CHAPTER

5

Windows Azure 应用程序开发：Table 存储服务

本文将会介绍 Windows Azure Storage 三种服务之一：存储表服务（Table Storage Service），表服务是专门用来存储结构化数据的服务，类似关系型数据库的表格一样，您可以将您应用程序内可结构化的数据保存到表服务中。

5.1 认识表服务

表存储服务是一个模拟关系型数据库的结构化数据（structured data）访问服务，它就像是在云中的表格一样，允许应用程序在 Table 存储器中声明并访问自己的数据结构。而在 Table 存储器的内部，则是横跨多个服务器与磁盘存储区的基础架构，微软的 Windows Azure 开发小组将内核的所有操作都隐藏起来，只呈现出一组 REST API 供外部应用程序访问，而且都是通过相同的 URL 来调用，因此 Table 基本上并不是存储在应用程序所在的 VM，而是在 Windows Azure 内部自动规范的存储区域中。

其实表服务很像在云计算界所称的多租户技术（Multi-tenant Technology），多租户技术重视的是数据的可维护以及扩展性，而且也可以在多个应用程序同时运行时依照不同的租户标识码（Tenant Identification）来取得各自的数据，实现起来也不会太复杂。在表格服务中，存储了几个关键性的数据：分区标识码（Partition Key）、行标识码（Row Key）以及时间戳（Timestamp），这些数据不但构成了一个 Table 的基础架构，也让 Table 具有分散存储在不同服务器和不同存储区的功能。Table 存储器是利用单一 URL 入口来判断要读取哪个账户的数据，然后利用 URL 来设置要访问哪些数据，就算是同一个账户，不同的数据可能会被分散存放在相同数据中心的不同机房的不同机柜中的某个磁盘驱动器中。

NOTE

Table 存储器的入口是 https://[accountname].table.core.windows.net。

为了简化在这么复杂的环境中取出表服务中数据的能力，Table Storage Service 的开发小组使出了浑身解数，把复杂的资源搜索以及定位和访问的调用都包装在 WCF Data Service Framework 的 Data Service Provider 中，将访问的功能呈现给 WCF Data Service 这个对象模型，外部开发人员只要关注 WCF Data Service 即可，不必在乎隐藏在它背后的复杂数据处理流程，那些事由 Table Storage 开发小组去伤脑筋就好了，我们只需要快乐地使用 WCF Data Service 就能访问表服务。

您可以将 Partition Key 当作表的名称（当然有其他的用法，不一定要将它当表名称），而 Row Key 则是在表中的行主键（Primary Key），这两个数据构成了表格中独一无二的数据行，而其他的属性基本上就是跟随这些数据行而存在的。在表服务存储器中，一个数据行被称为一个数据实体（Entity），一个数据实体会由 Table 存储器的三个必要数据（Partition Key、Row Key 以及 Timestamp）和开发人员自定义的结构数据组成，而且必须要由开发人员在访问之前先声明它的结构，例如下列的 XML 数据：

```[XML]
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
  <title />
  <updated>2008-09-18T23:46:19.3857256Z<updated/>
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
      <d:Address>Mountain View</d:Address>
      <d:Age m:type="Edm.Int32">23</d:Age>
      <d:AmountDue m:type="Edm.Double">200.23</d:AmountDue>
      <d:BinaryData m:type="Edm.Binary" m:null="true" />
      <d:CustomerCode m:type="Edm.Guid">c9da6455-213d-42c9-9a79-3e9149a57833
          </d:CustomerCode>
      <d:CustomerSince m:type="Edm.DateTime">2008-07-10T00:00:00</d:CustomerSince>
      <d:IsActive m:type="Edm.Boolean">true</d:IsActive>
      <d:NumOfOrders m:type="Edm.Int64">255</d:NumOfOrders>
      <d:PartitionKey>mypartitionkey</d:PartitionKey>
      <d:RowKey>myrowkey1</d:RowKey>
      <d:Timestamp m:type="Edm.DateTime">0001-01-01T00:00:00</d:Timestamp>
    </m:properties>
  </content>
</entry>
```

其中在<content>中的就是 Entity 的数据内容，包含了它的数据类型、字段名称以及数据内容等信息，因此 Table 存储器与 Table 本身和数据的关系如图 5-1 所示。

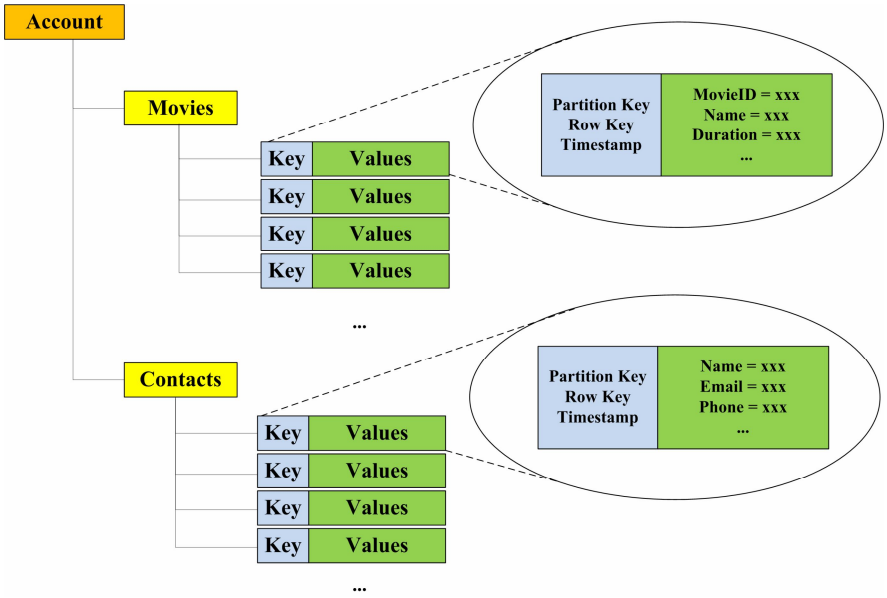


图 5-1　Table 的结构关系

由于 Table 本身并不是关系型数据库，它的限制会比关系型数据库要多很多，其中之一就是数据类型的限制，在一般的关系型数据库中可以支持十几种类型，但是在 Table 存储器中只能支持八种数据类型，如表 5-1 所示。

表 5-1　Table 存储器支持的数据类型

Data Services 类型	CLR 类型	说明
Edm.Binary	byte[]	字节数组，最长 64KB
Edm.Boolean	bool	布尔值（true/false）
Edm.DateTime	DateTime	以 64 位数值为主的 UTC 时间，可支持 1601/1/1 00:00:00～9999/12/31 23:59:59 之间的日期时间数据值
Edm.Double	double	64 位浮点数
Edm.Guid	Guid	128 位 GUID 值
Edm.Int32	Int32 or int	32 位整数
Edm.Int64	Int64 or long	64 位整数
Edm.String	String	以 UTF-16 编码的字符串数据，最长 64KB

另外一个限制是，它无法像关系型数据库一样使用 JOIN 运算（JOIN operator），JOIN 运算在数据库中是十分重要的运算符，可以将两个或多个表格结合在一起执行运算，Table 存储器无法执行 JOIN 运算，这也代表着开发人员如果要在 Table 存储器上放置统计信息的话，在聚合运算的性能会有很显著的差异。

第三个限制是，由于数据都必须要转换成 XML 才可以提交给 Table 存储器，因此在数据的处理上都必须要兼容于 XML（也就是可以存储到 XML 的数据才可以），而且为了内部要编制索引或是需要将数据最小化以符合传输与访问的性能需求，表格的名称必须是 3-63 个字符、可以只包含英文字母与数字，但第一个字母不可以是数字，且名称不区分大小写。

NOTE

Table 存储器会使用正则表达式：^[A-Za-z][A-Za-z0-9]*来检查表格名称。

除了表格自己的限制外，Table 存储器也针对内部的字段名称和数据做了限制，像是：

1. 字段名称要依照 C#的 identifier（标识符）的规范来命名，最长 255 字符，若 XML 规范未支持的 C#规范也不可以使用。
2. 字段数最多 252 个（不含系统必要的 3 个），且整个数据行的数据长度不可以超过 1MB。
3. 系统字段 Timestamp 不可以修改（若在 POST 和 UPDATE 时加入 Timestamp 的话，系统会略过）。
4. 不可以使用“\”、“/”、“#”以及“?”四种字符。
5. 每次的新建、修改与删除操作，都必须要指定 Partition Key 以及 Row Key。

6. Partition Key 和 Row Key 长度最大 1KB，同时在相同的 Partition Key 中，Row Key 对于每个数据行都必须是唯一的值。
7. 所有字段的数据类型都必须要符合表 5-1 的规范。

因此，表服务存储器很适用于在一般网站上的资源索引文件（Resource Index File），像是某个用户将自己的影音数据存储的位置或是某场电影的座位信息亦或是简单的日历或电话簿等等，这种简单又不需要汇总的结构式数据，就是表服务存储器最能发挥实力的需求。

