
C++Builder FireDAC
資料庫開發手冊
embarcadero

# 序

FireDAC 應該算是 C++Builder 從 Borland 時代開始的第 3 代資料存取技術和框架了，C++Builder 歷經了 BDE/IDAPI，dbExpress 一直到現在的 FireDAC，這也代表了 C++Builder 從桌面開發，C/S，Web，Multi-Tier 轉變到現在著重跨平台和移動開發的需求演進。

FireDAC 是第一個完全由 C++Builder 程式語言撰寫的資料庫存取框架，以前的 BDE/IDAPI 和 dbExpress 是混合了 Object Pascal 和 C 語言撰寫的資料庫存取框架，因此隨著 C++Builder 程式語言支援多平台，FireDAC 也可以輕易的在多個平台中執行。但除了程式語言的原因之外，到底為什麼要使用 FireDAC 來取代 BDE/IDAPI 和 dbExpress 呢?



最主要的原因就是這 BDE/IDAPI，dbExpress 和 FireDAC 設計的目標和架構，BDE/IDAPI 在近 20 年前設計的目標是讓 C++Builder 在桌面和稍後出現的 C/S 架構中使用，而 dbExpress 設計的目標則是讓 C++Builder 除了能夠在原本的桌面和 C/S 架構中使用之外，也能夠在 Web 和 Multi-Tier 架構中使用。但隨著移動和穿戴式設備的出現和開發的需求，C++Builder 也需要一個能夠適用在所有平台的資料存取技術和框架，而 FireDAC 正好能夠滿足這個目標和需求。

FireDAC 的功能其實非常類似 BDE/IDAPI 和 dbExpress 的結合體，它在使用上非常接近 BDE/IDAPI，但又具備 dbExpress 的連線存取和離線資料處理的能力，再加上 FireDAC 不需要部署額外 DLL 檔案而能夠直接連結客戶端程式碼的特性以及精簡型資料集的功能，讓 FireDAC 也非常適合使用在移動和穿戴式設備的應用。因為如果您需要使用 C++Builder 開發任合需要處理資料的應用程式，那麼您絕對應該認真考慮使用 FireDAC。

本書的目的是希望讓讀者能夠快速學習和使用 FireDAC 來開發 C++Builder 的資料庫應用程式，希望在您閱讀完本書的內容之後就具備了足夠的知識和技術善用 FireDAC 開發出跨平台的資料庫應用程式。

# 目錄

本書的範例程式請至下列網頁下載：

http://embarcadero.gcomgroup.com.tw/download/cfd.zip

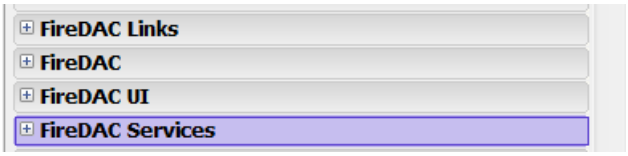
# FireDAC技術篇

# 第1章 開始學習使用FireDAC開發資料庫應用程式吧

FireDAC 是非常易於使用的資料存取技術和框架，但又蘊含了強大的功能。但高樓平地起，讓我們從如何使用 FireDAC 元件開發簡單的資料庫應用程式開始討論起吧。



在 TOKYO 中 FireDAC 提供了 4 類元件組：



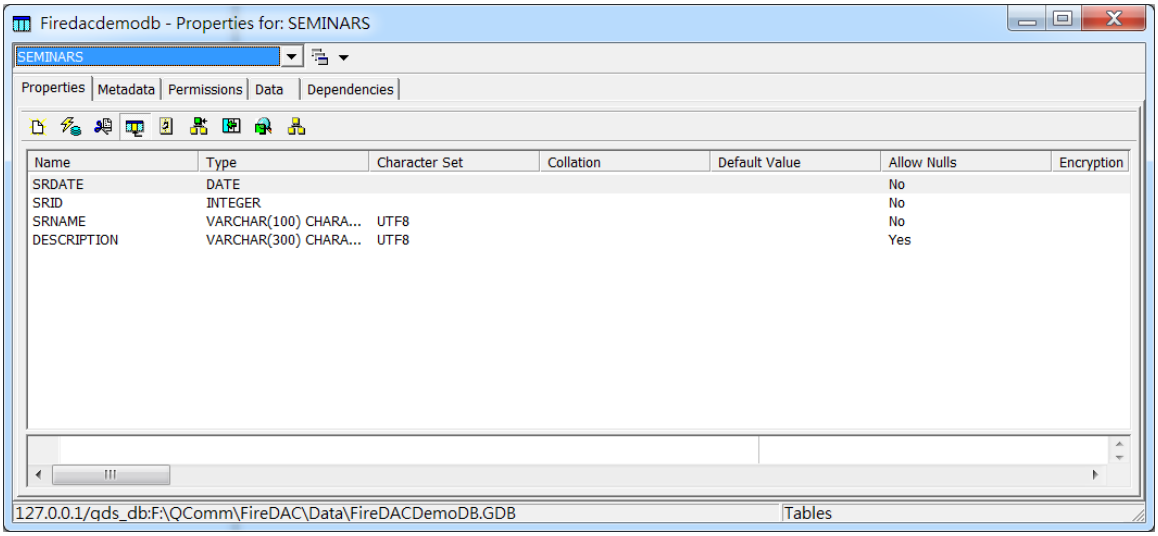

下面的表格簡單的說明了這 4 類元件組的功能：

| 名稱 | 說明 |
|---|---|
| FireDAC Links | 提供 FireDAC 呼叫各資料庫客戶端函式庫的元件，開發人員只需要使用這些元件即可連結各資料庫而不再需要部署額外的 FireDAC DLL(但仍需要部署各資料庫的客戶端函式庫)。此外本類組也包含了可監督連線狀況的元件。 |
| FireDAC | FireDAC 的核心元件，使用來連結和處理資料的元件 |
| FireDAC UI | 提供 FireDAC 處理資料時的相關 UI 元件 |
| FireDAC Services | 提供 FireDAC 額外的服務元件，主要是提供 SQLite 和 InterBase 的相關服務 |

學習 FireDAC 元件框架的第一步當然就是使用它連結資料庫和處理資料了。本書使用的資料庫主要是 InterBase，除了因為 InterBase 是 C++Builder 內附的資料庫之外，主要是因為 InterBase 可以使用在 Windows，iOS 和 Android 平台，這 3 個平台也是本書討論的主要平台。

# 1-1 使用 FireDAC 連結資料庫

本小節將使用 2 個 InterBase 範例資料表 Seminars 和 SeminarAttendee，因為這 2 個資料表之間有主從的關係，包含這 2 個資料表的範例資料庫則是 Firedacdemodb.GDB。下面是 Seminars 資料表包含的欄位資訊：

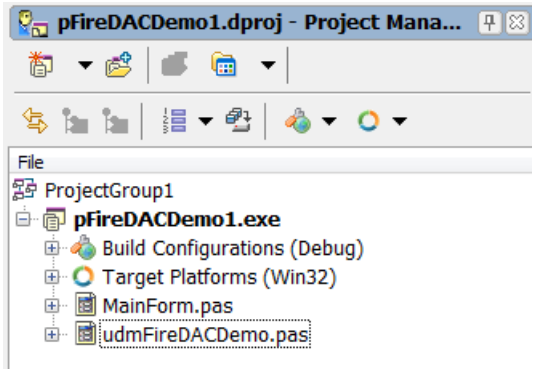



要使用 FireDAC 存取資料庫非常的簡單，基本上只需要使用 4 個 FireDAC 元件即可，下面的表格說明了本小節使用的 FireDAC 元件：

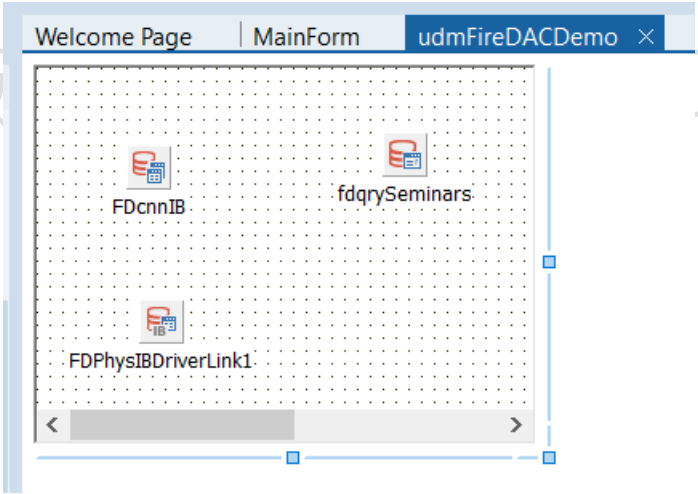
| 元件 | 名稱 | 說明 |
|---|---|---|
| | TFDConnection | 使用來連結資料庫的元件 |
| | TFDQuery | 使用來執行 SQL 命令的元件 |
| | TFDPhysIBDriverLink | 連結到 Interbase 的驅動程式元件 |
| | TFDGUIxWaitCursor | 控制 UI 等候游標的元件 |

先讓我們說明如何使用這 4 個元件連結並處理 Seminars 資料表再深入討論進階的用法。

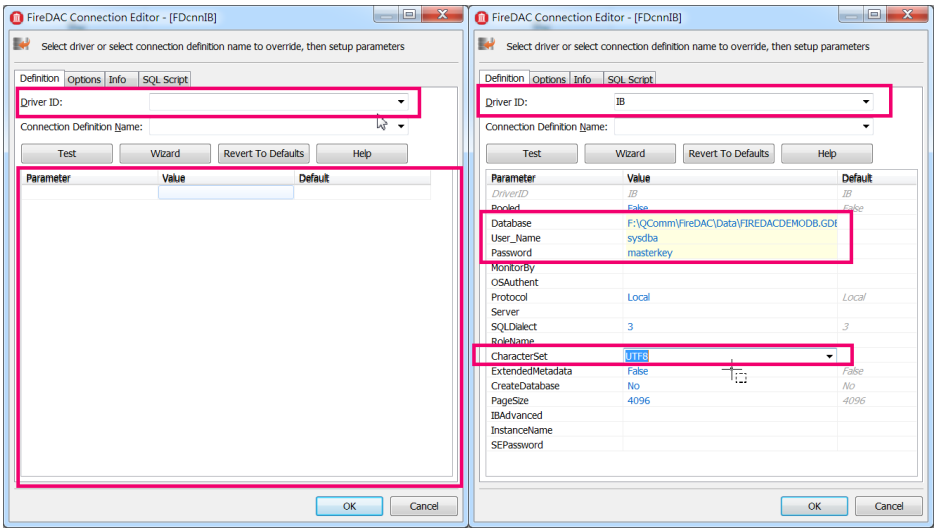
在 C++Builder IDE 中建立一個 FireMonkey Desktop Application 專案並且在其中建立一個資料模組，主表單以 MainForm 為名稱儲存，資料模組以 udmFireDACDemo 為名稱儲存，此時專案管理員如下：




在資料模組中放入 TFDConnection，TFDQuery 和 TFDPhysIBDriverLink 元件，設定 TFDConnection 的 Name 特性值為 FDcnnIB，TFDQuery 的 Name 特性值為 fdqrySeminars：

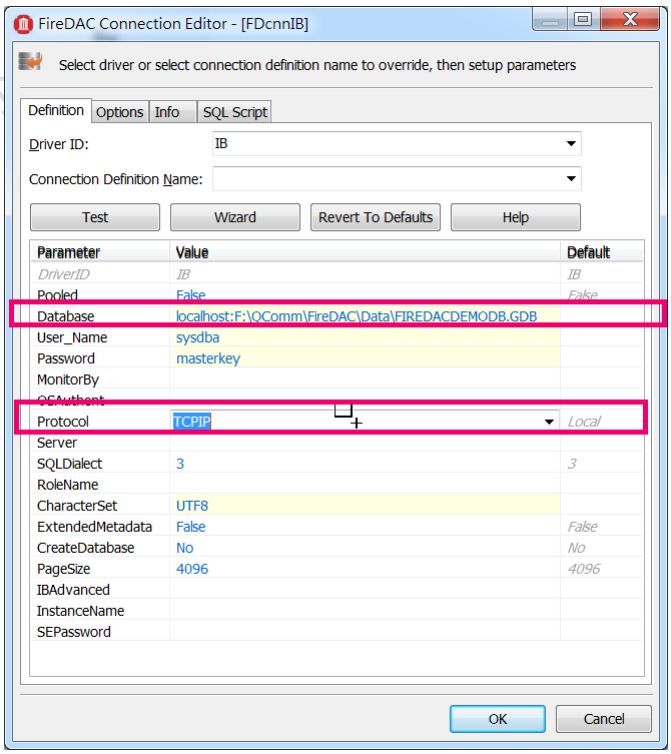



現在讓我們先使用 FDcnnIB 連結範例資料庫，請使用滑鼠雙擊 FDcnnIB 就會啟動 TFDConnection 的元件編譯器，此時元件編譯器如下左圖所示內容大都是空白的。由於我們要連結到 InterBase，因此請先在元件編譯器上方的 Driver ID 欄位中選擇 IB 代表要連結到 InterBase，一旦選擇之後元件編譯器就會出現連結 InterBase 需要設定的選項，例如 InterBase 資料庫所在位置，登入 InterBase 的使用者名稱和密碼，使用的連結通訊協定以及使用的字元集等。例如下面右圖顯示了筆者連結到本機的 InterBase 資料庫 F:\QComm\FireDAC\Data\FIREDACDEMODB.GDB，因此下面的 Protocol 欄位選擇了 Local：




當然如果筆者要連結到伺服器中的 InterBase 資料庫的話就可以如下選擇連結通訊協定為 TCPIP 並且在 InterBase 資料庫名稱之前加上伺服器位置，例如

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

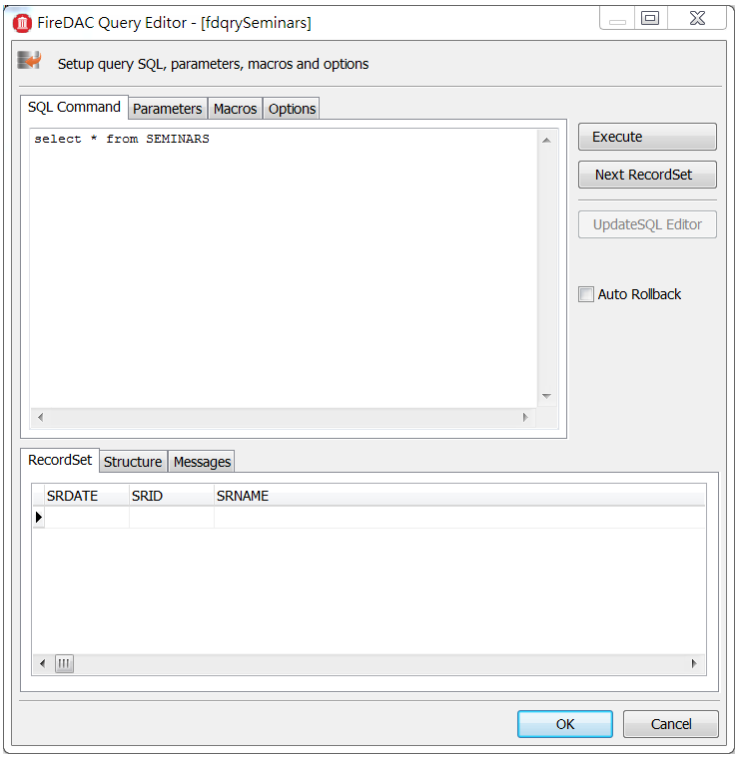



最後再設定 FDcnnIB 元件的 LoginPrompt 特性值為 False 免除每次開啟資料庫連結時都需要登入，現在就暫時完成了 FDcnnIB 的設定。

現在設定 fdqrySeminars 元件，請在物件檢視器中檢查 fdqrySeminars 的 Connection 特性值應該已經自動設定為 FDcnnIB 了，如果沒有的話請設定為 FDcnnIB。使用滑鼠雙擊 fdqrySeminars 元件啟動它的元件編譯器就會看到如下的畫面，在 SQL Command 頁次中輸入

```
Select * from SEMINARS
```

如果此時點選右方的『Execute』按鈕就可以在下方看到目前在 SEMINARS 資料表中的資料了，由於目前我們尚未在其中加入任何的資料因此現在是沒有資料的，點選 OK 按鈕以儲存此 SQL 命令到 fdqrySeminars 的 SQL 特性值中：




完成了 FDcnnIB 和 fdqrySeminars 的設定現在就可以在物件檢視器中把 FDcnnIB 的 Connected 設定為 True，再設定 fdqrySeminars 的 Active 為 True，這樣就可以連結 InterBase 範例資料表 Seminars 並且把其中的資料存取到範例程式了，或是在資料模組的 OnCreate 事件處理及式中撰寫如下的程式碼：

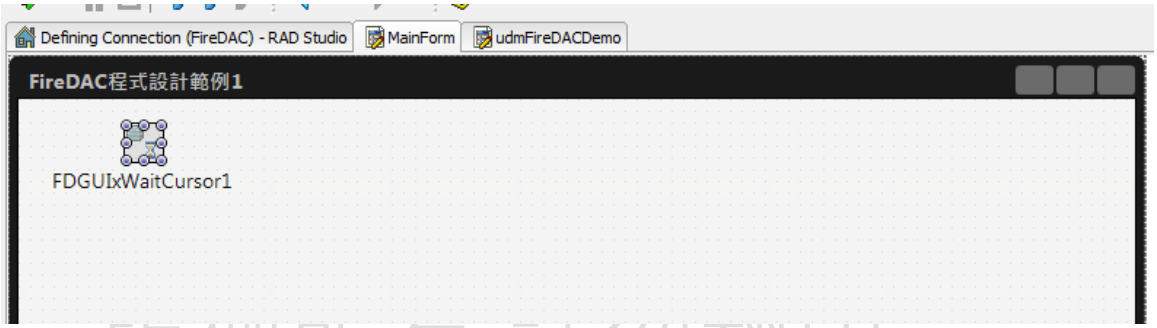
```
void __fastcall TdmFireDACDemo1::DataModuleCreate(TObject *Sender)
{
  FDcnnIB->Connected = true;
  fdqrySeminars->Active = true;
}
```

最後一定要在資料模組的 OnDestroy 事件處理及式中把 FDcnnIB 關閉：

```
void __fastcall TdmFireDACDemo1::DataModuleDestroy(TObject *Sender)
{
  FDcnnIB->Connected = false;
}
```

現在資料既然已經存取到客戶端的範例程式中了，我們自然可以開始把資料和 UI 連結起來了。

回到主表單中加入 TFDGUIxWaitCursor 元件，現在就可以使用這 4 個 FireDAC 元件來處理資料了。






要對 SEMINARS 資料表進行新增，修改，刪除和簡單資料查詢的工作只需要呼叫 TFDQuery 的 Insert()，Delete()，Edit()和 Locate()方法即可。例如要在 SEMINARS 資料表加入一筆新的資料，只需要使用如下的程式碼：

```
fdqrySeminars->Insert();
fdqrySeminars->FieldByName("SRNAME")->Value = L"測試";
fdqrySeminars->FieldByName("SRDATE")->Value = Now();
...
fdqrySeminars->Post();
```

在呼叫了 Insert()之後再呼叫 TFDQuery 的 FieldByName()方法藉由欄位名稱一一的把每一個欄位值填入最後再呼叫 Post()方法即可把新資料立刻加入到 SEMINARS 資料表中。

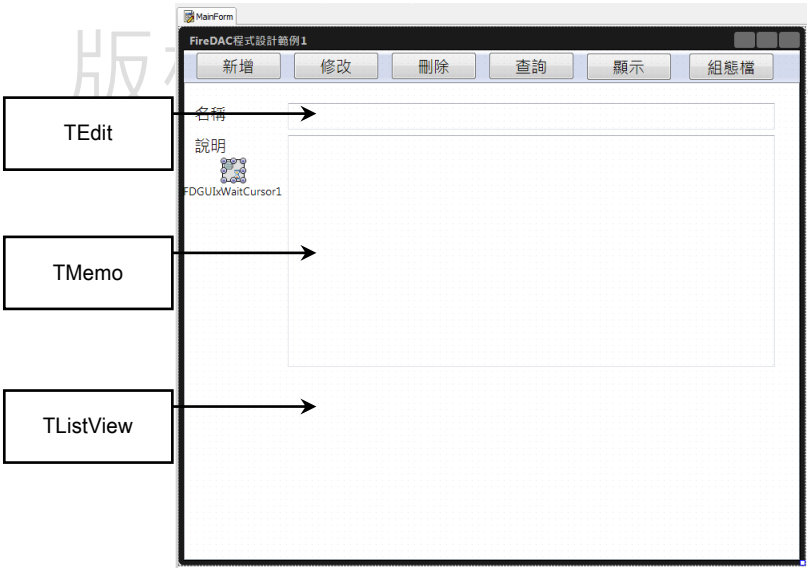
如果要修改資料的話需要先找到您要修改的資料記錄再呼叫 Edit()方法進入修改模式，然後也是藉由呼叫 FieldByName()方法把要修改的欄位值填入新的數值，最後也是再呼 Post()方法。

但要如何搜尋修要的資料呢？這可以藉由 Locate()方法，Locate()方法可以藉由搜尋特定的欄位值來尋找資料。例如如果我們要搜尋特定的研討會資料，那麼可以使用：

```
TLocateOptions Opts;
Opts.Clear();
…
if (fdqrySeminars->Locate("SRNAME", L"要尋找的研討會名稱", Opts))
{
  ...
}
```

Locate()方法會搜尋符合條件的第一筆資料，如果找到的話就回傳 true 沒找到就回傳 false，因此判斷 Locate()方法的回傳值就可以知道是否找到了需要搜尋的資料。

讓我們使用一個簡單的範例看看如何真的在 SEMINARS 資料表中對資料進行新增，修改，刪除和資料的工作。首先在 C++Builder IDE 中建立一個 Multi-Device Application 專案，在主表單中加入如下的 FireMonkey 元件：




主表單上的按鈕意思應該很清楚了，讓我們從如何顯示資料開始說明。

當我們設定 TFDQuery 元件的 Active 特性值為 true 或是呼叫它的 Open()方法後 TFDQuery 元件就會執行它的 SQL 特性值中 SQL 命令從資料表中取回的資料就稱為資料集(DataSet)，之後我們就可以呼叫下面的方法在此資料集中的資料進行移動和瀏覽的動作：

| 方法 | 說明 |
|---|---|
| First() | 到資料集中的第一筆資料 |
| Last() | 到資料集中的最後一筆資料 |
| Next() | 到目前下一筆資料 |
| Prior() | 到目前上一筆資料 |

在移動和瀏覽資料時又有下面 2 個特性可判斷資料集是否在最開始的位置和最後的位置：

| 特性 | 說明 |
|---|---|
| Bof | 目前在資料集起始的位置 |
| Eof | 目前在資料集結尾的位置 |

因此如果要在資料集中移動到每一筆資料的位置可以使用下面的程式碼樣板(以 fdqrySeminars 為範例)：

```
fdqrySeminars->First();
while (!fdqrySeminars->Eof)
{
  ....
  fdqrySeminars->Next();
}
```



當然您也可以反相移動：

```
fdqrySeminars->Last();
while (!fdqrySeminars->Bof)
{
  ....
  fdqrySeminars->Prior();
}
```

這正是主表單中『顯示』按鈕使用來顯示 SEMINARS 資料表中所有資料的方法，當點選『顯示』按鈕後它呼叫 DisplayAllSeminars()方法顯示所有資料。DisplayAllSeminars()方法先在 011 行呼叫 TFDQuery 元件的 GetBookmark()方法把目前資料的位置暫存下來，再於 015~021 行使用前面介紹的方法把 SEMINARS 資料表的 SRNAME 和 SRDATE 欄位顯示在 TListView 元件中，023~024 在顯示完資料後再回到原先資料的位置。

```
001    void __fastcall TfmMainForm::btnShowDataClick(TObject
```

```
*Sender)
002    {
003      DisplayAllSeminars();
004    }
005
006    void TfmMainForm::DisplayAllSeminars()
007    {
008      TListViewItem *lvi;
009
010      ListView1->Items->Clear();
011      TBookmark  bk =
dmBCBFireDACDemo1->fdqrySeminars->GetBookmark();
012      try
013      {
014        dmBCBFireDACDemo1->fdqrySeminars->First();
015        while (! dmBCBFireDACDemo1->fdqrySeminars->Eof)
016        {
017          lvi = ListView1->Items->Add();
018          lvi->Text =
dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRNAME")->Value;
019          lvi->Detail =
dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRDATE")->AsStrin
g;
020          dmBCBFireDACDemo1->fdqrySeminars->Next();
021        }
022      }
023      __finally
024      {
025        dmBCBFireDACDemo1->fdqrySeminars->GotoBookmark(bk);
026        dmBCBFireDACDemo1->fdqrySeminars->FreeBookmark(bk);
027      }
028    }
```



『新增』按鈕使用下面的程式碼在 SEMINARS 資料表中加入一筆資料：

```
void __fastcall TfmMainForm::btnInsertClick(TObject *Sender)
{
```

```
  dmBCBFireDACDemo1->fdqrySeminars->Insert();
  dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRDATE")->Value
= Now();
  dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRNAME")->Value
= edtSeminar->Text;
  dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRID")->Value =
GetSRIDFromDateTime(Now());

dmBCBFireDACDemo1->fdqrySeminars->FieldByName("DESCRIPTION")->Va
lue = mmSeminarDescription->Lines->Text;
  dmBCBFireDACDemo1->fdqrySeminars->Post();

  edtSeminar->Text = "";
  mmSeminarDescription->Lines->Text = "";
  DisplayAllSeminars();
}
```

『修改』按鈕先呼叫 **SearchSeminar()**方法使用 **Locate()**找到資料再呼叫 **Edit()**進入修改模式最後再呼叫 **Post()**把資料更新回資料表：



```
001    void __fastcall TfmMainForm::btnModifyClick(TObject
*Sender)
002    {
003      if (SearchSeminar(edtSeminar->Text))
004      {
005        dmBCBFireDACDemo1->fdqrySeminars->Edit();
006
  dmBCBFireDACDemo1->fdqrySeminars->FieldByName("SRNAME")->Value
= edtSeminar->Text;
007
  dmBCBFireDACDemo1->fdqrySeminars->FieldByName("DESCRIPTION")->
Value = mmSeminarDescription->Lines->Text;
008        dmBCBFireDACDemo1->fdqrySeminars->Post();
009      }
010    }
011
012    bool TfmMainForm::SearchSeminar(const String sSeminar)
```

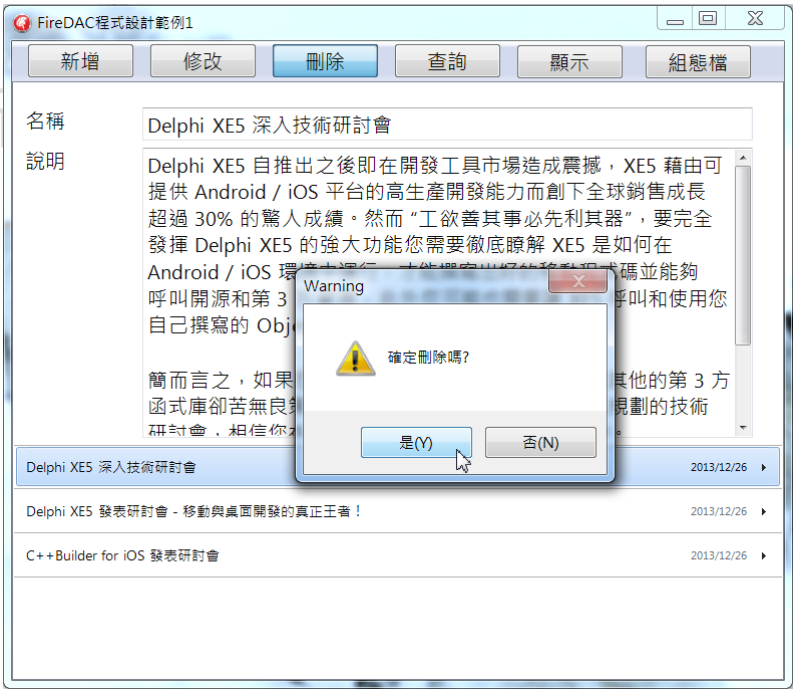
```
013    {
014      TLocateOptions Opts;
015      Opts.Clear();
016
017      Opts << loCaseInsensitive << loPartialKey;
018      return dmBCBFireDACDemo1->fdqrySeminars->Locate("SRNAME",
sSeminar, Opts);
019    }
```

最後的『刪除』按鈕也是使用 SearchSeminar()方法先找到要刪除的資料再呼叫 TFDQuery 元件的 Delete()方法從資料表中刪除資料：

```
001    void __fastcall TfmMainForm::btnDeleteClick(TObject
*Sender)
002    {
003      if (SearchSeminar(edtSeminar->Text))
004      {
005        int iResult = MessageDlg("確定刪除嗎?",
TMsgDlgType::mtWarning, TMsgDlgButtons() << TMsgDlgBtn::mbYes <<
TMsgDlgBtn::mbNo, 0);
006        if (iResult == mrYes)
007          dmBCBFireDACDemo1->fdqrySeminars->Delete();
008      }
009      DisplayAllSeminars();
010      ListView1ItemClick(ListView1,
ListView1->Items->Item[0] );
011    }
```



執行範例程式就可以看到下面的結果，的確可以在 EMINARS 資料表進行 CRUD 的工作了。


## 1-1-1 連結資料庫的方式

在前面說明如何連結到 InterBase 資料庫時我們時直接使用 TFDConnection 的元件編輯器來設定和 InterBase 的連結，但如果每個程式都這樣做就顯得麻煩。因此 FireDAC 另外提供了 2 種方式適合使用在多人環境中，例如主從架構環境，Web 或是多

層架構中。第一種是使用組態檔方式，這個方式也類似以前 BDE/IDAPI 和 dbExpress 的組態檔方式。這個概念是把連結資料庫的資訊先儲存到一個組態檔中，並且和程式一起分發這個組態檔或是把這個組態檔放在網路之中，那麼當 FireDAC 程式執行時先讀入這個組態檔，再依照組態檔的內容連結資料庫。

因此對照前面的範例，當開發人員在 IDE 中成功設定了資料庫連結之後可以把這個資料庫連結寫成組態檔，再把這個組態檔做為公共使用的組態檔即可。瞭解了組態檔概念之後就讓我們來說明如何使用它。

## 使用組態檔

組態檔就是一個文字檔，在您的 c:\Users\Public\Documents\RAD Studio\FireDAC\目錄下有一個範例 FireDAC 組態檔 FDConnectionDefs.ini，您可以依照其中的格式來撰寫或是修改成您需要的連結內容，例如其中就包含了連結到 InterBase 範例資料庫 Employee 的連結內容：

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



FDConnectionDefs.ini 是 FireDAC 內定使用的組態檔名稱，當 FireDAC 程式執行時定會在目前的執行目錄中尋找 FDConnectionDefs.ini，如果找不到的話就會在 Windows 系統註冊的

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

中尋找 FDConnectionDefs.ini，而

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

的預定值是

```
C:\Users\Public\Documents\RAD
```

```
Studio\FireDAC\FDConnectionDefs.ini.
```

當然使用 Windows 系統註冊來尋找組態檔並不是好方法，因為這就限定了您的 FireDAC 程式只能在 Windows 中執行，因此把組態檔放在目前的執行目錄中或是放在網路中特定的位置再讓 FireDAC 程式去尋找使用比較好。

此外把程式師在 IDE 中設定的連結寫成組態檔來使用也不錯。現在就讓我們來說明如何建立組態檔以及如何使用組態檔。

在前面的範例主表單中有一個『組態檔』按鈕，當您點選它之後它就會把 TFDConnection 元件連結 InterBase 的設定寫入一個組態檔以便分發給其他程式使用。

在 FireDAC 中要建立和使用組態檔都需要使用 FireDAC 的 TFDManager 元件。TFDManager 的 ConnectionDefFileName 特性可以讓您指定組態檔，而 ConnectionDefs 特性則可以讓您建立連結資訊。ConnectionDefs 特性是 IFDStanConnectionDefs 介面，其中的 AddConnectionDef()方法可以建立一個新的 FireDAC 連結資訊物件，在您設定了連結資訊物件的特性值之後呼叫它的 MarkPersistent()和 Apply()方法就可以儲存一個 FireDAC 連結資訊和組態檔了。



例如下面就是『組態檔』按鈕的實作程式碼，004 行設定組態檔位置和名稱，006 行先建立 IFDStanConnectionDef 介面物件 007 行設定這個組態名稱，08~013 設定連結資訊，014~015 儲存組態檔：

```
001    void TdmBCBFireDACDemo1::WriteMyConnectionDef()
002    {
003      if (! FDManager()->ConnectionDefFileLoaded)
004        FDManager()->ConnectionDefFileName = MYCONNECTIONFILE;
005
006      _di_IFDStanConnectionDef MyFireDACDef =
FDManager()->ConnectionDefs->AddConnectionDef();
007      MyFireDACDef->Name = MYCONNECTIONNAME;
008      MyFireDACDef->Params->DriverID = FDcnnIB->DriverName;
009      MyFireDACDef->Params->Database =
FDcnnIB->Params->Values["Database"];
010      MyFireDACDef->Params->UserName =
FDcnnIB->Params->Values["User_Name"];
011      MyFireDACDef->Params->Password =
FDcnnIB->Params->Values["Password"];
```

```
012     MyFireDACDef->Params->Values["Protocol"] =
FDcnnIB->Params->Values["Protocol"];
013     MyFireDACDef->Params->Values["CharacterSet"] =
FDcnnIB->Params->Values["CharacterSet"];
014     MyFireDACDef->MarkPersistent();
015     MyFireDACDef->Apply();
016   }
```

注意，由於每個在連結每種資料庫時不同的資料庫可能需要不同的設定，因此 IFDStanConnectionDef 介面只定義了大多數資料庫都會使用的設定，例如：

| 特性 | 說明 |
|---|---|
| UserName | 資料庫登錄名稱 |
| Password | 資料庫登錄密碼 |
| OSAuthent | 是否由 OS 認證 |
| Server | 資料庫伺服器位置 |
| Port | 資料庫伺服器使用的通訊埠 |
| DriverID | 資料庫驅動程式 ID |
| Pooled | 是否使用資料庫連結池 |



對於沒有列在 IFDStanConnectionDef 介面的資料庫設定，例如 InterBase 的 Protocol 和 CharacterSet 等，我們都可以藉由使用 IFDStanConnectionDef 的 Params 特性加入特定資料庫需要進行的特定設定。

