
Delphi DataSnap
开发手册
embarcadero

目录

本书的范例程序请至下列网页下载：

http://embarcadero.qcomgroup.com.tw/download/dds.zip

第1章 开发 DataSnap/RESTful应用系统

Delphi/BCB 的 DataSnap 技术早期称为 Midas，Midas 主要是使用来开发多层分布式应用系统的技术，它是以 COM 技术为核心发展出来的多层框架，随着 PC 信息技术不断的演进，DataSnap 也从基于 COM 核心的框架发展到以 JSON 和 REST 技术的分布式架构。使用 JSON 和 REST 架构做为 DataSnap 核心技术的好处是 JSON 和 REST 都是可以跨平台的标准，JSON 提供了跨平台交换数据的标准格式，而 REST 更是基于 HTTP 通讯协议的技术。因此使用 Delphi 10.3 开发的 DataSnap/RESTful 服务器虽然是执行于 Windows 平台，但却可以服务所有平台的客户端的呼叫和请求。

从本章开始我们将讨论如何使用 Delphi 10.3 开发基于 JSON 和 REST 技术的分布式像应用系统，我们将从如何开发 DataSnap/RESTful 服务器，DataSnap/RESTful 客户端谈起，然后一直讨论到进阶的 DataSnap 技术，现在让我们从如何开发 DataSnap/RESTful 服务器说起。

1-1 开发 DataSnap/RESTful 服务服务器

在 Delphi 整合发展环境中点选 File|New|Other...启动 New Items 对话盒并且在 DataSnap Server 选项中选择建立 DataSnap Server 图像，如下图所示:

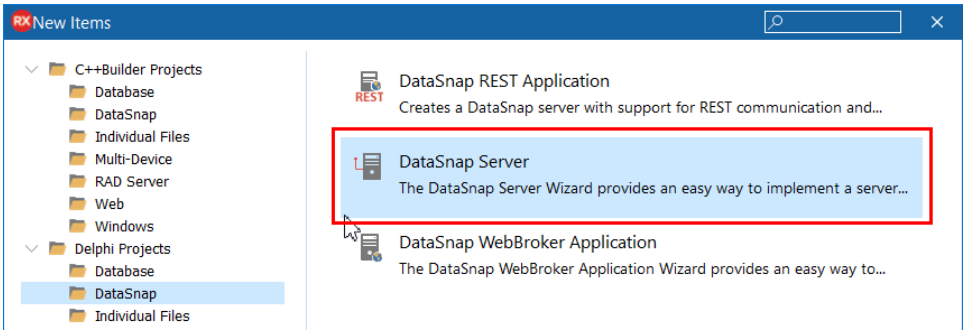
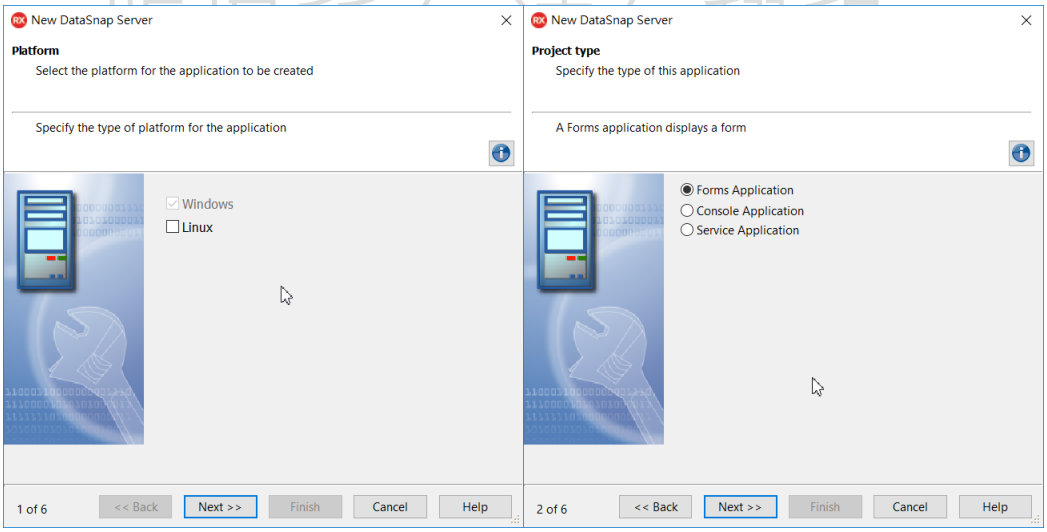


图 1-1 建立 DataSnap 服务器应用程序

点选了 DataSnap Server 图像之后 Delphi 会开始询问开发人员如何设定此 DataSnap 服务器。首先 DataSnap 精灵会询问要建立什么型态的 DataSnap 服务器，Delphi 10.3 提供三种不同的服务器型态，分别是:

服务器型态	说明
VCL Forms Application	使用VCL图形用户接口框架的应用程序
Console Application	主控程序型态的服务器
Service Application	Windows服务型态的服务器

为了说明方便起见，让我们在这选择建立 VCL 窗体型态的服务器，如下所示:



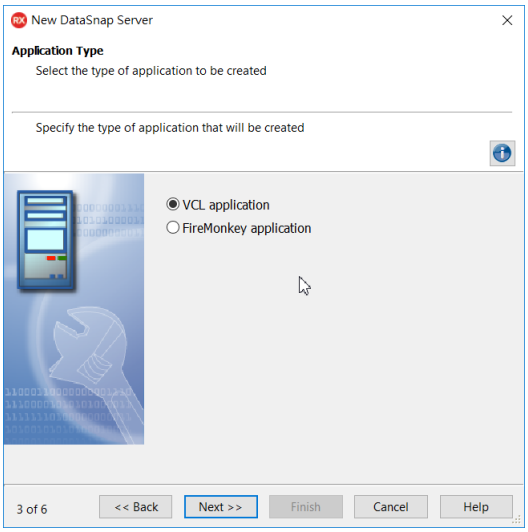


图 1-2 选择建立 DataSnap 服务器的型态

接着 DataSnap 精灵会询问自动建立的服务器包含的基础功能，例如使用什么通讯协议?是否需要使用安全认证功能? 是否需要建立范例服务方法? 在这里让我们点选对话盒下方的 Select all 勾选盒要求建立所有的基础功能，如下图所示:

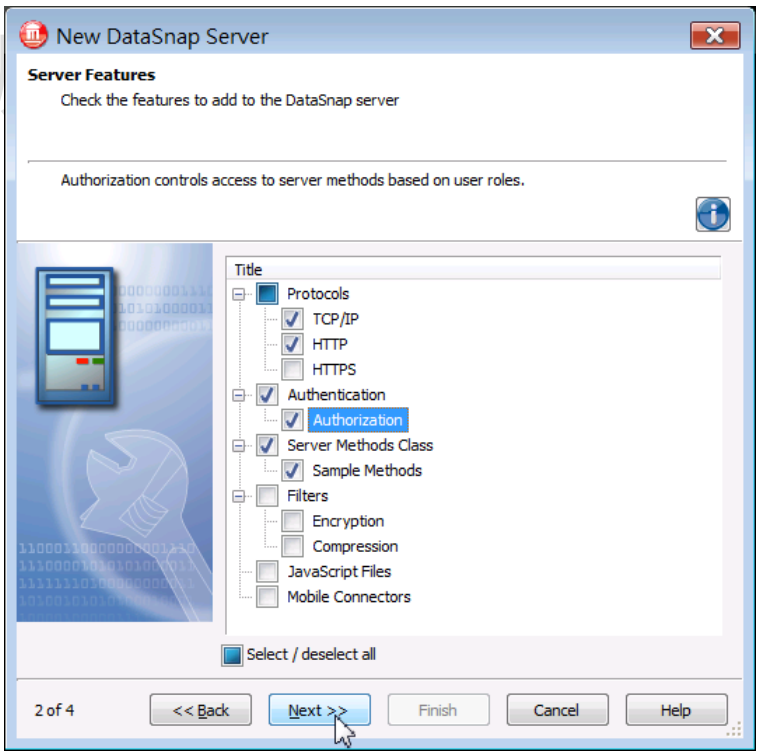


图 1-3 选择自动建立的服务器功能

图 1-3 中的 Filters，JavaScript Files 和 Mobile Connectors 等功能会在稍后的章节中说明，读者现在可以暂时不用担心这些选项。

接着 DataSnap 精灵会询问在上一步骤选择使用的通讯协议的通信埠，在内定上 TCP/IP 通讯协议是使用通信埠 211，HTTP 通讯协议是使用通信埠 8080，开发人员可以设定其他的通信埠，或是点选对话盒中 Find Open Port 按钮让 DataSnap 精灵自动搜寻其他可使用的通信埠，如下图所示:

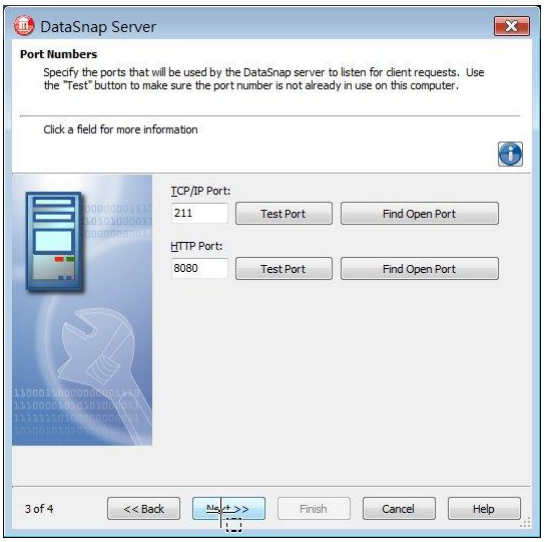


图 1-4 选择使用的通信埠

最后 DataSnap 精灵会询问服务器输出服务的父代类别，可以做为父代类别的分别是:

父代类别	说明
TComponent	如果开发人员想使用最少的资源输出服务方法，就选择此类别
TDataModule	如果开发人员习惯使用数据模块或是需要把旧的数据模块升级，就可以选择这个类别
TDSServerModule	最典型的父代类别，如果开发人员没有特别的升级考虑或是建立新的DataSnap服务器，那么请尽量选择使用这个类别。

在这里让我们选择使用 TDSServerModule 如下图所示:

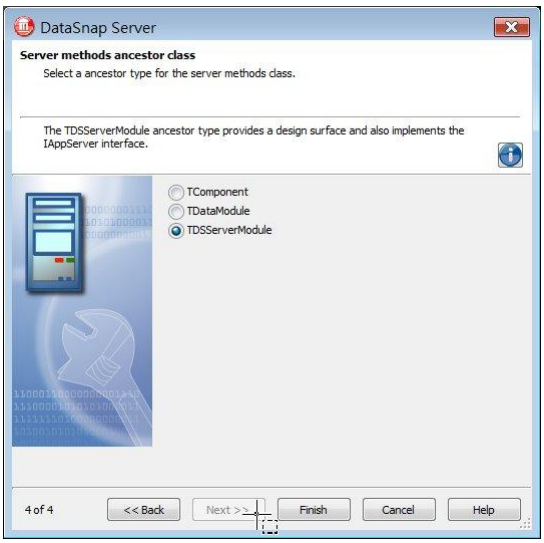


图 1-5 选择服务器服务类别的父代类别

下表列出了选择使用不同的父代类别所产生的程序代码差异，例如如果选择使用 TComponent 做为父代类别，那么就需要使用**{$METHODINFO ON}**和**{$METHODINFO OFF}**这两个编译程序指令，因为使用这两个编译程序指令的类别才会由 Delphi 编译程序自动产生 RTTI 信息以自动输出类别中的方法。

使用 TDSServerModule 父代类别	使用 TComponent 父代类别

TServerMethods5 = class(TDSServerModule)
private
{ Private declarations }
public
{ Public declarations }
function EchoString(Value: string): string;
function ReverseString(Value: string): string;
end;
	{$METHODINFO ON}
	TServerMethods5 = class(TComponent)
	private
	{ Private declarations }
	public
	{ Public declarations }
	function EchoString(Value: string): string;
	function ReverseString(Value: string): string;
	end;
	{$METHODINFO OFF}

点选 Finish 按钮之后，DataSnap 精灵便会自动帮助我们根据刚才在 DataSnap 精灵中进行的设定来产生项目，例如在下面的项目经理中我们把这个项目命名为 pDataSnapDemoServer，并且分别储存其中的文件名如下所示:

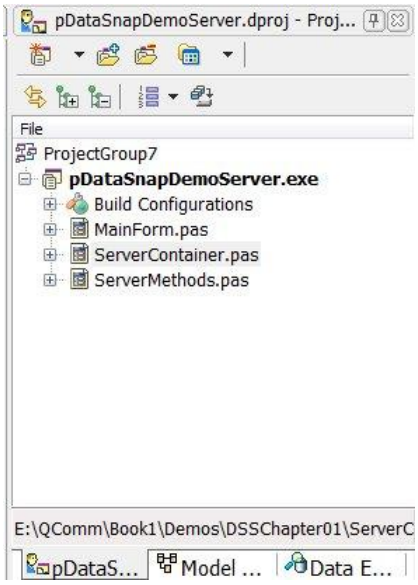


图 1-6 DataSnap 精灵自动建立的项目

如果现在我们开启项目中的 ServerContainer 程序单元就可以看到类似下图的结果，在 ServerContainer 中包含了数个不同的组件，这些组件的用途如下所述:

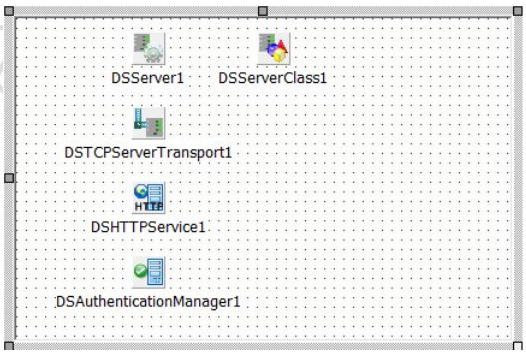


图 1-7 DataSnap 精灵自动建立的 ServerContainer 模块中包含的组件

组件名称	组件类别	意义
DSServer1	TDSServer	提供DataSnap服务器基本功能
DSTCPServerTransport1	TDSTCPServerTransport	提供TCP/IP通讯协议的支持
DSHTTPService1	TDSHTTPService	提供HTTP通讯协议和RESTful架构的基本支持
DSAuthenticationManager1	TDSAuthenticationManager	提供安全认证功能
DSServerClass1	TDSServerClass	提供自动输出伺

		服端类别让客户端呼叫的基本功能

好了，在 Delphi 10.3 自动产生了项目之后，基本上现在这个项目已经提供了 TCP/IP 以及 RESTful 的服务功能了，不信吗?我们现在就可以测试一下这个自动产生的项目。

前面说过 TDSServerClass 类别中定义的方法都可以自动输出并且被客户端呼叫，由于现在项目中的 DSServerClass1 定义了 EchoString 和 ReverseString 这两个范例方法，因此我们自然应该可以呼叫它们。因此 4 现在让我们编译并且执行此服务器，然后启动浏览器(笔者使用 Google 的 Chrome)，在浏览器中输入如下的 URL:

```
http://localhost:8080/DataSnap/rest/TServerMethods5/ReverseString/嗨, 大家好!
```

因为这个 DataSnap 服务器也是一个 REST 服务器，因此我们可以使用标准的 REST 语法架构来呼叫它的服务。让我们拆解一下上面 URL 的意义。

由于 REST 是使用 HTTP 通讯协议而且前面我们在建立此 DataSnap 服务器时选择使用了 HTTP 以及 8080 通信埠，因此 http://localhost:8080 就代表要使用 HTTP 通讯协议连结(请求)本机中位于 8080 通信埠的服务提供商。『DataSnap/』是 TDSHTTPService 组件使用的 DSContent 特性值，而『rest/』则是它使用的 RESTContext 特性值，这两个数值都可以藉由设定 TDSHTTPService 组件的 DSContent 和 RESTContext 特性来改变。TServerMethods5 则是服务器中提供输出方法的类别，由于我们要呼叫 ReverseString 这个方法，因此使用了『/ReverseString』。最后的『/嗨，大家好!』则是传递给 ReverseString 方法的参数，因为 ReverseString 方法定义接受一个字符串型态的参数，例如下面就是 ReverseString 的定义，我们可以看到它接受一个命为 Value 的参数，因此『/嗨，大家好!』就会设定成传递给 ReverseString 的 Value 参数值。


```
function ReverseString(Value: string): string;
```

例如下图就是在 Chrome 中呼叫 DataSnap 服务器的结果，证明了它的确支持 RESTful 架构并且使用 JSON 格式传递数据:

图 1-8 使用浏览器呼叫 DataSnap 服务器中的范例方法

那么，现在读者是不是知道了如何在浏览器中呼叫 EchoString 呢？没错，只要使用下面的 URL 在浏览器中就可以呼叫 EchoString 方法了。

```
http://localhost:8080/DataSnap/rest/TServerMethods5/EchoString/嗨，大家好!
```

如何，是不是很简单呢？由于支持了 REST 架构，因此任何支持 HTTP 的客户端都可以呼叫 DataSnap 服务器中的服务了。

现在让我们继续为这个 DataSnap 服务器加入一些有用的功能。让我们看看如何让 DataSnap 输出数据到传统的 Delphi Win32 客户端以便开发分布式数据库应用系统。

首先开启项目中的 ServerMethods 程序单元，由于它是从 TDSServerClass 类别继承下来的，因此任何在其中宣告，定义的方法和资料都可以输出让客户端呼叫，现在让我们使用 dbExpress 的 TSQLConnection，TSQLDataSet，TDataSetProvider 和 TClientDataSet 组件链接到数据，如下所示(有关 dbExpress 程序设计的主题，请参考 dbExpress 相关的书籍):

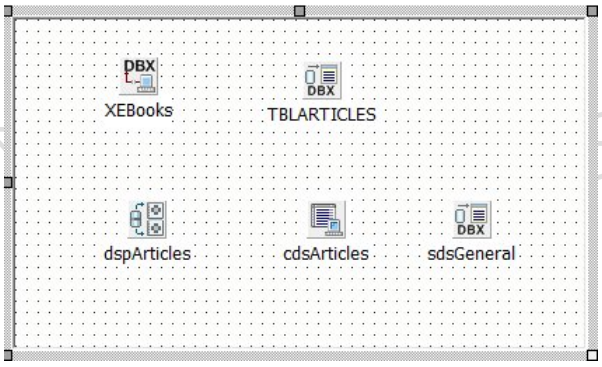


图 1-9 在 ServerMethods 程序单元中使用 dbExpress 组件链接数据库

在说明如何开发客户端之前，再让我们为服务器定义两个方法，这个两个方法除了在稍后会被客户端呼叫之外，稍后也可以再次使用浏览器来呼叫它们以证明这个 DataSnap 服务器的确支持 ERSTful 架构并且使用 JSON 封装和传递数据。

首先让我们在输出类别 TServerMethods5 中再定义 GetArticleID 方法，它使用 TSQLDataSet 组件执行 SQL 叙述以便取得 Articles 数据表中最大笔数加 1 的数值并且回传到客户端:

```
function TServerMethods5.GetArticleID: Integer;
begin
  sdsGeneral.Active := False;
  try
```

```
    sdsGeneral.CommandText := 'select count(*) from TBLARTICLES';
    sdsGeneral.Active := True;
    Result := sdsGeneral.Fields[0].AsInteger + 1;
  finally
    sdsGeneral.Active := False;
  end;
end;
```

现在我们就可以使用下面的 URL 在浏览器中呼叫 GetArticleID，

```
http://localhost:8080/DataSnap/Rest/TserverMethods5/GetArticleID
```

由于笔者在使用浏览器呼叫 GetArticleID 方法时，在 Articles 数据表中已经存在了 2 笔数据，因此在下面的浏览器截图中可以清楚的看到 GetArticleID 方法果然以 JSON 格式封装数值『3』回传到客户端:

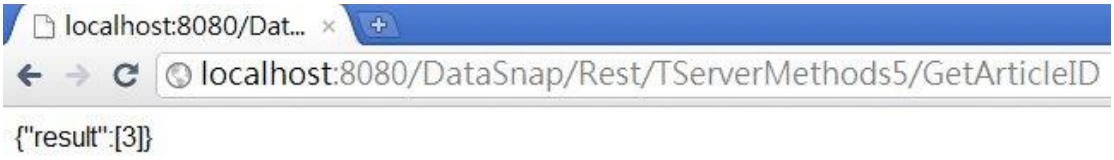


图 1-10 在浏览器中呼叫 GetArticleID 方法

第二个方法是则是使用 JSON 数组回传所有 Articles 数据表中文章名称的 QueryAllArticles 方法，下面是 QueryAllArticles 的实作:

```
001   Sfunction TServerMethods5.QueryAllArticles: TJSONArray;
002   var
003     ja : TJSONArray;
004     aBK : TBookmark;
005   begin
006     if (not cdsArticles.Active)  then
007       cdsArticles.Active := True;
008
009     ja := TJSONArray.Create;
010     try
011       aBK := cdsArticles.GetBookmark;
012       cdsArticles.First;
013       while (not cdsArticles.Eof) do
014       begin
015
ja.AddElement(TJSONString.Create(cdsArticles.FieldByName('NAME').AsString));
```

```
016         cdsArticles.Next;
017       end;
018     finally
019       cdsArticles.GotoBookmark(aBK);
020       cdsArticles.FreeBookmark(aBK);
021     end;
022     Result := ja;
023   end;
```

在 009 行先建立一个 TJSONArray 对象，011 行先使用 TBookMark 对象保留 cdsArticles 目前的记录位置，013 行进入 while 循环从第一笔记录取得 Name 字段的数值并且以 TJSONString 对象封装并且储存在 TJSONArray 对象中，一直到最后一笔记录。最后 019 行使用 TBookMark 对象让 cdsArticles 回到呼叫 QueryAllArticles 方法之前的位置，020 行释放 TBookMark 对象，最后在 022 行回传 TJSONArray 对象到客户端。

现在同样使用浏览器并且在 URL 位置使用:

```
http://localhost:8080/DataSnap/Rest/TserverMethods5/QueryAllArticles
```

呼叫 QueryAllArticles 方法，在下面的截图中可以看到我们果然可以在浏览器中成功呼叫 QueryAllArticles 方法，QueryAllArticles 方法是以正确的 JSON 数组封装并且回传所有的文章名称，而且文章名称都是以 Unicode 编码的:

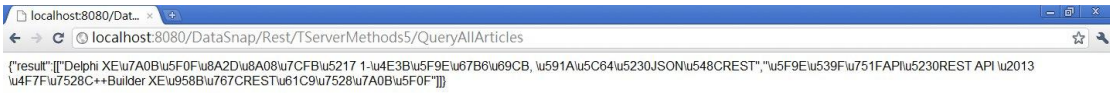


图 1-11 在浏览器中呼叫 QueryAllArticles 方法

由上面两个实作方法证明了 Delphi 10.3 的 DataSnap 能够使用 JSON 格式回传任何的数据，只要这些数据使用 JSON 规范来封装即可。

现在我们已经完成了第一个范例 DataSnap/RESTful 服务器，现在请编译并且执行这个 DataSnap/RESTful 服务器，因为接着我们要说明如何开发客户端应用程序来呼叫这个 DataSnap/RESTful 服务器提供的服务。

1-2 开发客户端 Win32 应用程序

现在回到项目群组，在其中建立一个新的 VCL Form 应用程序项目，并且命名它为 pDataSnapDemoClient，如下图所示:

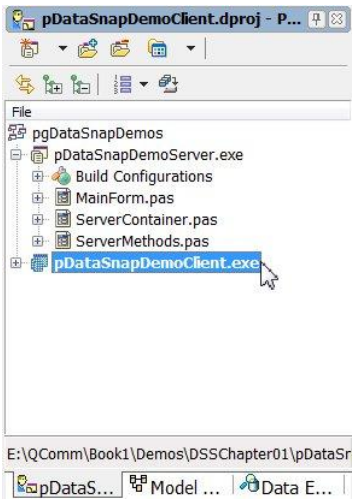


图 1-12 在项目中建立一个 VCL Form 应用程序项目做为客户端应用程序

Delphi 提供了数种方式让客户端应用程序链接和呼叫 DataSnap 服务器，其中最简单，方便的方式就是使用 DataSnap 客户端精灵。现在就让我们使用它来自动在客户端项目中建立 DataSnap 客户端模块以链接和呼叫 DataSnap 服务器。

点选 File|New|Other...菜单启动 New Items 对话盒，在 DataSnap Server 项目中点选 DataSnap Client Module 图像以便在客户端应用程序项目中建立 DataSnap 客户端模块，如下图所示:

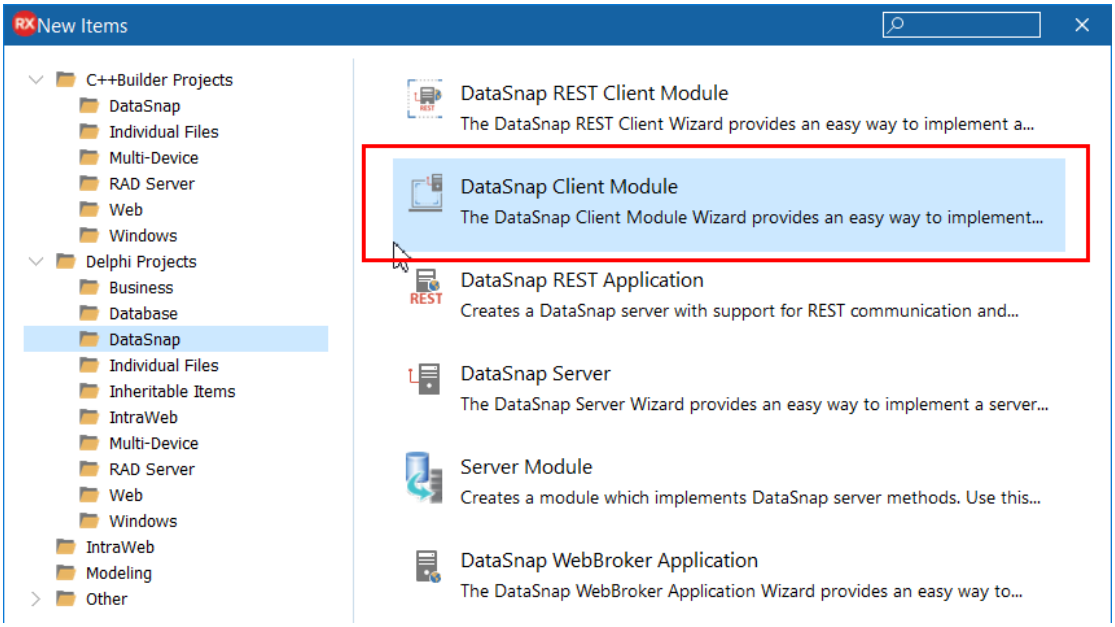


图 1-13 在客户端应用程序中建立 DataSnap Client Module

点选 DataSnap Client Module 图像之后 DataSnap Client Module 精灵会询问有关 DataSnap 服务器的特性以便进行对应的设定，首先会询问要链接

的 DataSnap 服务器的位置，由于我们的范例 DataSnap 服务器是于本机中，因此我们选择 Local Server:

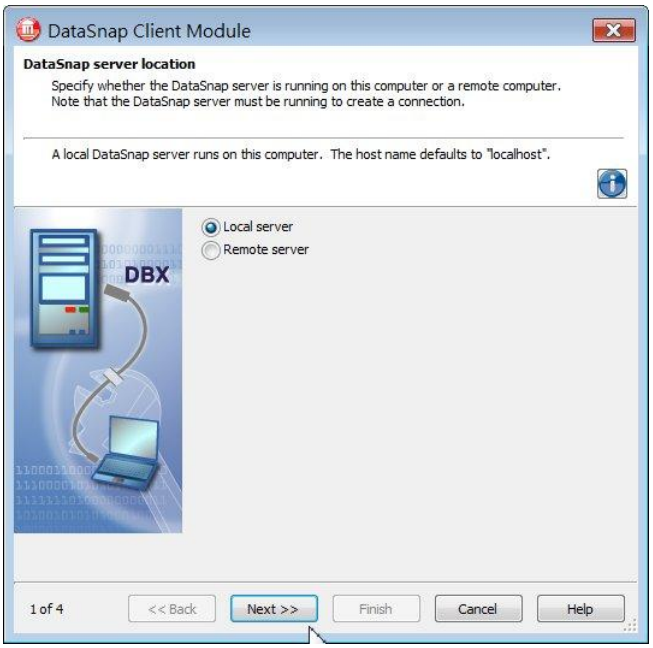


图 1-14 设定 DataSnap 服务器的特性

接着 DataSnap Client Module 精灵会询问要链接的 DataSnap 服务器的型态，由于在前面我们是开发 VCL Form 应用程序型态的服务器，因此在这里我们选择 DataSnap stand alone server 型态:

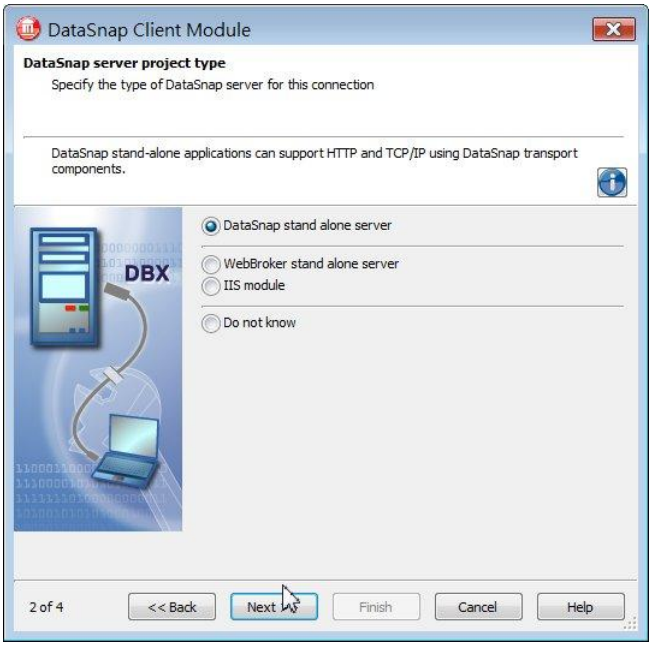


图 1-15 设定 DataSnap 服务器的型态

接着 DataSnap Client Module 精灵会询问要使用什么通讯协议连结 DataSnap 服务器，由于现在我们是开发原生 Win32 客户端应用程序，因此让我们选择使用 TCP/IP 通讯协议：

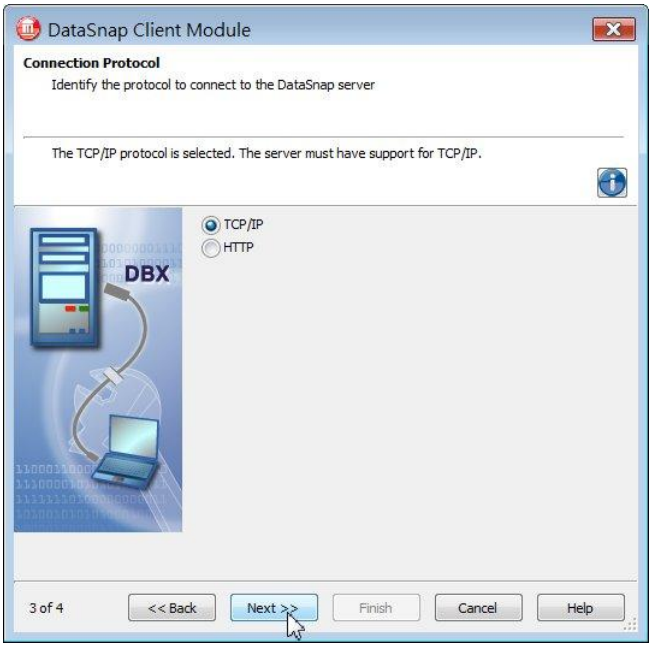


图 1-16 选择连结 DataSnap 服务器的通讯协议

接着 DataSnap Client Module 精灵会询问使用通信端口以及服务器的登录信息，我们使用 TCP/IP 而且 DataSnap 服务器是使用内定通信埠 211，因此在下面的 Port 选项中设定 211，由于我们尚未说明如何使用 DataSnap 的安全认证机制，因此在这里可以先不用管 User name 和 Password 的选项设定：

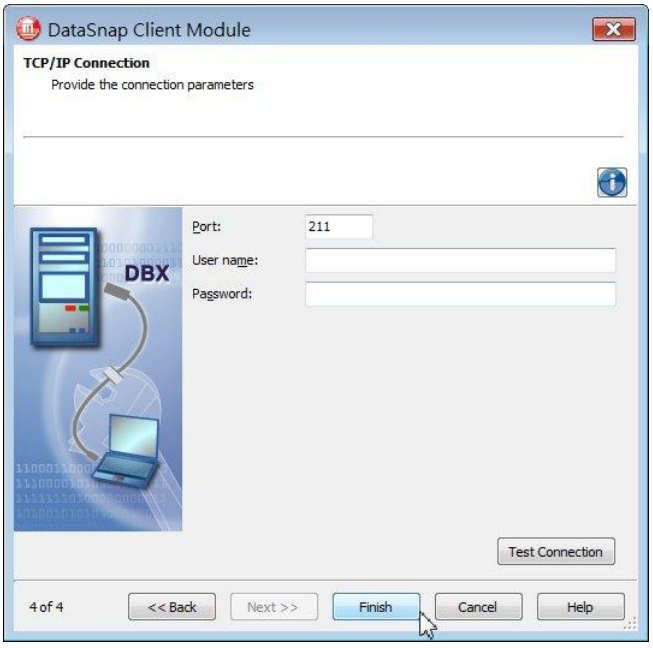


图 1-17 设定通讯协议的通信端口以及登录信息

点选 Finish 按钮之后让我们储存这个 DataSnap Client Module 为 ClientModule.pas，并且储存 DataSnap Client Module 精灵产生的原始程序为 ServerProxy.pas，此时项目管理员应该类似如下图所示：

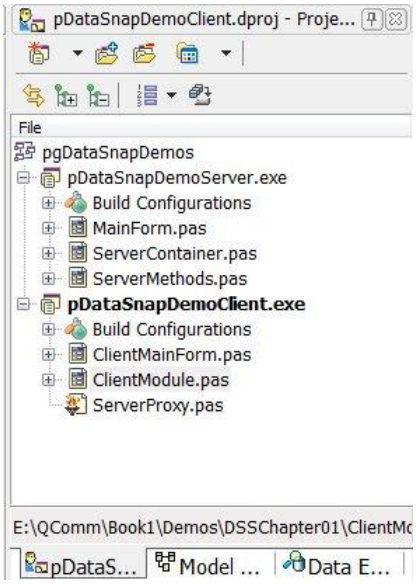


图 1-18 项目管理员

如果我们开启 ClientModule 就会看到 DataSnap Client Module 精灵在 DataSnap Client Module 中产生的 TSQLConnection 组件，如下图所示，笔者已经将这个 TSQLConnection 组件命名为 sqlcnDataSnapServer，请注意在对象查看器中它的 Driver 已经被设定为 DataSnap 了：

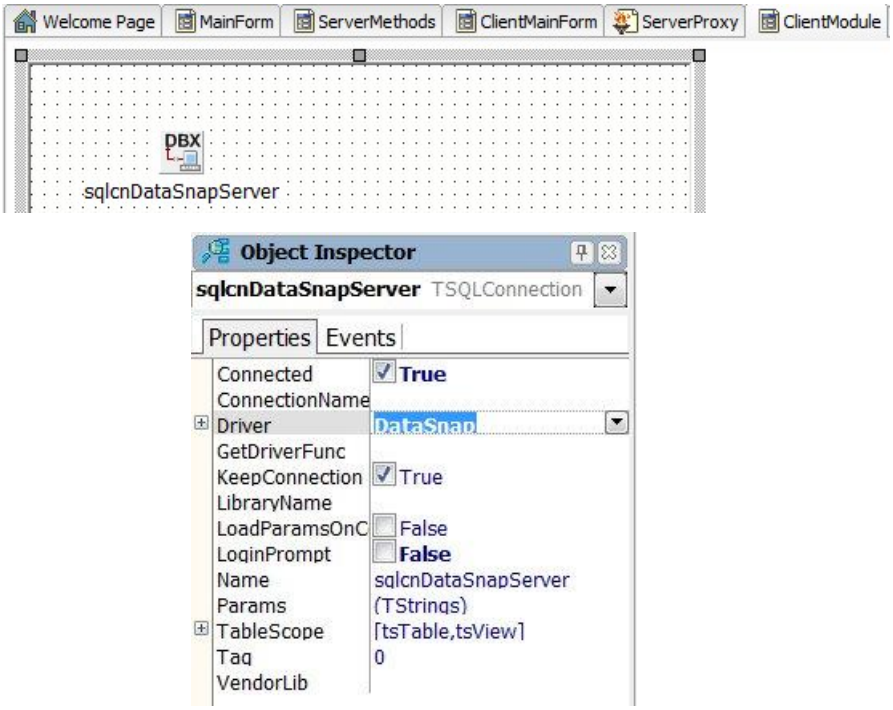


图 1-19 DataSnap Client Module 精灵在 DataSnap Client Module 中产生的 TSQLConnection 组件

现在如果范例 DataSnap 服务器已经在执行中，那么我们可以到整合发展环境的 Data Explorer 中，连结此 DataSnap 服务器，例如在下图中我们于 Data Explorer 中对 DATASNAP 选项中的 LOCAL SEVER 进行设定之后就可以链接到范例 DataSnap 服务器:

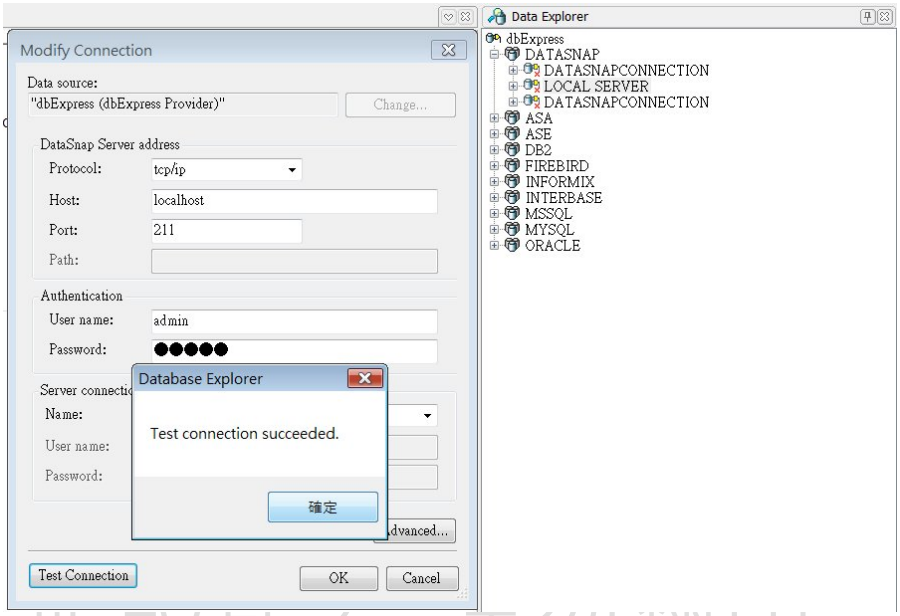


图 1-120 使用 Data Explorer 链接范例 DataSnap 服务器

如果我们再开启 LOCAL SEVER，就可以看到类似下图的结果，我们果然可以在 LOCAL SERVER 的 Procedures 选项中看到 TServerMethod5 输出的方法，对于熟悉 Midas 技术的读者来说，请注意 TServerMethod5 甚至输出了远程数据模块(TRemoteDataModule)的 IAppServer 接口之中的方法，这也代表旧的 Midas/DataSnap 应用系统也可以很容易的升级到新的基于 REST/JSON 架构的 DataSnap 应用系统。

图 1-19 在 Data Explorer 中检视范例 DataSnap 服务器输出的服务方法

OK，现在我们就可以开始实作客户端应用程序了，首先让我们看看如何从 DataSnap 服务器中取得文章数据表中的数据。在 dbExpress 程序中我们知道使用 TDataSetProvider 组件链接 TSQLDataSet，再使用 TClientDataSet 就可以取得资料。但当我们要连结基于 JSON/REST 的 DataSnap 服务器时，我们需要使用 TDSProviderConnection 组件链接远程 DataSnap 服务器中输出方法的服务，如果远程 DataSnap 服务器中提供了 TDataSetProvider 组件，那么就可以使用 TClientDataSet 组件藉由 TDSProviderConnection 组件取得远程的 TDataSetProvider 组件了。

因此现在让我们在客户端的 DataSnap ClientModule 中放入 TDSProviderConnection 和 TClientDataSet 组件，如下图所示：

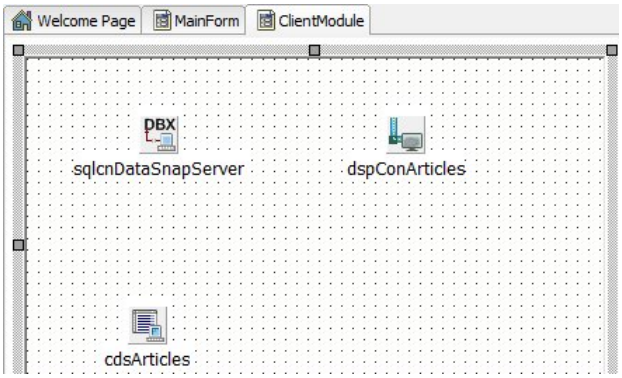


图 1-20 在 Client Module 中加入 TDSProviderConnection 和 TClientDataSet 组件

接着设定 TDSProviderConnection 的特性如下:

特性	特性值
ServerClassName	TServerMethods5
SQLConnection	SqlcnDataSnapServer
Name	dspConArticles

设定 TDSProviderConnection 组件的 ServerClassName 特性值为 TServerMethods5 是因为 DataSnap 服务器中输出服务方法的类别就是 TServerMethods5。

接着设定 TClientDataSet 的特性如下:

特性	特性值
RemoteServer	dspConArticles
ProviderName	dspArticles
Name	cdsArticles

在设定 cdsArticles 时，一旦设定了 RemoteServer 特性值，那么当我们设定 ProviderName 特性值时会感觉到对象查看器暂停了一下，之后可以设定的特性值会出现在下拉盒中，这是因为此时 Delphi 整合发展环境就藉由 TDSProviderConnection 连结到 DataSnap 服务器并且搜寻其中输出的 TDataSetProvider 组件，由于前面范例 DataSnap 服务器输出了名为 dspArticles 的 TDataSetProvider 组件，因此在 ProviderName 特性下拉盒中会出现 dspArticles。

现在于客户端主窗体中使用 TDataSource 组件链接 cdsArticles，就再设定 cdsArticles 的 Active 特性值为 True 之后，就可以看到远程的 Article 数据表的数据果然出现在客户端的应用程序中了，如下图所示:

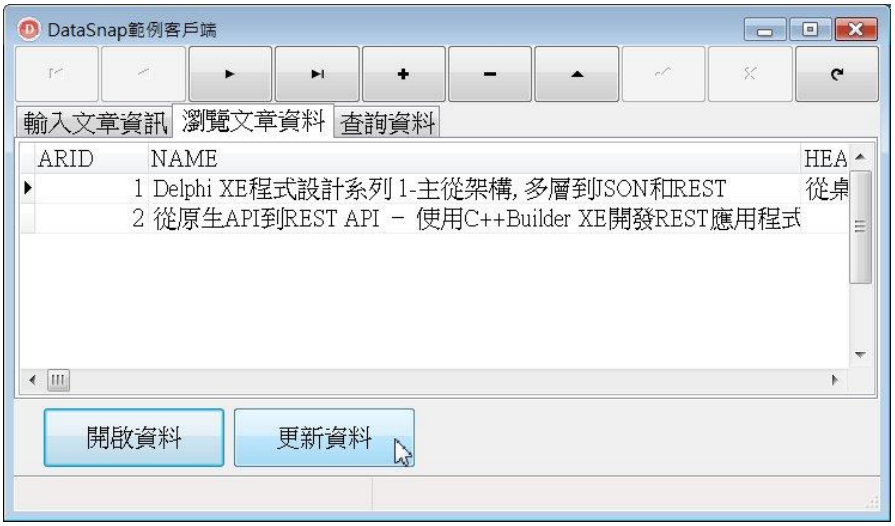


图 1-25 在 Client Module 中加入 TDSProviderConnection 和 TClientDataSet 组件

那么我们如何在客户端应用程序对于 JSON/REST 的 DataSnap 服务器进行数据的异动呢?事实上这就是使用 dbExpress 和 DataSnap 的好处了，我们仍然可以使用 dbExpress 的方式对于 JSON/REST 的 DataSnap 服务器进行数据的异动。

例如现在如果我们要在客户端新增资料到 JSON/REST 的 DataSnap 服务器，那么我们可以首先在 Client Module 中为 cdsArticles 定义 AfterInsert 事件处理函式，在 AfterInsert 事件处理函式中我们自动为新增资料的 ARID 和 ADate 字段产生域值，ARID 是文章的序号而 ADate 则是文章新增到数据表的日期:


```
procedure TcmArticles.cdsArticlesAfterInsert(DataSet: TDataSet);
begin
  cdsArticles.FieldByName('ARID').Value := GetArticleID;
  cdsArticles.FieldByName('ADate').Value := Now;
end;
```

接着我们就可以在主窗体的『更新数据』按钮的 OnClick 事件处理函式中呼叫 cdsArticles 的 ApplyUpdates 方法更新回 JSON/REST 的 DataSnap 服务器即可:

```
procedure TMainForm.btnApplyUpuetsClick(Sender: TObject);
begin
  cmArticles.cdsArticles.ApplyUpdates(0);
  UpdateStatusBar(IntToStr(cmArticles.cdsArticles.ChangeCount) + '笔资料被异动了');
end;
```

例如下图就是使用客户端应用程序新增一笔数据回 JSON/REST 的 DataSnap 服务器:

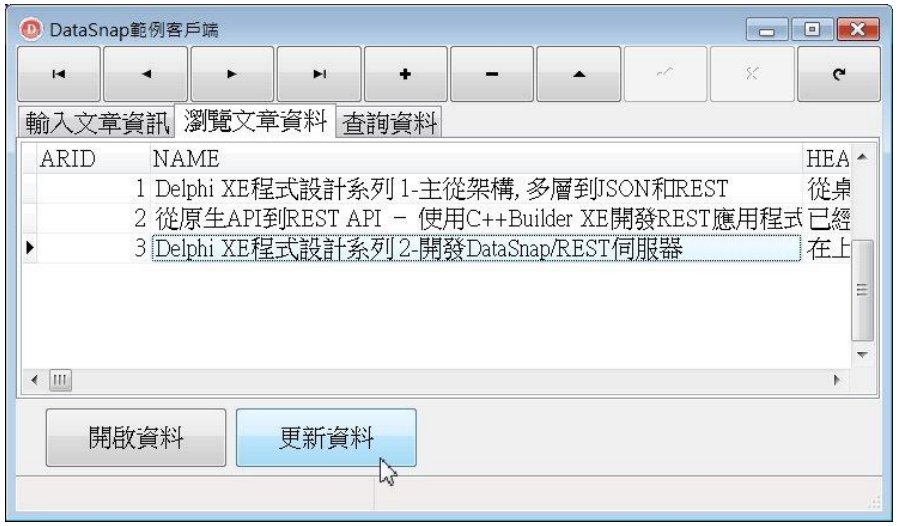


图 1-26 呼叫 TClientDataSet 的 ApplyUpdates 方法更新数据回 JSON/REST 的 DataSnap 服务器

一旦客户端使用 TDSProviderConnection 和 TClientDataSet 组件向 JSON/REST 的 DataSnap 服务器取得数据之后，在客户端它就向是直接使用 dbExpress 取得数据一样，在客户端也可以使用 TClientDataSet 的查询数据能力来搜寻数据，例如下面的程序代码即使用了 TClientDataSet 的 Lookup 方法查询数据:


```
procedure TMainForm.btnQueryARIDClick(Sender: TObject);
begin
  edtARName.Text := cmArticles.cdsArticles.Lookup('ARID', StrToInt(edtARID.Text),
'Name');
end;
```

下图就是查询数据的结果，输入文章 ID 即可查询到相对应的文章。

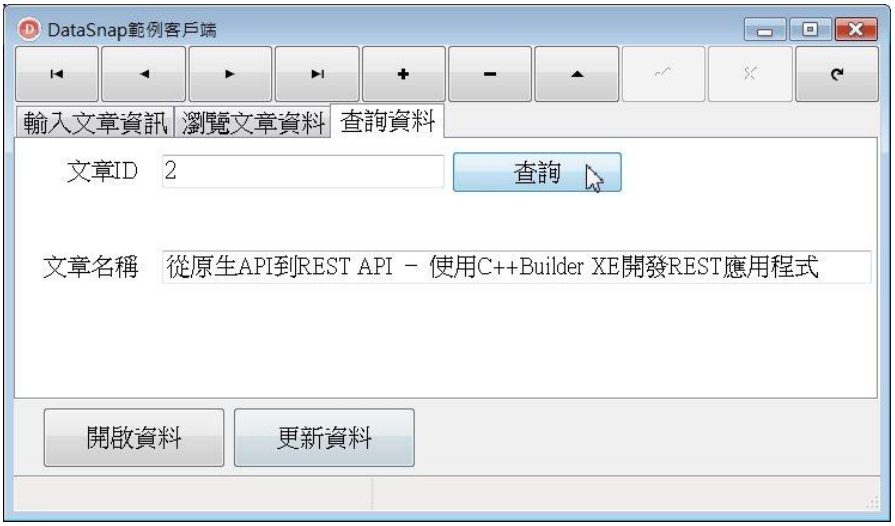


图 1-27 使用 TClientDataSet 的 Lookup 方法查询数据

在这个范例文章数据表中同时有一个 BLOB 型态的字段，它的目的是储存文章的 PDF 电子档案，那么我们又如何从客户端把 PDF 档案储存到 JSON/REST 的 DataSnap 服务器中呢?使用 dbExpress 和 DataSnap 可以免除我们把二进制数据转换成 JSON 格式的麻烦，我们仍然可以要求 TClientDataSet 组件帮助我们完成这个工作，我们只需要呼叫客户端 TClientDataSet 代表 BLOB 字段的对象的 LoadFromFile 方法即可。例如在下面的程序代码中，cdsArticlesCONTENTS 是代表远程 BLOB 字段的 TBlobField 型态的对象，因此我们在决定需要上传那一个 PDF 档案到远程文章数据表之中后，就可以直接呼叫 cdsArticlesCONTENTS 的 LoadFromFile 即可，非常的简单，也是使用传统 dbExpress 程序设计的方式。


```
procedure TMainForm.btnOpenFileClick(Sender: TObject);
begin
  if (odlgopenFile.Execute) then
  begin
    if (cmArticles.cdsArticles.State <> dsEdit) then
      cmArticles.cdsArticles.Edit;
      cmArticles.cdsArticlesCONTENTS.LoadFromFile(odlgopenFile.FileName);
      cmArticles.cdsArticles.Post;
      cmArticles.cdsArticles.ApplyUpdates(0);
  end;
end;
```

最后笔者要提醒的是，如果读者在 JSON/REST 的 DataSnap 服务器中加入了新的方法让客户端呼叫，那么必须在客户端重新产生连结服务器的客户端类别才能够看到新增的方法，要重新产生连结服务器的客户端类别，读者可以到

Client Module 中，右击 TSQLConnection 组件并且从快捷菜单中选择『Generate DataSnap client classes』，如下图所示即可：

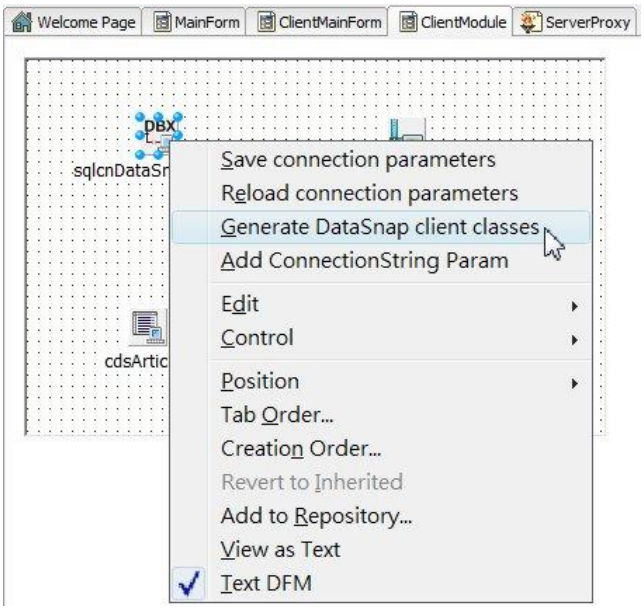


图 1-28 客户端的 TSQLConnection 组件重新产生客户端链接服务器的程序代码

在重新产生了客户端类别之后，就可以在客户端应用程序中使用程序代码来呼叫 JSON/REST 的 DataSnap 服务器新加入的方法了。

其实 Delphi XE 的 DataSnap 框架也提供了自动产生客户端类别的机制，但这属于高阶的 DataSnap 技术，我们会在以后的章节中讨论这个技术。

好了，现在您已经了解了如何开发 DataSnap 服务器和客户端应用程序了，在以后的章节中我们将继续讨论进阶的 DataSnap 技术，现在是您动手开发您自己的 DataSnap 应用系统的时候了。

第2章 JSON程序设计

从 Delphi 2009 开始 Delphi 便开始支持 JSON，DataSnap 2009 开始使用 JSON 做为开发分布式应用系统的基础技术，开始逐渐舍弃使用 COM/DCOM/COM+。由于 JSON 具备跨平台，跨程序语言和跨工具的特性，因此 DataSnap 在结合 JSON 和 dbExpress 之后也逐渐具备了跨平台的基础。但是 Delphi 2009 对于 JSON 的支持只限于使用在 DataSnap 之中，因此如果开发人员需要进行通用的 JSON 程序设计，Delphi 2009 在这方面仍然显得不足。

到了 Delphi 2010 情形开始改变，因为 VCL 框架开始内建支持 JSON 的类别，因此 Delphi 的开发人员就可以利用这些和 JSON 相关的类别来进行 JSON 程序设计的工作。到了 Delphi XE 之后，JSON 已经成为 DataSnap 的核心技术了，因为 Delphi 即将进入跨平台的世代，JSON 具备跨平台的特性自然就成为最佳的数据传递和交换的标准，因此在 Delphi XE 之后的 RTL 以及 DataSnap 中又再次大幅强化了对于 JSON 支持的能力。由于 DataSnap 在分布式和 Web 架构中都使用 JSON 封装数据，此外 dbExpress 也能够处理 JSON 封装的资料，因此 Delphi 的程序员必须了解和掌握 JSON 技术。

本章的重点就是讨论 VCL 框架对于 JSON 的支持，本章将讨论如何使用 VCL 框架中和 JSON 相关的类别来进行 JSON 开发的工作。不过在讨论这些类别之前，我们需要先简单的介绍什么是 JSON。

2-1 JSON 是什么?

简单的说，JSON 是一种数据封装格式以及数据交换格式，JSON 是 JavaScript Object Notation 的缩写，它是一种轻量级的数据交换格式，易于一般人阅读和编写，同时也易于机器解析和生成。

JSON 是基于 JavaScript（Standard ECMA-262 3rd Edition - December 1999）的一个子集，JSON 采用完全独立语言的文字格式，但是也

使用了类似于 C 语言家族的习惯（包括 C, C++, C#, Java, JavaScript, Perl, Python 等），这些特性使 JSON 成为理想的数据交换语言。

JSON 是由下面的两个基本架构组成的:

- “名称/值”对的集合（A collection of name/value pairs）。在不同的程序语言中，这个架构经常以对象（object），纪录（record），结构（struct），字典（dictionary），哈希表（hash table），有键列表（keyed list），或者关联数组 （associative array）来实作。
- 值的有序列表（An ordered list of values）。在大部分的程序语言中，这个架构经常使用数组（array）来实作。

OK，看了上述的定义之后，读者可能还是觉得模模糊糊，也许让我们使用实际的范例来说明可以让读者更容易了解。

假设现在我们有一个 DataSnap 服务器，它使用 JSON 和客户端进行数据交换的工作，那么现在客户端向这个 DataSnap 服务器查询笔者所撰写的书籍，例如本书『实战 Delphi 2010』，那么如何把这个信息以 JSON 的格式传递给客户端呢?

首先让我们观察下图，在 JSON 中对象的代表方式是以“{”开始，以“}”结束，另外在前面我们也说明 JSON 是以『名称/值』的格式来代表资料，因此从下图中我们可以看到名称和值之间是以“：”符号分隔，其中『名称』是字符串格式，而值则是以数值格式来代表:

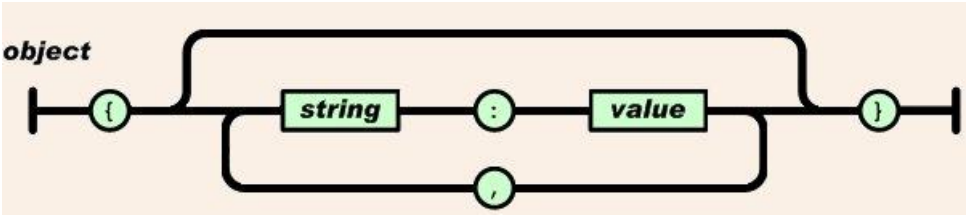


图 1 JSON 封装对象的格式

在 JSON 规范中上图的字符串规范如下图所示:

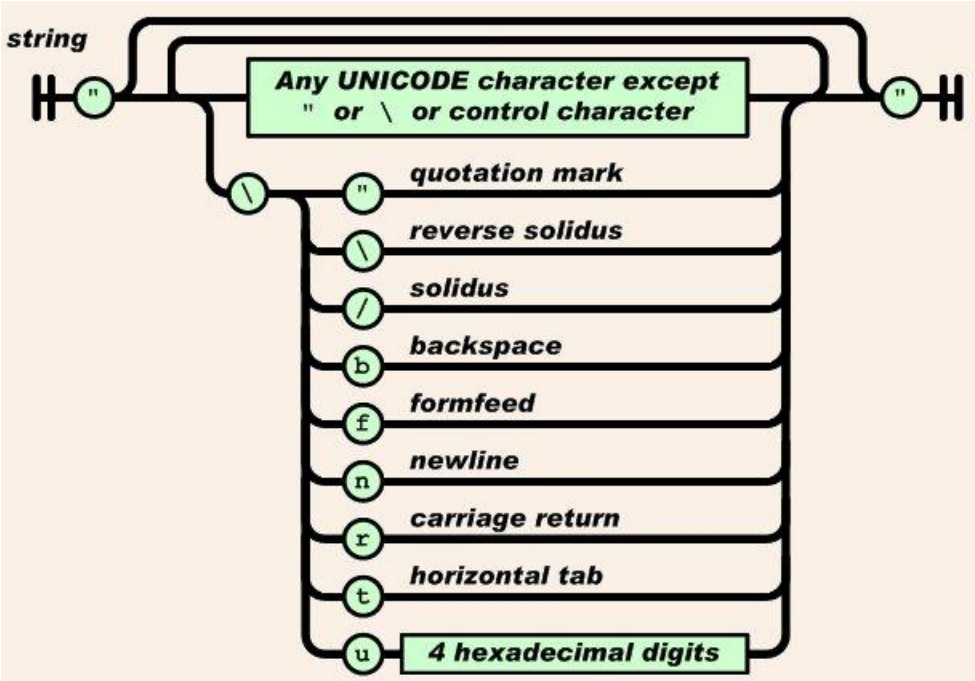


图 2 JSON 封装字符串的格式

从上图中我们可以知道在 JSON 中字符串是以”开始，以”结束，可包含 Unicode 的字符组。而图 1 中的数值格式则如下图所示:

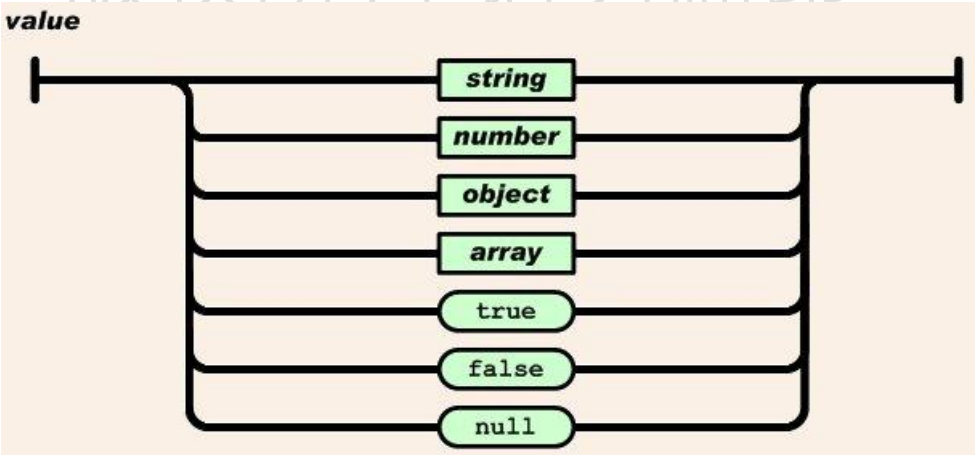


图 3 JSON 封装数值的格式

从图 3 中可知，JSON 的数值可以是字符串，数字，对象，数组或是 true/false/null 值。由于 JSON 的数值可以是对象或是数组，而对象又可以包含字符串:数值配对，因此 JSON 规范可以封装任何复杂的数据。

有了上述对于 JSON 基本的了解之后，我们就可以使用如下的格式从 DataSnap 服务器传递数据到客户端:

```
{“书名”:”实战 Delphi 2010”}
```

从这个格式中我们可以知道这是用 JSON 中封装对象的规则，把『字符串:数值』配对包含在{和}符号之中。

如果我们希望也传递作者的信息，那么可以使用逗号分离每一个『字符串:数值』配对，如下所示，读者也可以回头再参考图 1 就可以发现这是图 1 的规则。

```
{"书名":"实战 Delphi 2010","作者":"利瓦伊"}
```

如果现在再把书籍出版年份也传递，那么就可以使用下面的格式:

```
{"书名":"实战 Delphi 2010","作者":"利瓦伊","出版年份":2010}
```

请注意的是，出版年份的数值是 2010 这个数字，因为如图 3 所示数值可以是数字(number)，而在 JSON 中数字的定义如下:

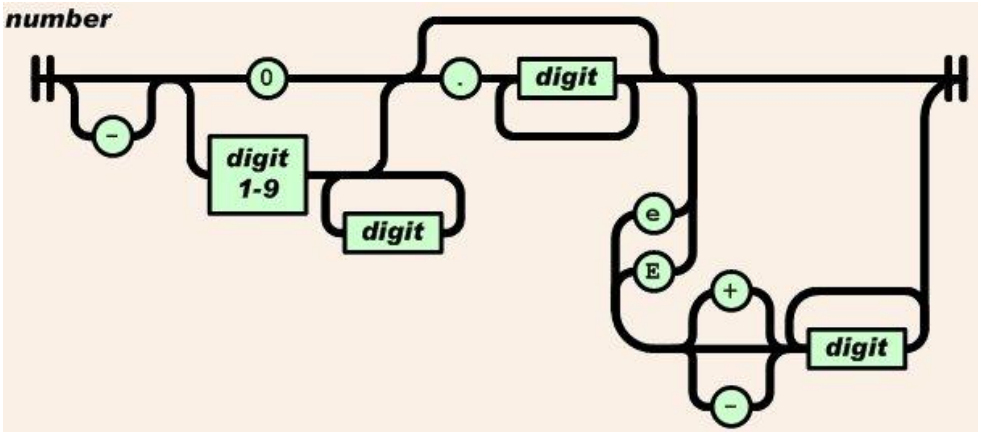


图 4 JSON 封装数值的格式

从图 4 我们可以知道，在 JSON 中数字可以是整数或是浮点数。

```
{"书名":"实战 Delphi 2010","作者":"利瓦伊","出版年份":2010}
```

因此如果我们再传递书籍价格，那么可以使用如下的格式:

```
{"书名":"实战 Delphi 2010","作者":"利瓦伊","出版年份":2010,"价格":45.95}
```

最后如果我们再加入书籍已出版否信息，那么可以使用如下的格式:

```
{"书名":"实战 Delphi 2010","作者":"利瓦伊","出版年份":2010,"价格":45.95,"已出版否":false}
```

从上面的范例中我们可以看到 JSON 规范如何封装各种不同型态的数据。那么如果我们需要一次传递多笔数据到客户端的话，又要如何封装呢?这可以使用 JSON 中数组的规范。

图 5 是 JSON 数组的封装规则，数组“[”开始，以“]”结束，而数组中的每一个元素都是数值，每一个元素使用逗号分隔。回头参考图 3 JSON 封装数

值的格式，由于数值可以是对象，因此我们如果需要一次传递多笔数据到客户端，那么就可以使用 JSON 的数组来封装。

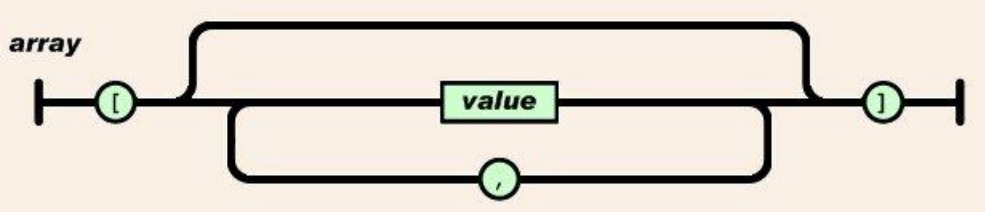


图 5 JSON 封装数组的格式

例如下面就是使用 JSON 的数组封装性三个代表书籍的对象，数组中的每一个元素数值是对象，每一个元素数值使用逗号分隔:

```
[{"书名":"实战 Delphi 2010"},{"书名":"Inside VCL"},{"书名":"Borland 传奇"}]
```

如何？一旦了解了 JSON 封装数据的规则之后读者是不是觉得使用 JSON 非常的简洁呢？正由于 JSON 在封装数据的规则比较简单，因此在解析 JSON 数据时速度也比较快，JSON 的简洁规则让 JSON 拥有较少的数据流量和较快的处理速度，因此让 JSON 在 Web 应用方面占有优势，也愈来愈受欢迎，这也是 DataSnap 现在选择使用 JSON 做为基础技术的原因。

在离开本小节之前，让我们比较一下使用 XML 和 JSON 在封装和交换数据方面的差异。下面是使用 XML 封装一个员工的信息:

```
<?xml version="1.0" encoding="utf-8"?>
<user>
    <name>李大明</name>
    <password>123456</password>
    <department>R&D</department>
    <gender>男</gender>
    <age>26</age>
</user>
```

如果我们使用 JSON 来封装的话，那么就如下所示：

```
{
    "name":"李大明",
    "password":"123456",
    "department":"R&D",
    "gender":"男",
    "age":"26"
}
```

我们可以看到 JSON 使用了较 XML 少的资料量来封装相同的信息，此 JSON 在数据交换方面拥有比较多的优势。

读者可以参考 www.json.org 以了解更多有关 JSON 的信息。

在对于 JSON 有了基本的了解之后，更重要的是我们需要知道如何在 Delphi 中进行 JSON 的程序设计。

2-2 VCL 框架中支持 JSON 的类别

Delphi 在 2009 开始使用 JSON 做为 DataSnap 2009 封装和传递数据的格式，但是 Delphi 2009 对于 JSON 的支持和 DataSnap 的功能撰写的非常紧密，开发人员除了能够在 DataSnap 2009 中使用 JSON 之外，并不容易使用 Delphi 2009 来开发其他应用型态的 JSON 应用程序。到了 Delphi 2010 这个现象获得了大幅的改善，Delphi 2010 提供了许多通用的 JSON 相关类别，开发人员可以使用这些通用的 JSON 相关类别开发任何型态的 JSON 应用程序而不只限于 DataSnap 应用程序。

Delphi 2010 在新的 DBXJSON.pas 程序单元中提供了这些 JSON 相关的类别，如果读者仔细观察前面 JSON 规则图的话，就可以发觉如果我们需要使用类别来实作支持 JSON 的规范，那么至少需要下面的类别:

类别	说明
JSONValue	用来代表和实作图 3 的实体和关系
JSONPair	用来代表 JSON 规范中的『名称/值』配对实体和关系
JSONObject	用来代表和实作图 1 的实体和关系
JSONString	用来代表和实作图 2 的实体和关系
JSONArray	用来代表和实作图 5 的实体和关系
JSONNumber	用来代表和实作图 4 的实体和关系

当然在上述的基础类别中还存在继承的关系，例如由于 JSON 规范中数值可以代表字符串，对象，数组等实体，因此上面的 JSONObject，JSONString 等类别可以从 JSONValue 类别继承下来。

Delphi 2010 中支持 JSON 等相关类别就是使用这个观念实作出来的，因此只要读者了解了前面解释 JSON 规范的观念，就可以非常直觉的了解这些实作类别。在下面的表格中整理了这些类别，并且提供了简单的叙述:

类别	说明
TJSONAncestor	抽象类，是所有 JSON 相关类别的根类别

TJSONPair	实作 JSON 规范中『名称/值』配对的类别
TJSONValue	实作 JSON 规范中数值的类别，TJSONValue 是 TJSONAncestor 的继承类别
TJSONTrue	实作 JSON 规范中代表 True 的类别，从 TJSONValue 继承下来
TJSONString	实作 JSON 规范中代表字符串的类别，从 TJSONValue 继承下来
TJSONNumber	实作 JSON 规范中代表数值的类别，它从 TJSONString 继承下来，因为 JSON 使用文字格式传递封装和传递数据，因此所有数值都将转换为文字字符串型态传递，到达目的地之后再反转回数值
TJSONObject	实作 JSON 规范中代表对象的类别，从 TJSONValue 继承下来
TJSONNull	实作 JSON 规范中代表 Null 的类别，从 TJSONValue 继承下来
TJSONFalse	实作 JSON 规范中代表 False 的类别，从 TJSONValue 继承下来

下图是这些类别之间的继承和关连关系图，请读者对照下图和以前图 1 到图 5 的 JSON 规范，就可以了解这些类别的实作是完全从 JSON 规范中设计出来的，简单又符合直觉:

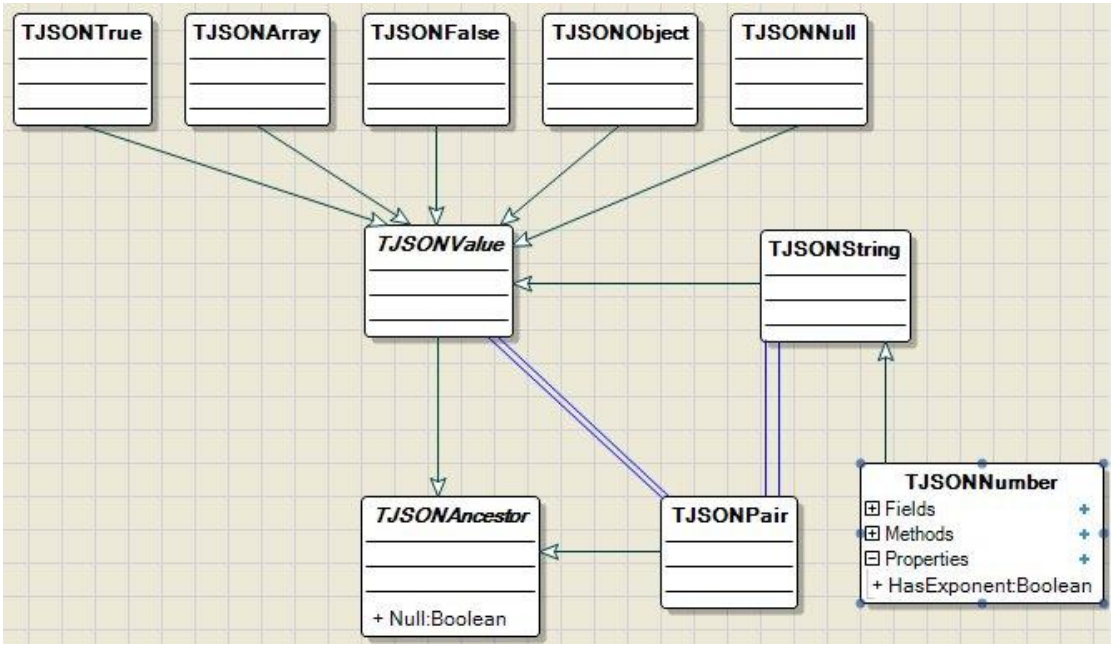


图 6 Delphi 中支持 JSON 的类别架构

Delphi XE 又加入了许多新的对于 JSON 支持的服务，让 Delphi 的开发人员能够更简单的在程序代码中处理 JSON。例如 Delphi XE 加入 TDBXJSONTools 类别

方法	说明
ValueTypeToJSON	封装 dbExpress 型态的信息为 JSON 的格式，主要的目的是使用 JSON 传递 DBX 数值的 Metadata
JSONToValueType	ValueTypeToJSON 的反相函式，把 JSON 封装的 metadata 转换回 DBX 的型态对象
TableToJSON	把 dbExpress 封装的 TDBXReader 中的数据转换为以 JSON 型态封装的数据，这个函式允许开发人员在伺服端把 TDataSet 等对象封装的数据以 JSON 的形式回传到客户端
DBXToJSON	封装 DBX 的数值为 JSON 的型态，把 DBX 能够代表的数值都转换为以 JSON 封装的形式，以便在伺服端和客户端传递数据
JSONToDBX	DBXToJSON 的反相函式，把 JSON 封装的数据转换回适当的 DBX 的数值型态
StreamToJSON	把 TStream 型态的数据转换为 TJSONArray 封装的数据，通常是使用在传递 2 进位型态的数据于伺服端和客户端之间
JSONToStream	StreamToJSON 的反相函式，把 TJSONArray 封装的数据转换为 TStream 型态的数据

TDBXJSONTools 类别的目的是帮助开发人员更容易的在服务器和客户端之间以 JSON 的形式传递 dbExpress 封装的数据。

在 10.3 中为了增加处理 TDBXArrayList 和 TJSONArray 对象的便利性，因此增加了 2 个 Enumerator 类别:

类别	说明
TJSONPairEnumerator	封装 TDBXArrayList 对象的 Enumerator 类别
TJSONArrayEnumerator	封装 TJSONArray 对象的 Enumerator 类别

这 2 个 Enumerator 类别的目的是为了让开发人员可以使用 for..in...循环来处理 TJSONPair 和 TJSONArray 对象中的对象。

接下来让我们简单的说明这些类别提供的服务，如此一来读者就可以了解如何在程序代码中使用它们。

2-2-1 TJSONAncestor 类别

TJSONAncestor 类别是所有 JSON 相关类别的根类别，它主要定义了三个虚拟方法让它的衍生类别来复载实作，下面的表格叙述了这些虚拟方法：

虚拟方法	说明
function ToString: UnicodeString; virtual;	ToString 虚拟方法会把 JSON 的内容转换为文字字符串形式回传，TJSONAncestor 的衍生类别需要实作这个虚拟方法，例如 TJSONObject 类别会实作这个虚拟方法并且把对象内容转换为字符串的形态回传
function EstimatedByteSize: Integer; virtual; abstract;	EstimatedByteSize 虚拟方法回传 JSON 对象预估的字节大小，TJSONAncestor 的衍生类别需要实作这个虚拟方法，每一个衍生类别都会回传它本身需要占据字节的大小。EstimatedByteSize 虚拟方法的目的是在封装和传递 JSON 对象时程序代码能够配置足够的内存大小之用。
function ToBytes(const Data: TBytes; const Offset: Integer): Integer; virtual; abstract;	ToBytes 虚拟方法能够把 JSON 对象以字节的形式回传，TJSONAncestor 的衍生类别需要实作这个虚拟方法，每一个衍生类别都会在这个复载的虚拟方法中把自己转换为字节形态。

TJSONAncestor 本身是一个抽象类，因此开发人员并不应该直接在程序代码中使用它，而是应该使用下面介绍的衍生类别。

2-2-2 TJSONValue 类别

TJSONValue 类别本身也是一个抽象类，它是直接从 TJSONAncestor 继承下来的，TJSONValue 类别只是做为一个 Placeholder，它主要的目的是代表图 3 中 JSON 规范 value 的概念，因此它的定义非常简单，如下所示：

```
TJSONValue = class abstract(TJSONAncestor)
end;
```

稍后介绍的许多类别都是从 TJSONValue 继承下来的，几乎和图 3 显示的规范架构一样。

2-2-3 TJSONPair 类别

TJSONPair 类别是实作 JSON 规范中『名称/值』配对的类别，它从 TJSONAncestor 直接继承下来，由于 TJSONPair 是实作『名称/值』配对，因此它提供了两个最重要的特性，JsonString 和 JsonValue:

```
  property JsonString: TJSONString read GetJsonString write SetJsonString;
  property JsonValue: TJSONValue read GetJsonValue write SetJsonValue;
```

JsonString 是代表『名称/值』配对中名称的特性，而 JsonValue 则代表其中值的特性。请注意的是 JsonValue 的型态是 TJSONValue，而根据前面表格所叙述，TJSONValue 可以代表任何从 TJSONValue 继承下来的类别，因此 JsonValue 可以代表 TJSONString,TJSONNumber 或是 TJSONObject 等。

那么如何建立 TJSONPair 呢？TJSONPair 定义了四个复载的建构函式，其中最重要的三个如下所示:

```
constructor Create(const Str: TJSONString; const Value: TJSONValue); overload;
constructor Create(const Str: UnicodeString; const Value: TJSONValue); overload;
constructor Create(const Str: UnicodeString; const Value: UnicodeString); overload;
```

上面第一个建构函式可以藉由 TJSONString 和 TJSONValue 对象来建立 TJSONPair 对象，第二个建构函式可以使用 Unicode 字符串和 TJSONValue 对象来建立 TJSONPair 对象，至于第 3 个建构函式则是使用两个 Unicode 字符串来建立 TJSONPair 对象。

例如，如果我们要建立如下的 JSON『名称/值』配对:

```
"书名":"实战 Delphi 2010"
```

那么我们可以使用如下的程序代码:

```
var
  jp : TJSONPair;
begin
  jp := TJSONPair.Create(TJSONString.Create('书名'), TJSONString.Create('实战 Delphi
2010'));
  try
    Memo1.Lines.Add(jp.ToString);
  finally
    jp.Free;
  end;
```

上面的程序代码使用了第一个建构函式建立 TJSONPair 对象，TJSONPair 类别的 ToString 方法可以把 JSON 对象之中的内容以文字字符串形式回传，最后释放 TJSONPair 对象时，TJSONPair 对象会自动释放传入建构函式之中的两个 TJSONString 对象。

当然您也可以使用第二个建构函式来建立 TJSONPair 对象，如下所示:

```
jp := TJSONPair.Create('书名', TJSONString.Create('实战 Delphi 2010'));
```

或是使用第 3 个建构函式，因为在这个范例中『名称/值』配对中的名称和值都是字符串:

```
jp := TJSONPair.Create('书名', '实战 Delphi 2010');
```

当然，如果您需要建立如下的 JSON『名称/值』配对:

```
"价格":45.95
```

由于值是数字，因此我们只能使用第一个或是第二个建构函式来建立:

```
jp := TJSONPair.Create(TJSONString.Create('价格'), TJSONNumber.Create(45.95));
```

或是:

```
jp := TJSONPair.Create('价格', TJSONNumber.Create(45.95));
```

一旦建立了 TJSONPair 对象，我们也可以藉由存取它的 JsonString 和 JsonValue 特性来存取『名称/值』配对中的名称或是值了，例如:

```
Memo1.Lines.Add('JSONString : ' + jp.JsonString.ToString);
Memo1.Lines.Add('JSONValue : ' + jp.JsonValue.ToString);
```

2-2-4 TJSONString 类别

TJSONString 是从 TJSONValue 类别继承下来，它是使用来代表 Unicode 字符串的数据，它可以是『名称/值』配对中的名称或是值或是两者。

TJSONString 最重要的方法应该是它的建构函式了，它接受一个 Unicode 字符串做为参数:

```
constructor Create(const Value: UnicodeString); overload;
```

在 Unicode 字符串使用建立 TJSONString 对象之后，如果开发人员需要加入额外的字符，那么可以呼叫 AddChar 虚拟程序，AddChar 会在原本的 Unicode 字符串之后加入参数 Ch 的字符内容。

```
procedure AddChar(const Ch: WideChar); virtual;
```

如果开发人员需要 TJSONString 对象中字符串的内容值，可以呼叫 ToString:

```
  function ToString: UnicodeString; override;
```

如果开发人员只是需要 TJSONString 对象中字符串的内容值，而不需要额外的开始”符号以及结束的”符号，那么可以呼叫 TJSONString 的 Value 函式

```
  function Value: UnicodeString; override;
```

例如下面的程序代码:

```
var
  js : TJSONString;
begin
  js := TJSONString.Create('实战 Delphi 2010');
  try
    Memo1.Lines.Add('TJSONString.ToString : '+ js.ToString);
    Memo1.Lines.Add('TJSONString.Value : '+ js.Value);
  finally
    js.Free;
  end;
```


下面是呼叫 ToString 和 Value 的差异:

```
TJSONString.ToString : "实战 Delphi 2010"
TJSONString.Value : 实战 Delphi 2010
```

2-2-5 TJSONObject 类别

TJSONObject 类别从 TJSONValue 直接继承下来，它代表 JSON 规范中封装对象的类别。要建立 TJSONObject 对象非常简单，只需要呼叫它的建构函式即可:

```
constructor Create;
```

那么如果我们需要建立如下的 JSON 对象内容，那么应该如何做呢?

```
{“书名”:”实战 Delphi 2010”}
```

请注意上面的结构，其实是在 JSON 对象之中包含一个 TJSONPair 对象，因此我们只需要执行下面的步骤即可:

- 建立 TJSONObject 对象
- 建立 TJSONPair 对象

- 把 TJSONPair 对象加入到 TJSONObject 对象

要把 TJSONPair 对象或是 JSON『名称/值』配对加入到 TJSONObject 对象中，我们可以使用 TJSONObject 类别中下面两个复载的 AddPair 方法:

```
  procedure AddPair(const Pair: TJSONPair); overload; virtual;
  procedure AddPair(const Str: TJSONString; const Val: TJSONValue); overload; virtual;
```

因此如果我们需要建立如下内容的 TJSONObject 对象:

```
{"书名":"实战 Delphi 2010","出版年份":2010}
```

那么可以使用如下的程序代码:

```
var
  jo : TJSONObject;
  jp : TJSONPair;
begin
  jo := TJSONObject.Create;
  try
    jp := TJSONPair.Create('书名', '实战 Delphi 2010');
    jo.AddPair(jp);
    jo.AddPair(TJSONString.Create('出版年份'), TJSONNumber.Create(2010));
    Memo1.Lines.Add(jo.ToString);
  finally
    jo.Free;
  end;
```

同样的 TJSONObject 类别的 ToString 方法可以把其中的内容以文字字符串型态回传:

```
  function ToString: UnicodeString; override;
```

2-2-6 TJSONNumber 类别

TJSONNumber 类别是从 TJSONString 类别继承下来的，这是因为在 JSON 规范中整数和浮点数都必须使用字符串形式来代表，由于 TJSONString 已经提供了使用字符串形式来代表 JSON 内容的功能，因此 TJSONNumber 只需要从它继承下来并且把整数和浮点数转换为文字字符串型态即可。

在前面我们看过多次建立 TJSONNumber 对象的范例了，它的建构函式接受一个 double 值的参数:

```
constructor Create(const Value: Double); overload;
```

一旦建立了 TJSONNumber 对象，呼叫它的 ToString 方法即可取得其内容。

在离开本小节之前再让我们看看三个剩下简单的类别，它们是 TJSONTrue，TJSONFalse 和 TJSONNull。这三个类别分别实作图 3 中的 true，false 和 null 三个 JSON 数值。例如我们如果需要下面的 JSON 内容:

```
{"书名":"实战 Delphi 2010","出版否":false, "撰写中":true, "出版商":null}
```

那么我们使用下面的程序代码即可:

```
var
  jo : TJSONObject;
  jp: TJSONPair;
begin
  jo := TJSONObject.Create;
  try
    jp := TJSONPair.Create('书名', '实战 Delphi 2010');
    jo.AddPair(jp);
    jp := TJSONPair.Create('书名', TJSONFalse.Create);
    jo.AddPair(jp);
    jp := TJSONPair.Create('撰写中', TJSONTrue.Create);
    jo.AddPair(jp);
    jp := TJSONPair.Create('出版商', TJSONNull.Create);
    jo.AddPair(jp);
    Memo1.Lines.Add(jo.ToString);
  finally
    jo.Free;
  end;
```

2-2-7 TJSONArray 类别

TJSONArray 类别从 TJSONValue 继承下来，它的目的是实作 JSON 定义中 JSON 数组的元素，在 JSON 规范中 JSON 数组的定义如图 7 所示:

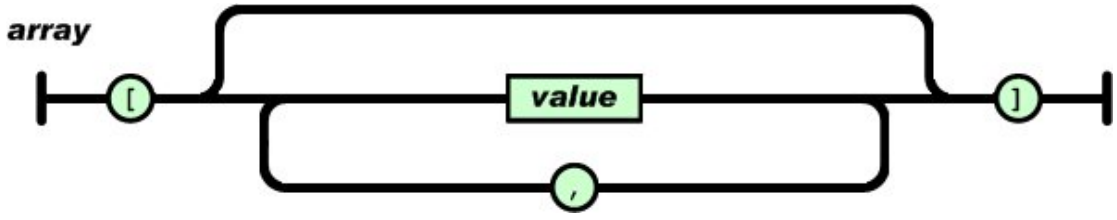


图 7 JSON 数组的定义

JSON 数组可封装 JSON 数值为数组的元素，每一个封装的 JSON 数值元素以分号分隔。由于在 VCL 中封装 JSON 数值的类别是 TJSONValue，如前所述 TJSONValue 是抽象类，也是每一个封装 JSON 元素的祖先类别，因此这代表 TJSONArray 类别几乎可以封装任何的复杂的数据，例如当 dbExpress 以 JSON 形式传递数据时，以会使用 TJSONArray 封装 TDataSet 包含的数据。

TJSONArray 类别提供了数个方法可让开发人员在其中加入元素，例如下面即是 TJSONArray 类别提供相关加入元素的方法:

```
  procedure AddElement(const Element: TJSONValue);
  function Add(const Element: UnicodeString): TJSONArray; overload;
  function Add(const Element: Integer): TJSONArray; overload;
  function Add(const Element: Double): TJSONArray; overload;
  function Add(const Element: Boolean): TJSONArray; overload;
  function Add(const Element: TJSONObject): TJSONArray; overload;
  function Add(const Element: TJSONArray): TJSONArray; overload;
```

开发人员可以呼叫 AddElement 在 TJSONArray 对象中加入 TJSONValue 型态的对象元素，或是呼叫 Add 方法直接在其中加入字符串，整数，浮点数，甚至是 TJSONObject 或是 TJSONArray 型态的对象。请注意的是，由于 Add 方法会回传 TJSONArray 对象本身，因此我们可以类似下面的程序代码在 TJSONArray 对象中加入元素:


```
procedure TForm18.Button1Click(Sender: TObject);
var
  aJA : TJSONArray;
begin
  aJA := TJSONArray.Create;
  try
    aJA.Add('JSON 数组元素').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);
    Memo1.Lines.Text := aJA.ToString;
  finally
    aJA.Free;
  end;
end;
```

下面是执行的结果画面:

图 8 使用 TJSONArray 对象封装元素

要取得 TJSONArray 中的元素，开发人员可以呼叫 Get 方法并且传入元素的索引位置:

```
   function Get(const Index: Integer): TJSONValue;
```

例如我们可以使用下面的程序代码取得前面范例中 TJSONArray 对象中的每一个元素:

```
procedure TForm18.Button2Click(Sender: TObject);
var
  aJA : TJSONArray;
  iIndex : Integer;
begin
  aJA := TJSONArray.Create;
  try
    aJA.Add('JSON 数组元素').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);


    for iIndex := 0 to aJA.Size - 1 do
    begin
      ListBox1.Items.Add(aJA.Get(iIndex).ToString);
    end;
  finally
    aJA.Free;
  end;
end;
```

下面是执行的结果画面:

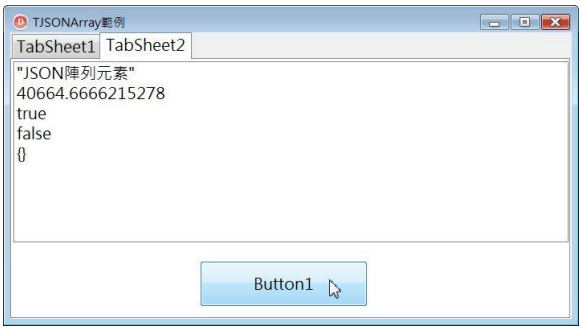


图 9 使用 Get 方法撷取 TJSONArray 中的元素

当然在上面的程序代码中使用了 for 循环从 TJSONArray 中把每一个元素取出，但我们也可以使用 10.3 中新的 TJSONArrayEnumerator 类别来进行一样的工作，例如下面的程序代码在 010 行建立 TJSONArrayEnumerator 对象并且传入 TJSONArray 对象做为参数，接着就可以在 011 行藉由呼叫 TJSONArrayEnumerator 类别的 MoveNext 方法来判断是否还有元素可存取，如果 MoveNext 回传 True 的话，就存取 TJSONArrayEnumerator 对象的 Current 特性以取得目前 TJSONArray 中对应的元素。

```
001   procedure TForm18.Button3Click(Sender: TObject);
002   var
003     aJA : TJSONArray;
004     jae : TJSONArrayEnumerator;
005   begin
006     aJA := TJSONArray.Create;
007     try
008       aJA.Add('JSON 数组元素
').Add(Now).Add(True).Add(False).Add(TJSONObject.Create);
009
010       jae := TJSONArrayEnumerator.Create(aJA);
011       while jae.MoveNext do
012         ListBox2.Items.Add(jae.Current.ToString);
013     finally
014       jae.Free;
015       aJA.Free;
016     end;
017   end;
```

在读者了解了如何使用这些 JSON 相关的类别之后，让我们看看如何使用它们在分布式应用系统之中。

前面的章节简单的了如何使用DataSnap开发基于JSON分布式应用系统，Delphi 藉由原本的 Midas/DataSnap 和 dbExpress 的组件提供这个新的分布式计算能力，但使用这些组件都是数据的方式来存取远程服务，然而我们也可以使用前面介绍的 JSON 类别再结合 DataSnap 来实作出存取远程数值对象(Value Object)或是所谓的 DTO(Data Transfer Object)的应用，让客户端可以存取远程的对象。

在下面的范例中我们将展示如何开发使用VO/DTO的DataSnap应用程序服务器，并且在客户端撷取其中封装的数据和对象。这个范例将开发一个可以让客户端查询数据和对象的 DataSnap 应用程序服务器，当客户端查询对象时，我们将使用 TJSONObject 封装对象并且传递到客户端。

2-3-1 开发数值对象服务器

首先建立一个 VCL Form 应用程序项目，在它的主窗体中加入如下的组件:

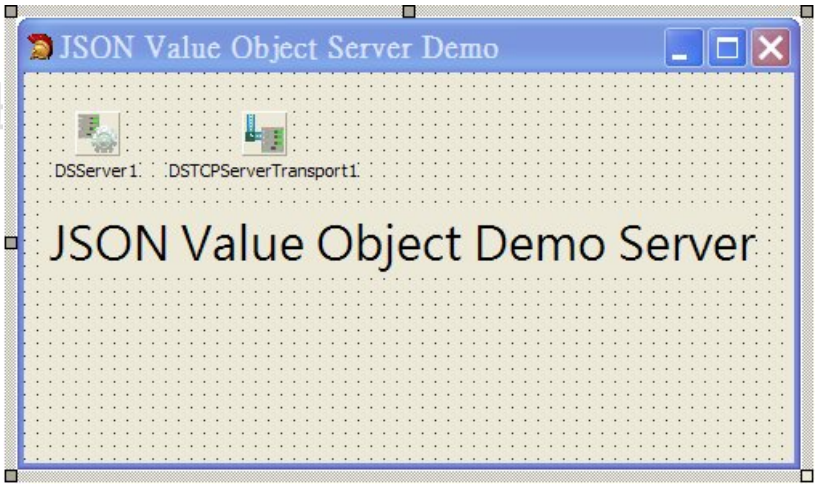


图 10 VO DataSnap 应用程序服务器

接着在项目中建立一个新的程序单元，于其中实作一个简单的 TEmployee 类别如下:

```
unit uEmployee;

interface

uses SysUtils, classes;

type
```

```
  TEmployee = class
  private
    { Private declarations }
    FName : string;
    FEMail : string;
    FPhone : string;
  public
    { Public declarations }
    constructor Create(sName  : string = ''; sEMail : string = ''; sPhone : string = '');

    property Name : string read FName write FName;
    property EMail : string read FEmail write FEMail;
    property Phone : string read FPhone write FPhone;
  end;

implementation

{ TEmployee }

constructor TEmployee.Create(sName, sEMail: string; sPhone: string);
begin
  FName := sName;
  FEmail := sEMail;
  FPhone := sPhone;
end;

end.
```

再于项目中建立一个新的程序单元称为 uEmployeeValueObject，其中定义 TEmployeeVO 类别如下，请注意 TEmployeeVO 类别使用了{$MethodInfo ON}和{$MethodInfo Off}编译程序指令要求编译程序把 TEmployeeVO 类别的 Reflection 信息编译到执行文件中。

```
  {$MethodInfo ON}
  TEmployeeVO = class(TComponent)
  private
    { Private declarations }
    FEmployees : TObjectList<TEmployee>;
```

```
  function CreateEmployeeJSONObject(employee : TEmployee) : string;
  function CreateEmployeeJSONArray : string;
  function CreateEmployeeJSONObjectJ(employee : TEmployee) : TJSONObject;
public
  { Public declarations }
  destructor Destroy; override;
  function GetEmployeeJ(const sName : string) : TJSONObject;
  function GetAllEmployeesJ : TJSONArray;
  procedure AddEmployee(const sName : string; const sEMail : string; const sPhone : string);
end;
{$MethodInfo Off}
```

TEmployeeVO 宣告了三个方法说明如下:

方法名称	说明	备注
AddEmployee	让客户端呼叫增加员工信息	
GetEmployeeJ	让客户端以员工姓名查询员工对象	请注意 GetEmployeeJ 回传 TJSONObject 型态的结果值，这代表范例应用程序将把员工对象封装在 TJSONObject 对象之中并且回传给客户端
GetAllEmployeesJ	让客户端查询所有的员工信息	查询后端所有员工信息，请注意 GetAllEmployeesJ 回传型态为 TJSONArray 的结果值，这代表范例应用程序将把所有查询结果封装在 TJSONArray 对象中回传

下面小节将简单的说明这些方法的实作。

AddEmployee 方法的实作

AddEmployee 方法非常的简单，它从客户端接受三个字符串型态的参数，并且根据它们来建立 TEmployee 对象，最后再把建立的 TEmployee 对象加入在宣告为 TObjectList<TEmployee>泛型型态的变量 FEmployees 之中。

```
procedure TEmployeeVO.AddEmployee(const sName, sEMail, sPhone: string);
var
  employee : TEmployee;
begin
```

```
  if (FEmployees = nil) then
    FEmployees := TObjectList<TEmployee>.create(true);
  employee := TEmployee.Create(sName, sEmail, sPhone);
  FEmployees.Add(employee);
end;
```

