Windows Phone Mango编程实践

***Windows Phone Mango Programming Practice***

第二篇 Silverlight交互篇

# 数据存储

## 独立存储

### 概述

独立存储（**isolated storage**）应用程序可以使用独立存储功能创建和维护本地存储。所有 I/O 操作限制于独立存储区，且不能直接访问底层操作系统的文件系统。此设计有助于提高安全性，并防止未经授权的访问损坏数据。

下图就是独立存储（**isolated storage**）的文件结构，如图6-1 文件结构。



图6-1 文件结构

Windows Phone独立存储APIs，如表6-1 独立存储APIs。

|  |  |
| --- | --- |
| **类和命名空间** | **作用** |
| IsolatedStorageException System.IO.IsolatedStorage | 独立存储操作失败时引发的异常 |
| System.IO.IsolatedStorage.IsolatedStorageFile | 表示文件和目录的独立存储区 |
| System.IO.IsolatedStorage.IsolatedStorageSettings | 独立存储数据词典 |
| System.IO.IsolatedStorage.IsolatedFileStream | 以文件流形式访问存储在独立存储中的文件 |

表6-1 独立存储APIs

### 最佳实践

Windows Phone 对于应用程序的独立存储占用的空间大小没有限制。因为限制应用程序使用的空间大小，可能会导致不友好的用户使用方案。

当卸载应用程序时，独立存储区内的文件夹和文件将一并被删除。

管理独立存储空间的最佳实践是什么呢？

对于临时数据和文件而言，如果您的应用程序创建的任何临时数据都采用独立存储区内，那么要定期清除不再使用的临时数据和文件。

对于用户生成的数据而言，如果应用程序允许用户创建数据，应该具有删除数据的选项。比如照片的应用程序，允许用户在拍照操作完成后删除照片。可以考虑将文件同步保存至云端，只保留最相关的数据，减少文件占用的空间。例如，Microsoft Outlook 的Windows Phone应用程序默认只在智能手机上保留三天的邮件。

对于应用程序数据而言，比如商店购物列表，或字典应用程序中所加载的单词列表。可考虑只将应用程序使用频率最高的数据从云端缓存到本地。

## 动手实践——独立存储实践

本节展示如何在应用程序中执行以下的独立存储任务：

* 获取应用程序的虚拟存储
* 创建一个父文件夹
* 创建和添加独立的存储文件中的文本
* 读取独立存储的文本文件

1. 打开**Visual Studio 2010**或者**Visual Studio 2010 Express**，新建**Silverlight for Windows Phone**工程，类型为**Windows Phone Application**，工程名称为[**IsolatedStorageApp**]，如图6-2 新建**Windows Phone**应用程序。

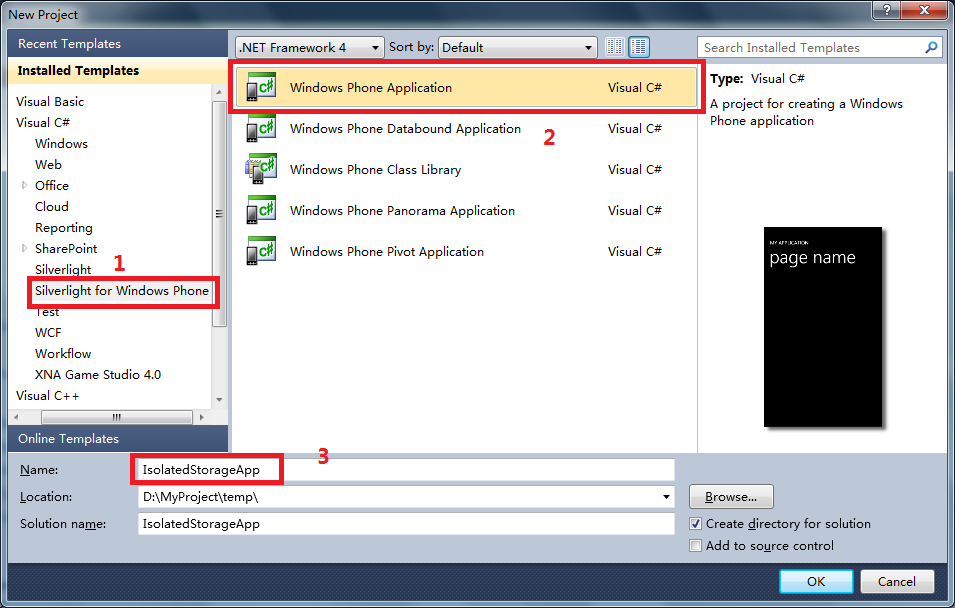


图6-2 新建Windows Phone应用程序

1. 添加独立存储命名空间（[Isolated Storage Namespaces](javascript:void(0))）。

打开MainPage.xaml.cs，添加IsolatedStorage命名空间。

Silverlight Project: IsolatedStorageApp File: MainPage.xaml.cs

using System.IO;

using System.IO.IsolatedStorage;

1. 设计首页显示样式。

将**TextBox**、**TextBlock**和**Button**控件从ToolBox拖拽至MainPage.xaml。修改**TextBoxTextBlock**和**Button**控件的**Name**和**Text** 属性，代码如下。

