ISingleResult<CustOrdersOrdersResult>)(result.ReturnValue);
```

Обратите внимание, что я просто выполняю приведение члена `ReturnValue` объекта `IExecuteResult` к `ISingleResult<T>`, чтобы получить результаты.

Поскольку `ISingleResult<T>` наследуется от `IEnumerable<T>`, хорошая новость состоит в том, что вы имеете доступ к возвращенным результатам, как к любой другой последовательности LINQ.

Обращение к результатам из ISingleResult<T>

```
foreach (CustomersByCityResult cust in results)
{
    ...
}
```

В главе 16 мы рассмотрим метод ExecuteMethodCall вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в одной форме.

ReturnValue

Интерфейс ISingleResult<T> представляет свойство ReturnValue, которое работает так же, как в интерфейсе IExecuteResult. Прочтите предыдущий раздел о свойстве ReturnValue интерфейса IExecuteResult, чтобы понять, как обращаться к этому свойству.

IMultipleResults

Когда хранимая процедура возвращает результаты в нескольких формах, то эти результаты возвращаются в объекте, реализующем интерфейс IMultipleResults. Этот возвращенный объект, реализующий IMultipleResults, является переменной IExecuteResult.ReturnValue. Ваш код должен выглядеть примерно так, как показано ниже.

Обращение к возвращенным результатам, имеющим несколько форм

```
IExecuteResult result = this.ExecuteMethodCall(...);
IMultipleResults results = (IMultipleResults)(result.ReturnValue);
```

Чтобы получить доступ к множественным возвращенным формам, вызывайте метод IMultipleResults.GetResult<T>, описанный ниже.

В главе 16 мы рассмотрим метод ExecuteMethodCall вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в нескольких формах.

Интерфейс IMultipleResults предоставляет одно свойство по имени ReturnValue для доступа к возвращенному хранимой процедурой значению, и один метод по имени GetResult<T> для извлечения IEnumerable<T> для каждой возвращенной формы, где T — сущностный класс, соответствующий форме.

ReturnValue

Интерфейс IMultipleResults<T> предоставляет свойство ReturnValue, которое работает, как и в интерфейсе IExecuteResults. Прочтите предыдущий раздел о свойстве ReturnValue интерфейса IExecuteResults, чтобы понять, как обращаться к этому свойству.

GetResult<T>

Интерфейс IMultipleResults предоставляет метод GetResult<T>, где тип T представляет тип данных, хранящий возвращенную форму. Метод GetResult<T> используется для получения повторяющихся записей указанной результирующей формы, и эти записи возвращаются в IEnumerable<T>, где T представляет сущностный класс, используемый для хранения записи формы. Ваш код должен выглядеть примерно так, как показано ниже.

Обращение к нескольким формам, возвращенным хранимой процедурой

```
[StoredProcedure(Name="A Stored Procedure")]
[ResultType(typeof(Shape1))]
[ResultType(typeof(Shape2))]

...
IExecuteResult result = this.ExecuteMethodCall(...);
IMultipleResults results = (IMultipleResults)(result.ReturnValue);
foreach(Shape1 x in results.GetResult<Shape1>()) {...}
foreach(Shape2 y in results.GetResult<Shape2>()) {...}
```

Я включил атрибуты, которые должны быть перед методом, содержащим этот код, чтобы вы видели контекст атрибутов ResultType и формы, возвращенные хранимой процедурой.

В приведенном коде мне известно, что записи, отображаемые на Shape1, будут возвращены хранимой процедурой первыми, после чего пойдут записи, отображаемые на Shape2. Поэтому я сначала выполняю перечисление последовательности `IEnumerable<Shape1>`, которая возвращается первым вызовом метода `GetResult<T>`, а затем — перечисление последовательности `IEnumerable<Shape2>`, возвращенной вторым вызовом метода `GetResult<T>`. Важно то, что я знаю, что записи Shape1 возвращаются первыми, а за ними — записи Shape2, и потому извлекаю их методом `GetResult<T>` именно в таком порядке.

В главе 16 мы рассмотрим метод `ExecuteMethodCall` вместе с примером доступа к выходным параметрам хранимой процедуры, когда она возвращает их в нескольких формах.

Резюме

Эта глава содержала углубленное рассмотрение сущностных классов LINQ to SQL, сложностей, сопровождающих их самостоятельное написание, а также их атрибутов и свойств этих атрибутов.

Важно помнить, что если вы пишете ваши собственные сущностные классы, то отвечаете за реализацию уведомлений об изменении и обеспечение согласованности графа. Эти детали нетривиальны и могут оказаться сложными в реализации. К счастью, как я уже указывал в этой главе, и SQLMetal, и Object Relational Designer благополучно справляются со всеми этими сложностями.

Для написания собственных сущностных классов вы также должны иметь четкое представление об их атрибутах и свойствах этих атрибутов. В этой главе я описал каждое из них и предоставил соответствующую реализацию в концентрированном виде, обсуждая генерированные SQLMetal сущностные классы для базы данных Northwind.

Также я описал преимущества проецирования ваших результатов запроса на сущностные классы по сравнению с несущностными. Если у вас нет необходимости в модификации данных и сохранении изменений, то несущностные классы обычно подходят. Но если вы хотите иметь возможность изменять возвращенные данные и сохранять их обратно в базе данных, то у вас одна дорога — проецирование результата на сущностные классы.

И, наконец, мы обсудили некоторые часто используемые классы из пространства имен `System.Data.Linq`, а также их использование в LINQ to SQL.

К этому моменту вы уже должны быть экспертом в области сущностных классов. Мы достаточно глубоко обсудили их и объяснили устройство генерированного кода. Конечно, к этим сущностным классам обычно обращается класс-наследник `DataContext`, который нам еще предстоит рассмотреть детально. Этим мы и займемся в следующей главе.

ГЛАВА 16

DataContext

В большинстве предыдущих глав, посвященных LINQ to SQL, я ссылался на класс `DataContext`, но пока еще не объяснял его в полной мере. Теперь я устранил это упущение.

В этой главе я объясню устройство класса `DataContext`, что он может делать для вас, и как он это делает. Мы обсудим основные его методы и рассмотрим примеры вызова каждого. Понимание класса `DataContext` необходимо для успешного применения LINQ to SQL, и к моменту окончания чтения этой главы вы должны стать мастером в классе `DataContext`.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры настоящей главы, вам понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите раздел “Предварительные условия для запуска примеров” главы 12 и выполните то, что там сказано.

Некоторые общие методы

В добавок для запуска примеров этой главы вам понадобятся некоторые общие методы, которые будут использованы в примерах. Прочтите раздел “Некоторые общие методы” главы 12 и выполните то, что там сказано.

Использование LINQ to SQL API

Чтобы запускать примеры этой главы, вам может понадобиться добавить соответствующие ссылки и директивы `using` к проекту. Прочтите раздел “Использование LINQ to SQL API” главы 12 и выполните то, что там сказано.

Дополнительно для некоторых примеров настоящей главы вам также понадобится добавить директиву `using` для пространства имен `System.Data.Linq.Mapping`:

```
using System.Data.Linq.Mapping;
```

Класс [Your]DataContext

Хотя я еще его не описывал, но один из классов LINQ to SQL, который вы будете часто использовать — `System.Data.Linq.DataContext`. Это класс, который вы будете применять для установления вашего соединения с базой данных.

При создании или генерации сущностных классов принято создавать класс-наследник `DataContext`. Это производный класс вы обычно будете называть по имени

базы данных, к которой он будет подключаться. Поскольку я использую базу данных Northwind в примерах этой главы, мой производный класс базы данных будет называться Northwind. Однако, поскольку имя производного класса изменяется в зависимости от используемой базы данных, я буду часто ссылаться на него по имени [Your]DataContext. Это — намек на то, что я говорю о созданном вами или сгенерированном классе — наследнике DataContext.

Класс DataContext

Класс DataContext обрабатывает ваше подключение к базе данных. Он также обрабатывает запросы, обновления, вставки в базу данных, отслеживает идентичность, отслеживает изменения, обрабатывает их, обеспечивает целостность транзакций и даже создание базы данных.

Класс DataContext транслирует ваши запросы сущностных классов в операторы SQL, которые выполняются на подключенной базе данных.

Производный от DataContext класс [Your]DataContext предоставляет доступ к целой группе методов базы данных, таких как ExecuteQuery, ExecuteCommand и SubmitChanges. В дополнение к этим унаследованным методам класс [Your]DataContext будет содержать свойства типа System.Data.Linq.Table<T> для каждой таблицы и представления в базе данных, с которой вы намерены использовать LINQ to SQL, где каждый тип T — это сущностный класс, отображенный на конкретную таблицу или представление.

Например, взглянем на класс Northwind, который был сгенерирован для меня инструментом SQLMetal. Это — класс [Your]DataContext для базы данных Northwind. Приведу примечательную часть этого класса, выделив важные части полужирным.

Часть сгенерированного класса Northwind

```
public partial class Northwind : System.Data.Linq.DataContext
{
    ...
    static Northwind()
    {
    }
    public Northwind(string connection) :
        base(connection, mappingSource)
    {
        OnCreated();
    }
    public Northwind(System.Data.IDbConnection connection) :
        base(connection, mappingSource)
    {
        OnCreated();
    }
    public Northwind(string connection,
        System.Data.Linq.Mapping.MappingSource mappingSource) :
        base(connection, mappingSource)
    {
        OnCreated();
    }
    public Northwind(System.Data.IDbConnection connection,
        System.Data.Linq.Mapping.MappingSource mappingSource) :
        base(connection, mappingSource)
    {
        OnCreated();
    }
}
```

```

...
public System.Data.Linq.Table<Customer> Customers
{
    get
    {
        return this.GetTable<Customer>();
    }
}
...
}

```

Как видите, этот класс действительно наследуется от `DataContext`. Вы можете также видеть, что в нем объявлено пять конструкторов. Обратите внимание, что конструктор по умолчанию является приватным (`private`), поскольку для него не указан модификатор доступа, так что вы не сможете создать экземпляр `[Your]DataContext` без параметров. Каждый из открытых конструкторов корелирует с одним из конструкторов `DataContext`. Каждый конструктор `[Your]DataContext` вызывает эквивалентный конструктор класса `DataContext` в своем инициализаторе, а в теле конструктора вызывает только частичный метод `OnCreated`. Это позволяет разработчику-пользователю этого класса реализовывать частичный метод `OnCreated`, который будет вызван при каждом создании объекта `[Your]DataContext`.

Кроме того, в классе `Northwind` есть свойство по имени `Customers` типа `Table<Customer>`, где тип `Customer` представляет существенный класс. Этот существенный класс `Customer` отображается на таблицу `Customers` базы данных `Northwind`.

На самом деле не обязательно писать код, использующий класс `[Your]DataContext`. Вполне можно работать и со стандартным классом `DataContext`. Однако применение класса `[Your]DataContext` делает написание остального кода более удобным. Например, если вы используете класс `[Your]DataContext`, то каждая таблица представлена свойством, которое доступно непосредственно через объект `[Your]DataContext`. Пример показан в листинге 16.1.

Листинг 16.1. Пример, демонстрирующий доступ к таблице через свойство

```

Northwind db =
    new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> query = from cust in db.Customers
                                where cust.Country == "USA"
                                select cust;
foreach(Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}

```

На заметку! Вам может понадобиться подкорректировать строки соединения в примерах этой главы, чтобы заставить их работать.

В предыдущем коде, поскольку я выполняю подключение с использованием класса `[Your]DataContext` по имени `Northwind`, у меня есть возможность обращаться к таблице заказчиков `Table<Customer>` как к свойству `Customers` класса `[Your]DataContext`. Вот результат работы этого кода:

```

Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop

```

```
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Если бы вместо этого я подключался с применением самого класса DataContext, то должен был бы использовать метод GetTable<T> объекта DataContext, как показано в листинге 16.2.

Листинг 16.2. Пример, демонстрирующий доступ к таблице через метод GetTable<T>

```
DataContext dc =
    new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> query = from cust in dc.GetTable<Customer>()
                                where cust.Country == "USA"
                                select cust;
foreach(Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Однако этот код выдаст тот же результат:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Так что использование класса [Your]DataContext просто удобнее, и это стоит взять на вооружение.

Главные цели

В дополнение ко всем методам, о которых я расскажу в этой главе, класс DataContext предоставляет три основных службы: отслеживание идентичности, отслеживание изменений и их обработку.

Отслеживание идентичности

Одной из проблем, для преодоления которых предназначен LINQ to SQL, является проблема *объектно-реляционной потери соответствия* (object-relational impedance mismatch). Этот термин указывает на неизбежные трудности, вызванные тем фактом, что большинство наиболее распространенных баз данных являются реляционными, в то время как большинство современных языков программирования — объектно-ориентированными. Из-за этого различия и возникают проблемы.

Одним из проявлений объектно-реляционной потери соответствия является ожидаемый нами способ поведения идентичности. Если мы запрашиваем одну и ту же запись из базы данных во многих местах кода, то ожидаем, что возвращенные данные будут находиться в разных местах в памяти. Мы ожидаем, что модификация полей записей в одной части кода не затронет поля той же записи, извлекаемые в другой части кода. Мы ожидаем этого потому, что извлеченные данные хранятся в разных переменных, находящихся по разным адресам в памяти.

Поведение, ожидаемое нами от объектов, отличается от этого. Мы ожидаем, что когда у нас есть объект в памяти, скажем, объект `Customer`, то можно рассчитывать, что во всех местах кода, имеющий ссылку на одного и того же заказчика, будет ссылаться на одно и то же местоположение в памяти. Если мы обновим свойство — имя объекта заказчика в одном месте нашей программы, то можно ожидать, что заказчик, ссылка на которого у нас имеется, в другой части кода будет иметь новое имя.

Служба отслеживания идентичности `DataContext` — вот что обеспечивает нам такое поведение. Когда запись запрашивается из базы данных первый раз с момента создания экземпляра объекта `DataContext`, эта запись сохраняется в таблице идентичности с использованием ее первичного ключа, создается объект идентичности и сохраняется в кэше. Последующие запросы, определяющие, что некоторая запись должна быть возвращена, сначала проверяют таблицу идентичности, и если запись в этой таблице существует, то из кэша возвращается уже существующий объект. Эта важная концепция, которую нужно понимать, поэтому я еще раз пройдусь по ней, немного иным путем. Когда выполняется запрос, если запись в базе данных отвечает критерию поиска и если ее сущностный объект уже кэширован, то возвращается этот кэшированный объект. Это значит, что действительные данные, возвращенные запросом, могут быть не теми же самыми, что содержатся в записи базы данных. Но служба слежения за идентичностью объекта `DataContext` определит, какие данные возвращаются. Это может привести к проблеме, которую я называю *несоответствием кэша результирующего набора*.

Несоответствие кэша результирующего набора

Работая над некоторыми примерами для этой книги, я заметил достаточно странное поведение. Разработчик из Microsoft уверял меня, что это поведение совершенно корректно и сделано намеренно.

Мне пока еще не встречалось какое-то название для такого поведения, поэтому я буду называть его “несоответствием кэша результирующего набора”. Поскольку я твердый сторонник того, чтобы сначала попробовать, а потом покупать, поэтому вы можете использовать эту фразу 30 раз, но потом — пожалуйста, пришлите мне чек с гонораром.

Несоответствие кэша результирующего набора может случиться, когда запись в базе данных несогласована с сущностным объектом, представляющим ее в кэше вашего объекта `DataContext`. Когда вы выполняете запрос, действительная база данных опрашивается на предмет поиска записи, отвечающей запросу. Если запись в базе соответствует критерию поиска, то сущностный объект этой записи будет включен в возвращенный результирующий набор. Однако если сущностный объект записи из результирующего набора уже кэширован в объекте `DataContext`, то кэшированный сущностный объект будет возвращен запросом вместо того, чтобы прочесть последнюю версию записи из базы данных.

В результате вы имеете сущностный объект, кэшированный в `DataContext`, а другой контекст обновляет поле записи этого сущностного объекта в базе данных, и вы выполняете запрос LINQ, специфицируя это поле в критерии поиска, так что он соответствует новому значению из базы данных, и такая запись будет включена в результирующий набор. Однако поскольку она уже есть в кэше, вы получите кэшированный сущностный объект с полем, которое не отвечает нашему критерию поиска.

Возможно, вам станет понятнее, если я приведу конкретный пример. Сначала я запросу определенного заказчика, который, как я знаю, не соответствует критерию поиска, указанном в последующем запросе. Я использую заказчика LONEP. Регион этого заказчика — OR, поэтому я буду искать заказчиков из региона WA. Затем я отобразжу этих заказчиков из региона WA. После этого обновлю регион заказчика LONEP, заменив его WA, используя для этого ADO.NET — как если бы это сделал некоторый другой контекст, внешний по отношению к моему процессу. В этой точке LONEP будет иметь регион OR в моем сущностном объекте, но WA — в базе данных. Далее я выполню тот же запрос снова, чтобы извлечь всех заказчиков из региона WA. Если вы взглянете на код, то не увидите там нового определения запроса. Вы просто увидите, что я выполняю перечисление на возвращенной последовательности custs. Помните, что из-за отложенного выполнения запросов мне нужно лишь перечислять результаты, чтобы снова запустить запрос. Поскольку в базе данных регион заказчика LONEP теперь WA, эта запись будет включена в результирующий набор. Но поскольку сущностный объект этой записи уже находится в кэше, будет возвращен именно этот кэшированный сущностный объект, а у этого объекта значение региона — по-прежнему OR. Затем я отобразжу регион каждого возвращенного сущностного объекта. Когда очередь дойдет до заказчика LONEP, его регионом окажется OR, несмотря на тот факт, что в запросе указана необходимость в заказчиках из региона WA. Листинг 16.3 представляет код, демонстрирующий это несоответствие.

Листинг 16.3. Пример, демонстрирующий несоответствие кэша результирующего набора

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Получим заказчика для модификации, который находится вне запрашиваемого
// региона == 'WA'.
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
Console.WriteLine("Customer {0} has region = {1}. {2}",
                  cust.CustomerID, cust.Region, System.Environment.NewLine);
// Порядок, регионом LONEP является OR.
// Теперь получим последовательность заказчиков из 'WA', которая
// не включает LONEP, поскольку его регион — OR.
IEnumerable<Customer> custs = (from c in db.Customers
                                   where c.Region == "WA"
                                   select c);
Console.WriteLine("Customers from WA before ADO.NET change - start ...");
foreach(Customer c in custs)
{
    // Отобразить Region каждого сущностного объекта.
    Console.WriteLine("Customer {0}'s region is {1}.", c.CustomerID, c.Region);
}
Console.WriteLine("Customers from WA before ADO.NET change - end. {0}",
                  System.Environment.NewLine);
// Теперь я изменю регион заказчика LONEP на WA, что приведет
// к включению его в результаты предыдущего запроса.
// Изменим регион заказчика через ADO.NET.
Console.WriteLine("Updating LONEP's region to WA in ADO.NET...");
ExecuteStatementInDb(
    "update Customers set Region = 'WA' where CustomerID = 'LONEP'");
Console.WriteLine("LONEP's region updated. {0}", System.Environment.NewLine);
Console.WriteLine("So LONEP's region is WA in database, but ...");
Console.WriteLine("Customer {0} has region = {1} in entity object. {2}",
                  cust.CustomerID, cust.Region, System.Environment.NewLine);
// Теперь в базе данных регион LONEP — WA, но в сущностном объекте — по-прежнему WA.
// Теперь снова выполним запрос.

```

474 Часть V. LINQ to SQL

```
// И снова отобразим регион сущностного объекта заказчика.
Console.WriteLine("Query entity objects after ADO.NET change - start ...");
foreach(Customer c in custs)
{
    // Отобразим Region каждого сущностного объекта.
    Console.WriteLine("Customer {0}'s region is {1}.", c.CustomerID, c.Region);
}
Console.WriteLine("Query entity objects after ADO.NET change - end.{0}",
    System.Environment.NewLine);
// Измененное значение вернем в исходное состояние, чтобы
// данный код можно было запустить более одного раза.
Console.WriteLine("{0}Resetting data to original values.", 
    System.Environment.NewLine);
ExecuteStatementInDb(
    "update Customers set Region = 'OR' where CustomerID = 'LONEP'");
```

И вот результат:

```
Customer LONEP has region = OR.
Customers from WA before ADO.NET change - start ...
Customer LAZYK's region is WA.
Customer TRAIH's region is WA.
Customer WHITC's region is WA.
Customers from WA before ADO.NET change - end.
Updating LONEP's region to WA in ADO.NET...
Executing SQL statement against database with ADO.NET ...
Database updated.
LONEP's region updated.
So LONEP's region is WA in database, but ...
Customer LONEP has region = OR in entity object.
Query entity objects after ADO.NET change - start ...
Customer LAZYK's region is WA.
Customer LONEP's region is OR.
Customer TRAIH's region is WA.
Customer WHITC's region is WA.
Query entity objects after ADO.NET change - end.
```

Как видите, даже несмотря на то, что я запросил заказчиков из региона WA, заказчик LONEP был включен в результат, при том, что его регион — OR. Конечно, это верно, что в базе данных у LONEP регион указан WA, но у объекта, на который я имею ссылку в моем коде, это не так. Еще у кого-нибудь есть ощущение тошноты?

Другим проявлением этого поведения является тот факт, что вставленные сущности не могут быть запрошены, а удаленные — могут, если это происходит до вызова SubmitChanges. Опять-таки, это объясняется тем, что хотя сущность и вставлена, при выполнении запроса результирующий набор определяется действительным содержимым базы данных, а не кэшем объекта DataContext. Поскольку изменения не были зафиксированы, вставленной сущности в базе данных еще нет. С удаленными сущностями — обратная картина. В листинге 16.4 представлен пример, демонстрирующий это поведение.

Листинг 16.4. Другой пример, демонстрирующий несоответствие кэша результирующего набора

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("First I will add customer LAWN.");
db.Customers.InsertOnSubmit(
    new Customer
    {
        CustomerID = "LAWN",
```

```

CompanyName = "Lawn Wranglers",
ContactName = "Mr. Abe Henry",
ContactTitle = "Owner",
Address = "1017 Maple Leaf Way",
City = "Ft. Worth",
Region = "TX",
PostalCode = "76104",
Country = "USA",
Phone = "(800) MOW-LAWN",
Fax = "(800) MOW-LAWO"
});
Console.WriteLine("Next I will query for customer LAWN.");
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LAWN"
                  select c).SingleOrDefault<Customer>();
Console.WriteLine("Customer LAWN {0}.{1}",
                  cust == null ? "does not exist" : "exists",
                  System.Environment.NewLine);
Console.WriteLine("Now I will delete customer LONEP");
cust = (from c in db.Customers
        where c.CustomerID == "LONEP"
        select c).SingleOrDefault<Customer>();
db.Customers.DeleteOnSubmit(cust);
Console.WriteLine("Next I will query for customer LONEP.");
cust = (from c in db.Customers
        where c.CustomerID == "LONEP"
        select c).SingleOrDefault<Customer>();
Console.WriteLine("Customer LONEP {0}.{1}",
                  cust == null ? "does not exist" : "exists",
                  System.Environment.NewLine);
// Незачем возвращать базу в исходное состояние, поскольку не был вызван SubmitChanges()

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод InsertOnSubmit, вызванный в предыдущем коде, назывался Add, а метод DeleteOnSubmit — Remove.

В предыдущем коде я вставляю заказчика LAWN, затем запрашиваю его, чтобы проверить его существование. Затем удаляю другого заказчика — LONEP — и выполняю запрос, чтобы проверить его наличие. Все это я делаю без вызова метода SubmitChanges, так что кэшированные сущностные объекты не сохраняются в базе данных. Вот результат запуска этого кода:

```

First I will add customer LAWN.
Next I will query for customer LAWN.
Customer LAWN does not exist.
Now I will delete customer LONEP
Next I will query for customer LONEP.
Customer LONEP exists.

```

Тот разработчик из Microsoft, что говорил мне, что это поведение так и было задумано, утверждал, что данные, извлеченные запросом, фиксируют момент, когда вы извлекли их, и что данные, кэшированные в DataContext, не предназначены для долговременного кэширования. Если вам нужна лучшая изоляция и согласованность, рекомендовал он, поместите все в транзакцию. Примеры того, как это делается, вы найдете в разделе "Пессимистический параллелизм" главы 17.

Отслеживание изменений

Как только служба идентичности создает сущностный объект в кэше, служба отслеживания изменений начинает наблюдать за этим объектом. Отслеживание изменений работает, сохраняя первоначальные значения сущностного объекта. Эта служба продолжает свою работу до тех пор, пока вы не вызовете метод `SubmitChanges`. Вызов этого метода сохраняет изменения сущностного объекта в базе данных, первоначальные значения забываются, а измененные значения становятся первоначальными. Это позволяет снова приступать к отслеживанию изменений.

Все это работает хорошо до тех пор, пока сущностные объекты извлекаются из базы данных. Однако простое создание нового экземпляра сущностного объекта не вызывает никакого отслеживания идентичности или изменений до тех пор, пока `DataContext` не узнает об его существовании. Чтобы сообщить объекту `DataContext` о существовании сущностного объекта, просто вставьте сущностный объект в одно из свойств `Table<T>`. Например, в моем классе `Northwind` у меня есть свойство типа `Table<Customer>` по имени `Customers`. Я могу вызвать метод `InsertOnSubmit` на свойстве `Customers`, чтобы вставить сущностный объект `Customer` в `Table<Customer>`. Когда это сделано, `DataContext` начнет отслеживать идентичность и изменения этого сущностного объекта. Ниже приведен пример кода, вставляющего нового заказчика.

```
db.Customers.InsertOnSubmit(
    new Customer {
        CustomerID = "LAWN",
        CompanyName = "Lawn Wranglers",
        ContactName = "Mr. Abe Henry",
        ContactTitle = "Owner",
        Address = "1017 Maple Leaf Way",
        City = "Ft. Worth",
        Region = "TX",
        PostalCode = "76104",
        Country = "USA",
        Phone = "(800) MOW-LAWN",
        Fax = "(800) MOW-LAWO"}) ;
```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод `InsertOnSubmit`, вызванный в предыдущем коде, назывался `Add`.