INFO

在业界有一个最近才创出的名词：NOSQL（Not Only SQL），这个名词最早是 1998 年提出的开源关系型数据库系统的一个项目所使用，在 2009 年又由 Rackspace 的员工 Eric Evans 再度提出，它是一种分布式的数据存储方式，它的结构和运作方式与关系型数据库类似，它拥有 Key 和 Value 的结构，很适合用作分布式的结构化数据存储，Windows Azure 的表存储服务也有 NOSQL 的影子。

5.2 WCF Data Service 简介

在本章（以及接下来的第 6 章至第 8 章）中，我们会看到很多 WCF Data Service 的影子，因此笔者在开始进行表服务的介绍前，先来介绍一下这个作为存储服务的 REST 平台组件。

早期在开发 Web 应用程序时，假如要将数据发布到网络上时，HTTP 服务（例如 Web Service）的开发是免不了的。Web Service 被称为 Web 化的可编程组件（Programmable Component），不论是 Web 或是 Windows 应用程序，都能够通过 HTTP 通信协议与 SOAP 数据协议来访问与交换数据，利用 Microsoft .NET 以及 Web Service Enhancement（WSE）工具开发的 Web Service，更可轻松地享受到 WS-I 组织所制订的 Web Service 标准与安全规范，Visual Studio 也提供了相当丰富的客户端支持，让客户端应用程序能够用最简单的方式来访问 Web Service 上的资源。虽然一切看起来如此的美好，但是应用程序架构的演化与客户百变的需求总是会让我们了解事实的残酷…☺。

在 2006 年时，AJAX 开始在 Web 应用程序平台界发酵，不但微软专为 ASP.NET 开发 AJAX Framework，更多已经在风行的 JavaScript Framework（例如 jQuery）也拜 AJAX 流行之赐，在 Web 应用程序开始流行一股 Client Scripting 风潮，许多 Web 应用程序都以 AJAX 的免重整浏览器即可更新画面内容作为强推的诉求，而这也是 AJAX 应用最大的特色之一，它让一般的 Web 应用程序在浏览器上更具亲和力，同时它也是现阶段唯一可在不借助外部组件（如 Flash）的支持下和服务器端直接通信的方法，所以 AJAX 逐渐受到 Web 开发人员的喜爱。

NOTE

AJAX 技术其实最早是在微软的 Internet Explorer 4.0 出现，它是利用 MSXML 组件内的 XMLHTTP 对象由 JavaScript 控制进行数据的通信，微软也用了这个技术开发了 Outlook Web Access，堪称是 AJAX 技术第一个成功的商用应用程序，但它却是到了 2006 年才逐渐受到 Web 开发人员的青睐。现在许多主流浏览器（包括 IE7+、FireFox、Chrome 等）都内置了原生的 XMLHTTP 支持。

AJAX 以及最近新兴的 Rich Internet Application（RIA）架构，使得传统以 Web Service 供应数据的架构受到很大的冲击，传统的 Web Service 由于要背负许多通信与数据协议（如 WSDL 与 SOAP），使得它在数据供应上的弹性受到极大限制，其中之一就是 Web Service 的数据输出口必须要是独立的方法（例如 XML Web Service 上的 WebMethod，或是 WCF Service 上的 Operation Contract 等），这代表客户端只能遵从由 Web Service 公开的方法来访问数据，而不能像是使用 SQL 或 ADO.NET 一样可由客户端依需求来向服务器提交要求，这对像 Silverlight 或是 Flash 等可将客户端逻辑实现在浏览器沙箱内应用程序的技术来说，Web Service 等于是变相的限制 AJAX 或 RIA 应用程序的发展性（见图 5-2）。

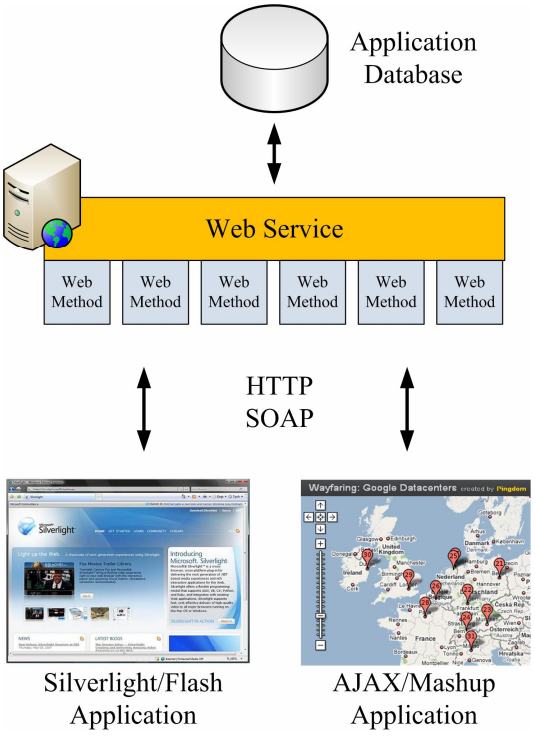


图 5-2　Web Service 应用程序架构

NOTE

Mashup（混搭）应用程序是一种放在 Web 网页内，可以独立运行的应用程序，它通常是很小规模的 Flash 或 AJAX 应用程序，最典型的就是 iGoogle 内的 Widget 应用程序。

在日渐壮大的 RIA 与 AJAX 阵营的压力之下，许多平台供货商也开始思考要如何简化数据供应层的部分，让 RIA 与 AJAX 应用程序得以发挥它们的特性，并且还要将数据在网络上传输的大小尽量缩小，数据供应层还需要尽可能在安全的情况下，将数据呈现在网络上，由 RIA 与 AJAX 应用程序来决定要取用的数据，这代表要求数据的条件是由客户端决定。微软也看到了这个需求，因此在开发新一代的 ADO.NET Entity Framework 时，启动了一个新的项目，代号为 Astoria，它作为 ADO.NET Entity Framework 的 Entity Data Model 呈现（expose）在网络上的一个接口（如图 5-3 所示），使用的协议为 HTTP 与 REST 风格的接口，输出的方式为 XML（ATOM）与 JSON 格式，符合大多数轻量化 Web 应用程序的数据访问需求。Astoria 后来更名为 ADO.NET Data Services，与 ADO.NET Entity Framework 一起包装在.NET Framework 3.5 SP1 一起发布。

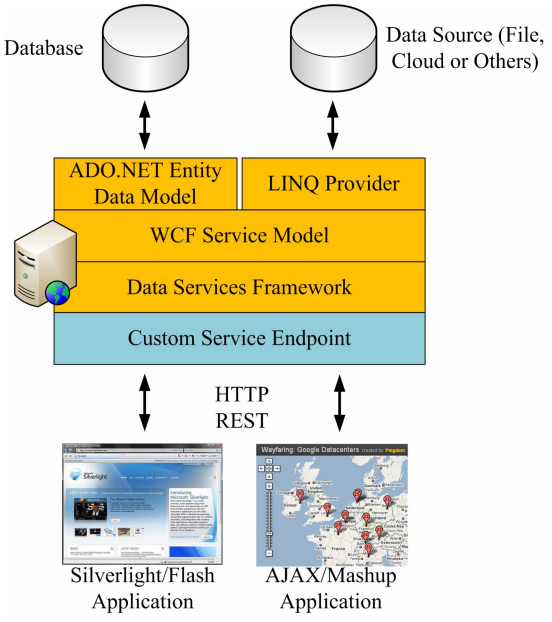


图 5-3　ADO.NET Data Service（WCF Data Services）架构

ADO.NET Data Services 是以 WCF 为基础开发的，它是一个很简单的应用程序平台，开发人员用简单的几个步骤（配合 Visual Studio 内置的 Data Service Endpoint 模板）就可以将数据库开放给网络上的应用程序，而且具有 REST API 风格以及 XML 或 JSON 数据格式的供应能力。下列程序代码就是使用 ADO.NET Data Services 开放数据给 HTTP 客户端的服务端程序代码：

```
[C#]
using System;
using System.Collections.Generic;
using System.Data.Services;
using System.Data.Services.Common;
using System.Linq;
using System.Linq.Expressions;
using System.ServiceModel.Web;
using System.Web;
```

```
namespace DataServiceClientSample
{
    public class NorthwindDataServices : DataService<NorthwindEntities>
    {
        public static void InitializeService(IDataServiceConfiguration config)
        {
            config.SetEntitySetAccessRule("*", EntitySetRights.All);
            config.SetServiceOperationAccessRule(
                "GetCategories", ServiceOperationRights.All);
            config.SetServiceOperationAccessRule(
                "GetCategoryByName", ServiceOperationRights.All);
        }

        protected override void HandleException(HandleExceptionArgs args)
        {
            args.UseVerboseErrors = true;
            //args.Exception = new Exception("My Exception");
            base.HandleException(args);
        }

        [WebGet]
        public IQueryable<Categories> GetCategories()
        {
            return this.CurrentDataSource.Categories;
        }

        [WebGet]
        public IQueryable<Categories> GetCategoryByName(string CategoryName)
        {
            return from c in this.CurrentDataSource.Categories
                   where c.CategoryName == CategoryName
                   select c;
        }
    }
}
```