下面是 MYCONNECTIONFILE 和 MYCONNECTIONNAME 的定義：

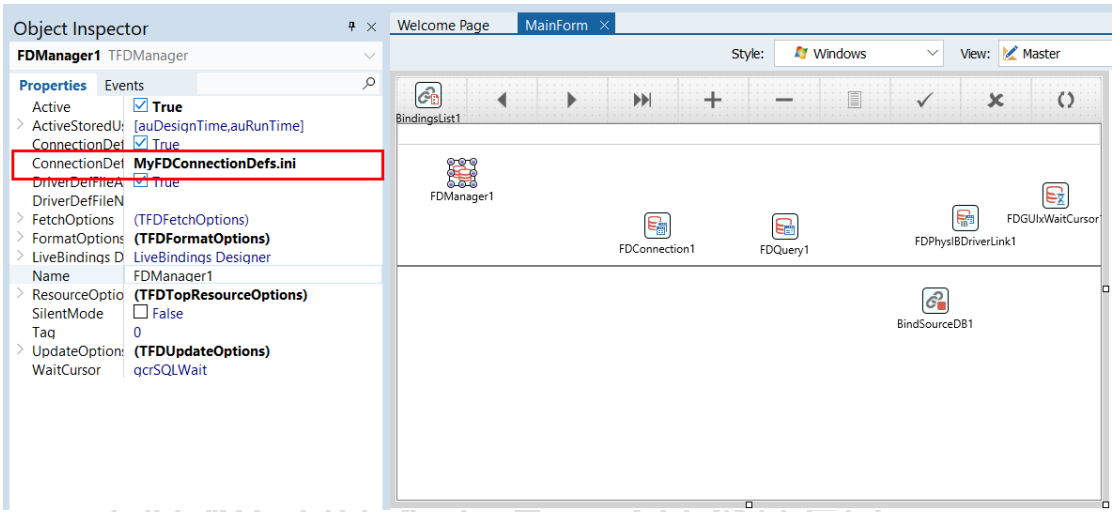
```
const String MYCONNECTIONFILE = ".\\MyFDConnectionDefs.ini";
const String MYCONNECTIONNAME = "MyIB_Connection";
const String DEMODBNAME = "FIREDACDEMODB.GDB";
```

執行上面的程式碼就可以在目前目錄下看到產生了 MyFDConnectionDefs.ini 組態檔，其中包含了下面的連結內容：

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

```
Protocol=TCPIP
CharacterSet=UTF8
```

現在建立一個新的 FireMonkey Desktop Application 專案，在主表單中加入如下的元件，請注意在其中使用了 TFDManager 元件，並且在物件檢視器中於它的 ConnectionDefFileName 特性中輸入 MyFDConnectionDefs.ini，代表要使用執行目錄下 MyFDConnectionDefs.ini 這個組態檔中的連結資訊來連結資料庫：






接著在主表單的 OnShow 事件處理函式中開啟 TFDManager，TFDConnection 和 TFDQuery 元件，並且在 TFDConnection 的 BeforeConnect 事件處理函式中設定它要使用組態檔中 MYCONNECTIONNAME 類別的連結資訊：

```
const String MYCONNECTIONFILE = ".\\MyFDConnectionDefs.ini";
const String MYCONNECTIONNAME = "MyIB_Connection";

//---------------------------------------------------------------------
__fastcall TfmMainForm::TfmMainForm(TComponent* Owner)
  : TForm(Owner)
{
}
//---------------------------------------------------------------------
void __fastcall TfmMainForm::FormShow(TObject *Sender)
{
  FDManager1->Active = true;
  FDConnection1->Connected = true;
```

```
  FDQuery1->Active = true;

}
//---------------------------------------------------------------------

void __fastcall TfmMainForm::FormDestroy(TObject *Sender)
{
  FDConnection1->Connected = false;
  FDManager1->Active = false;
}
//---------------------------------------------------------------------

void __fastcall TfmMainForm::FDConnection1BeforeConnect(TObject
*Sender)
{
  FDConnection1->ConnectionDefName = MYCONNECTIONNAME;
}
```



現在如果執行此範例程式就可以看到如下的執行畫面，FireDAC 元件果然使用組態檔連結到 InterBase 了。


當然要記得把 MyFDConnectionDefs.ini 組態檔和範例程式放在同一個目錄中。

## 1-1-2 直接使用程式碼

第 2 種方式是直接把如何連結資料庫的方式用程式碼直接寫在程式中，不過如此一來如果資料庫設定改善的話就要改程式碼，這在多人使用的環境中並不實際，不過這種方式適合使用在移動平台中，因為移動平台都有沙盒的概念，資料庫部署和連結方式都是固定的，因此使用這種方式很適合。在稍後我們就可以看到使用 FireDAC 開發移動平台資料庫 App 的範例，不過在這裡先讓我們說明如何使用這種方式連結資料庫。

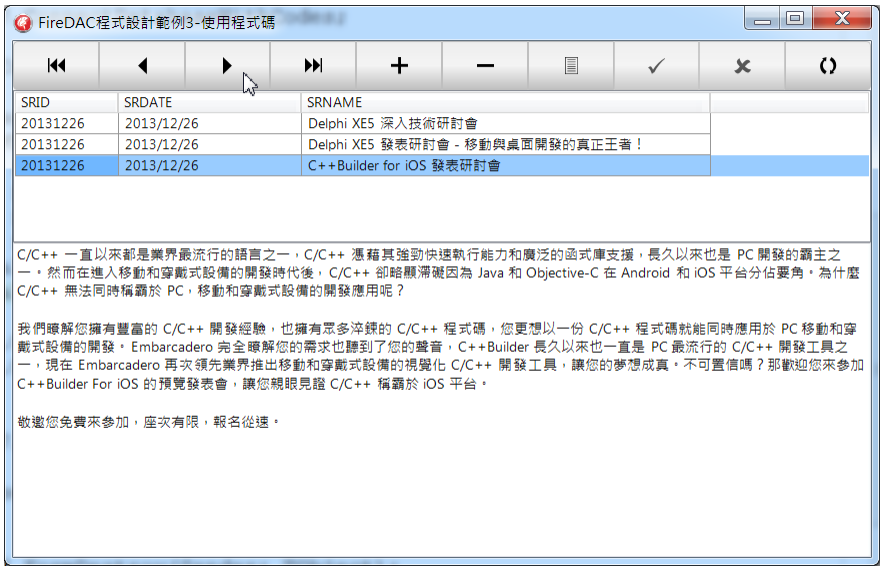
本書的第 3 個範例使用了程式碼的方式連結資料庫，在主表單的 OnCreate 事件處理函式中呼叫了 ConnectDatabaseWithCodes() 連結 InterBase 資料庫，ConnectDatabaseWithCodes()方法也是先在 011 行藉由 TFDManager 建立一個 IFDStanConnectionDef 介面物件，在 012~018 行設定連結資料庫的特性值，019 行呼叫 IFDStanConnectionDef 介面的 Apply()方法把這些設定寫入到 TFDManager 中，最後 020 行設定 TFDConnection 元件使用 011 行設定的連結名稱之後再於 021 行連結資料庫：

```
001    const String MYCONNECTIONNAME = "MyIB_Connection";
002
003    void __fastcall TfmMainForm::FormCreate(TObject *Sender)
004    {
005      ConnectDatabaseWithCodes();
006    }
007
008
//---------------------------------------------------------------
009    void TfmMainForm::ConnectDatabaseWithCodes()
010    {
011      _di_IFDStanConnectionDef ibDef =
FDManager1->ConnectionDefs->AddConnectionDef();
012      ibDef->Name = MYCONNECTIONNAME;
013      ibDef->Params->DriverID = "IB";
014      ibDef->Params->Database =
"127.0.0.1:E:\\QComm\\FireDAC\\Data\\FIREDACDEMODB.GDB";
015      ibDef->Params->UserName = "sysdba";
016      ibDef->Params->Password = "masterkey";
017      ibDef->Params->Values["Protocol"] = "TCPIP";
018      ibDef->Params->Values["CharacterSet"] = "UTF8";
```

```
019     ibDef->Apply();
020     FDConnection1->ConnectionDefName = MYCONNECTIONNAME;
021     FDConnection1->Connected = true;
022   }
023
024   void __fastcall TfmMainForm::FormDestroy(TObject *Sender)
025   {
026     FDConnection1->Connected = false;
027   }
028
//---------------------------------------------------------------------
029
030   void __fastcall TfmMainForm::FormShow(TObject *Sender)
031   {
032     FDConnection1->Connected = true;
033     FDQuery1->Active = true;
034   }
```



現在執行這個範例就可以看到如下的執行畫面，證明使用程式碼也可以直接連結資料庫：



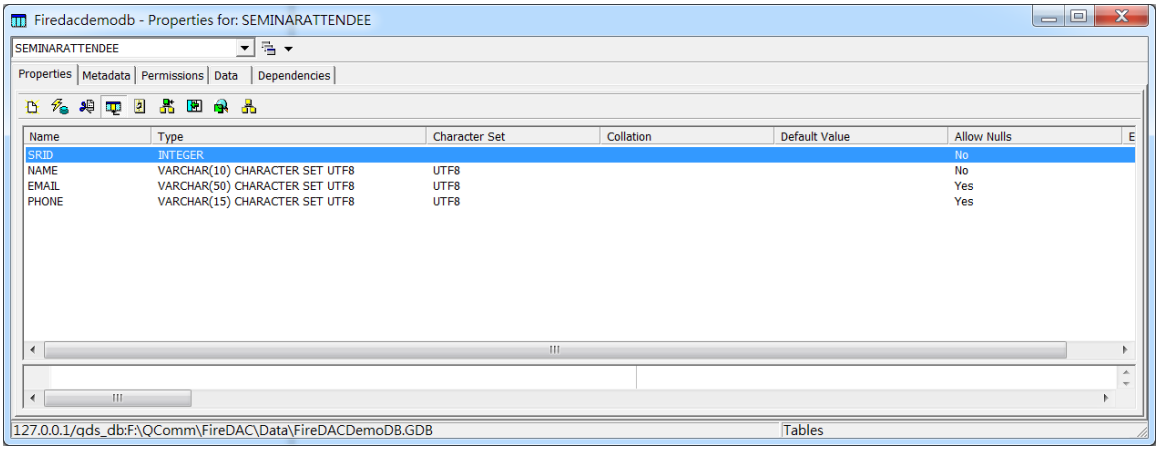

如同前面說明的，使用程式碼連結資料庫特別適合使用在移動平台，稍後就會看到使用的範例和說明。

# 1-2 處理資料

在前一小節已經說明了如何使用 FireDAC 進行單資料表的 CRUD 的工作，在這一小節讓我們再多討論一點使用 FireDAC 處理資料的方式，首先先說明如何處理關連 2 個以上的資料表。

## 1-2-1 主從關連資料

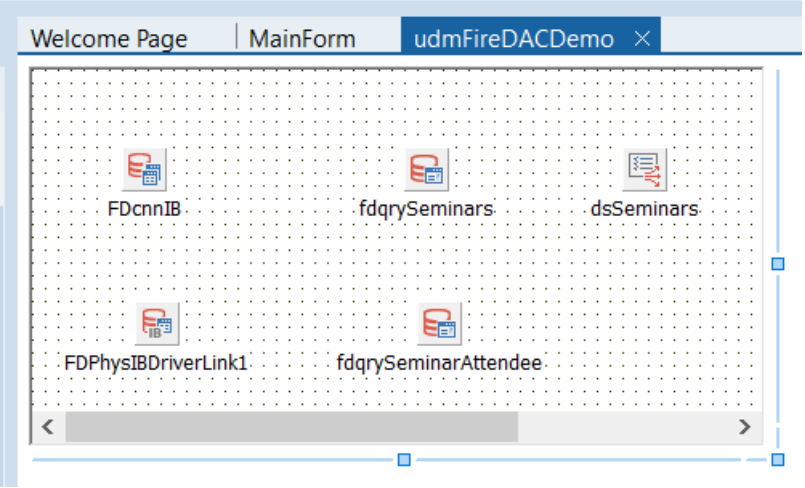
為了本小節說明之用，讓我們加入第 2 個資料表 SeminarAttendee，它和前面的 Seminar 資料表擁有主從的關係，這 2 個資料表是藉由 SRID 欄位關連在一起：




FireDAC 提供了非常方便的方法讓開發人員處理主從關連資料，比 BDE/dbExpress 都方便，而且 FireDAC 提供了數種方法處理主從關連資料。

### 1-2-1-1 使用客戶端範圍機制

第一種方式非法的簡單，其原理是主資料表和從資料表的資料從資料庫中取回客戶端，再於客戶端藉由 FireDAC 的分段範圍機制當主資料表中的資料移動位置時再篩選從資料表中符合的資料，現在讓我們用一個範例來說明，先建立一個 FireMonkey Desktop Application 專案，在其中建立一個資料模組於其中使用下列的 FireDAC 元件從範例 Seminars 資料中存取資料：




在上面的資料模組中 fdqrySeminars 在它的 SQL 特性中使用了

```
select * from SEMINARS
```

從 Seminars 資料表中擷取資料，而 fdqrySeminarAttendee 在它的 SQL 特性中使用了

```
select * from SEMINARATTENDEE
```

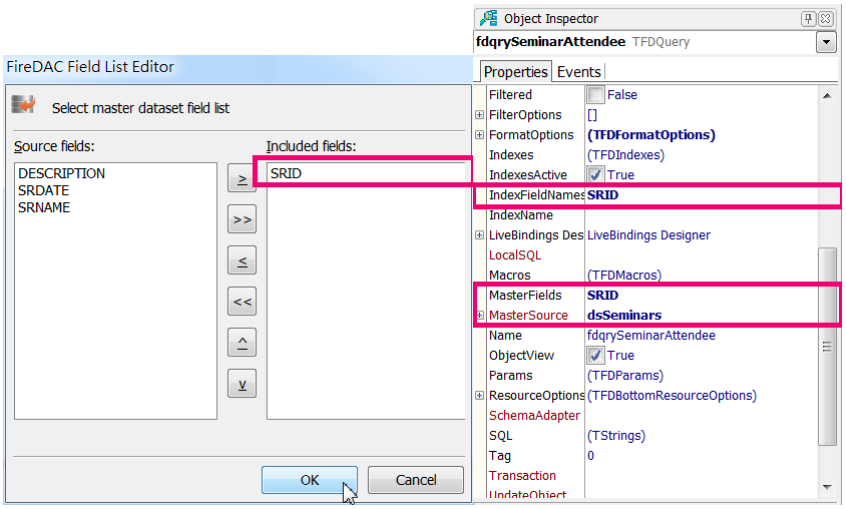


從 SeminarAttendee 資料表中擷取資料，fdqrySeminars 和 fdqrySeminarAttendee 之間的主從關係是藉由資料模組中的 dsSeminars 這個 TDataSource 元件繫結的。

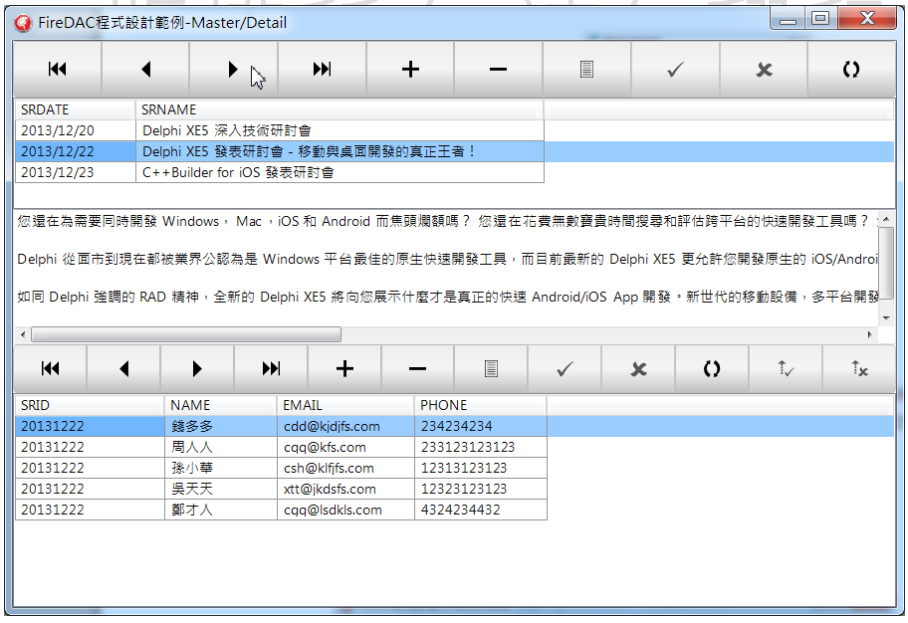
先設定 dsSeminars 的 DataSet 特性值為資料模組中的 fdqrySeminars，接著在物件檢視器中設定 fdqrySeminarAttendee 如下的特性值：

| 特性 | 特性值 |
|---|---|
| MasterDataSource | dsSeminars |
| MasterFields | SRID |
| IndexFieldNames | SRID |

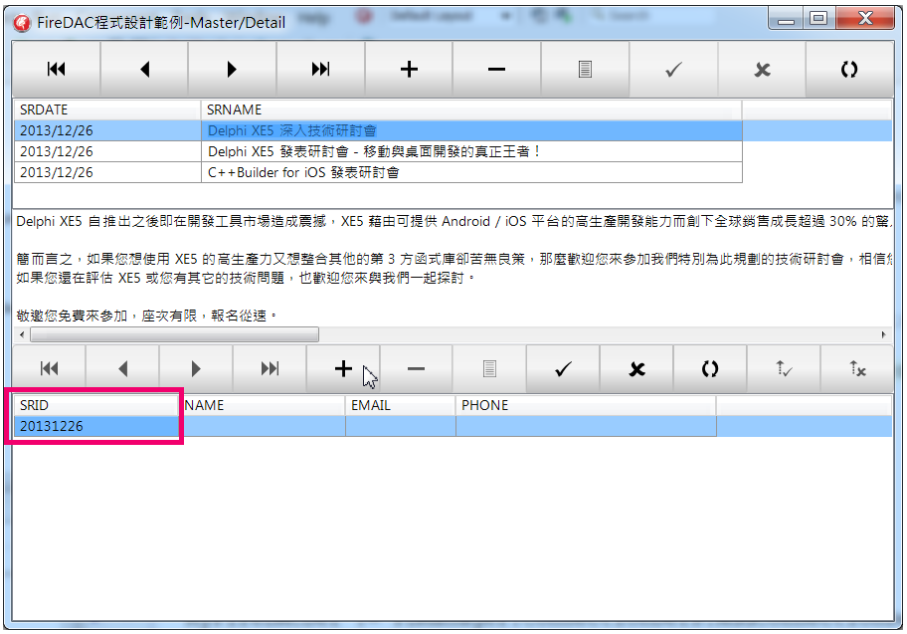
如下所示：




上面設定的意義肩簡單，就是 fdqrySeminarAttendee 之中的資料要根據 fdqrySeminars 資料表中 SRID 欄位值的資料篩選，只有在 fdqrySeminarAttendee 資料表的 SRID 欄位中擁有相同數值的資料才被篩選出來。現在如果執行這個範例程式並且在 SeminarAttendee 資料表中新增一些資料，那麼在瀏覽 Seminars 資料表資料時就可以看到下方的 SeminarAttendee 資料表的資料也會跟著改變：




而且當您在 SeminarAttendee 資料表中新增資料時 SeminarAttendee 資料表的 SRID 欄位會自動新增為目前 Seminars 資料表的 SRID 欄位的數值，如下所示：




使用這種方式處理主從資的問題是如果從資料表中的資料很大量的話會導致客戶端從資料庫存取大量的資料到客戶端，消耗大量網路和客戶端記憶體的資源。因此在從資料表有大量資料的應用中您應該考慮下一個方式。

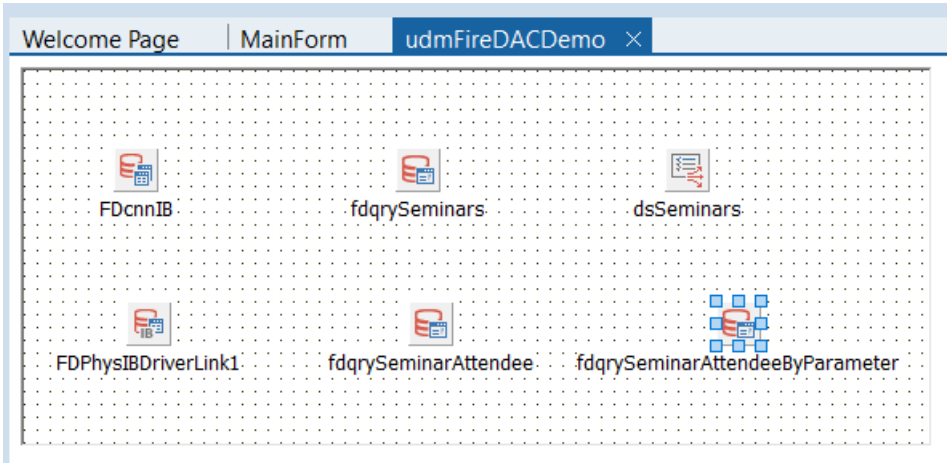


## 1-2-1-2 使用伺服端動態查詢機制

第 2 種方式是使用動態查詢的機制，為從資料表寫一個包含參數的 SQL 命令再把主資料表和從資料表關連起來，如此一來當主資料表中的資料移動時就會自動把關連的欄位值帶入資料表的 SQL 命令參數中再進行查詢的動作。

現在在資料模組中放入一個名為 fdqrySeminarAttendeeByParameter 的 TFDQuery 元件，在它的 SQL 特性中使用：

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




上面的:SRID 就是動態參數，這個參數的名稱和主資料表中關連的欄位相同，因此在完成了稍後的設定之後主資料表就會在每次移動資料位置時自動把新的 SRID 欄位值填入這個 SQL 命令並且執行它。

接著在物件檢視器中設定 fdqrySeminarAttendeeByParameter 如下的特性值：

| 特性 | 特性值 |
|---|---|
| MasterDataSource | dsSeminars |
| MasterFields | SRID |



再次執行就可以得到和上一個方式一樣的結果。

使用動態查詢機制的問題是每次主資料表的資料移動時就需要執行一次 SQL 命令，如果客戶端的數量太多會造成資料庫龐大的執行負荷，因此這種方式不適合使用在擁有大量客戶端的應用中。

那麼如果您擁有大量的從資料表資料又擁有大量的客戶端怎麼辦呢？這不困難，請結合這 2 個方法，再結合後面章節說明的快儲機制(Cache)即可。

要結合這 2 方法就是

1. 使用動態參數 SQL 命令
2. 設定從資料的 TFDQuery 的 MasterDataSource，MasterFields 和 IndexFieldNames 這 3 個特性值，如同 2-1-1 小節說明的
3. 設定從資料表的 FetchOptions.Cache 特性值包含 fiDetails

如何一來當一開始從資料表尚未有資料時 FireDAC 就會使用 2-1-2 小節的方式，但當主資料表的位置改變時，先前從資料表中的資料不會被丟棄而會被快儲下來，如果主資料表的位置稍後又回到這個位置那麼快儲下來的從資料表資料就可以再被使用而無需再執行一次 SQL 命令了。

稍後的討論快儲機制的章節會有範例說明。

# 1-3 開發移動資料庫 App

FireDAC 是目前 C++Builder 提供的 3 種資料存取技術（BDE/IDAPI，dbExpress 和 FireDAC）中最適合使用來開發移動 App 的，因為

- FireDAC 無需部署額外的 DLL，
- FireDAC 可提供又小又快的執行速度
- FireDAC 直接支援移動平台的本地資料庫：InterBase ToGo 和 SQLite
- FireDAC 可提同時連線和離線資料處理的功能



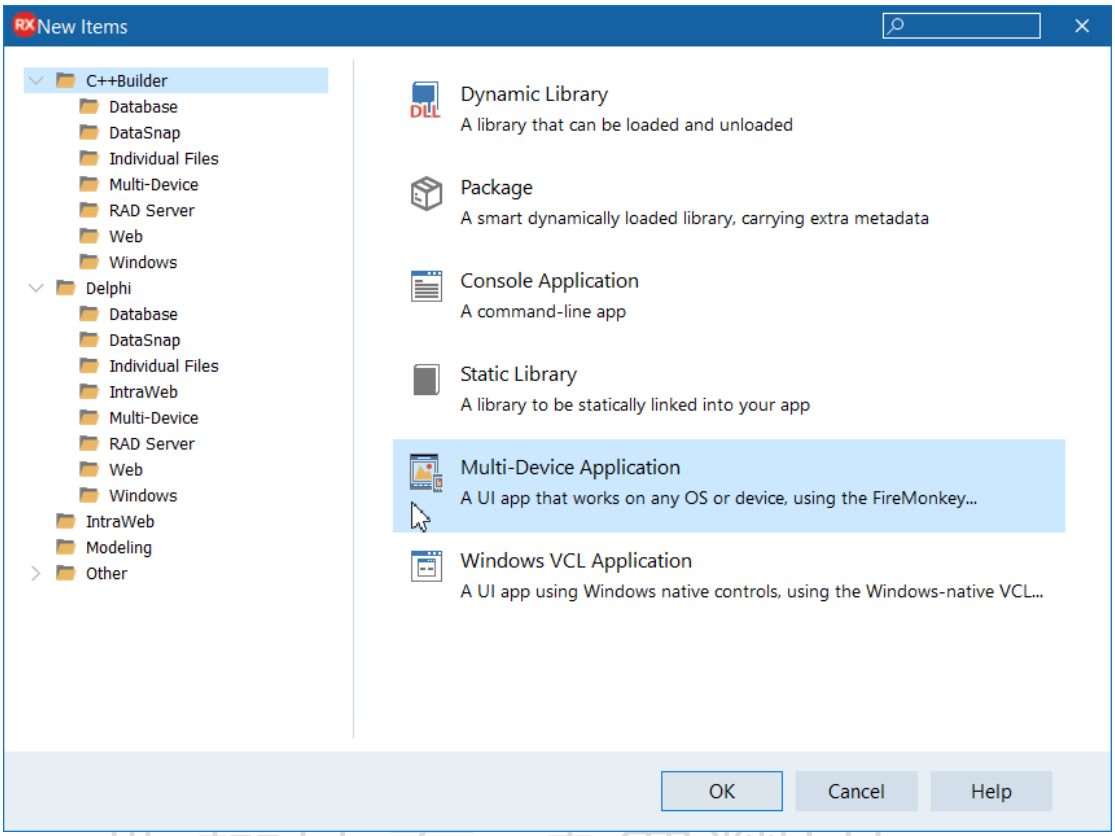
不過如果您想讓移動平台和後端的資料庫伺服器連結的話，那麼您必須配合使用 DataSnap，在 TOKYO 的版本中由於安全的因素 FireDAC 尚不支援直接從移動平台連結後端的資料庫伺服器。

本小節將說明如何使用 FireDAC 開發 iOS/Android 平台的資料庫 App，不過本小節說明的是存取 iOS/Android 手機中的本地資料庫，並不會說明如何藉由 DataSnap 連結後端的資料庫伺機器，同時本小節也是使用 InterBase ToGo 做為說明的資料庫，當然您也可以使用 SQLite。

## 1-3-1 開發和部署 iOS/Android 手機 App

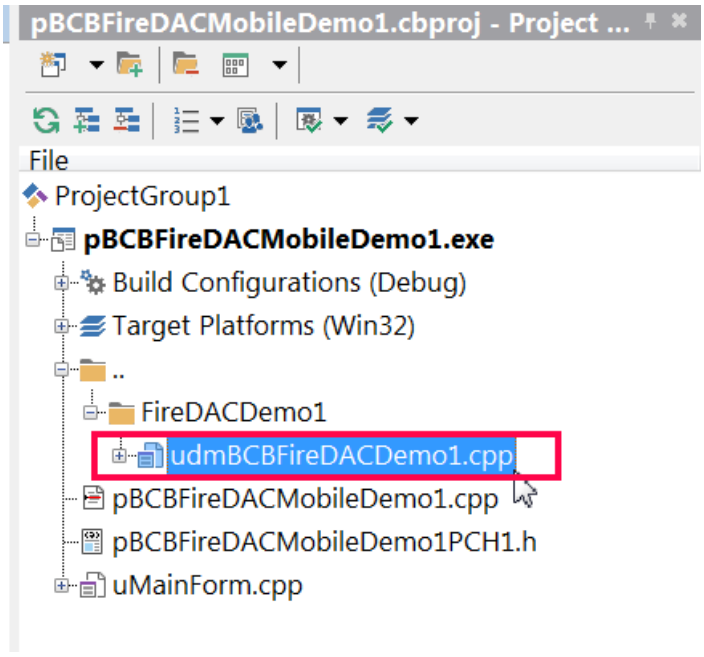
先讓我們使用前面第 1 個範例 FireDAC 應用程式做為開發和部署就明的範例，現在我們要說明的就是如何把第 1 個範例 FireDAC 程式部署到 iOS 和 Android 平台中，由於第 1 個範例使用了資料模組來存取資料，而資料模組和平台 UI 無關因此可以重覆使用在移動平台中。

首先建立一個 FireMonkey Mobile Application 專案：






儲存專案並且把第 1 個範例的資料模組加入專案：



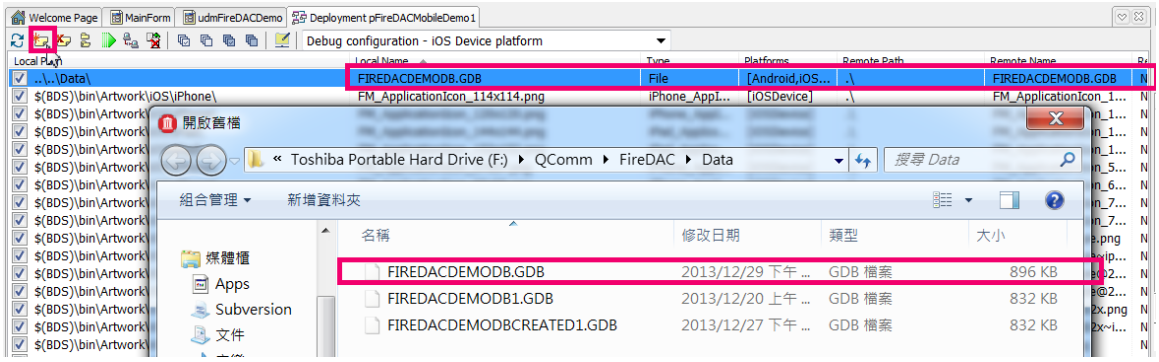

要開發手機上的資料庫 App 有 2 種方式：

1. 部署 InterBase 或 SQLite 資料庫到手機中

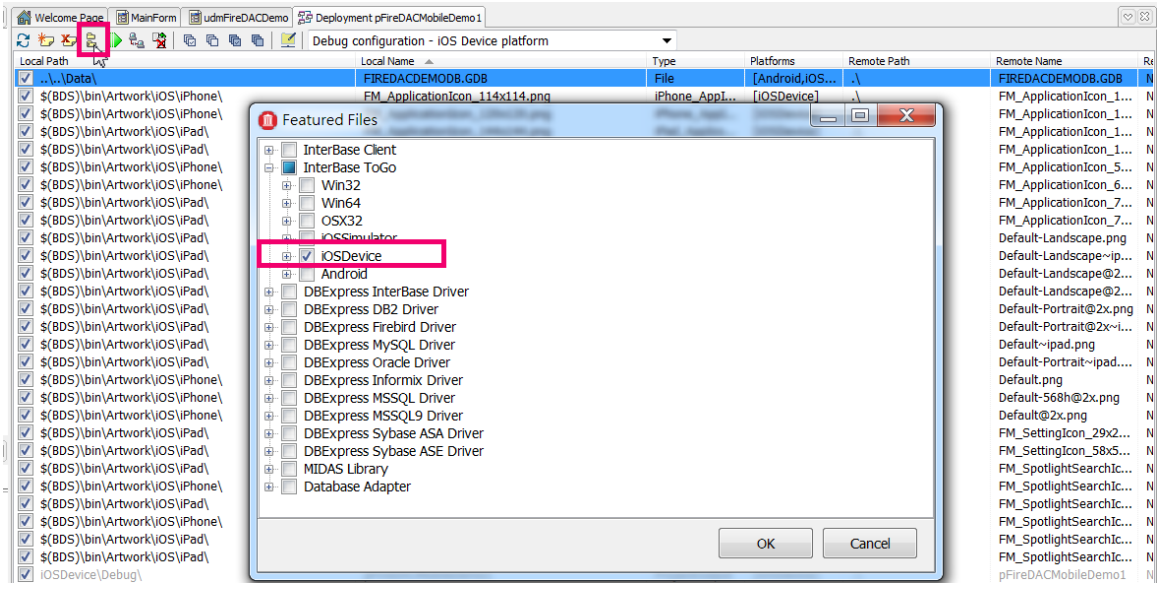
2. 直接使用程式碼在手機中建立需要使用的資料庫

當然如果是結合 DataSnap 開發分散式手機資料庫 App，那就有第 3 種方式，本小節將先說明第 1 種方式。

現在先讓我們部署前面使用的範例資料庫 FIREDACDEMODB.GDB 到手機中，讓我們先開發 iOS App，請在專案管理員中雙擊 Target Platforms 節點中的 iOS Device 節點，再於 IDE 中點選 Project | Deployment 啟動部署精靈，如下圖點選部署精靈，先點選部署精靈左上方的『Add Files』按鈕加入範例資料庫 FIREDACDEMODB.GDB，如下所示：