GetEmployeeJ 方法的实作

GetEmployeeJ 方法就非常的有趣了，它展示了如何于 DataSnap 应用程序服务器中使用 TJSONValue 的相关类别，GetEmployeeJ 回传型态为 TJSONObject 的对象回客户端。

GetEmployeeJ 根据客户端传递来查询的员工姓名，在 Fem ployees 中一一的搜寻具有相同名称的 TEmployee 对象，找到之涕在 013 行呼叫 CreateEmployeeJSONObjectJ 来建立回传的 TJSONObject 对象。

```
001   function TEmployeeVO.GetEmployeeJ(const sName: string): TJSONObject;
002   var
003     ie : TList<uEmployee.TEmployee>.TEnumerator;
004     employee : TEmployee;
005   begin
006     Result := nil;
007     ie := FEmployees.GetEnumerator;
008     while (ie.MoveNext) do
009     begin
010       employee := ie.Current;
011       if (employee.Name = sName) then
012       begin
013         Result := CreateEmployeeJSONObjectJ(employee);
014         break;
015       end;
016     end;
017   end;
```

CreateEmployeeJSONObjectJ 方法根据找到的 TEmployee 对象来建立 TJSONObject 对象，它呼叫我们前面学习过的 AddPair 方法，把每一个 TEmployee 对象的特性名称和特性值做为一个 JSON 的中『名称/值』配对加入到 TJSONObject 对象。

```
function TEmployeeVO.CreateEmployeeJSONObjectJ(employee: TEmployee): TJSONObject;
var
```

```
  aJO : TJSONObject;
begin
  aJO := TJSONObject.Create;
  aJO.AddPair(TJSONString.Create('姓名'), TJSONString.Create(employee.Name));
  aJO.AddPair(TJSONString.Create('EMail'), TJSONString.Create(employee.EMail));
  aJO.AddPair(TJSONString.Create('电话'), TJSONString.Create(employee.Phone));
  Result := aJO;
end;
```

GetAllEmployeesJ 方法的实作

GetAllEmployeesJ 方法会把所有存在泛型型态变量 FEmployees 之中的 TEmployee 对象封装在 TJSONArray 对象中回传给客户端。

在 013 行仍然是呼叫 CreateEmployeeJSONObjectJ 为每一个 CreateEmployeeJSONObjectJ 建立一个 TJSONObject 对象，并且加入到回传的 TJSONArray 对象中。

```
001   function TEmployeeVO.GetAllEmployeesJ: TJSONArray;
002   var
003     ie : TList<uEmployee.TEmployee>.TEnumerator;
004     employee : TEmployee;
005     jo : TJSONObject;
006     ja : TJSONArray;
007   begin
008     ie := FEmployees.GetEnumerator;
009     ja := TJSONArray.Create;
010     while (ie.MoveNext) do
011     begin
012       employee := ie.Current;
013       jo := CreateEmployeeJSONObjectJ(employee);
014       ja.AddElement(jo);
015     end;
016     Result := ja;
017   end;
```

最后不要忘记在主窗体中注册 TEmployeeVO 类别，如此一来客户端才能看见并且呼叫 TEmployeeVO 输出的方法:

```
procedure TForm10.FormCreate(Sender: TObject);
begin
```

```
  if DSServer1.Started then
    DSServer1.Stop;
  RegisterServers;
  DSServer1.Start;
end;

procedure TForm10.FormDestroy(Sender: TObject);
begin
  DSServer1.Stop;
end;

procedure TForm10.RegisterServers;
begin
  uEmployeeValueObject.RegisterServerClasses(Self, DSServer1);
end;
```

现在编译并且执行 DataSnap 应用程序服务器，并且准备开发客户端。

2-3-2 开发范例客户端

其实这个范例的重点就在于客户端如何呼叫远程支持 TJSONValue 相关类别的方法，这是因为在 DataSnap 2009 中无法支持 TJSONValue 相关类别，到了 DataSnap 2010 才支持。但是在笔者撰写本章时 Delphi 2010 的 Beta 版仍然无法自动产生代表远程类别的客户端 Proxy 类别，因此想要呼叫前面实作的 GetEmployeeJ 和 GetAllEmployeesJ 方法，我们必须了解如何修改由 Delphi 产生的 DataSnap 客户端类别。

> 笔者认为当 Delphi 2010 正式版释出时，CodeGear 应该会把这个问题修正，如果您发现您使用的 Delphi 已经能够产生正确的客户端 Proxy 类别，那么就不需要如下所叙述的修改了。

首先建立一个 VCL Form 应用程序，放入 TSQLConnection 组件，设定 Driver 为 DataSnap 再把 connected 设定为 True(请确定 DataSnap 应用程序服务器已经在执行)，接着用点选鼠标右键，选择 Generate DataSnap client classes 选项，储存产生的程序单元为 uServerProxy.pas，然后搜寻 GetEmployeeJ 方法，把 012 行修改为如下:

```
001   function TEmployeeVOClient.GetEmployeeJ(sName: string): TJSONObject;
002   begin
```

```
003     if FGetEmployeeJCommand = nil then
004     begin
005       FGetEmployeeJCommand := FDBXConnection.CreateCommand;
006       FGetEmployeeJCommand.CommandType := TDBXCommandTypes.DSServerMethod;
007       FGetEmployeeJCommand.Text := 'TEmployeeVO.GetEmployeeJ';
008       FGetEmployeeJCommand.Prepare;
009     end;
010     FGetEmployeeJCommand.Parameters[0].Value.SetWideString(sName);
011     FGetEmployeeJCommand.ExecuteUpdate;
012     Result := FGetEmployeeJCommand.Parameters[1].Value.GetJSONValue as TJSONObject;
013   end;
```

这是因为远程 GetEmployeeJ 方法回传 TJSONObject 型态的对象，因此我们需要把 012 行呼叫远程方法执行的结果转变型态为 TJSONObject 型态。

同样的，我们也需要修改 GetAllEmployeesJ 方法，转变型态为回传 TJSONArray 对象，如下所示:

```
function TEmployeeVOClient.GetAllEmployeesJ: TJSONArray;
begin
  if FGetAllEmployeesJCommand = nil then
  begin
    FGetAllEmployeesJCommand := FDBXConnection.CreateCommand;
    FgetAllEmployeesJCommand.CommandType := TDBXCommandTypes.DSServerMethod;
    FGetAllEmployeesJCommand.Text := 'TEmployeeVO.GetAllEmployeesJ';
    FGetAllEmployeesJCommand.Prepare;
  end;
  FGetAllEmployeesJCommand.ExecuteUpdate;
  Result := FGetAllEmployeesJCommand.Parameters[0].Value.GetJSONValue as TJSONArray;
end;
```

了解了如何以及为什么需要修改 DataSnap 客户端类别之后，我们就可以开始实作呼叫远程方法的程序代码了。

首先下面的程序代码藉由自动产生的 TEmployeeVOClient 类别呼叫 AddEmployee 方法,把客户端输入的员工数据加入在远程的应用程序服务器中:

```
procedure TForm11.btnAddEmployeeClick(Sender: TObject);
var
  evo : TEmployeeVOClient;
begin
```

```
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    evo.AddEmployee(edtAddName.Text, edtAddEMail.Text, edtAddPhone.Text);
    edtAddName.Text := '';
    edtAddEMail.Text := '';
    edtAddPhone.Text := '';
  finally
    evo.Free;
  end;
end;
```

下图是执行范例客户端应用程序并且点选『增加员工』按钮以执行上述程序代码的画面:

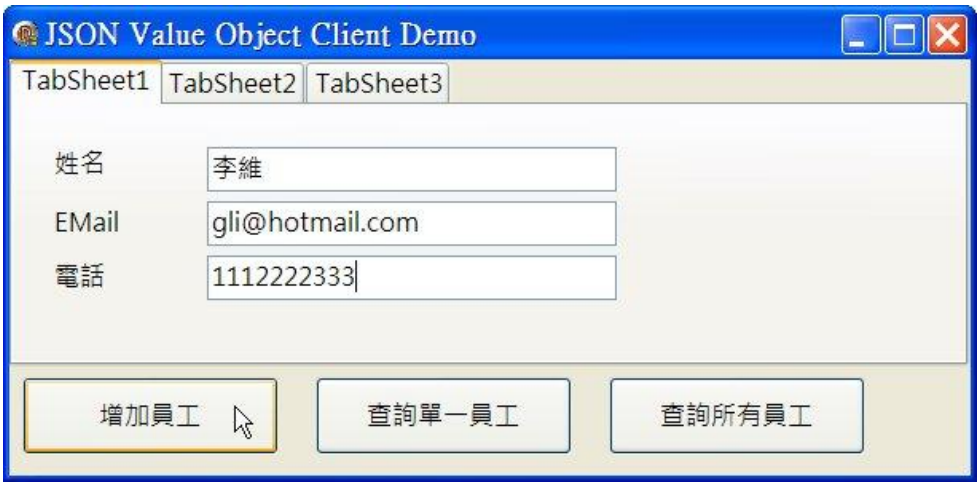


图 11 客户端应用程序呼叫远程 AddEmployee 方式加入员工信息

接着是藉由 TEmployeeVOClient 呼叫 GetEmployeeJ 查询员工数据的实作程序代码:

```
procedure TForm11.btnQueryEmployeeClick(Sender: TObject);
var
  evo : TEmployeeVOClient;
  jo : TJSONObject;
  employee : TEmployee;
begin
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    jo := evo.GetEmployeeJ(edtQname.Text);
    employee := TEmployee.Create(jo.Get(0).JsonValue.ToString,
jo.Get(1).JsonValue.ToString, jo.Get(2).JsonValue.ToString);
```

```
    Self.edtQEMail.Text := employee.EMail;
    Self.edtQPhone.Text := employee.Phone;
  finally
    FreeAndNil(employee);
    FreeAndNil(jo);
    evo.Free;
  end;
end;
```

在上面的程序代码中呼叫 GetEmployeeJ 取得代表员工的 TJSONObject 对象，再根据 TJSONObject 对象中的信息于客户端建立 TEmployee 对象，再显示员工信息于窗体中，最后不要忘记释放 TEmployee 对象，TJSONObject 对象和 TEmployeeVOClient 对象。

下图是先增加利瓦伊这笔员工数据之后，再输入利瓦伊来查询的画面:

图 12 输入员工姓名查询员工数据

点选上图中的『查询单一员工』按钮之后我们的确可以取得远程员工对象的信息，如下图所示:

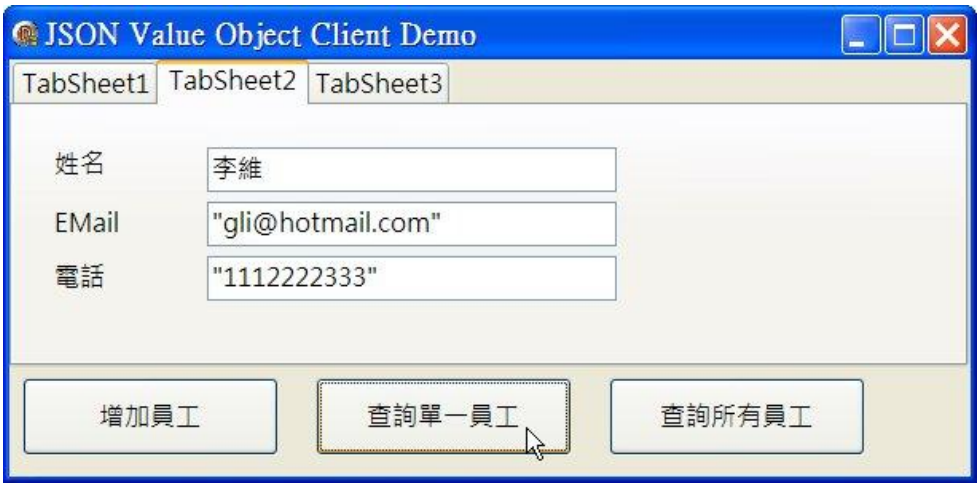


图 13 查询的结果画面

但是为什么上图中的员工信息，例如员工的 Email 有引号包围呢？这当然是因为这是 JSON 封装字符串的规范，而在上面的程序代码中当我们建立 TEmployee 对象时是直接呼叫 ToString 方法，如果我们不希望 TEmployee 对象的特性值有引号包围，那么我们可以修改程序代码如下，改呼叫 Value 方法：

```
   employee := TEmployee.Create(jo.Get(0).JsonValue.Value, jo.Get(1).JsonValue.Value,
jo.Get(2).JsonValue.Value);
```

那么就会有如下正确的结果：

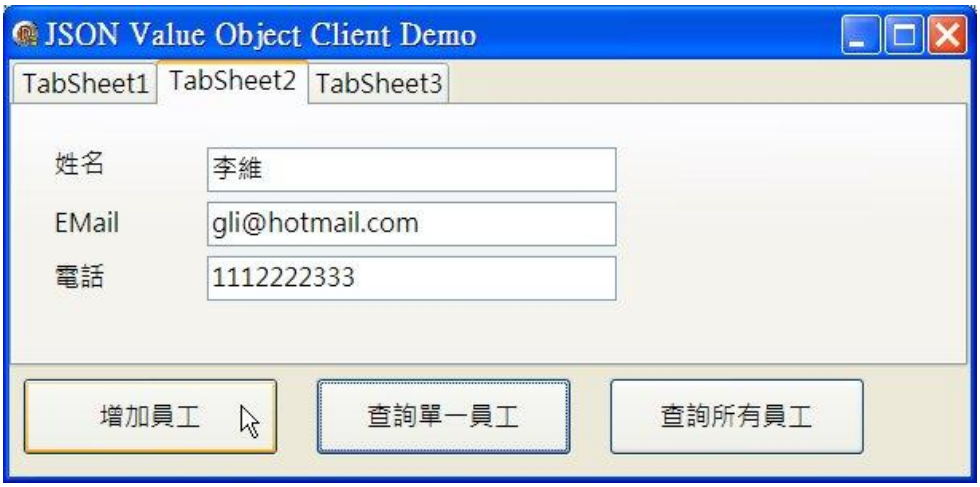


图 14 查询的结果画面

最后让我们看看如何查询所有的员工数据。下面的程序代码藉由 TEmployeeVOClient 呼叫 GetAllEmployeesJ 取得 TJSONArray 对象，然后进入 for 循环把其中的每一个元素取出再转变型态为 TJSONObject 对象，最后再根据 TJSONObject 对象一一的建立客户端的员工对象。

```
procedure TForm11.btnQueryAllEmployeesClick(Sender: TObject);
```

```
var
  evo : TEmployeeVOClient;
  jo : TJSONObject;
  ja : TJSONArray;
  employee : TEmployee;
  iIndex: Integer;
begin
  evo := TEmployeeVOClient.Create(Self.SQLConnection1.DBXConnection);
  try
    ja := evo.GetAllEmployeesJ;
    for iIndex := 0 to ja.Size - 1 do
    begin
      jo := ja.Get(iIndex) as TJSONObject;
      employee := TEmployee.Create(jo.Get(0).JsonValue.ToString,
jo.Get(1).JsonValue.ToString, jo.Get(2).JsonValue.ToString);
      Memo1.Lines.Add(employee.Name);
      FreeAndNil(employee);
    end;
  finally
    FreeAndNil(ja);
    FreeAndNil(employee);
    evo.Free;
  end;
end;
```

下图是执行此查询的结果，我们可以看到客户端的确可以查询到远程的所有员工的信息。

图 15 所有员工数据查询的结果画面

当然，我们一样可以修改上面的程序代码如下:

```
    employee := TEmployee.Create(jo.Get(0).JsonValue.Value, jo.Get(1).JsonValue. Value,
jo.Get(2).JsonValue. Value);
```

那么我们会看到如下的结果:

图 16 所有员工数据查询的结果画面

2-4 使用 JSON 封装和传递数据

现在再让我们看看如何使用 TDBXJSONTools 类别帮助开发人员封装和传递复杂的数据。

2-4-1 开发 DataSnap REST 服务器

首先在 Delphi 整合发展环境中建立一个 DataSnap REST 应用程序项目，接着在 ServerMethodsUnit1 程序单元中宣告下面的两个方法:

```
  function GetData : String;
  function GetImage : TJSONArray;
```

GetData 方法将把 TDataSet 中包含的数据以 JSON 的形式传递到客户端，让读者了解如何把数据集转换为 JSON 数据。

GetImage 方法把是把数据集中 Blob 字段封装的图形数据转换为 JSON 形式的数据，让读者了解如何把 2 进位的数据换为 JSON 数据。

下面的 GetData 方法的实作程序代码，首先它使用 TDBXCommand 对象执行 SQL 命令，TDBXCommand 对象的 ExecuteQuery 方法执行 SQL 叙述之后会回传 TDBXReader 对象，其中就封装了 SQL 叙述执行的结果数据集，接着我们就可以呼叫 TDBXJSONTools 的类别方法 TableToJSON 把

TDBXReader 对象封装的数据转换为以 JSON 封装的字符串数据，再回传给客户端。

```
function TServerMethods1.GetData: String;
var
  aCommand : TDBXCommand;
begin
  Result := '';
  aCommand := CHINESEDEMO.DBXConnection.CreateCommand;
  try
    aCommand.Text := 'select CATEGORY, COMMON_NAME, TOPOTYPE from BIOLIFE';
    Result := TDBXJSONTools.TableToJSON(aCommand.ExecuteQuery, 10, True).ToString;
  finally
    aCommand.Free;
  end;
end;
```

GetImage 方法则是把 BIOLIFE 数据表中 Graphic 字段包含的图形数据先储存到 TMemoryStream 对象中，再呼叫 TDBXJSONTools 的类别方法 StreamToJSON 把 2 进位形态的数据转换为 JSON 封装的字符串形态数据再传递回客户端。


```
function TServerMethods1.GetImage: TJSONArray;
var
  aMS : TMemoryStream;
begin
  BIOLIFE.Active := True;
  aMS := TMemoryStream.Create;
  try
    BIOLIFEGRAPHIC.SaveToStream(aMS);
    aMS.Position := 0;
    Result := TDBXJSONTools.StreamToJSON(aMS, 0, aMS.Size);
  finally
    BIOLIFE.Active := False;
    aMS.Free;
  end;
end;
```

现在编译并且执行此范例 DataSnap REST 应用程序服务器。

2-4-2 开发范例客户端

接着在项目群组中建立一个 VCL Form 应用程序项目，再于项目中建立 DataSnap Client Module，然后在主窗体中撰写如下的程序代码呼叫服务器的 GetData 方法:

```
procedure TForm18.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Text := ClientModule1.ServerMethods1Client.GetData;
end;
```

下图是执行上述程序代码的结果，我们可以清楚的看到数据集的数据被封装成 TJSONObject 传递到客户端，每一笔数据又藉由 TJSONArray 封装。

图 17 传递 TDataSet 包含的数据

接着再于主窗体中撰写如下的程序代码以呼叫服务器的 GetImage 方法:

```
procedure TForm18.btnJSONImageClick(Sender: TObject);
var
  aJA : TJSONArray;
  aStream : TStream;
begin
  aJA := ClientModule1.ServerMethods1Client.GetImage;
  Memo1.Lines.Text := aJA.ToString;
  try
    aStream :=  TDBXJSONTools.JSONToStream(aJA);
    Image1.Picture.Bitmap.LoadFromStream(aStream);
  finally
```

```
    aStream.Free;
  end;
end;
```

在上面的程序代码中首先把呼叫 GetImage 方法取得的 JSON 封装的数据显示在主窗体的 TMemo 控件中，接着再藉由呼叫 TDBXJSONTools 的类别方法 JSONToStream 把伺服端传递来的 TJSONArray 对象转换回 TStream 对象(其实是 TBytesStream 对象)，最后藉由主窗体中的 TImage 组件把图形数据显示在客户端的主窗体中。

下面的画面是伺服端回传的图形数据，我们可以清楚的看到图形数据被封装成 TJSONArray 对象:

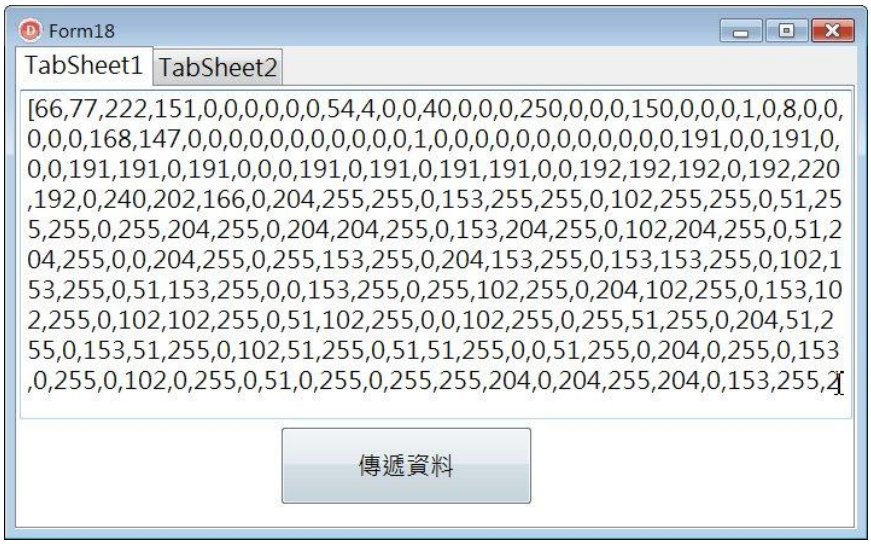


图 18 以 JSON 型态封装的图形数据

而下面的画面则是使用 TDBXJSONTools. JSONToStream 方法把上图中的字符串型态的数据转换回图形数据并且显示在 TImage 组件中的结果，我们可以看到藉由 TDBXJSONTools 类别，开发人员可以轻易的封装和传递复杂型态的数据。

图 19 藉由 TDBXJSONTools 类别把数据转换回图形

由于此范例服务器是 REST 服务器，因此我们也可以使用浏览器来呼叫 GetData 和 GetImage 方法。例如下图就是在浏览器中呼叫 GetData 方法的结果：

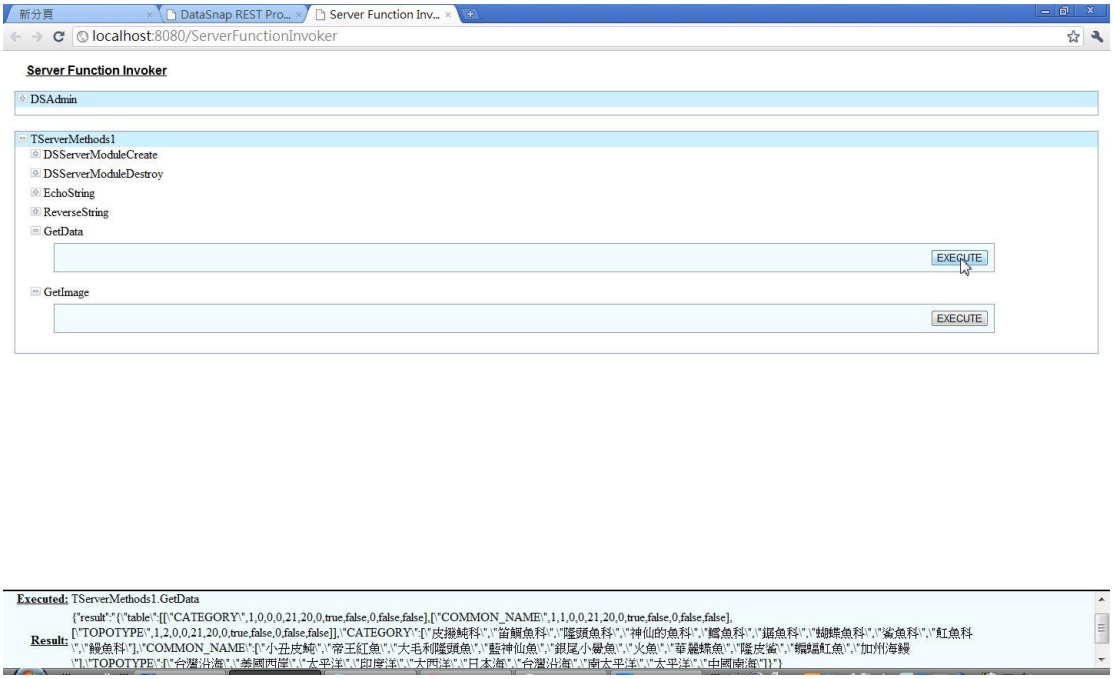


图 20 使用浏览器呼叫服务器的 GetData 方法

而下图则是在浏览器中呼叫 GetImage 方法的结果：

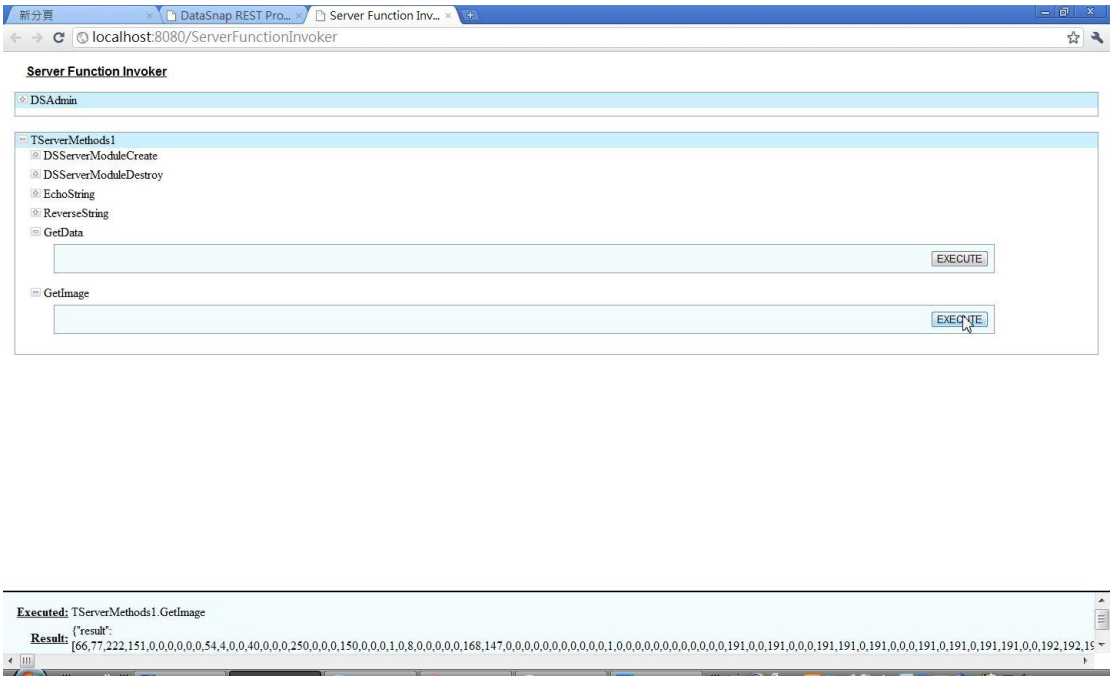


图 21 使用浏览器呼叫服务器的 GetImage 方法

最后读者想提醒读者的是，虽然结合 DataSnap 相关的 JSON 类别和 TDBXJSONTools 类别可以封装和传递任何复杂的数据，但使用 JSON 传递大量的数据仍然是不智的，因为以 JSON 传递大量的数据不仅缓慢而且会消耗网络带宽，JSON 只适合传递少量的数据，其目的是跨平台，和整合跨平台的应用。如果开发人员真的要使用 JSON 传递传统数据，也应该尽量减少数据量。

对于主从架构，多层应用系统而言 Delphi 和 C++Builder 的开发人员可以在传递大量资料时仍然使用 dbExpress，需要在 Web，跨平台传递数据时再选用 JSON，Delphi 和 C++Builder 提供同时提供了效率(dbExpress)和弹性(JSON)的方式让开发人员自行选择传递/存取数据的技术。

2-5 结论

本章叙述了 Delphi VCL 框架中支持 JSON 开发的相关类别，读者从本章中可以了解这些不但是依照 JSON 规范设计的，而且非常的直觉，好用，只要我们对于 JSON 规范有基本的掌握就可以顺利的使用它们。

此外新版的 DataSnap 不但使用 JSON 做为封装和传递数据的基础，现在也加入支持使用 TJSONValue 和衍生类别做为远程方法的参数和回传值，如此一来可以让开发人员撰写客制化的 JSON 应用，例如使用 VO/DTO 设计样例来传递数据和对象，使用这样的技术，开发人员也可以使用.NET，Java 或是任何支持 JSON 的程序语言来开发客户端应用程序。

DataSnap 10.3 提供了 TDBXJSONTools 类别可帮助开发人员以 JSON 封装数据集或是任何 2 进位形式的数据，再搭配 TJSONArray 和 TJSONObject 类别，开发人员就可以使用 JSON 封装和传递任何复杂型态的数据了。

第3章 DataSnap/REST服务器的授权和认证

在前一章中我们讨论了如何开发 DataSnap/REST 服务器和 Win32 的客户端应用程序，然而由于 REST 和 JSON 适用于任何平台和客户端的应用，因此即使 Web 或是移动的客户端也能够连结到 DataSnap/REST 服务器并且呼叫其中的服务，本章将从如何开发 DataSnap/REST 的 Web 应用程序说起，接着会讨论如何对于 DataSnap/REST 服务器中公开的服务方法进行授权和认证的安全机制功能。

3-1 开发 DataSnap/RESTful Web 客户端应用程序

> 注意:本范例需要使用 VCL For Web(IntraWeb)组件组

现在让我们说明如何开发 VCL For Web 的应用程序来呼叫使用 DataSnap/REST 服务器。在 Delphi 整合发展环境中开启上一章的项目群组，在项目群组中建立 VCL For Web 应用程序， 如下图所示:

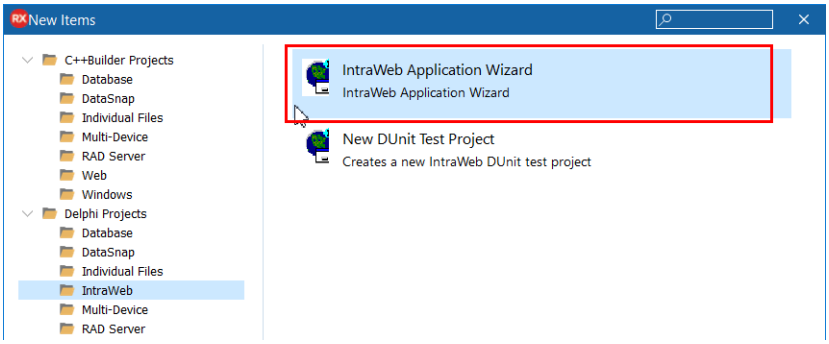


图 3-1 建立 VCL For Web 应用程序

点选了 OK 按钮之后，再让我们选择建立 StandAlone 型态的应用程序，如下图所示：

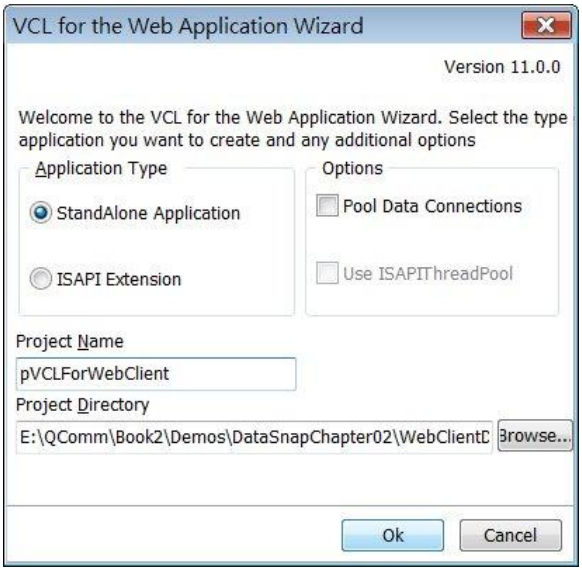


图 3-2 建立 StandAlone 型态的应用程序

开启 VCL For Web 项目中的 UserSession 模块，并且在其中放入 TSQLConnection 组件链接前一章的范例 DataSnap/REST 服务器，再放入 TDSProviderConnection 和 TClientDataSet 组件链接范例 DataSnap/REST 服务器中的 TDataSetProvider 组件 dspArticles，如同前一章讨论如何开发 DataSnap 客户端程序一样，此时 VCL For Web 项目中的 UserSession 看起来如下所示：

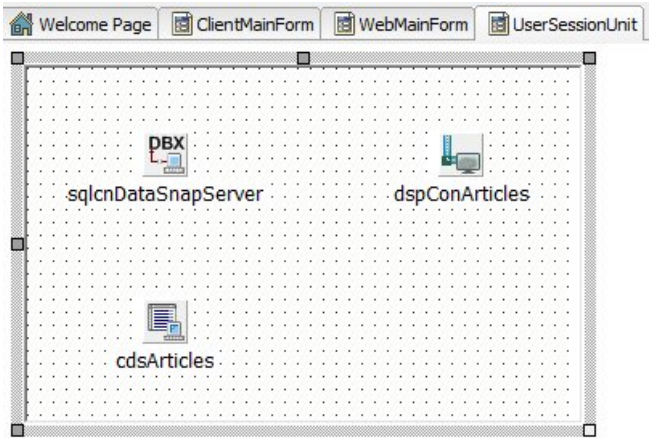


图 3-3 在 UserSession 程序单元模块中加入 dbExpress 组件链接 DataSnap/REST 服务器

现在我们就可以使用 UserSession 中的 TSQLConnection 组件自动产生 DataSnap 客户端类别以便呼叫范例 DataSnap/REST 服务器输出的服务。请

使用鼠标右击 sqlcnDataSnapServer 组件，从快捷菜单中选择『Generate DataSnap client classes』，如下图所示:

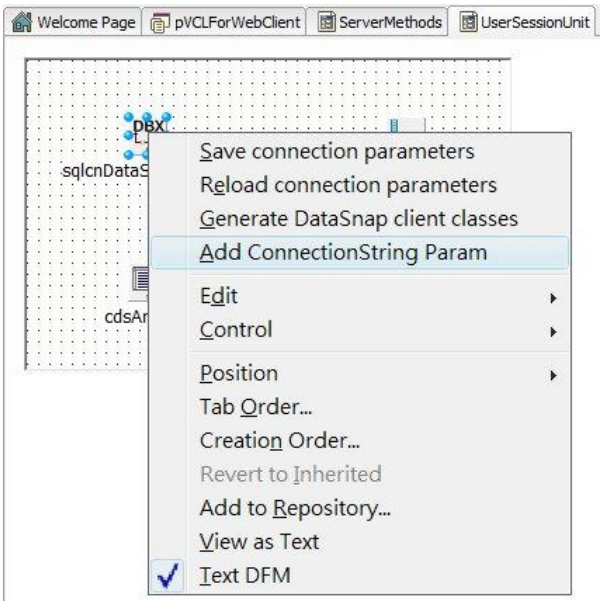


图 3-4 点选 TSQLConnection 组件，选择产生 DataSnap 客户端类别

然后储存自动产生的 DataSnap 客户端类别为 ServerProxy.pas，并且重新命名主窗体为 WebMainForm.pas，此时 VCL For Web 项目应该如下图所示:

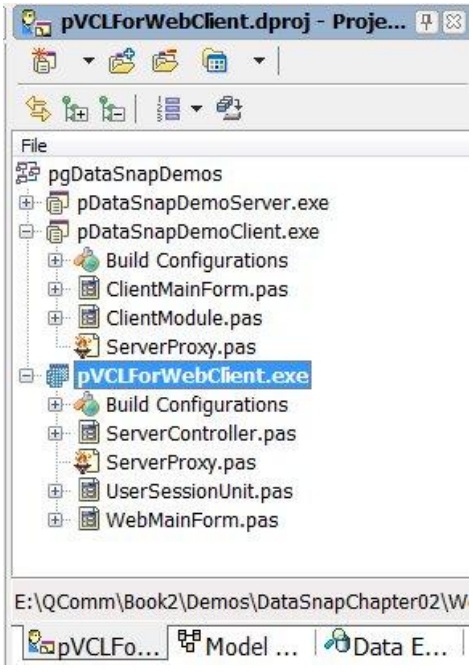


图 3-5 储存 DataSnap 客户端类别之后 VCL For Web 项目的内容

开启专案中的 WebMainForm，放入 TIWDBNavigator，TIWDBGrid，TIWMemo，TIWButton 和 TDataSource 组件如下:

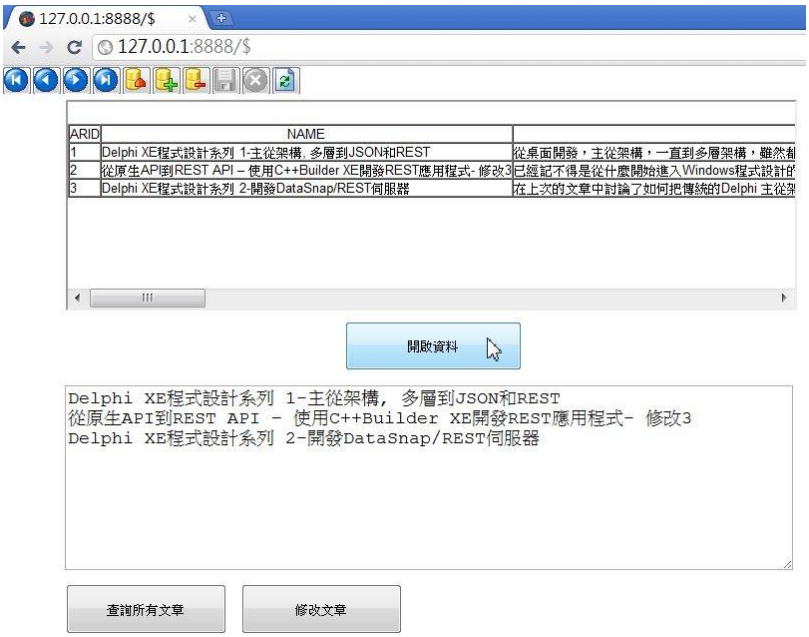


图 3-6 VCL For Web 主窗体链接 DataSnap/REST 服务器

接着在『开启数据』按钮的事件处理函式中开启 UserSession 中的 TClientDataSet:

```
procedure TiwMainForm.iwbtnOpenTableClick(Sender: TObject);
begin
  UserSession.cdsArticles.Active := True;
end;
```

VCL For Web 应用程序便轻松的从范例 DataSnap/REST 服务器中取得数据显示在 Web 主窗体中，『查询所有文章』按钮则呼叫了 DataSnap/REST 服务器的 QueryAllArticles 方法取得所有文章的 TJSONArray，再一一的从其中取出文章名称并且显示在主窗体的 TIWMemo 组件中:

```
procedure TiwMainForm.iwbtnQueryAllArticlesClick(Sender: TObject);
var
  aServer : TServerMethods5Client;
  ja : TJSONArray;
  js : TJSONString;
  iIndex: Integer;
begin
  aServer :=
```

```
TServerMethods5Client.Create(UserSession.sqlcnDataSnapServer.DBXConnection);
  try
    ja := aServer.QueryAllArticles;
    for iIndex := 0 to ja.Size - 1 do
    begin
      js := ja.Get(iIndex) as TJSONString;
      IWMemo1.Lines.Add(js.Value);
    end;
  finally
    aServer.Free;
  end;
end;
```

接着在 VCL For Web 项目中再建立另外一个 VCL For Web 新的窗体，并且设计其窗体如下:

图 3-7 呼叫 DataSnap/REST 服务器查询和修改数据的窗体

要查询文章非常的简单，直接使用 dbExpress 的 Lookup 方法即可:

```
procedure TiwfmArticle.iwbtnQueryArticleClick(Sender: TObject);
begin
  if (UserSession.cdsArticles.Active = False) then
    UserSession.cdsArticles.Active := True;
  iwedtArticleName.Text := UserSession.cdsArticles.Lookup('ARID',
StrToInt(iwedtArticleID.Text), 'Name');
  iwmmHeadline.Lines.Text := UserSession.cdsArticles.Lookup('ARID',
StrToInt(iwedtArticleID.Text), 'HEADLINE');
end;
```

要修改文章也很简单，除了也可以直接使用 dbExpress 之外，当然也可以使用 JSON 格式直接呼叫 DataSnap/REST 服务器提供的服务即可。例如如果客户端是 JavaScript，Ruby，PHP 或是其他不使用 dbExpress 的客户端，那么就可以藉由呼叫 DataSnap/REST 服务器提供的服务取得 JSON 封装的资料再加以处理即可。

例如在图 3-6 中的『修改文章』按钮使用了下面的程序代码让用户在 Web 窗体中对于文章数据的异动更新回 DataSnap/REST 服务器中:

```
procedure TiwfmArticle.iwbtnUpdateArticleClick(Sender: TObject);
var
  aServer : TServerMethods5Client;
  jaArticle : TJSONArray;
begin
  aServer :=
TServerMethods5Client.Create(UserSession.sqlcnDataSnapServer.DBXConnection);
  try
    jaArticle := CreateUpdateArticleObject;
    aServer.ModifyArticle(jaArticle);
    UserSession.cdsArticles.Refresh;
  finally
    aServer.Free;
  end;
end;
```

『修改文章』按钮的 OnClick 事件处理函式先建立 TServerMethods5Client 对象，再呼叫 CreateUpdateArticleObject 方法建立一个以 JSON 格式封装的异动过的文章数据，最后再藉由 TServerMethods5Client 对象呼叫 DataSnap/REST 服务器输出的 ModifyArticle 方法把异动过的数据更新回服务器/数据库之中。

而 CreateUpdateArticleObject 方法在 032 行建立 TJSONArray 对象，接着呼叫 CreateArticleIDObject，CreateArticleNameObject 和 CreateArticleHeadlineObject 方法分别建立封装文章 ID，名称和标题的 TJSONObject 对象并且加入到 TJSONArray 对象中，最后再回传 TJSONArray 对象给前面的 iwbtnUpdateArticleClick 事件处理函式并且呼叫服务器的 ModifyArticle 方法。

```
001   function TiwfmArticle.CreateUpdateArticleObject: TJSONArray;
002   var
```

```
003     jpArticle : TJSONPair;
004
005     function CreateArticleIDObject : TJSONObject;
006     begin
007       jpArticle := TJSONPair.Create;
008       Result := TJSONObject.Create;
009       jpArticle.JsonString := TJSONString.Create('ARID');
010       jpArticle.JsonValue :=  TJSONString.Create(iwedtArticleID.Text);
011       Result.AddPair(jpArticle);
012     end;
013
014     function CreateArticleNameObject : TJSONObject;
015     begin
016       jpArticle := TJSONPair.Create;
017       Result := TJSONObject.Create;
018       jpArticle.JsonString := TJSONString.Create('NAME');
019       jpArticle.JsonValue :=  TJSONString.Create(iwedtArticleName.Text);
020       Result.AddPair(jpArticle);
021     end;
022
023     function CreateArticleHeadlineObject : TJSONObject;
024     begin
025       jpArticle := TJSONPair.Create;
026       Result := TJSONObject.Create;
027       jpArticle.JsonString := TJSONString.Create('HEADLINE');
028       jpArticle.JsonValue :=  TJSONString.Create(iwmmHeadline.Lines.Text);
029       Result.AddPair(jpArticle);
030     end;
031   begin
032     Result := TJSONArray.Create;
033     Result.AddElement(CreateArticleIDObject);
034     Result.AddElement(CreateArticleNameObject);
035     Result.AddElement(CreateArticleHeadlineObject);
036   end;
```

下图就是在 VCL For Web 应用程序的 Web 窗体中先查询文章数据，再于 Web 窗体中修改文章数据，最后点选『修改文章』按钮把文章数据更新回服务器中。

图 3-8 VCL For Web 应用程序呼叫 DataSnap/REST 服务器修改数据

在使用 VCL For Web 开发 Web 应用程序时，开发人员仍然可以使用 DataSnap 技术以及 JSON 封装的资料，和开发一般的 Wi32 DataSnap 客户端窗口应用程序是一样的。

3-2 认证和授权

在开发 DataSnap/REST 服务器时很重要的一个设计要素就是认证和授权，所谓认证是指可连结 DataSnap/REST 服务器的使用者，而授权则是指每一个登录和连结的使用者可呼叫的服务。例如我们在上一章中设计的 DataSnap/REST 服务器提供了许多的方法可让客户端呼叫，使用。但是如果我们希望只有系统合法的用户才能够连结使用，而且有一些服务方法是只有特定的使用者才可以呼叫使用，那么我们应该如何完成这个控制存取的功能?

例如假设我们把上一章服务器中的服务方法重新设计如下:

```
TServerMethods5 = class(TDSServerModule)
  …
  public
    {访客可呼叫的方法}
    function QueryAllArticles : TJSONArray;
    function QueryHeadline(Name : string) : string;

    {Admin, Gordon 可呼叫的方法}
    function GetArticleID : Integer;
    function ModifyArticle(jaArticle : TJSONArray) : boolean;
    function AddArticle(Name : string; Headline : string) : boolean;
```

```
    {只有 Gordon 可呼叫的方法}
    function GetArticle(Name : String) : TDataSet;
  end;
```

在 TServerMethods5 类别中我们把服务方法分成不同的群组，而且有的群组的方法只能让特定的使用者呼叫，例如 AddArticle 方法只有 Admin 群组的使用者和 Gordon 这个特定的使用者可以呼叫，而 GetArticle 方法则只限 Gordon 可以呼叫。那么我们如何能够实作这些认证和授权的功能呢?

其实我们只要能够了解 DataSnap 如何执行下面的工作就可以立刻掌握如何使用 DataSnap 的认证和授权的功能:

- 如何传递用户登录信息
- 如何验证使用者
- 如何授权用户可呼叫的方法

在下面的小节将详细的说明如何在 DataSnap 中完成上面的工作。

传递用户登录信息

对于 DataSnap/REST 服务器进行安全控管的第一步当然就是认证系统合法的用户，当客户端连结 DataSnap/REST 服务器欲进行服务呼叫的一开始，DataSnap/REST 服务器必须先认证此客户端是否是合法的使用者。在 DataSnap 中客户端可以使用两种不同的技术让 DataSnap/REST 服务器对于连结的客户端进行认证的工作。第一种是使用 dbExpress 技术，当然目前这仅限于 Delphi 和 C++Builder 的客户端，第 2 种方法是使用 JavaScript，那么任何支持 JavaScript 的客户端都可以使用这种方式让 DataSnap/REST 服务器认证。

让我们先说明如何使用 Delphi 来传递认证信息给 DataSnap/REST 服务器。

使用 Delphi 客户端传递认证信息

在说明如何使用 Delphi 客户端传递认证信息之前，我们需要先解释 DataSnap 服务器如何进行认证的工作。当开发人员建立 DataSnap/REST 服务器时，如果勾选了使用安全机制，那么 DataSnap 精灵会在 ServerContainer 模块中加入 TDSAuthenticationManager，DataSnap/REST 服务器中认证和授权的工作就是由 TDSAuthenticationManager 负责。

TDSAuthenticationManager 组件提供了两个事件处理函式进行认证和授权，它的 OnUserAuthorize 会对连结的客户端进行认证的工作，它的 OnUserAuthenticate 事件处理函式则会在客户端呼叫服务方法时检查是否授权呼叫，下面的表格说明了这两个事件处理函式的功能:

服务器型态	说明
OnUserAuthenticate	客户端连结时DataSnap呼叫此事件处理函式认证是否为合法的系统用户
OnUserAuthorize	客户端呼叫服务器的服务时，DataSnap呼叫此事件处理函式检查此客户端是否能够呼叫此特定的服务方法

因此要对客户端进行认证是否为合法的系统用户，第一个方法就是在 TDSAuthenticationManager 组件的 OnUserAuthenticate 事件处理函式中进行认证。OnUserAuthenticate 的定义原型如下:

```
TDSAuthenticationEvent = procedure(Sender: TObject; const Protocol: UnicodeString; const Context: UnicodeString; const User: UnicodeString; const Password: UnicodeString; var valid: boolean; UserRoles: TStrings) of object;
```


TDSAuthenticationEvent 接受数个参数，这些参数由 DataSnap 框架传递进入，下面的表格说明其中 User，Password，valid 和 UserRoles 参数的意义:

参数名称	说明
User	客户端使用者的登录ID
Password	客户端用户的登录密码
valid	如果是合法的使用者就需要设定valid为True，否则就设定valid为False
UserRoles	开发人员可根据连结的客户端指定使用者到不同的角色角色，例如一般的使用者可指定为'users'角色群组，系统管理员可指定为'admin' 角色群组。稍后在授权用户可呼叫的服务方法时可以根据角色群组来授权

而 TDSAuthenticationEvent 的 Protocol 和 Context 参数则会根据连结的客户端的种类而包含不同的参数值，下面的表格说明了这两个参数根据不同的客户端被指定的参数值:

参数名称	说明
Delphi/C++Builder使用dbExpress连结的客户端	Protocol参数值='tcp/ip'

	Context = ‘’
Web应用程序客户端	Protocol 参数值 =’http’或’https’ Context = ‘’
RESTful客户端，例如在浏览器中使用如下的REST呼叫语法呼叫DataSnap/REST服务器时: http://localhost:8080/**datasnap**/**rest**/TServerMethods5/EchoString/test	Protocol参数值=’datasnap’ Context = ‘/rest’

授权用户

至于当客户端呼叫服务器时是否授权客户端呼叫则由 OnUserAuthorize 事件处理函式决定，它的定义原型如下:

```
TDSAuthorizationEvent = procedure(Sender: TObject; AuthorizeEventObject:
TDSAuthorizeEventObject; var valid: boolean) of object;
```

TDSAuthorizationEvent 接受两个参数，其中 AuthorizeEventObject 是型态为 TDSAuthorizeEventObject 的对象，而 TDSAuthorizeEventObject 又是从 TDSServerMethodUserEventObject 继承下来的，TDSServerMethodUserEventObject 类别提供了四个重要的特性，这些特性在下面的表格中说明:

特性	说明
UserName	客户端使用者的登录名称
UserRoles	此客户端使用者属于的角色群组，也就是在前面 TDSAuthenticationEvent 事件处理函式中设定的角色群组
AuthorizedRoles	其中包含了可呼叫此服务方法的角色群组
DeniedRoles	其中包含了不可呼叫此服务方法的角色群组

由于 TDSServerMethodUserEventObject 又是从 TDSServerMethodEventObject 类别继承下来的，而 TDSServerMethodEventObject 类别又提供了三个重要的特性，其说明如下:

特性	说明
ServerClass	被呼叫的服务方法所属的类别
MethodAlias	被呼叫的服务方法的名称，这是以字符串型态代表的名称，其格式为’类别名称.方法名称’
MethodInstance	被呼叫的服务方法的样例(指标)

由上面的说明我们可以了解，开发人员可以从 AuthorizeEventObject 参数中取得是那一个使用者，那一个角色群组要呼叫什么类别的什么方法。因此开发人员就可以在事件处理函式中使用程序代码来决定是否授权用户呼叫服务方法。

而 TDSAuthorizationEvent 的第二个参数 valid 决定了客户端是否能够呼叫服务方法，如果设定为 True 的话，那么客户端就被允许呼叫服务方法，设定为 False 的话客户端就会被拒绝呼叫。

了解了这 DataSnap 框架如何控制认证和授权后，就可以开始说明客户端要如何传递认证信息给 DataSnap/REST 服务器。

- 使用 TSQLConnection 组件

最简单的方法就是使用 TSQLConnection 组件来设定客户端链接的用户名称和密码来让 DataSnap/REST 服务器进行认证的工作，在客户端的 TSQLConnection 的 Driver 特性中的 DSAuthUser 和 DSAuthPassword 两个子特性就可以让开发人员设定用户名称和密码，例如下图显示了这两个子特性：

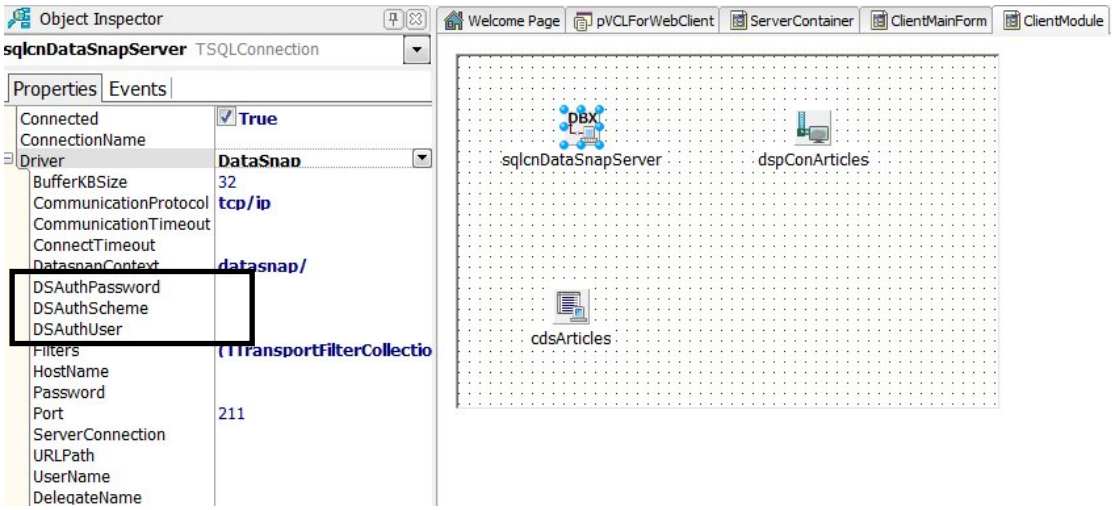


图 3-9 使用 TSQLConnection 组件设定登录服务器的用户名称和密码

开发人员可以在对象查看器中设定这两个认证信息。

- 使用 Delphi 程序代码

当然使用 TSQLConnection 组件设定认证信息固然方便，但产生的问题是没有弹性，在实际的应用中程序应该让用户在执行客户端应用程序时输入登录信息，再据此链接使用 DataSnap/REST 服务器。要使用程序代码传递客户端认证信息到 DataSnap/REST 服务器，开发人员必须使用 TDBXDatasnapProperties 对象。

TSQLConnection 组件的 ConnectionData 特性是 TConnectionData 对象，而 TConnectionData 中的 Properties 特性是 TDBXProperties 对象，当 TSQLConnection 是链接 DataSnap/REST 服务器时，TSQLConnection. ConnectionData. Properties 则是 TDBXDatasnapProperties 物件。TDBXDatasnapProperties 类别中就包含了 DSAuthUser 和 DSAuthPassword 这两个可传递到 DataSnap/REST 服务器认证信息的特性。

因此在程序代码中，开发人员可以使用下面的程序代码把用户登录的信息传递到 DataSnap/REST 服务器进行认证:

```
001   procedure TForm17.Button4Click(Sender: TObject);
002   var
003     prop : TDBXDatasnapProperties;
004     aServer : TServerMethods5Client;
005     cm : TClientModule1;
006   begin
007     cm := TClientModule1.Create(nil);
008     try
009
cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionUser] := edtUserName.Text;
010
cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionPassword] := edtPassword.Text;
011       aServer := cm.ServerMethods5Client;
012       aServer.AddArticle(edtAddedArticleName.Text, mmHeadline.Text);
013     finally
014       cm.Free;
015     end;
016   end;
```

在上面的程序代码中，007 行先建立 DataSnap 客户端模块对象，并且在 011 行使用客户端模块对象取得代表远程 DataSnap/REST 服务器对象 aServer 之前，先在 009 行设定使用者名称以及在 010 行设定用户密码，才于 011 行取得 aServer，接着才在 012 行呼叫 DataSnap/REST 服务器的 AddArticle 服务方法加入文章信息。

为什么要在 011 行之前先藉由 TDBXDatasnapProperties 设定用户名称和密码? 这是因为 011 行呼叫的 GetServerMethods5Client 方法中会开启客

户端模块中的 TSQLConnection，下面的程序代码中 005 行显示了这个动作，因此在呼叫 GetServerMethods5Client 之前必须先设定好用户名称和密码。

```
001   function TClientModule1.GetServerMethods5Client: TServerMethods5Client;
002   begin
003     if FServerMethods5Client = nil then
004     begin
005       sqlcnDataSnapServer.Open;
006       FServerMethods5Client:=
TServerMethods5Client.Create(sqlcnDataSnapServer.DBXConnection, FInstanceOwner);
007     end;
008     Result := FServerMethods5Client;
009   end;
```

认证和授权客户端范例

现在让我们看看一个使用程序代码来进行客户端认证的范例，首先让我们开启上一章的范例 DataSnap/REST 服务器，开启它的 ServerContainer 模块，在模块中的 TDSAuthenticationManager 组件的 OnUserAuthenticate 事件处理函式中撰写如下的程序代码:


```
procedure TServerContainer5.DSAuthenticationManager1UserAuthenticate(
  Sender: TObject; const Protocol, Context, User, Password: string;
  var valid: Boolean; UserRoles: TStrings);
begin
  valid := CheckUser(User, Password, UserRoles);
end;
```

在 DSAuthenticationManager1UserAuthenticate 事件中我们把 valid 指定为呼叫 CheckUser 的执行结果来判断目前连结的客户端是否为合法的使用者。而 CheckUser 的实作如下:

```
function TServerContainer5.CheckUser(const User, Password: string;
  UserRoles: TStrings): boolean;
begin
  Result := True;
  if ((User = '') and (Password = '')) then
    UserRoles.Add('guest')
  else
    if ((User = 'admin') and (Password = 'admin')) then
      UserRoles.Add('Admin')
```

```
   else
   if ((User = 'Gordon') and (Password = '123456')) then
     UserRoles.Add('Master')
   else
     Result := False;
end;
```

在 CheckUser 中我们检查从客户端传递来的用户认证信息，并且把不同的使用者指定到不同的角色群组中，例如如果客户端使用者没有输入任何的用户名称和密码，就被指定为 guest 角色群组，如果用户名称是 admin，密码也是 admin 的话，就指定为 Admin 角色群组。

接着让我们在 TDSAuthenticationManager 组件的 On User Authorize 事件处理函式中撰写如下的程序代码:

```
procedure TServerContainer5.DSAuthenticationManager1UserAuthorize(
  Sender: TObject; EventObject: TDSAuthorizeEventObject;
  var valid: Boolean);
begin
  valid := CanAuthorize(EventObject);
end;
```

在上面的程序代码中也是把呼叫 CanAuthorize 函式的结果指定给 valid 参数。而 CanAuthorize 实作如下:

```
001   function TServerContainer5.CanAuthorize(
002     EventObject: TDSAuthorizeEventObject): Boolean;
003   begin
004     Result := True;
005
006     if ((EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles') or
(EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles')) then
007     begin
008       if ((EventObject.UserRoles.IndexOf('guest') <> -1) or
009           (EventObject.UserRoles.IndexOf('Admin') <> -1) or
010           (EventObject.UserRoles.IndexOf('Master') <> -1)) then
011         Result := True
012       else
013         Result := False;
014       exit;
015     end;
```

```
016
017     if ((EventObject.MethodAlias = 'TServerMethods5.ModifyArticle') or
(EventObject.MethodAlias = 'TServerMethods5.AddArticle')
018         or (EventObject.MethodAlias = 'TServerMethods5.GetArticleID')) then
019     begin
020       if ((EventObject.UserRoles.IndexOf('Admin') <> -1) or
021           (EventObject.UserRoles.IndexOf('Master') <> -1)) then
022         Result := True
023       else
024         Result := False;
025       exit;
026     end;
027   end;
```

CanAuthorize 函式先于 004 行设定回传结果为 True 以便让客户端能够呼叫所有服务器输出的服务方法(在上一章讨论开发服务器的内文中我们在 DataExplorer 中展示了 DataSnap 服务器输出了许多其他的系统方法)，接着在 006 行我们藉由 EventObject 对象的 MethodAlias 特性值，以

```
'类别名称.方法名称'
```

的格式来检查目前客户端呼叫的服务方法是什么？如果是 TServerMethods5 类别输出的方法，那么就根据 EventObject 对象中 UserRoles 特性来判断目前呼叫服务方法的使用者群组是那一个，再根据使用者群组来决定是否允许呼叫。例如如果是呼叫查询函式，那么 guest，Admin 和 Master 角色群组都可以呼叫，但如果是呼叫 ModifyArticle'等方法，那么只有 Admin 和 Master 角色群组都可以呼叫。

当然，前面的程序代码只是为了展示如何使用 EventObject 的 UserRoles 特性，由于 004 行已经设定回传结果为 True，因此我们可以把上面的程序代码改成如下的实作:

```
001   function TServerContainer5.CanAuthorize(
002     EventObject: TDSAuthorizeEventObject): Boolean;
003   begin
004     Result := True;
005
006     if ((EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles') or
(EventObject.MethodAlias = 'TServerMethods5.QueryAllArticles')) then
007     begin
008       if (not ((EventObject.UserRoles.IndexOf('guest') <> -1) or
```

```
009             (EventObject.UserRoles.IndexOf('Admin') <> -1) or
010             (EventObject.UserRoles.IndexOf('Master') <> -1)) ) then
011         Result := False;
012       exit;
013     end;
…
```

完成了服务器认证和授权的实作之后，现在请编译并且执行范例 DataSnap/REST 服务器。

接着建立一个客户端应用程序，并且使用下面的程序代码呼叫 DataSnap/REST 服务器中的 GetArticleID 服务方法:

```
procedure TForm17.Button6Click(Sender: TObject);
var
  aServer : TServerMethods5Client;
  cm : TClientModule1;
begin
  cm := TClientModule1.Create(nil);
  try

cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionUser] := edtUserName.Text;

cm.sqlcnDataSnapServer.ConnectionData.Properties.Values[TDBXPropertyNames.DSAuthentica
tionPassword] := edtPassword.Text;
    aServer := cm.ServerMethods5Client;
    Edit5.Text := IntToStr(aServer.GetArticleID);
  finally
    cm.Free;
  end;
end;
```

由于 GetArticleID 服务方法在 DataSnap/REST 服务器中已经被定义为只有 Admin 群组和 Gordon 可以呼叫，因此如下图所示，当我们呼叫 GetArticleID 而没有输入用户名称和密码时，客户端会被指定为 guest 角色群组，而 guest 角色群组是无法呼叫 GetArticleID 的，因此客户端会得到一个呼叫例外错误:

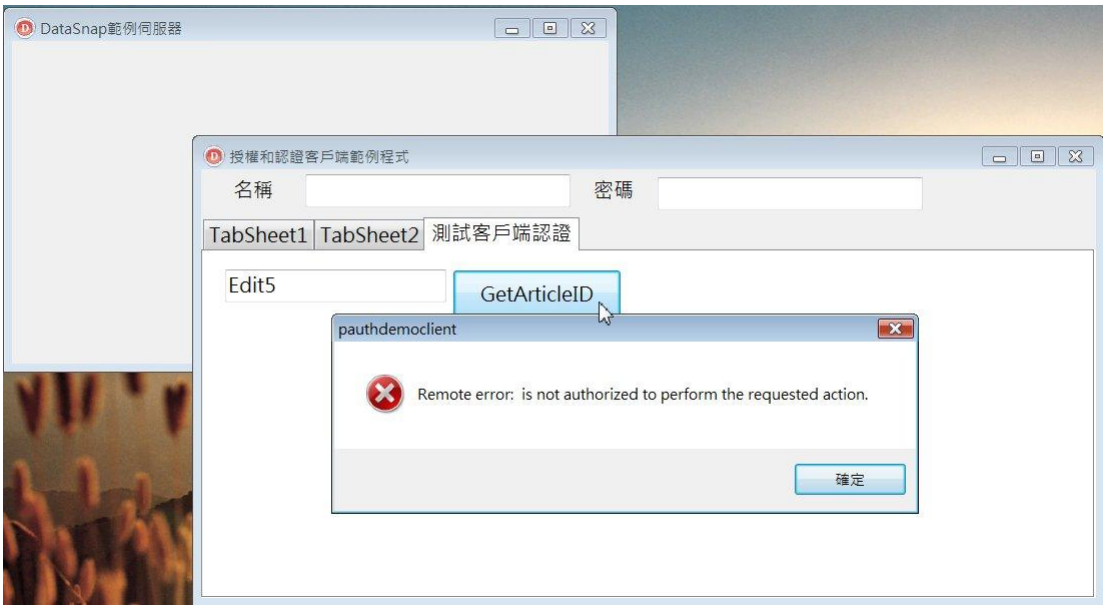


图 3-10 呼叫 GetArticleID 失败，因为没有输入用户认证信息

但是当客户端使用 admin 或是 Gordon 这两个使用者登录时，如下面两个图形所示，就可以成功呼叫 GetArticleID，因为 admin 和 Gordon 会分别被指定 Admin 和 Master 这两个角色群组，而这 Admin 和 Master 这两个角色群组是被授权可呼叫 GetArticleID 服务方法的。

图 3-11 呼叫 GetArticleID 成功，因为 admin 和 Gordon 这两个使用者都允许呼叫 GetArticleID

使用 JavaScript 客户端传递认证信息

除了 Window 的客户端应用程序之外，DataSnap XE 也可以产生 JavaScript 的程序代码来使用 DataSnap 的认证和授权的机制，因此允许任何支持 JavaScript 的开发工具，框架或是程序语言使用 DataSnap 的认证和授权的机制。DataSnap XE 提供了数个内建的 JavaScript 档案允许客户端使用 JavaScript 来连结和使用 DataSnap XE。这些 JavaScript 档案不但允许客户

端使用 JavaScript 来连结和使用 DataSnap 或是 DataSnap REST 服务器，也可以使用 DataSnap XE 提供的回叫机制和认证和授权的机制等进阶的功能，开发人员只需要了解如何使用这些内建的 JavaScript 档案，再加入 DataSnap XE 能够自动产生 DataSnap 或是 DataSnap REST 服务器以 JavaScript 封装的远程服务方法，如此一来客户端几乎就可以使用这些内建的 JavaScript 档案以及自动产生的 JavaScript 档案来使用任何的 DataSnap 功能。由于本章是的重点是讨论认证和授权的机制，因此本小节讨论的重点就是如何藉由 JavaScript 让非 Delphi/C++Builder 的客户端使用 DataSnap XE 的论认证和授权的机制。

首先让我们在 Delphi IDE 中建立一个 HTML 档案，如下所示:

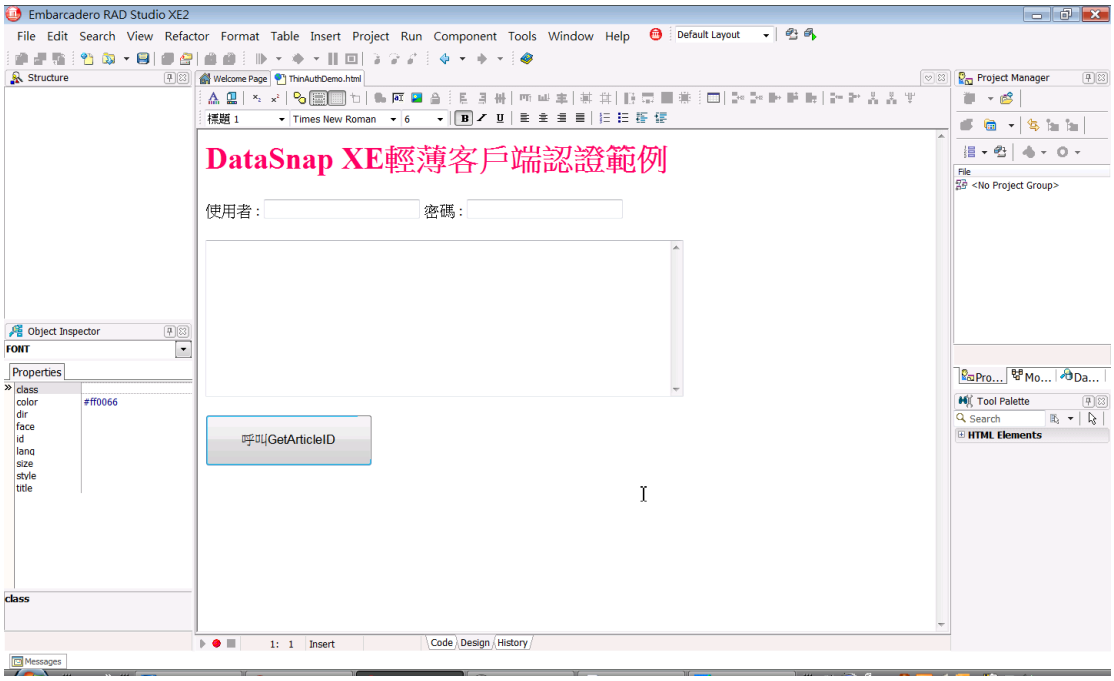


图 3-12 在 Delphi IDE 中建立一个 HTML 档案并且设计图形用户接口

要让客户端使用 JavaScript 使用 DataSnap XE 的论认证和授权的机制，那么必须在 JavaScript 中执行下面的步骤:

1. 撷取客户端输入的用户名称和密码

2. 使用 DataSnap XE 内建的 base64.js 档案中的 convertStringToBase64 函式以下面的格式把客户端输入的用户名称和密码转换为 Base64 编码的格式:

```
使用者名称:密码
```

3. 使用 DataSnap XE 内建的 connection.js 档案中的 connectionInfo 格式建立一个 connectionInfo 的变量。connectionInfo 格式如下:

```
{"authentication":步骤 2 产生的结果}
```

4. 使用 JavaScript 程序代码建立 DataSnap/REST 服务器中的服务类别对象并且把步骤 3 产生的结果传入做为参数

5. 使用步骤 4 建立的服务类别对象呼叫远程 DataSnap/REST 服务器提供的服务

从上面的步骤来看藉由 JavaScript 使用 DataSnap XE 的功能并不困难，因为关键的 JavaScript 程序代码都已经由 Delphi/C++Builder 帮开发人员完成了，开发人员只需要了解如何使用这些内建和自动产生的 JavaScript 程序代码即可。下面就是一个 JavaScript 档案藉由 DataSnap XE 的论认证和授权的机制呼叫前面范例 DataSnap 服务器的实作程序代码。

在 007~012 行中加入使用 Delphi 内建的 JavaScript 程序代码，010 行的 ServerFunctions.js 是由 DataSnap XE 框架自动根据 DataSnap/REST 服务器产生的封装服务器服务的 JavaScript 档案。

```
001   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
002   <html>
003     <head>
004       <title>DataSnap XE 轻薄客户端认证范例
005   </title>
006   <meta HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=UTF-8"><meta
http-equiv="cache-control" content="no-cache">
007       <script type="text/javascript" src="base64.js"></script>
008       <script type="text/javascript" src="json-min.js"></script>
009       <script type="text/javascript" src="serverfunctionexecutor.js"></script>
010       <script type="text/javascript" src="ServerFunctions.js"></script>
011       <script type="text/javascript" src="connection.js"></script>
012       <script type="text/javascript">
013
014       function callServer()
015       {
016         var user = document.getElementById("edtName").value;
017         var password = document.getElementById("edtPassword").value;
018         var auth = convertStringToBase64(user + ":" + password);
019         var connectionInfo = {"authentication":auth};
020
021         var server = new TServerMethods5(connectionInfo);
```

```
022         var result = server.GetArticleID();
023
024         if(result != null)
025         {
026           var resultId = result.result;
027         }
028         else
029         {
030           var resultId = result.toJSONString();
031         }
032
033         document.getElementById("mmCallback").value = resultId;
034       }
035
036       </script>
037     </head>
038     <body>
039     <div>
040        <h1><font color="#ff0066">DataSnap XE 轻薄客户端认证范例
041   </font></h1>
042        <form onsubmit="callServer(); return false;">
043           <p>使用者 : <input id="edtName"> 密码 : <input id="edtPassword"></p>
044          <textarea rows="10" cols="60" id="mmCallback"></textarea><br><br />
045          <input id="btncallServer" type="submit" value="呼叫 GetArticleID" style="WIDTH:
217px; HEIGHT: 67px" size="34" />
046        </form>
047     </div>
048     </body>
049   </html>
```

016~017 行使用 JavaScript 撷取使用者在浏览器中输入的用户名称和密码，018 行执行前面讨论的步骤 2，019 行执行前面讨论的步骤 3，021 行执行前面讨论的步骤 4，之后就可以使用 021 行建立的远程服务对象来呼叫 DataSnap/REST 服务器中提供的服务了。

现在让我们试着在浏览器中执行这个范例 HTML 档案，下图是在浏览器中加载和执行此范例 HTML 档案的画面:

图 3-13 在浏览器中执行范例 HTML 档案

接着以 guest 登录并且试着呼叫远程 DataSnap/REST 服务器中的 GetArticleID 方法，由于 guest 群组没有权限呼叫 GetArticleID，因此浏览器无法呼叫 GetArticleID:

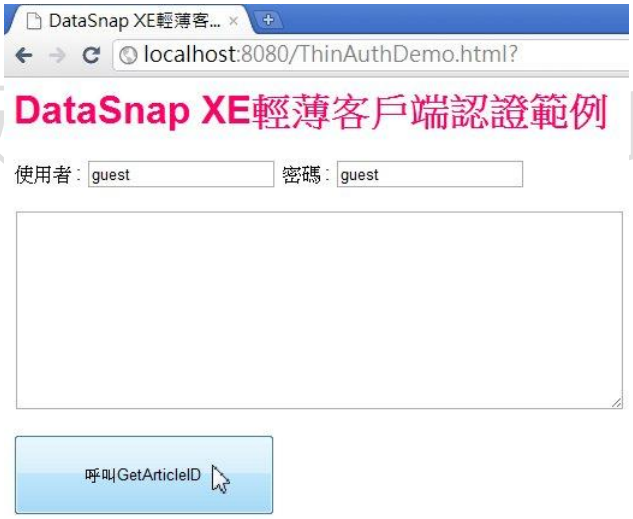


图 3-14 使用 guest 登录无法在浏览器中呼叫 GetArticleID

接着如果我们以 Gordon 登录，再呼叫 GetArticleID 方法，由于 Gordon 的群组拥有呼叫 GetArticleID 的授权，因此浏览器成功的取得了执行 GetArticleID 方法的结果:

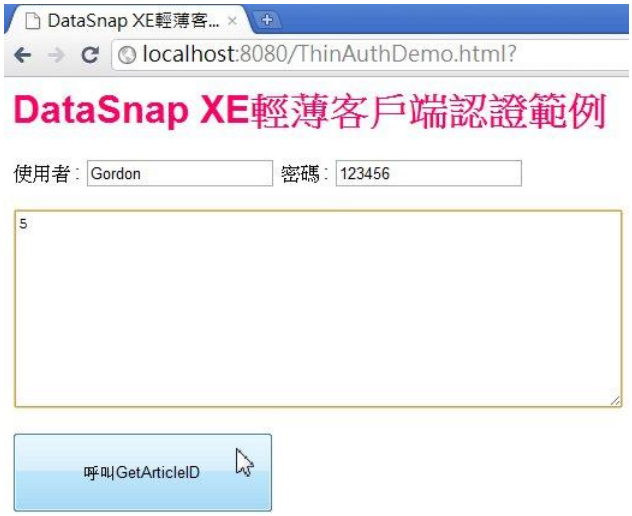


图 3-15 使用 Gordon 登录则可在浏览器中呼叫 GetArticleID

当开发人员在 Delphi 整合发展环境中建立 DataSnap REST 应用程序项目时，Delphi 会自动在项目中产生相关的 JavaScript 档案并且储存于项目的 js 子目录之下，同时在项目中也可以看到这些 JavaScript 档案，当开发人员开发了服务器服务方法并且执行 DataSnap REST 服务器时，DataSnap XE 也会自动在 js 子目录中产生封装服务器服务方法的 ServerFunctions.js 档案，之后开发人员就可以使用它来让非 Delphi/C++Builder 的客户端藉由这些 JavaScript 档案来呼叫 DataSnap/REST 服务器了。

使用 TDSAuthenticationManager 的特性值编辑器授权

使用程序代码来进行授权的控制是最具弹性的方法，但 Delphi 仍然提供了其他的方法让开发人员进行授权的控制，第 2 种方法就是直接在整合发展环境中使用对象查看器来进行。请开启范例 DataSnap/REST 服务器项目中的 ServerContainer 模块，点选 TDSAuthenticationManager 组件，于对象查看器中 TDSAuthenticationManager 组件定义了 Roles 特性，如下所示：

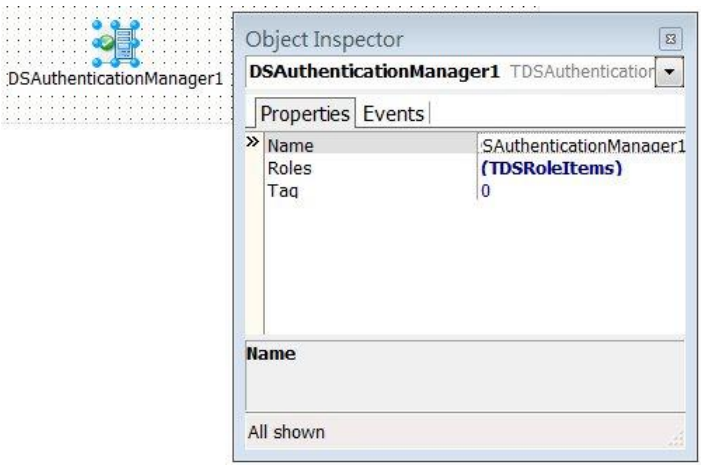


图 3-16 点选 TDSAuthenticationManager 组件，使用对象查看器进行授权的控制

请双击 Roles 特性，对象查看器便会启动 Roles 特性的特性值编辑器，开发人员就可以开始为每一个服务方法定义什么角色群组可以呼叫，什么角色群组无法呼叫。双击 Roles 特性之后 Roles 特性值编辑器后，请点选左上方的新增按钮，此时在对象查看器中就可以定义一个授权控件目，其中开发人员可以定义三个子项目，它的们的说明如下:

特性名称	说明
ApplyTo	这个授权适用的服务方法
AuthorizedRoles	可呼叫此服务方法的角色群组
DeniedRoles	不可呼叫此服务方法的角色群组

在定义 ApplyTo 特性值时，开发人员可以使用 3 种格式:

ApplyTo格式	说明	范例
类别名称	此授权定义适用此类别所有的服务方法	如果是 TServerMethods5 类别，那么所有此类别的方法都适用此授权定义
类别名称.服务方法名称	此授权定义只适用此类别的此服务方法	如果是 TServerMethods5.GetArticle，那么此授权只适用此服务方法
服务方法名称	任何类别只要拥有相同的服务方法名称就适用此授权定义	如果是 GetArticle，那么任何类别的 GetArticle 方法都适用此授权定义

例如我们在 Roles 特性启动特性值编辑器，就可以如下图定义 ApplyTo，AuthorizedRoles 和 DeniedRoles 3 个子特性值:

图 3-17 使用 Roles 特性的特性值编辑器定义授权控制

而下图显示了我们定义 GetArticle 服务方法只能由 Master 角色群组呼叫，guest 和 Admin 角色群组都无法呼叫：

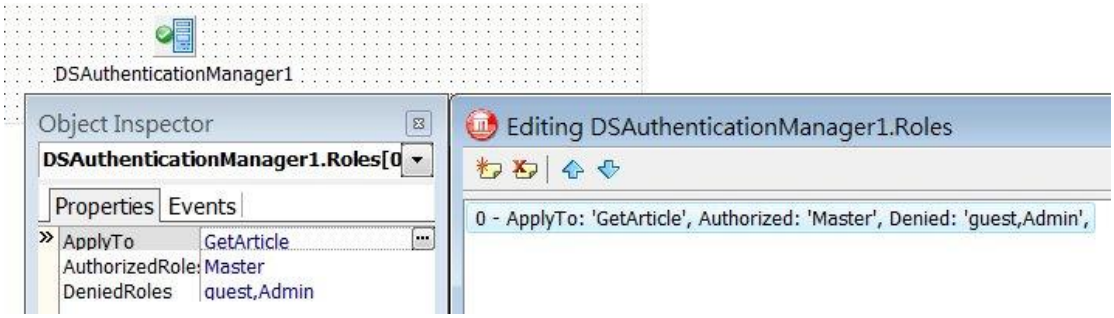


图 3-18 定义 GetArticle 服务方法只能由 Master 角色群组呼叫，guest 和 Admin 角色群组无法呼叫

下图则是使用 Roles 特性值编辑器完整的定义和前面使用程序代码进行授权控制相同的效果：

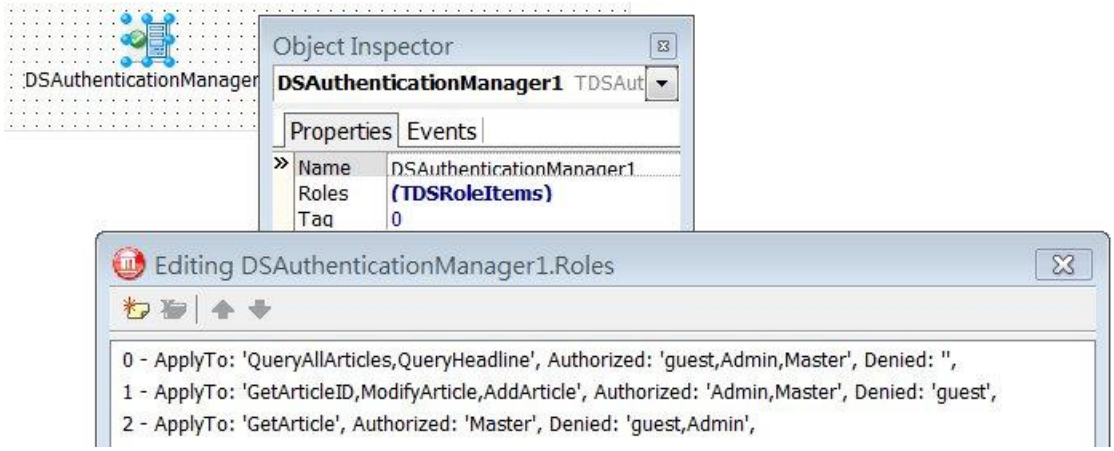


图 3-19 藉由 TDSAuthenticationManager 的 Roles 特性值编辑器来定义授权的控制

使用程序代码批注授权

最后一种定义授权控制的方法是使用程序代码批注，开发人员只需要在类别或是类别的方法程序代码中使用 Delphi 程序语言的批注功能来定义什么角色群组可以呼叫什么类别或是什么类别方法即可，而无需在 OnUserAuthorize 事件处理函式使用程序代码来控制。

DataSnap XE 定义了 TRoleAuth 类别来定义批注授权，它的宣告如下：

```
TRoleAuth = class(TCustomAttribute)
…
public
  …
```

```
   constructor Create(AuthorizedRoles: String; DeniedRoles: String = ''); overload;
virtual;
   constructor Create(AllowRoles: TStrings; DenyRoles: TStrings;
                  DesignTime: Boolean = False); overload; virtual;
…
```

TRoleAuth 有两个建构函式，都可以接受可呼叫服务方法的角色群组和不可呼叫服务方法的角色群组。因此现在让我们开启范例 DataSnap/REST 服务器项目中的 ServerMethods 程序单元，然后在类别定义中使用下面的程序代码批注来定义每一个服务方法的授权定义。由于 TRoleAuth 的第一个参数是可呼叫服务方法的角色群组，第二个参数是不可呼叫服务方法的角色群组，因此我们可以使用 CSV 的字符串格式来定义这两个角色群组。

```
   {访客可呼叫的方法}
   [TRoleAuth('guest, Admin, Master', '')]
   function QueryAllArticles : TJSONArray;
   [TRoleAuth('guest, Admin, Master', '')]
   function QueryHeadline(Name : string) : string;

   {Admin 可呼叫的方法}
   [TRoleAuth('Admin, Master', 'guest')]
   function GetArticleID : Integer;
   [TRoleAuth('Admin, Master', 'guest')]
   function ModifyArticle(jaArticle : TJSONArray) : boolean;
   [TRoleAuth('Admin, Master', 'guest')]
   function AddArticle(Name : string; Headline : string) : boolean;

   {只有 Gordon 可呼叫的方法}
   [TRoleAuth('Master', 'guest, Admin')]
   function GetArticle(Name : String) : TDataSet;
```

现在我们就可以重新编译范例 DataSnap/REST 服务器并且执行它，再启动范例客户端来呼叫它的服务方法，下图显示了 Admin 群组果然无法呼叫 GetArticle 方法:

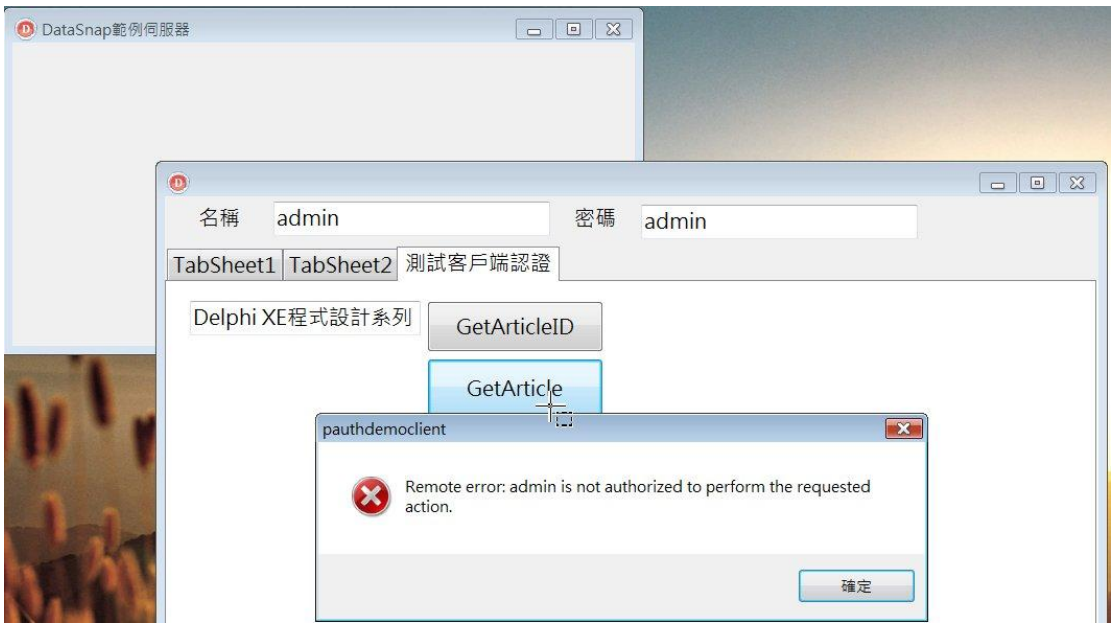


图 3-20 藉由批注来定义授权的控制，Admin 角色群组不能呼叫 GetArticle 服务方法

而下图也显示了 Master 角色群组果然可以藉由程序代码批注方式来呼叫 GetArticle 方法:

图 3-21 藉由批注来定义授权的控制，只有 Master 角色群组能呼叫 GetArticle 服务方法

您已经掌握了 DataSnap 的认证和授权的机制了，现在该您自己动手试试了。

第4章 DataSnap回叫机制

DataSnap 从 2010 版便开始加入回叫(Call Back)机制，当伺服端方法在执行的过程中可以回叫客户端提供的方法以通知客户端有关伺服端方法执行的状态。DataSnap 10.3 之后又大幅强化了回叫机制的功能，加入了回叫信道(Callback Channel)以及信道广播 (Channel Broadcast)等新功能，本章讨论的重点便在说明如何使用 DataSnap 的各种回叫功能。

DataSnap 的回叫机制非常适合使用在需要较长时间执行的伺服端方法，例如如果伺服端方法需要执行长时间的查询时就很适合使用，或是当程序启动或是执行时需要进行许多查询的工作，那么也都可以使用回叫机制。

使用 DataSnap 最基本的回叫机制非常的简单，开发人员只需要实作一个从 TDBXCallback 继承下来的实体类别，并且在呼叫伺服端方法时把此实体类别的样例当做参数传递给伺服端方法即可。至于回叫信道和信道广播则需要更多的手续，但也提供了更强大的功能。在一开始我们将先说明如何使用 DataSnap 最基本的回叫机制，接着再说明回叫信道和信道广播等功能。

4-1 DataSnap 基本的回叫机制

由于使用 DataSnap 2010 基本的回叫机制并不困难，开发人员只需要遵照下面的步骤即可完成:

- 在客户端建立一个从 TDBXCallback 继承下来的实体类别对象并且复载实作虚拟方法 Execute
- 呼叫服务器方法时传递上述的对象给伺服端方法
- 实作伺服端方法时在每一个执行步骤之后呼叫客户端传递来的对象中的虚拟方法 Execute 以代表执行进度

让我们使用一个范例来说明读者就可以很快的了解如何完成上述的步骤。

在下列的范例中本文将使用 Delphi 中新的 System.IOUtils 程序单元中的类别进行档案搜寻和计数的工作，由于这将花上一些时间，因此我们正好使用它来展示使用同步和回叫机制的差异。

范例 DataSnap 服务器

首先让我们建立一个 DataSnap 伺服端，下面是这个服务器输出的伺服端方法，请注意的是，TServerMethods1 输出了两个方法 GetServerDirectoryInfo 和 GetServerDirectoryInfoAsync。这两个方法都执行相同的工作，它们使用 TDirectory 类别搜寻和计数特定伺服端目录下的档案总数，它们的差异是 GetServerDirectoryInfo 使用同步的方式执行，因此当客户端呼叫它时，客户端会同时暂停反应直到 GetServerDirectoryInfo 执行完毕。

而 GetServerDirectoryInfoAsync 则是使用回叫机制的方式执行，因此当客户端呼叫它之后，GetServerDirectoryInfoAsync 在执行的过程中则可以藉由客户端传递来的参数 **ACallback: TDBXCallback**，来回叫回客户端，告诉客户端执行的状态，客户端因此也根据目前伺服端执行的情形来更新客户端的信息。

```
{$METHODINFO ON}
 TServerMethods1 = class(TPersistent)
 private
   { Private declarations }
   fTotalFiles : Integer;
   FResult : TJSONArray;
   FCallback: TDBXCallback;

   procedure ProcessPath(const sPath : string);
   procedure ProcessPathAsync(const sPath : string);
   procedure ShowMessage(sMessage : string);
   procedure ProcessThisDirectory(const sPath : string);
 public
   { Public declarations }
   function EchoString(Value: string): string;
   function GetServerDirectoryInfo(const sPath : string) : TJSONArray;
   function GetServerDirectoryInfoAsync(ACallback: TDBXCallback; const sPath : string):
TJSONArray;
 end;
{$METHODINFO OFF}
```

GetServerDirectoryInfoAsync 方法是如何回叫回客户端呢?其实非常的简单，因为客户端在呼叫它时已经把客户端的回叫方法当成参数传递过来了，因此 GetServerDirectoryInfoAsync 方法只需要藉由这个参数即可回叫回客户端。

因此我们可以从下面 18 行的程序代码看到，伺服端直接使用这个回叫参数呼叫客户端，并且建立一个 TJSONString 型态的对象做为参数，在这个 TJSONString 对象中告诉了客户端目前伺服端正在处理那一个目录。

```
001   function TServerMethods1.GetServerDirectoryInfoAsync(ACallback: TDBXCallback;
002     const sPath: string): TJSONArray;
003   begin
004     FCallback := ACallBack;
005     FTotalFiles := 0;
006     FResult := TJSONArray.Create;
007     ProcessPathAsync(sPath);
008     FResult.AddElement(TJSONString.Create('总档案数' + ' : ' + IntToStr(FTotalFiles)));
009     Result := FResult;
010   end;
011
012   procedure TServerMethods1.ProcessPathAsync(const sPath: string);
013   var
014     rootDirectories : TStringDynArray;
015     i: Integer;
016   begin
017     ProcessThisDirectory(sPath);
018     FCallback.Execute(TJSONString.Create('处理目录' + sPath + '中...'));
019     rootDirectories := TDirectory.GetDirectories(sPath);
020     for i := 0 to Length(rootDirectories) - 1 do
021       ProcessPathAsync(rootDirectories[i]);
022   end;
```

同步客户端

范例的同步客户端非常的简单，它只是直接呼叫伺服端的 GetServerDirectoryInfo 方法。

```
procedure TForm3.btnGetServerInfoClick(Sender: TObject);
var
  aServer : TServerMethods1Client;
```

```
  ja : TJSONArray;
  jv : TJSONValue;
  I: Integer;
begin
  lStart := GetTickCount;
  aServer := TServerMethods1Client.Create(Self.SQLConnection1.DBXConnection);
  try
    ja := aServer.GetServerDirectoryInfo(Edit1.Text);
    lEnd := GetTickCount;
    for I := 0 to ja.Size - 1 do
    begin
      jv := ja.Get(I);
      lbResult.Items.Add(jv.ToString);
    end;
  finally
    aServer.Free;
    ShowRunTime(lEnd - lStart);
  end;
end;
```

在客户端呼叫 GetServerDirectoryInfo 方法的过程中，客户端暂停反应，用户也无从了解伺服端执行的状态。

回叫客户端

再看看回叫客户端，这个客户端的关键从下面的 012 行开始，012 行建立了 TDSCallbackWithMethod 对象，并且建立一个匿名方法做为客户端的回叫方法传递给伺服端。从 013 行开始的匿名方法在被客户端回叫的时候首先在 013 行把伺服端传递来的参数型态转换为 TJSONString 的型态，接着更新客户端的 UI 以通知用户伺服端目前正在处理那一个目录。最后在 023 行客户端回叫方法如果执行成功就需要回传 TJSonTrue 对象，如果失败的话就需要回传 TJSonFalse 对象。

```
001   procedure TForm3.btnGetServerInfoAsyncClick(Sender: TObject);
002   var
003     aServer : TServerMethods1Client;
004     LCallback : TDSCallbackWithMethod;
005     ja : TJSONArray;
006     jv : TJSONValue;
007     I: Integer;
```

```
008   begin
009     lStart := GetTickCount;
010     aServer := TServerMethods1Client.Create(Self.SQLConnection1.DBXConnection);
011     try
012       LCallback := TDSCallbackWithMethod.Create(
013         function(const Args: TJSONValue): TJSONValue
014         var
015           asyncResult: TJSONString;
016           I: Integer;
017           LMessage: string;
018         begin
019           asyncResult := TJSONString(Args);
020           lbAsync.Items.Add(asyncResult.ToString);
021           lbAsync.Update;
022           Application.ProcessMessages;
023           Result := TJSonTrue.Create;
024         end
025         );
026       ja := aServer.GetServerDirectoryInfoAsync(LCallback, Edit1.Text);
027
028
029       lEnd := GetTickCount;
030       for I := 0 to ja.Size - 1 do
031       begin
032         jv := ja.Get(I);
033         lbResult.Items.Add(jv.ToString);
034       end;
035     finally
036       aServer.Free;
037       ShowRunTime(lEnd - lStart);
038     end;
039   end;
```

那么什么是 TDSCallbackWithMethod 类别呢？这要从 TDBXCallback 抽象类开始谈起。

TDBXCallback 抽象类

在讨论 TDSCallbackMethod 之前我们必须先说明 TDBXCallback 抽象类，因为 TDSCallbackMethod 是从 TDBXCallback 继承下来的实体类别。事实上 TDBXCallback 类别即是使用 DataSnap 回叫机制的关键，要使用回叫机制，开发人员必须实作一个从 TDBXCallback 继承下来的实体类别，并且传递此实体类别的样例给伺服端方法做为参数，如此一来伺服端方法就可以藉由这个样例参数呼叫回客户端，以通知客户端伺服端方法执行的状态。

TDBXCallback 的虚拟方法 Execute 是衍生类别需要复载实作的，Execute 接受一个型态为 TJSONValue 的参数并且回传一个型态为 TJSONValue 的结果值，伺服端方法在回叫客户端的方法时，可以把需要传递给客户端的数值或是对象转换为 TJSONValue 型态并且当成 Execute 方法的参数传递回客户端，而客户端的方法在被回叫执行完毕之后，也可以把执行结果转换为 TJSONValue 型态并且回传给伺服端。下面即是 TDBXCallback 抽象类的宣告:

```
001     TDBXCallback = class abstract
002     public
003       function Execute(const Arg: TJSONValue): TJSONValue; virtual; abstract;
004     protected
005       procedure SetConnectionHandler(const ConnectionHandler: TObject); virtual;
006       procedure SetOrdinal(const Ordinal: Integer); virtual;
007     public
008       property ConnectionHandler: TObject write SetConnectionHandler;
009       property Ordinal: Integer write SetOrdinal;
010     end;
```