就几行简单的程序代码，就可以很容易地将数据呈现给 HTTP 客户端，而且服务器端的开发人员也可以善用如 Query Interceptor 来自定义查询规则，或是进一步自定义 Service Operation（服务方法）来改变呈现数据到 HTTP 的方式。客户端可以利用 HTTP 协议（GET/POST/PUT/DELETE 动词）以及 REST URL 来访问数据，就连使用浏览器输入的方式也可以得到数据，.NET 开发人员更可以使用 Client Library，使用 LINQ 来访问 Data Services，这也是表格服务可以应用的一种访问方法。下列程序代码就是在 Silverlight 应用程序中使用 LINQ 访问 Data Services 的范例。

```
[C#]
private void lstEmployeeList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
```

```
NorthwindEntities db = new NorthwindEntities(new Uri("http://localhost/
    NorthwindDataServices.svc/"));

this.labelMessage.Text = "Loading...";

try
{
    var query = from c in db.Employees
                where c.EmployeeID == Convert.ToInt32((this.lstEmployeeList.
                    SelectedItem as TextBlock).Tag.ToString())
                select c;

    AsyncCallback ac = asyncResult =>
        {
            if (asyncResult.IsCompleted)
            {
                try
                {
                    db = asyncResult.AsyncState as NorthwindEntities;

                    var queryResult = (query as DataServiceQuery<NorthwindModel.
                        Employees>).EndExecute(asyncResult);
                    byte[] imageData = queryResult.First().PhotoJPG;

                    BitmapImage image = new BitmapImage();
                    MemoryStream ms = new MemoryStream(imageData);
                    image.SetSource(ms);

                    ImageBrush imgBrush = new ImageBrush();
                    imgBrush.ImageSource = image;
                    imgBrush.Stretch = Stretch.Fill;

                    this.imageBox.Fill = imgBrush;
                    this.imageBox.UpdateLayout();

                    db = null;
                    this.labelMessage.Text = "Please select a employee to view photo.";
                }
                catch (Exception ex2)
                {
                    this.labelMessage.Text = ex2.Message;
                }
            }
        };

    (query as DataServiceQuery<NorthwindModel.Employees>).BeginExecute(ac, db);
}
```

```
    catch (Exception ex)
    {
        this.labelMessage.Text = ex.Message;
    }
}
```

NOTE

您可以参考 MSDN 的 WCF Data Service 的帮助文档，了解 WCF Data Service 的功能，笔者就不在此详细说明。

网址：http://msdn.microsoft.com/en-us/library/cc668792.aspx。

随后在 Visual Studio 2010 与.NET Framework 4.0 开发的时期，ADO.NET Data Service 也列为重点更新项目，并且添加了几个开发人员需要的功能，例如查询（Query Projection，$select 选项的支持）、数量输出（$count 选项的支持）、服务器端分页（Server-Driven Paging）、二进制数据串行化等等新的功能，同时配合 Open Data Protocol（OData）协议的发表，ADO.NET Data Service 也更新为 WCF Data Service，它也是首批支持 OData 协议的应用程序开发平台之一。

NOTE

OData 协议是一个在 Web 上交换数据的协议，它定义了通过 HTTP 消费数据的格式与方法，REST 是 OData 钦定的要求方式，WCF Data Services 可完全支持 OData 的协议。详细的协议文件可参考：http://www.odata.org。

5.3 开发表服务应用程序

表服务基本上是由 HTTP 协议，以及 REST API 所组成，兼容于所有可以通过 HTTP 传输 XML 的客户端应用程序，而应用程序可以是在云内的应用程序，或是在云外的 Console 应用程序或是 Silverlight 应用程序等，只要遵循表服务开放的 API 以及验证授权规范，即可成功访问表服务内的数据。而.NET 开发人员比其他平台的开发人员更幸福，因为微软把这些 REST API 包装起来，以.NET Client Library 对象模型呈现给.NET 的开发人员使用。

NOTE

Table Service 对外开放的功能，基本上是以 WCF Data Services 为基础开发的，因此不论是它的定义方式，以及访问它的方法（REST API 或.NET 函数库）都与 WCF Data Service 有很多类似的地方。其实不只是 Table Service，就连 BLOB 和 Queue 也都是与 WCF Data Service 有很深的关系。

若要开发表服务的应用程序，您必须要先做好两项准备工作：

❶ 在云的角色项目内添加连接存储服务的字符串，您可以在云角色项目属性中，添加一个新的连接字符串，并且指定使用开发环境模拟的存储区（如图 5-4 所示）。

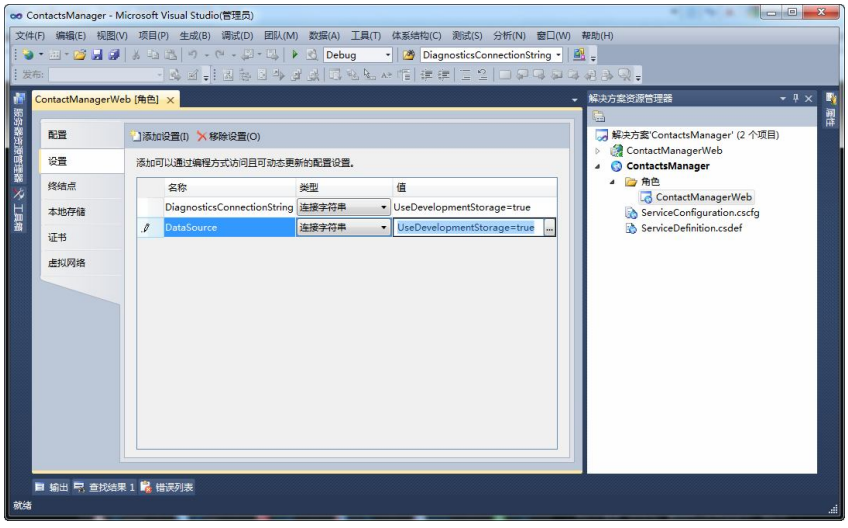


图 5-4　云角色项目设置

❷ 在要使用 Windows Azure 存储功能的类中，添加 Microsoft.WindowsAzure.StorageClient 组件的引用。

NOTE

您可能会觉得奇怪，原本就有一个 DiagnosticsConnectionString，为何不直接使用？因为 Windows Azure 的存储服务，限定一个应用程序只能用一组连接字符串，在 4.5 节中笔者也提到了，Windows Azure 的 Diagnostics 是由外部的一个独立进程使用，因此云项目不可以再使用同一组连接字符串。

接着，您必须要告诉表服务您的表的长相（数据结构），因为表服务无法默认和推断您的表结构，您需要让表服务知道，才可以顺利地将数据存储到表服务的存储区内。因此您会需要创建一个新的类文件，并且添加类似下面的程序代码：

```
[C#]
public class Contact : TableServiceEntity
{
    public string Name { get; set; }
    public string Address { get; set; }
    public string Phone { get; set; }
    public string Cellphone { get; set; }

    public Contact()
    {
        base.PartitionKey = "Contact";
        base.RowKey = Guid.NewGuid().ToString();
    }
```

```
    public Contact(string Name, string Address, string Phone, string Cellphone)
    {
        base.PartitionKey = "Contact";
        base.RowKey = Guid.NewGuid().ToString();
        this.Name = Name;
        this.Address = Address;
        this.Phone = Phone;
        this.Cellphone = Cellphone;
    }

    public Contact(string ContactID)
    {
        CloudStorageAccount storageAccount =
            CloudStorageAccount.FromConfigurationSetting("DataSource");
        CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
        TableServiceContext context = tableClient.GetDataServiceContext();

        var queryItem = from contacts in context.CreateQuery<Contact>("Contacts")
                        where contacts.RowKey == ContactID
                        select contacts;

        if (queryItem.First<Contact>() == null)
        {
            storageAccount = null;
            tableClient = null;
            throw new ArgumentException("系统无法找到指定代码的联系人数据。");
        }

        this.PartitionKey = queryItem.First<Contact>().PartitionKey;
        this.RowKey = queryItem.First<Contact>().RowKey;
        this.Address = queryItem.First<Contact>().Address;
        this.Name = queryItem.First<Contact>().Name;
        this.Phone = queryItem.First<Contact>().Phone;
        this.Cellphone = queryItem.First<Contact>().Cellphone;
        this.Timestamp = queryItem.First<Contact>().Timestamp;

        context = null;
        storageAccount = null;
        tableClient = null;
    }
}
```

