Delphi FireDAC

数据库开发手册

embarcadero

序

FireDAC 应该算是 Delphi 从 Borland 时代开始的第 3 代数据存取技术和框架了，Delphi 历经了 BDE/IDAPI，dbExpress 一直到现在的 FireDAC，这也代表了 Delphi 从桌面开发，C/S，Web，Multi-Tier 转变到现在着重跨平台和移动开发的需求演进。

FireDAC 是第一个完全由 Delphi 程序语言撰写的数据库存取框架，以前的 BDE/IDAPI 和 dbExpress 是混合了 Object Pascal 和 C 语言撰写的数据库存取框架，因此随着 Delphi 程序语言支持多平台，FireDAC 也可以轻易的在多个平台中执行。但除了程序语言的原因之外，到底为什么要使用 FireDAC 来取代 BDE/IDAPI 和 dbExpress 呢？

最主要的原因就是这 BDE/IDAPI，dbExpress 和 FireDAC 设计的目标和架构，BDE/IDAPI 在近 20 年前设计的目标是让 Delphi 在桌面和稍后出现的 C/S 架构中使用，而 dbExpress 设计的目标则是让 Delphi 除了能够在原本的桌面和 C/S 架构中使用之外，也能够在 Web 和 Multi-Tier 架构中使用。但随着移动和穿戴式设备的出现和开发的需求，Delphi 也需要一个能够适用在所有平台的数据存取技术和框架，而 FireDAC 正好能够满足这个目标和需求。

FireDAC 的功能其实非常类似 BDE/IDAPI 和 dbExpress 的结合体，它在使用上非常接近 BDE/IDAPI，但又具备 dbExpress 的联机存取和脱机数据处理的能力，再加上 FireDAC 不需要部署额外 DLL 档案而能够直接连结客户端程序代码的特性以及精简型数据集的功能，让 FireDAC 也非常适合使用在移动和穿戴式设备的应用。因为如果您需要使用 Delphi 开发任合需要处理数据的应用程序，那么您绝对应该认真考虑使用 FireDAC。

本书的目的是希望让读者能够快速学习和使用 FireDAC 来开发 Delphi 的数据库应用程序，希望在您阅读完本书的内容之后就具备了足够的知识和技术善用 FireDAC 开发出跨平台的数据库应用程序。

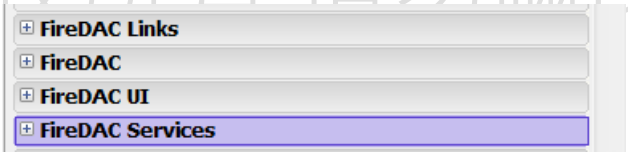
目录

FireDAC技术篇

第1章 开始学习使用FireDAC开发数据库应用程序吧

FireDAC 是非常易于使用的数据存取技术和框架，但又蕴含了强大的功能。但高楼平地起，让我们从如何使用 FireDAC 组件开发简单的数据库应用程序开始讨论起吧。

在 DX10 中 FireDAC 提供了 4 类组件组：



下面的表格简单的说明了这 4 类组件组的功能：

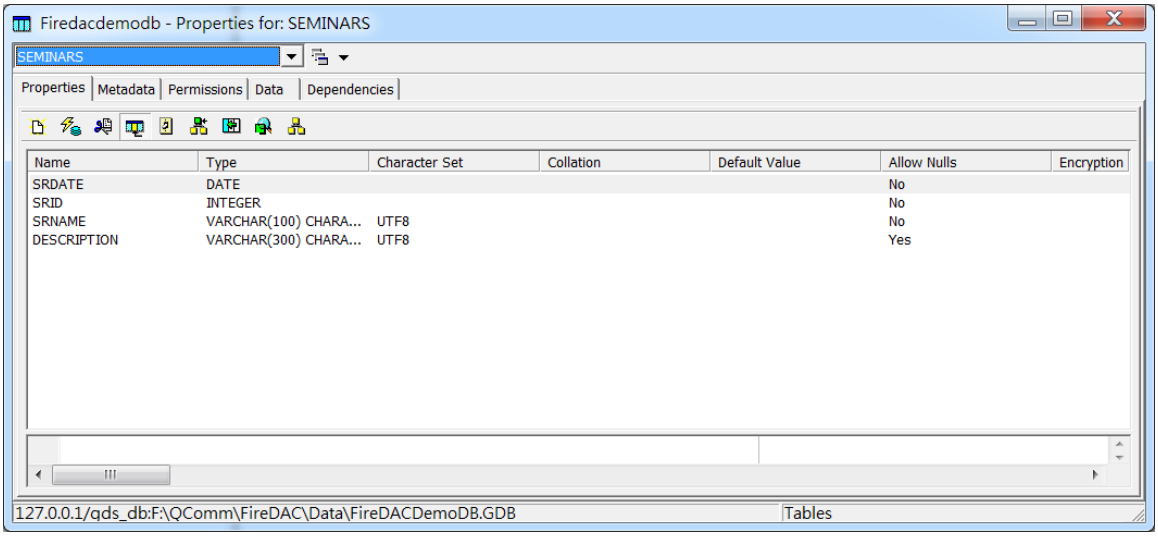
名称	说明
FireDAC Links	提供 FireDAC 呼叫各数据库客户端函式库的组件，开发人员只需要使用这些组件即可链接各数据库而不再需要部署额外的 FireDAC DLL(但仍需要部署各数据库的客户端函式库)。此外本类组也包含了可监督联机状况的组件。
FireDAC	FireDAC 的核心组件，使用来链接和处理数据的组件
FireDAC UI	提供 FireDAC 处理数据时的相关 UI 组件
FireDAC Services	提供 FireDAC 额外的服务组件，主要是提供 SQLite 和 InterBase 的相关服务

学习 FireDAC 组件框架的第一步当然就是使用它链接数据库和处理数据了。本书使用的数据库主要是 InterBase，除了因为 InterBase 是 Delphi 内附的数据库之外，主要

是因为 InterBase 可以使用在 Windows，iOS 和 Android 平台，这 3 个平台也是本书讨论的主要平台。

1-1 使用 FireDAC 链接数据库

本小节将使用 2 个 InterBase 范例数据表 Seminars 和 SeminarAttendee，因为这 2 个数据表之间有主从的关系，包含这 2 个数据表的范例数据库则是 Firedacdemodb.GDB。下面是 Seminars 数据表包含的字段信息：

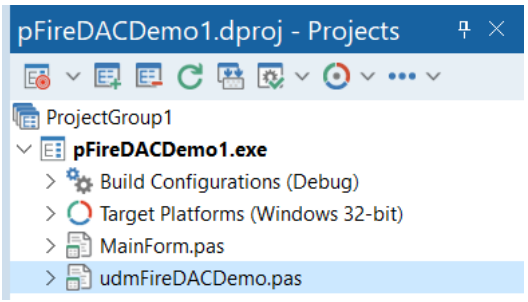


要使用 FireDAC 存取数据库非常的简单，基本上只需要使用 4 个 FireDAC 组件即可，下面的表格说明了本小节使用的 FireDAC 组件：

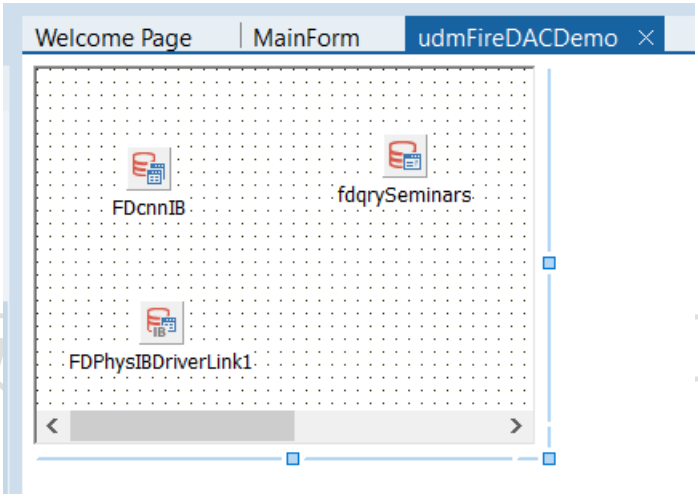
组件	名称	说明
	TFDConnection	使用来链接数据库的组件
	TFDQuery	使用来执行 SQL 命令的组件
	TFDPhysIBDriverLink	链接到 Interbase 的驱动程序组件
	TFDGUIxWaitCursor	控制 UI 等候光标的组件

先让我们说明如何使用这 4 个组件链接并处理 Seminars 数据表再深入讨论进阶的用法。

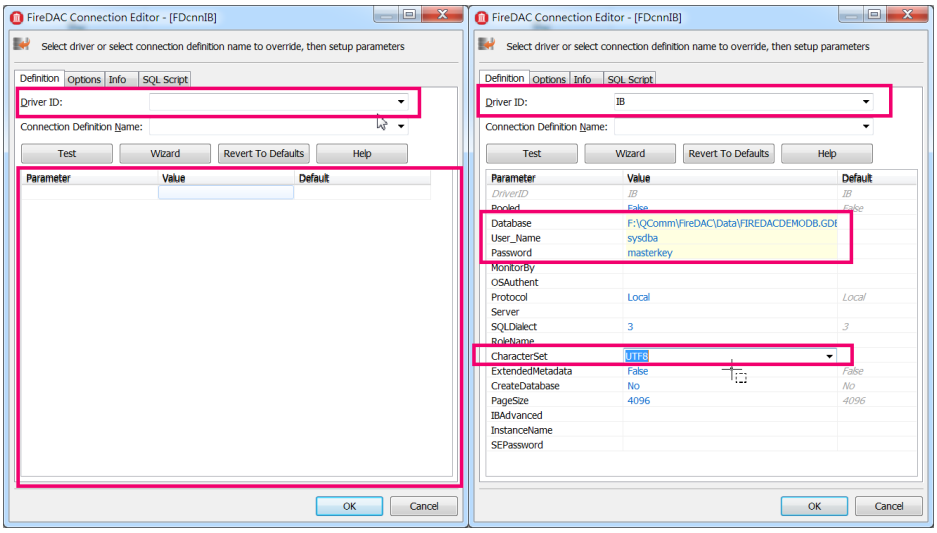
在 Delphi IDE 中建立一个 FireMonkey Desktop Application 项目并且在其中建立一个数据模块，主窗体以 MainForm 为名称储存，数据模块以 udmFireDACDemo 为名称储存，此时项目管理员如下：



在数据模块中放入 TFDConnection，TFDQuery 和 TFDPhysIBDriverLink 组件，设定 TFDConnection 的 Name 特性值为 FDcnnIB，TFDQuery 的 Name 特性值为 fdqrySeminars：

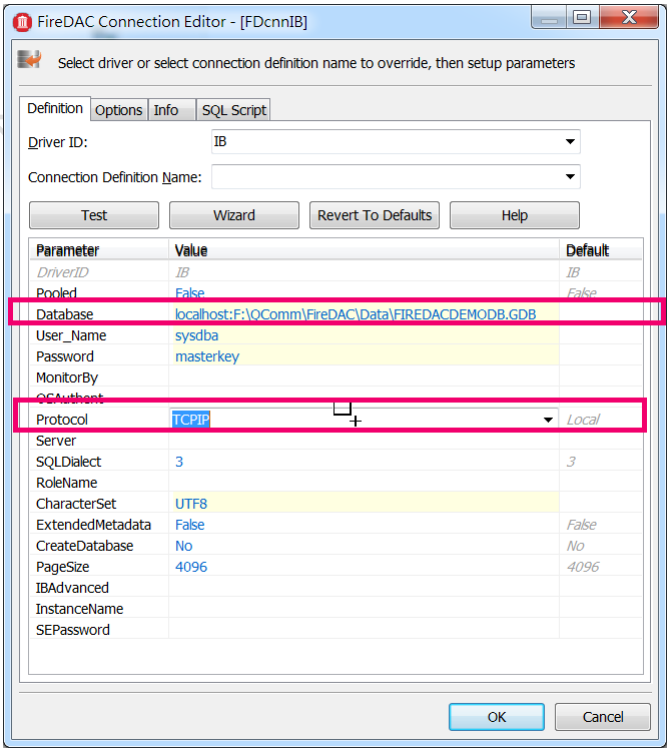


现在让我们先使用 FDcnnIB 链接范例数据库，请使用鼠标双击 FDcnnIB 就会启动 TFDConnection 的组件编译程序，此时组件编译程序如下左图所示内容大都是空白的。由于我们要连结到 InterBase，因此请先在组件编译程序上方的 Driver ID 字段中选择 IB 代表要连结到 InterBase，一旦选择之后组件编译程序就会出现链接 InterBase 需要设定的选项，例如 InterBase 数据库所在位置，登入 InterBase 的用户名称和密码，使用的链接通讯协议以及使用的字符集等。例如下面右图显示了笔者链接到本机的 InterBase 数据库 F:\QComm\FireDAC\Data\FIREDACDEMODB.GDB，因此下面的 Protocol 字段选择了 Local：



当然如果笔者要连结到服务器中的 InterBase 数据库的话就可以如下选择连结通讯协议为 TCPIP 并且在 InterBase 数据库名称之前加上服务器位置，例如

```
127.0.0.1:F:\QComm\FireDAC\Data\FIREDACDEMODB.GDB
```



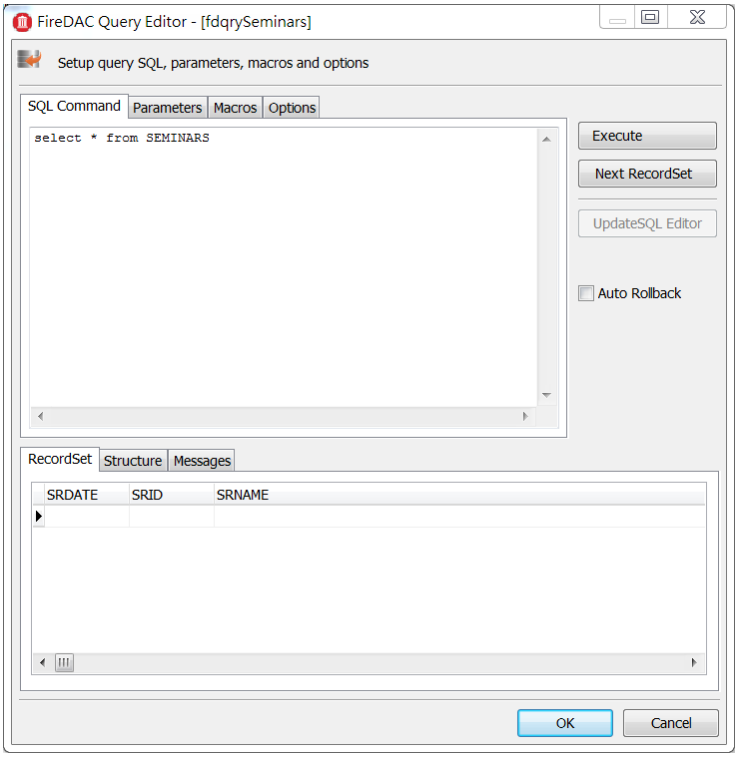
最后再设定 FDcnnIB 组件的 LoginPrompt 特性值为 False 免除每次开启数据库链接时都需要登入，现在就暂时完成了 FDcnnIB 的设定。

现在设定 fdqrySeminars 组件，请在对象查看器中检查 fdqrySeminars 的 Connection 特性值应该已经自动设定为 FDcnnIB 了，如果没有的话请设定为

FDcnnIB。使用鼠标双击 fdqrySeminars 组件启动它的组件编译程序就会看到如下的画面，在 SQL Command 页次中输入

```
Select * from SEMINARS
```

如果此时点选右方的『Execute』按钮就可以在下方看到目前在 SEMINARS 数据表中的资料了，由于目前我们尚未在其中加入任何的资料因此现在是没有数据的，点选 OK 按钮以储存此 SQL 命令到 fdqrySeminars 的 SQL 特性值中：



完成了 FDcnnIB 和 fdqrySeminars 的设定现在就可以在对象查看器中把 FDcnnIB 的 Connected 设定为 True，再设定 fdqrySeminars 的 Active 为 True，这样就可以链接 InterBase 范例数据表 Seminars 并且把其中的数据存取到范例程序了，或是在数据模块的 OnCreate 事件处理及式中撰写如下的程序代码：

```
procedure TdmFireDACDemo1.DataModuleCreate(Sender: TObject);
begin
  FDcnnIB.Connected := True;
  fdqrySeminars.Active := True;
end;
```

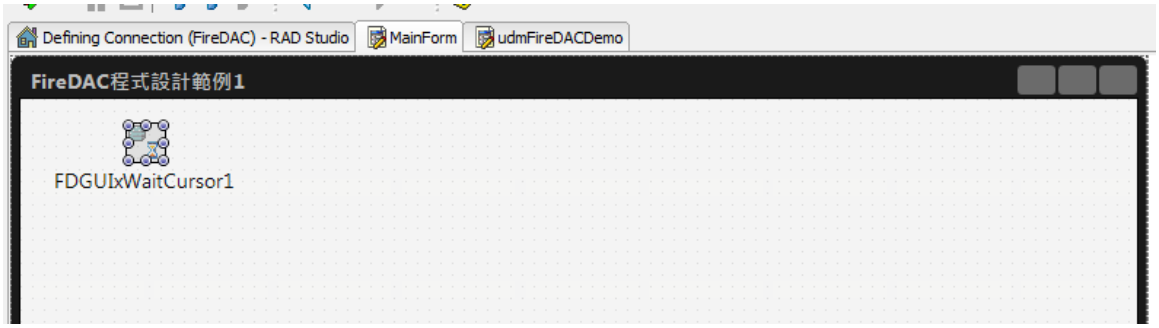
最后一定要在数据模块的 OnDestroy 事件处理及式中把 FDcnnIB 关闭：

```
procedure TdmFireDACDemo1.DataModuleDestroy(Sender: TObject);
begin
```

```
  FDcnnIB.Connected := False;
end;
```

现在数据既然已经存取到客户端的范例程序中了，我们自然可以开始把数据和 UI 链接起来了。

回到主窗体中加入 TFDGUIxWaitCursor 组件，现在就可以使用这 4 个 FireDAC 组件来处理数据了。



要对 SEMINARS 数据表进行新增，修改，删除和简单数据查询的工作只需要呼叫 TFDQuery 的 Insert，Delete，Edit 和 Locate 方法即可。例如要在 SEMINARS 数据表加入一笔新的数据，只需要使用如下的程序代码：


```
fdqrySeminars.Insert;
fdqrySeminars.FieldByName('SRNAME').Value := '测试';
fdqrySeminars.FieldByName('SRDATE').Value := Now;
...
fdqrySeminars.Post;
```

在呼叫了 Insert 之后再呼叫 TFDQuery 的 FieldByName 方法藉由域名一一的把每一个域值填入最后再呼叫 Post 方法即可把新数据立刻加入到 SEMINARS 数据表中。

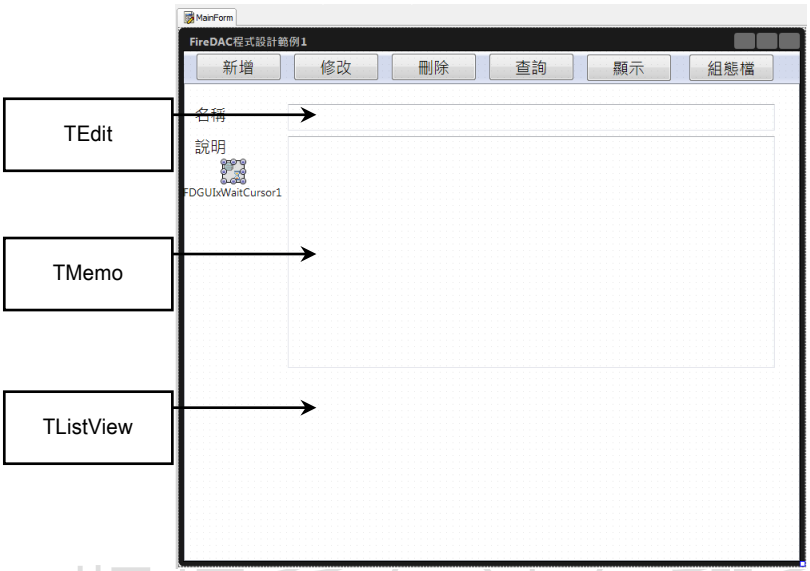
如果要修改数据的话需要先找到您要修改的数据记录再呼叫 Edit 方法进入修改模式，然后也是藉由呼叫 FieldByName 方法把要修改的域值填入新的数值，最后也是再呼 Post 方法。

但要如何搜寻修要的数据呢？这可以藉由 Locate 方法，Locate 方法可以藉由搜寻特定的域值来寻找数据。例如如果我们要搜寻特定的研讨会资料，那么可以使用：

```
if (fdqrySeminars.Locate('SRNAME', '要寻找的研讨会名称', [])) then
begin
  ...
end;
```

Locate 方法会搜寻符合条件的第一笔数据，如果找到的话就回传 True 没找到就回传 False，因此判断 Locate 方法的回传值就可以知道是否找到了需要搜寻的资料。

让我们使用一个简单的范例看看如何真的在 SEMINARS 数据表中对数据进行新增，修改，删除和数据的工作。首先在 Delphi IDE 中建立一个 FireMonkey Desktop Application 项目，在主窗体中加入如下的 FireMonkey 组件：



主窗体上的按钮意思应该很清楚了，让我们从如何显示数据开始说明。

当我们设定 TFDQuery 组件的 Active 特性值为 True 或是呼叫它的 Open 方法后 TFDQuery 组件就会执行它的 SQL 特性值中 SQL 命令从数据表中取回的数据就称为数据集(DataSet)，之后我们就可以呼叫下面的方法在此数据集中的资料进行移动和浏览的动作：

方法	说明
First	到数据集中的第一笔资料
Last	到数据集中的最后一笔数据
Next	到目前下一笔资料
Prior	到目前上一笔资料

在移动和浏览数据时又有下面 2 个特性可判断数据集是否在最开始的位置和最后的位置：

特性	说明
Bof	目前在数据集起始的位置
Eof	目前在数据集结尾的位置

因此如果要在数据集中移动到每一笔数据的位置可以使用下面的程序代码样板(以 fdqrySeminars 为范例)：

```
fdqrySeminars.First;
while (not fdqrySeminars.Eof) do
begin
  ....
  fdqrySeminars.Next
end;
```

当然您也可以反相移动：

```
fdqrySeminars.Last;
while (not fdqrySeminars.Bof) do
begin
  ....
  fdqrySeminars.Prior
end;
```

这正是主窗体中『显示』按钮使用来显示 SEMINARS 数据表中所有数据的方法，当点选『显示』按钮后它呼叫 DisplayAllSeminars 方法显示所有数据。DisplayAllSeminars 方法先在 012 行呼叫 TFDQuery 组件的 GetBookmark 方法把目前数据的位置暂存下来，再于 014~021 行使用前面介绍的方法把 SEMINARS 数据表的 SRNAME 和 SRDATE 字段显示在 TListView 组件中，023~024 在显示完数据后再回到原先数据的位置。


```
001   procedure TfmMain.btnShowDataClick(Sender: TObject);
002   begin
003     DisplayAllSeminars;
004   end;
005
006   procedure TfmMain.DisplayAllSeminars;
007   var
008     bk : TBookMark;
009     lvi : TListViewItem;
010   begin
011     ListView1.Items.Clear;
012     bk := dmFireDACDemo1.fdqrySeminars.GetBookmark;
013     try
```

```
014       dmFireDACDemo1.fdqrySeminars.First;
015       while (not dmFireDACDemo1.fdqrySeminars.Eof) do
016       begin
017         lvi := ListView1.items.Add;
018         lvi.Text :=
dmFireDACDemo1.fdqrySeminars.FieldByName('SRNAME').Value;
019         lvi.Detail :=
dmFireDACDemo1.fdqrySeminars.FieldByName('SRDATE').AsString;
020         dmFireDACDemo1.fdqrySeminars.Next;
021       end;
022     finally
023       dmFireDACDemo1.fdqrySeminars.GotoBookmark(bk);
024       dmFireDACDemo1.fdqrySeminars.FreeBookmark(bk);
025     end;
026   end;
```

『新增』按钮使用下面的程序代码在 SEMINARS 数据表中加入一笔数据：

```
procedure TfmMain.btnInsertClick(Sender: TObject);
begin
  dmFireDACDemo1.fdqrySeminars.Insert;
  dmFireDACDemo1.fdqrySeminars.FieldByName('SRDATE').Value :=
Now;
  dmFireDACDemo1.fdqrySeminars.FieldByName('SRNAME').Value :=
edtSeminar.Text;
  dmFireDACDemo1.fdqrySeminars.FieldByName('SRID').Value :=
GetSRIDFromDateTime(Now);

dmFireDACDemo1.fdqrySeminars.FieldByName('DESCRIPTION').Value :=
mmSeminarDescription.Lines.Text;
  dmFireDACDemo1.fdqrySeminars.Post;

  edtSeminar.Text := String.Empty;
  mmSeminarDescription.Lines.Text := String.Empty;
  DisplayAllSeminars;
end;
```

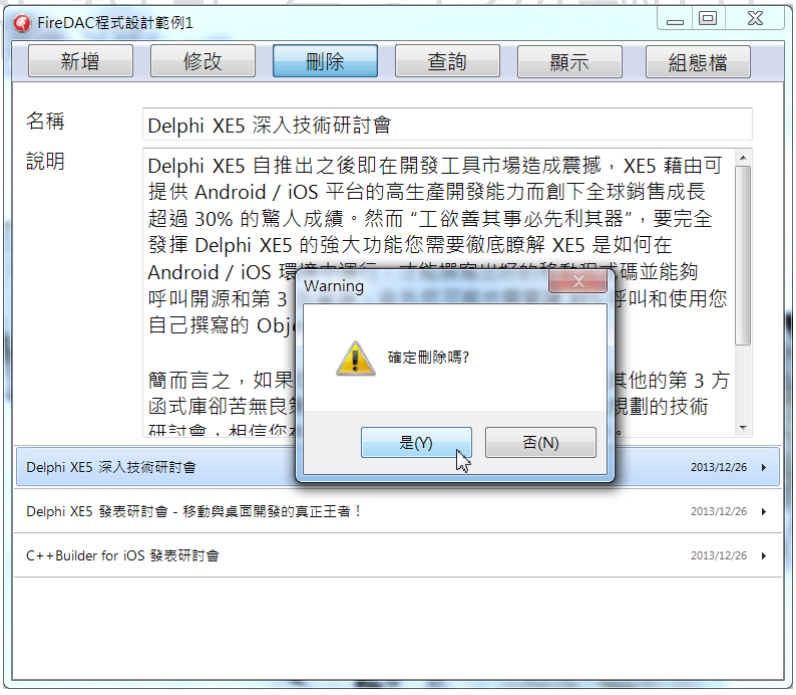

『修改』按钮先呼叫 SearchSeminar 方法使用 Locate 找到数据再呼叫 Edit 进入修改模式最后再呼叫 Post 把数据更新回数据表：

```
001   procedure TfmMain.btnModifyClick(Sender: TObject);
002   begin
003     if (SearchSeminar(edtSeminar.Text)) then
004     begin
005       dmFireDACDemo1.fdqrySeminars.Edit;
006
dmFireDACDemo1.fdqrySeminars.FieldByName('SRNAME').Value :=
edtSeminar.Text;
007
dmFireDACDemo1.fdqrySeminars.FieldByName('DESCRIPTION').Value :=
mmSeminarDescription.Lines.Text;
008     end;
009   end;
010
011   function TfmMain.SearchSeminar(const sSeminar: String):
Boolean;
012   begin
013     Result := dmFireDACDemo1.fdqrySeminars.Locate('SRNAME',
sSeminar, [loCaseInsensitive, loPartialKey]);
014   end;
```


最后的『删除』按钮也是使用 SearchSeminar 方法先找到要删除的数据再呼叫 TFDQuery 组件的 Delet 方法从数据表中删除数据：

```
001   procedure TfmMain.btnDeleteClick(Sender: TObject);
002   var
003     iResult : Integer;
004   begin
005     if (SearchSeminar(edtSeminar.Text)) then
006     begin
007       iResult := MessageDlg('确定删除吗?',
TMsgDlgType.mtWarning, [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo], 0);
008       if (iResult = mrYes) then
009         dmFireDACDemo1.fdqrySeminars.Delete;
010     end;
011     DisplayAllSeminars;
012     ListView1ItemClick(ListView1, ListView1.Items[0]);
013   end;
```

执行范例程序就可以看到下面的结果，的确可以在 EMINARS 数据表进行 CRUD 的工作了。



1-1-1 链接数据库的方式

在前面说明如何链接到 InterBase 数据库时我们时直接使用 TFDConnection 的组件编辑器来设定和 InterBase 的连结，但如果每个程序都这样做就显得麻烦。因此

FireDAC 另外提供了 2 种方式适合使用在多人环境中，例如主从架构环境，Web 或是多层架构中。第一种是使用组态档方式，这个方式也类似以前 BDE/IDAPI 和 dbExpress 的组态档方式。这个概念是把链接数据库的信息先储存到一个组态文件中，并且和程序一起分发这个组态档或是把这个组态文件放在网络之中，那么当 FireDAC 程序执行时先读入这个组态档，再依照组态文件的内容链接数据库。

因此对照前面的范例，当开发人员在 IDE 中成功设定了数据库链接之后可以把这个数据库链接写成组态档，再把这个组态档做为公共使用的组态档即可。了解了组态档概念之后就让我们来说明如何使用它。

使用组态档

组态档就是一个文本文件，在您的 c:\Users\Public\Documents\RAD Studio\FireDAC\目录下有一个范例 FireDAC 组态文件 FDConnectionDefs.ini，您可以依照其中的格式来撰写或是修改成您需要的连结内容，例如其中就包含了链接到 InterBase 范例数据库 Employee 的链接内容：

```
[EMPLOYEE]
DriverID=IB
Protocol=TCPIP
Database=localhost:C:\ProgramData\Embarcadero\InterBase\gds_db\examples\database\employee.gdb
User_Name=sysdba
Password=masterkey
CharacterSet=
ExtendedMetadata=True
```


FDConnectionDefs.ini 是 FireDAC 内定使用的组态文件名称，当 FireDAC 程序执行时定会在目前的执行目录中寻找 FDConnectionDefs.ini，如果找不到的话就会在 Windows 系统注册的

```
HKCU\Software\Embarcadero\FireDAC\ConnectionDefFile
```

中寻找 FDConnectionDefs.ini，而

```
HKCU\Software\Embarcadero\FireDAC\ConnectionDefFile
```

的预定值是

```
C:\Users\Public\Documents\RAD Studio\FireDAC\FDConnectionDefs.ini.
```

当然使用 Windows 系统注册来寻找组态档并不是好方法，因为这就限定了您的 FireDAC 程序只能在 Windows 中执行，因此把组态档放在目前的执行目录中或是放在网络中特定的位置再让 FireDAC 程序去寻找使用比较好。

此外把程序员在 IDE 中设定的连结写成组态档来使用也不错。现在就让我们来说明如何建立组态档以及如何使用组态档。

在前面的范例主窗体中有一个『组态文件』按钮，当您点选它之后它就会把 TFDConnection 组件链接 InterBase 的设定写入一个组态档以便分发给其他程序使用。

在 FireDAC 中要建立和使用组态档都需要使用 FireDAC 的 TFDManager 组件。TFDManager 的 ConnectionDefFileName 特性可以让您指定组态档，而 ConnectionDefs 特性则可以让您建立链接信息。ConnectionDefs 特性是 IFDStanConnectionDefs 接口，其中的 AddConnectionDef 方法可以建立一个新的 FireDAC 链接信息对象，在您设定了链接信息对象的特性值之后呼叫它的 MarkPersistent 和 Apply 方法就可以储存一个 FireDAC 链接信息和组态文件了。

例如下面就是『组态文件』按钮的实作程序代码，006 行设定组态文件位置和名称，008 行先建立 IFDStanConnectionDef 接口对象 009 行设定这个组态名称，010~015 设定链接信息，016~017 储存组态档：


```
001   procedure TdmFireDACDemo1.WriteMyConnectionDef;
002   var
003     MyFireDACDef: IFDStanConnectionDef;
004   begin
005     if (not FDManager.ConnectionDefFileLoaded) then
006       FDManager.ConnectionDefFileName := MYCONNECTIONFILE;
007
008     MyFireDACDef :=
FDManager.ConnectionDefs.AddConnectionDef;
009     MyFireDACDef.Name := MYCONNECTIONNAME;
010     MyFireDACDef.DriverID := FDcnnIB.DriverName;
011     MyFireDACDef.Database :=
FDcnnIB.Params.Values['Database'];
012     MyFireDACDef.UserName :=
FDcnnIB.Params.Values['User_Name'];
013     MyFireDACDef.Password :=
FDcnnIB.Params.Values['Password'];
014     MyFireDACDef.Params.Values['Protocol'] :=
```

```
FDcnnIB.Params.Values['Protocol'];
015     MyFireDACDef.Params.Values['CharacterSet'] :=
FDcnnIB.Params.Values['CharacterSet'];
016     MyFireDACDef.MarkPersistent;
017     MyFireDACDef.Apply;
018   end;
```

注意，由于每个在链接每种数据库时不同的数据库可能需要不同的设定，因此IFDStanConnectionDef接口只定义了大多数数据库都会使用的设定，例如：

特性	说明
UserName	数据库登录名称
Password	数据库登录密码
OSAuthent	是否由OS认证
Server	数据库服务器位置
Port	数据库服务器使用的通讯端口
DriverID	数据库驱动程序ID
Pooled	是否使用数据库链接池

对于没有列在 IFDStanConnectionDef 接口的数据库设定，例如 InterBase 的 Protocol 和 CharacterSet 等，我们都可以藉由使用 IFDStanConnectionDef 的 Params 特性加入特定数据库需要进行的特定设定。

下面是 MYCONNECTIONFILE 和 MYCONNECTIONNAME 的定义：

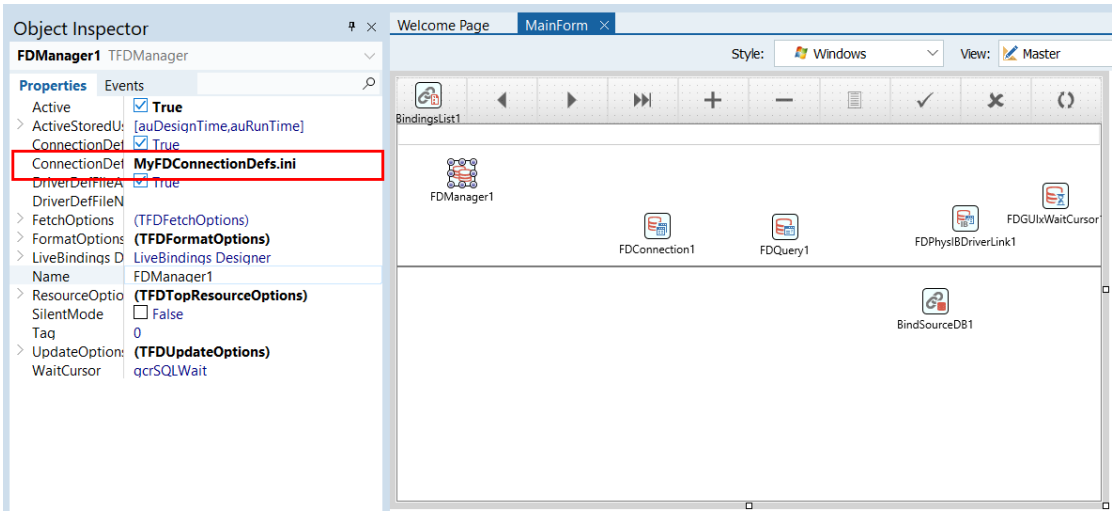
```
const MYCONNECTIONFILE = '.\MyFDConnectionDefs.ini';
const MYCONNECTIONNAME = 'MyIB_Connection';
```

执行上面的程序代码就可以在目前目录下看到产生了 MyFDConnectionDefs.ini 组态档，其中包含了下面的连结内容：

```
[MyIB_Connection]
DriverID=IB
Database=127.0.0.1:F:\QComm\FireDAC\Data\FIREDACDEMODB.GDB
User_Name=sysdba
Password=masterkey
Protocol=TCPIP
CharacterSet=UTF8
```

现在建立一个新的 FireMonkey Desktop Application 项目，在主窗体中加入如下的组件，请注意在其中使用了 TFDManager 组件，并且在对象查看器中于它的

ConnectionDefFileName 特性中输入 MyFDConnectionDefs.ini，代表要使用执行目录下 MyFDConnectionDefs.ini 这个组态文件中的链接信息来链接数据库：



接着在主窗体的 OnShow 事件处理函式中开启 TFDManager，TFDConnection 和 TFDQuery 组件，并且在 TFDConnection 的 BeforeConnect 事件处理函式中设定它要使用组态档中 MYCONNECTIONNAME 类别的链接信息：


```
001    const MYCONNECTIONFILE = '.\MyFDConnectionDefs.ini';
002    const MYCONNECTIONNAME = 'MyIB_Connection';
003
004    procedure TForm7.FDConnection1BeforeConnect(Sender:
TObject);
005    begin
006      FDConnection1.ConnectionDefName := MYCONNECTIONNAME;
007    end;
008
009    procedure TForm7.FormDestroy(Sender: TObject);
010    begin
011      FDConnection1.Connected := False;
012      FDManager1.Active := False;
013    end;
014
015    procedure TForm7.FormShow(Sender: TObject);
016    begin
017      FDManager1.Active := True;
018      FDConnection1.Connected := True;
```

```
019      FDQuery1.Active := True;
020    end;
```

现在如果执行此范例程序就可以看到如下的执行画面，FireDAC 组件果然使用组态文件连结到 InterBase 了。

当然要记得把 MyFDConnectionDefs.ini 组态文件和范例程序放在同一个目录中。

1-1-2 直接使用程序代码

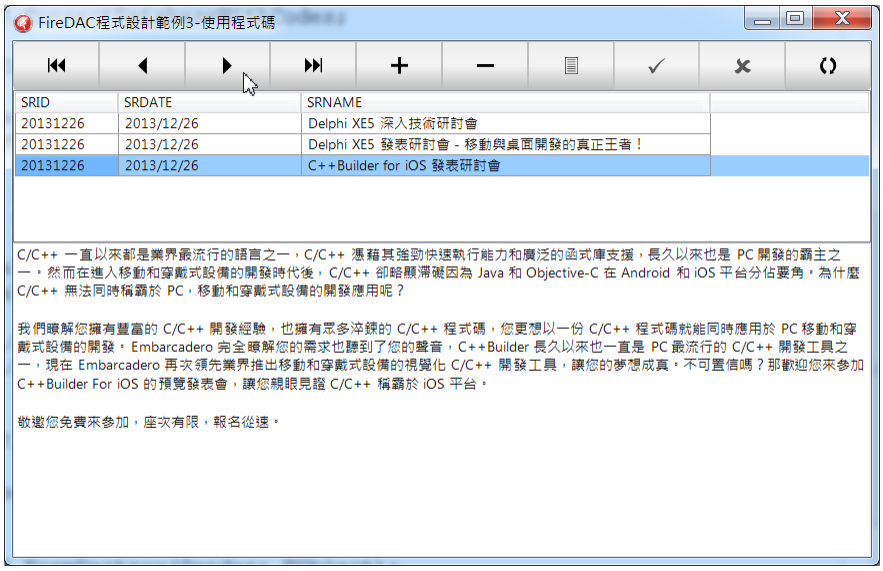
第 2 种方式是直接把如何链接数据库的方式用程序代码直接写在程序中，不过如此一来如果数据库设定改善的话就要改程序代码，这在多人使用的环境中并不实际，不过这种方式适合使用在移动平台中，因为移动平台都有沙盒的概念，数据库部署和链接方式都是固定的，因此使用这种方式很适合。在稍后我们就可以看到使用 FireDAC 开发移动平台数据库 App 的范例，不过在这里先让我们说明如何使用这种方式链接数据库。

本书的第 3 个范例使用了程序代码的方式链接数据库，在主窗体的 OnCreate 事件处理函式中呼叫了 ConnectDatabaseWithCodes 链接 InterBase 数据库，ConnectDatabaseWithCodes 方法也是先在 007 行藉由 TFDManager 建立一个 IFDStanConnectionDef 接口对象，在 008~014 行设定链接数据库的特性值，015 行呼叫 IFDStanConnectionDef 接口的 Apply 方法把这些设定写入到 TFDManager 中，最后 016 行设定 TFDConnection 组件使用 008 行设定的连结名称之后再于 017 行链接数据库：

```
001    const MYCONNECTIONNAME = 'MyIB_Connection';
002
003    procedure TForm7.ConnectDatabaseWithCodes;
004    var
```

```
005     ibDef: IFDStanConnectionDef;
006   begin
007     ibDef := FDManager1.ConnectionDefs.AddConnectionDef;
008     ibDef.Name := MYCONNECTIONNAME;
009     ibDef.Params.DriverID := 'IB';
010     ibDef.Params.Database :=
'127.0.0.1:F:\QComm\FireDAC\Data\FIREDACDEMODB.GDB';
011     ibDef.Params.UserName := 'sysdba';
012     ibDef.Params.Password := 'masterkey';
013     ibDef.Params.Values['Protocol'] := 'TCPIP';
014     ibDef.Params.Values['CharacterSet'] := 'UTF8';
015     ibDef.Apply;
016     FDConnection1.ConnectionDefName := MYCONNECTIONNAME;
017     FDConnection1.Connected := True;
018   end;
019
020   procedure TForm7.FormCreate(Sender: TObject);
021   begin
022     ConnectDatabaseWithCodes;
023   end;
024
025   procedure TForm7.FormDestroy(Sender: TObject);
026   begin
027     FDConnection1.Connected := False;
028   end;
029
030   procedure TForm7.FormShow(Sender: TObject);
031   begin
032     FDConnection1.Connected := True;
033     FDQuery1.Active := True;
034   end;
```


现在执行这个范例就可以看到如下的执行画面，证明使用程序代码也可以直接链接数据库：



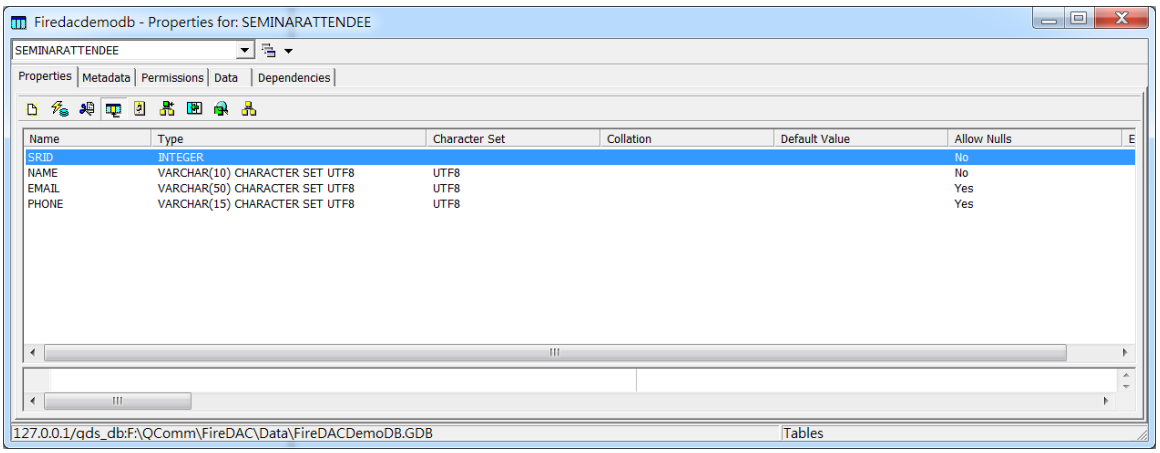
如同前面说明的，使用程序代码链接数据库特别适合使用在移动平台，稍后就会看到使用的范例和说明。

1-2 处理数据

在前一小节已经说明了如何使用 FireDAC 进行单数据表的 CRUD 的工作，在这一小节让我们再多讨论一点使用 FireDAC 处理数据的方式，首先先说明如何处理关连 2 个以上的数据表。

1-2-1 主从关连资料

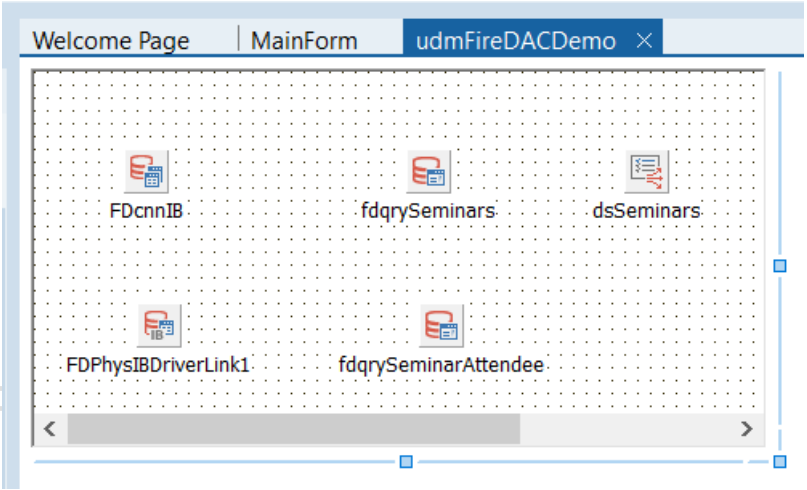
为了本小节说明之用，让我们加入第 2 个数据表 SeminarAttendee，它和前面的 Seminar 数据表拥有主从的关系，这 2 个数据表是藉由 SRID 字段关连在一起：



FireDAC 提供了非常方便的方法让开发人员处理主从关连资料，比 BDE/dbExpress 都方便，而且 FireDAC 提供了数种方法处理主从关连资料。

1-2-1-1 使用客户端范围机制

第一种方式非法的简单，其原理是主数据表和从数据表的数据从数据库中取回客户端，再于客户端藉由 FireDAC 的分段范围机制当主数据表中的数据移动位置时再筛选从数据表中符合的数据，现在让我们用一个范例来说明，先建立一个 FireMonkey Desktop Application 项目，在其中建立一个数据模块于其中使用下列的 FireDAC 组件从范例 Seminars 数据中存取数据：



在上面的数据模块中 fdqrySeminars 在它的 SQL 特性中使用了

```
select * from SEMINARS
```

从 Seminars 数据表中撷取数据，而 fdqrySeminarAttendee 在它的 SQL 特性中使用了

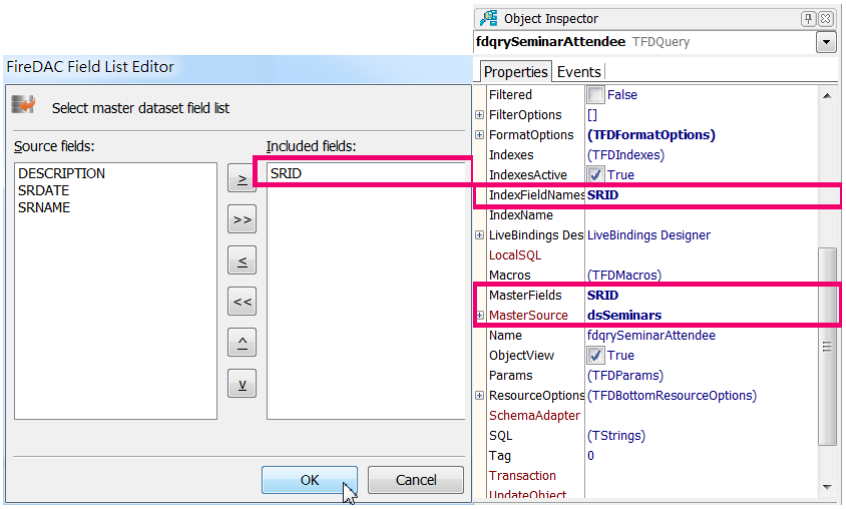
```
select * from SEMINARATTENDEE
```

从 SeminarAttendee 数据表中撷取数据，fdqrySeminars 和 fdqrySeminarAttendee 之间的主从关系是藉由数据模块中的 dsSeminars 这个 TDataSource 组件系结的。

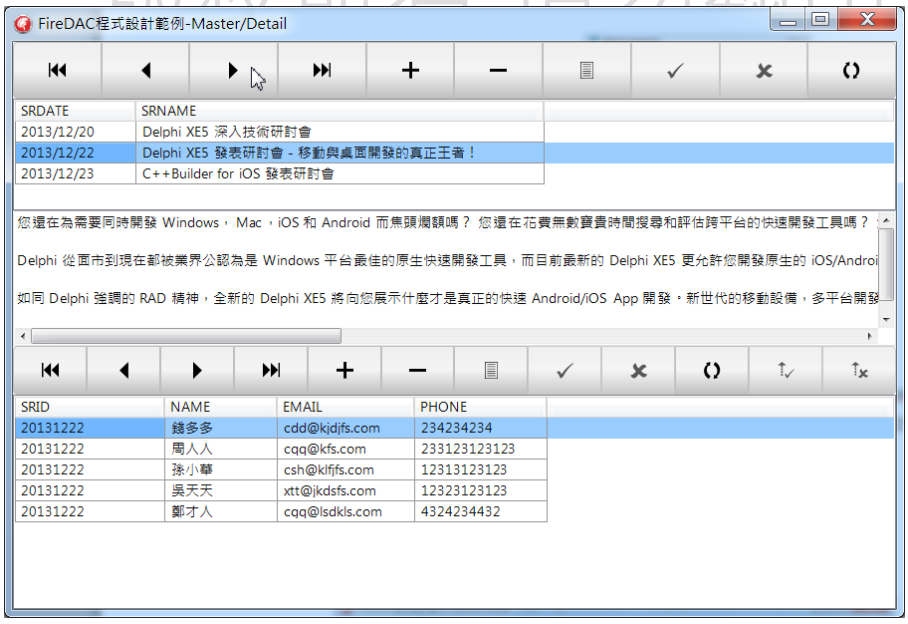
先设定 dsSeminars 的 DataSet 特性值为数据模块中的 fdqrySeminars，接着在对象查看器中设定 fdqrySeminarAttendee 如下的特性值：

特性	特性值
MasterDataSource	dsSeminars
MasterFields	SRID
IndexFieldNames	SRID

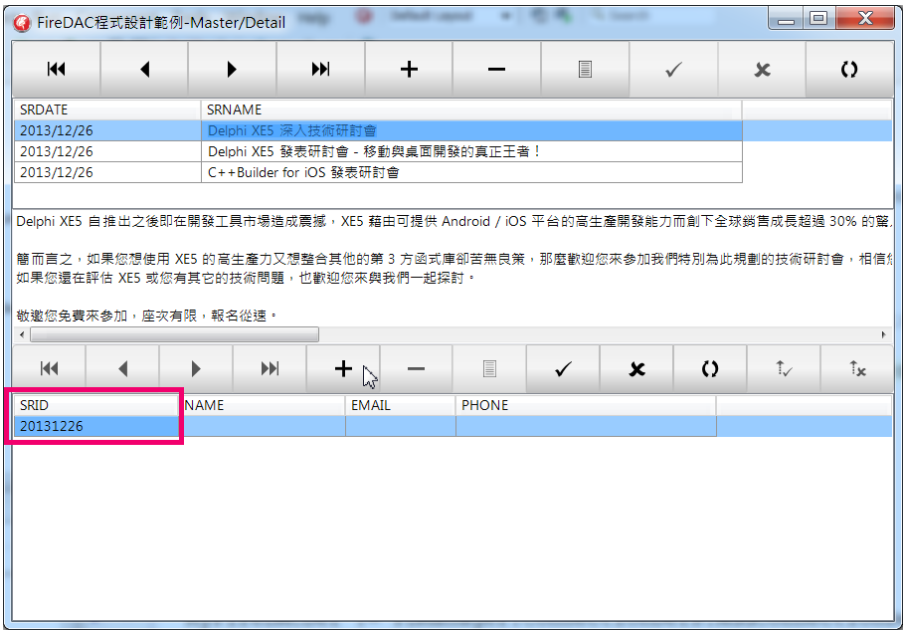
如下所示：



上面设定的意义肩简单，就是 fdqrySeminarAttendee 之中的资料要根据 fdqrySeminars 数据表中 SRID 域值的资料筛选，只有在 fdqrySeminarAttendee 数据表的 SRID 字段中拥有相同数值的数据才被筛选出来。现在如果执行这个范例程序并且在 SeminarAttendee 数据表中新增一些数据，那么在浏览 Seminars 数据表数据时就可以看到下方的 SeminarAttendee 数据表的数据也会跟着改变：



而且当您在 SeminarAttendee 数据表中新增数据时 SeminarAttendee 数据表的 SRID 字段会自动新增为目前 Seminars 数据表的 SRID 字段的数值，如下所示：



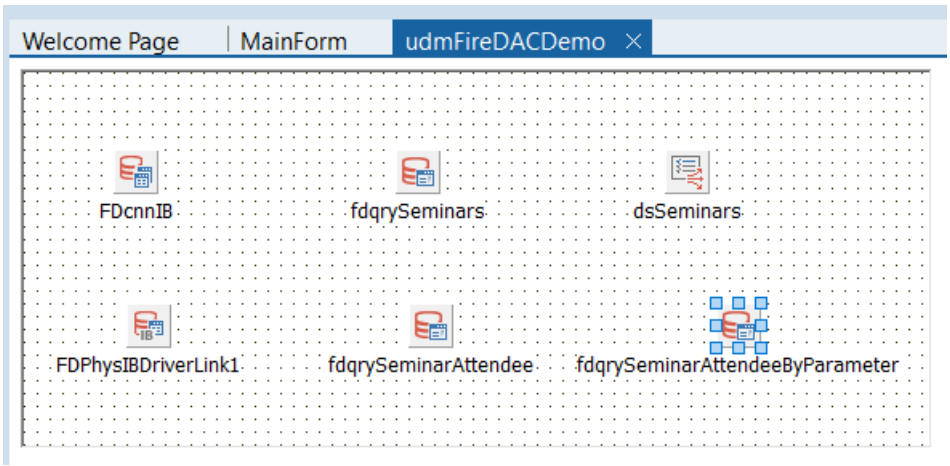
使用这种方式处理主从资的问题是如果从数据表中的数据很大量的话会导致客户端从数据库存取大量的数据到客户端，消耗大量网络和客户端内存的资源。因此在从数据表有大量数据的应用中您应该考虑下一个方式。

1-2-1-2 使用伺服端动态查询机制

第 2 种方式是使用动态查询的机制，为从数据表写一个包含参数的 SQL 命令再把主数据表和从数据表关连起来，如此一来当主数据表中的数据移动时就会自动把关连的域值带入数据表的 SQL 命令参数中再进行查询的动作。

现在在数据模块中放入一个名为 fdqrySeminarAttendeeByParameter 的 TFDQuery 组件，在它的 SQL 特性中使用：

```
select * from SEMINARATTENDEE where SRID = :SRID
```



上面的:SRID 就是动态参数，这个参数的名称和主数据表中关连的字段相同，因此在完成了稍后的设定之后主数据表就会在每次移动数据位置时自动把新的 SRID 域值填入这个 SQL 命令并且执行它。

接着在对象查看器中设定 fdqrySeminarAttendeeByParameter 如下的特性值：

特性	特性值
MasterDataSource	dsSeminars
MasterFields	SRID

再次执行就可以得到和上一个方式一样的结果。

使用动态查询机制的问题是每次主数据表的数据移动时就需要执行一次 SQL 命令，如果客户端的数量太多会造成数据库庞大的执行负荷，因此这种方式不适合使用在拥有大量客户端的应用中。

那么如果您拥有大量的从数据表数据又拥有大量的客户端怎么办呢？这不困难，请结合这 2 个方法，再结合后面章节说明的快储机制(Cache)即可。

要结合这 2 方法就是

1. 使用动态参数 SQL 命令

2. 设定从数据的 TFDQuery 的 MasterDataSource，MasterFields 和 IndexFieldNames 这 3 个特性值，如同 2-1-1 小节说明的

3. 设定从数据表的 FetchOptions.Cache 特性值包含 fiDetails

如何一来当一开始从数据表尚未有数据时 FireDAC 就会使用 2-1-2 小节的方式，但当主数据表的位置改变时，先前从数据表中的数据不会被丢弃而会被快储下来，如果主数据表的位置稍后又回到这个位置那么快储下来的从数据表资料就可以再被使用而无需再执行一次 SQL 命令了。

稍后的讨论快储机制的章节会有范例说明。

1-3 开发移动数据库 App

FireDAC 是目前 Delphi 提供的 3 种资料存取技术（BDE/IDAPI，dbExpress 和 FireDAC）中最适合使用来开发移动 App 的，因为

- FireDAC 无需部署额外的 DLL，

- FireDAC 可提供又小又快的执行速度
- FireDAC 直接支持移动平台的本地数据库：InterBase ToGo 和 SQLite
- FireDAC 可提同时联机和脱机数据处理的功能

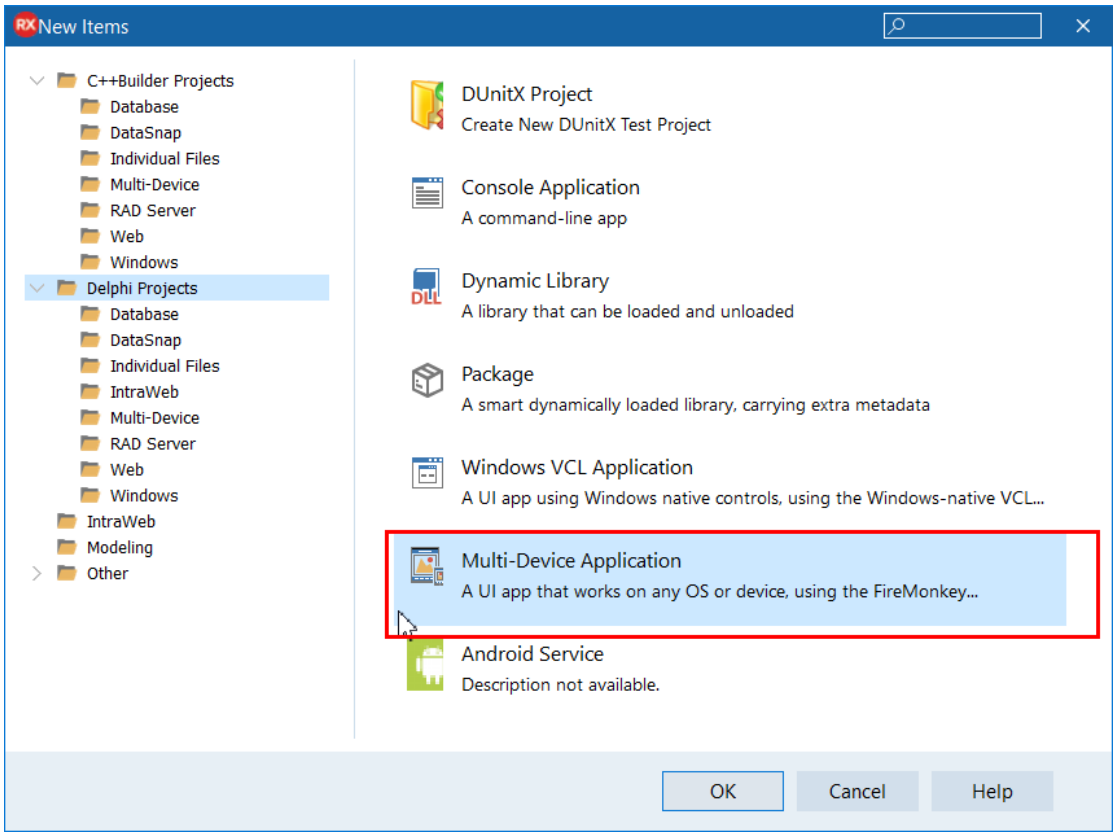
不过如果您想让移动平台和后端的数据库服务器链接的话，那么您必须配合使用 DataSnap，在 DX10 的版本中由于安全的因素 FireDAC 尚不支持直接从移动平台链接后端的数据库服务器。

本小节将说明如何使用 FireDAC 开发 iOS/Android 平台的数据库 App，不过本小节说明的是存取 iOS/Android 手机中的本地数据库，并不会说明如何藉由 DataSnap 链接后端的数据库伺机器，同时本小节也是使用 InterBase ToGo 做为说明的数据库，当然您也可以使用 SQLite。

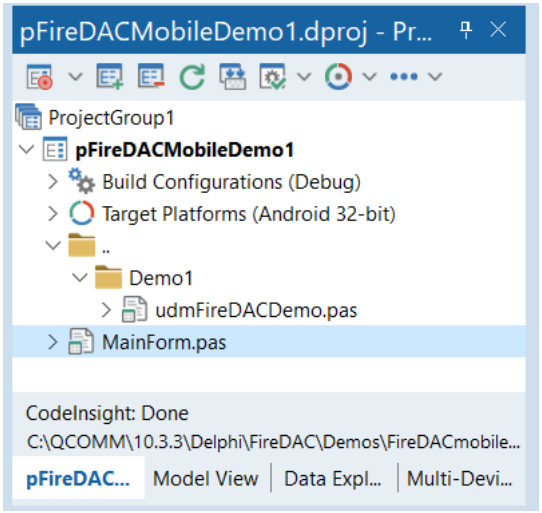
1-3-1 开发和部署 iOS/Android 手机 App

先让我们使用前面第 1 个范例 FireDAC 应用程序做为开发和部署就明的范例，现在我们要说明的就是如何把第 1 个范例 FireDAC 程序部署到 iOS 和 Android 平台中，由于第 1 个范例使用了数据模块来存取数据，而数据模块和平台 UI 无关因此可以重复使用在移动平台中。

首先建立一个 Multi-Device Application 项目：



储存项目并且把第 1 个范例的数据模块加入项目：



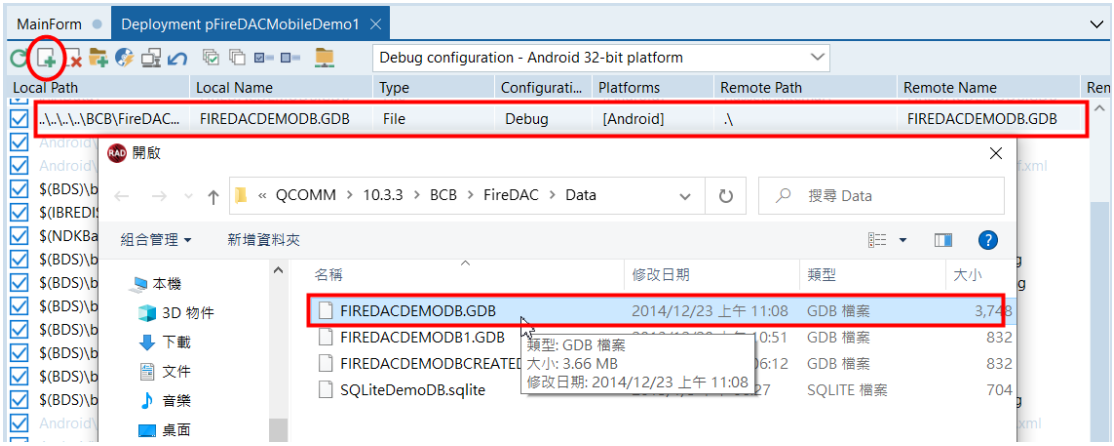
要开发手机上的数据库 App 有 2 种方式：

1. 部署 InterBase 或 SQLite 数据库到手机中

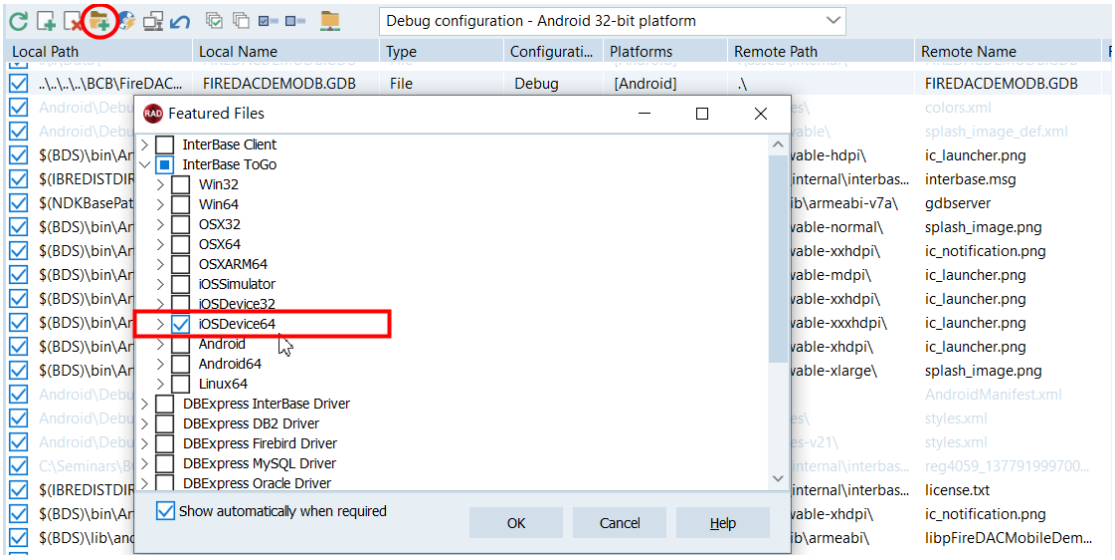
2. 直接使用程序代码在手机中建立需要使用的数据库

当然如果是结合 DataSnap 开发分布式手机数据库 App，那就有第 3 种方式，本小节将先说明第 1 种方式。

现在先让我们部署前面使用的范例数据库 FIREDACDEMODB.GDB 到手机中，让我们先开发 iOS App，请在项目管理员中双击 Target Platforms 节点中的 iOS Device 节点，再于 IDE 中点选 Project|Deployment 启动部署精灵，如下图点选部署精灵，先点选部署精灵左上方的『Add Files』按钮加入范例数据库 FIREDACDEMODB.GDB，如下所示：



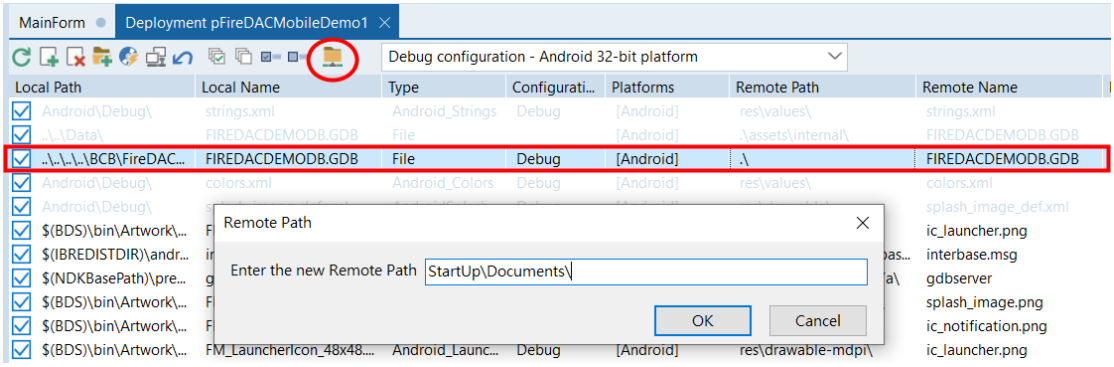
再点选部署精灵左上方的『Add Featured Files...』按钮加入 InterBase ToGo iOS 平台的相关档案，如下所示：



由于移动平台都有沙盒的概念。因此必须把范例数据库和 InterBase ToGo 的授权文件部署到移动平台正确的目录中，对于 iOS 平台范例数据库需要部署到

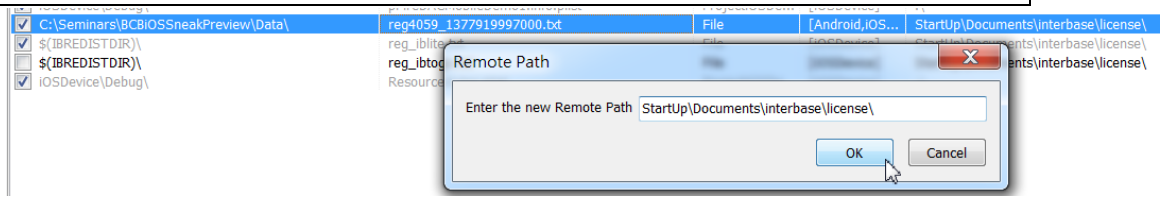
```
StartUp\Documents\
```

因此选部署精灵中的『Change Remote Path for Selected Items』按钮设定范例数据库到 StartUp\Documents\：



InterBase ToGo 的授权档部署到

```
StartUp\Documents\Interbase\license\
```

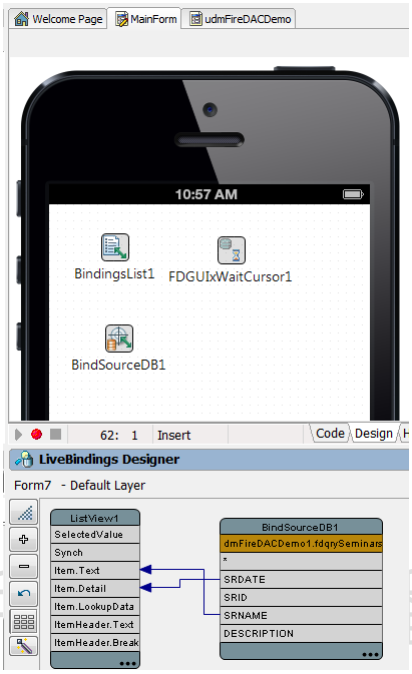


接着当然要修改 InterBase 数据库的所在地，要根据应用程用程序执行的平台从不同的地方连结或是加载。请开启数据模块为其中的 TFDConnection 建立如下的 OnBeforeConnect 事件处理函式：

```
procedure TdmFireDACDemo1.FDcnnIBBeforeConnect(Sender: TObject);
begin
  FDcnnIB.Params.Values['Database'] := String.Empty;
  {$IFDEF MSWINDOWS}
    FDcnnIB.Params.Values['Database'] :=
'127.0.0.1:F:\QComm\FireDAC\Data\' + DEMODBNAME;
  {$ENDIF}
  {$IF DEFINED(IOS) or DEFINED(ANDROID)}
    FDcnnIB.Params.Values['Database'] := TPath.GetDocumentsPath +
PathDelim + DEMODBNAME;
    FDcnnIB.Params.Values['Protocol'] := 'Local';
  {$ENDIF}
end;
```

在 FDcnnIBBeforeConnect 中我们使用 Delphi 的编译程序指令判断目前的执行平台，如果是 iOS 或 Android 平台就藉由 TPath 类别的 GetDocumentsPath 方法就可以取得前面用部署精灵的部署目录了。

回到项目的主画面放入 TFDGUIxWaitCursor 组件，再使用 Live Visual Bindings 功能链接范例数据库 FIREDACDEMODB.GDB 中的 Seminars 数据表：



编译之后就可以看到这个范例项目成功执行在 iOS 手机中了，而且显示了前面 Windows 版范例程序加入的数据：

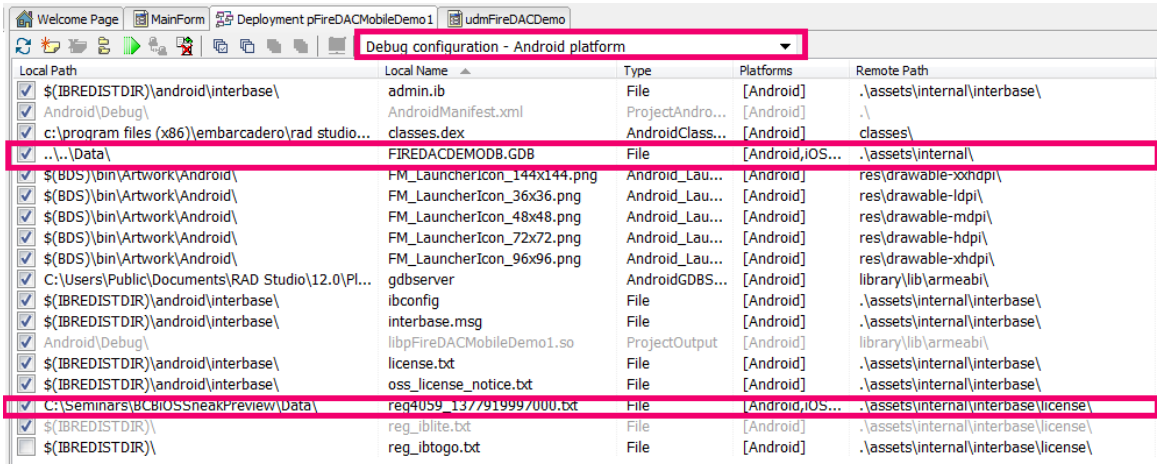
现在看看如何部署到 Android 平台，由于 Android 的沙盒目录不同因此现请先在项目经理中点选 Android 节点再开启部署精灵，如下图把范例数据库需要部署到

```
.\assets\internal\
```

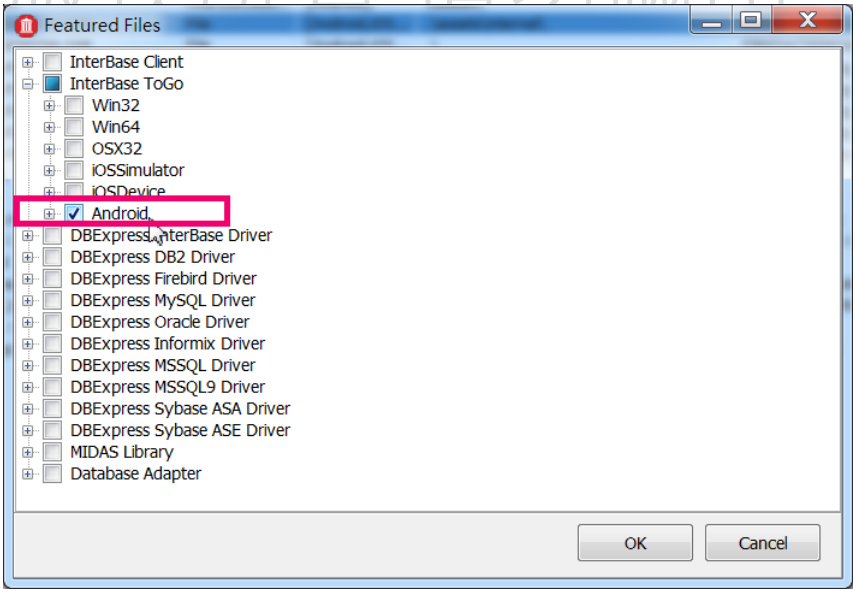
InterBase ToGo 的授权档部署到

```
.\assets\internal\interbase\license\
```

如下图所示：



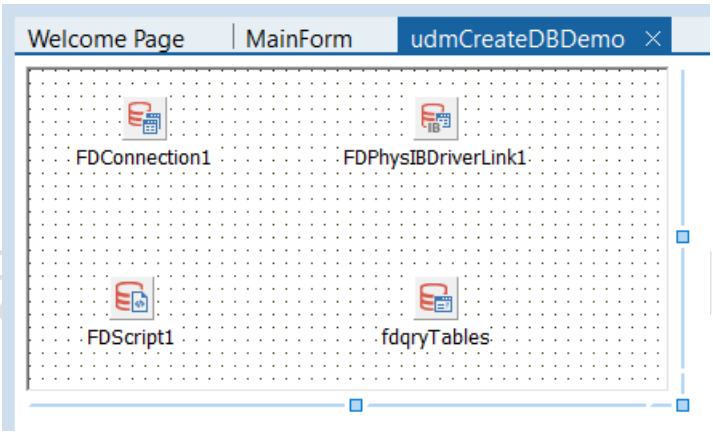
记得加入 Android 的相关支持档案：



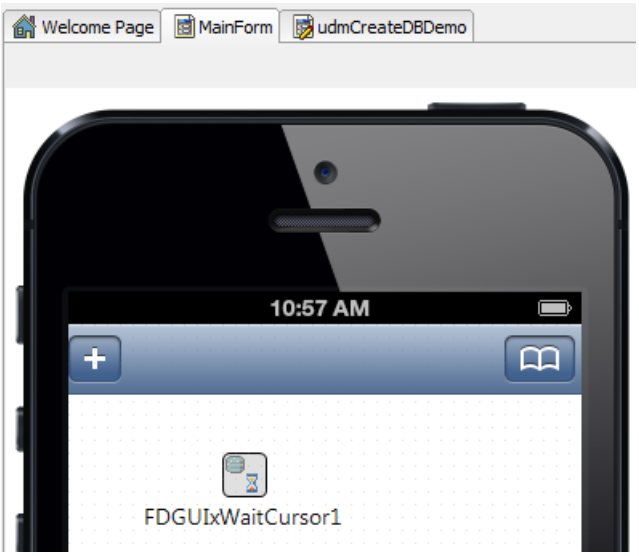
编译之后就可以看到这个范例项目成功执行在 Android 手机中了：

1-3-2 直接在 iOS/Android 手机中建立数据库

如果觉得根据不同的平台部署数据库很麻烦，那可以选择使用程序代码的方式直接建立手机平台中的数据库。例如您可以建立一个新的 FireMonkey Mobile Application 项目，在其中建立一个数据模块并且在其中放入 TFDConnection，TFDQuery，TFDPhysIBDriverLink 以及 TFDScript 组件：



在主窗体中加入 ToolBar 和 2 个 TButton 组件，TListView 组件以及 TFDGUIxWaitCursor：



当点选 ToolBar 左边的按钮时呼叫数据模块中的 CreateDatabase 方法建立数据库，再呼叫 SetupDatabase 设定数据库：

```
procedure TForm7.Button1Click(Sender: TObject);
begin
  dmCreateDBDemo.CreateDatabase;
  dmCreateDBDemo.SetupDatabase;
  DisplayTables;
end;
```

GetDatabase 方法使用 DDL 建立数据库并且藉由 TFDScript 组件执行此 DDL 建立范例数据库以及其中的范例数据表 SEMINARS 和 SEMINARATTENDEE：

```
const DEMODBNAME = 'FIREDACDEMODB.GDB';

function TdmCreateDBDemo.GetDatabase: String;
begin
  Result := String.Empty;
  {$IFDEF MSWINDOWS}
    Result := '127.0.0.1:F:\QComm\FireDAC\Data\' + DEMODBNAME;
  {$ENDIF}
  {$IF DEFINED(IOS) or DEFINED(ANDROID)}
    Result := TPath.GetDocumentsPath + PathDelim + DEMODBNAME;
  {$ENDIF}
end;

procedure TdmCreateDBDemo.CreateDatabase;
var
  sl : TStringList;
  sDatabaseSQL : String;
  sDatabase : String;
begin
  sDatabase := GetDatabase;
  if (FileExists(sDatabase)) then
    TFile.Delete(sDatabase);

  sl := TStringList.Create;
  try
```

```
    sl.Add('SET SQL DIALECT 3;');
    sDatabaseSQL := 'CREATE DATABASE %s USER ''SYSDBA'' PASSWORD
''masterkey'' PAGE_SIZE 4096 DEFAULT CHARACTER SET UTF8';
```

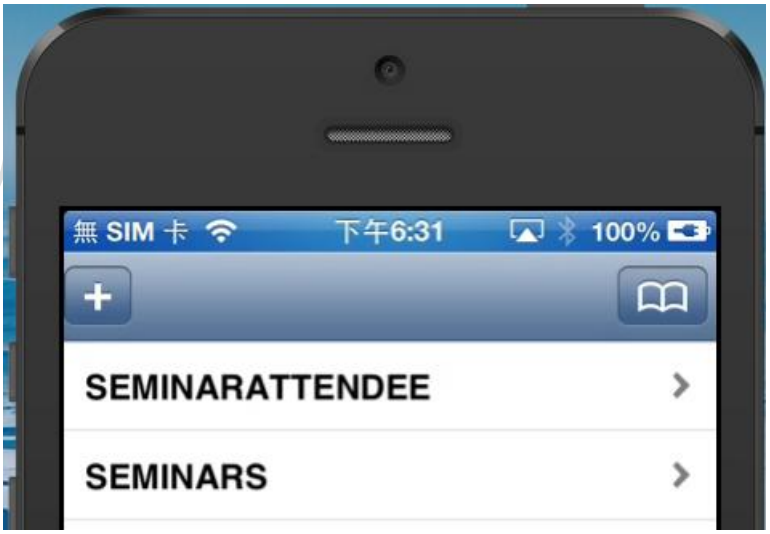
```
    sDatabaseSQL := Format(sDatabaseSQL, [sDatabase]);
    sl.Add(sDatabaseSQL);
    sl.Add('COMMIT;');
    sl.Add('CREATE TABLE "SEMINARATTENDEE" ');
    sl.Add('(');
    sl.Add('"SRID" INTEGER NOT NULL,');
    sl.Add('"NAME" VARCHAR(10) NOT NULL,');
    sl.Add('"EMAIL"   VARCHAR(50),');
    sl.Add('"PHONE"   VARCHAR(15)');
    sl.Add(');');
    sl.Add('CREATE TABLE "SEMINARS"');
    sl.Add('(');
    sl.Add('"SRDATE"  DATE NOT NULL,');
    sl.Add('"SRID" INTEGER NOT NULL,');
    sl.Add('"SRNAME"  VARCHAR(100) NOT NULL,');
    sl.Add('"DESCRIPTION" VARCHAR(600)');
    sl.Add(');');
  finally
    dmCreateDBDemo.FDScript1.ExecuteScript(sl);
    sl.Free;
  end
```

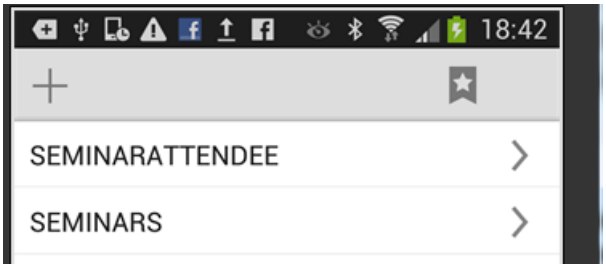

SetupDatabase 方法则使用前面介绍的使用程序代码的方式设定和链接范例数据库：

```
procedure TdmCreateDBDemo.SetupDatabase;
var
  sDatabase : String;
begin
  sDatabase := GetDatabase;
  if (FileExists(sDatabase)) then
  begin
    dmCreateDBDemo.FDConnection1.Connected := False;
    dmCreateDBDemo.FDConnection1.Params.Clear;
    dmCreateDBDemo.FDConnection1.DriverName := 'IB';
```

```
    dmCreateDBDemo.FDConnection1.Params.Values['User_Name'] :=
'sysdba';
    dmCreateDBDemo.FDConnection1.Params.Values['Password'] :=
'masterkey';
    dmCreateDBDemo.FDConnection1.Params.Values['Database'] :=
sDatabase;
    dmCreateDBDemo.FDConnection1.Params.Values['CharacterSet'] :=
'UTF8';
    dmCreateDBDemo.FDConnection1.Connected := True;
  end;
end;
```

编译之后就可以看到这个范例项目成功执行在 iOS/Android 手机中并且显示范例数据库中的范例数据表而无需使用部署精灵先根据不同的平台部署数据库了。





1-4 结论

本章先介绍如何使用 FireDAC 组件链接数据库和处理简易的数据，下一章会讨论更多使用 FireDAC 处理数据的技术。

第2章 处理数据

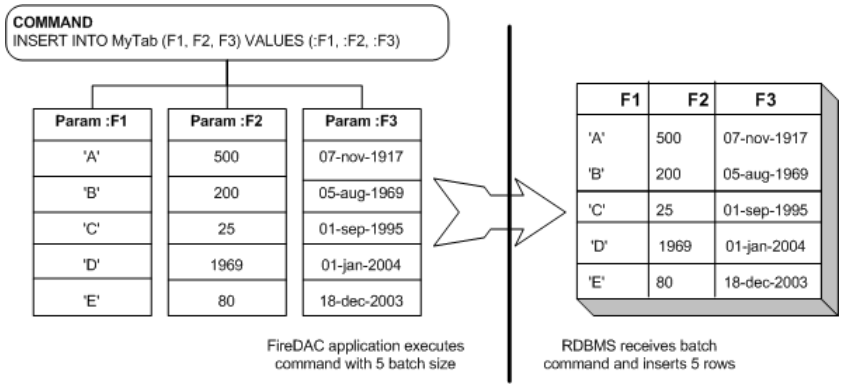
本章要说明如何使用 FireDAC 来处理数据，例如搜寻，快储机制，异步处理数据等实用的功能，在您阅读完本章内容之后就能够使用很有效率的方式在应用程序中处理数据了。

2-1 使用 Array DML 处理大量数据

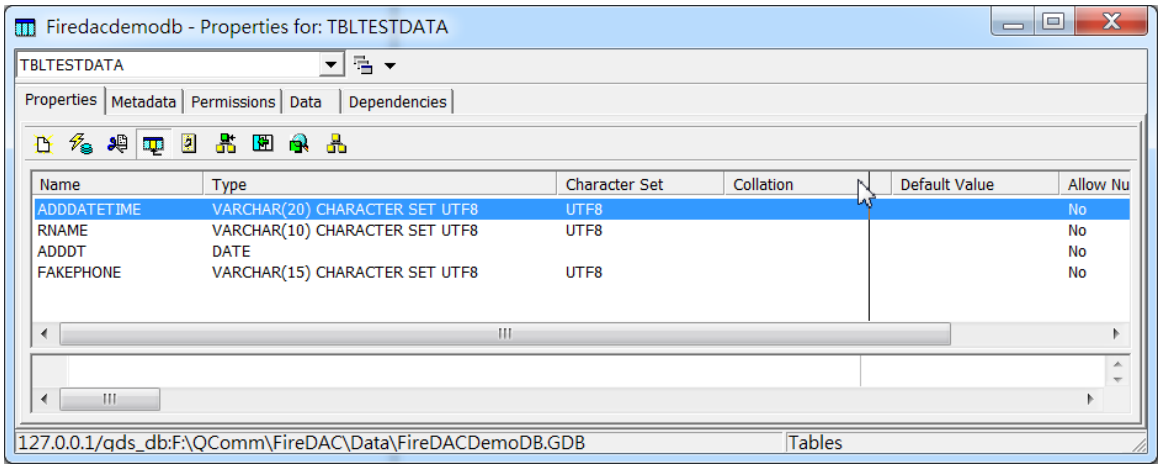
由于本章需要使用大量的随机数据做为说明本章的范例，因此需要在范例数据表中新增大量的随机数据，因此正好藉由这个需求先来说明如何使用 FireDAC 快速有效率的在应用程序中处理大量的数据。

FireDAC 提供了所谓的数组数据处理功能(Array Data Manipulation Language, Array DML)来处理应用程序的大量数据需求。例如需要大量新增数据或是大量修改数据的应用中。Array DML 可以想成批处理的概念，在一般的应用中如果客户端要新增大量的数据，那么客户端需要为每一笔数据呼叫一次 Insert 和 Post 方法，或是执行一次 SQL 的 Insert 指令。但如果客户端想一次加入 10000 笔数据那这需要执行 1000 次，如此一来速度当然不好也会造成网络频繁的负荷。因此 Array DML 的概念是在客户端一次把 10000 笔数据送到数据库并且让数据库一次执行完毕，如此一来速度当然就快多了。

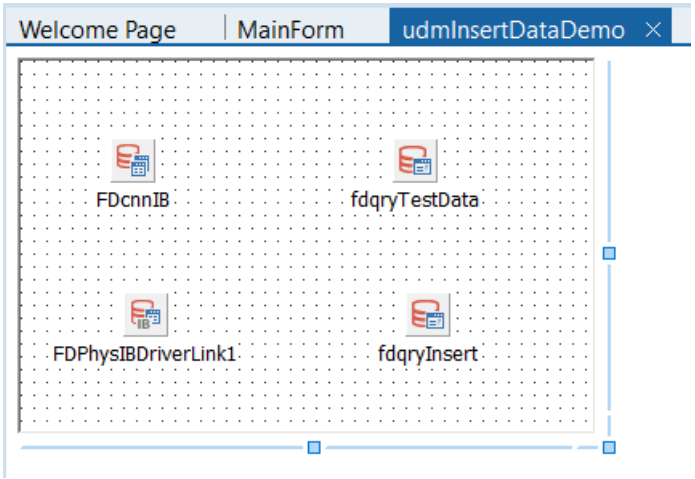
FireDAC 文件中的一个图形清楚的说明了 Array DML 的概念，下图显示了客户端使用 Array DML 新增 5 笔资料,FireDAC 的 Array DML 会一次把这些数据送到后端并且新增到数据库中：



现在就让我们使用数据表 TBLTESTDATA 来做为说明的范例：



请建立一个 FireMonkey Desktop Application 项目并且在其中再建立一个数据模块，接着在其中放入 TFDConnection，2 个 TFDQuery 和 TFDPhysIBDriverLink 组件，如下所示：



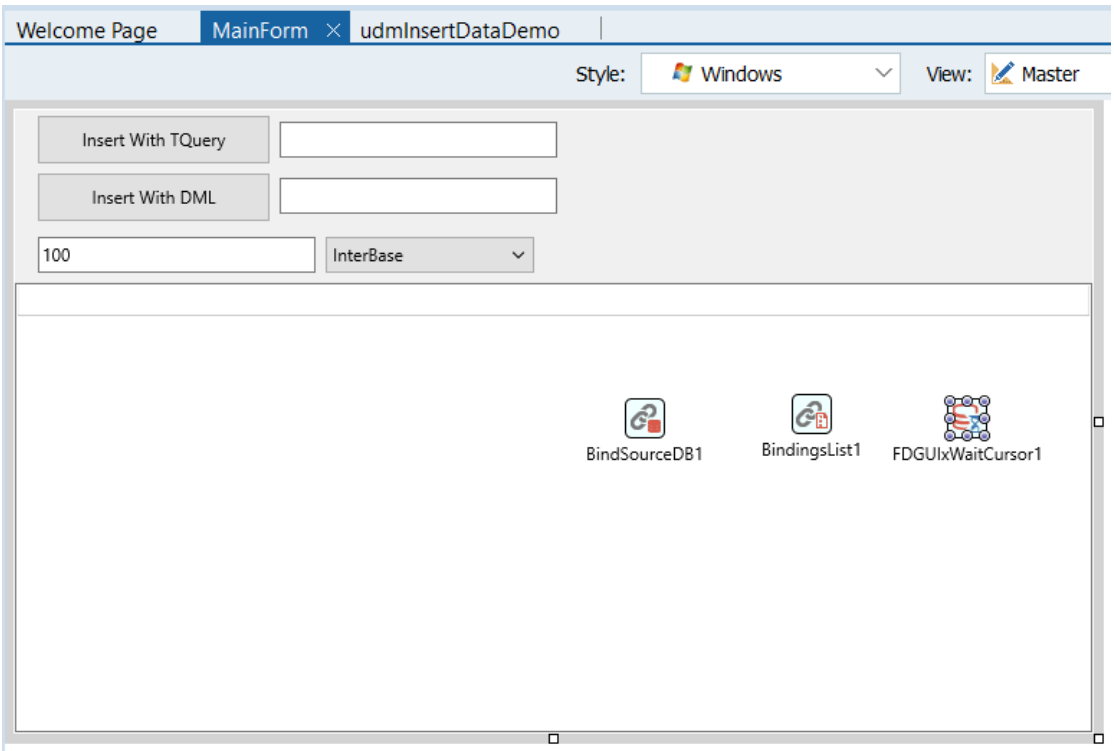
在上图的数据模块中 fdqryTestData 的 SQL 特性使用下面的 SQL 从 TBLTESTDATA 撷取数据：

```
select * from TBLTESTDATA order by RNAME
```

而 fdqryInsert 组件的 SQL 特性使用下面的 SQL 在 TBLTESTDATA 数据表中新增数据：

```
INSERT INTO TBLTESTDATA
(ADDDATETIME, RNAME, ADDDT, FAKEPHONE)
VALUES (:NEW_ADDDATETIME, :NEW_RNAME, :NEW_ADDDT, :NEW_FAKEPHONE)
```

在主窗体中放入 2 个 TButton 按钮分别以不同的方式在 TBLTESTDATA 中加入数据。『Insert With TQuery』按钮使用一般的方式为每一笔新增数据执行一次上面的 Insert SQL 命令：



『Insert With TQuery』按钮的 OnClick 实作程序代码如下，它藉由 TFDQuery 的 Params 特性把每笔新增数据的参数填入，最后呼叫 ExecSQL 方法新增数据到数据表中：

```
procedure TdmInsertDataDemo.InsertDataWithQuery(const iRecord :
Integer);
var
  iIndex: Integer;
begin
  for iIndex := 1 to iRecord do
```

```
  begin
    fdqryInsert.Params.ParamByName('NEW_ADDDT').Value := Now;
    fdqryInsert.Params.ParamByName('NEW_ADDDATETIME').Value :=
GetRandomDateTime(Now);
    fdqryInsert.Params.ParamByName('NEW_RNAME').Value :=
GetRandomName;
    fdqryInsert.Params.ParamByName('NEW_FAKEPHONE').Value :=
GetRandomPhone;
    fdqryInsert.ExecSQL;
  end;
end;
```

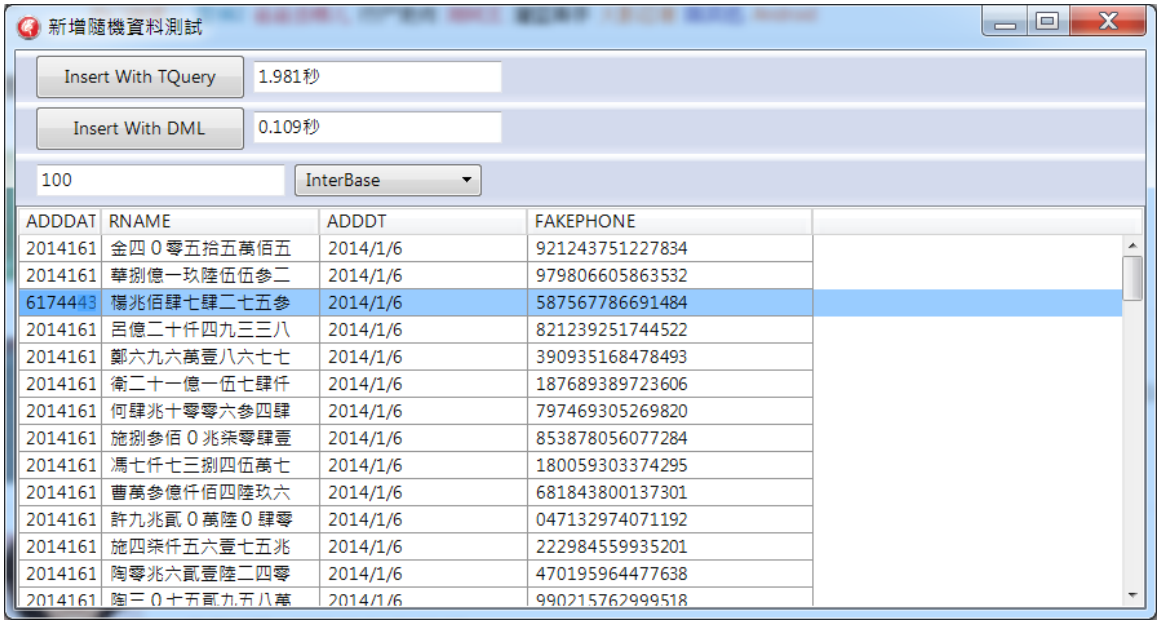
『Insert With DML』按钮使用 Array DML 新增数据，要使用 Array DML 首先要指定 TFDQuery 的 Params 特性的 ArraySize 特性告知 FireDAC 要一次处理的数据笔数，一旦指定了 ArraySize 特性值之后就可以使用 TFDQuery 的 Params 特性的 ParamByName 方法填入参数，但要注意的是在 Array DML 中 ParamByName 回传的是一个参数数组，因此在填入参数值时需要填入到正确的数组元素中。

例如下面就是『Insert With DML』按钮的 OnClick 事件处理函式，在 005 行先指定我们需要新增的笔数，接着进入循环为每一笔数据填入参数值，最后在 013 行执行一次 TFDQuery 的 Execute 方法并传入数组的大小(新增的笔数)即可：


```
001   procedure TdmInsertDataDemo.InsertDataWithDML(const
iRecord : Integer);
002   var
003     iIndex: Integer;
004   begin
005     fdqryInsert.Params.ArraySize := iRecord;
006     for iIndex := 0 to iRecord - 1 do
007     begin
008
fdqryInsert.Params.ParamByName('NEW_ADDDT').AsDates[iIndex] :=
Now;
009
fdqryInsert.Params.ParamByName('NEW_ADDDATETIME').AsStrings[iInd
ex] := GetRandomDateTime(Now);
010
fdqryInsert.Params.ParamByName('NEW_RNAME').AsStrings[iIndex] :=
```

```
GetRandomName;
011
fdqryInsert.Params.ParamByName('NEW_FAKEPHONE').AsStrings[iIndex
] := GetRandomPhone;
012      end;
013      fdqryInsert.Execute(fdqryInsert.Params.ArraySize);
014    end;
```

现在执行这个范例程序并且使用不同的方法新增数据，从下图执行的结果我们可以看到其执行速度相差了 10 几倍，可见 Array DML 的快速以及适合使用在大量数据处理的应用中：



虽然本范例是使用新增数据，事实上 Array DML 也可以使用在修改资料的应用中。此外如果再结合稍后的异步处理的功能，您可以藉由 FireDAC 写出非常高效的数据库应用程序。

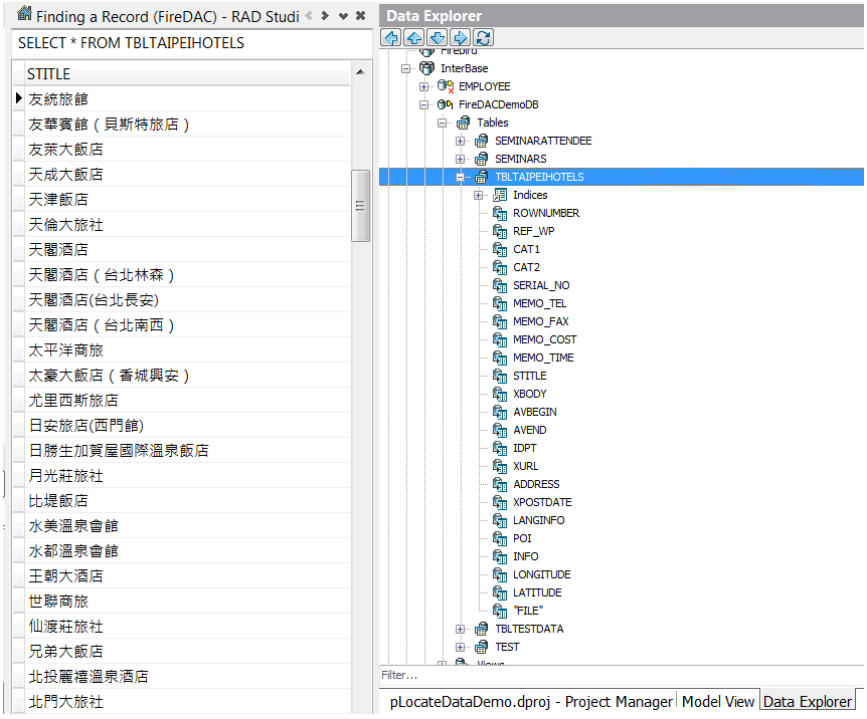
2-2 搜寻数据

FireDAC 提供了数种方式让程序员在 FireDAC 的数据集组件中搜寻数据，当程序员藉由 TFDQuery 组件执行 SQL 命令从后端数据库取得数据之后这些数据就暂时储存在 TFDQuery 组件中。而在一般的应用程序中我们经常会需要在其中搜寻特定的数据，FireDAC 提供了许多的方法让程序员在数据集中搜寻数据，本小节将说明如何使用这些功能来搜寻数据。

本小节使用的范例数据是台北市公布的台北市旅馆资料，其位于：

```
http://data.taipei.gov.tw/opendata/apply/NewDataContent?oid=3962
468C-7AA9-49F4-965A-5EC30CE272E3
```

下面是笔者将这些数据转换到InterBase数据库中以方便说明如何使用FireDAC搜寻数据：



SELECT * FROM TBLTAIPEIHOTELS

ROWNUMBER	REF_WP	CAT1	CAT2	SERIAL_NO	MEMO_TEL	MEMO_FAX	MEMO_COST
33	6	住宿	一般旅館	B0225	0227735177	0227727569	1280以上
47	6	住宿	一般旅館	B0004	0227630505	0227699571	1000以上
34	6	住宿	一般旅館	B0243	0225978800	0225951115	1550以上
38	6	住宿	一般觀光旅館	B0326	0223617856	0223118902	4500以上
42	6	住宿	一般旅館	B0092	0225365678	0225365228	1500以上
35	6	住宿	一般旅館	B0313	0228813133	0228813133	1000以上
57	6	住宿	一般旅館	B0390	0225288000	0225287676	7600以上
56	6	住宿	一般旅館	B0389	0225319999	0225816777	5700以上
53	6	住宿	一般旅館	B0499	022531-7777	022562-6777	6560以上
37	6	住宿	一般旅館	B0297	0225679999		5900以上
30	6	住宿	一般旅館	B0232	0287802000	0287808100	5200以上
32	6	住宿	一般旅館	B0224	0227195152	0227195156	2500以上
58	6	住宿	一般旅館	B0471	022562-5082	0225622049	2800以上
60	6	住宿	一般旅館	B0462	0223317770	2523317772	
62	6	住宿	一般觀光旅館	B0436	0228911111	0228922222	24000以上
45	6	住宿	一般旅館	B0163	0228914478	0228955850	500以上
41	6	住宿	一般旅館	B0269	0225066768	0225066755	2350以上
50	6	住宿	一般旅館	B0065	0228983838	0228984505	5600以上
51	6	住宿	一般旅館	B0055	0228979060	0228979065	2880以上
39	6	住宿	一般旅館	B0344	0227197199	0225459288	7200以上
93	6	住宿	一般旅館	B0246	0225626161	0225626448	2400以上
84	6	住宿	一般旅館	B0150	0228913537	0228960663	800以上
89	6	住宿	國際觀光旅館	B0343	0227123456	0227173334	4300以上
68	6	住宿	一般觀光旅館	B0457	0228988888	0228988088	10560以上
79	6	住宿	一般旅館	B0080	0225567797		600以上
85	6	住宿	一般旅館	B0105	0225017601	0225043635	3200以上
82	6	住宿	一般旅館	B0167	0223067408	0223081388	800以上
74	6	住宿	一般旅館	B0485	0223140245	0223822425	
64	6	住宿	國際觀光旅館	B0430	0277038888	0277038899	15000以上
69	6	住宿	一般旅館	B0459	0223755111	0223710077	
67	6	住宿	一般觀光旅館	B0434	0221819999	0221819988	11000以上
88	6	住宿	國際觀光旅館	B0348	0227201234	0227201111	14000以上
66	6	住宿	一般觀光旅館	B0433	0223146611	0223145511	8000以上
86	6	住宿	一般旅館	B0279	0223113201	0223113209	1800以上
71	6	住宿	一般旅館	B0502			
76	6	住宿	一般旅館	B0483	0225682270	0225682201	1700以上

2-2-1 Locate 和 LocateEx

FireDAC 的 TFDQuery 组件提供了兼容于 BDE/dbExpress 的 Locate 方法以及其强化的 LocateEx 方法让程序员在 TFDQuery 中搜寻数据。

当使用 Locate 方法搜寻数据时，开发人员可以使用任何的字段条件来搜寻，而不用管这个字段是不是索引字段．当然，当开发人员使用索引字段来搜寻数据时 Locate 会直接使用索引来帮助搜寻，因此速度会非常的快速．如果开发人员使用非索引字段搜寻数据，那么 Locate 也将使用目前它知道最好的方式来搜寻数据．

此外 Locate 方法不只能够搜寻一个单一的字段，它更能够同时的以数个字段的条件来搜寻数据．开发人员可以组合数个字段的搜寻条件在结果数据集中搜寻数据．

由于 Locate 能够搜寻各种不同数据型态的字段，因此 Locate 方法在设定搜寻条件时是以 Variant 型态的变量来储存搜寻数值．当开发人员要使用多个字段来搜寻数据时必须建立一个 Variant 数组来储存搜寻数值．

此外 Locate 方法在搜寻数据时也能够使用模糊条件标准来寻找特定的资料，例如开发人员可以要求 Locate 在搜寻资料时不分辨大小写，或是以部份字符串来搜寻数据，提供开发人员非常大的弹性空间．

下面就是 Locate 的方法原型：

```
   function Locate(const AKeyFields: string; const AKeyValues:
Variant;
     AOptions: TLocateOptions = []): Boolean; override;
```

Locate 方法接受三个参数，第一个参数 AKeyFields 是开发人员要搜寻的域名。如果开发人员要搜寻单一字段，那么只需要直接传入此域名。如果要以多个字段条件来搜寻，那么开发人员便需传入所有的域名，并且以分号分隔每一个域名。例如，如果要搜寻 Name 和 Email 两个字段，那么 AKeyFields 就必须是:

```
'Name;EMail'
```

第二个参数 AKeyValues 是指开发人员欲搜寻的条件数值。它的型态是 Variant，因为 Variant 几乎可以代表任何的型态，因此开发人员可以搜寻整数，小数，字符串，或是布尔值的条件。同样的如果开发人员只搜寻一个条件数值，那么就可以直接在这个参数位置传入。如果是要以多个字段条件来搜寻，那么开发人员必须建立一个 Variant 数组，然后在这个数组中的每一个元素中指定条件数值，再传递这个 Variant 数组到这个参数中。至于 Variant 数组则可以使用 VarArrayOf 方法，或是使用 VarArrayCreate 方法来建立，在稍后的范例中会有程序代码说明。

Locate 方法的最后一个参数 TLocateOptions 则是让开发人员在搜寻字符串字段时，指定以什么标准来搜寻数据。开发人员可以指明不分大小写来搜寻字符串数据，或是以部份字符串数值来搜寻数据。下面就是 TLocateOptions 的型态定义：

```
TLocateOption = (loCaseInsensitive, loPartialKey);
TLocateOptions = set of TLocateOption;
```

在使用 Locate 时，如果使用 loCaseInsensitive 就代表不分大小写搜寻数据，如果使用 loPartialKey 就代表要以部份字符串来搜寻数据。

Locate 方法的回传数值是布尔值，它代表 Locate 方法是否成功的找到了要搜寻的数据。如果找到的话，就回传 True，否则就回传 False。当 Locate 方法成功的搜寻到数据之后，它就会移动目前的记录位置到这笔数据之上，否则就会停留在 Locate 开始搜寻之前的记录位置上。请注意 Locate 方法搜寻数据的结果是一笔数据，因此如果你想搜寻符合条件的一群资料，那么你可以使用稍后介绍的过滤器(Filter)功能。

现在让我们使用数个范例来说明如何使用 Locate 方法。下面的范例程序代码即是以一个字段来搜寻数据，它是以数据表的 NAME 字段来搜寻拥有”利瓦伊”数值的这笔数据，由于最后一个参数是空集合，因此这代表必须 NAME 字段拥有一模一样的”利瓦伊”这个数值才算搜寻成功。

```
FDQuery1.Locate('NAME', '利瓦伊', []);
```

下面的程序代码则是以两个字段 City 和 District 来搜寻数据，搜寻的条件是 City 字段拥有"台北"数值，而且 District 字段拥有" 大安区"数值的资料。

```
FDQuery1.Locate('City;District', VarArrayOf(['台北,大安区']),[]);
```

下面的程序代码和第一个范例非常的相像，只是这个程序代码搜寻的是第一笔在 NAME 字段以"李"数值开头的数据。

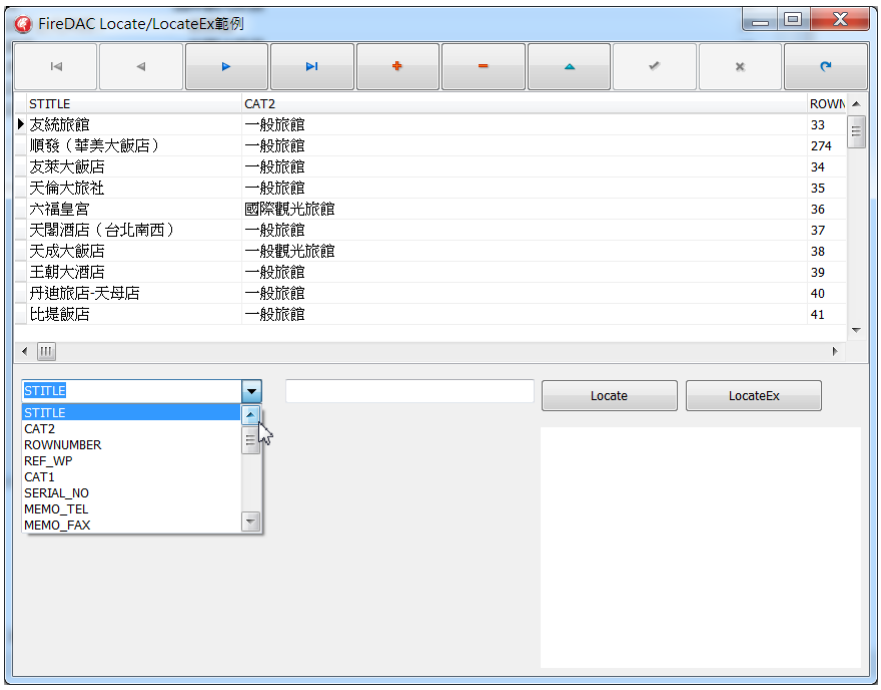
```
FDQuery1.Locate('NAME', '李', [loPartialKey]);
```

最后一个范例则是搜寻 ID 字段中任何以"A12"数值开头的第一笔，而且不分 A 大小写的数据。

```
FDQuery1.Locate('ID', 'A12', [loCaseInsensitive ,loPartialKey]);
```

现在就让我们使用 Locate 方法在范例应用程序中搜寻数据。

在下面的 FireDAC Locate/LocateEx 范例中先使用 TFDQuery 组件从范例数据表 TBLTAIPEIHOTELS 中取得数据，再于 TComboBox 中填入 TBLTAIPEIHOTELS 所有的域名：

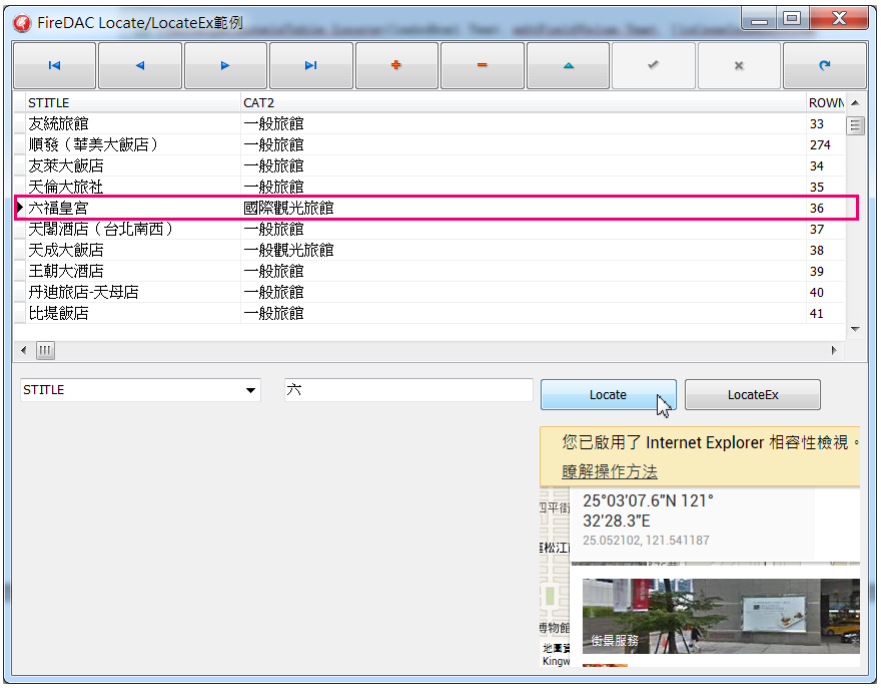


Locate 单字段搜寻

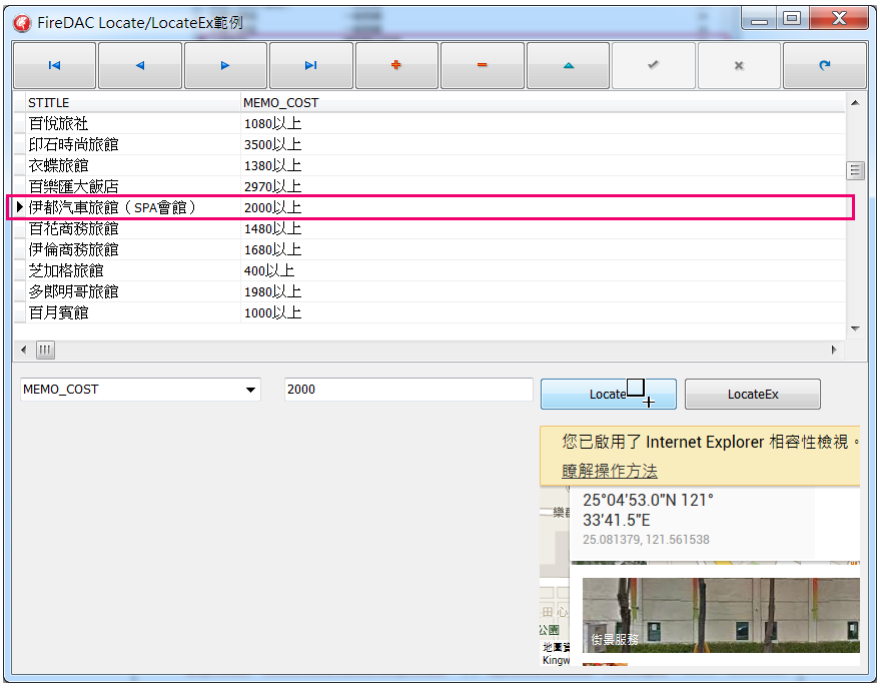
现在就让我们使用 Locate 方法在范例应用程序中搜寻数据，先让我们以单一的字段来展示如何搜寻数据，稍后再说明如何以多个字段搜寻数据。在上图 Locate 按钮的 OnClick 事件中使用 TFDQuery 的 Locate 方法搜寻使用者选择要搜寻的字段以及域值：

```
procedure TfmMainForm.Button1Click(Sender: TObject);
begin
  if (TbltaipeihotelsTable.Locate(ComboBox1.Text, edtFieldValue.Text,
[loCaseInsensitive, loPartialKey])) then

DisplaySearchedHotel(TbltaipeihotelsTable.FieldByName('LONGITUDE').As
String, TbltaipeihotelsTable.FieldByName('LATITUDE').AsString);
end;
```

执行此范例程序在 TComboBox 中选择要搜寻的字段和输入部份域值之后点选 Locate 按钮应该就可以找到您要搜寻的数据了，例如下图就是搜寻名称以＂六＂开头的旅馆：



如果使用其他字段，例如 MEMO_COST 也可以搜寻到数据：

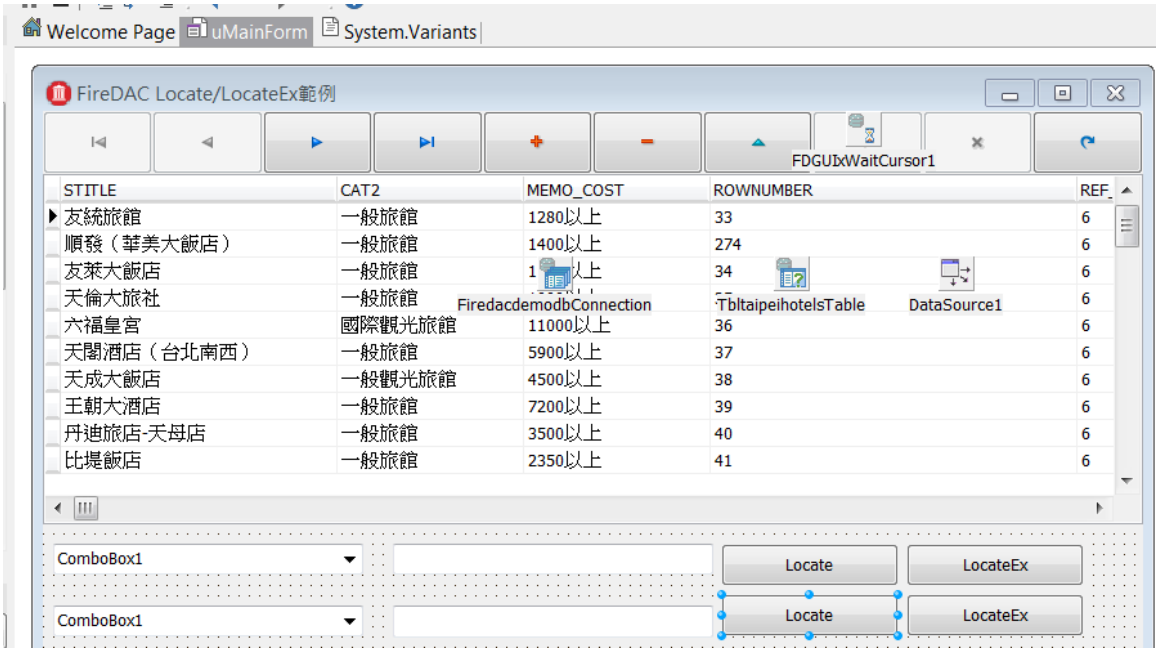


Locate 多字段搜寻

如果同时要以多个字段来搜寻，那么我们必须把搜寻字段以分号分隔做为 Locate 方法的第一个参数同时把所有搜寻域值以 VarArrayOf 方法建成 Varaint 做为 Locate 方法的第二个参数，例如下面就是同时使用 STITLE 和 MEM_COST 这 2 个字段来搜寻数据：


```
FDQuery1.Locate('STITLE;MEM_COST', VarArrayOf([edtFieldValue.Text,
edtFieldValue2.Text]), [loCaseInsensitive, loPartialKey]);
```

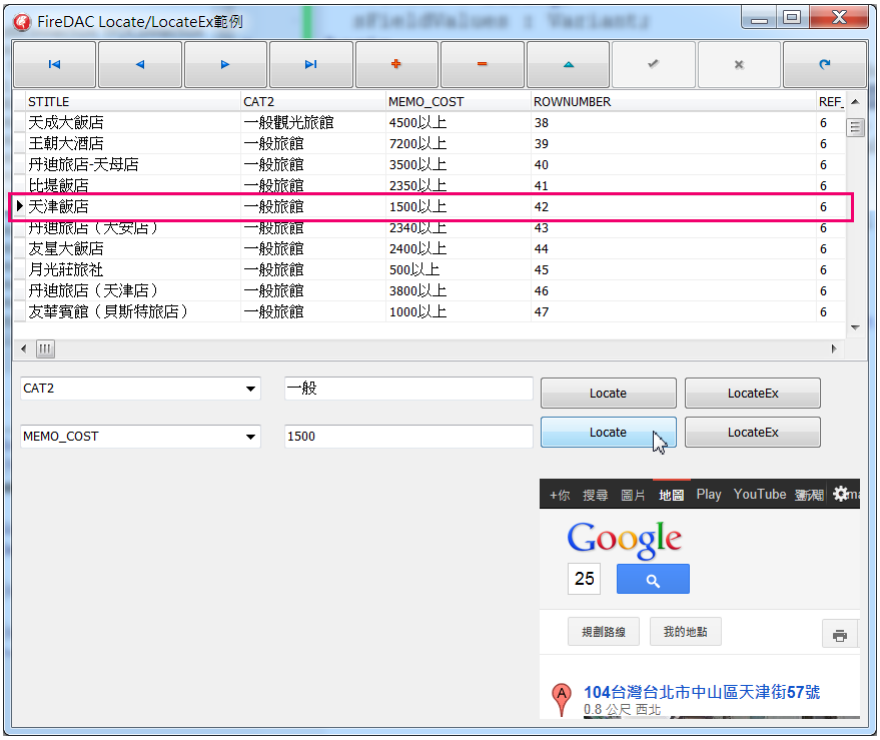
现在回到刚才的范例程序加入第 2 个 TComboBox 和 TEdit 和 TButton 组件如下：



