第**14**章

**使用云服务**

**本章内容**

**-------------------------------------------------------------------------------------------------------**

➤生成基本的Web请求

➤使用压缩技术缩减请求大小

➤调用WCF服务

➤在 XML 和 JSON 中使用OData 服务

虽然许多Windows Phone服务(诸如位置和推送通知)都依赖于云来执行部分或全部的数据处理，不过您可能也需要连接到其它的一些远程服务。这些可能是由您的组织提供的本地服务或云服务，也可能是由第三方提供的可以通过基于云的API进行访问的服务。

在本章中，您将了解到如何集成Windows通信层（Windows Communication Foundation，简称WCF）和简单对象访问协议（Simple Object Access Protocol，简称SOAP）服务。此外还将讨论如何通过使用RESTful（Representational State Transfer，表述性状态转移）服务来降低消息开销。

**14.1 HTTP请求**

当您需要访问Web中的内容时，无论它是一项服务或静态内容，如图像或文档，Windows Phone都为您提供了若干选择以便访问这些内容。不过，它们都可归结为执行一个HTTP请求。Windows Phone 7没有提供对原始的传输控制协议（Transmission Control Protocol，简称TCP）或用户数据报协议（User Datagram Protocol，简称UDP）套接字（socket- level）级别通信的支持，这意味着HTTP请求可供使用的最低通信级别。在本节中，您将了解如何使用WebClient和HttpWebRequest类来访问Web上的内容。之后，将学习如何使用WCF为服务请求的生成提供一个更高级的封装。

使用WebClient或HttpWebRequest类时需要注意的一件事情就是所有的网络调用（例如下载或上传内容）都是异步的。传统上，您可以在同步或异步调用之间进行选择，由于异步调用具有额外的复杂性，所以我们通常避免使用它。不过，当执行长时间运行或高延迟任务时使用异步调用会更好一些，因为它降低了构建未响应用户界面（UI）的可能性。Silverlight从Web技术继承而来，并因此严重依赖高延迟的网络调用，同时又要将框架保持得尽可能小，所以只提供了异步网络调用是合理的。

**14.1.1 WebClient**

如果您只是想简单的从指定的URL下载内容，那么使用WebClient是迄今为止最为简单的方式。要下载内容，您需要创建一个WebClient实例，并附加一个事件处理程序，在下载完成时调用事件处理程序，然后开始下载。在下面的示例中，使用DownloadStringAsync方法下载了一个XML文件。WebClient实例关联了两个事件处理程序，一个用于获得进度信息，另一个用于在下载完成时获得一条通知。DownloadStringAsync方法的第二个参数是一个任意的对象或标识符，它们可以在事件处理程序中被访问，以此确定事件所针对的下载内容。这样，一个单独的事件处理程序可以在WebClient 类的多个实例之间重复使用：

WebClient client = new WebClient();

public MainPage(){

InitializeComponent();

client.DownloadProgressChanged+=client\_DownloadProgressChanged; client.DownloadStringCompleted+=client\_DownloadStringCompleted; client.OpenReadCompleted += client\_OpenReadCompleted;

}

private void WebClientButton\_Click(object sender, RoutedEventArgs e){

client.DownloadStringAsync(

new Uri(“http://localhost/ServicesApplication/rssdump.xml”),

“sample rss”);

}

Void client\_DownloadProgressChanged(object sender, DownloadProgressChangedEventArgs e){

if (e.UserState as string == “sample rss”){

this.DownloadProgress.Value = e.ProgressPercentage;

}

}

Void client\_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e){

this.DownloadedText.Text = e.Result;

}

*MainPage.xaml.cs中的代码段*

每当WebClient触发一个事件，您都会看到更新会被直接反应在用户界面上。WebClient的操作（例如DownloadStringAsync）在后台线程中执行，而它们所生成的事件都将自动在创建WebClient实例的同一线程中触发。在本例中，由于该实例是在UI线程中创建的，所以事件将在该线程上触发，因而无须使用Dispatcher.BeginInvoke来封装任何UI的更新。

如您所知，DownloadStringAsync方法非常适合于下载那些可以表示为字符串的内容，例如，XML或者文本文档。如果您需要下载二进制内容（例如，图像），则可以使用OpenReadAsync方法，它会打开一个二进制流，以便您可以逐步读取已下载的内容：

private void WebClientButton2\_Click(object sender, RoutedEventArgs e){

client.OpenReadAsync(new Uri(“http://localhost/ServicesApplication/desert.jpg”),

“my picture”);

}

private void client\_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e){

var strm = e.Result;

var img = new BitmapImage();

img.SetSource(strm);

this.SampleImage.Source = img;

}

*MainPage.xaml.cs中的代码段*

需要注意的一点是，当使用OpenReadAsync方法时，不会触发DownloadProgressChanged事件。如果您希望使用下载状态来更新进度条，则必须自己来实现。例如，以下代码将内容分为100块，并且当每块从流中读取一块后便报告下载进度：

private void client\_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e){

var strm = e.Result;

var ms = new MemoryStream((int)strm.Length);

var buffer = new byte[strm.Length / 100];

var cnt = 0;

var progress = 0;

while (strm.Position < strm.Length){

cnt = strm.Read(buffer, 0, buffer.Length);

ms.Write(buffer, 0, cnt);

progress++;

this.Dispatcher.BeginInvoke(() = > {

this.DownloadProgress.Value = progress;

});

}

ms.Seek(0, SeekOrigin.Begin);

var img = new BitmapImage();

img.SetSource(ms);

this.Dispatcher.BeginInvoke(() = > {

this.SampleImage.Source = img;

});

}

*MainPage.xaml.cs中的代码段*

图14-1说明了分别展示了使用DownloadStringAsync和OpenReadAsync下载 XML文档（中间图像）和图像（最右侧的图像）。最左侧的图像展示了表示下载状态的进度条。

**上传内容**

借助UploadStringAsync（用于字符串数据）或OpenWriteAsync（用于二进制数据），WebClient还可以用于上传内容。在查看展示工作原理的示例之前，您需要在ASP.NET Web应用程序中创建一个简单的服务，用以接收上传的内容。首先，新建一个名为SimpleService的WCF服务，并如下定义契约（contract）和实现（implementation）：

[ServiceContract]

public interface ISimpleService{

[WebInvoke]

[OperationContract]

bool FileUpload(Stream input);

}

*ISimpleService.cs中的代码段*

< %@ ServiceHost Language=”C#” Debug=”true”

Service=”ServicesApplication.SimpleService”

CodeBehind=”SimpleService.svc.cs”

Factory= “System.ServiceModel.Activation.WebServiceHostFactory”% >

*SimpleService.svc中的代码段*

public class SimpleService : ISimpleService{

public bool FileUpload(Stream input){

var reader = new StreamReader(input);

var txt = reader.ReadToEnd();

if (txt.Length > 0){

return true;