TDSCallbackMethod 实体类别

了解了 TDBXCallback 扮演的角色之后解释 TDSCallbackMethod 实体类别就简单了，由于此范例应用程序要使用异步呼叫机制，因此宣告 TDSCallbackMethod 从 TDBXCallback 继承下来并且实作虚拟方法 Execute。在 TDSCallbackMethod 的 Execute 方法中它只是简单的呼叫在建构函式中储存下来的客户端方法的方法指标。

```
unit AsyncUtils;


interface

```

```
uses
  Classes,
  DbxDatasnap,
  DBXJson;

type
  TDSCallbackMethod = reference to function(const Args: TJSONValue): TJSONValue;
  TDSCallbackWithMethod = class(TDBXCallback)
  private
    FCallbackMethod: TDSCallbackMethod;
  public
    constructor Create(ACallbackMethod: TDSCallbackMethod);
    function Execute(const Args: TJSONValue): TJSONValue; override;
  end;

implementation

constructor TDSCallbackWithMethod.Create(ACallbackMethod: TDSCallbackMethod);
begin
  FCallbackMethod := ACallbackMethod;
end;

function TDSCallbackWithMethod.Execute(const Args: TJSONValue): TJSONValue;
var
  aString: string;
begin
  Assert(Assigned(FCallbackMethod));
  Result := FCallbackMethod(Args);
end;

end.
```

下面的图形分别是 DataSnap 伺服端执行的画面以及同步客户端，回叫客户端的执行画面，从图 2 同步客户端画面可以看到它虽然比回叫客户端执行的快(在笔者的机器中执行了 7.078 秒)，但是在同步客户端呼叫 DataSnap 服务器时它的整个 UI 是暂停的，因此它无法在窗体下方的 ListBox 中显示任何伺服端执行的信息，用户必须等待它完全执行完毕才能取回客户端应用程序的控制权。

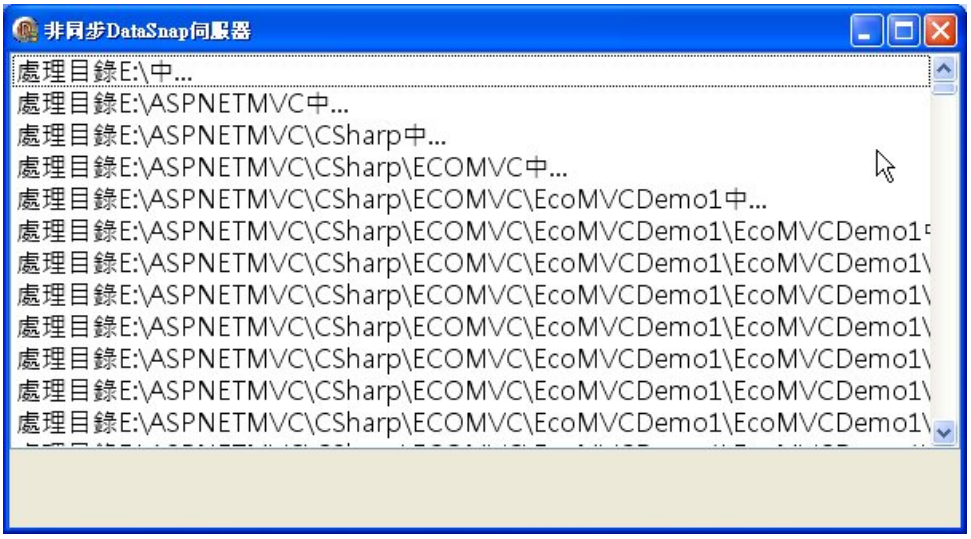


图 1 DataSnap 服务器执行画面

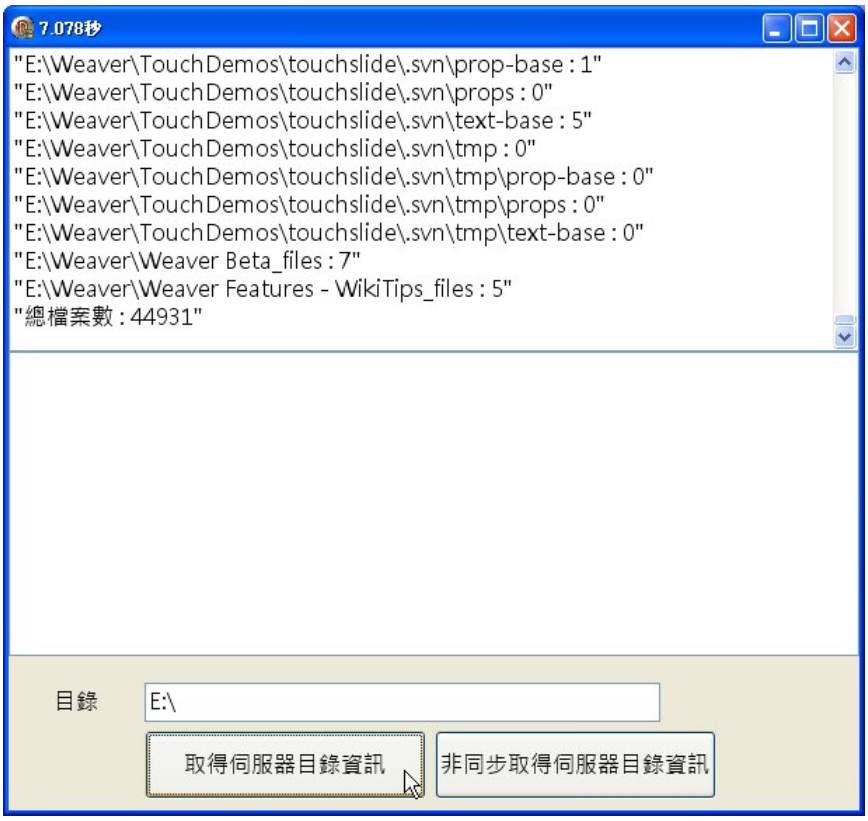


图 2 同步客户端呼叫 DataSnap 服务器执行结果

相反的画面 3 则是回叫客户端的执行结果，我们看到它是比同步客户端慢(在笔者的机器中执行了 12.984 秒)，但是在整个执行过程中使用者仍然可以控制客户端应用程序，而且回叫客户端能够不停的在窗体下方的 ListBox 中显示目前伺服端正在处理的目录信息。因此就使用者经验来说，回叫客户端是比同步客户端好多了。

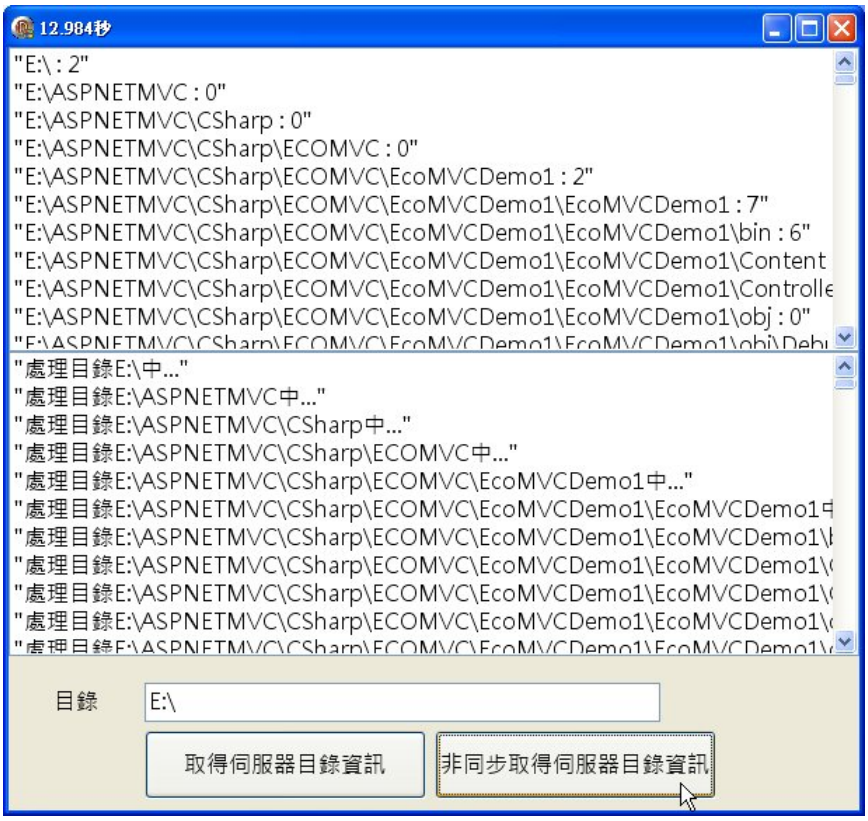


图 3 DataSnap 服务器藉由回叫功能呼叫异步客户端执行画面

4-2 进阶 DataSnap 回叫功能

前一小节讨论了 DataSnap 基本的回叫功能，接着让我们讨论从 DataSnap 10.3 版加入的回叫通道等新的回叫机制。

DataSnap 10.3 在原有的基础回叫机制之上加入了许多强大的新功能，从 DataSnap 10.3 开始开发人员可以使用下面的回叫功能:

- 客户端可向伺服端注册回叫信道，如此一来服务器可以一次回叫所有在同回叫通道中所有注册的客户端回叫函式
- 客户端可以同时注册多个不同的回叫通道
- 客户端可以藉由回叫通道呼叫不同的客户端
- 新增回叫组件以帮助开发人员简化开发回叫机制

DataSnap 10.3 新的回叫功能虽然很多，但使用起来仍然相当的容易，下面说明了如何使用这些 DataSnap 10.3 新的回叫功能的基本步骤:

- 客户端使用 TDSClientCallbackChannelManager 向服务器注册一个回叫通道

- 服务器使用 TDSServer 组件的 BroadcastMessage 方法回叫所有注册的客户端

当然，开发人员可以更进一步的使用 DataSnap 10.3 进阶的回叫功能，不过在那之前也许我们应该先说明数个范例让读者了解如何使用这些基本的步骤。

开发回叫 DataSnap 服务器

在 Delphi 整合发展环境中建立一个 DataSnap Server 项目：

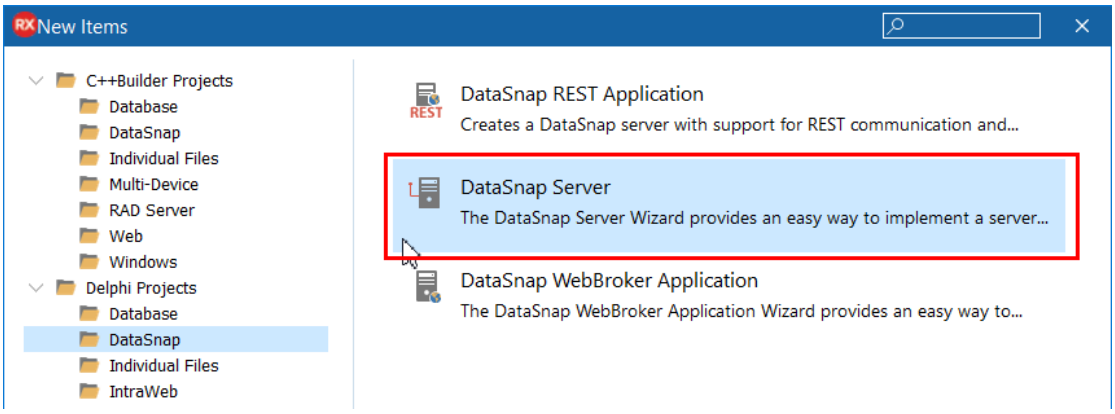


图 4 建立 DataSnap 回叫服务器

在设定此服务器的特性时，让我们目前只选择使用 TCP/IP 通讯协议，如下图所示：

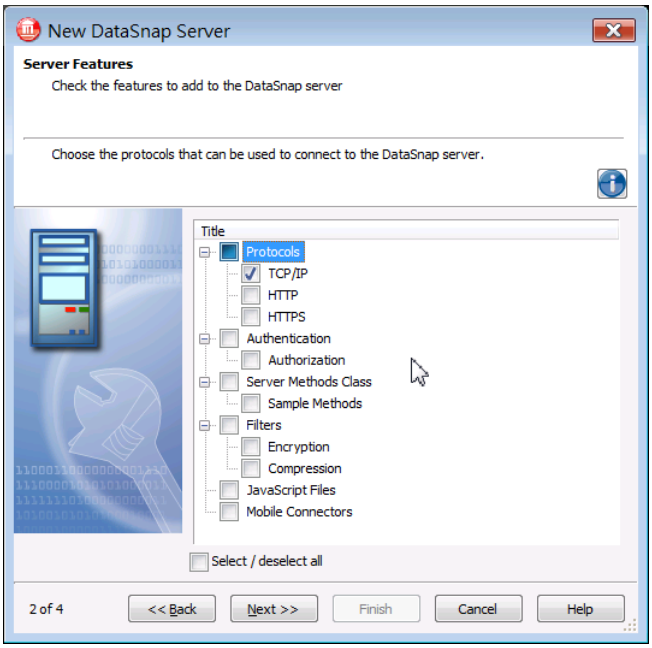


图 5 建立 DataSnap 回叫服务器，目前只选择使用 TCP/IP 通讯协议

开启项目中的 ServerContainer 程序单元，此时在 ServerContainer 中产生了两个组件，TDSServer 以及 TDSTCPServerTransport，由于接下来我们将先展示 Windows 客户端的回叫功能，因此现在使用 TCP/IP 通讯协议就足够了。

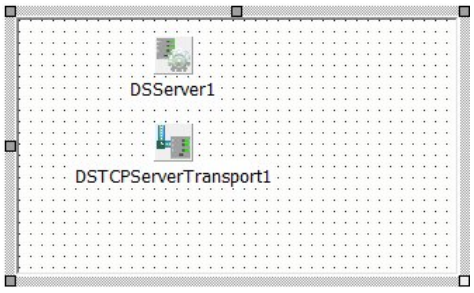


图 6 ServerContainer 包含的组件

现在开启项目主窗体，并且在其中置入如下的组件：

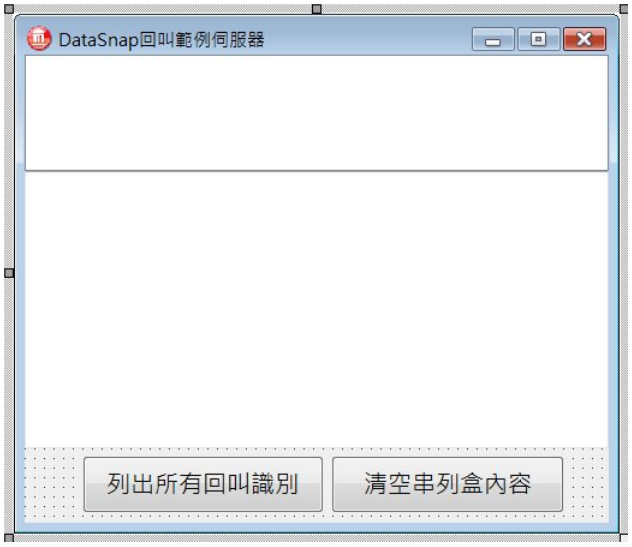


图 7 DataSnap 回叫服务器主窗体

主表格上方使用了 TListBox 组件，它可以显示所有客户端注册的回叫识别 ID，而下方的 TMemo 组件则是使用来回叫注册客户端，在稍后我们将在此 TMemo 组件中输入信息，这些输入的信息就会藉由回叫通道自动传递给客户端。

DataSnap 服务器要回叫所有注册的客户端是非常容易的，只需要藉由 TDSServer 类别定义的 BroadcastMessage 方法即可，TDSServer 类别中定义了两个 BroadcastMessage 方法原型如下:


```
function BroadcastMessage(const ChannelName: String; const Msg: TJSONValue; const ArgType:
Integer = TDBXCallback.ArgJson): boolean; overload;


function BroadcastMessage(const ChannelName: String; const CallbackId: String; const Msg:
TJSONValue; const ArgType: Integer = TDBXCallback.ArgJson): boolean; overload;
```

这两个 BroadcastMessage 方法的差异在于上述第一个 BroadcastMessage 可以传递讯息给它第一个参数 ChannelName 指定的通道中所有的回叫客户端，而第二个 BroadcastMessage 方法则是只传递讯息给它第一个参数 ChannelName 指定的通道中由第二个参数 CallbackId 指定的回叫客户端，最后这个两个 BroadcastMessage 方法传递给客户端的讯息则由第二个参数 Msg 封装。

了解了如何使用 BroadcastMessage 方法之后，我们就可以看看如何把 DataSnap 服务器中于 TMemo 组件中输入的讯息传递给回叫客户端。现在为主窗体中的 TMemo 组件实作 OnChange 事件处理函式如下:

```
001    procedure TForm17.mmMessageChange(Sender: TObject);
002    var
```

```
003     vMessage : TJSONString;
004   begin
005     vMessage := TJSONString.Create(mmMessage.Lines.Text);
006     ServerContainer5.DSServer1.BroadcastMessage(DEMOChannel, vMessage);
007   end;
```

在 005 行我们把输入于 TMemo(mmMessage)中的信息以 TJSONString 对象封装，然后在 006 行藉由呼叫 ServerContainer 中的 TDSServer 组件的 BroadcastMessage 方法传递给所有注册的客户端。但谁是注册的客户端呢?请看 BroadcastMessage 的第一个参数 DEMOChannel，这代表 DataSnap 服务器会传递信息给所有在 DEMOChannel 信道中注册的客户端。而 DEMOChannel 是一个通道的名称，我们在 DataSnap 服务器中定义它如下:

```
const
  DEMOChannel = 'DemoChannel';
```

因此客户端只要使用这个名称向服务器注册回叫通道的话，就可以让 DataSnap 服务器回叫客户端，当然也客户端可以先向服务器查询已经定义在 DataSnap 服务器中的回叫通道名称，或是由客户端自行在 DataSnap 服务器中建立指定名称的回叫通道。

由于回叫客户端是向 DataSnap 服务器中指定名称的回叫通道注册，而且每一个客户端都使用一个特定的回叫识别 ID 来代表，因此我们也可以藉由 TDSServer 组件来查询某一个名称的回叫通道中所有注册的客户端回叫识别 ID。

此范例 DataSnap 服务器主窗体中的『列出所有回叫识别』按钮的 OnClick 事件处理函式就可以在主窗体上方的 TListBox 中列出特定回叫通道中所有的客户端回叫识别 ID，下面就是它的 OnClick 实作程序代码:

```
001   procedure TForm17.Button1Click(Sender: TObject);
002   var
003     aIdList : TList<String>;
004     sId : String;
005   begin
006     aIdList := ServerContainer5.DSServer1.GetAllChannelCallbackId(DEMOChannel);
007     try
008       for sId in aIdList do
009         ListBox1.Items.Add(sId);
010     finally
011       aIdList.Free;
```

```
012     end
013   end;
```

GetAllChannelCallbackId 方法会回传 TList<String>型态的执行结果，其中即包含了所有在此通道名称中注册的客户端回叫识别 ID，因此在 006 行藉由 TDSServer 组件呼叫 GetAllChannelCallbackId 之后，就可以取得到在 DEMOChannel 中注册的识别 ID，接着从 008 行到 012 行就把这些识别 ID 显示在主窗体的 TListBox 中。

现在请编译并且执行此范例 DataSnap 服务器，接着我们就可以实作回叫客户端了，首先让我们先说明如何建立 Windows 应用程序型态的客户端，接着再说明如何建立浏览器型态的客户端。

开发回叫 DataSnap 客户端

在项目管理员中再建立一个 VCL Form 应用程序项目，并且在其中放入 TSQLConnection，TDSClientCallbackChannelManager，TMemo，两个 TButton 组件，如下图所示。其中 TSQLConnection 是链接上一小节的范例 DataSnap 服务器，而 TDSClientCallbackChannelManager 则是使用来向 DataSnap 服务器注册回叫客户端的组件。

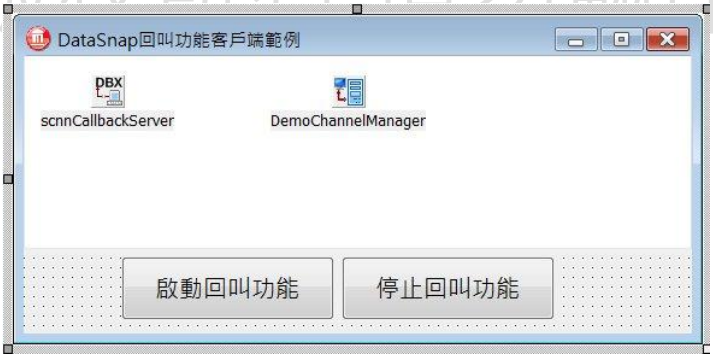


图 8 DataSnap 回叫服务器主窗体使用的组件

接着设定 TDSClientCallbackChannelManager 特性值如下:

特性名称	特性值
ChannelName	DemoChannel
CommunicationProtocol	tcp/ip
DSHostname	localhost
DSPort	211
Name	DemoChannelManager

设定 TDSClientCallbackChannelManager 的 ChannelName 特性值为 DemoChannel 是因为前面范例 DataSnap 服务器使用的信道名称就是 DemoChannel，而且前面范例 DataSnap 服务器是支持 TCP/IP 通讯协议和使用 211 通信埠，因此我们需要设定 TDSClientCallbackChannelManager 相对应的特性值，下图显示了在对象查看器中设定 TDSClientCallbackChannelManager 组件的特性值:

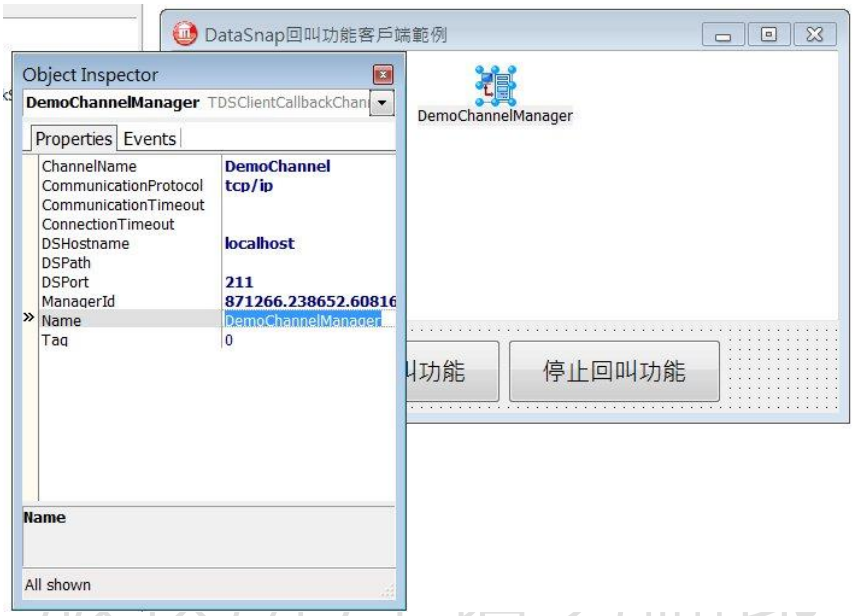


图 9 设定 TDSClientCallbackChannelManager 组件的特性值

设定好 TDSClientCallbackChannelManager 组件之后，我们就可以看看客户端主窗体中的『启动回叫功能』按钮的实作程序代码了:

```
001   procedure TfmMainForm.btnStartClick(Sender: TObject);
002   begin
003     SetupTask;
004     EnableDisableButtons(False, True);
005     DemoChannelManager.RegisterCallback(callbackId, TDemoCallback.Create)
006   end;
007
008   procedure TfmMainForm.SetupTask;
009   begin
010     if not scnnCallbackServer.Connected then
011     begin
012       scnnCallbackServer.Connected := True;
013     end;
014     callbackId := DateTimeToStr(Now);
```

```
015     Self.Caption := callbackId;
016   end;
```

在 btnStartClick 事件处理函式中先于 003 行呼叫 SetupTask 方法以开启 TSQLConnection 组件链接范例 DataSnap 服务器，并且在 014 行根据目前的时间建立一个独特的识别 ID，callbackId。最后在 005 行呼叫 TDSClientCallbackChannelManager 组件的 RegisterCallback 方法向范例 DataSnap 服务器注册这个回叫客户端。

TDSClientCallbackChannelManager 组件的 RegisterCallback 方法原型定义如下:

```
function RegisterCallback(const CallbackId: String; const Callback: TDBXCallback): boolean;
overload;
```

RegisterCallback 接受两个参数，第一个是每一个回叫客户端的识别 ID，第二个参数则是型态为 TDBXCallback 的对象，在前面的小节中我们已经解释过 TDBXCallback 类别，因此在这里我们需要建立一个从 TDBXCallback 衍生类别的对象，在这个衍生类别中我们需要复载虚拟方法 Execute，如此一来范例 DataSnap 服务器就可以藉由呼叫复载虚的拟方法 Execute 来呼叫到客户端。

由于我们在上面的 005 行是传递 TDemoCallback 对象给 RegisterCallback 方法，因此我们需要在这个范例客户端应用程序中定义和实作 TDemoCallback 类别，它需要从 TDBXCallback 继承下来:

```
type
  TDemoCallback = class(TDBXCallback)
  public
    constructor Create;
    function Execute(const Arg: TJSONValue): TJSONValue; override;
  end;
```

TDemoCallback 需要实作虚拟方法 Execute，当 DataSnap 服务器回叫客户端时就会执行虚拟方法 Execute。由于在这个范例中我们希望范例 DataSnap 服务器在 TMemo 中输入的信息能够立刻显示在客户端，因此我们实作虚拟方法 Execute 如下:

```
001   function TDemoCallback.Execute(const Arg: TJSONValue): TJSONValue;
002   var
003     sDemoMessage : String;
004   begin
005     Result := TJSONTrue.Create;
```

```
006
007     if (Arg is TJSONString) then
008     begin
009       sDemoMessage := TJSONString(Arg).Value;
010       TThread.Synchronize(nil,
011                           procedure
012                           begin
013                             fmMainForm.mmDemoMessage.Lines.Text := sDemoMessage;
014                           end
015                           );
016     end;
017   end;
```

当 DataSnap 服务器藉由回叫功能呼叫客户端复载的 Execute 时，DataSnap 服务器可以把伺服端传递到客户端的信息封装在 Execute 的参数 Arg 中。由于在前面的范例 DataSnap 服务器是把在伺服端 TMemo 中的信息封装成 TJSONString 传递到客户端，因此在 007 行先判断伺服端传递来的是否是 TJSONString 型态，如果是的话，就 00 行取出伺服端传递来的信息，接着由于我们需要把这个信息显示在客户端主窗体中的 TMemo 组件中，因此我们藉由呼叫 TThread 类别的类别方法 Synchronize 来更新客户端的用户接口(这是因为客户端用户接口的主线程和在背景的回叫线程是不同的，因此需要使用 Synchronize 来让背景回叫线程更新主线程中的组件)。由于 Synchronize 定义了如下的原型:


```
class procedure TThread.Synchronize(AThread: TThread; AMethod: TThreadMethod);
```

因此在上面的 010 行中我们直接使用匿名方法来更新主窗体的 TMemo 组件。

最后当我们不再需要让客户端被回叫时，可以呼叫 TDSClientCallbackChannelManager 组件的 UnregisterCallback 方法并且传递客户端识别 ID，例如下面的程序代码就是主窗体中『停止回叫功能』按钮 OnClick 事件处理函式的实作程序代码:

```
procedure TfmMainForm.btnStopClick(Sender: TObject);
begin
  EnableDisableButtons(True ,False);
  DemoChannelManager.UnregisterCallback(callbackId);
end;
```

现在编译并且执行此范例回叫客户端，从下图我们可以看到，点选范例 DataSnap 服务器中的『列出所有回叫识别』按钮就可以在上方的 TListBox 中

列出客户端传递来的识别 ID，而且只要客户端应用程序点选了主窗体中的『启动回叫功能』按钮注册回叫客户端，那么我们在范例 DataSnap 服务器的 TMemo 中输入的任何信息就会立刻显示在客户端应用程序的 TMemo 组件中，证明了回叫功能是成功的。

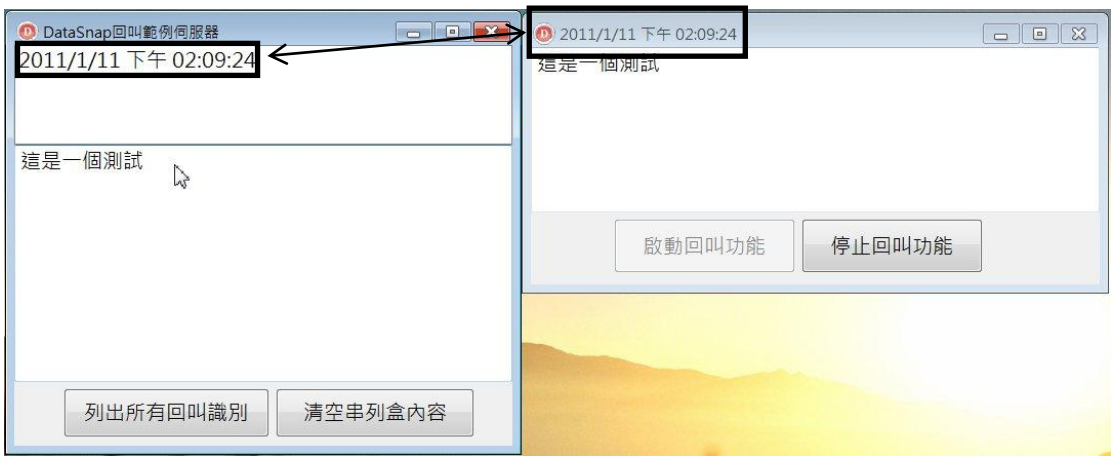


图 10 在范例 DataSnap 服务器中输入的信息可以立刻藉由回叫功能显示在客户端

当然，客户端可以注册多个不同的回叫通道，在同一个回叫通道中也可以注册多个识别 ID，例如让我们修改刚才的范例客户端应用程序，加入另外一个 TMemo 组件，以及一个 TEdit 组件让用户可以输入不同的客户端识别 ID 来注册：

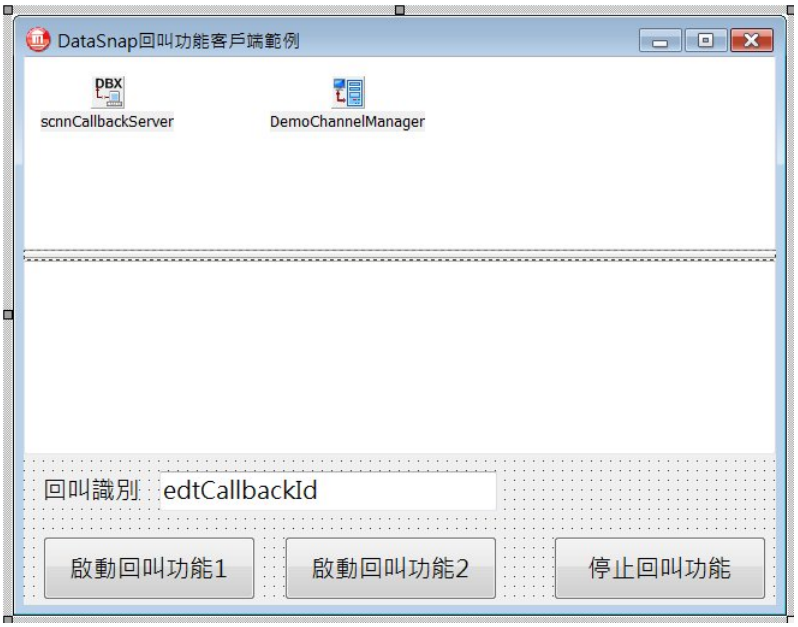


图 11 修改客户端应用程序的主窗体

接着在『启动回叫功能 1』和『启动回叫功能 2』按钮中都使用下面的程序代码实作，现在客户端识别 ID 就由使用者输入而不是使用目前的日期：

```
procedure TfmMainForm.btnStartClick(Sender: TObject);
begin
  SetupTask;
  EnableDisableButtons(False, True, True);
  DemoChannelManager.RegisterCallback(edtCallbackId.Text, TDemoCallback.Create);
  aIdList.Add(edtCallbackId.Text);
end;
```

编译并且执行新的客户端应用程序，在下面的图形中我们可以看到笔者在客户端应用程序中注册了两个客户端识别 ID，分别是『客户端识别 ID1』和『客户端识别 ID2』，而范例 DataSnap 服务器也可以列出这两个不同的客户端识别 ID，在客户端注册了两个不同的识别 ID 之后，范例 DataSnap 服务器可以藉由同时回叫这两个不同的客户端回叫物件。例如下图就显示了当我们在范例 DataSnap 服务器的 TMemo 中输入了讯息之后，这些讯息会立刻显示在客户端应用程序中两个不同的 TMemo 组件之中：

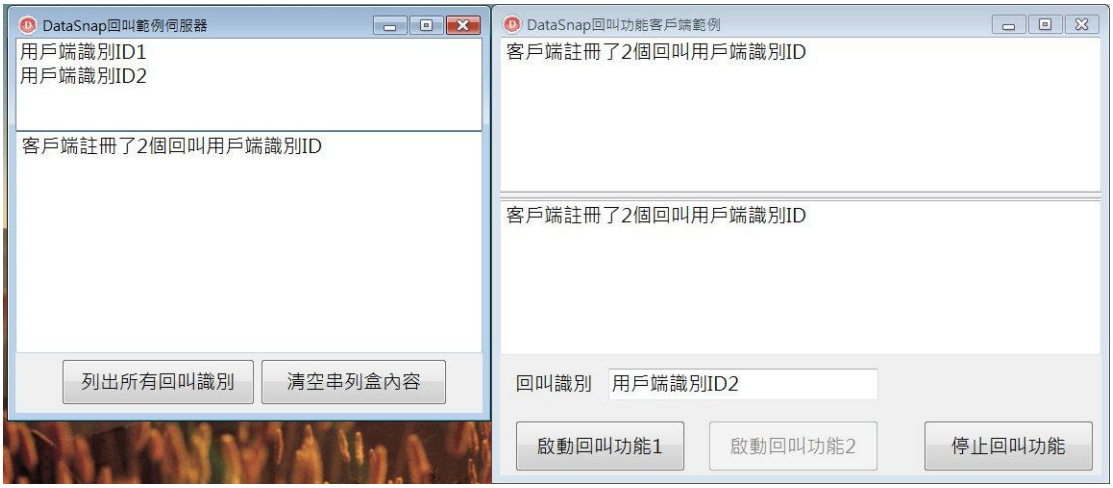


图 12 客户端在同一个回叫通道中注册了两个识别 ID

如何? DataSnap 10.3 的回叫机制是不是更强大了?不过故事还没结束，接下来我们将继续讨论如何让不同的客户端都能够藉由回叫机制来沟通，以及如何开发以浏览器为基础的回叫客户端，这些是更精彩的主题。

不同客户端藉由回叫功能沟通

虽然一个客户端应用程序可以藉由注册一个回叫信道并且在回叫信道中注册多个回叫 ID，但另外一个非常实用的回叫功能就是如何让不同的客户端能够藉由回叫来相互沟通。DataSnap XE 的回叫功能也支持不同的客户端藉由回叫机制来相互传递任何的数据，在说明如何实作之前，让我们总结一下 DataSnap 框架回叫机制可由客户端注册变动的部份包含了：

- 通道名称
- 客户端识别
- 回叫识别

因此要让不同的客户端能够藉由回叫功能互动，那么客户端只要掌握上述的三个变动的即可。例如客户端 1 要和客户端 2 藉由回叫互动，客户端 1 只需要知道客户端 2 存在的通道名称，客户端 2 的客户端识别以及客户端 2 的回叫识别，那么客户端 1 就可以回叫客户端 2 来传递任何信息了。

现在就让我们看看如何实作不同客户端藉由回叫功能相互沟通。

修改 DataSnap 服务器

回到范例回叫项目群组，开启范例 DataSnap 服务器的主窗体，在主窗体上方加入一个 TListBox 以显示所有客户端识别信息，另外再加入两个 TButton，分别是『列出所有客户端识别』以及『清除所有客户端识别』，如下图所示：

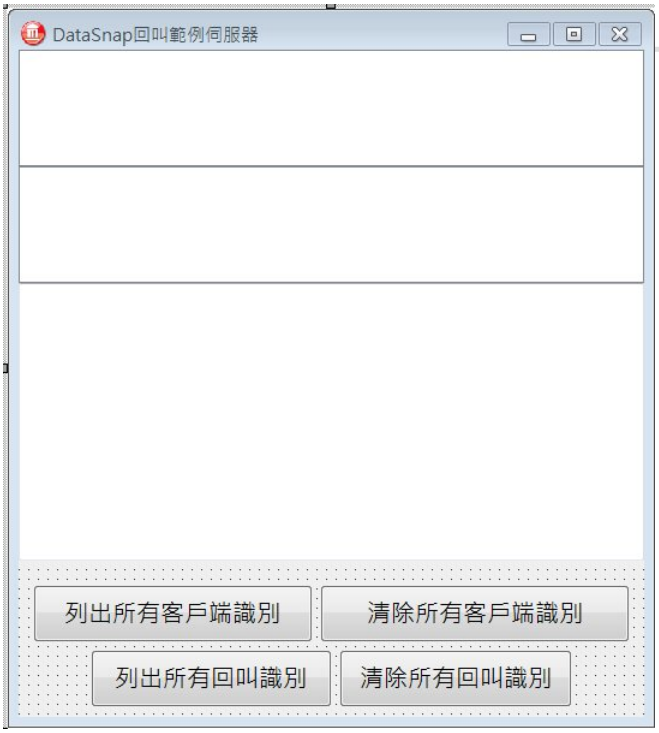


图 13 修改范例 DataSnap 服务器的主窗体

要取得所有链接到 DataSnap 服务器的客户端识别信息，我们可以呼叫 TDSServer 组件的 GetAllChannelClientId 方法，GetAllChannelClientId 能够回传在一个特定的回叫通道中所有注册的客户端识别，它的原型如下：

```
function GetAllChannelClientId(const ChannelName: String): TList<String>;
```

我们只需要传递特定的回叫通道给 GetAllChannelClientId 即可从回传的 TList 对象中取得所有注册的客户端识别，在这个范例中我们只需要传入 DEMOChannel 即可。因此『列出所有客户端识别』按钮的 OnClick 事件处理函式实作如下:

```
procedure TForm17.btnListAllClientIdsClick(Sender: TObject);
var
  aIdList : TList<String>;
  sId : String;
begin
  aIdList := ServerContainer5.DSServer1.GetAllChannelClientId(DEMOChannel);
  try
    for sId in aIdList do
      lbAllClientIds.Items.Add(sId);
  finally
    aIdList.Free;
  end;
end;
```


现在就可以修改范例 DataSnap 客户端应用程序了。

修改 DataSnap 客户端应用程序

开启范例 DataSnap 客户端应用程序的主窗体，在其中加入两个 TEdit 组件以便让用户输入这个客户端的客户端识别以及回叫识别，再置入一个『回叫客户端』按钮以及一个 TMemo 组件，当我们在新的 TMemo 组件中输入任何信息时，再点选『回叫客户端』按钮就可以把新的 TMemo 组件中的信息藉由回叫功能传递给另外一个客户端应用程序。但我们如何指定要回叫那一个客户端应用程序呢?这就由新加入的两个 TComboBox 来指定了，我们可以在这两个 TComboBox 中输入要沟通的另外一个客户端的客户端识别以及回叫识别，那么『回叫客户端』按钮的 OnClick 事件处理函式就可以藉由回叫功能来回叫另外一个客户端应用程序了，此时范例客户端应用程序的主窗体如下所示:

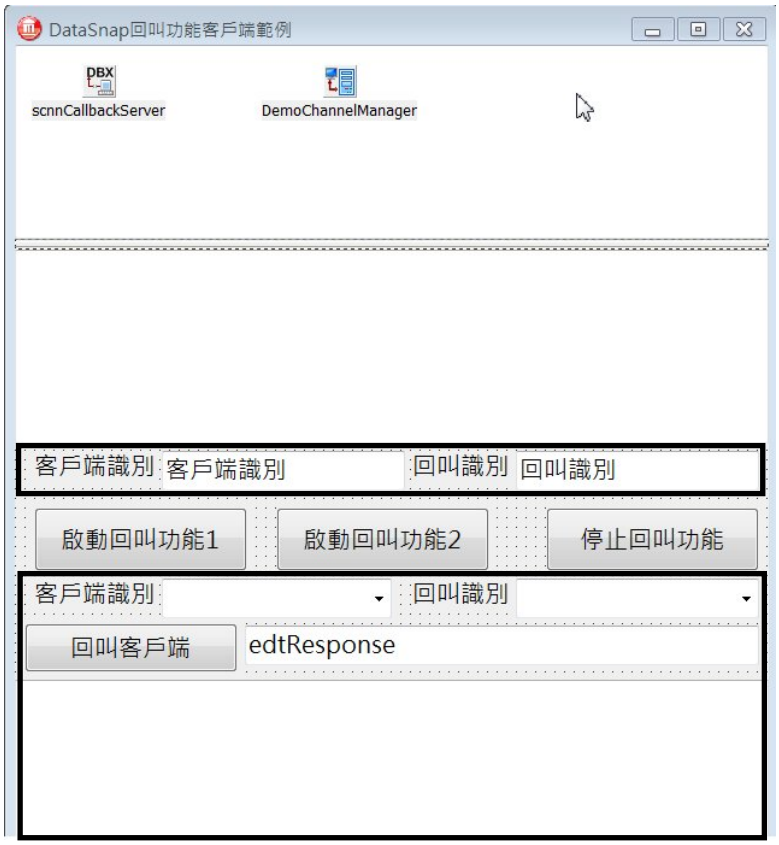


图 14 修改范例 DataSnap 客户端应用程序的主窗体

接下来就解决的问题是一个客户端应用程序如何能够回叫另外一个客户端应用程序呢？这可以使用 TDSAdminClient 类别的 NotifyCallback 方法。TDSAdminClient 是位于 DSProxy 程序单元中的类别，在 XE 版中因应强化的回叫功能而加入了数个新的方法，其中的 NotifyCallback 方法可以回叫指定回叫通道中特定客户端识别以及回叫识别的客户端，它的原型宣告如下：

```
function NotifyCallback(ChannelName: string; ClientId: string; CallbackId: string; Msg:
TJSONValue; out Response: TJSONValue): Boolean;
```

NotifyCallback 接受数个参数，下面的表格说明了每一个参数的目的：

参数名称	说明
ChannelName	回叫通道名称
ClientId	客户端识别
CallbackId	回叫识别
Msg	传递的信息
Response	被回叫客户端的回传信息

因此我们只需要为每一个客户端应用程序在注册时加入一个独特的客户端识别，如此一来我们就可以藉由 NotifyCallback 让任何两个客户端或是多个客

户端相互沟通了。那么我们如何设定客户端识别？还记得我们在客户端应用程序中使用了 TDSClientCallbackChannelManager 组件来注册客户端的回叫信息吗？TDSClientCallbackChannelManager 拥有一个 ManagerId 特性，这个 ManagerId 特性就可以做为客户端的识别，因此让我们修改客户端中『启动回叫功能』按钮的 OnClick 事件处理函式如下：

```
001   procedure TfmMainForm.btnStartClick(Sender: TObject);
002   begin
003     SetupTask;
004     AddIdsToComboBox(edtClientId.Text, edtCallbackId.Text);
005     EnableDisableButtons(False, True, True);
006     DemoChannelManager.ManagerId := edtClientId.Text;
007     DemoChannelManager.RegisterCallback(edtCallbackId.Text, TDemoCallback.Create);
008     aIdList.Add(edtCallbackId.Text);
009   end;
```

在 004 行我们把用户于 edtClientId 组件输入的客户端识别加入到 cbClientIds 中，并且把用户于 edtCallbackId 组件输入的回叫识别加入到 cbCallbackIds 中：

```
procedure TfmMainForm.AddIdsToComboBox(aClientId, aCallbackId: String);
begin
  cbClientIds.Items.Add(aClientId);
  cbCallbackIds.Items.Add(aCallbackId);
end;
```

接着在 006 行设定 TDSClientCallbackChannelManager 组件的 ManagerId 特性为用户于 edtClientId 组件输入的客户端识别，最后才于 007 行呼叫 RegisterCallback 注册这个客户端的回叫信息。一旦设定了客户端识别，其他客户端就可以使用这个客户端识别来和这个客户端进行沟通。

完成了上述的修改之后，我们就可以实作主窗体中『回叫客户端』按钮的 OnClick 事件处理函式了：

```
001   procedure TfmMainForm.btnBroadcastToClientClick(Sender: TObject);
002   var
003     LClient: TDSAdminClient;
004     LMessage: TJSONString;
005     LResponse: TJSONValue;
006     LConnection: TDBXConnection;
007   begin
```

```
008     LConnection := scnnCallbackServer.DBXConnection;
009     LClient := TDSAdminClient.Create(LConnection, False);
010     try
011       LMessage := TJSONString.Create(Format('呼叫通道: %s, 回叫识别: %s, 客户端识别: %s,
回叫讯息: %s',
012         [DemoChannelManager.ChannelName, cbCallbackIds.Text, cbClientIds.Text,
mmChannelCallbacks.Text]));
013       try
014         LClient.NotifyCallback(DemoChannelManager.ChannelName, cbClientIds.Text,
cbCallbackIds.Text, LMessage, LResponse);
015         try
016           if LResponse <> nil then
017             edtResponse.Text := Format('客户端响应: %s', [LResponse.ToString])
018           else
019             edtResponse.Text := Format('客户端响应: %s', ['nil']);
020         finally
021           LResponse.Free;
022         end;
023       finally
024         LMessage.Free;
025       end;
026     finally
027       LClient.Free;
028     end;
029   end;
```

首先我们在 009 行先建立 TDSAdminClient 对象，在 010 行建立要传递给其他客户端应用程序的以 TJSONString 封装的讯息，接着在 014 行就藉由 NotifyCallback 方法呼叫其他的客户端，请注意我们传入的信息，首先参数是回叫通道名称，这可以从 TDSClientCallbackChannelManager 组件的 ChannelName 特性值取得，第 2 个参数是客户端识别，我们传入主窗体中由用户在 TComboBox 组件 cbClientIds 的 Text 特性值，第 3 个参数是客户端回叫识别，我们传入主窗体中由用户在 TComboBox 组件 cbCallbackIds 的 Text 特性值，第 4 个参数是这个客户端要传递给其他客户端的信息，我们传入在 011 行封装的 TJSONString 对象 LMessage，最后一个参数则必须传入型态为 TJSONValue 的对象以接受其他客户端回传的执行结果。

现在请编译并且执行 2 份不同的范例客户端应用程序，如下所示:

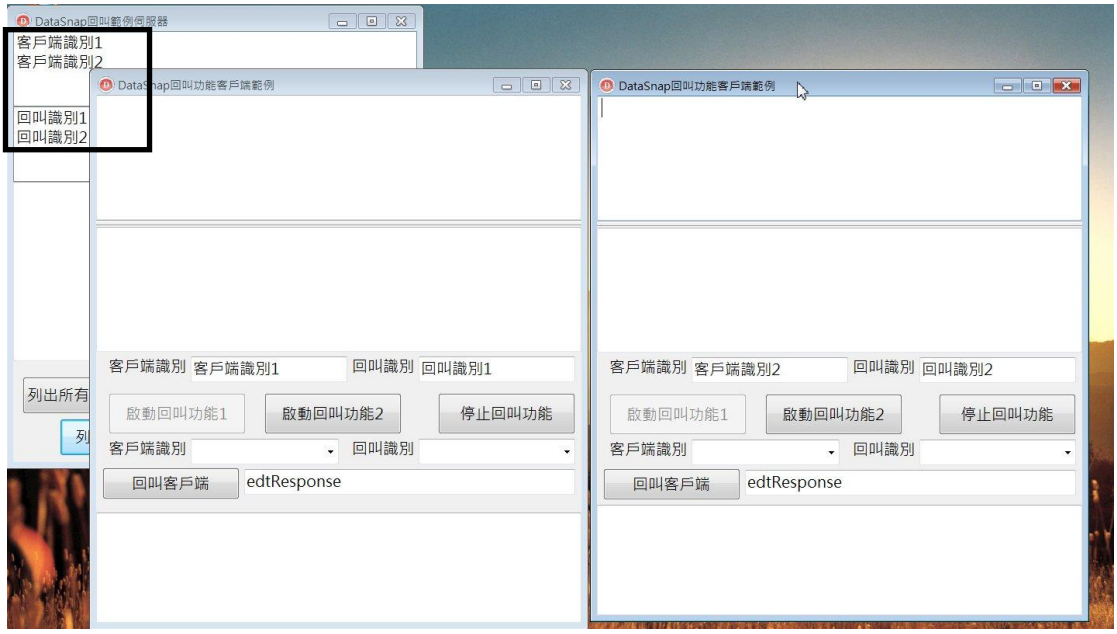


图 15 启动范例 DataSnap 服务器以及两个客户端应用程序，并且让不同的客户端应用程序注册不同的客户端识别

让我们在第一个客户端应用程序输入其客户端识别为『客户端识别 1』，其回叫识别为『回叫识别 1』，接着在第二个客户端应用程序中输入其客户端识别为『客户端识别 2』，其回叫识别为『回叫识别 2』。之后我们如果点选范例 DataSnap 服务器中来『列出所有客户端识别』按钮和『列出所有回叫识别』按钮，那么就可以如上图左上方看到，范例 DataSnap 服务器果然可在 DemoChannel 这个回叫信道中找到这些信息。

现在请在客户端应用程序 1 中的两个 TComboBox 组件输入『客户端识别 2』和『回叫识别 2』让客户端应用程序 1 回叫客户端应用程序 2。同样的请在请在客户端应用程序 2 中的两个 TComboBox 组件输入『客户端识别 1』和『回叫识别 1』让客户端应用程序 1 回叫客户端应用程序 1。

之后在客户端应用程序 1 中下方的 TMemo 组件中输入任何的信息，并且点选『回叫客户端』按钮，那么我们立刻可以在客户端应用程序 2 的上方 TMemo 组件中看到由客户端应用程序 1 中传递来的信息。同样的，如果我们在客户端应用程序 2 中下方的 TMemo 组件中输入任何的信息，并且点选『回叫客户端』按钮，那么我们立刻可以在客户端应用程序 1 的上方 TMemo 组件中看到由客户端应用程序 2 中传递来的信息，此时这两个客户端应用程序看起来如下所示：

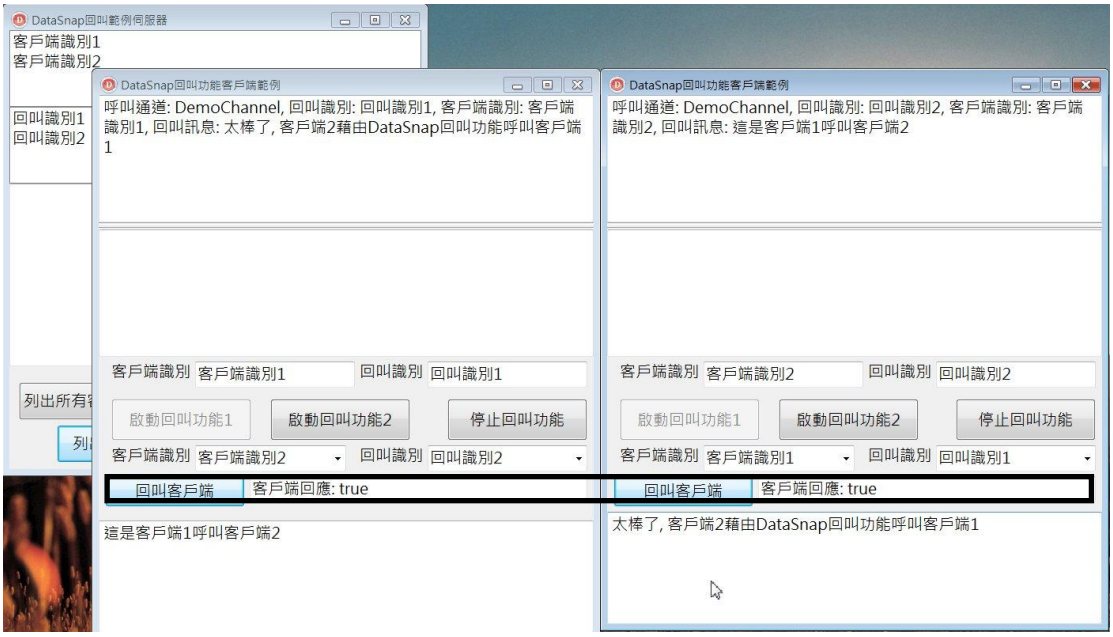


图 16 两个不同的客户端应用程序果然可藉由回叫功能相互沟通了

我们可以看到藉由 DataSnap 的回叫功能，我们果然可以让不同的客户端应用程序都藉由回叫通道来沟通并且相互传递任何的信息。此外请注意上图中两个客户端都相互回传『客户端响应:true』的讯息，这是因为我们在前面实作客户端的回叫函式时，客户端的回叫函式都回传 TJSONTrue 物件。由于 NotifyCallback 可以回传回叫客户端的执行结果，因此如果我们希望回叫函式回传其他型态的信息，那么我们可以修改两个范例客户端回叫函式如下:


```
001   function TDemoCallback.Execute(const Arg: TJSONValue): TJSONValue;
002   var
003     sDemoMessage : String;
004   begin
005   //  Result := TJSONTrue.Create;
006     Result := TJSONString.Create('成功呼叫客户端');
007
008     if (Arg is TJSONString) then
009     begin
010       sDemoMessage := TJSONString(Arg).Value;
011       TThread.Synchronize(nil,
012                           procedure
013                           begin
014                             fmMainForm.mmDemoMessage.Lines.Text := sDemoMessage;
015                           end
016                           );
```

```
017     end;
018   end;
019
020   { TDemoCallback2 }
021
022   constructor TDemoCallback2.Create;
023   begin
024
025   end;
026
027   function TDemoCallback2.Execute(const Arg: TJSONValue): TJSONValue;
028   var
029     sDemoMessage : String;
030   begin
031   //  Result := TJSONTrue.Create;
032     Result := TJSONString.Create('成功呼叫客户端');
033
034     if (Arg is TJSONString) then
035     begin
036       sDemoMessage := TJSONString(Arg).Value;
037       TThread.Synchronize(nil,
038                           procedure
039                           begin
040                             fmMainForm.mmDemoMessage2.Lines.Text := sDemoMessage;
041                           end
042                           );
043     end;
044 end;
```

在上面的两个回叫函式中我们修改回传 TJSONString 封装的信息而不是单纯的 TJSONTrue 对象，这是果然是因为 NotifyCallback 回传的型态是 TJSONValue，因此任何从 TJSONValue 类别继承下来的类别对象都可以做为回叫函式的执行回传结果。

现在如果我们再次执行两个客户端应用程序，那么就可以看到类似如下的结果:

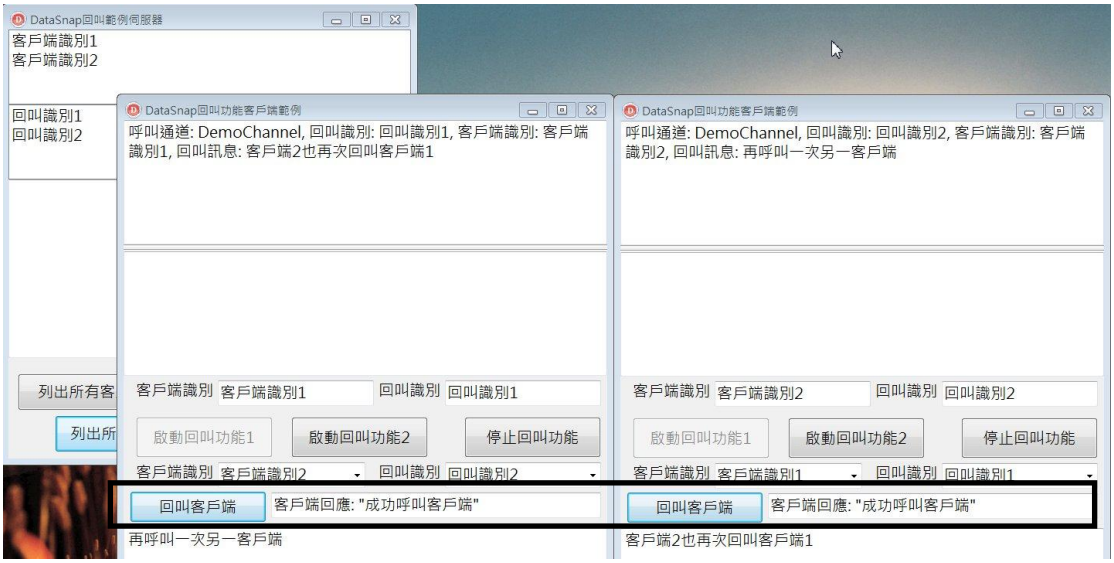


图 17 不同客户端的回叫函式可传递执行结果回其他客户端应用程序

从上图中我们可以看到不同的客户端果然可以藉由回叫函式传递任何的信息回其他的客户端应用程序。

现在您应该已经充分的了解了 DataSnap 的回叫功能了，也许您可以自己试着完成一个小功课，请继续修改范例 DataSnap 服务器和客户端应用程序让客户端应用程序可以注册不同的回叫通道而不像前面的范例应用程序只使用固定的 DemoChannel 这个回叫通道，如何？应该不算太难而且应该很有趣吧。

开发轻薄型回叫 DataSnap 客户端

DataSnap 10.3 除了可以让 Delphi/C++Builder 的客户端参与回叫机制之外，也提供了支持 JavaScript 客户端的能力，这意谓任何支持 JavaScript 的框架，开发工具或是程序语言都可以参与 DataSnap 的回叫机制，例如 Java 的客户端，.NET 的客户端或是 Ruby/PHP 等的客户端，这让 DataSnap XE 的回叫机制变得非常的实用。在前面的章节中我们已经细节的说明了如何使用 DataSnap 的回叫机制以及开发了数个 Delphi 的范例，在本小节中我们将进一步说明如何使用 JavaScript 来参与 DataSnap 10.3 的回叫机制。

首先让我们建立一个支持 HTTP/HTTPS 的 DataSnap 服务器:

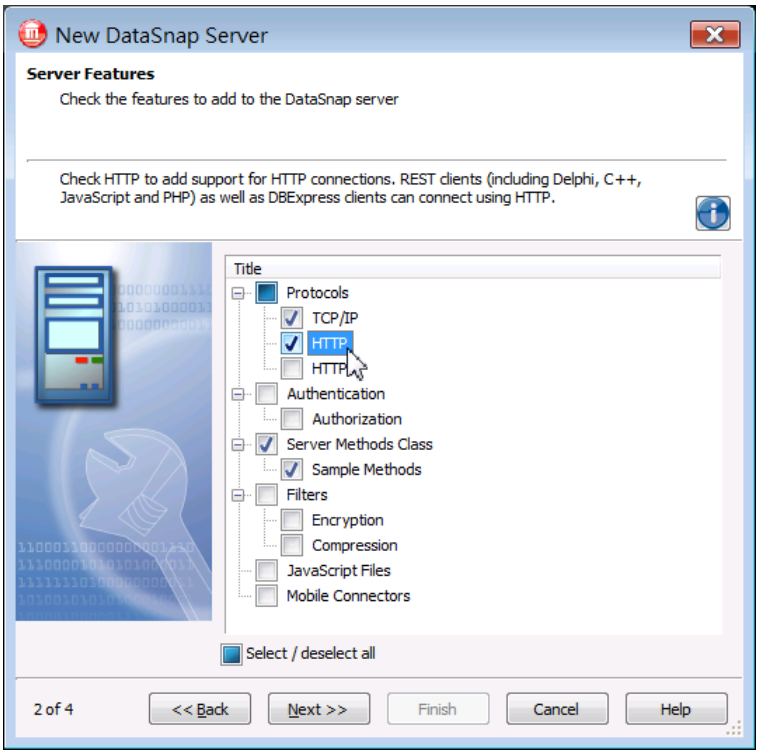


图 18 建立支持 HTTP/HTTPS 和 TCP 的 DataSnap 服务器

接着在 ServerContainer 中放入如下的组件:

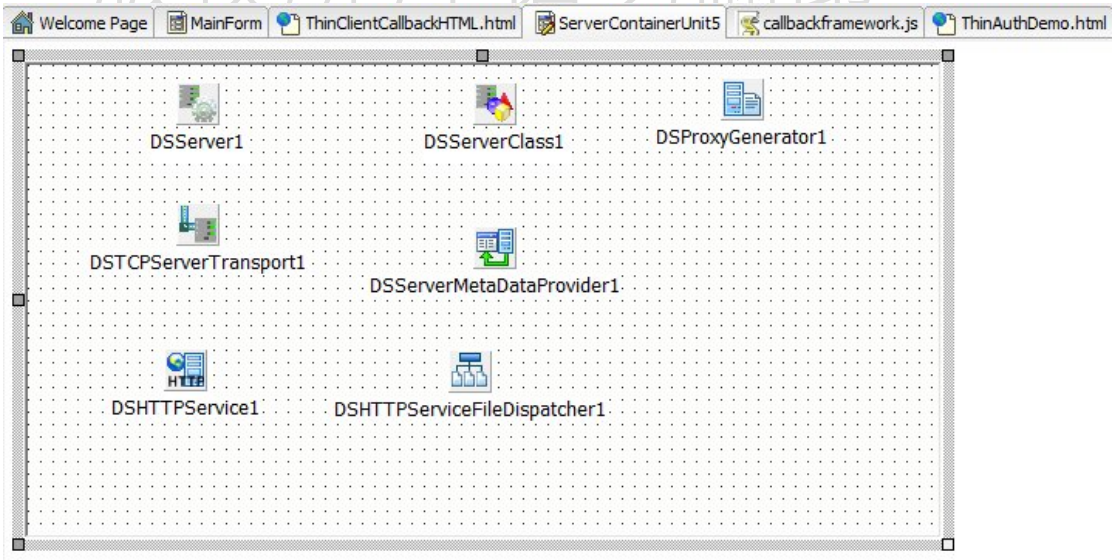


图 19 在服务器的 ServerContainer 加入相关的组件以支持浏览器的呼叫

并且设定每一个组件的特性值如下:

TDSServerMetaDataProvider 组件:

特性	特性值
Server	DSServer1

TDSHTTPServiceFileDispatcher 组件:

特性	特性值
Service	DSHTTPService1
RootDirectory	E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\

TDSHTTPServiceFileDispatcher 组件的 RootDirectory 特性是指定稍后使用 JavaScript 参与回叫的 HTML 档案的目录所在地。

TDSProxyGenerator 组件:

特性	特性值
MetaDataProvider	DSServerMetaDataProvider1
TargetDirectory	E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\
TargetUnitName	serverfunctions.js
Writer	Java Script REST

TDSProxyGenerator 组件的 TargetDirectory 是指稍后由 DataSnap 自动产生的 JavaScript 档案的储存目录，这个自动产生的 JavaScript 档案就是 TargetUnitName 特性值，这也就是说 TDSProxyGenerator 会在目录 E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\ 中产生一个名为 serverfunctions.js 的档案，serverfunctions.js 中的 JavaScript 程序代码封装了 DataSnap 服务器提供的服务。


```
procedure TForm18.btnListAllClientIdsClick(Sender: TObject);
var
  aIdList : TList<String>;
  sId : String;
begin
  aIdList := ServerContainer5.DSServer1.GetAllChannelClientId(DEMOChannel);
  try
    for sId in aIdList do
      lbAllClientIds.Items.Add(sId);
  finally
    aIdList.Free;
  end;
end;


procedure TForm18.mmCallbackMessagesChange(Sender: TObject);
var
```

```
  vMessage : TJSONString;
begin
  vMessage := TJSONString.Create(mmCallbackMessages.Lines.Text);
  ServerContainer5.DSServer1.BroadcastMessage(DEMOChannel, vMessage);
end;
```

为了让 TDSProxyGenerator 组件在稍后范例 HTML 档案被存取时能够自动产生 serverfunctions.js，因此我们需要在 TDSHTTPServiceFileDispatcher 组件的 BeforeDispatch 事件中撰写如下的程序代码。下面的程序代码是当目前客户端的 HTML 档案在浏览器中被存取时，检查的目录中是否已经包含了 serverfunctions.js，如果没有的话就呼叫 TDSProxyGenerator 组件的 Write 方法自动产生 serverfunctions.js。

```
procedure TServerContainer5.DSHTTPServiceFileDispatcher1BeforeDispatch(
  Sender: TObject; const AFileName: string; AContext: TDSHTTPContext;
  Request: TDSHTTPRequest; Response: TDSHTTPResponse; var Handled: Boolean);
begin
  if (SameFileName(ExtractFileName(AFileName), DSProxyGenerator1.TargetUnitName) and not
FileExists(AFileName)) then
    DSProxyGenerator1.Write;
end;
```

例如下图就是稍后在浏览器中执行范例 HTML 档案 ThinClientCallbackHTML.html 后便会在范例目录 E:\QComm\Book2\Demos\DataSnapChapter03\ThinClientCallbackDemos\Client\中产生为 serverfunctions.js:

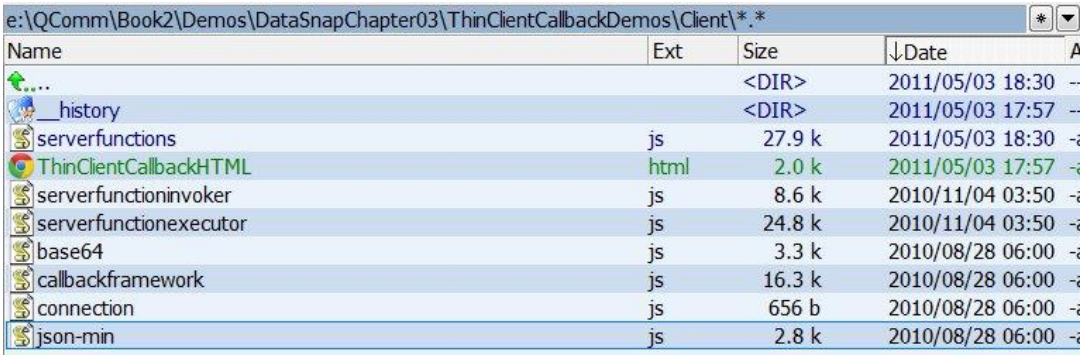


图 20 范例目录中的档案以及由范例服务器自动产生的 serverfunctions.js

现在于 Delphi 整合发展环境中建立一个新的 HTML 档案，然后设计如下的图形用户接口:

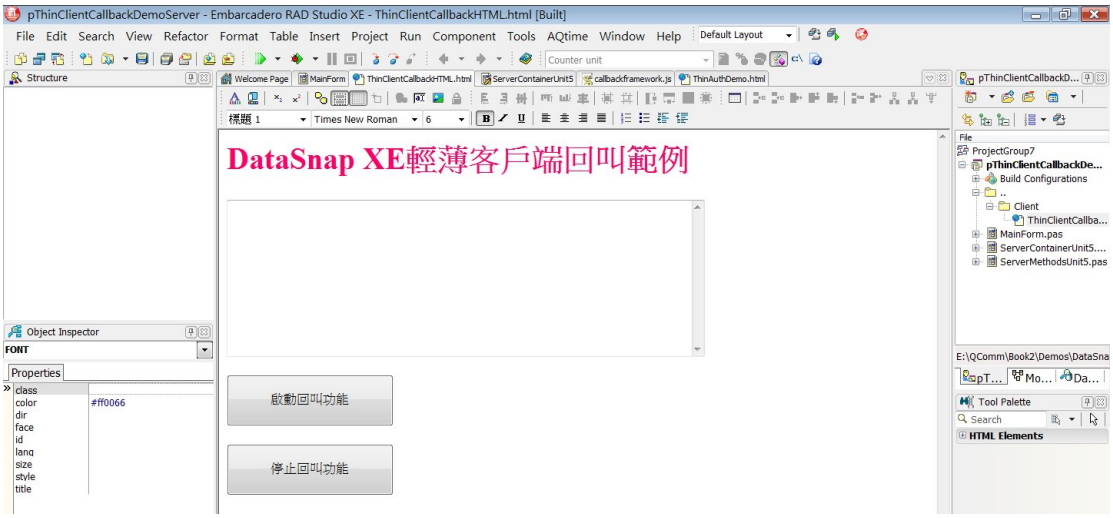


图 21 使用 Delphi IDE 建立范例 HTML 以准备在浏览器中使用回叫功能

接着在 HTML 档案的程序代码中撰写如下的程序代码:

```
001    <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
002    <html>
003      <head>
004        <title>DataSnap XE 轻薄客户端认证范例
005    </title>
006    <meta HTTP-EQUIV="Content-Type" CONTENT="text/html; CHARSET=UTF-8"><meta
http-equiv="cache-control" content="no-cache">
007        <script type="text/javascript" src="base64.js"></script>
008        <script type="text/javascript" src="json-min.js"></script>
009        <script type="text/javascript" src="serverfunctionexecutor.js"></script>
010        <script type="text/javascript" src="ServerFunctions.js"></script>
011        <script type="text/javascript" src="connection.js"></script>
012        <script type="text/javascript" src="callbackframework.js"></script>
013        <script type="text/javascript">
014
015        function startCallback()
016        {
017          var connectionInfo = {"host":"localhost","port":"8089"};
018          var channel = new ClientChannel("jsc1", "DemoChannel", connectionInfo);
019          var callback = new ClientCallback(channel, "uid2", function(jsonValue)
020                                {
021                                  if (jsonValue != null && jsonValue.created == null)
022                                  {
023                                    document.getElementById("mmCallback").value = jsonValue;
```

```
024                          }
025                          return true;
026                        });
027        channel.connect(callback);
028       }
029
030       function stopCallback()
031       {
032         if (channel != null)
033         {
034           channel.disconnect();
035         }
036
037
038       }
039
040       </script>
041     </head>
042     <body>
043     <div>
044        <h1><font color="#ff0066">DataSnap XE 轻薄客户端回叫范例
045   </font></h1>
046        <form onsubmit="startCallback(); return false;">
047          <textarea rows="10" cols="60" id="mmCallback"></textarea><br><br />
048          <input id="btncallServer" type="submit" value="启动回叫功能" style="WIDTH:
217px; HEIGHT: 67px" size="34" onclick="StartCallback" />
049        </form>
050        <form onsubmit="stopCallback(); return false;">
051          <input id="btncallServer" type="submit" value="停止回叫功能" style="WIDTH:
217px; HEIGHT: 67px" size="34" onclick="StopCallback" />
052        </form>
053     </div>
054     </body>
055   </html>
```

在上面的程序代码中，当使用者在浏览器中点选『启动回叫功能』按钮后就会呼叫 startCallback 函式，startCallback 首先在 017 行建立 connectionInfo 信息设定呼叫的服务器地址以及 DataSnap 服务器使用的通信埠。018 行建立 ClientChannel 对象并且传入前面小节已经说明的参数，019 行建立

ClientCallback 对象并且注册客户端 JavaScript 的回叫函式，这个客户端回叫函式接受 DataSnap 服务器输入的讯息并且同步显示在客户端的浏览器之中，就如同前面的 4-2 节的范例一样，只是在这里我们是使用 JavaScript 和浏览器做为回叫的客户端。

现在启动浏览器并且加载范例 HTML 档案 ThinClientCallbackHTML.html，接着在 DataSnap 服务器中点选『列出所有客户端识别』按钮的话，就可以在 DataSnap 服务器的主窗体中看到客户端浏览器使用 JavaScript 注册的通道 ID『jsc1』，此时如果在 DataSnap 服务器的主窗体的 TMemo 组件中讯息的话，就可以立刻在浏览器中看到这些讯息，因为 DataSnap 服务器会藉由回叫机制立刻通知客户端，如下图所示:

图 22 范例 DataSnap 服务器果然可藉由回叫机制回叫在浏览器中执行的 JavaScript 程序代码

如何 DataSnap 10.3 是回叫机制是不是很强大? 藉由 JavaScript 开发人员甚至可以撰写手机的客户端应用程序并且让 DataSnap 服务器藉由回叫机制通知客户端任何的信息，Cool!

4-3 结论

DataSnap 10.3 在原有的回叫机制下持续的增加功能，DataSnap 10.3 的回叫功能供支持了信道和 JavaScript 客户端的能力，除了 Delphi/C++Builder 的客户端之外，也允许所有支持 JavaScript 的客户端都能够参与和使用 DataSnap 的回叫机制，让 DataSnap 的回叫机制可提供跨平台从而能够建立更为复杂，实际和先进的回叫架构。

第5章 使用DataSnap过滤器

读者现在应该了解DataSnap的JSON是是使用字符串型态来传递数据的，因此所有非字符串型态的数据都必须转换为字符串的型态，虽然如此一来在处理上比较简单，但这也造成了其他的问题，例如一些敏感性的数据如果使用字符串型态来传递的话就不太实际。DataSnap 从 2010 开始为了解决这种问题因此加入了过滤器的机制，让开发人员在传递特殊的资料时可以藉由过滤器来进行额外的处理，例如在传递资料出去时先加密，并且在接受到数据之后再进行解密，在 DataSnap 2010 即提供了压缩和解压缩的过滤器，到了 DataSnap 2011 又增加了内建的加密/解密的过滤器，本章的目的即在于讨论如何使用 DataSnap 的过滤器。

使用 DataSnap 过滤器非常的简单，Delphi 2010 开始即内建了一个压缩过滤器，可以有效的压缩使用 TCP/IP 通讯协议的资料传递。让我们先说明如何使用这个内建的过滤器，稍后我们再深入的说明如何开发客制化过滤器。

5-1 使用内建的过滤器

DataSnap 10.3 版一共提供了三个过滤器可供开发人员使用，如果开发人员仍然觉得不够或是有特殊的需求，那么也可以撰写客制化过滤器来使用。由于使用 DataSnap 过滤器非常的简单，因此本节就以一个范例来说明如何使用 DataSnap 的压缩/解压缩过滤器。

在下面的范例中我们将使用一个包含图形的数据表来展示使用压缩/解压缩过滤器的效果。

5-1-1 建立 DataSnap 过滤器服务器

首先在 Delphi 10.3 中建立一个 DataSnap Server 项目，先开启项目中的 ServerMethodUnit 程序单元，然后从 Data Explorer 页次中拖曳范例数据表 BIOLIFE 到 ServerMethodUnit 程序单元中，IDE 便会自动产生

TSQLConnection 和 TSQLDataSet 组件链接到范例数据表 BIOLIFE，接着放入 TDataSetProvider 组件链接到程序单元中的 TSQLDataSet，如下所示：

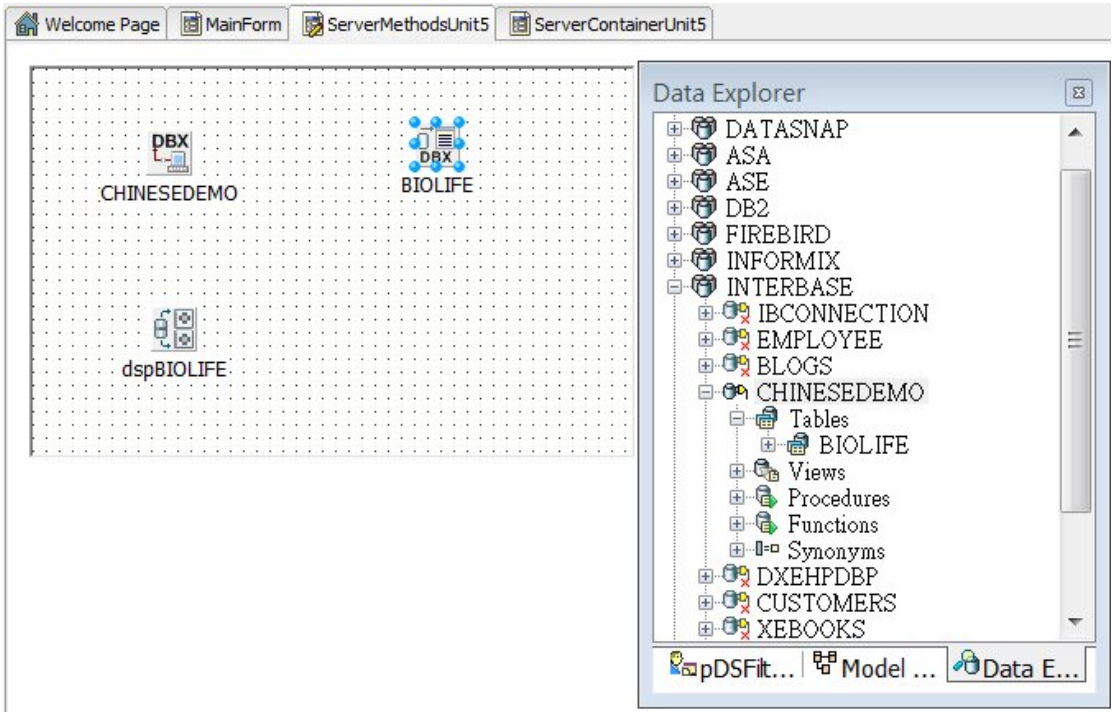


图 1 在 DataSnap 服务器的 ServerMethodUnit 中加入 dbExpress 组件链接到范例数据表 BIOLIFE

接着开启项目中的 ServerContainerUnit 程序单元，点选其中的 TDSTCPServerTransport 组件，接着在对象查看器中点选它的 Filters 特性，在显示的特性值编辑器中点选上方的 Add New 按钮加入一个新的过滤器，点选特性值编辑器中新加入的 TTransportFilterItem 过滤器，再次点选对象查看器，选择 FilterId 特性就可以从下拉盒中看到 DataSnap 提供了 PC1，RSA 以及 ZlibCompression 三个过滤器可供使用，如下图所示：

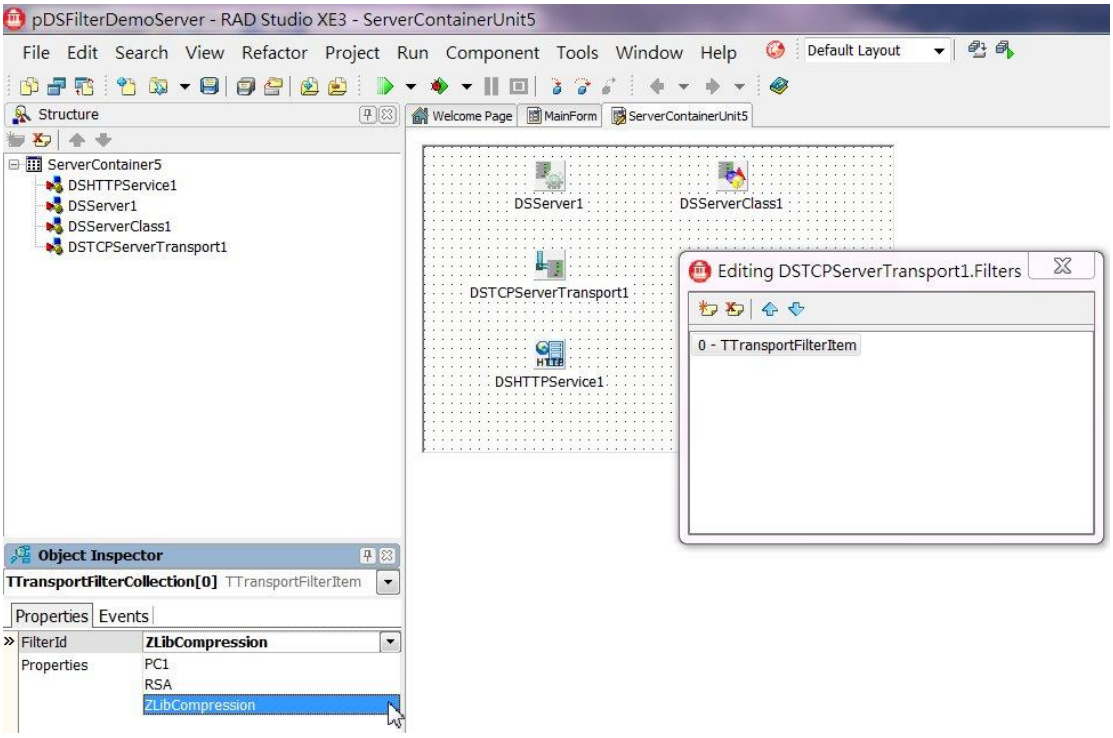


图 2 在 ServerContainerUnit 程序单元中点选 TDSTCPServerTransport 组件加入使用过滤器

在这里让我们选择使用 ZlibCompression 过滤器，如此一来就完成了使用 DataSnap 过滤器的步骤了，ZlibCompression 过滤器就是 DataSnap 内建的压缩过滤器，在加入了 ZlibCompression 过滤器之后，编译并且执行范例 DataSnap 服务器，现在 DataSnap 服务器就自动提供了压缩 JSON 数据的能力。

5-1-2 建立使用 DataSnap 过滤器的客户端应用程序

在项目群组中建立一个 VCL Form 应用程序项目，建立一个 DataSnap Client Module，连结到上一小节的范例 DataSnap 服务器，IDE 便会在 DataSnap Client Module 中产生 TSQLConnection 组件，接着在其中放入 TDSProviderConnection 和 TClientDataSet 组件，如下图所示:

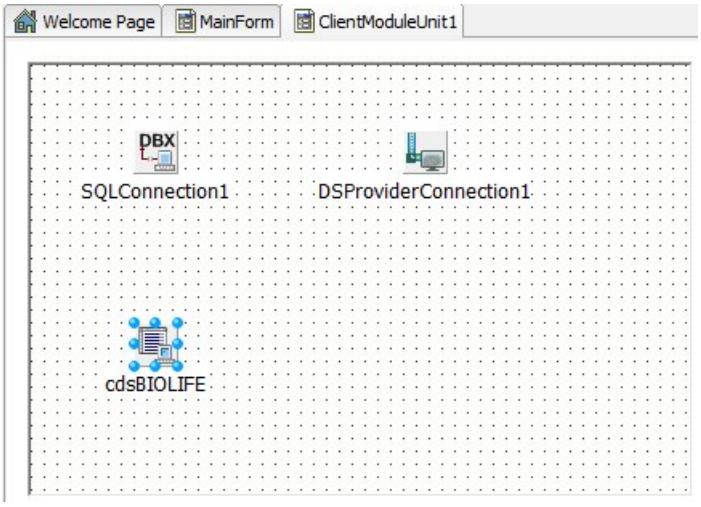


图 3 在 ClientModuleUnit 程序单元中加入 TDSProviderConnection 和 TClientDataSet 组件

设定 TDSProviderConnection 组件的特性值如下:

特性	特性值
SQLConnection	SQLConnection1
ServerClassName	TServerMethods5
Name	DSProviderConnection1

设定 TDSProviderConnection 组件的特性值如下:

特性	特性值
RemoteServer	DSProviderConnection1
ProviderName	DspBIOLIFE
Name	cdsBIOLIFE

开启主窗体，在主窗体中加入下面的组件并且链接 DataSnap Client Module 中的 cdsBIOLIFE 以准备显示范例数据表中的数据。

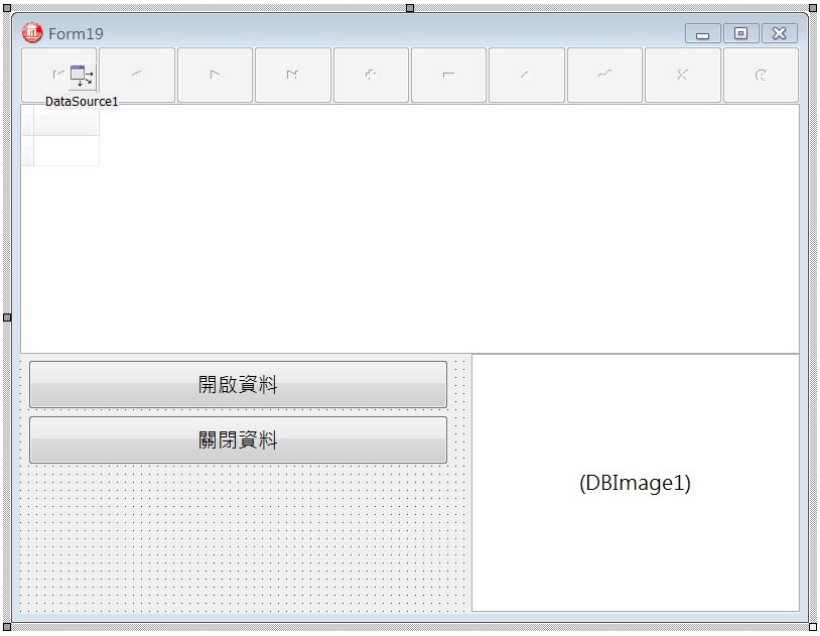


图 4 范例客户端应用程序的主窗体

接着开启主窗体的源代码，因为我们要在客户端应用程序中加入解压缩数据的能力，这非常的简单，我们只要在客户端应用程序的主窗体中加入使用 DBXCompressionFilter 程序单元即可，例如下面就是客户端应用程序加入 DBXCompressionFilter 程序单元的程序代码:


```
implementation


uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,Dialogs, Grids,
DBGrids, ExtCtrls, DBCtrls, DB, DbxCompressionFilter, StdCtrls;
```

现在编译并且执行客户端应用程序，并且让我们使用 TCP Viewer 来观察使用压缩过滤器之前的情形以及使用压缩过滤器之后的效果。

下图是 TCP Viewer 显示范例 DataSnap 应用系统使用压缩过滤器之前的情形，从下图中我们可以看到在 DataSnap 服务器和客户端应用程序之间传递的数据当然是使用字符串的型态，所有传递的数据都一清二楚，同时请读者注意下图右边显示了从服务器传递到客户端的数据量(1093368 字节)以及从客户端传递到伺服端的数据量(2326 字节)，由于范例数据表中包含了图形的数据，因此图形数据在传递时必须转换为 Base64 的字符串型态的数据，所以造成了伺服端和客户端之间必须传递大量的数据。

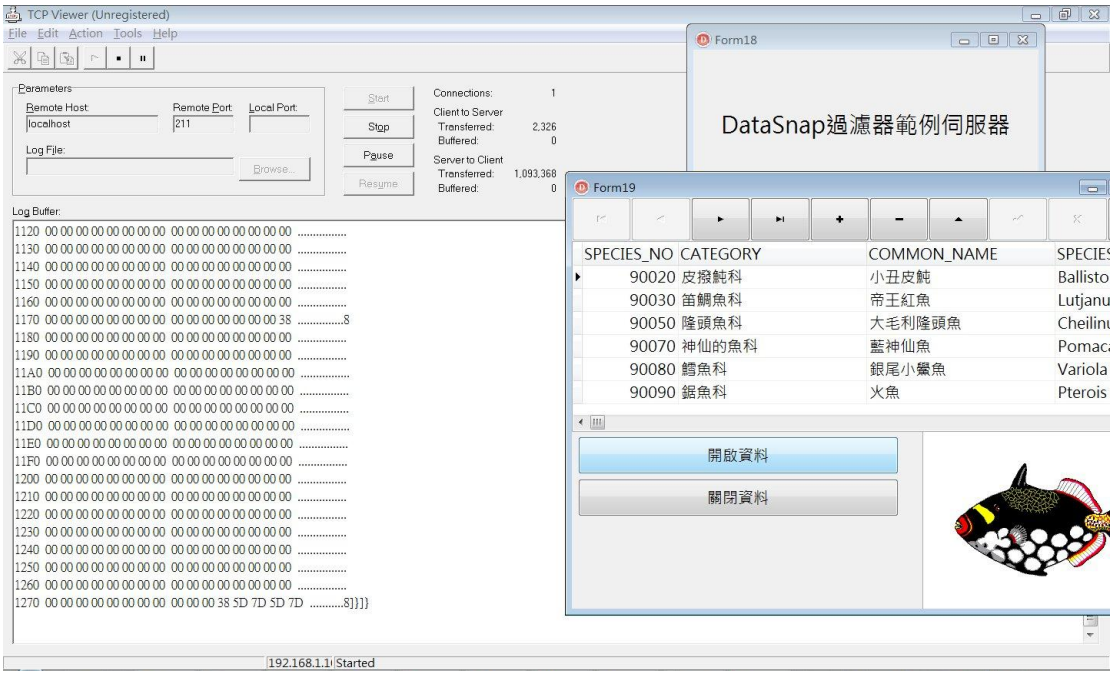


图 5 不使用 DataSnap 压缩过滤器传递包含图形数据表的结果

而下图则是使用压缩过滤器之后的效果：

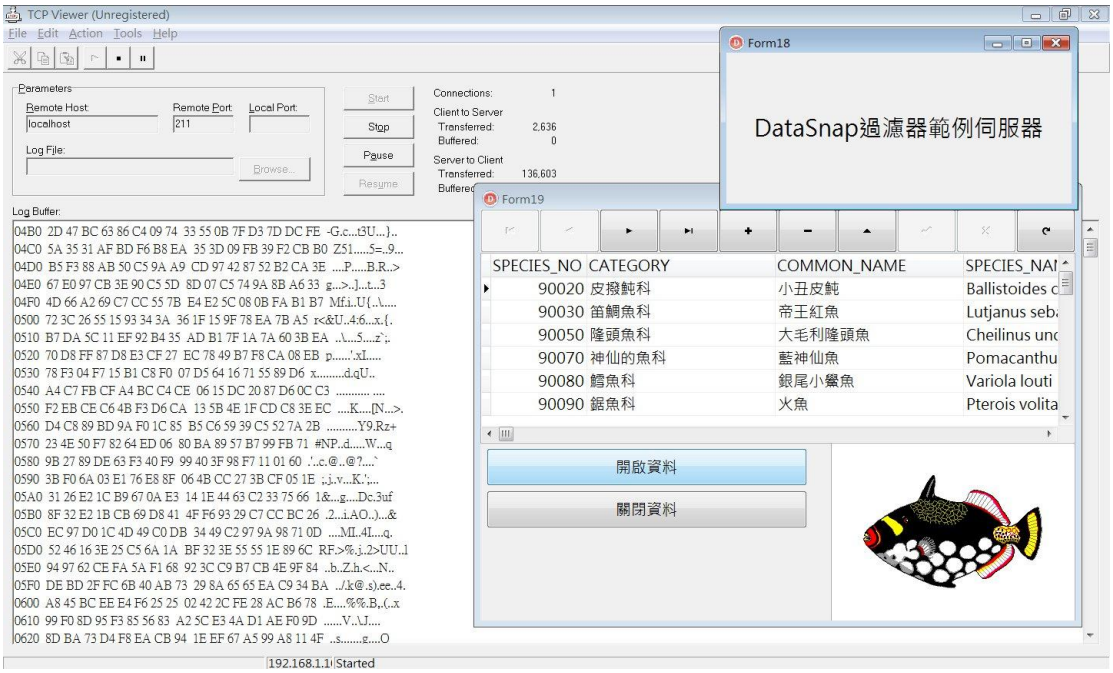


图 6 使用 DataSnap 压缩过滤器传递包含图形数据表的结果

我们可以很明显的看到使用了压缩过滤器之后，从服务器传递到客户端的数据量减少到 136603 字节，可见到压缩过滤器非常有效的减少了服务器和客户端之间的数据传递量，这不但可以增加分布式应用程序的执行速度，也可以增加支持的客户端的数量。

如何？使用过滤器是不是又简单，又有明显的效果？不过 DataSnap 2010 只提供了一个内建的过滤器实在太少，到了 DataSnap 10.3 虽然增加到了三个，但开发人员仍然可能需要使用客制化的过滤器，好在 DataSnap 过滤器架构在设计时就考虑到了允许让开发人员能够自行开发过滤器并且内嵌到 DataSnap 之中，下一小节将讨论如何开发客制化过滤器并且使用在 DataSnap 的分布式应用系统中。

5-2 开发客制化过滤器

要开发客制化过滤器，开发人员必须从 TTransportFilter 类别衍生子代类别并且实作 TTransportFilter 类别中相关的虚拟方法，下面的窗体说明了开发人员需要实作的虚拟方法:

函式名称	回传型态	说明
GetParameters	TDBXStringArray	回传所有的参数
GetUserParameters	TDBXStringArray	回传使用者可改变的参数
ProcessInput	TBytes	使用客制化程序代码正向处理传递的数据流
ProcessOutput	TBytes	使用客制化程序代码反向处理传递的数据流
Id	UnicodeString	过滤器的 ID
GetParameterValue	UnicodeString	取得特定名称的参数值
SetParameterValue	Boolean	设定特定名称的参数值

了解了需要实作那些虚拟方法之后，我们就可以开始动手开发一个客制化过滤器了。在本文中笔者将撰写一个非常简单的加密/解密过滤器，其实这个加密/解密过滤器只是在传递数据时和一个字符串进行 xor 的动作，到了另一端再次 xor 相同的字符串而已，当然这个范例加密/解密过滤器只是为了说明如何开发客制化过滤器，如果读者需要加密/解密的功能的话，请直接使用 DataSnap 10.3 中提供的 PC1 或是 RSA 过滤器。

首先让我们宣告范例 TTransportEncryptFilter 类别从 TTransportFilter 类别继承下来并且复载相关必要的虚拟方法:

```
TTransportEncryptFilter = class(TTransportFilter)
 private
   FEncrypt: TSimpleEncryptor;
   FParameters: TDictionary<String, String>;
 protected
   function GetParameters: TDBXStringArray; override;
```

```
  function GetUserParameters: TDBXStringArray; override;

public
  function GetParameterValue(const ParamName: UnicodeString): UnicodeString;
    override;
  function SetParameterValue(const ParamName: UnicodeString;
    const ParamValue: UnicodeString): Boolean; override;
  constructor Create; override;
  destructor Destroy; override;
  function ProcessInput(const Data: TBytes): TBytes; override;
  function ProcessOutput(const Data: TBytes): TBytes; override;
  function Id: UnicodeString; override;
end;
```

TTransportEncryptFilter 类别将使用 TSimpleEncryptor 进行字符串 xor 的运算，在 TSimpleEncryptor 类别中实作了两个方法，Encrypt 和 Decrypt，其实这两个方法的实作程序代码是一样的，只是为了说明方便分别实作成 Encrypt 和 Decrypt 以便让读者了解。Encrypt 和 Decrypt 接受 TBytes 型态的参数，这个参数在 Encrypt 方法中是代表传递出去的数据，Encrypt 方法使用程序代码加密之后再把加密过的数据以 TBytes 型态回传。

而 Decrypt 的参数则是代表接受来的数据，Decrypt 方法必须使用程序代码加以解密以还原数据。

```
TSimpleEncryptor = class
protected

public
  function Encrypt(const Data: TBytes): TBytes;
  function Decrypt(const Data: TBytes): TBytes;
  constructor Create;
end;
```

下面是这两个方法的实作程序代码，读者可以看到这两个方法的实作程序代码是 样的，它们都接受的参数以'DexterHighlanderTiburonWeaver'这个键值字符串进行 xor 的运算:

```
const
 EncryptKey = 'DexterHighlanderTiburonWeaver';

constructor TSimpleEncryptor.Create;
```

```
begin
  inherited Create;
end;

function TSimpleEncryptor.Decrypt(const Data: TBytes): TBytes;
var
  i: Integer;
  idx: Integer;

begin
  Result := Data;
  idx := 0;
  for i := 0 to Length(Data) - 1 do
  begin
    Result[i] := Byte(Chr(Ord(Data[i]) xor Ord(EncryptKey[idx])));
    Inc(idx);
    if (idx > Length(EncryptKey)) then
      idx := 0;
  end;
end;

function TSimpleEncryptor.Encrypt(const Data: TBytes): TBytes;
var
  i: Integer;
  idx: Integer;

begin
  Result := Data;
  idx := 0;
  for i := 0 to Length(Data) - 1 do
  begin
    Result[i] := Byte(Chr(Ord(Data[i]) xor Ord(EncryptKey[idx])));
    Inc(idx);
    if (idx > Length(EncryptKey)) then
      idx := 0;
  end;
end;
```