在每个要登录到表服务的表结构，都必须要包含 PartitionKey、RowKey 以及 Timestamp 三个属性（字段），这三个属性会由表服务的内核管理，分别作为数据存储拆分判断标识的 PartitionKey；标识同一分区内数据行的 RowKey；以及标识数据行编辑时间的 Timestamp。而除了 Timestamp 由系统自行控制外，开发人员必须在写入数据前设置 PartitionKey 以及 RowKey 两个属性值。这两个属性值关系着数据分散在存储区内的设置，也可以作为表服务扩展的基础数据，这个会在 5.4 节说明。

表数据的访问方式有两种，一种是直接访问（Direct Access），另一种是使用表格数据模型（Table Data Model）的方式访问，两个其实基础都一样，但只是做法不太相同而已，笔者先介绍直接访问的模式。

5.3.1 直接访问表服务的开发方法

我们以一个简单的联系人应用程序（Contact List Application）来示范如何开发一个表服务应用程序。这个程序有一个简单的列表页以及窗体页，可用来存储日常的联系人数据。

01 STEP 我们先在 Visual Studio 内置创建一个云项目，名称为 ContactManager，并且内含一个 ContactManagerWeb 的 Web 角色项目（创建过程请参考 4.3 节）。创建完成时您的项目结构应如图 5-5 所示。

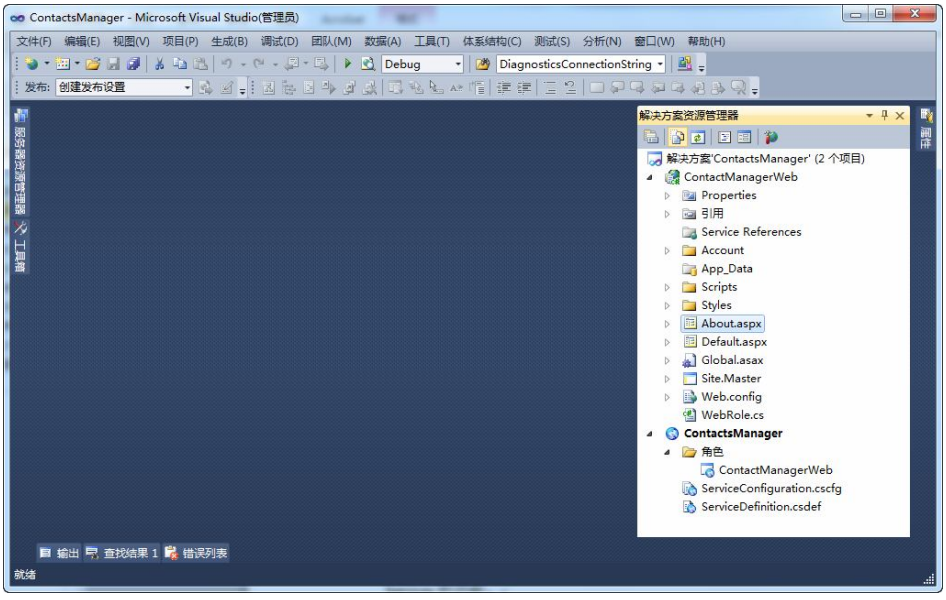


图 5-5 联系人管理应用程序项目结构

02 STEP 新建完成以后，请在 ContactManagerWeb 项目的引用中添加 System.Data. Services.dll 组件的引用，以在项目中使用 WCF Data Services 的功能。

NOTE

Windows Azure SDK 内的 Microsoft.WindowsAzure.StorageClient.dll 组件需要 System. Data.Services.dll 的引用，否则在编译时会发生找不到 DataServiceContext 类的错误。

03 STEP 新建一个 Contact 类，并且在里面添加下列程序代码：

```
[C#]
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Web;
using System.Data.Services;
using System.Data.Services.Client;
using Microsoft.WindowsAzure;
using Microsoft.WindowsAzure.StorageClient;

namespace ContactManagerWeb
{
    public class Contact : TableServiceEntity
    {
        public string Name { get; set; }
        public string Address { get; set; }
        public string Phone { get; set; }
        public string Cellphone { get; set; }

        public Contact()
        {
            base.PartitionKey = "Contact";
            base.RowKey = Guid.NewGuid().ToString();
        }

        public Contact(string Name, string Address, string Phone, string Cellphone)
        {
            base.PartitionKey = "Contact";
            base.RowKey = Guid.NewGuid().ToString();
            this.Name = Name;
            this.Address = Address;
            this.Phone = Phone;
            this.Cellphone = Cellphone;
        }

        public Contact(string ContactID)
        {
            CloudStorageAccount storageAccount = CloudStorageAccount.
                FromConfigurationSetting("DataSource");
            CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
            TableServiceContext context = tableClient.GetDataServiceContext();

            var queryItem = from contacts in context.CreateQuery<Contact>("Contacts")
                            where contacts.RowKey == ContactID
                            select contacts;

            if (queryItem.First<Contact>() == null)
            {
                storageAccount = null;
                tableClient = null;
                throw new ArgumentException("系统无法找到指定代码的联系人数据。");
            }

            this.PartitionKey = queryItem.First<Contact>().PartitionKey;
            this.RowKey = queryItem.First<Contact>().RowKey;
```

```
            this.Address = queryItem.First<Contact>().Address;
            this.Name = queryItem.First<Contact>().Name;
            this.Phone = queryItem.First<Contact>().Phone;
            this.Cellphone = queryItem.First<Contact>().Cellphone;
            this.Timestamp = queryItem.First<Contact>().Timestamp;

            context = null;
            storageAccount = null;
            tableClient = null;
        }
    }
}
```

04 STEP 请新增一个 ContactManager_TableDirect.aspx 文件（使用“使用母版页的 Web Form”模板，并设置使用 Site.Master 母版页），将它设为启动页，并且使用下列的命令将原有 BodyContent 的内容替换掉。

```
[HTML]
<asp:MultiView ID="mvContactManager" runat="server" ActiveViewIndex="0">
    <asp:View ID="vContactViewer" runat="server">
        <asp:LinkButton ID="cmdAddNewContact" runat="server" Text="新增联系人..."
            onclick="cmdAddNewContact_Click" /><br /><br />
        <asp:GridView ID="gvContactViewer" runat="server" CellPadding="4"
            ForeColor="#333333"
            GridLines="None" AutoGenerateColumns="false"
            onrowcommand="gvContactViewer_RowCommand">
            <AlternatingRowStyle BackColor="White" ForeColor="#284775" />
            <EditRowStyle BackColor="#999999" />
            <FooterStyle BackColor="#5D7B9D" Font-Bold="True" ForeColor="White" />
            <HeaderStyle BackColor="#5D7B9D" Font-Bold="True" ForeColor="White" />
            <PagerStyle BackColor="#284775" ForeColor="White" HorizontalAlign="Center" />
            <RowStyle BackColor="#F7F6F3" ForeColor="#333333" />
            <SelectedRowStyle BackColor="#E2DED6" Font-Bold="True" ForeColor="#333333" />
            <SortedAscendingCellStyle BackColor="#E9E7E2" />
            <SortedAscendingHeaderStyle BackColor="#506C8C" />
            <SortedDescendingCellStyle BackColor="#FFFDF8" />
            <SortedDescendingHeaderStyle BackColor="#6F8DAE" />
            <EmptyDataTemplate>
            您尚未定义任何联系人数据。
            </EmptyDataTemplate>
            <Columns>
               <asp:BoundField DataField="Name" HeaderText="姓名" ItemStyle-Width="150px" />
               <asp:BoundField DataField="Phone" HeaderText="电话" ItemStyle-Width="150px" />
               <asp:BoundField DataField="Cellphone" HeaderText="手机" ItemStyle-Width
                    ="150px" />
                <asp:BoundField DataField="Address" HeaderText="住址" ItemStyle-Width
                    ="250px" />
                <asp:TemplateField HeaderText="命令" ItemStyle-Width="100px">
                    <ItemStyle HorizontalAlign="Center" />
                    <ItemTemplate>
```

```
                    <asp:LinkButton ID="cmdEditContact" runat="server"
                    CommandName="EditContact" Text="编辑" CommandArgument=
                        '<%# Eval("RowKey") %>' />
                    <asp:LinkButton ID="cmdDeleteContact" runat="server" OnClientClick=
                        "return confirm('请问是否要删除？')"
                    CommandName="DeleteContact" Text="删除" CommandArgument=
                        '<%# Eval("RowKey") %>' />
                </ItemTemplate>
            </asp:TemplateField>
        </Columns>
    </asp:GridView>

</asp:View>
<asp:View ID="vContactForm" runat="server">
    <br />
    姓名：<asp:TextBox ID="txtName" runat="server"></asp:TextBox>
    <br />
    地址：<asp:TextBox ID="txtAddress" Width="400px" runat="server"></asp:TextBox>
    <br />
    电话：<asp:TextBox ID="txtPhone" MaxLength="30" runat="server"></asp:TextBox>
    <br />
    手机：<asp:TextBox ID="txtCellphone" MaxLength="10" runat="server"></asp:TextBox>
    <br />
    <br />
    <asp:Button ID="cmdOK" runat="server" Text=" 确定 " onclick="cmdOK_Click" />
      
    <asp:Button ID="cmdCancel" runat="server" Text=" 取消 "
        onclick="cmdCancel_Click" />
    <br />
</asp:View>
</asp:MultiView>
```

