21

Транзакции

Все или ничего — в этом главный смысл транзакции. При сохранении нескольких записей либо все должны быть записаны, либо вся операция должна быть отменена. Если происходит один-единственный сбой при внесении одной записи, то все, что было выполнено к этому моменту в пределах данной транзакции, откатывается.

Транзакции широко используются при работе с базами данных, но классы из пространства имен `System.Transaction` позволят вам выполнять транзакции с изменчивыми или находящимися в памяти объектами, такими как списки объектов. Если список поддерживает транзакции, объект добавляется или удаляется и транзакция завершается неудачей, то все операции со списком автоматически отменяются. Запись в списки, находящиеся в памяти может выполняться в той же транзакции, что и запись в базу данных.

В Windows Vista файловая система и реестр также поддерживают транзакции, то есть запись в файл и внесение изменений в реестр порождают транзакции.

В настоящей главе мы раскроем следующие темы, имеющие отношение к транзакциям:

- обзор;
- традиционные транзакции;
- фиксируемые транзакции;
- распространение транзакции;
- зависимые транзакции;
- внешние транзакции;
- уровень изоляции транзакций;
- настраиваемые диспетчеры ресурсов;
- транзакции в Windows Vista.

Обзор

Что такое транзакция? Представьте себе систему заказа книг на Web-сайте. Процесс заказа книг удаляет выбранную вами книгу со склада и помещает в вашу корзину покупок, а стоимость книги списывается с вашей кредитной карты. Оба эти действия должны либо завершиться успешно, либо не должно произойти ни одного из них.

Если случается сбой при получении книги со склада, оплата не должна быть списана с кредитной карты. Такой сценарий можно реализовать с помощью транзакций.

Наиболее распространенное применение транзакций — при внесении и обновлении информации в базе данных. Транзакции также могут выполняться при записи сообщения в очередь сообщений или при записи данных в файл или системный реестр. Несколько действий могут быть частями одной транзакции.

`System.Messaging` обсуждается в главе 39.

На рис. 21.1 показаны главные действующие лица транзакции. Транзакции управляются и координируются диспетчером транзакций. Любой ресурс, влияющий на исход транзакции, управляется диспетчером ресурсов. Диспетчер транзакций взаимодействует с диспетчерами ресурсов, определяя исход транзакции.

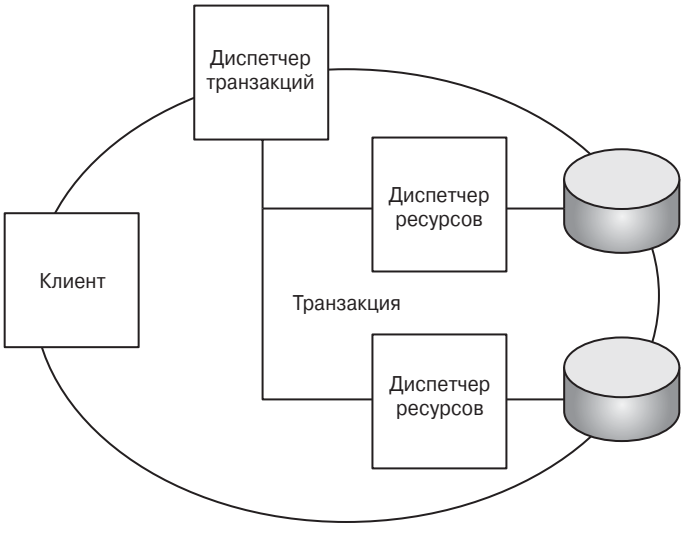

Рис. 21.1. Главные действующие лица транзакции

Фазы транзакции

Временные фазы транзакции таковы: *активная*, *подготовительная* и *фиксации*.

- **Активная фаза**. Во время активной фазы транзакция создается. Диспетчеры ресурсов, управляющие транзакцией для ресурсов, принимают участие в транзакции.
- **Подготовительная фаза**. Во время подготовительной фазы каждый диспетчер ресурсов может определять исход транзакции. Эта фаза стартует, когда инициатор транзакции посылает подтверждение для того, чтобы завершить транзакцию. Диспетчер транзакции посылает сообщение *Prepare* (подготовиться) всем диспетчерам ресурсов. Если диспетчер ресурсов может выполнить свою работу для успешного исхода транзакции, он посылает сообщение *Prepared* (готов) диспетчеру транзакций. Диспетчер ресурсов может прервать транзакцию, если он не может подготовиться к своей работе, заставив диспетчер транзакций выполнить откат. После того, как сообщение Prepared отправлено, диспетчер ресурсов должен гарантировать успешное завершение работы на фазе фиксации.

Чтобы это было возможно, устойчивые диспетчеры ресурсов должны писать журнал, внося в него информацию о состоянии готовности, чтобы иметь возможность продолжить работу с этого места, например, в случае отключения питания между фазами подготовки и фиксации.

- **Фаза фиксации**. Эта фаза начинается, когда все диспетчеры ресурсов успешно подготовились к работе, а именно — когда получено сообщение *Prepared* от всех участвующих диспетчеров ресурсов. Затем диспетчер транзакции может завершить работу транзакции и вернуть сообщение *Commited*.

Свойства ACID

Транзакция подчиняется некоторым специфическим требованиям; например, в результате работы транзакции должно быть получено корректное состояние системы. Корректное состояние также является необходимым условием в случае сбоя питания на сервере. Характеристики транзакции могут быть определены термином ACID, смысл которого мы поясним в следующем разделе.

ACID — это аббревиатура четырех слов, означающих *атомарность* (atomicity), *согласованность* (consistency), *изоляцию* (isolation) и *устойчивость* (durability).

- **Атомарность**. Атомарность представляет единицу работы. В отношении транзакции это означает, что либо вся единица работы будет успешно выполнена, либо ничего не будет изменено.
- **Согласованность**. Состояния перед стартом транзакции и после ее завершения должны быть корректными. Во время транзакции состояние может иметь промежуточные значения.
- **Изоляция**. Изоляция означает, что транзакции, выполняющиеся одновременно, изолированы от состояния, которое изменяется во время транзакции. Транзакция A не может видеть промежуточного состояния транзакции B до тех пор, пока она не будет завершена.
- **Устойчивость**. После завершения транзакции ее результат должен быть зафиксирован на постоянной основе. Это значит, что если произойдет сбой сети или сервера, то состояние должно быть восстановлено после перезапуска.

Базы данных и классы

Пример базы данных `CourseManagement`, которую мы будем использовать с транзакциями в настоящей главе, имеет структуру, показанную на рис. 21.2. Таблица `Courses` содержит информацию о курсах: номера и заголовки курсов. Так, например, курс номер 2124 озаглавлен “Programming C#”. Таблица `CourseDates` содержит даты проведения определенных курсов и связана с таблицей `Courses`. Таблица `Students` хранит информацию о лицах, назначенных на курс. Таблица `CourseAttendees` представляет связи между `Students` и `CourseDates`. Она определяет, какие студенты на какой курс назначены.

База данных вместе с исходным кодом для упражнений этой главы находится на прилагаемом компакт-диске.

В примере приложении настоящей главы используется библиотека сущностных классов и классов доступа к данным. Класс `Student` содержит свойства, определяющие студента, среди которых `Firstname`, `Lastname` и `Company`.

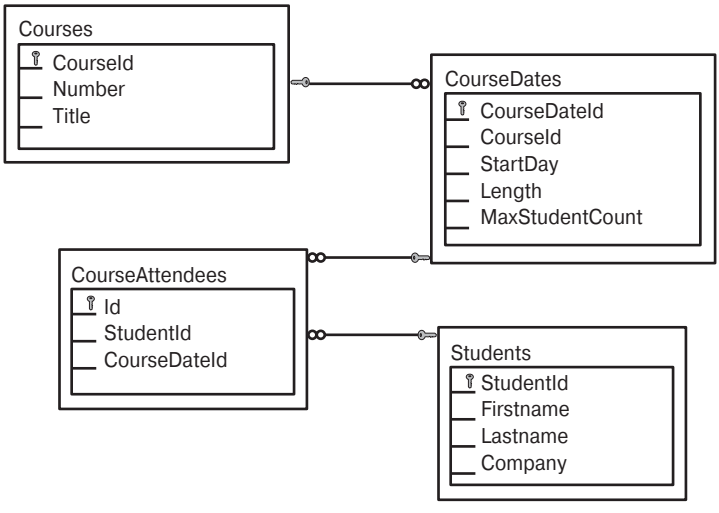

Рис. 21.2. Структура базы данных `CourseManagement`

```
using System;
namespace Wrox.ProCSharp.Transactions
{
[Serializable]
public class Student
{
  public Student() { }
  public Student(string firstname, string lastname)
  {
    this.firstname = firstname;
    this.lastname = lastname;
  }
  private string firstname;
  public string Firstname
  {
    get { return firstname; }
    set { firstname = value; }
  }
  private string lastname;
  public string Lastname
  {
    get { return lastname; }
    set { lastname = value; }
  }
  private string company;
  public string Company
  {
    get { return company; }
    set { company = value; }
  }
  private int id;
  public int Id
  {
    get { return id; }
    set { id = value; }
  }
```

```
  public override string ToString()
  {
    return firstname + " " + lastname;
  }
}
}
```