Silverlight Project: IsolatedStorageApp File: MainPage.xaml

<Grid x:Name="LayoutRoot" Background="Transparent">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="\*"/>

</Grid.RowDefinitions>

<!--TitlePanel contains the name of the application and page title-->

<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">

<TextBlock x:Name="ApplicationTitle" Text="ISOLATED STORAGE" Style="{StaticResource PhoneTextNormalStyle}"/>

<TextBlock x:Name="PageTitle" Text="独立存储" Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>

</StackPanel>

<!--ContentPanel - place additional content here-->

<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">

<TextBox Height="72" HorizontalAlignment="Left" Margin="-1,42,0,0" Name="txtWrite" Text="" VerticalAlignment="Top" Width="416" />

<Button Content="保存" Height="72" HorizontalAlignment="Left" Margin="-1,160,0,0" Name="btnWrite" VerticalAlignment="Top" Width="160" Click="txtWrite\_Click" />

<TextBlock Height="149" HorizontalAlignment="Left" Margin="12,275,0,0" Name="txtRead" Text="" VerticalAlignment="Top" Width="403" />

<Button Content="读取" Height="72" HorizontalAlignment="Left" Margin="-1,450,0,0" Name="btnRead" VerticalAlignment="Top" Width="160" Click="btnRead\_Click" />

</Grid>

</Grid>

1. 为**Button**控件添加事件响应处理。

双击两个**Button**控件，Visual Studio的智能感应将自动添加txtWrite\_Click和btnRead\_Click函数处理单击事件。

在txtWrite\_Click事件处理函数中，调用CreateDirectory方法创建文件夹FavorFolder，将TextBox中输入的内容保存在myFile.txt文件中。

Silverlight Project: IsolatedStorageApp File: MainPage.xaml.cs

private void txtWrite\_Click(object sender, RoutedEventArgs e)

{

//获取应用程序的独立存储空间

IsolatedStorageFile myStore = IsolatedStorageFile.GetUserStoreForApplication();

//创建新的文件夹

myStore.CreateDirectory("FavorFolder");

//Create a new file and assign a StreamWriter to the store and this new file (myFile.txt)

//Also take the text contents from the txtWrite control and write it to myFile.txt

StreamWriter writeFile = new StreamWriter(new IsolatedStorageFileStream("FavorFolder\\myFile.txt", FileMode.OpenOrCreate, myStore));

writeFile.WriteLine(txtWrite.Text);

writeFile.Close();

}

在btnRead \_Click事件处理函数中，读取FavorFolder中myFile.txt文件的内容，显示在TextBlock控件中。

Silverlight Project: IsolatedStorageApp File: MainPage.xaml.cs

private void btnRead\_Click(object sender, RoutedEventArgs e)

{

//Obtain a virtual store for application

IsolatedStorageFile myStore = IsolatedStorageFile.GetUserStoreForApplication();

//This code will open and read the contents of myFile.txt

//Add exception in case the user attempts to click “Read button first.

StreamReader readFile = null;

try

{

readFile = new StreamReader(new IsolatedStorageFileStream("FavorFolder\\myFile.txt", FileMode.Open, myStore));

string fileText = readFile.ReadLine();

// txtRead控件显示文件的内容

txtRead.Text = fileText;

readFile.Close();

}

catch

{

txtRead.Text = "Need to create directory and the file first.";

}

}

1. 编译调试结果如下。

图6-3 调试运行结果

## 本地数据库

Windows Phone Mango，可以将关系数据存储在本地数据库中，本地数据库作为一个文件存储在应用程序的独立存储空间。Windows Phone应用程序使用LINQ to SQL执行数据库的所有操作。LINQ to SQL用于定义数据库架构、选择数据，并将更改保存到数据库文件保存在在独立存储空间。LINQ to SQL是.Net框架的ORM（对象关系映射）平台的数据库。当应用程序执行LINQ语句在运行时，它转换为Transact - SQL对数据库执行操作；一旦数据库返回查询结果，LINQ to SQL将数据转换为应用程序对象。

LINQ to SQL提供了面向对象的方法，用于处理数据，包括对象模型和运行时。LINQ to SQL对象模型是由System.Data.Linq.DataContext对象构成，它充当代理服务器的本地数据库。LINQ to SQL 运行时负责连接本地数据库和应用程序的DataContext对象，DataContext对象对应数据库中的表，如图所示。

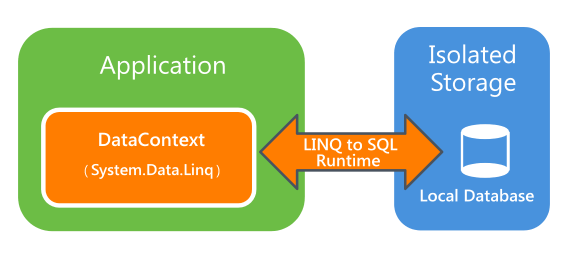


图6-4 LINQ to SQL

Windows Phone Mango的应用程序使用的LINQ to SQL访问本地数据库，LINQ to SQL提供一个用于操作存储在数据库中的数据的面向对象的方法。**System.Data.Linq.DataContext**类连接应用程序的对象模型与数据库中的数据。