这些 HTML 代码组成的用户界面如图 5-6 所示。

图 5-6　联系人应用程序主画面

05 STEP 修改 ContactManagerWeb 项目的 WebRole.cs，添加下列程序代码：

```
[C#]
public override bool OnStart()
{
    DiagnosticMonitor.Start("DiagnosticsConnectionString");

    // handling role state changes.
    RoleEnvironment.Changing += RoleEnvironmentChanging;

    // transfer configuration source to Service Configuration File.
    CloudStorageAccount.SetConfigurationSettingPublisher((configName, configSetter) =>
    {
        configSetter(RoleEnvironment.GetConfigurationSettingValue(configName));
        RoleEnvironment.Changed += (anotherSender, arg) =>
        {
            if (arg.Changes.OfType<RoleEnvironmentConfigurationSettingChange>()
                .Any((change) => (change.ConfigurationSettingName == configName)))
            {
                if (!configSetter(RoleEnvironment.GetConfigurationSettingValue
                    (configName)))
                {
                    RoleEnvironment.RequestRecycle();
                }
            }
        };
    });

    return base.OnStart();
}
```

在后面的步骤中我们都会由 Service Configuration 配置文件内读取数据，因此我们必须要先设置 CloudStorageAccount.FromConfiguration SettingPublisher()将配置文件转向给 Service Configuration 配置文件，而上面的程序代码就是在处理这件事，而且它必须要在 CloudStorageAccount. FromConfigurationSetting() 被调用之前调用，否则会引发“SetConfigurationSettingPublisher needs to be called before FromConfigurationSetting can be used”错误。

06 STEP 修改 ContactManager_TableDirect.aspx 的 Page_Load 事件处理程序如下：

```
[C#]
public enum EditState
{
    None,
    Add,
    Update
}

protected void Page_Load(object sender, EventArgs e)
```

```
{
    if (!Page.IsPostBack)
    {
        ViewState.Add("EditState", EditState.None);
        ViewState.Add("EditContactID", null);

        CloudStorageAccount storageAccount =
            CloudStorageAccount.FromConfigurationSetting("DataSource");
        CloudTableClient tableClient = storageAccount.CreateCloudTableClient();

        tableClient.CreateTableIfNotExist("Contacts");

        TableServiceContext context = tableClient.GetDataServiceContext();

        this.gvContactViewer.DataSource =
              from contacts in context.CreateQuery<Contact>("Contacts")
              select contacts;
        this.gvContactViewer.DataBind();

        context = null;
        storageAccount = null;
        tableClient = null;
    }
}
```

请注意在 Page_Load 事件处理程序内的程序，首先创建 CloudStorageAccount 对象，这个对象装载了 Windows Azure Storage 的账户信息，在前面我们都一直说明在配置文件内设置存储服务的连接字符串，在这里就派上用场了。当然您也可以自行设置这些值，但是由配置文件来生成 CloudStorageAccount 对象的做法相较起来更简单。

NOTE

若您不使用 CloudStorageAccount.FromConfigurationSetting()来生成 CloudStorageAccount 对象的话，就要用下列程序代码来生成 CloudStorageAccount 对象：

```
CloudStorageAccount storageAccount = new CloudStorageAccount(
    new StorageCredentialsAccountAndKey(
        "accountname",
        Convert.FromBase64String("account key")),
    new Uri("https://accountname.blob.core.windows.net"),
    new Uri("https://accountname.queue.core.windows.net"),
    new Uri("https://accountname.table.core.windows.net"));
```

当 CloudStorageAccount 生成完成后，就可以利用它来创建 CloudTableClient 对象（使用 CloudStorageAccount.CreateCloudTable Client()延伸方法），或者可以利用下列的程序代码来生成 CloudTableClient 对象：

```
[C#]
CloudTableClient tableClient2 = new CloudTableClient(
    "https://accountname.table.core.windows.net",
```

```
    new StorageCredentialsAccountAndKey(
        "accountname", Convert.FromBase64String("account key")));
```

到现在为止，我们已经创建了与表服务的连接（connection），但我们仍然要确认表是否存在，因此在 CloudTableClient 中提供了一个 CreateTableIfNotExists()方法，您可以直接调用这个方法，它会自动判断表是否存在，如果不存在就生成表。您也可以使用 CreateTable()方法，但这样就要自行判断表是否已经存在。

接着，我们需要查询现有的联系人资料，并设置到 GridView 内以显示出来，.NET Client Library 可以支持使用 LINQ 的查询来取得数据，但前提是要由 DataServiceContext 类的派生对象来提供这个功能。在 CloudTableClient 中有一个 GetDataServiceContext()方法，会返回基础的 TableServiceContext 对象，可以利用 TableServiceContext.CreateQuery <T>(“TableName”)产生一个 TableServiceQuery<T> 对象，针对表服务查询的 LINQ 查询都由这个对象在运行时将查询提交给表服务以查询并返回数据。例如下列的程序代码：

```
[C#]
this.gvContactViewer.DataSource =
from contacts in context.CreateQuery<Contact>("Contacts")
     select contacts;
```

实际生成的程序应该是：

```
[C#]
DataServiceQuery<Contact> queryProvider = context.CreateQuery<Contact>("Contacts");
this.gvContactViewer.DataSource = from contacts in queryProvider
select contacts;
```

它会被转换成这样的 REST API 调用：

```
[URL]
http://accountname.table.core.windows.net/Contacts()（云真实环境）
http://127.0.0.1:10002/devstoreaccount1/Contacts()（Development Storage）
```

再举一个例子，若您使用的查询是：

```
[C#]
this.gvContactViewer.DataSource =
    from contacts in context.CreateQuery<Contact>("Contacts")
    where contacts.RowKey == myRowKey
    select contacts;
```

那么实际被送到表格服务的 REST API 会是：

```
[URL]
http://accountname.table.core.windows.net/Contacts()?$filter=(RowKey eq myRowKey)
（云真实环境）
http://127.0.0.1:10002/devstoreaccount1/Contacts()?$filter=(RowKey eq myRowKey)（Development Storage）
```

因此基本上可以将Microsoft.WindowsAzure.StorageClient组件内的对象视为一个中继的组件，所有的存储服务都还是会转换成 REST API 来访问，这也代表您的 LINQ 字符串不会支持所有的命令，像是 SelectMany、Max/Min、Sum/Average、Single 与 Contains 这些命令在 LINQ to Table Service 都不支持，但随着.NET 4.0 以及未来的 WCF Data Service 功能强化，也许表服务会开放更多的 LINQ 命令。

NOTE

您可以在这里查到表服务的 LINQ 支持：

http://msdn.microsoft.com/en-us/library/dd135725.aspx

07 STEP 新建 cmdAddNewContact 按钮的 Click 事件处理程序，并在 cmdAddNewContact_Click 事件处理程序中添加下列程序代码，处理画面的切换。

```
[C#]
protected void cmdAddNewContact_Click(object sender, EventArgs e)
{
    ViewState["EditState"] = EditState.Add;
    this.txtName.Text = this.txtPhone.Text = this.txtAddress.Text =
        this.txtCellphone.Text
        = string.Empty;
    this.mvContactManager.SetActiveView(this.vContactForm);
}
```