}

else{

return false;

}

}

}

*SimpleService.cs 中的代码段*

除了WCF中常见的ServiceContract和OperationContract特性外，该服务还有一个应用于FileUpload方法的WebInvoke特性。这表明此方法可以通过REST服务调用而被调用。通常情况下，服务调用封装在一个SOAP信封（envelope）中，WCF会对其进行解析，并在向适当的方法发送有效负载前将其分离。在移动应用程序中，SOAP信封比有效负载本身占用更多的空间是很常见的。REST对在服务的URL内部调用的方法提供了更加轻量级的编码机制。若要启用REST终点，您需要使用如下system.serviceModel节来更新web.config文件，它使用webHttpBinding定义了一个终点。如果web.config文件中缺少此节，则可以在关闭配置标签前将其插入：

<system.serviceModel>

<behaviors>

<endpointBehaviors>

<behavior name=”WebBehavior”>

<webHttp />

</behavior>

</endpointBehaviors>

</behaviors>

<services>

<service name=”ServicesApplication.SimpleService”>

<endpoint address=”pox” binding=”webHttpBinding”

contract=”ServicesApplication.ISimpleService” behaviorConfiguration=”WebBehavior” />

</service>

</services>

</system.serviceModel>

*web.config中的代码段*

为了通过REST调用 FileUpload 方法，您需要给予下面的URL一个 http POST请求：

http://localhost/ServiceApplication/SimpleService.svc/FileUpload

此URL有4个部分：

（1） 宿主（host），在本例中即本地主机（localhost），因为ASP.NET Web应用程序运行于IIS。

（2） 虚拟目录，在本例中是ASP.NET Web应用程序的名称，即ServiceApplication。您可以通过更改IIS中的虚拟目录名称来对其进行更改。

（3） 服务基址，就是与服务名称相对应的SimpleService.svc。如果您想移除.svc扩展名，可以定义一条URL Rewrite规则（通过Microsoft Web Platform Installer for IIS 7下载URL Rewrite）。为您的ASP.NET Web应用程序的Web.Config文件添加如下代码，它定义了一个URL Rewrite，用来将SimpleService更改为SimpleService.svc。

<system.webServer>

<rewrite>

<rules>

<rule name=”RemoveSvc” stopProcessing=”true”>

<match url=”^SimpleService/(.\*)$”/>

<action type=”Rewrite”

url=”SimpleService.svc/{R:1}” />

</rule>

</rules>

</rewrite>

</system.webServer>

*web.config 中的代码段*

完成此更新后，新的URL为：

http://localhost/ServiceApplication/SimpleService/FileUpload

（4） 最后一部分是需要被调用的方法，在本例中即FileUpload方法。在Windows Phone应用程序中调用此方法其实只是调用OpenWriteAsync方法，然后将内容写入请求流而已，该流可以通过OpenWriteCompleted事件处理程序访问：

private void WebClientButton3\_Click(object sender, RoutedEventArgs e){

client.OpenWriteAsync(

new Uri(“http://localhost/ServicesApplication/SimpleService/FileUpload”),

“POST”);

}

private void client\_OpenWriteCompleted(object sender, OpenWriteCompletedEventArgs e){

using (var strm = e.Result){

var bytesToWrite= RawContent();

strm.Write(bytesToWrite,0,bytesToWrite.Length);

strm.Flush();

}

}

private byte[] RawContent(){

var content=”Test content to be uploaded”;

var raw = Encoding.UTF8.GetBytes(content);

return raw;

}

*MainPage.xaml.cs中的代码段*

在本例中，使用OpenWriteAsync方法有些小题大做，因为正在上传的内容实际上是一个字符串。这可以通过使用UploadStringAsync方法来替代，如下所示：

client.UploadStringAsync(

new Uri(“http://localhost/ServicesApplication/SimpleService/FileUpload”),

“POST”, “Test content to be uploaded”);

如同DownloadStringAsync那样，UploadProgressChanged和 UploadStringCompleted事件允许您的应用程序将上传进度和完成情况显示给用户。

**14.1.2 HttpWebRequest**

WebClient 类可用来做简单的HTTP操作，例如下载文件。不过，如果需要更好地控制HTTP请求，则应该使用HttpWebRequest类。这使它允许您设置标头（header），指定请求的内容类型以及使用流将数据传输到请求中。使用HttpWebRequest类和WebClient的差别不是很大。WebClient使用事件回调来指示下载已完成，或者指示应用程序可以开始读取。当使用HttpWebRequest时，作为请求的一部分，您需要提供一个回调方法。响应处理就绪时该方法被调用。下面的代码使用一个HttpWebRequest实例下载XML文件，而非WebClient

private void HttpWebRequestButton\_Click(object sender, RoutedEventArgs e){

var req = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/rssdump.xml”))

as HttpWebRequest;

req.BeginGetResponse(HttpWebRequestButton\_Callback, req);

}

private void HttpWebRequestButton\_Callback(IAsyncResult result){

var req = result.AsyncState as HttpWebRequest;

var resp = req.EndGetResponse(result);

var strm = resp.GetResponseStream();

var reader = new StreamReader(strm);

this.Dispatcher.BeginInvoke(() = > {

this.DownloadedText.Text = reader.ReadToEnd();

this.TextViewer.Visibility = System.Windows.Visibility.Visible;

});

}

*MainPage.xaml.cs中的代码段*

HttpWebRequest的回调在后台线程中完成。这意味着您不需要中断用户界面来处理响应。然而，它却意味着如果您想要更新任何UI元素，都需要使用Dispatcher.BeginInvoke切换到UI线程，如此例中所示。

**1 上传内容**

同WebClient一样，您可以使用HttpWebRequest对象上传内容。您需要调用BeginGetRequestStream方法，而不是立即调用BeginGetResponse方法。该方法的回调允许您将数据写入到将要上传的请求流。当您完成了对请求流的写入后，再调用BeginGetResponse方法。响应可用时，将调用相应的回调：

private void HttpWebRequestButton2\_Click(object sender, RoutedEventArgs e){

HttpWebRequest req = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/SimpleService/FileUpload”))

as HttpWebRequest;

req.Method = “POST”;

req.BeginGetRequestStream(HttpWebRequestButton2\_RequestCallback, req);

}

private void HttpWebRequestButton2\_RequestCallback(IAsyncResult result){

var req = result.AsyncState as HttpWebRequest;

using(var strm = req.EndGetRequestStream(result)){

var bytesToWrite = RawContent();

strm.Write(bytesToWrite, 0, bytesToWrite.Length);

strm.Flush();

}

req.BeginGetResponse(HttpWebRequestButton\_Callback, req);