在第 2 个 Locate 按钮的 OnClick 事件处理函式中撰写如下的程序代码：

```
001    procedure TfmMainForm.Button3Click(Sender: TObject);
002    var
003      sFields : String;
004      sFieldValues : Variant;
005    begin
006      sFields := ComboBox1.Text + ';' + ComboBox2.Text;
007      sFieldValues := VarArrayOf([edtFieldValue.Text,
edtFieldValue2.Text]);
008      if (TbltaipeihotelsTable.Locate(sFields, sFieldValues,
[loCaseInsensitive, loPartialKey])) then
009
DisplaySearchedHotel(TbltaipeihotelsTable.FieldByName('LONGITUDE').As
String, TbltaipeihotelsTable.FieldByName('LATITUDE').AsString);
010    end;
```

006 先以分号结合搜寻域名，007 行使用 VarArrayOf 函式以使用者输入的搜寻域值建立 Variant 数组，就可以在 008 行呼叫 Locate 方法搜寻数据了。下图就是执行的结果，我们可以看到 Locate 方法成功的以 2 个字段搜寻到了数据。



使用 LocateEx 搜寻数据

Locate 方法使用上非常的简单，但 FireDAC 又提供了强化的 LocateEx 方法让程序员可以使用更多的方式搜寻数据。FireDAC 提供了 2 个版本的 LocateEx 方法，下面是第 1 个宣告原型：

```
function LocateEx(const AKeyFields: string; const AKeyValues:
Variant;
     AOptions: TFDDataSetLocateOptions = []; ApRecordIndex:
PInteger = nil): Boolean; overload; virtual;
```

这个 LocateEx 方法的第 1 和第 2 个参数的意义和前面的 Locate 方法是一样，但第 3 个参数的型态是 TFDDataSetLocateOptions，它扩展了 Locate 方法的 TLocateOptions 定义。TFDDataSetLocateOptions 的定义如下：

```
  TFDDataSetLocateOption = (lxoCaseInsensitive, lxoPartialKey,
lxoFromCurrent,
    lxoBackward, lxoCheckOnly, lxoNoFetchAll);
  TFDDataSetLocateOptions = set of TFDDataSetLocateOption;
```

下面的表格说明了其中定义值的意义：

特性	特性值
lxoFromCurrent	从当前记录的位置开始搜寻
lxoBackward	从当前记录的位置开始往前搜寻
lxoCheckOnly	只搜寻看看要搜寻的数据存不存在而不改变当前记录位置
lxoNoFetchAll	只搜寻在 TFDQuery 数据集中的数据而不要把所有的数据从后端数据库中取出再搜寻

从上面的表格说明可以知道 LocateEx 和 Locate 不同的地方是：

1. Locate 只能从数据集数据开始的地方往后搜寻，而 LocateEx 则可以从开始处，从目前位置往前或是往后搜寻

2. Locate 在开始搜寻数据时会先把后端所有的数据取到数据集中再搜寻，而 LocateEx 不但可以向 Locate 一样，也可以控制只在目前的数据集中再搜寻而不要把所有的数据从后端数据库中取出再搜寻，这样可以减少数据的流量。

至于 LocateEx 的第 4 个参数 ApRecordIndex 如果有指定的话就会回传搜寻到的数据在数据集中的位置。

第 2 个版本的 LocateEx 方法宣告如下：


```
function LocateEx(const AExpression: string;
     AOptions: TFDDataSetLocateOptions = []; ApRecordIndex:
PInteger = nil): Boolean; overload; virtual;
```

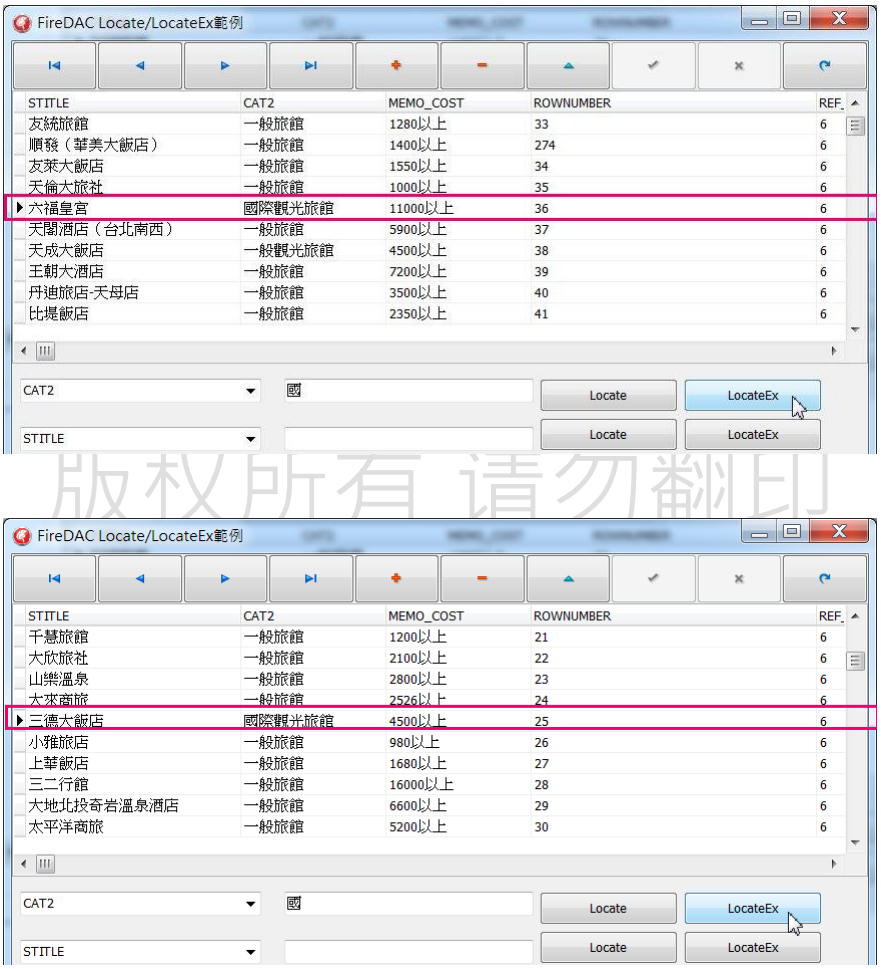
它不同的地方是第 1 个参数 AExpression。AExpression 是字符串型态，代表程序员可以使用一个表达式条件来搜寻资料，

如果现在我们希望能够根据特定域值来持续的搜寻所有符合的数据而不是像 Locate 方法永远只能找到第 1 笔，那么我们可以在范例程序的第 1 个 LocateEx 按钮中实作如下的程序代码：

```
procedure TfmMainForm.Button2Click(Sender: TObject);
begin
  if (TbltaipeihotelsTable.LocateEx(ComboBox1.Text, edtFieldValue.Text,
[lxoPartialKey, lxoFromCurrent])) then
  begin

DisplaySearchedHotel(TbltaipeihotelsTable.FieldByName('LONGITUDE').As
String, TbltaipeihotelsTable.FieldByName('LATITUDE').AsString);
    edtHotelName.Text :=
```

```
TbltaipeihotelsTable.FieldByName('STITLE').AsString;
  end
  else
    edtHotelName.Text := '没有找到!';
end;
```

我们呼叫 LocateEx 方法并且使用[lxoFromCurrent]要求 FireDAc 从当前记录持续搜寻数据，例如下图就是在 CAT2 字段中搜寻"国"开头的旅馆，就可以不断的搜寻国际观光旅馆，这是 Locate 方法做不到的：

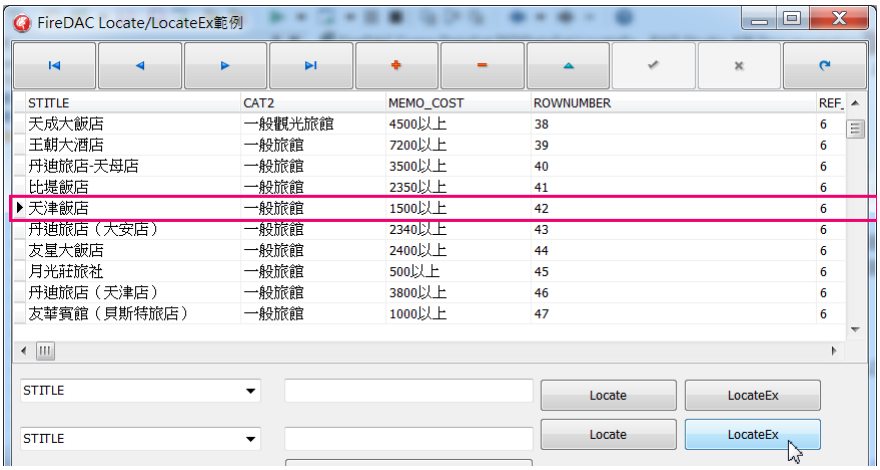


我们也可以使用表达式来搜寻，例如我们如果要搜寻 1500 到 2500 元之间的旅馆就可以使用如下的程序代码来呼叫 LocateEx：

```
procedure TfmMainForm.Button4Click(Sender: TObject);
var
  sExpr : String;
begin
  sExpr := 'MEMO_COST >= ' + '''' + '1500' + '''' + ' and ' +'MEMO_COST
```

```
<= ' + '''' + '2500' + '''';
  if (TbltaipeihotelsTable.LocateEx(sExpr, [lxoPartialKey,
lxoFromCurrent])) then
  begin

DisplaySearchedHotel(TbltaipeihotelsTable.FieldByName('LONGITUDE').As
String, TbltaipeihotelsTable.FieldByName('LATITUDE').AsString);
    edtHotelName.Text :=
TbltaipeihotelsTable.FieldByName('STITLE').AsString;
  end;
end;
```

在上面的程序代码的也使用了[lxoFromCurrent]，因此我们可以持续的搜寻符合 1500 到 2500 元之间旅馆的表达式来搜寻，例如下图就是执行的结果：



FireDAC 的 LocateEx 的确比传统的 Locate 方法多大多了，此外我们也可以结合索引功能增加 LocateEx 搜寻的速度，在稍后的章节会说明。

2-2-2 Lookup 和 LookupEx

Lookup 方法和上一小节介绍的 Locate 方法在使用上非常的类似，它们的差别是当 Locate 找到搜寻的资料后，它会把目前的记录位置移动到找到的这笔资料之上，但是当 Lookup 找到搜寻的数据后，它会回传找到的数据的特定字段数值，却不会移动目前的记录位置。下面即是 Lookup 方法的原型：

```
    function Lookup(const AKeyFields: string; const AKeyValues:
Variant;
      const AResultFields: string): Variant; override;
```

Lookup 方法的第一个参数也是使用者欲搜寻的字段命名，每一个欲搜寻的字段也是使用分号分隔。第二个参数则是欲搜寻的字段数值，如果欲搜寻多个字段，那么这个参数可以是 Variant 数组。Lookup 方法的第一和第二个参数的意义和前面 Locate 方法是一样的。

Lookup 方法的第三个参数则是指定当 Lookup 找到欲搜寻的数据之后，要回传这笔数据的那些字段数值。如果开发人员想要 Lookup 回传多个字段数值，那么每一个字段也是以分号分隔。

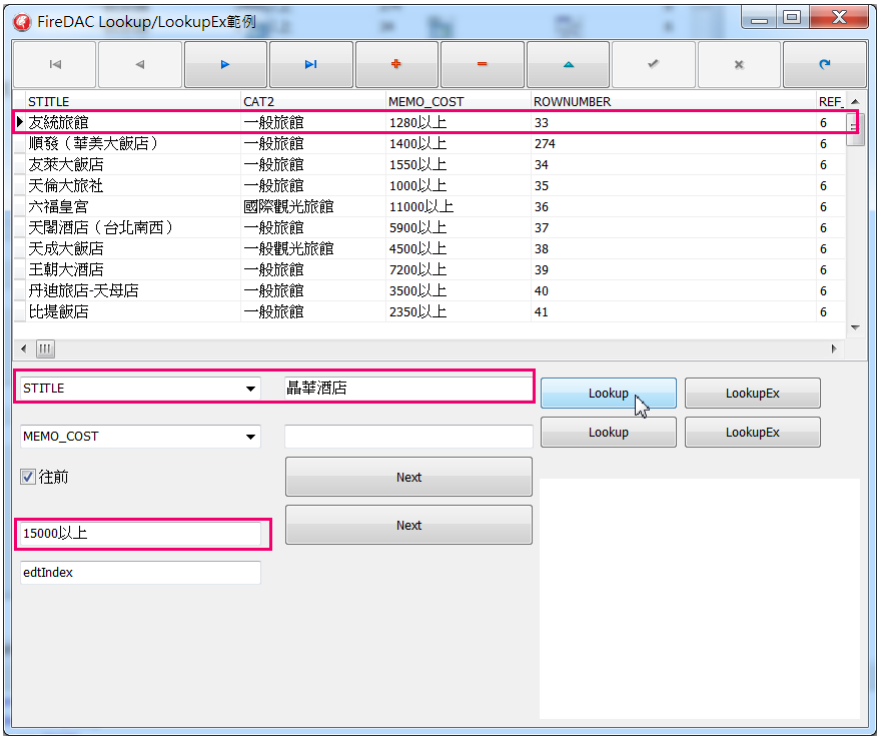
至于 Lookup 方法回传的数值则是第三个字段指定的字段数值，如果 Lookup 回传多个字段的话，那么这个回传数值就是一个 Variant 数组，每一个回传的字段便储存在这个 Variant 数组的元素之中，如果没有找到数据的话回传值就是一个 NULL 的 Variant，现在就让我们看看使用 Lookup 方法搜寻数据。

单字段搜寻

使用单字段搜寻非常的简单，我们只要使用如下的程序代码就可以在 " FireDAC Lookup/LookupEx 范例 " 中根据用户在 ComboBox1 中指定的搜寻字段以及在 ComboBox2 中指定的回传字段进行搜寻。由于只回传一个搜寻结果，因此在 006 行先呼叫 VarIsNull 来判断是否有回传结果，如果有的话就显示出来：


```
001   procedure TfmMainForm.Button1Click(Sender: TObject);
002   var
003     vResult : Variant;
004   begin
005     vResult := TbltaipeihotelsTable.Lookup(ComboBox1.Text,
edtFieldValue.Text, ComboBox2.Text);
006     if (not VarIsNull(vResult)) then
007       edtResult.Text := vResult;
008   end;
```

下图是使用 Lookup 搜寻”晶华酒店”价格的结果，从下图中可以看到 TbltaipeihotelsTable 当前记录位置没有改变，仍然在友统旅馆处，但 Lookup 使用 STITLE 域值搜寻并且要求回传 MEMO_COST 域值，而搜寻结果果然成功显示在 TEdit 中：



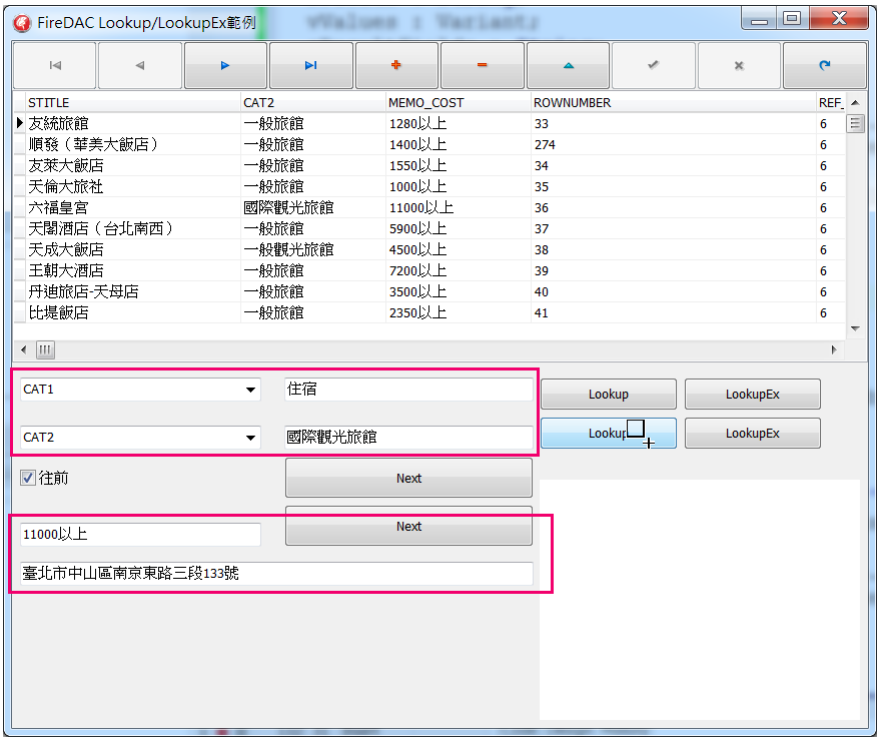
多字段搜寻

要进行多字段搜寻和回传多字段结果，我们需要把多个搜寻字段和回传结果域名以分号结合，再把搜寻域值填入 Variant 数组，最后呼叫 Lookup 方法即可，例如下面的实作程序代码在范例数据库中以 2 个字段搜寻并且要求回传 MEMO_COST 和 ADDRESS 这 2 个域值：

```
procedure TfmMainForm.Button3Click(Sender: TObject);
var
  sFields : String;
  vValues : Variant;
  sResultFields : String;
  vResult : Variant;
begin
  sFields := ComboBox1.Text + ';' + ComboBox2.Text;
  vValues := VarArrayOf([edtFieldValue.Text, edtFieldValue2.Text]);
  sResultFields := 'MEMO_COST;ADDRESS';
  vResult := TbltaipeihotelsTable.Lookup(sFields, vValues,
sResultFields);
  if (not VarIsNull(vResult)) then
  begin
    edtResult.Text := vResult[0];
```

```
    edtIndex.Text := vResult[1];
  end;
end;
```

由于有 2 个回传结果值，因此我们可以从 Lookup 回传的 Variant 数组中一一的取出。下面就是使用 CAT1 和 CAT2 字段搜寻，搜寻成功之后就把 MEMO_COST 和 ADDRESS 这 2 个域值显示在 2 个 TEdit 组件中。



使用 LookupEx

FireDAC 提了供强化的 LookupEx 方法，它同样有 2 个原型版本，第 1 个 LookupEx 和 Lookup 方法很像，只是它接受 5 个参数，它的第 4 和第 5 个参数和前面介绍的 LocateEx 方法的最后 2 个参数是一样的意义：

```
    function LookupEx(const AKeyFields: string; const AKeyValues:
Variant;
      const AResultFields: string; AOptions:
TFDDataSetLocateOptions = [];
      ApRecordIndex: PInteger = nil): Variant; overload; virtual;
```

第 2 个 LookupEx 原型如下，它的第 1 个参数则是使用一个表达式来搜寻：

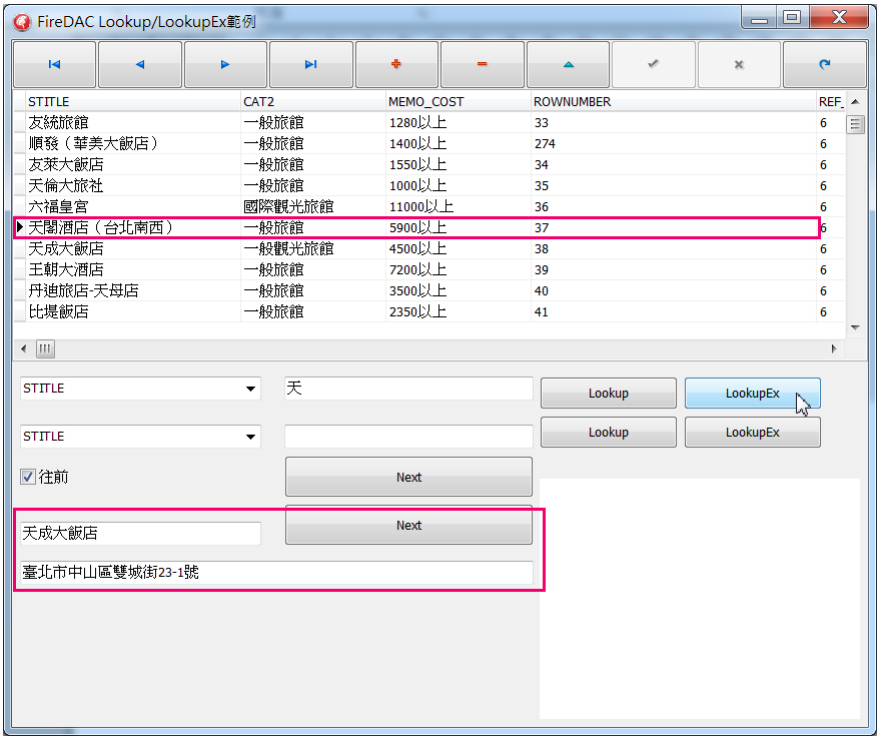
```
    function LookupEx(const AExpression, AResultFields: string;
```

```
    AOptions: TFDDataSetLocateOptions = []; ApRecordIndex:
PInteger = nil): Variant; overload; virtual;
```

例如我们可以使用下面的程序代码来搜寻资料：

```
procedure TfmMainForm.Button2Click(Sender: TObject);
var
  vResult : Variant;
begin
  vResult := TbltaipeihotelsTable.LookupEx(ComboBox1.Text,
edtFieldValue.Text, ComboBox2.Text, [lxoPartialKey, lxoFromCurrent]);
  if (not VarIsNull(vResult)) then
    edtResult.Text := vResult;
end;
```

在下图中请注意 FireDAC 的 Lookup 可以在当前记录”天阁酒店（台北南西）”之后搜寻以”天”为开头的数据并未成功找到”天成大饭店”，这当然是因为上面的程序代码使用了[lxoPartialKey, lxoFromCurrent]搜寻选择值，这是 Lookup 方法无法做到的：

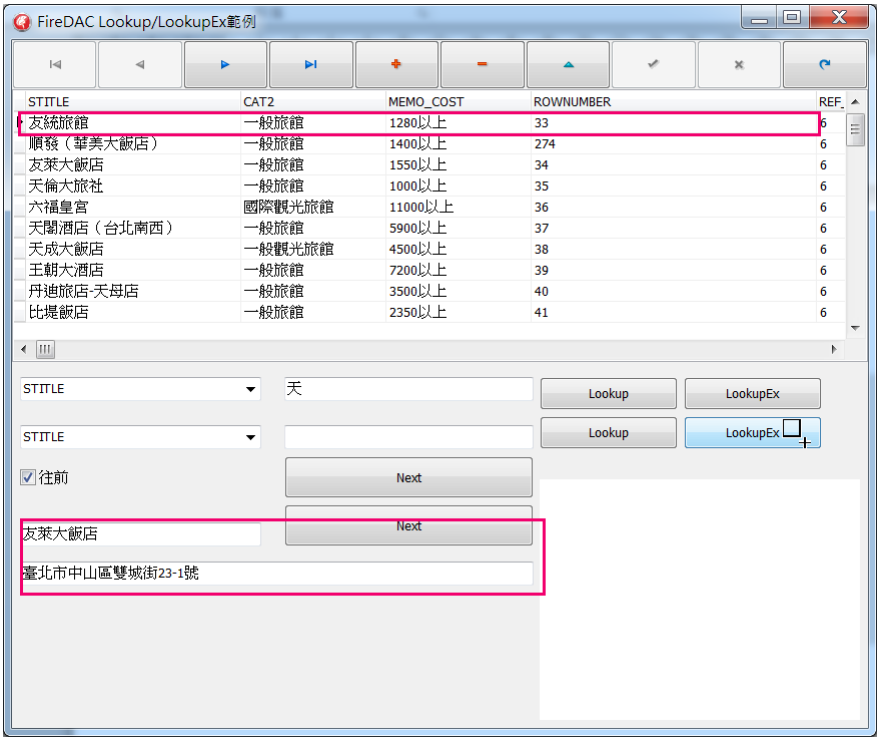


我们也可以使用下面的程序代码以表达式来搜寻：

```
procedure TfmMainForm.Button4Click(Sender: TObject);
var
  sExpr : String;
```

```
  sResultFields : String;
  vResult : Variant;
begin
  sExpr := 'MEMO_COST >= ' + '''' + '1500' + '''' + ' and ' +'MEMO_COST
<= ' + '''' + '2500' + '''';
  sResultFields := 'STITLE;ADDRESS';
  vResult := TbltaipeihotelsTable.LookupEx(sExpr, sResultFields,
[lxoPartialKey, lxoFromCurrent]);
  if (not VarIsNull(vResult)) then
  begin
    edtResult.Text := vResult[0];
    edtIndex.Text := vResult[1];
 end;
end;
```

在下图中可以看到它成功回传了＂友莱大饭店＂是符合表达式条件的第 1 笔资料：

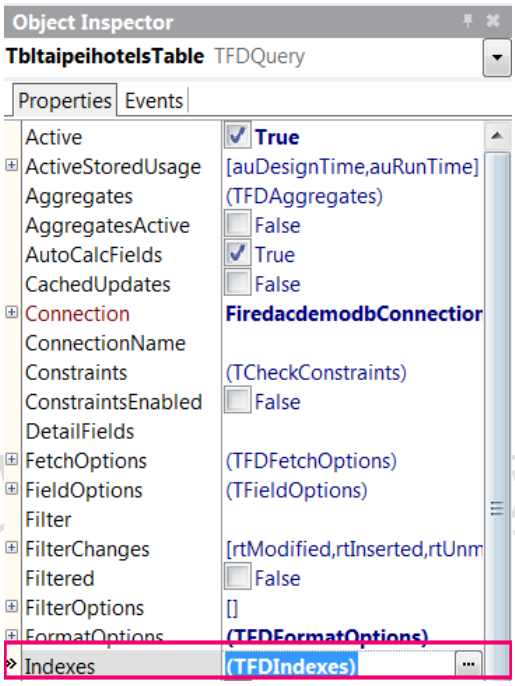


2-2-3 在客户端动态排序

在继续讨论其他搜寻方法之前先让我们讨论一下如何使用 FireDAC 的 TFDQuery 组件在客户端排序数据，因为在使用 FireDAC 搜寻数据时如果搜寻的字段有建立索引的话那么搜寻速度会非常快速。

当程序员使用 SQL 命令从后端取得数据到 TFDQuery 后，这些数据就储存在 TFDQuery 维护的内存中，这也就是所谓的数据集。如果程序员经常需要针对特定域值来搜寻资料，那么程序员可以在这个数据集中暂时建立索引来增加搜寻速度。

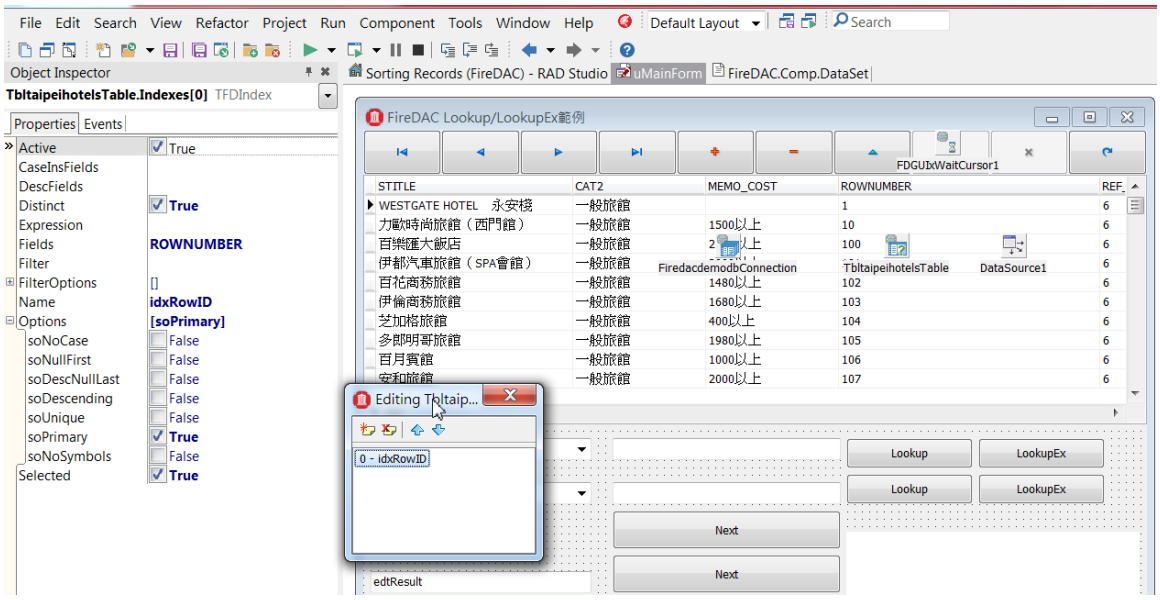
FireDAC 提供 2 种方式可以建立暂时的索引，一是使用 Indexes 特性，一是使用 IndexFieldNames 特性。这 2 种方式都可以快速建立索引并且排序数据集中的资料。例如下图是在 IDE 中双击前面范例的 TbltaipeihotelsTable 组件的 Indexes 特性启动 Indexes 特性值编译程序：



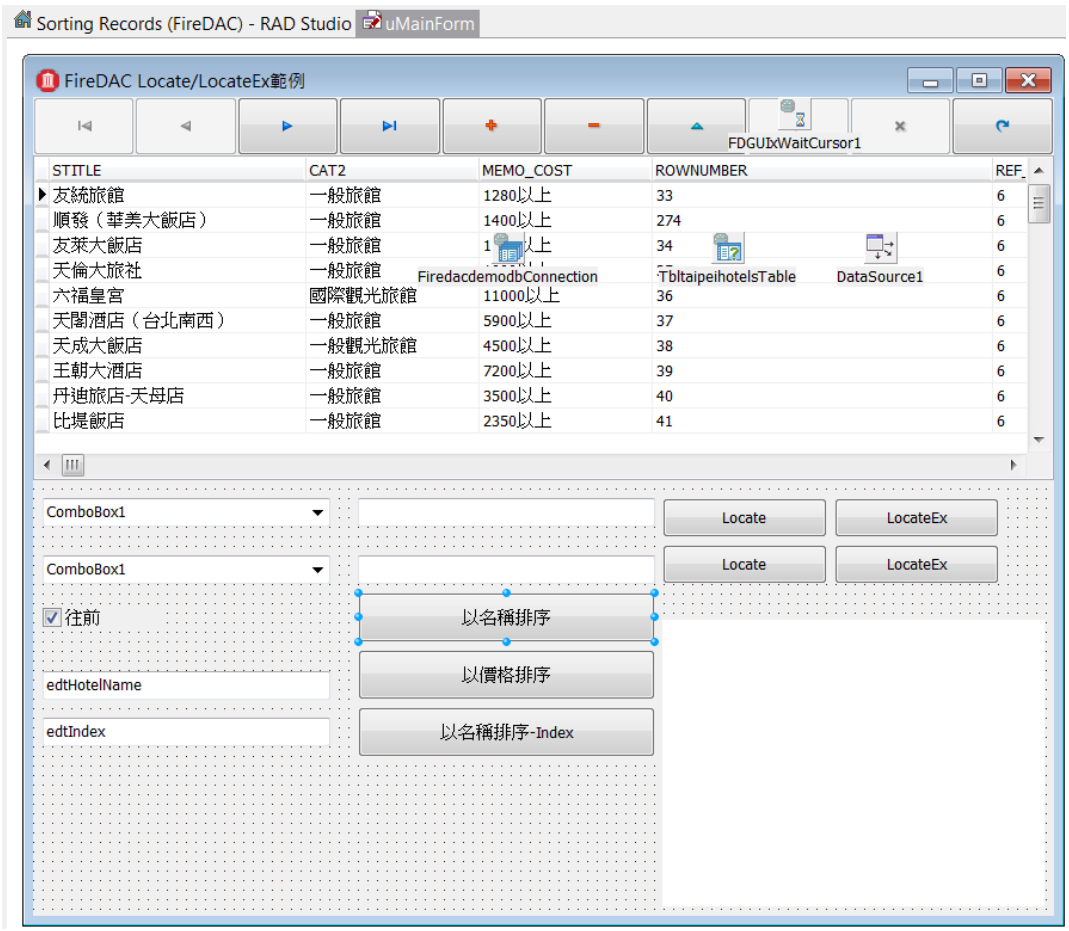
接着在 Indexes 特性值编译程序中建立一个 TFDIndex 对象，再于对象查看器中设定这个 TFDIndex 对象的特性值：

特性	特性值
Name	idxRowNumber
Fields	ROWNUMBER
Active	True
Selected	True

之后就可以在 DBGrid 中看到所有的资料都以 ROWNUMBER 字段自动排序了：



当然我们也可以在程序代码中动态建立索引，先在范例程序的主窗体中加入 3 个按钮如下：



第 1 个按钮”以名称排序”设定 TbltaipeihotelsTable 组件的 IndexFieldNames 特性为 STITLE 字段：

```
procedure TfmMainForm.Button5Click(Sender: TObject);
begin
  TbltaipeihotelsTable.IndexFieldNames := 'STITLE';
end;
```

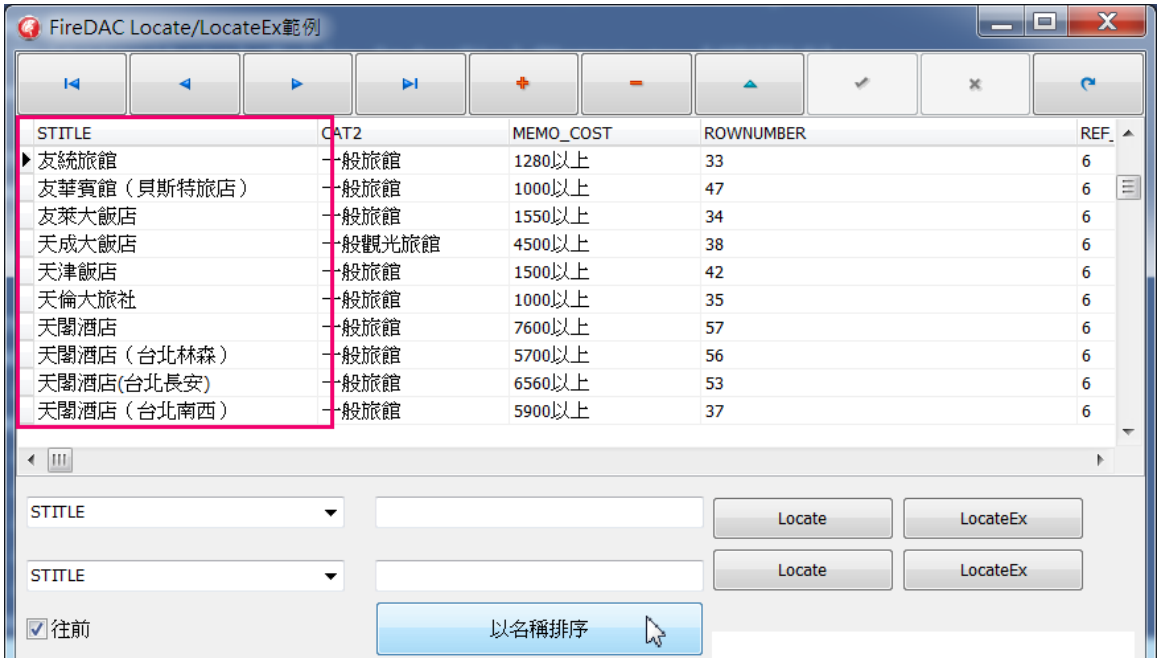
第 2 个按钮” 以价格排序”设定 TbltaipeihotelsTable 组件的 IndexFieldNames 特性为'MEMO_COST 字段：

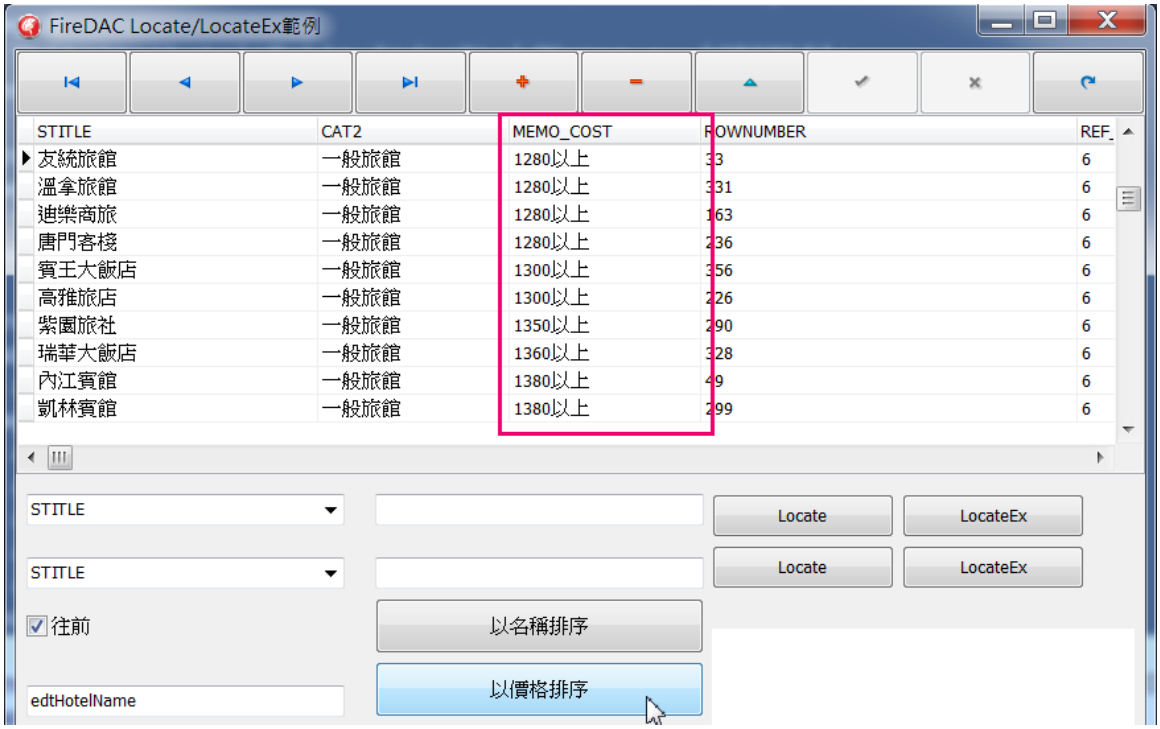
```
procedure TfmMainForm.Button6Click(Sender: TObject);
begin
  TbltaipeihotelsTable.IndexFieldNames := 'MEMO_COST';
end;
```

第 3 个按钮” 以域名-Index”则先建立一个 TFDIndex 对象，再设定它的 Name，Fields，Active 和 Selected 特性值：

```
procedure TfmMainForm.Button7Click(Sender: TObject);
var
  anIndex : TFDIndex;
begin
  anIndex := TbltaipeihotelsTable.Indexes.Add;
  anIndex.Name := 'idxName';
  anIndex.Fields := 'STITLE';
  anIndex.Active := true;
  anIndex.Selected := true;
end;
```

按着执行程序并且点选不同的排序按钮，就可以看到类似下面的结果，客户端数据集中的数据当然自动排序了。





建立动态索引的目的是增快使用 Locate，Lookup 和稍后讨论的过滤器等方法搜寻数据的速度，但是程序员必须了解如果后端数据库的数据很多的话就不适合在客户端建立动态索引，因为 FireDAC 会先把所有的数据取到客户端再建立动态索引和排序数据，这会造成大量数据在网络中传递的问题。如果在这种情形中程序员只想对目前数据集中的数据建立动态索引和排序，搜寻的话，可以使用 FireDAC 的内存数据表组件 TFDMemTable 来配合使用就可以避免把大量资料取到客户端的问题，下一章说明 TFDMemTable 组件时会讨论如何达成。

2-2-4 使用过滤器

除了 Locate 和 Lookup 之外，FireDAC 的过滤器功能也是非常有用的，因为过滤器不但能够搜寻数据，更棒的是它可以再把结果数据集中的数据分门别类，让应用程序只存取特定群组的数据。这个行为有点像是能够把从后端数据库回传的结果数据集中的数据再使用额外的条件来取得子结果数据集。

FireDAC 的过滤器功能不但可以让开发人员对单一字段下达过滤条件，也可以对多个字段下达过滤条件，同时不限定字段是否是索引字段。此外开发人员有两种方法来使用过滤器，第一个方法是使用 TFDQuery 组件的 Filter 特性值，第二个方法是使用 v 组件的 OnFilterRecord 事件处理函式。

这两种方法的差异点是 Filter 特性值只能让开发人员使用字符串来设定过滤条件，而 OnFilterRecord 事件处理函式却能够让开发人员使用 Delphi 程序语言来执行任何复杂的过滤条件，FireDAC 提供和过滤功能相关的特性和事件处理函式如下表所示：

特性/事件处理函式	意义
Filter特性	使用字符串条件来过滤数据
OnFilterRecord事件处理函式	使用事件处理函式程序代码来过滤数据
Filtered	决定是否开启过滤器功能的特性值
FilterOptions	过滤功能使用的额外条件

要使用过滤器功能，开发人员必须设定 Filtered 特性值为 True，一旦 Filtered 特性值为 True 之后，如果 Filter 特性值有任何的过滤条件，FireDAC 便会以这个过滤条件做为标准来过滤目前在结果数据集之中的数据，只有符合过滤条件的数据才能够显示在数据感知组件之中，或是被存取，而不符合过滤条件的数据会暂时的无法被存取到。此外如果开发人员有定义 OnFilterRecord 事件处理函式，那么 FireDAC 也会执行 OnFilterRecord 事件处理函式来过滤数据。因此如果开发人员同时设定了 Filter 特性值以及 OnFilterRecord 事件处理函式，那么这两个过滤条件都会被执行。

至于 FilterOptions 特性则类似 Locate 的第 3 个参数，用来指明在过滤数据时是否需要分别大小写，或是是否需要比对完整的字符串过滤值。

例如如果现在我们要在范例数据表中过滤出如下条件的旅馆：

1. 一般旅馆
2. 价格在 1000 到 3000 元之间

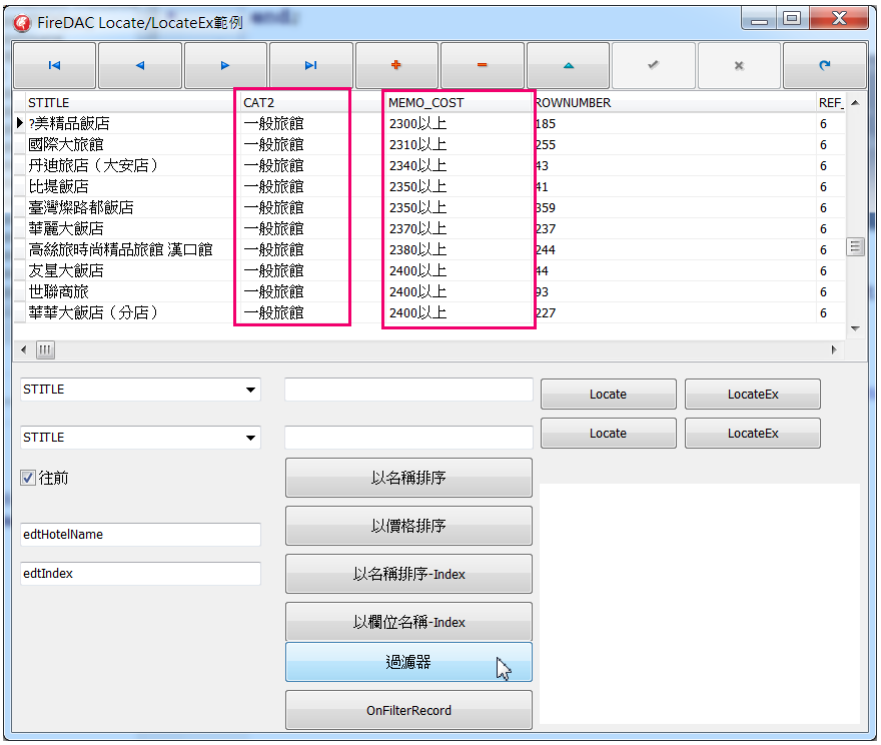
那么我们可以撰写如下的程序代码：

```
001    procedure TfmMainForm.Button8Click(Sender: TObject);
```

```
002    begin
003      TbltaipeihotelsTable.IndexFieldNames := 'CAT2';
004      TbltaipeihotelsTable.IndexFieldNames := ComboBox1.Text;
005      TbltaipeihotelsTable.Filter := 'CAT2=' + '''' + '一般旅馆' + ''''
+ ' and ' + 'MEMO_COST >= ' + '''' + '1000' + '''' + ' and ' + 'MEMO_COST
<= ' + '''' + '3000' + '''';
006      TbltaipeihotelsTable.Filtered := true;
007      TbltaipeihotelsTable.IndexFieldNames := 'MEMO_COST';
008    end;
```

由于上述的 2 个过滤条件的字段是 CAT2 和 MEMO_TEST，因此为了加快过滤速度 003~004 行先为此 2 个字段建立动态索引，005 行在范例数据表组件的 Filter 特性中写入过滤条件，再于 006 行开启过滤功能让 FireDAC 使用过滤条件过滤数据，最后 007 行再使用 MEMO_TEST 字段排序数据。

执行上面的程序代码就可以看到类似下面的执行结果，FireDAC 果然在数据集中过滤出了符合条件的数据：



当然如果要使用 OnFilterRecord 事件处理函式方式过滤资料也可以，下面是先设定建立动态索引再设定范例数据表组件的 Filtered 特性值为 True 以触发 OnFilterRecord 事件处理函式：

```
procedure TfmMainForm.Button10Click(Sender: TObject);
```

```
begin
  TbltaipeihotelsTable.IndexFieldNames := 'CAT2;MEMO_COST';
  TbltaipeihotelsTable.Filtered := True;
end;
```

接着在范例数据表组件的 OnFilterRecord 事件处理函式中撰写如下的程序代码：

```
procedure TfmMainForm.TbltaipeihotelsTableFilterRecord(DataSet:
TDataSet;
  var Accept: Boolean);
begin
  Accept := False;
  if (DataSet.FieldByName('CAT2').AsString = '一般旅馆') then
  begin
    if ( (DataSet.FieldByName('MEMO_COST').AsString >= '1500') and
(DataSet.FieldByName('MEMO_COST').AsString <= '3000') )  then
    Accept := True;
  end;
end;
```


OnFilterRecord 事件处理函式接受 2 个参数，第 1 个参数就是触发过滤的数据集组件，也就是本范例中的 TbltaipeihotelsTable 组件，第 2 个参数 Accept 则代表是否 1 要把第 1 个参数 DataSet 中目前的记录加入符合过滤条件的结果数据集中。因此在上面的程序代码中判断目前的记录的 CAT2 域值是否是'一般旅馆'，以及 MEMO_COST 域值是否在’1000’和’3000’之间，如果是的话就代表符合我们的过滤条件因此就设定 Accept 参数为 True。

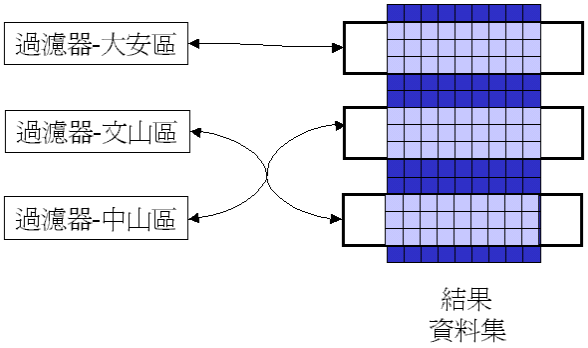
执行这个程序代码就可以得到和上面使用 Filter 特性一样的结果。

使用过滤器的场合

虽然过滤器可以像 TFDQuery 的 Locate 和 Lookup 一样来搜寻数据，但过滤器另外有一个非常特别的用途，那就是它可以在结果数据集中再以过滤条件来取得子结果数据集。这个用途在一些应用中是非常方便的，让我们使用一个例子和下面的图形来说明这个用途。

假设在你的应用程序中已经使用 SQL 命令从后端数据中撷取了数量不多的数据到结果数据集中，例如我使用 SQL 命令搜寻在台北市使用 Delphi 而且购买了本书的开发人员。那么如果我又想根据这些读者以行政区来分析购买的情形，例如以大安区，文山区和中山区等区域，那么我仍然可以再 SQL 命令来存取数据，但是既然这些数据已经存在于结果数据集中，那么我可以直接使用过滤器的功能来取得数据，而不需要再使用许多的

SQL 命令来重新的从后端数据中撷取，这种处理数据的方式的执行效率会比重新使用 SQL 命令来得有效率。



在笔者平常撰写数据库应用程序时，也经常的使用这个技巧从结果数据集中存取笔者需要的数据。由于过滤器的执行效率在结果数据集中数据不多时是不错的，因此开发人员也可以善用这个技巧来处理数据。

2-2-5 使用 SetRange

本章最后一个要说明的搜寻方法就是 SetRange。SetRange 方法可以让开发人员指定索引字段符合一定范围值的资料才可以被存取。藉由 SetRange 开发人员可以设定特定的索引域值来搜寻一笔数据，或是一组在指定范围值之内的数据。由于 SetRange 只能够适用在字段索引之上，因此它的适用性比前面介绍的搜寻方法来得小，不过 SetRange 在搜寻索引字段的数据时，会比其他的方法来得有效率的一点。

使用 SetRange 方法非常的简单，开发人员只需要指定索引范围值的起始数值和结尾数值，FireDAC 便会自动的从结果数据集中搜寻出所有符合范围值的数据。下面便是 SetRange 的函式原型：

```
procedure SetRange(const AStartValues, AEndValues: array of const;
AStartExclusive: Boolean = False; AEndExclusive: Boolean = False);
```

SetRange 的第一个参数 StartValues 便是搜寻范围的起始数值，而 EndValues 参数便是搜寻范围的结尾数值，在使用上非常的简单。由于 SetRange 的 2 个参数都是 array of const 的型态，这代表程序员可以同时传入多个范围值。

在使用 SetRange 之后如果要取得搜寻范围的效果程序员可以呼叫 CancelRange：

```
procedure CancelRange;
```

现在就让我们看看如何来使用 SetRange 来设定搜寻范围，如果我们希望使用 SetRange 来搜寻和刚才使用过滤器范例一样的数据，那么可以使用如下的程序代码：

```
001    procedure TfmMainForm.Button11Click(Sender: TObject);
002    begin
003      TbltaipeihotelsTable.IndexFieldNames := 'CAT2;MEMO_COST';
004      TbltaipeihotelsTable.SetRange(['一般旅馆', '1000'], ['一般旅馆',
'3000']);
005    end;
```

003 行同样先建立动态索引，再于 004 行呼叫 SetRange 方法，同时把'一般旅馆', '1000'这 2 个条件传递给 AStartValues 做为第 1 个参数值，把'一般旅馆', '3000'传递给 AEndValues 做为第 2 个参数值，如此一来就可以得到一样的结果了。

2-2-6 使用 FireDAC 在手机中搜寻数据

虽然前面的范例都是 Windows VCL 的应用程序，但 FireDAC 搜寻数据的功能当然也能够使用在移动平台，例如本书的 " pMobileSearchData " 范例就是在 Android 手机中搜寻旅馆的范例程序，它使用前面说明的过滤器功能以旅馆部份名称来搜寻旅馆，下面是它的搜寻部份的程序代码：


```
procedure TForm4.SearchEditButton1Click(Sender: TObject);
begin
  dmDemoDB.SearchHotel(Edit1.Text);
  LinkFillControlToField1.BindList.FillList;
end;

procedure TdmDemoDB.SearchHotel(const sName: String);
begin
  TbltaipeihotelsTable.Filter := 'STITLE like ' + '''' + sName + '%''';
  TbltaipeihotelsTable.Filtered := True;
end;
```

下面 2 个画面就是此范例程序执行在笔者 S4 手机中的画面，您可以看到只要输入”天”就可以从所有旅馆中搜寻到以”天”开头的旅馆。当然由于 FireMonkey 没有数据感知组件，因此您需要使用 LiveBindings 技术显示数据，请参考本书有关 LiveBindings 技术的章节说明。

2-3 快储机制

在 FireDAC 组件进行数据异动时数据是直接异动到后端的数据库中的，例如要在 FireDAC 数据集组件链接的数据库中新增数据程序员只需要使用 Insert 方法：

```
FDQuery1.Insert;
FDQuery1.FieldByName('域名').Value := 新增域值;
...
FDQuery1.Post;
```

要修改数据可使用 Edit 方法：

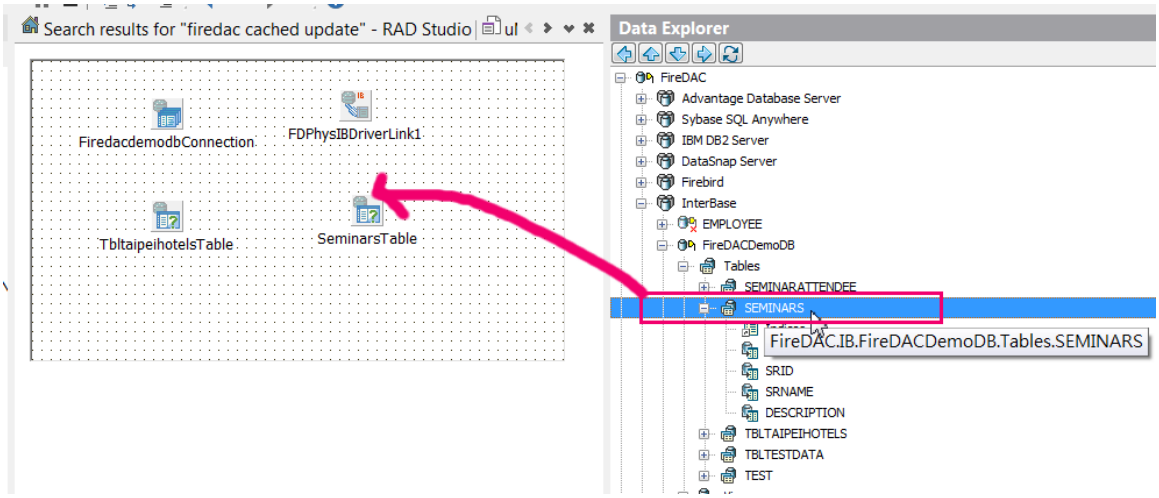
```
FDQuery1.Edit;
FDQuery1.FieldByName('域名').Value := 修改域值;
...
FDQuery1.Post;
```

最后要删除数据只需要使用 Delete 方法：

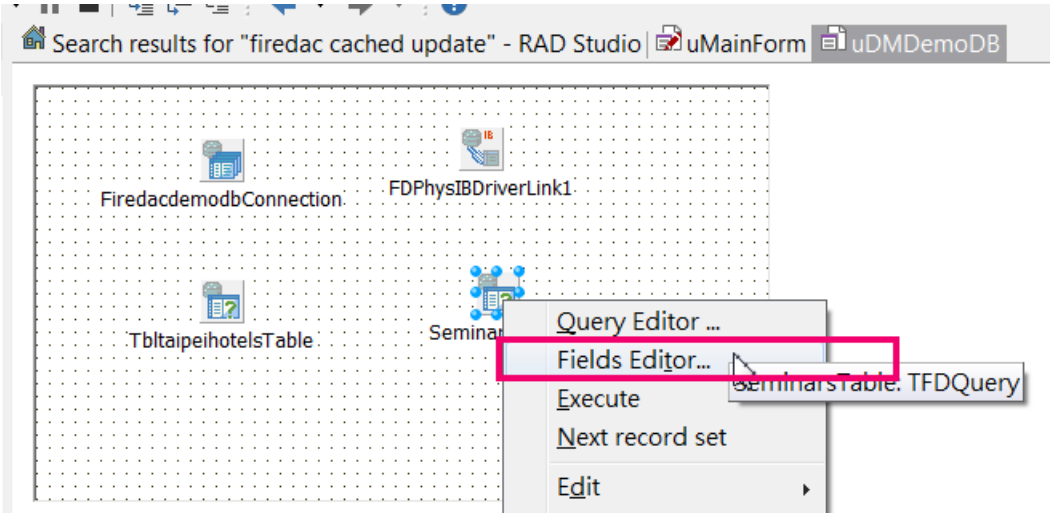
```
FDQuery1.Delete;
```

让我们使用一个简单的范例看看如何使用 FireDAC 在手机中对数据进行 CRUD 的工作。首先建立一个 FireMonkey Mobile 应用程序，在主窗体中加入如下的组件：

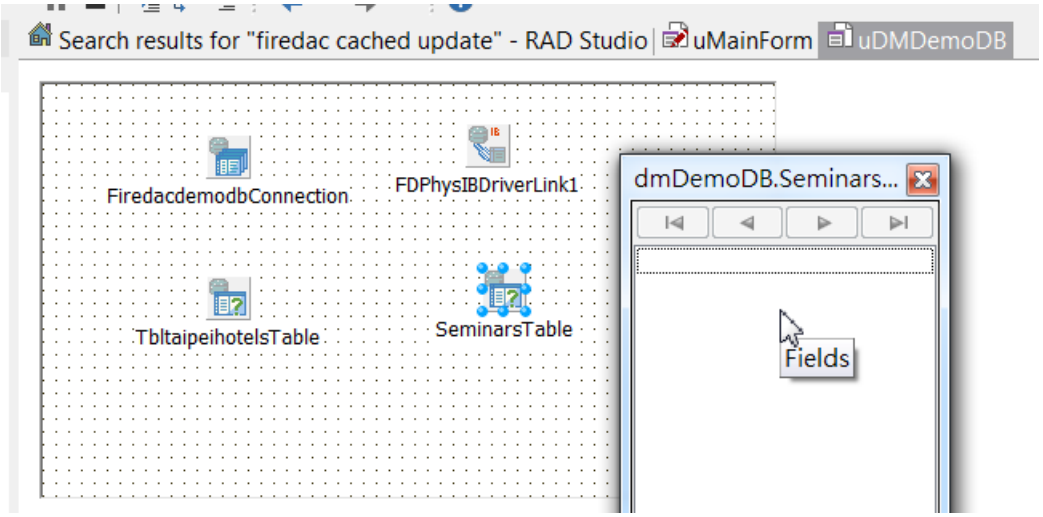
在项目中建立一个数据模块并且使用 FireDAC Explorer 把 SEMIANR 范例数据表拖曳到数据模块中，如下所示：

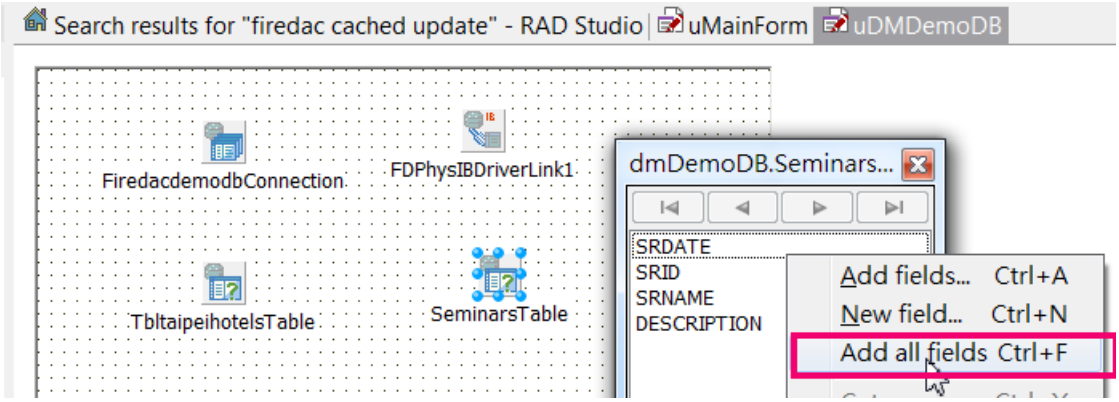


右擊鼠标 SeminarsTable 组件并在选单中选择 Fields Editor...选项：



在字段编辑器中右擊鼠标并选择 Add all fields...加入所有的字段对象：





回到主窗体在  按钮的 OnClick 事件中呼叫数据模块中 SeminarsTable 组件的 Insert 方法在数据表中加入一笔新的数据：

```
procedure TfmMainForm.SpeedButton1Click(Sender: TObject);
begin
  dmDemoDB.SeminarsTable.Insert;
  edtSeminarName.SetFocus;
end;
```

接着在  按钮的 OnClick 事件中呼叫数据模块的 PostSeminarData 方法把用户在主窗体中输入的数据更新回数据表中：

```
procedure TfmMainForm.Button1Click(Sender: TObject);
begin
  dmDemoDB.PostSeminarData(dtedtSeminarDate.Date, edtSeminarName.Text,
mmSeminarNote.Lines.Text);
  LinkFillControlToField1.BindList.FillList;
end;
```


PostSeminarData 方法先检查 SeminarsTable 组件是否在新增或是修改数据的状态中，如果是的话就把传入的输入数据写入相对应的数据表字段中最后呼叫 Post 方法把数据真正更新回后端的数据库中：

```
procedure TdmDemoDB.PostSeminarData(const dtDate : TDateTime; const
sName : String; const sNote : String);
begin
  if (SeminarsTable.State in [dsEdit, dsInsert]) then
  begin
    dmDemoDB.SeminarsTable.FieldByName('SRDATE').Value := dtDate;
    dmDemoDB.SeminarsTable.FieldByName('SRNAME').Value := sName;
    dmDemoDB.SeminarsTable.FieldByName('DESCRIPTION').Value := sNote;
    SeminarsTable.Post;
  end;
end;
```

例如下图就是在 Android 手机中新增数据并且更新数据回数据表的画面：

主窗体中其他的按钮分别实作了修改数据和删除数据，例如  按钮呼叫 SeminarsTable 组件的 Edit 方法让 SeminarsTable 进入修改模式：

```
procedure TfmMainForm.SpeedButton2Click(Sender: TObject);
begin
  EditSeminarData;
end;

procedure TfmMainForm.EditSeminarData;
begin
  SetupSeminarData;
  dmDemoDB.SeminarsTable.Edit;
  edtSeminarName.SetFocus;
end;
```

当然修改完数据之后仍然要呼叫 Post 方法把数据真正更新回后端的数据库中。

最后要删除数据表中目前的记录只需要呼叫 Delete 方法即可：

```
procedure TfmMainForm.SpeedButton3Click(Sender: TObject);
begin
  dmDemoDB.SeminarsTable. 呼叫 Post 方法;
  LinkFillControlToField1.BindList.FillList;
end;
```

藉由 FireDAC 我们可以非常简单的就完成一个能在手机中对数据进行 CRUD 功能的 App，读者可参考本书的 pCRUDDataDemo 范例。

但在这个范例中所有的数据异动在呼叫 Post 方法之后会立刻的更新回后端的数据库中。但在许多应用中您可能不希望每一笔数据在异动之后立刻更新数据库，例如：

1. 在异动大量资料时
2. 在 C/S 架构中
3. 在多层架构中
4. 为了执行效率考虑
5. …

在上面的情形中您可能希望异动的数据先暂时储存在客户端，等到适当的状况时再更新回后端数据库中，例如在异动数据到了一定数量，或是过了一段固定的时间后再更新回数据库中，那么您可以使用 FireDAC 提供的快储功能(Cached Update）。

2-3-1 使用 FireDAC 快储功能

FireDAC 快储功能简单的说就是在客户端暂时把所有对于数据的异动都先记录起来并且把异动的数据先异动在内存的数据集中而不会真接更新回后端的数据库中，一旦程序员决定把异动写回后端的数据库中才一次把所有的异动一次更新回后端。

因此要使用 FireDAC 快储功能，程序代码要进行下列的步骤：

1. 设定 TFDQuery 的 CachedUpdates 特性值为 True
2. 使用 Insert，Edit，Delete 和 Post 方法异动数据
3. 呼叫 ApplyUpadates 方法把客户端的异动一次写回后端数据库中
4. 如果数据写回成功就呼叫 TFDQuery 的 CommitUpdates 方法清除客户端的异动记录，如果发生错误就呼叫 TFDConnection 的 Rollback 方法取消数据写回
5. 处理写回数据发生错误的例外状况

此外 FireDAC 还提供了许多和快储功能相关的特性和方法，它们都是程序员在使用快储功能时经常会使用的，下面的表格针对它们做了简单的说明：

方法/特性	说明
ChangeCount 特性	代表目前在客户端已经被异动的资料笔数

UpdateStatus 特性	代表客户端中每一笔资料的异动状况
UpdatesPending 特性	代表客户端是否已经有异动的资料
SavePoint 特性	储存目前异动数据的版本信息，如此一来程序员可以回到前一个异动状态
RevertRecord 方法	取消对于当前记录的异动
UndoLastChange 方法	回复上一次对于数据集的异动
CancelUpadtes 方法	取消所有数据的异动

上表中的 UpdateStatus 特性代表目前被异动的数据的型态，例是新增数据，修改的数据或是被删除的数据，下面说明了 UpdateStatus 每一个特性值的意义：

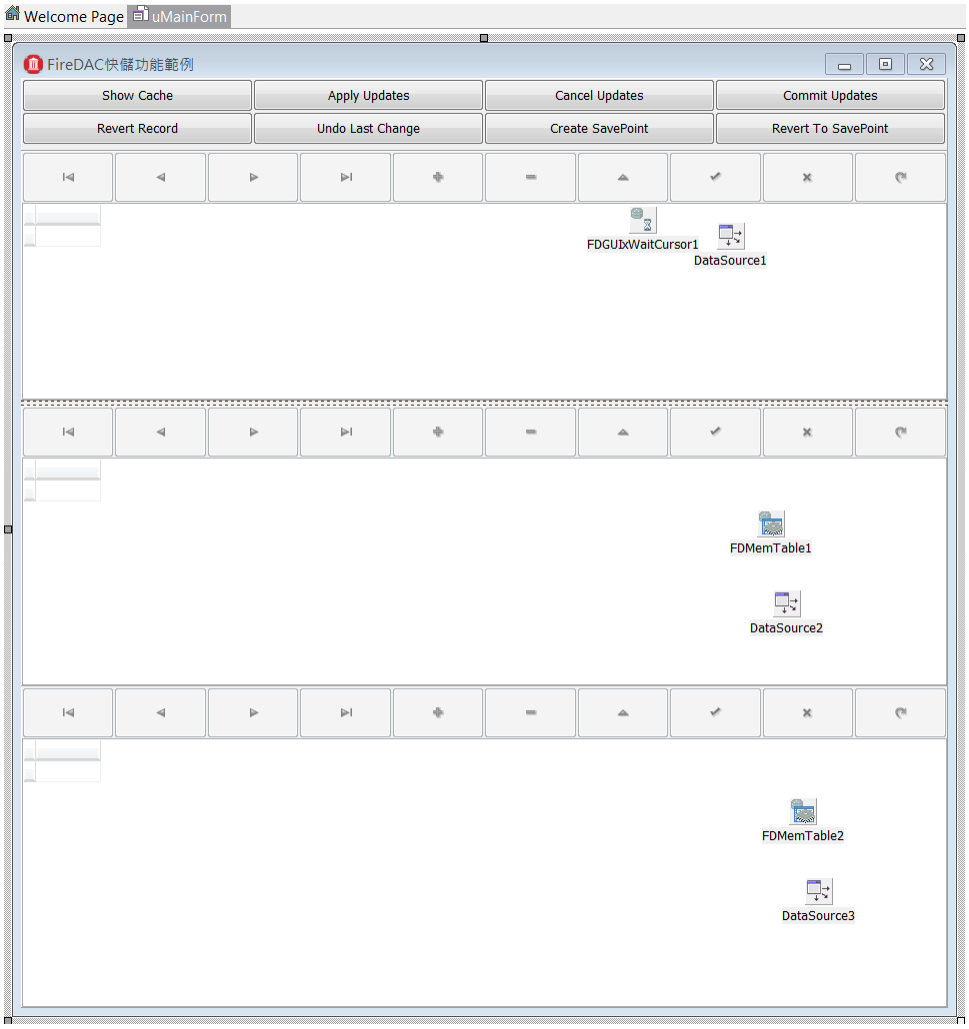
UpdateStatus特性值	说明
usUnmodified	这笔资料没有被异动过
usModified	这笔数据已经被修改过
usInserted	这笔数据是新增的数据
usDeleted	这笔数据已经被删除

在使用 FireDAC 快储功能时程序员也需要知道另外 2 个重要的特性，那就是 TFDQuery 的 Data 和 Delta 特性，下面的表格介绍了这 2 个特性：

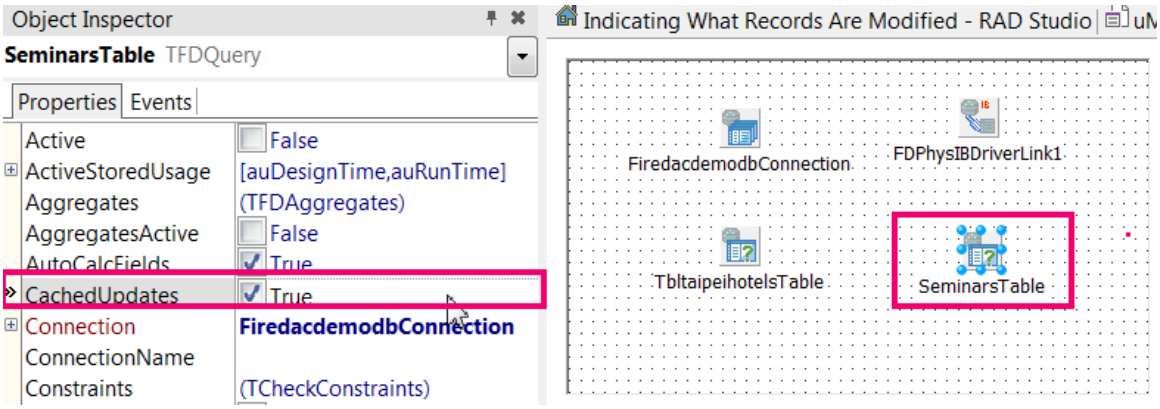
TFDQuery特性	说明
Data	TFDQuery 执行 SQL 命令取得的数据集便储存在 Data 特性中
Delta	用户对于 TFDQuery 的 Data 特性值中的数据进行的异动就储存在 Delta 特性中

藉由上面的方法和特性程序员就可以充分的掌握 FireDAC 快储功能，让我们使用一个范例来说明如何使用 FireDAC 快储功能。

首先建立一个 VCL 应用程序项目并且设定主窗体如下：

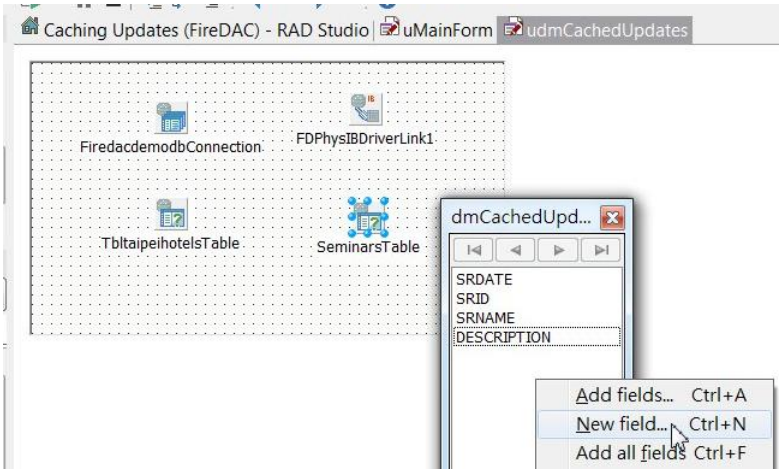


接着建立一个数据模块在其中使用 TFDConnection，TFDQuery 和 TFDPhysIBDriverLink 组件链接到范例数据表 Seminars，当然我们需要设定 SeminarsTable 组件的 CachedUpdates 特性值为 True 以开启快储功能，如下所示：

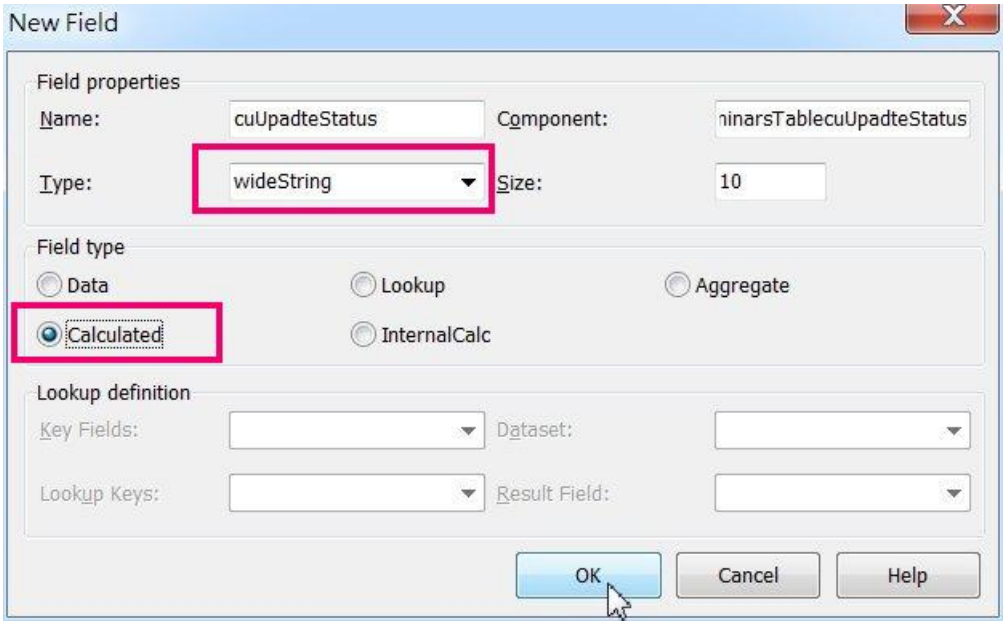


现在让我们在 SeminarsTable 组件中加入一个计算字段以便用来显示每一笔数据的异动状态。所谓＂计算字段＂是指这个字段并不真正储存在数据表中，而是在程序执行

时动态建立的暂时字段。要为 TFDQuery 加入计算字段请使用鼠标右击 SeminarsTable 组件再选择 New field...如下所示：



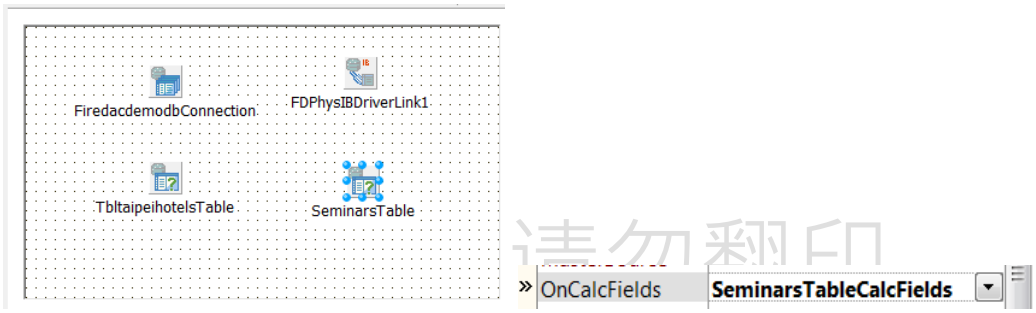
接着在 New Field 对话盒中的 Name 字段输入此计算字段的名称，再选择它的数据型态为 wideString，Field Type 为 Calculated，如下所示：



如此一来我们就在 SeminarsTable 组件中加入了一个名为 SeminarsTablecuUpadteStatus 的计算字段对象了，稍后就可以使用它来显示数据的异动状况了。

首先我们希望此范例程序一开始执行时能够自动显示目前数据的异动状态，例如下图所示。由于现在没有任何数据被异动，因此每笔数据的状态都是＂未异动＂：

由于我们前面已经定义了计算字段，因此请为 SeminarsTable 组件定义 OnCaclFields 事件处理函式：

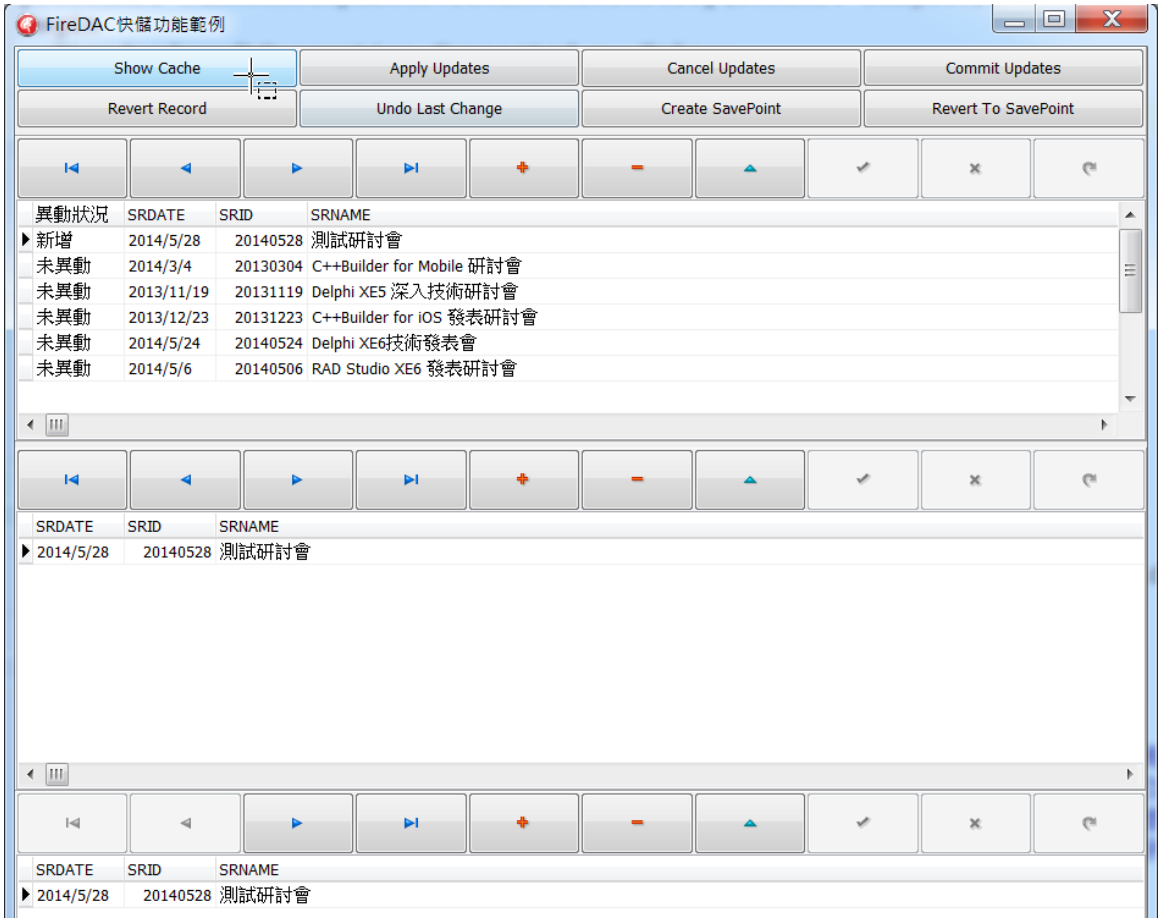


接着在此件处理函式判断 UpdateStatus 特性值并且在计算字段'cuUpadteStatus'中写入相对应的异动状况如下：

```
procedure TdmCachedUpdates.SeminarsTableCalcFields(DataSet: TDataSet);
begin
  case DataSet.UpdateStatus of
   usUnmodified: DataSet.FieldByName('cuUpadteStatus').AsString := '未异动';
    usModified: DataSet.FieldByName('cuUpadteStatus').AsString := '修改过';
    usInserted: DataSet.FieldByName('cuUpadteStatus').AsString := '新增';
     usDeleted: DataSet.FieldByName('cuUpadteStatus').AsString := '删除';
  end;
end;
```

现在如果您执行此范例程序就可以看到类似上面的执行画面了。

接着如果我们在数据表中加入一笔新的数据并且希望能看到快储功能中的异动数据，如下所示，



那么我们可以使用 2 种方式，第 1 种方式是使用过滤功能把 SeminarsTable 组件中被异动的数据过滤出来再显示出来。主窗体中的 Show Cache 按钮就提供这种方式，下面是它的实作程序代码：

```
procedure TfmMainForm.Button3Click(Sender: TObject);
begin
  if (dmCachedUpdates.SeminarsTable.Active) then
  begin
    if FDMemTable1.Active then
      FDMemTable1.Close;
    FDMemTable1.CloneCursor(dmCachedUpdates.SeminarsTable, True);
    FDMemTable1.FilterChanges := [rtModified, rtInserted, rtDeleted];
  end;
end;
```

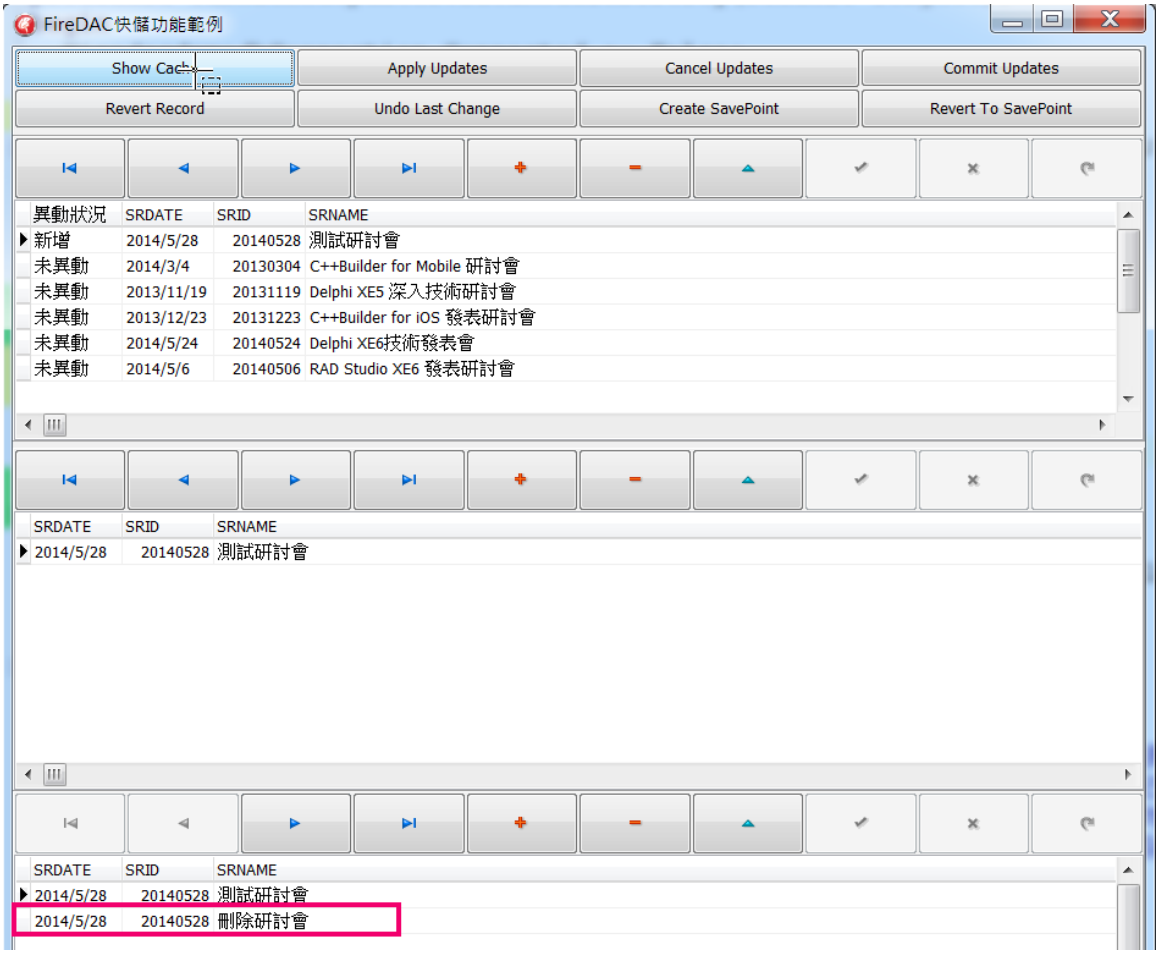
Button3Click 使用了 TFDMemTable 组件 FDMemTable1 的 CloneCursor 方法为 SeminarsTable 中的数据建立另外一个 Data View，再过滤其中被修改，新增和删除的数据并且显示在主窗体中间的 DataGrid 组件中。

第 2 种方式更简单，那就是直接显示 SeminarsTable 的 Delta 特性值，因此我们可以在 SeminarsTable 组件的 OnAfterPost 事件中撰写如下的程序代码：

```
procedure TdmCachedUpdates.SeminarsTableAfterPost(DataSet: TDataSet);
begin
  fmMainForm.FDMemTable2.Active := False;
  fmMainForm.FDMemTable2.Data := SeminarsTable.Delta;
  fmMainForm.FDMemTable2.Active := True;
end;
```

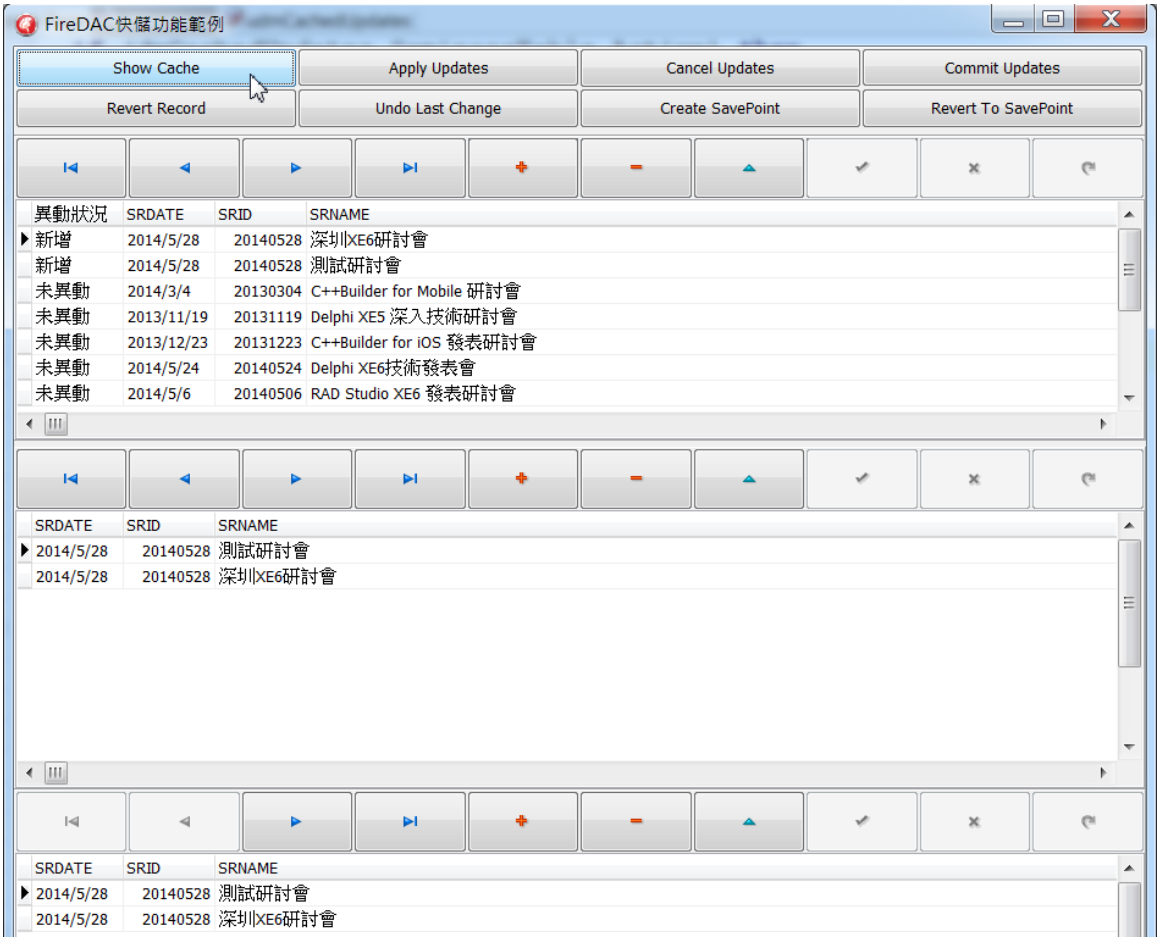
在上面的程序代码中直接把 SeminarsTable 组件的 Delta 特性值指定给 TFDMemTable 组件 FDMemTable2 的 Data 特性值就可以在主窗体下方的 DataGrid 组件中显示新增的数据了。

现在如果我们立刻再删除刚才就增的数据就可以看到如下的画面：



从上面的画面我们可以看到 TFDQuery 组件的 Delta 特性值的确可记录所有的异动数据状态。

如果我们在 SeminarsTable 组件中增加一些数据并且希望真的一次更新回后端的数据表，如下所示：



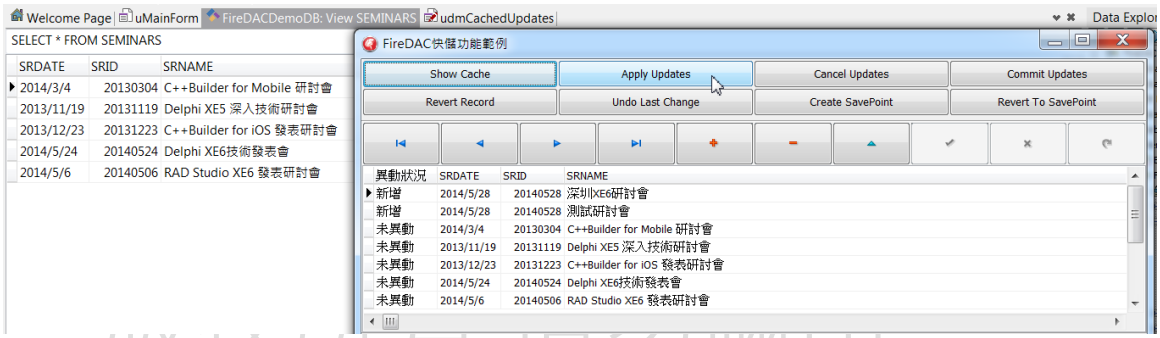
那么可以在主窗体的 " Apply Updates " 按钮中实作如下的程序代码：

```
001   procedure TfmMainForm.Button4Click(Sender: TObject);
002   var
003     Errors: Integer;
004   begin
005     dmCachedUpdates.FiredacdemodbConnection.StartTransaction;
006     Errors := dmCachedUpdates.SeminarsTable.ApplyUpdates(-1);
007     if Errors > 0 then
008       dmCachedUpdates.FiredacdemodbConnection.Rollback
009     else
010       dmCachedUpdates.FiredacdemodbConnection.Commit;
```

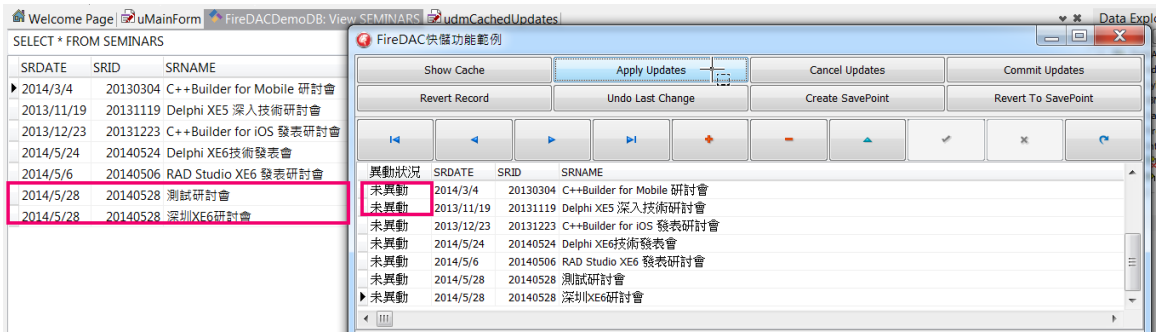
```
011      dmCachedUpdates.SeminarsTable.CommitUpdates;
012      dmCachedUpdates.SeminarsTable.Refresh;
013    end;
```

由于快储功能可能一次更新多笔数据回后端数据库，因此 005 行先呼叫 TFDConnection 组件的 StartTransaction 方法启动 FireDAC 的交易模式，006 行呼叫 ApplyUpdates 方法把所有的异动数据更新回去。ApplyUpdates 方法的-1 参数代表更新的资料不可以发生任何错误，如果发生错误就于 008 行呼叫 Rollback 方法回复更新数据的动作，如果没有发生错误就于 010 行呼叫 Commit 方法确定更新数据的动作。最后 011 行呼叫 CommitUpdates 方法清除快储功能中的数据。

下图同时显示了后端数据表中的数据以及范例应用程序使用快储功能新增 2 笔数据，读者可以看到范例应用程序新增的数据此时并没有真的更新到后端数据表中：



一直到用户点选 " Apply Updates " 按钮后范例应用程序新增的数据才一次更新回端数据表中：



在使用快储功能时程序员也可以使用数个方法来控制异动资料，例如在开始异动数据之前程序员可以先建立一个 SavePoint，如果因为特定原因应用程序需要放弃上次 SavePoint 到现在之间异动的资料，那么程序员可以回到上一次建立的 SavePoint，那么所有的异动都会自动恢复。

SavePoint

主窗体中的『Create SavePoint』按钮就是储存 SavePoint：

```
procedure TfmMainForm.Button9Click(Sender: TObject);
begin
  iSavePoint := dmCachedUpdates.SeminarsTable.SavePoint;
  ShowDeltaData;
end;
```

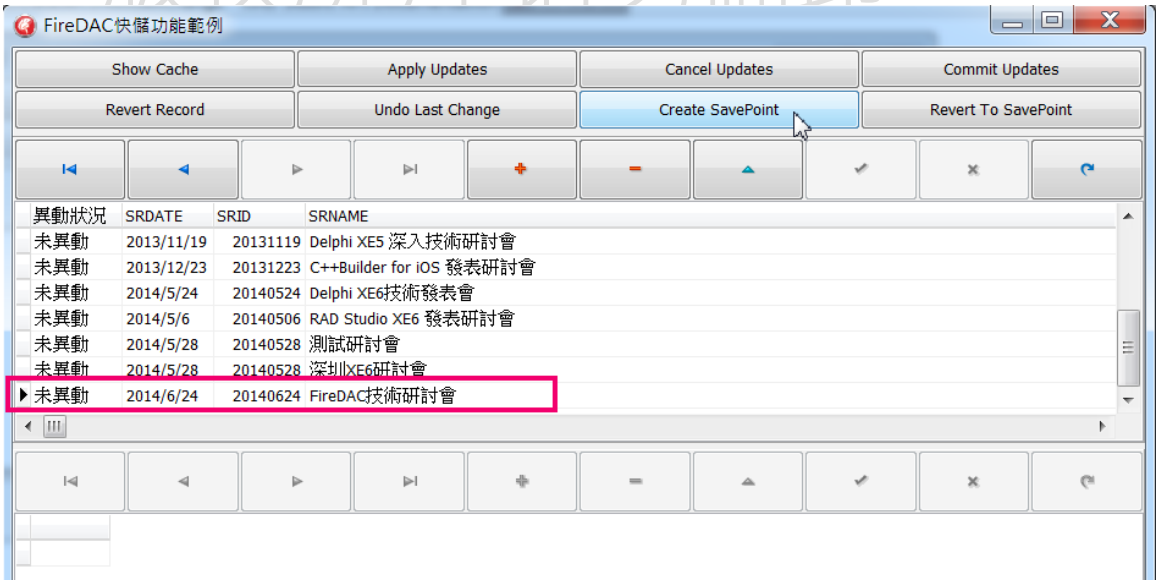
而 iSavePoint 只是型态为 Integer 的变量：

```
  iSavePoint : Integer;
```

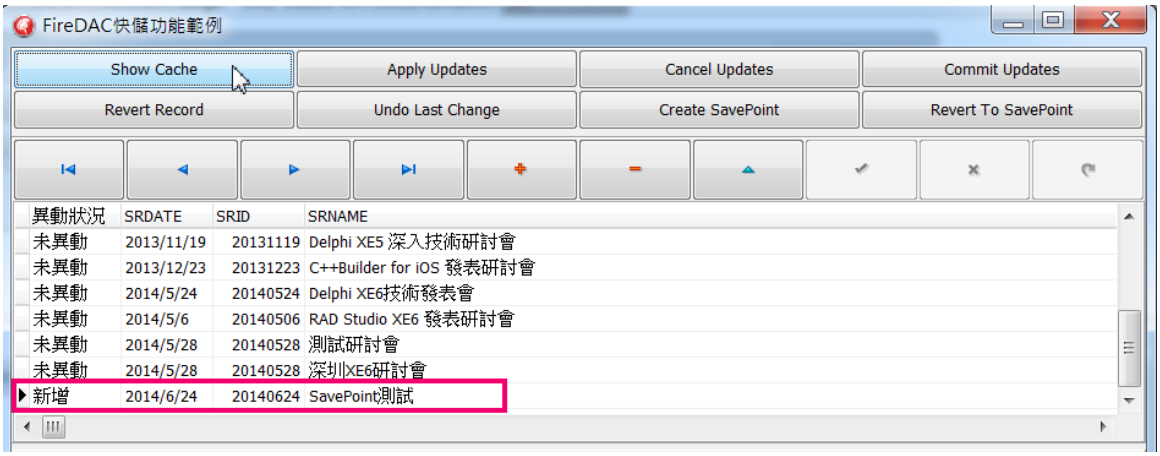
如果要放弃异动，只需要点选主窗体中的『Revert To SavePoint』按钮，它把上代建立储存的 SavePoint 更新回 SeminarsTable 组件的 SavePoint 特性值：

```
procedure TfmMainForm.Button10Click(Sender: TObject);
begin
  dmCachedUpdates.SeminarsTable.SavePoint := iSavePoint;
  ShowDeltaData;
  iSavePoint := 0;
end;
```

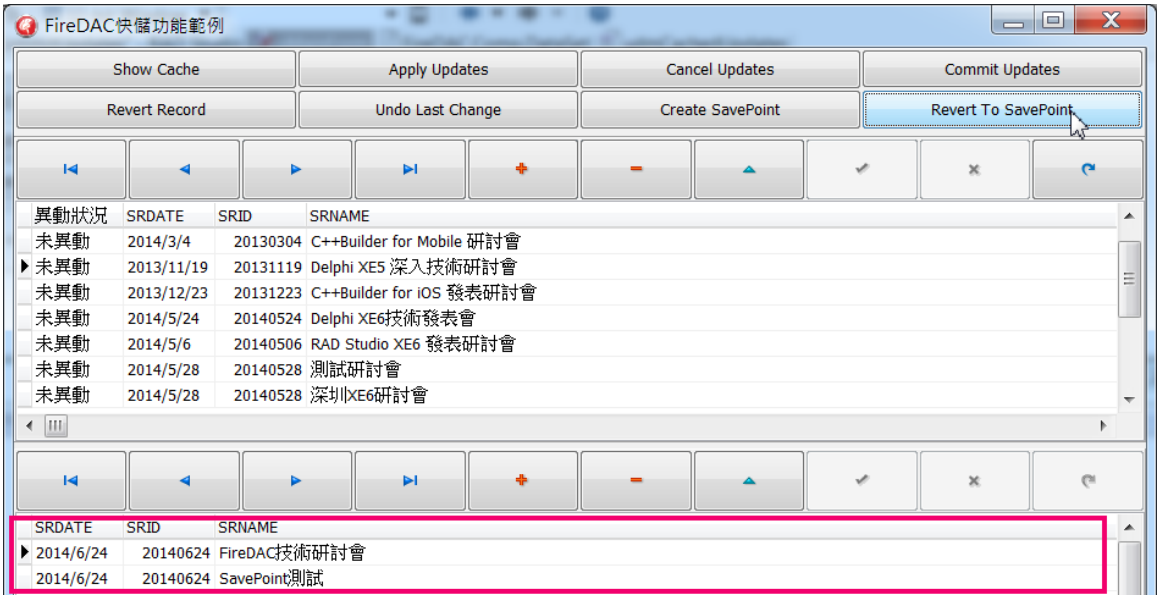
下面的数个画面显示了 SavePoint 方法的功能。下图显示数据从后端数据表中取出并且点选『Create SavePoint』按钮建立 SavePoint：



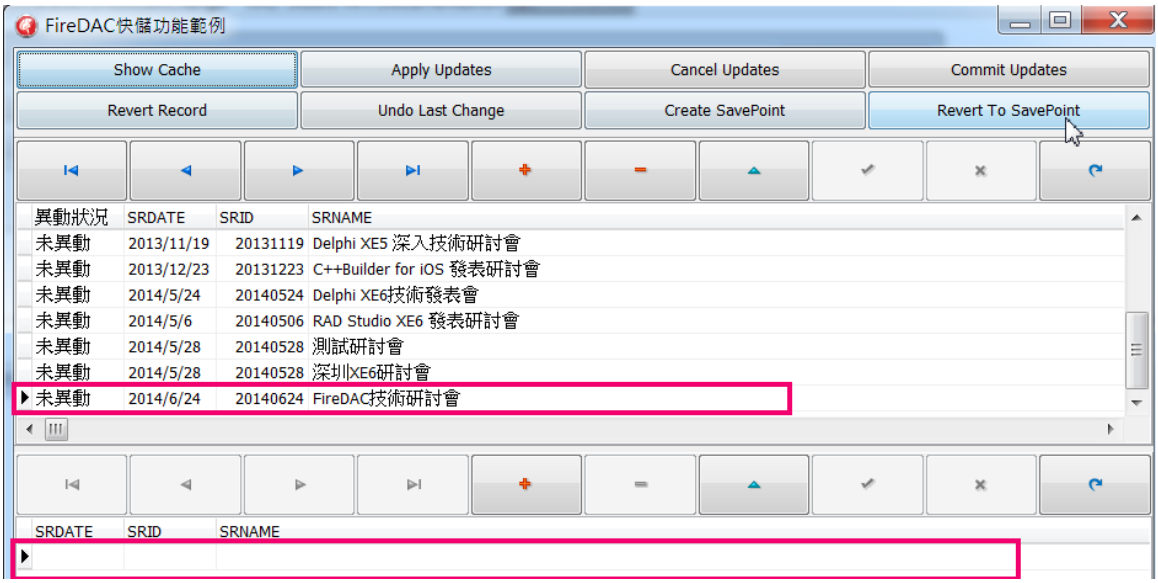
接着在其中加入一笔新的资料：



再删除 FireDAC 技术研讨会这笔资料，此时快储中有 2 笔资料的异动：



现在点选『Revert To SavePoint』按钮，从下图可以看到所有的异动都恢复到一开始异动数据的动作时：

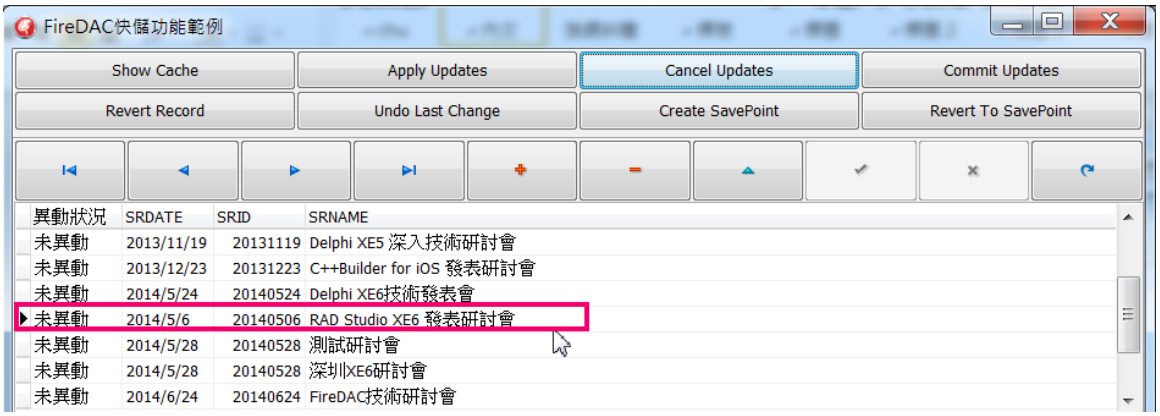


RevertRecord 方法

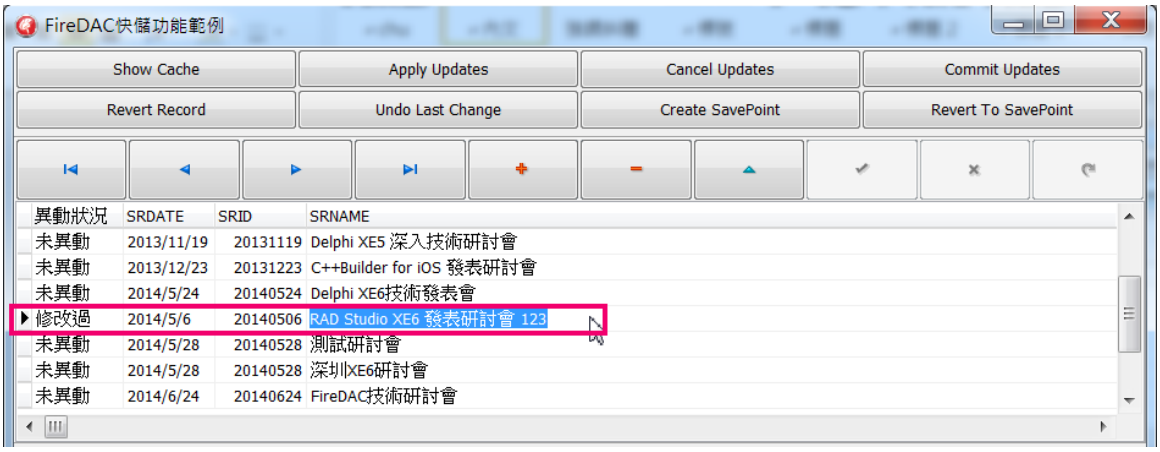
TFDQuery 的 RevertRecord 方法可取消对于当前记录的异动，主窗体中的『Revert Record』按钮实作了下面的程序代码，它先判断目前快储的资料如果不是未异动过的数据就呼叫 RevertRecord 方法回复对数据的异动：

```
procedure TfmMainForm.Button7Click(Sender: TObject);
begin
  if dmCachedUpdates.SeminarsTable.UpdateStatus <> usUnmodified then
    dmCachedUpdates.SeminarsTable.RevertRecord;
  ShowDeltaData;
end;
```

下面的画面显示了 RevertRecord 方法如何工作。例如一开始没有异动 RAD Studio XE6 的数据：



接着修改这笔数据：



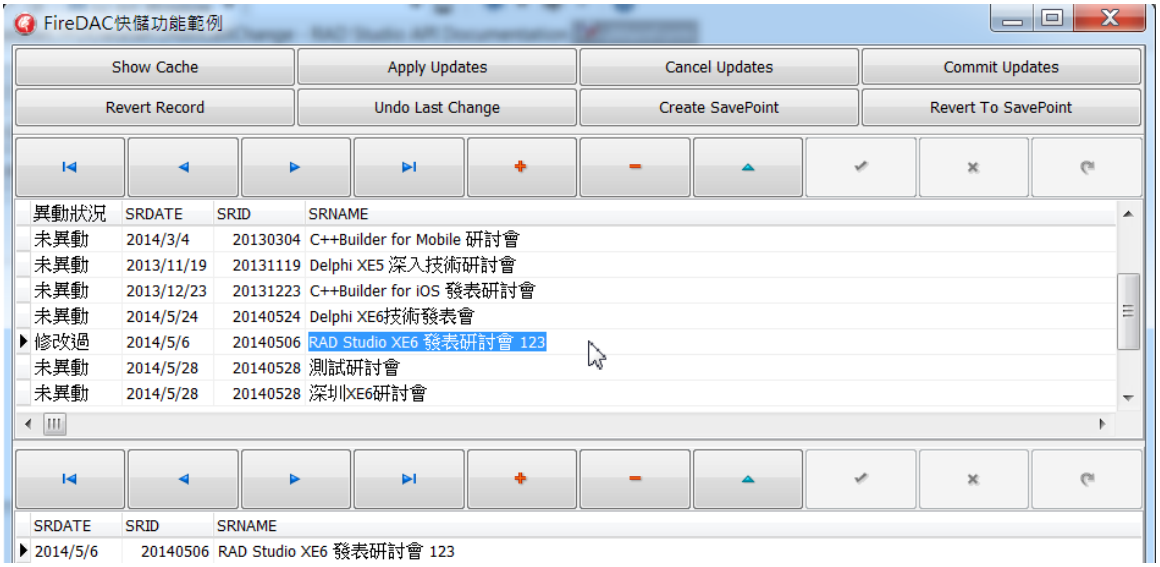
最后如果点选主窗体中的『Revert Record』按钮就可以看到 RAD Studio XE6 回复到原始的状态。

CommitUpdates 方法

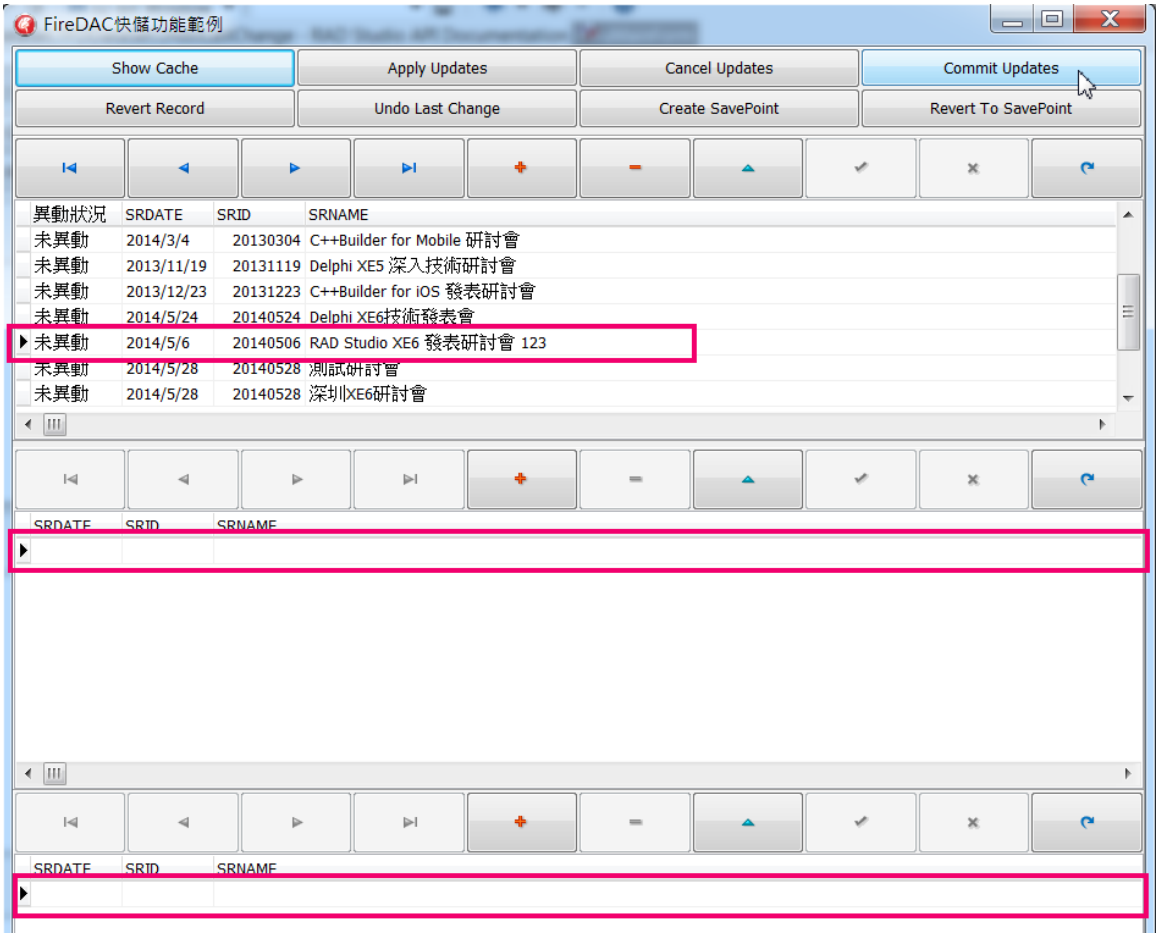
TFDQuery 的 CommitUpdates 方法可以清除所有的快储异动，主窗体中的『Commit Updates』按钮实作了下面的程序代码：

```
procedure TfmMainForm.Button6Click(Sender: TObject);
begin
  dmCachedUpdates.SeminarsTable.CommitUpdates;
  ShowDeltaData;
end;
```

现在让我们修改一笔数据：



再点选『Commit Updates』按钮，从下图我们可以看到快储异动(Delta)已经被清除了：



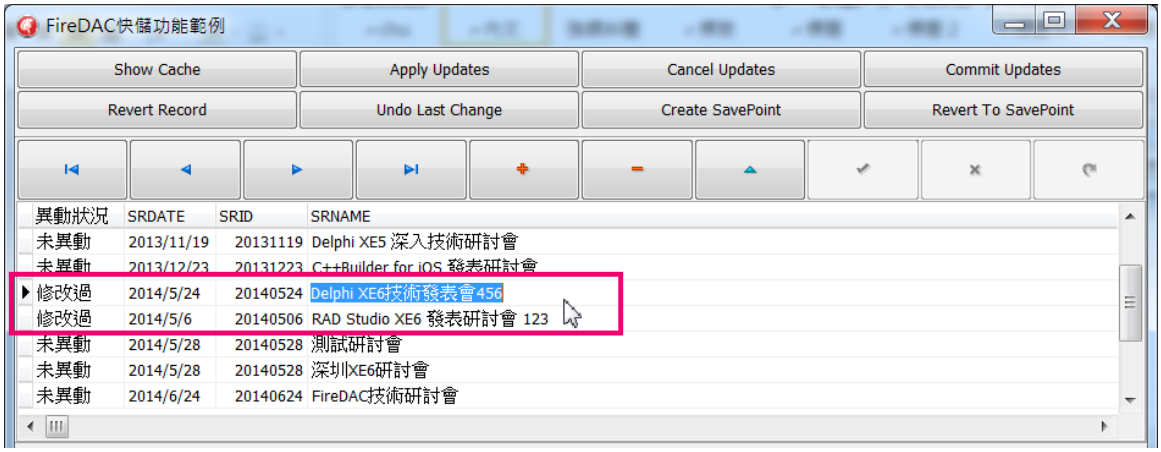
CommitUpdates 只会清除所有的快储异动，但并不会把异动数据更新回后端数据表中。

UndoLastChange 方法

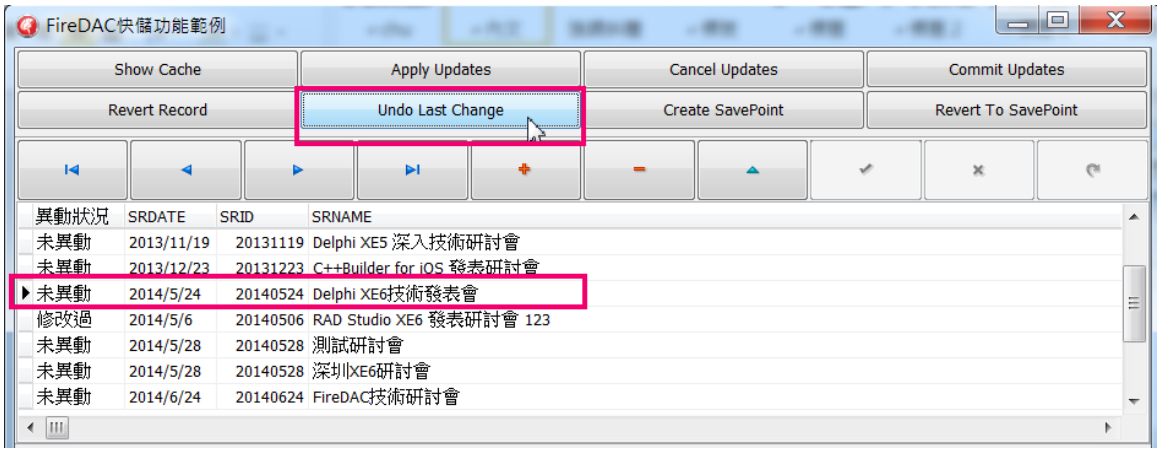
TFDQuery 的 UndoLastChange 方法可以以反相的方向逐一的恢复前一次对于数据的异动，一直到最开始的状态。下面是主窗体中的『Undo Last Change』按钮的实作程序代码

```
procedure TfmMainForm.Button8Click(Sender: TObject);
begin
  dmCachedUpdates.SeminarsTable.UndoLastChange(True);
  ShowDeltaData;
end;
```

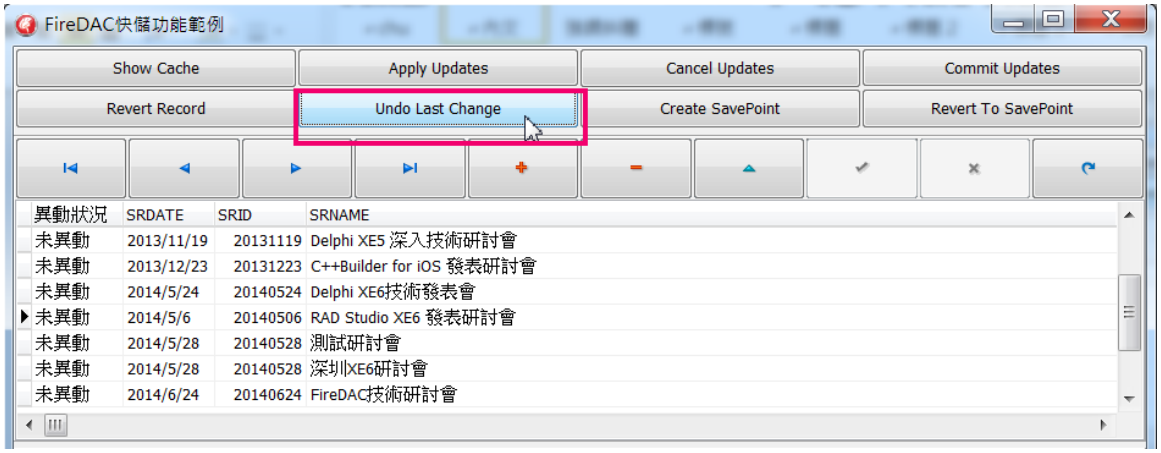
现在让我们修改 2 笔数据，先修改 RAD Studio XE6 再修改 Delphi XE6：



接着点选主窗体中的『Undo Last Change』按钮，从下图可以看到 Delphi XE6 这笔数据被恢复了：



再次点选主窗体中的『Undo Last Change』按钮就可以看到 RAD Studio XE6 的数据也恢复了：



读者可以参考本书的 pCachedUpdatesDemo.dproj 范例。

2-3-2 处理 FireDAC 快储更新错误

在使用 FireDAC 快储功能时，由于在客户端异动的数据是一次更新回后端，因此在更新数据时可能会发生问题，例如主键值冲突，数据字段数据值已经被其他人/其他客户端异动了或是数据已被其他人/其他客户端删除了等状况。这些状况都会造成 ApplyUpdates 方法产生错误，对于无法成功更新回后端的数据，FireDAC 会为每一笔无法更新的数据触发一次 OnReconcileError 事件处理函式，程序员需要在 OnReconcileError 事件处理函式处理这些产生错误的数据。

下面是 OnReconcileError 事件处理函式的宣告原型：

```
procedure TdmCachedUpdates.OnReconcileError(DataSet: TFDDataSet;
  E: EFDException; UpdateKind: TFDDatSRowState;
  var Action: TFDDAptReconcileAction);
begin

end;
```

下面的表格说明了 OnReconcileError 事件处理函式参数的意义：

OnReconcileError事件参数	说明
DataSet	目前进行更新并发生问题的数据集对象
E	例外错误对象，可从其中找出发生错误的原因
UpdateKind	发生错误的异动，其可能的数值是 rsInserted, rsDeleted, rsModified 或 rsUnchanged
Action	此 OnReconcileError 事件处理函式要采取的动作

在 OnReconcileError 事件处理函式中程序员的工作就是处理这个异动错误并且指定适当的值给 Action 参数，下面的表格说明了可指定给 Action 的数值以及其意义：

Action参数值	说明
raSkip	跳过这笔资料
raAbort	离开 Reconcile 方法
raMerge	内定值，清除当前记录的错误信息并且把异动数据合并到 TFDQuery 组件的数据集中
raCorrect	清除当前记录的错误信息并重新设定更新状态，等下次再次呼叫 ApplyUpadtes 方法时再次更新回后端
raCancel	取消当前记录的异动
raRefresh	取消当前记录的异动并重新从后端数据表中取出数据