下面则是 TTransportEncryptFilter 类别的实作程序代码，读者可以看到在在建构函式中建立了 TSimpleEncryptor 对象，并且分别在 ProcessInput 虚拟方法中呼叫 TSimpleEncryptor 对象的 Encrypt 方法加密传递的数据并且在 ProcessOutput 虚拟方法中呼叫 TSimpleEncryptor 对象的 Decrypt 方法以解密数据:

```
function TTransportEncryptFilter.GetUserParameters: TDBXStringArray;
begin
  SetLength(Result, 1);
  Result[0] := EncryptKey;
end;

function TTransportEncryptFilter.GetParameters: TDBXStringArray;
begin
  SetLength(Result, 1);
  Result[0] := EncryptKey;
end;

function TTransportEncryptFilter.GetParameterValue
  (const ParamName: UnicodeString): UnicodeString;
begin
  FParameters.TryGetValue(ParamName, Result);

  if ( ParamName = EncryptKey ) and ( Result = '' ) then
     Result := '0';
end;

function TTransportEncryptFilter.SetParameterValue
  (const ParamName, ParamValue: UnicodeString): Boolean;
begin
  FParameters.AddOrSetValue(ParamName, ParamValue);
  Result := True;
end;



constructor TTransportEncryptFilter.Create;
begin
  inherited Create;
```

```
  FParameters := TDictionary<String, String>.Create;
  FEncrypt := TSimpleEncryptor.Create;
end;

destructor TTransportEncryptFilter.Destroy;
begin
  FreeAndNil(FParameters);
  FreeAndNil(FEncrypt);
  inherited Destroy;
end;

function TTransportEncryptFilter.ProcessInput(const Data: TBytes): TBytes;
begin
  OutputDebugString(PWideChar('Encrypted - ' + Stringof(Data)));
  Result := FEncrypt.Encrypt(Data);
end;

function TTransportEncryptFilter.ProcessOutput(const Data: TBytes): TBytes;
begin
  OutputDebugString(PWideChar('Decrypted - ' + Stringof(Data)));
  Result := FEncrypt.Decrypt(Data);
end;

function TTransportEncryptFilter.Id: UnicodeString;
begin
  Result := EncryptFilterName;
end;
```

最后我们需要在 **initialization** 部份注册这个客制化过滤器并且在 **finalization** 部份解除注册客制化过滤器:

```
initialization

TTransportFilterFactory.RegisterFilter(EncryptFilterName,
  TTransportEncryptFilter);

finalization
```

```
TTransportFilterFactory.UnregisterFilter(EncryptFilterName);
```

使用客制化过滤器

OK，回到范例服务器，开启 ServerContainer 程序单元并且在它的 OnCreate 事件处理函式中使用 TDSTCPServerTransport 的 AddFilter 方法加入我们的客制化过滤器:

```
procedure TServerContainer1.DataModuleCreate(Sender: TObject);
var
  i : Integer;
begin
  i := DSTCPServerTransport1.Filters.AddFilter(uEncryptFilter.EncryptFilterName);
  for i := 0 to DSTCPServerTransport1.Filters.Count - 1 do
    Form1.lbFilters.Items.Add(DSTCPServerTransport1.Filters.GetFilter(i).Id);
end;
```

当然我们也需要在 ServerContainer 程序单元的 uses 句子中加入包含客制化过滤器的程序单元 uEncryptFilter:

```
implementation


uses Windows, ServerMethodsUnit1, MainForm, uEncryptFilter;
```

现在执行 DataSnap 服务器，我们就可以看到服务器显示它已经找到了我们的客制化过滤器:

图 7 范例客制化 DataSnap 过滤器服务器

接着开启客户端应用程序，在主窗体中也加入客制化过滤器的程序单元 uEncryptFilter，编译并且执行客户端应用程序，再使用 TCP Viewer 观察传递的数据，我们果然看到数据现在都经过加密了：

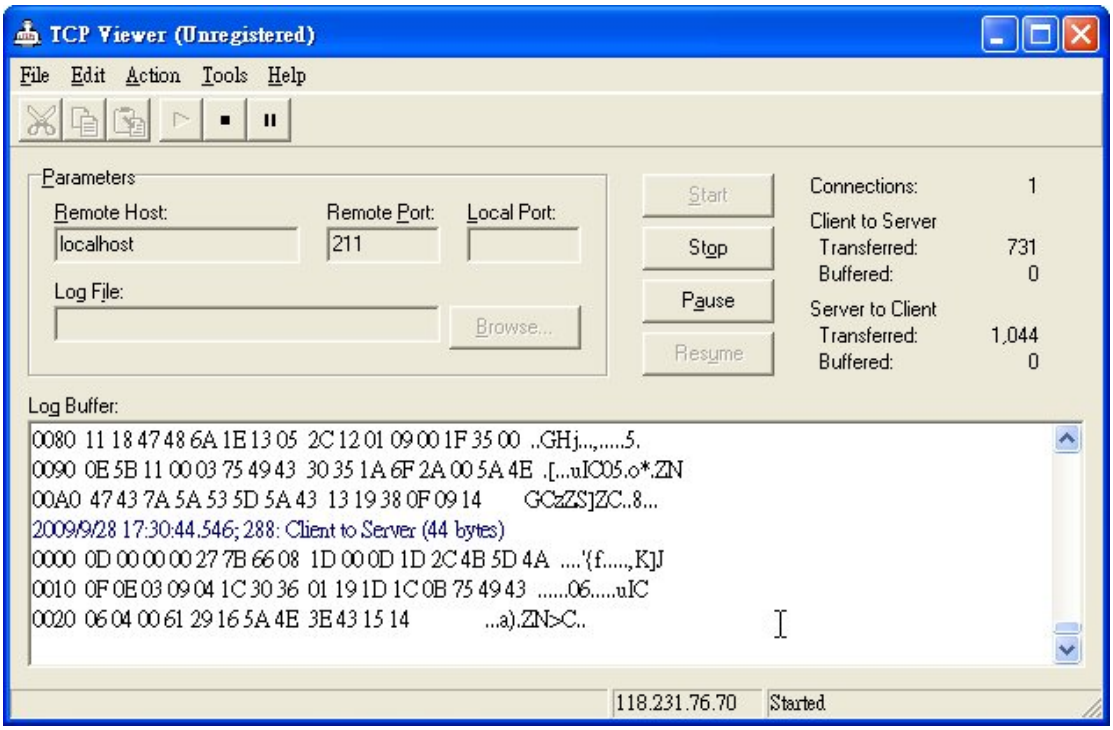


图 8 使用 TCP Viewer 观察的结果

我们可以看到传递的数据量增加了，但是客户端仍然可以正确的接受到数据：

图 9 范例客户端应用程序

当然我们也可以同时使用两个过滤器，享受加密又压缩的好处，下图是服务器同时支持了两个过滤器：

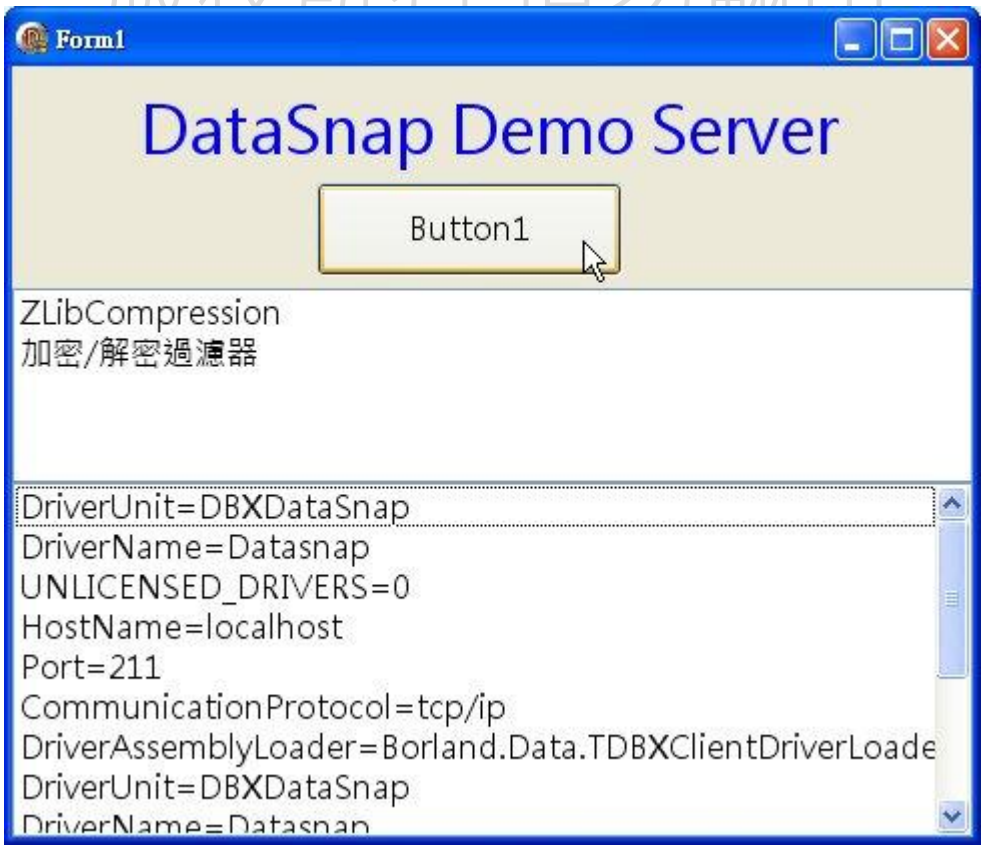


图 10 范例客制化 DataSnap 过滤器服务器

如果我们再次使用 TCP Viewer，就可以看到下图，享受加密又压缩的好处，因为数据加密了而且数据传递量又减少了。

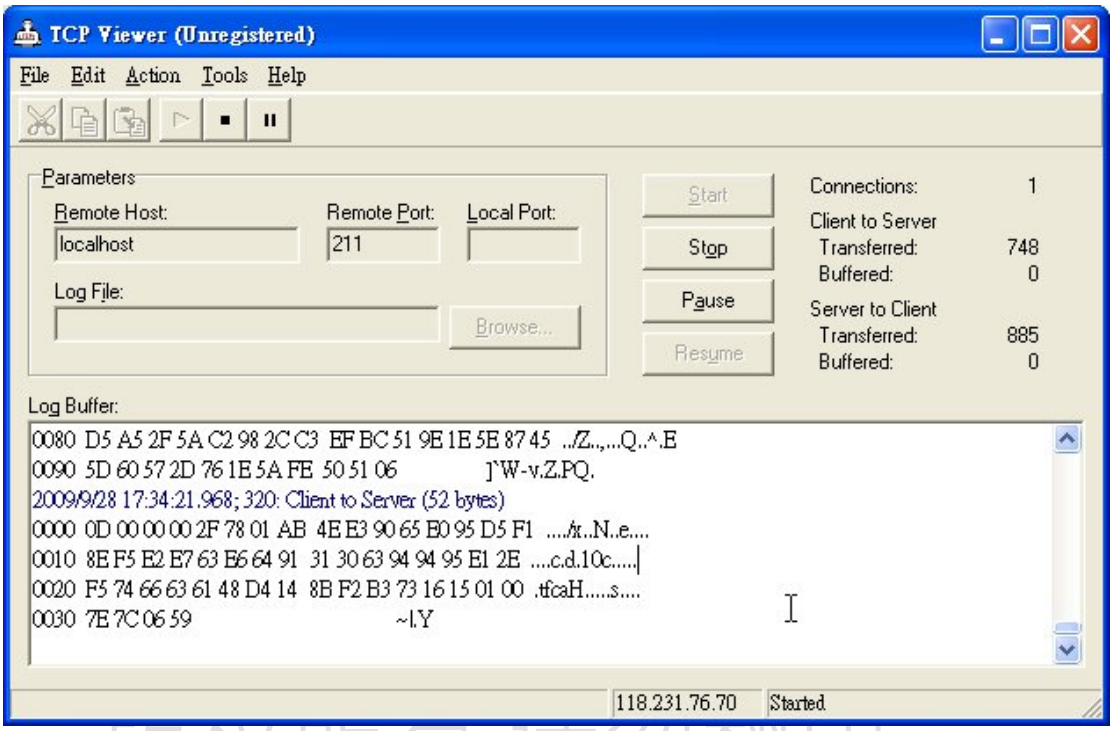


图 11 使用 TCP Viewer 观察的结果

现在您应该了解了如何使用 DataSnap 的过滤器功能以及如何开发客制化过滤器，接着让我们再讨论 DataSnap 10.3 中新增的请求过滤器(Request Filter)。

5-3 使用 DataSnap 的请求过滤器

所谓的请求过滤器是 DataSnap 在 10.3 版时加入的新型态的过滤器，它主要的目的是在客户端呼叫伺服端取得数据时，在请求的 URL 中加入特定的过滤器以便过滤伺服端回传的数据，例如客户端可以使用请求过滤器让伺服端只回传部份的数据，或是特定范围的数据，以减少不必要的数据占据网络带宽。

由于请求过滤器是使用在请求的 URL 中，因此 DataSnap 服务器必须支持 HTTP/HTTPS 通讯协议，或着是 DataSnap REST 型态的服务器。那客户端呢？读者可能会想由于客户端是藉由 URL 使用请求过滤器，因此客户端必须是浏览器型态的客户端，对吗？嗯原则上是正确的，因为浏览器可以直接使用 URL 向服务器请求服务，但由于 DataSnap 10.3 加入了新的组件 TDSRestConnection，因此即使是一般的 Windows 客户端也可以藉由 URL

使用请求过滤器，稍后我们会说明如何使用 TDSRestConnection 和请求过滤器，在那之前，我们需要先解释什么是请求过滤器。

5-3-1 请求过滤器的种类

DataSnap 10.3 提供了两个基本的请求过滤器:

请求过滤器	说明
SubString (ss)	SubString 请求过滤器允许客户端撷取部份由伺服端回传的字符串(String)或是串行流(Stream)。例如如果客户端只需要撷取伺服端回传的字符串的前 10 个字符，或是从位置第 20 到第 50 个的串行流字节。
Table (t)	和 SubString 非常的类似，但 Table 请求过滤器允许客户端撷取由伺服端回传的部份数据集中的数据，例如从第 15 笔到第 20 笔的数据。

SubString 和 Table 请求过滤器都提供了三个函式让客户端控制如何撷取数据，下面的表格说明了 SubString 请求过滤器的函式:

SubString (ss) 请求过滤器函式	说明
count 函式(c)	count 可控制从伺服端回传到客户端的字符数或是串行流的字节。count 函式接受一个参数，这个参数即是回传的字符数或是字节，而伺服端回传的字符或是字节的起始值是从 0 开始。例如如果客户端只需要撷取伺服端回传的前 10 个字符，那么我们可以使用如下的格式: ss.c=10 其中 ss 就是 SubString 请求过滤器 c 即指 count 函式 10 即是 count 函式的参数值，代表撷取前 10 个字符 例如下面的 URL http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.c=10 即是呼叫 GetDescription 方法并且只撷取回传字符串的前 10 个字符
offset 函式(o)	Offset 函式可跳过指定数目的字符或是字节，只撷取随后的数据。offset 函式接受一个参数，这个参数即是需要跳过的字符数或是字节。例如如果客户端不需要服务器回传字符串的前 10 个字符或是串行流的前 10 个字节，那么我们可以使用如下的格式:

	ss.o=10 其中 ss 就是 SubString 请求过滤器 o 即指 offset 函式 10 即是 offset 函式的参数值，代表跳过前 10 个字符 例如下面的 URL http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.o=10 即是呼叫 GetDescription 方法并且只撷取第 10 个字符之后的字符串数据
range 函式(r)	range 函式接受两个参数值，第一个参数值是指要从那一个位移位置开始撷取数据，第二个参数值是指要撷取几个字符数或是字节。例如，如果客户端只需要从第 10 个开始的字符并且存取其后的 5 个字符，那么我们可以使用如下的格式: ss.r=10, 5 其中 ss 就是 SubString 请求过滤器 r 即指 range 函式 10 即是 range 函式的第一个参数值，代表从第 10 个字符开始撷取数据， 5 即是 range 函式的第 2 个参数值，代表从第 10 个字符开始撷取 5 个字符的数据， 例如下面的 URL http://localhost:8080/datasnap/rest/TServerMethods1/GetDescription?ss.r=10,5 即是呼叫 GetDescription 方法并且只撷取第 10 个字符之后的 5 个字符的数据

下面的表格说明了 Table 请求过滤器的函式，它的使用方法和 SubString 非常的类似:

Table (t) 请求过滤器函式	说明
count 函式	count 可控制从伺服端回传到客户端的记录数。count 函式接受一个参数，这个参数即是回传记录数，而伺服端回传的记录数起始值是从 0 开始。例如如果客户端只需要撷取伺服端回传的前 10 笔资料，那么我们可以使用如下的格式: t.c=10 其中 t 就是 Table 请求过滤器 c 即指 count 函式 10 即是 count 函式的参数值，代表撷取前 10 笔资料 例如下面的 URL

	http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.c=10 即是呼叫 GetEmployee 方法并且只撷取回传 Employee 数据表中的的前 10 笔资料
offset 函式	Offset 函式可跳过指定数目记录，只撷取随后的数据。offset 函式接受一个参数，这个参数即是需要跳过的记录数。例如如果客户端不需要服务器回传字符串的前 10 笔数据，那么我们可以使用如下的格式: t.o=10 其中 t 就是 Table 请求过滤器 o 即指 offset 函式 10 即是 offset 函式的参数值，代表跳过前 10 笔资料 例如下面的 URL http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.o=10 即是呼叫 GetEmployee 方法并且只撷取第 10 个字符之后的字符串数据
Range 函式	range 函式接受两个参数值，第一个参数值是指要从那一个位移位置开始撷取数据，第二个参数值是指要撷取几笔资料。例如，如果客户端只需要从第 10 笔开始的数据并且存取其后的 5 笔资料，那么我们可以使用如下的格式: t.r=10, 5 其中 t 就是 Table 请求过滤器 r 即指 range 函式 10 即是 range 函式的第一个参数值，代表从第 10 笔数据开始撷取数据， 5 即是 range 函式的第 2 个参数值，代表从第 10 笔数据后开始撷取 5 笔数据， 例如下面的 URL http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.r=10,5 即是呼叫 GetEmployee 方法并且只撷取第 10 笔数据之后的 5 笔数据

5-3-2 使用请求过滤器

在前一小节说明 SubString 和 Table 请求过滤器的表格中已经列出了如何使用它们的范例，例如下图就是在浏览器中使用 Table 请求过滤器:

http://localhost:8080/datasnap/rest/TServerMethods1/GetEmployee?t.r=10,5

的结果:

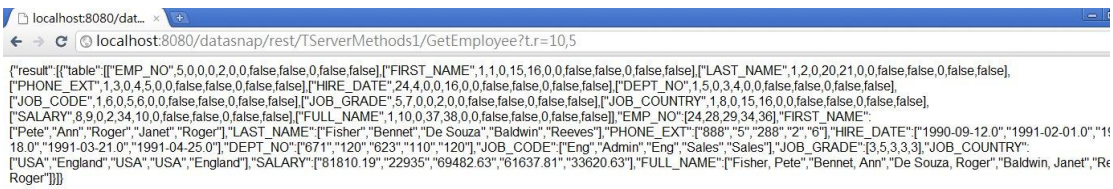


图 12 在浏览器中使用 Table 请求过滤器

但是除了浏览器之外，对于原生 Windows 应用程序的客户端来说请求过滤器也是非常有用的功能，因为请求过滤器可以在伺服端就过滤客户端需要的资料，如此一来就可以减少网络传递的数据量，但问题是如何在原生 Windows 应用程序中使用请求过滤器呢？答案就是使用 TDSRestConnection 组件并且藉由它来产生客户端的 Proxy 程序代码，再藉由它产生的客户端的 Proxy 程序代码使用请求过滤器。下面的小节即说明了如何使用 TDSRestConnection 组件和请求过滤器。

藉由 TDSRestConnection 元使用请求过滤器

首先建立一个 DataSnap REST Application 项目，在 ServerMethodUnit1 程序单元中使用 dbExpress 组件链接 InterBase 的 Employee 数据表:

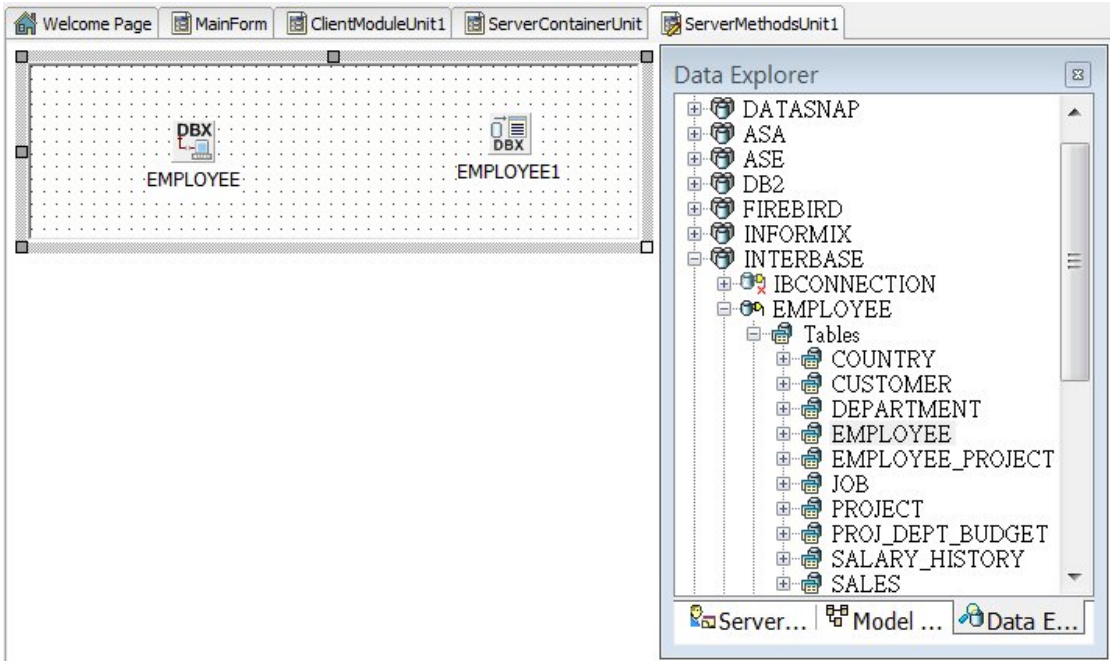


图 13 在 ServerMethodUnit1 程序单元中使用 dbExpress 组件链接 InterBase 的 Employee 数据表

开启 ServerMethodUnit1 程序单元的程序代码，并且加入两个范例方法:

```
public
   function GetDescription : string;
   function GetEmployee : TDataSet;
 end;
```

并且实作这两个范例方法如下:

```
function TServerMethods1.GetDescription: string;
```

```
begin
  Result := '让您一次掌握 Embarcadero 最尖端的产品和技术' +
            '改变软件架构和使用的革新性产品 AppWave' +
            '万众嘱目和期待的窗口原生开发工具新王者 Delphi 10.3 预览版';
end;

function TServerMethods1.GetEmployee: TDataSet;
begin
  EMPLOYEE.Connected := True;
  EMPLOYEE1.Active := True;
  Result := EMPLOYEE1;
end;
```

GetDescription 回传字符串的数据以准备稍后使用 SubString 请求过滤器来测试，而 GetEmployee 则是回传 Employee 数据表中的数据，以准备稍后使用 Table 请求过滤器来测试。

现在编译并且执行此范例 DataSnap REST 服务器。

接着在项目群组再建立一个 VCL Form Application 项目，再于其中建立一个 DataSnap Client Module，DataSnap Client Module 会产生一个 TSQLConnection 组件链接 DataSnap REST 服务器并且产生使用 TSQLConnection 组件链接服务器封装的客户端类别，现在在此 DataSnap Client Module 中加入一个 TDSRestConnection 组件，然后使用鼠标右键选择 Generate DataSnap client classes，如下图所示:

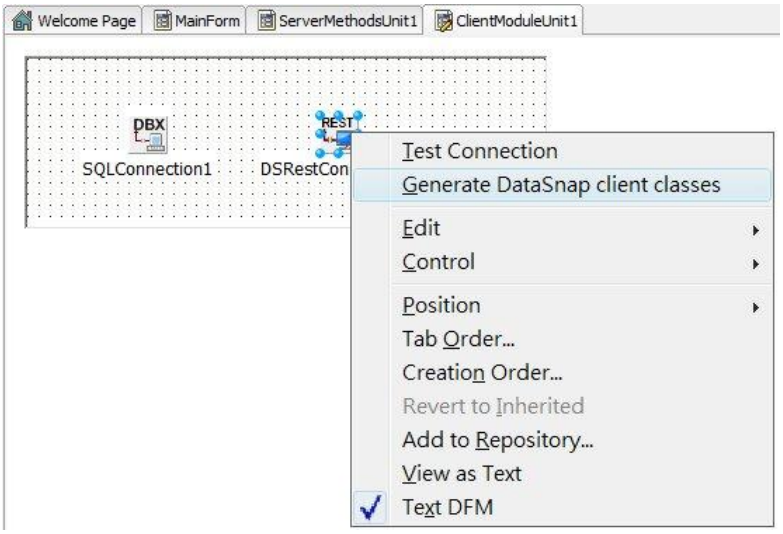


图 14 使用 TDSRestConnection 产生客户端 Proxy 类别程序代码

TDSRestConnection 此时便会产生另外一个封装服务器服务的客户端类别程序代码，这个新产生的类别程序代码和 TSQLConnection 组件产生的类别程序代码的差别在于 TDSRestConnection 组件产生的类别程序代码是使用 REST 呼叫方式来呼叫伺服端的服务，这也就是说这个类别程序代码会藉由 REST 的 URL 呼叫伺服端，因此我们也就可以在其中使用请求过滤器。

开启由 TDSRestConnection 组件产生的类别程序代码，我们可以在其中找到 GetDescription 和 GetEmployee 的宣告原型:

```
function GetDescription(const ARequestFilter: string = ''): string;
function GetEmployee(const ARequestFilter: string = ''): TDataSet;
```

从上面的程序代码中可以看到我们可以藉由参数的方式把请求过滤器传递给上述的函式，再由这些函式自动产生正确的 URL 并且向伺服端发出请求。反观如果我们开启由 TSQLConnection 产生的类别程序代码，我们可以看到下面的原型宣告，由 TSQLConnection 产生的类别程序代码是无法使用请求过滤器的:

```
function GetDescription: string;
function GetEmployee: TDataSet;
```


这个原因当然是因为 TSQLConnection 是使用 dbExpress 呼叫服务器的服务，而不是使用 REST 呼叫惯例。

OK，现在我们就可以开始测试由 TSQLConnection 产生的类别程序代码以及由 TDSRestConnection 组件产生的类别程序代码，现在让我们设计范例客户端应用程序的主表格如下:

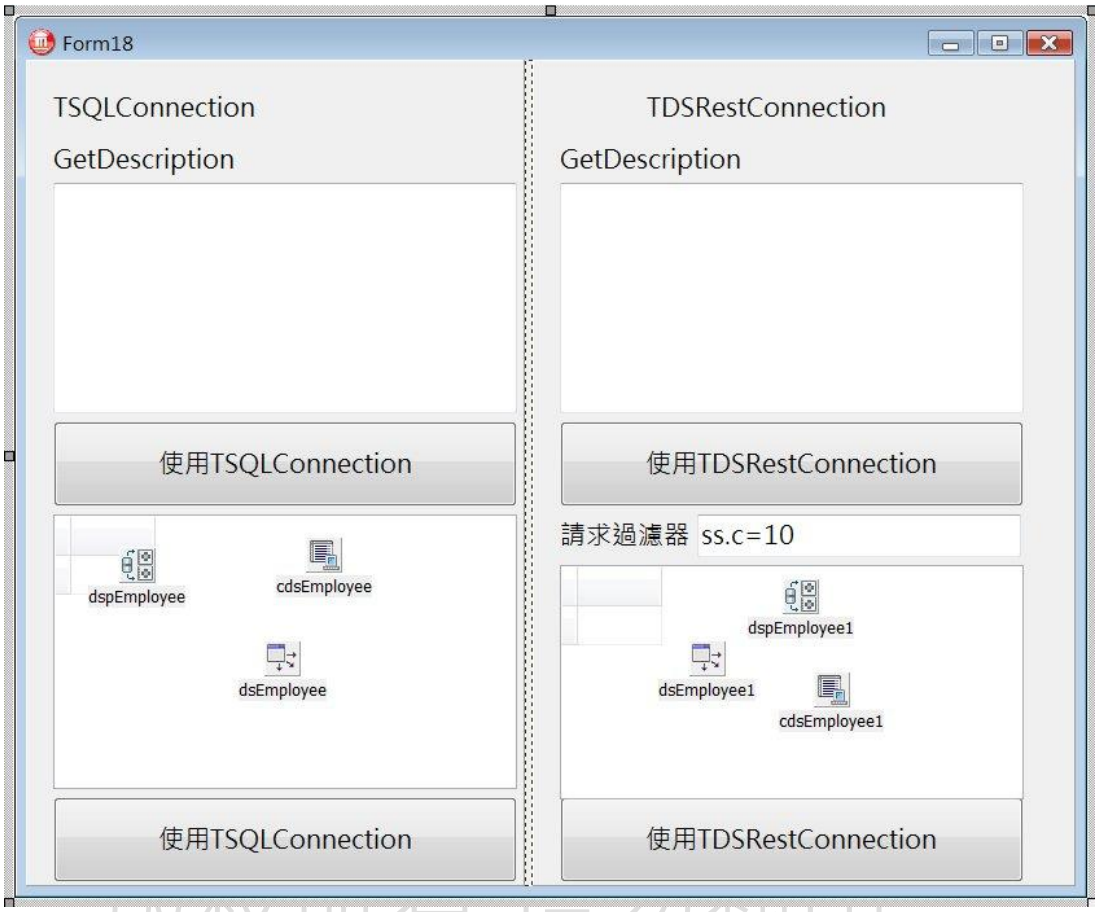


图 15 范例客户端主窗体

接着使用下面的程序代码藉由 TSQLConnection 组件呼叫伺服端的 GetDescription 方法:

```
procedure TForm18.Button1Click(Sender: TObject);
var
  aServer : TServerMethods2Client;
begin
  aServer := ClientModule1.ServerMethods2Client;
  mmNoRF.Lines.Text := aServer.GetDescription;
end;
```

再使用下面的程序代码藉由 TDSRestConnection 组件呼叫伺服端的 GetDescription 方法，由于藉由 TDSRestConnection 可使用请求过滤器，因此我们在呼叫 GetDescription 时把主表格中 TEdit 组件的 Text 特性值传入做为请求过滤器:

```
procedure TForm18.Button2Click(Sender: TObject);
var
  aRSServer : TServerMethods1Client;
```

```
begin
  aRSServer := TServerMethods1Client.Create(ClientModule1.DSRestConnection1);
  try
    mmRF.Lines.Text := aRSServer.GetDescription(edtFilter.Text);
  finally
    aRSServer.Free;
  end;
end;
```

编译并且执行范例客户端应用程序，从下图中我们可以看到使用 TSQLConnection 呼叫服务器的 GetDescription 取得了所有的字符串内容，但使用 TDSRestConnection 呼叫服务器时则可以使用请求过滤器，在这里我们使用了 ss.c=10 代表只存取前 10 个字符，从执行结果来看执行结果果然是正确的，服务器只会传递 10 个字符到客户端而不会传递整个字符串。

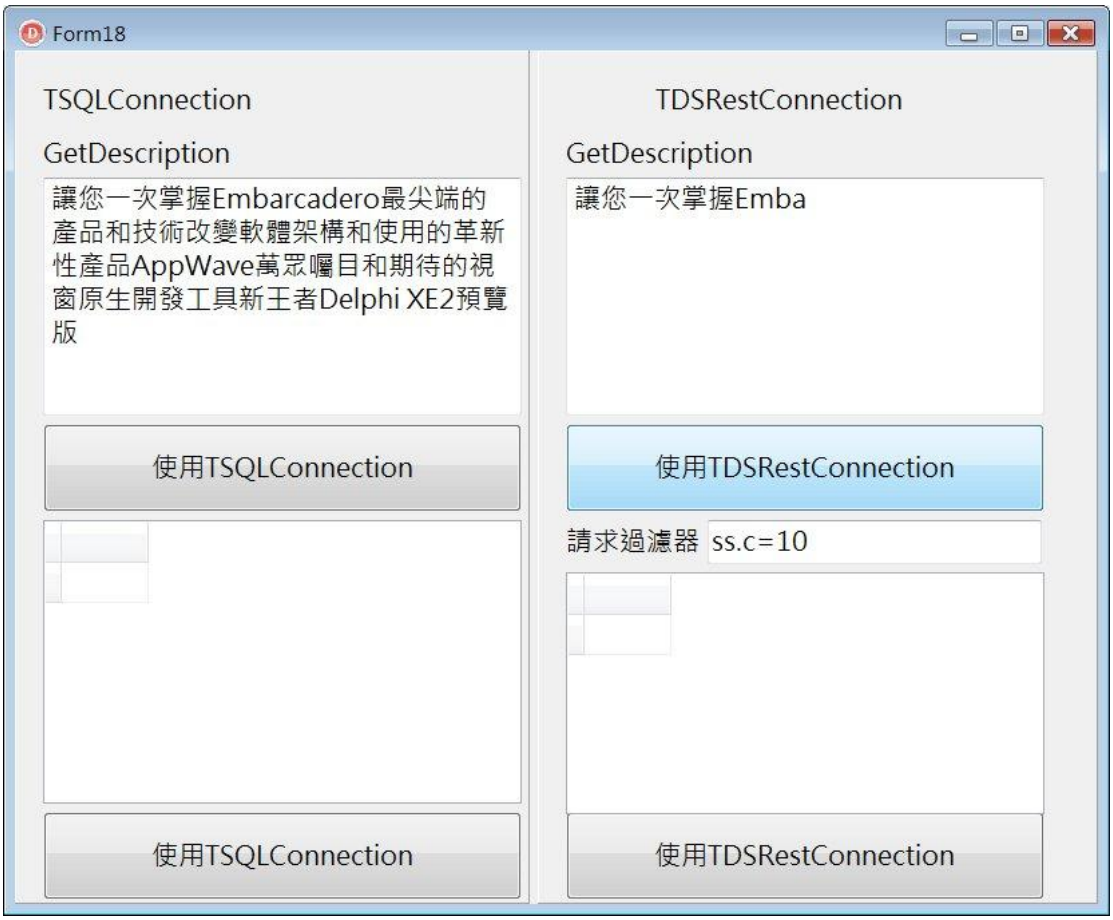


图 16 使用 SubString 请求过滤器的执行结果

再让我们呼叫 GetEmployee 存取数据表的数据看看请求过滤器的效果。同样的先让我们使用 TSQLConnection 呼叫服务器的 GetEmployee 方法:

```
procedure TForm18.Button3Click(Sender: TObject);
```

```
var
  aServer : TServerMethods2Client;
  aDataSet : TDataSet;
begin
  aServer := ClientModule1.ServerMethods2Client;
  cdsEmployee.Active := False;
  aDataSet := aServer.GetEmployee;
  dspEmployee.DataSet := aDataSet;
  cdsEmployee.Active := True;
end;
```

再使用 TDSRestConnection 呼叫服务器的 GetEmployee 方法，并且传入用户在主窗体中使用的请求过滤器:

```
procedure TForm18.Button4Click(Sender: TObject);
var
  aRSServer : TServerMethods1Client;
  aDataSet : TDataSet;
begin
  aRSServer := TServerMethods1Client.Create(ClientModule1.DSRestConnection1);
  try
    cdsEmployee.Active := False;
    aDataSet := aRSServer.GetEmployee(edtFilter.Text);
    dspEmployee.DataSet := aDataSet;
    cdsEmployee.Active := True;
  finally
    aRSServer.Free;
  end;
end;
```

编译并且执行范例客户端应用程序，这次让我们使用请求过滤器 t.c=3，代表只存取 Employee 数据表前 3 笔的资料：

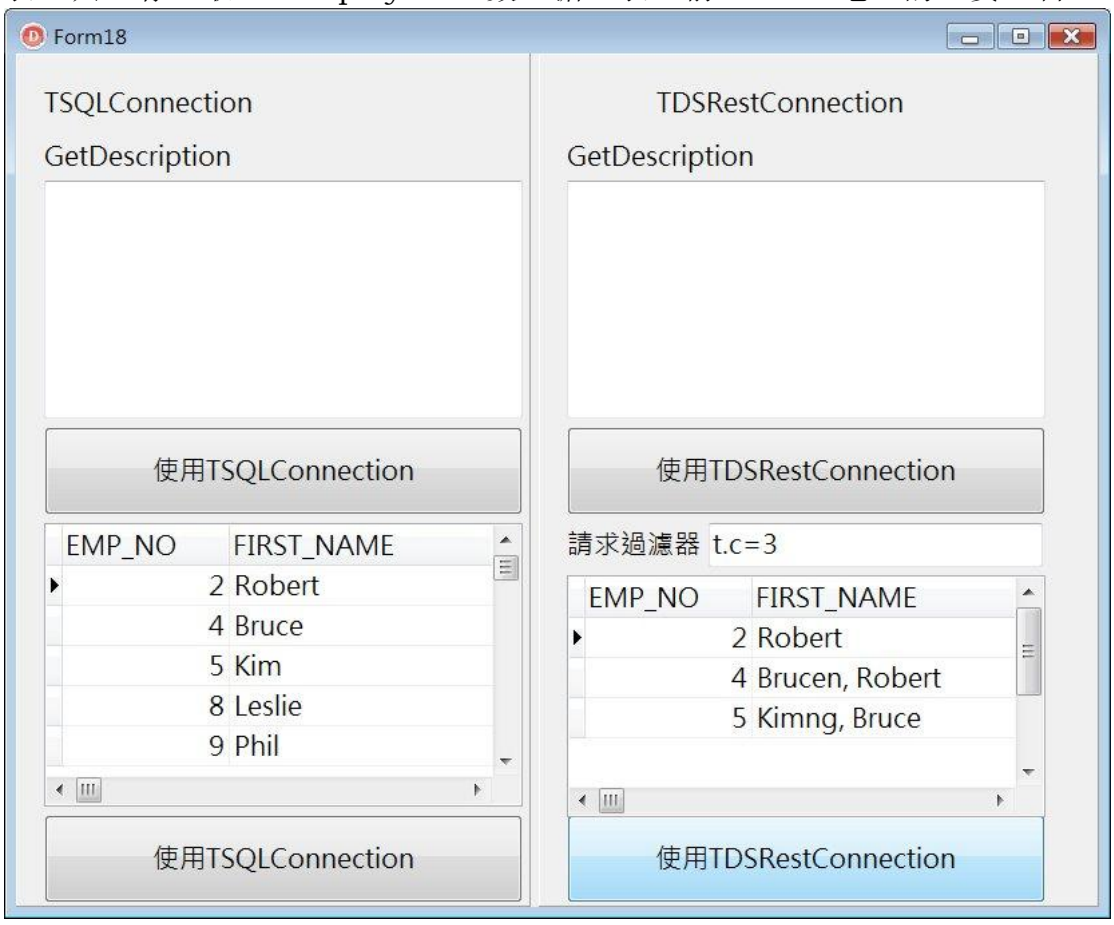


图 17 使用 Table 请求过滤器的执行结果

从上面的执行结果可以看到使用 TSQLConnection 果然取得了 Employee 数据表所有的数据，而右方使用 t.c=3 请求过滤器的也的确只存取 3 笔数据，伺服端的确只传递前 3 笔的数据回客户端。

5-4 结论

DataSnap 的过滤器功能允许开发人员控制传递 JSON 资料的实际格式，开发人员可以进行加/解密以保护以文字形式传递数据的 JSON 格式，而 DataSnap 10.3 新加入的请求过滤器则允许开发人员在客户端即可控制伺服端需要传递到客户端的数据条件，以消除不必要的数据占据网络带宽，或是在客户端即可使用 URL 来客制化伺服端传递到客户端的数据。当然，开发人员更可以结合 DataSnap 的过滤器和请求过滤器功能更进一步控制传递的数据，DataSnap 的过滤器是个非常实用的功能。

第6章 DataSnap生命周期和管理功能

在前面讨论的章节中建立的 DataSnap 伺服端的服务类别都是使用 Server 生命周期型态的服务器，除了 Server 型态之外，还有 Session 型态以及 Invocation 型态。每一种不同的 DataSnap 伺服端的服务型态都拥有不同的特性，本章将讨论 DataSnap 伺服端服务的生命周期的意义。

6-1 DataSnap 伺服端服务的生命周期

从 DataSnap XE 之后便允许开发人员建立不同生命周期的 DataSnap 伺服端的服务，在 10.3 中提供了三种不同的生命周期，开发人员可以在 TDSServerClass 组件的 LifeCycle 特性中设定，下面的表格说明了每一种生命周期的意义:

特性值	说明
Server	在整个 DataSnap 服务器中只会建立一个服务类别对象以服务所有的客户端，只有当 DataSnap 服务器结束时才会释放此服务类别对象
Session	在 DataSnap 服务器中会为每一个连结的客户端建立一个专属的服务类别对象服务此客户端，一旦客户端结束或是关闭 TSQLConnection 的链接，此服务类别对象便会被释放
Invocation	在 DataSnap 服务器中每当客户端执行一次请求时，在 DataSnap 服务器便会为这个请求建立一个服务类别对象服务此客户端请

	求，当请求执行结束后，DataSnap 服务器便会释放此服务类别对象

从上面表格的说明我们可以了解，使用 Server 生命周期的伺服端服务类别只会在 DataSnap 服务器中建立一个服务对象，使用 Session 生命周期的伺服端服务类别则视客户端使用使用多少 TSQLConnection 组件藉由 DataSnap 驱动程序链接到 DataSnap 服务器的数目而在 DataSnap 服务器中建立相对应的服务对象来服务服务，最后使用 Invocation 生命周期的伺服端服务类别则会在每一次客户端呼叫 DataSnap 服务器时被建立来服务客户端，因此被建立和释放的次数相当巨量。

那么开发人员应该如何决定使用那一种的生命周期伺服端服务类别呢?这当然时要看伺服端服务类别提供的服务种类，下面的表格简单的说明了每一种生命周期适用的场景:

生命周期	说明
Server	提供所有客户端公用的服务，由于所有客户端都使用单一的服端服务类别对象，因此使用这种生命周期的服务对象负荷都比较大，使用这种生命周期的服务对象适合提供快速，简单，无状态的服务为主。
Session	由于这种生命周期型态的服端服务类别对象为会每一个客户端的连结建立一个专属的服务对象，因此可提供客户端无状态以及有状态的服务，也可提供长期，负荷较大的服务。
Invocation	这种生命周期型态的服端服务类别对象只存在于每一个客户端的呼叫周期，因此适合提供可在背景执行的服务，或是执行数据库的预储程序，或是批处理等和客户端较无相关的服务。不过由于使用这种生命周期的服务类别会被频繁的建立和释放，因此这种服务类别应该尽量精简，如果需要使用数据库，那么也应该搭配使用 dbExpress 的链接池功能以加快服务速度。

6-1-1 Server 生命周期

由于使用 Server 生命周期的 DataSnap 伺服端对象只有一个并且服务所有的客户端请求，因此这种型态的伺服端服务类别对象负荷可能很重，开发人员应

该尽量让每一个客户端的请求快速完成，以便服务更多的客户端请求并且减少伺服端服务类别对象的负荷，要设定特定的服务类别为 Server 生命周期，开发人员只需如下图在对象查看器中设定TDSServerClass组件的LifeCycle特性值为Server即可：

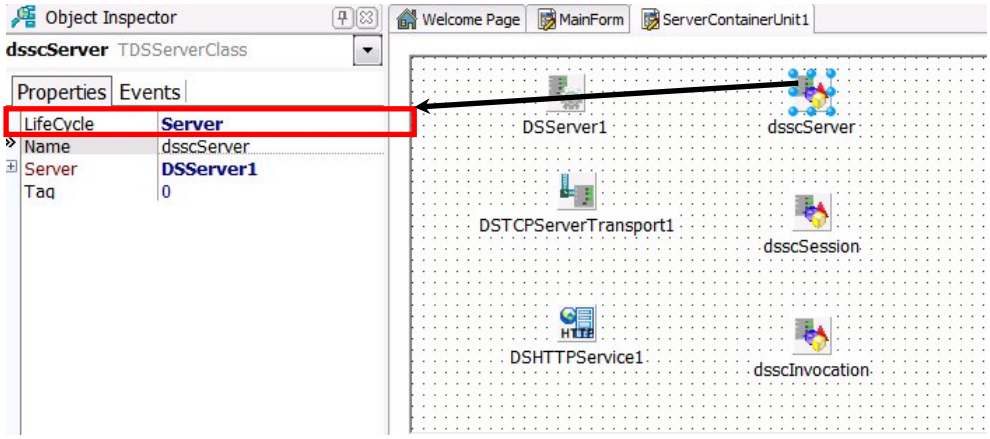


图 6-1 设定 TDSServerClass 组件的 LifeCycle 特性值为 Server 生命周期

那么由这个TDSServerClass组件输出的服务类别就会自动使用Server生命周期，例如上图的 dssServer 组件使用了下面的程序代码输出服务类别TdssmServer：


```
procedure TServerContainer1.dsscServerGetClass(
  DSServerClass: TDSServerClass; var PersistentClass: TPersistentClass);
begin
  PersistentClass := ServerMethodsUnit1.TdssmServer;
end;
```

因此 TdssmServer 类别就使用了 Server 生命周期，而在 TdssmServer 类别中输出了两个方法可让客户端查询邮政编码，或是使用邮政编码查询区域名称：

```
   function GetZipCode(const SDistrict : String) : Integer;
   function GetDistrictFromZipCode(sZipCode : String) : String;
```

为了加快执行速度以便在更短的时间内服务更多的客户端请求，因此TdssmServer服务类别使用了内存数据表来服务客户端：

```
function TdssmServer.CreateZipCodeTable: TClientDataSet;
begin
  Result := TClientDataSet.Create(nil);

  with Result do
  begin
```

```
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 20;
      Name := '地区名';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := '邮政编码';
    end;
    with IndexDefs.AddIndexDef do
    begin
      Fields := '地区名';
      Name := 'idxTD';
    end;
    CreateDataSet;
    IndexDefs.Update;
    IndexName := 'idxTD';
  end;
end;

function TdssmServer.GetDistrictFromZipCode(sZipCode : String) : String;
begin
  Result := cdsZipCode.Lookup('邮政编码', sZipCode, '地区名');
end;

function TdssmServer.GetZipCode(const SDistrict : String) : Integer;
begin
  Result := cdsZipCode.Lookup('地区名', sDistrict, '邮政编码');
end;
```

从下图的范例 DataSnap 应用系统可以看到范例 TdssmServer 服务对象在 DataSnap 服务器中只被建立了一个对象，这当然是因为使用了 Server 生命周期，而且在不到 1 秒的时间就完成了客户端的请求，因此 TdssmServer 是很好的使用 Server 生命周期的范例。

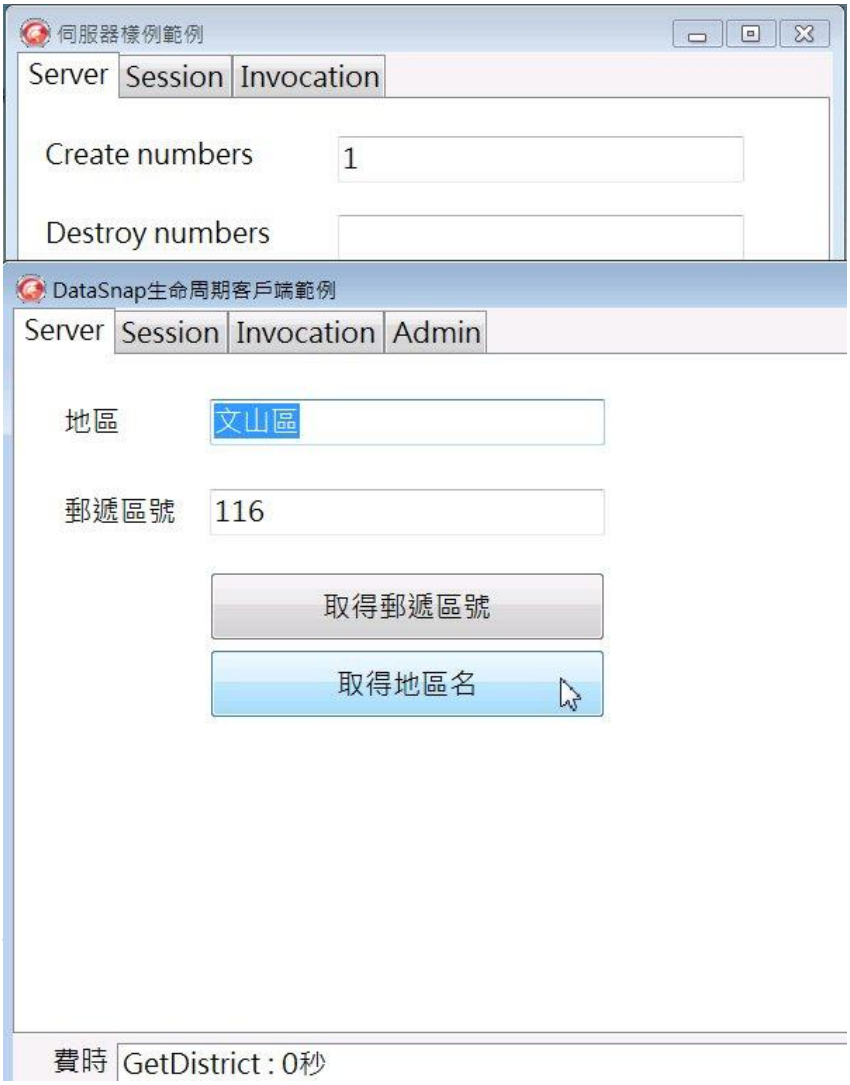


图 6-2 TdssmServer 服务对象快速的完成客户端的请求

6-1-2 Session 生命周期

Session 生命周期是 TDSServerClass 组件内定的生命周期型态，每一个使用 TSQLConnection 组件藉由 DataSnap 驱动程序链接到 DataSnap 服务器的客户端都会建立一个专属的服务对象，因此不同的客户端并不会相互干扰，要设定特定的服务类别为 Session 生命周期，开发人员只需如下图在对象查看器中设定 TDSServerClass 组件的 LifeCycle 特性值为 Session 即可:

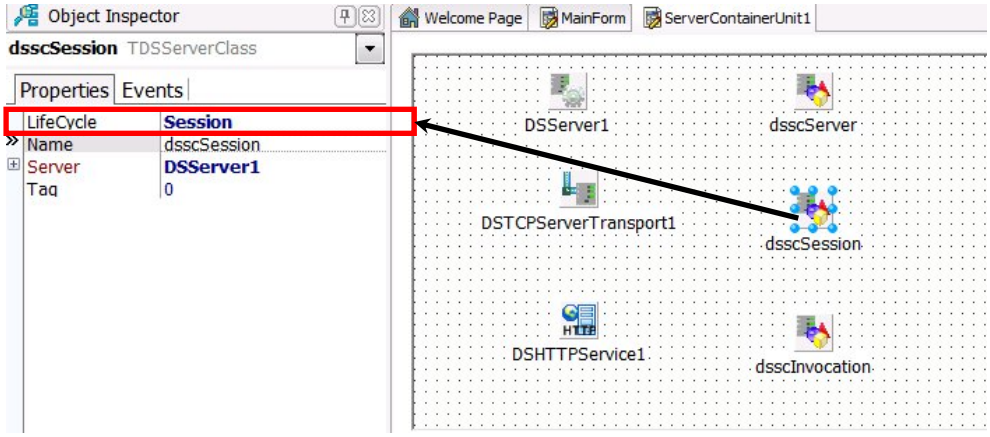


图 6-3 设定 TdssmSession 组件的 LifeCycle 特性值为 Session 生命周期

由于上图的 dssSession 组件输出了 TdssmSession 类别，因此 TdssmSession 服务对象会为每一个客户端建立一个专属的对象：

```
procedure TServerContainer1.dsscSessionGetClass(DSServerClass: TDSServerClass;
  var PersistentClass: TPersistentClass);
begin
  PersistentClass := uServerModuleSession.TdssmSession;
end;
```


在 TdssmSession 类别中输出了一个方法 GetThreadID 让客户端呼叫，在 GetThreadID 方法中使用了 dbExpress 组件链接到 InterBase 数据表，撷取其中的书名信息并且结合伺服端的服务线程 ID 回传给客户端：

```
001    function TdssmSession.GetThreadID : String;
002    begin
003      DXEHPDBP.Connected := True;
004      try
005        cdsSessionQuery.Active := True;
006        cdsSessionQuery.MoveBy(Random(cdsSessionQuery.RecordCount));
007        Result := cdsSessionQuery.FieldByName('BOOKNAME').AsString + ' : ' +
IntToStr(TThread.CurrentThread.ThreadID);
008      finally
009        cdsSessionQuery.Active := False;
010        DXEHPDBP.Connected := False;
011      end;
012    end;
```

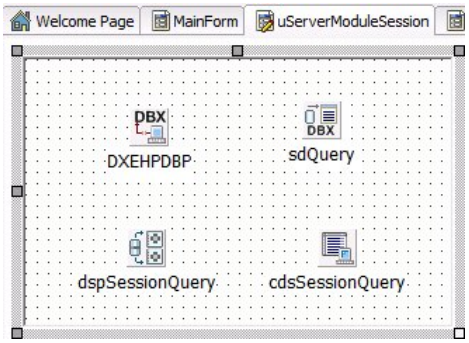


图 6-4 uServerModuleSession 数据模块使用了 dbExpress 组件链接 InterBase 数据库

如果现在我们执行数个这个范例 DataSnap 客户端应用程序，那么 DataSnap 服务器会为每一个范例 DataSnap 客户端应用程序建立一个专属的 TdssmSession 对象服务客户端。因此如果此时我们执行许多份的此范例 DataSnap 客户端应用程序，例如 100 个客户端应用程序，那么 DataSnap 服务器便会在伺服端建立 100 个 TdssmSession 对象，每一个 TdssmSession 对象又需要使用一个 TSQLConnection 链接到数据库，那么很快的所有数据库链接都会被使用完毕而造成错误，而且 DataSnap 服务器的负荷会非常的沉重，因此在使用 Session 或是稍后介绍的 Invocation 生命周期的服务对象如果会链接到数据库的话，那么笔者建议一定要开启 dbExpress 的链接池功能，以便让所有客户端分享数据库链接，而且在撰写服务方法时，一定要在方法执行完毕之际关闭数据库链接，以释放数据库链接回 dbExpress 的链接池让其他方法或是其他客户端重复使用，例如在上面的 GetThreadID 方法中最后在 010 行关闭了 TSQLConnection 对于数据库的链接以释放 InterBase 的连结回 dbExpress 的连结池。

例如下图是执行数个范例 DataSnap 客户端在没有开启 dbExpress 连结池的情形下，DataSnap 服务器开启了数个和 InterBase 的数据库链接，而且执行的速度大约为 0.15 秒服务每一次的客户端请求:

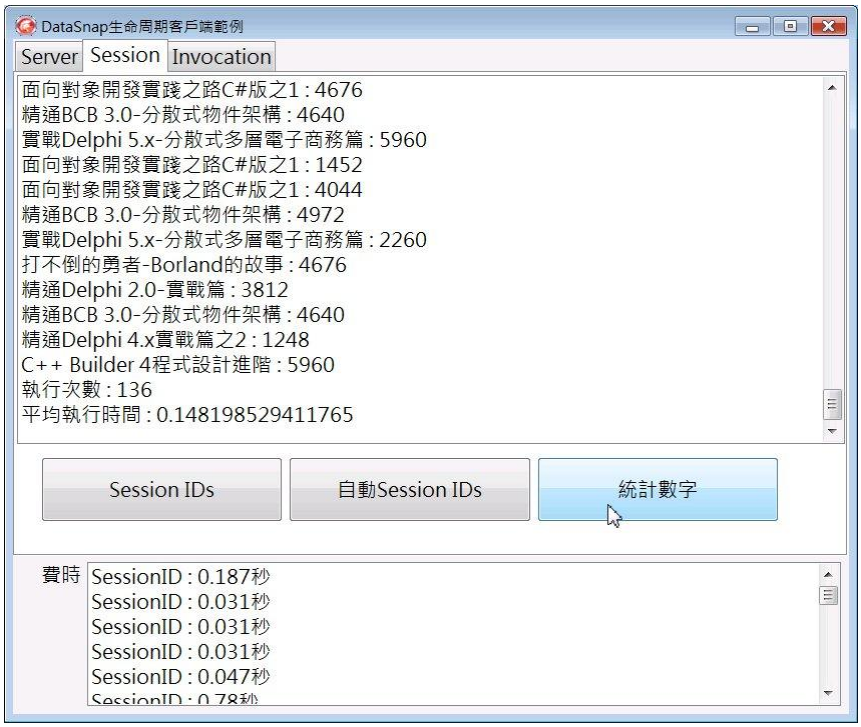


图 6-5 使用 Session 生命周期的服务对象服务客户端的请求

现在如果我们开启图 6-4 的 TSQLConnection 使用 dbExpress 的链接池功能，如下所示:

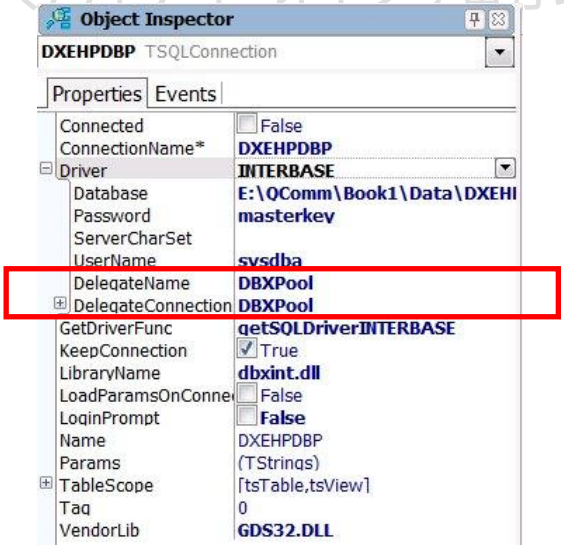


图 6-6 开启使用 dbExpress 的链接池功能

那么如果我们再次执行许多范例客户端 DataSnap 应用程序并且呼叫 GetThreadID 方法，那么在 DataSnap 服务器观察的话，就会发现使用的 InterBase 连结变少了，而且如下图所示执行效率也增加了，现在平均只需要 0.02 秒左右的运行时间。

图 6-7 使用 Session 生命周期的服务对象并且开启 dbExpress 链接池功能服务客户端的请求

6-1-3 Invocation 生命周期

使用 Invocation 生命周期的伺服端服务对象会在每一个客户端请求时被建立，服务完客户端请求之后会被释放，开发人员只需如下图在对象查看器中设定 TDSServerClass 组件的 LifeCycle 特性值为 Invocation 即可:

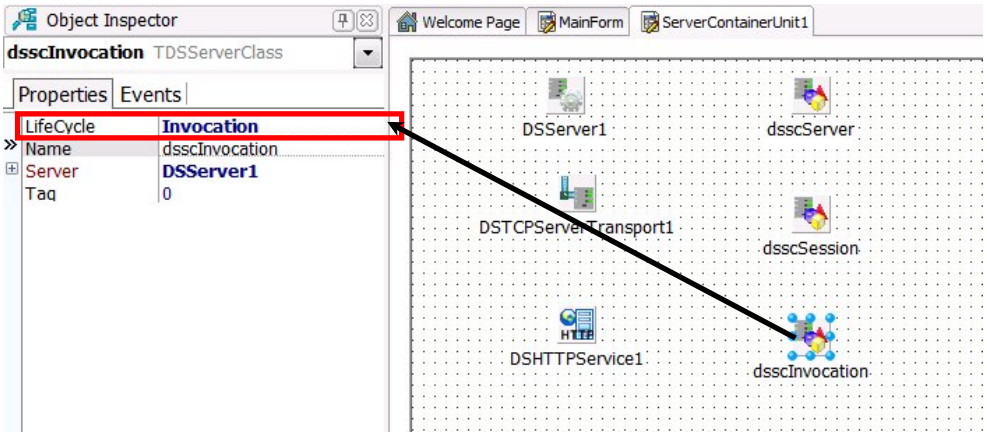


图 6-8 设定 TdssmInvocation 组件的 LifeCycle 特性值为 Invocation 生命周期

使用 Invocation 生命周期的服务类别由于会被频繁的建立和释放，因此适合使用在小型，快速服务和无状态的应用中，这种服务类别不应该很庞大以避免花费太多的时间在建立和释放的过程中，如果这种服务类别需要存取数据库，那么应该使用 dbExpress 链接池功能，而且也应该在服务方法执行完毕之后立刻释放数据库的链接。

不过由于目前 DataSnap 框架在释放 Invocation 生命周期的服务对象实作程序代码上有臭虫，因此目前 Invocation 生命周期的服务对象在服务之后不会自动被释放而造成内存漏失的错误，因此开发人员目前需要在程序代码中释放 Invocation 生命周期的服务对象。这个释放的工作可以藉由 TDSDestroyInstanceEventObject 类别来帮忙:

```
 TDSDestroyInstanceEventObject = class(TDSEventObject)
 public
   constructor Create(const ADbxContext: TDBXContext; const AServer: TDSCustomServer;
const ATransport: TDSServerTransport; const ADbxConnection: TDBXConnection);
 private
   FServerClassInstance: TObject;
 public
   property ServerClassInstance: TObject read FServerClassInstance write
FServerClassInstance;
 end;
```

TDSDestroyInstanceEventObject 类别的 ServerClassInstance 特性就是 Invocation 生命周期的服务对象样例，因此在 Invocation 生命周期的服务对象服务完客户端的请求之后，请开发人员在使用 Invocation 生命周期的 TDSServerClass 组件的 OnDestroyInstance 事件中撰写如下的程序代码以释放 Invocation 生命周期的服务对象:


```
procedure TServerContainer1.dsscInvocationDestroyInstance(
 DSDestroyInstanceEventObject: TDSDestroyInstanceEventObject);
begin
 DSDestroyInstanceEventObject.ServerClassInstance.Free;
end;
```

当使用 Invocation 生命周期的服务对象服务完客户端之后，DataSnap 框架会呼叫使用 Invocation 生命周期的 TDSServerClass 组件的 OnDestroyInstance 事件，并且传递 TDSDestroyInstanceEventObject 对象给这个事件，因此我们只需要在 OnDestroyInstance 事件中藉由存取 TDSDestroyInstanceEventObject 对象中的 ServerClassInstance 特性值，并且呼叫它的 Free 方法即可。当然，待日后 Embarcadero 修正了这个臭虫之后就无需上面的程序代码了。

6-2 DataSnap 管理功能

在 DataSnap 服务器中除了开发人员的服务类别之外，DataSnap 框架也提供了一些内建的类别，其中的 DSAdmin 类别是 DataSnap 服务器中内建的

管理类别，开发人员可以在 DataSnap 服务器中呼叫 DSAdmin 类别以存取它的服务，此外 DSAdmin 也可以被 DataSnap 客户端呼叫来存取其提供的服务，但是在 DataSnap 客户端，开发人员是使用 TDSAdminClient 类别来存取 DataSnap 服务器中的 DSAdmin 服务，下面的表格说明了这两个类别的使用方式:

类别	说明
DSAdmin	DataSnap 服务器中提供管理功能的类别
TDSAdminClient	DataSnap 客户端类别，客户端可使用此类别存取 DataSnap 服务器中 DSAdmin 类别对象提供的服务，通常客户端的 Proxy 类别会由此类别继承下来

DSAdmin 类别提供了许多的服务方法，下面的表格是 DSAdmin 类别提供的服务函式:

函式	说明
GetPlatformName	取得 DataSnap 服务器执行的平台名称
ClearResources	清除 DataSnap 服务器配置的资源
FindPackages	回传所有 DataSnap 服务器中的封包(Package)信息，这些信息都封装在回传的 TDBXReader 对象中
FindClasses	回传某特定封包中所有符合搜寻类别样例的信息，这些信息都封装在回传的 TDBXReader 对象中
FindMethods	回传某特定封包中，某特定类别中所有符合搜寻方法样例的信息，这些信息都封装在回传的 TDBXReader 对象中
CreateServerClasses	在 DataSnap 服务器中建立服务类别
DropServerClasses	在 DataSnap 服务器中删除服务类别
CreateServerMethods	在 DataSnap 服务器中建立服务方法
DropServerMethods	在 DataSnap 服务器中删除服务方法
GetServerClasses	取得 DataSnap 服务器中所有的服务类别信息，这些信息都封装在回传的 TDBXReader 对象中
ListClasses	取得 DataSnap 服务器中所有的服务类别信息，这些信息都封装在回传的 TJSONArray 对象中
DescribeClass	回传 DataSnap 服务器中特定服务类别的叙述信息，这些信息都封装在回传的 TJSONObject 对象中
ListMethods	取得 DataSnap 服务器中所有的服务方法信息，这些信息都封装在回传的 TJSONArray 对象中
DescribeMethod	回传 DataSnap 服务器中特定服务方法的叙述信息，

	这些信息都封装在回传的 TJSONObject 对象中
GetServerMethods	回传 DataSnap 服务器中所有的服务方法信息，这些信息都封装在回传的 TDBXReader 对象中
GetServerMethodParameters	回传 DataSnap 服务器中所有的服务方法的参数信息，这些信息都封装在回传的 TDBXReader 对象中
GetConnection	回传在传 DataSnap 服务器中的 TDBXConnection 对象
GetDatabaseConnectionProperties	回传在传 DataSnap 服务器中所有的数据库链接信息，这些信息都封装在回传的 TDBXReader 对象中

从上面的表格可以了解许多 DSAdmin 的服务方法都是回传 TDBXReader 对象或是 JSON 相关类别对象，在前面的章节中已经过如何使用这些对象。

如果在 DataSnap 客户端需要存取 DSAdmin 的服务的话，那么开发人员可以使用 TDSAdminClient 类别，下面的表格是 TDSAdminClient 类别提供的服务函式:

函式	说明
GetPlatformName	取得 DataSnap 服务器执行的平台名称
ClearResources	清除 DataSnap 服务器配置的资源
FindPackages	回传所有 DataSnap 服务器中的封包(Package)信息，这些信息都封装在回传的 TDBXReader 对象中
FindClasses	回传某特定封包中所有符合搜寻类别样例的信息，这些信息都封装在回传的 TDBXReader 对象中
FindMethods	回传某特定封包中，某特定类别中所有符合搜寻方法样例的信息，这些信息都封装在回传的 TDBXReader 对象中
GetServerMethods	回传 DataSnap 服务器中所有的服务方法信息，这些信息都封装在回传的 TDBXReader 对象中
GetServerMethodParameters	回传 DataSnap 服务器中所有的服务方法的参数信息，这些信息都封装在回传的 TDBXReader 对象中
GetDatabaseConnectionProperties	回传在传 DataSnap 服务器中所有的数据库链接信息，这些信息都封装在回传的 TDBXReader 对象中

BroadcastToChannel	传递讯息给回叫信道
BroadcastObjectToChannel	传递对象给回叫信道
ListClasses	取得 DataSnap 服务器中所有的服务类别信息，这些信息都封装在回传的 TJSONArray 对象中
DescribeClass	回传 DataSnap 服务器中特定服务类别的叙述信息，这些信息都封装在回传的 TJSONObject 对象中
ListMethods	取得 DataSnap 服务器中所有的服务方法信息，这些信息都封装在回传的 TJSONArray 对象中
DescribeMethod	回传 DataSnap 服务器中特定服务方法的叙述信息，这些信息都封装在回传的 TJSONObject 对象中
NotifyCallback	通知回叫
NotifyObject	通知回叫物件

请读者比较上面 DSAdmin 和 TDSAdminClient 类别提供的服务，我们可以注意到这两个类别提供的服务方法并不完全一致，TDSAdminClient 类别并没有在客户端提供完整的 DSAdmin 服务方法，因此如果开发人员需要在 DataSnap 客户端呼叫 TDSAdminClient 类别没有提供的 DSAdmin 服务方法，那么开发人员仍然可以使用程序代码从客户端呼叫 DataSnap 服务器中的 DSAdmin 服务方法，在稍后的范例中我们会看到如何做到。

许多 DSAdmin 和 TDSAdminClient 类别提供的服务都以回传 TDBXReader 对象或是 JSON 相关类别对象来代表执行的结果，而在这些回传的对象中都是由其他一些相关的类别来定义的，例如 DSAdmin 的 GetServerClasses 方法回传由 TDBXReader 对象代表的所有伺服端服务类别，而在这个回传的 TDBXReader 对象中的信息则是由 TDSClassEntity 类别定义的。下面的表格整理了这些相关的类别以及它们的说明:

类别	说明
TDSClassEntity	定义伺服端服务类别的信息
TDSConnectionEntity	定义伺服端链接的信息
TDSMethodEntity	定义伺服端服务方法的信息
TDSPackageEntity	定义伺服端服务封包的信息
TDSProcedureEntity	定义伺服端服务预储程序的信息
TDSProcedureParametersEntity	定义伺服端服务预储程序的参数的信息

例如 TDSClassEntity 类别定义了如下的特性，开发人员可以藉由这些特性取得或是了解伺服端服务类别的相关信息：

特性	说明
PackageName	类别的封包名称
ServerClassName	伺服端服务类别名称
RoleName	类别存取角色
LifeCycle	伺服端服务类别的生命周期

例如 DSAdmin 的 GetServerClasses 方法回传 TDBXReader 对象，其中的数据就是 TDSClassEntity 类别的特性值，因此在呼叫 GetServerClasses 之后，开发人员可以藉由存取这个 TDBXReader 对象的 Value[1].AsString 而得到伺服类别的名称，也就是上面表格的 ServerClassName 特性值。

让我们看看如何使用 DSAdmin 类别在 DataSnap 客户端应用程序中取得 DataSnap 服务器中相关的服务信息，下图是范例应用程序执行的画面，在 DataSnap 客户端应用程序中我们可以取得 DataSnap 服务器中的服务类别以及服务方法，从下图中我们也可以看到非常有趣和有用的信息，那就是除了前面小节讨论的 3 个生命周期的服务类别之外，请读者注意的是，在得 DataSnap 服务器中的 DSAdmin 和 DSMetadat 这两个内建的服务类别也都是使用 Session 生命周期：

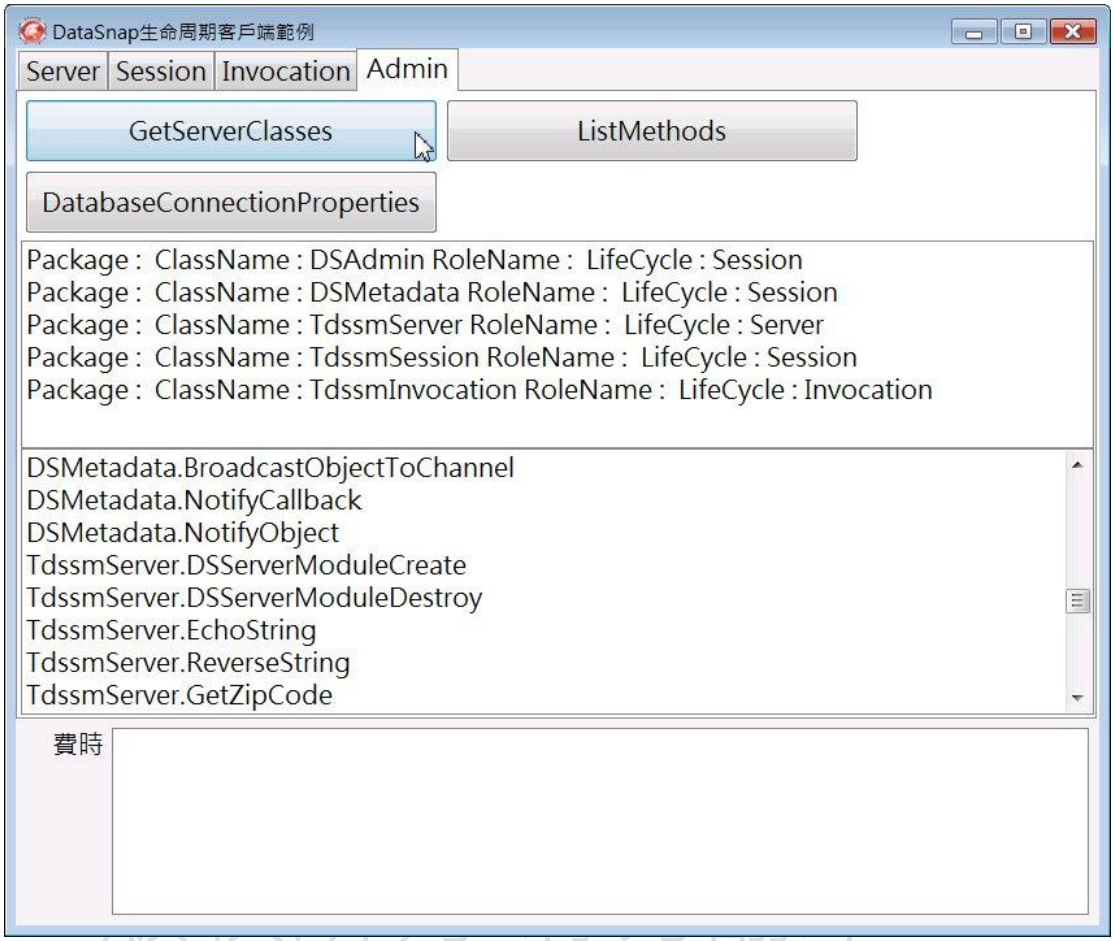


图 6-9 藉由 DSAdim 类别取得 DataSnap 服务器中的服务类别以及服务方法

上面画面的『GetServerClasses』按钮使用了下面的程序代码呼叫 DataSnap 服务器中的 DSAdmin 对象的 GetServerClasses 方法以取得伺服端所有的服务类别：

```
001   procedure TForm9.btnGetServerClassesClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
005     sClassData : TStringBuilder;
006   begin
007     if not ClientModule1.SQLConnection1.Connected then
008       ClientModule1.SQLConnection1.Connected := True;
009     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
010     sClassData := TStringBuilder.Create;
011     try
012       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
013       dsAdmin.Text := TDSAdminMethods.GetServerClasses;
```

```
014       dsAdmin.Prepare;
015       dsAdmin.ExecuteUpdate;
016       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
017       lbClasses.Clear;
018       while aReader.Next do
019       begin
020         sClassData.Append('Package : ' + aReader.Value[0].AsString);
021         sClassData.Append(' ClassName : ' + aReader.Value[1].AsString);
022         sClassData.Append(' RoleName : ' + aReader.Value[2].AsString);
023         sClassData.Append(' LifeCycle : ' + aReader.Value[3].AsString);
024         lbClasses.Items.Add(sClassData.ToString);
025         sClassData.Clear;
026       end;
027     finally
028       sClassData.Free;
029       dsAdmin.Free;
030     end;
031   end;
```


在上面的程序代码中展示了如何在 DataSnap 客户端呼叫位于 DataSnap 服务器中的DSAdmin对象，使用的方法就是先在009行建立TDBXCommand对象，在 012 行设定此 TDBXCommand 对象执行的命令是呼叫 DataSnap 服务器中的方法，013 行设定 TDBXCommand 对象执行的命令是 TDSAdminMethods.GetServerClasses，而 TDSAdminMethods.GetServerClasses 则是定义为 'DSAdmin.GetServerClasses'，也就是呼叫 DSAdmin 类别的 GetServerClasses 方法。

在016行取得回传的TDBXReader对象之后，其中的数据定义就是前面已经介绍的 TDSClassEntity 类别的特性值，因此 018 行之后就可以取得相关的数据并且显示在组件中了。

下面则是如何取得伺服端服务类别的程序代码，它和上面的程序代码非常的类似，同样是使用 TDBXCommand 呼叫 DSAdmin 类别的 GetServerMethods 方法：

```
001   procedure TForm9.btnListMethodsClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
```

```
005   begin
006     if not ClientModule1.SQLConnection1.Connected then
007       ClientModule1.SQLConnection1.Connected := True;
008     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
009     try
010       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
011       dsAdmin.Text := TDSAdminMethods.GetServerMethods;
012       dsAdmin.Prepare;
013       dsAdmin.ExecuteUpdate;
014       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
015       lbAdmin.Clear;
016       while aReader.Next do
017       begin
018         lbAdmin.Items.Add(aReader.Value[1].AsString + '.' +
aReader.Value[2].AsString);
019       end;
020     finally
021       dsAdmin.Free;
022     end;
end;
```

014 行取得的 TDBXReader 对象，其中的数据定义就是下表的 TDSMethodEntity 类别的特性值：

特性	说明
MethodAlias	服务方法的别名
ServerClassName	伺服端服务类别名称
ServerMethodName	服务方法名称
RoleName	类别存取角色

因此 016 行之后就可以藉由这些特性值取得服务类别名称和服务方法名称并且显示在画面中，当然我们也可以直接显示 TDSMethodEntity 的 MethodAlias 特性值，例如下图就是修改上面 018 行为：

```
    lbAdmin.Items.Add('MethodAlias : ' + aReader.Value[0].AsString);
```

之后的执行结果：

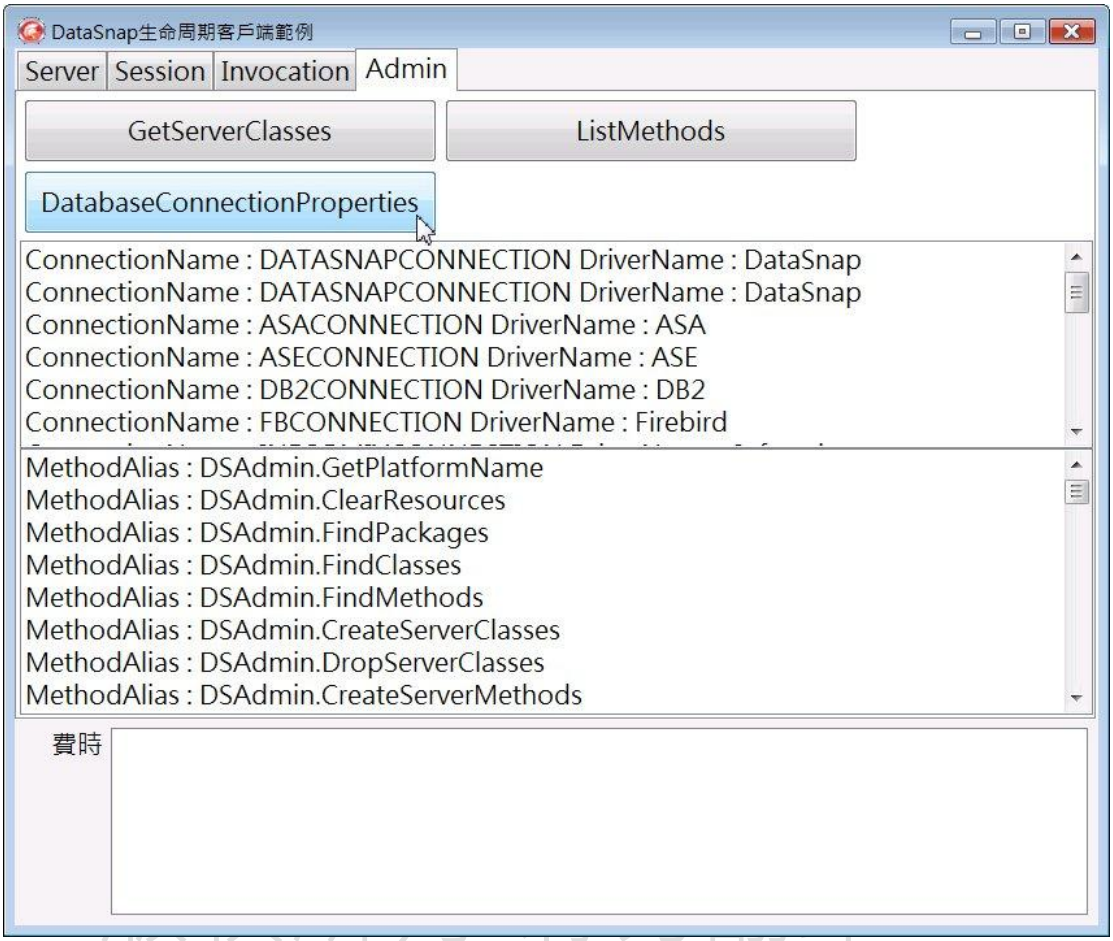


图 6-10 藉由 DSAdim 类别取得 DataSnap 服务器中的链接信息以及服务方法别名

同样的下面的程序代码呼叫 DSAdmin 类别的 GetDatabaseConnectionProperties 方法取得所有 DataSnap 伺服端的数据库链接信息：

```
001   procedure TForm9.btnDatabaseConnectionsClick(Sender: TObject);
002   var
003     dsAdmin : TDBXCommand;
004     aReader : TDBXReader;
005     sClassData : TStringBuilder;
006   begin
007     if not ClientModule1.SQLConnection1.Connected then
008       ClientModule1.SQLConnection1.Connected := True;
009     dsAdmin := ClientModule1.SQLConnection1.DBXConnection.CreateCommand;
010     sClassData := TStringBuilder.Create;
011     try
012       dsAdmin.CommandType := TDBXCommandTypes.DSServerMethod;
013       dsAdmin.Text := TDSAdminMethods.GetDatabaseConnectionProperties;
```

```
014       dsAdmin.Prepare;
015       dsAdmin.ExecuteUpdate;
016       aReader := dsAdmin.Parameters[0].Value.GetDBXReader(true);
017       lbClasses.Clear;
018       while aReader.Next do
019       begin
020         sClassData.Append('ConnectionName : ' + aReader.Value[0].AsString);
021         sClassData.Append(' DriverName : ' + aReader.Value[2].AsString);
022         lbClasses.Items.Add(sClassData.ToString);
023         sClassData.Clear;
024       end;
025     finally
026       sClassData.Free;
027       dsAdmin.Free;
028     end;
end;
```

这些数据库链接信息是由 TDSConnectionEntity 类别定义的：

特性	说明
ConnectionName	伺服端链接名称
ConnectionProperties	伺服端链接特性
DriverName	驱动程序名称
DriverProperties	驱动程序特性

在 DataSnap 客户端也可以使用 TDSAdminClient 类别来呼叫 DataSnap 服务器提供的管理服务，例如下面的程序代码直接藉由 DataSnap 客户端应用程序中的 TdssmServerClient 呼叫远程 DSAdmin 的 DescribeClass 方法，这是因为 TdssmServerClient 是由 TDSAdminClient 类别继承下来：

```
001   procedure TForm9.btnDescribeClassClick(Sender: TObject);
002   var
003     aCM: TClientModule1;
004     aServer : TdssmServerClient;
005     jsonObj : TJSONObject;
006   begin
007     aCM := TClientModule1.Create(Self);
008     try
009       aServer := aCM.dssmServerClient;
010       jsonObj := aServer.DescribeClass('TdssmServer');
```

```
011       lbClasses.Clear;
012       lbClasses.Items.Add(jsonObj.ToString);
013     finally
014       aCM.Free;
015     end;
016   end;
```

下面是上面程序代码的执行结果：

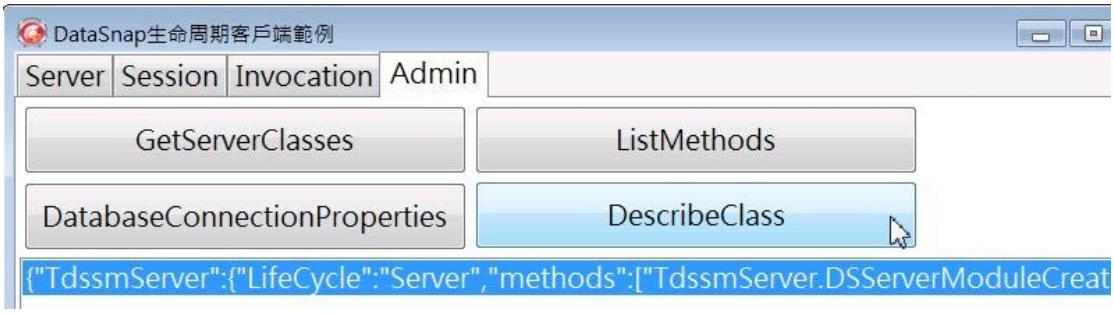


图 6-11 藉由 TDSAdminClient 类别呼叫 DataSnap 服务器中 DSAdmin 的 DescribeClass 方法

现在读者应该了解了如何使用 DSAdmin 和 TDSAdminClient 呼叫 DataSnap 服务器中的服务方法了。

6-3 结论

本章讨论了 DataSnap 服务器中服务类别的生命周期以及应该在什么情形下使用那一种生命周期，开发人员应该充分了解客户端的需求而选择使用最适合的生命周期。

最后本章讨论了如何在 DataSnap 伺服端以及客户端呼叫和使用 DataSnap 服务器提供的管理服务，开发人员可以分别藉由 DSAdmin 和 TDSAdminClient 这两个类别来存取 DataSnap 服务器的管理服务功能。

第7章 开发移动式DataSnap客户端

手机和移动式设备的开发在现今似乎变得愈来愈重要，因此许多应用系统都需要能够把手机和移动式设备整合到现有的系统之中做为新的客户端，在 RAD Studio 10.3 中提供了 Mobile Connector 的功能，允许开发人员开发 iPhone，Android 和黑莓机的 DataSnap 客户端，让主流手机的使用者也可以藉由手机连结到 DataSnap 服务器以存取 DataSnap 服务器提供的服务，本章的内容就在说明如何藉由 DataSnap Mobile Connector 的功能开发手机的 DataSnap 客户端应用程序。

7-1 DataSnap Mobile Connector

10.3 推出 DataSnap Mobile Connector 技术的目的是为了让手机客户端能够非常容易的连结到 Windows 平台的 DataSnap 服务器取得服务，如此一来就能够让原本的 Midas 分布式系统或是最新的 DataSnap 分布式系统和移动式客户端整合在一起。

目前由于不同的手机客户端必须使用不同的程序语言和技术来开发，因此开发人员如果需要整合数个不同的手机客户端和 Midas/DataSnap 分布式系统，那么将会是非常辛苦的工作，而 DataSnap Mobile Connector 正好解决了这个问题，因为 DataSnap Mobile Connector 藉由可自动产生不同手机客户端的程序代码并且统一使用 JSON/REST 的技术让不同的手机客户端链接到 Midas/DataSnap 分布式系统。

例如对于 iOS 客户端，DataSnap Mobile Connector 可自动产生 Object C 客户端程序代码，对于 Android 和 BlackBerry 客户端，DataSnap Mobile Connector1 可自动产生 Java 客户端程序代码，接着开发人员就可以使用这些

客户端程序代码使用 JSON/REST 存取 Midas/DataSnap 伺服端服务。此外由 DataSnap Mobile Connector 产生的不同客户端的程序代码都包含了一样的 JSON/REST 类别函式库，因此所有不同手机客户端都可使用 DataSnap Mobile Connector 自动产生的 JSON/REST 类别函式库。

下图说明了使用 DataSnap Mobile Connector 技术之后整合 Midas/DataSnap 分布式系统和各种不同移动式客户端的架构，读者可以注意到在这新的分布式架构中都使用 JSON/REST 来存取服务，不过在企业内部如果为了效率的考虑，那么在企业内部仍然可以使用 TCP/IP 来存取 Midas/DataSnap 服务。

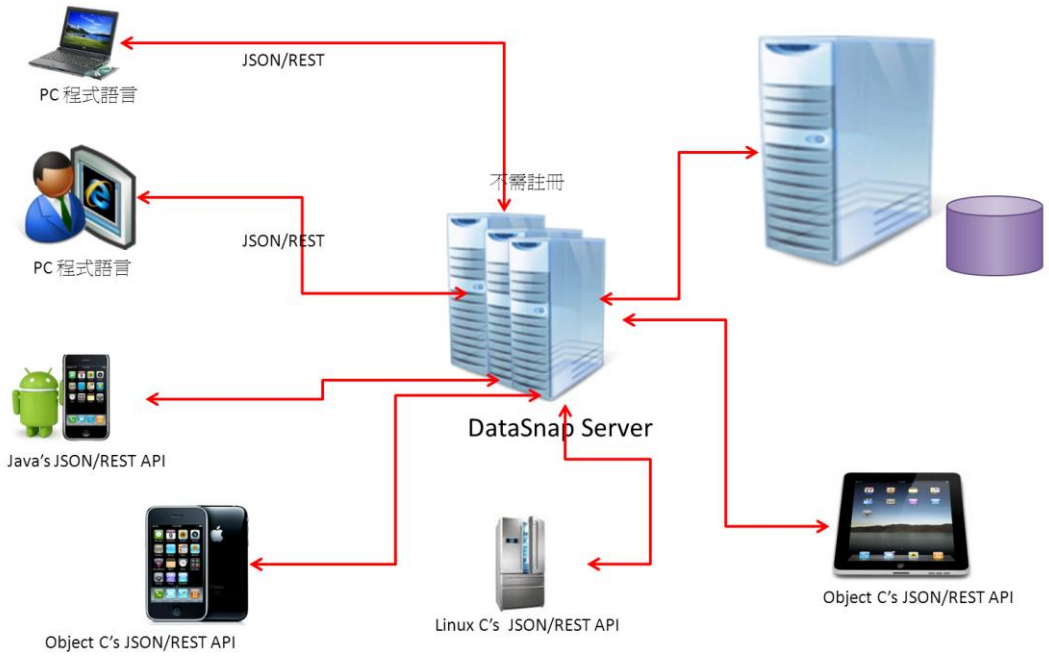


图 1 整合分布式和移动客户端架构

例如下图展示如果没有 DataSnap Mobile Connector 技术的话要如何整合 Midas/DataSnap 分布式系统和手机客户端，我们可以看到在不同的手机客户端需要使用不同的程序语言以及不同的 JSON/REST 函式库:

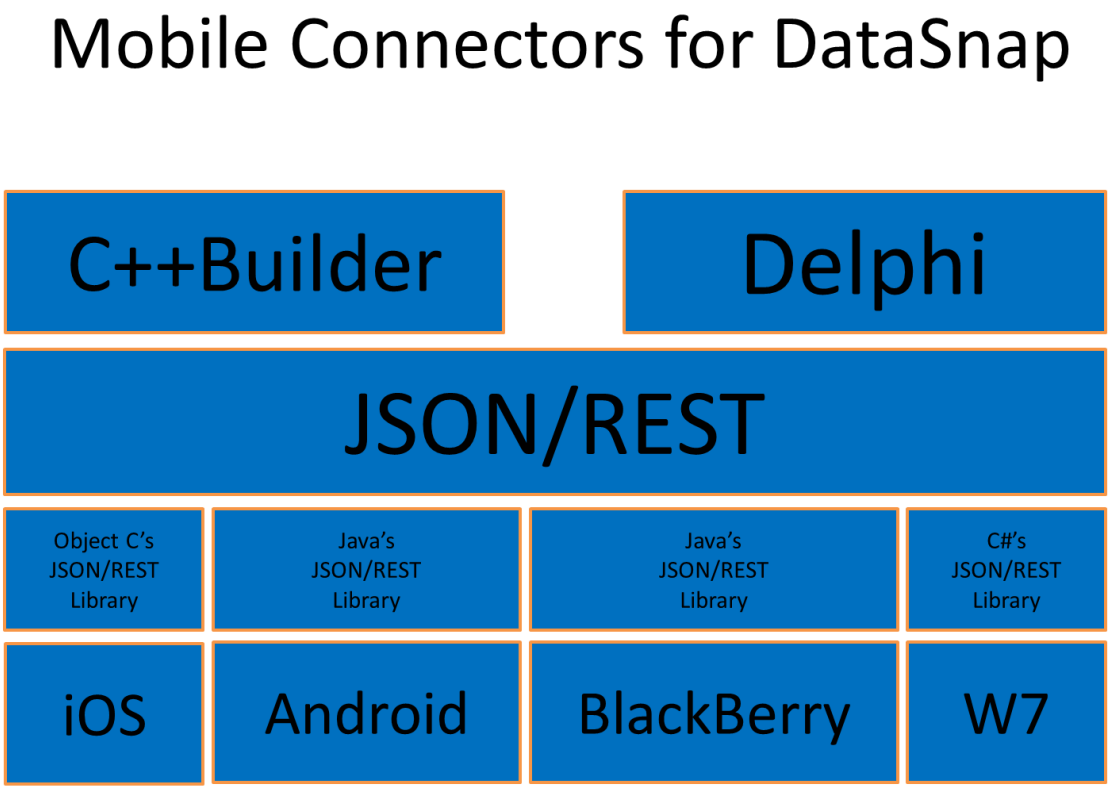


图 2 不使用 DataSnap Mobile Connector 整合分布式和移动客户端架构

而下图则展示了使用 DataSnap Mobile Connector 的好处，除了 DataSnap Mobile Connector 可以自动产生手机客户端程序代码之外，各种不同的手机客户端都可以使用相同的 JSON/REST 函式库存取 Midas/DataSnap 分布式系统:

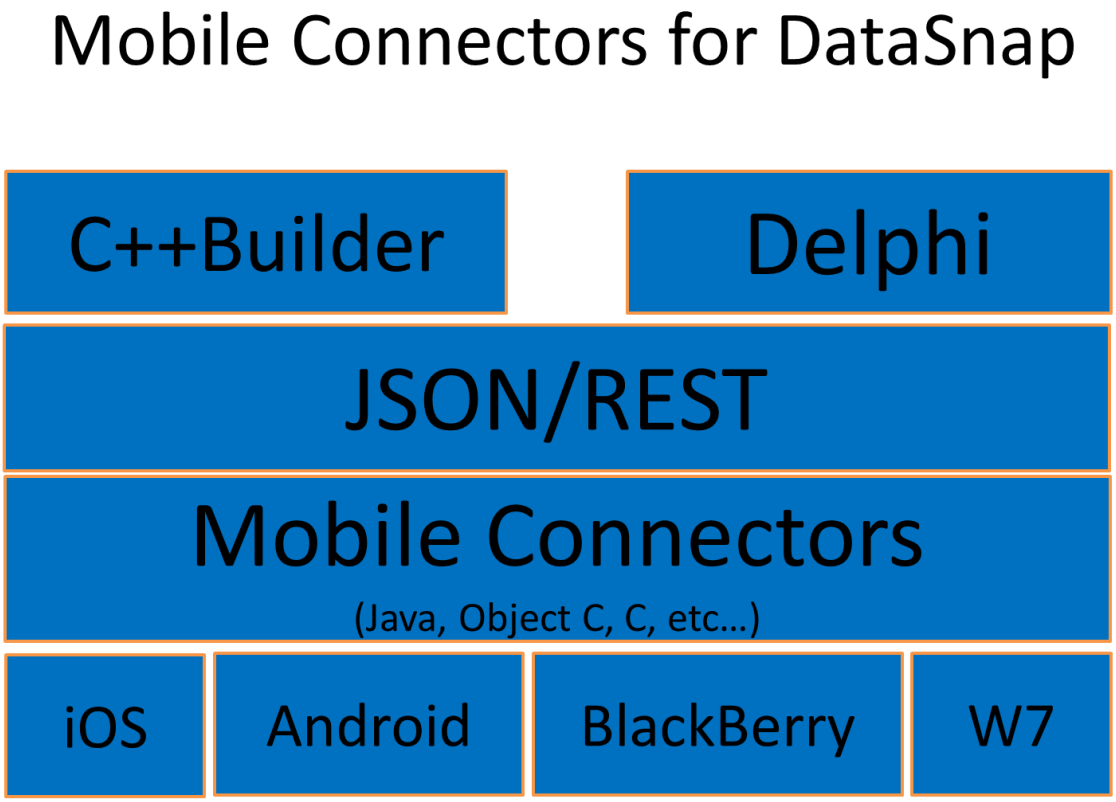


图 3 使用 DataSnap Mobile Connector 整合分布式和移动客户端架构

在 XE2 Update 4 中 Delphi 进一步的强化了 DataSnap 10.3 的功能，对于 iOS 客户端 DataSnap 10.3 直接提供了 Free Pascal 的客户端程序代码和 JSON/REST 函式库，因此对于 iOS 客户端在 Update 4 之后开发人员可以直接使用最熟悉的 Pascal 程序代码来整合 iOS 和 Midas/DataSnap 分布式系统而无需再使用 Object C 了，如下图所示:

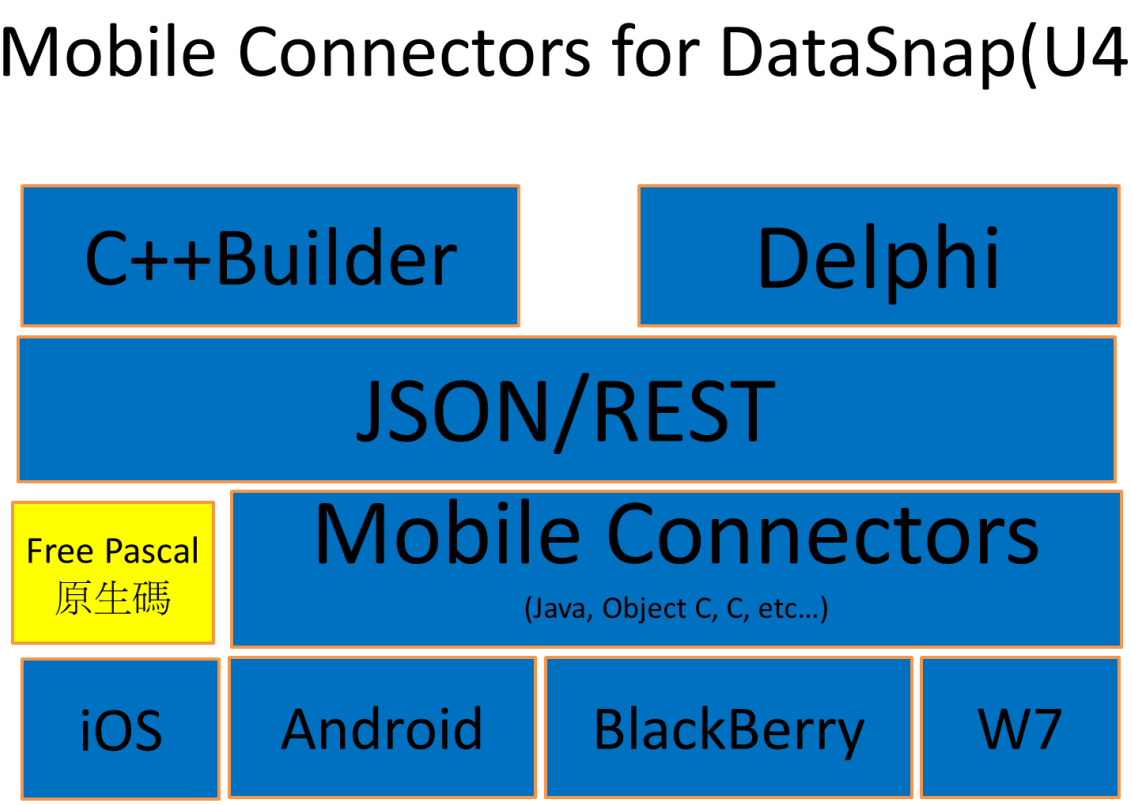


图 4 Update 4 可直接使用 Pascal 程序代码整合分布式和移动客户端架构

当然从 Update 4 这个发展迹象来看，DataSnap Mobile Connector 会持续的进化到允许开发人员直接使用 Delphi 程序代码来整合分布式和移动客户端，如下图所示：

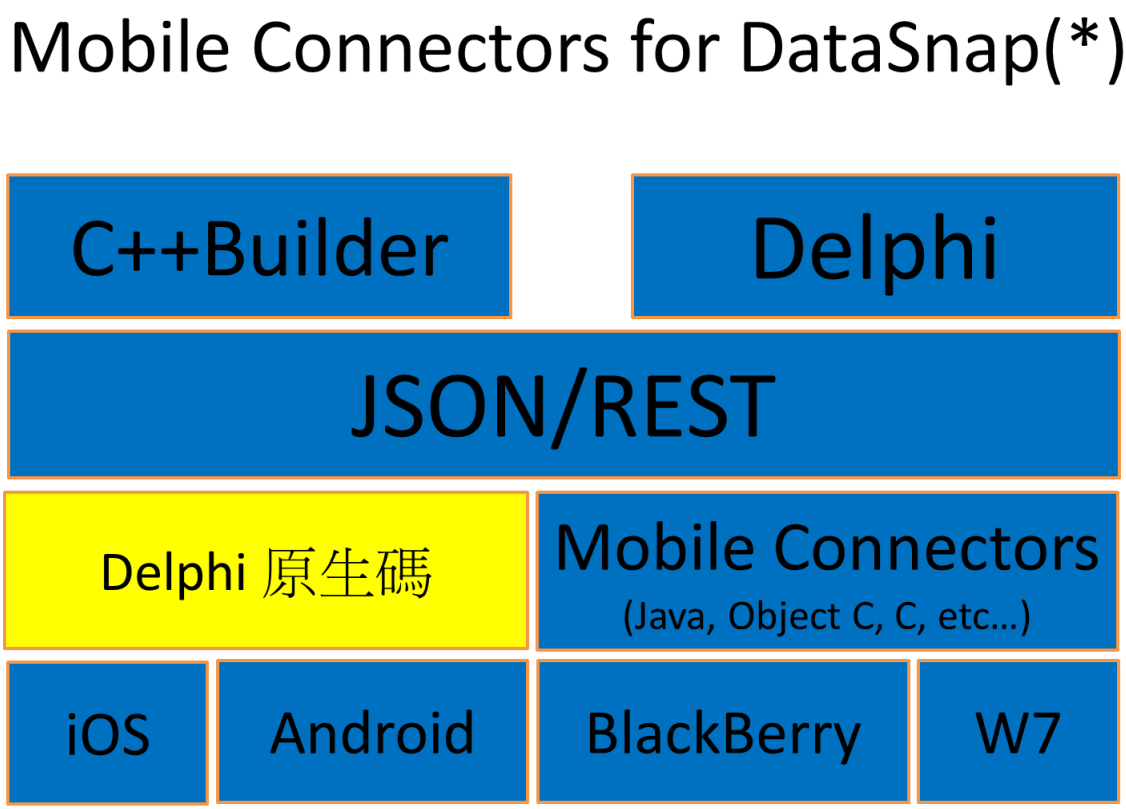


图 5 DataSnap Mobile Connector 会持续的进化

了解了 DataSnap Mobile Connector 工作原理和架构之后接下来当然就是说明如何使用 DataSnap Mobile Connector 了。

7-2 开发 Android 客户端

要让 Android 客户端连结到 DataSnap 服务器，开发人员必须执行下列的步骤:

- 从 DataSnap 服务器取得 Mobile Connector 的 Java 客户端程序代码，这份程序代码不但可以让 Android 的 Java 程序代码连结到 DataSnap 服务器，更重要的是其中包含了所有 DataSnap 服务器中的服务方法，可让 Android 的 Java 客户端程序代码直接呼叫 DataSnap 服务器。
- 把 Mobile Connector 的 Java 客户端程序代码汇入到 Eclipse For Android 中，再使用 Java 呼叫 DataSnap 服务器。

接下来就让我们使用一个实际的范例来说明如何使用 Mobile Connector。

7-2-1 建立 DataSnap 服务器

要让手机客户端能够连结到 DataSnap 服务器，在建立 DataSnap 服务器时必须加入支持 Mobile Connectors 的功能，因此先让我们建立一个新的范例 DataSnap REST 应用程序，如下所示：

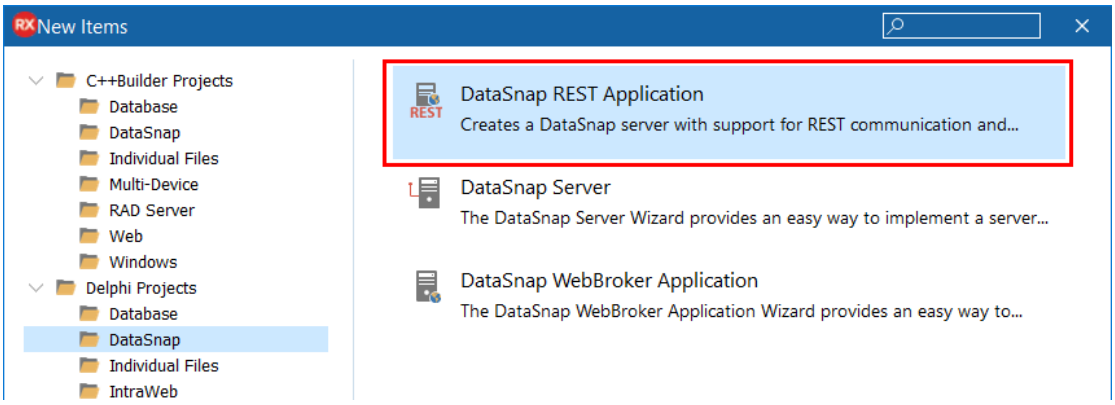


图 6 建立范例 DataSnap 服务器

接着在下一步中勾选支持 Mobile Connectors 功能，如下所示：

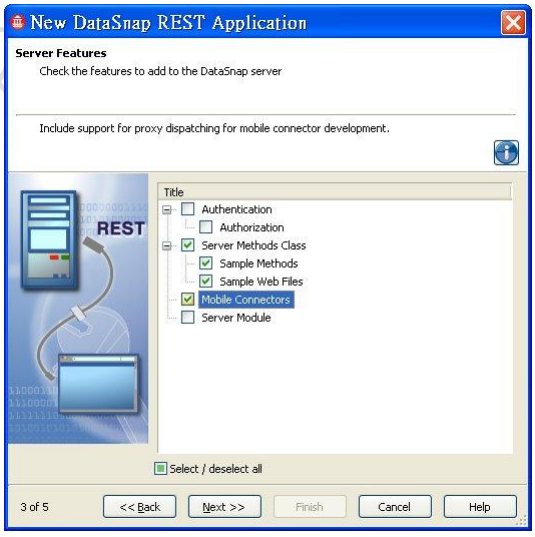


图 7 范例 DataSnap 服务器选择支持 Mobile Connectors 技术

请接着继续完成建立范例 DataSnap 服务器的其他步骤，在完成建立范例 DataSnap 服务器之后，如果开启 WebModule 程序单元，那么可以看到如下的组件。由于在前面的步骤中勾选支持 Mobile Connectors 技术，因此在此程序单元中会加入 DSProxyDispatcher 组件，这个组件可以让客户端藉由使用特定的 URL 来产生支持特定手机客户端的 Mobile Connectors 原始程序：

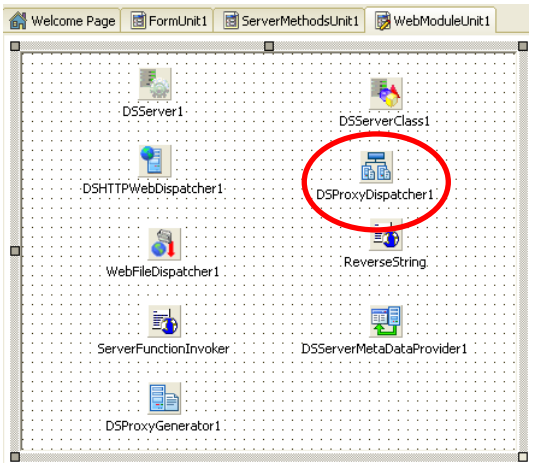


图 8 范例 DataSnap 服务器选择支持 Mobile Connectors 技术

如果现在读者开启 WebModule 程序单元的原始程序文件，会看到它加入了支持 Android(Datasnap.DSProxyJavaAndroid)，iOS(Datasnap.DSProxyObjectiveCiOS)等手机客户端的程序单元：

```
uses
  System.SysUtils, System.Classes, Web.HTTPApp, Datasnap.DSHTTPCommon,
  Datasnap.DSHTTPWebBroker, Datasnap.DSServer,
  Web.WebFileDispatcher, Web.HTTPProd,
  DSAuth,
  Datasnap.DSProxyDispatcher, Datasnap.DSProxyJavaAndroid,
  Datasnap.DSProxyJavaBlackBerry, Datasnap.DSProxyObjectiveCiOS,
  Datasnap.DSProxyCsharpSilverlight,
  Datasnap.DSProxyFreePascal_iOS,
  Datasnap.DSProxyJavaScript, IndyPeerImpl, Datasnap.DSClientMetadata,
  Datasnap.DSCommonServer;
```

现在如果我们编译而且执行范例 DataSnap 服务器，那么就可以使用浏览器藉由特定的 URL 来取得支持特定手机客户端的 Mobile Connectors 程序代码，接着就可以使用这些支持特定手机客户端的 Mobile Connectors 程序代码来连结并且存取 DataSnap 服务器中的服务。例如下图是在浏览器中使用：

```
http://localhost:8080/proxy/java_android.zip
```

这个 URL 取得 Android 客户端的 Mobile Connectors 程序代码：

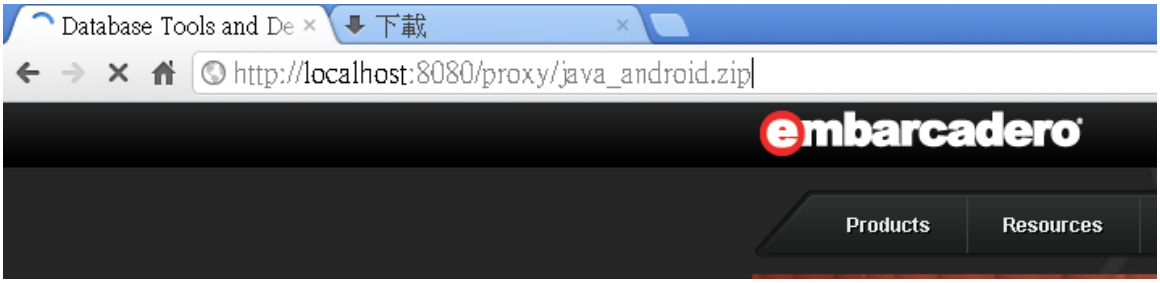


图 9 在浏览器中使用特定手机的 URL 取得特定手机的 Mobile Connectors 客户端程序代码

在浏览器中使用了上述的 URL 之后，范例 DataSnap 服务器就会传送 Android 客户端的 Mobile Connectors 程序代码(以 ZIP 文件压缩)到浏览器中，让浏览器下载。

下面的表格整理了 Mobile Connectors 技术对应不同手机客户端必须使用的 URL:

手机客户端	URL
Android	http://服务器地址:服务器通信埠/proxy/java_android.zip
iOS 4	http://服务器地址:服务器通信埠/proxy/java_android.zip
BlackBerry	http://服务器地址:服务器通信埠/proxy/java_android.zip
Win 7 Mobile	http://服务器地址:服务器通信埠/proxy/java_android.zip
iOS 5	http://服务器地址:服务器通信埠/proxy/freepascal_ios50.zip

如果我们解压缩下载的 java_android.zip 就可以看到其中包含的档案如下，这些档案都是 Java 原始程序，可以让开发人员在 Eclipse For Android 使用以连结到 DataSnap 服务器。

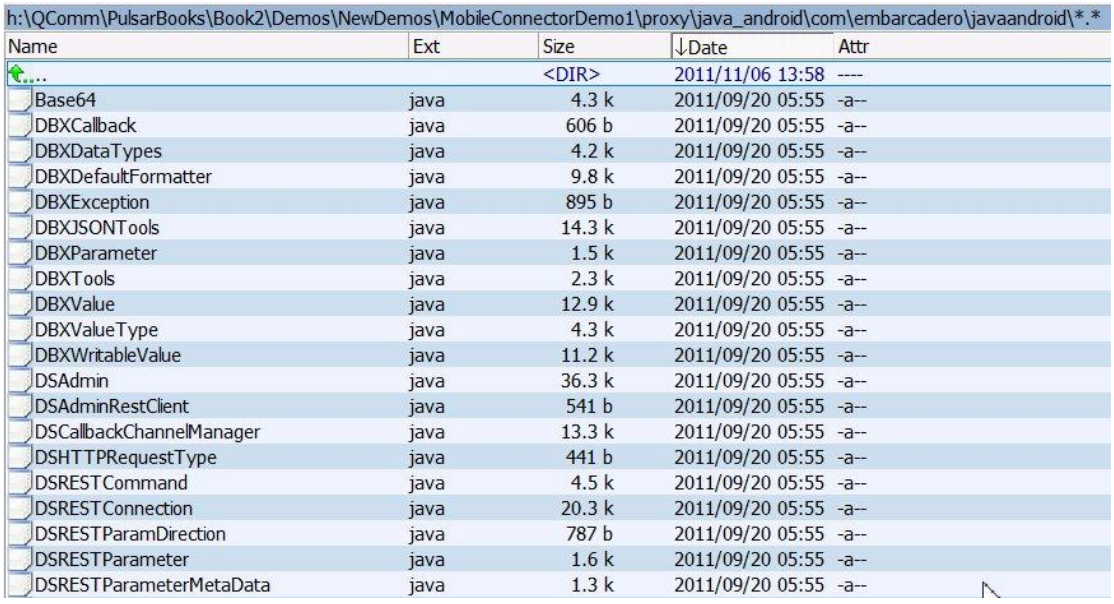


图 10 java_android.zip 中包含的 Mobile Connectors 原始程序

取得了 Android 客户端的 Mobile Connectors 原始程序之后，接下来就可以使用它来开发 Android 客户端的 App 了。

7-2-2 建立 Android 客户端

启动 Eclipse For Android 并且建立 Android 项目:

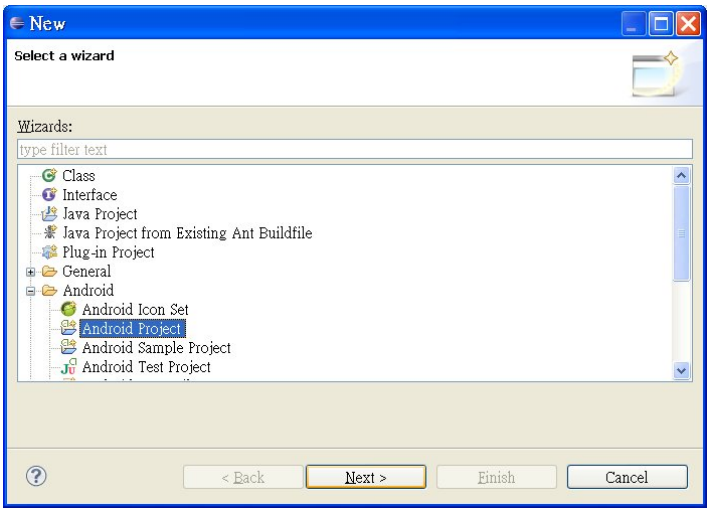


图 11 使用 Eclipse For Android 建立 Android 项目

接着选择使用 Android 2.1 SDK:

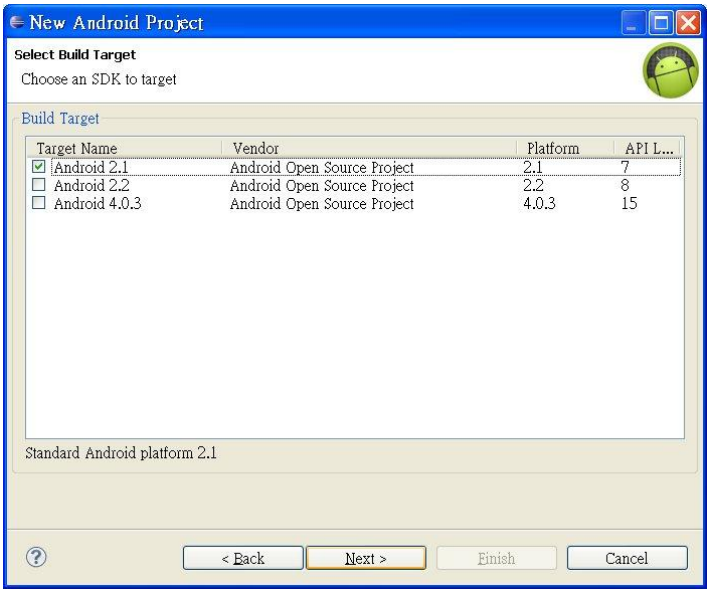


图 12 选择使用 Android 2.1 SDK

在建立完成 Android 项目后，请汇入 java_android.zip 到项目中，并且汇入到项目的 src 目录中:

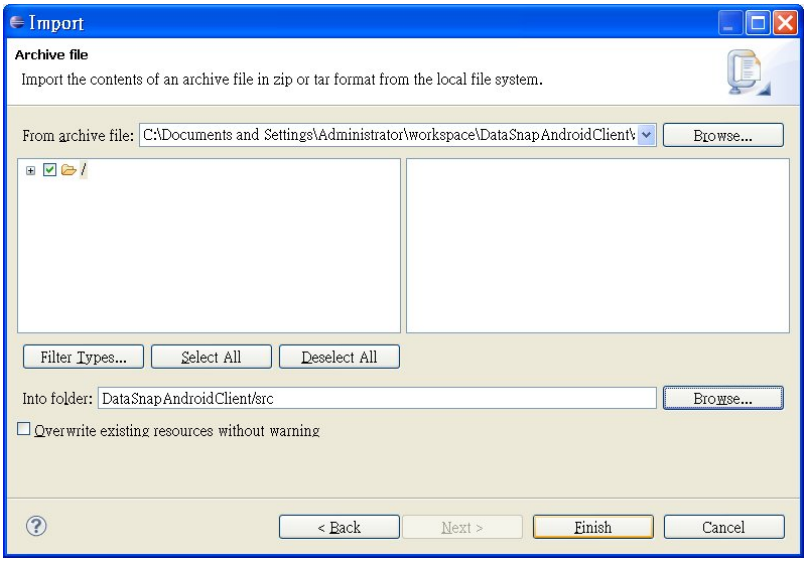


图 13 使用输入功能把 java_android.zip 档案汇入到 Android 项目的 src 目录中

在汇入完成之后如果开启 src 节点就可以看到 com.embarcadero.javaandroid 封包出现在 src 节点下:

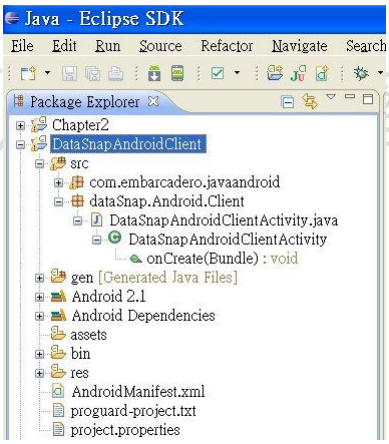


图 14 汇入 java_android.zip 档之后 dataSnap.Android.Client 出现在 src 目录中

为了让 Android 能够连结 DataSnap 服务器，我们必须开启 INTERNET 访问权限，请编辑项目中的 Manifest.xml 档案，加入 INTERNET 存取的用户权力，如下所示:

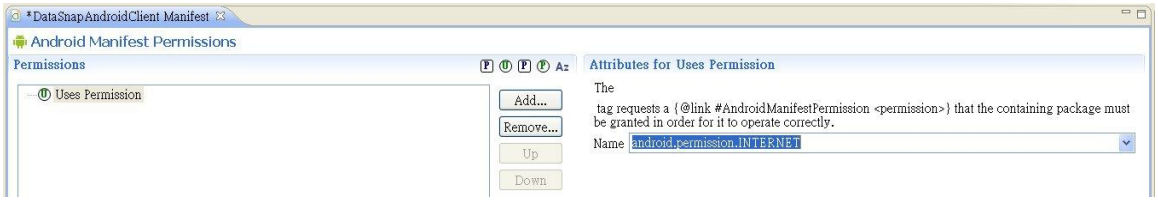


图 15 修改 Android 项目的 Mainfest.xml 檔，加入 INTERNET 存取的用户权力

最后让我们修改用户接口，请使用鼠标双击项目中 res/layout 节点之下的 main.xml 文件，此时 Eclipse 便会显示可视化设计接口，请在主窗体中加入一个 Label，2 个 EditText 和一个 Button 组件，修改后的 main.xml 如下所示:

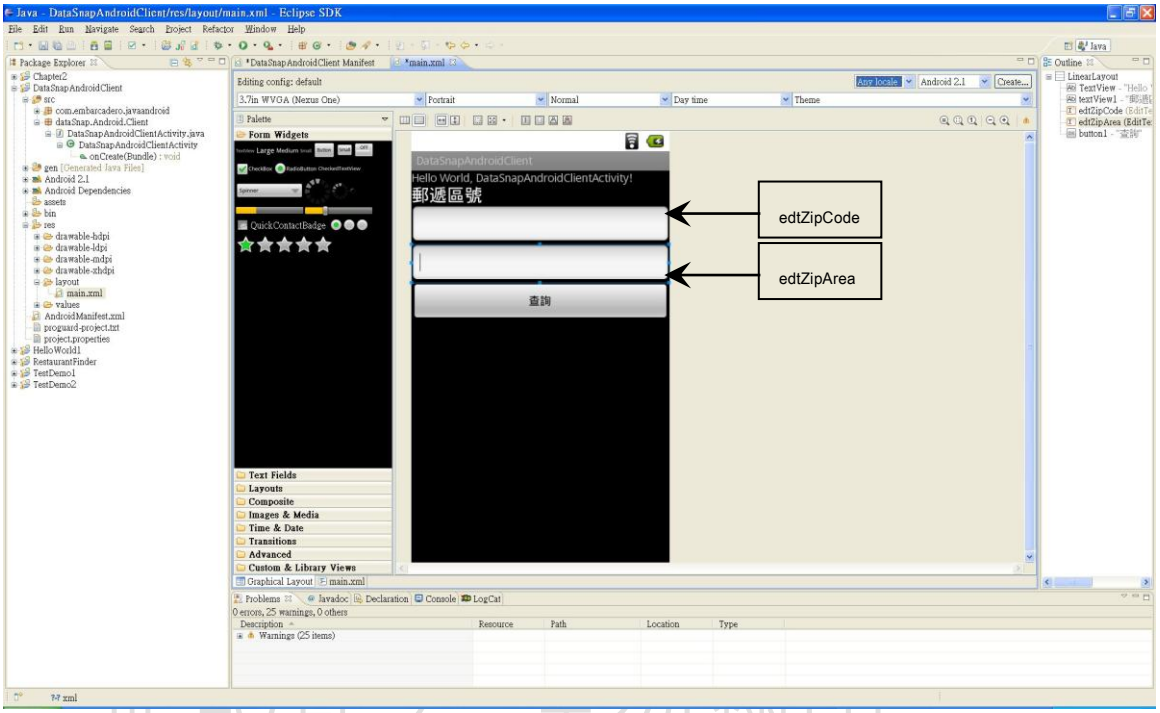


图 16 修改 Main.xml 设计用户接口

加入的第 1 个 EditText 是做为使用者输入查询的邮政编码之用，而第 2 个 EditText 是做为显示查询的结果，因此请右擊第 1 个 EditText，设定它的 ID 为 edtZipCode，右擊第 2 个 EditText，设定它的 ID 为 edtZipArea，最后右擊 Button，设定它的 Text 为“查询”，它的 ID 为 btnQuery。

设计完需要的接口之后就可以开始撰写实作程序代码了，请双擊 DataSnapAndroidClientActivity.java 开启源代码档，并且实作如下的程序代码：

```
001   package dataSnap.Android.Client;
002
003   import com.embarcadero.javaandroid.DSProxy.TServerMethods1;
004   import com.embarcadero.javaandroid.DSRESTConnection;
005
006   import android.app.Activity;
007   import android.os.Bundle;
008   import android.view.View;
009   import android.view.View.OnClickListener;
```

```
010   import android.widget.Button;
011   import android.widget.EditText;
012
013   public class DataSnapAndroidClientActivity extends Activity  implements
OnClickListener{
014       Button btnQuery;
015       EditText edtZipCode;
016       EditText edtZipArea;
017       /** Called when the activity is first created. */
018       @Override
019       public void onCreate(Bundle savedInstanceState) {
020          super.onCreate(savedInstanceState);
021          setContentView(R.layout.main);
022
023          btnQuery = (Button) findViewById(R.id.btnQuery);
024          edtZipCode = (EditText) findViewById(R.id.edtZipCode);
025          edtZipArea = (EditText) findViewById(R.id.edtZipArea);
026
027          btnQuery.setOnClickListener(this);
028       }
029       @Override
030       public void onClick(View arg0) {
031            // TODO Auto-generated method stub
032            DSRESTConnection conn = new DSRESTConnection();
033            conn.setHost("172.16.137.136");
034            conn.setPort(8080);
035            conn.setProtocol("http");
036            TServerMethods1 proxy = new TServerMethods1(conn);
037            try
038            {
039
     edtZipArea.setText(proxy.GetDistrictFromZipCode(edtZipCode.getText().toString()))
;
040            }
041            catch (Exception ex)
042            {
043                 ex.printStackTrace();
044            }
```

```
045         }
046     }
```

在前面汇入的java_android.zip中有2个最重要的java档是让开发人员使用REST连结DataSnap服务器，并且可用java程序代码呼叫DataSnap服务器中输出的服务方法。其中的DSRESTConnection.java檔中提供了DSRESTConnection类别让Android的java客户端可藉由REST技术连结DataSnap服务器，而其中的DSProxy.java则封装了DataSnap服务器中输出的服务方法。

因此在上面的程序代码中032行先建立DSRESTConnection对象，033行设定DataSnap服务器的地址，034行设定DataSnap服务器使用的通讯端口，035行再设定使用的连结通讯协议，036行就能够建立去装在DSProxy.java中的TServerMethods1类别对象并且呼叫DataSnap服务器中输出的服务方法了。

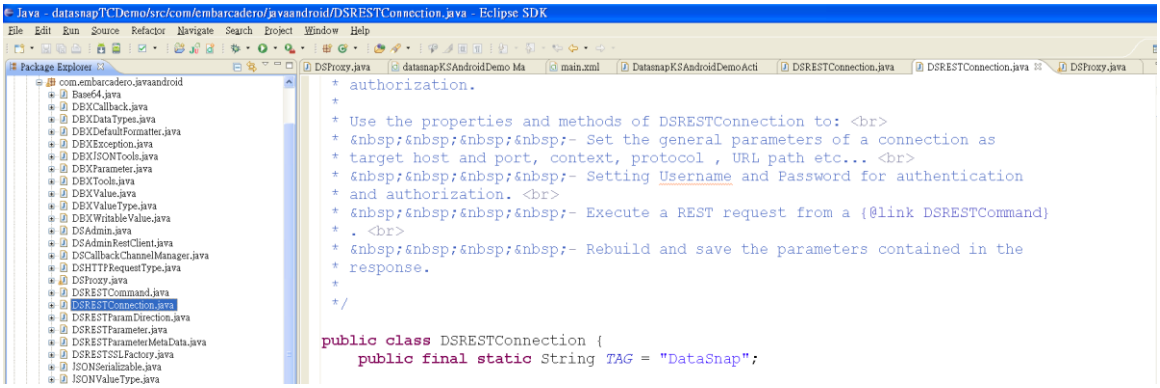


图 17 Mobile Connectors 自动产生的 DSRESTConnection.java 可使用 REST 连结 DataSnap 服务器

接着为范例项目建立一个执行时期组态设定，如下所示：

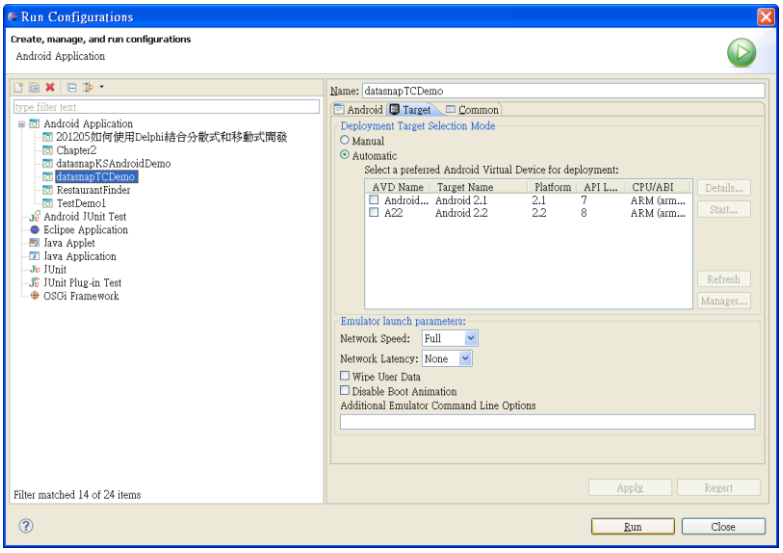


图 18 为范例项目建立执行时期组态设定

最后使用建立的执行时期组态设定执行范例项目，Eclipse 便会启动 Android Simulator 并且加载执行范例项目，请在范例程序中输入一个台北的邮政编码并且点选按钮查询，便会看到 Android 客户端的确呼叫 DataSnap 服务器进行查询，最后查询结果就传递回 Android 客户端，并且显示在用户接口中，如下所示：

图 19 在 Simulator 中执行 Android App 成功的连结 DataSnap 服务器并且呼叫其中的服务

藉由 Mobile Connectors 我们果然可以非常容易的整合 Android 客户端到 DataSnap 的分布式系统中，接下来让我们继续说明如何整合 iOS 的客户端。

7-3 开发 iOS 客户端

10.3 提供了原生 iOS 的开发能力和环境，比 XE2 提供的 iOS 开发功能进步太多了。要在 10.3 中开发 DataSnap iOS 客户端 App，请先在 IDE 中建立一个 Multi-Device Application 项目，如下所示：

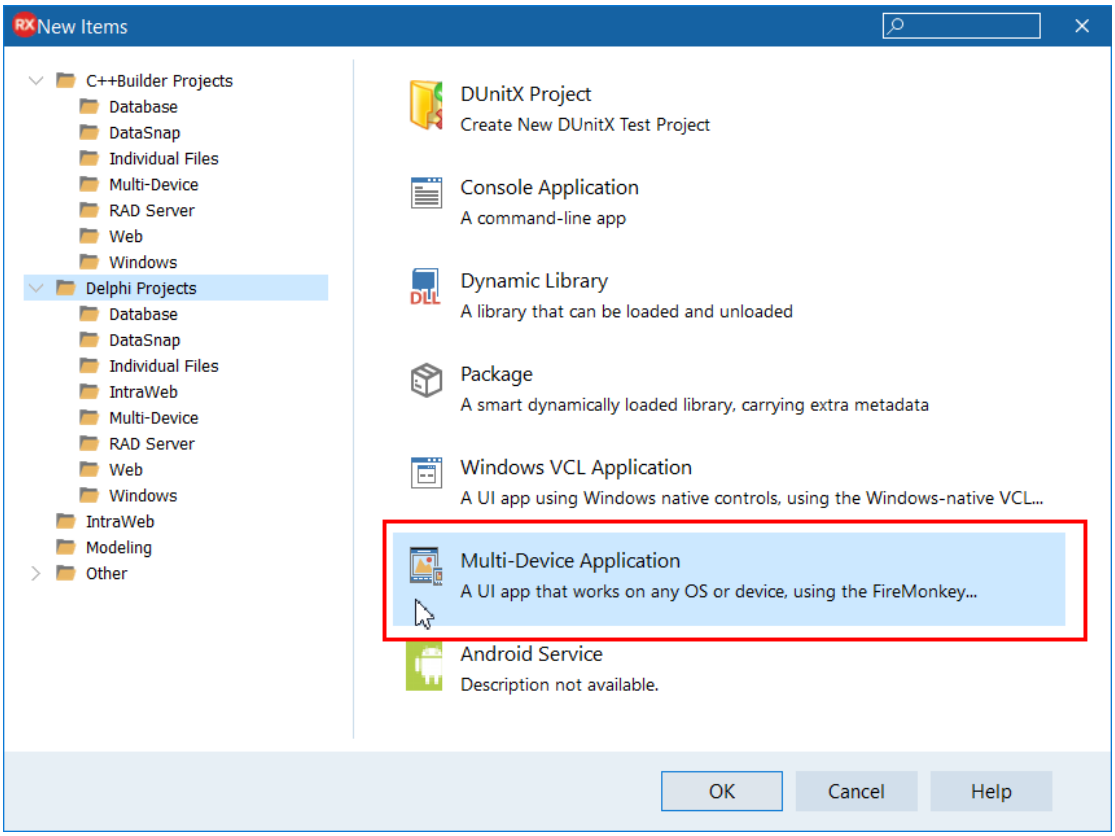


图 20 建立 iOS 项目

接着在众多的项目样版中让我们选择最简单的 Blank Application 样板，如下所示：

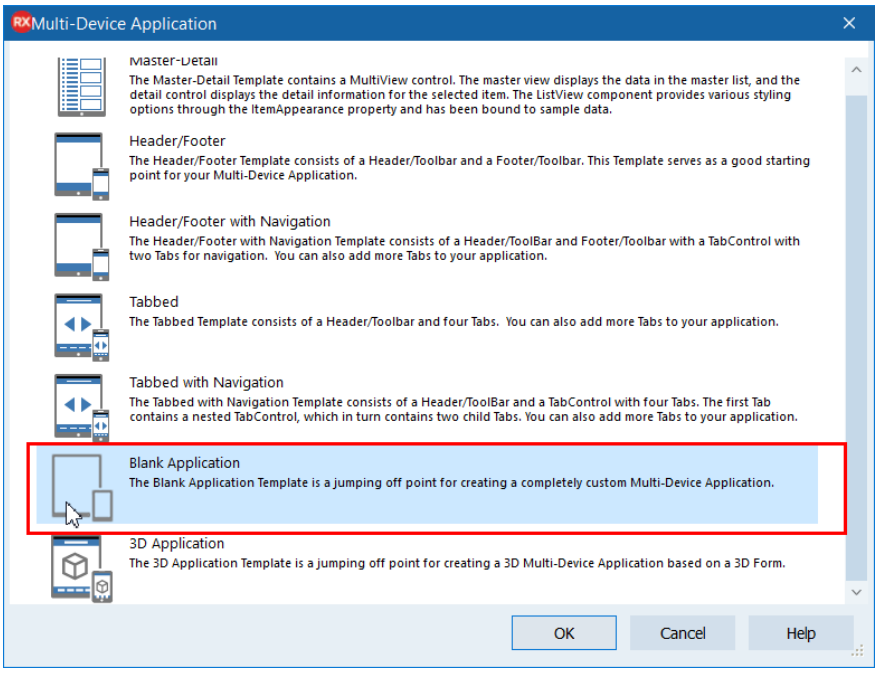


图 21 选择使用 Blank Application 样板

接着在此 iOS App 的主窗体中使用 TListBox，TLabel，TButton 和 TEdit 等组件设计如下的用户接口，最后再放入 TSQLConnection 组件准备连接范例 DataSnap 服务器，如下所示：

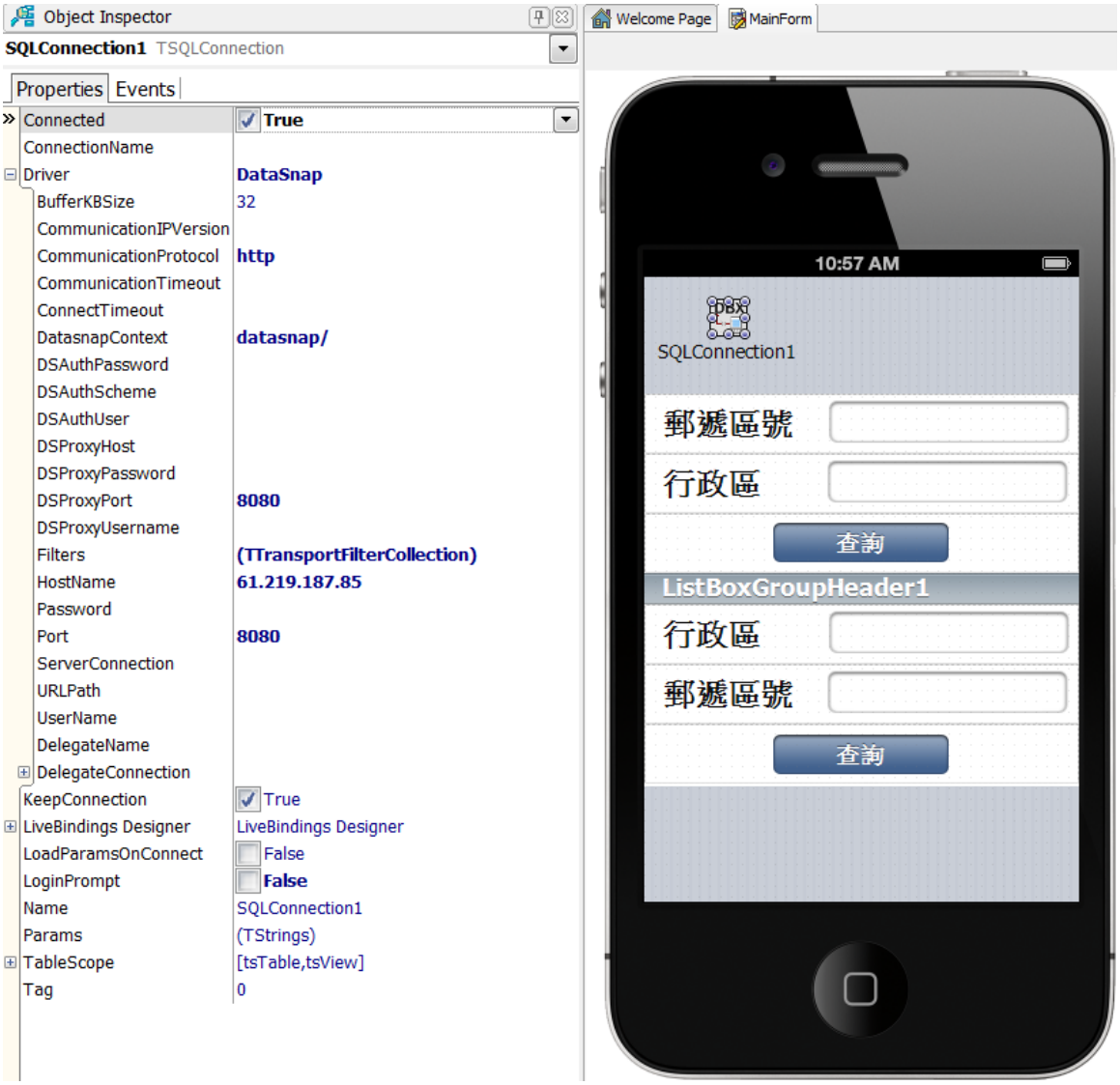


图 22 设计主接口,并且使用 TSQLConnection 组件链接范例 DataSnap 服务器

在对象查看器中设定 TSQLConnection 组件的特性值:

特性名称	特性值
Driver	DataSnap
HostName	范例 DataSnap 服务器 IP 地址
Port	8080
CommunicationProtocol	http
LoginPrompt	False

我们需要藉由TSQLConnection组件设定范例DataSnap服务器的所在地并且使用 DataSnap 驱动程序来连结。

在设定了上述的特性值之后，请再设定 TSQLConnection 组件的 Active 特性值为 True 让 TSQLConnection 组件真正链接到范例 DataSnap 服务器。

接着使用鼠标右击 TSQLConnection 组件，从突显示选单中选择”Generate DataSnap client classes”选项以便在 iOS 项目中产生可呼叫范例 DataSnap 服务器服务的客户端 Delphi 程序代码，如下所示：

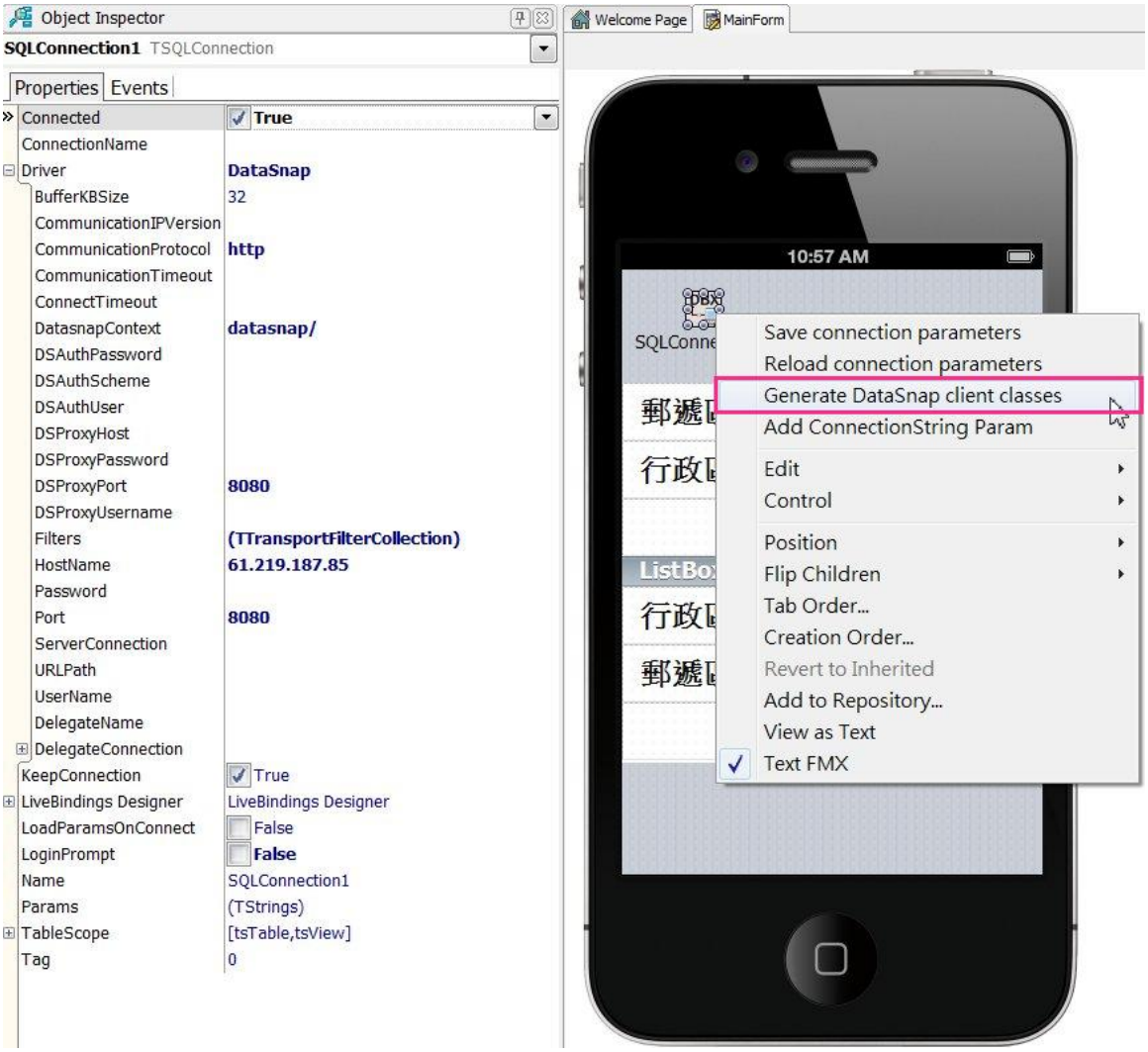


图 23 藉由 TSQLConnection 组件自动产生客户端链接程序码

点选之后 Delphi 便会产生一个新的程序单元，请储存此程序单元。在此程序单元中有一个名为 TServerMethods1Client 的类别，我们在客户端便可以藉由这个类别来呼叫范例 DataSnap 服务器中的服务，例如在下面的 TServerMethods1Client 的类别宣告中我们就可以看到我们需要呼叫的 GetDistrictFromZipCode 和 GetZipCode 这 2 个方法：

```
 TServerMethods1Client = class(TDSAdminClient)
 private
   FDSServerModuleDestroyCommand: TDBXCommand;
   FDSServerModuleCreateCommand: TDBXCommand;
   FEchoStringCommand: TDBXCommand;
   FReverseStringCommand: TDBXCommand;
   FGetDistrictFromZipCodeCommand: TDBXCommand;
   FGetZipCodeCommand: TDBXCommand;
 public
   constructor Create(ADBXConnection: TDBXConnection); overload;
   constructor Create(ADBXConnection: TDBXConnection; AInstanceOwner: Boolean); overload;
   destructor Destroy; override;
   procedure DSServerModuleDestroy(Sender: TObject);
   procedure DSServerModuleCreate(Sender: TObject);
   function EchoString(Value: string): string;
   function ReverseString(Value: string): string;
   function GetDistrictFromZipCode(sZipCode: string): string;
   function GetZipCode(SDistrict: string): Integer;
 end;
```

有了 TServerMethods1Client 类似之后就非常容易了，请在 iOS 主窗体中使用刚才储存的 TServerMethods1Client 类别程序单元，然后在主窗体的 2 个按钮的 OnClick 事件处理函式中撰写如下的程序代码:

```
procedure TForm50.btnQDistrictClick(Sender: TObject);
var
 ss : TServerMethods1Client;
begin
 ss := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
 try
   edtQDistrict.Text := ss.GetDistrictFromZipCode(edtZipDistrict.Text);
 finally
   ss.Free;
 end;
end;

procedure TForm50.btnQZipClick(Sender: TObject);
var
 ss : TServerMethods1Client;
```

```
begin
  ss := TServerMethods1Client.Create(SQLConnection1.DBXConnection);
  try
    edtQZip.Text := ss.GetZipCode(edtDistrictZip.Text).ToString;
  finally
    ss.Free;
  end;
end;
```

上面的程序代码只是简单的建立TServerMethods1Client对象然后分别呼叫 GetDistrictFromZipCode 和 GetZipCode 这 2 个方法。

请编译此范例 iOS App 并且分别在 iOS Simulator 和 iOS 实机中执行，您应该就可以像下面的画面一样成功的使用 iOS 呼叫远程的范例 DataSnap 服务器了。

图 24 范例 iOS App 成功的在 iOS Simulator 中呼叫远程的范例 DataSnap 服务器

图 25 范例 iOS App 成功的在 iOS 5 实机中呼叫远程的范例 DataSnap 服务器

7-4 结论

藉由 Mobile Connectors 技术，开发人员能够很快速，方便的开发 4 种移动平台的客户端并且整合到 DataSnap 分布式架构中，很快的开发人员就能够使用 Delphi 和 FireMonkey 开发 Android 平台任何种类的 App。

如是要开发 iOS 的客户端那么 10.3 已经提供原生的 iOS 开发能力，开发 DataSnap 的 iOS 客户端 App 就像开发原生的 Windows 客户端一样的方便了。

第8章 DataSnap监督功能

Delphi 10.3 在 DataSnap 方面也增加了一些功能，执行效率也比 XE 版进行了一些优化的改善。10.3 的 DataSnap 新功能主要是增加许多监督的功能，10.3 允许 DataSnap 伺服端和客户端能够在执行时期取得更多对方的信息以及对方的执行状态，以便处理突发的状态。例如 10.3 允许 DataSnap 伺服端能够在 DataSnap 客户端突然断线时能够取得这样的信息并且释放客户端在 DataSnap 伺服端配置的资源。

本章即将介绍 DataSnap 10.3 增加的新功能，以便让读者了解如何使用这些新功能来改善或是强化 DataSnap 应用系统。

8-1 DataSnap 10.3 新增监督功能

DataSnap 在 10.3 版中进行最多的改善功能应该就是为 DataSnap 伺服端和客户端加入了大量的监督，管理功能，以便让开发人员能够取得更多的控制和较详细的执行状态数据，首先让我们讨论如何在 DataSnap 伺服端取得客户端的信息。

8-1-1 DataSnap 伺服端监督功能

DataSnap 10.3 为服务器加入了取得 DataSnap 客户端的链接信息类别，藉由 TDBXClientInfo 记录，DataSnap 服务器可以取得链接客户端的 IP 地址，使用的通讯协议等信息，下面即是 TDBXClientInfo 的定义：

```
TDBXClientInfo = record
  IpAddress: String;
  ClientPort: String;
  Protocol: String;
  AppName: String;
end;
```

下面的表格说明了 TDBXClientInfo 记录中特性值的意义:

特性名称	说明
IpAddress	客户端的链接 IP 地址
ClientPort	客户端使用的连结通信埠
Protocol	客户端使用的通讯协议
AppName	客户端的应用程序名称(当客户端是 Web 客户端才有这项信息)

DataSnap 服务器要取得 TDBXClientInfo 中的信息可以藉由 TDSConnectEventObject 对象，当 DataSnap 客户端链接到 DataSnap 服务器时，会触发 TDSServer 组件的 OnConnect 事件处理函式，而 TDSServer 组件的 OnConnect 事件处理函式就会接受一个 TDSConnectEventObject 对象的参数。

下面是 TDSConnectEventObject 的定义，我们可以看到它定义了一个型态为 TDBXChannelInfo 的特性，FChannelInfo:

```
 TDSConnectEventObject = class(TDSEventObject)
 public
   constructor Create(const ADbxContext: TDBXContext; const AServer: TDSCustomServer;
const ATransport: TDSServerTransport; const AChannelInfo: TDBXChannelInfo; const
ADbxConnection: TDBXConnection; const AConnectProperties: TDBXProperties);
 private
   FConnectProperties: TDBXProperties;
   FChannelInfo: TDBXChannelInfo;
 Public
```

而 TDBXChannelInfo 的类别定义如下:

```
 TDBXChannelInfo = class
 public
   constructor Create(const AId: Integer);
 protected
   function GetInfo: UnicodeString; virtual;
 private
   FId: Integer;
   FClientInfo: TDBXClientInfo;
 public
   property Id: Integer read FId;
   property Info: UnicodeString read GetInfo;
```

```
  property ClientInfo: TDBXClientInfo read FClientInfo write FClientInfo;
end;
```

我们可以从 TDBXChannelInfo 类别中看到它定义了一个型态为 TDBXClientInfo 的特性 ClientInfo，因此我们只需要在发 TDSServer 组件的 OnConnect 事件处理函式中使用下面的程序代码就可以取得 TDBXClientInfo 中定义的 DataSnap 客户端信息:

```
DSConnectEventObject.ChannelInfo.ClientInfo
```

现在让我们展示如何在 DataSnap 服务器中显示链接的 DataSnap 客户端的信息。首先建立一个 DataSnap 服务器项目，并且选择同时支持 TCP/IP 和 HTTP 两种通讯协议:

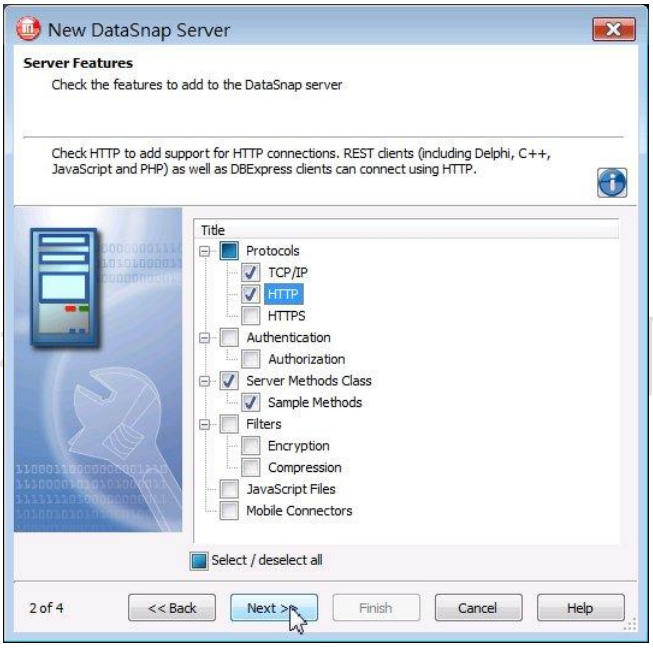


图 1 建立支持 TCP/IP 和 HTTP 两种通讯协议的 DataSnap 服务器

在建立了范例 DataSnap 服务器之后，开启 ServerContainerUnit 程序单元，撰写程序单元中 TDSServer 组件的 OnConnect 事件处理函式如下:

```
001   procedure TServerContainer2.DSServer1Connect(
002     DSConnectEventObject: TDSConnectEventObject);
003   begin
004     ShowClientConnections(DSConnectEventObject);
005   end;
006
007   procedure TServerContainer2.ShowClientConnections(
008     DSConnectEventObject: TDSConnectEventObject);
```

```
009   var
010     sb : TStringBuilder;
011   begin
012     sb := TStringBuilder.Create;
013     try
014       sb.Append('AppName : ' + DSConnectEventObject.ChannelInfo.ClientInfo.AppName);
015       sb.Append(' ');
016       sb.Append('Protocol : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.Protocol);
017       sb.Append(' ');
018       sb.Append('IpAddress : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.IpAddress);
019       sb.Append(' ');
020       sb.Append('ClientPort : ' +
DSConnectEventObject.ChannelInfo.ClientInfo.ClientPort);
021       sClient := sb.ToString;
022       TThread.Synchronize(nil, UpdateClientConnections);
023     finally
024       sb.Free;
025     end;
026   end;
```

在上面的程序代码中藉由 TDSConnectEventObject 对象的 ChannelInfo 特性值取得 TDBXChannelInfo 对象，再藉由 TDBXChannelInfo 对象的 ClientInfo 特性值取得 TDBXClientInfo 记录，然后就可以撷取其中的 DataSnap 客户端信息了。

下面的画面是执行此范例 DataSnap 服务器应用程序，并且在客户端分别执行使用 TCP/IP 通讯协议的 Window DataSnap 客户端以及使用 HTTP 通讯协议的 Web 客户端呼叫范例 DataSnap 服务器中的方法，从范例 DataSnap 服务器的主窗体中我们可以看到范例 DataSnap 服务器果然可以取得每一个链接 DataSnap 客户端的相关信息：

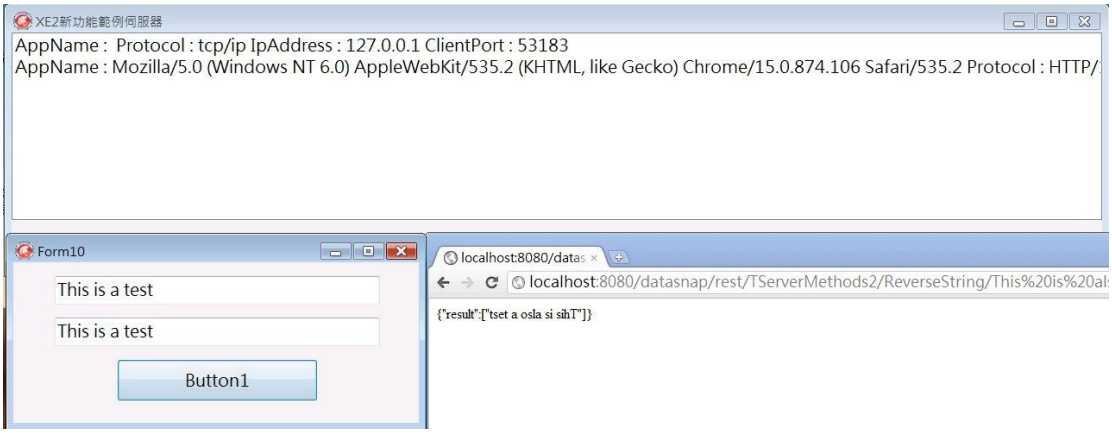


图 2 范例 DataSnap 服务器显示使用 TCP/IP 和 HTTP 通讯协议链接的 DataSnap 客户端的信息

8-2 DataSnap Session 功能

DataSnap 10.3 在伺服端也提供了 Session 对象的机制，也提供了许多有用的 Session 管理功能。在 DataSnap 框架中，伺服端的 Session 是由 TDSSession 类别定义的，TDSSession 类别提供了许多有用的方法和特性让开发人员能够取得服务或是取得重要的信息，例如取得 Session 状态，安排 Session 对象定时或是自动执行工作，或是在 Session 对象中暂时储存数据等。

如果开发人员需要暂时在 Session 对象中储存数据，那么开发人员可以使用下面的方法来储存字符串或是客制化类别对象：

方法	说明
function HasData(Key: String): Boolean	判断在 Session 对象中是否存有字符串数据
function GetData(Key: String): String	在 Session 对象中根据传入的键值取得字符串数据
procedure PutData(Key, Value: String)	把键值/字符串数值储存到 Session 对象中
procedure RemoveData(Key: String)	从 Session 对象中根据键值移除字符串数据
function HasObject(Key: String): Boolean	判断在 Session 对象中是否存有对象
function GetObject(Key: String): TObject	在 Session 对象中根据传入的键值取得对象
function PutObject(Key: String; Value: TObject): Boolean	把键值/对象储存到 Session 对象中
function RemoveObject(Key: String; InstanceOwner: Boolean = True): TObject	从 Session 对象中根据键值移除对象

除了 TDSSession 类别之外，DataSnap 框架也定义了 TDSSessionManager 类别来管理所有的 Session 对象，

TDSSessionManager 使用了 Singleton 设计样例，因此在整个 DataSnap 服务器中只有一个 TDSSessionManager 对象。

TDSSessionManager 类别提供了许多让开发人员可以为特定的 Session 加入监督事件，以便在特定 Session 事件发生时能够呼叫开发人员定义的事件处理函式。此外 TDSSessionManager 也提供了许多方法让开发人员能够拜访或是管理所有 TDSSessionManager 对象管理的 Session 对象。

现在让我们来看看如何使用这两个类别。

假设我们现在有下面的类别，当 DataSnap 客户端连结到 DataSnap 服务器时，我们希望建立一个 TMySessionData 对象并且储存在这个 DataSnap 客户端的 Session 对象中。

```
type
  TMySessionData = class
  private
    FdtData : longint;
  public
    property myTime : longint read FdtData write FdtData;
    destructor Destroy; override;
  end;

implementation

{ TMySessionData }

{ TMySessionData }

destructor TMySessionData.Destroy;
begin
  fdtData := 0;
  inherited;
end;
```

现在让我们开启 8-1 小节的范例 DataSnap 服务器，在 TServerMethods2 类别中加入两个新的方法 GetSessionData 和 StoreSessionData，如下所示:

```
001    procedure TServerMethods2.GetSessionData(const key: String;
002      out dtDateTime: longint);
003    var
```

```
004     session : TDSSession;
005     sessionData : TMySessionData;
006   begin
007     session := TDSSessionManager.GetThreadSession;
008     sessionData := TMySessionData(session.GetObject(key));
009     dtDateTime := sessionData.myTime;
010   end;
011
012   procedure TServerMethods2.StoreSessionData(dtDateTime : longint; out sKey : String);
013   var
014     session : TDSSession;
015     sessionData : TMySessionData;
016   begin
017     session := TDSSessionManager.GetThreadSession;
018     sessionData := TMySessionData.Create;
019     sessionData.myTime := dtDateTime;
020     sKey := session.SessionName;
021     if (session.PutObject(sKey, sessionData)) then
022       TThread.Synchronize(nil, UpdateDataStatus);
023   end;
```

在范例 DataSnap 客户端中先呼叫 DataSnap 服务器的 StoreSessionData 方法以储存参数 dtDateTime 的数值，并且从参数 sKey 中得到 DataSnap 服务器回传的键值(其实就是这个 DataSnap 客户端的 Session 对象的 Id 值)。而 StoreSessionData 在 017 行先藉由 TDSSessionManager 类别的类别方法 GetThreadSession 取得目前这个 DataSnap 客户端专属的 Session 对象，在 018 行建立 TMySessionData 对象，019 行把参数 dtDateTime 储存到 TMySessionData 对象的 myTime 特性中，最后在 020 行把 Session 对象的 SessionName 特性值储存到参数 sKey 中。请注意，由于参数 sKey 是定义为 out 型态的参数，因此这个参数值会回传回客户端。

而上面的 GetSessionData 方法则接受客户端传递来的键值(Session Id)，008 行呼叫 Session 对象的 GetObject 方法以键值取得对应的储存对象，由于 GetObject 回传的是 TObject，因此 008 行再转变型态为 TMySessionData 即可。

由于在前面我们建立了一个 TMySessionData 对象并且储存在 Session 对象，那么如果希望在 Session 关闭或是结束时能够自动释放 TMySessionData 对象的话，那么要如何做呢？我们可以藉由向 Session 对象注册一个回叫函式，

在 Session 对象发生特定的事件时自动呼叫我们的事件处理函式。在目前的 DataSnap 框架中为 Session 定义了如下的两个状态:

```
TDSSessionEventType = (SessionCreate, SessionClose);
```

SessionCreate 代表目前 Session 对象被建立，SessionClose 则代表目前 Session 对象在关闭或是结束状态。

由于所有的 Session 对象都是由 TDSSessionManager 单一对象管理的，因此我们可以向这个 TDSSessionManager 对象注册回叫事件，让 TDSSessionManager 在建立或是结束 Session 时呼叫我们的事件处理函式。要向 TDSSessionManager 注册回叫事件，我们可以呼叫 TDSSessionManager 的 AddSessionEvent 方法:

```
procedure AddSessionEvent(Event: TDSSessionEvent);
```

AddSessionEvent 接受一个型态为 TDSSessionEvent 的事件参数，而 TDSSessionEvent 则定义如下:

```
TDSSessionEvent = reference to procedure(Sender: TObject;
                        const EventType: TDSSessionEventType;
                        const Session: TDSSession);
```


因此我们就可以撰写一个原型为 TDSSessionEvent 的函式并且传递给 AddSessionEvent 方法。现在就让我们撰写一个回叫事件并且向 TDSSessionManager 对象注册。

现在开启范例 DataSnap 服务器的 TServerContainer2 程序单元，在它的 OnCreate 事件处理函式中呼叫 AddSessionListener 方法。AddSessionListener 方法在 008 行藉由 TDSSessionManager 类别的类别特性 Instance 取得 DataSnap 服务器中唯一的 TDSSessionManager 对象，接着呼叫它的 AddSessionEvent 方法，并且传递行 009 行开始的匿名程序做为参数，这个匿名程序的原型和 TDSSessionEvent 定义的是相符合的:

```
001   procedure TServerContainer2.DataModuleCreate(Sender: TObject);
002   begin
003     AddSessionListener;
004   end;
005
006   procedure TServerContainer2.AddSessionListener;
007   begin
008     TDSSessionManager.Instance.AddSessionEvent(
009       procedure(Sender: TObject;
```

```
010               const EventType: TDSSessionEventType;
011               const Session: TDSSession)
012       begin
013         case EventType of
014           SessionCreate :
015             begin
016               Inc(iSessionNumber);
017               TThread.Synchronize(nil, UpdateSessionNumber);
018             end;
019           SessionClose  :
020             begin
021               ReleaseAnySessionObject(Session);
022             end;
023         end;
024       end);
025   end;
026
027   procedure TServerContainer2.ReleaseAnySessionObject(const Session: TDSSession);
028   begin
029     if (Session.HasObject(Session.SessionName)) then
030     begin
031       Session.RemoveObject(Session.SessionName, True);
032     end;
033   end;
```

这个匿名程序中会根据它的 EventType 参数值来判断目前 Session 对象的状态，如果发现目前 Session 即将结束，那么就在 021 行呼叫 ReleaseAnySessionObject 方法来释放 TMySessionData 对象。

现在如果我们执行范例 DataSnap 服务器和范例 DataSnap 客户端以及使用浏览器存取范例 DataSnap 服务器，那么我们可以看到如下的执行结果，不论是 Windows DataSnap 客户端或是浏览器客户端宦都可以储存 TMySessionData 对象到每一个客户端专属的 Session 对象中并且从 DataSnap 服务器取得 Session Id。而当客户端直接结束时，也能够藉由匿名回叫程序自动释放 TMySessionData 对象。

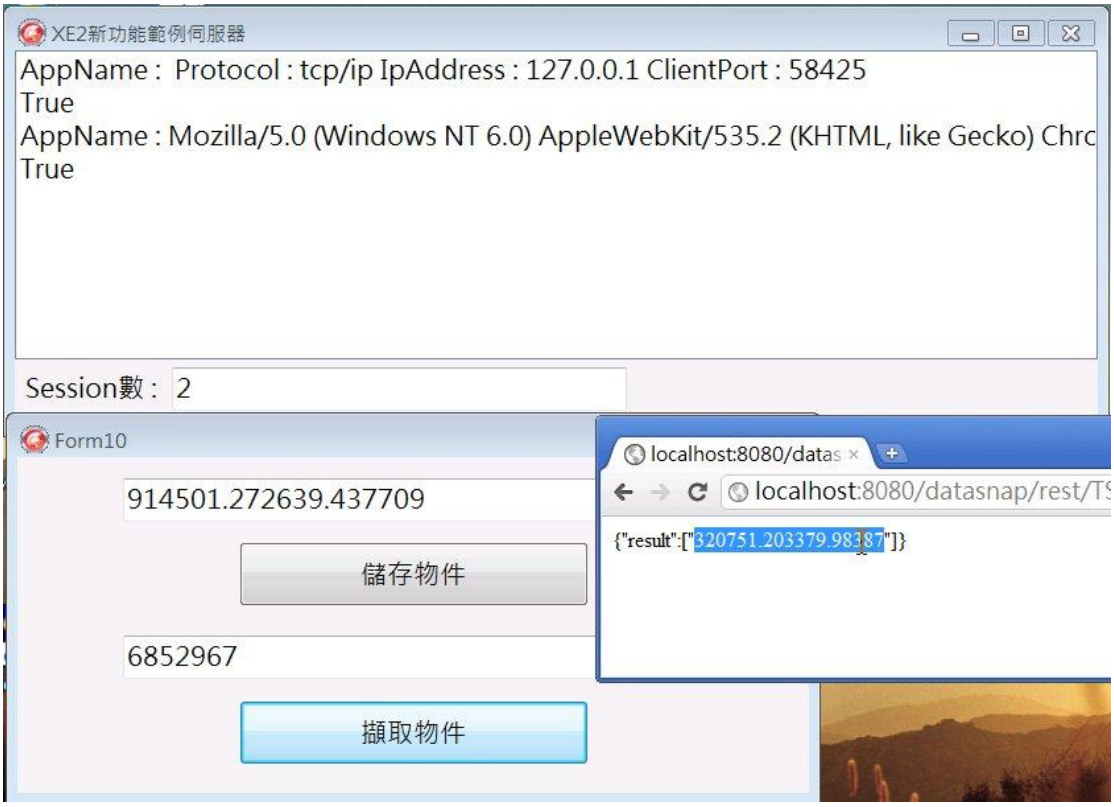


图 3 DataSnap 客户端应用程序能够储存对象在 DataSnap 服务器的 Session 之中

> 注意，前面讨论的是告诉读者如何使用 TDSSessionManager 和 TDSSession 类别，即使读者在 Session 对象中储存了客制化对象，但没有使用前面的方法释放客制化对象时，DataSnap 框架在结束 Session 对象时也会自动释放所有储存在 Session 对象中的客制化对象

8-3 TCP 链接监督功能

当使用 DataSnap 开发分布式应用系统时，DataSnap 服务器可以服务使用各种不同通讯协议连结的客户端，包括了使用 HTTP/HTTPS 和 TCP/IP 的客户端。对于 HTTP/HTTPS 客户端，DataSnap 服务器可以在完成客户端的请求之后即释放客户端在伺服端配置的资源，但对于使用 TCP/IP 链接的客户端应用程序而言，这个连结是具状态，且可能一直保持连结的状态，因此在服务 TCP/IP 的客户端时，DataSnap 伺服端无法自动在客户端完成服务请求时即释放资源，必须等待 TCP/IP 的客户端安全的脱机之后才能够释放此客户端在伺服端配置的资源。

但如果使用 TCP/IP 连结的客户端由于某种原因而断线时，DataSnap 伺服端可能并不知道 TCP/IP 链接的客户端断线了，因此无法释放伺服端的资源，最后造成 DataSnap 服务器无法负荷而异常终止执行。

为了解决这个问题，DataSnap 10.3 特别加强对于使用 TCP/IP 链接的客户端的监督功能，希望能够让 DataSnap 服务器能够掌握不正常 TCP/IP 客户端的联机和断线状况，以便适当的释放伺服端的资源。因此，开发人员如果了解 DataSnap 10.3 的 TCP 链接监督功能，再结合下一小节讨论的 KeepAlive 功能，那么就可以开发出处理 DataSnap 客户端不正常断线的状况。

DataSnap 框架中的 TDSTCPServerTransport 组件提供了 OnConnect 和 OnDisconnect 两个事件处理函式来通知 DataSnap 服务器使用 TCP/IP 的 DataSnap 客户端联机的断线的事件，开发人员可以在这两个事件处理函式执行程序代码以处理相对应的工作，下面是这两个事件处理函式的原型：

```
property OnConnect: TDSTCPConnectEvent read FTDSTCPConnectEvent write FTDSTCPConnectEvent;
property OnDisconnect: TDSTCPDisconnectEvent read FTDSTCPDisconnectEvent write
FTDSTCPDisconnectEvent;
```

其中的 OnConnect 事件特别重要，因为它可提供丰富的信息，让 DatsSnap 服务器能够记录客户端的信息，以便处理不正常断线的状况。OnConnect 事件是宣告为如下的函式原型型态：


```
TDSTCPConnectEvent = procedure(Event: TDSTCPConnectEventObject) of object;
```

TDSTCPConnectEvent 函式接受一个型态为 TDSTCPConnectEventObject 的 Event 参数，在 TDSTCPConnectEventObject 中宣告了使用 TCP/IP 客户端的联机信息，其中有两个重要的特性 :Connection 和 Channel 。TDSTCPConnectEventObject 是一个记录型态，它的宣告原型如下：

```
TDSTCPConnectEventObject = record
private
  FConnection: TObject;
  FChannel: TDSTCPChannel;
public
  constructor Create(AConnection: TObject; AChannel: TDSTCPChannel);
  property Connection: TObject read FConnection;
  property Channel: TDSTCPChannel read FChannel;
end;
```

下面表格说明了 Connection 和 Channel 特性的意义：

特性	说明
Connection	客户端的链接对象，在目前 10.3 中是 TIdTCPConnection 型态的对象
Channel	代表此链接的信道对象，是型态为 TDSTCPChannel 的对象

其中代表此链接通道的特性 Channel 是 TDSTCPChannel 的对象，而 TDSTCPChannel 类别中则包含了下一小节即将讨论的 KeepAlive 相关的函式，此外 TDSTCPChannel 类别中也包含了此连结客户端的 SessionId。

因此开发人员可以使用下面的步骤管理使用 TCP/IP 链接的 DataSnap 客户端应用程序:

- 在 TDSTCPServerTransport 的 OnConnect 事件处理函式中记录此链接 DataSnap 客户端的链接信息，以处理正常和不正常的断线状况。例如使用 TObjectDictionary 记录 Connection 和 Channel 特性，或是使用 TDictionary 记录 SessionId 和 Channel 特性
- 在 DataSnap 客户端应用程序不正常使用 DataSnap 服务器的服务时，藉由记录的 Channel 特性来强迫切断 DataSnap 客户端的连结
- 在使用 KeepAlive 功能时，当使用 TCP/IP 连结的 DataSnap 客户端不正常断线时，切断 DataSnap 客户端的连结并且释放 DataSnap 客户端在伺服端配置的资源

在下面的小节中我们将示范如何完成上述的前 2 项工作，至于上述第 3 项的工作将在下一节『KeepAlive 功能』中说明。

8-3-1 使用 TCP 链接监督功能

让我们继续使用前面小节的范例 DataSnap 服务器做为说明，此时我们需要为范例 DataSnap 服务器建立下面的功能:

1. 建立一个数据结构以记录使用 TCP/IP 通讯协议链接到 DataSnap 服务器的 DataSnap 客户端应用程序相关的 TIdTCPConnection 和 TDSTCPChannel 对象
2. 当需要强迫断线 DataSnap 客户端应用程序时，需要找到此 DataSnap 客户端应用程序被记录的 TIdTCPConnection 和 TDSTCPChannel 对象

3. 呼叫 TDSTCPChannel 对象的 Close 方法强迫切断 DataSnap 客户端应用程序的链接

首先让我们为的范例 DataSnap 服务器中的 ServerContainer 程序单元中的 TDSTCPServerTransport 组件建立一个 OnConnect 事件处理函式，以记录使用 TCP/IP 通讯协议链接到 DataSnap 服务器的 DataSnap 客户端应用程序相关的 TIdTCPConnection 和 TDSTCPChannel 对象:

```
001   procedure TServerContainer2.DSTCPServerTransport1Connect(
002     Event: TDSTCPConnectEventObject);
003   begin
004     System.TMonitor.Enter(FConnections);
005     try
006       FConnections.Add(TIdTCPConnection(Event.Connection), Event.Channel);
007     finally
008       System.TMonitor.Exit(FConnections);
009     end;
010     AddConnectionToList(TIdTCPConnection(Event.Connection), Event.Channel);
011     TThread.Synchronize(nil, UpdateTCPMonitorInfo);
012   end;
```

在 006 行使用了 FConnections 对象记录 DataSnap 客户端应用程序相关的 TIdTCPConnection 和 TDSTCPChannel 对象。

而 FConnections 则是宣告为 TObjectDictionary 的型态的对象，如下所示:

```
   FConnections: TObjectDictionary<TIdTCPConnection,TDSTCPChannel>;
```

FConnections 是在 ServerContainer 程序单元的 OnCreate 事件处理函式中建立的:

```
procedure TServerContainer2.DataModuleCreate(Sender: TObject);
begin
  FConnections := TObjectDictionary<TIdTCPConnection,TDSTCPChannel>.Create;
  AddSessionListener;
end;
```

现在回到范例 DataSnap 服务器的主窗体，让我们在其中加入一个『关闭选择的 TCP 客户端』Button 组件以及其下方的 TListBox 组件，为这个 TListBbox 取名为 lbTCPMonitorInfo，如下所示:

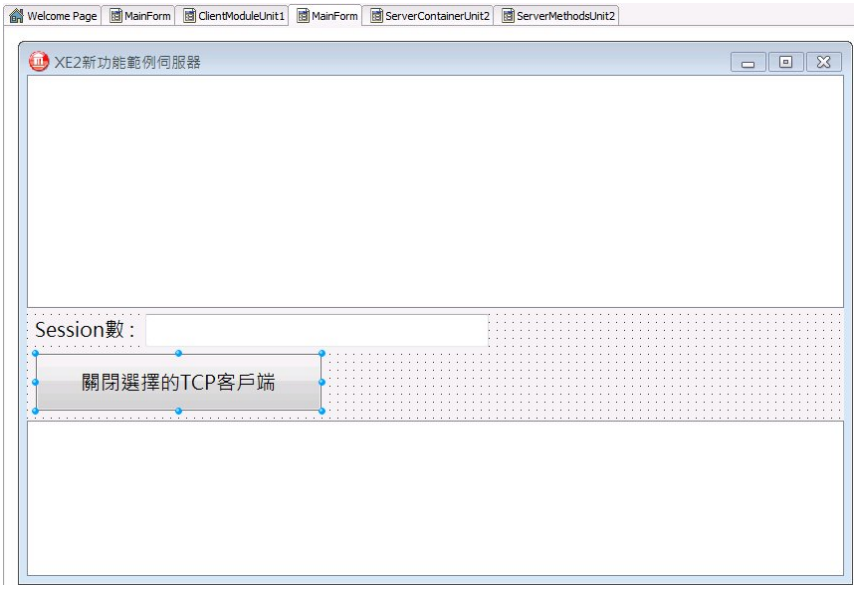


图 4 在范例 DataSnap 服务器的主窗体中加入『关闭选择的 TCP 客户端』Button 组件

现在我们要展示前面的第 2 个步骤，记录 TIdTCPConnection 和 TDSTCPChannel 对象在 lbTCPMonitorInfo 中，以便让使用者可以藉由点选 lbTCPMonitorInfo 之中的链接信息，再点选『关闭选择的 TCP 客户端』Button 组件以切断使用 TCP/IP 通讯协议链接的 DataSnap 客户端应用程序。

请读者注意的是，在前面 DSTCPServerTransport1Connect 的事件处理函式中第 010 行呼叫了 AddConnectionToList 方法，而 AddConnectionToList 则是把 TIdTCPConnection 对象藉由呼叫 lbTCPMonitorInfo 的 AddObject 方法把 TIdTCPConnection 对象加入到 TListBox 中:

```
001   procedure TServerContainer2.UpdateTCPMonitorInfo;
002   begin
003     frmMainForm.lbTCPMonitorInfo.Items.AddObject(ConnInfoStr, pConn);
004   end;
005
006   procedure TServerContainer2.AddConnectionToList(Conn: TIdTCPConnection; Channel:
TDSTCPChannel);
007   begin
008     pConn := Conn;
009     if (Conn <> nil) and (Channel <> nil) and (Channel.ChannelInfo <> nil) and
010        (Channel.ChannelInfo.ClientInfo.IpAddress <> EmptyStr) then
011     begin
012       with Channel.ChannelInfo.ClientInfo do
```

```
013       begin
014         ConnInfoStr := Format('%s:%s', [IpAddress, ClientPort]);
015       end;
016     end
017     else
018       ConnInfoStr := '信道信息错误.';
019   end;
```

最后，让我们实作『关闭选择的 TCP 客户端』Button 组件的 OnClick 事件处理函式。当使用者点选了其下方的 lbTCPMonitorInfo 其中某一个链接信息时，我们可以从其中撷取出已经储存在这个被选择的项目中的 TIdTCPConnection 对象，藉由 TIdTCPConnection 对象取得 TDSTCPChannel 对象，最后再呼叫 TDSTCPChannel 对象的 Close 方法即可。

下面是『关闭选择的 TCP 客户端』Button 组件的 OnClick 事件处理函式:

```
001   procedure TfrmMainForm.Button1Click(Sender: TObject);
002   var
003     pConn: TIdTCPConnection;
004     connstr : String;
005   begin
006     pConn := GetSelectedConnection;
007     ServerContainer2.DisConnectConnection(pConn);
008     ShowMessage('已切断 : ' + lbTCPMonitorInfo.Items[lbTCPMonitorInfo.ItemIndex] + '
的联机');
009   end;
010
011   function TfrmMainForm.GetSelectedConnection: TIdTCPConnection;
012   var
013     I, Count, Index: Integer;
014     Obj: TObject;
015   begin
016     Result := nil;
017     Index := -1;
018     Count := lbTCPMonitorInfo.Count;
019
020     if Count > 0 then
021     begin
022       for I := 0 to Count - 1 do
```

```
023       begin
024         if lbTCPMonitorInfo.Selected[I] then
025         begin
026           Index := I;
027           break;
028         end;
029       end;
030
031       if Index > -1 then
032       begin
033         Obj := lbTCPMonitorInfo.Items.Objects[Index];
034         if Obj <> nil then
035           Exit(TIdTCPConnection(Obj));
036       end;
037     end;
038   end;
```

006 行呼叫 GetSelectedConnection 取得在 lbTCPMonitorInfo 被选择的 TIdTCPConnection 对象，接着 007 行呼叫 ServerContainer 的 DisConnectConnection 程序关闭 TCP/IP 链接。

DisConnectConnection 非常的简单，因为一旦有了 TIdTCPConnection 对象，就可以藉由 FConnections 对象取得它相关的 TDSTCPChannel 对象了，最后呼叫 TDSTCPChannel 对象的 Close 方法:

```
procedure TServerContainer2.DisConnectConnection(theConnection : TIdTCPConnection);
var
  theChannel : TDSTCPChannel;
begin
  if (theConnection <> nil) then
  begin
    FConnections.TryGetValue(theConnection, theChannel);
    theChannel.Close;
  end;
end;
```

现在执行范例 DataSnap 服务器，再执行一个范例 DataSnap 客户端应用程序，接着点选范例 DataSnap 服务器主窗体中 lbTCPMonitorInfo 的链接信息，再点选『关闭选择的 TCP 客户端』Button 组件，我们就可以看到如下的执行结果，范例 DataSnap 服务器强迫中断了范例 DataSnap 客户端应用程序的 TCP/IP 链接:

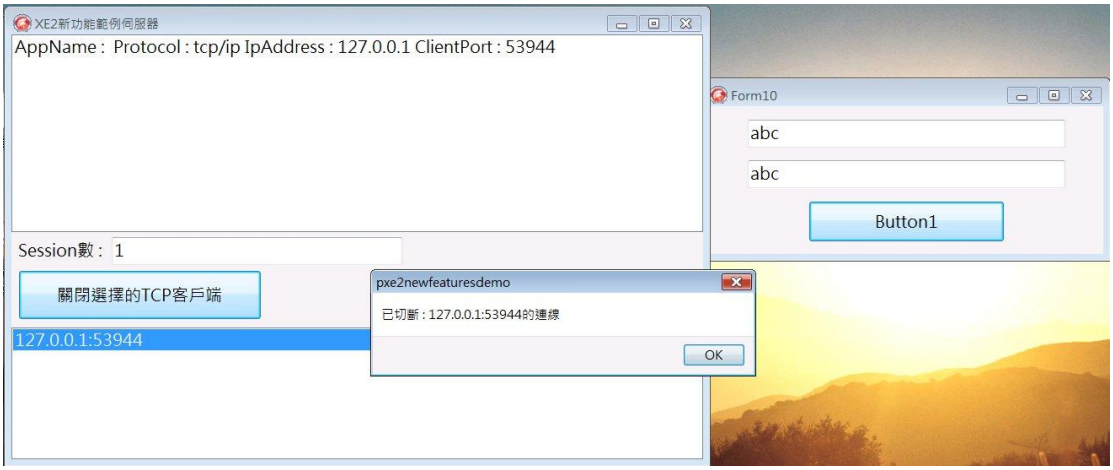


图 5 在范例 DataSnap 服务器中选择要切断的 DataSnap 客户端应用程序

现在如果 DataSnap 客户端应用程序想再存取范例 DataSnap 服务器就会产生错误，如下所示:

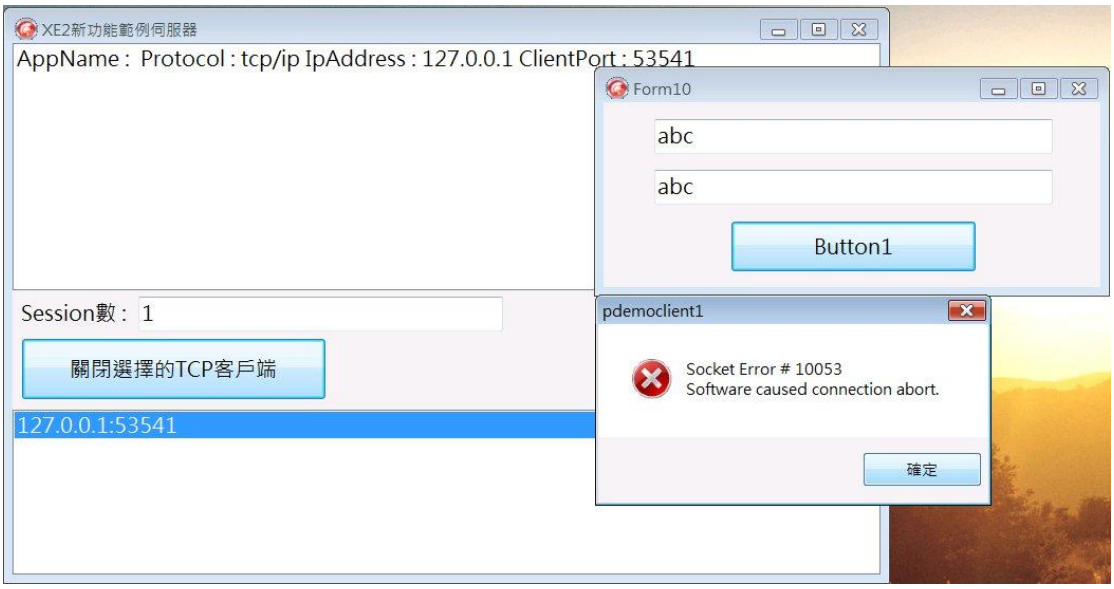


图 7 在范例 DataSnap 服务器切断的 DataSnap 客户端应用程序之后，如果 DataSnap 客户端应用程序想再存取范例 DataSnap 服务器就会产生错误

这个范例充分展现了新的 DataSnap 10.3 框架可监督 TCP 链接的功能。

8-4 KeepAlive 功能

DataSnap 10.3 框架最重要的功能之一就是加入了 DataSnap 服务器和 DataSnap 客户端的互动查询的功能，藉由 KeepAlive 功能，DataSnap 服务器可以主动在设定的时间之内查询 DataSnap 客户端的连结是否正常，如果 DataSnap 服务器一直无法查询到 DataSnap 客户端，那么 DataSnap 服务器

就会主动切断连结并且释放 DataSnap 客户端在 DataSnap 服务器中配置的资源。

这个功能是藉由 TDSTCPServerTransport 组件新的 3 个相关的 KeepAlive 特性来设定和控制的:

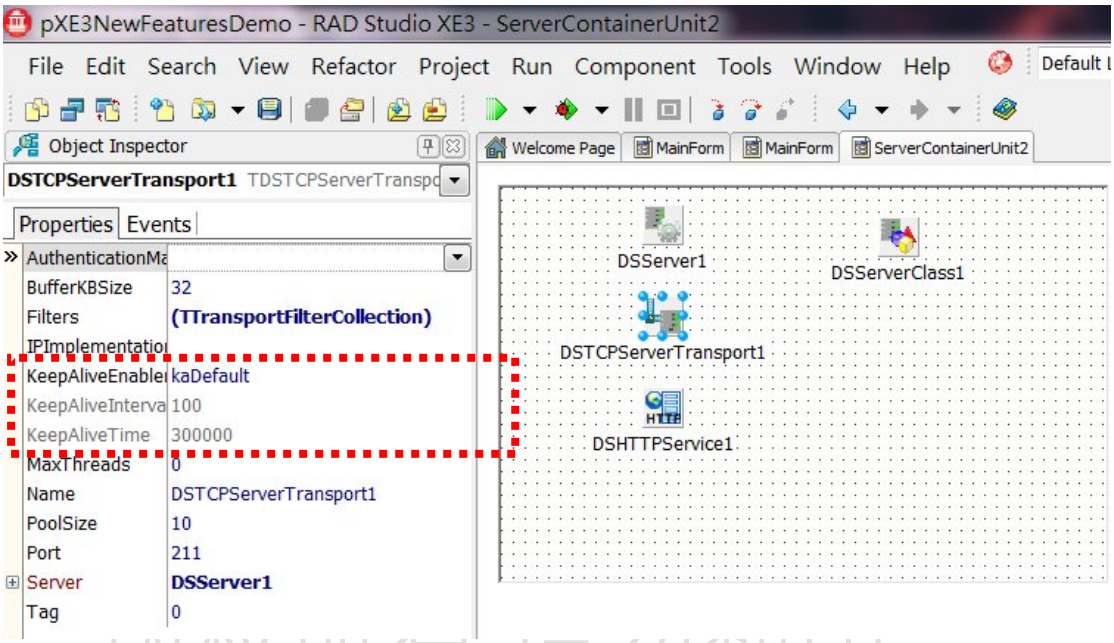


图 5 使用 TDSTCPServerTransport 组件新的 KeepAlive 相关特性设定 DataSnap 服务器和 DataSnap 客户端的互动查询

下面的表格说明了这 3 个相关的 KeepAlive 特性:

特性	说明
KeepAliveEnablement	如何设定 KeepAlive 的状态
KeepAliveInternal(毫秒)	只有 KeepAliveEnablement 特性设定为 kaEnabled 时才作用，它代表每次 DataSnap 服务器查询 DataSnap 客户端是否还存在的间隔时间
KeepAliveTime(毫秒)	只有 KeepAliveEnablement 特性设定为 kaEnabled 时才作用，它代表每次 DataSnap 服务器查询 DataSnap 客户端是否还存在的总时间，如果在这个总时间之内 DataSnap 客户端都没有响应，那么 DataSnap 服务器就会主动切断连结并且释放 DataSnap 客户端在 DataSnap 服务器中配置的资源

下面的表格说明了 KeepAliveEnablement 特性可以设定的特性值:

特性值	说明
kaDefault	使用系统内定的设定
kaDisabled	关闭 KeepAlive 功能
kaEnabled	开启 KeepAlive 功能

这整个的执行流程如下所述:

1. 当开发人员设定了 KeepAliveEnablement 特性值为 kaEnabled 之后，DataSnap 10.3 框架的 KeepAlive 功能便开始启动

2. 当 KeepAlive 功能启动之后，DataSnap 服务器便会等待 KeepAliveTime 特性值设定的时间之后查询 DataSnap 客户端是否还在线

3. 如果查询失败，那么 DataSnap 服务器便会等待 KeepAliveInternal 特性值设定的时间之后，再次查询 DataSnap 客户端是否还在线

4. DataSnap 服务器会根据操作系统设定的查询次数限制，例如 Windows 是查询 10 次，如果在查询了操作系统设定的次数之后 DataSnap 客户端还是没有响应，那么 DataSnap 服务器便会判定 DataSnap 客户端已经因为某种原因断线了

5. 接着 DataSnap 服务器就可以释放 DataSnap 客户端的 TCP/IP 联机以及 DataSnap 客户端在 DataSnap 服务器中配置的任何资源

现在有了 KeepAlive 功能之后，开发人员就可以避免 DataSnap 客户端不正常的断线，再不断的重新连结 DataSnap 服务器，造成 DataSnap 服务器无法释放先前链接所配置的资源，最后造成 DataSnap 服务器因为内存/资源不足而发生执行错误的情形了。

8-5 结论

本章讨论了 DataSnap 10.3 中重要的新功能，开发人员可以藉由这些新功能更完善，精确的控制 DataSnap 应用系统，以便开发出更安全，稳定的分布式应用系统。

开发高效率 DataSnap 篇

在本书的”开发高效率 DataSnap 篇”内容中将讨论如何开发高执行效率的 DataSnap 系统，这是因为在本书前面的内容中说明了如何开发基本的 DataSnap 应用系统，但由于从 Delphi 10.3 版本开始 DataSnap 又开始新增功能而且 Embarcadero 也开始调整 DataSnap 的内部实作方式，再加上本书前面的内容并没有说明如何结合 FireDAC 和 DataSnap 在一起开发，在这些因素相加相乘的影响下本书有必要加入新的章节以便让读者了解如何开发新一代的 DataSnap 架构以便提供高效率的 DataSnap 服务器并且连结移动客户端。