}

*MainPage.xaml.cs中的代码*

**2 Cookies**

您需要执行一个请求序列，HttpWebRequest会变得更加有用。这样的例子可能是一对服务方法，其中第一个方法用于在第二个方法获取所需数据之前验证用户的身份。在这种情况下，一个常见的解决方案是，让用于身份验证的服务方法返回一个cookie，该cookie必须同所有其它服务方法的请求一起传递。HttpWebRequest通过CookieContainer简化此操作，每次收到响应时都会更新CookieContainer，以便为所有后续请求定义cookie。下面分析用于验证用户身份的方法：

[WebGet(UriTemplate=”SignIn/{username}/{password}”)]

[OperationContract]

bool SignIn(string username, string password);

*ISimpleService.cs中的代码段*

public bool SignIn(string username, string password){

if (string.IsNullOrEmpty(username) || string.IsNullOrEmpty(password)) return false;

// Add additional logic to authenticate the user here

WebOperationContext.Current.OutgoingResponse.Headers[“Set-Cookie”] =

“username=” + username + “ & password=” + password + “; path=/”;

return true;

}

*SimpleService.svc.cs中的代码段*

该示例只是验证已指定的用户名和密码。更完整的实现是在数据库或类似的数据存储区中验证指定的凭据。如果分析接口定义，将会发现它使用的是WebGet而不是WebInvoke特性。这表明它是一个HTTP GET操作，并且URL的格式应该是方法名称SignIn，这两个参数后面由"/"符号隔开。假设提供了一个有效的用户名和密码，并且cookie被附加到传出的响应上。通常情况下，您希望添加更多的逻辑来验证用户身份，并将用户名和密码组合散列（hash）操作，以便它不容易被识别。

此序列中的第二种方法是查看传入的请求，并确定身份验证cookie是否已指定。如果它们已被指定，继而允许FileUpload方法：

[WebInvoke]

[OperationContract]

bool FileUploadCookieRequired(Stream input);

*ISimpleService.cs中的代码段*

public bool FileUploadCookieRequired(Stream input){

var authHeader = WebOperationContext.Current.IncomingRequest.Headers[“Cookie”];

if (string.IsNullOrEmpty(authHeader)) return false;

var cookies = authHeader.Split(‘ &’);

var authCookiesFound = (from cookie in cookies

where cookie.StartsWith(“username”) ||

cookie.StartsWith(“password”)

select cookie).Count() == 2;

if (!authCookiesFound) return false;

return FileUpload(input) ;

}

*SimpleService.svc.cs中的代码段*

要在Windows Phone应用程序中管理cookie，只需创建一个CookieContainer，使其可以在连续的HttpWebRequest调用之间存储所有相关的cookie即可：

private void HttpWebRequestButton3\_Click(object sender, RoutedEventArgs e){

string username = “Nick”;

string password = “MyPassword”;

HttpWebRequest req = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/SimpleService/SignIn/”

+ username + “/” + password))

as HttpWebRequest;

req.CookieContainer = new CookieContainer();

req.BeginGetResponse(SignIn\_Callback, req);

}

private void SignIn\_Callback(IAsyncResult result) {

var req = result.AsyncState as HttpWebRequest;

var resp = req.EndGetResponse(result);

resp.Close();

HttpWebRequest newReq = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/SimpleService/FileUploadCookieRequired”))

as HttpWebRequest;

newReq.CookieContainer = req.CookieContainer;

newReq.Method = “POST”;

newReq.BeginGetRequestStream(HttpWebRequestButton2\_RequestCallback, newReq);

}

*MainPage.xaml.cs中的代码段*

第一个HttpWebRequest调用了SignIn方法，通过 URL 传递用户名和密码。为了捕捉从此方法响应返回的cookie ，我们向第一个请求中附加了一个CookieContainer实例。然后，CookieContainer实例被传递到第二个HttpWebRequest对象，这样，cookie将作为请求的一部分发送回服务器。

要记住的一点是，这只是一个简单的场景，用于演示如何共享请求之间的 cookie。正如当前的示例所示，凭据都是以明文传输的，对于有人伪造身份验证cookie几乎没有保护措施。如果您要在生产系统中使用此凭据，那么就务必需要对凭据进行加密，或使用SSL（Secure Sockets Layer，安全套接层）信道进行通信。

**14.1.3 凭据**

除非您正在访问公有数据，否则您需要提供某种形式的身份认证。在上一节中，您看到了一个人为设计的在服务器上进行身份验证的示例。然而，在Windows Phone 7 开发中遇到的最常见的验证形式是基本、摘要、窗体和Windows/NTLM。WebClient和HttpWebRequest类均内置支持向请求中的Basic身份验证提供凭据。而对摘要、窗体和 Windows/NTLM 身份验证支持的不是很好。下面的代码已被设置为使用Basic身份验证访问文件夹中的内容。您可以使用 IIS Manager来配置您的Web应用程序中单个文件夹所需的身份验证。

private void WebClientButton2\_Click(object sender, RoutedEventArgs e){

client.Credentials = new NetworkCredential(“Nick”, “MyPassword”);

client.OpenReadAsync(

new Uri(“http://localhost/ServicesApplication/BasicAuth/desert.jpg”));

}

private void HttpWebRequestButton4\_Click(object sender, RoutedEventArgs e){

HttpWebRequest req = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/BasicAuth/rssdump.xml”))

as HttpWebRequest;

req.Credentials = new NetworkCredential(“Nick”, “MyPassword”);

req.BeginGetResponse(HttpWebRequestButton\_Callback, req);

}

*MainPage.xaml.cs中的代码段*

14.1.4 压缩

到目前为止，所有的请求一直使用未经压缩的数据。在构建Windows Phone应用程序时，您必须意识到网络可用性和带宽成本。尤其是当您的应有程序需要在基于蜂窝的网络上传输数据，这通常会比在家庭中或基于工作的PC环境中所使用的网络要慢一个数量级。如果您需要在应用程序和远程服务之间传输大量的信息，应该考虑对数据进行压缩。

在数据经由HTTP传输之前，您可以利用IIS 7内置支持的压缩技术在服务器端对其进行自动压缩，这可以大幅降低下载到您的应用程序中的数据大小和成本。IIS 同时支持静态和动态内容压缩。前者被应用于不随时间推移而发生变化的内容以及可以被缓存的压缩数据（如图像），而后者适用于动态内容。在这两种情况下，只有包含Accept-Encoding标头的内容向IIS发出请求时才被压缩，它被设置为“gzip”还是“deflate”则取决于压缩类型。这表示客户端可以接受压缩后的内容。