而当程序员在 OnReconcileError 事件处理函式中处理错误时，可使用 TFDQuery 的下面 2 个特性来取得每一个字段的原始值以及异动过的数值：

TFDQuery特性值	说明
OldValue	从后端取得的原始字段数据值
CurValue	目前的字段资料值
NewValue	目前的字段数据值，可能是经过异动的数据值

例如如果程序员现在使用 FireDAC 快储功能一次新增 10 笔数据，假设其中一笔数据的新增 ID 域值为”201406260945”并且呼叫 ApplyUpdates 把数据更新回后端，但在此之前或同时有另外一个客户端也新增了一笔数据并且使用了相同的新增 ID”201406260945”，那么 FireDAC 快储就会产生键值冲突的错误而触发 OnReconcileError 事件处理函式要求程序员解决，那么程序员可以在 OnReconcileError 事件处理函式中使用如下的程序代码来处理：

```
001   procedure TForm1.FDMemTable1ReconcileError(DataSet: TFDDataSet;
E: EFDException;
002     UpdateKind: TFDDatSRowState; var Action:
TFDDAptReconcileAction);
003   begin
004     if (UpdateKind = rsInserted) and (E is EFDDBEngineException) and
(EFDDBEngineException(E).Kind = ekUKViolated) then begin
005       DataSet.FieldByName('ID').AsInteger := GetNextFreeID;
006       Action := raCorrect;
007     end;
008   end;
```

004 行先判断是否是新增数据更新的错误并且是因为键值冲突，如果是的话就在 005h 行为产生错误的这笔数据产生一个新的 ID，最后 006 行设定 Action 参数为 raCorrect 以便下次再呼叫 ApplyUpdates 方法时把这笔数据更新回后端。

让我们使用一个范例来说明，让我们使用 FireDAC 快储功能先修改数据，但在呼叫 ApplyUpdates 方法更新回后端之前先修改后端数据表中相同的数据，再回头执行 ApplyUpdates 方法，这样做是模拟另外一个使用者已经先修改了相同的数据而会造成 FireDAC 快储功能触发 OnReconcileError。

请在前面的范例项目中加入一个新的窗体，其中 3 个 TEdit 组件分别显示 OldValue，CurValue 和 NewValue 数值，如下所示：

接着为项目中数据模块的 SeminarsTable 组件撰写如下的 OnReconcileError 事件处理函式：

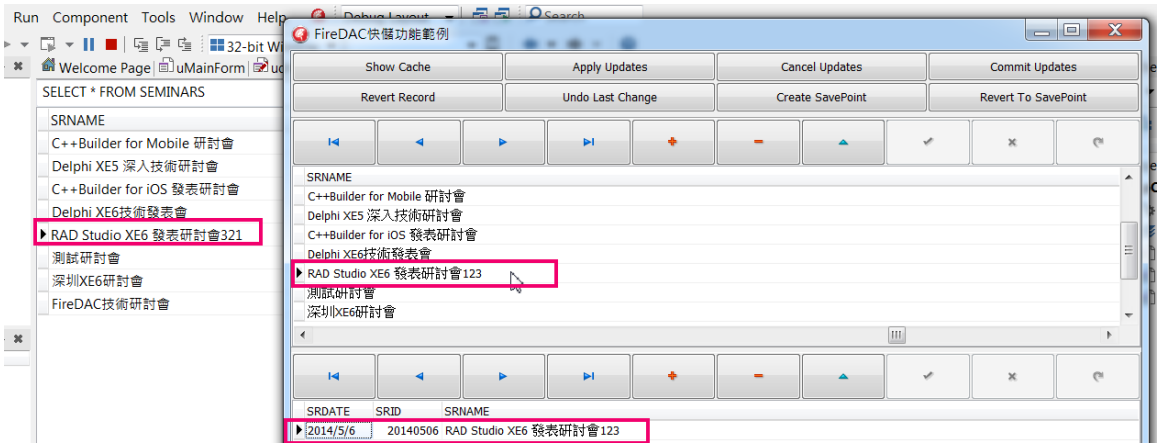
```
001    procedure TdmCachedUpdates.SeminarsTableReconcileError(DataSet:
TFDDataSet;
002      E: EFDException; UpdateKind: TFDDatSRowState;
003      var Action: TFDDAptReconcileAction);
004    begin
005      if (UpdateKind = rsModified) and (E is EFDDBEngineException) and
(EFDDBEngineException(E).Kind = ekNoDataFound) then
006      begin
007        if (TVarData(DataSet.FieldByName('SRNAME').OldValue).VType
<> varNull) then
008          fmUpdateError.edtOldValue.Text :=
DataSet.FieldByName('SRNAME').OldValue;
009        if (TVarData(DataSet.FieldByName('SRNAME').CurValue).VType
<> varNull) then
010          fmUpdateError.edtCurValue.Text :=
DataSet.FieldByName('SRNAME').CurValue;
011        if (TVarData(DataSet.FieldByName('SRNAME').NewValue).VType
<> varNull) then
012          fmUpdateError.edtNewValue.Text :=
DataSet.FieldByName('SRNAME').NewValue;
013        fmUpdateError.ShowModal;
014      end;
015    end;
```

由于我们是仿真 2 个用户修改了相同数据的错误，因此 005 行先判断 UpdateKind 是不是修改更新的错误，如果是的话再判断是不是 ekNoDataFound 的错误，ekNoDataFound 代表 FireDAC 快储要更新的数据已经找不到了，也就是已经被其他客

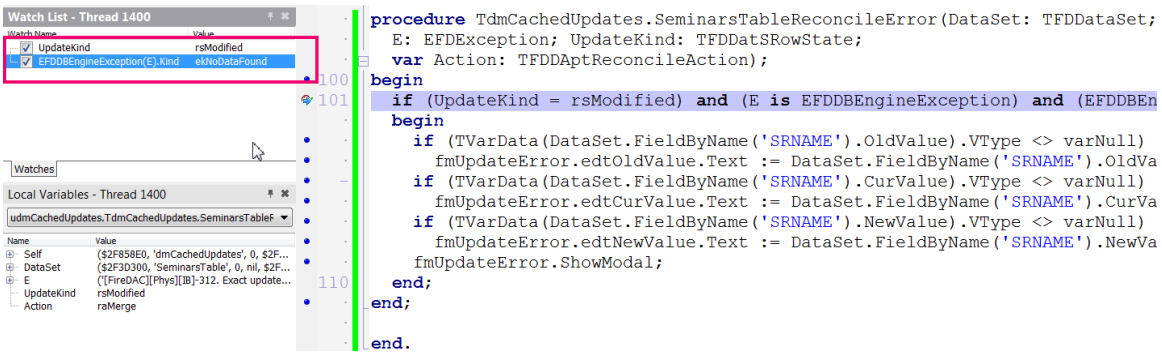
户端先修改了(或是删除了)。如果 2 个条件都符合就把修改字段的 OldValue，CurValue 和 NewValue 值显示出来。

现在执行此范例程序，下面是此时后端数据表中的资料，此时红线包围的资料其 SRNAME 域值是『RAD Studio XE6 发表研讨会』：

接着在范例程序中修改『RAD Studio XE6 发表研讨会』为『RAD Studio XE6 发表研讨会 123』，接着回后端把『RAD Studio XE6 发表研讨会』改为『RAD Studio XE6 发表研讨会 321』，如下所示：

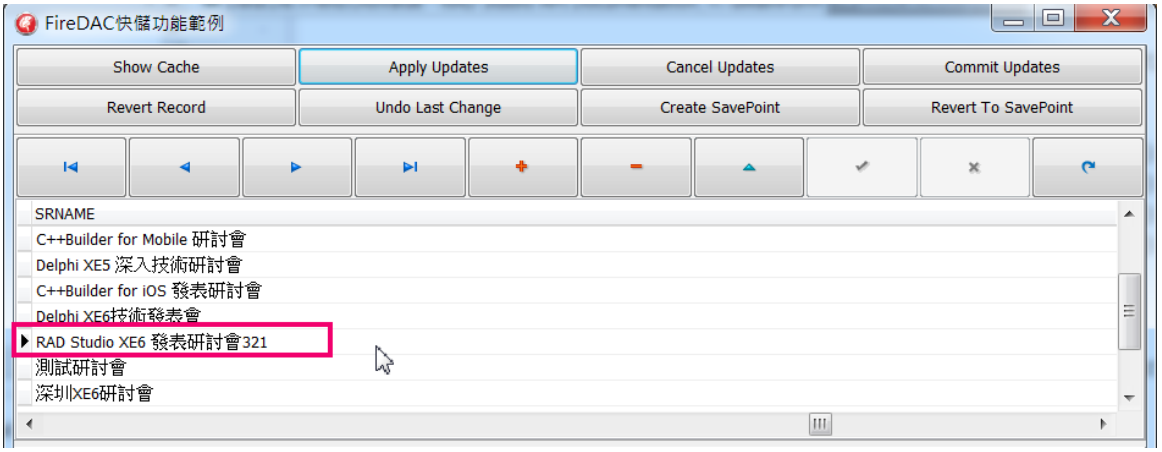


现在点选主窗体中的『Apply Updates』按钮把范例程序中修改的『RAD Studio XE6 发表研讨会 123』更新回后端。由于现在后端的『RAD Studio XE6 发表研讨会』已经被其他客户端改为『RAD Studio XE6 发表研讨会 321』，因此现在 FireDAC 快储试着更新时已经找不到原先的『RAD Studio XE6 发表研讨会』，就会触发 OnReconcileError 事件，从下面的除错画面读者可以看到此时 UpdateKind 的数值的确是 rsModified，而且(EFDDBEngineException(E).Kind 的数值是 ekNoDataFound：



而且新的窗体也显示了 OldValue 和 CurValue 的数值：

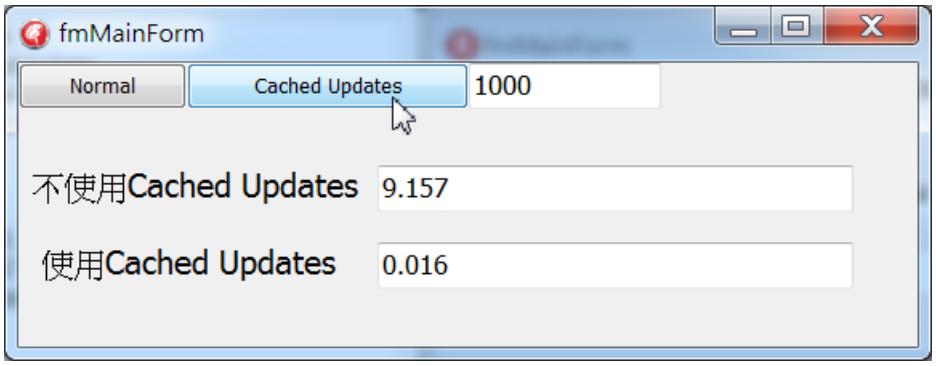
由于在前面的 SeminarsTableReconcileError 中我们没有指定 Action 的数值，因此 SeminarsTableReconcileError 结束时 Action 是被设定为 raMerge，也就是发生错误的域值会被合并更新为后端的最新数值，因此在关闭上面的窗体后范例程序的主窗体会显示如下的结果，SENAME 字段的值会成为『RAD Studio XE6 发表研讨会 321』：



2-3-3 处理 FireDAC 快储执行效率

使用 FireDAC 快储功能的目的就是增加执行效率并且在多个客户端的应用中减少网络的使用流量，只要程序员做好处理 OnReconcileError 事件中更新的错误，一般来

说 FireDAC 快储功能的确能增加不少的执行效率，例如下图就是使用一般更新功能以及使用 FireDAC 快储功能来随机更新数据，从下图可以看到在随机增加 1000 笔资料的情形下 FireDAC 快储功能比使用一般更新功能快上了许多：



2-4 监督数据处理

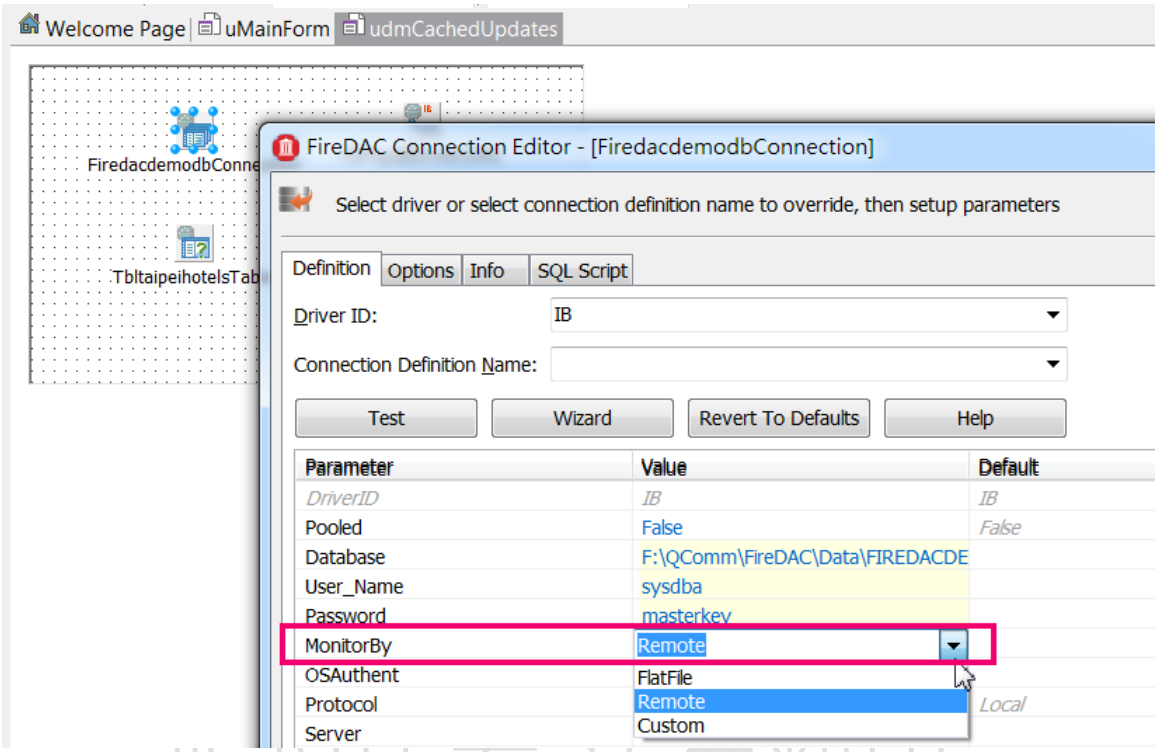
在使用 FireDAC 开发应用程序时，程序员可能需要监督或是观察 FireDAC 如何处理数据，因此 FireDAC 提供了非常完善的方式让程序员可以撷取或是监督/观察 FireDAC 处理数据使用的命令和方法。

FireDAC 可把这些资料储存在文本文件，程序员自行使用的方式或是显示在 FireDAC 的 Monitor Explorer 中，以便帮助程序员掌握数据处理特性。FireDAC 提供了 3 个组件让程序员监督/观察前端 FireDAC 应用程序和后端数据库之间的互动：

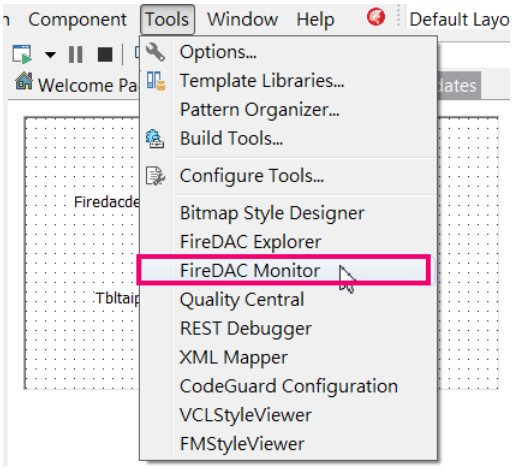
组件	说明
TFDMoniRemoteClientLink	把前端 FireDAC 应用程序和后端数据库之间处理数据的方式显示在 FireDAC Monitor Explorer 工具中
TFDMoniCustomClientLink	把前端 FireDAC 应用程序和后端数据库之间处理数据的方式输出到程序员定案的目标中
TFDMoniFlatFileClientLink	把前端 FireDAC 应用程序和后端数据库之间处理数据的方式输出到文本文件目标中

让我们来说明一下如何使用 TFDMoniRemoteClientLink 组件，因为藉由使用这个组件和 FireDAC Monitor Explorer 工具程序员可以充分掌握 FireDAC 应用程序处理数据的行为。请回到前面的快储范例项目中，现在让我们使用 TFDMoniRemoteClientLink 和 FireDAC Monitor Explorer 工具来观察为什么前面修改『RAD Studio XE6 发表研讨会』为『RAD Studio XE6 发表研讨会 123』时如果其他客户端已经修改了这笔数据的话就会产生 OnReconcileError。

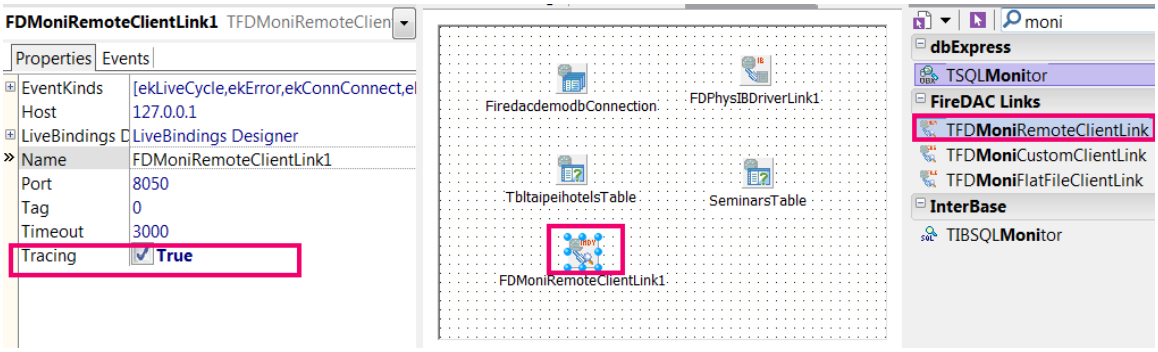
首先开启数据模块中的 FiredacdemodbConnection 组件的组件编辑对话盒，在它的 MonitorBy 选项中选择 Remote，如下所示：



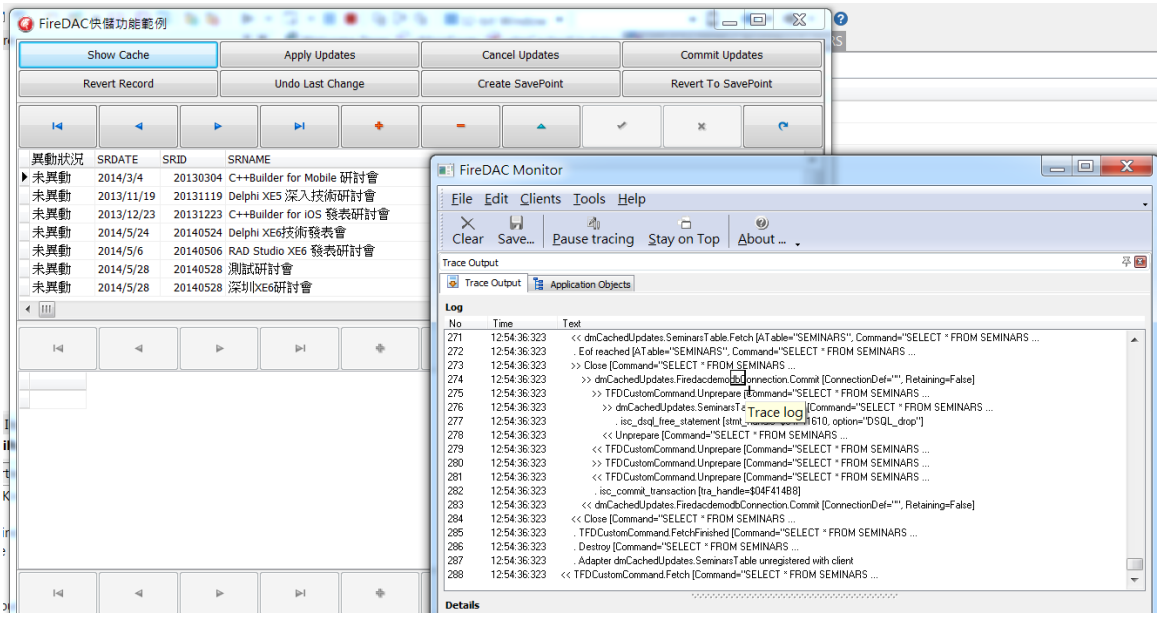
接着可点选 IDE 的 Tools|FireDAC Monitor 选单执行 FireDAC Monitor Explorer 工具：



再于数据模块中放入 TFDMoniRemoteClientLink 组件并且设定它的 Traced 特性值为 True：



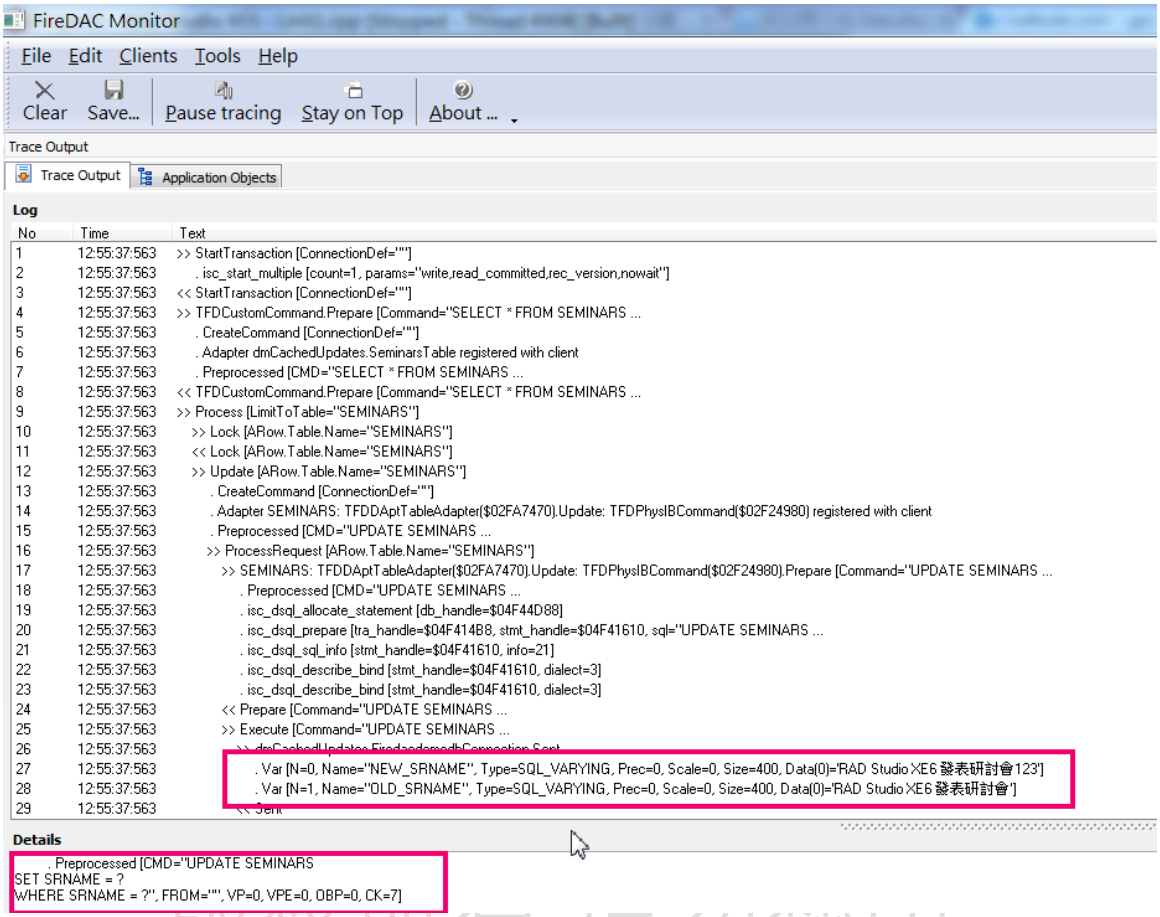
现在再次执行快储范例程序，就可以自到当 FireDAC 应用程序执行时所有 FireDAC 使用来处理数据的流程都会显示在 FireDAC Monitor Explorer 中：



如果我们再次执行前面修改『RAD Studio XE6 发表研讨会』为『RAD Studio XE6 发表研讨会 123』的步骤，就可以在 FireDAC Monitor Explorer 中看到当我们点选主窗体中的『Apply Updates』按钮后 FireDAC 使用如下的 SQL 命令更新数据：

```
"UPDATE SEMINARS
SET SRNAME = ?
WHERE SRNAME = ?", FROM="", VP=0, VPE=0, OBP=0, CK=7
```

而上面的 2 个 SRNAM 就是『RAD Studio XE6 发表研讨会』和『RAD Studio XE6 发表研讨会 123』，从下面的 FireDAC Monitor Explorer 也可以看到：



因此上面的 SQL 命令应该是：

```
"UPDATE SEMINARS
SET SRNAME = "『RAD Studio XE6 发表研讨会 123』"
WHERE SRNAME = "『RAD Studio XE6 发表研讨会』"", FROM="", VP=0, VPE=0,
OBP=0, CK=
```

而此时由于后端的 SRNAME 域值已经是『RAD Studio XE6 发表研讨会 321』了，因此找不到 SRNAME = "『RAD Studio XE6 发表研讨会』"的数据，因此发生错误 FireDAC 再触发 OnReconcileError 事件，而且错误原因是 ekNoDataFound，也就是找不到资料的错误，这一切从 FireDAC Monitor Explorer 中也可以看的一清二楚。

2-5 在移动平台使用快储功能

一般来说 FireDAC 的快储功能适合使用在多客户端的应用中，例如 C/S，多层和 Web 系统中，但同样的功能也可以使用在移动平台中。例如下图是本书 pCachedUpdateApp 范例在 Android 手机中使用快储功能执行的画面：

当使用者在第 1 个页面中点选一笔数据之后就可以在第 2 个页面修改数据，上图右方的 2 就代表现在在手机 App 中有 2 笔数据被异动了，最后点选上方最右边的按钮呼叫 ApplyUpdates 方法后就可以把数据都异动回手机的 InterBase 数据库了。

虽然手机也可以使用 FireDAC 的快储功能，但由于手机只有一个客户端在使用，因此就不一定要使用 FireDAC 的快储功能，除非您的手机 App 要连结中介服务器形成 MEAP 的架构，那就另当别论了。

2-5 结论

本章的内容主要是说明如何使用 FireDAC 处理资料，在稍后的章节中会讨论更多使用其他 FireDAC 组件处理数据的技巧。

第3章 使用内存数据组件

在 FireDAC 组件组中提供了 TFDMemTable，TFDMemTable 组件的功能是在内存中快速建立数据集对象的封装组件，它类似 dbExpress 中的 TClientDataSet 组件，但 TFDMemTable 的执行速度比 TClientDataSet 快而且 TFDMemTable 也提供了许多 TClientDataSet 没有的功能。在本章中将介绍如何使用 TFDMemTable 组件，因为 TFDMemTable 不单可使用在一般的数据库应用程序中，也是最适合使用在移动平台和多层架构中的组件。

3-1 使用 TFDMemTable

TFDMemTable 组件  是 FireDAC 框架的内存数据集组件，也是 FireDAC 框架中处理数据最快速的组件。简单的说 TFDMemTable 组件是把数据快储在内存中进行处理，因此 TFDMemTable 组件中的数据基本上是和后端的数据源是隔离的。

TFDMemTable 组件一般是使用在下面的场景中：

1. 把一些少量但经常会使用的数据放在 TFDMemTable 组件中，可提供最快速的数据处理速度，例如邮政编码查询，产品查询等。
2. 使用 SOAP/REST 取得的数据放在 TFDMemTable 组件中可提供最佳的速度。
3. 使用 TFDQuery 组件取得的数据再拷贝到 TFDMemTable 组件中进行处理。
4. 频繁异动的数据可暂时快储在 TFDMemTable 组件中，等待所有异动完成之后再一次更新回后端。

3-1-1 使用 TFDMemTable 组件提供快速查询

TFDMemTable 组件的 CreateDataSet 方法可以让程序员在内存中建立任何架构的数据集对象然后在其中处理数据，因此可提供最快速的数据处理速度。现在假设我们需要在一个手机 App 中开发一个邮政编码查询的功能，那么我们就可以使用 TFDMemTable 组件。

下面的程序代码是一个 Multi-Device Application 主窗体的 OnCreate 事件处理函式它先呼叫 CreateZipCodeTable 方法 TFDMemTable 组件在内存中建立邮政编码数据表，再呼叫 FillZipCodeData 方法显示所建立的邮政编码数据：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  CreateZipCodeTable;
  FillZipCodeData;
end;
```

CreateZipCodeTable 方法在 007 行先存取 FieldDefs 特性取得它的包含的字段对象，然后在 009 行呼叫 AddFieldDef 方法在 TFDMemTable 组件中建立 TFieldDef 字段对象，并且设定字段对象的名称和数据型态，于 019 行存取 IndexDefs 特性取得它的包含的索引对象，呼叫 AddIndexDef 方法在 TFDMemTable 组件中建立索引对象，最后在 023 行呼叫 CreateDataSet 方法完成在 TFDMemTable 组件中建立 2 个字段和一个索引的工作。


```
001   procedure TForm1.CreateZipCodeTable;
002   var
003     Defs : TFieldDefs;
004     aField : TFieldDef;
005     anIndex : TIndexDef;
006   begin
007      Defs := fdmtZipCodes.FieldDefs;
008
009      aField := Defs.AddFieldDef;
010      aField.DataType := ftInteger;
011      //aField.Size := 5;
012      aField.Name := '邮政编码';
013
014      aField := Defs.AddFieldDef;
015      aField.DataType := ftWideString;
```

```
016      aField.Size := 30;
017      aField.Name := '区名';
018
019      anIndex := fdmtZipCodes.IndexDefs.AddIndexDef;
020      anIndex.Fields := '邮政编码';
021      anIndex.Name := 'pnIndex';
022
023      fdmtZipCodes.CreateDataSet;
024    end;
```

接着就可以在现在 fdmtZipCodes 中新增邮政编码数据了 FillZipCodeData 方法就呼叫 Insert 和 Post 方法在 fdmtZipCodes 中新增数据：

```
001    procedure TForm1.FillZipCodeData;
002      procedure InsertZipData(const iZipCode : Integer; const
sName : String);
003      begin
004        fdmtZipCodes.Insert;
005        fdmtZipCodes.FieldByName('邮政编码').Value := iZipCode;
006        fdmtZipCodes.FieldByName('区名').Value := sName;
007        fdmtZipCodes.Post;
008      end;
009
010    begin
011      InsertZipData(100, '中正区');
012       ...
013      InsertZipData(116, '文山区');
014    end;
```


一但加入了数据之后就可以呼叫 DisplayZipCodes 方法显示邮政编码数据了：

```
procedure TForm1.DisplayZipCodes;
var
  alv : TListViewItem;
begin
  lvZipCodes.Items.Clear;
  fdmtZipCodes.First;
  while (not fdmtZipCodes.Eof) do
  begin
```

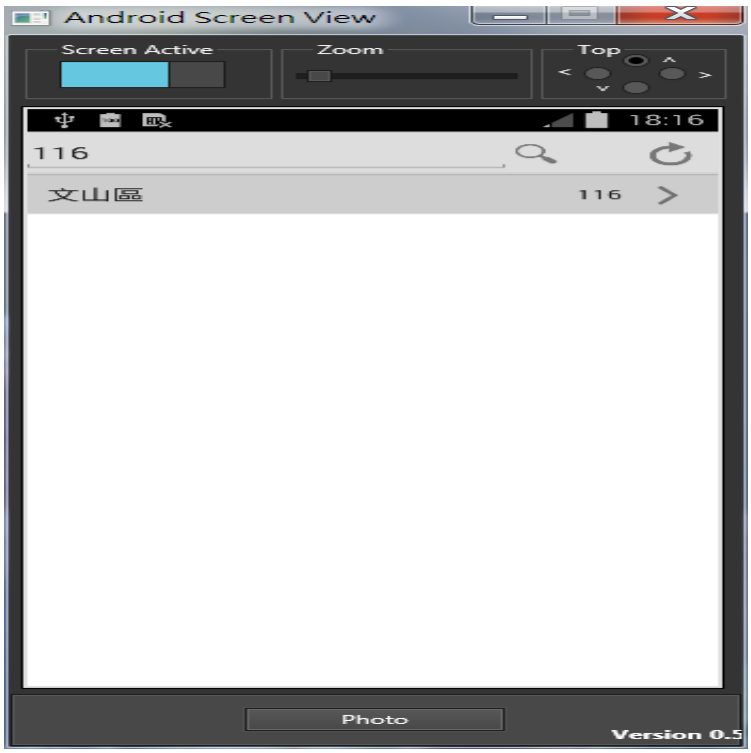
```
    alv := lvZipCodes.Items.Add;
    alv.Detail := fdmtZipCodes.FieldByName('邮政编码').Value;
    alv.Text := fdmtZipCodes.FieldByName('区名').Value;
    fdmtZipCodes.Next;
  end;
  fdmtZipCodes.First;
end;
```

如果现在执行此范例程序就可以在手机中看到如下的画面：

最后加入查询数据的功能，藉由使用过滤器功能查询：

```
001    procedure TForm1.QueryZipCode(const sZipCode : String);
002    begin
003      fdmtZipCodes.Filtered := False;
004      fdmtZipCodes.Filter := '"邮政编码"=' + edtZipCode.Text;
005      fdmtZipCodes.Filtered := True;
006      DisplayZipCodes;
007    end;
```

如果现在执行此范例程序就可以在手机中看到可以正确的查询邮政编码了：

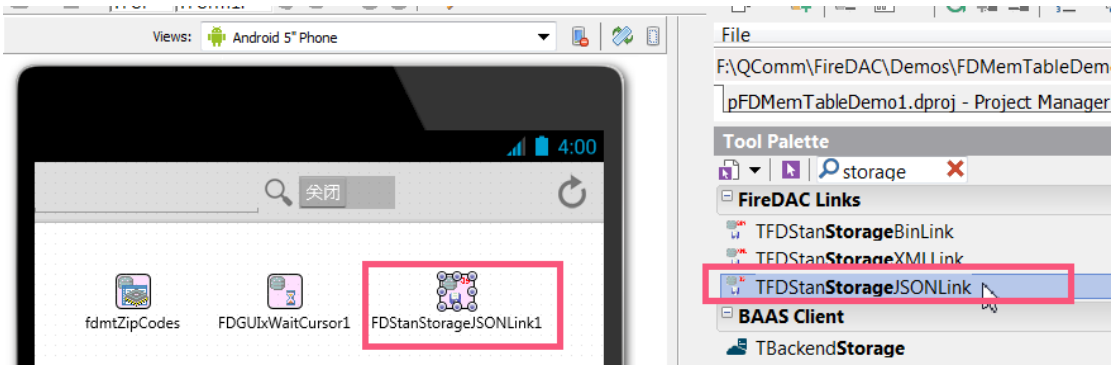


TFDMemTable 组件也可以把它包含的内存数据以不同的格式储存到档案中，或是再从档案中再读取回来，TFDMemTable 的 SaveToFile 和 LoadFromFile 方法就可以完成这 2 个工作。SaveToFile 和 LoadFromFile 方法的第 2 个参数可指定以什么格式储存/读回调案内容，目前 FireDAC 支持的格式有：

格式	说明
sfXML	XML 格式
sfBinary	二进制
sfJSON	JSON 格式
sfAuto	自动判断格式

现在让我们继续为上面的范例加入储存到档案的功能，假设璋在我们希望这个 App 在启动时建立了邮政编码之后就把这个数据表储存下来，在下次 App 启动时就直接从储存的档案中读取数据，而且我们要以 JSON 的格式储存邮政编码数据。

首先请在主表格中加入 TFDStanStorageJSONLink 组件让这个 App 支持以以 JSON 的格式储存和读取数据，如下所示：



修改主窗体的 OnCreate 事件处理函式，如果发现已经有先前储存的邮政编码档案就呼叫 LoadZipCodeFile 方法从邮政编码档案读回数据，如果没有的话就像先前一样建立邮政编码数据表最后加入呼叫 SaveZipCodeFile 方法储存递区号数据到邮政编码档案：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if (not FindZipCodeFile) then
  begin
    CreateZipCodeTable;
    FillZipCodeData;
    SaveZipCodeFile;
  end
  else
  begin
    LoadZipCodeFile;
    swLoadFromFile.IsChecked := True;
  end;
end;
```


LoadZipCodeFile 方法呼叫 LoadFromFile 方法从 App 的文件目录中读取邮政编码档案，传入 LoadFromFile 方法的第 2 个参数使用了 sfJSON 代表是读取以 JSON 格式储存的数据：

```
function TForm1.GetZipCodeFile: String;
begin
  Result := TPath.GetDocumentsPath + PathDelim + sZIPCODEFILE;
end;

procedure TForm1.LoadZipCodeFile;
```

```
begin
  fdmtZipCodes.LoadFromFile(GetZipCodeFile, sfJSON);
end;
```

SaveZipCodeFile 則是呼叫 SaveToFile 方法并且以 JSON 格式储存数据：

```
procedure TForm1.SaveZipCodeFile;
begin
  fdmtZipCodes.SaveToFile(GetZipCodeFile, sfJSON);
end;
```

現在如果执行这个范例 App 就可以发现第 2 次执行 App 时上方的 TSwitch 组件是在打开的状态，代表数据是从邮政编码档案中读回的了：

如果我们观察邮政编码档案的实际内容也可以看到如下的结果，可以确定邮政编码数据的确是以 JSON 的格式储存和读回的：

```
{"FDBS":{"Version":14,"Manager":{"UpdatesRegistry":true,"TableLi
st":[{"class":"Table","Name":"fdmtZipCodes","SourceName":"Table"
,"TabID":0,"EnforceConstraints":false,"MinimumCapacity":50,"Chec
kNotNull":false,"ColumnList":[{"class":"Column","Name":"邮政编码
```

```
","SourceName":"邮政编码
","SourceID":1,"DataType":"Int32","Searchable":true,"AllowNull":
true,"Base":true,"OAllowNull":true,"OInUpdate":true,"OInWhere":t
rue,"OriginColName":"邮政编码"},{"class":"Column","Name":"区名
","SourceName":"区名
","SourceID":2,"DataType":"WideString","Size":30,"Searchable":tr
ue,"AllowNull":true,"Base":true,"OAllowNull":true,"OInUpdate":tr
ue,"OInWhere":true,"OriginColName":"区名
","SourceSize":30}],"ConstraintList":[],"ViewList":[],"RowList":
[{"RowID":0,"RowState":"Unchanged","Original":{"邮政编码":100,"区
名":"中正区"}},{"RowID":1,"RowState":"Unchanged","Original":{"邮政
编码":103,"区名":"大同区"}},
    … ,{"RowID":22,"RowState":"Unchanged","Original":{"邮政编码
":251,"区名":"淡水区
"}}]}],"RelationList":[],"UpdatesJournal":{"SavePoint":23,"Chang
es":[]}}}}
```

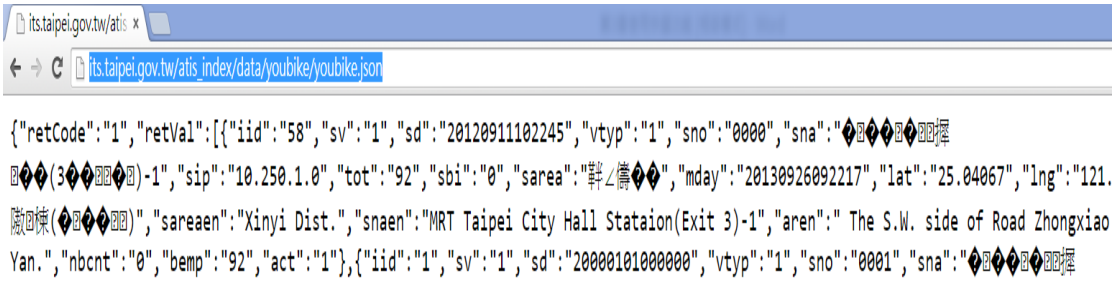

3-1-2 使用 TFDMemTable 处理 SOAP/REST 取得的数据

现在的应用环境中数据源愈来愈多样化，早已跳脱数据大多只从数据库来的应用，特别是在移动平台中数据可能是从网站或是远程服务而来。这些新的数据源格式可能是文字，X M L 或是现在最普遍的 JSON 格式。对于这些应用 TFDMemTable 组件也非常的适合，因为我们可以把这些数据源的数据转到 TFDMemTable 中之后就可以搭配数据感知组件或是 LiveBidning 技术来使用。

例如下面的 URL 以 JSON 格式提供了台北市 YouBike 的信息：

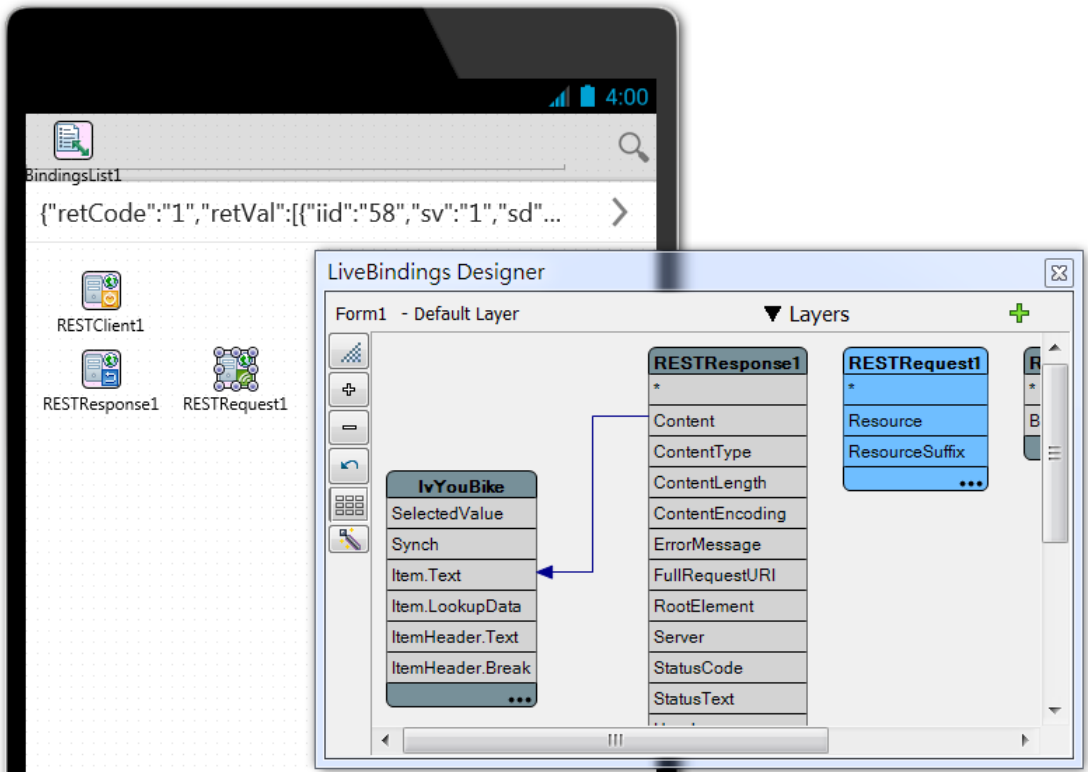
```
http://its.taipei.gov.tw/atis_index/data/youbike/youbike.json
```

如果使用浏览器浏览上面的 URL 可以看到这些资料的确是以 JSON 的格式提供的：

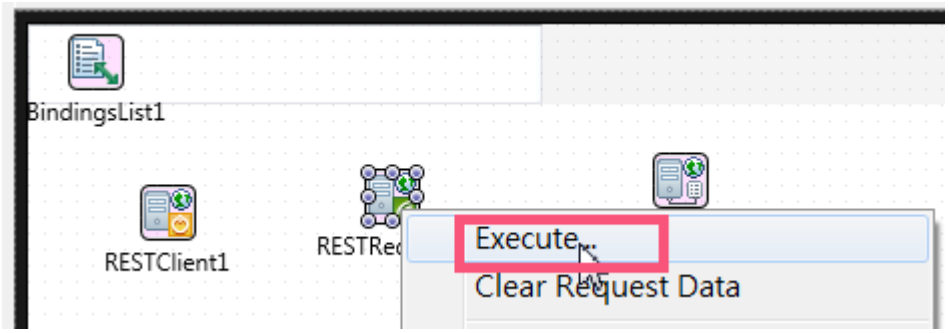


现在让我们使用 TFDMemTable 组件来处理上述的 JSON 格式资料，让我们可以在上面的资料中进行搜寻的工作。

建立一个 Multi-Device Application，在其中加入 TRESTClient，TRESTRequest，TRESTResponse 组件和 ToolBar，TEdit，TButton 和 TListView 组件下所示：

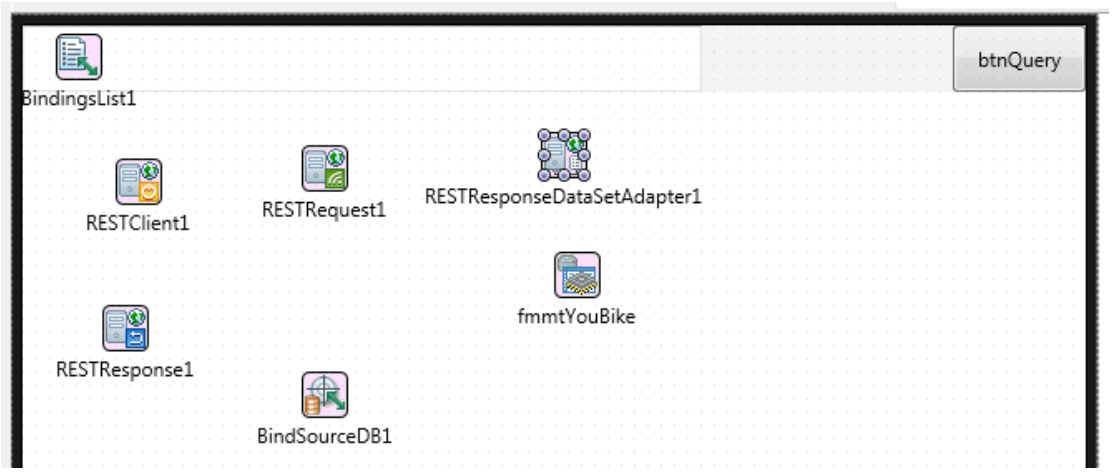


把上面的 URL 设定到 TRESTClient 的 BaseURL 特性中，点选 TRESTRequest 再点选鼠标右键选择其中的 Execute...选项提出请求欲取得 YouBike 资料：

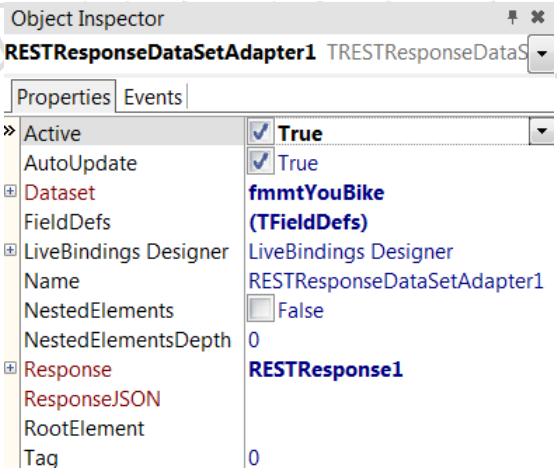


再使用 LiveBinding 技术链接 YouBike 数据和 TListView 组件。

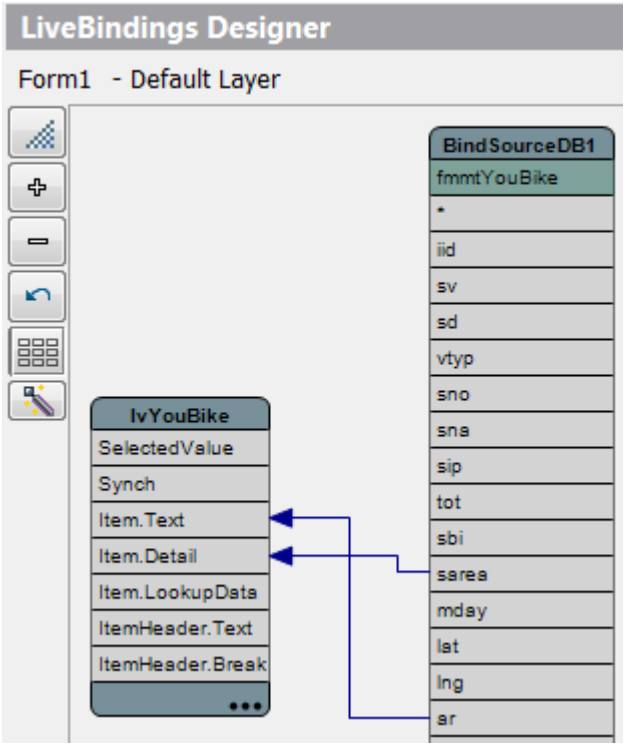
现在就可以准备把 JSON 的 YouBike 数据转到 TFDMemTable 中，请在主窗体中加入 TRESTResponseDataSetAdapter 和 TFDMemTable 组件：



设定 TRESTResponseDataSetAdapter 组件的 Response 特性值为主窗体中的 TRESTResponse 组件，设定 DataSet 特性值为刚加入的 TFDMemTable 组件。再把 TRESTResponse 组件的 RootElement 特性值设定为 retVal，最后再设定 TRESTResponseDataSetAdapter 组件的 Active 特性值为 True：



使用 LiveBidining 重新链接 ListView 组件和刚加入的 TFDMemTable 组件：



几在主窗体就可以看到如下的画面 YouBike 的 JSON 格式数据就转到 TFDMemTable 组件中了：

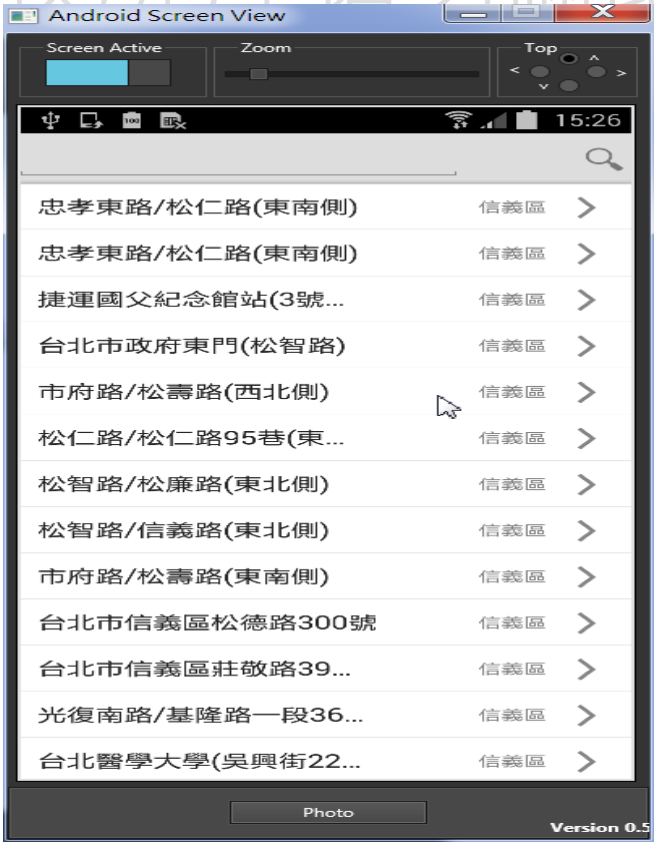
最后再主窗体的 OnActivate 事件中呼叫 TRESTRequest 组件的 Execute 方法在 App 执行时取得 YouBike 数据：

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  RESTRequest1.Execute;
end;
```

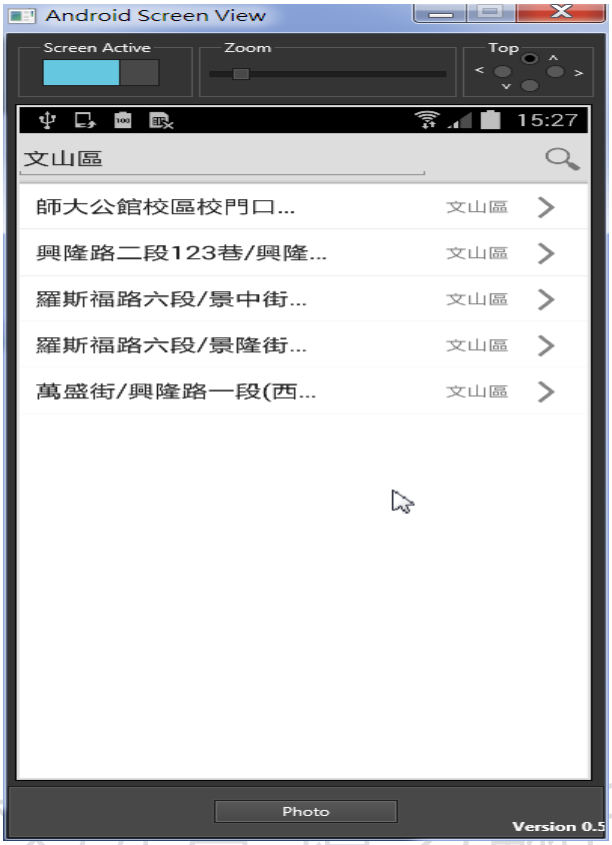
再实作查询 TButton 的 OnClick 事件如下，使用 TFDMemTable 组件的过滤器功能查询数据：

```
procedure TForm1.btnQueryClick(Sender: TObject);
begin
  fmmtYouBike.Filtered := False;
  fmmtYouBike.Filter := '"sarea"=' + '''' + edtArea.Text + '''';
  fmmtYouBike.Filtered := True;
  LinkFillControlToField1.BindList.FillList;
end;
```

执行范例 App 就可以在手机中看到如下的画面，显示了 YouBike 的数据：



也可以在主窗体的 TEdit 组件输入数据查询每一行政区的 YouBike 数据了：



3-1-3 使用 TFDMemTable 处理数据

TFDMemTable 另外一个经常使用的场景就是使用它来共享一个 TDataSet 组件中的数据然后再于 TFDMemTable 中处理数据，这样做的原因可能有很多，但最常应用程状况是因为在许多应用中我们不希望影响到原本 TDataSet 链接的数据应知组件，因此在另一个 TFDMemTable 中处理数据即可避免。

另外一个原因则是因为 TFDMemTable 处理数据的速度很快，因此可以在另外的线程中使用 TFDMemTable 而不是原本的 TDataSet。

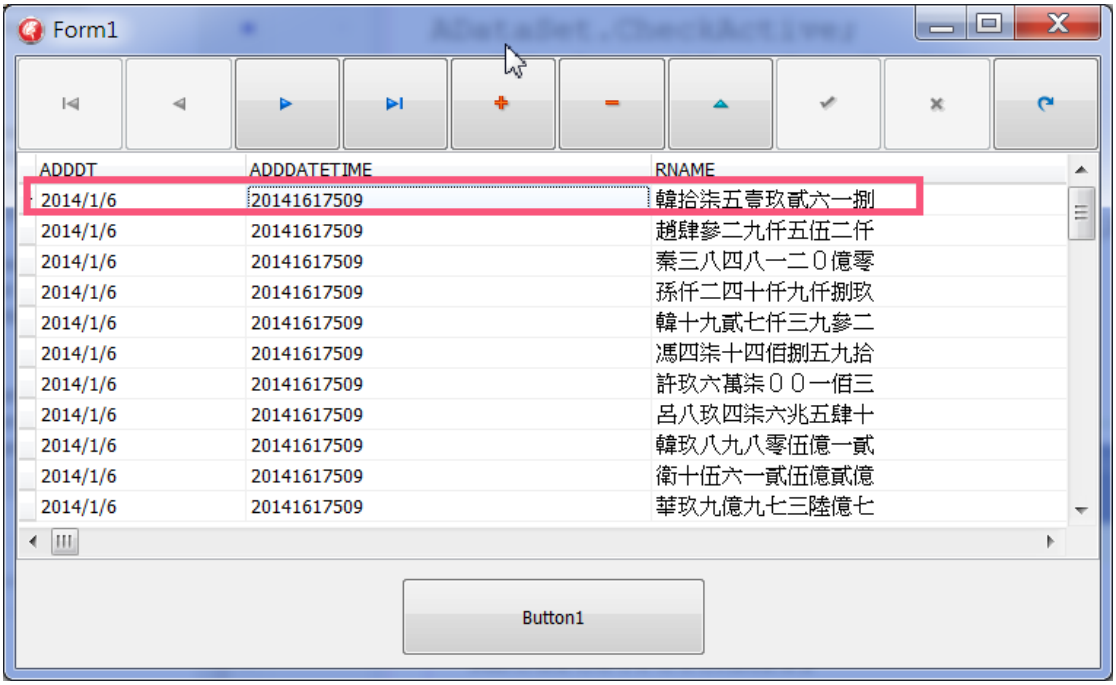
要让 TFDMemTable 共享 TDataSet 组件中的数据，程序员可以呼叫 CloneCursor 方法，下面是它的宣告原型：

```
procedure TFDDataSet.CloneCursor(ASource: TFDDataSet; AReset,
  AKeepSettings: Boolean);
```

CloneCursor 方法接受 3 个参数，第 1 个参数是来源 TDataSet 组件，第 2 个参数 AReset 如果设定为 True 的话代表要设定 TFDMemTable 组件所有的特性值都将设定为内定值，由于通常我们是希望分享来源 TDataSet 组件的设定，因此一般情形下都是设定

AReset 为 False。第 3 个参数 AKeepSettings 设定为 True 的话代表 TFDMemTable 的任何先前设定都不改变，因此一般情形下也都是设定 AKeepSettings 为 False。

让我们看个使用 CloneCursor 的范例，下面是一个使用 TFDQuery 取得数据的画面，假设现在我们要改变其中 ADDDT 字段中的日期，但又不希望影响到目前的 TDBGrid 显示数据的状态：



那我们就可以使用下面的程序代码，先呼叫 TFDMemTable 的 CloneCursor 方法让 TFDMemTable 可以分享 TbltestdataTable 中的数据，

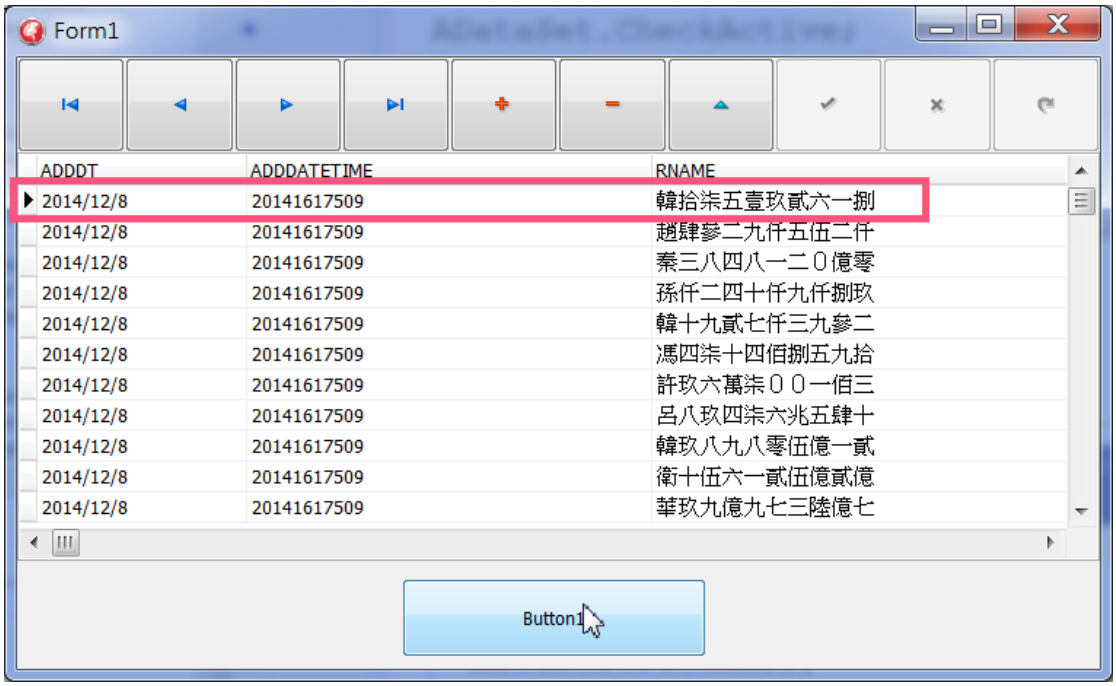
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FDMemTable1.CloneCursor(TbltestdataTable, False, False);
  Application.OnIdle := UpdateToToday;
end;
```

然后再使用 TFDMemTable 修改 ADDDT 字段中的日期数据：

```
procedure TForm1.UpdateToToday(Sender: TObject; var Done: Boolean);
begin
  FDMemTable1.First;
  while (not FDMemTable1.Eof) do
  begin
    FDMemTable1.Edit;
```

```
    FDMemTable1.FieldByName('ADDDT').Value := Now;
    FDMemTable1.Post;
    FDMemTable1.Next;
  end;
end;
```

从下面的结果可以看到，在点选了主窗体中的按钮之后 TFDMemTable 就立刻修改了 ADDDT 字段中的日期数据，但 TDBGrid 完全没有受到影响：



TFDMemTable 处理数据另外一个经常使用的场景就是开发多层的系统，在多层架构中通常是由客户端向中介的服务器请求服务，如果客户端向中介的服务器请求数据的话，中介服务器可以回传 TDataSet 对象回客户端。一旦客户端取得了回传的 TDataSet 对象我们就可以使用 TFDMemTable 来接受并处理回传的数据。

例如下面就是一个中介服务器的方法 Divisions，它回传 TDataSet 对象：

```
public
  { Public declarations }
  ...
  function Divisions : TDataSet;
```

...

下面是中介服务器使用 TFDQuery 组件从数据表中取出数据接着直接回传 TFDQuery 回客户端：

```
function TsmCRUDServer.Divisions: TDataSet;
begin
  if (not TbldivisionsTable.Active) then
    TbldivisionsTable.Active := True;
  Result := TbldivisionsTable;
end;
```

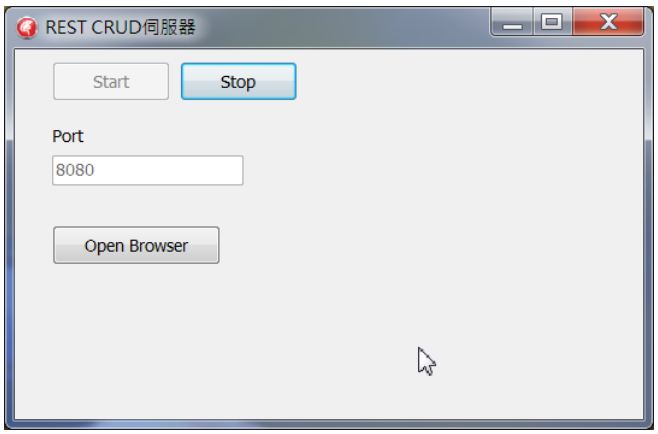
TFDMemTable 的 CopyDataSet 方法可以从一个 TDataSet 中拷贝数据，下面是 CopyDataSet 方法的宣告原型：

```
procedure TFDDataSet.CopyDataSet(ASource: TDataset;
  AOptions: TFDCopyDataSetOptions = [coRestart, coAppend]);
```

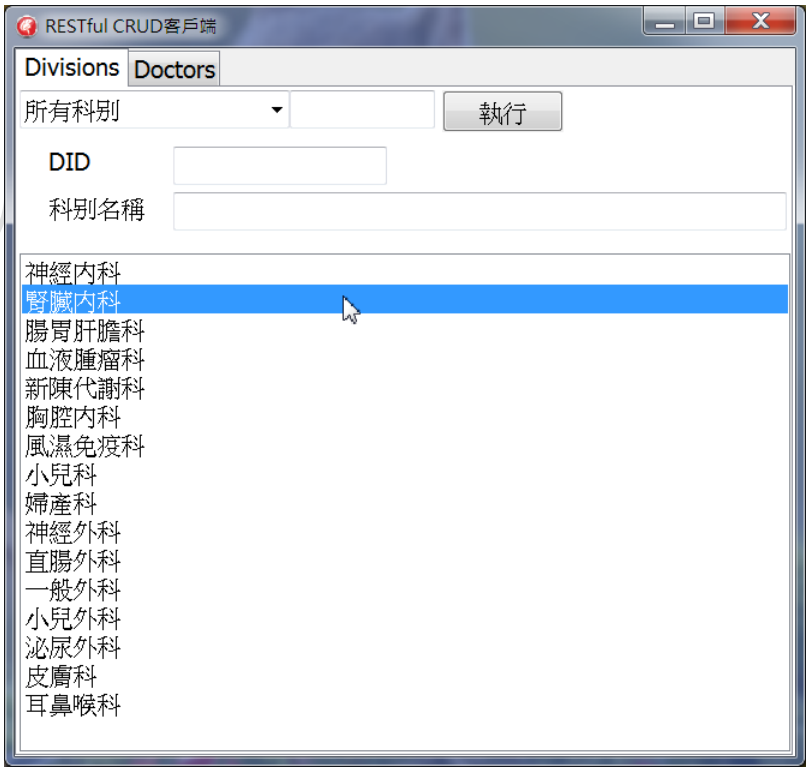
CopyDataSet 方法的接受 2 个参数，第 1 个参数是来源 TDataSet 对象，第 2 个参数 TFDCopyDataSetOptions 则代表要如何从第 1 个参数的 TDataSet 对象中拷贝数据。因此要拷贝一个回传到客户端的 TDataSet 对象，我们需要拷贝它的结构和数据。在下面的客户端程序代码中呼叫了 TFDMemTable 的 CopyDataSet 方法，其中传入的第 2 个参数值[coStructure, coRestart, coAppend]代表要拷贝来源 TDataSet 的结构，再拷贝数据到 TFDMemTable 中并且取后把目前数据位置移到第 1 笔：


```
procedure TfmMainForm.GetAllDivisions;
var
  aServer : TsmCRUDServerClient;
  aDS : TDataSet;
begin
  aServer :=
TsmCRUDServerClient.Create(dmClient.DSRestConnection1);
  try
    aDS := aServer.Divisions();
    dmClient.fdmtDivisions.CopyDataSet(aDS, [coStructure,
coRestart, coAppend]);
    DisplayDivisions;
  finally
    aServer.Free;
  end;
end;
```

执行了上面的程序代码之后中介服务器回传的数据就存在于 TFDMemTable 组件中了，接着我们就可以使用 TFDMemTable 处理这些数据。例如下面的个画面就是中介服务器：



下面则是客户端 Windows 程序，它从上面的中介服务器取得数据并拷贝到 TFDMemTable 组件中后就可以进行显示和处理的工作了：



3-2 结论

不论是桌上型，C/S，多层或是移动设备的应用程序，能够尽量在内存中有效的处理数据是开发数据库相关的应用程序最有效率的方法，本章讨论的 TFDMemTable 是

FireDAC 框架中处理数据最快速的方法，在读者掌握了这些技巧之后应该就可以开发出非常有效率的 FireDAC 数据库程序系统了。

第4章 FireDAC进阶功能

FireDAC 提供了丰富的功能让程序员处理数据，这些数据报含了数据库中的用户数据以及数据库的 MetaData。此外 FireDAC 也提了非常具有弹性的方式让程序员存取数据，例如 Macro，动态 SQL 等。

本章的内容将说明如何使用这些 FireDAC 的功能，让我们从 MetaData 开始吧。

4-1 存取 MetaData

FireDAC 提了数种方式让程序员可存取数据库的 MetaData，例如数据库的 Catalog，Schema，Package 等信息，或是数据库中的数据表，字段等信息。在 FireDAC 中程序员可以使用 TFDConnection 组件或是 TFDMetaInfoQuery 组件。

4-1-1 使用 TFDConnection 组件存取 MetaData

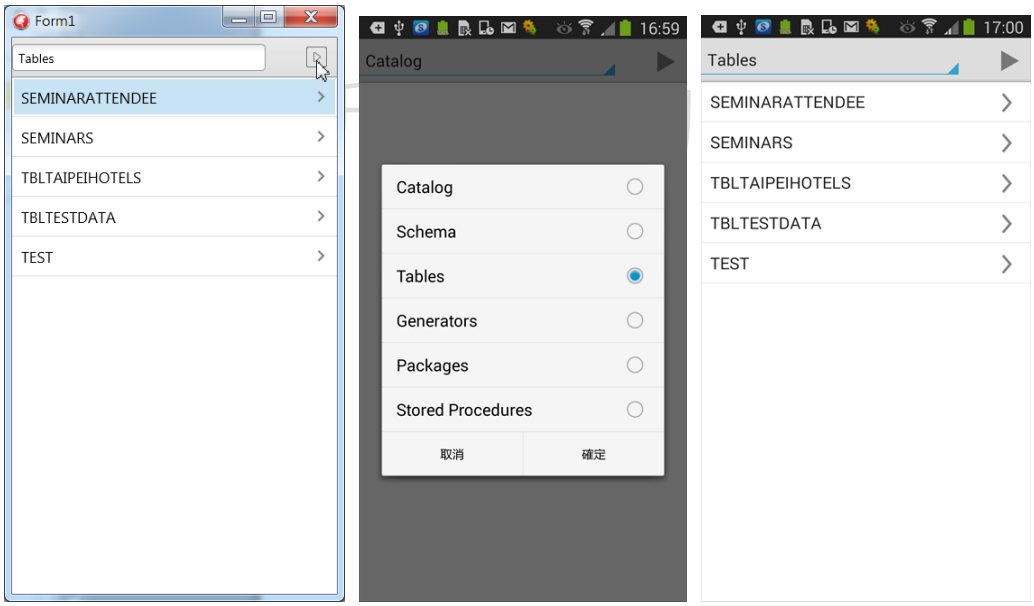
TFDConnection 组件提供了数个方法可让程序员存取数据库中的 MetaData，下面的表格说明了这些相关的函式：

TFDConnection方法	说明
GetCatalogNames	取得数据库 Catalog MetaData
GetSchemaNames	取得数据库 Schema MetaData
GetTableNames	取得数据库中所有数据表或 View 列表
GetFieldNames	取得数据库中数据表的字段列表
GetKeyFieldNames	取得数据库中数据表的键值字段列表
GetGeneratorNames	取得数据库中 Generato/Sequence 列表
GetPackageNames	取得数据库中 Package 列表
GetStoredProcNames	取得数据库中预储程序列表

例如要取得数据库中所有的数据表，那就可以呼叫 TFDConnection 组件的 GetTableNames 方法：

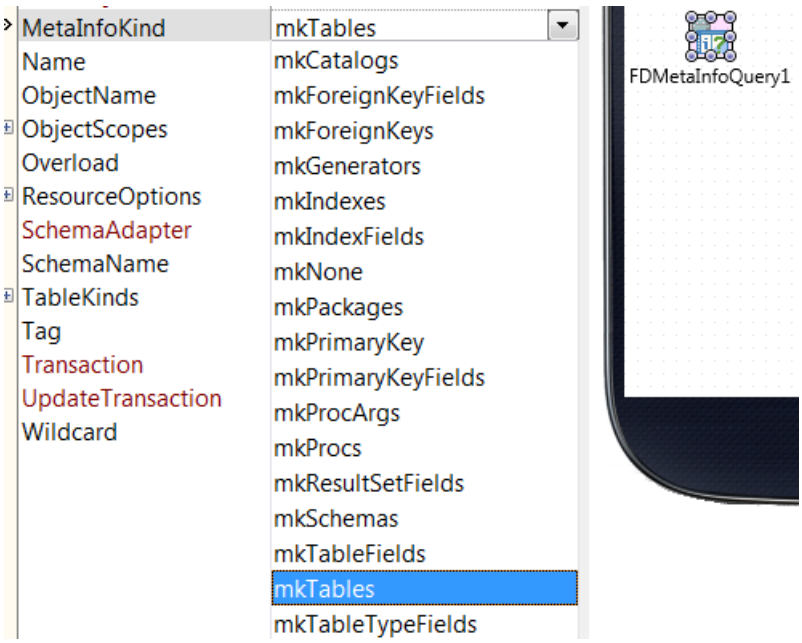
```
procedure TfmMainForm.GetTableMetaData;
var
  aSL : TStringList;
begin
  aSL := TStringList.Create;
  try
    dmMetaData.FiredacdemodbConnection.GetTableNames('', '', '', aSL);
    DisplayMetaData(aSL);
  finally
    aSL.Free;
  end;
end;
```

下图就是此程序代码执行的时取得范例数据库中所有范例数据表的结果：



4-1-2 使用 TFDMetaInfoQuery 组件存取 MetaData

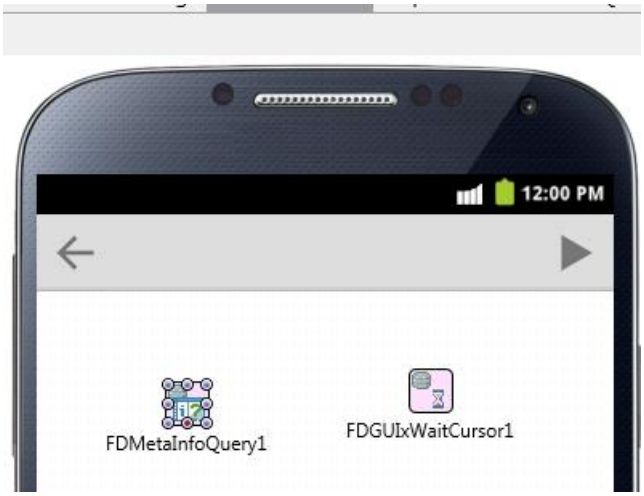
第 2 种方式就是使用 TFDMetaInfoQuery 组件，使用 TFDMetaInfoQuery 组件可以更方便的取得 MetaData，因为程序员只要设定它的 MetaInfoKind 特性即可取得相对应的 MetaData，如下所示：



当程序员使用 TFDMetaInfoQuery 组件取得 MetaData 之后会以数据集的方式储在在 TFDMetaInfoQuery 组件中，而每一种 MetaData 都定了不同的架构，程序员可以参考下面连结中说明的各种 MetaData 数据结构：


```
http://docwiki.embarcadero.com/RADStudio/XE6/en/Metadata_Structure_(FireDAC)
```

例如下面的范例 App 使用了 TFDMetaInfoQuery 组件来查询范例数据库中的所有数据表，如果使用者点选了一个查询到的数据表就可以再查询其中的所有字段：



要查询数据表，我们可以设定 TFDMetaInfoQuery 的 MetaInfoKind 特性值为 mkTables 再开启 TFDMetaInfoQuery 组件即可：

```
procedure TfmMainForm.SpeedButton1Click(Sender: TObject);
begin
  FDMetaInfoQuery1.MetaInfoKind := mkTables;
  FDMetaInfoQuery1.Open;
  DisplayTables;
end;
```

DisplayTables 方法则根据 http://docwiki.embarcadero.com/RADStudio/XE6/en/Metadata_Structure_(FireDAC)中定义的数据表 MetaData 数据结构从其中取出数据表的名称和型态：

```
procedure TfmMainForm.DisplayTables;
var
  alvi : TListViewItem;
  iTK : TFDPhysTableKind;
begin
  lvTables.Items.Clear;
  while (not FDMetaInfoQuery1.Eof) do
  begin
    alvi := lvTables.Items.Add;
    alvi.Text := FDMetaInfoQuery1.FieldByName('TABLE_NAME').AsString;
    iTK := FDMetaInfoQuery1.FieldByName('TABLE_TYPE').Value;

    if (iTK = TFDPhysTableKind.tkSynonym) then
      alvi.Detail := 'Synonym'
    else
      if (iTK = TFDPhysTableKind.tkTable) then
        alvi.Detail := 'Table'
      else
        if (iTK = TFDPhysTableKind.tkView) then
          alvi.Detail := 'View'
        else
          if (iTK = TFDPhysTableKind.tkTempTable) then
            alvi.Detail := 'TempTable'
          else
            if (iTK = TFDPhysTableKind.tkLocalTable) then
              alvi.Detail := 'LocalTable';
    FDMetaInfoQuery1.Next;
  end;
```

```
end;
```

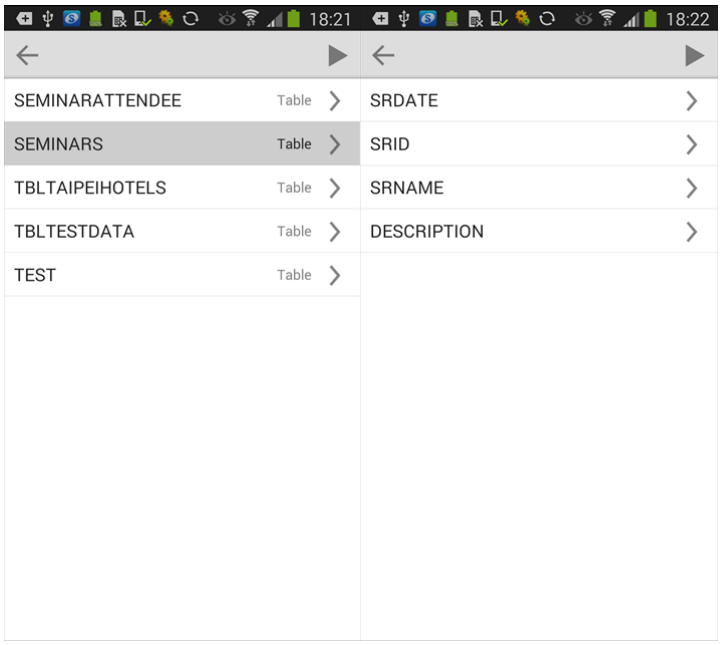
当使用者点选了某一数据表就呼叫 DisplayTableFields 方法显示数据表字段：

```
procedure TfmMainForm.lvTablesItemClick(const Sender: TObject;
  const AItem: TListViewItem);
begin
  DisplayTableFields(AItem.Text);
end;
```

而 DisplayTableFields 方法也是使用 TFDMetaInfoQuery 组件，设定它的 MetaInfoKind 特性值为 mkTableFields 并且设定 ObjectName 特性值为使用点选的数据表名称再开启 TFDMetaInfoQuery 组件即可：

```
procedure TfmMainForm.DisplayTableFields(const sTableName: String);
begin
  if (FDMetaInfoQuery1.Locate('TABLE_NAME', sTableName, [])) then
  begin
    FDMetaInfoQuery1.MetaInfoKind := mkTableFields;
    FDMetaInfoQuery1.ObjectName := sTableName;
    FDMetaInfoQuery1.Open;
    DisplayFields;
    TabControl1.TabIndex := 1;
  end;
end;
```

最后执行此范例 App 就可以看到类似如下的结果：



能够查询和处理 MetaData 有什么用呢？再说明之前先让我们再讨论另一个 FireDAC 的功能，那就是 Macro。

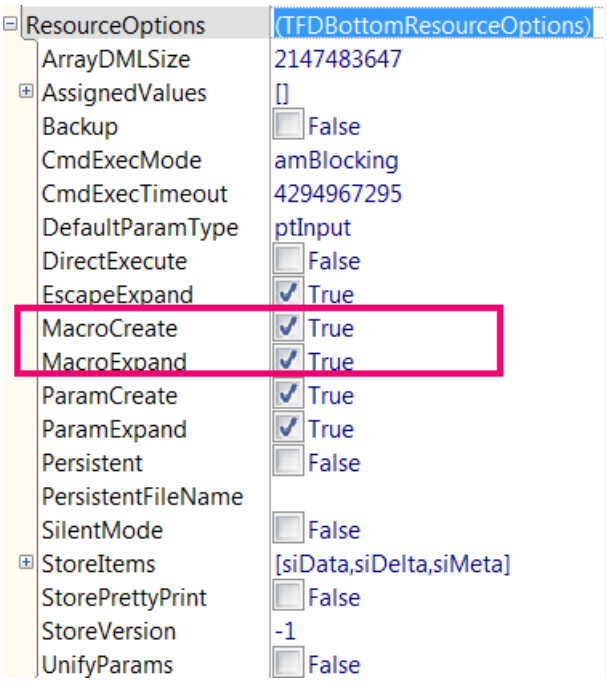
4-2 宏功能(Marco)

FireDAC 提供了丰富的 Macro 功能让开发人员使用，简单的说所谓的宏就是 FireDAC 在把 SQL 命令送到后端执行时会根据宏功能定义先把 SQL 中使用的宏替换成相对应的字符串之后再把替换的 SQL 命令送出。

FireDAC 基本上提供了 3 种宏，它们是：

宏功能	说明
替换变数	可替换 SQL 命令中的字段和数据表名称
Escape sequence	可在 SQL 命令中撰写和后端数据库独立的命令，FireDAC 会自动根据后端数据库替换成后端数据库专属的命令。Escape sequence 是使用{}包围的指令
条件替换	可在 SQL 命令中使用判断条件

要使用 FireDAC 的宏功能，要先把 TFDQuery 组件的 ResourceOptions.MacroCreate 和 ResourceOptions.MacroExpand 特性值设为 True：



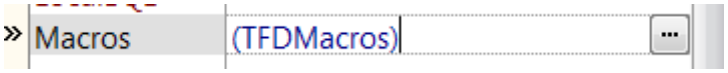
FireDAC 使用'!'或"符号代表宏，

宏符号	说明
!	代表"字符串"替换模式，直接替换
&	代表"SQL 命令"替换模式，FireDAC 会根据数据型态使用 RDMBS 的语法替换

例如如果我们在 TFDQuery 中撰写了下面的 SQL 命令：

```
SELECT * FROM &TableName
```

由于在此 SQL 命令中使用宏来代表要存取的数据表名称，因此在 TFDQuery 的 Macros 特性中：

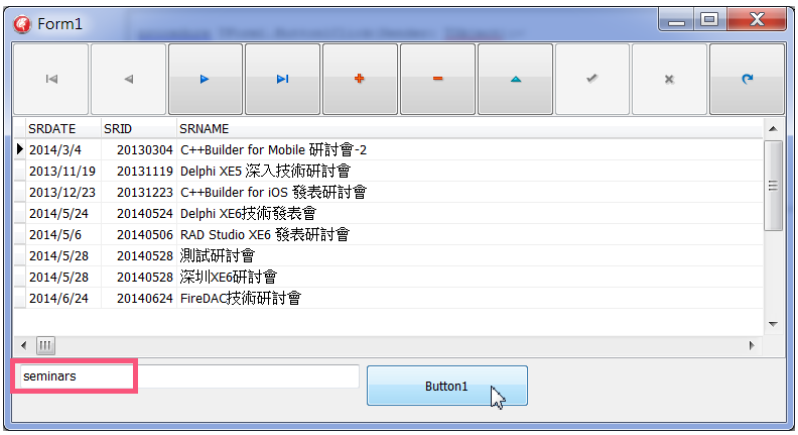


会看到 FireDAC 会解析此 SQL 命令并且发现一个宏，因此就会在 Macros 特性中有一个名为"TABLENAME"的宏：

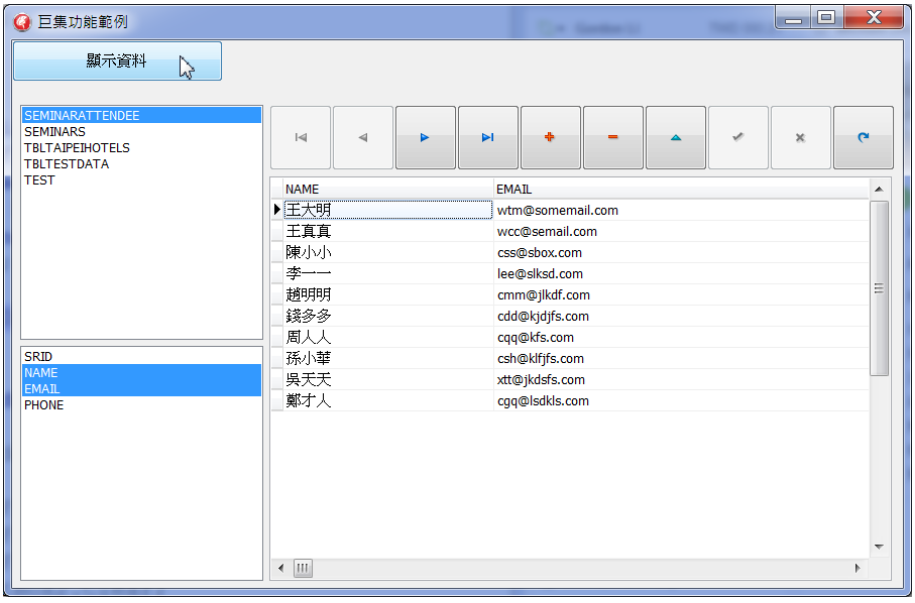
接着我们就可以在程序中使用下面的程序代码在程序执行时动态输入数据表名称来存取数据：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FDQuery1.Active := False;
  FDQuery1.MacroByName('TableName').AsRaw := Edit1.Text;
  FDQuery1.Active := True;
end;
```

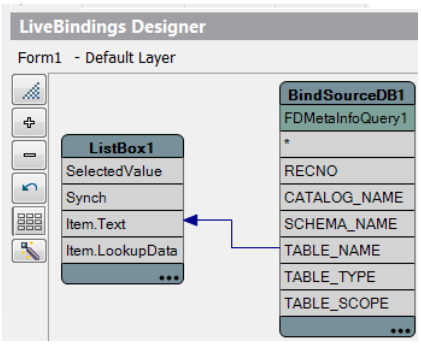
例如下面就是范例程序执行的结果，我们动态输入"seminars"数据表名称后的确可以存取到其中的数据：



除了可使用宏替换数据表名称外也可以替换 SQL 命令其他的内容，现在让我们看另外一个范例。下面是一个范例程序，在执行时此程序使用 TFDMetaInfoQuery 取得链接数据库中所有的数据表名称，使用者点选其中的数据表时左下方的 ListBox 就会显示此数据表中的字段。使用者可于其中选择字段，最后再点选左上方的”显示数据”按钮就可以显示任何数据表中任何字段的数据：



此范例程序使用了 2 个宏 SQL 命令，一个宏 SQL 命令动态从用户选择的数据表中撷取字段信息，一个宏 SQL 命令根据用户选择的字段撷取数据。一开始范例程序使用 TFDMetaInfoQuery 组件取得链接数据库中所有的数据表名称，再藉由 LiveBinding 显示在左上方的 ListBox 中：



当使用者在在左上方的 ListBox 中随意选择一个数据表之后，程序就使用下面的宏 SQL 命令从数据表中取得所有的字段信息，由于我们不希望取得任何数据而是只取得字段信息，因此在下面的宏 SQL 命令的 where 部份使用了永不成立的条件以避免取得任何资料：

```
select * from &TableName where 1 = 2
```

因此在左上方的 TListBox 的 OnClick 事件呼叫 LoadFields 方法藉由刚才说明的宏 SQL 命令从数据表中取得所有的字段信息：

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
  LoadFields;
end;
```


LoadFields 方法把使用者在左方上选择的数据表名称代入宏值中，开启 TFDQuery 组件，取得字段信息再显示在左下方的 ListBox2 之中：

```
procedure TForm1.LoadFields;
var
  iIndex : Integer;
begin
  qryFields.Close;
  qryFields.Macros[0].AsRaw :=
ListBox1.Items[ListBox1.ItemIndex];
  qryFields.Open;
  ListBox2.Clear;
  for iIndex := 0 to qryFields.FieldCount - 1 do
    ListBox2.Items.Add(qryFields.Fields[iIndex].FieldName);
end;
```

当使用者在左下方的 ListBox2 中点选了要显示数据的字段后，范例程序使用了下面的宏 SQL 命令来撷取数据：

```
SELECT &FieldList FROM &TableName
```

上面的 SQL 命令使用了 2 个宏，一个可动态指定数据表，另一个&FieldList 宏则可动态根据用户选择的字段来取得数据。

因此当使用选择完成字段之后可点选主窗体中的"显示数据"按钮，这个按钮呼叫 LoadData 方法取得数据：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  LoadData;
end;
```

LoadData 方法根据使用者在左下方的 ListBox2 中选择的字段组合成字段串行，再于 016~017 行把域名串行代入到宏&FieldList1 之中：

```
001   procedure TForm1.LoadData;
002   var
003     iIndex : Integer;
004     FieldList : String;
005
006     procedure BuildFieldList(Value: string);
007     begin
008       if FieldList = '' then
009         FieldList := Value
010       else
011         FieldList := FieldList + ', ' + Value;
012     end;
013
014     procedure GetData;
015     begin
016       qryData.Macros[0].AsRaw := FieldList;
017       qryData.Macros[1].AsRaw :=
ListBox1.Items[ListBox1.ItemIndex];
018       qryData.Open;
019     end;
020
```

```
021   begin
022     for iIndex := 0 to ListBox2.Items.Count - 1 do
023     begin
024       if (ListBox2.Selected[iIndex]) then
025       begin
026         BuildFieldList(ListBox2.Items[iIndex]);
027       end;
028     end;
029
030     GetData;
031   end;
```

最后数据就可以正确显示在程序的数据感知组件中。

除了替换宏之外，FireDAC 的 Escape sequence 宏也非常的有用，而且弹性非常大，因为 Escape sequence 宏允许程序员可在 SQL 命令中呼叫 FireDAC 定义的函式以及特定数据库中的函式。例如假设你有一个称为"List Price"的字段，但在不同的数据库中有的是使用单引号来指定：

```
Select 'List Price' from Orders
```


有的可能是使用双引号来指定：

```
Select 'List Price" from Orders
```

这会让程序员非常的麻烦，因为程序员需要根据不同的数据库使用不同的符号。在这种情形中 FireDAC 的 escape sequence 宏就非常方便了，程序员只需要使用：

```
Select {id List Price} from Orders
```

一个 SQL 命令即可，其中的{就是 escape sequence 宏，而{中的"id"则是 FireDAC 的宏指令，它可自动判断链接的数据库而替换此 SQL 命令为

```
Select 'List Price' from Orders
```

或是

```
Select 'List Price" from Orders
```

FireDAC 提供的宏指令分成 5 大类，它们是

宏功能	说明
字符宏	http://docwiki.embarcadero.com/RADStudio/XE7/en/Character_Macro_Functions_(FireDAC)
数值宏	http://docwiki.embarcadero.com/RADStudio/XE7/en/Numeric_Macro_Functions_(FireDAC)
日期宏	http://docwiki.embarcadero.com/RADStudio/XE7/en/Date_and_Time_Macro_Functions_(FireDAC)
系统宏	http://docwiki.embarcadero.com/RADStudio/XE7/en/System_Macro_Functions_(FireDAC)
转换宏	http://docwiki.embarcadero.com/RADStudio/XE7/en/Convert_Macro_Function_(FireDAC)

例如在日期宏中有一个 CURRENT_DATE()宏函式，它可回传今天的日期，因此当我们使用如下的程序代码就可以在数据集中加入 Today 字段：

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  qryGeneral.SQL.Text := 'select {CURRENT_DATE()} as Today, SRNAME
' +
    'from Seminars';
  qryGeneral.Active := True;
  DataSource1.DataSet := qryGeneral;
end;
```


TODAY	SRNAME
2014/11/28	C++Builder for Mobile 研討會-2
2014/11/28	Delphi XE5 深入技術研討會
2014/11/28	C++Builder for iOS 發表研討會
2014/11/28	Delphi XE6技術發表會
2014/11/28	RAD Studio XE6 發表研討會
2014/11/28	測試研討會
2014/11/28	深圳XE6研討會
2014/11/28	FireDAC技術研討會

另外在 SEMINARS 数据表中有 2013 和 2014 年的研讨会活动，那么我们如何可以使用宏功能从其中取得不同年次中的活动信息呢？

那么我们可以使用如下的宏 SQL 命令：

```
select {Current_Date()} as Today, s.* FROM &TableName s WHERE
```

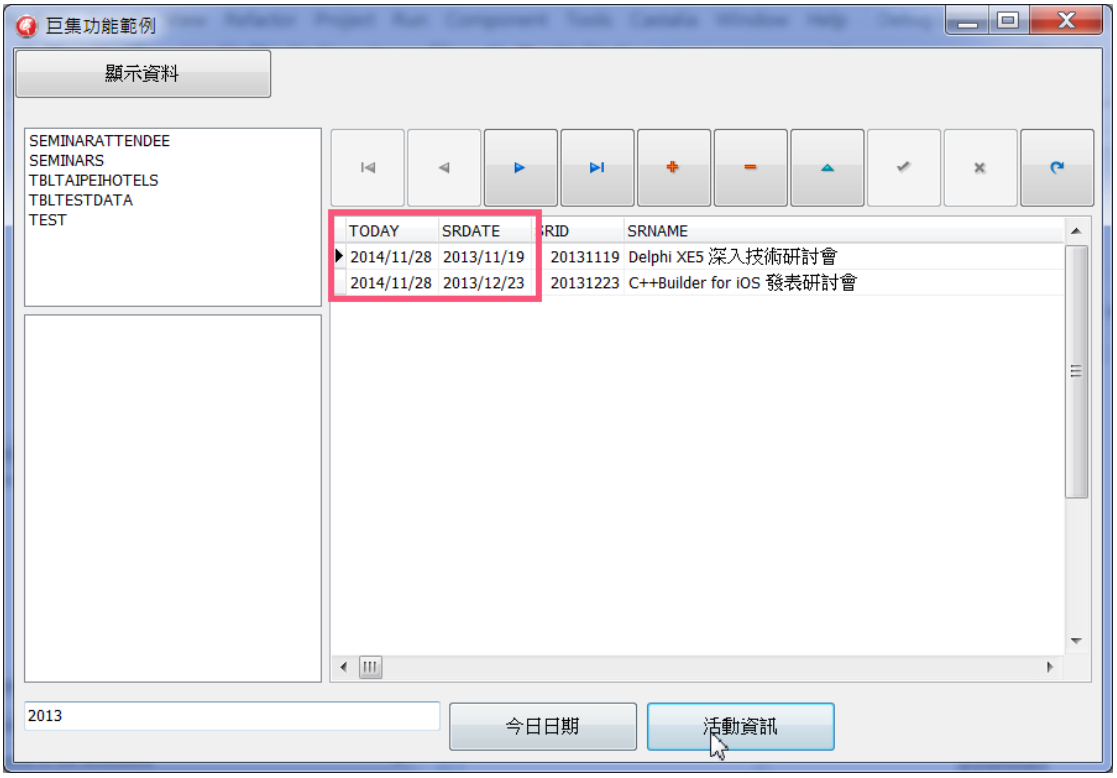
```
{Extract(Year, s.SRDATE)} = :iYear
```

在上的宏 SQL 命令中 Current_Date()，Extract 和 Year 都是 FireDAC 的宏函式，Extract(Year, s.SRDATE)可从 SEMINARS 数据表的 SRDATE 字段中取出年份数据。

最后可使用下面的程序代码动态代入数据表名称和使用者打入的年份来显示研讨会活动：

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  qrySeminars.Macros[0].AsRaw := 'SEMINARS';
  qrySeminars.ParamByName('IYEAR').Value :=
StrToInt(edtYear.Text);
  qrySeminars.Active := True;
  DataSource1.DataSet := qrySeminars;
end;
```

下面是执行范例程序并打入 2013 年，我们可以看到在今日(2014/11/28)日可以查询到 2013 年的活动。



4-3 Update SQL 处理客制化数据

在实际的数据库应用程序中经常会遇到使用 Join SQL 从多个数据表中撷取资料的状况，程序员可以在 TFDQuery 的 SQL 特性中使用 Join SQL 取得资料，例如：

```
Select S.SRDATE, S.SRNAME, A.NAME, A.Email from Seminars S,
SeminarAttendee A where S.SRID = A.SRID
```

就是从 Seminars 和 SeminarAttendee 这 2 个数据表中取得参加每一个研讨会的参加人员名单。

使用 TFDQuery 取得上述的数据虽然很简单，但如果程序员想把 TFDQuery 中的 Join 的数据更新回多个数据表的话，那么就不是这么简单了，因为程序员不能直接呼叫 Post 或是 ApplyUpdates 方法，因为 FireDAC 无法了解要把那一个字段更新回那一个数据表。那么程序员如何能把异动数据更新回多个数据表，或是说程序员如何能把异动数据以客制化的方式更新回后端？

这有数种方式可以完成这个需求，例如程序员可以自行撰写程序代码来更新数据，另外一比较省事的方式就是使用 FireDAC 的 TUpdateSQL 组件。

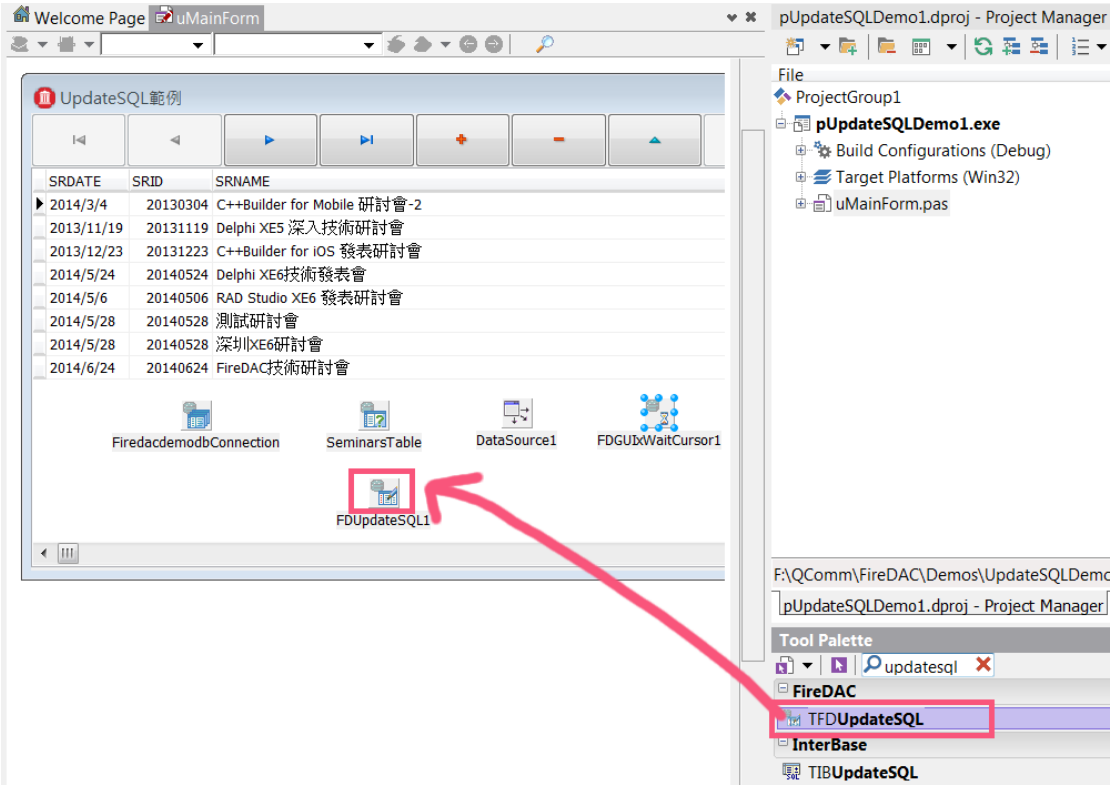
TUpdateSQL 组件有数个功能，例如它可以自动帮程序员产生处理数据的 DML SQL 命令，例如 Insert SQL，Update SQL 和 Delete SQL 等。但 TUpdateSQL 组件最主要提供了下面 2 项在处理复杂数据时最必要的功能：

1. 允许程序员定义客制化更新
2. 允许程序员在 FireDAC 无法处理类似 Join SQL 的数据更新时提供更新数据的能力

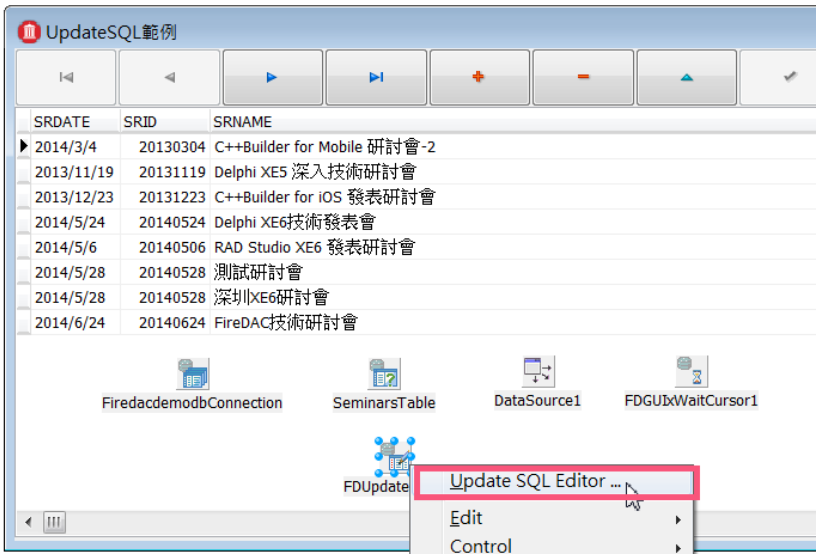
TUpdateSQL 组件和 TFDQuery 组件需要一起工作，在 TFDQuery 组件中有一个 UpdateObject 特性，程序员只需要把这个特性指定为一个 TUpdateSQL 组件即可。或是在 TFDQuery 组件的 OnUpdateRecord 事件中藉由程序代码和 TUpdateSQL 组件来完成工作。让我们使用 2 个范例来说明读者就可以很快的了解。

4-3-1 使用 TUpdateSQL 组件产生 DML

请建立一个 VCL 或是 FireMonkey 应用程序，虽后在其中放入 TFDConnection 和 TFDQuery 组件并链接到范例数据库中的 SEMINARS 数据表，例如下面显示的结果画面，然后从组件盘拖曳 TFDUpdateSQL 组件到主窗体中：

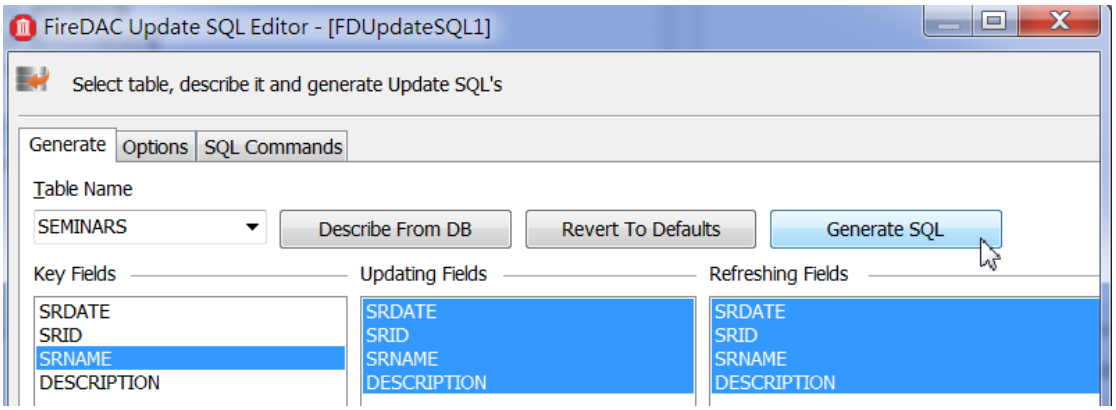


设定 TFDQuery 组件的 UpdateObject 特性为刚才拖曳的 TFDUpdateSQL 组件，点选 TFDUpdateSQL 组件再右击鼠标并点选"Update SQL Editor..."选单以启动 TFDUpdateSQL 的组件编辑器：

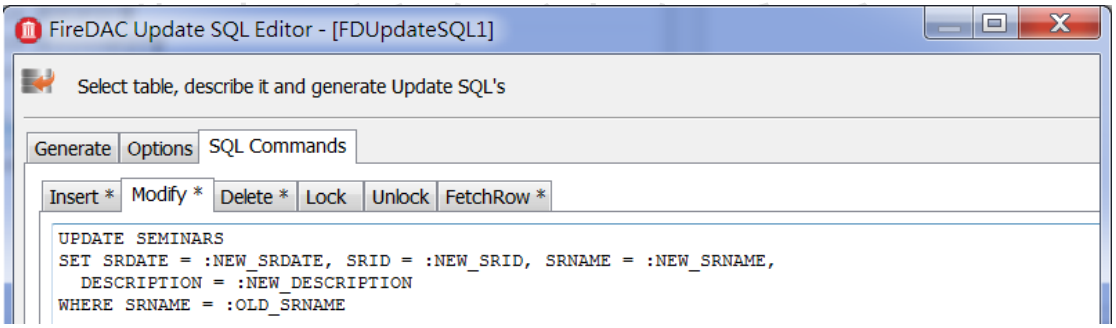


在 TFDUpdateSQL 的组件编辑器可以看到如下的画面，在下方分成 3 个 ListBox，最左方的是 SEMINARS 数据表的 Key Fields 字段其中会自动选择 SEMINARS 数据表的键值字段：SRNAME。中间的 ListBox 则列出所有 SEMINARS 数据表的字段，程序员可以多选任何需要更新的字段。最右方的 ListBox 也列出所有 SEMINARS 数据表的

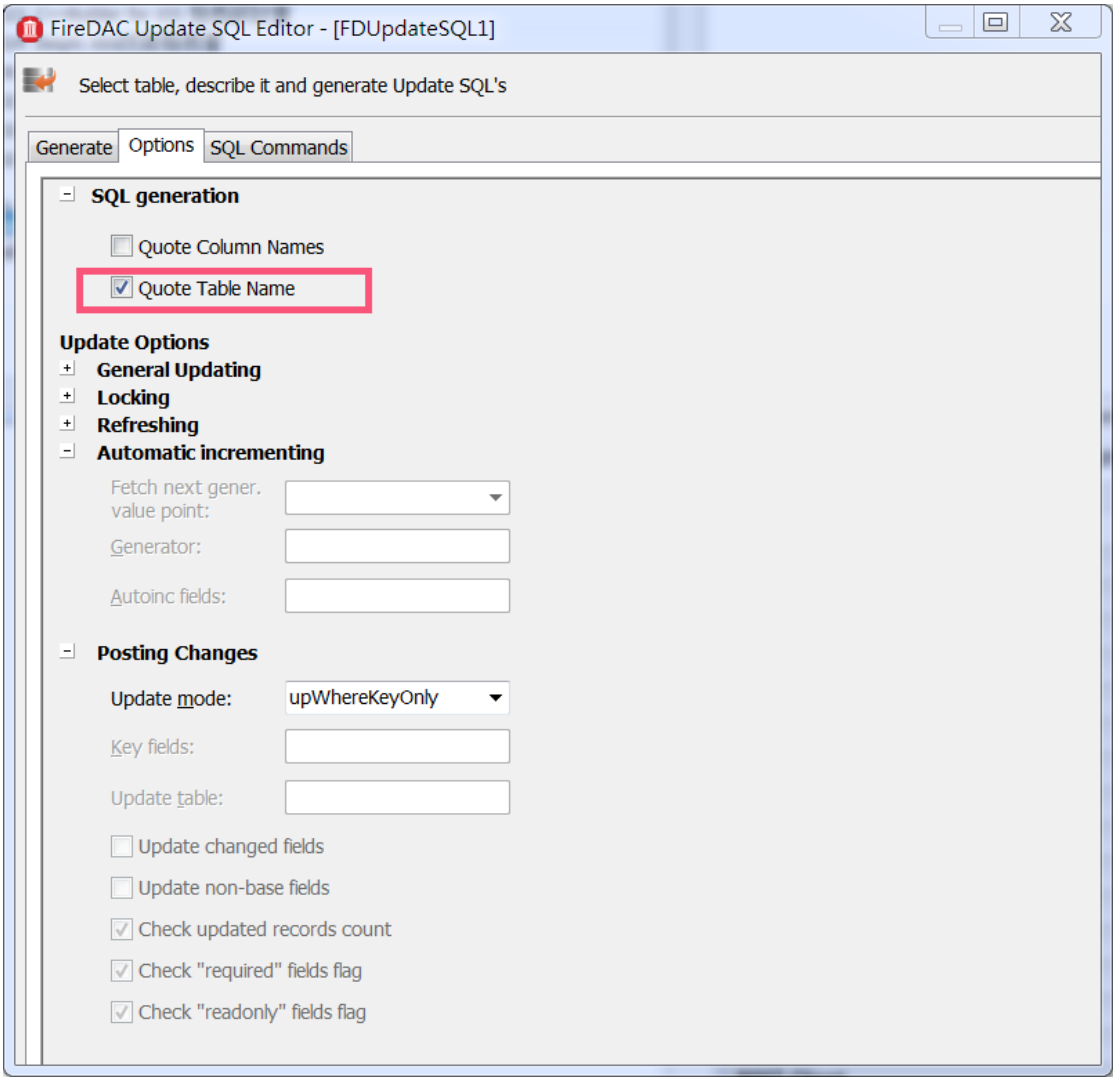
字段，程序员也可多选任何在数据更新之后需要重新显示的字段。在下方的画面中的选择代表 SRNAME 是键值字段，需要产生 Update SQL 更新所有的字段而且在更新数据之后要重新显示所有的字段。



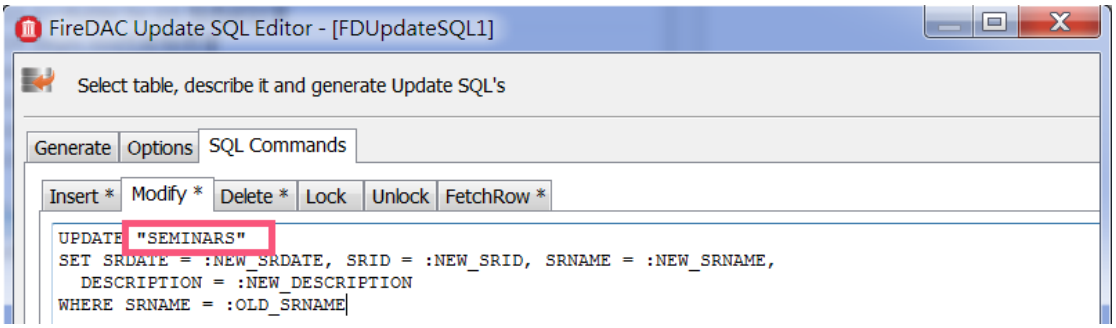
接着点选"Generate SQL"按钮就可以自动产生所有的 DML SQL，例如下面就显示了在点选"Generate SQL"按钮之后再点选 SQL Command 页次就可以看到 TFDUpdateSQL 组件自动根据我们刚才在"Generate"页面中的选择而自动产生的 Update SQL：



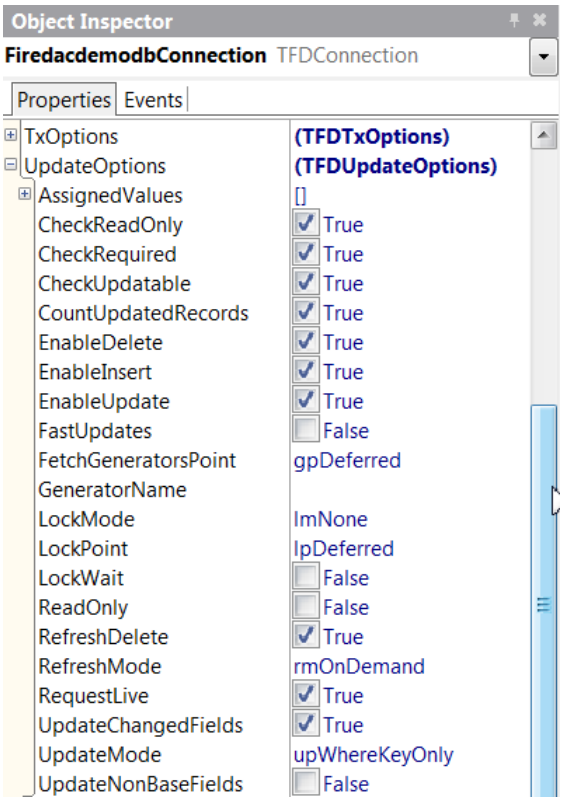
程序员也可以在"Options"页次做更多的控制设定，例如如果设定"Options"页次中的"Quote Table Name"：



选项再产生 DML 的话就可以看到 Update SQL 中的数据表名称被双引号包含起来了。



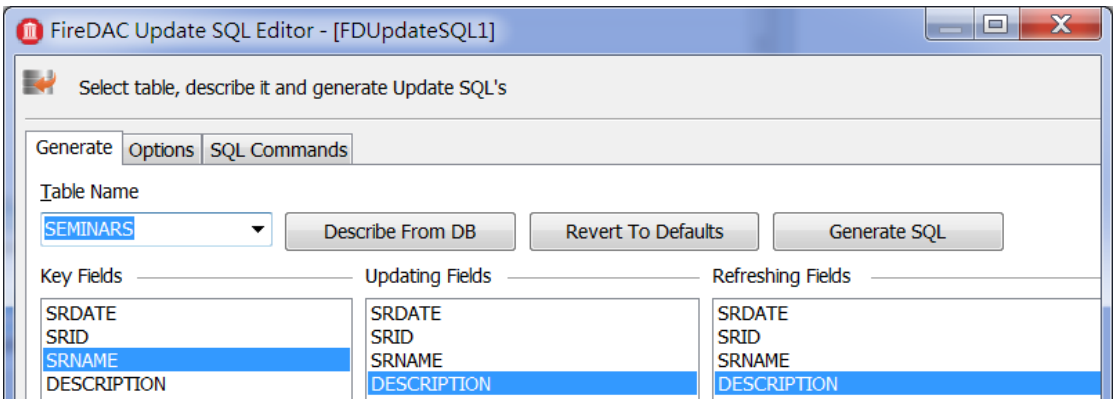
”Options”页次中的许多设定和 TFDConnection 的 UpdateOptions 特值以及 TFDQuery 的特性有关，在稍后的章节中会说明这些特性的意义以及如何使用这些特性。



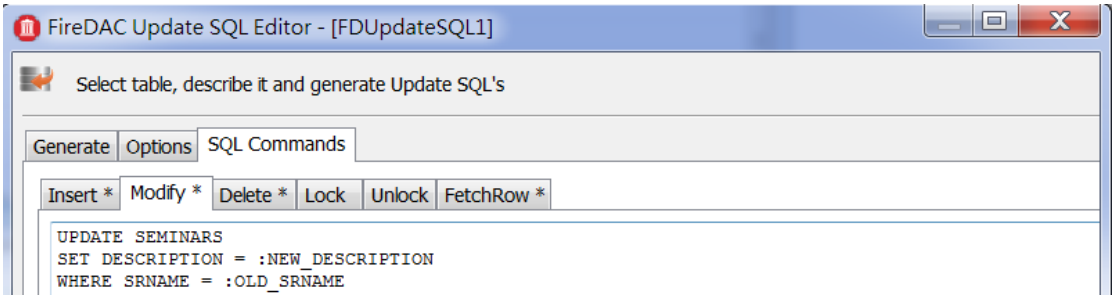
了解了如何使用 TFDUpdateSQL 组件自动产生 DML 之后就可以开始说明如何来使用它来更新资料了，

4-3-2 使用 TUpdateSQL 组件客制化数据更新

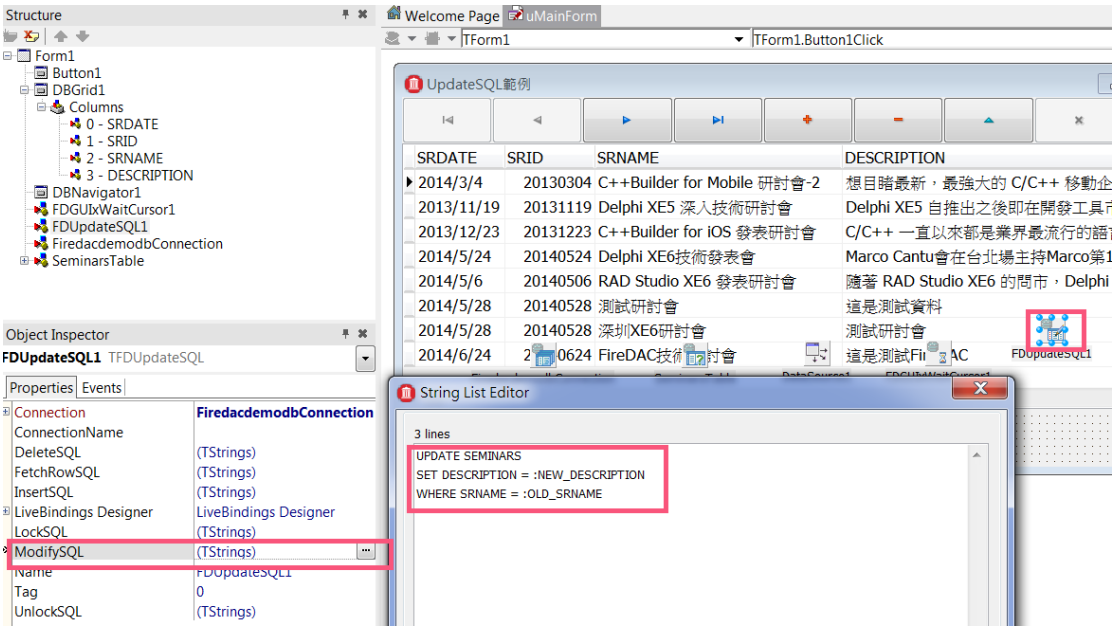
让我们继续用前面的 SEMINARS 数据表做为说明，假设在范例应用程序执行时我们只允许 SEMINARS 数据表的 DECCRIPTION 字段能进行更新，那么我们可以让 TFDUpdateSQL 组件自动产生更新 DECCRIPTION 字段的 Update SQL。请在 TFDUpdateSQL 组件中如下所示在 Update Fields 中只选择 DECCRIPTION 字段虽后点选”Update SQL”按钮：



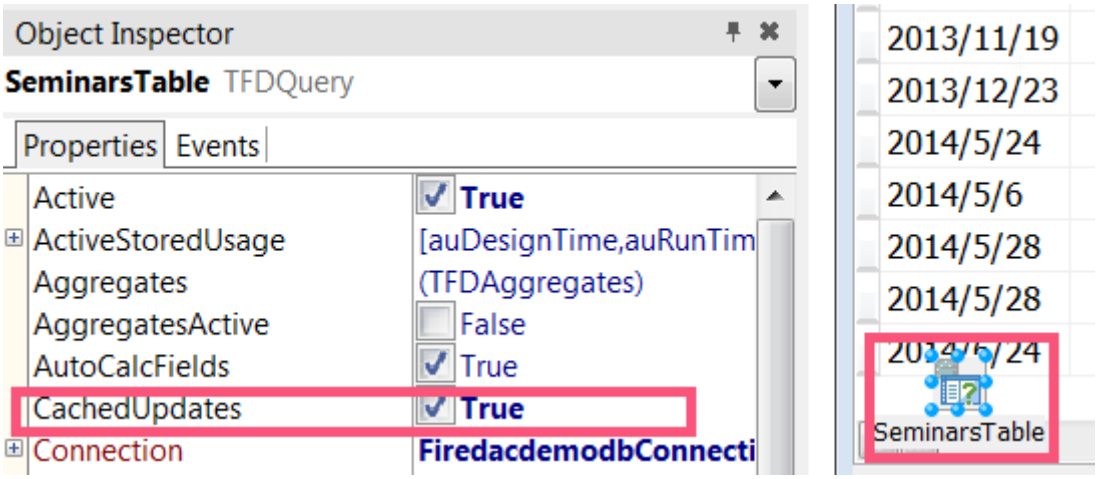
在 TFDUpdateSQL 组件的 Modify 页面中就会产生如下的 Update SQL：



拷贝这个 Update SQL 并把它贴到 TFDUpdateSQL 组件的 ModifySQL 特性中，如下所示：



接着要设定 SeminarsTable 这个 TFDQuery 组件的 CachedUpdates 特性值为 True 以开启 CachedUpdates 功能：

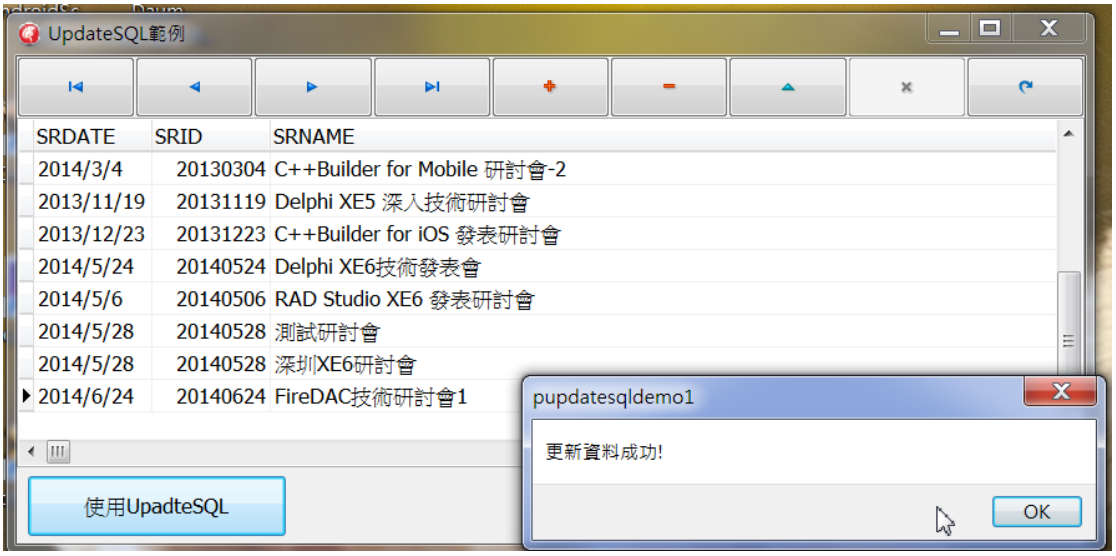


于主窗体中放入一个 TButton 组件并在其 OnClick 事件中撰写如下的程序代码：

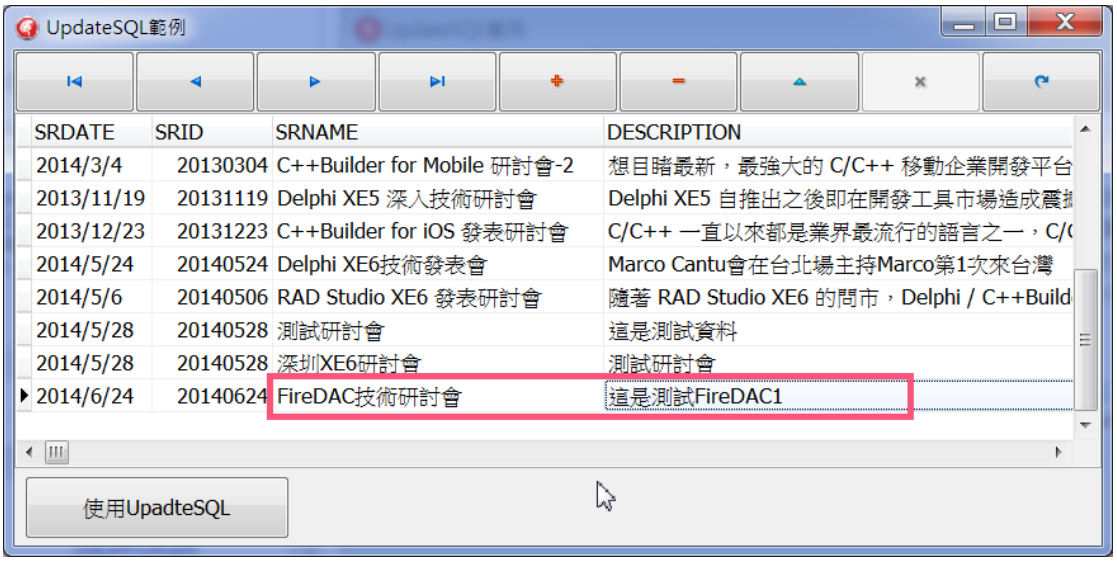
```
001    procedure TForm1.Button1Click(Sender: TObject);
002    var
003      NumErrors: Integer;
004    begin
005      SeminarsTable.OnUpdateRecord := nil;
006      SeminarsTable.UpdateObject := FDUpdateSQL1;
007      NumErrors := SeminarsTable.ApplyUpdates(0);
008      if NumErrors = 0 then
009      begin
010        SeminarsTable.CommitUpdates;
011        ShowMessage('更新数据成功!');
012      end
013      else
014        ShowMessage('产生 ' + IntToStr(NumErrors) + ' 个更新错误');
015    end;
```

由于现在先展示如何使用 TFDUpdateSQL 组件展示如何客制化更新数据稍后会说明如何使用 OnUpdateRecord 事件客制化数据更新，因此 005 行先把 OnUpdateRecord 事设定为 nil，006 行设定 SeminarsTable 的 UpdateObject 特性为 FDUpdateSQL1，007 行再呼叫 ApplyUpdates 把更新的数据更新回 SEMINARS 数据表。由于使用了 FDUpdateSQL1 组件，因此 ApplyUpdates 就会使用 FDUpdateSQL1 组件中的 ModifySQL 特性中的 Update SQL。

现在如果执行此范例程序，并且如下更新 FireDAC 技术研讨会那笔数据，让我们同时异动 SRNAME 和 DESCRIPTION 这 2 个字段的数据之后再点选”使用 UpdateSQL”按钮就可以看到显示” 更新数据成功!”讯息：



但重新显示数据可以看到 DESCRIPTION 这个字段的资料果然更新成功了，而 SRNAME 字段的数据没有更新：



这当然是因为 FDUpdateSQL1 组件中的 ModifySQL 特性的 Update SQL 只把 DESCRIPTION 字段更新回 SEMINARS 数据表：