### 定义数据上下文

System.Data.Linq.DataContext类映射应用程序数据对象到数据库实体。推荐使用SqlMetal工具生成 Windows Phone生成数据库上下文代码。

SqlMetal 命令行工具可为 .NET Framework 的 LINQ to SQL 组件生成代码和映射。

SQLMeta命令行工具与Visual Studio一起安装在Windows SDK 中。默认状态下，该文件位于 *drive*： \Program Files\Microsoft SDKs\Windows\v*n. nn\bin.*如果您未安装 Visual Studio，还可以通过下载[Windows SDK](http://go.microsoft.com/fwlink/?LinkId=142225) 来获取SQLMetal。

SQLMeta命令行工具示例如下。

生成一个包含提取的 SQL 元数据的 .dbml 文件：

sqlmetal /server:myserver /database:northwind /dbml:mymeta.dbml

使用 SQL Server Express 生成一个包含从 .mdf 文件中提取的 SQL 元数据的 .dbml 文件：

sqlmetal /dbml:mymeta.dbml mydbfile.mdf

生成一个包含从 SQL Server Express 中提取的 SQL 元数据的 .dbml 文件：

sqlmetal /server:.\sqlexpress /dbml:mymeta.dbml /database:northwind

基于 .dbml 元数据文件生成源代码：

sqlmetal /namespace:nwind /code:nwind.cs /language:csharp mymetal.dbml

直接基于 SQL 元数据生成源代码：

sqlmetal /server:myserver /database:northwind /namespace:nwind /code:nwind.cs /language:csharp

**注意：**

如果对 Northwind 示例数据库应用 **/pluralize** 选项，请注意以下行为。如果 SqlMetal 为表创建了行类型的名称，表名将采用单数形式。如果它为表创建了DataContext属性，则表名将采用复数形式。巧合的是，Northwind 示例数据库中的表名已采用复数形式。

若要创建一个本地数据库，必须首先定义数据上下文和实体的类，建立数据上下文和关系数据库之间数据的映射。LINQ to SQL对象的性能取决于数据上下文和关系数据库的映射。

为每个实体，通过使用 LINQ to SQL映射属性指定关系数据库的映射，如指定数据库的表、列、主键或者索引。例如，下面的代码显示数据，名为ToDoDataContext的数据库上下文和实体类的开头命名为ToDoItem。

public class ToDoDataContext : DataContext

{

// Specify the connection string as a static, used in main page and app.xaml.

public static string DBConnectionString = "Data Source=isostore:/ToDo.sdf";

// Pass the connection string to the base class.

public ToDoDataContext(string connectionString): base(connectionString) { }

// Specify a single table for the to-do items.

public Table<ToDoItem> ToDoItems;

}

// Define the to-do items database table.

[Table]

public class ToDoItem : INotifyPropertyChanged, INotifyPropertyChanging

{

// Define ID: private field, public property, and database column.

private int \_toDoItemId;

[Column(IsPrimaryKey = true, IsDbGenerated = true, DbType = "INT NOT NULL Identity", CanBeNull = false, AutoSync = AutoSync.OnInsert)]

public int ToDoItemId

{

get

{

return \_toDoItemId;

}

set

{

if (\_toDoItemId != value)

{

NotifyPropertyChanging("ToDoItemId");

\_toDoItemId = value;

NotifyPropertyChanged("ToDoItemId");

}

}

}

. . .

要在应用程序中使用本地数据库的功能，需要代码文件的顶部增加下面的引用。

using System.Data.Linq;

using System.Data.Linq.Mapping;

using Microsoft.Phone.Data.Linq;

using Microsoft.Phone.Data.Linq.Mapping;

一些常见的 LINQ to SQL映射属性如下表所示。

|  |  |  |
| --- | --- | --- |
| 属性 | 示例 | 说明 |
| TableAttribute | [Table] | 指定为实体类与数据库表相关联的类。 |
| ColumnAttribute | [Column(IsPrimaryKey = true)] | 将类与数据库表中的列相关联。IsPrimaryKey指定为主键，在默认情况下创建索引。 |
| IndexAttribute | [Index(Columns="Column1,Column2", IsUnique=true, Name="MultiColumnIndex")] | 在表级别，写入指定附加的索引，每个索引可以包括一个或多个列。 |
| AssociationAttribute | [Association(Storage="ThisEntityRefName", ThisKey="ThisEntityID", OtherKey="TargetEntityID")] | 指定一个属性来表示如主键关联的外键的关联。 |

表6-2 LINQ to SQL映射属性

创建DataContext对象后，您可以创建本地数据库和执行附加的数据库操作的数量。下面的代码示例演示如何创建数据库，基于ToDoDataContext类数据上下文。

// Create the database if it does not yet exist.

using (ToDoDataContext db = new ToDoDataContext("isostore:/ToDo.sdf"))

{

if (db.DatabaseExists() == false)

{

// Create the database.

db.CreateDatabase();

}

}

### 数据库查询

Windows Phone使用语言集成查询 (LINQ) 查询数据库。因为 SQL 查询在 LINQ 中引用的对象映射到数据库中的记录，LINQ to SQL有别于其他 LINQ 技术正在执行的查询的方式。一般LINQ查询在应用程序的内存中执行，而LINQ to SQL查询经TRANSACT-SQL转换后直接在数据库中执行。此设计在选择大型数据库的少量记录的查询时性能会明显增强。

在下面的示例中，一个名为toDoDB的DataContext对象的 LINQ to SQL 查询和结果放到ObservableCollection的ToDoItem对象命名的待办事项集合中延迟执行，延迟执行的数据库查询在待办事项集合中进行实例化。

// Define query to gather all of the to-do items.

var toDoItemsInDB = from ToDoItem todo in toDoDB.ToDoItems

select todo;

// Execute query and place results into a collection.

