Технология доступа к базам данных

ADO.NET

Урок №3

Фабрика провайдеров, обзор асинхронных режимов доступа

Содержание

Поставщики данных DbProviderFactory 3
Общие принципы 3
Пример использования 5
Дополнительные возможности DbDataAdapter 17
Управление данными в DataSet 17
Поддержка транзакций 19
Асинхронная работа с БД 26
Классический поход к асинхронности 27
Использование callback методов 30
Использование класса WaitHandle 37
Использование опроса дополнительного потока . 42
Домашнее задание . . . . . `43

Поставщики данных DbProviderFactory

Общие принципы

Вы уже получили представление о том, как происходит работа с источниками данных в ADO.NET. Давайте сейчас вернемся к тому моменту, что в нашем распоряжении есть иерархии классов для доступа к разным источникам данных. Эти классы являются производными от абстрактных классов. Например, от абстрактного DbConnection, происходят классы OdbcConnection, OleDbConnection, OracleConnection и SqlConnection. В наших приложениях мы использовали производные классы SqlXXX, потому, что мы использовали сервер MS SQL Server. Если бы нам надо было работать с Access, мы бы использовали классы OleDbXXX, если бы мы хотели работать с Oracle, то, понятно, использовали бы классы OracleXXX. И вот здесь напрашивается такой вопрос — а можно ли написать код, который будет одинаковым для разных источников данных? Чтобы в случае, когда придется поменять сервер БД, нам не пришлось бы переписывать и пересобирать наше приложение. Это необходимо для того, чтобы при изменении поставщика данных у клиента, у которого работает ваше приложение, вам не пришлось бы ехать к нему в командировку и изменять код.

Итак, наша цель — написать код, инвариантный относительно источника данных. Понятно, что такой код должен

быть параметризированным. Т.е. код должен получать каким-либо образом указание, с каким источником данных надо работать и дальше должен однотипно выполнять все необходимые приложению действия. Для написания такого кода нам надо познакомиться с классом DbProviderFactory.

Каждый поставщик данных включает в свой состав класс-фабрику, производный от класса DbProviderFactory. Если получить доступ к этому классу-фабрике для конкретного поставщика данных, то с его помощью можно создать объекты Connection, Command, Adapter и другие для этого поставщика данных.

Откуда приложение может узнать, какие поставщики данных доступны ему и как получить фабрику? В этом приложению поможет операционная система Windows. На каждом компьютере под управлением Windows есть файл с именем machine.config. В этом файле и зарегистрированы все доступные поставщики данных. Вы не будете удивлены, когда узнаете, что machine.config является XML файлом. Следовательно, прочитать этот файл и извлечь необходимую информацию будет просто. На моем компьютере этот файл находится по пути: C:\Windows\Microsoft.NET\Framework\v2.0.50727\CONFIG. Раздел этого файла, в котором находится нужная нам информация, выглядит так:

```
<system.data>
    <DbProviderFactories>
       <add name="Odbc Data Provider" invariant="System.
Data.Odbc" description=".Net Framework Data Provider
for Odbc" type="System.Data.Odbc.OdbcFactory,
System.Data, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
```

```
        <add name="OleDb Data Provider"
invariant="System.Data.OleDb" description=".Net
Framework Data Provider for OleDb" type="System.Data.
OleDb.OleDbFactory, System.Data, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
        <add name="OracleClient Data Provider"
invariant="System.Data.OracleClient" description=".
Net Framework Data Provider for Oracle" type="System.
Data.OracleClient.OracleClientFactory, System.Data.
OracleClient, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
        <add name="SqlClient Data Provider"
invariant="System.Data.SqlClient" description=".
Net Framework Data Provider for SqlServer"
type="System.Data.SqlClient.SqlClientFactory,
System.Data, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"/>
        <add name="Microsoft SQL Server Compact Data
Provider 4.0" invariant="System.Data.SqlServerCe.4.0"
description=".NET Framework Data Provider for
Microsoft SQL Server Compact" type="System.Data.
SqlServerCe.SqlCeProviderFactory, System.Data.
SqlServerCe, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=89845dcd8080cc91"/>
    </DbProviderFactories>
</system.data>
```

Пример использования

Давайте создадим приложение, которое реализует такую логику:

- прочитает из machine.config список допустимых поставщиков данных и выведет его пользователю (в Combobox и DataGridView);
- позволит пользователю выбрать из списка требуемого поставщика (Combobox) и прочитает для этого поставщика строку подключения (из AppConfig);

- позволит пользователю ввести SQL запрос для БД (соответствующей прочитанной строке подключения);
- выполнит запрос и выведет результаты его работы (в DataGridView);

Чтобы продемонстрировать, что написанное нами приложение может работать с разными поставщиками данных, создадим еще таблицу в MS Access. Наше приложение будет работать с нашей БД Library и с таблицей в MS Access. Если у вас есть другие доступные БД, можете использовать их. Пусть таблица в MS Access тоже называется Books, хотя имя ее может быть произвольным. Структура этой таблицы и данные в ней будут отличаться от структуры и данных в БД Library. Вообще говоря, таблица может быть совершенно произвольной, ее структура и данные в ней не имеют никакого значения.

Перед непосредственной разработкой приложения кратко рассмотрим те новые классы, с которыми надо будет работать.

Прежде всего это класс DbProviderFactories. Используя его статический метод GetFactoryClasses(), мы можем получить список доступных источников данных. Метод GetFactoryClasses() читает информацию из файла machine.config и возвращает результат в виде объекта DataTable.

Используя другой статический метод этого класса — GetFactory(), мы сможем получить фабрику для конкретного поставщика данных. Методу GetFactory() в качестве параметра надо указать, фабрику какого поставщика мы

хотим получить. Например, чтобы получить фабрику для MS SQL Server, этому метода надо передать строку «System.Data.SqlClient». Откуда берутся эти строки для метода GetFactory() увидите чуть позже.

Фабрика конкретного поставщика данных — это объект типа DbProviderFactory. Имея в своем распоряжении такую фабрику, мы можем использовать фабричные методы CreateConnection(), CreateCommand(), и CreateDataAdapter() и получить объекты DbConnection, DbCommand и DbDataAdapter для конкретного поставщика данных. Все остальное вы делать уже умеете.

Как мы уже договорились, чтобы иметь в своем распоряжении источник данных, отличный от System.Data.SqlClient, создадим таблицу в MS Access 2007. Таблица может быть совершенно произвольной, нам просто надо убедиться, что мы получим к ней доступ. Мы будем подключаться к этой таблице через поставщика данных System.Data.OleDb. В моем случае созданная БД называется Library.accdb и для простоты этот файл располагается в корневой папке создаваемого приложения.

- Запускаем Visual Studio 2015 и создаем новый Windows Forms проект с окном такого вида: две кнопки — “Get All Providers” и “Execute request”;
- комбобокс — для выбора провайдера;
- два текстовых поля — для отображения строки подключения (Readonly) и для ввода запросов;
- DataGridView — для отображения результатов выполнения запросов.