Добавление информации о студенте в базу данных выполняется методом `AddStudent()` класса `StudentData`. Здесь создается подключение ADO.NET для соединения с базой данных SQL Server, объект `SqlCommand` определяет SQL-оператор, а команда выполняется вызовом `ExecuteQuery()`.

```
using System;
using System.Collections.Generic;
using System.Data.SqlClient;
using System.Data;
using System.Transactions;
namespace Wrox.ProCSharp.Transactions
{
public class StudentData
{
  public void AddStudent(Student s)
  {
     SqlConnection connection = new SqlConnection(
           Properties.Settings.Default.CourseManagementConnectionString);
     connection.Open();
     try
     {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "INSERT INTO Students " +
             "(Firstname, Lastname, Company) VALUES " +
             "(@Firstname, @Lastname, @Company)";
        command.Parameters.AddWithValue("@Firstname", s.Firstname);
        command.Parameters.AddWithValue("@Lastname", s.Lastname);
        command.Parameters.AddWithValue("@Company", s.Company);
        command.ExecuteNonQuery();
     }
     finally
     {
        connection.Close();
     }
  }
}
}
```

Подробности об ADO.NET раскрываются в главе 25.

Традиционные транзакции

До того, как появилось пространство имен `System.Transaction`, вы должны были создавать транзакции непосредственно в ADO.NET либо реализовывать их с помощью компонентов, атрибутов и исполняющей системы COM+, которые содержались в пространстве имен `System.EnterpriseServices`. Теперь, чтобы вы могли сравнить новую модель транзакций с традиционными способами выполнения транзакций, рассмотрим вкратце, как работают транзакции ADO.NET и транзакции Enterprise Services.

Транзакции ADO.NET

Начнем с традиционных транзакций ADO.NET. Если вы не создаете транзакцию вручную, то возможна единственная транзакция с каждым SQL-оператором. Если несколько операторов должны участвовать в одной и той же транзакции, вам придется создавать транзакцию вручную.

Следующий фрагмент кода демонстрирует работу с транзакциями ADO.NET. Класс `SqlConnection` определяет метод `BeginTransaction()`, возвращающий объект типа `SqlTransaction`. Этот объект транзакции затем должен быть ассоциирован с каждой командой, участвующей в транзакции. Чтобы ассоциировать команду с транзакцией, установите свойство `Transaction` класса `SqlCommand` в экземпляр `SqlTransaction`. Если возникнет ошибка, вам нужно вызвать метод `Rollback()`, и все изменения будут отменены. Вы можете проверять наличие ошибок с помощью конструкции `try/catch` и выполнять откат внутри блока `catch`.

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;
namespace Wrox.ProCSharp.Transactions
{
public class CourseData
{
  public void AddCourse(Course c)
  {
     SqlConnection connection = new SqlConnection(
     Properties.Settings.Default.CourseManagementConnectionString);
     SqlCommand courseCommand = connection.CreateCommand();
     courseCommand.CommandText =
            "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
     connection.Open();
     SqlTransaction tx = connection.BeginTransaction();
     try
     {
        courseCommand.Transaction = tx;
        courseCommand.Parameters.AddWithValue("@Number", c.Number);
        courseCommand.Parameters.AddWithValue("@Title", c.Title);
        courseCommand.ExecuteNonQuery();
        tx.Commit();
     }
     catch (Exception ex)
     {
        Trace.WriteLine("Error: " + ex.Message);
        tx.Rollback();
     }
     finally
     {
        connection.Close();
     }
  }
}
}
```

Если у вас есть несколько команд, которые должны выполняться в одной транзакции, то каждая из них должна быть ассоциирована с транзакцией. Как каждая транзакция ассоциирована с соединением, так и каждая из этих команд должна быть ассоциирована с тем же экземпляром соединения; локальная транзакция всегда ассоциирована с одним соединением.

Если вы создадите объектную модель постоянного хранения, используя множество объектов, например, классы `Course` и `CourseDate`, которые должны существовать внутри одной транзакции, то в этом случае становится очень трудно использовать транзакции ADO.NET. Здесь необходимо передавать транзакцию всем объектам, принимающим участие в этой транзакции.

> Транзакции ADO.NET не являются распределенными. В транзакциях ADO.NET трудно заставить работать различные объекты в пределах одной и той же транзакции.

System.EnterpriseServices

В составе `System.EnterpriseServices` вы получаете бесплатно множество полезных служб. Одна из них — автоматические транзакции. Использование транзакции с `System.EnterpriseServices` обладает тем преимуществом, что вам не приходится иметь дело с транзакциями непосредственно; транзакции автоматически создаются исполняющей системой. Вы просто добавляете атрибут `[Transaction]` с транзакционными требованиями в класс. Атрибут `[Autocomplete]` помечает метод для автоматической установки бита состояния транзакции: если метод успешен, устанавливается бит успеха, так что транзакция может быть зафиксирована. Если случается исключение, транзакция откатывается.

```
using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
[Transaction(TransactionOption.Required)]
public class CourseData : ServicedComponent
{
  [AutoComplete]
  public void AddCourse(Course c)
  {
     SqlConnection connection = new SqlConnection(
            Properties.Settings.Default.CourseManagementConnectionString);
     SqlCommand courseCommand = connection.CreateCommand();
     courseCommand.CommandText =
            "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
     connection.Open();