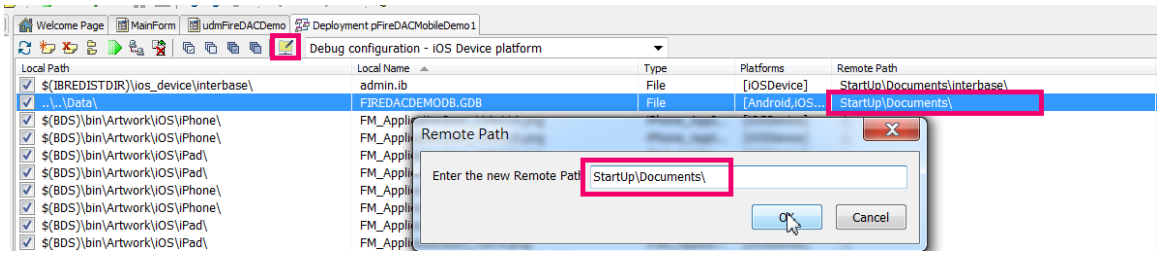

再點選部署精靈左上方的『Add Featured Files...』按鈕加入 InterBase ToGo iOS 平台的相關檔案，如下所示：



由於移動平台都有沙盒的概念。因此必須把範例資料庫和 InterBase ToGo 的授權檔部署到移動平台正確的目錄中，對於 iOS 平台範例資料庫需要部署到

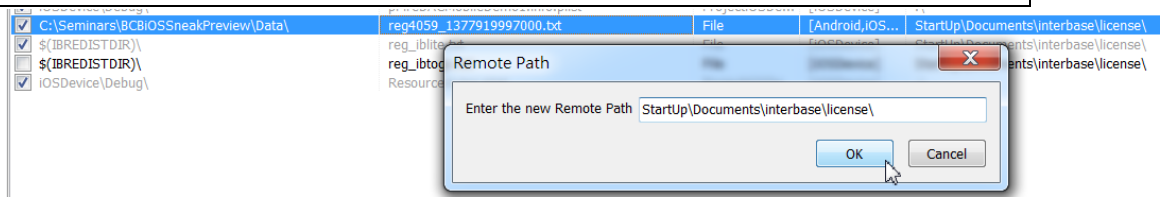
```
StartUp\Documents\
```

因此選部署精靈中的『Change Remote Path for Selected Items』按鈕設定範例資料庫到 StartUp\Documents\：




InterBase ToGo 的授權檔部署到

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



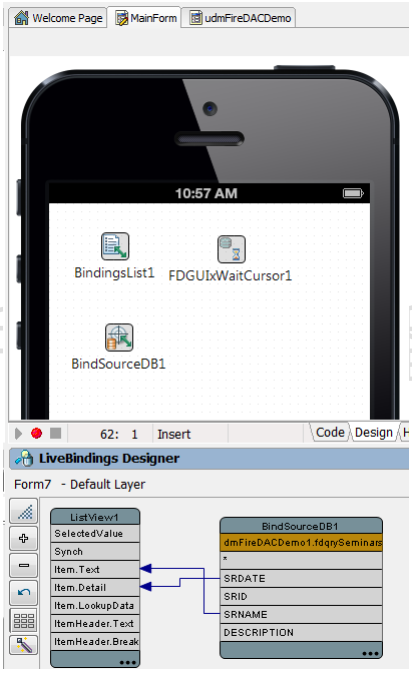

接著當然要修改 InterBase 資料庫的所在地，要根據應用程用程式執行的平台從不同的地方連結或是載入。請開啟資料模組為其中的 TFDConnection 建立如下的 OnBeforeConnect 事件處理函式：

```
void __fastcall TdmBCBFireDACDemo1::FDcnnIBBeforeConnect(TObject
*Sender)
{
  FDcnnIB->Params->Values["Database"] = "";
#if defined(TARGET_OS_IPHONE) || defined(TARGET_IPHONE_SIMULATOR)
|| defined(__ANDROID__)
  FDcnnIB->Params->Values["Database"] =
IncludeTrailingPathDelimiter(
      System::Ioutils::TPath::GetDocumentsPath()) + DEMODBNAME;
  FDcnnIB->Params->Values["Protocol"] = "Local";
#else
```

```
  FDcnnIB->Params->Values["Database"] =
"127.0.0.1:e:\\QComm\\FireDAC\\Data\\" + DEMODBNAME;
#endif
}
```

在 FDcnnIBBeforeConnect 中我們使用 C++Builder 的編譯器指令判斷目前的執行平台，如果是 iOS 或 Android 平台就藉由 TPath 類別的 GetDocumentsPath()方法就可以取得前面用部署精靈的部署目錄了。

回到專案的主畫面放入 TFDGUIxWaitCursor 元件，再使用 Live Visual Bindings 功能連結範例資料庫 FIREDACDEMODB.GDB 中的 Seminars 資料表：




編譯之後就可以看到這個範例專案成功執行在 iOS 手機中了，而且顯示了前面 Windows 版範例程式加入的資料：


現在看看如何部署到 Android 平台，由於 Android 的沙盒目錄不同因此現請先在專案經理中點選 Android 節點再開啟部署精靈，如下圖把範例資料庫需要部署到

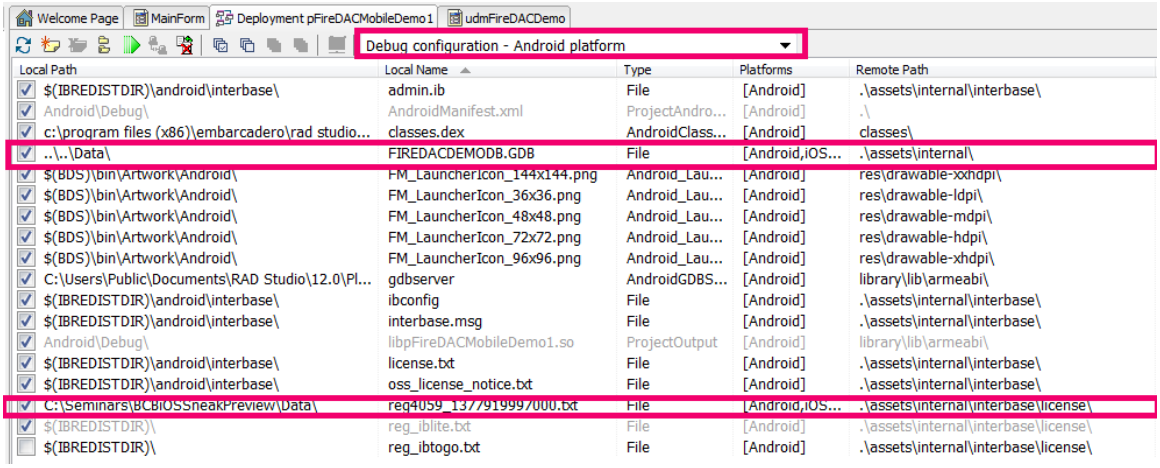
```
.\assets\internal\
```

InterBase ToGo 的授權檔部署到

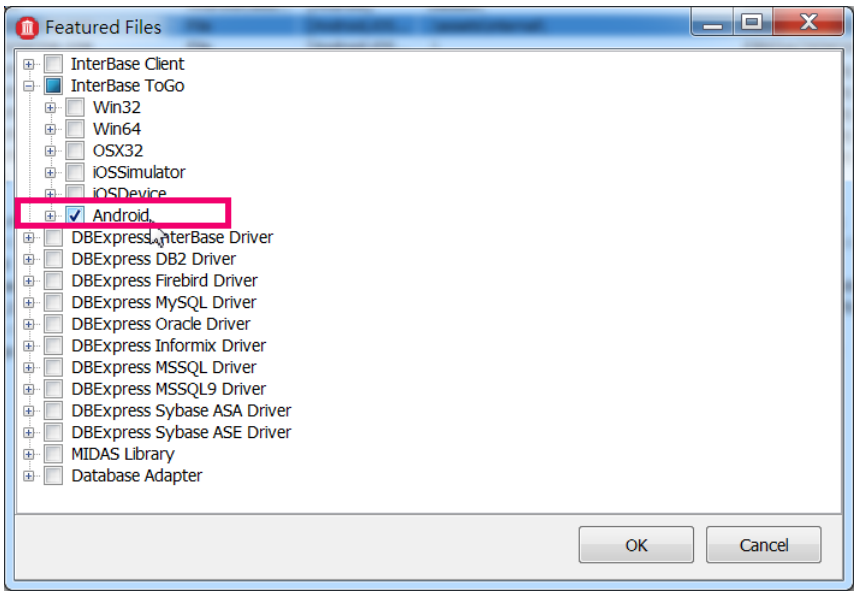


```
.\assets\internal\interbase\license\
```

如下圖所示：




記得加入 Android 的相關支援檔案：



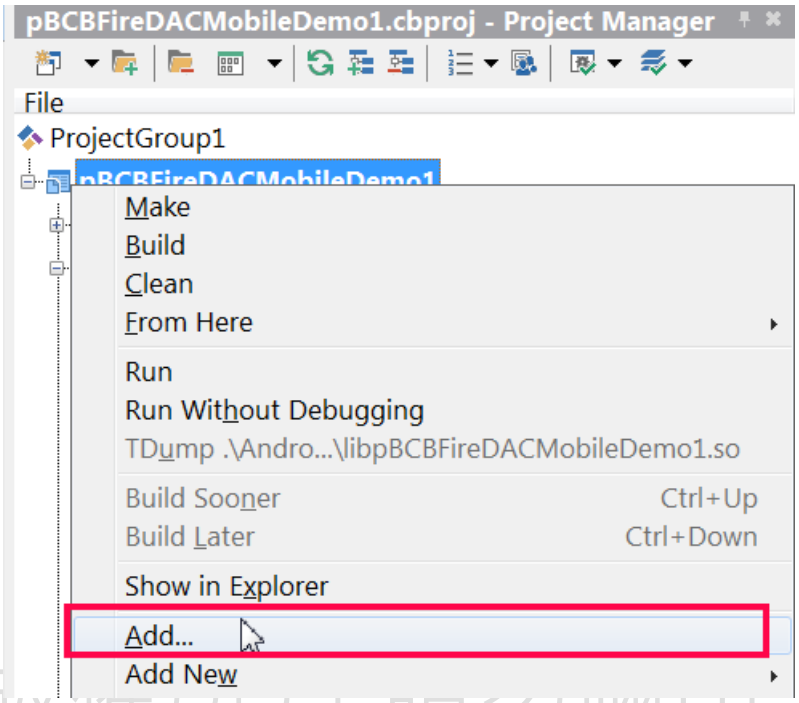

編譯之後就可以看到這個範例專案成功執行在 Android 手機中了：




在前面說明的使用 C++Builder IDE 中的部署精靈以及 iOS 和 Android 平台使用不同的路徑”StartUp\Documents\”和”.\assets\internal\”是為了讓讀者瞭解整個部署的流程和原理。在讀者了解之後那麼我們可以接著說明 TOKYO 對於部署功能的強化，在下面讓我們使用 SQLite 資料庫來說明。

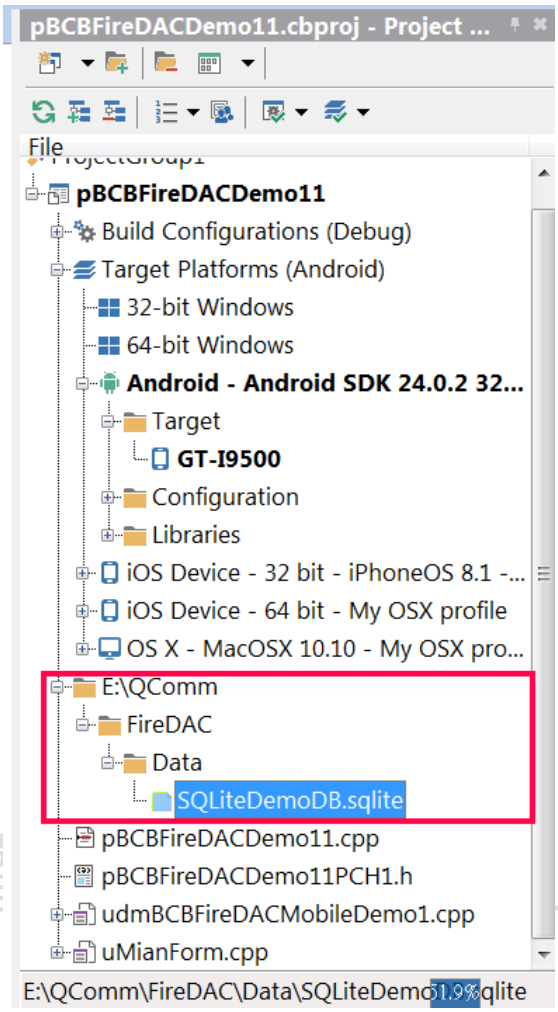
在 TOKYO 之後當開發人員要部署額外的檔案到 iOS/Android 平台時可以直接在 IDE 的專案管理員中加入要部署的額外檔案即可，IDE 便會自動判斷額外檔案的類別，如果部署的額外檔案需要額外的函式庫，例如 InterBase，那麼 IDE 也會自動把需要的函式庫打包到 App。現在讓我們使用一個 SQLite 資料庫”SQLiteDemoDB.sqlite”來說明。

例如如果在前面使用 InterBase 的範例中我們改用 SQLite 資料庫”SQLiteDemoDB.sqlite”，那麼在 TOKYO 中要部署 SQLiteDemoDB.sqlite,請在專案管理員中右擊滑鼠，點選 Add…選項：



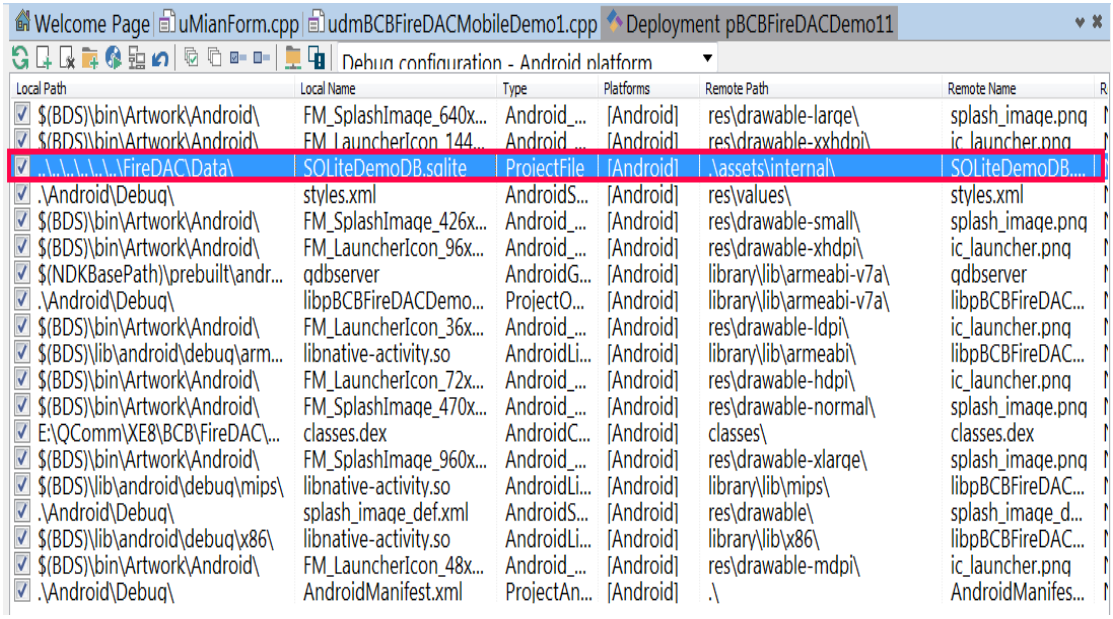



然後在檔案對話盒中選擇加入 SQLiteDemoDB.sqlite，那麼就可以看到 SQLiteDemoDB.sqlite 出現在專案管理員中：






此時砸點選 Project|Deployment 啟動部署精靈就可以看到 SQLiteDemoDB.sqlite 自動正確的設定要部署到 Android 的”.\assets\internal\”路徑：




由於 iOS/Android 手機中已內建安裝了 SQLite 的函式庫，因此 C++Builder 不再需要部署這些函式庫，所以在部署精靈中我們可以看到只需要部署 SQLiteDemoDB.sqlite 檔案即可，當然我們只需要修改 TFDConnection 使用的資料庫檔案為 SQLiteDemoDB.sqlite：

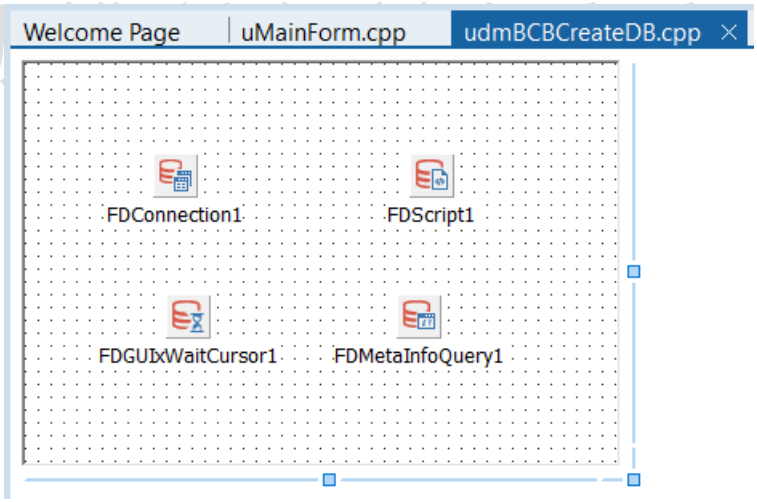


```
const String DEMODBNAME = "SQLiteDemoDB.sqlite";
void __fastcall TdmBCBMobileDemo::FDcnnIBBeforeConnect(TObject
*Sender)
{
#if defined(TARGET_OS_IPHONE) || defined(TARGET_IPHONE_SIMULATOR)
|| defined(__ANDROID__)
  FDcnnIB->Params->Values["Database"] =
IncludeTrailingPathDelimiter(
      System::Ioutils::TPath::GetDocumentsPath()) + DEMODBNAME;
#else
  FDcnnIB->Params->Values["Database"] = DEMODBNAME;
#endif
}
```

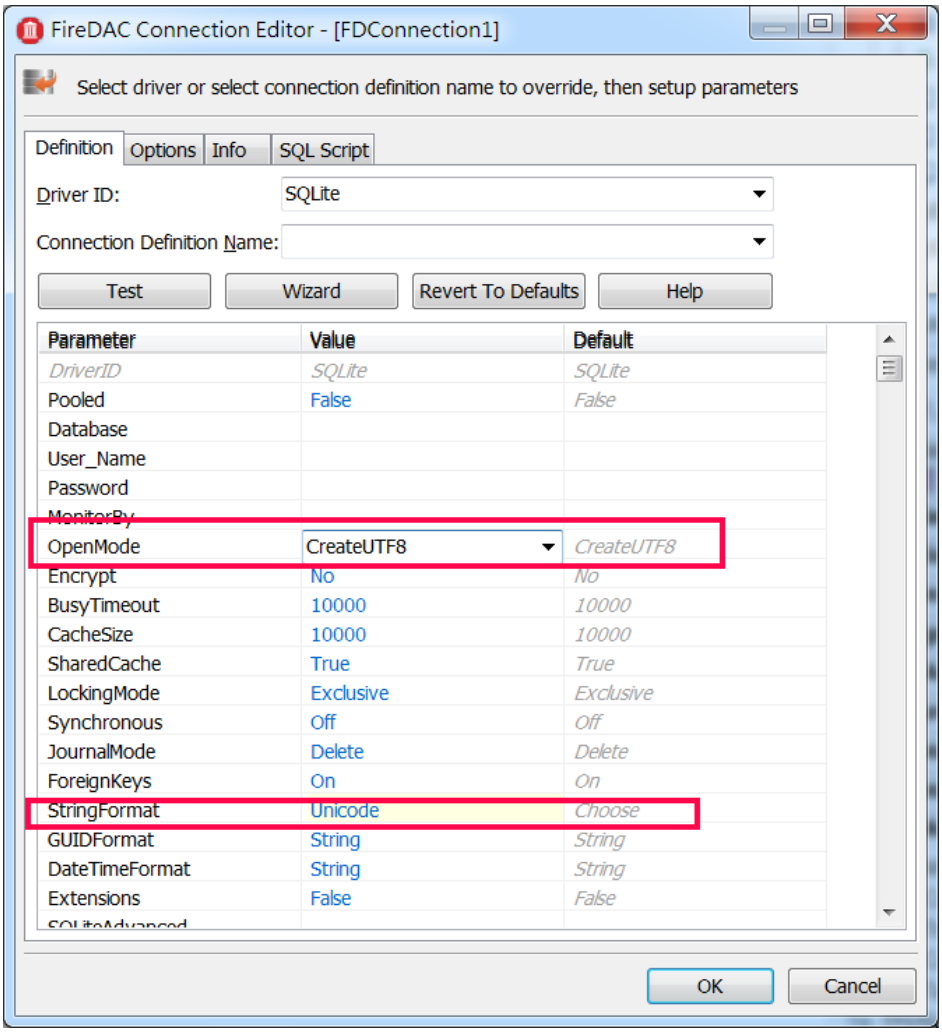
而下面的畫面顯示了這個使用 SQLite 資料庫在 Android 手機中成功執行的結果：


## 1-3-2 直接在 iOS/Android 手機中建立資料庫

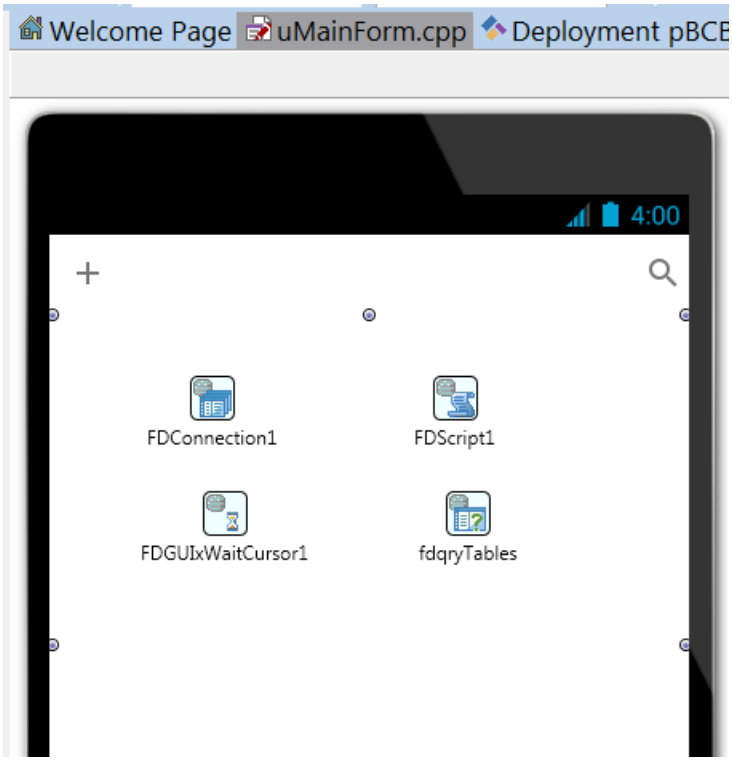
如果覺得根據不同的平台部署資料庫很麻煩，那可以選擇使用程式碼的方式直接建立手機平台中的資料庫。例如您可以建立一個新的 FireMonkey Mobile Application 專案，在其中建立一個資料模組並且在其中放入 TFDConnection，TFDMetaInfoQuery，TFDPhysIBDriverLink 以及 TFDScript 元件：




接著雙擊 TFDConnection 元件啟動元件編譯器，設定 OpenMode 為 CreateUTF8 讓 TFDConnection 元件可自動建立 SQLite 資料庫，再設定 StringFormat 為 UniCode：




在主表單中加入 ToolBar 和 2 個 TButton 元件，TListView 元件以及 TFDGUIxWaitCursor：




當點選 ToolBar 左邊的按鈕時呼叫資料模組中的 CreateDatabase()方法建立資料庫，再呼叫 SetupDatabase()設定資料庫：



```
void __fastcall TFfmMainForm::btnCreateDBClick(TObject *Sender)
{
  CreateSQliteDB();
  SetupDatabase();
}
```

GetDatabase 方法使用 DDL 建立資料庫並且藉由 TFDScript 元件執行此 DDL 建立範例資料庫以及其中的範例資料表 SEMINARS 和 SEMINARATTENDEE：

```
const String DEMODBNAME = "FIREDACBCBDEMODB.sqlite";

String TFfmMainForm::GetDatabase()
{
  String sResult = "";
#if defined(TARGET_OS_IPHONE) || defined(TARGET_IPHONE_SIMULATOR)
|| defined(__ANDROID__)
  sResult =
```

```
IncludeTrailingPathDelimiter(System::Ioutils::TPath::GetDocument
sPath()) + DEMODBNAME;
#else
  sResult = "127.0.0.1:E:\\QComm\\FireDAC\\Data\\" + DEMODBNAME;
#endif

  return sResult;
}
//---------------------------------------------------------------
void TFfmMainForm::OpenAndAutoCreateDB(const String sDB)
{
  dmBCBCreateDB->FDConnection1->Params->Values["Database"] = sDB;
  dmBCBCreateDB->FDConnection1->Connected = true;
}
//---------------------------------------------------------------
void TFfmMainForm::CreateSQliteDB()
{
  String sDatabase = GetDatabase();
  if (FileExists(sDatabase))
  TFile::Delete(sDatabase);
  OpenAndAutoCreateDB(sDatabase);

  TStringList *sl = new TStringList();
  try
  {
  sl->Add("CREATE TABLE \"SEMINARATTENDEE\" ");
  sl->Add("(");
  sl->Add("\"SAID\"   INTEGER PRIMARY KEY AUTOINCREMENT  NOT NULL
UNIQUE ,");
  sl->Add("\"NAME\"   VARCHAR,");
  sl->Add("\"EMAIL\"  VARCHAR,");
  sl->Add("\"PHONE\"  VARCHAR");
  sl->Add("\"ADATE\"  DATETIME");
  sl->Add("\"SRID\"   INTEGER");
  sl->Add(");");
  sl->Add("CREATE TABLE \"SEMINARS\"");
```

```
  sl->Add("(");
  sl->Add("\"SRDATE\" DATETIME,");
  sl->Add("\"SRID\"   INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL
UNIQUE,");
  sl->Add("\"SRNAME\" VARCHAR,");
  sl->Add("\"DESCRIPTION\" VARCHAR");
  sl->Add(");");
  Memo1->Lines->Text = sl->Text;
  dmBCBCreateDB->FDScript1->ExecuteScript(sl);
  }
  __finally
  {
  delete sl;
  }
}
```

**SetupDatabase()**方法則使用前面介紹的使用程式碼的方式設定和連結範例資料庫：



```
void TFfmMainForm::SetupDatabase()
{
#if defined(TARGET_OS_IPHONE) || defined(TARGET_IPHONE_SIMULATOR)
|| defined(__ANDROID__)
  dmBCBCreateDB->FDConnection1->Params->Values["Database"] =
IncludeTrailingPathDelimiter(
      System::Ioutils::TPath::GetDocumentsPath()) + DEMODBNAME;
#else
  dmBCBCreateDB->FDConnection1->Params->Values["Database"] =
DEMODBNAME;
#endif
}
```

主表單加上方的”開啟資料庫”按鈕呼叫 **DisplayTables()**方法擷取並顯示動態建立的 **FIREDACBCBDEMODB.sqlite** 資料庫中的 2 個資料表以證明我們成功的使用 DDL 建立了資料表：

```
void __fastcall TFfmMainForm::btnOpenDBClick(TObject *Sender)
```

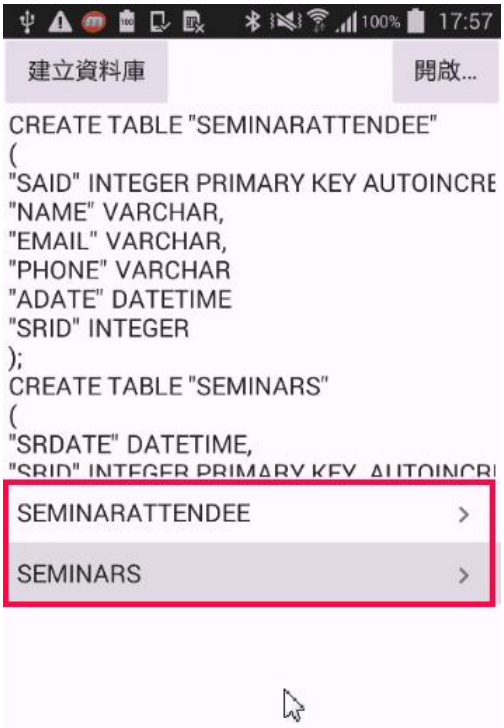
```
{
  DisplayTables();
}
```

DisplayTables()方法使用 TFDMetaInfoQuery 元件取得資料庫中所有的資料表並顯示在主表單中的 TListView 元件中(在本書稍後的章節會說明 FireDAC 的 MetaData 功能)：

```
void TFfmMainForm::DisplayTables()
{
  try
  {
  dmBCBCreateDB->FDMetaInfoQuery1->Active = true;

  while (! dmBCBCreateDB->FDMetaInfoQuery1->Eof)
  {

AddMessage(dmBCBCreateDB->FDMetaInfoQuery1->Fields->operator[](3
)->Value);
    dmBCBCreateDB->FDMetaInfoQuery1->Next();
  }
  }
  __finally
  {
  dmBCBCreateDB->FDMetaInfoQuery1->Active = false;
  }
}
```



編譯之後就可以看到這個範例專案成功執行在 iOS/Android 手機中並且顯示範例資料庫中的範例資料表而無需使用部署精靈先根據不同的平台部署資料庫了。






# 1-4 結論

本章先介紹如何使用 FireDAC 元件連結資料庫和處理簡易的資料，下一章會討論更多使用 FireDAC 處理資料的技術。

# 第2章 處理資料

本章要說明如何使用 FireDAC 來處理資料，例如搜尋，快儲機制，非同步處理資料等實用的功能，在您閱讀完本章內容之後就能夠使用很有效率的方式在應用程式中處理資料了。

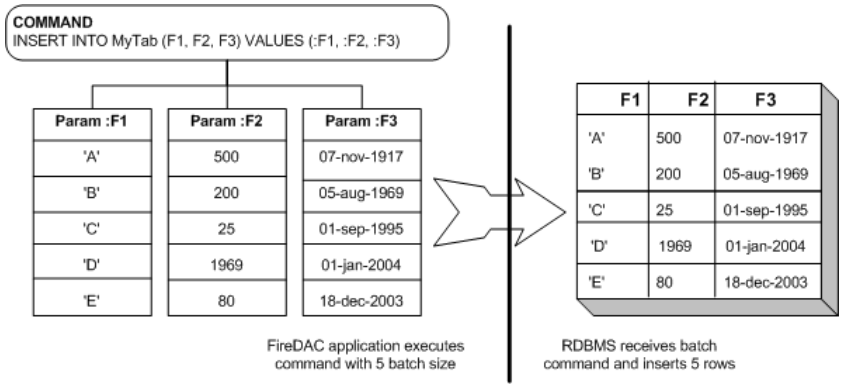
## 2-1 使用 Array ＤＭＬ處理大量資料



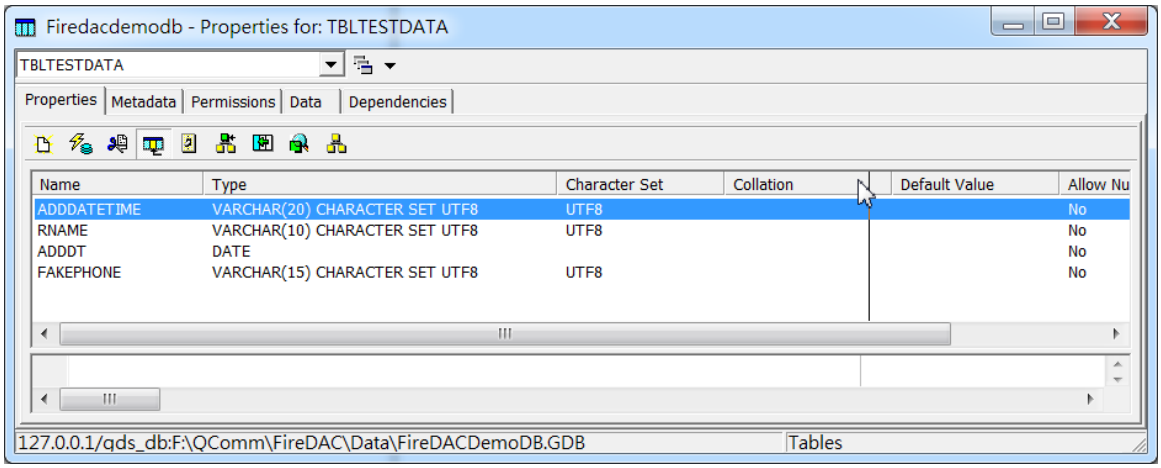
由於本章需要使用大量的隨機資料做為說明本章的範例，因此需要在範例資料表中新增大量的隨機資料，因此正好藉由這個需求先來說明如何使用 FireDAC 快速有效率的在應用程式中處理大量的資料。

FireDAC 提供了所謂的陣列資料處理功能(Array Data Manipulation Language, Array DML)來處理應用程式的大量資料需求。例如需要大量新增資料或是大量修改資料的應用中。Array DML 可以想成批次處理的概念，在一般的應用中如果客戶端要新增大量的資料，那麼客戶端需要為每一筆資料呼叫一次 Insert 和 Post 方法，或是執行一次 SQL 的 Insert 指令。但如果客戶端想一次加入 10000 筆資料那這需要執行 1000 次，如此一來速度當然不好也會造成網路頻繁的負荷。因此 Array DML 的概念是在客戶端一次把 10000 筆資料送到資料庫並且讓資料庫一次執行完畢，如此一來速度當然就快多了。

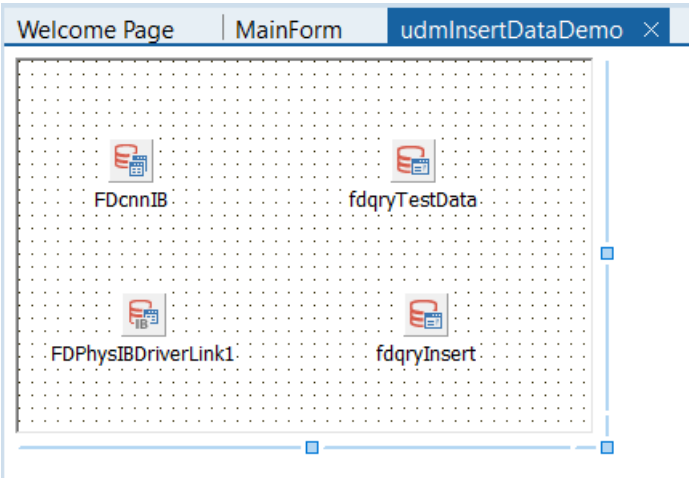
FireDAC 文件中的一個圖形清楚的說明了 Array DML 的概念，下圖顯示了客戶端使用 Array DML 新增 5 筆資料，FireDAC 的 Array DML 會一次把這些資料送到後端並且新增到資料庫中：




現在就讓我們使用資料表 TBLTESTDATA 來做為說明的範例：




請建立一個 FireMonkey Desktop Application 專案並且在其中再建立一個資料模組，接著在其中放入 TFDConnection，2 個 TFDQuery 和 TFDPhysIBDriverLink 元件，如下所示：