Как только я вызываю метод `InsertOnSubmit`, начинается отслеживание идентичности и изменений заказчика `LAWN`.

Сначала служба отслеживания изменений показалась мне несколько запутанной. Понять базовую концепцию достаточно просто, но уверенность в обращении с ней приходит не сразу. Понимание отслеживания изменений становится еще более важным, если вы пишете ваши сущностные классы вручную. Прочтите еще раз раздел, озаглавленный “Уведомления об изменениях”, в главе 15, чтобы получить более полное представление о работе службы слежения за изменениями.

Обработка изменений

Одним из наиболее важных служб, предоставленных классом `DataContext`, является отслеживание изменений в сущностных объектах. Когда вы вставляете, изменяете или удаляете сущностный объект, то `DataContext` выполняет мониторинг того, что происходит. Однако никакие изменения в действительности еще не распространяются на базу данных. Изменения кэшируются в `DataContext` до тех пор, пока вы не вызовете метод `SubmitChanges`.

Когда вы вызываете метод `SubmitChanges`, процессор изменений объекта `DataContext` управляет обновлением базы данных. Сначала процессор изменений вставит все вновь вставленные сущностные объекты в свой список отслеживаемых объектов. Затем он упорядочит все измененные сущностные объекты на основе их зависимостей, обусловленных внешними ключами и ограничениями уникальности. Затем, если в области определения нет никаких транзакций, он создаст транзакцию, так что все команды SQL, выполняемые во время вызова метода `SubmitChanges`, будут обладать транзакционной целостностью. Он использует уровень изоляции SQL Server по умолчанию — `ReadCommitted`, что означает, что прочитанные данные не будут физически повреждены и только зафиксированные (`committed`) данные будут прочитаны, но поскольку блокировка является разделяемой (`shared`), ничто не помешает данным измениться до окончания транзакции. И, наконец, он выполняет перечисление упорядоченного списка измененных сущностных объектов, создает необходимые операторы SQL и выполняет их.

Если в процессе перечисления измененных сущностных объектов происходят любые ошибки, если метод `SubmitChanges` использует `ConflictMode` из `FailOnFirstConflict`, то процесс перечисления прерывается, транзакция откатывается, отменяя все изменения в базе данных, и генерируется исключение. Если специфицирован `ConflictMode` из `ContinueOnConflict`, все измененные сущностные объекты будут перечислены и обработаны, несмотря на любые возникающие конфликты, пока `DataContext` строит список конфликтов. Но, опять-таки, транзакция откатывается, отменяя все изменения в базе данных, и генерируется исключение. Однако пока данные не сохранены в базе данных, все изменения сущностных объектов присутствуют в этих объектах. Это дает возможность разработчику попытаться разрешить проблему и снова вызвать метод `SubmitChanges`.

Если все изменения проведены в базе данных успешно, транзакция фиксируется, и информация об изменениях сущностных объектов удаляется, так что отслеживание изменений может начать работать заново.

`DataContext()` и `[Your]DataContext()`

Класс `DataContext` обычно наследуется для создания класса `[Your]DataContext`. Он существует для подключения к базе данных и обработки всего взаимодействия с базой данных. Вы будете использовать один из следующих конструкторов для создания объекта `DataContext` или `[Your]DataContext`.

Прототипы

Конструктор `DataContext` имеет четыре прототипа, описанных ниже.

Первый прототип конструктора `DataContext`

```
DataContext(string fileOrServerOrConnectionString);
```

Этот прототип конструктора принимает строку соединения ADO.NET и, возможно, вы будете использовать его большую часть времени. Этот прототип используется также в большинстве примеров LINQ to SQL этой книги.

Второй прототип конструктора `DataContext`

```
DataContext(System.Data.IDbConnection connection);
```

Поскольку `System.Data.SqlClient.SqlConnection` наследуется от `System.Data.Common.DbConnection`, который реализует `System.Data.IDbConnection`, вы можете создать экземпляр `DataContext` или `[Your]DataContext` с только что созданным `SqlConnection`. Этот прототип конструктора удобен, когда приходится смешивать код LINQ to SQL с уже существующим кодом ADO.NET.

Третий прототип конструктора `DataContext`

```
DataContext(string fileOrServerOrConnection,
           System.Data.Linq.MappingSource mapping);
```

Этот прототип конструктора удобен, когда у вас нет класса `[Your]DataContext`, а вместо него XML-файл отображения. Иногда у вас уже может быть существующий бизнес-класс, к которому вы не можете добавить соответствующие атрибуты LINQ to SQL. Возможно, у вас даже нет его исходного кода. Вы можете сгенерировать файл отображения утилитой `SQLMetal`, либо написать вручную, чтобы он работал с уже существующим бизнес-классом или любым другим классом того же рода. Вы представляете нормальную строку соединения ADO.NET для установки соединения.

Четвертый прототип конструктора `DataContext`

```
DataContext (System.Data.IDbConnection connection,
            System.Data.Linq.MappingSource mapping)
```

Этот прототип позволит создать соединение LINQ to SQL из уже существующего соединения ADO.NET и предоставить XML-файл отображения. Эта версия прототипа удобна в тех случаях, когда вы комбинируете код LINQ to SQL с существующим кодом ADO.NET, и у вас нет сущностных классов, оснащенных атрибутами.

Примеры

Для примера первого прототипа конструктора `DataContext` в листинге 16.5 я подключусь к физическому файлу `.mdf`, используя строку соединения типа ADO.NET.

Листинг 16.5. Первый прототип конструктора `DataContext`, подключаемого к файлу базы данных

```
DataContext dc = new DataContext(@"C:\Northwind.mdf");
IQueryable<Customer> query = from cust in dc.GetTable<Customer>()
                                where cust.Country == "USA"
                                select cust;
foreach (Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

На заметку! Вам придется модифицировать путь, переданный конструктору `DataContext`, чтобы он мог найти ваш файл `.mdf`.

Я просто предоставляю путь к файлу `.mdf`, чтобы создать экземпляр объекта `DataContext`. Поскольку я создаю объект `DataContext`, а не `[Your]DataContext`, я должен вызывать метод `GetTable<T>` для доступа к заказчикам в базе данных. Вот результат:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Затем я хочу продемонстрировать тот же базовый код, но на этот раз, в листинге 16.6 я использую мой класс [Your]DataContext, которым в данном случае является Northwind.

Листинг 16.6. Первый прототип конструктора [Your]DataContext, подключаемого к файлу базы данных

```
DataContext db = new Northwind(@"C:\Northwind.mdf");
IQueryable<Customer> query = from cust in db.Customers
                                where cust.Country == "USA"
                                select cust;
foreach (Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Обратите внимание, что вместо вызова метода GetTable<T> я просто ссылаюсь на свойство Customers для доступа к заказчикам в базе данных. Неудивительно, что этот код выдает тот же результат:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

В целях полноты картины я представлю еще один пример первого прототипа, но на этот раз использую строку соединения для действительного подключения к серверу базы данных SQL Server, содержащему присоединенную базу Northwind. И, поскольку моя обычная практика заключается в применении класса [Your]DataContext, я воспользуюсь ею и в листинге 16.7.

Листинг 16.7. Первый прототип конструктора [Your]DataContext, подключаемого к базе данных

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind");
IQueryable<Customer> query = from cust in db.Customers
                                where cust.Country == "USA"
                                select cust;
foreach (Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Результат все тот же:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
```

```

Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets

```

Поскольку второй прототип для класса `DataContext` удобен при комбинировании кода LINQ to SQL с кодом ADO.NET, этому будет посвящен мой следующий пример в листинге 16.8. Первым делом, я создам `SqlConnection` и через него вставлю запись в таблицу `Customers`. Затем воспользуюсь `SqlConnection` для создания экземпляра класса `[Your]DataContext`. И, наконец, используя ADO.NET, я удалю вставленную мною запись из таблицы `Customers`, на этот раз применяя LINQ to SQL, и отобразжу результаты.

Листинг 16.8. Второй прототип конструктора `[Your]DataContext`, подключаемого с помощью разделяемого соединения ADO.NET

```

System.Data.SqlClient.SqlConnection sqlConn =
    new System.Data.SqlClient.SqlConnection(
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
string cmd = @"insert into Customers values ('LAWN', 'Lawn Wranglers',
    'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
    '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')";
System.Data.SqlClient.SqlCommand sqlComm =
    new System.Data.SqlClient.SqlCommand(cmd);
sqlComm.Connection = sqlConn;
try
{
    sqlConn.Open();
    // Вставить запись.
    sqlComm.ExecuteNonQuery();
    Northwind db = new Northwind(sqlConn);
    IQueryables<Customer> query = from cust in db.Customers
        where cust.Country == "USA"
        select cust;
    Console.WriteLine("Customers after insertion, but before deletion.");
    foreach (Customer c in query)
    {
        Console.WriteLine("{0}", c.CompanyName);
    }
    sqlComm.CommandText = "delete from Customers where CustomerID = 'LAWN'";
    // Удалить запись.
    sqlComm.ExecuteNonQuery();
    Console.WriteLine("{0}{0}Customers after deletion.", System.Environment.NewLine);
    foreach (Customer c in query)
    {
        Console.WriteLine("{0}", c.CompanyName);
    }
}
finally
{
    // Закрыть соединение.
    sqlComm.Connection.Close();
}

```

Обратите внимание, что я определяю запрос LINQ лишь однажды, но заставляю выполниться дважды — два раза выполняя перечисление возвращенной последовательности. Напомню, что из-за отложенного выполнения запроса определение запроса LINQ в действительности не запускает его выполнения. Запрос выполняется только при перечислении результатов. Это доказывает тот факт, что результаты отличаются между двумя перечислениями. Листинг 16.8 также демонстрирует замечательную интеграцию ADO.NET и LINQ to SQL и то, насколько хорошо они работают вместе. Вот результат:

```
Customers after insertion, but before deletion.
Great Lakes Food Market
Hungry Coyote Import Store
Lawn Wranglers
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
Customers after deletion.
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Для примера третьего прототипа я даже не стану использовать существенный класс Northwind. Представьте, что у меня его даже нету. Вместо этого я применяю класс Customer, написанный мною вручную, и сокращенный файл отображения. По правде говоря, мой вручную написанный класс Customer — это сгенерированный SQLMetal класс Customer, который я несколько сократил, удалив из него атрибуты LINQ to SQL. Взглянем на мой вручную написанный класс Customer.

Мой написанный вручную класс Customer

```
namespace Linqdev
{
    public partial class Customer
    {
        private string _CustomerID;
        private string _CompanyName;
        private string _ContactName;
        private string _ContactTitle;
        private string _Address;
        private string _City;
        private string _Region;
        private string _PostalCode;
```

```
private string _Country;
private string _Phone;
private string _Fax;
public Customer()
{
}
public string CustomerID
{
    get
    {
        return this._CustomerID;
    }
    set
    {
        if ((this._CustomerID != value))
        {
            this._CustomerID = value;
        }
    }
}
public string CompanyName
{
    get
    {
        return this._CompanyName;
    }
    set
    {
        if ((this._CompanyName != value))
        {
            this._CompanyName = value;
        }
    }
}
public string ContactName
{
    get
    {
        return this._ContactName;
    }
    set
    {
        if ((this._ContactName != value))
        {
            this._ContactName = value;
        }
    }
}
public string ContactTitle
{
    get
    {
        return this._ContactTitle;
    }
    set
    {
        if ((this._ContactTitle != value))
        {
            this._ContactTitle = value;
        }
    }
}
```

```
public string Address
{
    get
    {
        return this._Address;
    }
    set
    {
        if ((this._Address != value))
        {
            this._Address = value;
        }
    }
}
public string City
{
    get
    {
        return this._City;
    }
    set
    {
        if ((this._City != value))
        {
            this._City = value;
        }
    }
}
public string Region
{
    get
    {
        return this._Region;
    }
    set
    {
        if ((this._Region != value))
        {
            this._Region = value;
        }
    }
}
public string PostalCode
{
    get
    {
        return this._PostalCode;
    }
    set
    {
        if ((this._PostalCode != value))
        {
            this._PostalCode = value;
        }
    }
}
public string Country
{
```

```
        get
        {
            return this._Country;
        }
        set
        {
            if ((this._Country != value))
            {
                this._Country = value;
            }
        }
    }
    public string Phone
    {
        get
        {
            return this._Phone;
        }
        set
        {
            if ((this._Phone != value))
            {
                this._Phone = value;
            }
        }
    }
    public string Fax
    {
        get
        {
            return this._Fax;
        }
        set
        {
            if ((this._Fax != value))
            {
                this._Fax = value;
            }
        }
    }
}
```

Наверное, это наихудший написанный вручную сущностный класс за все времена. Я не обрабатываю уведомления об изменениях, и я удалил значительную часть кода, который делал этот сущностный класс полноценным. Прочтите главу 15, чтобы узнать, как правильно писать полноценные сущностные классы.

Обратите внимание, что я указал, что этот класс живет в пространстве имен Linqdev. Это важно, потому что мне понадобится не только указать это в моем коде примера, чтобы отличать этот класс Customer от того, который имеется в пространстве nwind, но это же пространство имен должно быть специфицировано во внешнем файле отображения.

Однако что важнее всего в этом примере, так это наличие свойства для каждого поля базы данных, отображенного во внешнем файле. Теперь давайте взглянем на этот внешний файл отображения, который я буду использовать для этого примера.

Сокращенный внешний XML-файл отображения

```

<?xml version="1.0" encoding="utf-8"?>
<Database Name="Northwind"
  xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
  <Table Name="dbo.Customers" Member="Customers">
    <Type Name="Linqdev.Customer">
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID"
        DbType="NChar(5) NOT NULL CanBeNull=false IsPrimaryKey=true />
      <Column Name="CompanyName" Member="CompanyName" Storage="_CompanyName"
        DbType="NVarChar(40) NOT NULL CanBeNull=false />
      <Column Name="ContactName" Member="ContactName" Storage="_ContactName"
        DbType="NVarChar(30)" />
      <Column Name="ContactTitle" Member="ContactTitle" Storage="_ContactTitle"
        DbType="NVarChar(30)" />
      <Column Name="Address" Member="Address" Storage="_Address"
        DbType="NVarChar(60)" />
      <Column Name="City" Member="City" Storage="_City" DbType="NVarChar(15)" />
      <Column Name="Region" Member="Region" Storage="_Region"
        DbType="NVarChar(15)" />
      <Column Name="PostalCode" Member="PostalCode" Storage="_PostalCode"
        DbType="NVarChar(10)" />
      <Column Name="Country" Member="Country" Storage="_Country"
        DbType="NVarChar(15)" />
      <Column Name="Phone" Member="Phone" Storage="_Phone" DbType="NVarChar(24)" />
      <Column Name="Fax" Member="Fax" Storage="_Fax" DbType="NVarChar(24)" />
    </Type>
  </Table>
</Database>
```

Обратите внимание, что я специфицировал, что класс Customer, на который выполняются отображения, находится в пространстве имен Linqdev.

Я поместил этот код XML в файл под названием abbreviatednorthwindmap.xml, а этот файл — в мой каталог bin\Debug.

В листинге 16.9 я использую этот вручную написанный класс Customer вместе с внешним файлом отображения для выполнения запроса LINQ to SQL без использования атрибутов.

Листинг 16.9. Третий прототип конструктора DataContext, подключаемого к базе данных и использующего файл отображения

```

string mapPath = "abbreviatednorthwindmap.xml";
XmlMappingSource nwindMap =
  XmlMappingSource.FromXml(System.IO.File.ReadAllText(mapPath));
DataContext db = new DataContext(
  @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;",
  nwindMap);
IQueryable<Linqdev.Customer> query =
  from cust in db.GetTable<Linqdev.Customer>()
  where cust.Country == "USA"
  select cust;
foreach (Linqdev.Customer c in query)
{
  Console.WriteLine("{0}", c.CompanyName);
}
```

На заметку! Для этого примера я поместил файл abbreviatednorthwindmap.xml в каталог bin\Debug моего проекта Visual Studio, поскольку выполняю компиляцию и запуск с отладочной (Debug) конфигурацией.

Как видите, я создаю экземпляр объекта XmlMappingSource из файла отображения и передаю этот XmlMappingSource конструктору DataContext. Также обратите внимание, что я не могу просто обратиться к свойству Customers Table<Customer> в моем объекте DataContext для запроса LINQ to SQL, потому что использую базовый класс DataContext вместо моего [Your]DataContext, который здесь не существует.

Также заметьте, что везде, где я ссылаюсь на класс Customer, я еще специфицирую явно пространство Linqdev — просто чтобы быть уверенным, что нечаянно не применяю генерированный SQLMetal класс Customer, используемый большинством прочих примеров. Вот результат запуска примера из листинга 16.9:

```
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
```

Хотя в этом примере используется сырой класс Customer, которому недостает большей части кода, необходимого полноценному сущностному классу, я хотел показать вам пример использования файла отображения и класса, лишенного атрибутов LINQ to SQL.

Четвертый прототип — просто комбинация второго и третьего. В листинге 16.10 показан пример его применения.

Листинг 16.10. Четвертый прототип конструктора DataContext, подключаемого к базе данных с помощью разделяемого соединения ADO.NET и использующего файл отображения

```
System.Data.SqlClient.SqlConnection sqlConn =
    new System.Data.SqlClient.SqlConnection(
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
string cmd = @"insert into Customers values ('LAWN', 'Lawn Wranglers',
    'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
    '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')";
System.Data.SqlClient.SqlCommand sqlComm =
    new System.Data.SqlClient.SqlCommand(cmd);
sqlComm.Connection = sqlConn;
try
{
    sqlConn.Open();
    // Вставить запись.
    sqlComm.ExecuteNonQuery();
    string mapPath = "abbreviatednorthwindmap.xml";
    XmlMappingSource nwindMap =
        XmlMappingSource.FromXml(System.IO.File.ReadAllText(mapPath));
    DataContext db = new DataContext(sqlConn, nwindMap);
```

```

IQueryable<Linqdev.Customer> query =
    from cust in db.GetTable<Linqdev.Customer>()
    where cust.Country == "USA"
    select cust;
Console.WriteLine("Customers after insertion, but before deletion.");
foreach (Linqdev.Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}

sqlComm.CommandText = "delete from Customers where CustomerID = 'LAWN'";
// Удалить запись.
sqlComm.ExecuteNonQuery();
Console.WriteLine("{0}{0}Customers after deletion.", System.Environment.NewLine);
foreach (Linqdev.Customer c in query)
{
    Console.WriteLine("{0}", c.CompanyName);
}
}
finally
{
    // Закрыть соединение.
    sqlComm.Connection.Close();
}

```

Код в листинге 16.10 зависит от класса Linqdev.Customer и внешнего файла abbreviatednorthwindmap.xml, как и код в листинге 16.9.

Это замечательный пример использования LINQ to SQL для запроса базы данных без оснащенного атрибутами кода сущностного класса и интегрированного с кодом ADO.NET. Результат именно такой, как следовало ожидать:

```

Customers after insertion, but before deletion.
Great Lakes Food Market
Hungry Coyote Import Store
Lawn Wranglers
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets
Customers after deletion.
Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
Lonesome Pine Restaurant
Old World Delicatessen
Rattlesnake Canyon Grocery
Save-a-lot Markets
Split Rail Beer & Ale
The Big Cheese
The Cracker Box
Trail's Head Gourmet Provisioners
White Clover Markets

```

Как можно видеть из предыдущих примеров, получить подключенный `DataContext` или `[Your]DataContext` совсем не сложно.

SubmitChanges ()

Объект `DataContext` будет кэшировать все изменения, проведенные в сущностных объектах, до тех пор, пока не будет вызван метод `SubmitChanges`. Метод `SubmitChanges` инициирует процессор изменений, и эти изменения в сущностных объектах будут сохранены в базе данных.

Если объекту `DataContext` недоступна объемлющая транзакция, которую можно было бы задействовать при вызове метода `SubmitChanges`, транзакция будет создана, и все изменения будут проведены в ее рамках. В этом случае, если транзакция потерпит неудачу, все ее изменения в базе данных будут отменены.

В случае возникновения конфликтов параллельного доступа будет сгенерировано исключение `ChangeConflictException`, что даст вам возможность попытаться разрешить конфликты и повторить попытку фиксации изменений. И что действительно хорошо — `DataContext` содержит коллекцию `ChangeConflicts`, которая предоставляет метод `ResolveAll`, чтобы выполнить такое разрешение конфликтов за вас. Неплохо?

Конфликты параллельного доступа детально рассматриваются в главе 17.

Прототипы

Метод `SubmitChanges` имеет два прототипа, описанных ниже.

Первый прототип SubmitChanges

```
void SubmitChanges()
```

Этот прототип метода не принимает аргументов и по умолчанию устанавливает значение `ConflictMode` равным `FailOnFirstConflict`.

Второй прототип SubmitChanges

```
void SubmitChanges(ConflictMode failureMode)
```

Этот прототип метода позволяет вам специфицировать `ConflictMode`. Его возможные значения: `ConflictMode.FailOnFirstConflict` и `ConflictMode.ContinueOnConflict`.

Режим `ConflictMode.FailOnFirstConflict` работает в соответствии со своим названием; вынуждая метод `SubmitChanges` генерировать исключение `ChangeConflictException` при самом первом обнаруженному конфликте. Режим `ConflictMode.ContinueOnConflict` пытается провести все обновления базы данных, чтобы, когда сгенерировано исключение `ChangeConflictException`, можно было получить отчет о них всех и разрешить за один прием.

Конфликты оцениваются в терминах количества конфликтующих записей, а не количества конфликтующих полей. Вы можете иметь два поля из одной записи, приведших к конфликту, но это вызовет только один конфликт.

Примеры

Поскольку многие примеры из главы 14 вызывают метод `SubmitChanges`, тривиальный пример этого метода, вероятно, покажется вам старым знакомым. Вместо того чтобы утомлять вас еще одним базовым примером вызова метода `SubmitChanges`, дабы просто сохранить изменения в базе данных, я хочу предложить вам несколько более сложный пример. В примере вызова первого прототипа `SubmitChanges` я хочу сначала доказать вам, что изменения не проводятся в базе данных до тех пор, пока не будет вызван метод `SubmitChanges`. Поскольку этот пример сложнее многих предыдущих, я буду сопровождать его код необходимыми пояснениями. Листинг 16.11 содержит код этого примера.

Листинг 16.11. Пример вызова первого прототипа SubmitChanges

```

System.Data.SqlClient.SqlConnection sqlConn =
    new System.Data.SqlClient.SqlConnection(
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
try
{
    sqlConn.Open();
    string sqlQuery = "select ContactTitle from Customers where CustomerID = 'LAZYK'";
    string originalTitle = GetStringFromDb(sqlConn, sqlQuery);
    string title = originalTitle;
    Console.WriteLine("Title from database record: {0}", title);
    Northwind db = new Northwind(sqlConn);
    Customer c = (from cust in db.Customers
                  where cust.CustomerID == "LAZYK"
                  select cust).
        Single<Customer>();
    Console.WriteLine("Title from entity object : {0}", c.ContactTitle);
}

```

В приведенном коде я создаю соединение с базой данных ADO.NET и открываю его. Затем запрашиваю в базе ContactTitle заказчика LAZYK, используя мой общий метод GetStringFromDb, и отображаю его. Далее создаю объект Northwind, используя соединение с базой данных ADO.NET, запрашиваю того же заказчика, используя LINQ to SQL, и отображаю их ContactTitle. В этой точке два полученных ContactTitle должны совпадать.

```

Console.WriteLine(String.Format(
    "{0}Change the title to 'Director of Marketing' in the entity object:",
    System.Environment.NewLine));
c.ContactTitle = "Director of Marketing";
title = GetStringFromDb(sqlConn, sqlQuery);
Console.WriteLine("Title from database record: {0}", title);

Customer c2 = (from cust in db.Customers
                  where cust.CustomerID == "LAZYK"
                  select cust).
        Single<Customer>();
Console.WriteLine("Title from entity object : {0}", c2.ContactTitle);

```

В этом фрагменте я изменяю ContactTitle сущностного объекта LINQ to SQL, представляющего заказчика. Затем я запрашиваю ContactTitle из базы данных и из сущностного объекта и отображаю их. На этот раз значения ContactTitle не должны совпадать, потому что изменения еще не были сохранены в базе данных.

```

db.SubmitChanges();
Console.WriteLine(String.Format(
    "{0}SubmitChanges() method has been called.",
    System.Environment.NewLine));
title = GetStringFromDb(sqlConn, sqlQuery);
Console.WriteLine("Title from database record: {0}", title);
Console.WriteLine("Restoring ContactTitle back to original value ...");
c.ContactTitle = "Marketing Manager";
db.SubmitChanges();
Console.WriteLine("ContactTitle restored.");
}
finally
{
    sqlConn.Close();
}

```

В предыдущем коде я вызываю метод `SubmitChanges` и затем вновь извлекаю `ContactTitle`, чтобы отобразить его снова. На этот раз значение в базе данных должно быть обновлено, потому что метод `SubmitChanges` уже сохранил изменение в базе.

И, наконец, я устанавливаю `ContactTitle` обратно в исходное значение и сохраняю его в базе данных, используя метод `SubmitChanges`, чтобы вернуть базу данных в исходное состояние, так чтобы можно было запустить пример несколько раз, и чтобы это никак не повлияло на остальные примеры.