     try
     {
        courseCommand.Parameters.AddWithValue("@Number", c.Number);
        courseCommand.Parameters.AddWithValue("@Title", c.Title);
        courseCommand.ExecuteNonQuery();
     }
     finally
     {
        connection.Close();
     }
  }
}
}
```

Огромное преимущество создания транзакций с помощью `System.EnterpriseServices` состоит в том, что множество объектов могут быть легко запущены в одной и той же транзакции, и транзакции очень легко использовать. Недостатком же является то, что требуется модель хостинга COM+, и то, что класс, использующий средства этой технологии, должен наследоваться от базового класса `ServicedComponent`.

`Enterprise.Services` и применение транзакционных служб COM+ рассматривается в главе 38.

System.Transactions

Пространство имен `System.Transactions`, доступное, начиная с версии .NET 2.0, привнесло новую транзакционную модель в приложения .NET. На рис. 21.3 показана диаграмма классов Visual Studio с транзакционными классами и их отношениями из пространства имен `System.Transactions`: `Transaction`, `CommittableTransaction`, `DependentTransaction` и `SubordinateTransaction`. Члены этих классов перечислены в табл. 21.1.

`Transaction` — базовый класс для всех транзакционных классов, определяет свойства, методы и события, доступные во всех транзакционных классах. `CommittableTransaction` — единственный транзакционный класс, поддерживающий фиксацию. У этого класса есть метод `Commit()`; все прочие транзакционные классы могут только выполнять откат. Класс `DependentTransaction` используется с транзакциями, зависящими от других транзакций. Зависимая транзакция может зависеть от транзакции, созданной внутри другой фиксируемой транзакции. Затем зависимая транзакция добавляет свое действие к исходу фиксируемой транзакции, если она успешна или нет. Класс `SubordinateTransaction` применяется в сочетании с координатором распределенных транзакций (Distributed Transaction Coordinator — DTC). Этот класс представляет транзакцию, не являющуюся корневой, но управляемой DTC.

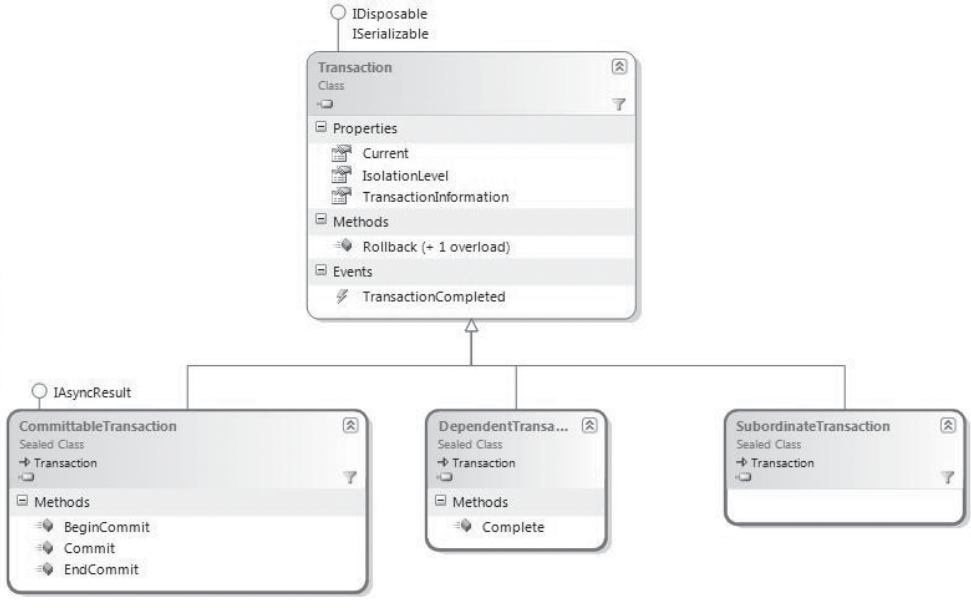

Рис. 21.3. Диаграмма классов транзакций Visual Studio

Таблица 21.1. Члены классов транзакций

Член	Описание
`Current`	`Current` — статическое свойство, не требующее наличия экземпляра. `Transaction.Current` возвращает объемлющую (включающую) транзакцию, если таковая существует. Объемлющие транзакции объясняются ниже.
`IsolationLevel`	Свойство `IsolationLevel` возвращает объект типа `IsolationLevel`. Этот тип представляет собой перечисление, определяющее тип доступа других транзакций к промежуточным результатам транзакции. Это затрагивает "I" из аббревиатуры ACID; не все транзакции являются изолированными.
`TransactionInformation`	Свойство `TransactionInformation` возвращает объект `TransactionInformation`. `TransactionInformation` представляет информацию о текущем состоянии транзакции, время создания транзакции и ее идентификаторы.

`EnlistVolatile()`
`EnlistDurable()`
`EnlistPromotableSinglePhase()`
	Методами `EnlistVolatile()`, `EnlistDurable()` и `EnlistPromotableSinglePhase()` вы можете задействовать диспетчеры ресурсов, участвующие в транзакции.

`Rollback()`	Методом `Rollback()` вы можете прервать транзакцию и отменить все, что было сделано в ее рамках, возвращая все в состояние, предшествовавшее началу транзакции.
`DependentClone()`	Методом `DependentClone()` вы можете создать транзакцию, зависящую от текущей транзакции.
`TransactionCompleted`	`TransactionCompleted` — это событие, инициируемое при завершении транзакции — будь оно успешным или неудачным.

Для демонстрации средств `System.Transaction` класс `Utilities` внутри отдельной сборки предлагает некоторые статические методы. Метод `AbortTx()` возвращает `true` или `false`, в зависимости от пользовательского ввода.

Метод `DisplayTransactionInformation()` получает объект `TransactionInformation` в качестве параметра, и отображает всю информацию о транзакции: время ее создания, состояние, локальные и распределенные идентификаторы.

```
public static class Utilities
{
public static bool AbortTx()
{
  Console.Write("Отменить транзакцию (y/n)?");
  return Console.ReadLine() == "y";
}
public static void DisplayTransactionInformation(
     TransactionInformation ti)
{
  if (ti != null)
  {
     Console.WriteLine("Время создания: {0:T}", ti.CreationTime);
     Console.WriteLine("Состояние: {0}", ti.Status);
     Console.WriteLine("Локальный ID: {0}", ti.LocalIdentifier);
     Console.WriteLine("Распределенный ID: {0}", ti.DistributedIdentifier);
     Console.WriteLine();
  }
}
}
```

Фиксируемые транзакции

Класс Transaction не может быть зафиксирован (commit) программно; он не имеет метода для фиксации транзакции. Базовый класс Transaction просто поддерживает прерывание транзакции. Единственный транзакционный класс, поддерживающий фиксацию, — это CommitableTransaction.

В ADO.NET транзакция может быть получена вместе с соединением. Чтобы обеспечить такую возможность, добавим метод AddStudent() к классу StudentData, который будет принимать объект System.Transactions.Transaction в качестве второго параметра. Объект tx будет задействован с соединением посредством вызова метода ElisTransaction класса SqlConnection. Таким образом, соединение ADO.NET ассоциируется с транзакцией.

```
public void AddStudent(Student s, Transaction tx)
{
SqlConnection connection = new SqlConnection(
     Properties.Settings.Default.CourseManagementConnectionString);
connection.Open();
try
{
  if (tx != null)
     connection.EnlistTransaction(tx);
  SqlCommand command = connection.CreateCommand();
  command.CommandText = "INSERT INTO Students (Firstname, Lastname, Company)
     VALUES (@Firstname, @Lastname, @Company)";
  command.Parameters.AddWithValue("@Firstname", s.Firstname);
  command.Parameters.AddWithValue("@Lastname", s.Lastname);
  command.Parameters.AddWithValue("@Company", s.Company);
  command.ExecuteNonQuery();
}
finally
{
  connection.Close();
}
}
```

В методе Main() консольного приложения CommitableTransaction первым делом создается транзакция типа CommitableTransaction. После создания транзакции информация отображается на консоли. Затем создается объект Student, и этот объект записывается в базу данных в методе AddStudent(). Если вы попробуете проверить запись в базе извне транзакции, то не сможете увидеть добавленного студента вплоть до ее завершения. В случае провала транзакции выполняется откат, и студент в базу не записывается.

После вызова метода AddStudent() вызывается вспомогательный метод Utilities.AbortTx(), чтобы запросить необходимость отмены транзакции. В случае если пользователь отменит транзакцию, возбуждается исключение типа ApplicationException, и в блоке catch происходит откат транзакции посредством вызова метода Rollback() класса Transaction. Запись не вносится в базу данных. Если же пользователь не отменит транзакцию, то метод Commit() ее подтвердит, и финальное состояние транзакции будет зафиксировано.

```
static void Main()
{
CommittableTransaction tx = new CommittableTransaction();
Utilities.DisplayTransactionInformation("TX создана",
     tx.TransactionInformation);
try
```

```
{
  Student s1 = new Student();
  s1.Firstname = "Neno";
  s1.Lastname = "Loye";
  s1.Company = "thinktecture";
  StudentData db = new StudentData();
  db.AddStudent(s1, tx);
  if (Utilities.AbortTx())
  {
    throw new ApplicationException("транзакция отменена");
  }
  tx.Commit();
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
  Console.WriteLine();
  tx.Rollback();
}
Utilities.DisplayTransactionInformation(tx.TransactionInformation);
}
```

Ниже вы можете видеть вывод приложения, в котором транзакция активна и имеет локальный идентификатор. При первом запуске выбрана отмена. После завершения транзакции вы можете видеть ее отмененное состояние.

```
TX Создана
Время создания: 7:30:49 PM
Состояние: Active
Локальный ID: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Распределенный ID: 00000000-0000-0000-0000-000000000000

TX завершена
Время создания: 7:30:49 PM
Состояние: Aborted
Локальный ID: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Распределенный ID: 00000000-0000-0000-0000-000000000000
Press any key to continue ...
```

При втором запуске приложения транзакция не прерывается. Она получает состояние зафиксированной, и данные записываются в базу.

```
TX Создана
Время создания: 7:33:04 PM
Состояние: Active
Локальный ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Распределенный ID: 00000000-0000-0000-0000-000000000000

Отменить транзакцию (y/n)? n

TX завершена
Время создания: 7:33:04 PM
Состояние: Committed
Локальный ID: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Распределенный ID: 00000000-0000-0000-0000-000000000000

Press any key to continue ...
```

Распространение транзакции

`System.Transactions` поддерживает распространяемые транзакции. В зависимости от ресурсов, принимающих участие в транзакции, создается локальная или распределенная транзакция. SQL Server 2005 поддерживает распространяемые транзакции, а

до сих пор вы имели дело только с транзакциями локальными. Пока распределенный идентификатор у нас всегда был установлен в 0. С ресурсами, не поддерживающими распространяемые транзакции, создаются транзакции распределенные. Если множество ресурсов добавляется к единственной транзакции, то такая транзакция может начинаться как локальная и при необходимости превращаться в распределенную. Подобное превращение происходит, когда к одной транзакции добавляется множество соединений с базой данных SQL Server 2005. Транзакция стартует, как локальная и затем превращается в распределенную.

Изменим теперь наше консольное приложение таким образом, чтобы оно добавляло второго студента, используя тот же объект транзакции `tx`. Поскольку каждый вызов метода `AddStudent()` открывает новое соединение, то получается, что два соединения ассоциированы с транзакцией после добавления второго студента.