```
UPDATE SEMINARS
SET DESCRIPTION = :NEW_DESCRIPTION  WHERE SRNAME = :OLD_SRNAME
```


这个范例显示了如何使用 TFDUpdateSQL 组件控制如何更新数据，

4-3-3 使用 OnUpdateRecord 事件客制化数据更新

TFDQuery 的 OnUpdateRecord 事件可以让程序员更精确的控制如何把数据更新回后端。当用户更新了数据之后，如果程序员使用 OnUpdateRecord 事件，那么对每一笔异动的资料 TFDQuery 就会呼叫一次 OnUpdateRecord 事件让程师来控制如何把数据更新回去。下面是 OnUpdateRecord 事件的宣告原型：

```
  TFDUpdateRecordEvent = procedure (ASender: TDataSet; ARequest:
TFDUpdateRequest;
    var AAction: TFDErrorAction; AOptions: TFDUpdateRowOptions) of
object;
```

下面的表格说明了 OnUpdateRecord 事件参数的意义：

参数	说明
ASender: TDataSet	ASender 代表要进行异动资的 TFDQuery 组件
ARequest: TFDUpdateRequest	代表数据异动的种类，是新增，修改，删除还是

	其他的异动
var AAction: TFDErrorAction	代表发生异动错误时要采取的行动
AOptions: TFDUpdateRowOptions	代表要进行这笔数据异动时要采取的额外选项

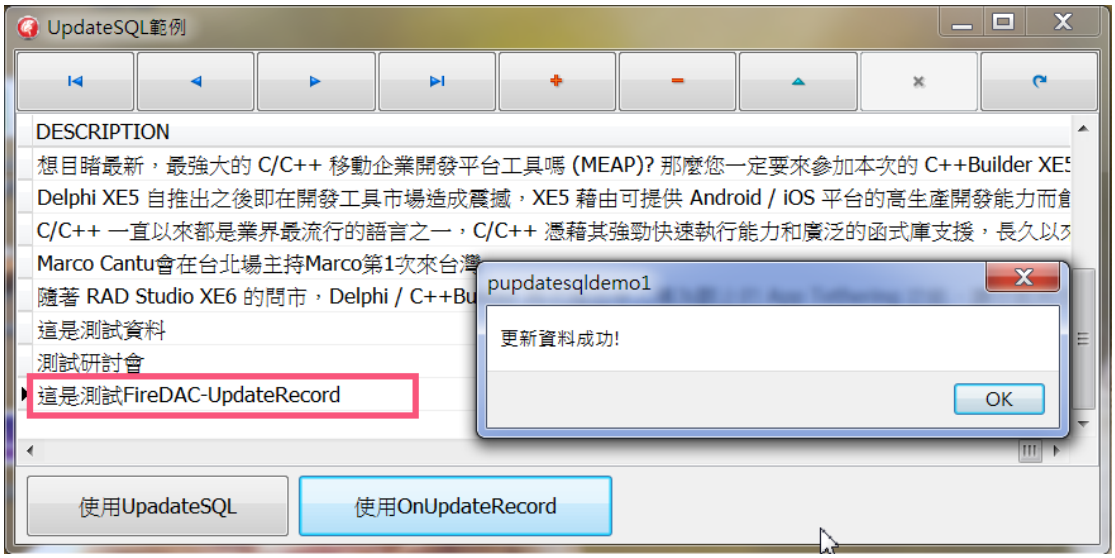
所以如果要使用 OnUpdateRecord 事件来进行和上一小节一样的更新动作，我们可以使用如下的程序代码：

```
001   procedure TForm1.Button2Click(Sender: TObject);
002   var
003     NumErrors: Integer;
004   begin
005     SeminarsTable.OnUpdateRecord :=
SeminarsTableUpdateRecord;
006     SeminarsTable.UpdateObject := nil;
007     NumErrors := SeminarsTable.ApplyUpdates(0);
008     if NumErrors = 0 then
009     begin
010       SeminarsTable.CommitUpdates;
011       ShowMessage('更新数据成功!');
012     end
013     else
014       ShowMessage('产生 ' + IntToStr(NumErrors) + ' 个更新错误');
015   end;
016
017   procedure TForm1.SeminarsTableUpdateRecord(ASender:
TDataSet;
018     ARequest: TFDUpdateRequest; var AAction: TFDErrorAction;
019     AOptions: TFDUpdateRowOptions);
020   begin
021     if (ARequest = arUpdate) then
022     begin
023       FDUpdateSQL1.DataSet := SeminarsTable;
024       FDUpdateSQL1.Apply(ARequest, AAction, AOptions);
025       AAction := TFDErrorAction.eaApplied;
026     end;
027   end;
```

首先 005 行先设定 SeminarsTable 的 OnUpdateRecord 事件处理函式再于 007 行呼叫 ApplyUpdates 方法让 OnUpdateRecord 事件处理函式 SeminarsTableUpdateRecord 来更新数据。

017 行开始的 SeminarsTableUpdateRecord 先于 021 行判断目前的异动是不是更新的动作，如果是的话就进行 023~025 行呼叫 FDUpdateSQL1 的 Apply 方法真正把数据更新回 SEMINARS 数据表再于 025 行设定 AAction 参数为 TFDErrorAction.eaApplied 代表更新成功，

现在再次执行范例程序，更改同样一笔数据如下所示，再点选”使用 OnUpdateRecord”按钮就可以看到也可以成功把数据更新回后端。



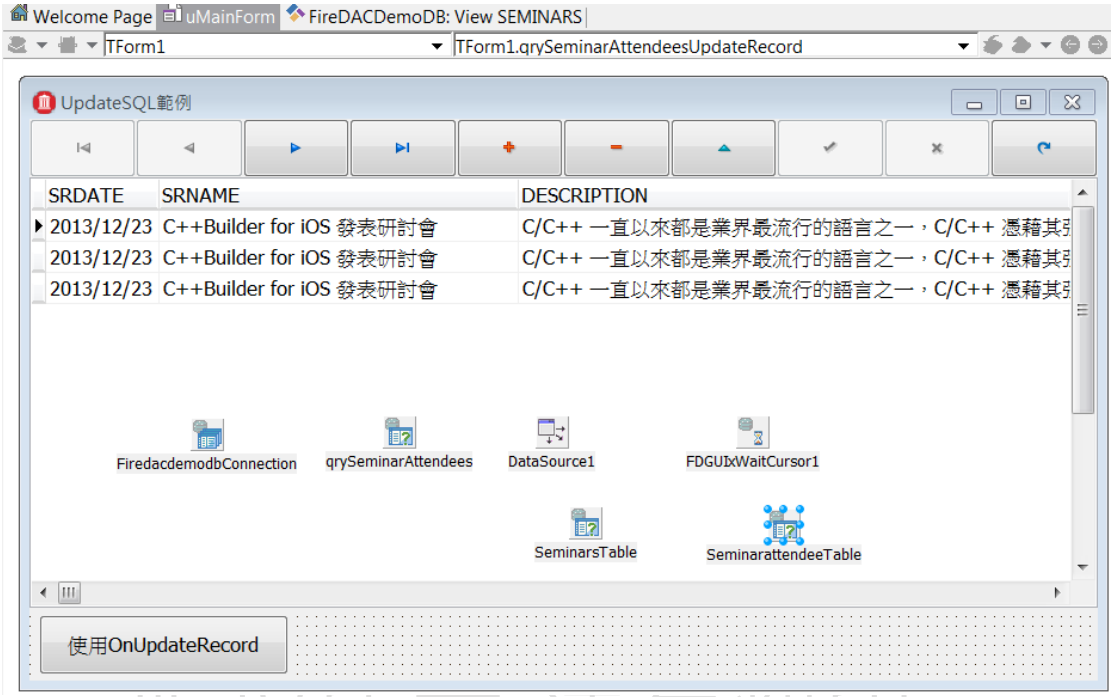
4-3-4 使用 TUpdateSQL 组件处理复杂数据更新

如果数据库应用程序使用了 JOIN SQL 显示数据并允许用户修改数据的话，那么要如何做到？例如下面的 SQL 命令从 SEMINARS 和 SEMINARATTENDEE 这 2 个数据表中撷取数据，那如何能让 FireDAC 的应用程序可以修改并把数据更新回后端？

```
Select S.SRDATE, S.SRNAME, S.DESCRIPTION, A.NAME, A.Email, A.Phone
from SEMINARS S, SEMINARATTENDEE A where S.SRID = A.SRID
```

一般来说这并不容易，因为使用 Join SQL 取得的数据都是只读的，而且 TFDQuery 也无法自动把这种数据更新回后端多个数据表，那么是否可用 Update SQL 的功能克服这个需求？答案是可以的，Update SQL 不但可以提供 UI 修改 Join SQL 取得的资料，也可以让程序员使用程序代码这种资料更新回后端多个数据表。

例如下面显示的第 2 个 Update SQL 的范例程序: pUpdateSQLDemo2.dproj，其主窗体中的 qrySeminarAttendees 组件就使用了上面的 Join SQL 从 SEMINARS 和 SEMINARATTENDEE 这 2 个数据表中撷取数据并显示在 DBGrid 中：



qrySeminarAttendees 组件设定了其 CachedUpdates 特性值为 True 以使用 Update SQL 功能：

但如果现在您执行此范例应用程序会发现在 DBGrid 中只能修改 SEMINARS 数据表的字段数据：SRDATE，SRNAME 和 DESCRIPTION，而无法修改 SEMINARATTENDEE 数据表的字段数据：Name，Email 和 Phone。为什么？

如同前面说的 Join 的数据是只读的，打开 CachedUpdates 功能也只能让用户可修改主数据表的数据，也就是 SEMINARS，而无法修改从数据表，也就是 SEMINARATTENDEE。因此要允许可修改从数据表，除了开启 CachedUpdates 功能程序员必须再开启 TFDQuery 组件的 UpdateOptions|UpdateNonBaseFields 特性：

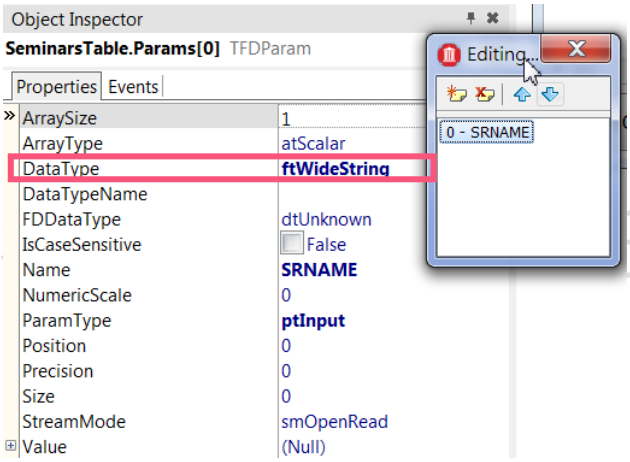
一旦设定 qrySeminarAttendees 的 UpdateOptions|UpdateNonBaseFields 特性为 True 之后现在就可以在 DBGrid 中修改数据了。

为了稍后简化说明的程序代码，因此让我们设定要修改并且更新 qrySeminarAttendees 中的 DESCRIPTION 和 NAME 这 2 个字段。DESCRIPTION 是属于 SEMINARS 数据表，而 NAME 则是属于 SEMINARATTENDEE 数据表，这也代表我们需要同时更新 2 个数据表。

在这个范例中使用了 2 个 TFDQuery 组件来更新数据回 SEMINARS 和 SEMINARATTENDEE 数据表，其中的 SeminarsTable 组件使用下面的 SQL 命令：

```
SELECT * FROM SEMINARS where SRNAME = :SRNAME
```

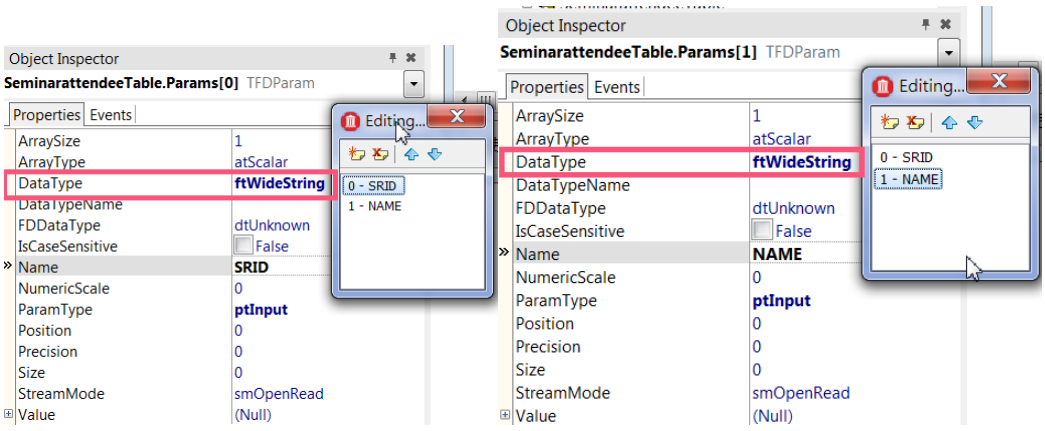
SeminarsTable 组件会使用 SRNAME 字段到 SEMINARS 数据表中搜寻被异动的数据然后更新它的 DESCRIPTION 域值。由于 SeminarsTable 组件使用了动态参数，因此请记得要设定动态参数的数据型态，由于 SRNAME 字段是字符串字段，因此我们设定它的数据型态是 ftWideString：



SeminarattendeeTable 组件使用下面的 SQL 命令更新 NAME 字段：

```
SELECT * FROM SEMINARATTENDEE where SRID = :SRID and NAME = :NAME
```

它使用了 2 个动态参数，因此我们也需要设定这 2 个动态参数的数据型态如下：



完成了这些设定之后就可以开始撰写程序代码来进行实际的资料更新工作了。首先在主窗体的"使用 OnUpdateRecord"按钮中撰写如下的程序代码使用 OnUpdateRecord 事件来更新数据：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NumErrors : Integer;
begin
  qrySeminarAttendees.OnUpdateRecord :=
qrySeminarAttendeesUpdateRecord;
  qrySeminarAttendees.UpdateObject := nil;
  NumErrors := qrySeminarAttendees.ApplyUpdates(0);
  if NumErrors = 0 then
  begin
    qrySeminarAttendees.CommitUpdates;
    ShowMessage('更新数据成功!');
  end
  else
    ShowMessage('产生 ' + IntToStr(NumErrors) + ' 个更新错误');
end;
```


在 OnUpdateRecord 事件处理函式中 006 行先判断是否为修改的动作，如果是的话就在 007 行呼叫 UpdateSeminars 方法更新数据回 SEMINARS 数据表，008 行呼叫 UpdateSeminarAttendees 方法更新数据回 SEMINARATTENDEE 数据表：

```
001   procedure TForm1.qrySeminarAttendeesUpdateRecord(ASender:
TDataSet;
002     ARequest: TFDUpdateRequest; var AAction: TFDErrorAction;
003     AOptions: TFDUpdateRowOptions);
004   begin
005     if (ARequest = arUpdate) then
006     begin
007       UpdateSeminars(ASender);
008       UpdateSeminarAttendees(ASender);
009       AAction := TFDErrorAction.eaApplied;
010     end;
011   end;
```

UpdateSeminars 方法先在 006 行呼叫 FindSeminar 方法使用 SeminarsTable 组件在 SEMINARS 数据表中搜寻被修改的数据，如果找到的话就把修改的 DESCRIPTION 域值更新回 SEMINARS 数据表：

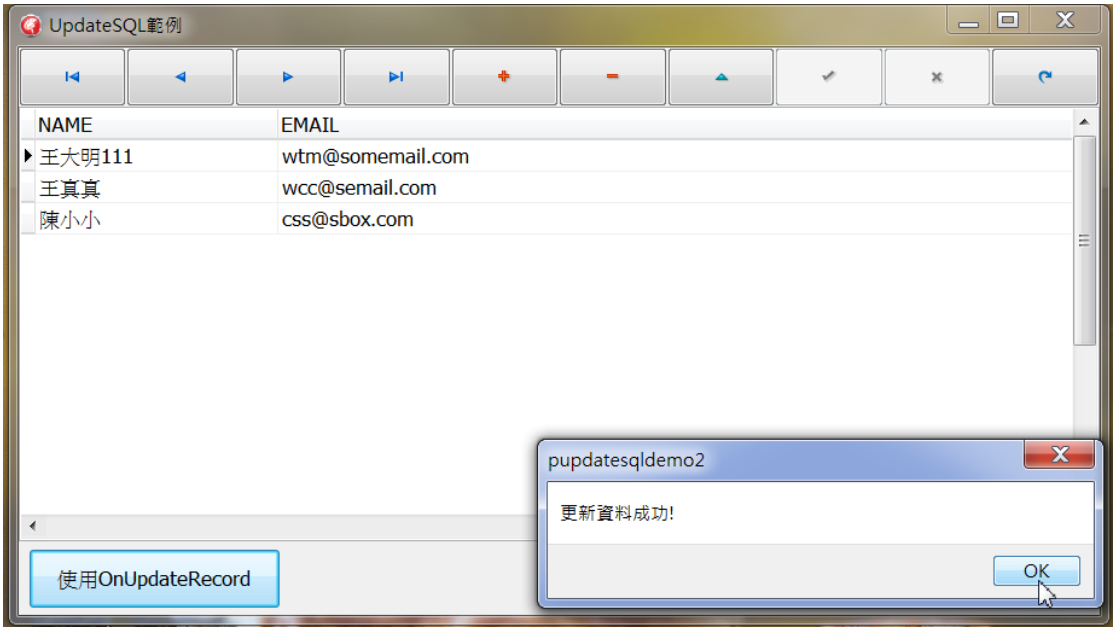
```
001   procedure TForm1.UpdateSeminars(ASender: TDataSet);
002   var
003     sName : String;
004   begin
005     sName := ASender['SRNAME'];
006     if (FindSeminar(sName)) then
007     begin
008       if (not
VarIsNull(ASender.FieldByName('DESCRIPTION').Value)) then
009       begin
010         SeminarsTable.Edit;
011         SeminarsTable.FieldByName('DESCRIPTION').Value :=
ASender.FieldByName(
012           'DESCRIPTION').Value;
013         SeminarsTable.Post;
014       end;
015     end;
016   end;
```


UpdateSeminarAttendees 方法在 006 行呼叫 FindSeminar 方法使用 SeminarsTable 组件在 SEMINARS 数据表中搜寻被修改的数据，再藉由链接 SEMINARS 和 SEMINARATTENDEE 数据表和 SRID 域值在 008 行呼叫 FindAttendee 在 SEMINARATTENDEE 数据表中找到被修改的数据再把 NAME 域值更新：

```
001   procedure TForm1.UpdateSeminarAttendees(ASender: TDataSet);
002   var
003     sName : String;
004   begin
005     sName := ASender['SRNAME'];
006     if (FindSeminar(sName)) then
007     begin
008       if
(FindAttendee(SeminarsTable.FieldByName('SRID').AsString,
ASender.FieldByName('NAME').OldValue)) then
009       begin
```

```
010         if (not VarIsNull(ASender.FieldByName('NAME').Value))
then
011         begin
012           SeminarattendeeTable.Edit;
013           SeminarattendeeTable.FieldByName('NAME').Value :=
ASender.FieldByName(
014               'NAME').Value;
015           SeminarattendeeTable.Post;
016         end;
017       end;
018     end;
019   end;
```

现在执行此范例程序就可以看到如下的画面，我们不但可以在 DBGrid 中修改数据，也可以把 DESCRIPTION 和 NAME 字段的数据分别更新回 SEMINARS 和 SEMINARATTENDEE 数据表。我们藉由 OnUpdateRecord 功能完成了一般状况下无法做到的工作。



在这个范例中我们是使用 2 个 TFDQuery 组件更新数据，当然程序员也可以直接使用 Update SQL 来更新资料。

不过上面的程序代码虽然可以正常工作，但严格来说并不正确，为什么？再仔细回去看看前面的 Button1Click 事件处理函式，它在呼叫了 qrySeminarAttendees 的 ApplyUpdates 方法后只呼叫 qrySeminarAttendees 的 CommitUpdates 方法确定数据更新动作完成。但在这个范例中真正进行数据更新的动作是由 SeminarsTable 和 SeminarattendeeTable 组件进行的，而且 qrySeminarAttendees 并不能更新数据(它

是只读的)。因此在上面的程序代码中在内定的模式中 SeminarsTable 和 SeminarattendeeTable 组件分别会启动一个数据库交易，因此可能会发生 SeminarsTable 更新成功 SeminarattendeeTable 失败或是反过来的情形。

因此我们需要更改 Button1Click 事件处理函式，在 qrySeminarAttendees 呼叫 ApplyUpdates 方法之前先手动启动数据库交易，把 SeminarsTable 和 SeminarattendeeTable 组件更新数据的动作置于一个相同的数据库交易环境中：

```
001   procedure TForm1.Button1Click(Sender: TObject);
002   var
003     NumErrors : Integer;
004   begin
005     qrySeminarAttendees.OnUpdateRecord :=
qrySeminarAttendeesUpdateRecord;
006     qrySeminarAttendees.UpdateObject := nil;
007     FiredacdemodbConnection.StartTransaction;
008     NumErrors := qrySeminarAttendees.ApplyUpdates(0);
009     if NumErrors = 0 then
010     begin
011       qrySeminarAttendees.CommitUpdates;
012       FiredacdemodbConnection.Commit;
013       ShowMessage('更新数据成功!');
014     end
015     else
016     begin
017       FiredacdemodbConnection.Rollback;
018       ShowMessage('产生 ' + IntToStr(NumErrors) + ' 个更新错误');
019     end;
020   end;
```


如此一来 SeminarsTable 和 SeminarattendeeTable 组件就可保证可同时更新成功或是同时恢复更新。

但由于我们只是藉由 qrySeminarAttendees 触发 OnUpdateRecord 事件，数据并不是由 qrySeminarAttendees 更新回后端，因此最后我们可以再修改 Button1Click 如下，使用例外处理程序代码 try...execept 来判断是否是确认更新成功或是恢复数据更新的动作：

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
var
  NumErrors : Integer;
begin
  qrySeminarAttendees.OnUpdateRecord :=
qrySeminarAttendeesUpdateRecord;
  qrySeminarAttendees.UpdateObject := nil;
  FiredacdemodbConnection.StartTransaction;
  try
    NumErrors := qrySeminarAttendees.ApplyUpdates(0);
    qrySeminarAttendees.CommitUpdates;
    FiredacdemodbConnection.Commit;
    ShowMessage('更新数据成功!');
  except
    FiredacdemodbConnection.Rollback;
    ShowMessage('产生 ' + IntToStr(NumErrors) + ' 个更新错误');
  end;
end;
```

藉由前面的 2 个范例您应该就可以使用 FireDAC 的 Update SQL 功能来更新一个或是多个数据表的数据了。

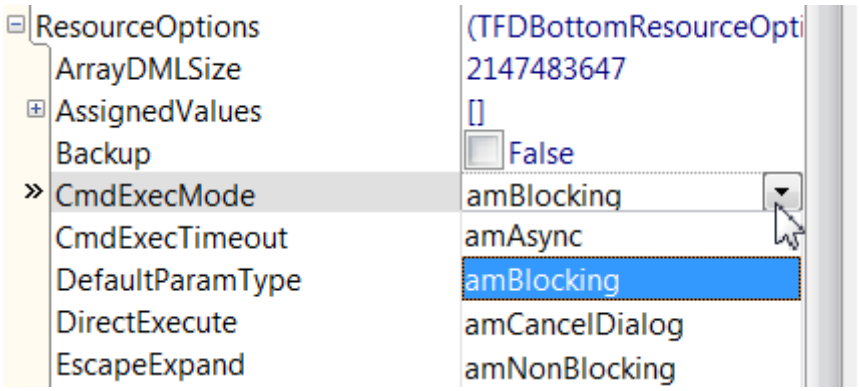
4-4 异步处理数据

FireDAC 和 BDE/dbExpress 不同的地方之一就是 FireDAC 允许开发人员使用异步的方式来处理数据，虽然 FireDAC 在内定上还无法同时支持使用多个线程来查询和处理数据，但 FireDAC 至少在内定功能中已经建立了异步处理数据的能力，让开发人员可以把 UI 和数据处理的动作分离以避免要长时间处理数据时会暂停 UI 的反应。

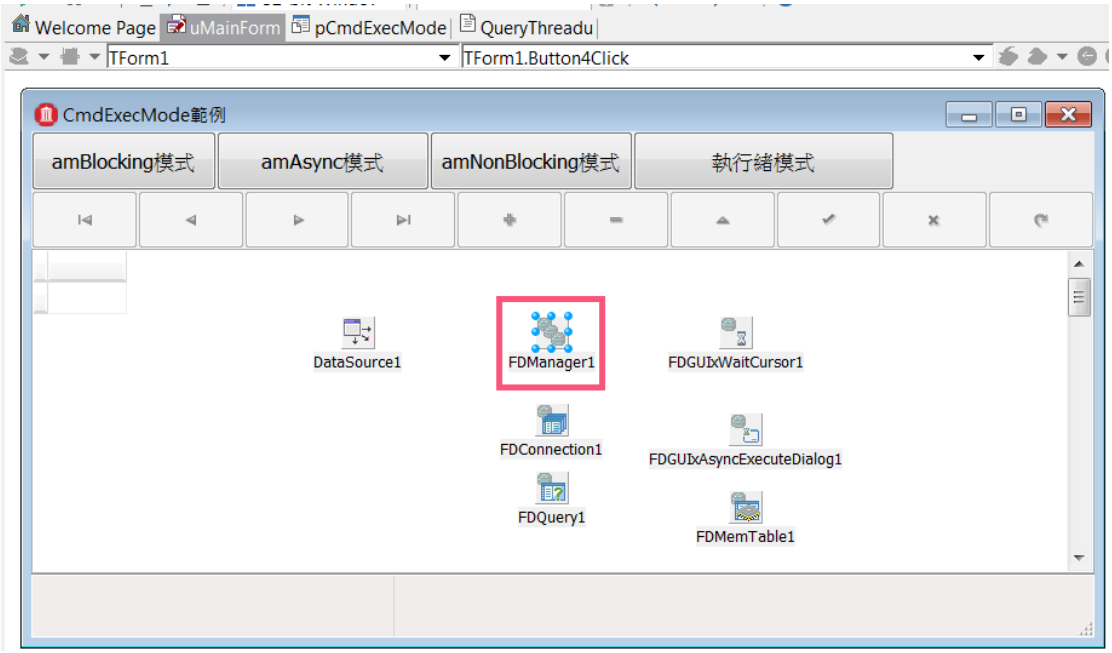
FireDAC 的数据处理功能提供了 4 种模式，下面的表格说明了这 4 种不同的方式：

模式	说明
amBlocking	呼叫线程和 UI 都会暂停一直到数据处理动作完成
amNonBlocking	呼叫线程会暂停一直到数据处理动作完成,但 UI 不会被暂停
amCancelDialog	呼叫线程和UI 都会暂停一直到数据处理动作完成.但FireDAC提供一个对话盒可取消数据处理动作
amAsync	呼叫线程和 UI 都不会暂停. 呼叫数据处理动作之后会立刻回到呼叫线程继续工作. 数据处理动作在后端执行.

要使用 FireDAC 的异步数据处理功能，程序员必须搭配使用 TFDManager 组件，再设定 TFDQuery 组件的 ResourceOptions.CmdExecMode 特性值：



下面的范例程序展示了如何使用这 4 种模式，在使用不同的模式时程序员需要注意一些事情，那就是 UI 和线程的关系，让我们使用这个范例程序的程序代码来说明，



在范例程序的主窗体中可以看到必须使用 TFDManager 组件，接着看看下面使用”amBlocking 模式”的实作程序代码。由于 amBlocking 模式其实就是使用主线程去执行数据处理的工作，因此主窗体中的数据感知组件可以直接链接到进行数据处理工作的 FDQuery1 组件：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  FDQuery1.DisableControls;
  lStart := GetTickCount;
```

```
  try
    DataSource1.DataSet := FDQuery1;
    FDQuery1.AfterOpen := nil;
    FDQuery1.ResourceOptions.CmdExecMode := amBlocking;
    FDQuery1.Open;
  finally
    lEnd := GetTickCount;
    FDQuery1.EnableControls;
  end;
  ShowRunTime(lEnd - lStart);
end;
```

但在下面的 amNonBlocking 模式中，我们再最在 FDQuery1 进行数据处理工作之前先切断 FDQuery1 和数据感知组件的链接，也就是要设定连结到 FDQuery1 的 DataSource1 的 DataSet 特性值为 Nil：

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  lStart := GetTickCount;
  FDQuery1.Close;
  FDQuery1.DisableControls;
  DataSource1.DataSet := Nil;
  FDQuery1.AfterOpen := QueryAfterOpen;
  FDQuery1.ResourceOptions.CmdExecMode := amNonBlocking;
  FDQuery1.Open;
  lEnd := GetTickCount;
  ShowRunTime(lEnd - lStart);
end;
```


如果是使用 amAsync 模式那么是一样要先切断 FDQuery1 和数据感知组件的链接，如下面 006 行所示，这是因为使用 amAsync 模式时，处理数据工作的线程和主线程不同，因为在数据工作的线程执行时，必须和 UI 隔离。

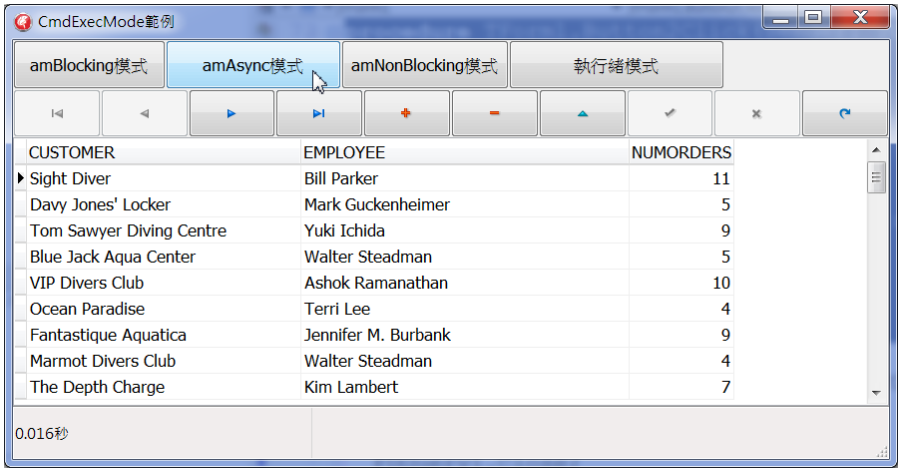
```
001   procedure TForm1.Button2Click(Sender: TObject);
002   begin
003     lStart := GetTickCount;
004     FDQuery1.Close;
005     FDQuery1.DisableControls;
006     DataSource1.DataSet := Nil;
```

```
007      FDQuery1.AfterOpen := QueryAfterOpen;
008      FDQuery1.ResourceOptions.CmdExecMode := amAsync;
009      FDQuery1.Open;
010      lEnd := GetTickCount;
011      ShowRunTime(lEnd - lStart);
012    end;
```

等到处理资料工作的线程执行完毕后再链接 UI 的数据感知组件，因此在才 FDQuery1 的因 On AfterOpen 事件中执行设定 DataSource1 的 DataSet 特性值回 FDQuery1：

```
procedure TForm1.QueryAfterOpen(DataSet: TDataSet);
begin
  DataSource1.DataSet := FDQuery1;
  DataSource1.DataSet.EnableControls;
end;
```

例如下面的画面就是 amAsync 模式成功执行的结果：



最后程序员也可以使用 Delphi 的 TThread 类别使用多个不同的线程来执行不同的数据处理工作，例如下面是”线程模式”的实作程序代码，它建立了一个 TQueryThread 对象并把一个执行数据处理工作的 SQL 命令和一个 TFDMemTable 组件传递给它，再呼叫 Start 方法启动它：

```
procedure TForm1.Button4Click(Sender: TObject);
var
  QueryThread: TQueryThread;
begin
  DataSource1.DataSet := Nil;
  QueryThread := TQueryThread.Create(FDQuery1.SQL.Text,
```

```
FDMemTable1);
  QueryThread.FreeOnTerminate := True;
  QueryThread.OnTerminate := ThreadDone;
  lStart := GetTickCount;
  QueryThread.Start;
end;
```

当这个 TQueryThread 对象执行完毕之后再链接数据感知组件：

```
procedure TForm1.ThreadDone(Sender: TObject);
begin
  lEnd := GetTickCount;
  ShowRunTime(lEnd - lStart);
  DataSource1.DataSet := FDMemTable1;
end;
```

TQueryThread 类别是从 TThread 类别继承下来：

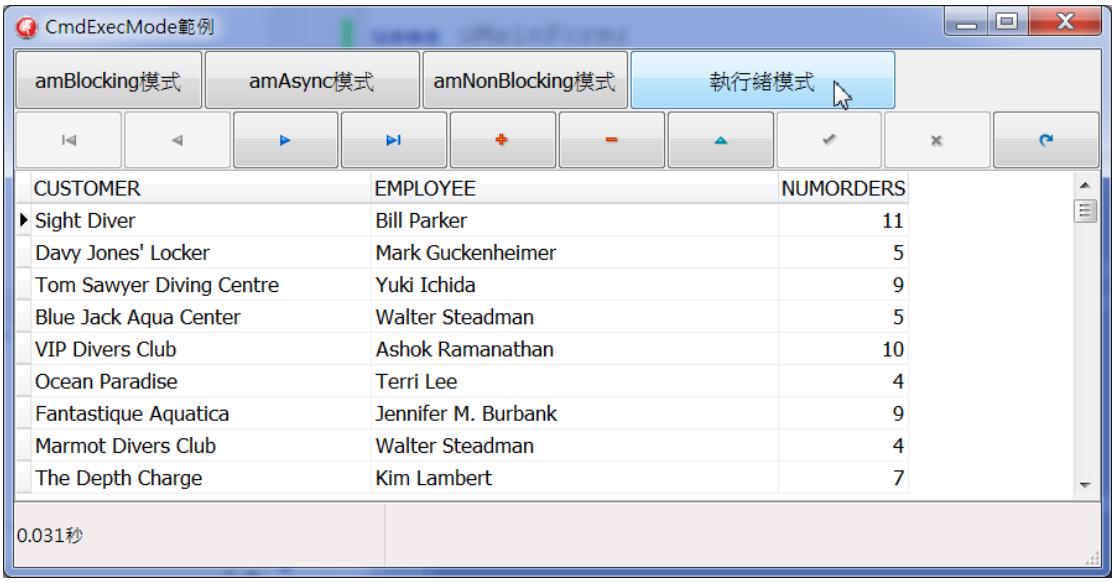
```
  TQueryThread = class(TThread)
```


它会自行建立一个 TFDConnection 组件再使用它链接到数据库然后执行传入的 SQL 命令，最后再把执行结果的数据集拷贝到传入的 TFDMemTable 组件中：

```
001   constructor TQueryThread.Create(SQLText: string;
FDMemTable: TFDMemTable);
002   begin
003     inherited Create(True);
004     FDConnection := TFDConnection.Create(nil);
005     FDConnection.ConnectionDefName := 'dbDemos';
006     FDConnection.LoginPrompt := False;
007     FDConnection.Open;
008     FDQuery := TFDQuery.Create(nil);
009     FDQuery.Connection := FDConnection;
010     FDQuery.SQL.Text := SQLText;
011     FDQuery.ResourceOptions.CmdExecMode := amBlocking;
012     FFDMemTable := FDMemTable;
013   end;
014
015   destructor TQueryThread.Destroy;
016   begin
```

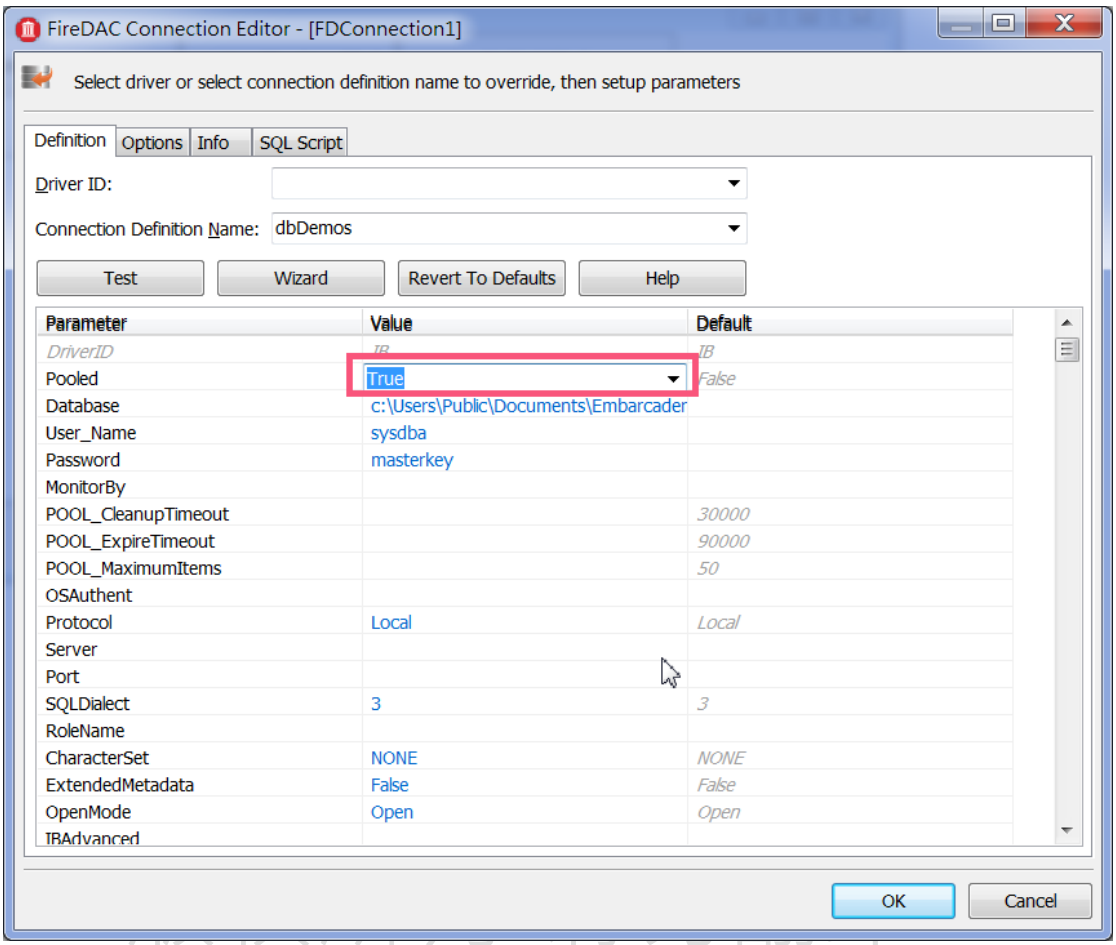
```
017      FDQuery.Free;
018      FDConnection.Free; //The connection will return to the
connection pool
019      inherited;
020    end;
021
022    procedure TQueryThread.Execute;
023    begin
024      FDQuery.Open;
025      Synchronize(procedure
026                  begin
027                    FFDMemTable.CloneCursor(FDQuery, True);
028                  end);
029    end;
```

下面的画面就是线程模式成功执行的结果：



不过程序员一定要记得如果使用线程模式使用多个线程执行数据处理的工作的话，就一定要开启数据库的链接池功能以避免建立过多的数据库链接。

例如下面即显示了这个范例程序开启了使用的 InterBase 的链接池功能：



4-5 结论

本章的内容主要是说明如何使用 FireDAC 使用一些进阶的技巧来处理数据，这些技巧在开发较为复杂的数据库应用程序时会非常的有帮助，正确使用这些技巧也能够让程序员开发出高执行效率的数据库应用程序，当然本章讨论的内容也都可以使用在移动平台中，在稍后的章节中会讨论更多使用其他 FireDAC 组件处理数据的技巧。

第5章 FireDAC 更多的功能

在前面的章节中已经讨论了许多 FireDAC 的功能，在本章中将继续说明其他的功能，这些功能有的是在较新的 FireDAC 版本才出现的，这些新功能在使用上并不困难但却非常的实用。

5-1 批处理

在传统的数据存取中如果我们需要执行 2 个 SQL 命令，例如下面的 SQL

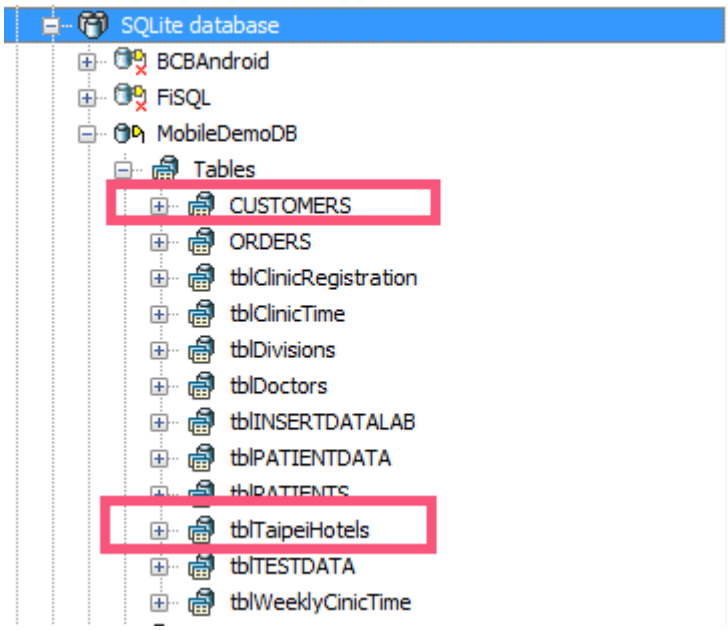
```
SELECT * FROM tblTaipeiHotels;
select * from   CUSTOMERS;
```

在 BDE/dbExpress 中我们需要使用 2 个 Query 组件并执行 2 次以取得 tblTaipeiHotels 和 CUSTOMERS 这 2 个数据表的资料。不过在 FireDAC 中我们可以利用所谓的 Batch SQL 功能一次就执行这 2 个 SQL 命令。

FireDAC 的 Batch SQL 功能可以让程序员使用一个 TFDQuery 组件一次执行多个 SQL 命令，这样做有几个好处，一是可以减少网络的来回次数，二是可以利用数据库的 Batch 执行功能以增加执行 SQL 的效率，三是可以减少客户端的资源。

要使用 FireDAC 的 Batch SQL 功能，程序员只需要把 SQL 命令都指定给 TFDQuery 组件的 SQL 特性，再设定 Active 特性值为 True 或是呼叫 Open 方法即可。当 Batch SQL 执行完毕之后后端会传回多个数据集对象，程序员可以呼叫 TFDQuery 的 NextRecordSet 方法从第 1 个数据集顺序访问每一个回传的数据集物件。

让我们使用一个简单的范例来说明，例如在下面的 SQLite 数据库中假释我们要存取 CUSTOMERS 和 tblTaipeiHotels 这 2 个数据表中的资料：



那可以使用下面的程序代码，先呼叫 OpenBatchSQL 方法使用 Batch SQL 一次从这 2 个数据表取得数据，再呼叫 CopyToMemoryTable 方法把回传的数据集对象在客户端指定给 TFDMemTable，最后再呼叫 DisplayData 显示数据：


```
procedure TForm1.btnRunBatchClick(Sender: TObject);
begin
  OpenBatchSQL;
  CopyToMemoryTable;
  DisplayData;
end;
```

OpenBatchSQL 方法先把执行 Batch SQL 的 TFDQuery 组件的 FetchOptions.AutoClose 特性值设定为 False，否则 TFDQuery 在内定上接到回传的第 1 个数据集对象之后就会舍弃其他回传的数据集对象。

接着把 2 个 SQL 命令指定给 TFDQuery 的 SQL 特值，执行 2 个 SQL 命令，最后呼叫 FetchAll 方法一次把 2 个数据表中的数据都存取回客户端：

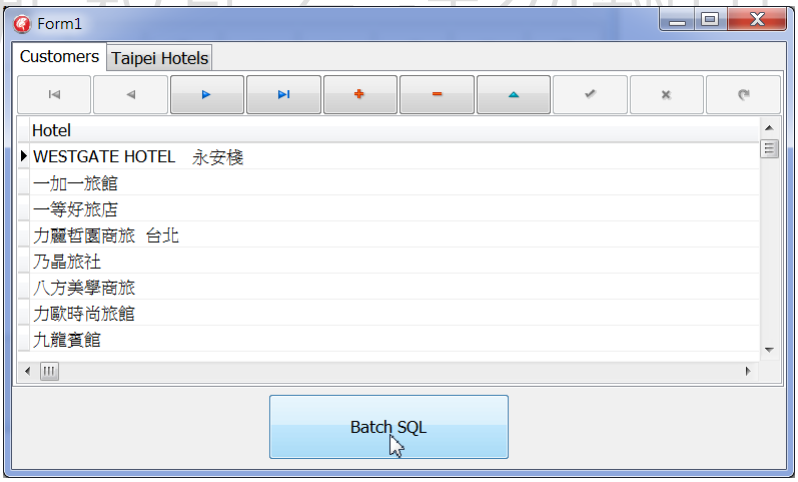
```
procedure TForm1.OpenBatchSQL;
begin
  qryGeneral.FetchOptions.AutoClose := False;
  qryGeneral.SQL.Text := 'SELECT * FROM tblTaipeiHotels; select *
from ' +
```

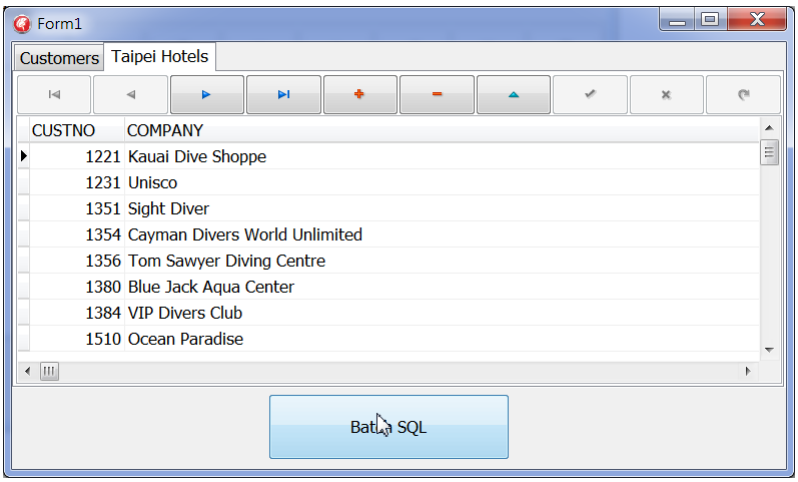
```
    'CUSTOMERS';
  qryGeneral.Active := True;
  qryGeneral.FetchAll;
end;
```

CopyToMemoryTable 方法先把第 1 个回传的数据集对象中的数据指定给 fdmtCustomers 这个 TFDMemTable 组件，呼叫 TFDQuery 组件的 NextRecordSet 方法取得下一个回传的数据集对象之后再指定给 fdmtTaipeiHotels 个 TFDMemTable 组件：

```
procedure TForm1.CopyToMemoryTable;
begin
  fdmtCustomers.Data := qryGeneral.Data;
  qryGeneral.NextRecordSet;
  fdmtTaipeiHotels.Data := qryGeneral.Data;
end;
```

最后我们只需要把 fdmtCustomers 和 fdmtTaipeiHotels 这 2 个 TFDMemTable 组件链接到数据感知组件即可把数据显示出来：





使用 Batch SQL 功能一个网络来回就可以存取到多个数据表的数据，不过并不是每个数据库都支持 Batch SQL 功能，下面的表格列出了支持 Batch SQL 功能的数据表：

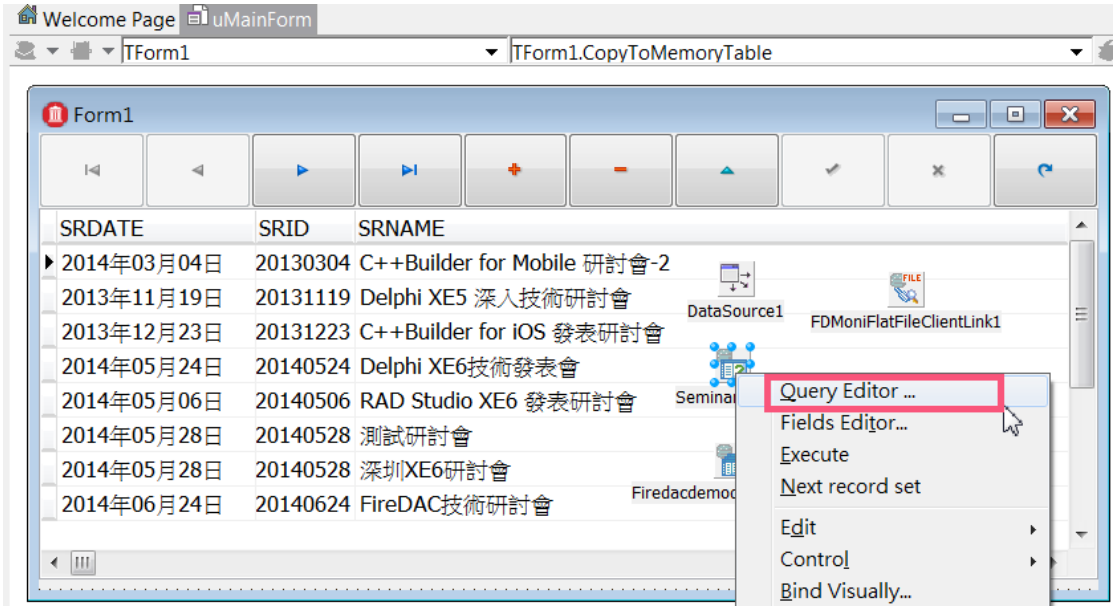
数据库	Batch SQL
IBM DB2	✓
Firebird	✓
Informix	✓
Microsoft SQL Server	✓
MySQL	✓
Oracle	✓
PostgreSQL	✓
SQLite	✓
SQL Anywhere	✓

5-2 控制数据的显示和更新

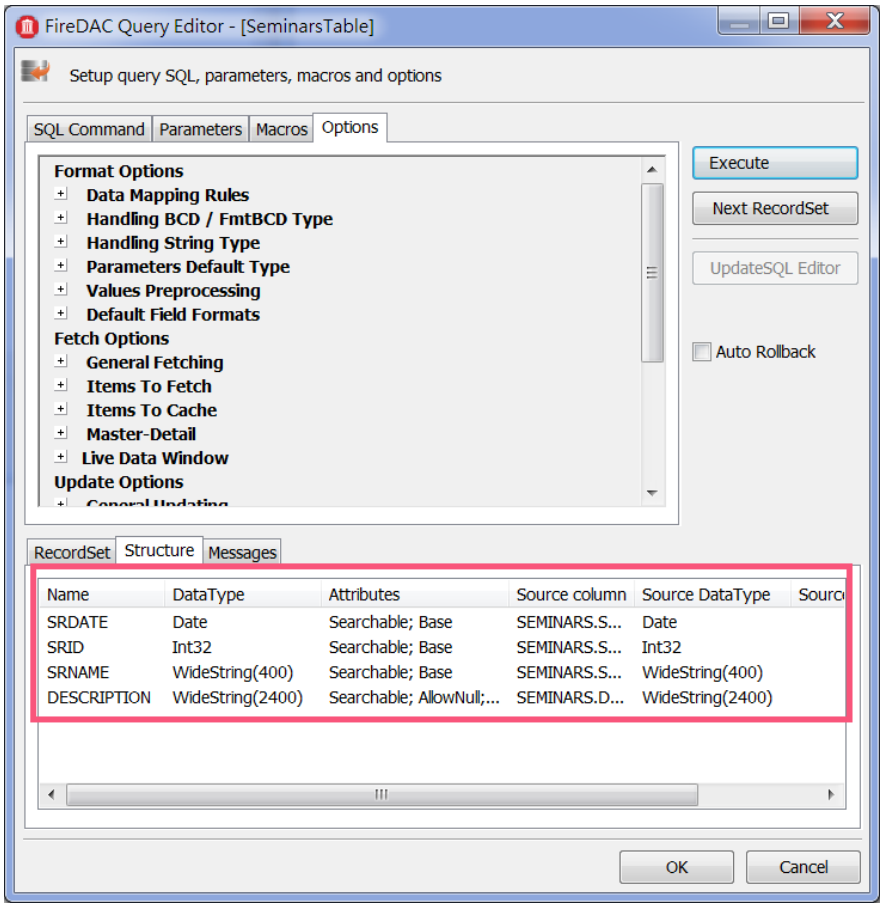
FireDAC 在数据的转换，对映和显示都有非常良好的弹性和控制力，能够让程序员对进行处理的数据有更大的控制力。在本小节中我们将这些功能进行说明。

在使用 TFDQuery 组件存取后端数据时 FireDAC 允许程序员对于如何处理资料有更进一步的控制，显示数据源的结构元数据，如何显示数据，如何对映数据，一直到如何存取数据和更新数据都可以进生深入的设定和控制。

程序员可以藉由右击 TFDQuery 组件再点选 Query Editor...选项启动组件编辑器，

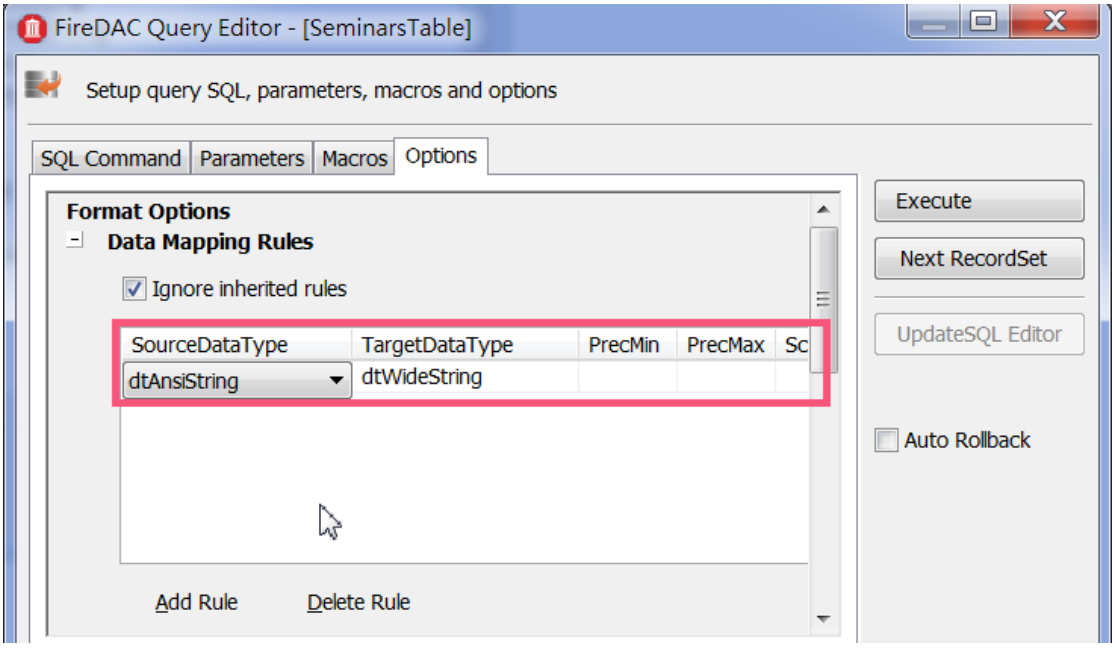


在 TFDQuery 组件编辑器的 Options 页次可以看到 TFDQuery 组件中的数据集对象的结构元数据：

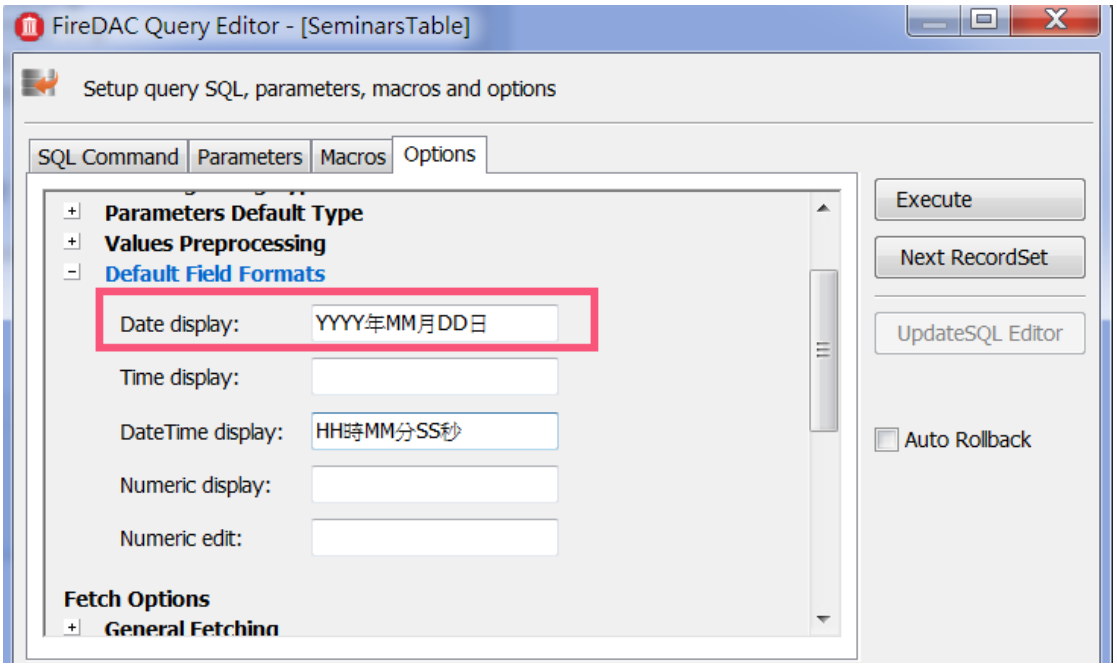


在 Options 页次中有许多重要的选项可设定，例如在 Format Options 中可以设定数据对映的规则以及数据显示的内定格式等。例如假设读者有一个数据表中的一个字段的数

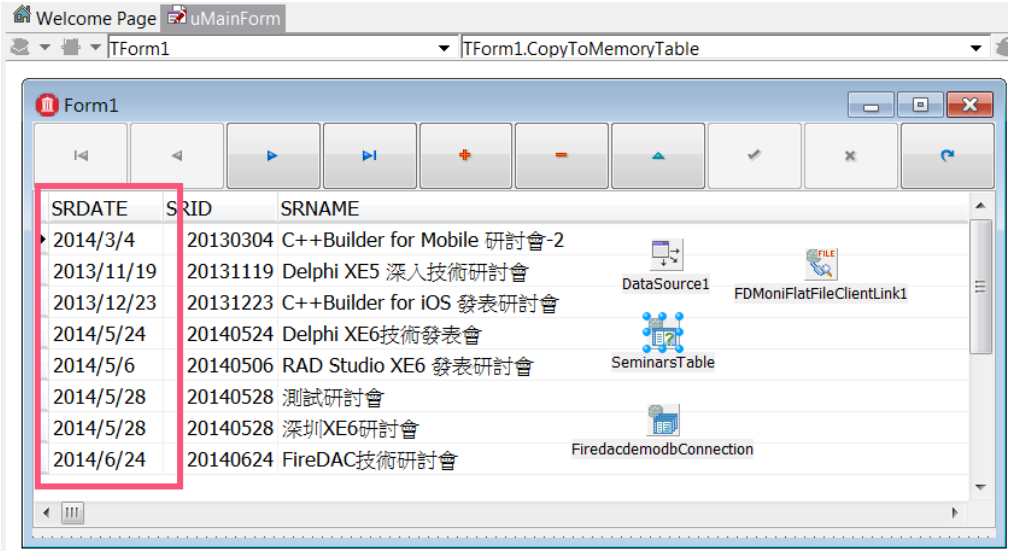
据当初是定义为 AnsiString，现在想以 WideString 显示和处理，那么就可以在 Data Mapping Rules 中定义 AnsiString 的数据要对映为 WideString。下图就是先勾选忽视内定的继承规则，并定期来源是 AnsiString 的数据要对映成 WideString：

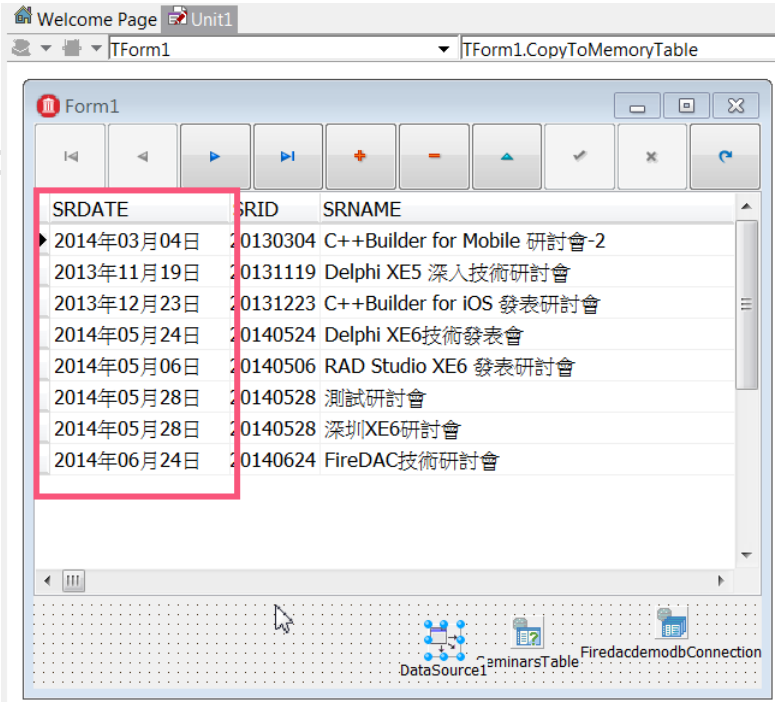


此外也可以在 Options 页次定义数据的内定显示格式，例如下面就是在 Options 页次中定期日期数据要以中文的"年月日"的格式来显示，时间数据则以"时分秒" 的格式来显示：



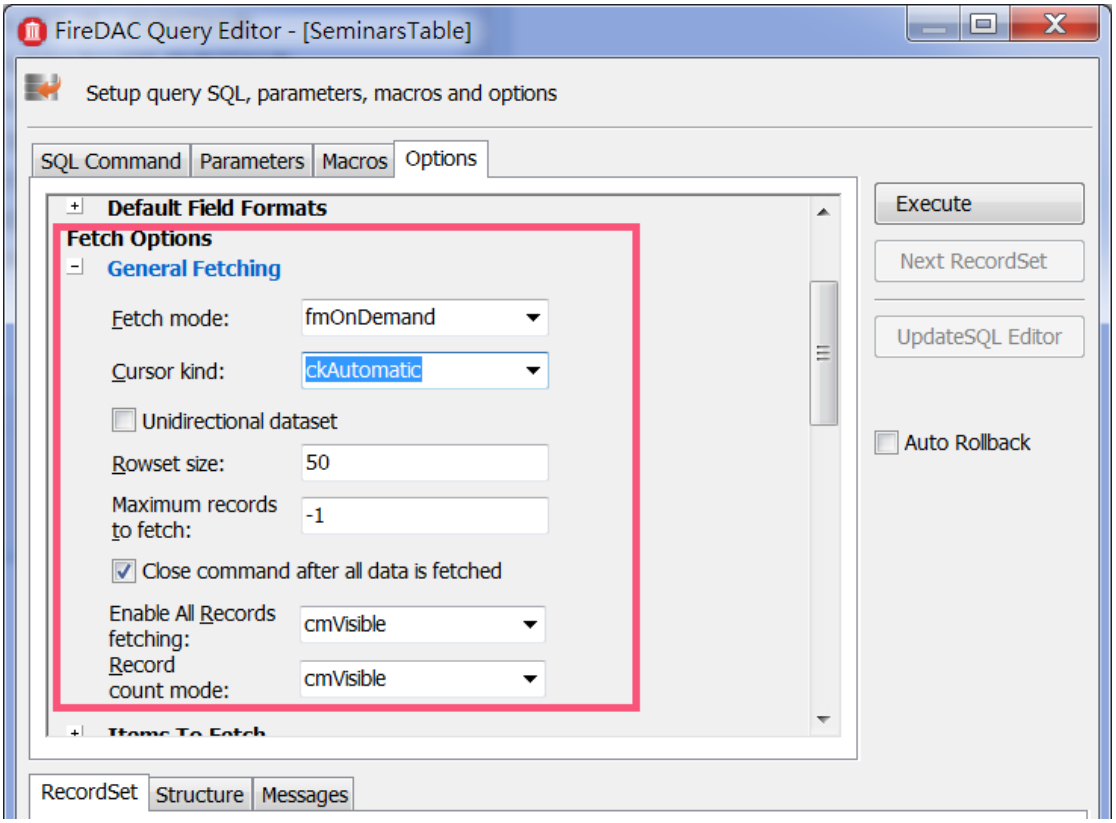
如上在 Options 页次定义好之后就可以看到原本 SRDATE 字段的数据格式改变成我们定义的”年月日”格式了：





在 Options 页次中最重要的应该是 Fetch Options 和 Posting Changes 了，因为这 2 者的设定都和执行效率有很大的关系。下图说明了 FireDAC 如何存取数据，例如如果应用程序只是依序读取数据，那设定下图中的 Unidirectional dataset 选项就可以增加执行速度。Rowset size 特性值控制了 FireDAC 一次从后端数据源存取的资料笔数，它的内定值 50 代表一次存取 50 笔，因此如果后端数据表中有 1000 笔数据那就要用 20 次的网络来回取得所有数据，因此 Rowset 愈大网络来回次数就愈少。不过设定太大的 Rowset 特性值会让每一次取取的时间加长，因此程序员应该根据实际的执行环境设定

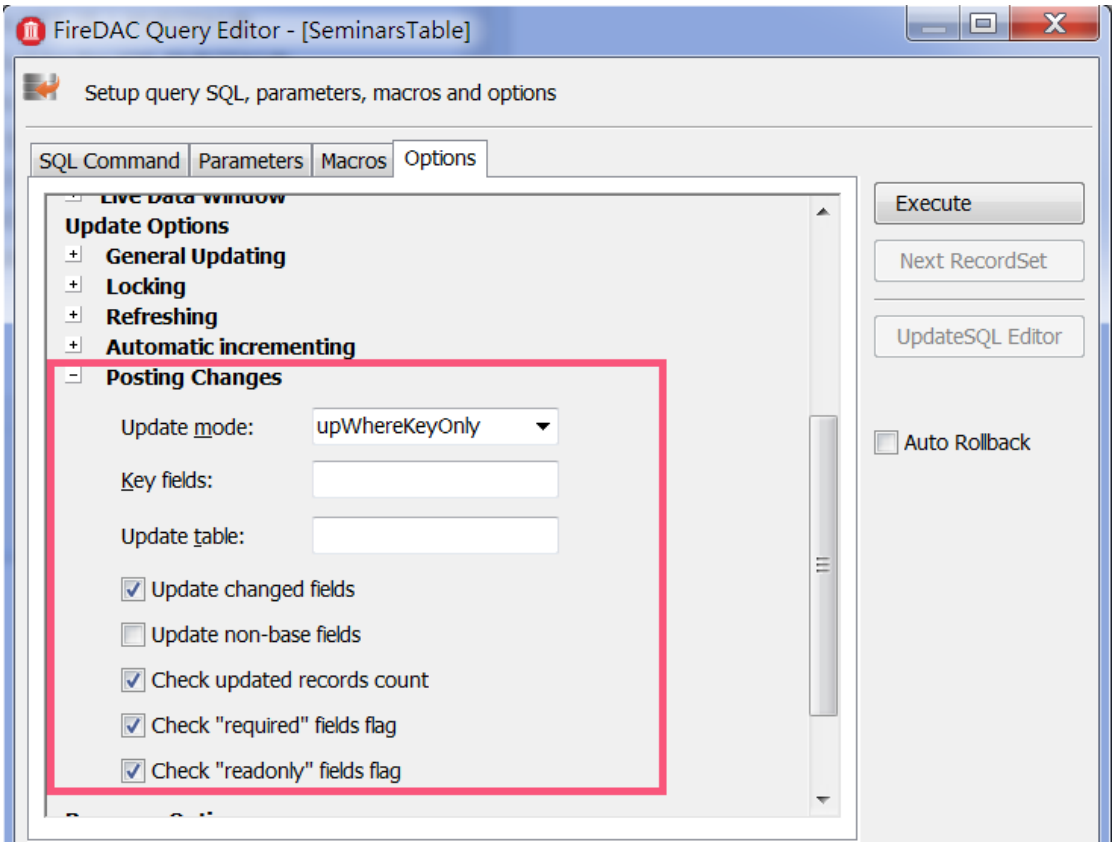
Rowset 特性值，例如在 PC 环境中也许应该加大 Rowset 特性值，在移动平台则应该稍微减小 Rowset 特性值。在 Options 页次中的 Rowset size 就是 TFDQuery 的 FetchOptions.RowsetSize 特性值。



但 FireDAC 一次存取多少数据笔数也受到 Fetch Mode 的影响，Options 页次中的 Fetch Mode 是就是 TFDQuery 的 FetchOptions.Mode 特性值，它的功能说明如下：

FetchOptions.Mode	说明
fmOnDemand	由 FetchOptions.RowsetSize 特性值自动控制存取数据
fmAll	一次就取得所有的数据，等于呼叫 TFDQuery 的 FetchAll 方法
fmManual	由程序员自己呼叫 TFDQuery 的 FetchNext 或 FetchAll 方法取得数据
fmExactRecsMax	一次取得 FetchOptions.RecsMax 笔的数据，如果取得的数据笔数和 FetchOptions.RecsMax 的特性值不同就会产生例外错误。FetchOptions.RecsMax 的内定值是-1，就和 fmAll 是一样的意思

当 FireDAC 拇把数据更新回后端时，如何找到后端要更新的数据再予以更新就是由 Options 页次中的 Posting Changes 选项来控制。在一般的应用中当使用者在前端异动了数据并实际更新回后端时，FireDAC 会使用被异动数据的键值在后端找到这笔数据之后再对它进行更新，这个规则就是由 TFDQuery 的 UpdateOptions.UpdateMode 控制，也就是 Options 页次中的 Update mode：

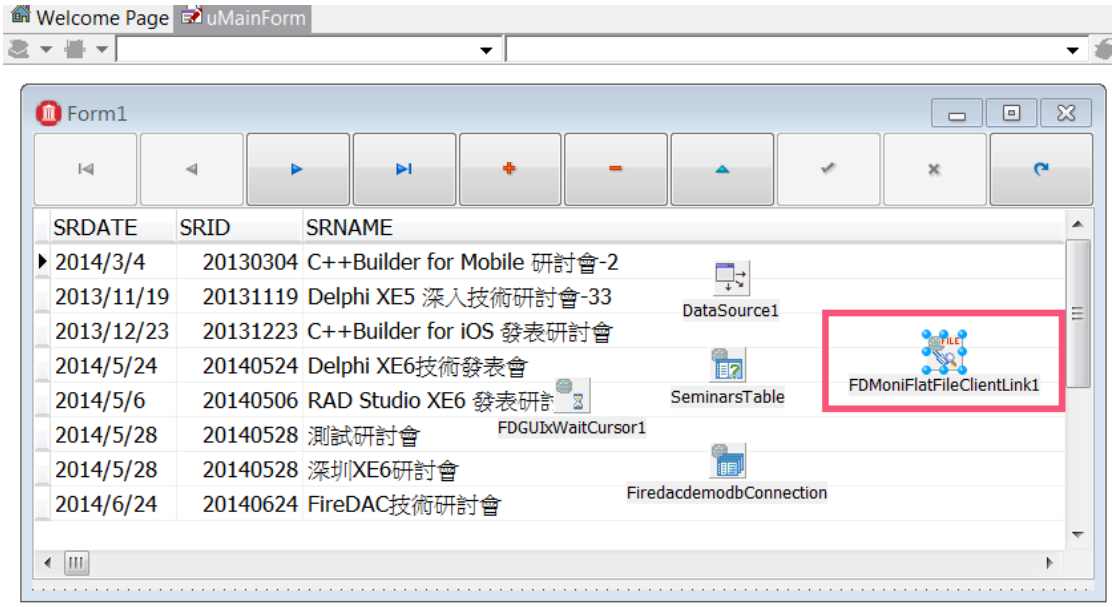


UpdateOptions.UpdateMode 可以有 3 个设定值，说明如下：

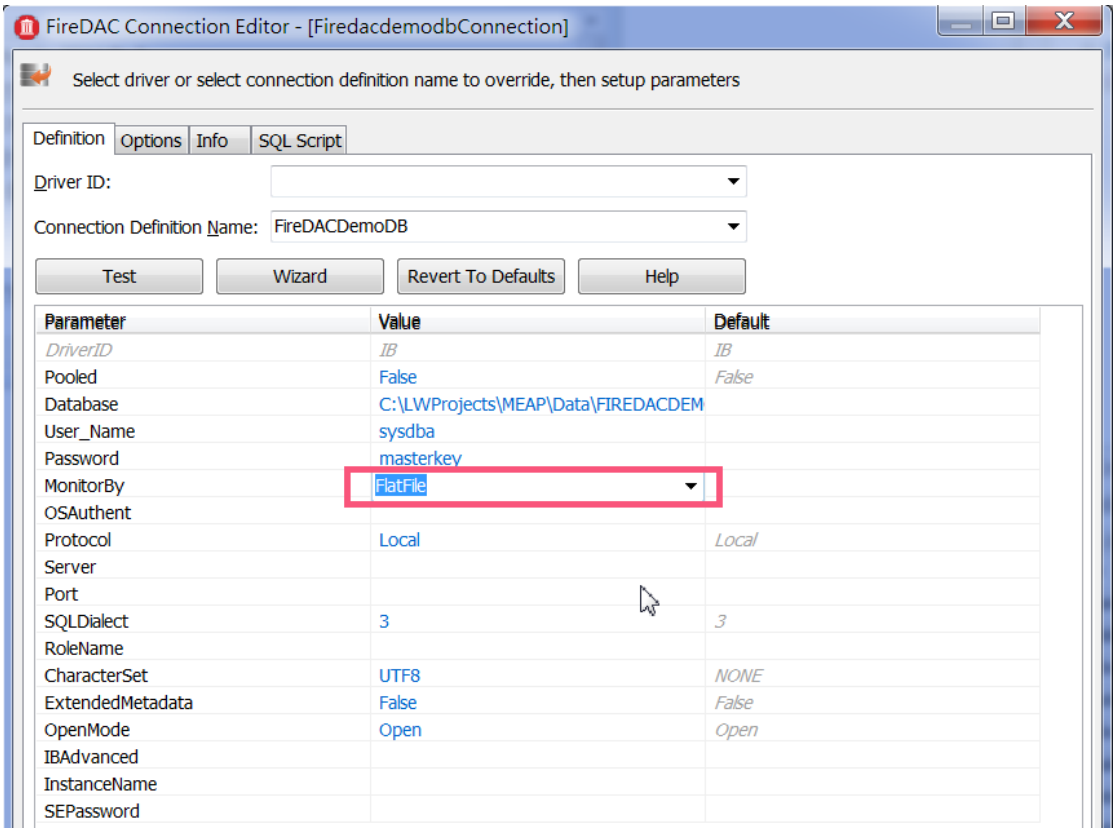
特性值	说明
upWhereAll	用所有字段的旧值来搜寻要更新的后端数据
upWhereChanged	用更新数据前的键值和更新字段的旧值来搜寻要更新的后端数据
upWhereKeyOnly	用只更新数据前的键值来搜寻要更新的后端数据

upWhereKeyOnly 是 UpdateOptions.UpdateMode 的内定值，让我们使用一个简单的范例来说明 UpdateOptions.UpdateMode 设定对于更新数据的影响。

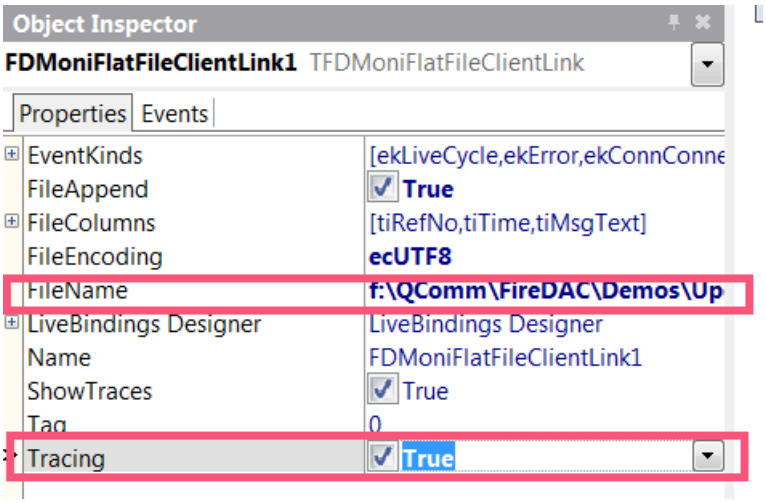
下面是 pUpdateModeDemo.dproj 项目，定使用 TFDQuery 存取 SEMINARS 数据表并且使用了 TFDMoniFlatFileClientLink 组件把前端 FireDAC 程序如何和后端数据库互动的命令都记录在一个文本文件中好人让我们检查 UpdateOptions.UpdateMode 如何影响数据更新的行为：



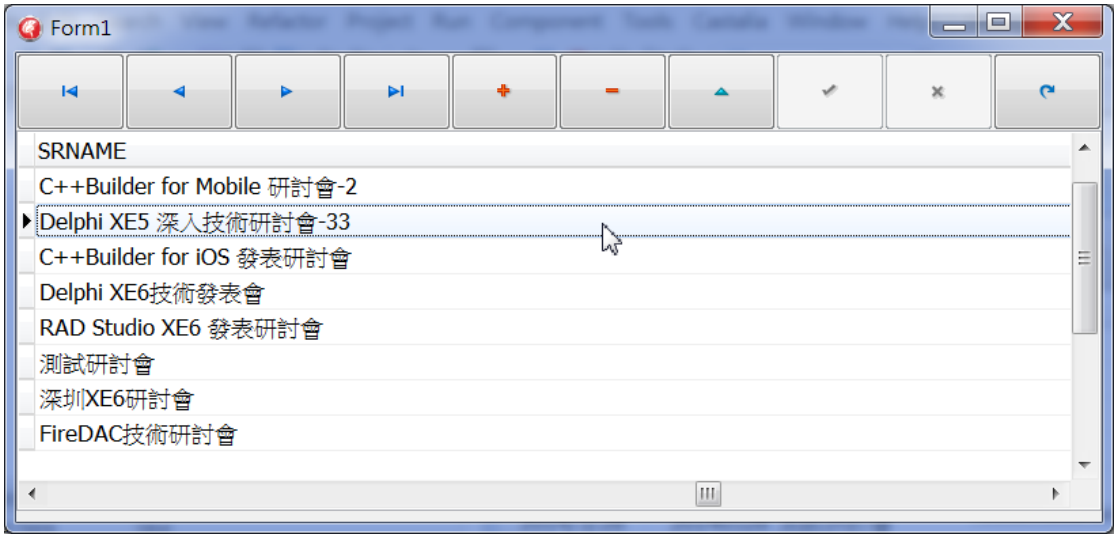
加入 TFDMoniFlatFileClientLink 组件后要在 TFDConnection 组件编辑器中设定 MonitorBy 为”FlatFile”：



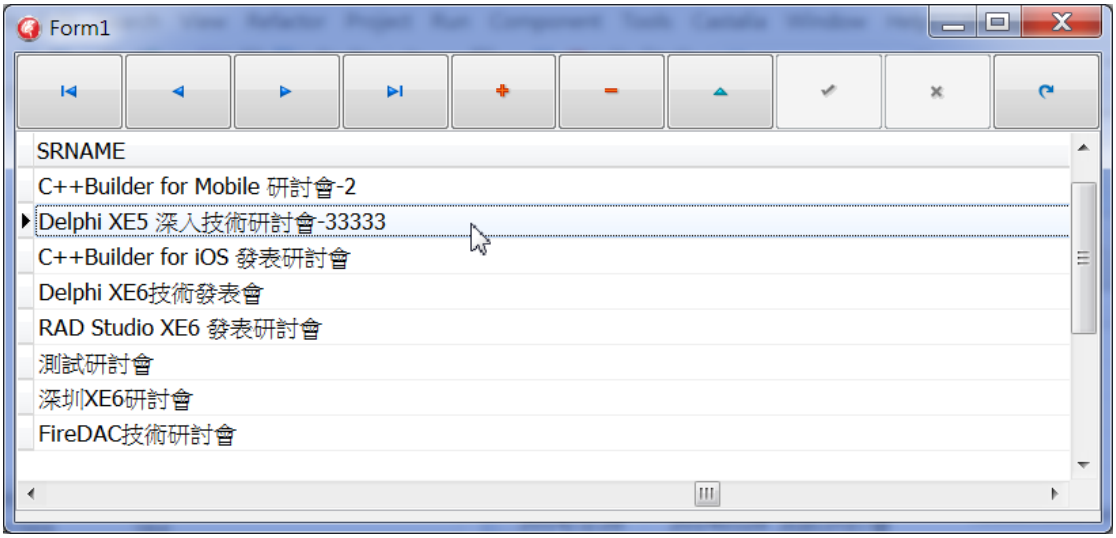
然后设定 TFDMoniFlatFileClientLink 组件的 FileName 特性值为一个文本文件再设定它的 Tracing 特性值为 True：



执行范例程序并修改一笔数据然后更新回数据库，例如修改 " Delphi XE5 深入技术研讨会-33”



为”Delphi XE5 深入技术研讨会-33333”：

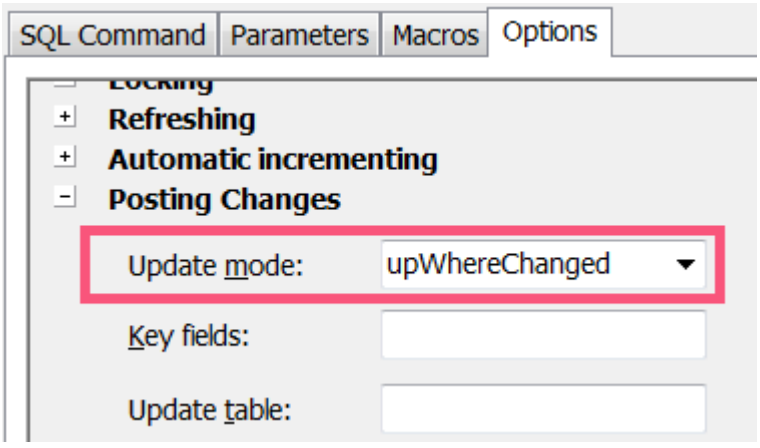


由于 upWhereKeyOnly 是 UpdateOptions.UpdateMode 的内定值，因此如果现在检查 TFDMoniFlatFileClientLink 组件写入的文本文件，可以看到 FireDAC 使用了如下的 Update 命令来寻找旧的数据并更新：

```
Prepare [Command="UPDATE SEMINARS
SET SRNAME = :NEW_SRNAME, DESCRIPTION = :NEW_DESCRIPTION
WHERE SRNAME = :OLD_SRNAME"]
```


在 where 中我们可以看到 FireDAC 只使用键值字段” SRNAME”来寻找旧的数据。

但如果我们设定 UpdateOptions.UpdateMode 为 upWhereChanged：



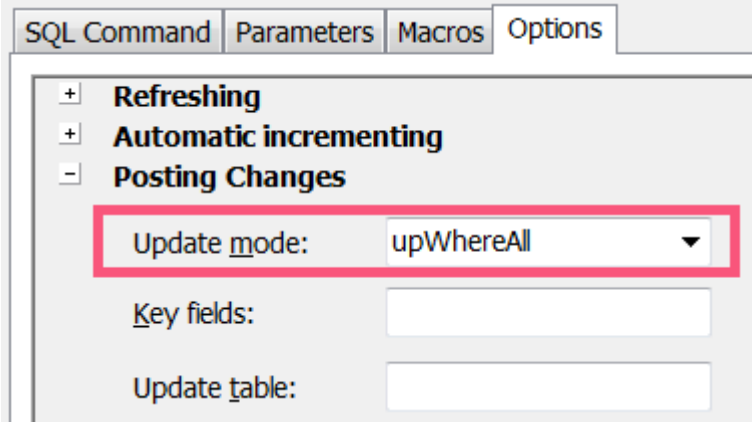
那么可以看到 FireDAC 使用了如下的 Update 命令来寻找旧的数据并更新：

```
"UPDATE SEMINARS
SET SRNAME = :NEW_SRNAME, DESCRIPTION = :NEW_DESCRIPTION
```

```
WHERE SRNAME = :OLD_SRNAME AND DESCRIPTION = :OLD_DESCRIPTION"
```

在 where 中我们可以看到 FireDAC 只使用了键值字段”SRNAME”和 DESCRIPTION 字段来寻找旧的数据，这是因为我们同时修改了 DESCRIPTION 字段中的数值。

最后如果我们设定 UpdateOptions.UpdateMode 为 upWhereAll：



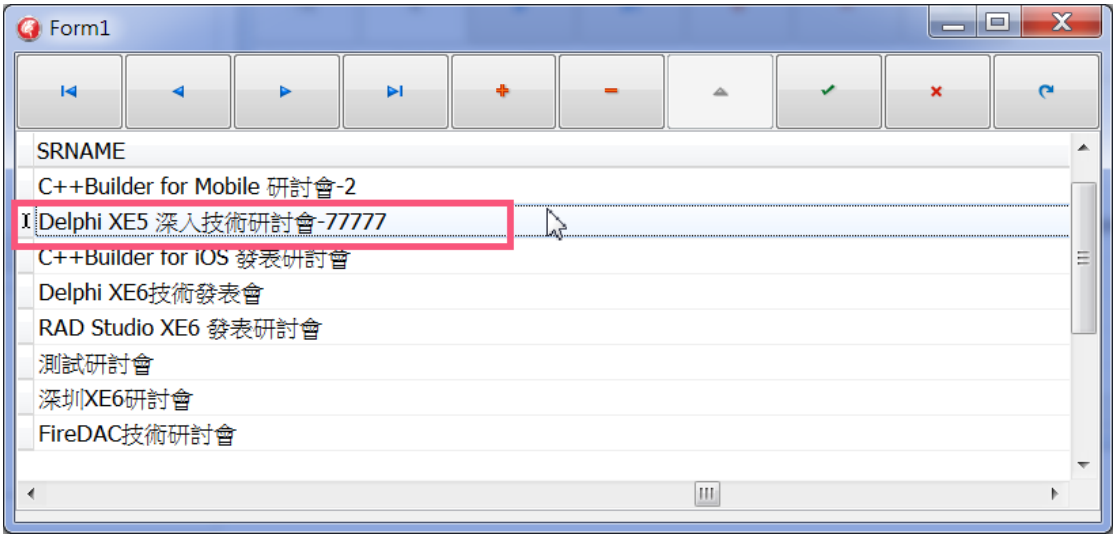
那么可以看到 FireDAC 使用了如下的 Update 命令来寻找旧的数据并更新：

```
"UPDATE SEMINARS
SET SRNAME = :NEW_SRNAME, DESCRIPTION = :NEW_DESCRIPTION
WHERE SRDATE = :OLD_SRDATE AND SRID = :OLD_SRID AND SRNAME
= :OLD_SRNAME AND
  DESCRIPTION = :OLD_DESCRIPTION"
```

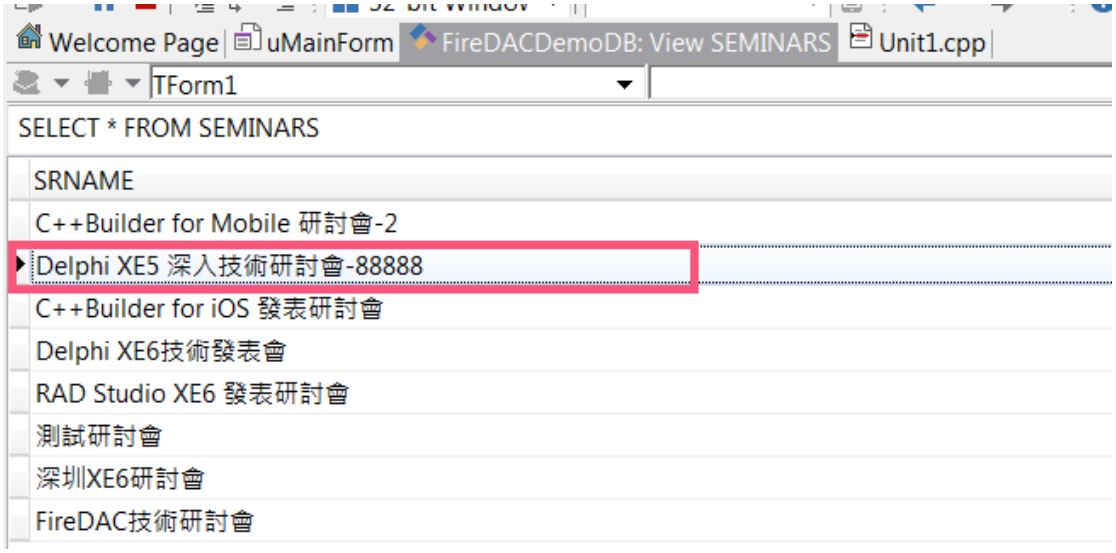

在 where 中我们可以看到 FireDAC 使用了所有未更新资料前的旧域值来寻找旧的数据。

在多人使用的环境中 UpdateOptions.UpdateMode 的设定会决定数据是否能成功更新回后端，这是因为刚才说明 UpdateOptions.UpdateMode 控制了在数据更新回数据表之前如何先找到要被异动的数据。让我们使用一个小小的范例说明读者就会了解了。

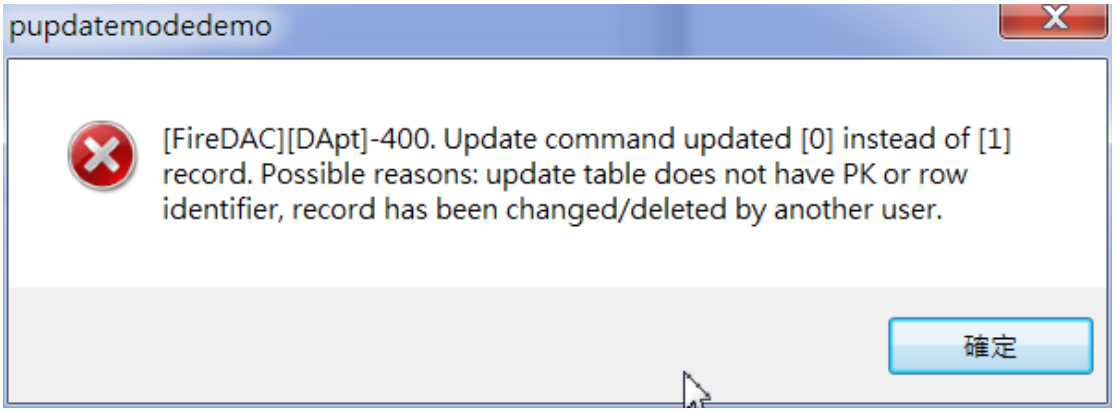
再次执行前的范例程序，让我们如同下图修改”Delphi XE5 深入技术研讨会”这笔资料为”Delphi XE5 深入技术研讨会-77777”：



但在按下 Post 按钮把数据更新回数据表之前，先使用 IDE 中的 Data Explorer(或是任何可修改数据的工具)把”Delphi XE5 深入技术研讨会”这笔资料改成”Delphi XE5 深入技术研讨会-88888”然后立即更新回数据表：



接着再回到范例程序点选更新按钮后就会看到范例程序出现如下的错误：



为什么会出现错误呢？很简单，因为范例程序在更新数据回数据表时会先以”Delphi XE5 深入技术研讨会”这个键值到数据表中搜寻，但由于这笔数据的键值已经被另一客户端 Data Explorer 改成”Delphi XE5 深入技术研讨会-88888”，因此范例程序找不到这笔要被更新的数据，因此就显示此笔数据已经被其他用户更改/删除的错误了。

因此在多人使用的环境中 UpdateOptions.UpdateMode 的设定会决定数据是否能成功更新回后端。

5-3 数据转换

从 XE7 起 FireDAC 提供了一组新的 ETL(Extract-Transform-Load)组件让程开发人员可以使用来在不同的数据源中转换资料，例如如果开发人员想把数据转成文字的形式，或是把数据从 SQLite 转到 InterBase 等，这组组件称为 FireDAC ETL。

FireDAC ETL 基本上使用了 Reader/Writer 设计样例，FireDAC ETL 提供了 3 组 Reader/Writer，在数据/文字，数据集/数据集和 SQL 数据/SQL 数据之间转换。每一个功能都提供一个 FireDAC 组件让开发人员使用。下面的表格说明了这些组件：

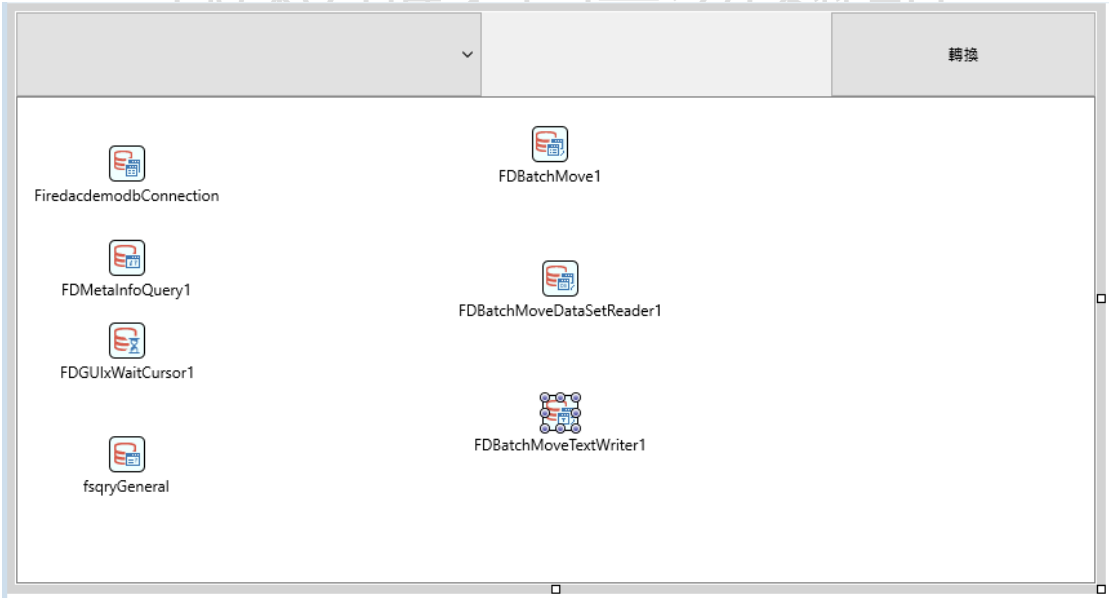
TFDBatchMove	提供实际转换功能
TFDBatchMoveTextReader	把数据以文字形式读出
TFDBatchMoveTextWriter	把数据以文字形式写出
TFDBatchMoveDataSetReader	把资料从数据集中读出

TFDBatchMoveDataSetWriter	把资料从数据集中写出
TFDBatchMoveSQLReader	把数据从 SQL 命令执行的结果中读出
TFDBatchMoveSQLWriter	把数据从 SQL 命令执行的结果中写出
TFDBatchMoveJSONWriter	以 JSON 的格式写出数据

FireDAC ETL 组件组使用上很简单，只需要链接到 TFDConnection 组件，使用 TFDBatchMoveXXXReader 组件读出数据，再使用 TFDBatchMoveXXXWriter 写出数据，最后呼叫 TFDBatchMove 组件的 Execute 方法即可，下面的 2 小节分别简单的说明如何在不同的数据源中转换数据。

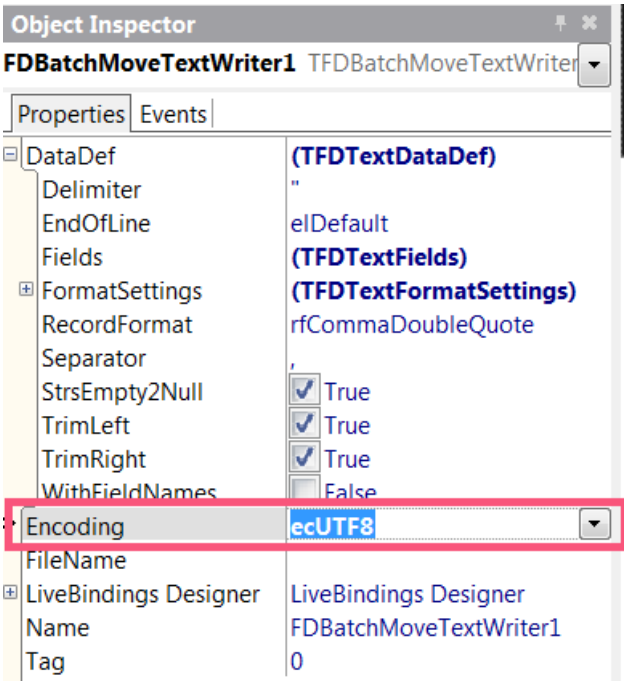
5-3-1 数据换文字格式

建立一个 Multi-Device Application 项目，在主窗体中加入如下的组件，这个范例将可以把范例数据库 FIREDACDEMODB.GDB 中的任何数据表中的数据转成文字的形象：

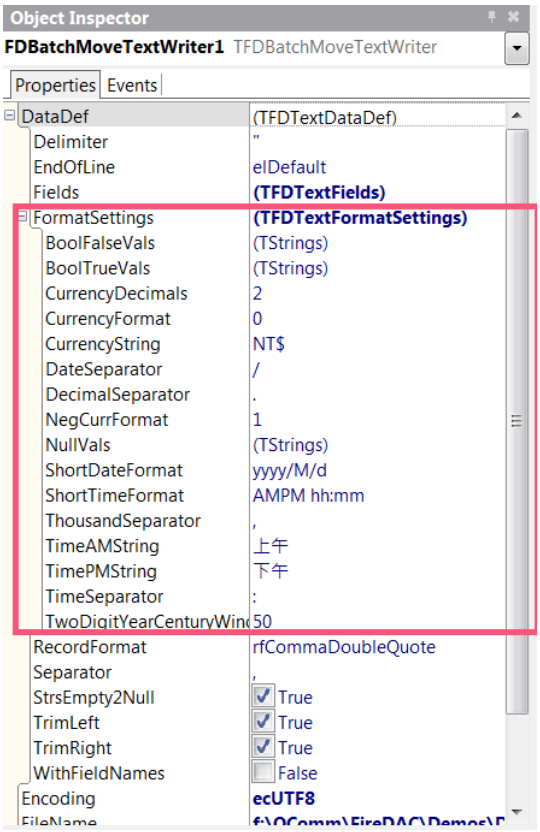


在主窗体中使用了 TFDBatchMove，TFDBatchMoveDataSetReader 和 TFDBatchMoveTextReader 这 3 个 ETL 组件，因为我们要把主窗体中用 fsqryGeneral 组件取得的数据转成文字，而 fsqryGeneral 是一个数据集组件因此使用 TFDBatchMoveDataSetReader 从它读取数据再使用 TFDBatchMoveTextReader 组件写出数据。

TFDBatchMoveTextReader 元年可以设定使用什么格式的字符串形式写出数据，例如在对象查看器中可以设定文字的 Delimiter，Separaor 等：



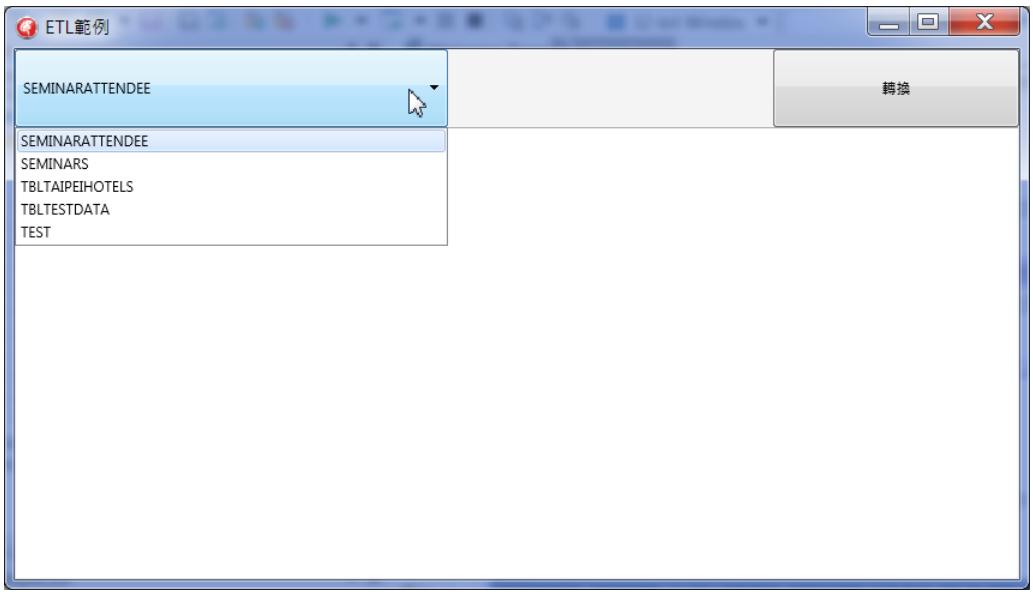
也可以在 FormatSettings 中进一步设定更多的格式，例如日期格式，布尔值格式等等：



范例应用程序在启动时使用 TFDMetaInfoQuery 组件取得数据库中所有数据表的信息：

```
procedure TfmMainForm.FillTableInfo;
begin
  if (ComboBox1.Items.Count = 0) then
  begin
    FDMetaInfoQuery1.Active := True;
    while (not FDMetaInfoQuery1.Eof) do
    begin

ComboBox1.Items.Add(FDMetaInfoQuery1.FieldByName('TABLE_NAME').A
sString);
     FDMetaInfoQuery1.Next;
    end;
    ComboBox1.ItemIndex := 0;
  end;
end;

procedure TfmMainForm.FormActivate(Sender: TObject);
begin
  FillTableInfo;
end;
```


范例应用程序在执行后用户就可以在左上方的 TComboBox 中选择要转换数据的数据表：



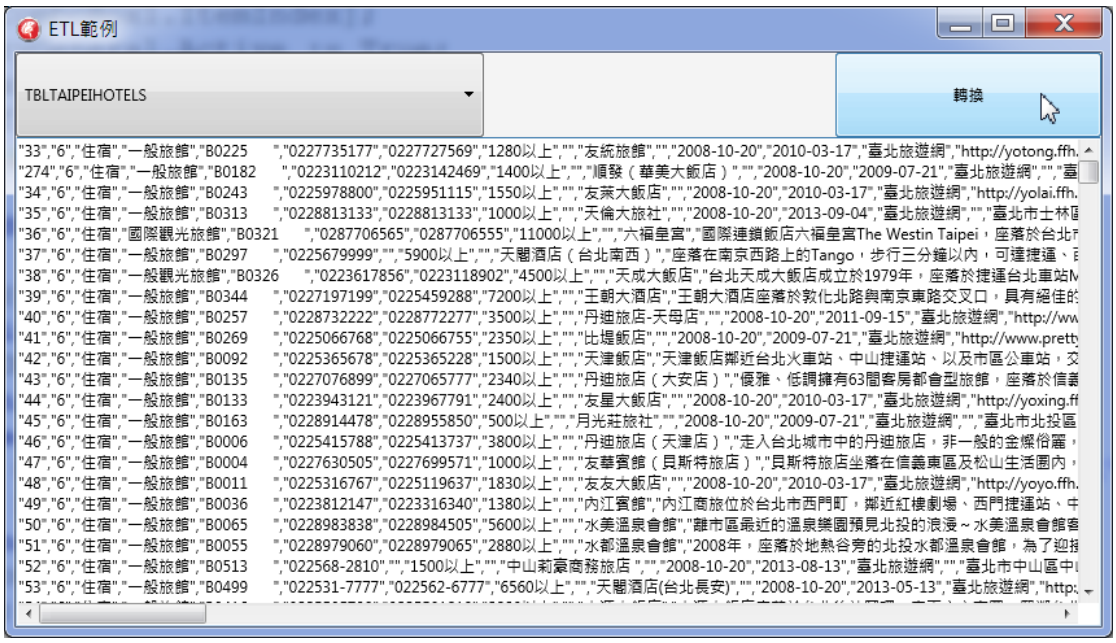
接着点选转换按钮范例程序就会执行下面的程序代码：

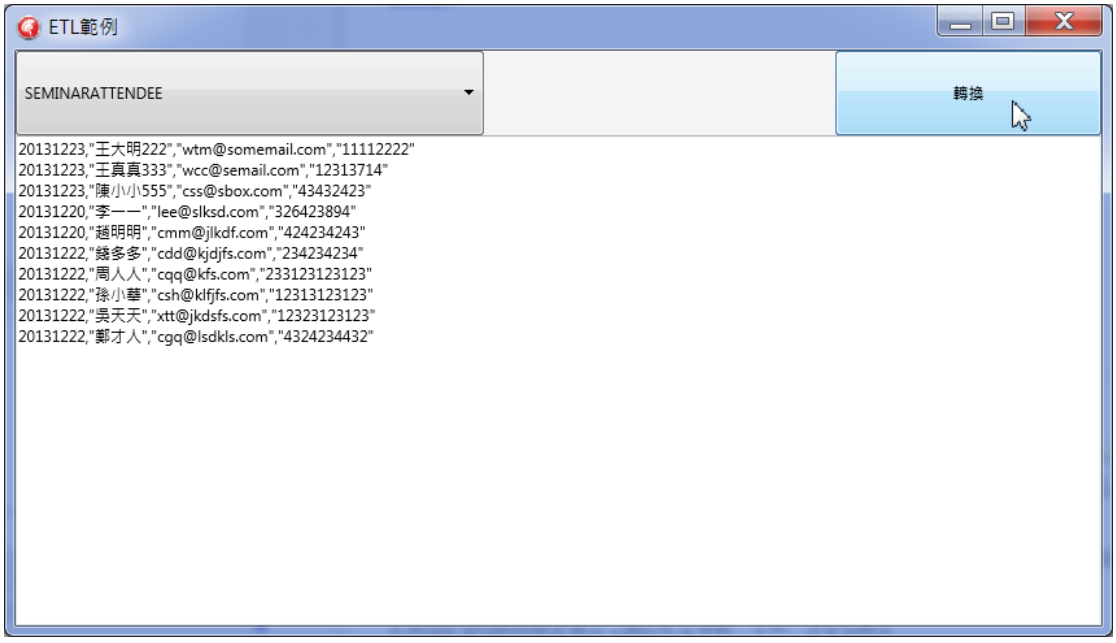
```
001    uses System.IOUtils;
002
003    procedure TfmMainForm.btnETLClick(Sender: TObject);
004    begin
005      if (TFile.Exists(FDBatchMoveTextWriter1.FileName) )then
006        TFile.Delete(FDBatchMoveTextWriter1.FileName);
007      fsqryGeneral.Active := False;
008      fsqryGeneral.MacroByName('TABLENAME').AsRaw :=
ComboBox1.Items[
009         ComboBox1.ItemIndex];
010      fsqryGeneral.Active := True;
011      FDBatchMove1.Execute;
012
Self.mmData.Lines.LoadFromFile(FDBatchMoveTextWriter1.FileName);
013    end;
```


007~010 使用 FireDAC 的 Macro 功能把用户选择的数据表名称代入 SQL 命令中再执行此 SQL 命令。因此 fsqryGeneral 的 SQL 特性中使用了如下的 SQL Macro 指令：

```
select * from &TableName
```

011 行就呼叫 T FDBatchMove 组件的 Execute 方法就可以非常简单的完成数据转文字的工作了,下面就是范例程序转换 TBLTAIPEIHOTEL 和 SEMINARATTENDEE 这 2 个数据表的结果：



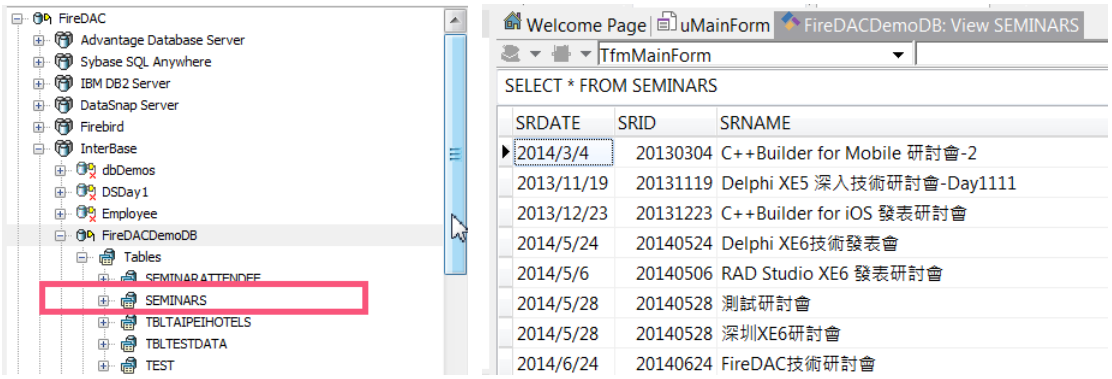


5-3-2 在不同数据源中转换数据

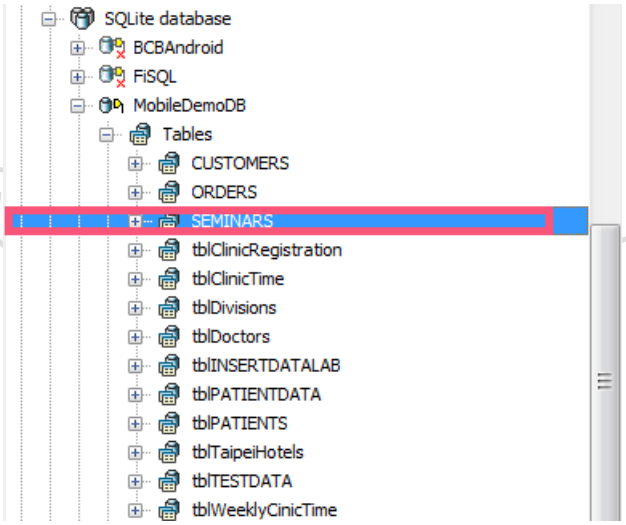
使用 FireDAC ETL 组件组在不同的数据表格中转换数据非常的容易，程序员只需要使用 2 个 TFDConnection 链接到不同的数据库，再使用 1 个 TFDQuery 指面源数据表格，使用另一个 TFDQuery 指面目的地数据表格，最后使用 TFDBacthMove 组件指到这 2 个 TFDQuery 组件再呼叫 TFDBacthMove 的 Execute 方法即可，例如下面是笔

者如何使用 FireDAC ETL 组件组把 InterBase 数据库中的数据自动转换到 SQLite 数据库中。

下面的图形显示了存在于 InterBase 中的 SEMINARS 数据表中的数据：

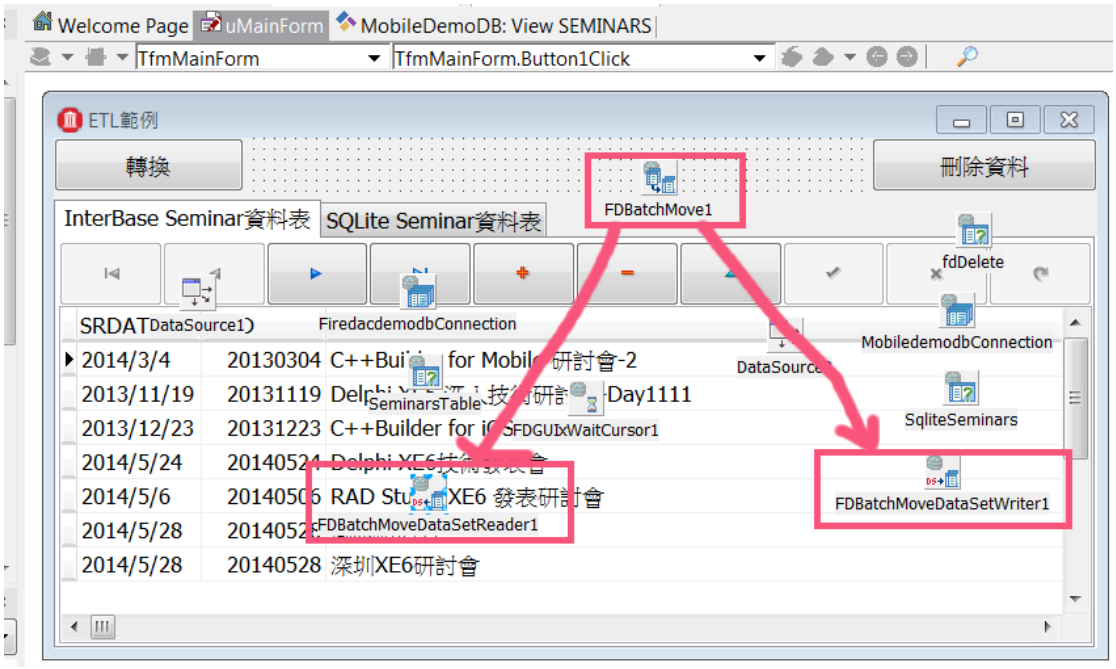


现在我们要使用 FireDAC ETL 组件组把上面的数据转换到下面的 SQLite 数据表中：

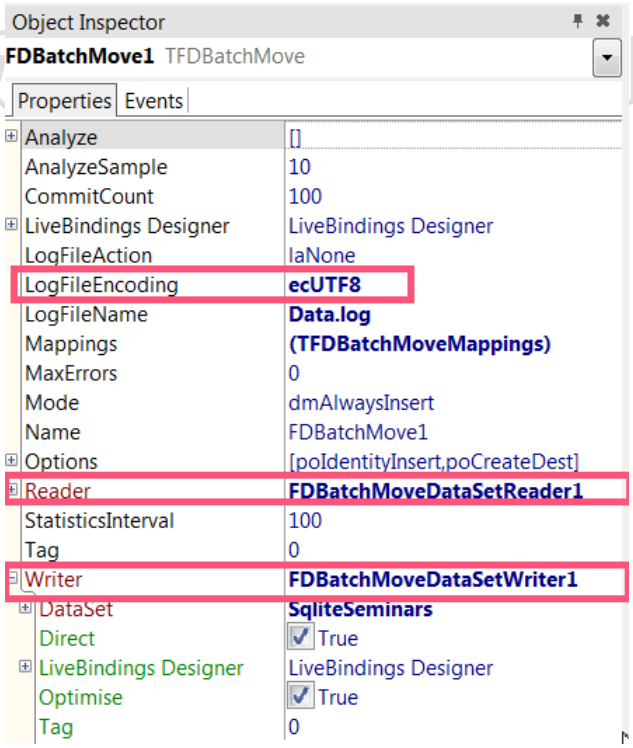


此范例程序使用了 TFDConnection 组件 FiredacdemodbConnection 链接到 InterBase 数据库，再使用 TFDQuery 组件 SeminarsTable 链接到 InterBase 的 SEMINARS 数据表，再使用 TFDConnection 组件 MobiledemodbConnection 链接到 SQLite 数据库，再使用 TFDQuery 组件 SqliteSeminars 链接到 InterBase 的 SEMINARS 数据表。

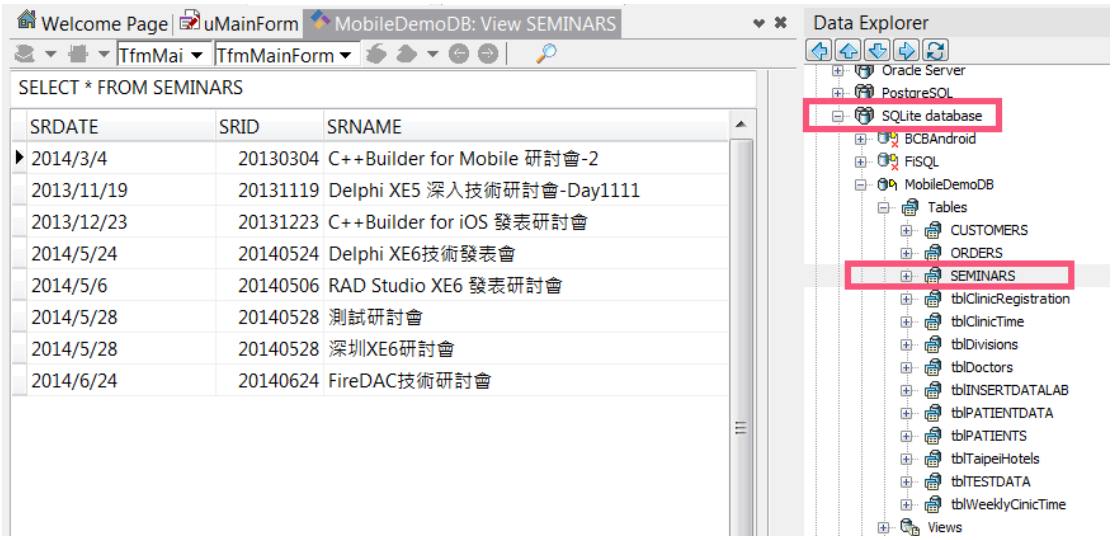
再放入 TFDBatchMoveDataSetReader 组件 FDBatchMoveDataSetReader1 链接到 SeminarsTable，TFDBatchMoveDataSetWriter 组件 FDBatchMoveDataSetWriter1 链接到 SqliteSeminars，最后再放入 TFDBatchMove 组件 FDBatchMove1：



设定 FDBatchMove1 的 LogFileEncoding 为 ecUTF8，Reader 特性值为 FDBatchMoveDataSetReader1，Writer 特性值为 FDBatchMoveDataSetWriter1：



再呼叫 FDBatchMove1 的 Execute 方法后 InterBase 中的数据就立刻的转换到下面的 SQLite 数据库的 SEMINARS 数据表中了：



FireDAC ETL 组件组使用上非常的简单，却可提供非常方便的数据转换功能。

5-4 处理自动增加值字段 (Auto-Increment Field)

许多数据库都支持所谓的自动增加值字段型态，例如 MS SQL Server 的标识列以及 SQLite 的 Autoinc 字段。这种字段通常是由后端的数据库自己维护其数值而不是由程序员撰写程序代码写入数值。不过在实际的应用中程序员会需要取得后端数据库在这种字段中写入的数值，例如这种字段通常都会被定义成键值，程序员需要根据此键值来搜寻相关的数据或是进行主从数据的处理，那么要如何使用 FireDAC 处理这种型态的字段？

比起以前的 BDE/IDAPI 和 dbExpress，FireDAC 提供了非常方便的机制让程序员来使用和处理自动增加值的字段。FireDAC 提供了 2 种方式处理自动增加值的字段：

处理模式	说明
自动模式	在这种模式中当开启拥有自动增加值字段的数据表时，FireDAC 便会试着找出这个自动增加值的字段并且自动进行适当的特性值设定。
手动模式	如果程序员不想使用自动模式或是 FirDAC 无法辨识出自动增加值的字段，那么程序员可以自行设定适当的特性值。

让我们使用一个范例来说明如何让 FireDAC 处理自动增加值字段。

下面是一个 SQLite 的范例数据库，其中的 SRID 字段就是自动增加值字段型态：

Create statement