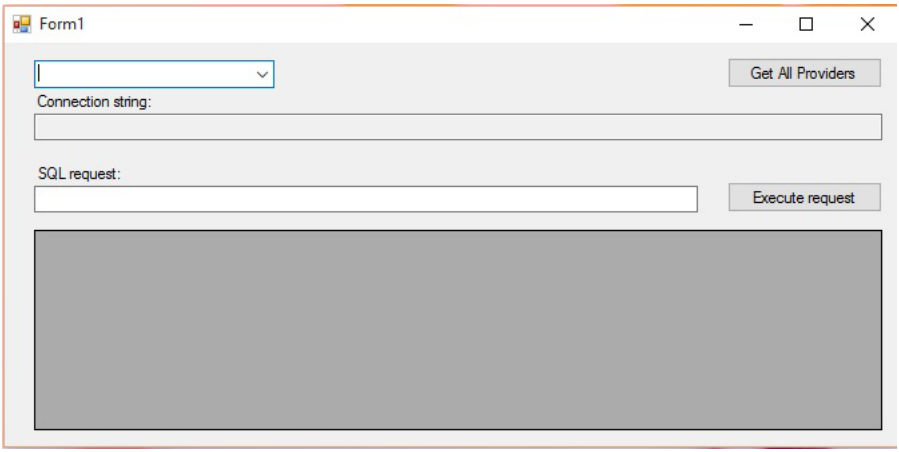

Рис. 1. Главное окно приложения

Добавьте в конфигурационный файл созданного приложения строки подключения к тем БД, с которыми вы планируете работать. В моем случае App.config выглядит так:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <clear/>
    <add name="MyLibrary"
     providerName="System.Data.SqlClient"
     connectionString=
     "Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=Library; Integrated Security=SSPI;"
    />
    <add name="MyAccess"
     providerName="System.Data.OleDb"
     connectionString=
     "Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=Library.accdb;Persist Security Info=False;"
    />
    </connectionStrings>
</configuration>
```

Добавьте в проект ссылку на пространство имен System.Configuration и директиву using System.Configuration.

Приведите код приложения к такому виду:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Data.Common;
using System.Configuration;

namespace ADO_ProviderFactory
{
   public partial class Form1 : Form
   {
      DbConnection conn=null;
      DbProviderFactory fact=null;
      string providerName="";

      public Form1()
      {
         InitializeComponent();
         button2.Enabled = false;
      }

      /// <summary>
      /// Читаем список зарегистрированных
      /// поставщиков данных
      /// возвращаем список в виде таблицы
      /// отображаем список в dataGridView1
      /// значения из колонки InvariantName
      /// созданной таблицы
      /// заносим в comboBox1
      /// </summary>
      /// <param name="sender"></param>
```

```
        /// <param name="e"></param>
        private void button1_Click(object sender, EventArgs e)
        {
            DataTable t = DbProviderFactories.GetFactoryClasses();
            dataGridView1.DataSource = t;
            comboBox1.Items.Clear();

        foreach (DataRow dr in t.Rows)
        {
            comboBox1.Items.Add(dr["InvariantName"]);
        }
    }
    /// <summary>
    /// по выбранному в comboBox1 значению с помощью
    /// метода GetFactory()
    /// создаем фабрику для выбранного поставщика
    /// с помощью метода GetConnectionStringByProvider()
    /// получаем из App.config строку подключения к БД
    /// отображаем строку подключения в textBox1
    /// и сохраняем в глобальной строке providerName
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void comboBox1_SelectedIndexChanged(object
                                        sender, EventArgs e)
    {
        fact = DbProviderFactories.GetFactory(comboBox1.
             SelectedItem.ToString());
        conn = fact.CreateConnection();
        providerName =
             GetConnectionStringByProvider(comboBox1.
                                  SelectedItem.ToString());
        textBox1.Text = providerName;
    }
    /// <summary>
    /// читаем из App.config строку подключения с значением
    /// providerName,
   /// совпадающим с параметром providerName
```

```
/// возвращаем найденную строку подключения или null
/// </summary>
/// <param name="providerName"></param>
/// <returns></returns>
static string GetConnectionStringByProvider(string
   providerName)
{
   string returnValue = null;

   // читаем все строки подключения из App.config
   ConnectionStringSettingsCollection settings =
      ConfigurationManager.ConnectionStrings;

   // ищем и возвращаем строку подключения
   // для providerName
   if (settings != null)
   {
      foreach (ConnectionStringSettings cs in settings)
      {
         if (cs.ProviderName == providerName)
         {
            returnValue = cs.ConnectionString;
            break;
         }
      }
    }
        return returnValue;
   }

   /// <summary>
   /// имея в свом распоряжении фабрику для выбранного
   /// поставщика выполняем стандартные действия с БД
   /// для демонстрации работоспособности кода
   /// </summary>
   /// <param name="sender"></param>
   /// <param name="e"></param>
   private void button2_Click(object sender, EventArgs e)
   {
      conn.ConnectionString = textBox1.Text;
      // создаем адаптер из фабрики
```

```
            DbDataAdapter adapter = fact.CreateDataAdapter();
            adapter.SelectCommand = conn.CreateCommand();
             adapter.SelectCommand.CommandText = textBox2.
                   Text.ToString();
             // выполняем запрос select из адаптера
             DataTable table = new DataTable();
             adapter.Fill(table);
             // выводим результаты запроса
             dataGridView1.DataSource = null;
             dataGridView1.DataSource = table;
        }
        private void textBox2_TextChanged(object sender, EventArgs e)
        {
            if (textBox2.Text.Length > 5)
                button2.Enabled = true;
            else
                button2.Enabled = false;
        }
    }
}
```

Посмотрим, как работает наше приложение, и разберем подробнее некоторые моменты. Запустите приложение и нажмите кнопку Get All Providers. В нижней части окна вы увидите таблицу, в которой отображена информация о зарегистрированных поставщиках данных из файла machine.config. Особое внимание обратите на колонку InvariantName. В ней содержится уникальное имя для каждого поставщика. Именно по этому имени мы будем выбирать конкретную фабрику. Значения из этой колонки всегда размещаются в строке подключения в файле App.config в атрибуте providerName. Наше приложение заносит в комбобокс именно значения InvariantName для зарегистрированных поставщиков данных.

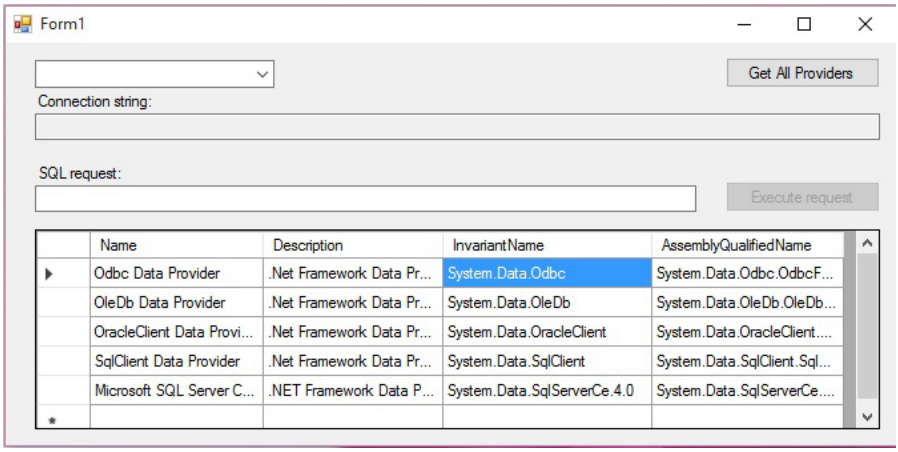

Рис. 2. Список поставщиков данных

Выберите в комбобоксе требуемое имя поставщика данных, в нашем случае — System.Data.SqlClient или System.Data.OleDb. Ведь для других поставщиков мы не предусмотрели строки подключения.