```
static void Main()
{
CommittableTransaction tx = new CommittableTransaction();
Utilities.DisplayTransactionInformation("TX создана",
     tx.TransactionInformation);
try
{
  Student s1 = new Student();
  s1.Firstname = "Neno";
  s1.Lastname = "Loye";
  s1.Company = "thinktecture";
  StudentData db = new StudentData();
  db.AddStudent(s1, tx);
  Student s2 = new Student();
  s2.Firstname = "Dominick";
  s2.Lastname = "Baier";
  s2.Company = "thinktecture";
  db.AddStudent(s2, tx);
  Utilities.DisplayTransactionInformation("установлено 2-е соединение",
       tx.TransactionInformation);
  if (Utilities.AbortTx())
  {
    throw new ApplicationException("транзакция отменена");
  }
  tx.Commit();
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
  Console.WriteLine();
  tx.Rollback();
}
Utilities.DisplayTransactionInformation("TX завершена",
     tx.TransactionInformation);
}
```

Теперь, запустив это приложение, вы можете видеть при добавлении первого студента, что распределенный идентификатор равен 0, но при добавлении второго транзакция получает отличный от нуля распределенный идентификатор, ассоциированный с этой транзакцией.

```
TX создана
Время создания: 7:56:24 PM
Состояние: Active
Локальный ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Распределенный ID: 00000000-0000-0000-0000-0000000000
```

```
Установлено 2-е соединение
Время создания: 7:56:24 PM
Состояние: Active
Локальный ID: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Распределенный ID: 70762617-2ee8-4d23-aa87-6ac8c1418bdfd

Отменить транзакцию (y/n)?
```

Распространение транзакции требует запуска координатора распределенных транзакций (DTC). Если в вашей системе распределенная транзакция не работает, убедитесь в том, что запущена служба DTC. Запустив оснастку Component Services (Службы компонент) консоли MMC, вы можете видеть действительное состояние всех DTC-транзакций, запущенных в системе. Выбрав Transaction List (Список транзакций) в дереве, вы можете просмотреть все активные транзакции. Так, на рис. 21.4 видно, что имеется активная транзакция с тем же распределенным идентификатором, что был показан ранее на консольном выводе. Если вы проверяете вывод системы, убедитесь, что транзакция оснащена тайм-аутом, и будет прервана в случае его достижения. После тайм-аута вы более не сможете видеть транзакцию в списке активных транзакций. С помощью того же инструмента вы можете просмотреть статистику транзакций. Элемент Transaction Statistics (Статистика транзакций) показывает количество зафиксированных и отмененных транзакций.

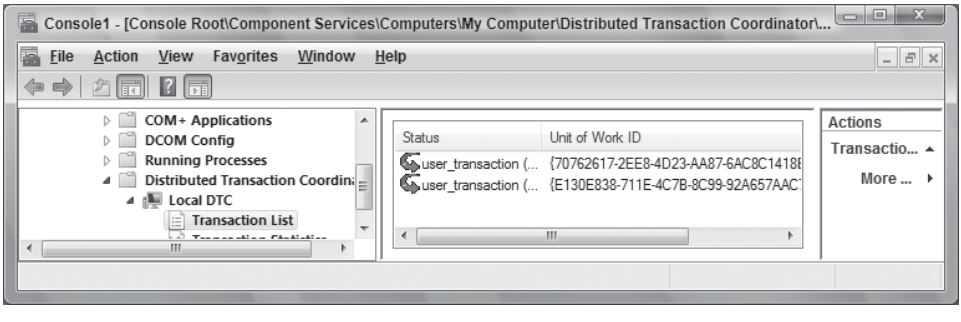

Рис. 21.4. Элемент Transaction Statistics

Зависимые транзакции

В случае зависимых транзакций вы можете влиять на одну транзакцию из множества потоков. Зависимая транзакция зависит от некоторой другой транзакции и влияет на ее исход.

Пример приложения `DependantTransaction` создает зависимую транзакцию для нового потока. `TxThread()` — это метод нового потока, которому объект `DependentTransaction` передается в качестве параметра.

Информация о зависимой транзакции отображается вспомогательным методом `DisplayTransactionInformation()`. Прежде чем поток завершится, вызывается метод `Complete()` зависимой транзакции, определяющий ее исход. Зависимая транзакция может определять исход транзакции, вызвав либо метод `Complete()`, либо метод `Rollback()`. Метод `Complete()` устанавливает бит успеха. Если корневая транзакция завершается, и все зависимые транзакции имеют бит успеха, установленный в `true`, то транзакция фиксируется. Если любая из зависимых транзакция устанавливает бит прерывания, вызывая `Rollback()`, то и вся корневая транзакция отменяется.

```
static void TxThread(object obj)
{
  DependentTransaction tx = obj as DependentTransaction;
  Utilities.DisplayTransactionInformation("Зависимая транзакция",
     tx.TransactionInformation);
  Thread.Sleep(3000);
  tx.Complete();
  Utilities.DisplayTransactionInformation("Зависимая TX завершена",
  tx.TransactionInformation);
}
```

В методе `Main()` первым делом создается корневая транзакция, путем создания экземпляра класса `CommitableTransaction`, и отображается информация об этой транзакции. Эта зависимая транзакция передается методу `TxThread()`, определенному, как точка входа нового потока.

Метод `DependentClone()` требует аргумента типа `DependentCloneOption`. `DependentCloneOption` представляет собой перечисление, состоящее из двух возможных значений: `BlockCommitUntilComplete` и `RollbackIfNotComplete`. Эта опция важна, если корневая транзакция завершается перед зависимой транзакцией. Установив данную опцию в `RollbackIfNotComplete`, транзакция прерывается, если зависимая транзакция не вызывает метод `Complete()` перед вызовом `Commit()` корневой транзакции. Установив опцию `BlockCommitUntilComplete`, метод `Commit()` ожидает, пока не будет ясен исход всех зависимых транзакций.

Затем вызывается метод `Commit()` класса `CommittableTransaction`, если пользователь не прерывает транзакцию.

Описанные ниже вопросы обсуждаются в главе 18.

```
static void Main()
{
CommittableTransaction tx = new CommittableTransaction();
Utilities.DisplayTransactionInformation("Корневая TX создана",
     tx.TransactionInformation);
try
{
  new Thread(TxThread).Start(
       tx.DependentClone(DependentCloneOption.BlockCommitUntilComplete));
  if (Utilities.AbortTx())
  {
     throw new ApplicationException("транзакция прервана");
  }
  tx.Commit();
}
catch (Exception ex)
{
  Console.WriteLine(ex.Message);
  tx.Rollback();
}

Utilities.DisplayTransactionInformation("TX завершена",
     tx.TransactionInformation);
}
```

В выводе приложения вы можете видеть корневую транзакцию вместе с ее идентификатором. Из-за опции `DependentCloneOption.BlockCommitUntilComplete` корневая транзакция ожидает в методе `Commit()`, пока не будет определен исход зависимой транзакции. Как только зависимая транзакция завершена, вся транзакция фиксируется.