08 STEP 新建 GridView 的 RowCommand 处理程序，以处理数据的修改和删除功能，请在 gvContactViewer_RowCommand 事件处理程序中添加下列程序代码：

```
[C#]
protected void gvContactViewer_RowCommand(object sender, GridViewCommandEventArgs e)
{
    CloudStorageAccount storageAccount = CloudStorageAccount.FromConfigurationSetting
        ("DataSource");
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
    TableServiceContext context = tableClient.GetDataServiceContext();
    Contact contact = null;

    switch (e.CommandName)
    {
        case "EditContact":

            contact = new Contact(e.CommandArgument.ToString());

            this.txtName.Text = contact.Name;
            this.txtPhone.Text = contact.Phone;
            this.txtCellphone.Text = contact.Cellphone;
            this.txtAddress.Text = contact.Address;

            ViewState["EditState"] = EditState.Update;
            ViewState["EditContactID"] = e.CommandArgument.ToString();

            this.mvContactManager.SetActiveView(this.vContactForm);
```

```
            break;

        case "DeleteContact":

            var queryItem = from contacts in context.CreateQuery<Contact>("Contacts")
                            where contacts.RowKey == e.CommandArgument.ToString()
                            select contacts;

            context.DeleteObject(queryItem.First<Contact>());
            context.SaveChanges(SaveChangesOptions.Batch);

            // re-bind gridview.
            this.gvContactViewer.DataSource = from contacts in context.CreateQuery
            <Contact>
                ("Contacts")
                                              select contacts;
            this.gvContactViewer.DataBind();

            break;
    }

    context = null;
    storageAccount = null;
    tableClient = null;
}
```

此程序的重点在修改和删除，修改命令是将内容读出再切换到编辑窗体中，做法与前面读入清单的类似，在此就不赘述。删除功能则是先将要删除的数据取出来，然后使用TableServiceContext.DeleteObject()将数据标记删除，再用 TableServiceContext.SaveChanges()将数据的状态返回给表服务，它会执行实际的删除操作。您在程序中看到的SaveChangesOptions 是一个控制 SaveChanges()行为的枚举对象，在范例中使用的 SaveChangesOptions.Batch 表示将所有的更新以一个批次传到表服务进行处理（所有可用的值如表 5-2 所示）。

表 5-2　SaveChangesOptions 可用值

SaveChangesOptions 值	说明
None	默认值，以默认的行为来执行 SaveChanges 命令
Batch	将所有的更新集中到一个批处理内执行
ContinueOnError	在更新发生错误时继续处理，默认是一发生错误就会中断
ReplaceOnUpdate	更新数据时以所有值替换的方式来进行，默认是只会修改更改过的值

NOTE

在执行 UpdateObject()和 DeleteObject()时，由于 WCF Data Services 会在数据取回后对该对象进行更改跟踪（Change Tracking），因此若对象没有受到 DataServiceContext 的更改跟踪时，调用 UpdateObject()或 DeleteObject()会收到“The context is already tracking the entity”的错误信息。

09 STEP 新建 cmdOK 的 Click 事件处理程序，并添加下列程序代码在 cmdOK_Click 事件处理程序中：

```
[C#]
protected void cmdOK_Click(object sender, EventArgs e)
{
    CloudStorageAccount storageAccount = CloudStorageAccount.FromConfigurationSetting
        ("DataSource");
    CloudTableClient tableClient = storageAccount.CreateCloudTableClient();
    TableServiceContext context = tableClient.GetDataServiceContext();

    if (EditState.Add == ((EditState)ViewState["EditState"]))
    {
        context.AddObject("Contacts",
            new Contact(this.txtName.Text, this.txtAddress.Text, this.txtPhone.Text,
                this.txtCellphone.Text));
    }
    else
    {
        var queryItem = from contacts in context.CreateQuery<Contact>("Contacts")
                        where contacts.RowKey == ViewState["EditContactID"].ToString()
                        select contacts;

        Contact contact = queryItem.First<Contact>();

        contact.Address = this.txtAddress.Text;
        contact.Name = this.txtName.Text;
        contact.Phone = this.txtPhone.Text;
        contact.Cellphone = this.txtCellphone.Text;

        context.UpdateObject(contact);
    }

    context.SaveChangesWithRetries(SaveChangesOptions.Batch);

    this.gvContactViewer.DataSource = from contacts in context.CreateQuery<Contact>
        ("Contacts")
                                      select contacts;
    this.gvContactViewer.DataBind();

    this.mvContactManager.SetActiveView(this.vContactViewer);

    ViewState["EditState"] = EditState.None;
    ViewState["EditContactID"] = null;

    context = null;
    storageAccount = null;
    tableClient = null;
}
```

这段程序的主要目的是处理添加和修改联系人数据，其中 TableServiceContext.AddObject()是将数据添加到表格中，UpdateObject()则是修改数据，并且由 SaveChanges()返回表服务

处理实际的更新，但本段使用的是 SaveChangesWithRetries()方法，这个方法和 SaveChanges()有相同的能力，但是它会在无法连接时使用 RetryPolicy 所定义的方法进行重试，直到成功，或是 RetryPolicy 定义的终止条件成立为止。若您有自己的重试逻辑算法，可以自行实现这个委托，并设置给 TableServiceContext.RetryPolicy 即可。

NOTE

默认的 RetryPolicy 重试原则会使用随机指数回滚（Randomized Exponential Backoff）算法来决定重试的次数，指数回滚算法是用于预防网络堵塞的算法，可参考：
http://en.wikipedia.org/wiki/Exponential_backoff

10 STEP 新建 cmdCancel 的 Click 事件处理程序，并在 cmdCancel_Click 事件处理程序中添加下列程序，以处理画面的切换。

```
[C#]
protected void cmdCancel_Click(object sender, EventArgs e)
{
    this.txtName.Text = this.txtPhone.Text = this.txtAddress.Text =
        this.txtCellphone.Text
        = string.Empty;
    this.mvContactManager.SetActiveView(this.vContactViewer);
}
```

当上述程序完成后，请按 F5 键启动 Visual Studio 调试器以及 Windows Azure 模拟环境，您会看到图 5-7 的画面。

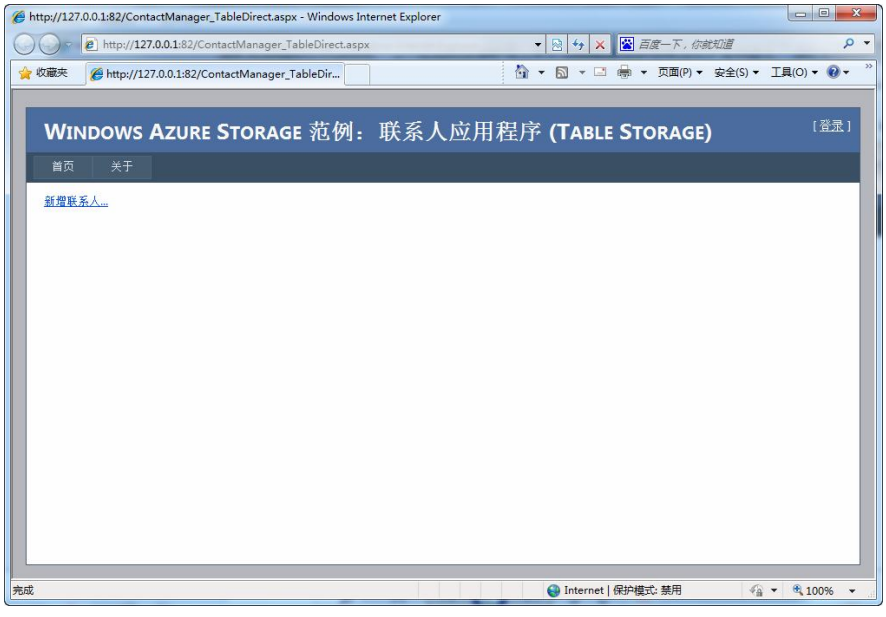


图 5-7　新增联系人

可以单击“新增联系人...”进入新建窗体，如图 5-8 所示。

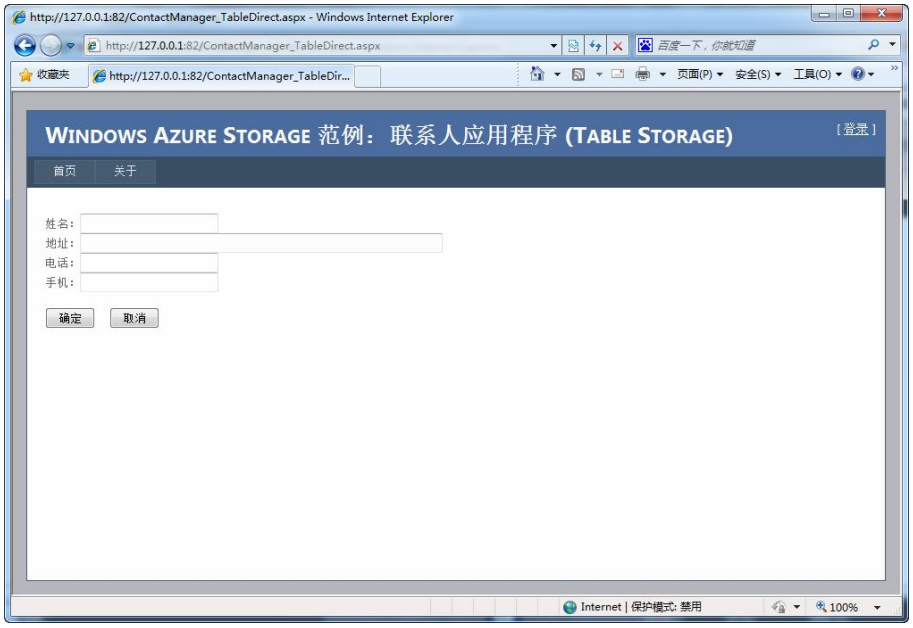


图 5-8　新建窗体

在新建窗体输入数据以后单击“确定”，即可将数据输入到表格中，如图 5-9 所示。

图 5-9　输入数据

您可以自行测试“编辑”和“删除”功能，看是否可以修改和删除数据。

NOTE

由于 Development Storage 无法处理汉字，因此您不能在测试环境中使用中文数据（会得到“Invalid Input”的错误信息），但在 Windows Azure 云环境上可以支持中文。