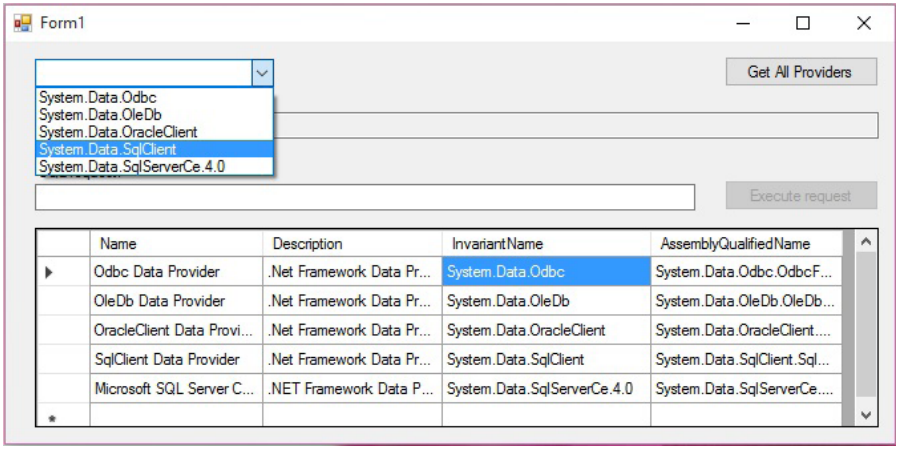

Рис. 3. Выбор поставщика данных

После выбора поставщика вы увидите строку подключения в текстовом поле. Обратите внимание, что кнопка

выполнения запроса неактивна до тех пор, пока вы не занесете запрос во второе текстовое поле.

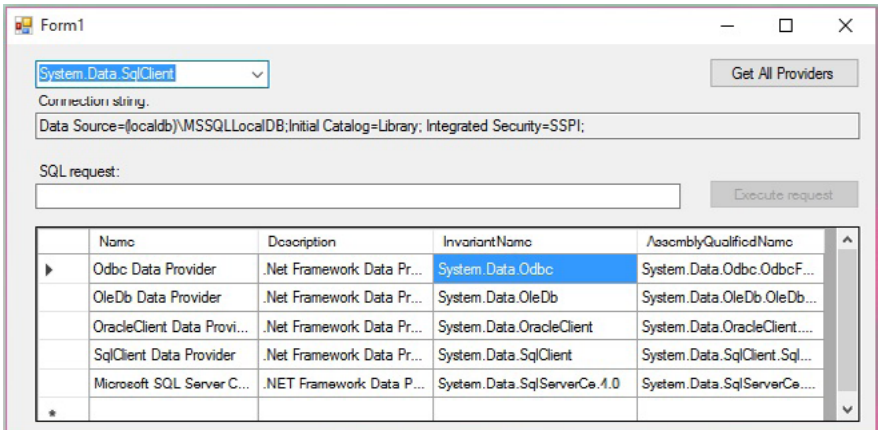

Рис. 4. Отображение строки подключения

Занесите в текстовое поле для запросов какой-либо запрос к своей БД и нажмите кнопку Execute request. Запрос будет выполнен, а его результаты будут выведены в dataGridView1. Итак, по крайней мере, с поставщиком данных System.Data.SqlClient, наш код работает.

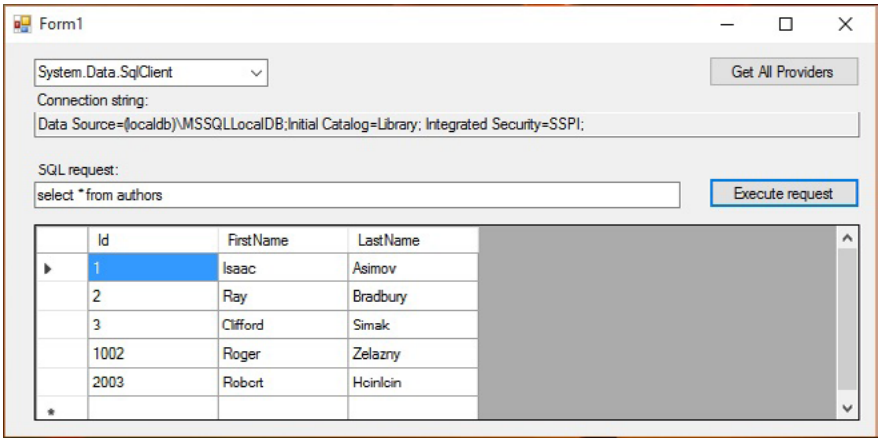

Рис. 5. Отображение результатов запроса

Двигаемся дальше, чтобы проверить, позволяет ли этот код работать с другими поставщиками данных. Теперь выполним в нашем приложении те же действия для второй БД. Выберем в комбобоксе значение System.Data.OleDb. Выбираем именно это значение потому, что для этого поставщика данных у нас есть БД и строка подключения к этой БД в конфигурационном файле нашего приложения. Строка подключения с именем MyAccess использует поставщика данных System.Data.OleDb, как это видно по значению атрибута providerName. Теперь введем подходящий для этой БД запрос select и выполним этот запрос, нажав кнопку Execute request. В нижней части окна мы увидим результат выполнения запроса. При выборе поставщика данных System.Data.OleDb у меня получилась такая картинка (запрос select я ввел вручную):

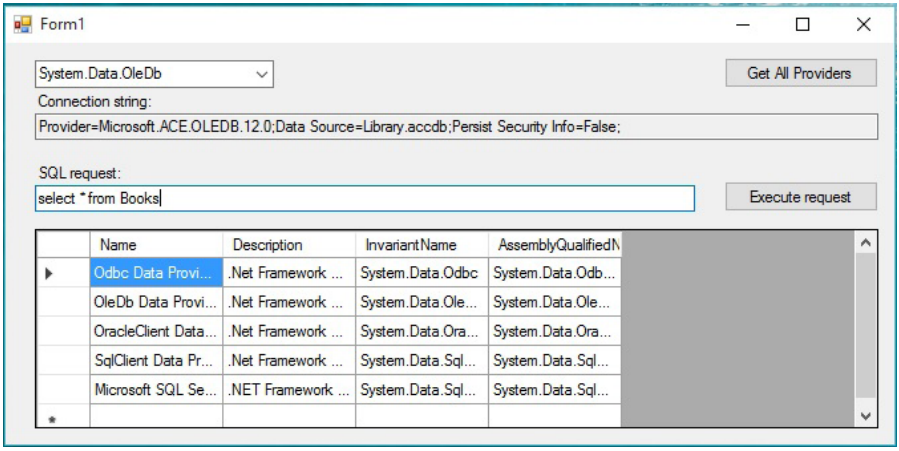

Рис. 6. Доступ к таблице MS Access 2007

После выполнения запроса “select * from books” для этой таблицы я получил такой результат:

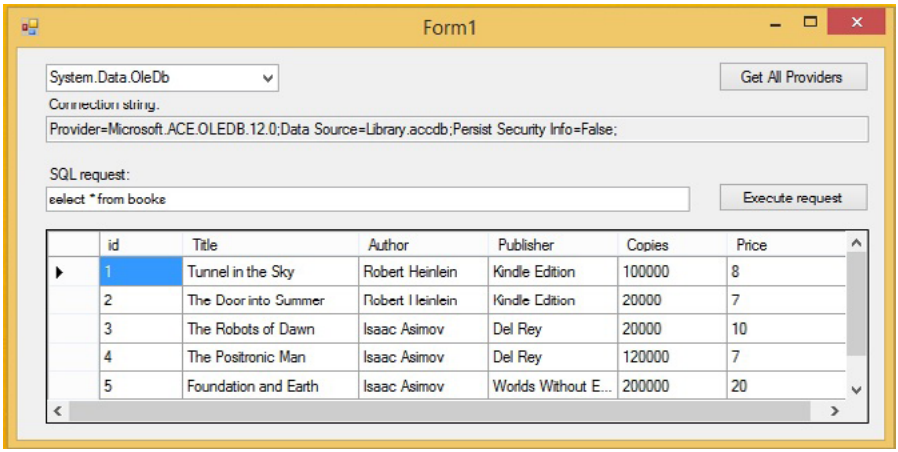

Рис. 7. Данные из таблицы MS Access 2007

Данное приложение демонстрирует, что один и тот же код может работать с разными поставщиками данных. Вы понимаете, что это возможно благодаря классам DbProviderFactory и DbProviderFactories. Хотите попробовать еще какого-нибудь поставщика? Добавьте строку подключения к соответствующей БД в наш конфигурационный файл и убедитесь, что требуемый поставщик зарегистрирован на вашем компьютере. В коде приложения изменять ничего не надо.

И еще один момент. Для разминки. Посмотрите внимательно на данные из таблицы в MS Access 2007, приведенные на последнем рисунке. Подумайте, почему такая таблица не должна использоваться в реальной БД?

Ответ, конечно же, заключается в том, что приведенная таблица не нормализирована. Она не приведена даже к 1НФ.

Дополнительные возможности DbDataAdapter

Управление данными в DataSet

Предположим, что вы выполнили какой-либо запрос select и получили результат выполнения этого запроса в DataSet. Затем вы отключились от источника данных и продолжаете работать с полученными данными локально. Теперь предположим, что вам потребовалось каким-либо образом отсортировать или фильтровать данные в своем DataSet. Как это можно сделать? Конечно, можно выполнить новый запрос, в котором задать требуемые условия сортировки и отбора. А можно ли сделать это без повторного обращения к БД? Ведь требуемые данные у нас уже есть в DataSet и мы только хотим отобразить их в другом виде?

Любую фильтрацию и сортировку локальных данных в DataSet можно выполнить с помощью класса DataViewManager. Управлять фильтрацией и сортировкой данных объект DataViewManager позволяет с помощью своих свойств RowFilter и Sort. Рассмотрим, каким образом это можно сделать.

```
//создаем адаптер с запросом к таблице Authors
SqlDataAdapter adapter = new SqlDataAdapter("SELECT *
FROM Authors", conn);

//выполняем запрос и заносим результаты в DataSet
adapter.Fill(set, "Authors");
```

```
//создаем объект DataViewManager для DataSet
//с нашими результатами
DataViewManager dvm = new DataViewManager(set);

//задаем условия отбора и сортировки для требуемой
//таблицы в DataSet
dvm.DataViewSettings["Authors"].RowFilter = "id < 100";
dvm.DataViewSettings["Authors"].Sort = "LastName ASC";

//создаем объект содержащий отобранные
//и сортированные данные
//и связываем этот объект с dataGridView1
DataView dataView1 =
      dvm.CreateDataView(set.Tables["Authors"]);

dataGridView1.DataSource = dataView1;
```

Если вы вставите этот фрагмент кода в наше третье приложение, то получите на экране такой результат:

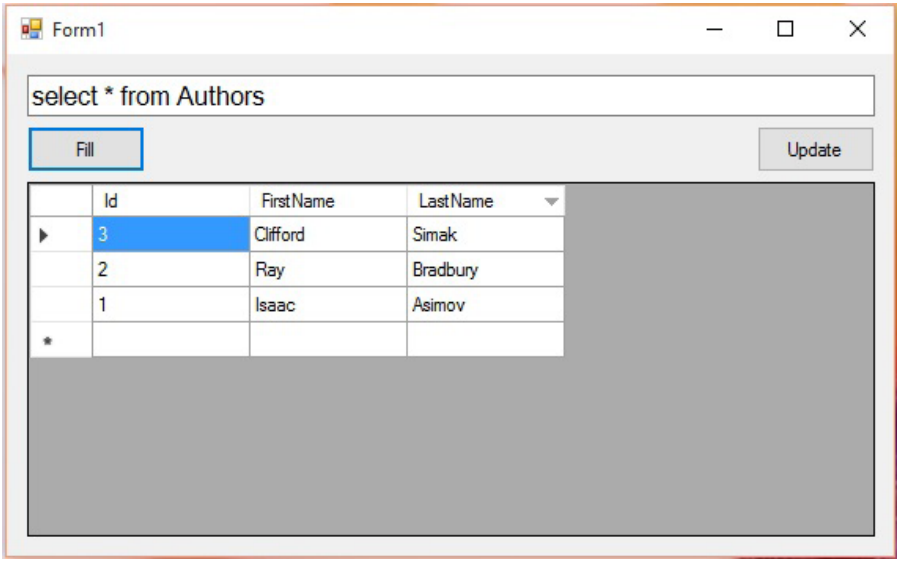

Рис. 8. Применение фильтрации и сортировки

Понятно, что вставлять управление фильтрацией и сортировкой жестко в код — не лучшее решение. Правильнее было бы создать графический интерфейс, позволяющий управлять отбором данных. Но для нашего демонстрационного примера создавать такой интерфейс не целесообразно.

Теперь вы знаете, как добавить в ваше приложение управление отображением данных в DataSet, используя класс DataViewManager. Эффективность этого подхода объясняется тем, что вам не надо выполнять новый запрос всякий раз, когда вы хотите увидеть данные в новом виде. Запрос выполнен один раз, а затем разные выборки выполняются из локальных данных.