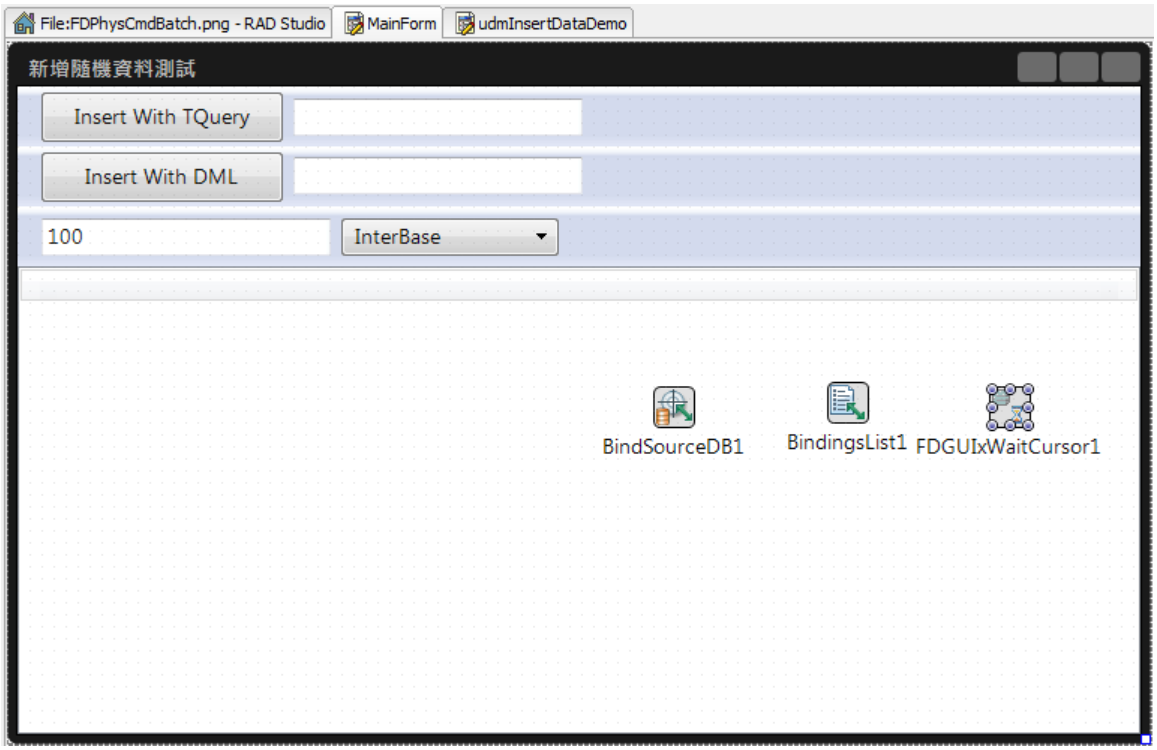
在上圖的資料模組中 fdqryTestData 的 SQL 特性使用下面的 SQL 從 TBLTESTDATA 擷取資料：

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

而 fdqryInsert 元件的 SQL 特性使用下面的 SQL 在 TBLTESTDATA 資料表中新增資料：

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

在主表單中放入 2 個 TButton 按鈕分別以不同的方式在 TBLTESTDATA 中加入資料。『Insert With TQuery』按鈕使用一般的方式為每一筆新增資料執行一次上面的 Insert SQL 命令：




『Insert With TQuery』按鈕的 OnClick 實作程式碼如下，它藉由 TFDQuery 的 Params 特性把每筆新增資料的參數填入，最後呼叫 ExecSQL()方法新增資料到資料表中：

```
void TdmInsertDataDemo::InsertDataWithQuery(const int iRecord)
```

```
{
  for (int iIndex = 0; iIndex < iRecord; iIndex++)
  {
  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();
  }
}
```

『Insert With DML』按鈕使用 Array DML 新增資料，要使用 Array DML 首先要指定 TFDQuery 的 Params 特性的 ArraySize 特性告知 FireDAC 要一次處理的資料筆數，一旦指定了 ArraySize 特性值之後就可以使用 TFDQuery 的 Params 特性的 ParamByName 方法填入參數，但要注意的是在 Array DML 中 ParamByName 回傳的是一個參數陣列，因此在填入參數值時需要填入到正確的陣列元素中。

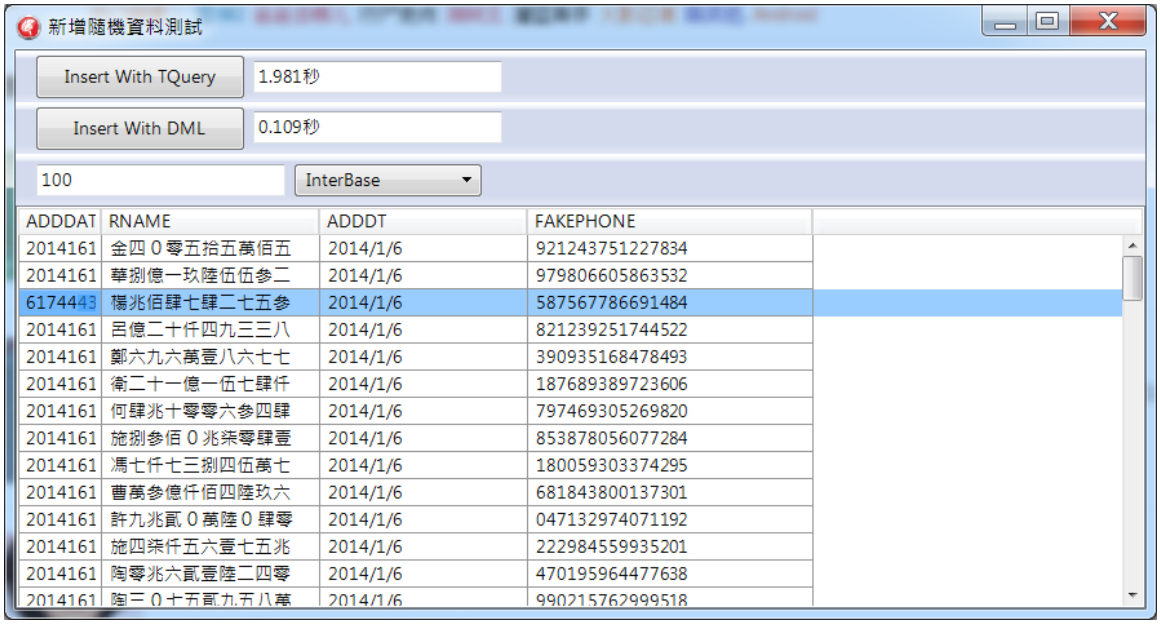


例如下面就是『Insert With DML』按鈕的 OnClick 事件處理函式，在 003 行先指定我們需要新增的筆數，接著進入迴圈為每一筆資料填入參數值，最後在 010 行執行一次 TFDQuery 的 Execute()方法並傳入陣列的大小(新增的筆數)即可：

```
001    void TdmInsertDataDemo::InsertDataWithDML(const int
iRecord)
002    {
003      fdqryInsert->Params->ArraySize = iRecord;
004      for (int iIndex = 0; iIndex < iRecord; iIndex++)
005      {
006
  fdqryInsert->Params->ParamByName("NEW_ADDDT")->AsDates[iIndex]
= Now();
007
  fdqryInsert->Params->ParamByName("NEW_ADDDATETIME")->AsStrings
[iIndex] = GetRandomDateTime(Now());
008
```

```
  fdqryInsert->Params->ParamByName("NEW_RNAME")->AsStrings[iInde
x] = GetRandomName();
009
  fdqryInsert->Params->ParamByName("NEW_FAKEPHONE")->AsStrings[i
Index] = GetRandomPhone();
010      }
011      fdqryInsert->Execute(fdqryInsert->Params->ArraySize);
012    }
```

現在執行這個範例程式並且使用不同的方法新增資料，從下圖執行的結果我們可以看到其執行速度相差了 10 幾倍，可見 Array DML 的快速以及適合使用在大量資料處理的應用中：




雖然本範例是使用新增資料，事實上 Array DML 也可以使用在修改資料的應用中。此外如果再結合稍後的非同步處理的功能，您可以藉由 FireDAC 寫出非常高效的資料庫應用程式。

## 2-2 搜尋資料

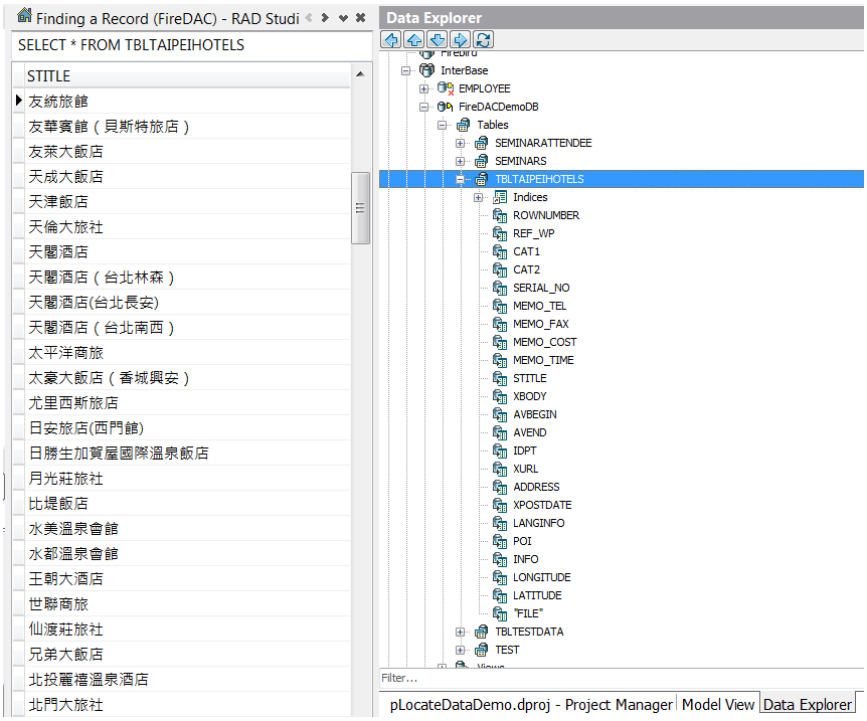
FireDAC 提供了數種方式讓程式師在 FireDAC 的資料集元件中搜尋資料，當程式師藉由 TFDQuery 元件執行 SQL 命令從後端資料庫取得資料之後這些資料就暫時儲存在 TFDQuery 元件中。而在一般的應用程式中我們經常會需要在其中搜尋特定的資料，

FireDAC 提供了許多的方法讓程式師在資料集中搜尋資料，本小節將說明如何使用這些功能來搜尋資料。

本小節使用的範例資料是台北市公佈的台北市旅館資料，其位於：

```
http://data.taipei.gov.tw/opendata/apply/NewDataContent?oid=3962468C-7AA9-49F4-965A-5EC30CE272E3
```

下面是筆者將這些資料轉換到 InterBase 資料庫中以方便說明如何使用 FireDAC 搜尋資料：



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 元件提供了相容於 BDE/dbExpress 的 Locate()方法以及其強化的 LocateEx()方法讓程式師在 TFDQuery 中搜尋資料。

當使用 Locate()方法搜尋資料時，開發人員可以使用任何的欄位條件來搜尋，而不用管這個欄位是不是索引欄位．當然，當開發人員使用索引欄位來搜尋資料時 Locate()會直接使用索引來幫助搜尋，因此速度會非常的快速．如果開發人員使用非索引欄位搜尋資料，那麼 Locate()也將使用目前它知道最好的方式來搜尋資料．

此外 Locate()方法不只能夠搜尋一個單一的欄位，它更能夠同時的以數個欄位的條件來搜尋資料．開發人員可以組合數個欄位的搜尋條件在結果資料集中搜尋資料．

由於 Locate()能夠搜尋各種不同資料型態的欄位，因此 Locate()方法在設定搜尋條件時是以 Variant 型態的變數來儲存搜尋數值．當開發人員要使用多個欄位來搜尋資料時必須建立一個 Variant 陣列來儲存搜尋數值．

此外 Locate()方法在搜尋資料時也能夠使用模糊條件標準來尋找特定的資料，例如開發人員可以要求 Locate()在搜尋資料時不分辨大小寫，或是以部份字串來搜尋資料，提供開發人員非常大的彈性空間。

下面就是 Locate()的方法原型：

```
virtual bool __fastcall Locate(const System::UnicodeString
AKeyFields, const System::Variant &AKeyValues,
Data::Db::TLocateOptions AOptions = Data::Db::TLocateOptions() );
```

Locate()方法接受三個參數，第一個參數 AKeyFields 是開發人員要搜尋的欄位名稱。如果開發人員要搜尋單一欄位，那麼只需要直接傳入此欄位名稱。如果要以多個欄位條件來搜尋，那麼開發人員便需傳入所有的欄位名稱，並且以分號分隔每一個欄位名稱。例如，如果要搜尋 Name 和 Email 兩個欄位，那麼 AKeyFields 就必須是:

```
'Name;EMail'
```

第二個參數 AKeyValues 是指開發人員欲搜尋的條件數值。它的型態是 Variant，因為 Variant 幾乎可以代表任何的型態，因此開發人員可以搜尋整數，小數，字串，或是布林值的條件。同樣的如果開發人員只搜尋一個條件數值，那麼就可以直接在這個參數位置傳入。如果是要以多個欄位條件來搜尋，那麼開發人員必須建立一個 Variant 陣列，然後在這個陣列中的每一個元素中指定條件數值，再傳遞這個 Variant 陣列到這個參數中。至於 Variant 陣列則可以使用 VarArrayOf()方法，或是使用 VarArrayCreate()方法來建立，在稍後的範例中會有程式碼說明。



Locate()方法的最後一個參數 TLocateOptions 則是讓開發人員在搜尋字串欄位時，指定以什麼標準來搜尋資料。開發人員可以指明不分大小寫來搜尋字串資料，或是以部份字串數值來搜尋資料。下面就是 TLocateOptions 的型態定義：

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

在使用 Locate()時，如果使用 loCaseInsensitive 就代表不分大小寫搜尋資料，如果使用 loPartialKey 就代表要以部份字串來搜尋資料。

Locate()方法的回傳數值是布林值，它代表 Locate()方法是否成功的找到了要搜尋的資料。如果找到的話，就回傳 true，否則就回傳 false。當 Locate()方法成功的搜尋到資料之後，它就會移動目前的記錄位置到這筆資料之上，否則就會停留在 Locate()開始搜尋之前的記錄位置上。請注意 Locate()方法搜尋資料的結果是一筆資料，因此如果你想搜尋符合條件的一群資料，那麼你可以使用稍後介紹的過濾器(Filter)功能。

現在讓我們使用數個範例來說明如何使用 Locate()方法。下面的範例程式碼即是以一個欄位來搜尋資料，它是以資料表的 NAME 欄位來搜尋擁有"李維"數值的這筆資料，由於最後一個參數是空集合，因此這代表必須 NAME 欄位擁有一模一樣的"李維"這個數值才算搜尋成功。

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
FDQuery1->Locate("NAME", "李維", SearchOptions);
```

下面的程式碼則是以兩個欄位 City 和 District 來搜尋資料，搜尋的條件是 City 欄位擁有"台北"數值，而且 District 欄位擁有" 大安區"數值的資料。

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
Variant locvalues[2];
locvalues[0] = Variant("台北");
locvalues[1] = Variant("大安區");
FDQuery1->Locate("City;District", VarArrayOf (locvalues, 1),
             SearchOptions);
```



下面的程式碼和第一個範例非常的相像，只是這個程式碼搜尋的是第一筆在 NAME 欄位以"李"數值開頭的資料。

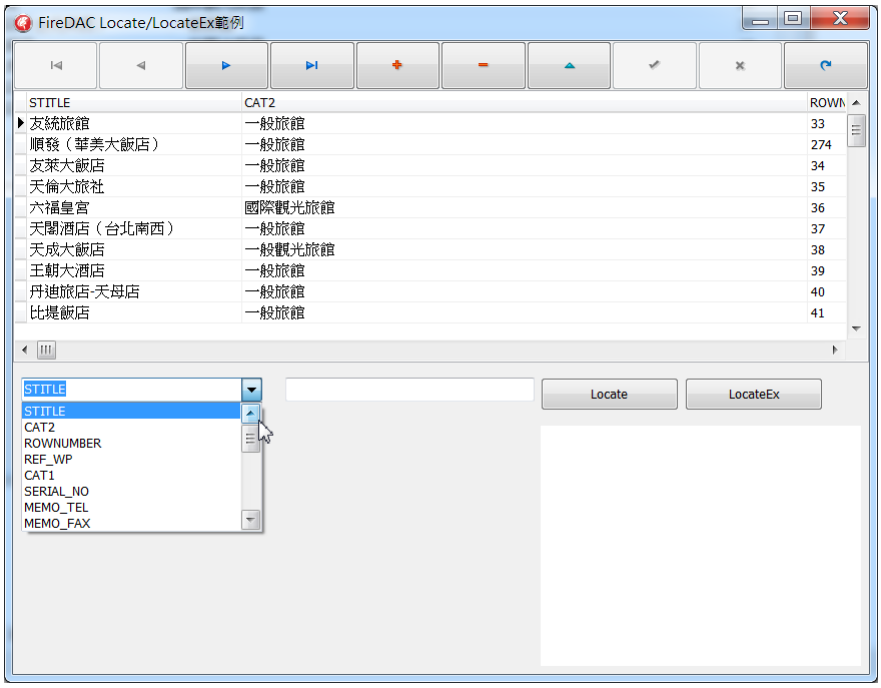
```
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey;
FDQuery1->Locate("NAME", "李", SearchOptions);
```

最後一個範例則是搜尋 ID 欄位中任何以"A12"數值開頭的第一筆，而且不分 A 大小寫的資料。

```
TLocateOptions SearchOptions;
SearchOptions.Clear();
SearchOptions << loPartialKey << loCaseInsensitive;
FDQuery1->Locate("ID", "A12", SearchOptions);
```

現在就讓我們使用 Locate()方法在範例應用程式中搜尋資料。

在下面的 FireDAC Locate()/LocateEx()範例中先使用 TFDQuery 元件從範例資料表 TBLTAIPEIHOTELS 中取得資料，再於 TComboBox 中填入 TBLTAIPEIHOTELS 所有的欄位名稱：



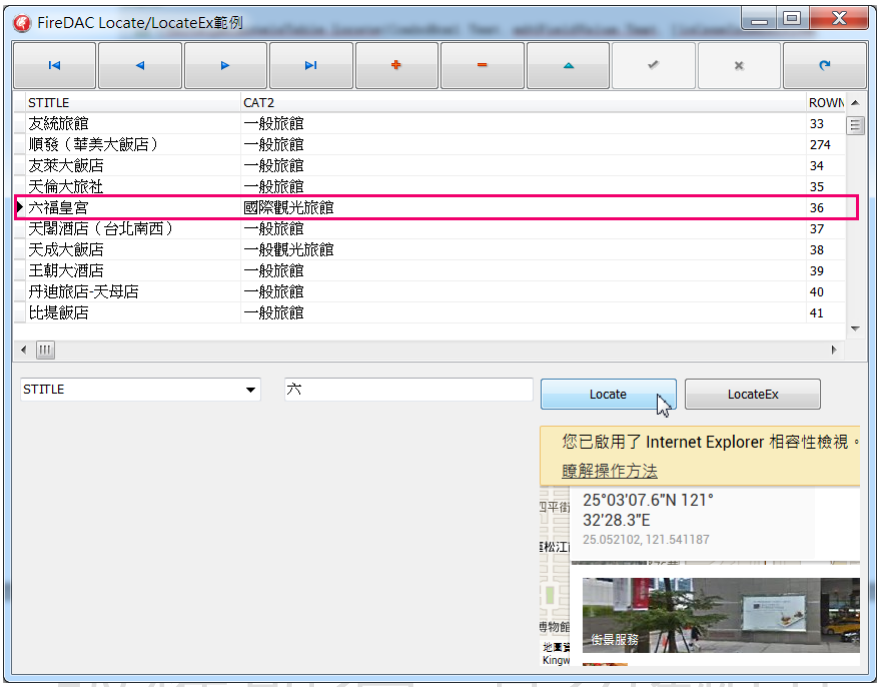

## Locate 單欄位搜尋



現在就讓我們使用 Locate()方法在範例應用程式中搜尋資料，先讓我們以單一的欄位來展示如何搜尋資料，稍後再說明如何以多個欄位搜尋資料。在上圖 Locate()按鈕的 OnClick 事件中使用 TFDQuery 的 Locate()方法搜尋使用者選擇要搜尋的欄位以及欄位值：

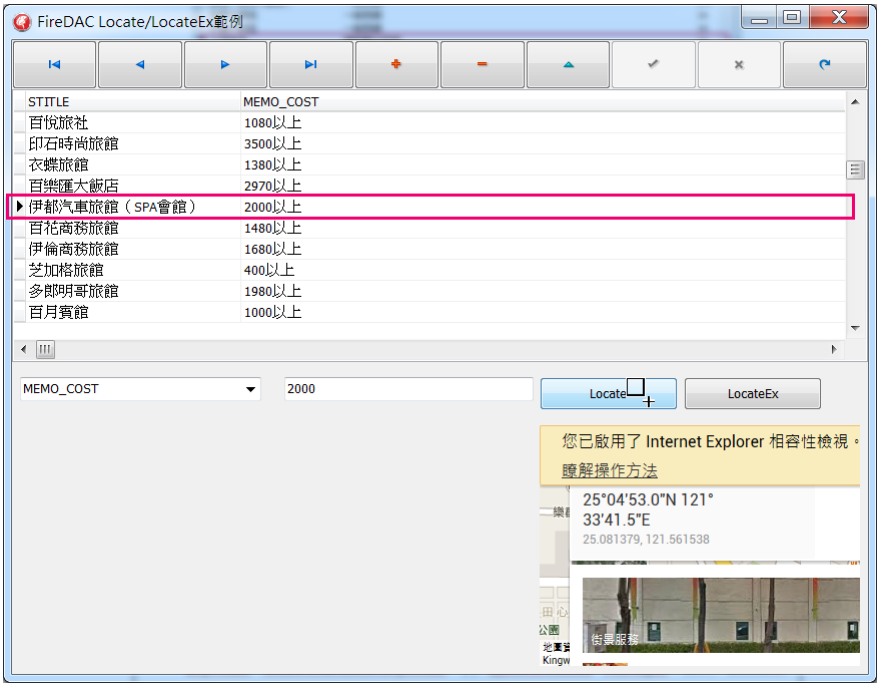
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  TLocateOptions SearchOptions;
  SearchOptions.Clear();
  SearchOptions << loPartialKey << loCaseInsensitive;

  if (TbltaipeihotelsTable->Locate(ComboBox1->Text,
edtFieldValue->Text, SearchOptions))
    DisplaySearchedHotel(TbltaipeihotelsTable->FieldByName("LONGITUDE
")->AsString,
TbltaipeihotelsTable->FieldByName("LATITUDE")->AsString);
}
```

執行此範例程式在 TComboBox 中選擇要搜尋的欄位和輸入部份欄位值之後點選 Locate()按鈕應該就可以找到您要搜尋的資料了，例如下圖就是搜尋名稱以”六”開頭的旅館：






如果使用其他欄位，例如 MEMO_COST 也可以搜尋到資料：

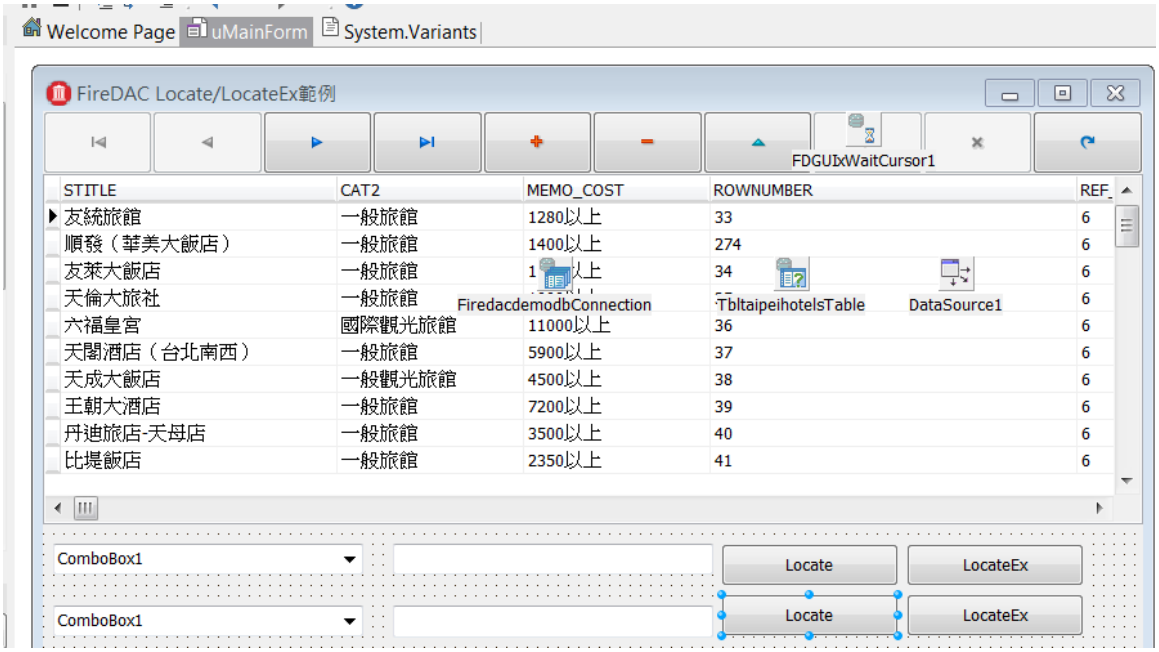


## Locate 多欄位搜尋

如果同時要以多個欄位來搜尋，那麼我們必須把搜尋欄位以分號分隔做為 Locate() 方法的第一個參數同時把所有搜尋欄位值以 VarArrayOf 方法建成 Varaint 做為 Locate() 方法的第二個參數，例如下面就是同時使用 STITLE 和 MEM_COST 這 2 個欄位來搜尋資料：

```
FDQuery1->Locate("STITLE;MEM_COST", VarArrayOf(locvalues, 1),
SearchOptions);
```

現在回到剛才的範例程式加入第 2 個 TComboBox 和 TEdit 和 TButton 元件如下：

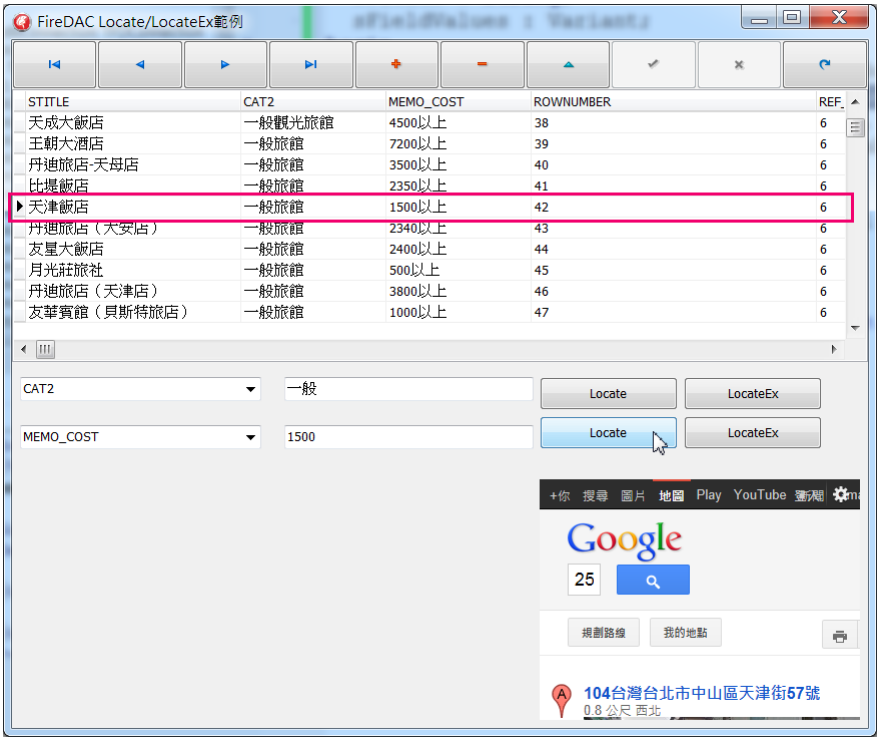



在第 2 個 Locate 按鈕的 OnClick 事件處理函式中撰寫如下的程式碼；

```
001    void __fastcall TfmMainForm::Button3Click(TObject *Sender)
002    {
003      Variant locvalues[2];
004      locvalues[0] = Variant(edtFieldValue->Text);
005      locvalues[1] = Variant(edtFieldValue2->Text);
006      TLocateOptions SearchOptions;
007      SearchOptions.Clear();
008      SearchOptions << loCaseInsensitive << loPartialKey;
009
010      String sFields = ComboBox1->Text + ";" + ComboBox2->Text;
```

```
011     if (TbltaipeihotelsTable->Locate(sFields,
VarArrayOf(locvalues, 1), SearchOptions))
012
    DisplaySearchedHotel(TbltaipeihotelsTable->FieldByName("LONGITUDE
")->AsString,
TbltaipeihotelsTable->FieldByName("LATITUDE")->AsString);
013    }
```

010 先以分號結合搜尋欄位名稱，011 行使用 VarArrayOf 函式以使用者輸入的搜尋欄位值建立 Variant 陣列，就可以在 008 行呼叫 Locate()方法搜尋資料了。下圖就是執行的結果，我們可以看到 Locate()方法成功的以 2 個欄位搜尋到了資料。




## 使用 LocateEx 搜尋資料

Locate 方法使用上非常的簡單，但 FireDAC 又提供了強化的 LocateEx()方法讓程式師可以使用更多的方式搜尋資料。FireDAC 提供了 2 個版本的 LocateEx()方法，下面是第 1 個宣告原型：

```
virtual bool __fastcall LocateEx(const System::UnicodeString
AKeyFields, const System::Variant &AKeyValues,
TFDDataSetLocateOptions AOptions = TFDDataSetLocateOptions() ,
```

```
System::PInteger ApRecordIndex = (System::PInteger)(0x0))/*
overload */;
```

這個 LocateEx()方法的第 1 和第 2 個參數的意義和前面的 Locate()方法是一樣，但第 3 個參數的型態是 TFDDataSetLocateOptions，它擴展了 Locate()方法的 TLocateOptions 定義。TFDDataSetLocateOptions 的定義如下：

```
typedef System::Set<TFDDataSetLocateOption,
TFDDataSetLocateOption::lxoCaseInsensitive,
TFDDataSetLocateOption::lxoNoFetchAll> TFDDataSetLocateOptions;
```

下面的表格說明了其中定義值的意義：

| 特性 | 特性值 |
|---|---|
| lxoFromCurrent | 從目前記錄的位置開始搜尋 |
| lxoBackward | 從目前記錄的位置開始往前搜尋 |
| lxoCheckOnly | 只搜尋看看要搜尋的資料存不存在而不改變目前記錄位置 |
| lxoNoFetchAll | 只搜尋在 TFDQuery 資料集中的資料而不要把所有的資料從後端資料庫中取出再搜尋 |



從上面的表格說明可以知道 LocateEx()和 Locate()不同的地方是：

1. Locate()只能從資料集資料開始的地方往後搜尋，而 LocateEx()則可以從開始處，從目前位置往前或是往後搜尋

2. Locate()在開始搜尋資料時會先把後端所有的資料取到資料集中再搜尋，而 LocateEx()不但可以向 Locate()一樣，也可以控制只在目前的資料集中再搜尋而不要把所有的資料從後端資料庫中取出再搜尋，這樣可以減少資料的流量。

至於 LocateEx()的第 4 個參數 ApRecordIndex 如果有指定的話就會回傳搜尋到的資料在資料集中的位置。

第 2 個版本的 LocateEx 方法宣告如下：

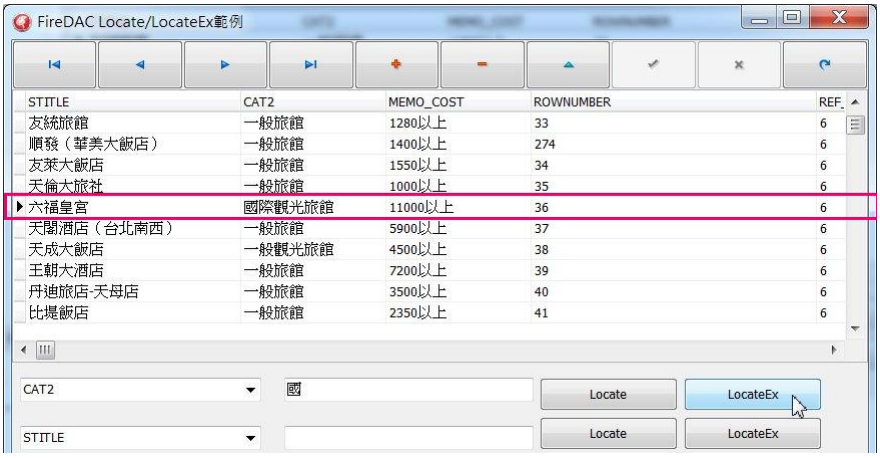
```
virtual bool __fastcall LocateEx(const System::UnicodeString
AExpression, TFDDataSetLocateOptions AOptions =
TFDDataSetLocateOptions() , System::PInteger ApRecordIndex =
(System::PInteger)(0x0))/* overload */;
```

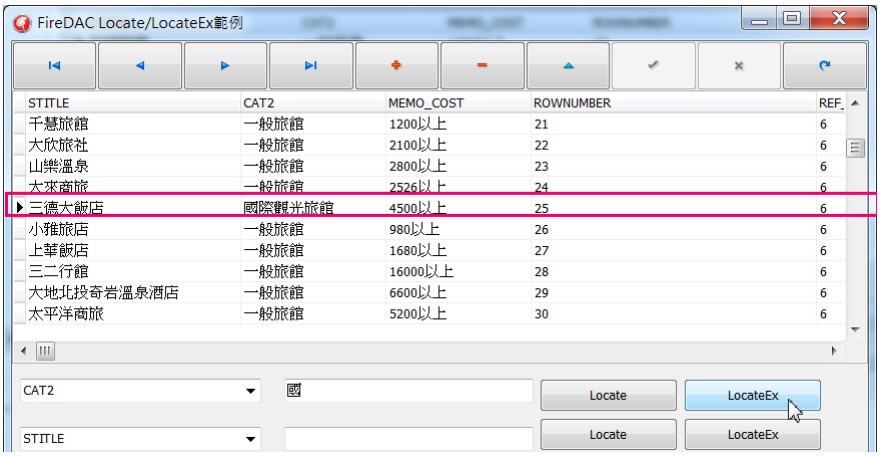
它不同的地方是第 1 個參數 AExpression。AExpression 是字串型態，代表程式師可以使用一個運算式條件來搜尋資料，

如果現在我們希望能夠根據特定欄位值來持續的搜尋所有符合的資料而不是像 Locate()方法永遠只能找到第 1 筆，那麼我們可以在範例程式的第 1 個 LocateEx()按鈕中實作如下的程式碼：

