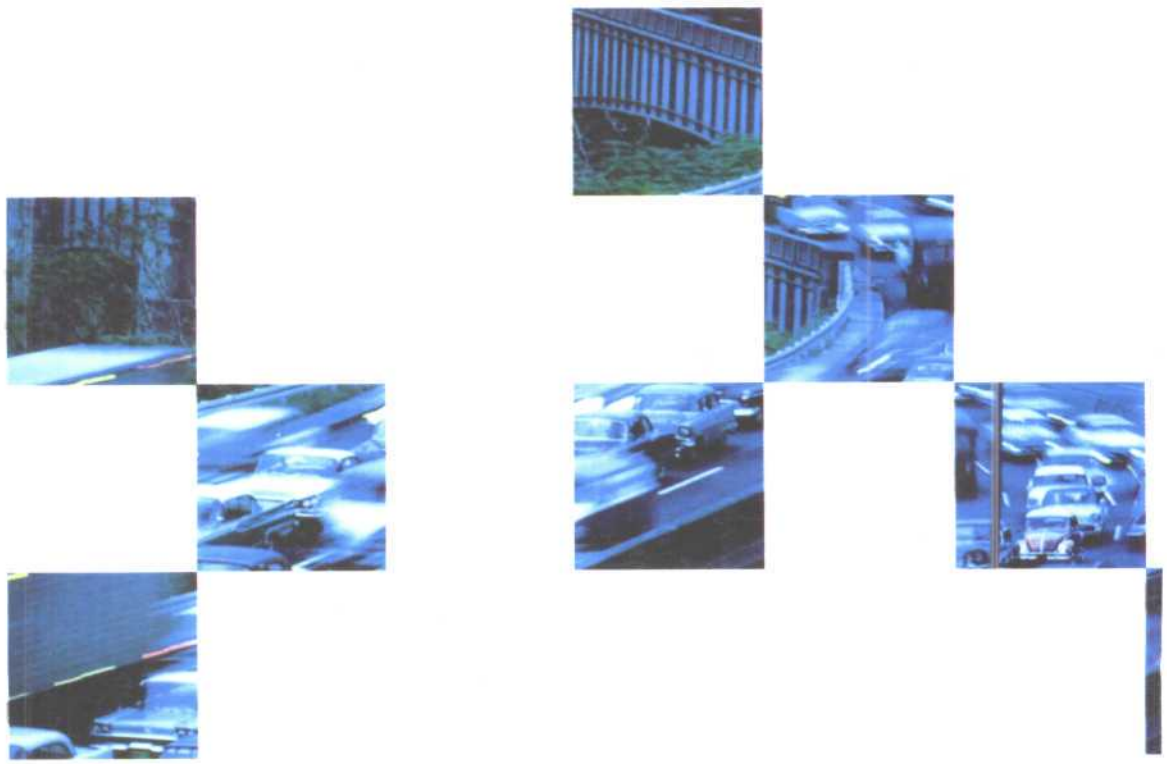
Предисловие Криса Селлза

Microsoft .net Development Series

Практическое использование ADO.NET

Доступ к данным в Internet



Шон Вилдермьюс

Практическое использование ADO.NET

Pragmatic ADO.NET

Data Access for the *Internet World*

Shawn Wildermuth



ADDISON-WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris - Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Практическое использование ADO.NET

Доступ к данным в *Internet*

Шон Вилдермьюс



Москва • **Санкт-Петербург** • **Киев**
2003

ББК 32,973.26-018.2.75
В44
УДК 681.3,07

Издательский **дом** "Вильямс"

Зав. редакцией С.Я. *Тригуб*

Руководитель проекта *В. В. Александров*

Перевод с английского *А.А. Борисенко, А.В. Журавлева, О.А. Лещинского*

Под редакцией *А. В. Журавлева*

По общим вопросам обращайтесь в Издательский дом "Вильяме" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Вилдермьюс, Шон.

В44 Практическое использование ADO.NET. Доступ к данным в Internet. : Пер. с англ. — М. : Издательский дом "Вильяме", 2003. — 288 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0450-1 (рус.)

Эта книга представляет собой практическое руководство по использованию первой библиотеки доступа к данным, спроектированной специально для упрощения создания Web-приложений. Содержащийся в книге материал поможет разработчикам изучить основные концепции ADO.NET и познакомиться с практическими методами решения распространенных задач. На первых страницах книги автор предлагает совершить небольшой экскурс в историю создания компанией Microsoft технологий универсального доступа к данным и проследить эволюционный путь ADO.NET. Большая часть книги посвящена использованию библиотеки ADO.NET для взаимодействия с базами данных и остальной частью .NET Framework. Кроме того, автор дает ряд полезных советов в отношении создания масштабируемых и высокопроизводительных приложений. В конце книги автор подробно излагает стратегию преобразования кода ADO в код ADO.NET.

Книга рассчитана на пользователей средней и высокой квалификации.

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 Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Pearson Esucation, Inc.

Copyright © 2003 by Shawn Wildermuth

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to (he Agreement with R&I Enterprises International, Copyright © 2003

ISBN 5-8459-0450-1 (рус.)
ISBN 0-201-74568-2 (англ.)

© Издательский дом "Вильяме", 2003
© Shawn Wildermuth, 2003

Оглавление

Введение 12
Предисловие 13
Благодарности 16

Часть I. Основы ADO.NET 17
Глава 1. Причины возникновения и краткий обзор ADO.NET 18
Глава 2. ADO.NET: подключение к источнику данных 32
Глава 3. Выполнение команд 55
Глава 4. Получение данных 76

Часть II. Класс DataSet 95
Глава 5. Создание объекта DataSet 96
Глава 6. Типизированные классы DataSet 128
Глава 7. Манипулирование объектом DataSet 156
Глава 8. Обновление базы данных 178

Часть III. Практическое использование ADO.NET 211
Глава 9. ADO.NET и XML 212
Глава 10. Привязка данных в ADO.NET 235
Глава 11. Масштабируемость и производительность приложений ADO.NET 251
Приложение А. Стратегии перехода от ADO к ADO.NET 265
Предметный указатель 277

Содержание

Введение 12

Предисловие 13
Для кого предназначена эта книга 13
Часть I. Основы ADO.NET 13
Глава 1. Причины возникновения и краткий обзор ADO.NET 13
Глава 2. ADO.NET: подключение к источнику данных 13
Глава 3. Выполнение команд 13
Глава 4. Получение данных 14
Часть II. Класс DataSet 14
Глава 5. Создание объекта DataSet 14
Глава 6. Типизированные классы DataSet 14
Глава 7. Манипулирование объектом DataSet 14
Глава 8. Обновление базы данных 14
Часть III. Практическое использование ADO.NET 14
Глава 9. ADO.NET и XML 15
Глава 10. Привязка данных в ADO.NET 15
Глава 11. Масштабируемость и производительность приложений ADO.NET 15
Приложение А. Стратегии перехода от ADO к ADO.NET 15

Благодарности 16

Часть I. Основы **ADO.NET** **17**

Глава 1. Причины возникновения и краткий обзор ADO.NET 18
Краткая история универсального доступа к данным 18
Преимущества ADO.NET 21
Краткий курс ADO.NET 23
Пространства имен ADO.NET 23
Структуры данных ADO.NET 23
Объектная модель управляемого поставщика данных ADO.NET 27
Резюме 31

Глава 2. ADO.NET: подключение к источнику данных 32
Первые впечатления 32
Соединения 33
Строки соединений 34
Встроенная система безопасности 39
Изменение базы данных 39
Организация пула соединений 41
События объекта Connection 45
Фабрика соединений 48

Получение информационной схемы базы данных с помощью поставщика OLE DB 50
Обработка ошибок в .NET 51
Исключения ADO.NET 53
Резюме 54

Глава 3. Выполнение команд 55

Команды 55
Создание объекта Command 55
Типы команд 56
Выполнение команд 57
Результирующий набор данных 58
Использование параметров 58
Транзакции и ADO.NET 67
Уровни изоляции 70
Точки сохранения в транзакциях SQL Server 71
Службы Enterprise Services и COM+ 72
Пакетные запросы 72
Резюме 75

Глава 4. Получение данных 76

Чтение данных 76
Краткий обзор SQL-оператора SELECT 76
Объект DataReader 79
Создание объекта DataReader 79
Функциональность объекта DataReader 80
Доступ к данным с помощью объекта DataReader 80
Параметры метода ExecuteReader 82
Что такое результирующие наборы данных? 86
Обработка множественных результирующих наборов данных 86
Работа с метаданными объекта DataReader 86
Создание простого приложения, взаимодействующего с базой данных 89
Доступ к базе данных 90
Объекты данных 90
Код Windows-формы 93
Резюме 94

Часть II. Класс DataSet 95

Глава 5. Создание объекта DataSet 96

Что такое объект DataSet? 96
Структура объекта DataSet 96
Объект DataSet и управляемые поставщики данных 97
Заполнение объекта DataSet 98
Объект DataAdapter 98
Создание объекта DataSet на основе информации, хранящейся в базе данных 99
Создание объекта DataSet на основе XML-документа 103
Создание объекта DataSet программным путем 105
Определение схемы объекта DataSet 106
Зачем нужно определять схему объекта DataSet? 106
Вывод схемы объекта DataSet с помощью объекта DataAdapter 107
Использование языка XSD для определения схемы объекта DataSet 109

Создание схемы объекта DataSet программным путем ПО
Схема столбца 121
Резюме 127

Глава 6. Типизированные классы DataSet 128

Что такое типизированный класс DataSet? 128
Создание типизированного класса DataSet 131
Использование Visual Studio .NET для создания типизированного класса DataSet 131
Использование программы XSD.exe для создания типизированного класса **DataSet** 139
Настройка сгенерированного кода с помощью специальных аннотаций 140
Использование типизированного класса DataSet 142
Упрощение уровня бизнес-объектов 143
Разработка бизнес-логики 144
Резюме 155

Глава 7. Манипулирование объектом DataSet 156

Изменение данных 156
Добавление строк 157
Удаление строк 158
Чтение и запись значений столбцов строки 159
Версия строки 164
Состояние строки 166
Перемещение по объекту DataSet 167
Перемещение вдоль отношений 167
Объект Data View 170
Поиск в объекте DataSet 173
Поиск с помощью метода DataTable.Select 173
Поиск с помощью объекта DataView 174
Слияние объектов DataSet 175
Резюме 177

Глава 8. Обновление базы данных 178

Проблемы, связанные с использованием отсоединенных данных 178
Параллелизм **в ADO.NET 179**
Обновление объекта DataSet с помощью объекта DataAdapter 179
Реализация оптимистического параллелизма 180
Реализация пессимистического параллелизма 197
Реализация деструктивного параллелизма 201
Несколько наиболее распространенных вопросов, связанных с обновлением данных 204
Работа с объектами DataSet, состоящими из нескольких таблиц 204
Использование локальных транзакций для обновления базы данных 207
Получение идентификатора новой строки от базы данных SQL Server 209
Резюме 210

Часть III. Практическое использование ADO.NET 211

Глава 9. ADO.NET и XML 212

.NET и XML **212**
Класс DataSet и XML 212
Преобразование данных объекта DataSet в формат **XML** 212

Сохранение данных объекта DataSet в формате XML 218
Пространства имен объекта DataSet 221
Заполнение объекта DataSet данными из XML-файла 221
Стратегии использования XML-документа в формате DiffGram 225
Схема объекта DataSet 226
Класс XmlDataDocument 229
Поиск данных в объекте DataSet с помощью запросов XPath 231
Преобразование объекта DataSet с помощью языка XSLT 233
Резюме 234

Глава 10. Привязка данных в ADO.NET 235

Что такое привязка данных в .NET? 235
Привязка данных в Windows-формах 235
Простая привязка данных 237
Сложная привязка данных 239
Привязка данных к элементу управления DataGrid 240
Привязка типа "родитель-потомок" 242
Использование класса CurrencyManager 243
Привязка данных в ASP.NET 246
Простая привязка данных 246
Сложная привязка данных 247
Привязка данных к элементам управления, предназначенным для работы с данными 247
Привязка данных с помощью объекта DataReader 248
Некоторые соображения, касающиеся производительности 249
Резюме 250

Глава 11. Масштабируемость и производительность приложений ADO.NET 251

А стоит ли волноваться? 251
Проектирование масштабируемых систем 252
Связность компонентов системы 253
Что предшествовало ADO.NET 253
Проблемы, связанные с подсоединенными данными 253
Проблемы масштабирования сервера баз данных 254
Как может помочь ADO.NET 255
Кэширование данных на Web-сервере 255
Масштабирование информации, хранящейся в базе данных 256
На практике 258
Можно ли масштабировать объекты DataReader? 259
Производительность ADO.NET 260
Взаимодействие с базой данных 260
Взаимодействие с объектами DataSet 260
Несколько полезных советов 260
Используйте схему объекта DataSet 261
Используйте типизированные объекты DataSet для создания уровней бизнес-правил 261
Сокращайте количество обращений к базе данных 261
Кэшируйте данные часто и заранее 261
Найдите и не отпускайте от себя администратора базы данных 262
Изолируйте разработчиков от базы данных 262
Ограничьте использование объекта DataReader на страницах ASP.NET 262
Используйте фабрики соединений 263

Не помещайте строки соединения в исходный код 263
Не предоставляйте пользователям доступ к базе данных 263
Резюме 264

Приложение А. Стратегаи перехода от ADO к ADO.NET 265

Планирование перехода на ADO.NET 265
Изменение архитектуры ADO-приложений 265
Чего не хватает в ADO.NET? 270
ADO.NET-эквиваленты объектов ADO 271
Отображение типов данных ADO на типы данных .NET 271
Поставщики и управляемые поставщики 272
Преобразование кода установки соединения 272
Преобразование кода создания объекта команды 273
Преобразование кода создания объекта Recordset 274
Использование объектов ADO Recordset в ADO.NET 275
Резюме 276

Предметный указатель 277

Моей дорогой Трише,
без терпения и заботы которой
этой книги никогда бы не было.

Введение

Впервые я столкнулся с Шоном несколько лет назад. Он подписался на те же списки рассылки, что и я, часто высказывал интересные идеи. Это побудило меня рассматривать его как одного из потенциальных помощников при работе над проектами. Однажды поздно вечером я отправил электронное письмо, рассчитывая получить ответ на следующий день. Через сорок пять минут, уже собираясь лечь спать и последний раз проверив свою почту, я с удивлением обнаружил ответ Шона, хотя в той временной зоне, где он находился, была глубокая ночь. Кроме того, что Шон просто желал мне помочь, он с большим энтузиазмом отнесся к данному проекту; у него был опыт, и его переполняли различные идеи. Так было положено начало взаимоотношений, которые вылились в ежедневное общение на протяжении последующих трех лет. Стоит ли говорить, что все это время работа велась очень продуктивно.

Когда наш совместный проект близился к завершению, Шона попросили оказать помощь в создании еще одного проекта — на этот раз просьба исходила от сообщества разработчиков баз данных Microsoft. Миссия Шона заключалась в том, чтобы помочь разобраться в последней технологии доступа к данным компании Microsoft: ADO.NET. Как и ранее, Шон с энтузиазмом принялся за дело. Он создал свой собственный Web-форум, посвященный ADO.NET, а также подписался на новый список рассылки по .NET, организованный компанией DevelopMentor, где отвечал на огромное количество вопросов. Вскоре в базе знаний Шона был собран огромный материал о важности ADO.NET в современном "сетевом" мире, снабженный детальной информацией о практическом применении этой технологии. В итоге у Шона возник вполне законный вопрос — а не написать ли ему обо всем этом книгу? Я ответил, что если у него появится свободное время, то следует так и сделать. Написать хорошую книгу (а в случае с Шоном о другом речь просто идти не могла) очень сложно. Единственный известный способ добиться этого заключается в том, чтобы иметь в своем сердце что-то, о чем ты просто не можешь не рассказать всему остальному миру.

К счастью для всех нас, рассказ Шона нашел свое место на страницах книги, которую вы сейчас держите в руках. И не важно, помогал ли вам Шон разобраться с вашими проблемами ранее, или же вы впервые сталкиваетесь с его работой, я уверен, что, как и в случае со мной, это будут очень продуктивные взаимоотношения.

Крис Селлз (Chris Sells),
www.sellsbrothers.com

Предисловие

ADO.NET — не ADO. Наверное, это наиболее важный момент, который освещается в книге. Понимание ADO.NET предполагает не только умение извлечь информацию из базы данных или обновить ее, но и, прежде всего, понимание целей проектирования ADO.NET. Созданная компанией Microsoft новая технология доступа к данным существенно отличается от ее предыдущих разработок и превращает ADO в некоего доисторического монстра,

Эта книга не является справочником или "полным" и "всеобъемлющим" руководством по ADO.NET. Прежде всего она написана для того, чтобы помочь разработчикам делать свою повседневную работу. Другими словами, книга содержит в себе множество полезных советов по использованию ADO.NET.

Для кого предназначена эта книга

Книга предназначена для разработчиков, у которых есть основные знания о базах данных и платформе .NET. И хотя читателю не обязательно иметь опыт работы с ADO, тем, у кого он есть, будет легче усвоить изложенный в книге материал.

Часть I. Основы ADO.NET

В первой части рассматриваются базовые понятия новой технологии доступа к данным ADO.NET.

Глава 1. Причины возникновения и краткий обзор ADO.NET

Прежде чем познакомить читателя с принципами использования ADO.NET, его вниманию предлагается предыстория создания этой библиотеки доступа к данным. В главе 1 рассматриваются различные технологии доступа к данным, разработанные Microsoft, и объясняется, каким путем компания пришла к созданию ADO.NET.

Глава 2. ADO.NET: подключение к источнику данных

Перед тем как выполнить какие-либо действия по отношению к базе данных, с ней необходимо установить соединение. Глава 2 полностью посвящена вопросу подключения к базе данных с использованием ADO.NET. В частности, здесь рассматриваются строки соединения, организация пула соединений при использовании различных управляемых поставщиков данных, шаблоны установки соединения, а также способы минимизации количества подключений к базе данных.

Глава 3. Выполнение команд

В конечном итоге взаимодействие с базой данных сводится к выполнению команд и получению результатов. В этой главе рассматривается объект Command и его использование для выполнения простых SQL-операторов, параметризованных запросов, хранимых процедур и операторов пакетного выполнения.

Глава 4. Получение данных

Объект ADO.NET DataReader предназначен для извлечения информации из базы данных. В главе 4 рассматривается использование объекта DataReader для доступа к базе данных и приводится пример класса, в котором этот объект выступает в качестве источника данных.

Часть II. Класс DataSet

Объект DataSet — это сердце ADO.NET. Для его эффективного использования необходимо понять, как он может помочь при решении повседневных задач доступа к данным.

Глава 5. Создание объекта DataSet

Эта глава поможет разобраться в том, что представляет собой объект **DataSet и** почему его необходимо использовать. Рассматривается множество способов создания объекта **DataSet,** включая применение для этой цели объекта **DataAdapter** и документа XML. В эту главу вошло также исчерпывающее описание использования схемы объекта **DataSet.**

Глава 6. Типизированные классы DataSet

Типизированный объект **DataSet** помогает разработчику создавать код, способный адаптироваться к изменению схемы. Глава 6 полностью посвящена новой модели программирования, в соответствии с которой типизированные объекты **DataSet** создаются и используются в качестве основы для уровня бизнес-логики приложения. При рассмотрении примеров создания типизированных объектов **DataSet** используются средства Visual Studio .NET и командной строки.

Глава 7. Манипулирование объектом DataSet

После создания объекта DataSet вам необходимо научиться использовать его для манипулирования данными и понять его внутреннюю структуру. В главе 7 рассматривается объектная модель DataSet, а также приведены примеры решения типичных проблем, возникающих при обработке данных.

Глава 8. Обновление базы данных

После проведения манипулирования с объектом DataSet необходим способ отразить внесенные изменения в базе данных. Глава 8 посвящена решению задачи параллелизма в рамках отсоединенного режима доступа к информации. В нее включены примеры использования встроенных средств обеспечения оптимистического параллелизма, а также описан способ реализации пессимистического и деструктивного параллелизма. Кроме того, в данной главе обсуждается вопрос нарушения параллелизма и приводятся конкретные примеры решения проблем, связанных с возникновением таких нарушений.

Часть III. Практическое использование ADO.NET

Теперь, когда мы знаем, как получить доступ к данным, манипулировать ими и отражать внесенные **изменения** в базе данных, нам необходимы сведения о способе взаимодействия **ADO.NET** с остальной частью .NET Framework.

Глава 9. ADO.NET и XML

XML — это формат представления информации. В свою очередь библиотека ADO.NET обеспечивает средства для работы с данными в среде .NET. Два указанных факта создают предпосылку для тесной интеграции ADO.NET и XML с целью объединения информации, поступающей из различных источников. Глава 9 посвящена рассмотрению XML и средств поддержки этого формата в библиотеке ADO.NET.

Глава 10. Привязка данных в ADO.NET

В среду .NET интегрированы две технологии, основанные на использовании форм: Windows-формы и Web-формы. К счастью, все объекты-контейнеры ADO.NET (DataReader, DataSet и DataTable) поддерживают непосредственную привязку данных, о чем и рассказывается в главе 10.

Глава 11. Масштабируемость и производительность приложений ADO.NET

Эта глава подытоживает все изложенное в предыдущих главах. Рассматриваемые в ней вопросы непосредственно касаются построения легкомасштабируемых и высокопроизводительных систем. Кроме этого, здесь приводятся ценные практические советы по созданию ориентированных на работу с базами данных приложений в целом и использованию ADO.NET в частности.

Приложение A. Стратегии перехода от ADO к ADO.NET

Мы не можем гарантировать того, что весь код .NET-приложения будет использовать новые технологии. Существует огромное количество кода, который необходимо перенести на платформу .NET. В этом приложении приводятся различные стратегии "выживания" в смешанных системах, предполагающих обращение к структурам ADO из кода ADO.NET и наоборот.

Благодарности

Я хочу поблагодарить всех, кто помог мне написать эту книгу. Прежде всего я хочу выразить признательность Крису Селлзу (Chris Sells) за его помощь на каждой стадии данного проекта. От самого начала работы над книгой до скрупулезного пересмотра каждой ее главы большее количество раз, чем я могу сосчитать, Крис делал все, чтобы книга получилась хорошей. К тому же он был моим наставником и вдохновителем, поддерживающим меня на каждом шагу и помогающим понять, на чем нужно сосредоточиться прежде всего. Я также благодарен бессчетному количеству людей из списка рассылки DevelopMentor .NET (discuss.develop.com), которые отвечали на мои вопросы, а также задавали мне такие вопросы, которых я даже и не думал касаться в этой книге. Кроме того, я признателен всем сотрудникам компании OneSource Information Systems за терпение, проявленное ими во время моей работы над этой книгой.

Напоследок я хотел бы поблагодарить всех тех людей, которые способствовали выходу книги в свет. Это Триша Палее (Tricia Palese), Стефани Томас (Stephanie Thomas), Крис Таварес (Chris Tavares), Боб Бошемин (Bob Beauchemin), Питер Залекси (Peter Zaleksy), Кристоф Фок (Cristof Falk), Скотт Юранек (Scott Juranek), Пуруш Рудракшала (Purush Rudrakshala), Гленн Тиммз (Glenn Thimmes), Эдвард Хинтон (Edward Hinton), Марк Изрейел (Mark Israel), Джозеф Фикара (Joseph Ficara), Дэйвид Авакян (David Avakian), Тодд Клеметсон (Todd Clemetson), Стивен Райт (Steven Wright), Маршал Харрисон (Marshall Harrison), Кристин Эриксон (Kristin Erickson), Дженнифер Аллен (Jennifer Allen) и Омри Гэцит (Omri Gazitt).

Шон Вилдермьюс (Shawn Wildermuth),
Тьюксбери, штат Массачусетс,
июнь 2002,
www.adoguy.com

Часть I

Основы ADO.NET

Глава 1. Причины возникновения и краткий обзор ADO.NET
Глава 2. ADO.NET: подключение к источнику данных
Глава 3, Выполнение команд
Глава 4, Получение данных

Глава 1

Причины возникновения и краткий обзор ADO.NET

В этой главе...

Краткая история универсального доступа к данным

Преимущества ADO.NET

Краткий курс ADO. NET

Добро пожаловать в мир ActiveX Data Objects for .NET или же просто ADO.NET — именно так мы будем называть эту библиотеку в дальнейшем. К счастью для нас, ADO.NET — не просто еще один API, это в самом буквальном смысле новая философия доступа к данным. Тем не менее прежде чем перейти к непосредственному изучению ADO.NET, давайте совершим небольшой экскурс в историю.

Краткая история универсального доступа к данным

Для того чтобы ясно представить себе мотивы создания компанией Microsoft библиотеки ADO.NET, следует рассмотреть ее предыдущие API доступа к данным. В течение последних десяти лет компания Microsoft пыталась решить проблему универсального доступа к данным. В Microsoft понимали, что для разработчиков всегда важно соблюдение баланса между простотой и производительностью. К сожалению, эти две веши часто были несовместимы.

Поскольку доступ к базе данных очень важен для большого числа приложений, в самом начале 90-х годов Microsoft разработала стратегию, нацеленную на помощь своим разработчикам, часто создававшим приложения, которым необходимо было взаимодействовать с базами данных на других платформах, таких, как миникомпьютеры и мэйнфреймы UNIX. Предложенное Microsoft решение заключалось в использовании интерфейса прикладного программирования (Application Programming Interface — API) ODBC (Open Database Connectivity — открытый интерфейс доступа к базам данных), который позволял получать доступ к данным на подобных системах.

Вместе с тем Microsoft знала, что многим разработчикам потребуется также решение для настольных баз данных. Поэтому в 1992 году увидел свет язык Visual Basic 2.0 и интерфейс VT Objects, позволяющий писать на Visual Basic код доступа к ODBC API. VT Objects представлял собой очень простой интерфейс, который обеспечивал немногим более, чем подключение к серверным базам данных. Другими словами, в этом интерфейсе Microsoft представила только ту часть ODBC API, которая могла удовлетворить потребности VB-разработчиков в области доступа к данным.

Компании Microsoft также было известно, что разработчики столкнулись с необходимостью локального хранения информации в формате настольной базы данных. В результате компания выпустила настольную базу данных Access 1.0, включающую в себя новую технологию доступа к данным Jet и новую версию интерфейса VT Object с несколько измененным названием — DAO (Data Access Objects — объекты доступа к данным). Уже в первой версии DAO была представлена концепция объектов

Connection и Recordset, которая существует и по сегодняшний день. К сожалению, DAO были присущи и недостатки — отсутствие поддержки многопоточного окружения крайне негативно сказывалось при возрастании нагрузки и при работе в больших клиент-серверных окружениях.

Microsoft попыталась заново "изобрести колесо", выпустив интерфейс доступа к данным OLE DB (OLE for Databases — OLE для баз данных). OLE DB привязывал разработчиков баз данных к модели компонентных объектов Microsoft (Component Object Model — COM). Итак, OLE DB — это набор COM-интерфейсов, поддерживающих модель доступа к данным "потребитель/поставщик" на корпоративном уровне. Для OLE DB безразличен источник данных — будь то настольная база данных (например, Access), база данных масштаба предприятия (например, SQL Server или Oracle) или же источник, и вовсе не являющийся базой данных (например, электронная таблица Excel). Если разработчику было нужно предоставить свои данные через OLE DB, он должен был написать поставщик данных OLE DB. К сожалению, создание поставщиков OLE DB оказалось чрезвычайно сложным заданием для большинства разработчиков (разве что за исключением наиболее талантливых C++-программистов). Выпускать потребители данных OLE DB было проще, однако программистам пришлось использовать Visual C++, заплатив за это утратой свободы выбора языка программирования.

Одной из последних разработок Microsoft стала технология ActiveX Data Objects (ADO), обеспечивающая COM-оболочку для OLE DB. Интерфейс ADO был разработан исключительно с целью упростить работу с OLE DB. Вскоре после выхода ADO разразилась Internet-революция. Разработчикам понадобился легкий в использовании API для того, чтобы сделать свои Web-узлы и Web-приложения ориентированными на взаимодействие с базами данных. Так как код ADO легко встраивался в Web-страницы, новая технология отлично интегрировалась с информационным сервером Internet (Internet Information Server — IIS) и ASP-страницами (Active Server Page — активная серверная страница). Вскоре ADO стал стандартом де-факто для Web-узлов Internet. Действительно, небольшим Web-узлам ADO подходил как нельзя лучше; он был прост для понимания и легок для программирования. Но, к сожалению, ADO не смог справиться с уровнем нагрузки более крупных узлов. Множество Web-страниц регулярно обращались к базе данных для получения одной и той же информации — обнаружилась жесткая зависимость ADO от наличия соединения с базой данных.

В соответствии с правилами ADO для того чтобы запросить информацию из базы данных, с ней сперва необходимо создать соединение. Как показано в листинге 1.1, время жизни соединения равно времени выполнения запроса.

Листинг 1.1. Классический доступ к базе данных с использованием ADO

```
Option Explicit

' Создание объекта Connection.
Dim conn as Object
Set conn = CreateObject("ADODB.Connection")
Dim sConn as String
sConn = "DSN=LocalServer;UID=someuser;Database=ADONET;"
conn.connectionString = sConn

' Открытие соединения.
conn.Open

' Создание объекта Command.
Dim cmd as Object
Set cmd = CreateObject("ADODB.Command")
cmd.ActiveConnection = conn
```

```
cmd.CommandText - "SELECT * FROM CUSTOMER"

' Выполнение запроса к базе данных - получение
' объекта Recordset.
Dim rs as Object
Set rэ - cmd.Execute()

' Работа с объектом Recordset.
Do While not rs.EOF

    Dim sRecord as String
    Dim field as Object
    For Each field in rs.Fields
    sRecord = sRecord & field.namefi ": " & field.value
    Next
    MsgBox sRecord
rs.MoveNext

Loop

' Соединение закрывается только после выполнения всех
' необходимых манипуляций с данными.
conn.Close
```

В большинстве случаев соединение с базой данных оставалось открытым на протяжении всего времени манипулирования с объектом Recordset. Это происходило даже при извлечении больших массивов информации. Кроме того, для обеспечения параллелизма в ADO было принято блокировать записи или целые страницы. В результате базы данных испытывали огромную нагрузку, свободные соединения быстро заканчивались, особенно для крупных Web-приложений. Проблема нехватки соединений решалась путем перевода объекта Recordset в отсоединенный режим. Несмотря на то что это было очень удачное решение, зачастую воспользоваться им могли только наиболее опытные разработчики. Повторно связать объект Recordset с соединением и согласовать его содержимое с содержимым базы данных было чрезвычайно сложно.

Относительно недавно компания Microsoft представила всему миру платформу .NET. Одной из наиболее важных частей этой платформы является библиотека ADO.NET — спросите, почему? Microsoft позволяет .NET-приложениям "общаться" со старыми системами посредством уровня взаимодействия с COM. В листинге 1.2 приведено простое C#-приложение, демонстрирующее пример использования ADO через уровень взаимодействия с COM.

Листинг 1.2. Использование ADO в управляемом коде

```
using System.Runtime.InteropServices;
using ADODB;

// Установка соединения с базой данных.
Connection conn = new Connection();
conn.Open("Provider=SQLOLEDB;Server=localhost;" +
          "Database=ADONET","someuser", "", 0);

// Выполнение запроса к базе данных.
Command cmd = new Command();
cmd.ActiveConnection = conn;
cmd.CommandText = "SELECT * FROM CUSTOMER";
object recaffected = null;
object prms = new object();
```

```
_Recordset rs = cmd.Execute(out recaffected, ref prms, 0);

// Вывод всех записей на консоль.
while (!rs.EOF)
{
    for (int x - 0; x < rs.Fields.Count; x++)
    {
      Console.Write(rs.Fields[x].Value.ToString() + ":");
    }
    Console.WriteLine(""); // Endline
    rs.MoveNext();
}

// Закрытие соединения.
conn.Close();
```

По своей функциональности эта программа практически аналогична той, что рассмотрена нами в предыдущем примере. Поскольку подобная схема использования ADO работает, при переходе на .NET многие предпочтут именно ее. К сожалению, это не позволяет решить основную проблему предшественников ADO.NET — преодолеть сложность работы с данными в отсоединенном режиме. С приходом ADO.NET все становится гораздо проще.

Преимущества ADO.NET

Согласно старой философии доступа к данным, беспокоиться об открытых соединениях с базой данных было незачем. Естественно, разработчики писали код, в котором "драгоценное" время соединения сводилось к минимуму, однако у них практически не было выбора, если возникала необходимость активного изменения данных.

Поддержка открытых соединений не вызывала никаких проблем при работе настольных приложений, так как было известно время удержания соединения пользователем. Ситуация изменилась несколько лет назад, когда весь мир начал ориентироваться на Web-разработки и распределенные вычисления. Предугадать время жизни соединения уже не представлялось возможным.

Поначалу разработчики попытались использовать старые методы доступа к данным, открывая соединения и помещая их в кэш, на то время, пока пользователь не закончит свою работу. К сожалению, Web-приложения не поддерживали такую возможность. Так, закрытие Web-обозревателя не приводило к уведомлению о том, что пользователь завершил работу. Единственным способом закрыть подобное соединение было его уничтожение после исчерпания лимита времени удержания помещенных в кэш данных. К счастью, этот метод оказался совсем не плох, и все были довольны, Однако затем, когда число обращений к среднестатистическому Web-узлу увеличилось с 1000 до 100000 или даже до миллиона раз в день, серверы баз данных резко замедлили свою работу, если не полностью вышли из строя. Подобно обиженному ребенку, они начинали отвергать все попытки установки соединения.

Протокол передачи гипертекстовых файлов (Hypertext Transfer Protocol — HTTP), на который была сделана ставка при разработке Web-клиентов, преподнес неприятный сюрприз — оказывается, старые методы доступа к данным были неэффективны в долгосрочном периоде. Протокол HTTP не имеет состояния, поэтому возникла необходимость в разработке метода доступа к данным, который бы копировал такое поведение. Поначалу была сделана попытка придумать механизмы, уменьшающие потребность в установке соединений с базой и блокировке страниц; но, к сожалению, полученный в результате код оказался сложным для написания и отладки. Несмотря на то что проблема могла быть решена за счет использования отсоединенных объектов

ADO Recordset, это также повышало сложность и громоздкость кода. Все понимали, что в данной ситуации нужно найти качественно новый подход.

ADO.NET позволяет продолжать быть "привязанным" к базе данных. В листинге 1.3 приведен фрагмент C#-программы, имитирующей логику рассмотренных ранее в этой главе примеров с использованием ADO,

Листинг 1.3. Пример ADO.NET-программы в стиле ADO

```
// Установка соединения с базой данных.
OleDbConnection conn = new
                       OleDbConnection("Provider=SQLOLEDB;" +
                                       "Server=localhost;" +
                                       "Database=ADONET;" +
                                       "UID=someuser;");
conn.Open();

// Выполнение запроса к базе данных.
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Выполнение команды для создания объекта DataReader.
OleDbDataReader reader = cmd.ExecuteReader();

// Вывод всех записей на консоль.
while (reader.Read())
{
    for (int x = 0; x < reader.FieldCount; x++)
    {
      Console.Write(reader.GetValue(x).ToString());
    }
    Console.WriteLine(""); // Вывод пустой строки.
}

// Закрытие соединения с базой данных.
conn.Close();
```

В этой форме ADO.NET не отличается от своих предшественников. Ключевой момент заключается в том, что ADO.NET сам по себе не решает проблем масштабируемости. Преобразовав старый ADO-код в код ADO.NET, вы ничуть не улучшите его. Важно понять, что ADO.NET изначально был разработан для работы в отсоединенном режиме доступа к данным. Несмотря на то что в нем реализован старый способ работы с базой данных в подсоединенном режиме, эта парадигма программирования больше не поощряется. Другими словами, ADO.NET изначально настраивает нас на работу в "отсоединенном" мире.

ADO.NET позволяет сократить "время жизни" соединения, как показано в листинге 1.4.

Листинг 1.4. Сокращение "времени жизни" соединения в ADO.NET

```
using System;
using System.Data;
using System.Data.OleDb;

// Создание объекта Connection.
OleDbConnection conn = new
                       OleDbConnection("Provider=SQLOLEDB;" +
                                       "Server=localhost;" +
                                       "Database=ADONET;" +
                                       "UID=someuser;");
```

```
// Установка соединения с базой данных.
conn.Open(J ;

// Создание объектов DataSet и Command.
OleDbCommand cmdAuthors = new OleDbCommand (
                                "SELECT COUNT(*)  FROM CUSTOMER",
                                conn);

// Выполнене команды.
int count = (int)cmdAuthors.ExecuteScalar();

// Закрытие соединения.
conn.Close();
```

Краткий курс ADO.NET

ADO.NET — это библиотека .NET-классов, которые позволяют подсоединяться к данным и манипулировать ими. Несмотря на то что большинство примеров в этой книге написаны на языке C#, общеязыковая исполняющая среда (Common Language Runtime — CLR) .NET позволяет писать практически аналогичный код на Visual Basic .NET, C++ с управляемыми расширениями или на любом другом управляемом языке[1].

Пространства имен ADO.NET

С целью разграничения функциональности классов ADO.NET они рассредоточены по различным пространствам имен. На рис. 1.1 показаны взаимоотношения этих пространств имен и содержащихся в них классов.

В ADO.NET пространства имен используются для отделения различных частей модели управляемого поставщика данных. Изображенное на рис. 1.1 пространство имен System. Data включает в себя общие структуры данных, не зависящие от конкретного поставщика. В него входит класс DataSet и целое семейство связанных с ним классов (DataTabLe, DataColumn, DataRow, DataRelation, Constraint и т.д.). Внутри пространства имен System. Data содержится пространство имен System. Data. Common, в которое входят базовые для управляемых поставщиков данных классы. Эти классы определяют стандарт, которому должен соответствовать управляемый поставщик данных ADO.NET. Другие управляемые поставщики разрабатываются путем создания набора классов, реализующих эти интерфейсы. Пространство имен YourProvider (дословный перевод — "ваш поставщик", *прим. ред.),* изображенное на рис. 1.1, является примером пространства имен пользовательского управляемого поставщика данных.

Структуры данных ADO.NET

ADO.NET поддерживает три способа непосредственного доступа к хранящейся в базе данных информации: через объекты Command, DataReader и DataSet.

Классы команд (SqlCommand для управляемого поставщика SQL и OleDbCommand для управляемого поставщика OLE DB) используются для получения результатов выполнения запросов к базе данных. Класс команды всегда реализует интерфейс

[1] **Изначально компания Microsoft создала компиляторы для Visual Basic,** JScript, C# и C++. Кроме этого, **сторонние** разработчики выпустили **компиляторы для таких языков, как** Eiffel, Perl, Python, Scheme, Smalltalk, **COBOL, Component Pascal, APL, Standard ML, Mercury и** Oberon.

IDbCommand, методы которого могут использоваться для получения скалярного результата (значения первого столбца первой строки результирующего набора данных) (метод IDbCommand.ExecuteScalar()) или выходных параметров хранимой процедуры (метод IDbCommand.ExecuteNonQuery()).

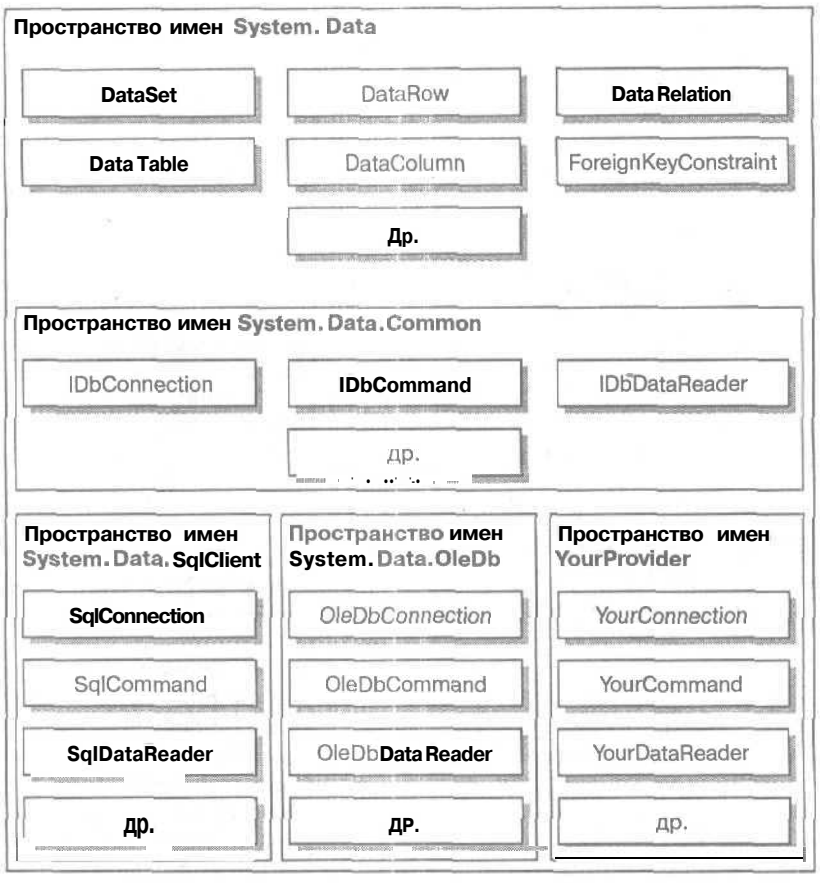


Рис. 1.1. Взаимоотношения между пространствами имен ADO.NET

По своим функциональным возможностям объект DataReader (классы SqlDataReader и OleDbDataReader) похож на объект ADO **Recordset.** Поскольку при чтении данных используется режим однонаправленного курсора, в ADO нужно быть очень осторожным, постоянно отслеживая вызовы метода MoveNext() и индикатор конца записи. Механизм чтения данных объекта **DataReader** является более устойчивым к ошибкам, позволяя за один шаг считать текущую запись в память и проверить наличие индикатора конца записи. Наиболее эффективно объект **DataReader** применяется при получении результатов выполнения запроса и их передаче на дальнейшую обработку. Стоит отметить, что компания Microsoft использует объект **DataReader** для заполнения объекта **DataSet** через объект **DataAdapter** в своих управляемых поставщиках данных. Пример использования объекта **DataReader** приведен в листинге 1.3.

Вне всякого сомнения, "центром вселенной" ADO.NET является новая структура данных — объект **DataSet,** напоминающий, на первый взгляд, объект **Rowset** в **OLE** DB или объект **Recordset** в ADO. Тем не менее **DataSet** — это гораздо более сложная

структура данных, позволяющая представлять в памяти почти любую реляционную модель. Объект **DataSet** содержит коллекцию таблиц, с которыми могут быть связаны экземпляры классов DataRelation, Constraint или даже ForeignKeyConstraint. Указанные взаимоотношения представлены на рис. 1.2.

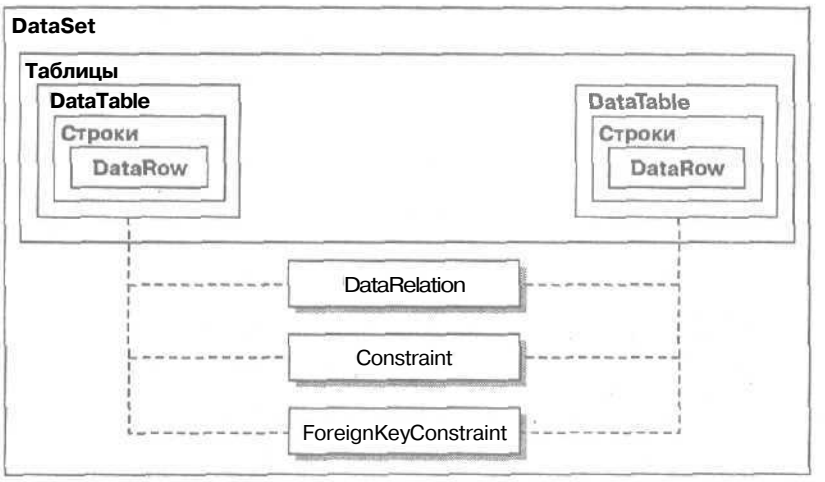


Рис. 1.2. Структура объекта ADO.NET DataSet

Для обычного разработчика это означает, что класс DataSet предоставляет средства моделирования реляционных данных в памяти. И если ADO.NET в целом предполагает работу с данными в отсоединенном режиме, объект **DataSet** позволяет "скрыть" этот факт и обращаться к информации так, как будто бы она находится в базе данных. Листинг 1.5 является, по сути, новой версией листинга 1.4, на этот раз с использованием объекта **DataSet.**

Листинг 1.5. Использование объекта *DataSet*

```
using System;
using System.Data;
using System.Data.OleDb;

// Создание объекта Connection.
OleDbConnection conn = new
                       OleDbConnection("Provider=SQLOLEDB;" +
                                       "Server=localhost;" +
                                       "Database=ADONET;" +
                                       "UID=someuser;");

// Установка соединения с базой данных.
conn.Open();

// Создание объектов DataSet and Command.
DataSet ds = new DataSet();
OleDbDataAdapter daAuthors = new OleDbDataAdapter(
                                 "SELECT * FROM CUSTOMER",
                                 conn);

// Заполнение объекта DataSet.
daAuthors.Fill(ds);
```

```
// Теперь мы можем закрыть соединение.
conn.Close() ;

// Извлечение таблицы из объекта DataSet.
DataTable tbl = ds.Tables["Table"];

// Обработка всех строк таблицы.
foreach (DataRow row in tbl.Rows)
{
    // Вывод на консоль значения всех полей заданной строки.
    foreach (Object val in row.ItemArray)
    {
      Console.Write(val.ToString());
    }
    Console.WriteLine(""); // Вывод пустой строки.
}
```

Этот код не сложнее того, в котором использовался объект **DataReader**. В нем применяется объект **DataAdapter** (в данном случае — экземпляр класса OleDbDataAdapter) и, в первый раз, объект **DataSet**.

Объект DataAdapter представляет собой специальный тип составной команды, содержащий четыре отдельных объекта **Command** — команды выборки, вставки, обновления и удаления данных. Несмотря на то что сейчас это выглядит совершенно не важным, вы оцените всю мощь объекта **DataAdapter** при рассмотрении механизма обновления с помощью объекта **DataSet** хранящейся в базе данных информации.

Возможно, вы думаете: "Использование объекта **DataSet** для обращения к данным в памяти пригодится только при чтении данных, но как же быть, если необходимо их изменить или добавить?". Не беспокойтесь, объект **DataSet** предоставляет возможность извлечения подмножества данных, которые были изменены, удалены или добавлены, что позволяет проверить их достоверность перед тем, как отразить изменения непосредственно в базе данных. Обычное использование объекта **DataSet** предусматривает добавление, изменение и удаление записей. Поскольку мы уже создали объект **DataAdapter,** нам остается связать с ним объект **CommandBuilder** и вызвать метод DataAdapter.Update() для обновления, добавления или удаления записей. Соответствующие операции выполняются с помощью команд, заданных свойствами объекта **DataAdapter** InsertCommand, UpdateCommand и DeleteCommand (листинг 1.6).

Листинг 1.6. Обновление базы данных с помощью объекта DataSet

```
using System;
using System.Data;
using System.Data.OleDb;

// Создание объекта Connection.
OleDbConnection conn = new
                       OleDbConnection("Provider=SQLOLEDB;" +
                                       "Server=localhost;" +
                                       "Database=ADONET;" +
                                       "UID=someuser;");

// Создание объектов DataSet и Command.
DataSet ds = new DataSet();
OleDbDataAdapter daAuthors =
       new OleDbDataAdapter("SELECT * FROM CUSTOMER", conn);

// Создание объекта OleDbCommandBuilder -
```

```
// оболочки объекта DataAdapter, поддерживающей
// динамическое генерирование команд
// обновления, добавления и удаления данных.
OleDbCommandBuilder bldr =
       new OleDbCommandBuilder(daAuthors);

// Заполнение объекта DataSet.
daAuthors.Fill(ds);

// Извлечение таблицы из объекта DataSet.
DataTable tbl = ds.Tables["Table"];

// Установка первичного ключа.
DataColumn[] colArr = new DataColumn[1];
colArr[0] = tbl.Columns[0];
tbl.PrimaryKey = colArr;

// Вставка новой строки.
object[] rowVals = new object[3];
rowVals[0] * Guid.NewGuid();
rowVals[1] = "Greg";
rowVals[2] = "Maddux";
DataRow insertedRow = tbl.Rows.Add(rowVals);

// Удаление строки.
tbl.Rows[0].Delete();

// Изменение полей в строке.
tbl.Rows[1].BeginEdit();
tbl.Rows[1]["FirstName"] = "New Name";
tbl.Rows[1].EndEdit();

// Сохранение изменений в базе данных.
conn.Open();
daAuthors.Update(ds);
```

Отметим несколько важных моментов. Во-первых, мы нигде не устанавливали значение свойства InsertCommand, UpdateCommand или DeleteCommand. Вместо этого был создан экземпляр класса OleDbCommandBuilder. При создании объекта CommandBuilder конструктору передается объект DataAdapter. Объект CommandBuilder регистрирует себя в качестве "слушателя" объекта DataAdapter и, при необходимости, на лету создает команды обновления, удаления или вставки данных. К тому же мы установили значение свойства PrimaryKey для того, чтобы объект CommandBuilder знал правила создания объектов Command. Обратите внимание: после выполнения запроса к базе данных (в методе Fill) соединение было закрыто, затем мы заново открыли его непосредственно перед выполнением метода Update().

Объектная модель управляемого поставщика данных ADO.NET

В первоначальном выпуске **ADO.NET** содержатся два управляемых поставщика данных: **OLE** DB и SQL Server. Управляемый поставщик **SQL** Server позволяет подключаться **к базе** данных Microsoft SQL Server, используя для этого ее "родные" **механизмы.** Для доступа к базам данных посредством абстрактного уровня **OLE DB** в **ADO.NET** предусмотрен управляемый поставщик OLE DB. Технология **OLE DB** позволяет подключаться к огромному числу различных хранилищ данных, таких, как

SQL Server, Oracle, DB2, Access, dBase, FoxPro, и даже к текстовым файлам с символами-разделителями.

Управляемый поставщик OLE DB препятствует использованию поставщика OLE DB для ODBC. В связи с этим компания Microsoft выпустила отдельный управляемый поставщик ODBC, позволяющий получать доступ к любым источникам данных, для которых предусмотрен драйвер ODBC. Принятие решения об использовании существующего поставщика OLE DB или драйвера ODBC зависит от конкретной базы данных. В некоторых случаях быстрее работает OLE DB, а в некоторых — ODBC.

Получить доступ к базе данных SQL Server можно как с помощью управляемого поставщика SQL Server, так и с помощью управляемого поставщика OLE DB. Возникает вполне законный вопрос: "А какой из них лучше?". После того как я продемонстрирую вам отличия этих поставщиков, вы сами сможете дать на него ответ. Ниже приведены два примера выполнения одной и той же операции: в листинге 1.7 используется управляемый поставщик OLE DB, а в листинге 1.8 — управляемый поставщик SQL Server.

Листинг 1.7. Доступ к базе данных SQL Server с помощью управляемого поставщика *OleDb*

```
// Управляемый поставщик OLE DB.

using System;
using System.Data;
using System.Data.OleDb;

// Создание объекта Connection.
OleDbConnection  conn =
                 new OleDbConnection("Provider=SQLOLEDB;" +
                                     "Server=localhost;" +
                                     "Database=ADONET;" +
                                     "UID=someuser;");

// Создание объектов DataSet и Command.
DataSet ds = new DataSet();
OleDbDataAdapter  daAuthors = new OleDbDataAdapter(
                                    "SELECT * FROM CUSTOMER",
                                    conn);

// Заполнение объекта DataSet.
daAuthors.Fill(ds);

// Извлечение таблицы из объекта DataSet.
DataTable tbl = ds.Tables["Table"];

// Обработка всех строк таблицы.
foreach( DataRow row in tbl.Rows)
{
    // Вывод на консоль значений всех полей заданной строки.
    foreach (Object val in row.ItemArray)
    {
        Console.Write(val.ToString());
    }

    Console.WriteLine(""); // Вывод пустой строки.
}
```

Листинг 1.8. Доступ к базе данных SQL Server с помощью управляемого поставщика SqlClient

```
// Управляемый поставщик SQL Server.

using System;
using System.Data;
using System.Data.SqlClient;

// Создание объекта Connection.
SqlConnection  conn = new SqlConnection();
conn.ConnectionString * "Server=localhost;" +
                        "Database=ADONET;" +
                       "UserID=someuser;";

// Создание объектов DataSet и Command.
DataSet ds = new DataSet();
SqlDataAdapter  daAuthors = new SqlDataAdapter(
                                "SELECT * FROM CUSTOMER",
                                conn);

// Заполнение объекта DataSet.
daAuthors.Fill(ds);

// Извлечение таблицы из объекта DataSet.
DataTable tbl = ds.Tables["Table"];

// Обработка всех строк таблицы.
foreach( DataRow row in tbl.Rows)
{
    // Вывод на консоль значений всех полей заданной строки.
    foreach (Object val in row.ItemArray)
    {
       Console.Write(val.ToString());
    }
    Console.WriteLine(""); // Вывод пустой строки.
}
```

Единственное изменение, сделанное нами в SQL-версии кода помимо переименования классов, заключается в том, что нам больше не нужно определять поставщик данных в строке соединения. Синтаксис строк соединения OLE DB напоминает синтаксис строк соединения SQL Server, хотя они не полностью идентичны. Как вы видите, поправки, необходимые для замены одного поставщика данных другим, в целом несущественны. Поскольку мир ADO.NET DataSet-центричен, большая часть кода не будет зависеть от используемого поставщика.

Общее правило гласит; выбирайте управляемый поставщик SQL Server, если вам точно известно, что в качестве хранилища данных всегда будет использоваться SQL Server. Теоретически управляемый поставщик SQL Server должен работать лучше. Для доступа к базе данных он использует не OLE DB, а протокол TDS (Tabular Data Stream — поток табличных данных), позволяющий улучшить производительность.

Тем не менее мои собственные тесты показали, что производительность двух управляемых поставщиков данных практически одинакова. Таким образом, настоящая причина, по которой я советую использовать поставщик данных SQL Server, заключается в предпочтении иметь функциональность, специфичную только для SQL Server (например, получение результатов запросов в формате XML), или же работать с типами данных SQL Server вместо типов OLE DB. Как мы увидим в главе 9, управляемый поставщик SQL Server предоставляет набор классов, предназначенных для оптимизации

процесса получения результатов запросов в формате XML для их последующей обработки с помощью XML-классов .NET.

В ADO.NET классы управляемых поставщиков взаимодействуют с хранилищем данных посредством базовых методов установки соединения и выполнения команд. На рис. 1.3 показана взаимосвязь между классами соединения, команды и параметров. Создавая объекты **Connection, Command и Parameter,** управляемые поставщики позволяют подключаться к базе данных и производить с ней какие-либо действия.

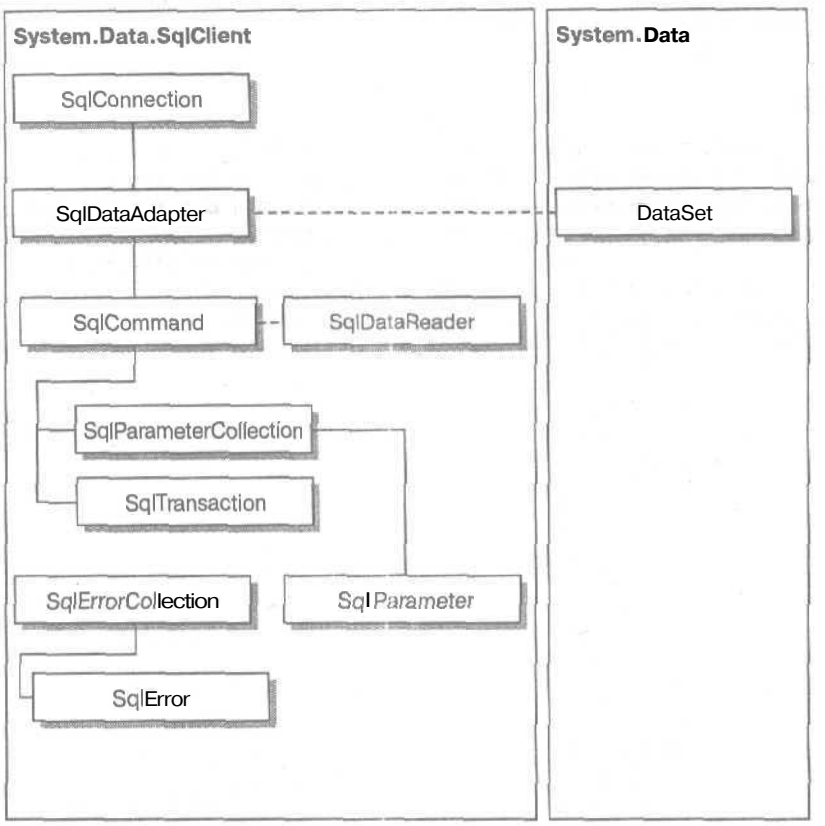


Рис. 1.3. Взаимоотношения между классами поставщика SqlClient

Классы поставщиков данных ADO.NET располагаются в пространстве имен System.Data. В пространстве имен System.Data.Common содержатся базовые классы управляемых поставщиков, которые нельзя создавать; как правило, это абстрактные классы, определяющие базовую функциональность управляемых поставщиков, следовательно вы никогда не будете использовать данные классы напрямую. Каждый управляемый поставщик имеет свое собственное пространство имен. Так, управляемый поставщик SQL Server находится в пространстве имен System.Data.SqlClient, управляемый поставщик OLE DB — в пространстве имен System.Data.OleDb, управляемый поставщик Microsoft Oracle — в пространстве имен System.Data.OracleClient и, наконец, управляемый поставщик ODBC — в пространстве имен System.Data.Odbc. Я не удивлюсь, если к моменту выхода этой книги компании Sybase и IBM также напишут свои собственные управляемые поставщики данных.

Глава 9

ADO.NET и XML

В этой главе...

.NET и XML

Класс DataSet и XML

Класс XmlDataDocument

Существуют различные мнения относительно важности XML в разработке программного обеспечения. Компания Microsoft разработала собственную стратегию .NET, направленную на использование XML и Web-служб в качестве "клея", обеспечивающего такое взаимодействие между компаниями, которое не зависит от используемых в них операционных систем и механизмов хранения данных. Поскольку XML — это данные, а ADO.NET — механизм работы с данными в .NET, тот факт, что ADO.NET поддерживает работу с XML-документами наравне с остальными данными, выглядит вполне естественно. В этом аспекте библиотека ADO.NET является тесно интегрированной с инфраструктурой XML .NET.

.NET и XML

ADO.NET-стратегию Microsoft в отношении XML необходимо рассматривать с двух сторон: с одной стороны, Microsoft стремится предоставить те же инструменты для доступа к XML-содержимому, которые используются для доступа к информации, хранящейся в базе данных, с другой — обеспечить в ADO.NET средства преобразования информации базы данных в формат XML. Кстати сказать, название ADO.NET не совсем корректное, так как ADO.NET не является прямым "наследником" ADO. Поскольку ADO.NET — всего лишь механизм работы с данными в .NET, то не кажется ли вам, что с этой точки зрения XML представляет собой просто еще один формат данных?

Во времена ADO/OLE DB для обработки информации из некоего источника данных (например, базы данных) необходимо было создать поставщик OLE DB, который довольно сложен и запутан. В .NET для доступа к информации из источника, отличного от базы данных, можно либо создать управляемый поставщик, либо воспользоваться преимуществами тесной интеграции ADO.NET с XML.

Класс DataSet и XML

Класс DataSet имеет встроенную поддержку XML. Поскольку манипуляция с данными и интеграция с XML являются одинаково важными аспектами функциональности класса DataSet, становится ясно, что они были заложены в него еще на этапе проектирования.

Преобразование данных объекта DataSet в формат XML

Получение XML-представления объекта DataSet — это довольно простая задача. Единственным нетривиальным моментом здесь является "тонкая" настройка выходного

Часть III

Практическое использование ADO.NET

Глава 9. ADO.NET и XML
Глава 10. Привязка данных в ADO.NET
Глава 11. Масштабируемость и производительность приложений ADO.NET

```
// для заполнения объекта DataSet.
dataAdapter.Fill(dataSet, "Products");

// Определение переменной для упрощения доступа к таблице.
DataTable prodTable = dataSet.Tables["Products"];

// Добавление сведений о новом товаре.
DataRow newRow = prodTable.NewRow();
newRow.BeginEdit();
newRow["Description"] = "Home Base Broom";
newRow["Vendor"] = "Smith's Hardware";
newRow["Cost"] = 12.54;
newRow["Price"] = 15.00;
newRow.EndEdit();
prodTable.Rows.Add(newRow);

try
{
  // Обновление базы данных.
  dataAdapter.Update(prodTable);
  Console.Write("Successfully Updated the Database");
}
catch (SqlException ex)
{
  Console.WriteLine(ex.Message);
}
```

Обратите внимание на добавление в команду INSERT дополнительного оператора SELECT, который используется для извлечения из базы данных значения последнего созданного идентификатора и его сохранения в параметре @ProductID. Поскольку в результате выполнения команды значение параметра будет изменено, его необходимо определить как выходной параметр. Благодаря этому после вставки строки она будет иметь корректное значение идентификационного столбца. Несмотря на то что идентификационный столбец можно определить как автоинкрементный в объекте DataTable (см. главу 5, "Создание объекта DataSet"), вы можете воспользоваться приведенным выше примером для перепоручения этой функции базе данных.

Единственным неудобством, которое возникает при использовании описанного механизма, является невозможность обращения к новой записи из другой таблицы до тех пор, пока не будет обновлена база данных. Т.е. с практической точки зрения автоинкрементный столбец может оказаться лучшим решением. К сожалению, если обновление базы данных произойдет за пределами кода, подход с использованием автоинкрементного столбца окажется неэффективным. Более подробно автоинкрементные столбцы рассматриваются в главе 5.

Резюме

Несмотря на то что доступ к данным в отсоединенном режиме предоставляет огромные преимущества, за него также следует платить. Первой задачей, с которой необходимо справиться разработчику, является реализация параллелизма при доступе к отсоединенным данным. При этом самое сложное решение, как правило, заключается в выборе оптимальной модели параллелизма для конкретного проекта или приложения. Кроме того, в этой главе мы познакомились с "обратной стороной медали" объекта CommandBuilder.

от хода выполнения транзакции она либо фиксируется, либо отменяется. Следует отметить, что самая важная часть приведенного выше кода заключается именно в присваивании транзакции свойству DataAdapter.SelectCommand.Transaction (на первый взгляд, это может оказаться совершенно неочевидным). Дело в том, что когда объект CommandBuilder создает команды INSERT, UPDATE и DELETE, он копирует все настройки команды DataAdapter.SelectCommand.

Получение идентификатора новой строки от базы данных SQL Server

Бывают ситуации, в которых создание идентификатора новой строки в таблице возлагается на базу данных. С этой целью база данных SQL Server поддерживает поле IDENTITY и серверную переменную @@IDENTITY, хранящую значение последнего созданного идентификатора. Для того чтобы воспользоваться этой функциональностью в ADO.NET, необходимо выполнить следующие действия (листинг 8.17).

Листинг 8.17. Получение идентификатора новой строки от базы данных SQL Server

```
...

// Создание объекта DataAdapter.
SqlDataAdapter dataAdapter -
          new SqlDataAdapter("SELECT * FROM PRODUCT", conn);

// Создание команды INSERT.
string insQry - "";
insQry += ''INSERT INTO PRODUCT( ";
insQry += "  Description, Vendor, Cost, Price) ";
insQry += ''VALUES ( @Description, ";
insQry += "  @Vendor, @Cost, @Price) ";
insQry += ''\n";
insQry += "SELECT @ProductID = @@IDENTITY";

SqlCommand insCmd = conn.CreateCommand0 ;
insCmd. CommandText * insQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection insParams = insCmd.Parameters;

// Определение параметров.
insParams.Add("@ProductID", SqlDbType.Int, 0, "ProductID");
insParams["@ProductID"].Direction -
                                ParameterDirection.Output;
insParams.Add("@Description", SqlDbType.NVarChar, 255,
              "Description");
insParams.Add("@Vendor", SqlDbType.NVarChar, 255, "Vendor");
insParams.Add("@Cost", SqlDbType.Money, 0, "Cost");
insParams.Add("@Price", SqlDbType.Money, 0, "Price");

// Установка значения свойства InsertCommand
// объекта DataAdapter.
dataAdapter.InsertCommand = insCmd;

// Создание объекта DataSet.
DataSet dataSet = new DataSet();

// Использование объекта DataAdapter
```

```
DataRow delRow = custTable.Rows[1];
delRow.Delete();

// Создание объектов CommandBuilder для генерирования
// команд вставки, изменения и удаления данных.
SqlCommandBuilder custBldr = new SqlCommandBuilder(custDA);
SqlCommandBuilder invBldr » new SqlCommandBuilder(invDA);

// Обновление базы данных.
SqlTransaction tx - null;
try
(
  // Для того чтобы начать транзакцию/
  // необходимо открыть соединение.
  conn.Open{)/'

  // Создание транзакции для команды SELECT.
  // Объект CommandBuilder распространит
  // транзакцию на остальные команды.
  tx • conn.BeginTransaction(IsolationLevel.Serializable);
  invDA, SelectCommand.Transaction = tx;
  custDA.SelectCommand.Transaction = tx;

  // Удаление информации из базы данных.
  // Используем обратный порядок - сначала
  // обновляется дочерняя таблица.
  invDA.Update(invTable.GetChanges(DataRowState.Deleted));
  custDA.Update(custTable.GetChanges(DataRowState.Deleted));
  // Добавление и изменение информации в базе данных.
  // Используем прямой порядок - сначала
  // обновляется родительская таблица.
  custDA.Update(custTable.GetChanges(DataRowState.Added I
                                     DataRowState.Modified));
  invDA.Update(invTable.GetChanges(DataRowState.Added I
                                   DataRowState.Modified));

  // На данном этапе транзакцию можно зафиксировать.
  tx.Commit();
}
catch (SqlException ex)
{
  Console.WriteLine("Sql Error: {0}", ex.Message);

  // Отменить транзакцию при возникновении ошибки.
  if (tx != null) tx.Rollback();
}
finally
{
  if (conn.State != ConnectionState.Closed)
  t
    conn.Close();
  }
}
```

Для того чтобы провести все операции обновления в рамках транзакции, мы создаем локальную транзакцию соединения, после чего присваиваем ее свойству DataAdapter. Selectcommand. Transaction каждого объекта DataAdapter. В зависимости

Как показано в приведенном выше фрагменте кода, при выполнении операции удаления информации из базы данных используется обратный порядок обновления содержимого таблиц, а при выполнении операций добавления и изменения — прямой порядок. Это можно реализовать, передав методу Update объект DataTable, содержащий только определенные (удаленные, добавленные или измененные) строки. Поскольку метаданные уровня столбцов для полученных в результате вызова метода Getchanges объектов DataTable соответствуют первоначальным таблицам, объект CommandBuilder будет работать как ни в чем не бывало.

Использование локальных транзакций для обновления базы данных

Что произойдет, если при выполнении кода из предыдущего примера случится ошибка? Останется ли база данных в непротиворечивом состоянии? Да, если для связывания операций обновления будут использованы локальные транзакции. Локальные транзакции позволяют провести отмену внесенных в базу данных изменений в случае возникновения ошибки на стороне клиента. Пример использования транзакций приведен в листинге 8.16.

Листинг 8.16. Использование транзакций для обновления базы данных

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];
DataTable invTable = dataSet.Tables["Invoices"];

// Определение отношения между таблицами.
DataColumn custCustIDColumn =
               custTable.Columns["CustomerID"];
DataColumn invCustIDColumn = invTable.Columns["CustomerID"];
dataSet.Relations.Add("rel", custCustIDColumn,
                      invCustlDColumn, true);

// Добавление сведений о новом клиенте.
DataRow newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Remlinger";
newRow["FirstName"] = "Mike";
newRow["Address"] = "10 Aaron Way";
newRow["City"] = "Atlanta";
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] = "(404) 543-8765";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Изменение сведений о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["Address"] = "53 Peachtree Center";
oldRow["Zip"] = "30342";

// Изменение информации о счете.
DataRow oldInv = oldRow.GetChildRows("rel") [0];
oldInv["Terms"] = "Net 10th";

// Удаление клиента.
```

```
    invDA.Update(invTable);
    custDA.Update(custTable);
}
catch (SqlException ex)
{
    Console.WriteLine("Sql Error: {0}", ex.Message);
}
```

Легко заметить, что приведенный выше код напоминает код обновления базы данных на основе объекта **DataSet** с одной таблицей. Разница заключается в том, что для каждой таблицы создается собственный объект **DataAdapter** и **CommandBuilder**.

Порядок вызова метода DataAdapter. Update () очень важен. Как видите, сначала мы обновляем строки в дочерней таблице. Это связано с тем, что проводится удаление клиента, а определенное отношение **DataRelation** поддерживает каскадные удаления. Следует отметить, что если каскадные удаления поддерживаются как базой данных, так и объектом DataSet, то при обновлении базы данных могут возникнуть ошибки, связанные с тем, что объект DataSet попытается удалить дочерние элементы, которые уже были удалены базой данных при удалении родительских строк. Обновление "в обратном порядке" (начиная с родительской таблицы) позволяет обойти эту проблему, однако вызывает дополнительные трудности при добавлении новых родительских и дочерних строк. Дело в том, что если сначала будет обновляться дочерняя таблица, имеющая ограничения по внешнему ключу, то добавление новых дочерних строк будет требовать наличия соответствующих строк в родительской таблице. Выходом из этой ситуации является отдельное обновление, вставка и удаление информации из базы данных. В листинге 8.15 показан расширенный код, обеспечивающий корректное выполнение операции добавления информации в базу данных.

Листинг 8.15. Добавление информации в базу данных при использовании объекта *DataSet*, состоящего из нескольких таблиц

```
...

// Создание объектов CommandBuilder для генерирования
// команд удаления, изменения и вставки данных.
SqlCommandBuilder custBldr - new SqlCommandBuilder(custDA);
SqlCommandBuilder invBldr - new SqlCommandBuilder(invDA);

// Обновление базы данных.
try
{
    // Удаление информации из базы данных.
    // Используем обратный порядок - сначала
    // обновляется дочерняя таблица.
    invDA.Update(invTable.GetChanges(DataRowState.Deleted));
    custDA.Update(custTable.GetChanges(DataRowState.Deleted));

    // Добавление и изменение информации в базе данных.
    // Используем прямой порядок - сначала
    // обновляется родительская таблица.
    custDA.Update(custTable.GetChanges(DataRowState.Added |
                                       DataRowState.Modified));
    invDA.Update(invTable.GetChanges(DataRowState.Added |
                                     DataRowState.Modified));
}
catch (SqlException ex)
{
    Console.WriteLine("Sql Error: {0}", ex.Message);
}
```

```
SqlDataAdapter invDA =
        new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание объекта DataSet.
DataSet dataSet = new DataSet();

// Использование объекта DataAdapter
// для заполнения объекта DataSet.
custDA.Fill(dataSet, "Customers");
invDA.Fill(dataSet, "Invoices");

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];
DataTable invTable = dataSet.Tables["Invoices"];

// Определение отношения.
DataColumn custCustIDColumn =
                    custTable.Columns["CustomerID"];
DataColumn invCustIDColumn = invTable.Columns["CustomerID"];
dataSet.Relations.Add("rel", custCustIDColumn,
                     invCustIDColumn, true);

// Добавление сведений о новом клиенте.
DataRow newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Remlinger";
newRow["FirstName"] = "Mike";
newRow["Address"] = "10 Aaron Way";
newRow["City"] = "Atlanta";
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] = "(404) 543-8765";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Изменение сведений о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["Address"] = "53 Peachtree Center";
oldRow["Zip"] = "30342";

// Изменение информации о счете.
DataRow oldInv = oldRow.GetChildRows("rel")[0];
oldInv["Terms"] = "Net 10th";

// Удаление клиента.
DataRow delRow = custTable.Rows[1];
delRow.Delete();

// Создание объектов CommandBuilder для генерирования
// команд удаления, изменения и вставки данных.
SqlCommandBuilder custBldr = new SqlCommandBuilder(custDA);
SqlCommandBuilder invBldr = new SqlCommandBuilder(invDA);

// Обновление базы данных.
try
{
  // Сначала обновляется дочерняя таблица, потому что
  // мы осуществляем удаление данных.
```

```
newRow["City"] = "Atlanta";
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] = "(404) 543-8765";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Удаление клиента.
DataRow delRow = custTable.Rows[1];
delRow.Delete();

// Изменение сведений о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["MiddleName"] = "Andrew";

try
{
  // Обновление Сазы данных.
  dataAdapter.Update(custTable);
  Console.Write("Successfully Updated the Database");
}
catch (SqlException ex)
{
  Console.WriteLine(ex.Message);
}
```

Этот пример практически идентичен предыдущему примеру улучшенной реализации оптимистического параллелизма. Однако не используется поле Stamp, позволяющее удостовериться в том, что строка не изменилась с момента ее извлечения из базы данных. Обновление и удаление строк всегда происходит без учета возможности возникновения исключений.

Несколько наиболее распространенных вопросов, связанных с обновлением данных

В своей повседневной работе вы будете сталкиваться со множеством различных аспектов обновления данных в отсоединенном режиме. Я хочу обратить ваше внимание на наиболее распространенные вопросы, связанные с обновлением данных, и предложить возможные способы их решения.

Работа с объектами DataSet, состоящими из нескольких таблиц

В большинстве случаев вы будете иметь дело с объектами **DataSet**, состоящими из нескольких таблиц. Чем же отличается обновление базы данных на основе объекта **DataSet,** содержащего одну таблицу, от обновления базы данных на основе объекта **DataSet,** содержащего несколько таблиц? Как показано в листинге 8.14, основное различие заключается в том, что каждой таблице объекта **DataSet** должен соответствовать собственный объект **DataAdapter.**

Листинг 8.14. Удаление информации из базы данных при использовании объекта *DataSet*, состоящего из нескольких таблиц

```
...

// Создание объектов DataAdapter.
SqlDataAdapter custDA =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);
```

```
// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection updParams = updCmd.Parameters;

// Определение параметров.
updParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
updParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
updParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
updParams["@MiddleName"].IsNullable = true;
updParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
updParams["@LastName"].IsNullable = true;
updParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
updParams["@Address"].IsNullable = true;
updParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
updParams["@Apartment"].IsNullable = true;
updParams.Add("@City", SqlDbType.NVarChar, 50, "City");
updParams["@City"].IsNullable = true;
updParams.Add("@State", SqlDbType.NChar, 2, "State");
updParams["@State"].IsNullable = true;
updParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
updParams["@Zip"].IsNullable = true;
updParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
updParams["@HomePhone"].IsNullable = true;
updParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
updParams["@BusinessPhone"].IsNullable = true;
updParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
updParams["@DOB"].IsNullable = true;
updParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
updParams["@Discount"].IsNullable = true;

// Установка значения свойства UpdateCommand
// объекта DataAdapter.
dataAdapter.UpdateCommand = updCmd;

// Создание объекта DataSet.
DataSet dataSet = new DataSet();

// Использование объекта DataAdapter для
// заполнения объекта DataSet.
dataAdapter.Fill(dataSet, "Customers");

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];

// Добавление сведений о новом клиенте.
DataRow newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Remlinger";
newRow["FirstName"] = "Mike";
newRow["Address"] = "10 Aaron Way";
```

```
insParams.Add("@City", SqlDbType.NVarChar, 50, "City") ;
insParams["@City"].IsNullable = true;
insParams.Add("@State", SqlDbType.NChar, 2, "State");
insParams["@State"].IsNullable = true;
insParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
insParams["@Zip"].IsNullable = true;
insParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
insParams["@HomePhone"].IsNullable = true;
insParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
insParams["@BusinessPhone"].IsNullable = true;
insParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
insParams["@DOB"].IsNullable = true;
insParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
insParams["@Discount"].IsNullable = true;

// Установка значения свойства InsertCommand
// объекта DataAdapter.
dataAdapter.InsertCommand = insCmd;

// Определение команды DELETE.
string delQry =
delQry = @"DELETE FROM CUSTOMER
           WHERE CustomerID * @CustomerID";

SqlCommand delCmd = conn.CreateCommand();
delCmd.CommandText = delQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection delParams - delCmd.Parameters;

// Определение параметров,
delParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");

// Установка значения свойства DeleteCommand
// объекта DataAdapter,
dataAdapter.DeleteCommand = delCmd;

// Определение команды UPDATE.
string updQry = "";
updQry += "UPDATE CUSTOMER SET ";
updQry += "  CustomerlD = @CustomerID, ";
updQry += "  FirstName = @FirstName, ";
updQry += "  LastName = @LastName, ";
updQry += "  MiddleName = @MiddleName, ";
updQry += "  Address = @Address, ";
updQry += "  Apartment = @Apartment, ";
updQry += "  City = @City, ";
updQry += "  State = @State, Zip = @Zip, ";
updQry += "  HomePhone = @HomePhone, ";
updQry += "  BusinessPhone = @BusinessPhone, ";
updQry += "  DOB = @DOB, ";
updQry += "  Discount = @Discount ";
updQry += "WHERE CustomerlD = @CustomerID";

SqlCommand updCmd = conn. CreateCommand ();
updCmd.CommandText = updQry;
```

Следует отметить, что в том случае, когда необходимо реализовать отсоединенный пессимистический параллелизм, рекомендуется использовать шаблон "извлечь для редактирования/вернуть после окончания редактирования".

Реализация деструктивного параллелизма

Включив в материал данной главы раздел о деструктивном параллелизме, я немного слукавил. На самом деле, деструктивный параллелизм — это вообще не параллелизм. "Последний побеждает" — вот основной принцип такого подхода к поддержке параллельного доступа к данным. Тем не менее существуют ситуации, в которых подобная модель параллелизма является наиболее приемлемой. К сожалению, объект CommandBuilder не поддерживает деструктивный параллелизм, а следовательно, его необходимо реализовывать самостоятельно, как показано в листинге 8.13.

Листинг 8.13. Реализация деструктивного параллелизма

```
...