您首先要做的是确保IIS中的压缩已启用。默认情况下安装的是静态压缩。但是，如果您要压缩的数据来自一项服务，您还应该通过Microsoft Web Platform Installer安装Dynamic Content Compression（动态内容压缩）模块（可从 www.microsoft.com/web/downloads/ platform.aspx 站点获取），如图14-2所示。

安装Compression模块后，打开IIS Manager并导航到您要启用压缩的虚拟目录。选择Compression功能，并确保已启用两个复选框，如图14-3所示。

默认情况下，这只会为文本和x-javascript MIME（Multipurpose Internet Mail Extensions，多功能Internet邮件扩充服务）类型的数据启用压缩。为了完整起见，您需要为所有的SOAP、 JSON 和REST服务启用动态压缩。这不会影响任何现有的服务或客户端，因为压缩是客户端必须选择的功能。任何现有的客户端将会继续接收未经压缩的数据。若要包括对其它MIME 类型的动态压缩，则需要在IIS Manager中选择根服务器节点（不是虚拟目录或网站），并从Features视图中选择Configuration Editor。从屏幕顶部的下拉列表框中，选择system.webServerhttpCompression，然后选择要编辑的 dynamicTypes属性。这将显示一个类似于图14-4所示的对话框。

请确保所有不同的MIME类型都出现了，因为它们将确保对SOAP、JSON和REST服务调用启用动态内容压缩。您还可以向staticTypes属性添加image/jpg MIME 类型，以启用对JPEG图像的静态压缩。一旦您完成了操作，请确保应用所做的更改，然后重新启动IIS。您必须这样做；否则，将继续获得未经压缩的数据。

若要确保其正常工作，您可以使用诸如Fiddler(www.fiddler2.com)的工具来验证返回的数据是否被压缩。运行Fiddler，然后选择Request Builder选项卡。请确保方法类型设置为GET，并输入您的Web应用程序中包含的图像URL（例如，http://localhost/ServicesApplication/desert.jpg）。在Request Headers框中，输入Accept- Encoding: gzip, deflate，然后单击Execute按钮。首次这样做时，可能会得到显示在图 14-5右侧的响应，即未压缩。然后，如果您重复请求，应该会看到显示在图 14-5左侧的响应，它表明响应已经压缩。如果您单击黄色的Notice栏，Fiddler将为您解压缩图像，再次在右图中显示响应。

在您首次请求时不能压缩的缘由与IIS中的静态压缩工作方式有关。当第一次访问资源时，它立即返回未经压缩的数据。同时，资源在后台异步的压缩和缓存。任何后续的请求将收到压缩的版本。此过程仅适用于静态内容，指向相同URI的两个或多个请求将返回完全相同的响应

另一方面，Dynamic压缩不会缓存压缩版的内容。这意味着每个传入的请求处理时间会稍长，因为需要重新生成响应并从头开始压缩。但是，它可以确保对所有请求都提供压缩的数据，这在数据量十分庞大时可能是有益的。

当接收到的请求设置了Accept-Encoding标头，IIS将对数据进行压缩。但是，当从一个Windows Phone应用程序发出请求时不能设置此标头，因为它是一个受限的标头。一种解决方案是在 Windows Phone应用程序请求中指定一个自定义的标头值，并在IIS处理请求之前将它转换成适当的Accept-Encoding标头值。这可以通过定义URL Rewrite规则来实现，该规则将检测自定义标头（本例中标头名称为Compress-Data，值为true）并重写请求使其包含值为gzip，deflate的Accept-Encoding标头。

默认情况下，URL Rewrite模块不允许设置Accept-Encoding标头（当使用重写规则时该标头被称为服务器端变量）。若要重写此行为，您必须允许 web.config文件的重写部分，以重写可以被修改的一组服务器端变量。找到并打开IIS配置文件applicationHost.config（通常位于C:\Windows\System32\inetsrv\config）。修改位于rewrite的sectionGroup中的allowedServerVariables节，将overrideModeDefault设置Allow。

<configuration>

<configSections>

<sectionGroup name=”system.webServer”>

<sectionGroup name=”rewrite”>

<section name=”allowedServerVariables” **overrideModeDefault=”Allow”** />

接下来，您只需向ASP.NET Web应用程序中web.config文件的system.webServer部分添加以下重写规则。

<system.webServer>

<rewrite>

<allowedServerVariables>

<add name=”HTTP\_ACCEPT\_ENCODING” />

</allowedServerVariables>

<rules>

<rule name=”RewriteWithAcceptEncoding”

patternSyntax=”Wildcard” stopProcessing=”false”>

<match url=”\*” />

<conditions>

<add input=”{HTTP\_COMPRESS\_DATA}” pattern=”true” />

</conditions>

<action type=”None” />

<serverVariables>

<set name=”HTTP\_ACCEPT\_ENCODING” value=”gzip,deflate” />

</serverVariables>

</rule>

</rules>

</rewrite>

</system.webServer>

*web.config中的代码段*

为了从Windows Phone应用程序请求压缩的数据，您只需要包含值为true的Compress-Data标头即可。此类请求会被RewriteWithAcceptEncoding规则截取，该规则用于标识Compress-Data标头（请注意此处被指定为HTTP\_COMPRESS\_DATA的输入值，该规则是要将标头名称转换为大写，并且-被替换为\_，附加的HTTP\_前缀表明它是一个标头）。然后它将Accept-Encoding标头（标头名称被转换为名为HTTP\_ACCEPT\_ENCODING的服务器端变量）设置为gzip，deflate，这将使得IIS返回压缩的数据。

不同于服务器，它内置支持压缩，而Windows Phone却不能自动处理HTTP响应的解压缩。若要解压缩响应，您需要编写自己的解压缩库或使用第三方库。在这里，您将使用SharpZipLib（可从http://sharpdevelop.net/OpenSource/SharpZipLib/获得）。遗憾的是，并没有可以直接用于Windows Phone项目的SharpZipLib，所以您需要创建自己的Windows Phone类库，并包含这些源文件。您需要定义以下额外的编译常量，NETCF\_2\_0 NETCF（参见项目的Properties窗口中的Build选项卡），然后修复所有剩余的生成错误，其中都是很容易解决的小问题。完成后，可以向您的Windows Phone应用程序添加对该项目的引用。

为了使您的Windows Phone应用程序能告知服务器，它愿意接受压缩的HTTP响应，您需要在HTTP请求中包含设置为true的Compress-Data标头。设置此标头仅仅是表明了一个请求，所以当收到响应时，您必须检查它实际是否真正的压缩了。这可以通过检查响应中是否包含设置为GZIP的Content- Encoded标头，它与您所请求的压缩类型匹配。该标头是否被设置决定着内容是否应当压缩。

private void HttpWebRequestButton5\_Click(object sender, RoutedEventArgs e){

HttpWebRequest req = HttpWebRequest.Create(

new Uri(“http://localhost/ServicesApplication/desert.jpg”))

as HttpWebRequest;

req.Headers[“Compress-Data”] = “true”;

req.BeginGetResponse(HttpWebRequestButton\_CompressedCallback, req);