```
void __fastcall TfmMainForm::Button2Click(TObject *Sender)
{
  TFDDataSetLocateOptions SearchOptions;
  SearchOptions.Clear();
  SearchOptions << lxoPartialKey << lxoFromCurrent;
  if (TbltaipeihotelsTable->LocateEx(ComboBox1->Text,
edtFieldValue->Text, SearchOptions))
  {
    DisplaySearchedHotel(TbltaipeihotelsTable->FieldByName("LONGITUDE
")->AsString,
TbltaipeihotelsTable->FieldByName("LATITUDE")->AsString);
    edtHotelName->Text =
TbltaipeihotelsTable->FieldByName("STITLE")->AsString;
  }
  else
    edtHotelName->Text = "沒有找到!";
}
```

我們呼叫 LocateEx()方法並且使用[lxoFromCurrent]要求 FireDAc 從目前記錄持續搜尋資料，例如下圖就是在 CAT2 欄位中搜尋"國"開頭的旅館，就可以不斷的搜尋國際觀光旅館，這是 Locate()方法做不到的：



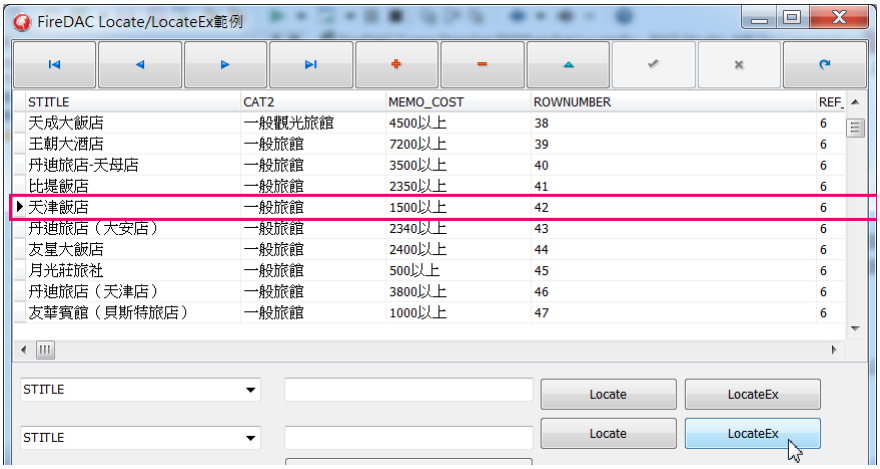



我們也可以使用運算式來搜尋，例如我們如果要搜尋 1500 到 2500 元之間的旅館就可以使用如下的程式碼來呼叫 **LocateEx()**：

```
void __fastcall TfmMainForm::Button4Click(TObject *Sender)
{
  TStringBuilder *pSB = new TStringBuilder();
  pSB->Append("MEMO_COST >=
")->Append("'")->Append("1500")->Append("'")->Append(" and
")->Append("MEMO_COST <= ")->Append("'")->Append("2500")->Append("'");
  try
  {
      String sExpr = pSB->ToString();
      TFDDataSetLocateOptions SearchOptions;
      SearchOptions.Clear();
      SearchOptions << lxoPartialKey << lxoFromCurrent;
      if (TbltaipeihotelsTable->LocateEx(sExpr, SearchOptions))
      {

    DisplaySearchedHotel(TbltaipeihotelsTable->FieldByName("LONGITUDE
")->AsString,
TbltaipeihotelsTable->FieldByName("LATITUDE")->AsString);
        edtHotelName->Text =
TbltaipeihotelsTable->FieldByName("STITLE")->AsString;
      }
  }
  __finally
```

```
  {
      delete pSB;
  }
}
```

在上面的程式碼的也使用了[lxoFromCurrent]，因此我們可以持續的搜尋符合 1500 到 2500 元之間旅館的運算式來搜尋，例如下圖就是執行的結果：






FireDAC 的 LocateEx()的確比傳統的 Locate()方法強大多了，此外我們也可以結合索引功能增加 LocateEx()搜尋的速度，在稍後的章節會說明。

## 2-2-2 Lookup 和 LookupEx

Lookup()方法和上一小節介紹的 Locate()方法在使用上非常的類似，它們的差別是當 Locate()找到搜尋的資料後，它會把目前的記錄位置移動到找到的這筆資料之上，但是當 Lookup()找到搜尋的資料後，它會回傳找到的資料的特定欄位數值，卻不會移動目前的記錄位置。下面即是 Lookup()方法的原型：

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

Lookup()方法的第一個參數也是使用者欲搜尋的欄位命名，每一個欲搜尋的欄位也是使用分號分隔。第二個參數則是欲搜尋的欄位數值，如果欲搜尋多個欄位，那麼這個參數可以是 Variant 陣列。Lookup()方法的第一和第二個參數的意義和前面 Locate()方法是一樣的。

Lookup()方法的第三個參數則是指定當 Lookup()找到欲搜尋的資料之後，要回傳這筆資料的那些欄位數值。如果開發人員想要 Lookup()回傳多個欄位數值，那麼每一個欄位也是以分號分隔。

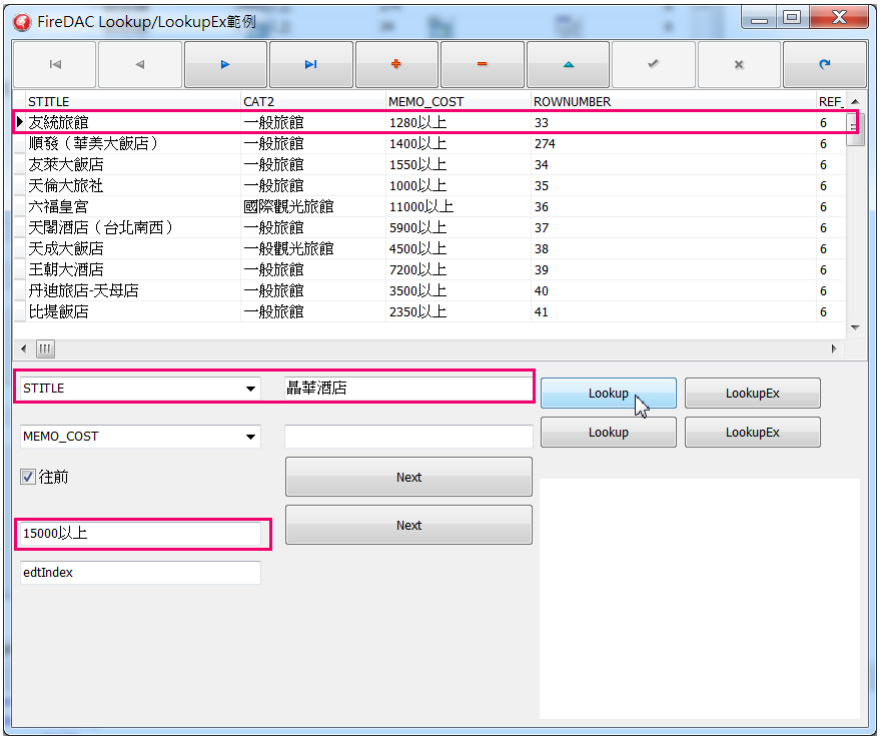
至於 Lookup()方法回傳的數值則是第三個欄位指定的欄位數值，如果 Lookup()回傳多個欄位的話，那麼這個回傳數值就是一個 Variant 陣列，每一個回傳的欄位便儲存在這個 Variant 陣列的元素之中，如果沒有找到資料的話回傳值就是一個 NULL 的 Variant，現在就讓我們看看使用 Lookup()方法搜尋資料。

## 單欄位搜尋

使用單欄位搜尋非常的簡單，我們只要使用如下的程式碼就可以在" FireDAC Lookup/LookupEx 範例" 中根據使用者在 ComboBox1 中指定的搜尋欄位以及在 ComboBox2 中指定的回傳欄位進行搜尋。由於只回傳一個搜尋結果，因此在 004 行先呼叫 VarIsNull 來判斷是否有回傳結果，如果有的話就顯示出來：

```
001    void __fastcall TfmMainForm::Button1Click(TObject *Sender)
002    {
003      Variant vResult = TbltaipeihotelsTable->Lookup(ComboBox1->Text,
edtFieldValue->Text, ComboBox2->Text);
004      if (! VarIsNull(vResult))
005      edtResult->Text = vResult;
006    }
```

下圖是使用 Lookup 搜尋"晶華酒店"價格的結果，從下圖中可以看到 TbltaipeihotelsTable 目前記錄位置沒有改變，仍然在友統旅館處，但 Lookup 使用 STITLE 欄位值搜尋並且要求回傳 MEMO_COST 欄位值，而搜尋結果果然成功顯示在 TEdit 中：




## 多欄位搜尋

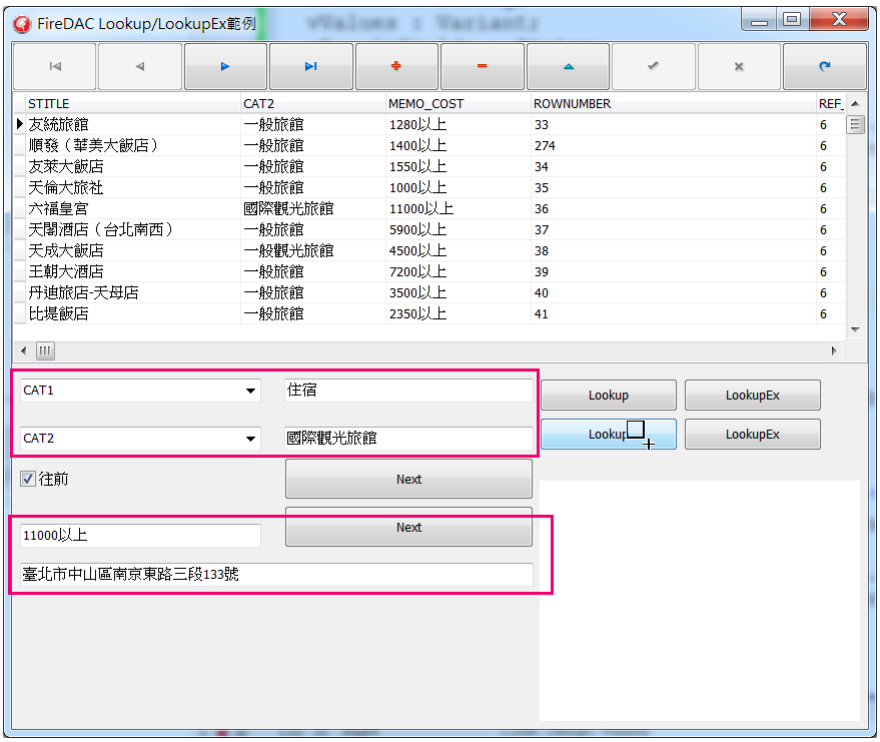


要進行多欄位搜尋和回傳多欄位結果，我們需要把多個搜尋欄位和回傳結果欄位名稱以分號結合，再把搜尋欄位值填入 Variant 陣列，最後呼叫 Lookup()方法即可，例如下面的實作程式碼在範例資料庫中以 2 個欄位搜尋並且要求回傳 MEMO_COST 和 ADDRESS 這 2 個欄位值：

```
void __fastcall TfmMainForm::Button3Click(TObject *Sender)
{
  String sFields = ComboBox1->Text + ";" + ComboBox2->Text;
  Variant vValues = VarArrayOf(OPENARRAY(Variant, (edtFieldValue->Text,
edtFieldValue2->Text)));
  String sResultFields = "MEMO_COST;ADDRESS";
  Variant vResult = TbltaipeihotelsTable->Lookup(sFields, vValues,
sResultFields);
  if (! VarIsNull(vResult))
  {
    edtResult->Text = vResult.GetElement(0);
    edtIndex->Text = vResult.GetElement(1);
  }
```

```
}
```

由於有 2 個回傳結果值，因此我們可以從 Lookup()回傳的 Variant 陣列中呼叫 GetElement()方法一一的取出回傳結果值。下面就是使用 CAT1 和 CAT2 欄位搜尋，搜尋成功之後就把 MEMO_COST 和 ADDRESS 這 2 個欄位值顯示在 2 個 TEdit 元件中。




## 使用 LookupEx

FireDAC 提了供強化的 LookupEx()方法，它同樣有 2 個原型版本，第 1 個 LookupEx()和 Lookup()方法很像，只是它接受 5 個參數，它的第 4 和第 5 個參數和前面介紹的 LocateEx()方法的最後 2 個參數是一樣的意義：

```
virtual System::Variant __fastcall LookupEx(const
System::UnicodeString AKeyFields, const System::Variant
&AKeyValues, const System::UnicodeString AResultFields,
TFDDataSetLocateOptions AOptions = TFDDataSetLocateOptions() ,
System::PInteger ApRecordIndex = (System::PInteger)(0x0))/*
overload */;
```

第 2 個 LookupEx 原型如下，它的第 1 個參數則是使用一個運算式來搜尋：

```
virtual System::Variant __fastcall LookupEx(const
```

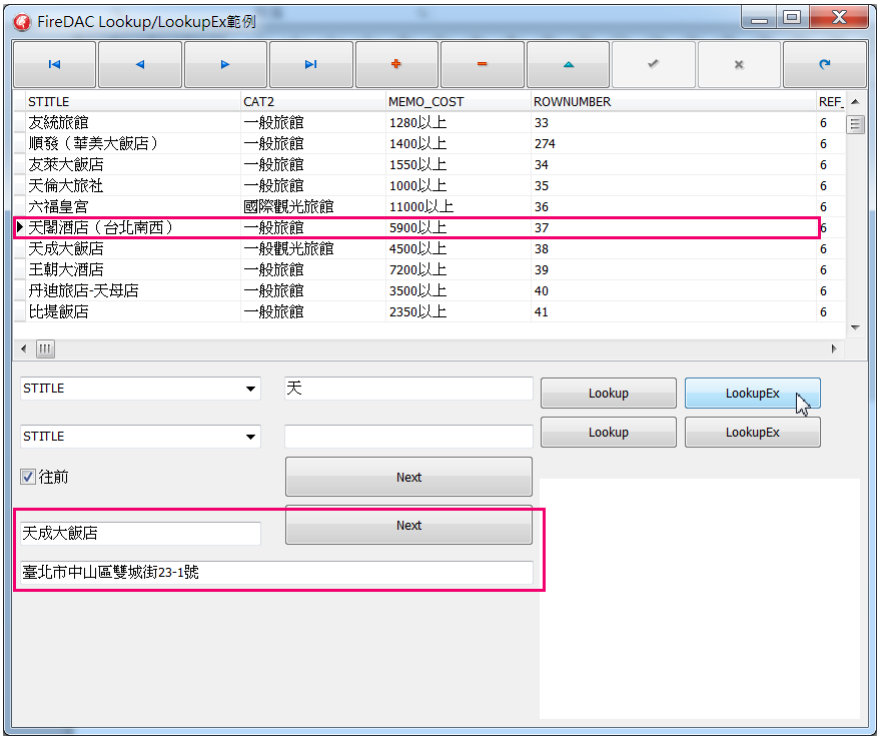
```
System::UnicodeString AExpression, const System::UnicodeString
AResultFields, TFDDataSetLocateOptions AOptions =
TFDDataSetLocateOptions() , System::PInteger ApRecordIndex =
(System::PInteger)(0x0))/* overload */;
```

例如我們可以使用下面的程式碼來搜尋資料：

```
void __fastcall TfmMainForm::Button2Click(TObject *Sender)
{
  TFDDataSetLocateOptions AOptions;
  AOptions.Clear();
  AOptions << lxoPartialKey << lxoFromCurrent;
  Variant vResult = TbltaipeihotelsTable->LookupEx(ComboBox1->Text,
edtFieldValue->Text, ComboBox2->Text, AOptions);
  if (! VarIsNull(vResult))
     edtResult->Text = vResult;
}
```

在下圖中請注意 FireDAC 的 Lookup()可以在目前記錄”天閣酒店（台北南西）”之後搜尋以”天”為開頭的資料並未成功找到”天成大飯店”，這當然是因為上面的程式碼使用了[lxoPartialKey, lxoFromCurrent]搜尋選擇值，這是 BDE/dbExpress 的 Lookup() 方法無法做到的：

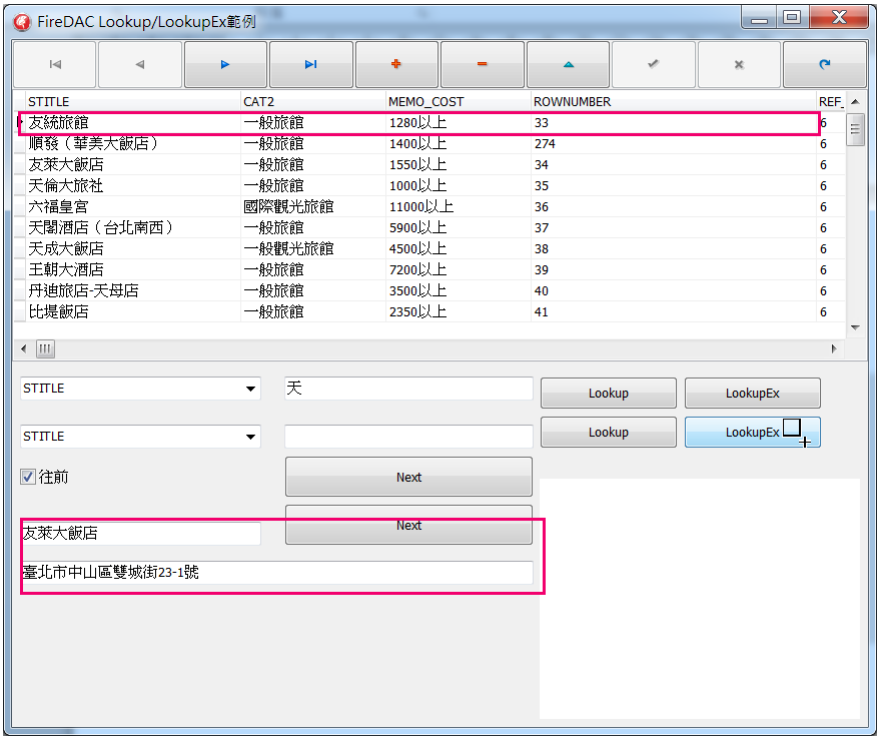




我們也可以使用下面的程式碼以運算式來搜尋：

```
void __fastcall TfmMainForm::Button4Click(TObject *Sender)
{
  TStringBuilder *pSB = new TStringBuilder();
  pSB->Append("MEMO_COST >=
")->Append("'")->Append("1500")->Append("'")->Append(" and
")->Append("MEMO_COST <= ")->Append("'")->Append("2500")->Append("'");
  try
  {
      String sExpr = pSB->ToString();
      String sResultFields = "STITLE;ADDRESS";
      TFDDataSetLocateOptions AOptions;
      AOptions.Clear();
      AOptions << lxoPartialKey << lxoFromCurrent;
      Variant vResult = TbltaipeihotelsTable->LookupEx(sExpr,
sResultFields, AOptions);
      if (! VarIsNull(vResult))
      {
        edtResult->Text = vResult.GetElement(0);
        edtIndex->Text = vResult.GetElement(1);
      }
  }
  __finally
  {
      delete pSB;
  }
}
```

在下圖中可以看到它成功回傳了”友萊大飯店”是符合運算式條件的第 1 筆資料：



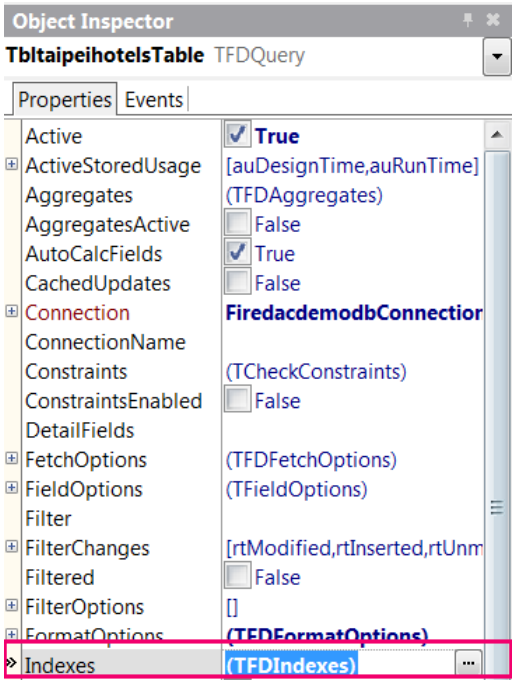

## 2-2-3 在客戶端動態排序



在繼續討論其他搜尋方法之前先讓我們討論一下如何使用 FireDAC 的 TFDQuery 元件在客戶端排序資料，因為在使用 FireDAC 搜尋資料時如果搜尋的欄位有建立索引的話那麼搜尋速度會非常快速。

當程式師使用 SQL 命令從後端取得資料到 TFDQuery 後，這些資料就儲存在 TFDQuery 維護的記憶體中，這也就是所謂的資料集。如果程式師經常需要針對特定欄位值來搜尋資料，那麼程式師可以在這個資料集中暫時建立索引來增加搜尋速度。

FireDAC 提供 2 種方式可以建立暫時的索引，一是使用 Indexes 特性，一是使用 IndexFieldNames 特性。這 2 種方式都可以快速建立索引並且排序資料集中的資料。例如下圖是在 IDE 中雙擊前面範例的 TbltaipeihotelsTable 元件的 Indexes 特性啟動 Indexes 特性值編譯器：

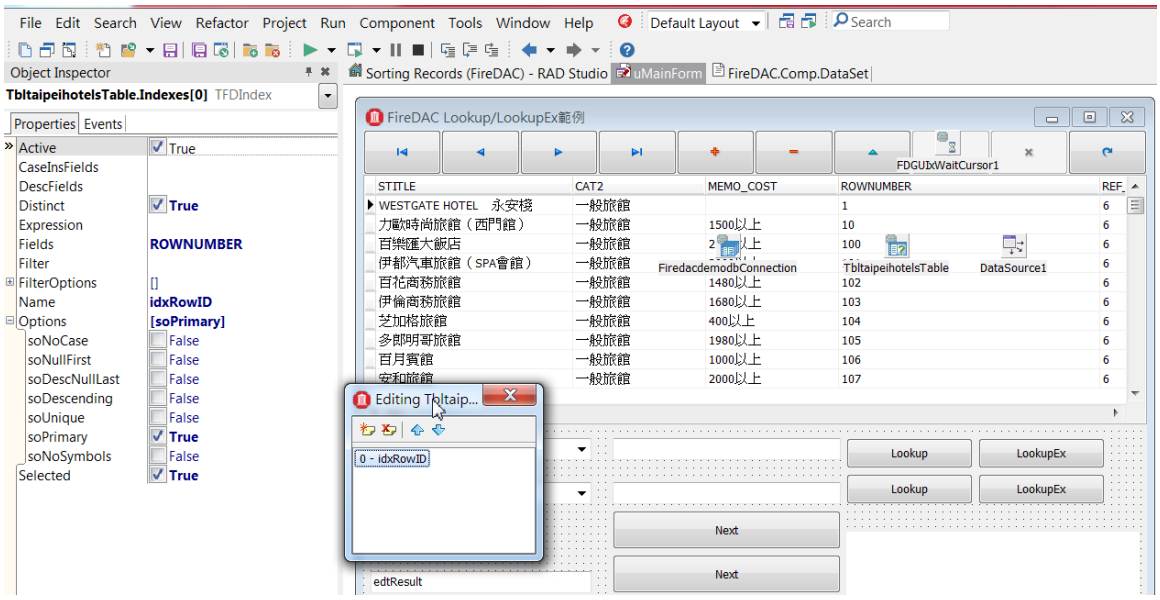



接著在 Indexes 特性值編譯器中建立一個 TFDIndex 物件，再於物件檢視器中設定這個 TFDIndex 物件的特性值：

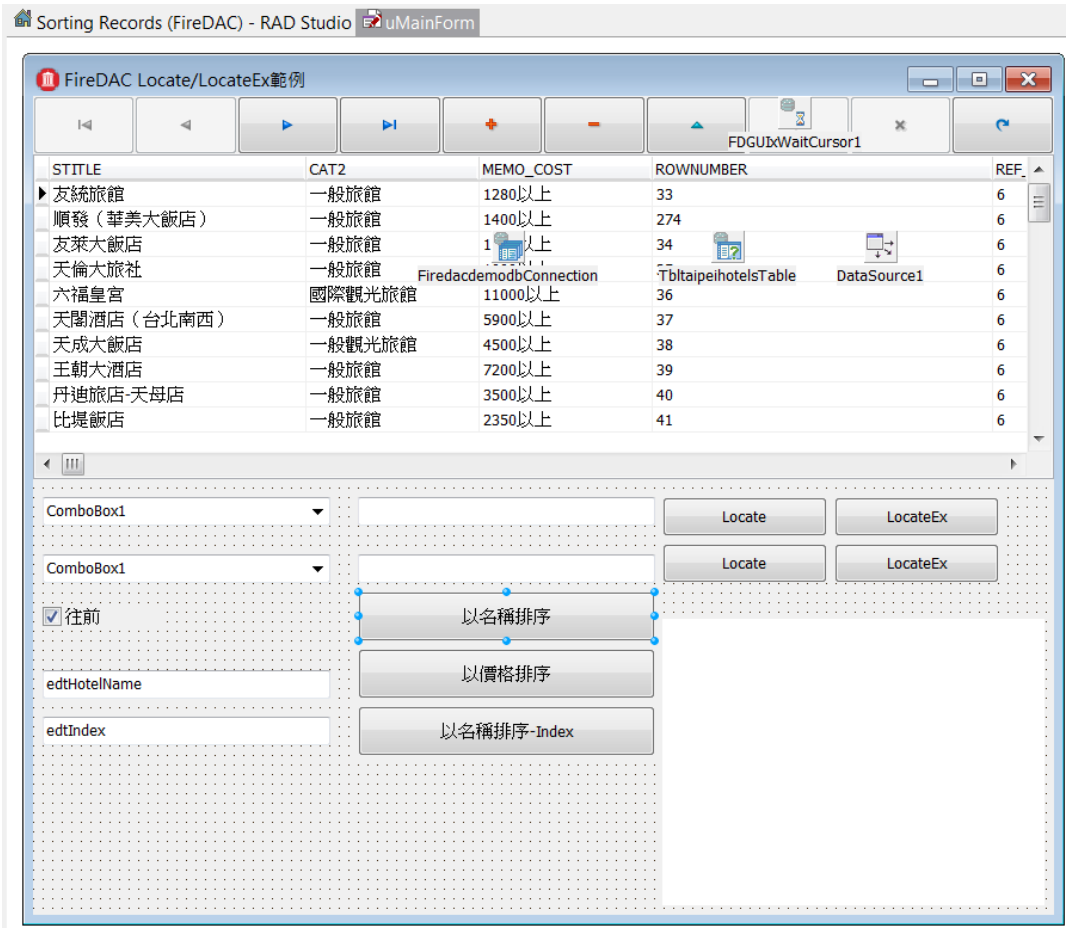


| 特性 | 特性值 |
|---|---|
| Name | idxRowNumber |
| Fields | ROWNUMBER |
| Active | True |
| Selected | True |

之後就可以在 DBGrid 中看到所有的資料都以 ROWNUMBER 欄位自動排序了：




當然我們也可以在程式碼中動態建立索引，先在範例程式的主表單中加入 3 個按鈕如下：



第 1 個按鈕"以名稱排序"設定 TbltaipeihotelsTable 元件的 IndexFieldNames 特性為 STITLE 欄位：

```
void __fastcall TfmMainForm::Button5Click(TObject *Sender)
{
  TbltaipeihotelsTable->IndexFieldNames = "STITLE";
}
```

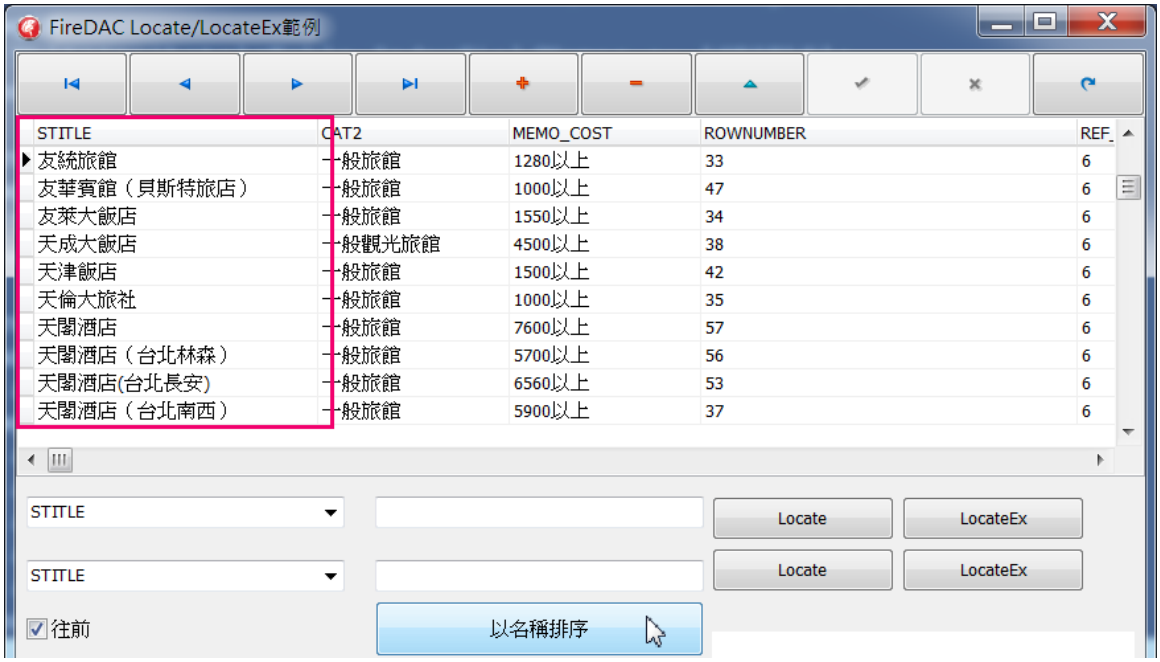
第 2 個按鈕" 以價格排序"設定 TbltaipeihotelsTable 元件的 IndexFieldNames 特性為'MEMO_COST 欄位：

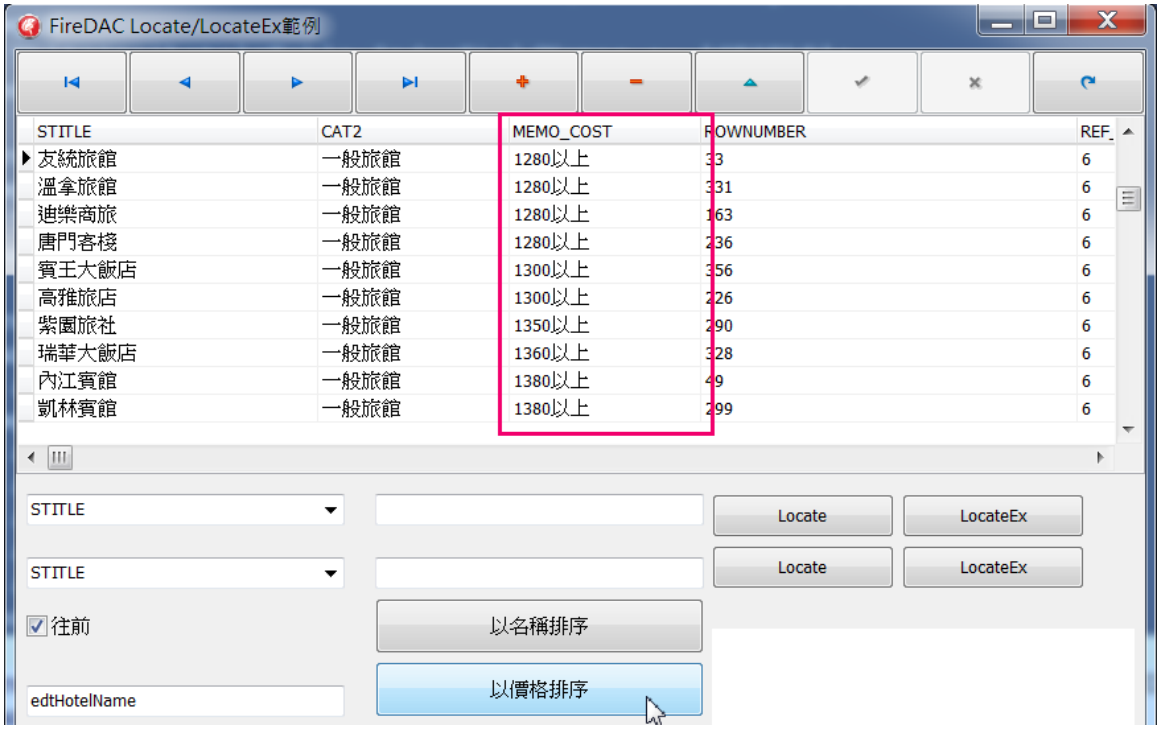
```
void __fastcall TfmMainForm::Button6Click(TObject *Sender)
{
  TbltaipeihotelsTable->IndexFieldNames = "MEMO_COST";
}
```

第 3 個按鈕" 以欄位名稱-Index"則先建立一個 TFDIndex 物件，再設定它的 Name，Fields，Active 和 Selected 特性值：