// Создание объекта DataAdapter.
SqlDataAdapter dataAdapter =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Создание команды INSERT.
string insQry =
insQry = @"INSERT INTO CUSTOMER(
             CustomerID, FirstName, LastName,
             MiddleName, Address, Apartment,
             City, State, Zip, HomePhone,
             BusinessPhone, DOB, Discount)
           VALUES ( @CustomerID, @FirstName,
             @LastName, @MiddleName, @Address,
             @Apartment, @City, @State, @Zip,
             @HomePhone, @BusinessPhone, @DOB,
             @Discount )";

SqlCommand insCmd = conn.CreateCommand();
insCmd.CommandText = insQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection insParams = insCmd.Parameters;

// Определение параметров.
insParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
insParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
insParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
insParams["@MiddleName"].IsNullable = true;
insParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
insParams["@LastName"].IsNullable = true;
insParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
insParams["@Address"].IsNullable = true;
insParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
insParams["@Apartment"].IsNullable = true;
```

```
// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection updParams = updCmd.Parameters;

// Определение параметров,
updParams.Add("@CustomerID", SqlDbType.UniqueIdentifier,
              0, "CustomerID");
updParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
updParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
updParams["@MiddleName"].IsNullable = true;
updParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
updParams["@LastName"].IsNullable = true;
updParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
updParams["@Address"].IsNullable = true;
updParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
updParams["@Apartment"].IsNullable = true;
updParams.Add("@City", SqlDbType.NVarChar, 50, "City");
updParams["@City"].IsNullable = true;
updParams.Add("@State", SqlDbType.NChar, 2, "State");
updParams["@State"].IsNullable = true;
updParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
updParams["@Zip"].IsNullable = true;
updParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
updParams["@HomePhone"].IsNullable = true;
updParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
updParams["@BusinessPhone"].IsNullable - true;
updParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
updParams["@DOB"].IsNullable = true;
updParams.Add("@Discount", SqlDbType.Float, 8,
              "Discount");
updParams["@Discount"].IsNullable = true;

// Установка значения свойства UpdateCommand
// объекта DataAdapter.
dataAdapter.UpdateCommand = updCmd;

// Обновление базы данных.
try
{
  dataAdapter.Update(custTable);
  Console.WriteLine("Finished updating the database");
}

catch (SqlException ex)
{
  Console.WriteLine("SqlException: {0}", ex.Message);
}

// Сброс объекта DataSet.
dataSet.Reset();
```

```
            /* На этом этапе транзакцию
               можно зафиксировать. */
            COMMIT TRAN";

// Уточнение запроса - указание
// идентификационного номера клиента.
getSQL = string.Format(getSQL, custID);

// Создание объекта DataAdapter.
SqlDataAdapter dataAdapter = new SqlDataAdapter(getSQL,
                                                conn);

// Создание объекта DataSet.
DataSet dataSet = new DataSet() ;

// Использование объекта DataAdapter для
// заполнения объекта DataSet.
try
{
  dataAdapter.Fill(dataSet, "Customers");
}
catch (SqlException ex)
{
  Console.WriteLine("SqlException: {0}", ex.Message);
  return;
}

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables[0];

// Обновление сведений о клиенте.
DataRow customer = custTable.Rows[0];
customer["Address"] = "55 Peachtree Center";
customer["Zip"] = "30312";

// Создание команды обновления.
// В идеале здесь нужно было бы
// использовать хранимую процедуру.
string updQry;
updQry = @"UPDATE CUSTOMER SET
             CustomerID = @CustomerID,
             FirstName = @FirstName/
             LastName = @LastName,
             MiddleName = @MiddleName,
             Address = @Address,
             Apartment = @Apartment,
             City = @City,
             State = @State, Zip = @Zip,
             HomePhone = @HomePhone,
             BusinessPhone = @BusinessPhone,
             DOB = @DOB,
             Discount = @Discount,
             CheckedOut = 0
           WHERE Stamp = @Stamp AND
                 CustomerID = @CustomerID
           AND CheckedOut = 1";

SqlCommand updCmd = conn.CreateCommand();
updCmd.CommandText = updQry;
```

влеченную для редактирования и получить ее. Поскольку все операции с базой данных заключены в транзакции, ситуация, в которой запись клиента уже помечена, но еще не извлечена из базы данных, исключается.

После внесения изменений запись клиента нужно возвратить обратно в базу данных. При этом она будет обновлена, а флаг "извлечена для редактирования" должен быть сброшен (установлен в ноль или false). Обратная операция не требует использования транзакции, поскольку она атомарна (все изменения базы данных выполняются в пределах одного обращения). В завершение необходимо сбросить объект DataSet, для того чтобы разрешить доступ к записи клиента. Пример реализации пессимистического параллелизма приведен в листинге 8.12.

Листинг 8.12. Реализация пессимистического параллелизма

```
...

// Код T-SQL для извлечения из базы данных записи
// клиента (в идеале здесь нужно было бы использовать
// хранимую процедуру). CustID представляет собой
// идентификационный номер клиента.

string getSQL;
getSQL = @"/* Начало транзакции. */
           BEGIN TRAN

           DECLARE @isCheckedOut bit
           DECLARE @CustID varchar(38)
           SELECT @CustID = '{0}';

           /* Проверим, не является ли запись клиента
              извлеченной для редактирования. */
           SELECT @isCheckedOut = CheckedOut
             FROM Customer
             WHERE CustomerID = @CustID

           IF @isCheckedOut = 1
           BEGIN
             RAISERROR ('Customer already checked out',16,1)
             ROLLBACK TRAN
             RETURN
           END

           /* Отметить запись как извлеченную
              для редактирования. */
           UPDATE Customer SET CheckedOut = 1
             WHERE CustomerID = @CustID

           /* Извлечение записи клиента из базы данных. */
           SELECT CustomerlD, FirstName, LastName,
                  MiddleName, Address, Apartment,
                  City, State, Zip, HomePhone,
                  BusinessPhone, DOB, Discount,
                  Stamp
             FROM Customer
             WHERE CustomerlD = @CustID
```

```
            Console.WriteLine("  Row: State={0}, ID={1}",
                              row["CustomerID"],
                              row.RowState);
          }
        }
        else
        {
          Console.WriteLine("ConcurrencyViolation: {0}",
                            ex.Message);
          Console.WriteLine("  Row: State={0}, ID={1}",
                            row["CustomerID"],
                            row.RowState);
        }

        // Сброс свойства UpdateCommand.
        dataAdapter.UpdateCommand = oldUpdateCmd;
      }
      else
      {

        // Вывод сообщения об остальных типах
        // нарушения параллелизма.
        Console.WriteLine("ConcurrencyViolation: {0}",
                          ex.Message);
        Console.WriteLine("  Row: State={0}, ID={1}",
                          row["CustomerID"],
                          row.RowState);
      }
    }
  }
```

В этом (признаться, достаточно длинном) примере нарушение параллелизма обрабатывается путем повторного обновления строк, столбцы которых не были обновлены по причине возникновения ошибки. Самой интересной частью кода является фрагмент, в котором создаются SQL-операторы на основе "грязных" столбцов каждой строки. Следует отметить, что эта особенность создания SQL-операторов обуславливает использование метода `DataAdapter.Update()` для обновления за один раз только одной строки.

Мы рассмотрели всего лишь один пример обработки нарушений параллельного доступа к данным, наглядно демонстрирующий степень возможного участия программиста в разрешении проблем параллелизма.

Реализация пессимистического параллелизма

Пессимистический параллелизм — это просто блокировка строк. Если из базы данных была извлечена строка, она блокируется и никто другой не может получить к ней доступ. Пессимистический параллелизм идеально согласуется с моделью работы с данными типа "извлечь для редактирования/вернуть после окончания редактирования". Для того чтобы реализовать этот тип параллелизма, необходимо иметь возможность ограничить доступ к строкам базы данных, пока они находятся в состоянии "извлечены для редактирования". Так как база данных не в состоянии блокировать строки в отсоединенном режиме, нам придется придумать свое собственное решение.

Предположим, что нам необходимо создать SQL-код, который будет извлекать из базы данных для редактирования запись определенного клиента. Следует отметить, что большая часть "магии" пессимистического параллелизма заключена в SQL-операторах (в идеальном случае — в хранимых процедурах). Необходимо убедиться, что запись клиента еще не извлечена из базы данных, и, если все в порядке, отметить ее как из-

```
        updParams.Add("@old" + col.ColumnName, null);
        updParams["@old" + col.ColumnName].SourceColumn =
                          col.ColumnName;
        updParams["@old" + col.ColumnName].SourceVersion =
                          DataRowVersion.Original;

        // Обновление SQL-операторов.
        updateSQL += string.Format("{0} = @{0}, ",
                                   col.ColumnName);
        whereSQL += string.Format(" (({0} = @old{0}) OR " +
                                  "({0} IS NULL)) AND ",
                                  col.ColumnName);
      }
    }

    // Корректировка строк и их конкатенация.
    if (updateSQL.Substring(updateSQL.Length - 2) ==
                            ", " &&
        whereSQL.Substring(whereSQL.Length - 4) == "AND ")
    {
      updateSQL =
            updateSQL.Substring(0, updateSQL.Length - 2);
      whereSQL += " CustomerID = @CustomerID";

      // Добавление параметра @CustomerID.
      updParams.Add("@CustomerID",
                    SqlDbType.UniqueIdentifier,
                    0, "CustomerID");

      // Создание SQL-оператора.
      updCmd.CommandText = updateSQL + whereSQL;

      // Определение значения свойства UpdateCommand
      // объекта DataAdapter.
      dataAdapter.UpdateCommand = updCmd;

      // Создание массива, состоящего из одной строки,
      // и выполнение обновления базы данных
      // с помощью объекта DataAdapter.
      DataRow[] arrayDRs = {row};
      try
      {
        dataAdapter.Update(arrayDRs);
        Console.WriteLine("DatabaseUpdated! " +
                    "(After Row Level Collision Testing");
      }
      catch (DBConcurrencyException nestedEx)
      {
        Console.WriteLine("ConcurrencyViolation: {0}",
                          nestedEx.Message);
        Console.WriteLine(" Row: State={0}, ID={1}",
                          row["CustomerID"],
                          row.RowState);
      }
      catch (SqlException sqlex)
      {
      Console.WriteLine("SQLException: {0}",
                        sqlex.Message);
```

- **Предложить пользователю перезаписать данные. К сожалению, это не всегда возможно и не всегда приемлемо.**
- **Если позволяют данные, можно допустить коллизию на уровне столбцов. Это означает, что вам необходимо проанализировать текущую запись в базе данных и проверить на предмет изменения каждый из ее столбцов. Если столбец не был изменен, его можно обновить. Как правило, при этом возникает проблема, связанная с необходимостью убедиться в том, что одновременно были обновлены сразу несколько столбцов (например, Address, City, State и zip).**

Выбор правильного подхода к обработке нарушений параллелизма полностью зависит от предметной области. В листинге 8.11 показан пример обработки нарушений параллелизма, допускающий коллизию на уровне строк.

Листинг 8.11. Обработка нарушений параллелизма, допускающая коллизию на уровне строк

```
...

// Поиск нарушений параллелизма.
if (custTable.HasErrors)
{
  foreach (DataRow row in custTable.GetErrors())
  {
    // Мы можем устранить только ошибки обновления данных.
    if (row.RowState == DataRowState.Modified)
    {
      ////////////////////////////////////////////
      // Создание новой команды UpdateCommand.
      // Новая команда обновления данных будет
      // использоваться для устранения ошибок.

      // Сохраняем старую команду обновления
      // (если она существовала).
      SqlCommand oldUpdateCmd = null;
      if (dataAdapter.UpdateCommand != null)
      {
        oldUpdateCmd = dataAdapter.UpdateCommand;
      }

      // Создание новой команды.
      SqlCommand updCmd = conn.CreateCommand();

      // Анализ измененных строк и создание
      // нового SQL-оператора.
      string updateSQL = "UPDATE CUSTOMER SET ";
      string whereSQL = "WHERE ";
      SqlParameterCollection updParams = updCmd.Parameters;
      for (int x = 0; x < row.ItemArray.Length; ++x)
      {
        if (row[x, DataRowVersion.Original] !=
                              row[x, DataRowVersion.Current])
        {
          // Добавление параметров.
          DataColumn col = custTable.Columns[x];
          updParams.Add("@" + col.ColumnName, null);
          updParams["@" + col.ColumnName].SourceColumn =
                              col.ColumnName;
```

```
// Создание объекта CommandBuilder для генерации
// команд вставки, обновления и удаления данных.
SqlCommandBuilder bldr = new SqlCommandBuilder(dataAdapter);

try
{
  dataAdapter.Update(custTable);
}
catch (DBConcurrencyException ax)
{
  // Вывод сообщения о нарушении параллелизма.
  Console.WriteLine("ConcurrencyViolation:{0}",
                    ex.Message);
  if (custTable.HasErrors)
  {
    foreach (DataRow row in custTable.GetErrors())
    {
     Console.WriteLine(" Violation Row: (CustomerID: {0})",
                       row["CustomerID"]);
    }
  }
}
```

В приведенном примере имитируется внесение изменений в базу данных с целью нарушения параллелизма (изменения вносятся до фактического обновления объекта **DataTable),** В результате этого сразу же после попытки обновления таблицы Customers будет сгенерировано исключение DBConcurrencyException. После такой неудачной попытки обновления таблицы мы можем проанализировать ее строки и выявить те из них, которые привели к нарушению параллелизма. В данном примере мы не стараемся исправить ситуацию, а просто сообщаем о ней пользователю. Поскольку объект **DataTable** содержит информацию об ошибках, ее можно использовать для фактического решения проблемы, т.е. для того чтобы заставить объект **DataAdapter** снова обновить объект **DataTable.**

По умолчанию объект **DataAdapter** пытается обновить каждую измененную строку. Когда происходит нарушение параллелизма, он генерирует исключение и прекращает обновление базы данных. Отмены успешно внесенных изменений при этом не происходит.

Такое поведение объекта может оказаться не тем, которое требуется в конкретной ситуации. Если установить значение свойства `DataAdapter.ContinueUpdateOnError` в `true` (по умолчанию используется значение `false`), то объект **DataAdapter** попытается обновить все строки и отметит те из них, которые привели к нарушению параллелизма. В этом случае после завершения обновления таблицы вам придется вручную проверить свойство `DataTable.HasErrors` на наличие информации об ошибках. Если таковые имелись, то в свойстве `RowError` каждой "сбойной" строки будет содержаться информация об ошибке. Если же ошибок не было, то есть основание считать, что обновление базы данных прошло успешно.

Способ обработки нарушений параллелизма зависит от типа создаваемого проекта. Другими словами, принятие решения в данной ситуации целиком и полностью ложится на плечи программиста. Существует несколько основных подходов к обработке нарушений параллелизма.

- Указать пользователю на ошибку параллелизма и сообщить ему о том, что все изменения будут утеряны. На этом этапе можно обновить данные на основе содержимого базы данных и предложить пользователю повторно ввести требуемые изменения.

```
}
catch (SqlException ex)

{
  // Если обновление не удалось,
  // сообщаем об этом пользователю.
  Console.WriteLine(ex.Message);
}
```

Довольно большой объем кода, не правда ли? У вас может возникнуть вопрос: а почему нельзя просто использовать объект **CommandBuilder**? Действительно, в большинстве случаев это не только возможно, но и нужно. Тем не менее если на кон поставлена производительность приложения, то создание собственных команд обновления базы данных может существенно улучшить ситуацию. Другими словами, это просто одно из многих решений, которое принимается на этапе проектирования.

Обработка нарушений параллелизма

Неприятности иногда случаются. К примеру, вы хотите изменить строку в таблице, но оказывается, что после извлечения из базы данных она уже была изменена. В таком случае необходимо либо обработать нарушение параллелизма, либо отменить изменения объекта **DataSet**. Рассмотрим, как это реализовать на практике.

В процессе обновления базы данных объект **DataAdapter** подсчитывает количество добавленных, измененных или удаленных строк. Если результат выполнения хотя бы одной из указанных операций равен нулю (обычно этого не происходит из-за того, что не нашлось строки, соответствующей предложению WHERE, которая была изменена после своего извлечения из базы данных), объект DataAdapter генерирует исключение DBConcurrencyException. Как правило, это свидетельствует о нарушении параллелизма. В следующем примере (листинг 8.10) мы создадим нарушение параллелизма и научимся его обрабатывать.

Листинг 8.10. Обработка исключения *DBConcurrencyException*

```
// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];

// Внесение изменений в информацию о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["MiddleName"] = "Andrew";

// Имитация внесения изменений в базу данных.
SqlCommand changeCmd = conn.CreateCommand();
changeCmd.CommandText = "UPDATE Customer SET Zip = '{0}'" +
                        " WHERE CustomerID = '{1}'";

// Определение параметров (идентификационного номера клиента и
// ZIP-кода) команды changeCmd.
changeCmd.CommandText =
          string.Format(changeCmd.CommandText,
                        Convert.ToInt32(oldRow["Zip"]) + 1,
                        oldRow["CustomerID"]);

// Внесение изменений в базу данных.
conn.Open();
changeCmd.ExecuteNonQuery();
conn.Close();
```

```
updParams.Add("@Zip", SqlDbType.NVarChar,10, "Zip");
updParams["@Zip"].IsNullable = true;
updParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
updParams["@HomePhone"].IsNullable = true;
updParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
updParams["@BusinessPhone"].IsNullable = true;
updParams.Add("@DOB", SqlDbType.DateTime,0,"DOB");
updParams["@DOB"].IsNullable = true;
updParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
updParams["@Discount"].IsNullable = true;
updParams.Add("@Stamp", SqlDbType.Timestamp, 0, "Stamp");
updParams["@Stamp"].SourceVersion = DataRowVersion.Original;
updParams["@Stamp"].Direction =
                              ParameterDirection.InputOutput;

// Установка значения свойства UpdateCommand
// объекта DataAdapter.
dataAdapter.UpdateCommand = updCmd;

// Создание объекта DataSet
DataSet dataSet = new DataSet();

// Использование объекта DataAdapter
// для заполнения объекта DataSet.
dataAdapter.Fill(dataSet, "Customers");

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];

// Добавление сведений о новом клиенте.
DataRow newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Remlinger";
newRow["FirstName"] = "Mike";
newRow["Address"] = "10 Aaron Way";
newRow["City"] = "Atlanta";
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] = "(404) 543-8765";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Удаление клиента.
DataRow delRow = custTable.Rows[1];
delRow.Delete();

// Изменение информации о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["MiddleName"] = "Andrew";

try
{
  // Обновление базы данных,
  dataAdapter.Update(custTable);
  Console.Write("Successfully Updated the Database");
```

```
// Определение параметров.
delParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
delParams.Add("@Stamp", SqlDbType.Timestamp, 0, "Stamp");
delParams["@Stamp"].SourceVersion = DataRowVersion.Original;

// Установка значения свойства DeleteCommand
// объекта DataAdapter.
dataAdapter.DeleteCommand - delCmd;

// Создание команды UPDATE.
string updQry =
updQry - @"UPDATE CUSTOMER SET
             CustomerlD = @CustomerID,
             FirstName = @FirstName,
             LastName = @LastName,
             MiddleName = @MiddleName,
             Address = @Address,
             Apartment = @Apartment,
             City = @City,
             State - @State, Zip = @Zip,
             HomePhone = @HomePhone,
             BusinessPhone = @BusinessPhone,
             DOB = @DOB,
             Discount - @Discount
           WHERE Stamp = @Stamp AND CustomerlD =
                         @CustomerID
           SELECT @Stamp = Stamp FROM CUSTOMER
           WHERE CustomerlD = @CustomerID";

SqlCommand updCmd = conn.CreateCommand();
updCmd.CommandText = updQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection updparams = updCmd.Parameters;

// Определение параметров.
updParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
updParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
updParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
updParams["@MiddleName"].IsNullable = true;
updParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
updParams["@LastName"].IsNullable = true;
updParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
updParams["@Address"].IsNullable = true;
updParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
updParams["@Apartment"].IsNullable = true;
updParams.Add("@City", SqlDbType.NVarChar, 50, "City");
updParams["@City"].IsNullable = true;
updParams.Add("@State", SqlDbType.NChar, 2, "State");
updParams["@State"].IsNullable = true;
```

```
SqlCommand insCmd = conn.CreateCommand();
insCmd.CommandText = insQry;

// Создание переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection insParams = insCmd.Parameters;

// Определение параметров.
insParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
insParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
insParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
insParams["@MiddleName"].IsNullable = true;
insParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
insParams["@LastName"].IsNullable = true;
insParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
insParams["@Address"].IsNullable = true;
insParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
insParams["@Apartment"].IsNullable = true;
insParams.Add("@City", SqlDbType.NVarChar, 50, "City");
insParams["@City"].IsNullable = true;
insParams.Add("@State", SqlDbType.NChar, 2, "State");
insParams["@State"].IsNullable = true;
insParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
insParams["@Zip"].IsNullable = true;
insParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
insParams["@HomePhone"].IsNullable = true;
insParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
insParams["@BusinessPhone"].IsNullable = true;
insParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
insParams["@DOB"].IsNullable = true;
insParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
insParams["@Discount"].IsNullable = true;

// Установка значения свойства InsertCommand
// объекта DataAdapter.
dataAdapter.InsertCommand = insCmd;

// Создание команды DELETE.
string delQry

delQry = @"DELETE FROM CUSTOMER
          WHERE CustomerlD = @CustomerID AND
            Stamp = @Stamp";

SqlCommand delCmd = conn.CreateCommand();
delCmd.CommandText = delQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection delParams = delCmd. Parameters;
```

```
updParams["@Apartment"].IsNullable = true;
updParams.Add("@City", SqlDbType.NVarChar, 50, "City");
updParams["@City"].IsNullable - true;
updParams.Add("@State", SqlDbType.NChar, 2, "State");
updParams["@State"].IsNullable = true;
updParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
updParams["@Zip"].IsNullable = true;
updParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
updParams["@HomePhone"].IsNullable = true;
updParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
updParams["@BusinessPhone"].IsNullable = true;
updParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
updParams["@DOB"].IsNullable = true;
updParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
updParams["@Discount"].IsNullable = true;
updParams.Add("@Stamp", SqlDbType.Timestamp, 0, "Stamp");
updParams["@Stamp"].SourceVersion = DataRowVersion.Original;
updParams["@Stamp"].Direction =
                              ParameterDirection.InputOutput;

// Установка значения свойства UpdateCommand
// объекта DataAdapter.
dataAdapter.UpdateCommand = updCmd;
```

В свойствах параметра `@Stamp` указано не только то, что необходимо использовать первоначальную версию строки, но и что этот параметр должен быть входным-выходным (`ParameterDirection.InputOutput`). Таким образом, параметр `@Stamp` будет возвращен базой данных и его значение будет равно текущему значению столбца stamp. Необходимость иметь новое значение поля stamp обусловлена требованием иметь самую "свежую" метку времени создания/изменения строки при каждом вызове метода `DataAdapter.Update()` с целью обеспечения оптимистического параллелизма.

В листинге 8.9 приведено определение всех команд объекта **DataAdapter.**

Листинг 8.9. Определение всех команд объекта *DataAdapter*

```
// Создание и открытие соединения.
SqlConnection conn = new SqlConnection(
  "server=localhost;" +
  "database=ADONET;" +
  "Integrated Security=true;");

// Создание объекта DataAdapter.
SqlDataAdapter dataAdapter =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Создание команды INSERT.
string insQry =
insQry = @"INSERT INTO CUSTOMER(
            CustomerID, FirstName,  LastName,
            MiddleName, Address, Apartment,
            City, State, Zip, HomePhone,
            BusinessPhone, DOB, Discount)
          VALUES ( @CustomerID, @FirstName,
            @LastName, @MiddleName, @Address,
            @Apartment, @City, @State, @Zip,
            @HomePhone, @BusinessPhone, @DOB,
            @Discount )";
```

Команда UPDATE

Команда UPDATE, создаваемая объектом **CommandBuilder**, требует передачи первоначального и нового значения строки, это позволяет убедиться в том, что строка не изменилась с момента ее извлечения из базы данных. Для повышения эффективности команды л использовал метку времени создания/обновления строки. При обновлении строки необходимо получить новую метку времени создания/обновления, чтобы в следующий раз иметь самую новую версию последней. К счастью, ADO.NET позволяет осуществить это за одно обращение к базе данных — сначала производится обновление значения строки, после чего извлекается новое значение поля stamp, как показано в листинге 8.8.

Листинг 8.8. Определение значения свойства *UpdateCommand* объекта *DataAdapter*

```
...

// Создание команды UPDATE.
string updQry =
updQry = @"UPDATE CUSTOMER SET
              CustomerID = @CustomerID,
              FirstName = @FirstName,
              LastName = @LastName,
              MiddleName = @MiddleName,
              Address = @Address,
              Apartment = @Apartment,
              City = @City,
              State = @State, Zip - @Zip,
              HomePhone = @HomePhone,
              BusinessPhone = @BusinessPhone,
              DOB - @DOB,
              Discount = @Discount
            WHERE Stamp = @Stamp AND CustomerlD =
                          @CustomerID
            SELECT @Stamp = Stamp FROM CUSTOMER
            WHERE CustomerlD = @CustomerID";

SqlCommand updCmd = conn.CreateCommand();
updCmd.CommandText = updQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection updParams = updCmd.Parameters;

// Определение параметров.
updParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
updParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
updParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
updParams["@MiddleName"].IsNullable = true;
updParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
updParams["@LastName"].IsNullable = true;
updParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
updParams["@Address"].IsNullable = true;
updParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
```

Команда INSERT, созданная объектом CommandBuilder, практически полностью удовлетворяет наши потребности. Код, содержащийся в листинге 8.6, дублирует эту команду.

Один из интересных моментов приведенного выше кода заключается в использовании методом SqlParameterCollection.Add() имени поля, из которого необходимо извлечь значение. Четвертый параметр этого метода, фактически определяет отображение параметров на столбцы таблицы базы данных. В течение выполнения метода DataAdapter. Update {} созданная нами команда INSERT вызывается по одному разу для каждой измененной строки.

Команда DELETE

Объект CommandBuilder создает сложную команду DELETE, которая прежде чем удалить строку, проверяет, не изменилась ли она с момента ее последнего изменения (извлечения из базы данных). В рассматриваемом примере я использую для этой цели поле SQL Server Timestamp (большинство баз данных имеют что-то подобное этому полю). Поле Timestamp предназначено для обновления метки времени создания/обновления строки при каждом ее (строки) изменении. Наличие этого поля позволяет нам гарантировать отсутствие изменений строки с момента ее извлечения из базы данных. Поскольку мы используем поле Timestamp, а не проводим проверку значений столбцов строки, команда DELETE будет принимать только два параметра, как показано в листинге 8.7.

Листинг 8.7. Определение значения свойства *DeleteCommand* объекта *DataAdapter*

```
...

// Создание команды DELETE.
string delQry

delQry = @"DELETE FROM CUSTOMER
          WHERE CustomerID = @CustomerID AND
            Stamp - @Stamp";

SqlCommand delCmd = conn.CreateCommand();
delCmd.CommandText = delQry;

// Определение переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection delParams = delCmd.Parameters;

// Определение параметров.
delParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
delParams.Add("@Stamp", SqlDbType.Timestamp, 0, "Stamp");
delParams["@Stamp"].SourceVersion = DataRowVersion.Original;

// Установка значения свойства DeleteCommand
// объекта DataAdapter.
dataAdapter.DeleteCommand = delCmd;

...
```

Обратите внимание на то, что свойство SourceVersion параметра stamp устанавливается равным DataRowVersion.Original (первоначальная версия строки). Это означает, что даже если кто-то изменит значение поля Stamp, сравнение будет все равно происходить с использованием первоначального значения данного поля.

Листинг 8.6. Определение значения свойства *InsertCommand* объекта *DataAdapter*

```
...

// Создание команды INSERT.
string insQry =
insQry = @"INSERT INTO CUSTOMER(
             CustomerID, FirstName,  LastName,
             MiddleName, Address, Apartment,
             City, State, Zip, HomePhone,
             BusinessPhone, DOB, Discount)
           VALUES ( @CustomerID, ©FirstName,
             ©LastName, @MiddleName, @Address,
             @Apartment, @City, @State, @Zip,
             ©HomePhone, @BusinessPhone, @DOB,
             @Discount )";

SqlCommand insCmd = conn.CreateCommand() ;
insCmd.CommandText = insQry;

// Создание переменной для упрощения
// доступа к коллекции параметров.
SqlParameterCollection insParams = insCmd.Parameters;

// Определение параметров.
insParams.Add("@CustomerID", SqlDbType.UniqueIdentifier, 0,
              "CustomerID");
insParams.Add("@FirstName", SqlDbType.NVarChar, 50,
              "FirstName");
insParams.Add("@MiddleName", SqlDbType.NVarChar, 50,
              "MiddleName");
insParams["@MiddleName"].IsNullable = true;
insParams.Add("@LastName", SqlDbType.NVarChar, 50,
              "LastName");
insParams["@LastName"].IsNullable = true;
insParams.Add("@Address", SqlDbType.NVarChar, 50,
              "Address");
insParams["@Address"].IsNullable = true;
insParams.Add("@Apartment", SqlDbType.NVarChar, 50,
              "Apartment");
insParams["@Apartment"].IsNullable = true;
insParams.Add("@City", SqlDbType.NVarChar, 50, "City");
insParams["@City"].IsNullable = true;
insParams.Add("@State", SqlDbType.NChar, 2, "State");
insParams["@State"].IsNullable = true;
insParams.Add("@Zip", SqlDbType.NVarChar, 10, "Zip");
insParams["@Zip"].IsNullable = true;
insParams.Add("@HomePhone", SqlDbType.NVarChar, 14,
              "HomePhone");
insParams["@HomePhone"].IsNullable = true;
insParams.Add("@BusinessPhone", SqlDbType.NVarChar, 14,
              "BusinessPhone");
insParams["@BusinessPhone"].IsNullable = true;
insParams.Add("@DOB", SqlDbType.DateTime, 0, "DOB");
insParams["@DOB"].IsNullable = true;
insParams.Add("@Discount", SqlDbType.Float, 8, "Discount");
insParams["@Discount"].IsNullable = true;

// Установка значения свойства InsertCommand объекта DataAdapter.
dataAdapter.InsertCommand = insCmd;
```

В большинстве случаев при работе с серверами баз данных (такими, как SQL Server, Oracle, DB2 и т.д.) увеличения быстродействия можно достичь с помощью помещения всех команд INSERT, DELETE и UPDATE в хранимые процедуры. Для облегчения восприятия материала предложенные мной примеры будут написаны на "родном" коде SQL. При желании вы можете использовать их в качестве основы при создании хранимых процедур для конкретного сервера баз данных.

Создание параметризированных команд для объекта DataAdapter

Прежде чем создать собственные команды, необходимо понять, как команды объекта **DataAdapter** используют параметры при определении обновляемых данных. Напомним, что объект **DataAdapter** содержит команды (объекты **Command)** для выполнения операций вставки, обновления и удаления информации из базы данных. Каждая из этих команд имеет набор параметров, представляющих столбцы базы данных. Объект **DataAdapter** использует параметры в качестве своеобразной точки "стыковки" между объектом DataRow и базой данных. Ниже перечислены самые важные свойства классов параметров (OleDbParameter, SqlParameter, OracleParameter и OdbcParameter).

- SourceColumn — имя столбца в таблице DataTable, которому соответствует этот параметр.
- SourceVersion — версия (DataRowVersion) строки **DataRow,** которой соответствует этот параметр. Более подробно версии строк рассматривались в главе 7, "Манипулирование объектом DataSet". С помощью этого свойства можно отобразить первоначальную версию столбца на один параметр, а модифицированную версию — на другой.
- Direction — это свойство уже рассматривалось нами в главе 3, "Выполнение команд". Здесь оно имеет особое значение, так как после выполнения команды значение выходного (Output) или входного-выходного (InputOutput) параметра будет помещено в столбец SourceColumn строки **DataRow.**

Для создания собственных команд обновления необходимо провести их "тонкую" настройку. Первые два свойства параметра (SourceColumn и SourceVersion) привязывают его к определенному столбцу. Свойство SourceVersion предназначено для обозначения версии строки, с которой необходимо связать параметр. В общем случае рекомендуется связывать с версией DataRowVersion. Current параметры, предназначенные для фактического обновления информации в базе данных, а с версией DataRowVersion.Original — параметры, предназначенные для использования в предложении WHERE с целью идентификации строки.

По умолчанию свойство Direction имеет значение Input, однако вам, вероятно, потребуется изменить его, создавая параметр, использующийся для возвращения значения из базы данных (например, идентификационного номера для вставляемой в таблицу строки). При желании вы можете настроить оператор INSERT так, чтобы столбец первичного ключа был выходным параметром, позволяя тем самым SQL-оператору или хранимой процедуре установить его (параметра) значение после вставки строки в базу данных. Более подробно возвращение из базы данных идентификационного номера строки рассматривается в разделе "Получение идентификатора новой строки от базы данных SQL Server", далее в этой главе.

Команда INSERT

Команда INSERT, созданная объектом CommandBuilder, является достаточно эффективной. Но в связи с тем, что мы хотим полностью избавиться от объекта CommandBuilder, нам придется собственноручно создать все команды обновления базы данных, как показано в листинге 8.6.

```
((Discount IS NULL AND @p26 IS NULL) OR
 (Discount = @p27)) AND
((CheckedOut IS NULL AND @p28 IS NULL) OR
 (CheckedOut = @p29)) )
```

Листинг 8.4. Запрос ***DELETE***

```
DELETE FROM CUSTOMER WHERE ( (CustomerlD = @p3) AND
     ((FirstName IS NULL AND @p4 IS NULL) OR
      (FirstName - @p5)) AND
     ((LastName IS NULL AND Gp6 IS NULL) OR
      (LastName = @p7)) AND
     ((MiddleName IS NULL AND @p8 IS NULL) OR
      (MiddleName = @p9)) AND
     ((Address IS NULL AND @p10 IS NULL) OR
      (Address = @p11)) AND
     ((Apartment IS NULL AND @p12 IS NULL) OR
      (Apartment = @p13)) AND
     ((City IS NULL AND @p14 IS NULL) OR (City = @p15)) AND
     ((State IS NULL AND @p16 IS NULL) OR (State = @p17)) AND
     ((Zip IS NULL AND @p18 IS NULL) OR (Zip = @p19)) AND
     ((HomePhone IS NULL AND @p20 IS NULL) OR
      (HomePhone = @p21)) AND
     ((BusinessPhone IS NULL AND @p22 IS NULL) OR
      (BusinessPhone = @p23)) AND
     ((DOB IS NULL AND @p24 IS NULL) OR (DOB - @p25)) AND
     ((Discount IS NULL AND @p26 IS NULL) OR
      (Discount = @p27)) AND
     ((CheckedOut IS NULL AND @p28 IS NULL) OR
      (CheckedOut = @p29)) )
```

Листинг 8.5. Запрос ***INSERT***

```
INSERT INTO CUSTOMER! CustomerlD , FirstName , LastName ,
                      MiddleName , Address , Apartment ,
                      City , State , Zip , HomePhone ,
                      BusinessPhone , DOB , Discount )
VALUES ( @pl , @p2 , @p3 , @p4 , @p5 , @p6 , @p7 , @p8 ,
         @p9 , @p10 , @pll , @p12 , @p13 )
```

Операторы SQL, созданные объектом **CommandBuilder**, очень жизнеспособны, но не очень эффективны, так как сгенерированный SQL-код имеет слишком большой размер, приводящий к увеличению времени компиляции запроса на сервере и объема передаваемой базе данных информации. К тому же объект **CommandBuilder** создает SQL-операторы во время выполнения программы, что негативно влияет на скорость обновлений (SQL-операторы создаются заново для каждого обновления). К счастью, в большинстве случаев вы будете располагать дополнительными сведениями о базе данных, что позволит вам найти более эффективное решение. К примеру, я предпочитаю использовать хранимые процедуры для выполнения каждой операции обновления, и благодаря созданию плана запроса, а также предварительной компиляции кода на сервере существенно увеличивается быстродействие приложения.

Повышение эффективности команд, создаваемых объектом CommandBuilder

Объект **CommandBuilder** наиболее удобно использовать при работе с аморфными данными или когда большее значение имеет скорость разработки, а не производительность. Если же ситуация отличается от описанной, вам следует создать свои собственные объекты команд DELETE, INSERT и UPDATE для объекта **DataAdapter.**

- Если после заполнения объекта DataTable с помощью метода DataAdapter. SelectCommand в схему были внесены изменения, перед вызовом метода DataAdapter.Update() необходимо вызвать метод CommandBuider. RefreshSchema(). В противном случае, объект **CommandBuilder** не сможет связать столбец таблицы **DataTable** с информацией, полученной из свойства SelectCommand.
- **Объект** CommandBuilder создает команды по мере необходимости. Объект команды существует немногим дольше того времени, которое требуется на обновление данных. Это означает также, что жизненный цикл объекта **CommandBuilder** должен перекрывать жизненный цикл команды, выполняемой в процессе вызова метода DataAdapter.Update().
- Для того чтобы объект **CommandBuilder** проводил обновление с помощью транзакции, удостоверьтесь, что транзакция соединения связана со свойством SelectCommand. Это приведет к наследованию всех связанных с данным соединением транзакций.
- Объект **CommandBuilder** создает команды на основе метаданных столбцов таблицы. Таким образом, обновление базы данных будет осуществляться без учета отношений, ограничений или наличия других таблиц в объекте **DataSet.**

Если все эти условия будут удовлетворены, вы можете рассчитывать на безукоризненное создание объектом **CommandBuilder** команд INSERT, UPDATE и DELETE.

Объект **CommandBuilder** создает SQL-код, который обеспечивает оптимистический параллелизм путем проверки того, что состояние каждого столбца в обновляемых и удаляемых строках идентично его первоначальному состоянию. Поддержка оптимистического параллелизма в отсоединенном режиме доступа к данным достаточно проста — мы всего лишь убеждаемся в том, что строка не претерпела изменений с момента ее извлечения из базы данных. Сгенерированный SQL-код практически идентичен для каждого управляемого поставщика данных за исключением некоторых мелких деталей (например, в управляемом поставщике OLE DB вместо именованных параметров используются знаки вопроса). В листингах 8.3, 8.4 и 8.5 приведены примеры параметрических запросов, созданных с помощью объекта **CommandBuilder.**

Листинг **8.3. Запрос *UPDATE***

```
UPDATE CUSTOMER SET Address = @p1 , Zip = @p2
WHERE ( (CustomerID = @p3) AND
     ((FirstName IS NULL AND @p4 IS NULL) OR
      (FirstName - @p5)) AND
     ((LastName IS NULL AND @p6 IS NULL) OR
      (LastName = @p7)) AND
     ((MiddleName IS NULL AND @p8 IS NULL) OR
      (MiddleName - @p9) } AND
     ((Address IS NULL AND @p10 IS NULL) OR
      (Address = @p11)) AND
     ((Apartment IS NULL AND @p12 IS NULL) OR
      (Apartment - @p13)) AND
     ((City IS NULL AND @p14 IS NULL) OR (City - @p15)) AND
     ((State IS NULL AND @p16 IS NULL) OR (State = @p17)) AND
     ((Zip IS NULL AND @p18 IS NULL) OR (Zip = @p19)) AND
     ((HomePhone IS NULL AND @p20 IS NULL) OR
      (HomePhone = @p21)) AND
     ((BusinessPhone IS NULL AND @p22 IS NULL) OR
      (BusinessPhone = @p23)) AND
     ((DOB IS NULL AND @p24 IS NULL) OR (DOB = @p25)) AND
```

Идентификация строк при оптимистическом параллелизме

Реализация оптимистического параллелизма в отсоединенном режиме доступа к данным зависит от возможности идентифицировать строки, которые изменились с момента их извлечения из базы данных. Существует несколько способов такой идентификации (табл. 8.1).

Таблица 8.1. Методы идентификации строк, использующиеся при реализации оптимистического параллелизма

Метод	Описание
Создание метки времени создания/изменения	Этот метод предполагает создание столбца, который будет использоваться исключительно для проверки параллелизма. Обычно метка времени создания/изменения (метка даты) обновляется при вставке и каждом обновлении данных. Реализация этого метода возможна только при наличии контроля над схемой используемой базы данных
Сравнение строк	Этот подход используется классом CommandBuilder и предполагает проверку равенства значений строки значениям, полученным после извлечения строки из базы данных
Определение контрольной суммы строки	Этот метод представляет собой более сложную версию метода, предполагающего создание метки времени создания/изменения. Контрольная сумма определяется на основе всех полей строки, а сравнение производится между контрольной суммой строки, извлеченной из базы данных, и таковой, хранящейся в базе данных на текущий момент. Код, вычисляющий контрольную сумму строки, достаточно сложен. Метод с вычислением контрольной суммы более эффективен, чем метод, предполагающий сравнение строк, так как сравнение контрольных сумм возможно без фактического сравнения строк. Тем не менее само по себе вычисление контрольной суммы также требует определенных ресурсов

Класс CommandBuilder

Класс CommandBuilder отвечает за генерацию запросов по мере возникновения необходимости в них в объекте DataAdapter. Создав объект **CommandBuilder,** его необходимо передать в конструктор объекта DataAdapter. Как только объект CommandBuilder узнает об объекте **DataAdapter,** он использует свойство DataAdapter. SelectCommand, чтобы получить информацию о столбцах объекта **DataTable** и иметь возможность создать команды вставки, обновления и удаления данных. Если объект **DataAdapter** уже содержит некоторые команды (такие, как UPDATE и DELETE), объект **CommandBuilder** создает только отсутствующие команды. Для создания необходимого SQL-оператора объект CommandBuilder регистрируется для получения уведомления о событии DataAdapter.RowUpdating(). Каждый **управляемый поставщик** содержит собственную реализацию класса CommandBuilder (SqlCommandBuider, OleDbCommandBuilder, OdbcCommandBuider и OracleCommandBuider). Для **того** чтобы гарантировать нормальное функционирование объекта **CommandBuilder,** необходимо учесть несколько моментов.

- Свойство DataAdapter.SelectCommand должно содержать действительную команду, которая использовалась для заполнения обновляемого объекта **DataTable.** Объект **CommandBuilder** применяет SQL-оператор SELECT, для того чтобы иметь возможность создавать операторы вставки, обновления и удаления данных.
- Таблица **DataTable,** которая будет обновляться, должна либо содержать столбец уникальных значений, либо для нее должен быть определен первичный ключ. В противном случае объект **CommandBuilder** не сможет идентифицировать запись в таблице. В частности, при отсутствии уникального столбца не **сможет** быть создано предложение WHERE операторов UPDATE и DELETE.

```
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] - "(404) 543-8765";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Добавление нового клиента.
newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Jones";
newRow["FirstName"] = "Chipper";
newRow["Address"] = "10 Aaron Way";
newRow["City"] = "Atlanta";
newRow["State"] = "GA";
newRow["Zip"] = "30307";
newRow["HomePhone"] = "(404) 543-8768";
newRow.EndEdit();
custTable.Rows.Add(newRow);

// Изменение информации о клиенте.
DataRow oldRow = custTable.Rows[0];
oldRow["Address"] = "53 Peachtree Center";
oldRow["Zip"] = "30342";

// Удаление клиента.
DataRow delRow = custTable.Rows[1];
delRow.Delete();

// Создание объекта CommandBuilder для генерирования
// команд вставки, обновления и удаления данных.
SqlCommandBuilder bldr = new SqlCommandBuilder(dataAdapter) ;

// Обновление базы данных.
dataAdapter.Update(custTable);
```

Большая часть этого кода должна быть вам уже знакома. Мы создаем две новых строки в таблице Customer, изменяем информацию о существующем клиенте, после чего удаляем его из таблицы. При вызове метода DataAdapter.Update() он анализирует таблицу custTable, вставляет новые записи, удаляет запись, предназначенную для удаления, а также проводит обновление заданной строки. Команды INSERT, UPDATE и DELETE **при** этом не определяются, так как объект **CommandBuilder** создает их сам по мере необходимости.

Метод Update DataAdapter-классов **имеет** множество переопределенных вариантов. В этом примере в качестве параметра методу Update передавался объект DataTable. Несмотря на то что это самый распространенный способ его использования, метод Update может принимать в качестве параметра массив объектов **DataRow** или же целый объект **DataSet.** Недостатком использования переопределенного варианта метода Update, принимающего в качестве параметра объект **DataSet,** является то, что он предполагает наличие в объекте **DataSet** таблицы с **именем** "Table". По идее можно было бы предположить, что вызов метода DataAdapter.Update() с объектом **DataSet** в качестве параметра приведет к обновлению всего объекта **DataSet,** однако на самом деле это приведет только к обновлению таблицы "Table". Альтернативный способ использования метода update заключается в передаче ему в качестве параметра объекта **DataSet** и имени таблицы, которую необходимо обновить.

```
// Создание объектов DataAdapter.
SqlDataAdapter customerAdapter =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);
SqlDataAdapter invoiceAdapter =
        new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание объекта DataSet,
DataSet dataSet - new DataSet();

// Использование объектов DataAdapter
// для заполнения объекта DataSet.
customerAdapter.Fill(dataSet, "Customers");
invoiceAdapter.Fill(dataSet, "Invoices");

// Внесение изменений в объект DataSet.
...

// Обновление объекта DataSet
// (каждая таблица должна быть обновлена отдельно).
invoiceAdapter.Update(dataSet, "Invoices");
customerAdapter.Update(dataSet, "Customers");
```

Реализация оптимистического параллелизма

Конечной целью реализации оптимистического параллелизма является предотвращение перезаписи последних обновлений базы данных. Предположим, что у нас есть система управления клиентами и мы извлекли из нее информацию об определенном клиенте. В это же время менеджер по работе с клиентами извлек из базы данных информацию о том же клиенте и изменил его демографические сведения. Если теперь вы попытаетесь внести некоторые изменения в информацию о клиенте, должны ли эти изменения перезаписать информацию, внесенную менеджером по работе с клиентами? Как разрешить подобную проблему? Оптимистический параллелизм позволяет изменять данные только в том случае, если они не были изменены с момента их загрузки из базы данных, В рассматриваемой ситуации вы получите сообщение о невозможности изменения данных.

В случае использования отсоединенных данных достичь оптимистического параллелизма можно путем гарантирования того, что все записи в объекте DataSet идентичны записям, извлеченным из базы данных, К счастью, ADO.NET позволяет осуществлять такую проверку автоматически с помощью класса CommandBuilder. Пример оптимистического параллелизма показан в листинге 8.2.

Листинг 8.2. Оптимистический параллелизм, обеспечиваемый классом *CommandBuilder*

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable custTable = dataSet.Tables["Customers"];

// Добавление нового клиента.
DataRow newRow = custTable.NewRow();
newRow.BeginEdit();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Remlinger";
newRow["FirstName"] = "Mike";
newRow["Address"] = "10 Aaron Way";
newRow["City"] = "Atlanta";
```

Параллелизм в ADO.NET

Прежде всего необходимо определить тип параллелизма, который требуется в нашем конкретном случае. ADO.NET поддерживает три типа параллелизма.

- **Оптимистический параллелизм (optimistic concurrency).** Все пользователи могут получить доступ к данным объекта DataSet до тех пор, пока кто-либо из них не начнет осуществлять запись информации в базу данных. Это самая распространенная модель параллелизма в ADO.NET.
- **Пессимистический параллелизм (pessimistic concurrency).** Нельзя получить доступ к данным объекта **DataSet,** пока кто-либо из пользователей владеет копией данных (в этом случае говорят, что данные блокируются).
- **Деструктивный параллелизм (destructive concurrency). Все** пользователи имеют доступ к объекту **DataSet,** но в базе данных будут зафиксированы только самые последние изменения. На самом деле это означает фактическое отсутствие контроля параллелизма. Компания Microsoft называет такую модель параллелизма моделью "последний побеждает".

Для того чтобы определить нужный тип параллелизма, необходимо проанализировать характер использования информации, хранящейся в базе данных. Сделанный в результате этого выбор будет представлять собой компромисс между ограничением доступа к данным и поддержкой их целостности.

ADO.NET поддерживает оптимистический параллелизм с помощью класса `CommandBuilder`. В этой главе будет рассмотрено использование класса `CommandBuilder` для реализации оптимистического параллелизма, а также создание собственных механизмов для обеспечения оптимистического, пессимистического и деструктивного параллелизма.

Обновление объекта DataSet с помощью объекта DataAdapter

Прежде чем рассмотреть аспекты реализации различных моделей параллелизма в ADO.NET, необходимо понять поведение объекта **DataAdapter** при вызове метода Update. Поскольку с помощью объекта **DataAdapter** можно обновить только один объект **DataTable,** необходимо поддерживать взаимооднозначное соответствие объектов **DataAdapter** объектам **DataTable.** При вызове метода `DataAdapter.Update()` объект **DataAdapter** анализирует таблицу и составляет список изменившихся строк. Для каждой изменившейся строки объект **DataAdapter** выполняет соответствующую команду. К примеру, для новой строки выполняется команда `InsertCommand`, для изменившейся строки — `UpdateCommand`, а для удаленной — `DeleteCommand`. При выполнении каждой команды объект **DataAdapter** делает одно обращение к базе данных; другими словами, он не поддерживает пакетные обновления.

Для того чтобы оптимизировать работу объекта **DataAdapter,** необходимо поддерживать его существование на протяжении всей жизни объекта DataSet. После заполнения объекта **DataSet** объект **DataAdapter** может быть использован повторно для обновления базы данных, как показано в листинге **8.1.**

Листинг 8.1. Использование нескольких объектов *DataAdapter*

```
// Создание и открытие соединения.
SqlConnection conn = new SqlConnection(
                                "server=localhost;" +
                                "database=ADONET;" +
                                "Integrated Security=true;");
```

Глава 8

Обновление базы данных

***В этой** главе...*

Проблемы, связанные с использованием отсоединенных данных

Параллелизм в ADO.NET

Несколько наиболее распространенных вопросов, связанных с обновлением данных

До сих пор отсоединенные данные упоминались, в основном, в положительном смысле. Действительно, мы можем поддерживать локальный кэш данных, виосить в них изменения и осуществлять к ним доступ без создания соединения с базой данных. К сожалению, за все нужно платить.

Проблемы, связанные с использованием отсоединенных данных

Самым сложным в использовании отсоединенных данных является синхронизация копии объекта DataSet и базы данных. Несмотря на то что в ADO.NET сохранение информации в базе данных осуществляется достаточно просто, настоящие проблемы начинаются тогда, когда нам нужно обеспечить поддержку параллелизма.

Ранее уровни доступа к данным возлагали поддержку параллелизма целиком и полностью на саму базу данных. Разработчику достаточно было указать тип параллелизма, который необходимо поддерживать, после чего база данных брала на себя основную работу — блокировку строк по мере обращения к ним пользователей.

Свойство `IsolationLevel` интерфейса `IDbTransaction` (поддерживается классами SqlTransaction, `OleDbTransaction`, `OdbcTransaction` и `OracleTransaction`) определяет уровень изоляции транзакции. Несмотря на то что транзакции можно использовать для облегчения обработки параллельных обращений, они жестко связаны с соединениями. Другими словами, их нельзя применять при работе с отсоединенными данными ADO.NET. К примеру, мы не можем начать транзакцию (открыть соединение) и оставлять ее открытой на протяжении всего времени использования объекта DataSet.

Предположим, что мы разрабатываем типичное Web-приложение для электронной коммерции. Чтобы пользователь подтвердил свой почтовый адрес и адрес доставки товара, его данные должны быть извлечены из базы данных. В большинстве случаев пользователи не меняют подобную информацию. Тем не менее если для блокировки записи на время ожидания подтверждения от пользователя будет применена транзакция, эта запись окажется заблокированной и для остальных пользователей (например, для отдела доставки). Так как мы имеем дело с Web-приложением, пользователь просто закрывает обозреватель, заставляя приложение удерживать блокировку записи на протяжении определенного периода времени, что, конечно же, является неприемлемым. Решение подобной ситуации напрашивается само собой — параллелизм должен обрабатываться за пределами базы данных, что, однако, не означает полный отказ от поддержки параллелизма со стороны базы данных.

`DataSet.Merge(DataSet)`, потому что параметры false и `MissingSchemaAction.Add` определяют поведение, принятое по умолчанию. С другой стороны, во втором вызове мы ожидаем идентичность схем объектов DataSet, а поэтому при нарушении этого условия необходимо сгенерировать исключение.

Резюме

Механизмы манипуляции и перемещения по данным, поддерживаемые объектом DataSet, предоставляют функциональность, которая делает его чем-то большим, чем просто база данных, хранящаяся в памяти. Упрощенный способ изменения строк, а также способность объекта **DataRow** хранить "историю" данных, делает механизм создания, чтения, обновления информации в объекте DataSet и удаления информации из него намного более мощным. Кроме этого, перемещение по отношениям между объектами **DataTable** обеспечивает функциональность, прежде не свойственную уровням доступа к данным Microsoft. Наконец, даже несмотря на отсутствие поддержки языка SQL, ADO.NET позволяет фильтровать, сортировать данные и проводить их поиск в объектах DataTable с помощью метода `DataTable.Select()`, который при совместном применении его с объектом **DataView** в состоянии удовлетворить запросы самых искушенных разработчиков.

Метод DataSet.Merge() поддерживает все три сценария слияния объектов **DataSet** с помощью переопределенных вариантов, принимающих различные параметры. При вызове метода Merge помните, что передаваемые ему данные будут объединены с данными используемого объекта **DataSet. В** процессе слияния объектов **DataSet** первой объединяются **схемы,** затем — данные.

Ниже приведены примеры трех вариантов слияния объектов **DataSet.**

```
// Слияние двух объектов DataSet.
targetDataSet.Merge(sourceDataSet);

// Слияние двух объектов DataSet.
targetDataSet.Merge(sourceDataSet.Tables["Products"]);
```

В приведенных выше примерах с целевым **объектом DataSet** объединяются исходный объект **DataSet и** таблица Products исходного объекта **DataSet.** Если в схеме целевого объекта **DataSet** будет отсутствовать заданный **объект DataTable,** по умолчанию **к схеме** добавляется необходимый фрагмент.

```
// Слияние изменений, происшедших в одном объекте DataSet,
// с изменениями, происшедшими в другом
// объекте DataSet (оба имеют одинаковую схему).
targetDataSet.Merge(sourceDataSet.GetChanges());

// Слияние изменений, происшедших в одном объекте DataSet,
// с изменениями, происшедшими в другом
// объекте DataSet (оба имеют одинаковую схему).
targetDataSet.Merge(
            sourceDataSet.Tables["Products"].GetChanges());
```

В приведенных выше примерах показано слияние одних лишь изменений исходного объекта **DataSet** с целевым объектом **DataSet.**

```
// Слияние двух объектов DataSet с установкой
// состояния строк в Unchanged.
targetDataSet.Merge(sourceDataSet, false);

// Слияние двух объектов DataSet с сохранением
// состояния и версий строк.
targetDataSet.Merge(sourceDataSet, true);
```

В приведенном выше примере переопределенный вариант метода Merge позволяет указать, следует ли сохранять при слиянии объектов **DataSet** изменения в строках исходного объекта **DataSet** (состояние и версию строк). Если в качестве второго параметра указать true, то при слиянии объектов **DataSet** сохраняется состояние и различные версии строк. Вызов метода с false в качестве второго параметра равносильно его отсутствию.

```
// Слияние двух объектов DataSet с различными схемами.
targetDataSet.Merge(sourceDataSet,
                    false,
                    MissingSchemaAction.Add);
// Слияние двух объектов DataSet с похожими схемами
//  (в случае различия схем будет сгенерировано исключение).
targetDataSet.Merge(sourceDataSet,
                    false,
                    MissingSchemaAction.Error);
```

Наконец, в приведенном выше примере переопределенный вариант метола Merge поддерживает определение действия на случай отсутствия нужной **схемы в** целевом объекте **DataSet.** Первый вызов метода Merge идентичен вызову метода

порядковый **номер** строки в объекте DataView (а не в объекте **DataTable),** а метод DataView.FindRows() — массив объектов **DataRowView,** которые представляют строки, соответствующие критерию (листинг 7.18).

Листинг 7.18, Поиск **с помощью объекта** *DataView*

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Определение порядка сортировки.
dataTable.DefaultView.Sort = "Vendor";

// Поиск товара по производителю (Wilson).
int found = dataTable.DefaultView.Find("Wilson");
Console.WriteLine("Vendor: {0}",
                  dataTable.DefaultView[found]["Vendor"]);

// Поиск товара по производителю (Rawlings).
DataRowView[] rows *
                  dataTable.DefaultView.FindRows("Rawlings");
Console.WriteLine("Rawlings count: {0} ", rows.Length);

// Определение порядка сортировки.
dataTable.DefaultView.Sort = "Vendor, Price DESC";

// Поиск товара по производителю и цене.
object[] criteria - new object[] {"Wilson", 29.99};
found = dataTable.DefaultView.Find(criteria);
Console.WriteLine("Compound Vendor Search: {0}",
                  dataTable.DefaultView[found]["Vendor"]);
```

Последний тип критерия представляет собой пример составного ключа. При поиске по двум ключам необходимо использовать массив ключей. Составной ключ может использоваться как с методом DataView.Find(), так и с методом DataView.FindRows().

Слияние объектов DataSet

Иногда может возникнуть необходимость в слиянии двух объектов DataSet. К примеру, ваше приложение должно провести анализ данных, содержащихся в нескольких объектах DataSet. Для того чтобы создать отношения между таблицами различных объектов DataSet, последние необходимо объединить. С другой стороны, при проведении удаленного обновления объектов DataSet через Web-службу вам может понадобиться объединить изменения этих объектов в локальном объекте DataSet. Для поддержки такой функциональности ADO.NET предоставляет три способа:

- копирование содержимого объекта DataSet с той же схемой с целью объединения всех строк объектов DataSet;
- обновление состояния записей из одного объекта **DataSet** в другой объект **DataSet** с использованием той же схемы;
- слияние двух объектов **DataSet** с различными **схемами с целью** получения объекта **DataSet,** *содержащего* все объекты **DataTable из обоих** объектов **DataSet.**

```
    Console.WriteLine("  Item: {0} - Vendor: "+
                      "{1} - Price: {2}",
                      row["ProductID"],
                      row["Vendor"],
                      row["Price"]);
}
```

При работе с объектом **DataSet,** который содержит связанные объекты DataTable, возвращаемый массив также содержит связанные объекты DataRow. Это может оказаться полезным, так как результат проведения поиска в объекте **DataTable,** содержащемся в реляционном объекте **DataSet,** может быть использован для получения подмножества объекта **DataSet,** соответствующего заданному критерию (листинг 7.17). В этом примере демонстрируется слияние объектов **DataSet,** которое будет рассмотрено далее в этой главе.

Листинг 7.17. Выделение подмножества объекта *DataSet* с помощью методов *Select* и *Merge*

```
...

// Поиск клиентов, проживающих в штате Джорджия.
DataRow[] gaCusts =
        dataSet.Tables["Customers"].Select("State = 'GA'");

// Создание нового объекта DataSet.
DataSet newDataSet = new DataSet();

// Копирование схемы старого объекта DataSet в поток.
MemoryStream strm = new MemoryStream();
dataSet.WriteXmlSchema(strm);

// Помещение указателя на начало потока.
strm.Position = 0;

// Считывание схемы в новый объект DataSet.
newDataSet.ReadXmlSchema(strm);

// Копирование строк в новый объект DataSet.
newDataSet.Merge(gaCusts);
```

Поиск с помощью объекта DataView

Объект **DataView** может использоваться для проведения поиска в объекте **DataTable** с помощью свойства RowFilter. Следует отметить, что довольно часто установка значения свойства RowFilter полностью функционально эквивалента поиску. Напомним: свойство RowFilter используется для отбора строк объекта DataTable, удовлетворяющих определенному критерию. Несмотря на то что эта методика очень похожа на использование метода DataTable.Select(), она может оказаться более удобной в том случае, если вы уже используете объекты **DataView.** Во-первых, это позволяет избежать создания новых копий данных, а во-вторых, получаемый в результате прямоугольный набор данных более удобен в использовании, чем массив объектов DataRow.

В отличие от этого, методы Find и FindRows класса DataView предназначены для поиска строк по заданному значению ключа. Для того чтобы провести поиск в объекте **DataView,** необходимо установить значение свойства DataView.Sort равным ключу (ключам), по которому будет производиться поиск, после чего вызвать метод DataView.Find() или DataView.FindRows(). Метод DataView.Find() возвращает

ки, **которые были изменены с момента последнего вызова метода** DataTable.AcceptChanges().

- None — **не отображается ни одна строка.**
- OriginalRows — отображаются первоначальные версии всех строк (неизмененных, измененных, новых и удаленных).
- Unchanged — отображает только те строки, которые остались неизмененными с момента последнего вызова метода DataTable, AcceptChanges O .

Поиск в объекте DataSet

К сожалению, объект **DataSet** не поддерживает язык запросов SQL во всей его полноте. В то же время ADO.NET поддерживает проведение простого поиска в пределах объекта **DataTable,** в результате которого возвращаются строки, соответствующие заданному критерию. Кроме этого, объект **DataView** поддерживает поиск на основе индекса.

Поиск с помощью метода DataTable.Select

Для поиска строк в объекте **DataTable,** удовлетворяющих определенному критерию, используется метод DataTable.Select O . Критерий поиска задается выражением, синтаксис которого совпадает с синтаксисом свойства DataTable.Expression. В результате вызова метода DataTable.Select() возвращается массив строк **DataRow,** соответствующих заданному критерию. Если критерию не соответствует ни одна строка, массив имеет нулевой размер, Пример использования метода Select показан в листинге 7.16,

Листинг 7.16. Поиск в объекте *DataTable* с помощью метода *Select()*

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable dataTable - dataSet.Tables[0];

// Поиск строк по заданному критерию.
DataRow[] rows = dataTable.Select("Vendor - 'Wilson'");
Console.WriteLine("Simple Select Search:");

// Вывод строк, соответствующих
// заданному критерию, на консоль.
foreach (DataRow row in rows)
{
  Console.WriteLine("  Item: {0} - Vendor: {1}",
                    row["ProductID"],
                    row["Vendor"]);
}

// Поиск строк по заданному составному критерию.
DataRow[] compoundRows = dataTable.Select("Vendor =
                         'Wilson' " + "AND Price > 20.00");
Console.WriteLine ("Compound Select Search:");

// Вывод строк, соответствующих
// заданному критерию, на консоль.
foreach (DataRow row in compoundRows)
{
```

которых начинаются с Microsoft, вне зависимости от того, сколько еще символов находится после первых девяти.

Родительские отношения

```
dataView.RowFilter = "Parent.Birthdate < #01/01/1978#";
```

Ссылаться на данные таблицы-родителя фильтруемых строк можно с помощью префикса Parent.

Дочерние отношения

```
dataView.RowFilter - "Child.InvDate >= #01/01/2001#";
dataView.RowFilter = "Child(Cust2Inv).InvDate >= " +
                     "#01/01/1969#";
```

Точно так же, ссылаться на данные в дочерней таблице фильтруемых строк можно с помощью префикса Child, если у таблицы есть только одно дочернее отношение. В противном случае необходимо указывать имя дочернего отношения.

Фильтрация по свойству RowState с помощью объекта DataView

Объект DataView поддерживает фильтрацию по состоянию строк в объекте DataTable, как показано в листинге 7.15.

Листинг 7.15. Фильтрация по свойству *RowState* с помощью объекта *DataView*

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Определение фильтра для просмотра первоначальных данных
// таблицы (для измененных строк будет отображено их
// исходное состояние).
dataTable.DefaultView.RowStateFilter =
                        DataViewRowState.OriginalRows;
```

Для того чтобы провести фильтрацию на основе свойства RowState, необходимо воспользоваться перечислением DataViewRowState, которое поддерживает как фильтрацию на основе свойства RowState, так и отображение различных версий строки. Ниже приведено краткое описание каждого элемента этого перечисления.

- `CurrentRows` — отображает все версии строки, кроме удаленной. Это значение используется по умолчанию.
- Added — отображает строки, добавленные после последнего вызова метода `DataTable.AcceptChanges()`.
- Deleted — отображает строки, удаленные с момента последнего вызова метода `DataTable.AcceptChanges()`.
- `ModifiedCurrent` — отображает текущую версию измененных строк. Строка считается измененной, если было изменено значение хотя бы одного ее столбца. При использовании этого значения объект **DataView** отобразит только те строки, которые были изменены с момента последнего вызова метода `DataTable.AcceptChanges()`.
- `ModifiedOriginal` — отображает первоначальную версию измененных строк. При использовании этого значения объект **DataView** отобразит только те стро-

Фильтрация данных с помощью объекта DataView

Класс DataView поддерживает фильтрацию данных с помощью свойства DataView.RowFilter. Свойство RowFilter представляет собой строку, описывающую критерий фильтрации. Пример фильтрации данных с помощью объекта DataView показан в листинге 7.14.

Листинг 7.14. Фильтрация данных с помощью объекта *DataView*

```
...

// Определение переменной для упрощения доступа к строке.
DataTable dataTable = dataSet.Tables[0];

// Определение фильтра.
dataTable.DefaultView.RowFilter = "Vendor = 'Rawlings'";

// Подсчет строк, удовлетворяющих фильтру.
Console.WriteLine("Rawlings Products: {0}",
                  dataTable.DefaultView.Count);
```

Синтаксис строки критерия фильтра идентичен синтаксису свойства DataColumn.Expression[1] и представляет собой синтаксис клиентского языка запросов (языка запросов к отсоединенным данным). В следующих разделах приведены некоторые примеры установки свойства RowFilter.

Текстовые фильтры

```
dataTable.DefaultView.RowFilter = "FirstName = 'John'";
```

Простые текстовые сравнения принимают форму точного соответствия. Тем, кто привык использовать оператор сравнения (==) языка C#, стоит быть внимательными, так как в этом синтаксисе оператор сравнения имеет форму =. Кроме этого, оператор неравенства (о) отличается от оператора неравенства языка C# (!=).

Фильтр по дате

```
dataView.RowFilter = "BirthDate = #04/24/1969#";
```

В качестве критерия фильтрации можно использовать дату, которую необходимо окружить символами диеза (ft).

Операторы сравнения

```
dataView.RowFilter = "Cost > 24.99";
```

Поддерживаются операторы сравнения <, >, <= и >=.

Составные выражения

```
dataView.RowFilter = "(Cost <= 24.99) AND (Cost > 50.00)";
```

Для составных выражений поддерживаются булева конкатенация (AND, OR и NOT) и группировка с помощью скобок.

Групповые символы

```
dataView.RowFilter = "Company LIKE 'Microsoft*'";
```

В рамках синтаксиса LIKE для текстового сравнения поддерживаются групповые символы — * и %. Приведенный выше фильтр пропустит все компании, названия

[1] **В первоначальной версии Visual Studio .NET ссылка MSDN была такой:** msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataDataColumnClassExpressionTopic.asp.

Объект DataView

Класс Dataview используется для осуществления привязки данных к элементам управления Windows-форм и Web-форм **ASP.NET.** (Более подробно Windows- и Web-формы рассматриваются в главе 10, "Привязка данных в ADO.NET".) В этом разделе будет продемонстрировано использование объекта **DataView** для поиска, сортировки и фильтрации содержимого объекта **DataTable.** Каждый объект **DataTable** содержит объект **DataView. Объект DataView,** используемый по умолчанию, называется DataTable. Defaultview, но вы можете создать столько объектов **DataView,** сколько необходимо. Это может оказаться полезным при отображении одних и тех **же** данных, отфильтрованных или отсортированных различным образом (листинг 7.12).

Листинг 7.12. Объект *DataView,* используемый по умолчанию

```
...

// Создание переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Подсчет количества строк в представлении,
// используемом по умолчанию.
Console.WriteLine("DefaultView Count: {0}",
                  dataTable.DefaultView.Count);

// Создание нового объекта DataView,
DataView sortedView = new DataView(dataTable);
sortedView.Sort = "LastName";
```

В этом примере используются два различных представления — представление, принятое по умолчанию, и представление sortedView. Они никак не влияют на данные; можно сказать, что они содержат на них различную "точку зрения".

Сортировка данных с помощью объекта DataView

Класс DataView поддерживает сортировку данных с помощью свойства DataView. Sort. Свойство Sort представляет собой строку, описывающую критерий сортировки. Для того чтобы провести сортировку с помощью объекта **DataView,** просто установите значение свойства Sort, использовав для этого имя одного или нескольких столбцов с модификатором ASC или DESC, определяющим порядок сортировки (по возрастанию или по убыванию). На практике по умолчанию используется возрастающий порядок сортировки, поэтому его можно не указывать (листинг 7.13).

Листинг 7.13. Сортировка данных с помощью объекта *DataView*

```
...

// Создание переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Сортировка по LastName (по возрастанию).
dataTable.DefaultView.Sort = "LastName";

// Сортировка по LastName (по убыванию).
dataTable.DefaultView.Sort = "LastName DESC";

// Сортировка по LastName (по возрастанию),
// а затем по FirstName (по убыванию).
dataTable.DefaultView.Sort = "LastNarae, FirstName DESC";
```

```
    Decimal grandTotal = 0;

    // Вывод атрибутов счета.
    DataRow[] itemRows =
                        invRow.GetChildRows("Invoice_Item");
    foreach (DataRow itemRow in itemRows)
    {
      DataRow productRow =
                      itemRow.GetChildRows("Item_Product")[0];
      int quantity = ((int)itemRow["Quantity"]);
      Decimal total = ((Decimal)productRow["Price"]) *
                      quantity;
      grandTotal += total;
      Console.WriteLine("    {0} {1} @ {2:C} ea",
                        productRow["Description"],
                        quantity,
                        total);
    }
    Console.WriteLine("    Total: {0:C}", grandTotal);
  }

}
```

В результате выполнения этого фрагмента кода будет получен следующий результат:

```
Maddux, Greg
  100 : 3/21/2002
    Basketball 2 @ $29.00 ea
    Baseball Mitt 3 @ $89.97 ea
    Total: $327.91
  101 : 3/31/2002
    Baseball Bat, Slugger 1 @ $19.99 ea
    Total: $19.99
Glavine, Tom
  102 : 3/24/2002
    Baseball Mitt 1 @ $29.99 ea
    Total: $29.99
Smoltz, John
  103 : 3/27/2002
    Baseball Bat, Slugger 1 @ $19.99 ea
    Total: $19.99
  104 : 4/1/2002
    Baseball 1 @ $1.75 ea
    Total: $1.75
```

Вы видите, что мы получаем только непосредственно связанные строки. Так, для получения массива объектов **DataRow,** связанных с помощью отношения с дочерней таблицей, был использован метод GetChildRows(). В последний раз при использовании метода GetChildRows() мы знали, что только одна строка таблицы Product содержит информацию о товаре, а поэтому достаточно было извлечь лишь первую строку. Если этот фрагмент будет присутствовать в конечном коде приложения, в него необходимо добавить код, проверяющий, что возвращаемые массивы объектов **DataRow** не содержат значения null (это происходит, когда дочерняя таблица не содержит подходящих строк).

Для получения родительских строк класс DataRow предоставляет два практически идентичных метода: GetParentRows() и GetParentRow(). Единственная разница между ними заключается в том, что первый метод возвращает массив объектов **DataRow,** а второй — единственную (первую) строку.

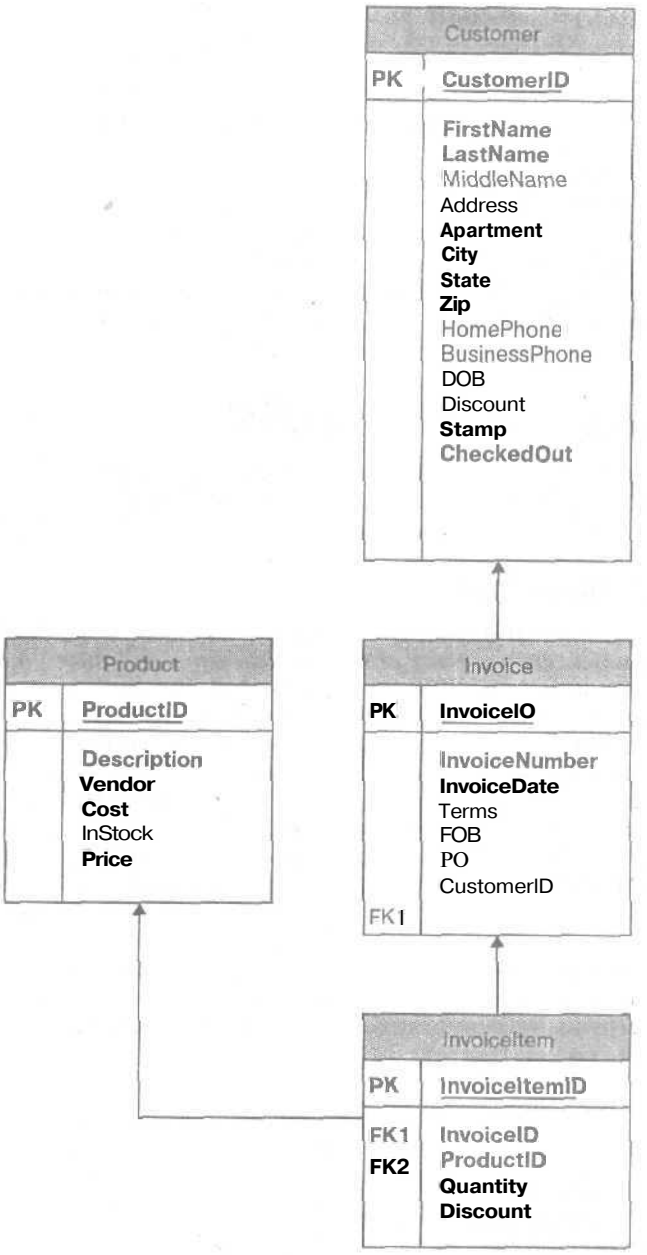


Рис. 7.2. Отношения между четырьмя таблицами

```
foreach(DataRow invRow in invoiceRows)
{
  string invoiceDate =
    ((DateTime)invRow["InvoiceDate"]).ToShortDateString();
  Console.WriteLine("  {0} : {1}",
                    invRow["InvoiceNumber"],
                    invoiceDate);
```

```
Console.WriteLine("Unchanged Row State:\t{0}",
                  unchangedRow.RowState);
```

В результате выполнения этого фрагмента кода на консоль будет выведен следующий текст:

```
New Row State:          Added
Deleted Row State:      Deleted
Changed Row State:      Modified
Unchanged Row State:    Unchanged
```

В этом примере демонстрируется изменение свойства Rowstate, отражающего текущее состояние строки. После обновления хранилища данных состояние строки устанавливается равным RowState.Unchanged. Более подробно обновление базы данных рассматривается в главе 8, "Обновление базы данных".

Перемещение по объекту DataSet

Перемещение по объекту DataSet является чем-то большим, чем простой перебор строк в таблице. ADO.NET предоставляет мощные механизмы, предназначенные для перемещения по хранящейся в объекте DataSet информации.

Перемещение вдоль отношений

Перемещение вдоль отношений — одна из наиболее существенных возможностей объекта DataSet. Отношения, допускающие перемещение, позволяют рассматривать данные как иерархический объектный граф. В отличие от отношений в схеме базы данных, по отношениям между таблицами DataTable можно перемещаться. В схеме базы данных объединение таблиц осуществляется в тех же точках, которые используются при определении отношения (таких, как внешние ключи). Результат объединения двух таблиц базы данных может содержать повторяющуюся информацию в том случае, когда одна строка таблицы-родителя соответствует нескольким дочерним строкам. В ADO.NET все по-другому — каждая таблица остается сама по себе, а объект DataSet позволяет перемещаться по отношению между таблицами на уровне строк.

На рис. 7.2 показаны отношения между таблицами Customer, Invoice, InvoiceItem и Product, которые позволяют перемещаться по этим таблицам в обоих направлениях. Для того чтобы переместиться вверх или вниз по отношению, необходимо воспользоваться объектом DataRow, поддерживающим перемещение как к дочерним, так и к родительским строкам (листинг 7.11).

Листинг 7.11. Перемещение по отношениям

```
...

// Итерация по таблице клиентов
// с выводом соответствующих счетов.
foreach (DataRow custRow in
         dataSet.Tables["Customers"].Rows)
{
  // Вывод имени клиента.
  Console.WriteLine("{0}, {1}", custRow["LastName"],
                                custRow["FirstName"]);

  // Вывод счетов.
  DataRow[] invoiceRows =
                    custRow.GetChildRows("Customer_Invoice");
```

Проверить, имеет ли строка DataRow добавочные версии, можно с помощью вызова метода DataRow.HasVersion(). Этот метод очень полезен, когда вам необходимо получить определенную версию строки. Версия DataRowVersion.Default всегда зависит от состояния строки. Если же версия строки не будет указана, то по умолчанию возвращается версия DataRowVersion.Default.

Состояние строки

При каждом изменении строки (включая добавление и удаление) в объекте DataRow сохраняется ее текущее состояние, на основании которого осуществляется обновление хранилища данных (листинг 7.10).

Листинг 7.10. Получение состояния строки

```
...