```
Корневая TX создана
Время создания: 8:35:25 PM
Состояние: Active
Локальный ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный ID: 00000000-0000-0000-0000-0000000000

Прервать транзакцию (y/n)? n

Зависимая транзакция
Время создания: 8:35:25 PM
Состояние: Active
Локальный ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный ID: 00000000-0000-0000-0000-0000000000

Зависимая TX завершена
Корневая TX завершена
Время создания: 8:35:25 PM
Состояние: Committed
Локальный ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный ID: 00000000-0000-0000-0000-0000000000

Время создания: 8:35:25 PM
Состояние: Committed
Локальный ID: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный ID: 00000000-0000-0000-0000-0000000000

Press any key to continue ...
```

Включающие транзакции

Наибольшим преимуществом `System.Transactions` является средство включающих (ambient) транзакций. При использовании включающих транзакций нет необходимости вручную связывать соединение с транзакцией; это делается автоматически из включающих транзакций, поддерживающих ресурсы.

Включающая транзакция ассоциируется с текущим потоком. Вы можете получать и устанавливать включающую транзакцию через статическое свойство `Transaction.Current`. API-интерфейсы, поддерживающие включающие транзакции, проверяют это свойство, чтобы получить такую транзакцию и соединить с текущей локальной транзакцией. Соединения ADO.NET поддерживают включающие транзакции.

Вы можете создать объект `CommittableTransaction` и присвоить его свойству `Transaction.Current` для инициализации включающей транзакции. Другой способ создания такой транзакции состоит в применении класса `TransactionScope`. Конструктор `TransactionScope` создает включающую транзакцию. Поскольку реализован интерфейс `IDisposable`, вы можете легко использовать область действия транзакции, применив оператор `using`.

Члены класса `TransactionScope` перечислены в табл. 21.2.

Таблица 21.2. Члены класса `TransactionScope`

Член	Описание
Конструктор	С помощью конструктора `TransactionScope` вы можете определить транзакционные требования. Также вы можете передать ему существующую транзакцию и задать ее тайм-аут.
`Complete()`	Вызов `Complete()` позволяет установить бит успеха области действия транзакции.
`Dispose()`	Метод `Dispose()` завершает область действия и фиксирует или отменяет действия транзакции, если область действия ассоциирована с корневой транзакцией. Если бит успеха установлен в зависимой транзакции, метод `Dispose()` выполняет фиксацию, в противном случае осуществляется откат.

Поскольку класс TransactionScope реализует интерфейс IDisposable, вы можете определить область ее действия с помощью оператора using. Конструктор по умолчанию создает новую транзакцию. Немедленно после создания экземпляра TransactionScope транзакция ассоциируется со средством доступа get свойства Transaction.Current для отображения информации о транзакции на консоли.

Чтобы получить информацию о завершении транзакции, предусмотрен метод OnTransactionCompleted() как реакция на событие TransactionCompleted включающей транзакции.

Теперь новый объект Student создается и записывается в базу данных вызовом метода StudentData.AddStudent(). С включающей транзакцией больше нет необходимости передавать этому методу объект Transaction, поскольку класс SqlConnection поддерживает включающие транзакции и автоматически связывает их с соединением. Затем метод Complete() класса TransactionScope устанавливает бит успеха, и в конце оператора using экземпляр TransactionScope освобождается, то есть выполняется фиксация. Если метод Complete() не вызывается, то метод Dispose() отменяет транзакцию.

Если соединение ADO.NET не должно связываться с включающей транзакцией, вы можете установить значение `Enlist=false` *в строке соединения.*

```
static void Main()
{
  using (TransactionScope scope = new TransactionScope())
  {
    Transaction.Current.TransactionCompleted += OnTransactionCompleted;
    Utilities.DisplayTransactionInformation("Включающая TX создана",
           Transaction.Current.TransactionInformation);
    Student s1 = new Student();
    s1.Firstname = "Ingo";
    s1.Lastname = "Rammer";
    s1.Company = "thinktecture";
    StudentData db = new StudentData();
    db.AddStudent(s1);
    if (!Utilities.AbortTx())
      scope.Complete();
    else
      Console.WriteLine("транзакция будет отменена");
  }
} // Освобождение
static void OnTransactionCompleted(object sender, TransactionEventArgs e)
{
  TransactionInformation ti = e.Transaction.TransactionInformation;
  Console.WriteLine("Транзакция завершена; состояние: {0}, id: {1}",
       ti.Status, ti.LocalIdentifier);
}
```

Запустив приложение, вы можете увидеть активную включающую транзакцию после создания класса TransactionScope. Последний вывод приложения — вывод обработчика события TransactionCompleted, отображающий финальное состояние транзакции.

```
Включающая TX создана
Время создания: 9:55:40 PM
Состояние: Active
Локальный ID: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Распределенный ID: 00000000-0000-0000-0000-0000000000

Прервать транзакцию (y/n)? n
```

```
Транзакция завершена; состояние: Committed, id:
a06df6fb-7266-435e-b90e-f024f1d6966e:1

Press any key to continue ...
```

Вложенные области действия включающих транзакций

Используя класс TransactionScope, вы можете вкладывать области действия друг в друга. Вложенная область может располагаться непосредственно внутри другой области или внутри метода, вызванного из этой области. Вложенная область может использовать ту же транзакцию, что и внешняя область, подавлять транзакцию либо создавать новую транзакцию, независимую от внешней области. Требование для области действия заключается в том, что она должна быть определена перечислением TransactionScopeOption, передаваемым конструктору класса TransactionScope.

Значения доступные в перечислении TransactionScopeOption и их функциональность описаны в табл. 21.3.

Таблица 21.3. Значения перечисления TransactionScopeOption

Значение	Описание
Required	Required определяет, что область действия требует транзакции. Если внешняя область (контекст) уже содержит включающую транзакцию, то внутренняя область использует эту существующую транзакцию. Если же включающая транзакция не существует, то создается новая.
RequiresNew	RequiresNew всегда создает новую транзакцию. Если во внешнем контексте уже определена транзакция, то транзакция из вложенной области действия полностью независима. Обе транзакции могут фиксироваться или отменяться независимо друг от друга.
Suppress	С Suppress область действия не содержит включающей транзакции, независимо от того, содержит внешняя область транзакцию или нет.

В следующем примере определены две области, причем внутренняя область с помощью опции TransactionScopeOption.RequiresNew сконфигурирована как требующая новой транзакции.

```
using (TransactionScope scope = new TransactionScope())
{
  Transaction.Current.TransactionCompleted += OnTransactionCompleted;
  Utilities.DisplayTransactionInformation("Включающая TX создана",
     Transaction.Current.TransactionInformation);
  using (TransactionScope scope2 =
     new TransactionScope(TransactionScopeOption.RequiresNew))
  {
     Transaction.Current.TransactionCompleted += OnTransactionCompleted;
     Utilities.DisplayTransactionInformation(
        "Внутренний транзакционный контекст",
        Transaction.Current.TransactionInformation);
     scope2.Complete();
  }
  scope.Complete();
}
```

Запустив это приложение, вы увидите, что обе области имеют разные идентификаторы транзакций, хотя используют один и тот же поток. При наличии одного потока с разными включающими транзакциями в разных контекстах (областях), идентификаторы транзакций отличаются в последнем номере, следующем за GUID.