5.3.2 使用表数据模型方式开发

在这一节中，我们会以表数据模型的方式来开发与前一小节相同功能的联络人应用程序，让您可以实际体会使用直接访问和使用表格数据模型的差异性。

01 STEP 在前一小节创建的联系人项目 ContactManagerWeb 中，添加一个新的网页，名称为 ContactManager_TableModel.aspx，添加的方式与前一节的 ContactManager_TableDirect.aspx 相同，在新增完成后，请粘贴与前一节相同的 HTML 代码。

02 STEP 修改前一节所创建的 Contact.cs 文件，在 Contact 类下方添加下列程序代码：

```
[C#]
public class ContactDataContext : TableServiceContext
{
    public ContactDataContext() : base(
            CloudStorageAccount.FromConfigurationSetting("DataSource").
                TableEndpoint.AbsoluteUri,
            CloudStorageAccount.FromConfigurationSetting("DataSource").Credentials)
    {
    }

    public void AddContact(Contact ContactItem)
    {
        base.AddObject("Contacts", ContactItem);
        base.SaveChanges();
    }

    public void UpdateContact(Contact ContactItem)
    {
        base.UpdateObject(ContactItem);
        base.SaveChanges();
    }

    public void DeleteContact(Contact ContactItem)
    {
        base.DeleteObject(ContactItem);
        base.SaveChanges();
    }

    public IQueryable<Contact> Contacts
    {
        get { return base.CreateQuery<Contact>("Contacts"); }
    }

    public List<Contact> ListByName(string Name)
    {
        var query = from contact in base.CreateQuery<Contact>("Contacts")
                    select contact;
```

```
        List<Contact> results = new List<Contact>();

        foreach (var contact in query)
        {
            if (contact.Name.Contains(Name))
                results.Add(contact);
        }

        return results;
    }

    public Contact GetByID(string RowKey)
    {
        return (from contact in base.CreateQuery<Contact>("Contacts")
                where contact.RowKey == RowKey
                select contact).First();
    }

    public int GetContactCount()
    {
        return (from contact in base.CreateQuery<Contact>("Contacts")
                select contact).Count();
    }
}
```

这是表数据模型的核心，它继承了 TableServiceContext 的所有功能，并且定义了自己的功能，也就是说可以通过这个类来操控 Contact 表数据，不用像前一小节的直接访问方式写那么多的程序代码，我们在接下来的步骤会有很明显的感受。其中最重要的是 Contacts 属性，这个属性会在后面被调用，以定义表的名称，因此每个表数据模型都必须要具备以表名称为属性名称，且会回调 IQueryable<T>的属性。

03 STEP 修改 WebRole.cs，并在 base.OnStart()命令前添加下列的程序代码：

```
[C#]
// create table from object model.
CloudStorageAccount storageAccount = CloudStorageAccount.FromConfigurationSetting("DataSource");
CloudTableClient.CreateTablesFromModel(typeof(ContactDataContext),
    storageAccount.TableEndpoint.AbsoluteUri, storageAccount.Credentials);
```

这是表数据模型中最重要的一段，CloudTableClient. CreateTablesFromModel()会依照传入的 TableServiceContext 的类来创建表数据，它会调用在该类内具有 IQueryable<T>属性的成员，然后依照它的名称来设置表名称，您就不需要在程序中再调用 CloudTableClient.CreateTableIfNotExist()方法了。

04 STEP 修改 ContactManager_TableDirect.aspx 的 Page_Load 事件处理程序，如下列程序代码所示：

```
[C#]
public enum EditState
{
    None,
    Add,
    Update
}

protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        ViewState.Add("EditState", EditState.None);
        ViewState.Add("EditContactID", null);

        ContactDataContext context = new ContactDataContext();

        this.gvContactViewer.DataSource = from contacts in context.CreateQuery
            <Contact>("Contacts")
                                          select contacts;
        this.gvContactViewer.DataBind();

        context = null;
    }
}
```

可以比较一下这段程序和前一小节的 Page_Load 程序是否精简了很多，这也就是为什么 Windows Azure Platform Training Kit 内的文件要使用表数据模型来创建表应用程序的原因了。

05 STEP 新增 cmdAddNewContact 按钮的 Click 事件处理程序，并在 cmdAddNewContact_Click 事件处理程序中添加下列程序代码，处理画面的切换。

```
[C#]
protected void cmdAddNewContact_Click(object sender, EventArgs e)
{
    ViewState["EditState"] = EditState.Add;
    this.txtName.Text = this.txtPhone.Text = this.txtAddress.Text = this.txtCellphone.Text
        = string.Empty;
    this.mvContactManager.SetActiveView(this.vContactForm);
}
```

06 STEP 添加 GridView 的 RowCommand 处理程序，以处理数据的修改和删除功能，请在 gvContactViewer_RowCommand 事件处理程序中添加下列程序代码：

```
[C#]
protected void gvContactViewer_RowCommand(object sender, GridViewCommandEventArgs e)
{
    ContactDataContext context = new ContactDataContext();
    Contact contact = null;

    switch (e.CommandName)
    {
```

```
        case "EditContact":

            contact = new Contact(e.CommandArgument.ToString());

            this.txtName.Text = contact.Name;
            this.txtPhone.Text = contact.Phone;
            this.txtCellphone.Text = contact.Cellphone;
            this.txtAddress.Text = contact.Address;

            ViewState["EditState"] = EditState.Update;
            ViewState["EditContactID"] = e.CommandArgument.ToString();

            this.mvContactManager.SetActiveView(this.vContactForm);

            break;

        case "DeleteContact":

            var queryItem = from contacts in context.Contacts
                            where contacts.RowKey == e.CommandArgument.ToString()
                            select contacts;

            context.DeleteContact(queryItem.First<Contact>());

            // re-bind gridview.
            this.gvContactViewer.DataSource = from contacts in context.Contacts
                                              select contacts;
            this.gvContactViewer.DataBind();

            break;
    }

    context = null;
}
```

此程序的重点在修改和删除，与前一节的程序不同的是，删除功能移到了ContactDataContext 类内，程序变得简洁了不少。

07 STEP 新建 cmdOK 的 Click 事件处理程序，并添加下列程序代码在 cmdOK_Click 事件处理程序中：

```
[C#]
protected void cmdOK_Click(object sender, EventArgs e)
{
    ContactDataContext context = new ContactDataContext();

    if (EditState.Add == ((EditState)ViewState["EditState"]))
    {
        context.AddContact(
            new Contact(this.txtName.Text, this.txtAddress.Text, this.txtPhone.Text,
                this.txtCellphone.Text));
    }
    else
    {
```

```
        var queryItem = from contacts in context.CreateQuery<Contact>("Contacts")
                        where contacts.RowKey == ViewState["EditContactID"].ToString()
                        select contacts;

        Contact contact = queryItem.First<Contact>();

        contact.Address = this.txtAddress.Text;
        contact.Name = this.txtName.Text;
        contact.Phone = this.txtPhone.Text;
        contact.Cellphone = this.txtCellphone.Text;

        context.UpdateContact(contact);
    }


    this.gvContactViewer.DataSource = from contacts in context.CreateQuery<Contact>
        ("Contacts")
                                      select contacts;
    this.gvContactViewer.DataBind();

    this.mvContactManager.SetActiveView(this.vContactViewer);

    ViewState["EditState"] = EditState.None;
    ViewState["EditContactID"] = null;

    context = null;
}
```

这段程序的主要目的是处理添加和修改联系人数据，与前一节不同的是，实际处理更新数据的程序移到 ContactDataContext 类处理，如此前端用户界面无须再处理这个部分，这也是很多开发模式采用的数据处理逻辑和用户界面分离的原则。