Этот код много чего делает, но его цель — доказать, что изменения, проведенные в существенном объекте, не сохраняются в базе данных до тех пор, пока не будет вызван метод `SubmitChanges`. Когда вы видите вызов метода `GetStringFromDb`, это значит, что он извлекает `ContactTitle` непосредственно из базы данных, используя ADO.NET. Ниже показан результат:

```
Title from database record: Marketing Manager
Title from entity object : Marketing Manager
Change the title to 'Director of Marketing' in the entity object:
Title from database record: Marketing Manager
Title from entity object : Director of Marketing
SubmitChanges() method has been called.
Title from database record: Director of Marketing
Restoring ContactTitle back to original value ...
ContactTitle restored.
```

Как видите, значение `ContactTitle` не изменилось в базе данных до тех пор, пока не был вызван метод `SubmitChanges`.

Для примера второго прототипа `SubmitChanges` я намеренно вызову ошибки параллельного доступа к двум записям, обновляя их с ADO.NET между моментом запроса записей с LINQ to SQL, и моментом, когда я попытаюсь обновить их с LINQ to SQL. Я создам две конфликтующих записи, чтобы продемонстрировать разницу между `ConflictMode`, `FailOnFirstConflict` и `ConflictMode.ContinueOnConflict`.

Также вы увидите далее код, который вернет в базе данных значения `ContactTitle` в их исходное состояние. Это позволит запускать пример многократно. Если вы прекратите выполнение кода в отладчике, то вам может понадобиться вручную сбросить эти значения. В первом примере второго прототипа метода `SubmitChanges` в листинге 16.12 я установлю `ConflictMode` в `ContinueOnConflict`, чтобы вы сначала могли увидеть, как он обрабатывает множественные конфликты. Поскольку пример довольно сложен, я приведу его по частям, и буду сопровождать пояснениями.

Листинг 16.12. Второй прототип `SubmitChanges`, демонстрирующий `ContinueOnConflict`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("Querying for the LAZYK Customer with LINQ.");
Customer cust1 = (from c in db.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
Console.WriteLine("Querying for the LONEP Customer with LINQ.");
Customer cust2 = (from c in db.Customers
                  where c.CustomerID == "LONEP"
                  select c).Single<Customer>();
```

В этом коде я создаю объект `Northwind` `DataContext` и запрашиваю двух заказчиков — LAZYK и LONEP.

```
string cmd = @"update Customers set ContactTitle = 'Director of Marketing'
               where CustomerID = 'LAZYK';
               update Customers set ContactTitle = 'Director of Sales'
               where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);
```

Далее я обновляю значение ContactValue в базе данных для обоих заказчиков, применяя общий метод ExecuteStatementInDb, который использует ADO.NET для проведения изменений. В этой точке я создаю потенциальную возможность конфликта параллельного доступа для каждой записи.

```
Console.WriteLine("Change ContactTitle in entity objects for LAZYK and LONEP.");
cust1.ContactTitle = "Vice President of Marketing";
cust2.ContactTitle = "Vice President of Sales";
```

В этой части кода я обновляю ContactTitle для каждого заказчика, так что при вызове метода SubmitChanges в следующей части процессор изменений объекта DataContext попытается сохранить изменения для этих двух заказчиков и обнаружить конфликты параллельного доступа.

```
try
{
    Console.WriteLine("Calling SubmitChanges() ...");
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
    Console.WriteLine("SubmitChanges() called successfully.");
}
```

В этом коде я вызываю метод SubmitChanges. Это заставит процессор изменений объекта DataContext попытаться сохранить этих двух заказчиков; но поскольку значение каждого ContactTitle каждого заказчика в базе данных отличается от того, что было изначально из нее загружено, будет обнаружен конфликт параллельного доступа.

```
catch (ChangeConflictException ex)
{
    Console.WriteLine("Conflict(s) occurred calling SubmitChanges(): {0}.",
        ex.Message);
    foreach (ObjectChangeConflict objectConflict in db.ChangeConflicts)
    {
        Console.WriteLine("Conflict for {0} occurred.",
            ((Customer)objectConflict.Object).CustomerID);
        foreach (MemberChangeConflict memberConflict in objectConflict.MemberConflicts)
        {
            Console.WriteLine(" LINQ value = {0}{1} Database value = {2}",
                memberConflict.CurrentValue,
                System.Environment.NewLine,
                memberConflict.DatabaseValue);
        }
    }
}
```

В этом фрагменте я перехватаю исключение ChangeConflictException. Именно здесь начинается самое интересное. Обратите внимание, что я сначала выполняю перечисление коллекции ChangeConflicts объекта DataContext — db. Эта коллекция будет хранить объекты ObjectChangeConflict. Заметьте, что объекты ObjectChangeConflict имеют свойство по имени Object, которое ссылается на действительный сущностный объект, при сохранении которого случается конфликт параллельного доступа. Я просто выполняю приведение члена Object к типу данных сущностного класса, чтобы обратиться к значениям свойств сущностного объекта. В данном случае я обращаюсь к свойству CustomerID.

Затем для каждого объекта ObjectChangeConflict я выполняю перечисление его коллекции объектов MemberChangeConflict и отображаю информацию каждого из тех, что меня интересуют. В данном случае я отображаю значение LINQ и значение из базы данных.

```

Console.WriteLine("{0}Resetting data to original values.",  

    System.Environment.NewLine);  

cmd = @"update Customers set ContactTitle = 'Marketing Manager'  

        where CustomerID = 'LAZYK';  

        update Customers set ContactTitle = 'Sales Manager'  

        where CustomerID = 'LONEP'";  

ExecuteStatementInDb(cmd);

```

В этом фрагменте кода я просто восстанавливаю базу данных обратно в ее исходное состояние, чтобы пример можно было запускать много раз.

Я продемонстрировал довольно большой объем кода. Имейте в виду, что ни одно из этих перечислений по различным коллекциям конфликтов не является необходимым. Я просто показываю, как вы можете делать это, и предоставляю некоторую доступную информацию о конфликтах, которую вы можете увидеть.

Также обратите внимание, что я ничего не делаю в этом примере в плане разрешения конфликтов. Я просто сообщаю о них.

Вот результат работы этого кода:

```

Querying for the LAZYK Customer with LINQ.  

Querying for the LONEP Customer with LINQ.  

Executing SQL statement against database with ADO.NET ...  

Database updated.  

Change ContactTitle in entity objects for LAZYK and LONEP.  

Calling SubmitChanges() ...  

Conflict(s) occurred calling SubmitChanges(): 2 of 2 updates failed.  

Conflict for LAZYK occurred.  

    LINQ value = Vice President of Marketing  

    Database value = Director of Marketing  

Conflict for LONEP occurred.  

    LINQ value = Vice President of Sales  

    Database value = Director of Sales  

Resetting data to original values.  

Executing SQL statement against database with ADO.NET ...  

Database updated.

```

Как видите, было обнаружено два конфликта — по одному для каждой из двух записей, для которых я создал конфликт. Это демонстрирует, что процессор изменений не прекратил попыток сохранить изменения в базе данных после того, как произошел первый конфликт. Это связано с тем, что при вызове `SubmitChanges` я передал `ConflictMode`, равное `ContinueOnConflict`.

Листинг 16.13 содержит такой же код, за исключением того, что в нем при вызове `SubmitChanges` я передаю `ConflictMode`, равное `FailOnFirstConflict`.

Листинг 16.13. Второй прототип `SubmitChanges`, демонстрирующий `FailOnFirstConflict`

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");  

Console.WriteLine("Querying for the LAZYK Customer with LINQ.");  

Customer cust1 = (from c in db.Customers  

                    where c.CustomerID == "LAZYK"  

                    select c).Single<Customer>();  

Console.WriteLine("Querying for the LONEP Customer with LINQ.");  

Customer cust2 = (from c in db.Customers  

                    where c.CustomerID == "LONEP"  

                    select c).Single<Customer>();  

string cmd = @"update Customers set ContactTitle = 'Director of Marketing'  

        where CustomerID = 'LAZYK';  

        update Customers set ContactTitle = 'Director of Sales'  

        where CustomerID = 'LONEP'";  

ExecuteStatementInDb(cmd);

```

```

Console.WriteLine("Change ContactTitle in entity objects for LAZYK and LONEP.");
cust1.ContactTitle = "Vice President of Marketing";
cust2.ContactTitle = "Vice President of Sales";
try
{
    Console.WriteLine("Calling SubmitChanges() ...");
    db.SubmitChanges(ConflictMode.FailOnFirstConflict);
    Console.WriteLine("SubmitChanges() called successfully.");
}
catch (ChangeConflictException ex)
{
    Console.WriteLine("Conflict(s) occurred calling SubmitChanges(): {0}",
        ex.Message);
    foreach (ObjectChangeConflict objectConflict in db.ChangeConflicts)
    {
        Console.WriteLine("Conflict for {0} occurred.",
            ((Customer)objectConflict.Object).CustomerID);
        foreach (MemberChangeConflict memberConflict in objectConflict.MemberConflicts)
        {
            Console.WriteLine(" LINQ value = {0}{1} Database value = {2}",
                memberConflict.CurrentValue,
                System.Environment.NewLine,
                memberConflict.DatabaseValue);
        }
    }
}
Console.WriteLine("{0}Resetting data to original values.",
    System.Environment.NewLine);
cmd = @"update Customers set ContactTitle = 'Marketing Manager'
        where CustomerID = 'LAZYK';
        update Customers set ContactTitle = 'Sales Manager'
        where CustomerID = 'LONEP'";
ExecuteStatementInDb(cmd);

```

На этот раз результат должен указывать на то, что обработка изменений в сущностных объектах останавливается, как только обнаружен первый конфликт параллельного доступа. Взглянем на результат:

```

Querying for the LAZYK Customer with LINQ.
Querying for the LONEP Customer with LINQ.
Executing SQL statement against database with ADO.NET ...
Database updated.
Change ContactTitle in entity objects for LAZYK and LONEP.
Calling SubmitChanges() ...
Conflict(s) occurred calling SubmitChanges(): Row not found or changed.
Conflict for LAZYK occurred.
    LINQ value = Vice President of Marketing
    Database value = Director of Marketing
Resetting data to original values.
Executing SQL statement against database with ADO.NET ...
Database updated.

```

Как видите, даже несмотря на то, что я инициировал два конфликта, процессор изменений прекращает попытки сохранить изменения в базе данных, как только возникает конфликт, о чём свидетельствует единственное сообщение о конфликте.

DatabaseExists ()

Метод DatabaseExists может быть использован для определения существования базы данных. Определение существования базы данных основано на строке соединения, специфицированной при создании экземпляра DataContext. Если вы специфицируете путь к файлу .mdf, он будет искать базу данных в пути с указанным именем. Если вы специфицируете сервер, он проверит сервер.

Метод DatabaseExists часто применяется в сочетании с методами DeleteDatabase и CreateDatabase.

Прототипы

Метод DatabaseExists имеет один прототип, описанный ниже.

Единственный прототип DatabaseExists

```
bool DatabaseExists()
```

Этот метод вернет true, если база данных, специфицированная в строке соединения, существует при создании экземпляра DataContext. В противном случае он вернет false.

Примеры

К счастью, этот метод продемонстрировать очень просто. В листинге 16.14 я просто создам экземпляр DataContext и вызову метод DatabaseExists, чтобы посмотреть, существует ли база данных Northwind. И, конечно же, мне известно, что это так.

Листинг 16.14. Пример вызова метода DatabaseExists

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("The Northwind database {0}.",
    db.DatabaseExists() ? "exists" : "does not exist");
```

Вот результат:

The Northwind database exists.

Если вы отключите базу данных Northwind и запустите пример снова, то получите следующий результат:

The Northwind database does not exist.

Если вы попробуете это сделать, не забудьте затем вновь подключить базу данных Northwind, чтобы работали другие примеры.

CreateDatabase ()

Поскольку сущностные классы знают настолько много о структуре базы данных, на которую они отображаются, был предоставлен метод CreateDatabase, предназначенный для создания базы данных.

Вы должны осознавать, однако, что он может создавать только те части базы данных, о которых ему известно через сущностные классы или файл отображения. Поэтому содержимое таких вещей, как хранимые процедуры, триггеры, определенные пользователем функции и ограничения (constraints) не будут включены в базу данных, создаваемую подобным способом, поскольку нет атрибутов, специфицирующих эту информацию. Для простого приложения, однако, это может оказаться вполне приемлемым.

На заметку! В отличие от большинства прочих изменений, которые вы проводите в базе данных через `DataContext`, метод `CreateDatabase` выполняется немедленно. Нет необходимости вызывать метод `SubmitChanges`, и выполнение не является отложенным. Это предоставляет вам преимущество в виде возможности создать базу данных и начать вставлять в нее данные немедленно.

Прототипы

Метод `CreateDatabase` имеет один прототип, описанный ниже.

Единственный прототип `CreateDatabase`

```
void CreateDatabase()
```

Этот метод не принимает аргументов и ничего не возвращает.

Примеры

Опять-таки, применение этого метода продемонстрировать просто, и листинг 16.15 доказывает это.

Листинг 16.15. Пример вызова метода `CreateDatabase`

```
Northwind db = new Northwind(@"C:\Northwnd.mdf");
db.CreateDatabase();
```

На заметку! Я намеренно написал `Northwnd` без буквы *i* в листинге 16.15, чтобы это не затронуло базу `Northwind` (с буквой *i*), если она у вас есть.

Этот код не производит никакого экранного вывода, так что нет результатов, которые стоило бы показать. Однако если я загляну в каталог `C:\`, то увижу там файлы `Northwnd.mdf` и `Northwnd.ldf`. Также, если я загляну в SQL Server Management Studio, то увижу присоединенный файл `C:\Northwnd.mdf`. Этот метод лучше всего сочетать с методом `DatabaseExists`. Если вы попытаетесь вызвать метод `CreateDatabase`, когда база данных уже существует, будет сгенерировано исключение. Чтобы продемонстрировать сказанное, просто запустите код из листинга 16.15 второй раз, не удаляя или не отключая файл `C:\Northwnd.mdf` от вашего SQL Server Management Studio или Enterprise Manager, и вы получите следующий вывод:

```
Unhandled Exception: System.Data.SqlClient.SqlException: Database 'C:\Northwnd.mdf' already exists. Choose a different database name.
```

```
Необработанное исключение: System.Data.SqlClient.SqlException: База данных 'C:\Northwnd.mdf' уже существует. Выберите другое имя для базы данных.
```

...

Кроме того, не допускайте ошибку, предполагая, что вы можете просто удалить файлы базы данных `Northwind`, которые были созданы, из файловой системы, чтобы избавиться от базы данных, дабы вы могли запустить пример снова. SQL Server по-прежнему будет содержать их в своем каталоге. Чтобы второй запуск метода `CreateDatabase` удался, вы должны в правильном порядке отсоединить и удалить базу данных.

Вы можете удалить или отключить вновь созданную базу данных, чтобы предотвратить путаницу в будущем, или же просто оставить ее на месте для того, чтобы ее удалил следующий пример, показанный в листинге 16.16.

DeleteDatabase()

LINQ to SQL предоставляет нам возможность удаления базы данных с помощью метода `DeleteDatabase` объекта `DataContext`. Попытка удалить базу данных, которая не существует, приведет к генерации исключения, так что будет лучше вызывать этот метод после проверки существования базы данных методом `DatabaseExists`.

Внимание! В отличие от большинства других изменений, которые вы проводите в базе данных через `DataContext`, метод `DeleteDatabase` выполняется немедленно. Нет необходимости вызывать `SubmitChanges`, и выполнение не является отложенным.

Прототипы

Метод `DeleteDatabase` имеет единственный прототип, описанный ниже.

Единственный прототип `DeleteDatabase`

```
void DeleteDatabase()
```

Этот метод не принимает аргументов и ничего не возвращает.

Примеры

В листинге 16.16 я удалю базу данных, только что созданную в листинге 16.15.

Листинг 16.16. Пример вызова метода `DeleteDatabase`

```
Northwind db = new Northwind(@"C:\Northwnd.mdf");
db.DeleteDatabase();
```

Этот пример не создает никакого экранного вывода при запуске — до тех пор, пока указанная база данных существует, но после запуска его вы обнаружите, что два файла базы данных, которые были созданы вызовом `CreateDatabase`, исчезли.

Вызов этого метода при несуществующей базе данных вызовет генерацию следующего исключения:

Unhandled Exception: System.Data.SqlClient.SqlException: An attempt to attach an auto-named database for file C:\Northwnd.mdf failed. A database with the same name exists, or specified file cannot be opened, or it is located on UNC share.

Необработанное исключение: `System.Data.SqlClient.SqlException`: Попытка присоединения к автоматически именованной базе данных для файла C:\Northwnd.mdf завершилась сбоем. База данных с тем же именем уже существует, или указанный файл не может быть открыт, или он расположен на разделяемом ресурсе UNC.

...

CreateMethodCallQuery()

Первое, что вам нужно знать о методе `CreateMethodCallQuery` — это то, что это защищенный (`protected`) метод. Это значит, что вы не можете вызывать этот метод из кода вашего приложения, и что вы должны наследовать свой класс от `DataContext`, дабы иметь возможность вызвать его.

Метод `CreateMethodCallQuery` используется для вызова возвращающих табличные значения пользовательских функций. Метод `ExecuteMethodCall` применяется для вызова пользовательских функций, возвращающих скалярные значения, и его мы обсудим далее в этой главе.

Прототипы

Метод CreateMethodCallQuery имеет один прототип, описанный ниже.

Единственный прототип CreateMethodCallQuery

```
protected internal IQueryable<T> CreateMethodCallQuery<T>(
    object instance,
    System.Reflection.MethodInfo methodInfo,
    params object[] parameters)
```

Метод CreateMethodCallQuery принимает ссылку на объект DataContext или [Your]DataContext, членом которого является метод, вызывающий CreateMethodCallQuery, объект MethodInfo для этого вызывающего метода и массив params параметров для возвращающей табличное значение пользовательской функции.

Примеры

Поскольку метод CreateMethodCallQuery является protected и может быть вызван только из класса DataContext или его наследника, вместо представления примера, который действительно вызывает метод CreateMethodCallQuery, я предполагаю обсудить метод, сгенерированный SQLMetal для пользовательской функции, возвращающей табличное значение ProductsUnderThisUnitPrice из расширенной базы данных Northwind. Вот этот метод.

Сгенерированный SQLMetal метод, вызывающий CreateMethodCallQuery

```
[Function(Name="dbo.ProductsUnderThisUnitPrice", IsComposable=true)]
public IQueryable<ProductsUnderThisUnitPriceResult>
    ProductsUnderThisUnitPrice(
        [Parameter(DbType="Money")] System.Nullable<decimal> price)
{
    return this.CreateMethodCallQuery<ProductsUnderThisUnitPriceResult>(
        this, ((MethodInfo)(MethodInfo.GetCurrentMethod())), price);
}
```

В приведенном выше коде вы можете видеть, что метод ProductsUnderThisUnitPrice снабжен атрибутом Function, поэтому мы знаем, что он собирается вызывать либо хранимую процедуру, либо определяемую пользователем функцию по имени Products UnderThisUnitPrice. Поскольку свойство IsComposable атрибута установлено в true, мы знаем, что это определяемая пользователем функция, а не хранимая процедура. Поскольку сгенерированный код вызывает метод CreateMethodCallQuery, мы знаем, что специфицированная пользовательская функция ProductsUnderThisUnitPrice возвращает табличное значение, а не скалярное.

Из аргументов, переданных методу CreateMethodCallQuery, первый аргумент — ссылка на производный от DataContext класс, сгенерированный для меня SQLMetal. Второй аргумент объект MethodInfo текущего метода. Это позволит методу CreateMethodCallQuery получить доступ к атрибутам, так что он будет иметь необходимую информацию для вызова определяемой пользователем функции, возвращающей табличное значение, такую как ее имя. Третий аргумент, переданный методу CreateMethodCallQuery — это единственный параметр, который принимает указанная пользовательская функция, которым в данном случае является цена.

Значение, возвращенное вызовом метода CreateMethodCallQuery, будет возвращено методом ProductsUnderThisUnitPrice, и им будет последовательность объектов ProductsUnderThisUnitPriceResult. SQLMetal был настолько любезен, что также сгенерировал для меня класс ProductsUnderThisUnitPriceResult.

Код, о котором шла речь, показывает, как вызывать метод `CreateMethodCallQuery`, но, просто чтобы задать некоторый контекст, давайте рассмотрим пример вызова сгенерированного метода `ProductsUnderThisUnitPriceResult`, чтобы вы увидели его в действии.

Листинг 16.17. Пример вызова метода `ProductsUnderThisUnitPrice`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<ProductsUnderThisUnitPriceResult> results =
    db.ProductsUnderThisUnitPrice(new Decimal(5.50));
foreach (ProductsUnderThisUnitPriceResult prod in results)
{
    Console.WriteLine("{0} - {1:C}", prod.ProductName, prod.UnitPrice);
}
```

Вот результат работы этого примера:

```
Guaraná Fantástica - $4.50
Geitost - $2.50
```

ExecuteQuery()

Нет сомнений, что LINQ to SQL великолепен. Использование стандартной точечной нотации LINQ или синтаксиса выражений превращает построение запросов LINQ в пару пустяков. Но рано или поздно у каждого из нас возникает потребность просто выполнить запрос SQL. С помощью LINQ to SQL можно делать и это. На самом деле вы можете поступать так и получать в результате те же сущностные объекты. Это потрясающе.

Метод `ExecuteQuery` позволяет вам специфицировать запрос SQL как строку, и даже включать в нее параметры для подстановки, как вы это делаете при вызове метода `String.Format`, причем полученный результат будет транслирован в последовательность сущностных объектов.

Это очень просто. Я догадываюсь, что вы скажете — как насчет ошибок внедрения SQL? Разве соответствующий способ делать это требует использования параметров? Да, требует. И метод `ExecuteQuery` позаботится обо всем для вас! Теперь я знаю, что вы с нетерпением ждете примера.

Прототипы

Метод `ExecuteQuery` имеет один прототип, описанный ниже.

Единственный прототип `ExecuteQuery`

```
IEnumerable<T> ExecuteQuery<T>(string query, params object[] parameters)
```

Этот метод принимает, как минимум, один аргумент — запрос SQL и ноль или более параметров. Стока запроса и необязательные параметры работают как метод `String.Format`. Метод возвращает последовательность типа T, где T — сущностный класс.

Имейте в виду, что если вы специфицируете значение столбца в конструкции `where` самой строки запроса, то должны заключать столбцы символьного типа в одиночные кавычки, как это принято в нормальном запросе SQL. Но если вы подставляете значение столбца как параметр, нет необходимости заключать спецификатор параметра, такой как `{0}`, в одиночные кавычки.

Чтобы столбец в запросе распространился на сущностный объект, имя столбца должно соответствовать одному из отображенных полей сущностного объекта. Конечно, вы можете достичь этого, добавляя `"as <columnname>"` к действительному имени столбца, где `<columnname>` — отображенный в сущностном объекте столбец.

Каждое отображенное поле не обязательно должно быть возвращено запросом, но первичные ключи — безусловно, должны. И вы можете извлекать поля в запросе, которые не отображаются ни на одно поле сущностного объекта, но они не будут распространяться на сущностный объект.

Примеры

Для демонстрации простого примера вызова метода ExecuteQuery в листинге 16.18 я обращусь к таблице Customers.

Листинг 16.18. Простой пример вызова метода ExecuteQuery

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
    @"select CustomerID, CompanyName, ContactName, ContactTitle
      from Customers where Region = {0}", "WA");
foreach (Customer c in custs)
{
    Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
        c.CustomerID, c.CompanyName, c.ContactName);
}
```

К этому примеру добавить нечего. Обратите внимание, что поскольку я использую средство подстановки параметров метода, специфицируя в качестве параметра "WA" вместо жесткого кодирования его в запросе, мне незачем заключать спецификатор формата в одиночные кавычки. Вот результат:

```
ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski
```

Если бы я хотел выполнить тот же запрос, но без подстановки параметров, то должен был бы заключить часть "WA" в одиночные кавычки, как это делается в нормальном SQL-запросе. Листинг 16.19 содержит соответствующий код.

Листинг 16.19. Другой простой пример вызова метода ExecuteQuery

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
    @"select CustomerID, CompanyName, ContactName, ContactTitle
      from Customers where Region = 'WA'");
foreach (Customer c in custs)
{
    Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
        c.CustomerID, c.CompanyName, c.ContactName);
}
```

В случае если вы сразу не заметили — WA теперь заключено в одиночные кавычки в строке запроса. Результат запуска этого кода такой же, как и предыдущего:

```
ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski
```

Вдобавок ко всему этому вы можете добавить специфицированное имя столбца, если реальное его имя не соответствует имени столбца в базе данных. Поскольку вы можете выполнять соединения в строке запроса, вы можете опрашивать столбцы с разными именами из разных таблиц, но специфицировать их имя, как одно из отображенных полей сущностного класса. Пример этого представлен в листинге 16.20.