}

private void HttpWebRequestButton\_CompressedCallback(IAsyncResult result){

var req = result.AsyncState as HttpWebRequest;

var resp = req.EndGetResponse(result);

var strm = resp.GetResponseStream();

var ms = new MemoryStream();

if ((resp.Headers[“Content-Encoding”] + “”).Contains(“gzip”)){

using (var gzip = new GZipInputStream(strm)){

gzip.PipeTo(ms);

}

}

else {

strm.PipeTo(ms);

}

ms.Seek(0, SeekOrigin.Begin);

this.Dispatcher.BeginInvoke(() = >

{

var img = new BitmapImage();

img.SetSource(ms);

this.SampleImage.Source = img;

});

}

*Utilities.cs中的代码段*

此代码使用了一个名为PipeTo的扩展方法，它将内容从一个流传递到另一个流中：

public static class Utilities{

public static void PipeTo(this Stream inputStream, Stream outputStream){

var buffer = new byte[1000];

var bytesRead = 0;

while ((bytesRead = inputStream.Read(buffer, 0, buffer.Length)) > 0){

outputStream.Write(buffer, 0, bytesRead);

}

}

}

*Utilities.cs中的代码*

如果多次运行此代码，应该会看到第一次，或者可能前两次下载的内容是未经压缩的。之后将下载经过压缩的。IIS静态压缩会为已经缓存的内容设置一个有效期限，所以一段空闲时间过后内容将被刷新，从而导致它下次下载的内容是未经压缩的。如果您想要确保内容始终压缩，则可考虑在IIS Manager 中将image/jpeg MIME类型更改为动态内容压缩。

**14.2 WCF/ASMX服务**

WebClient 和HttpWebRequest都提供一种从远程服务器上下载内容以及上传内容到远程服务器的简单方式。如果您试图与基于SOAP 的服务通信，如ASMX Web Service或WCF Service，这种简化就没有那么明显了。幸运的是，Visual Studio可以为您自动地生成代理类，而代理类可以利用这些更高级的规范来隐藏调用服务的复杂性。

**14.2.1 服务配置**

默认情况下，ASMX Web服务不需要做任何更改就可以从Windows Phone应用程序访问。不过，同样的情况可能并不适合基于WCF的服务。由于受到.NET Compact Framework的限制，它不能使用一些更复杂的WCF绑定，如 wsHttpBinding。这意味着，如果您有想要从Windows Phone应用程序中访问的WCF服务，将必须确保该服务的配置使用的是basicHttpBinding。

正如本章前面所介绍的，您也可以访问那些使用webHttpBinding的WCF服务。但是，您不能使用 Visual Studio的 Add Service Reference功能来封装对这类服务的调用。相反，您必须使用HttpWebRequest或WebClient类来手动地发出服务请求。

您将要创建一个从联机数据存储区中返回联系人列表的简单服务。首先，向ASP.NET Web应用程序添加一个新的名为ContactService.svc的WCF服务。如果您在.NET Framework v4中创建此服务，那么 web.config文件中将不会包含任何条目。您可能会对此感到惊讶，但新版的简化配置系统意味着，在默认情况下，当您向一个虚拟目录放置一个WCF服务时，该目录会成为服务的基址，并且basicHttpBinding被自动赋值为HTTP协议。这意味着运行在.NET Framework 4.0上的一个新的WCF 服务可以从基于Windows Phone的应用程序直接访问。

但是，如果您需要调整WCF服务配置（例如，增加请求的最大值或响应大小）或者您的WCF服务正在运行在一个旧版.NET Framework上，那么您需要知道如何对服务进行配置以使用basicHttpBinding。最简单的方法是使用WCF Service Configuration Editor，它可以通过如图14-6左图所示的Tools菜单访问。

首次打开Editor时，右击解决方案管理器的web.config文件，您会发现上下文菜单中出现了一个新的项目。从右键菜单中选择Edit WCF Configuration可以以最快的速度在Editor中打开web.config文件。如图 14-7所示，在Editor中打开web.config文件后，您应该会看到两个主窗格。

在这里，先前创建的REST服务都列在了Configuration窗体中的Services节点下，而非ContactService节点。如果您正在使用的是旧版.NET Framework，可能会有ContactService节点。如果缺少该项，您可以在Services窗格中选择Create a New Service，然后会出现一个New Service Element Wizard，通过它即可向Services列表添加ContactService。下面列出设置服务配置的步骤：

(1)输入ServicesApplication.ContactService作为服务类型。如果单击Browse按钮，您可以导航到 bin 文件夹中，接着双击ServicesApplication程序集。

(2)契约应当被正确的选择为ServicesApplication.IContactService。

(3)通信模式应当保持为HTTP。

(4)由于您要确保从 Windows Phone应用程序可访问，所以您需要继续使用Basic Web Services互操作模式。

(5)使Address字段为空（忽略弹出的警告）。

一旦您确认了新的服务配置细节，您应当返回到Configuration Editor中，并选定新创建的服务终结点，如图 14-8所示。如果您已为 ServicesApplication.ContactService进行了服务配置，需要确保将终结点的Binding设置为basicHttpBinding。

**14.2.2 添加服务引用**

由于所有的服务器端服务均设置为使用basicHttpBinding，所以您现在可以在Windows Phone应用程序中创建代理类。右击资源管理器中的项目，选择Add Service Reference。这将显示如图14-9所示的Add Service Reference对话框。在Address字段中输入服务的地址，并单击 Go。一旦发现该服务，Services and Operations窗格中将显示服务的有关信息。

然后输入一个命名空间，在本例中即Contacts，并单击OK按钮来添加服务引用。您应该看到在Windows Phone项目中出现了一个Service References文件夹，它的下面有一个Contacts节点。

**14.2.3 服务实现与执行**

目前，由于ContactService是通过WCF服务项目模板配置的，所以其中只包含一个单独的方法， DoWork，而且它什么也不做。我们将此方法更改为RetrieveContacts，使它返回一个Contact对象的数组。您还需要创建另一个名为SaveContact的方法，它接受一个联系人作为参数。接口和实现如下：

[ServiceContract]

public interface IContactService{

[OperationContract]

Contact[] RetrieveContacts();

[OperationContract]

public void SaveContact(Contact contactToSave);

}

*IContactService.cs中的代码*

对ContactService执行这些更改后，您将需要更新在Windows Phone应用程序中创建的代理类。在Visual Studio 中可以通过右击解决方案管理器中Service References节点下的适当节点并选择Update Service Reference而简化该过程。