```
void __fastcall TfmMainForm::Button7Click(TObject *Sender)
{
  TFDIndex *anIndex = TbltaipeihotelsTable->Indexes->Add();
  anIndex->Name = "idxName";
  anIndex->Fields = "STITLE";
  anIndex->Active = true;
  anIndex->Selected = true;
}
```

按著執行程式並且點選不同的排序按鈕，就可以看到類似下面的結果，客戶端資料集中的資料當然自動排序了。




建立動態索引的目的是增快使用 Locate()，Lookup()和稍後討論的過濾器等方法搜尋資料的速度，但是程式師必須瞭解如果後端資料庫的資料很多的話就不適合在客戶端建立動態索引，因為 FireDAC 會先把所有的資料取到客戶端再建立動態索引和排序資料，這會造成大量資料在網路中傳遞的問題。如果在這種情形中程式師只想對目前資料集中的資料建立動態索引和排序，搜尋的話，可以使用 FireDAC 的記憶體資料表元件

TFDMemTable 來配合使用就可以避免把大量資料取到客戶端的問題，下一章說明 TFDMemTable 元件時會討論如何達成。

## 2-2-4 使用過濾器

除了 Locate()和 Lookup()之外，FireDAC 的過濾器功能也是非常有用的，因為過濾器不但能夠搜尋資料，更棒的是它可以再把結果資料集中的資料分門別類，讓應用程式只存取特定群組的資料。這個行為有點像是能夠把從後端資料庫回傳的結果資料集中的資料再使用額外的條件來取得子結果資料集。

FireDAC 的過濾器功能不但可以讓開發人員對單一欄位下達過濾條件，也可以對多個欄位下達過濾條件，同時不限定欄位是否是索引欄位。此外開發人員有兩種方法來使用過濾器，第一個方法是使用 TFDQuery 元件的 Filter 特性值，第二個方法是使用元件的 OnFilterRecord 事件處理函式。

這兩種方法的差異點是 Filter 特性值只能讓開發人員使用字串來設定過濾條件，而 OnFilterRecord 事件處理函式卻能夠讓開發人員使用 C++Builder 程式語言來執行任何複雜的過濾條件，FireDAC 提供和過濾功能相關的特性和事件處理函式如下表所示：

| 特性/事件處理函式 | 意義 |
|---|---|
| Filter特性 | 使用字串條件來過濾資料 |
| OnFilterRecord事件處理函式 | 使用事件處理函式程式碼來過濾資料 |
| Filtered | 決定是否開啟過濾器功能的特性值 |
| FilterOptions | 過濾功能使用的額外條件 |

要使用過濾器功能，開發人員必須設定 Filtered 特性值為 true，一旦 Filtered 特性值為 true 之後，如果 Filter 特性值有任何的過濾條件，FireDAC 便會以這個過濾條件做為標準來過濾目前在結果資料集之中的資料，只有符合過濾條件的資料才能夠顯示在資料感知元件之中，或是被存取，而不符合過濾條件的資料會暫時的無法被存取到。此外如果開發人員有定義 OnFilterRecord 事件處理函式，那麼 FireDAC 也會執行 OnFilterRecord 事件處理函式來過濾資料。因此如果開發人員同時設定了 Filter 特性值以及 OnFilterRecord 事件處理函式，那麼這兩個過濾條件都會被執行。

至於 FilterOptions 特性則類似 Locate 的第 3 個參數，用來指明在過濾資料時是否需要分別大小寫，或是是否需要比對完整的字串過濾值。

例如如果現在我們要在範例資料表中過濾出如下條件的旅館：

1. 一般旅館

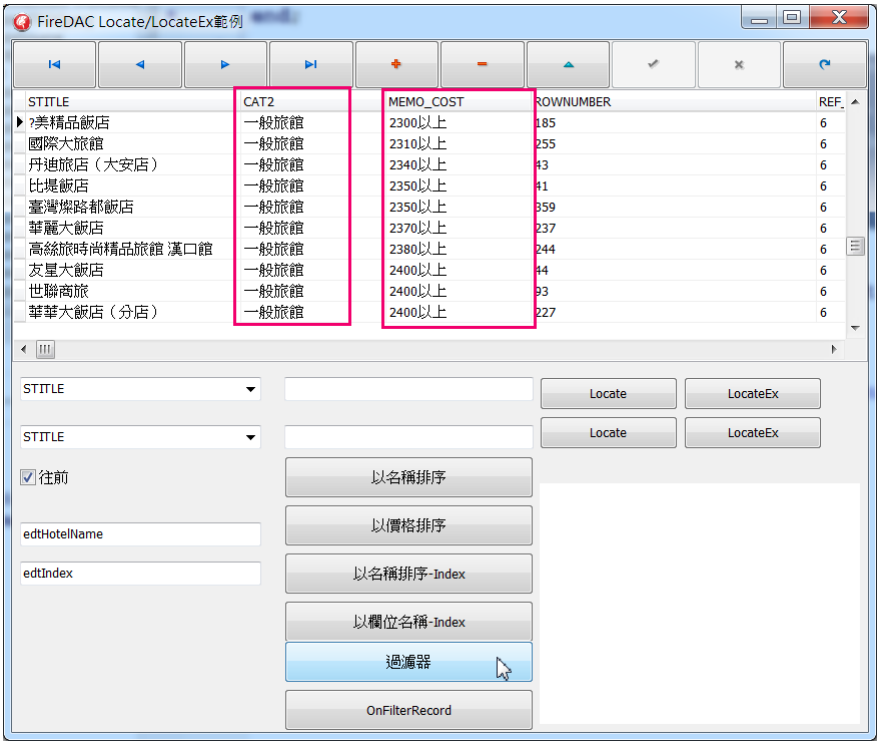
2. 價格在 1000 到 3000 元之間

那麼我們可以撰寫如下的程式碼：

```
001   void __fastcall TfmMainForm::Button8Click(TObject *Sender)
002   {
003     TStringBuilder *pSB = new TStringBuilder();
004     pSB->Append("CAT2= ")->Append("'")->Append("一般旅館
")->Append("'")->Append(" and ")->Append("MEMO_COST >=
")->Append("'")->Append("1000")->Append("'")->Append(" and
")->Append("MEMO_COST <= ")->Append("'")->Append("3000")->Append("'");
005     try
006     {
007     String sExpr = pSB->ToString();
008     TbltaipeihotelsTable->IndexFieldNames = "CAT2";
009     TbltaipeihotelsTable->IndexFieldNames = ComboBox1->Text;
010     TbltaipeihotelsTable->Filter = sExpr;
011     TbltaipeihotelsTable->Filtered = true;
012     TbltaipeihotelsTable->IndexFieldNames = "MEMO_COST";
013     }
014     __finally
015     {
016     delete pSB;
017     }
018   }
```

由於上述的 2 個過濾條件的欄位是 CAT2 和 MEMO_TEST，因此為了加快過濾速度 008~012 行先為此 2 個欄位建立動態索引，010 行在範例資料表元件的 Filter 特性中寫入過濾條件，再於 011 行開啟過濾功能讓 FireDAC 使用過濾條件過濾資料，最後 012 行再使用 MEMO_TEST 欄位排序資料。

執行上面的程式碼就可以看到類似下面的執行結果，FireDAC 果然在資料集中過濾出了符合條件的資料：




當然如果要使用 OnFilterRecord 事件處理函式方式過濾資料也可以，下面是先設定建立動態索引再設定範例資料表元件的 Filtered 特性值為 true 以觸發 OnFilterRecord 事件處理函式：



```
void __fastcall TfmMainForm::Button10Click(TObject *Sender)
{
  TbltaipeihotelsTable->IndexFieldNames = "CAT2;MEMO_COST";
  TbltaipeihotelsTable->Filtered = true;
}
```

接著在範例資料表元件的 OnFilterRecord 事件處理函式中撰寫如下的程式碼：

```
void __fastcall TfmMainForm::TbltaipeihotelsTableFilterRecord(TDataSet
*DataSet, bool &Accept)

{
  Accept = false;
  if (DataSet->FieldByName("CAT2")->AsString == "一般旅館")
  {
    if ( (DataSet->FieldByName("MEMO_COST")->AsString >= "1500") &&
(DataSet->FieldByName("MEMO_COST")->AsString <= "3000") )
```

```
    Accept = true;
  }
}
```

OnFilterRecord 事件處理函式接受 2 個參數，第 1 個參數就是觸發過濾的資料集元件，也就是本範例中的 TbltaipeihotelsTable 元件，第 2 個參數 Accept 則代表是否 1 要把第 1 個參數 DataSet 中目前的記錄加入符合過濾條件的結果資料集中。因此在上面的程式碼中判斷目前的記錄的 CAT2 欄位值是否是'一般旅館'，以及 MEMO_COST 欄位值是否在'1000'和'3000'之間，如果是的話就代表符合我們的過濾條件因此就設定 Accept 參數為 True。

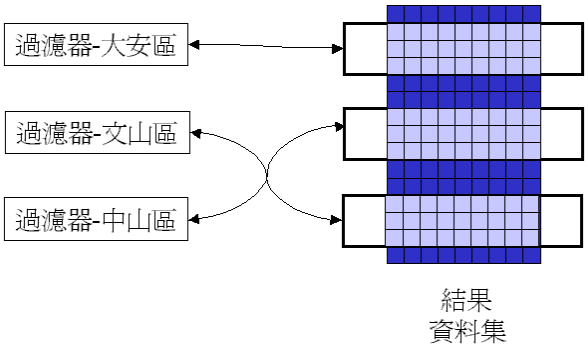
執行這個程式碼就可以得到和上面使用 Filter 特性一樣的結果。

## 使用過濾器的場合

雖然過濾器可以像 TFDQuery 的 Locate()和 Lookup()一樣來搜尋資料，但過濾器另外有一個非常特別的用途，那就是它可以在結果資料集中再以過濾條件來取得子結果資料集。這個用途在一些應用中是非常方便的，讓我們使用一個例子和下面的圖形來說明這個用途。



假設在你的應用程式中已經使用 SQL 命令從後端資料中擷取了數量不多的資料到結果資料集中，例如我使用 SQL 命令搜尋在台北市使用 C++Builder 而且購買了本書的開發人員。那麼如果我又想根據這些讀者以行政區來分析購買的情形，例如以大安區，文山區和中山區等區域，那麼我仍然可以再 SQL 命令來存取資料，但是既然這些資料已經存在於結果資料集中，那麼我可以直接使用過濾器的功能來取得資料，而不需要再使用許多的 SQL 命令來重新的從後端資料中擷取，這種處理資料的方式的執行效率會比重新使用 SQL 命令來得有效率。



在筆者平常撰寫資料庫應用程式時，也經常的使用這個技巧從結果資料集中存取筆者需要的資料。由於過濾器的執行效率在結果資料集中資料不多時是不錯的，因此開發人員也可以善用這個技巧來處理資料。

## 2-2-5 使用 SetRange

本章最後一個要說明的搜尋方法就是 SetRange()。SetRange()方法可以讓開發人員指定索引欄位符合一定範圍值的資料才可以被存取。藉由 SetRange()開發人員可以設定特定的索引欄位值來搜尋一筆資料，或是一組在指定範圍值之內的資料。由於 SetRange()只能夠適用在欄位索引之上，因此它的適用性比前面介紹的搜尋方法來得小，不過 SetRange()在搜尋索引欄位的資料時，會比其他的方法來得有效率的一點。

使用 SetRange()方法非常的簡單，開發人員只需要指定索引範圍值的起始數值和結尾數值，FireDAC 便會自動的從結果資料集中搜尋出所有符合範圍值的資料。下面便是 SetRange()的函式原型：

```
void __fastcall SetRange(System::TVarRec const *AStartValues, const
int AStartValues_High, System::TVarRec const *AEndValues, const int
AEndValues_High, bool AStartExclusive = false, bool AEndExclusive
= false);
```



SetRange 的第一個參數 StartValues 便是搜尋範圍的起始數值，而 EndValues 參數便是搜尋範圍的結尾數值，在使用上非常的簡單。由於 SetRange 的 2 個參數都是 array of const 的型態，這代表程式師可以同時傳入多個範圍值。

在使用 SetRange 之後如果要取得搜尋範圍的效果程式師可以呼叫 CancelRange()：

```
void CancelRange();
```

## 2-2-6 使用 FireDAC 在手機中搜尋資料

雖然前面的範例都是 Windows VCL 的應用程式，但 FireDAC 搜尋資料的功能當然也能夠使用在移動平台，例如本書的”pMobileSearchData”範例就是在 Android 手機中搜尋旅館的範例程式，它使用前面說明的過濾器功能以旅館部份名稱來搜尋旅館，下面是它的搜尋部份的程式碼：

```
void __fastcall TfmMainForm::Button2Click(TObject *Sender)
{
```

```
  dmBCBDemoDB->TbltaipeihotelsTable->IndexFieldNames = "Hotel";
  LinkFillControlToField1->BindList->FillList();
}
//---------------------------------------------------------------------------
void __fastcall TfmMainForm::SearchEditButton1Click(TObject *Sender)
{
  dmBCBDemoDB->SearchHotel(Edit1->Text);
  LinkFillControlToField1->BindList->FillList();
}
```

下面 2 個畫面就是此範例程式執行在筆者 S4 手機中的畫面，您可以看到只要輸入”天”就可以從所有旅館中搜尋到以”天”開頭的旅館。當然由於 FireMonkey 沒有資料感知元件，因此您需要使用 LiveBindings 技術顯示資料，請參考本書有關 LiveBindings 技術的章節說明。


## 2-3 快儲機制

在 FireDAC 元件進行資料異動時資料是直接異動到後端的資料庫中的，例如要在 FireDAC 資料集元件連結的資料庫中新增資料程式師只需要使用 Insert()方法：

```
FDQuery1->Insert();
FDQuery1->FieldByName("欄位名稱")->Value = 新增欄位值;
...
FDQuery1->Post();
```

要修改資料可使用 **Edit()**方法：

```
FDQuery1->Edit();
FDQuery1->FieldByName("欄位名稱")->Value = 修改欄位值;
...
FDQuery1.Post();
```

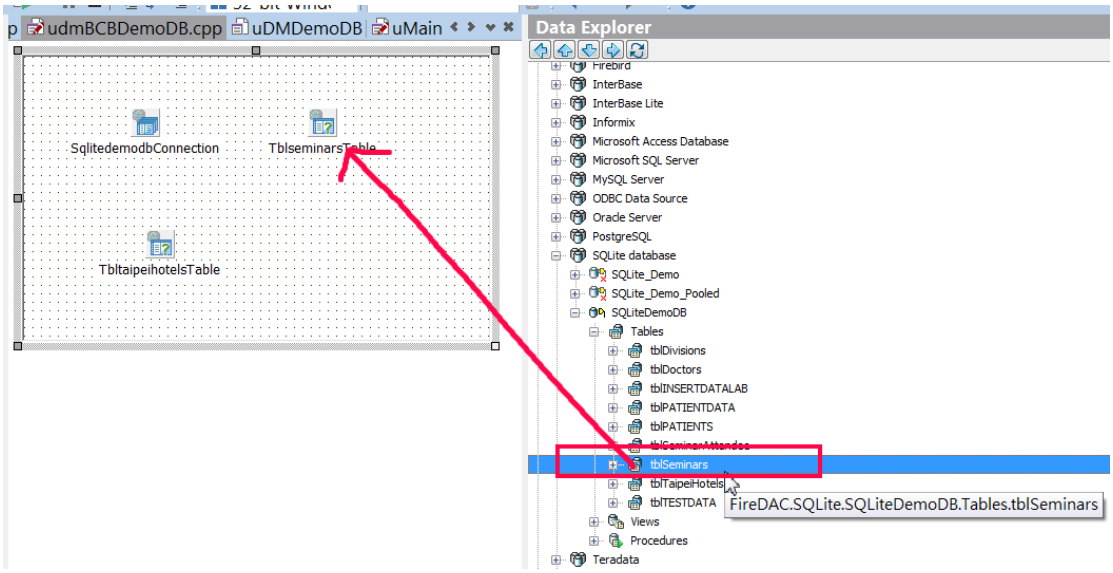
最後要刪除資料只需要使用 **Delete()**方法：

```
FDQuery1->Delete();
```

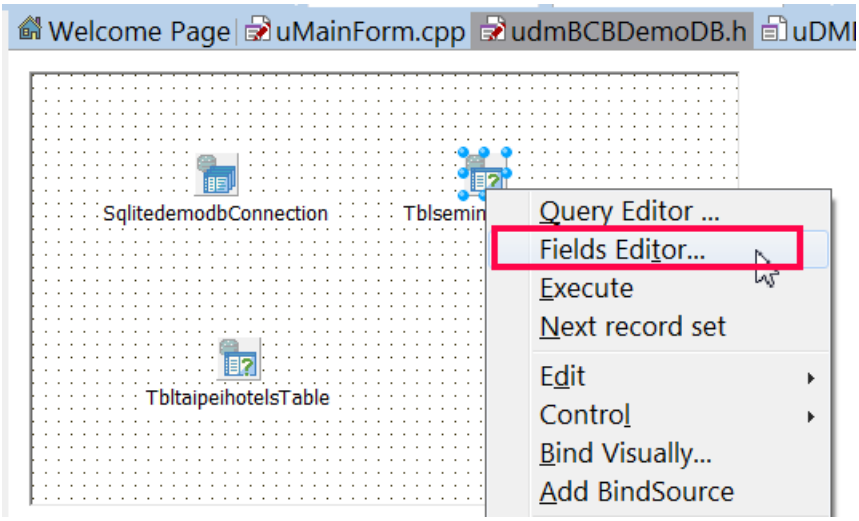
讓我們使用一個簡單的範例看看如何使用 **FireDAC** 在手機中對資料進行 **CRUD** 的工作。首先建立一個 **FireMonkey Mobile** 應用程式，在主表單中加入如下的元件：



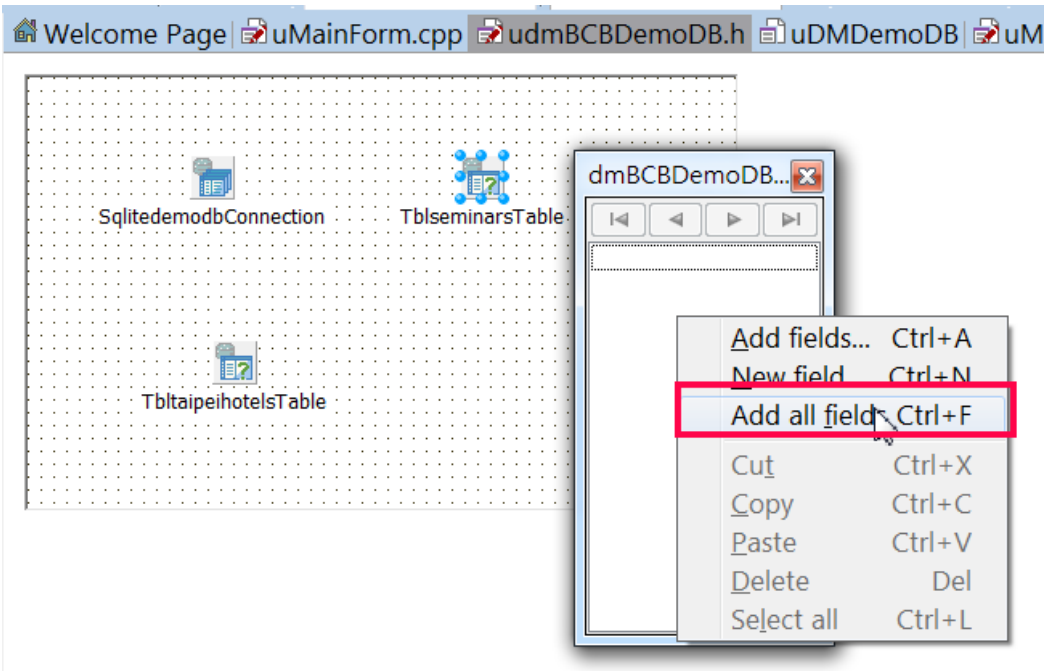
在專案中建立一個資料模組並且使用 FireDAC Explorer 把 tblSeminars 範例資料表拖曳到資料模組中，如下所示：

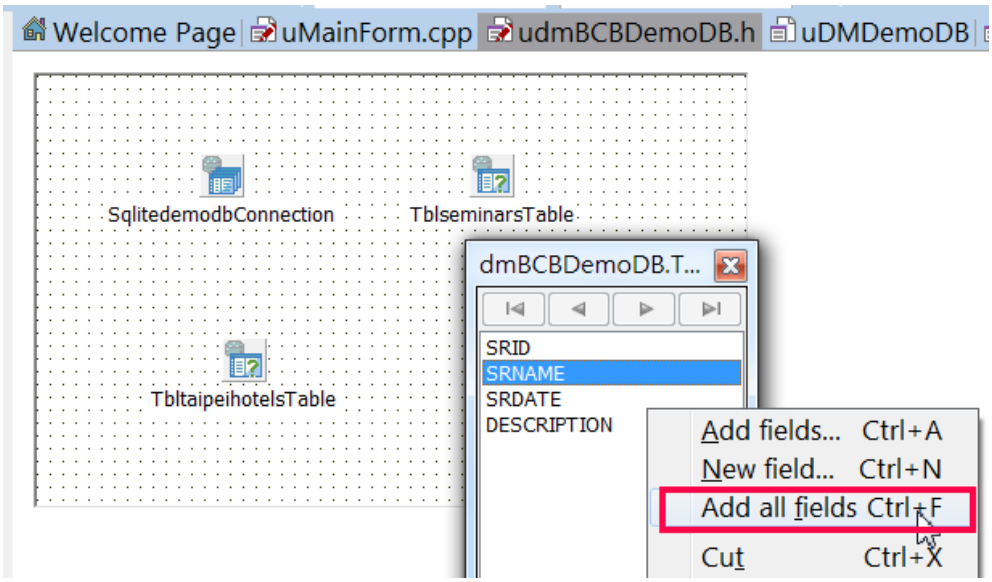



右擊滑鼠 TblseminarsTable 元件並在選單中選擇 Fields Editor...選項：




在欄位編輯器中右擊滑鼠並選擇 Add all fields...加入所有的欄位物件：




回到主表單在  按鈕的 OnClick 事件中呼叫資料模組中 TblseminarsTable 元件的 Insert()方法在資料表中加入一筆新的資料：

```
void __fastcall TfmMainForm::SpeedButton1Click(TObject *Sender)
{
  dmBCBDemoDB->TblseminarsTable->Insert();
  edtSeminarName->SetFocus();
}
```

接著在  按鈕的 OnClick 事件中呼叫資料模組的 PostSeminarData()方法把使用者在主表單中輸入的資料更新回資料表中：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  dmBCBDemoDB->PostSeminarData(dtedtSeminarDate->Date,
edtSeminarName->Text, mmSeminarNote->Lines->Text);
  LinkFillControlToField1->BindList->FillList();
}
```

PostSeminarData()方法先檢查 TblseminarsTable 元件是否在新增或是修改資料的狀態中，如果是的話就把傳入的輸入資料寫入相對應的資料表欄位中最後呼叫 Post()方法把資料真正更新回後端的資料庫中：

```
void TdmBCBDemoDB::PostSeminarData(const TDateTime dtDate, const String
sName, const String sNote)
{
  if ( (TblseminarsTable->State == dsEdit) || (TblseminarsTable->State
== dsInsert) )
  {
    TblseminarsTable->FieldByName("SRDATE")->Value = dtDate;
    TblseminarsTable->FieldByName("SRNAME")->Value = sName;
    TblseminarsTable->FieldByName("DESCRIPTION")->Value = sNote;
    TblseminarsTable->Post();
  }
}
```

例如下圖就是在 Android 手機中新增資料並且更新資料回資料表的畫面：


主表單中其他的按鈕分別實作了修改資料和刪除資料，例如  按鈕呼叫 TblseminarsTable 元件的 Edit()方法讓 TblseminarsTable 進入修改模式：



```
void __fastcall TfmMainForm::SpeedButton2Click(TObject *Sender)
{
  EditSeminarData();
}
//---------------------------------------------------------------------------
void TfmMainForm::EditSeminarData()
{
  SetupSeminarData();
  dmBCBDemoDB->TblseminarsTable->Edit();
  edtSeminarName->SetFocus();
}
```

當然修改完資料之後仍然要呼叫 Post()方法把資料真正更新回後端的資料庫中。

最後要刪除資料表中目前的記錄只需要呼叫 Delete()方法即可：

```
void __fastcall TfmMainForm::SpeedButton3Click(TObject *Sender)
{
  dmBCBDemoDB->TblseminarsTable->Delete();
  LinkFillControlToField1->BindList->FillList();
```

```
}
```

藉由 FireDAC 我們可以非常簡單的就完成一個能在手機中對資料進行 CRUD 功能的 App，讀者可參考本書的 pBCBCRUDDataDemo 範例。

但在這個範例中所有的資料異動在呼叫 Post 方法之後會立刻的更新回後端的資料庫中。但在許多應用中您可能不希望每一筆資料在異動之後立刻更新資料庫，例如：

1. 在異動大量資料時
2. 在 C/S 架構中
3. 在多層架構中
4. 為了執行效率考量
5. …

在上面的情形中您可能希望異動的資料先暫時儲存在客戶端，等到適當的狀況時再更新回後端資料庫中，例如在異動資料到了一定數量，或是過了一段固定的時間後再更新回資料庫中，那麼您可以使用 FireDAC 提供的快儲功能(Cached Update）。



## 2-3-1 使用 FireDAC 快儲功能

FireDAC 快儲功能簡單的說就是在客戶端暫時把所有對於資料的異動都先記錄起來並且把異動的資料先異動在記憶體的資料集中而不會真接更新回後端的資料庫中，一旦程式師決定把異動寫回後端的資料庫中才一次把所有的異動一次更新回後端。

因此要使用 FireDAC 快儲功能，程式碼要進行下列的步驟：

1. 設定 TFDQuery 的 CachedUpdates 特性值為 true
2. 使用 Insert()，Edit()，Delete()和 Post()方法異動資料
3. 呼叫 ApplyUpadates()方法把客戶端的異動一次寫回後端資料庫中
4. 如果資料寫回成功就呼叫 TFDQuery 的 CommitUpdates()方法清除客戶端的異動記錄，如果發生錯誤就呼叫 TFDConnection 的 Rollback()方法取消資料寫回
5. 處理寫回資料發生錯誤的例外狀況

此外 FireDAC 還提供了許多和快儲功能相關的特性和方法，它們都是程式師在使用快儲功能時經常會使用的，下面的表格針對它們做了簡單的說明：

| 方法/特性 | 說明 |
|---|---|
| ChangeCount 特性 | 代表目前在客戶端已經被異動的資料筆數 |
| UpdateStatus 特性 | 代表客戶端中每一筆資料的異動狀況 |
| UpdatesPending 特性 | 代表客戶端是否已經有異動的資料 |
| SavePoint 特性 | 儲存目前異動資料的版本資訊，如此一來程式師可以回到前一個異動狀態 |
| RevertRecord 方法 | 取消對於目前記錄的異動 |
| UndoLastChange 方法 | 回復上一次對於資料集的異動 |
| CancelUpadtes 方法 | 取消所有資料的異動 |

上表中的 UpdateStatus 特性代表目前被異動的資料的型態，例是新增資料，修改的資料或是被刪除的資料，下面說明了 UpdateStatus 每一個特性值的意義：

| UpdateStatus特性值 | 說明 |
|---|---|
| usUnmodified | 這筆資料沒有被異動過 |
| usModified | 這筆資料已經被修改過 |
| usInserted | 這筆資料是新增的資料 |
| usDeleted | 這筆資料已經被刪除 |

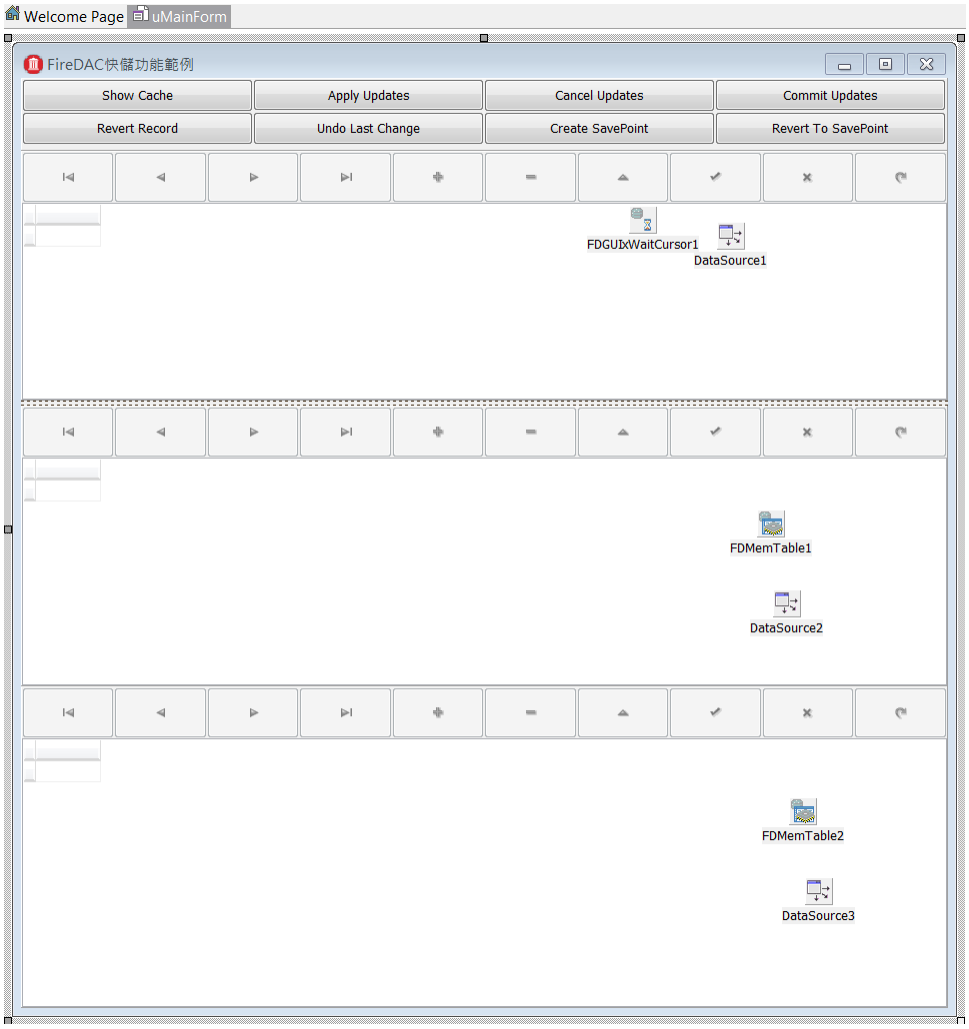


在使用 FireDAC 快儲功能時程式師也需要知道另外 2 個重要的特性，那就是 TFDQuery 的 Data 和 Delta 特性，下面的表格介紹了這 2 個特性：

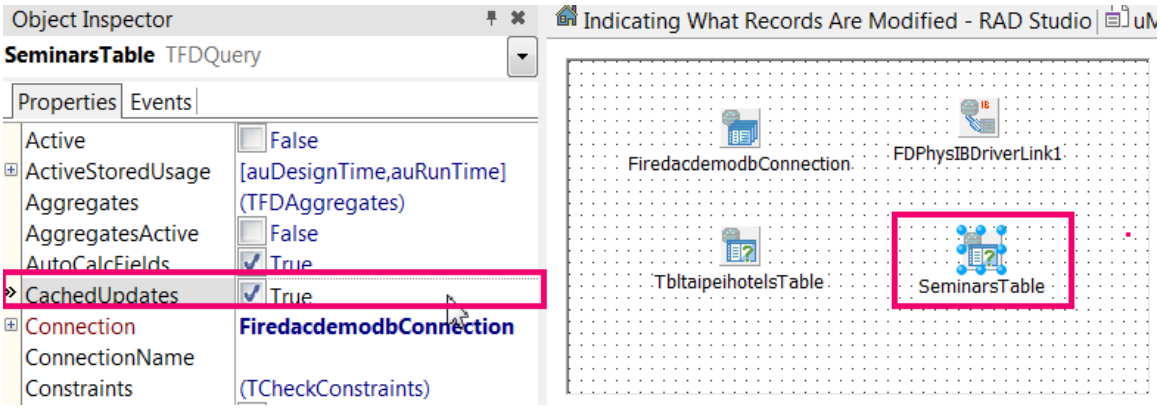
| TFDQuery特性 | 說明 |
|---|---|
| Data | TFDQuery 執行 SQL 命令取得的資料集便儲存在 Data 特性中 |
| Delta | 使用者對於 TFDQuery 的 Data 特性值中的資料進行的異動就儲存在 Delta 特性中 |

藉由上面的方法和特性程式師就可以充分的掌握 FireDAC 快儲功能，讓我們使用一個範例來說明如何使用 FireDAC 快儲功能。

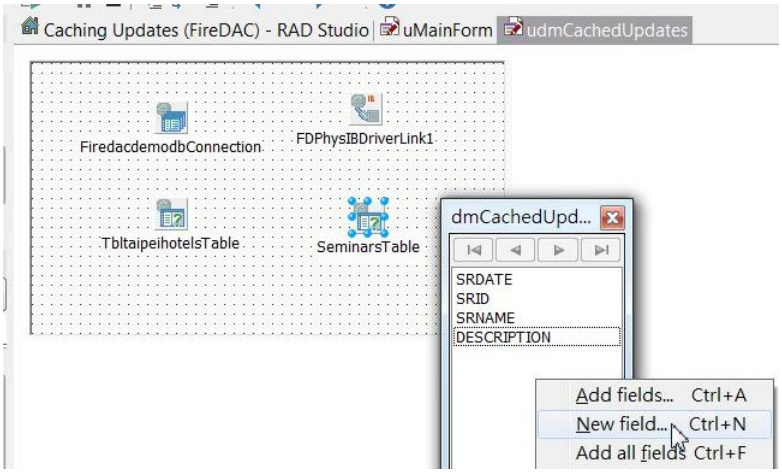
首先建立一個 VCL 應用程式專案並且設定主表單如下：




接著建立一個資料模組在其中使用 TFDConnection，TFDQuery 和 TFDPhysIBDriverLink 元件連結到範例資料表 Seminars，當然我們需要設定 SeminarsTable 元件的 CachedUpdates 特性值為 True 以開啟快儲功能，如下所示：