Листинг 16.20. Пример вызова метода ExecuteQuery с указанием имени отображенного поля

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = db.ExecuteQuery<Customer>(
    @"select CustomerID, Address + ', ' + City + ', ' + Region as Address
     from Customers where Region = 'WA'");
foreach (Customer c in custs)
{
    Console.WriteLine("Id = {0} : Address = {1}",
        c.CustomerID, c.Address);
}

```

Интересная часть этого примера в том, что я соединяю несколько столбцов базы данных и строковых литералов, специфицируя имя отображенного поля, чтобы получить адрес, город и регион в единственном члене Address сущностного объекта. В этом случае все поля взяты из одной таблицы, но они вполне могли поступать из соединения с другой таблицей. Вот результат:

```

Id = LAZYK : Address = 12 Orchestra Terrace, Walla Walla, WA
Id = TRAIH : Address = 722 DaVinci Blvd., Kirkland, WA
Id = WHITC : Address = 305 - 14th Ave. S. Suite 3B, Seattle, WA

```

Конечно, применяя такой трюк, не забудьте о том, что если один из возвращенных сущностных объектов будет модифицирован, а затем будет вызван метод SubmitChanges, вы можете получить некоторые записи базы данных с сомнительной информацией. Но при правильном применении эта техника может быть весьма удобна.

Translate()

Метод Translate подобен ExecuteQuery в том, что он транслирует результаты SQL-запроса в последовательность сущностных объектов. Отличие заключается в том, что вместо передачи строки, содержащей оператор SQL, вы передаете ему объект типа System.Data.Common.DbDataReader, такой как SqlDataReader. Этот метод удобен для интеграции кода LINQ to SQL с существующим кодом ADO.NET.

Прототипы

Метод Translate имеет один прототип, описанный ниже.

Единственный прототип Translate

```
IEnumerable<T> Translate<T>(System.Data.Common.DbDataReader reader)
```

Вы передаете методу Translate объект типа System.Data.Common.DbDataReader, и метод Translate возвращает последовательность сущностных объектов.

Примеры

В листинге 16.21 я создам запрос, используя ADO.NET. Затем я применю метод Translate для трансляции результатов запроса в последовательность сущностных объектов Customer. Поскольку листинг 16.21 сложнее, чем обычно, я буду объяснять, что делаю.

Листинг 16.21. Пример вызова метода Translate

```

System.Data.SqlClient.SqlConnection sqlConn =
    new System.Data.SqlClient.SqlConnection(
        @"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=SSPI;");
string cmd = @"select CustomerID, CompanyName, ContactName, ContactTitle
               from Customers where Region = 'WA'";

```

```

System.Data.SqlClient.SqlCommand sqlComm =
    new System.Data.SqlClient.SqlCommand(cmd);
sqlComm.Connection = sqlConn;
try
{
    sqlConn.Open();
    System.Data.SqlClient.SqlDataReader reader = sqlComm.ExecuteReader();
}

```

Для этого примера давайте предположим, что весь предыдущий код уже существует. Представьте, что это унаследованный код, который мне нужно обновить, и я хочу воспользоваться преимуществом LINQ для решения моей задачи. Как вы можете видеть, в предыдущем коде нет никаких ссылок на LINQ. Подключение SqlConnection установлено, запрос сформирован, команда SqlCommand создана, соединение открыто, и запрос выполнен — все так, как в самом обычном запросе ADO.NET. Теперь давайте добавим некоторый код LINQ, чтобы сделать что-нибудь.

```

Northwind db = new Northwind(sqlConn);
IEnumerable<Customer> custs = db.Translate<Customer>(reader);
foreach (Customer c in custs)
{
    Console.WriteLine("ID = {0} : Name = {1} : Contact = {2}",
        c.CustomerID, c.CompanyName, c.ContactName);
}

```

В этом коде я создаю экземпляр Northwind DataContext, используя объект SqlConnection ADO.NET. Затем я вызываю метод Translate, передавая ему уже созданный reader, так чтобы результаты запроса могли быть преобразованы в существенные объекты, которые можно перечислять и отображать результаты.

Обычно, поскольку этот код унаследован, в нем присутствует еще что-то, что делается с результатами, но в нашем примере это не обязательно. Все, что остается добавить — код очистки метода.

```

}
finally
{
    sqlComm.Connection.Close();
}

```

Предыдущий код просто закрывает соединение. Этот пример демонстрирует, насколько изящно LINQ to SQL может взаимодействовать с ADO.NET. Взглянем на результат работы листинга 16.21.

```

ID = LAZYK : Name = Lazy K Kountry Store : Contact = John Steel
ID = TRAIH : Name = Trail's Head Gourmet Provisioners : Contact = Helvetius Nagy
ID = WHITC : Name = White Clover Markets : Contact = Karl Jablonski

```

ExecuteCommand()

Подобно ExecuteQuery, метод ExecuteCommand позволяет вам специфицировать действительный оператор SQL, чтобы выполнить его в базе данных. Это значит, что вы можете использовать его для выполнения операторов вставки, обновления или удаления, а также для вызова хранимых процедур. Кроме того, как в методе ExecuteQuery, вы можете передавать ему параметры.

Одна вещь, о которой следует помнить при вызове метода ExecuteCommand: он выполняется немедленно, потому вызывать метод SubmitChanges незачем.

Прототипы

Метод ExecuteCommand имеет единственный прототип, описанный ниже.

Единственный прототип ExecuteCommand

```
int ExecuteCommand(string command, params object[] parameters)
```

Этот метод принимает строку command и ноль или более необязательных параметров, и возвращает целое число, указывающее количество строк, затронутых запросом.

Имейте в виду, что если вы специфицируете значение столбца в конструкции where самой строки запроса, то должны заключать столбцы символьного типа в одиночные кавычки, как это принято в нормальном запросе SQL. Но если вы подставляете значение столбца как параметр, то нет необходимости заключать спецификатор параметра, такой как {0}, в одиночные кавычки.

Примеры

В листинге 16.22 я вставлю запись, используя метод ExecuteCommand. Поскольку я всегда отменяю изменения, которые провожу в базе данных, чтобы не пострадали последующие примеры, здесь я также использую метод ExecuteCommand для удаления вставленной записи.

Листинг 16.22. Пример использования метода ExecuteCommand для вставки и удаления записи

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Console.WriteLine("Inserting customer ...");
int rowsAffected = db.ExecuteCommand(
    @"insert into Customers values ({0}, 'Lawn Wranglers',
    'Mr. Abe Henry', 'Owner', '1017 Maple Leaf Way', 'Ft. Worth', 'TX',
    '76104', 'USA', '(800) MOW-LAWN', '(800) MOW-LAWO')", "LAWN");
Console.WriteLine("Insert complete.{0}", System.Environment.NewLine);
Console.WriteLine("There were {0} row(s) affected. Is customer in database?", 
    rowsAffected);
Customer cust = (from c in db.Customers
    where c.CustomerID == "LAWN"
    select c).DefaultIfEmpty<Customer>().Single<Customer>();
Console.WriteLine("{0}{1}",
    cust != null ?
    "Yes, customer is in database." : "No, customer is not in database.",
    System.Environment.NewLine);
Console.WriteLine("Deleting customer ...");
rowsAffected =
    db.ExecuteCommand(@"delete from Customers where CustomerID = {0}", "LAWN");
Console.WriteLine("Delete complete.{0}", System.Environment.NewLine);
```

Как видите, в этом примере нет ничего особенного. Я вызываю метод ExecuteCommand и передаю строку команды, плюс некоторые параметры. Затем выполняю запрос, используя LINQ to SQL — просто, чтобы убедиться, что запись действительно вставлена в базу данных, и отображаю результаты запроса на консоли. Для возврата базы в исходное состояние я снова вызываю метод ExecuteCommand, чтобы удалить только что вставленную запись. Этот код производит следующий результат:

```
Inserting customer ...
Insert complete.
There were 1 row(s) affected. Is customer in database?
Yes, customer is in database.
Deleting customer ...
Delete complete.
```

ExecuteMethodCall()

Первое, что вам нужно знать о методе ExecuteMethodCall — то, что этот метод является защищенным (`protected`). Это значит, что вы не сможете вызвать этот метод из кода вашего приложения, и что должны наследовать класс от класса `DataContext`, чтобы иметь возможность вызывать его.

Метод `ExecuteMethodCall` используется для вызова хранимых процедур и пользовательских функций, возвращающих скалярные значения. Чтобы вызывать пользовательские функции, возвращающие табличные значения, прочтите раздел настоящей главы, посвященный методу `CreateMethodCallQuery`.

Прототипы

Метод `ExecuteMethodCall` имеет один прототип, описанный ниже.

Единственный прототип ExecuteMethodCall

```
protected internal IExecuteResult ExecuteMethodCall(
    object instance,
    System.Reflection.MethodInfo methodInfo,
    params object[] parameters)
```

Методу `ExecuteMethodCall` передается ссылка на объект `DataContext` или `[Your]DataContext`, членом которого является метод, вызвавший `ExecuteMethodCall`, объект `MethodInfo` для вызова метода и массив `params` параметров хранимой процедуры или пользовательской функции, возвращающей скалярное значение.

Обратите внимание, что поскольку мы должны передавать объект `MethodInfo`, наш метод должен быть оснащен соответствующим атрибутом хранимой процедуры или пользовательской функции, с соответствующими свойствами атрибута. LINQ to SQL затем использует объект `MethodInfo` для доступа к атрибуту метода `Function`, чтобы получить имя хранимой процедуры или пользовательской функции, возвращающей скалярное значение. Также он использует объект `MethodInfo` для получения имен и типов параметров.

Метод `ExecuteMethodCall` возвращает объект, реализующий интерфейс `IExecuteResult`. Я расскажу о нем в главе 15.

Если вы используете `SQLMetal` для генерации ваших сущностных классов, он создаст методы сущностных классов, которые вызовут `ExecuteMethodCall` для всех хранимых процедур базы данных, если вы специфицируете опцию `/sprocs`, и для пользовательских функций базы данных — если укажете опцию `/function`.

Примеры

Прежде чем мы рассмотрим код первого примера, я хочу рассказать о методе по имени `CustomersCountByRegion`, который `SQLMetal` сгенерировал для вызова хранимой процедуры базы данных `Customers Count By Region`. Вот как выглядит этот генерированный метод.

Использование метода ExecuteMethodCall для вызова хранимой процедуры

```
[Function(Name="dbo.Customers Count By Region")]
[return: Parameter(DbType="Int")]
public int CustomersCountByRegion([Parameter(DbType="NVarChar(15)")] string param1)
{
    IExecuteResult result =
        this.ExecuteMethodCall(
            this,
            ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
    return ((int)(result.ReturnValue));
}
```

Как видите, методу `CustomersCountByRegion` передается параметр `string`, который затем передается в качестве параметра методу `ExecuteMethodCall`, который, в свою очередь, передает его хранимой процедуре `Customers Count By Region`.

Метод `ExecuteMethodCall` возвращает переменную, реализующую `IExecuteResult`. Чтобы получить целочисленное возвращаемое значение, метод `CustomersCountByRegion` просто ссылается на свойство возвращенного объекта `ReturnValue` и выполняет приведение его к `int`.

Теперь давайте заглянем в листинг 16.23, чтобы увидеть некоторый код, вызывающий сгенерированный метод `CustomersCountByRegion`.

Листинг 16.23. Пример вызова сгенерированного метода `CustomersCountByRegion`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
int rc = db.CustomersCountByRegion("WA");
Console.WriteLine("There are {0} customers in WA.", rc);
```

Это очень тривиальный пример, без каких-либо сюрпризов. Вот результат:

`There are 3 customers in WA`

Теперь я хочу поговорить о вызове хранимой процедуры, которая возвращает выходной параметр. Опять-таки, глядя на сгенерированные `SQLMetal` сущностные классы для базы данных `Northwind`, я расскажу о методе `CustOrderTotal`, сгенерированном для вызова хранимой процедуры `CustOrderTotal`.

Пример использования метода `ExecuteMethodCall` для вызова хранимой процедуры, которая возвращает выходной параметр

```
[Function(Name="dbo.CustOrderTotal")]
[return: Parameter(DbType="Int")]
public int CustOrderTotal(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID,
    [Parameter(Name="TotalSales", DbType="Money")] ref System.Nullable<decimal>
        totalSales)
{
    IExecuteResult result =
        this.ExecuteMethodCall(
            this,
            ((MethodInfo)(MethodInfo.GetCurrentMethod())),
            customerID,
            totalSales);
    totalSales = ((System.Nullable<decimal>)(result.GetParameterValue(1)));
    return ((int)(result.ReturnValue));
}
```

Обратите внимание, что второй параметр метода `CustOrderTotal` — `totalSales` — специфицирует ключевое слово `ref`. Это намек на то, что хранимая процедура собирается возвращать это значение. Заметьте, что для того, чтобы получить значение после вызова метода `ExecuteMethodCall`, код вызывает метод `GetParameterValue` на возвращенном объекте, реализующем `IExecuteResult`, и передает ему 1, поскольку нас интересует второй параметр. Код в листинге 16.24 вызывает метод `CustOrderTotal`.

Листинг 16.24. Пример вызова сгенерированного метода `CustOrderTotal`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
decimal? totalSales = 0;
int rc = db.CustOrderTotal("LAZYK", ref totalSales);
Console.WriteLine("Customer LAZYK has total sales of {0:C}.", totalSales);
```

Мне пришлось специфицировать ключевое слово `ref` для второго параметра `totalSales`. Вот результат:

```
Customer LAZYK has total sales of $357.00.
```

Теперь давайте взглянем на пример, который вызывает хранимую процедуру вызовом метода `ExecuteMethodCall`.

Пример использования метода ExecuteMethodCall для вызова хранимой процедуры, возвращающей одну форму.

```
[Function (Name="dbo.Customers By City")]
public ISingleResult<CustomersByCityResult>
    CustomersByCity([Parameter (DbType="NVarChar(20)")] string param1)
{
    IExecuteResult result =
        this.ExecuteMethodCall(
            this,
            ((MethodInfo) (MethodInfo.GetCurrentMethod())),
            param1);
    return ((ISingleResult<CustomersByCityResult>) (result.ReturnValue));
}
```

Обратите внимание, что сгенерированный метод возвращает объект типа `ISingleResult<CustomersByCityResult>`. Сгенерированный метод получает этот объект, выполняя приведение свойства `ReturnValue` возвращенного объекта к этому типу. `SQLMetal` был настолько любезен, что также сгенерировал для меня класс `CustomersByCityResult`, хотя я не буду обсуждать его здесь. Листинг 16.25 содержит код вызова сгенерированного метода `CustomersByCity`.

Листинг 16.25. Пример вызова сгенерированного метода CustomersByCity

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
ISingleResult<CustomersByCityResult> results = db.CustomersByCity("London");
foreach (CustomersByCityResult cust in results)
{
    Console.WriteLine("{0} - {1} - {2} - {3}", cust.CustomerID, cust.CompanyName,
        cust.ContactName, cust.City);
}
```

Как видите, я выполняю перечисление возвращенного объекта типа `ISingleResult<CustomersByCityResult>`, как если бы это была последовательность LINQ. Это потому, что он наследуется от `IEnumerable<T>`, о чем я упоминал в главе 15. Затем я отображаю результаты на консоли.

Результаты выглядят так:

```
AROUT - Around the Horn - Thomas Hardy - London
BSBEV - B's Beverages - Victoria Ashworth - London
CONSH - Consolidated Holdings - Elizabeth Brown - London
EASTC - Eastern Connection - Ann Devon - London
NORTS - North/South - Simon Crowther - London
SEVES - Seven Seas Imports - Hari Kumar - London
```

Теперь давайте взглянем на некоторые примеры, возвращающие результат в нескольких формах. Для тех из вас, кто не знаком с термином *форма (shape)* в данном контексте, поясню: форма результата определяется типом возвращенных данных. Когда запрос возвращает идентификатор (ID) и имя заказчика — это форма. Если запрос возвращает ID заказа, дату заказа и код поставки — это еще одна форма. Если запрос возвращает и то, и другое, запись, содержащую ID заказчика и имя, и что-то еще, а также

запись, содержащую ID заказа, дату заказа и код поставки, то такой запрос возвращает несколько результирующих форм. Поскольку хранимые процедуры обладают такой способностью, LINQ to SQL необходим какой-то способ, чтобы справиться с этим, и такой способ есть.

Для первого примера, возвращающего несколько форм, представим сценарий, где форма результата определяется некоторым условием. К счастью, расширенная база данных Northwind имеет хранимую процедуру такого типа. Ее имя — Whole Or Partial Customers Set. SQLMetal сгенерировал для меня метод для вызова этой хранимой процедуры по имени WholeOrPartialCustomersSet, который показан ниже.

Пример, использующий вызов метода ExecuteMethodCall для вызова хранимой процедуры, которая возвращает результаты различной формы в зависимости от условия

```
[Function(Name="dbo.Whole Or Partial Customers Set")]
[ResultType(typeof(WholeOrPartialCustomersSetResult1))]
[ResultType(typeof(WholeOrPartialCustomersSetResult2))]
public IMultipleResults WholeOrPartialCustomersSet(
[Parameter(DbType="Int")] System.Nullable<int> param1)
{
    IExecuteResult result =
        this.ExecuteMethodCall(
            this,
            ((MethodInfo)(MethodInfo.GetCurrentMethod())),
            param1);
    return ((IMultipleResults)(result.ReturnValue));
}
```

Обратите внимание, что есть два атрибута ResultType, специфицирующих две возможные формы результата. SQLMetal был настолько любезен, что сгенерировал для меня два указанных класса. Разработчик, вызывающий метод WholeOrPartialCustomersSet, должен иметь в виду, что хранимая процедура возвращает разные результаты в зависимости от значения param1.

Поскольку речь идет о хранимой процедуре, я знаю, что если param1 равен 1, то хранимая процедура вернет все поля из таблицы Customer, и потому метод вернет последовательность объектов типа WholeOrPartialCustomersSetResult1. Если же значение param1 равно 2, то будет возвращен сокращенный набор полей в последовательности объектов типа WholeOrPartialCustomersSetResult2.

Также заметьте, что типом возврата метода WholeOrPartialCustomersSet является IMultipleResults. Метод получает его, приводя свойство ReturnValue объекта, возвращенного методом ExecuteMethodCall к IMultipleResults. Я рассказывал об этом интерфейсе в главе 15.

В листинге 16.26 представлен пример вызова метода WholeOrPartialCustomersSet.

Листинг 16.26. Пример вызова сгенерированного метода WholeOrPartialCustomersSet

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IMultipleResults results = db.WholeOrPartialCustomersSet(1);
foreach (WholeOrPartialCustomersSetResult1 cust in
    results.GetResult<WholeOrPartialCustomersSetResult1>())
{
    Console.WriteLine("{0} - {1} - {2} - {3}", cust.CustomerID, cust.CompanyName,
        cust.ContactName, cust.City);
}
```

Обратите внимание, что результат имеет тип IMultipleResults.

Я передал значение 1, поэтому знаю, что получу последовательность типа `WholeOrPartialCustomersSetResult1`. Также заметьте, что для того, чтобы получить результаты, я вызываю метод `GetResult<T>` на переменной `IMultipleResults`, где `T` — тип возвращенных данных. Вот результат:

```
LAZYK - Lazy K Kountry Store - John Steel - Walla Walla
TRAIH - Trail's Head Gourmet Provisioners - Helvetius Nagy - Kirkland
WHITC - White Clover Markets - Karl Jablonski - Seattle
```

Эта хранимая процедура извлекает только заказчиков из региона "WA". Если бы я передал значение 2 при вызове метода `WholeOrPartialCustomersSet`, то получил бы последовательность типа `WholeOrPartialCustomersSetResult2`, так что везде в предыдущем коде, где я специфицировал тип `WholeOrPartialCustomersSetResult1`, нужно было бы заменить его на `WholeOrPartialCustomersSetResult2`.

Теперь нам осталось рассмотреть только случай, когда хранимая процедура возвращает несколько форм за один вызов. В этом нам поможет хранимая процедура из расширенной базы данных Northwind под названием `Get Customer And Orders`. Для начала, давайте посмотрим на метод, сгенерированный SQLMetal для вызова этой хранимой процедуры.

Пример использования метода `ExecuteMethodCall` для вызова хранимой процедуры, возвращающей множественные формы

```
[Function(Name="dbo.Get Customer And Orders")]
[ResultType(typeof(GetCustomerAndOrdersResult1))]
[ResultType(typeof(GetCustomerAndOrdersResult2))]
public IMultipleResults GetCustomerAndOrders(
    [Parameter(Name="CustomerID", DbType="NChar(5)")] string customerID)
{
    IExecuteResult result =
        this.ExecuteMethodCall(
            this,
            ((MethodInfo)(MethodInfo.GetCurrentMethod())),
            customerID);
    return ((IMultipleResults)(result.ReturnValue));
}
```

Как видите, типом возврата метода является `IMultipleResults`. Поскольку хранимая процедура возвращает результат в нескольких формах, на нас ложится ответственность в определении порядка возвращаемых форм. Поскольку я просмотрел хранимую процедуру `Get Customer And Orders`, мне известно, что она сначала вернет запись из таблицы `Customers`, а за ней — связанные записи из таблицы `Orders`.

Код в листинг 16.27 вызывает сгенерированный метод из предыдущего кода.

Листинг 16.27. Пример вызова сгенерированного метода `GetCustomerAndOrders`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IMultipleResults results = db.GetCustomerAndOrders("LAZYK");
GetCustomerAndOrdersResult1 cust =
    results.GetResult<GetCustomerAndOrdersResult1>().Single();
Console.WriteLine("{0} orders:", cust.CompanyName);
foreach (GetCustomerAndOrdersResult2 order in
    results.GetResult<GetCustomerAndOrdersResult2>())
{
    Console.WriteLine("{0} - {1}", order.OrderID, order.OrderDate);
}
```

Поскольку я знаю, что хранимая процедура вернет единственную запись, соответствующую типу `GetCustomerAndOrdersResult1`, я знаю, что могу вызвать операцию `Single` на последовательности, содержащей этот тип — до тех пор, пока уверен, что нужный мне заказчик с переданным `CustomerID` существует. Если же я не уверен в этом, то всегда могу вызвать операцию `SingleOrDefault`. Также мне известно, что после возврата единственного объекта `GetCustomerAndOrdersResult1` должно бытьозвращено ноль или более объектов `GetCustomerAndOrdersResult2`, поэтому я выполняю их перечисление, отображая интересующие меня данные. Вот результат:

```
Lazy K Kountry Store orders:
10482 - 3/21/1997 12:00:00 AM
10545 - 5/22/1997 12:00:00 AM
```

Это завершает примеры вызовов хранимых методом `ExecuteMethodCall`. В начале раздела, посвященного этому методу, я сказал, что этот метод используется и для вызова пользовательских функций, возвращающих скалярные значения. Давайте рассмотрим пример вызова пользовательской функции, возвращающей скалярное значение.

Сначала посмотрим на сгенерированный `SQLMetal` метод под названием `ExecuteMethodCall` для вызова пользовательской функции скалярного значения.

Пример использования метода `ExecuteMethodCall` для вызова определяемой пользователем функции, возвращающей скалярное значение

```
[Function(Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
[return: Parameter(DbType="Money")]
public System.Nullable<decimal>.MinUnitPriceByCategory(
    [Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
    return ((System.Nullable<decimal>) (this.ExecuteMethodCall(this,
        ((MethodInfo) (MethodInfo.GetCurrentMethod())), categoryID).ReturnValue));
}
```

Обратите внимание, что скалярное значение, возвращенное хранимой процедурой, получается через ссылку на свойство `ReturnValue` возвращенного методом `ExecuteMethodCall` объекта.

Я мог бы создать простой пример вызова сгенерированного метода `MinUnitPriceByCategory`. Однако вся прелесть определяемой пользователем функции проявляется при включении ее в запрос, подобно встроенным функциям SQL.

Взглянем на пример, представленный в листинге 16.28, который встраивает метод `MinUnitPriceByCategory` в запрос для идентификации всех продуктов с минимальной ценой в своей категории.

Листинг 16.28. Пример встраивания определенной пользователем функции в запрос

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Product> products = from p in db.Products
                                where p.UnitPrice ==
                                    db.MinUnitPriceByCategory(p.CategoryID)
                                select p;
foreach (Product p in products)
{
    Console.WriteLine("{0} - {1:C}", p.ProductName, p.UnitPrice);
}
```

В этом примере я встраиваю в конструкцию `where` вызов метода `MinUnitPriceByCategory`, который, в свою очередь, вызывает одноименную скалярную пользовательскую функцию. Вот результат:

Appleseed Syrup - \$10.00
 Ponbu - \$6.00
 Teatime Chocolate Biscuits - \$9.20
 Guaraná Fantástica - \$4.50
 Griffest - \$2.50
 Filo Mix - \$7.00
 Tourtière - \$7.45
 Longlife Tofu - \$10.00

GetCommand()

Еще один потенциально полезный метод — это `GetCommand`. Когда вызывается метод `GetCommand` на объекте `DataContext` и ему передается интерфейс LINQ to SQL по имени `IQueryable`, то возвращается объект типа `System.Data.Common.DbCommand`. Возвращенный объект `DbCommand` содержит доступ к нескольким ключевым компонентам, которые будут использованы запросом.

Извлекая объект `DbCommand` методом `GetCommand`, вы можете получить ссылку на объекты `CommandText`, `CommandTimeout`, `Connection`, `Parameters` и `Transaction`, наряду с некоторыми другими, относящимися к переданному запросу. Это позволяет вам не только просматривать эти объекты, но и модифицировать их, изменяя значения по умолчанию, без модификации одних и тех же запросов, которые будут выполнены текущим экземпляром `DataContext`. Возможно, для конкретного запроса вы пожелаете увеличить значение `CommandTimeout`, но так, чтобы не менять значение таймаута для всех остальных запросов, выполняемых с данным объектом `DataContext`.

Прототипы

Метод `GetCommand` имеет один прототип, описанный ниже.

Единственный прототип `GetCommand`

```
System.Data.Common.DbCommand GetCommand(IQueryable query)
```

Этому методу передается запрос LINQ to SQL в форме `IQueryable`, а возвращает он `System.Data.Common.DbCommand` этого запроса.

Примеры

В листинге 16.29 я получаю объект `DbCommand` для изменения `CommandTimeout` запроса и `CommandText` — для отображения, которым будет сам запрос SQL.