**测试WCF服务**

您开始构建更复杂的服务时，它们变得越来越难以调试和诊断。在开发Windows Phone应用程序时，很难确定问题是出在客户端还是服务器端代码。对于此类问题，我们可以使用Visual Studio自带的一个WCF Test Client，它可用于测试您的服务方法以及将客户端代码从复杂的环境中隔离出来。

图 14-10所示的WCF Test Client显示了对SaveContacts方法的调用。

当您运行包含要测试的WCF服务的项目时，WCF Test Client可能不会自动启动。这时，它可以从C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\ WcfTestClient.exe手动启动。一旦启动，您可以通过选择File ➪ Add Service并输入WCF服务的URL将其指向您要测试的服务。

在Windows Phone应用程序中调用这些服务的代码相对比较简单，并且在结构上类似于使用WebClient类。ContactServiceClient 代理类的实例创建后，它可用于调用RetrieveContacts和SaveContact方法。当对于方法调用的服务器响应可用时，这些方法相应的事件被触发：

Contacts.ContactServiceClient client = new Contacts.ContactServiceClient();

public ServicesPage(){

InitializeComponent();

client.RetrieveContactsCompleted += client\_RetrieveContactsCompleted;

client.SaveContactCompleted += client\_SaveContactCompleted;

}

private void GetContactsButton\_Click(object sender, RoutedEventArgs e){

client.RetrieveContactsAsync();

}

void client\_RetrieveContactsCompleted(object sender,

Contacts.RetrieveContactsCompletedEventArgs e){

ContactsList.ItemsSource = e.Result;

}

private void AddContactButton\_Click(object sender, System.Windows.RoutedEventArgs e){

var contact = new Contacts.Contact(){

Id = Guid.NewGuid(),

Name = this.NameText.Text,

EmailAddress = this.EmailText.Text

};

client.SaveContactAsync(contact,contact);

}

void client\_SaveContactCompleted(object sender,

AsyncCompletedEventArgs e){

var contact = e.UserState as Contacts.Contact;

var list = this.ContactsList.ItemsSource

as ObservableCollection < Contacts.Contact > ;

list.Add(contact);

}

*ServicesPage.xaml.cs中的代码段*

图 14-11所示的是RetrieveContacts方法被调用后页面的外观。返回的联系人显示在一个列表框中。输入一个名称和电子邮件地址后，可通过单击Add Contact按钮来添加新的联系人。

与WebClient一样，值得注意的是，当响应一个ContactServiceClient类所触发的事件时不需要使用Dispatcher.BeginInvoke切换到UI线程，因为这已经得到处理。另外，请注意无论您处理何种后期处理，它们均会在UI线程执行，这有可能会使用户体验大打折扣。如果您要在数据呈现前对其进行大量的处理，应该考虑使用后台线程来执行此操作。

**14.2.4 自定义标头**

在某些情况下，您需要指定自定义HTTP标头，这些标头应该包含在每个服务请求中。为此，您需要创建一个OperationContextScope，并对传出的消息应用一个新的 HttpRequestMessageProperty。下面的示例阐释了这种技术，它通过设置授权（Authorization）标头，就好像您使用Basic Authentication以保证您的服务一样：

using (new OperationContextScope(client.InnerChannel)){

HttpRequestMessageProperty prop = new HttpRequestMessageProperty();

string credentials = Convert.ToBase64String(Encoding.UTF8.GetBytes

“Nick” + “:” + “MyPassword”));

prop.Headers[“Authorization”] = “Basic “ + credentials;

OperationContext.Current.OutgoingMessageProperties.Add(

HttpRequestMessageProperty.Name, prop);

client.RetrieveContactsAsync();

}

**14.2.5 凭据**

在使用凭据时，没必要使用上面所示的自定义HTTP标头；相反地，生成的代理客户端上有一个ClientCredentials属性，您可以用它来指定用于在服务器上进行身份验证的用户名和密码：

client.ClientCredentials.UserName.UserName = “Nick”;

client.ClientCredentials.UserName.Password = “MyPassword”;

第18章中有关安全性的内容将详细介绍如何访问受到安全保护的服务。

**14.3 WCF数据服务**

开发人员长期以来对不断编写CRUD代码而深表失望。CRUD在这指的是在数据库中进行创建（Creates）、读取（Reads）、更新（Updates）和删除（Deletes）项目的代码。这样的代码可能会直接操纵数据库，也可能会以在客户端或服务器端编写代理代码的形式来简化应用程序的CRUD操作。WCF数据服务（WCF Data Services）提供了一个可扩展的工具，用于发布使用基于REST接口的数据，以使得客户端应用程序可以使用和更新，而无需开发CRUD样式的自定义代码。

**14.3.1 OData和WCF数据服务**

虽然基于SOAP的服务由于其自我描述的性质提供了许多好处，但它们也为往返发送的消息大小增加了很大的开销。大多数桌面应用程序都可以摆脱这些束缚，因为通常它们有一个到目标服务器的高速连接。 对于Windows Phone应用程序而言，连接和带宽都很有限，那么使用拥有最小开销的技术将更有意义。

数据服务使用开放数据协议（Open Data Protocol，简称OData）的网络协议（www.odata.org）发布和使用数据。默认的数据格式是XML，但稍后您将看到，您还可以在JSON中从WCF数据服务请求数据，使它非常紧凑并且容易在您的应用程序中使用。

在本节中，您将使用一个示例患者数据库。您将看到WCF数据服务如何用于发布一组现有的患者信息以及从 Windows Phone应用程序接收更新。

**1 ADO.NET Entity Data Model**

要创建一个WCF数据服务，您首先需要定义数据上下文。数据上下文提供了WCF数据服务和后端数据库提供的对象模型之间的映射。在本例中，向您的ASP.NET Web应用程序添加一个名为PatientModel.edmx的一个新的ADO.NET 实体数据模型。这将显示Entity Data Model Wizard，它提示您选择要在模型中包含的数据库和表。图14-12所示的是包含在实体模型中的简单数据模型。

一旦拥有了数据库的实体模型，您需要向项目中添加新的WCF数据服务。将其命名为PatientDataService.svc。这使得数据通过实体模型而被访问，并使其可通过REST终结点进行访问。您需要对WCF数据服务模板进行的更改仅仅是指定数据上下文，在本例中即PatientEntities，并描述哪些实体可以访问。在本例中，您希望客户端对所有的实体是可读和可写的。这可以通过添加一条新的带有\*和EntitySetsRights.All访问规则来指定。

public class PatientDataService : DataService < PatientEntities > {

public static void InitializeService(DataServiceConfiguration config){

config.SetEntitySetAccessRule(“\*”, EntitySetRights.All);

config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;