// Определение переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Добавление строки.
DataRow newRow - dataTable.NewRow();

newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Millwood";
newRow["FirstName"] = "Kevin";
dataTable.Rows.Add(newRow);

// Вывод на консоль состояния строки.
Console.WriteLine("New Row State:\t\t{0}", newRow.RowState);

// Определение переменной для облегчения
// доступа к первой строке.
DataRow deleteRow = dataSet.Tables[0].Rows[0];

// Удаление строки.
deleteRow.Delete();

// Вывод на консоль состояния строки.
Console.WriteLine("Deleted Row State:\t\t{0}",
                  deleteRow.RowState);

// Определение переменной для облегчения
// доступа ко второй строке.
DataRow changeRow = dataSet.Tables[0].Rows[1];

// Изменение строки.
changeRow["DOB"] - "04/24/1969";

// Вывод на консоль состояния строки.
Console.WriteLine("Changed Row State:\t{0}",
                    changeRow.RowState);

// Определение переменной для облегчения
// доступа к третьей строке.
DataRow unchangedRow = dataSet.Tables[0].Rows[2];

// Вывод на консоль состояния строки.
```

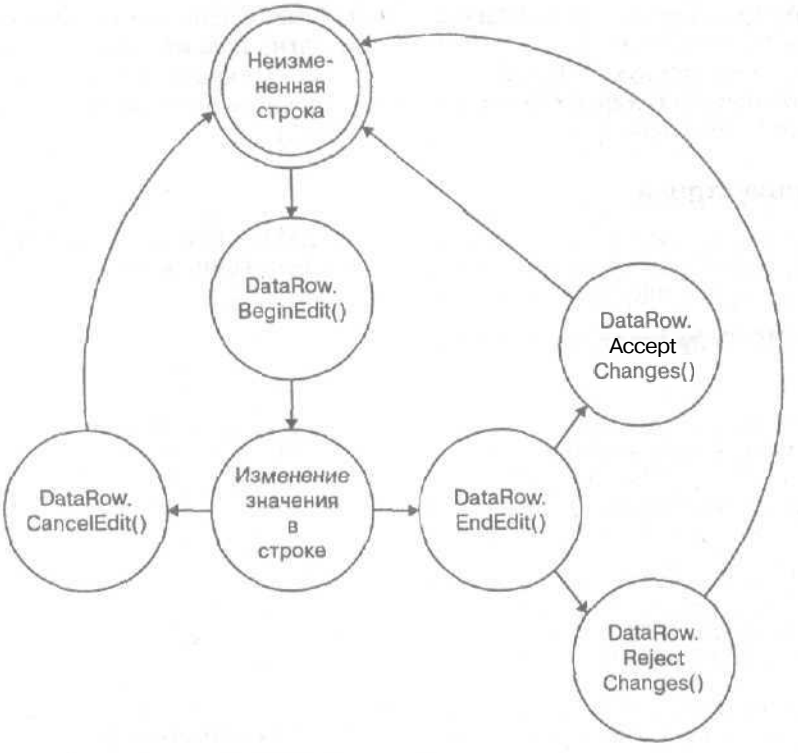


Рис. 7.1. Жизненный цикл строки

Конечно же, эти методы можно вызывать самостоятельно, однако обычно они используются объектом **DataAdapter** для изменения каждого объекта **DataRow** при обновлении базы данных. Важно понять, что если объект DataRow содержит изменения, которые не были приняты или отвергнуты, вы можете получить доступ к обеим версиям строки.

Для определения версии строки класс DataRow использует перечисление DataRowVersion. Ниже приведены элементы, входящие в это перечисление.

- Current — текущее значение строки. После внесения изменений это значение станет предложенным, а после вызова метода AcceptChanges — первоначальным.
- Default — значение строки по умолчанию в соответствии с ее текущим состоянием.
- Original — значение строки сразу же после ее создания или последнего вызова метода AcceptChanges.
- Proposed — измененное значение строки перед вызовом метода `AcceptChanges()`. Это значение становится доступным после завершения редактирования строки.

Для того чтобы получить доступ к конкретной версии строки, следует указать значение `DataRowVersion` в индексаторе, как показано ниже:

```
string oldString = row["FirstName",
                   DataRowVersion.Original];
string newString = row["FirstName",
                   DataRowVersion.Proposed]
```

```
dataTable.LoadDataRow(row, true);

// Завершение загрузки данных
//  (включение ограничений, уведомлений
// и индексов).
dataTable.EndLoadData();
```

В этом примере демонстрируется единовременное добавление строки с помощью метода `LoadDataRow()`. Сигнатура этого метода очень похожа на сигнатуру метода, позволяющего добавлять строку в виде массива значений. Если значение некоторого столбца в строке **DataRow** генерируется автоматически или если необходимо использовать значение по умолчанию, в качестве значения такого столбца следует передать объект **System.Object.Empty**.

Второй параметр метода `LoadDataRow()` определяет действие, которое должно выполняться после вызова данного метода. Если параметр равен true, то сразу же после вызова метода `LoadDataRow()` будет вызван метод `DataTable.AcceptChanges()`, сообщающий объекту **DataTable** о необходимости принятия всех внесенных изменений и дополнений. Это никак не влияет на саму базу данных, просто мы сообщаем объекту **DataTable**, что эти данные необходимо рассматривать как принятые. Следует отметить, что при вызове метода `DataTable.AcceptChanges` О будут приняты все внесенные в объект **DataTable** изменения (а не только те из них, которые связаны с добавлением строки **DataRow).** Если в качестве второго параметра метода `LoadDataRow()` указать `false`, строка **DataRow** будет помечена как вставленная и может быть добавлена в базу данных при следующем обновлении последней.

Версия строки

При работе с большинством старых уровней доступа к данным все действия по редактированию строк проходили одновременно в классе и в базе данных. Так как ADO.NET поддерживает отсоединенный режим доступа к данным, ни одно изменение не отразится на базе данных до тех пор, пока объект **DataSet** не получит об этом явное уведомление. Поскольку изменения не вносятся немедленно, в объекте **DataSet** необходимо хранить информацию об изменившихся данных. По идее мы могли бы предположить, что для каждой строки или значения в строке существует флаг `IsDirty`. На самом деле такой флаг был бы бесполезен из-за возможности возникновения необходимости отмены изменений или получения информации о первоначальной версии заданной строки с целью осуществления проверки параллелизма. Для поддержки подобной функциональности в объекте **DataRow** хранится до двух копий строки, а также флаг, описывающий ее состояние (сложный флаг `IsDirty`).

При редактировании определенной строки поддерживаются две ее версии: первоначальная и предлагаемая (если строке присваивается новое значение) версии. Жизненный цикл строки показан на рис. 7.1.

Как следует из диаграммы, при редактировании строки **DataRow** она помечена как измененная до тех пор, пока не будет вызван метод Acceptchanges {) или `RejectChanges()`. Выполнение этих методов заставляет объект **DataRow** выполнять одно из двух действий.

- AcceptChanges — первоначальное значение заменяется предложенным значением, при этом первоначальное значение удаляется. Строка **DataRow** помечается как неизмененная.
- RejectChanges — предложенное значение удаляется, а строка помечается как неизмененная.

```
// Завершение внесения изменений в строку.
row.EndEdit() ;

// Начало внесения изменений в строку.
row.BeginEdit O ;

// Установка значения столбца типа DateTime
// (выполняется успешно, так как указанное
// значение легко преобразуется в значение типа DateTime).
row["DOB"] = "04/24/1969";

// Мы хотим отменить изменения, для
// чего вызывается метод CancelEdit().
row.CancelEdit();
```

Массовая загрузка данных в объект DataTable

Необходимость отключения уведомлений и ограничений может потребоваться также при загрузке большого объема информации в объект **DataTable.** Класс DataTable реализует эту возможность посредством методов `BeginLoadData()` и `EndLoadData()`. Кроме подавления уведомлений и ограничений, во время загрузки данных не поддерживаются индексы, что в целом приводит к существенному ускорению загрузки. Для фактического добавления или изменения строк используется метод LoadDataRow() класса DataTable, как показано в листинге 7.9.

Листинг 7.9. Массовая загрузка данных в объект *DataTable*

```
...

// Определение переменной для упрощения доступа к строке.
DataTable dataTable = dataSet.Tables[0];

// Начало загрузки данных
// (отключаются ограничения, уведомления
// и индексы).
dataTable.BeginLoadData();

// Создание строки.
object[] row = new object[dataTable.Columns.Count];
row[0] = Guid.NewGuid();
row[1] = "Kevin";
row[2] - "Millwood";

// Добавление строки без
// подтверждения внесения
// изменений.
dataTable.LoadDataRow(row, false);

// Создание строки.
row - new object[dataTable.Columns.Count];
row[0] = System.Object.Empty;
row[1] = "Damian";
row[2] = "Moss";

// Добавление строки с
// подтверждением внесения
// изменений.
```

```
// Установить подобное значение не удастся,
// так как формат даты неправилен.
row["DOB"] = "04/31/1969";

// Установить подобное значение также не удастся,
// поскольку оно не может быть преобразовано в
// значение типа float (в отличие, например, от
// значения ".15").
row["Discount"] = "15%";
```

В этом примере при попытке внесения двух последних изменений будут сгенерированы исключения, сообщающие о невозможности преобразования типа. Индексатор не проводит проверку типов, его поведение больше похоже на вызов метода `Convert.ChangeType()`. В большинстве случаев объект DataRow пытается самостоятельно провести преобразование типа, что не всегда является приемлемым. Я бы посоветовал вам самостоятельно проводить явное преобразование значений к нужному типу. Как правило, в результате этого мы получаем более устойчивый к ошибкам код, чем при проведении преобразования "за кулисами",

Пакетные изменения строки

Существуют ситуации, в которых изменения в конкретную строку DataRow необходимо вносить параллельно. Обычно это делается тогда, когда одно изменение приводит к нарушению некоторого ограничения или когда вам необходима возможность отмены изменений перед их внесением в базу данных. Предположим, что у нас есть таблица с ограничением, требующим, чтобы один из двух столбцов не был равен `null` и чтобы оба столбца к тому же не были заполнены одновременно. Если попытаться создать код, реализующий это ограничение, то придется провести множество бессонных ночей в поисках промежуточного состояния строки, все еще удовлетворяющего ограничению. Для того чтобы обойти эту проблему, класс DataRow предоставляет методы `BeginEdit`, `EndEdit` и CancelEdit. Как только будет вызван метод `BeginEdit()`, изменения перестанут отражаться на объекте DataRow до тех пор, пока не будет вызван метод `EndEdit()`. Если внесенные изменения окажутся ошибочными, можно вызвать метод `CancelEdit()`, который вернет строку в первоначальное состояние (состояние, в котором она находилась до вызова метода BeginEdit). Самое главное, что на протяжении всего процесса изменения строки не проверяются ограничения и не срабатывают события. Пример внесения пакетных изменений приведен в листинге 7.8.

Листинг 7.8. Пакетные изменения строки *DataRow*

```
...

// Определение переменной для упрощения доступа к строке.
DataRow row = dataSet.Tables[0].Rows[0];

// Начало внесения изменений в строку.
row.BeginEdit();

// Установка нескольких корректных значений.
row["LastName"] = "Millwood";
row["FirstName"] = "Kevin";

// Установка значения типа Double.
row["Discount"] = .15;
```

```
                    dataTable.Columns[x].DataType.Name);
    }
    else
    {
      Console.WriteLine("  {0} : {1}",
                    obj.GetType().Name,
                    obj);
    }
  }
```

В данном примере осуществляется проверка типа объекта на его равенство Guid. Если это действительно Guid, пользователю выдается сообщение о том, что значение столбца должно быть предназначено только для чтения (особенно если данные получены непосредственно из базы данных). Осуществляется также проверка типа объекта на его равенство DBNull (специальный тип, используемый в качестве значения null в базе данных). В случае обнаружения значения DBNull единственным местом, содержащим информацию об ожидаемом базой данных типе, является объект **DataColumn**, соответствующий этому столбцу. В нашем примере пользователю выдается сообщение, в котором указывается тип столбца.

Альтернативой проверке типа столбца на DBNull является опрос строки на предмет равенства значения столбца null с помощью метода DataRow.IsNull(). Применяя этот метод, можно использовать те же самые переопределенные варианты, которые существуют для индексатора (имя, порядковый номер или объект DataColumn), например:

```
if  (row.IsNull("LastName"))  ...
if  (row.IsNull(1))  ...
if  (row, IsNull (lastNameColumn) )  ...
```

Помимо чтения данных, нам необходимо иметь возможность манипулировать ими. Как можно было ожидать, классы DataTable, DataRowCollection и DataRow поддерживают выполнение простых CRUD-операций.

Индексаторы класса DataRow позволяют установить новые значения столбцов строки. Будьте осторожны: при определении нового значения столбца объект DataRow сгенерирует исключение InvalidCastException в том случае, если это значение будет конфликтовать со свойством DataType объекта DataColumn. Несмотря на то что точного соответствия типу, определенному в объекте **DataColumn**, не требуется, он должен поддаваться преобразованию в тип столбца (листинг 7.7).

Листинг 7.7. Добавление значений, поддающихся преобразованию

```
...

// Определение переменной для упрощения доступа к строке.
DataRow row = dataSet.Tables[0].Rows[0];

// Установка корректных значений столбцов строки.
row["LastName"] = "Millwood";
row["FirstName"] = "Kevin";

// Установка значения столбца типа Double.
row["Discount"] = .15;

// Установка значения столбца типа DateTime
//  (выполняется успешно, так как указанное
// значение легко преобразуется в значение типа DateTime).
row["DOB"] = "04/24/1969";
```

```
Console.WriteLine("=============");
foreach (DataRow row in dataTable.Rows)
{
  Console.WriteLine("{0}, {1}", row[1], row[2]);
)

// Создание переменных для упрощения доступа к столбцам.
DataColumn lastNameColumn = dataTable.Columns["LastName"];
DataColumn firstNameColumn = dataTable.Columns["FirstName"];

// Вывод имени каждого клиента на консоль
// с использованием объекта DataColumn.
Console.WriteLine("Customer List");
Console.WriteLine("=============");
foreach (DataRow row in dataTable.Rows)
{
  Console.WriteLine("{0}, {1}", row[lastNameColumn],
                                row[firstNameColumn]);
}
```

Индексатор класса DataRow возвращает значение столбца строки в виде экземпляра класса System.Object. Следует отметить, что этому значению известен его тип (как и всем типам .NET). Вы можете привести значение столбца к нужному типу, а можете использовать его без приведения. **Кроме** информации о типе, существует доступ к коллекции Column объекта **DataTable,** содержащей информационную схему таблицы. Более подробно объект **DataColumn,** выступающий в качестве контейнера простой информационной схемы столбца, рассматривался в главе 5. Делая запросы к объекту **DataColumn,** на этапе выполнения можно принять решение о том, как обрабатывать данные, содержащиеся в столбце, или что делать при их отсутствии. В большинстве случаев этот подход неэффективен, однако при работе с аморфными данными такой механизм может оказаться очень полезным (листинг 7.6).

Листинг 7.6. Выполнение запросов к объекту *DataColumn*

```
...

// Вывод имени каждого клиента на консоль
// с использованием имени столбца.
Console.WriteLine("Customer List");
Console.WriteLine("=============");
foreach (DataRow row in dataTable.Rows)
{
  Console.WriteLine("{0}, {1}:", row["LastName"],
                                 row["FirstName"]);
  for (int x = 0; x < row.ItemArray.GetLength(0); ++x)
  {
    object obj = row.ItemArray[x];
    if (obj.GetType() == typeof(Guid))
    {
      Console.WriteLine("  {0} : {1} *Should be readonly*",
                        obj.GetType().Name,
                        obj);
    }
    else if (obj.GetType() == typeof(DBNull))
    {
      Console.WriteLine("  {0} : ColumnType: {1}",
                        obj.GetType().Name,
```

```
// Удаление строки на основе заданного экземпляра.
dataTable.Rows.Remove(newRow);

// Создание переменной для упрощения доступа к первой строке.
DataRow row = dataSet.Tables[0].Rows[0];

// Удаление строки.
row.Delete();

// Удаление строки на основе порядкового номера.
dataTable.Rows.RemoveAt(2);
```

Чтение и запись значений столбцов строки

Вывод данных, хранящихся в объекте **DataSet,** может быть осуществлен путем итерации по объектам **DataRow** заданного объекта **DataTable, как** показано в листинге 7.4.

Листинг 7.4. Чтение значений столбцов строки

```
...

// Создание переменной для упрощения доступа к таблице.
DataTable dataTable - dataSet.Tables[0];

// Вывод имени каждого клиента на консоль.
Console.WriteLine("Customer List");
Console.WriteLine("=============");
foreach (DataRow row in dataTable.Rows)
{
  Console.WriteLine("{0}, {1}", row["LastName"],
                                row["FirstName"]);
}
```

В этом примере продемонстрирован вывод данных, хранящихся в одной таблице объекта **DataSet.** Ключевым моментом здесь является доступ к объектам **DataRow** заданного объекта **DataTable.** Класс DataRow поддерживает синтаксис строкового индексатора (свойство Item в VB.NET) для получения значения столбца конкретной строки (например, `row["LastName"]`). Кроме этого, индексатор класса DataRow поддерживает использование порядковых номеров и объектов **DataColumn,** представляющих столбец таблицы **DataTable,** как показано в листинге 7.5.

Листинг 7.5. Индексаторы класса *DataRow*

```
...

// Вывод имени каждого клиента на консоль
// с использованием имени столбца.
Console.WriteLine("Customer List");
Console.WriteLine ( "=============");
foreach (DataRow row in dataTable.Rows)
{
  Console.WriteLine("{0}, {1}", row["LastName"],
                                row["FirstName"]);
}

// Вывод имени каждого клиента на консоль
// с использованием порядкового номера столбца.
Console.WriteLine("Customer List");
```

Следует отметить, что метод DataTable.NewRow() сам по себе не добавляет строку в объект **DataTable.** Для этого необходимо вызвать метод DataTable.Rows.Add(), передав ему в качестве параметра объект строки.

Новые строки не обязательно добавляются в конец объекта **DataTable.** Если вам необходимо вставить новую строку в середину коллекции, воспользуйтесь методом DataTable.Rows. InsertAt (), как показано в листинге 7.2.

Листинг 7.2. Вставка объекта *DataRow* в середину коллекции строк объекта *DataTable*

```
...
// Добавление новой строки путем создания объекта DataRow.
DataRow insertedRow = dataTable.NewRow();
insertedRow["CustomerID"] = Guid.NewGuid();
insertedRow["LastName"] = "Marquis";
insertedRow["FirstName"] = "Jason";
dataTable.Rows.InsertAt(insertedRow, 1);
```

Удаление строк

При использовании отсоединенных данных к удалению строки из коллекции предъявляется особое требование: строка должна продолжать существовать до тех пор, пока хранилище данных не будет обновлено с помощью объекта **DataSet.** Тем не менее сейчас нам это знать не обязательно. Удаление строки может быть осуществлено одним из трех способов:

- с помощью метода DataTable.Rows.Remove() на основе заданного экземпляра класса DataRow;
- с помощью метода DataTable.Rows.RemoveAt 0 на основе заданного порядкового номера строки;
- с помощью метода DataRow. Delete() (строка удаляет себя сама).

Все три способа можно использовать для удаления заданных строк. При этом важно понимать, что удаление строки может повлечь за собой изменение порядковых номеров строк. Если добавить и удалить строку перед тем, как изменения будут приняты объектом **DataTable,** то система фактически удалит элемент из коллекции. При удалении элемента, который существует за пределами объекта **DataTable,** строка сохраняется, но помечается как удаленная — информация, хранящаяся в строке, еще понадобится для проведения обновления хранилища данных. Таким образом, использование порядкового номера строки очень ненадежно, так как он может изменяться каждый раз при удалении или добавлении элементов. Различные способы удаления строк показаны в листинге 7.3.

Листинг **7.3, Удаление** строк

```
...
// Создание переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0];

// Добавление строки путем создания нового объекта DataRow.
DataRow newRow = dataTable.NewRow();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Millwood";
newRow["FirstName"] = "Kevin";
dataTable.Rows.Add(newRow); // <- Важно!
```

информацией и затем отправлять ее на хранение обратно в базу данных, объект DataSet является единственно правильным выбором. А вот использование объекта DataSet для создания простого отчета (например, Web-страницы) обычно является ошибкой. "Обычно", потому что из этого правила тоже существуют исключения.

Рассмотрим конкретную проблему — Web-узел электронной коммерции. Предполагая, что у нас есть много продаваемых товаров, я бы воспользовался объектом DataReader для предоставления списка товаров пользователю и объектом DataSet для поддержки "корзины" покупателя. Кэширование списка товаров привело бы к потере эффективности и разрастанию объема кэша. Обычно кэширование часто используемых элементов стоит поручить базе данных. В большинстве случаев такое решение полностью оправдано. С другой стороны, объект DataSet, содержащий информацию о пользователе и его "корзину", может быть кэширован в Web-сеансе с целью сократить количество обращений к базе данных. Если посетитель так ничего и не купит, его "корзина" просто исчезнет (и не будет ни разу сохранена) при уничтожении сеанса.

Запомните простое эмпирическое правило: если вам нужно что-то кэшировать, воспользуйтесь объектом DataSet, в противном случае обратите внимание на объект DataReader.

Добавление строк

Добавлять новые строки к объекту DataTable довольно просто. Вы можете либо создать массив значений и добавить его к коллекции строк в качестве новой строки, либо создать новую строку с помощью объекта **DataTable,** как показано в листинге 7.1.

Листинг 7.1. Добавление нового объекта *DataRow*

```
...

// Создание переменной для упрощения доступа к таблице.
DataTable dataTable = dataSet.Tables[0] ;

// Добавление новой строки путем создания объекта DataRow.
DataRow newRow = dataTable.NewRow();
newRow["CustomerID"] = Guid.NewGuid();
newRow["LastName"] = "Millwood";
newRow["FirstName"] = "Kevin";
dataTable.Rows.Add(newRow); // <- Важно!

// Добавление новой строки путем создания
// массива значений столбцов.
object[] newValues = new object[dataTable.Columns.Count];
newValues[0] = Guid.NewGuid();
newValues[1] = "Damian";
newValues[2] = "Moss";
dataTable.Rows.Add(newValues); // <- Важно!
```

Метод DataRowCollection.Add() поддерживает оба способа добавления новой строки. Какой из них лучше использовать в каждом конкретном случае? Я предпочитаю использовать метод DataTable.NewRow(), так как он позволяет заполнить новую строку значениями, гарантируя их соответствие требованиям столбцов. Другими словами, значения в новом объекте **DataRow** удовлетворяют схеме объекта **DataTable,** что делает невозможным добавление некорректных значений. К тому же явное обращение к элементам строки сделает код более читабельным. Действительно, первый способ добавления нового объекта **DataRow** намного более прозрачен, поскольку можно видеть, что нулевой столбец представляет собой идентификатор клиента, первый столбец — его имя, а третий столбец — его фамилию.

Глава 7

Манипулирование объектом DataSet

В этой главе...

Изменение данных
Перемещение по объекту DataSet
Поиск в объекте DataSet
Слияние объектов DataSet

Прочтя главы 5 и 6, мы научились создавать объекты **DataSet.** Теперь возникает вопрос: что с ними делать? Данные, которые хранятся внутри объекта **DataSet,** содержат не только информацию, необходимую для поддержки отсоединенного кэша базы данных, но также предоставляют возможность перемешаться по нему как по некоторой иерархической структуре.

Изменение данных

Основным предназначением объекта **DataSet** является хранение и изменение данных. Вне зависимости от того, был он создан в результате чтения информации из базы данных или программным путем, объект **DataSet** обязательно содержит данные.

Объекты **DataRow** являются основным хранилищем данных внутри объекта **DataSet.** Объект **DataRow** содержит массив значений, представляющий собой строку объекта **DataTable.** Объекты **DataRow** доступны из объекта **DataTablc** через свойство Rows, являющееся коллекцией объектов **DataRow** (для этого используется класс DataRowCollection). Объект **DataRow** отличается от традиционных коллекций полей, поскольку он, помимо текущего значения каждого столбца в строке, содержит в себе некоторую дополнительную информацию. Она представляет собой следующее:

- текущее значение каждого столбца в строке;
- первоначальное значение каждого столбца в строке, если она была изменена;
- состояние столбца (добавлен, удален, изменен и т.д.);
- средство перехода к родительской или дочерней строке (при наличии отношений).

DataSet или *DataReader*?

Общаясь с разработчиками, я часто слышу вопрос: "Какой подход является наиболее правильным?". Ответ на него целиком и полностью зависит от предметной области. Объект **DataReader** представляет собой однонаправленный курсор, предназначенный для чтения данных. В контексте создания отчетов на основе информации, хранящейся в базе данных, зачастую именно его использование является наиболее оптимальным решением. Однако если мы будем полагаться лишь на объект **DataReader,** нам, возможно, потребуется создать собственный контейнер, подобный DataSet, предназначенный для хранения считанной строки данных.

Объект **DataSet, с** другой стороны, является отличным способом хранения информации, извлеченной из базы данных. Тем не менее его заполнение связано с накладными расходами и негативно влияет на производительность системы. Если вам необходимо манипулировать

Переопределив индексаторы, мы заставляем пользователей применять для установки отдельных значений свойства, что позволяет не дублировать бизнес-логику или перенаправлять вызовы через свойства.

В рассмотренных выше примерах классы порождались от классов DataTable и DataRow, соответствующих таблице счетов. Однако наш типизированный класс DataSet имеет также обеспечивающие безопасность типов классы DataTable и DataRow, соответствующие таблице клиентов. Поскольку к ним не добавляется бизнес-логика, они не порождают новых классов. Другими словами, при наследовании типизированного класса DataSet нет необходимости наследовать все его классы DataTable, а достаточно наследовать только те классы, к которым необходимо добавить бизнес-логику.

Новые классы создавались нами путем ручного редактирования сгенерированного кода. Поскольку сгенерированный код может изменяться, обычно такие действия не приветствуются. Для того чтобы облегчить наследование, Крис Селлз (Chris Sells) и я создали дополнительный модуль для Visual Studio, замещающий стандартный генератор типизированных классов DataSet и вносящий все необходимые изменения в генерируемый код. При желании вы можете загрузить этот модуль с моего Web-узла по адресу: www.adoguy.com/book/AGDDataSetGenerator.

Резюме

Безопасность типов — это отличная штука! Создавая типизированные классы DataSet, мы сокращаем количество возможных ошибок и ненужных обращений к базе данных. В этой главе описано создание типизированных классов DataSet с помощью Visual Studio .NET и посредством инструмента командной строки. Мы рассмотрели также способы уточнения генерируемых классов с помощью специальных аннотаций.

Типизированные классы DataSet позволяют реализовать новую парадигму программирования, в соответствии с которой генерируемые классы могут включать в себя бизнес-логику приложения. Код, генерируемый инструментами Microsoft, не совсем удовлетворяет такой парадигме, однако вы всегда можете воспользоваться инструментом, размещенным на моем Web-узле.

```
        }
        set
        {
          if (value > DateTime.Today)
          {
            throw new StrongTypingException(
                       "Invoice Date Cannot be in the future",
                       null);
          }
          else
          {
            base.InvoiceDate = value;
          }
        }
      }
      ...
    }
...
}
```

Наша бизнес-логика предполагает проверку даты счета (не указывает ли она на будущее). Помещение этой логики в свойство позволяет гарантировать, что пользователи не смогут установить дату счета неправильно, но так как базовым классом является класс DataRow, нам необходимо удостоверится, что пользователи не смогут установить дату счета с помощью индексатора (который пропускает вызов нашего свойства). Избежать этого можно посредством переопределения индексатора, сделав его предназначенным только для чтения (листинг 6.19).

Листинг 6.19. Защита бизнес-логики

```
public class InheritedTDS : FixedCustomerTDS
{
...
  public class InheritedInvoiceRow : InvoiceRow
  {
    ...
    // Разрешается только операция чтения.
    public new object this[int index]
    {
      get
      {
        return base[index];
      }
    }
    ...
  }
...
}
```

```
    }
    get
    {
      return ( (InheritedInvoiceRow)(this.Rows[index]) );
    }
  }

  public new InheritedInvoiceRow
              AddInvoiceRow(Guid InvoiceID,
                            DateTime XnvoiceDate,
                            string Terms,
                            string FOB,
                            string PO,
                            CustomerRow
                          parentCustomerRowByCustomerInvoice)
  {
    ...
  }

  public new InheritedInvoiceRow NewInvoiceRow ( )
  {
    ...
  }

...
}

...
}
```

Для начала мы создаем новый индексатор, который будет возвращать экземпляр нового класса DataRow вместо экземпляра базового класса DataRow. После этого создаются методы AddInvoiceRow () и NewInvoiceRow () (созданные в типизированном классе DataSet), которые также возвращают экземпляр нового класса DataRow. Теперь порожденный класс будет обеспечивать безопасность типов и полностью наследовать базовый класс.

После выполнения "технической" работы в порожденный класс DataRow можно поместить необходимую бизнес-логику (листинг 6.18).

Листинг 6.18. Добавление бизнес-логики

```
public class InheritedTDS : FixedCustomerTDS
{

...

  public class InheritedInvoiceRow : InvoiceRow
  {

    ...

    public new DateTime InvoiceDate
    {
      get
      {
        return base.InvoiceDate;
```

В случае наследования типизированного класса DataRow нужно реализовать только один конструктор, принимающий в качестве параметра объект **DataRowBuilder**. Этот конструктор используется методом DataTable.NewRowFromBuilder(), который вызывается объектом **DataTable** при создании новой строки. При вызове данного метода базовый класс обеспечивает безопасность типов. Для того чтобы это работало, необходимо переопределить методы класса DataTable NewRowFromBuilder() и GetRowType() так, как показано в листинге 6.16.

Листинг 6.16. Разрешение создания новых экземпляров порожденного класса *DataRow*

```
public class InheritedTDS : FixedCustomerTDS
{

...

  public class InheritedInvoiceDataTable : InvoiceDataTable
  {

  ...

    protected override DataRow
                  NewRowFromBuilder(DataRowBuilder builder)
    {
      return new InheritedInvoiceRow(builder);
    }

    protected override System.Type GetRowType()
    {
      return typeof(InheritedInvoiceRow);
    }

  ...

  }

...

}
```

Это позволяет удостовериться, что все строки, созданные в объекте **DataTable,** имеют тип порожденного класса (такой, как InheritedInvoiceRow). Нам необходимо также, чтобы объект **DataTable** работал с новыми объектами **DataRow,** а не с их реализацией в базовом классе. Для того чтобы это сделать, нужно создать несколько новых методов, как показано в листинге 6.17.

Листинг 6.17. Переопределение метода создания объектов *DataRow*

```
public class InheritedTDS : FixedCustomerTDS
{

...

  public class InheritedInvoiceDataTable : InvoiceDataTable
  {

    ...

    public new InheritedInvoiceRow this[int index]
```

Теперь при попытке обратиться к объекту DataTable и добавить новый счет можно гарантировать, что клиент имеет достаточный кредит; в противном случае будет сгенерировано исключение. Реализация бизнес-логики на уровне таблицы может оказаться недостаточной. Вполне возможно, нам придется реализовать определенный контроль также и на уровне строк. Для того чтобы удостовериться в том, что ни один из пользователей не сможет создать счет, датированный будущим числом, необходимо добавить к свойству `InvoiceDate` класса `DataRow` бизнес-объект (листинг 6.14),

Листинг 6.14. Защита свойства *InvoiceDate*

```
public new DateTime InvoiceDate
{
  get
  {
    return base.InvoiceDate;
  }

  set
  {
    if (value > DateTime.Today)
    {
      return new StrongTypingException("Invoice Date" +
                                       "cannot be in the" +
                                       "future", null);
    }
    else
    {
      base.InvoiceDate = value;
    }
  }
}
```

Чтобы это осуществить, необходимо наследовать типизированный класс `DataRow`, как показано в листинге 6.15.

Листинг 6.15. Наследование класса *DataRow*

```
public class InheritedTDS : FixedCustomerTDS
{

...

  public class InheritedInvoiceRow ; InvoiceRow
  {

    public InheritedInvoiceRow (DataRowBuilder builder) :
    base (builder)
    }
    }

  ...

  }

...

}
```

Последний шаг заключается в создании нового свойства, позволяющего скрыть свойство DataTable базового класса, как показано в листинге 6.12.

Листинг 6.12. Сокрытие **свойства *DataTable***

```
public class InheritedTDS : FixedCustomerTDS
{

...

  public new InheritedInvoiceDataTable Invoice
  {
    get
    {
      return (InheritedInvoiceDataTable)base.Invoice;
    }
  }

...

}
```

Создав новое свойство Invoice, мы не только скрываем свойство invoice базового класса, но также гарантируем, что весь код, ссылающийся на свойство Invoice, будет работать с версией класса DataTable, производной от типизированного класса DataTable.

Как только внесение изменений завершено, мы помещаем бизнес-логику в класс-потомок типизированного класса DataTable, как показано в листинге 6.13.

Листинг 6.13. Добавление бизнес-логики к производному классу

```
public class InheritedTDS ; FixedCustomerTDS
{

...

public class InheritedInvoiceDataTable : InvoiceDataTable
  t

  ...

    public void AddInvoiceRow(InheritedInvoiceRow row)
    {
      if (DoesCustomerHaveCredit())
      {
        base.AddInvoiceRow(row);
      }
      else
      {
        throw new Exception(
                    "Customer Invoice cannot be created, " +
                    "no credit available");
      }
    }

  ...

  }

...

}
```

Листинг 6.11. **Изменение поведения типизированного объекта *DataSet*, *принятого* по умолчанию**

```
// Сгенерированный класс.
public class FixedCustomerTDS : DataSet
{

...

  private void InitClass()
  {

    ...

    /* Исходный код.
    this.tableInvoice = new InvoiceDataTable();
    */

    // Новый код.
    this.tableInvoice = CreateInvoiceDataTable(null);

    ...

  }

...

  protected FixedCustomerTDS(SerializationInfo info,
                             StreamingContext context)
  {

    ...

    /* Исходный код.
    this.Tables.Add(
    new InvoiceDataTable(ds.Tables["Invoice"]));
    */

    // Новый код.
    this.Tables.Add(
    CreateInvoiceDataTable(ds.Tables["Invoice"]));

    ...

  }

...

}
```

Нам необходимо заменить два фрагмента кода, в которых создается экземпляр типизированного класса DataSet. Каждый из этих фрагментов встречается в сгенерированном коде два раза. Первый фрагмент включает в себя оператор new *ИмяТаблицы*`DataTable()`**, который необходимо заменить на** `Create` *Имя Т а блицы* `DataTable()`**, второй фрагмент — оператор** *ИмяТаблицы*`DataTable(`*НекоторыйТаблица*`)`**, который необходимо заменить на** `Create`*ИмяТаблицы*`DataTable(`*НекоторыйТаблица*`)`**. В предыдущем примере оба фрагмента показаны в исходном и измененном виде.**

Метод CreateInvoiceDataTable возвращает типизированный объект DataTable. Если мы хотим наследовать типизированный класс DataSet, нам придется переопределить данный метод и возвратить тот же самый типизированный объект DataTable. Это позволяет удостовериться в том, что наш порожденный класс DataTable действительно наследует типизированный класс DataTable. Поскольку сгенерированный код будет содержать код, зависящий от типизированного класса DataTable, метод, написанный таким образом, позволяет гарантировать работоспособность сгенерированного кода.

Далее нам необходимо переопределить этот метод в случае наследования от типизированного класса DataSet. Так как метод CreateInvoiceDataTable является виртуальным (переопределяемым в VB.NET), то при создании экземпляров класса DataTable типизированный объект DataSet будет вызывать нашу версию метода CreateInvoiceDataTable. Итак, создадим класс-потомок типизированного класса DataSet и переопределим метод создания таблицы, как показано в листинге 6.10.

Листинг 6.10. Вызов метода создания таблицы

```
public class InheritedTDS : FixedCustomerTDS
{
  public InheritedTDS() : base()
  {
  }

  protected InheritedTDS(SerializationInfo info,
                         StreamingContext context)
    : base(info, context)
  {
  }

  protected override XnvoiceDataTable
                     CreateInvoiceDataTable (DataTable table)
  {
    if (table == null)
    {
      return new InheritedInvoiceDataTable()
                                          as InvoiceDataTable;
    }
    else
    {
      return new InheritedInvoiceDataTable(table)
                                          as InvoiceDataTable;
    }
  }

...

}
```

Данный метод очень похож на метод базового класса, за исключением того что в нем создается экземпляр класса-потомка типизированного класса DataTable, который возвращается как экземпляр базового класса.

После этого необходимо модифицировать сгенерированный код для замены всех вызовов конструкторов типизированного класса DataTable на вызов созданного нами метода (листинг 6.11).

Листинг 6.8. Наследование типизированного класса *DataSet*

```
public class InheritedTDS : FixedCustomerTDS
{
...

  public class InheritedInvoiceDataTable : InvoiceDataTable
  {
    internal InheritedInvoiceDataTable() : base()
    {
    }

    internal InheritedInvoiceDataTable(DataTable table) :
      base(table)
    {
    }

  ...

  }

...

}
```

Наследование типизированного класса DataTable также требует создания двух конструкторов. В качестве параметра второй конструктор принимает объект DataTable и используется для XML-десериализации. Создать класс InheritedInvoiceDataTable было несложно, гораздо труднее будет заставить типизированный класс DataSet использовать полученный класс вместо класса InvoiceDataTable. Сгенерированный код это усложняет, так как схема всего класса (включая класс DataTable) была определена на этапе создания экземпляра базового класса. Это означает, что для того чтобы заменить старый класс DataTable новой версией, нам придется заново создать большую часть схемы, которая находится в конструкторе. Этого можно избежать, если слегка изменить сгенерированный кол одним из нескольких способов. Во-первых, можно добавить новый виртуальный метод, создающий нужную нам таблицу, который будет вызываться на этапе создания типизированного объекта DataSet, как показано в листинге 6.9.

Листинг 6.9. Добавление метода создания таблицы

```
// Сгенерированный класс DataSet.
public class FixedCustomerTDS : DataSet
{

...

  protected virtual InvoiceDataTable
            CreateInvoiceDataTable(DataTable table)
  {
    if (table == null) return new InvoiceDataTable();
    else return new InvoiceDataTable(table);
  }

...
}
```

```
// от типизированного класса DataSet,
CustomersObject dataset = new CustomersObject();

// Использование объектов DataAdapter
// для заполнения объекта DataSet,
daCustomers.Fill(dataset, "Customer");
daInvoices.Fill(dataset, "Invoice");

// Выполнение следующего кода приведет к генерированию
// исключения, так как дата создания счета указывает
// на будущее.
CustomerTDS.InvoiceRow invoice;
Invoice = dataset.Invoice.AddInvoiceRow(Guid.NewGuid(),
                        DateTime.Now + new TimeSpan(4,0,0,0),
                        "",
                        "",
                        "",
                        dataset.Customer[0]);
```

Использование класса-потомка типизированного класса DataSet идентично использованию его родителя. Ну а поскольку мы зарегистрировались для получения уведомлений о событиях, возникающих при проведении определенных операций, это позволяет нам реализовать бизнес-логику. В данном случае при добавлении нового счета его дата намеренно смещается на четыре дня в будущее, что приводит к генерированию исключения с целью уведомить пользователя об ошибке.

Наследование типизированного класса DataSet

Второй подход заключается в том, чтобы породить не только класс-потомок типизированного класса DataSet, но и классы-потомки типизированных классов DataTable и DataRow. Это позволяет переопределить любое поведение при реализации бизнес-логики. Отметим, что наследовать каждый класс DataTable и DataRow не обязательно, достаточно сделать это только для тех классов, которые нуждаются в реализации специальной логики. При этом если нам необходимо наследовать класс DataRow, то придется наследовать и его родительский класс DataTable.

В приведенном ниже примере разрабатывается логика для таблицы Invoice, предусматривающая проверку кредита перед добавлением в таблицу нового счета. В итоге наша логика должна выглядеть так, как показано в листинге 6.7.

Листинг 6.7. Метод *AddInvoiceRow*

```
public void AddInvoiceRow(InheritedInvoiceRow row)
{
  if (DoesCustomerHaveCredit())
  {
    base.AddInvoiceRow(row);
  }
  else
  {
    throw new Exception(
                  "Customer Invoice cannot be created, " +
                  "no credit available");
  }
}
```

Для того чтобы иметь возможность определить подобную логику, необходимо начать с наследования типизированного класса DataTable, как показано в листинге 6.8.

```
  protected CustomersObject(SerializationInfo info,
                            StreamingContext context) :
    base(info, context)
  {
    Register();
  }

  private void Register()
  {
    Invoice.InvoiceRowChanging +=
          new InvoiceRowChangeEventHandler(InvoiceChanging);
  }

  private void InvoiceChanging(object source,
                               InvoiceRowChangeEvent args)
  {
    if (args.Action == DataRowAction.Add ||
        args.Action == DataRowAction.Change)
    {
      if (args.Row.InvoiceDate > DateTime.Today)
      {
        throw new Exception("Cannot Create Invoices" +
                            "in the Future");
      }
    }
  }
}
```

В приведенном выше листинге создаются новый класс-потомок типизированного класса DataSet (CustomerTDS) и два его конструктора, Первый конструктор предназначен для обычного создания экземпляра класса, а второй — для десериализации экземпляра класса из документа XML. Поддержку XML-сериализации обеспечивает реализация конструктора десериализации. Кроме вызова конструктора базового класса, все, что необходимо при этом сделать, — это вызвать метод Register(), позволяющий зарегистрироваться для получения уведомлений об интересующих нас событиях. В данном случае мы регистрируемся для получения уведомления о событии InvoiceRowChangingEvent, которое срабатывает перед фактическим изменением строки. В методе-обработчике (InvoiceChanging) производится проверка того, была строка изменена или добавлена в таблицу. В каждом из этих случаев необходимо убедиться в том, что дата счета не указывает на будущее, иначе будет сгенерировано исключение. Использование нового типизированного класса DataSet показано в листинге 6.6.

Листинг 6.6. Проверка бизнес-логики в управляемом событиями типизированном классе DataSet

```
...

// Создание объекта DataAdapter для каждой таблицы,
// извлекаемой из базы данных.
SqlDataAdapter daCustomers =
       new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

// Создание объекта DataAdapter для таблицы Invoice.
SqlDataAdapter daInvoices =
       new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание экземпляра класса, порожденного непосредственно
```

типизированный объект DataSet с несколькими таблицами и отношениями, вы тем самым автоматически определяете реляционно-иерархическое отображение. Приняв единственную строку единственной таблицы за вершину объектного графа, вы можете перемещаться по отношениям для получения дерева элементов. Это очень похоже на создание уровней бизнес-объектов с помощью слияний нескольких таблиц для получения объектного графа базы данныгх. Разница заключается в том, что в объекте DataSet и в типизированном объекте DataSet перемещаться по графу можно только по направлению к связанным строкам связанных таблиц. Таким образом, типизированный класс DataSet избавляет программиста от необходимости создавать реляционно-объектное отображение, потому что оно является неотъемлемой частью самого класса DataSet.

Следующая задача программиста заключается в создании кода для манипулирования базой данных. Так как в ADO.NET класс DataSet абстрагирован от кода управления базой данных (обычно подобный код сконцентрирован в управляемом поставщике), задача программиста существенно упрощается. Несмотря на то что вам все еше может понадобиться создавать хранимые процедуры для операций удаления, создания, обновления и чтения (Create, Read, Update, Delete — CRUD), связывая их с объектом **DataAdapter** (см. главу 8, "Обновление базы данных"), в целом всю основную работу по взаимодействию с базой данных берет на себя ADO.NET.

Таким образом, нам остается разработать только бизнес-логику, что, на мой взгляд, является самой простой частью всего задания. В некоторых системах бизнес-логика — это просто проверка данных на достоверность, в то время как в других системах сложность бизнес-логики сравнима со сложностью интеграции и синхронизации нескольких различных систем. На самом деле бизнес-логика — это любая логика, которая должна быть применена к информации, хранящейся в базе данных. При этом возникает вполне законный вопрос: если мы используем типизированный класс DataSet для того, чтобы избавиться от необходимости создавать уровень бизнес-логики, куда мы ее должны поместить? На мой взгляд, наиболее подходящее место для бизнес-логики — это класс, порожденный непосредственно от типизированного класса DataSet.

Разработка бизнес-логики

Код, сгенерированный на основе XSD-файла, является обычным классом, который можно наследовать. В этом разделе мы рассмотрим два подхода к определению бизнес-логики, каждый из которых является оптимальным решением в определенных условиях. Следует отметить, что какой бы подход мы ни использовали, все начинается с создания класса-потомка типизированного класса DataSet.

Управляемый событиями класс DataSet

При небольшом объеме бизнес-логики можно создать класс-потомок типизированного класса DataSet и зарегистрировать его для получения уведомлений о событиях. Из всего множества различных событий класса DataSet особый интерес представляют события `RowChanging` и RowChanged (листинг 6.5).

Листинг 6.5. Получение уведомлений о событиях из типизированного класса *DataSet*

```
public class CustomersObject : CustomerTDS
{
  public CustomersObject() : base()
  {
    Register();
  }
```

```
        new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание типизированного объекта DataSet.
CustomerTDS typedDS = new CustomerIDS() ;

// Заполнение объекта DataSet с помощью объектов DataAdapter.
customerAdapter.Fill(typedDS, "Customer");
invoiceAdapter.Fill(typedDS, "Invoice");

// Вывод адреса и количества счетов для каждого клиента.
foreach(CustomerTDS.CustomerRow cuatRow in typedDS.Customer)
{
  Console.WriteLine(custRow.FullName);
  Console.WriteLine(custRow.Address);
  Console.WriteLine("{0}, {1}  {2}",
                    custRow.City,
                    custRow.State,
                    custRow.Zip);
  Console.WriteLine("{0} (hm)",
                    custRow.IsHomePhoneNull() ?"" :
                    custRow.HomePhone);
  Console.WriteLine("{0}  (bus)",
                    custRow.IsBusinessPhoneNull() ? "" :
                    custRow.BusinessPhone);
  Console.WriteLine("Invoices: " +
           custRow.GetChildRows("CustomerInvoice").Length);
  Console.WriteLine("");
}
```

После заполнения типизированного объекта DataSet с ним можно обращаться точно так же, как и с обычным объектом DataSet (потому что он является экземпляром класса, непосредственно порожденного от класса DataSet). Как только объект будет заполнен данными, мы можем использовать типизированные методы доступа к строкам и столбцам, причем это будет делаться более коротким путем, чем в случае использования обычного (не типизированного) класса DataSet.

Поскольку столбцы HomePhone и BusinessPhone могут хранить null, перед получением доступа к ним необходимо осуществить проверку на наличие этого значения, В противном случае будет сгенерировано исключение TypedDataSetException. Избежать такого поведения можно только посредством внесения специальных аннотаций для изменения генерируемого кода.

Упрощение уровня бизнес-объектов

В течение своей профессиональной карьеры я занимался разработкой уровней бизнес-объектов дольше, чем любой другой работой. Создание уровня бизнес-объектов обычно включает в себя несколько задач:

- отображение реляционной модели на иерархическую модель;
- чтение и запись объектов в базу данных;
- добавление бизнес-логики для удовлетворения требований к данным.

Самыми сложными аспектами всей работы являются отображение реляционной модели на иерархическую модель, а также написание кода для манипулирования базой данных, К счастью, использование типизированных классов DataSet позволяет избавиться от необходимости самостоятельно разрабатывать этот код. Создав

```
    </xs:keyref>
...
  </xs:element>
</xs:schema>
```

Проанализируем XSD-файл, попытавшись разобраться в том, куда и зачем были вставлены аннотации.

1. Первое, что необходимо сделать, — это разместить ссылку на пространство имен в заголовке схемы. Таким образом мы сообщаем об определении пространства **имен** и разрешаем использование атрибутов.
2. После этого мы переходим к элементу Customer. Атрибут typedName используется для изменения **имени** типизированного объекта **DataRow** в объекте **DataTable** на OurCustomer. Другими словами, при запросе отдельной строки объекта **DataTable** вместо экземпляра класса Customer будет возвращен экземпляр класса OurCustomer. Аналогично, атрибут typedPlural используется для изменения имени свойства объекта **DataSet,** позволяющего получить доступ к конкретному объекту **DataTable.** Другими словами, при необходимости получить таблицу Customer нужно обращаться к свойству OurCustomers.
3. **Далее** мы добавили атрибут codegen:nullValue="_empty" к элементу MiddleName. Это указывает на необходимость интерпретации значения DBNull в элементе MiddleName как пустой строки в типизированном объекте **DataSet.**
4. Изменения, внесенные в элемент Invoice, совпадают с изменениями, внесенными в элемент Customer. Так, типизированный класс DataRow был переименован в TheirInvoice, а свойство, которое возвращает экземпляр этого класса, — в TheirInvoices.
5. Наконец, мы изменили **имена** каждой из сторон нашего отношения с помощью дескриптора keyref и атрибутов typedParent и typedChildren. Теперь класс Consumer будет иметь метод invoices (вместо GetInvoiceTable), позволяющий перемещаться вниз по отношению, а класс Invoice — метод Customer (вместо GetCustomerTable), позволяющий перемещаться вверх по отношению.

Использование типизированного класса DataSet

В основном типизированный класс DataSet можно использовать везде, где можно использовать обычный класс DataSet. Поскольку типизированный класс DataSet непосредственно наследует класс DataSet, все связанные с ним методы и классы будут работать как ни в чем не бывало. Одним из самых существенных преимуществ использования типизированного класса DataSet является сильная типизация и строгое именование элементов объекта DataSet. Использование нового класса несколько отличается от использования класса DataSet (листинг 6.4).

Листинг 6.4. Использование типизированного класса *DataSet*

```
...

// Создание объекта DataAdapter для таблицы Customer,
SqlDataAdapter customerAdapter =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Создание объекта DataAdapter для таблицы Invoice.
SqlDataAdapter invoiceAdapter =
```

- **_throw. При попытке доступа к значению** DBNull **генерируется исключение** (codegen: nullValue=_throw). **Если не указано обратное, это поведение принято по умолчанию.**
- **_null. При попытке доступа к значению DBNull** возвращается **указатель null. Если приложение ожидает получение значения структурного типа (value type), генерируется исключение.**
- **_empty. При попытке доступа к значению DBNull** возвращается **ссылка, созданная конструктором пустого объекта (к примеру,** String.Empty **для строк). Для структурных типов генерируется исключение.**

Эти аннотации должны быть добавлены непосредственно в файл .XSD. **Если вы используете Visual Studio** .NET, **то для внесения изменений в файл** .XSD **перейдите в режим XML-представления. Ниже приведен пример** аннотаций **в файле** .XSD.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs: schema id="AnnotatedTDS"
      targetNamespace="http://tempuri.org/AnnotatedTDS"
      elementFormDefault="qualified"
      attributeFormDefault="qualified"
      xmlns="http://tempuri.org/AnnotatedTDS"
      xmlns:mstns="http://tempuri.org/AnnotatedTDS"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
      xmlns:codegen="urn:schemas-microsoft-com:xml-msprop" >
  <xs:element name="AnnotatedTDS" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Customer"
                    codegen:typedName="OurCustomer"
                    codegen:typedPlural="OurCustomers">
...
              <xs:element name="MiddleName" type="xs:string"
                          minOccurs="0"
                          codegen:nullValue="_empty" />
...
        </xs:element>
        <xs:element name="Invoice"
                    codegen:typedName="TheirInvoice"
                    codegen:typedPlural="TheirInvoices">
...
              <xs:element name="Terms" type="xs:string"
                          minOccurs="0"
                          codegen:nullValue="Net 30" />
...
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:keyref name="CustomerInvoice"
               refer="ADONETKey1"
               msdata:AcceptRejectRule="Cascade"
               msdata:DeleteRule="Cascade"
               msdata:UpdateRule="Cascade"
               codegen:typedChildren="Invoices"
               codegen:typedParent="Customer">
      <xs:selector xpath=".//mstns:Invoice" />
      <xs:field xpath="mstns:CustomerID" />
```

В результате выполнения этого фрагмента кода будет создан файл CustomerDS.xsd, который можно передать на вход программе XSD.exe. Для того чтобы создать типизированный класс DataSet, перейдите в командную строку и воспользуйтесь инструментом XSD.exe так, как показано на рис. 6.17.

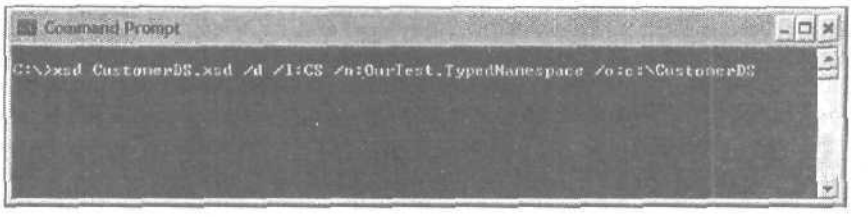


Рис. 6.17. Запуск программы XSD.exe из командной строки

Параметрами, вызывающими интерес, являются следующие.

- /d — указывает на необходимость создания класса, производного от класса DataSet. Этот параметр указывается каждый раз при создании типизированного класса DataSet.
- /l — определяет язык программирования, использующийся при генерировании классов. CS соответствует C#, VB — VB.NET, а JS — Jscript .NET.
- /n — определяет пространство имен, в котором должны находиться генерируемые классы. Этот параметр является необязательным.
- /o — определяет каталог, в который необходимо поместить результирующие файлы. Если вы используете этот параметр, то учтите, что каталог, в котором будут размещаться выходные файлы, должен существовать; инструмент XSD.exe не сможет создать каталог за вас. Если выходной каталог не указан, классы будут созданы в текущем каталоге.

После выполнения программы XSD.exe вы получаете файл класса, который необходимо включить в проект.

Настройка сгенерированного кода с помощью специальных аннотаций

ADO.NET позволяет управлять некоторыми аспектами сгенерированного кода с помощью добавления в XSD-файл, использующийся для генерации типизированного класса DataSet, специальных аннотаций. Для этого в XSD-файл нужно добавить несколько атрибутов, имеющих префикс codegen:

- typedName — определяет имя объекта;
- typedPlural — определяет имя коллекции объектов;
- typedParent — определяет имя родительского отношения;
- typedChildren — определяет имя дочернего отношения;
- nullValue — указывает способ обработки значения DBNull для заданного столбца; этот атрибут имеет несколько возможных значений.
  - Значение по умолчанию. Значение, которое будет возвращаться вместо DBNull по умолчанию (например, пустая строка — codegen :nullValue="").

Использование программы XSD.exe для создания типизированного класса DataSet

С помощью инструмента командной строки XSD.exe типизированный класс DataSet можно создать непосредственно из файла .XSD (XML Schema Definition — определение схемы XML). Программа XSD.exe входит в состав набора инструментальных средств разработки (SDK) .NET Framework. Процесс создания типизированного класса DataSet с помощью программы XSD.exe по своей сути практически ничем не отличается от создания типизированного класса DataSet в Visual Studio .NET. Если у вас нет Visual Studio .NET или вы отдаете предпочтение инструментам командной строки, то программа XSD.exe подходит как нельзя лучше. Если же вы используете Visual Studio .NET, то реализованная в ней поддержка типизированных классов DataSet поможет вам создать те же самые классы, затратив на это намного меньше усилий.

Программа XSD.exe используется для генерации исходного кода на основе файлов .XSD — как классов, наследующих классы .NET XML, так и классов, наследующих семейство классов DataSet. Перед тем как начать создание типизированного класса DataSet, нам необходимо получить документ XSD, для создания которого можно воспользоваться объектом DataSet, как показано в листинге 6.3.

Листинг 6.3. Создание XSD-документа на основе объекта *DataSet*

```
...

// Создание объекта DataAdapter для каждой таблицы,
// извлекаемой из базы данных.
SqlDataAdapter daCustomers =
        new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");

// Создание переменной для упрощения
// доступа к таблице Customers.
DataTable customersTable = dataSet.Tables["Customers"];

// Уточнение схемы.
customersTable.Columns["CustomerID"].ReadOnly - true;
customersTable.Columns["CustomerID"].Unique = true;
customersTable.Columns["LastName"].MaxLength = 50;
customersTable.Columns["LastName"].AllowDBNull =false;
customersTable.Columns["FirstName"].MaxLength = 50;
customersTable.Columns["FirstName"].AllowDBNull = false;
customersTable.Columns["MiddleName"].MaxLength = 50;
customersTable.Columns["State"].DefaultValue = "MA";
customersTable.Columns["State"].MaxLength = 2;
customersTable.Columns["HomePhone"].Unique = true;

// Сохранение XSD-файла, который будет использоваться
// для создания типизированного класса DataSet.
dataSet.WriteXmlSchema(@"c:\CustomerDS.xsd");
```

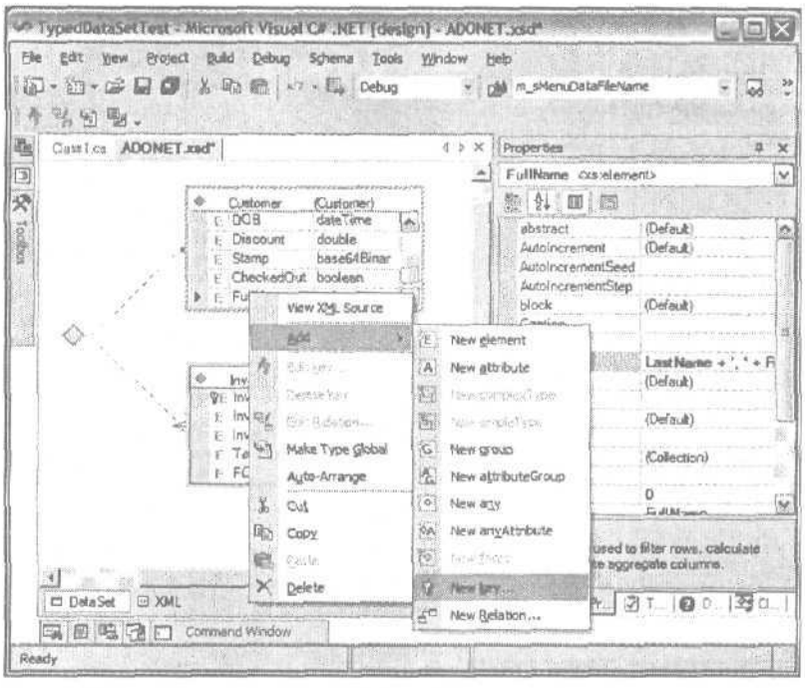


Рис. 6.14. Выполнение команды Add⇨New key...

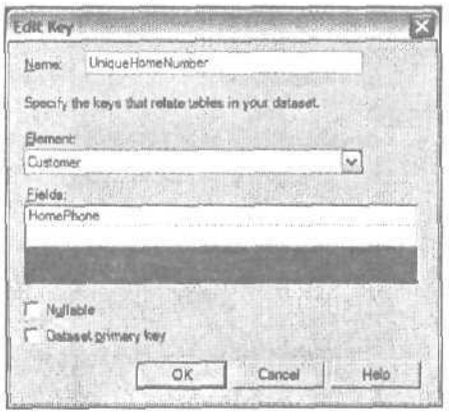


Рис. 6.15. Определение имени ключа

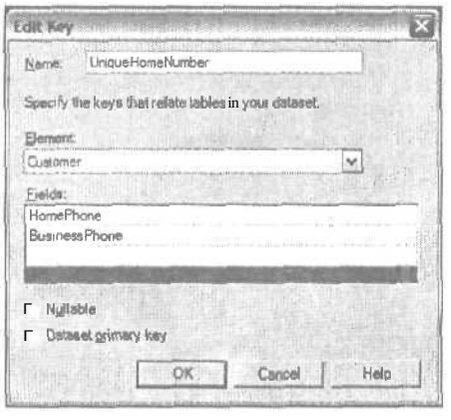


Рис. 6.16. Определение составного ключа

Если на данном этапе попросту ввести имя и щелкнуть на кнопке OK, то будет создано ограничение уникальности. Однако возможны случаи, в которых необходимо определить ключ, состоящий из нескольких столбцов. Для того чтобы это сделать, необходимо добавить входящие в ключ поля (рис. 6.16).

Созданное ограничение заставит объект DataSet проверять уникальность всех комбинаций значений полей `HomePhone` и `BusinessPhone`. Теперь вас можно поздравить — типизированный класс DataSet полностью определен!

Перед тем как просмотреть код, сгенерированный Visual Studio .NET, необходимо осуществить сборку проекта. После этого вновь созданный типизированный класс `DataSet` будет доступен для использования.

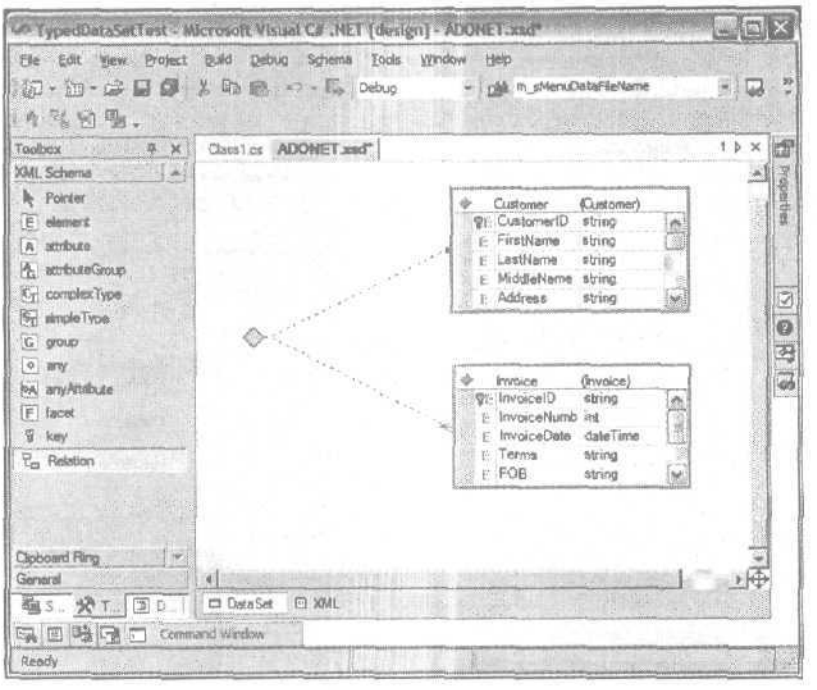


Рис. 6.12. Таблицы с созданным отношением

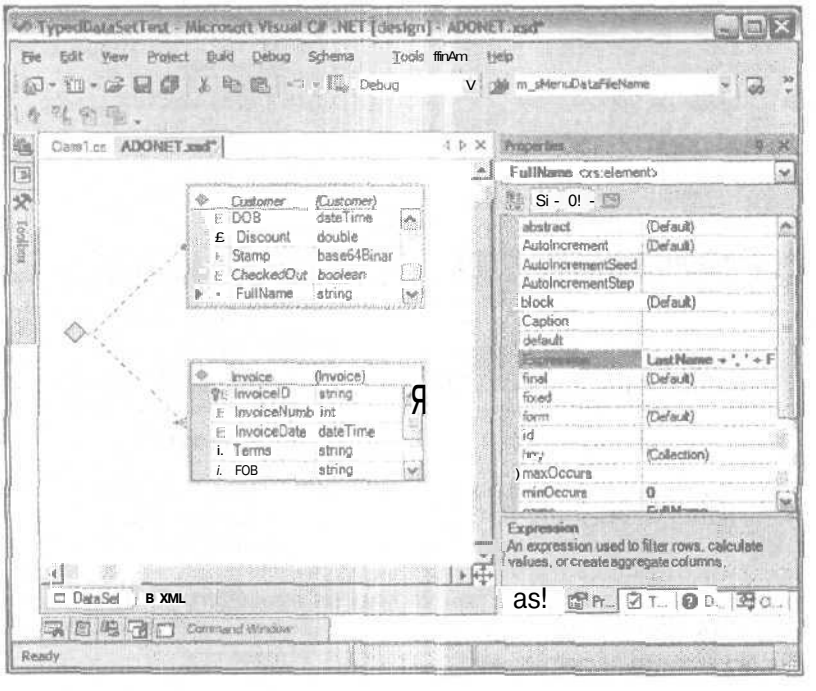


Рис. 6.13. Добавление вычисляемого столбца

В диалоговом окне Edit Relation определяются особенности отношения. Из раскрывающегося списка Child element (Дочерний элемент) выберите дочернюю таблицу (в данном случае таблицу Invoice). Обычно Visual Studio .NET корректно определяет поля внешнего ключа. При необходимости вы можете изменить правила каскадного обновления объекта DataSet с помощью раскрывающихся списков Update rule, Delete rule и Accept/Reject rule. После внесения всех изменений диалоговое окно Edit Relation должно выглядеть примерно так, как показано на рис. 6.11.

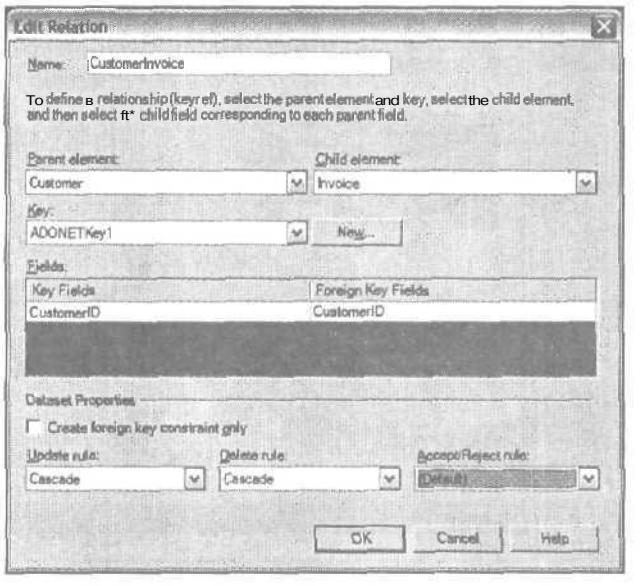


Рис. 6.11. Диалоговое ***окно Edit Relation после*** *внесения* ***всех*** *изменений*

После щелчка на кнопке OK вы увидите вновь созданное отношение (рис. 6.12).

Добавив две таблицы и отношение между ними, добавим к таблице `Customer` вычисляемый столбец. Для того чтобы это сделать, перейдем в конец таблицы Customer и добавим новый столбец `FullName`. В свойствах этого столбца определим его выражение:

```
LastName + ', ' + FirstName
```

В результате все должно выглядеть так, как показано на рис. 6.13.

Наконец, добавим ограничение уникальности для столбца `HomePhone`, чтобы быть уверенными в том, что домашние телефоны наших клиентов не повторяются. В контексте типизированного класса DataSet это будет означать добавление ключа. Выберите столбец, щелкните на нем правой кнопкой мыши и выполните команду Add⇨New key... (Добавить⇨Новый ключ), как показано на рис. 6.14. На экране появится диалоговое окно Edit Key (Редактирование ключа), показанное на рис. 6.15.

Начнем с добавления отношения между **таблицами** Customer и invoice. **Перетащите новый элемент** Relation из панели **Toolbox в** таблицу `Customer` **(рис. 6.9). На экране появится диалоговое окно** Edit **Relation (Редактирование отношения), как показано на рис. 6.10.**

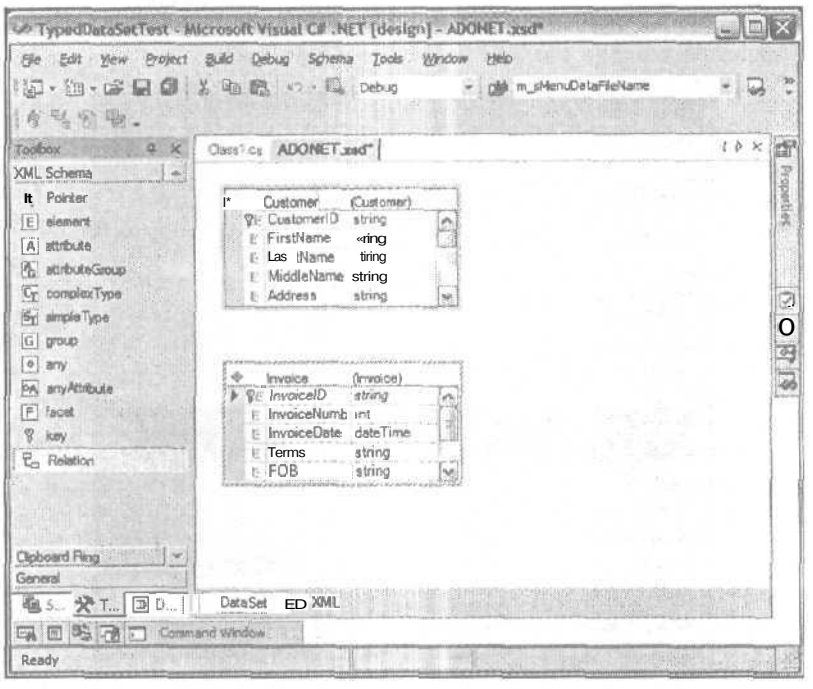


Рис. ***6.9.*** *Добавление* ***отношения***

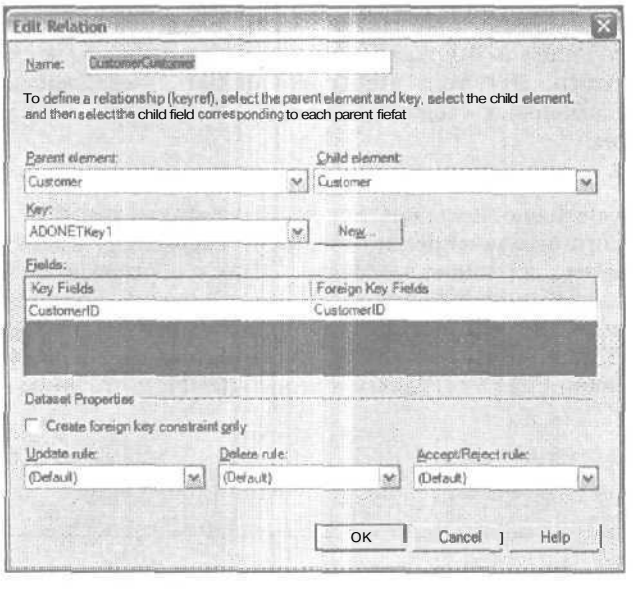


Рис. 6. /ft *Диалоговое* ***окно Edit*** *Relation*

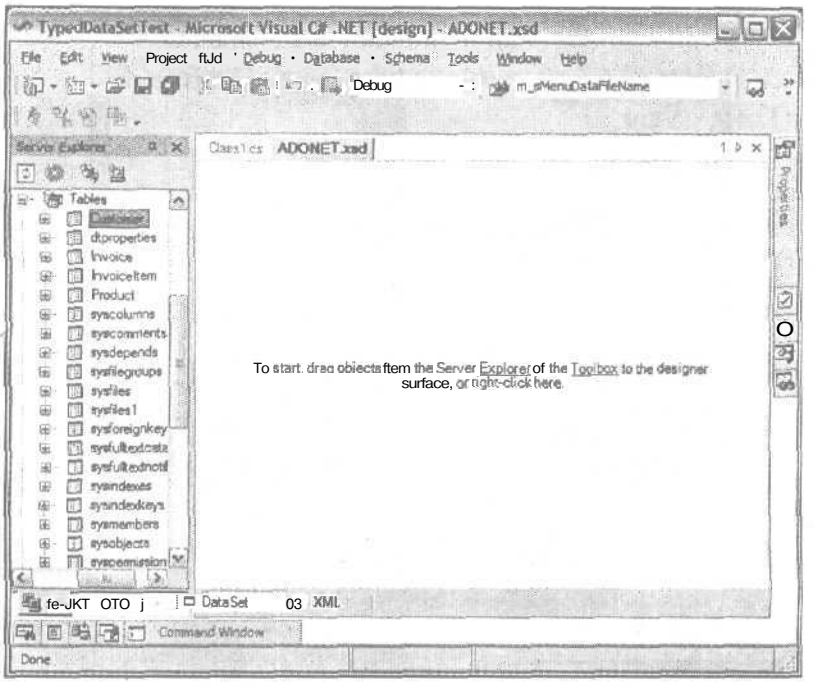


Рис. 6.7. Окно Server Explorer

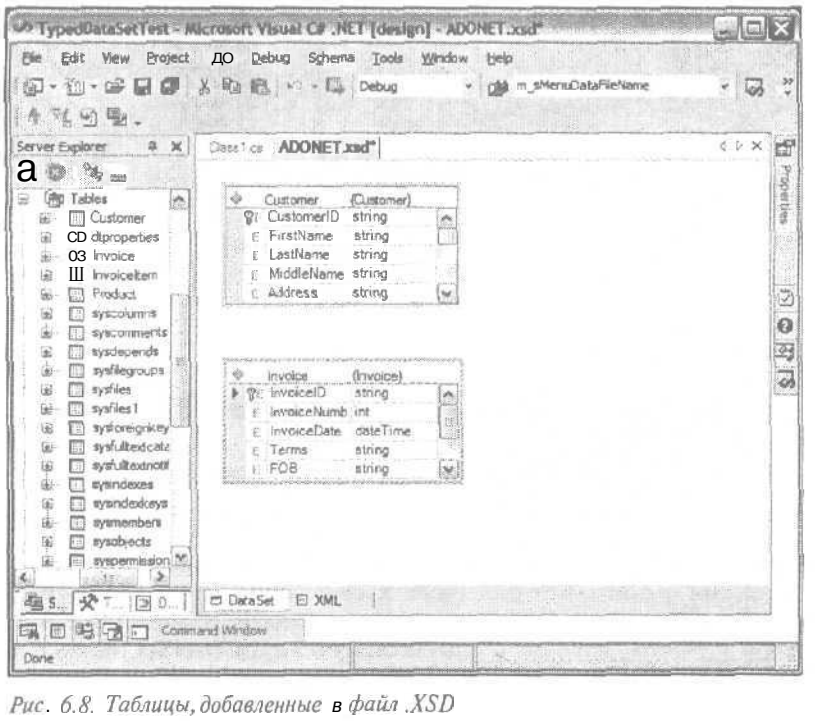


Рис. 6.8. Таблицы, добавленные в файл .XSD

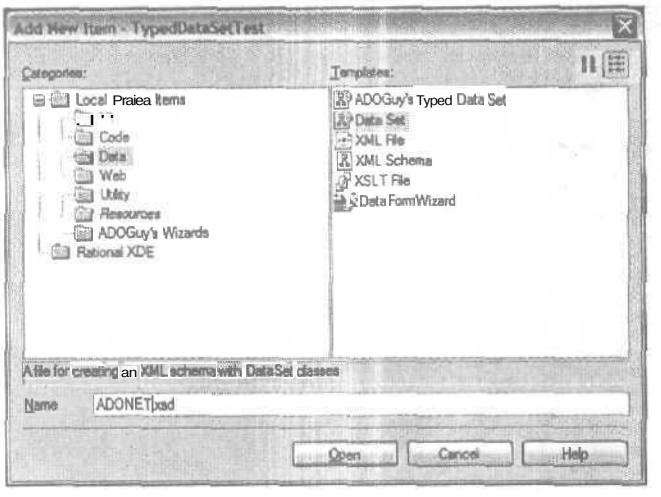


Рис. 6.5. Добавление класса DataSet

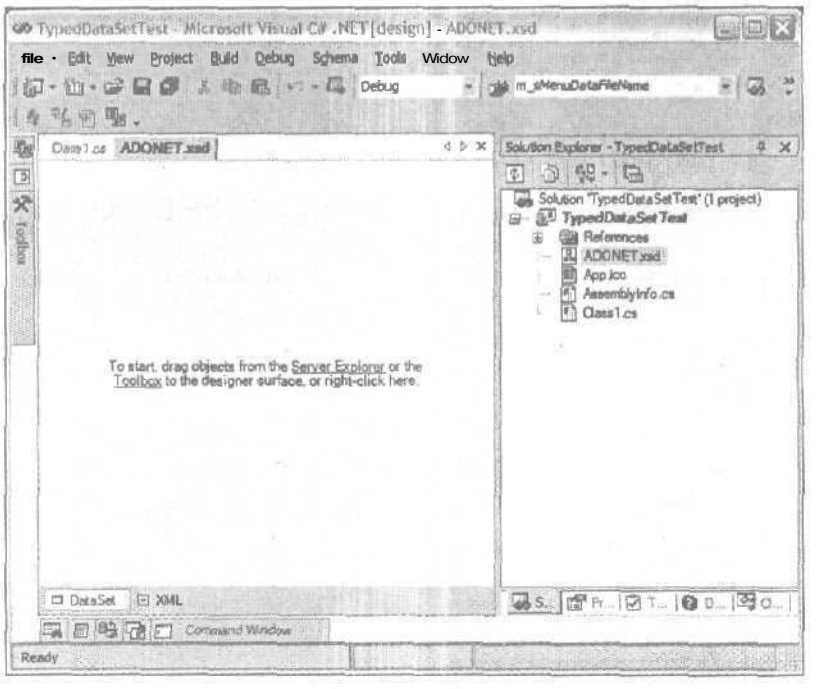


Рис. 6.6. Пустая основа класса DataSet

На этом этапе новый типизированный класс DataSet включает в себя две таблицы, но между ними отсутствует отношение. К сожалению, Visual Studio .NET даже не пытается получить полную информацию о схеме, которая содержится в базе данныгх. Для того чтобы уточнить информационную схему объекта **DataSet,** ее необходимо добавить в файл .XSD вручную.