Листинг 16.29. Пример вызова метода `GetCommand`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<Customer> custs = from c in db.Customers
                               where c.Region == "WA"
                               select c;
System.Data.Common.DbCommand dbc = db.GetCommand(custs);
Console.WriteLine("Query's timeout is: {0}{1}", dbc.CommandTimeout,
                  System.Environment.NewLine);
dbc.CommandTimeout = 1;
Console.WriteLine("Query's SQL is: {0}{1}",
                  dbc.CommandText, System.Environment.NewLine);
Console.WriteLine("Query's timeout is: {0}{1}", dbc.CommandTimeout,
                  System.Environment.NewLine);
foreach (Customer c in custs)
{
    Console.WriteLine("{0}", c.CompanyName);
}
```

Об этом примере говорить особо нечего. Я просто объявляю запрос и передаю его методу GetCommand. Затем отображаю значение CommandTimeout для возвращенного объекта DbCommand. Затем устанавливаю значение CommandTimeout в 1 и отображаю сам запрос SQL вместе с новым значением CommandTimeout. И последнее — выполняю перечисление результатов, возвращенных запросом.

Вот результат запуска этого кода на моей машине:

```
Query's timeout is: 30
Query's SQL is: SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[Region] = @p0
Query's timeout is: 1
Lazy K Kountry Store
Trail's Head Gourmet Provisioners
White Clover Markets
```

Конечно, если этот запрос потребует слишком много времени для выполнения на вашей машине, запрос прервется по таймауту, и вы получите другие результаты.

GetChangeSet()

Иногда может быть удобно иметь возможность получить список всех сущностных объектов, которые будут вставлены, изменены или удалены при вызове метода SubmitChanges. Метод GetChangeSet служит именно для этого.

Прототипы

Метод GetChangeSet имеет один прототип, описанный ниже.

Единственный прототип GetChangeSet

```
ChangeSet GetChangeSet()
```

Этот метод не принимает ничего и возвращает объект ChangeSet. Объект ChangeSet содержит коллекции типа `IList<T>` для вставленных, модифицированных и удаленных объектов, где `T` — сущностный класс. Эти свойства-коллекции называются `Inserts`, `Updates` и `Deletes` соответственно.

Вы можете затем выполнить перечисление каждой из этих коллекций для того, чтобы просмотреть содержащиеся в них объекты.

Примеры

В листинге 16.30 я модифицирую, вставлю и удалю сущностный объект. Затем я извлеку ChangeSet, используя метод GetChangeSet, и выполню перечисление каждой коллекции.

Листинг 16.30. Пример вызова метода GetChangeSet

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
cust.Region = "Washington";
db.Customers.InsertOnSubmit(
    new Customer
    {
        CustomerID = "LAWN",
```

```

CompanyName = "Lawn Wranglers",
ContactName = "Mr. Abe Henry",
ContactTitle = "Owner",
Address = "1017 Maple Leaf Way",
City = "Ft. Worth",
Region = "TX",
PostalCode = "76104",
Country = "USA",
Phone = "(800) MOW-LAWN",
Fax = "(800) MOW-LAWO"
});
Customer cust2 = (from c in db.Customers
                    where c.CustomerID == "LONEP"
                    select c).Single<Customer>();
db.Customers.DeleteOnSubmit(cust2);
cust2 = null;
ChangeSet changeSet = db.GetChangeSet();
Console.WriteLine("{0}First, the added entities:", System.Environment.NewLine);
foreach (Customer c in changeSet.Inserts)
{
    Console.WriteLine("Customer {0} will be added.", c.CompanyName);
}
Console.WriteLine("{0}Second, the modified entities:", System.Environment.NewLine);
foreach (Customer c in changeSet.Updates)
{
    Console.WriteLine("Customer {0} will be updated.", c.CompanyName);
}
Console.WriteLine("{0}Third, the removed entities:", System.Environment.NewLine);
foreach (Customer c in changeSet.Deletes)
{
    Console.WriteLine("Customer {0} will be deleted.", c.CompanyName);
}

```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод InsertOnSubmit из приведенного выше кода назывался Add, а метод DeleteOnSubmit — Remove. Кроме того, коллекции ChangeSet назывались AddedEntities, ModifiedEntities и RemovedEntities, в противоположность Inserts, Updates и Deletes, как в листинге 16.30.

В предыдущем примере я сначала модифицирую Region заказчика LAZYK. Затем вставляю заказчика LAWN и удаляю заказчика LONEP. Далее я получаю ChangeSet, вызывая метод GetChangeSet. Затем я выполняю перечисление каждой коллекции — Inserts, Updates и Deletes — и отображаю каждый существенный объект в соответствующей коллекции.

Вот результат:

```

First, the added entities:
Customer Lawn Wranglers will be added.
Second, the modified entities:
Customer Lazy K Kountry Store will be updated.
Third, the removed entities:
Customer Lonesome Pine Restaurant will be deleted.

```

Конечно, в предыдущем примере я могу выполнять перечисление каждой коллекции, предполагая, что каждый элемент является объектом Customer, потому что знаю, что так оно и есть.

Однако во многих случаях в коллекции могут существовать более одного типа объектов, и вы не можете делать таких предположений. В таких ситуациях вы должны будете написать собственный перечислимый код для обработки множественных типов данных. Операция OfType может в этом помочь.

GetTable()

Метод GetTable используется для получения ссылки на последовательность Table из DataContext для определенной таблицы базы данных. Этот метод обычно применяется только именно вместе с классом DataContext, а не [Your]DataContext. Использование класса [Your]DataContext является предпочтительной техникой, поскольку этот класс уже будет иметь свойство-последовательность Table, ссылающееся на каждую отображенную таблицу.

Прототипы

Метод GetTable имеет два прототипа, описанных ниже.

Первый прототип GetTable

```
Table<T> GetTable<T>()
```

Этому методу предоставляется указанный отображаемый сущностный тип T, и он возвращает последовательность Table объектов типа T.

Второй прототип GetTable

```
ITable GetTable(Type type)
```

Этот метод принимает Type сущностного объекта и возвращает интерфейс таблицы. Затем вы можете использовать этот интерфейс ITable по своему усмотрению. Если вы хотите применять интерфейс ITable, как если бы это была таблица, не забудьте выполнить его приведение к IQueryable<T>.

Примеры

Для примера первого прототипа в листинге 16.31 я использую стандартный класс DataContext вместо моего класса [Your]DataContext — Northwind, чтобы извлечь определенного заказчика.

Листинг 16.31. Пример вызова первого прототипа GetTable

```
DataContext db =
    new DataContext(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.GetTable<Customer>()
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
Console.WriteLine("Customer {0} retrieved.", cust.CompanyName);
```

Здесь язываю метод GetTable для получения ссылки на таблицу Customer, чтобы иметь возможность извлечь конкретного заказчика. Вот результат:

```
Customer Lazy K Kountry Store retrieved.
```

Для примера второго прототипа метода GetTable я использую DataContext вместо моего [Your]DataContext. Листинг 16.32 содержит тот же базовый пример, что и предыдущий, но использует второй прототип.

Листинг 16.32. Пример вызова второго прототипа GetTable

```
DataContext db =
    new DataContext(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in ((IQueryable<Customer>)db.GetTable(typeof(Customer)))
                  where c.CustomerID == "LAZYK"
                  select c).Single<Customer>();
Console.WriteLine("Customer {0} retrieved.", cust.CompanyName);
```

Но должно быть сюрпризом, что результат работы кода из листинга 16.32 совпадает с тем, что давал код в листинге 16.31:

```
Customer Lazy K Kountry Store retrieved.
```

Refresh()

Метод Refresh позволяет вам вручную обновлять сущностные объекты из базы данных. В некоторых ситуациях это делается для вас, когда вы вызываете метод ResolveAll коллекции ChangeConflicts объекта DataContext в случае возникновения конфликтов во время вызова метода SubmitChanges. Однако могут быть ситуации, в которых вы никогда не вызываете метод SubmitChanges, но хотите получать обновления из базы данных.

Примером может служить приложение, которое отображает данные о состоянии некоторого сущностного объекта, системы или процесса, доступные только для чтения. Вы можете пожелать регулярно обновлять данные из базы через какой-то интервал времени. Для этой цели может быть использован метод Refresh.

Методом Refresh вы можете обновлять отдельный сущностный объект либо последовательность сущностных объектов — имеется в виду результат запроса LINQ to SQL.

Прототипы

Метод Refresh имеет три прототипа, описанных ниже.

Первый прототип Refresh

```
void Refresh(RefreshMode mode, object entity)
```

Этот метод принимает режим обновления и единственный сущностный объект и ничего не возвращает.

Второй прототип Refresh

```
void Refresh(RefreshMode mode, params object[] entities)
```

Этот метод принимает режим обновления и массив params сущностных объектов и ничего не возвращает.

Третий прототип Refresh

```
void Refresh(RefreshMode mode, System.Collections.IEnumerable entities)
```

Этот метод принимает режим обновления и последовательность сущностных объектов и ничего не возвращает.

Перечисление RefreshMode имеет три возможных значения: KeepChanges, KeepCurrentValues и OverwriteCurrentValues. Документация Visual Studio о перечислении RefreshMode определяет эти значения так, как описано в табл. 16.1.

Таблица 16.1. Перечисление RefreshMode

Имя члена	Описание
KeepCurrentValues	Заставляет метод Refresh обменять оригинальное значение объекта со значениями, извлеченными из базы данных.
KeepChanges	Заставляет метод Refresh сохранять текущее значение, которое было изменено, но обновляет другие значения прочитанными из базы.
OverwriteCurrentValues	Заставляет метод Refresh заменить все текущие значения значениями, взятыми из базы.

Поведение каждой из этих установок более подробно описано в главе 17.

Примеры

Для примера первого прототипа в листинге 16.33 я запрошу заказчика, используя LINQ to SQL, и отобразжу имя и титул контактного лица. Затем я изменю имя контактного лица этого заказчика в базе данных с помощью ADO.NET. Я изменю титул контактного лица в сущностном объекте. И чтобы доказать, что текущий сущностный объект не имеет понятия об изменении в базе данных, но имеет измененный титул контакта, я отобразжу снова имя и титул контактного лица, и вы увидите, что имя контактного лица не изменилось, тогда как титул изменился.

После этого я вызову метод Refresh с RefreshMode, равным KeepChanges, и отобразжу контактное имя и титул из сущностного объекта еще раз, и вы убедитесь, что он действительно содержит новое значение имени контактного лица из базы данных, в то же время предохраняя мое изменение в титуле контакта.

После этого я верну контактное имя в его оригинальное значение, чтобы этот пример можно было запускать многократно. Ниже показан необходимый код.

Листинг 16.33. Пример вызова первого прототипа метода Refresh

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = (from c in db.Customers
                  where c.CustomerID == "GREAL"
                  select c).Single<Customer>();
Console.WriteLine("Customer's original name is {0}, ContactTitle is {1}. {2}",
                  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set ContactName = 'Brad Radaker' where CustomerID = 'GREAL'"));
cust.ContactTitle = "Chief Technology Officer";
Console.WriteLine("Customer's name before refresh is {0}, ContactTitle is {1}. {2}",
                  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
db.Refresh(RefreshMode.KeepChanges, cust);
Console.WriteLine("Customer's name after refresh is {0}, ContactTitle is {1}. {2}",
                  cust.ContactName, cust.ContactTitle, System.Environment.NewLine);
// Я должен вернуть старые значения, чтобы код
// можно было запускать более одного раза.
Console.WriteLine("{0}Resetting data to original values.",
                  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set ContactName = 'John Steel' where CustomerID = 'GREAL'"));
```

В предыдущем коде я выполняю запрос LINQ to SQL для получения ссылки на объект Customer с идентификатором GREAL. Затем отображаю ContactName и ContactTitle объекта Customer.

После этого я обновлю ContactName заказчика в базе данных, используя для этого ADO.NET, и обновлю ContactTitle в моем извлеченном сущностном объекте Customer. В этой точке мой сущностный объект Customer не знает, что ContactName было изменено в базе данных, и я доказываю это, отображая значения полей ContactName и ContactTitle объекта Customer на консоли.

Далее я вызываю метод RefreshMethod с KeepChanges из RefreshMode. Это должно заставить любые свойства объекта Customer, которые были изменены в базе данных, загрузиться в мой сущностный объект, если только я не изменял их самостоятельно. В этом случае, поскольку ContactName было изменено в базе данных, оно должно быть обновлено из этой базы данных.

Затем я отображаю ContactName и ContactTitle объекта Customer, при этом ContactName должно быть отображено взятое из базы данных, а ContactTitle — измененное мною в сущностном объекте.

И последнее: я восстанавливая старые значения в базе данных, чтобы этот пример можно было запустить снова, и чтобы ни один последующий пример не пострадал.

Взглянем на результаты работы кода из листинга 16.33.

```
Customer's original name is John Steel, ContactTitle is Marketing Manager.
```

```
Executing SQL statement against database with ADO.NET ...
```

```
Database updated.
```

```
Customer's name before refresh is John Steel, ContactTitle is Chief Technology Officer.
```

```
Customer's name after refresh is Brad Radaker, ContactTitle is Chief Technology Officer.
```

```
Resetting data to original values.
```

```
Executing SQL statement against database with ADO.NET ...
```

```
Database updated.
```

Как видите, сущностный объект не знает об изменении ContactName на "Brad Radaker" в базе данных, пока я не вызвал метод Refresh, но как только я вызвал его, он об этом узнает.

Для примера вызова второго прототипа в листинге 16.34 я извлеку заказчиков из региона "WA", используя LINQ to SQL. Я выполню перечисление членов возвращенной последовательности объектов Customer и отобразжу их свойства CustomerID, Region и Country. Затем, используя ADO.NET, я обновлю поле Country у каждого заказчика в базе данных, чей регион "WA". В этой точке значение поля Country для этих заказчиков будет отличаться в базе данных от того, что есть в соответствующем свойстве сущностных объектов, которые были извлечены ранее. Я снова выполню перечисление последовательности извлеченных заказчиков — просто чтобы доказать, что сущностные объекты не имеют понятия об изменении поля Region в базе данных.

Далее я вызову операцию ToArray на последовательности объектов Customer, чтобы получить массив, содержащий объекты Customer.

Потом последний раз я выполню перечисление последовательности сущностных объектов Customer и отобразжу их свойства CustomerID, Region и Country, чтобы доказать, что поле Country на самом деле было обновлено по базе данных.

Конечно, мне придется также восстановить базу данных в исходное состояние, поэтому я использую ADO.NET, чтобы установить Country заказчика обратно в их начальное значение.

Код показан в листинге 16.34.

Листинг 16.34. Пример вызова второго прототипа метода Refresh

```
Northwind db = new Northwind(@"Data Source=\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = (from c in db.Customers
                                 where c.Region == "WA"
                                 select c);
```

```

Console.WriteLine("Entity objects before ADO.NET change and Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
        c.CustomerID, c.Region, c.Country);
}
Console.WriteLine("{0}Updating customers' country to United States in ADO.NET...",
    System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set Country = 'United States' where Region = 'WA'"));
Console.WriteLine("Customers' country updated.{0}", System.Environment.NewLine);
Console.WriteLine("Entity objects after ADO.NET change but before Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
        c.CustomerID, c.Region, c.Country);
}
Customer[] custArray = custs.ToArray();
Console.WriteLine("{0}Refreshing params array of customer entity objects ...",
    System.Environment.NewLine);
db.Refresh(RefreshMode.KeepChanges, custArray[0], custArray[1], custArray[2]);
Console.WriteLine("Params array of Customer entity objects refreshed.{0}",
    System.Environment.NewLine);
Console.WriteLine("Entity objects after ADO.NET change and Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
        c.CustomerID, c.Region, c.Country);
}
// Нужно вернуть измененные значения в исходное состояние,
// чтобы можно было запустить пример более одного раза.
Console.WriteLine("{0}Resetting data to original values.",
    System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set Country = 'USA' where Region = 'WA'"));

```

В предыдущем коде не происходит ничего интересного вплоть до момента вызова операции `ToArray`. Как только я получаю массив объектов `Customer`, то сразу же вызываю `RefreshMethod` и передаю `custArray[0]`, `custArray[1]` и `custArray[2]`.

Взглянем на результаты:

```

Entity objects before ADO.NET change and Refresh() call:
Customer LAZYK's region is WA, country is USA.
Customer TRAIH's region is WA, country is USA.
Customer WHITC's region is WA, country is USA.
Updating customers' country to United States in ADO.NET...
Executing SQL statement against database with ADO.NET ...
Database updated.
Customers' country updated.
Entity objects after ADO.NET change but before Refresh() call:
Customer LAZYK's region is WA, country is USA.
Customer TRAIH's region is WA, country is USA.
Customer WHITC's region is WA, country is USA.
Refreshing params array of customer entity objects ...
Params array of Customer entity objects refreshed.
Entity objects after ADO.NET change and Refresh() call:
Customer LAZYK's region is WA, country is United States.
Customer TRAIH's region is WA, country is United States.
Customer WHITC's region is WA, country is United States.

```

```
Resetting data to original values.
Executing SQL statement against database with ADO.NET ...
Database updated.
```

Как можно видеть из этого результата, проведенные мною изменения в поле Country в базе данных не отражаются в сущностном объекте Customer до тех пор, пока я не вызову метод Refresh.

В листинге 16.34 каждый сущностный объект, обновленный мною, был одного и того же типа — Customer. Для второго прототипа метода Refresh не обязательно, чтобы все переданные сущностные объекты были одного и того же типа. Я мог бы передать сущностные объекты разных типов. В случае листинга 16.34 на самом деле было бы проще передать последовательность сущностных объектов методу Refresh, потому что такая последовательность у меня была. К счастью, третий прототип метода Refresh позволяет вас передавать последовательность.

Итак, для примера третьего прототипа в листинге 16.35 я использую тот же базовый код, что и в листинге 16.34, но на этот раз передам последовательность извлеченных сущностных объектов Customer. Ниже показан код.

Листинг 16.35. Пример вызова третьего прототипа метода Refresh

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IEnumerable<Customer> custs = (from c in db.Customers
                                 where c.Region == "WA"
                                 select c);
Console.WriteLine("Entity objects before ADO.NET change and Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
                      c.CustomerID, c.Region, c.Country);
}
Console.WriteLine("{0}Updating customers' country to United States in ADO.NET...",
                  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set Country = 'United States' where Region = 'WA'"));
Console.WriteLine("Customers' country updated.{0}", System.Environment.NewLine);
Console.WriteLine("Entity objects after ADO.NET change but before Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
                      c.CustomerID, c.Region, c.Country);
}
Console.WriteLine("{0}Refreshing sequence of customer entity objects ...",
                  System.Environment.NewLine);
db.Refresh(RefreshMode.KeepChanges, custs);
Console.WriteLine("Sequence of Customer entity objects refreshed.{0}",
                  System.Environment.NewLine);
Console.WriteLine("Entity objects after ADO.NET change and Refresh() call:");
foreach (Customer c in custs)
{
    Console.WriteLine("Customer {0}'s region is {1}, country is {2}.",
                      c.CustomerID, c.Region, c.Country);
}
// Нужно вернуть измененные значения в исходное состояние,
// чтобы можно было запустить пример более одного раза.
Console.WriteLine("{0}Resetting data to original values.",
                  System.Environment.NewLine);
ExecuteStatementInDb(String.Format(
    @"update Customers set Country = 'USA' where Region = 'WA'"));
```

Код в листинге 16.35 тот же, что и в листинге 16.34, за исключением того, что при вызове метода Refresh я передаю ему последовательность custs. Взглянем на результат:

```

Entity objects before ADO.NET change and Refresh() call:
Customer LAZYK's region is WA, country is USA.
Customer TRAIH's region is WA, country is USA.
Customer WHITC's region is WA, country is USA.
Updating customers' country to United States in ADO.NET...
Executing SQL statement against database with ADO.NET ...
Database updated.
Customers' country updated.
Entity objects after ADO.NET change but before Refresh() call:
Customer LAZYK's region is WA, country is USA.
Customer TRAIH's region is WA, country is USA.
Customer WHITC's region is WA, country is USA.
Refreshing sequence of customer entity objects ...
Sequence of Customer entity objects refreshed.
Entity objects after ADO.NET change and Refresh() call:
Customer LAZYK's region is WA, country is United States.
Customer TRAIH's region is WA, country is United States.
Customer WHITC's region is WA, country is United States.
Resetting data to original values.
Executing SQL statement against database with ADO.NET ...
Database updated.

```

Как видите, несмотря на тот факт, что я изменил Country у извлеченных заказчиков на "United States" в базе данных, я не увидел изменений в сущностных объектах до тех пор, пока не вызвал метод Refresh.

Резюме

Я знаю, что понадобилось немало времени, чтобы достичь понимания всего того, что может сделать для вас класс DataContext. LINQ to SQL не тривиален, потому что предполагает понимание LINQ и понимание запросов данных и SQL. Из-за этого для овладения LINQ to SQL нужно узнать достаточно много, и большая часть этих знаний касается класса DataContext в его переплетении с сущностными классами; поэтому что-то познается в начале, а что-то — в конце.

Хотя в этой главе было представлено много информации, возможно, наиболее важные темы, поднятые здесь — это работа трех служб DataContext: отслеживание идентичности, отслеживание изменений и обработка изменений. Конечно, ни одна из этих служб ничего не стоит, если вы не можете даже создать экземпляр объекта DataContext или [Your]DataContext, так что конструкторы классов DataContext и [Your]DataContext не менее важны.

Помимо конструкторов DataContext и [Your]DataContext, скорее всего, наиболее часто вы будете использовать метод SubmitChanges, потому что этот метод вызывается для сохранения ваших изменений в базе данных.

Важно помнить, что когда вы пытаетесь сохранить данные в базе, иногда могут возникнуть конфликты параллельного доступа, сопровождаемые генерацией исключений. До сих пор в главах, посвященных LINQ to SQL, я не раз упоминал о конфликтах параллелизма, но пока не рассматривал их детально. Поэтому в следующей главе мы обратимся к детальному рассмотрению конфликтов параллельного доступа.

ГЛАВА 17

Конфликты параллельного доступа

Сколько раз вы уже слышали от меня, что вам следует обнаруживать конфликты параллельного доступа и разрешать их? В большинстве предыдущих глав о LINQ to SQL я упоминал о конфликтах параллельного доступа, но пока еще не рассматривал их настолько детально, как они того заслуживают. В этой главе я восполню это упущение.

Предварительные условия для запуска примеров

Для того чтобы запустить примеры этой главы, вам понадобится расширенная версия базы данных Northwind и сгенерированные для нее существенные классы. Прочтите и выполните требования, изложенные в разделе “Предварительные условия для запуска примеров” главы 12.

Некоторые общие методы

Дополнительно, чтобы запустить примеры этой главы, вам понадобятся некоторые общие методы, которые будут использованы в примерах. Прочтите и выполните то, что написано в разделе “Некоторые общие методы” главы 12.

Использование LINQ to SQL API

Чтобы запустить примеры этой главы, вам может понадобиться добавить соответствующие ссылки и директивы `using` к вашему проекту. Прочтите и выполните то, что сказано в разделе “Использование LINQ to SQL API” главы 12.

Конфликты параллелизма

Когда одно соединение с базой данных пытается обновить часть данных, которая была изменена в другом соединении после того, как запись была прочитана первым соединением, возникает конфликт параллельного доступа. То есть, если первый процесс читает данные, после чего те же данные читает второй процесс, и второй процесс обновляет те же данные до того, как это сможет сделать первый процесс, возникает конфликт, когда первый процесс попытается обновить эти данные. Также верно, что если первый процесс обновляет данные перед вторым процессом, то второй процесс столкнется с конфликтом параллельного доступа при попытке обновить данные. Если несколько соединений могут обращаться к базе данных и проводить изменения, то возникновение конфликтов — вопрос лишь времени и везения.

Когда возникает конфликт, приложение должно предпринять какие-то действия для его разрешения. Например, администратор Web-сайта может быть на странице, отображающей данные для обычного пользователя, которая позволяет администратору обновлять данные обычного пользователя. Если после того, как страница администратора прочитает пользовательские данные из базы данных, и обычный пользователь обратится к странице, отображающей его данные, и внесет изменение, то возникнет конфликт, когда администратор сохранит свои изменения в базе данных. Если же конфликт не возникнет, то изменения обычного пользователя будут перекрыты и потеряны. Может случиться иначе: изменения обычного пользователя будут сохранены, а изменения администратора — потеряны. Какое поведение следует считать правильным в каждом конкретном случае — представляет собой сложную проблему. Первый шаг — обнаружить ее. Второй шаг — разрешить.

Есть два базовых подхода к разрешению конфликтов параллельного доступа — оптимистический и пессимистический.

Оптимистический параллелизм

Как следует из названия, оптимистический подход к разрешению конфликтов параллельного доступа исходит из того, что в большинстве случаев конфликты не происходят. Поэтому никаких блокировок на данные не устанавливается во время их чтения из базы. Когда вдруг случится конфликт при попытке обновить одни и те же данные, тогда мы им и займемся. Оптимистическая обработка конфликтов параллелизма более сложна, чем пессимистическая, но работает лучше в современных приложениях с очень большим количеством пользователей. Представьте, насколько раздражало бы, если бы каждый раз, когда вы хотите взглянуть на лот на вашем любимом аукционном сайте, вы не могли бы этого сделать потому, что кто-то другой в этот момент просматривает тот же самый лот, и нужная вам запись блокирована, потому что этот кто-то может предложить свою цену за него? У вас надолго пропадет желание заглядывать на этот сайт.

LINQ to SQL использует оптимистический подход к обработке конфликтов параллелизма. К счастью, LINQ to SQL в состоянии обнаружить и разрешить конфликты параллельного доступа настолько просто, насколько это вообще возможно. Он даже предусматривает метод обработки разрешения вместо вас, если вам это подходит. Но я сказал “оптимистический” подход, а не безрассудный.

Обнаружение конфликтов

Как я уже упоминал, первый шаг — обнаружение конфликтов. В LINQ to SQL реализовано два подхода к обнаружению конфликтов параллелизма. Если специфицировано свойство `IsVersion` атрибута `Column` свойства сущностного класса, и оно имеет значение `true`, то значение этого свойства сущностного класса, и только этого свойства, будет использовано для определения возникновения конфликта параллельного доступа.