```
CREATE TABLE "tblSeminars" ("SRID" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE , "SRNAME" VARCHAR, "SRDATE" DATETIME, "VENUE" VARCHAR)
```

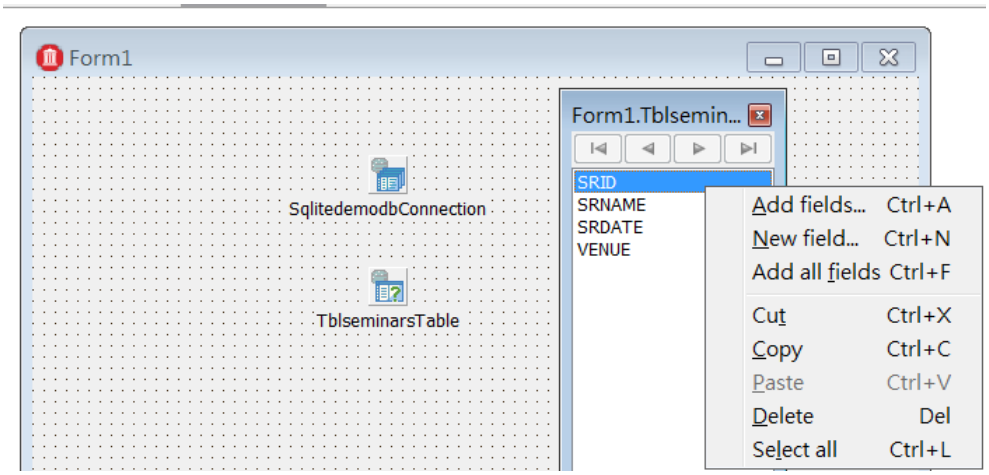
More Info

No. of Records: 0 No. of Indexes: 1 No. of Triggers: 0

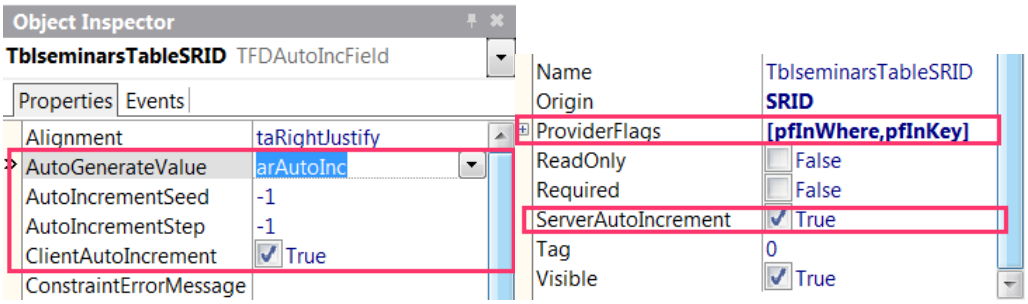
Columns (4)

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	SRID	INTEGER	1	null	1
1	SRNAME	VARCHAR	0	null	0
2	SRDATE	DATETIME	0	null	0
3	VENUE	VARCHAR	0	null	0

如果我们使用 FireDAC 连结并开启上面的 tblSeminars 数据表并且使用 TFDQuery 的组件编辑器点选其中的 SRID 字段：



那么在对象查看器中就可以看到 FireDAC 自动对 SRID 字段进行了如下重要的设定：

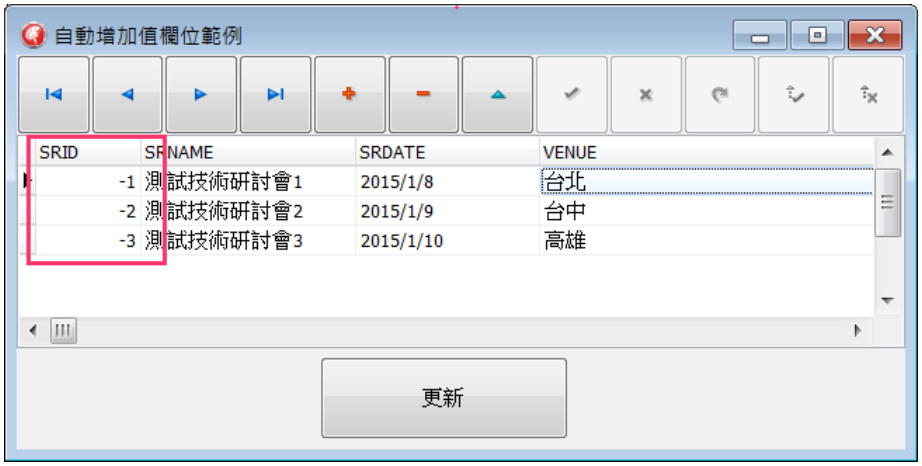


我们使用下面的表格来说明这些特性值的意义：

处理模式	说明
ClientAutoIncrement	由于自动增加值字段的数值是由后端的数据库设定，但在前端 FireDAC 处理自动增加值字段时仍然暂时需要设定这个域值，因

	此 FireDAC 会设定 ClientAutoIncrement 特性值为 True，如此一来当客户端新增数据时，会暂时使用 AutoIncrementSeed 和 AutoIncrementStep 这 2 个特性值设定动增加值字段的暂时数值。例如在上面的图形中 AutoIncrementSeed 是-1 代表客户端新增一笔数据时动增加值字段的数值就设定为-1，第 2 笔新增的数据就以 AutoIncrementStep 的特性值增加，因此就是-2。
AutoGenerateValue	设定为 arAutoInc 代表此域值是自动增加的
ProviderFlags	设定为 pfInWhere，如果此自动增加值字段是键值的话就再加入设定 pfInKey
ServerAutoIncrement	代表此字段的数值是由后端数据库设定

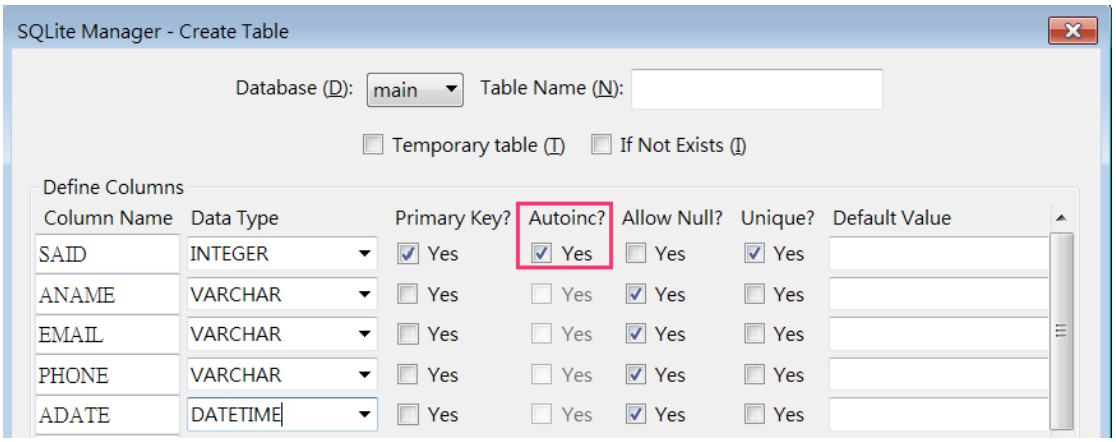
下面的范例显示了 FireDAC 处理自动增加值字段的行为，在下面的范例中我们先开启 Cached Updates 功能以便让我们可详细观察客户端和后端数据库如何处理自动增加值字段。首先我们可看到在客户端新增数据时自动增加值字段的暂时数值由 FireDAC 处理，FireDAC 使用带面说明的 AutoIncrementSeed 和 AutoIncrementStep 这 2 个特性值设定自动增加值字段的暂时数值，从-1 开始增加：



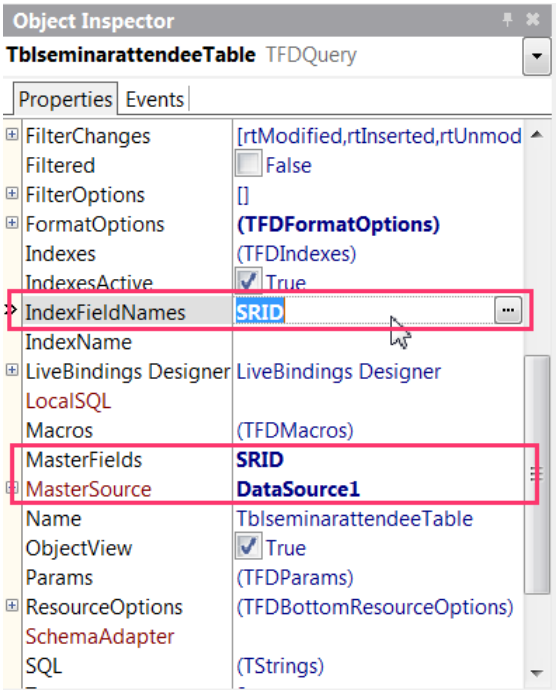
一旦我们呼叫了 ApplyUpdates 方法把数据更新回后端数据库就可以从下面看到 FireDAC 从后端数据库取得了自动增加值字段由后端数据库真正指定的数值：

再让我们看看如何在主/从资料中处理自动增加值字段。

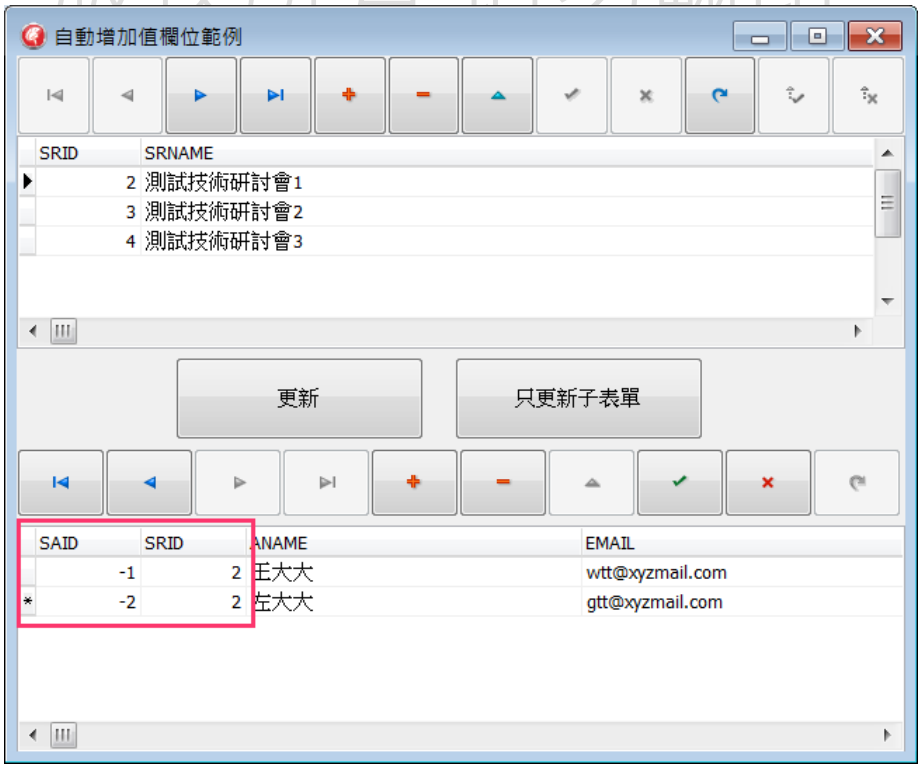
下面是 tblSeminarAttendee 数据表，它也拥有一个自动增加值字段 SAID 并且使用 SRID 字段和前面的 tblSeminars 数据表关连：



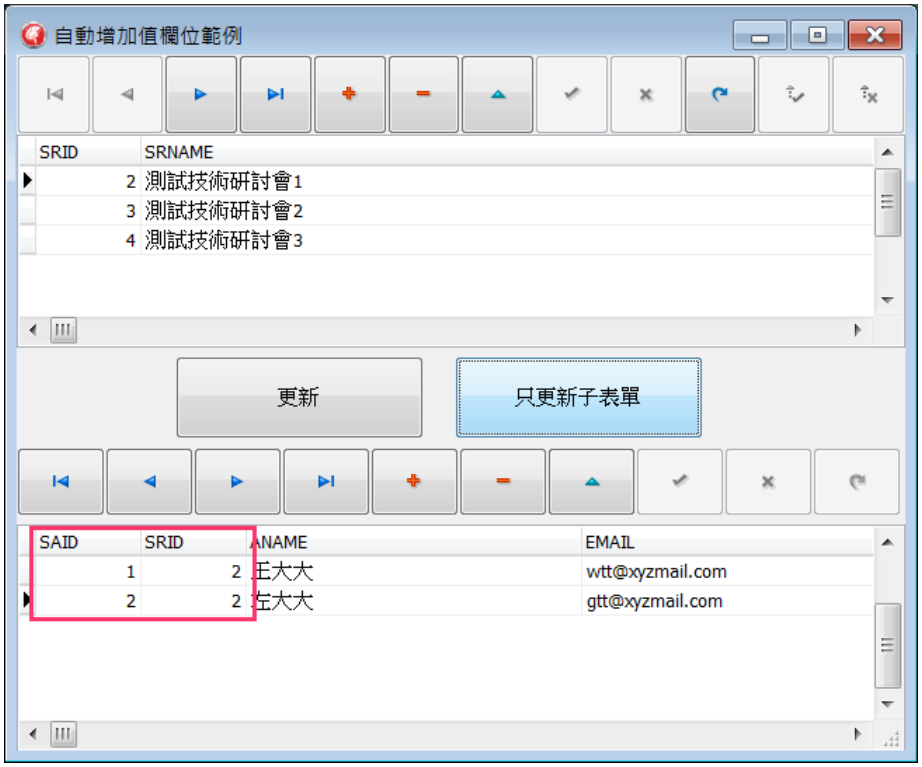
存取 tblSeminarAttendee 数据表的 TFDQuery 组件 TblseminarattendeeTable 使用 SRID 字段和存取 tblSeminars 数据表的 TFDQuery 组件 TblseminarsTable 设定关连：



执行此范例程序并且开始在 tblSeminarAttendee 数据表加入数据，从下面的图形可以看到 tblSeminarAttendee 数据表的自动增加值字段 SAID 也被客户端的 FireDAC 指定暂时数值，而关连字段 SRID 则正确设定成主数据表的 SRID 字段的数值：



在点选”只更新子窗体”按钮后 tblSeminarAttendee 数据表的自动增加值字段 SAID 会正确的被后端的数据库设定数值：

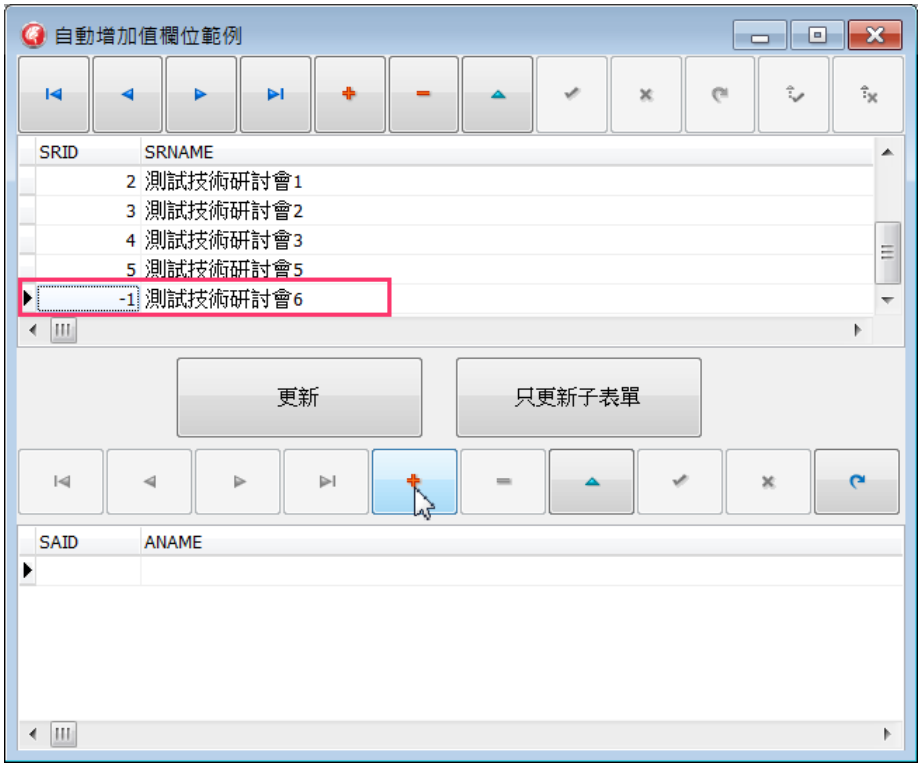


但如果想在子数据表 tblSeminarAttendee 新增数据，那么我们必须在它的 TFDQuery 组件 TblseminarattendeeTable 的 BeforeInsert 事件中撰写如下的程序代码：

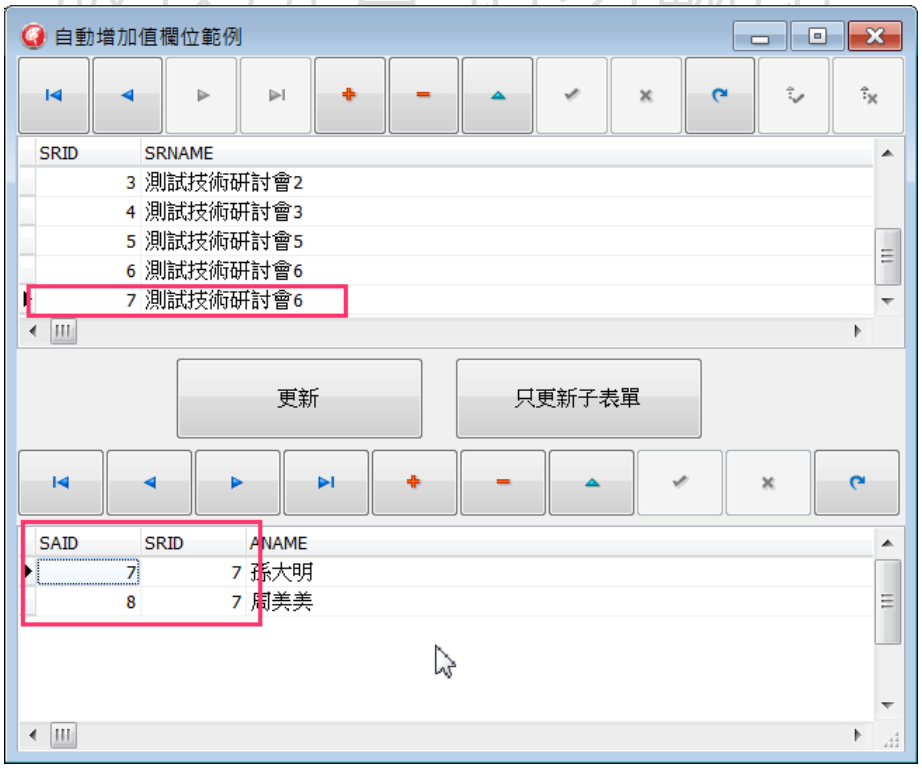

```
procedure TForm1.TblseminarattendeeTableBeforeInsert(DataSet:
TDataSet);
begin
  TblseminarsTable.ApplyUpdates(0);
  TblseminarsTable.Refresh;
end;
```

在 BeforeInsert 事件中我们需要先更新主数据表新增的数据，再取得后端数据库真正指定给 tblSeminars 数据表的 SRID 域值，再由 FireDAC 指定给 tblSeminarAttendee 数据表的 SRID 域值。

从下面的执行结果可以看到现在在主数据表 tblSeminars 中新增数据此时 SRID 域值为 FireDAC 指定的-1：



接着在子数据表 tblSeminarAttendee 新增数据并更新回后端数据库后可以看到 tblSeminarAttendee 数据表的 SRID 域值和 SAID 域值都是正确的：



从这个范例可以证明 FireDAC 对于自动增加值字段的支持是非常强大又方便使用的。

每个数据库对于如何支持/实作自动增加值的字段的方式不同，FireDAC 会尽可能的使用自动模式帮忙程序员处理这种字段，如果 FireDAC 无法自动处理您使用数据库的自动增加值字段型态，请参考您的数据库和 FireDAC 的手册说明。

5-5 使用计算字段

除了代表真正数据表字段的 TField 类别外，FireDAC 也支持计算字段(Calculated Field)和聚集字段(Aggregated Field)。计算字段是所谓的虚拟字段，计算字段并不存在于实际的数据表中，而是在应用程序执行时暂时存在的字段。计算字段一般是因为应用程序在执行时需要暂时储存一些必要的计算数值而存在的，在应用程序结束后这些计算数值并不需要储存在数据库中。

FireDAC 支持 4 种不同的计算字段：

计算字段种类	说明
fkCalculated	最简单的计算字段，通常只是使用来进行简单的计算使用。这种计算字段可在 TDataSet 的 OnCalcFields 事件处理及式中进行。
fkInternalCalc	进阶的计算字段，它的数值可像一般的 TField 对象一样暂时储存在数据集的快储内存中。这种计算字段可在 TDataSet 的 OnCalcFields 事件处理及式中进行或是使用 TField 的 DefaultExpression 特性值来计算。
fkLookup	这种计算字段可自动执行查询的结果值。
fkAggregate	这种计算字段是属于聚集字段，它使用 TAggregateField 的 DefaultExpression 特性值来计算。

上面的 fkAggregate 计算字段提供了下面 5 个操作数让程序员使用：

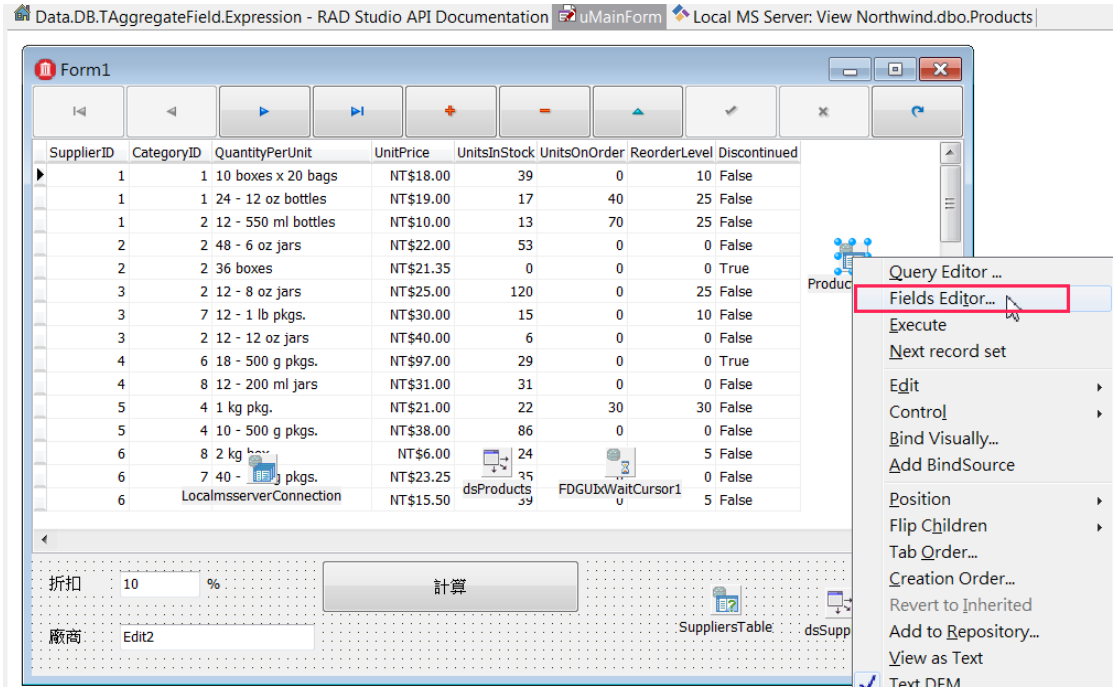
操作数	说明
Sum	对数值或表达式提供总计值
Avg	对数值或表达式提供平均值
Count	对字段对象或表达式提供非空白值的个数
Max	提供对字符串，数值或表达式中的最大值
Min	提供对字符串，数值或表达式中的最小值

让我们使用一个范例来说明如何使用计算字段。

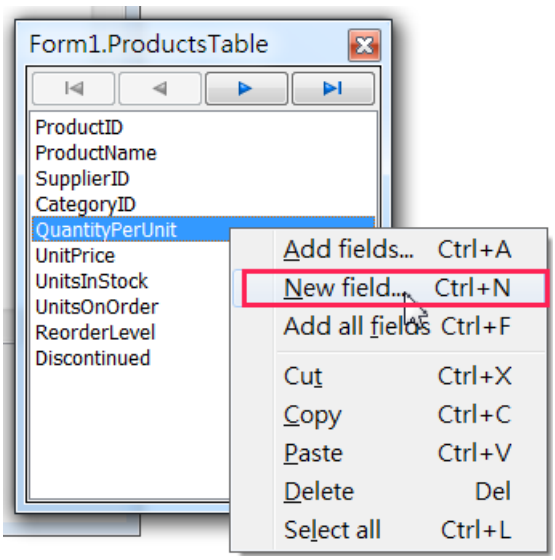
下面是使用 FireDAC 存取 MS SQL Server 的 NorthWind 数据库中的 Products 数据表程序，现在假设我们想提供其中 UnitPrice 字段乘上 UnitOnOrder 字段的收入数值，接着再提供这 2 个字段乘上折扣的收入数值等信息，最后又可在 Products 数据表中看到供应厂商的名称。

要如此做我们可以建立 2 个计算字段提供 UnitPrice 字段乘上 UnitOnOrder 字段的收入数值以及这 2 个字段乘上折扣的收入数值，最后再使用 Lookup 型态的计算字段根据 Products 的 SupplierID 域值来查找供应厂商的名称。

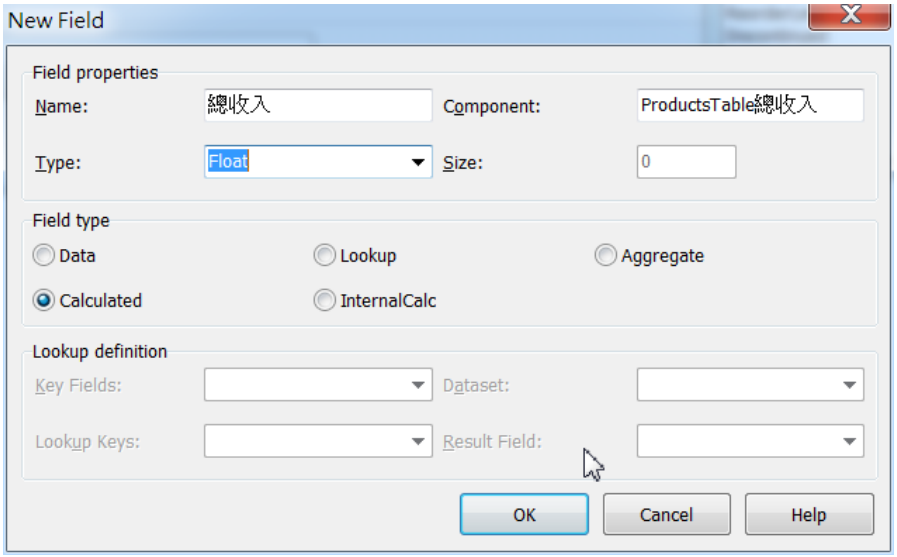
首先点选主窗体中的 TFDQuery 组件 ProductsTable，右击鼠标从突显式选单中选择 Fields Editor...选项：



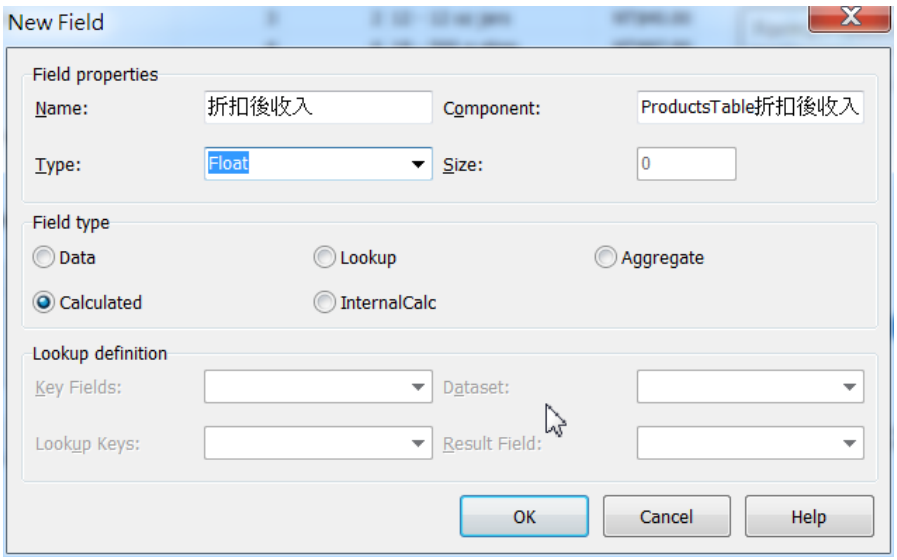
在其中点选 New field...选项建立新的计算字段对象：



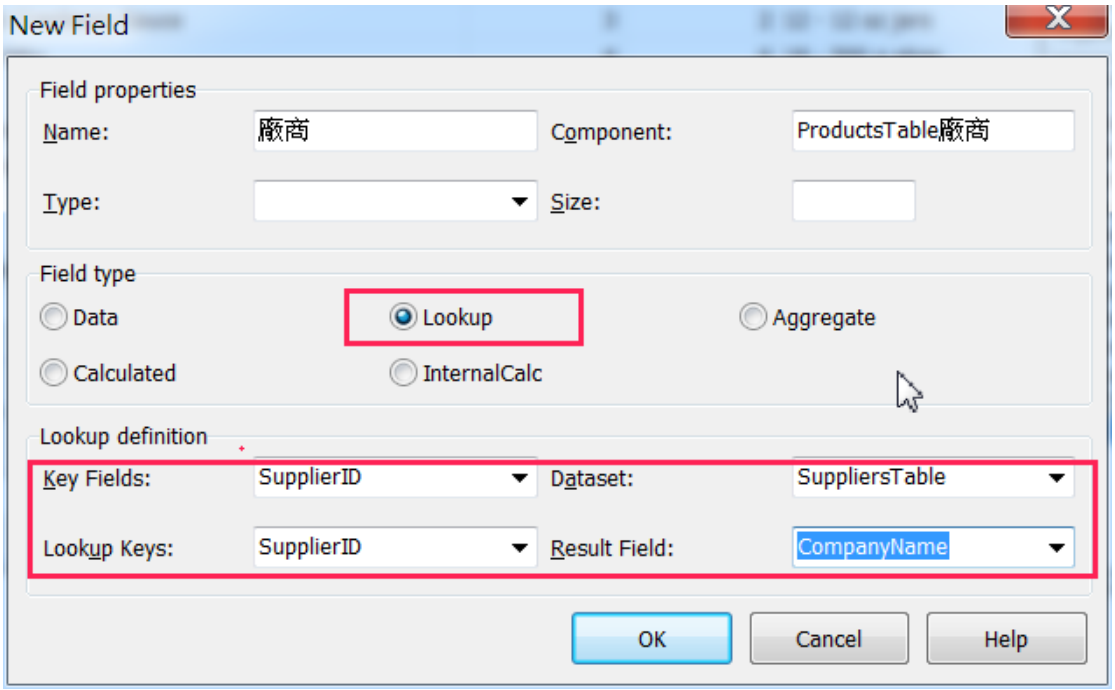
先如下建立一个总收入计算字段对象：



再建立一个折扣后收入计算字段对象：



最后再建立一个 Lookup 型态的计算字段对象：



这个 Lookup 型态的计算字段对象使用了主窗体中的另外一个 TFDQuery 自动到 Supplier 数据表中根据 SupplierID 域值查找并回传 CompanyName 的域值。

最后 ProductsTable 的字段编辑器如握有如下的正常字段和新增的计算字段：

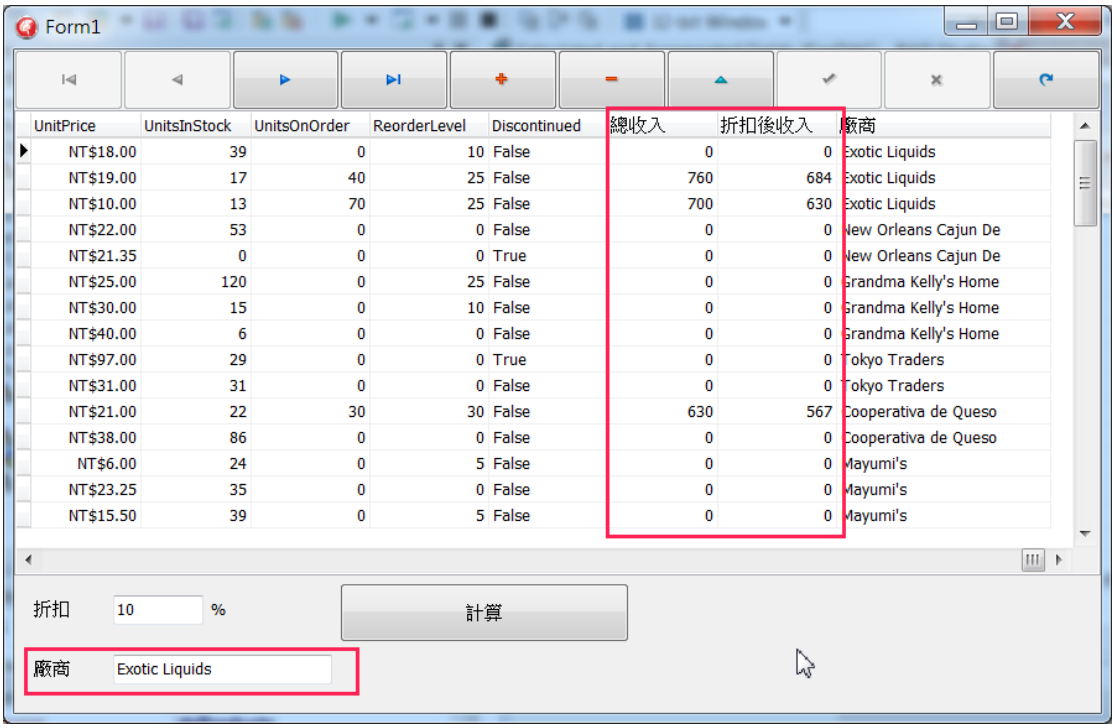
接着在 ProductsTable 的 OnCalcFields 事件处理函式中撰写如下的程序代码：

```
procedure TForm1.ProductsTableCalcFields(DataSet: TDataSet);
var
  iDiscount : Integer;
begin
  iDiscount := StrToInt(edtDiscount.Text);
  ProductsTable总收入.AsFloat := ProductsTableUnitPrice.AsFloat *
ProductsTableUnitsOnOrder.AsInteger;
  ProductsTable折扣后收入.AsFloat :=
(ProductsTableUnitPrice.AsFloat * ((100 - iDiscount) / 100.0) ) *
ProductsTableUnitsOnOrder.AsInteger;
end;
```

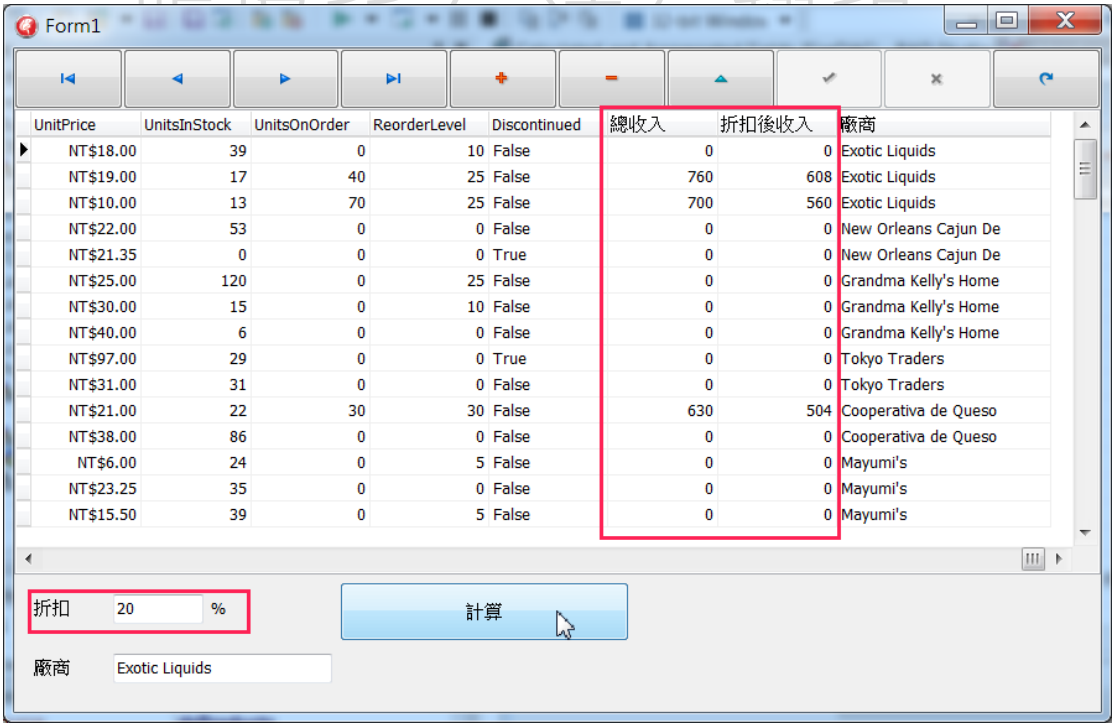

最后在主窗体"计算"按钮的 OnClick 件处理函式中撰写如下的程序代码以便在使用者输入新的折扣值之后强迫 ProductsTable 重新计算所有计算字段的数值：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ProductsTable.Refresh;
end;
```

执行此范例程序便可看到如下的结果，总收入计算字段和折扣后收入计算字段都提供了正确而需要的信息：



如果输入新的折扣再点选”计算”按钮就可以看到总收入计算字段和折扣后收入计算字段重新进行计算并提供计算后的数值了：



5-6 结论

本章说明了许多 FireDAC 处理数据的技巧，善用这些技巧可以大幅增加您的 FireDAC 应用程序的功能以及执行效率。

第6章 MongoDB数据库开发

NoSQL 型态的数据库在这几年非常的流行，也被愈来愈多的公司 / 企业所接受和使用，FireDAC 也在 DX10 的版本中开始加入支持 NoSQL 型态的数据库，并选择其中最为流行的 MongoDB 为第 1 个支持的目标。

本节将介绍如何使用 FireDAC 开发 MongoDB 的应用程序。

6-1 MongoDB 的基本介绍

MongoDB 是 10gen 开发出来的的 NoSQL 数据库，Mongo 的数据体结构是以(Key,Value)组合的，储存的方式是使用 JSON 格式，不过为了执行速度考虑，在内部处理上的格上是使用 BSON。所谓 BSON 指的是 Binary JSON 的意思，读者可以在下面的 URL 找到 BSON 的说明：

```
https://en.wikipedia.org/wiki/BSON
```

MongoDB 的特点就是每一笔文件的是字段的数据型态是不一定的，字段的存在性也是不一定的，这和传统的 RDBMS 是很不一样的。例如在 RDBMS 中一个字段在使用之前一定要定义字段的数据型态，一旦定义好之后该字段储存的数值就一定必须是该数据型态的性质，但在 MongoDB 中则无需如此，我们不需要事先定义字段型态，每一笔数据相同字段的数值的数据型态也可以不同。不过在 MongoDB 中所谓的每笔数据是称为文件(Document)，下面的表格整理了传统数据库中的对象在 MongoDB 中的名称：

关连式数据库(RDBMS)	MongoDB
数据库(Database)	DataBase
数据表(Table)	Collection

资料(Record/Row)	Document
字段(Column)	Field
主索引(PK)	_id
函式(function)	Function()
预储程序(stored procedure)	mapreduce

因此在 RDBMS 中一个数据库有许多的数据表，而在 MongoDB 中则称为一个数据库有许多的 Collections，RDBMS 中一个数据表中有许多笔资料，而 MongoDB 中则称为一个 Collection 有许多 Documents。

MongoDB 有许多的特点，下面是比较重要的特点：

1.MongoDB 可以处理数据库为 T 级量 的数据库，也就是处理大数据的数据库。

2.分布式的数据库模式，可以把众多数据库串联后处理大数的数据。

3.MongoDB 可直接储存对象，每个字段也可以储存对象

4.Monogo 基本上是使用 JavaScript 和 JSON 的数据库

例如在 MongoDB 数据库中假设有一个称为 employee 的 Collection，在这个 Collection 中我们可以使用下面的 JSON 格式储存一个 Document：


```
{
	"_id" : ObjectId("55da85e6a797e189008fec46"),
	"name" : "李大明",
	"account" : "LTM",
	"country" : "tw",
	"age" : 36
}
```

如果我们注意上面的格式就可以发现它是一个 JSON 对象，其中包含了 4 个 JSON Pair，不过上面的”_id” 字段是由 MongoDB 自行产生的键值，由于 MongoDB 的字段也可以忽略，因此我们又可以使用下面的 JSON 新增另一 Document，而且新增了一个 email 字段，这个 email 字段则是一个 JSON 数组对象：

```
{
	"_id" : ObjectId("55dad3b0a797e189008fec47"),
	"name" : "王小华",
	"account" : "WSH",
	"country" : "tw",
```

```
        "age" : 22,
        "email" : [
                "wsh@gmail.com.tw",
                "wsh@hotmail.com"
        ]
}
```

有了这些基本的说明后我们就可以开始说明如何使用 FireDAC 来开发 MongoDB 的应用程序了，如果读者想了解更多有关 MongoDB 的观念，那么可以到 MongoDB 官方网站或是参考市面上的 MongoDB 专业书籍。

6-2 下载和安装 MongoDB

在使用 FireDAC 开发 MongoDB 应用程序之前读者必须先下载和安装 MongoDB，要下载 MongoDB，请到 MongoDB 官方网站：

http://www.mongodb.org/downloads

例如笔者下载和安装的是 Win32 3.0.4 的版本，


```
mongodb-win32-x86_64-2008plus-ssl-3.0.4-signed.msi
```

由于上面的版本使用了 SSL，因此在您的执行目录中必须拥有 SSL 的 libeay32.dll 和 ssleay32.dll 这 2 个 DLL。

下载和安装完 MongoDB 后会在 Program Files (x86)或是 Program Files MongoDB 目录中看到如下的内容：

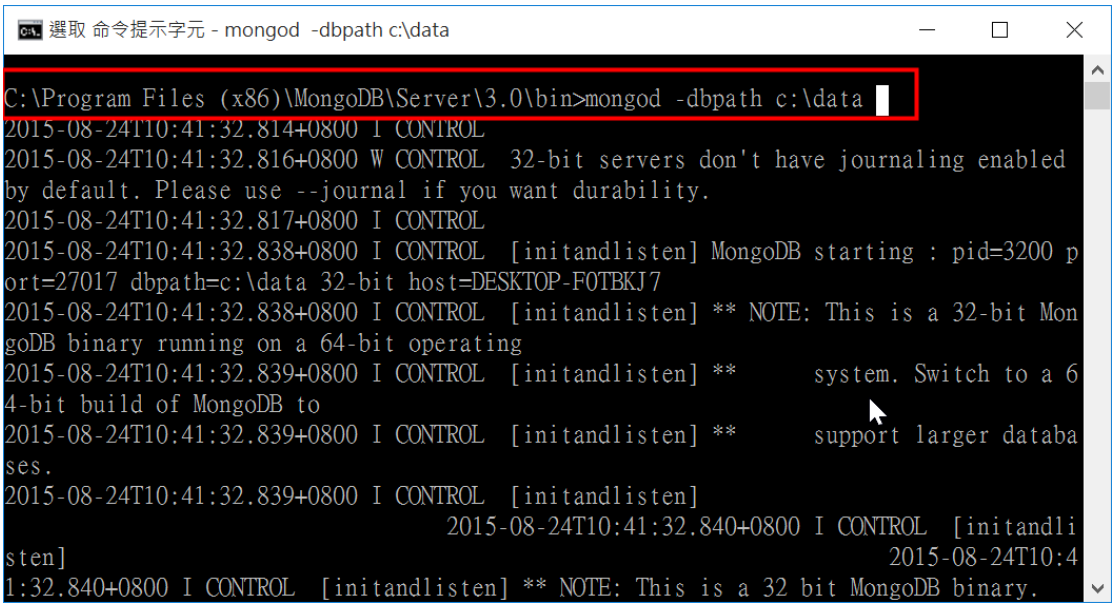
c:\Program Files (x86)\MongoDB\Server\3.0\bin*.*

Name	Ext	Size	Date	Attr
[..]		<DIR>	2015/08/24 10:39	----
bsondump	exe	3,500,032	2015/06/15 16:15	-a--
mongo	exe	5,706,752	2015/06/15 16:17	-a--
mongod	exe	10,802,688	2015/06/15 16:20	-a--
mongod	pdb	104,747,008	2015/06/15 16:20	-a--
mongodump	exe	4,941,312	2015/06/15 16:15	-a--
mongoexport	exe	4,782,080	2015/06/15 16:15	-a--
mongofiles	exe	4,741,632	2015/06/15 16:15	-a--
mongoimport	exe	4,961,792	2015/06/15 16:15	-a--
mongooplog	exe	4,506,624	2015/06/15 16:15	-a--
mongoperf	exe	9,336,832	2015/06/15 16:20	-a--
mongorestore	exe	5,049,344	2015/06/15 16:15	-a--
mongos	exe	5,126,144	2015/06/15 16:20	-a--
mongos	pdb	57,348,096	2015/06/15 16:20	-a--
mongostat	exe	4,691,968	2015/06/15 16:15	-a--
mongotop	exe	4,578,304	2015/06/15 16:15	-a--

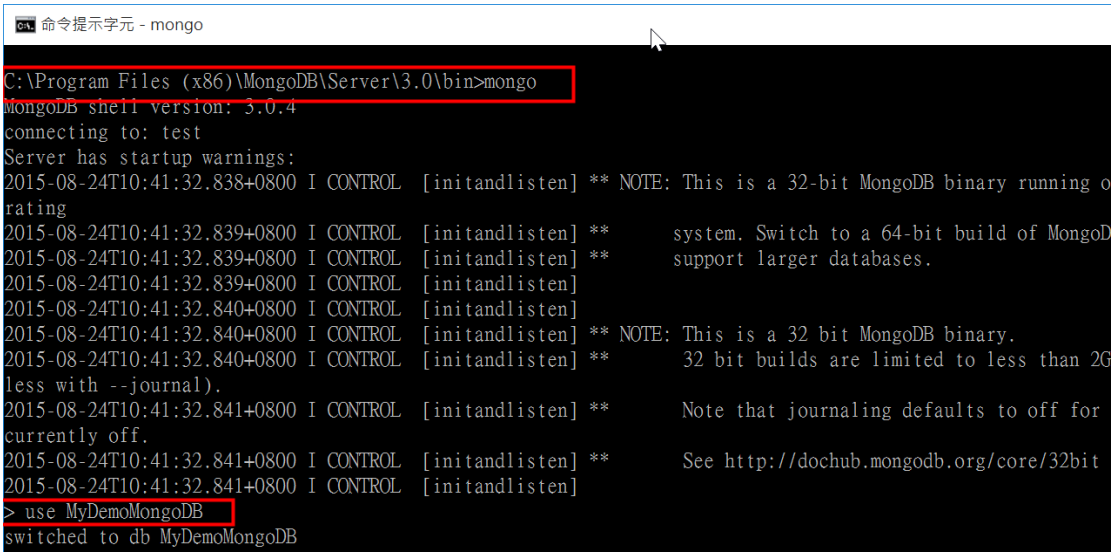
其中的 Mongod.exe 就是 MongoDB 的数据库服务器，执行它就可以启动 MongoDB，当 MongoDB 执行时它会到预定的数据库目录寻找数据库，在 Windows 平台这个预定的数据库目录是

```
C:\data\db\
```

我们可以在启动 MongoDB 时使用 Mongod.exe 的执行选项—dbpath 来指定想使用的预定的数据库目录。例如下面的图形显示在 MongoDB 目录中使用 DOS 命令窗口执行 MongoDB 并使用 c:\data 做为预定的数据库目录：



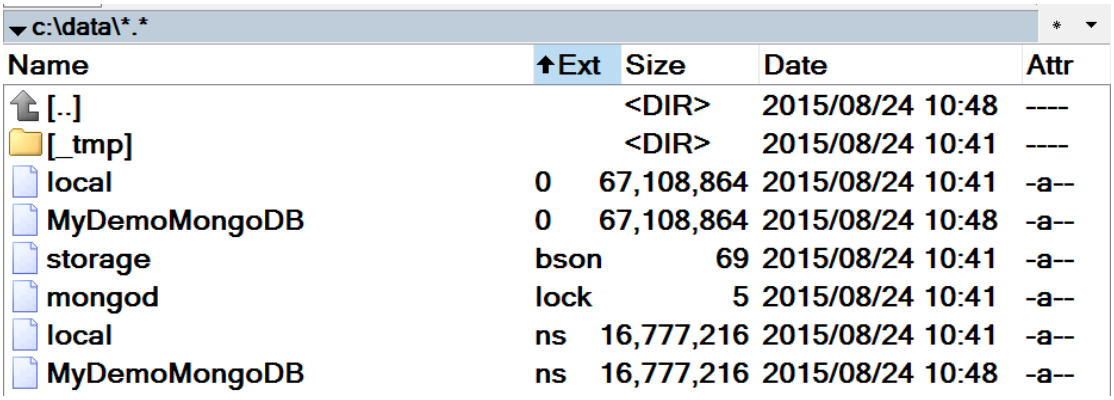
启动 MongoDB 数据库后就可以再使用 MongoDB 的命令工具 Mongo.exe 来管理 MongoDB 数据库，Mongo.exe 使用 JavaScript 语言来执行 MongoDB 命令．例如下面的图形显示在 MongoDB 目录中使用另外一个 DOS 命令窗口执行 Mongo.exe：



例如现在我们要建立一个本节使用的范例 MongoDB 数据表(即 MongoDB 的 Collection) “MyDemoMongoDB”，我们可以在 Mongo.exe 中使用：

```
use MyDemoMongoDB
```

如上图所示，那么在 Mongo.exe 执行完毕之后，我们就可以在 c:\data 目录中看到如下的结果：



c:\data*.*

Name	Ext	Size	Date	Attr
[..]		<DIR>	2015/08/24 10:48	----
[_tmp]		<DIR>	2015/08/24 10:41	----
local	0	67,108,864	2015/08/24 10:41	-a--
MyDemoMongoDB	0	67,108,864	2015/08/24 10:48	-a--
storage	bson	69	2015/08/24 10:41	-a--
mongod	lock	5	2015/08/24 10:41	-a--
local	ns	16,777,216	2015/08/24 10:41	-a--
MyDemoMongoDB	ns	16,777,216	2015/08/24 10:48	-a--

在 c:\data 目录中就可以看到上面的结果，MongoDB 为我们建立了 MyDemoMongoDB 以及其他相关的档案。那么如果我们继续在 Mongo.exe 中执行


```
db.employee.insert({"name" : "李大明", "account" : "LTM", "country" :
"tw","age" : 36})
```

如下图所示，那么就可以在 MyDemoMongoDB 数据库中建立一个 employee 数据表(即 MongoDB 的 Document)建立一笔数据了：

```
connecting to: test
Server has startup warnings:
2015-08-25T12:02:45.776+0800 I CONTROL  [initandlisten] ** NOTE: This is a 32-bit MongoDB b
inary running on a 64-bit operating
2015-08-25T12:02:45.777+0800 I CONTROL  [initandlisten] **      system. Switch to a 64-bit
build of MongoDB to
2015-08-25T12:02:45.777+0800 I CONTROL  [initandlisten] **      support larger databases.
2015-08-25T12:02:45.778+0800 I CONTROL  [initandlisten]
                        2015-08-25T12:02:45.778+0800 I CONTROL  [initandlisten]
2015-08-25T12:02:45.778+0800 I CONTROL  [initandlisten] ** NOTE: This is a 32 bit MongoDB b
inary.
2015-08-25T12:02:45.778+0800 I CONTROL  [initandlisten] **       32 bit builds are limited
to less than 2GB of data (or less with --journal).
2015-08-25T12:02:45.779+0800 I CONTROL  [initandlisten] **      Note that journaling defau
lts to off for 32 bit and is currently off.
2015-08-25T12:02:45.779+0800 I CONTROL  [initandlisten] **      See http://dochub.mongodb.
org/core/32bit
2015-08-25T12:02:45.780+0800 I CONTROL  [initandlisten]
> use MyDemoMongoDB
switched to db MyDemoMongoDB
> db.employee.insert({"name" : "李大明", "account" : "LTM", "country" : "tw","age" : 36})
```

Ok，这些都是使用 MongoDB 的工具和命令来管理和处理 MongoDB，但对于 Delphi 的开发人员来说使用熟悉的知识和技术来管理和处理 MongoDB 才是最重要的。在 DX10 中 FireDAC 加入了支持 MongoDB 的功能，让 Delphi 的开发人员可以同时使用 MongoDB 的 API 或是 FireDAC 组件来管理和处理 MongoDB，这正是下一小节的内容。

6-3 FireDAC 对 MongoDB 的支援

DX10 中的 FireDAC 又增加了许多的新功能，其中最大的新增功能就是支持 MongoDB，FireDAC 同时提供了类别和组件来支持 MongoDB，这也代表 Delphi 开发人员可以藉由 Delphi 类别来呼叫 MongoDB API 来管理和处理 MongoDB，或是使使用 FireDAC 的 MongoDB 组件。

基本上 DX10 使用了下面的类别来封装 MongoDB 的 API：

MongoDB	FireDAC类别
DataBase	TMongoDatabase
Collection	TFDMongoDataSet
Document	TMongoDocument
Field	TField
Mongo Environment	TMongoEnv
Mongo Command	TMongoCommand

由于 MongoDB 使用 JSON/BSON 来处理和封装数据，因此读者也必须了解 DX10 中的新 JSON 框架，例如 TJsonReader，TJsonWriter 和 TJSONCollectionBuilder 等类别，请参考本系列的"Delphi 开发手册一书"。

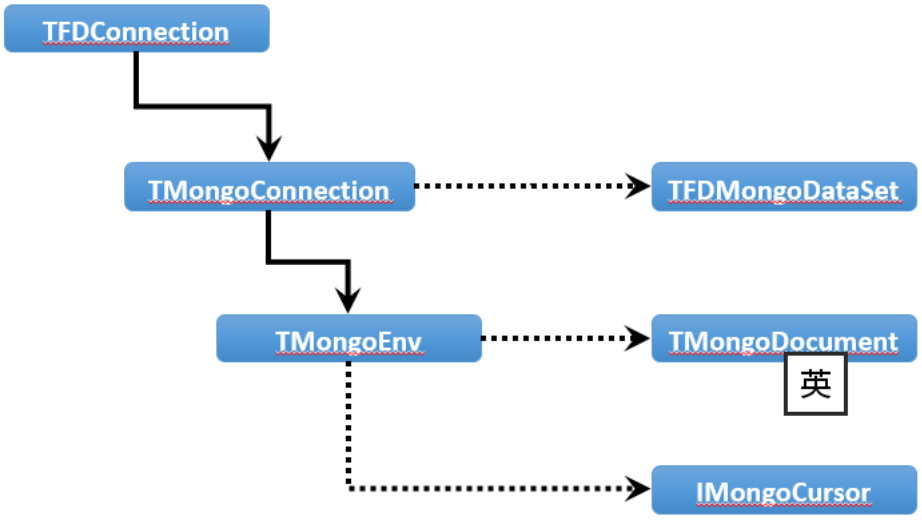
此外 FireDAC 也封装了下面的组件帮助 Delphi 开发人员使用 TDataSet 的观念和技术来使用 MongoDB：

组件	名称	说明
	TFDPhysMongoDriverLink	提供链接和驱动 MongoDB 的能力
	TFDMongoDataSet	封装 MongoDB 的 Database 或是 Collection 的组件
	TFDMongoQuery	封装 MongoDB 执行命令的功能
	TFDMongoPipeline	封装 MongoDB Aggregate 功能的组件

为了让读者了解如何使用这些类别的组件，在下面的 2 个小节中将分别以数个范例来说明。

6-3-1 使用 Delphi 类别处理 MongoDB

基本上要使用 Delphi 类别处理 MongoDB 的话，开发人员可以藉由从 TFDConnection 组件开始取取得 TMongoConnection，TMongoEnv 等类别对象再呼叫其中的方法就可以操作和处理 MongoDB 了，下面图形显示出了如从 TFDConnection 组件开始取得相关 MongoDB 相关的类别对象：



在『Delphi 开发手册』中讨论 DX10 新的 JSON/BSON 框架一章中我们使用了如下的范例：

```
{
        "name" : "王小华",
        "account" : "WSH",
        "country" : "tw",
        "age" : “22”,
        "email" : [
                "wsh@gmail.com.tw",
                "wsh@hotmail.com"
        ]
}
```

现在让我们继续使用这个范例，让我们建立一个名为”FireDACDemoDB”的数据库和名为”FireDACColletions”的 Collection 中建立数个类似"name" : "王小华"结构的 Document。

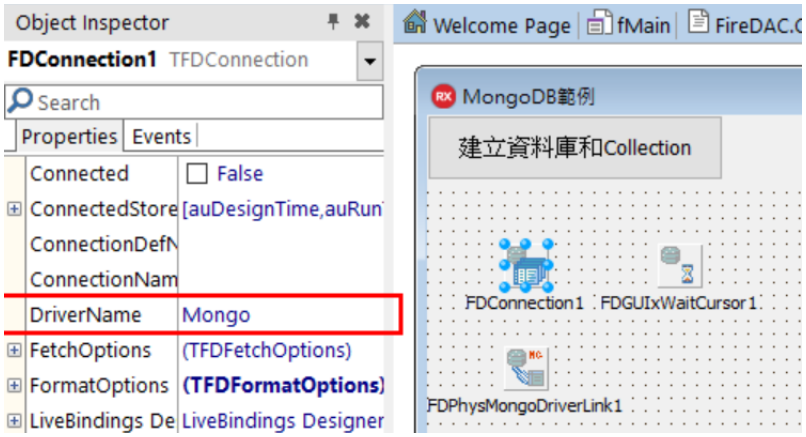
要在 MongoDB 中建立 Collection 和 Document，程序员可以藉由存取 TMongoConnection 和 TMongoEnv 对象完成，一旦取得了 TMongoConnection 对象就可以藉由它的 Collections 特性值在 MongoDB 中建立数据库和 Collection：

```
property Collections[const ADBName, AColName: String]:
TMongoCollection read GetCollectionsProp;
```

请注意 Collections 特性接受 2 个索引参数，一个就是数据库名称，另外一个就是 Collection 名称，而使用数据库名称和 Collection 名称存取 Collections 特性时，如果在 MongoDB 中此时没有此数据库名称和 Collection 名称，那么 MongoDB 就会自动帮我们建立此数据库和 Collection，非常的方便，现在就让我们说明如何完成这 2 个工作。

存取 TMongoConnection 和 TMongoEnv 物件

首先建立一个 VCL Application(FireMonkey Application 也可以)，在主窗体中加入 TFDConnection，TFDPhysMongoDriverLink 和 TFDGUIxWaitCursor 组件，再设定 TFDConnection 组件的 DriverName 特性为 Mongo，如下所示：



在 TFDConnection 类别中的 CliObj 特性值就封装了 TMongoConnection 对象，因此藉由存取此特性值再转换型态为 TMongoConnection 即可：

```
  property CliObj: Pointer read GetCliObj;
```

取特了 TMongoConnection 对象后，在它的 Env 特性值就封装了 TMongoEnv 对象：

```
  property Env: TMongoEnv read FEnv;
```


因此在上面的范例主窗体的 OnCreate 事件中就可以使用下面的程序代码取得 TMongoConnection 对象和 TMongoEnv 对象：

```
procedure TfmMainForm.FormCreate(Sender: TObject);
begin
  FDConnection1.Connected := True;
  FDCon := TMongoConnection(FDConnection1.CliObj);
  FDEnv := FDCon.Env;
end;
```

当然我们要宣告 FDCon 和 FDEnv 如下：

```
  FDEnv: TMongoEnv;
  FDCon: TMongoConnection;
```

接着就可以在主窗体的” 建立数据库和 Collection”按钮中撰写程序代码开始建立此数据库和 Collection 并新增 Document：

```
procedure TfmMainForm.Button1Click(Sender: TObject);
begin
  CreateDBAndCollection;
```

```
  InsertDemoData;
  ShowDocuments;
end;
```

CreateDBAndCollection 方法只需要使用'FireDACDemoDB'数据库名称和'FireDACColletions' Collection 名称那么 MongDB 就会自动帮我们建立'FireDACDemoDB'数据库和'FireDACColletions' Collection，而 RemoveAll()方法是先把'FireDACColletions'中所有的 Documents 删除以便开始新增 Document：

```
procedure TfmMainForm.CreateDBAndCollection;
begin
  FDCon['FireDACDemoDB']['FireDACColletions'].RemoveAll();
end;
```

InsertDemoData 方法呼叫 InsertDocument 方法取得新增了数据的 TMongoDocument 对象，再呼叫 TMongoCollection 对象的 Insert 方法把 TMongoDocument 对象加入到 TMongoCollection 对象中以便把 Document 对象写入 Collection 中：


```
procedure TfmMainForm.InsertDemoData;
var
  FDDoc: TMongoDocument;
begin
  FDDoc := InsertDocument('王小华', 'wsh', 'tw', '22',
'wsh@gmail.com.tw', 'wsh@hotmail.com');
  FDCon['FireDACDemoDB']['FireDACColletions'].Insert(FDDoc);
end;
```

InsertDocument 方法先呼叫 TMongoEnv 的 NewDoc 方法建立 TMongoDocument 对象，再使用类似 TJsonObejectBuilder(请参考"Delphi 开发手册")的方式新增 JSON 型态的数据：

```
function TfmMainForm.InsertDocument(const sName, sAccount,
sCountry, sAge,
  sEMail1, sEmail2: String): TMongoDocument;
begin
  Result := FDEnv.NewDoc;

  Result
    .BeginObject('员工')
```

```
      .Add('name', sName)
      .Add('account', sAccount)
      .Add('country', sCountry)
      .Add('age', sAge)
      .BeginArray('email')
        .Add('信箱 1', sEMail1)
        .Add('信箱 2', sEMail2)
      .EndArray
    .EndObject;
end;
```

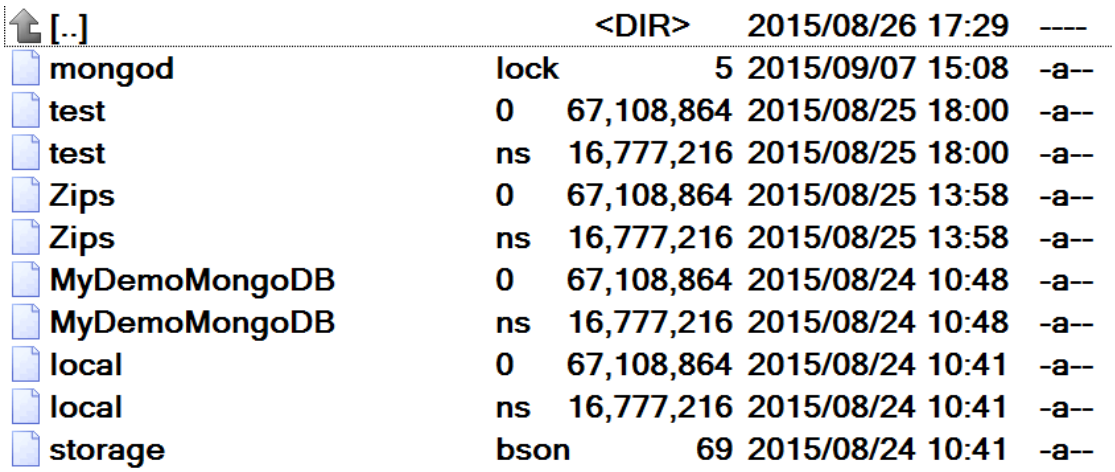
最后的 ShowDocuments 方法是藉由 IMongoCursor 接口一一的显示 'FireDACColletions' Collection 中所有的 Document。

007 行可藉由 TMongoConnection 取得目前使用的 MongoDB 数据库版本信息，010 行呼叫 TMongoCollection 的 Find 方法搜寻到所有的 Document，Find 方法会回传 IMongoCursor 接口，IMongoCursor 接口的 Next 方法可以一一的取得每一个 Document 对象，013 行就可以藉由 IMongoCursor 接口的 Doc 特性取得 TMongoDocument 对象，再把其中的资料以 JSON 格式回传并显示出来：

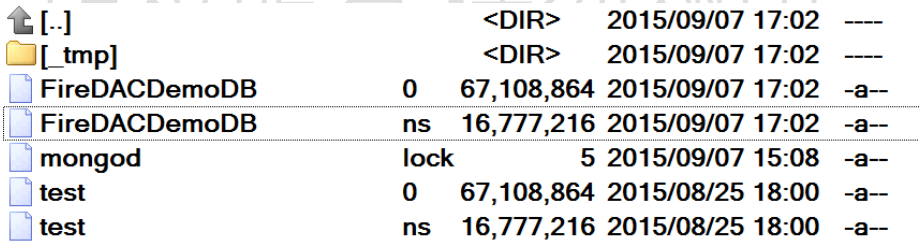

```
001   procedure TfmMainForm.ShowDocuments;
002   var
003     IFDCrs: IMongoCursor;
004     sData : String;
005   begin
006     Memo1.Lines.Clear;
007     Memo1.Lines.Add(FDCon.ServerVersion.ToString());
008     Memo1.Lines.Add(#13#10);
009
010     IFDCrs :=
FDCon['FireDACDemoDB']['FireDACColletions'].Find();
011     while IFDCrs.Next do
012     begin
013       sData := IFDCrs.Doc.AsJSON;
014       Memo1.Lines.Add(sData);
015     end;
016     Memo1.Lines.Add(  #13#10'总笔数 : ' +
017
FDCon['FireDACDemoDB']['FireDACColletions'].Count().Value().ToSt
```

```
ring() );
018    end;
```

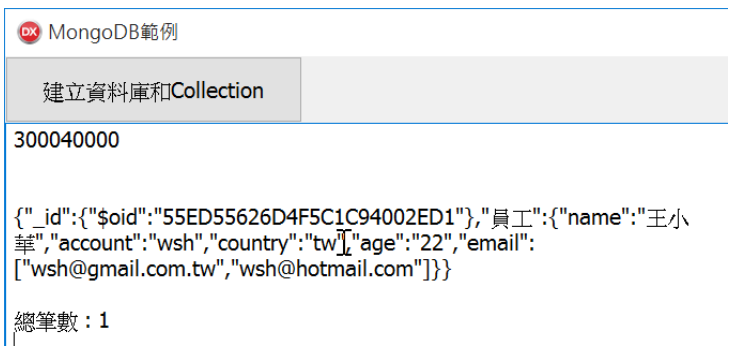
执行上面的范例程序之前下图是笔者 MongoDB 使用的数据目录，现在我们没有看到 FireDACDemoDB 数据库：



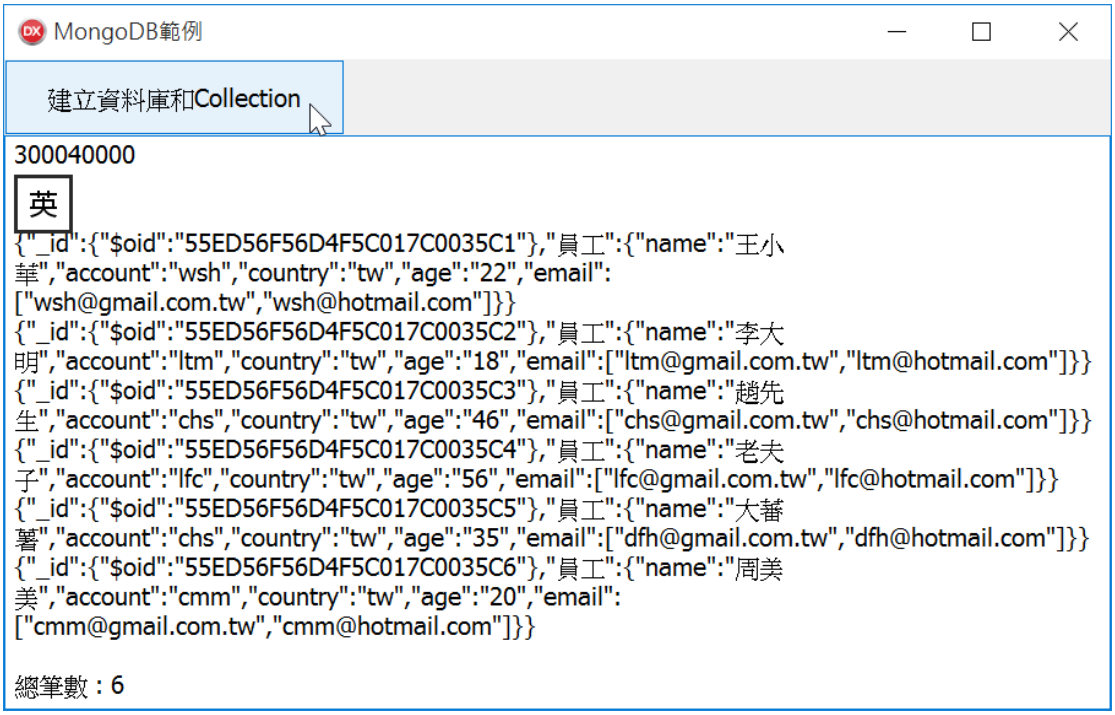
但在执行上面的范例程序之后可以在目录中看到 FireDACDemoDB 数据库被自动建立了：



下图则是此范例程序执行的结果，我们可以看到笔者使用的 MongoDB 数据库版本是 3.04，而且成功的在 FireDACDemoDB 数据库的'FireDACColletions' Collection 中新增了”王小华”这个 Document：



而下图则是在'FireDACColletions' Collection 中新增多个 Document 的结果：



从上面的说明可以了解藉由 TFDConnection 组件程序员可以直接取得封装 MongoDB 功能的类别来处理 MongoDB。

但是 Delphi 的程序员更习惯于使用 TDataSet 的概念来处理资料，因此 DX10 的 FireDAC 也开始封装 MongoDB 的功能到 TDataSet 组件中，下面的小节将说明如何使用 Delphi 程序员熟悉的 TDataSet 概念和技术来使用 MongoDB。

6-3-2 使用 FireDAC 组件处理 MongoDB

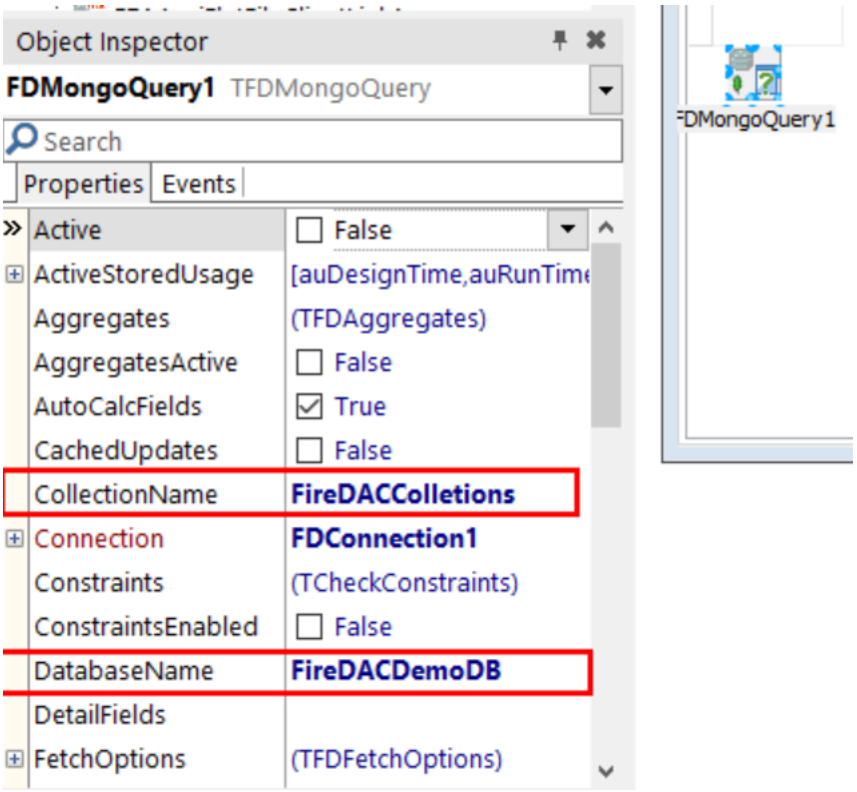
为了让 Delphi 程序员避免自己学习和呼叫复杂的 MongoDB API，因此 DX10 的 FireDAC 提供了 TFDMongoQuery 和 TFDMongoDataSet 等 TDataSet 组件封装了许多 MongoDB API 的功能让 Delphi 程序员以熟悉的方式来使用 MongoDB，如此一来可以减少许多的开发时间，本小节将简单的说明如何使用这 2 个 TDataSet 组件。

> DX10 的 FireDAC 开始了封装 MongoDB API 的工作，但目前尚未完全封装，未来将持续的开发支持 MongoDB 的功能，并未会整合到 IDE 中让 Data Explorer 也可以直接使用 MongoDB。不过这也不用担心，因为 Delphi 程序员如果发现目前不足的地方也可以直接呼叫 MongoDB 类别的方法。

要使用 TFDMongoQuery 组件，程序员需要设定 TFDMongoQuery 组件下面的 3 个特性值：

特性	说明
Connection	设定链接到 TFDConnection 组件
DatabaseName	设定为 MongoDB 的数据库名称
CollectionName	设定为 MongoDB 的 Collection 名称

现在就让我们使用 TFDMongoQuery 组件来搜寻数据。首先在范例程序的主窗体加入 TFDMongoQuery 组件，再如下设定它的 DatabaseName 和 CollectionName 特性值：



接着在主窗体加入一个”搜寻数据”按钮，于它的 OnClick 事件中使用我们已熟悉的 Close()和 Open()方法执行查询。但请注意的是 TMongoQuery 不使用 SQL 语法查询数据，而是使用 JSON 语法查询数据。因此要查询数据我们需要把查询的 JSON 指定给 TMongoQuery 组件的 QMatch 特性：

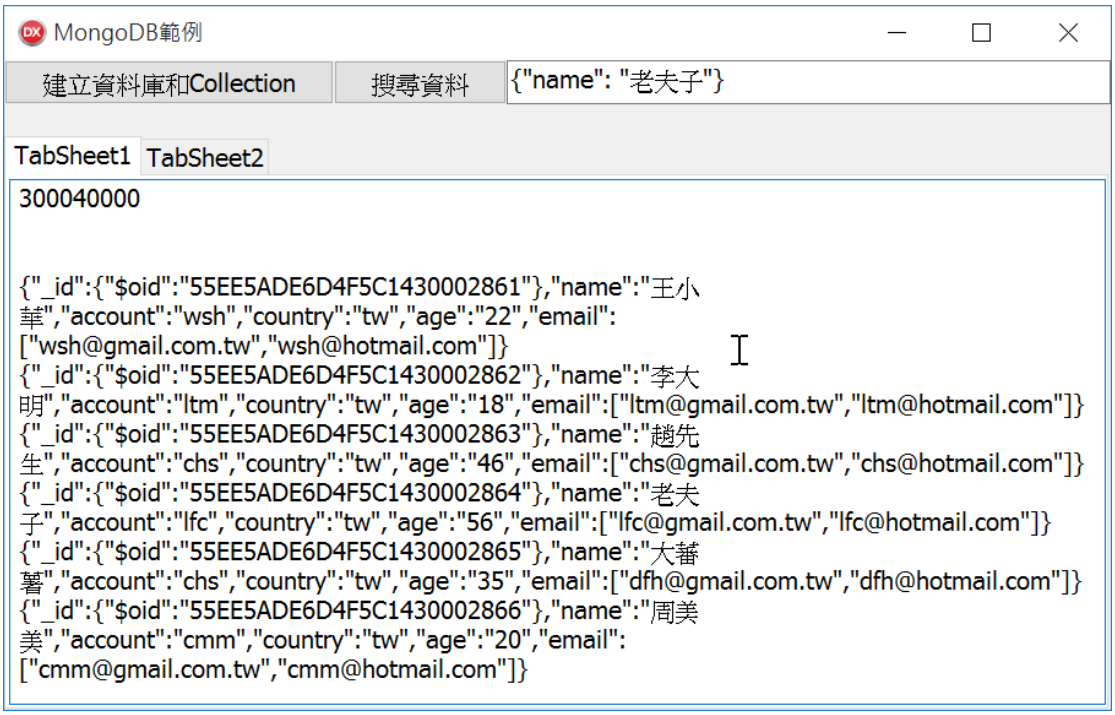
```
procedure TfmMainForm.Button2Click(Sender: TObject);
begin
  FDMongoQuery1.Close();
  FDMongoQuery1.QMatch := Edit1.Text;
```

```
  FDMongoQuery1.Open();
end;
```

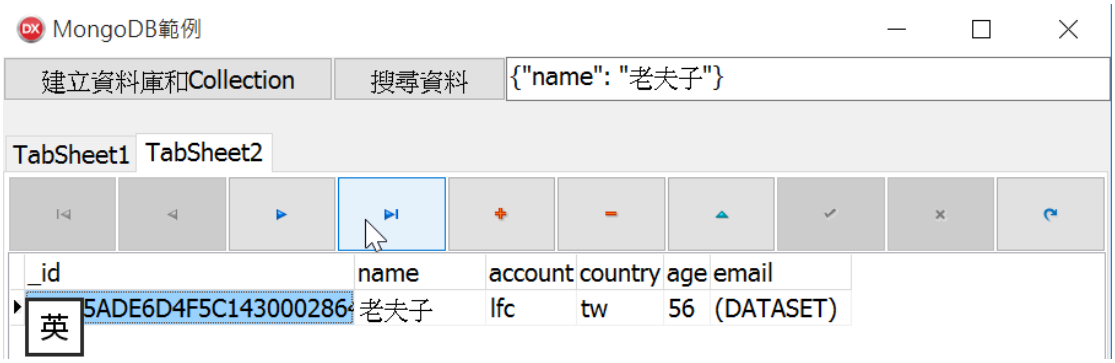
现在执行范例程序并输入：

```
{"name": "老夫子"}
```

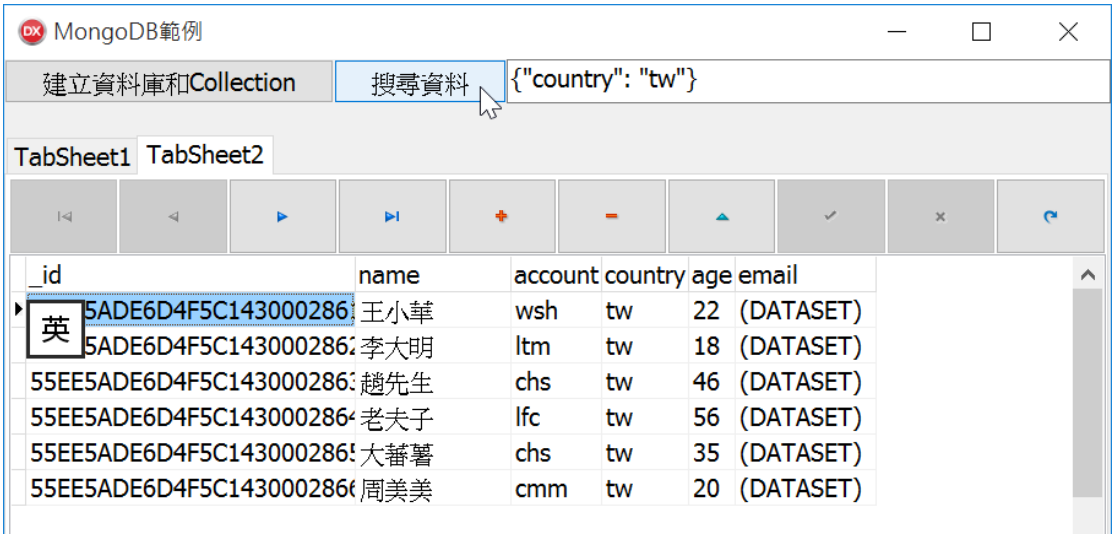
在 Collection 中查询"老夫子"这笔 Document，下图是先在'FireDACColletions' Collection 中新增多个 Document 的结果：



下图则是使用 TFDMongoQuery 组件的 QMatch 特性查询"老夫子"这笔 Document 的结果：



而下图则是查询所有 country=tw Document 的结果：

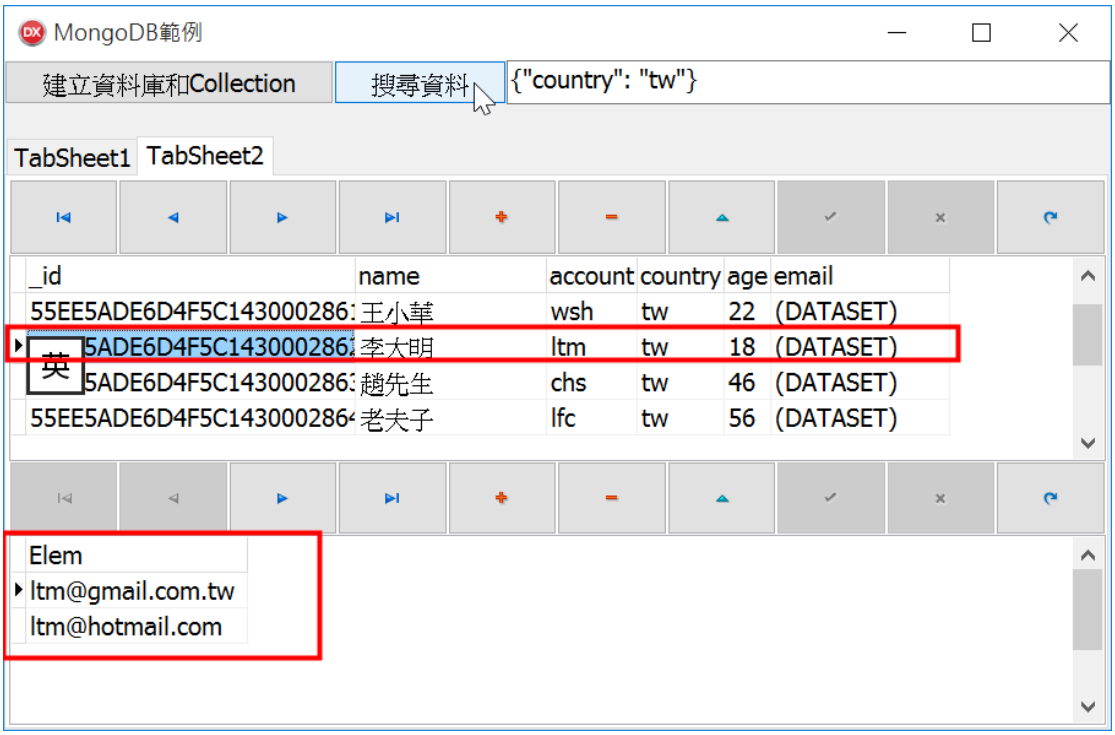


请注意上图中的 email 字段显示的 DATASET，为什么？还记得在前面 email 是一个 JSON 的数组对象吗？当使用 TMongoQuery 组件进行查询时如果一个 MongoDB Document 的字段是 JSON 的数组或是 JSON 对象的话，TMongoQuery 组件就会以内嵌数据集对象(Nested Dataset)来代表，因此我们可以藉由先把这个域值转型为 TDataSetField 对象，再存取 它的 NestedDataSet 特性值就可以取得代表 JSON 的数组或是 JSON 对象的 TDataSet 对象。

例如我们修改一下”搜寻数据”按钮的 OnClick 程序代码，加入下面的 007 行就可以取得 email 字段 JSON 的数组值：

```
001    procedure TfmMainForm.Button2Click(Sender: TObject);
002    begin
003      FDMongoQuery1.Close();
004      FDMongoQuery1.QMatch := Edit1.Text;
005      FDMongoQuery1.Open();
006
007      dsNestedDataSet.DataSet :=
(FDMongoQuery1.FieldByName('email') as
TDataSetField).NestedDataSet;
008      PageControl1.ActivePageIndex := 1;
009    end;
```

下面就是在主窗体中再加入另外一个 TDBNavigator 和 TDBGrid 组件，就可以看到每一个 Document 数据的 email JSON 数组的内容：



使用 TFDMongoQuery 组件处理 MongoDB 的 Document 数据是很方便的，也让 Delphi 程序员可以使用熟悉的 TDataSet 技术，但是我们仍然可以直接使用 TMongoQuery 类别来处理 MongoDB 的 Document 数据。

6-3-3 使用 TMongoQuery 搜寻数据

Delphi 的 TMongoQuery 类别封装了 IMongoCursor 接口，Delphi 程序员可以直接使用 TMongoQuery 类别来处理 MongoDB。例如在 TMongoQuery 类别中提供了 Project，Match 和 Sort 方法可以查询和排序 Collection 中的 Document，而 Open 方法则可以回传包含结果 Document 的 IMongoCursor 接口：

```
function Project(const AJSON: String = ''): TProjection;
function Match(const AJSON: String = ''): TExpression;
function Sort(const AJSON: String = ''): TSort;
function Open: IMongoCursor;
```

现在盲让我们简单的展示一下如何使用 TMongoQuery 类别。

让我们在主窗体中加入一个” 使用 TMongoQuery 搜寻”按钮，在 OnClick 事件中撰写如下的尬式码：

```
procedure TfmMainForm.Button3Click(Sender: TObject);
var
```

```
  IFDCrs: IMongoCursor;
  sData : String;
begin
  Memo2.Lines.Clear;
  IFDCrs := SearchDocument(Edit1.Text);
  while IFDCrs.Next do
  begin
    sData := IFDCrs.Doc.AsJSON;
    Memo2.Lines.Add(sData);
  end;
end;
```

我们先呼叫 SearchDocument 方法搜寻符合条件的 Document，SearchDocument 方法会回传搜寻结果的 IMongoCursor 接口，我们再使用它把搜寻结果显示出来。

SearchDocument 方法在 006 行直接建立 TMongoQuery 对象，008~012 行设定搜寻 JSON 条件，最后 013 行呼叫 TMongoCollection 的 Find 方法进行搜寻：


```
001   function TfmMainForm.SearchDocument(const sMatch: String):
IMongoCursor;
002   var
003     MQuery: TMongoQuery;
004   begin
005     Result := Nil;
006     MQuery := TMongoQuery.Create(FDEnv);
007     try
008       MQuery
009         .Match
010         .Clear
011         .Append(sMatch)
012       .&End;
013       Result :=
FDCon['FireDACDemoDB']['FireDACColletions'].Find(MQuery);
014     finally
015       MQuery.Free;
016     end;
```

```
017    end;
```

下面就是使用 TMongoQuery 搜寻"老夫子"这笔 Document 的结果，我们可以看到直接使用 TMongoQuery 物件也可以成功搜寻我们需要的 Document：

此外我们也可以对 Document 进行排序，我们只需要在 TMongoQuery 的 Sort 特性值中写入排序条件即可，例如现在我们要把所有的 Document 以 Age 字段排序，那我们可以在 S Sort 特性值中写入：

```
{"age" : 1}
```


上面的数值 1 代表要以升幂来排序，如果是要以降序来排序就传入-1：

```
{"age" : -1}
```

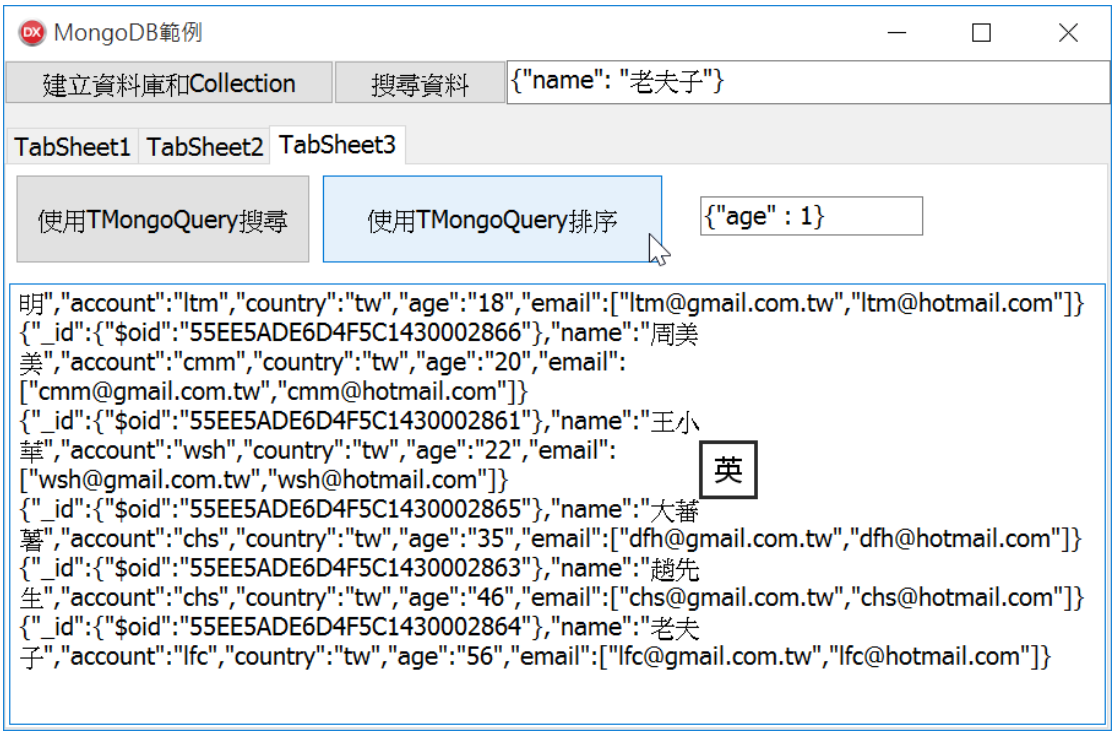
现在就可以在主窗体中加入另一个” 使用 TMongoQuery 排序”按钮并写入如下的程序代码：

```
procedure TfmMainForm.Button4Click(Sender: TObject);
var
  IFDCrs: IMongoCursor;
  sData : String;
begin
  Memo2.Lines.Clear;
  IFDCrs := SortDocument(Edit1.Text, Edit2.Text);
  while IFDCrs.Next do
  begin
    sData := IFDCrs.Doc.AsJSON;
    Memo2.Lines.Add(sData);
  end;
end;
```

SortDocument 方法同样建立 TMongoQuery 对象，并于下面的 014~018 行设定排序条件，最后再呼叫 Find()方法：

```
001   function TfmMainForm.SortDocument(const sMatch: String;
const sSort: String): IMongoCursor;
002   var
003     MQuery: TMongoQuery;
004   begin
005     Result := Nil;
006     MQuery := TMongoQuery.Create(FDEnv);
007     try
008       MQuery
009         .Sort
010         .Clear
011         .Append(sMatch)
012       .&End;
013
014       MQuery
015         .Sort
016         .Clear
017         .Append(sSort)
018       .&End;
019       Result :=
FDCon['FireDACDemoDB']['FireDACColletions'].Find(MQuery);
020     finally
021       MQuery.Free;
022     end;
023   end;
```


下面就是执行的结果，我们可以看到所有的 Document 都是以 Age 字段排序了：



了解了上面的内容后读者应该就可以使用 TFDConnection 和 TFDMongoQuery 等 FireDAC 的组件来开发 MongoDB 的应用程序了，而且在了解了如何直接使用 FireDAC 封装的 MongoDB API 类别后，在读者进一步阅读 MongoDB 的书籍时现在也知道了如何使用封装的 MongoDB API 类别来处理 MongoDB 的资料了。

资料系结篇(Data Binding)

第7章 开发第1个实时数据系结应用程序

Delphi 在 XE2 开始进入跨平台的开发领域，能够同时使用 Delphi 程序语言开发 Win32，Win64，MacOS 和 iOS。由于 VCL 框架只能使用在 Win32 和 Win64 平台，因此如果开发人员需要开发跨平台的图形用户接口应用程序，那么必须使用新的 FireMonkey 框架，由于许多的 FireMonkey 框架的控件都是动态产生的，因此当开发人员需要结合 FireMonkey 控件和数据功能时，传统像 VCL 的数据感知组件的使用方式并不适合使用于 FireMonkey 控件，因为 VCL 的控件是属于静态控件，因此为了让 FireMonkey 控件也能够像 VCL 的数据感知组件提供类似的数据系结能力，Delphi 必须提供另外一种动态绑定数据的能力以便和 FireMonkey 控件共同使用在一起。

实时数据系结(Live Data Binding)是 Delphi XE2 全新推出的数据存取功能，它允许 FireMonkey 控件使用动态的系结表达式(Binding Expression)来系结特定的数据源，一旦系结的数据源数据有变化时，FireMonkey 控件使用动态的系结表达式可以自动重新运算并且显示新的运算结果的资料。由于实时数据系结不但能够系结数据库，也能够系结其他的数据源，例如让控件系结到其他的控件，而且能够使用系结表达式来进行运算，因此提供了比 VCL 数据感知组件更有弹性的能力。

简单的说，XE2 的实时数据系结提供了下面的功能

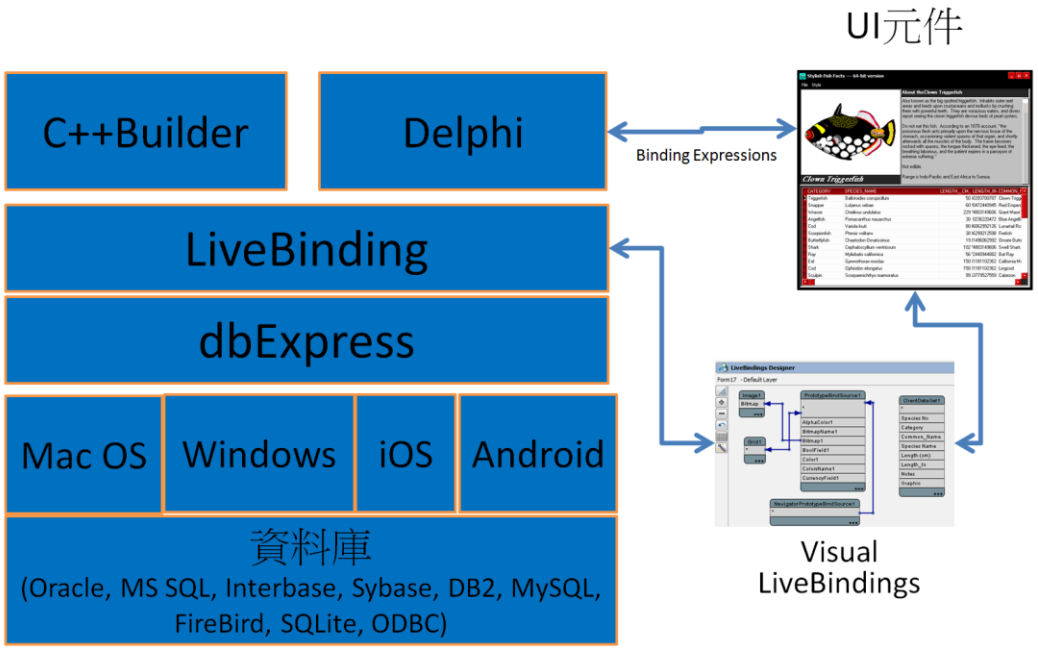
- 提供执行函数式的服务以及指定运算结果给目的控件的能力
- 提供数据型态自动转换的能力
  - 例如当函数式执行完毕之后要指定给目的控件时，实时数据系结能够根据指定的目的控件的数据型态自动转换系结表达式的运算结果为正确的数据型态再指定给目的控件
- 实时数据系结提供运算范围的能力，这是指当执行系结表达式时，开发人员可以指定系结表达式中使用的变量，方法，操作数和特性值等是在什么范围中定

义的。例如实时数据系结提供了 TBindScopeDB 组件来指定系结表达式中使用的变量，方法，操作数和特性值是 TBindScopeDB 组件系结的数据库。实时数据系结也提供 TBindScopeComponent 组件来指定系结表达式中使用的变量，方法，操作数或特性值是 TBindScopeComponent 组件系结的组件中定义的。

- 实时数据系结提供自动更新的能力，当函数式中的数据有变化时，系结表达式能够自动重新运算。

不过由于 XE2 的实时数据系结需要开发人员直接使用系结表达式来撰写，因此造成开发人员在学习使用实时数据系结技术时产生困难。DX10 为了帮助开发人员降低学习系结表达式的困难，因此提供了可视化实时数据系结(Visual LiveBindings)技术，让开发人员藉由使用可视化的设计家来自动产生系结表达式，让开发人员能够快速完成 FireMonkey 或 VCL 的数据库应用程序的开发工作。

虽然实时数据系结技术可以为 FireMonkey 或 VCL 应用程序链接数据源，但实时数据系结在存取数据源时仍然是使用 dbExpress 框架，由于 dbExpress 是跨平台的数据存取引擎，因此使用实时数据系结技术的应用程序也能够执行在不同的平台中。下图说明了实时数据系结，可视化实时数据系结，系结表达式以及 dbExpress 框架之间的关系:



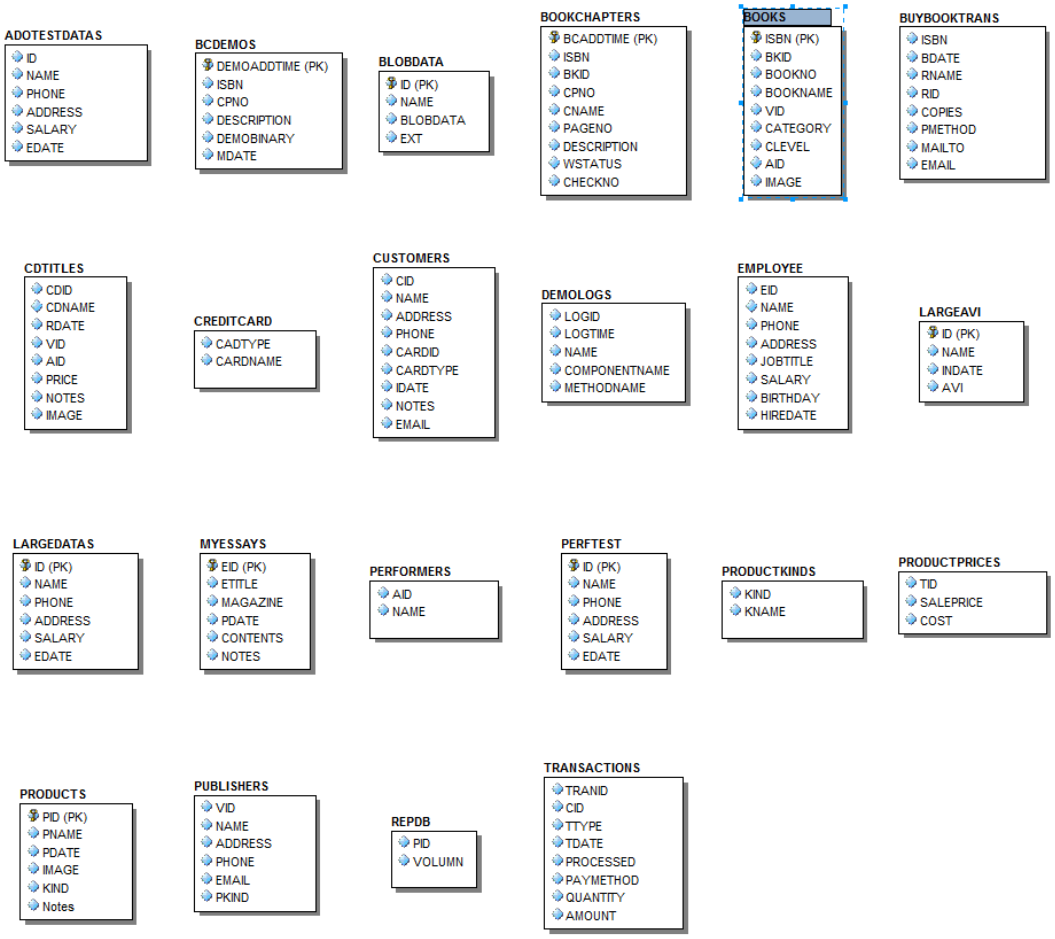
对于使用 DX10 开发 FireMonkey 和 VCL 的数据库应用程序的开发人员来说，学习实时数据系结和 dbExpress 框架都是必要的，在本书的第一部份将说明如何使用实时数据系结技术开发 FireMonkey 的数据库应用程序，第 2 部份将说明如何使用 dbExpress 框架。

虽然实时数据系结主要是为了 FireMonkey 框架开发出来的数据存取技术，但由于实时数据系结技术是独立的框架，因此实时数据系结也可以使用在 VCL 应用程序中，在本章中我们将先说明如何开发实时数据系结应用程序，稍后的章节再逐渐讨论实时数据系结框架较为深入的主题和技术。

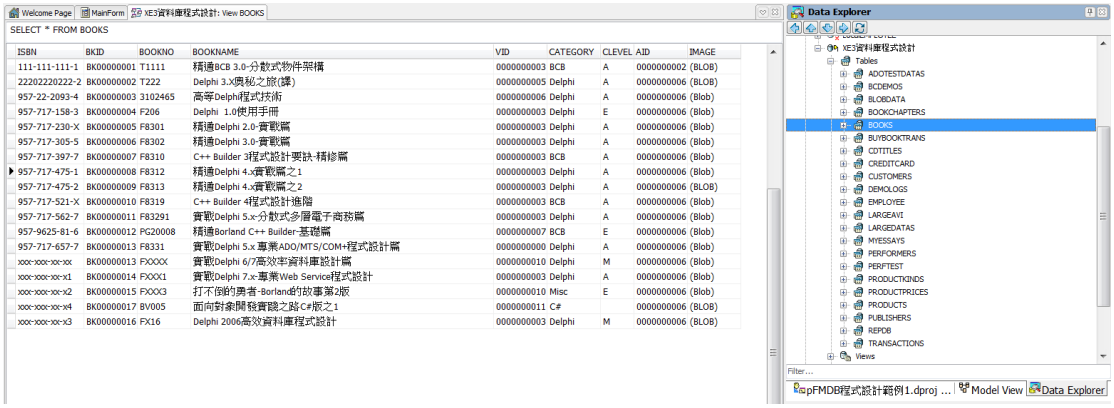
从本章这里之后讨论数据系结技术中同时使用了 dbExpress 和 FireDAC 2 种不同的技术展示资料系结技术可和任一 Delphi 的数据存取技术共同使用，如果读者不想使用 dbExpress 技术，那可以自行根据前面章节对于 FireDAC 的了解把 dbExpress 组件都置换成使用 FireDAC 组件即可。

7-1 开发第一个 FireMonkey 数据库应用程序

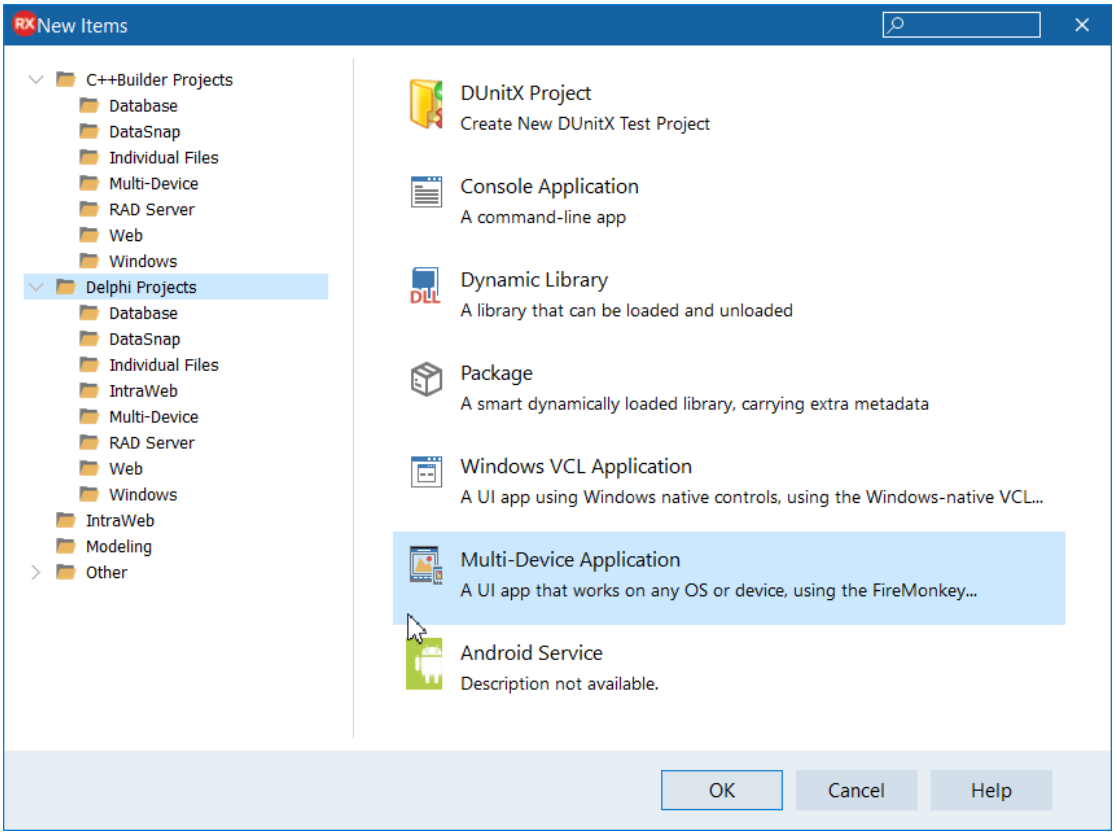
本书将使用 InterBase 做为范例数据库，在其中有数个数据表之间拥有 Master/Detail 的关系，在说明如何使用可视化实时数据系结技术逐渐开发 FireMonkey 数据库应用程序的流程中，将一一的带入使用这些数据表。不过让我们先从简单的开发工作开始说明如何使用可视化实时数据系结。



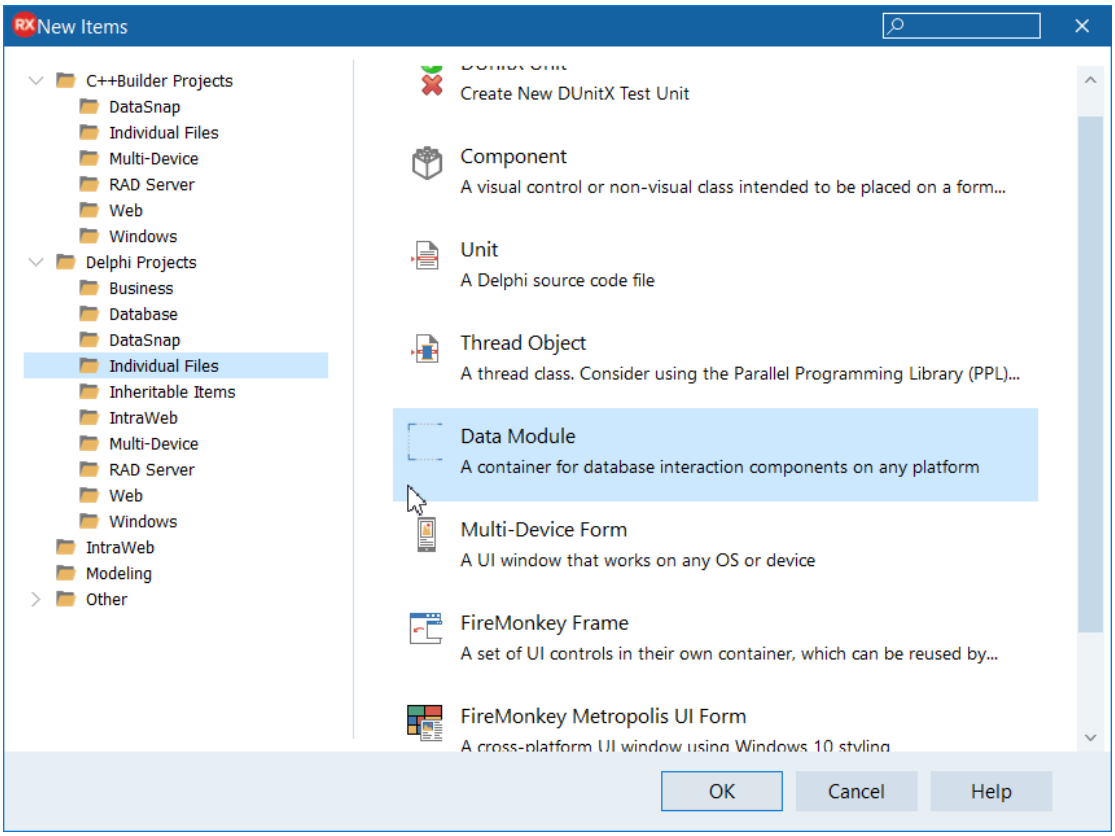
让我们先从 BOOKS 范例数据表开始，下面是 BOOKS 数据表中的数据，现在就让我们使用实时数据系结开发一个 FireMonkey 数据库应用程序，让读者了解 Delphi 10.3 的可视化实时数据系结技术是多么的容易使用。



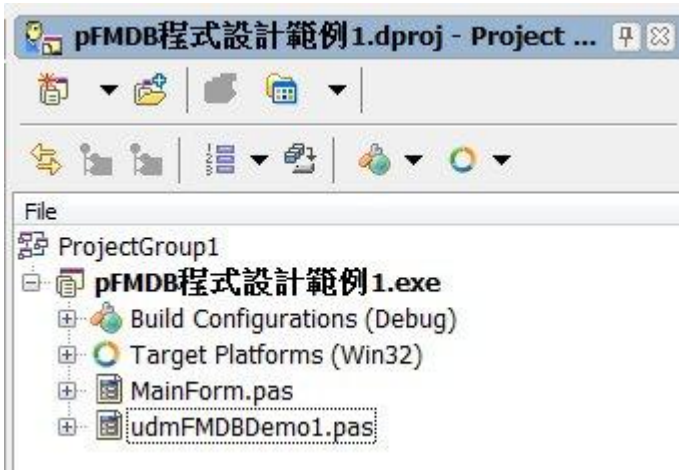
请先建立一个 Multi-Device 桌面应用程序项目，



接着在项目中建立一个数据模块：



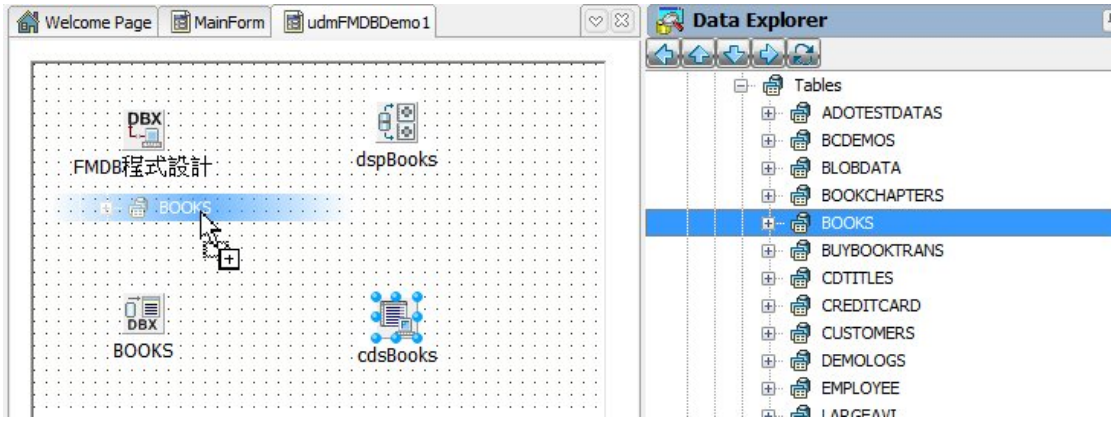
最后储存此 Multi-Device 桌面应用程序项目如下所示:



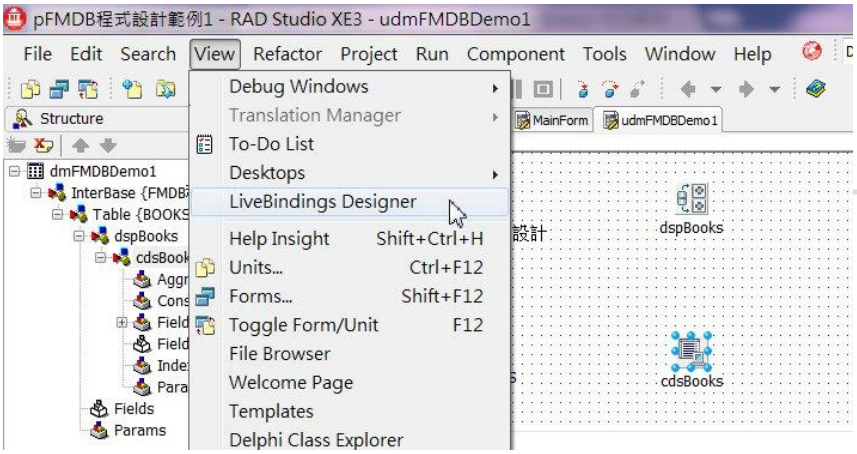
开启数据模块从 Data Explorer 页面中拖曳 BOOKS 数据表到数据模块中，数据模块即会自动产生 TSQLConnection 和 TSQLDataSet 组件，接着再于数据模块中放入 TDataSetProvider 和 TClientDataSet 组件，并且设定它们的特性值如下:

组件	组件名称	设定的特性值
TDataSetProvider	dspBooks	DataSet = BOOKS
TClientDataSet	cdsBooks	ProviderName = dspBooks

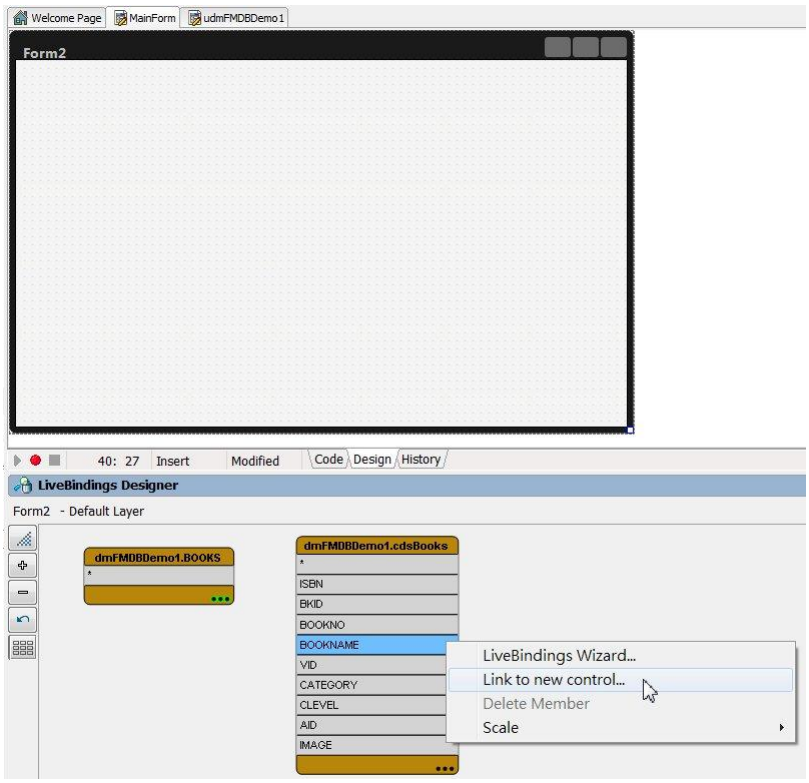
此时数据模块应该如下所示:



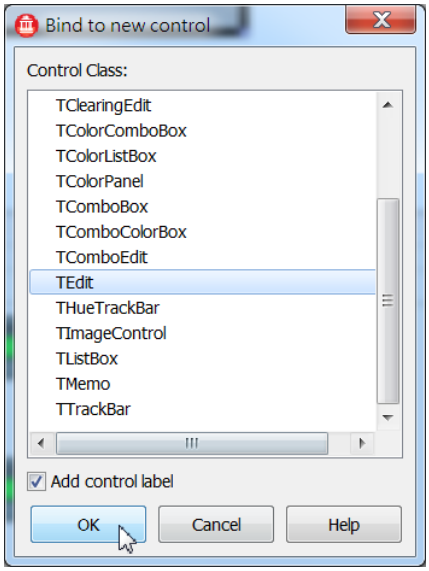
回到主窗体，点选 File|Use Unit…菜单使用数据模块，接着点选 View|LiveBindings Designer 启动可视化实时数据系结设计家，如下所示:



由于主窗体使用了数据模块，因此读者可以从下图中看到可视化实时数据系结设计家中显示了数据模块中的 cdsBooks 实体(Entity)，假设现在我们希望 cdsBooks 中的 BOOKNAME 字段显示在主窗体中，那么我们可以在可视化实时数据系结设计家中点选 cdsBooks 的 BOOKNAME 字段，再点选鼠标右键，就会出现一个快捷菜单，请选择『Link to new control...』 如下图所示:

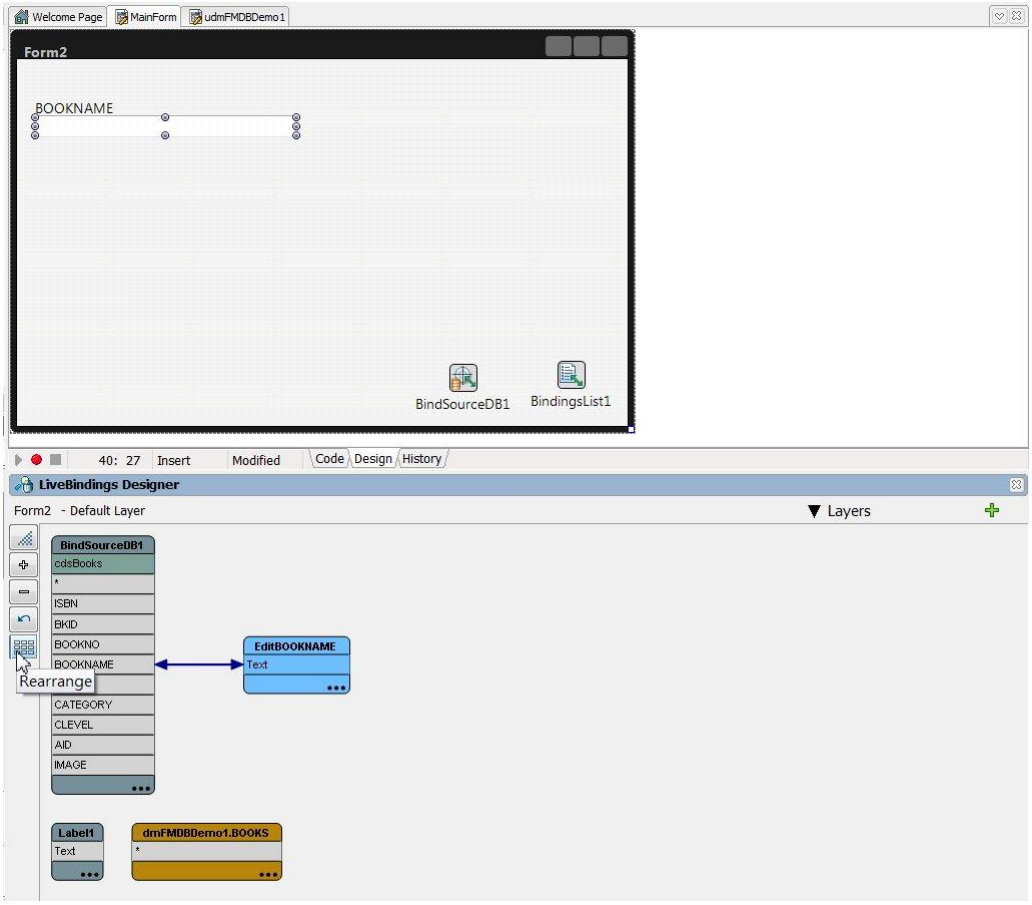


此时可视化实时数据系结精灵会显示如下的对话盒，让您选择希望系结 BOOKNAME 字段的控件，您可以浏览对话盒中出现的控件，现在让我们选择 TEdit 组件让 BOOKNAME 字段的数值显示在 TEdit 组件中，并且点选 OK:

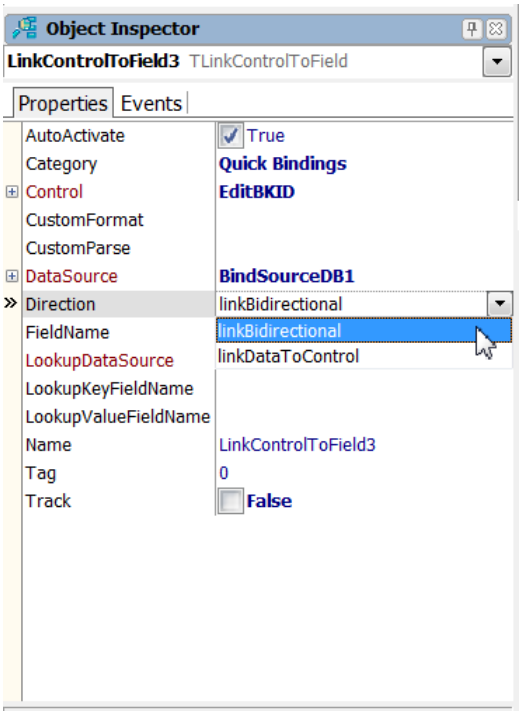


此时可视化实时数据系结精灵会显示如下的对话盒，让您选择希望系结 BOOKNAME 字段的控件，您可以浏览对话盒中出现的控件，现在让我们选择 TEdit 组件让 BOOKNAME 字段的数值显示在 TEdit 组件中:

接着读者就可以在主窗体中看到一个 TEdit 组件，而且在可视化实时数据系结设计家中读者可以看到 cdsBooks 实体和一个新出现名为 EditBOOKNAME 的实体之间拥有一个双向箭头的实线，双向箭头的实线代表可双向更新数据的关系，也就是说 BOOKNAME 字段的数值可以出现在此 TEdit 组件中，如果用户修改 TEdit 组件中显示的 BOOKNAME 字段数值，那么这也会更新回 BOOK 数据表中。



要改变实体之间箭头的方向，例如如果我们希望 BOOKNAME 字段的数值在 TEdit 组件中只能显示而不能修改，那么开发人员可以在可视化实时数据系结设计家中点选双向的箭头，再于对象查看器中修改它的 Direction 特性值，例如如果如下图改变 Direction 特性值为『linkDataToControl』，那么 TEdit 组件中的 BOOKNAME 域值就是只读的。



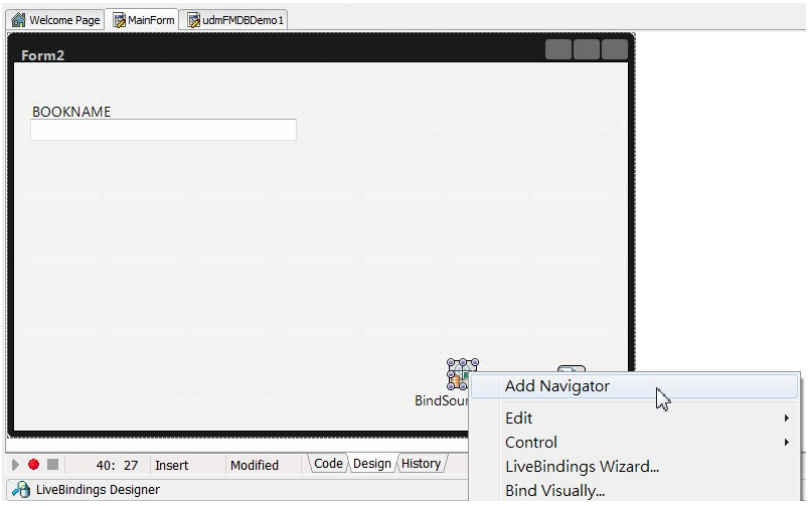
主窗体中除了 TEdit 组件之外，也会自动产生 TBindSourceDB 和 TBindingList 组件。TBindSourceDB 组件是链接到数据模块中 cdsBooks 的组件，它负责协调和管理数据源之中的数据，而 TBindingList 组件则负责产生和执行其包含的系结表达式，而开发人员使用可视化实时数据系结设计家自动产生的系结表达式也管理在 TBindingList 组件中。

现在如果您数据模块中的 cdsBooks 组件是开启的，那么您就可以在主表格中看到 TEdit 组件显示了 BOOKNAME 字段的数值。

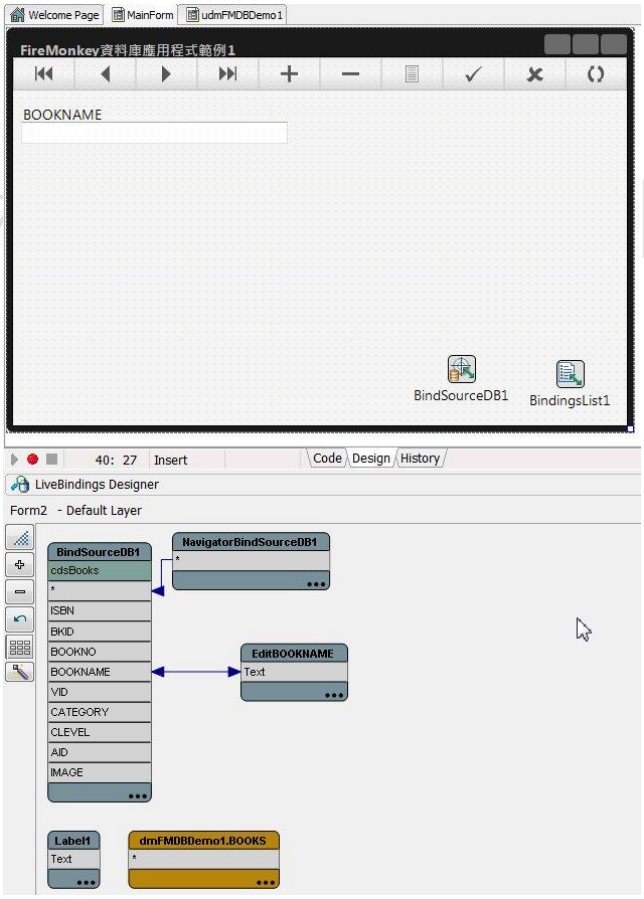
现在让我们在主表格中加入一个 Navigator 让我们能够在主表格中来回浏览 BOOKS 数据表中的数据，请点选主窗体中的 TBindSourceDB 组件，再点选鼠标右键，从快捷菜单中选择『Add Navigator』选项，如下所示:

此时主窗体中就会出现 TBindNavigator 组件，请设定它的特性值如下:

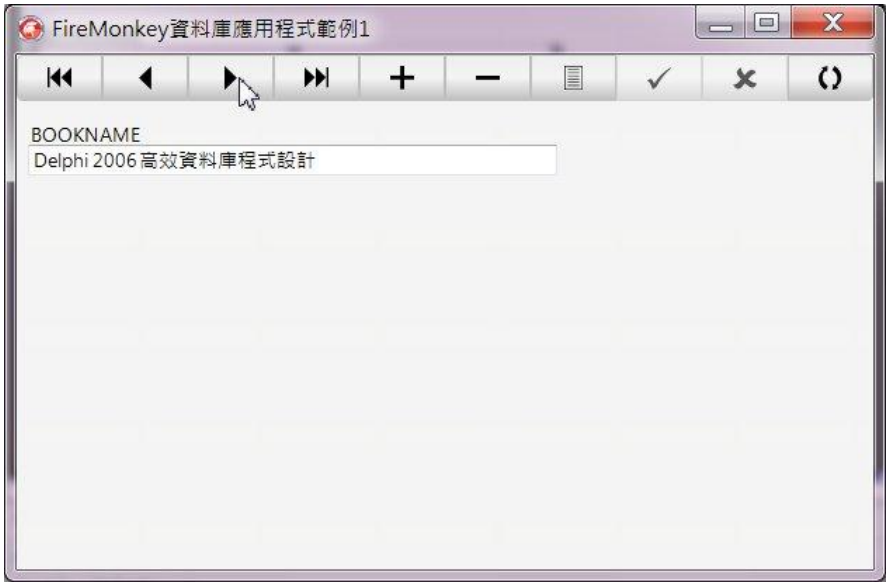
组件	特性值
TBindNavigator	Aligh = alTop



现在主窗体应该看起来如下所示，第一个范例 FireMonkey 数据库应用程序也准备好执行了:



请按下 Shift+Ctrl+F9 执行范例应用程序，就可以看到如下的结果画面，BOOKS 数据表中的 BOOKNAME 字段的数值果然可以出现在 TEdit 组件，而且如果点选主窗体上方的 TBindNavigator 组件中的按钮也能够在数据之间浏览了。现在我们已经成功的使用可视化实时数据系结完成了第一个范例 FireMonkey 数据库应用程序。

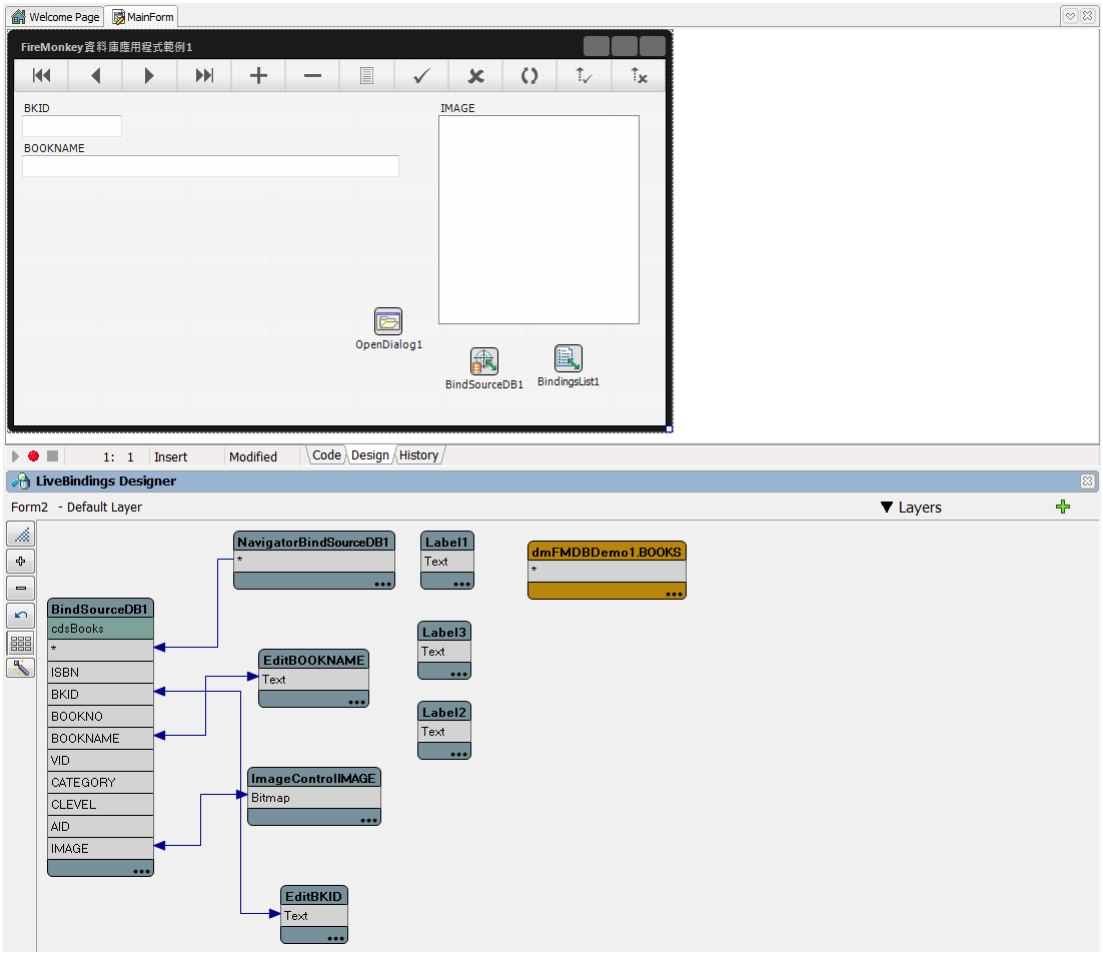


现在请结束范例 FireMonkey 数据库应用程序回到主窗体，再设定 TBindNavigator 组件的 VisibleButtons 特性值如下:

组件	特性值
TBindNavigator	VisibleButtons = [nbFirst,nbPrior,nbNext,nbLast,nbInsert,nbDelete, nbEdit,nbPost,nbCancel,nbRefresh, nbApplyUpdates,nbCancelUpdates]

设定 TBindNavigator 组件的 VisibleButtons 特性值如上所述就会加入 ApplyUpdates 和 CancelUpdates 按钮，让 TBindNavigator 组件能够把数据更新回 cdsBooks 代表的 BOOKS 数据表中。

接着再使用可视化实时数据系结设计家，如同前面系结 BOOKNAME 字段到 TEdit 组件一样的方式把 BKID 字段系结到另外一个 TEdit 组件以及把 IMAGE 字段系结到 TImageControl 组件，如下所示:

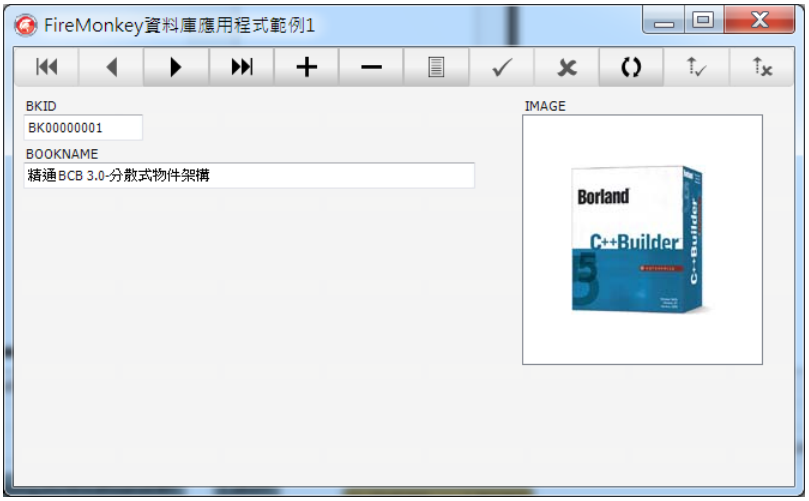


現在主窗体能够显示两个额外的 BOOKS 数据表的字段资料了，请注意 cdsBooks 实体和 TImageControl 组件之间是双向箭头，这代表我们可以更改 TImageControl 组件中显示的图形而实时数据系结会把图形数据自动更新回 BOOKS 数据表中，我们当然也可以直接使用 dbExpress 框架更新图形数据，请在主窗体中加入一个 TOpenDialog 组件并且在 TImageControl 组件的 OnMouseDown 事件处理函式中撰写如下的程序代码:

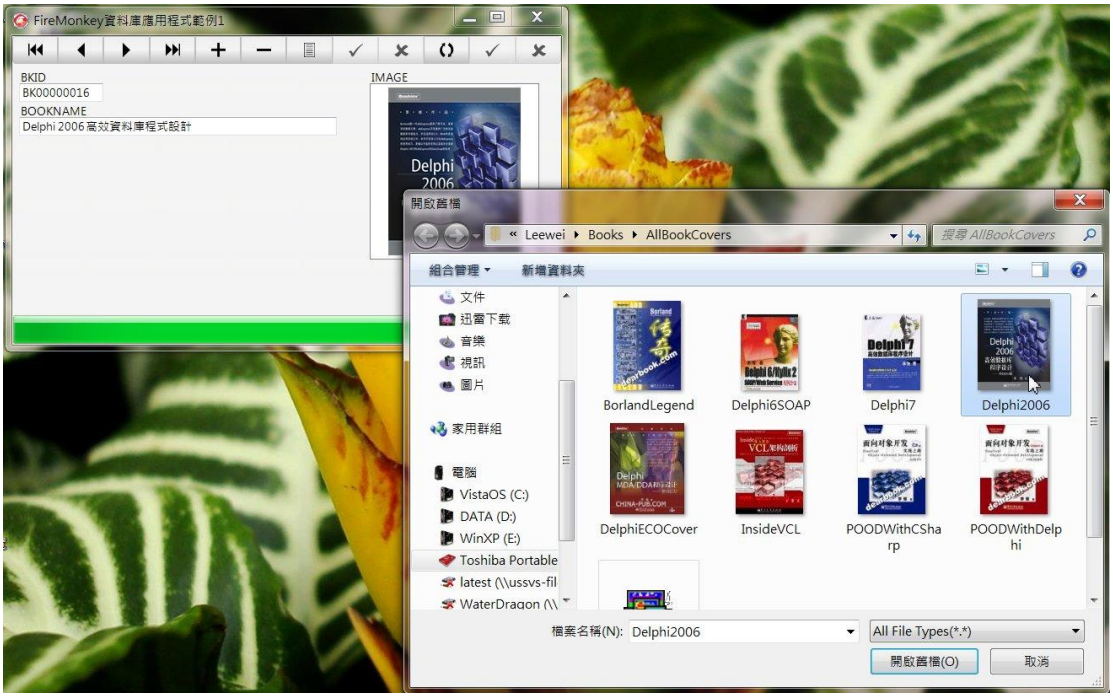
```
procedure TForm2.ImageControlIMAGEMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Single);
begin
  if (Button = TMouseButton.mbRight) then
  begin
    if (OpenDialog1.Execute) then
    begin
      dmFMDBDemo1.cdsBooks.Edit;
      dmFMDBDemo1.cdsBooksIMAGE.LoadFromFile(OpenDialog1.FileName);
      dmFMDBDemo1.cdsBooks.Post;
      dmFMDBDemo1.cdsBooks.ApplyUpdates(0);
```

```
    end;
  end;
end;
```

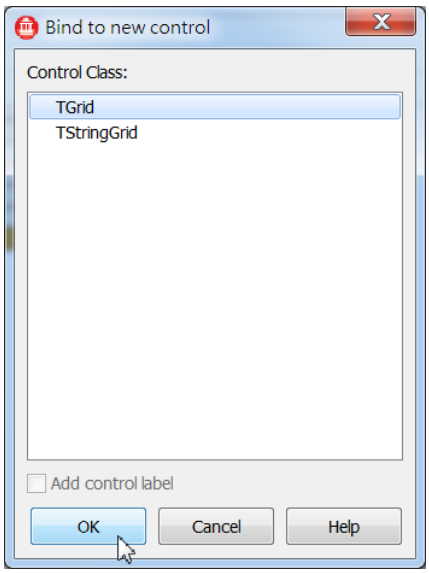
再执行范例 FireMonkey 数据库应用程序，就可以看到 TBindNavigator 多了个 ApplyUpdates 的按钮，BKID 和 IMAGE 字段的数据都会出现了，如下所示:



如果在 TImageControl 中点选鼠标右键就可以看到如下图会出现对话盒让您加载一个新的图形并且加载到 BOOKS 数据表中的 IAMGE 字段中，再点选上方 Navigator 中的 ApplyUpdates 按钮之后图形的数据就会真正的更新回数据库之中了。

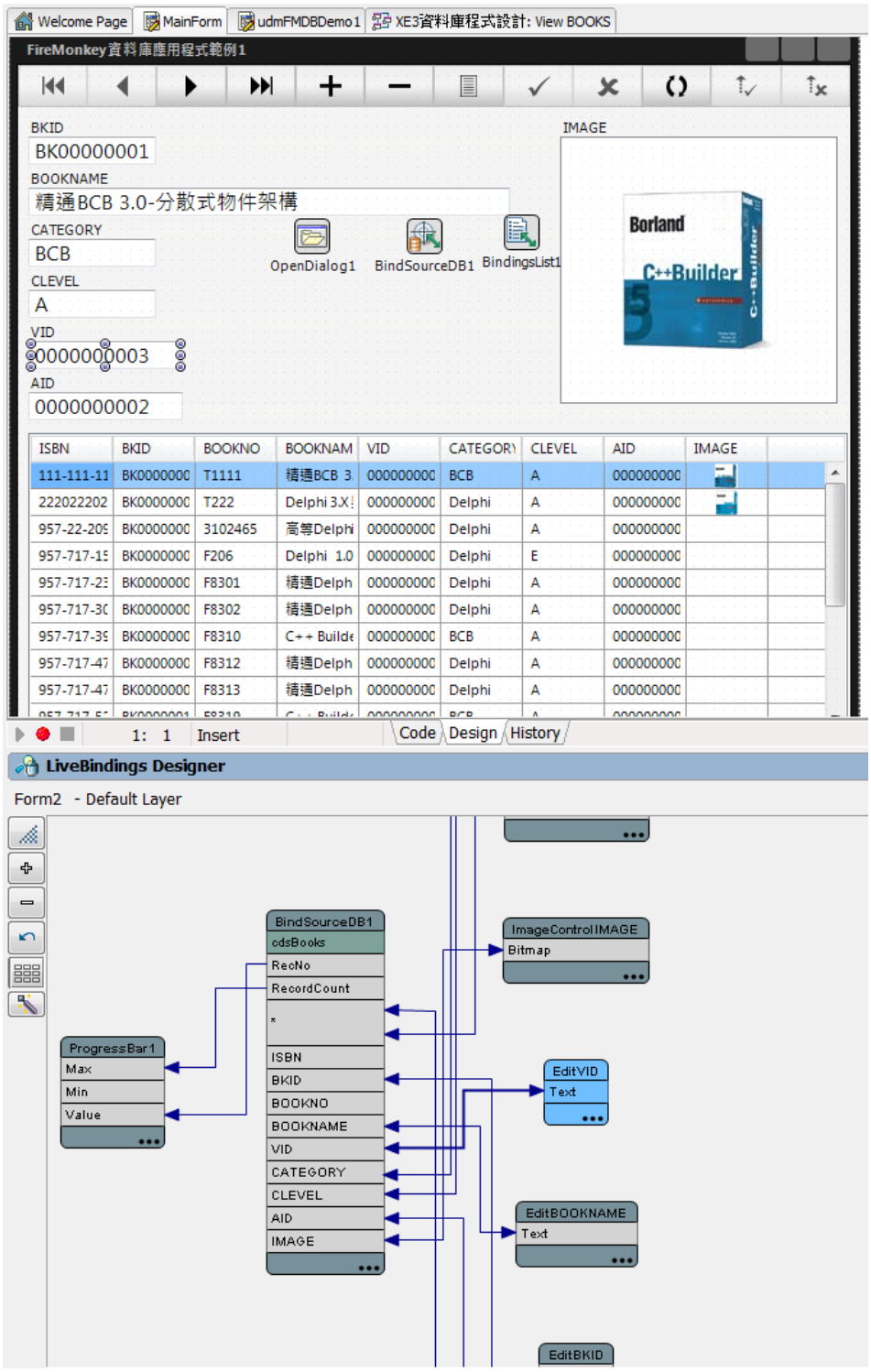


让我们继续在主窗体中加入一些组件来显示 BOOKS 数据表更多的资料，首先让我们加入一个 TGrid 组件。请点选可视化实时数据系结设计家中 cdsBooks 样例中的『*』字段然后再点选鼠标右键，从突显式选单中选择『link to new control』选项，可视化实时数据系结精灵便会显示如下的对话盒，请双击选择其中的 TGrid 组件:

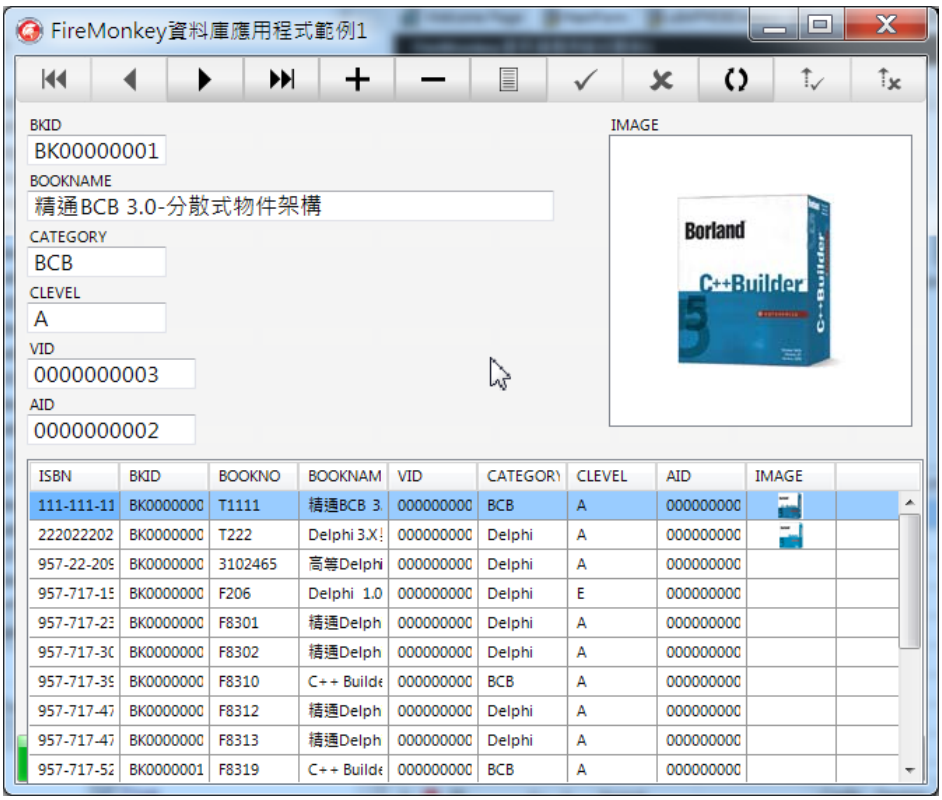


接着再于主窗体中放入 2 个 TEdit 组件，分别命名为 EditVID 和 EditAID，于可视化实时数据系结设计家中先使用鼠标左键点选 cdsBooks 实体中的 VID 字段在不放开鼠标左键的情形下拖曳鼠标到 EditVID 实体的 Text 特性以系结这 2 者，此时在可视化实时数据系结设计家中这 2 者之间就会出现一个双向箭头的线条。

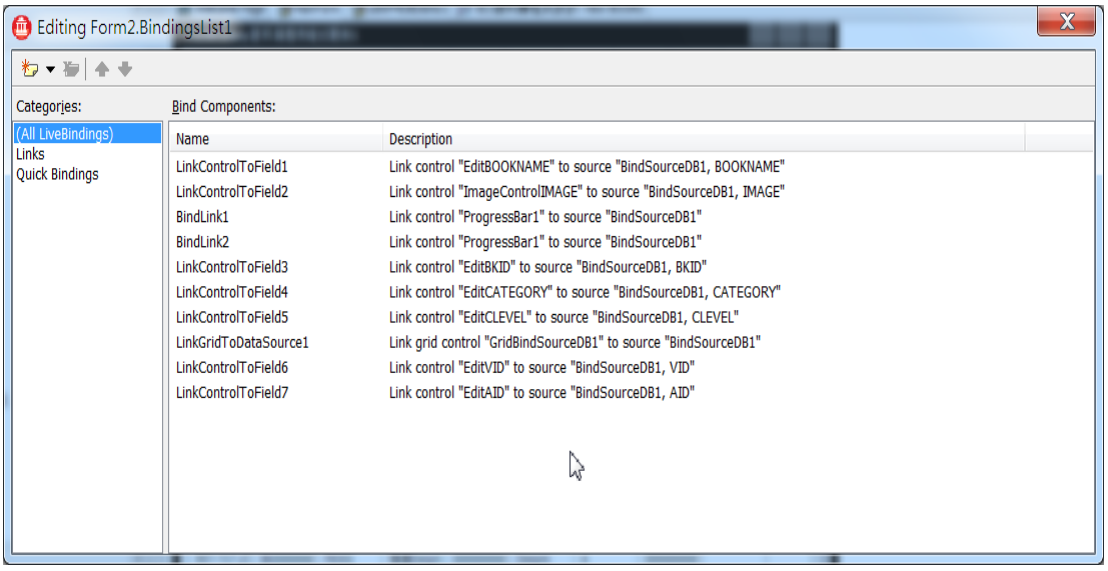
再使用同样的方法系结 cdsBooks 实体中的 AID 字段和 EditAID 的 Text 特性，这 2 者之间也会出现一个双向箭头的线条，最后主窗体和可视化实时数据系结设计家看起来如下所示:



现在您可执行此 FireMonkey 应用程序并且试着修改数据再藉由上方 Navigator 组件中的 ApplyUpdates 按照把数据更新回 BOOKS 数据表中，您会发现实时数据系结果然可以真的把数据更新回 InterBase 数据库中，现在您已经完成了 FireMonkey 第 1 个数据库应用程序开发了，使用可视化实时数据系结技术真的很简单，不是吗?



在离开本小节之前请您注意在主窗体中有一个 TBindingsList 组件，事实上在前面我们所有使用可视化实时数据系结设计家进行的系结都会被自动转换为系结表达式储存在此 TBindingsList 组件中，如果您双击主窗体中的 TBindingsList 组件就会看到如下的编辑器：

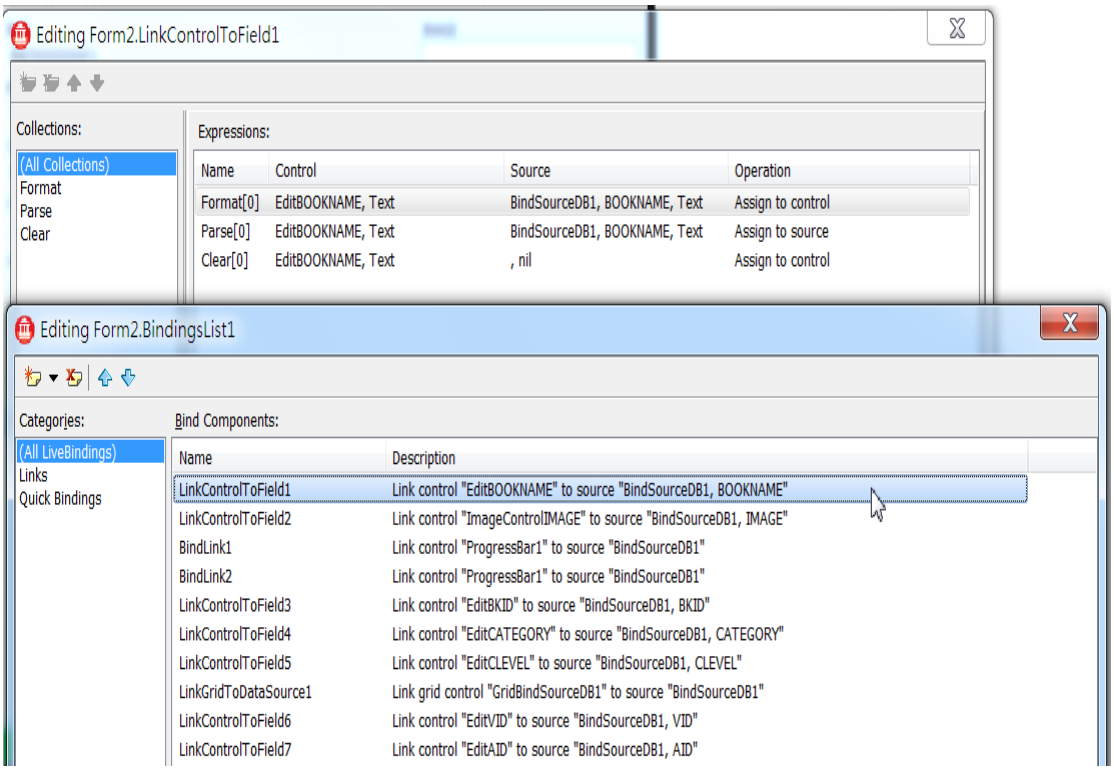


其中的内容就是前面所有在我们可视化实时数据系结设计家中进行的工作而自动产生的系结表达式，您可以点选其中任何的系结表达式并且在对象查看器中观察它，例如请双击编辑器中第 1 个系结表达式您就可以看到这个系结表达式的详细信息，在新出现的

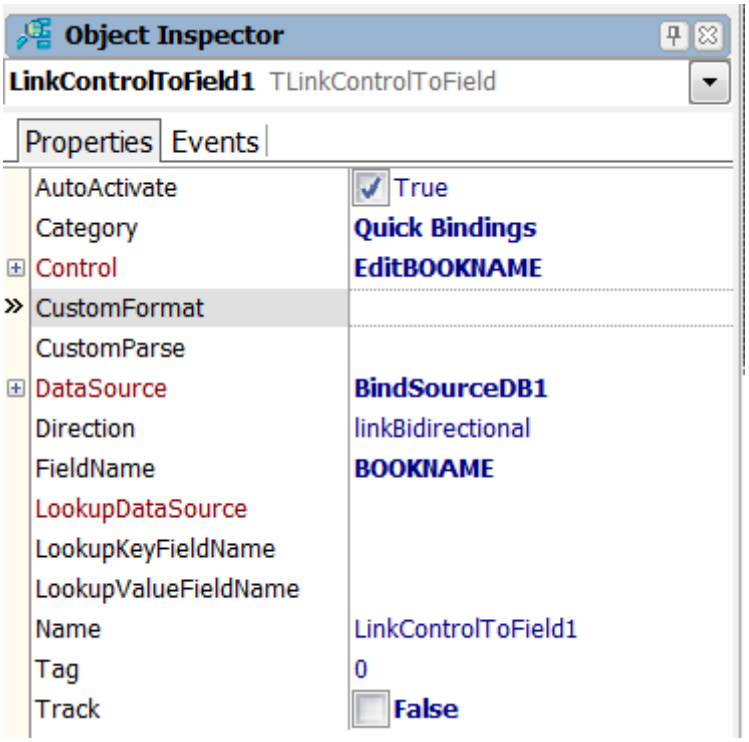
编辑器中您可以看到这个系结表达式的意义就是把 BindSouceDB1(即 cdsBooks)中的 BOOKNAME 字段的 Text 数值系结到主窗体中 EditBOOKNAME 这个 TEdit 组件的 Text 特性值中，这也就是说把 BOOKS 数据表中的 BOOKNAME 字段的数值显示在 EditBOOKNAME 组件中。

在下图上方的第 2 个系结表达式则是反相把 EditBOOKNAME 的 Text 特性值写回 BindSouceDB1(即 cdsBooks)中的 BOOKNAME 字段的 Text 数值，这也就是说把 EditBOOKNAME 组件中的数值更新回 BOOKS 数据表的 BOOKNAME 字段。

至于第 3 个系结表达式则是代表清除 EditBOOKNAME 中的数值。



如果我们点选上面编辑器中的 LinkControlToField1 对象并且观察对象查看器的话就可以看到如下的内容:



我们从上面的对象查看器中可看到 LinkControlToField1 的 Category 特性值是 Quick Bindings，这种系结对象是 DX10 才出现的新型态的系结对象，它可快速绑定控件和数据源，这种型态的系结对象主要是搭配可视化实时数据系结设计家使用的，但 DX10 尚有许多其他型态的系结对象可让开发人员使用，现在就让我们使用其一个简单型态的系结对象:TBindLink。

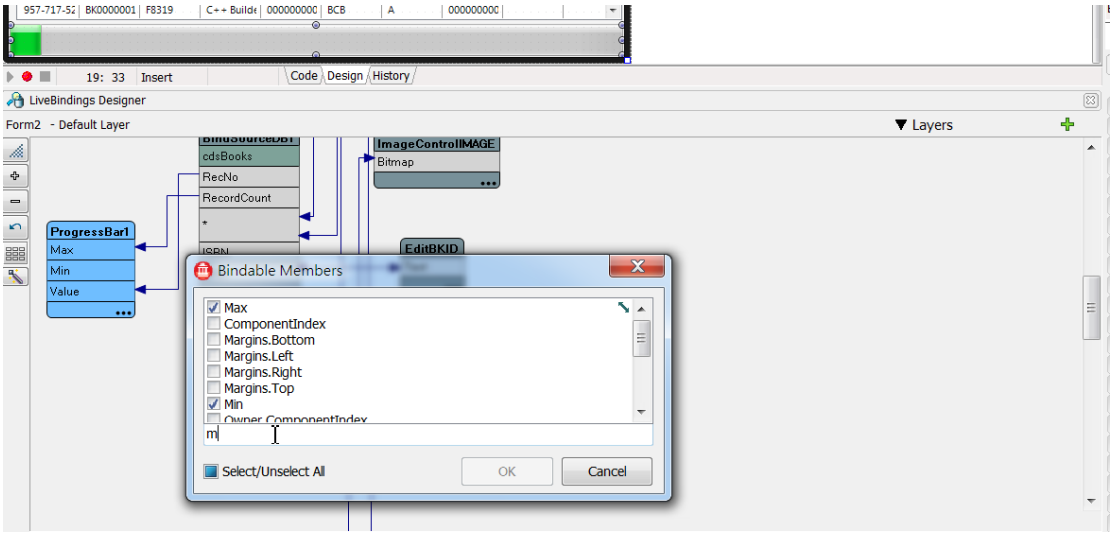
8-1-1 浅尝系结表达式

例如现在让我们在主窗体中加入一个 TProgressBar 在主窗体的下方，我们希望这个 TProgressBar 能够显示目前 cdsBOOKS 中数据的相对位置，因此我们需要进行下列的 2 个系结工作:

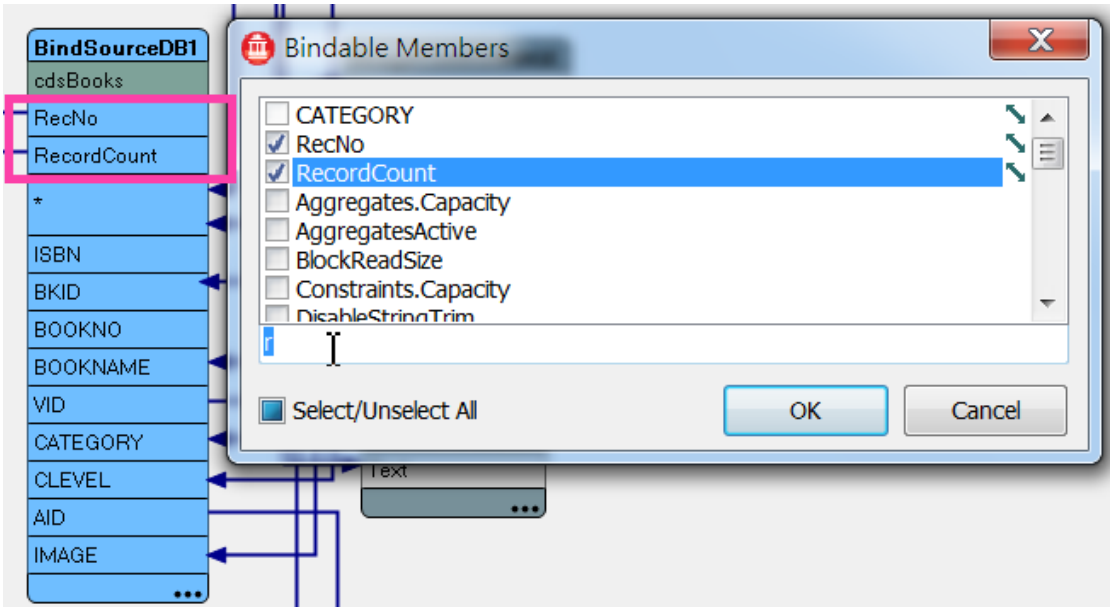
1. 把 cdsBooks 的 RecordCount 特性值系结到 TProgressBar 的 Max 特性值

2. 把 cdsBooks 的 RecNo 特性值系结到 TProgressBar 的 Value 特性值

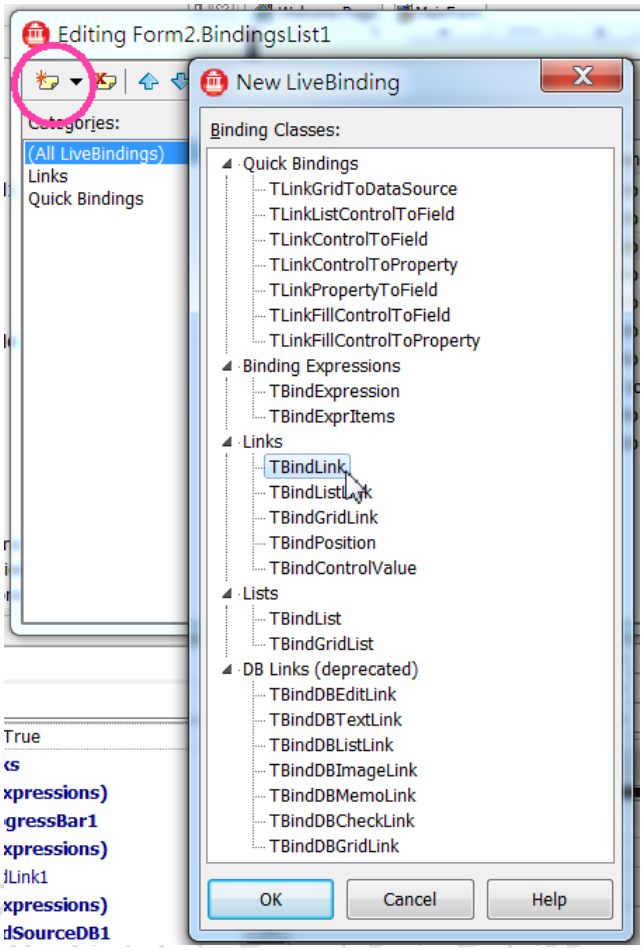
请先在主窗体中加入一个 TProgressBar 组件，设定它的 Align 特性值为 alBottom，回到可视化实时数据系结设计家点选其中 TProgressBar 实体右下方的 3 个黑点开启 TProgressBar 可供系结的成员对话盒，勾选其中的 Max 和 Value 特性值，再点选成员对话盒的 OK 按钮之后 Max 和 Value 特性值就会出现在 TProgressBar 实体中，如下所示:



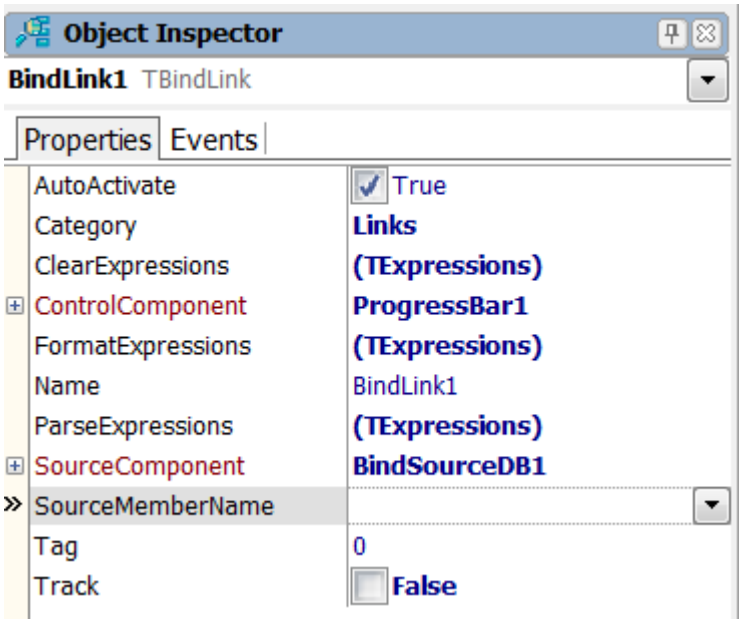
接着使用同样的方法为 cdsBooks 实体加入 RecNo 和 RecordCount 特性如下所示:



双击主窗体中的 TBindingsList 组件，再点选左上方的 New Binding 按钮以加入新的系结对象并且选择加入 2 个 TBindLink 对象如下所示:

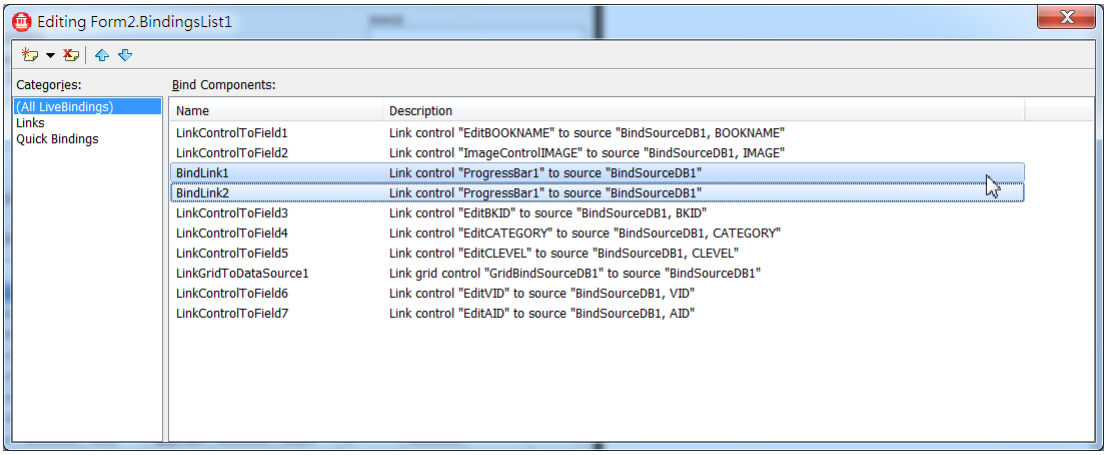


先点选 BindingList 编辑盒中的 2 个 TBindLink 对象，于对象查看器中设定 2 个 TBindLink 对象的 SourceComponent 特性值为 BindSourceDB1，设定 ControlComponent 特性值为 TProgressBar1，如下所示:

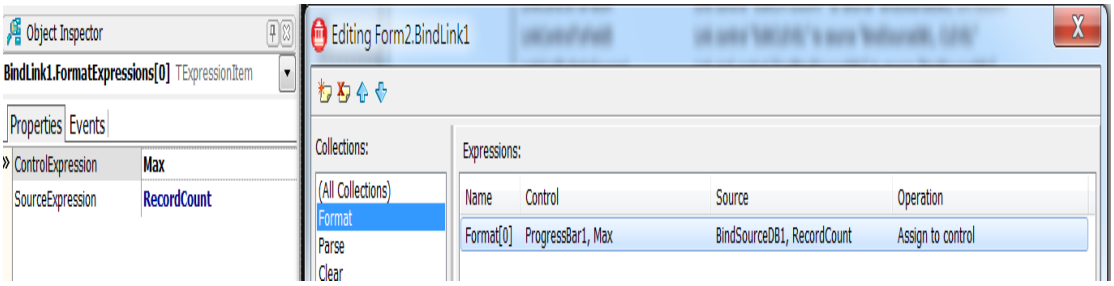


这代表要系结 BindSourceDB1(即 cdsBooks)到 TProgressBar 物件。

接着双击 BindingList 编辑盒中的第 1 个 TBindLink 对象 BindLink1 如下所示:

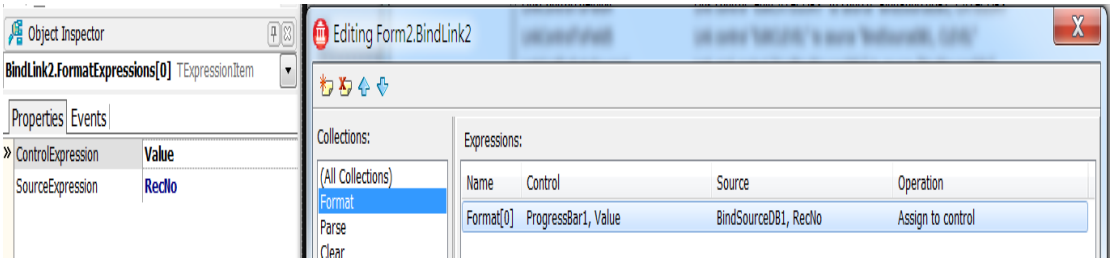


再于对象查看器中设定第 1 个 TBindLink 对象的 SourceExpression 特性值为 RecordCount，设定 ControlExpression 特性值为 Max:



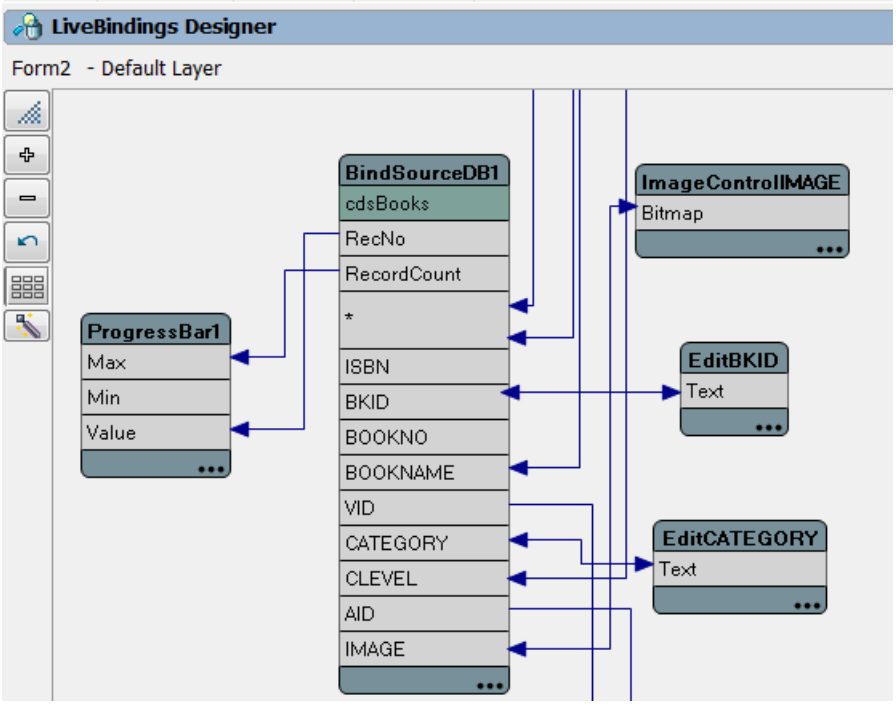
Source	Control
BindSourceDB1, RecordCount	ProgressBar1, Max

于对象查看器中设定第 2 个 TBindLink 对象的 SourceExpression 特性值为 RecNo，设定 ControlExpression 特性值为 Value:

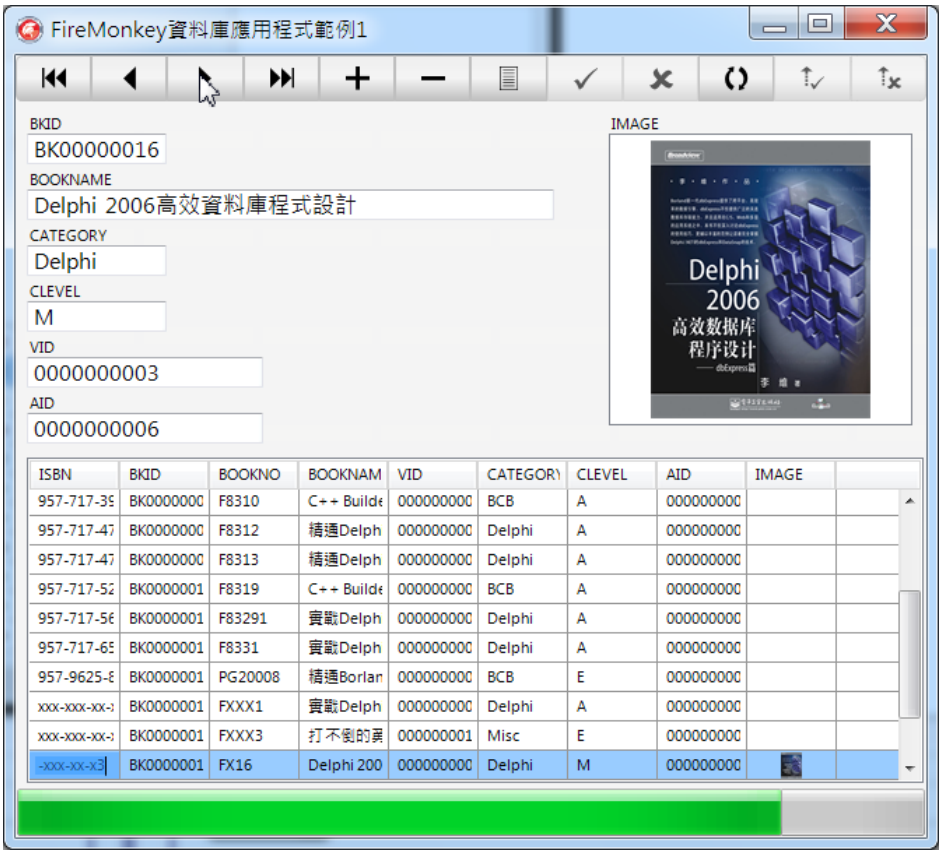


组件	特性值
BindSourceDB1, RecNo	ProgressBar1, Value

设定完成之后可视化实时数据系结设计家就会显示 cdsBooks 和 TProgressBar 之间有如下的系结关系:



最后编译并执行范例 FireMonkey 应用程序，从下图我们可以看到当我们使用 Navigator 在数据中浏览时，TProgressBar 能够正确的显示当前记录的相对位置了:



从这个范例我们可以证实开发人员在可视化实时数据系结设计家中进行的系结会自动产生系结表达式并且由实时数据系结引擎负责执行，当然开发人员也可以直接在 TBindingsList 组件中建立各种不同型态的系结对象并且直接使用系结，在稍后的章节中本书会进一步的讨论系结对象。

7-2 使用 TBindSourceDBX 组件

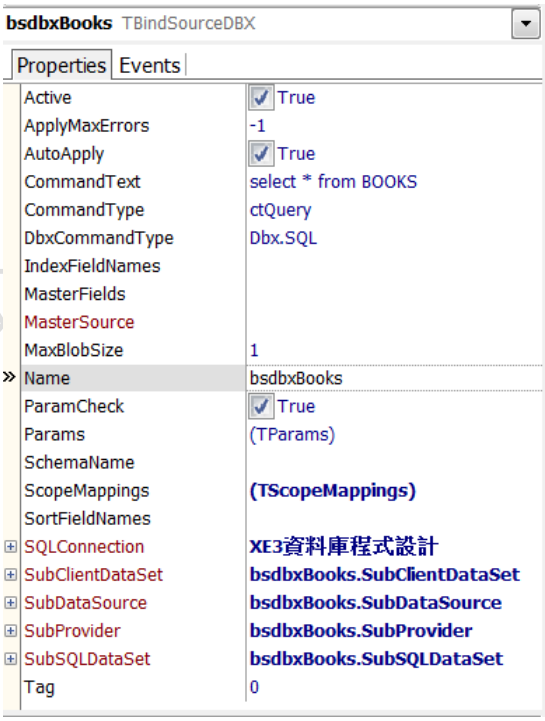
DX10 的实时数据系结也增加了数个新的组件能够帮助开发人员更方便的使用实时数据系结技术来开发和数据库/数据源相关的应用程序。本小节将说明如何使用 TBindSourceDBX 组件。

TBindSourceDBX 组件结合了数个组件于一身让开发人员只需要使用这一个组件就可以链接数据库并且绑定控件。简单的说 TBindSourceDBX 内部包含了 TSQLDataSet，TDataSetProvider，TClientDataSet 和 TDataSource 等组件，因此开发人员只需要链接 TBindSourceDBX 到 TSQLConnection 组件即可开始进行系结的开发工作，可节省许多重复使用 TDataSetProvider，TClientDataSet 和 TDataSource 组件的时间。在本小节中将以一个简单的范例说明如何使用 TBindSourceDBX 来进行 Master/Detail 的开发。

首先在 Delphi 整合发展环境中建立一个 FireMonkey Desktop Application 项目，拖曳 Data Explorer 中的『XE3 数据库程序设计』节点到主窗体中以建立一个 TSQLConnection 组件，再放入一个 TBindSourceDBX 组件，设定它的特性值如下：

特性	特性值
Name	bsdbxBooks
SQLConnection	XE3数据库程序设计
CommandText	select * from BOOKS

请注意下面的对象查看器，在其中我们可以看到 TBindSourceDBX 组件包含了 SubClientDataSet，SubDataSource，SubProvider 和 SubSQLDataSet 四个特性，这四个特性就是 TBindSourceDBX 组件中包含的 TSQLDataSet，TDataSetProvider，TClientDataSet 和 TDataSource 对象。

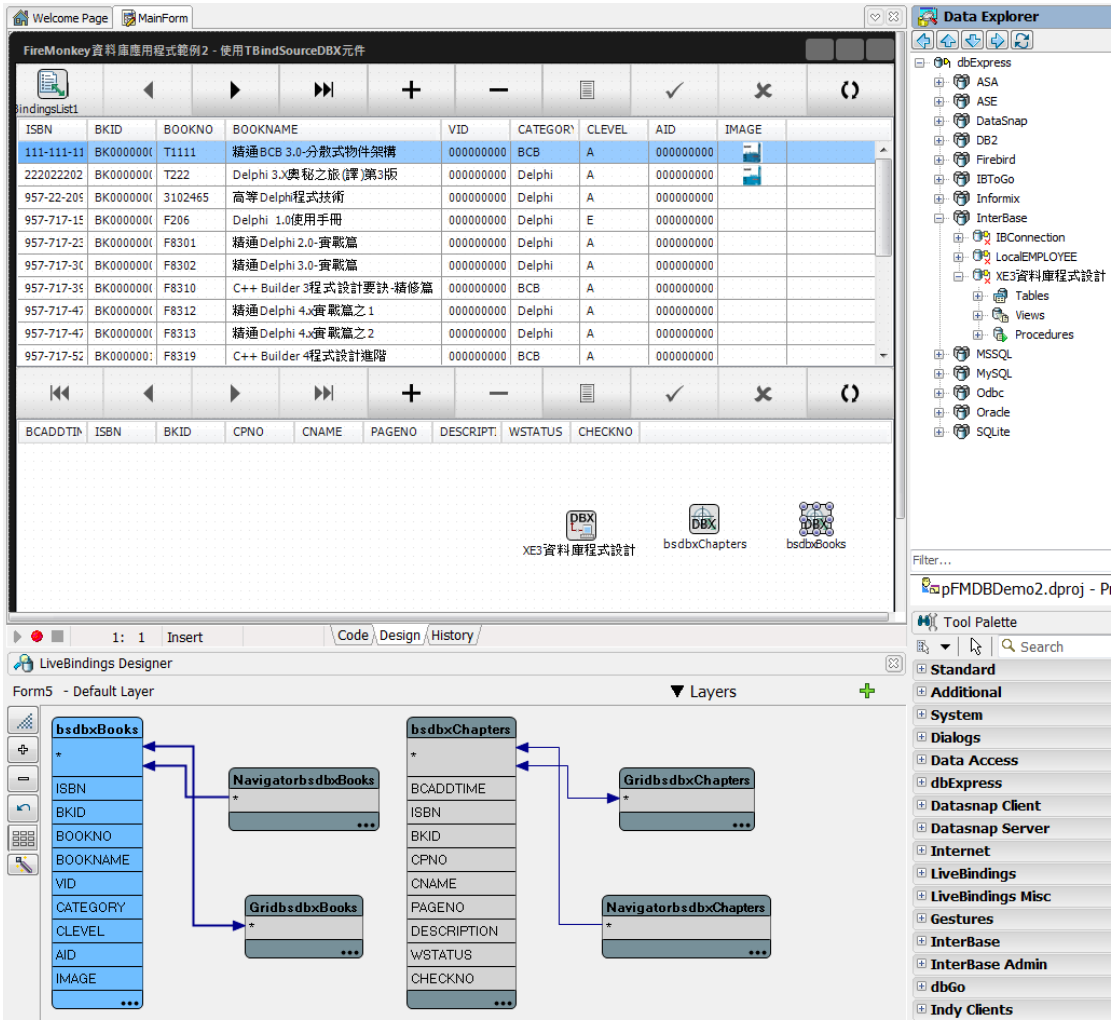


现在再放入另外一个 TBindSourceDBX 组件，设定它的特性值如下：

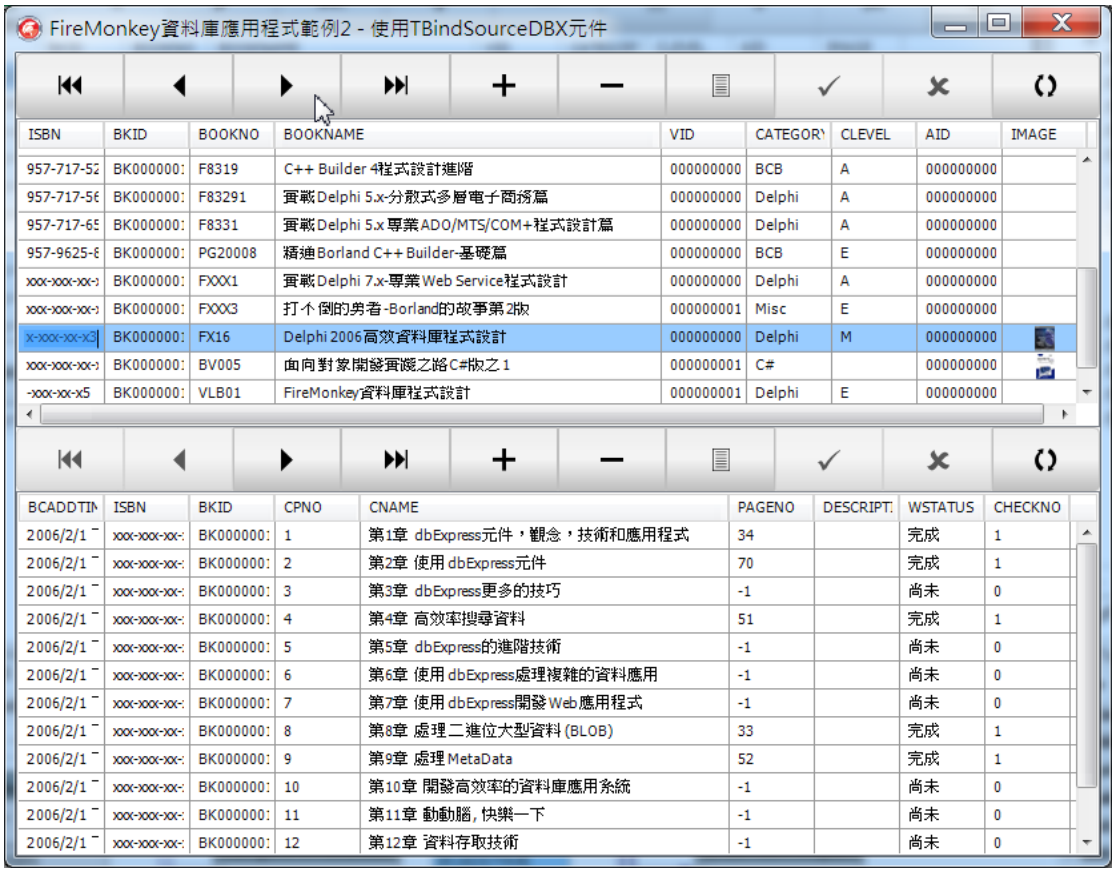
特性	特性值
Name	bsdbxChapters
SQLConnection	XE3数据库程序设计
CommandText	select * from BOOKCHAPTERS
MasterSource	bsdbxBooks
MasterFields	BKID
IndexFieldNames	BKID

由于 BOOKS 和 BOOKCHAPTERS 这 2 个数据表之间是以 BKID 这个键值字段关连在一起，因只要我们设定 bsdbxChapters 的 MasterSource，MasterFields 和 IndexFieldNames 这 3 个特性就可以在 bsdbxBooks 和 bsdbxChapters 之间建立 Master/Detail 的关系。

最后启动可视化实时数据系结设计家，分别系结 TGrid 和 TBindNavigator 到两个 TBindSourceDBX 组件，最后在整合发展环境中的设计结果如下所示：



现在读者就可以编译和执行这个范例 FireMonkey 应用程序了，从下图可以看到使用 TBindSourceDBX 组件果然可以轻易的在 FireMonkey 应用程序中建立数据源之间的 Master/Detail 关系。

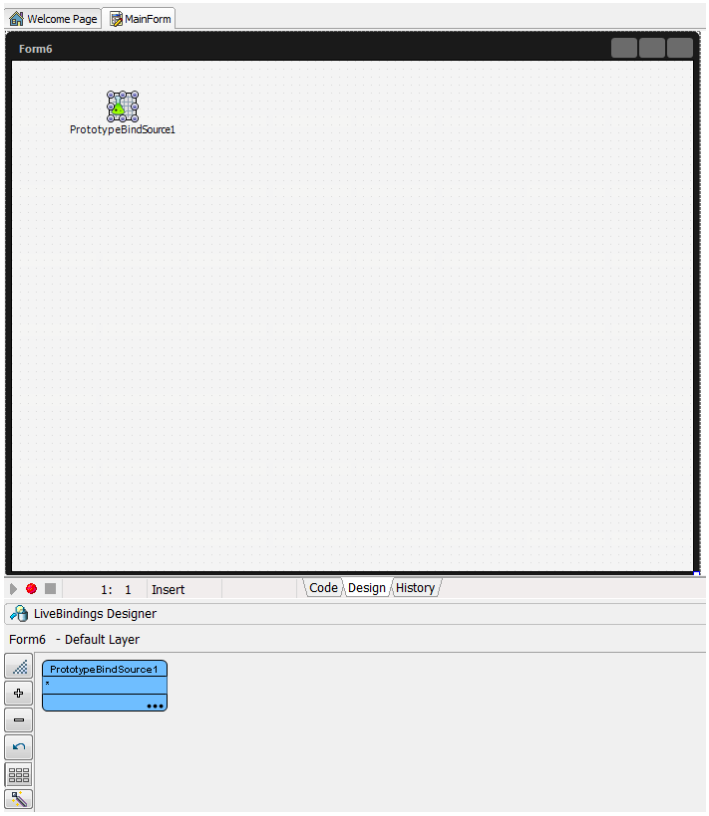


7-3 使用 TPrototypeBindSource 组件

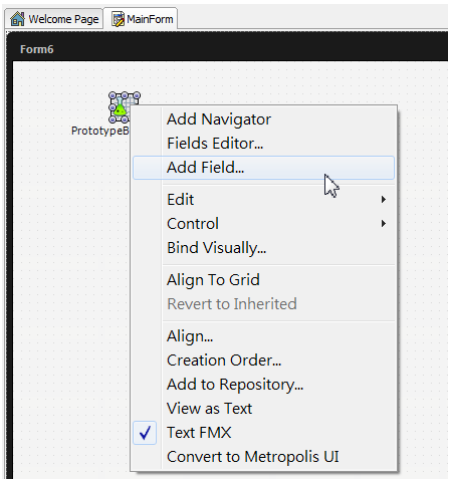
TPrototypeBindSource 也是 DX10 才出现的新组件，它的主要目的是快速为开发人员建立一个类似虚拟的数据表，其中可包含开发人员需要的任何种类的数据型态的字段，并且在这个虚拟数据表中产生随机资料，以便让开发人员可以进行 POC(Proof of concept)的开发，或是进行快速的雏形开发工作。

开发人员可以使用 TPrototypeBindSource 组件快速开发 VCL/FireMonkey 的数据相关应用程序，以便向客户展示应用程序的特定功能。由于 TPrototypeBindSource 组件能够自动且快速的产生随机数据，因此也适合使用来进行测试的目的。本小节将说明如何使用 TPrototypeBindSource 组件，以便让读者也能够使用它来快速开发 VCL/FireMonkey 的数据相关应用程序。

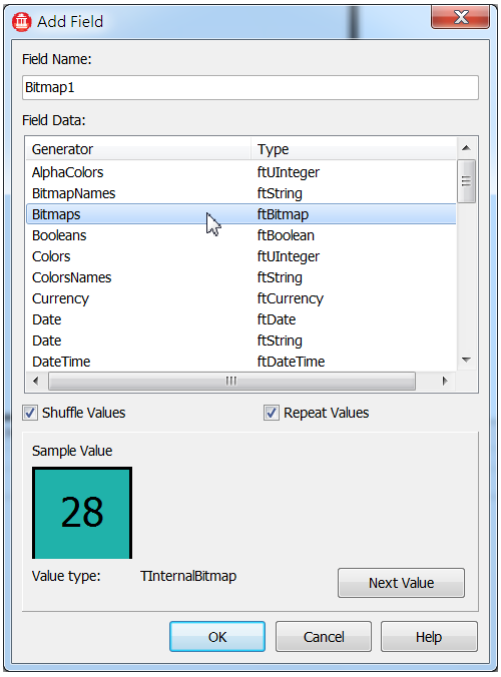
首先在整合发展环境中建立一个 FireMonkey Desktop Application 项目，于主窗体中加入一个 TPrototypeBindSource 组件，在可视化实时数据系结设计家中就会出现 TPrototypeBindSource 实体，如下所示:



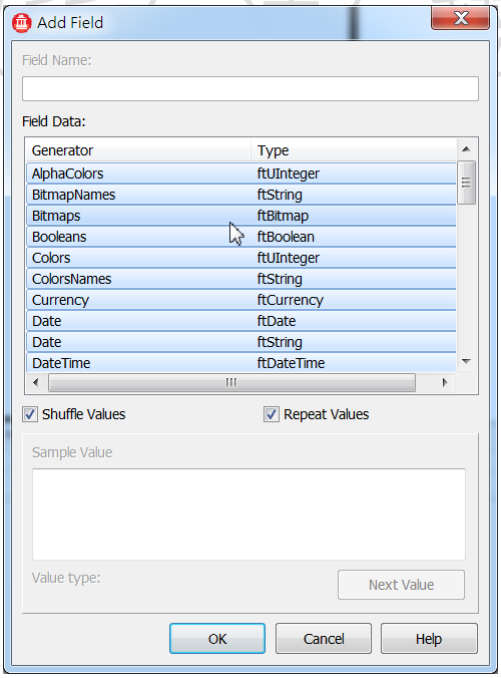
使用鼠标右击 TPrototypeBindSource 组件并且从快捷菜单中选择『Add Field…』选项开始在 TPrototypeBindSource 组件中加入字段，如下所示：



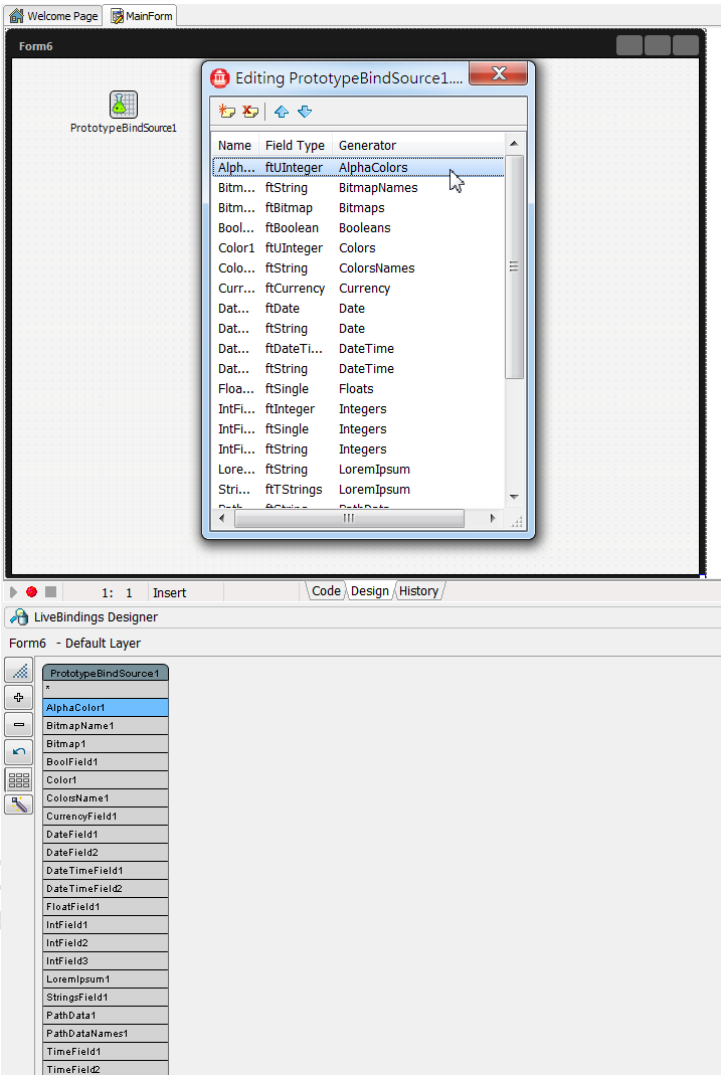
此时便会出现如下的 Add Field 对话盒，其中包含了内定的各种数据型态的字段，读者可以选择要加入到 TPrototypeBindSource 组件中的字段，当读者点选其中的字段时可以在对话盒的下方中看到选择的数据字段会产生的随机范例数据：



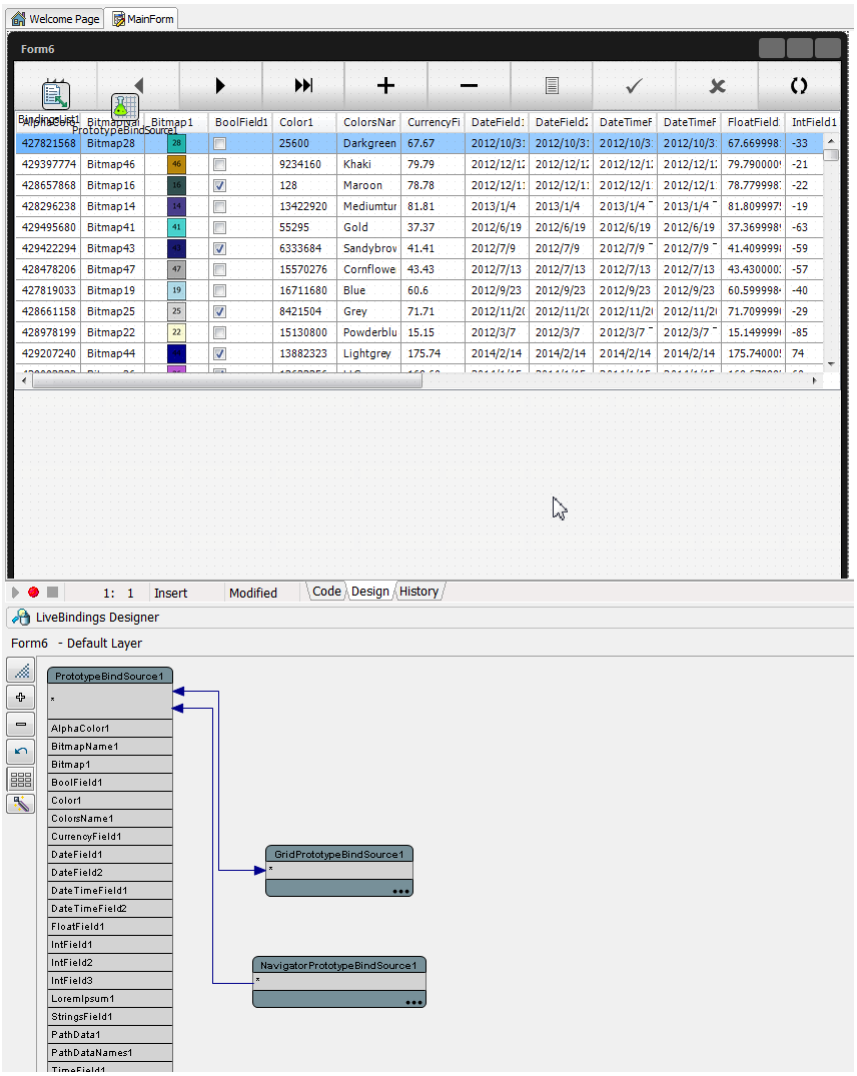
我们可以在按下 Shift 键时使用鼠标点选多个字段以选择加入 TPrototypeBindSource 组件中，如下所示，在选择要加入的字段之后点选 OK 按钮关闭对话盒:



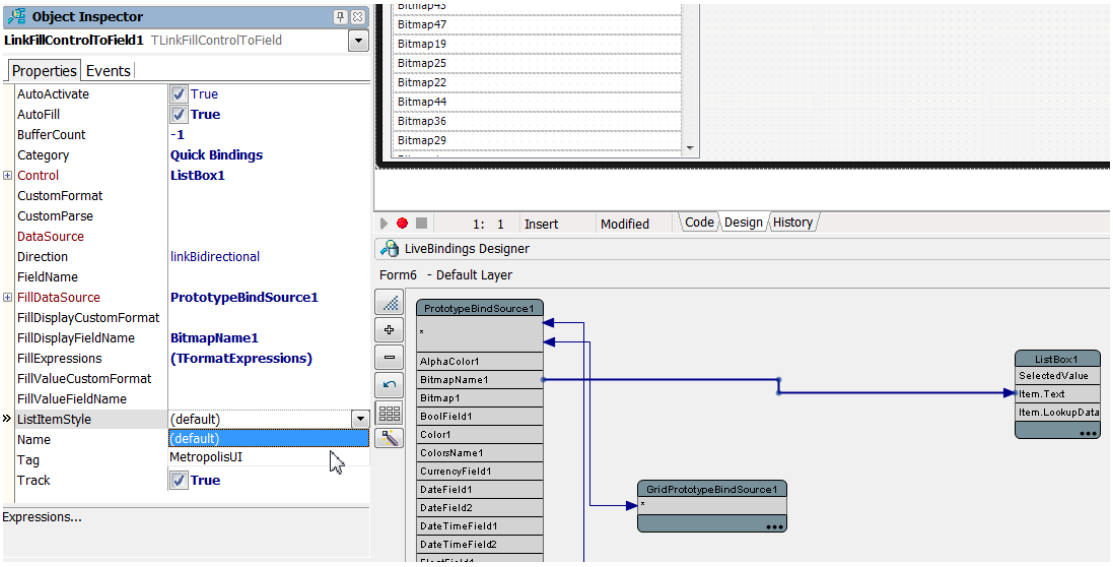
接着再次使用鼠标右击 TPrototypeBindSource 组件并且从快捷菜单中选择『Fields Editor….』选项，就可以在字段编辑器中看到刚才加入的所有字段，如下所示，而且可视化实时数据系结设计家在 TPrototypeBindSource 实体中也会显示刚加入的字段对象:



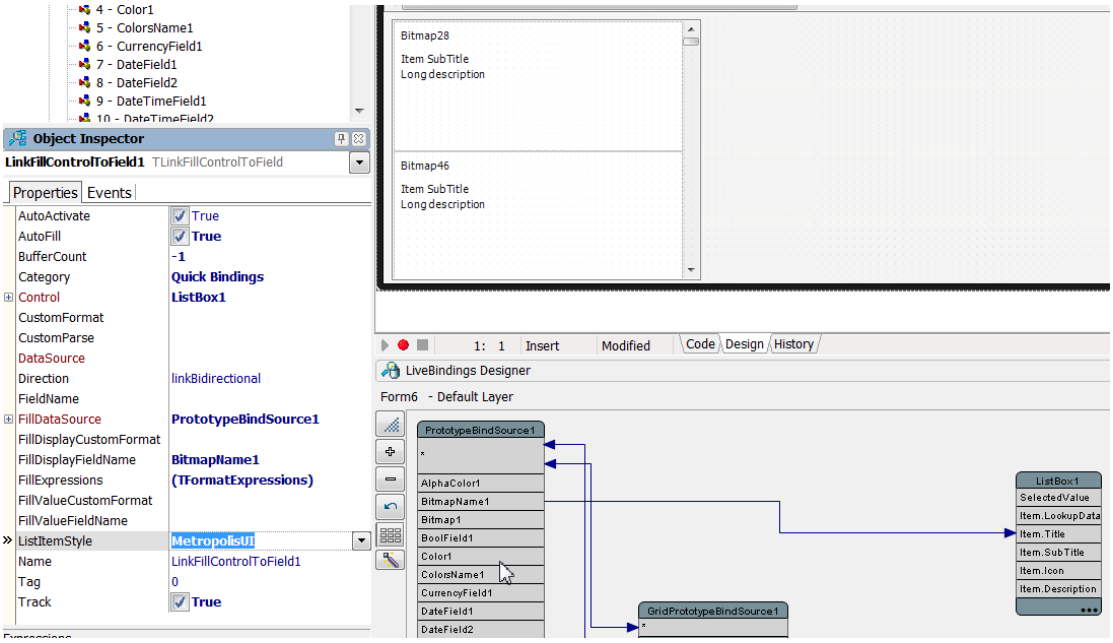
现在请在可视化实时数据系结设计家中链接 TPrototypeBindSource 实体和 TGrid 组件，再系结到 TBindNavigator，在下图中读者就可以看到 TPrototypeBindSource 组件中包含的字段以及随机产生的数据都出现在主窗体的 TGrid 组件中了：



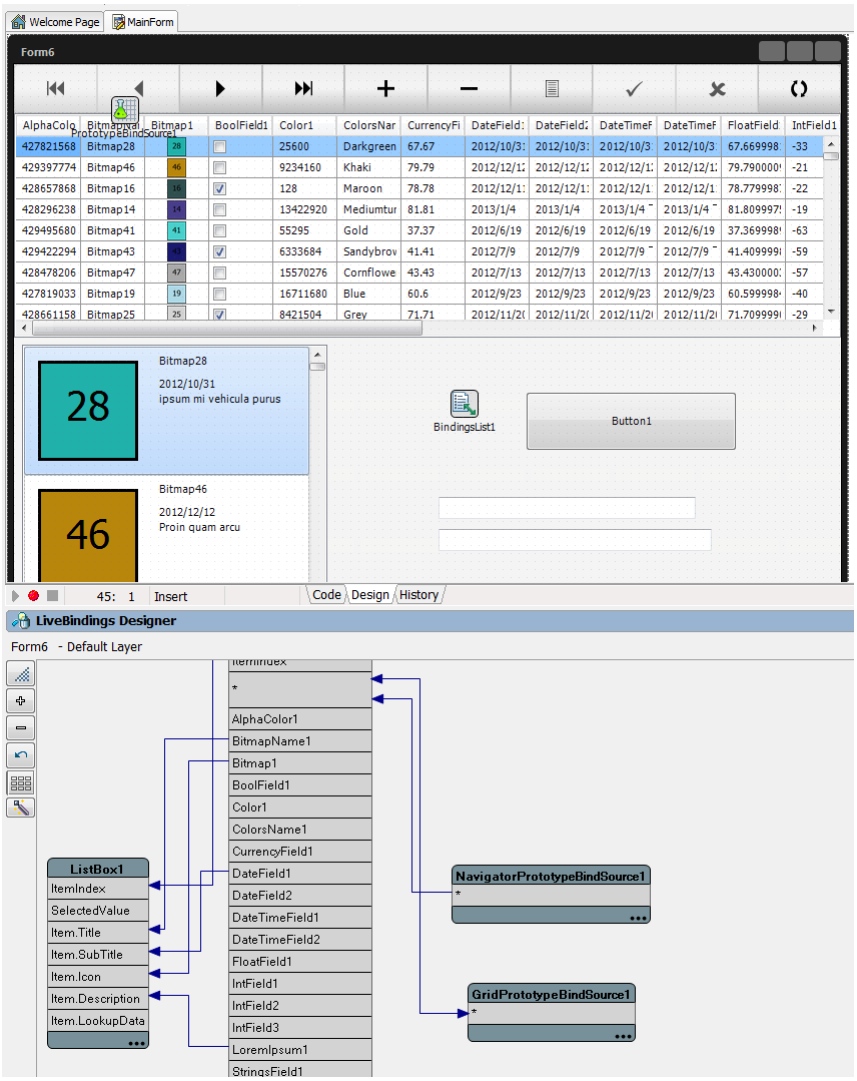
现在再于主窗体中放入一个 TListBox 组件，在可视化实时数据系结设计家中点选 TPrototypeBindSource 实体的 BitmapName1 字段并拖曳鼠标到 TListBox 组件的 Text 特性以系结这 2 个对象，此时 TListBox 组件中就会显示 TPrototypeBindSource 组件中 BitmapName1 字段的所有值，接着在可视化实时数据系结设计家中点选系结 TPrototypeBindSource 实体 BitmapName1 字段和 TListBox 组件 Text 特性之间的链接线，再于对象查看器中点选它的 ListItemStyle 特性，选择 MetropolisUI，如下图所示:



在选择了 MetropolisUI 特性值之后主窗体中的 TListBox 组件的内容就会变化成使用 MetropolisUI 的格式，如下图所示:

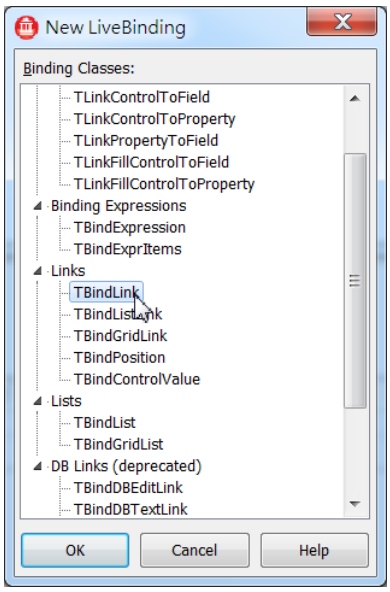


在可视化实时数据系结设计家中系结 BitmapName1 字段和 ListBox1 实体的 Item.Title 特性，在系结设计家中系结 Bitmap1 字段和 ListBox1 实体的 Item.Icon 特性，在系结设计家中系结 DateField1 字段和 ListBox1 实体的 Item.Subtile 特性，最后这些数据就都会出现在主窗体的 TListBox 中，如下图所示:

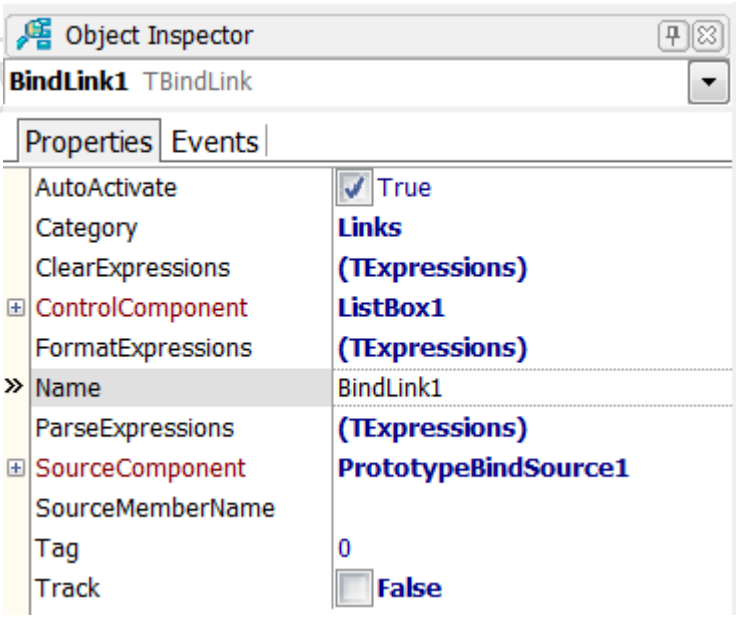


在离开本小节之前让我们再试着使用另外一个系结对象来系结 TPrototypeBindSource 和 TListBox，让使用者在浏览 TPrototypeBindSource 中的数据时也能够同步 TListBox 中的数据。

请双击主窗体中的 TBindingsList 对象，点选左上方的 New Binding 按钮以增加一个新的系结对象，在 New LiveBinding 对话盒中选择建立 TBindLink 对象，如下图所示：

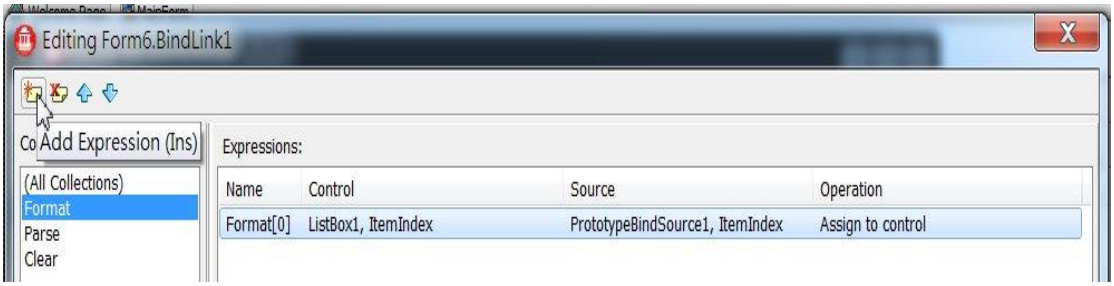


接着在 TBindingsList 对话盒中点选此新建立的 TBindLink 对象，再于对象查看器中设定它的 SourceComponent 特性值为 PrototypeBindSource1，设定它的 ControlComponent 特性值为 ListBox1，如下图所示:

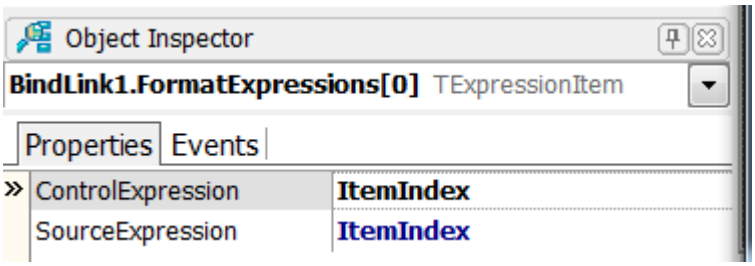


一旦做了如上的设定就代表使用 TBindLink 系结对象来系结 TPrototypeBindSource 和 TListBox 对象，接下来我们需要再设定系结表达式来定义系结 TPrototypeBindSource 和 TListBox 对象之中的什么特性。

现在请双击 TBindingsList 对话盒中的 TBindLink 对象，此时 TBindLink 对象的系结表达式编辑器就会出现，如下所示:



接着在对象查看器中设定 SourceExpression 为 ItemIndex，也设定 ControlExpression 为 ItemIndex，如下所示：

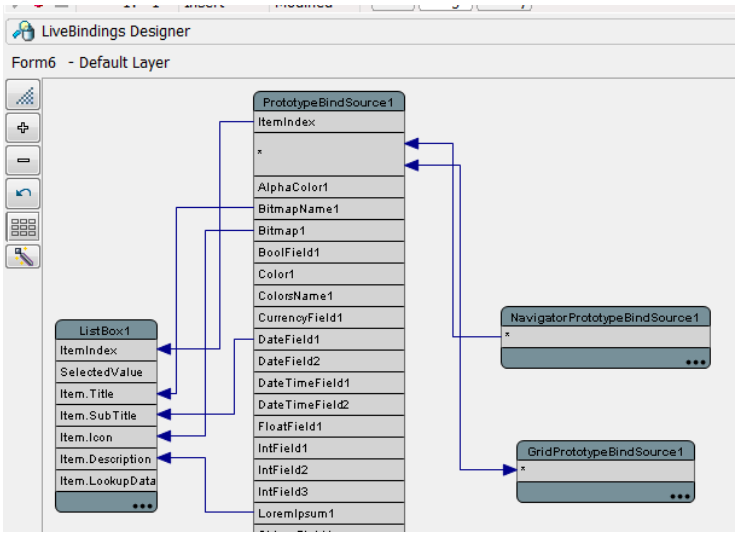


这个系结表达式就代表在执行时期会进行如下的系结：

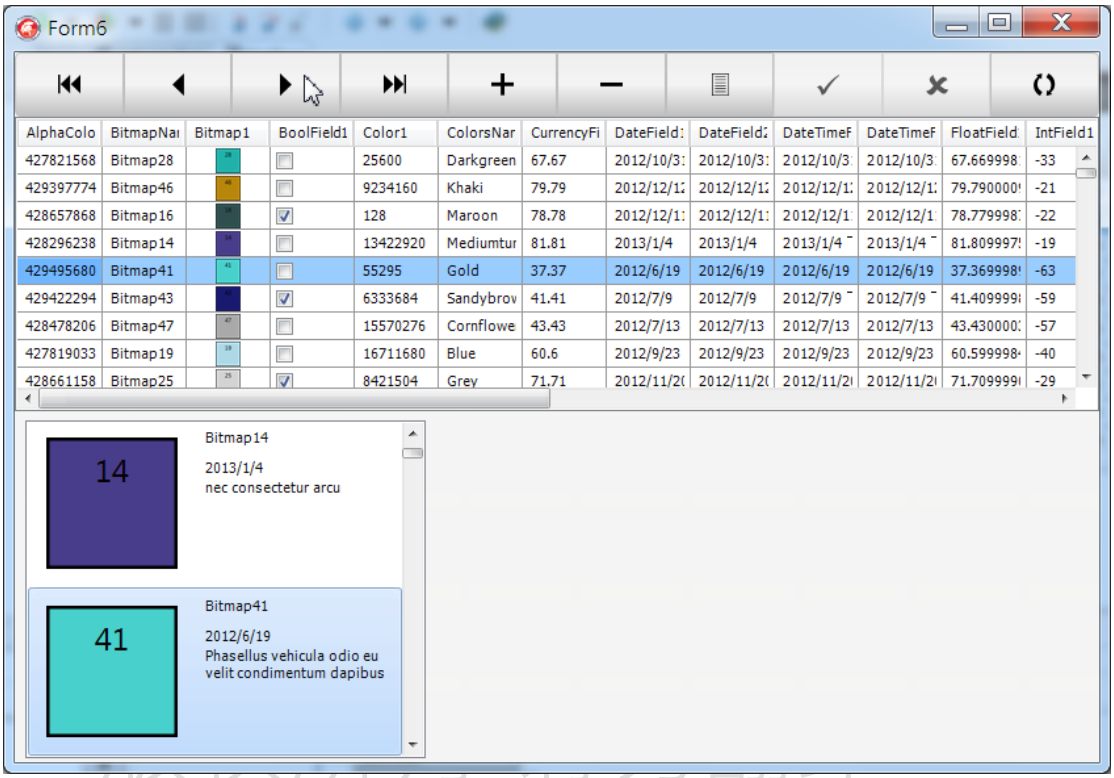
```
ListBox1.ItemIndex := PrototypeBindSource1.ItemIndex;
```


也就是说当我们浏览 TPrototypeBindSource 中的数据时，也会相对应的浏览到 TListBox 中包含的项目。

现在回到可视化实时数据系结设计家就可以看到 PrototypeBindSource1 实体和 ListBox1 实体之间的 ItemIndex 有了系结的关系了，如下所示：



请编译并且执行范例应用程序，就可以看到如下的执行结果，当我们使用 TBindNavigator 浏览 PrototypeBindSource1 的数据时，ListBox1 也浏览到相对应的数据了。



使用可视化实时数据系结设计家进行系结工作虽然非常的简单，但实时数据系结仍然有一些深入的技术，我们将在下一章中进行说明。

7-4 结论

本章说明了如何使用 DX10 的可视化实时数据系结技术来系结数据源和控件，藉由可视化实时数据系结设计家开发人员可以轻易的系结 FireMonkey 控件和数据源，如此一来开发人员就可以轻易的开发出数据库相关的 FireMonkey 应用程序了。

在本章讨论的过程中也简单的说明了如何使用系结对象并且使用简单的系结表达式来绑定控件，让读者了解除了可使用可视化实时数据系结设计家进行系结工作之外，也可以直接使用系结对象和系结表达式来进行其他的系结工作，在稍后的章节中会更详细的讨论系结类别和系结技术。

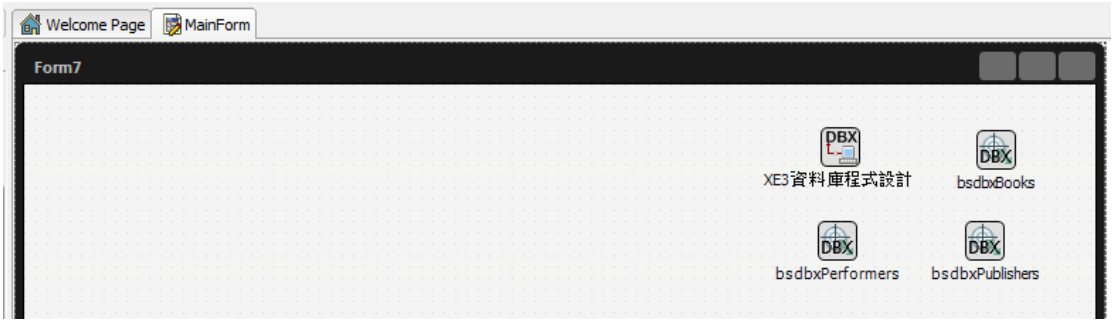
第8章 更多的实时数据系结技术

在开发数据库应用程序时经常需要根据一个字段来查询另外一个查询值，例如当使用者输入身分证字号时应用程序需要根据身分证字号来查询并且显示此身分证字号的人名，这种应用在实时数据系结技术称为 Lookup 的关系，在下面的小节中将详细的讨论如何使用这个功能。

8-1 使用实时数据系结技术的 Lookup 功能

本书将使用 InterBase 做为范例数据库，在其中有数个数据表之间拥有 Master/Detail 的关系，在说明如何使用可视化实时数据系结技术逐渐开发 FireMonkey 数据库应用程序的流程中，将一一的带入使用这些数据表。不过让我们先从简单的开发工作开始说明如何使用可视化实时数据系结。

我们还是以第 1 章中的『XE3 数据库程序设计』为范例数据库，首先在 IDE 中建立一个 FireMonkey Desktop Application 项目，拖曳 Data Explorer 的『XE3 数据库程序设计』节点到主窗体中建立 TSQLConnection 组件，再放入 3 个 TBindSourceDBX 组件链接到 BOOKS，Publishers 和 Performers 这 3 个数据表，如下所示:



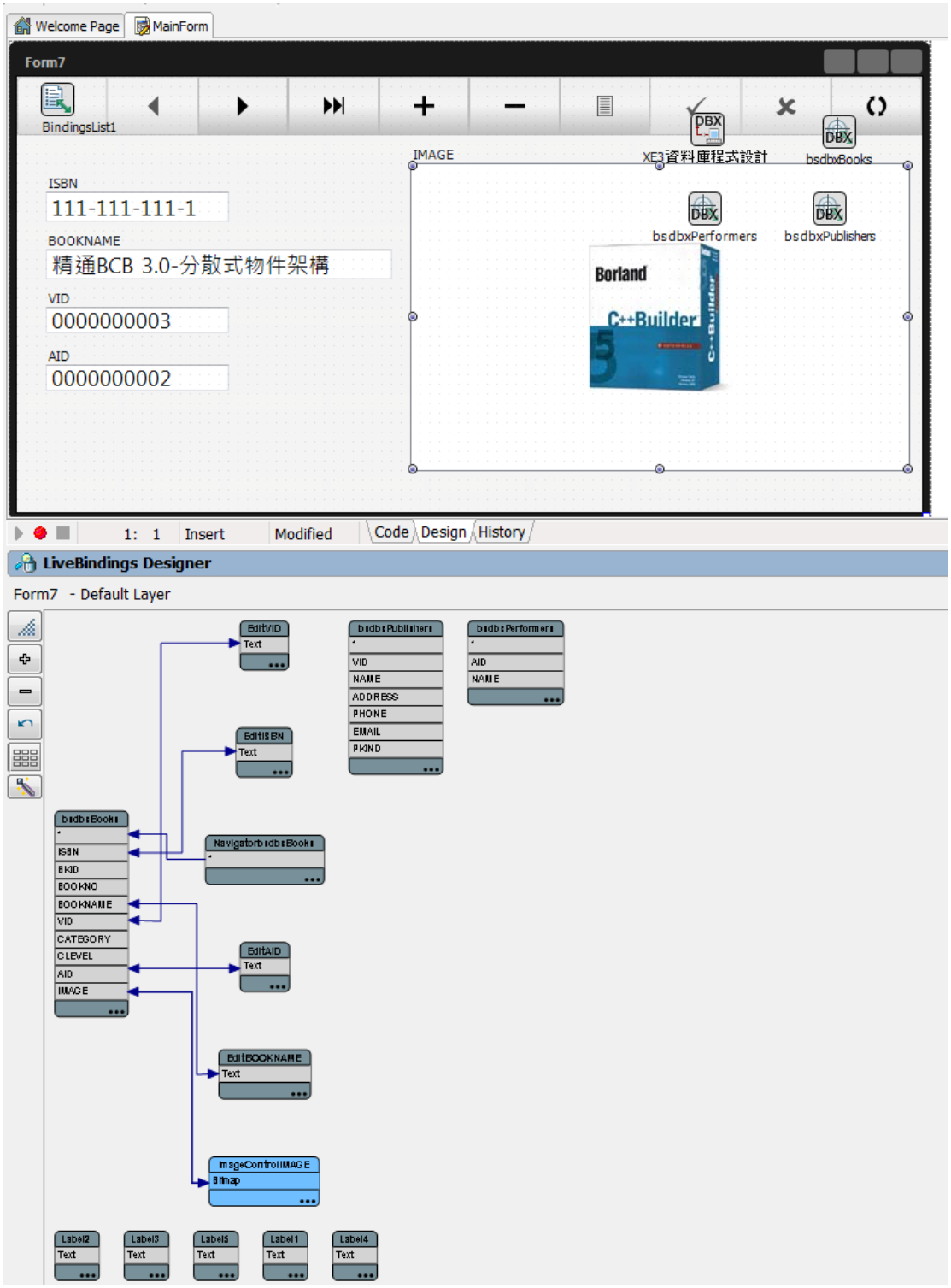
设定主窗体中 3 个 TBindSourceDBX 组件的特性值如下:

组件	特性值
Name	bsdbxBooks
SQLConnection	XE7数据库程序设计
CommandType	ctQuery
CommandText	select * from BOOKS

组件	特性值
Name	bsdbxPublishers
SQLConnection	XE7数据库程序设计
CommandType	ctQuery
CommandText	select * from PUBLISHERS

组件	特性值
Name	bsdbxPerformers
SQLConnection	XE7数据库程序设计
CommandType	ctQuery
CommandText	select * from PERFORMERS

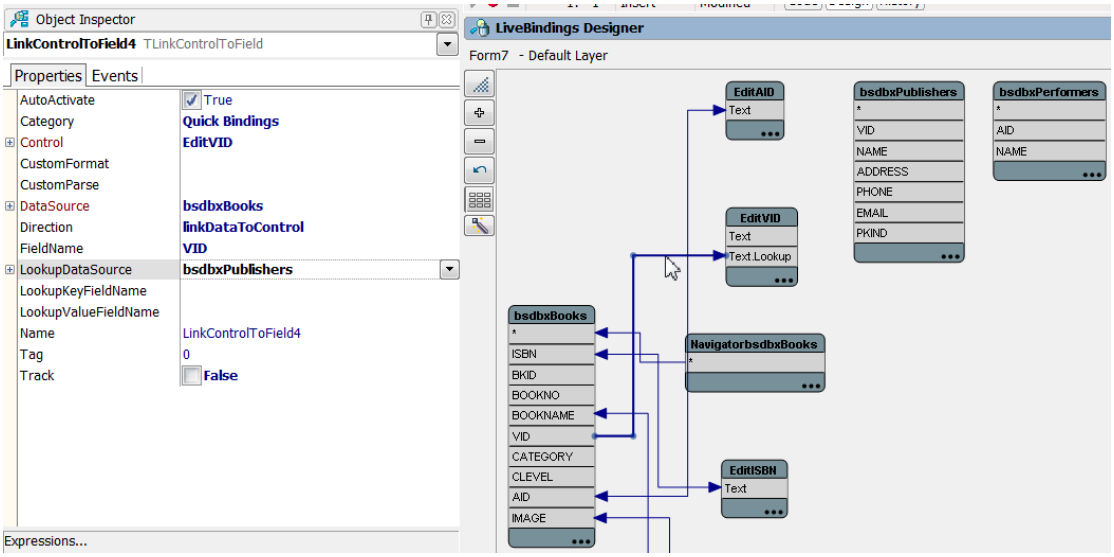
同时设定这 3 个 TBindSourceDBX 组件的 Active 特性值为 True 以开启数据表:



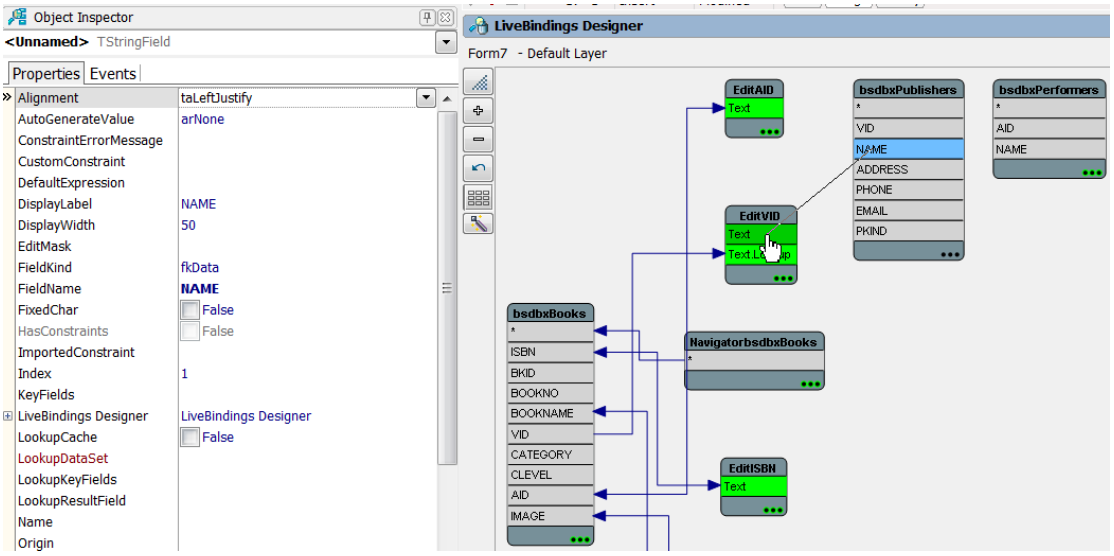
请注意主窗体中的 VID 和 AID 这 2 个字段，它们分别是 Publishers 和 Performers 这 2 个数据表的键值，因此我们并不希望显示 VID 和 AID 的域值，而是这 2 个键值代表的数值，因此我们需要使用 VID 到 Publishers 数据表中查询它代表的数值，使用 AID 到 Performers 数据表中查询它代表的数值，这在实时数据系结中就称为 Lookup 的动作。

那么我们要如何使用实时数据系结的 Lookup 功能来根据 VID 和 AID 来查询它们代表的数值呢?