ToDoItems = new ObservableCollection<ToDoItem>(toDoItemsInDB);

### 插入数据

将数据插入数据库是一个两步过程。首先将对象添加到数据上下文中，然后调用SubmitChanges方法来保持数据作为数据库中的行的数据上下文。

在下面的示例中，创建ToDoItem对象并添加到名称为toDoDB的数据上下文的待办事项集合。

// Create a new to-do item based on text box.

ToDoItem newToDo = new ToDoItem { ItemName = newToDoTextBox.Text };

// Add the to-do item to the observable collection.

ToDoItems.Add(newToDo);

// Add the to-do item to the local database.

toDoDB.ToDoItems.InsertOnSubmit(newToDo);

**注意：**

在执行SubmitChanges方法之前，数据上下文中的数据不会保存到关系数据库。

### 更新数据

更新本地数据库中的数据有三个步骤。第一，要更新的对象数据库中查询。第二，修改所需的对象。最后，调用SubmitChanges方法，以将所做的更改保存到本地数据库。

下面的代码示例显示了应用程序OnNavigatedFrom方法中调用SubmitChanges更新本地数据库中的数据。在SubmitChanges方法调用之前，数据是不会更新至数据库的。

protected override void OnNavigatedFrom(System.Windows.Navigation.NavigationEventArgs e)

{

//Call base method

base.OnNavigatedFrom(e);

//Save changes to the database

toDoDB.SubmitChanges();

}

### 删除数据

删除数据库中的数据也包括三个步骤。首先，查询的数据库中删除的对象。然后，取决于您是否要删除一个或多个对象，调用DeleteOnSubmit或DeleteAllOnSubmit方法，分别放在挂起的删除状态的那些对象。最后，调用SubmitChanges方法，以将所做的更改保存到本地数据库。

在下面的示例中，一个ToDoItem对象是从名为toDoDB的数据库中删除。因为只有一个对象将被删除，在SubmitChanges之前调用DeleteOnSubmit方法。

//Get a handle for the to-do item bound to the button

ToDoItem toDoForDelete = button.DataContext as ToDoItem;

//Remove the to-do item from the observable collection

ToDoItems.Remove(toDoForDelete);

//Remove the to-do item from the local database

toDoDB.ToDoItems.DeleteOnSubmit(toDoForDelete);

//Save changes to the database

toDoDB.SubmitChanges();

**注意：**

在执行SubmitChanges方法之前，关系数据库中的数据不会被删除。

### [[http://i.msdn.microsoft.com/Hash/030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))更改数据库架构](javascript:void(0))

Widows Phone应用程序可能需要更改本地数据库架构。Microsoft.Phone.Data.Linq命名空间提供了有关于数据库架构更改的DatabaseSchemaUpdater类。

DatabaseSchemaUpdater类可以执行数据库，例如添加表、 列、 索引。对于更复杂的更改，需要创建一个新的数据库，并将数据复制到新的架构。DatabaseSchemaUpdater类提供了可用于以编程方式区分您的数据库的不同版本的DatabaseSchemaVersion属性。

数据库不会反映来自DatabaseSchemaUpdater对象的更新，直到调用Execute方法。当调用该方法时，所有的更改将被提交到本地数据库作为单个事务，包括版本更新。

下面的示例演示如何使用DatabaseSchemaUpdater类修改基于DatabaseSchemaVersion属性的数据库。

using (ToDoDataContext db = new ToDoDataContext(("isostore:/ToDo.sdf")))

{

//Create the database schema updater

DatabaseSchemaUpdater dbUpdate = db.CreateDatabaseSchemaUpdater();

//Get database version

int dbVersion = dbUpdate.DatabaseSchemaVersion;

//Update database as applicable

if (dbVersion < 5)

{ //Copy data from existing database to new database

MigrateDatabaseToLatestVersion();

}

else if (dbVersion == 5)

{ //Add column to existing database to match the data context

dbUpdate.AddColumn<ToDoItem>("TaskURL");

dbUpdate.DatabaseSchemaVersion = 6;

dbUpdate.Execute();

}

}

**注意：**

在应用程序更新过程中不会改变任何独立存储，包括本地数据库文件中保存的文件。

### [[http://i.msdn.microsoft.com/Hash/030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))数据库安全](javascript:void(0))

本地数据库提供密码保护和加密来帮助保护您的数据库。当您使用数据库密码时，加密整个数据库。

下面的示例演示如何创建加密的数据库。

// Create the data context, specify the database file location and password

ToDoDataContext db = new ToDoDataContext ("Data Source=’isostore:/ToDo.sdf’;Password=’securepassword’");

// Create an encrypted database after confirming that it does not exist

if (!db.DatabaseExists()) db.CreateDatabase();