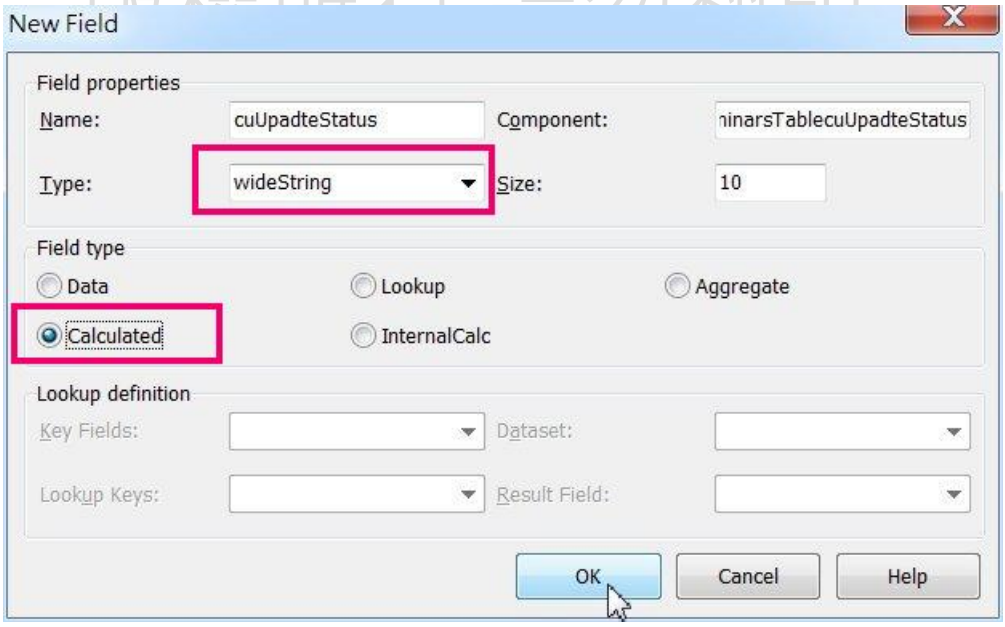


現在讓我們在 SeminarsTable 元件中加入一個計算欄位以便用來顯示每一筆資料的異動狀態。所謂”計算欄位”是指這個欄位並不真正儲存在資料表中，而是在程式執行時動態建立的暫時欄位。要為 TFDQuery 加入計算欄位請使用滑鼠右擊 SeminarsTable 元件再選擇 New field...如下所示：




接著在 New Field 對話盒中的 Name 欄位輸入此計算欄位的名稱，再選擇它的資料型態為 wideString，Field Type 為 Calculated，如下所示：

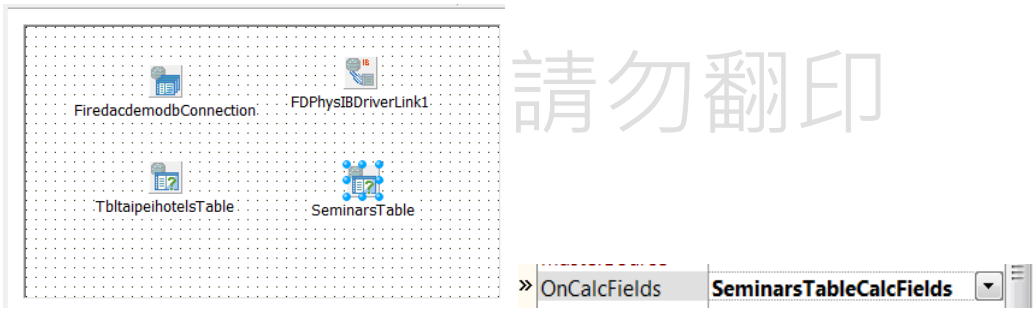





如此一來我們就在 SeminarsTable 元件中加入了一個名為 SeminarsTablecuUpadteStatus 的計算欄位物件了，稍後就可以使用它來顯示資料的異動狀況了。

首先我們希望此範例程式一開始執行時能夠自動顯示目前資料的異動狀態，例如下圖所示。由於現在沒有任何資料被異動，因此每筆資料的狀態都是”未異動”：


由於我們前面已經定義了計算欄位，因此請為 SeminarsTable 元件定義 OnCaclFields 事件處理函式：





接著在此件處理函式判斷 UpdateStatus 特性值並且在計算欄位'cuUpadteStatus'中寫入相對應的異動狀況如下：

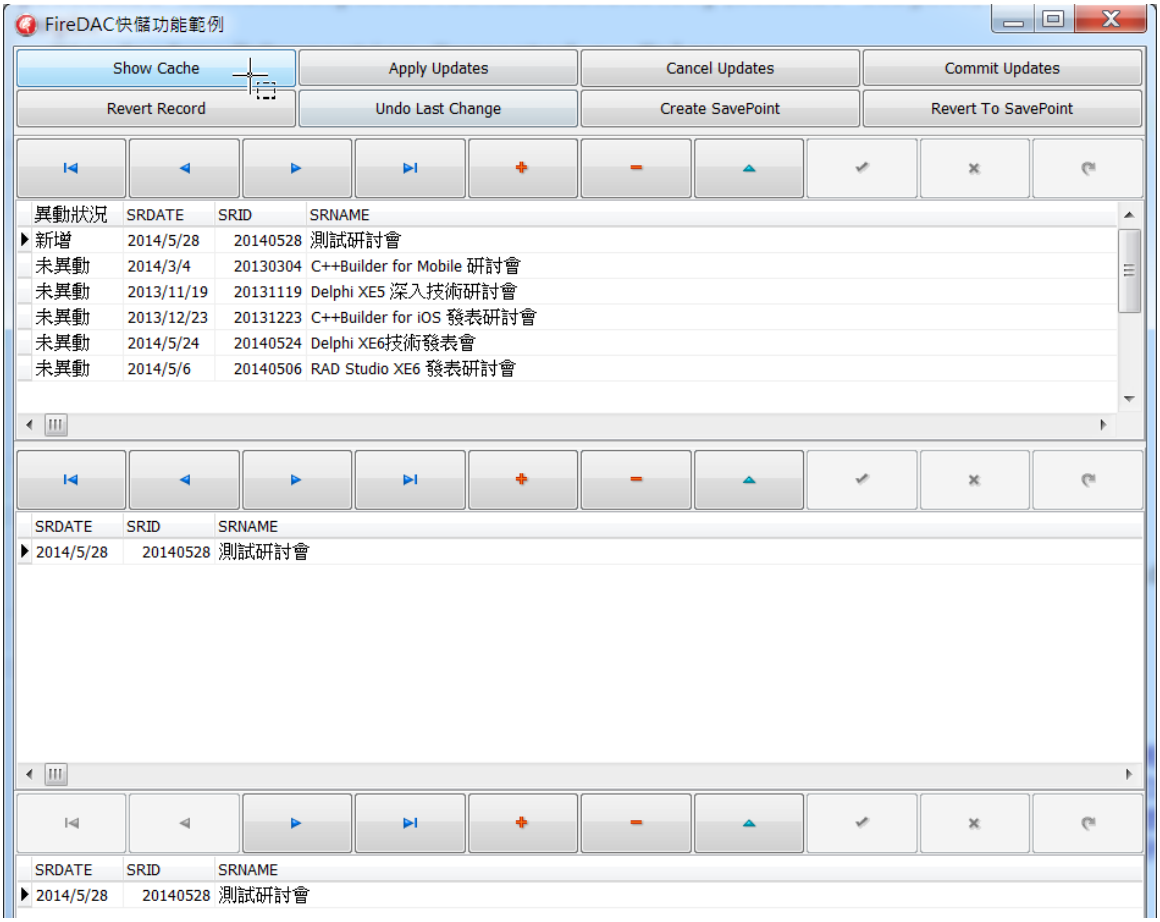
```
void __fastcall TdmCachedUpdates::SeminarsTableCalcFields(TDataSet
*DataSet)
{
  switch (DataSet->UpdateStatus())
  {
    case usUnmodified :
    {
      DataSet->FieldByName("cuUpadteStatus")->AsString = "未異動";
      break;
    }
```

```
    case usModified :
    {
      DataSet->FieldByName("cuUpadteStatus")->AsString = "修改過";
      break;
    }
    case usInserted :
    {
      DataSet->FieldByName("cuUpadteStatus")->AsString = "新增";
      break;
    }
    case usDeleted :
    {
      DataSet->FieldByName("cuUpadteStatus")->AsString = "刪除";
      break;
    }
  }
}
```

現在如果您執行此範例程式就可以看到類似上面的執行畫面了。



接著如果我們在資料表中加入一筆新的資料並且希望能看到快儲功能中的異動資料，如下所示，




那麼我們可以使用 2 種方式，第 1 種方式是使用過濾功能把 SeminarsTable 元件中被異動的資料過濾出來再顯示出來。主表單中的 Show Cache 按鈕就提供這種方式，下面是它的實作程式碼：

```
void __fastcall TfmMainForm::Button3Click(TObject *Sender)
{
  if (dmCachedUpdates->SeminarsTable->Active)
  {
    if (FDMemTable1->Active)
      FDMemTable1->Close();
    FDMemTable1->CloneCursor(dmCachedUpdates->SeminarsTable, true);
    FDMemTable1->FilterChanges.Clear();
    FDMemTable1->FilterChanges << Firedac::Comp::Dataset::rtModified
<< Firedac::Comp::Dataset::rtInserted <<
Firedac::Comp::Dataset::rtDeleted;
  }
```

```
}
```

Button3Click 使用了 TFDMemTable 元件 FDMemTable1 的 CloneCursor()方法為 SeminarsTable 中的資料建立另外一個 Data View，再過濾其中被修改，新增和刪除的資料並且顯示在主表單中間的 DataGrid 元件中。

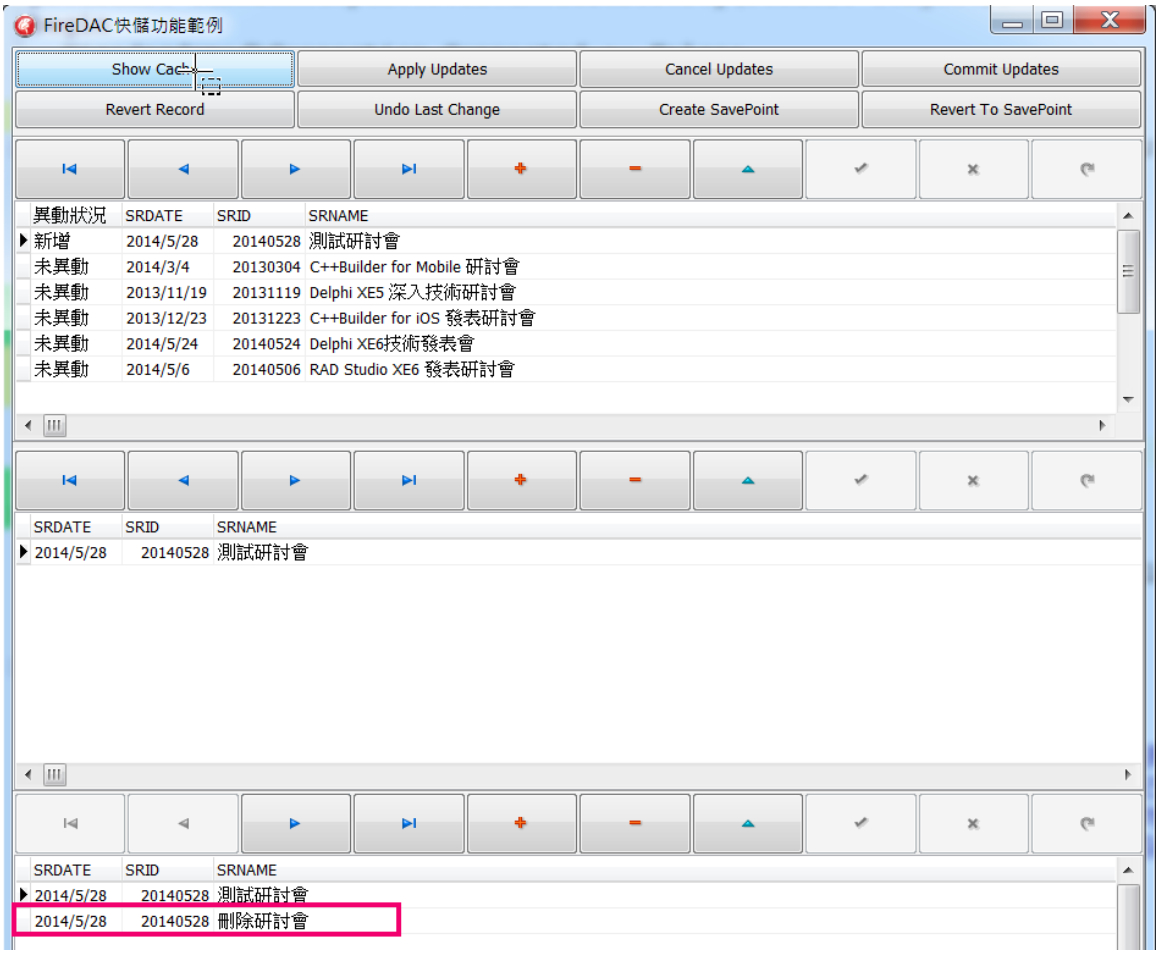
第 2 種方式更簡單，那就是直接顯示 SeminarsTable 的 Delta 特性值，因此我們可以在 SeminarsTable 元件的 OnAfterPost 事件中撰寫如下的程式碼：

```
void __fastcall TdmCachedUpdates::SeminarsTableAfterPost(TDataSet
*DataSet)
{
  fmMainForm->FDMemTable2->Active = false;
  fmMainForm->FDMemTable2->Data = SeminarsTable->Delta;
  fmMainForm->FDMemTable2->Active = true;
}
```

在上面的程式碼中直接把 SeminarsTable 元件的 Delta 特性值指定給 TFDMemTable 元件 FDMemTable2 的 Data 特性值就可以在主表單下方的 DataGrid 元件中顯示新增的資料了。

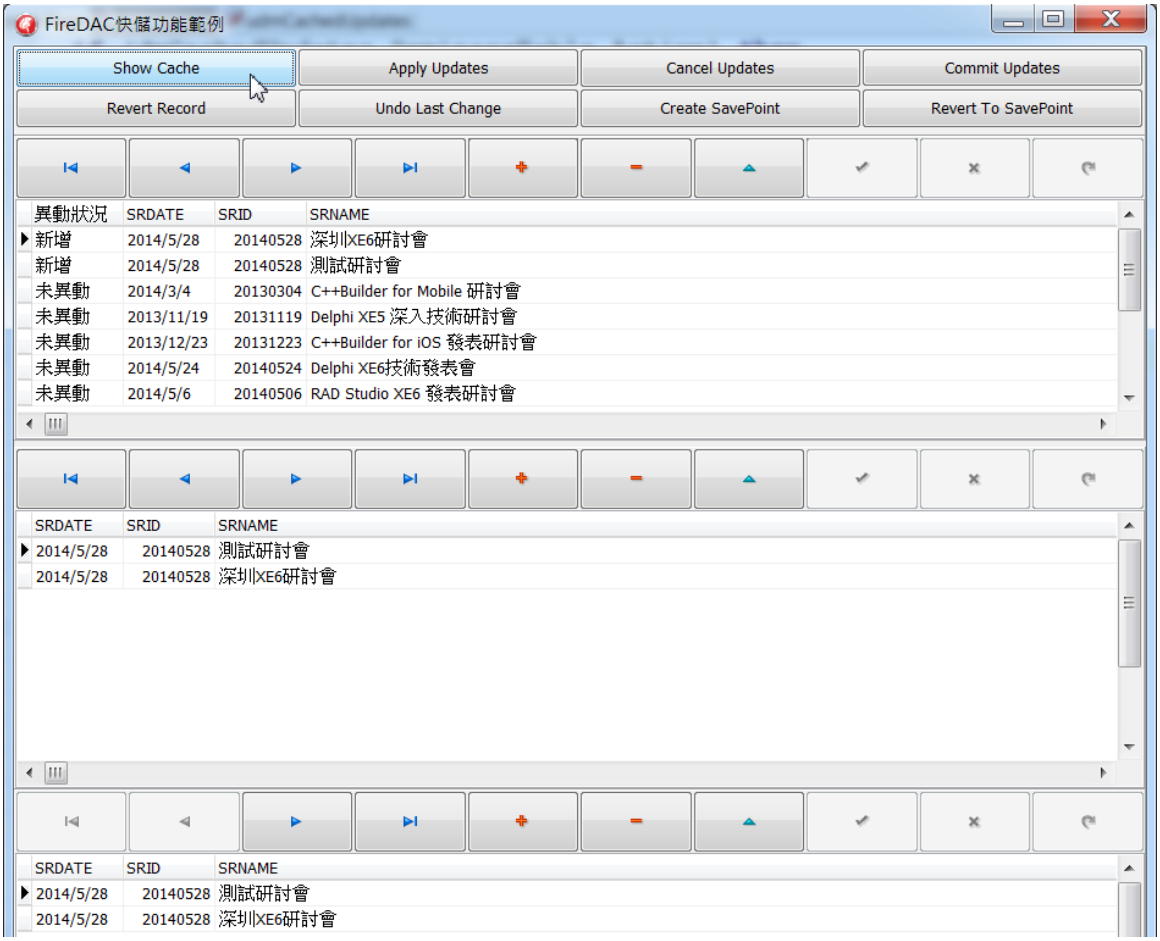


現在如果我們立刻再刪除剛才就增的資料就可以看到如下的畫面：




從上面的畫面我們可以看到 TFDQuery 元件的 Delta 特性值的確可記錄所有的異動資料狀態。

如果我們在 SeminarsTable 元件中增加一些資料並且希望真的一次更新回後端的資料表，如下所示：



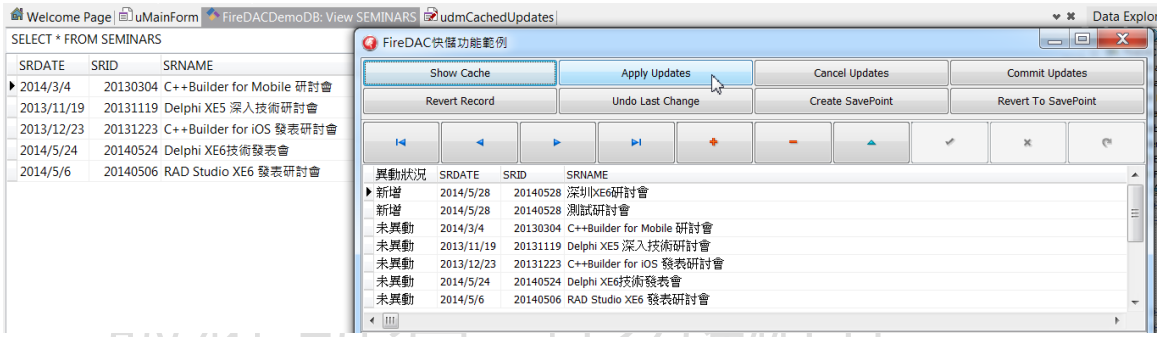

那麼可以在主表單的”Apply Updates”按鈕中實作如下的程式碼：

```
001    void __fastcall TfmMainForm::Button4Click(TObject *Sender)
002    {
003      dmCachedUpdates->FiredacdemodbConnection->StartTransaction();
004      int Errors = dmCachedUpdates->SeminarsTable->ApplyUpdates(-1);
005      if (Errors > 0)
006      dmCachedUpdates->FiredacdemodbConnection->Rollback();
007      else
008      dmCachedUpdates->FiredacdemodbConnection->Commit();
009      dmCachedUpdates->SeminarsTable->CommitUpdates();
010      dmCachedUpdates->SeminarsTable->Refresh();
011
012      if (Errors > 0)
013
    dmCachedUpdates->HandleMyReconcileError(dmCachedUpdates->Seminars
```

```
Table);
014    }
```

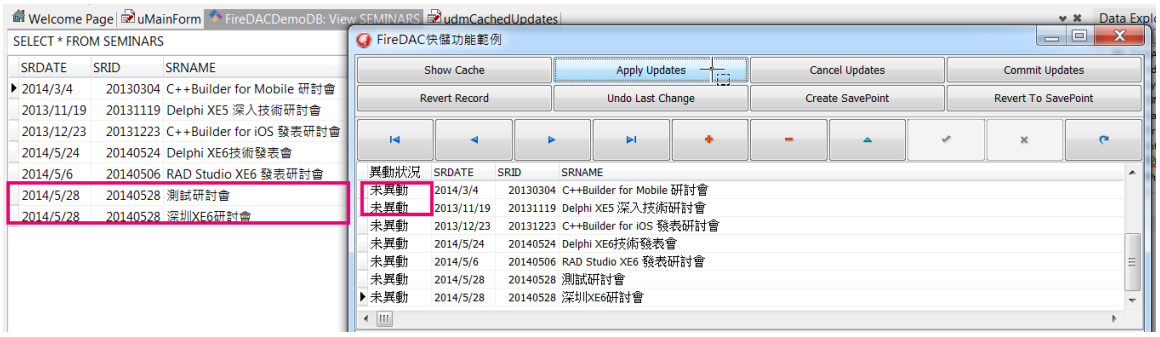
由於快儲功能可能一次更新多筆資料回後端資料庫，因此 003 行先呼叫 TFDConnection 元件的 StartTransaction()方法啟動 FireDAC 的交易模式，004 行呼叫 ApplyUpdates()方法把所有的異動資料更新回去。ApplyUpdates()方法的-1 參數代表更新的資料不可以發生任何錯誤，如果發生錯誤就於 008 行呼叫 Rollback()方法回覆更新資料的動作，如果沒有發生錯誤就於 008 行呼叫 Commit()方法確定更新資料的動作。最後 009 行呼叫 CommitUpdates()方法清除快儲功能中的資料。

下圖同時顯示了後端資料表中的資料以及範例應用程式使用快儲功能新增 2 筆資料，讀者可以看到範例應用程式新增的資料此時並沒有真的更新到後端資料表中：






一直到使用者點選”Apply Updates”按鈕後範例應用程式新增的資料才一次更新回端資料表中：




在使用快儲功能時程式師也可以使用數個方法來控制異動資料，例如在開始異動資料之前程式師可以先建立一個 SavePoint，如果因為特定原因應用程式需要放棄上次 SavePoint 到現在之間異動的資料，那麼程式師可以回到上一次建立的 SavePoint，那麼所有的異動都會自動恢復。

## SavePoint

主表單中的『Create SavePoint』按鈕就是儲存 SavePoint：

```
void __fastcall TfmMainForm::Button9Click(TObject *Sender)
{
  iSavePoint = dmCachedUpdates->SeminarsTable->SavePoint;
  ShowDeltaData();
}
```

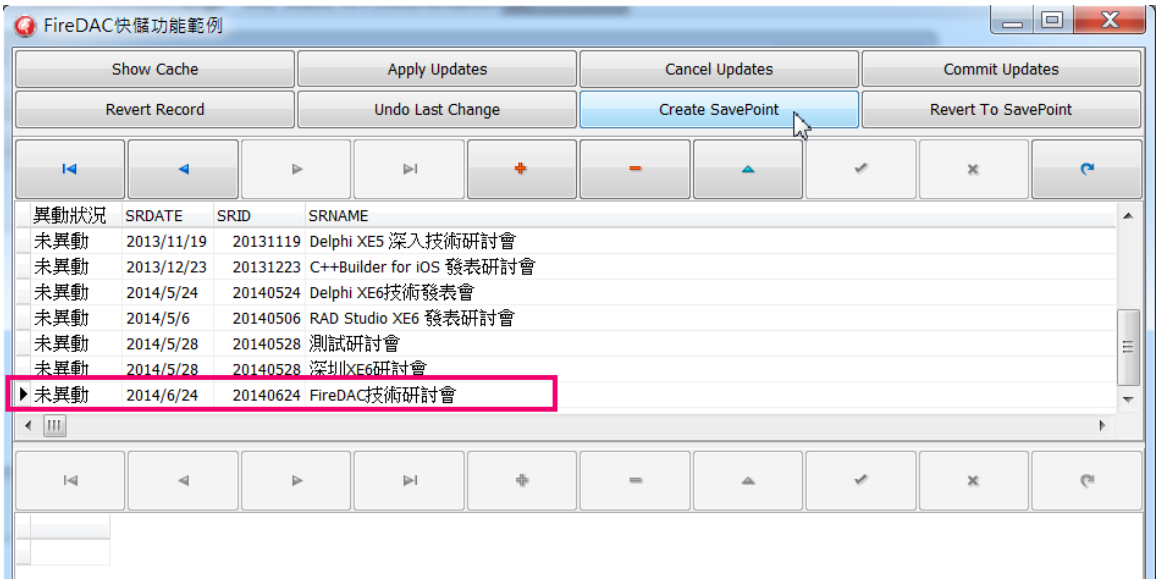
而 iSavePoint 只是型態為 Integer 的變數：

```
    int     iSavePoint;
```

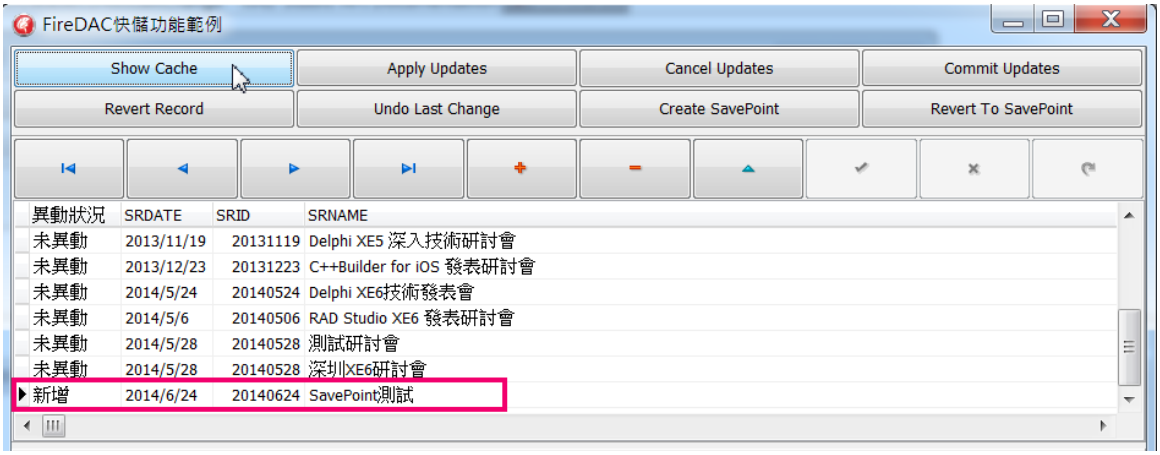
如果要放棄異動，只需要點選主表單中的『Revert To SavePoint』按鈕，它把上代建立儲存的 SavePoint 更新回 SeminarsTable 元件的 SavePoint 特性值：

```
void __fastcall TfmMainForm::Button10Click(TObject *Sender)
{
  dmCachedUpdates->SeminarsTable->SavePoint = iSavePoint;
  ShowDeltaData();
  iSavePoint = 0;
}
```

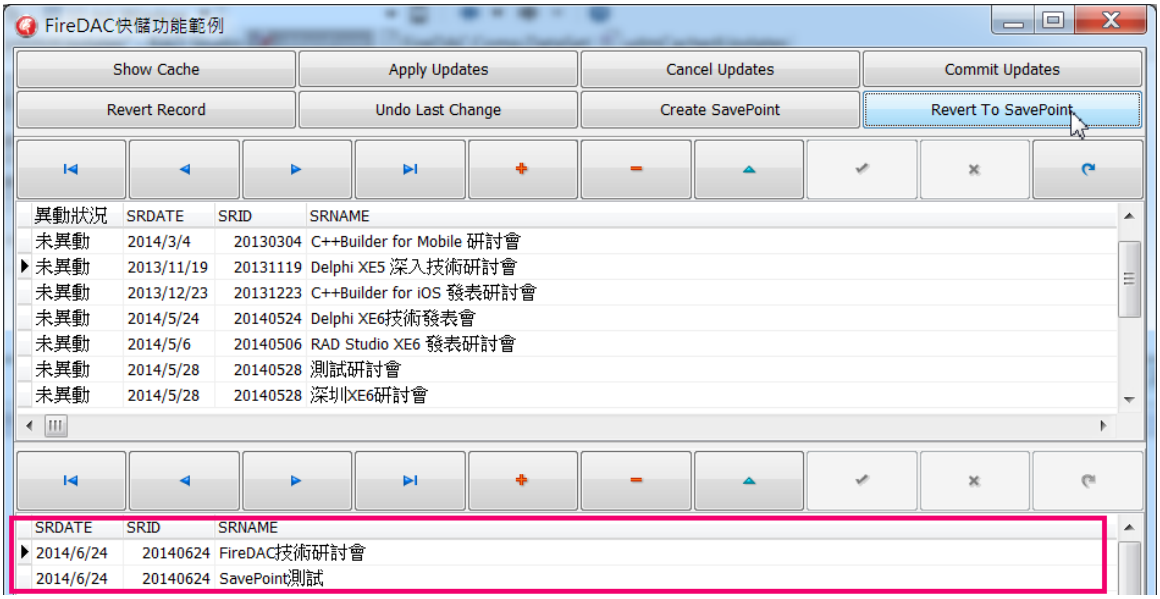
下面的數個畫面顯示了 SavePoint 方法的功能。下圖顯示資料從後端資料表中取出並且點選『Create SavePoint』按鈕建立 SavePoint：



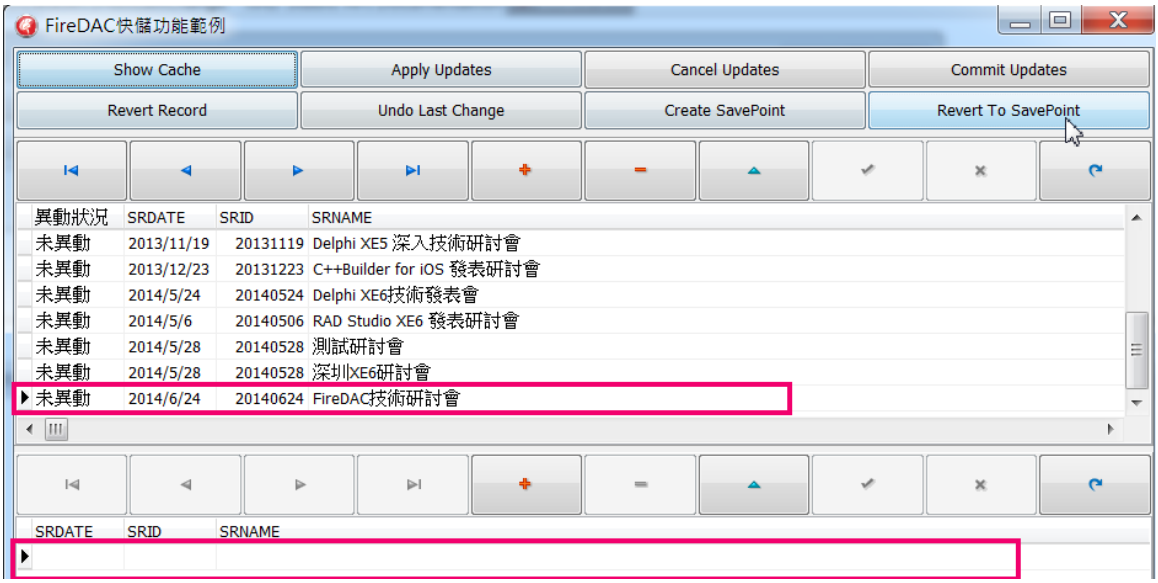
接著在其中加入一筆新的資料：




再刪除 FireDAC 技術研討會這筆資料，此時快儲中有 2 筆資料的異動：




現在點選『Revert To SavePoint』按鈕，從下圖可以看到所有的異動都恢復到一開始異動資料的動作時：



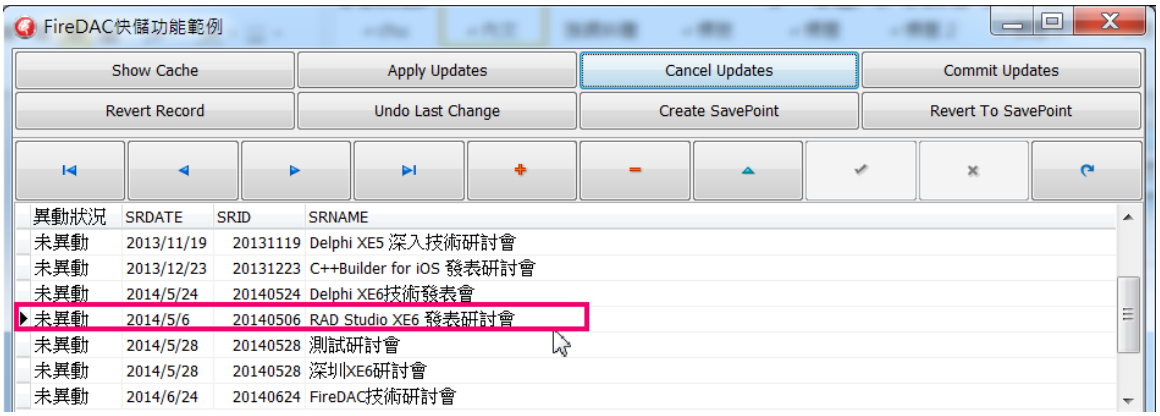

## RevertRecord 方法

TFDQuery 的 RevertRecord 方法可取消對於目前記錄的異動，主表單中的『Revert Record』按鈕實作了下面的程式碼，它先判斷目前快儲的資料如果不是未異動過的資料就呼叫 RevertRecord()方法回復對資料的異動：

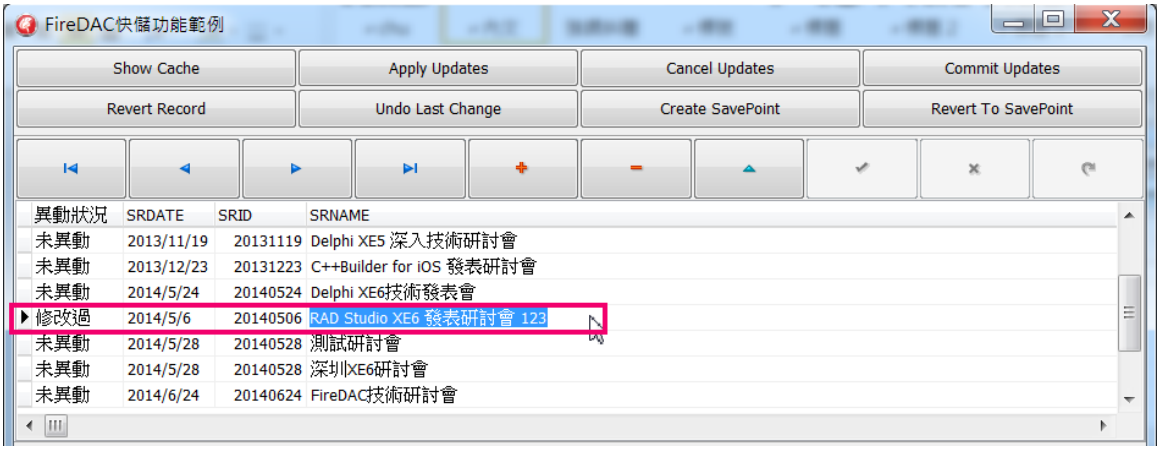


```
void __fastcall TfmMainForm::Button7Click(TObject *Sender)
{
  if (dmCachedUpdates->SeminarsTable->UpdateStatus() != usUnmodified)
     dmCachedUpdates->SeminarsTable->RevertRecord();
  ShowDeltaData();
}
```