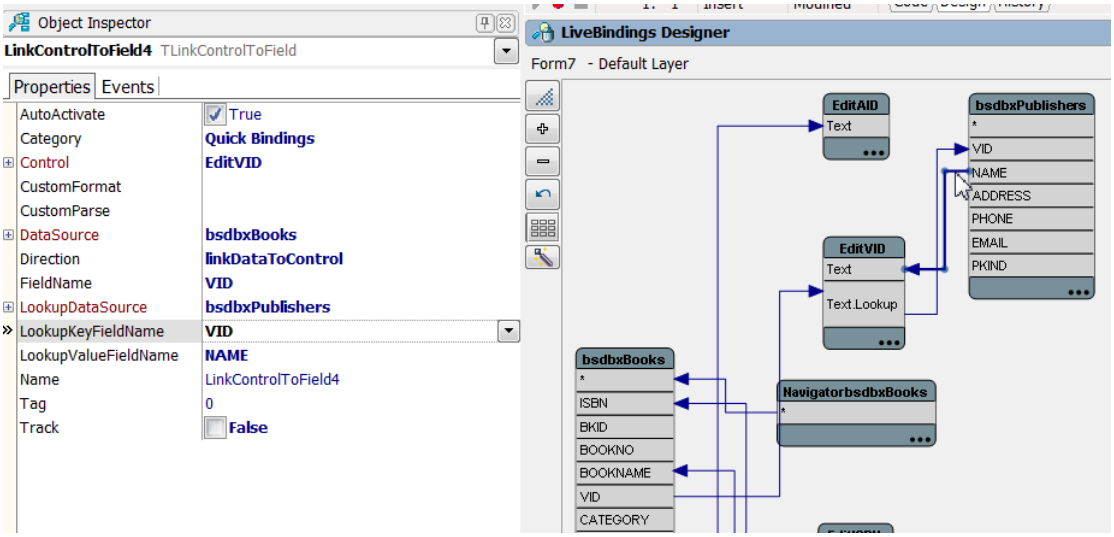
现在请回到可视化实时数据系结设计家，点击链接 bsdbxBooks 实体 VID 字段和 EditVID 实体之间的链接线，接着在对象查看器中的 LookupDataSource 特性中选择 bsdbxPublishers，因为 VID 是 bsdbxPublishers 代表的数据表中的键值，注意一旦您在对象查看器中设定了 LookupDataSource 特性值为 bsdbxPublishers，那么 EditVID 实体立刻会出现一个新的字段『Text.Lookup』，而且从 bsdbxBooks 实体 VID 字段来的链接线会改变到链接到 EditVID 实体的『Text.Lookup』字段，如下所示:



接着请点选鼠标左键，从 bsdbxPublishers 实体的 NAME 字段数据拖曳到 EditVID 实体的 Text 字段，如下所示:



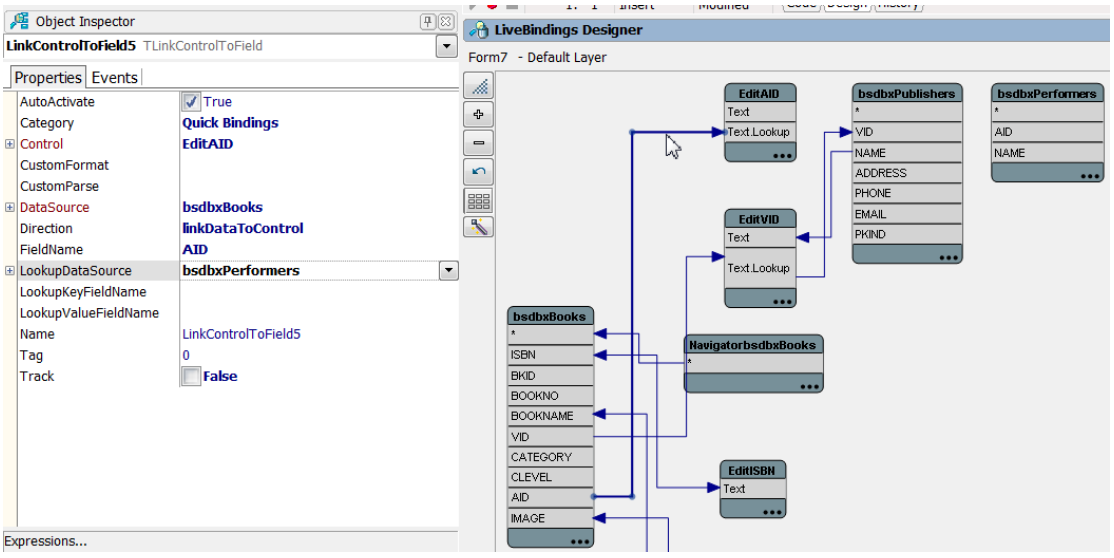
最后请点击链接 bsdbxPublishers 实体的 NAME 字段和 EditVID 实体的 Text 字段之间的链接线，在对象查看器中设定 LookupFieldName 特性值为 VID，如下所示:



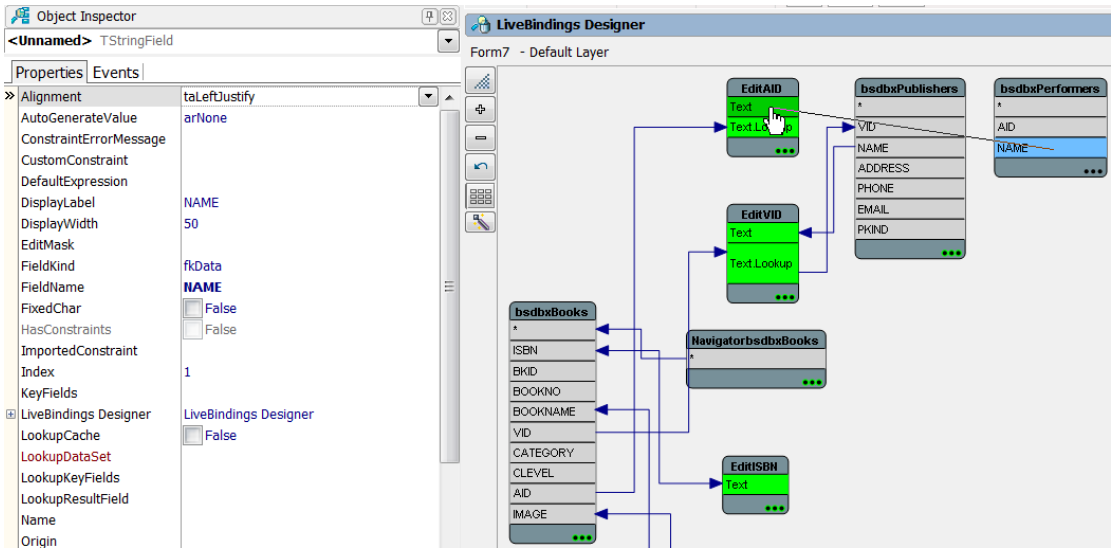
如此一来就完成了使用 bsdbxBooks 中 VID 域值到 bsdbxPublishers 进行 Lookup 查询的工作，如果现在读者检视主窗体中的 EditVID 组件就会发现它的 Text 特性值改变成显示 Publishers 数据表中 VID 域值代表的数值了。

现在让我们使用相同的方法完成 AID 字段的 Lookup 查询工作。

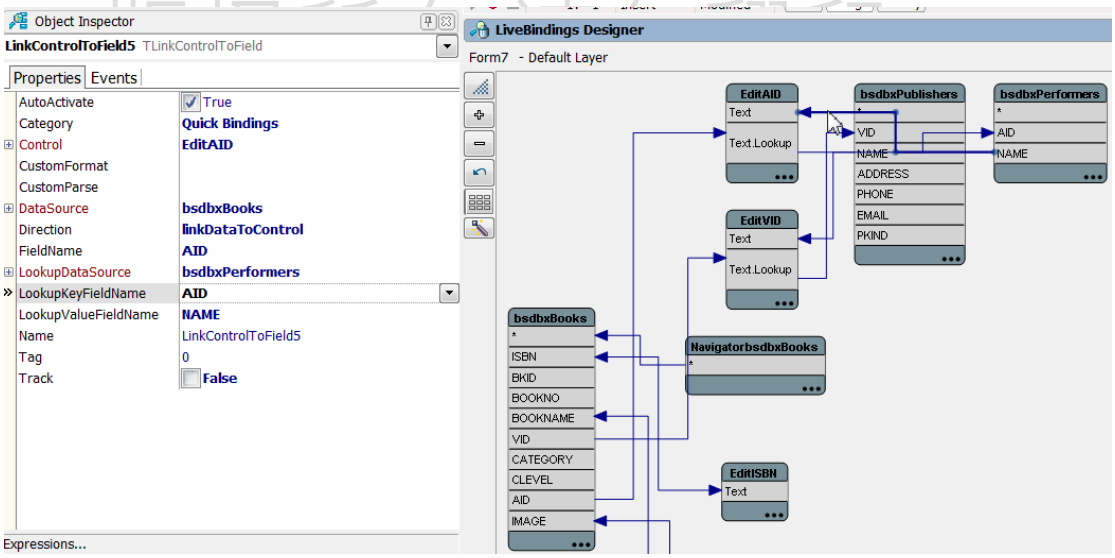
现在请回到可视化实时数据系结设计家，点击链接 bsdbxBooks 实体 AID 字段和 EditAID 实体之间的链接线，接着在对象查看器中的 LookupDataSource 特性中选择 bsdbxPerformers，因为 AID 是 bsdbxPerformers 代表的数据表中的键值，注意一旦您在对象查看器中设定了 LookupDataSource 特性值为 bsdbxPerformers，那么 EditAID 实体立刻会出现一个新的字段『Text.Lookup』，而且从 bsdbxBooks 实体 AID 字段来的链接线会改变到链接到 EditAID 实体的『Text.Lookup』字段，如下所示:



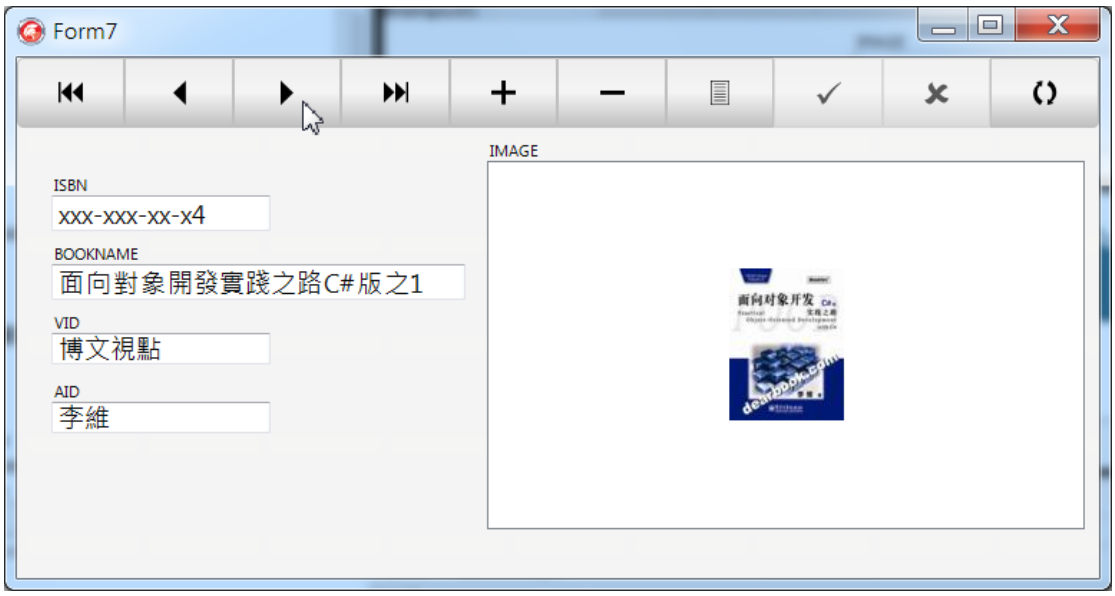
接着请点选鼠标左键，从 bsdbxPerformers 实体的 NAME 字段数据拖曳到 EditAID 实体的 Text 字段，如下所示：



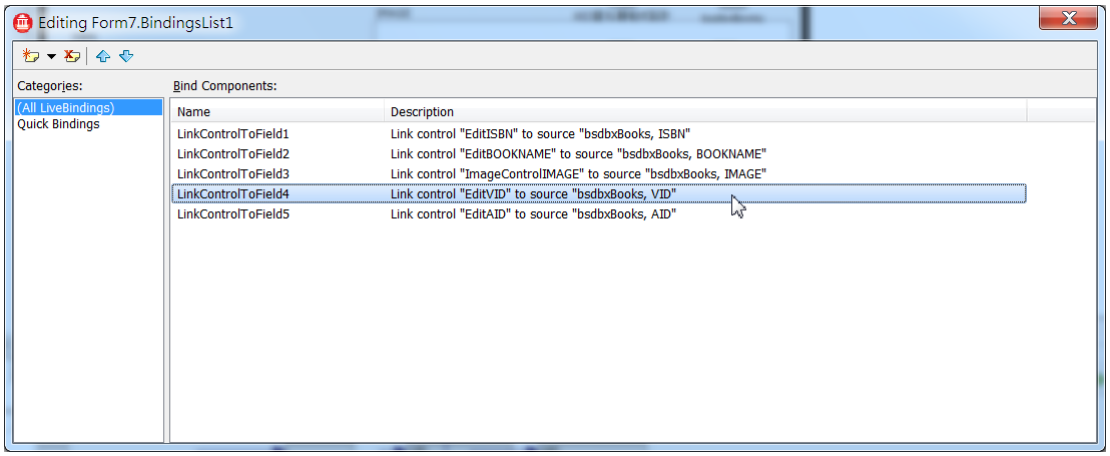
最后请点击链接 bsdbxPerformers 实体的 NAME 字段和 EditAID 实体的 Text 字段之间的链接线，在对象查看器中设定 LookupFieldName 特性值为 AID，如下所示：



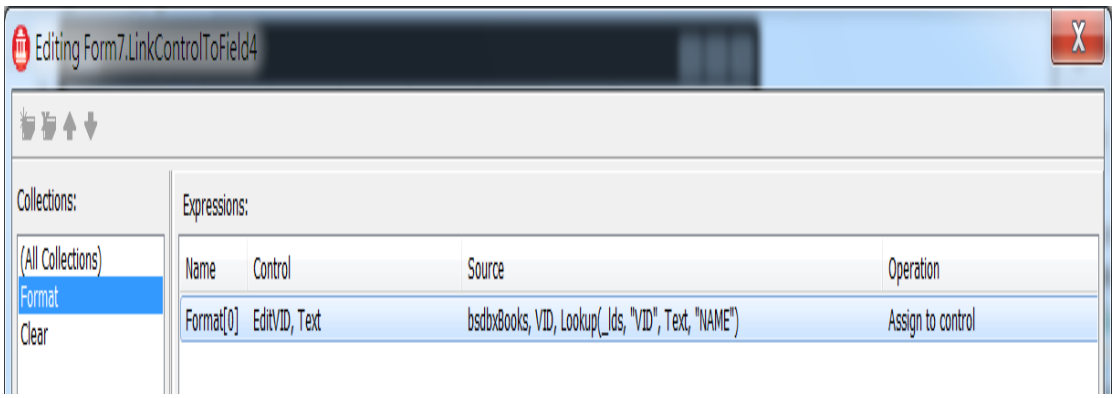
现在请编译并且执行此范例应用程序，在使用 Navigator 浏览数据时，主窗体中的 EditVID 和 EditAID 组件就能够根据 bsdbxBooks 数据表中 VID 和 AID 字段的数值到 bsdbxPublishers 和 bsdbxPerformers 数据表中查询正确的代表数值并且显示在 EditVID 和 EditAID 组件中，如下所示：



在离开本小节之前让我们双击主窗体中的 TBindingsList 组件开启系结表达式编辑器，点选其中代表 VID 执行查询的系结对象，如下所示：



双击此系结对象，我们就可以看到如下的系结表达式：



上图中的系结表达式基本的意思就是:

```
EditVID.Text := bdsdbxBooks.VID.Lookup(_lds, 'VID', Text, 'NAME');
```

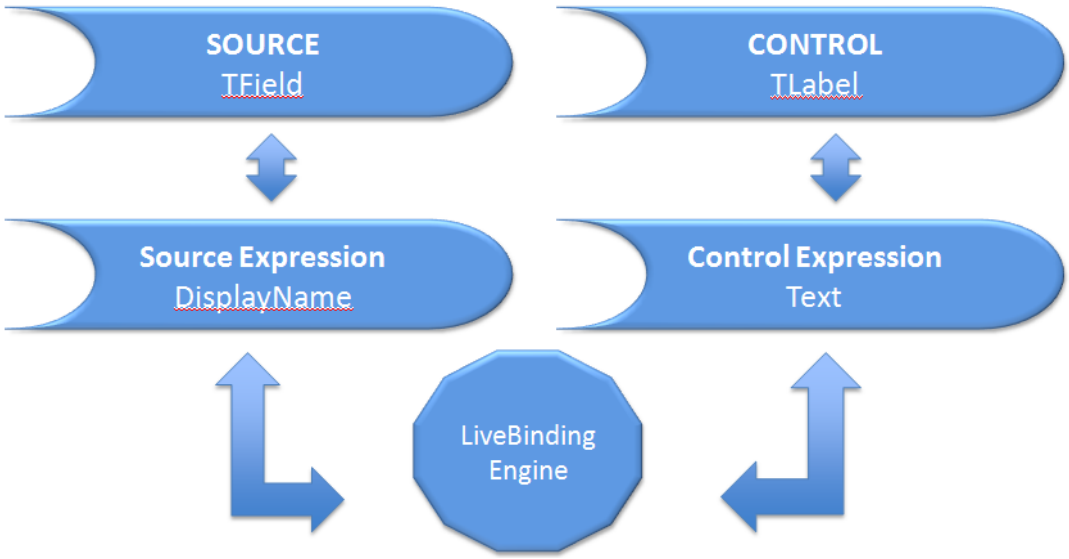
也就是说把 bdsdbxBooks.VID.Lookup(_lds, 'VID', Text, 'NAME')执行的结果指定给 EditVID.Text，但 bdsdbxBooks.VID.Lookup(_lds, 'VID', Text, 'NAME')到底是什么？如果不是由可视化实时数据系结设计家自动产生这个系结表达式，我们似乎根本写不出这个系结表达式，现在连看都不太懂这个系结表达式的意思，当然更别说要写出来了。

因此在我们继续使用实时数据系结技术之前，也许读者需要对于实时数据系结技术有一些基本的观念。

8-2 什么是实时数据系结

实时数据系结技术就是提供应用程序中组件/对象和数据源之间一个可擦写的连结，这个可擦写的连结就是所谓的系结表达式(Binding Expression)，在系结表达式中有个对象，一方称为来源组件(Source Component)，另一方则称为控制组件(Control Component)。

系结表达式有 2 种型态，即单向系结(unidirectional)和双向系结(bidirectional)，如果是单向系结，那么运作的方式就是由来源组件执行系结表达式之后再把结果指定给控制组件。如果是双向系结，那么控制组件也可以执行它的系结表达式再指定回给来源组件，这整个概念可以使用下面的图形来说明:



从上图我们可以了解，系结表达式是由实时数据系结引擎来解释和执行的，在目前 DX10 的实作中实时数据系结引擎可执行下列 3 种的系结表达式:

- 简单的系结表达式(Simple Expressions)
- 未拖管系结表达式(Unmanaged Bindings)
- 拖管系结表达式(Managed Bindings)

系结表达式是由字符串组成的表达式，在这个字符串表达式中开发人员可使用下列的元素:

1. +, -, /, *这 4 个操作数
2. 常数值，例如字符串常数值 'abc'，数值常数值 1， 1.23，布尔常数值 True， False，和 Nil
3. 对象的方法和特性，全局方法和系结引擎提供的内定方法。

系结表达式中最重要的元素就是上述的第 3 项，这需要读者清楚的了解它的函意。

当系结表达式被系结引擎执行时，系结引擎会使用一个所谓的**执行范围**(Scope)来解释系结表达式中包含的元素，让我们使用一些范例来说明这个意思。

例如下面是一个简单的系结表达式:

```
Self.Count + 123
```

那么在上面的系结表达式中 Self.Count 到底是什么呢? 这就要看看这个系结表达式是在什么执行范围内执行，例如如果在执行这个系结表达式之前先系结执行范围到一个 TListBox 对象，那么系结表达式中的 Self 就指这个 TListBox 对象，因此 Self.Count 就是指存取这个这个 TListBox 对象的 Count 特性值，因此上面的系结表达式的意义就是指:

```
把 TListBox 对象的 Count 特性值加上 123 再指定给控制对象
```

有了这个基本的观念之后我们就可以解释上一节系结表达式的意义了:

```
bdsdbxBooks.VID.Lookup(_lds, 'VID', Text, 'NAME');
```

在这个系结表达式中的 _lds 暂时变量指的就是它的执行范围，也就是 LookupDataSource 特性值代表的对象 bdsdbxPublishers 又由于 bdsdbxPublishers 组件是 TBindSourceDBX 类别型态，而 TBindSourceDBX 类别实作了 IScopeLookup 接口，在 IScopeLookup 接口中提供了 Lookup 方法的宣告:

```
IScopeLookup = interface
  ['{95C4149E-E1AD-4D21-A8DF-A84A33B6D2D9}']
  function Lookup(const KeyFields: string; const KeyValues: TValue;
```

```
    const ResultFields: string): TValue;
  procedure GetLookupMemberNames(AList: TStrings);
end;
```

因此 bdsdbxPublishers 组件实作了 Lookup 方法，所以这个系结表达式才能够呼叫 Lookup 方法，现在这个系结表达式就不难了解了，对吧!

其实 IScopeLookup 接口的 Lookup 方法和 TClientDataSet 的 Lookup 方法功能是类似的，它的第一个参数是搜寻域名，第二个参数是搜寻值，第 3 个参数则是 Lookup 方法回传的域值，因此 Lookup(_lds, 'VID', Text, 'NAME')的意思就是使用 EditVID 组件的 Text 特性值搜寻 bdsdbxPublishers 数据源的 VID 字段，搜寻到符合的 VID 值之后就回传 bdsdbxPublishers 数据源中 NAME 字段的数值。

现在我们就可以解释这 3 种系结表达式的意义了。

简单的系结表达式(Simple Expressions)

由执行范围，TBindingExpression 对象和字符串表达式形成的系结表达式，简单的系结表达式一般是使用来计算数值并且和拖管系结表达式或是未拖管系结表达式一起使用。

拖管系结表达式(Managed Bindings)

拖管系结表达式和未拖管系结表达式是开发人员在使用实时数据系结技术时最常使用的 2 种表达式，这 2 种表达式都是使用来执行系结表达式然后把执行结果指定给控制组件，但拖管系结表达式是由实时数据系结引擎所拥有并且可自动被执行和重新计算。

未拖管系结表达式(Unmanaged Bindings)

未拖管系结表达式是由应用程序所拥有而且必须藉由呼叫系结对象的 Evaluate 方法才会被实时数据系结引擎解释和执行，它不像拖管系结表达式是由实时数据系结引擎自动解释和执行的。

在稍后的章节会使用程序代码来真正的说明如何使用这 3 种系结表达式，现在让我们对实时数据系结原理的说明暂时的告一段落，在稍后的章节中本书会详细的说明，现在让我们继续讨论实时数据系的 Lookup 功能。

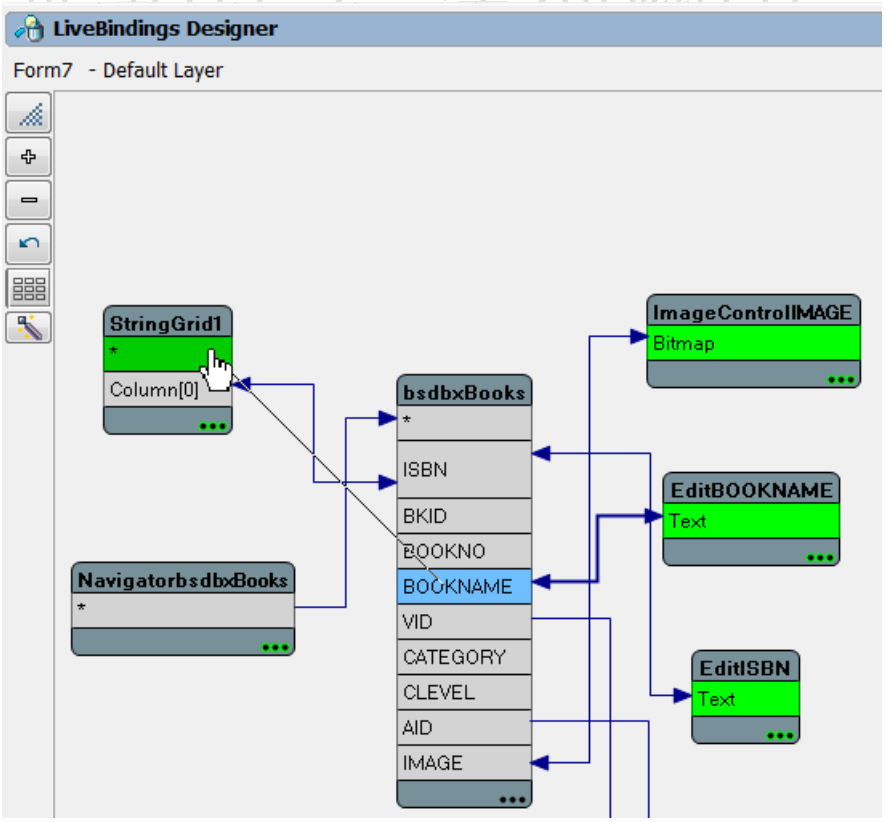
8-3 进阶 Lookup 功能

在 2-1 节说明了如何使用实时数据系结的 Lookup 功能，在简单的应用中开发人员可以藉由可视化实时数据系结设计家来完成 Lookup 功能，但对于比较复杂的 Lookup 应用开发人员仍然需要使用系结表达式。

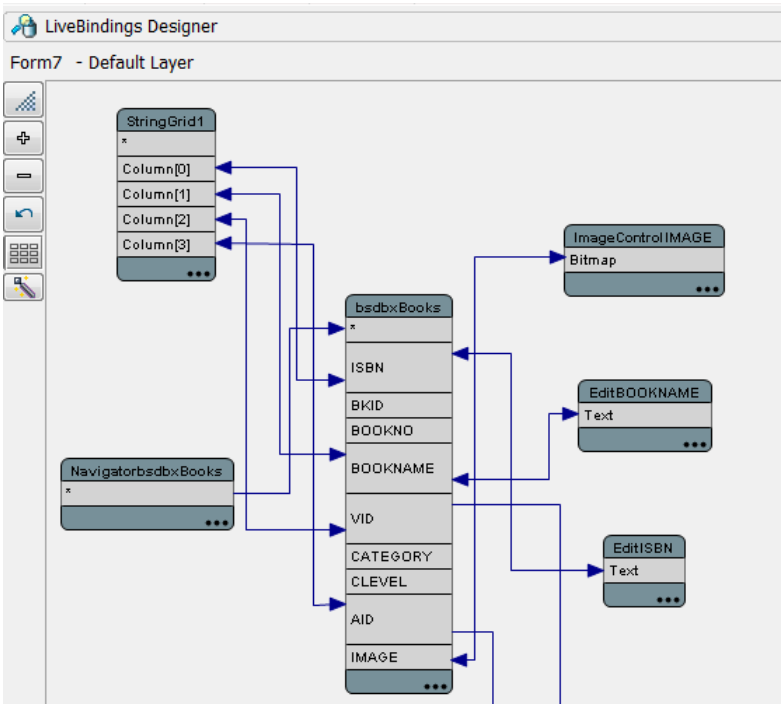
在阅读完 2-2 小节之后读者应该已经具备了系结表达式的基本概念，现在就让我们使用系结表达式进行比较深入的 Lookup 应用。

假设现在我们想使用 Grid 组件来显示 BOOKS 数据表中的数据，但我们仍然希望在 Grid 中显示 VID 和 AID 字段时也是显示 Publishers 和 Performers 数据表中的 NAME 字段而不是 BOOKS 数据表的 VID 和 AID 域值。但问题是如何让 Grid 组件也能够执行 Lookup 的查询?

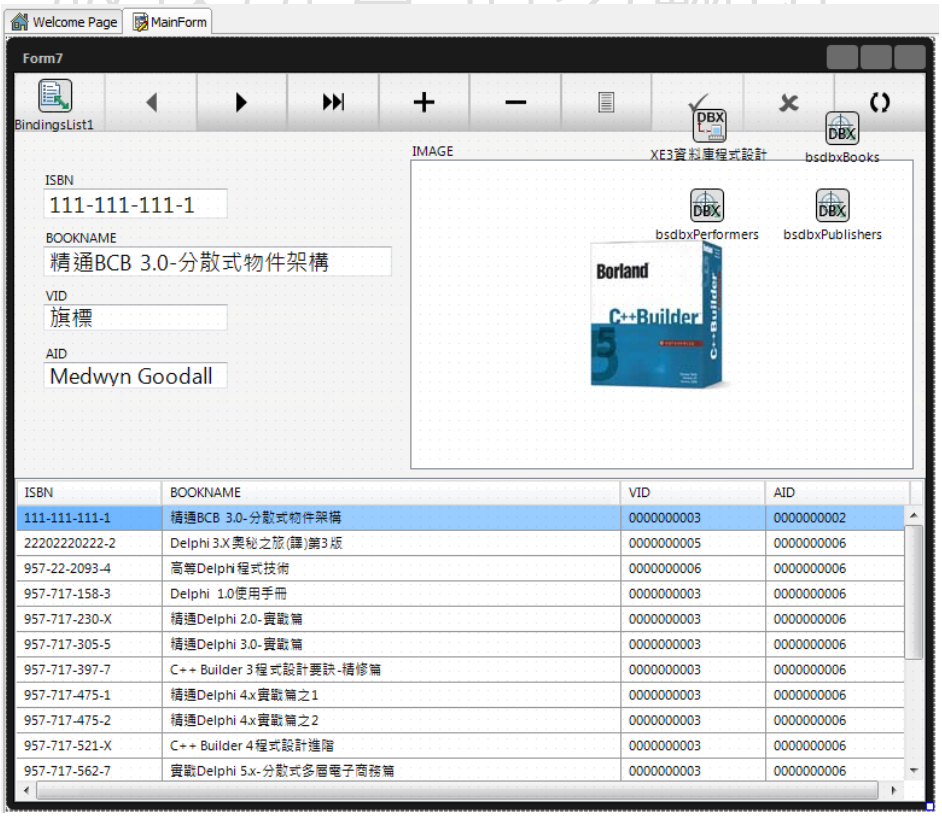
现在请开启 2-2 小节中的范例，在主窗体中加入一个 TStringGrid 组件并且对齐在主窗体的下方，再开启可视化实时数据系结设计家，分别拖曳 ISBN，BOOKNAME，VID 和 AID 到 StringGrid1 实体的『*』 字段中，就可以把这些字段显示在 TStringGrid 中，如下所示:



拖曳完这 4 个字段后，在可视化实时数据系结设计家中就可以看到 StringGrid1 实体产生了 4 个 Column 字段来显示这个 4 个字段，如下显示:



现在主窗体应该看起来如下所示，请注意 TStringGrid 组件显示的 VID 和 AID 字段数值并不是我们希望看到的数值，我们希望 VID 是显示 Publishers 数据表中的 NAME 域值，而 AID 则是显示 Performers 数据表中的 NAME 域值。



为了要达到这个目的我们就需要撰写系结表达式了，但想写系结表达式我们需要弄清楚系结表达式的执行范围，由于 TStringGrid 组件是系结到 bsdbxBooks，因此在 TStringGrid 组件中撰写的系结表达式其执行范围就是 bsdbxBooks，由于 bsdbxBooks 实作了 IScopeLookup 接口的 Lookup 方法，因此我们就可以藉由 Lookup 方法到 Publishers 和 Performers 数据表中根据 VID 和 AID 查询 NAME 域值，但是在 bsdbxBooks 的执行范围中要如何能够存取代表 Publishers 和 Performers 数据表的 bsdbxPublishers 和 bsdbxPerformers 呢?

基本上我们现在要做的事情等于是要从 bsdbxBooks 执行范围存取到 bsdbxPublishers 和 bsdbxPerformers 的执行范围，才能够再分别呼叫 bsdbxPublishers 和 bsdbxPerformers 实作的 Lookup 方法来根据 VID 和 AID 查询 NAME 域值，如果您还想不通的话，那就想想如果是使用程序代码的话，您是不是也要使用类似下面的程序代码来查询呢?

```
PublishersClientDataSet.Lookup('VID'...);
PerformersClientDataSet. .Lookup('AID'...);
```

上面的 PublishersClientDataSet 和 PerformersClientDataSet 也就是类似执行范围的意思，现在读者了解了吗?

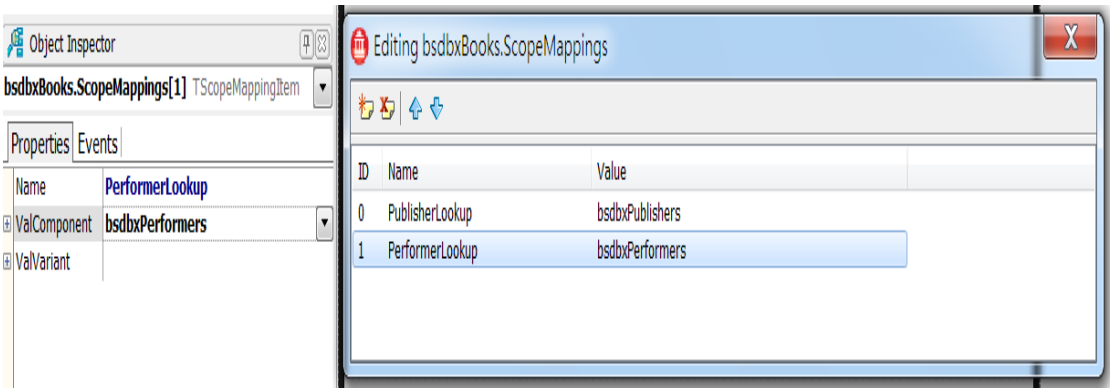
在 TBindSourceDBX 组件中有一个 ScopeMappings 特性，这个特性的功能就是让开发人员在其中定义『名称:执行范围』的数值，如此一来开发人员就可以藉由 ScopeMappings 特性值中的定义来存取其他的执行范围，再从其他的执行范围来执行系结表达式。

这不就解决了我们的问题了吗? 让我们在 bsdbxBooks 的 ScopeMappings 特性中定义如下的数值:

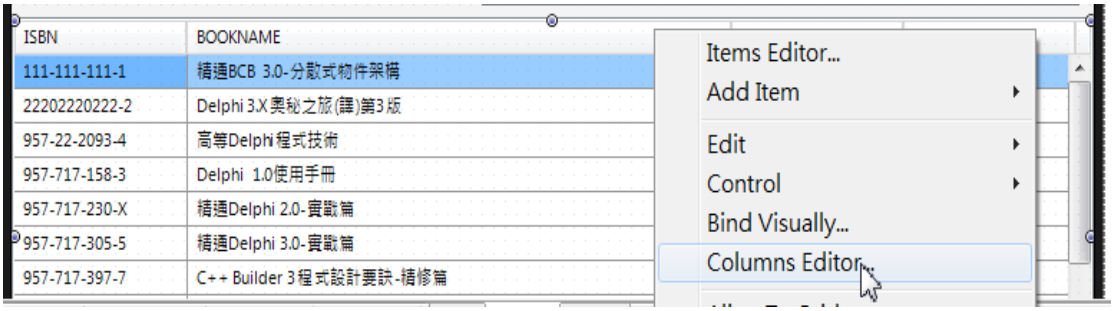
```
PublisherLookup:bsdbxPublishers
PerformerLookup:bsdbxPerformers
```

那么就可以在 bsdbxBooks 的执行范围中藉由 PublishersLookup 这个标识符来存取 bsdbxPublishers 的执行范围，再藉由 PerformersLookup 标识符来存取 bsdbxPerformers 的执行范围了。

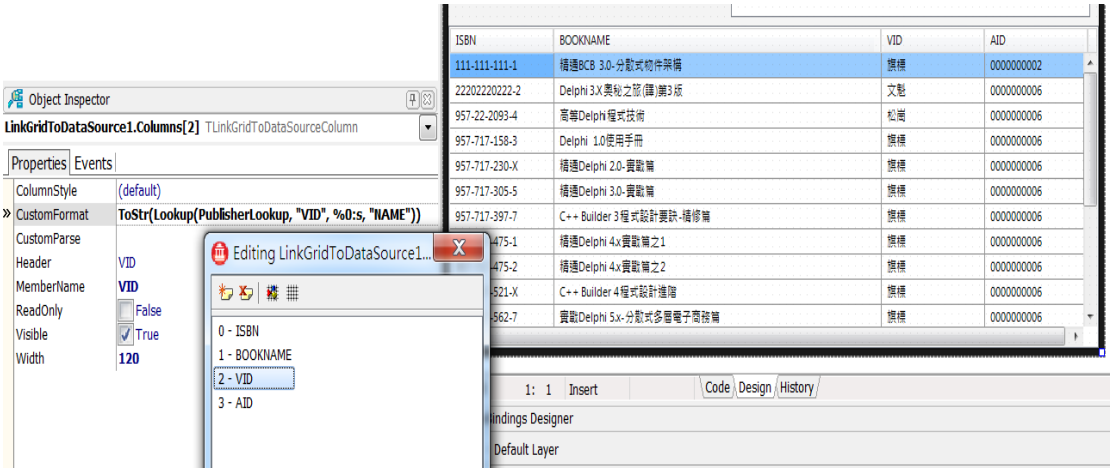
现在请点选主窗体中的 bsdbxBooks 组件，在对象查看器中双击它的 ScopeMappings 特性，在 ScopeMappings 特性的编辑器中点选左上方的  按钮以增加新的『名称:执行范围』定义，在对象查看器中于 Name 特性输入 PublisherLookup，于 ValComponent 特性输入 bsdbxPublishers。接着用同样的方法加入『PerformerLookup:bsdbxPerformers』，如下所示:



接着点选主窗体中的 TStringGrid 组件，右键鼠标从快捷菜单中选择『Column Editors…』如下所示：



然后从字段编辑器中选择 VID 字段，在它的 CustomFormat 特性中输入系结表达式，如下所示：



VID 字段的 CustomFormat 特性是指当 TStringGrid 组件要显示 VID 域值时，会使用 CustomFormat 特性值中定义的格式来显示内容，因此我们只需要在 CustomFormat 特性值中以 bsdbxBooks 的 VID 的域值从 bsdbxPublishers 执行范围使用 Lookup 方法根据 bsdbxBooks.VID 来查询 bsdbxPublishers.NAME 来取代原先要显示的数值即可，因此请在 VID 字段的 CustomFormat 特性中输入如下的系结表达式：

```
ToStr(Lookup(PublisherLookup, "VID", %0:s, "NAME"))
```

让我们解释上面系结表达式的意义。

先看看

```
Lookup(PublisherLookup, "VID", %0:s, "NAME")
```

它的意思就是在 bsdbxBooks 执行范围中存取 PublisherLookup 标识符，由于 PublisherLookup 代表 bsdbxPublishers，因此上面的系结表达式也就代表:

```
Lookup(bsdbxPublishers, "VID", %0:s, "NAME")
```

这个系结表达式是非常正统的面向对象呼叫惯例，因为在面向对象程序语言中呼叫对象方法时第一个隐藏参数就是对象本身，在 Delphi 程序语言中就是 **Self**，因此上面的系结表达式可以看成是:

```
bsdbxPublishers.Lookup(bsdbxPublishers, "VID", %0:s, "NAME");
```

也就是:

```
bsdbxPublishers.Lookup("VID", %0:s, "NAME");
```

根据 VID 字段来查询并且回传 NAME 域值，上面的第二个参数值%0:s 就是原本 bsdbxBooks 的 VID 域值，『%0:s』只是要求 bsdbxBooks 的 VID 域值根据这个字符串格式带入到 Lookup 方法中。

上面的系结表达式中的 ToStr 则是系结引擎内建的全局方法，把它的参数转换为字符串型态。

现在读者应该充分了解了上面的系结表达式的意义了吧，系结表达式并不困难，只要了解了执行范围的观念之后，就等于呼叫对象的方法或是存取对象的特性值而已。

最后其实上面的系结表达式中的 ToStr 方法是不需要呼叫的，为什么？ 因为 Lookup 方法查询的 NAME 字段本身是字符串型态的字段，因此我们可以把上面的系结表达式最终修改成:

```
Lookup(PublisherLookup, "VID", %0:s, "NAME")
```

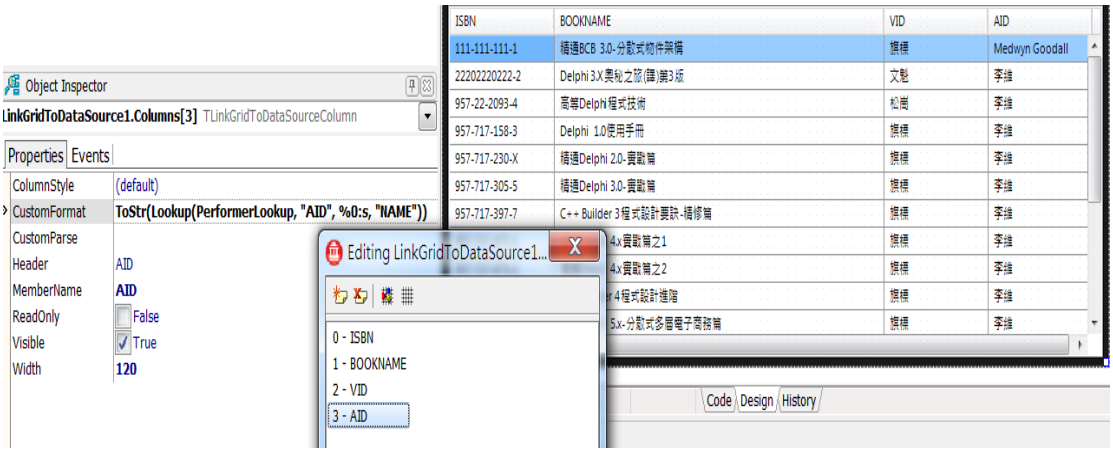
现在请为 AID 字段进行相同的系结表达式撰写流程，当然 AID 字段的系结表达式应该是:

```
ToStr(Lookup(PerformerLookup, "AID", %0:s, "NAME"))
```

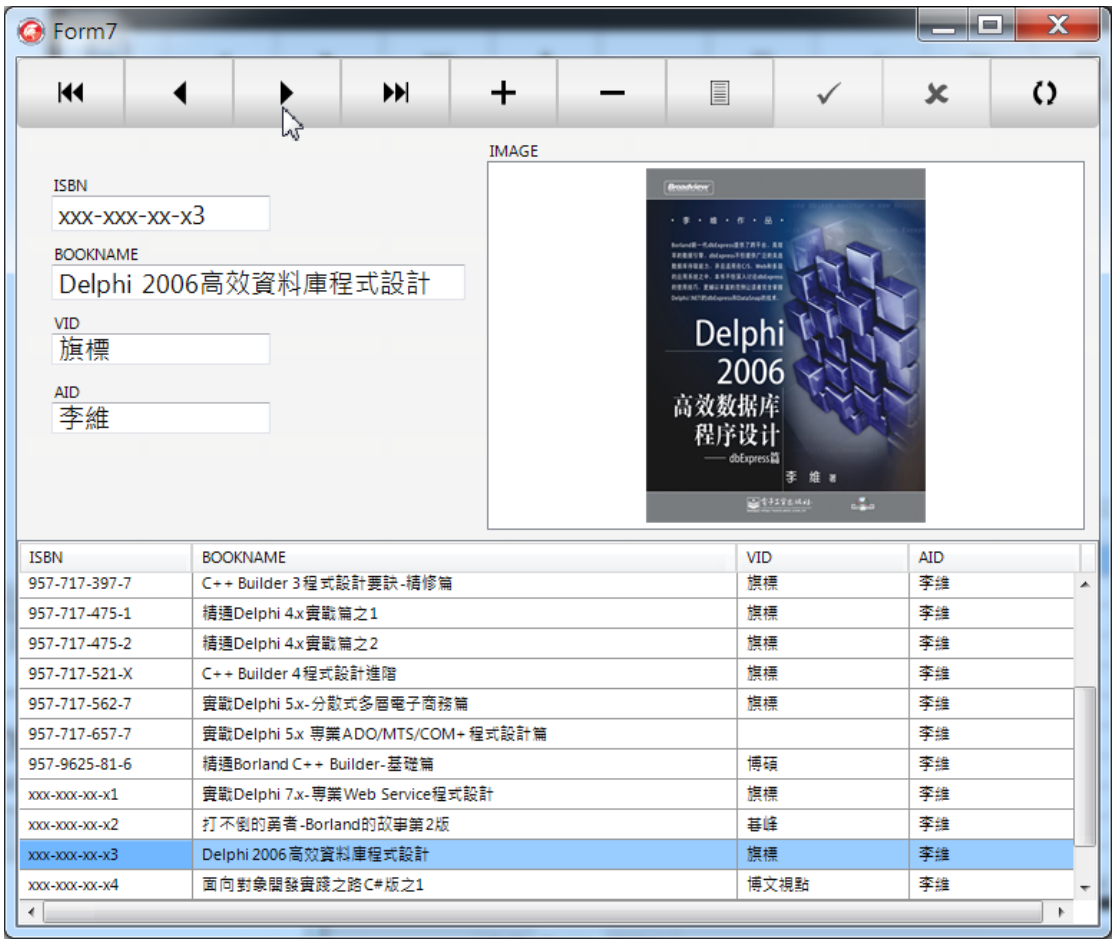
当然 AID 字段的最终系结表达式应该写成:

```
Lookup(PerformerLookup, "AID", %0:s, "NAME")
```

即可，也无需再呼叫 ToStr。



現在请编译和执行此范例 FireMonkey 应用程序，它应该看起来如下所示，读者可以看到在 TStringGrid 的 VID 和 AID 字段果然显示了我们需要的数值了。开发人员在了解和掌握了系结表达式之后就可以在可视化实时数据系结设计家无法自动产生系结表达式的场合中直接使用系结表达式来进行更具弹性的设计了。



8-4 结论

本章说明了如何使用实时数据系结的 Lookup 功能来进行查询的功能，也说明了实时数据系结重要的观念:系结表达式的种类，执行范围等，开发人员必须了解这些基本观念才能够了解可视化实时数据系结设计家在做什么，也能够自己撰写系结表达式和在程序代码中使用系结对象。

在掌握了实时数据系结的基本观念之后开发人员就可以使用系结表达式来进行高等的 Lookup 功能，以便在可视化实时数据系结设计家无法自动产生系结表达式的场合中直接撰写系结表达式。

对于开发一般的 FireMonkey 数据库应用程序来说，开发人员使用可视化实时数据系结设计家，设定 Master/Detail 关系和使用 Lookup 功能应该就能够完成大多数的开发工作了，再加上开发人员了解 dbExpress 框架技术之后，开发有效率的 FireMonkey 数据库应用程序就应该很简单了，但要充分掌握实时数据系结技术，开发人员仍然需要学习系结类别和系结对象，这些正是随后章节讨论的重点。

第9章 实时数据系结框架

DX10 的可视化实时数据系结设计家虽然非常的易于使用，但开发人员要能够彻底了解可视化实时数据系结设计家产生的系结表达式的原理以及正确的使用系结表达式，那么开发人员必须掌握实时数据系结框架。

实时数据系结框架是由许多类别所组成，不同的类别负责不同系结表达式的运作，但不管是框架中的那一个类别，基本上开发人员在使用系结类别和程序代码来运作系结表达式时，都必须以下列的步骤来执行系结表达式:

1. 定义系结执行范围
2. 撰写系结表达式
3. 呼叫系结对象的 **Evaluate** 方法真正要求系结引擎根据执行范围来解释和执行系结表达式

在前面讨论可视化实时数据系结的章节中这 3 个步骤已经由实时数据系结框架和可视化实时数据系结设计家帮我们实行了，因此我们并没有撰写任何的程序代码，但为了让读者真正的掌握实时数据系结框架，我们就必须清楚的了解如何执行这 3 个步骤，现在让我们从最简单的系结对象开始，藉由程序代码来说明系结对象如何使用系结表达式。

9-1 建立实时数据系结概念

让我们使用系结表达式框架中最简单的系结类别 TBindingExpression 来说明如何前面的观念，我们将使用 TBindingExpression 类别对象来执行系结表达式，让读者了解在使用系结表达式执行系结表达式时需要的执行步骤。

首先请在 Delphi 整合发展环境中建立一个 FireMonkey Desktop Application 项目，并且在主窗体中放入一个 TEdit 组件以准备输入系结表达式，一个 TListBox 组件以

显示执行系结表达式的结果，以及一个『Evaluate』按钮以执行 TEdit 组件中的系结表达式。

首先在『Evaluate』按钮的 OnClick 事件处理函式中呼叫 ProcessExpressions 方法执行 TEdit 组件中用户输入的系结表达式:

```
procedure TForm7.Button1Click(Sender: TObject);
begin
  ProcessExpressions;
end;
```

下面的程序代码就是 ProcessExpressions 方法的实作程序代码和相关的方法，让我们详细的说明这些程序代码的意义:

```
001   procedure TForm7.DisplayValue(const sExpression : String;
AValueIntf: IValue);
002   var
003     LValue: TValue;
004     LType: PTypeInfo;
005   begin
006     LValue := AValueIntf.GetValue;
007     LType := AValueIntf.GetType;
008     if LValue.IsEmpty then
009       Memo1.Lines.Add('Empty')
010     else
011     begin
012       if LValue.IsObject then
013         Memo1.Lines.Add(Format('%s ClassName: %s', [sExpression,
LValue.AsObject.ClassName]))
014       else
015         try
016           Memo1.Lines.Add(Format('%s %s ToString: "%s"',
[sExpression, LType.Name, LValue.ToString]));
017         except
018           on E: Exception do
019             Memo1.Lines.Add(E.ClassName + ': ' + E.Message);
020         end;
021     end;
022   end;
023
```

```
024   procedure TForm7.ProcessExpressions;
025   var
026     LInputExpr: string;
027     LBindingExpression: TBindingExpression;
028     LScope: IScope;
029     LTestObject: TObject;
030   begin
031     LTestObject := TDictionaryScope.Create;
032     try
033       LScope := WrapObject(LTestObject);
034       LBindingExpression := TBindings.CreateExpression( LScope,
edtExpression.Text);
035       try
036         DisplayValue('系结表达式 ' + edtExpression.Text + ' ==> ' ,
LBindingExpression.Evaluate);
037       finally
038         LBindingExpression.Free;
039       end;
040     finally
041       LScope := nil;
042       LTestObject.Free;
043     end;
044   end;
```

让我们先讨论 024 行开始的 ProcessExpressions 方法，首先 031 行先建立了 TDictionaryScope 对象，033 行呼叫 WrapObject 方法并且传递 TDictionaryScope 对象做为参数，呼叫 WrapObject 方法的目的就是把 TDictionaryScope 对象封装为执行范围，也就是说稍后执行的系结表达式其存取的范围就是 TDictionaryScope 对象。

TDictionaryScope 类别是系结表达式框架中一个执行范围类别,读者可以把它想成一个空白的执行范围，它是能够提供

的功能，也就是说它可以把一个执行范围和一个标识符系结在一起，因此在系结表达式中使用这个标识符就可以存取到这个执行范围。

另外执行范围是可以巢状化的，这是说开发人员可以在一个执行范围中再内嵌其他的执行范围，稍后我们会看到这个巢状执行范围的范例，现在让我们继续说明上面程序代码的意义。

请读者看看 034 行，它呼叫 TBindings 类别的类别方法 CreateExpression 以建立一个 TBindingExpression 对象，它的原型宣告如下:

```
class function TBindings.CreateExpression(const InputScopes: array of
IScope; const BindExprStr: string): TBindingExpression;
```

CreateExpression 方法接受执行范围和系结表达式做为参数再回传建立的 TBindingExpression 对象，之后程序代码就在 036 行呼叫 TBindingExpression 对象的 Evaluate 方法根据执行范围来执行系结表达式。

到这里为止读者可以看到上面的程序代码果然是依照:

1. 定义系结执行范围 – TDictionaryScope 对象

2. 撰写系结表达式 – 034 行

3. 呼叫系结对象的 **Evaluate** 方法真正要求系结引擎根据执行范围来解释和执行系结表达式 – 036 行

的步骤来执行系结表达式。

再让我们继续说明下去。

下面是 TBindingExpression 类别的 Evaluate 方法的原型宣告:

```
   function Evaluate: IValue; virtual; abstract;
```

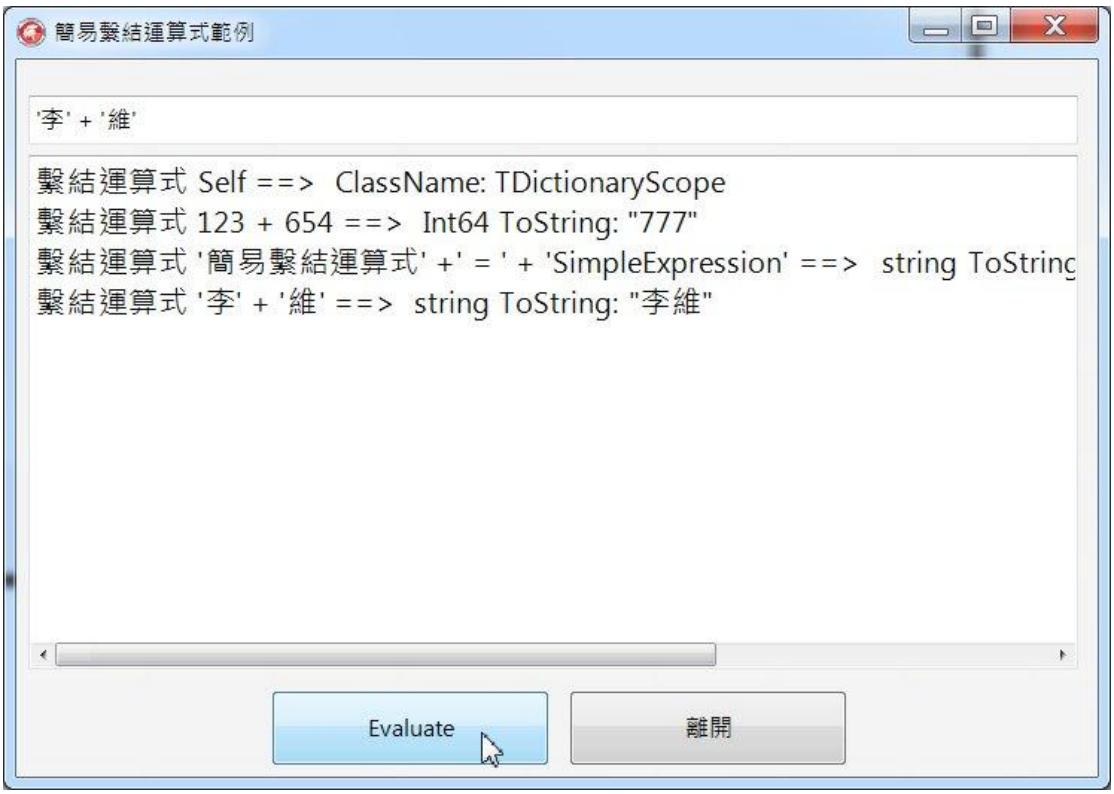
在 TBindingExpression 类别的 Evaluate 方法呼叫系结引擎执行了系结表达式之后，Evaluate 方法会回传 IValue 接口代表执行结果，而下面是 IValue 接口的宣告原型:

```
  IValue = interface
    ['{A495F901-72F5-4384-BA50-EC3B4B42F6C2}']
    /// <summary>Used to obtain the type information for the actual
value.</summary>
    /// <returns>The type information of the actual value.</returns>
    function GetType: PTypeInfo;
    /// <summary>Gives the actual value wrapped by this
interface.</summary>
    /// <returns>The actual value.</returns>
    function GetValue: TValue;
  end;
```

在 IValue 接口中 GetType 方法可以取得执行结果的型态信息，而 GetValue 则可以取得执行结果值，这些都属于 Delphi RTTI 的相关接口和类别，系结表达式框架使用 RTTI 来动态执行系结表达式和使用 RTTI 来提供执行结果的信息。

因此在 DisplayValue 方法中就使用 IValue 接口来取得执行结果并且显示在主表格的 TMemo 组件中。

现在请编译和执行此范例程序并且试着在主表格的 TEdit 组件中输入一些系结表达式再点选『Evaluate』按钮来执行系结表达式。在下面的画面中您可以看到笔者执行的几个系结表达式:



下面的表格说明了这些系结表达式的意义:

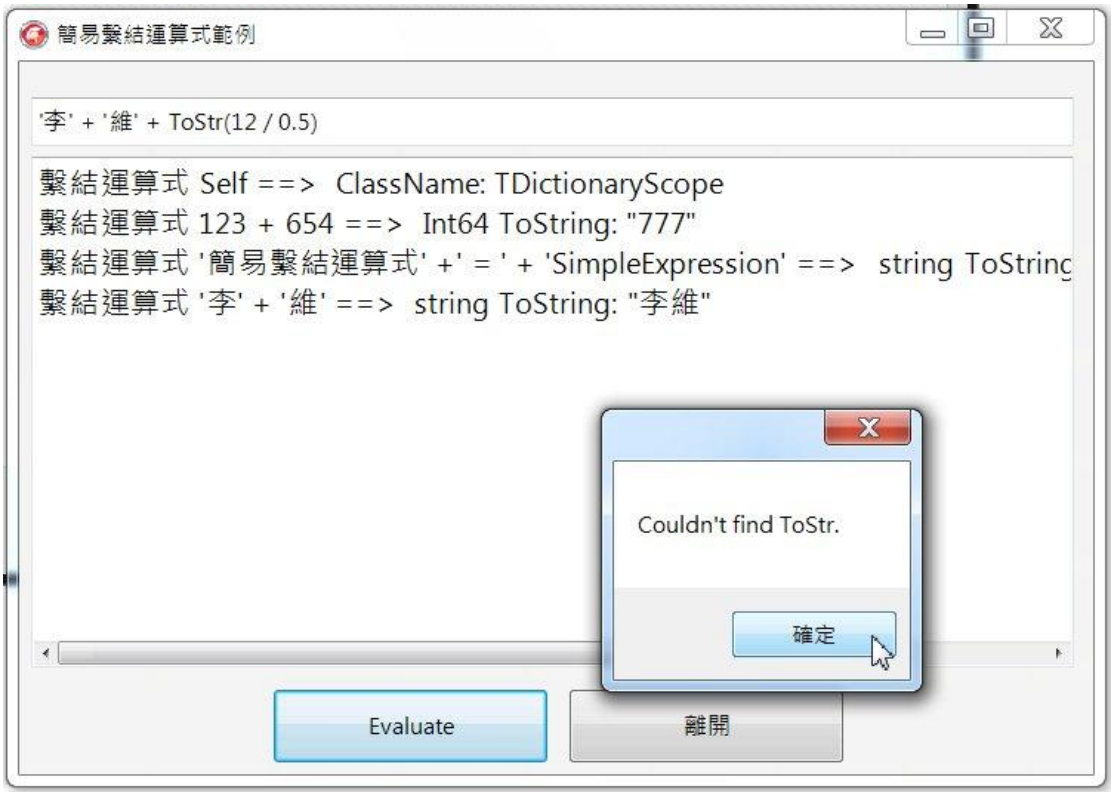
系结表达式	执行结果
Self	TDictionaryScope 类别对象，这是因为上面的 031~033 行的程序代码系结 TDictionaryScope 为执行范围，而 Self 就是指执行范围
123 + 654	777，这个系结表达式可证明简单的系结表达式可执行常数值的运算
'李' + '维'	'利瓦伊'，这个系结表达式可证明简单的

	系结表达式可执行字符串的简单运算

但现在笔者在下面的画面中试着执行:

```
'李' + '维' + ToStr(12/ 0.5)
```

却失败了，为什么？从下图中我们可以看到范例应用程序说找不到 ToStr 函式，这就是充分的暗示了，因为在目前的执行范围中没有 ToStr 方法的定义，因为 TDictionaryScope 类别没有定义 ToStr 方法。

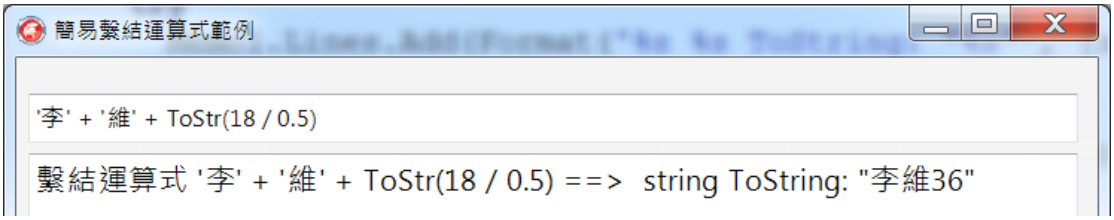


当然如果我们定义的执行范围中有 ToStr 方法的定义的话那么在系结表达式中就可以呼叫 ToStr，现在让我们回到范例项目，在 WrapObejct 程序代码下加入新的一行程序代码 TNestedScope 如下:

```
   LScope := WrapObject(LTestObject);
   LScope :=
TNestedScope.Create(LScope,TBindingMethodsFactory.GetMethodScope);
```

TNestedScope 类别可以在原本的执行范围中再嵌入另外的执行范围以形成巢状式执行范围，因此在上面的程序代码中是在原本的执行范围中再加入系结引擎的全局方法，因为 TBindingMethodsFactory 类别的 GetMethodScope 类别方法即可回传系结引擎的全局方法。因此上面这行新加入的程序代码的意思就是在原本只能存取 TDictionaryScope 对象的方法和特性的执行范围中再加入可存取系结引擎全局方法的执行范围。

现在再次执行范例应用程序就可看到在系结表达式中可以呼叫 ToStr 方法了，因为 ToStr 方法正是系结引擎的全局方法之一:



我们可以使用下面的程序代码显示出系结引擎的全局方法:

```
procedure TForm7.DisplayMethods;
var
  LDescription: TMethodDescription;
begin
  Memo1.Lines.Add('系结引擎全局方法: ');
  for LDescription in TBindingMethodsFactory.GetRegisteredMethods do
    if LDescription.DefaultEnabled then
      Memo1.Lines.Add(LDescription.Name)
    else
      Memo1.Lines.Add(LDescription.Name + ' (暂停使用)');
end;
```

TBindingMethodsFactory 的类别方法 GetRegisteredMethods 可回传所有在系结引擎中注册的全局方法，下面是 GetRegisteredMethods 的宣告原型：

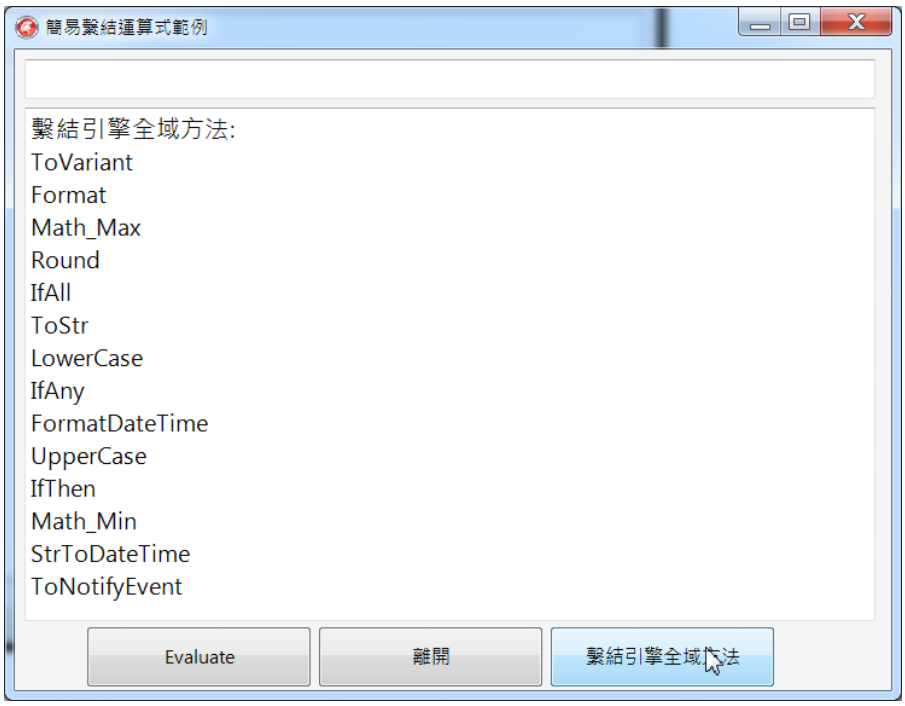
```
    class function GetRegisteredMethods: TArray<TMethodDescription>;
```

GetRegisteredMethods 可回传 TArray<TMethodDescription>型态的执行结果，在回传的 TArray 中包含了叙述每一个全局方法的 TMethodDescription 记录型态对象，从 TMethodDescription 中我们就可以在 for...in ...循环中取得全局方法的信息，例如方法的名称，方法属于的程序单元名称，方法是否作用中，以及可透过 TMethodDescription 记录型态对象的 Invokable 特性呼叫全局方法等，下面就是 TMethodDescription 记录型态的定义：

```
  TMethodDescription = record
  strict private
    FInvokable: IInvokable;
    FID: string;
    FName: string;
    FUnitName: string;
    FDefaultEnabled: Boolean;
```

```
    FFrameworkClass: TPersistentClass;
    FDescription: string;
  public
    constructor Create(const AInvokable: IInvokable; const AID, AName,
AUnitName: string;
     ADefaultEnabled: Boolean; const ADescription: string;
AFrameworkClass: TPersistentClass); overload;
    property ID: string read FID;
    property Name: string read FName;
    property UnitName: string read FUnitName;
    property DefaultEnabled: Boolean read FDefaultEnabled;
    property FrameWorkClass: TPersistentClass read FFrameWorkClass;
    property Invokable: IInvokable read FInvokable;
    property Description: string read FDescription;
  end;
```

执行上面的程序代码可以看到如下的执行结果:



在 TListBox 中果然显示出了目前在系结引擎中注册的所有全局方法。

再回到范例应用程序，让我们修改执行范例为一个 TStringList，读者可以看到在下面的程序代码中我们是建立 TStringList 对象，并且根据 TStringList 对象来建立执行范围：

```
procedure TForm7.CreateScopeObejct;
begin
  if (LTestObject = Nil) then
  begin
    LTestObject := TStringList.Create;
    LScope := WrapObject(LTestObject);
    LScope := TNestedScope.Create(LScope,
TBindingMethodsFactory.GetMethodScope);
  end;
end;
```

下面的程序代码则是根据上面建立的执行范围来执行系结表达式:

```
procedure TForm7.Button4Click(Sender: TObject);
var
  LInputExpr: string;
  LBindingExpression: TBindingExpression;
begin
  LBindingExpression := TBindings.CreateExpression( LScope,
edtExpression.Text);
  try
    DisplayValue('系结表达式 ' + edtExpression.Text + ' ==> ' ,
LBindingExpression.Evaluate);
  finally
    LBindingExpression.Free;
  end;
end;
```

下面是范例应用程序执行的结果画面:

下面的表格说明了这些系结表达式的意义:

系结表达式	执行结果
Self	TStringList，这是因为上面的程序代码系结 TStringList 为执行范围，而 Self 就是指执行范围
Self.Add('实时数据系结')	使用系结表达式呼叫 TStringList 对象的 Add 方法在 TStringList 对象中加入一个字符串。系结表达式回传 0，代表加入的字符串索引位置
Self.Add('DataSnap XE7') Self.Add('Mobile Studio')	同上系结表达式回传 1，2，分别代表加入的字符串索引位置
Self.Text	使用系结表达式存取 TStringList 对象的 Text 特性值
Self.Strings[2]	使用系结表达式存取 TStringList 对象的 Strings 特性值，系结表达式回传'Mobile

	Studio’

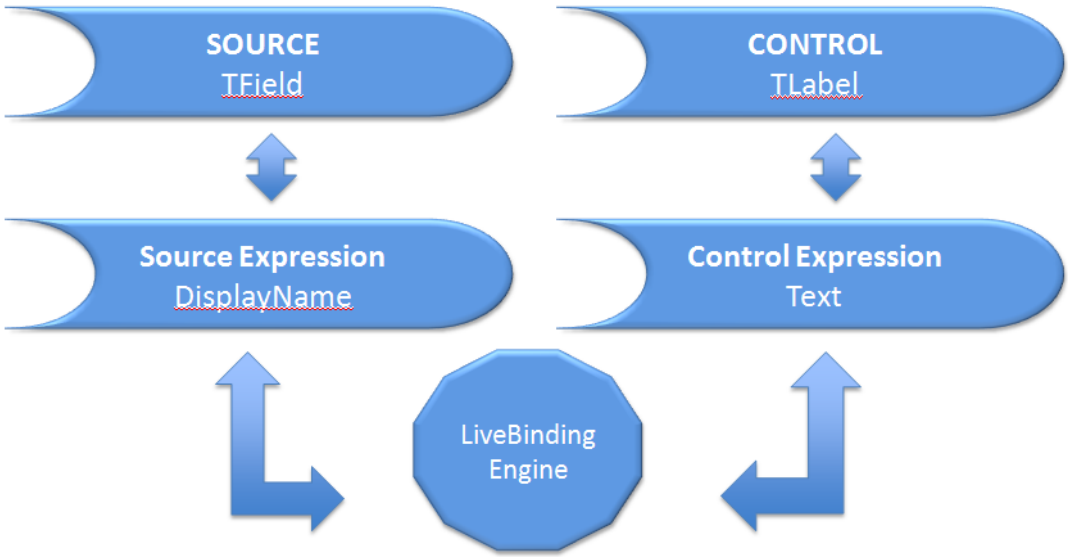
从这个范例我们可以看到在执行范围内我们可以存取这个执行范围中对象的方法和特性，就像上面的系结表达式呼叫 TStringList 对象的方法和存取 TStringList 对象的特性值一样。这个范例也说明了开发人员几乎可使用系结表达式来执行 Delphi 程序代码可执行的工作，但系结表达式可在 Delphi 应用程序执行时动态的改变并重新执行以提供动态语言的机制，这是目前 Delphi 程序语言无法提供的。

现在读者应该可以了解为什么系结框架这么的强大了，因为 XE7 的系结框架可以为 Delphi 所有的对象和组件建立执行范围，一旦建立了执行范围之后系结表达式就可以进行处理了。

了解了系结原理之后让我们直接使用系结组件来印证一下前面所学习的概念。

9-1-1 使用 TBindExpression 组件

在说明如何使用 TBindExpression 组件来使用系结表达式以印证前面讨论的概念之前，请读者再次观看一下上一章中已经说明的实时数据系结架构图:

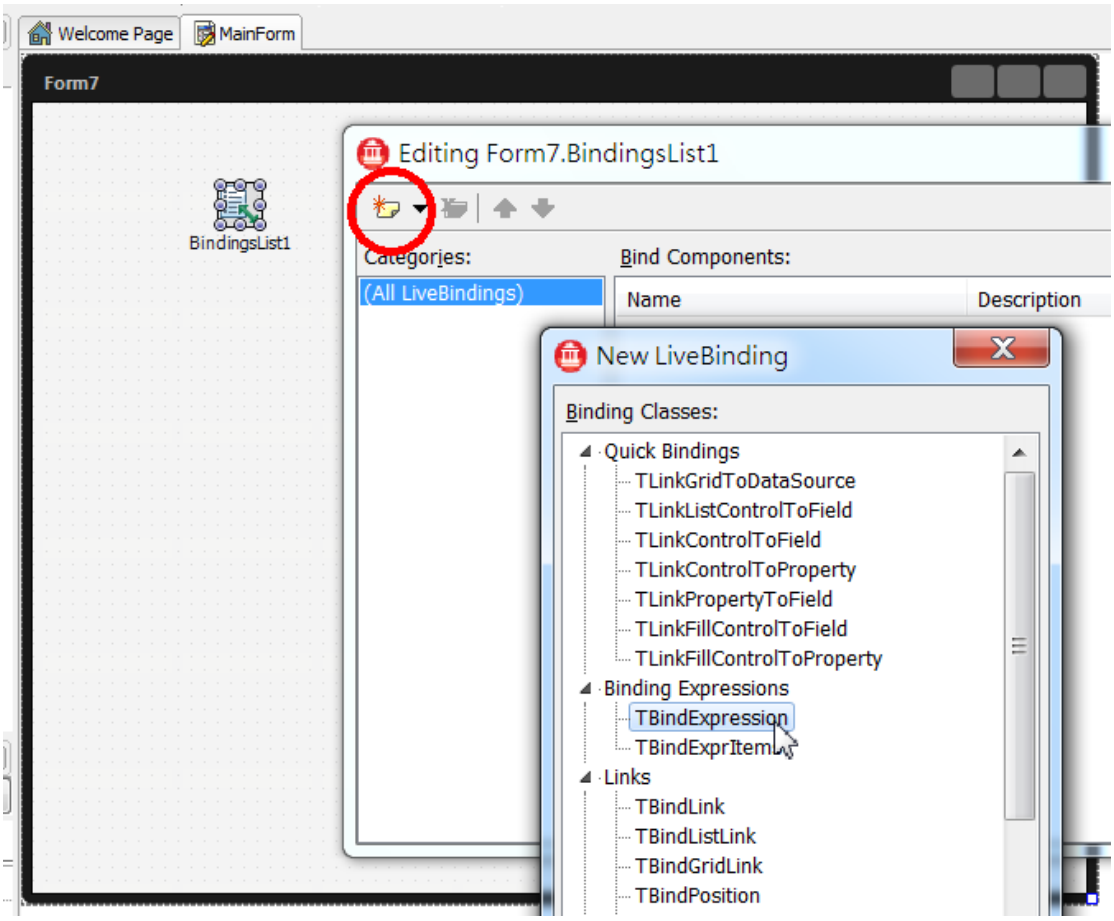


从上图中可以看到不管是来源组件或是控制组件都有一个系结表达式，在上一小节中我们只使用了来源系结表达式，这是因为上一小节是使用程序代码来说明最简单的系结表达式应用，但在 Delphi 支持系结框架的组件中都是使用上图的架构，因此 Delphi 的系结组件是以下列步骤来提供系结表达式的运作:

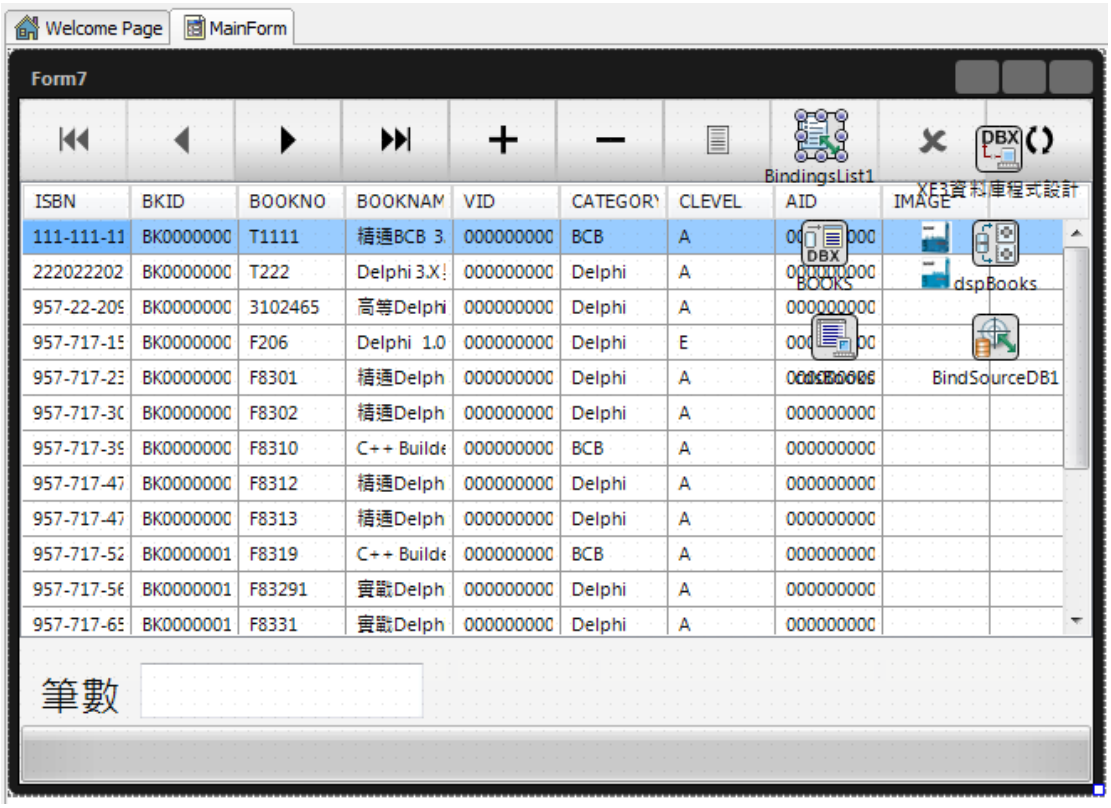
> 来源组件藉由系结引擎执行来源系结表达式之后，当来源组件要把系结表达式的执行结果指定给控制组件时，控制组件也会藉由系结引擎执行控制系结表达

式，然后来源组件的执行结果才会根据控制组件的控制系结表达式的执行结果方式指定给控制组件。

现在我们就可以藉由 Delphi 的 TBindExpression 组件来印证和说明了，请在 IDE 中建立一个 FireMonkey Desktop Application 项目，在主窗体中放入 TBindingsList 组件，双击它开启组件编辑器，再点选左上方的『New Binding』按钮以新增系结组件，再于 New LiveBinding 对话盒中选择建立 TBindExpression 组件，请在其中建立 3 个 TBindExpression 组件，如下所示：



接着在主窗体中使用 TSQLConnection 组件链接『XE7 数据库程序设计』，再使用 TSQLDataSet，TDataSetProvider，TClientDataSet 连结 BOOKS 数据表，再使用 TBindSourceDB 连结 TClientDataSet，最后在连结 TBindNavigator 和 TStringGrid，并且在主窗体下方放入 TEdit 和 TProgressBar 组件，我们希望在 TEdit 组件和 TProgressBar 组件中显示当前记录的相对位置，我们将使用荆面建立的 3 个 TBindExpression 组件来完成这些工作，最后主窗体如下所示：



OK，现在想想如何在主窗体的 TEdit 组件显示 cdsBooks 当前记录的相对位置呢?这很简单，因为 TClientDataSet 的 RecNo 和 RecordCount 特性值正好是我们要的，我们只需要在 TEdit 中显示『RecNo/RecordCount』这个信息即可，因此现在我们就可以写出两边的系结表达式了。

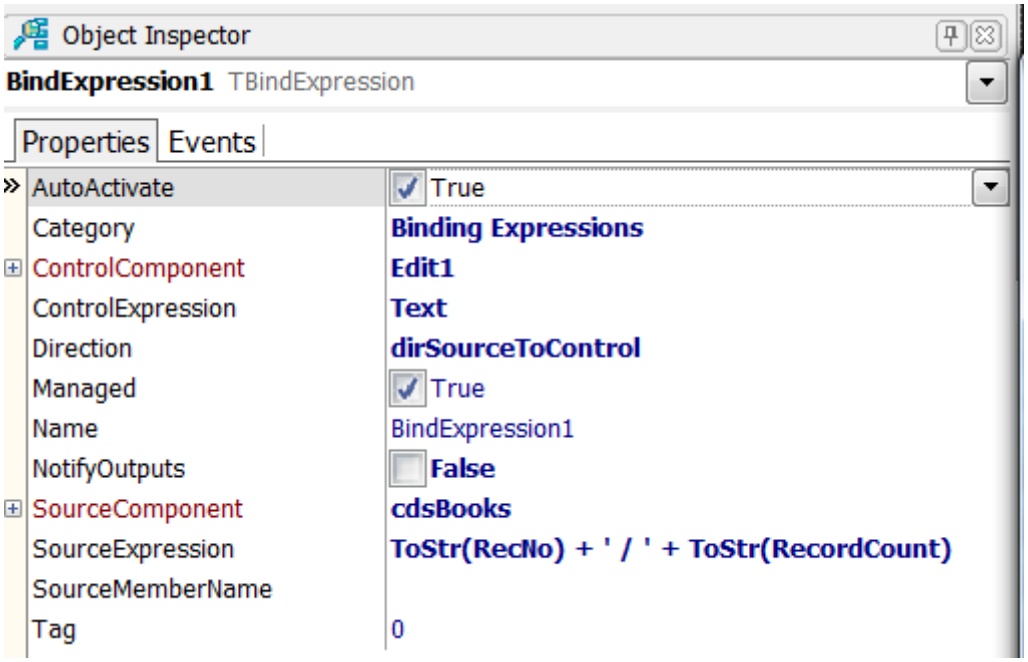
首先是来源方，来源的执行范围当然是 cdsBooks 组件因为我们要存取 cdsBooks 的 RecNo 和 RecordCount 特性值，而来源系结表达式则是『RecNo/RecordCount』，但由于控制方的数据源是字符串型态的数值，因此来源系结表达式应该是 ToStr(RecNo) + '/' + ToStr(RecordCount):

来源执行范围	来源系结表达式
cdsBooks	ToStr(RecNo) + '/' + ToStr(RecordCount)

控制方的执行范围当然就是 TEdit 组件了，那么控制方的系结表达式呢?由于我们是希望把结果显示在 TEdit 组件中，因此控制系结表达式当然就是 TEdit 组件的 Text 特性了，因此最后的结果是:

控制执行范围	控制系结表达式
Edit1	Text

了解了 2 方的执行范围和系结表达式之后，就请点选 TBindingsList 中的第 1 个 TBindExpression 对象，然后在对象查看器中设定如下的特性值:



在上面的对象查看器中读者可以看到 SourceComponent 特性设定为 cdsBooks，SourceExpression 特性设定为 ToStr(RecNo) + '/' + ToStr(RecordCount)，ControlComponent 特性设定为 Edit1，ControlExpression 特性设定为 Text，和前面讨论的系结表达式一模一样。

同样的要设定主窗体中的 TProgressBar 也可显示相对位置，那么我们需要撰写 2 个系结表达式，分别系结 cdsBooks 的 RecNo 和 RecordCount 特性和 TProgressBar 的 Value 和 Max 特性，下面的表格显示了这 2 个系结表达式:

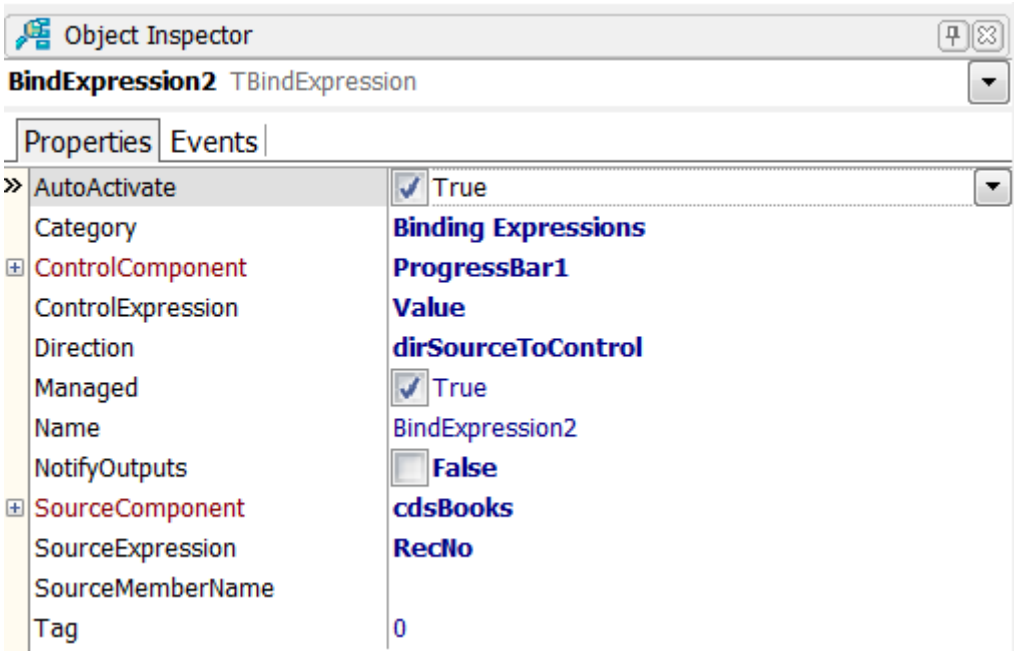
来源执行范围	来源系结表达式
cdsBooks	RecNo

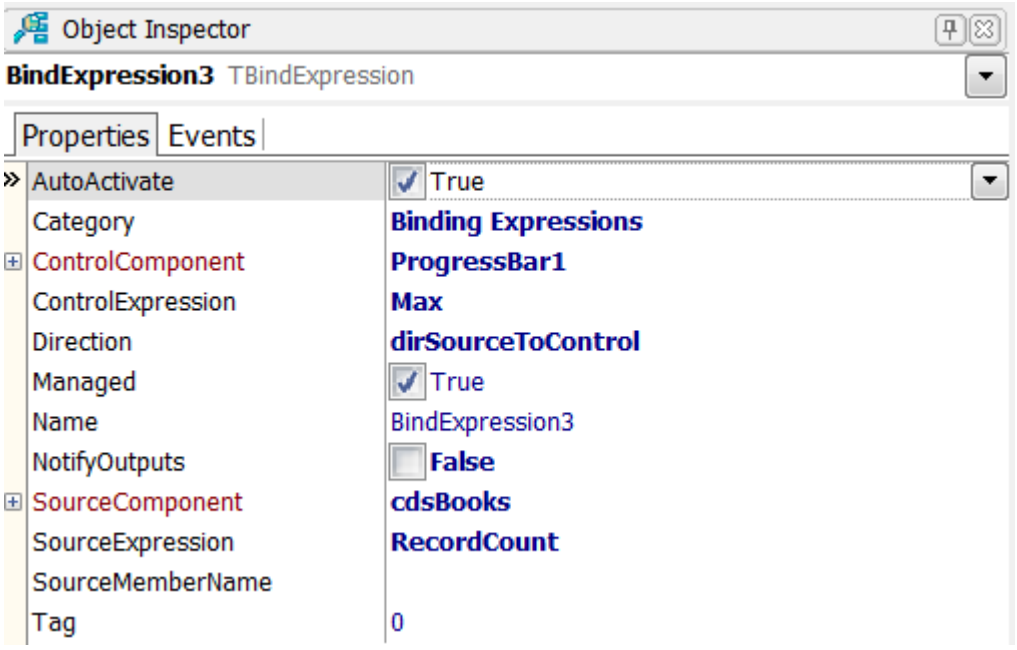
控制执行范围	控制系结表达式
ProgressBar1	Value

来源执行范围	来源系结表达式
cdsBooks	RecordCount

控制执行范围	控制系结表达式
ProgressBar1	Max

在下面的 2 个对象查看器中可以看到根据上面的系结表达式设定 TBindingsList 组件中第 2 和第 3 个 TBindExpression 对象:



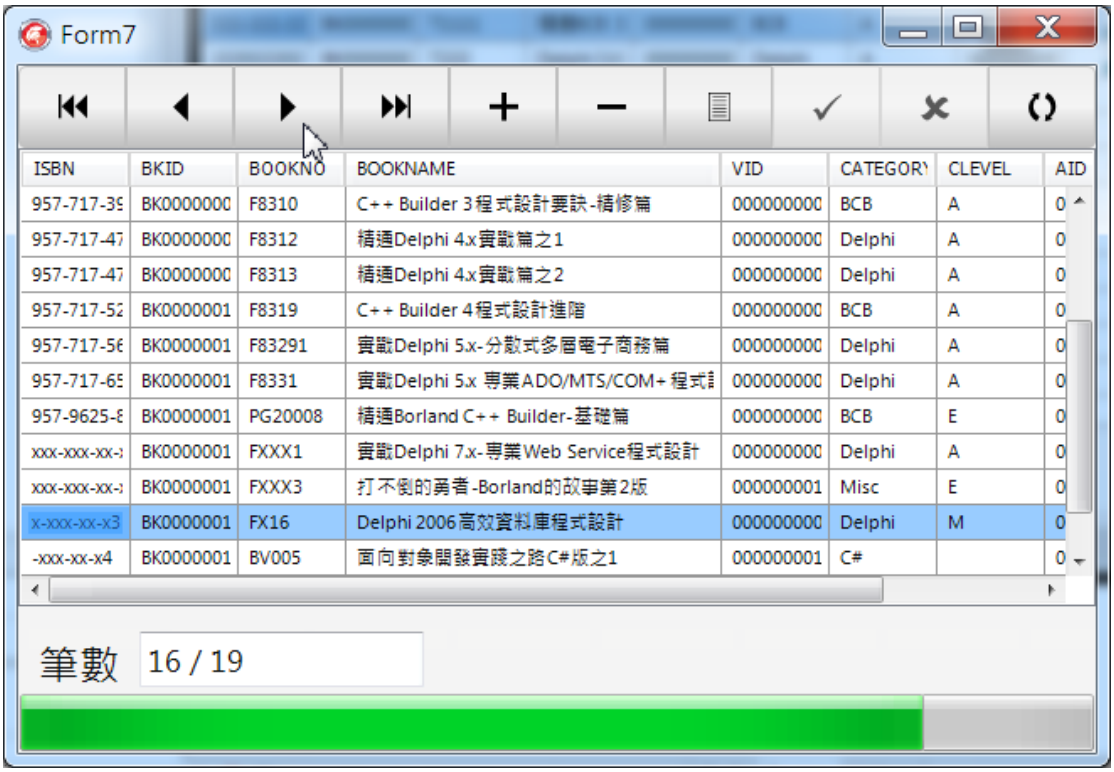


最后回到程序代码中在 cdsBooks 的 AfterOpen 和 AfterScroll 事件处理函式中呼叫 TBindExpression 对象的 Evaluate 方法要求系结引擎执行上述的系结表达式:

```
procedure TForm7.cdsBooksAfterOpen(DataSet: TDataSet);
begin
  BindExpression1.Evaluate;
```

```
  BindExpression3.Evaluate;
end;

procedure TForm7.cdsBooksAfterScroll(DataSet: TDataSet);
begin
  BindExpression1.Evaluate;
  BindExpression2.Evaluate;
end;
```

现在编译和执行此范例 FireMonkey 应用程序，从下面的画面中我们可以看到这 2 个系结表达式果然能够正确的执行了。



从上面的范例中可以看到我们印证了系结框架的基本概念，了解了上面范例和程序代码的意义后，读者应该已经掌握了系结框架的基本概念了，我们就可以再进一步的讨论稍微复杂的其他 2 种系结表达式了。

9-1-2 未拖管系结表达式

未拖管系结表达式是由应用程序所建立和拥有，当开发人员想执行未拖管系结表达式中的系结表达式时必须呼叫未拖管系结表达式对象的 Evaluate 方法。

未拖管系结表达式也包含了来源系结表达式和控制系结表达式，要建立未拖管系结表达式开发人员需要呼叫 TBindings 类别的 CreateUnmanagedBinding 类别方法，下面就是 CreateUnmanagedBinding 类别方法的宣告原型：

```
   class function CreateUnmanagedBinding(
     const InputScopes: array of IScope;
     const BindExprStr: string;
     const OutputScopes: array of IScope;
     const OutputExpr: string;
     const OutputConverter: IValueRefConverter;
     BindingEventRec: TBindingEventRec;
     Options: TCreateOptions = [coNotifyOutput]): TBindingExpression;
overload;
   class function CreateUnmanagedBinding(
     const InputScopes: array of IScope;
     const BindExprStr: string;
     const OutputScopes: array of IScope;
     const OutputExpr: string;
     const OutputConverter: IValueRefConverter;
     Options: TCreateOptions = []): TBindingExpression; overload;
```

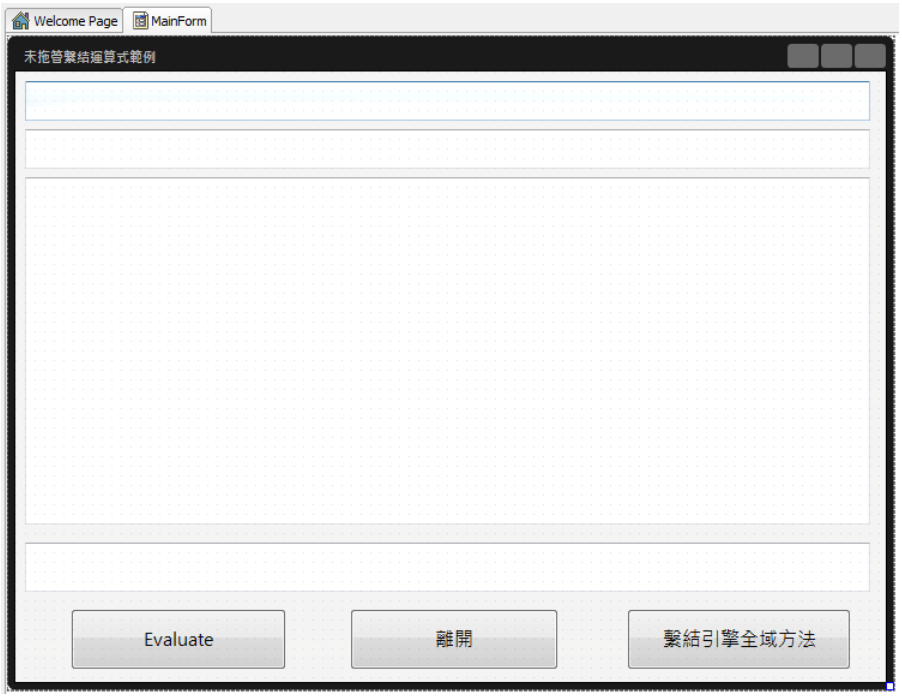
CreateUnmanagedBinding 有 2 个复载原型，分别接受不同的参数，下面的表格说明了这些参数的意义：

参数名称	参数说明
InputScopes	来源执行范围
BindExprStr	来源系结表达式
OutputScopes	控制执行范围
OutputExpr	控制系结表达式
OutputConverter	输出数据型态转换函式
BindingEventRec	系结事件处理函式
Options	影响系结表达式执行行为的选项

CreateUnmanagedBinding 类别方法回传 TBindingExpression 对象，这个对象就是未拖管系结对象，它可以使用来执行未拖管系结表达式。

输出数据型态转换函式和系结事件处理函式在稍后会说明，现在让我们使用一个范例来说明。

在 IDE 中建立一个 FireMonkey Desktop Application 项目，在主窗体中放入 3 个 TEdit 组件，一个 TMemo 组件和 3 个 TButton 组件，如下所示：



在这个范例中我们将系结 TStringList 对象和主窗体中下方的 TEdit 组件，而主窗体上方的 2 个 TEdit 组件则是用来输入来源系结表达式和控制系结表达式。

下面是主窗体中 Evaluate 按钮的 OnClick 的实作程序代码中:

```
001   procedure TForm7.ProcessExpressions;
002   var
003     LInputExpr: string;
004     LBindingExpression: TBindingExpression;
005     LSourceScope: IScope;
006     LControlScope: IScope;
007     LTestObject: TObject;
008   begin
009     LTestObject := TStringList.Create;
010     (LTestObject as TStringList).Add('实时数据系结');
011     (LTestObject as TStringList).Add('FM2');
012     (LTestObject as TStringList).Add('Mobile Studio');
013     try
014       LSourceScope := WrapObject(LTestObject);
015       LSourceScope := TNestedScope.Create(LSourceScope,
TBindingMethodsFactory.GetMethodScope);
016       LControlScope := WrapObject(edtControl);
017       LControlScope := TNestedScope.Create(LControlScope,
```

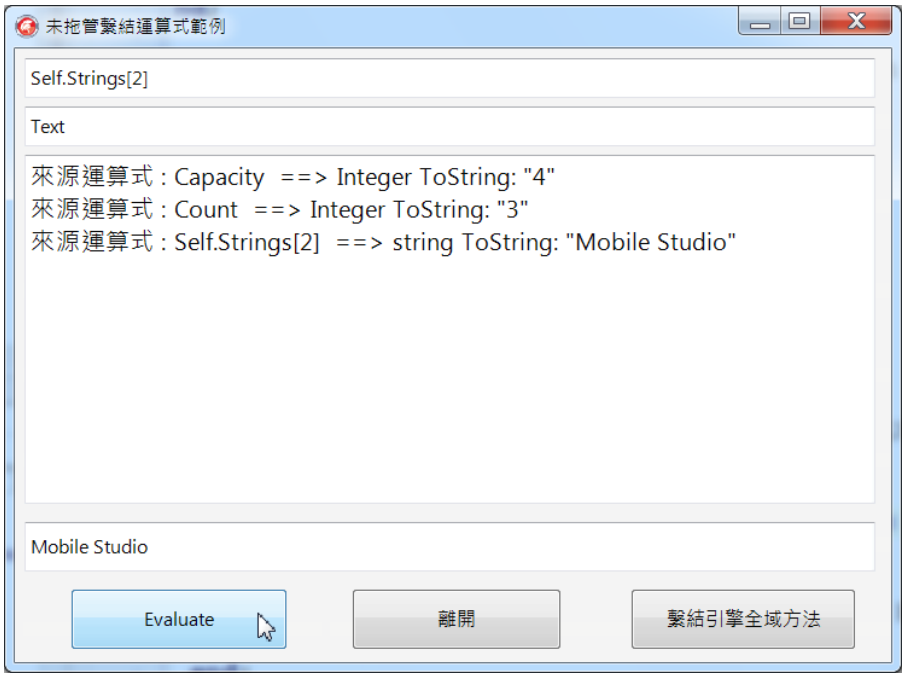
```
TBindingMethodsFactory.GetMethodScope);
018
019       LBindingExpression := TBindings.CreateUnManagedBinding(
020         LSourceScope,
021         edtSourceExpression.Text,
022         LControlScope,  // Output scope same as input
023         edtControlExpression.Text,
024         nil);  // Use default output converters
025       try
026         DisplayValue(edtSourceExpression.Text,
LBindingExpression.Evaluate);
027       finally
028         LBindingExpression.Free;
029       end;
030     finally
031       LSourceScope := nil;
032       LControlScope := nil;
033       LTestObject.Free;
034     end;
035   end;
```

下面的表格说明了上面程序代码的意义:

程序代码行数	程序代码说明
009	建立来源对象 TStringList
010~012	在来源对象 TStringList 中加入 3 个字符串数据以准备稍后展示用未拖管系结表达式来操作
014~015	建立来源执行范围，即 TStringList 对象
014~017	建立控制执行范围，即主窗体下方的 TEdit 组件
019~024	呼叫 TBindings 的 CreateUnManagedBinding 方法建立未拖管系结表达式对象，分别把来源执行范围，来源系结表达式，控制执行范围，控制系结表达式传入当参数。请注意最后一个参数是 Nil，代表使用系结引擎内定的输出数据型态转换函式
026	呼叫未拖管系结表达式对象的 Evaluate 方法执行系结表达式，并且藉由 DisplayValue 方法在主窗体中的 TMemo 组件显示执行信息

从上面的程序代码就可以知道，一旦在主窗体上方的 2 个 TEdit 组件中输入系结表达式之后，再点选下方的 Evaluate 按钮就会在主窗体下方的 TEdit 中显示系结的运算结果。

现在执行范例 FireMonkey 应用程序，在下面的画面中读者可以看到笔者在主窗体上方的 2 个 TEdit 组件中输入来源系结表达式和控制表达式之后，点选『Evaluate』按钮之后在 TMemo 和下方 TEdit 组件的执行结果：



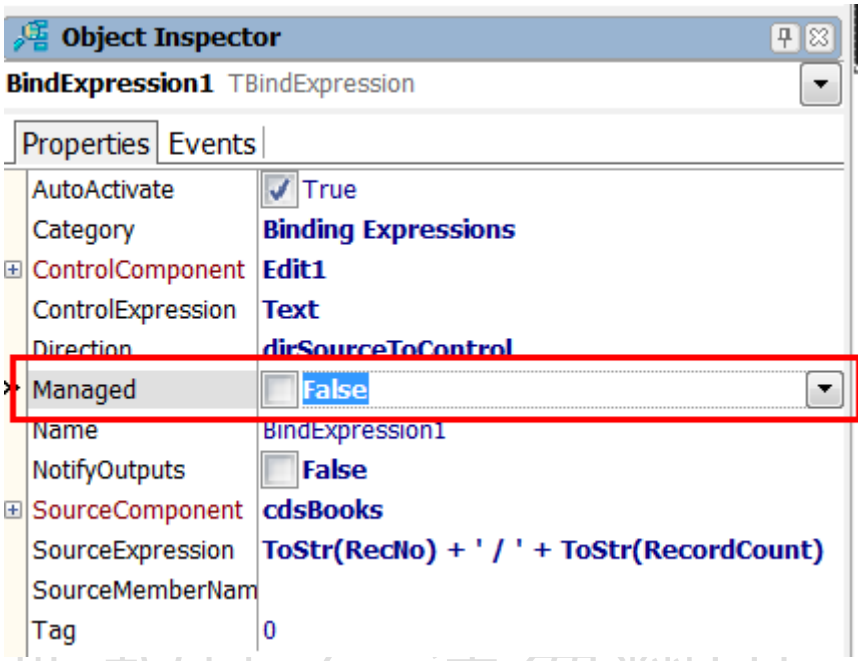
由于范例 FireMonkey 应用程序系结了 TStringList 和主窗体下方的 TEdit 组件，因此在点选『Evaluate』按钮之后系结表达式执行的结果会立刻自动出现在主窗体下方的 TEdit 组件中，下面的表格说明了执行的系结表达式以及执行的结果：

来源系结表达式	控制系结表达式	edtControl.Text
Capacity	Text	4
Count	Text	3
Self.Strings[2]	Text	Mobile Studiio

从这个范例我们也可以看到未拖管系结表达式也可以系结任何的对象和组件，非常具有弹性，此外从这个范例读者也可以了解在使用未拖管系结表达式时，执行范围也有 2 个，即来源执行范围和控制执行范围，来源系结表达式是在来源执行范围中执行而控制系结表达式是在控制执行范围中执行。

那么如果我们是直接在应用程序中使用系结框架的组件时要如何使用未拖管系结表达式呢?很简单，因为 Delphi 的系结组件都有一个『Managed』特性，它的内定值是 True，

代表 Delphi 的系结组件都是拖管系结表达式，如果开发人员要使用未拖管系结表达式，那么只需要取消这个特性即可。例如下图上面小节使用的范例 TBindExpression 组件，它的『Managed』特性在对象查看器中是勾选的，要让它成为未拖管系结表达式，只需要如下取消『Managed』特性即可。



9-1-3 拖管系结表达式

拖管系结表达式和未拖管系结表达式很类似，也拥有来源系结表达式和控制系结表达式，但这 2 者不同的地方是拖管系结表达式是由系结引擎所拥有，因此当系结表达式中有任何的系结组件或是系结表达式改变时，在系结引擎中相关的拖管系结表达式就会自动被重新执行，而无需由开发人员主动呼叫 Evaluate 方式。

这个意思是说，假设在系结引擎中拥有 2 个拖管系结表达式，这 2 个拖管系结表达式中都系结到了 A 组件的 B 特性，那么如果第二个拖管系结表达式改变了 A 组件的 B 特性值，那么系结引擎就会自动重新执行第一个拖管系结表达式。不过开发人员仍然可以呼叫拖管系结表达式的 Evaluate 方法主动要求执行拖管系结表达式。

要建立拖管系结表达式开发人员需要呼叫 TBindings 类别的 CreateManagedBinding 类别方法，下面就是 CreateManagedBinding 类别方法的宣告原型：

```
class function CreateManagedBinding(
  const InputScopes: array of IScope;
  const BindExprStr: string;
  const OutputScopes: array of IScope;
```

```
     const OutputExpr: string;
     const OutputConverter: IValueRefConverter;
     BindingEventRec: TBindingEventRec;
     Manager: TBindingManager = nil;
     Options: TCreateOptions = [coNotifyOutput]): TBindingExpression;
overload;
   class function CreateManagedBinding(
     const InputScopes: array of IScope;
     const BindExprStr: string;
     const OutputScopes: array of IScope;
     const OutputExpr: string;
     const OutputConverter: IValueRefConverter;
     Manager: TBindingManager = nil;
     Options: TCreateOptions = [coNotifyOutput]): TBindingExpression;
overload;
```

CreateManagedBinding 也有 2 个复载原型，而且其参数的意义和前面介绍 CreateUnmanagedBinding 方法的参数是一样的，而且 CreateManagedBinding 方法也是回传 TBindingExpression 对象，不过这个 TBindingExpression 物件是拖管系结物件。

Delphi 的系结组件在内定上都是属于拖管系结表达式，在稍后的章节中会有范例说明。

9-2 数据型态转换函式

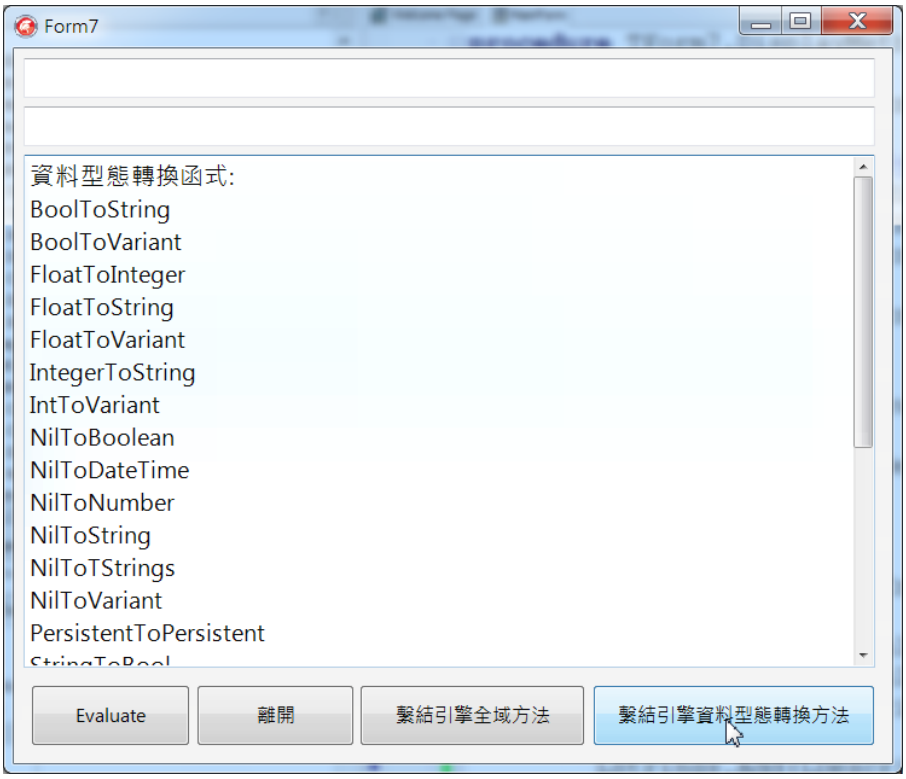
当开发人员使用系结引擎把来源表达式的执行结果指定给控制组件时，来源表达式的执行结果的数据型态可能和控制系结表达式执行结果的数据型态不同，在这种情形中就需要进行数据型态转换的工作。例如如果把 TStringList 对象的 Count 特性值系结到 TEdit 组件的 Text 特性，那么由于 Count 是整数值而 Text 是字符串型态，因此来源系结表达式必须把执行结果从整数型态转换为字符串型态。

在系结引擎中在来源方和控制方进行自动数据型态转换的函式就称为来『输出转换元(OutputConverters)』，基本上开发人员在使用系结表达式时并不需要特别的注意输出转换元，因为系结引擎会藉由 RTTI 自动判断是否需要呼叫特定的输出转换元来进行数据型态转换的工作。

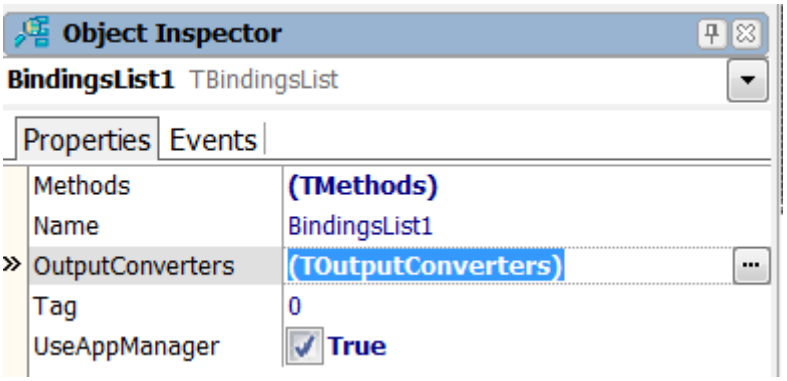
我们可以使用 TValueRefConverterFactory 类别的类别方法 GetConverterDescriptions 来取得目前系结引擎中所有的输出转换元，例如使用下面的程序代码就可以得到输出转换元的信息：

```
procedure TForm7.DisplayConverters;
var
  LDescription: TConverterDescription;
  LStrings: TStringList;
  iCount : Integer;
begin
  LStrings := TStringList.Create;
  try
    for LDescription in
TValueRefConverterFactory.GetConverterDescriptions do
     if LDescription.DefaultEnabled then
       LStrings.Add(LDescription.DisplayName)
     else
       LStrings.Add(LDescription.DisplayName + ' (disabled)');
    LStrings.Sort;

    Memo1.Lines.Add('数据型态转换函式: ');
    for iCount := 0 to LStrings.Count - 1 do
      Memo1.Lines.Add(LStrings[iCount]);
  finally
    LStrings.Free;
  end;
end;
```

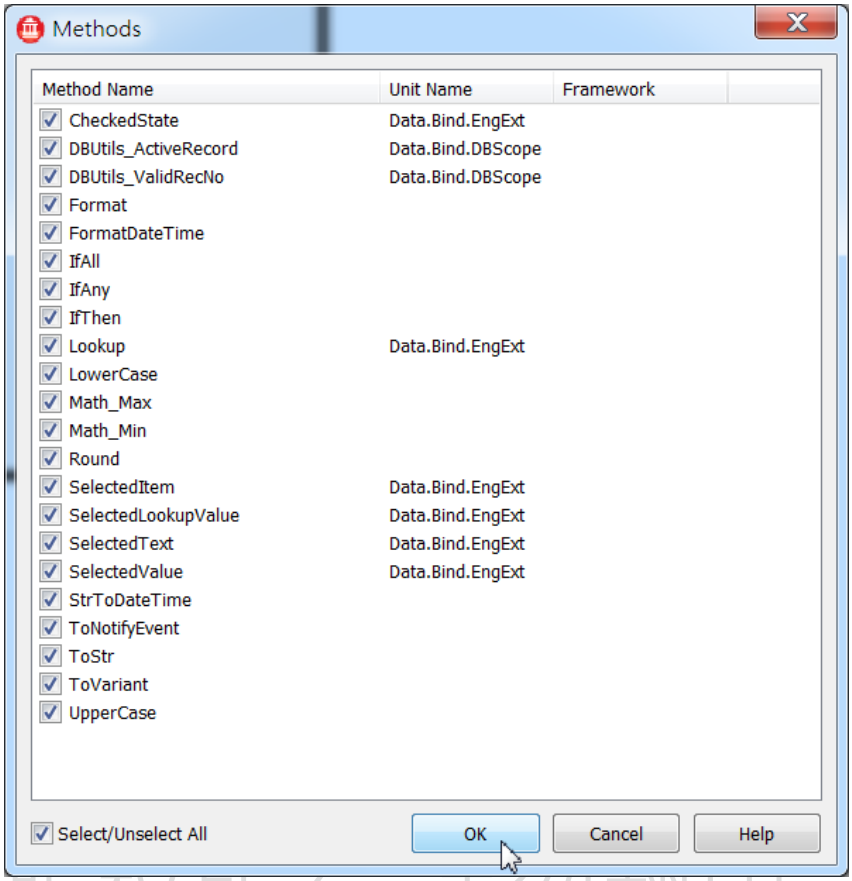
在下面的画面中读者可以看到系结引擎中所有的内建输出转换元：

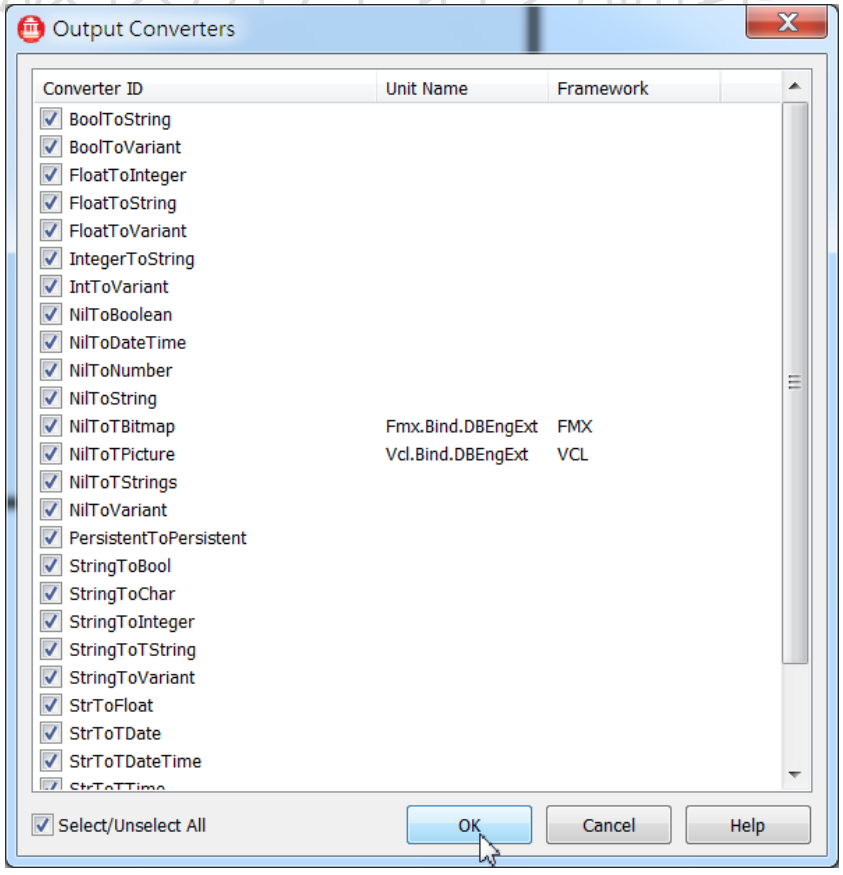


在 TBindingsList 组件中就拥有系结引擎中的全局方法和输出转换元，它们分别在 TBindingsList 组件的 Methods 和 OutputConverters 特性中:



如果读者双擊上面的 2 个特性值就可以看到下面的 2 个对话盒其中列出了目前系结引擎的全局方法和输出转换元:


Methods
Method Name
Unit Name
Framework
CheckedState
Data.Bind.EngExt
DBUtils_ActiveRecord
Data.Bind.DBScope
DBUtils_ValidRecNo
Data.Bind.DBScope
Format
FormatDateTime
IfAll
IfAny
IfThen
Lookup
Data.Bind.EngExt
LowerCase
Math_Max
Math_Min
Round
SelectedItem
Data.Bind.EngExt
SelectedLookupValue
Data.Bind.EngExt
SelectedText
Data.Bind.EngExt
SelectedValue
Data.Bind.EngExt
StrToDateTime
ToNotifyEvent
ToStr
ToVariant
UpperCase
Select/Unselect All
OK
Cancel
Help


Output Converters
Converter ID
Unit Name
Framework
BoolToString
BoolToVariant
FloatToInteger
FloatToString
FloatToVariant
IntegerToString
IntToVariant
NilToBoolean
NilToDateTime
NilToNumber
NilToString
NilToTBitmap
Fmx.Bind.DBEngExt
FMX
NilToTPicture
Vcl.Bind.DBEngExt
VCL
NilToTStrings
NilToVariant
PersistentToPersistent
StringToBool
StringToChar
StringToInteger
StringToTString
StringToVariant
StrToFloat
StrToTDate
StrToTDateTime
Select/Unselect All
OK
Cancel
Help

如果开发人员需要特别的输出转换元来进行数据型态转换，那么开发人员也可以向系结引擎注册客制化的输出转换元。

9-3 实时数据系结相关的类别

DX10 的实时数据系结框架提供了许多不同的实时数据系结类别以提供不同的服务，开发人员在使用实时数据系结技术时可以根据需求选择使用不同的实时数据系结类别，在前面讨论中的内容中本书已经介绍过数个不同的实时数据系结类别，例如 TBindExpression 下面的表格说明了每一个实时数据系结类别的功能:

类别	说明
TBindExpression	系结单一来源对象到控制对象的类别，可提供双向系结，或是单向系结
TBindExprItems	系结单一来源对象到控制对象的类别，其中可包含数个不同的函数式，分别储存在其 FormatExpressions 和 ClearExprerssions 特性中
TBindLink	系结单一来源对象到控制对象的类别，其中可包含数个不同的函数式，分别储存在其 FormatExpressions，ParseExpressions 和 ClearExprerssions 特性值中。通常是使用在数据库相关的系结。
TBindListLink	系结单一来源对象到串行型态的控制对象的类别，其中可包含数个不同的函数式，分别储存在其 FormatExpressions，FormatControlExpressions，ParseExpressions 和 ClearExprerssions，ClearControlExprerssions 特性值中。通常是使用在数据库相关的系结。
TBindGridLink	系结 Grid 组件到数据源的类别，例如系结 FireMonkey 的 TStringGrid 或是 VCL 的 TStringGrid 组件到数据源。通常是使用在数据库相关的系结。
TBindPosition	系结具有位置相关特性的组件，例如系结 TScrollBar，TProgressBar 和 TTrackBar 等组件和数据源。
TBindControlValue	系结控制对象到组件的特性，当控制对象被用户更新后系结的组件特性也会自动更新
TBindList	系结单一来源对象到串行型态的控制对象的类别，例如系结数据结构中的数据到 TListBox 等应用。
TBindGridList	系结单一来源对象到串行型态的控制对象的类别，例如系

	结数据结构中的数据到 TStringGrid 等应用

除了这些 XE2 就出现的系结类别之外，DX10 又增加了『Quick Bindings』类别，这些新的快速系结类别是为了和可视化实时数据系结设计家结合以提供开发人员快速开发数据库相关的系结工作，因此原先在 XE2 中的『DB Links』系结类别就宣告为过时的系结类别，从 DX10 之后开发人员应该这些新的快速系结类别来系结数据源。下面的表格说明了这些新的快速系结类别的意义:

类别	说明
TLinkGridToDataSource	系结 TGrid 或是 TStringGrid 组件到数据源，因此来源组件是 TGrid 或是 TStringGrid 组件，控制组件是数据源
TLinkListControlToField	系结串行组件到数据集中的字段，例如系结 TListBox 等串行到 TClientDataSet 的字段。来源组件是串行组件，控制组件是字段对象
TLinkControlToField	绑定控件到字段，例如系结 TEdit，TComboBox 到 TClientDataSet 的字段。来源组件是控件，控制组件是字段对象
TLinkControToProperty	系结结控件到某组件的特性，例如系结 TEdit 到 TButton 的 Text 特性，来源组件是控件，控制组件是组件的特性
TLinkPropertyToField	系结组件的特性到字段，例如系结 TEdit 的 Text 特性值到 TClientDataSet 的字段，来源组件是组件的特性值，控制组件是字段对象
TLinkFillControlToField	系结需要填入数据的组件到数据集中的字段，例如系结 TComboBox 到 TClientDataSet 的字段。来源组件是填入数据控制项，控制组件是字段对象
TLinkFillControlToProperty	系结需要填入数据的组件到组件的特性，例如系结 TComboBox 到 TEdit 的 Text 特性。来源组件是填入数据控制项，控制组件是组件的特性

让我们继续使用 BOOKS 数据表来说明如何使用这些系结类别读者就可以很容易的了解了。

9-3-1 使用 TBindExprItems 类别

TBindExprItems 系结类别可同时包含数个系结表达式，因此我们可以把它看成是多个 TBindExpression 类别对象的集合类别，开发人员藉由在它的 Format 特性中撰写任意数目的系结表达式，也可以在它的 Clear 特性中撰写清除系结表达式，但 TBindExprItems 仍然是系结来源组件和控制组件，因此这些多个系结表达式仍然是执行在这 2 方的执行范围中。

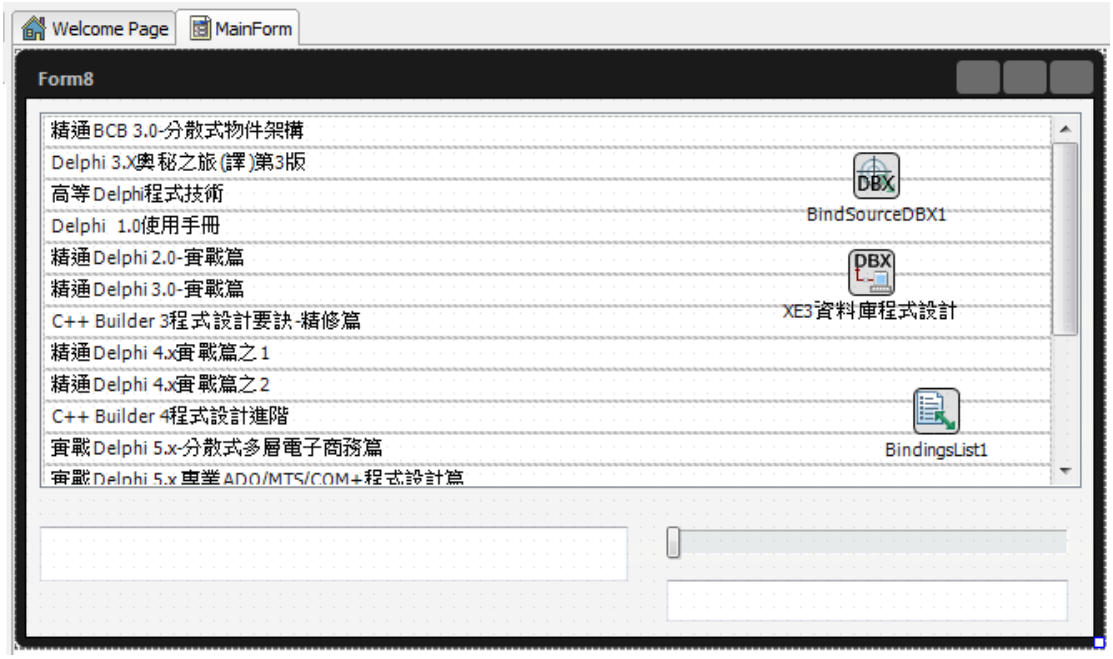
例如如果我们想系结 TTrackBar 组件和数据源，那么我们通常需要把 0 系结到 TTrackBar 组件的 Min 特性值，数据源的 RecordCount 系结到 TTrackBar 组件的 Max 特性值，再把数据源的 RecNo 系结到 TTrackBar 组件的 Value 特性值，如此一来当我们卷动 TTrackBar 组件时就可以在数据源中不同的数据中浏览。在上面的说明中有 3 个系结的工作，我们可以使用 3 个 TBindExpresssion 对象来分别系结，但我们也可以使用一个 TBindExprItems 对象来撰写和执行这 3 个系结工作。

现在就让我们使用一个范例来说明 TBindExprItems 系结类别以及拖管系结表达式的意义。

先在 IDE 中建立一个 FireMonkey Desktop Application 项目，在主窗体中拖曳 Data Explorer 中的『XE7 数据库程序设计』节点到主窗体中以建立 TSQLConnectin 组件，再放入 TBindSourceDBX 组件，设定它的特性如下:

特性名称	特性值
SQLConnection	XE7 数据库程序设计
CommandText	Select * from BOOKS
CommandType	Dbx.SQL

再放入一个 TListBox 组件，2 个 TEdit 组件和一个 TTrackBar 组件，最后使用可视化实时数据系结设计家系结 TBindSourceDBX 组件的 BOOKNAME 字段到 TListBox 的 Text 特性，现在主窗体如下所示:



在这个范例中我们将使用系结类别来完成下列的工作:

1. 点选 TListBox 中的项目时，此项目会自动出现在主窗体左下方的 TEdit 组件中

2. 系结主窗体右下方的 TTrackBar 组件和 TListBox 组件，因此当拖曳主窗体加下方的 TTrackBar 组件时，TListBox 中的项目也会自动根据 TTrackBar 的位置卷动到相对的项目

3. 当拖曳 TTrackBar 时，TTrackBar 的位置数值会自动出现在主窗体右下方的 TEdit 组件中

4. 当输入数值到主窗体右下方的 TEdit 组件中时，也会改变 TTrackBar 的位置

上面的工作看起来很多，但使用系结表达式却可以轻易的完成这些工作，而且由于稍后我们会使用拖管系结表达式，因此会产生连动的效果。现在让我们一步一步的使用系结表达式完成上面的 4 个工作。

步骤 1-系结 TListBox 和 TEdit 组件

由于这只需要一个系结表达式就可以完成，因此我们只需要使用 TBindExpression 系结对象即可，这个系结也很简单，读者现在应该可以非常容易的完成它。

由于这只需要一个系结表达式就可以完成，因此我们只需要使用 TBindExpression 系结对象即可，这个系结也很简单，读者现在应该可以非常容易的完成它，我们只需要把 TListBox 中目前被选择的项目系结到 TEdit 的 Text 特性即可，因此我们可以得到下面的系结表达式:

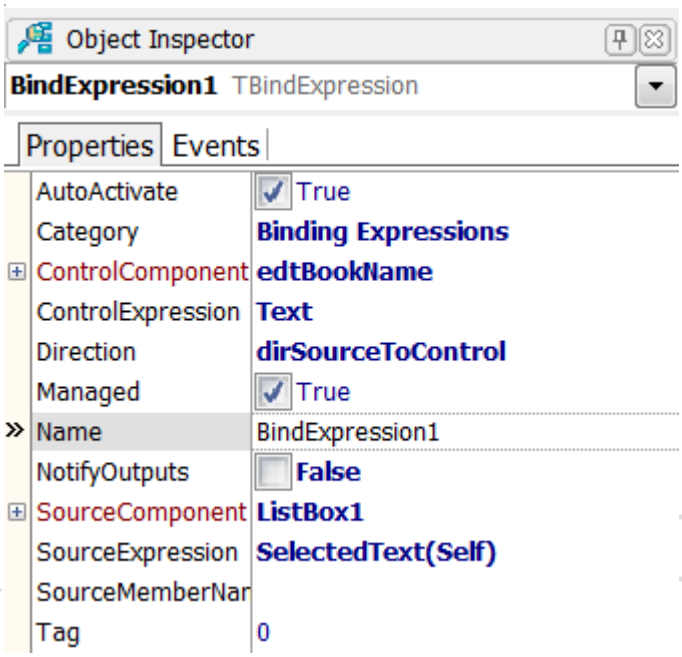
系结表达式	设定值
来源组件	TListBox
来源系结表达式	**?**
控制组件	TEdit
控制系结表达式	Text

上面唯一要确定的是如何使用系结表达式取得 TListBox 组件中目前被选择的项目呢? 这很简单，因为在系结引擎中有一个全局函式:SelectedText，它可以从串行组件中取得目前被选择的项目，因此最后的系结表达式是:

系结表达式	设定值
来源组件	TListBox
来源系结表达式	SelectedText(Self)
控制组件	TEdit
控制系结表达式	Text

上面的来源系结表达式中的 Self 就是来源系结表达式的执行范围，它当然就是来源组件 TListBox 了，因此 SelectedText(Self)就等于 SelectedText(ListBox1)，但在系结表达式中不可以使用 SelectedText(ListBox1)，因为 ListBox1 是 Delphi 的对象变量名称，系结表达式并无法存取 ListBox1，因此必须使用 SelectedText(Self)。

因此现在请在主窗体中的 TBindingsList 中新增一个 TBindExpression 组件，并且如下面的对象查看器一样设定我们上面的系结表达式。



接着我们当然需要呼叫 TBindExpression 对象的 Evaluate 方法才能执行上面的系结表达式，因此请在 TListBox 的 OnChange 事件处理函式中撰写如下的程序代码:

```
procedure TForm8.ListBox1Change(Sender: TObject);
begin
  BindExpression1.Evaluate;
end;
```

现在如果您执行此范例应用程序那么当您点选 TListBox 中的项目时它就会自动出现在主窗体左下方的 TEdit 组件中了。

步骤 2-系结 TListBox 和 TTrackBar 组件

要系结 TListBox 和 TTrackBar 需要数个系结表达式同时运作，因为我们需要:

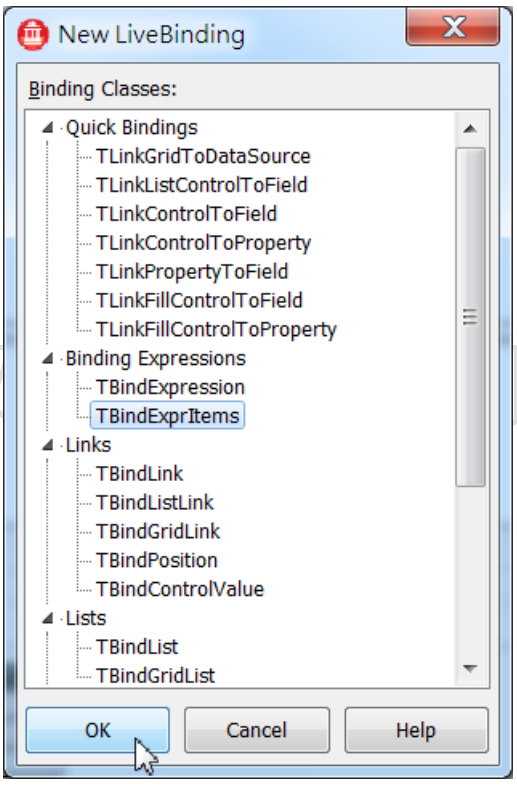
1. 把 TListBox 的 Count - 1 特性值系结到 TTrackBar 的 Max 特性
2. 把 TListBox 的 0 特性值系结到 TTrackBar 的 Min 特性