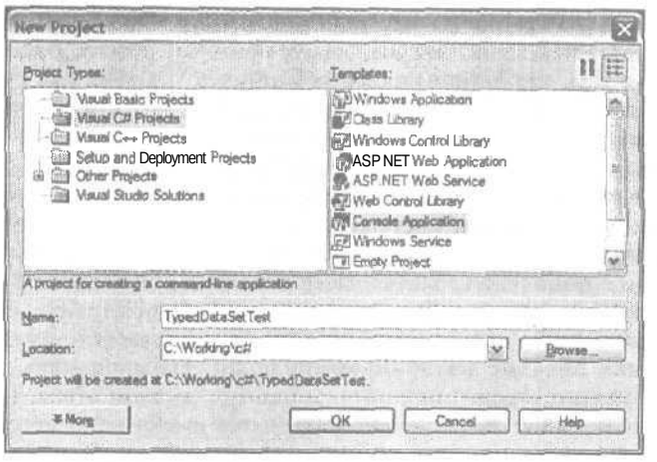


Рис. 6.3. Создание нового проекта C#

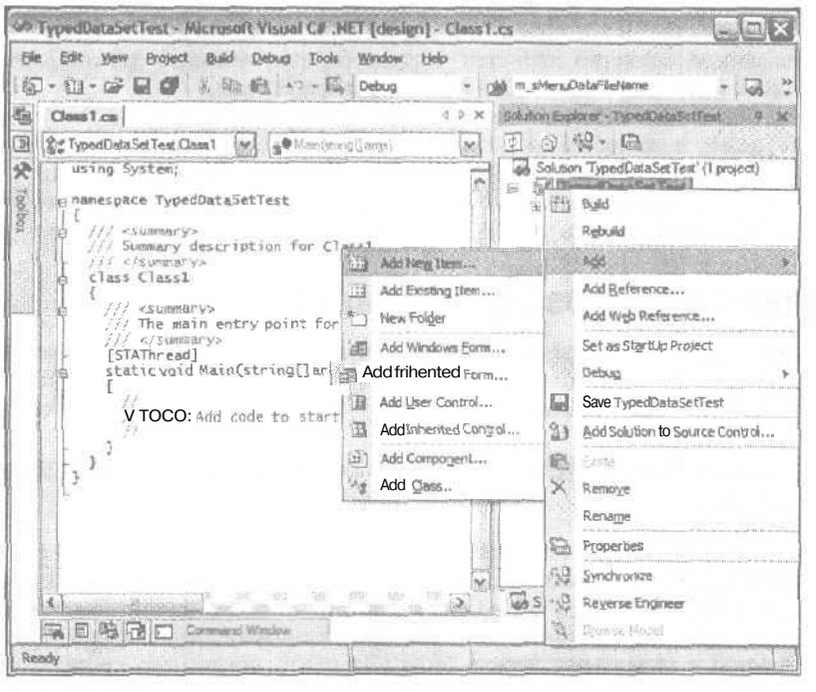


Рис. 6.4. Добавление нового элемента

После добавления к проекту типизированного класса DataSet вы можете с помощью диалогового окна Server Explorer (Обозреватель серверов) добавить к нему таблицы, а с помощью панели Toolbox (Панель инструментов) — элементы схемы. Использование диалогового окна Server Explorer показано на рис. 6.7.

После того как в окне Server Explorer будет выбрана существующая база данных, перетащите все необходимые таблицы в файл .XSD. В итоге у вас должно получится нечто, подобное тому, что показано на рис. 6.8.

Следует обратить внимание на несколько моментов. Во-первых, типизированный объект DataSet создается точно так же, как и обычный объект **DataSet.** Разница заключается лишь в том, что в типизированном объекте **DataSet** уже существует схема. Во-вторых, хотя это и типизированный объект **DataSet,** класс CustomerTDS непосредственно наследует класс DataSet. Таким образом, типизированный объект **DataSet** можно заполнять с помощью объектов DataAdapter. На самом деле типизированный класс DataSet является просто специализированным классом DataSet. Вы также, вероятно, заметили, что синтаксис доступа к полям и таблицам при использовании типизированного класса DataSet существенно упрощается. Теперь обращение к каждой таблице возможно с помощью свойств класса CustomerTDS. Каждое поле, в **свою** очередь, является свойством строки. Следует отметить, что кроме упрощения синтаксиса, вы получаете возможность проверять правильность написания имен элементов типизированного класса DataSet на этапе компиляции. Наконец, при попытке указать номер счета с помощью строки мы получим ошибку на этом этапе, поскольку сгенерированный класс DataSet знает, что номера счетов имеют целочисленный тип.

Типизированный класс DataSet наряду с обычными столбцами может содержать вычисляемые столбцы, что также позволяет обеспечить безопасность типов. В предыдущем примере мы можем обратиться к свойству FullName таблицы Customer, где FullName — вычисляемый столбец, в котором объединяются имя и фамилия клиента. Так как указанное свойство является частью типизированного класса DataSet, результат вычисления выражения возвращается в виде строки.

Наконец, как будет показано далее в этой главе, использование типизированных классов DataSet в качестве основы для уровней объектов данных или бизнес-объектов является чрезвычайно мощным инструментом. Непосредственное наследование от типизированного класса DataSet позволяет избавиться от необходимости самостоятельно разрабатывать эти уровни и обеспечить безопасность типов.

Создание типизированного класса DataSet

В данном разделе мы попытаемся создать свой собственный типизированный класс DataSet. Это можно сделать двумя способами: с помощью инструмента командной строки XSD.exe или посредством Visual Studio .NET. Начнем со второго способа.

Использование Visual Studio .NET для создания типизированного класса DataSet

В этом разделе описывается создание типизированного класса DataSet с помощью средств среды разработки Visual Studio .NET. Для начала создайте новый проект консольного приложения C# (рис. 6.3). После этого с помощью диалогового окна Solution Explorer (Обозреватель решения) добавьте к проекту новый элемент (рис. 6.4).

Из вложенной в папку Local Project Items (Элементы локального проекта) папки Data (Данные) выберите элемент **DataSet** и назовите его ADONET.xsd (рис. 6.5). Если на этом этапе добавить XML-схему (которая также является файлом .XSD), то типизированный класс DataSet создан не будет. Итак, нам нужно добавить именно класс DataSet. После этого окно Visual Studio .NET будет выглядеть примерно так, как показано на рис. 6.6.

Легко заметить, что типизированный класс DataSet имеет расширение .xsd. Это связано с тем, что его исходный код представляет собой XML-схему. Другими словами, все, что требует от пользователя Visual Studio .NET, — это создать новый файл .xsd, содержащий схему объекта **DataSet.** Более подробно информационная схема объекта **DataSet** рассматривается в главе 5, "Создание объекта DataSet". Файл .XSD может включать в себя имена таблиц и столбцов, а также ключи, отношения и ограничения.

```
// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");

// Вывод имени клиента на консоль.
Console.WriteLine(dataSet.Tables["Customers"].
                        Rows[0]["FirstName"].ToString());
Console.WriteLine(dataSet.Tables["Customers"].
                        Rows[0]["LastName"].ToString());
Console.WriteLine(dataSet.Tables["Customers"].
                        Rows[0]["HomePhone"].ToString());

// Попытка занести в поле номера счета строку.
// Этот код не будет выполняться, так как
// поле InvoiceNumber может хранить только
// целочисленные значения.
dataSet.Tables["Invoices"].
                        Rows[0]["InvoiceNumber"] = "15234";
```

Для доступа ко всем уровням иерархии объекта **DataSet** (вплоть до строк) необходимо использовать индексаторы. В любой момент можно неправильно написать какое-то имя, в результате чего при выполнении кода будет выдано сообщение об ошибке. Кроме того, в последней строке примера продемонстрирована попытка занести в поле номера счета строковое значение. Поскольку объект **DataSet** знает, что этот столбец может содержать только целочисленные значения, на этапе выполнения будет выдано сообщение об ошибке. В листинге 6.2 показано выполнение тех же самых операций с использованием типизированного класса DataSet,

Листинг 6.2. Использование типизированного класса DataSet

```
// Создание объекта DataAdapter для каждой таблицы,
// извлекаемой из базы данных.
SqlDataAdapter daCustomers =
        new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

// Создание объекта DataAdapter для таблицы Invoice.
SqlDataAdapter daInvoices =
        new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание пустого объекта DataSet.
CustomerTDS dataSet = new CustomerTDS();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");

// Вывод имени клиента на консоль.
Console.WriteLine(dataSet.Customer[0].FirstName);
Console.WriteLine(dataSet.Customer[0].LastName);
Console.WriteLine(dataSet.Customer[0].HomePhone);
Console.WriteLine(DataSet.Customer[0].FullName);

// Попытка занести в поле номера счета строку.
// Этот код не будет компилироваться, так как
// поле InvoiceNumber может хранить только
// целочисленные значения.
dataSet.Invoice[0].InvoiceNumber = "12345";
```

Типизированный класс DataSet, в свою Очередь, наследует эти классы (рис. 6.2).

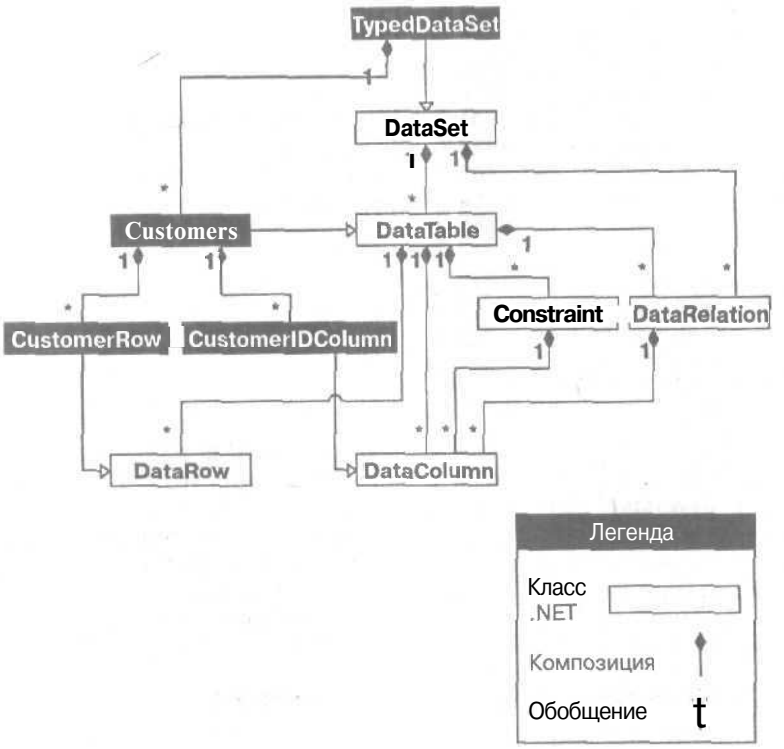


***Рис. 6.2. Структура** типизированного класса DataSet*

Откуда же типизированный класс DataSet получил свое название? Как рассказывалось в главе 5, для того чтобы указать тип данных, который будет храниться в столбцах таблицы **DataTable**, нужно создать объект **DataColumn** для каждого столбца. Это гарантирует проверку типов на этапе выполнения, но было бы неплохо быть уверенным в безопасности типов объекта DataSet еще на этапе написания кода. Типизированный класс DataSet предоставляет именно такую функциональность. В листинге 6.1 приведен код, в котором используется обычный (не типизированный) класс DataSet.

Листинг 6.1. Использование класса DataSet

```
...

// Создание объекта DataAdapter для каждой таблицы,
// извлекаемой из базы данных.
SqlDataAdapter daCustomers =
        new SqlDataAdapter("SELECT * FROM CUSTOMER;", conn);

// Создание объекта DataAdapter для таблицы Invoice.
SqlDataAdapter daInvoices =
        new SqlDataAdapter("SELECT * FROM INVOICE", conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();
```

Глава 6

Типизированные классы DataSet

В этой главе...

Что такое типизированный класс DataSet?

Создание типизированного класса DataSet

Использование типизированного класса DataSet

Упрощение уровня бизнес-объектов

В главе 5 мы рассмотрели использование объекта **DataSet** в качестве базы данных, хранимой в памяти. Следует отметить также, что читателю было предложено написать большое количество кода. В этой главе **мы** познакомимся с типизированными объектами **DataSet,** позволяющими облегчить участь программиста и обеспечить безопасность типов на этапе компиляции.

Что такое типизированный класс DataSet?

Типизированный класс `DataSet` отличается от всего, что было рассмотрено ранее в этой книге. Он не является классом .NET Framework, а представляет собой набор классов, порожденных непосредственно от классов семейства DataSet. На рис. 6.1 показана диаграмма классов из главы 5, демонстрирующая отношение между элементами класса DataSet.

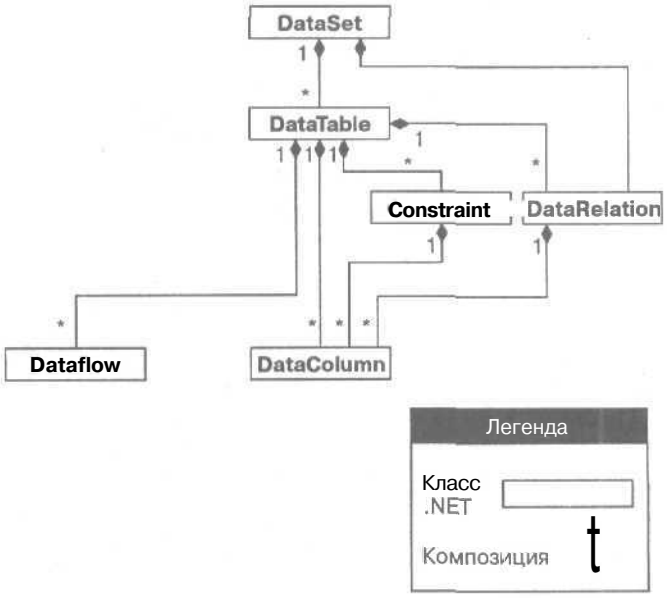


Рис. 6.1. Структура класса DataSet

Резюме

В этой главе были показаны все многочисленные возможности объекта DataSet, используемого в качестве базы данных, хранящейся в памяти. Возможность определения схемы объекта DataSet позволяет увеличить производительность механизма проверки данных на стороне клиента (или на среднем уровне). Несмотря на то что база данных может и сама обеспечить целостность хранящейся в ней информации, для этого требуется, чтобы последняя была сначала занесена в базу данных. В то же время схема объекта DataSet позволяет выявить все проблемы, связанные с целостностью информации, без необходимости использования дорогих сетевых ресурсов.

```
                InvoiceItem.ProductID = Product.ProductID";

SqlDataAdapter daItems = new SqlDataAdapter(sql, conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
daItems.Fill(dataSet, "Items");

// Создание вычисляемого столбца.
DataColumn exColumn = new DataColumn("LineTotal");
exColumn.DataType = typeof(float);
exColumn.Expression =
          "((Price - (Price * Discount)) * Quantity)";

// Добавление вычисляемого столбца в таблицу.
dataSet.Tables["Items"].Columns.Add(exColumn);
```

Синтаксис вычисляемых столбцов отличается от синтаксиса, используемого в языках C# и VB.NET. В дополнение к математическим расчетам (см. приведенный выше листинг) синтаксис вычисляемых столбцов поддерживает агрегацию и конкатенацию:

```
// Конкатенация.
exColumn.Expression = "LastName + ', ' + FirstName";

// Агрегация.
exColumn.Expression = "Avg(Price)";
```

Следует также отметить возможность агрегирования информации вдоль отношений. Для того чтобы обратиться к родительской таблице, нужно воспользоваться префиксом Parent:

```
// Обращение к родительской таблице.
exColumn.Expression = "Max(Parent.Cost)";
```

Для обращения к дочерней таблице служит префикс child:

```
// Обращение к дочерней таблице.
exColumn.Expression = "Avg(Child.Price)";
```

Если с таблицей связано более одной дочерней таблицы, необходимо обращаться к ней по имени:

```
// Обращение к заданной дочерней таблице.
exColumn.Expression = "Avg(Child(InvoiceItem2Product).Price)";
```

Обычно перемещение вдоль отношений осуществляется только при проведении агрегирования информации. Существуют ситуации, в которых вам может понадобиться включить связанное поле в вычисляемый столбец. Единственный способ сделать это заключается в имитации агрегации с помощью функции Max:

```
// Определение стоимости всего заказа.
exColumn.Expression = "Quantity * Max(Child.Price)";
```

Полное описание синтаксиса выражений вычисляемых столбцов можно найти в документации к свойству DataColumn.Expression[3].

---

[3] См. msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataDataColumnClassExpressionTopic.asp.

Листинг 5.24, Использование автоинкрементных столбцов

```
// Создание пустого объекта DataSet.
DataSet dataSet * new DataSet O;

// Создание объекта DataTable.
DataTable dataTable = dataSet.Tables.Add("AutoInc");

// Создание нового столбца типа int
// (столбец типа AutoIncrement всегда
// должен иметь тип int).
DataColumn autoIncColumn =
               dataTable.Columns.Add("TheKey", typeof(int));

// Определяем столбец как автоинкрементный.
autoIncColumn.AutoIncrement = true;

// Определение начального значения.
autoIncColumn.AutoIncrementSeed = 1;

// Определение приращения
// (число, на которое должно увеличиваться значение
// автоинкрементного столбца при добавлении каждой
// новой строки).
autoIncColumn.AutoIncrementStep = 1;
```

При создании столбцов типа AutoIncrement в объекте **DataColumn** должно храниться значения типа int. В противном случае при попытке установки свойства AutoIncrement в значение true будет сгенерировано исключение.

Использование вычисляемых столбцов

Для использования вычисляемых столбцов в ADO.NET предусмотрены две стандартные стратегии. Первая из них заключается в том, чтобы производить вычисления при получении данных с помощью SQL-запроса, например:

```
SELECT Description,
       Vendor,
       Price,
       Quantity,
       Discount,
       ((Price - (Price * Discount))* Quantity) as LineTotal
  FROM InvoiceItem
  JOIN Product on InvoiceItem.ProductID = Product.ProductID
```

Второй подход состоит в использовании вычисляемых столбцов. Его преимущество заключается в возможности применять расчеты к только что добавленной информации, а не только к информации, извлеченной из базы данных (листинг 5.25).

Листинг 5.25. Использование вычисляемых столбцов

```
...

// Создание объекта DataAdapter для каждой таблицы,
// извлекаемой из базы данных.
string sql = @"SELECT Description, Vendor, Price,
               Quantity, Discount
               FROM  Invoiceltem
               JOIN Product on
```

```
    form.WriteLine("  ReadOnly?:\t{0}", col.ReadOnly      );
    form.WriteLine("  Unique?:\t{0}",   col.Unique        );
    form.WriteLine("  MaxLength:\t{0}",col.MaxLength      );
    form.WriteLine("  Allow Null:\t{0}",col.AllowDBNull   );
}
```

Ниже показан результат выполнения приведенного выше фрагмента кода.

```
CustomerID
  Type:        Guid
  Default:
  ReadOnly?:  True
  Unique?:    True
  Max Length: -1
  Allow Null: True
FirstName
  Type:          String
  Default:
  ReadOnly?:  False
  Unique?:    False
  Max Length: 50
  Allow Null: False
LastName
  Type:          String
  Default:
  ReadOnly?:  False
  Unique?:    False
  Max Length: 50
  Allow Null: False
MiddleName
  Type:          String
  Default:
  ReadOnly?:  False
  Unique?:    False
  Max Length: 50
  Allow Null: True
State
  Type:          String
  Default:    MA
  ReadOnly?:  False
  Unique?:    False
  Max Length: 2
  Allow Null: True

...
```

В предыдущем примере мы продемонстрировали возможности объекта DataSet относительно проверки достоверности данных. Создание схемы в базе данных гарантирует целостность хранящейся в ней информации, в то время как с помощью "усиления" этой схемы в объекте DataSet можно обнаруживать проблемы, касающиеся данных, на раннем этапе, что позволяет избежать лишнего обращения к базе данных. Другими словами, определение схемы объекты DataSet может существенно облегчить труд программиста.

Использование автоинкрементных столбцов

Если после прочтения раздела "Стратегии, использующиеся при определении первичного ключа" вы все же решились применять автоинкрементные столбцы, обратите внимание на приведенный в листинге 5.24 код, позволяющий сделать установку значения таких столбцов практически тривиальным заданием.

```
  ReadOnly?:   False
  Unique?:     False
  Max Length: -1
  Allow Null:  True

...
```

Поскольку объект DataSet был создан непосредственно на основе запроса к базе данных, ADO.NET не имеет возможности определить его полную схему. Управляемые поставщики Microsoft определяют информационную схему объекта DataSet на основе сведений, предоставленных объектом DataReader (который также используется для заполнения объектов DataTable). Судя по полученному результату видно, что тип данных был определен правильно, однако остальная часть информационной схемы отсутствует ввиду недостатка сведений, предоставленных объектам DataSet и DataAdapter. В этом случае схему необходимо доработать, чтобы сделать объект DataSet более устойчивым. К примеру, столбец Customer ID должен быть предназначен только для чтения, так как это позволит предотвратить случайное изменение первичного ключа. Необходимо также установить максимальную длину некоторых строк так, чтобы они соответствовали схеме базы данных. В листинге 5.23 приведена расширенная версия предыдущего кода, предполагающая улучшение схемы столбцов.

Листинг 5.23. Улучшенная схема столбцов

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из Сазы данных таблицы.
SqlDataAdapter daCustomers = new
                  SqlDataAdapter("SELECT * FROM CUSTOMER;",
                  conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
daCustomers.Fill(dataSet, "Customers");

// Создание переменной, упрощающей доступ к таблице.
DataTable customersTable = dataSet.Tables["Customers"];

// Улучшение схемы столбцов.
customersTable.Columns["CustomerID"].ReadOnly = true;
customersTable.Columns["CustomerID"].Unique = true;
customersTable.Columns["LastName"].MaxLength = 50;
customersTable.Columns["LastName"].AllowDBNull = false;
customersTable.Columns["FirstName"].MaxLength = 50;
customersTable.Columns["FirstName"].AllowDBNull = false;
customersTable.Columns["MiddleName"].MaxLength = 50;
customersTable.Columns["State"].DefaultValue = "MA";
customersTable.Columns["State"].MaxLength= 2;

// Вывод информации о схеме столбцов таблицы Customers.
foreach (DataColumn col in customersTable.Columns)
{
  form.WriteLine("{0}",              col.ColumnName    );
  form.WriteLine("  Type:\t\t{0}",   col.DataType.Name );
  form.WriteLine("  Default:\t{0}",  col.DefaultValue  );
```

```
SqlDataAdapter daCustomers = new
                  SqlDataAdapter("SELECT * FROM CUSTOMER;",
                  conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet ();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
daCustomers.Fill(dataSet, "Customers");

// Создание переменной, упрощающей доступ к таблице.
DataTable customersTable = dataSet.Tables["Customers"];

// Вывод информации о схеме столбцов таблицы Customers.
foreach (DataColumn col in customersTable.Columns)
{
  Console.WriteLine("{0}",              col.ColumnName    );
  Console.WriteLine(" Mapping:\t{0}",   col.ColumnMapping );
  Console.WriteLine(" Type:\t\t{0}",    col.DataType.Name );
  Console.WriteLine(" Default:\t{0}",   col.DefaultValue  );
  Console.WriteLine(" ReadOnly?:\t{0}", col.ReadOnly      );
  Console.WriteLine(" Unique?:\t{0}",   col.Unique        );
  Console.WriteLine(" Max Length:\t{0}",col.MaxLength     );
}
```

Ниже показан результат выполнения приведенного выше фрагмента кода.

```
CustomerID
  Type:        Guid
  Default:
  ReadOnly?:   False
  Unique?:     False
  Max Length:  -1
  Allow Null:  True
FirstName
  Type:        String
  Default:
  ReadOnly?:   False
  Unique?:     False
  Max Length:  -1
  Allow Null:  True
LastName
  Type:        String
  Default:
  ReadOnly?:   False
  Unique?:     False
  Max Length:  -1
  Allow Null:  True
MiddleName
  Type:        String
  Default:
  ReadOnly?:   False
  Unique?:     False
  Max Length:  -1
  Allow Null:  True
State
  Type:        String
  Default:
```

```
{
  Console.WriteLine("After Column Changed:" +
                    " LastName={0};
                    " Column={1};
                    " original LastName={2}",
                    e.Row["LastName"],
                    e.Column.ColumnName,
                    e.Row["LastName", DataRowVersion.Original] );
}
```

В этом примере мы добавляем новые обработчики событий (DataRowChangeEventHandler и DataColumnChangeEventHandler) в объект DataTable. При срабатывании этих событий объект получает уведомление. Следует отметить, что события изменения срабатывают только при вставке и обновлении, но не при удалении информации.

Два различных события изменения служат разным целям. События DataRowChanging и DataRowChanged используются в случае, когда нам необходимо отследить факт изменения строки. В то же время события DataColumnChanging и DataColumnChanged предназначены для отслеживания факта изменения столбца в пределах строки. Аргументы, пересылаемые вместе с **этими** двумя событиями, также служат разным целям. При срабатывании события объекта **DataRow** передается вся изменившаяся строка, а при срабатывании события объекта **DataColumn** — только изменившийся столбец.

Схема столбца

Определив схему таблицы, необходимо также указать, какие данные могут в ней храниться. Объект **DataTable** состоит из строк данных. Каждая строка в свою очередь представляет собой коллекцию столбцов. Для того чтобы определить правило для хранимой в столбце информации, необходимо создать его схему. Объект **DataTable** имеет свойство Columns типа DataColumnCollection, в котором хранится набор правил для столбцов таблицы. Каждый объект **DataColumn** является одним столбом в таблице **DataTable. В** нем хранятся следующие сведения:

- DataType — тип данных, которые могут храниться в столбце;
- ColumnName — **имя столбца;**
- MaxLength — максимальный размер данных, которые могут храниться в столбце;
- DefaultValue — значение столбца по умолчанию;
- Readonly — определяет возможность внесения изменений в столбец;
- AllowNull — определяет возможность хранения в столбце значения DBNull;
- Unique — определяет, должно ли значение столбца быть уникальным в пределах объекта **DataTable.**

При создании нового объекта **DataSet** на основе информации из базы данных или XML-файла в ваше распоряжение предоставляется минимальная схема столбцов, включающая стандартный набор информации, необходимой для обеспечения функционирования объекта **DataSet. В** листинге 5.22 показан пример извлечения информации о схеме столбцов объекта **DataSet.**

Листинг 5.22. Схема столбцов объекта DataSet

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
```

Перечисление DataRowAction дает возможность точно определить, что произошло со строкой. Так как объект DataRowChangeEventArgs используется для всех событий строк, можно ожидать, что события RowChanging и RowChanged сработают и при удалении строки. На самом деле, это не так — при удалении строк срабатывают только события RowDeleting и RowDeleted.

При отслеживании изменений строки может возникнуть необходимость в точной локализации модифицированной информации. Именно для этого предназначены события ColumnChanging и ColumnChanged, которые при срабатывании предоставляют ссылку как на изменившуюся строку, так и на изменившийся столбец.

Для того чтобы определить триггер, необходимо добавить новый обработчик для требуемого события (листинг 5.21).

Листинг 5.21. События объекта *DataSet*

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
SqlDataAdapter daCustomers = new SqlDataAdapter(
                                    "SELECT * FROM CUSTOMER;",
                                    conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");

// Добавление нового обработчика события изменения строки
//  (изменения отслеживаются на уровне строки и столбцов;
// это напоминает триггер базы данных).
dataSet.Tables[0].RowChanged += new
             DataRowChangeEventHandler(RowIsChanged);
dataSet.Tables[0].ColumnChanged += new
             DataColumnChangeEventHandler(ColumnIsChanged);

// Изменение строки с целью убедиться
// в срабатывании события.
dataSet.Tables[0].Rows[0]["LastName"] = "FooBar";

...

// Обработчик события RowChanged.
public void RowIsChanged(object sender,
                         DataRowChangeEventArgs e)
{
  Console.WriteLine("After Row changed:" +
                    " Action {0}, " +
                    " Row Value: {1}",
                    e.Action,
                    e.Row["LastName"] );
}

// Обработчик события ColumnChanged.
public void ColumnIsChanged(object sender,
                            DataColumnChangeEventArgs e)
```

Задача ограничения UniqueConstraint достаточно проста, а следовательно, прост и одноименный класс. Он содержит массив столбцов, к которым применяется ограничение, и булево значение, указывающее на факт применения ограничения к первичному ключу.

С другой стороны, ограничение типа ForeignKeyConstraint является более сложным. Его назначение состоит в хранении набора правил, определяющих действия, предпринимаемые по отношению к дочерним строкам, в случае изменения соответствующих строк родительской таблицы. К примеру, таблица Customer является родительской, а таблица Invoice — дочерней. Должен ли объект **DataSet** автоматически применять каскадное удаление всех строк заданного потребителя из таблицы Invoice? Это решение определяется в ограничении типа ForeignKeyConstraint.

Класс ForeignKeyConstraint содержит два массива объектов DataColumn. Первый массив (Columns) предназначен для хранения столбцов, являющихся частью текущей таблицы. Второй массив (RelatedColumns) состоит из столбцов связанной таблицы. Следует отметить, что на этом этапе объект **DataSet** уже может проводить проверку своей целостности.

Триггеры (события объекта DataSet)

Большинство систем управления базами данных позволяет определить набор действий, выполняемых при изменении, добавлении или удалении информации. Например, при добавлении в базу данных нового потребителя может сработать триггер, который внесет этого потребителя в таблицу поиска. Объект DataSet поддерживает регистрацию для получения уведомлений о проведении определенных операций над объектами **DataTable,** что позволит выполнять соответствующий код обработки. Ниже перечислены типы событий, поддерживаемых объектом DataSet:

- RowChanging — срабатывает перед изменениями строк объекта **DataTable;**
- RowChanged — срабатывает после изменения строк объекта **DataTable;**
- ColumnChang ing — срабатывает перед изменениями данных в столбцах;
- ColumnChanged — срабатывает после изменений данных в столбцах;
- RowDeleting — срабатывает перед удалением строки из объекта **DataTable;**
- RowDeleted — срабатывает после удаления строки из объекта **DataTable.**

Подозрительным выглядит отсутствие события добавления строки. На самом деле оно неявно здесь есть, поскольку ADO.NET рассматривает любые изменения свойства Rows (включая добавление новой строки) как предпосылку для срабатывания событий RowChanged и RowChanging. События RowChanging, RowChanged, RowDeleting и RowDeleted предоставляют ссылку на измененную, удаленную или добавленную строку и объект DataRowChangeEventArgs, который содержит значение из перечисления типа DataRowAction. Последнее можно использовать для выяснения того, что же произошло со строкой. Ниже перечислены возможные значение этого перечисления:

- Add — строка была добавлена в таблицу **DataTable;**
- Change — строка была изменена;
- Commit — измененная строка были внесена в базу данных;
- Delete — строка была удалена;
- Nothing — строка не была изменена;
- Rollback — произошла отмена последних внесенных в строку изменений.

```
  Name: Constraint1
    UNIQUECONSTRAINT
    Is on Primary Key: True
      Columns
        Column: InvoiceID
  Name: Customers_Invoices
    FOREIGNKEYCONSTRAINT
    Related Table: Customers
      Columns
        Column: CustomerID
      Related  Columns
        Column: CustomerlD
      Delete Rule:      Cascade
      Update Rule:      Cascade
      AcceptRejectRule: None
Table: InvoiceItems
  Name: Constraint1
    UNIQUECONSTRAINT
    Is on Primary Key: True
      Columns
        Column: InvoiceItemID
  Name: Invoices_InvoiceItems
    FOREIGNKEYCONSTRAINT
    Related Table: Invoices
      Columns
        Column: InvoicelD
      Related Columns
        Column: InvoicelD
      Delete Rule:      Cascade
      Update Rule:      Cascade
      AcceptRejectRule: None
```

Можно заметить, что поскольку для всех таблиц были указаны первичные ключи, для каждого первичного ключа существует ограничение типа UniqueConstraint. Кроме этого, существуют также два ограничения типа ForeignKeyConstraint — отношение между таблицами Customer и invoice и отношение между таблицами Invoice и InvoiceItem.

Несложно догадаться, что наличие в классе UniqueConstraint свойства IsOnPrimaryKey свидетельствует о возможности создавать собственные ограничения типа UniqueConstraint, не относящиеся к ограничению первичного ключа. В действительности подобное ограничение должен иметь каждый столбец базы данных, отмеченный как уникальный (листинг 5.20).

Листинг 5.20. Создание ограничения типа *UniqueConstraint* для нескольких столбцов

```
...

// Создание ограничения UniqueConstraint, гарантирующего
// уникальность пары LastName/FirstName.

// Создание массива, состоящего из
// столбцов LastName и FirstName.
DataColumn[] uniqueCols = new DataColumn[]
  { customerTable.Columns["LastName"],
    customerTable.Columns["FirstName"] };

// Создание ограничения UniqueConstraint.
customerTable.Constraints.Add("UniqueNamesPlease",
                              uniqueCols, false);
```

```
// Вывод состояния всех ограничений таблиц.
foreach (DataTable tbl in dataSet.Tables)
{
  form.WriteLine("Table: {0}", tbl.TableName);
  foreach (Constraint con in tbl.Constraints)
  {
    form.WriteLine(" Name: {0} ", con.ConstraintName);
    if (con is UniqueConstraint)
    {
      form.WriteLine("   UNIQUECONSTRAINT");
      UniqueConstraint u = con as UniqueConstraint;
      form.WriteLine("    Is on Primary Key: {0}",
        u.IsPrimaryKey);
      DumpColumns("Columns",u.Columns, "    ");
    }
    else if (con is ForeignKeyConstraint)
    {
      form.WriteLine("    FOREIGNKEYCONSTRAINT");
      ForeignKeyConstraint f = con as ForeignKeyConstraint;
      form.WriteLine("    Related Table: {0}",
        f.RelatedTable.TableName);
      DumpColumns("Columns", f.Columns, "    ");
      DumpColumns("RelatedColumns", f.RelatedColumns,
                  "    ");
      form.WriteLine("      Delete Rule:      {0}",
        f.DeleteRule);
      form.WriteLine("      Update Rule:      {0}",
        f.UpdateRule);
      form.WriteLine("     AcceptRejectRule: {0}",
        f.AcceptRejectRule);
    }
    else
    {
      form.WriteLine("    UNKNOWN TYPE");
    }
  }
}

private void DumpColumns(string title,
  DataColumn[] cols,
  string indent)
{
  Console.WriteLine("{0}{1}",indent, title);
  foreach (DataColumn col in cols)
  {
    Console.WriteLine("{0} Column: {1}", indent,
                      col.ColumnName);
  }
}
```

Выполнение этого фрагмента кода программы приведет к следующему результату:

```
Table: Customers
  Name: Constraint1
    UNIQUECONSTRAINT
    Is on Primary Key: True
      Columns
        Column: CustomerID
Table: Invoices
```

Обратившись к одному из предыдущих примеров, мы попытаемся считать состояние ограничений (для удобства в листинге 5.19 приведен полный вариант исходного кода этого примера).

Листинг 5,19. Создание полной схемы объекта *DataSet* программным путем

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из Сазы данных таблицы.
SqlDataAdapter daCustomers = new
  SqlDataAdapter("SELECT * FROM CUSTOMER;",
  conn);
SqlDataAdapter daInvoices = new
  SqlDataAdapter("SELECT * FROM INVOICE;",
  conn);
SqiDataAdapter daInvoiceItems = new
  SqlDataAdapter("SELECT * FROM INVOICEITEM;",
  conn);

// Создание пустого объекта DataSet,
DataSet dataSet - new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");
daInvoiceItems.Fill(dataSet, "InvoiceItems");

// Создание переменных, упрощающих доступ к таблицам.
DataTable customerTable    - dataSet.Tables["Customers"];
DataTable invoiceTable     = dataSet.Tables["Invoices"];
DataTable invoiceItemTable - dataSet.Tables["InvoiceItems"];

// Определение первичных ключей.
customerTable.PrimaryKey = new DataColumn[]
                    { customerTable.Columns["CustomerID"] };
invoiceTable.PrimaryKey = new DataColumn[]
                    { invoiceTable.Columns["InvoiceID"] };
invoiceItemTable.PrimaryKey = new DataColumn[]
                    { invoiceItemTable.Columns["InvoiceItemID"] };

// Определение отношений.

// Создание первого отношения (между таблицами Invoices и
// InvoiceItems). При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
dataSet.Relations.Add( "Invoices_InvoiceItems",
  dataSet.Tables["Invoices"].Columns["InvoiceID"],
  dataSet.Tables["InvoiceItems"].Columns["InvoiceID"],
  true);

// Создание второго отношения
//  (между таблицами Customers и Invoices).
dataSet.Relations.Add( "Customers_Invoices",
  dataSet.Tables["Customers"].Columns["CustomerID"],
  dataSet.Tables["Invoices"].Columns["CustomerID"],
  true);
```

Ограничения

Ограничения, относящиеся к объекту DataSet, являются правилами, которым он должен неуклонно следовать. В ADO.NET существуют ограничения двух типов:

- `UniqueConstraint` — позволяет гарантировать уникальность строки на основе столбца или набора столбцов;
- `ForeignKeyConstraint` — указывает объекту DataSet на необходимость применения определенных действий по отношению к строке дочерней таблицы в ответ на изменение соответствующей строки в родительской таблице (таким действием может быть, к примеру, каскадное удаление).

Эти ограничения создаются при определении первичного ключа таблицы и создании отношения с ограничением. Каждое ограничение ForeignKeyConstraint предполагает определение трех правил:

- `DeleteRule` — поведение дочерних строк при удалении родительской строки;
- `UpdateRule` — поведение дочерних строк при обновлении дочерней строки;
- `AcceptRejectRule` — правило, определяющее поведение измененных и удаленных строк при вызове методов `AcceptChanges` и `RejectChanges` объекта DataSet, DataTabEe или **DataRow**.

Различие между этими правилами заключается в том, что правила DeleteRule и UpdateRule применяются при фактическом удалении строк из объекта DataSet, а правило AcceptRejectRule применяется только при вызове методов AcceptChanges и RejectChanges.

Свойства DeleteRule и UpdateRule содержат значение из перечисления Rule. Возможные значения этого перечисления и соответствующее им поведение приведены ниже.

- Cascade — изменения в родительской строке дублируются в дочерней строке. Это поведение принято по умолчанию.
- None — изменения в родительской строке не оказывают никакого эффекта на дочерние строки, что может привести к появлению "осиротевших" строк в дочерних таблицах.
- `SetDefault` — изменения в родительских строках приводят к установке столбцов дочерних строк в значения, принятые по умолчанию. Это также может привести к появлению "осиротевших" строк в дочерних таблицах, однако тогда значение ключа, входящего в ограничение, устанавливается согласно схеме в значение, принятое по умолчанию.
- `SetNull` — изменения в родительской строке приводят к установке столбцов дочерних строк в значения DBNull. Как и SetDefault, значение SetNull может привести к появлению "осиротевших" строк, однако при этом значение ключа, входящего в ограничение, устанавливается равным `DBNull`.

Перечисление AcceptRejectRule содержит только подмножество значений из перечислений DeleteRule/AcceptRule, а именно Cascade и None.

- Cascade — изменения в родительской строке дублируются в дочерней строке. Это поведение принято по умолчанию.
- None — изменения в родительской строке не оказывают никакого влияния на дочерние строки, что может привести к появлению "осиротевших" строк в дочерних таблицах.

Если мы решим расширить предыдущий пример, то к исходному коду должен быть добавлен код из листинга 5.17, ответственный за создание отношений между таблицами.

Листинг 5.17. Создание отношений

```
...

// Определение отношений.

// Создание первого отношения (между таблицами Invoices и
// InvoiceItems). При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
dataSet.Relations.Add( "Invoices_InvoiceItems",
  dataSet.Tables["Invoices"].Columns["InvoiceID"],
  dataSet.Tables["InvoiceItems"].Columns["InvoiceID"],
  true);

// Создание второго отношения (между
// таблицами Customers и Invoices).
dataSet.Relations.Add( "Customers_Invoices",
  dataSet.Tables["Customers"].Columns["CustomerID"],
  dataSet.Tables["Invoices"].Columns["CustomerID"],
  true);
```

Новое отношение добавляется с помощью метода DataRelationCollection. Add(). Первым параметром является имя отношения. Следующие два параметра представляют собой имена родительского и дочернего столбца в отношении соответственно. Наконец, последний параметр является булевой переменной, определяющей необходимость создания ограничения для этого отношения. Создав ограничение, вы тем самым сообщаете объекту DataSet, что каждое значение уникального ключа родительского столбца должно присутствовать в дочернем объекте DataTable.

Но что делать в том случае, когда первичный ключ таблицы состоит из нескольких столбцов? Пример создания массива ключевых столбцов и определение соответствующего ограничения приведены в листинге 5.18.

Листинг 5.18. Добавление отношения для таблиц с составным первичным ключом

```
...

// Создание массива столбцов, входящих а первичный
// ключ родительской таблицы.
DataColumn[] aParentCols =
                    { ds.Tables["Table1"].Columns["key1"],
                      ds.Tables["Table1"].Columns["key2"]};

// Создание массива столбцов, входящих в первичный
// ключ дочерней таблицы.
DataColumn[] aChildCols =
                    {ds.Tables["Table2"].Columns["key1"],
                      ds.Tables["Table2"].Columns["key2"]};

// Создание нового отношения на основе
// составных первичных ключей таблиц,
ds.Relations.Add( "DualKeyRelation",
                  aParentCols,
                  aChildCols,
                  false);
```

являются *реляционными (relational)*, т.е. такими, в которых одни наборы данных связаны с другими наборами данных. В нашем случае таблица `Customer` связана с таблицей Invoice, которая, в свою очередь, связана с таблицей `InvoiceItem`, а та — с таблицей Product (рис. 5.2).

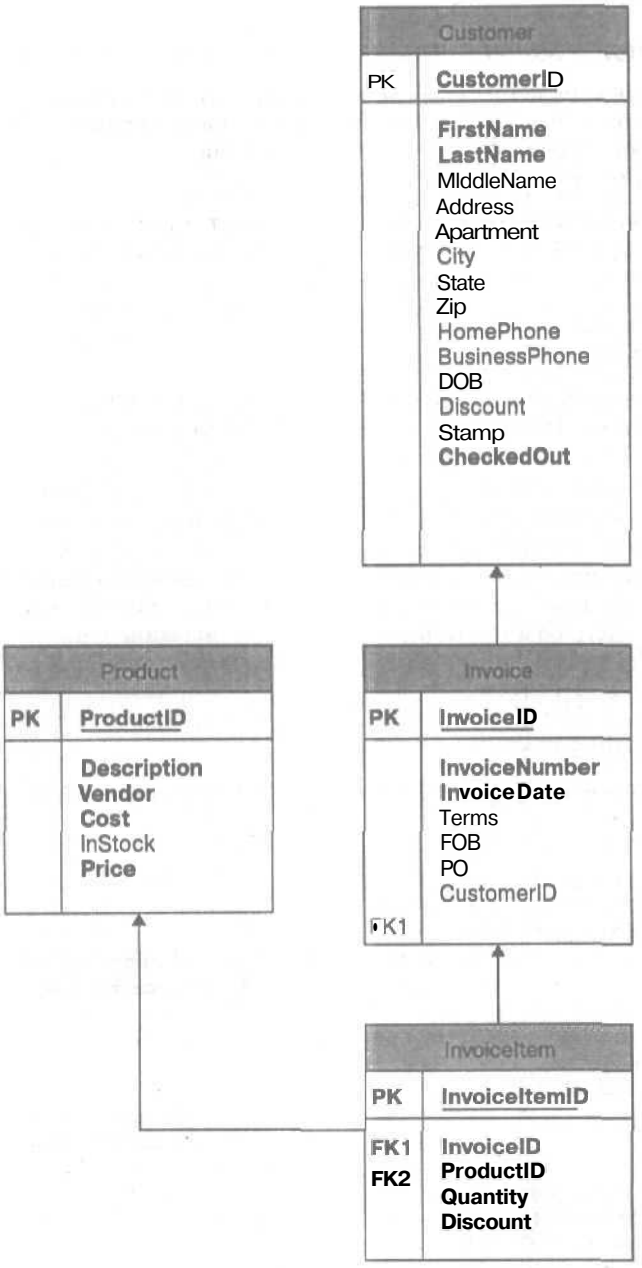


Рис. 5.2. Отношения, существующие между четырьмя таблицами базы данных

```
    // ключа, не содержат уникальных значений.
    Console.WriteLine("Column(s) in the PrimaryKey are " +
                      "not unique.");
    Console.WriteLine("  Exception Thrown:  {0}", ex.Message);
}
```

Стратегии, использующиеся при определении первичного ключа

Одно из первых (и самых важных) решений, которое необходимо принять при проектировании базы данных, касается выбора первичных ключей таблиц. Ниже приведены три наиболее распространенных методики определения первичного ключа, состоящего из одного столбца.

- **Автоинкрементное значение.** Столбец содержит число, однозначно идентифицирующее строку. При добавлении к таблице новой строки число в этом столбце автоматически увеличивается на заданную величину (обычно на единицу). **Если** вам необходимо использовать автоинкрементные ключи на стороне сервера в SQL Server, обратитесь к главе 8, "Обновление базы данных", для получения более подробной информации.
- **Уникальное имя.** Ключевой столбец содержит информацию, уникальную для данной строки. Следует отметить, что в этом случае необходимо быть полностью уверенным в уникальности подобной информации. К примеру, в таблице пациентов в качестве первичного ключа может быть использован номер социального страхования, а в таблице посетителей Web-узла — адрес электронной почты.
- **Глобальный уникальный идентификатор (Globally Unique Identifier — GUID).**[2] Ключевой столбец содержит GUID, который однозначно идентифицирует строку.

Каждый из подходов имеет свои плюсы и минусы. В контексте объекта **DataSet** я бы порекомендовал использовать в качестве первичного ключа столбец, содержащий уникальное имя или GUID. Главная проблема при использовании автоинкрементного столбца заключается в неизбежном перекрытии значений ключа в том случае, если вставка данных в одну таблицу осуществляется посредством более чем одного объекта **DataSet.** Если значения ключа пересекаются, их (а также ссылки на них) необходимо изменить еще до того, как новые строки будут сохранены в базе данных. При использовании GUID или уникальных имен этих проблем можно избежать. Следует отметить, что все приведенные выше предположения делаются с учетом того, что существует контроль над схемой объекта **DataSet.**

Бывают ситуации, в которых программисту приходится работать со старыми системами. Если в вашей базе данных используются автоинкрементные поля, не пытайтесь применять их также и в объектах **DataSet.** В этом случае создание идентификационных номеров можно возложить на базу данных и извлекать их в процессе обновления (более подробно возврат идентификатора из базы данных рассматривается в главе 8, "Обновление базы данных").

Создание отношений

После определения первичного ключа необходимо сообщить объекту DataSet об отношениях, существующих между таблицами. Большинство баз данных, которые используются на сегодняшний день (SQL Server, Oracle, Microsoft Access, DB2 и т.д.),

[2] GUID — это 128-битовые числа, которые гарантированно являются уникальными. В .NET Framework класс `Guid` может использоваться для генерации, анализа и чтения идентификаторов GUID.

```
daInvoiceItems.Fill(dataSet, "InvoiceItems");

// Создание переменных, упрощающих доступ к таблицам.
DataTable customerTable    = dataSet.Tables["Customers"];
DataTable invoiceTable     = dataSet.Tables["Invoices"];
DataTable invoiceItemTable - dataSet.Tables["InvoiceItems"];

// Определение первичных ключей таблиц.
customerTable.PrimaryKey = new DataColumn[]
            { customerTable.Columns["CustomerID"] };
invoiceTable.PrimaryKey = new DataColumn[]
            { invoiceTable.Columns["InvoiceID"] };
invoiceItemTable. PrimaryKey = new DataColumn[]
            { invoiceItemTable.Columns["InvoiceItemID"] };
```

Как вы, наверняка, отметили, в листинге 5.15 первичные ключи представляют собой массивы объектов DataColumn. И это действительно так! Хотя в большинстве ситуаций первичный ключ представляет собой единственный столбец, иногда необходимо использовать ключ, состоящий из нескольких столбцов. Для достижения единообразия обработки первичного ключа ADO.NET требует, чтобы свойство PrimaryKey являлось массивом столбцов.

При определении первичного ключа для таблицы, которая уже содержит данные, нарушающие условие первичного ключа, генерируется исключение ArgumentException. Пример обработки исключения типа ArgumentException приведен в листинге 5.16.

Листинг 5.16. Обработка исключения типа *ArgumentException*

```
// Создание объекта DataAdapter для таблицы Customer.
SqlDataAdapter daCustomers = new
                  SqlDataAdapter("SELECT * FROM CUSTOMER",
                                 conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
daCustomers.Fill(dataSet, "Customers");

// Создание переменной, упрощающей доступ к таблице Customer.
DataTable customerTable    = dataSet.Tables["Customers"];

// Добавление дублирующей строки перед определением первичного
// ключа - это позволит нам обработать исключение.
customerTable.Rows.Add(customerTable.Rows[0].ItemArray);

try
{
  // Определение первичного ключа.
  customerTable.PrimaryKey = new DataColumn[]
                  { customerTable.Columns["CustomerID"] };
}
catch (ArgumentException ex)
{
  // Вывод на консоль сообщения об ошибке. В данной случае
  // исключение типа ArgumentException говорит нам о том, что
  // столбцы, определенные в качестве (части) первичного
```

Создание схемы объекта DataSet программным путем

Бывают ситуации, в которых создание полной схемы объекта **DataSet** возможно только программным путем. На первый взгляд это может показаться очень утомительным занятием, требующим написания кода огромного размера, однако существуют и весьма приемлемые альтернативы. В конце концов на определенном этапе у вас обязательно возникнет желание разобраться во внутренней структуре объекта **DataSet.**

Кроме того, я советую создавать классы, которые непосредственно наследуют класс DataSet (несмотря на то что в примерах данной главы это не отражено). Создав класс-потомок класса DataSet, вы можете указать его схему на этапе инициализации, как показано в листинге 5.14.

Листинг 5.14. Наследование класса *DataSet*

```
class PragmaticDataSet : DataSet
{
  public PragmaticDataSet()
  {
    // Здесь можно разместить определение схемы.
  }

  ...

}
```

Первичные ключи

Как и при работе с базой данных, вы можете определить для таблицы первичный ключ. Первичный ключ используется для обеспечения уникальности вводимых в таблицу строк. Установив значение свойства PrimaryKey объекта **DataTable,** можно быть уверенным, что при внесении в таблицу повторяющейся строки будет сгенерировано исключение. Это очень эффективная техника, позволяющая не обращаться к базе данных только лишь затем, чтобы выяснить, что такая строка в таблице уже есть. Пример добавления первичного ключа приведен в листинге 5.15.

Листинг 5.15. Создание первичного ключа

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
SqlDataAdapter daCustomers = new
          SqlDataAdapter("SELECT * FROM CUSTOMER",
                         conn);
SqlDataAdapter daInvoices = new
          SqlDataAdapter("SELECT * FROM INVOICE",
                         conn);
SqlDataAdapter daInvoiceItems = new
          SqlDataAdapter("SELECT * FROM INVOICEITEM",
                         conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");
```

```
// из базы данных таблицы.
SqlDataAdapter dataAdapter =
          new SqlDataAdapter("SELECT * FROM CUSTOMER",conn);

// При необходимости объект DataAdapter создаст схему,
// включив в нее информацию о первичном ключе.
dataAdapter.MissingSchemaAction =
                              MissingSchemaAction.AddWithKey;

// Создание пустого объекта "DataSet.
DataSet dataSet = new DataSet();

// Добавление в объект DataSet новой таблицы DataTable
// с именем "Customers".
dataAdapter.FillSchema(dataSet,
                       SchemaType.Mapped,
                       "Customers");
```

Использование языка XSD для определения схемы объекта DataSet

Как вы далее увидите, создание сложной схемы объекта **DataSet** может потребовать написания большого объема кода, С целью повышения производительности и облегчения поддержки стоит обеспечить возможность сериализаиии схемы объекта **DataSet** так, чтобы ею можно было манипулировать отдельно от кода, проводящего заполнение объекта **DataSet** данными. Это можно сделать путем создания объекта **DataSet и** сохранения его схемы в файле .XSD. В результате при каждом создании объекта **DataSet** файл .XSD **можно** будет использовать для определения схемы, как показано в листинге 5.13.

Листинг 5.13. **Определение схемы объекта** *DataSet* с помощью **файла** .XSD

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
SqlDataAdapter dataAdapter =
          new SqlDataAdapter("SELECT * FROM CUSTOMER",conn);

// Если объект DataAdapter обнаружит отсутствие
// схемы, он сгенерирует исключение.
dataAdapter.MissingSchemaAction = MissingSchemaAction.Error;

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Загрузка схемы из XSD-файла.
dataSet.ReadXmlSchema(@"customer.xsd");

// В этом случае не должно генерироваться исключение, так как
// в файле customer.xsd содержится схема таблицы Customers.
dataAdapter.Fill(dataSet, "Customers");
```

Обратите внимание, что в приведенном выше коде загружается схема для всего объекта **DataSet.** Создание модульных файлов .XSD, использующихся для заполнения отдельных объектов **DataTable,** не поддерживается. Более подробно интеграция XML и объекта **DataSet** рассматривается в главе 9.

живать нижележащий поставщик данных OLE DB. Более подробно перечисление MissingSchemaAction описано в документации MSDN.

- Error — если в объекте DataSet отсутствует схема, необходимая для добавления данных из объекта DataAdapter, генерируется исключение.
- Ignore — дополнительные столбцы, о которых "знает" объект **DataAdapter** и которые отсутствуют в **схеме** объекта **DataSet,** игнорируются.

Значения из перечисления MissingSchemaAction позволяют указать поведение объекта **DataAdapter** в том случае, если необходимая схема отсутствует. Проще указать схему заранее и установить свойство MissingSchemaAction в значение Error. После этого, если объект **DataAdapter** попытается внести в объект **DataSet** не ожидавшиеся данные, будет сгенерировано сообщение об ошибке, и нам не придется иметь дело с "фантомными" строками (листинг 5.11)

Листинг 5.11. Заполнение объекта DafaSet с использованием параметра *MissingSchemaAction. Error*

```
// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
SqlDataAdapter dataAdapter =
          new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);

// Если объект DataAdapter обнаружит отсутствие
// схемы, он сгенерирует исключение.
dataAdapter.MissingSchemaAction = MissingSchemaAction.Error;

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// В данном случае будет сгенерировано исключение,
// поскольку объект DataSet пуст (не содержит схемы).
dataAdapter.Fill(dataSet, "Customers");
```

Для того чтобы схема была создана объектом **DataAdapter,** можно воспользоваться методом DataAdapter. FillSchema(). Этот метод выводит схему на основе информации, хранящейся в базе данных, точно так же, как это делается обычно; отличие заключается в том, что схема создается заранее. Метод Fillschema() может оказаться полезным при создании объекта **DataSet,** который будет заполняться позднее или будет заполнен пользовательскими данными. В качестве параметров метод FillSchema () принимает объект **DataSet** или DataTable и значение из перечисления SchemaType. Параметр типа SchemaType используется для указания объекту **DataAdapter** на необходимость использования свойства TableMappings (SchemaType.Mapped). В противном случае используется схема, хранящаяся в базе данных (SchemaType. Source). Так как определение значения свойства TableMappings объекта **DataAdapter** предполагает проведение активных действий, параметр типа SchemaType почти всегда будет иметь значение SchemaType .Mapped. Если же свойство TableMappings необходимо игнорировать, просто не задавайте его значение. Свойство MissingSchemaAction, как и прежде, используется для указания методу FillSchema() на способ добавления схемы, как показано в листинге 5.12.

Листинг 5.12. Заполнение объекта *DataSet* с параметром *MissingSchemaAction.AddWithKey*

```
...

// Создание объекта DataAdapter для каждой извлекаемой
```

шлось познакомиться с сотнями различных вариантов схем баз данных, большая часть которых представляла собой минимально необходимые схемы. Однако как можно определить этот минимум? Несмотря на то что каждому конкретному случаю соответствовала своя схема, чаще всего она включала в себя определения таблиц, столбцов и, иногда, первичных ключей. При более глубоком изучении исходного кода приложения оказывалось, что в нем была реализована функциональность, которую легко могла обеспечить сама база данных Тем не менее я не хочу выступать здесь в роли беспощадного критика, так как сам иногда опускался до подобной практики.

Как определить, обладает ли схема базы данных достаточной полнотой? Если вы когда-либо писали код (это касается и класса ADO.NET DataSet) для реализации отношений, ограничений, триггеров и других функциональных возможностей схемы базы данных, то вам определенно стоит уделить более пристальное внимание вопросу проектирования схемы базы данных.

На самом деле я хочу сказать только то, что база данных (объект DataSet) должна выполнять "свою" работу, например:

- если в приложении присутствует код для проверки уникальности строк таблицы, добавьте к таблице первичный ключ;
- если в приложении присутствует код для проверки уникальности отдельного столбца, добавьте к таблице ограничение;
- если в приложении присутствует код для связывания таблиц родителей и потомков, создайте отношение;
- если в приложении присутствует код, обеспечивающий гарантированное изменение или удаление связанных строк, разрешите для соответствующего отношения каскадные обновления и удаления данных;
- если в приложении присутствует код, отслеживающий изменение состояния базы данных, добавьте к последней триггер;
- если все запросы к базе данных содержат много операторов JOIN, создайте представление.

Более подробно использование схемы базы данных рассматривается в книгах Джо Келко (Joe Celko) (www.celco.com/books.html).

Вывод схемы объекта DataSet с помощью объекта DataAdapter

Каждый раз, когда объект **DataSet** заполняется с помощью объекта **DataAdapter**, **он** должен содержать схему столбцов (объекты **DataColumn**) для обеспечения возможности принимать новые **строки** данных. Если на текущий момент объект **DataSet** не содержит **схемы,** объект **DataAdapter** выведет ее на основе результатов выполнения запроса (SelectCommand). Процессом вывода схемы можно управлять. **Объект DataAdapter** содержит свойство MissingSchemaAction, позволяющее указать действия, которые необходимо предпринять в том случае, если в объекте **DataSet** отсутствует нужная схема. Свойство MissingSchemaAction может принимать следующие значения.

- Add — к схеме будут добавлены столбцы, необходимые для заполнения объекта **DataSet** с помощью объекта **DataAdapter** (значение по умолчанию).
- AddWithKey — **к схеме** будут добавлены столбцы, необходимые для заполнения объекта **DataSet** с помощью объекта **DataAdapter.** Кроме того, объект **DataAdapter** пытается извлечь из базы данных информацию о ключах и определяет первичные ключи заполняемых им объектов DataTable. При использовании управляемого поставщика OLE DB эту функциональность должен поддер-

```
dataTable.Columns.Add("fname",
                      System.Type.GetType("System.String"));

// Добавление строк данных.
object[] aSmoltz = {1, "Smoltz", "John"};
object[] aMaddux = {2, "Maddux", "Greg"};
object[] aGlavine = {3, "Glavine", "Tom"};
dataTable.Rows.Add( aSmoltz );
dataTable.Rows.Add( aMaddux );
dataTable.Rows.Add( aGlavine );
```

После завершения этих действий с полученным объектом **DataSet** можно обращаться точно так же, как и с объектом **DataSet,** полученным на основе информации из базы данных или на основе информации из XML-файла.

Определение схемы объекта DataSet

Правильна ли ваша схема? Не мешает ли она в работе? Предназначение схемы базы данных заключается в определении правил, которым должна соответствовать хранящаяся в базе данных информация. Проблема при этом заключается в том, что программисты вынуждены отражать такие правила в своем коде, чтобы не допустить ошибок в программе. **Схема** объекта **DataSet** позволяет эффективно определить эти правила, избавляя программиста от необходимости писать сложный код.

После загрузки данных в объект **DataSet** его создание можно считать завершенным только наполовину. В большинстве примеров объектов **DataSet,** которые я встречал, второй шаг был попросту пропущен. Что же собой представляет схема объекта **DataSet?** После загрузки данных в объект **DataSet** таблицы имеют схему с минимально возможным объемом информации, состоящую, в основном, из имен таблиц, столбцов и типов столбцов. Полагаясь лишь на такую схему, можно серьезно усложнить себе жизнь — в ней отсутствуют ключи, ограничения, отношения и триггеры.

Для того чтобы заставить объект **DataSet** вести себя так же, как база данных, необходимо определить все перечисленные выше элементы. Для этого можно воспользоваться несколькими способами: заставить объект **DataAdapter** вывести схему из запросов, загрузить содержащий необходимую схему XSD-файл, а также создать схему программным путем. Все эти способы определения схемы объекта **DataSet** будут рассмотрены в следующих разделах данной главы.

Зачем нужно определять схему объекта DataSet?

Каждый объект **DataSet,** содержащий данные, имеет определенную схему. Таблица **DataTable** с заданными правилами для каждого столбца строки **DataRow** включает в себя объекты **DataColumn. A** как бы со схемой, содержащейся в базе данных? Зачем вам необходимы в объекте **DataSet** все **эти** ключи, отношения, ограничения и триггеры, особенно если точно такая же схема присутствует в базе данных? Причина дублирования схемы базы данных в объекте **DataSet** заключается в возможности обнаружения ошибки задолго до того, как будет предпринята попытка сохранить информацию в базе данных. Таким способом можно не только уменьшить общее количество обращений к базе данных, но также избежать обращений, результатом которых станет всего лишь уведомление о нарушении схемы. Чем раньше мы найдем ошибки в схеме, тем проще их будет исправить.

Обладает ли схема базы данных достаточной полнотой?

Я занимаюсь разработкой приложений, ориентированных на взаимодействие с базами данных, дольше, чем, возможно, мне этого хотелось бы. В течение этого времени мне при-

```
SqlDataAdapter dataAdapter = new SqlDataAdapter(query,
                                                conn);

// Создание объекта DataSet и его заполнение информацией,
// хранящейся в базе данных.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet информацией,
// хранящейся в базе данных.
dataAdapter.Fill(dataSet);

// Добавление в объект DataSet данных из XML-документа.
dataSet.ReadXml(@"products.xml",
                XmlReadMode.InferSchema);

...
```

Как показано в приведенном выше примере, часть информации считывается из базы данных, а часть — из XML-документа. При вызове метода ReadXml() используется директива XmlReadMode.InferSchema, потому что по умолчанию при чтении XML-данных в объект **DataSet,** уже содержащий данные, метод ReadXml() попытается сопоставить схему XML-файла с текущей схемой объекта **DataSet.** Поскольку при вызове метода в объекте **DataSet** отсутствует таблица Product, без указания на необходимость вывода схемы объект **DataSet** не сможет принять данные из XML-файла. При желании эту проблему можно было обойти, загрузив XML-данные прежде информации из базы данных, так как в случае отсутствия схемы объект **DataSet** создает ее сам (согласитесь, что вывод схемы только в случае ее отсутствия является несколько сбивающим с толку поведением). Если же считываемый XML-файл содержит в себе дополнительные строки таблиц, уже существующих в объекте **DataSet,** директива InferSchema нам просто не нужна.

Более подробно интеграция XML и объекта **DataSet** рассматривается в главе 9, "ADO.NET и XML".

Создание объекта DataSet программным путем

В некоторых случаях нам может понадобиться создать объект **DataSet** программным путем. Создание объекта **DataTablc** вручную и его заполнение данными позволит вам лучше узнать структуру объекта **DataSet.** Объект **DataSet** создается сверху вниз. Другими словами, сначала мы создаем сам объект **DataSet,** затем добавляем в него объект **DataTable и** определяем информационную схему, создавая объекты **DataColumn.** Наконец, в объект **DataTable** добавляются строки данных, как показано в листинге 5.10.

Листинг 5.10. Создание объекта DataSet программным **путем**

```
// Создание объекта DataSet.
DataSet dataSet = new DataSet();

// Добавление нового объекта DataTable в коллекцию
// объектов DataTable объекта DataSet.
DataTable dataTable - dataSet.Tables.Add("Pitchers");

// Добавление столбцов (определение информационной схемы).
dataTable.Columns.Add("pitcher_id",
                      System.Type.GetType("System.Int64"));
dataTable.Columns.Add("lname",
                      System.Type.GetType("System.String"));
```

из XML-файла или из места, указанного в URL-адресе, не используя при этом управляемый поставщик, как показано в листинге 5.8.

Листинг 5.8. Создание объекта *DataSet* на основе XML-документа

```
// Заполнение объекта DataSet из XML-файла
// или из места, указанного в URL-адресе.
DataSet dsFile = new DataSet();
dsFile.ReadXml(@"c:\test.xml");

// Заполнение объекта DataSet на основе объекта TextReader
// (класс StringReader является производным от TextReader).
DataSet dsTextReader = new DataSet();
StringReader textReader =
             new StringReader("<xml><foo>hello</foo></xml>");
dsTextReader.ReadXml(textReader);
textReader.Close();

// Заполнение объекта DataSet на основе объекта XmlReader
// (класс XmlTextReader является производным от XmlReader).
DataSet dsXmlReader = new DataSet();
XmlTextReader xmlReader = new XmlTextReader(@"c:\test.xml");
dsXmlReader.ReadXml(xmlReader);
xmlReader.Close();

// Заполнение объекта DataSet на основе объекта Stream
// (класс FileStream является производным от Stream).
DataSet dsStream = new DataSet();
FileStream fs =
          new FileStream(@"c:\test.xml", FileMode.Open);
dsXmlReader.ReadXml(fs);
fs.Close();
```

Метод ReadXml O может принимать несколько вариантов входных данных: текст в формате XML, путь к файлу, объект **XmlReader,** объект **StringReader** или даже объект **Stream.**

Так как объект **DataSet** имеет реляционную структуру, его создание на основе XML-документа может показаться странным — ведь по своей природе последний иерархичен, Несмотря на то что объект **DataSet** не является самым лучшим контейнером XML-данных, он идеально подходит в качестве средства "смешивания" информации из базы данных и данных в формате XML. Чтобы "приготовить" эту смесь, необходимо использовать объект **DataAdapter** для извлечения информации из базы данных и метод ReadXml () для ее загрузки из XML-документа, как показано в листинге 5.9.

Листинг 5.9. "Смешивание" информации из базы данных **и XML документа**

```
...

// Создание строки-запроса query.
// Для того чтобы извлечь из базы данных несколько
// таблиц, мы используем пакетный запрос, возвращающий
// несколько результирующих наборов данных.
string query = "";
query += "SELECT * FROM CUSTOMER;";
query += "SELECT * FROM INVOICE;";
query += "SELECT * FROM INVOICEITEM;";

// Создание объекта DataAdapter для извлечения информации
// из базы данных.
```

```
dataAdapter.TableMappings.Add("ADONET1", "Invoices");
dataAdapter.TableMappings.Add("ADONET2", "InvoiceItems");

// Определение отображений имен столбцов.
dataAdapter.TableMappings["ADONET"].
                ColumnMappings.Add("CustomerID", "ID");
dataAdapter.TableMappings["ADONET1"].
                ColumnMappings.Add("InvoiceID", "ID");
dataAdapter.TableMappings["ADONET2"].
                ColumnMappings.Add("InvoiceItemID", "ID");

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
dataAdapter.Fill(dataSet, "ADONET");
```

Как показано в листинге 5.6, объект DataAdapter получает указание переименовать поле `CustomerID` таблицы Customer, поле `InvoiceID` таблицы Invoice и поле `InvoiceItemID` таблицы `InvoiceItem` в ID. Единственным требованием для переименования столбцов в объекте DataTable является существование соответствующего объекта **DataTableMapping**. Если использовать объект **DataTableMapping** для конкретного объекта DataTable нет необходимости, то с целью обеспечения поддержки объекта **DataColumn**Mapping можно создать пустой объект DataTableMapping, как показано в листинге 5.7.

Листинг 5.7. Использование свойства *ColumnMappings* без изменения имен таблиц

```
...

// Создание объекта DataAdapter для каждой извлекаемой
// из базы данных таблицы.
SqlDataAdapter dataAdapter = new SqlDataAdapter(
  "SELECT * FROM CUSTOMER",
  conn);

// Определение 'фиктивного' отображения DataTableMapping.
dataAdapter.TableMappings.Add("Customers", "Customers");

// Определение отображения для имени столбца.
dataAdapter.TableMappings["Customers"].
                ColumnMappings.Add("CustomerID", "ID");

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
dataAdapter.Fill(dataSet, "Customers");
```

В этом примере создается "фиктивный" объект DataTableMapping, не производящий переименования таблицы. Обратите внимание, что имена таблиц в вызове метода `TableMappings.Add()` должны совпадать с именем таблицы в вызове метода `DataAdapter.Fill()`.

Создание объекта DataSet на основе XML-документа

Времена меняются — сейчас мы должны быть готовы интегрировать наши данные с данными из произвольных источников (отличных от базы данных). Для того чтобы создать объект DataSet на основе XML-документа, необходимо считать в него данные

Листинг 5.5. Использование свойства *TableMappings* при заполнении объекта *DalaSet* таблицами с измененными именами

```
...

// Создание строки-запроса query.
// Для того чтобы извлечь из базы данных несколько
// таблиц, мы используем пакетный запрос, возвращающий
// несколько результирующих наборов данных.
string query = "";
query += "SELECT * FROM CUSTOMER;";
query += "SELECT * FROM INVOICE;";
query += "SELECT * FROM INVOICEITEM;";

// Создание объекта DataAdapter для извлечения данных.
SqlDataAdapter dataAdapter = new SqlDataAdapter(query,
                                                conn);

// Определение отображений.
dataAdapter.TableMappings.Add("ADONET", "Customers");
dataAdapter.TableMappings.Add("ADONET1", "Invoices");
dataAdapter.TableMappings.Add("ADONET2", "InvoiceItems");

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с использованием временного имени.
dataAdapter.Fill(dataSet, "ADONET");
```

В этом листинге метод `DataAdapter.Fill()` использует в качестве базового имени таблицы имя "ADONET", в результате чего после выполнения пакетного запроса таблицы получают имена ADONET, ADONET1 и ADONET2, а стандартным базовым именем таблицы является "Table". На практике одновременное использование свойства TableMappings и определение имени таблицы при вызове метода `Fill` встречается редко.

Кроме определения имен таблиц, свойство TableMappings может использоваться для переименования столбцов (при этом первоначальные имена столбцов, возвращаемые в результате выполнения запроса, заменяются новыми именами, использующимися в объекте **DataTable**). Для поддержки этой функциональности класс `DataTableMapping` содержит коллекцию объектов **DataColumnMapping**, задающих отображение между именами полей в запросе и **именами** полей в объекте **DataSet** (листинг 5.6).

Листинг 5.6. Использование свойства *ColumnMappings*

```
...

// Создание строки-запроса query.
// Для того чтобы извлечь из базы данных несколько
// таблиц, мы используем пакетный запрос, возвращающий
// несколько результирующих наборов данных.
string query = "";
query 4= "SELECT * FROM CUSTOMER;";
query += "SELECT * FROM INVOICE;";
query += "SELECT * FROM INVOICEITEM;";

// Определение отображений имен таблиц.
dataAdapter.TableMappings.Add("ADONET", "Customers");
```

```
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter,
// присваивая таблицам DataTable информативные имена.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");
daInvoiceItems.Fill(dataSet, "InvoiceItems");
```

Хотя теперь таблицы имеют более понятные имена, при этом совершается три обращения к базе данных, что очень неэффективно. Для того чтобы **иметь** информативные имена таблиц при использовании пакетного запроса, следует воспользоваться свойством объекта **DataAdapter** TableMappings.

Использование свойства TableMappings

Свойство TableMappings объекта **DataAdapter** является коллекцией объектов DataTableMapping, которые используются для сообщения объекту **DataAdapter** имен объектов **DataTable** внутри объекта **DataSet,** а также **имен** столбцов внутри каждой таблицы. Для того чтобы использовать свойство TableMappings для назначения имен таблицам, просто добавьте к нему отображение старого имени таблицы на новое имя таблицы, как показано в листинге 5.4.

Листинг 5.4. Использование свойства *TableMappings*

```
...

// Создание строки-запроса query.
// Для того чтобы извлечь из базы данных несколько
// таблиц, мы используем пакетный запрос, возвращающий
// несколько результирующих наборов данных.
string query = "";
query += "SELECT * FROM CUSTOMER;";
query += "SELECT * FROM INVOICE;";
query += "SELECT * FROM INVOICEITEM;";

// Создание объекта DataAdapter для извлечения данных.
SqlDataAdapter dataAdapter = new SqlDataAdapter(query,
                                                conn);
// Определение отображений.
dataAdapter.TableMappings.Add("Table", "Customers");
dataAdapter.TableMappings.Add("Table1", "Invoices");
dataAdapter.TableMappings.Add("Table2", "InvoiceItems");

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
dataAdapter.Fill(dataSet);
```

Как уже упоминалось ранее, в результате вызова метода DataAdapter.Fill() создаются таблицы с именами Table, Table1 и Table2. Поскольку нам известен алгоритм назначения стандартных имен, достаточно просто задать отображение имен с целью переименования Table в Customers, Table1 в Invoices, а Table2 в Invoiceltems. Кроме того, методу DataAdapter.Fill О можно указать на необходимость присвоения объектам **DataTable** более полезных имен, как показано в листинге 5.5.

Множественные объекты DataTable

Вскоре вы поймете, что больше всего достоинства объекта DataSet проявляются при его использовании в качестве реляционного хранилища данных, расположенного в памяти. Вам довольно часто придется создавать объекты **DataSet,** содержащие более одного объекта **DataTable.** Это можно сделать разными способами. Мы рассмотрим каждый из них, отмечая по ходу их плюсы и минусы. В листинге 5.2 показано использование пакетных запросов[1] для получения из базы данных нескольких таблиц,

Листинг 5.2. Использование пакетного запроса

```
...

// Создание строки-запроса query.
// Для того, чтобы извлечь из базы данных несколько
// таблиц, мы используем пакетный запрос, возвращающий
// несколько результирующих наборов данных.
string query = "";
query += "SELECT * FROM CUSTOMER;";
query += "SELECT * FROM INVOICE;";
query += "SELECT * FROM INVOICEITEM;";

// Создание объекта DataAdapter для извлечения данных,
SqlDataAdapter dataAdapter = new SqlDataAdapter(query,
                                                conn);

// Создание и заполнение объекта DataSet.
DataSet dataSet = new DataSet();
dataAdapter.Fill(dataSet);
```

Все было бы хорошо, если бы только таблицам автоматически не присваивались "универсальные" имена. В этом примере объекты **DataTable** названы Table, Table1 и Table2 соответственно. Если вам необходимо иметь более информативные имена таблиц, вы можете использовать один объект **DataAdapter** на каждую из таблиц объекта **DataSet,** как показано в листинге 5.3.

Листинг **5.3.** Заполнение **объекта *DataSet* с использованием информативных имен таблиц**

```
...

// Создание объекта DataAdapter для каждой из таблиц,
// извлекаемых из базы данных.
SqlDataAdapter daCustomers = new
  SqlDataAdapter("SELECT * FROM CUSTOMER;",
  conn);
SqlDataAdapter daInvoices = new
  SqlDataAdapter("SELECT * FROM INVOICE;",
  conn);
SqlDataAdapter daInvoiceItems - new
  SqlDataAdapter("SELECT * FROM INVOICEITEM;",
  conn);

// Создание пустого объекта DataSet.
```

[1] Обратите внимание, что не все производители баз данных поддерживают пакетные запросы. Для того чтобы ознакомиться со всеми способами заполнения объекта DataSet несколькими таблицами, прочитайте весь раздел. Другие описанные в нем методы более независимы от производителя базы данных.

DataAdapter в качестве "моста" между объектом **DataSet** и базой данных **можно** встретить на всем протяжении этой главы, а также в последующих главах части II этой книги.

Еше одной важной задачей объекта **DataAdapter** является минимизация времени, в течение которого соединение будет оставаться открытым. При рассмотрении объекта **DataAdapter** вы наверняка обратите внимание на то, что явного открытия или закрытия соединения не происходит. **DataAdapter** знает, что соединение должно быть как можно более коротким, и самостоятельно управляет его открытием и закрытием. Если использовать объект **DataAdapter** совместно с уже открытым соединением, то состояние соединения будет сохранено (т.е. соединение не будет ни открыто, ни закрыто).

Создание объекта DataSet на основе информации, хранящейся в базе данных

В самом простом случае объект **DataSet** будет использоваться для хранения одной таблицы, как показано в листинге 5.1.

Листинг 5.1. Создание объекта *DataSet* на основе информации, хранящейся в базе данных

```
...

// Создание объекта Command для извлечения из базы
// данных таблицы Customer.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Создание объекта DataAdapter для заполнения объекта DataSet.
SqlDataAdapter dataAdapter = new SqlDataAdapter(cmd);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объекта DataAdapter.
dataAdapter.Fill(dataSet);

...
```

В этом примере объект **DataSet** заполняется результирующим набором данных — таблицей CUSTOMER. На данном этапе читатель должен быть знаком с созданием объектов **Connection и Command,** а поэтому нам достаточно рассмотреть подробно только выделенный код.