**注意：**

如果只有有限的非索引列需要进行加密，可通过添加到数据库之前将数据加密，而不是加密整个数据库来实现更好的性能。

## 动手实践——本地数据库

### 前提

安装[**Silverlight for Windows Phone Toolkit**](http://silverlight.codeplex.com/)，下载地址<http://silverlight.codeplex.com/> ，如图6-5 Silverlight for Windows Phone Toolkit。





图6-5 Silverlight for Windows Phone Toolkit

### MVVM设计模型

应用程序采用MVVM设计模型，MVVM设计模型包含View、ViewModel和Model。

* **View** –展示用户界面以及响应用户操作
* **ViewModel** – 连接用户操作和应用程序的数据。
* **Model** – 管理应用程序的数据

本例的本地数据库访问的MVVM设计模型的Model由继承于**System.Data.Linq.DataContext**的**DataContextBase**类实现，**ViewModelItemsBase**类和**ViewModelBase**类则构成ViewModel。

### 添加引用

在工程[LocalDatabaseSample]中右键点击引用[References]，选择[Add Reference…]，添加Microsoft.Phone.Controls.Toolkit的引用。在Browse选项卡中查找Microsoft.Phone.Controls.Toolkit.dll，并将其添加到工程中。默认安装地址C:\Program Files\Microsoft SDKs\Windows Phone\v7.0\Toolkit\Feb11\Bin。具体的文件位置取决于您安装Silverlight for Windows Phone Toolkit的位置。

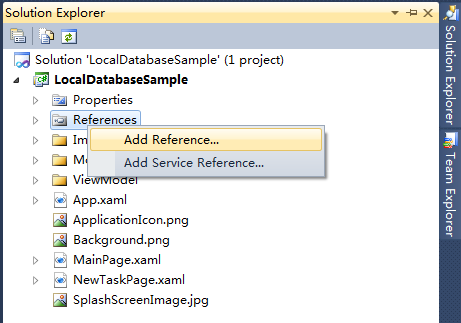
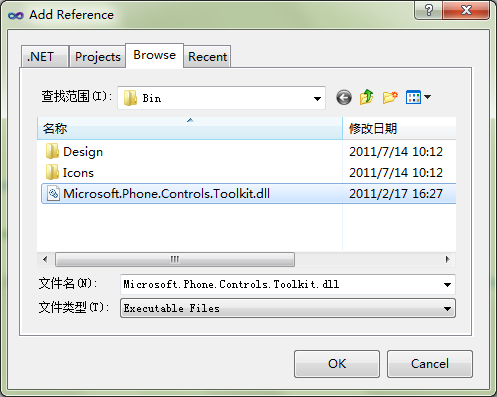
 

图6-6 添加引用

### 创建模型Model

在本节中创建LINQ to SQL数据上下文和表示数据库表和关联的对象。首先将创建文件，并为每个表添加模板。然后将生成出每个表和创建数据上下文。

1. 准备数据模型文件

在ToDoDataContext.cs中有如下的引用，包含数据模型的类的命名空间。

Silverlight Project: LocalDatabaseSample File: ToDoDataContext.cs

using System;

using System.ComponentModel;

using System.Data.Linq;

using System.Data.Linq.Mapping;

在ToDoDataContext.cs，下面的代码表示本地数据库表的基本模板。

[table]属性指定的类将表示数据库表。

INotifyPropertyChanged接口用于跟踪属性值变更。

INotifyPropertyChanging接口跟踪变更引起的内存限制。

[Column(IsVersion = true)]属性表示的二进制版本号，能显著提高了表更新性能。

[Table]

public class AddTableNameHere : INotifyPropertyChanged, INotifyPropertyChanging

{

//

// TODO: Add columns and associations, as applicable, here.

//

// Version column aids update performance.

[Column(IsVersion = true)]

private Binary \_version;

#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged;

// Used to notify that a property changed

private void NotifyPropertyChanged(string propertyName)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

#endregion

#region INotifyPropertyChanging Members

public event PropertyChangingEventHandler PropertyChanging;

// Used to notify that a property is about to change

private void NotifyPropertyChanging(string propertyName)

{

if (PropertyChanging != null)

{

PropertyChanging(this, new PropertyChangingEventArgs(propertyName));

}

}

#endregion

}

**注意：**

表类不能被继承。若要实现INotifyPropertyChanged和INotifyPropertyChanging，INotifyPropertyChanged和INotifyPropertyChanging的成员需要添加到表中的每个类。

以ToDoItem重命名AddTableNameHere类。此类将存储待办任务信息。

以ToDoCategory重命名AddTableNameHere类。此类将存储类别的列表。

1. 实现ToDoItem 类

ToDoItem 类三个属性添加至数据库：ToDoItemId, ItemName, 和IsCompleted。[Column]为LINQ to SQL运行时属性，表示数据库的列。

ToDoCategory类与ToDoItem 类的实现方式相同。

Silverlight Project: LocalDatabaseSample File: ToDoDataContext.cs

[Table]

public class ToDoItem : INotifyPropertyChanged, INotifyPropertyChanging

{

// Define ID: private field, public property, and database column.

private int \_toDoItemId;