在本书接下来的章节中将讨论下面的内容：

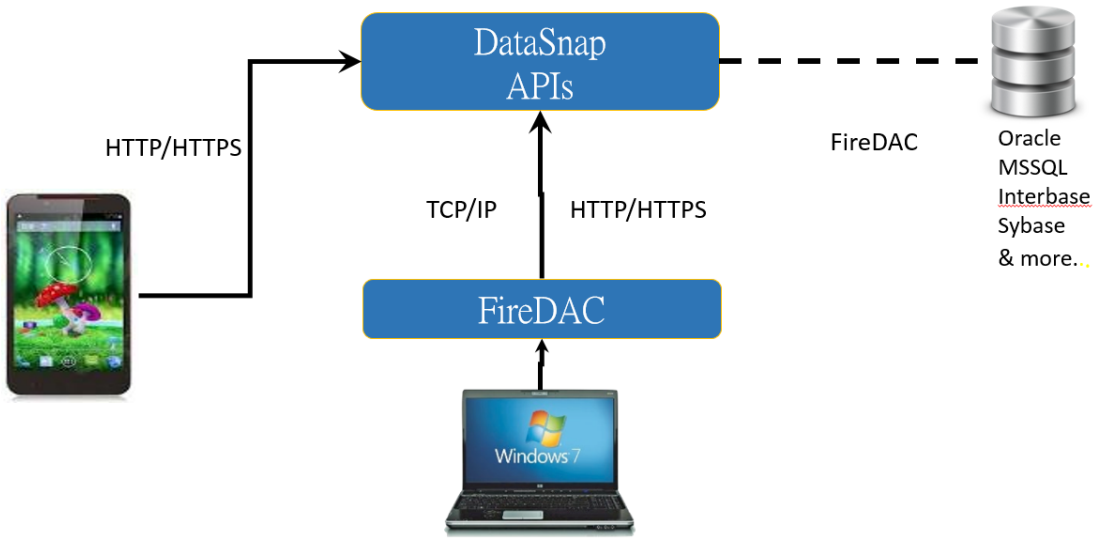
- 使用 FireDAC 开发 DataSnap 应用系统
- 开发安全，高效率的 DataSnap 应用系统

当然读者在前面章节学习到的各种 DataSnap 技术，例如 DataSnap 生命周期，过滤器，回叫机制等都可以继续使用在本篇新的架构中。在稍后的第 9 章中将详细说明如何使用 FireDAC 开发 DataSnap 应用系统，在读者了解如何结合 FireDAC 和 DataSnap 之后第 10 章将进一步讨论如何开发高效率的 FireDAC+DataSnap 应用系统。

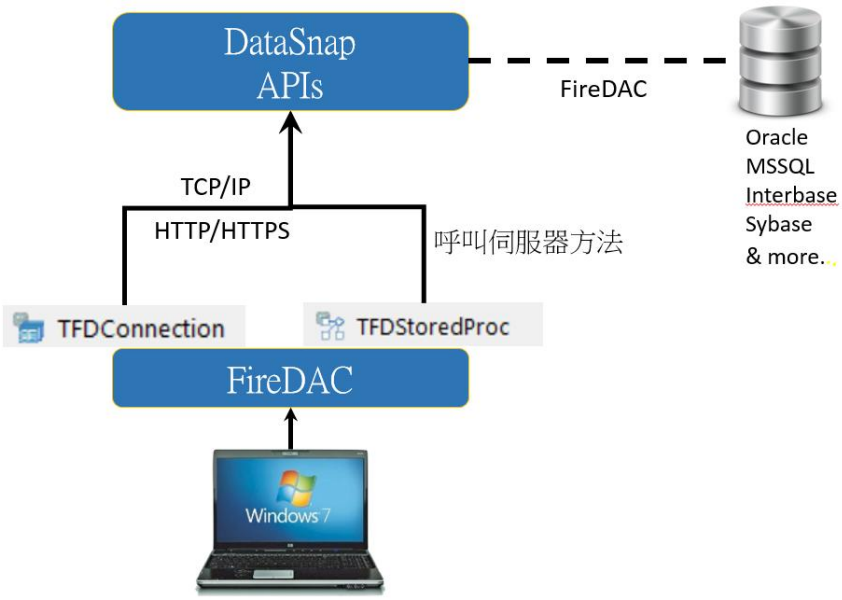
第9章 使用FireDAC开发DataSnap应用系统

当使用 FireDAC 开发 DataSnap 应用系统时和使用前面章节讨论的 dbExpress 有些不同，FireDAC 是把 DataSnap 服务器当成 API 来呼叫，而不像 dbExpress 使用 IAppServer 接口。

FireDAC 客户端可使用 TCP/IP 和 HTTP/HTTPS 连结使用 FireDAC 开发的 DataSnap 服务器，FireDAC 客户端以 API 的方式呼叫 FireDAC 的 DataSnap 服务器，如下图所示：

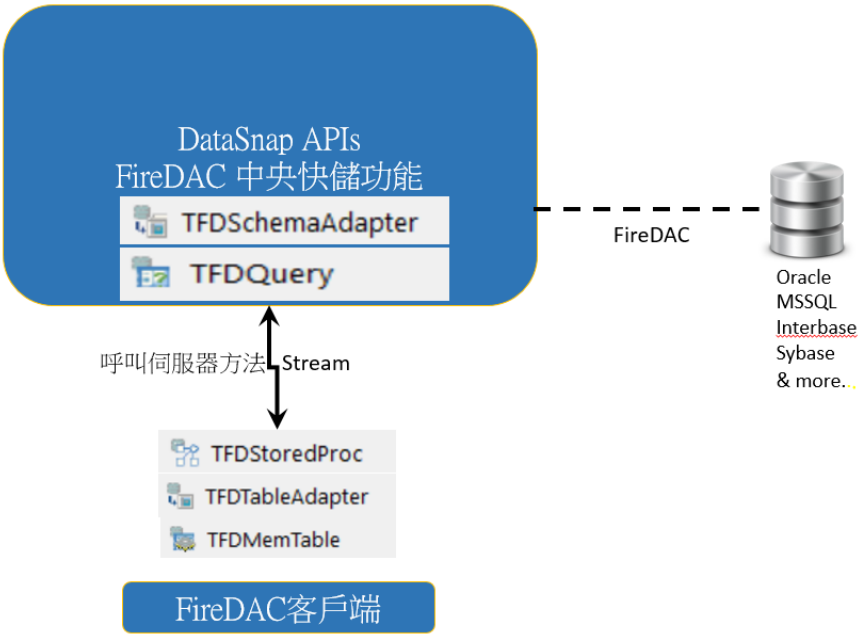


在 FireDAC 客户端也是使用 TFDConnection 组件链接 FireDAC 的 DataSnap 服务器并使用 TFDStoredProc 组件呼叫服器的 API，如下图所示：



如果客户端呼叫的 API 要回传数据，那么 FireDAC 会把数据以 Stream 的格式传递数据，如果要对数据进行包含异动的工作(CRUD)，那么可搭配使用 FireDAC 的中央快储功能来帮助程序员对数据进行异动。

在 DataSnap 架构中要使用 FireDAC 的中央快储功能，程序员需要在伺服端使用 TFDSchemaAdapter 组件，并让 TFDQuery 等组件链接到 TFDSchemaAdapter 组件。而在 FireDAC 客户端则需使用 TFDTableAdapter 组件把在客户端 TFDMemTable 组件中的数据从伺服端取回或是从客户端把异动的数据更新回伺服端，如下图所示：



因此要使用 FireDAC 开发 DataSnap 系统，程序员需要完成下列的步骤：

1. 开发使用 FireDAC 的 DataSnap 服务器，并使用 FireDAC 中央快储功能
2. 开发使用 FireDAC 的客户端，使用 TFDConnection 连结服务器
3. 使用 TFDStoredProc 组件呼叫服器的 API
4. 处理数据数据流(Stream Data)
5. 使用 TFDSchemaAdapter 组件把数据储存到客户端的 TFDMemTable 中
6. 使用 TFDStoredProc 组件呼叫服器的 API 把异动数据回传给 FireDAC 的 DataSnap 服务器，再藉由 FireDAC 中央快储功能把数据更新回数据库

在接下来的内容中将使用一个范例一步一步的说明如何开发一个 FireDAC DataSnap 架构可查询数据并可异动多个数据表的数据。

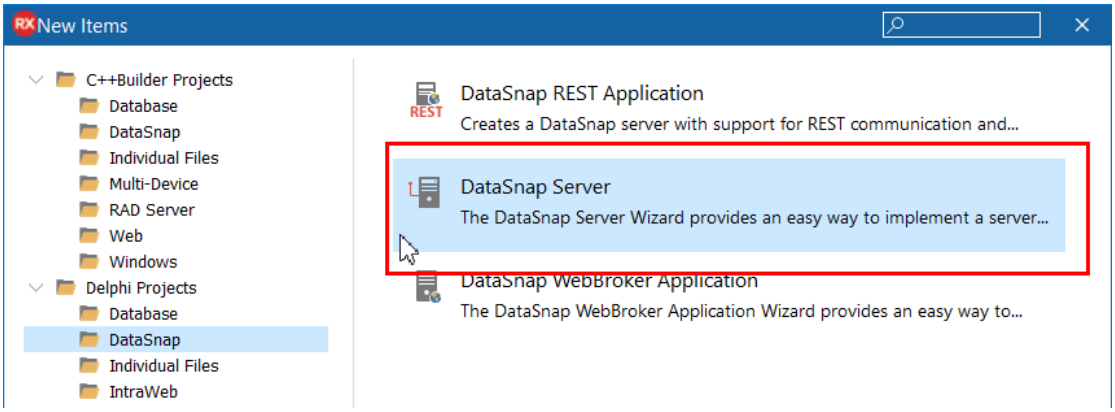
9-1 开发可查询的 FireDAC DataSnap 系统

在本小节中将使用范例数据表 TBLTAIPEIHOTELS 来开发一个能在客户端 PC 和手机中查询台北市旅馆数据的 DataSnaph 系统架构，在读者了解如何使用 FireDAC 开发单一查询的 DataSnap 系统架构之后，下一小节再说明多数据表的 CRUD 应用架构。

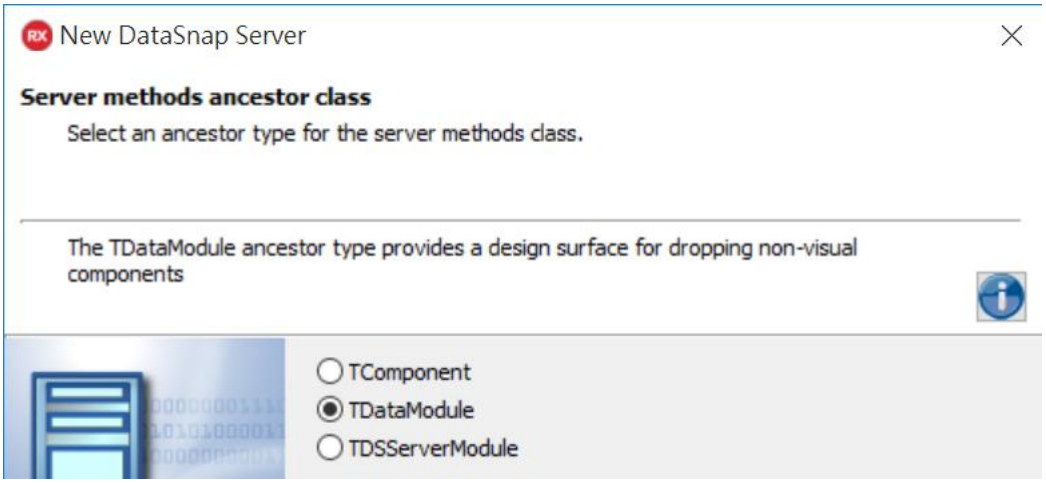
9-1-1 开发 DataSnap 服务器

为了同时让 PC 和手机客户端都能使用 DataSnap 服务器，我们可以使用下列任何种类的 DataSnap 架构，在这里先让我们使用下面的 DataSnap Server 做为范例 DataSnap 服务器，稍后再说明如何使用 RESTful 架构的 DataSnap Server。

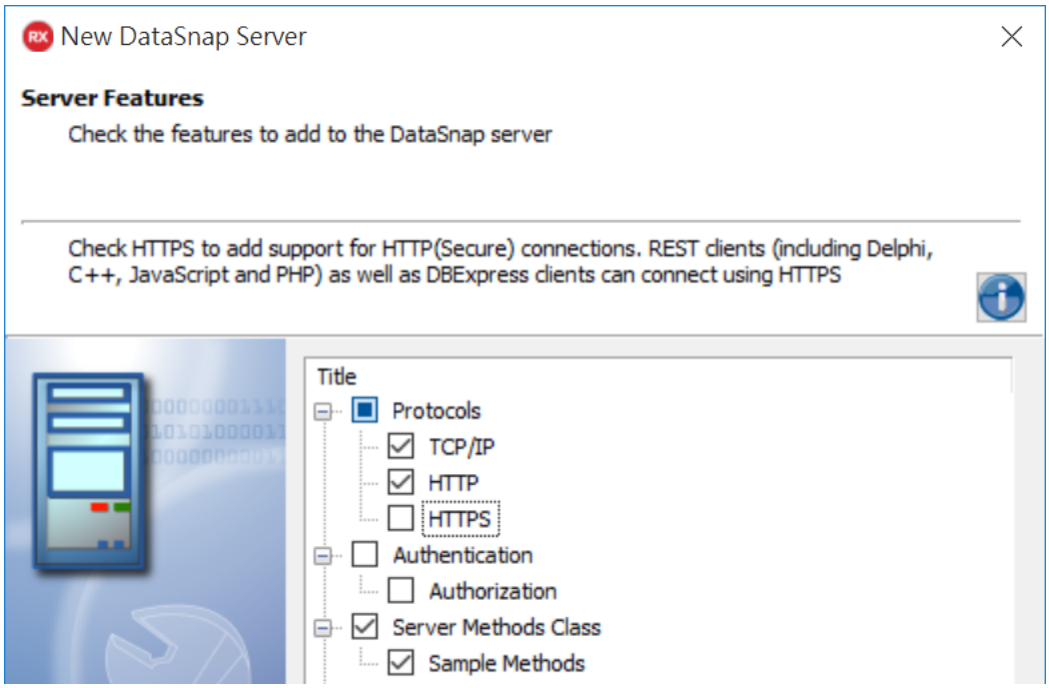
请在 IDE 下方的 New Items 对话盒中选择如下的 DataSnap Server 图像：



由于我们不需要使用 IAppServer 接口，因此可以选择使用 TDataModule 做为伺服端 API 的类别：

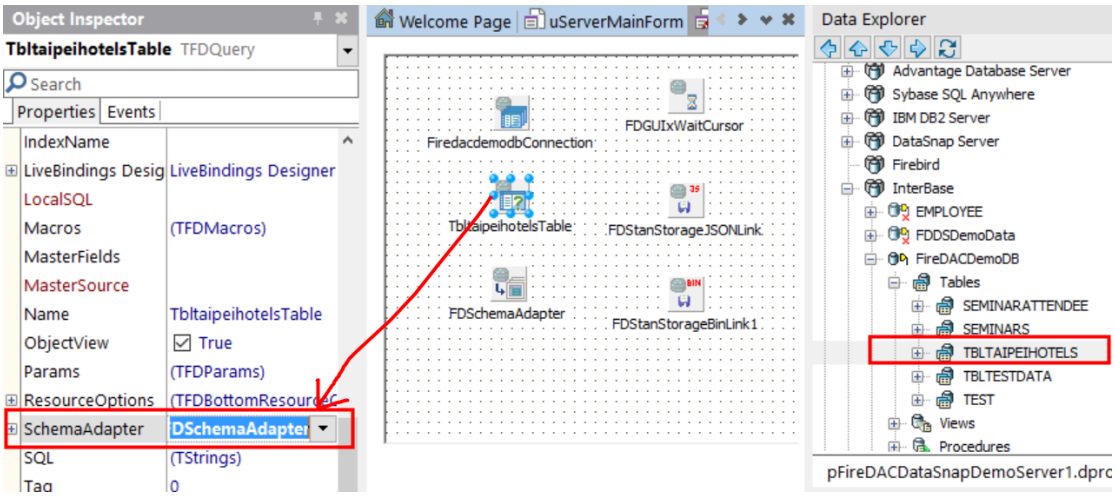


接着在下一个话盒中可选择要支持的功能，例如我们可以让此范例 DataSnap 服务器同时支持 TCP/IP 和 HTTP 通讯协议：

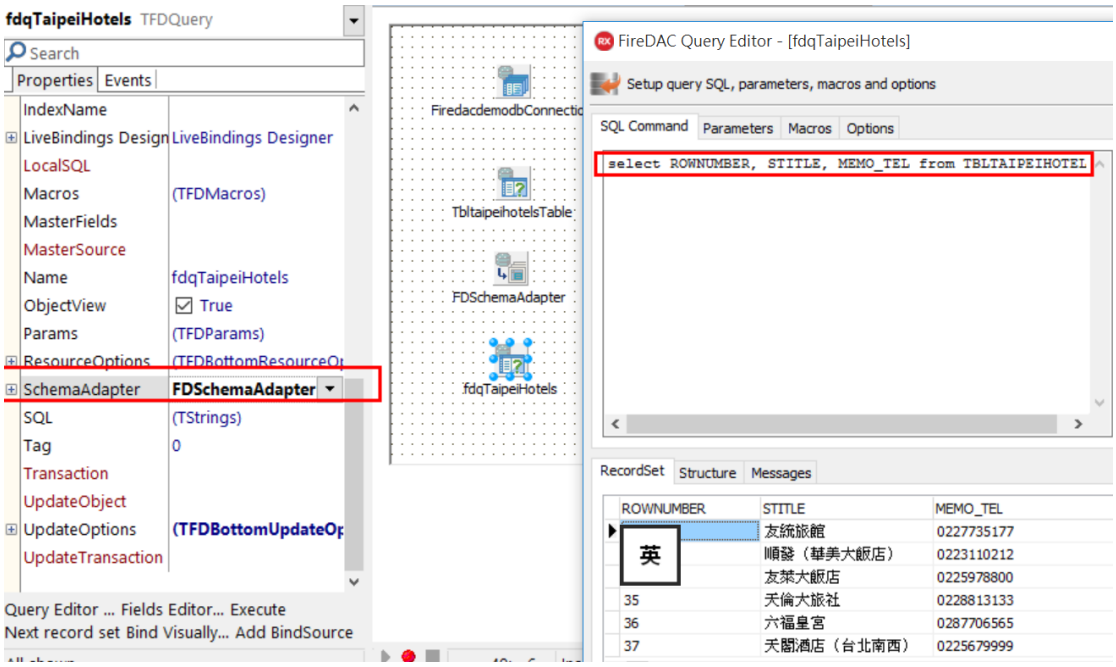


在 IDE 建立好了此范例项目后我们可以开启项目中的 Server Method 程序单元，从 IDE 的 Data Explorer 中拖曳范例数据库中的 TBLTAIPEIHOTELS 这个包含台北市旅馆数据的数据表到数据模块中，并在其中加入如下其他的组件，由于我们要使用 FireDAC 的中央快储功能，因此也加入了下方名为 "FDSchemaAdapter" 的 TFDSchemaAdapter 组件，并且要设定 TbltaipeihotelsTable 组件的 SchemaAdapter 特性为 FDSchemaAdapter。

至于 FDStanStorageJSONLink 和 FDStanStorageBinLink1 组件则是为了让 FireDAC 可处理 2 进位(TCP/IP)和 JSON(HTTP)格式的数据：



接着再放入另外一个 TFDQuery 组件：fdqTaipeiHotels，fdqTaipeiHotels 组件是为了让客户端查询旅馆名称和电话数据使用的组件：



fdqTaipeiHotels 使用了如下的 SQL 命令：


```
select ROWNUMBER, STITLE, MEMO_TEL from TBLTAIPEIHOTELS
```

由于在前面我们选择使用 TDataModule 做为伺服端 API 的类别，因此我们需要使用{$MethodInfo ON}/{$MethodInfo OFF}这对编译程序指令输出 TDataModule 中的公共方法让客户端可以呼叫，并在其中加入下方 008 行的 GetTaipeiHotels()方法让客户端查询旅馆信息。在 FireDAC 中要在伺服端和客户端传递数据，我们只需要传递 TStream 型态的数据即可：

```
001     {$MethodInfo ON}
002     TsmFireDACDataSnapDemo1 = class(TDataModule)
003      ...
004     private
005       { Private declarations }
006     public
007       { Public declarations }
008       function GetTaipeiHotels: TStream;
009     end;
010     {$MethodInfo OFF}
```

最后我们需要实作 GetTaipeiHotels()方法，它的实作程序代码非常简单，003 行先建立一个 TMemoryStream 对象，再开启 fdqTaipeiHotels 组件取得数据，007 行把 fdqTaipeiHotels 组件中的数据集对象藉由 TMemoryStream 类别的 SaveToStream()方法把它拷贝到 TMemoryStream 对象中，在 008 行要把 TMemoryStream 对象中的数据串行流位置重置到开始的位置，最后把 TMemoryStream 对象回传到客户端即可：

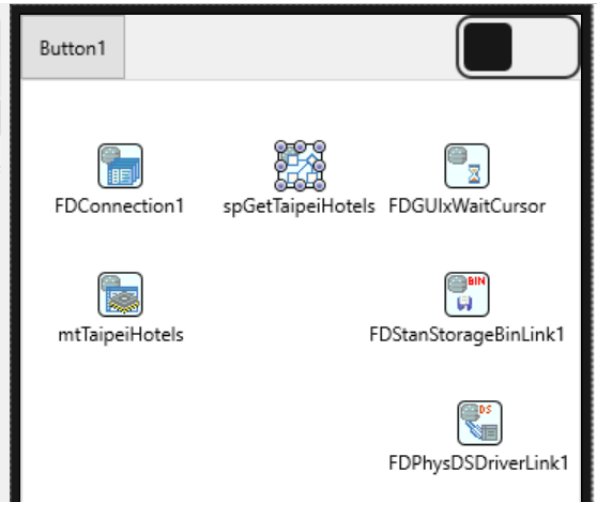
```
001   function TServerMethods1.GetTaipeiHotels: TStream;
002   begin
003     Result := TMemoryStream.Create;
004     try
005       fdqTaipeiHotels.Close;
006       fdqTaipeiHotels.Open;
007       fdqTaipeiHotels.SaveToStream(Result, TFDStorageFormat.sfBinary);
008       Result.Position := 0;
009     except
010       raise;
011     end;
012   end;
```


现在先让我们暂时实作范例 DataSnap 服务器致此，马上就开始开发客户端看看如何让客户端使用 FireDAC 从服务器取得数据。

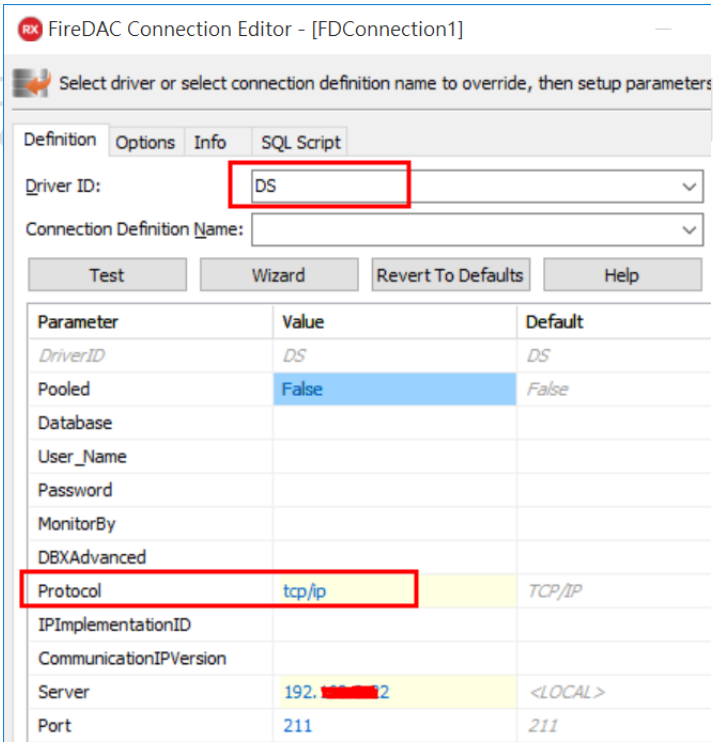
9-1-2 开发 DataSnap 客户端

在使用 FireDAC 呼叫 DataSnap 服务器的 API 时，程序员可使用 TFDStoredProc 组件，如果 DataSnap 服务器的 API 回传包含数据的 TStream 对象，那么客户端可以使用 TFDMemTable 组件来还原数据。

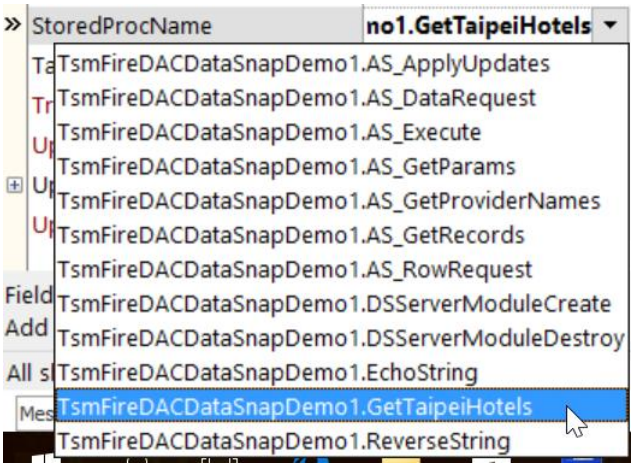
因此请在项目群组中建立一个 Multi-Device Application 客户端项目，在主窗体中加入如下的组件，spGetTheHotel 组件是稍后呼叫 DataSnap 服务器 GetTaipeiHotels()方法使用的，而 mtTaipeiHotels 组件则是用来还原 GetTaipeiHotels()方法回传的包含数据集的 TStream 对象。



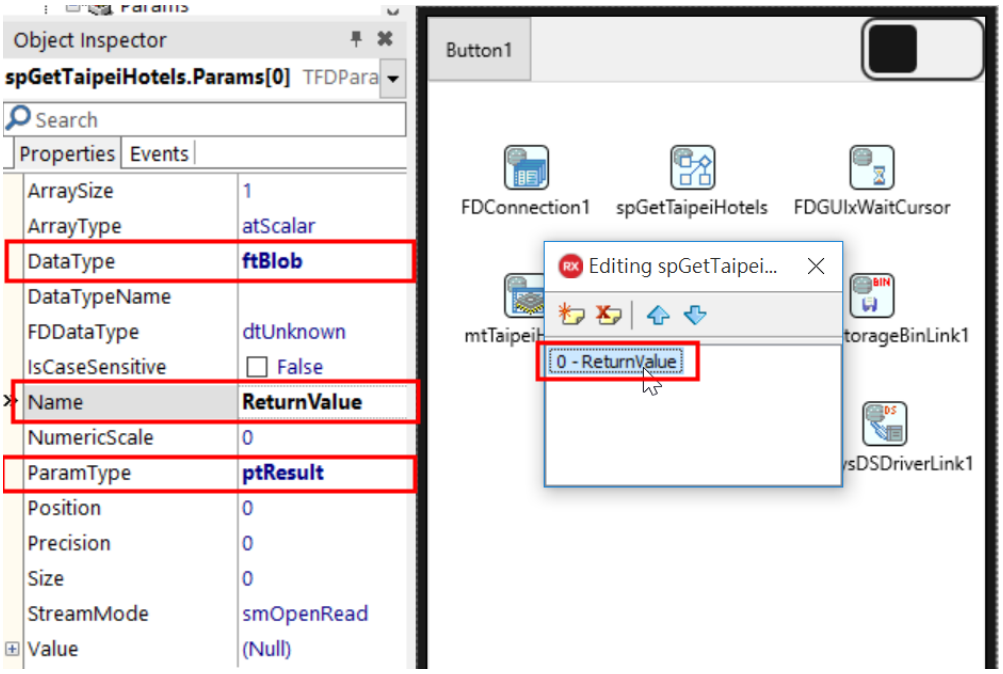
首先需要设定 FireDAC 的 TFDConnection 组件链接到 DataSnap 服务器，请右击上面的 FDConnection1 组件，在下面的对话盒中设定它使用的 Driver ID 为 DS，代表要使用 DataSnap 链接程序，再于 Protocol 字段中设定使用 TCP/IP 并于 Server 字段中设定 DataSnap 服务器的 IP 地址：



正确设定 FDConnection1 后就可以点选主窗体中的 spGetTheHotel 组件设，在对象查看器中设定它的 StoredProcName 特性，从其中就可以看到 DataSnap 服务器提供的服务方法，让我们选择要呼叫 GetTaipeiHotels()方法，如下所示：



选择了 GetTaipeiHotels()方法后 spGetTheHotel 就会取得 DataSnap 服务器 GetTaipeiHotels()方法的元数据，此时在对象查看器中点选它的 Params 特性就可以看到如下所示 GetTaipeiHotels()方法会回传一个参数，它的数据型态是 ftBlob：



现在就可以实作从客户端向 DataSnap 服务器查询数据了，在主窗体的 Button1 的 OnClick 事件中呼叫 GetTaipeiHotels()方法取得查询资料，再呼叫 ShowTaipeiHotels()方法显示数据：

```
procedure TfmMainForm.Button1Click(Sender: TObject);

begin
```

```
  GetTaipeiHotels;
  ShowTaipeiHotels;
end;
```

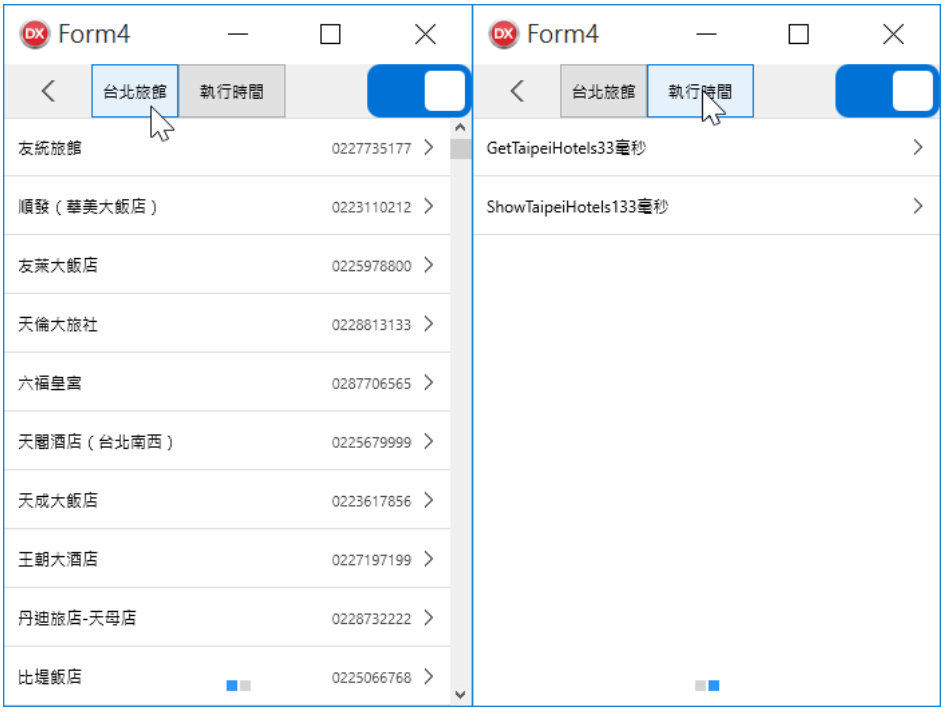
在 GetTaipeiHotels()方法 006 行只需要呼叫 spGetTaipeiHotels 的 ExecProc()方法 FireDAC 就可以呼叫远方 DataSnap 服务器中指定的服务方法(TsmFireDACDataSnapDemo1.GetTaipeiHotels)，呼叫成功之后 DataSnap 服务器回传的结果会储存在 spGetTaipeiHotels 的第 1 个参数(ReturnValue)中，而且型态是 ftBlob。因此在 007 行建立一个 TStringStream 对象并把 spGetTaipeiHotels 的第 1 个参数内容做为建构元参数，如此一来 TStringStream 对象的内容就是回传的结果，010 行把 TStringStream 对象包含的数据流位置设定到起始位置，012 行再使用 mtTaipeiHotels 组件的 LoadFromStream()方法从 TStringStream 对象中读取数据流并还原成数据集对象即可：

```
001   procedure TfmMainForm.GetTaipeiHotels;
002   var
003     LStringStream: TStringStream;
004   begin
005     lStart := Now;
006     spGetTaipeiHotels.ExecProc;
007     LStringStream :=
TStringStream.Create(spGetTaipeiHotels.Params[0].asBlob);
008     try
009       if (LStringStream <> nil) then
010       begin
011         LStringStream.Position := 0;
012         mtTaipeiHotels.LoadFromStream(LStringStream,
TFDStorageFormat.sfBinary);
013       end;
014     finally
015       LStringStream.Free;
016       lEnd := Now;
017     end;
018   end;
```

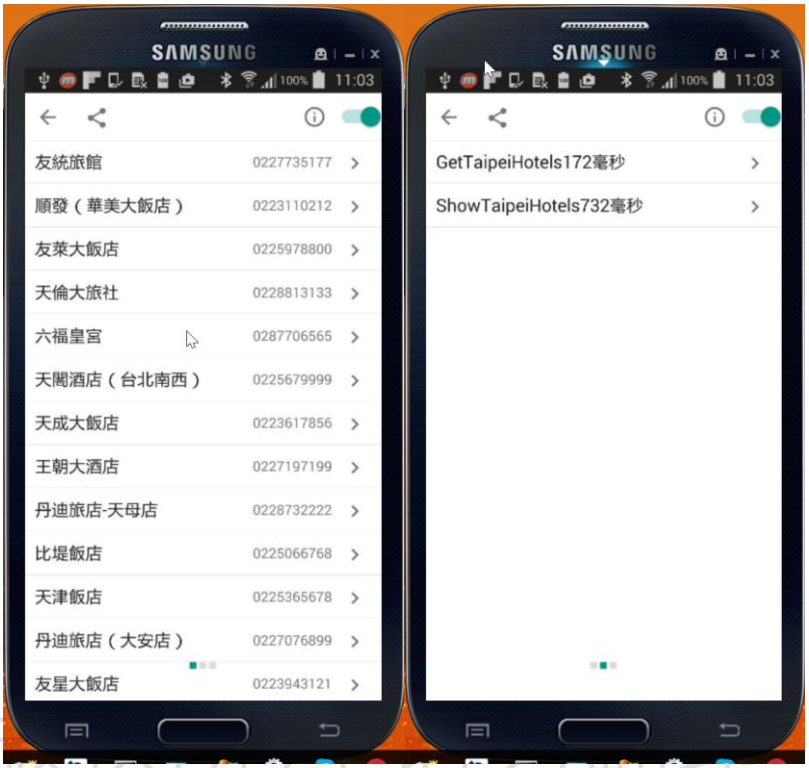

接着 ShowTaipeiHotels()方法就可以使用我们已经熟悉的方法把数据从 TFDMemTable 组件中显示在主窗体的 TListView 组件中：

```
procedure TfmMainForm.ShowTaipeiHotels;
var
  lvi : TListViewItem;
begin
  lStart := Now;
  lvTaipeiHotels.Items.Clear;
  mtTaipeiHotels.First;
  while (not mtTaipeiHotels.Eof) do
  begin
    lvi := lvTaipeiHotels.Items.Add;
    lvi.Text := mtTaipeiHotels.FieldByName('STITLE').AsString;
    lvi.Detail := mtTaipeiHotels.FieldByName('MEMO_TEL').AsString;
    mtTaipeiHotels.Next;
  end;
  lEnd := Now;
end;
```


现在如果编译并执行此范例客户端就可以看到它在笔者的 Windows 10 中以 Win32 程序执行并从 DataSnap 服务器成功查询到台北市旅馆数据：



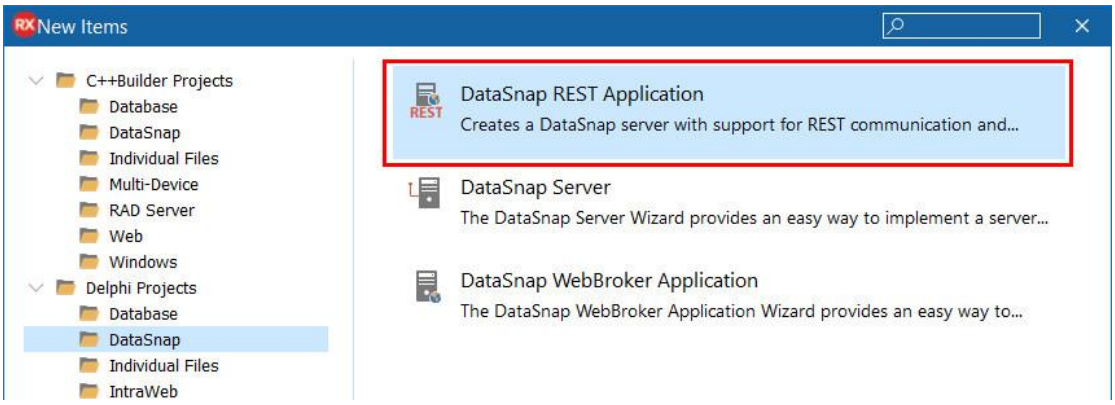
如果此时把此客户端部署到笔者的 Samsung S4 手机中可以可以看到也能正常执行，而且从笔者的虚拟 Windows 10 的 DataSnap 服务器中取得 396 笔旅馆数据也只要 172 毫秒：



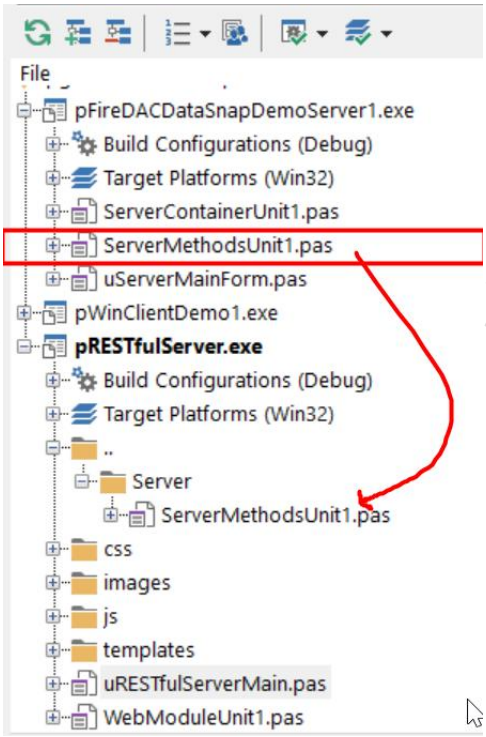
9-1-3 开发 RESTful DataSnap 服务器

使用 TCP/IP 连结 DataSnap 服务器和客户端仅限于客户端是使用 Delphi 或是 C++Builder 开发的，如果我们希望客户端可以其他语言或是工具开发的，那么我们可以使用 RESTful 架构，让 DataSnap 服务器使用 HTTP/HTTPS 通讯协议并使用 JSON 格式传递数据。

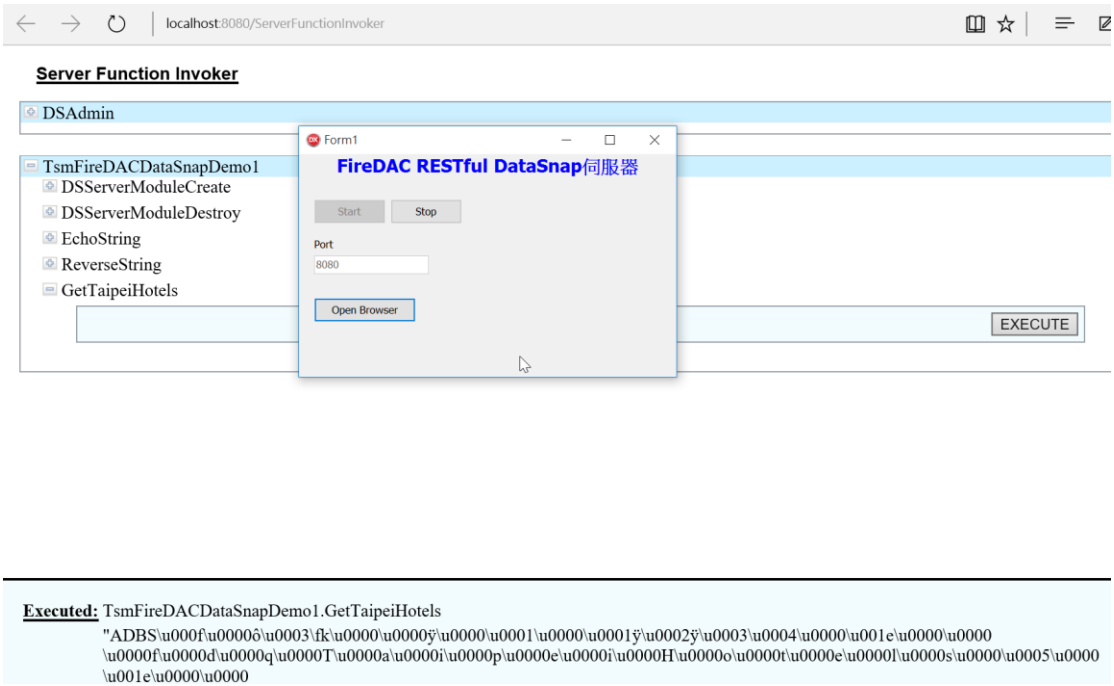
由于在前面建立 DataSnap 服务器时服务器提供的服务方法是实作在 TDataModule 中，因此也可以使用在 RESTful 的 DataSnap 服务器中。因此请在项目群组中再建立一个如下所示的 DataSnap REST Application 项目：



把此项目以 pRESTfulServer 名称储存，接着在此项目中加入前面第 1 个 DataSnap 服务器项目中的 SeverMethodsUnit1 程序单元到此新项目中：



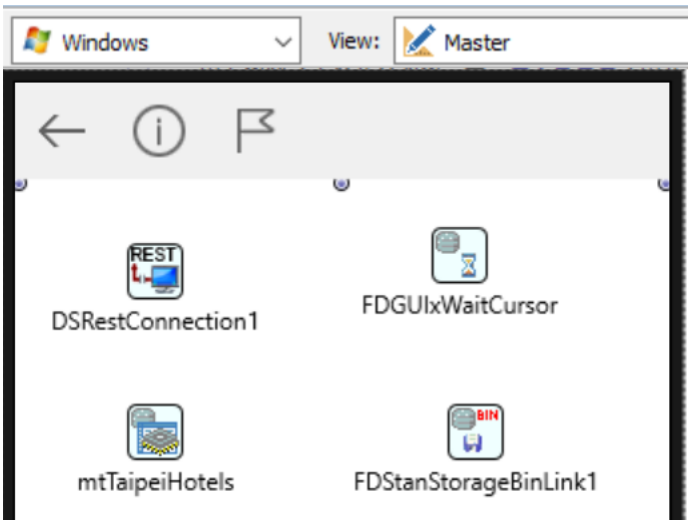
现在我们只需要编译此新项目执行即可：



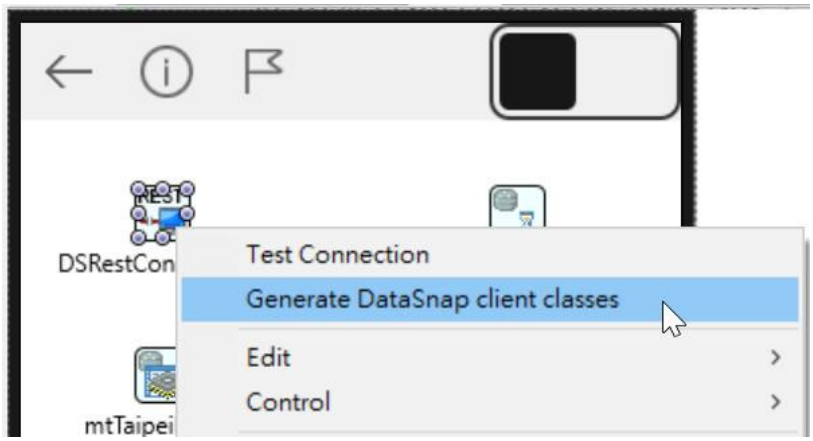
由于这是一个 RESTful DataSnap 服务器，因此从上图可以看到笔者使用 Windows 10 的 Edge 浏览器呼叫 ShowTaipeiHotels()方法并取得资料(Unicode 格式)，接下来就可以开发一个手机客户端来查询数据了。

9-1-4 开发 RESTful 手机客户端

再于项目群组中建立一个 Muli-Device Applicartion 项目，要使用 RESTful 架构连结 RESTful DataSnap 服务器，我们可以使用 TDSRestConnection 组件，因此在主窗体中加入 TDSRestConnection 和 TFDMemTable 组件：



右击 TDSRestConnection 组件在突显示选单中选择 Generate DataSnap client classe 选项：



TDSRestConnection 组件便会产生伺服端服务的类别，GetTaipeiHotels() 方法便在其中：

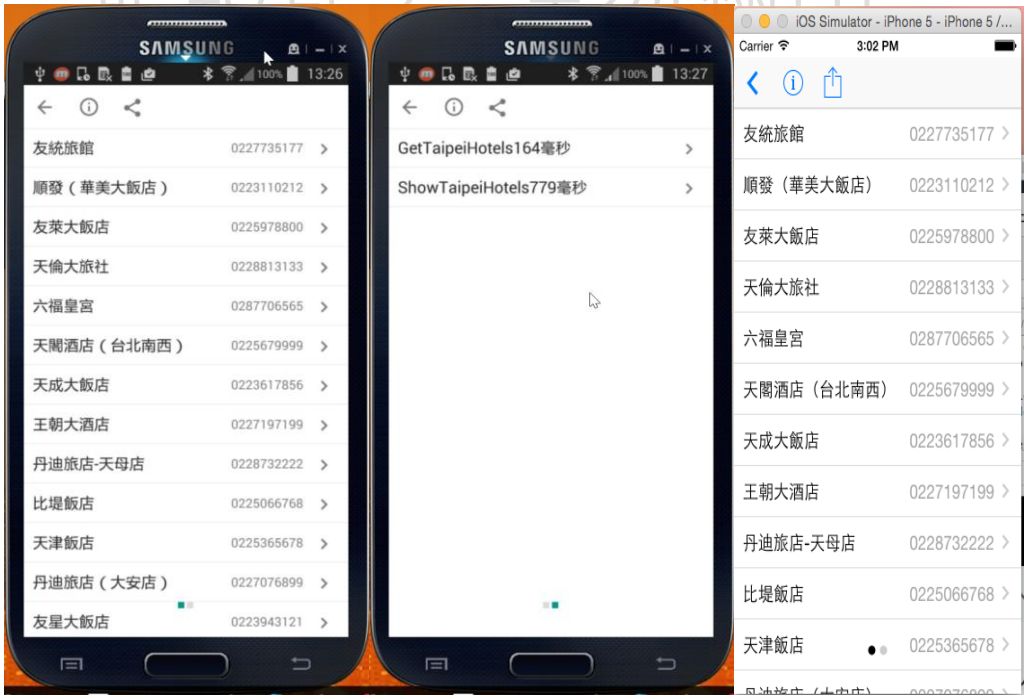
```
TsmFireDACDataSnapDemo1Client = class(TDSAdminRestClient)
private
  ...
public
  ...
  function EchoString(Value: string; const ARequestFilter: string = ''): string;
  function ReverseString(Value: string; const ARequestFilter: string = ''): string;
  function GetTaipeiHotels(const ARequestFilter: string = ''): TStream;
  function GetTaipeiHotels_Cache(const ARequestFilter: string = ''):
IDSRestCachedStream;
end;
```


接着使用如下的程序代码呼叫 RESTful DataSnap 服务器中的服务方法，请注意下面的程序代码和前面几乎一样，除了在 008 行是建立刚才自动产生的 TsmFireDACDataSnapDemo1Client 对象并把 TDSRestConnection 组件传入做为建构元的参数：

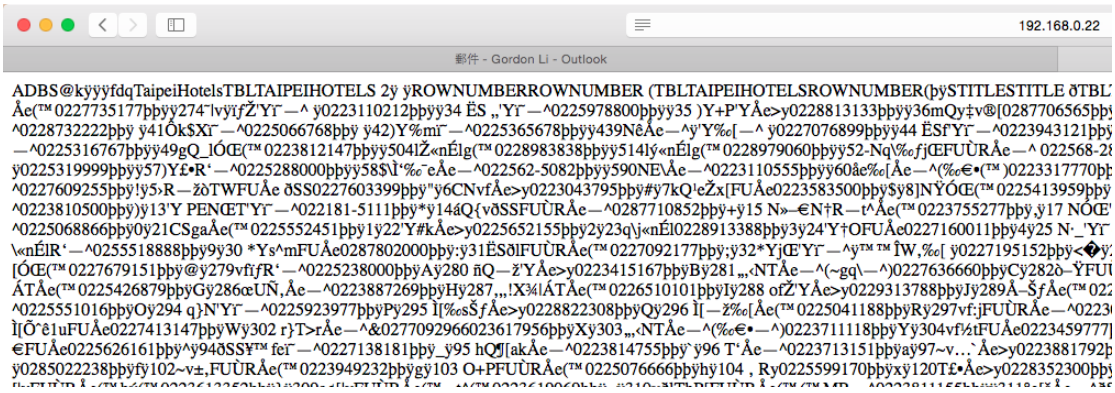
```
001   procedure TfmMainForm.GetTaipeiHotels;
002   var
003     LStream: TStream;
004     aServer : TsmFireDACDataSnapDemo1Client;
005   begin
006     lStart := Now;
```

```
007
008     aServer := TsmFireDACDataSnapDemo1Client.Create(Self.DSRestConnection1);
009     try
010       LStream := aServer.GetTaipeiHotels();
011       if (LStream <> nil) then
012       begin
013         LStream.Position := 0;
014         mtTaipeiHotels.LoadFromStream(LStream, TFDStorageFormat.sfBinary);
015       end;
016     finally
017       LStream.Free;
018       lEnd := Now;
019       aServer.Free;
020     end;
021   end;
```

编译并执行此范例 RESTful 客户端，我们可以看到下面的结果画面，这个范例 RESTful 客户端可以成功执行在 Android 和 iPhone Simulator 中：



由于这是个 RESTful 的架构，因此笔者也可以使用 Mac OSX 中的 Safari 浏览器呼叫范例 RESTful DataSnap 服务器并取得查询结果(Unicode 格式)：



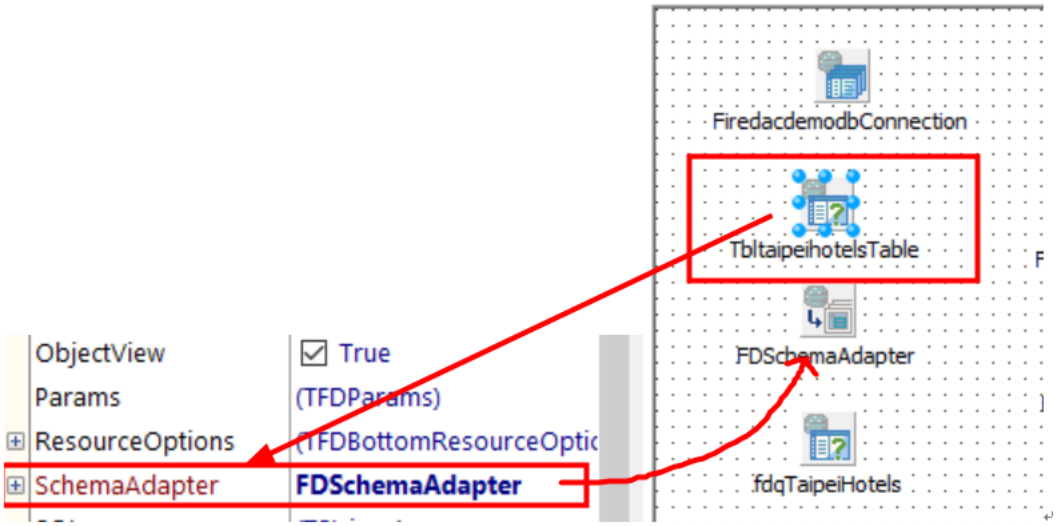
9-1-5 如何更新旅馆数据

那么是否可以在 PC 或是手机端对资料进行异动呢？当然可以，而且也很简单，接下来让我们继续修改范例让我们在手机端可修改旅馆数据。

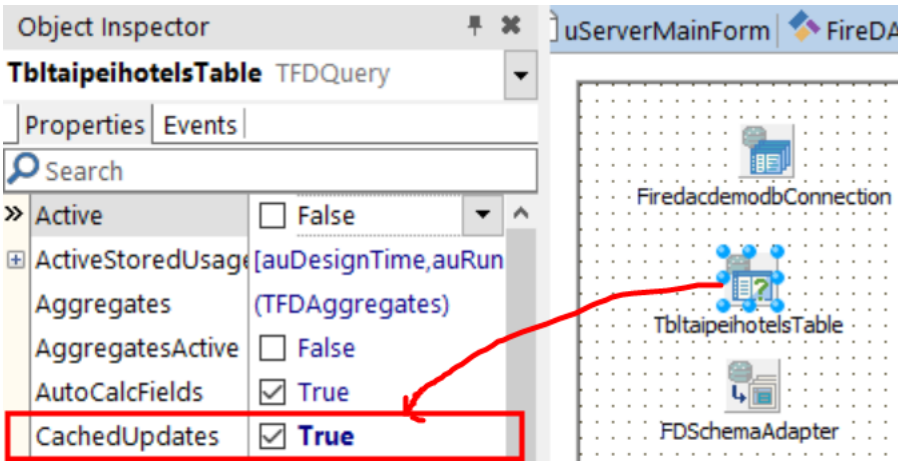
修改 FireDAC DataSnap 服务器

FiriDAC 的中央快储功能可让程序员在 DataSnap 架构中对数据进行 CRUD 的工作，现在我们想让稍后的手机端可在远程修改旅馆数据，因此我们需要先在服务器中加入可把数据异动回数据库的服务方法。

回到第 1 个范例 DataSnap 服务器在 ServerMethodsUnit1 程序单元确定 TbltaipeihotelsTable 组件的 SchemaAdapter 设定为数据模块中的 FDSchemaAdapter：



而且一定要设定 TbltaipeihotelsTable 组件使用快储功能，即设定它的 CachedUp[dates 特性值为 True：



设定 TbltaipeihotelsTable 组件的 SQL 特性值为如下的 SQL 命令：

```
SELECT * FROM TBLTAIPEIHOTELS where ROWNUMBER = :ROWNUMBER
```

现在让我们加入一个修改数据的功能，当用户查询了旅馆数据后可在 TListView 组件中点选任一旅馆然后我们允许使用修改旅馆名称，电话或是价格信息。因此在 DataSnap 服务器需要 2 个新的方法，GetTheHotel()可取得用户在 TListView 组件中点选的旅馆，而 PostHotel()方法则可把手机端异动的信息更新回数据库中：


```
  function GetTaipeiHotels: TStream;
  function GetTheHotel(const sHotelID : String) : TStream;
  function PostHotel(AStream: TStream) : Integer;
```

GetTheHotel()方法实作很简单，只是把客户端传来的旅馆 ID 带入 TbltaipeihotelsTable 组件的 SQL 特性值的参数中，再开启 TbltaipeihotelsTable 组件并存入 TMemoryStream 对象中再回传到客户端：

```
function TsmFireDACDataSnapDemo1.GetTheHotel(const sHotelID: String): TStream;
begin
  Result := TMemoryStream.Create;
  try
    TbltaipeihotelsTable.Close;
    TbltaipeihotelsTable.Params.ParamByName('ROWNUMBER').Value := sHotelID;
    TbltaipeihotelsTable.Open;
    FDSchemaAdapter.SaveToStream(Result, TFDStorageFormat.sfBinary);
    Result.Position := 0;
  except
    raise;
```

```
  end;
end;
```

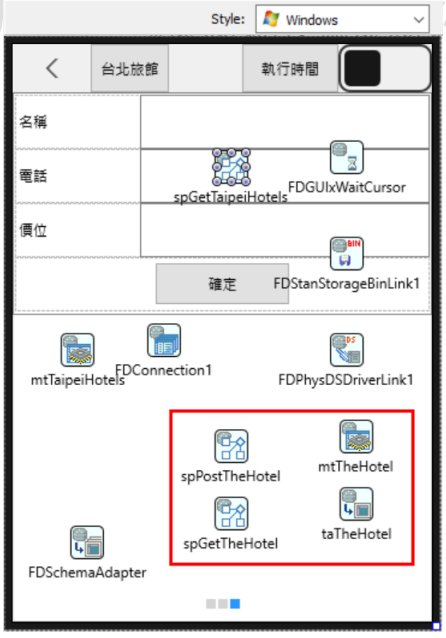
PostHotel()方法是把客户端异动的数据更新回数据库，在稍后实作手机端时会看到手机是把客户端 TFDMemTable 组件的 Delta 传回服务器，因此 PostHotel()方法先在 027 行呼叫 CopyStream()方法把客户端传来的数据流先拷贝到 TMemoryStream 对象中，028 行重置数据流位置到起始位置，031 行藉由 TFDSchemaAdapter 类别的 LoadFromStream()方法把数据流还原到 TbltaipeihotelsTable 组件中，最后在 032 行呼叫 TFDSchemaAdapter 类别的 ApplyUpdates()方法，TFDSchemaAdapter 即会根据传来的 Delta 和元资料自动产生 SQL 命令把异动的数据更新回数据库：

```
001   function CopyStream(const AStream: TStream): TMemoryStream;
002   var
003     LBuffer: TBytes;
004     LCount: Integer;
005   begin
006     Result := TMemoryStream.Create;
007     try
008       SetLength(LBuffer, 1024 * 32);
009       while True do
010       begin
011         LCount := AStream.Read(LBuffer, Length(LBuffer));
012         Result.Write(LBuffer, LCount);
013         if LCount < Length(LBuffer) then
014           break;
015       end;
016     except
017       Result.Free;
018       raise;
019     end;
020   end;
021
022   function TsmFireDACDataSnapDemo1.PostHotel(AStream: TStream) : Integer;
023   var
024     LMemStream: TMemoryStream;
025     LErrors: Integer;
026   begin
```

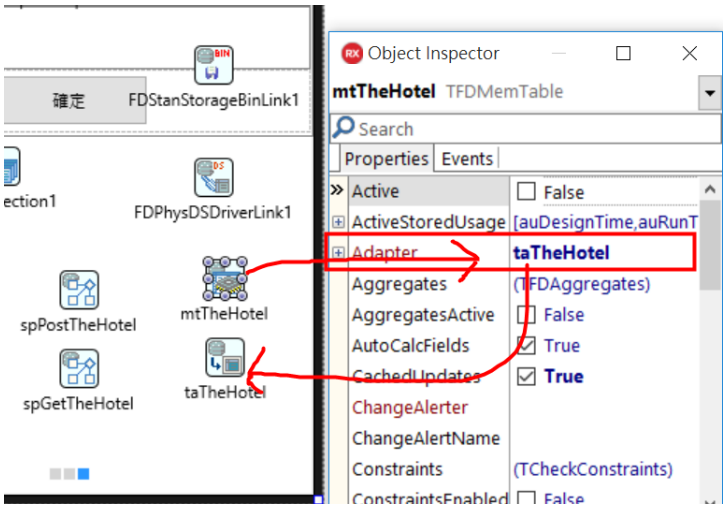
```
027     LMemStream := CopyStream(AStream);
028     LMemStream.Position := 0;
029
030     try
031       FDSchemaAdapter.LoadFromStream(LMemStream, TFDStorageFormat.sfBinary);
032       Result := FDSchemaAdapter.ApplyUpdates;
033     finally
034       LMemStream.Free;
035     end;
036   end;
```

修改 FireDAC DataSnap 手机客户端

再回到前面第 2 个范例客户端项目，在主窗体中加入 spGetTheHotel 组件呼叫服务器中的 GetTheHotel() 方法取得使用者点选的旅馆并还原在 mtTheHotel 组件中。而 taTheHotel 是 TFDTableAdapter 组件，它必须链接到 taTheHotel 组件以使用中央快储功能，最后 spPostTheHotel 组件则是把客户端异动的数据藉由呼叫服务器中的 PostHotel 方法更新数据：



接着在对象查看器中设定 taTheHotel 的 Adapter 特性值为 taTheHotel 组件：



现在就可以开始实作客户端程序代码，首先在 TListView 组件的 OnItemClick 事件中呼叫 GetTheHotel()方法向服务器查询用户在 TListView 组件中点选的旅馆，再呼叫 DisplayTheHotel()方法把点选的旅馆显示出来并准备编辑：

```
001   procedure TfmMainForm.lvTaipeiHotelsItemClick(const Sender: TObject;
002     const AItem: TListViewItem);
003   begin
004     if (GetTheHotel(AItem.Text) ) then
005     begin
006       DisplayTheHotel;
007       EditTheHotel;
008     end;
009   end;
```


由于 GetTheHotel()方法使用了和前面说明 GetTaipeiHotels()方法一样的技巧，因此就不再赘述了，请读者自行参考范例程序代码。

这里的重点是当用户修改完旅馆数据并点选主窗体中的”确定”按钮时，它呼叫 PostTheHotel()方法把数据更新回数据库：

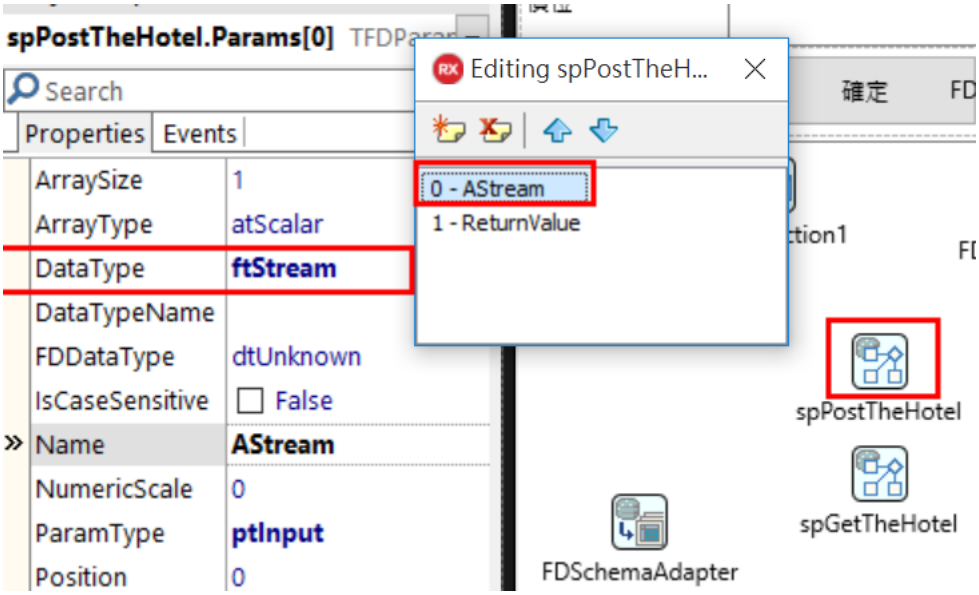
```
procedure TfmMainForm.Button4Click(Sender: TObject);
begin
  PostTheHotel;
end;
```

PostTheHotel()方法先把用户输入的数据更新回 mtTheHotel 组件中，再于 016 行建立一个 TMemoryStream 对象，018 行非常重要，它设定

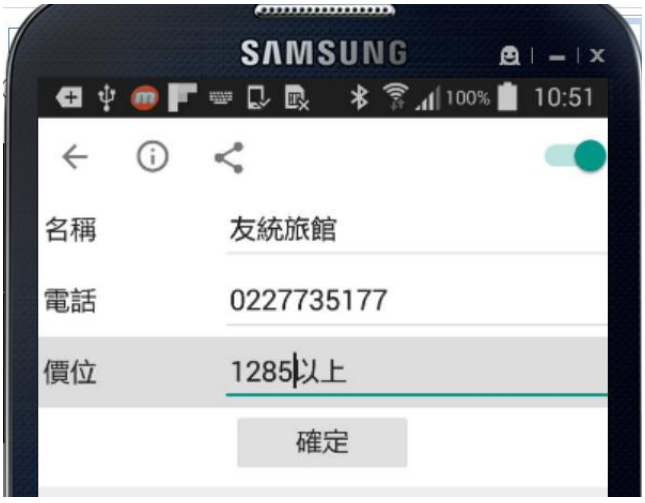
FDSchemaAdapter 的 ResourceOptions. StoreItems 为[siDelta, siMeta]，这代表在客户端是把 mtTheHotel 组件的 Delta 回传，而 siMeta 代表要储存数据集的元资料，以便服务器中的 FDSchemaAdapter 组件可以根据元数据来产生更新数据的 SQL 命令。接着把 TMemoryStream 对象中的数据流写入 spPostTheHotel 组件的参数中，最后藉由 spPostTheHotel 组件呼叫服务器中的 PostHotel()方法：

```
001   procedure TfmMainForm.PostTheHotel;
002   var
003     LMemStream: TMemoryStream;
004     I: integer;
005     LDataSet: TDataSet;
006   begin
007     lStart := Now;
008
009     mtTheHotel.Edit;
010     mtTheHotel.FieldByName('ROWNUMBeR').Value :=
mtTheHotel.FieldByName('ROWNUMBeR').Value;
011     mtTheHotel.FieldByName('STITLE').Value := edtTitle.Text;
012     mtTheHotel.FieldByName('MEMO_TEL').Value := edtPhone.Text;
013     mtTheHotel.FieldByName('MEMO_COST').AsString := edtPrice.Text;
014     mtTheHotel.Post;
015
016     LMemStream := TMemoryStream.Create;
017     try
018       FDSchemaAdapter.ResourceOptions.StoreItems := [siDelta, siMeta];
019       FDSchemaAdapter.SaveToStream(LMemStream, TFDStorageFormat.sfBinary);
020       LMemStream.Position := 0;
021       spPostTheHotel.Params[0].asStream:= LMemStream;
022       spPostTheHotel.ExecProc;
023     except
024       On E: Exception do
025         raise Exception.Create(E.Message);
026     end;
027     lEnd := Now;
028   end;
```

在这里要注意的是spPostTheHotel组件的第1个参数的数据型态一定要设定为 ftStream 如下所示，否则数据集无法以正确的格式回传回服务器，如此一来也就无法正确更新回数据库中。



现在就可以再次编译手机客户端执行，从下图中可以看到我们在 S4 中点选了”友统旅馆”，并在编辑页面中修改了它的价位数据：



再点选”确定”按钮把数据更新回去，从下图可以看到速度很理想：

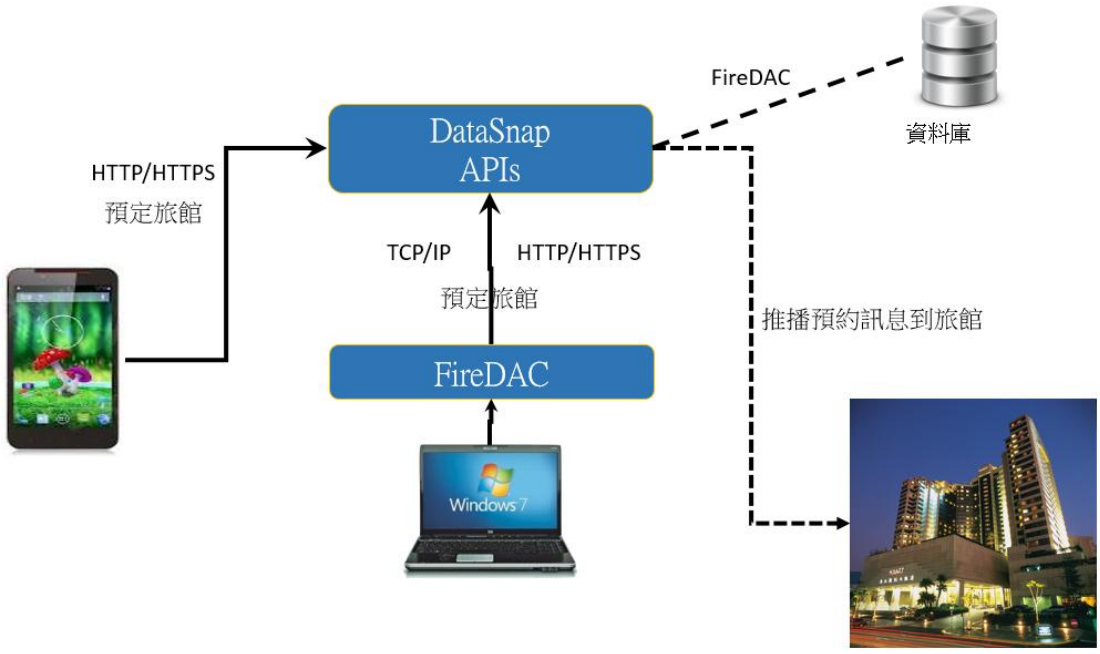
而且异动的数据果然成功的由 DataSnap 服务器藉由 FireDAC 的中央快储功能更新回 InterBase 了：

ROWNUMBER	STITLE	MEMO_COST
33	友統旅館	1285以上
47	友華賓館（貝斯特旅店）	1000以上
34	友萊大飯店	1550以上
38	天成大飯店	4500以上
42	天津飯店	1500以上
35	天倫大旅社	1000以上
57	天閣酒店	7600以上

9-2 开发可异动多数据表的 FireDAC DataSnap 系统

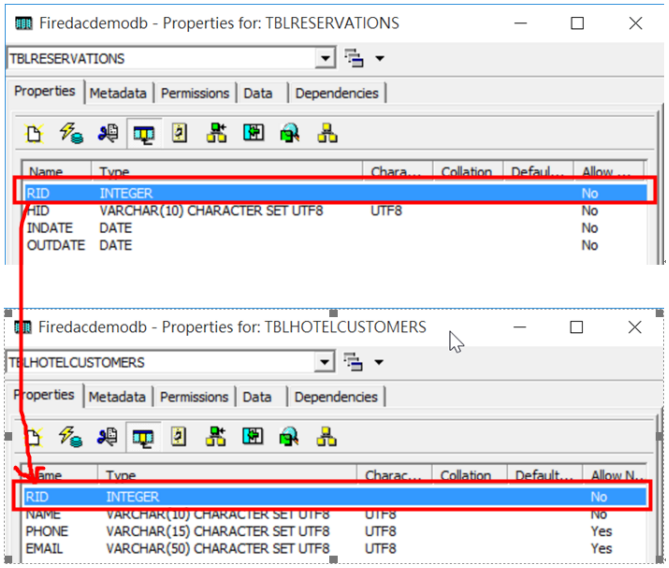
在 9-1 小节中本书说明了如何在 FireDAC 的 DataSnap 架构中更新一个数据表的数据，在本小节将说明如何在 FireDAC 的 DataSnap 架构中更新 Master/Detail 的数据。让我们试着在前面的范例 DataSnap 架构中加入可让使用者在查询到想要的旅馆后可使用手机预定旅馆的应用。

例如下图展示了本小节将实作的架构，我们甚至可以再结合推播功能把 DataSnap 服务器中预定的数据直接推播到预定旅馆的实际信息系统：

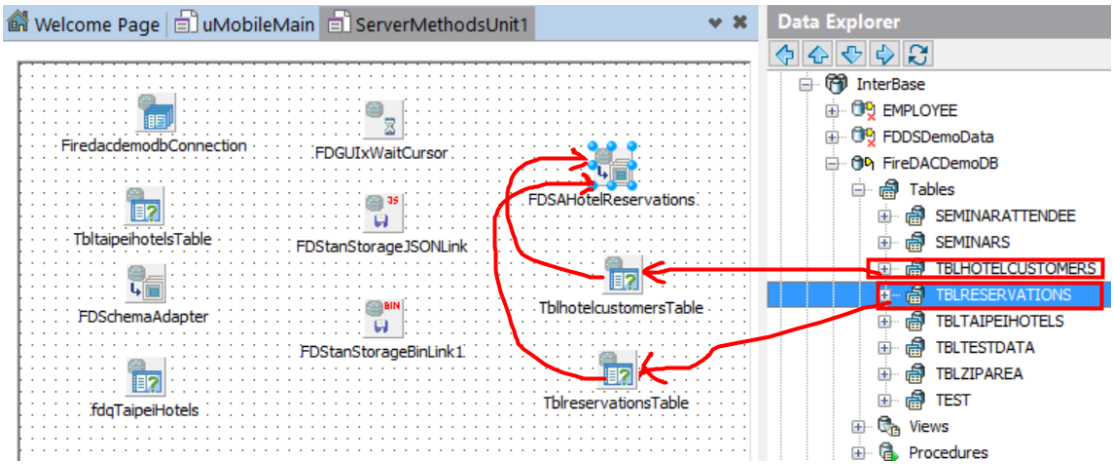


9-2-1 修改 FireDAC DataSnap 服务器

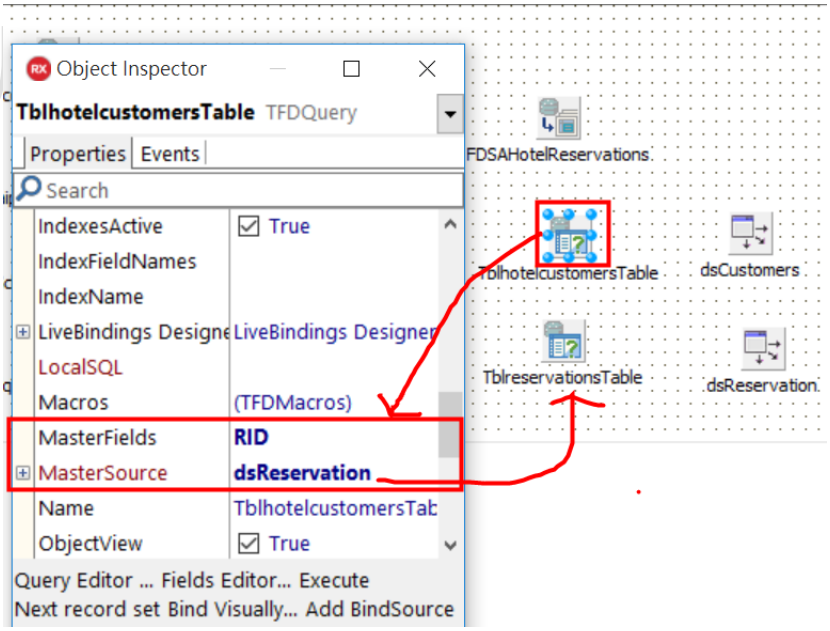
在范例数据库中 FIREDACDEMODB.GDB 中有 2 个简单的范例数据表 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS，TBLRESERVATIONS 将储存预定旅馆信息而 TBLHOTELCUSTOMERS 则储存预定客户数据，这 2 个数据表是藉由 RID 这个字段关连，如下所示：



因此请回到范例 DataSnap 服务器项目，开启 ServerMethodsUnit1 程序单元，放入 TFDSchemaAdapter 组件 FDSAHotelReservations，再从 Data Explorer 中拖曳这 2 个数据表到其中，并设定拖入的 2 个 TFDQuery 组件的 SchemaAdapter 特性值为 FDSAHotelReservations，如下所示：



再放入 2 个 TDataSource 组件分别链接到拖入的 2 个 TFDQuery 组件，并把 TblhotelcustomersTable 组件的 MasterSource 和 MasterFields 设定如下以便和 TblreservationsTable 形成 Master/Detail 的关系，如下所示：



由于在稍后使用 FireDAC 的中央快储功能时我们只是希望 FireDAC 把这 2 个数据表的元信息传递到客户端以便客户端的 TFDMemoryTable 能够在客户端新增数据，因此为了避免把所有这 2 个数据表的预定数据传递到客户端，我们在 TblreservationsTable 的 SQL 特性值中使用有参数的 SQL 命令：

```
SELECT * FROM TBLRESERVATIONS where RID = :RID
```

接着在服务器加入如下的 2 个服务方法：

```
function PostHotelReservation(AStream: TStream) : Integer;
function GetReservation(const iRID : Integer) : TStream;
```

GetReservation()方法的目的是让客户端呼叫并把 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 这 2 个数据表的元信息传递回客户端，在稍后实作客户端会看到客户端会传递 -1 给 GetReservation() 方法让 TblreservationsTable 不会把任何预定数据传回而只是传回元信息：

```
001   function TsmFireDACDataSnapDemo1.GetReservation(const iRID : Integer):
TStream;
002   begin
003     Result := TMemoryStream.Create;
004     try
005       TblreservationsTable.Close;
006       TblreservationsTable.Params[0].Value := iRID;
007       TblreservationsTable.Open;
008       TblhotelcustomersTable.Close;
009       TblhotelcustomersTable.Open;
010       FDSAHotelReservations.SaveToStream(Result, TFDStorageFormat.sfBinary);
011       Result.Position := 0;
012     except
013       raise;
014     end;
015   end;
```


PostHotelReservation()方法则是把客户端传递来的预定数据藉由中央快储功能把预定数据更新回 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 这 2 个数据表。要如此做很简单，由于 TblreservationsTable 和 TblhotelcustomersTable 组件把经链接到 FDSAHotelReservations，因此只需要在下面的 008 行呼叫 FDSAHotelReservations 的 LoadFromStream()方法把客户端传来的异动数据 (Delta) 分别读入 TblreservationsTable 和 TblhotelcustomersTable 组件，再于 009 行呼叫 FDSAHotelReservations 的 ApplyUpdates() 方法就可以把数据更新回 TblreservationsTable 和 TblhotelcustomersTable 组件链接的 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 这 2 个数据表：

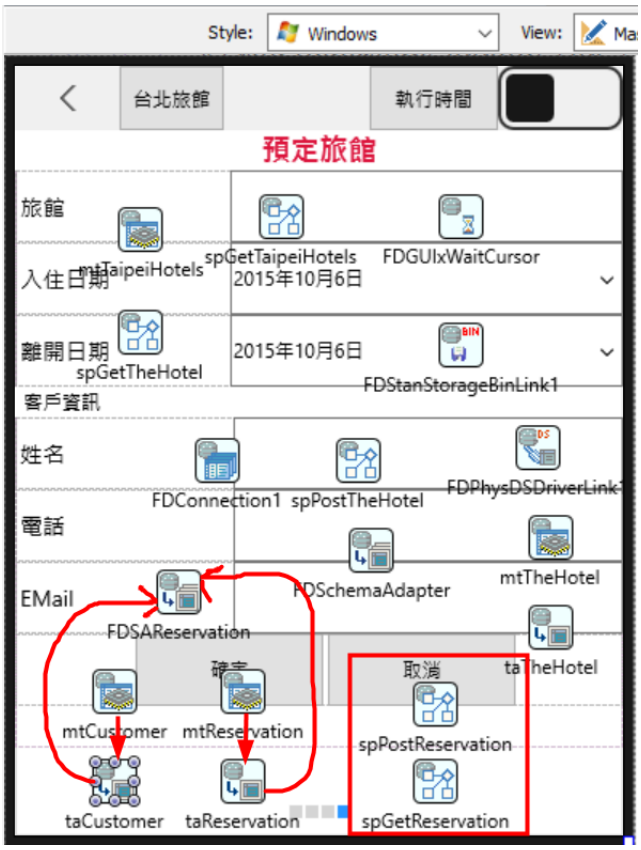
```
001   function TsmFireDACDataSnapDemo1.PostHotelReservation(AStream: TStream):
Integer;
```

```
002   var
003     LMemStream: TMemoryStream;
004   begin
005     LMemStream := CopyStream(AStream);
006     LMemStream.Position := 0;
007     try
008       FDSAHotelReservations.LoadFromStream(LMemStream,
TFDStorageFormat.sfBinary);
009       Result := FDSAHotelReservations.ApplyUpdates;
010     finally
011       LMemStream.Free;
012     end;
013   end;
```

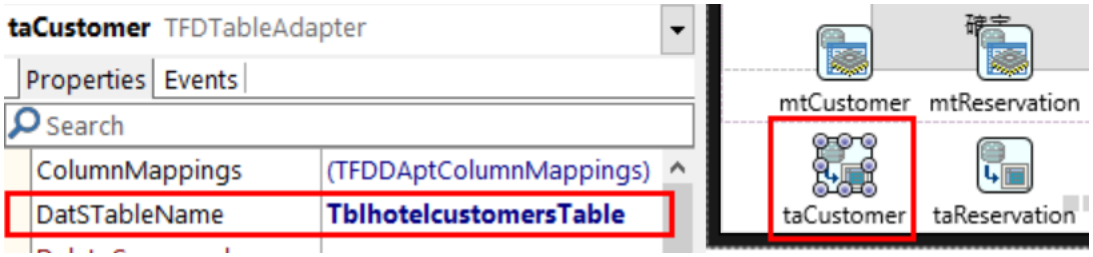
请重新编译并执行此范例 DataSnap 服务器。

9-2-2 修改手机客户端

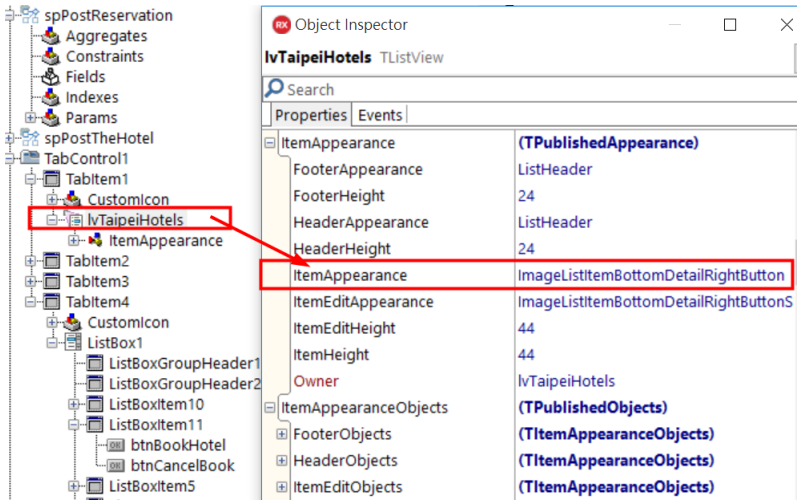
回到范例客户端，设计如下可输入预定信息的 UI，并放入 FDSAReservation 这个 TFDSchemaAdapter 组件，并让 2 个 TFDMemTable 组件分别链接到 2 个 TFDTableAdapter 组件，再把 2 个 TFDTableAdapter 组件链接到 FDSAReservation。而 spGetReservation 是准备呼叫服务器的 GetReservation() 方法而 spPostReservation 则是呼叫服务器的 PostHotelReservation()方法：



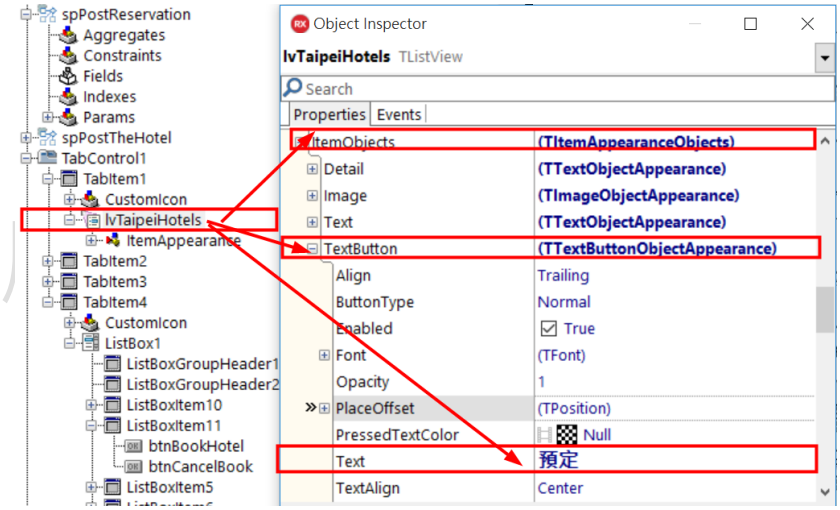
上面的 taCustomer 和 taReservation 的 DatSTableName 特性值必须设定为服务器 ServerMethodsUnit1 程序单元中的 2 个 TFDQuery 组件的名称：



接着修改主窗体中的 TListView 组件，设定它的 ItemAppearance 子特性值为 ImageListItemBottomDetailRightButton 以便加入一个点选按钮：



再设定它的 ItemObjects.TextButton.Text 子特性值为”预定”：



在新加入的”确定”按钮中实作如下的程序代码：

```
procedure TfmMainForm.lvTaipeiHotelsButtonClick(const Sender: TObject;
  const AItem: TListItem; const AObject: TListItemSimpleControl);
begin
  if (FListItemTextButtonClicked) then
  begin
    DoBookHotel(lvTaipeiHotels.Items[AItem.Index].Text);
    TabControl1.ActiveTab := TabItem4;
  end;
end;
```

DoBookHotel()方法呼叫 GetReservation()方法从 DataSnap 服务器取得 TBLRESERVATIONS 和 TBLHOTELCUSTOMERS 这 2 个数据表的元数据以

便据此设定客户端的 2 个 TFDMemTablem 组件。PostDataToMTable()方法把用户输入的预定信息更新到客户端的 TFDMemTablem 组件中，最后 PostHotelReservation() 方法呼叫 DataSnap 服务器的 PostHotelReservation()服务方法把预定信息写入数据库中：

```
procedure TfmMainForm.btnBookHotelClick(Sender: TObject);
begin
  GetReservation;
  PostDataToMTable;
  PostHotelReservation;
end;
```

GetReservation()方法使用的技巧在前而已经说明过，由于它的目的只是取得元数据，因此在 006 行传入-1 以避免取得任何真的的预定资料：

```
01   procedure TfmMainForm.GetReservation;
002   var
003     LStringStream: TStringStream;
004   begin
005     lStart := Now;
006     spGetReservation.Params[0].Value := -1;
007     spGetReservation.ExecProc;
008     LStringStream := TStringStream.Create(spGetReservation.Params[1].asBlob);
009     try
010       if (LStringStream <> nil) then
011       begin
012         LStringStream.Position := 0;
013         FDSAReservation.LoadFromStream(LStringStream,
TFDStorageFormat.sfBinary);
014       end;
015     finally
016       LStringStream.Free;
017       lEnd := Now;
018     end;
019   end;
```

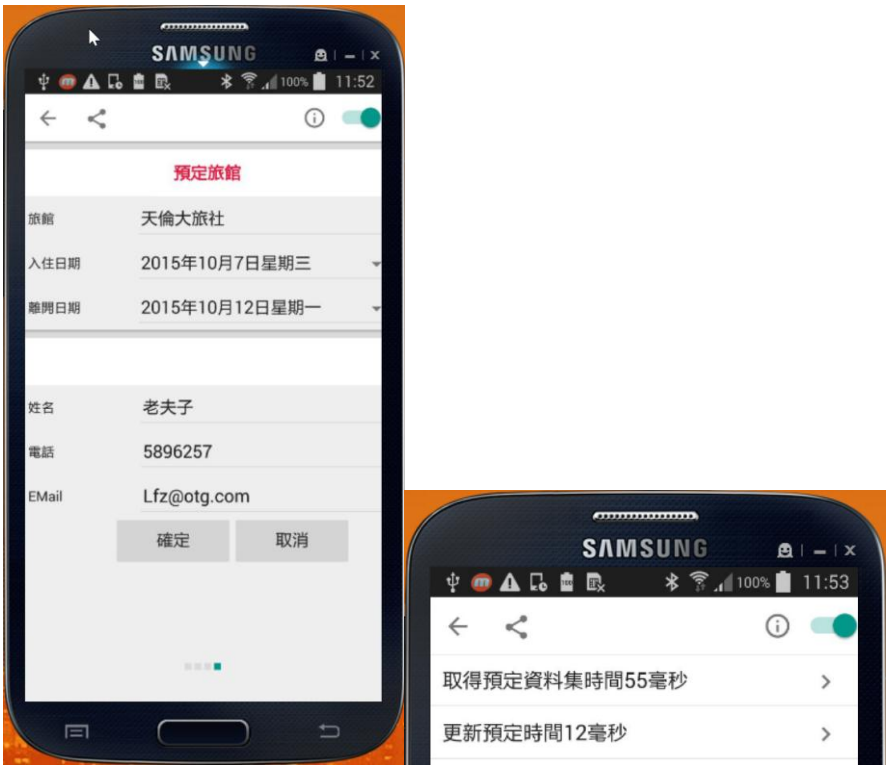

在上面的 013 行执行完毕后，客户端的 2 个 TFDMemTablem 组件就设好元数据(字段架构和信息)。

PostHotelReservation()方法在 008~014 行是确定它链接的 2 个 TFDMemTablem 组件完成写入预定数据的动作，016~027 行藉 spPostReservation 组件呼叫 DataSnap 服务器的 PostHotelReservation()方法把数据写入数据库中：

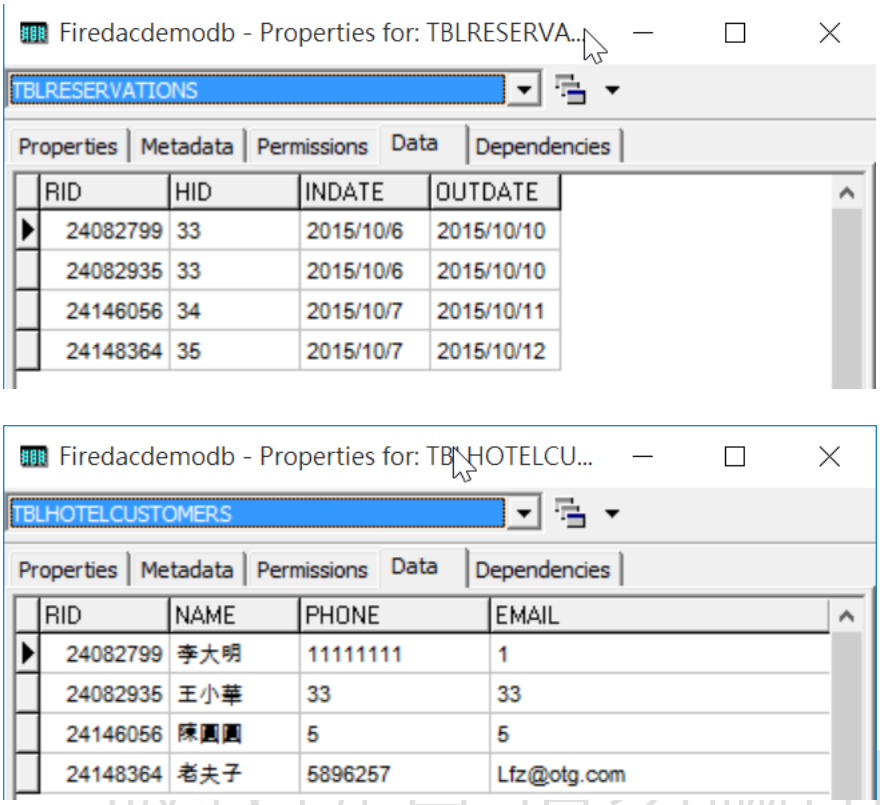
```
001   procedure TfmMainForm.PostHotelReservation;
002   var
003     LMemStream: TMemoryStream;
004     I: integer;
005     LDataSet: TDataSet;
006   begin
007     lStart := Now;
008     for I := 0 to FDSAReservation.Count - 1 do
009     begin
010       LDataSet := FDSAReservation.DataSets[I];
011       if LDataSet <> nil then
012         if LDataSet.State in dsEditModes then
013           LDataSet.Post;
014     end;
015
016     LMemStream := TMemoryStream.Create;
017     try
018       FDSAReservation.ResourceOptions.StoreItems := [siDelta, siMeta];
019       FDSAReservation.SaveToStream(LMemStream, TFDStorageFormat.sfBinary);
020       LMemStream.Position := 0;
021       spPostReservation.Params[0].asStream:= LMemStream;
022       spPostReservation.ExecProc;
023     except
024       On E: Exception do
025         raise Exception.Create(E.Message);
026     end;
027     lEnd := Now;
028   end;
```


现在编译并执行客户端，下图是范例 App 在 S4 手机中执行的画面，点选任一旅馆右方的”预定”按钮：

可以在下方左图的 UI 输入预定信息再点选”确定”按钮后就可以把数据藉由范例 DataSnap 服务器写回 InterBase 数据库中，从下方右图可以看到执行速度非常理想，在客户端从范例 DataSnap 服务器取得元数据只需要 55 毫秒，从客户端把输入的预定数据藉由范例 DataSnap 服务器写回 InterBase 数据库只需要 12 毫秒：



从下方 InterBase 数据库的管理工具 IBConsole 中可以看到资料果然正确写入到了 InterBase 数据库中：



从本小节的说明中可以看到使用 FireDAC 的中央快储功能在 DataSnap 架构中处理多数据表数据异动的应用时非常的方便。

9-2-3 使用 JSON 更新数据

在上一小节中使用的技巧是 FireDAC 的中央快储功能，除此之外也很有很多其他的方法，例如直接使用 JSON。我们可以在客户端根据用户输入的预定数据建立一个 JSON 对象然后传递给范例 DataSnap 服务器，再由范例 DataSnap 服务器解析其中的数据再写回 InterBase 数据库。本小节就说明如何使用 JSON 的方式把客户端预定数据更新回数据库。

让我们把客户端的预定数据封装成一个 JSON 对象，并把预定客户封装成一个 JSON 数组内嵌在 JSON 对象中，例如如下的一个 JSON 对象范例：

```
{"RID":"24172071","HID":"36","INDATE":"2015/10/7 下午
06:27:12","OUTDATE":"2015/10/7 下午 06:27:12","客户":["李
JSON","0933000000","ljson@hotmail.com"]}
```

首先在范例 DataSnap 服务器中加入一个新的方法 PostHotelReservationByJSON()：

```
function PostHotelReservationByJSON(const sData : String) : Boolean;
```

它接受一个字符串的参数，此字符串参数的内容就如上面说明的 JSON 对象。

PostHotelReservationByJSON()方法需要解析传入的 JSON 对象并取得正确的数据更新回数据库中。因此 056~088 行就是藉由新的 JSON 框架解析 SON 对象并新增数据到 TblreservationsTable 和 TblhotelcustomersTable 组件中：

```
001   function TsmFireDACDataSnapDemo1.PostHotelReservationByJSON(const sData :
String): Boolean;
002   var
003     sr : TStringReader;
004     jr : TJSONTextReader;
005     iPos : Integer;
006     iRID : Integer;
007
008     procedure HandleReservationField;
009     var
010       sField : String;
011       dt : TDateTime;
012     begin
013       sField := jr.Value.ToString;
014       if (sField = 'RID') then
015       begin
016         iRID := jr.ReadAsInteger;
017         TblreservationsTable.FieldByName(sField).Value := iRID;
018       end
019       else
020         if (sField = 'HID') then
021           TblreservationsTable.FieldByName(sField).Value := jr.ReadAsString
022         else
023         begin
```

```
024          if ( (sField = 'INDATE') or (sField = 'OUTDATE') ) then
025          begin
026            dt := StrToDateTime(jr.ReadAsString);
027            TblreservationsTable.FieldByName(sField).Value := dt;
028          end;
029        end;
030      end;
031
032      procedure HandleCustomerField;
033      var
034        sFieldValue : String;
035      begin
036        sFieldValue := jr.Value.ToString;
037        case iPos of
038          0 :
039            TblhotelcustomersTable.FieldByName('NAME').Value := sFieldValue;
040          1 :
041            TblhotelcustomersTable.FieldByName('PHONE').Value := sFieldValue;
042          2 :
043            TblhotelcustomersTable.FieldByName('EMAIL').Value := sFieldValue;
044        end;
045      end;
046
047    begin
048      Result := True;
049      FiredacdemodbConnection.StartTransaction;
050
051      if (not TblreservationsTable.Active) then
052        TblreservationsTable.Open();
053      if (not TblhotelcustomersTable.Active) then
054        TblhotelcustomersTable.Open();
055
056      try
057          sr := TStringReader.Create(sData);
058          jr := TJSONTextReader.Create(sr);
059          TblreservationsTable.Insert;
060          try
061            jr.Rewind;
```

```
062          while (jr.Read) do
063          begin
064            case jr.TokenType of
065              TJsonToken.PropertyName:
066              begin
067                HandleReservationField;
068              end;
069              TJsonToken.StartArray:
070              begin
071                TblhotelcustomersTable.Insert;
072                TblhotelcustomersTable.FieldByName('RID').Value := iRID;
073                iPos := 0;
074                while (jr.Read) do
075                begin
076                  case jr.TokenType of
077                    TJsonToken.String:
078                      HandleCustomerField;
079                  end;
080                  Inc(iPos);
081                end;
082              end;
083            end;
084          end;
085        finally
086          sr.Free;
087          jr.Free;
088        end;
089
090      TblhotelcustomersTable.Post;
091      TblreservationsTable.Post;
092      TblhotelcustomersTable.ApplyUpdates(0);
093      TblreservationsTable.ApplyUpdates(0);
094      FiredacdemodbConnection.Commit;
095    except
096      FiredacdemodbConnection.Rollback;
097      Result := False;
098    end;
```