1. Создается экземпляр класса SqlDataAdapter. SqlDataAdapter является специализированным классом, выполняющим функцию контейнера для команд, осуществляющих чтение и запись информации в базу данных.
2. Создается экземпляр класса DataSet. Только что созданный объект **DataSet** является пустым.
3. Вызывается метод Fill о объекта **DataAdapter** для заполнения объекта **DataSet** данными из запроса, определенного ранее в объекте **Command.**

Это очень простой пример заполнения объекта **DataSet** информацией из одной таблицы на основе одного SQL-запроса. Далее в этой главе мы рассмотрим более сложные аспекты создания объектов DataSet, не затрагивая при этом вопрос обновления базы данных. **Более** подробно обновление базы данных рассматривается в главе 8.

- `System.Data.Common` — содержит базовые классы и интерфейсы поставщиков данных;
- `System.Data.SqlClient` — содержит классы управляемого поставщика данных SQL Server;
- `System.Data.OleDb` — содержит классы управляемого поставщика данных OLE DB;
- `System.Data.Odbc` — содержит классы управляемого поставщика данных ODBC;
- `System.Data.OracleClient` — содержит классы управляемого поставщика данных Oracle.

В отличие от предыдущих уровней доступа к данным Microsoft, объект **DataSet** не имеет никакого внутреннего представления об источнике данных. Таким образом, объект DataSet оказывается отсоединенным не только от данных, но и от информации об их происхождении. Следовательно, нам необходимо иметь что-то, что позволило бы соединить объект DataSet и источник данных.

Заполнение объекта DataSet

Так как объект **DataSet** не связан с базой данных, его можно воспринимать как реляционное хранилище данных. Для того чтобы заполнить объект DataSet, совсем не обязательно использовать информацию, хранящуюся в базе данных. На практике существует три метода заполнения объекта **DataSet:** с помощью объекта DataAdapter (при этом информация, как правило, извлекается из базы данных), на основе XML-документа и вручную.

Объект DataAdapter

Объект **DataAdapter** является связующим звеном между объектом **DataSet** и хранилищем данных. С помощью объекта **DataAdapter** можно заполнять объект DataSet информацией из хранилища данных, а также обновлять хранилище данных на основе объекта **DataSet.** Фактически объект **DataAdapter** представляет собой метакоманду, Так, объект **DataAdapter** состоит из четырех объектов Command, каждый из которых выполняет строго определенную задачу:

- `SelectCommand` — используется для извлечения данных из хранилища;
- `InsertCommand` — используется для добавления новых записей, созданных в объекте **DataSet,** в нижележащее хранилище;
- `UpdateCommand` — используется для обновления существующих записей в хранилище на основе изменений в объекте **DataSet;**
- `DeleteCommand` — используется для удаления существующих записей из хранилища на основе удалений в объекте **DataSet.**

Объект **DataAdapter** используется каждый раз, когда объект **DataSet** нуждается в непосредственном взаимодействии с источником данных. Один из наиболее важных моментов здесь заключается в том, что объект **DataAdapter** содержит объекты **Command** для каждой из основных операций, производимых над базами данных: SELECT, INSERT, UPDATE и DELETE. Основное предназначение объекта **DataAdapter** заключается в формировании "моста" между объектом **DataSet** и базой данных. Поскольку **таким** "мостом" является сам объект **DataAdapter,** он должен иметь возможность выполнять все основные операции над базой данных. Примеры использования объекта

- **DataColumn** — коллекция правил, описывающая данные, которые можно хранить в объектах **DataRow.**
- **DataRow** — коллекция данных, которая представляет собой одну строку таблицы **DataTable. Объект DataRow** является фактическим хранилищем данных.
- **Constraint** — правило, которое задает допустимость хранения определенных данных в объекте **DataTable.** Объект **Constraint** используется для определения бизнес-правил объекта **DataSet.**
- **DataRelation** — описание навигационных связей между различными объектами **DataTable.** В большинстве случаев объект **DataRelation** сопровождается объектом **Constraint,** позволяющим строго задать отношение.

Как показано на рис. 5.1, внутри объекта **DataSet может** храниться коллекция объектов **DataTable.** В свою очередь внутри объектов **DataTable** хранятся коллекции объектов **DataRow, DataColumn, Constraint и** две коллекции объектов **DataRelation.** Коллекция объектов **DataRelation** соответствует формирующим связь между объектами **DataTable** отношениям "родитель-потомок". Коллекция объектов **DataRelation** внутри объекта **DataSet** является агрегированным представлением всех объектов **DataRelation** во всех объектах **DataTable.**

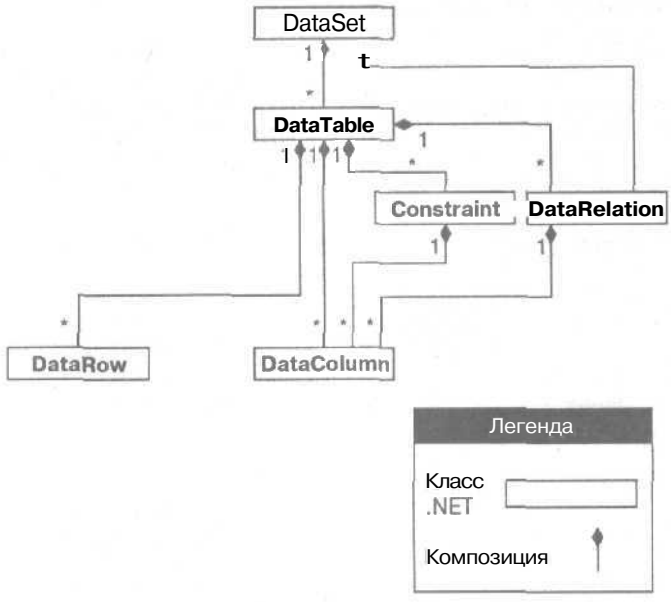


Рис. 5.1. Структура объекта DataSet

Объект DataSet и управляемые поставщики данных

В предыдущих главах рассматривались методы взаимодействия с хранилищами данных (как правило, с базами данных). По идее, было бы вполне логично предположить существование какой-либо связи между объектом **DataSet и** доступом к хранилищу данных. К сожалению, это не так. Компания Microsoft специально спроектировала пространства **имен,** с тем чтобы подчеркнуть отсутствие подобной связи:

- `System.Data` — содержит классы, которые входят в объект **DataSet,** а также интерфейсы, использующиеся для определения управляемых поставщиков;

Глава 5

Создание объекта DataSet

В этой главе...

Что такое объект DataSet?

Заполнение объекта DataSet

Определение схемы объекта DataSet

Так что же представляет собой объект **DataSet?** Для начала попытайтесь отвлечься от сложностей отсоединенного режима доступа к данным. Объект **DataSet** — это просто реляционная структура, хранимая в памяти. Но что это дает в контексте повседневной работы с объектом? Если вспомнить мои замечания о стоимости сетевого трафика и соединений с базой данных, то становится понятно, что хранение данных и правил, связанных с этими данными, на стороне клиента или на среднем уровне может сократить общую стоимость применения правил к данным.

Что такое объект DataSet?

Если вы знакомы с другими API доступа к данным, то можете попытаться найти в них объекты, аналогичные объекту **DataSet.** Не скрою, что этот поиск, скорее всего, завершится неудачей. Какое отношение имеет объект **DataSet** к программированию доступа к базе данных или, скажем, к XML? А может это что-то совершенно новое? Объект **DataSet — это:**

- набор информации, извлеченной из базы данных; доступ к этому набору осуществляется в отсоединенном режиме;
- база данных, расположенная в памяти;
- сложная реляционная структура данных с встроенной поддержкой XML-сериализации.

В самом деле, объект **DataSet** является всем этим одновременно. Роль объекта **DataSet** в ADO.NET заключается в предоставлении отсоединенного хранилища информации, извлеченной из базы данных, и в обеспечении для .NET возможностей базы данных, хранимой в памяти. Объект DataSet — это простая коллекция структур данных, использующихся для организации отсоединенного хранилища информации.

Структура объекта DataSet

Объект **DataSet** состоит из нескольких связанных друг с другом структур данных. Концептуально он представляет собой полный набор реляционной информации. Внутри объекта **DataSet** могут хранится пять объектов.

- **DataTable** — прямоугольный набор данных, организованный в столбцы и строки. Этот объект является аналогом объекта ADO Recordset и объекта OLE DB RowSet.

Часть II

Класс DataSet

Глава 5. Создание объекта DataSet
Глава 6. Типизированные классы DataSet
Глава 7. Манипулирование объектом DataSet
Глава 8. Обновление базы данных

```
// Запрос к базе данных.
string query =
    "SELECT Customer.CustomerID, FirstName, LastName, " +
    "       HomePhone, Address, City, State, Zip, DOB, " +
    "       InvoiceNumber, InvoiceDate, FOB, PO, Terms " +
    "  FROM Customer " +
    "    JOIN Invoice on Customer.CustomerID = " +
    "         Invoice.CustomerID " +
    "  ORDER BY Customer.CustomerID, Invoice.InvoiceID ";
// Подключение к базе данных и выполнение запроса.
SqlConnection conn = new SqlConnection(
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "Integrated Security=true;");
conn.Open();

// Создание объекта команды для выполнения запроса.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = query;

// Создание объекта DataReader.
SqlDataReader rdr =
    cmd.ExecuteReader(CommandBehavior.KeyInfo |
    CommandBehavior.CloseConnection);

// Использование объекта DataReader для создания
// объектной модели нашего приложения.
customerList = new CustomerList(rdr);

// Закрытие объекта DataReader.
rdr.Close();

// Заполнение комбинированного списка.
cbCustomers.Items.AddRange(customerList.ToArray());

// Установка фокуса на первом элементе
// в комбинированном списке.
cbCustomers.SelectedIndex = 0;
```

Поскольку мы спроектировали достаточно неплохую объектную модель, ее реализация в приложении достигается путем создания экземпляра класса CustomerList, конструктору которого передается объект **DataReader.** Полный исходный код приложения можно загрузить с моего Web-узла по адресу: www.adoguy.com/book.

Резюме

Дочитав книгу до этой страницы, вы уже должны были проникнуться идеей "пожарного шланга" и понять всю ее мощь. Так как в ADO.NET однонаправленный курсор представлен отдельным типом объекта **(DataReader),** считывание данных с помощью "пожарного шланга" стало куда более простым занятием, чем в старых API доступа к данным. Кроме того, вы должны были узнать о способе использования объекта **DataReader** с целью обеспечения доступа к данным для ваших собственных классов. Во второй части этой книги мы воспользуемся полученными знаниями для того, чтобы рассмотреть, какую роль во всей этой схеме играет объект DataSet.

Класс InvoiceList

Класс InvoiceList (как и класс CustomerList) — это простая оболочка класса ArrayList, позволяющая иметь конструктор, которЬЕЙ принимает интерфейс IDataReader. Самая важная часть исходного кода данного класса — это часть, в которой реализуется добавление счета в список. Если посмотреть на запрос, который используется для извлечения информации из базы данных, можно заметить, что каждая строка результирующего набора содержит сведения как о клиенте, так и о его счете. Следует отметить, что эти же сведения применяются и при добавлении в базу данных счета определенного клиента. В самом начале исходного кода конструктора я "кэширую" идентификационный номер клиента (CustomerID). В цикле do ... while я добавляю все счета клиентов до тех пор, пока идентификационный номер клиента текущей строки совпадает с "кэшированным" идентификационным номером. Вы могли обратить внимание на то, что когда я заношу в "кэш" идентификационный номер клиента, я не делаю проверку на null. В этом нет необходимости, так как идентификационный номер клиента — это первичный ключ соответствующей таблицы. Если идентификационный номер клиента был бы равен null, это означало бы наличие неожиданной и неисправимой ошибки, требующей генерации исключения. Исходный код класса InvoiceList показан в листинге 4.16.

Листинг 4.16. Класс *InvoiceList*

```
public class InvoiceList : ArrayList
{

    public InvoiceList(IDataReader rdr)
    {
      // "Кэшируем" идентификатор текущего клиента.
      Guid customerlD = rdr.GetGuid(
      rdr.GetOrdinal("CustomerID"));

      // Добавляем счета до тех пор,
      // пока не изменится идентификатор клиента или
      // не будет достигнут конец объекта DataReader.
      do
      {
        Invoice inv = new Invoice(rdr);
        Add(inv);
      }
      while (rdr.Read() &&
        customerlD == rdr.GetGuid(
                              rdr.GetOrdinal("CustomerID")));

    }

}
```

Код Windows-формы

Написав запрос и разработав классы, использующиеся для представления данных, рассмотрим код, непосредственно применяющийся для создания объектной модели, как показано в листинге 4.17.

Листинг 4.17. Код Windows-формы

```
// Необходимо для поддержки проектировщика Windows-форм.
InitializeComponent();
```

```
        // Добавление клиентов в список,
        while (rdr.Read())
        {
          Customer cust = new Customer(rdr);
          Add(cust);
        }
    }
}
```

Класс Invoice

Класс Invoice **очень похож на** класс Customer. **Это простой** класс, в котором **содержится информация о счете клиента.** Для представления **списка счетов я решил применить элемент пользовательского интерфейса** Listview, **самый эффективный** способ **заполнения которого заключается в создании массива. Ответственность за создание массива я решил возложить на класс** Invoice, **разработав** метод ToStringArray(). **Исходный код класса** Invoice **приведен в листинге 4.15.**

Листинг 4.15. **Класс *Invoice***

```
public class Invoice
{

    public Invoice(IDataReader rdr)
    {

      // Если столбец не равен null, извлечь всю информацию
      // о счете клиента из объекта DataReader.
      InvoiceID     = Field.GetGuid(rdr,     "InvoiceID");
      InvoiceNumber = Field.GetInt32(rdr,    "InvoiceNumber");
      InvoiceDate   = Field.GetDateTime(rdr, "InvoiceDate");
      PO            = Field.GetString(rdr,   "PO");
      FOB           = Field.GetString(rdr,   "FOB");
      Terms         = Field.GetString(rdr,   "Terms");

    }

    // Члены класса - открытые на случай, если нам
    // понадобится получить к ним прямой доступ.
    public Guid        InvoiceID;
    public Int32       InvoiceNumber;
    public DateTime    InvoiceDate;
    public string      PO;
    public string      FOB;
    public string      Terms;

    public string[] ToStringArray()
    {
      // Создание массива строк, необходимого для более
      // эффективного заполнения объекта ListView.
      string[] values = new string[6];
      values[0] = InvoiceNumber.ToString();
      values[1] = InvoiceDate.ToString();
      values[2] = PO;
      values[3] = FOB;
      values[4] = Terms;
      return values;
    }
}
```

Листинг 4.13. Класс Customer

```
public class Customer
{

    public Customer(IDataReader rdr)
    {

      // Если столбец не равен null, извлечь всю информацию
      // о клиенте из объекта DataReader.
      CustomerID = Field.GetGuid(rdr,     "CustomerID");
      LastName =   Field.GetString(rdr,   "LastName");
      FirstName =  Field.GetString(rdr,   "FirstName");
      HomePhone =  Field.GetString(rdr,   "HomePhone");
      Address =    Field.GetString(rdr,   "Address");
      City =       Field.GetString(rdr,   "City");
      State =      Field.GetString(rdr,   "State");
      Zip =        Field.GetString(rdr,   "Zip");
      DOB =        Field.GetDateTime(rdr, "DOB");

      // Создать объект InvoiceList.
      Invoices = new InvoiceList(rdr);
    }
    // Члены класса.
    public Guid     CustomerlD;
    public string   LastName;
    public string   FirstName;
    public string   HomePhone;
    public string   Address;
    public string   City;
    public string   State;
    public string   Zip;
    public DateTime DOB;
    public InvoiceList Invoices;

    // Переопределение метода object.ToString() для обеспечения
    // вывода полного имени клиента в комбинированном списке.
    public override string ToString()
    {
      return LastName + ", " + FirstName;
    }
}
```

Класс CustomerList

Класс CustomerList — это тонкая "оболочка" класса ArrayList, необходимая для создания простого упорядоченного списка клиентов. Цель создания подобной оболочки состоит в том, чтобы иметь конструктор, принимающий в качестве аргумента объект DataReader. Исходный код класса CustomerList приведен в листинге 4.14.

Листинг 4.14. Класс *CustomerList*

```
// Оболочка класса ArrayList, поддерживающая создание
// объекта CustomerList на основе объекта DataReader.
public class CustomerList : ArrayList
{

    public CustomerList(IDataReader rdr)
    {
```

Доступ к базе данных

Получить всю информацию о клиентах и их счетах можно несколькими способами. Во-первых, можно извлечь из базы данных всю таблицу Customer и всю таблицу Invoice (в которой хранится информация о счетах), после чего придется провести сопоставление каждого клиента его счетам. Согласитесь, что такая сложность является излишней.

Для того чтобы упростить задачу, я написал запрос, который возвращает всю необходимую информацию в одном результирующем наборе данных. Этот запрос выбирает из базы данных все необходимые поля таблиц Customer и Invoice. Ниже приведена SELECT-часть такого запроса:

```
SELECT Customer.CustomerID, FirstName, LastName, HomePhone,
       Address, City, State, Zip, DOB,
       InvoiceNumber, InvoiceDate, FOB, PO, Terms
```

Обратите внимание, что единственным полем с полностью определенным именем является поле CustomerID, так как оно существует более чем в одной таблице. Так как мы запрашиваем информацию из нескольких таблиц, их необходимо объединить, Ниже приведена JOIN-часть запроса:

```
FROM Customer
  JOIN Invoice on Customer.CustomerID = Invoice.CustomerID
```

И, наконец, для того чтобы упорядочить извлеченные данные, нам необходимо добавить предложение ORDER BY (иначе счета клиентов могут "перемешаться"): ORDER BY Customer. CustomerlD, Invoice. InvoiceID. А вот как выглядит наш запрос целиком:

```
SELECT Customer.CustomerID, FirstName, LastName, HomePhone,
       Address, City, State, Zip, DOB,
       InvoiceNumber, InvoiceDate, FOB, PO, Terms
  FROM Customer
    JOIN Invoice on Customer.CustomerID = Invoice.CustomerID
  ORPER BY Customer.CustomerID, Invoice.InvoceID
```

Объекты данных

Одним из решений, принятых мною в самом начале разработки приложения, было создание классов для представления клиентов и счетов. Эти классы предназначены для простого хранения информации, извлеченной из базы данных. Поскольку для выполнения последней операции я планировал использовать объект DataReader, эти классы помогли бы мне создать копию данных в оперативной памяти и я мог бы закрыть соединение с базой данных сразу же после считывания всей нужной информации.

Наконец, мне нужно было, чтобы объекты данных создавались на основе объекта DataReader и не зависели от конкретного управляемого поставщика. Для этого я воспользовался конструкторами, принимающими в качестве аргумента интерфейс IDataReader и разрешающими при необходимости сменить поставщик данных.

Класс Customer

Начнем с класса Customer. В нем должны содержаться демографические сведения о клиенте, взятые из базы данных. Кроме того, здесь должен храниться список всех счетов клиента, представленный классом InvoiceList (см. листинг 4.16).

Когда я отлаживал этот класс, то понял, что для хранения информации о всех клиентах мне необходимо воспользоваться комбинированным списком. Чтобы изменить вид имени клиента, я переопределил метод ToString() класса Object так, чтобы он возвращал имя и фамилию клиента. Когда объект Customer добавляется в комбинированный список, метод ToString() автоматически вызывается для определения отображаемого имени клиента. Исходный код класса Customer показан в листинге 4.13.

судят о том, использует ли база данных SQL Server в качестве ключа таблицы глобальный уникальный идентификатор (GUID).

- `IsUnique`. Булево выражение, указывающее на уникальность хранящихся в столбце данных. Если значение столбца таблицы метаданных `IsKeyColumn` равно `true`, то значение столбца `IsUnique` также будет равно `true`.
- `IsKeyColumn`. Булево выражение. Значение `true` указывает на то, что столбец является либо первичным ключом таблицы, либо его частью.
- `IsAutoIncrement`. Булево выражение, указывающее на автоматическое увеличение базой данных значения столбца на определенную величину.
- `BaseSchemaName`. Имя схемы в базе данных, содержащей таблицу столбца. `null`, если это невозможно определить.
- `BaseCatalogName`. Имя каталога в базе данных, содержащей таблицу столбца. `null`, если это невозможно определить.
- `BaseTableName`. **Имя таблицы базы данных, в которой содержится** столбец. `null`, **если это невозможно определить.**
- `BaseColumnName`. Имя столбца в таблице базы данных. Не зависит от имени столбца в объекте **Command.**

Если вам необходимо получить более подробные сведения об информационной схеме базы данных, обратитесь к разделу "Получение информационной схемы базы данных с помощью поставщика OLE DB" главы 2.

Создание простого приложения, взаимодействующего с базой данных

В этом разделе мы постараемся объединить весь рассмотренный ранее материал в простом приложении Windows Forms, позволяющем просматривать счета клиентов (диалоговое окно приложения показано на рис. 4.2).

При желании вы можете загрузить исходный код этого приложения с моего Web-узла по адресу: `www.adoguy.com/book`. В следующих разделах данной главы остановимся на рассмотрении способа использования в приложении объекта **DataRcadcr.**

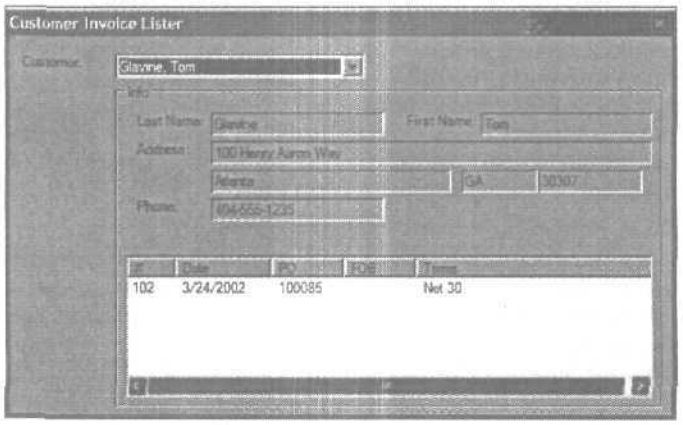


Рис. 4.2. *Простое приложение Windows* ***Forms***

```
      LastName :
        Type:         System.String
        IsUnique:     False
        AllowDBNull: False

...

Result Set
    Column:
      InvoiceID :
        Type:         System.Guid
        IsUnique:     False
        AllowDBNull: False
    Column:
      InvoiceNumber :
        Type:         System.Int32
        IsUnique:     False
        AllowDBNull: False
    Column:
      InvoiceDate :
        Type:         System.DateTime
        IsUnique:     False
        AllowDBNull: False
```

В столбцах схемы хранится различная информация о возвращенных данных. В следующем списке перечислены все столбцы таблицы метаданных результирующего набора.

- ColumnName. Имя столбца, определенное поставщиком. Если имя столбца указано в объекте Command, используется это имя. С конкретной реализацией метода GetSchemaTable() не связано никаких гарантий того, что имя столбца будет уникальным или что оно не будет равно null.
- ColumnOrdinal. Порядковый номер столбца. Отсчет порядковых номеров столбцов начинается с единицы.
- ColumnSize. Если тип данных имеет фиксированную длину, то здесь содержится максимальный размер столбца таблицы.
- NumericPrecision. Максимальная точность числа (если в столбце хранятся числовые данные). Если в столбце хранятся не числовые данные, то здесь содержится значение null.
- NumericScale. Количество знаков после запятой (если в столбце хранятся числовые данные). Если в столбце хранятся не числовые данные или числовые данные, которые не поддерживают дробные значения, то здесь содержится null.
- DataType, **CLR-тип столбца.**
- DbType. Тип столбца, специфичный для поставщика данных.
- IsLong. Булево выражение, определяющее возможность хранения в столбце длинных двоичных данных (например, BLOB).
- AllowDBNull. Булево выражение, определяющее возможность хранения в столбце значения null.
- IsReadOnly. Булево выражение, определяющее возможность изменения значения столбца.
- IsRowVersion. Булево выражение. Значение true указывает на то, что в столбце хранится идентификатор строки. Как правило, по значению этого выражения

метод GetSchemaTable(), который возвращает объект **DataTable,** содержащий информацию о типах данных результирующего набора. Более подробно объект **DataTable** рассматривается в главе 6. В листинге 4.12 приведен упрощенный пример извлечения информации о столбцах результирующего набора данных.

Листинг 4.12. Работа с **метаданными объекта** *DataReader*

```
...

// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM CUSTOMER\n" +
                  "SELECT * FROM INVOICE";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем результирующим наборам данных.
do
{

    // Получение метаданных результирующего набора
    // в виде объекта DataTable.
    DataTable schema = rdr.GetSchemaTable();

    // Вывод заголовка 'Result Set'.
    Console.WriteLine("ResultSet");

    // Вывод информационной схемы.
    foreach (DataRow row in schema.Rows)
    {
      Console.WriteLine("  Column:");
      Console.WriteLine("    {0} : ", row["ColumnName"]);
      Console.WriteLine("      Type:         {0}",
      row["DataType"]);
      Console.WriteLine("      IsUnique:     {0}",
      row["IsUnique"]);
      Console.WriteLine("      AllowDBNull: {0}",
      row["AllowDBNull"]);
    }

}
while (rdr.NextResult());
```

Если выполнить этот код, на консоль будут выведены метаданные таблиц клиентов и счетов. Ниже приведен отрывок из них:

```
Result Set
    Column:
      CustomerID :
        Type:         System.Guid
        IsUnique:     False
        AllowDBNull:: False
    Column:
      FirstName :
        Type:         System.String
        IsUnique: .   False
        AllowDBNull:  False
    Column:
```

В большинстве остальных случаев имеет смысл использовать поведение команды CommandBehavior.KeyInfo. Блокировка базы данных без необходимости приводит к недоступности строк для других процессов, работающих с базой данных, и может стать причиной снижения производительности.

Что такое результирующие наборы данных ?

В ответ на запрос база данных формирует так называемый результирующий набор данных. Если в запросе требовалось возвратить только одну порцию информации (т.е. единичный запрос), база данных сформирует один результирующий набор. В то же время, при вызове хранимой процедуры или выполнении пакетного оператора база данных может сформировать несколько результирующих наборов. Например, при выполнении следующих операторов будет возвращено два результирующих набора данных:

```
SELECT * FROM CUSTOMER
SELECT * FROM INVOICE
```

Обработка множественных результирующих наборов данных

Извлечение множества результирующих наборов данных из одного объекта DataReader — вполне обыденное занятие. Для итерации по этому множеству используется метод объекта DataReader NextResult(). В отличие от метода Read(), он не должен вызываться перед началом считывания данных. Пример использования метода NextResult () приведен в листинге 4.11.

Листинг 4.11. Использование объекта *DataReader* для возврата множественных результирующих наборов данных

```
...

// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM CUSTOMER\n" +
                  "SELECT * FROM INVOICE";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем результирующим наборам данных.
do
{
    // Итерация по всем записям результирующего набора.
    while (rdr.Read())
    {
      Console.WriteLine(rdr[0]);
    }
}
while (rdr.NextResult());
```

Как видно из этого примера, метод NextResult() вызывается только после того, как мы закончили обработку первого результирующего набора данных.

Работа с метаданными объекта DataReader

В большинстве случаев нам известен тип информации, извлекаемой из базы данных, но так бывает не всегда. К счастью, объект DataReader предоставляет сведения о типах данных для каждого результирующего набора. Все DataReader-классы имеют

Этот вспомогательный класс предлагает более простой метод определения приемлемого значения для null-полей. Используя класс Field, я могу сделать код более удобочитаемым. Листинг 4.9 — яркий этому пример (для сравнения см. листинг 4.7).

Листинг 4.9. **Использование класса *Field***

```
public Invoice(IDataReader rdr)
{
    // Если столбец не содержит значение null, извлечь
    // хранящуюся в нем информацию.
    InvoiceID = Field.GetGuid(rdr, "InvoiceID");
    InvoiceNumber = Field.GetInt(rdr, "InvoiceNumber");
    InvoiceDate = Field.GetDateTime(rdr, "InvoiceDate");
    Terms = Field.GetString(rdr, "Terms");
    PO = Field.GetString(rdr, "PO");
    FOB = Field.GetString(rdr, "FOB");
}
```

Существуют единичные случаи, когда поле не может содержать значения null. Типичными примерами являются первичные ключи и обязательные поля. Фактически для определения подобных полей мы могли бы воспользоваться информационной схемой базы данных. Если информационная схема показывает, что поле является обязательным, проверка `IsDbNull()` — это простая трата времени. Если поле обязательно, однако ADO.NET сообщает, что оно является null-полем, генерируется исключение. Следует отметить, что в данном случае это весьма кстати, поскольку после возникновения такой неожиданной ситуации приложение вряд ли может продолжать нормально свою работу. В любом случае если вы решите использовать мой класс Field, то можете обойти это требование, рискуя встретить null-строку в совершенно неожиданном месте.

Блокировка базы данных

Каждая современная многопользовательская база данных поддерживает блокировку. Блокировка записей — это способ сообщить базе данных о том, что считываемая пользователем информация может быть вскоре изменена для предотвращения модификации последней другими системами или пользователями. Объект ADO.NET **DataReader** не поддерживает сложную блокировку базы данных. Блокировку считываемых записей можно произвести при вызове метода `IDbCommand.ExecuteReader()`. В этом случае блокировка записей осуществляется по умолчанию, как показано в листинге 4.10.

Листинг 4.10. Использование **объекта *DataReader* с блокировкой записей базы** данных

```
// Записи базы данных блокируются.
SqlDataReader rdr = cmd.ExecuteReader();

// Блокировки записей базы данных не происходит.
SqlDataReader rdr =
                cmd.ExecuteReader(CommandBehavior.KeyInfo);
```

Блокировать записи при работе с объектом **DataReader** нужно по двум причинам.

- Записи будут модифицироваться, и вы не хотите, чтобы другие системы или пользователи изменяли их, пока вы не закончите внесение собственных изменений.
- Вы не хотите, чтобы изменения вносились в базу данных во время анализа считанной вами информации. При этом обычно речь идет о критической необходимости в согласованности данных во время проведения статистического анализа или составления важных отчетов.

```
    }

    if (!rdr.IsDBNull(rdr.GetOrdinal("InvoiceNumber")))
    {
      InvoiceNumber = rdr.GetString(
      rdr.GetOrdinal("InvoiceNumber"));
    }

    if (!rdr.IsDBNull(rdr.GetOrdinal("InvoiceDate")))
    {
      InvoiceDate = rdr.GetString(
      rdr.GetOrdinal("InvoiceDate"));
    }

    if (!rdr.IsDBNull(rdr.GetOrdinal("Terms")))
    {
      Terras = rdr.GetString(rdr.GetOrdinal("Terms"));
    }

    if (!rdr.IsDBNull(rdr.GetOrdinal("PO")))
    {
      PO = rdr.GetString(rdr.GetOrdinal("PO"));
    }

    if (!rdr.IsDBNull(rdr.GetOrdinal("FOB")))
    {
      FOB = rdr.GetString(rdr.GetOrdinal("FOB"));
    }
}
```

Достаточно много работы для простого копирования значений из шести полей! Необходимо предложить более эффективное решение. Вы можете загрузить класс **Field** с моего Web-узла, расположенного по адресу: www.adoguy.com/book. В листинге 4.8 приведена часть его (класса Field) исходного кода.

Листинг 4.8. **Класс** *Field*

```
public class Field
{

    ...

    // Доступ к полю осуществляется по его имени.
    static public string GetString(IDataRecord rec,
                                   string fldname)
    {
      return GetString(rec, rec.GetOrdinal(fldname));
    }

    static public decimal GetDecimal(IDataRecord rec,
                                     string fldname)
    {
      return GetDecimal(rec, rec.GetOrdinal(fldname));
    }
    ...

}
```

помогает более эффективно работать с запросами большого количества двоичных данных, когда реально работа ведется только с их частью.

- `SingleResult`. В результате выполнения запроса ожидается возвращение только одного набора данных.
- `SingleRow`. В результате выполнения запроса ожидается возвращение только одной строки.

Работа с Null-столбцами

Обеспечивающие безопасность типов get-методы объекта **DataReader** (например, GetString, `GetBoolean` и т.д.), удобно использовать при извлечении значений столбцов из объекта **DataReader. В** идеале их существование могло бы полностью удовлетворить разработчика. К сожалению, при считывании данных из столбца нет никаких гарантий того, что они там вообще существуют. Начинающих разработчиков это может сбить с толку. Например, если в таблице клиентов есть запись о клиенте, который не сообщил свой адрес, то это поле может иметь значение null. На самом деле содержимое этого поля полностью зависит от разработчика базы данных. В том случае, если он решит вообще не заполнять это поле в записи клиента (вместо того, чтобы заполнить его пустой строкой), оно будет иметь значение null.

Когда вы совершаете итерацию по записям в объекте **DataReader,** при попытке извлечь значение из null-поля (а точнее, при попытке преобразования null-поля в требуемый тип данных) ADO.NET сгенерирует исключение. Чтобы избежать подобной ситуации, объект **DataReader** имеет метод `IsDBNull()`, предназначенный для индикации null-поля. Пример использования метода `IsDBNull()` показан в листинге 4.6.

Листинг 4.6. Работа с *null*-столбцами

```
// cmd - это экземпляр класса SqlCommand.
SqlDataReader rdr = cmd.ExecuteReader();

while (rdr.Read())
{
    // Вывод на консоль значения поля, если оно не равно null.
    if (!rdr.IsDBNull(0))
    {
      // Вывод на консоль первого поля строки.
      Console.WriteLine(rdr.GetString(0));
    }
}
```

Несмотря на то что этот метод работает, в большинстве приложений он приводит к созданию огромного и некрасивого кода. Взгляните на листинг 4.7, представляющий собой пример типичного копирования содержимого объекта **DataReader** в некоторое локальное хранилище.

Листинг 4.7. Множественная проверка на *null*-значения в объекте DataReader

```
public Invoice(IDataReader rdr)
{

    // Если столбец не содержит значение null, извлечь
    // хранящуюся Ё нем информацию.
    if (!rdr.IsDBNull(rdr.GetOrdinal("InvoiceID")))
    {
      InvoiceID = rdr.GetString(rdr.GetOrdinal("InvoiceID"));
```

```
// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем строкам.
while (rdr.Read())
{
    // Функциональность этих строк кода идентична.
    Console.WriteLine(rdr.GetDateTime(
                          rdr.GetOrdinal("InvoiceDate")));
    Console.WriteLine((DateTime) rdr["InvoiceDate"]);
}
```

Методы GetOrdinal O и GetName() объекта DataReader в некотором смысле дополняют друг друга, позволяя узнать порядковый номер и имя столбца, соответственно.

Параметры метода ExecuteReader

В большинстве случаев вызов метода ExecuteReader объекта Command без параметров является оптимальным решением. Тем не менее в более сложных сценариях метод ExecuteReader может быть вызван с флагом CommandBehavior, который изменяет способ создания объекта DataReader и влияет на содержащиеся в нем типы данных. Флаг CommandBehavior может принимать одно или более из перечисленных ниже значений (поддерживается их побитовая комбинация).

- Default. С данным значением не ассоциировано никакое специальное поведение. Следовательно, команда будет выполнена на сервере без установки/снятия блокировки строк; соединение при этом также не подвергается какому-либо воздействию. Значение Default используется по умолчанию при вызове метода ExecuteReader () без указания флага CommandBehavior.
- CloseConnection. Когда объект DataReader достигает конца всех результирующих наборов данных или попросту уничтожается, уничтожается также и ассоциированный с ним объект Connection.
- KeyInfo. Запрошенная информация возвращается без блокировки каких-либо строк. Данное значение обычно применяется только при чтении, Предполагается, что результирующий набор данных, возвращенный объектом DataReader, не будет обновлен. Это очень важно, так как если не заблокировать строки во время работы с объектом DataReader, то любые из них (еще не извлеченные) могут быть изменены, что способно привести к весьма неожиданным результатам. К тому же управляемый поставщик SQL Server рассматривает значение Keylnfo как указание поместить предложение FOR BROWSE в выполняемую команду. При работе с SQL Server такое поведение может привести к различным побочным эффектам. Более подробно эта тема рассматривается в документации no SQL Server.[2]
- SchemaOnly. В объекте DataReader возвращается информационная схема базы данных, однако запрос при этом не выполняется и в базу данных не вносятся никакие изменения.
- SequentialAccess. Строки считываются последовательно на уровне столбцов, что удобно при использовании методов GetChars или GetBytes для извлечения из строк объекта DataReader длинных двоичных данных. Если не использовать это значение, то последние будут считаны в объект DataReader вне зависимости от того, будут они использованы или нет. Таким образом, это значение

[2] См. msdn.microsoft.com/library/en-us/tsqlref/ts_sa-ses_9sfo.asp#_from_clause.

Поскольку индексатор возвращает объект, его можно привести к типу столбца или вызвать для этого один из get-методов объекта DataReader, обеспечивающих безопасность типов, как показано в листинге 4.3.

Листинг 4.3. Использование get-метода объекта *DataReader*, обеспечивающего безопасность типов

```
// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем строкам.
while (rdr.Read())
{
    Console.WriteLine( (string) rdr[0]);      // Эти две строки кода
    Console.WriteLine(rdr.GetString(0));      // имеют одинаковую
                                              // функциональность.

}
```

Обеспечивающие безопасность типов get-методы не производят никаких преобразований — они просто осуществляют приведение типа. Если попытаться извлечь столбец, тип которого не соответствует типу, к которому осуществляется приведение, метод сгенерирует исключение `InvalidCastException` — такое же, которое бы возникло в результате выполнения некорректной операции приведения типа. Если бы в предыдущем примере первый столбец имел целочисленный тип, было бы сгенерировано исключение InvalidCastException. В тон случае, когда вам нужно провести именно преобразование типа, воспользуйтесь классом `System.Convert`, как показано в листинге 4.4.

Листинг 4.4. Использование класса *System.Convert*

```
...
cmd.CommandText = "SELECT * FROM PRODUCT";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем строкам.
while (rdr.Read())

{
    Console.WriteLine(Convert.ToString(rdr["Price"]));
}
```

Все get-методы, обеспечивающие безопасность типов, требуют передачи в качестве параметра порядкового номера столбца, что, естественно, менее удобно, чем использование его имени. Альтернативным способом получения порядкового номера столбца является применение метода GetOrdinal(), как показано в листинге 4.5.

Листинг 4.5. Использование метода *GetOrdinal*

```
// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM INVOICE";
```

```
SqlCommand cmd - new SqlCommand();
...

// Все правильно.
SqlDataReader rdr = cmd.ExecuteReader();

// Неверный код (не будет компилироваться).
SqlDataReader rdr = new SqlDataReader();
```

Функциональность объекта DataReader

Постараемся разобраться в функциональности, заложенной в объект **DataReader. В** главе 1 говорилось о возможности работать с данными в отсоединенном режиме, однако это совершенно не применимо к объекту **DataReader.** На всем протяжении его жизни (при чтении или даже просто при просмотре строк) соединение должно оставаться открытым. Действительно, объект **DataReader** очень напоминает своих предшественников, предполагавших обработку данных в подсоединенном режиме.

Так как при работе с объектом **DataReader** соединение остается открытым, он может не извлекать за один раз все данные, полученные в результате выполнения запроса. В контексте проектов, ориентированных на работу со складом данных или большой корпоративной базой данных, это имеет решающее значение. Действительно, ведь требование вернуть за один раз миллион записей, которые должны быть размещены в памяти, — это сущее безумие. Объект **DataReader** позволяет оптимальным способом решить эту проблему, загружая в память за один раз небольшую часть результирующего набора данных.

Поскольку объект **DataReader** предполагает работу с данными в подсоединенном режиме, некоторые разработчики не решаются его использовать, отдавая предпочтение объекту DataSet. Более подробно выбор между использованием объектов **DataReader и DataSet** обсуждается в главе 7.

Доступ к данным с помощью объекта DataReader

Как было показано в листинге 4.1, объект **DataReader** позволяет осуществлять доступ к отдельным столбцам, представляя результирующий набор данных в виде массива. Посредством соответствующего индексатора ([]) указывается либо порядковый номер столбца, либо его имя (листинг 4.2).

Листинг 4.2. Доступ к столбцам результирующего набора данных с помощью объекта *DataReader*

```
...

// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText - "SELECT * FROM CUSTOMER";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем строкам.
while (rdr.Read())

{
    Console.WriteLine(rdr[0]);              // Эти две строки кода
    Console.WriteLine(rdr["CustomerID"]); // имеют одинаковую
                                          // функциональность.
}
```

```
WHERE Customer.LastName = 'Maddux'
ORDER BY InvoiceDate DESC
```

Во всех приведенных выше примерах используется синтаксис **SQL-92. Как** правило, все разработчики баз данных (включая Microsoft, IBM и Oracle) следуют этому синтаксису, несмотря на то что каждый из них создал собственное расширение языка SQL (например, T-SQL для SQL Server и PL/SQL для Oracle). Вообще говоря, написание универсального SQL-кода — очень сложное занятие. Если вам крайне необходимо написать именно такой код, будьте готовы к снижению производительности вашей системы, поскольку каждый разработчик баз данных реализует **механизм** выполнения SQL-операторов по-своему.

Объект **DataReader**

После того как вы научились создавать запросы, пришло время научиться их выполнять, используя для этого средства ADO.NET. В отличие от большинства других интерфейсов доступа к данным, в которых для считывания и изменения данных используется один и тот же объект, в ADO.NET операции чтения и изменения данных разделены. В частности, большая часть вашего кода будет просто считывать информацию из базы данных и представлять ее пользователю, используя для этого основной объект **ADO.NET,** предназначенный исключительно для чтения данных, — объект **DataReader.**

Объект **DataReader** обеспечивает доступ к данным, извлеченным в результате выполнения SQL-запроса, в режиме только для чтения. **Объект DataReader** представлен соответствующим классом — OleDbDataReader, SqlDataReader, OracelDataReader и OdbcDataReader — в каждом из управляемых поставщиков. Пример использования класса SqlDataReader приведен в листинге 4.1.

Листинг 4.1. **Использование класса *SqlDataReader***

```
// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Создание объекта DataReader в результате выполнения запроса.
SqlDataReader rdr = cmd.ExecuteReader();

// Итерация по всем строкам.
while (rdr.Read())
{
    Console.WriteLine(rdr[0]);
}
```

В этом примере приведен наиболее простой способ использования объекта **DataReader.** Для перемещения по результирующему набору данных предназначен **метод** `Read()`. Обратите внимание, что метод `Read` О обязательно должен быть вызван один раз еще до считывания первой записи. Благодаря этому объект **DataReader** более устойчив к ошибкам, так как перемещение строкового курсора и проверка того, достигнут ли конец результирующего набора данных, производится в одном операторе. По достижении конца результирующего набора данных метод `Read()` возвратит `false`.

Создание объекта DataReader

Объект **DataReader** не может быть создан напрямую из клиентского кода — он создается объектом команды во время ее выполнения с помощью метода `IDbCommand.ExecuteReader()`. Следует отметить, что конструктор `DataReader`-класса скрыт, для того чтобы предотвратить создание нового объекта **DataReader:**

```
SELSECT FirstName, LastName, HomePhone FROM Customer
  WHERE State = 'MA'
```

Еще одним из часто использующихся предложений является JOIN, позволяющее извлекать связанную информацию из разных таблиц. Так, предложение JOIN позволит нам получить не только сведения о клиенте, но и информацию о каждом соответствующем ему счете. Ниже приведен простой пример использования предложения JOIN:

```
SELECT FirstName, LastName, HomePhone, InvoiceNumber FROM
Customer
  JOIN Invoice
    ON Customer.CustomerID = Invoice.CustomerID
```

В этом примере таблицы Customer и Invoice объединяются с помощью общего для обоих поля CustomerlD. При желании можно добавить еще одно предложение JOIN для объединения таблиц Invoice и InvoiceItem:

```
SELECT FirstName, LastName, HomePhone,
       InvoiceNumber, Quantity, ProductID
  FROM Customer
    JOIN Invoice
      ON Customer.CustomerID = Invoice.CustomerID
    JOIN Invoiceltem
      ON Invoice.InvoiceID = InvoiceItem.InvoiceID
```

Для того чтобы ограничить поиск только счетами клиентов с фамилией Maddux, добавим предложение WHERE:

```
SELECT FirstName, LastName, HomePhone,
       InvoiceNumber, Quantity, ProductID
  FROM Customer
    JOIN Invoice
      ON Customer.CustomerID - Invoice.CustomerID
    JOIN Invoiceltem
      ON Invoice.InvoiceID = InvoiceItem.InvoiceID
  WHERE Customer.LastName = 'Maddux'
```

И, наконец, применим предложение ORDER BY для сортировки возвращаемого результата по дате оформления счета. Для этого добавим ORDER BY в коней оператора SELECT и укажем столбец, по которому будет производиться сортировка, например:

```
SELECT FirstName, LastName, HomePhone,
       InvoiceNumber, Quantity, ProductID
  FROM Customer
    JOIN Invoice
      ON Customer.CustomerID - Invoice.CustomerID
    JOIN InvoiceItem
      ON Invoice.InvoiceID = InvoiceItem.InvoiceID
  WHERE Customer.LastName = 'Maddux'
  ORDER BY InvoiceDate
```

Если необходимо сортировать по убыванию, а не по возрастанию (стандартный порядок сортировки), добавьте в конец оператора ключевое слово DESC:

```
SELECT FirstName, LastName, HomePhone,
       InvoiceNumber, Quantity, ProductID
  FROM Customer
    JOIN Invoice
      ON Customer.CustomerID = Invoice.CustomerID
    JOIN Invoiceltem
      ON Invoice.InvoiceID = InvoiceItem.InvoiceID
```

о языке SQL, обратитесь к Web-узлу www.sql.org. Кроме того, стоит обратить внимание на замечательные книги Джо Келко (Joe Celko): *Instant SQL Programming* (ISBN: 1874416509) и, для более опытных SQL-программистов, *SQL for Smarties: Advanced SQL Programming* (ISBN: 1558605762).

Чтобы считывать данные с помощью SQL-запроса, необходимо досконально изучить возможности оператора SELECT, который используется для извлечения из базы данных набора информации. Предположим, что у нас есть четыре таблицы, показанные на рис. 4,1 (все они взяты из базы данных, использующейся в примерах этой книги — см. Web-узел **ADOGuy** по адресу: www.adoguy.com/book).

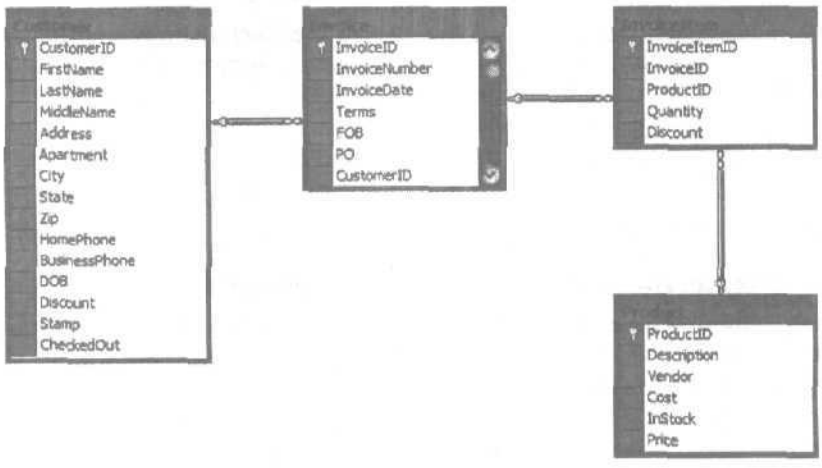


Рис. 4.1. Таблицы, использующиеся в примерах этой книги

Ниже приведен пример простейшего запроса к базе данных:

```
SELECT * FROM Customer
```

В результате выполнения этого запроса из базы данных извлекаются все строки таблицы Customer. Символ звездочки (*) — это маска, которая указывает базе данных на необходимость извлечения всех столбцов таблицы. Предложение FROM определяет таблицу базы данных, из которой нужно считать информацию; в нашем случае это таблица Customer.

В большинстве случаев нам не нужны все строки таблицы. Одним из распространенных требований к извлекаемым из таблицы данным является их сортировка в определенном порядке. В этом случае символ звездочки заменяется именами столбцов, которые необходимо извлечь. В следующем примере база данных возвратит столбцы FirstName, LastName и Phone таблицы Customer:

```
SELECT FirstName, LastName, Phone FROM Customer
```

SQL поддерживает функции агрегирования, такие как COUNT (используется для подсчета записей, которые были бы возвращены в результате выполнения запроса) и MAX (используется для получения максимального значения определенного поля):

```
SELECT COUNT(*) FROM Customer
SELECT MAX(Price) FROM Product
```

Кроме того, часто из таблицы необходимо извлечь только подмножество данных. Для этого используется предложение WHERE. Например, в результате выполнения следующего запроса из базы данных будут извлечены строки таблицы Customer, содержащие информацию о клиентах, проживающих в штате Массачусетс:

Глава 4

Получение данных

В этой главе...

Чтение данных

Объект DataReader

Создание простого приложения, взаимодействующего с базой данных

Моя первая работа в качестве программиста заключалась в написании отчетов для давно забытой реляционной базы данных FMS-80 в системе CP/M[1]. В те дни написание отчетов состояло в простом считывании информации из базы данных и выводе на консоль множества строк текста. Несмотря на то что в современных системах написание отчетов для баз данных стало гораздо сложнее, в корне так ничего и не изменилось — сначала данные откуда-то считываются, а затем куда-то выводятся.

Одним из первых правил, которые я использовал при программировании доступа к базам данных, являлось правило "пожарного шланга", применявшееся для уменьшения нагрузки на СУБД. *"Пожарный шланг" (firehose)* — это термин, который обозначал запрос у базы данных однонаправленного курсора. Сообщив базе данных о своих намерениях, можно ускорить процесс чтения данных и уменьшить объем использования ресурсов. В ADO.NET концепция однонаправленного курсора представлена объектом DataReader. Именно о нем и пойдет речь в этой главе.

Чтение данных

Чтение данных и составление отчетов — наиболее часто встречающиеся задачи разработчика приложений, основанных на взаимодействии с базой данных. Вне зависимости от того, предназначена ваша программа для просмотра котировок акций, выдачи списка пациентов или составления отчета о месячных продажах, чтение данных и составление отчетов должны быть наиболее простыми ее функциями. К сожалению, самой трудной частью процесса чтения информации из базы данных является создание соответствующего запроса.

Язык структурированных запросов (Structured Query Language — SQL) — это язык баз данных. SQL-86, SQL-92 и SQL-99 — стандарты языка SQL, разработанные национальным институтом стандартизации США (American National Standards Institute — ANSI) и использующиеся большинством производителей баз данных. К тому же язык SQL используется для извлечения информации из всех управляемых поставщиков Microsoft. Приведенные в этой книге фрагменты SQL-кода основаны на стандарте SQL-92,

Краткий обзор SQL-оператора SELECT

Язык SQL — слишком обширная тема для того, чтобы рассматривать ее на страницах этой книги, тем не менее я попытаюсь кратко рассказать вам об основах построения запросов к базе данных. Для того чтобы получить более подробную информацию

[1] Вот видите **какой я** старый и мудрый! **Для того** чтобы **познакомиться с системой CP/M,** обратитесь по адресу: `jfrace.sourceforge.net/appletCPM.html`.

```
    // Перебор всех строк заданного результирующего набора.
    while (rdr.Read())
    {
      Console.WriteLine(rdr[0]);
    }
// После просмотра первого результирующего набора
// данных переходим к следующему результирующему набору.
} while (rdr.NextResult());
```

Резюме

Несмотря на то что ADO.NET ориентирована на работу с данными в отсоединенном режиме, иногда нам все же нужно подключаться к базе данных. Приведенная в этой главе информация будет использоваться в *каждом* написанном вами приложении — способ, позволяющий избежать выполнения команд еще не придуман. В последующих главах этой книги мы сконцентрируемся на обработке полученных данных.

В данном примере для выполнения пакетного запроса был использован метод ExecuteNonQuery(). Хотя указанный метод эффективен, он, к сожалению, не обеспечивает обработку ошибок. Для разрешения этой проблемы вам нужно собственноручно добавить SQL-код с проверкой ошибок и (необязательной) поддержкой транзакций:

```
BEGIN TRAN

  INSERT INTO Customer
    (CustomerID, LastName, FirstName, Phone, Zip)
  VALUES (newid(), 'Smoltz', 'John', '503-432-4565',
    '12345');

  IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRAN
    RETURN
  END

  INSERT INTO Invoice
    (InvoiceID, InvoiceNumber, InvoiceDate, Terms)
  VALUES (newid(), '123456', '05/01/2001', 'Net 20');

  IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRAN
    RETURN
  END

COMMIT TRAN
```

Теперь наш пакетный запрос будет выполнять две задачи: осуществлять вставку новой информации в базу данных и проводить проверку на наличие ошибок. Если во время выполнения транзакции произошел сбой, она будет отклонена, в противном случае — принята. Пакетный запрос — это не что иное, как несколько запросов, объединенных в одну команду. Поскольку ADO.NET поддерживает множественные результирующие наборы данных, нет никакой серьезной причины не использовать пакетные запросы (листинг 3.19).

Листинг 3.19. Получение множественных результирующих наборов данных с помощью объекта *DataReader*

```
...

// Создание команды, включающей в себя два запроса SELECT
// и возвращающей два результирующих набора данных.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM Customer;" +
                  " SELECT * FROM Invoice;";

// Создание объекта DataReader для навигации по
// результирующим наборам данных.
SqlDataReader rdr = cmd.ExecuteReader();

// Перебор всех строк в каждом результирующем
// наборе данных.
do
{
```

```
INSERT INTO Invoice
  (InvoiceID, InvoiceNumber, InvoiceDate, Terms)
VALUES (newid(), '123456', '05/01/2001', 'Net 20');
```

В листинге 3.17 приведен ADO.NET-код, в котором используются обычные (а не пакетные) запросы.

Листинг 3.17. Использование обычных запросов

```
string sQuery1 = "INSERT INTO Customer " +
                 "(CustomerID, LastName, FirstName, Phone, Zip) " +
                 "VALUES (newid(), 'Smoltz', 'John'," +
                 "'503-432-4565', '12345');";

string sQuery2 = "INSERT INTO Customer " +
                 " (CustomerlD, LastName, FirstName, Phone, Zip) " +
                 "VALUES (newid(), 'Maddux', 'Greg'," +
                 "'503-432-4566', '12345');";

// Подключение к базе данных.
SqlConnection sqlconn = new SqlConnection();

//...

SqlCommand sqlcmd = sqlconn.CreateCommand();
sqlcmd.CommandType = CommandType.Text;

// Первый запрос.
sqlcmd.CommandText = sQuery1;
sqlcmd.ExecuteNonQuery();

// Второй запрос.
sqlcmd.CommandText = sQuery2;
sqlcmd.ExecuteNonQuery();
```

Несмотря на то что этот код будет выполняться, он неэффективен, так как требует трех обращений к базе данных. Его можно упростить, воспользовавшись пакетными запросами, как показано в листинге 3.18.

Листинг 3.18. Использование пакетных запросов

```
string sQuery = "INSERT INTO Customer " +
                "(CustomerID, LastName, FirstName, Phone, Zip) " +
                "VALUES (newid(), 'Smoltz', 'John'," +
                "'503-432-4565', '12345'); " +
                "INSERT INTO Customer " +
                "(CustomerID, LastName, FirstName, Phone, Zip) " +
                "VALUES (newid(), 'Maddux', 'Greg', " +
                "'503-432-4566', '12345');";

SqlConnection sqlconn = new SqlConnection();

//...

SqlCommand sqlcmd = sqlconn.CreateCommand();
sqlcmd.CommandType = CommandType.Text;

// Пакетный запрос.
sqlcmd.CommandText = sQuery;
sqlcmd.ExecuteNonQuery();
```

```
            "(InvoiceID, InvoiceNumber, InvoiceDate, Terms) " +
            "VALUES (newid(), '123456', '05/01/2001'," +
            "'Net 20');";
    cmd.ExecuteNonQuery();

    }
    catch(Exception)
    {
      // Отклонить транзакцию то точки сохранения -
      // момента внесения в базу данных информации о
      // новом клиенте.
      cmd.Transaction.Rollback("New Customer");
    }

    // Если выполнение было передано этому оператору,
    // принять транзакцию.
    cmd.Transaction.Commit();
}
catch(Exception ex)
{
    cmd.Transaction.Rollback( );
    Console.WriteLine("Command failed:" +
                      "{0}\nTransaction Rolled back", ex.Message);
}
```

В этом коде мы создали точку сохранения с именем "New Customer" (Новый клиент). Если занесение в базу данных информации о новом клиенте прошло успешно, однако во время создания нового счета произошла ошибка, нам не хотелось бы терять уже внесенные в базу данных сведения о новом клиенте. Поэтому мы отклоняем транзакцию вплоть до точки сохранения и затем принимаем ее.

Службы Enterprise Services и COM+

Иногда транзакции баз данных являются довольно масштабными, включающими в себя выполнение ряда операций на нескольких серверах. Для поддержки подобных транзакций в .NET предусмотрены службы Enterprise Services, которые являются .NET-оболочками для компонентов COM+. Следует отметить, что в этой книге службы Enterprise Services не описываются. Если вам необходима какая-либо дополнительная информация по программированию транзакций, советую обратиться к великолепной книге Тима Эвальда (Tim Ewald) *Transactional COM+* ***Programming*** **или к его статье "COM+ Integration: How .NET Enterprise Services Can Help You Build Distributed Applications", которую можно найти на Web-узле MSDN по адресу:** msdn.microsoft.com/msdnmag/issues/01/10/complus/complus.asp.

Пакетные запросы

При выполнении команд базы данных часто приходится проводить целую серию различных операций. В этом случае вы можете просто выполнять команды одну за другой, однако для этого потребуются многочисленные обращения к базе данных (по одному для каждой команды). С целью оптимизации подобных операций используются так называемые пакетные запросы. Предположим, что нам необходимо выполнить следующие действия:

```
INSERT INTO Customer
  (CustomerID, LastName, FirstName, Phone, Zip)
VALUES (newid(), 'Smoltz', 'John', '503-432-4565', '12345');
```

- **RepeatableRead.** Транзакция **не может считывать данные, которыми манипулируют другие незавершенные транзакции. Иными словами, данные, которыми манипулирует транзакция, на время ее работы блокируются.**
- **Serializable. Транзакция полностью изолирована от других транзакций.**

Несмотря на то что перечисление IsolationLevel имеет атрибут FlagsAttribute, позволяющий **использовать комбинацию нескольких уровней изоляции, подобная функциональность не поддерживается в силу специфики элементов перечисления.** Использующийся **по умолчанию уровень изоляции** ReadCommitted, **конечно же, не может идеально подходить для каждого случая.**

Точки сохранения в транзакциях SQL Server

При внесении множественных изменений в базу данных SQL Server может возникнуть ситуация, когда предпочтительнее отклонить не всю транзакцию, а только ее часть. Текущие **версии SQL Server не поддерживают вложенные транзакции, однако в них реализована концепция точек сохранения, с помощью которой можно осуществить подобную функциональность. Точка сохранения — это средство SQL Server, позволяющее производить частичное отклонение транзакции. Другими словами, точка сохранения** представляет **собой "закладку", определяемую посредством вызова метода Save класса** SqlTransaction **в любом месте** транзакции, **в которое вы хотели бы вернуться в случае ее отклонения (без отмены всей транзакции). Пример создания точки сохранения и частичного отклонения транзакции приведен в листинге 3.16.**

Листинг 3.16. Частичное отклонение транзакции

```
// Открытие соединения.
SqlConnection conn = new SqlConnection(
                             "Server=localhost;" +
                             "Database=ADONET;" +
                             "Integrated Security=true;");
conn.Open();

// Создание команды.
SqlCommand cmd = conn.CreateCommand();

// Создание транзакции.
cmd.Transaction =
        conn.BeginTransaction(IsolationLevel.ReadCommitted);

try
{
  // Попытка добавить в базу данных сведения о новом клиенте.
  cmd.CommandText = "INSERT INTO Customer " +
            "(CustomerID, LastName, FirstName, Phone, Zip) " +
            "VALUES (newid(), 'Smoltz', 'John'," +
            "'503-432-4565', '12345');";
  cmd.ExecuteNonQuery();

  // Создание точки сохранения.
  cmd.Transaction.Save("New Customer");

  try
  {

  // Попытка добавить в базу данных сведения о новом счете.
  cmd.CommandText = "INSERT INTO Invoice " +
```

Функционально этот код ничем не отличается от SQL-кода, приведенного ранее в этом разделе. Недостаток использования клиентских транзакций заключается в том, что они требуют гораздо больше обращений к базе данных, чем обычный SQL-код. Именно поэтому предпочтение отдается серверным транзакциям (см. SQL-код перед листингом 3.14) как более эффективным по сравнению с клиентскими транзакциями. Использование последних (см. листинг 3.15) может быть оправдано необходимостью связывания транзакций базы данных с какой-то внешней системой. Например, если ваш код одновременно сохраняет изменения в базе данных и обновляет Web-службу, сбой в последней операции может заставить вас отменить транзакцию базы данных со стороны клиента.

Если необходимо обеспечить всего лишь поддержку транзакций, то, вероятно, стоит использовать серверные транзакции. С другой стороны, с целью абстрагирования от специфики реализации конкретного ядра базы данных все транзакции можно выполнять на стороне клиента. При этом следует помнить, что использование транзакций только на стороне клиента может существенно ухудшить производительность системы.

Как вы уже, наверняка, заметили, транзакция создается путем вызова метода BeginTransaction класса соединения. В результате вызова метода создается объект транзакции. После этого ни одна из команд, ассоциированных с данным соединением, не будет принята до вызова метода класса транзакции Commit. В случае возникновении ошибки для отмены всех команд, выполненных с момента создания транзакции, может быть вызван метод Rollback.

После начала выполнения транзакции ее нужно либо принять, либо отклонить. Если этого не сделать, то во время освобождения ресурсов транзакции она будет автоматически отклонена. Другими словами, транзакция автоматически отклоняется при закрытии соединения или освобождении занятых им ресурсов, а также при уничтожении объекта транзакции сборщиком мусора. Поскольку об этой особенности в документации ничего не говорится, на нее не стоит обращать внимание — в конце концов в следующей версии .NET все может происходить по-другому. Кроме того, уничтожение объекта (вызов метода Finalize) может произойти не сразу, что зависит от особенностей реализации сборщика мусора. Таким образом, если вы не хотите сохранять изменения в базе данных, немедленно отмените их.

Уровни изоляции

Перед началом выполнения транзакции необходимо определить ее уровень изоляции — способ блокировки базы данных во время работы транзакции. Ниже перечислены поддерживаемые уровни изоляции транзакций.

- Chaos. Транзакция не может перезаписать другие не принятые транзакции с большим уровнем изоляции, но может перезаписать изменения, внесенные без использования транзакций. Поскольку данные не блокируются, записи, внесенные во время работы транзакции, могут быть перезаписаны после ее принятия.
- `ReadCommitted`. Транзакция не может считать данные, которыми манипулирует другая транзакция. Этот уровень позволяет избежать недействительных результатов чтения, так как данные, которыми манипулирует транзакция, на время ее работы блокируются.
- `ReadUncommitted`. Транзакция может считывать данные, которыми манипулирует другая транзакция. Выбор этого уровня изоляции способен привести к недействительным результатам чтения и появлению фантомных строк, поскольку после чтения информации, внесенной в базу данных в результате выполнения той или иной транзакции, последняя может быть отменена.

```
  // Закрытие соединения.
  conn.Close();

  // Если значение переменной result равно 0, команда была
  // принята, и мы можем возвратить true.
  int result = (int) cmd.Parameters["@RETURN"].Value;
  return (result == 0);
}
```

ADO.NET обеспечивает такую же степень контроля над базой данных, какая достигается при использовании обычного кода SQL. В ADO.NET транзакция представляется специальным объектом (экземпляром класса SqlTransaction, OleDbTransaction или OdbcTransaction). В листинге 3.15 приведен пример, аналогичный представленному выше фрагменту SQL-кода.

Листинг 3.15. Использование транзакций

```
// Открытие соединения.
SqlConnection conn = new SqlConnection(
                             "Server=localhost;" +
                             "Database=ADONET;" +
                             "Integrated Security=true;");

conn.Open();

// Создание команды.
SqlCommand cmd = conn.CreateCommand();

// Создание транзакции.
cmd.Transaction = conn.BeginTransaction();

try
{
  // Попытка добавить в базу данных сведения о новом клиенте.
  cmd.CommandText = @"INSERT INTO Customer
                      (CustomerID, LastName, FirstName,
                       Phone, Zip)
                      VALUES (newid(),
                       'Smoltz', 'John', '503-432-4565',
                       '12345');";
  cmd.ExecuteNonQuery();

  // Попытка добавить в базу данных сведения о новом счете.
  cmd.CommandText = @"INSERT INTO Invoice
                      (InvoiceID, InvoiceNumber,
                      InvoiceDate, Terms)
                      VALUES (newid(), ' 123456',
                       '05/01/2001', 'Net 20');";
  cmd.ExecuteNonQuery();

  // Если выполнение программы было передано этому
  // оператору, подтверждаем транзакцию.
  cmd.Transaction.Commit();
}
catch(Exception ex)
{
  cmd.Transaction.Rollback();
  Console.WriteLine("Command failed: " +
                    ex.Message +
                    "\nTransaction Rolled back");
}
```

которые либо выполняются все вместе, либо не выполняются вообще. Обычно это делается с помощью хранимой процедуры или набора SQL-операторов. Например, в следующем фрагменте SQL-кода продемонстрировано использование транзакции для восстановления системы, оказавшейся в несогласованном состоянии.

```
BEGIN TRAN

  INSERT INTO Customer
    (CustomerID, LastName, FirstName, Phone, Zip)
  VALUES (newid(), 'Smoltz', 'John', '503-432-4565',
                                     '12345');
  IF @@ERROR <> 0
  BEGIN
    ROLLBACK TRAN
    RETURN @@ERROR
  END

  INSERT INTO Invoice
    (InvoiceID, InvoiceNumber, InvoiceDate, Terms)
  VALUES (newid(), '123456', '05/01/2001', 'Net 20');

  IF @@ERROR 0 0
  BEGIN
    ROLLBACK TRAN
    RETURN @@ERROR
  END

COMMIT TRAN
RETURN 0
```

В этом запросе осуществляется попытка добавить в базу данных сведения о новом клиенте и о новом счете. В случае возникновения проблемы (например, такой идентификационный номер клиента уже используется) весь набор команд будет отклонен (так, как если бы они вообще никогда не выполнялись). Если же все три команды выполнятся успешно, то транзакция будет принята, а внесенные в базу данных сведения зафиксированы. Для того чтобы убедиться в том, что транзакция была принята, следует проанализировать код возврата, как показано в листинге 3.14 (0 означает принятие транзакции).

Листинг 3.14. Вызов хранимой процедуры без использования транзакции

```
// Этот метод можно использовать для выполнения команды
// и возвращения сведений о том, была она принята или нет.
bool RunCommand (SqlConnection conn, string sqlStatement)
{
  // Создание команды.
  SqlCommand cmd = conn.CreateCommand();
  cmd.CommandText = sqlStatement;

  // Определение параметра для возвращаемого значения.
  cmd.Parameters.Add("@RETURN", DbType.Int32).Direction =
                                ParameterDirection.ReturnValue;
  // Открытие соединения.
  conn.Open();

  // Выполнение команды.
  cmd.ExecuteScalar();
```

```
cmd.CommandText * string.Format(@"SELECT * FROM Customer
                          WHERE CustomerID = {0}", custID);

// Создание команды с помощью объекта StringBuilder.
StringBuilder bldr = new StringBuilder();
bldr.Append("SELECT * FROM Customer WHERE CustomerlD = ");
bldr.Append(custID);
cmd.CommandText = bldr.ToString();
```

Обычно управляемые поставщики попросту принимают команду и формируют из нее строку для отправки серверу баз данных. Использование параметров может привести к нежелательному разрастанию программного кода и, возможно, даже к снижению производительности.

Так зачем же вообще использовать объекты параметров? В конце концов, мы можем с тем же успехом вызвать хранимую процедуру или параметризированный запрос с помощью кода SQL, как показано в листинге 3.13.

Листинг 3.13. Вызов хранимой процедуры без использования **объектов параметров**

```
// Подсоединение к базе данных.
SqlConnection conn = new SqlConnection(
                              "Server=localhost;" +
                              "Database=master;" +
                              "Integrated Security=true;");
conn.Open();

// Создание команды для вызова хранимой процедуры.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText =
              "EXEC sp_stored_procedures NULL, 'dbo', NULL";

// Выполнение хранимой процедуры.
SqlDataReader rdr = cmd.ExecuteReader();

...

// Освобождение ресурсов.
conn.Close();
```

Дело в том, что все это справедливо только для входных параметров. Если же необходимо извлечь информацию из базы данных (отличную от результирующих наборов данных), без объектов параметров не обойтись. Простое эмпирическое правило гласит, что объекты параметров используются для обработки выходных, входных-выходных параметров, а также возвращаемых значений. Если же вы имеете дело лишь с входными параметрами, то, вероятно, зря тратите усилия, используя объекты параметров вместо строк.

Транзакции и ADO.NET

Одна из наиболее распространенных проблем программирования взаимодействия с базами данных связана с необходимостью обеспечения совместного выполнения нескольких дискретных операций. Например, если вы разрабатываете систему управления медицинскими карточками и ваша база данных не смогла сохранить информацию о новом пациенте, сумев при этом сохранить сведения о его визите, система будет находиться в несогласованном состоянии. Для решения подобных задач и существуют транзакции. Транзакция базы данных позволяет создать оболочку для набора операций,

Приведенный выше параметризированный запрос позволяет узнать количество записей, соответствующих критерию, определенному параметром запроса.

Параметры в параметризированных запросах используются таким же образом, как и в хранимых процедурах с той лишь разницей, что вам не нужно создавать параметр для возвращаемого значения — в параметризированных запросах его попросту нет.

Следует отметить одну важную особенность — формат параметризированных запросов "родного" управляемого поставщика SQL Server отличается от формата таковых управляемых поставщиков OLE DB и ODBC. В табл. 3.1 приведен соответствующий синтаксис для разных комбинаций управляемых поставщиков и баз данных. Использование параметризированного запроса демонстрируется в листинге 3.11.

Таблица 3.1. Синтаксис параметров для различных управляемых поставщиков и баз данных

Управляемый поставщик	База *данных*	*Синтаксис параметра*
SQL Server	SQL Server	@Имя параметра
OLE DB	SQL Server	?
OLE DB	Oracle	?
OLE DB	MS Access (Let 4.0)	? или @Имя_параметра
ODBC	SQL Server	?
ODBC	Oracle	?
ODBC	DB2	?
ODBC	MS Access	?

Листинг 3.11. Использование параметризированных запросов

```
// Создание команды, содержащей параметризированный запрос.
SqlCommand cmd = conn.CreateCommand() ;
cmd.CommandText =
        "SELECT * FROM Customer Where CustomerID = @CustID";

// Определение параметра.
cmd.Parameters.Add("@CustID", DbType.Guid).Direction =
                                ParameterDirection.Input;
cmd.Parameters["@CustID"].Value = Guid.NewGuid();

...
```

Альтернатива использованию объектов параметров

Вообще говоря, создание команд не должно быть трудной задачей. В большинстве случаев весь процесс можно свести к созданию обычных строк, как показано в листинге 3.12.

Листинг 3.12. Выполнение запроса без использования параметров

```
SqlConnection conn = new SqlConnection("...");

SqlCommand cmd = conn.CreateCommand();

// Значение столбца CustomerlD.
int custID = 12345;

// Создание команды с помощью метода string.Format().
```

```
// Если процедура была выполнена успешно,
// извлекаем значение ключа.
Int32 key;
if (sp.RETURN_VALUE — 0)
{
  key = sp.Key;
}
```

При желании можно определить каждый параметр отдельно и обращаться к классу команды напрямую, как показано в листинге 3.10.

Листинг 3.10. Определение **параметров хранимой процедуры через ее оболочку**

```
SqlConnection conn — new SqlConnection(
                              "Server=localhost;" +
                              "Database=ADONET;" +
                              "Integrated Security=true;");
conn.Open();

// Создание оболочки хранимой процедуры.
spAddMember sp = new spAddMember(conn);

// Определение параметров.
sp.FirstName = "Greg";
sp.LastName = "Maddux";
sp.Address - "123 Main Street";
sp.City = "Atlanta";
sp.State = "GA";
sp.Zip = "30307";
sp.Phone = "404-555-1212";
sp.Fax - "404-555-1213";

// Выполнение хранимой процедуры.
sp.Command.ExecuteNonQuery();

// Если процедура была выполнена успешно,
// извлекаем значение ключа.
Int32 key;
if (sp.RETURN_VALUE == 0)
{
  key = sp.Key;
}
```

Иногда возникает необходимость в многократном использовании специфичных **запросов, которые нецелесообразно создавать в виде хранимых процедур, В подобном случае следует прибегнуть к разработке** параметризированных **запросов.**

Параметризированные запросы

Рассмотрим один из наиболее простых вариантов создания параметризированных запросов. Предположим, что у нас есть запрос, который возвращает число записей, соответствующих **определенному критерию.**

```
SELECT COUNT(*) FROM CUSTOMER WHERE SATS = 'MA'
```

Этот запрос возвращает результирующий набор данных, состоящий из одной записи, — количества строк, соответствующих заданному критерию. С помощью параметра можно динамически указать этот критерий, например:

```
SELECT COUNT(*) FROM CUSTOMER WHERE STATE = ?
```

```
    _Zip.DbType = DbType.String;
    _Zip.Direction = ParameterDirection.Input;
    _Zip.SourceVersion = DataRowVersion.Current;
    _cmd.Parameters.Add(_Zip);

    // Параметр Phone.
    SqlParameter _Phone = new SqlParameter();
    _Phone.ParameterName = "@Phone";
    _Phone.DbType = DbType.String;
    _Phone.Direction = ParameterDirection.Input;
    _Phone.SourceVersion = DataRowVersion.Current;
    _cmd.Parameters.Add(_Phone);

    // Параметр Fax.
    SqlParameter _Fax = new SqlParameter();
    _Fax.ParameterName = "@Fax";
    _Fax.DbType = DbType.String;
    _Fax.Direction = ParameterDirection.Input;
    _Fax.SourceVersion = DataRowVersion.Current;
    _cmd.Parameters.Add(_Fax);

    // Параметр Key.
    SqlParameter _Key = new SqlParameter();
    _Key.ParameterName = "@Key";
    _Key.DbType = DbType.Int32;
    _Key.Direction = ParameterDirection.Output;
    _Key.SourceVersion = DataRowVersion.Current;
    _cmd.Parameters.Add(_Key);
  }

  // Защищенное поле, представляющее объект команды.
  protected SqlCommand _cmd;
}
```

На Web-узле **ADOGuy** (www.adoguy.com/book) находится средство, позволяющее создавать аналогичный класс-оболочку для любой хранимой процедуры. Генерируемый класс-оболочка прост в использовании, что и продемонстрировано в листинге 3.9.

Листинг 3.9. Использование класса-оболочки хранимой процедуры

```
SqlConnection conn = new SqlConnection(
                                "Server=localhost;" +
                                "Database=ADONET;" +
                                "Integrated Security=true;");
conn.Open();

// Создание оболочки хранимой процедуры.
spAddMember sp = new spAddMember(conn);

// Вызов хранимой процедуры.
sp.Execute("Maddux",
           "Greg",
           "123 Main Street",
           "Atlanta",
           "GA",
           "30307",
           "404-555-1212",
           "404-555-1213");
```

```
// Защищенный метод, в котором создается объект команды.
protected void ConstructCommand()
{
  _cmd = new SqlCommand("spAddMember") ;
  _cmd.CommandType = CommandType.StoredProcedure;

  // Параметр RETURN_VALUE.
  SqlParameter _RETURN_VALUE = new SqlParameter();
  _RETURN_VALUE.ParameterName = "@RETURN_VALUE";
  _RETURN_VALUE.DbType = DbType.Int32;
  _RETURN_VALUE.Direction =
                        ParameterDirection.ReturnValue;
  _RETURN_VALUE.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_RETURN_VALUE);

  // Параметр FirstName.
  SqlParameter _FirstName = new SqlParameter();
  _FirstName.ParameterName = "@FirstName";
  _FirstName.DbType = DbType.String;
  _FirstName.Direction = ParameterDirection.Input;
  _FirstName.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_FirstName);

  // Параметр LastName.
  SqlParameter _LastName = new SqlParameter();
  _LastName.ParameterName = "@LastName";
  _LastName.DbType = DbType.String;
  _LastName.Direction = ParameterDirection.Input;
  _LastName.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_LastName);

  // Параметр Address.
  SqlParameter _Address = new SqlParameter();
  _Address.ParameterName = "@Address";
  _Address.DbType = DbType.String;
  _Address.Direction = ParameterDirection.Input;
  _Address.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_Address);

  // Параметр City.
  SqlParameter _City = new SqlParameter ();
  _City.ParameterName = "@City";
  _City.DbType = DbType.String;
  _City.Direction = ParameterDirection.Input;
  _City.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_City);

  // Параметр State.
  SqlParameter _State = new SqlParameter();
  _State.ParameterName = "@State";
  _State.DbType = DbType.String;
  _State.Direction = ParameterDirection.Input;
  _State.SourceVersion = DataRowVersion.Current;
  _cmd.Parameters.Add(_State);

  // Параметр Zip.
  SqlParameter _Zip = new SqlParameter();
  _Zip.ParameterName = "@Zip";
```

```
    }
  }
  public  String LastName
  {
    set
    {
      _cmd.Parameters["@LastName"].Value = value;
    }
  }
  public  String Address
  {
    set
    {
      _cmd.Parameters["@Address"].Value = value;
    }
  }
  public  String City
  {
    set
    {
      _cmd.Parameters["@City"].Value = value;
    }
  }
  public  String State
  {
    set
    {
      _cmd.Parameters["@State"].Value = value;
    }
  }
  public  String Zip
  {
    set
    {
      _cmd.Parameters["@Zip"].Value = value;
    }
  }
  public  String Phone
  {
    set
    {
      _cmd.Parameters["@Phone"].Value = value;
    }
  }
  public  String Fax
  {
    set
    {
      _cmd.Parameters["@Fax"].Value = value;
    }
  }
  public  Int32 Key
  {
    get
    {
      return (Int32) _cmd.Parameters["@Key"].Value;
    }
  }
```

Листинг 3.8. Создание класса-оболочки хранимой процедуры

```
public class spAddMember : IDisposable
{
  public spAddMember(SqlConnection conn)
  {
    ConstructCommand();
    _cmd.Connection = conn;
  }

  public SqlCommand Command
  {
    get
    {
      return _cmd;
    }
  }

  public Int32 Execute( string firstName,
                        string lastName,
                        string address,
                        string city,
                        string state,
                        string zip,
                        string phone,
                        string fax)
  {
    FirstName = firstName;
    LastName = lastName;
    Address = address;
    City = city;
    State = state;
    Zip = zip;
    Phone = phone;
    Fax = fax;
    _cmd.ExecuteNonQuery();
    return RETURN_VALUE;
  }

  // Реализация интерфейса IDisposable.
  public void Dispose()
  {
    _cmd.Dispose();
  }

  // Свойства для доступа к параметрам.
  public  Int32 RETURN_VALUE
  {
    get
    {
      return (Int32) _cmd.Parameters["@RETURN_VALUE"].Value;
    }
  }
  public  String FirstName
  {
    set
    {
      _cmd.Parameters["@FirstName"].Value = value;
```

(NVarChar, VarChar, Char) свойство Size представляет максимальный размер строки. Значение по умолчанию определяется на основе типа DbType параметра — в большинстве случаев его можно использовать, не боясь возникновения каких-либо проблем. Если код хранимой процедуры надежен, определение максимального размера параметра способно уменьшить вероятность передачи процедуре заведомо неверных значений, длина которых превышает максимально допустимую (в этом случае база данных генерирует ошибку).