}

}

*PatientDataService.svc.cs中的代码段*

当您运行应用程序并导航到新创建的WCF数据服务时，应该会看到数据服务公开的一个实体列表。在本例中，包括Patients、Notes和Appointments。如果您现在向数据服务URL的末尾追加这些实体之一，应看到这些实体的列表。图14-13说明了测试数据库中Patients列表的开始部分。

现在已经创建了WCF数据服务并通过基于REST的API公开了数据库的内容，那么您就可以从Windows Phone应用程序中使用它了。您可以使用一种称为DataSvcUtil的命令行工具来自动生成一组代理类，而不必使用HttpWebRequest类以及手动地创建所需的URL和XML解析代码。打开命令提示符窗口，然后导航到您的Windows Phone应用程序的文件夹。运行以下命令来创建WCF数据服务代理类（如图14-14）：

“%windir%\Microsoft.NET\Framework\v4.0.30319\DataSvcUtil.exe” /version:2.0

/dataservicecollection /language:CSharp /out:PatientData.cs

/uri:http://localhost/ServicesApplication/PatientDataService.svc

然后，在Windows Phone项目中，右击解决方案管理器并选择Add Existing Item。找到新创建的PatientData.cs 文件，然后单击Add按钮。此时编译该工程将会导致错误，这是由于生成的包装器要依赖OData Client Library，它可从[www.microsoft.comdownloads](http://www.microsoft.comdownloads)站点获得。下载并安装该库，然后在您的Windows Phone项目中对它添加引用。在Add Reference对话框中，您需要切换到Browse选项卡，并导航到安装该库的文件夹。选择System.Data.Services.Client.dll，然后单击OK按钮。

**2 查询**

使用WCF数据服务和使用较为传统的SOAP服务稍有不同，SOAP服务是以面向方法调用为主的。相反，WCF数据服务是围绕使用语言集成查询（Language Integrated Query，简称LINQ）表达式查询和选择数据的概念而构建的。下面首先列举一个返回所有患者的简单查询开始：

首先您会注意到，使用WCF数据服务的代理类与使用HttpWebRequest对象有着相似之处。当您通过BeginExecute方法提交查询时提供了一个回调，该回调会在获得服务器的响应后被调用。如果您需要执行用户界面的任何更新，在此回调函数内您必须使用Dispatcher.BeginInvoke。

在考虑您可以编写的查询表达式类型之前，您应该注意到查询将最终被转换为用于REST请求的唯一的URL。在上例中，生成的URL和图14-13显示的一样。

http://localhost/ServicesApplication/PatientDataService.svc/Patients

首先，我们添加一个筛选器，仅列举名字为Tom的患者。正如您所看到的，您可以使用LINQ表达式来筛选PatientEntities对象的Patients属性：

private void GetFilteredPatientsButton\_Click(object sender, RoutedEventArgs e){

var query = from p in entities.Patients where p.FirstName == “Tom” select p;

var dsquery = query as DataServiceQuery < Patient > ;

dsquery.BeginExecute(Patients\_Callback, dsquery);

}

*DataServicesPage.xaml.cs中的代码段*

注意，完成到DataServiceQuery类型的转换是必需的；否则，BeginExecute方法不能被访问。默认情况下，LINQ查询返回一个 IQueryable对象，但在本例中，由于您正在使用WCF数据服务，所以处理类型转换使得您能够访问如BeginExecute的功能，而它只适用于 WCF数据服务上下文中。正如您所想象的，执行此查询将返回一个数据集，其中包含名字为Tom的患者列表。您可以通过检查此查询的URL来验证这一点，它清楚地显示了应用程序的筛选条件：

http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$filter=FirstName eq’Tom’

当执行一个查询时，您可能想要对最终得到的数据进行集排序：

var query = from p in entities.Patients orderby p.FirstName descending select p;

//http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$orderby= FirstName desc

默认情况下，对患者实体的WCF数据服务查询将只返回患者对象本身的详细信息。图14-12所示的ADO.NET实体数据模型表明Patients还包含Notes和Appointments。在查询中，您可以使用Expand方法，以指示查询应该深度加载一个或更多的相关实体

var query = from p in entities.Patients.Expand(“Appointments”)

select p;

//http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$expand= Appointments

var query = from p in entities.Patients.Expand(“Notes”)

select p;

//http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$expand= Appointments

var query = from p in entities.Patients.Expand(“Notes,Appointments”)

select p;

//http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$expand=Notes, Appointments

如果约见（appointment）实体有一个相关的笔记（script）实体，您就可以使用以下查询进行深度加载，以便返回与所选患者（patient）约见（appointment）时记录的全部笔记（script）：

var query = from p in entities.Patients.Expand(“Appointments/Scripts”)

select p;

//http://localhost/ServicesApplication/PatientDataService.svc/Patients()?$expand= Appointments/Scripts

**3 自定义方法与存储过程**

鉴于WCF数据服务的默认行为是充分使用基本查询访问数据，所以会有一些限制。例如，您难以执行一个查询，来检索至少有一个预约的所有患者，或者某个位置特定距离范围内的所有患者。而在Windows Phone客户端中，完全可以执行与此类似的更为复杂的查询，只需将所有患者和相关实体的列表下载下来并在本地进行筛选即可。但是，这会带来很大的开销，因为所有数据都要被传送到客户端的设备。一个更好的替代方案是在服务器端执行自定义的SQL查询。仅向客户端返回所需的行。对于本例，您将在SQL Server数据库上使用一个名为PatientsWithAppointments的存储过程，其定义如下所示：

CREATE PROCEDURE PatientsWithAppointments

AS

BEGIN

select \*

from Patient p

where (select COUNT(\*) from Appointment apt where p.PatientId=apt.PatientId) > 0

END

为了使用该存储过程，需要在您的ADO.NET实体数据模型中将其添加为一个函数。一旦在数据库中创建了存储过程，然后在您的ASP.NET Web应用程序中打开实体数据模型，并选择Update Model from Database。通过向导一步一步执行，确保它找到并添加了PatientsWithAppointments方法。在Model Browser Tool窗口中，导航到位于PatientModel.Store ➪ Stored Procedure下的PatientsWithAppointments节点。右击该节点并选择Add Function Import。图14-15所示的是Add Function Import对话框。请确保选择了正确的存储过程名称，并且返回类型设置为Patient类型的Entities，然后单击OK按钮。

通过ADO.NET实体数据模型即可访问存储过程，您现在需要通过WCF数据服务将其公开。这可以通过定义具有WebGet属性的服务方法而完成。在这里您将定义两种此类方法来执行之前讨论的更为复杂的查询。请注意，第一个方法使用您所定义的存储过程返回至少拥有一个预约的患者。第二个方法接受两个参数来定义搜索区域的中心。距离指定位置一定距离内的患者被返回。此外，您已经扩展了 InitializeService方法，以允许访问新创建的服务操作。