Поддержка транзакций

Мы уже говорили с вами о том, что одной из важнейших задач любой СУБД является сохранение целостности данных в БД. Одним из основных инструментов для этого являются транзакции. Давйте вспомним, что такое транзакция и как она работает. Транзакция — это механизм, предназначенный соблюдать целостность данных в БД при выполнении каких-либо изменений этих данных. Транзакцию можно представлять, как набор каких-либо действий с данными, оформленных как единое целое, как пакет команд. Если все единичные операции, входящие в состав транзакции успешно выполняются, БД переходит в новое, измененное, но целостное состояние. Если хоть одно из действий, входящих в состав транзакции, по какой-либо причине не может быть выполнено, отменяются все, выполненные до этого

действия из текущей транзакции. В этом случае говорят, что транзакция откатывается и БД возвращается к исходному целостному состоянию, в котором она находилась до начала выполнения транзакции. Формально в терминах SQL транзакция начинается инструкцией Begin (начало транзакции), а завершается либо инструкцией Rollback (откат транзакции), либо инструкцией Commit (выполнение транзакции). Рассмотрим пример транзакции по переводу денег в размере @Value со счета с id равным значению @FromId на счет с id равным значению @ToId.

```
-- Начало транзакции
BEGIN TRANSACTION
-- добавляем деньги на счет получателя
UPDATE Account
        SET Balance = Balance + @Value
        WHERE Id = @ToId

    -- проверяем, добавились ли деньги на счет
    -- получателя
    IF (@@error <> 0)
        -- Отменить транзакцию, если есть ошибки
        ROLLBACK TRANSACTION -- если ошибка —
        -- выполняем откат транзакции


-- снимаем деньги со счета отправителя
UPDATE Account
        SET Balance = Balance - @Value
        WHERE Id = @FromId

    -- проверяем, снялись ли деньги со счета
    -- отправителя
    IF (@@error <> 0)
        ROLLBACK TRANSACTION -- если ошибка —
        -- выполняем откат транзакции
-- Успешное завершение транзакции
COMMIT TRANSACTION
```

Конечно же, этот пример условный, но он демонстрирует логику применения транзакции.

ADO.NET должен предоставлять нам возможность работать с транзакциями. Давайте рассмотрим, какие классы применяются для создания транзакций и как с ними надо работать. Для поддержки транзакций используется класс Transaction, вместе с ним в этом случае применяются уже известные нам классы Connection и Command.

Алгоритм создания транзакции выглядит так.

- Вызывается метод Connection.BeginTransaction() для создания транзакции;
- В свойство Command.Transaction заносится созданная транзакция;
- Формируется список запросов или вызовов хранимых процедур, из которых должна состоять транзакция;
- С помощью методов Transaction.Commit() или Transaction.Rollback() оформляется завершения транзакции;

Давайте теперь после краткого теоретического обзора добавим транзакцию в наше приложение из второго урока, которому я присвоил имя LibraryTest3. Оформим транзакцию в виде отдельного метода и повесим ее на собственную кнопку. Добавленный метод может выглядеть таким образом:

```
private void btntran_Click(object sender, EventArgs e)
{
    conn = new SqlConnection(cs);
    SqlCommand comm = conn.CreateCommand();
```

```
    SqlTransaction tran = null;
    try
    {
        conn.Open();
        tran = conn.BeginTransaction();
        comm = conn.CreateCommand();
        comm.Transaction = tran;
        comm.CommandText = @"create table tmp3(
            id int not null identity(1,1) primary key,
            f1 varchar(20), f2 int)";
        comm.ExecuteNonQuery();
        comm.CommandText = @"insert into tmp3(f1, f2)
            values(/Text value 1/, 555)";
        comm.ExecuteNonQuery();
        comm.CommandText = @"insert into tmp4(f1, f2)
            values(/Text value for second row/, 777)";
         comm.ExecuteNonQuery();

         tran.Commit();
    }

    catch (Exception ex)
    {
         MessageBox.Show(ex.Message);
         tran.Rollback();
    }
    finally
    {
         conn.Close();
    }
}
```

Разберем приведенный код. Нас интересуют строки, выделенные желтым цветом. Создаем ссылку типа SqlTransaction и инициализируем ее вызовом метода Connection.BeginTransaction(). Для созданного объекта

SqlCommand инициализируем свойство Transaction только что созданной транзакцией. Теперь последовательно заносим в объект SqlCommand требуемые запросы и выполняем их вызовом метода ExecuteNonQuery(), поскольку эти запросы ничего не возвращают. Обратите внимание, что в данном случае результаты выполнения каждого запроса НЕ применяются сразу к БД и НЕ изменяют состояния БД! Результаты выполнения каждого запроса, входящего в транзакцию, сохраняются в оперативной памяти до момента либо вызова метода Commit(), и тогда они занесутся в БД, либо вызова метода Rollback(), и тогда они будут проигнорированы.

Я намеренно допустил ошибку в третьем запросе, входящем в состав транзакции, указав там неверное имя таблицы (tmp4). Если вы сейчас нажмете кнопку и выполните приведенную транзакцию, вы не увидите в БД таблицы с именем tmp3. Несмотря на то, что запросы по созданию таблицы и по занесению в нее первой записи написаны правильно, они не будут выполнены, так как входят в состав транзакции с одной неправильной инструкцией. Одна неправильная инструкция отменяет результат выполнения двух предыдущих правильных инструкций. Так и должна вести себя транзакция.

Исправьте в третьем запросе имя таблицы на tmp3 и выполните транзакцию снова. В этот раз вы не увидите сообщения об исключительной ситуации. Если вы сейчас занесете в окно нашего приложения запрос ”select * from tmp3” и нажмете кнопку Fill, вы увидите данные из новой таблицы в нашем dataGridView1. Транзакция работает.

Удалите созданную таблицу tmp3 из БД, т.к. эта таблица нам не понадобится. Хотелось бы еще пару слов сказать о том, какие именно действия надо выполнять в виде явной транзакции. Как правило, это диктуется логикой приложения. Классический пример — перевод денег с одного банковского счета на другой. Ваше приложение должно на заданную величину уменьшить баланс отправителя и на эту же величину увеличить баланс получателя. Вы снимаете деньги со счета отправителя, а приложение по какой-либо причине завершает работу, не добавив деньги на счет получателя. Целостность данных нарушена. А если бы снятие денег с одного счета и их добавление на другой были оформлены, как транзакция, нарушения целостности не произошло бы.

У вас может возникнуть вопрос, если транзакция настолько полезна в обеспечении целостности данных, то надо ли все действия в БД оформлять в виде транзакций? Чтобы ответить на этот вопрос точно, надо придерживаться специальной терминологии. Вы должны различать такие понятия, как «явная транзакция» и «неявная транзакция» (или просто «транзакция»).

Явная транзакция — это набор инструкций SQL, начинающийся инструкцией BEGIN TRANSACTION и завершающийся либо инструкцией ROLLBACK TRANSACTION либо инструкцией COMMIT TRANSACTION. Если какое-либо действие, входящее в состав явной транзакции, завершается ошибкой — выполняется ROLLBACK TRANSACTION и транзакция откатывает все изменения в данных до того состояния, какое было до начала выполнения транзакции.

Если все действия успешно завершены — выполняется COMMIT TRANSACTION и данные переходят в новое целостное состояние.

Неявная транзакция не включает в свой состав инструкции BEGIN TRANSACTION или ROLLBACK TRANSACTION или COMMIT TRANSACTION, но при этом выполняется тоже, как транзакция. Например, DDL запросы языка SQL (select, insert, update и delete) выполняются как неявные транзакции. Другими словами, если вы выполняете запрос insert, который должен заполнить в какой-либо таблице двадцать полей, то этот запрос не сможет завершиться, заполнением только 12 или 15 полей. Он либо заполнит все 20 полей, либо не заполнит ни одного поля, в случае какой-либо ошибки. О таком выполнении этих запросов побеспокоились разработчики соответствующих поставщиков данных.