3. 把 TListBox 的 ItemIndex 系结到 TTrackBar 的 Value 特性

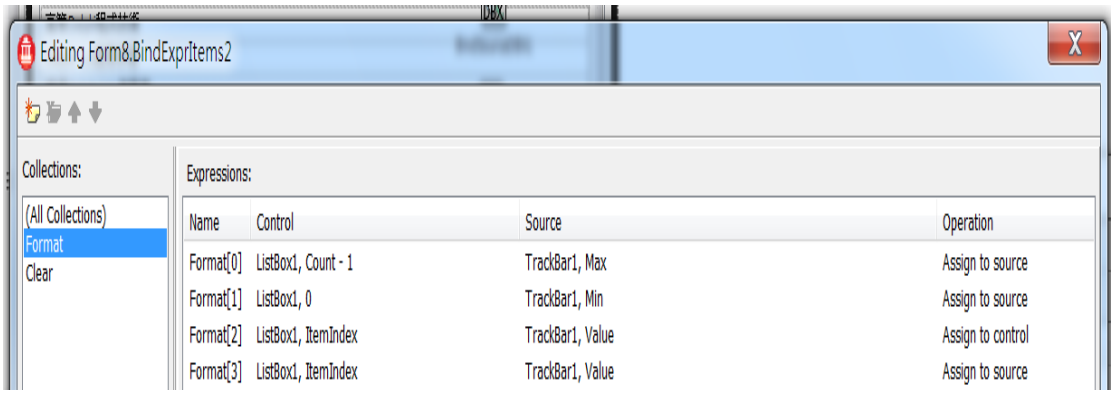
4. 把 TTrackBar 的 Value 特性系结到 TListBox 的 ItemIndex 系结

一旦完成了上面的第 3,4 步骤之后 TListBox 和 TTrackBar 就可以连动了，拖曳 TTrackBar 可以改变 TListBox 中目前的选择项目，而改变 TListBox 中目前的选择项目则可以连动改变 TTrackBar 的拖曳点(即 Value 特性值)。

由于需要同时系结上述的工作，因此我们可以建立一个 TBindExprItems 对象并且在它的 Format 特性中同时输入上面的系结表达式。因此请在主窗体的 TBindingsList 组件中建立一个 TBindExprItems 对象，如下所示:



接着双击 TBindingsList 组件编辑器中新建立的 TBindExprItems 对象开启 TBindExprItems 对象的组件编辑器，点选左上方的『Add Expression』按钮以新增系结表达式对象，请在其中建立 4 个系结表达式，然后分别撰写如下的系结表达式以系结 TListBox 和 TTrackBar:



下面的表格说明了上面 4 个系结表达式的设定：

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Max
控制组件	ListBox1
控制系结表达式	Count – 1
方向	Assign to Source，即把控制系结表达式的执行结果指定给来源组件
系结表达式意义	设定 TrackBar1 的最大值为 ListBox1 中的项目总数

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Min
控制组件	TListBox
控制系结表达式	0
方向	Assign to Source，即把控制系结表达式的执行结果指定给来源组件
系结表达式意义	设定 TrackBar1 的最小值为 0

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Value
控制组件	ListBox1

控制系结表达式	ItemIndex
方向	Assign to Control，即把来源系结表达式的执行结果指定给控制组件
系结表达式意义	设定 TrackBar1 的 Value 特性值指定给 ListBox1 的 ItemIndex 特性，这代表拖曳 TrackBar1 就会改变 ListBox1 中目前被选择的项目

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Value
控制组件	ListBox1
控制系结表达式	ItemIndex
方向	Assign to Source，即把控制系结表达式的执行结果指定给来源组件
系结表达式意义	设定 ListBox1 的 ItemIndex 特性值指定给 TrackBar1 的 Value 特性，这代表改变 ListBox1 中目前被选择的项目就会改变 TrackBar1 的拖曳位置

完成了上面的系结表达式之后同样的要藉由 Evaluate 方法执行，不过由于在上面有数个系结表达式，因此我们可以呼叫 TBindingsList 的 Notify 方法来通知系结引擎执行系结表达式。TBindingsList 的 Notify 方法接受 2 个参数，第 1 个是改变系结条件的对象，第 2 个参数则是改变系结条件的特性名称，它的原型如下:

```
procedure TCustomBindingsList.Notify(const AObject: TObject;
  const AProperty: string);
```

由于在上面的 4 个系结表达式中会有 2 个对象可能改变系结条件，那就是 ListBox1 和 TrackBar1，而可能改变系结条件的特性则有 TracBar1 的 Min，Max 和 Value 以及 ListBox1 的 ItemIndex 特性，因此我们可以传递空字符串给 Notify 方法的第 2 个参数代表第 1 个参数对象所有的特性都改变了因此要求系结引擎重新执行所有和第 1 个参数对象相关的系结表达式。

因此我们需要在 TrackBar1 的 OnChange 事件处理函式中撰写如下的程序代码，代表 TrackBar1 要求系结引擎重新执行所有和 TrackBar1 相关的系结表达式。

```
procedure TForm8.TrackBar1Change(Sender: TObject);
begin
```

```
  BindingsList1.Notify(TrackBar1, '');
end;
```

BindingsList1.Notify(TrackBar1, '')代表 TrackBar1 要求系结引擎重新计算和执行所有系结到 TrackBar1 的任何特性相关的系结表达式，在执行了此行程序代码之后，前面讨论的所有和 TrackBar1 相关的系结表达式都会被重新执行。

另外我们也需要要求系结引擎重新计算和执行所有系结到 ListBox1 的任何特性相关的系结表达式，因此请在 ListBox1 的 OnChange 事件处理函式中修改原先的程序代码如下:

现在我们也只需要呼叫 BindingsList1.Notify(ListBox1, '') 即可让所有和 ListBox1 相关的系结表达式都能够重新执行，因此我们不再需要单独呼叫 BindExpression1.Evaluate 方法了:

```
procedure TForm8.ListBox1Change(Sender: TObject);
begin
//  BindExpression1.Evaluate;
  BindingsList1.Notify(ListBox1, '');
end;
```


现在如果执行范例程序的话请读者注意在拖曳 TrackBar1 时除了 ListBox1 中目前被选择的项目也会改变之外，主窗体左下方的 TEdit 组件的内容也会改变，为什么？这当然就是因为 TrackBar1 系结了 ListBox1，而 ListBox1 又系结了左下方的 TEdit 组件，而这 2 个系结运对象(TBindExpression 和 TBindExprItems 对象)都是拖管系结对象，因此当上面 ListBox1Change 事件处理函式藉由 Notify 方法通知系结引擎 ListBox1 的系结条件改变时，系结引擎会自动重新执行所有和 ListBox1 相关的系结表达式，因此 BindExpression1 就会被系结引擎重新执行，因此左下方的 TEdit 组件就会重新显示 ListBox1 中目前被选择的项目内容，这种会被系结引擎自动根据系结条件变化而重新执行系结表达式的能力就是拖管系结表达式的特性，未拖管系结表达式则没有这种特性。

步骤 3-系结 TTrackBar 和 TEdit 组件

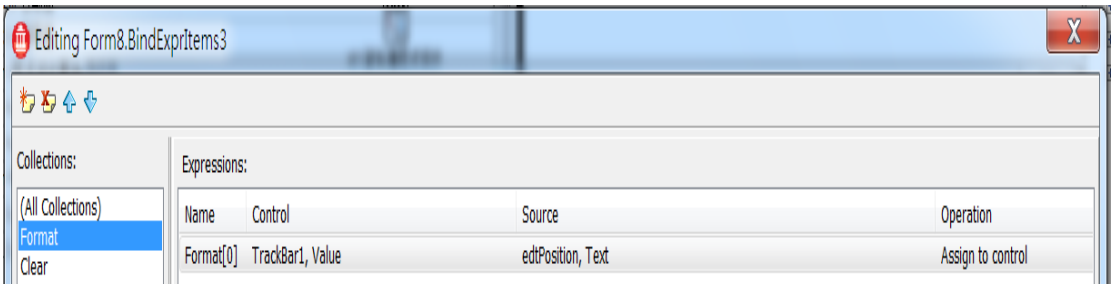
要在拖曳 TTrackBar 时，把 TTrackBar 的位置数值会自动显示在主窗体右下方的 TEdit 组件中就非常的容易了，我们只需要使用下面的系结表达式即可:

下面的表格说明了要完成这个工作我们只需要把 TrackBar1 的 Value 特性值系结到主窗体右下方的 TEdit 组件 edtPosition 的 Text 特性即可:

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Value
控制组件	edtPosition
控制系结表达式	Text
方向	Assign to Control，即把来源系结表达式的执行结果指定给控制组件

步骤 4-系结 TEdit 和 TTrackBar 组件

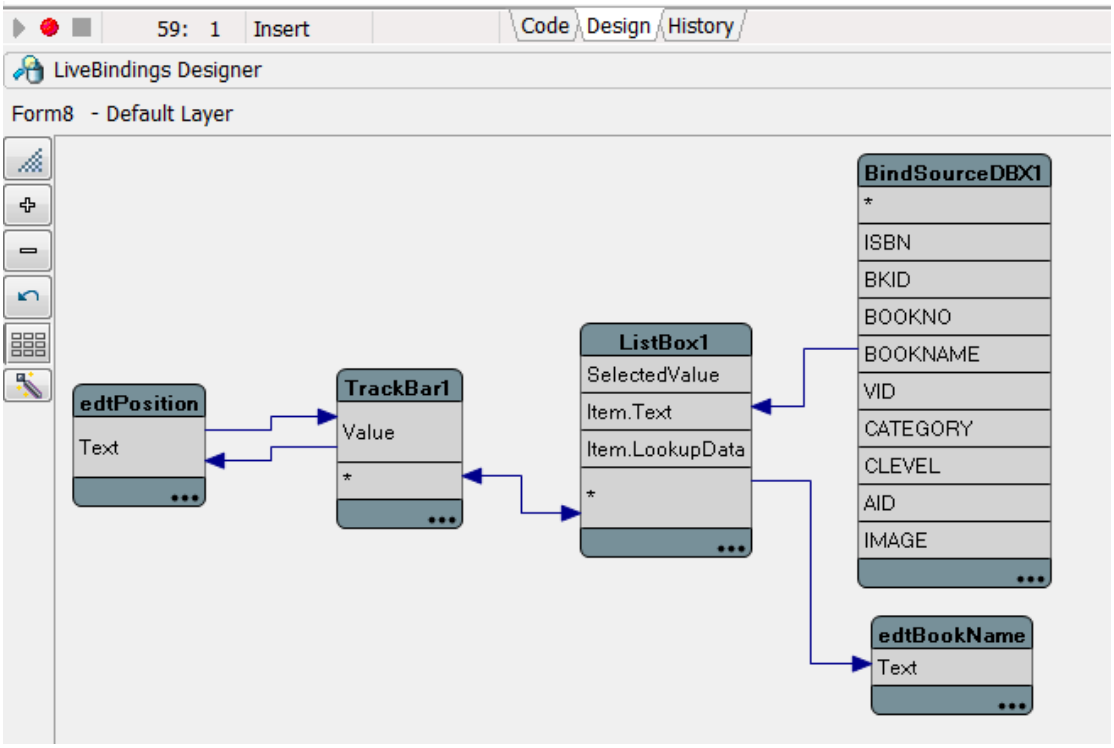
最后一个系结工作也正好是步骤 3 的反方向，我们只需把 edtPosition 的 Text 特性系结到 TrackBar1 的 Value 特性即可，因此我们只需使用下面的系结表达式：



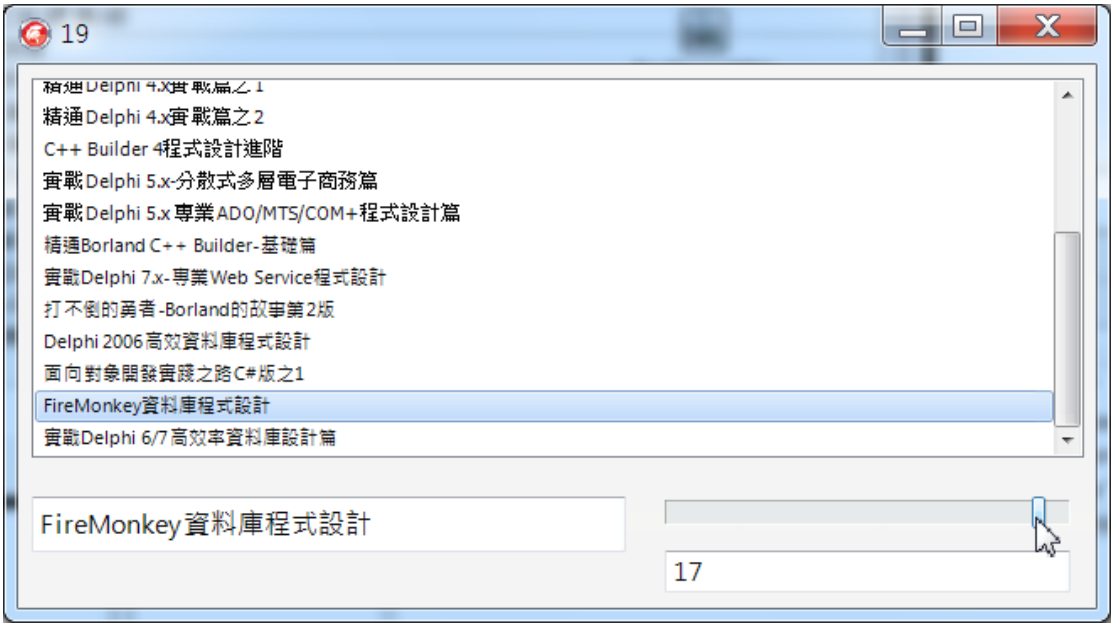
下面的表格说明了要完成这个工作我们只需要把 edtPosition 的 Text 特性值系结到 TrackBar1 的 Value 特性即可:

系结表达式	设定值
来源组件	TrackBar1
来源系结表达式	Value
控制组件	edtPosition
控制系结表达式	Text
方向	Assign to Source，即把控制系结表达式的执行结果指定给来源组件

完成了这些系结表达式之后如果读者现在开启可视化实时数据系结设计家，那么就会看到类似如下的结果，上面的系结表达式虽然无法使用实时数据系结设计家来设计和撰写(因为实时数据系结设计家主要是使用快速系结类别对象)，但这些由开发人员撰写的系结表达式仍然能够展现在实时数据系结设计家中。

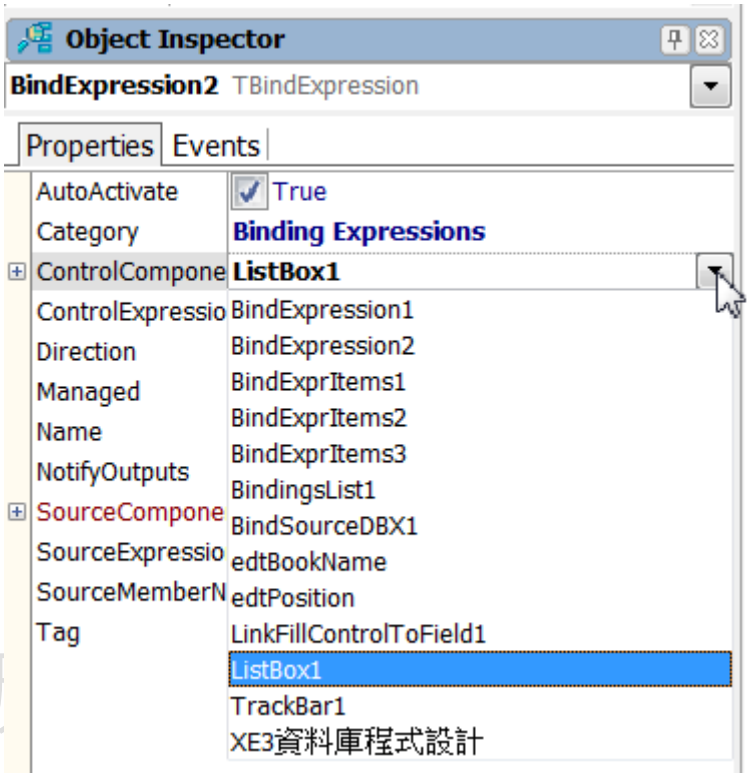


现在请编译和执行范例 FireMonkey 应用程序，如果拖曳 TTrackBar，点选 TListBox 或是在主窗体左下方的 TEdit 中输入数值都会发现主窗体中所有的组件都可连动显示，如下图所示：

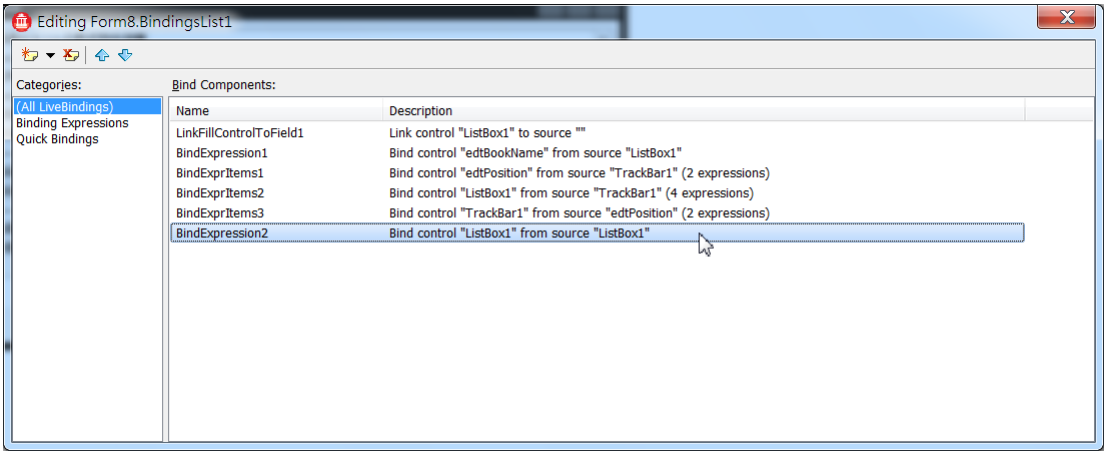


在离开本小节之前再让我们撰写一个系结表达式让读者了解如何在系结表达式中藉由执行范围存取其他的组件。假设现在我们希望在执行此范例 FireMonkey 应用程序时，当拖曳 TTrackBar 时，我们也希望 ListBox1 中目前被选择的项目也能够出现在主窗体左上方的表头中，那么我们应该如何撰写这个系结表达式呢？

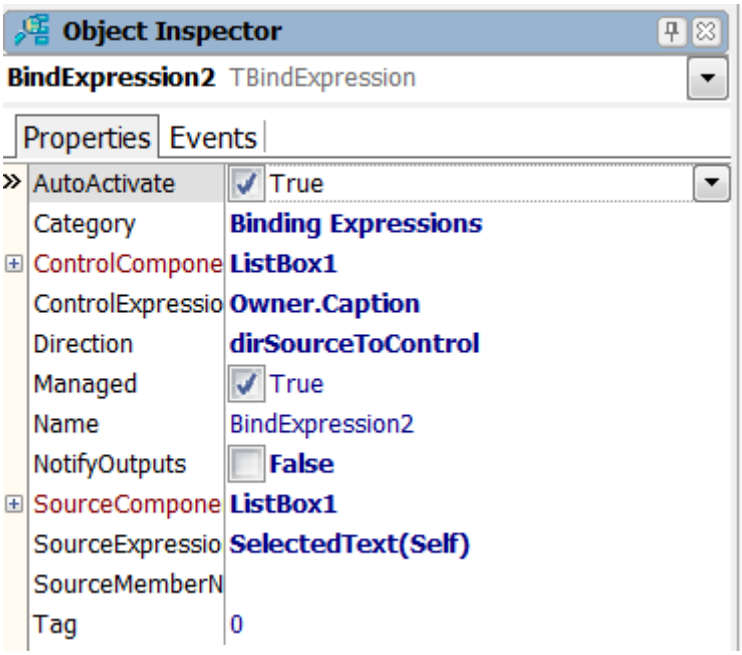
OK，先让我们找出来源组件和控制组件，它们分别应该是 ListBox1 和主窗体，但如果我们在 TBindingsList 中建立一个 TBindExpression 系结对象，然后在对象查看器中点选 ControlComponent 特性，在它的下拉盒中却无法找到主窗体对象，如下所示，那么我们如何把主窗体对象设定为控制组件呢？



答案是不能直接设定主窗体为控制组件，因为目前的实时数据系结只适用于主窗体中的组件，那么现在我们要如何解决这个系结工作呢？很简单，我们只要把控制组件设定为 ListBox1 的 Owner 不就可以了吗？因为 ListBox1 的 Owner 就是主窗体对象。因此让我们先在 TBindingsList 中建立一个 TBindExpression 系结对象，如下所示：



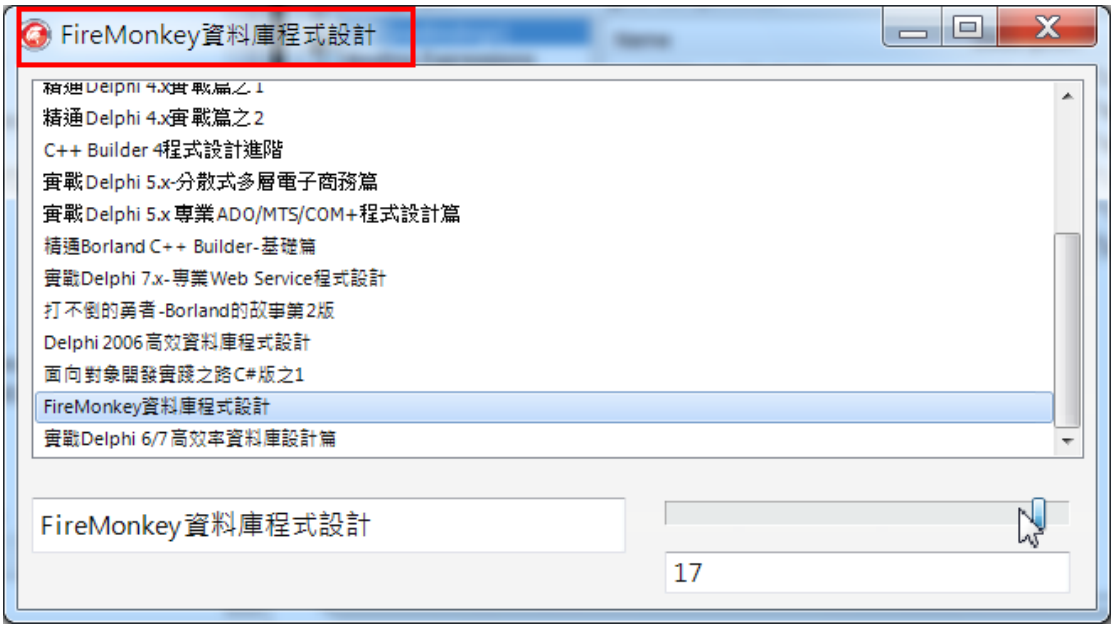
然后在对象查看器中设定如下的系结表达式：



系结表达式	设定值
来源组件	ListBox1
来源系结表达式	SelectedText(Self)
控制组件	ListBox1
控制系结表达式	**Owner**.Caption
方向	Assign to Control，即把控制系结表达式的执行结果指定给来源组件

在上面的系结表达式中我们把来源和控制组件都设定为 ListBox1，但在控制系结表达式中先使用关键词 Owner 来取得 ListBox1 的 Owner(即主窗体)，再系结到它的 Text 特性即可。

现在再次编译和执行范例 FireMonkey 应用程序，拖曳主窗体中的 TrackBar1 时 ListBox1 中目前被选择的项目也会自动出现在主窗体左上方的表头中了。



这个范例说明了在系结表达式中开发人员可以使用 Owner 关键词来存取目前执行范围对象的父代对象，这是很有用的技巧之一。

9-4 TBindingsList 提供的可呼叫方法

除了函数式之外，TBindingsList 也提供的一些方法允许开发人员使用在函数式中，例如在前面我们已经看过许多次的 ToStr，SelectedText(Self)等。这些可使用于函数式中的方法可以藉由在对象查看器中点选 TBindingsList 的 Methods 特性找到，在前面的章节中已经说明过。

这些方法在实时数据系结框架中是使用如下的程序代码注册并且提供给开发人员使用:

```
const
  sIDToStr = 'ToStr';
  sIDToVariant = 'ToVariant';
  sIDFormat = 'Format';
  sIDUpperCase = 'UpperCase';
  sIDLowerCase = 'LowerCase';
  ...

procedure RegisterBasicMethods;
begin
  TBindingMethodsFactory.RegisterMethod(
```

```
    TMethodDescription.Create(
      MakeMethodFormat,
      sIDFormat,
      sIDFormat, '', True,
      sFormatDesc,
      nil));
  TBindingMethodsFactory.RegisterMethod(
    TMethodDescription.Create(
      MakeMethodLowerCase,
      sIDLowerCase,
      sIDLowerCase, '', True,
      sLowerCaseDesc,
      nil));
  TBindingMethodsFactory.RegisterMethod(
    TMethodDescription.Create(
      MakeMethodUpperCase,
      sIDUpperCase,
      sIDUpperCase, '', True,
      sUpperCaseDesc,
      nil));
  TBindingMethodsFactory.RegisterMethod(
    TMethodDescription.Create(
      MakeMethodToStr,
      sIDToStr,
      sIDToStr, '', True,
      sToStrDesc,
      nil));
  TBindingMethodsFactory.RegisterMethod(
    TMethodDescription.Create(
      MakeMethodToVariant,
      sIDToVariant,
      sIDToVariant, '', True,
      sToVariantDesc,
      nil));
end;
....
```

当然，开发人员也可以实作自己的方法，向实时数据系结框架注册之后就可以使用在系结表达式中。下面的表格说明了这些方法的应用:

方法	说明
CheckedState(Self)	系结布尔值和 TCheckedBox 的方法，例如开发人员可以系结数据表布尔值字段和 TCheckedBox
Format	提供在函数式中呼叫 RTL 的 Format 方法
LowerCase	把字符串转换为小写型态
SelectedItem	存取目前系结的组件中的项目，例如存取目前 TListBox 或是 TGridBox 中被选择的项目
SelectedText	存取目前系结的组件中被选择的项目的 Text 特性值，例如存取目前 TListBox 或是 TGridBox 中被选择的项目的 Text 特性值
ToStr	把函数式中的运算结果转换为字符串型态，例如把函数式中的整数或是浮点数转换为字符串
ToVariant	把函数式中的运算结果转换为 Variant 型态
UpperCase	把字符串转换为大写型态
IfThen	接受 3 个参数，第 1 个是布尔值型态的函式参数，如果第 1 个参数是 True 的话就回传第 2 个参数值，如果第 1 个参数是 False 的话就回传第 3 个参数值。
IfAny	接受最多 100 个布尔值型态的函式参数，只要有任何的函式参数的结果值是 True 的话，就回传 True
IfAll	接受最多 100 个布尔值型态的函式参数，只要有任何的函式参数的结果值是 False 的话，就回传 False，只有所有的函式参数都是 True 才会回传 True
DButils_Activerecord	取得第 1 个参数(TDataSet 型态)的当前记录值
DBUtils_ValidRecNo	设定 1 个参数(TDataSet 型态)的当前记录值为第 2 个参数的数值
FormatDateTime	格式化 TDateTime 型态的参数并回传结果
Lookup	根据执行范例以(键值域名，键值域值，回传域名) 的参数信息查询回传域值
Math_Max	接受 2 个整数型态的参数，并且回传拥有最大整数数值的参数
Math_Min	接受 2 个整数型态的参数，并且回传拥有最小整数数值的参数
Round	四舍五入参数值并回传结果值
SelectedLookupValue	从串行型态的组件中回传查询的数值
SelectedValue	从串行型态的组件中回传目前版选择的数值

StrToDateTime	转换字串为 TDateTime 型态的数值
ToNotifyEvent	呼叫事件处理函式

DX10 的系结引擎的全局方法比起 XE2 的系结引擎的全局方法大幅增加了许多，让开发人员可以在系结表达式中进行更复杂的控制，在本书的许多内容中都会看到我们在系结表达式中使用这些系结引擎的全局方法。

9-5 系结编辑器，观察元和系结范例组件

在系结框架中还有几个观念是读者必须了解的，它们是系结编辑器，观察元和执行范围组件。

系结编辑器(Editors)定义了如何修改控制的抽象接口，实作系结编辑器的类别可以和许多控件结合在一起以提供系结表达式执行时修改控件的内容，例如系结编辑器可以和 TEdit，TListBox 结合。在系结类别中的 TBindList 就使用了系结编辑器来修改串行控件的内容。系结框架为许多的 FMX 和 VCL 控件都实作了系结编辑器。

观察元(Observers)定义了观察一个控件是否被修改的抽象接口，实作观察元的类别可以和控件结合在一起以提供使用者互动的观察机制，例如 TBindLink 系结类别便使用了观察元来观察它系结的 TEdit 组件是否被用户修改了。系结框架也为 FMX 和 VCL 控件实作了许多观察元。

下面的表格说明了使用系结编辑器和观察元的系结类别:

组件	使用系结编辑器	使用观察元
TBindExpression, TBindExprItems		
TBindPosition		X
TBindList, TBindGridList	X	
TBindLink		X
TBindGridLink, TBindListLink	X	X

在前面小节介绍系结框架时讨论过了系结执行范围的观念，不过在前面的小节中是使用程序代码来取得执行范围，Delphi 已经把一些通用的执行范围封装为不同的组件让开发人员可以直接使用，在前面的范例中我们已经使用了许多的执行范围组件，例如 TBindScopeDB 和 TBindScopeDBX 等，这两个组件都是系结数据源为系结执行范围，下面的表格说明了 DX10 中所有的执行范围组件:

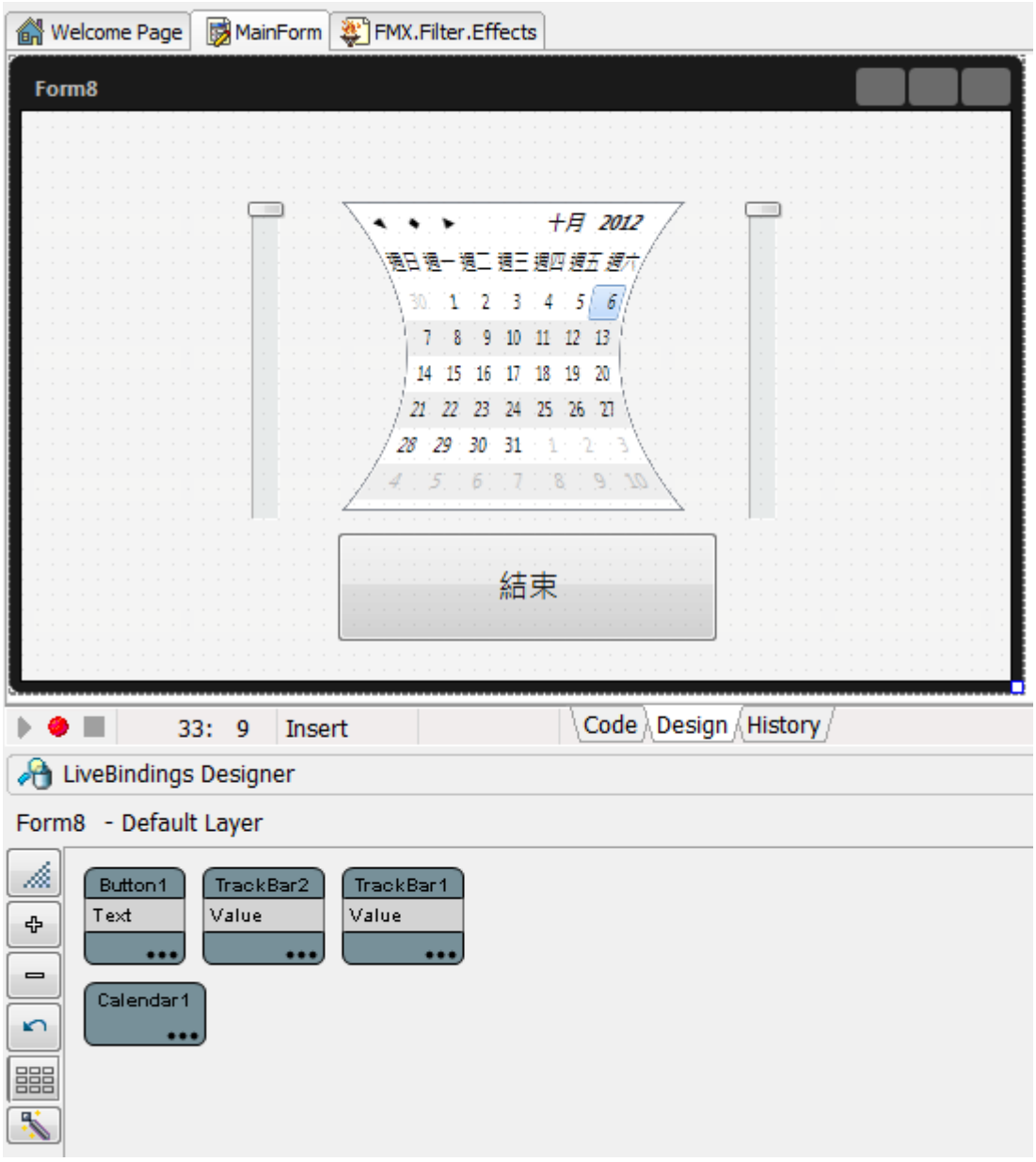
系结范围组件	说明
 TBindScope	系结客制化类似对象为执行范围
 TBindScopeDB	系结通用数据源为执行范围
 TPrototypeBindSource	系结随机产生数据的虚拟数据表为执行范围
 TBindSourceDBX	系结 dbxExpress 组件为执行范围，例如 TClientDataSet
 TAdapterBindSource	做为系结不同系结范围组件的 Adapter，使用 TAdapterBindSource 组件开发人员可以在不同的系结范围组件间切换

在下一章讨论客制化类别和实时数据系结时会讨论如何使用 TBindScope 组件和 TAdapterBindSource 件。

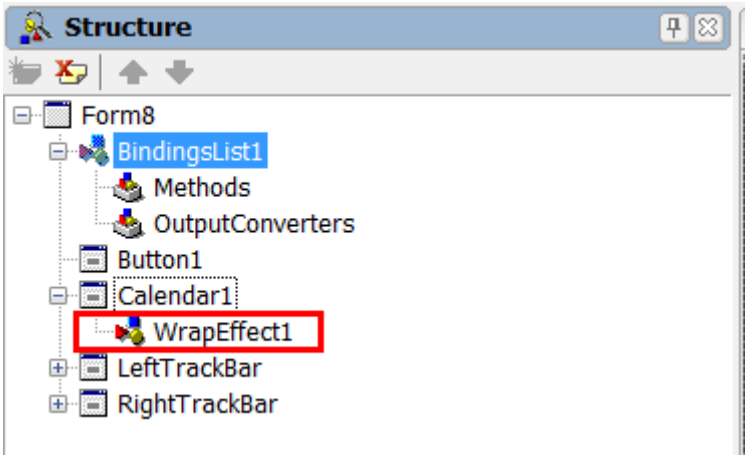
9-6 使用实时数据系结设定

当开发人员使用可视化实时数据系结设计家进行系结设计时，并不是所有的 Delphi 组件都会出现在设计家中，因为在内定上 Delphi 整合发展环境只设定了开发人员最常使用的组件才会出现在设计家中，但开发人员可以改变这个内定的设定让任何开发人员想使用的组件都能够出现在设计家中。

在 Delphi 整合发展环境中建立一个 FireMonkey Desktop Application 应用程序，在主窗体中放入 2 个 TTrackBar 组件，一个 TCalendar 组件，一个内嵌在 TCalendar 组件中的 TWrapEffect 组件以及一个 TButton 组件，开启可视化实时数据系结设计家，此时主窗体和可视化实时数据系结设计家如下所示:

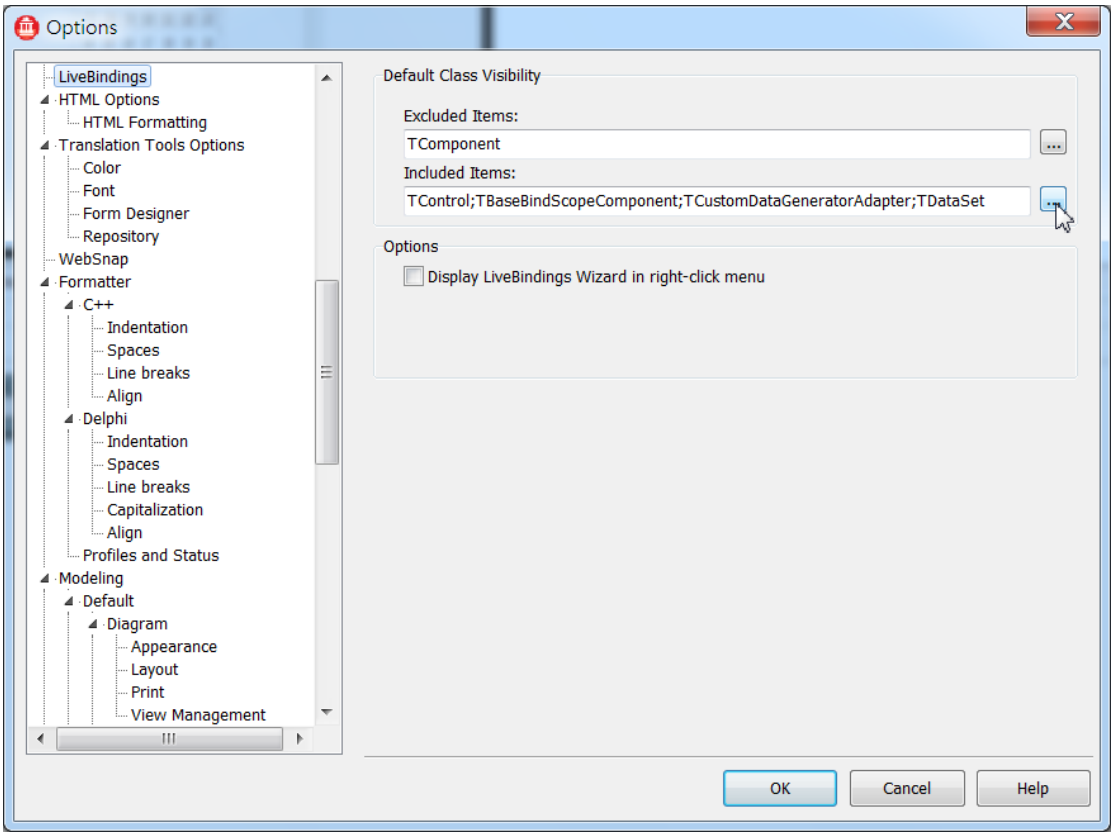


由于 TWrapEffect 组件是内嵌在 TCalendar 组件中，因此我们可以在整合发展环境左上方的树状架构窗口中看到如下的架构：

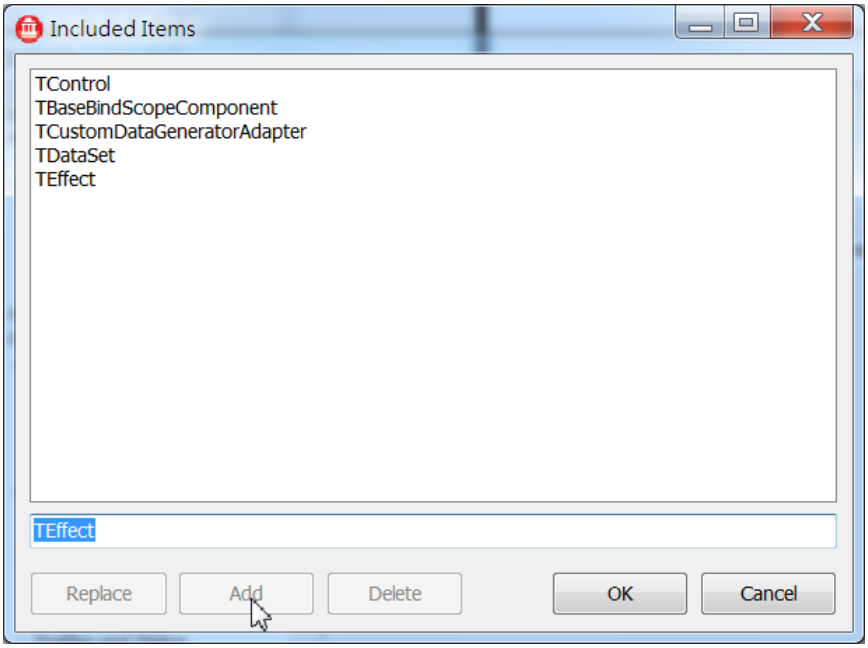


但请读者观察上面的可视化实时数据系结设计家，我们会发现 TWrapEffect 组件并没有出现在设计家中，为什么？ 这就是因为 TEffect 相关的类别在内定被设定为不出现在可视化实时数据系结设计家。但现在我们希望能够系结主窗体中的 2 个 TTrackBar 组件到 TWrapEffect 组件以便能够拖曳 TTrackBar 组件来改变 TWrapEffect 组件对于 TCalendar 组件的影响程度，因此我们需要在实时数据系结设计家中系结 TTrackBar 和 TWrapEffect，因此我们需要 TWrapEffect 能够出现在设计家中。

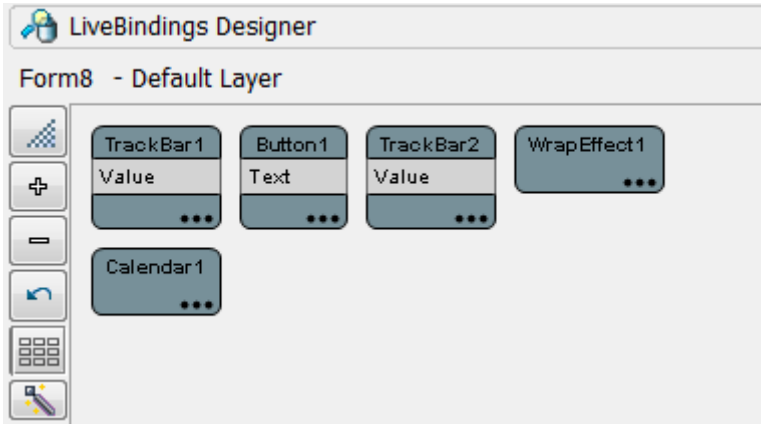
要让 TWrapEffect 出现在设计家中很简单，我们只需要改变可视化实时数据系结设计家的内定设定即可。请点选 Tools|Options 菜单开启 Options 对话盒，在 LiveBindings 选项中点选 Included Items 控件右方的单选按钮，如下所示：



在 Included Items 对话盒中下方的 Edit 控件中输入 TEffect 类别名称把 TEffect 类别和所有的衍生类别都加入到可显示于可视化实时数据系结设计家中的类别，然后点选下方的 Add 按钮，再点选 OK 按钮。

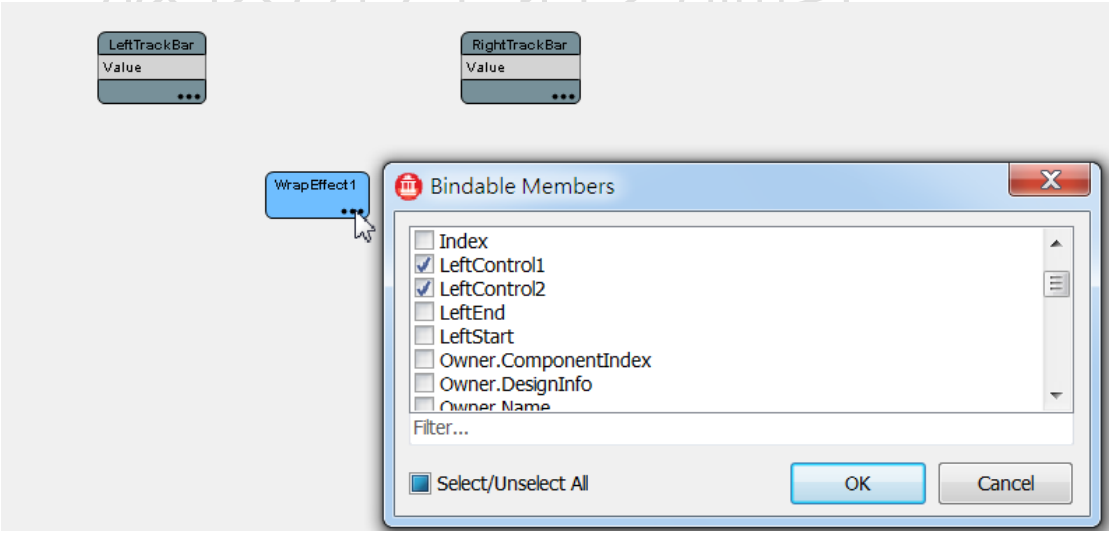


回到可视化实时数据系结设计家中此时就可以看到类似下面的结果，现在TWrapEffect组件已经出现在可视化实时数据系结设计家中了：

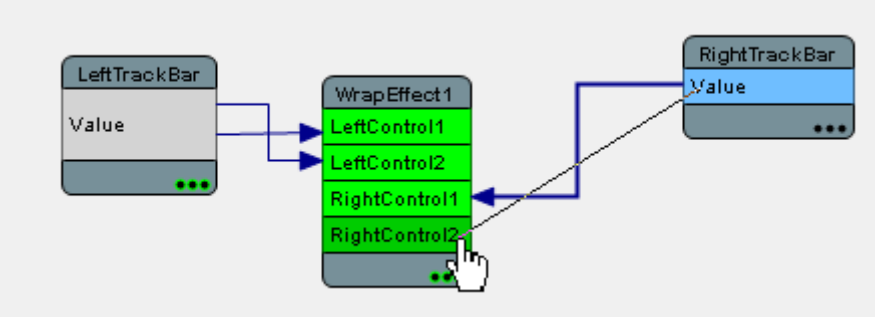


现在我们希望系结主窗体中的2个TTrackBar组件和TWrapEffect组件，当拖曳主窗体中左右2方的TTrackBar组件时能够改变TWrapEffect组件的特性值，如此一来就可以影响主窗体中TCalendar组件的缠绕效果。

请点选可视化实时数据系结设计家中WrapEffect1实体右下方的『…』，于Bindable Members对话盒中勾选LeftControl1，LeftControl2，RightControl1和RightControl2特性以把这4个特性显示于WrapEffect1实体中，如下所示：



接着拖曳LeftTrackBar的Value特性到WrapEffect1实体的LeftControl1，LeftControl2特性，再拖曳RightTrackBar的Value特性到WrapEffect1实体的RightControl1和RightControl2特性，以系结左右2个TTrackBar组件到WrapEffect1组件的4个特性，如下所示：



現在执行此范例 FireMonkey 应用程序并且拖曳主窗体上的 2 个 TTrackBar 组件就可以看到 TWrapEffect 组件对于 TCalendar 组件的影响了，如下所示:

9-7 结论

本章说明了实时数据系结概念，让读者了解在撰写系结表达式时，只要了解了来源执行范围，控制执行范围，以及在执行范围中能够存取的组件，方法和特性之后，撰写系结表达式就非常的简单了。

掌握实时数据系结概念之后本章也使用 Delphi 的系结组件来印证实时数据系结概念，并且介绍了 DX10 的时数据系结类别，也使用了一些时数据系结类别开发了一些范例让读者了解如何使用数据系结类别

最后本章说明了如何设定可视化实时数据系结的选项以便让开发人员能够控制出现在可视化实时数据系结设计家中的类别。

在下一章将讨论更多实时数据系结的技术和使用方法。

第10章 执行范围组件和Adapter

DX10 的实时数据系结技术开始引进了系结 Adapter 观念，系结 Adapter 允许开发人员藉由系结 Adapter 而能够让控件和不同的系结来源系结，如此一来开发人员就可以切换使用不同的系结来源。例如当开发人员使用实时数据系结技术进行 POC(Proof Of Concept)时，可以系结雏型数据源和控件以进行快速开发，当客户或是使用者认可之后再切换到真正的数据源来继续使用实时数据系结技术进行后续的开发工作。

例如在前面的章节中介绍的 TPrototypeBindSource 组件可以快速产生随机的数据以形成虚拟数据表，让开发人员可以使用这些随机产生的资料进行雏型开发和展示之用，当客户或是使用者确认之后就可以切换成使用 dbExpress 组件链接到实际的数据库继续进行开发工作。

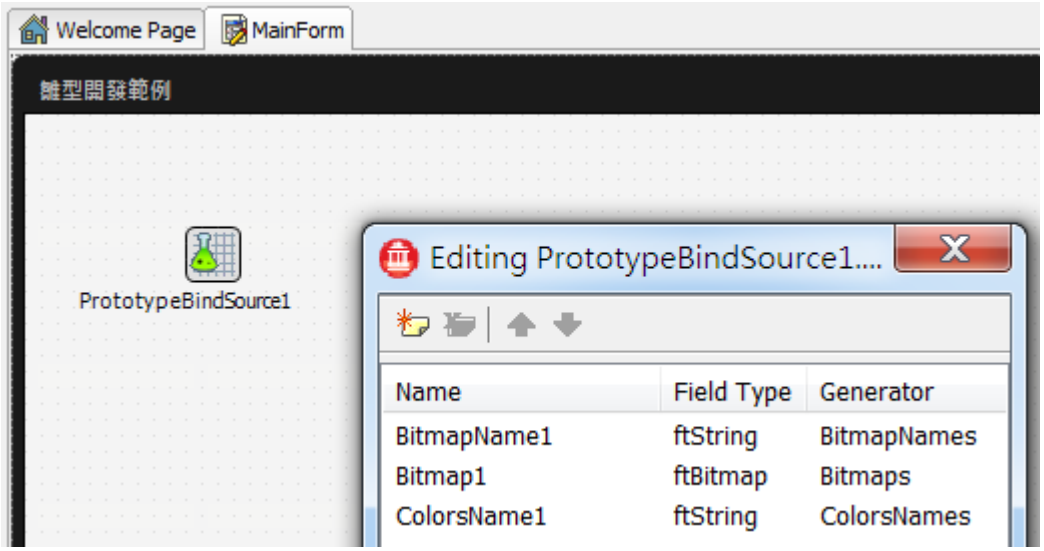
此外在许多的应用中，开发人员并不一定是系结数据源和控件，而是希望系结客制化类别对象和控件，在这种应用中开发人员也可以藉由使用 Delphi 的执行范围系结组件来完成这种工作。

在本章中我们将详细说明这 2 种系结技术，让读者可以掌握所有的系结技术。

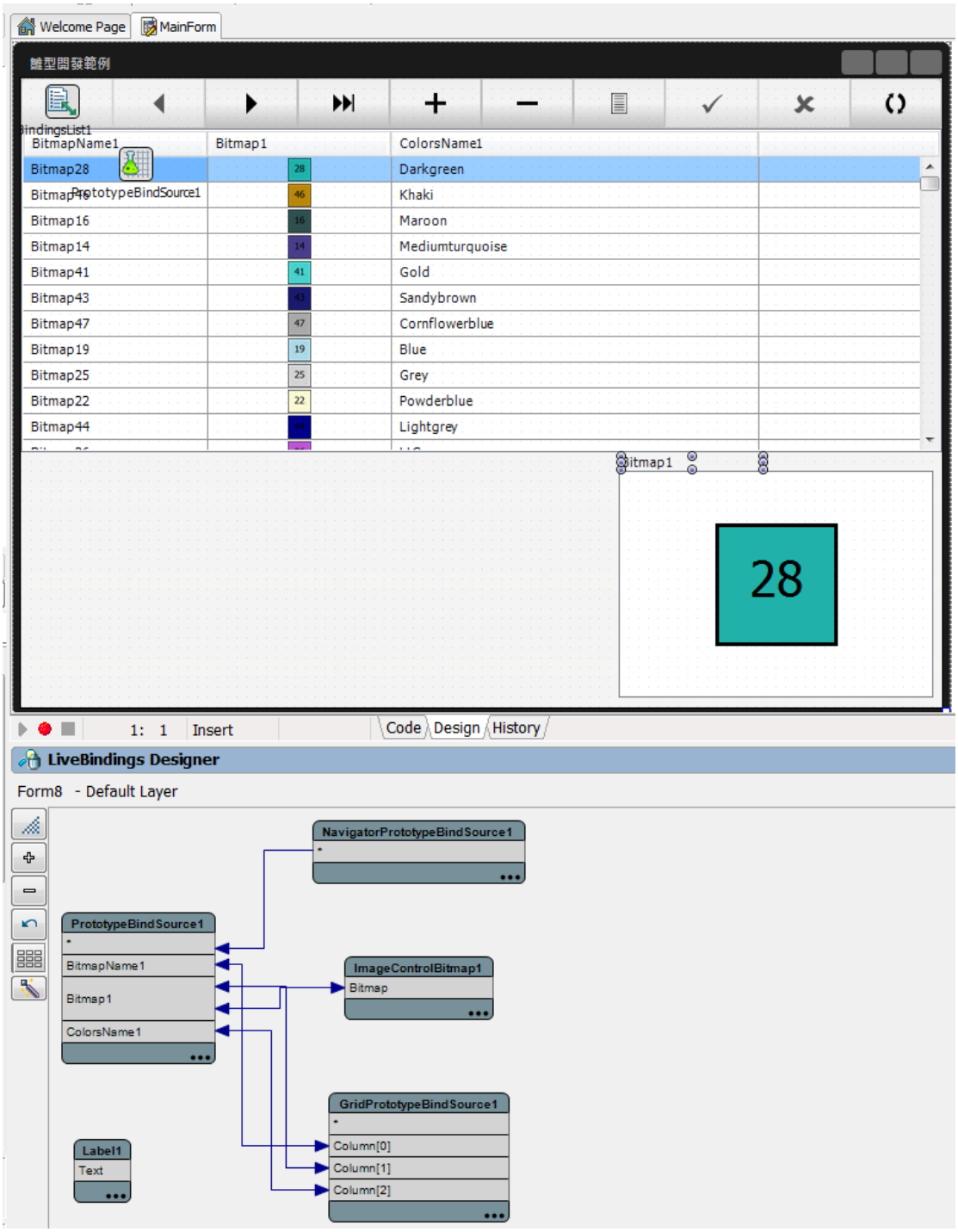
10-1 TPrototypeBindSource 组件和雏型开发

首先让我们说明如何使用 TPrototypeBindSource 组件来进行雏型开发，接着代换成实际数据源的技巧。

请在整合发展环境中建立一个 FireMonkey Desktop Application 项目，在主窗体中放入 TPrototypeBindSource 组件，然后在其中选择建立一些数据字段如下所示:



接着启动可视化实时数据系结设计家，连结 TGrid，TBindNavigator，TImageControl 等组件到 TPrototypeBindSource 组件如下所示:



由于 TPrototypeBindSource 组件中的字段和数据都是随机产生的，因此假设这是一个雏型应用程序，在用户认可之后，那么我们是否可以把这个范例应用程序的随机数据代替成实际的数据，而无需重新建立一个项目再重头开始呢?

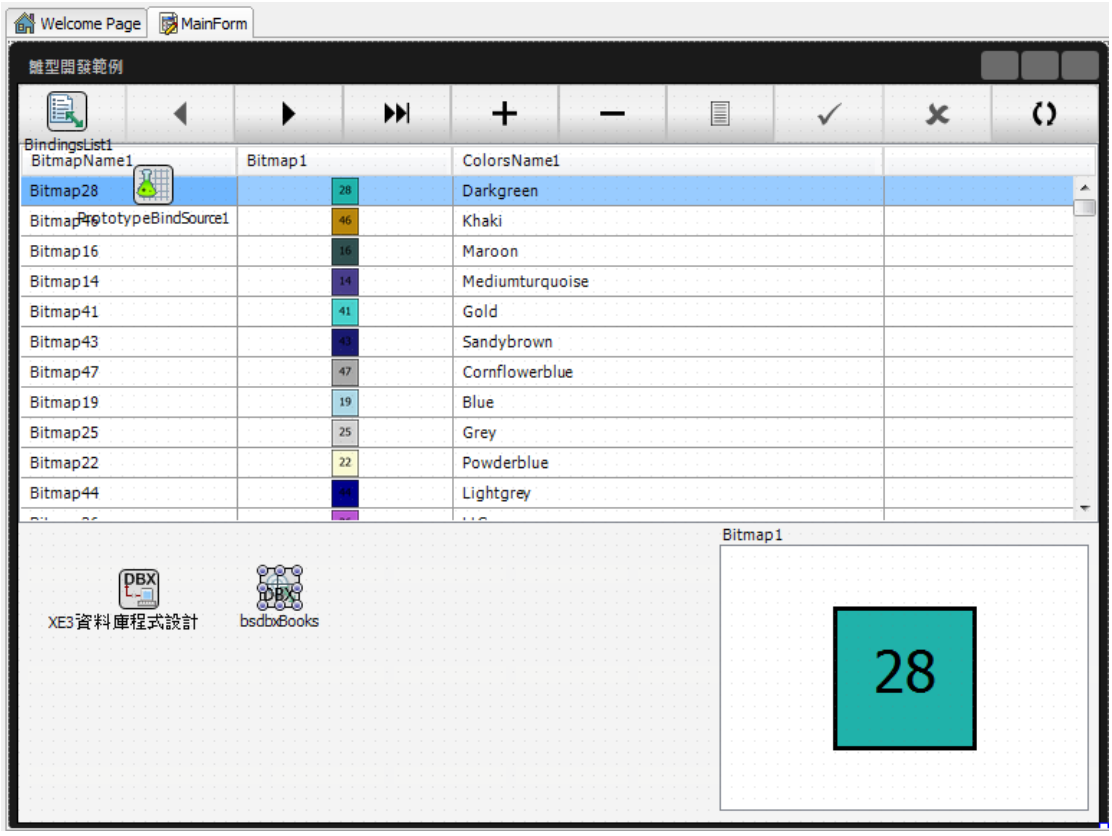
这当然是可以的，因为 DX10 的系结引擎提供了 Adapter 的观念，因此在这种情形中开发人员可以使用数种方法把随机的数据代换成真正的数据。开发人员可以使用可视化

实时数据系结设计家来代换数据或是使用 Adapter 组件和程序代码来代换数据。在本范例中先让我们说明如何使用可视化实时数据系结设计家来代换数据。

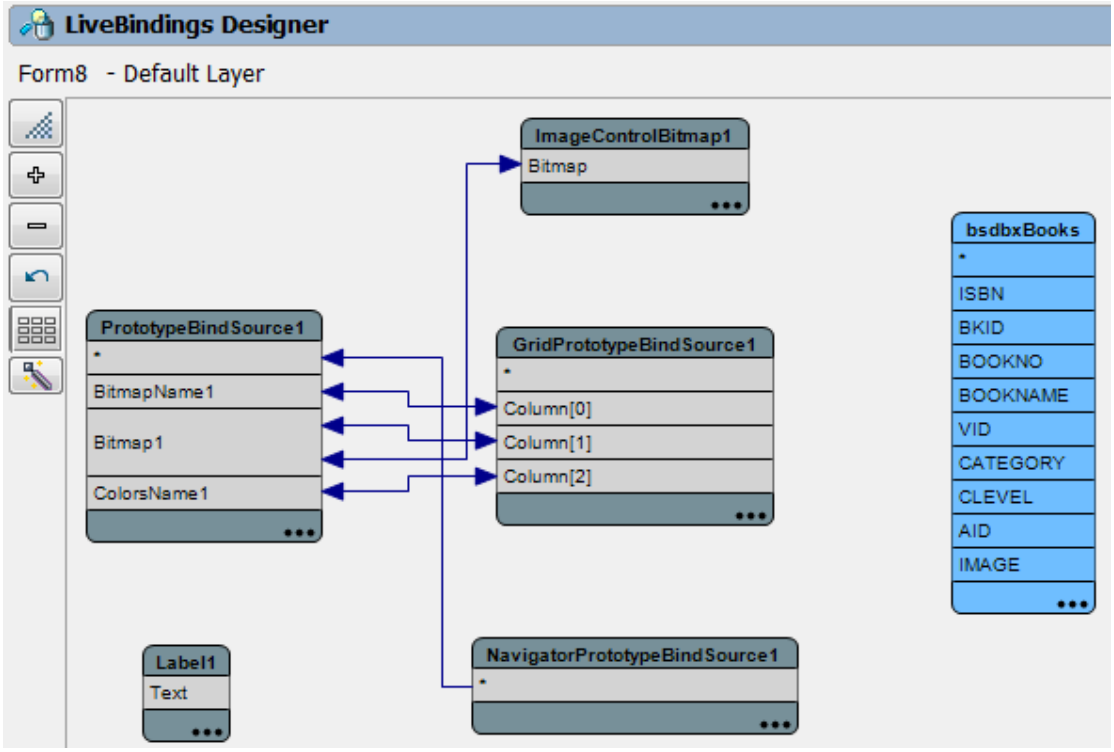
假设现在我们想取代 TPrototypeBindSource 组件的随机数据成 BOOKS 数据表中实际的数据，那么要如何使用可视化实时数据系结设计家完成工作？很简单，现在请回到主窗体，拖曳 Data Explorer 中的『XE7 数据库程序设计』节点到主窗体中以建立 TSQLConnection 组件，设定它的 Active 特性值为 True，再放入 TBindSourceDBX 组件链接 TSQLConnection 组件，再于它的 CommandText 特性中输入：

```
Select * from BOOKS
```

如下所示：



接着设定 TBindSourceDBX 组件的 Active 特性值为 True，此时可视化实时数据系结设计家中就会出现 TBindSourceDBX 实体，如下所示：

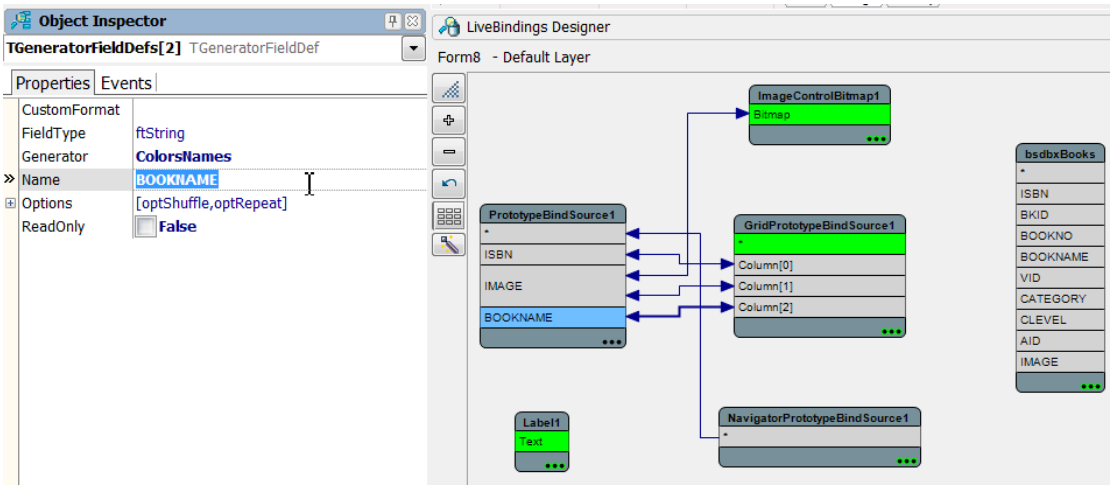


现在我们就可以开始资料代换的工作，假设现在我们希望使用 BOOKS 数据表中的 ISBN，BOOKNAME 和 IMAGE 这 3 个字段的数据，那么我们只需要进行下面的步骤即可完成数据代换的工作:

1. 把代 TPrototypeBindSource 组件中相对应的数据型态的域名改变为和 BOOKS 数据表中的 ISBN，BOOKNAME 和 IMAGE

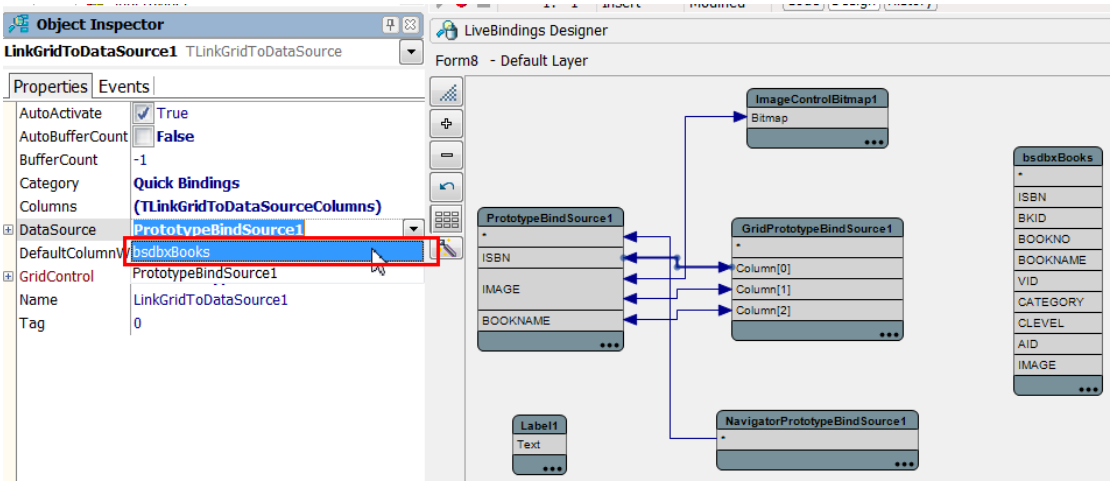
2. 在可视化实时数据系结设计家中点选所有原先链接 TPrototypeBindSource 实体的链接线，在对象查看器中设定它的 DataSource 为 TBindSourceDBX 实体

现在请回到可视化实时数据系结设计家，一一点选 TPrototypeBindSource 实体中的字段，再于对象查看器中点选它的『NAME』 特性值，把 TPrototypeBindSource 组件中的域名改变成 ISBN，BOOKNAME 和 IMAGE，如下所示:

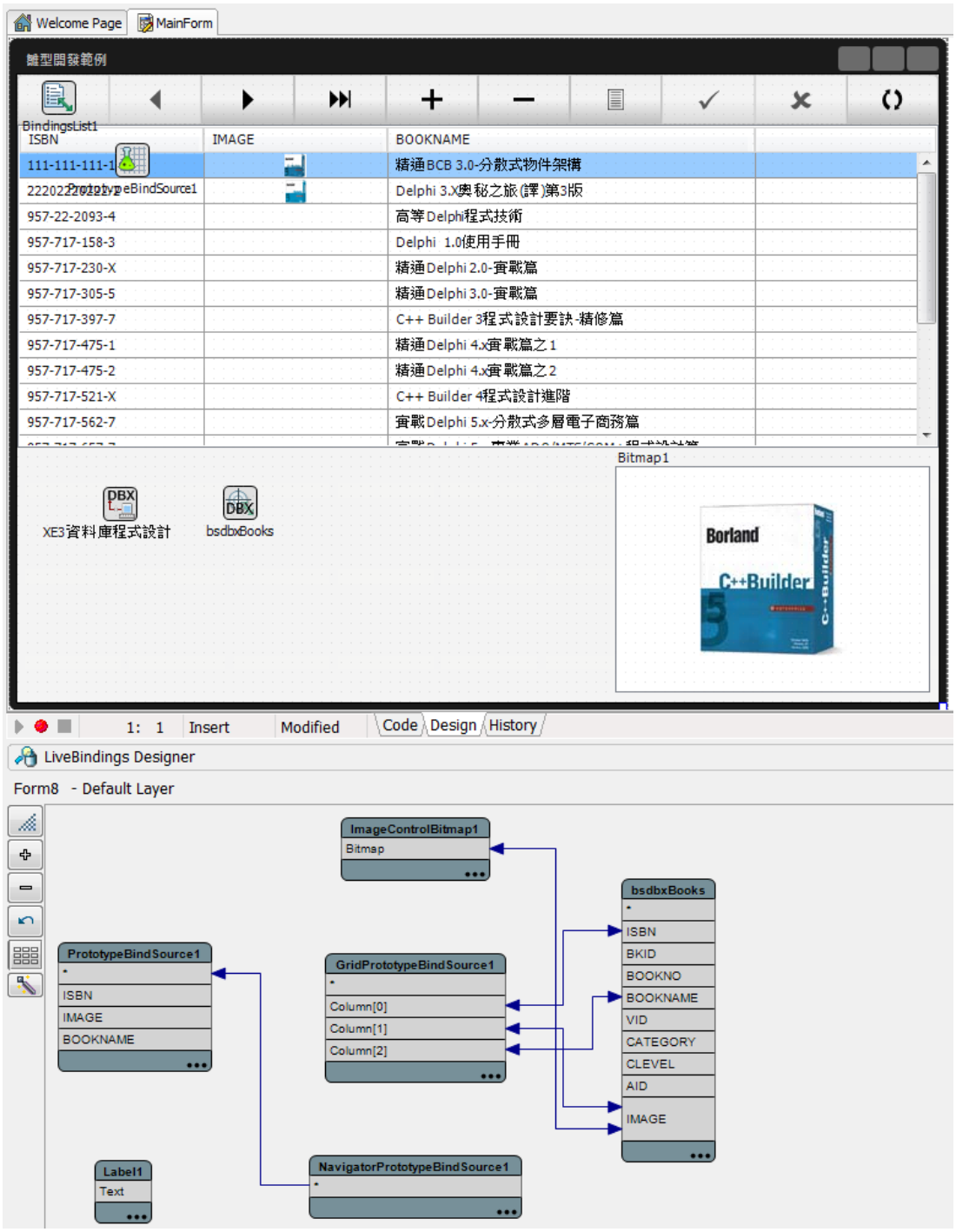


现在 TPrototypeBindSource 组件中的域名和真正的 BOOKS 数据表中的数据型态和域名都一样了。

接着在可视化实时数据系结设计家中点选所有链接 TPrototypeBindSource 实体的链接线，在对象查看器中设定它的 DataSource 为 TBindSourceDBX 实体，如下所示:



一旦把连结线的 DataSource 特性值改为 TBindSourceDBX 组件之后就可以看到如下的结果，不但可视化实时数据系结设计家中的连结线现在改变到链接 TBindSourceDBX 实体，主表格中的数据也都切换成 BOOK 数据表中实际的资料了:



这个代换数据的技巧适用于数据源，但如果是客制化的数据，例如用户的类别对象，那么如何能够使用系结引擎和控件结合呢？这就需要使用系结引擎的 Adapter 类别了。

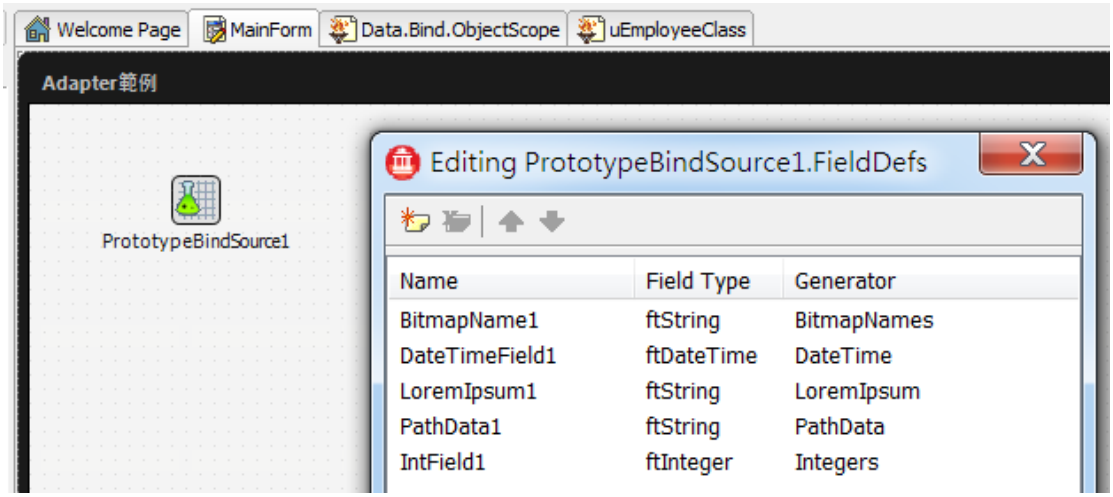
10-2 系结引擎 Adapter 类别

假设现在在项目有下面的 TEmployee 客制化类别，我们希望这个客制化类别也能够使用实时数据系结技术链接 TEmployee 类别对象和 FireMonkey 的控件，那么我们要如何完成这个需求。

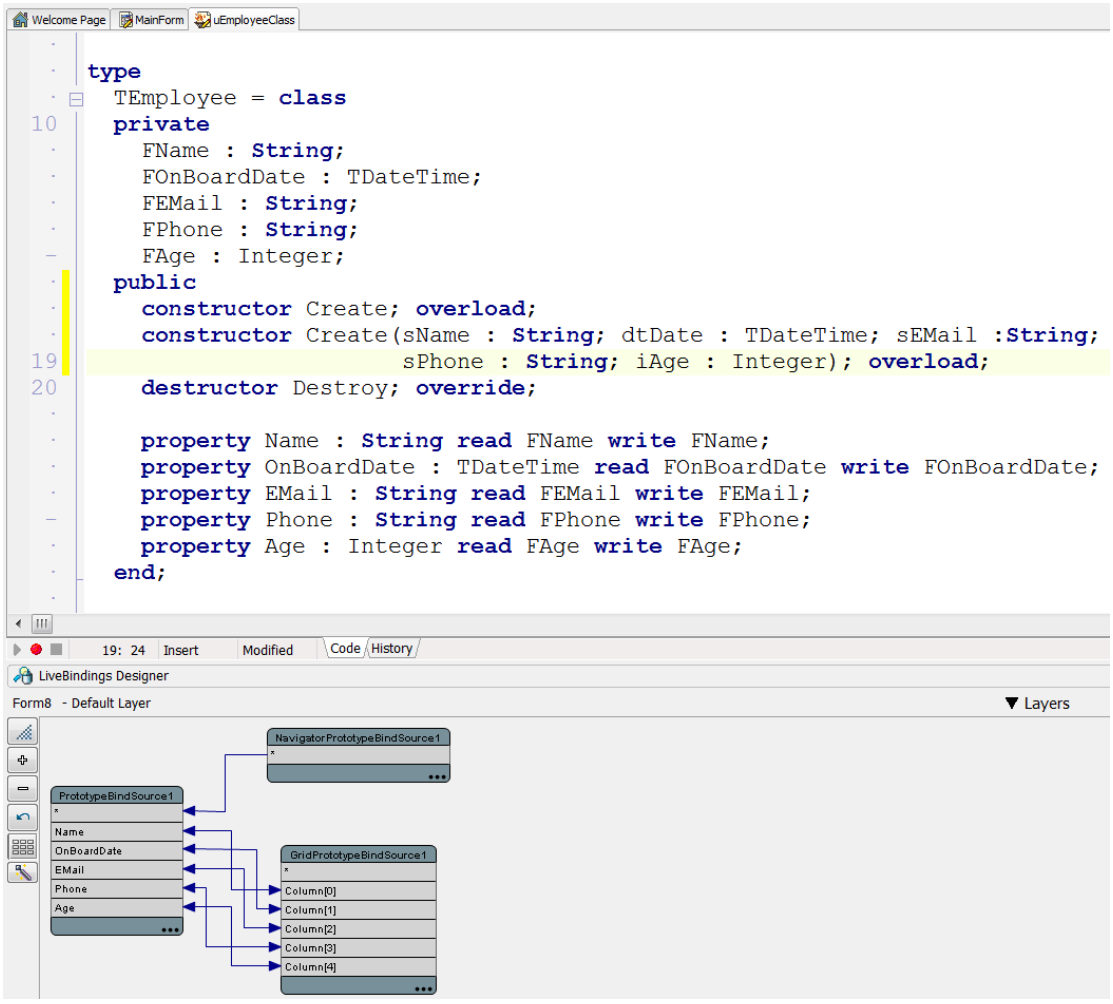
```
type
  TEmployee = class
  private
    FName : String;
    FOnBoardDate : TDateTime;
    FEMail : String;
    FPhone : String;
    FAge : Integer;
  public
    constructor Create; override;
    destructor Destroy; override;

    property Name : String read FName write FName;
    property OnBoardDate : TDateTime read FOnBoardDate write
FOnBoardDate;
    property EMail : String read FEMail write FEMail;
    property Phone : String read FPhone write FPhone;
    property Age : Integer read FAge write FAge;
  end;
```

要让客制化类别和控件系结在一起，在 DX10 的系结引擎中可以使用 Adapter 类别，其中最主要的两个 Adapter 类别就是 TObjectBindSourceAdapter 和 TListBindSourceAdapter 类别，这两个类别都是泛型类别，现在就让我们使用这两个类别来说明如何系结 TEmployee 客制化类别和 FireMonkey 的控件。

首先建立一个 FireMonkey Desktop Application 项目，在主窗体中放入 TPrototypeBindSource 组件，接着在其中建立 5 个字段对象，因为在 TEmployee 类别中拥有 5 个特性，另外请注意的是这 5 个字段对象的型态必须一一的匹配 TEmployee 类别中的 5 个特性，如下图所示:



接着使用可视化实时数据系结设计家系结 TGrid 和 TBindNavigator 组件到主窗体的 TPrototypeBindSource 组件，下图显示了现在的 TEmployee 类别和可视化实时数据系结设计家:



OK，现在我们就可以使用 Adapter 类别以 TEmployee 的类别对象来取代 TPrototypeBindSource 组件中的随机数据了。

10-2-1 TObjectBindSourceAdapter 类别

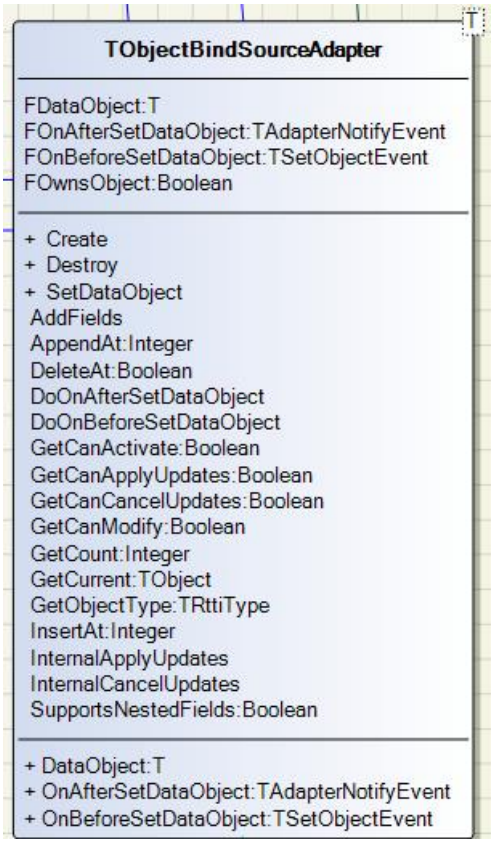
当 TPrototypeBindSource 组件系结到控件并且准备显示数据时，TPrototypeBindSource 组件会根据开发人员在其中建立的字段对象的数据型态来产生随机数据，不过 TPrototypeBindSource 组件在产生数据之前会先触发一个事件处理函式，这个事件处理函式允许开发人员使用 Adapter 类别对象来取代 TPrototypeBindSource 组件原先产生随机数据的机制。因此我们可以利用这个原理来系结 TEmployee 类别对象和控件。TPrototypeBindSource 组件的这个事件处理函式称为 OnCreateAdapter ，它的宣告原型如下:

```
  TCreateAdapterEvent = procedure(Sender: TObject; var
ABindSourceAdapter: TBindSourceAdapter) of object;
```

OnCreateAdapter 事件处理函式接受两个参数，其中第二个参数 ABindSourceAdapter 是型态为 TBindSourceAdapter 型态的对象，开发人员需要在这个事件处理函式中建立 TBindSourceAdapter 类别的衍生类别对象并且指定给参数 ABindSourceAdapter，如此一来当 TPrototypeBindSource 组件要产生数据并且绑定控件时，就会向 ABindSourceAdapter 参数中的 TBindSourceAdapter 的衍生类别对象要求数据。

因此请读者回头看前面的可视化实时数据系结设计家中 TPrototypeBindSource 组件是以名称为 NAME，OnBoardDate，EMail，Phone 和 Age 的字段系结到 TGrid 组件，因此如果我们在 TPrototypeBindSource 组件的 OnCreateAdapter 事件处理函式中建立 TBindSourceAdapter 的衍生类别对象，再让这个 TBindSourceAdapter 的衍生类别对象链接到 TEmployee 类别对象，那么由于 TEmployee 类别对象也拥有相同名称的 NAME，OnBoardDate，EMail，Phone 和 Age 特性，因此 TPrototypeBindSource 组件就会使用 RTTI 机制以这些名称向 TEmployee 类别对象要求数据，如此一来 TEmployee 类别对象的数据就会出现在 TGrid 组件中，这也就完成了系结 TEmployee 类别对象和控件的需求。

由于 TEmployee 是类别，而且我们需要使用 Adapter 类别和 TPrototypeBindSource 组件系结，因此我们需要的是 TBindSourceAdapter 的衍生类别而且必须是一个泛型类别，因此 TObjectBindSourceAdapter 类别就是最好的的目标类别，下图是 TBindSourceAdapter 类别的定义，请读者注意下图的右上角有一个『T』符号，代表它是一个泛型类别:



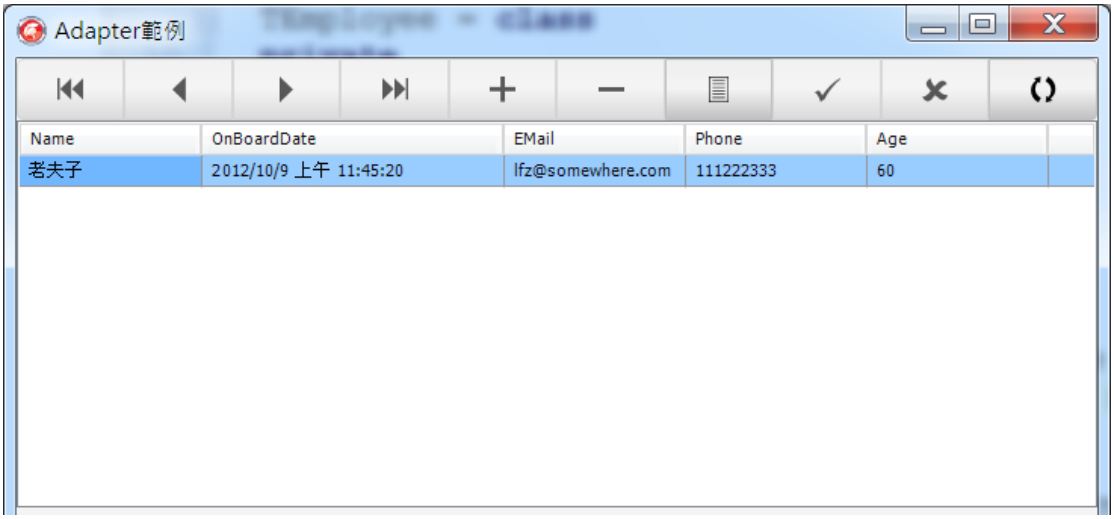
而下面则是 TObjectBindSourceAdapter 的定义，它接受一个类别做为泛型参数：

```
  /// <summary>Adapter to provide an arbitrary object to
TAdapterBindSource</summary>
  TObjectBindSourceAdapter<T: class> =
class(TBaseObjectBindSourceAdapter)
```

现在我们就可以在主窗体的 TPrototypeBindSource 组件的 OnCreateAdapter 事件处理函式中撰写如下的程序代码：

```
001    procedure TForm8.PrototypeBindSource1CreateAdapter(Sender:
TObject;
002      var ABindSourceAdapter: TBindSourceAdapter);
003    var
004      anEmployee : TEmployee;
005    begin
006      anEmployee := TEmployee.Create('老夫子', Now,
'lfz@somewhere.com', '111222333', 60);
007      ABindSourceAdapter :=
TObjectBindSourceAdapter<TEmployee>.Create(Self, anEmployee);
008    end;
```

在 006 行先建立一个 TEmployee 对象，接着 007 行建立 TObjectBindSourceAdapter 对象并且传入 TEmployee 类别做为泛型参数，这样就完成了取代 TPrototypeBindSource 组件的随机数据成 TEmployee 对象的工作了，现在请编译和执行此 FireMonkey 范例应用程序，在下图中读者可以看到现在系结在主窗体 TGrid 组件中的数据就是我们在程序代码中建立的 TEmployee 对象的数据了：

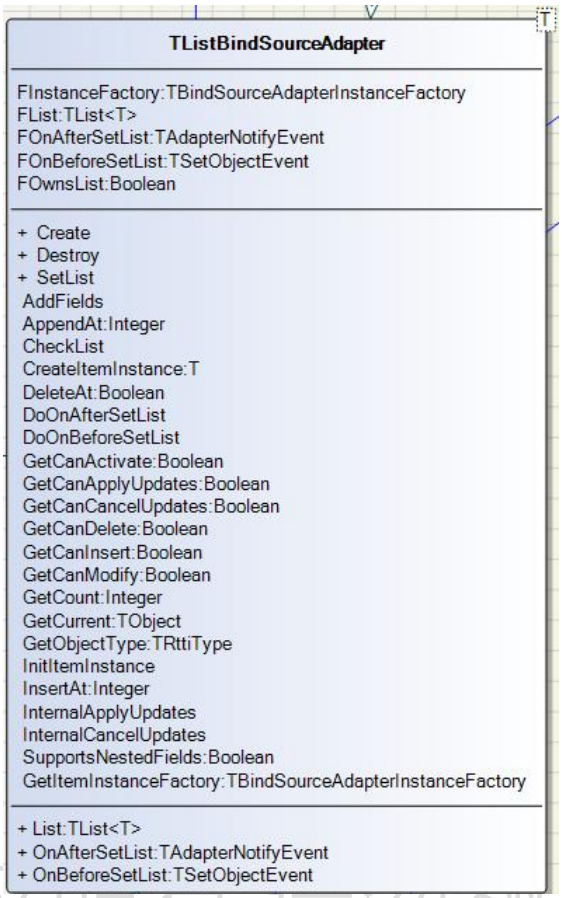


上图中的资料是可以进行修改的，但由于在前面的程序代码中只有一个 TEmployee 对象，因此现在只有一笔数据，那么如果应用程序中有多个 TEmployee 对象的话又应该如何系结呢？

就需要使用 TListBindSourceAdapter 类别了。

10-2-2 TListBindSourceAdapter 类别

TListBindSourceAdapter 类别能够让开发人员系结对象串行和控件，它和使用 TObjectBindSourceAdapter 类似的方法类似，同样是在 TPrototypeBindSource 组件的 OnCreateAdapter 事件处理函式中建立 TListBindSourceAdapter 类别对象并且指定给 ABindSourceAdapter 参数，但开发人员可以在程序代码中建立多个客制化对象，再呼叫 TListBindSourceAdapter 类别对象的 Add 方法把客制化对象加入到 TListBindSourceAdapter 类别对象中，如此一来 TPrototypeBindSource 组件就能够藉由 TListBindSourceAdapter 类别对象绑定控件了，下图是 TListBindSourceAdapter 类别的定义，请读者注意下图的右上角有一个『T』符号，代表它是一个泛型类别：



下面是是 TListBindSourceAdapter 的定义，它也接受一个类别做为泛型参数：

```
TListBindSourceAdapter<T: class> = class(TBaseListBindSourceAdapter)
```

了解了 TListBindSourceAdapter 类似之后我们就可以修改 TPrototypeBindSource 的 OnCreateAdapter 事件处理函式如下:

```
001   procedure TForm8.PrototypeBindSource1CreateAdapter(Sender:
TObject;
002     var ABindSourceAdapter: TBindSourceAdapter);
003   //var
004   //  anEmployee : TEmployee;
005   begin
006   //  anEmployee := TEmployee.Create('老夫子', Now,
'lfz@somewhere.com', '111222333', 60);
007   //  ABindSourceAdapter :=
TObjectBindSourceAdapter<TEmployee>.Create(Self, anEmployee);
008     employeeList := CreateEmployeeList;
009     ABindSourceAdapter :=
TListBindSourceAdapter<TEmployee>.Create(Self, employeeList);
```

```
010    end;
```

在008行我们先呼叫CreateEmployeeList方法建立一个TList<TEmployee>类别对象，在这个类似对象中我们会建立数个TEmployee对象以便让TPrototypeBindSource做为系结资料，下面就是CreateEmployeeList方法，读者可以看到在其中我们先建立TList<TEmployee>类别对象，再建立6个TEmployee对象并且加入到TList<TEmployee>类别对象中，最后在上面的009行我们建立TListBindSourceAdapter对象并且传递TEmployee做为泛型参数，并且传递TList<TEmployee>类别对象做为第2个实际参数。

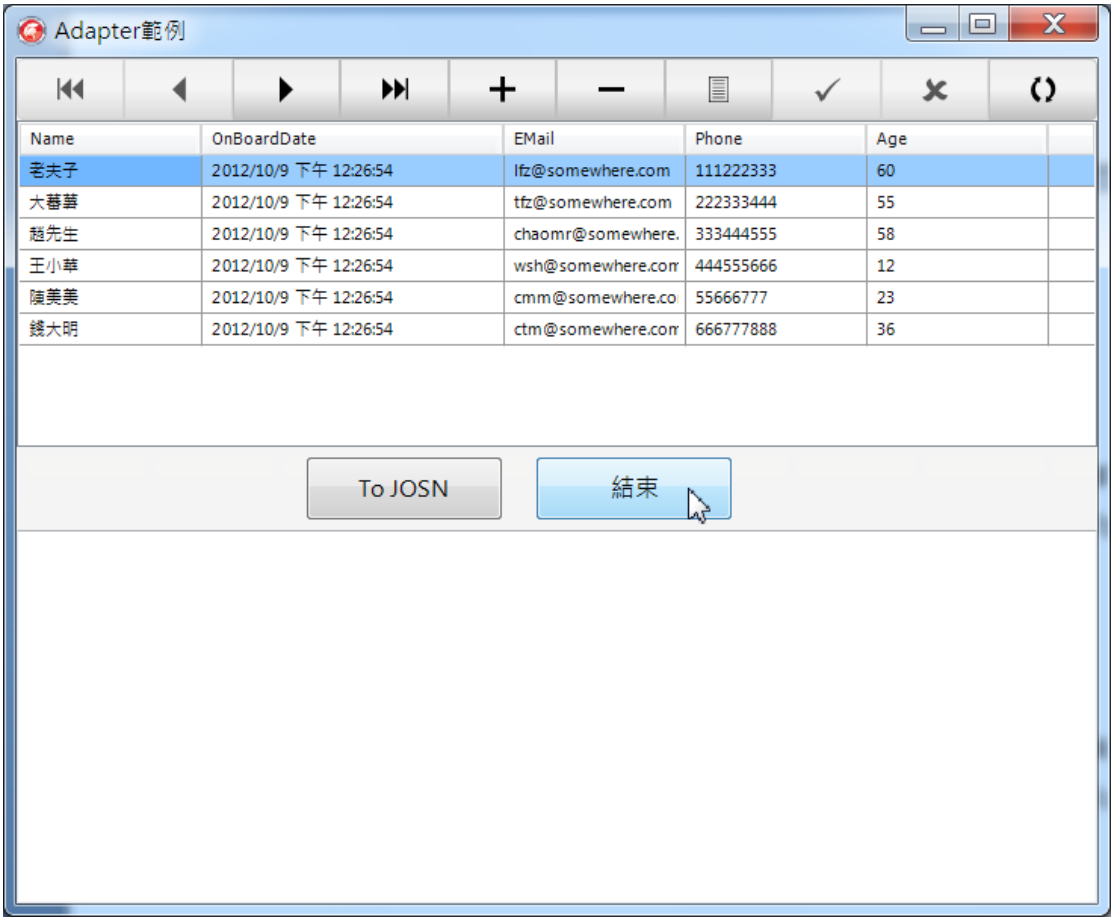
```
function TForm8.CreateEmployeeList: TList<TEmployee>;
var
  anEmployee : TEmployee;
begin
  Result :=  TList<TEmployee>.Create;

  anEmployee := TEmployee.Create('老夫子', Now, 'lfz@somewhere.com',
'111222333', 60);
  Result.Add(anEmployee);

  anEmployee := TEmployee.Create('大地瓜', Now, 'tfz@somewhere.com',
'222333444', 55);
  Result.Add(anEmployee);

  anEmployee := TEmployee.Create('赵先生', Now, 'chaomr@somewhere.com',
'333444555', 58);
  Result.Add(anEmployee);

  anEmployee := TEmployee.Create('王小华', Now, 'wsh@somewhere.com',
'444555666', 12);
  Result.Add(anEmployee);

  anEmployee := TEmployee.Create('陈美美', Now, 'cmm@somewhere.com',
'55666777', 23);
  Result.Add(anEmployee);

  anEmployee := TEmployee.Create('钱大明', Now, 'ctm@somewhere.com',
'666777888', 36);
  Result.Add(anEmployee);
```

```
end;
```

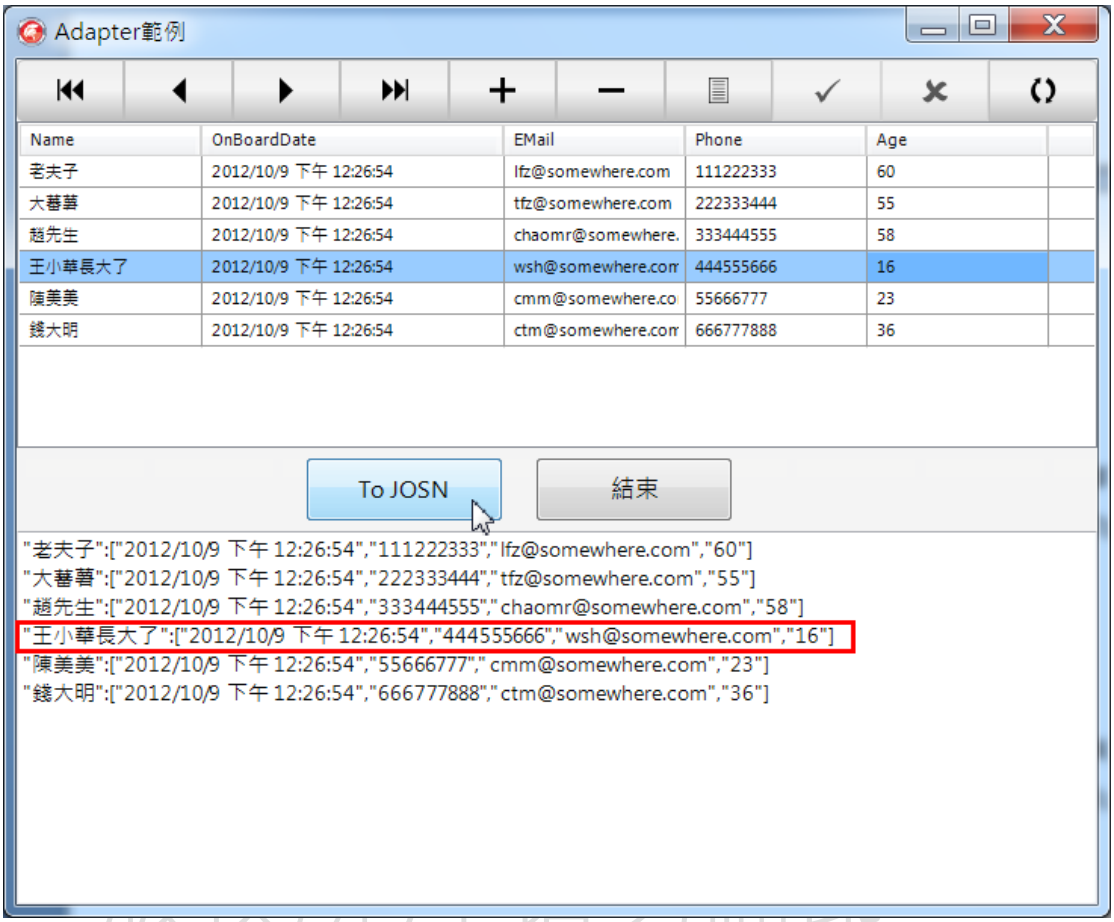
现在我们就可以准备再次执行范例程序了，但为了证明使用 Adapter 类别绑定控件时，用户也可以在控件中修改数据就像是系结到数据源一样，因此让我们在此范例程序中再撰写下面的程序代码把 TList<TEmployee>类别对象中的 TEmployee 对象封装成 JSON 格式并且在稍后我们使用控件修改了 TEmployee 对象之后使用 JSON 输出，来观察修改的数据。如果 Adapter 类别真的可以让客制化类别绑定控件并且进行 CRUD 的工作，那么开发人员也就可以再把客制化类别对象写回任何的数据源处。

```
procedure TForm8.Button1Click(Sender: TObject);
var
  jsonObjects : TJSONObject;
  jp : TJSONPair;
  ja : TJSONArray;
  iCount: Integer;
  anEmployee : TEmployee;

begin
  jsonObjects := TJSONObject.Create;
  try
    for iCount := 0 to EmployeeList.Count - 1 do
    begin
      anEmployee := EmployeeList[iCount];
      jp := TJSONPair.Create;
      jp.JsonString := TJSONString.Create(anEmployee.Name);
      ja := TJSONArray.Create;

ja.Add(DateTimeToStr(anEmployee.OnBoardDate)).Add(anEmployee.Phone).A
dd(anEmployee.EMail).Add(IntToStr(anEmployee.Age));
      jp.JsonValue := ja;
      jsonObjects.AddPair(jp);
    end;
    for iCount := 0 to jsonObjects.Size - 1 do
      Memo1.Lines.Add(jsonObjects.Get(iCount).ToString);
  finally
    jsonObjects.Free;
  end;
end;
```

现在请编译并且执行此范例 FireMonkey 应用程序，在下图的画面中读者可以看到藉由 TListBindSourceAdapter 类别对象，我们果然可以系结客制化 TEmployee 对象串行和控件在一起，这些 TEmployee 对象都正确的出现在主表格的 TGrid 组件中了。



现在请修改其中的一笔数据并且点选主窗体上方的 TBindNavigator 的 Post 按钮把修改更新回 TEmployee 对象中，如下所示:



接着点选主窗体中的『ToJSON』按钮把 TList<TEmployee>类别对象中的每一个 TEmployee 对象封装成 JSON 的格式并且显示在 TMemo 组件中，从上图我们可以看到刚才在 TGrid 中修改并 Post 回的那笔数据果然被成功更新回 TEmployee 对象中了。

现在读者也可以使用系结引擎中的 Adapter 类别来系结客制化类别和 FireMonkey 的控件了.

10-3 执行范围组件- TBindScope

TBindScope 组件就是一个系结执行范围组件，TBindScope 组件的功能是能够让不是组件型态的对象也能够藉由实时系结技术和 FireMonkey 或是 VCL 的组件系结在一起，例如 TBindScope 组件允许串行对象型态(TList，TObjectList)，或是泛型串行对象(TObjectList<TXXX>)，或是 JSON 对象串行对象和 FireMonkey 或是 VCL 的组件系结。

TBindScope 对象提供了让开发人员使用函数式动态绑定串行对象和图形用户接口组件的能力，这不但提供了动态绑定的能力，也可以让程序代码大为简化。TBindScope

组件提供了 DataObject 特性，开发人员只需要把数据指定给这个特性，再使用系结表达式即能够系结组件。现在让我们使用一个范例来说明如何使用 TBindScope 的动态绑定能力。

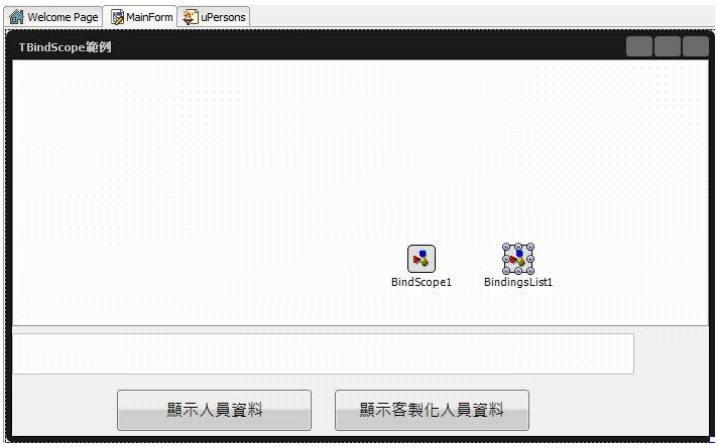
我们在这个范例中将建立一个 TPerson 对象的串行，由于 TPerson 对象串行不是组件，但我们可以藉由 TBindScope 把这个 TPerson 对象串行和 TListBox 系结在一起，让 TListBox 能够显示 TPerson 对象串行中每一个 TPerson 的资料。由于使用了 TBindScope 的实时数据系结能力，我们可以在应用程序执行时藉由动态改变系结表达式在 TListBox 中显示不同的数据，而无需修改程序代码。

首先让我们定义 TPerson 类别和 TPerson 对象串行类别如下:

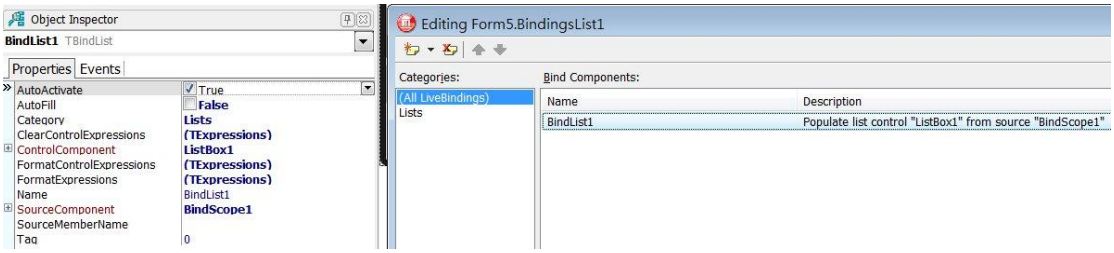
```
TPerson = class(TCollectionItem)
private
  FName: string;
  FAge: Integer;
  FEMail : string;
  FAddress : string;
public
  destructor Destroy; override;
  property Name : string read FName write FName;
  property Age: Integer read FAge write FAge;
  property EMail : string read FEMail write FEMail;
  property Address : string read FAddress write FAddress;
end;

TPersonList = class(TCollection)
end;
```

我们可以看到 TPerson 类别非常的简单，它从 TCollectionItem 类别继承下来，而 TPerson 对象串行类别 TPersonList 则从 TCollection 继承下来。

接着在 Delphi IDE 中建立一个 FireMonkey HD 应用程序项目，在主窗体中放入 TBindingsList，TBindScope，TListBox，TEdit 和两个 TButton 组件，如下所示:



接着在 TBindingsList 组件中建立一个 TBindList 对象，设定 TBindList 对象的 SourceComponent 特性值为 BindScope1，即主窗体中的 TBindScope 组件，再设定 ControlComponent 特性值为 ListBox1，即主窗体中的 TListBox 组件，如下所示:



TBindList 组件能够系结串行对象到控件，例如前面的 TPersonList 类别中将包含数个 TPerson 对象，因此 TPersonList 类别就是一个串行类别，藉由 TBindList 对象我们就可以把 TPersonList 类别对象系结到 FireMonkey 的组件，例如这个 FireMonkey 范例程序主窗体中的 TListBox 组件。

此外在上面我们把 TBindList 对象的 SourceComponent 特性值为 BindScope1，这代表一旦我们再把 BindScope1 这个组件和串行对象系结在一起那就代表 TBindList 对象的 SourceComponent 特性值是串行对象，也就代表把串行对象和 ListBox1 组件系结在一起了，而这正是我们即将做的工作。

设定 TBindList 对象系结 TBindScope 和 TListBox 的原因是稍后在程序代码中我们将再系结 TPersonList 对象和 TBindScope，如此一来 TPersonList 对象中所有的数据就可显示在 TListBox 中了。

现在让我们实作主窗体中『显示人员数据』按钮的功能，首先在主窗体的 OnCreate 事件处理函式中撰写如下的程序代码:

```
procedure TForm5.FormCreate(Sender: TObject);
```

```
begin
  CreatePersons;
end;
```

CreatePersons 会建立数个范例 TPerson 对象并且一一的加入到 TPersonList 对象中，其实作程序代码如下:

```
procedure TForm5.CreatePersons;
var
  aPerson : TPerson;
begin
  FPersonList := TPersonList.Create(TPerson);
  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '王大明', 21, 'wtm@xxxmail.com', '台北市');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '李大华', 25, 'lth@xxxmail.com', '台北市');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '林淑敏', 19, 'lsm@xxxmail.com', '台中市');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '吴月华', 23, 'wgh@xxxmail.com', '台南市');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '赵克明', 29, 'ccm@xxxmail.com', '高雄市');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '孙捷妤', 22, 'sgu@xxxmail.com', '宜兰县');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '周明明', 30, 'cmm@xxxmail.com', '花莲县');

  aPerson := FPersonList.Add as TPerson;
  SetupPerson(aPerson, '吴思捷', 25, 'wsg@xxxmail.com', '台东县');
end;

procedure TForm5.SetupPerson(aPerson : TPerson; Name: String; Age:
Integer; EMail,
```

```
  Address: String);
begin
  aPerson.Name := Name;
  aPerson.Age := Age;
  aPerson.EMail := EMail;
  aPerson.Address := Address;
end;
```

建立好了 TPersonList 对象之后，我们就可以在『显示人员数据』按钮的 OnClick 事件处理函式中撰写如下的程序代码:

```
001    procedure TForm5.Button1Click(Sender: TObject);
002    begin
003      if BindList1.FormatExpressions.Count = 0 then
004        BindList1.FormatExpressions.Add;
005      BindList1.FormatExpressions[0].SourceExpression := '''Name : ''
+ Name + '' EMail : '' + EMail + '' Age : '' + ToStr(Age) + '' Address :
'' + Address';
006      BindList1.FormatExpressions[0].ControlExpression := 'Text';
007      BindScope1.DataObject := FPersonList;
008      BindList1.FillList;
009    end;
```

在 003 行首先判断 TBindList 对象中是否已经建立了系结表达式，如果没有的话就在 004 行建立一个 TExpressionItem 系结对象，现在先让我们解释第 007 行之后再解释 005 和 006 行读者会比较容易了解。

在 007 行设定 BindScope1 的 DataObject 特性值为 FPersonList 串行对象，这代表在前面 TBindingsList 对象中定义的 TBindList 对象是让 FPersonList 串行对象中的数个 TPerson 对象和 TListBox 对象系结在一起，因此我们在 005 行设定 TExpressionItem 对象的 SourceExpression 特性值为:

```
'''Name : '' + Name + '' EMail : '' + EMail + '' Age : '' + ToStr(Age)
+ '' Address : '' + Address';
```

这是一个系结表达式，它取得 TPerson 的四个特性值并且显示在 TListBox 的一个 TListBoxItem 对象中，为什么可以在这个函数式中存取 TPerson 的特性值? 就是因为刚才说明的现在是 TPerson 和 TListBox 对象系结在一起,因此 Source 来源是 TPerson 对象。

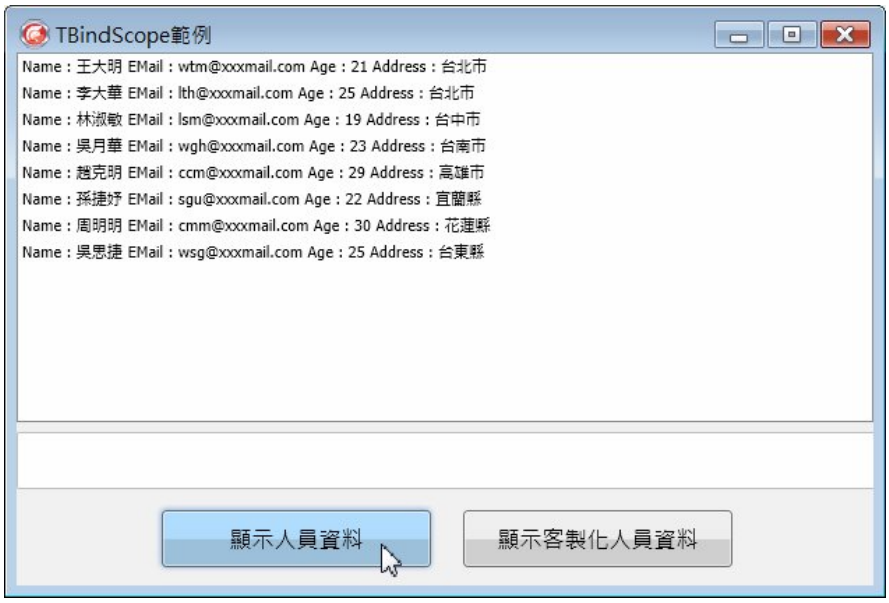
再看 006 行的函数式为什么可以使用'Text'? 这是因为 TListBox 中的对象其实是 TListBoxItem 对象，因此控制对象其实就是 TListBox 中的 TListBoxItem 对象，简单

的说最终藉由 TBindScope 和 TBindingsList 中的 TBindList 系结的双方其实是 TPerson2 件和 TListBoxItem 对象，而 Text 正是 TListBoxItem 对象的特性，因此 006 行才能够使用'Text'这个函数式。

来源执行范围	来源系结表达式
BindScope1->TPersonList->TPerson	'"Name : " + Name + " EMail : " + EMail + " Age : " + ToStr(Age) + " Address : " + Address'

控制执行范围	控制系结表达式
TListBox->TListBoxItem	Text

现在如果我们执行这个范例应用程序，点选『显示人员数据』按钮就可以看到类似如下的结果，果然每一个 TPerson 对象成功的显示在 TListBox 中了。

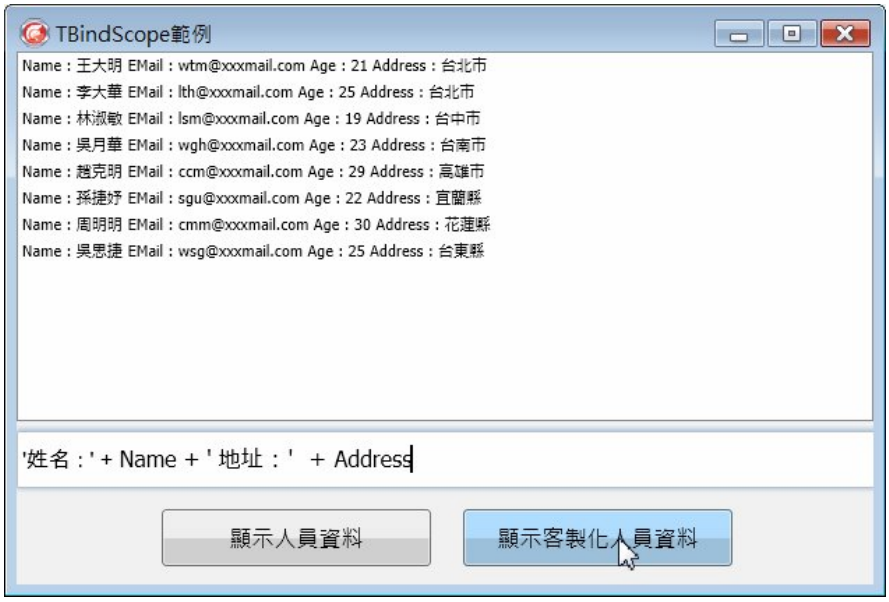


现在让我们实作一个重要的功能来展示为什么要使用 TBindScope 组件，这是因为 TBindScope 可以提供动态执行的能力，让我们在『显示客制化人员数据』按钮的 OnClick 事件处理函式中撰写如下的程序代码：

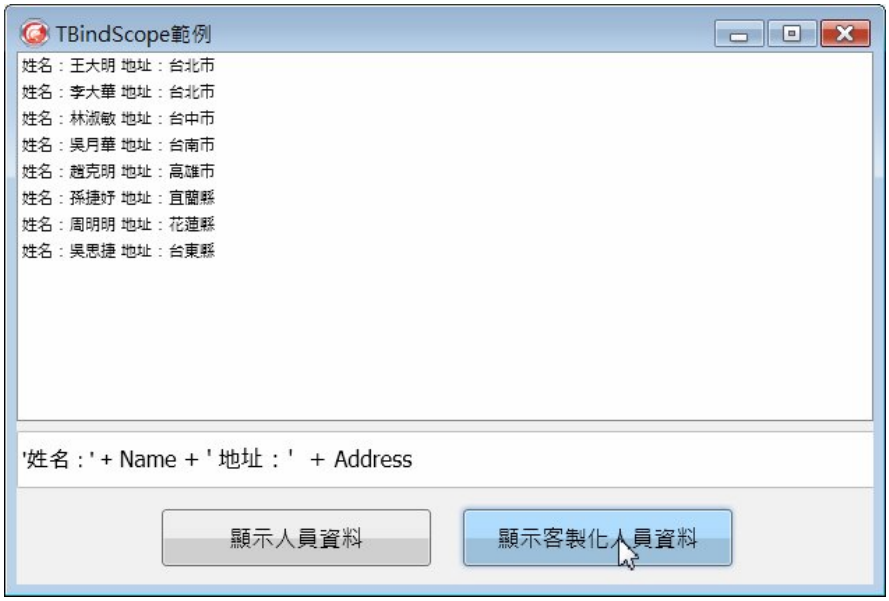
```
001    procedure TForm5.Button2Click(Sender: TObject);
002    begin
003      BindList1.ClearList;
004      BindList1.FormatExpressions[0].SourceExpression :=
edtExpression.Text;
005      BindList1.FillList;
006    end;
```

003 行先呼叫 TBindList 的 ClearList 方法以清除 TListBox 中系结的内容，004 行设定用户在主窗体中 TEdit 组件中输入的系结表达式，最后于 005 行呼叫 TBindList 的 FillList 方法重新执行系结表达式并且显示内容。

现在再次执行范例应用程序，如下图般于 TEdit 组件输入新的系结表达式:



点选『显示客制化人员数据』按钮，我们就可以看到如下的结果，藉由动态改变系结表达式就能在 TListBox 中显示不同的信息:



前面说明 TBindScope 能够和任何串行对象系结，因此 TBindScope 也可以和 TJSONArray 等对象系结，让我们再看一个小范例。现在让我们在程序代码中建立一个

TJSONArray 对象，并且在其中加入数个 TJSONString 对象，然后在 008 行把这个 TJSONArray 对象指定给 TBindScope 的 DataObject 特性，如下的程序代码所示:

```
001   procedure TForm5.Button3Click(Sender: TObject);
002   begin
003     CreateJSONArray;
004     if BindList1.FormatExpressions.Count = 0 then
005       BindList1.FormatExpressions.Add;
006     BindList1.FormatExpressions[0].SourceExpression := '''JSON 字符
串: '' + ToString() ';
007     BindList1.FormatExpressions[0].ControlExpression := 'Text';
008     BindScope1.DataObject := FJSONArray;
009     BindList1.FillList;
010   end;
011
012   procedure TForm5.CreateJSONArray;
013   begin
014     if (not assigned(FJSONArray)) then
015     begin
016       FJSONArray := TJSONArray.Create;
017       FJSONArray.AddElement(TJSONString.Create('Delphi XE2'));
018       FJSONArray.AddElement(TJSONString.Create('C++Builder XE2'));
019       FJSONArray.AddElement(TJSONString.Create('Delphi Prism
XE2'));
020       FJSONArray.AddElement(TJSONString.Create('RadPHP XE2'));
021     end;
022   end;
```

一旦在 008 行设定 BindScope1.DataObject 为 FJSONArray，就代表来源执行范围是 FJSONArray 对象。

006 行重新设定函数式，006 行的 ToString()是呼叫 TJSONArray 中 TJSONString 对象的 ToString()方法，因为现在是 TListBoxItem 系结 TJSONString 对象。

来源执行范围	来源系结表达式
BindScope1->FJSONArray-> TJSONString	"JSON 字符串: " + ToString()

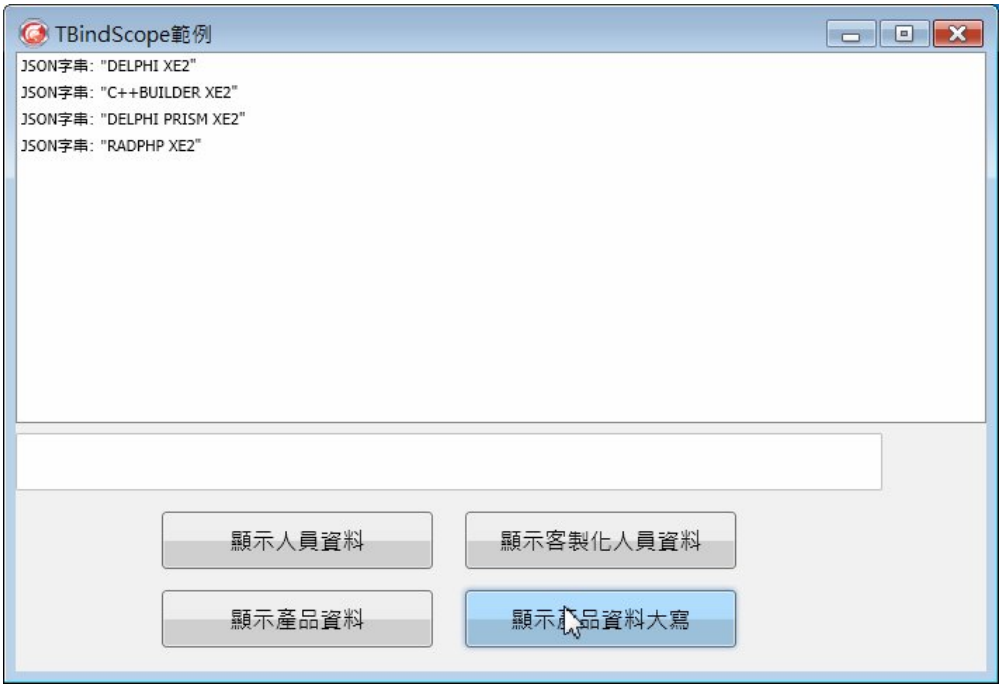
控制执行范围	控制系结表达式
TListBox->TListBoxItem	Text

执行范例应用程序，点选如下的『显示产品数据』按钮就可以看到下面的结果，TListBox 果然正确的显示了 TJSONArray 中的每一个 TJSONString 对象：

最后再让我们在范例应用程序中加入一个『显示产品数据大写』按钮，并且实作它的 OnClick 事件处理函式如下：


```
001    procedure TForm5.Button4Click(Sender: TObject);
002    begin
003      CreateJSONArray;
004      if BindList1.FormatExpressions.Count = 0 then
005        BindList1.FormatExpressions.Add;
006      BindList1.FormatExpressions[0].SourceExpression := '''JSON 字符串: '' + UpperCase(ToString()) ';
007      BindList1.FormatExpressions[0].ControlExpression := 'Text';
008      BindScope1.DataObject := FJSONArray;
009      BindList1.FillList;
010    end;
```

在这个事件处理函式我们希望所有显示的产品信息都以大写形式显示，因此我们在 006 行的系结表达式中呼叫系结引擎的全局 UpperCase 方法把 TJSONArray 中每一个 TJSONString 自动转换为大写的形式，下面是执行范例应用程序的结果：



TBindScope 提供了程序代码和组件之间动态绑定的能力，让开发人员可以藉由程序代码和系结表达式开发出更具动态能力的图形用户接口应用程序。

10-4 结论

本章讨论了更多有关实时数据系结框架的使用方法和技术，读者在阅读完本章的内容之后应该可以使用实时数据系结框架结合 FireMonkey 框架开发一般的应用程序并且系结到任何的数据源了，最后以下面表格总结本书讨论框架的使用目的:

框架	说明
dbExpress 框架	提供 Delphi/C++Builder 跨平台数据存取/处理能力
FireDAC 框架	提供 C++Builder/C++Builder 跨平台资料存取/处理能力，所有新的和资料存取有关的应用程序和 App 都应使用 FireDAC
FireMonkey 框架	提供 Delphi/C++Builder 跨平台图形用户接口能力
实时数据系结框架	提供 Delphi/C++Builder 跨平台数据系结能力

这三个框架应该是一般 Delphi/C++Builder 应用程序共同使用的跨平台框架，并且能够相互整合提供一致性的服务。

在未来即将推出的 Mobile Studio 中也将使用这 3 个框架来开发 iOS，Android 和其他移动平台的应用程序，因此这 3 个框架也将是未来数年中 Delphi 最重要的框架，开发人员必须完全的了解和掌握它们。

本书的范例程序请至下列网页下载：

http://embarcadero.gcomgroup.com.tw/download/dfd.zip

embarcadero
电话: +86 010 5332 2090
电邮: china.sales@embarcadero.com