- Direction. Данное свойство определяет способ передачи параметра. Его возможные значения — Input, Output, InputOutput и ReturnValue — представлены перечислением ParameterDirection. По умолчанию используется значение Input.
- IsNullable. Это свойство определяет, может ли параметр принять значение null. По умолчанию используется значение false.
- Value. Значение параметра. Для параметров типа Input или InputOutput это свойство должно быть установлено до выполнения команды, а для параметров типа InputOutput, Output и ReturnValue его значение устанавливается в результате выполнения команды. Чтобы передать пустой входной параметр, нужно либо не устанавливать значение свойства Value, либо установить его равным DBNull. По умолчанию используется значение DBNull.
- Precision. Определяет число знаков (слева от запятой), использующихся для представления значения параметра. По умолчанию используется значение 0.
- Scale. Определяет число десятичных разрядов (число знаков справа от запятой), использующихся для представления значения параметра. По умолчанию используется значение 0.
- SourceColumn. Данное свойство определяет способ использования параметра с объектом **DataAdapter** (за более детальной информацией обратитесь к главе 8).
- SourceVersion. Определяет, как и предыдущее свойство, способ использования параметра с объектом DataAdapter (см. главу 8).

Создание оболочки для хранимой процедуры

Разработка ADO.NET-кода для доступа к хранимой процедуре может оказаться утомительным и скучным занятием, так как при этом обычно требуется определять отдельно каждый параметр. В листинге 3.7 показан пример упрощения этой операции.

Листинг 3.7. Определение параметров

```
// Слишком много кода,
SqlParameter param = new SqlParameter();
param.ParameterName = "@RETURN_VALUE";
param.DbType = DbType.Int32;
param.Direction = ParameterDirection.ReturnValue;
_cmd.Parameters.Add(param);

// Сейчас все выглядит кратко и лаконично.
_cmd.Parameters.Add("@RETURN_VALUE",
                    DbType.Int32).Direction =
                    ParameterDirection.ReturnValue;
```

Преимущество первого способа заключается в предельной ясности кода. С другой стороны, второй способ, действительно, очень лаконичен. При желании вы можете воспользоваться еще одним способом, заключающимся в инкапсуляции всей хранимой процедуры в класс-оболочку, как показано в листинге 3.8.

```
param = new SqlParameter("@sp_name", SqlDbType.NVarChar);
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add(param);

param = new SqlParameter("@sp_owner", SqlDbType.NVarChar);
param.Direction = ParameterDirection.Input;
param.Value = "dbo";
cmd.Parameters.Add(param);

param =
      new SqlParameter("@sp_qualifier", SqlDbType.NVarChar);
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add(param);

// Выполнение хранимой процедуры.
SqlDataReader rdr = cmd.ExecuteReader();

// Перебор всех записей и вывод на консоль
// имен хранимых процедур.
while (rdr.Read())
{
  Console.WriteLine("Proc: {0}", rdr["PROCEDURE_NAME"]);
}

// Освобождение ресурсов.
conn.Dispose();
```

Мы определили четыре параметра, которые необходимо передать хранимой процедуре sp_stored_procedure. Наверняка, вы заметили, что некоторые входные параметры не были инициализированы — по умолчанию, им будет присвоено значение null (в данном случае — DBNull). Другими словами, для того чтобы присвоить параметру значение null, его не нужно инициализировать. Для каждого параметра были заданы также несколько различных свойств. Некоторые из свойств параметров перечислены ниже (следует отметить, что не каждый параметр требует определения всех свойств).

- ParameterName. Имя параметра. Как правило, каждый разработчик баз данных имеет собственное мнение относительно именования параметров. Так, в управляемом поставщике SQL Server и поставщике OLE DB для Access имена параметров предваряются символом @. С другой стороны, большинство остальных поставщиков OLE DB и все поставщики ODBC и Oracle определяют параметры по их позиции. В этом случае нужно создавать параметры в том порядке, в котором они передаются в хранимую процедуру или параметризованный запрос.
- DbType. Тип хранящихся в параметре данных. В .NET-перечислении DbType содержатся типы, которые можно определить для свойства DbType. Кроме того, каждый управляемый поставщик имеет свойство, более точно отражающее реальный тип данных используемой СУБД. Например, в управляемом поставщике SQL Server есть перечисление SqlDbType. При выборе специфичного для поставщика элемента перечисления автоматически устанавливается соответствующее значение DbType. Именно так производится отображение типов данных СУБД на типы управляемого поставщика. Например, установка свойства DbType параметра SqlParameter в значение DbType.Boolean приведет к автоматической установке свойства SqlDbType в значение SqlDbType.Bit.
- size. Свойство Size зависит от типа данных параметра и обычно используется для указания его максимальной длины. Например, для строковых типов

```
SqlCommand sqlcmd = sqlconn.CreateCommand();
sqlcmd.CommandType = CommandType.Text;
sqlcmd.CommandText = "SELECT * FROM Customer FOR XML AUTO;";

//...

XmlReader xml = sqlcmd.ExecuteXmlReader();
```

Разумеется, мы должны хорошо представлять себе, что происходит в результате выполнения той или иной команды. Например, если нам необходимо внести изменения в базу данных, то, вероятнее всего, для этого будет достаточно удостовериться в успешном выполнении команды, вместо того чтобы анализировать полученный результирующий набор данных.

Результирующий набор данных

Наиболее типичное использование команд заключается в выполнении единственной команды для возвращения одного набора данных. Большинство выполняемых команд предназначены для получения информации. Используя объект **DataReader**, мы обычно имеем дело с прямоугольным набором данных, полученным в результате выполнения запроса или хранимой процедуры и называющимся *результирующим набором данных (result set)*.

Однако сможем ли мы получить все результирующие наборы данных в случае выполнения нескольких команд или хранимой процедуры, осуществляющей более одного запроса? В этой ситуации база данных возвращает множество отдельных результирующих наборов данных. Как мы узнаем из главы 4, объект DataReader поддерживает средства для работы с каждым из этих наборов.

Использование параметров

По своей сути параметры базы данных аналогичны параметрам Visual Basic, C++ или C#. Параметры используются для передачи часто изменяющейся информации серверу баз данных и обратно. Объекты параметров могут использоваться как с хранимыми процедурами, так и с параметризированными запросами. ADO.NET позволяет определять параметры при выполнении каждого из вышеназванных типов команд. В листинге 3.6 приведен пример вызова простой хранимой процедуры с помощью управляемого поставщика SQL Server.

Листинг 3.6. **Использование** параметров

```
// Подключение к базе данных.
SqlConnection conn = new SqlConnection(
                        "Server=localhost;Database=master;" +
                        "Integrated  Security=true;");
conn.Open();

// Создание команды для выполнения хранимой процедуры.
SqlCommand cmd = conn.CreateCommand() ;
cmd.CommandText = "sp_stored_procedures";
cmd.CommandType — CommandType.StoredProcedure;

// Определение параметров.
SqlParameter param =
                    new SqlParameter("@RETURN", SqlDbType.Int);
param.Direction = ParameterDirection.ReturnValue;
cmd.Parameters.Add(param);
```

Выполнение команд

За подготовкой команды следует ее выполнение. В ADO.NET существует несколько способов выполнения команд, которые отличаются лишь информацией, возвращаемой из базы данных. Ниже перечислены методы выполнения команд, поддерживаемые всеми управляемыми поставщиками.

- `ExecuteNonQuery()`. Как правило, этот метод применяется для выполнения команд, которые не должны возвращать результирующий набор данных. Так как при вызове метода `ExecuteNonQuery()` возвращается число строк, добавленных, измененных или удаленных в результате выполнения команды, он может использоваться в качестве индикатора успешного завершения операции.
- `ExecuteScalar()`. Этот метод выполняет команду и возвращает первый столбец первой строки первого результирующего набора данных. Метод `ExecuteScalar()` может быть полезен для извлечения из базы данных аналитической информации (например, `SELECT COUNT(userid) FROM USERS;`).
- `ExecuteReader()`. Этот метод выполняет команду и возвращает объект DataReader (более подробно он рассматривается в следующей главе), представляющий собой однонаправленный поток записей базы данных. Возвращаемый каждой версией метода `ExecuteReader()` объект DataReader представлен классом из пространства имен соответствующего управляемого поставщика (`OleDbDataReader`, `SqlDataReader` или `OdbcDataReader`), как показано в листинге 3.4.

Листинг 3.4. Использование метода *ExecuteReader()*

```
//...
SqlDataReader sqlrdr = sqlcmd.ExecuteReader();

while (sqlrdr.Read())
{
  //...
}

//...
OleDbDataReader odbrdr = odbcmd.ExecuteReader();

while (odbrdr.Read())
{
  //...
}
```

Кроме того, управляемый поставщик SQL Server поддерживает метод, возвращающий объект XmlReader.

- `ExecuteXmlReader()`. Этот метод выполняет команду и возвращает экземпляр класса, производного от класса XmlReader, который используется для навигации по результирующему набору данных. Метод `ExecuteXmlReader()` поддерживается только SQL Server 2000 или более поздней версией и требует, чтобы возвращаемые запросом или хранимой процедурой результаты были в формате XML, как показано в листинге 3.5.

Листинг 3.5. Использование метода *ExecuteXmlReader()*

```
SqlConnection sqlconn = new SqlConnection();

//...
```

```
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM Customer;";
```

В последнем случае вместе с созданием нового объекта команды происходит определение соответствующего ему объекта соединения. Так или иначе, требования "привязывать" соединение к определенной команде не существует.

Типы команд

Команды — это мощный инструмент, позволяющий проводить сложные операции с базой данных. В ADO.NET существует три типа команд.

- Text. Текстовая команда состоит из инструкций, указывающих управляемому поставщику на необходимость выполнения определенных действий на уровне базы данных. В большинстве случаев такая команда написана на SQL-диалекте соответствующей базы данных (T-SQL для SQL Server, PL/SQL для Oracle и т.д.). Обычно текстовые команды передаются в базу данных без предварительной обработки (за исключением случаев передачи параметров). Этот тип команд поддерживается всеми управляемыми поставщиками данных: SQL Server, OLE DB, Oracle и ODBC.
- StoredProcedure. Хранимая процедура — это команда, вызывающая процедуру, которая находится в самой базе данных. Такой тип команд поддерживается всеми управляемыми поставщиками: SQL Server, OLE DB, Oracle и ODBC.
- TableDirect. Команда типа TableDirect предназначена для извлечения из базы данных полной таблицы. Она аналогична текстовой команде SELECT * FROM *ИМЯ_ТАБЛИЦЫ*.Команда типа TableDirect поддерживается только управляемым поставщиком OLE DB. (Что касается меня, то я стараюсь избегать ее использования.)

Все команды, включенные в рассмотренные выше типы, можно выполнять по отношению к базе данных. По умолчанию свойство CommandType объекта команды установлено в значение Text. Для определения типа команды используется перечисление CommandType, расположенное в пространстве имен System.Data, как показано в листинге 3.3.

Листинг 3.3. Определение типа команды

```
SqlConnection sqlconn = new SqlConnection();

//...

SqlCommand sqlcmd = sqlconn.CreateCommand();

// Вы можете установить свойство CommandText в Text,
// что, однако же, является излишним.
sqlcmd.CommandText = "SELECT * FROM Customer;";

OleDbConnection odbconn = new OleDbConnection();

//...

OleDbCommand odbcmd = odbconn.CreateCommand();
odbcmd.CommandType = CommandType.StoredProcedure;
odbcmd.CommandText = "spMyStoredProc";
```

Глава 3

Выполнение команд

В этой главе...

Команды

Выполнение команд

Транзакции и ADO.NET

Пакетные запросы

Итак, пришло время познакомиться с действительно серьезными вещами. Эта глава поможет вам научиться использовать объекты команд для выполнения основных задач, связанных со взаимодействием с базой данных. Кроме того, вы узнаете о способах уменьшения сетевого трафика, вызванного выполнением команд. В большинстве систем основная часть кода, обеспечивающего сетевое взаимодействие, так или иначе связана с базами данных. Оптимизация этих операций позволяет эффективно уменьшить общий объем трафика, передаваемого по сети вашей системой.

Команды

Если сравнить соединение с базой данных с телефонным звонком, то разговор, несомненно, соответствует последовательности команд. По существу, соединение с базой данных абсолютно бесполезно без объекта ADO.NET **Command.** Хотим ли мы "попросить" базу данных предоставить определенную информацию или же просто вставить или обновить данные, мы неизбежно сталкиваемся с необходимостью создания объекта **Command.**

Создание объекта Command

Существует два основных способа создания объекта **Command.** Во-первых, можно воспользоваться стандартным синтаксисом создания объекта команды, как показано в листинге 3.1.

Листинг 3.1. Создание объекта команды

```
SqlConnection conn = new SqlConnection();

//...

SqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM Customer;";
```

Во-вторых, объект команды можно создать на основе объекта **Connection** (листинг 3.2).

Листинг 3.2. Создание объекта команды на основе объекта соединения

```
SqlConnection conn = new SqlConnection();

//...
```

```
    try
    {
      // Попытка вызова метода SomeFunction()
      // для получения nResult.
      SomeFunction();
    }
    catch (SqlException ex)
    {
      string error = "";

      // Извлечение сообщения, содержащегося в исключении.
      error += ex.Message + "\n";

      // Присоединение сообщений, содержащихся в объектах Error,
      foreach (SqlError err in ex.Errors)
      {
        error += err.Message + "\n";
      }

      // Если управление было передано этому фрагменту кода,
      // значит, возникло исключение. Нам остается всего лишь
      // сообщить пользователю о типе исключения и подчеркнуть,
      // что во всем виноват поставщик SQL.
      MessageBox.Show("SQL Provider Error: " +
                      exception.Message);
    }
    finally
    {
      // Этот фрагмент кода выполняется вне зависимости
      // от того, было сгенерировано исключение или нет.
      nResult = -1;
    }
    return nResult;
}
```

Резюме

Соединения — это "альфа и омега" доступа к данным. Они являются первыми создаваемыми и последними уничтожаемыми объектами. Кроме того, соединения очень важны в аспекте масштабируемости приложения. Используя пул соединений и в полной мере усвоив политику их создания, можно существенно улучшить производительность и доступность системы. Надеюсь, мне удалось помочь вам разобраться в тайнах "магии" строк соединений. Несмотря на то что в них могут содержаться десятки параметров, использование доступных возможностей должно способствовать повышению эффективности вашей работы.

```
        {
          conn.Open();

          SqlCommand cmd =
                new SqlCommand("SELECT * FROM Teams", conn);

          SqlDataReader reader = cmd.ExecuteReader();

          while (reader.Read())
          {
            Console.WriteLine(reader.GetString(2) +
                        " " +reader.GetString(1));
          }
        }
        catch (Exception ex)
        {
          Console.WriteLine(
                    "An Error Occurred: " + ex.Message);
        }
        finally
        {
          // Закрыть соединение в том случае,
          // если оно было корректно открыто.
          if (conn.State == ConnectionState.Open)
          {
            conn.Close();
          }
        }
      }
    }
}
```

Несмотря на то что новый код значительно больше, логический поток практически не изменился. За исключением того, что в него введен блок catch и дополнительная проверка перед вызовом метода `conn.Close()`, код остался прежним.

С целью облегчения восприятия материала в последующих примерах этой книги код обработки ошибок удален. Следует отметить, что без данного кода вся информация об ошибках будет представлена непосредственно пользователю, это, мягко говоря, выглядит не совсем дружелюбно. Перехват исключения — пусть даже на последнем уровне вашей программы — гуманный акт, даже если это делается лишь с целью вставки логотипа вашей компании сразу же после сообщения об ошибке.

Исключения ADO.NET

В управляемых поставщиках SQL Server и OLE DB предусмотрены специализированные классы исключений. В каждом из этих классов есть свойство Errors, которое является набором объектов **Error** (типа SqlError и OleDbError). В данных объектах содержится дополнительная информация об определенном исключении, полученная от базы данных. При желании ошибки можно проанализировать, чтобы выяснить причину возникновения исключения (листинг 2.12).

Листинг 2.12. Обработка исключения *SqlException*

```
public int Test ()
{
    int nResult = 0;
```

```
    }
    catch (SqlException exception)
    {
      // Если управление было передано этому фрагменту кода,
      // значит, возникло исключение. Нам остается всего лишь
      // сообщить пользователю о типе исключения и подчеркнуть,
      // что во всем виноват поставщик SQL.
      MessageBox.Show("SQL Provider Error: " 4
                      exception.Message);
    }

    catch (Exception exception)
    {
      // Если управление было передано этому фрагменту кода,
      // значит, возникло исключение - сообщим
      // пользователю о его типе.
      MessageBox.Show(exception.Message);
    }
    finally
    {
      // Этот фрагмент кода выполняется вне зависимости
      // от того, было сгенерировано исключение или нет.
      nResult = -1;
    }
    return nResult;
}
```

В приведенном выше примере была показана базовая конструкция try-catch-finally. Если вы хотите разобраться в ней детальнее, обратитесь к документации .NET. При разработке реальных ADO.NET-приложений вы должны быть готовы к исключениям, сгенерированным ввиду невозможности корректного взаимодействия с сервером баз данных. На самом деле исключения свидетельствуют не о наличии ошибки в программном коде, а о возникновении неожиданной ситуации (такой, как отключение кабеля). Обработка исключения дает вам шанс осуществить попытку установки соединения через некоторое время или же отказаться от нее, сообщив при этом пользователю о сбое. Следует также отметить, что вы можете обрабатывать исключения только определенного типа, например исключения поставщика данных, и игнорировать при этом все системные исключения.

В листинге 2.11 приведен переработанный с учетом необходимости обработки исключений пример из листинга 2.1.

Листинг **2,11. Использование блока** *finally*

```
namespace BetterHelloADONET
{
    using System;
    using System.Data;
    using System.Data.SqlClient;

    class Classl
    {
      static void Main(string[] args)
      {
        SqlConnection conn =
                new SqlConnection("Server=localhost;" +
                                  "Integrated Security=true");

        try
```

```
            conn.GetOleDbSchemaTable(OleDbSchemaGuid.Columns,
            new object[] {null, null, row["TABLE_NAME"], null});

        // Изменение имени таблицы (по умолчанию она будет
        // названа Columns, однако мы не можем добавлять
        // таблицы с одинаковыми именами).
        tbl.TableName = row["TABLE_NAME"].ToString();

        // Добавление таблицы в объект DataSet.
        ds.Tables.Add(tbl);
    }
```

При вызове метода GetOleDbSchemaTable указывается одно из полей типа Guid класса OleDbSchemaGuid, что позволяет определить тип необходимой схемы. При первом вызове этого метода возвращается полный список таблиц базы данных, при втором — столбцы определенной таблицы. Метод GetOleDbSchemaTable всегда возвращает объект **DataTable,** содержащий информацию о схеме базы данных. Более подробную информацию об извлечении информационной схемы можно получить, обратившись к документации по определенному полю класса OleDbSchemaGuid (например, OleDbSchemaGuid.Tables).

Обработка ошибок в .NET

К сожалению, приведенные выше примеры не иллюстрируют реального использования ADO.NET. Ведь, по сути дела, взаимодействие с базой данных — это прежде всего сетевое программирование, а выполнение запросов по сети усложняет даже самые простые функции. Несмотря на то что все не так уж плохо, нам придется принять некоторые меры предосторожности, связанные с необходимостью обработки ошибок. Когда имеешь дело с сетью, нужно настроиться на худшее: сервер, к которому вы пытаетесь подсоединиться, может неожиданно "замолчать"; ваших полномочий может оказаться недостаточно; сетевой кабель может быть отсоединен какой-то слишком заботливой уборщицей.

Учитывая все эти обстоятельства, мы должны быть готовы к эффективной обработке ошибок. При возникновении ошибки платформа .NET генерирует исключение. Некоторые разработчики хорошо знакомы с этой темой, но, например, для Visual Basic-разработчиков, она может оказаться совершенно новой. Visual Basic-разработчики, скажем, могут представлять себе обработку исключений чем-то похожим на синтаксис On Error Goto.

Смысл обработки исключений таков: при неудачной попытке что-то сделать процесс генерирует объект исключения, который может быть перехвачен вызывающим кодом. Для перехвата исключений (и их обработки) в среде .NET предусмотрена структура try-catch-finally, типичный сценарий с ее использованием приведен в листинге 2.10.

Листинг 2.10. Обработка ошибок в .NET

```
public int Test()
{
    int nResult = 0;

    try
    {
      // Попытка вызова метода SomeFunction()
      // для получения nResult.
      SomeFunction();
```

```
            return conn;
        }
        catch (Exception e)
        {
            // ...
            return null;
        }
    }
}
```

Приведенная в качестве примера динамическая фабрика объектов соединения предусматривает абстрагирование от поставщика OLE DB и сервера баз данных, что позволяет интегратору или отделу информационных систем производить изменения без внесения правки в программный код. Тем не менее этот пример является не очень показательным, так как мы предполагали, что параметры .NET-приложения сохраняются в .config-файле (app.config, web.config, machine.config или даже enterprise.config).

Получение информационной схемы базы данных с помощью поставщика OLE DB

OLE DB поддерживает извлечение информационной схемы базы данных с помощью *наборов строк схемы (schema rowsets)*. Управляемый поставщик OLE DB предоставляет данную функциональность в виде метода OleDbConnection.GetOleDbSchemaTable(). Чтобы получить информационную схему, необходимо указать тип схемы и список ограничений (определяет число возвращаемых схем). Пример получения информационной схемы базы данных с помощью поставщика OLE DB приведен в листинге 2.9.

Листинг 2.9. Получение информационной схемы базы данных с помощью поставщика OLE DB

```
// Создание соединения с базой данных.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = "Provider=SQLOLEDB;" +
                        "Server=localhost;" +
                        "Database=ADONET;" +
                        "Integrated Security=SSPI;";
conn.Open();

// Создание объекта DataSet.
DataSet ds = new DataSet();

// Извлечение информации о таблицах, хранящихся в базе данных.
ds.Tables.Add(conn.GetOleDbSchemaTable(
                  OleDbSchemaGuid.Tables,
                  new object[] {null, null, null, "TABLE"}));

// Добавление новых таблиц в объект DataSet - по одной
// на каждую таблицу, хранящуюся в Сазе данных.
foreach (DataRow row in ds.Tables[0].Rows)
{
    // Извлечение таблицы, содержащей столбцы таблицы,
    // хранящейся а базе данных.
    DataTable tbl =
```

```
    {
      OleDbConnection conn - new OleDbConnection();

      // Формирование строки соединения.
      conn.ConnectionString = "Provider=SQLOLEDB.1; " +
                              "Data Source=localhost; " 4
                              "Database=" + sDatabase + ";" +
                              "UID=sa;" +
                              "Pwd=password;" +
                              "ConnectionTimeout=5;";
      return conn;
    }
}
```

Особенность фабрики соединений заключается в том, что подробности создания соединения спрятаны от разработчика. Если бы задача состояла в абстрагировании от используемого управляемого поставщика данных, то вместо специфического объекта соединения фабрика вернула бы интерфейс IDbConnection. В этом случае пользователи все равно могли бы открывать и закрывать соединения, не задумываясь о том, какой именно используется поставщик.

Рассмотренная ранее фабрика соединений слишком упрошена. В реальных ситуациях чаще всего используется динамическая фабрика соединений, предусматривающая получение информации о соединении из реестра компьютера, на котором выполняется приложение (листинг 2.8).

Листинг 2.8. Динамическая фабрика соединений

```
public class ConnectionFactory
{
    static public OleDbConnection CreateRegisteredConnection()
    {
      try
      {
        // Параметры строки соединения могут быть сохранены
        // в реестре, файле app.config или machine.config.

        // Извлечение параметров строки соединения из реестра.
        string sKey =
               "Software\\AW\\ADO.NET\\Chapter2\\ConnFactory";
        RegistryKey fac =
               Registry.LocalMachine.OpenSubKey(sKey);
        string sSrc = fac.GetValue("Data Source").ToString();
        string sDB = fac.GetValue("Database").ToString();
        string sUID = fac.GetValue("Database").ToString();
        string sPwd = fac.GetValue("Pwd").ToString();

        // Создание объекта Connection.
        OleDbConnection conn = new OleDbConnection();

        // Формирование строки соединения.
        conn.ConnectionString = "Provider=SQLOLEDB.1; " +
                                "Data Source=" + sSrc +
                                ";Database=" + sDB +
                                ";UID=" + sUID +
                                ";Pwd=" + sPwd +
                                ";ConnectionTimeout=5;";

        fac.Close();
```

```
    ' Обработчик события InfoEvent.
    Public Sub ConnInfo (ByVal sender As Object,
                         ByVal e As OleDbInfoMessageEventArgs)

      Console.WriteLine( "Connection Info Message:" +
                         e.Message)
    End Sub

    ' Обработчик события StataChange.
    Public Sub ConnChange (ByVal sender As Object,
                           ByVal e As StateChangeEventArgs)

      Console.WriteLine ("Connection State Changed: From: " & _
                         e.OriginalState ToString() fi _
                         " to " + _
                         e.CurrentState.ToString())
    End Sub
End Class
```

В отдельных случаях событие InfoMessage поддерживается в стандартных управляемых поставщиках и поставщиках OLE DB. Одни поставщики предоставляют большое количество информационных сообщений, другие — сообщают о чем-либо крайне неохотно. Событие InfoMessage предназначено для доставки пользователю информационных и предупреждающих сообщений, однако если последние не генерируются нижележащим поставщиком, управляемый поставщик остается не у дел,

В этом смысле событие StateChange является более традиционным. Вследствие организации пула соединений последнее событие StateChange обычно не срабатывает, так как соединение не закрывается до тех пор, пока не истечет время его нахождения в пуле. Событие InfoMessage можно использовать для ведения журнала трассировки, однако в целом эти события имеют весьма ограниченную ценность.

Фабрика соединений

При разработке ADO.NET-приложения имеет смысл использовать единообразные соединения с целью обеспечения возможности их занесения в пул и упрощения создания. К сожалению, компания Microsoft решила сделать классы соединений изолированными, что препятствует порождению от них потомков. Общепринятым способом создания единообразных объектов Connection в ADO.NET является использование класса ConnectionFactory. Шаблон ConnectionFactory позволяет создавать новые соединения по мере необходимости, тем самым уменьшая количество повторяющегося программного кода. Пример реализации фабрики соединений приведен в листинге 2.7.

Листинг **2.7. Фабрика** соединений

```
public class ConnectionFactory
{

    static public OleDbConnection CreateOleDbConnection ()
    {
      return CreateOleDbConnection ("ADONET");
    }

    static public OleDbConnection CreateOleDbConnection(
                                    string sDatabase]
```

```
        Console.WriteLine(  "Connection State Changed: Prom: " +
                            e.OriginalState.ToString<) +
                            " to " +
                            e.CurrentState.ToString());
    }

}
```

Синтаксис обработки событий и использования делегатов в VB.NET немного отличается от такового в C#. VB.NET не поддерживает синтаксис += для добавления обработчика события. Для этого в нем предусмотрен оператор AddHandler. В листинге 2.6 приведен аналогичный код, написанный на VB.NET.

Листинг 2.6. События класса соединения (VB.NET)

```
Public Class ShowEvents

    Public Sub Show()

      ' Создание объекта Connection.
      Dim conn As OleDbConnection = New OleDbConnection()

      conn.ConnectionString = "Provider=SQLOLEDB.1;" & _
                              "Server=localhost;" & _
                              "Database=Northwind;" & _
                              "Integrated Security=SSPI;"

      ' Регистрация обработчиков событий.
      AddHandler (conn.InfoMessage, AddressOf ConnInfo)
      AddHandler (conn.StateChange, AddressOf ConnChange)

      ' Открытие соединения.
      conn.Open()

      ' Создание объекта Command.
      Dim cmd As OleDbCommand = conn.CreateCommand()
      cmd.CommandText = "SELECT * from [Order Details];" +
                        "SELECT * from orders;"

      ' Создание объекта DataReader
      ' в результате выполнения запроса.
      Dim rdr As OleDbDataReader = cmd.ExecuteReader()

      ' Вывод результатов выполнения запроса на консоль.
      Do

        Console.WriteLine("Result:")
        ' Перебор всех записей результирующего
        ' набора данных.
        While rdr.Read()

          Console.WriteLine(rdr.Item(0))

        End While

      Loop While rdr.NextResult()

    End Sub
```

В листинге 2.5 показаны простые способы обработки событий.

Листинг 2.5. События класса **соединения (C#)**

```
public class ShowEvents
{
    public void Show()
    {
      // Создание объекта соединения,
      OleDbConnection conn = new OleDbConnection() ;
      conn.ConnectionString = "Provider=SQLOLEDB.1;" +
                              "Server=localhost;" +
                              "Database=Northwind;" +
                              "Integrated Security=SSPI;";

      // Регистрация для получения уведомления о событии.
      conn.InfoMessage +=
            new OleDbInfoMessageEventHandler(this.ConnInfo);
      conn.StateChange +=
            new StateChangeEventHandler(this.ConnChange);

      // Открытие соединения.
      conn.Open();

      // Создание объекта Command.
      OleDbCommand cmd = conn.CreateCommand() ;
      cmd.CommandText - "SELECT * from [Order Details];" +
                        "SELECT * from orders;";

      // Создание объекта DataReader
      // в результате выполнения запроса.
      OleDbDataReader rdr = cmd.ExecuteReader();

      // Вывод результатов выполнения запроса на консоль.
      do
      {
        Console.WriteLine("Result:");
        // Перебор всех записей результирующего
        // набора данных.
        while (rdr.Read())
        {
          Console.WriteLine(rdr[0]) ;
        }
      }
      while (rdr.NextResult()) ;
    }

    // Обработчик события InfoEvent.
    void ConnInfo(object sender, OleDbInfoMessageEventArgs e)
    {
      Console.WriteLine("Connection Info Message:" +
                        e.Message);
    }

    // Обработчик события StateChange.
    void ConnChange(object sender, StateChangeEventArgs e)
    {
```

```
    [System.Runtime.InteropServices.DllImport
     ("odbc32.dll",
     CharSet=System.Runtime.InteropServices.CharSet.Auto)]

    private static extern int SQLSetEnvAttr(
                              long Environment/
                              long EnvAttribute,
                              long ValuePtr,
                              long StringLength);

    const long SQL_ATTR_CONNECTION_POOLING = 201;
    const long SQL_CP_ONE_PER_DRIVER - 1;
    const long SQL_IS_INTEGER - -6;
    const long SQL_CP_OFF - 0;

    static int Enable()
    {
      return SQLSetEnvAttr( 0,
                            SQL_ATTR_CONNECTION_POOLING,
                            SQL_CP_ONE_PER_DRIVER,
                            SQL_IS_INTEGER);
    }
    static int Disable()
    {
      return SQLSetEnvAttr( 0,
                            SQL_ATTR_CONNECTION_POOLING,
                            SQL_CP_OFF,
                            SQL_IS_INTEGER);
    }
}
```

Активизирование механизма организации пула соединений необходимо осуществлять один раз для каждого процесса. Как вы уже догадались, приведенный выше класс может быть использован для эффективного активизирования/деактивизирования механизма организации пула соединений ODBC. К сожалению, "тонкая" настройка параметров пула не поддерживается.

События объекта Connection

Во время жизни объекта Connection его статус и роль могут меняться множество раз. Классы соединения всех трех стандартных управляемых поставщиков данных поддерживают уведомляющие события на случай получения информационных сообщений и изменения состояния соединения. Обработка этих событий предполагает регистрацию для получения уведомления о соответствующих событиях класса соединения (табл. 2.6).

Таблица **2.6. События класса соединения**

Событие	*Описание*
Disposed	Возникает при вызове метода `Dispose` класса соединения. Обычно используется для освобождения ресурсов, которые не могут быть освобождены автоматически сборщиком мусора
`InfoMessage`	Возникает, когда поставщик возвращает информационное или предупреждающее сообщение, Вызывается на усмотрение управляемого поставщика. Одни поставщики извлекают пользу из этого события, другие — нет
`StateChange`	Возникает при открытии или закрытии соединения. Данное событие позволяет получить информацию о новом и старом состоянии соединения

Первый способ заключается в использовании вкладки Connection Pooling диалогового окна средства ODBC Data Source Administrator (Администратор источника данных ODBC) версии 2.3 или выше (рис. 2.2). Дважды щелкните на имени драйвера и разрешите или запретите заносить соединения в пул (рис. 2.3). Следует отметить, что совершаемые вами действия имеют влияние на все программное обеспечение компьютера.

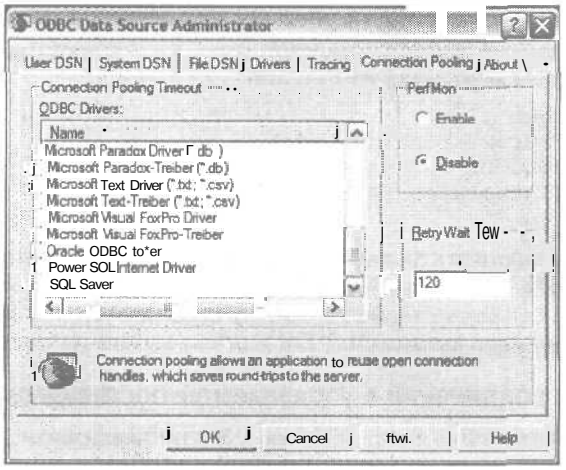


Рис. 2.2. Вкладка Connection Pooling диалогового окна ODBC Data Source Administrator

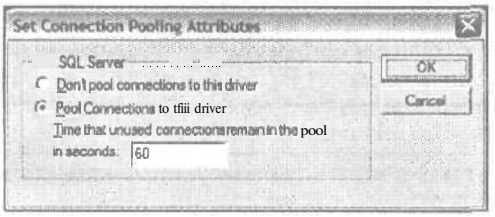


Рис. 2.3. Активизирование механизма организации пула соединений

Второй способ активизирования механизма организации пула соединений ODBC заключается в использовании вызова ODBC API во время выполнения процесса. Импортирование ODBC-вызова осуществляется с помощью атрибута `DLLImport`. В листинге 2.4 показан пример создания класса-оболочки для вызовов ODBC API.

Листинг 2.4. Активизирование механизма организации пула соединений ODBC

```
/*
    Пример использования класса:
    ODBCPooling.Enable();
*/
public class ODBCPooling
f
```

Поставщик OLE DB поддерживает несколько служб, включая механизм организации пула соединений (называемый в данном случае *механизмом организации пула ресурсов (resource pooling)*), автоматическое связывание транзакций и механизм клиентского курсора. Активизировать или деактивизировать эти службы позволяет ключ реестра `OLEDB_SERVICES`. Поскольку изменение ключа реестра влияет на все приложения, находящиеся на данном компьютере, для изменения этих настроек лучше использовать параметр строки соединения OLE DB Services, возможные значения которого приведены в табл. 2.4.

Ниже приведен пример деактивизирования механизма организации пула соединений и автоматического связывания транзакций.

```
OleDbConnection conn = new
OleDbConnection("Server=localhost;" +
                "OLE DB Services=-4; " +
                "Integrated Security=true");
conn.Open();
```

Использование параметров строки соединения — это единственный способ контроля за службами OLE DB. Несмотря на возможность использования OLE DB-метода `IDataInitialize::GetDataSource()`, это представляется целесообразным только в случае непосредственного вызова OLE DB или его вызова из поставщика OLE DB.

Организация пула соединений в управляемом поставщике данных Oracle

В управляемом поставщике Oracle пул соединений реализован почти так же, как и в управляемом поставщике SQL Server. Другими словами, в рамках одного процесса сроки соединения всех занесенных в пул соединений идентичны. Для изменения стандартного поведения механизма организации пула соединений управляемого поставщика Oracle используются несколько параметров (табл. 2.5).

Таблица 2.5. Параметры механизма организации пула соединений управляемого поставщика Oracle

Параметр	*Описание*
Pooling	Булево выражение, которое определяет необходимость использования пула соединений, Значение по умолчанию — t rue
Min Pool Size	Минимальное число соединений в пуле в любой момент времени. Значение по умолчанию — 0
Max Pool Size	Максимальное число соединений в пуле. При исчерпании свободных соединений следующий клиент будет ждать до тех пор, пока не появится свободное соединение или пока не истечет время ожидания установки соединения. Значение по умолчанию — 100
Enlist	Булево выражение, определяющее необходимость автоматического связывания соединения с контекстом текущей транзакции потока. Значение по умолчанию — true
Connection Lifetime	Определяет время (в секундах), по истечении которого открытое соединение будет закрыто и удалено из пула. Сравнение времени создания соединения с текущим временем проводится по возвращении соединения в пул. Если соединение не запрашивается, а время, заданное параметром Connection Lifetime, истекло, соединение закрывается

Организация пула соединений в управляемом поставщике данных ODBC

В управляемом поставщике ODBC нет собственной поддержки механизма организации пула соединений, однако в ODBC версии 3.0 и выше такая поддержка имеется. Существует два способа активизирования механизма организации пула соединений ODBC, но ни в одном из них не предусмотрено применение возможностей ADO.NET.

Таблица **2.3. Параметры механизма** организации **пула** соединений **управляемого** поставщика **SQL Server**

Параметр	*Описание*
Pooling	Булево выражение, которое определяет необходимость использования пула соединений. Значение по умолчанию — true
Min Pool size	Минимальное число соединений в пуле в любой момент времени. Значение по умолчанию — 0
Max Pool size	Максимальное число соединений в пуле. При исчерпании свободных соединений следующий клиент будет ждать до тех пор, пока не появится свободное соединение или пока не истечет время ожидания установки соединения. Значение по умолчанию — 100
Enlist	Булево выражение, определяющее необходимость автоматического связывания соединения с контекстом текущей транзакции потока. Значение по умолчанию — true
isolation Level	Уровень изоляции транзакции соединения. Значение по умолчанию — ReadCommitted
Connection Lifetime	Определяет время (в секундах), по истечении которого открытое соединение будет закрыто и удалено из пула. Сравнение времени создания соединения с текущим временем проводится при возвращении соединения в пул. Если соединение не запрашивается, а время, заданное параметром Connection Lifetime, истекло, соединение закрывается

Организация пула соединений в управляемом поставщике данных OLE DB

Организация пула соединений в управляемом поставщике OLE DB значительно отличается от таковой в управляемом поставщике SQL Server. Контроль за пулом соединений проводится нижележащим поставщиком OLE DB (а не управляемым поставщиком ADO.NET) с помощью класса `OleDbConnection`. Этот процесс должен быть абсолютно прозрачен для разработчика ADO.NET, за исключением статического метода `OleDbConnection.ReleaseObjectPool()`, который оповещает нижележащего поставщика OLE DB о том, что приложение какое-то время не будет обращаться к источнику данных и что после возвращения в пул всех соединений его можно удалить. Подобное поведение помогает поставщику OLE DB более эффективно уничтожать пул соединений. В связи с тем, что у пула и у каждого из хранящихся в нем соединений есть определенное "время жизни", пул не может быть уничтожен до истечения этого времени. Уничтожение пула можно ускорить, вызвав метод `ReleaseObjectPool()`.

Таблица **2.4. Значения параметра строки** соединения **OLE *DB Services***

Предоставляемые службы	*Значение параметра OLE DB Services*
Все службы (по умолчанию)	-1
Все службы, кроме механизма организации пула соединений и автоматического связывания транзакций	-4
Все службы, кроме механизма клиентского курсора	-5
Все службы, кроме механизма организации пула соединений, автоматического связывания транзакций и механизма клиентского курсора	-8
Только службы организации пула соединений и автоматического связывания транзакций (агрегирование на уровне сеансов)	3
Службы OLE DB не используются	0

Организация пула соединений

Соединения — это ценный товар, а потому написание кода, минимизирующего число одновременно открытых соединений (а, значит, и нагрузку на сервер), способно улучшить производительность базы данных. К счастью, ADO.NET (как и ее предшественники) предусматривает наличие средства управления соединениями — так называемого пула соединений. Организация пула соединений представляет собой процесс управления совместно используемыми ресурсами (соединениями), которые выделяются из пула недавно использованных соединений. При организации пула соединений учитывается тот факт, что для большинства приложений требуется установка соединения на очень небольшой промежуток времени, тогда как создание и освобождение объекта соединения представляют собой довольно "дорогую" операцию. Другими словами, пул соединений — это способ их повторного использования. Настоящее же чудо происходит тогда, когда все соединения закрываются, поскольку на самом деле они еще продолжают храниться в пуле некоторое время (заданное одним из параметров пула). Если в течение этого времени приложение запросит установку соединения, пул предоставит ему уже открытое соединение, экономя время на освобождении и создании нового объекта соединения. Используя пул соединений, вы тем самым уменьшаете вероятность обращения пользователя к базе данных только для того, чтобы обнаружить отсутствие свободных соединений. Пул соединений позволяет также уменьшить время, необходимое для уведомления приложения об отсутствии свободных соединений. Фактически с помощью пула соединений можно блокировать запрос на установку соединения до тех пор, пока не освободится новое соединение, что позволяет ограничить нагрузку на сервер баз данных.

Реализация пула соединений отличается в каждом из управляемых поставщиков. Несмотря на то что она прозрачна для разработчиков баз данных, понимание различий между существующими механизмами управления пулом соединений позволяет создавать наиболее эффективный в каждом конкретном случае код.

Организация пула соединений в управляемом поставщике данных SQL Server

Управляемый поставщик SQL Server создает пул соединений с абсолютно идентичными (байт в байт) строками соединения; при этом используется информация о соединении, заданная при его установке, а не определенная вами вручную. Например, при создании соединения с параметрами `"Server=localhost; Database=ADONET;"` в строке соединения будут содержаться стандартные значения, включая информацию, относящуюся к безопасности (если только вы не запретите сохранять подобные сведения). Это означает, что попытка поместить в пул соединение при условии использования встроенной системы безопасности завершится неудачно, так как относящаяся к безопасности информация нового соединения будет отличаться от информации, ассоциированной с уже содержащимися в пуле соединениями (это вполне естественно, если учесть тот *факт,* что каждому пользователю назначены его собственные параметры безопасности, которые могут не совпадать с аналогичными параметрами других пользователей).

В управляемом поставщике SQL Server средства создания пула соединений реализованы внутри пространства имен `System.Data`[4]. Если вы хотите понаблюдать за процессом организации пула, воспользуйтесь средством SQL Server Profiler, позволяющим наблюдать создание и уничтожение соединений. Управляемый поставщик SQL Server предоставляет ограниченный контроль за работой механизма организации пула посредством параметров строки соединения (табл. 2.3).

---

[4] Пул соединений управляемого поставщика SQL Server недоступен во время работы отладчика Visual Studio .NET.

Листинг **2.3. Изменение базы данных**

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace EmptyProject
{
    class ShowDatabaseChange
    {
        static void Main(string[] args)
        {
          // Создание объекта Connection и формирование
          // строки соединения.
          SqlConnection conn = new SqlConnection();
          conn.ConnectionString = "Server=localhost;" +
                                  "Database=ADONET;" +
                                  "Integrated Security=True; " +
                                  "ConnectionTimeout=5;";

          // Использование объекта Connection.
          SqlCommand cmd = new SqlCommand(
                           "SELECT COUNT(au_id) FROM CUSTOMER;");
          cmd.Connection = conn;

          // Открытие соединения.
          conn.Open();

          // Выполнение команды.
          int result = (int) cmd.ExecuteScalar();

          // Изменение текущей базы данных.                    *
          conn.ChangeDatabase("master");

          // Изменение команды.
          cmd.CommandText =
                  "SELECT COUNT(name) FROM dbo.sysdatabases; ";

          // Выполнение новой команды.
          int nDatabases = Convert.ToInt32(cmd.ExecuteScalar());

          // Закрытие соединения.
          conn.Close();

          // Освобождение ресурсов.
          conn.Dispose();
        }
    }
}
```

Идея заключается в том, что создание новых соединений (в отличие от открытия уже существующих) — это напрасная трата времени и программного кода. Если обращаться с объектами **Connection** как с соединениями с источником данных (т.е. с определенным сервером), а не соединениями с определенной базой данных (хранилищем данных на этом сервере), можно существенно оптимизировать повторное использование объектов **Connection.**

Продолжительность ожидания установки соединения определяется в строке соединения; кроме того, ADO.NET позволяет узнать это значение посредством предназначенного только для чтения свойства объекта Connection (т.е. без необходимости анализа строки соединения). Следует отметить, что продолжительность ожидания установки соединения можно изменить лишь путем модификации строки соединения и повторного открытия последнего.

Встроенная система безопасности

Управляемые поставщики SQL Server, OLE DB и Oracle поддерживают концепцию встроенной системы безопасности. Параметр Integrated Security — это средство, позволяющее ADO.NET или OLE DB определить аутентификационную информацию базы данных на основе соответствующих параметров текущего пользователя Windows. Например, текущий пользователь интерактивного приложения, зарегистрировавшийся в сети, считается зарегистрировавшимся и на сервере SQL Server или Oracle.

Использование встроенной системы безопасности может привести к возникновению проблем, связанных с соединениями ASP.NET-страниц — если только не применяется заимствование прав доступа или информация из каталога COM+, текущий пользователь будет представлять собой достаточно "слабую" сущность (MACHINE или другую непривилегированную учетную запись). Данная учетная запись пользователя генерируется при установке IIS-сервера и является достаточно безопасной, так как дает право на доступ только к файловой системе, на которой размешен Web-узел. Я не рекомендую предоставлять подобным пользователям доступ к базе данных SQL Server, чтобы избежать появления потенциальной бреши в системе безопасности. Гораздо лучше создать новую учетную запись пользователя, специально предназначенную для аутентификации в базе данных. При этом требуется чтобы либо управляемый поставщик не использовал встроенную систему безопасности, либо ASP.NET-страница применяла заимствование прав доступа — в любом случае это небольшая цена за безопасность. В Web-приложениях рекомендуется интегрировать аутентификационную информацию в строку соединения, вместо того чтобы использовать встроенную систему безопасности. Что же касается настольных приложений, то это вполне допустимо.

Изменение базы данных

Объект Connection представляет собой соединение с источником данных, а не с одной базой данных. Параметр Database строки соединения позволяет выбрать часть сервера баз данных, которая будет использоваться на протяжении всего соединения. Такие параметры, как User ID и Password, поддерживают изменение строки соединения, позволяя выбрать логическую область источника данных. Когда соединение будет закрыто, можно изменять все остальные параметры и только после этого — строку соединения.

В следующем примере рассматривается использование метода ChangeDatabase(). После создания объекта Connection и определения строки соединения последнее открывается, а затем выполняется запрос к базе данных ADONET. Но что делать в том случае, если нам необходимо извлечь дополнительную информацию из того же сервера, однако хранящуюся в другой базе данных? Для этого необходимо выполнить метод ChangeDatabase(), например чтобы сменить текущую базу данных на базу данных master.

Следует отметить, что при использовании метода ChangeDatabase() нужно убедиться в том, что соединение все еще открыто. Если же изменить базу данных нужно при закрытом соединении, внесите соответствующие коррективы в строку соединения, с тем чтобы сократить время привязки к базе данных (листинг 2.3).

стороны, использование DSN позволяет сохранить определенный контроль над строками соединения. Так, если меняется местоположение сервера или аутентификационная информация, вам придется изменить всего лишь атрибуты DSN, а не программный код.

Атрибуты DSN можно использовать также для динамического генерирования информации о соединении. В этом случае параметры строки соединения можно указывать непосредственно в строке соединения (а не прятать их в атрибуте DSN), что очень похоже на определение параметров в управляемых поставщиках SQL Server и Oracle. Исчерпывающий список поддерживаемых параметров можно найти в документации поставщика ODBC. Ниже перечислены наиболее часто используемые параметры.

- DSN (Имя источника данных). Имя источника данных или путь к файлу.
- Driver (Драйвер). Имя драйвера ODBC, обычно заключенное в фигурные скобки. В табл. 2.2 перечислены наиболее часто используемые драйверы,

Таблица 2.2. Наиболее часто используемые драйверы ODBC

Драйвер	*Источник данных*
SQL Server	**SQL** Server
Oracle ODBC Driver	**Oracle**
Microsoft Access Driver (*.mdb)	Файлы Microsoft Access (.mdb)

- Server (Сервер). Имя или URL-адрес сервера баз данных, например: 192.168.0.1, myserver.raydomain.com или `localhost` **(для локального сервера).**
- Database (База данных). При подключении к серверу баз данных (такому, как SQL Server или Oracle) в этом параметре хранится имя базы данных. В случае использования "файловых" поставщиков ODBC, таких, как Microsoft Access, параметр `Database` не используется.
- UID (Идентификатор пользователя). Идентификатор, который должен применяться при аутентификации пользователя на сервере баз данных.
- `PWD` (Пароль). Пароль, который должен применяться для аутентификации пользователя на сервере баз данных.

Продолжительность ожидания установки соединения

Иногда возникает ситуация, когда сервер баз данных слишком "занят" и не может обработать запрос. К счастью, определение продолжительности ожидания установки соединения находится полностью в вашей компетентности — идея заключается в том, чтобы обеспечить более эффективный способ определения "порога", за которым сервер баз данных будет считаться перегруженным. Для поставщиков данных SQL Server и OLE DB значение продолжительности ожидания установки соединения по умолчанию равно 15 с. Этот параметр можно установить также и в нуль (бесконечное ожидание), чего, однако, делать не рекомендуется, поскольку причина, по которой невозможно установить соединение с сервером может оказаться неустранимой.

Ввиду необходимости обеспечить определенный уровень реагирования и в зависимости от сервера баз данных вам может понадобиться использовать собственное значение продолжительности ожидания установки соединения. К примеру, при разработке Web-страниц можно ограничиться очень небольшим значением, чтобы Web-сервер сразу сообщил пользователю о перегрузке сервера баз данных. Всегда лучше сразу же сообщить пользователю о том, что он не может получить доступ к информации, чем заставлять его ждать тот же ответ в течение 30 или более секунд.

активизировать встроенную систему безопасности, в строке соединения установите значение параметра Integrated Security=SSPI. Более подробно вопрос использования встроенной системы безопасности рассмотрен далее в этой главе.

- Connection Timeout (Продолжительность попытки установки соединения). Время в секундах, в течение которого будет осуществляться попытка установки соединения. Значение по умолчанию равно 15 с.

Строки соединения управляемого поставщика Oracle

Управляемый поставщик Oracle[3] поддерживает синтаксис строки соединения, очень похожий на синтаксис строки соединения SQL Server. В действительности синтаксис строки соединения Oracle — это подмножество синтаксиса строки соединения SQL Server. Исчерпывающий список поддерживаемых параметров можно найти в документации поставщика Oracle. Ниже перечислены наиболее часто используемые параметры,

- Server (Сервер). Имя или сетевой адрес базы данных Oracle.
- Integrated Security (Встроенная система безопасности). Булев флаг, который указывает, будет Oracle использовать NT-аутентификацию или производить проверку пары "имя пользователя/пароль". Установив значение параметра Integrated Security в true, вы указываете Oracle на необходимость использования текущего дескриптора NT Security для аутентификации доступа к базе данных.
- User ID (Идентификатор пользователя). Идентификатор, который должен применяться для аутентификации пользователя на сервере баз данных (предполагается, что значение параметра Integrated Security равно false).
- Password (Пароль). Пароль, который должен применяться для аутентификации пользователя на сервере баз данных (предполагается, что значение параметра Integrated Security **равно** false).

Строки соединения управляемого поставщика ODBC

Строки соединения управляемого поставщика ODBC немного отличаются от строк соединения SQL Server или OLE DB. Управляемый поставщик ODBC поддерживает два различных метода создания строки соединения:

- создание строки соединения на основе имени источника данных (Data Source Name - DSN);
- использование динамических строк соединения.

DSN — это набор атрибутов соединения, определенных для локального компьютера и хранящихся на нем. DSN создается с помощью аплета Источники данных (ODBC) (ODBC Data Source Administrator), который находится либо непосредственно на Панели управления (Control Panel), либо в папке Администрирование (Administrative Tools). Определить строку соединения в этом случае довольно просто:

```
OdbcConnection conn = new OdbcConnection();
conn.ConnectionString = "DSN=MyDSN;";
```

Недостаток использования DSN заключается в том, что каждый компьютер должен либо быть специально настроенным, либо иметь доступ к DSN-файлам. В частности, это бывает проблематичным в таких системах, как кластер Web-серверов. С другой

[3] **На момент написания книги** управляемый **поставщик Oracle от компании Microsoft существовал только в бета-версии. Пожалуйста,** просмотрите документацию, **распространяющуюся вместе с** управляемым **поставщиком Oracle,** дабы убедиться, **что синтаксис** строки **соединения не изменился.**

- Password (Пароль). Пароль, который должен применяться для аутентификации пользователя на сервере баз данных (предполагается, что значение параметра Integrated Security **равно** false).
- Connection Timeout (Продолжительность попытки установки соединения). Время в секундах, в течение которого будет осуществляться попытка установки соединения. Значение по умолчанию равно 15 с.

Строки соединения управляемого поставщика OLE DB

Строки соединения управляемого поставщика OLE DB похожи на строки соединения SQL Server. С точки зрения синтаксиса строка соединения управляемого поставщика OLE DB — это строка соединения OLE DB. Все параметры строки соединения, за исключением Provider, определяются специфическим поставщиком OLE DB. Исчерпывающий список поддерживаемых параметров можно найти в документации поставщика OLE DB. Чтобы получить информацию обо всех поставщиках OLE DB компании Microsoft, обратитесь к Web-узлу MSDN (msdn.microsoft.com). Там же вы сможете найти полное описание всех поддерживаемых параметров управляемого поставщика OLE DB[2]. Ниже перечислены наиболее часто используемые параметры строки соединения управляемого поставщика OLE DB.

- Provider (Поставщик). Идентификатор ProgID для конкретного поставщика данных OLE DB. Список наиболее распространенных поставщиков приведен в табл. 2.1.

Таблица 2.1. Наиболее **распространенные поставщики данных OLE DB**

ProgID	*Источник данных*
SQLOLEDB.1	SQLServer
MSDASQL.1	Поставщик OLE DB для ODBC специально не поддерживается (его использование приводит к возникновению ошибки времени исполнения), В этом случае рекомендуется использовать управляемый поставщик ODBC
MSDAORA.1	Oracle
Microsoft.Jet.OLEDB.4.0	Файлы Microsoft Access(.mdb)

- Server (Сервер). Имя, сетевой адрес сервера баз данных или имя файла базы данных (для "файловых" поставщиков OLE DB, таких, как Microsoft Access или FoxPro), например: 192.168.0.1, myserver.mydomain.com, localhost или c:\access\northwind.mdb.
- Database (База данных). При подключении к серверу баз данных (такому, как SQL Server или Oracle) в этом параметре хранится имя базы данных. В случае использования "файловых" поставщиков OLE DB, таких, как Microsoft Access, параметр Database не используется.
- UID (Идентификатор пользователя). Идентификатор, который должен применяться при аутентификации пользователя на сервере баз данных.
- PWD (Пароль). Пароль, который должен применяться для аутентификации пользователя на сервере баз данных.
- Integrated Security (Встроенная система безопасности). Тип используемой встроенной системы безопасности (если таковая вообще используется). Чтобы

[2] См. msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataOleDbOleDbConnectionClassConnectionStringTopic.asp

параметров строки соединения подробно описаны в документации, я не буду на них останавливаться. При желании вы можете посетить Web-узел Карла Протмэна (Carl Prothman) по адресу: www.able-consulting.com/ado_conn.htm, — на котором он собрал огромное число примеров строк соединений.

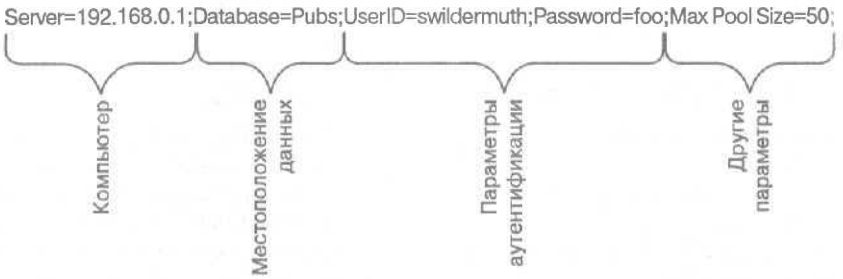


Рис. 2.1. Типичная строка соединения

Значения некоторых параметров строки соединения доступны в управляемых поставщиках через значения свойств, предназначенных только для чтения. Вне зависимости от того, определите вы соответствующие параметры в строке соединения или нет, вы сможете узнать их значения. Например, если в строке соединения SQL Server значение параметра ConnectionTimeout не определено, вы все равно можете узнать его с помощью свойства SqlConnection.ConnectionTimeout.

В следующих разделах этой главы рассматриваются особенности обработки строки соединения управляемыми поставщиками данных, а также наиболее часто использующиеся параметры строки соединения для каждого управляемого поставщика.

Строки соединения управляемого поставщика SQL Server

Строки соединения управляемого поставщика SQL Server содержат множество параметров, однако наиболее часто используются только некоторые из них. (Полное описание поддерживаемых параметров можно найти на Web-узле MSDN (Microsoft Developer Network))[1]. Ниже перечислены самые распространенные параметры.

- Server (Сервер). Имя или адрес компьютера с установленным сервером SQL Server (например, 192.168.0.1, myserver.mydomain.com или localhost).
- DataBase (База данных). База данных, находящаяся на сервере (например, Pubs или Master).
- Integrated Security (Встроенная система безопасности). Булев флаг, который указывает, будет SQL Server использовать NT-аутентификацию или производить проверку пары "имя пользователя/пароль". Установив значение параметра Integrated Security в true, вы указываете SQL Server на необходимость использования текущего дескриптора NT Security для аутентификации доступа к базе данных. По умолчанию этот параметр равен false. Более подробно вопрос использования встроенной системы безопасности рассмотрен ниже, в разделе "Встроенная система безопасности".
- User ID (Идентификатор пользователя). Идентификатор, который должен применяться для аутентификации пользователя на сервере баз данных (предполагается, что значение параметра Integrated Security равно false).

---

[1] См. msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataSqlClientSqlConnectionClassConnectionStringTopic.asp

```
// Изменение команды.
cmd.CommandText = "SELECT MAX(Discount)" +
                  " FROM Customer;";

// Повторное открытие соединения.
conn.Open();

// Выполнение команды,
float maxDiscount = (float) cmd.ExecuteScalar();

// Закрытие соединения.
conn.Close();
```

Это код наглядно иллюстрирует тот факт, что объект **Connection** может существовать дольше, чем один цикл открытия/закрытия соединения. Другими словами, открытие и закрытие соединения совсем не обязательно соответствует созданию и уничтожению объекта **Connection** — соединение может быть открыто и закрыто несколько раз за время его существования.

При создании объекта **Connection** определяется строка соединения, в которой описывается местонахождение источника данных (в нашем случае — локальный сервер баз данных). Затем соединение открывается и выполняется команда — запрос к базе данных ADONET. Как только получена нужная информация, соединение закрывается. После обработки этой информации можно опять открыть соединение и продолжать работать с тем же источником данных. Обратите внимание, что объект **Command можно** использовать повторно, так как необходимости в его создании для каждой выполняемой операции нет. Пока обрабатывается извлеченная из базы данных информация, соединение закрыто, чтобы уменьшить загрузку сервера базы данных. Хотя на повторное открытие соединения будет потрачено некоторое время, ADO.NET предусматривает наличие пула соединений (рассматривается далее в этой главе), что позволяет минимизировать затраты.

Строки соединений

Класс соединения — это способ осуществить "телефонный звонок" к базе данных в ADO.NET. Как и при телефонном разговоре, существует протокол, предназначенный для установки диалога между абонентами. Вам необходимо знать номер телефона, имя и/или фамилию человека, с которым нужно поговорить, и, иногда, иметь доступ к личной информации (такой, как PIN-код, девичья фамилия матери или номер социального страхования). Имея в наличии всю эту информацию, выполнить звонок очень просто. Строка соединения — это не что иное, как способ "доставки" информации, необходимой для установки соединения с базой данных, управляемому поставщику.

Правил, определяющих синтаксис строки соединения для всех поставщиков данных, не существует; каждый из них может иметь свой собственный синтаксис. К счастью, все управляемые поставщики Microsoft используют очень похожие строки соединения, которые представляют собой список параметров (пара "имя/значение"), разделенных точкой с запятой. Каждый параметр состоит из имени, знака равенства (=), и значения, определенного для этого параметра. В подобных строках соединения регистр и пробелы не имеют значения. Типичная строка соединения состоит из нескольких параметров, как показано на рис. 2.1.

В строке соединения управляемых поставщиков необходимо, как минимум, указать местоположение базы данных и требуемую аутентификационную информацию. Помимо этого, каждый поставщик данных определяет дополнительные параметры соединения. Если в строке соединения не указаны значения всех возможных параметров, они считаются установленными по умолчанию. Поскольку стандартные значения

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Customers",
                                conn);

// Выполнение запроса.
SqlDataReader reader = cmd.ExecuteReader();

// Вывод результатов запроса на консоль.
while (reader.Read())
{
    Console.WriteLine(reader["FirstName"] + " " +
                      reader["LastName"]);
}

// Закрытие соединения.
conn.Close();
```

Этот пример очень похож на рассмотренный нами в главе 1 пример **из** листинга 1.4. Вначале мы создаем соединение с базой данных и открываем его, затем указываем на необходимость передать нам всю таблицу Customers и, наконец, выводим все строки таблицы на консоль. Таким образом, этот пример демонстрирует базовую функциональность объектов **Connection и Command:** установку соединения и выполнение команды.

Соединения

ADO.NET-соединение представлено отдельным классом соединения управляемого поставщика данных, который соответствует соединению с источником данных, однако это не означает возможность выполнения только одного запроса. Поскольку ADO.NET ориентирована на работу с данными в отсоединенном режиме, класс соединения должен поддерживать повторяющиеся открытия и закрытия соединения. В листинге 2.2 показан рекомендуемый жизненный цикл объекта **Connection.**

Листинг 2.2. Жизненный цикл соединения

```
// Создание объекта соединения.
SqlConnection conn =
              new SqlConnection("Server=localhost;" +
                                "Database=ADONET;" +
                                "Integrated Security=true;");

// Использование соединения.
SqlCommand cmd = new SqlCommand("SELECT COUNT(CustomerID) " +
                                " FROM Customer",
                                conn);

// Открытие соединения.
conn.Open();

// Выполнение команды.
int result = (int) cmd.ExecuteScalar();

// Закрытие соединения.
conn.Close();

// Обработка полученных данных.
// ...
```

Глава 2

ADO.NET: подключение к источнику данных

В этой главе...

Первые впечатления

Соединения

Получение информационной схемы базы данных с помощью поставщика OLE DB

Обработка ошибок в .NET

Я часто сравниваю программирование доступа к базе данных в отсоединенном режиме с заказом пиццы на ужин. Сначала следует позвонить в пиццерию и заказать пиццу с теми или иными ингредиентами. Как только мне доставляют пиццу, я оплачиваю услуги курьера и отпускаю его, чтобы он мог продолжать свою работу. Мне бы и в голову не пришло попросить его подождать, пока я буду есть пиццу. По существу, работа с базой данных представляет собой последовательность "доставок пиццы". Прежде всего нам нужно установить соединение (позвонить в пиццерию), затем выполнить команду (заказать пиццу) и, наконец, закрыть соединение и обработать полученную информацию (отпустить курьера и начать наслаждаться процессом употребления пиццы). В ADO.NET установка соединения аналогична звонку по телефону, а выполнение команды — заказу пиццы. Более подробно выполнение команд рассматривается в главе 3, а процесс "потребления" данных — в главах 4–8.

Первые впечатления

Перед тем как мы изучим все аспекты работы с соединениями, обратимся к простейшему примеру — своеобразному "Hello, world!" ADO.NET. В листинге 2.1 приведен пример использования ADO.NET для запроса из базы данных одной таблицы и вывода ее содержимого на консоль.

Листинг 2,1. Использование соединения

```
using System;
using System. Data;
using System.Data.SqlClient;

// Создание объекта соединения.
SqlConnection conn =
              new SqlConnection("Server=localhost;" +
                                "Database=ADONET;" 4
                                "Integrated Security=true;");

// Открытие соединения.
conn.Open();

// Создание объекта команды.
```

Для того чтобы создать поставщик данных ADO.NET, разработчику необходимо определить новое пространство имен и набор классов, реализующих базовые интерфейсы из пространства имен System.Data.Common. Следует отметить, что в новое пространство имен не обязательно должны входить только те классы, которые реализуют интерфейсы из пространства имен System.Data.Common. К примеру, управляемый поставщик SQL Server имеет несколько специфичных классов, реализующих функциональность, характерную для SQL Server. В главах 2 и 3 я постараюсь тщательно проследить взаимосвязи между управляемыми поставщиками данных и пространством имен System. Data. Common.

Резюме

Надеюсь, что эта глава полностью убедила вас в необходимости перехода от технологий доступа к данным, ориентированных на установку соединения, к "отсоединенной" философии ADO.NET. Ведь если разобраться, то каждый разработчик, стремящийся "присвоить" себе соединение на больший срок, чем это необходимо, — просто жадина. К счастью, ADO.NET позволяет практически не выходить за пределы "отсоединенного" мира. Вы сможете написать лучший код, ваши базы данных перестанут капризничать, а это значит, что однажды вы все-таки сможете позволить себе купить ту яхту, которую уже давно облюбовали!

XML-документа. **В листинге 9.1 показан пример сохранения уже рассматривавшегося нами объекта DataSet в формате XML.**

Листинг 9.1. Получение XML-представления объекта *DataSet*

```
...

// Создание объекта DataAdapter для каждой
// таблицы, извлекаемой из базы данных.
SqlDataAdapter daCustomers = new
  SqlDataAdapter("SELECT * FROM CUSTOMER;",
                 conn);
SqlDataAdapter daInvoices = new
  SqlDataAdapter("SELECT * FROM INVOICE;",
                 conn);
SqlDataAdapter dalnvoiceltems = new
  SqlDataAdapter("SELECT * FROM INVOICEITEM;",
                 conn);

// Создание пустого объекта DataSet.
DataSet dataSet - new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");
daInvoiceItems.Fill(dataSet, "InvoiceItems");

// Закрытие соединения.
conn.Close();

// Определение переменных для
// упрощения доступа к таблицам.
DataTable customerTable    = dataSet.Tables["Customers"];
DataTable invoiceTable     = dataSet.Tables["Invoices"];
DataTable invoiceItemTable = dataSet.Tables["InvoiceItems"];

// Определение первичных ключей таблиц.
customerTable.PrimaryKey = new DataColumn[]
  { customerTable.Columns["CustomerID"] };
invoiceTable.PrimaryKey = new DataColumn[]
  { invoiceTable.Columns["InvoiceID"] };
invoiceItemTable.PrimaryKey = new DataColumn[]
  { invoiceItemTable.Columns["InvoiceItemID"] };

// Определение отношений.
// Создание первого отношения (между таблицами Invoices и
// InvoiceItems). При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
dataSet.Relations.Add( "Invoices_InvoiceItems",
  dataSet.Tables["Invoices"].Columns["InvoiceID"],
  dataSet.Tables["InvoiceItems"].Columns["InvoiceID"],
  true);

// Создание второго отношения (между
// таблицами Customers и Invoices).
// При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
```

```
dataSet.Relations.Add( "Customers_Invoices",
  dataSet.Tables["Customers"].Columns["CustomerID"],
  dataSet.Tables["Invoices"].Columns["CustomerID"],
  true);

// Вывод XML-представления объекта DataSet на консоль.
Console.WriteLine(dataSet.GetXml());
```

В результате выполнения этого кода на консоль будет выведен следующий XML-код:

```
<NewDataSet>
  <Customers>
    <CustomerID>
      a945884d-d309-4b47-ba42-1aff5cd035cf
    </CustomerID>
    <FirstName>Mike</FirstName>
    <LastName>Remlinger</LastName>
    <Address>10 Aaron Way</Address>
    <City>Atlanta</City>
    <State>GA</State>
    <Zip>30307</Zip>
    <HomePhone>(404) 543-8765</HomePhone>
    <Stamp>AAAAAAAAAIA=</Stamp>
    <CheckedOut>false</CheckedOut>
  </Customers>
...
  <Invoices>
    <InvoiceID>
      463cdf10-2609-41f3-99a3-05f4cedf9fe2
    </InvoiceID>
    <InvoiceNumber>100</InvoiceNumber>
    <InvoiceDate>
      2002-03-21T00:00:00.0000000-05:00
    </InvoiceDate>
    <Terms>Net 10th</Terms>
    <PO>100042</PO>
    <CustomerID>
      11d59cb7-bf61-4540-9317-4f154d717796
    </CustomerID>
  </Invoices>
...
  <InvoiceItems>
    <InvoiceItemID>
      4150d81c-79cb-458b-a303-047e3cd6dbea
    </InvoiceItemID>
    <InvoiceID>
      5e1e9beb-5915-4513-8d67-0f3e0f6a81d0
    </InvoiceID>
    <ProductID>1</ProductID>
    <Quantity>1</Quantity>
    <Discount>0</Discount>
  </InvoiceItems>
...
</NewDataSet>
```

Формат XML очень прост. Каждая сущность находится на один уровень ниже основного контейнера и не связана с другими сущностями. Дело в том, что все отношения описаны в схемах таблиц, а при простом выводе XML-представления объекта DataSet информация о схеме не используется.

При желании вложенность объектов DataRelation в объект **DataSet** можно указать явно, что позволит генерировать информацию об отношениях при XML-сериализаиии, Для того чтобы это сделать, добавьте к листингу 9.1 код из листинга 9.2.

Листинг 9.2. Помечаем отношения как вложенные

```
...

// Помечаем отношения как вложенные.
dataSet.Relations["Customers_Invoices"].Nested = true;
dataSet.Relations["Invoices_InvoiceItems"].Nested = true;

// Вывод XML-представления объекта DataSet на консоль.
Console.WriteLine(dataSet.GetXml());
```

Теперь сгенерированный XML код будет выглядеть так:

```
<NewDataSet>
  <Customers>
    <CustomerID>
      11d59cb7-bf61-4540-9317-4f154d717796
    </CustomerID>
    <FirstName>Greg</FirstName>
    <LastName>Maddux</LastName>
    <Address>53 Peachtree Center</Address>
    <City>Atlanta</City>
    <State>GA</State>
    <Zip>30342</Zip>
    <HomePhone>404-555-1234</HomePhone>
    <Stamp>AAAAAAAAAH8=</Stamp>
    <CheckedOut>false</CheckedOut>
    <Invoices>
      <InvoiceID>
         463cdf10-2609-41f3-99a3-05f4cedf9fe2
      </InvoiceID>
      <InvoiceNumber>100</InvoiceNumber>
      <InvoiceDate>
         2002-03-21T00:00:00.0000000-05:00
      </InvoiceDate>
      <Terms>Net 10th</Terms>
      <PO>100042</PO>
      <CustomerID>
         11d59cb7-bf61-4540-9317-4f154d717796
      </CustomerID>
      <InvoiceItems>
      <InvoiceItemID>
         c7395127-891d-4ed4-8c37-d4e73bcfdf55
      </InvoiceItemID>
      <InvoiceID>
         463cdf10-2609-41f3-99a3-05f4cedf9fe2
      </InvoiceID>
      <ProductID>4</ProductID>
      <Quantity>2</Quantity>
      <Discount>4.25</Discount>
  -</InvoiceItems>
    <InvoiceItems>
      <InvoiceItemID>
        b30b585f-5fe7-48fd-b97a-e7a31badOd72
      </InvoiceItemID>
```

```
      <InvoiceID>
        463cdf10-2609-41f3-99a3-05f4cedf9fe2
      </InvoiceID>
      <ProductID>2</ProductID>
      <Quantity>3</Quantity>
      <Discount>0</Discount>
    </InvoiceItems>
    </Invoices>
    <Invoices>
      <InvoiceID>
        5e1e9beb-5915-4513-8d67-0f3e0f6a81d0
      </InvoiceID>
      <InvoiceNumber>101</InvoiceNumber>
      <InvoiceDate>
         2002-03-31T00:00:00.0000000-05:00
      </InvoiceDate>
      <Terms>Net 30</Terms>
      <PO>100104</PO>
      <CustomerID>
        11d59cb7-bf61-4540-9317-4f154d717796
      </CustomerID>
      <InvoiceItems>
        <InvoiceItemID>
          4150d81c-79cb-458b-a303-047e3cd6dbea
        </InvoiceItemID>
        <InvoiceID>
           5e1e9beb-5915-4513-8d67-0f3e0f6a81d0
        </InvoiceID>
        <ProductID>1</ProductID>
        <Quantity>1</Quantity>
        <Discount>0</Discount>
      </InvoiceItems>
    </Invoices>
  </Customers>
</NewDataSet>
```

В рамках информации о каждом клиенте хранится информация о его счетах, а в рамках информации о каждом счете — сведения о его составляющих. При необходимости иерархического представления XML-данных таблицы могут быть сделаны вложенными.

Определение структуры объектов DataColumn

По умолчанию для представления данных столбцов таблицы используются вложенные элементы, а не атрибуты. Тем не менее иногда может возникнуть необходимость сериализации каждого столбца таблицы именно в виде атрибута. Чтобы указать на необходимость использования атрибутов для представления столбцов таблиц, следует определить дополнительные свойства объекта **DataColumn**, как показано в листинге 9.3.

Листинг 9.3. Определение значения свойства ***DataColumn.MappingType***

```
...

// Теперь все столбцы будут представлены
// в XML-документе в виде атрибутов.
foreach(DataColumn col in customerTable.Columns)
  col.ColumnMapping = MappingType.Attribute;
```

```
foreach(DataColumn col in invoiceTable.Columns)
  col.ColumnMapping = MappingType.Attribute;
foreach(DataColumn col in invoiceItemTable.Columns)
  col.ColumnMapping = MappingType.Attribute;

// Вывод XML-документа в файл.
dataSet.WriteXml("test.xml");
```

В результате выполнения приведенного выше фрагмента кода будет получен следующий XML-документ:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Customers CustomerID=
             "11d59cb7-bf61-4540-9317- 4f154d717796"
             FirstName="Greg"
             LastName="Maddux"
             Address="100 Henry Aaron Way"
             City="Atlanta"
             State="GA"
             Zip="30307"
             HomePhone="404-555-1234"
             Stamp="AAAAAAAAAGk="
             CheckedOut="false">
    <Invoices InvoiceID=
              "463cdf10-2609-41f3-99a3-05f4cedf9fe2"
              InvoiceNumber="100"
              InvoiceDate=
                    "2002-03-21T00:00:00.0000000-05:00"
              Terms="Net 30"
              PO="100042"
              CustomerID=
                    "11d59cb7-bf61-4540-9317-4f154d717796">
      <InvoiceItems
         InvoiceItemID=
                    "c7395127-891d-4ed4-8c37-d4e73bcfdf55"
         InvoiceID="463cdf10-2609-41f3-99a3-05f4cedf9fe2"
         ProductID="4"
         Quantity="2"
         Discount="4.25" />

...