Теперь вернемся к нашему вопросу о том, надо ли все действия в БД оформлять в виде транзакций. Правильно этот вопрос должен быть сформулирован так: «надо ли все действия в БД оформлять в виде явных транзакций?». Ответ на такой вопрос будет «нет». Дело в том, что в виде явной транзакции надо оформлять только действия, логически связанные друг с другом, например увязывающие в одно целое обработку нескольких таблиц, как в примере с переводом денег. Явная транзакция — вещь достаточно ресурсоемкая и пользоваться ею надо «без фанатизма». О транзакционном выполнении других действий в БД позаботились сами разработчики поставщиков данных.

Асинхронная работа с БД

Пришло время поговорить об асинхронности при работе с поставщиками данных. Вы прекрасно понимаете, что некоторые действия с БД было бы полезно выполнять асинхронно. Что такое асинхронность? Это отсутствие «синхронности» или же отсутствие блокировок. Наверное, вы все сталкивались с ситуацией, когда какая-либо программа начинает выполнять какое-то действие и вдруг окно этой программы становится неактивным, в заголовке может появиться сообщение «не отвечает», а любые пользовательские действия с программой становятся невозможными? Через какое-то время все восстанавливается, и программа снова ведет себя «как надо». Это типичный пример блокировки. Такая блокировка возникает потому, что активированное действие заставляет всех ожидать, пока оно (это действие) не завершится. А если такое действие выполняется в первичном потоке приложения, то это очень плохо. Дело в том, что первичный поток приложения предназначен для обслуживания пользовательских действий с элементами управления окна программы. И если первичный поток занять трудоемкой операцией, то обслуживать работу с окном программы будет некому. Чтобы не возникали такие ситуации, трудоемкие действия надо выполнять асинхронно, т.е. так, чтобы они не вызывали блокировки. Один из способов обеспечения асинхронности — выполнение блокирующего действия в дополнительном потоке.

Многие считают, что асинхронность — это и есть работа в дополнительном потоке. Это не так. Асинхронность — это отсутствие блокировок. А отсутствие блокировок можно достичь и без использования дополнительных потоков. Вы научитесь делать это при рассмотрении спецификаторов async и await.

Сейчас мы рассмотрим реализацию асинхронности с применением дополнительных потоков. Вы должны понимать, что дополнительные потоки — штука очень ресурсоемкая. Поэтому использовать дополнительные потоки надо аккуратно. Давайте, прежде всего познакомимся с теми способами, которые ADO.NET предоставляет нам для работы с БД в дополнительных потоках. А затем, еще раз поговорим о том, в каких случаях следует использовать асинхронность.

На сегодняшний день у нас есть два способа писать асинхронный код, использующий дополнительные потоки для выполнения в них блокирующих действий: использовать классический паттерн BeginXXX() и EndXXX() или использовать новые средства async и await.

Классический поход к асинхронности

В какой момент начинается работа приложения с БД и пересылка данных между приложением и сервером БД? Это происходит в момент выполнения запросов к БД. В момент, когда приложение вызывает методы ExecuteNonQuery(), ExecuteReader() или ExecuteScalar(). Значит именно для этих методов надо обеспечить асинхронность. Обычно асинхронное выполнение какого-либо действия обеспечивается двумя

методами. Один метод, условно называемый BeginAction(), начинает выполнение требуемого действия в дополнительном потоке. Другой метод, условно называемый EndAction(), должен быть вызван, когда действие в дополнительном потоке будет завершено. Вы должны понимать, что в методах BeginAction() и EndAction() слово Action означает название какого-либо конкретного действия. Например, Read или Write. BeginAction() и EndAction() — это обобщенные названия для асинхронных методов, реально методов с такими именами нет. А есть, например, методы BeginRaed() и EndRead(), BeginWrite() и EndWrite(), и те асинхронные методы для работы с БД, с которыми мы сейчас познакомимся. Продолжим рассмотрение схемы реализации асинхронности.

В этой схеме надо понимать два момента:

- Если действие, выполняемое в дополнительном потоке в методе BeginAction(), должно вернуть какой-либо результат, то этот результат можно получить, только после вызова метода EndAction();
- Если вызвать метод EndAction() до того, как действие в дополнительном потоке завершится, то этот вызов EndAction() станет блокирующим и заблокирует, вызвавший его поток;

Таким образом, задача сводится к тому, как можно узнать, когда действие в дополнительном потоке завершилось. Чтобы для получения результата вызвать метод EndAction() и не блокировать приложение. Для этого в нашем распоряжении есть три способа:

- Использование callback делегатов;
- Использование класса WaitHandle;
- Опрос дополнительного потока;

Рассмотрим все эти способы. Какими инструментами мы располагаем, чтобы создать средства асинхронного доступа к БД?

Это асинхронные варианты командных методов: BeginExecuteNonQuery() и EndExecuteNonQuery(), BeginExecuteReader() и EndExecuteReader(). Для метода ExecuteScalar() асинхронного варианта нет.

Кроме этих методов, важную роль в реализации асинхронности играет интерфейс IAsyncResult. Что надо знать об этом интерфейсе?

- Методы BeginExecuteNonQuery() и BeginExecuteReader() имеют тип возвращаемого значения IAsyncResult.
- Callback методы, используемые в асинхронном механизме, имеют единственный параметр типа IAsyncResult.
- В свойстве IAsyncResult.AsyncWaitHandle находится объект WaitHandle, позволяющий управлять вызовом метода EndAction().
- Через свойство IAsyncResult.AsyncState можно предавать данные между дополнительным и основным потоками.

Обо всех этих характеристиках типа IAsyncResult мы подробнее поговорим, создавая код приложения. Рассмотрим все три способа выполнения асинхронных действий.

Использование callback методов

Давайте создадим вариант асинхронного обращения к БД в виде отдельного метода и снова используем его в приложении LibraryTest3 из второго урока. Добавьте в указанный проект еще одну кнопку, подпишите ее, например, Async Callback, и создайте для этой кнопки обработчик. Вставьте в созданный обработчик такой код:

```
private void btAsync_Click(object sender, EventArgs e)
{
    /// блок 1
    const string AsyncEnabled =
              "Asynchronous Processing=true";
    if (!cs.Contains(AsyncEnabled))
    {
        cs = String.Format("{0}; {1}", cs, AsyncEnabled);
    }
    ///

    conn = new SqlConnection(cs);
    SqlCommand comm = conn.CreateCommand();

    /// блок 2
    comm.CommandText = "WAITFOR DELAY /00:00:05/;
         SELECT * FROM Books;";
    comm.CommandType = CommandType.Text;
    comm.CommandTimeout = 30;
    ///
    try
    {
        conn.Open();
        /// блок 3
        AsyncCallback callback =
                  new AsyncCallback(GetDataCallback);
```

```
            comm.BeginExecuteReader(callback, comm);
            MessageBox.Show("Added thread is working...");
            ///
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

В этом простом обработчике выделено три блока кода, о которых надо рассказать подробнее.

В первом блоке происходит модификация строки подключения. Дело в том, что при использовании асинхронного доступа к источнику данных, строка подключения должна содержать атрибут «Asynchronous Processing=true». Поэтому в этом блоке проверяется наличие этого атрибута, и если его нет в строке подключения, он добавляется в прочитанную копию строки. Сама строка подключения при этом не изменяется, т.к. мы не планируем использовать асинхронность все время.

Во втором блоке мы имитируем выполнение длительной операции на сервере БД. Для этого используется SQL запрос “WAITFOR DELAY ‘00:00:05’”. Этот запрос вызывает остановку сервера БД на 5 секунд, чтобы мы имели время убедиться, что окно нашего приложения не блокируется на время выполнения запроса к БД. Для этой же цели мы выводим дальше диалоговое окно с сообщением «Added thread is working...», чтобы показать, что после запуска работы в дополнительном потоке приложение продолжает

работу в первичном потоке. Здесь надо обратить внимание на инициализацию свойства comm.CommandTimeout. По умолчанию значение этого свойства равно 30 секунд. Это время, выделяемое приложению на подключение к серверу БД и на выполнение запроса. Если вы полагаете, что вашему приложению может потребоваться больше времени, вы должны изменить значение этого свойства. В нашем случае, мы оставляем значение по умолчанию, просто, чтобы продемонстрировать необходимость помнить об этом свойстве.

И, наконец, в третьем блоке выполняется очень важная часть работы. Здесь создается делегат типа AsyncCallback, в который заносится адрес callback метода с именем GetDataCallback(). Имя этого метода произвольно, а вот сигнатура должна быть такой: тип возвращаемого знасения — void и один параметр типа IAsyncResult. Давайте посмотрим, как используется этот метод при запуске асинхронной операции, а потом приведем его код и разберем подробно, как он работает. Работа в дополнительном потоке начинается при вызове метода