Если ни одно из свойств сущностного класса не имеет установленного в `true` свойства `IsVersion` атрибута `Column`, то LINQ to SQL позволяет вам контролировать, какие именно свойства сущностного класса участвуют в обнаружении конфликтов параллельного доступа через свойство `UpdateCheck` атрибута `Column`, специфицированного на отображаемом свойстве сущностного класса. Перечисление `UpdateCheck` предусматривает три возможных значения: `Never`, `Always` и `WhenChanged`.

`UpdateCheck`

Если свойство `UpdateCheck` атрибута для отображаемого свойства сущностного класса установлено в `UpdateCheck.Never`, то это свойство сущностного класса не будет участвовать в обнаружении конфликтов параллельного доступа. Если же оно установ-

лено в UpdateCheck.Always, то такое свойство сущностного класса всегда будет участвовать в разрешении конфликтов параллелизма, независимо от того, было ли изменено его значение с момента первоначального извлечения и помещения в кеш объекта DataContext. Если свойство UpdateCheck установлено в UpdateCheck.WhenChanged, то это свойство сущностного класса будет участвовать в проверке обновления, только если его значение было изменено с момента первоначальной загрузки в кеш объекта DataContext. Если свойство UpdateCheck атрибута не специфицировано, по умолчанию принимается UpdateCheck.Always.

В понимании того, как технически работает обнаружение конфликтов, может помочь понимание его текущей реализации. Когда вы вызываете метод SubmitChanges, процессор изменений генерирует необходимые операторы SQL, чтобы сохранить все изменения сущностных объектов в базе данных. Когда требуется обновить запись, то вместо того, чтобы специфицировать только первичный ключ в конструкции where для нахождения соответствующей записи для обновления, он специфицирует наряду с первичным ключом все столбцы, участвующие в обнаружении конфликта. Если свойство UpdateCheck атрибута сущностного класса установлено в UpdateCheck.Always, столбец, на который отображено это свойство сущностного класса, и его первоначальное значение будут всегда специфицированы в конструкции where. Если свойство UpdateCheck атрибута сущностного класса установлено в UpdateCheck.WhenChanged, то лишь текущее значение свойства сущностного объекта, которое было изменено по сравнению с его первоначальным значением, будет участвовать в процессе, и его первоначальное значение будет специфицировано в конструкции where. Если свойство UpdateCheck имеет значение UpdateCheck.Never, то столбец, на который отображено данное свойство сущностного класса, не будет специфицирован в конструкции where.

Например, предположим, что сущностный объект Customer специфицирует свойство UpdateCheck для CompanyName как UpdateCheck.Always, ContactName — как UpdateCheck.WhenChanged и ContactTitle — как UpdateCheck.Never. Если все эти три свойства сущностного класса были модифицированы в объекте заказчика, сгенерированный оператор SQL будет выглядеть следующим образом:

```
Update Customers
Set CompanyName = 'Art Sanders Park',
    ContactName = 'Samuel Arthur Sanders',
    ContactTitle = 'President'
Where CompanyName = 'Lonesome Pine Restaurant' AND
      ContactName = 'Fran Wilson' AND
      CustomerID = 'LONEP'
```

В этом примере значения столбцов в выражении where — первоначальные значения, прочитанные из базы данных при первом извлечении сущностного объекта, успешном завершении вызова метода SubmitChanges или же вызове метода Refresh.

Вы можете видеть, что поскольку свойство UpdateCheck атрибута свойства CompanyName специфицировано как UpdateCheck.Always, соответствующий ему столбец будет включен в конструкцию where, независимо от того, было ли это свойство изменено в сущностном объекте. А так как свойство UpdateCheck атрибута свойства ContactName специфицировано как UpdateCheck.WhenChanged, и значение этого свойства сущностного класса было изменено в сущностном объекте, оно также включается в выражение where. И поскольку свойство UpdateCheck атрибута свойства ContactTitle специфицировано как UpdateCheck.Never, соответствующий ему столбец не включен в выражение where, несмотря на тот факт, что значение этого свойства также было изменено.

При выполнении этого оператора SQL, если любое из значений свойств сущностного класса, специфицированных в части where, не будет соответствовать тому, что

есть в базе данных, то запись не будет найдена, а, следовательно, и не будет обновлена. Именно так обнаруживается конфликт параллельного доступа. В случае обнаружения конфликта генерируется исключение `ChangeConflictException`.

Мое объяснение того, как обнаруживаются конфликты, несколько запутано, но реализация от Microsoft не специфицирована и не так наглядна, как это было в ранних выпусках LINQ. В ранних версиях после выполнения оператора `update` должен был быть выполнен сгенерированный оператор `select`, содержащий проверку `@@ROWCOUNT`, возвращенного оператором `update`, что давало знать процессору обновлений, когда обновлялось ноль записей, а это свидетельствовало о конфликте. Вы, конечно, не можете полагаться на эту реализацию, поскольку она не специфицирована, но я предлагаю эту технику, чтобы вы знали, как минимум, один способ реализации обнаружения конфликтов параллелизма, если вам придется ею заниматься.

Чтобы увидеть, как именно выглядит сгенерированный оператор `update`, давайте рассмотрим код в листинге 17.1.

Листинг 17.1. Инициализация обновления в базе данных для того, чтобы увидеть, как обнаруживаются конфликты параллельного доступа

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.Log = Console.Out;
Customer cust = db.Customers.Where(c => c.CustomerID == "LONEP").SingleOrDefault();
string name = cust.ContactName; // чтобы восстановить позднее.
cust.ContactName = "Neo Anderson";
db.SubmitChanges();
// Восстановление базы данных.
cust.ContactName = name;
db.SubmitChanges();
```

Об этом запросе сказать особо нечего. Фактически единственное, что нужно здесь отметить — это вызов операции `SingleOrDefault` вместо `Single`, которую я использую обычно — просто, чтобы подстраживаться на случай, если запись не будет найдена. В этом случае я знаю, что запись будет найдена, но хочу напомнить вам, что вы должны всегда заботиться о том, чтобы код мог правильно справиться с подобными ситуациями.

Все, что меня действительно интересует — это сгенерированный оператор `update`. Посмотрим на результат:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
-- @p0: Input String (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
UPDATE [dbo].[Customers]
SET [ContactName] = @p11
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2) AND
([ContactTitle] = @p3) AND ([Address] = @p4) AND ([City] = @p5) AND ([Region] = @p6)
AND ([PostalCode] = @p7) AND ([Country] = @p8) AND ([Phone] = @p9) AND ([Fax] =
@p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- @p1: Input String (Size = 24; Prec = 0; Scale = 0) [Lonesome Pine Restaurant]
-- @p2: Input String (Size = 11; Prec = 0; Scale = 0) [Fran Wilson]
-- @p3: Input String (Size = 13; Prec = 0; Scale = 0) [Sales Manager]
-- @p4: Input String (Size = 18; Prec = 0; Scale = 0) [89 Chiaroscuro Rd.]
-- @p5: Input String (Size = 8; Prec = 0; Scale = 0) [Portland]
```

```
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [OR]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [97219]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9573]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9646]
-- @p11: Input String (Size = 12; Prec = 0; Scale = 0) [Neo Anderson]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
UPDATE [dbo].[Customers]
SET [ContactName] = @p11
WHERE ([CustomerID] = @p0) AND ([CompanyName] = @p1) AND ([ContactName] = @p2) AND
([ContactTitle] = @p3) AND ([Address] = @p4) AND ([City] = @p5) AND ([Region] = @p6)
AND ([PostalCode] = @p7) AND ([Country] = @p8) AND ([Phone] = @p9) AND ([Fax] =
@p10)
-- @p0: Input StringFixedLength (Size = 5; Prec = 0; Scale = 0) [LONEP]
-- @p1: Input String (Size = 24; Prec = 0; Scale = 0) [Lonesome Pine Restaurant]
-- @p2: Input String (Size = 12; Prec = 0; Scale = 0) [Neo Anderson]
-- @p3: Input String (Size = 13; Prec = 0; Scale = 0) [Sales Manager]
-- @p4: Input String (Size = 18; Prec = 0; Scale = 0) [89 Chiaroscuro Rd.]
-- @p5: Input String (Size = 8; Prec = 0; Scale = 0) [Portland]
-- @p6: Input String (Size = 2; Prec = 0; Scale = 0) [OR]
-- @p7: Input String (Size = 5; Prec = 0; Scale = 0) [97219]
-- @p8: Input String (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p9: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9573]
-- @p10: Input String (Size = 14; Prec = 0; Scale = 0) [(503) 555-9646]
-- @p11: Input String (Size = 11; Prec = 0; Scale = 0) [Fran Wilson]
-- Context: SqlProvider(Sql2005) Model: AttributedMetaModel Build: 3.5.20706.1
```

Обратите внимание, что в первом операторе update конструкция where указывает, что ContactName должно быть равно "Fran Wilson", т.е. первоначальному значению ContactName. Если какой-то другой процесс изменит ContactName после его прочтения, ни одна запись не будет соответствовать условию where, поэтому ни одна запись не будет обновлена.

Поскольку ни одно из свойств сущностного класса Customer не специфицирует свойство атрибута UpdateCheck, все они по умолчанию устанавливаются в UpdateCheck. Always, так что все отображенные свойства сущностного класса специфицированы в конструкции where этого оператора update.

SubmitChanges ()

Обнаружение конфликтов параллельного доступа происходит при вызове метода SubmitChanges (). Когда вы вызываете метод SubmitChanges, то имеете возможность специфицировать, нужно ли прерывать процесс сохранения изменений в базе данных в случае первого обнаружения конфликта, или же пытаться провести все изменения, накапливая конфликты. Вы управляете этим поведением посредством аргумента ConflictMode, который может быть передан методу SubmitChanges. Если вы передаете значение ConflictMode.FailOnFirstConflict, то, как следует из его названия, процесс будет прерван после первого же конфликта. Если же вы передаете ConflictMode.ContinueOnConflict, то процесс продолжит попытки провести все необходимые изменения, даже если случатся конфликты. Если вы предпочтете не указывать ConflictMode, то метод SubmitChanges по умолчанию примет ConflictMode.FailOnFirstConflict.

Независимо от специфициированного ConflictMode, если объемлющая транзакция не активна в данном контексте при вызове метода SubmitChanges, транзакция будет создана для всех попыток изменения базы данных, предпринимаемых в рамках данного вызова метода SubmitChanges. Если же активна объемлющая транзакция, то DataContext использует ее. При генерации исключения во время вызова метода SubmitChanges транзакция будет отменена (rolled back). Это значит, что даже измене-

ния не конфликтующих сущностных объектов, успешно помещенные в базу данных, также будут отменены.

ChangeConflictException

Если случается конфликт параллельного доступа, то независимо от того, установлен ли `ConflictMode` в `FailOnFirstConflict` или `ContinueOnConflict`, исключение `ChangeConflictException` будет сгенерировано.

Перехватывая исключение `ChangeConflictException`, вы можете обнаружить возникновение конфликта.

Разрешение конфликтов

Как только вы обнаружили конфликт параллельного доступа, перехватив `ChangeConflictException`, вашим следующим шагом, скорее всего, будет его разрешение. Вы можете предпочесть предпринять какое-то другое действие, но разрешение конфликтов — наиболее вероятный выход. Когда я впервые прочитал о том, что я должен разрешать конфликты, мне представился чудовищно сложный код, пытающийся анализировать, что нужно делать с каждой частью данных в каждой возможной ситуации. К счастью, LINQ to SQL также облегчает эту задачу, предоставляя в наше распоряжение метод `ResolveAll` и два метода `Resolve`.

RefreshMode

Когда мы действительно разрешаем конфликт, используя встроенную функциональность разрешения LINQ to SQL посредством вызова метода `ResolveAll` или `Resolve`, то контролируем способ разрешения конфликта, специфицируя режим `RefreshMode`. Его три допустимых значения — это `KeepChanges`, `KeepCurrentValues` и `OverwriteCurrentValues`. Эти опции контролируют то, какие данные остаются в свойствах сущностного объекта, когда `DataContext` выполняет разрешение конфликта.

Опция `RefreshMode.KeepChanges` сообщает методу `ResolveAll` или `Resolve` о том, что нужно загрузить изменения из базы данных в текущие значения свойств сущностного класса для каждого столбца, измененного с момента начальной загрузки данных, если только текущий пользователь также не изменил это свойство — в этом случае его значение сохраняется. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, перезагруженные измененные значения столбцов базы данных, значения свойств сущностного класса, измененные текущим пользователем.

Опция `RefreshMode.KeepCurrentValues` сообщает методу `ResolveAll` или `Resolve` о том, что нужно сохранить первоначальные значения свойств сущностного класса и изменения текущего пользователя, отклонив все изменения, проведенные в базе после начальной загрузки. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, измененные текущим пользователем значения свойств сущностного класса.

Опция `RefreshMode.OverwriteCurrentValues` сообщает методу `ResolveAll` или `Resolve` о том, что нужно загрузить изменения из базы данных для любого столбца, измененного с момента начальной загрузки данных, и отбросить изменения свойств сущностного класса, проведенные текущим пользователем. Порядок приоритетов предохранения данных, от низшего к высшему, такой: первоначальные значения свойств сущностного класса, затем — перезагруженные значения столбцов.

Подходы к разрешению конфликтов

Существует три подхода к разрешению конфликтов: простейший, легкий и ручной. Простейший подход заключается в простом вызове метода `ResolveAll` на коллекции

`DataContext.ChangeConflicts` с передачей `RefreshMode` и необязательного булевского значения, говорящего о том, нужно ли автоматически разрешать удаленные записи.

Автоматическое разрешение удаленных записей означает маркировку соответствующего удаленного сущностного объекта как успешно удаленного, даже когда оно не произошло из-за конфликта параллельного доступа, чтобы при следующем вызове метода `SubmitChanges` объект `DataContext` не пытался удалить снова записи базы данных, соответствующие удаленным сущностным объектам. По сути, мы говорим LINQ to SQL, чтобы он представил, будто бы они успешно удалены, потому что кто-то удалил их раньше — и порядок.

Легкий подход состоит в перечислении всех `ObjectChangeConflict` из коллекции `DataContext.ChangeConflicts` с вызовом метода `Resolve` на каждом из них.

Если же, однако, вам нужна какая-то специальная обработка, у вас всегда есть выбор обработать разрешение конфликта самостоятельно, перечислив элементы коллекции `ChangeConflicts` объекта `DataContext`, а затем перечислив все элементы коллекции `MemberConflicts` объекта `ObjectChangeConflict`, вызывая метод `Resolve` на каждом объекте `MemberChangeConflict` из этой коллекции. Даже при ручной обработке имеющиеся методы позволяют решить эту задачу достаточно легко.

`DataContext.ChangeConflicts.ResolveAll()`

Разрешение конфликтов не сложно. Вы просто перехватываете исключение `ChangeConflictException` и вызываете метод `ResolveAll` на коллекции `DataContext.ChangeConflicts`. Все что от вас требуется — это решить, какой режим `RefreshMode` использовать, и хотите ли вы автоматически разрешать конфликты удаленных записей.

Применение такого подхода вызовет одинаковое разрешение всех конфликтов на базе переданного `RefreshMode`. Если вам нужен более тонкий контроль при разрешении конфликтов, используйте один из несколько более сложных подходов, которые я опишу ниже.

В листинге 17.2 я разрешу конфликт, используя этот подход. Поскольку этот пример несколько сложен, я сопровожу его пояснениями.

Листинг 17.2. Пример разрешения конфликтов с помощью

`DataContext.ChangeConflicts.ResolveAll()`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
ExecuteStatementInDb(String.Format(
    @"update Customers
    set ContactName = 'Samuel Arthur Sanders'
    where CustomerID = 'LAZYK'"));
```

Я создаю `Northwind` `DataContext`, запрашиваю заказчика с помощью LINQ to SQL и провожу изменения в значении столбца `ContactName` извлеченного заказчика в базе данных посредством ADO.NET. Тем самым закладываю фундамент для потенциального конфликта параллельного доступа.

Теперь мне нужно провести изменение в моем сущностном объекте и попытаться сохранить его в базе данных.

```
cust.ContactTitle = "President";
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{}
```

Обратите внимание, что я поместил вызов метода `SubmitChanges` в блок `try/catch`. Чтобы правильно обнаруживать конфликты параллельного доступа, я перехватаю исключение `ChangeConflictException`. Мне нужно только вызвать метод `ResolveAll` и снова попытаться сохранить изменения.

```
db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
    cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
    Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
        cust.ContactName, cust.ContactTitle);
}
catch (ChangeConflictException)
{
    Console.WriteLine("Conflict again, aborting.");
}
}
```

В приведенном коде язываю метод `ResolveAll` и передаю `RefreshMode` со значением `KeepChanges`. Затем я вновьзываю метод `SubmitChanges`, помещенный в собственный блок `try/catch`. После этого опять запрашиваю заказчика из базы данных и отображаю `ContactName` и `ContactTitle` заказчика, чтобы доказать, что ни изменение ADO.NET, ни мое изменение LINQ to SQL не утеряны. Если этот вызов метода `SubmitChanges` сгенерирует исключение, я просто выдам сообщение об этом и прекращу дальнейшие попытки.

Все, что осталось сделать — это восстановить базу данных в исходное состояние, чтобы пример можно было запускать более одного раза.

```
// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
    @"update Customers
    set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
    where CustomerID = 'LAZYK'"));
```

Если присмотреться, то если не считать кода, вызвавшего конфликт, который вам обычно писать не придется, и кода восстановления базы данных в конце примера, который вам также писать не придется, то разрешение конфликтов параллелизма при таком подходе чрезвычайно просто. Вы помещаете вызов метода `SubmitChanges` в блок `try/catch`, перехватываете исключение `ChangeConflictException`, вызываете метод `ResolveAll` и повторяете вызов метода `SubmitChanges`. Вот и все. Взглянем на результат запуска кода из листинга 17.2.

```
Executing SQL statement against database with ADO.NET ...
Database updated.
ContactName = Samuel Arthur Sanders : ContactTitle = President
Executing SQL statement against database with ADO.NET ...
Database updated.
```

Как видно из результата, и изменение ADO.NET в `ContactName`, и изменение LINQ to SQL в `ContactTitle` сохранены в базе данных. Это очень простой подход к разрешению конфликтов параллельного доступа.

ObjectChangeConflict.Resolve()

Если разрешение всех конфликтов с тем же `RefreshMode` или `autoResolveDeletes` у вас не работает, вы можете выбрать подход с перечислением всех конфликтов из коллекции `DataContext.ChangeConflicts` и обрабатывать их индивидуально. Вы можете

обрабатывать каждый из них, вызывая на нем метод `Resolve`. Это позволит вам передавать разные `RefreshMode` и `autoResolveDeletes` для каждого конфликта.

Разрешение конфликтов на этом уровне сродни разрешению их на уровне сущностного объекта. Переданный `RefreshMode` будет применен к каждому свойству сущностного класса в конфликтующем сущностном объекте. Если вам нужен более тонкий контроль, чем позволяет такой подход, рассмотрите использование ручного подхода, о котором я расскажу далее.

В листинге 17.3 я продемонстрирую этот подход. Код будет таким же, как в листинге 17.2, только вызов метода `DataContext.ChangeConflicts.ResolveAll` будет заменен перечислением коллекции `ChangeConflicts`.

Листинг 17.3. Пример разрешения конфликтов с помощью `ObjectChangeConflict.Resolve()`

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
ExecuteStatementInDb(String.Format(
    @"update Customers
        set ContactName = 'Samuel Arthur Sanders'
        where CustomerID = 'LAZYK'");
cust.ContactTitle = "President";
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
    foreach (ObjectChangeConflict conflict in db.ChangeConflicts)
    {
        Console.WriteLine("Conflict occurred in customer {0}.",
            ((Customer)conflict.Object).CustomerID);
        Console.WriteLine("Calling Resolve ...");
        conflict.Resolve(RefreshMode.KeepChanges);
        Console.WriteLine("Conflict resolved.{0}", System.Environment.NewLine);
    }
    try
    {
        db.SubmitChanges(ConflictMode.ContinueOnConflict);
        cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
        Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
            cust.ContactName, cust.ContactTitle);
    }
    catch (ChangeConflictException)
    {
        Console.WriteLine("Conflict again, aborting.");
    }
}
// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
    @"update Customers
        set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
        where CustomerID = 'LAZYK'");
```

Обратите внимание, что вместо вызова метода `DataContext.ChangeConflicts.ResolveAll` я выполняю перечисление коллекции `ChangeConflicts` и вызываю метод `Resolve` на каждом объекте `ObjectChangeConflict` из этой коллекции. Затем, как и в предыдущем листинге, я снова вызываю метод `SubmitChanges`, снова запрашиваю за-

казчика и отображаю важнейшие свойства сущностного класса. Конечно, после этого я восстанавливаю базу данных в исходное состояние.

Вот результат работы листинга 17.3:

```
Executing SQL statement against database with ADO.NET ...
Database updated.
Conflict occurred in customer LAZYK.
Calling Resolve ...
Conflict resolved.
ContactName = Samuel Arthur Sanders : ContactTitle = President
Executing SQL statement against database with ADO.NET ...
Database updated.
```

Все работает, как мне и хотелось. В реально рабочем коде вы можете пожелать запустить цикл с вызовами метода `SubmitChanges` и разрешением конфликта — просто чтобы противостоять невезению, связанному с появлением дополнительных конфликтов при таком редком стечении обстоятельств. Если вы сделаете это, не забудьте как-то ограничить количество итераций цикла, чтобы предотвратить бесконечное его выполнение в случае, когда в системе возникнут серьезные неполадки.

MemberChangeConflict.Resolve()

При первом подходе язываю метод для разрешения всех конфликтов одним способом. Это — простейший способ разрешения конфликтов. При втором подходе язываю метод для разрешения конфликта для единственного конфликтующего объекта. Это обеспечивает гибкость для разрешения конфликтов с каждым сущностным объектом по-своему. Это — легкий способ. Что осталось? Остался подход к разрешению конфликтов вручную.

Пусть мое объяснение не пугает вас. Даже при нормальном подходе обнаружение конфликтов параллелизма вероятно, проще, чем вы могли бы ожидать. Принятие такого подхода позволит вам применять разные значения `RefreshMode` для индивидуальных свойств сущностного объекта.

Подобно второму подходу к разрешению конфликтов, я выполню перечисление объектов `ObjectChangeConflict` из коллекции `DataContext.ChangeConflicts`. Но вместо вызова метода `Resolve` на каждом объекте `ObjectChangeConflict` я выполню перечисление коллекции `MemberConflicts` и вызову метод `Resolve` каждого объекта `MemberChangeConflict`.

На этом уровне объект `MemberChangeConflict` принадлежит к определенному свойству сущностного класса из конфликтующего сущностного объекта. Это позволяет вам отступать от общего `RefreshMode` для любого свойства сущностного класса по вашему усмотрению.

Метод `Resolve` позволяет вам передавать либо `RefreshMode`, либо действительное значение, которое должно сменить текущее значение. Это обеспечивает замечательную гибкость.

Для примера ручного разрешения конфликтов в листинге 17.4 предположим, что существует требование, что если случится конфликт со столбцом `ContactName` в базе данных, код должен оставить значение базы как есть, но любые другие столбцы в записи могут быть обновлены.

Чтобы реализовать это, я использую тот же базовый код, что и в листинге 17.3, но вместо вызова метода `Resolve` на объекте `ObjectChangeConflict` выполню перечисление членов коллекции `MemberConflicts` каждого объекта. Затем для каждого объекта `MemberConflict` из этой коллекции, если свойство сущностного объекта, вызвавшего конфликт — это `ContactName`, то я поддержу значение в базе данных, передав `RefreshMode`, равное `RefreshMode.OverwriteCurrentValues`, методу `Resolve`.

Если же конфликтующее свойство сущностного объекта — не ContactName, я поддержу новое значение, передав методу Resolve значение RefreshMode, равное RefreshMode.KeepChanges.

Также, чтобы сделать пример более интересным, когда я обновляю базу данных с ADO.NET, чтобы создать конфликтную ситуацию, я также обновлю столбец ContactTitle. Это заставит конфликтовать два свойства сущностного объекта. Одно — ContactName — должно быть обработано так, чтобы сохранилось значение из базы данных. Второе — ContactTitle — должно быть обработано так, чтобы сохранилось значение, установленное LINQ to SQL. Взглянем на код в листинге 17.4.

Листинг 17.4. Пример ручного разрешения конфликтов

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
Customer cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
ExecuteStatementInDb(String.Format(
    @"update Customers
        set ContactName = 'Samuel Arthur Sanders',
            ContactTitle = 'CEO'
        where CustomerID = 'LAZYK'");
cust.ContactName = "Viola Sanders";
cust.ContactTitle = "President";
try
{
    db.SubmitChanges(ConflictMode.ContinueOnConflict);
}
catch (ChangeConflictException)
{
    foreach (ObjectChangeConflict conflict in db.ChangeConflicts)
    {
        Console.WriteLine("Conflict occurred in customer {0}.",
            ((Customer)conflict.Object).CustomerID);
        foreach (MemberChangeConflict memberConflict in conflict.MemberConflicts)
        {
            Console.WriteLine("Calling Resolve for {0} ...",
                memberConflict.Member.Name);
            if (memberConflict.Member.Name.Equals("ContactName"))
            {
                memberConflict.Resolve(RefreshMode.OverwriteCurrentValues);
            }
            else
            {
                memberConflict.Resolve(RefreshMode.KeepChanges);
            }
            Console.WriteLine("Conflict resolved.{0}", System.Environment.NewLine);
        }
    }
    try
    {
        db.SubmitChanges(ConflictMode.ContinueOnConflict);
        cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
        Console.WriteLine("ContactName = {0} : ContactTitle = {1}",
            cust.ContactName, cust.ContactTitle);
    }
    catch (ChangeConflictException)
    {
        Console.WriteLine("Conflict again, aborting.");
    }
}

```

```
// Сбросить базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
    @"update Customers
        set ContactName = 'John Steel', ContactTitle = 'Marketing Manager'
        where CustomerID = 'LAZYK'"));

```

Одним из существенных изменений является то, что я также обновляю ContactTitle через ADO.NET. Это приводит к конфликту в двух свойствах объекта при вызове SubmitChanges. Затем вместо вызова метода Resolve на объекте ObjectChangeConflict я перечисляю члены коллекции MemberConflicts, проверяя каждое свойство существующего объекта. Если текущее свойство — это ContactName, то я вызываю метод Resolve с RefreshMode, установленным в RefreshMode.KeepChanges, чтобы сохранить значение, выставленное моим кодом LINQ to SQL.

Знаю, что вы не можете дождаться результата. Взглянем на этот результат работы кода в листинге 17.4.