```
Включающая TX создана
Время создания: 11:01:09 PM
Состояние: Active
Локальный ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Распределенный ID: 00000000-0000-0000-0000-0000000000
Внутренний транзакционный контекст
Время создания: 11:01:09 PM
Состояние: Active
Локальный ID: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Распределенный ID: 00000000-0000-0000-0000-0000000000
Транзакция завершена: состояние: Committed, id:
54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Транзакция завершена: состояние: Committed, id:
54ac1276-5c2d-4159-84ab-36b0217c9c84:1
```

Если несколько потоков должны использовать одну и ту же включающую транзакцию, вы не можете просто создать транзакционный контекст и внутри него создать новый поток. Поскольку включающие транзакции привязаны к потоку, вновь созданный поток не имеет включающей транзакции. Если это необходимо, вы можете создать зависимую транзакцию, которая будет зависеть от включающей транзакции. Внутри нового потока вы можете назначить зависимую транзакцию включающей транзакции.

Уровень изоляции

В начале настоящей главы вы познакомились со свойствами ACID, используемыми для описания транзакций. Буква *I* (Изоляция) аббревиатуры *ACID* требуется не всегда. Из соображений производительности вы можете снизить степень требуемой изоляции, но при этом вы должны представлять себе возможные последствия изменения уровня изоляции. Проблемы, с которыми вы можете столкнуться, когда не полностью изолируете контекст извне транзакции, делятся на три описанных ниже категории.

- **Грязное чтение**. При *грязном чтении* другая транзакция может читать записи, измененные внутри данной транзакции. Поскольку данные, изменяемые внутри транзакции, могут быть откатаны к своему исходному состоянию, чтение такого промежуточного состояния из другой транзакции трактуется как "грязное", то есть данные не были зафиксированы. Этого можно избежать блокировкой изменяемых записей.
- **Невоспроизводимое чтение**. *Невоспроизводимое чтение* случается, когда данные читаются внутри транзакции, и пока эта транзакция выполняется, другая транзакция изменяет те же самые записи. Если запись читается повторно внутри транзакции, то вы получаете отличающийся результат — невоспроизводимый. Это можно предотвратить посредством блокировки чтения записей.
- **Фантомное чтение**. *Фантомное чтение* случается при чтении диапазона данных, например, с использованием конструкции `WHERE`. Другая транзакция может добавить новую запись, которая попадает в диапазон подлежащих чтению в транзакции. Новая запись с тем де выражением `WHERE` возвратит другое количество строк. Фантомное чтение может стать серьезной проблемой при выполнении оператора `UPDATE` для диапазона записей. Например, `UPDATE Addresses SET Zip=4711 WHERE (Zip=2315)` обновляет почтовый код во всех записях с 2315 на 4711. Однако после выполнения обновления в таблице могут остаться записи со значением почтового кода 2315, если другой пользователь добавит новую запись с кодом 2315 в то время, пока выполняется обновление. Этой проблемы можно избежать, если применить блокировку диапазона.

При определении требований изоляции вы можете установить уровень изоляции. Уровень изоляции устанавливается перечислением `IsolationLevel`, которое конфигурируется при создании транзакции (либо конструктором класса `CommitedTransaction`, либо конструктором класса `TransactionScope`). `IsolationLevel` определяет поведение блокировки. В табл. 21.4 перечислены значения перечисления `IsolationLevel`.

Таблица 21.4. Значения перечисления `IsolationLevel`

Значение	Описание
`ReadUncommitted`	С `ReadUncommitted` транзакции не изолированы друг друга. Этот уровень не предполагает ожидания блокированных записей из других транзакций. При этом незафиксированные данные могут быть прочитаны из других транзакций, то есть имеет место грязное чтение. Этот уровень обычно используется для чтения тех записей, для которых не страшно внесение промежуточных изменений, например, для отчетов.
`ReadCommitted`	`ReadCommitted` ожидает записей, заблокированных по записи из других транзакций. При этом не может произойти никакого грязного чтения. Этот уровень устанавливает блокировку чтения текущей записи и блокировку записи тех, что должны быть записаны до завершения транзакции. Поскольку при чтении каждая запись разблокируется при переходе к следующей, при этом может случиться невоспроизводимое чтение.
`RepeatableRead`	`RepeatableRead` удерживает блокировку чтения записей до тех пор, пока не завершится транзакция. При этом снимается проблема невоспроизводимого чтения. Однако все-таки могут возникнуть фантомные записи.
`Serializable`	`Serializable` удерживает блокировку диапазона. Пока исполняется транзакция, невозможно добавить новую запись, относящуюся к тому же диапазону, из которого прочитаны данные.
`Snapshot`	Уровень изоляции `Snapshot` — единственно возможный в SQL Server 2005. Этот уровень ограничивает блокировки, поскольку модифицируемые записи копируются. Таким образом, другие транзакции все же могут читать старые данные без необходимости ожидания разблокировки.
`Unspecified`	Уровень `Unspecified` указывает, что поставщик использует другое значение уровня блокировки.
`Chaos`	Уровень `Chaos` подобен `ReadUncommitted`, но в дополнение к действиям, характерным для `ReadUncommitted`, `Chaos` не блокирует обновляемые записи.

В табл. 21.5 резюмируются проблемы, которые могут возникнуть в результате установки наиболее часто используемых уровней изоляции транзакции.

Таблица 21.5. Проблемы, которые могут возникнуть в результате установки наиболее часто используемых уровней изоляции транзакции

Уровень изоляции	Грязное чтение	Невоспроизводимое чтение	Фантомное чтение
ReadUncommitted	Да	Да	Да
ReadCommitted	Нет	Да	Да
RepeatableRead	Нет	Нет	Да
Serializable	Нет	Нет	Нет

В следующем фрагменте кода показано, как может быть установлен уровень изоляции в классе TransactionScope. С помощью конструктора TransactionScope вы можете установить TransactionScopeOption, о которой мы говорили ранее, и TransactionOptions. Класс TransactionOptions позволяет определить IsolationLevel и Timeout.

```
TransactionOptions options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadUncommitted;
options.Timeout = TimeSpan.FromSeconds(90);
using (TransactionScope scope =
  new TransactionScope(TransactionScopeOption.Required, options))
{
  // Чтение данных без ожидания блокировок из других транзакций;
  // возможно грязное чтение.
}
```

Специализированные диспетчеры ресурсов

Одним из наибольших преимуществ новой транзакционной модели является то, что относительно легко создать специализированные диспетчеры ресурсов, участвующие в транзакции. Диспетчер ресурсов не обязательно управляет постоянными ресурсами, он также может управлять не постоянными или находящимися в памяти ресурсами; например, простыми int и обобщенными списками.

На рис. 21.5 показано отношение между диспетчером ресурсов и транзакционными классами. Диспетчер ресурсов реализует интерфейс IEnlistmentNotification, определяющий методы Prepare(), InDoubt(), Commit() и Rollback(). Диспетчер ресурсов реализует этот интерфейс, чтобы управлять транзакциями для ресурса. Чтобы быть частью транзакции, диспетчер ресурсов должен быть задействован в классе Transaction. Диспетчеры непостоянных ресурсов вызывают метод EnlistVolatile(); диспетчеры же постоянных ресурсов вызывают EnlistDurable(). В зависимости от исхода транзакции диспетчер транзакции вызывает методы интерфейса IEnlistmentNotification с диспетчером ресурсов.

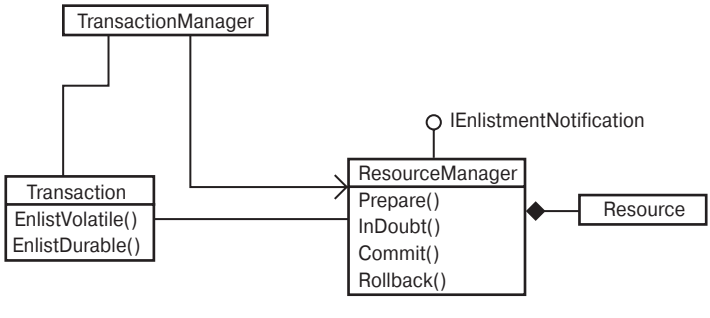

Рис. 21.5. Отношение между диспетчером ресурсов и транзакционными классами

В табл. 21.6 описаны методы интерфейса IEnlistmentNotification, которые вы должны реализовать в собственных диспетчерах ресурсов.

Помните о фазах, описанных ранее в настоящей главе: активной, подготовительной и фиксации.

Таблица 21.6. Методы интерфейса `IEnlistmentNotification`, которые необходимо реализовать в собственных диспетчерах ресурсов

Член	Описание
`Prepare()`	Диспетчер транзакций вызывает метод `Prepare` для подготовки к выполнению транзакции. Диспетчер ресурсов завершает подготовку, вызывая метод `Prepared()` параметра `PreparingEnlistment`, переданного методу `Prepare()`. Если работа не может быть выполнена успешно, диспетчер ресурсов информирует диспетчер транзакций, вызывая метод `ForceRollback()`. Диспетчер постоянных (долгоживущих) ресурсов должен записать в журнал о том, что он может завершить транзакцию успешно после фазы подготовки.
`Commit()`	Когда все диспетчеры ресурсов успешно подготовились к транзакции, диспетчер транзакций вызывает метод `Commit()`. Диспетчер ресурсов после этого может завершить свою работу, сделать ее результаты видимыми извне транзакции и вызвать метод `Done()` параметра `Enlistment`.
`Rollback()`	Если один из ресурсов не может успешно завершить подготовку к транзакции, диспетчер транзакций вызывает метод `Rollback()` для всех участвующих диспетчеров ресурсов. После того, как все будет возвращено в состояние, предшествовавшее транзакции, диспетчер ресурсов вызывает метод `Done()` параметра `Enlistment`.
`InDoubt()`	Если после того, как диспетчер транзакций вызвал метод `Commit()` (а ресурсы не возвратили итоговую информацию посредством вызова `Done()`), диспетчер транзакций вызывает метод `InDoubt()`.

Транзакционные ресурсы

Транзакционный ресурс должен хранить "живое" и временное значения. Живое значение читается извне транзакции и определяет корректное состояние при откате транзакции. Временное значение определяет корректное состояние транзакции при ее фиксации.

Чтобы сделать нетранзакционные типы транзакционными, пример класса `Transactional<T>` организует оболочку для необобщенного типа, так что вы можете использовать его примерно так:

```
Transactional<int> txInt = new Transactional<int>();
Transactional<string> txString = new Transactional<string>();
```

Взглянем на реализацию класса `Transactional<T>`. Живое значение управляемого ресурса содержит переменная `liveValue`, а временное значение, ассоциированное с транзакцией, хранится внутри `ResoureManager<T>`. Переменная `enlistedTransaction` ассоциирована с включающей транзакцией, если таковая существует.