[Column(IsPrimaryKey = true, IsDbGenerated = true, DbType = "INT NOT NULL Identity", CanBeNull = false, AutoSync = AutoSync.OnInsert)]

public int ToDoItemId

{

get { return \_toDoItemId; }

set

{

if (\_toDoItemId != value)

{

NotifyPropertyChanging("ToDoItemId");

\_toDoItemId = value;

NotifyPropertyChanged("ToDoItemId");

}

}

}

// Define item name: private field, public property, and database column.

private string \_itemName;

[Column]

public string ItemName

{

get { return \_itemName; }

set

{

if (\_itemName != value)

{

NotifyPropertyChanging("ItemName");

\_itemName = value;

NotifyPropertyChanged("ItemName");

}

}

}

// Define completion value: private field, public property, and database column.

private bool \_isComplete;

[Column]

public bool IsComplete

{

get { return \_isComplete; }

set

{

if (\_isComplete != value)

{

NotifyPropertyChanging("IsComplete");

\_isComplete = value;

NotifyPropertyChanged("IsComplete");

}

}

}

// Version column aids update performance.

[Column(IsVersion = true)]

private Binary \_version;

#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged;

// Used to notify that a property changed

private void NotifyPropertyChanged(string propertyName)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

#endregion

#region INotifyPropertyChanging Members

public event PropertyChangingEventHandler PropertyChanging;

// Used to notify that a property is about to change

private void NotifyPropertyChanging(string propertyName)

{

if (PropertyChanging != null)

{

PropertyChanging(this, new PropertyChangingEventArgs(propertyName));

}

}

#endregion

}

此代码定义的ToDoItem和ToDoCategory的表之间的关联。\_categoryId字段存储对应的待办事项的类别的标识符。\_Category实体引用标识要与此表相关联的其他表。

Silverlight Project: LocalDatabaseSample File: ToDoDataContext.cs

// Internal column for the associated ToDoCategory ID value

[Column]

internal int \_categoryId;

// Entity reference, to identify the ToDoCategory "storage" table

private EntityRef<ToDoCategory> \_category;

// Association, to describe the relationship between this key and that "storage" table

[Association(Storage = "\_category", ThisKey = "\_categoryId", OtherKey = "Id", IsForeignKey = true)]

public ToDoCategory Category

{

get { return \_category.Entity; }

set

{

NotifyPropertyChanging("Category");

\_category.Entity = value;

if (value != null)

{

\_categoryId = value.Id;

}

NotifyPropertyChanging("Category");

}

}

1. 实现ToDoCategory类

ToDoCategory类和ToDoItem类相关联， ToDoCategory类中定义实体关系的集合方的设置，包括删除和添加

Silverlight Project: LocalDatabaseSample File: ToDoDataContext.cs

[Table]

public class ToDoCategory : INotifyPropertyChanged, INotifyPropertyChanging

{

// Define ID: private field, public property, and database column.

private int \_id;

[Column(DbType = "INT NOT NULL IDENTITY", IsDbGenerated = true, IsPrimaryKey = true)]

public int Id

{

get { return \_id; }

set

{

NotifyPropertyChanging("Id");

\_id = value;

NotifyPropertyChanged("Id");

}

}

// Define category name: private field, public property, and database column.

private string \_name;

[Column]

public string Name

{

get { return \_name; }

set

{

NotifyPropertyChanging("Name");

\_name = value;

NotifyPropertyChanged("Name");

}

}

// Define the entity set for the collection side of the relationship.

private EntitySet<ToDoItem> \_todos;

[Association(Storage = "\_todos", OtherKey = "\_categoryId", ThisKey = "Id")]

public EntitySet<ToDoItem> ToDos

{

get { return this.\_todos; }

set { this.\_todos.Assign(value); }

}

// Assign handlers for the add and remove operations, respectively.

public ToDoCategory()

{

\_todos = new EntitySet<ToDoItem>(

new Action<ToDoItem>(this.attach\_ToDo),

new Action<ToDoItem>(this.detach\_ToDo)

);

}

// Called during an add operation

private void attach\_ToDo(ToDoItem toDo)

{

NotifyPropertyChanging("ToDoItem");

toDo.Category = this;

}

// Called during a remove operation

private void detach\_ToDo(ToDoItem toDo)

{

NotifyPropertyChanging("ToDoItem");

toDo.Category = null;

}

// Version column aids update performance.

[Column(IsVersion = true)]

private Binary \_version;

#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged;

// Used to notify that a property changed

private void NotifyPropertyChanged(string propertyName)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

#endregion

#region INotifyPropertyChanging Members

public event PropertyChangingEventHandler PropertyChanging;

// Used to notify that a property is about to change

private void NotifyPropertyChanging(string propertyName)

{

if (PropertyChanging != null)

{

PropertyChanging(this, new PropertyChangingEventArgs(propertyName));

}

}

#endregion

}

1. 创建LINQ to SQL数据上下文

此数据上下文将指定两个表：Items和Categories。Items表将存储待办任务，基于ToDoItem类。Categories表存储待办任务类别，基于ToDoCategory类。

Silverlight Project: LocalDatabaseSample File: ToDoDataContext.cs

public class ToDoDataContext : DataContext

{

// Pass the connection string to the base class.