```
Executing SQL statement against database with ADO.NET ...
Database updated.
Conflict occurred in customer LAZYK.
Calling Resolve for ContactName ...
Conflict resolved.
Calling Resolve for ContactTitle ...
Conflict resolved.
ContactName = Samuel Arthur Sanders : ContactTitle = President
Executing SQL statement against database with ADO.NET ...
Database updated.
```

Здесь вы можете видеть, оба свойства — ContactName и ContactTitle — конфликтовали, и эти конфликты были разрешены. Также, просматривая вывод свойств ContactName и ContactTitle в конце, вы можете видеть, что значение из базы данных было сохранено для свойства ContactName, но проигнорировано для свойства ContactTitle, которое сохранило значение, установленное кодом LINQ to SQL. Именно этого я и хотел добиться.

Действительный код, обрабатывающий разрешение конфликта вручную не так плох. Но, конечно, все эти усилия оправданы только для специализированного разрешения конфликтов.

Пессимистический параллелизм

Как следует из названия, пессимистический параллелизм предполагает худшее, и вы можете просто рассчитывать на тот факт, что запись, которую вы читаете, вызовет конфликт на момент ее обновления. К счастью, у нас есть возможность справиться и с этим. Для этого нужно просто поместить чтение и обновление в базе данных внутрь одной транзакции.

При пессимистическом подходе к параллелизму нет конфликтов, которые нужно разрешать, потому что база данных блокирована вашей транзакцией, так что никто другой не сможет модифицировать ее у вас за спиной.

Чтобы проверить это, я создам объект TransactionScope и получу сущностный объект для заказчика LAZYK. Затем я создам другой объект TransactionScope со свойством TransactionScopeOption, равным RequiresNew. Я делаю так, что код ADO.NET не участвует в объемлющей транзакции, принадлежащей ранее созданному объекту TransactionScope. После этого я попытаюсь обновить ту же запись в базе данных с помощью ADO.NET. Но поскольку уже существует открытая транзакция, блокирующая базу данных, оператор обновления ADO.NET будет блокирован и в конечном итоге отме-

иен по таймауту. Далее я обновлю свойство ContactName сущностного объекта, вызову метод SubmitChanges, запрошу снова запись заказчика для отображения ContactName, чтобы доказать, что оно было обновлено LINQ to SQL, и завершу транзакцию.

На заметку! Чтобы следующий пример скомпилировался, потребуется добавить ссылку на сборку System.Transactions.dll к вашему проекту.

Листинг 17.5 содержит код описанного примера.

Листинг 17.5. Пример пессимистического параллелизма

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
using (System.Transactions.TransactionScope transaction =
    new System.Transactions.TransactionScope())
{
    Customer cust =
    db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
    try
    {
        Console.WriteLine("Let's try to update LAZYK's ContactName with ADO.NET.");
        Console.WriteLine(" Please be patient, we have to wait for timeout ...");

        using (System.Transactions.TransactionScope t2 =
            new System.Transactions.TransactionScope(
                System.Transactions.TransactionScopeOption.RequiresNew))
        {
            ExecuteStatementInDb(String.Format(
                @"update Customers
                    set ContactName = 'Samuel Arthur Sanders'
                    where CustomerID = 'LAZYK'");
            t2.Complete();
        }
        Console.WriteLine("LAZYK's ContactName updated.{0}",
            System.Environment.NewLine);
    }
    catch (Exception ex)
    {
        Console.WriteLine(
            "Exception occurred trying to update LAZYK with ADO.NET:{0} {1}{0}",
            System.Environment.NewLine, ex.Message);
    }
    cust.ContactName = "Viola Sanders";
    db.SubmitChanges();
    cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
    Console.WriteLine("Customer Contact Name: {0}", cust.ContactName);

    transaction.Complete();
}
// Вернуть базу данных в исходное состояние.
ExecuteStatementInDb(String.Format(
    @"update Customers
        set ContactName = 'John Steel',
        ContactTitle = 'Marketing Manager'
        where CustomerID = 'LAZYK')");

```

Совет. Если вы получаете исключение типа "MSDTC on server '[server]\SQLEXPRESS' is unavailable" ("Служба MSDTC на сервере '[сервер]\SQLEXPRESS' не доступна") при работе с любым примером, использующим объект TransactionScope, убедитесь, что запущена служба по имени Distributed Transaction Coordinator (распределенный координатор транзакций).

Этот код не так сложен, как может показаться на первый взгляд. Первое, что я делаю — это создание объекта TransactionScope. Теперь я использую пессимистический подход к параллелизму, предотвращающий модификацию моих данных ком-либо. Затем я запрашиваю моего заказчика, используя LINQ to SQL. Далее создаю другой объект TransactionScope, чтобы помешать коду ADO.NET, который я собираюсь вызывать, участвовать в транзакции моего первоначального объекта TransactionScope. После создания второго объекта TransactionScope я пытаюсь обновить заказчика в базе данных, используя ADO.NET. Код ADO.NET не будет иметь возможности выполнить обновление из-за моей начальной транзакции, в результате чего будет сгенерировано исключение таймаута. Затем я изменяю ContactName заказчика, сохраняю изменение в базе данных, вызывая метод SubmitChanges, запрашиваю заказчика снова и отображаю его ContactName, чтобы доказать, что изменение сохранено. Далее я завершаю исходную транзакцию, вызывая на ней метод Complete.

Конечно, как всегда, в конце кода я восстанавливую состояние базы данных. Вот результат запуска кода из листинга 17.5:

```
Let's try to update LAZYK's ContactName with ADO.NET.
Please be patient, we have to wait for timeout ...
Executing SQL statement against database with ADO.NET ...
Exception occurred trying to update LAZYK with ADO.NET:
Timeout expired. The timeout period elapsed prior to completion of the operation
or the server is not responding.
The statement has been terminated.
Customer Contact Name: Viola Sanders
Executing SQL statement against database with ADO.NET ...
Database updated.
```

Обратите внимание, что когда я пытаюсь обновить базу данных посредством ADO.NET, генерируется исключение таймаута. Пусть вас не вводит в заблуждение отложенное выполнение запроса. Помните, что многие из операций LINQ являются отложенными. В случае этого примера мой запрос LINQ to SQL вызывает операцию SingleOrDefault, так что запрос не является отложенным, что требует объявления запроса внутри контекста объекта TransactionScope. Если бы я не вызывал операцию SingleOrDefault, этот запрос мог быть объявлен перед созданием объекта TransactionScope, если действительное выполнение запроса происходило бы внутри контекста TransactionScope. Таким образом, я мог бы просто заставить запрос LINQ вернуть последовательность I Enumerable<T> до создания объекта TransactionScope, и затем внутри контекста объекта TransactionScope вызвать операцию SingleOrDefault на возвращенной последовательности, возвращая единственный объект Customer, соответствующий запросу.

При использовании этого подхода вы всегда должны оценивать, насколько много работы выполняете внутри контекста объекта TransactionScope, потому что на все это время база данных будет блокирована.

На заметку! При отладке этого примера вы можете столкнуться с ситуацией, когда объект TransactionScope прервет выполнение по таймауту.

Альтернативный подход для средних звеньев и серверов

Существует и альтернативный подход для обработки конфликтов параллельного доступа, когда они случаются в среднем звене или на сервере. Иногда, когда возникают такие конфликты, может быть проще только создать новый `DataContext`, применить изменения и вызвать `SubmitChanges`.

Рассмотрим пример Web-приложения ASP.NET. Из-за характера коммуникаций между клиентом-браузером и Web-сервером, предполагающим работу без установки соединения, вы с успехом можете создавать новый `DataContext` при всякой отправке HTTP на Web-сервер, когда требуется выполнить запрос LINQ to SQL. Помните, что поскольку прочтенные из базы данных считаются немедленно устаревающими, будет не слишком хорошей идеей удерживать объект `DataContext` открытым в течение длительного времени, если вы намерены проводить изменения.

Когда пользователь впервые обращается к Web-странице и для него извлекаются данные, нет особого смысла "подвешивать" объект `DataContext` в ожидании обратной отправки клиента, пытающейся изменить данные. `DataContext` все равно не будет доступен в процессе этого ожидания, если только не сохранять его каким-то образом между подключениями, например, в состоянии сеанса. Но даже если он будет доступен, задержки между соединениями могут быть очень длинными. Чем дольше вы ожидаете между первоначальным чтением базы данных для отображения страницы и попыткой обновления этой базы данных в последующей обратной отправке, тем более устареют ваши данные. Вместо того чтобы пытаться удерживать `DataContext` в таком сценарии, может быть более оправдано создавать новый `DataContext` при каждой обратной отправке, когда необходимо сохранить данные. Если это так, и возникает конфликт параллельного доступа, не будет большого вреда от создания еще одного `DataContext`, повторного применения изменений и нового вызова метода `SubmitChanges`. И поскольку задержка между первоначальным чтением данных при обратной отправке, применением изменений и вызовом метода `SubmitChanges` будет очень короткой, маловероятно, что вы столкнетесь с конфликтом параллельного доступа при первой попытке, и гораздо меньше — при второй.

Если вы решили использовать такой подход, при обратной отправке, после конструирования нового `DataContext`, вы можете извлечь необходимый сущностный объект, как я только что описал, или же поступить иначе. Вместо извлечения сущностного объекта вы можете создать новый сущностный объект, наполнить необходимые его свойства соответствующими значениями и присоединить их к соответствующей таблице, применив метод `Attach` объекта `Table<T>`. В этой точке все будет выглядеть так, будто сущностный объект был извлечен из базы данных, несмотря на тот факт, что не каждое поле этого объекта могло быть заполнено значениями.

Прежде, чем прикреплять сущностный объект к `Table<T>`, вы должны установить необходимые свойства сущностного класса в соответствующие значения. Это не значит, что вы должны запрашивать базу данных, чтобы получить эти значения; они могут поступать откуда угодно, например, из другого звена. Необходимые свойства сущностного класса включают все, составляющие первичный ключ или устанавливающие идентичность, все свойства, которые вы собираетесь изменить, а также все свойства, участвующие в проверке обновления. Вы должны включить свойства сущностного класса, устанавливающие идентичность, чтобы `DataContext` мог правильно отследить идентичность объекта сущностного класса. Вы должны включить все свойства, которые собираетесь изменить, чтобы они могли быть обновлены, и правильно работало обнаружение конфликтов параллельного доступа. Также вы должны включить все свойства сущностного класса, участвующие в проверке обновления, для обнаружения конфликтов параллельного доступа. Если сущностный класс имеет свойство, специфицирующее

свойство `IsVersion` атрибута `Column` со значением `true`, такое свойство класса должно быть установлено перед вызовом метода `Attach`.

Посмотрим в листинге 17.6, как это делается.

Листинг 17.6. Пример использования `Attach()` для подключения вновь созданного сущностного объекта

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
// Создать сущностный объект.
Console.WriteLine("Constructing an empty Customer object.");
Customer cust = new Customer();

// Сначала должны быть установлены все поля, определяющие идентичность.
Console.WriteLine("Setting the primary keys.");
cust.CustomerID = "LAZYK";

// Затем устанавливаем каждое изменяемое поле.
Console.WriteLine("Setting the fields I will change.");
cust.ContactName = "John Steel";

// И, наконец, устанавливаем все поля, участвующие в проверке обновления.
// К сожалению, для сущностного класса Customer это будут все поля.
Console.WriteLine("Setting all fields participating in update check.");
cust.CompanyName = "Lazy K Kountry Store";
cust.ContactTitle = "Marketing Manager";
cust.Address = "12 Orchestra Terrace";
cust.City = "Walla Walla";
cust.Region = "WA";
cust.PostalCode = "99362";
cust.Country = "USA";
cust.Phone = "(509) 555-7969";
cust.Fax = "(509) 555-6221";

// Теперь присоединимся к Table<T> Customers;
Console.WriteLine("Attaching to the Customers Table<Customer>.");
db.Customers.Attach(cust);

// В этой точке мы можем провести наши изменения и вызвать SubmitChanges().
Console.WriteLine("Making my changes and calling SubmitChanges().");
cust.ContactName = "Vickey Rattz";
db.SubmitChanges();
cust = db.Customers.Where(c => c.CustomerID == "LAZYK").SingleOrDefault();
Console.WriteLine("ContactName in database = {0}", cust.ContactName);
Console.WriteLine("Restoring changes and calling SubmitChanges().");
cust.ContactName = "John Steel";
db.SubmitChanges();
```

Как видите, я установил свойства класса, составляющие первичный ключ, свойства класса, которые собираюсь изменять, и свойства класса, участвующие в проверке обновления. Как я уже упоминал, я также должен установить эти свойства в соответствующие значения. Это не значит, однако, что я должен запросить базу данных. Возможно, я храню нужные значения в скрытых переменных или в состоянии представления (*view state*), или же они были получена от другого звена системы. Затем я вызываю метод `Attach` на `Customers Table<Customer>`. Далее провожу мои изменения и, наконец, вызываю метод `SubmitChanges`. После этого я запрашиваю моего заказчика из базы данных и отображаю его `ContactName`, чтобы доказать, что его значение было изменено в базе данных. И, наконец, как всегда, восстанавливаю базу данных в ее начальное состояние. Посмотрим на вывод кода из листинга 17.6.

```
Constructing an empty Customer object.  
Setting the primary keys.  
Setting the fields I will change.  
Setting all fields participating in update check.  
Attaching to the Customers Table<Customer>.  
Making my changes and calling SubmitChanges().  
ContactName in database = Vickey Rattz  
Restoring changes and calling SubmitChanges().
```

Вставка и удаление объектов сущностного класса не требует такого подхода. Вы можете просто вставлять или удалять объект сущностного класса перед вызовом метода `SubmitChanges`. Читайте подразделы “Вставки” и “Удаления” в разделе “Стандартные операции для баз данных” главы 14.

Резюме

Мы прошли изрядный путь. Я не раз упоминал об обнаружении конфликтов параллельного доступа и их разрешении в предыдущих главах, посвященных LINQ to SQL. И пришла пора предоставить вам все необходимое для того, чтобы с ними справляться.

Я сам был весьма впечатлен тем, насколько просто в LINQ to SQL осуществляется обнаружение и разрешение конфликтов параллельного доступа, и надеюсь, что вы тоже. Я надеюсь, что вы достигли внутреннего успокоения, изучив эту часто пугающую тему.

Мы почти закончили наше странствие по LINQ to SQL. В следующей, завершающей главе я попытаюсь дополнить повествование о LINQ to SQL некоторой разнородной информацией, для которой не нашлось места в предыдущих главах.

ГЛАВА 18

Дополнительные возможности SQL

В этой завершающей главе, посвященной LINQ to SQL, мы обсудим только несколько разных тем. Среди них: список представлений базы данных, за ней — наследование сущностных классов и, наконец, я хотел еще немного поговорить о транзакциях.

Предварительные условия для запуска примеров

Для того чтобы запускать примеры из этой главы, вам понадобится расширенная версия базы данных Northwind и сгенерированные для нее сущностные классы. Прочтите и выполните то, что описано в разделе “Предварительные условия для запуска примеров” главы 12.

Использование LINQ to SQL API

Чтобы запускать примеры этой главы, вам может понадобиться добавить соответствующие ссылки и директивы `using` к вашему проекту. Прочтите и выполните то, что изложено в разделе “Использование LINQ to SQL API” главы 12.

Использование LINQ to XML API

Некоторые примеры этой главы требуют дополнительной директивы `using` для пространства имен `System.Xml.Linq`.

Представления базы данных

Когда я генерировал сущностные классы для базы данных Northwind в главе 12, то специфицировал опцию `/views`, чтобы получить сущностные классы, отображаемые на представления (`views`) базы данных, но пока ничего не говорил о том, как выполнять запросы к ним. Инструменты генерации сущностных классов — `SQLMetal` и `Object Relational Designer` — генерируют свойство `Table<T>` в классе `[Your]DataContext` для каждого представления базы данных и создают соответствующий класс `T`. Вы опрашиваете их подобно таблицам. Вообще они ведут себя подобно таблицам, отличаясь только тем, что доступны лишь для чтения.

Поскольку сущностные классы, сгенерированные для представлений, не содержат свойств сущностных классов, отображаемых на первичные ключи, они доступны только для чтения. Если вы вспомните, что без первичных ключей `DataContext` не имеет эффективной возможности обеспечить отслеживание идентичности, это очевидно имеет смысл.

Например, база данных Northwind включает представление, которое называется Category Sales for 1997. В связи с этим SQLMetal сгенерировал общедоступное (public) свойство по имени CategorySalesFor1997s.

Общедоступное свойство для представления базы данных

```
public System.Data.Linq.Table<CategorySalesFor1997> CategorySalesFor1997s
{
    get
    {
        return this.GetTable<CategorySalesFor1997>();
    }
}
```

SQLMetal также сгенерировал для меня сущностный класс CategorySalesFor1997. Посмотрим, как опрашивается представление базы данных в листинге 18.1.

Листинг 18.1. Запрос к представлению базы данных

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
IQueryable<CategorySalesFor1997> seq = from c in db.CategorySalesFor1997s
                                         where c.CategorySales > (decimal)100000.00
                                         orderby c.CategorySales descending
                                         select c;
foreach (CategorySalesFor1997 c in seq)
{
    Console.WriteLine("{0} : {1:C}", c.CategoryName, c.CategorySales);
}
```

Обратите внимание, что в листинге 18.1 я запрашиваю представление точно так же, как таблицу. Посмотрим на результат:

```
Dairy Products : $114,749.78
Beverages : $102,074.31
```

Как я упоминал, представления доступны только для чтения. В листинге 18.2 я пытаюсь вставить запись в представление.

Листинг 18.2. Неудачная попытка вставить запись в представление

```
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
db.CategorySalesFor1997s.InsertOnSubmit(
    new CategorySalesFor1997
    { CategoryName = "Legumes", CategorySales = (decimal) 79043.92 });
```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод InsertOnSubmit, вызванный в предыдущем коде, назывался Add.

Обратите внимание, что в листинге 18.2 даже не предпринимается попытка вызвать метод SubmitChanges. Это потому, что я знаю, что код не сможет сделать этого без генерации исключения. Посмотрим на результат:

```
Unhandled Exception: System.InvalidOperationException: Can't perform Create, Update or Delete operations on 'Table(CategorySalesFor1997)' because it is read-only.
```

Необработанное исключение: System.InvalidOperationException: Не удается выполнить операции создания, обновления и удаления на 'Table(CategorySalesFor1997)', потому что она предназначена только для чтения.

...

Позвольте мне сделать одно предупреждение. Хотя методы `InsertOnSubmit` и `DeleteOnSubmit` генерируют исключение при вызове `Table<T>`, отображенной на представление базы данных, ничто не помешает вам внести изменения в свойство сущностного объекта. Вы можете изменить значение свойства и даже вызвать метод `SubmitChanges` без генерации исключений, но изменение в свойстве сущностного объекта не будет сохранено в базе данных.

Наследование сущностных классов

До сих пор в нашем разговоре о LINQ to SQL всегда присутствовал единственный сущностный класс, отображаемый на единственную таблицу, для которой имелся такой отображаемый класс. Поэтому отображение между сущностными классами и таблицами до сих пор строилось по схеме “один к одному”.

Внимание! Пример, используемый в этом разделе, создает модель данных, содержащую классы `Square` и `Rectangle`. Рассуждая с точки зрения геометрии, квадрат является прямоугольником, но прямоугольник не всегда является квадратом. Однако в модели данных, созданной для этого примера, справедливо обратное отношение. Эта модель классов определяет прямоугольник как наследник квадрата. Поэтому прямоугольник — есть квадрат, но квадрат — не обязательно прямоугольник. Обоснование этого изложено в тексте.

LINQ to SQL также предлагает альтернативу этому, известную как наследование сущностных классов. Наследование сущностных классов позволяет отобразить иерархию классов на единственную таблицу базы данных. Для этой единственной таблицы должен существовать базовый сущностный класс с соответствующими атрибутами отображения класса на таблицу базы данных. Этот базовый класс будет содержать все свойства, общие для всех классов в иерархии, унаследованных от базового класса, в то время как производные классы будут содержать лишь свойства, специфичные для этого производного класса, как это принято в любой объектной модели. Ниже приведен пример базового сущностного класса без отображенных производных классов.

Мой базовый сущностный класс без отображенных сущностных классов

```
[Table]
public class Shape
{
    [Column(IsPrimaryKey = true, IsDbGenerated = true,
        DbType = "Int NOT NULL IDENTITY")]
    public int Id;

    [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
    public string ShapeCode;

    [Column(DbType = "Int")]
    public int StartingX;

    [Column(DbType = "Int")]
    public int StartingY;
}
```

Как видите, я специфицировал атрибут `Table`, и поскольку свойство `Name` этого атрибута было задано, базовый сущностный класс отображается на одноименную с классом таблицу, т.е. на таблицу `Shape`. Не беспокойтесь, что у нас в данный момент нет таблицы `Shape`. Я использую метод `CreateDatabase` объекта `DataContext`, чтобы создать базу данных. К этому моменту никакие производные классы не отображены.

Позднее я вернусь к этому базовому сущностному классу, чтобы отобразить некоторые производные классы.

Идея, положенная в основу наследования сущностных классов, заключается в том, что единственная таблица базы данных — Shape — содержит столбец, значение которого определяет, объект какого именно сущностного класса должен быть сконструирован из записи таблицы, когда он будет извлечен LINQ to SQL. Этот столбец известен как столбец-дискриминатор, и специфицируется свойством `IsDiscriminator` атрибута `Column`.

Значение столбца дискриминатора известно под названием значения дискриминатора или кода дискриминатора. При отображении вашего базового сущностного класса на таблицу базы данных в дополнение к атрибуту `Table` вы специфицируете атрибуты `InheritanceMapping` для отображения кодов дискриминатора на классы-наследники базового сущностного класса. Но на этот раз в предыдущем классе `Shape` никакое наследование не отображается. Обратите внимание, что у меня есть несколько общедоступных членов, каждый отображен на столбец базы данных и специфицированы типы столбцов базы данных. Указание типов столбцов базы данных в моем случае необходимо, потому что позднее я буду вызывать метод `CreateDatabase`, а чтобы сделать это, я должен знать соответствующий тип. Также заметьте, что для члена `ShapeCode` я специфицировал свойство `IsDiscriminator` атрибута со значением `true`, тем самым создав столбец дискриминатора. Это значит, что столбец базы данных `ShapeCode` будет диктовать тип сущностного класса, используемый для конструирования сущностного объекта из каждой записи.

В этом классе у меня есть члены для `Id`, `ShapeCode` и начальных координат `X` и `Y` для фигуры на экране. Пока это единственные члены, наличие которых можно предвидеть в каждой фигуре.

Затем вы можете создать иерархию классов фигур, наследуя новые классы от этого базового. Производные классы должны наследоваться от базового сущностного класса. Производные классы не будут специфицировать атрибут `Table`, но будут специфицировать атрибуты `Column` для каждого общедоступного члена, который будет отображен на базу данных. Ниже представлены мои производные сущностные классы.

Мои производные сущностные классы

```
public class Square : Shape
{
    [Column(DBType = "Int")]
    public int Width;
}
public class Rectangle : Square
{
    [Column(DBType = "Int")]
    public int Length;
}
```

Для начала, рассматривая этот пример, вы должны забыть геометрическое определение квадрата и прямоугольника; т.е., рассуждая геометрически, квадрат есть прямоугольник, но прямоугольник — не обязательно квадрат. В этом примере с наследованием сущностных классов, поскольку стороны квадрата должны быть равны, необходимо значение только одного измерения — ширины (`width`). В этом смысле, с точки зрения наследования классов, прямоугольник есть квадрат, но квадрат — не есть прямоугольник. Хотя это противоположно геометрическому определению, оно вполне укладывается в мою модель наследования сущностных классов.

Общедоступные члены каждого из этих классов являются членами, специфичными для каждого класса. Например, поскольку `Square` нужна ширина, он имеет свойство

Width. Поскольку Rectangle наследуется от Square, в дополнение к унаследованному свойству Width ему понадобится свойство Length (длина).

Теперь у меня есть производные классы. Единственное, что пропущено — это отображение значений дискриминатора на базовый и производные сущностные классы. После добавления необходимых атрибутов InheritanceMapping, мой базовый класс будет выглядеть так, как показано ниже.

Мой базовый сущностный класс с отображениями производных классов

```
[Table]
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
[InheritanceMapping(Code = "S", Type = typeof(Square))]
[InheritanceMapping(Code = "R", Type = typeof(Rectangle))]
public class Shape
{
    [Column(IsPrimaryKey = true, IsDbGenerated = true,
        DbType = "Int NOT NULL IDENTITY")]
    public int Id;

    [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
    public string ShapeCode;

    [Column(DbType = "Int")]
    public int StartingX;

    [Column(DbType = "Int")]
    public int StartingY;
}
```

Добавленные отображения связывают различные значения столбца дискриминатора с сущностными классами. Поскольку столбец ShapeCode является столбцом дискриминатора, если запись имеет значение "G" в этом столбце, то из этой записи конструируется объект класса Shape. Если же запись имеет значение "S" в столбце ShapeCode, то из нее конструируется объект класса Square. А если запись имеет значение "R" в этом столбце, то из этой записи конструируется объект Rectangle.