08 STEP 新建 cmdCancel 的 Click 事件处理程序，并在 cmdCancel_Click 事件处理程序中添加下列程序，以处理画面的切换。

```
[C#]
protected void cmdCancel_Click(object sender, EventArgs e)
{
    this.txtName.Text = this.txtPhone.Text = this.txtAddress.Text = this.txtCellphone.Text
        = string.Empty;
    this.mvContactManager.SetActiveView(this.vContactViewer);
}
```

程序编辑完成后，您可以按下 F5 键启动 Visual Studio，并浏览 ContactManager_TableModel.aspx，其操作模式与 ContactManager_TableDirect.aspx 相同，功能也完全相同。

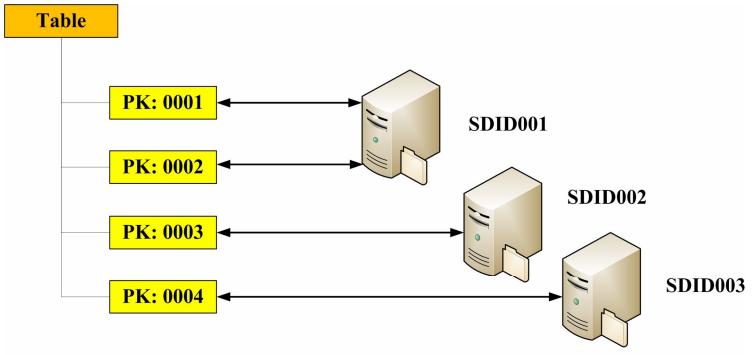
5.4 表的自动化扩展：谈 PartitionKey 的用途

在本章的好几个地方都有提到每个 Entity（数据行）都要有一个 Partition Key，但您

知道 Windows Azure 为什么要定义这个 Partition Key 吗？

可以回顾一下 3.3.1 节“Windows Azure Storage 存储服务”以及 2.2 节“海量分布式数据处理”，在一个大型数据中心提供的云平台，都有用 SLA 来规范服务的可用性，为了预防可用性低于 SLA 的规定，云平台势必要进行分布式且冗余的数据存储，表服务就具有这样的能力，不但是分散存储，而且还可以依某个特定的标识值将相同标识的数据存储在相同的磁盘区中，以保障针对该标识值的查询作业可以很快的完成，这个标识值就是 PartitionKey。

有了 PartitionKey，即便是在相同的表，数据实际的存储地点可能也会在不同的服务器内，如图 5-10 所示。



AccountName	TableaName	PartitionKey	StorageID
100102	Products	0001	SDID001
100102	Products	0002	SDID001
100102	Products	0003	SDID002
100102	Products	0004	SDID003

图 5-10　Partition Key 分布式存储架构

分布式的表存储不但可以有助于表服务的 HA（高可用性），也可以帮助表服务的索引服务器分散数据的存储地，相同表但异地存储的机制，可以确保在搜索相同 PartitionKey 数据的情况下，有效地提升表服务的核心执行访问 I/O 的效率，这个机制称为 Entity Locality（实体本地化），和操作系统的 Locality of Reference 的功能差不多，简单来说就是将要搜索的标的范围集中，通过扫描的范围缩小来加快搜索的速度。

NOTE

想更了解 Locality of Reference 机制的读者，可参考操作系统的教科书，或是维基百科的说明：http://en.wikipedia.org/wiki/Locality_of_reference。

5.5 表服务设计建议

表服务虽然可以辅助应用程序存储结构化数据，但就像我们使用关系型数据库一样，适当的查询以及数据结构可以增加数据库的读写效率，同样的，表服务基于架构上的限

制，也需要一些设计上的考虑，才能够让表服务的访问效率较高，而且成本会比较低。

1. 适当使用 PartitionKey 来拆分表格内容

我们在 5.4 节说明了 PartitionKey 的用意，Windows Azure 的表服务核心会利用 PartitionKey 作为索引值的其中一个，它也是分散数据到不同的存储器的主要标识码，就算是同一个表的数据，如果 PartitionKey 不同的话，表格核心程序也会将这些数据分散到不同的存储器来存储，这和平时使用的关系型数据库不同，也因为如此，适当的 PartitionKey 设置是很重要的。我们接下来的讨论，很多都和 PartitionKey 的设置有关系。

2. 预防不当的 Append-Only 写入行为

所谓的 Append-Only，是指将数据写入数据流的后端，只会新增数据的行为。在表格服务中，Append-Only 行为是指每笔数据的 PartitionKey 都是不同的，这样虽然方便，但在查询（尤其是清单型查询）的时候会导致查询要扫过所有的表格数据，这代表着查询处理器要读遍所有的表格数据（这称为全表格扫描），速度一定会比只扫描一个区域来得慢。例如，用 UTC 时间当 PartitionKey 就是一种 Append-Only 的声明法，在每次访问这个表时用的都会是全表扫描，当数据量够大时，速度就会变得很慢。所以使用表服务时，最好不要在 PartitionKey 中放不同的值（除非要做 Grouping），否则查询速度一定会受到影响。

3. 在大型表格中，预防全表查询行为

在前面已经提到尽量不要使用 Append-Only 的设置方式，同样的，如果表中散落过多的 PartitionKey，导致您的查询必须要跨越不同的 PartitionKey 时，您可能就要思考一下是否要重新设计数据结构了，数据结构的设计和查询应尽量避免跨越 PartitionKey，否则会很容易发生全表扫描，导致查询速度会变得很慢。

对表服务来说，最好也最快的查询行为是有指定 PartitionKey 与 RowKey 的查询，可以接受的查询则是 PartitionKey 固定，RowKey 不同，这种条件会让表服务的实体本地化特性发挥作用，可以缩小查询的范围。但表 5-3 的查询则最好不要使用。

表 5-3　表格查询方式的可能副作用

查询命令	可能的问题
PartitionKey == "SciFi" and "Sphere"≤RowKey≤"Star Wars"	问题较低，因为 PartitionKey 是固定的，在表服务的实体本地化（Entity Locality）特性，扫描的速度不会受到太大影响
"Action"≤PartitionKey≤"Thriller"	PartitionKey 不固定，会在表格索引上产生较多的扫描负载，但负载不会太高（范围查询）
PartitionKey == "Action" ‖ PartitionKey == "Thriller"	PartitionKey 不固定，会在表格索引上产生较多的扫描负载，但负载会较高（指定值查询），在现阶段会引发全表扫描
"Cars"≤RowKey≤"Star Wars"	最差的查询法，此种查询一定会引发全表扫描，同时每个 PartitionKey 的数据列都会被扫描比较过

4. 使用批次更新

我们在 3.5 节说明存储服务的交易计费时有特别提到，如果在表服务更新大量数据时，

若没有使用批次更新，则交易次数会依更新的笔数为主（例如更新 100 笔，就算是 100 个交易），但若使用实体组事务（Entity Group Transaction）的话，不论多少笔都只会算一个事务数，而且实体组交易不只是更新操作才能用，就连查询也可以用，若查询需要执行多个 HTTP 请求时，也可以应用实体组事务。但实体组事务必须要满足几个条件：

- 参与更新的数据行都有同一个 PartitionKey。
- 每笔数据在事务中只会执行一次，且更新的操作也只有一次。
- 事务数量不超过 100 笔，且 100 笔的数据总长度不超过 4MB。
- 所有的数据行都满足表格存储服务的要求。

若表格满足了上述要求后，以.NET Client Library 执行 Entity Group Transaction 查询时，只要简单的使用下列命令：

```
[C#]
var queryItem = (from contacts in context.CreateQuery<Contact>("Contacts")
                 where contacts.RowKey == ViewState["EditContactID"].ToString()
                 select contacts).AsTableServiceQuery();
```

若要执行更新，则是使用下列命令：

```
[C#]
context.SaveChangesWithRetries(SaveChangesOptions.Batch);
```

NOTE

您可以参考 Windows Azure SDK 的说明，以取得使用 REST API 执行实体组事务的方法。

5. 处理新建时的冲突（Conflict）以及删除时的对象不存在（NotFound）问题

当您使用 SaveChangesWithRetries()执行更新时，如果发生新增对象时对象已存在，或是删除数据时数据已经不存在的状况时，SaveChangesWithRetries()仍然会依指数后退的方式重试，此时会有浪费交易次数的情况，因此假如预期可能会有这种情况发生时，请将 TableServiceContext 的 RetryPolicy 属性设为 RetryPolicies.NoRetry 以防止自动的重试行为。

5.6 结语

本章带您认识 Windows Azure 存储服务三剑客的第一个服务——表服务，包含它的架构、限制、开发方法以及性能最佳化等等，同时也提出了设计表服务时的建议事项，以帮助您在设计表服务时可以减少一些可能的性能问题。

下一章我们将会看到第二种存储——BLOB 存储服务，它和表服务完全不同，但访问方式非常类似，因此您也不必太过担心。