public ToDoDataContext(string connectionString)

: base(connectionString)

{ }

// Specify a table for the to-do items.

public Table<ToDoItem> Items;

// Specify a table for the categories.

public Table<ToDoCategory> Categories;

}

### [[http://i.msdn.microsoft.com/Hash/030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))创建ViewModel](javascript:void(0))

在本节中，创建的应用程序的 ViewModel，ViewModel执行对数据库操作，实现跟踪INotifyPropertyChanged接口的更改。

本地数据库上执行操作的是 ViewModel 的核心功能。LINQ to SQL 数据方面，toDoDB，是引用在 ViewModel 和 ViewModel 的构造函数中创建。SaveChangesToDB方法调用SubmitChanges执行对数据库的操作。

Silverlight Project: LocalDatabaseSample File: ToDoViewModel.cs

using System.Collections.Generic;

using System.Collections.ObjectModel;

using System.ComponentModel;

using System.Linq;

// Directive for the data model.

using LocalDatabaseSample.Model;

namespace LocalDatabaseSample.ViewModel

{

public class ToDoViewModel : INotifyPropertyChanged

{

// LINQ to SQL data context for the local database.

private ToDoDataContext toDoDB;

// Class constructor, create the data context object.

public ToDoViewModel(string toDoDBConnectionString)

{

toDoDB = new ToDoDataContext(toDoDBConnectionString);

}

//

// TODO: Add collections, list, and methods here.

//

// Write changes in the data context to the database.

public void SaveChangesToDB()

{

toDoDB.SubmitChanges();

}

#region INotifyPropertyChanged Members

public event PropertyChangedEventHandler PropertyChanged;

// Used to notify Silverlight that a property has changed.

private void NotifyPropertyChanged(string propertyName)

{

if (PropertyChanged != null)

{

PropertyChanged(this, new PropertyChangedEventArgs(propertyName));

}

}

#endregion

}

}

1. 创建可观察的集合和列表

下面的代码指定可观察的集合，用于在主页面的Pivot（枢轴）控制：AllToDoItems、HomeToDoItems、WorkToDoItems、HobbiesToDoItems。此外可以指定使用新的任务页的ListPicker控制的类别列表。

Silverlight Project: LocalDatabaseSample File: ToDoViewModel.cs

// All to-do items.

private ObservableCollection<ToDoItem> \_allToDoItems;

public ObservableCollection<ToDoItem> AllToDoItems

{

get { return \_allToDoItems; }

set

{

\_allToDoItems = value;

NotifyPropertyChanged("AllToDoItems");

}

}

// To-do items associated with the home category.

private ObservableCollection<ToDoItem> \_homeToDoItems;

public ObservableCollection<ToDoItem> HomeToDoItems

{

get { return \_homeToDoItems; }

set

{

\_homeToDoItems = value;

NotifyPropertyChanged("HomeToDoItems");

}

}

// To-do items associated with the work category.

private ObservableCollection<ToDoItem> \_workToDoItems;

public ObservableCollection<ToDoItem> WorkToDoItems

{

get { return \_workToDoItems; }

set

{

\_workToDoItems = value;

NotifyPropertyChanged("WorkToDoItems");

}

}

// To-do items associated with the hobbies category.

private ObservableCollection<ToDoItem> \_hobbiesToDoItems;

public ObservableCollection<ToDoItem> HobbiesToDoItems

{

get { return \_hobbiesToDoItems; }

set

{

\_hobbiesToDoItems = value;

NotifyPropertyChanged("HobbiesToDoItems");

}

}

// A list of all categories, used by the add task page.

private List<ToDoCategory> \_categoriesList;

public List<ToDoCategory> CategoriesList

{

get { return \_categoriesList; }

set

{

\_categoriesList = value;

NotifyPropertyChanged("CategoriesList");

}

}

1. 加载集合和列表

ViewModel从本地数据库加载集合和数据的列表。

Silverlight Project: LocalDatabaseSample File: ToDoViewModel.cs

// Query database and load the collections and list used by the pivot pages.

public void LoadCollectionsFromDatabase()

{

// Specify the query for all to-do items in the database.

var toDoItemsInDB = from ToDoItem todo in toDoDB.Items

select todo;

// Query the database and load all to-do items.

AllToDoItems = new ObservableCollection<ToDoItem>(toDoItemsInDB);

// Specify the query for all categories in the database.

var toDoCategoriesInDB = from ToDoCategory category in toDoDB.Categories

select category;

// Query the database and load all associated items to their respective collections.

foreach (ToDoCategory category in toDoCategoriesInDB)

{

switch (category.Name)

{

case "Home":

HomeToDoItems = new ObservableCollection<ToDoItem>(category.ToDos);

break;

case "Work":

WorkToDoItems = new ObservableCollection<ToDoItem>(category.ToDos);

break;

case "Hobbies":

HobbiesToDoItems = new ObservableCollection<ToDoItem>(category.ToDos);

break;

default:

break;

}

}

// Load a list of all categories.