```
using System;
using System.Diagnostics;
using System.Transactions;
using System.Collections.Generic;
namespace Wrox.ProCSharp.Transactions
{
public partial class Transactional<T>
{
  private T liveValue;
  private ResourceManager<T> enlistment;
  private Transaction enlistedTransaction;
```

В конструкторе `Transactional` живое значение присваивается переменной `liveValue`. Если конструктор вызывается изнутри включающей транзакции, то вызывается вспомогательный метод `GetEnlistment()`. Этот метод сначала проверяет наличие включающей транзакции и предупреждает, если таковой нет. Если транзакция еще не задействована, создается экземпляр вспомогательного класса `ResourceManager<T>`, и диспетчер ресурсов связывается с транзакцией вызовом метода `EnlistVolatile()`. Также в переменной `enlistedTransaction` устанавливается ссылка на включающую транзакцию. Если включающая транзакция не совпадает с задействованной транзакцией, возбуждается исключение. Реализация не поддерживает изменения одного и того же значения изнутри двух разных транзакций. Если перед вами стоит такое требование, вы можете создать блокировку и ожидать ее снятия изнутри одной транзакции, прежде чем выполнять изменения в другой транзакции.

```
public Transactional(T value)
{
  if (Transaction.Current == null)
  {
     this.liveValue = value;
  }
  else
  {
     this.liveValue = default(T);
     GetEnlistment().Value = value;
  }
}
public Transactional() : this(default(T)) {}
private ResourceManager<T> GetEnlistment()
{
  Transaction tx = Transaction.Current;
  Trace.Assert(tx != null, "Следует вызывать с включающей транзакцией");
  if (enlistedTransaction == null)
  {
     enlistment = new ResourceManager<T>(this, tx);
     tx.EnlistVolatile(enlistment, EnlistmentOptions.None);
     enlistedTransaction = tx;
     return enlistment;
  }
  else if (enlistedTransaction == Transaction.Current)
  {
     return enlistment;
  }
  else
  {
     throw new TransactionException(
            "Этот класс поддерживает участие только в одной транзакции");
  }
}
```

Свойство `Value` возвращает значение содержащегося класса и устанавливает его. Однако с транзакциями вы не можете просто устанавливать и возвращать переменную `liveValue`. Это может происходить только в том случае, когда объект находится вне транзакции. Чтобы сделать код более читабельным, свойство `Value` использует в реализации методы `GetValue()` и `SetValue()`.

```
public T Value
{
  get { return GetValue(); }
  set { SetValue(value); }
}
```

Метод GetValue() проверяет наличие включающей транзакции. Если ее нет, возвращается значение liveValue. Если же включающая транзакция обнаружена, ранее показанный метод GetEnlistment() возвращает диспетчер ресурсов, а свойство Value возвращает временное значение содержащегося в транзакции объекта.

Метод SetValue() очень похож на GetValue(); отличие лишь в том, что он изменяет живое или временное значение.

```
protected virtual T GetValue()
{
  if (Transaction.Current == null)
  {
     return liveValue;
  }
  else
  {
     return GetEnlistment().Value;
  }
}
protected virtual void SetValue(T value)
{
  if (Transaction.Current == null)
  {
     liveValue = value;
  }
  else
  {
     GetEnlistment().Value = value;
  }
}
```

Методы Commit() и Rollback(), реализованные в классе Transactional<T>, вызываются из диспетчера ресурсов. Метод Commit() устанавливает живое значение по временному значению, полученному в первом аргументе, и обнуляет переменную enlistedTransaction по завершении транзакции. С методом Rollback() транзакция также завершается, но временное значение игнорируется, а живое — используется.

```
  internal void Commit(T value, Transaction tx)
  {
     liveValue = value;
     enlistedTransaction = null;
  }
  internal void Rollback(Transaction tx)
  {
     enlistedTransaction = null;
  }
}
```

Поскольку диспетчер ресурсов, используемый классом Transactional<T>, применяется только внутри самого класса Transactional<T>, он реализован как вложенный класс. В конструкторе родительская переменная устанавливается так, что она ассоциирована с транзакционным классом-оболочкой. Временная переменная, используемая внутри транзакции, копируется из живого значения. Не забывайте о требованиях изоляции транзакций.

```
using System;
using System.Transactions;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

```
namespace Wrox.ProCSharp.Transactions
{
public partial class Transactional<T>
{
  internal class ResourceManager<T1> : IEnlistmentNotification
  {
     private Transactional<T1> parent;
     private Transaction currentTransaction;
     private T1 tempValue;
     internal ResourceManager(Transactional<T1> parent, Transaction tx)
     {
       this.parent = parent;
       tempValue = DeepCopy(parent.liveValue);
       currentTransaction = tx;
     }
     public T1 Value
     {
       get { return tempValue; }
       set { tempValue = value; }
     }
```

Поскольку временное значение может изменяться внутри транзакции, живое значение класса-оболочки не должно изменяться внутри транзакции. При создании копий некоторых классов можно вызвать метод `Clone()`, определенный в интерфейсе `ICloneable`. Однако если определен метод `Clone()`, он может позволять реализациям создавать и неглубокую, и глубокую копии. Если тип `T` содержит ссылочные типы и реализует неглубокое копирование, изменение временного значения также изменит исходное значение. Это будет противоречить принципу изоляции и средствам согласованности транзакции. В этом случае обязательно требуется глубокое копирование.

Чтобы создать глубокую копию, метод `DeepCopy()` сериализует и десериализует объект в потоке. Поскольку в C# 2.0 невозможно определить ограничение, устанавливающее обязательное требование сериализации типа `T`, статический конструктор класса `Transactional<T>` проверяет сериализуемость типа с помощью свойства `IsSerializable` объекта `Type`.

```
static ResourceManager()
{
  Type t = typeof(T1);
  Trace.Assert(t.IsSerializable, "Тип " + t.Name + " не сериализуем");
}
private T1 DeepCopy(T1 value)
{
  using (MemoryStream stream = new MemoryStream())
  {
     BinaryFormatter formatter = new BinaryFormatter();
     formatter.Serialize(stream, value);
     stream.Flush();
     stream.Seek(0, SeekOrigin.Begin);
     return (T1)formatter.Deserialize(stream);
  }
}
```

Интерфейс `IEnlistmentNotification` реализуется классом `ResourceManager<T>`. Это обязательное требование для участия в транзакциях.

Реализация метода `Prepare()` просто отвечает вызовом `Prepared()` с `preparingEnlistment`. Не должно быть проблемы с присвоением значения временной переменной живой переменной, так что метод `Prepare()` выполняется успешно. Данная реализация метода `Commit()` вызывает метод `Commit()` родителя, где пере-

менной liveValue присваивается значение tempValue. Метод Rollback() просто завершает работу и оставляет живое значение без изменений. В случае изменчивого (непостоянного) ресурса вам не придется много делать в методе InDoubt(). Здесь полезно, например, внести запись в журнал.

```
        public void Prepare(PreparingEnlistment preparingEnlistment)
        {
           preparingEnlistment.Prepared();
        }
      public void Commit(Enlistment enlistment)
      {
         parent.Commit(tempValue, currentTransaction);
         enlistment.Done();
      }
      public void Rollback(Enlistment enlistment)
      {
         parent.Rollback(currentTransaction);
         enlistment.Done();
      }
      public void InDoubt(Enlistment enlistment)
      {
         enlistment.Done();
      }
    }
  }
  }
```

Теперь класс Transactional<T> может использоваться для того, чтобы превращать нетранзакционные классы в транзакционные, например, int и string, или более сложные вроде Student — только если тип является сериализируемым.

```
using System;
using System.Transactions;
namespace Wrox.ProCSharp.Transactions
{
class Program
{
  static void Main()
  {
     Transactional<int> intVal = new Transactional<int>(1);
     Transactional<Student> student1 = new Transactional<Student>(new
     Student());
     student1.Value.Firstname = "Andrew";
     student1.Value.Lastname = "Wilson";
     Console.WriteLine("перед транзакцией, значение: {0}", intVal.Value);
     Console.WriteLine("перед транзакцией, студент: {0}", student1.Value);
     using (TransactionScope scope = new TransactionScope())
     {
       intVal.Value = 2;
       Console.WriteLine("внутри транзакции, значение: {0}", intVal.Value);
       student1.Value.Firstname = "Ten";
       student1.Value.Lastname = "Sixty-Nine";
       if (!Utilities.AbortTx())
            scope.Complete();
     }
     Console.WriteLine("вне транзакции, значение: {0}", intVal.Value);
     Console.WriteLine("вне транзакции, студент: {0}", student1.Value);
  }
}
}
```