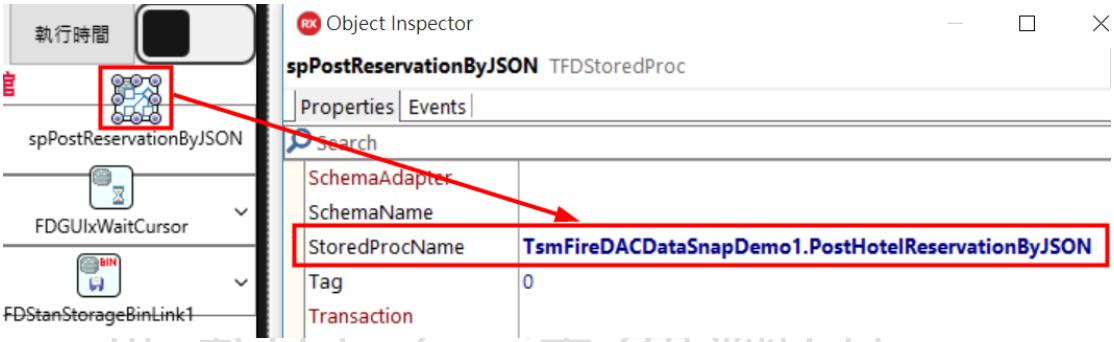
```
099    end;
```

090~099 行呼叫 2 个组件的 ApplyUpdates()方法实际把数据写回数据库中。

请再次编译和执行此范例 DataSnap 服务器。

修改范例客户端

在客户端加入一个新 TFDStoredProc 组件 spPostReservationByJSON 并在对象查看器中设定它呼叫范例 DataSnap 服务器的 PostHotelReservationByJSON()方法：



再加入一个新的” 确定(JSON)”按钮并实作如下的程序代码：

```
procedure TfmMainForm.btnPostByJSONClick(Sender: TObject);
var
  sJSONData : String;
begin
  sJSONData := CreateReservationJSONObject;
  PostHotelReservationByJSON(sJSONData);
end;
```

它先呼叫 CreateReservationJSONObject()方法根据用户输入的数据建立如前所述的 JSON 对象并储存在 sJSONData 中，再呼叫 PostHotelReservationByJSON()方法传递给范例 DataSnap 服务器。

CreateReservationJSONObject()方法使用新的 JSON 框架建立 JSON 对象(请参考”Delphi 开发手册”一书)：

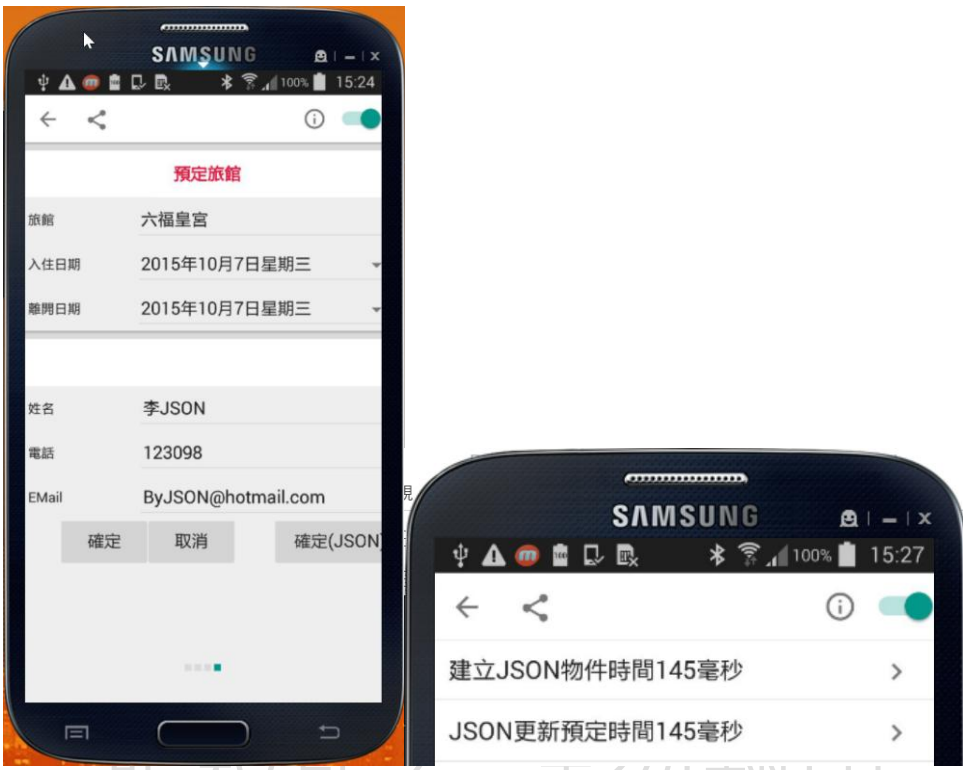
```
001    function TfmMainForm.CreateReservationJSONObject : String;
```

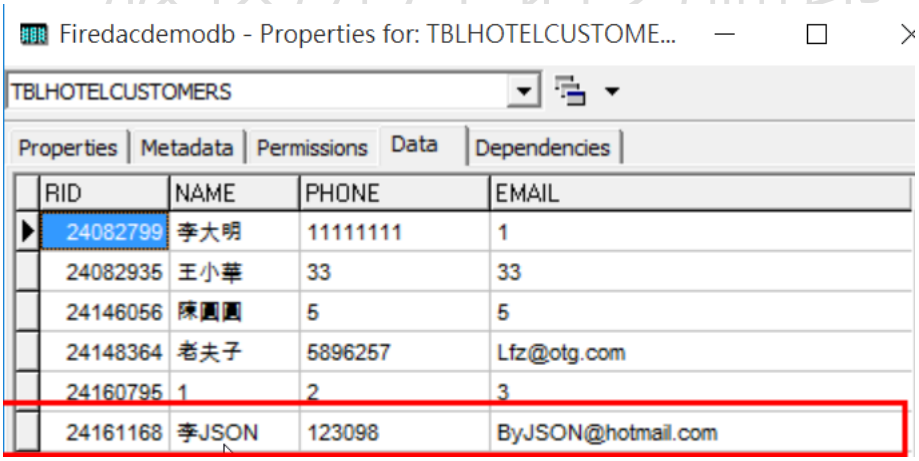
```
002   var
003     sw : TStringWriter;
004     jtw : TJsonTextWriter;
005     joBuilder : TJSONObjectBuilder;
006     sRowNumber : String;
007   begin
008     sRowNumber := GetHotelRowNumber(edtBookName.Text);
009     sw := TStringWriter.Create;
010     jtw := TJsonTextWriter.Create(sw);
011     joBuilder := TJSONObjectBuilder.Create(jtw);
012     try
013       joBuilder.BeginObject
014        .Add('RID', SecondOfTheYear(Now).ToString())
015        .Add('HID',sRowNumber)
016        .Add('INDATE', DateTimeToStr(dedtIn.DateTime))
017        .Add('OUTDATE', DateTimeToStr(dedtOut.DateTime))
018        .BeginArray('客户')
019          .Add(edtCustomerName.Text)
020          .Add(edtCustomerPhone.Text)
021          .Add(edtCustomerEMail.Text)
022        .EndAll;
023     finally
024       Result := sw.ToString;
025       joBuilder.Free;
026       jtw.Free;
027       sw.Free;
028     end;
029   end;
```


PostHotelReservationByJSON()方法非常简单，只是执行 ExecProc()方法把 JSON 对象递给范例 DataSnap 服务器：

```
procedure TfmMainForm.PostHotelReservationByJSON(const sJSONData : String);
begin
  spPostReservationByJSON.Params[0].AsString:= sJSONData;
  spPostReservationByJSON.ExecProc;
end;
```

编译和执行此范例手机 App，从下图 S4 手机画面可看到客户端成功藉由使用 JSON 把预定数据更新回 InterBase 数据库，而且速度也不错：





如果读者仔细比较前 2 小节使用 FireDAC 中央快储和使用 JSON 处理数据的方式，会发现 FireDAC 中央快储的执行速度比使用 JSON 更快，不过使用 JSON 的好处是可以其他开发工具或是程序语言写的客户端也可以呼叫范例 DataSnap 服务器。

9-3 使用 TFDJSONDataSets 功能

在 Delphi XE5 Update 2 中加入了一个非常重要的 FireDAC 和 DataSnap 功能，那就是新的 TFDJSONDataSets 相关类别，TFDJSONDataSets 相关类别随后不断的强化功能并在 10.3 的版本加入了压缩数据的功能让执行速度更快速。

那 TFDJSONDataSets 相关类别到底是做什么用的呢？简单的说 TFDJSONDataSets 相关类别可以结合 DataSnap 开发使用 JSON 和 RESTful 的系统架构，让程序员可以使用比较简单的方式开发 CRUD 的 App。在上一小节说明使用 JSON 技术开发 DataSnap 系统时，程序员都需要执行下面的工作：

- 客户端需自行解析内容
- 伺服端和客户端都需自行处理 CRUD
- 直接回传 TDataSet 回客户端的话，客户端需解析 TDataSet 的 JSON 内容

例如在上一小节中客户端就需要把旅馆预定数据封装成 JSON 对象再传递给 DataSnap 服务器，而服务器又需要再解析 JSON 对象。

但如果使用 TFDJSONDataSets 相关类别，那这些类别可提供如下的功能：

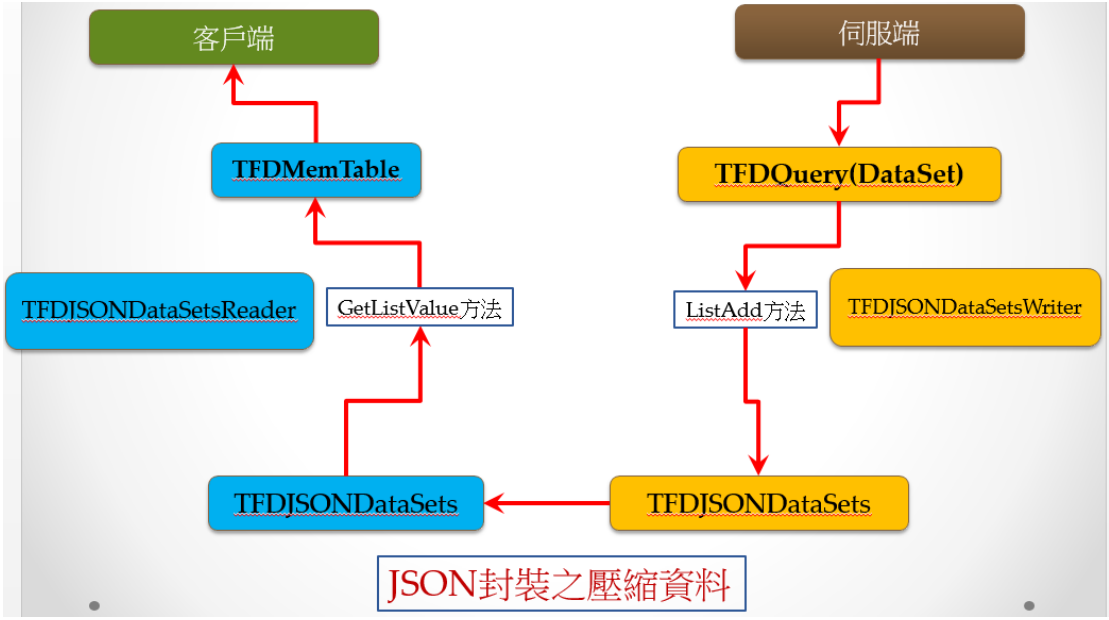
- 可在单一呼叫中处理多个数据集(DataSet)
- 可和 TFDMemTable 共同使用
- 可直接在单一呼叫中回传客户端的多个数据异动(Delta)
- 非常容易在伺服端处理 CRUD 的数据
- 可使用客制化格式读取或是异动数据

从上面的说明可以了解，由于 TFDJSONDataSets 相关类别可在一次网络的回来呼叫中处理多个数据集，因此它们可以减少网络的回来次数而增加执行效率，另外 TFDJSONDataSets 相关类别又可压缩数据也可增加执行效率，最后 TFDJSONDataSets 相关类别即可提供数据集对象和 JSON 对象之间的转换和解析，因此也可以免除程序员需要自行处理 JSON 内容的工作，进而加快程序员的开发速度。

TFDJSONDataSets 相关类别是使用 Reader/Writer 设计样例，程序员使用 Writer 把数据集对象写入 TFDJSONDataSets 中，藉由 TFDJSONInterceptor 转成 JSON 再传递出去。在接收方再使用 TFDJSONInterceptor 把 JSON 转回 TFDJSONDataSets，最后再使用 Reader 把数据集对象读出。一般来说程序员只需要使用下表的 4 个类别即可：

类别	说明
TFDJSONDataSets	FireDAC 使用的 JSON DataSet, 其中可包含多个 DataSet 对象
TFDJSONDataSetsWriter	使用此类别把 TDataSet 写入 TFDJSONDataSets
TFDJSONDataSetsReader	使用此类别把 TDataSet 从 TFDJSONDataSets 中读回
TFDJSONInterceptor	使用此类别把 TFDJSONDataSets 和 JSON 格式之间做转换

使用 TFDJSONDataSets 相关类别的方式如同下图所示：



在伺服端使用 TFDJSONDataSetsWriter 类别的类别方法 ListAdd()把 TFDQuery 对象写入 TFDJSONDataSets 对象，再传递到另一端，而客户端则可使用 TFDJSONDataSetsReader 类别的类别方法 GetListValue()或是 GetListValueByName()把数据集对象读出再加载 TFDMemTable 对象中即可存取其中的数据。

TFDJSONDataSetsWriter 类别的类别方法 ListAdd()定义如下：

```
class procedure ListAdd(const ADataList: TFDJSONDataSets; const AName: string;
```

```
const ADataSet: TFDAdaptedDataSet); overload;
   class procedure ListAdd(const ADataList: TFDJSONDataSets; const ADataSet:
TFDAdaptedDataSet); overload;
```

它接受 1 个 TFDJSONDataSets 对象，一个字符串名称参数和一个 TFDAdaptedDataSet 对象。这代表在 TFDJSONDataSets 中使用下列的方式储存数据集对象：

```
Name : DataSet
```

例如如果我们使用 TFDJSONDataSets 对象来处理前面旅馆预定数据的应用，那么在 TFDJSONDataSets 对象中我们可储存如下的 2 对数据：

```
'预定信息': TblreservationsTable 和'客户' : TblhotelcustomersTable
```

因此可使用如下的程序代码：

```
TFDJSONDataSetsWriter. ListAdd (aFDJSONDataSets ,'预定信息': TblreservationsTable);
TFDJSONDataSetsWriter.ListAdd (aFDJSONDataSets ,'客户' : TblhotelcustomersTable);
```

而如果我们是想查询台北市旅馆数据，那那么在 TFDJSONDataSets 对象中我们可储存如下的数据：


```
'台北市旅馆': fdqTaipeiHotels
```

由于台北市旅馆数据只有一个数据集对象，因此可使用上面第 2 个 ListAdd() 类别方法：

因此可使用如下的程序代码：

```
TFDJSONDataSetsWriter. ListAdd (aFDJSONDataSets , fdqTaipeiHotels);
```

而当 TFDJSONDataSets 对象传递到客户端后，客户端就可以使用 TFDJSONDataSetsReader 类别的类别方法 GetListValue() / GetListValueByName()读出 TFDAdaptedDataSet：

```
   class function GetListValue(const ADataList: TFDJSONDataSets; I: Integer):
TFDAdaptedDataSet; static;
   class function GetListValueByName(const ADataList: TFDJSONDataSets;const AName:
string): TFDAdaptedDataSet; static;
```

因此使用如下的程序代码即可读回上面的北市旅馆数据：

```
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,0);
```

使用如下的程序代码即可读回上面的旅馆预定数据和客户数据：

```
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,0);
TFDJSONDataSetsReader.GetListValue (aFDJSONDataSets ,1);
```

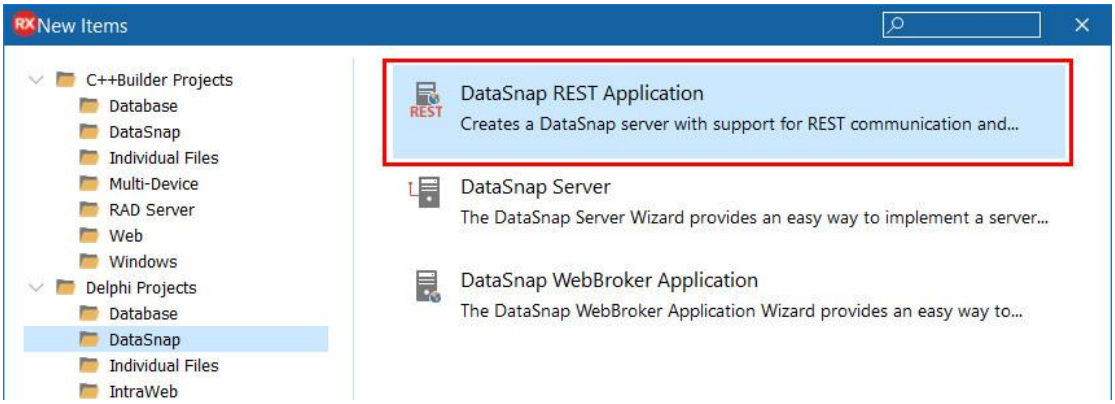
或是：

```
TFDJSONDataSetsReader. GetListValueByName (aFDJSONDataSets , '预定信息');
TFDJSONDataSetsReader. GetListValueByName (aFDJSONDataSets , '客户');
```

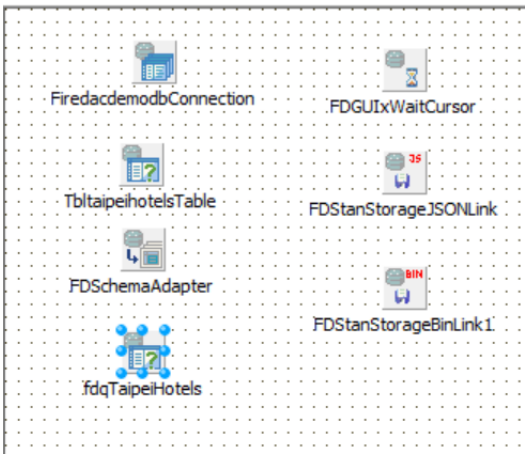
了解了如何使用 TFDJSONDataSets 相关类别后就可以让我们使用开发查询台北市旅馆信息的 RESTful 系统了。

9-3-1 开发 RESTful DataSnap 服务器

首先建一个 DataSnap REST Application 项目：



再于 ServerMethodsUnit1 中加入如前面范例 DataSnap 服务器一样的组件：



接着在 ServerMethodsUnit1 程序单元的 public 部分同样宣服务方法 GetTaipeiHotels()：

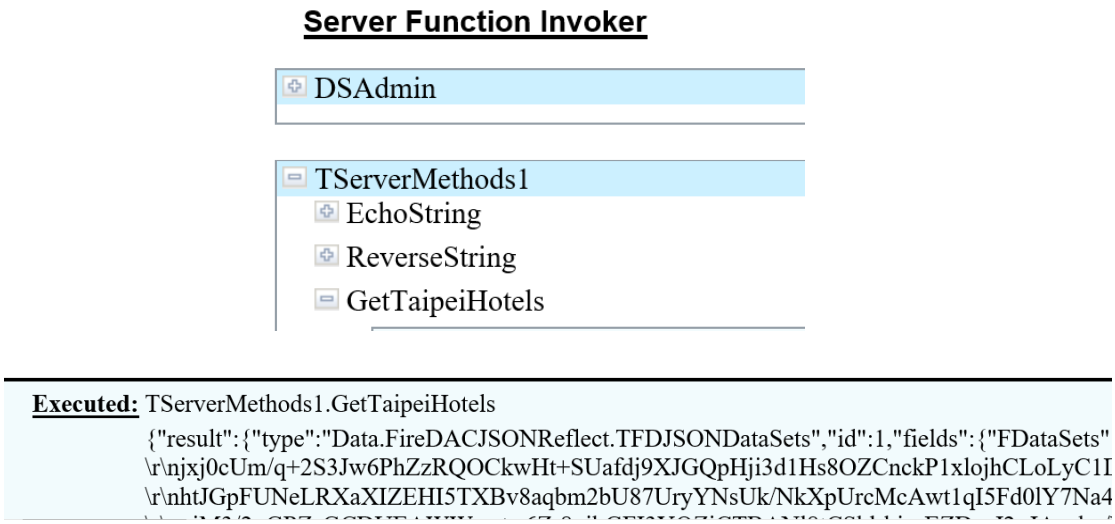
```
Public
  { Public declarations }
…
  function GetTaipeiHotels : TFDJSONDataSets;
```

再实作如下：

```
001    function TServerMethods1.GetTaipeiHotels: TFDJSONDataSets;
002    begin
003      fdqTaipeiHotels.Active := False;
004      Result := TFDJSONDataSets.Create;
005      TFDJSONDataSetsWriter.ListAdd(Result, fdqTaipeiHotels);
006    end;
```

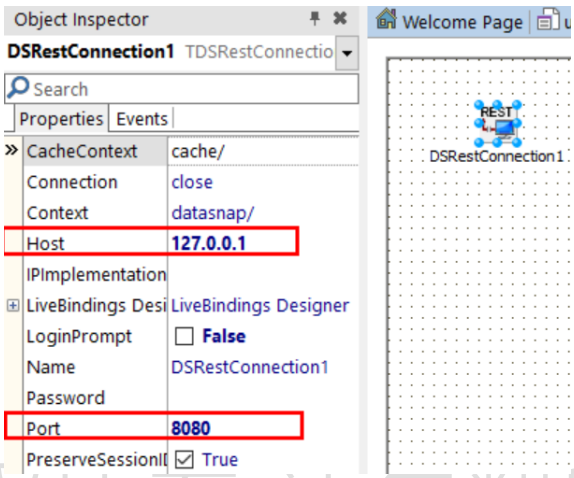
GetTaipeiHotels()方法于 003 行先把 fdqTaipeiHotels 组件关闭，004 行建立回传的 TFDJSONDataSets 对象，最后于 005 行呼叫 TFDJSONDataSetsWriter 类别的类别方法 ListAdd()把 fdqTaipeiHotels 写入要回传的 TFDJSONDataSets 对象中即可。

现在执行此范例服务器，开启浏览器就可以如下呼叫 GetTaipeiHotels()方法，这当然是因为此范例服务器是 RESTful 服务器且回传 JSON 数据，从下图下半部分可看到 GetTaipeiHotels()方法以 JSON 对象封装回传 fdqTaipeiHotels 组件中的数据集对象：

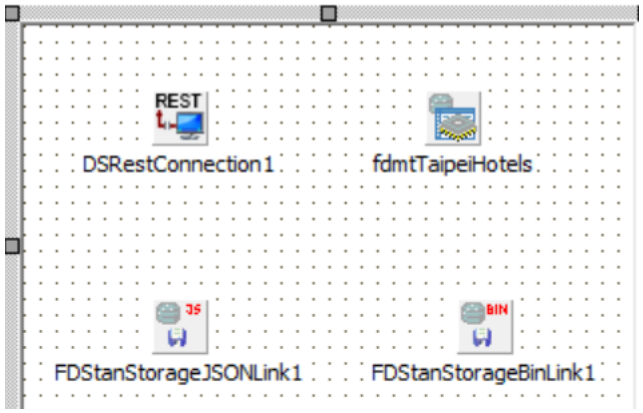


9-3-2 开发 RESTful 客户端

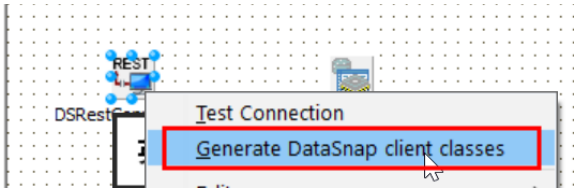
现在于刚才的范例项目中再建立一个新的 Multi-Device Applicatio 项目，于其中再建立一个数据模块并在其中加入一个 TDSRestConnection 组件，设定它的 Host 和 Port 特性值为刚才的范例 RESTful DataSnap 服务器的位置使用的 IP 地址和通讯端口。由于笔者是在笔者的机器中执行范例 RESTful DataSnap 服务器，因此设定 Host 为 127.0.0.1，而 8080 则是 RESTful DataSnap 服务器使用的预定通讯端口：



接着再放入 TFDMemTable 和 TFDStanStorageBinLink 以及 TFDStanStorageJSONLink 组件：



再右击 TDSRestConnection 组件选择 Generate DataSnap client classes 选项以自动产生客户端链接范例 RESTful DataSnap 服务器的程序代码(范例 RESTful DataSnap 服务器必须先执行)：



在上面步骤自动产生的客户端程序代码中就可以看到范例 RESTful DataSnap 服务器提的服务方法 GetTaipeiHotels()：

```
function GetTaipeiHotels(const ARequestFilter: string = ''): TFDJSONDataSets;
```

设计此范例 RESTful 客户端的主窗体和上一个范例客户一样，并在”台北旅馆”按钮中先呼叫 GetTaipeiHotels()方法链接范例 RESTful DataSnap 服务器，呼叫服务方法最得回传的 TFDJSONDataSets 对象 LDataSetList，再呼叫 DisplayTaipeiHotels()方法显示回传的数据：

```
001   procedure TfmMainForm.Button1Click(Sender: TObject);
002   var
003     aServer : TServerMethods1Client;
004     LDataSetList: TFDJSONDataSets;
005   begin
006     aServer := Nil;
007     try
008       LDataSetList := GetTaipeiHotels(aServer);
009       DisplayTaipeiHotels(LDataSetList);
010     finally
011       aServer.Free;
012     end;
013   end;
```


客户端的 GetTaipeiHotels()方法先于 004 行建立远程服务器对象再于 005 行呼叫它的 GetTaipeiHotels()方法并取得回传结果：

```
001   function TfmMainForm.GetTaipeiHotels(var aServer : TServerMethods1Client) :
TFDJSONDataSets;
002   begin
003     Result := Nil;
004     aServer :=
TServerMethods1Client.Create(dmRESTfulClient.DSRestConnection1);
005     Result := aServer.GetTaipeiHotels();
006   end;
```

DisplayTaipeiHotels()方法显示了如何从 TFDJSONDataSets 对象取出结果数据集。006 行先关闭数据模块中的 fdmtTaipeiHotels 组件 007 行确定回传的 TFDJSONDataSets 对象中的确包含了一个 TFDAdaptedDataSet 对象(即伺服端回传的 fdqTaipeiHotels 组件中的数据集对象)，008 行呼叫 TFDJSONDataSetsReader 类别的类别方法 GetListValue()方法取出数据集对象，再于 009 行呼叫 fdmtTaipeiHotels 组件的 AppendData()方法把此数据集对象加入到 fdmtTaipeiHotels 组件中，最后即可藉由 fdmtTaipeiHotels 组件一一取出其中的回传台北市旅馆数据并显示出来：

```
001   procedure TfmMainForm.DisplayTaipeiHotels(LDataSetList: TFDJSONDataSets);
002   var
003     lvi : TListViewItem;
004     aDataSet : TFDAdaptedDataSet;
005   begin
006     dmRESTfulClient.fdmtTaipeiHotels.Active  := False;
007     Assert(TFDJSONDataSetsReader.GetListCount(LDataSetList) = 1);
008     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 0);
009     dmRESTfulClient.fdmtTaipeiHotels.AppendData(aDataSet);
010
011     dmRESTfulClient.fdmtTaipeiHotels.First;
012     while (not dmRESTfulClient.fdmtTaipeiHotels.Eof) do
013     begin
014       lvi := lvTaipeiHotels.Items.Add;
015       lvi.Text :=
dmRESTfulClient.fdmtTaipeiHotels.FieldByName('STITLE').AsString;
016       lvi.Detail :=
dmRESTfulClient.fdmtTaipeiHotels.FieldByName('MEMO_TEL').AsString;
017       dmRESTfulClient.fdmtTaipeiHotels.Next;
018     end;
019   end;
```


现在执行此范例客户端即可看到类似如下的执行结果，和前面范例不同的是这是一个 RESTful 架构，而数据都是使用 JSON 格式传递的。

9-3-3 开发 RESTful 多数据表查询

了解了上面讨论的内容，那 RESTful 多数据表查询就很容易实作了，我们只要把多个数据集组件写入回传到客户端的 TFDJSONDataSets 对象即可。为了简单的说明起见，让我们实作一个简单的功能，允许客户端查询特定旅馆的预定信息。

修改范例 DataSnap 服务器

先在范例 RESTful DataSnap 服务器提供 GetHotelReservation()方法，它接受旅馆 ID 然后从 TblreservationsTable 和 TblhotelcustomersTable 这 2 个数据表中取出数据，再于下列的 010 和 011 行把这 2 个数据表写入 TFDJSONDataSets 对象中并回传到客户端：

```
001    function TServerMethods1.GetHotelReservation(iHID: String): TFDJSONDataSets;
002    begin
003      TblreservationsTable.Active := False;
004      TblreservationsTable.Params[0].Value := iHID;
005      TblreservationsTable.Open();
006      TblhotelcustomersTable.Active := False;
007      TblhotelcustomersTable.Params[0].Value :=
TblreservationsTable.FieldByName('RID').Value;
```

```
008     TblreservationsTable.Close();
009     Result := TFDJSONDataSets.Create;
010     TFDJSONDataSetsWriter.ListAdd(Result, TblreservationsTable);
011     TFDJSONDataSetsWriter.ListAdd(Result, TblhotelcustomersTable);
012   end;
```

修改范例客户端

在客户端只需先在 009 行确定伺服端回传了 2 个数据表对象，再分别取出每一个数据表对象并加载客户端的 TFDMemeTable 对象中即可：

```
001   procedure TfmMainForm.DisplayHotelReservations(LDataSetList:
TFDJSONDataSets);
002   var
003     lvi : TListViewItem;
004     aDataSet : TFDAdaptedDataSet;
005   begin
006     lStart := Now;
007     dmRESTfulClient.mtRHotel.Active  := False;
008     dmRESTfulClient.mtRCustomers.Active := False;
009     Assert(TFDJSONDataSetsReader.GetListCount(LDataSetList) = 2);
010     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 0);
011     dmRESTfulClient.mtRHotel.AppendData(aDataSet);
012     aDataSet := TFDJSONDataSetsReader.GetListValue(LDataSetList, 1);
013     dmRESTfulClient.mtRCustomers.AppendData(aDataSet);
014
015     dmRESTfulClient.mtRCustomers.First;
016     while (not dmRESTfulClient.mtRCustomers.Eof) do
017     begin
018       lvi := lvRCustomers.Items.Add;
019       lvi.Text := dmRESTfulClient.mtRCustomers.FieldByName('NAME').AsString;
020       lvi.Detail :=
dmRESTfulClient.mtRCustomers.FieldByName('PHONE').AsString;
021       dmRESTfulClient.mtRCustomers.Next;
022     end;
023     lEnd := Now;
024   end;
```

9-3-4 开发 CRUD 功能伺服端

TFDJSONDataSets 相关类别也提供了非常易于使用但功能强大的 RESTful CRUD 的功能，提供 RESTful CRUD 功能的相关类别是 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates。下面的表格说明了这些类别提供的功能：

类别	说明
TFDJSONDeltas	包含异动数据(Delta)的类别，程序员把 Delta 写入 TFDJSONDeltas 对象再把它传递给另一方。
TFDJSONDeltasWriter	使用此类别把 Delta 写入 TFDJSONDeltas 对象中
TFDJSONDeltasApplyUpdates	使用此类别把 Delta 真正异动回数据库中

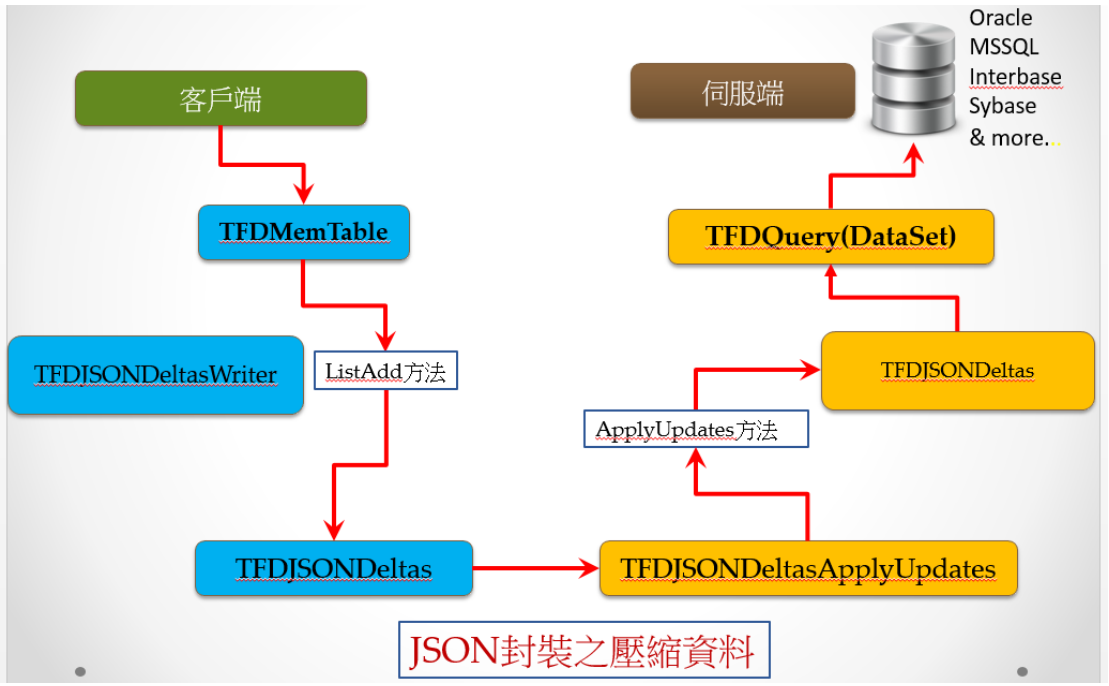
请注意，在使用 TFDJSONDeltasWriter 把 Delta 写入 TFDJSONDeltas 对象时也是使用”名称”和 Delta 的格式：

```
Name : Delta
```

因此 TFDJSONDeltasWriter 也可以把多个 Delta 写入 TFDJSONDeltas 对象中。

使用 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates 这 3 个类别的方式如下图所示。当客户端要把异动的数据更新回数据库时，就把 TFDMemTable 中的 Delta 藉由 TFDJSONDeltasWriter 类别的类别方法 ListAdd()写入 TFDJSONDeltas 对象，再把 TFDJSONDeltas 对象传递给 DataSnap 服务器。

到了 DataSnap 服务器后就呼叫 TFDJSONDeltasApplyUpdates 对象的 ApplyUpdates()方法把 TFDJSONDeltas 对象中的 Delta 藉由 TFDQuery 组件更新回数据库之中。



了解了上述的工作原理之后我们就可以回到范例 DataSnap 服务器，在它的 ServerMethodsUnit 中加入一个 PostTheHotel()方法，PostTheHotel()接受从客户端传来的 TFDJSONDeltas 对象参数，005 行先建立 TFDJSONDeltasApplyUpdates 对象并传入 TFDJSONDeltas 对象做为建构元参数，006 行呼叫它的 ApplyUpdates()方法同时传入 2 个参数，第 1 个参数是 TFDJSONDeltas 物件中名为 sTheHotel 的 Delta，第 2 个参数则是数据模块中 fdqTaipeiHotels 对象的 Command 特性值：

```
001   function TServerMethods1.PostTheHotel(const ADeltaList: TFDJSONDeltas):
Integer;
002   var
003     LApply: IFDJSONDeltasApplyUpdates;
004   begin
005     LApply := TFDJSONDeltasApplyUpdates.Create(ADeltaList);
006     Result:= LApply.ApplyUpdates(sTheHotel, fdqTaipeiHotels.Command);
007     if LApply.Errors.Count > 0 then
008       raise Exception.Create(LApply.Errors.Strings.Text);
009   end;
```

上面 ApplyUpdates()方法的第 1 个参数 sTheHotel 是伺服端和客户端共同使用的代表 TFDJSONDeltas 对象中的 Delta 名称字符串：

```
const
  sTheHotel = '旅馆资料';
```

另外需要注意的是在上面的程序代码中 LApply 事实上是 IFDJSONDeltasApplyUpdates 接口的变量，这是因为 TFDJSONDeltasApplyUpdates 类别实作了 IFDJSONDeltasApplyUpdates 接口：

```
TFDJSONDeltasApplyUpdates = class(TFDJSONDeltasReader, IFDJSONDeltasApplyUpdates)
```

而 ApplyUpdates()方法是宣告在 IFDJSONDeltasApplyUpdates 接口中：

```
  IFDJSONDeltasApplyUpdates = interface(IFDJSONDeltasReader)
    ['{58213D3C-BFE8-4BE3-8197-4236FFC215C8}']
    function ApplyUpdates(const AKey: string; const ASelectCommand:
TFDCustomCommand): Integer; overload;
    function ApplyUpdates(const Index: Integer; const ASelectCommand:
TFDCustomCommand): Integer; overload;
    function ApplyUpdates(const AKey: string; const AAdapter: TFDTableAdapter):
Integer; overload;
    function ApplyUpdates(const Index: Integer; const AAdapter: TFDTableAdapter):
Integer; overload;
    function GetErrors: TFDJSONErrors;
    property Errors: TFDJSONErrors read GetErrors;
  end;
```


现在再次编译和执行此范例 RESTful DataSnap 服务器。

9-3-4 开发 CRUD 功能客户端

回到范例客户端，在下面更新旅馆数据的确定按钮中呼叫 PostTheHotel()方法把使用者在手机中修改的旅馆数据更新回数据库：


```
procedure TfmMainForm.Button4Click(Sender: TObject);
begin
  PostTheHotel;
end;
```


PostTheHotel()方法同样在 015~020 行中把用户输入的数据写入 mtTheHotel 这个 TFDMemTable 组件中，022 行是关键，它在 GetDeltas()子方法中的 008 行先建立 TFDJSONDeltas 对象，再于 009 行呼叫类别方法 ListAdd()把 mtTheHotel 的 Delta 写入 TFDJSONDeltas 对象并以 sTheHotel 命名此 Delta，当然在客户端 sTheHotel 也是定义成相同的名称：

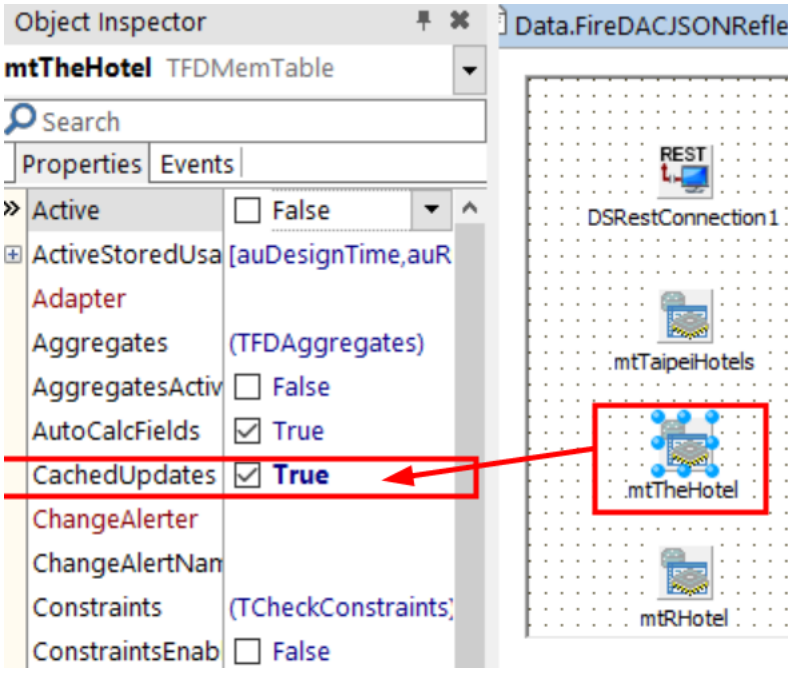
```
const
  sTheHotel = '旅馆资料';
```

最后在 025 行呼叫 DataSnap 服务器的 PostTheHotel()方法把 TFDJSONDeltas 对象传递给 DataSnap 服务器以完成更新旅馆数据的工作：

```
001    procedure TfmMainForm.PostTheHotel;
002    var
003      LDeltaList: TFDJSONDeltas;
004      aServer : TServerMethods1Client;
005
006      function GetDeltas: TFDJSONDeltas;
007      begin
008        Result := TFDJSONDeltas.Create;
```

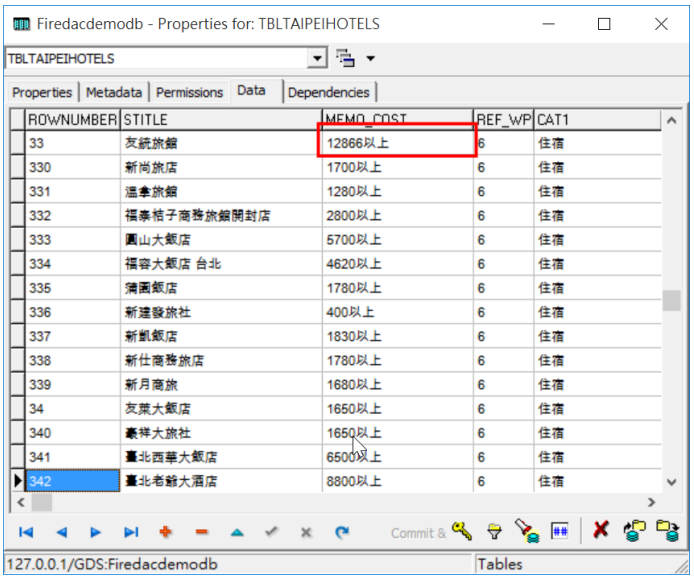
```
009       TFDJSONDeltasWriter.ListAdd(Result, sTheHotel,
dmRESTfulClient.mtTheHotel);
010     end;
011
012   begin
013     lStart := Now;
014
015     dmRESTfulClient.mtTheHotel.Edit;
016     dmRESTfulClient.mtTheHotel.FieldByName('ROWNUMBER').Value :=
dmRESTfulClient.mtTheHotel.FieldByName('ROWNUMBER').Value;
017     dmRESTfulClient.mtTheHotel.FieldByName('STITLE').Value := edtTitle.Text;
018     dmRESTfulClient.mtTheHotel.FieldByName('MEMO_TEL').Value :=
edtPhone.Text;
019     dmRESTfulClient.mtTheHotel.FieldByName('MEMO_COST').AsString :=
edtPrice.Text;
020     dmRESTfulClient.mtTheHotel.Post;
021
022     LDeltaList := GetDeltas;
023     aServer :=
TServerMethods1Client.Create(dmRESTfulClient.DSRestConnection1);
024     try
025       aServer.PostTheHotel(LDeltaList);
026     finally
027       aServer.Free;
028     end;
029     lEnd := Now;
030   end;
```


接着一个很重要的步骤就是必须开启 mtTheHotel 组件使用 CachedUpdates 功能，否则数据无法成功更新回数据库之中(因为没有 Delta)：



编译并执行范例手机 App，试着更新一个旅馆的数据如下所示我们把价位改成 12866 然后点选确定按钮更新数据：

之后到后端的 InterBase 数据库查看此笔旅馆的数据，可以看到下图在手机客户端异动的数据果然成功藉由 RESTful DataSnap 服务器更新回数据库中了：



了解了如何使用 TFDJSONDeltas，TFDJSONDeltasWriter 和 TFDJSONDeltasApplyUpdates 这 3 个类别进行对一个数据表的 RESTful 的 CRUD 工作后，就请读者再挑战一下修改此 RESTful 范例 DataSnap 服务器和范例客户端 App 来进行对多个数据表的 RESTful 的 CRUD 工作吧，Have Fun！

第10章 开发安全，高效率的DataSnap应用系统

DataSnap 的效率如何？可以支持多少客户端？如何能够开发安全，高效率的 DataSnap 应用系统呢？这应该是阅读本书的读者在学习了如何开发 DataSnap 应用系统之后最想询问的问题。

然而要回答这些问题并不是简单的事情，因为这牵涉到许多不同的内容，例如您如何设计您的 DataSnap 应用系统？您如何设计企业类别和对象？你使用的硬件能力和架构?你撰写的程序代码的效率？以及最后本章要讨论的主题，您如何开发您的 DataSnap 服务器？

因此本章主要讨论的内容将暂时排除硬件，设计和程序代码撰写方式的因素，本章将为读者说明如何开发安全，高效率的 DataSnap 服务器，让它可以支持合理数量的客户端并提供良好的响应速度。

为了稍后内容的讨论以及比较不同 DataSnap 服务器，因此先让我们讨论如何开发 DLL 型态的 DataSnap 服务器，这包含了 ISAPI 或 Apache DataSnap 服务器，不过为了简化说明的内容就让我们先以如何开发 ISAPI 型态的 DataSnap 服务器。那为什么要使用 DLL 型态的 DataSnap 服务器呢？简单的说 DLL 型态的 DataSnap 服务器其执行效率比前面章节说明的 EXE 型态的 DataSnap 服务器来得好上许多，因此如果您是真的要在实际环境中使用 DataSnap 应用架构，那一定要使用 DLL 型态的 DataSnap 服务器，因为 EXE 型态的 DataSnap 服务器是使用来学习和易于除错之用。

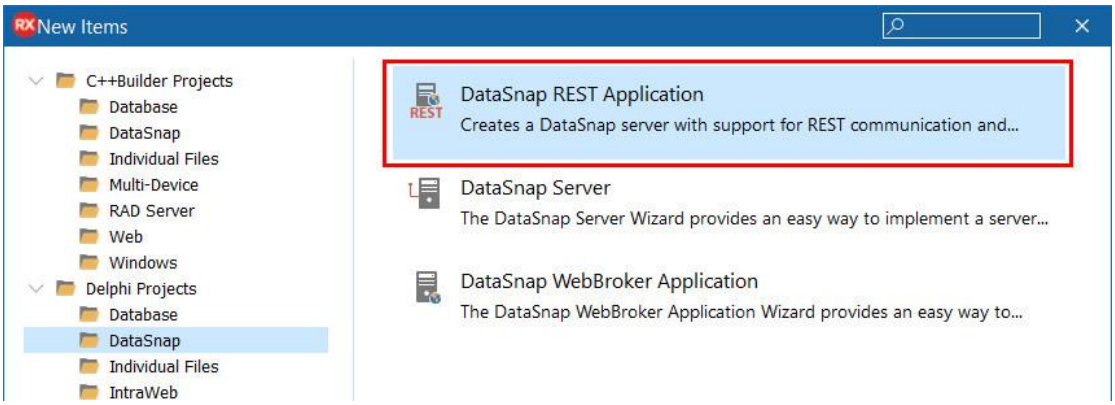
现在就让我们开始探究执行效率这个重要又有趣的旅程吧。

10-1 开发 DLL 型态的 DataSnap 服务器

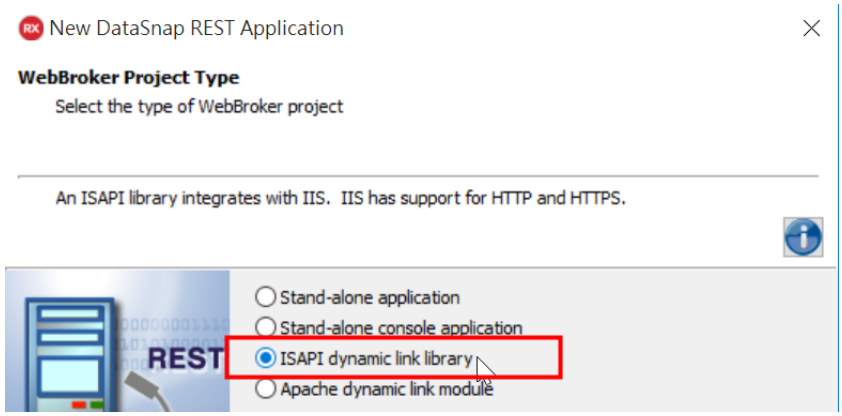
开发 DLL 型态的 DataSnap 服务器和前面讨论 EXE 型态的 DataSnap 服务器差不多，只是在选择建立的 DataSnap 型态时要注意选择 ISAPI 或是 Apache。

10-1-1 开发 ISAPI 型态的 DataSnap 服务器

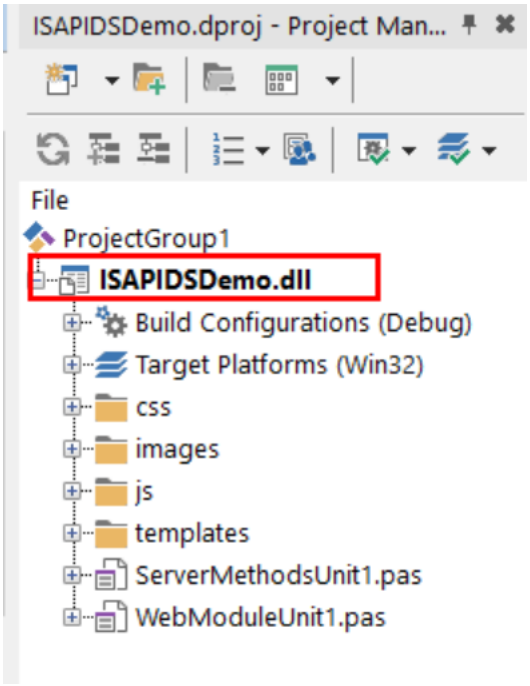
请在 IDE 中选择建立 DataSnap REST Application 项目：



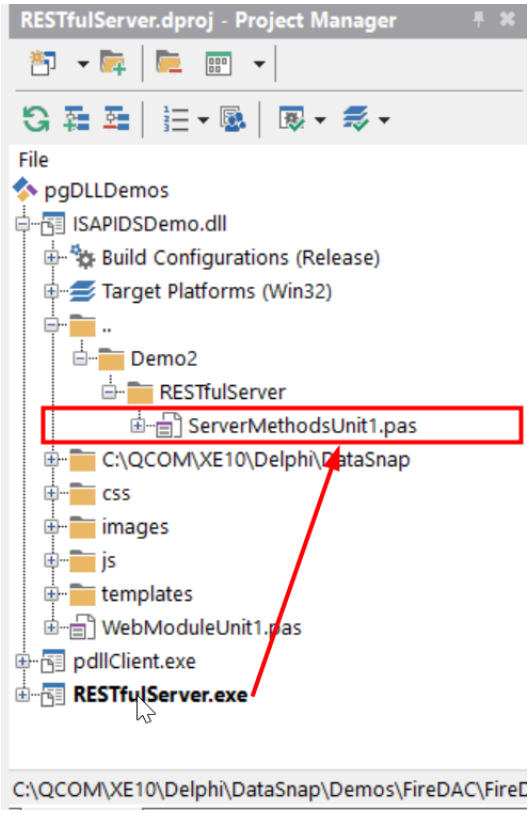
在按下来的选项中请选择建立如下的 ISAPI 或是 Apache 选项：



在执行完所有的建立过程后 Delphi 会建立一个 DLL 型态的项目,如下所示：



请读者注意的是不管是 DLL 或是 EXE 型态的 DataSnap 服务器都使用 ServerMethodUnit 程序单元，因此只要我们在 ServerMethodUnit 程序单元中小心撰写程序代码，那么 ServerMethodUnit 程序单元可以同时使用在 DLL 和 EXE 型态的 DataSnap 服务器中，如此一来当我们在 EXE 型态的 DataSnap 服务器除错和单元测试完成之后就可以很放心的使用在 DLL 型态的 DataSnap 服务器中了，例如在下图笔者的 IDE 中即将讨论的 ISAPI DataSnap 服务器中的 ServerMethodUnit 程序单元就是重复使用前面使用在 EXE 型态的 DataSnap 服务器中的 ServerMethodUnit 程序单元：



开发 ISAPI DataSnap 服务器和前面开发 EXE 型态的 DataSnap 服务器是一样的，都是在 ServerMethodUnit 程序单元实作服务方法，差别只是当开发 ISAPI DataSnap 服务器完成之后要部署到 IIS 之中，稍后会详细说明如何部署 ISAPI DataSnap 服务器，现在先让我们在此项目的 ServerMethodUnit 程序单元中加入一个服务方法 GetTaipeiHotelsByJSONFromJSONFile()。

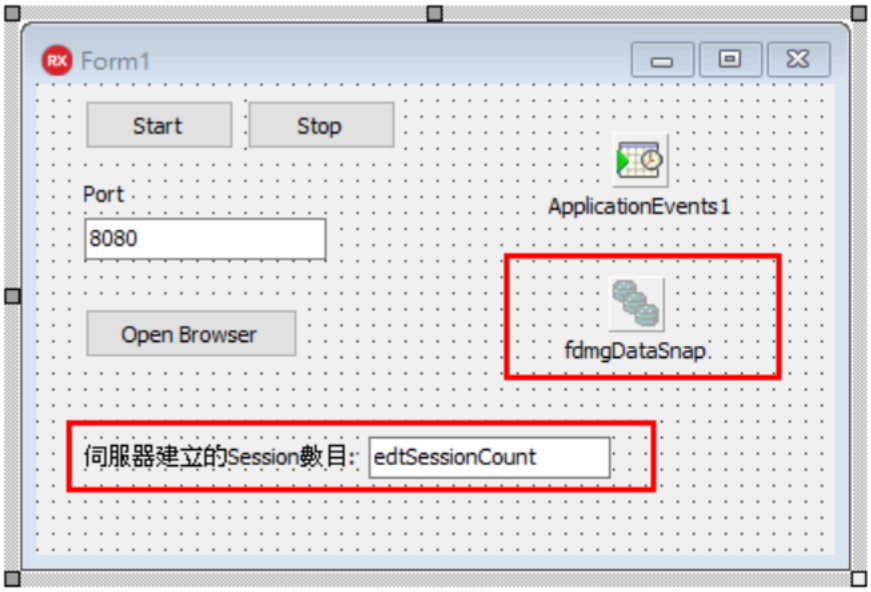
为了稍后测试执行效率，因此笔者把前面 TBLTAIPEIHOTELS 数据表中的数据以 JSON 格式都储存在 TaipeiHotels.JSON 档案中，因此 GetTaipeiHotelsByJSONFromJSONFile()方法先使用 TFDQuery2 组件把它加载成为数据集，再使用 TJSONArrayBuilder 对象把所有旅馆名称以 JSON 数组的格式回传：

```
const
  TaipeiHotelsJSONFilePath =
'c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\Data\TaipeiHotels.JSON';


function TServerMethods1.GetTaipeiHotelsByJSONFromJSONFile: String;
var
  sw : TStringWriter;
  jtw : TJsonTextWriter;
  jaBuilder : TJSONArrayBuilder;
```

```
  ele : TJSONCollectionBuilder.TElements;
begin
  Result := '';
  if (not FileExists(TaipeiHotelsJSONFilePath)) then
    Exit;

  sw := TStringWriter.Create;
  jtw := TJsonTextWriter.Create(sw);
  jaBuilder := TJSONArrayBuilder.Create(jtw);
  fdqGeneral.LoadFromFile(TaipeiHotelsJSONFilePath, sfJSON);
  fdqGeneral.Open();
  try
    ele := jaBuilder.BeginArray;
    while (not fdqGeneral.Eof) do
    begin
      ele := ele.Add(fdqGeneral.FieldByName('STITLE').AsString);
      fdqGeneral.Next;
    end;
    ele.EndAll;
    Result := sw.ToString;
  finally
    fdqGeneral.Close();
    jaBuilder.Free;
    jtw.Free;
    sw.Free;
  end;
end;
```


由于 ServerMethodUnit 程序单元同时使用于 2 种型态的 DataSnap 服务器中，因此也顺便让我们同时修改 EXE 型的 DataSnap 服务器加入 TFDManager 组件以便让 2 种型态的 DataSnap 服务器能够开启 FireDAC 多线程功能以便增加执行效率。

请在 IDE 中开启前面章节讨论的 RESTfulServer 项目，在它的主窗体中加入一个 TFDManager 组件：fdmgDataSnap，并且加入可显示客户端链接时 EXE 型的 DataSnap 服务器会建立的 Session 对象数目：



为了让 FireDAC 顺利执行在 IIS 中执行，我们需要正确的设定 FireDAC 使用的组态文件路径，因此让我们先在 EXE 型的 DataSnap 服务器的主窗体中定义路径常数：

```
const
  sTheHotel = '旅馆资料';
  DS_APPPATH = 'c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\';
  DEFFILE = 'FDConnectionDefs.ini';
  DRIVERFILE = 'FDDrivers.ini';
```


接着在主窗体的 OnCreate 事件呼叫 SetupFDManager()方法设定 TFDManager 组件：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  SetupFDManager;
end;
```

SetupFDManager()方法的工作就是设定要从什么路径加载 FireDAC 的 2 个 INI 组态档：

```
procedure TForm1.SetupFDManager;
begin
  fdmgDataSnap.Active := False;
  fdmgDataSnap.ConnectionDefFileName := DS_APPPATH + PathDelim + DEFFILE;
```

```
  fdmgDataSnap.DriverDefFileName := DS_APPPATH + PathDelim + DRIVERFILE;
  fdmgDataSnap.Active := True;
end;
```

再切换到刚才建立的 DLL DataSnap 服务器的主程序，同样修改 DLL 主程序如下：

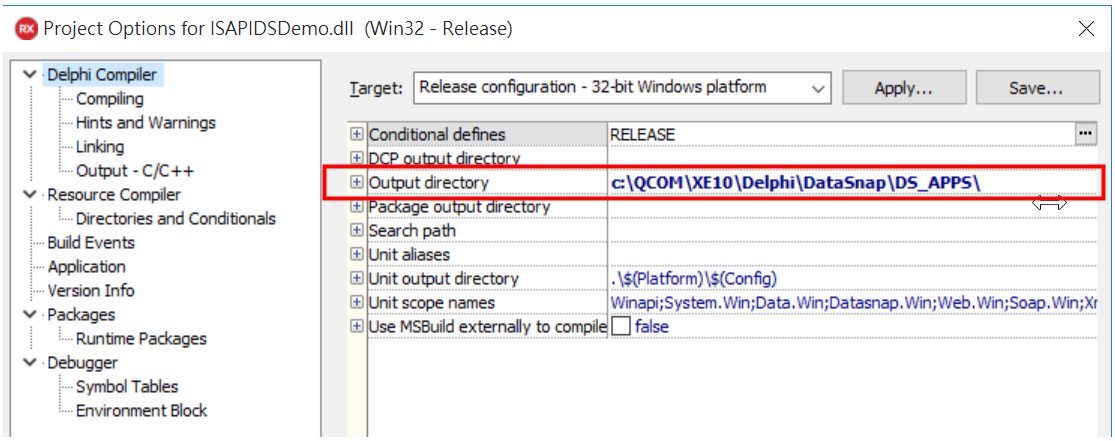
```
var
  fdmgDataSnap : TFDManager;

procedure SetupFDManager;
begin
  if (fdmgDataSnap = Nil) then
    fdmgDataSnap := TFDManager.Create(Nil);
  fdmgDataSnap.Active := False;
  fdmgDataSnap.ConnectionDefFileName := DS_APPPATH + PathDelim + DEFFILE;
  fdmgDataSnap.DriverDefFileName := DS_APPPATH + PathDelim + DRIVERFILE;
  fdmgDataSnap.Active := True;
end;
```



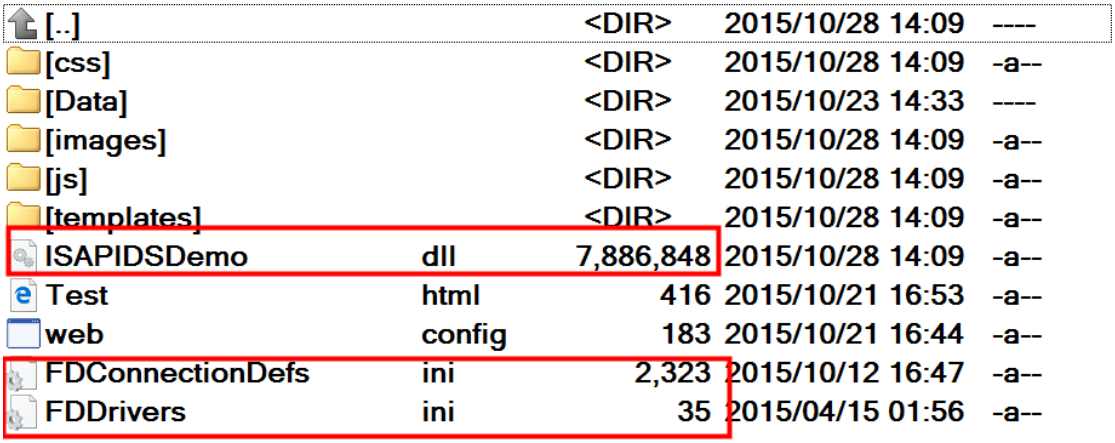
```
procedure TerminateThreads;
begin
  if (fdmgDataSnap <> Nil) then
  begin
    fdmgDataSnap.Close;
    FreeAndNil(fdmgDataSnap);
  end;
  TDSSessionManager.Instance.Free;
  Data.DBXCommon.TDBXScheduler.Instance.Free;
end;

begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.WebModuleClass := WebModuleClass;
  SetupFDManager;
  TISAPIApplication(Application).OnTerminate := TerminateThreads;
  Application.Run;
```

```
end.
```

由于稍后我们会建立 IIS 的站台，因此我们需要把编译的 DLL 部署到我们的 IIS 站台路径之中，请在 DLL 项目的 Options 对话盒的 Delphi Compiler | Output Directory 中设定稍后 IIS 站台使用的路径：



现在请同时编译 DLL 和 EXE 型态的 DataSnap 服务器，并把 FireDAC 使用的 2 个组态档都部署到笔者使用的”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”路径中，下图显示了编译并部署的 ISAPIDSDemo.dll 及 FireDAC 使用的 2 个组态档：

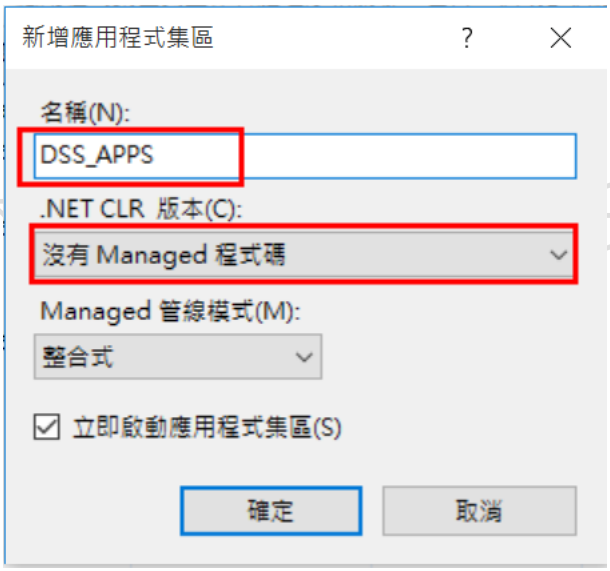


接下来的重要工作就是部署此 ISAPI DataSnap 服务器到 IIS 之中以便让稍后的客户端呼叫。

10-1-2 部署 ISAPI DataSnap 服务器

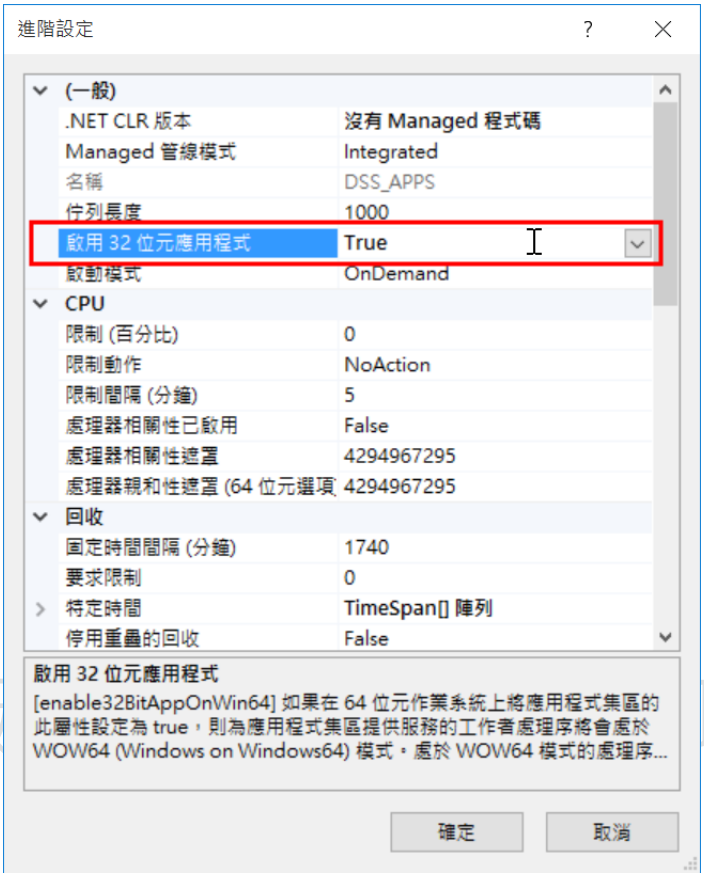
首先请启动 IIS 管理员(如果您找不到 IIS 管理员，那应该是您尚未安装 IIS，请到控制台 | 开启或关闭 Windows 功能中安装 IIS)，如下图所示在”应用程序集区” 右击鼠标再选择”新增应用程序集区...”选项：

在＂新增应用程序集区＂对话盒中为您的集区取一个名称再选择使用”没有 Managed 程序代码”选项(这是当然的，因为 Delphi 编译出来的 DLL 是原生的 DLL)：

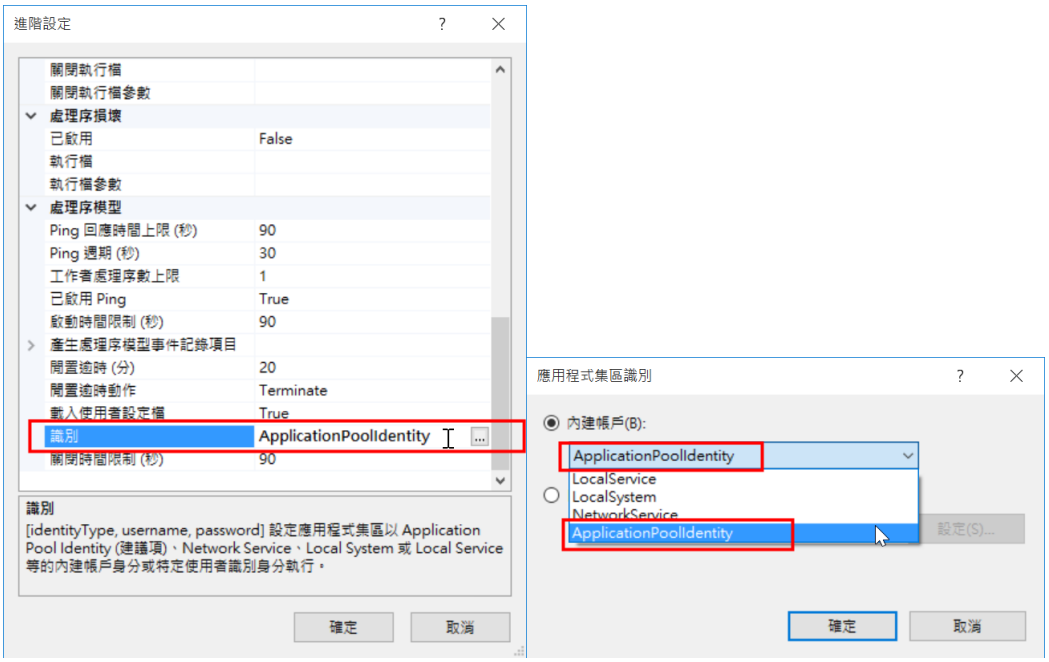


回选确定之后应该就可以看到我们建立了 DSS_APPS 集区，接下来请如下图在右方点选”进阶设定”：

由于我们编译的 ISAPIDSDemo.dll 是 32 位的 DLL 应用程序而笔者使用的 OS 是 64 位，因此需要在”进阶设定”中设定如下”启用 32 位应用程序”选项，当然如果读者是把 ISAPIDSDemo.dll 编译成 64 位就不需要进行这个设定：



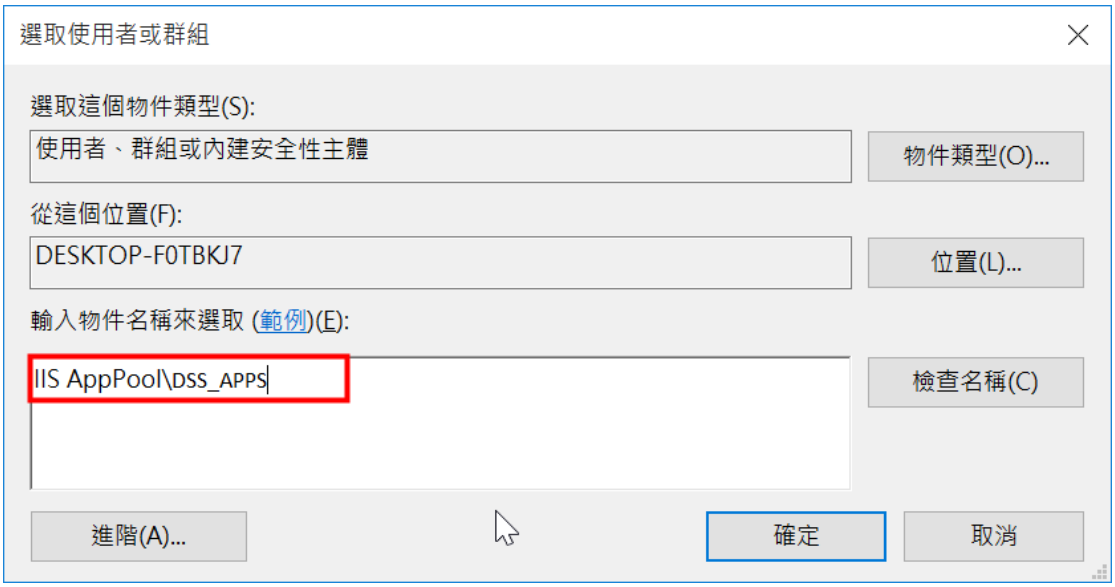
接着您需要设定 IIS 使用什么身份识别来执行此 DLL 应用程序，在 IIS 7.5 之后新增了”ApplicationPoolIdentity”这个虚拟账户，在下图中可以看到”进阶设定”中就内定使用 ApplicationPoolIdentity 账号权限来执行此应用程序集区，因此如果您的 ISAPI DataSnap 服务器会存取到一些系统资源，例如档案，数据库等那么您必须确定 ApplicationPoolIdentity 有适当的权限，例如我们的范例 ISAPI DataSnap 服务器会存取到”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目录下的 FireDAC 组态档，那么就必须设定 ApplicationPoolIdentity 拥有可以读取此目录的权限。



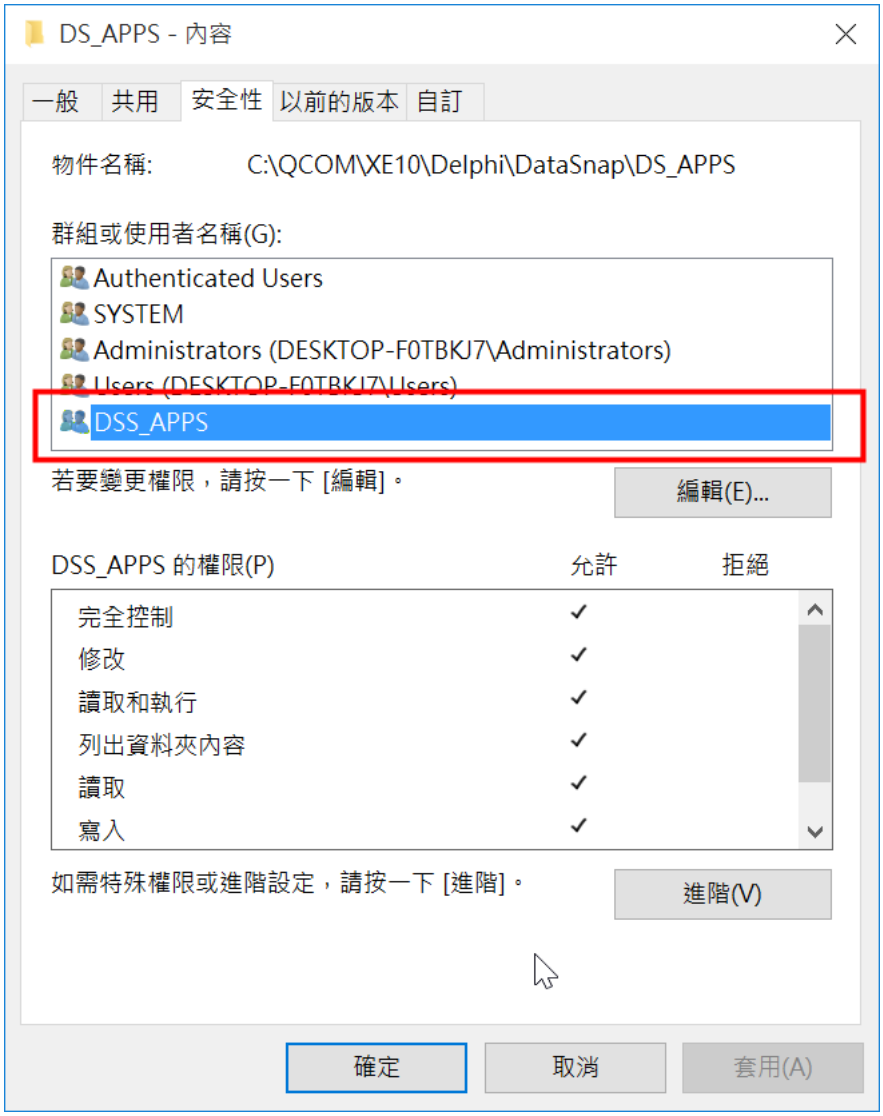
要指定 ApplicationPoolIdentity 可拥有存取”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目录的权限，我们可以使用文件管理器在此目录的安全性设定中使用如下的格式来加入刚才建立的 DSS_APPS 集区有访问权限：


```
“IIS AppPool\应用程序集区名称”
```

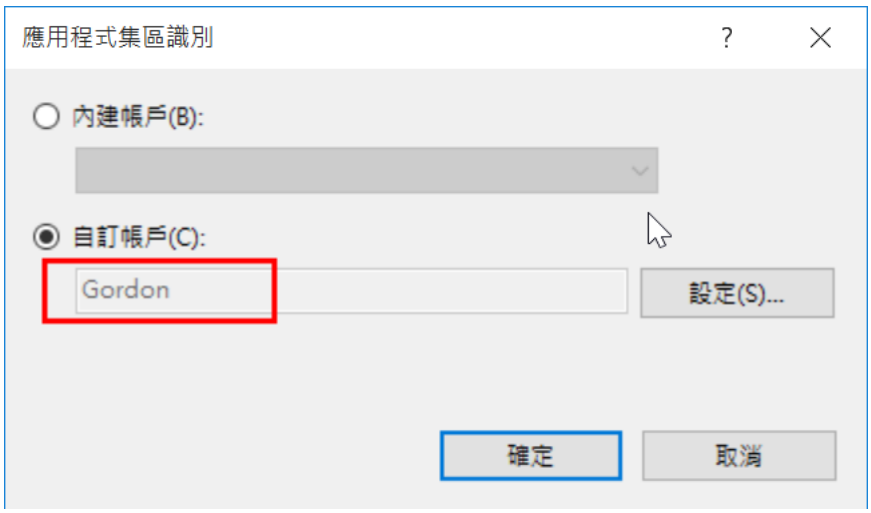
因此我们可以在”c:\QCOM\XE10\Delphi\DataSnap\DS_APPS\”目录中加入 IIS AppPool\DSS_APPS 集区：



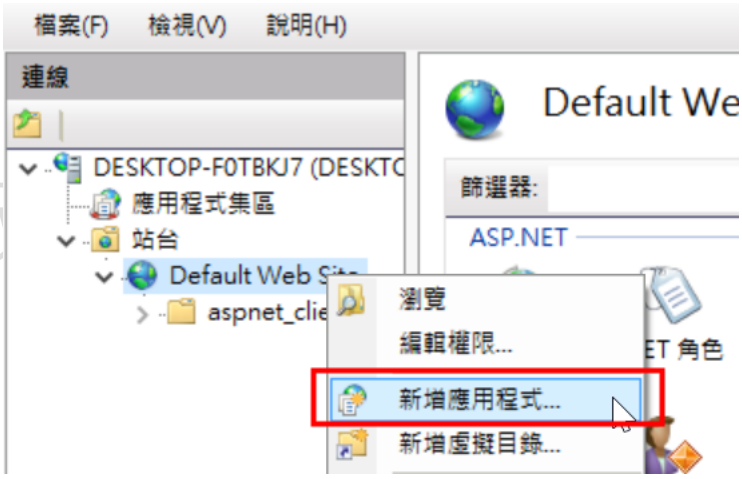
最后在此目录可存取的群组或使用者中就可以看到 DSS_APPS 集区，接着读者可以设定 DSS_APPS 集区的访问权限：



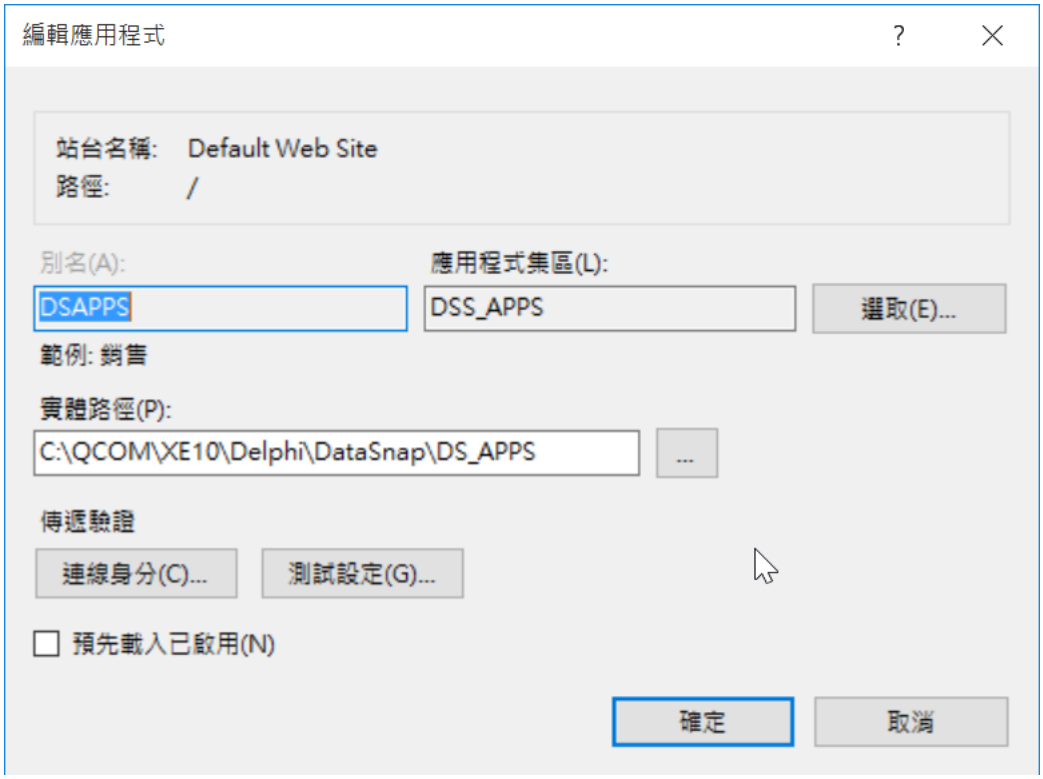
当然您可也可以决定使用特定的账户来执行：



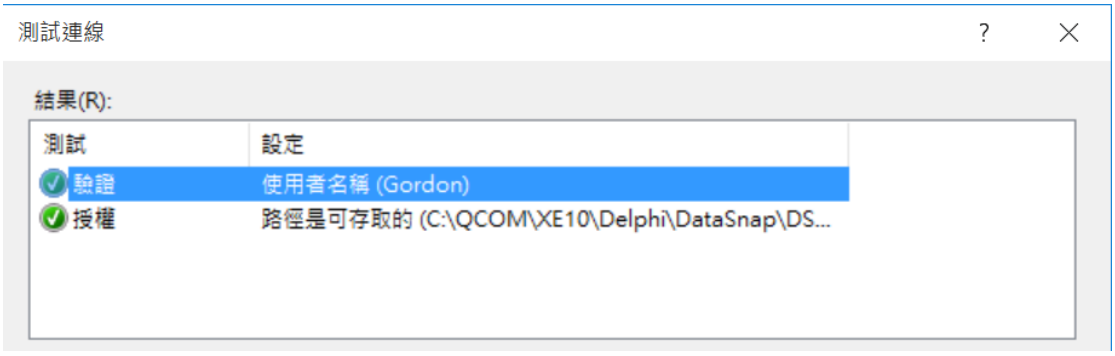
设定好应用程序集区之后接下来就可以建立站台了，请如下图点选 Default Web Site 右击鼠标再点选”新增祇用程序...”选项：



在编辑应用程序对话盒中取一个站台别名，选择程序集区为 DSS_APPS，再设定此站台使用的实体路径：

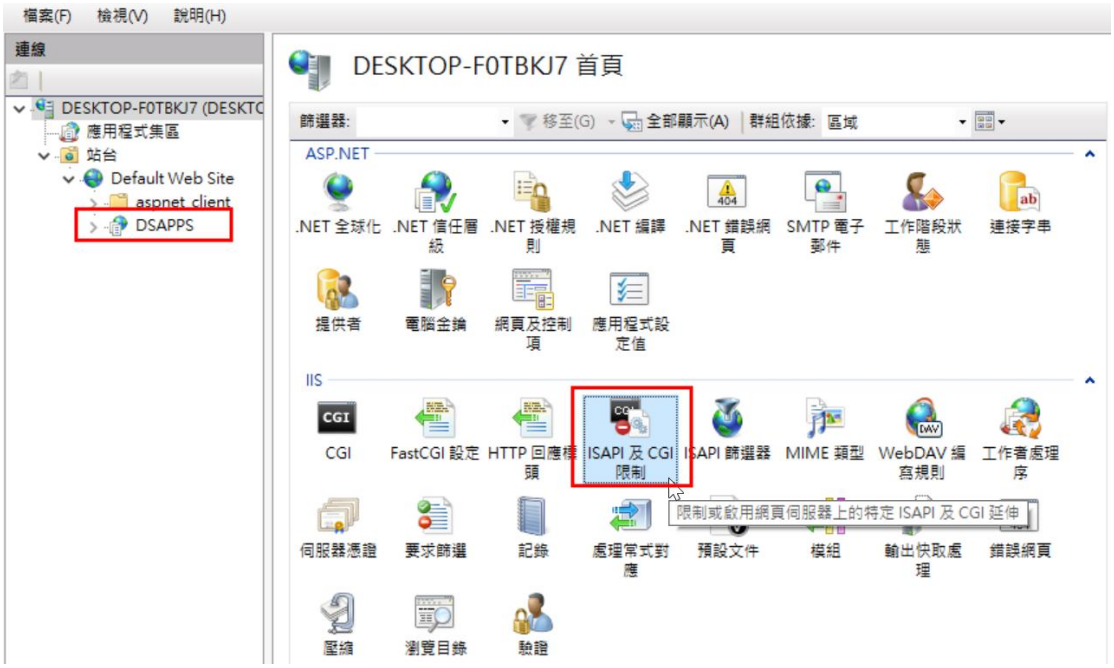


您可以点选"测试设定..."选项，如果您正确设定了使用账号的访问权限的话，应该就可以看到如下图显示测试成功的画面：

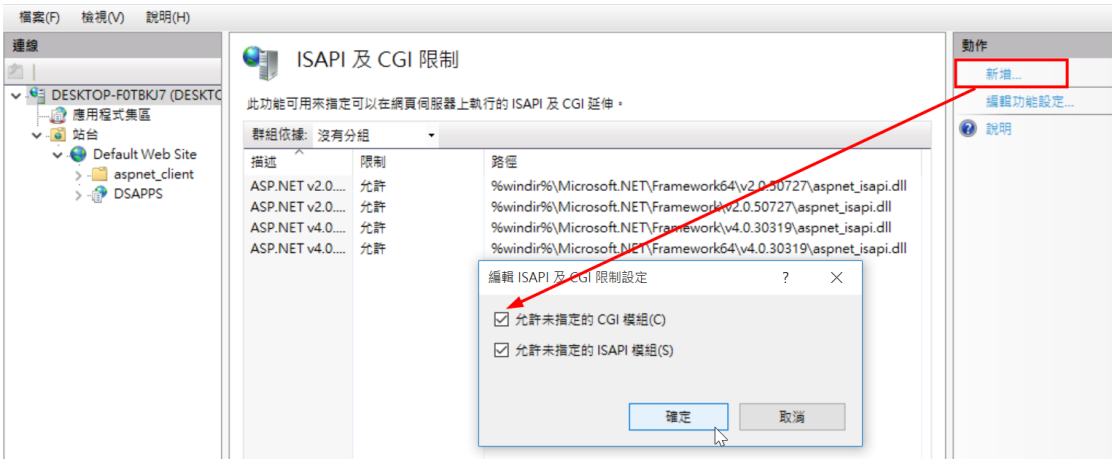


接着，由于我们部署的 DLL DataSnap 服务器是 ISAPI 应用程序，因此我们需要设定 IIS 允许客户端呼叫原生的 ISAPI DLL 应用程序。

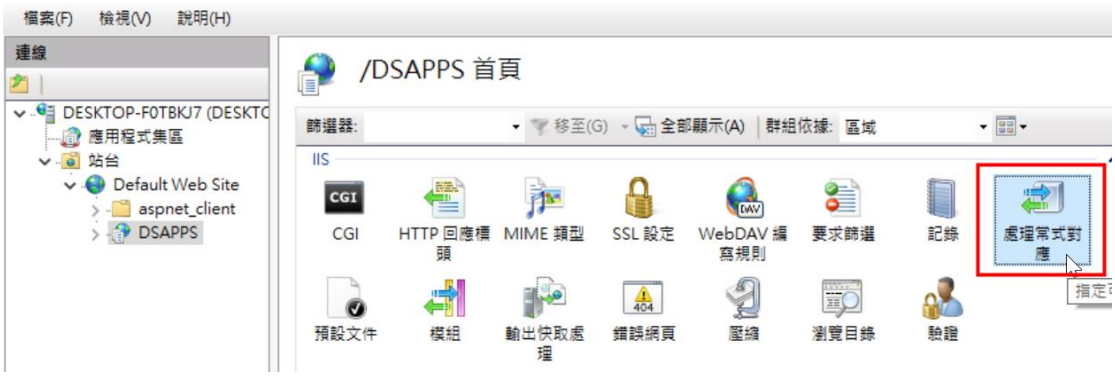
请在刚才建立的 DSAPPS 站台中点选 ISAPI 及 CGI 限制图像：



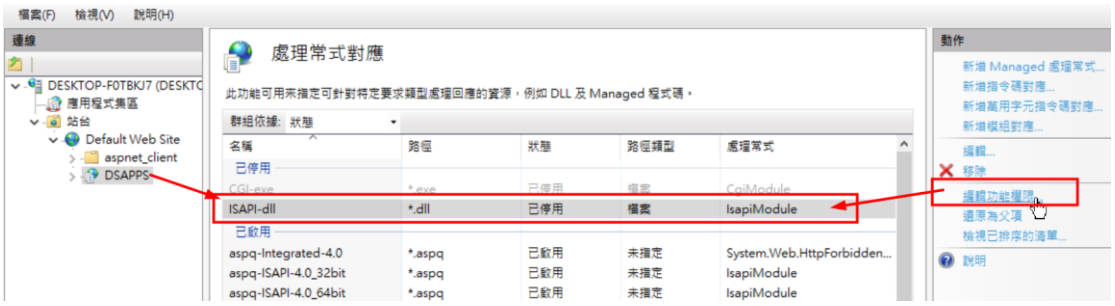
再点选右方的”新增...”选项，如下图勾选允许 CGI 和 ISAPI：



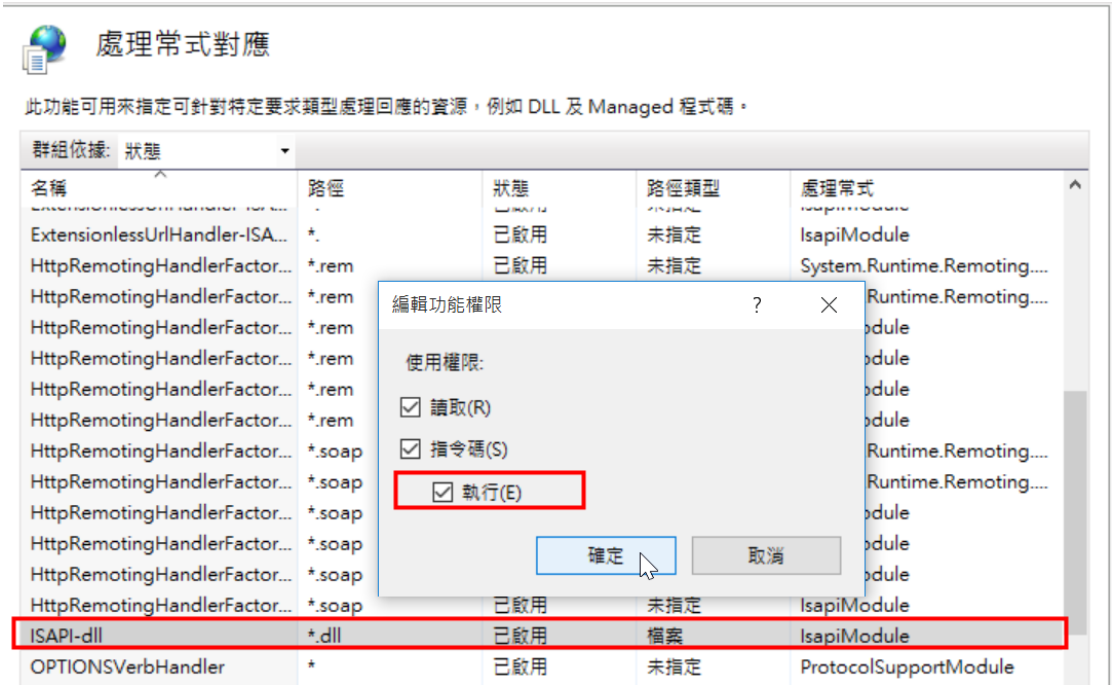
由于我们的 DataSnap 服务器是编译成 DLL 形式，因当我们使用 RESTful URL 执行服务方法时需要设定 IIS 执行此 DataSnap 服务器 DLL 而不是下载它。请点选下图的”处理程序对应”图像：



在下图中可以看到在内定上 IIS 是停用 ISAPI-dll 的，所以我们需要开启它。请点选右方的”编辑功能权限...”选项：

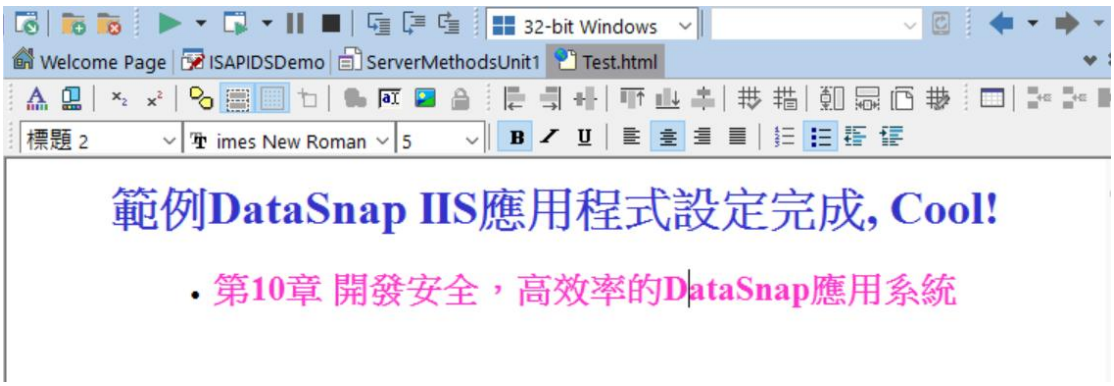


在出现的对话盒中勾选”执行”选项再点选确定按钮：

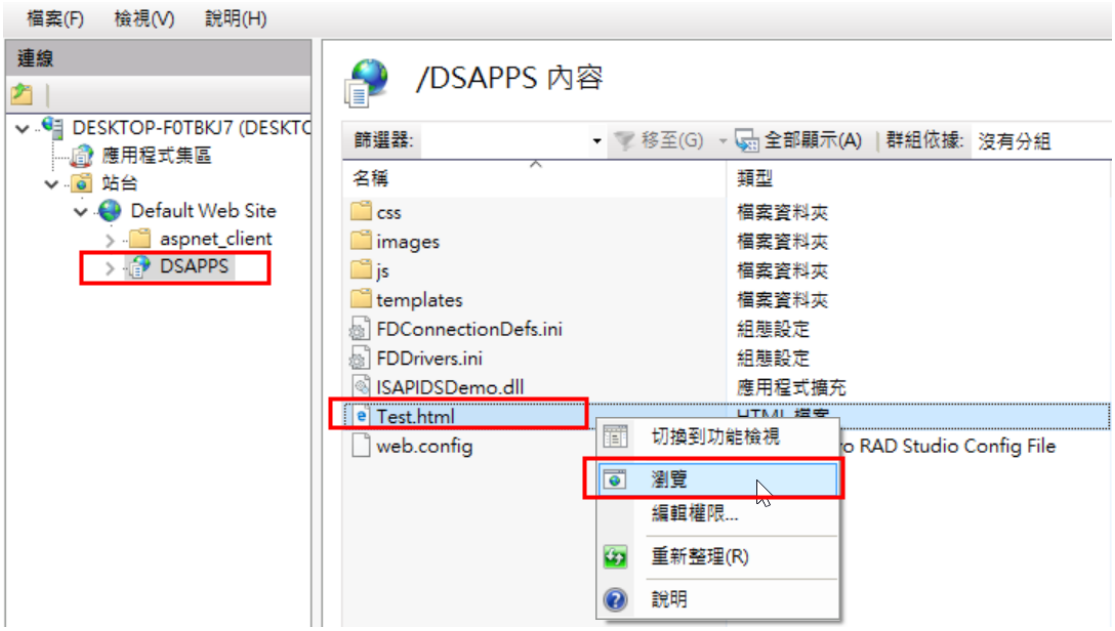


到了这里所有的设定应该就完成了，现在我们可以测试一下在前面设定的站台是否可以正常工作。我们可以在 Delphi IDE 中建立一个如下的 Test.html

档案(File | New | Other... | Web Document | HTML Page)并简单的打入一些测试文字：



然后把它储存到站台的实体目录中，回到 IIS 管理员点选 DSAPPS 站台，再点选下方的"内容检视页面"就可以看到 Test.HTML，使用鼠标右击它再点选"浏览"选项：

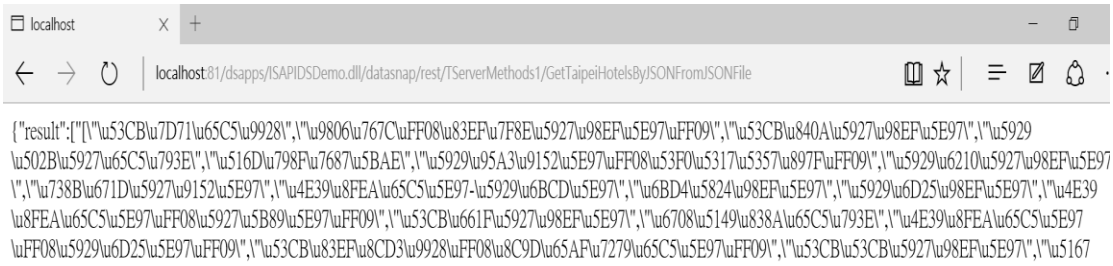


如果能在浏览器中看到如下的画面就代表我们建立的站台可以正常工作了：

由于这 2 个范例 DataSnap 服务器都是 RESTful 型态的服务器，因此我们现在就可以直接使用浏览器来呼叫其中的服务方法，例如使用如下的 URL 就可以呼叫 ISAPI 的 DataSnap 服务器：