```
comm.BeginExecuteReader(callback, comm);
```

Обратите внимание, что мы передаем созданный делегат с адресом callback метода в дополнительный поток. Этот делегат сделает так, что наш callback метод GetDataCallback() будет автоматически вызван системой, когда работа в дополнительном потоке будет завершена. Кроме этого, мы передаем в дополнительный поток наш объект comm. Давайте сейчас разберем код метода

GetDataCallback(), который также надо добавить в наш проект рядом с созданным обработчиком.

```
private void GetDataCallback(IAsyncResult result)
{
    SqlDataReader reader = null;
    try
    {
        /// блок 1
        SqlCommand command = (SqlCommand)result.AsyncState;
        ///
        /// блок 2
        reader = command.EndExecuteReader(result);
        ///
        table = new DataTable();

        int line = 0;

        do
        {
            while (reader.Read())
            {
                if (line == 0)
                {
                    for (int i = 0; i <
                              reader.FieldCount; i++)
                   {
                    table.Columns.Add(reader.GetName(i));
                   }
                   line++;
                }
                DataRow row = table.NewRow();
                for (int i = 0; i < reader.FieldCount; i++)
                {
                    row[i] = reader[i];
                }
```

```
                table.Rows.Add(row);
               }
          } while (reader.NextResult());
          DgvAction();
     }

     catch (Exception ex)
     {
         MessageBox.Show("From Callback 1:"+ex.Message);
     }
     finally
     {
         try
         {
             if (!reader.IsClosed)
             {
                reader.Close();
             }
         }
         catch (Exception ex)
         {
            MessageBox.Show("From Callback 2:" +
                            ex.Message);
         }
     }
}
```

Поскольку действие в этом методе выполняется в дополнительном потоке, мы не можем обращаться к DataGridView из этого метода напрямую. Поэтому мы создали метод DgvAction() из которого обращаемся к DataGridView, чтобы отобразить в нем результаты запроса. Для этого нам надо сделать объект table глобальным, чтобы доступ к нему был и из GetDataCallback() и из DgvAction().

```
private void DgvAction()
{
    if (dataGridView1.InvokeRequired)
    {
        dataGridView1.Invoke(new Action(DgvAction));
        return;
    }
    dataGridView1.DataSource = table;
}
```

Помните, при вызове BeginExecuteReader(), во втором параметре мы передали в дополнительный поток объект comm. Запомните, все, что вы передаете в callback метод, оказывается в свойстве AsyncState параметра этого callback метода. Свойство AsyncState имеет тип Object, поэтому мы можем передавать в дополнительный поток любые типы. В первом блоке выделенного кода мы просто забираем из входного параметра переданный объект comm. Вы спросите, а зачем нам нужен этот объект в дополнительном потоке? Дело в том, что метод BeginExecuteReader() мы вызывали от этого объекта, и завершить начатое действие надо вызовом метода EndExecuteReader() от этого же объекта. Что мы и делаем во втором блоке. Еще раз отметьте, что получить результат выполнения асинхронного действия можно ТОЛЬКО после вызова метода EndAction(). В нашем случае — метода EndExecuteReader(). В нашем примере результатом выполнения действия является заполненный данными объект reader, вот его мы и получаем после вызова EndExecuteReader() и дальше с ним работаем.

Дальше идет стандартная и уже знакомая вам обработка SqlDataReader, извлечение данных в DataTable

и отображение их в dataGridView1. Отметьте, что в командном свойстве объекта comm находятся сразу два запроса, и оба они нормально обрабатываются. При запуске приложения я получил такой результат. На приведенном рисунке я уже закрыл диалоговое окно с сообщением «Added thread is working...».

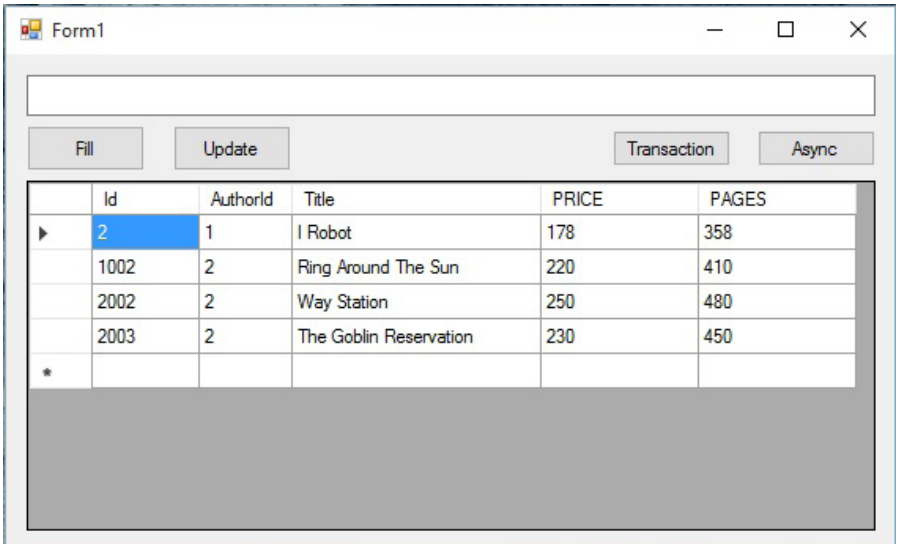

Рис. 9. Асинхронный доступ к БД

Теперь акцентируем внимание на том, каким образом был выполнен наш асинхронный запрос к БД. После вызова метода BeginExecuteReader() автоматически был создан новый поток, и в нем наше приложение подключилось к серверу БД и начало выполнять на сервере переданные запросы. В это время, основной поток нашего приложения продолжал свою работу — в частности, сформировал и вывел диалоговое окно с сообщением. В какой-то момент времени работа

в дополнительном потоке завершилась. И вот здесь свою роль сыграл наш callback метод GetDataCallback(). Эта роль заключается в том, что этот метод был вызван системой автоматически (на то он и callback метод), как только работа в дополнительном потоке завершилась. Все остальное, что происходит в методе GetDataCallback(), мы уже обсудили. Нам не пришлось явно определять завершение работы дополнительного потока. За нас это сделал callback метод GetDataCallback(). И еще обратите внимание на то, что мы вызывали метод BeginExecuteReader() не получая от него его возвращаемое значение. При использовании callback методов можно обходиться без этого значения.

Использование класса WaitHandle

Сейчас мы рассмотрим другой механизм асинхронного доступа к БД. Мы будем использовать объект класса синхронизации WaitHandle, чтобы узнать, завершена ли работа в дополнительном потоке и вызвать метод EndAction(). Снова добавим новую кнопку в окне приложения LibraryTest3 и создадим обработчик для этой кнопки:

```
private void btAsync2_Click_1(object sender, EventArgs e)
{
    const string AsyncEnabled =
                    "Asynchronous Processing=true";
    if (!cs.Contains(AsyncEnabled))
    {
        cs = String.Format("{0}; {1}", cs, AsyncEnabled);
    }
```

```
        conn = new SqlConnection(cs);
        SqlCommand comm = conn.CreateCommand();
        comm.CommandText = "WAITFOR DELAY /00:00:05/;
                            SELECT * FROM Books;";
        comm.CommandType = CommandType.Text;
        comm.CommandTimeout = 30;
        try
        {
            conn.Open();
            /// блок 1
            IAsyncResult iar = comm.BeginExecuteReader();
            ///

            /// блок 2
            WaitHandle handle = iar.AsyncWaitHandle;
            ///
            /// блок 3
            if(handle.WaitOne(10000))
            {
                /// блок 4
                GetData(comm, iar);
                ///
            }
            else
            {
                MessageBox.Show("TimeOut exceeded");
            }
            ///
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
```

Разберем приведенный код. В нем выделены только те строки, которые отличаются от предыдущего варианта.

В первом блоке вызывается метод BeginExecuteReader(). Теперь при вызове мы получаем возвращаемое значение этого метода и не передаем ему никаких параметров. Как вы видите, никаких делегатов и callback методов тоже нет.

Во втором блоке мы берем из возвращенного методом BeginExecuteReader() значения объект типа WaitHandle, с помощью которого мы узнаем о завершении работы в дополнительном потоке. Пусть вас не смущает то, что мы получаем возвращаемое значение от метода BeginExecuteReader(), который все еще продолжает работу в дополнительном потоке. Дело в том, что этот метод не блокирующий. Он стартует работу в дополнительном потоке и тут же возвращает управление, вызвавшему его потоку. В возвращенном значении, в свойстве AsyncWaitHandle как раз и располагается объект WaitHandle, который доступен из дополнительного потока и который просигнализирует нам о том, что работа в дополнительном потоке завершилась. В третьем блоке мы проверяем, завершилась ли работа ли работа дополнительного потока. Когда дополнительный поток завершиться, метод WaitOne() вернет true и наш код вызовет обычный (не callback) метод GetData() для получения результата. Обратите внимание, что в методе WaitOne() задан таймаут, чтобы приложение не зависло в случае какой-либо ошибки в дополнительном потоке.

Вы уже понимаете, для чего в этот метод GetData() передаются объекты comm и iar. Ниже приведен код метода GetData(), который очень похож на callback метод из предыдущего раздела.

```
private void GetData(SqlCommand command, IAsyncResult ia)
{
    SqlDataReader reader = null;
    try
    {
        reader = command.EndExecuteReader(ia);
        DataTable table = new DataTable();
        dataGridView1.DataSource = null;

        int line = 0;

        do
        {
        while (reader.Read())
        {
            if (line == 0)
            {
               for (int i = 0; i < reader.FieldCount;
                               i++)
               {
                  table.Columns.Add(reader.GetName(i));
               }
               line++;
            }
            DataRow row = table.NewRow();
            for (int i = 0; i < reader.FieldCount; i++)
            {
               row[i] = reader[i];
            }
            table.Rows.Add(row);
        }
        } while (reader.NextResult());
        dataGridView1.DataSource = table;
    }
    catch (Exception ex)
    {
          MessageBox.Show("From GetData:" + ex.Message);
```

```
        }
        finally
        {
            try
            {
                if (!reader.IsClosed)
                {
                    reader.Close();
                }
            }
            catch
            {
            }
        }
    }
```

Запустите приложение и убедитесь, что оно работает. Что надо отметить об этом способе? Вы понимаете, что вызов WaitOne() блокирующий и наше приложение ждет завершения основного потока. Казалось бы, что это недостаток. Но в некоторых случаях приложению бывает просто нечего делать до завершения дополнительного потока. Кроме этого, главное преимущество использования объекта WaitHandle проявляется при необходимости следить за завершением нескольких дополнительных потоков. У этого класса, кроме метода WaitOne(), есть еще статические методы WaitAll() и WaitAny(). Эти методы принимают массивы объектов WaitHandle, соответствующих запущенным дополнительным потокам, и умеют сигнализировать о завершении всех отслеживаемых потоков, или о завершении одного из отслеживаемых потоков. Чтобы научиться использовать эти методы вам будет предложено соответствующее домашнее задание.

Использование опроса дополнительного потока

В двух предыдущих примерах вы увидели, каким образом основной поток может узнавать о завершении работы дополнительного потока, чтобы вызвать метод EndAction() и получить результат. Существует еще один способ сделать это. Этот способ заключается в опросе свойства IsCompleted, объекта возвращенного методом BeginExecuteReader():

```
IAsyncResult iar = comm.BeginExecuteReader();
while (!iar.IsCompleted)
{
    Console.WriteLine("Waiting...");
}
```

Использование этого способа — это «холостой ход» в работе приложения, или же бесцельная трата процессорного времени. В реальных приложениях использовать такой подход, вряд ли стоит. Хотя, возможно, иногда может понадобиться и такой прием.

Домашнее задание

В этом домашнем задании вам надо создать приложение, которое будет подключаться к БД и читать данные из двух разных таблиц одновременно в двух разных потоках. После того, как оба потока завершат работу, они должны будут вывести некие итоговые результаты своей работы. Поскольку БД у нас небольшая и работа потоков будет выполняться очень быстро, желательно добавить в каждый запрос какую-нибудь задержку в несколько секунд.

Обратите внимание — результаты должны выводиться, когда оба потока завершат работу. Если какой-либо поток завершит работу раньше, он должен будет подождать до завершения работы второго потока. Реализация этого момента — одна из самых интересных задач в этом задании. Другая задача — придумать, каким способом создавать дополнительные потоки. Выберите и реализуйте решения этих задач самостоятельно.

Итак. Создайте Windows Forms приложение с кнопкой «Start» и двумя небольшими текстовыми полями, в которые будут выводиться результаты работы потоков. Текстовые поля можно подписать, например «Поток 1» и «Поток 2». Кнопка «Start» должна запускать создание двух дополнительных потоков. В одном потоке должен выполняться запрос «select * from Books», во втором — «select * from Authors». Первый поток должен вычислить суммарное для всех строк количество

символов в поле title в таблице Books. Второй поток должен вычислить суммарное количество символов в полях FirsName и LastName в таблице Authors. Эти вычисленные количества символов и должны быть выведены в соответствующие текстовые поля.