    </Invoices>
  </Customers>
</NewDataSet>
```

В этом примере производится итерация по всем столбцам всех таблиц, в результате которой свойство столбцов `ColumnMapping` устанавливается в необходимое значение (в данном случае это `MappingType.Attribute`). Перечисление MappingType содержит следующие элементы (типы отображения).

- Element — в результирующем XML-коде данному столбцу будет соответствовать элемент, вложенный в элемент таблицы. Это значение используется по умолчанию.
- Attribute — в результирующем XML-коде данному столбцу будет соответствовать атрибут элемента таблицы.
- Hidden — данный столбец будет исключен из результирующего XML-кода.

- SimpleContent — в результирующем XML-коде данному столбцу будет соответствовать текст внутри элемента таблицы. Это отображение не может быть использовано, если хотя бы для одного столбца было определено отображение Element или хотя бы один столбец имеет вложенные отношения.

Свойство ColumnMapping определяется для каждого конкретного столбца, что позволяет задавать его XML-отображение. При необходимости можно преобразовать часть столбцов в атрибуты, а часть — оставить в виде элементов, как показано в листинге 9.4.

Листинг 9.4. Преобразование произвольных столбцов в атрибуты XML-документа

```
...

// Теперь идентификационные столбцы будут представлены
// атрибутами в результирующем XML-документе.
customerTable.Columns["CustomerID"].ColumnMapping =
                                    MappingType.Attribute;
invoiceTable.Columns["InvoiceID"].ColumnMapping =
                                    MappingType.Attribute;

// Вывод XML-документа в файл.
dataSet.WriteXml("test.xml");
```

Ниже приведен фрагмент результирующего файла test.xml.

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Customers CustomerID=
                    "11d59cb7-bf61-4540-9317-4f154d717796">
    <FirstName>Greg</FirstName>
    <LastName>Maddux</LastName>
    <Address>100 Henry Aaron Way</Address>
    <City>Atlanta</City>
    <State>GA</State>
    <2ip>30307</Zip>
    <HomePhone>404-555-1234</HomePhone>
    <Stamp>AAAAAAAAAGk=</Stamp>
    <CheckedOut>false</CheckedOut>
    <Invoices InvoiceID=
                    "463cdf10-2609-41f3-99a3-05f4cedf9fe2">
      <InvoiceNumber>100</InvoiceNumber>

...

</NewDataSet>
```

В рассмотренном выше примере единственными столбцами, сериализированными в виде атрибутов, являются идентификационные столбцы. Все остальные столбцы сериализируются в виде элементов.

Сохранение данных объекта DataSet в формате XML

Более распространенной причиной использования XML-представления данных является необходимость их сохранения или потоковой передачи. Для обработки операций такого типа в классе DataSet предусмотрен метод WriteXml. Пример сохранения XML-представления данных объекта DataSet в файле приведен в листинге 9.5.

Листинг 9.5. Сохранение XML-представления данных объекта *DataSet* в формате XML

```
...

// Вывод XML-документа в файл.
dataSet.WriteXml("test.xml");
```

В предыдущем разделе мы обсуждали возможность определения структуры XML-файла, однако иногда возникает необходимость также и в определении его формата. Пример определения формата XML-файла приведен в листинге 9.6.

Листинг 9.6. Определение формата XML-файла

```
using System.Xml;
using System.IO;

...

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter("test.xml");

// Создание объекта XmlTextWriter
// лля форматирования XML-файла.
XmlTextWriter xmlWriter = new XmlTextWriter(writer);

// Определение способа форматирования XML-файла.
xmlWriter.Formatting = Formatting.Indented;
xmlWriter.Indentation = 3;

// Сохранение XML-файла.
dataSet.WriteXml(xmlWriter);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close{'};
```

В результате выполнения приведенного выше фрагмента кода будет создан XM L-файл, отформатированный с помощью абзацных отступов.

```
<NewDataSet>
   <Customers>
      <CustomerID>
        11d59cb7-bf61-4540-9317-4f154d717796
      </CustomerID>
      <FirstName>Greg</FirstName>
      <LastName>Maddux</LastName>
      <Address>100 Henry Aaron Way</Address>

...

   </Customers>
</NewDataSet>
```

Результирующий XML-документ начинает приобретать законченный вид. Форматирование XML-файла совместно с определением способа представления столбцов является основным способом управления результирующим XML-кодом.

До сих пор в XML-файле сохранялись лишь данные объекта DataSet. Однако метод DataSet.WriteXml () поддерживает указание режима сохранения XML-файла (перечисление XmlWriteMode), позволяя выбрать один из трех форматов.

- IgnoreSchema **— используется по умолчанию. В файле XML сохраняются только данные объекта DataSet. Если данные не загружены, XML-файл не создается.**

- `WriteSchema` — то же, что и `IgnoreSchema` с добавлением информации о схеме в качестве встроенного XSD-документа.
- `DiffGram` — в файле XML сохраняются первоначальные и текущие данные объекта **DataSet**. Более подробно этот режим сохранения XML-файла рассматривается в разделе этой главы "Использование режима XmlWriteMode.DiffGram".

Параметр типа `XmlWriteMode` передается вторым во всех стандартных переопределенных вариантах метода `WriteXml()`, как показано в листинге 9.7.

Листинг 9.7. Определение режима сохранения данных объекта *DataSet* в формате XML

```
...

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter("test.xml");

// Вывод содержимого объекта DataSet в XML-файл.
dataSet.WriteXml(xmlWriter, XmlWriteMode.WriteSchema);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();
```

Использование режима XmlWriteMode.WriteSchema

Наиболее полное сохранение содержимого объекта **DataSet** (включая схему отношений, первичных ключей и данных) достигается в режиме WriteSchema. Следует отметить, что в этом случае данные будут представлены только одной копией — текущей. Если целью является сериализация объекта **DataSet** с сохранением состояния всех строк (в том числе измененных), стоит воспользоваться форматом DiffGram (см. следующий раздел). При заполнении другого объекта **DataSet** на основе полученного XML-кода все строки вновь созданного объекта **DataSet** будут считаться новыми, что может привести к потере информации об изменениях, которые должны быть отражены в базе данных,

Использование режима XmlWriteMode. DiffGram

В отличие от режима WriteSchema, режим DiffGram используется в том случае, когда необходимо сохранить *все* данные объекта DataSet (как первоначальные, так и текущие значения). DiffGram-документ не включает в себя схему; предполагается, что он будет использоваться для заполнения объекта **DataSet** с заранее определенной схемой. Как было отмечено выше, при сериализации объекта **DataSet** в XML-формате DiffGram сохраняются все данные, а не только изменения. Для того чтобы создать DiffGram-документ, состоящий только из изменений, нужно вызвать метод `DataSet.GetChanges()` и записать возвращенные этим методом данные в XML-файл, как показано в листинге 9.8.

Листинг 9.8. Создание XML-документа DiffGram

```
...

// Изменение нескольких строк.
invoiceItemTable.Rows[0]["Quantity"] = 5;

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter("test.xml");

// Создание объекта XmlTextWriter.
```

```
XmlTextWriter xmlWriter = new XmlTextWriter(writer) ;
xmlWriter.Formatting = Formatting.Indented;

// Получение измененной части объекта DataSet.
DataSet dataSetChanges = dataSet.GetChanges();

// Сохранение полученного объекта DataSet
// в XML-документе DiffGram.
dataSetChanges.WriteXml(xmlWriter, XmlWriteMode.DiffGram);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();
```

Более подробно использование формата DiffGram рассматривается в разделе этой главы "Стратегии использования XML-документа в формате DiffGram".

Пространства имен объекта DataSet

Классы DataSet, DataTable и DataColumn обладают свойствами, позволяющими указать пространство имен и префикс для создаваемого или считываемого XML-кода. Пример определения пространства имен "ADONET" и префикса "an" приведен в листинге 9.9.

Листинг 9.9. Определение пространства имен и префикса объекта DafaSef

```
...

// Определение пространства имен и
// префикса XML для объекта DataSet.
dataSet.Namespace = "ADONET";
dataSet.Prefix = "an";

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter ("test.xml") ;

// Создание объекта XmlTextWriter.
XmlTextWriter xmlWriter = new XmlTextWriter(writer);
xmlWriter.Formatting = Formatting.Indented;

// Сохранение объекта DataSet Б XML-файле.
dataSet.WriteXml(xmlWriter);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();
```

В результате выполнения приведенного выше фрагмента кода создается XML-файл, в котором элемент, соответствующий объекту **DataSet,** входит в указанное пространство имен и имеет указанный префикс. При желании вы можете определить пространства имен для объектов **DataTable и** DataColumn.

Заполнение объекта DataSet данными из XML-файла

Для заполнения объекта **DataSet** данными из XML-файла применяется метод DataSet.ReadXml(). Поскольку результат выполнения этого метода должен бить зеркальным отражением результата выполнения метода WriteXml(), необходимо, чтобы XML-документ, использующийся для заполнения объекта **DataSet,** соответствовал модели данных объекта **DataSet,** Другими словами, объект **DataSet** нельзя запол-

нить с помощью любого XML-документа, который вы случайно нашли в сети. **Ниже** приведен один из примеров XML-документа[1].

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <system.net>
    <webRequestModules>
      <add prefix="http"
           type="System.Net.HttpRequestCreator"/>
      <add prefix="https"
           type="System.Net.HttpRequestCreator"/>
      <add prefix="file"
           type="System.Net.FileWebRequestCreator"/>
    </webRequestModules>
    <authenticationModules>
      <add type="System.Net.DigestClient"/>
      <add type="System.Net.NegotiateClient"/>
      <add type="System.Net.KerberosClient"/>
      <add type="System.Net.NtlmClient"/>
      <add type="System.Net.BasicClient"/>
    </authenticationModules>
    <connectionManagement>
      <add address="*" maxconnection="2"/>
    </connectionManagement>
  </system.net>
</configuration>
```

Этот простой фрагмент XML-кода вызовет у объекта **DataSet** шок! Ниже приводится сообщение, которое будет выведено в результате попытки заполнения объекта **DataSet.**

```
The same table (add) cannot be the child table in two
nested relations.
```

Вот примерный перевод этого сообщения:

```
Одна и та же таблица (add) не может быть дочерней в двух
вложенных отношениях.
```

Дело в том, что для объекта **DataSet** наличие двух элементов add различного типа является неожиданностью. Поначалу он попытается создать таблицу Add, но из-за того что различные элементы add содержатся в элементах более высокого уровня, объект **DataSet** не сможет создать отношение и "сдастся". Единственный способ загрузить представленный выше XML-код в объект **DataSet** — привести XML-документ к более дружественной по отношению к объекту **DataSet** схеме.

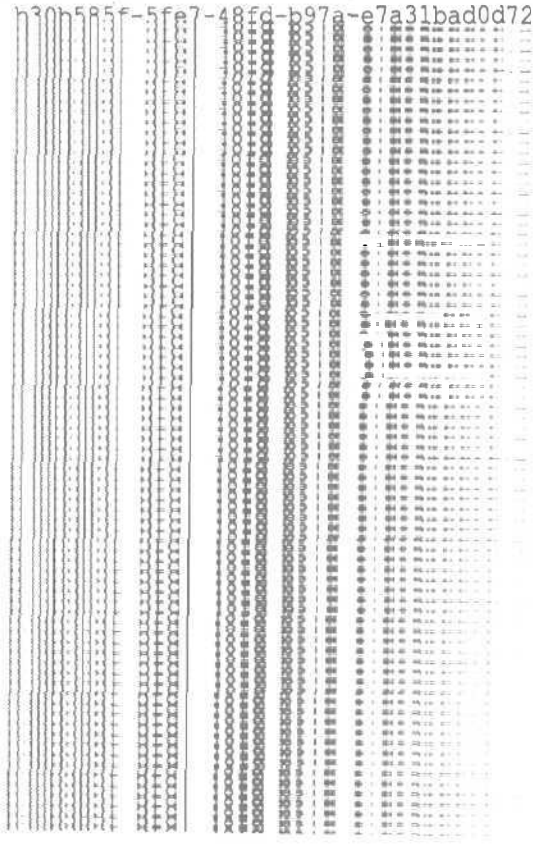
В следующем примере XML-кода сущности определены значительно лучше.

```
<?xml version="1.0" encoding="UTF-8" ?>
<NewDataSet>
  <Customers>
    <CustomerID>
      11d59cb7-bf61-4540-9317-4f154d717796
    </CustomerID>
    <FirstName>Greg</FirstName>
    <LastName>Maddux</LastName>
    <Address>53 PeachtreeCenter</Address>
    <City>Atlanta</City>
```

[1] **Это фрагмент файла** machine.config, **который устанавливается вместе с** .NET **Framework**

```
<State>GA</State>
<Zip>30342</Zip>
<HomePhone>404-555-1234</HomePhone>
<Stamp>AAAAAAAAAH8=</Stamp>
<CheckedOut>false</CheckedOut>
<Invoices>
  <InvoiceID>
    463cdf10-2609-41f3-99a3-05f4cedf9fe2
  </InvoiceID>
  <InvoiceNumber>100</InvoiceNumber>
  <InvoiceDate>
    2002-03-21T00:00:00.0000000-05:00
  </InvoiceDate>
  <Terms>Net 10th</Terms>
  <PO>100042</PO>
  <CustomerID>
    11d59cb7-bf61-4540-9317-4f154d717796
  </CustomerID>
  <InvoiceItems>
    <InvoiceItemID>
       c7395127-891d-4ed4-8c37-d4e73bcfdf55
    </InvoiceItemID>
```

```
    <InvoiceID>
      463cdf10-2609-41f3-99a3-05f4cedf9fe2
    </InvoiceID>
    <ProductID>4</ProductID>
    <Quantity>2</Quantity>
    <Discount>4.25</Discount>
  </InvoiceItems>
  <InvoiceItems>
    <InvoiceItemID>
      h30b585f-5fe7-48fd-b97a-e7a31bad0d72
```



```
if (node != null)
{
  Console.WriteLine("Found:");
  Console.WriteLine(node.OuterXml);
}
else
{
  Console.WriteLine("No Nodes found");
}
```

Выполнение этого фрагмента кода приведет к следующему результату:

объектов **Binding.** Каждый объект данной коллекции описывает привязку данных к

```
      <State>GA</State>
      <Zip>30342</Zip>
      <HomePhone>404-555-1234</HomePhone>
      <Stamp>AAAAAAAAAH8=</Stamp>
      <CheckedOut>false</CheckedOut>
      <Invoices>
        <InvoiceID>
          463cdf10-2609-41f3-99a3-05f4cedf9fe2
        </InvoiceID>
        <InvoiceNumber>100</InvoiceNumber>
        <InvoiceDate>
          2002-03-21T00:00:00.0000000-05:00
        </InvoiceDate>
        <Terms>Net 10th</Terms>
        <PO>100042</PO>
        <CustomerID>
          11d59cb7-bf61-4540-9317-4f154d717796
        </CustomerID>
        <InvoiceItems>
          <InvoiceItemID>
             c7395127-891d-4ed4-8c37-d4e73bcfdf55
          </InvoiceItemID>
          <InvoiceID>
            463cdf10-2609-41f3-99a3-05f4cedf9fe2
          </InvoiceID>
          <ProductID>4</ProductID>
          <Quantity>2</Quantity>
          <Discount>4.25</Discount>
        </InvoiceItems>
        <InvoiceItems>
          <InvoiceItemID>
            b30b585f-5fe7-48fd-b97a-e7a31bad0d72
          </InvoiceItemID>
          <InvoiceID>
            463cdf10-2609-41f3-99a3-05f4cedf9fe2
          </InvoiceID>
          <ProductID>2</ProductID>
          <Quantity>3</Quantity>
          <Discount>0</Discount>
        </InvoiceItems>
      </Invoices>
    </Customers>
</NewDataSet>
```

Теперь идентифицировать сущности не составит труда. Элемент Customer содержит элемент Invoices, который, в свою очередь, содержит элемент InvoiceItems. Пример заполнения объекта DataSet с помощью этого XML-файла приведен в листинге 9.10.

Листинг 9.10. Считывание содержимого XML-файла в объект *DataSet*

```
// Заполнение объекта DataSet на основе XML-документа.
DataSet dataSet = new DataSet();
dataSet.ReadXml(@"good.xml");

Console.WriteLine(dataSet.GetXml());
```

В реальном мире метод ReadXml(), как правило, используется для загрузки содержимого XML-документа, полученного в результате сериализации этого же объекта **DataSet,** как показано в листинге 9.11.

Листинг 9.11. Считывание содержимого XML-файла в объект DataSet

```
...

// Создание объекта StreamWriter.
StreamWriter writer - new StreamWriter("test.xml");

// Создание объекта XmlTextWriter.
XmlTextWriter xmlWriter = new XmlTextWriter(writer);

// Сохранение содержимого объекта DataSet в XML-файле.
dataSet.WriteXml(xmlWriter);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();

// Заполнение объекта DataSet на основе XML-файла.
DataSet newDataSet = new DataSet();
newDataSet.ReadXml(@"test.xml");

// Сравнение двух объектов DataSet и вывод результата.
bool areTheySame = dataSet.GetXml() == newDataSet.GetXml();

Console.WriteLine("Are they the same? {0}", areTheySame);
```

Итак, если XML-код был получен на основе неизмененного объекта **DataSet,** он может быть использован для создания точной копии этого объекта.

Как и метод DataSet.WriteXml(), метод DataSet.ReadXml() имеет множество переопределенных вариантов, соответствующих различным режимам считывания XML-документа. Для обозначения режима считывания используется перечисление XmlReadMode, элементы которого приведены ниже.

- Auto — это значение используется по умолчанию. Объект **DataSet** попытается самостоятельно определить лучший метод считывания XML-кода. **Если** же этот метод известен заранее, его явное указание приведет к увеличению быстродействия.
- DiffGram — изменения, отраженные в XML-документе формата DiffGram, считываются и вносятся в объект **DataSet.** Состояние строк (свойство RowState) объекта **DataSet** при этом сохраняется. Действия, происходящие в режиме DiffGram, напоминают метод DataSet.Merge().
- Fragment — считывание XML-документа. Все встроенные пространства имен воспринимаются как часть схемы. Данный режим используется при чтении стандартных XML-документов, таких, как документы, созданные в результате применения предложения SQL Server FOR XML.
- IgnoreSchema — напоминает режим Fragment, отличаясь лишь тем, что встроенная схема игнорируется, а вместо нее используется текущая схема объекта **DataSet.** Все данные, которые не соответствуют существующей схеме, отбрасываются, включая данные из другого пространства имен. Если объект **DataSet** не содержит схемы, будут отброшены все данные.

- `InferSchema` — напоминает режим `IgnoreSchema`, однако схема объекта DataSet выводится на основе структуры данных, содержащихся в XML-документе. Если объект DataSet уже содержит схему, она при необходимости расширяется.
- `ReadSchema` — считываются встроенная схема и данные, после чего осуществляется попытка добавления считанной схемы к схеме объекта **DataSet**. В данном режиме, в отличие от режима Fragment, при существовании схемы в объекте DataSet генерируется исключение.

Стратегии использования XML-документа в формате DiffGram

XML-документы в формате DiffGram занимают особое место в ADO.NET. Как указывалось ранее, DiffGram-документ содержит всю информацию о данных объекта **DataSet,** включая первоначальное и текущее значение строк. Эти сведения можно использовать в двух целях — для проведения слияния объектов **DataSet** и для обновления базы данных.

Для того чтобы провести слияние двух объектов DataSet, сначала нужно получить изменения одного объекта **DataSet** в формате DiffGram, а затем применить их к другому объекту **DataSet.** Эта функциональность идентична функциональности метода `DataSet.Merge()`. Последний удобно применять в том случае, когда необходимо слияние двух локальных объектов **DataSet,** а DiffGram-документ— при необходимости использования Web-службы или очереди сообщений для отправки изменений на средний уровень. Следует отметить, что применение формата DiffGram связано с определенными трудностями (листинг 9.12).

Листинг 9.12. Использование **DiffGram-документа** для обмена информацией об изменениях между двумя объектами *DataSet*

```
...

// Определение переменных для упрощения доступа к таблицам.
DataTable customerTable    = dataSet.Tables["Customers"];
DataTable invoiceTable     = dataSet.Tables["Invoices"];
DataTable invoiceItemTable = dataSet.Tables["InvoiceItems"];

// Определение первичных ключей.
customerTable.PrimaryKey = new DataColumn[]
  { customerTable.Columns["CustomerID"] };
invoiceTable.PrimaryKey = new DataColumn[]
  { invoiceTable.Columns["InvoiceID"] };
invoiceItemTable.PrimaryKey = new DataColumn[]
  { invoiceItemTable.Columns["InvoiceItemID"] };

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter("test.xml");

// Создание объекта XmlTextWriter.
XmlTextWriter xmlWriter = new XmlTextWriter(writer);

// Сохранение содержимого объекта в XML-файле.
dataSet.WriteXml(xmlWriter, XmlWriteMode.WriteSchema);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();
```

```
// Создание точной копии первоначального объекта DataSet
// с помощью считывания XML-документа.
DataSet newDataSet = new DataSet();
newDataSet.ReadXml(@"test.xml", XmlReadMode.ReadSchema);
newDataSet.AcceptChanges();

// Изменение обоих объектов DataSet.
dataSet.Tables["Customers"].Rows[0]["Zip"] = "98765";
newDataSet.Tables["Customers"].Rows[1]["State"] = "FL";

// Создание объектов типа XmlTextWriter и Stream.
Stream newStream = new MemoryStream() as Stream;
XmlTextWriter xmlNewWriter =
                          new XmlTextWriter(newStream, null);
xmlNewWriter.Formatting = Formatting.Indented;

// Создание DiffGram-документа на основе объекта DataSet.
newDataSet.GetChanges().WriteXml(xmlNewWriter,
                                 XmlWriteMode.DiffGram);

// Вывод DiffGram-документа на консоль.
newStream.Position = 0;
StreamReader rdr = new StreamReader(newStream);
Console.WriteLine(rdr.ReadToEnd());

// Сброс текущей позиции в потоке.
newStream.Position = 0;

// Применение изменений к объекту DataSet.
dataSet.ReadXml(newStream, XmlReadMode.DiffGram);
```

Узнать о том, что строку DiffGram-документа необходимо добавить в **объект DataSet,** метод DataSet.ReadXml О может лишь с помощью первичных ключей, позволяющих провести идентификацию строки. При созданий копии объекта **DataSet** посредством считывания XML-документа, созданного на основе исходного объекта **DataSet,** используется метод DataSet.AcceptChanges(), который переводит все строки в состояние Unchanged. При желании этого можно избежать, создав копию объекта **DataSet** с помощью XML-файла в формате DiffGram. Следует отметить, что для формата DiffGram характерна детализация на уровне строк. Так, если изменения строки содержатся как в объекте **DataSet,** так и в DiffGram-документе, то при считывании последнего изменения будут перезаписаны. Для того чтобы сделать слияние как можно более эффективным, файл в формате DiffGram создается на основе измененного объекта **DataSet** посредством вызова метода WriteXml() по отношению к результату вызова метода GetChanges(). В итоге полученный DiffGram-документ будет содержать только изменения объекта **DataSet.** Если бы DiffGram-документ создавался на основе всего объекта DataSet, то при слиянии осуществлялась бы попытка обновления каждой строки. Несмотря на то что в приведенном примере это обстоятельство не имело бы особого значения, при увеличении размеров объекта **DataSet** снижение производительности стало бы более заметным.

Схема объекта DataSet

В предыдущих разделах было показано, что встроенная схема базы данных может быть добавлена в XML-код, создаваемый на основе объекта **DataSet,** хотя для этого потребуются определенные усилия. В связи с этим возникает вопрос: можно ли получить от объекта **DataSet** лишь схему?

Сохранение XSD-файла на основе объекта DataSet

XSD-файл можно создать, вызвав один из предназначенных для этого методов класса DataSet. Так же, как и при считывании и записи XML-файла, здесь существуют Get-, Read- и Write-методы.

- **GetXmlSchema — возвращает строку, содержащую информацию о схеме (XSD-документ) объекта DataSet.**
- **ReadXmlSchema — считывает информацию о схеме из указанного файла, объекта XmlReader или объекта Stream.**
- **WriteXmlSchema — записывает информацию о схеме в объект XmlWriter, объект Stream или в указанный файл.**

Для того чтобы сохранить XSD-файл и загрузить его в другой объект **DataSet,** используется код, приведенный в листинге 9.13.

Листинг 9.13. Сохранение XSD-документа и считывание схемы объекта *DataSet* из него

```
...

// Создание объекта StreamWriter.
StreamWriter writer = new StreamWriter("test.xsd");

// Создание объекта XmlTextWriter.
XmlTextWriter xmlWriter = new XmlTextWriter(writer);

// Сохранение XSD-файла.
dataSet.WriteXmlSchema(xmlWriter);

// Закрытие объекта XmlTextWriter.
xmlWriter.Close();

// Создание объекта DataSet с той же схемой,
// что была у исходного объекта DataSet.
DataSet newDataSet = new DataSet();
newDataSet.ReadXmlSchema(@"test.xsd");
```

Ниже приведено содержимое XSD-файла, созданного с помощью метода `WriteXmlSchema()`.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="NewDataSet"
           xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata=
                    "urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="NewDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Customers">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CustomerID"
                          msdata:DataType="System.Guid"
                          type="xs:string"
                          minOccurs="0" />
              <xs:element name="FirstName"
                          type="xs:string"
                          minOccurs="0" />
```

```
            <xs:element name="LastName"
                        type="xs: string"
                        minOccurs="0" />
            <xs:element name="MiddleName"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="Address"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="Apartment"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="City"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="State"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="Zip"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="HomePhone"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="BusinessPhone"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="DOB"
                        type="xs:dateTime"
                        minOccurs="0" />
            <xs:element name="Discount"
                        type="xs:double"
                        minOccurs="0" />
            <xs:element name="Stamp"
                        type="xs:base64Binary"
                        minOccurs="0" />
            <xs:element name="CheckedOut"
                        type="xs:boolean"
                        minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Invoices">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="InvoiceID"
                        msdata:DataType="System.Guid"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="InvoiceNumber"
                        type="xs:int"
                        minOccurs="0" />
            <xs:element name="InvoiceDate"
                        type="xs:dateTime"
                        minOccurs="0" />
            <xs:element name="Terms"
                        type="xs:string"
                        minOccurs="0" />
```

```
            <xs:element name="FOB"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="PO"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="CustomerID"
                        msdata:DataType="System.Guid"
                        type="xs:string"
                        minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="InvoiceItems">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="InvoiceItemID"
                        msdata:DataType="System.Guid"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="InvoiceID"
                        msdata:DataType="System.Guid"
                        type="xs:string"
                        minOccurs="0" />
            <xs:element name="ProductID"
                        type="xs:int"
                        minOccurs="0" />
            <xs:element name="Quantity"
                        type="xs:int"
                        minOccurs="0" />
            <xs:element name="Discount"
                        type="xs:double"
                        minOccurs="0" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Класс XmlDataDocument

В предыдущих разделах этой главы было показано, как получить XML-файл на основе объекта **DataSet**. Если же вам необходимо манипулировать с данными объекта **DataSet**, представленными в виде XML-кода (выполнять запросы XPath, преобразования XSLT и другие операции), создайте объект XmlDocument и продолжайте работать с ним. Следует отметить, что ADO.NET предлагает еще более оптимальное решение — использование класса XmlDataDocument. Этот класс содержит ссылку на объект DataSet и предоставляет все его данные в виде XML-документа. Манипуляция объектом XmlDataDocument приводит к соответствующим изменениям в объекте DataSet и наоборот.

Класс XmlDataDocument непосредственно наследует класс XmlDocument, а поэтому его экземпляр имеет такую же функциональность, как и экземпляр класса XmlDocument. Наиболее существенная разница между двумя классами заключается в том, что в классе XmlDataDocument объект DataSet необходимо указывать в виде

свойства (в конструкторе или уже после создания экземпляра класса). После создания объекта XmlDataDocument его можно использовать для перемещения по узлам, поиска информации с помощью выражений XPath или проведения XML-трансформаций посредством языка XSLT (листинг 9.14).

Листинг 9.14. Использование объекта *XmlDataDocument*

```
// Создание и открытие соединения.
SqlConnection conn = new SqlConnection(
  "server=localhost;" +
  "database=ADONET;" +
  "Integrated Security=true;");
conn.Open();

// Создание объекта DataAdapter для каждой
// таблицы, извлекаемой из базы данных.
SqlDataAdapter daCustomers = new
  SqlDataAdapter("SELECT * FROM CUSTOMER;",
                 conn);
SqlDataAdapter daInvoices = new
  SqlDataAdapter("SELECT * FROM INVOICE;",
                 conn);
SqlDataAdapter dalnvoiceltems = new
  SqlDataAdapter("SELECT * FROM INVOICEITEM;",
                 conn);

// Создание пустого объекта DataSet.
DataSet dataSet = new DataSet();

// Заполнение объекта DataSet с помощью объектов DataAdapter.
daCustomers.Fill(dataSet, "Customers");
daInvoices.Fill(dataSet, "Invoices");
daInvoiceItems.Fill(dataSet, "InvoiceItems");

// Закрытие соединения.
conn.Close();

// Определение переменных для упрощения доступа к таблицам.
DataTable customerTable    = dataSet.Tables["Customers"];
DataTable invoiceTable     = dataSet.Tables["Invoices"];
DataTable invoiceItemTable = dataSet.Tables["InvoiceItems"];

// Определение первичных ключей.
customerTable.PrimaryKey = new DataColumn[]
  { customerTable.Columns["CustomerID"] };
invoiceTable.PrimaryKey = new DataColumn[]
  { invoiceTable.Columns["InvoiceID"] };
invoiceItemTable.PrimaryKey = new DataColumn[]
  { invoiceItemTable.Columns["InvoiceItemID"] };

// Определение отношений.

// Создание первого отношения (между таблицами Invoices и
// InvoiceItems). При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
dataSet.Relations.Add( "Invoices_InvoiceItems",
  dataSet.Tables["Invoices"].Columns["InvoiceID"],
  dataSet.Tables["InvoiceItems"].Columns["InvoiceID"],
  true);
```

```
// Создание второго отношения (между
// таблицами Customers и Invoices).
// При создании отношения определяется
// ограничение, гарантирующее, что отношение
// создается на основе первичных ключей таблиц.
dataSet.Relations.Add( "Customers_Invoices",
  dataSet.Tables["Customers"].Columns["CustomerID"],
  dataSet.Tables["Invoices"].Columns["CustomerID"],
  true);

// Создание объекта XmlDataDocument.
XmlDataDocument dataDoc = new XmlDataDocument(dataSet);

// Выполнение запроса XPath.
XmlNode node = dataDoc.SelectSingleNode("//Invoices");

// Вывод XML-кода первого найденного узла.
Console.WriteLine(node.OuterXml);
```

В этом коде создается экземпляр класса XmlDataDocument, конструктору которого передается в качестве параметра объект DataSet. После создания объекта XmlDataDocument выполняется XPath-запрос, возвращающий первого потомка с именем Invoices.

```
<Invoices>
  <InvoiceID>
    463cdf10-2609-41f3-99a3-05f4cedf9fe2
  </InvoiceID>
  <InvoiceNumber>100</InvoiceNumber>
  <InvoiceDate>
    2002-03-21T00:00:00.0000000-05:00
  </InvoiceDate>
  <Terms>Net 30</Terms>
  <PO>100042</PO>
  <CustomerID>
    11d59cb7-bf61-4540-9317-4f154d717796
  </CustomerID>
</Invoices>
```

Поиск данных в объекте DataSet с помощью запросов XPath

В классе DataSet отсутствует механизм запросов SQL, в результате чего для поиска данных, содержащихся в объекте DataSet, разработчики используют фильтры и средства поиска класса DataView. В принципе класс Dataview хорошо справляется со своей задачей, но поскольку мы располагаем классом XmlDataDocument, было бы вполне естественно попытаться использовать его язык запросов XPath для поиска данных в объекте DataSet. Продолжая рассматривать предыдущий пример, попытаемся применить стандартный механизм поиска XPath для нахождения первого клиента с именем "Тот", как показано в листинге 9.15.

Листинг 9.15. Поиск данных в объекте *DataSet* с помощью выражений XPath

```
...

// Создание объекта XmlDataDocument.
XmlDataDocument dataDoc = new XmlDataDocument(dataSet);

// Поиск в объекте XmlDataDocument.
string xpath = "//Customers[FirstName = \"Tom\"]";
XmlNode node = dataDoc.SelectSingleNode(xpath);
```

```
if (node != null)
{
  Console.WriteLine("Found:");
  Console.WriteLine(node.OuterXml);
}
else
{
  Console.WriteLine("No Nodes found");
}
```

Выполнение этого фрагмента кода приведет к следующему результату:

```
Found:
<Customers>
  <CustomerID>
    80fce1c0-176f-4367-aefa-66deb58e6bed
  </CustomerID>
  <FirstName>Tom</FirstName>
  <LastName>Glavine</LastName>
  <Address>100 Henry Aaron Way</Address>
  <City>Atlanta</City>
  <State>GA</State>
  <Zip>30307</Zip>
  <HomePhone>404-555-1235</HomePhone>
  <Stamp>AAAAAAAAAGo=</Stamp>
  <CheckedOut>false</CheckedOut>
</Customers>
```

Поскольку объект **XmlDataDocument** практически аналогичен объекту XmlDocument, над ним можно выполнять все XML-операции. Преимущество такого подхода заключается в экономии времени, поскольку не нужно его тратить на создание и анализ XML-документа. Документ, содержащийся внутри объекта **XmlDataDocument,** непосредственно указывает на данные, хранящиеся в объекте **DataSet.**

Более подробная информация об использовании выражений XPath приведена **в документации MSDN по адресу:** msdn.microsoft.com/library/en-us/cpguide/html/cpconcompileselectevaluatematcheswithxpathxpathexpressions.asp.

Пространства имен и поиск данных в объекте DataSet

Класс DataSet поддерживает свойство, позволяющее указать пространство имен объекта **DataSet.** Обычно в этом нет необходимости, однако когда объект **DataSet** сохраняется с известной **XSD-схемой,** все происходит по-другому. Чаще всего пространства имен определяются для экземпляров типизированных классов DataSet, поскольку типизированные классы DataSet основаны на XSD-документах.

Поиск данных с помощью языка XPath в ассоциированном пространстве имен вызывает некоторые затруднения; это касается не только ADO.NET или класса DataSet. В большинстве XSD-документов определено пространство имен, использующееся по умолчанию. К сожалению, язык XPath не поддерживает поиск в нем. Для того чтобы провести поиск в объекте XmlDataDocument, с которым ассоциировано стандартное пространство имен, необходимо воспользоваться объектом XmlNamespaceManager, позволяющим указать пространство имен и создать псевдоним для пространства имен, используемого по умолчанию (листинг 9.16).

Листинг 9.16. Выполнение запроса XPath с использованием префикса пространства имен

```
...

// Создание объекта XmlDataDocument.
XmlDataDocument dataDoc = new XmlDataDocument(dataSet);
```

```
// Создание диспетчера пространств имен.
XmlNamespaceManager nsMgr;
nsMgr = new XmlNamespaceManager(dataDoc.NameTable);

// Добавление пространства имен.
nsMgr.AddNamespace("x", dataSet.Namespace);

// Выполнение поиска в объекте XmlDataDocument.
string xpath = "//x:Customers";
XmlNode node = dataDoc.SelectSingleNode(xpath, nsMgr);

if (node != null)
{
  Console.WriteLine("Found:");
  Console.WriteLine(node.OuterXml);
}
else
{
  Console.WriteLine("No Nodes found");
}
```

Для того чтобы иметь возможность проводить поиск в объекте **XmlDataDocument**, необходимо создать экземпляр класса XmlNamespaceManager, передав в качестве параметра его конструктору таблицу имен (NameTable) объекта **XmlDataDocument.** Пчеле этого можно добавить пространство имен объекта **DataSet,** назначив ему псевдоним x. При создании запроса XPath сущность нужно идентифицировать с помощью префикса — - псевдонима пространства имен (//x:customers). Наконец, объект **XmlNamespaceManager** и запрос XPath необходимо передать методу SelectSingleNode.

Преобразование объекта DataSet с помощью языка XSLT

Использование языка XSLT для преобразования XML-кода объекта **DataSet** предельно упростилось с созданием объекта **XmlDataDocument**. Как и в предыдущем примере, мы существенно экономим время за счет пропуска нескольких этапов, которые в обычной ситуации являются необходимыми (сохранение XML-файла и его анализ). Для того чтобы провести XSLT-преобразование, нужно выполнить действия, приведенные в листинге 9.17.

Листинг 9.17. XSLT-преобразование объекта *XmlDataDocument*

```
using System.Xml.Xsl;

...

// Создание объекта XmlDataDocument.
XmlDataDocument dataDoc = new XmlDataDocument(dataSet);

// Создание объекта StringWriter
// для хранения результатов преобразования.
StringWriter writer = new StringWriter();

// Выполнение XSLT-преобразования.
XslTransform xslt = new XslTransform();
xslt.Load(@"XmlDataDocSample.xsl");
xslt.Transform(dataDoc, null, writer);

// Вывод результатов XSLT-преобразования на консоль.
Console.WriteLine(writer.ToString());
```

Как и в остальных случаях, этот код не меняется при проведении преобразования объекта **XmlDocument**. Вообще говоря, XSLT-преобразования XSL — это сложная тема. Более подробную информацию по этому вопросу можно получить по адресу: `msdn.microsoft.com/library/en-us/cpguide/html/cpconxslttransformationswithxsltransformclass.asp`.

Резюме

Ядро интеграции ADO.NET и XML — это, безусловно, класс DataSet. Заполняя объект **DataSet** на основе XML-документа или применяя последний для определения схемы объекта **DataSet**, мы используем универсальный язык обмена данными между пользовательскими приложениями и окружающим миром. Кроме того, класс `XmlDataDocument` позволяет рассматривать объект **DataSet** как экземпляр класса XmlDocument, что позволяет проводить над ним все известные XML-операции.

Глава 10

Привязка данных в ADO. NET

В этой главе...

Что такое привязка данных в .NET?

Привязка данных в Windows-формах

Привязка данных в ASP.NET

В предыдущих главах мы рассматривали способы извлечения информации из источника данных. Следующим шагом является предоставление этой информации пользователю через интерфейсы, с которыми он привык ежедневно "общаться". Именно здесь возникает необходимость в привязке данных.

Что такое привязка данных в .NET?

Компания Microsoft знала, что разработчики основанных на взаимодействии с базами данных приложений тратят большую часть своего времени на создание экранов ввода и отчетов. Она также знала, что создание простого способа связывания элементов управления и информации из базы данных сэкономит программистам много времени. Повышение эффективности труда разработчиков является основной целью привязки данных в .NET и, как следствие, в ADO.NET.

Существовавшие ранее механизмы привязки данных Microsoft позволяли достаточно легко связывать фактические значения в элементах управления со значениями, хранящимися в базе данных. К сожалению, подобная функциональность не была предусмотрена для Web-приложений. Средства привязки данных .NET доступны Web-разработчикам а форме, очень напоминающей форму аналогичных средств для настольных приложений.

Механизм привязки данных .NET должен обеспечивать больше, чем просто доставку данных пользователю, а именно: он должен поддерживать управляемые данными настольные приложения и Web-приложения. С этой целью механизм привязки данных позволяет связать с источником данных любое свойство элемента управления, предназначенное для записи. Например, если у вас есть база данных, содержащая информацию о цвете, размере и положении элемента управления, то механизм привяжи данных легко справится с помещением этой информации в элемент управления,

Привязка данных в Windows-формах

До появления .NET большинство механизмов управления оконным интерфейсом (MFC, Visual Basic и т.д.) поддерживали в той или иной степени привязку данных. В этом смысле Windows-формы не представляют собой ничего нового — они обеспечивают привязку любых данных к любому свойству элемента управления,

Windows-формы поддерживают два типа привязки данных. Для элементов управления, содержащих единственное значение (таких, как `TextBox`, `CheckBox` и т.п.), Windows-формы поддерживают простую привязку данных, а для элементов управления, содержащих несколько значений (таких, как ListBox, `ComboBox`, DataGrid и т.д.) — сложную привязку данных.

Прежде чем с головой погрузиться в различные вопросы привязки данных, необходимо получить те сами данные, к которым потом будет осуществляется привязка. В листинге ЮЛ приведен код, с помощью которого из базы данных извлекается информация, которая будет использоваться в примерах этой главы.

Листинг 10.1. Код, извлекающий информацию из базы данных

```
private DataSet dataSet = new DataSet();
private DataTable customerTable;
private DataTable invoiceTable;
private DataTable invoiceItemTable;
private DataTable productTable;

private void InitializeData()
{
  // Создание и открытие соединения.
  SqlConnection conn = new SqlConnection(
    "server=localhost;" +
    "database=ADONET;" +
    "Integrated Security=true;");

  // Создание объекта DataAdapter для каждой
  // таблицы, извлекаемой из базы данных.
  SqlDataAdapter daCustomers =
      new SqlDataAdapter("SELECT * FROM CUSTOMER", conn);
  SqlDataAdapter daInvoices =
      new SqlDataAdapter("SELECT * FROM INVOICE", conn);
  SqlDataAdapter daInvoiceItems =
      new SqlDataAdapter("SELECT * FROM INVOICEITEM", conn);
  SqlDataAdapter daProducts =
      new SqlDataAdapter("SELECT * FROM PRODUCT", conn);

  // Создание пустого объекта DataSet.
  DataSet dataSet = new DataSet();

  // Заполнение объекта DataSet
  // с помощью объектов DataAdapter.
  daCustomers.Fill(dataSet, "Customers");
  daInvoices.Fill(dataSet, "Invoices");
  daInvoiceItems.Fill(dataSet, "InvoiceItems");
  daProducts.Fill(dataSet, "Products");

  // Определение переменных для
  // упрощения доступа к таблицам.
  customerTable    = dataSet.Tables["Customers"];
  invoiceTable     = dataSet.Tables["Invoices"];
  invoiceltemTable = dataSet.Tables["InvoiceItems"];
  productTable     = dataSet.Tables["Products"];

  // Определение первичных ключей.
  customerTable.PrimaryKey = new DataColumn[]
              { customerTable.Columns["CustomerID"] };
  invoiceTable.PrimaryKey = new DataColumn[]
              { invoiceTable.Columns["InvoiceID"] };
  invoiceItemTable.PrimaryKey = new DataColumn[]
              { invoiceItemTable.Columns["InvoiceItemID"] };
  productTable.PrimaryKey = new DataColumn[]
              { productTable.Columns["ProductID"] };
```

```
    // Определение отношений.
    // Создание первого отношения (между таблицами Invoices к
    // InvoiceItems). При создании отношения определяется
    // ограничение, гарантирующее, что отношение
    // создается на основе первичных ключей таблиц.
    dataSet.Relations.Add( "Invoices_InvoiceItems",
      dataSet.Tables["Invoices"].Columns["InvoiceID"],
      dataSet.Tables["InvoiceItems"].Columns["InvoiceID"],
      true);

    // Создание второго отношения (между
    // таблицами Customers и Invoices).
    // При создании отношения определяется
    // ограничение, гарантирующее, что отношение
    // создается на основе первичных ключей таблиц.
    dataSet.Relations.Add( "Customers_Invoices",
      customerTable.Columns["CustomerID"],
      invoiceTable.Columns["CustomerID"],
      true);

    // Создание третьего отношения (между
    // таблицами Invoices и Products).
    // При создании отношения определяется
    // ограничение, гарантирующее, что отношение
    // создается на основе первичных ключей таблиц.
    dataSet.Relations.Add( "InvoiceItems_Products",
      invoiceItemTable.Columns["ProductID"],
      productTable.Columns["ProductID"],
      false);

    // Определение нескольких вычисляемых столбцов.
    customerTable.Columns.Add("FullName", typeof(string),
                              "LastName + ', ' + FirstName");
    invoiceItemTable.Columns.Add("Price", typeof(string),
                                 "Avg(Child.Price)");
    invoiceItemTable.Columns.Add("Total", typeof(string),
                                 "Quantity * Avg(Child.Price)");
    invoiceItemTable.Columns.Add("Description",
                                 typeof(string),
                                 "MAX(Child.Description)");
    invoiceTable.Columns.Add("Description", typeof(string),
                             "InvoiceNumber + ': ' + InvoiceDate");
}
```

Большая часть приведенного выше должна быть вам хорошо знакома. Обратите внимание на создание нескольких новых столбцов, к которым будет осуществляется привязка. Это так называемые вычисляемые столбцы (или столбцы-выражения), с которыми очень легко связывать элементы управления, поскольку формат данных, содержащихся в этих столбцах, определяется автоматически.

Простая привязка данных

Для того чтобы осуществить привязку к данным элемента управления Windows-форм, необходимо добавить новый объект **Binding** в коллекцию DataBindings элемента управления. Свойство DataBindings определено в базовом классе Control, от которого порождены все классы Windows-форм, и предоставляет доступ к коллекции

объектов Binding. Каждый объект данной коллекции описывает привязку данных к определенному свойству элемента управления. Например, значение свойства Text может определяться данными из одной таблицы, а цвет текста элемента управления — данными из другой таблицы.

Для .того чтобы создать объект привязки данных, необходимо выполнить следующие действия:

- указать свойство элемента управления, к которому необходимо осуществить привязку;
- указать источник данных;
- указать при необходимости часть источника данных.

В приложениях ADO.NET свойство `dataSource` обычно указывает на объект **DataTable** или **DataView**, а свойство `dataMember` — на столбец, данные которого необходимо извлечь. Создание объекта привязки данных приобретает следующий формат:

```
{Элемент_управления}.DataBindings.Add("{Свойство}",
                                      {dataSource},
                                      "{dataMember}");
```

Предположим, что существует таблица, которая содержит информацию об определенном товаре, хранящемся на складе. В листинге 10.2 приведен пример привязки описания товара к текстовому полю, а информации о его доступности — к флажку.

Листинг 10.2. Простая привязка данных к элементам управления Windows-форм

```
// Привязка текстового поля к описанию товара.
txtDesc.DataBindings.Add("Text",
                         productTable,
                         "Description");

// Привязка флажка к столбцу InStock.
chkInStock.DataBindings.Add("Checked",
                            productTable,
                            "InStock");
```

В результате выполнения приведенного выше фрагмента кода диалоговое окно примет вид, показанный на рис. 10.1.

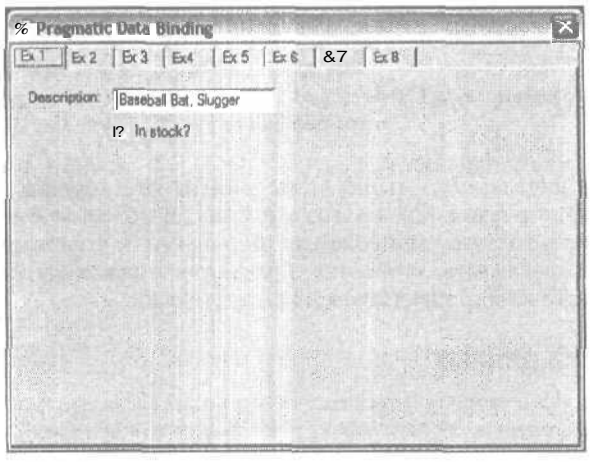


Рис. 10.1. Простая привязка данных

При осуществлении привязки данных должен быть способ указания текущей записи источника данных. Более подробно этот вопрос рассматривается в разделе "Использование класса CurrencyManager", далее в этой главе.

Сложная привязка данных

На самом деле сложная привязка данных (как ее называет Microsoft) не так уж сложна. Под этим термином Microsoft подразумевает привязку данных к списку или коллекции элементов[1], что очень удобно в контексте манипулирования информацией, хранящейся в базе данных. Вновь обратимся к таблице базы данных, хранящей информацию о товарах. Чтобы показать все товары, которые когда-либо были в наличии, необходимо связать таблицу со списком. Для этого нужно определить значения трех свойств объекта **ListBox**.

- DataSource — экземпляр класса, реализующего интерфейс `IList` (например, класс DataTable и большинство классов коллекций).
- DisplayMember — свойство объекта-источника (DataSource), которое будет отображаться в элементе управления.
- `ValueMember` — идентификатор, хранящийся в элементе управления и определяющий строку объекта-источника, к которой происходит обращение (например, значение свойства DisplayMember может быть равно `FullName`, а значение свойства ValueMember — `CustomerID`).

В приведенном ниже примере мы хотим предоставить пользователю описание товара и при этом иметь возможность узнать значение поля `ProductID` на тот случай, если пользователь выберет один из товаров (листинг 10.3).

Листинг 10.3. Сложная привязка данных к элементам управления Windows-форм

```
// Привязка таблицы товаров к элементу управления ListBox.
listProducts.DataSource = productTable;
listProducts.DisplayMember = "Description";
listProducts.ValueMember = "ProductiD";
```

В результате выполнения приведенного выше фрагмента кода диалоговое окно примет вид, показанный на рис. 10.2.

Для того чтобы убедиться в привязке значения поля ProductiD к определенному свойству элемента управления ListBox, добавим следующий код, отображающий на экране диалоговое окно со значением поля ProductiD при щелчке на кнопке.

```
MessageBox.Show(String.Format( "Selected Product's ID: {0}",
                               listProducts.SelectedValue));
```

Как показано выше, значение поля ProductiD (указанного в свойстве элемента управления ValueMember), соответствующее выбранному пользователем элементу, хранится в свойстве SelectedValue элемента управления ListBox. Оно отличается от значения свойства `SelectedItem`, которое хранит значение поля, указанного в свойстве DisplayMember.

---

[1] **Более** подробно Windows-формы, **включая** элементы **управления,** поддерживающие **привязку** данных, **рассматриваются в** готовящейся **к** выходу в **свет** книге **Криса** Селлза **(Chris** Sells) *Essential Windows* ***Forms.***

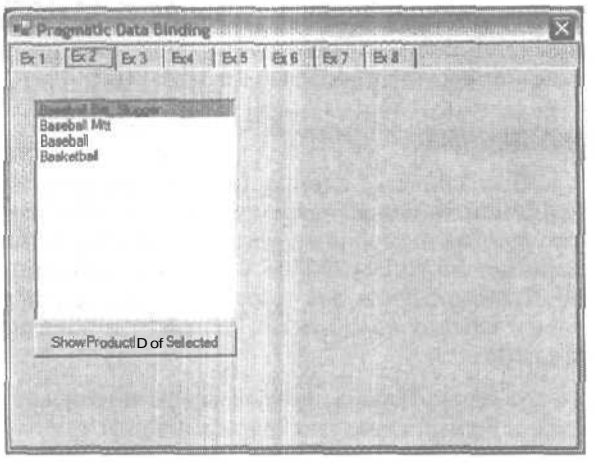


Рис. 10.2. Элемент управления ListBox, содержащий список товаров

Привязка данных к элементу управления DataGrid

Привязка данных к элементу управления DataGrid отличается от привязки данных к остальным **элементам** управления Windows-форм. При простой привязке данных элементу управления ставится в соответствие атомарное значение, при сложной привязке данных — список значений. Особенность привязки данных к элементу управления DataGrid заключается в том, что **мы имеем** дело с более обширным множеством данных. Элемент управления DataGrid позволяет устанавливать значение свойства DataSource равным фактическому источнику данных (такому, как объект **DataTable, DataSet,** DataView и т.д.). Если изменить значение свойства Datasource на этапе выполнения, оно вступит в силу только после перезагрузки элемента управления DataGrid, осуществляемой с помощью метода `DataGrid.SetDataBinding()`. В качестве параметров метод SetDataBinding () принимает новые значения свойств DataMember и DataSource. Значение свойства `DataMember` — это именованная сущность, с которой будет связан элемент управления DataGrid. Если объект **DataSet** содержит несколько таблиц, можно произвести привязку непосредственно к **нему. В** случае привязки к определенной таблице **DataTable** указывать ее имя нет необходимости. Осуществить привязку к объекту **DataTable** можно и путем привязки к объекту **DataSet** и установки значения свойства DataMember равным имени объекта **DataTable,** как показано в листинге 10.4.

Листинг 10.4. Привязка данных к элементу управления *DataGrid* Windows-форм

```
// Привязка к таблице ProductTable.
theGrid.SetDataBinding(productTable, "");

// Привязка к таблице ProductTable.
theGrid.SetDataBinding(dataSet, "Products");

// Привязка к объекту DataSet.
theGrid.SetDataBinding(dataSet, "");
```

Результат привязки к таблице показан на рис. 10.3.

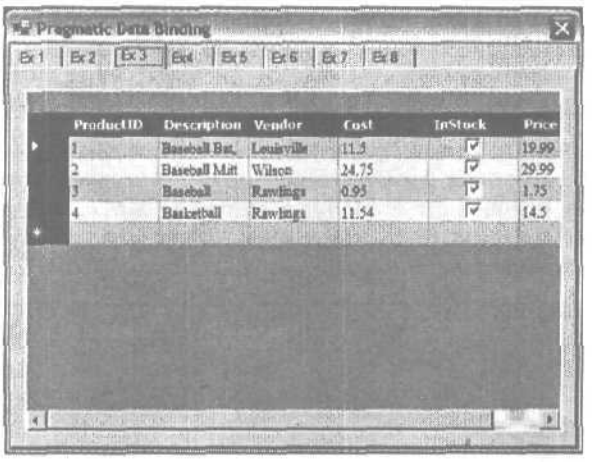


Рис. 10.3. Привязка элемента управления DataGrid к таблице

Элемент управления DataGrid **можно** связать непосредственно с объектом **DataView** (листинг 10.5).

Листинг 10.5. Связывание элемента управления *DataGrid* Windows-форм с объектом *DataView*

```
//Создание представления данных DataView,
// отсортированного по столбцу Price.
DataView sortedView = new DataView(productTable,
                                   И",
                                   "Price",
                                   DataViewRowState.CurrentRows);

// Определение источника данных, отображаемых
// в элементе управления DataGrid.
theDataViewGrid.SetDataBinding(sortedView, "");
```

При необходимости отсортировать или отфильтровать нижележащие **данные** создается **объект DataView, к которому позже будет** осуществлена **привязка. Элемент управления DataGrid, отображающий отсортированные данные, показан на рис. 10.4.**

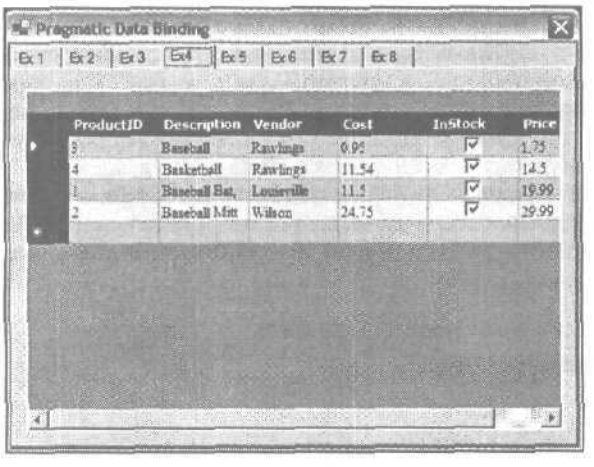


Рис. 10.4. Сортировка данных

Элемент управления DataGrid поддерживает реляционную природу объекта DataSet. Так, если осуществить привязку к таблице клиентов, можно перемещаться по ее отношениям:

```
theGrid.SetDataBinding(customerTable, "");
```

Обратите внимание, что сама привязка производится аналогично привязке к таблице productTable. Разница заключается в том, что элемент управления DataGrid распознает наличие отношений в таблице customerTable (таблица productTable отношений не имеет). После привязки данных элемент управления DataGrid будет выглядеть так, как показано на рис. 10.5.

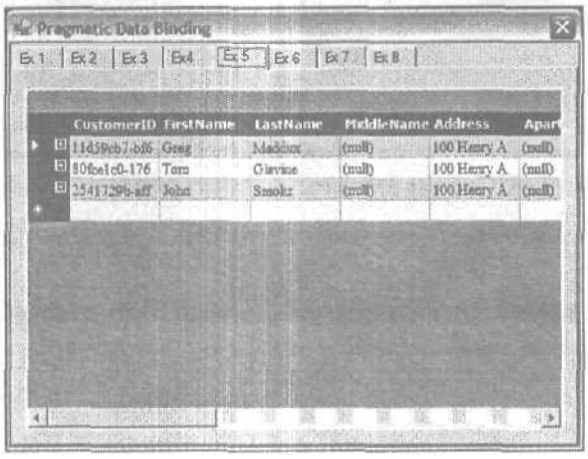


Рис. 10.5. Привязка элемента управления DataGrid к объекту DataTable, содержащему отношения

Полученная таблица позволяет перемешаться по отношениям для того, чтобы просмотреть счета, принадлежащие заданному клиенту, и товары, которые входят в эти счета. Несмотря на то что это очень удобно, привязка типа "родитель-потомок" обладает еще большими преимуществами.

Привязка типа "родитель-потомок"

Привязка типа "родитель-потомок" предполагает привязку одного элемента управления к родительской таблице, а другого — к дочерней. Как показано на рис. 10.6, верхняя сетка данных привязана к родительской таблице, а нижняя — к дочерней.

Для того чтобы это сделать, свяжите родительскую таблицу с родительской сеткой данных, указав соответствующий объект DataTable в качестве значения свойства DataSource, но не указывая значение свойства DataMember. Затем свяжите дочернюю таблицу с дочерней сеткой данных, указав родительский объект DataTable в качестве значения свойства DataSource, а имя отношения — в качестве значения свойства DataMember (листинг 10.6).

Листинг 10.6. Привязка типа "родитель-потомок"

```
// Привязка родительской таблицы.
masterGrid.SetDataBinding(customerTable, "");

// Привязка дочерней таблицы.
detailGrid.SetDataBinding(customerTable,
                          "Customers_Invoices");
```

Теперь выбор элемента в родительской сетке приведет к отображению в дочерней сетке только тех элементов, которые состоят в отношении с выделенным родительским элементом. Механизм привязки данных поддерживает многоуровневые отношения "родитель-потомок". Если связать верхнюю сетку с таблицей customerTable, а нижнюю — с таблицей `invoiceTable` (как это сделано в листинге 10.6), можно создать еще одну сетку данных и связать ее с таблицей `invoiceItemTable` (листинг 10.7).

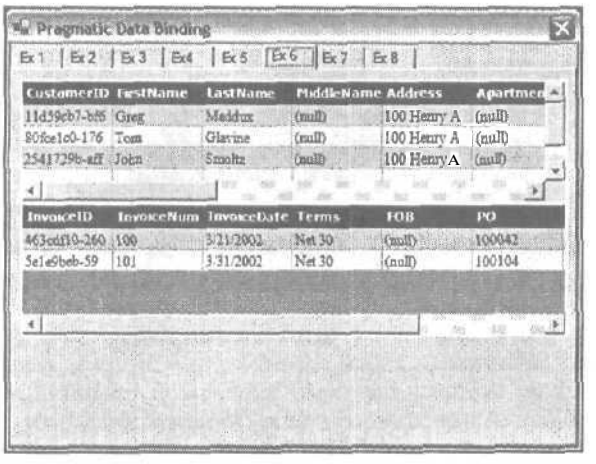


Рис. 10.6. ***Привязка типа*** *"родитель-потомок"*

Листинг 10.7. Многоуровневые отношения "родитель-потомок"

```
// Привязка родительской таблицы.
masterGrid.SetDataBinding(customerTable, "");

// Привязка дочерней таблицы.
detailGrid.SetDataBinding(customerTable,
                          "Customers_Invoices");

// Привязка дочерней таблицы.
detail2Grid.SetDataBinding(customerTable,
                "Customers_Invoices.Invoices_InvoiceItems");
```

Все сетки привязываются к таблице customerTable, однако свойство `DataMember` дочерних сеток указывает на то, что они используются для отражения отношения между таблицами. В этом случае такими отношениями являются отношение `Customer_Invoices` таблицы Customer и отношение `Invoices_InvoiceItems` таблицы invoice. Реляционная природа синхронизации элементов управления основана на том факте, что все сетки используют в качестве источника данных таблицу Customer. Результирующее диалоговое окно будет выглядеть так, как показано на рис. 10.7.

Использование класса CurrencyManager

Позвольте представить вам класс .NET Framework CurrencyManager, названный не вполне естественным образом. Этот класс используется для определения курсора источника данных и управления им в рамках сложной привязки данных. В предыдущих примерах курсор источника данных изменял свое положение каждый раз после щелчка на другом элементе родительской сетки. С помощью класса CurrencyManager вы можете изменить это поведение.

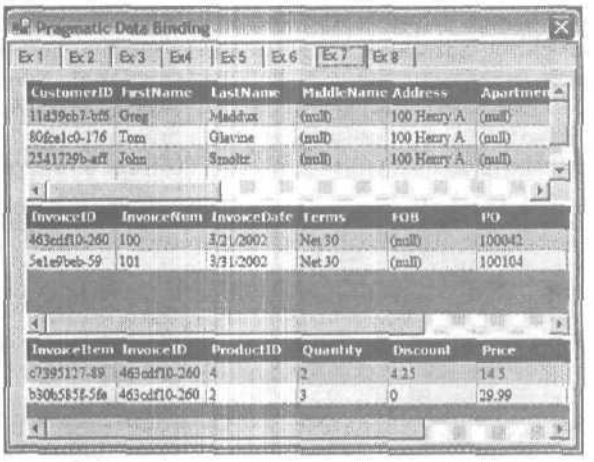


Рис. 10.7. Двухуровневые ***отношения*** *"родитель-потомок"*

Предположим, у нас есть форма с несколькими элементами управления, связанными со столбцами таблицы Customer (имя, адрес и т.д.), элементом управления ListBox, связанным со счетами текущего клиента, и сеткой, предназначенной для отображения подробностей выбранного счета. Обратите внимание, что мы не используем сетку со списком всех клиентов. Наша задача заключается в реализации навигационной панели, имеющей четыре кнопки: "вперед", "назад", "первый", "последний" (как в элементе управления Visual Basic DataBinding). Для начала мы связываем элементы управления с источником данных (во всех случаях используется таблица customerTable, как показано в листинге 10.8).

Листинг 10.8. Использование класса ***CurrencyManager***

```
// Привязка родительской таблицы.
txtCustName.DataBindings.Add( "Text",
                              customerTable,
                              "FullName");
txtAddress.DataBindings.Add(  "Text",
                              customerTable,
                              "Address");
txtCity.DataBindings.Add(     "Text",
                              customerTable,
                              "City");
txtState.DataBindings.Add(    "Text",
                              customerTable,
                              "State");
txtZip.DataBindings.Add(      "Text",
                              customerTable,
                              "Zip");
txtHomePhone.DataBindings.Add("Text",
                              customerTable,
                              "HomePhone");
txtWorkPhone.DataBindings.Add("Text",
                              customerTable,
                              "BusinessPhone");

// Привязка дочерней таблицы.
listInvoices.DataSource = customerTable;
```

```
listInvoices.DisplayMember =
                "Customers_Invoices.Description";

// Привязка дочерней таблицы.
Detail6.SetDataBinding(customerTable,
                "Customers_Invoices.Invoices_InvoiceItems");
```

После этого необходимо определить перемещение курсора по мере щелчков на навигационных кнопках. Мы уже рассказывали, как объект **CurrencyManager** был создан неявно при определении привязки данных в листинге 10.8. Обратиться к объекту **CurrencyManager** можно с помощью свойства формы BindingContext, которое имеет индексатор, принимающий в качестве параметра значение свойства DataSource элемента управления и возвращающий соответствующий ему объект **CurrencyManager**. Для того чтобы изменить положение курсора, нужно изменить значение свойства Position объекта **CurrencyManager,** как показано в листинге 10.9.

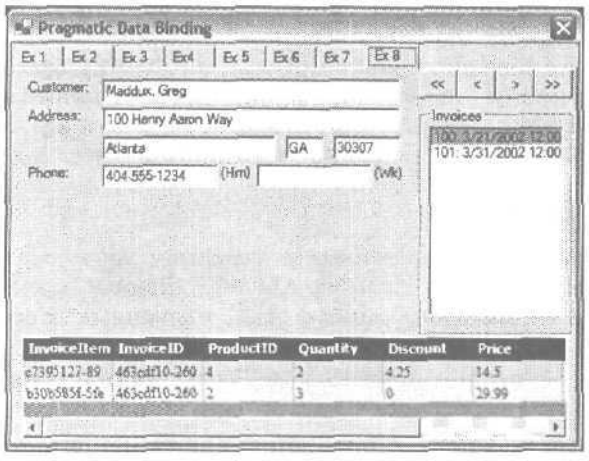


*Рис. **10.8. Использование** объекта **CurrencyManager***

Листинг 10.9. Перемещение курсора с помощью объекта *CurrencyManager*

```
private void btnFirst_Click(object sender,
                            System.EventArgs e)
{
  BindingContext[customerTable].Position = 0;
}

private void btnPrev_Click(object sender,
                           System.EventArgs e)
{
  if (BindingContext[customerTable].Position != 0)
    BindingContext[customerTable].Position -= 1;
}

private void btnNext_Click(object sender,
                           System.EventArgs e)
{
  if (BindingContext[customerTable].Position <
              BindingContext[customerTable].Count - 1 )
    BindingContext[customerTable].Position += 1;
}
```

```
private void btnLast_Click(object sender,
                           System.EventArgs e)
{
  BindingContext[customerTable].Position =
                    BindingContext[customerTable].Count - 1;
}
```

Если объединить вместе код предыдущих примеров, диалоговое окно примет вид, показанный на рис. 10.8,

Привязка данных в ASP.NET

При разработке Web- и ASP.NET-приложений большое значение имеет отсоединенная природа протокола HTTP[2]. Привязка данных к элементам управления Web-форм является отсоединенной и осуществляется в режиме только для чтения, что подчеркивает ее отсоединенную природу. Это не похоже на привязку данных к элементам управления Windows-форм, характеризующуюся непосредственной связью с источником данных. Другими словами, привязка данных в ASP.NET более удобна для помещения данных в элементы управления Web, чем для связывания элементов управления с данными. Это несколько упрощает привязку данных в ASP.NET по сравнению с привязкой данных к элементам управления Windows-форм.

Простая привязка данных

В ASP.NET простая привязка данных к элементам управления осуществляется с помощью синтаксиса <%# %> на странице ASP.NET. Данные, с которыми связывается элемент управления, — это любые данные, "присоединенные" к странице, например:

```
<asp:TextBox id="theTextBox"
             text='<%# productTable.Rows[0]["Price"] %>'
             runat="server" />
```

ASP.NET-страница обратится к таблице productTable, извлечет из нее значение столбца Price первой строки и поместит его в текст элемента управления TextBox. Для того чтобы осуществить привязку данных, необходимо вызвать метод DataBind() элемента управления TextBox. В этом-то и заключается принципиальная разница между привязкой данных к элементам управления Windows-форм и привязкой данных к элементам управления Web-форм. Элементы управления Windows-форм связываются с реальными данными (изменения в элементе управления влекут за собой изменения в данных), в то время как элементы управления Web-форм — с их копией. В листинге 10.10 приведен пример привязки данных к элементу управления Web-форм.

Листинг 10.10. Вызов метода *DataBindO* элемента управления Web-форм

```
// Вызов метода DataBindO элемента управления
// Web-форм, размещенного на ASP.NET-странице,
// приведет к осуществлению фактической привязки
// данных к этому элементу управления. Для того
// чтобы связать с данными все элементы управления
// ASP.NET-страницы, следует вызвать метод DataBindO
// для всей страницы.
theTextBox.DataBind();
```

[2] Hypertext **Transport** Protocol; см. также www.w3.org/protocols.

Следует отметить, что на самом деле привязка данных осуществляется ядром ASP.NET путем создания кода привязки и его размещения в фоновом коде страницы. В результате привязка данных к текстовому полю ввода будет выглядеть так:

```
target.Text =
            Convert.ToString(productTable.Rows[0]["Price"]);
```

Для того чтобы привязка данных прошла успешно, необходимо сделать таблицу productTable членом фонового кода страницы. Поскольку мы не связываем с элементами управления список значений, все работает хорошо. Тем не менее в большинстве случаев к элементам управления Web-форм привязываются большие объемы данных.

Сложная привязка данных

Как и в Windows-формах, целью сложной привязки данных является связывание коллекции или списка элементов с элементом пользовательского интерфейса — как правило, с элементом управления ASP.NET ListBox или DropDownList. Эти элементы управления (и другие, подобные им) поддерживают следующие свойства.

- DataSource — объект данных, с которым будет связан элемент управления. Обычно это объект DataTable или DataView.
- DataTextField — поле объекта, указанного в свойстве DataSource, которое будет использовано для отображения в элементе управления.
- DataTextFormatString — необязательная строка форматирования, которая позволяет элементу управления определять формат столбца, указанного в свойстве DataTextField. Например, установка значения свойства DataTextFormatstring в {0:d} позволяет вывести дату в короткой форме.
- DataValueField — **поле объекта, указанного** в свойстве DataSource, **в котором хранится значение элемента управления.**

Пример сложной привязки **данных показан в листинге** 10.11.

Листинг 10.11. Сложная привязка данных к элементам управления Web-форм

```
// Привязка данных к элементу управления ListBox.
theListBox.DataSource - Session["customerTable"];
theListBox.DataTextField = "FullName";
theListBox.DataValueField = "CustomerID";
theListBox.DataTextFormatString = "Name: {0}";

// Привязка данных к элементу управления DropDownList.
theDropDown.DataSource = Session["productTable"];
theDropDown.DataTextField = "Description";
theDropDown.DataValueField = "ProductID";

// Привязка данных ко всем элементам управления.
DataBind();
```

Привязка данных к элементам управления, предназначенным для работы с данными

В ASP.NET существует несколько элементов управления, которые позволяют отображать более одного столбца таблицы, — DataGrid, DataList и Repeater. Привязка данных к этим элементам управления осуществляется точно так же, как и к элементу управления DataGrid Windows-форм — достаточно установить значение свойства DataSource и необязательного свойства DataMember (листинг 10.12).

Листинг 10.12. Привязка данных к элементам управления Web-форм, предназначенных для работы с данными

```
// Привязка таблицы productTable к элементу
// управления Web-форм DataGrid.
theGrid.DataSource = productTable;

// Альтернативный вариант.
// theGrid.DataSource = dataSet;
// theGrid.DataMember = "Products";

// Фактическая привязка данных.
DataBind();
```

Как только привязка осуществлена, можно определить специфические элементы управления для привязки к ним определенных столбцов или позволить им автоматически генерировать столбцы или строки.

Привязка данных с помощью объекта DataReader

Так как привязка данных в ASP.NET осуществляется в режиме только для чтения, элементы управления Web-форм можно связать с объектом **DataReader** несмотря на то, что он предоставляет всего лишь однонаправленный курсор. В этом случае ASP.NET просто проходит по объекту DataReader и заполняет соответствующий элемент управления каждой строкой объекта **DataReader,** как показано в листинге 10.13.

Листинг 10.13. Привязка данных с помощью объекта **DataReader**

```
// Создание соединения.
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "Server=localhost;" +
                        "Database=ADONET;" +
                        "User ID=ADOGUY;" +
                        "Password=ADOGUY";
conn.Open();

// Создание объекта команды.
SqlCommand cmd = conn.CreateCommand();
cmd.CommandText =
               "SELECT Description, ProductID FROM PRODUCT";

// Создание объекта DataReader для привязки данных
// к элементу управления ListBox.
SqlDataReader rdr = cmd.ExecuteReader();

// Привязка данных к элементу управления ListBox.
theListBox.DataSource = rdr;
theListBox.DataTextField = "Description";
theListBox.DataValueField = "ProductID";
theListBox.DataBind();