```
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

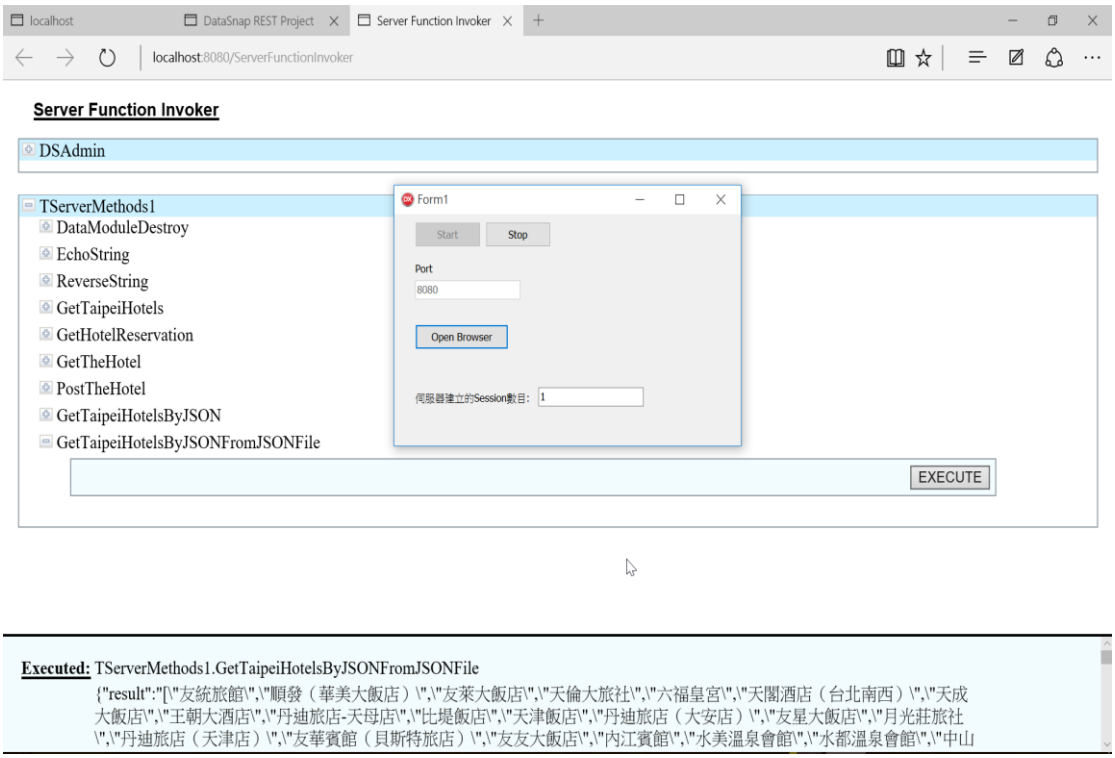
因此启动浏览器并使用如上的 URL 后如果看到如下的结果就代表成功的呼叫了 ISAPI 的 DataSnap 服务器提供的 GetTaipeiHotelsByJSONFromJSONFile()方法(别担心显示的结果，那只是 Unicode)：



如果要呼叫 EXE 的 DataSnap 服务器，那可以在浏览器中使用如下的 URL：


```
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

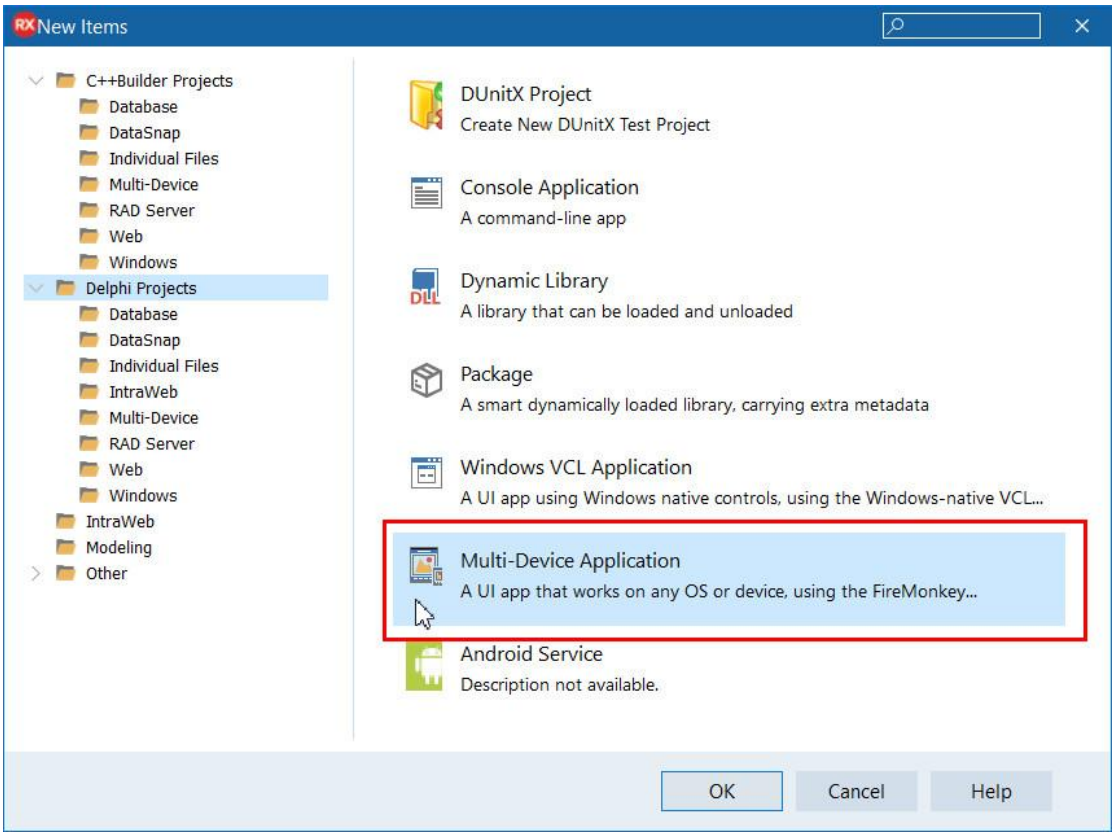
或是直接藉由 EXE 的 DataSnap 服务器启动浏览器来呼叫 GetTaipeiHotelsByJSONFromJSONFile()方法，如果看到如下的结果画面代表也成功的呼叫了 EXE 的 DataSnap 服务器：



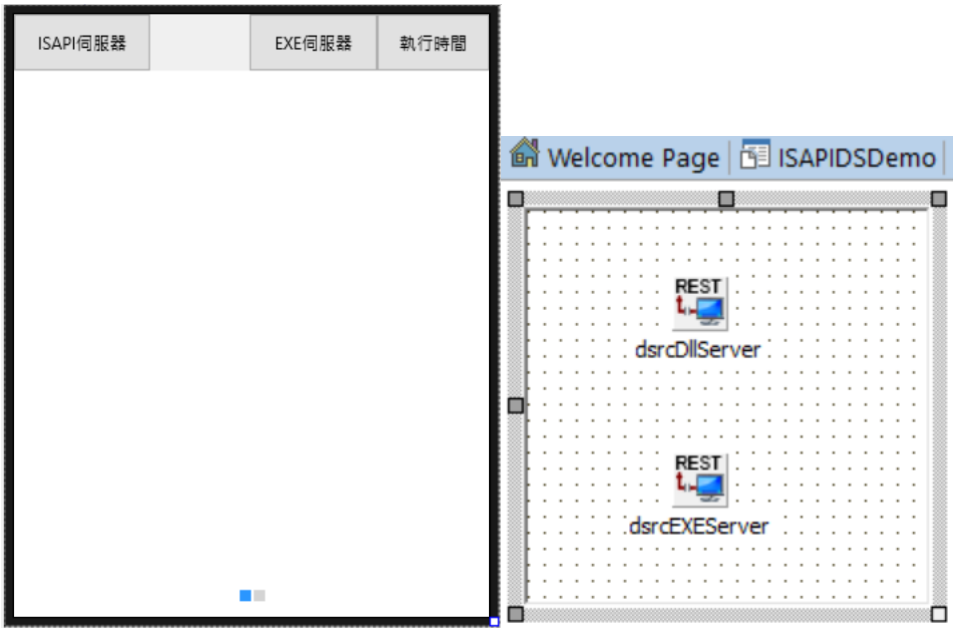
现在已经测试了不管是 EXE 或是 DLL 型态的 DataSnap 服务器都可以正常的工作，接着让我们开发一个简单的 FireMonkey 客户端来呼叫 GetTaipeiHotelsByJSONFromJSONFile() 方法并准备开始比较这 2 种 DataSnap 服务器的执行效率，更重要的是开始调校它们的执行效率。

10-1-2 开发客户端

先在项目群组中建立一个 Multi-Device 项目：



再于 Multi-Device 项目中建立一个数据模块，放入 2 个 TDSRestConnection 组件分别链接到 DLL 和 EXE 的 DataSnap 服务器，再于主表格中使用 2 个 TButton 组件呼叫：



由于呼叫 DLL 和 EXE 的 DataSnap 服务器几乎是一样的，因此让我们只说明一个场景即可。下面是主表格中”EXE 服务器”按钮的实作程序代码，它呼

叫 GetTaipeiHotelsFromEXEServer()方法来执行 EXE 的 DataSnap 服务器中的服务方法：

```
procedure TfmMainForm.btnEXEGetHotelsClick(Sender: TObject);
begin
  GetTaipeiHotelsFromEXEServer;
  ShowRunTime('向 EXE 服务器取得旅馆时间');
end;
```

GetTaipeiHotelsFromEXEServer() 藉由建立客户端的 TServerMethods1Client 对象后就可直接呼叫 EXE 的 DataSnap 服务器中的 GetTaipeiHotelsByJSONFromJSONFile()方法：

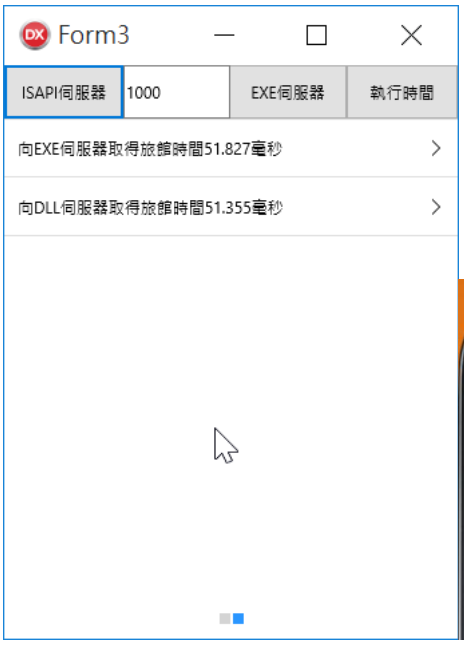
```
procedure TfmMainForm.GetTaipeiHotelsFromEXEServer;
var
  aServer : TServerMethods1Client;
  sJSONData : String;
begin
  lStart := Now;
  aServer := TServerMethods1Client.Create(dmClient.dsrcEXEServer);
  try
    sJSONData := aServer. GetTaipeiHotelsByJSONFromJSONFile();
    ShowJSONHotels(lvTaipeiHotels, sJSONData);
  finally
    aServer.Free;
    lEnd := Now;
  end;
end;
```


由于 GetTaipeiHotelsByJSONFromJSONFile()方法回传 JSON 数组的数据，因此 ShowJSONHotels()方法只是使用 TJSONTextReader 类别对象解析其中的数据并显示在 TListView 组件中：

```
procedure TfmMainForm.ShowJSONHotels(aLV : TListView; sHotels: String);
var
  sr : TStringReader;
  jr : TJSONTextReader;
  iCount : Integer;
begin
  aLV.Items.Clear;
```

```
  sr := TStringReader.Create(sHotels);
  jr := TJSONTextReader.Create(sr);
  iCount := 0;
  try
    while (jr.Read) do
    begin
      case jr.TokenType of
        TJsonToken.String:
        begin
          aLV.items.Add.Text := jr.Value.ToString;
          Inc(iCount);
        end;
      end;
    end;
  finally
    aLV.items.Add.Text := '一共取得 ' + iCount.ToString() + ' 笔资料';
    sr.Free;
    jr.Free;
  end;
end;
```


下面的 2 个画面是分别用 Windows 和 Android 手机呼叫 1000 次后平均每一次存取 396 笔台北旅馆名称的执行结果。



从上面的执行结果来看似乎 DLL 型态的 DataSnap 服务器执行效率虽然比 EXE 型态的 DataSnap 服务器来得好，但也没有太大的差距，那么值得花那么多的功夫使用和部署 DLL 型态的 DataSnap 服务器吗？

其实上面的执行结果只是单一客户端，多次呼叫的结果。然而在实际的应用中 DataSnap 服务器会服务大量的客户端，因此我们应该使用大量客户端呼叫这 2 种 DataSnap 服务器执再来比较，也才看得出来在真实世界中这 2 种 DataSnap 服务器的差异。

笔者并没有大量客户端的硬件环境，但没关系，我们可以藉由各种压力测试工具来模拟各种状况的客户端来对这 2 种 DataSnap 服务器进行压力测试并根据压力测试结果来进行效率调整，这正是下一节要讨论的内容。

10-2 DataSnap 服务器效能比较

现在我们需要使用实际的工具来找出上一小节 DLL 和 EXE 型态的 DataSnap 服务器的差别，笔者将使用下面的 3 个开源工具来对不同型态 DataSnap 服务器进行压力测试：

测试工具	URL
ApacheBench	http://httpd.apache.org/docs/2.2/programs/ab.html
WeigHttp	https://build.opensuse.org/package/show/home:stbuehler:lighttpd-utils/weighttp
Apache JMeter	http://jmeter.apache.org/

同时笔者也将使用下面的商用工具来对不同型态 DataSnap 服务器进行压力测试：

测试工具	URL
WAPT	http://www.loadtestingtool.com/

一开始让我们直接测试前面开发的 EXE 和 DLL 型态的 DataSnap 服务器，看看这 2 者之间的真实差异。在下面的测试中，笔者是在 VMWare 的虚拟机中执行压力测试，使用 4G Ram 以及 2.3G 的 Intel Core i7 CPU。

现在请执行前面的范例 EXE DataSnap 服务器并启动 IIS。

ApacheBench

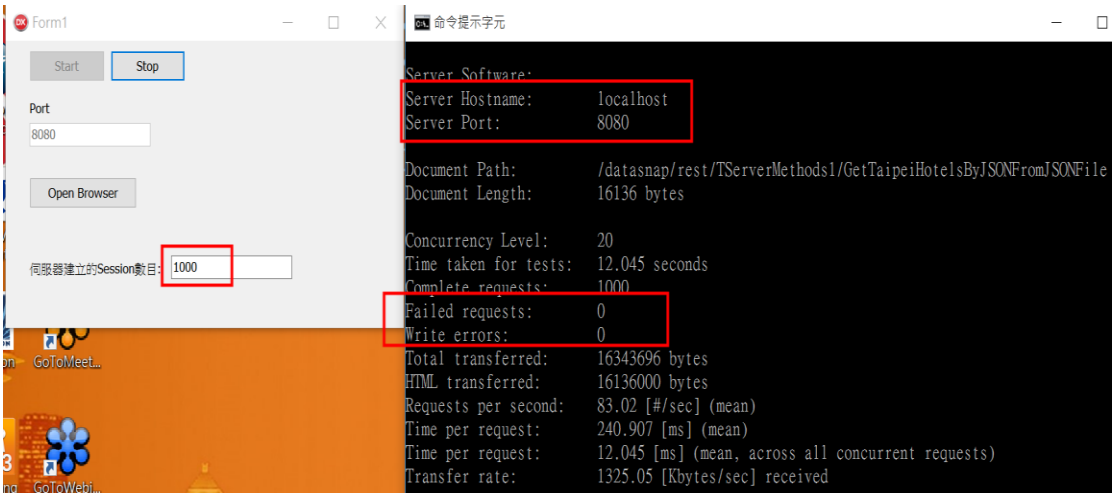
首先让我们使用下面的指令来仿真 20 个客户端提出 1000 个请求，分别呼叫 EXE 和 DLL 的 DataSnap 服务器，下面是呼叫 EXE DataSnap 服务器的指令：

```
ab -n 1000 -c 20
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

下面是呼叫 DLL DataSnap 服务器的指令：

```
ab -n 1000 -c 20
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
eiHotelsByJSONFromJSONFile
```

从下图可以看到当 ApacheBench 向 EXE 型态的 DataSnap 服务器提出请求时 EXE 型态的 DataSnap 服务器在伺服端建立了 1000 个 Session，而且没有发生任何的错误：



但如果我们继续增压模拟 40 个客户端：

```
ab -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

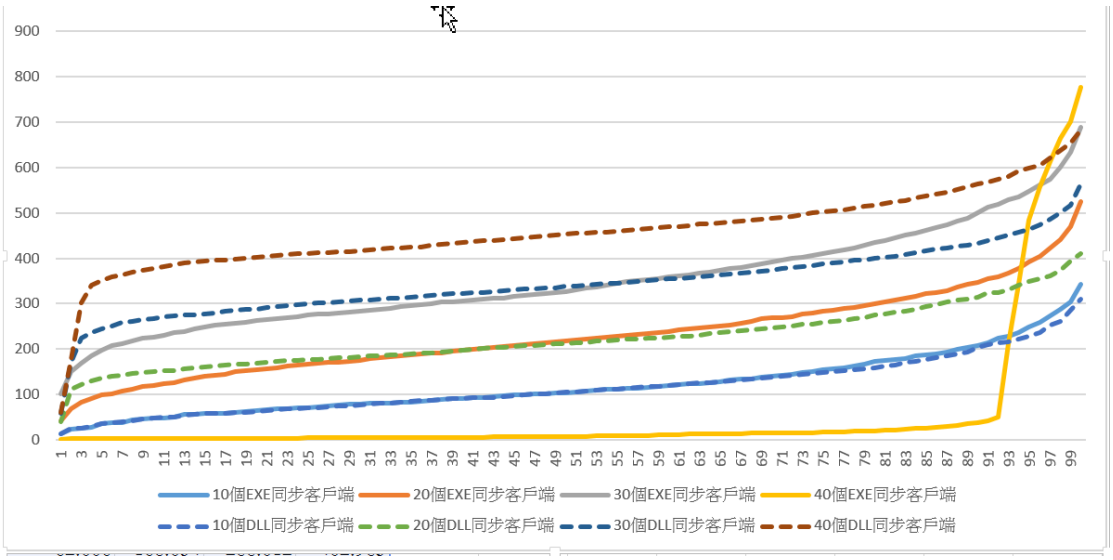
就会看到 EXE 型态的 DataSnap 服务器开始出现错误：

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        74 bytes

Concurrency Level:      40
Time taken for tests:   1.455 seconds
Complete requests:      1000
Failed requests:        80
   (Connect: 0, Receive: 0, Length: 80, Exceptions: 0)
Write errors:           0
```

但同样仿真 40 个客户端 DLL 型态的 DataSnap 服务器仍然不会出现任何的错误：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsB
yJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      40
Time taken for tests:   11.600 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
```

笔者使用 ApacheBench 同时模拟 10，20，30 和 40 个客户端绘出下面的比较图，徙这第 1 个比较图中可以看到 2 个现象，第 1 是 DLL 型态的 DataSnap 服务器随着客户端要求次数愈来愈后执行效率都比 EXE 型态的 DataSnap 服务器好，另外就是下图最下方的 40 个 EXE 同步客户端很明显的出了问题，无法响应客户端的请求：



徙上面的观察可以了解 EXE 型态的 DataSnap 服务器在客户端超过 30 左右之后就会开始发生问题，但使用同一份程序的 DLL 型态的 DataSnap 服务器从下图可以看到 ApacheBench 在笔者的虚拟机中持续加压到 500 个客户端时仍然没有发生任何的错误：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsB
yJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      500
Time taken for tests:   11.694 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16374671 bytes
HTML transferred:       16136000 bytes
Requests per second:    85.51 [#/sec] (mean)
Time per request:       5847.245 [ms] (mean)
Time per request:       11.694 [ms] (mean, across all concurrent requests)
Transfer rate:          1367.39 [Kbytes/sec] received
```

EXE 型态的 DataSnap 服务器只能服务 30 多个客户端的应用是可能被接受的，如果真的如此那 EXE 型态的 DataSnap 服务器只能使用来做上课学习之用，无法被使用在实际的应用中，但为什么使用完全相同程序代码的 DLL 型态的 DataSnap 服务器却可以服务超过 500 个客户端呢？因此这其中一定发生了什么问题，但在找出问题并解决之前，让我们再用其他的测试工具来加压看看，是不是也会反映相同的现象。

WeigHttp

WeigHttp 是笔者另外一个常使用的工具，它和 ApacheBench 使用的方法差不多，只是 WeigHttp 的好处是如果测试的机器如果有多核心的话，那么 WeigHttp 可以同时使用多核心同时加压测试让测试过程更快速完成。

我们使用下面的命令行测试 2 种 DataSnap 服务器：

```
weighttp -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

```
weighttp -n 1000 -c 40
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

从下面测试 EXE 型态的 DataSnap 服务器的结果来看 WeigHttp 也明确的指出用 40 个客户端测试时就会出现错误：

但对于 DLL 型态的 DataSnap 服务器即使使用 500 客户端，使用 10 个线程加压测试：

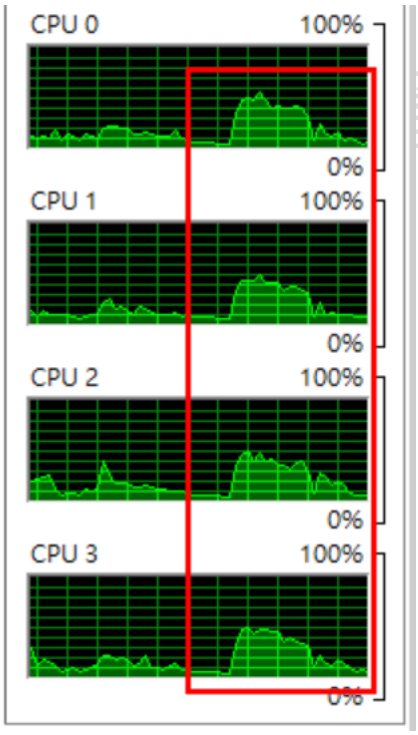
```
weighttp -n 1000 -c 500  -t 10
http://localhost:81/dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaip
```

```
eiHotelsByJSONFromJSONFile
```

我们仍然可以从下图看到 DLL 型态的 DataSnap 服务器完全可以应付请求，不会发生错误：

```
Progress: 100% done.

Finished in 15 sec, 74 millisec and 999 microsec,
          66 req/s, 1060 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          1000 succeeded, 0 failed, 0 errored.

Status codes: 1000 2xx, 0 3xx, 0 4xx, 0 5xx.

Traffic: 16374668 bytes total, 238668 bytes http, 16136000 bytes data.
```

而且 OS 也可明确看到 WeigHttp 同时使用了笔者 VM 中的 4 个核心同时加压测试：

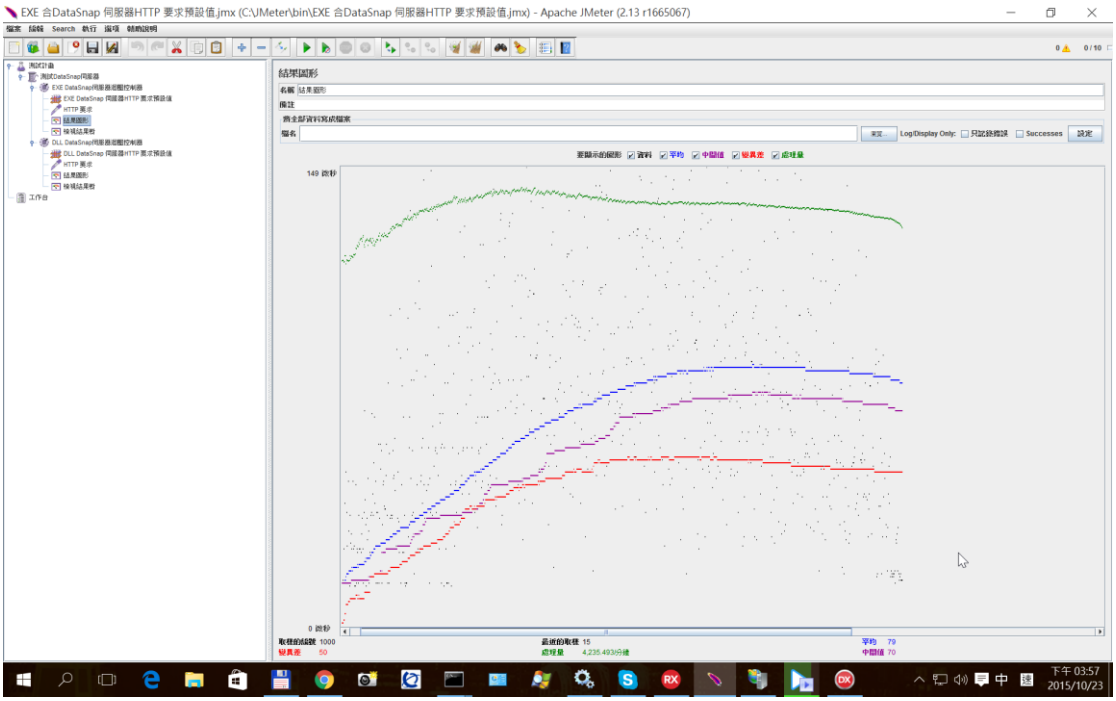


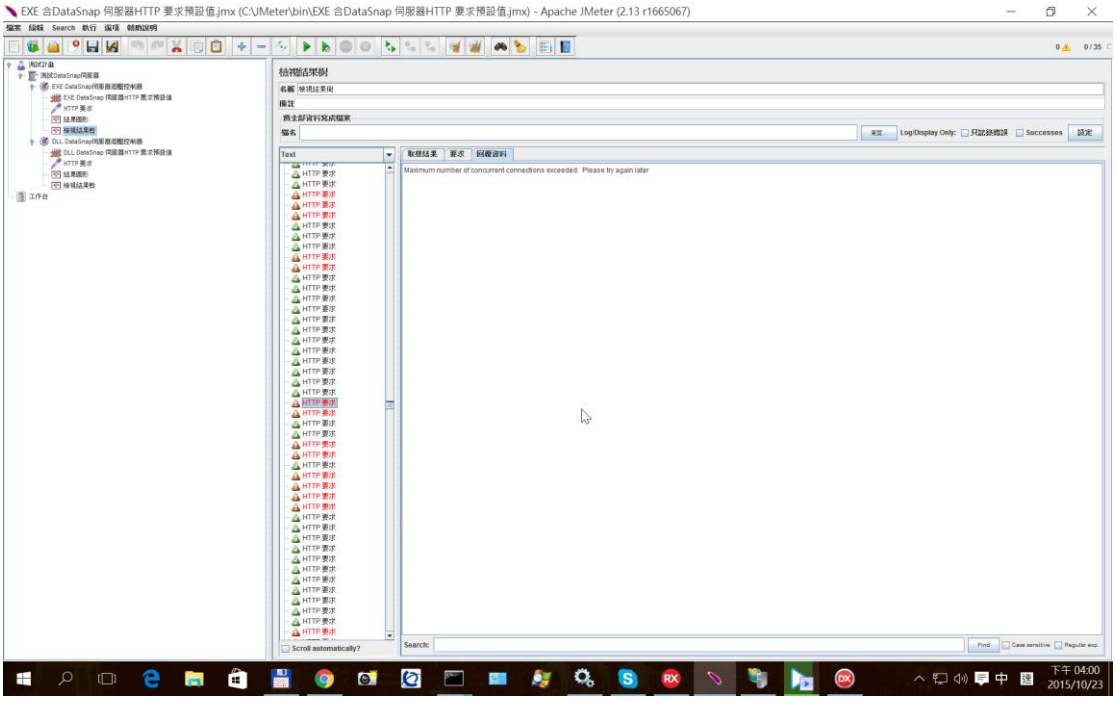
Apache JMeter

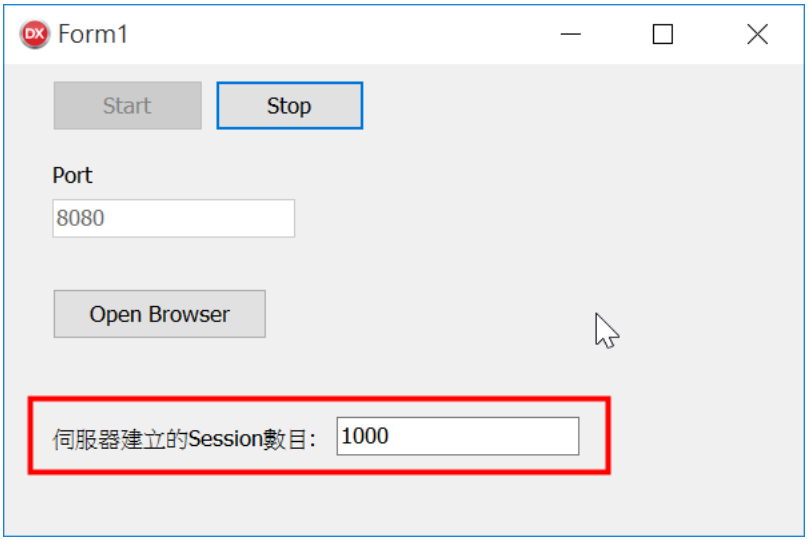
最后让我们使用 Apache 的 JMeter 来测试这 2 种 DataSnap 服务器，JMeter 应该是这 3 个压力测试工具中最强大的，功能繁多且可提供详细的结果

报告，例如 JMeter 就可以告诉我们当 EXE 型态的 DataSnap 服务器在服务 40 个客户端时发生了什么错误。

从下面测试 EXE 型态的 DataSnap 服务器的结果来看 JMeter 也明确的指出用 40 个客户端测试时就会出现错误：



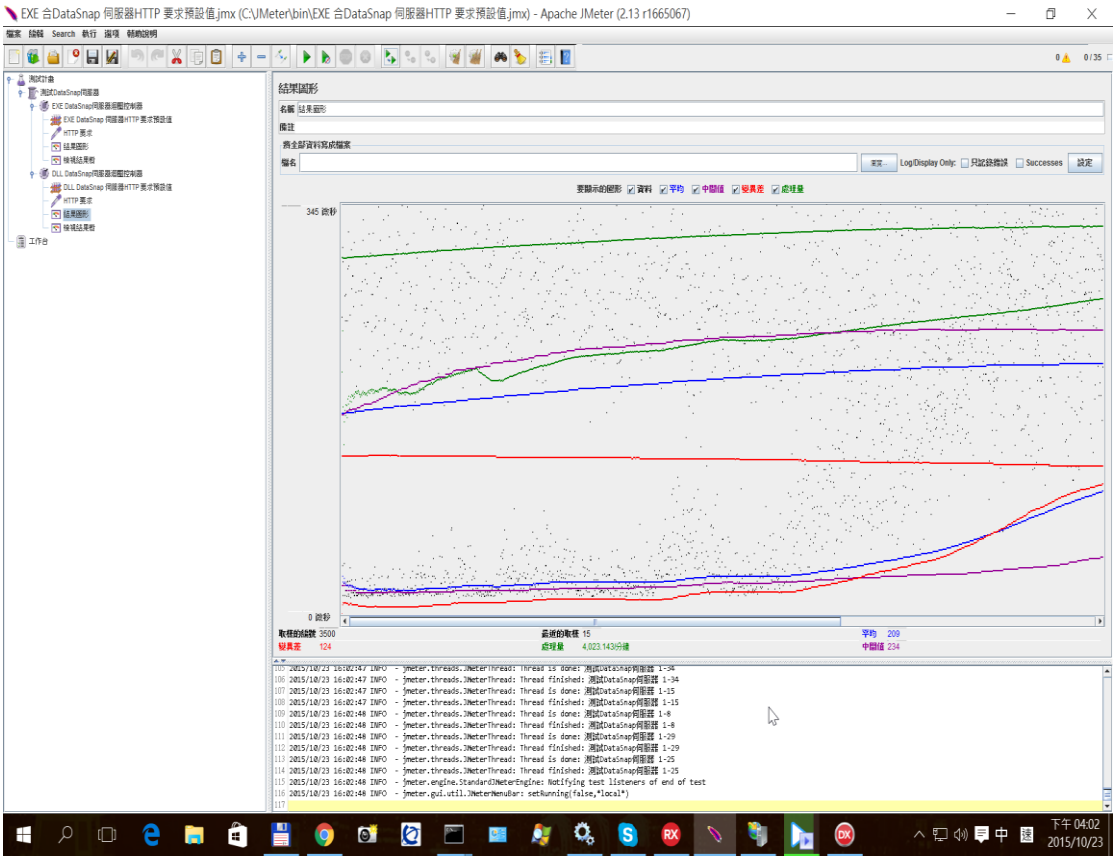




从 JMeter 出的报告可以明确看到 EXE 型态的 DataSnap 服务器已经超过最大连结数目，因此无法再让客户端连结，自然也就无法提供客户端服务了：

```
Thread Name: 测试 DataSnap 服务器 1-13
Sample Start: 2015-11-02 10:55:46 TST
Load time: 17
Connect Time: 1
Latency: 17
Size in bytes: 233
Headers size in bytes: 159
Body size in bytes: 74
Sample Count: 1
Error Count: 1
Response code: 500
Response message: Internal Server Error
. . .
Maximum number of concurrent connections exceeded.  Please try again later
```


但使用 JMeter 测试 DLL 型态的 DataSnap 服务器同样可以看到完全没有错误：



10-3 调校效能

在前面小节中使用压力测试让我们真实看到了 EXE 和 DLL 型态 DataSnap 服务器的差异，EXE 型态的 DataSnap 服务器在客户端超过 32 个之后就会有错误发生，发生的错误种类是 DataSnap 服务器已无法服务客户端的请求，相反的 DLL 型态的 DataSnap 服务器在却可同时服务超过 500 个客户端，为什么这 2 种型态的 DataSnap 服务器有如此大的差距？有没有可能让这 2 种型态的 DataSnap 服务器有更好的执行效率呢？

这正是本小节要讨论的重点。

调校 EXE 型态的 DataSnap 服务器

调校 EXE 型态的 DataSnap 服务器的第 1 步是解除在 30~40 个同步客户端请求时发生”最大连结数目”错误的状况。由于 DataSnap 是使用 Indy 组件组做为核心通讯技术，因此这应该是 Indy 或是 DataSnap 框架的问题，因此我们可以在 IDE 中使用搜寻功能寻找 Indy 和 DataSnap 原始码中有”MAXConnection”的字眼

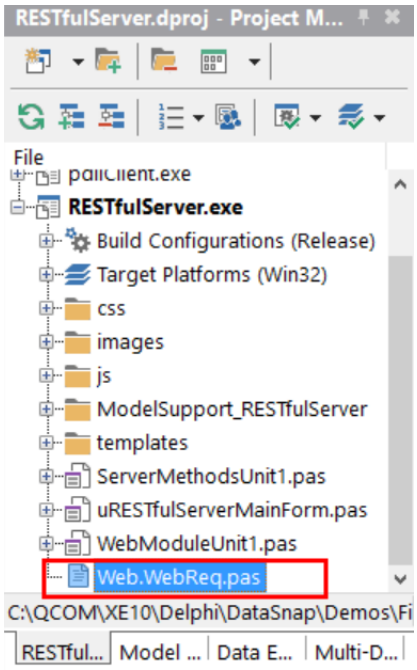
很快的就可以在 TWebRequestHandler 类别中找到如下的程序代码，我们可以看到 TWebRequestHandler 设定最大的连结数目是 32，这也解释了为什么前面 EXE 型态的 DataSnap 服务器在 30~40 个同步客户端请求时便会发生错误而且显示 " 服务器已经超过最大连结数目 " 的错误：

```
constructor TWebRequestHandler.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FCriticalSection := TCriticalSection.Create;
  FActiveWebModules := TList.Create;
  FInactiveWebModules := TList.Create;
  FMaxConnections := 32;
  FCacheConnections := True;
end;
```

因此让我们先解除这个 DataSnap 框架限制，看看有什么效果。

但使用同样 DataSnap 框架原始码的 DLL 型态的 DataSnap 服务器不会有这种错误呢？读者可以先想想吗？

璋在请把 c:\Program Files (x86)\Embarcadero\Studio\17.0\source\data\dsnap\ 目录下的 Web.WebReq.pas 拷贝到前面范例 RESTfulServer.dproj 项目的目录中，再把 Web.WebReq.pas 加入到项目中：



开启专案中的 Web.WebReq.pas 原始码，修改 FMaxConnections 为 0，代表不限制客户端的连结数目：

```
constructor TWebRequestHandler.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FCriticalSection := TCriticalSection.Create;
  FActiveWebModules := TList.Create;
  FInactiveWebModules := TList.Create;
//  FMaxConnections := 32;
  FMaxConnections := 0;
  FCacheConnections := True;
end;
```

再重新编译此 EXE 型态的 DataSnap 服务器并重新执行它，现在再让我们使用压力测试工具来测试这个修改过的 EXE 型态的 DataSnap 服务器，看看它现在是否能处理超过 32 个同步客户端的请求。

让我们使用 40 个同步客户端做为测试基准，测量它的反应时间做为随后调整效率的起始值：


```
ab -n 1000 -c 40
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSO
NFile
```

下面是 ApacheBench 的测试结果：

```
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      40
Time taken for tests:   11.594 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343669 bytes
HTML transferred:       16136000 bytes
Requests per second:    86.25 [#/sec] (mean)
Time per request:       463.775 [ms] (mean)
Time per request:       11.594 [ms] (mean, across all concurrent requests)
Transfer rate:          1376.58 [Kbytes/sec] received
```

下面是 WeigHtpp 的测试结果：

```
Finished in 11 sec, 963 millisec and 999 microsec,
          83 req/s, 1334 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          1000 succeeded, 0 failed, 0 errored.

Status codes: 1000 2xx, 0 3xx, 0 4xx, 0 5xx.
```

我们可以看到修改了 FMaxConnections 之后 EXE 型态的 DataSnap 服务器不再出现错误，最后让我们使用 JMeter 加大同步客户端到 500 个，看看它是否能像 DLL 型态的 DataSnap 服务器一样，完全可以应付客户端的请求。

下图是使用 JMeter 模拟 500 个同步客户端，我们可以确定现在 EXE 型态的 DataSnap 服务器也可以完全应付，不会发生错误了：

Label	取樣數	平均值	中間值	90% Line	95% Line	99% Line	最小值	最大值	錯誤率	處理量	每秒件位元組
HTTP 要求	500	3469	2860	7259	7693	7884	376	7987	0.00%	59.7/sec	952.8
總計	500	3469	2860	7259	7693	7884	376	7987	0.00%	59.7/sec	952.8

现在我们整理此基准 EXE 型态的 DataSnap 服务器的执行特征如下：

特性值	执行结果
同步客户端	40
每次请求费时	11.594ms
发生错误次数	0

EXE 型态的 DataSnap 服务器改良版 1

现在我们就可以开始调校 EXE 型态的 DataSnap 服务器的执行效率了，第 1 步当然就是加快它接受客户端请求的速度，这可以藉由增加它倾听线程数目。这是因为在它启动时它是建立了 TIdHTTPWebBrokerBridge 对象倾听客户端的请求：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  …
```

而 TIdHTTPWebBrokerBridge 是从 TIdCustomHTTPServer 类别继承下来的，因此 TIdHTTPWebBrokerBridge 是一个 Indy 框架的类别。

```
TIdHTTPWebBrokerBridge = class(TIdCustomHTTPServer)
```

TIdCustomHTTPServer 类别又是从 TIdCustomTCPServer 继承下来的：

```
TIdCustomHTTPServer = class(TIdCustomTCPServer)
```

而在 TIdCustomTCPServer 类别中有一个特性 ListenQueue：

```
  property ListenQueue: integer read FListenQueue write FListenQueue default
IdListenQueueDefault;
```

是 Indy 使用来倾听客户端请求的线程数目，它的内定值版设定成 15：

```
const
  IdListenQueueDefault = 15;
```

但这个值对现今的服务器硬件来说实在太保守了，甚至对笔者的 VM 来说都太保守了，因此先让我们加大这个设定值。

请开启 EXE 型态的 DataSnap 服务器的主窗体，修改它的 OnCreate 事件处理函式如下：


```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  FServer.ListenQueue := 150; //增加 Listener 线程数目
  SetupFDManager;
end;
```

增加 Indy 框架的倾听线程数目可以加快服务器响应的速度。

重新编译此 EXE 型态的 DataSnap 服务器并重新执行它，再使用上面相同的条件来测试此 EXE 型态的 DataSnap 服务器，下面是 ApacheBench 的结果：

```
Concurrency Level:      40
Time taken for tests:   11.575 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343663 bytes
HTML transferred:       16136000 bytes
Requests per second:    86.39 [#/sec] (mean)
Time per request:       463.019 [ms] (mean)
Time per request:       11.575 [ms] (mean, across all concurrent requests)
Transfer rate:          1378.83 [Kbytes/sec] received
```

特性值	执行结果
同步客户端	40
每次请求费时	11.575ms
发生错误次数	0

ApacheBench 显示 EXE 型态的 DataSnap 服务器现在的确更快了一点.

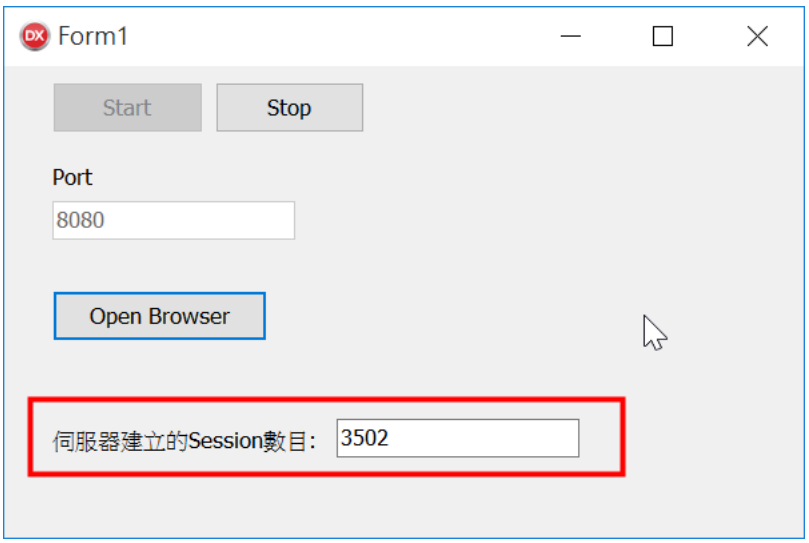
WeigHttp 也样显示现在的反应更快了：

```
Finished in 11 sec, 691 millisec and 999 microsec,
          85 req/s, 1365 kbyte/s.

Requests: 1000 total, 1000 started, 1000 done,
          1000 succeeded, 0 failed, 0 errored.
```

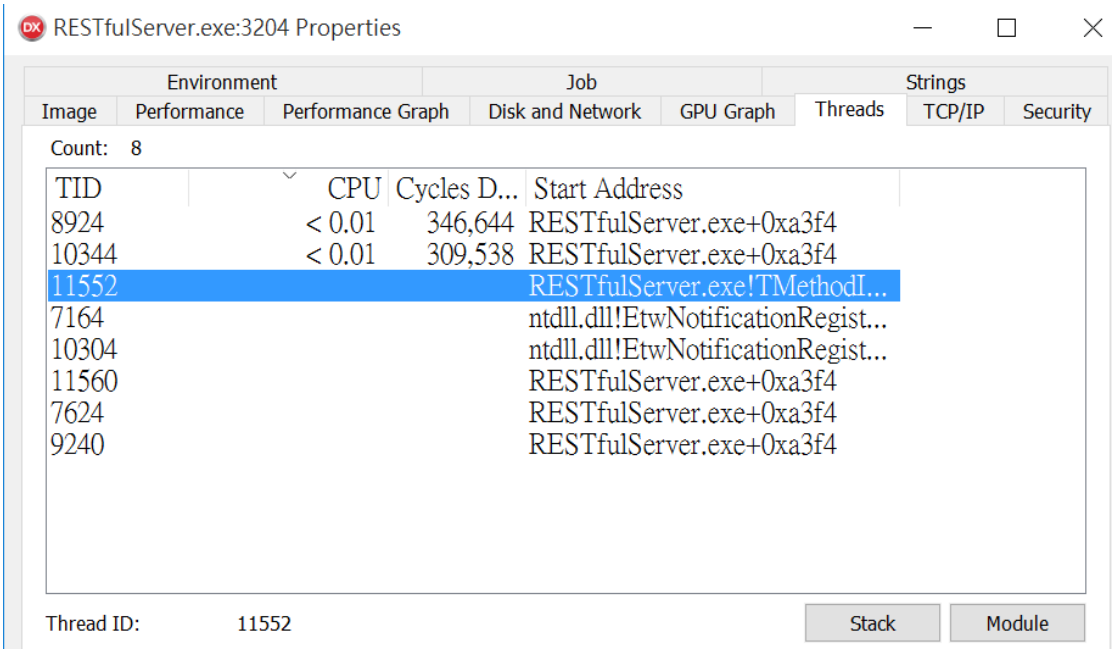
读者也许觉得这只快了一点点，但别忘了在实际上线后当大量的客户肯不断的提出请求并且持续执行 24*7 的应用中，这将会有更明显的差异。

EXE 型态的 DataSnap 服务器改良版 2

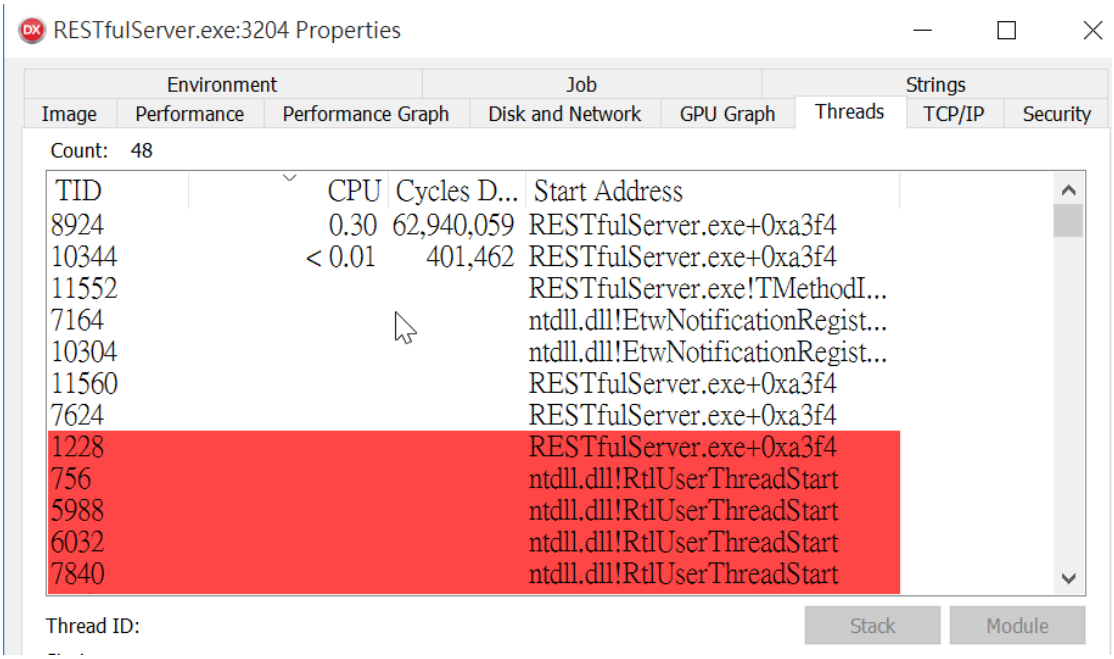
下一步改良版 DataSnap 服务器的执行效率方法来自观察当客户端连结并提出服务请求时，从 EXE 型态的 DataSnap 服务器的主表格可以看到每一个连结的客户端都会在 DataSnap 服务器建立一个 Session，如下所示：



如果我们启动任务管理器进一步观察 DataSnap 服务器，那么可以看到类似下面的初始状态，DataSnap 服务器在启动后会建立数个线程同时执行：



但当客户不断增加并提出请求时在任务管理器中我们会看到 DataSnap 服务器会不断的建立新的线程服务客户端并在服务完毕之后释放线程，如此不断的进行这种执行行为：



这个现象说明了 DataSnap 服务器浪费了许多时间不断的重复建立和释放线程，会造成这个现象的原因是 Indy 建立的线程池太小，因此当客户端数目愈来愈多时就造成线程不够使用，因此 DataSnap 服务器需要不断的建立和释放线程。因此要进一步改善 DataSnap 服务器执行效率，我们只需要启动 Indy 的线程池功能并增加线程池的大小即可改善 DataSnap 服务器不断的重复建立和释放线程的负荷。

Indy 框架的线程池功能是实作在 TIdSchedulerOfThreadPool 类别中：

```
TIdSchedulerOfThreadPool = class(TIdSchedulerOfThread)
```

而 TIdSchedulerOfThreadPool 是定义在 IdSchedulerOfThreadPool 程序单元中，因此我们需要在 DataSnap 服务器主窗体中加入使用这个程序单元：

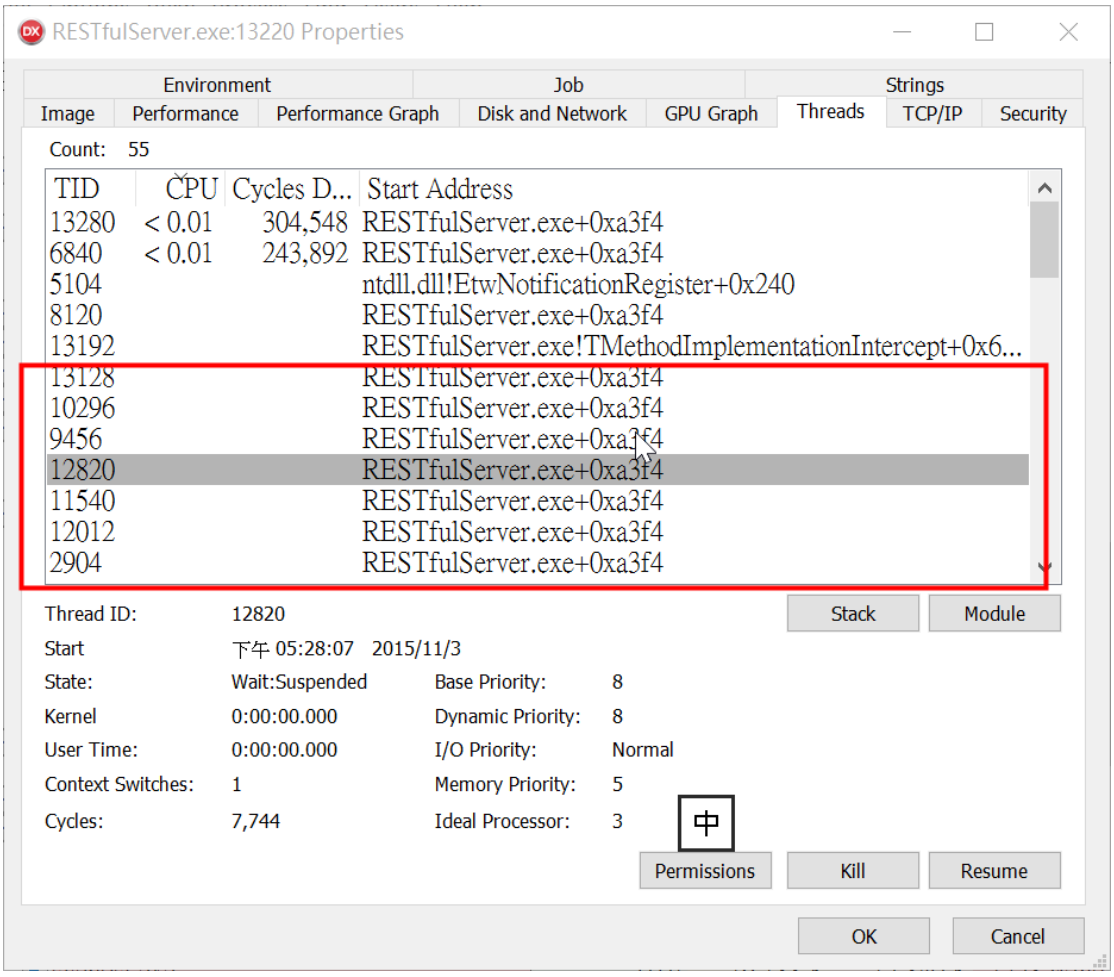
```
uses IdSchedulerOfThreadPool
```

并在主窗体的 OnCreate 事件中建立 TIdSchedulerOfThreadPool 对象，设定它的 PoolSize 特性值，PoolSize 特性即控制了线程池的大小，例如笔者在下面的程序代码中设定线程池的大小为 50，当然如果读者用的机器硬件配备很好，那么可以再增加到 100 左右，线程池大小设定需要根据实际的环境来决定。对于笔者的ＶＭ来说 50 已经很多了。

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Scheduler: TIdSchedulerOfThreadPool;
```

```
begin
  FServer := TIdHTTPWebBrokerBridge.Create(Self);
  FServer.ListenQueue := 150; //增加 Listener 线程数目

  Scheduler := TIdSchedulerOfThreadPool.Create(FServer); //增加线程池
  Scheduler.PoolSize := 50; //建线程池中预先建立的线程
  FServer.Scheduler := Scheduler;
  SetupFDManager;
end;
```

重新编译并执行此范例 DataSnap 服务器，使用 ApacheBench 来测试，并使用任务管理器观察范例 DataSnap 服务器，从下面的画面可以看到范例 DataSnap 服务器启动后就会建立 PoolSize 特性设定的线程数目，而且当客户端不断增加执行数目并提出请求时，范例 DataSnap 服务器几乎不会再不断的建立和释放线程，因此增加了 DataSnap 服务器的执行效率：



下面是 ApacheBench 在笔者 VM 中加压测试的结果，在客户端增加到 1000 个时，服务每一个客户端只需要 12.529 毫秒；

```
Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      1000
Time taken for tests:   12.529 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16343684 bytes
HTML transferred:       16136000 bytes
Requests per second:    79.82 [#/sec] (mean)
Time per request:       12528.563 [ms] (mean)
Time per request:       12.529 [ms] (mean, across all concurrent requests)
Transfer rate:          1273.94 [Kbytes/sec] received
```

JMeter 加压测试则显示在笔者的 VM 中每一秒在 90%的请求都不超过 5435 毫秒：

Label	取樣數	平均值	中間值	90% Line	95% Line	99% Line	最小值	最大值	錯誤率	處理量	每秒千位元組
HTTP要求	500	3313	3293	5435	5531	5758	113	5947	0.00%	51.8/sec	826.7
總計	500	3313	3293	5435	5531	5758	113	5947	0.00%	51.8/sec	826.7

范例 DataSnap 服务器的确是愈来愈快而且可服务的客户端已经从 32 个到现在超过 1000 个都没问题了。

但，还能更快，服务更多的客户端吗？

EXE 型态的 DataSnap 服务器改良版 3

要再进一步调校范例 DataSnap 服务器的执行效率之前我们需要再次想想基本的问题，那就是要使用 DataSnap 服务器的原因为何？当然就程序代码开发而言使用 DataSnap 服务器有其好处，但在这里让我们专注在 DataSnap 服务器的应用原因。

应用 DataSnap 服务器可能有：

1. 想服务大量客户端

2. 想连结移动客户端和其他智能型设备

其实上面的 2 个原因互有牵连，因为就是要 " 结移动客户端和其他智能型设备 " 因此会有比以前的应用有更多的客户端。

而连结更多的客户端会对 DataSnap 服务器造成沉重的负荷是因为如果每一个客户端都需要对 DataSnap 服务器使用过多的资源，那么 DataSnap 服务器当然就只能服务一定数量的客户端。

因此在使用 DataSnap 服务器的多层应用中我们应该尽可能的减少每一个客户端在 DataSnap 服务器中使用 / 占用的资源：

1. 使用的资源是指服务程序代码要精简快速

2. 占用的资源是指使用资源之后立刻归还，不要无义意的占据

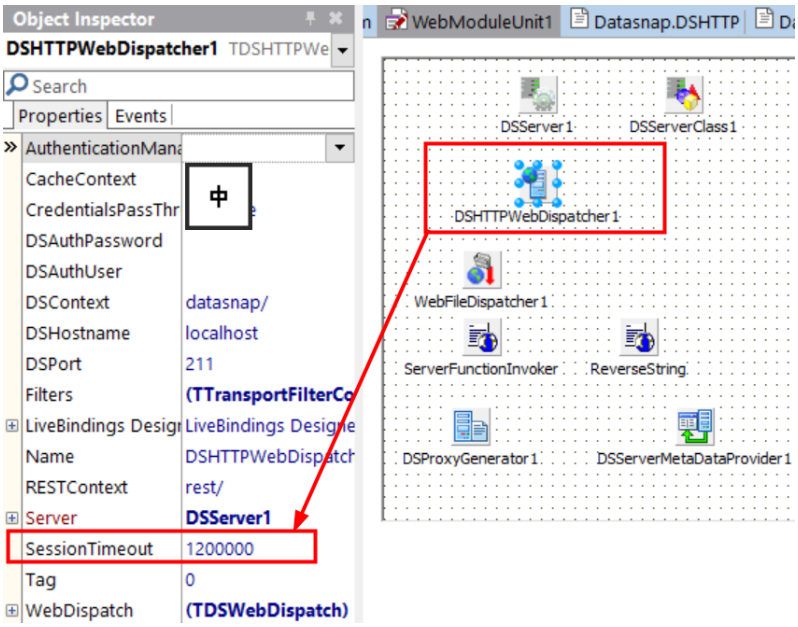
例如前面第 2 步的调校步骤，服务客户端的线程使用完毕之后立刻归还，重复使用。另外一个例子就是在以前的 C/S 架构中客户端对于数据库的链接是一直占据的，在 DataSnap 多层架构中应该是在客户端取得需要的数据之后立刻归还数据库链接，重复使用。

因此本步骤的调校就是再进一步主动释放客户端使用的资源，一旦 DataSnap 服务器服务完毕客户端的请求后就回收客户端使用的资源，重复使用。

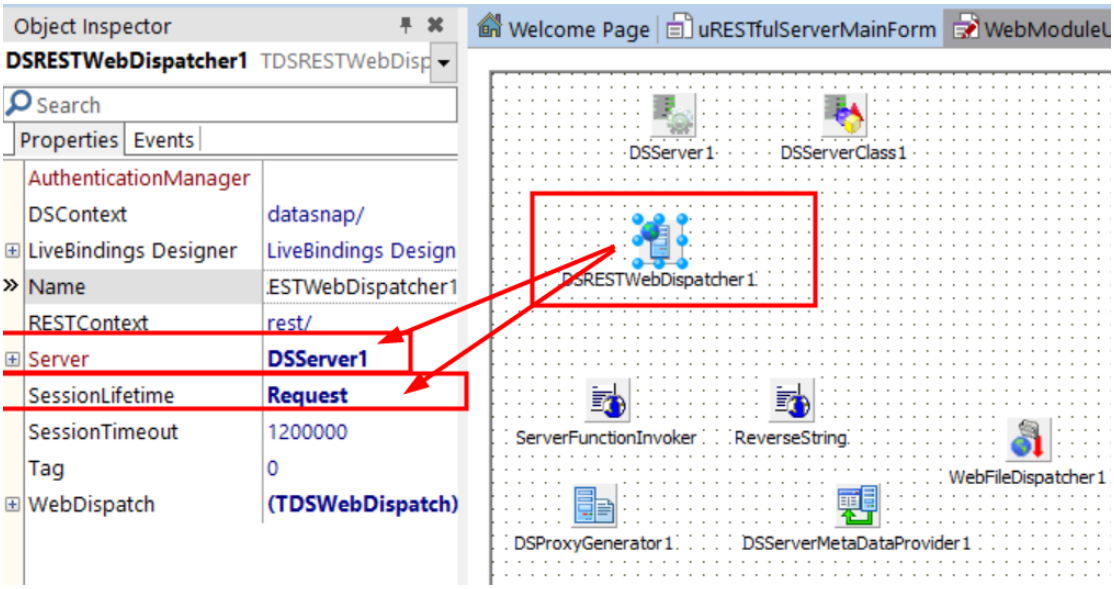
现在回到范例 DataSnap 服务器，由于它本身就是一个 RESTful 服务器，因此客户端在提出请求并取得结果之后本就应该离开并归还资源，但请读者开启范例项目中的 WebModuleUnit1 程序单元，在其中可以看到 DataSnap 处理 HTTP 命令的组件 TDSHTTPWebDispatcher，它的 SessionTimeout 特性值是设定为 1200000 毫秒，这代表在范例 DataSnap 服务器处理了客户端的请求后仍然会在 DataSnap 服务器中保留这个客户端的 Session 对象最少 20 分钟之后才会释放，因此如果有大量的客户端时这就对 DataSnap 服务器造成了沉重的负荷。

既然我们使用了 RESTful 的架构，因此就应该尽量在服务完客户端请求后就释放为客户端建立的 Session 对象。也许读者会立刻反应把 SessionTimeout 特性值设定为 0 不就好了？噢这样做有反效果，因为设定 SessionTimeout 特性值为 0 代表永不释放 Session 对象。

因此我们需要的组件是服务完客户端请求后能就自动释放 Session 对象，在 DataSnap Server 组件组中正好有这么一个组件：TDSRESTWebDispatcher。



我们可以使用 TDSRESTWebDispatcher 组件来取代原本的 TDSHTTPWebDispatcher 组件，因请在上图的 WebModuleUnit1 程序单元中加入一个 TDSRESTWebDispatcher 组件，再删除原来的 TDSHTTPWebDispatcher 组件，如下所示：



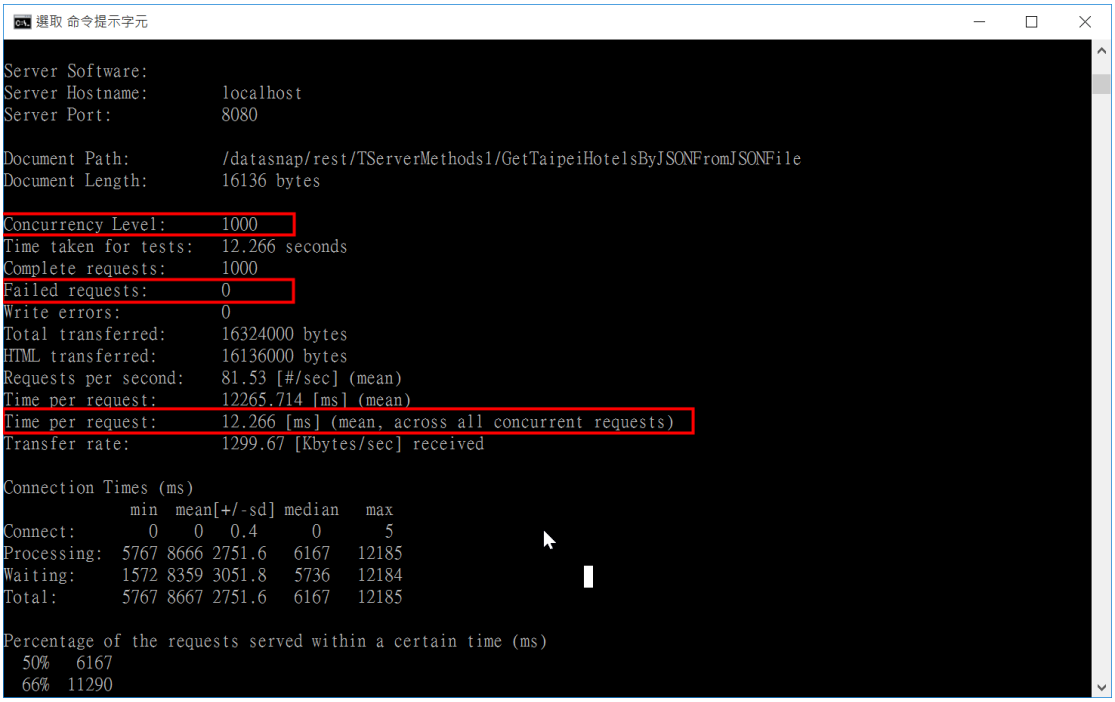
再于对象查看器中设定 TDSRESTWebDispatcher 组件的特性如下：

特性	特性值
Server	DSServer1
SessionLifetime	Request

重新编译并执行此范例 DataSnap 服务器，使用 ApacheBench 来测试：

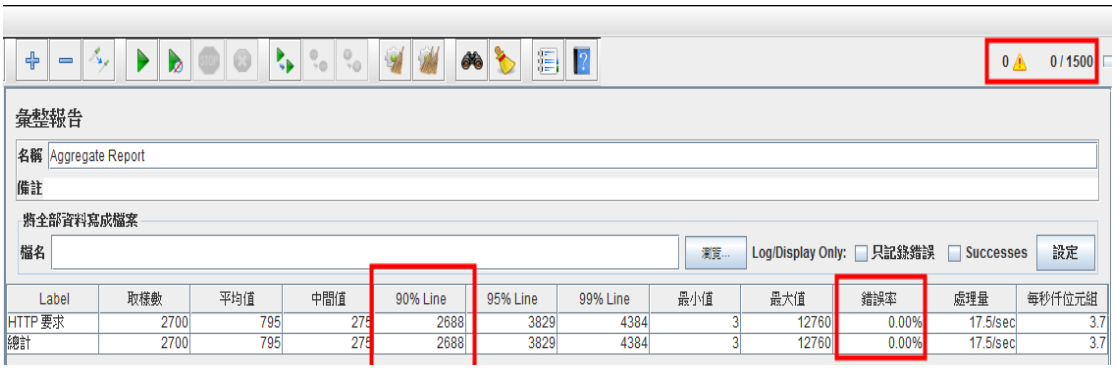
```
ab -n 1000 -c 1000
http://localhost:8080/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
```

从下面 ApacheBench 测试的结果来看使用 TDSRESTWebDispatcher 组件的范例 DataSnap 服务器可以更快的速度服务更多的客户端：



特性值	执行结果
同步客户端	1000
每次请求费时	12.266ms
发生错误次数	0

Jmeter 也显示这个调校的范例 DataSnap 服务器在 1000 个同步客户端不断请求服务时有 90%的客户端可以在 2688 毫秒服务完毕并且没有发生任何错误：



EXE 型态的 DataSnap 服务器到这里要再进一步的增加执行效率就需要讨论到设计架构和程序员撰写的程序代码质量了，在稍后我们会讨论一些其他的技巧再次增进 EXE 型态 DataSnap 服务器的执行效率。

还是不要忘记，在实际执行的应用环境中笔者还是建议使用 DLL 型态的 DataSnap 服务器。

调校 DLL 型态的 DataSnap 服务器

现在我们可以开始讨论如何增加 DLL 型态的 DataSnap 服务器的执行效率，上面对于增加 EXE 型态 DataSnap 服务器的技巧都可以使用在 DLL 型态的 DataSnap 服务器中，在下面的小节中讨论的技巧是只针对 DLL 型态的 DataSnap 服务器。

DLL 型态的 DataSnap 服务器改良版 4

对于 IIS 的 DLL 型态的 DataSnap 服务器而言，在进行了和上面类似的调整步骤之后，第 1 个额外的调校就是设定 IIS 对于客户端连结的数目限制，这可以在 IIS 主程序的地方设定它的 MaxConnections 特性值为 0，如下所示：

```
begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.WebModuleClass := WebModuleClass;
  SetupFDManager;
  Application.MaxConnections := 0;
  TISAPIApplication(Application).OnTerminate := TerminateThreads;
  Application.Run;
end.
```

进行了这个调整之后再使用 AppBench 对调整前和调整后比较我们可以看到调整后的执行速度再次有了明显的增加，

调整前：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemo.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      20
Time taken for tests:   14.124 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16374682 bytes
HTML transferred:       16136000 bytes
Requests per second:    70.80 [#/sec] (mean)
Time per request:       282.472 [ms] (mean)
Time per request:       14.124 [ms] (mean, across all concurrent requests)
Transfer rate:          1132.21 [Kbytes/sec] received
```

调整后：

```
Server Software:        Microsoft-IIS/10.0
Server Hostname:        localhost
Server Port:            81

Document Path:          /dsapps/ISAPIDSDemoStep4.dll/datasnap/rest/TServerMethods1/GetTaipeiHotelsByJSONFromJSONFile
Document Length:        16136 bytes

Concurrency Level:      20
Time taken for tests:   11.267 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      16355000 bytes
HTML transferred:       16136000 bytes
Requests per second:    88.75 [#/sec] (mean)
Time per request:       225.350 [ms] (mean)
Time per request:       11.267 [ms] (mean, across all concurrent requests)
Transfer rate:          1417.50 [Kbytes/sec] received
```

就一个简单的设定就伯有这么明显的差别。

DLL 型态的 DataSnap 服务器改良版 5

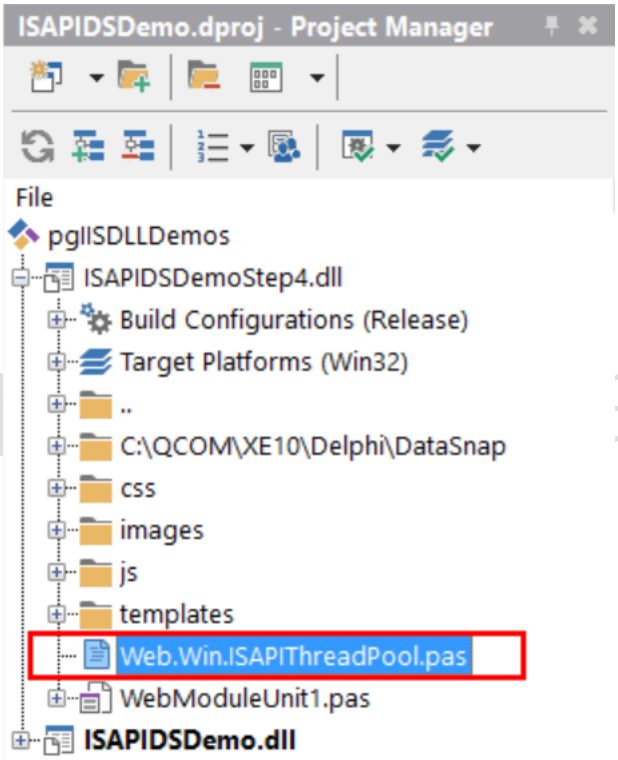
对于第 2 个 IIS DLL 型态的 DataSnap 服务器调整步骤就是设定它的线程连结池数目的设定。

在 Delphi 的 Web.Win.ISAPIThreadPool 程序单元中定义了 Delphi IIS 型态的应用程序使用的线程链接池数目，在内定上 Delphi 是设定使用 25 个线程：

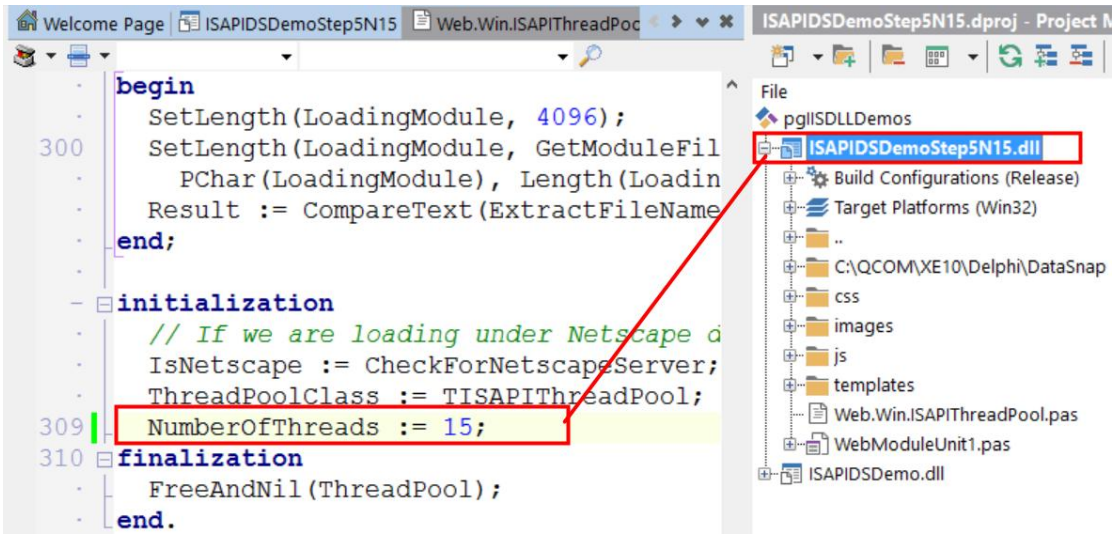
```
initialization
  // If we are loading under Netscape do not use the thread pool
```

```
IsNetscape := CheckForNetscapeServer;

ThreadPoolClass := TISAPIThreadPool;

NumberOfThreads := 25;
```

但这个内定值不一定是最好的，开发人员同样应该根据实际的环境来设定线程连结池的大小，因为这会影响执行效率。为了接下来比较不同 NumberOfThreads 设定值对于执行效率的影响，下面的图形显示到目前为止 2 个 DLL DataSnap 服务器项目，而 ISAPIDSDemoStep4 项目就是刚才 MaxConnections 特性值为的项目，现在让我们在此项目中加入 Web.Win.ISAPIThreadPool 程序单元，准备设定 NumberOfThreads 数值：



然后我们可以在 IDE 中设定不同的 NumberOfThreads 数值并产生不同的输出档：

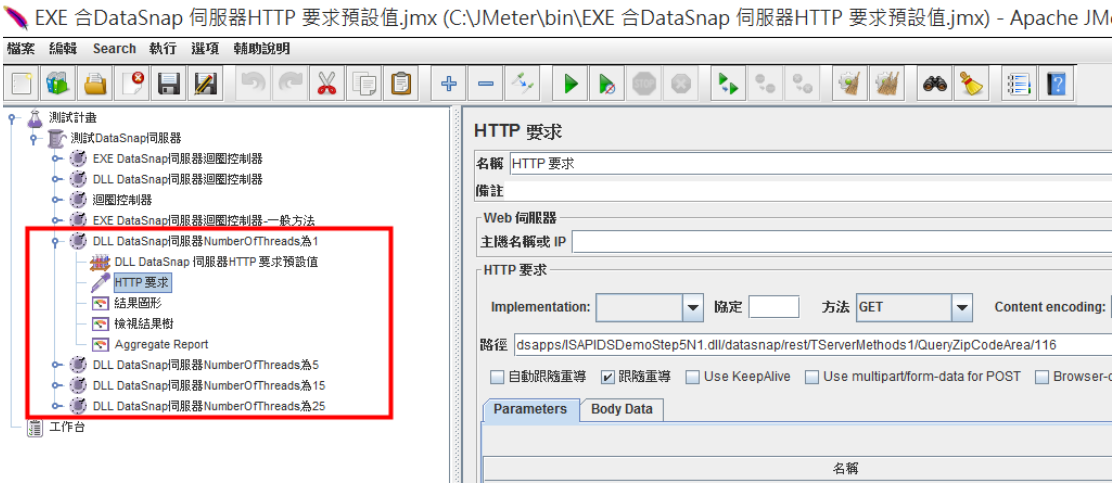


例如下面是分别设定 NumberOfThreads 数值为 1，5，15，25 编译出的如输出档：

ISAPIDSDemo	dll	7,887,360	2015/11/02 10:48	-a--
ISAPIDSDemoStep4	dll	7,885,824	2015/12/03 14:23	-a--
ISAPIDSDemoStep5N1	dll	7,885,824	2015/12/04 18:02	-a--
ISAPIDSDemoStep5N15	dll	7,885,824	2015/12/04 17:59	-a--
ISAPIDSDemoStep5N25	dll	7,885,824	2015/12/04 17:58	-a--
ISAPIDSDemoStep5N5	dll	7,885,824	2015/12/04 18:02	-a--

使用 ApacheBench 和 JMeter 测试上面不同 NumberOfThreads 数值设定的结果显示同时在 1000 个客户端存取服务时显示似乎 NumberOfThreads 设定在 10~25 之间的数值是拥有比较好的效率，不过当客户端超过 1000 个时，增加 NumberOfThreads 数值设定则似乎有比较少的无法服务错误发生。

```
ab -n 1000 -c 1000
http://localhost:81/dsapps/ISAPIDSDemoStep5N100.dll/datasnap/rest/TServerMethods1/QueryZipCodeArea/116
```



由于笔者个人使用的 VM 是 Windows 7 专业版，如果把调校到这里的范例 DLL 型态的 DataSnap 服务器部署到 Windows Server 2008 上执行的话应该可以应付大于 1500 个同步客户的请求了。

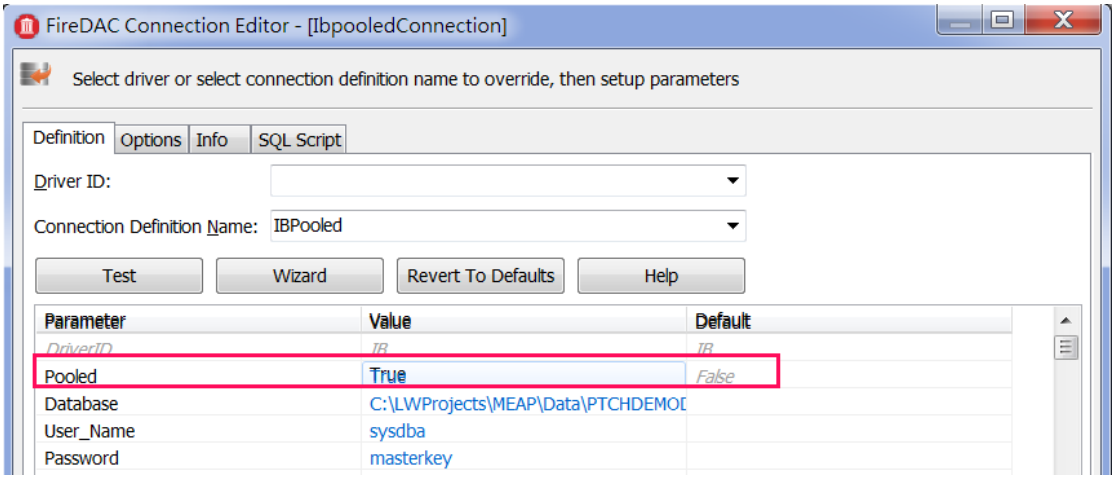
10-4 改善程序代码调校效能

完成了上述的调校步骤之后我们仍然有方法可以简单的再次增加 DataSnap 服务器的执行效率。在前面的调校步骤中我们增加了线程池的功能，降低了 Session 的负荷，使用 RESTful Broker 增加执行效率，接下来我们可以再开启数据库的连接池功能让 DataSnap 服务器更有效率的存取数据库。

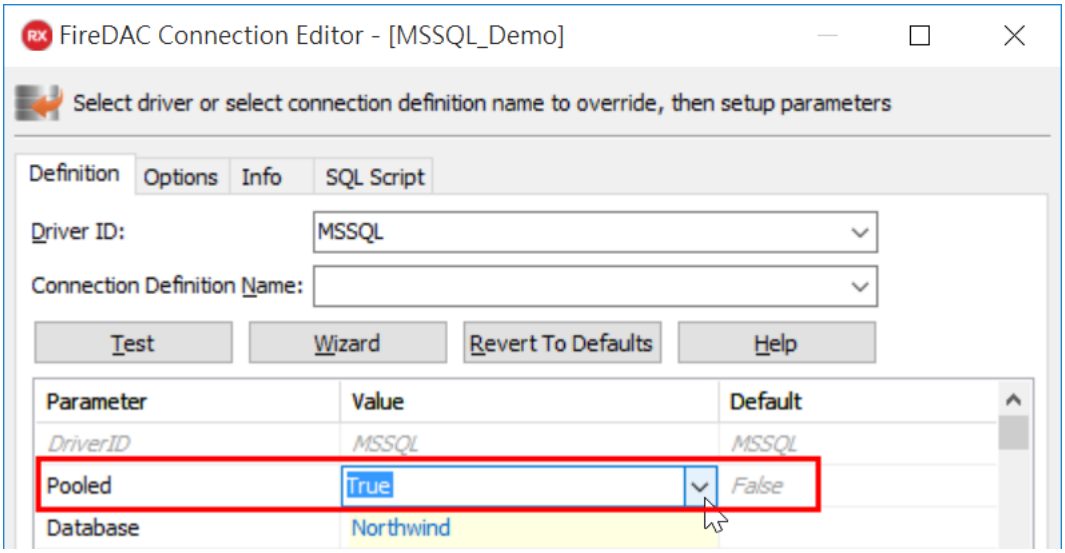
10-4-1 开启数据库链接池

在 FireDAC 中要开启数据库链接池功能非常的简单，只要启动 TFDConnection 的组件编译程序就可以如下图看其中包含了一个”Pooled”选项，在内定上 FireDAC 会关闭数据库链接池功能，因此下图中的”Pooled”选项内定值是 False，但读者可以把它设定为 True，以开启数据库链接池功能。

例如下图是笔者使用的 InterBase XE7 版在 FireDAC 中开启数据库链接池功能：



同样对于 MS SQL Serevr 2012 也可以在 FireDAC 中开启数据库链接池功能：



一旦设定好了开启数据库链接池功能之后，我们就可以在 FireDAC 的 FDConnectionDefs.ini 组态档中看到 Pooled=True 设定：

```
[MSSQL_Demo]
Server=DESKTOP-XXX\SQLEXPRESS
Database=Northwind
MetaDefSchema=dbo
MetaDefCatalog=Northwind
ExtendedMetadata=True
Pooled=True
```

```
DriverID=MSSQL
```

读者在部署 DataSnap 服务器时一定也要部署此开启数据库链接池功能的组态文件，否则在 DataSnap 服务器实际执行时仍然不会使用数据库链接池功能。

一旦使用料库链接池功能之后从下面的测试程序结果可知开启料库链接池功能比不开启料库链接池功能快了 8 倍的执行速度：

一个简单的设定就可以增加支持的客户端数目又可大幅增加执行效率，何乐而不为呢？

10-4-2 结合 ArrayDML

到了现在我们又加入了数据库链接池功能，那么还可以让 DataSnap 服务器更快吗？当然，还有架构设计和程序代码风格，但那些不在本书讨论内容之后，不过在结束本章内容之前让笔者再分享一个小秘诀，那就是结合 FireDAC 超快速的 ArrayDML 功能。

FireDAC 的 ArrayDML 功能可以数倍的速度进行大量数据的新增数据的工作，而且由于 ArrayDML 的原理是在客户端维持一个内存新增表格并一次把所有数据利用后端数据库 Bulk Insert 的功能一次把所有的数据写入数据库中，因此和 DataSnap 的多层架构非常的契合。当结合这 2 者的功能时效果异常的良好快速。

让我们使用一个简单的范例来说明。

DLL 型态的 DataSnap 服务器改良版 5-结合 ArrayDML

首先我们可以在 DataSnap 服务器中定义 2 个方法，一个是 GetData()取得数据，一个是 BulkInsert()方法可以藉由数据库的 Bulk Insert 功能一次把所有数据写入数据库：

```
  function GetData : TStream;
```

```
  function BulkInsert(dataStream : TStream) : Boolean;
```

GetData()方法把后端资料以 TStream 格式送到客户端显示：

```
function TServerMethods1.GetData: TStream;
begin
  Result := TMemoryStream.Create;
  try
    TbltestdataTable.Close;
    TbltestdataTable.Open;
    TbltestdataTable.SaveToStream(Result, TFDStorageFormat.sfBinary);
    Result.Position := 0;
  except
    raise;
  end;
end;
```

BulkInsert()方法使用了数个巧妙的技巧，首先 007~008 行巧妙的把 FireDAC 的 Stream 格式回转到 TMemoryStream 对象中，接着 010 行就能把数据还完成 FireDAC 的 DataSet，之后的程序代码就可以使用一般的 ArrayDML 功能把数据写入数据库中(请参考本系列"FireDAC 数据库程序设计"一书)：


```
001   function TServerMethods1.BulkInsert(dataStream: TStream): Boolean;
002   var
003     iIndex: Integer;
004     LMemStream : TMemoryStream;
005   begin
006     Result := True;
007     LMemStream := CopyStream(dataStream);
008     LMemStream.Position := 0;
009     try
010       fdqBulkInsert.LoadFromStream(LMemStream, TFDStorageFormat.sfBinary);
011
012       fdqryInsert.Params.ArraySize := fdqBulkInsert.RecordCount;
013       for iIndex := 0 to fdqBulkInsert.RecordCount - 1 do
014       begin
015         fdqryInsert.Params.ParamByName('NEW_ADDDT').AsDates[iIndex] :=
fdqBulkInsert.FieldByName('ADDDT').Value;
```

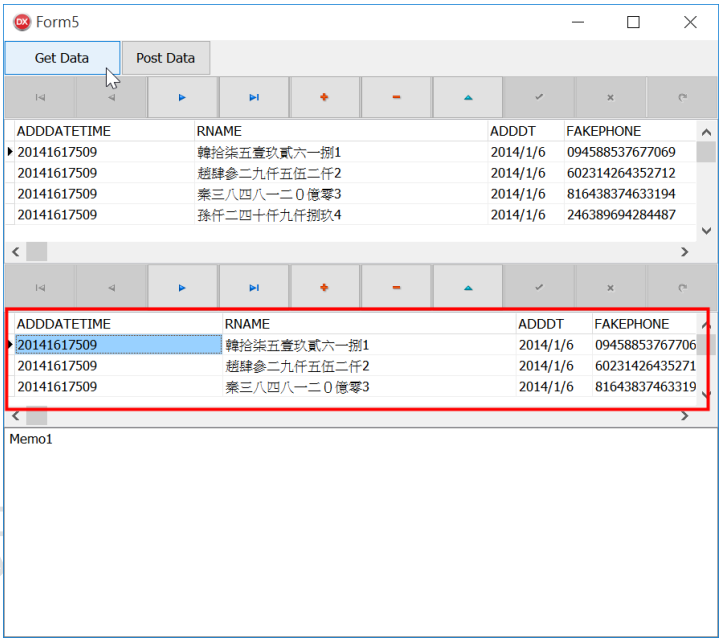
```
016
fdqryInsert.Params.ParamByName('NEW_ADDDATETIME').AsStrings[iIndex] :=
fdqBulkInsert.FieldByName('ADDDATETIME').Value;
017        fdqryInsert.Params.ParamByName('NEW_RNAME').AsStrings[iIndex] :=
fdqBulkInsert.FieldByName('RNAME').Value;
018        fdqryInsert.Params.ParamByName('NEW_FAKEPHONE').AsStrings[iIndex] :=
fdqBulkInsert.FieldByName('FAKEPHONE').Value;
019        fdqBulkInsert.Next;
020      end;
021      fdqryInsert.Execute(fdqryInsert.Params.ArraySize);
022    except
023      Result := False;
024      LMemStream.Free;
025    end;
026  end;
```

之后客户端就可以使用如下的程序代码一之把大量的数据新增到数据库中了：

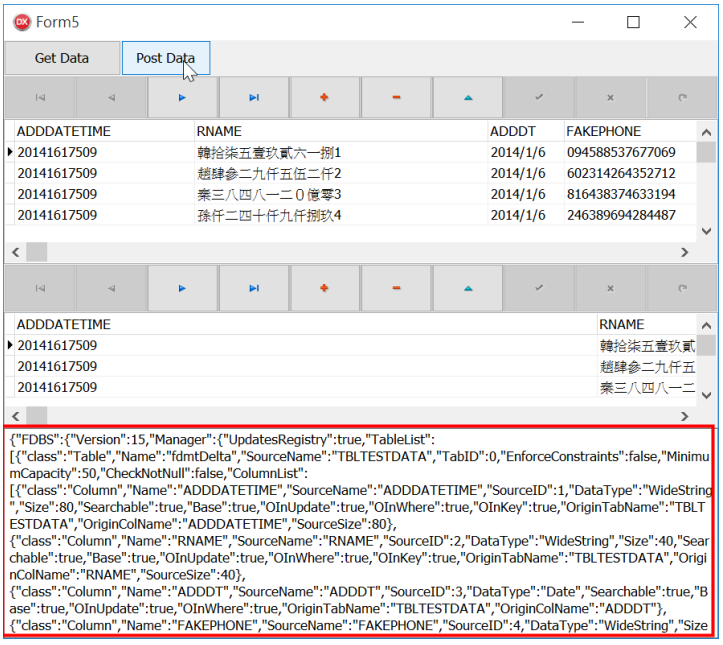

```
001  procedure TForm5.PostData;
002  var
003    ms : TMemoryStream;
004    dss : TStringStream;
005  begin
006    ms := TMemoryStream.Create;
007    dss := TStringStream.Create;
008    try
009      fdmtDelta.ResourceOptions.StoreItems := [siData,  siDelta, siMeta];
010      fdmtDelta.SaveToStream(ms, TFDStorageFormat.sfBinary);
011      fdmtDelta.SaveToStream(dss, TFDStorageFormat.sfJSON);
012      Memo1.Lines.Text := dss.DataString;
013      ms.Position := 0;
014      fdspBulkInsert.Params.ParamByName('dataStream').AsStream := ms;
015      fdspBulkInsert.ExecProc;
016
017      fdmtDelta.Close;
018      dss.Position := 0;
019      fdmtDelta.LoadFromStream(dss, TFDStorageFormat.sfJSON);
```

```
020     finally
021       ms.Free;
022       dss.Free;
023     end;
024   end;
```

下图是客户端呼叫 GetData()方法从 DataSnap 服务器取得数据:



下图是在客户端新增大量数据之后呼叫 DataSnap 服务器的 BulkInsert()方法一次把数据更新回数据库，速度超快：

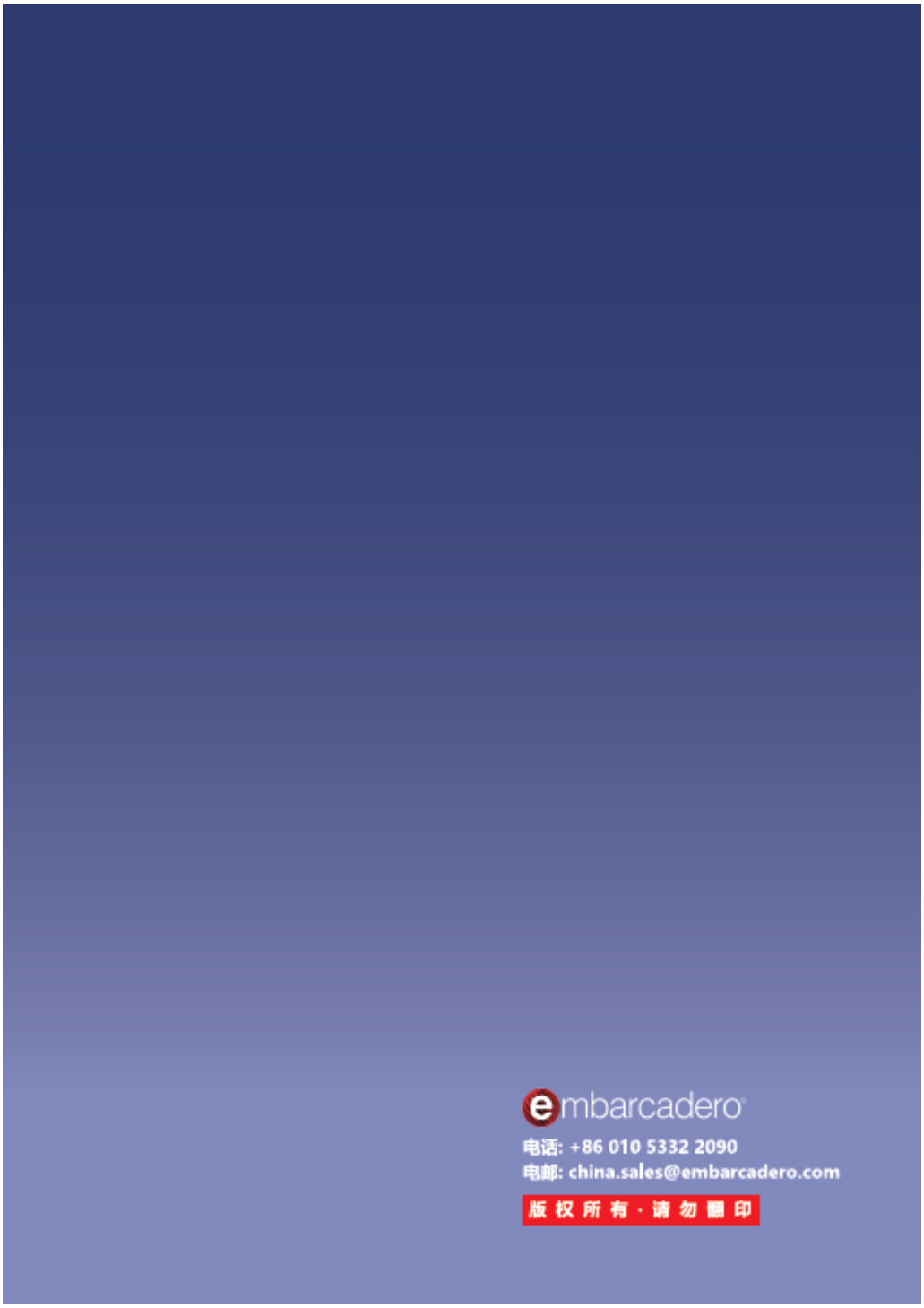


结合上面的调校技术和 ArrayDML 功能之后，在要处理大量新增资料的 DataSnap 应用中执行速度又提升了 3~5 倍的速度，真是令人印象深刻。

10-5 结论

本章讨论了各种技术让读者能够学习开发并调校 DataSnap 服务器的执行效率，是非常实用的内容。DataSnap 服务器在采用这些调校技术后的确可以使用在实际的应用中处理大量，复杂的应用。如果读者再搭配 DataSnap 的过滤器，安全功能和 HTTPS 等功能，那么仍果就一定是一个”安全，高效率的 DataSnap 应用系统 " 了。

Have Fun!


embarcadero
电话: +86 010 5332 2090
电邮: china.sales@embarcadero.com
版 权 所 有 · 请 勿 翻 印