Следующий консольный вывод демонстрирует запуск приложения с зафиксированной транзакцией:

```
перед транзакцией, значение: 1
перед транзакцией, студент: Andrew Wilson
внутри транзакции, значение: 2
Прервать транзакцию (y/n)? n
вне транзакции, значение: 2
вне транзакции, студент: Ten Sixty-Nine
Press any key to continue . . .
```

Транзакции в Windows Vista

Вы можете написать собственный диспетчер постоянных ресурсов, работающий с классами `File` и `Registry`. Основанный на файлах диспетчер постоянных ресурсов может копировать исходный файл и записывать изменения во временный файл внутри временного каталога, чтобы сделать изменения постоянными. При фиксации транзакции исходный файл заменяется временным файлом. Написание собственных диспетчеров ресурсов для файлов и реестра в Windows Vista не требуется. Эта система поддерживает "родные" транзакции, работающие с файловой системой и системным реестром. Для этого в Windows Vista предусмотрен новый API-интерфейс, включающий такие вызовы, как `CreateHardLinkTransacted`, `CreateSymbolicLinkTransacted`, `CopyFileTransacted` и так далее. Объединяет все эти вызовы то, что они требуют передачи им дескриптора транзакции в качестве аргумента; они не поддерживают включающих транзакций. Вызовы этого транзакционного API недоступны из .NET 3.0, но вы можете создать для них собственную обертку, применив метод `Invoke`.

Метод `Invoke` более подробно обсуждается в главе 23.

При вызове "родных" методов их параметры должны быть отображены на типы данных .NET. Из соображений безопасности .NET 2.0 предлагает класс `SafeHandler` и поддерживает критичную финализацию дескрипторных ресурсов. В зависимости от допустимых значений дескриптора, классы-наследники `SafeHandleMinusOneIsInvalid` и `SafeHandleZeroOrMinusOneIsInvalid` могут использоваться для помещения в оболочки "родных" дескрипторов. Чтобы отобразить дескриптор на транзакцию, предусмотрен класс `SafeTransactionHandle`.

```
using System;
using Microsoft.Win32.SafeHandles;
using System.Runtime.Versioning;
using System.Runtime.InteropServices;
using System.Runtime.ConstrainedExecution;
using System.Security.Permissions;
namespace Wrox.ProCSharp.Transactions
{
[SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
public sealed class SafeTransactionHandle : SafeHandleZeroOrMinusOneIsInvalid
{
  private SafeTransactionHandle() : base(true) { }
  public SafeTransactionHandle(IntPtr preexistingHandle, bool ownsHandle): base(ownsHandle)
  {
    SetHandle(preexistingHandle);
  }
  [DllImport("Kernel32.dll", SetLastError = true)]
  [ResourceExposure(ResourceScope.Machine)]
  [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
  [return: MarshalAs(UnmanagedType.Bool)]
```

```
  private static extern bool CloseHandle(IntPtr handle);
  [ResourceExposure(ResourceScope.Machine)]
  [ResourceConsumption(ResourceScope.Machine)]
  protected override bool ReleaseHandle()
  {
    return CloseHandle(handle);
  }
}
}
```

Интерфейс `IKernelTransaction` используется для получения дескриптора транзакции и передачи его транзакционным вызовам Windows API. Это — COM-интерфейс, который должен быть помещен в оболочку .NET посредством атрибутов COM Interop, как показано выше. GUID атрибута должен иметь точный идентификатор, как вы можете видеть в определении интерфейса, поскольку именно этот идентификатор используется в определении COM-интерфейса.

COM Interop рассматривается в главе 23.

```
using System;
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.Transactions
{
  [ComImport]
  [Guid("79427A2B-F895-40e0-BE79-B57DC82ED231")]
  [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
  public interface IKernelTransaction
  {
    void GetHandle(out SafeTransactionHandle ktmHandle);
  }
}
```

Класс `TransactedFile` является оболочкой для вызова Windows API `CreateFileTransacted()`. Параметры, определенные в вызове Windows API, отображаются на типы данных .NET. Параметр `txHandle` представляет дескриптор транзакции и относится к ранее определенному типу `SafeTransactionHandle`. Атрибут `DllImport` определяет вызов Windows API, реализованный в системной DLL-библиотекой `Kernel32.dll`.

```
using System;
using System.IO;
using System.Transactions;
using Microsoft.Win32.SafeHandles;
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.Transactions
{
public static class TransactedFile
{
  [DllImport("Kernel32.dll", CallingConvention = CallingConvention.StdCall,
             CharSet = CharSet.Unicode)]
  internal static extern SafeFileHandle CreateFileTransacted(
             String lpFileName,
             uint dwDesiredAccess,
             uint dwShareMode,
             IntPtr lpSecurityAttributes,
             uint dwCreationDisposition,
             int dwFlagsAndAttributes,
             SafeFileHandle hTemplateFile,
             SafeTransactionHandle txHandle,
             IntPtr miniVersion,
             IntPtr extendedParameter);
```

Чтобы облегчить использование вызовов Windows API в приложениях .NET, в классе `TransactedFile` определен метод `GetTransactedFileStream()`. Этот метод требует имени файла в качестве параметра и возвращает `System.IO.FileStream`.

`TransactionInterop.GetDtcTransaction()` создает интерфейсную ссылку `IKernelTransaction` на включающую транзакцию, которая передается в качестве аргумента методу `GetDtcTransaction()`. С использованием интерфейса `IKernelTransaction` создается дескриптор типа `SafeTransactionHandle`. Этот дескриптор затем передается помещенному в оболочку API-вызову `CreateFileTransacted()`. На основе возвращенного дескриптора файла создается новый экземпляр `FileStream`, который возвращается в вызывающий контекст.

```
internal const short FILE_ATTRIBUTE_NORMAL = 0x80;
internal const short INVALID_HANDLE_VALUE = -1;
internal const uint GENERIC_READ = 0x80000000;
internal const uint GENERIC_WRITE = 0x40000000;
internal const uint CREATE_NEW = 1;
internal const uint CREATE_ALWAYS = 2;
internal const uint OPEN_EXISTING = 3;
public static FileStream GetTransactedFileStream(string filename)
{
   IKernelTransaction ktx = (IKernelTransaction)
          TransactionInterop.GetDtcTransaction(Transaction.Current);
   SafeTransactionHandle txHandle;
   ktx.GetHandle(out txHandle);
   SafeFileHandle fileHandle = TransactedFile.CreateFileTransacted(
          filename, GENERIC_WRITE, 0, IntPtr.Zero, CREATE_ALWAYS,
          FILE_ATTRIBUTE_NORMAL, null, txHandle, IntPtr.Zero, IntPtr.Zero);
   return new FileStream(fileHandle, FileAccess.Write);
}
}
}
```

После этого использовать транзакционный API из кода .NET очень легко. Вы можете создать включающую транзакцию с помощью класса `TransactionScope` и применить класс `TransactedFile` внутри контекста включающего транзакционного контекста. Если транзакция прерывается, файл не записывается. Если же она фиксируется, вы найдете ваш файл в каталоге `temp`.

```
using System;
using System.Transactions;
using System.IO;
namespace Wrox.ProCSharp.Transactions
{
class Program
{
  static void Main()
  {
     using (TransactionScope scope = new TransactionScope())
     {
       FileStream stream =
            TransactedFile.GetTransactedFileStream("c:/temp/sample.txt");
       StreamWriter writer = new StreamWriter(stream);
       writer.WriteLine("Write a transactional file");
       writer.Close();
       if (!Utilities.AbortTx())
            scope.Complete();
     }
  }
}
}
```

То есть теперь вы можете использовать базы данных, временные ресурсы и файлы в пределах одной и той же транзакции.

Резюме

В настоящей главе вы ознакомились с атрибутами транзакций и способами создания и управления транзакциями посредством классов из пространства имен `System.Transactions`.

Транзакциям присущи свойства ACID: атомарность, согласованность, изоляция и постоянство. Не всегда требуются все эти свойства, например, в случае непостоянных (временных) ресурсов, которые не поддерживают опций постоянства и изоляции.

Простейший способ обращения с транзакциями — это создание включающей транзакции и применение класса `TransactionScope`. Используя одну и ту же транзакцию между несколькими потоками, вы можете применять класс `DependentTransaction` для описания зависимости от другой транзакции. Привлекая диспетчер ресурсов, реализующий интерфейс `IEnlistmentNotification`, вы можете создавать собственные ресурсы, способные участвовать в транзакциях.

И, наконец, вы узнали, как работать с транзакциями Windows Vista в рамках .NET Framework и C#.