现在您可以从Windows Phone应用程序中查询这些方法。因为自定义方法没有强类型支持，所以没有必要重新生成代理类。为了调用这些自定义的方法，您可以使用CreateQuery或BeginExecute方法。BeginExecute方法接受一个URI作为参数，允许指定URI映射到您自定义的方法。在下面的示例中，URI指向PatientsWithAppointments方法：

entities.BeginExecute < Patient > (

new Uri(“http://localhost/ServicesApplication/PatientDataService.svc/” +

“PatientsWithAppointments”), Patients\_Callback,null);

*DataServicesPage.xaml.cs中的代码段*

可以使用CreateQuery方法通过编程的方式构建查询URI，您只需要传入想要调用的方法名称以及任何参数的名称和值即可。在这里您将调用 PatientsInRegion方法，传入纬度和经度值。生成的URI将在查询字符串中包含这些参数。

var query = entities.CreateQuery < Patient > (“PatientsInRegion”)

.AddQueryOption(“latitude”, -37.998352)

.AddQueryOption(“longitude”, 145.083596);

query.BeginExecute(Patients\_Callback, query);

//http://localhost/ServicesApplication/PatientDataService.svc/PatientsInRegion()? latitude=-37.998352 & longitude=145.083596

*DataServicesPage.xaml.cs 中的代码段*

**4 更新、插入与删除**

使用由WCF数据服务返回的数据相对比较简单，并在一定程度上可以被视为在服务器上直接使用ADO.NET实体数据模型。如果您修改客户端上的一个实体，直到您调用UpdateObject然后再调用 BeginSaveChanges后才会提交更改。

private void SavePatientButton\_Click(object sender, RoutedEventArgs e){

var p = this.PatientsList.SelectedItem as Patient;

p.FirstName = “New Name”;

entities.UpdateObject(p);

entities.BeginSaveChanges(SaveComplete\_Callback, null);

}

private void SaveComplete\_Callback(IAsyncResult result){

var resp = entities.EndSaveChanges(result);

}

*DataServicesPage.xaml.cs中的代码段*

若要插入一个新的实体，您只需创建entity类的一个新实例，并设置相应的属性即可。对于每个实体类型都有一个对应的add方法，在本例中即AddToPatients。若要向服务器提交更改，则需调用BeginSaveChanges 方法：

var p = new Patient(){

PatientId = Guid.NewGuid(),

FirstName = “Nick”,

LastName = “Randolph”,

PhoneNumber = “00 00 00 0000”,

Doctor = “Fred”,

Latitude = -32.0,

Longitude = 151.0

};

entities.AddToPatients(p);

entities.BeginSaveChanges(SaveComplete\_Callback, null);

*DataServicesPage.xaml.cs 中的代码段*

通过DeleteObject方法，然后调用BeginSaveChanges来删除一个实体：

private void SavePatientButton\_Click(object sender, RoutedEventArgs e){

var p = this.PatientsList.SelectedItem as Patient;

entities.DeleteObject(p);

entities.BeginSaveChanges(SaveComplete\_Callback, null);

}

*DataServicesPage.xaml.cs 中的代码段*

**14.3.2 JSON**

WCF数据服务一个有趣的地方是您的数据不仅可以使用结构化的XML格式发布，还可以通过基于JSON的格式发布。这是一个更为紧凑的表示法，可以显著地减少大型数据集所消耗的带宽量。这是用于移动应用程序的一种理想格式。表14-1说明了XML和JSON格式之间的尺寸差异。正如您所看到的，表示相同数据集时JSON格式比XML表示形式的一半还要少。

遗憾的是，这些代理类并不支持使用JSON格式化的数据，这意味着您将需要使用HttpWebRequest类创建您自己的请求，并且需要手动地将响应解析成可以使用的对象。创建请求类似于您在本章前面所看到的，只需简单地指定您要访问的URI即可。唯一的区别是您需要指定Accept HTTP标头，以便服务器以JSON的形式发送响应，而非XML。

private void GetPatientsInJsonButton\_Click(object sender, RoutedEventArgs e){

var request =HttpWebRequest.Create(

“http://localhost/ServicesApplication/PatientDataService.svc/Patients”)

as HttpWebRequest;

request.Accept = “application/json”;

request.BeginGetResponse(JsonPatients\_Callback, request);

}

*DataServicesPage.xaml.cs中的代码段*

虽然您可以手动解析返回的JSON数据，不过使用位于System.ServiceModel.Web程序集内的 DataContractJsonSerializer类则是一种更为简单的形式。它会解析一个JSON流，并返回在其中发现的对象。为此，它需要知道哪种类型的对象可以在流中找到。下面的类声明反映了通过服务器返回的JSON结构：

public class PatientData{

public JsonPatient[] d { get; set; }

}

public class JsonPatient{

public MetaData \_\_metadata { get; set; }

public string PatientId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Latitude { get; set; }

public string Longitude { get; set; }

public string PhoneNumber { get; set; }

public DeferredItem Appointments { get; set; }

public DeferredItem Notes { get; set; }

}

public class MetaData{

public string uri { get; set; }

public string type { get; set; }

}

public class DeferredItem{

public Deferred \_\_deferred { get; set; } }

public class Deferred{

public string uri { get; set; }

}

*DataServicesPage.xaml.cs 中的代码段*

有了这些类声明，您就可以使用DataContractJsonSerializer从响应中提取对象图（object graph）了。

private void JsonPatients\_Callback(IAsyncResult result)

{

var request = result.AsyncState as HttpWebRequest;

var response = request.EndGetResponse(result);

var deserializer = new DataContractJsonSerializer(typeof(PatientData));

var data = deserializer.ReadObject(response.GetResponseStream()) as PatientData;

this.Dispatcher.BeginInvoke(() = >

{

this.PatientsList.ItemsSource = data.d;

});

}

*DataServicesPage.xaml.cs 中的代码段*

在图14-16中您可以看到JSON以原始格式（左图）以及将解析的对象设置为列表框的源（右图）时的显示方式。

DataContractJsonSerializer 类位于System.Runtime.Serialization .Json 命名空间内。I要使用该类，您需要在应用程序中添加对 System.ServiceModel.Web.dll 的引用。

**14.4 本章总结**

在本章中，您了解到了如何通过WebClient和HttpWebRequest类访问您的Windows Phone应用程序内的远程数据。尽管它们对处理远程数据提供最低级别的机制，但是使用基于SOAP或REST的API提供的结构化数据却很难。WCF和ASMX Web Services均提供一个高级抽象来调用远程服务，而WCF数据服务为任何想要从远程数据源执行CRUD样式操作的Windows Phone应用程序提供了一个非常棒的起点。JSON内容类型和IIS压缩一同使用，可以产生一个带宽最优化的解决方案。