CategoriesList = toDoDB.Categories.ToList();

}

1. 添加和删除数据的操作

向应用程序添加新待办任务调用AddToDoItem方法。

Silverlight Project: LocalDatabaseSample File: ToDoViewModel.cs

// Add a to-do item to the database and collections.

public void AddToDoItem(ToDoItem newToDoItem)

{

// Add a to-do item to the data context.

toDoDB.Items.InsertOnSubmit(newToDoItem);

// Save changes to the database.

toDoDB.SubmitChanges();

// Add a to-do item to the "all" observable collection.

AllToDoItems.Add(newToDoItem);

// Add a to-do item to the appropriate filtered collection.

switch (newToDoItem.Category.Name)

{

case "Home":

HomeToDoItems.Add(newToDoItem);

break;

case "Work":

WorkToDoItems.Add(newToDoItem);

break;

case "Hobbies":

HobbiesToDoItems.Add(newToDoItem);

break;

default:

break;

}

}

应用程序中调用DeleteToDoItem方法实现删除数据。

Silverlight Project: LocalDatabaseSample File: ToDoViewModel.cs

// Remove a to-do task item from the database and collections.

public void DeleteToDoItem(ToDoItem toDoForDelete)

{

// Remove the to-do item from the "all" observable collection.

AllToDoItems.Remove(toDoForDelete);

// Remove the to-do item from the data context.

toDoDB.Items.DeleteOnSubmit(toDoForDelete);

// Remove the to-do item from the appropriate category.

switch (toDoForDelete.Category.Name)

{

case "Home":

HomeToDoItems.Remove(toDoForDelete);

break;

case "Work":

WorkToDoItems.Remove(toDoForDelete);

break;

case "Hobbies":

HobbiesToDoItems.Remove(toDoForDelete);

break;

default:

break;

}

// Save changes to the database.

toDoDB.SubmitChanges();

}

### [创建View](javascript:void(0))

在App.xaml.cs文件添加Model和ViewModel的引用。

Silverlight Project: LocalDatabaseSample File: App.xaml.cs

// Directives

using LocalDatabaseSample.Model;

using LocalDatabaseSample.ViewModel;

在App.xaml.cs文件添加Model和ViewModel的构造函数。

Silverlight Project: LocalDatabaseSample File: App.xaml.cs

// The static ViewModel, to be used across the application.

private static ToDoViewModel viewModel;

public static ToDoViewModel ViewModel

{

get { return viewModel; }

}

应用程序对象被实例化时，首先判断本地数据库是否存在，如果不存在则创建本地数据库，然后ViewModel对象创建和加载数据。

Silverlight Project: LocalDatabaseSample File: App.xaml.cs

// Specify the local database connection string.

string DBConnectionString = "Data Source=isostore:/ToDo.sdf";

// Create the database if it does not exist.

using (ToDoDataContext db = new ToDoDataContext(DBConnectionString))

{

if (db.DatabaseExists() == false)

{

// Create the local database.

db.CreateDatabase();

// Prepopulate the categories.

db.Categories.InsertOnSubmit(new ToDoCategory { Name = "Home" });

db.Categories.InsertOnSubmit(new ToDoCategory { Name = "Work" });

db.Categories.InsertOnSubmit(new ToDoCategory { Name = "Hobbies" });

// Save categories to the database.

db.SubmitChanges();

}

}

// Create the ViewModel object.

viewModel = new ToDoViewModel(DBConnectionString);

// Query the local database and load observable collections.

viewModel.LoadCollectionsFromDatabase();

在主页面MainPage.xaml.cs，下面代码显示页面的事件处理代码。当单击新任务按钮时，NavigationService对象用来导航到新的任务页面。当调用删除按钮时，相应的ToDoItem对象检索并发送到 ViewModel 的DeleteToDoItem方法。每当用户导航离开该页面，在数据方面的更改将自动保存到本地数据库使用SaveChangesToDB方法。

Silverlight Project: LocalDatabaseSample File: MainPage.xaml.cs

private void newTaskAppBarButton\_Click(object sender, EventArgs e)

{

NavigationService.Navigate(new Uri("/NewTaskPage.xaml", UriKind.Relative));

}

private void deleteTaskButton\_Click(object sender, RoutedEventArgs e)

{

// Cast the parameter as a button.

var button = sender as Button;

if (button != null)

{

// Get a handle for the to-do item bound to the button.

ToDoItem toDoForDelete = button.DataContext as ToDoItem;

App.ViewModel.DeleteToDoItem(toDoForDelete);

}

// Put the focus back to the main page.

this.Focus();

}

protected override void OnNavigatedFrom(System.Windows.Navigation.NavigationEventArgs e)

{

// Save changes to the database.

App.ViewModel.SaveChangesToDB();

}

### 调试应用程序

按F5运行应用程序，或者点击Start Debugging按钮运行，如图6-7 Start Debugging。



图6-7 Start Debugging

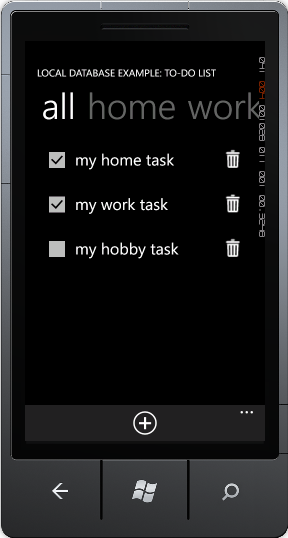
 

图6-8 运行结果