// Закрытие соединения.
conn.Close();
```

Привязка данных с помощью объекта **DataReader** оправдана при работе с часто изменяющимися данными (т.е. когда кэширование является неэффективным).

Некоторые соображения, касающиеся производительности

Возможность привязки данных в ASP.NET имеет большое значение, однако способ хранения данных за пределами Web-страницы играет очень важную роль, фактически, определяя скорость ее работы. При этом обычно рассматривается вопрос о необходимости кэширования и, если решение принимается в пользу кэширования — о месте его проведения. Более подробно быстродействие и возможность масштабирования решений на базе ADO.NET рассматривается в главе 11, "Масштабируемость и производительность приложений ADO.NET".

Объекты DataReader (данные не кэшируются)

Большинство ASP-узлов даже не пытаются кэшировать данные, предполагая, что база данных имеет достаточный запас производительности для мгновенного предоставления информации для каждой страницы. На удивление, это оказывается довольно эффективным решением. Если разрабатывается приложение для интрасети или небольшой Web-узел, не рассчитанный на большую нагрузку, то использование объектов DataReader без кэширования является приемлемым решением.

Кэширование на стороне сервера

Если ввиду размера или масштаба Web-узла кэширование становится необходимостью, решением может стать кэширование на стороне сервера. Для поддержки некоторой информации в памяти с целью обеспечения ее доступности для ASP-страниц были созданы целые системы. ASP.NET предоставляет еще больше вариантов выбора.

В отличие от ASP-страниц, в ASP.NET для хранения данных, относящихся к определенному пользователю, применяется система управления состоянием сеансов (Session State Management System). ASP.NET поддерживает состояние сеанса локального уровня, уровня Web-кластера и уровня SQL Server, так что механизм поддержки состояния сеанса найдется для каждой ситуации.

Для кэширования данных, не относящихся к конкретному пользователю (например, списка товаров), можно применить статический класс (класс, все члены которого статические) или сохранить информацию на уровне приложения. Кэширование на уровне приложения больше имеет смысл применять в случае доступности для всех пользователей одной и той же информации. Недостаток такого подхода заключается в том, что при изменении нижележащих данных они меняются для всех сеансов одновременно.

Кэширование на стороне клиента

Ранее при необходимости сохранить состояние сеанса на стороне клиента нужно было создать скрытые поля, предназначенные для хранения информации. Когда пользователь отправлял заполненную форму, данные из скрытых полей передавались на сервер. В ASP.NET существует встроенная поддержка кэширования данных Web-форм, которая называется *представлением сеанса (view state)*. Представление сеанса практически аналогично старому методу со скрытыми полями, но более тесно интегрировано в Web-формы. Представление сеанса страницы или элемента управления хранится в одном скрытом поле, содержащем информацию о состоянии страницы или элемента управления. К сожалению, использование состояния сеанса приводит к резкому возрастанию размеров Web-страниц. В то время как простая форма с одним элементом управления без поддержки представления сеанса может занимать 1384 байта, эта же форма с поддержкой представления сеанса увеличивается в размере до 3800 байт. Этот пример не описывает все 100% случаев, но достаточно точно отражает суть проблемы. В зависимости от того, кто является конечным пользователем, данный недостаток может оказаться несущественным, а может и серьезно ухудшить ситуацию. Действительно,

если пользователь будет подключен к Internet через модемное соединение со скоростью 28,8 Кбит/с, то он заметит "лишние" 2416 байт. Ну а если приложение разрабатывается для внутренней сети, то эта разница будет ничтожно мала.

Резюме

При разработке Windows- и Web-форм ASP.NET привязка данных к элементам управления является довольно мощным инструментом в арсенале программиста. Для оптимального использования этого инструмента важно уметь разбираться в различных типах привязки данных. Знание разницы между простой и сложной привязкой данных, а также понимание того, где уместна привязка типа "родитель-потомок", позволит быстрее разрабатывать более дружественные пользовательские интерфейсы.

Глава 11

Масштабируемость и производительность приложений ADO.NET

В этой главе...

А стоит ли волноваться?
Что предшествовало ADO.NET
Как может помочь ADO.NET
Можно ли масштабировать объекты DataReader?
Производительность ADO.NET
Несколько полезных советов

В этой главе рассматриваются вопросы, находящиеся на стыке теории и практики. ADO.NET предоставляет инструменты, помогающие создавать масштабируемые высокопроизводительные приложения, но с помощью этих инструментов можно "умудриться" создать программное обеспечение, которое не будет обладать способностью к масштабированию или будет работать слишком медленно. Важно понимать, в чем заключается разница между "плохим" и "хорошим" приложением. Моя задача как автора книги заключается в том, чтобы научить читателя понимать эту разницу и дать ему достаточно знаний для эффективного поиска решений возникающих проблем.

А стоит ли волноваться?

Когда разработчики проектируют системы, большая часть их времени тратится на обеспечение работоспособности различных функциональных элементов. Однако же если проектирование сводится только к обеспечению нужной функциональности, проект обречен на неудачу. Я хотел бы рассказать вам историю, которая произошла со мной в прошлом, когда я больше заботился о функциональности, а не о производительности проектов.

Несколько лет назад я занимался созданием приложения, позволяющего врачам выбирать медикаменты для рецепта с помощью планшетного компьютера. Поскольку врач должен иметь возможность выбрать любое лекарство, был создан список и перечислены всевозможные наименования препаратов, Я разработал необходимые запросы к базе данных, экраны пользовательского интерфейса и отчеты, которые будут выводиться на принтер (рецепты). На моем компьютере все работало просто отлично.

Передав программу на тестирование, я собрался провести весь день в лаборатории проверки качества, наблюдая за реакцией пользователей. Ожидалось, что мне придется ответить на несколько простых вопросов, касающихся работы приложения, но как только пользователи попадали на экран выбора лекарства, он зависал. Посреди экрана переворачивались песочные часы, но больше ничего не происходило. После нескольких минут ожидания я сдался.

Вечером того же дня персонал лаборатории забыл перезагрузить одну из систем, а поэтому она продолжала работать всю ночь. К нашему удивлению, на следующее утро оказалось, что экран планшетного компьютера работает. Я попытался выполнить некоторые действия с приложением и обратил внимание на то, что названия всех лекарств начинаются на букву "А". Как только я попытался переместить полосу прокрутки, экран опять завис. Я спросил персонал тестовой лаборатории, сколько наименований лекарств хранится в базе данных. Они ответили: "Около двенадцати тысяч, а что?". Теперь все стало на свои места. Ведь дома при тестировании приложения я пользовался списком из сорока медикаментов.

В конце концов я заменил список таблицей, которая поддерживает загрузку только тех наименований лекарств, которые могут уместиться на экране. Чему я тогда научился? Узнал, что наличие информации о масштабах системы имеет огромное значение в аспекте нахождения правильного решения проблемы. Так стоит ли волноваться? Конечно же, стоит.

Проектирование масштабируемых систем

Все решается на этапе проектирования. При проектировании компонента, приложения или системы нужно постараться уравновесить требования к проекту. Необходимо понять, насколько производительность, масштабируемость, жизнеспособность, время выхода на рынок, переносимость и безопасность проекта являются критическими для приложения. Взвесив все перечисленные характеристики, вы быстро оцените реальные требования к масштабируемости и производительности. В результате вам не удастся спроектировать программное обеспечение, которое будет обладать высокой степенью масштабируемости и производительности во всех возможных ситуациях.

Что же подразумевается под способностью к масштабированию и высокой производительностью? Это субъективные термины, которые имеют различное значение в различных ситуациях. Когда идет речь о масштабируемости, имеется ввиду степень изменения производительности системы при увеличении размера проблемной области. Другими словами, при росте требований к системе (например, при увеличении числа одновременно работающих пользователей или объемов хранимой информации) она должна иметь возможность адаптироваться к ним, предоставлять уровень обслуживания, аналогичный уровню обслуживания до увеличения требований. Для большинства современных систем существуют прозрачные метрики, характеризующие степень масштабирования системы. Как любит повторять Тим Эвальд (Tim Ewald), "приложение или масштабируется, или нет"[1]. Действительно, ведь в конечном итоге нам необходимо получить систему, полностью соответствующую предъявляемым к ней требованиям. И если требования заключаются в необходимости поддержки тысячи одновременно работающих пользователей, то проектируемая система должна уметь справляться с сотней тысяч пользователей. С другой стороны, производительность системы характеризует скорость ее реагирования на каждый запрос. Масштабируемость и производительность идут рука об руку, потому что система должна работать быстро вне зависимости от того, сколько пользователей работают в ней в данный момент. Метрики производительности должны отражать тот факт, что время реакции системы не изменяется при увеличении нагрузки.

Вне всяких сомнений, масштабируемость и производительность должны рассматриваться совместно. Проектирование с учетом масштабируемости и производительности предполагает создание системы, допускающей диверсификацию приложения по мере увеличения размера проблемной области. К примеру, при проектировании Web-узла с требованием среднего времени реакции, равного пяти секундам, отсутствие

[1] *Transactional COM+: Building Scalable Applications*, Addison-Wesley Publishers, ISBN: 0201615940

предполагаемого числа одновременно работающих пользователей делает значение "пять секунд" совершенно бесполезным. Легко создать Web-узел, предназначенный для обслуживания единственного пользователя, и намного сложнее — Web-узел, который будет быстро реагировать на запросы любого количества пользователей.

Связность компонентов системы

Дни проектирования монолитных систем закончились, не правда ли? Это не совсем так. Только из-за того, что при проектировании системы используются компоненты и объектно-ориентированное программирование, система не перестанет быть монолитной или начнет легко масштабироваться. Так что же подразумевается под монолитностью? Под монолитностью подразумевается связность. При проектировании системы важно не ориентироваться на то, где будет выполняться код или где будут храниться данные. Если создать тесную связь между компонентами или другими фрагментами кода и данных, то система сможет масштабироваться только "вверх" (увеличение размеров дисков, памяти и количества процессоров сервера). Если система проектируется для масштабирования "вширь" (т.е. планируется распределение фрагментов системы по отдельным компьютерам), то решение о ее разделении можно принимать на основании текущей нагрузки. С ростом нагрузки будет расти и система.

С моей точки зрения, масштабирование "вверх" — это путь в никуда. В то же время масштабирование "вширь" — это чистая магия. Другими словами, если система представляет собой один гигантский фрагмент кода, существующий на одном сервере, она не сможет масштабироваться. Чем больше компонентов, которые (при необходимости) могут быть перемещены в распределенное окружение, будет включать в себя система, тем она будет легче масштабироваться. Я считаю, что серверы баз данных монолитны (во всяком случае, так было до последнего времени, пока не появились связанные серверы и представления, распределенные по нескольким машинам), а Web-узлы более склонны к разделению на части, поскольку вы можете разместить в пределах одного кластера несколько машин, предоставляющих идентичную информацию, и переместить бизнес-уровень на еще один набор компьютеров. Отдельные слабо связанные фрагменты системы легче поддаются масштабированию.

Что предшествовало ADO.NET

Вопрос масштабируемости рассматривается не первый год. На текущем этапе уже известен способ масштабирования приложений, ориентированных на взаимодействие с базой данных. К сожалению, до появления ADO.NET осуществлять масштабирование было очень сложно, и для этого требовалось большое количество времени. Как правило, в начале работы над проектом все полны намерениями создать масштабируемую систему, но под давлением жесткого расписания и технических проблем о масштабируемости забывают. Двумя проблемами, которые занимали умы программистов в последнее время, были масштабирование ориентированных на данные Web-приложений и масштабирование самих баз данных.

Проблемы, связанные с подсоединенными данными

Когда Microsoft создала активные серверные страницы (Active Server Pages — ASP), она хотела облегчить для разработчиков процесс создания Web-узлов. По большому счету, Microsoft преуспела. В то время использование интерфейса ADO было единственным способом получения информации из базы данных. Модель ADO упростила создание долгоживущих соединений с базой данных. Несмотря на то что эта модель прекрасно работала с приложениями, предназначенными для настольных компьютеров

и даже для небольших клиент-серверных систем, разрабатывать Web-приложения с ее помощью оказалось чрезвычайно трудно.

Много компаний создали свои первые Web-узлы на основе ADO и ASP, К сожалению, слабым местом таких узлов оказались серверы баз данных, которые превышали лимит открытых соединений. Все работало хорошо, когда нагрузка не превышала сотни посещений в сутки, но как только она увеличивалась, система становилась неработоспособной. Необходимо было срочно что-то делать. Некоторые программисты пытались решить проблему, сократив фактическое время соединения с базой данных, Компании настаивали на том, чтобы разработчики использовали соединения только в пределах конкретной страницы. Это помогало только до тех пор, пока не возникала необходимость в кэшировании данных. Разработчики создали множество различных решений для реализации механизма кэширования данных, но большинство из них основывались на копировании информации из базы данных в собственные структуры и их сохранении в кэше, содержащемся в оперативной памяти. Опять же, все работало прекрасно, пока не возникала необходимость в масштабировании кэша за пределы одного компьютера. Для того чтобы поддерживать синхронизацию распределенного кэша, каждой из его частей передавался сигнал об изменении в базе данных, что, в свою очередь, приводило к обращению всех частей кэша к базе данных для обновления своего содержимого. С этим можно было смириться вплоть до момента возрастания нагрузки на базу данных, когда ее изменение приводило к целой лавине запросов, обусловленной необходимостью одновременного обновления содержимого распределенного кэша. Существовали также иные решения данной проблемы: одни из них были довольно удачными, а другие — нет.

Фактически для обеспечения масштабируемости ориентированных на данные Web-узлов разработчики были вынуждены создавать целые системы только лишь для того, чтобы уменьшить зависимость от соединений с базами данных. Эти уровни кэширования должны были обеспечивать следующую функциональность:

- манипулирование информацией, хранящейся в базе данных;
- предоставление структур данных, в которых будет храниться информация из базы данных за пределами ADO;
- предоставление кэша в памяти, предназначенного для хранения этих структур данных.

Microsoft поняла, что такая тенденция принимает угрожающий характер, и попыталась добавить поддержку отсоединенного объекта ADO Recordset. Это немного улучшило ситуацию, Разработчики знали о такой возможности, и предполагалось, что они умеют работать с данными в отсоединенном режиме. Поскольку на создание проекта отводилось ограниченное время, разработчики использовали подсоединенный доступ к данным с твердым намерением перейти в ближайшем будущем на отсоединенный доступ. Обычно этого так никогда и не происходило.

Проблемы масштабирования сервера баз данных

В конце концов тактика сокращения количества соединений с базой данных исчерпывает себя, и уже не удается добиться увеличения быстродействия системы. Зачастую следующим шагом является попытка улучшить масштабируемость сервера баз данных. Как всегда, существует два способа масштабирования — "вверх" и "вширь". Масштабирование "вверх" обычно предполагает модернизацию аппаратного обеспечения, добавление процессоров, увеличение объема оперативной памяти или добавление более быстрых жестких дисков. Эти меры эффективны лишь до определенного момента.

При масштабировании сервера баз данных "вширь" обычно осуществляется сегментация (размещение баз данных на нескольких компьютерах), дублирование (создание копий данных на различных компьютерах) или организация кластеров (соединение нескольких компьютеров в один логический блок с целью разделения нагрузки).

Для сегментации требуется поддержка информации о фактическом месте хранения данных и реализация механизма получения нужных данных из определенного места. Это может быть сделано с помощью переключения серверов баз данных, однако по-прежнему не исключается необходимость наличия кода, который будет знать о сегментации. Кроме того, когда первичная сегментация исчерпает себя, потребуется проведение второго цикла сегментации. (Примером может служить размещение данных о поставщиках и клиентах на одном компьютере, а данных о счетах и товарах — на другом. После того как нагрузка на серверы заметно возрастет, придется размешать отдельные таблицы базы данных на отдельных серверах.) Большая часть кода при этом потребует кардинального пересмотра, необходимо будет также выполнить определенную работу по фактическому перемещению данных.

Дублирование применяется при работе с данными, предназначенными только для чтения, так как несколько копий данных можно предоставлять без возникновения конфликтов. Тем не менее при добавлении новых компьютеров необходимо сообщить программному обеспечению, что существует новая машина, к которой можно обращаться за данными.

Создание кластеров компьютеров для размещения сервера баз данных является еще одним популярным вариантом решения проблемы. На практике, однако же, в кластеры объединяются лишь два или четыре компьютера, что значительно ограничивает проведение масштабирования "вширь".

Ни одно из этих решений не является оптимальным, а проведение масштабирования "вширь" после первой итерации занимает много времени и затрагивает множество систем.

Как может помочь ADO.NET

ADO.NET действительно может помочь, но и это не является панацеей при плохой практике разработки программного обеспечения. Вам все еше необходимо самостоятельно создавать код доступа к базе данных, однако с помощью ADO.NET этот процесс намного ускоряется и уменьшается необходимость в повторении фрагментов кода. ADO.NET позволяет использовать базу данных не только в подсоединенном режиме. Подход ADO, заключавшийся в непосредственном редактировании записей на сервере баз данных с их блокировкой, в ADO.NET попросту не реализован. В этом аспекте ADO.NET помогает избавиться от кода, который приводил к возникновению множества проблем при создании ASP-страниц.

Тогда как необходимость работы в подсоединенном режиме в ADO является основной причиной проблем масштабирования, отсоединенная архитектура ADO.NET позволяет обойти множество подводных камней. Но как именно следует использовать ADO.NET? Это зависит от природы приложения. Для обеспечения масштабируемости можно воспользоваться несколькими стратегиями.

Кэширование данных на Web-сервере

Большинство Web-серверов современных публичных Web-узлов масштабируется с помощью объединения одинаковых компьютеров в кластер для обслуживания входящих запросов. Масштабирование при этом достигается за счет сокращения зависимости от базы данных вследствие кэширования данных на Web-сервере, Информация считывается в объекты DataSet (или типизированные объекты DataSet) и сохраняется

на Web-сервере. Уровень кэширования зависит от выбора приложения. В табл. 11.1 перечислены наиболее распространенные сценарии кэширования.

Таблица 11.1. Распространенные сценарии кэширования данных

Сценарий	*Описание*	*Недостатки*
Кэширование информации на уровне пользователя	Кэшированные объекты **DataSet** хранятся в состоянии сеанса, а поэтому в рамках каждого сеанса осуществляется только одно обращение к базе данных	Если пользователи находятся на Web-узле недолго, то преимущества кэширования минимальны
Кэширование информации на уровне приложения	Кэшированные объекты **DataSet** хранятся либо в состоянии приложения, либо по одному объекту на процесс, а поэтому для приложения или процесса существует только одна копия данных	Если данные быстро меняются, то такой метод кэширования связан с проблемами, так как требует частого обновления кэша. Кроме этого, должен существовать автоматизированный процесс, копирующий изменения информации в объектах DataSet обратно в базу данных
Кэширование информации на уровне сервера	Создается единственный объект **DataSet** для каждого компьютера, доступный всем приложениям	Те же недостатки, что и у предыдущего варианта

Кэширование данных на Web-сервере напоминает ситуацию в ASP-мире, не считая упрощения компонентного проектирования за счет использования объектов **DataSet** и типизированных объектов **DataSet.** Кроме того, при использовании состояния сеанса ASP.NET становится доступен жизнеспособный и масштабируемый механизм кэширования, основанный на сеансах. Времена, когда каждая компания была вынуждена создавать собственную систему кэширования сеансов, уже прошли.

Масштабирование информации, хранящейся в базе данных

Еще одним способом обеспечения масштабируемости системы является полный отказ от непосредственного взаимодействия Web-сервера с базой данных. Как говорилось ранее, масштабирование сервера баз данных вызывает ряд проблем. К счастью, ADO.NET предоставляет инструментарий для масштабирования базы данных.

Использование объектов DataSet вместо непосредственного доступа к базе данных

Объект **DataSet** очень напоминает базу данных, хранимую в памяти. При желании можно хранить таблицы и схему данных в объекте **DataSet** и заставить приложения работать только с **этим** объектом. К сожалению, многие разработчики привыкли использовать SQL-операторы для доступа к информации, хранящейся в базе данных, так что отсутствие механизма поддержки SQL становится для них серьезным барьером. Естественно, что от подобных привычек следует избавляться. В системе с хорошо структурированными данными необходимо изолировать Web-разработчика от SQL вне зависимости от того, что для этого будет использовано, — уровень бизнес-объектов или хранимые процедуры. В рамках парадигмы ADO.NET это достигается за счет использования в качестве объектной модели Web-узла объекта **DataSet** или типизированного объекта **DataSet.** Помните, что сегодня информация может храниться в базе данных, а завтра — в формате XML или, что еще лучше, в своеобразном "гибриде" первого и второго. Использование класса `XmlDataDocument` позволяет представить хранимую в памяти базу данных в формате XML и дает разработчикам возможность применения языка XPath для создания запросов.

К счастью, большинству разработчиков не требуется поддержка сложной аналитической обработки данных, как правило, все, что им необходимо, — это извлечь из базы данных определенную сущность. Например, при создании Web-узла электронной коммерции разработчику может потребоваться обратиться к сущности пользователя. В этом случае использование языка запросов XPath для поиска информации в объекте **DataSet** является более чем адекватным решением. Надеюсь, что когда язык запросов XQuery[2] будет стандартизирован, классы DataSet и `XmlDataDocument` также будут его поддерживать.

Масштабирование объектов DataSet

Итак, на текущем этапе у нас есть объект **DataSet,** который представляет собой хранимую в памяти базу данных, и способ осуществления запросов к этому объекту. Но как это может пригодиться при масштабировании базы данных? Дело в том, что мы можем размещать объекты **DataSet** на компьютерах среднего уровня. Вне зависимости от того, как осуществляется доступ к объектам **DataSet** — через удаленные вызовы или через Web-службы, — можно создать кластер компьютеров, который позволит нам масштабировать базу данных. Это лучше, чем традиционные системы кэширования, поскольку компьютеры с объектами **DataSet** могут передавать друг другу DiffGram-документы, позволяющие поддерживать все компьютеры в одинаковом состоянии и избежать обращений к базе данных.

Кроме этого, в .NET можно регулировать нагрузку на серверы среднего уровня с помощью тех же приемов, которые использовались для балансировки нагрузки на Web-серверы. Если рассмотреть удаленные вызовы .NET, то можно заметить, что создание экземпляров классов осуществляется с помощью URL, Как и Web-приложения, нагрузку на Web-службы можно легко дозировать. Если будет принято решение об использовании удаленных вызовов, регулирование нагрузки все еще будет возможно, потому что и удаленные вызовы по протоколу HTTP, и удаленные вызовы по протоколу TCP создают объекты на основании адресов URL. Поскольку это всего лишь адрес URL, нагрузку на него можно регулировать так же, как и нагрузку на Web-приложение или Web-службу.

Какие данные следует хранить в удаленных объектах DataSet

Самым существенным вопросом, с которым я столкнулся при создании удаленных объектов **DataSet,** был выбор способа обработки изменений в нижележащих данных и изменений в различных экземплярах самого класса DataSet. На то время, пока я пытался разобраться в этом вопросе, я выделил часть данных конкретной системы, характеризующуюся достаточной статичностью. Например, в типичном приложении электронной коммерции использование удаленных объектов **DataSet** для хранения данных каталога (который обычно статичен) избавляет от необходимости обеспечивать обработку параллелизма при доступе **к этим** объектам. К сожалению, в системах учета или мониторинговых системах реального времени такой подход попросту бесполезен. Выясните, какая часть данных вашей системы часто изменяется. В целом чем реже изменяются данные, тем меньше работы необходимо выполнять программисту.

Дублирование или сегментация?

После принятия решения об использовании удаленных объектов **DataSet** необходимо определить стратегию хранения информации. Стоит ли создавать дубликат всей базы данных на каждом компьютере среднего уровня или следует разбить данные на сегменты?

---

[2] **Подробную информацию,** касающуюся **языка запросов** Xquery, **можно найти по** адресу: `www.w3.org/XML/Query`, **а информацию о первоначальной реализации** XQuery для .NET **Framework — по адресу:** `xqueryservices.com`.

Оба решения связаны с возникновением определенных проблем. Дублирование предполагает необходимость обеспечения синхронизации копий базы данных. С другой стороны, сегментация требует реализации механизма поиска удаленного объекта DataSet, который содержит искомые данные. К тому же сегментация работает не очень хорошо, если нужно проводить объединение или слияние данных из различных сегментов. Теоретически можно попытаться использовать оба решения: дублировать сегменты базы данных, хранящиеся в объектах DataSet. В любом случае вам придется взвешивать все за и против и на основе требований конкретной предметной области выбрать оптимальное решение.

Синхронизация

Если данные в объектах DataSet могут быть модифицированы или пополнены, вам придется взять на себя заботу о синхронизации различных экземпляров данных. К счастью, в ADO.NET это не очень сложно. Синхронизация данных, хранящихся на нескольких различных компьютерах, осуществляется довольно просто. К примеру, неплохим решением является использование формата DiffGram (см. главу 9) для распространения информации об изменениях между несколькими компьютерами. Немного сложнее выглядит синхронизация компьютеров с базой данных. Для доставки сообщений об изменении базы данных SQL Server всегда можно воспользоваться службой Microsoft Message Queue (MSMQ), однако при работе с другими базами данных аналогичная задача решается с помощью средств, специфичных для конкретной системы. Предпочтительнее вносить изменения в базу данных на среднем уровне и рассылать DiffGram-документы остальным компьютерам сразу же после окончания ее обновления. Тогда фактическое обновление базы данных будет осуществлять только тот компьютер, на котором произошло изменение информации, а остальные серверы будут использовать DiffGram-документ для поддержки целостности данных. По сравнению со старыми системами кэширования этот метод намного эффективнее, так как необходимость обращения к базе данных со стороны остальных серверов попросту отпадает.

В реальности все обстоит гораздо проще. В большинстве случаев данные либо предназначены только для чтения (например, каталог товаров в системе электронной коммерции), либо редактируются одновременно лишь в одном сеансе (посетитель добавляет товар в "корзину" покупателя). При использовании состояния сеанса ASP.NET проблема параллельного доступа теряет свою актуальность.

На практике

На практике можно выбрать решение, основанное на использовании рассмотренных ранее техник. Так, данные о потребителе могут кэшироваться в состоянии сеанса в виде XML-кода, а объекты DataSet могут быть распределенны по компьютерам среднего уровня (образуется еше один уровень масштабирования). Один из способов проектирования системы при разработке приложения электронной коммерции показан на рис. 11.1.

В этом примере получение информации о каталоге и пользователе происходит с помощью удаленных объектов DataSet. Информация о каталоге кэшируется на каждом Web-сервере, в то время как информация о пользователях извлекается непосредственно из удаленных объектов DataSet. Единственной частью системы, непосредственно взаимодействующей с базой данных, является программное обеспечение на компьютерах среднего уровня. В качестве положительного побочного эффекта можно рассматривать возможность избежать непосредственных соединений между Web-сервером и сервером базы данных.

Естественно, это решение не подходит для всех ситуаций. Поскольку оптимальная стратегия кэширования данных определяется для каждой конкретной системы, вам придется оценить все за и против предложенного выше решения.



Рис. 11.1. Структура масштабируемой системы, использующей объекты DataSet

Можно ли масштабировать объекты DataReader?

С помощью ADO.NET можно создать программное обеспечение, которое будет медленно работать и плохо масштабироваться. Отмечу еще раз, что ADO.NET не является панацеей при принятии неверных решений во время проектирования. Вместе с тем ADO.NET способствует тому, чтобы разработчики двигались в правильном направлении. С этой точки зрения объект **DataReader** ничем не отличается от объекта DataSet. Мы имеет дело с двумя совершенно различными сущностями: объект DataSet представляет собой хранилище данных, расположенное в памяти, а объект **DataReader** — средство для непосредственного чтения информации из базы данных. Вполне возможно, что при заполнении информацией объекта **DataSet** "за кулисами" используется объект **DataReader.** Этот единственный факт способен подсказать программисту правильный способ работы с объектом **DataReader.**

Фактически существует два случая, когда использование объекта **DataReader** в Web-приложении является оправданным.

- **Наличие часто изменяющихся данных.** В этом случае объект **DataReader** непосредственно взаимодействует с базой данных и всегда имеет доступ к самой "свежей" информации. Это означает, что, с одной стороны, объекты **DataReader** будут не очень хорошо масштабироваться, а с другой — при работе с часто изменяющимися данными использование объектов **DataSet** существенно затруднено.
- **Необходимость работы со складом данных.** Объект **DataReader** является естественным инструментом для работы с большими складами данных. В такой ситуации, скорее всего, нет смысла кэшировать терабайты информации (даже если для этого есть физическая возможность). Поскольку объект **DataReader** предоставляет доступ к потоку записей из базы данных, у вас никогда не возникнет необходимость одновременно хранить все данные в памяти. Несмотря на то что это решение также не слишком-то хорошо масштабируется, попробуйте найти оптимальный подход к масштабированию терабайта информации!

Конечно же, использование механизма кэширования вывода ADO.NET способствует уменьшению количества проблем, связанных с масштабированием объектов **DataReader. Как** говорилось ранее в этой главе, механизм кэширования вывода

позволяет кэшировать страницы на основе заданного критерия (строки запроса, отправляемых на обработку данных и т.п.). Например, при использовании объекта DataReader для считывания информации о конкретном товаре можно кэшировать каждую версию страницы на основе идентификационного номера товара. Более подробно кэширование вывода рассматривалось в разделе этой главы "Кэширование данных на Web-сервере".

Если вы все же приняли решение об использовании объектов DataReader на конкретной странице, обязательно рассмотрите вопросы, касающиеся возможности масштабирования. (Ведь каждый запрос этой страницы будет требовать наличия соединения с базой данных!) При определенных обстоятельствах это решение может оказаться верным, но так бывает очень редко.

Производительность ADO.NET

Одна из целей ADO.NET заключается в ограничении возможностей пользовательского кода, связанных с непосредственным взаимодействием с базой данных. С учетом этого факта становится ясно, что добиться приемлемого быстродействия приложения ADO.NET будет непросто. Положительный момент при этом заключается в том, что непосредственный доступ к кэшу данных осуществляется намного быстрее, чем обращение к базе данных.

В ADO.NET производительность можно измерить на двух этапах: на этапе взаимодействия с базой данных (извлечение и обновление данных) и на этапе взаимодействия между кодом приложения и объектами DataSet.

Взаимодействие с базой данных

На производительность приложения ADO.NET влияет лишь *оптимизация* запросов и выбор способа обновления базы данных (см. главу 8, "Обновление базы данных"). О настройке производительности базы данных в этой книге речь не идет, но в разделе "Несколько полезных советов" мы рассмотрим несколько важных вопросов, касающихся этой темы.

Взаимодействие с объектами DataSet

В большинстве случаев настройка взаимодействия с объектами DataSet позволяет сэкономить несколько секунд в рамках пятичасового марафона. Другими словами, это не стоит принимать в расчет, так как в данном случае овчинка выделки не стоит.

Внутри объекта DataSet при ссылке на таблицы, столбцы или значения строк указанные элементы можно представлять в понятном человеку формате (например, `myDataSet.Tables["Customers"]`), что связано с определенными затратами времени на проведение поиска соответствующих имен. Если вам действительно необходимо выиграть несколько миллисекунд, использование порядковых номеров или типизированных объектов DataSet может сократить время поиска, так как разрешение имен проводится однократно при инициализации, а все остальные ссылки на элементы сильно типизированы (и таким образом подпадают под раннюю привязку).

Несколько полезных советов

Со временем я научился использовать ADO.NET оптимальным образом, и поэтому те несколько советов, которые я хочу дать, помогут вам, надеюсь, обойти наиболее распространенные "подводные камни" ADO.NET. В следующих разделах высказываются соображения по ряду вопросов, касающихся разработки приложений, ориенти-

рованных на взаимодействие с базой данных с помощью библиотеки ADO.NET. Возможно, они помогут читателю сохранить хоть какую-то часть волос на его голове в процессе разработки приложения.

Используйте схему объекта DataSet

Я ненавижу обращения к базе данных, особенно не люблю те из них, которые спровоцированы нарушениями схемы. Избежать таких обращений поможет определение схемы объекта DataSet; выполнив его, вы можете заранее получить информацию о том, что данные неправильны. Например, если создать ограничение в объекте DataSet, обеспечивающее уникальность всех номеров социального страхования, то при попытке дважды добавить в базу данных одинакового пользователя, объект DataSet сообщит об ошибке (еще до того, как произойдет неудачная попытка обновления базы данных).

Помимо этого, использование отношений в объекте **DataSet** позволяет передавать связанные деревья объектов, что избавляет от необходимости создания набора классов, предназначенного для отображения реляционной структуры на иерархическую.

Используйте типизированные объекты DataSet для создания уровней бизнес-правил

Как было показано в главе 6, типизированные объекты **DataSet** создаются для перекладывания на них задач обеспечения взаимодействия с базой данных и сериализации. Наследование типизированных классов DataSet позволяет разработать бизнес-правила без необходимости создания полноценного отображения типов приложения на типы базы данных. При использовании типизированных объектов **DataSet** уменьшается необходимость в написании нового интерфейса к данным или какого-либо специального кода. При изменении схемы данных типизированный класс DataSet нужно сгенерировать заново, а также модифицировать код, который от него наследуется. Отметим, что вы создаете меньший объем кода как на начальном этапе, так и на этапе изменения схемы. Хватит работать сверхурочно. Уйдите с работы в шесть и проведите вечер с семьей.

Сокращайте количество обращений к базе данных

Обращения к базе данных стоят дорого. Повторяйте за мной: "обращения к базе данных стоят дорого". Сокращение количества обращений к базе данных позволяет заметно увеличить производительность клиентского кода. При этом нагрузка на сервер баз данных изменится не намного, однако клиентский код (или Web-узел) начнет работать заметно быстрее. ADO.NET поддерживает запросы, возвращающие множественные результаты (следует отметить, что такой возможностью обладают не все управляемые поставщики), а поэтому пакетное извлечение или обновление данных может привести к увеличению производительности.

Кэшируйте данные часто и заранее

ADO.NET предоставляет тщательно продуманный механизм кэширования данных. В Web-приложениях кэширование данных может привести к гигантскому приросту производительности. Избегайте решений, в которых база данных является узким местом всей системы. При кэшировании всей информации, которая может быть использована повторно на среднем уровне или на Web-сервере, приложение начнет не только хорошо масштабироваться, но и быстро работать.

Кэширование данных может оказаться очень полезным и для настольных приложений. Обычно плата за кэширование в таких приложениях не заметна до тех пор, пока количество пользователей не достигнет гигантских размеров, однако преимущества от использования привязки данных Windows-форм способны перевесить любые дополнительные затраты.

Найдите и не отпускайте от себя администратора базы данных

Я очень люблю администраторов баз данных. Если проект может позволить себе такую роскошь, как администратор базы данных, попросите его просмотреть структуру базы данных, включая схему и хранимые процедуры.

Обычно администраторы баз данных являются экспертами в одной или нескольких базах данных. Позвольте им работать в качестве эксперта предметной области. Я часто сталкивался с ситуациями, когда разработчики не хотели пользоваться подобными услугами из-за "подковерных войн" или по политическим причинам. Поверьте мне, администратор базы данных является самым лучшим союзником разработчика.

Наконец, если администратор базы данных будет знаком с текстом программного кода, реализующего взаимодействие с базой данных, он может помочь оптимизировать его, тем самым позволив добиться прироста производительности на этапе выполнения. Администраторы способны на большее, чем резервное копирование базы данных каждый вечер. И обычно их квалификация намного выше, чем необходимо для выполнения такой работы.

Изолируйте разработчиков от базы данных

Помимо создания уровня бизнес-логики с целью изоляции некоторых разработчиков от базы данных следует сделать еще один шаг. Везде, где это возможно, используйте хранимые процедуры. Зачем? Это еще лучше изолирует уровень бизнес-логики от базы данных.

За годы разработки API, скрывающих подробности реализации от конечного разработчика, стало ясно, что подобный подход имеет множество преимуществ. К сожалению, этот шаг не был сделан для API доступа к базам данных. Иногда для создания сложного запроса действительно необходимо хорошо разбираться в используемой базе данных, однако чаще всего работа с базой данных сводится к вызову именованной хранимой процедуры или извлечению таблицы четко определенного формата. Подробности извлечения таблицы из базы данных не должны беспокоить конкретного разработчика. К тому же это позволяет администратору базы данных проводить с ней различные операции (настраивать, масштабировать и даже изменять структуру), не требующие внесения изменений в код приложения. Пока хранимые процедуры возвращают необходимый результат, нет смысла разбираться в том, что и как работает "под капотом". Кроме того, хранимые процедуры предоставляют более высокую производительность, чем динамические запросы.

Недостатком использования хранимых процедур является усложнение создания систем, которые могут работать с различными серверами баз данных. На мой взгляд, этот недостаток компенсируется многочисленными преимуществами. Вместо того чтобы создавать систему, которая будет демонстрировать посредственную производительность на всех серверах баз данных, лучше потратить время на разработку хранимых процедур для каждого конкретного случая, что приведет к заметному увеличению производительности.

Ограничьте использование объекта DataReader на страницах ASP.NET

В некоторых ситуациях объект DataReader может оказаться незаменимым, но стоит быть очень осторожным — его использование часто становится помехой для масштабирования и повышения производительности. Если на странице ASP.NET используется

объект DataReader, то каждый запрос этой страницы потребует создания соединения с базой данных. Механизмов кэширования в этом случае не существует. При желании определенного кэширования можно добиться с помощью компиляции запросов на сервере баз данных, но в любом случае быстродействие будет существенно снижено за счет обращений к базе данных,

Используйте фабрики соединений

Существует несколько причин, по которым можно порекомендовать использование классов фабрик соединений с базой данных.

- **Упрощение кэширования соединений.** Вне зависимости от используемой методики кэширования соединений (каждый управляемый поставщик имеет свою методику), генерирование строки соединения в одном единственном месте позволяет максимизировать количество доступных кэшированных соединений.
- **Изолирование разработчиков.** Создав фабрику соединений, большинство разработчиков могут не беспокоиться о внешнем виде строки соединения вашей организации. На самом деле фабрика соединений должна получать строки соединения от внешнего кода.
- **Изолирование изменений.** Так как соединения создаются в одном единственном месте, это облегчает внесение изменений в строку соединения.

Не помещайте строки соединения в исходный код

Не используйте примеры из этой книги в качестве модели для хранения строк соединений. Такие примеры были созданы для обеспечения их понимания и для удобства чтения, но хранение строки соединения в коде является очень плохой идеей. Лучше всего хранить строки соединения за пределами кода. Встраивая строку соединения в код, программист предоставляет каждому, кто получил доступ к этой сборке, возможность получить также и доступ к базе данных. Аналогично, помещение строки соединения в файл `web.config` на Web-сервере является потенциальной брешью в защите. Обычно информацию такого рода следует помещать в файл `machine.config` или в некое централизованное хранилище, которое может обслуживаться не разработчиками (хорошим выбором может быть Active Directory).

Не предоставляйте пользователям доступ к базе данных

Никогда не предоставляйте конечным пользователям доступ к базе данных только для того, чтобы они могли работать с вашим приложением (это касается как Web-приложений, так и настольных приложений). Если у пользователей есть возможность получить данные с помощью приложения, то они смогут получить доступ к данным и в обход него. По тем же причинам не позволяйте иметь возможность прямого доступа к базе данных учетным записям SYSTEM, `IUSR_XXX` или пользователю ASPNET. В каждом из этих случаев возникновение ошибки в системе безопасности обеспечит взломщику полный доступ к базе данных. Я предпочитаю создавать на сервере баз данных специфичные для приложения учетные записи, которые будут использоваться приложением для получения доступа к базе данных. Например, для выполнения программ-примеров из этой книги я создал пользователя ADONET, который имеет непосредственный доступ только к базе данных `ADONET`. Система безопасности сервера баз данных может ограничить права доступа пользователя ADONET до чтения данных или доступа к хранимым процедурам (что исключает непосредственный доступ к таблицам).

Резюме

На этом этапе вам должно стать ясно, что ADO.NET предоставляет мощные инструменты для разработки масштабируемых решений, ориентированных на взаимодействие с базой данных. Так как конечной целью является сокращение числа обращений к базе данных, в качестве кэша информации можно использовать объект DataSet. Объект DataSet позволяет существенно упростить создание системы кэширования, поскольку все манипуляции с базой данных, реляционно-иерархическое отображение и определение структур данных возлагается на ADO.NET. Разработчику остается лишь создать уровень бизнес-логики как часть типизированного класса DataSet. Кроме того, применение механизма кэширования вывода ASP.NET позволяет сократить количество обращений к кэшу данных,

Вы также узнали, что для оптимального использования ADO.NET необходимо понять что ADO.NET — это всего лишь уровень доступа к базе данных. Другими словами, повышение эффективности взаимодействия приложения с базой данных может быть достигнуто за счет все тех же методов SQL-оптимизации, которые существовали до ADO.NET. Несмотря на то что использование сильно типизированных классов позволяет повысить производительность взаимодействия приложения с ADO.NET, в общем масштабе эффект от таких изменений минимален.

Наконец, существует несколько правил, которые могут помочь разработчику создавать более эффективный код ADO.NET. Большинство этих правил предполагают использование всего потенциала ADO.NET без принесения в жертву безопасности или производительности системы.

Приложение А

Стратегии перехода от ADO к ADO .NET

В этом приложении...

Планирование перехода на ADO.NET

ADO.NET-эквиваленты объектов ADO

Когда Microsoft выпустила .NET, она забыла создать магический инструмент, помогающий осуществлять переход на платформу .NET со старых систем. Таким образом, переносить код ADO на платформу .NET нам придется самостоятельно. Для того чтобы это сделать, необходимо понять фундаментальные различия между ADO и ADO. NET.

Планирование перехода на ADO.NET

Самая сложная часть преобразования кода ADO в код ADO.NET обусловлена не различиями в синтаксисе, а фундаментальной разницей в архитектуре этих технологий. По своей природе интерфейс ADO ориентирован на работу с подсоединенными данными, а ADO.NET — с отсоединенными. Создание моста, связывающего два края этой "пропасти", и является самой трудной частью переноса кода ADO на платформу .NET.

Изменение архитектуры ADO-приложений

При необходимости перенести на платформу .NET фрагмент ADO-кода его следует приспособить для работы в отсоединенной среде. Несмотря на то что это позволит добиться улучшения его производительности и способности к масштабированию, изменить фундаментальную структуру кода ADO будет не просто.

Любой код ADO можно разделить на две логические группы, что позволит облегчить процесс изменении его структуры. Если код ADO используется для создания отчетов или для работы с базой данных в режиме чтения, то переход будет довольно простым, а если он используется для изменения базы данных, все становится гораздо сложнее.

Преобразование кода, использующегося для создания отчетов на основе хранящейся в базе данных информации

Большая часть кода ADO предназначена для выполнения простых запросов и создания отчетов на основе результатов этих запросов. Вне зависимости от применения такого кода — используется он для выполнения простого запроса с целью заполнения списка в настольном приложении или для выполнения сложного запроса с целью отображения финансовой информации на Web-странице — он является достаточно простым для того, чтобы его можно было легко преобразовать. Ниже приведен пример VBScript-кода, создающего запрос к базе данных и добавляющего все возвращенные записи в элемент управления SELECT страницы ASP.

```
...

<SELECT>
<%
Dim cmd, conn, rs

' Создание строки соединения.
const DBDSN = "Provider=SQLOLEDB;" 5 _
              "Server=localhost;" & _
              "Database=ADONET;" & _
              "UID=ADOGUY;" &
              "Password=ADOGUY;"

' Создание объекта соединения.
set conn = CreateObject ("ADODB.Connection")
conn.Open DBDSN

' Создание объекта команды.
set cmd = CreateObject ("ADODB.Command")
cmd.ActiveConnection = conn
cmd.CommandText = "SELECT LastName FROM CUSTOMER"

' Получение результатов запроса.
set rs = cmd.Execute ()

' Итерация по результатам запроса.
while rs.EOF <> true

%>
  <OPTION><%= rs ("LastName") %></OPTION>
<%
  rs .MoveNext
wend

' Закрытие соединения.
conn.Close

set conn = nothing
%>
</SELECT>

...
```

Преобразование этого кода в код ADO.NET делается довольно просто. На самом деле создание страницы ASP.NET потребует больше затрат, чем само преобразование кода, которое можно осуществить путем замены объектов.

1. Объект Connection заменяется объектом **OleDbConnection**.
2. Объект Command заменяется объектом **OleDbCommand**.
3. Объект RecordSet (возвращаемый в результате вызова метода Command. Execute) заменяется объектом DataReader.

Результирующий код ASP.NET будет выглядеть очень похожим на первоначальный код ADO (вместо языка VBScript используется язык C#).

```
...

<select>
<%
```

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;";

// Установка соединения.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = DBDSN;
conn.Open();

// Создание объекта команды.
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandText - "SELECT LastName FROM CUSTOMER";

// Получение результатов запроса.
OleDbDataReader rdr = cmd.ExecuteReader();

// Итерация по результатам запроса.
while (rdr.Read())
{
%>
  <option><%= rdr["LastName"].ToString() %></option>
<%
}

%>
</select>

...
```

Структура **данного кода аналогична структуре кода ADO, а это значит, что она обладает теми же ограничениями — запрос ADO выполняется каждый раз при запросе страницы. В таком случае мы не используем никаких преимуществ ADO.NET. Изменим этот код для того, чтобы иметь возможность кэширования результатов** запроса **в состоянии сеанса.**

```
...

<select>
<%
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;";

// Если это новый сеанс,
// кэшируем данные.
if (Session.IsNewSession)
{
  DataSet dataSet = new DataSet();

  // Создание объекта соединения.
  OleDbConnection conn = new OleDbConnection();
  conn.ConnectionString = DBDSN;
```

```
  // Мы больше не нуждаемся в явном открытии соединения,
  // поскольку собираемся использовать объект DataAdapter.
  // conn.Open();

  // Создание объекта команды.
  OleDbCommand cmd = new OleDbCommand();
  cmd.Connection = conn;
  cmd.CommandText = "SELECT LastName FROM CUSTOMER";

  // Создание объекта DataAdapter.
  OleDbDataAdapter dataAdapter = new OleDbDataAdapter(cmd);

  // Заполнение объекта DataSet.
  dataAdapter.Fill(dataSet, "Customer");

  // Сохранение объекта DataSet в состоянии сеанса.
  Session.Add("CustomerDataSet", dataSet);
}

// Получение результатов запроса.
DataSet custDataSet = (DataSet) Session["CustomerDataSet"];

// Итерация по результатам запроса.
foreach (DataRow row in custDataSet.Tables[0].Rows)
{
%>
  <option><%= row["LastName"].ToString() %></option>
<%
}

%>
</select>
```

Измененный с целью обеспечения возможности кэширования содержимого объекта **DataSet** код все еще очень напоминает первоначальный код ADO. Состояние сеанса используется для кэширования данных на уровне страницы. Важным аспектом является то, что при преобразовании этого кода **мы** также делаем шаг в сторону отсоединенного доступа к данным.

Преобразование кода, использующегося для манипулирования хранящейся в базе данных информацией

Преобразование кода, использующегося для манипулирования хранящейся в базе данных информацией, представляет определенную сложность в связи с тем, что ADO.NET не поддерживает непосредственное редактирование таблиц, так характерное для **ADO.** Ниже приведен пример изменения уже существующих строк и добавления новой строки таблицы.

```
Dim conn, rs

' Создание строки соединения
const DBDSN = "Provider=SQLOLEDB;" & _
              "Server=localhost;" & _
              "Database=ADONET;" б _
              "UID=ADOGUY;" & _
              "Password=ADOGUY;"
```

```
' Элементы перечислений ADO.
const adModeReadWrite  = 3
const adOpenKeyset = 1
const adLockBatchOptimistic = 4

' Создание объекта соединения.
set conn = CreateObject("ADODB.Connection")
conn.Mode = adModeReadWrite
conn.Open DBDSN

' Получение результатов запроса.
set rs = CreateObject("ADODB.RecordSet")
rs.Open "CUSTOMER", conn, adOpenKeyset, _
        adLockBatchOptimistic

' Изменение информации в базе данных.
if rs.EOF <> true then

  ' Редактирование текущей записи.
  rs("Zip") = "12345"

  ' Добавление новой записи.
  rs.AddNew
  rs("FirstName") = "Bob"
  rs("LastName") = "Smith"

  ' Обновление базы данных.
  rs.UpdateBatch

end if

rs.Close
conn.Close
```

ADO.NET не поддерживает подобное обновление данных, но его можно довольно точно скопировать путем использования объекта **DataSet в** отсоединенном режиме доступа.

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;"; .

// Создание объекта соединения.
OleDbConnection conn - new OleDbConnection();
conn.ConnectionString = DBDSN;

// Создание объекта команды.
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Создание объекта DataAdapter.
OleDbDataAdapter dataAdapter = new OleDbDataAdapter(cmd);

// Получение результатов запроса.
DataSet dataSet = new DataSet();
```

```
dataAdapter.Fill(dataSet, "Customer");

// Определение переменной для упрощения доступа к таблице.
DataTable custTbl = dataSet.Tables["Customer"];

// Изменение первой строки таблицы.
custTbl.Rows[0]["Zip"] = "12345";

// Добавление в таблицу новой строки.
DataRow newRow = custTbl.NewRow();
newRow["FirstName"] = "Bob";
newRow["LastName"] = "Smith";
custTbl.Rows.Add(newRow);

// Создание объекта CommandBuilder
// для обновления базы данных.
OleDbCommandBuilder bld = new
                          OleDbCommandBuilder(dataAdapter);

// Обновление базы данных.
dataAdapter.Update(dataSet, "Customer");
```

В итоге мы получаем идентичную функциональность, реализованную в отсоединенном режиме доступа к данным. Следует особо отметить, что в приведенном выше коде нет оператора открытия соединения. Задача объекта DataAdapter заключается в поддержке соединений как можно более короткой длительности; следовательно, он автоматически открывает соединение при загрузке данных и закрывает его сразу же после ее завершения. Аналогичные действия предпринимаются при вызове метода Update. Кроме того, для создания команд обновления и вставки данных используется объект **CommandBuilder**. Этот объект предоставляет функциональность, идентичную той, что скрыта "за кулисами" в ADO, а именно: создание команд SQL с целью сообщить базе данных о требуемых изменениях.

Чего не хватает в ADO.NET?

При оценке того, что должно измениться при переходе на ADO.NET, может оказаться, что некоторые специфичные элементы функциональности ADO попросту не имеют эквивалентов в ADO.NET,

- Курсоры. В ADO можно указать тип курсора, который будет использоваться при перемещении по базе данных. В ADO.NET информация сначала извлекается из базы данных и уже потом используется на стороне клиента. Объект DataReader предоставляет функциональность, напоминающую однонаправленный курсор.
- Редактирование "на месте". Все манипуляции с базой данных в ADO.NET совершаются в отсоединенном режиме доступа. Обновление и добавление записей в соответствии с принципом "добавить...редактировать...обновить" не имеет эквивалента в ADO.NET.
- Блокировка баз данных. В ADO.NET нет способа блокировки базы данных. Изъятие блокировок из ADO.NET было обусловлено их очень высокой ценой и неспособностью к масштабированию. Другими словами, при редактировании строки ее нельзя заблокировать. В такой ситуации необходимо либо создать код для обеспечения параллелизма в отсоединенном режиме доступа к данным (см. главу 8, "Обновление базы данных"), либо использовать объект CommandBuilder для обеспечения оптимистического параллелизма при обновлении базы данных.

ADO.NET-эквиваленты объектов ADO

При анализе кода ADO с целью его преобразования в код ADO.NET может возникнуть соблазн отобразить код ADO на код ADO.NET построчно. Хотя это и не всегда возможно (как было показано в предыдущем разделе), существуют некоторые специфичные стратегии, позволяющие преобразовать типы данных и объекты ADO в типы данных и объекты ADO.NET.

Отображение типов данных ADO на типы данных .NET

Не каждый тип данных ADO имеет прямой эквивалент в .NET. При работе с данными ADO в среде .NET (используя взаимодействие с COM или собственные компоненты, возвращающие данные непосредственно из ADO) необходимо знать отображение типов данных ADO на типы данных .NET Framework. Тип объекта ADO Recordset определяется динамически, что напоминает определение типов объектов DataColumn и DataTable. Для того чтобы отобразить фактический тип объекта ADO на соответствующий тип .NET Framework, необходимо знать отображение типов ADO на типы ADO.NET. Это отображение показано в табл. A.2.[1]

Таблица A.2. Отображение между типами ADO и типами .NET

Тип ADO (DataTypeEnum)	*Тип .NET Framework*	*Тип ADO (DataTypeEnum)*	*Тип .NET Framework*
adEmpty	Null	adDBTime	DateTime
adBoolean	Int16	adDBTimeStamp	DateTime
adTinyInt	SByte	adFileTime	DateTime
adSmallInt	Int16	adGUID	Guid
adInteger	Int32	adError	ExternalException
adBigInt	Int64	adIUnknown	Object
adUnsignedTinyInt	Приводится к Int16	adIDispatch	Object
adUnsignedSmallInt	Приводится к Int32	adVariant	Object
adUnsignedInt	Приводится к Int64	adPropVariant	Object
adUnsignedBigInt	Приводится к Decimal	adBinary	byte[]
adSingle	Single	adChar	String
adDouble	Double	adWChar	String
adCurrency	Decimal	adBSTR	String
adDecimal	Decimal	adChapter	Не поддерживается
adNumeric	Decimal	adUserDefined	Не поддерживается
adDate	DateTime	adVarNumeric	Не поддерживается
adDBDate	DateTime		

[1] Эта таблица взята непосредственно с Web-узла MSDN. Ее можно найти по адресу: msdn.microsoft.com/library/en-us/cpguide/html/cpconadotypemappingtonetframeworktype.asp.

Поставщики и управляемые поставщики

При преобразовании кода ADO в код ADO.NET для доступа к базе данных можно использовать управляемый поставщик OLE DB. Управляемый поставщик OLE DB поддерживает те же строки соединения и поставщики OLE DB, которые использовались раньше, с одним исключением — управляемый поставщик OLE DB специально не поддерживает поставщик ODBC. Для того чтобы использовать поставщик ODBC, вам придется воспользоваться управляемым поставщиком ODBC, который не поставляется в составе Visual Studio .NET или .NET Framework. Управляемый поставщик ODBC можно загрузить с Web-узла Microsoft по адресу: msdn.microsoft.com/downloads/sample.asp?url=/msdn-files/027/001/668/msdncompositedoc.xml. **При работе с управляемым поставщиком ODBC можно использовать те же строки соединения, за исключением атрибута Provider. Так, код**

```
Provider=MSDASQL.1;DSN=ASONET;UID=ADONET;PWD=ADONET;
```

должен быть преобразован следующим образом:

```
DSN=ADONET;UID=ADONET;PWD=ADONET;
```

Преобразование кода установки соединения

ADO поддерживает целый набор различных типов и режимов соединений, которые предоставляют достаточно широкий объем функциональности на стороне сервера. Так как библиотека ADO.NET ориентирована на отсоединенный режим доступа к данным, в ней отсутствует поддержка курсоров, что существенно упрощает создание соединений. Ниже приведен код ADO, предназначенный для открытия соединения в режиме чтения/записи данных.

```
' Создание строки соединения.
const DBDSN = "Provider=SQLOLEDB;" & _
              "Server=localhost;" & _
              "Database=ADONET;" S _
              "UID=ADOGUY;" & _
              "Password=ADOGUY;"

' Элемент перечисления ADO.
const adModeReadWrite  = 3

' Установка соединения.
set conn = CreateObject("ADODB.Connection")
conn.Mode = adModeReadWrite
conn.Open DBDSN
```

Прямое преобразование такого кода провести невозможно, поскольку в ADO.NET отсутствует понятие соединения, предназначенного для чтения/записи данных. Ниже приведен код ADO.NET, примерно соответствующий коду ADO.

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;";

// Установка соединения.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = DBDSN;
conn.Open();
```

ADO и ADO.NET отличаются способом закрытия соединений при отсутствии их явного закрытия. Если бы соединения всегда закрывались явно, все было бы отлично. Как в ADO, так и в ADO.NET соединение закрывается (если оно не было закрыто ранее) при уничтожении объекта. Интерфейс ADO основан на технологии COM, а поэтому для уничтожении объектов используется счетчик ссылок. В .NET используется механизм сбора мусора. Именно в этом и заключается проблема: при освобождении соединения ADO соответствующий объект будет уничтожен, если счетчик ссылок равен нулю, что, в свою очередь, приведет к закрытию соединения. В ADO.NET все работает по-другому.

Сборщик мусора проводит уничтожение объектов только при нехватке памяти, а поэтому нельзя точно сказать, когда будет уничтожен объект после его выхода из области видимости. На самом деле, в долго живущем процессе объект может быть уничтожен даже через несколько дней. С целью смягчения данной проблемы в .NET предоставляется интерфейс IDisposable, поддерживающий единственный метод Dispose(). Интерфейс IDisposable позволяет уничтожать объекты детерминированным образом. Я хочу дать вам два совета, касающихся преобразования кода установки соединения ADO. Во-первых, всегда явно вызывайте метод Connection.close O при завершении работы с соединением. Во-вторых, вызывайте также и метод Dispose(). Вот так должен выглядеть код использования объекта соединения:

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;";

// Создание объекта соединения.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = DBDSN;
conn.Open();

...

// Закрытие соединения и уничтожение его объекта.
conn.Close();
conn.Dispose();
```

Преобразование кода создания объекта команды

Аналогично объектам соединений, объекты команды ADO очень напоминают слои аналоги из ADO.NET. В связи с этим преобразование кода делается довольно просто. Приведенный ниже код ADO:

```
' Создание объекта команды.
set cmd = CreateObject("ADODB.Command")
cmd.ActiveConnection = conn
cmd.CommandText = "SELECT LastName FROM CUSTOMER"
```

будет преобразован в следующий код ADO.NET:

```
// Создание объекта команды.
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT LastName FROM CUSTOMER";
```

Преобразование кода создания объекта Recordset

Объект **ADO Recordset** не имеет единственного эквивалента **в ADO.NET. На** самом деле он имеет два эквивалента. Так, объект **DataReader** очень напоминает однонаправленный объект **Recordset.** В большинстве случаев можно осуществить преобразование по типу "один к одному", в результате чего приведенный ниже код ADO:

```
' Получение результатов запроса.
set rs = cmd.Execute()
```

трансформируется в код ADO.NET:

```
// Получение результатов запроса.
QleDbDataReader rdr - cmd.ExecuteReader();
```

Особенность этого подхода заключается в том, что объект **DataReader** представляет собой однонаправленный курсор, позволяющий перемешаться только вперед и не предусматривающий редактирования данных.

Кроме этого, объект **DataReader** нельзя создать без объектов **Connection и Command. В ADO** при создании объекта **Recordset** ему можно передать информацию, которая в **ADO.NET** содержится в объектах **Command и Connection.** Следовательно, приведенный ниже **код ADO:**

```
' Создание строки соединения.
const DBD3N - "Provider=SQLOLEDB;" & _
              "Server=localhost;" & _
              "Database=ADONET;" & _
              "UID=ADOGUY;" & _
              "Password=ADOGUY;"

' Получение результатов выполнения запроса.
set rs = CreateObject("ADODB.RecordSet")
rs.Open "CUSTOMER", DBDSN
```

будет преобразован в следующий код ADO.NET:

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
                     "Database=ADONET;" +
                     "UID=ADOGUY;" +
                     "Password=ADOGUY;";

// Создание объекта соединения.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = DBDSN;

// Создание объекта команды.
OleDbCommand cmd = new OleDbCommand ();
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Получение результатов выполнения запроса.
OleDbDataReader rdr = cmd.ExecuteReader();
```

Вторым эквивалентом **объекта Recordset** является объект **ADO.NET DataTable.** Объекты **DataTable — таблицы** данных, которые всегда содержатся в объекте **DataSet. Ниже** приведен код **ADO.NET,** в котором используется объект **DataTable.**

```
// Создание строки соединения.
const string DBDSN = "Provider=SQLOLEDB;" +
                     "Server=localhost;" +
```

```
                    "Database=ADONET;" +
                    "UID=ADOGUY;" +
                    "Password=ADOGUY;";

// Создание объекта соединения.
OleDbConnection conn = new OleDbConnection();
conn.ConnectionString = DBDSN;

// Создание объекта команды.
OleDbCommand cmd = new OleDbCommand() ;
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM CUSTOMER";

// Создание объекта DataAdapter.
OleDbDataAdapter dataAdapter = new OleDbDataAdapter(cmd);

// Создание объекта DataSet (который
// будет содержать объект DataTable).
DataSet dataSet = new DataSet();

// Получение результатов выполнения запроса.
dataAdapter.Fill(DataSet, "Customer");

// Определение переменной для упрощения доступа к таблице.
DataTable customerTable = DataSet.Tables["Customer"];
```

Использование объектов ADO Recordset в ADO.NET

Распространенной стратегией в ADO являлось создание компонентов, работающих с отсоединенным или подсоединенным объектом **Recordset.** Объекты **Recordset** использовались в качестве контейнеров данных, которые могли передаваться даже между различными уровнями в распределенном окружении. Так как не все могут позволить себе роскошь преобразования целой системы, вы можете воспользоваться способностью ADO.NET принимать в качестве входного параметра объект **Recordset.** Для того чтобы это сделать, необходимо добавить ссылки на ADO и на компонент, возвращающий объект **Recordset.** Microsoft включила в Visual Studio .NET стандартную сборку взаимодействия с ADODB, поэтому необходимо лишь добавить ссылку на сборку adodb. После этого вы можете использовать объект **Recordset** в качестве параметра при заполнении объекта **DataSet,** как показано ниже:

```
...

// Получение объекта Recordset.
ADODB._RecordSet rs = YourComponent.GetARecordSet();

// Создание объекта DataAdapter без указания
// команды или соединения, так как эта информация
// хранится в объекте Recordset.
OleDbDataAdapter dataAdapter = new OleDbDataAdapter();
DataSet dataSet = new DataSet();

// Заполнение таблицы MyRS объекта DataSet
// с помощью объекта RecordSet.
dataAdapter.Fill(dataSet, rs, "MyRS");
```

Это приведет к заполнению объекта **DataSet,** но обратного пути нет, т.е. не существует встроенной поддержки обновления объекта **Recordset** на основе объекта **DataSet** или **DataTable.**

Резюме

Если изменить свою собственную точку зрения на механизмы взаимодействия с базами данных, можно легко преобразовать код ADO в код ADO.NET. Библиотека ADO.NET изначально проектировалось с целью устранения некоторых присущих ADO проблем, касающихся разработки Web-приложений и систем с высокой нагрузкой. Тем не менее "слепое" преобразование кода ADO в код ADO.NET не позволит избавиться от накопленных ранее проблем, связанных с производительностью и масштабированием. Переход к ADO.NET требует принципиально нового взгляда на работу в отсоединенном режиме доступа к данным.

Предметный указатель

A

Active Server Pages (ASP), 19; 253
ActiveX Data Objects (ADO), 19
ActiveX Data Objects for .NET (ADO.NET), 18
ADO.NET
  краткая история универсального доступа к данным, 18
  объектная модель управляемого поставщика данных, 27
  преимущества использования, 21
  пространства имен, 23
  структуры данных, 23
American National Standards Institute (ANSI), 76
Application Programming Interface (API), 18

C

COM+, 72
Component Object Model (COM), 19
Create, Read, Update, Delete (CRUD), 144

D

Data Access Objects (DAO), 18
Data Source Name (DSN), 37

G

Globally Unique Identifier (GUID), 112

H

Hypertext Transfer Protocol (HTTP), 21

I

Internet Information Server (IIS), 19

M

Microsoft Developer Network (MSDN), 35
Microsoft Message Queue (MSMQ), 258

O

OLE for Databases (OLE DB), 19
Open Database Connectivity (ODBC), 18

S

Structured Query Language (SQL), 76

T

Tabular Data Stream (TDS), 29

X

XML Schema Definition (XSD), 139

Д

Делегат
  DataColumnChangeEventHandler, 121
  DataRowChangeEventHandler, 121

И

Интеграция с XML
  .NET и **XML,** 212
  заполнение объекта DataSet данными из XML-файла, 221
  класс DataSet и XML, 212
  класс XmlDataDocument, 229
  поиск данных в объекте DataSet с помощью запросов XPath, 231
  преобразование данных объекта DataSet в формат XML, 212
  преобразование объекта DataSet с помощью языка XSLT, 233
  пространства имен объекта DataSet, 221
  сохранение данных объекта DataSet в формате XML, 218
  стратегии использования XML-документа в формате DiffGram, 225
  схема объекта DataSet, 226
Интерфейс
  IDataReader, 90
  IDbCommand, 24
  IDbConnection, 49

IDbTransaction, 178
IList, 239
Исключения ADO.NET, 53

K

Команда, 55
выполнение, 57
использование параметров, 58
пакетные запросы, 72
параметризированные запросы, 65
создание, 55
создание оболочки для хранимой процедуры, 60
типы, 56
транзакция, 67
Кэширование данных
на стороне клиента, 249
на стороне сервера, 249
несколько полезных советов, 261

M

Метод
DataRowCollection.Remove(), 158
DataRowCollection.RemoveAt(), 158
Метод
AcceptChanges(), 164
BeginEdit(), 162
BeginLoadData(), 163
CancelEdit(), 162
ChangeDatabase(), 39
CommandBuider.RefreshSchema(), 183
Commit(), 70
Convert.ChangeType(), 162
DataAdapter.Fill(), 101; 102
DataAdapter.FillSchema(), 108
DataAdapter.Update(), 26; 179; 206
DataBind(), 246
DataGrid.SetDataBinding(), 240
DataRelationCollection.Add(), 114
DataRow.Delete(), 158
DataRow.HasVersion(), 166
DataRow.IsNull(), 161
DataRowCollection.Add(), 157
DataRowCollection.InsertAt(), 158
DataSet.AcceptChanges(), 226
DataSet.GetChanges(), 220
DataSet.Merge(), 176
DataSet.ReadXml(), 221
DataSet.WriteXml(), 219
DataTable.AcceptChanges(), 164
DataTable.NewRow(), 157
DataTable.NewRowFromBuilder(), 152
DataTable.Select(), 173
DataView.Find(), 174
DataView.FindRows(), 174
EndEdit(), 162
EndLoadData(), 163
ExecuteNonQuery(), 57
ExecuteReader(), 57
ExecuteScalar(), 57
ExecuteXmlReader(), 57
GetChildRows(), 169
GetName(), 82
GetOrdinal(), 82
GetParentRow(), 169
GetParentRows(), 169
GetRowType(), 152
GetSchemaTable(), 87
IDataInitialize::GetDataSource(), 43
IDbCommand.ExecuteNonQuery(), 24
IDbCommand.ExecuteReader(), 79
IDbCommand.ExecuteScalar(), 24
IsDBNull(), 83
LoadDataRow(), 163
MoveNext(), 24
NextResult(), 86
OleDbConnection.GetOleDbSchema Table(), 50
OleDbConnection.ReleaseObjectPool(), 42
Read(), 79
ReadXml(), 104
Register(), 145
RejectChanges(), 164
Rollback(), 70
Save(), 71
Table Mappings.Add(), 103
ToString(), 90
ToStringArray(), 92
WriteXmlSchema(), 227

*

O

Обработка ошибок в .NET, 51
Объект DataAdapter
вывод схемы объекта DataSet, 107
использование свойства TableMappings, 101
множественные объекты DataTable, 100
обновление объекта DataSet, 179
описание, 98
создание объекта DataSet на основе информации, хранящейся в базе данных, 99
Объект DataReader

доступ к данным, 80
масштабирование, 259
обработка множественных результирующих наборов данных, 86
описание, 79
привязка данных, 248
работа с метаданными, 86
создание, 79
функциональность, 80
чтение данных, 76
Объект DataSet
версии строк, 164
вывод схемы с помощью объекта DataAdapter, 107
добавление строк, 157
заполнение данными, 98
изменение данных, 156
использование автоинкрементных столбцов, 124
использование вычисляемых столбцов, 125
использование свойства TableMappings, 101
использование языка XSD для определения схемы, 109
множественные объекты DataTable, 100
описание, 96
определение схемы, 106
первичные ключи, 110
перемещение вдоль отношений, 167
поиск данных с помощью запросов XPath, 231
поиск с помощью метода DataTable. Select, 173
поиск с помощью объекта Data View, 174
преобразование данных объекта DataSet в формат XML, 212
преобразование с помощью языка XSLT, 233
пространства имен объекта DataSet, 221
реализация деструктивного параллелизма, 201
реализация оптимистического параллелизма, 180
реализация пессимистического параллелизма, 197
слияние, 175
создание на основе XML-документа, 103
создание на основе информации, хранящейся в базе данных, 99
создание ограничений, 115
создание отношений, 112
создание программным путем, 105
создание схемы программным путем, ПО
сортировка данных с помощью объекта DataView, 170
состояние строки, 166
сохранение данных объекта DataSet в формате XML, 218
структура, 96
схема столбца, 121
типизированный объект DataSet
создание, 131
использование, 142
настройка сгенерированного кода с помощью специальных аннотаций, 140
упрощение уровня бизнес-объектов, 143
триггеры, 119
удаление строк, 158
фильтрация данных с помощью объекта DataView, 171
чтение и запись значений столбцов строки, 159

П

Пакетные запросы, 72
Параллелизм, 179
деструктивный параллелизм, 179; 201
обработка нарушений параллелизма. 193
оптимистический параллелизм, 179; 180
пессимистический параллелизм, 179; 197
Переход к ADO.NET, 265
ADO.NET-эквиваленты объектов ADO, 271
изменение архитектуры ADO-приложений, 265
использование объектов ADO Recordset в ADO.NET, 275
отображение типов данных ADO на типы данных .NET, 271
поставщики и управляемые поставщики, 272
преобразование кода создания объекта Recordset, 274
преобразование кода создания объекта команды, 273
преобразование кода установки соединения, 272

Перечисление
AcceptRejectRule, 115
CommandType, 56
DataRowAction, 120
DataRowVersion, 165
DataViewRowState, 172
IsolationLevel, 71
MappingType, 217
MissingSchemaAction, 108
ParameterDirection, 60
SqlDbType, 59
XmlReadMode, 224
XmlWriteMode, 219
Привязка данных, 235
ASP.NET, 246
Windows-формы, 235
привязка к элементу управления DataGrid, 240
привязка с помощью объекта DataReader, 248
привязка типа "родитель-потомок", 242
простая привязка, 237; 246
сложная привязка, 239; 247
Пространство имен
System.Data, 23; 97
System.Data.Common, 30; 98
System.Data.Odbc, 30; 98
System.Data.OleDb, 30; 98
System.Data.OracleClient, 30; 98
System.Data.SqlClient, 30; 98

C

Свойство
BindingContext, 245
DataAdapter.ContinueUpdateOnError, 194
DataBindings, 237
DataColumn.Expression, 126
DataSource, 247
DataTable.HasErrors, 194
DataTextField, 247
DataTextFormatString, 247
DataValueField, 247
DataView.Sort, 170
DeleteCommand, 98
InsertCommand, 98
IsOnPrimaryKey, 118
MissingSchemaAction, 107
Position, 245
PrimaryKey, 27
RowError, 194
RowFilter, 171
Rows, 119
SelectCommand, 98
SqlConnection.ConnectionTimeout, 35
TableMappings, 101
UpdateCommand, 98
Событие
ColumnChanged, 119
ColumnChanging, 119
DataColumnChanged, 121
DataColumnChanging, 121
DataRowChanged, 121
DataRowChanging, 121
Disposed, 45
InfoMessage, 45
RowChanged, 119
RowChanging, 119
RowDeleted, 119
RowDeleting, 119
RowUpdating, 182
StateChange, 45
Соединение, 33
встроенная система безопасности, 39
изменение базы данных, 39
организация пула соединений, 41
управляемый поставщик данных ODBC, 43
управляемый поставщик данных OLE DB, 42
управляемый поставщик данных Oracle, 43
управляемый поставщик данных SQL Server, 41
продолжительность ожидания установки соединения, 38
события объекта Connection, 45
строка соединения, 34
управляемый поставщик ODBC, 37
управляемый поставщик OLE DB, 36
управляемый поставщик Oracle, 37
управляемый поставщик SQL Server, 35
фабрика соединений, 48

T

Транзакция, 67
службы Enterprise Services и COM+, 72
точки сохранения в транзакциях SQL Server, 71
уровни изоляции, 70

Научно-популярное издание

Шон Вилдермьюс

Практическое использование ADO.NET. Доступ к данным в Internet

Литературный редактор С. *Г. Езерницкая*
Верстка *М.А. Удалое*
Художественный редактор *С. А. Чернокозинский*
Корректоры *Л.А. Гордиенко, О. В. Мишутина*

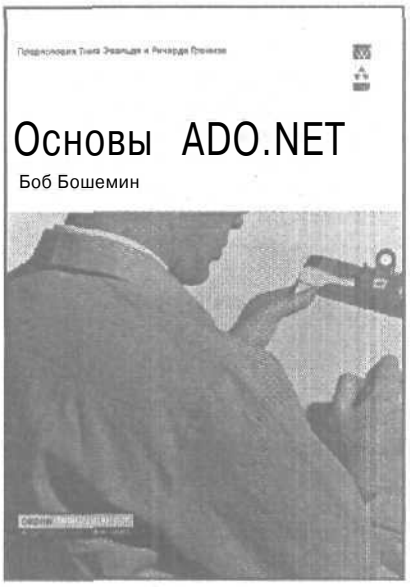
Издательский дом "Вильяме".
101509, Москва, ул. Лесная, д. 43, стр. 1.
Изд. лиц. ЛР № 090230 от 23.06.99
Госкомитета РФ по печати.

Подписано в печать 22.04.2003. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 23,22. Уч.-изд. л. 14,95.
Тираж 3000 экз. Заказ № 2851.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций,
197110, Санкт-Петербург, Чкаловский пр., **15.**

ОСНОВЫ **ADO.NET**

Боб Бошемин



www.wiliiam5publishing.com

в продаже

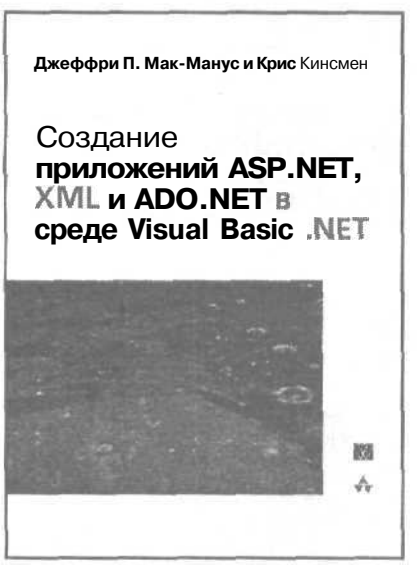
Разработанная компанией Microsoft технология доступа к данным ADO.NET позволяет Windows-приложениям осуществлять взаимодействие с базами данных множества различных производителей, что является одним из основных требований при создании Internet-приложений и построении распределенных вычислительных сред.

В этой книге содержится исчерпывающее описание ADO.NET, рассмотрены возможности классов, интерфейсов, свойств и методов. Кроме этого, на примерах различных типов структур данных продемонстрировано использование ADO.NET при решении всевозможных задач, связанных с обеспечением доступа к данным. Особое внимание уделено первичным целям проектирования ADO.NET – достижению баланса между универсальностью и эффективностью, обеспечению возможности масштабирования, параллельной обработке данных и устойчивости. Программисты, использующие такие API доступа к данным, как OLE DB, ADO, ODBC и JDBC, могут обращаться к этой книге как к справочнику, содержащему информацию о соответствии классов и функций вышеназванных API классам и методам ADO.NET.

Основы ADO.NET – это наиболее полный источник информации по одной из ключевых частей .NET Framework. Данная книга поможет вам раскрыть все возможности ADO.NET вне зависимости от того, являетесь ли вы новичком или уже успели приобрести определенный опыт в этой области.

СОЗДАНИЕ ПРИЛОЖЕНИЙ ASP.NET, XML И ADO.NET В СРЕДЕ VISUAL BASIC.NET

Джеффри П. Мак-Манус, Крис Кинсмен



www.wilHamspublishing.com

в продаже

Данная книга представляет собой руководство для разработчиков, которое содержит подробное описание средств языка Visual Basic .NET, опирающихся на возможности классов общей среды выполнения (CLR — Common Language Runtime) и инфраструктуры .NET. В ней значительное внимание уделено вопросам перехода на новую инфраструктуру поддержки приложений и приведены исчерпывающие рекомендации по подготовке существующего кода ASP для работы на платформе ASP.NET. В книге подробно описаны классы инфраструктуры страницы в среде ASP.NET, включая сам объект Page, его дочерние классы и элементы управления пользовательским интерфейсом (элементы управления HTML и серверные элементы управления). Рассматриваются методы проектирования пользовательского интерфейса для таких устройств с малыми форм-факторами, как мобильные телефоны и карманные компьютеры. Книга предназначена для программистов начального и среднего уровня, стремящихся освоить новые средства языка Visual Basic .NET.

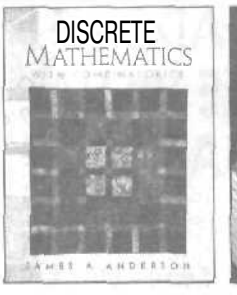
Мир книг по компьютерным наукам от издательской группы "ДИАЛЕКТИКА-ВИЛЬЯМС"



ISBN 5-8459-0179-0



ISBN 5-8459-0109-X





ISBN 5-8459-0122-7



ISBN 5-8459-0310-6



ISBN 5-8459-0262-2



ISBN 5-8459-0388-2



ISBN 5-8459-0192-8



ISBN 5-8459-0330-0



ISBN 5-8459-0384-X





www.dialektika.com

КАЯ

www.williamspubli5hing.com

CISCO SYSTEMS

www.ciscopress.ru

Мир книг по технологиям программирования от издательской группы *"ДИАЛЕКТИКА - ВИЛЬЯМС"*



ISBN 5-8459-0080-8



ISBN 5-8459-0189-8



ISBN 5-8459-0261-4



ISBN 5-8459-0384-X



ISBN 5-8459-0179-0



ISBN 5-8459-0360-2



ISBN 5-8459-0210-X



ISBN 5-8459-0192-8



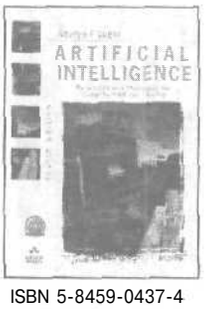
ISBN 5-8459-0388-2



ISBN 5-8459-0330-0



ISBN 5-8459-0438-2



ISBN 5-8459-0437-4



www.dialektika.com



www.williamspublishing.com



www.ciscopress.ru

Мир книг по базам данных от издательской группы "ДИАЛЕКТИКА-ВИЛЬЯМС"



ISBN 5-8459-0138-3



ISBN 5-8459-0109-X



ISBN 5-8459-0384-X



ISBN 5-8459-0297-5



ISBN 5-8459-0355-6



ISBN 5-8459-0270-3



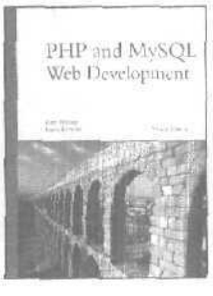
ISBN 5-8459-0158-8



ISBN 5-8459-0291-6



ISBN 5-8459-0276-2





ISBN 5-8459-0302-5



ISBN 5-8459-0179-0

fcl3

www.dialektika.com

www.williamspublishing.com

Cisco Systems

www.ciscopress.ru

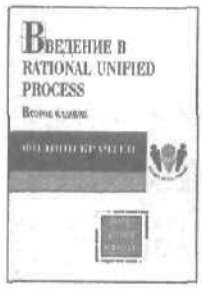
Мир книг по UML от издательской группы *"ДИАЛЕКТИКА - ВИЛЬЯМС"*



ISBN 5-8459-0276-2



ISBN 5-8459-0250-9



ISBN 5-8459-0239-8



ISBN 5-8459-0203-7



ISBN 5-8459-0265-7



ISBN 5-8459-0368-8



ISBN 5-8459-0346-7



ISBN 5-8459-0301-7



ISBN 5-8459-0393-9



ISBN 5-8459-0299-1



ISBN 5-8459-0425-0



ISBN 5-8459-0355-6

www.dialektika.com www.williamspublishing.com www.ciscopress.ru

Мир книг по разработке программного обеспечения от издательской группы "ДИАЛЕКТИКА-ВИЛЬЯМС"



ISBN 5-8459-0276-2



ISBN 5-8459-0394-7



ISBN 5-8459-0275-4



ISBN 5-8459-0367-X



ISBN 5-8459-0406-4



ISBN 5-8459-0365-3



ISBN 5-8459-0336-X



ISBN 5-8459-0329-7



ISBN 5-8459-0413-7



ISBN 5-8459-0233-9



ISBN 5-8459-0270-3



ISBN 5-8459-0448-X

www.dialektika.com

www.williamspublishing.com

www.ciscopress.ru

"Это моя любимая книга по ADO.NET. Ее автор, безусловно, обладает уникальными знаниями в области разработки приложений баз данных и прекрасно владеет материалом. Данная книга представляет интерес как для опытных разработчиков, так и для тех, кто делает первые шаги в освоении платформы Microsoft .NET".

— Гленн Тиммз, старший инженер по разработке программного обеспечения, New Dawn Technologies (ранее инженер по поддержке процесса разработки, Microsoft Corporation)

Эта книга является практическим руководством по использованию первой библиотеки доступа к данным, спроектированной специально для упрощения создания Web-приложений. Содержащийся в книге материал поможет разработчикам изучить основные концепции ADO.NET и познакомиться с практическими методами решения распространенных задач.

На первых страницах книги автор предлагает совершить небольшой экскурс в историю создания компанией Microsoft технологий универсального доступа к данным и проследить эволюционный путь ADO.NET. Большая часть книги посвящена использованию библиотеки ADO.NET для взаимодействия с базами данных и остальной частью .NET Framework. Кроме того, автор дает ряд полезных советов по созданию масштабируемых и высокопроизводительных приложений. Книга включает в себя множество примеров исходного кода на языке C#, а также имеет Web-узел поддержки по адресу: *www.adoguy.com/book,* на котором, помимо примеров на языке Visual Basic .NET, можно найти различные средства, упрощающие создание ADO.NET-приложений. В конце книги автор подробно излагает стратегию преобразования кода ADO в код ADO.NET.

В книге рассмотрены следующие темы:

- Работа с данными в отсоединенном режиме
- Подключение к базе данных с использованием средств ADO.NET
- Использование объектов Command и DataReader
- Создание объекта DataSet
- Создание и использование типизированных классов DataSet
- Манипулирование данными, содержащимися в объекте DataSet
- Обновление базы данных на основании информации, содержащейся в объекте DataSet
- Интеграция ADO.NET и XML
- Привязка данных к элементам графического интерфейса
- Повышение масштабируемости и производительности ADO.NET-приложений

Лаконичное изложение материала в сочетании с полезными советами и подробными примерами делает данную книгу бесценным источником знаний для всех разработчиков, желающих приобрести навыки практического использования ADO.NET.

Шон Вилдермьюс — основатель Web-узла ADOGuy.com и разработчик приложений баз данных более чем с 16-летним стажем. Шон создал множество приложений баз данных для компаний, занимающихся бухгалтерским учетом, торговлей недвижимостью, предоставлением медицинских услуг, а также различных услуг через Internet. Его статьи печатаются во многих журналах, среди которых *MSDN Magazine* и *Windows 2000 Magazine.*



Книги серии **Microsoft .NET Development Series** написаны и одобрены ведущими специалистами и разработчиками технологий Microsoft .NET, включая команду создателей платформы Microsoft .NET и специалистов компании DevelopMentor. Освещая вопросы проектирования, структуры и реализации Microsoft .NET, эти книги предназначены для разработчиков и студентов, поддержавших .NET-революцию.

ВИЛЬЯМС www.williamspublishing.com

Microsoft .net

ADDISON WESLEY http://www.adoguy.com/book
http://www.awprofessional.com/msdotnetseries

ISBN 5-8459-0450-1

9 785845 904508 03044