Вдобавок почти всегда должны быть отображения по умолчанию, когда значение столбца дискриминатора не соответствует ни одному из значений, отображенных на сущностный класс. Какое именно отображение принято по умолчанию, специфицируется свойством IsDefault атрибута. В данном примере отображение на класс Shape принято по умолчанию. Поэтому, если запись имеет, скажем, значение "Q" в столбце ShapeCode, из этой записи по умолчанию будет сконструирован объект Shape, поскольку это значение не соответствует ни одному из специфицированных кодов дискриминатора.

Все изложенное почти полностью раскрывает концепцию отображения наследования сущностных классов. Теперь давайте рассмотрим полный код класса DataContext.

Мой полный класс DataContext

```
public partial class TestDB : DataContext
{
    public Table<Shape> Shapes;
    public TestDB(string connection) :
        base(connection)
{}
```

```

public TestDB(System.Data.IDbConnection connection) : base(connection)
{
}
public TestDB(string connection,
              System.Data.Linq.Mapping.MappingSource mappingSource) :
    base(connection, mappingSource)
{
}
public TestDB(System.Data.IDbConnection connection,
              System.Data.Linq.Mapping.MappingSource mappingSource) :
    base(connection, mappingSource)
{
}
}

[Table]
[InheritanceMapping(Code = "G", Type = typeof(Shape), IsDefault = true)]
[InheritanceMapping(Code = "S", Type = typeof(Square))]
[InheritanceMapping(Code = "R", Type = typeof(Rectangle))]
public class Shape
{
    [Column(IsPrimaryKey = true, IsDbGenerated = true,
            DbType = "Int NOT NULL IDENTITY")]
    public int Id;

    [Column(IsDiscriminator = true, DbType = "NVarChar(2)")]
    public string ShapeCode;

    [Column(DbType = "Int")]
    public int StartingX;

    [Column(DbType = "Int")]
    public int StartingY;
}
public class Square : Shape
{
    [Column(DbType = "Int")]
    public int Width;
}
public class Rectangle : Square
{
    [Column(DbType = "Int")]
    public int Length;
}

```

Здесь нет ничего нового помимо помещения ранее упомянутых классов в `[Your]DataContext` по имени `TestDB` и добавления к нему некоторых конструкторов. Теперь, в листинге 18.3, я вызову код создания базы данных.

Листинг 18.3. Код создания базы данных моего примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");
db.CreateDatabase();
```

Этот код не дает никакого экранного вывода, но если вы заглянете на ваш сервер базы данных, то увидите, что там появилась база под названием `TestDB` с единственной таблицей по имени `Shape`. Проверьте таблицу `Shape`, чтобы удостовериться, что в ней нет записей. Теперь, имея таблицу, давайте занесем в нее некоторые данные, используя код `LINQ to SQL` из листинга 18.4.

Листинг 18.4. Код создания некоторых данных в базе данных моего примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");
db.Shapes.InsertOnSubmit(new Square { Width = 4 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 3, Length = 6 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 11, Length = 5 });
db.Shapes.InsertOnSubmit(new Square { Width = 6 });
db.Shapes.InsertOnSubmit(new Rectangle { Width = 4, Length = 7 });
db.Shapes.InsertOnSubmit(new Square { Width = 9 });
db.SubmitChanges();
```

На заметку! В Visual Studio 2008 Beta 2 и более ранних выпусках метод InsertOnSubmit, вызванный в предыдущем коде, назывался Add.

В этом коде нет ничего нового. Я создаю объект DataContext и объекты сущностных классов, и вставляю их в таблицу Shapes. Затем вызываю метод SubmitChanges, чтобы сохранить их в базе данных. После запуска этого кода вы должны увидеть записи, представленные в табл. 18.1, в таблице Shape базы данных TestDB.

Таблица 18.1. Результаты выполнения предыдущего примера

Id	ShapeCode	StartingX	StartingY	Length	Width
1	S	0	0	NULL	4
2	R	0	0	6	3
3	R	0	0	5	11
4	S	0	0	NULL	6
5	R	0	0	7	4
6	S	0	0	NULL	9

Поскольку Id — столбец идентичности, его значения будут меняться при многократном запуске этого кода.

Теперь я выполню пару запросов к этой таблице. Сначала в листинге 18.5 я запрошу квадраты, к которым также будут относиться и прямоугольники, поскольку прямоугольники наследуются от квадратов. Затем я запрошу только прямоугольники.

Листинг 18.5. Код, выполняющий запросы в базе данных моего примера наследования сущностных классов

```
TestDB db = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");
// Сначала я получу все квадраты, включая и прямоугольники.
IQueryable<Shape> squares = from s in db.Shapes
                                where s is Square
                                select s;
Console.WriteLine("The following squares exist.");
foreach (Shape s in squares)
{
    Console.WriteLine("{0} : {1}", s.Id, s.ToString());
}
// Теперь получу только прямоугольники.
IQueryable<Shape> rectangles = from r in db.Shapes
                                    where r is Rectangle
                                    select r;
Console.WriteLine("{0}The following rectangles exist.", System.Environment.NewLine);
```

```

foreach (Shape r in rectangles)
{
    Console.WriteLine("{0} : {1}", r.Id, r.ToString());
}

```

В листинге 18.5 я, по сути, выполняю один и тот же запрос дважды, но только в первом случае запрашиваю лишь те записи, из которых получаются экземпляры квадратов, к которым относятся и прямоугольники — согласно моей схеме наследования. Во втором случае я запрашиваю записи, которые превращаются в экземпляры прямоугольников, исключая квадраты. Ниже показан результат:

```

The following squares exist.
1 : LINQChapter18.Square
2 : LINQChapter18.Rectangle
3 : LINQChapter18.Rectangle
4 : LINQChapter18.Square
5 : LINQChapter18.Rectangle
6 : LINQChapter18.Square
The following rectangles exist.
2 : LINQChapter18.Rectangle
3 : LINQChapter18.Rectangle
5 : LINQChapter18.Rectangle

```

Наследование сущностных классов может быть удобной техникой для конструирования иерархии сущностей из базы данных.

Транзакции

Я уже говорил вам о том, что когда вызывается метод `SubmitChanges`, если никакая транзакция в данный момент не активна, метод `SubmitChanges` создает для вас новую транзакцию. В результате все попытки модификаций базы данных, произведенные в течение единственного вызова `SubmitChanges`, будут помещены в контекст единой транзакции. Это очень удобно, но что если вам нужна транзакция, распространяющаяся за пределы единственного вызова `SubmitChanges`?

Я хочу представить пример, демонстрирующий, как вы могли бы выполнять обновления за несколько вызовов метода `SubmitChanges`, помещенных в одну транзакцию. Более того, я хочу, чтобы вызовы метода `SubmitChanges` обновляли разные таблицы. В листинге 18.6 я проведу изменения в записях базы данных `Northwind` и базы данных `TestDB`, только что созданной в разделе “Наследование сущностных классов”. Обычно каждый вызов метода `SubmitChanges` на каждом из этих объектов `DataContext` должен быть помещен внутрь своей собственной индивидуальной транзакции. В моем примере я хочу, чтобы оба вызова `SubmitChanges` были помещены в одну и ту же транзакцию.

Поскольку листинг 18.6 будет несколько длиннее, чем обычно, я сопровожу его своими пояснениями.

На заметку! Для следующего примера в ваш проект должна быть добавлена ссылка на сборку `System.Transactions.dll`.

Листинг 18.6. Включение в объемлющую транзакцию

```

Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
TestDB testDb = new TestDB(@"Data Source=.\SQLEXPRESS;Initial Catalog=TestDB");
Customer cust = db.Customers.Where(c => c.CustomerID == "LONEP").SingleOrDefault();
cust.ContactName = "Barbara Penczek";
Rectangle rect = (Rectangle)testDb.Shapes.Where(s => s.Id == 3).SingleOrDefault();
rect.Width = 15;

```

В приведенном коде я создаю объект `DataContext` для каждой базы данных. Затем я запрашиваю сущностный объект из каждой, и провожу изменения в каждом полученном сущностном объекте.

```

try
{
    using (System.Transactions.TransactionScope scope =
        new System.Transactions.TransactionScope())
    {
        db.SubmitChanges();
        testDb.SubmitChanges();
        throw (new Exception("Just to rollback the transaction."));
        // Здесь компилятор выдаст предупреждение,
        // поскольку следующая строка кода недостижима.
        scope.Complete();
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

```

На заметку! Имейте в виду, что поскольку здесь имеется код после генерации исключения, компилятор выдаст предупреждение, поскольку вызов метода `scope.Complete` недостижим для выполнения.

В приведенном коде я создаю экземпляр объекта `TransactionScope`, так что появляется объемлющая транзакция для включения объектов `DataContext` с их вызовами метода `SubmitChanges`. После вызова метода `SubmitChanges` на каждом `DataContext` я намеренно генерирую исключение, так что метод `scope.Complete` не вызывается и происходит откат транзакции.

Если бы я не поместил вызовы метода `SubmitChanges` в контекст объекта `TransactionScope`, то каждый вызов `SubmitChanges` имел бы свою собственную транзакцию, и его изменения были бы зафиксированы немедленно по окончании успешного вызова `SubmitChanges`. Как только в предыдущем коде генерируется исключение, транзакция выходит из контекста, и поскольку метод `Complete` не был вызван, транзакция откатывается. В этой точке все изменения, проведенные в базе данных, отменяются.

```

db.Refresh(System.Data.Linq.RefreshMode.OverwriteCurrentValues, cust);
Console.WriteLine("Contact Name = {0}", cust.ContactName);
testDb.Refresh(System.Data.Linq.RefreshMode.OverwriteCurrentValues, rect);
Console.WriteLine("Rectangle Width = {0}", rect.Width);

```

Важно помнить, что даже несмотря на то, что изменения не были успешно сохранены в базе данных, сущностные объекты продолжают содержать модифицированные данные. Помните, что даже если метод `SubmitChanges` не был завершен успешно, изменения поддерживаются в сущностных объектах, чтобы вы могли разрешить конфликты параллельного доступа и снова вызвать метод `SubmitChanges`. В этом случае вызовы метода `SubmitChanges` завершены успешно. Кроме того, как вы можете вспомнить из раздела "Несоответствие кэша результирующего набора" главы 16, новый запрос объектов из базы данных не даст в результате текущих значений из базы данных. Запрос к базе лишь определит, какие сущности должны быть включены в результирующий набор. Если эти сущности уже кэшированы в `DataContext`, то будут возвращены именно эти кэшированные сущностные объекты. Поэтому, чтобы действительно знать, какие значения ранее опрошенных сущностных объектов содержатся в базе данных, сущностные объекты должны сначала быть освежены вызовом метода `Refresh`.

Таким образом, для каждого из двух извлеченных сущностных объектов я сначала обновляю их, а затем отображаю на консоли измененное мной свойство, чтобы доказать, что изменения действительно были отменены. Взглянем на результат:

```
Just to rollback the transaction.  
Contact Name = Fran Wilson  
Rectangle Width = 11
```

Как видите, изменения значений в базе данных были отменены.

Совет. Если вы получите исключение типа "MSDTC on server '[server]\SQLEXPRESS' is unavailable" ("Служба MSDTC на сервере '[сервер]\SQLEXPRESS' не доступна") при работе с любым из примеров, использующих объект TransactionScope, убедитесь, что запущена служба Distributed Transaction Coordinator (распределенный координатор транзакций).

Резюме

В этой главе я продемонстрировал, как выполняются запросы к представлениям базы данных. Помните, что они на самом деле отображаются как таблицы, доступные только для чтения, так что вы уже знаете, как их опрашивать.

Затем я раскрыл тему наследования сущностных классов. Это удобная техника, позволяющая создавать экземпляры разных, но связанных отношением наследования, классов из записей одной и той же таблицы.

И, наконец, я немного углубился в тему транзакций, и продемонстрировал, как можно включить ваши обновления базы данных средствами LINQ to SQL в одну объемлющую транзакцию.

И напоследок я хочу сказать вам следующее: LINQ to SQL — это динамит, но не думайте, что это весь LINQ. Если вы пропустили предыдущие части этой книги, перепрыгнув непосредственно к LINQ to SQL, пожалуйста, прочтите остальные части, потому что некоторые из прочих программных интерфейсов LINQ просто фантастичны. В частности, я уверен, что возможности выполнения запросов к находящимся в памяти коллекциям и трансформации данных из коллекции одного типа класса в другой — это одни из наиболее замечательных и часто недооцениваемых возможностей LINQ. Также, если вы пишете код, использующий XML, то программный интерфейс LINQ to XML API может просто унести вас за облака.

Поскольку эта глава — не только последняя из посвященных LINQ to SQL, но также вообще последняя в этой книге, я хочу представить вам еще один последний пример. Один из аргументов, которые выдвигают критики LINQ, часто принимает форму вопроса: "Что такого может делать LINQ, чего я не мог бы сделать другими средствами?". Это верно, что существует масса средств для решения многих задач, выполнение которых теперь упрощено LINQ, но давайте не будем недооценивать абстракцию опроса данных, которую предлагает LINQ, а также консолидацию приемов получения информации из разных доменов данных.

В этом контексте я хочу представить вам один последний пример. В этом финальном примере я соединю данные, полученные из базы данных с данными из XML — просто чтобы доказать, что подобное возможно.

В листинге 18.7 я создаю объект XElement, передавая строку данных. Данные XML просто представляют отображение сокращенных названий штатов США на их полные названия. Затем я использую LINQ to SQL для запроса заказчиков из США. Я вызываю метод AsEnumerable на возвращенной последовательности, чтобы затем иметь возможность выполнять остальную часть запроса локально, а не в базе данных. Это необходимо, поскольку остальная часть запроса не может быть транслирована в SQL. Затем я

соединяю результаты из базы данных с данными XML, сравнивая свойство Region заказчика с атрибутом ID в элементах XML. Соединяя вместе эти два источника данных, я могу получить описание штата, специфицированного в свойстве Region заказчика. После этого я проектирую результат соединения на объект анонимного типа, содержащий в себе существенный объект Customer и описание штата, взятое из данных XML. Затем я выполняю перечисление результатов и отображаю CompanyName и Region заказчика, а также описание для Region из данных XML. Делая так, я соединяю данные из базы и XML. Смотрите, насколько все гладко! И после этого вы будете думать, что LINQ не дает вам ничего нового? Эта книга начиналась с кода, поэтому выглядит логичным также и завершить ее кодом. Но сначала посмотрим на результат:

```

Customer = Great Lakes Food Market : OR : Oregon
Customer = Hungry Coyote Import Store : OR : Oregon
Customer = Lazy K Kountry Store : WA : Washington
Customer = Let's Stop N Shop : CA : California
Customer = Lonesome Pine Restaurant : OR : Oregon
Customer = Old World Delicatessen : AK : Alaska
Customer = Rattlesnake Canyon Grocery : NM : New Mexico
Customer = Save-a-lot Markets : ID : Idaho
Customer = The Big Cheese : OR : Oregon
Customer = The Cracker Box : MT : Montana
Customer = Trail's Head Gourmet Provisioners : WA : Washington
Customer = White Clover Markets : WA : Washington

```

Как видите, я на самом деле смог отобразить Region каждой записи из базы на соответствующую запись из данных XML, чтобы получить описание штата. Покажите мне, как бы вы сделали то же самое с помощью ADO.NET и W3C XML DOM в одном операторе. Теперь посмотрим на код в листинге 18.7, который позволил достичь такого результата.

Листинг 18.7. Запрос, соединяющий данные из базы с данными XML

```

string statesXml =
@"
<States>
    <State ID=""OR"" Description=""Oregon"" />
    <State ID=""WA"" Description=""Washington"" />
    <State ID=""CA"" Description=""California"" />
    <State ID=""AK"" Description=""Alaska"" />
    <State ID=""NM"" Description=""New Mexico"" />
    <State ID=""ID"" Description=""Idaho"" />
    <State ID=""MT"" Description=""Montana"" />
</States>";
XElement states = XElement.Parse(statesXml);
Northwind db = new Northwind(@"Data Source=.\SQLEXPRESS;Initial Catalog=Northwind");
var custs = (from c in db.Customers
            where c.Country == "USA"
            select c).AsEnumerable()
            .Join(states.Elements("State"),
                  c => c.Region,
                  s => (string)s.Attribute("ID"),
                  (c, s) => new { Customer = c,
                                 State = (string)s.Attribute("Description") });
foreach (var cust in custs)
{
    Console.WriteLine("Customer = {0} : {1} : {2}",
                      cust.Customer.CompanyName,
                      cust.Customer.Region,
                      cust.State);
}

```

Предметный указатель

D

Document Object Model (DOM), 180

L

LINQ (Language Integrated Query), 23

форум LINQ, 31

LINQ to DataSet, 22; 316

операции, 316

LINQ to Entities, 22

LINQ to Objects, 22; 62

LINQ to SQL, 20; 22; 354; 365; 387; 426

LINQ to SQL API, 361; 388; 468; 519; 536

LINQ to XML, 19; 22; 195; 231; 242; 258

аннотации, 249

атрибуты, 242

операции, 258

события, 253

LINQ to XML API, 536

O

Object-relational mapping (ORM), 355

Object Relational Designer, 373; 386; 422

S

SQL, 423

трансляция, 423

SQLMetal, 368; 386

W

W3C DOM XML API, 183

X

XML (Extensible Markup Language), 19

XPath, 313

A

Агрегация, 165

Ассоциация, 358; 394

Б

База данных Northwind, 359; 389

В

Выражение

дерево выражений, 37

запроса

грамматика, 51

трансляция, 52

лямбда, 32; 35; 307

Г

Грамматика выражений запросов, 51

Граф

согласованность графа, 432

Группировка, 109

Д

Делегат

Func, 67

Дерево выражений, 37

З

Загрузка

немедленная с помощью класса

DataLoadOptions, 398

отложенная, 396

Закон Сарбейнса-Оксли, 291

Запрос

LINQ to Objects, 62

LINQ to SQL, 392; 423

возвращающий последовательность

IQueryable<T>, 393

на базе данных Northwind, 392

на объектах Table<T>, 394

SQL с конструкцией IN, 412

XML с использованием LINQ to SQL, 20

XML с использованием LINQ to XML, 19

алфавитный указатель стандартных

операций запросов, 68

использующий синтаксис выражений

запросов, 50

использующий стандартный синтаксис

точечной нотации, 50

к базе данных SQL Server, 20

класса с реализованными частичными

методами, 461

к представлению базы данных, 537

отложенный, 30; 64; 196; 409

сложный, 285

с преднамеренным исключением,

отложенным до перечисления, 29

И

Инициализация

коллекций, 41

объектов, 40

Интерфейс

IComparer<T>, 94

IEqualityComparer<T>, 112

IEnumerable<T>, 63

IExecuteResult, 464

IGrouping<K, T>, 110

IMultipleResults, 466

IQueryable, 362

ISingleResult, 465

К

Квантификатор, 160

Класс

DataContext, 30; 357; 468

объект Log, 30

DataLoadOptions, 403

DataRow, 318; 329

DataSet, 346

DataTable, 339

Employee, 71; 133

EmployeeOptionEntry, 71; 134

Extensions, 313

string, 43

XAttribute, 201; 242

XCData, 210

XComment, 202

XContainer, 202

XDeclaration, 203

XDocument, 205

XDocumentType, 203

XElement, 199; 211

XName, 205

XNamespace, 206

XProcessingInstruction, 206

XStreamingElement, 208

ассоциированный

обновление, 414

сущностный (entity), 426

наследование, 538

экземпляр класса, 43

Ключевое слово

var, 26; 38

Конкатенация, 88

Конструирование

функциональное, 187

Л

Лямбда-выражение, 32; 35; 307

М

Метод

анонимный, 34

именованный, 33

расширяющий, 43; 46

фильтрации, 34

частичный, 47

Множество, 115

О

Объект

инициализация, 40

Объектная модель LINQ to XML, 195

Операция

Aggregate, 68; 175

All, 68; 162

Ancestors, 259

AncestorsAndSelf, 263

Any, 68; 160

AsEnumerable, 68; 124; 339

Attributes, 265

Average, 68; 173

Cast, 27; 68; 120

Concat, 68; 88

Contains, 68; 163; 413

CopyToDataTable, 340

Count, 165

DefaultEmpty, 68

DefaultIfEmpty, 126

DescendantNodes, 267

DescendantNodesAndSelf, 268

Descendants, 270

DescendantsAndSelf, 272

Distinct, 68; 115; 319

ElementAt, 68; 158

ElementAtOrDefault, 68; 159

Elements, 274

Empty, 68; 131

Except, 68; 118; 322

Field, 332

First, 68; 149

FirstOrDefault, 68; 151

GroupBy, 68

GroupJoin, 68; 108

InDocumentOrder, 276

Interest, 68

Intersect, 117; 324

Join, 68; 106

Last, 68; 152

LastOrDefault, 68; 154

LongCount, 68; 166

Max, 68; 171

Min, 68; 169

Nodes, 277

OfType, 27; 68; 122

OrderBy, 68; 92

OrderByDescending, 69; 96

Range, 69; 130

Remove, 279

Repeat, 69; 131

Reverse, 69; 105

Select, 42; 69; 74

SelectMany, 42; 69; 79

SequenceEqual, 69; 147; 327

SetField, 337

Single, 69; 155

SingleOrDefault, 69; 157

Skip, 69; 86

SkipWhile, 69; 87

Sum, 69; 168

Take, 69; 82

`TakeWhile`, 69; 84
`ThenBy`, 69; 99
`ThenByDescending`, 69; 102
`ToArrayList`, 69; 136
`ToDictionary`, 69; 139
`ToList`, 69; 138
`ToLookup`, 69; 143
`Union`, 69; 116; 326
`Where`, 69; 72
агрегирования, 165
генерации, 130
группировки, 110
-квантификатор, 160
конкатенации, 88
множества, 115
не отложенная, 136
отложенная, 70
преобразования, 120; 136
разбиения (partitioning), 82
соединения (join), 106
эквивалентности, 147
элемента, 126; 149

П

Последовательность, 63
Предикат, 45
Проблема
объектно-реляционной потери
соответствия, 471
Хэллоуина, 196
Программа
Object Relational Designer, 373; 386; 422
SQLMetal, 368; 386

С

Соединение
внешнее, 406
внутреннее, 405
случайное, 404

Тип

анонимный, 42
Транзакция, 543
SQL, 423
Трансформация XML, 290
с использованием XSLT, 291
с использованием функционального
конструирования, 293

У

Узел
добавление, 231
обновление, 238
обработка в плоской структуре, 299
удаление, 235

Ф

Форум LINQ, 31

Х

Хранимая процедура, 382; 383; 441; 449; 465
Customers Count By Region, 503
CustOrderTotal, 504
Get Customer And Orders, 449; 451; 507
InsertCustomer, 380–385
Whole Or Partial Customers Set, 506

Э

Экземпляр класса, 43
Элемент, 126

Я

Язык
XML, 19

Научно-популярное издание

Джозеф С. Раттц-мл.

LINQ: язык интегрированных запросов в C# 2008

для профессионалов

Верстка Т.Н. Артеменко
Художественный редактор В.Г. Павлютин

ООО "И.Д. ВИЛЬЯМС"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 25.04.2008. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 45,15. Уч.-изд. л. 32,9.
Тираж 1500 экз. Заказ № 9665.

Отпечатано по технологии СтР
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

LINQ ЯЗЫК ИНТЕГРИРОВАННЫХ ЗАПРОСОВ В C# 2008

ДЛЯ ПРОФЕССИОНАЛОВ

Джозеф Раттц-мл. занимается программированием на C++, Java, ASP, ASP.NET, C#, HTML, DHTML и XML. Он разрабатывал приложения для SCT, DocuCorp, IBM, комитета по проведению олимпийских игр в Атланте, CheckFree, NCR, EDS, Delta Technology, Radiant Systems и Genuine Parts Company. В настоящее время его можно найти в Genuine Parts Company — родительской компании NAPA, — в департаменте Automotive Parts Group Information Systems, где он трудится над своим детищем — Web-сайтом Storefront. Этот сайт обслуживает хранилища NAPA, предоставляя их счета и данные в сети систем AS/400.

НА WEB-САЙТЕ

Исходные коды всех примеров, рассмотренных в книге, можно загрузить с Web-сайта издательства по адресу:
<http://www.williamspublishing.com>



www.williamspublishing.com

Apress®

www.apress.com

Уважаемый читатель!

Эта книга целиком и полностью посвящена написанию кода. Она начинается с кода и кодом же завершается. При написании книги я преследовал цель предложить вам полный набор полезных примеров применения LINQ. Вместо того чтобы показывать одиночные и простые примеры, я постарался представить полную картину и продемонстрировать всю мощь операций и прототипов LINQ, которые призваны извлечь максимум из вложенных средств и усилий.

В книге я стремился давать информацию в такой форме, которая бы позволила использовать ее непосредственно в разрабатываемых проектах. Вместо того чтобы уводить читателя в сторону от базовых принципов, рассматривая малопригодные для применения крупные демонстрационные приложения, в этой книге внимание фокусируется на каждой операции, методе и классе LINQ. Однако там, где сложность нужна для демонстрации той или иной возможности, предлагаются и примеры соответствующего уровня. В частности, примеры кода, иллюстрирующие конфликты параллельного доступа, в действительности их и создают, так что вы можете пошагово пройти код, уловить суть проблемы и разобраться в способе ее разрешения.

Книга предназначена для тех, кто знаком с языком C# и желает изучить средства C# 3.0, связанные с LINQ. При этом знание всех особенностей версий C# 2.0 и C# 3.0 не требуется. Там, где нужны более глубокие знания, я начинаю с нуля, поэтому разобраться со всем сможет даже новичок в C#.

Джозеф Раттц-мл.

Категория: программирование на C#

Предмет рассмотрения: LINQ

Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-8459-1427-9



08120

9 785845 914279