下面的畫面顯示了 RevertRecord 方法如何工作。例如一開始沒有異動 RAD Studio XE6 的資料：



接著修改這筆資料：




最後如果點選主表單中的『Revert Record』按鈕就可以看到 RAD Studio XE6 回復到原始的狀態。

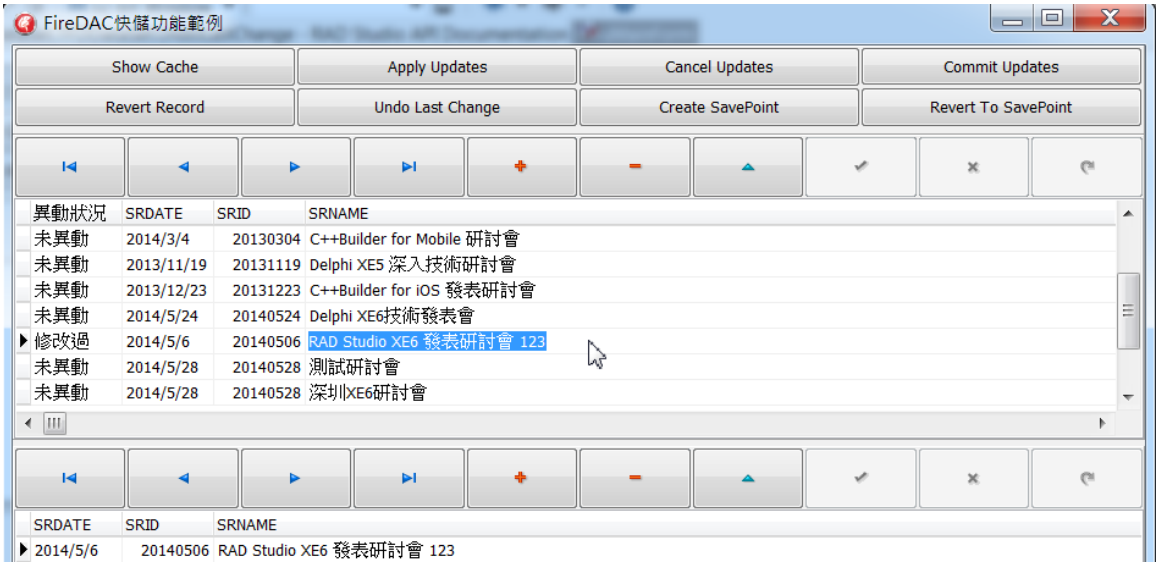
## CommitUpdates 方法

TFDQuery 的 CommitUpdates()方法可以清除所有的快儲異動，主表單中的『Commit Updates』按鈕實作了下面的程式碼：

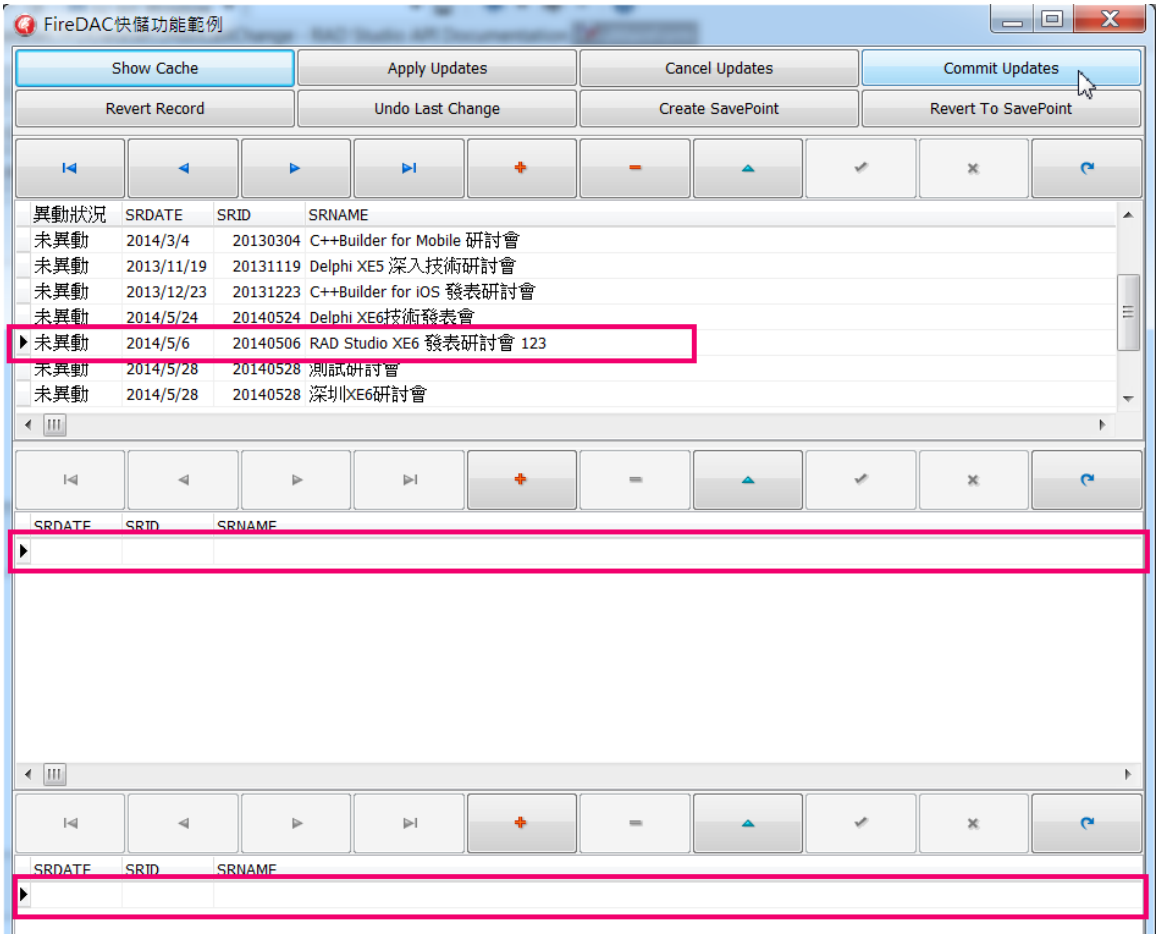


```
void __fastcall TfmMainForm::Button6Click(TObject *Sender)
{
  dmCachedUpdates->SeminarsTable->CommitUpdates();
  ShowDeltaData();
}
```

現在讓我們修改一筆資料：




再點選『Commit Updates』按鈕，從下圖我們可以看到快儲異動(Delta)已經被清除了：



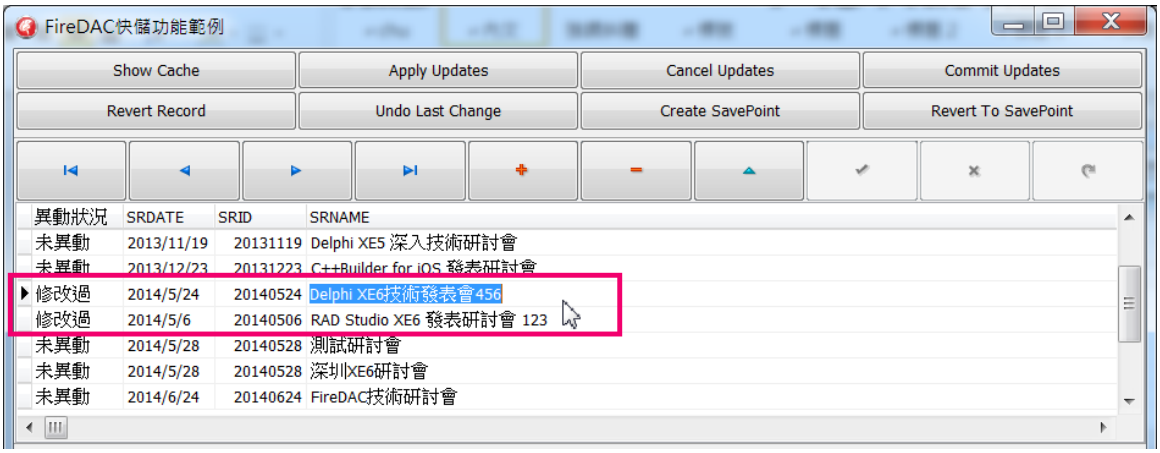
CommitUpdates 只會清除所有的快儲異動，但並不會把異動資料更新回後端資料表中。

## UndoLastChange 方法

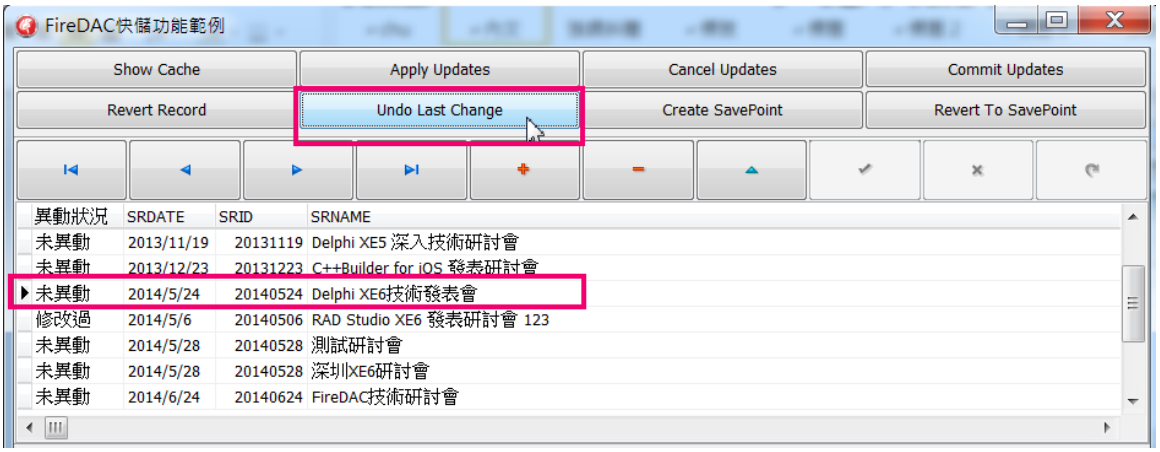
TFDQuery 的 UndoLastChange 方法可以以反相的方向逐一的恢復前一次對於資料的異動，一直到最開始的狀態。下面是主表單中的『Undo Last Change』按鈕的實作程式碼

```
void __fastcall TfmMainForm::Button8Click(TObject *Sender)
{
  dmCachedUpdates->SeminarsTable->UndoLastChange(true);
  ShowDeltaData();
}
```

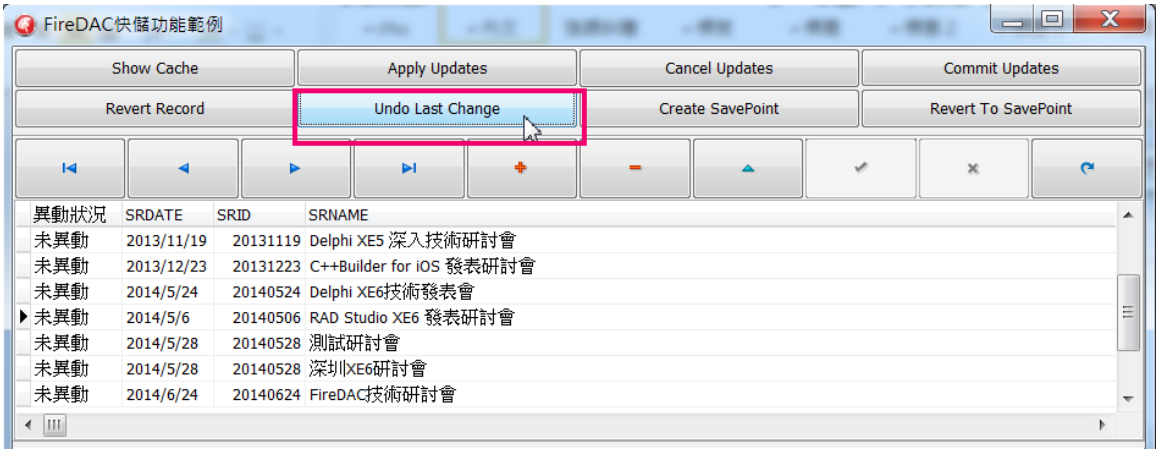
現在讓我們修改 2 筆資料，先修改 RAD Studio XE6 再修改 C++Builder XE6：




接著點選主表單中的『Undo Last Change』按鈕，從下圖可以看到 C++Builder XE6 這筆資料被恢復了：




再次點選主表單中的『Undo Last Change』按鈕就可以看到 RAD Studio XE6 的資料也恢復了：




讀者可以參考本書的 pCachedUpdatesDemo.dproj 範例。

## 2-3-2 處理 FireDAC 快儲更新錯誤

在使用 FireDAC 快儲功能時，由於在客戶端異動的資料是一次更新回後端，因此在更新資料時可能會發生問題，例如主鍵值衝突，資料欄位資料值已經被其他人/其他客戶端異動了或是資料已被其他人/其他客戶端刪除了等狀況。這些狀況都會造成 ApplyUpdates 方法產生錯誤，對於無法成功更新回後端的資料，FireDAC 會為每一筆無法更新的資料觸發一次 OnReconcileError 事件處理函式，程式師需要在 OnReconcileError 事件處理函式處理這些產生錯誤的資料。

下面是 OnReconcileError 事件處理函式的宣告原型：

```
void __fastcall
TdmCachedUpdates::SeminarsTableReconcileError(TFDDataSet *DataSet,
          EFDException *E, TFDDatSRowState UpdateKind,
TFDDAptReconcileAction &Action)

{
  …
}
```

下面的表格說明了 OnReconcileError 事件處理函式參數的意義：

| OnReconcileError事件參數 | 說明 |
|---|---|
| DataSet | 目前進行更新並發生問題的資料集物件 |
| E | 例外錯誤物件，可從其中找出發生錯誤的原因 |
| UpdateKind | 發生錯誤的異動，其可能的數值是 rsInserted, rsDeleted, rsModified 或 rsUnchanged |
| Action | 此 OnReconcileError 事件處理函式要採取的動作 |

在 OnReconcileError 事件處理函式中程式師的工作就是處理這個異動錯誤並且指定適當的值給 Action 參數，下面的表格說明了可指定給 Action 的數值以及其意義：



| Action參數值 | 說明 |
|---|---|
| raSkip | 跳過這筆資料 |
| raAbort | 離開 Reconcile 方法 |
| raMerge | 內定值，清除目前記錄的錯誤資訊並且把異動資料合併到 TFDQuery 元件的資料集中 |
| raCorrect | 清除目前記錄的錯誤資訊並重新設定更新狀態，等下次再次呼叫 ApplyUpadtes 方法時再次更新回後端 |
| raCancel | 取消目前記錄的異動 |
| raRefresh | 取消目前記錄的異動並重新從後端資料表中取出資料 |

而當程式師在 OnReconcileError 事件處理函式中處理錯誤時，可使用 TFDQuery 的下面 2 個特性來取得每一個欄位的原始值以及異動過的數值：

| TFDQuery特性值 | 說明 |
|---|---|
| OldValue | 從後端取得的原始欄位資料值 |
| CurValue | 目前的欄位資料值 |
| NewValue | 目前的欄位資料值，可能是經過異動的資料值 |

例如如果程式師現在使用 FireDAC 快儲功能一次新增 10 筆資料，假設其中一筆資料的新增 ID 欄位值為”201406260945”並且呼叫 ApplyUpdates 把資料更新回後端，但在此之前或同時有另外一個客戶端也新增了一筆資料並且使用了相同的新增 ID”201406260945”，那麼 FireDAC 快儲就會產生鍵值衝突的錯誤而觸發 OnReconcileError 事件處理函式要求程式師解決，那麼程式師可以在 OnReconcileError 事件處理函式中使用如下的程式碼來處理：

```
001    void __fastcall
TdmCachedUpdates::SeminarsTableReconcileError(TFDDataSet *DataSet,
002           EFDException *E, TFDDatSRowState UpdateKind,
TFDDAptReconcileAction &Action)
003
004    {
005      if (dynamic_cast<EFDException*>(E))
006      {
007      EFDDBEngineException *ee =
dynamic_cast<EFDDBEngineException*>(E);
008        if ( (UpdateKind == rsModified) &&  (ee->Kind == ekUKViolated) )
009          DataSet->FieldByName("ID")->AsInteger = GetNextFreeID();
010      Action = raCorrect;
011      }
012    }
```

008 行先判斷是否是新增資料更新的錯誤並且是因為鍵值衝突，如果是的話就在 009 行為產生錯誤的這筆資料產生一個新的 ID，最後 010 行設定 Action 參數為 raCorrect 以便下次再呼叫 ApplyUpdates()方法時把這筆資料更新回後端。

讓我們使用一個範例來說明，讓我們使用 FireDAC 快儲功能先修改資料，但在呼叫 ApplyUpdates()方法更新回後端之前先修改後端資料表中相同的資料，再回頭執行 ApplyUpdates()方法，這樣做是模擬另外一個使用者已經先修改了相同的資料而會造成 FireDAC 快儲功能觸發 OnReconcileError()。

請在前面的範例專案中加入一個新的表單，其中 3 個 TEdit 元件分別顯示 OldValue，CurValue 和 NewValue 數值，如下所示：


接著為專案中資料模組的 SeminarsTable 元件撰寫如下的 OnReconcileError 事件處理函式：

```
001    void __fastcall
TdmCachedUpdates::SeminarsTableReconcileError(TFDDataSet *DataSet,
002           EFDException *E, TFDDatSRowState UpdateKind,
TFDDAptReconcileAction &Action)
003
004    {
005      if (dynamic_cast<EFDException*>(E))
006      {
007      EFDDBEngineException *ee =
dynamic_cast<EFDDBEngineException*>(E);
008        if ( (UpdateKind == rsModified) &&  (ee->Kind ==
ekNoDataFound) )
009        {
010          if
(TVarData(DataSet->FieldByName("SRNAME")->OldValue).VType != varNull)
011            fmUpdateError->edtOldValue->Text =
DataSet->FieldByName("SRNAME")->OldValue;
012          if
(TVarData(DataSet->FieldByName("SRNAME")->CurValue).VType != varNull)
013            fmUpdateError->edtCurValue->Text =
DataSet->FieldByName("SRNAME")->CurValue;
014          if
(TVarData(DataSet->FieldByName("SRNAME")->NewValue).VType != varNull)
015            fmUpdateError->edtNewValue->Text =
```

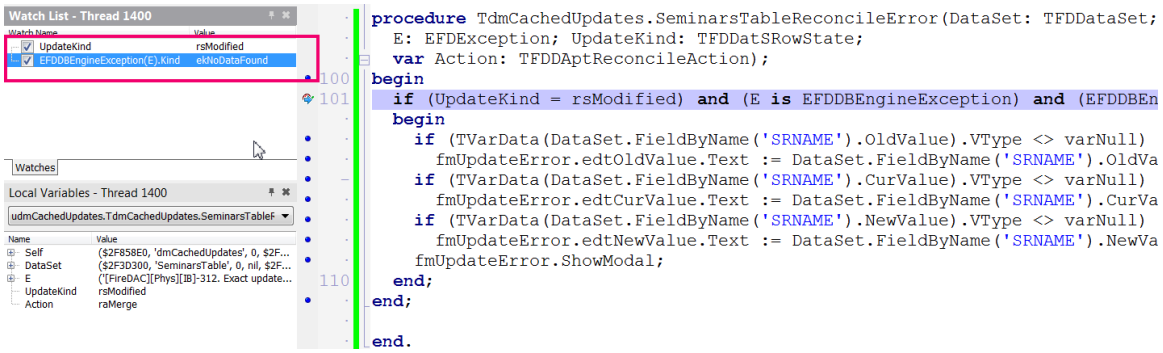
```
DataSet->FieldByName("SRNAME")->NewValue;
016          fmUpdateError->ShowModal();
017        }
018      }
019    }
```

由於我們是模擬 2 個使用者修改了相同資料的錯誤，因此 008 行先判斷 UpdateKind 是不是修改更新的錯誤，如果是的話再判斷是不是 ekNoDataFound 的錯誤，ekNoDataFound 代表 FireDAC 快儲要更新的資料已經找不到了，也就是已經被其他客戶端先修改了(或是刪除了)。如果 2 個條件都符合就把修改欄位的 OldValue，CurValue 和 NewValue 值顯示出來。

現在執行此範例程式，下面是此時後端資料表中的資料，此時紅線包圍的資料其 SRNAME 欄位值是『RAD Studio XE6 發表研討會』：


接著在範例程式中修改『RAD Studio XE6 發表研討會』為『RAD Studio XE6 發表研討會 123』，接著回後端把『RAD Studio XE6 發表研討會』改為『RAD Studio XE6 發表研討會 321』，如下所示：

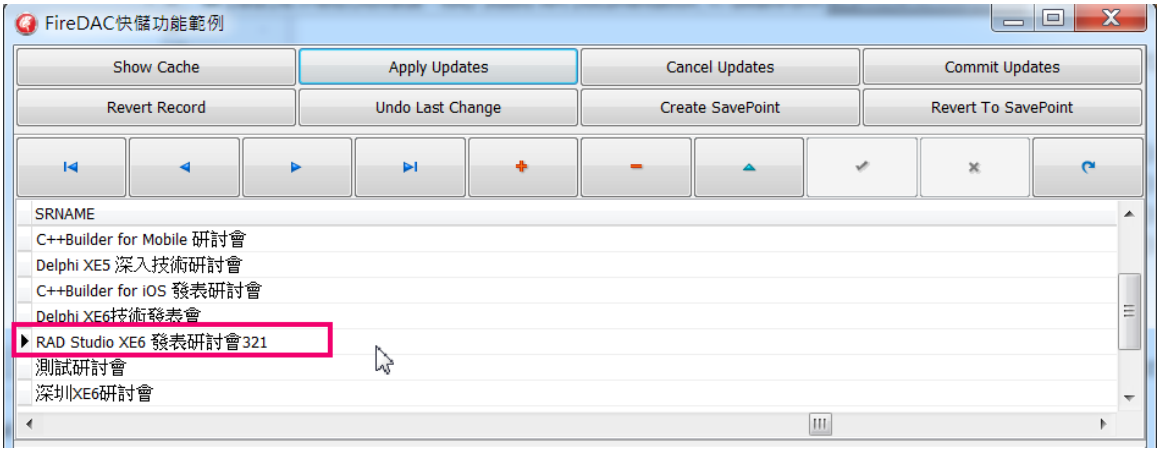
現在點選主表單中的『Apply Updates』按鈕把範例程式中修改的『RAD Studio XE6 發表研討會 123』更新回後端。由於現在後端的『RAD Studio XE6 發表研討會』已經被其他客戶端改為『RAD Studio XE6 發表研討會 321』，因此現在 FireDAC 快儲試著更新時已經找不到原先的『RAD Studio XE6 發表研討會』，就會觸發 OnReconcileError 事件，從下面的除錯畫面讀者可以看到此時 UpdateKind 的數值的確是 rsModified，而且 (EFDDBEngineException(E).Kind 的數值是 ekNoDataFound：




而且新的表單也顯示了 OldValue 和 CurValue 的數值：



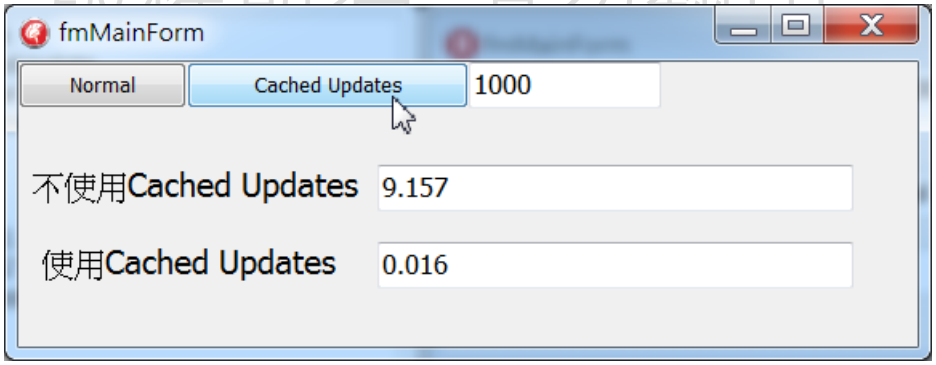

由於在前面的 SeminarsTableReconcileError 中我們沒有指定 Action 的數值，因此 SeminarsTableReconcileError 結束時 Action 是被設定為 raMerge，也就是發生錯誤的欄位值會被合併更新為後端的最新數值，因此在關閉上面的表單後範例程式的主表單會顯示如下的結果，SENAME 欄位的值會成為『RAD Studio XE6 發表研討會 321』：




## 2-3-3 處理 FireDAC 快儲執行效率

使用 FireDAC 快儲功能的目的就是增加執行效率並且在多個客戶端的應用中減少網路的使用流量，只要程式師做好處理 OnReconcileError 事件中更新的錯誤，一般來說 FireDAC 快儲功能的確能增加不少的執行效率，例如下圖就是使用一般更新功能以及使用 FireDAC 快儲功能來隨機更新資料，從下圖可以看到在隨機增加 1000 筆資料的情形下 FireDAC 快儲功能比使用一般更新功能快上了許多：






# 2-4 監督資料處理

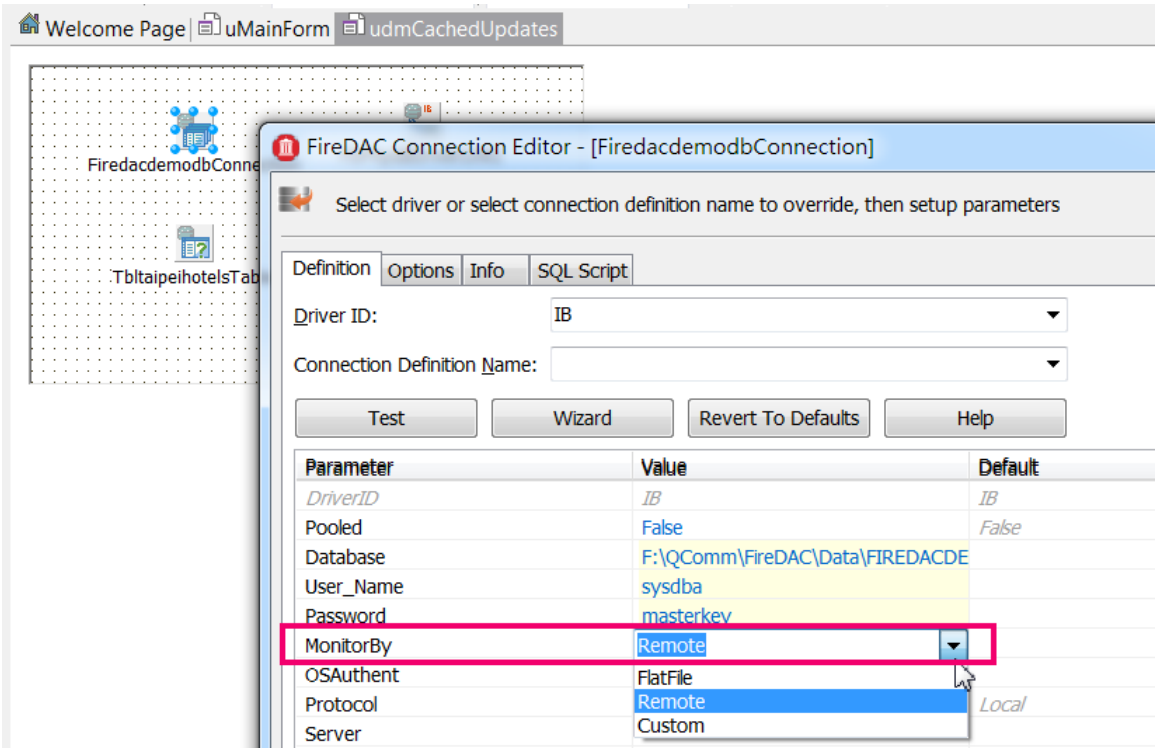
在使用 FireDAC 開發應用程式時，程式師可能需要監督或是觀察 FireDAC 如何處理資料，因此 FireDAC 提供了非常完善的方式讓程式師可以擷取或是監督/觀察 FireDAC 處理資料使用的命令和方法。

FireDAC 可把這些資料儲存在文字檔，程式師自行使用的方式或是顯示在 FireDAC 的 Monitor Explorer 中，以便幫助程式師掌握資料處理特性。FireDAC 提供了 3 個元件讓程式師監督/觀察前端 FireDAC 應用程式和後端資料庫之間的互動：

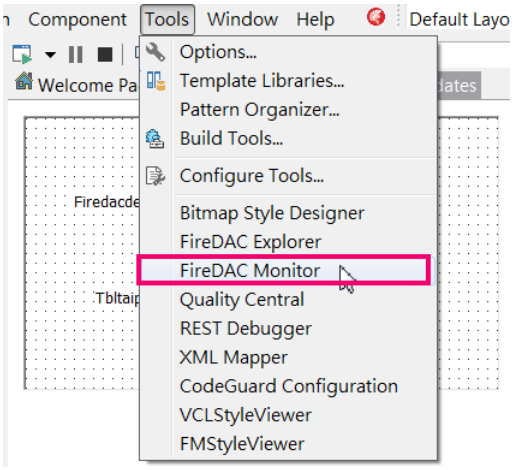
| 元件 | 說明 |
| --- | --- |
| TFDMoniRemoteClientLink | 把前端 FireDAC 應用程式和後端資料庫之間處理資料的方式顯示在 FireDAC Monitor Explorer 工具中 |
| TFDMoniCustomClientLink | 把前端 FireDAC 應用程式和後端資料庫之間處理資料的方式輸出到程式師定案的目標中 |
| TFDMoniFlatFileClientLink | 把前端 FireDAC 應用程式和後端資料庫之間處理資料的方式輸出到文字檔目標中 |

讓我們來說明一下如何使用 TFDMoniRemoteClientLink 元件，因為藉由使用這個元件和 FireDAC Monitor Explorer 工具程式師可以充分掌握 FireDAC 應用程式處理資料的行為。請回到前面的快儲範例專案中，現在讓我們使用 TFDMoniRemoteClientLink 和 FireDAC Monitor Explorer 工具來觀察為什麼前面修改『RAD Studio XE6 發表研討會』為『RAD Studio XE6 發表研討會 123』時如果其他客戶端已經修改了這筆資料的話就會產生 OnReconcileError。

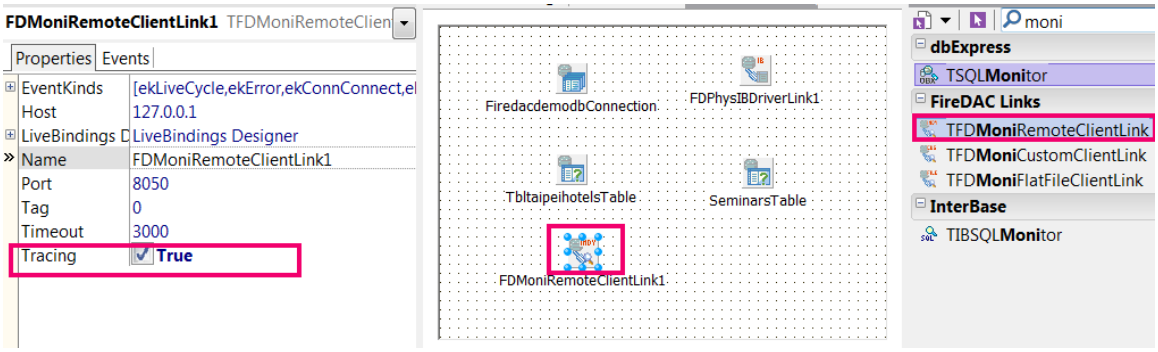
首先開啟資料模組中的 FiredacdemodbConnection 元件的元件編輯對話盒，在它的 MonitorBy 選項中選擇 Remote，如下所示：



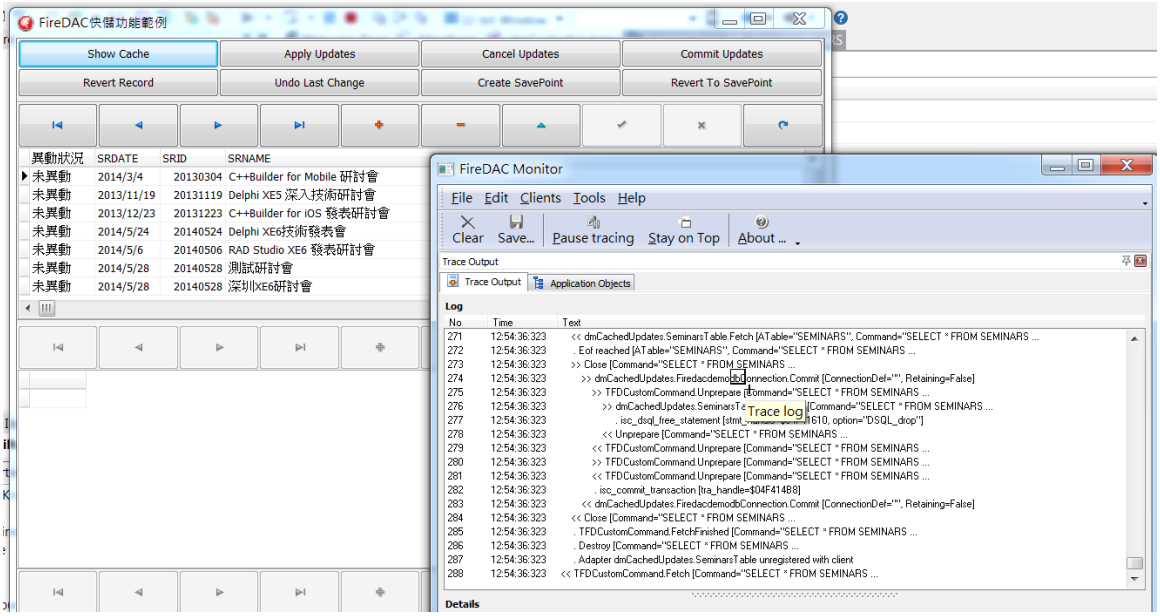

接著可點選 IDE 的 Tools|FireDAC Monitor 選單執行 FireDAC Monitor Explorer 工具：




再於資料模組中放入 TFDMoniRemoteClientLink 元件並且設定它的 Traced 特性值為 True：




現在再次執行快儲範例程式，就可以自到當 FireDAC 應用程式執行時所有 FireDAC 使用來處理資料的流程都會顯示在 FireDAC Monitor Explorer 中：

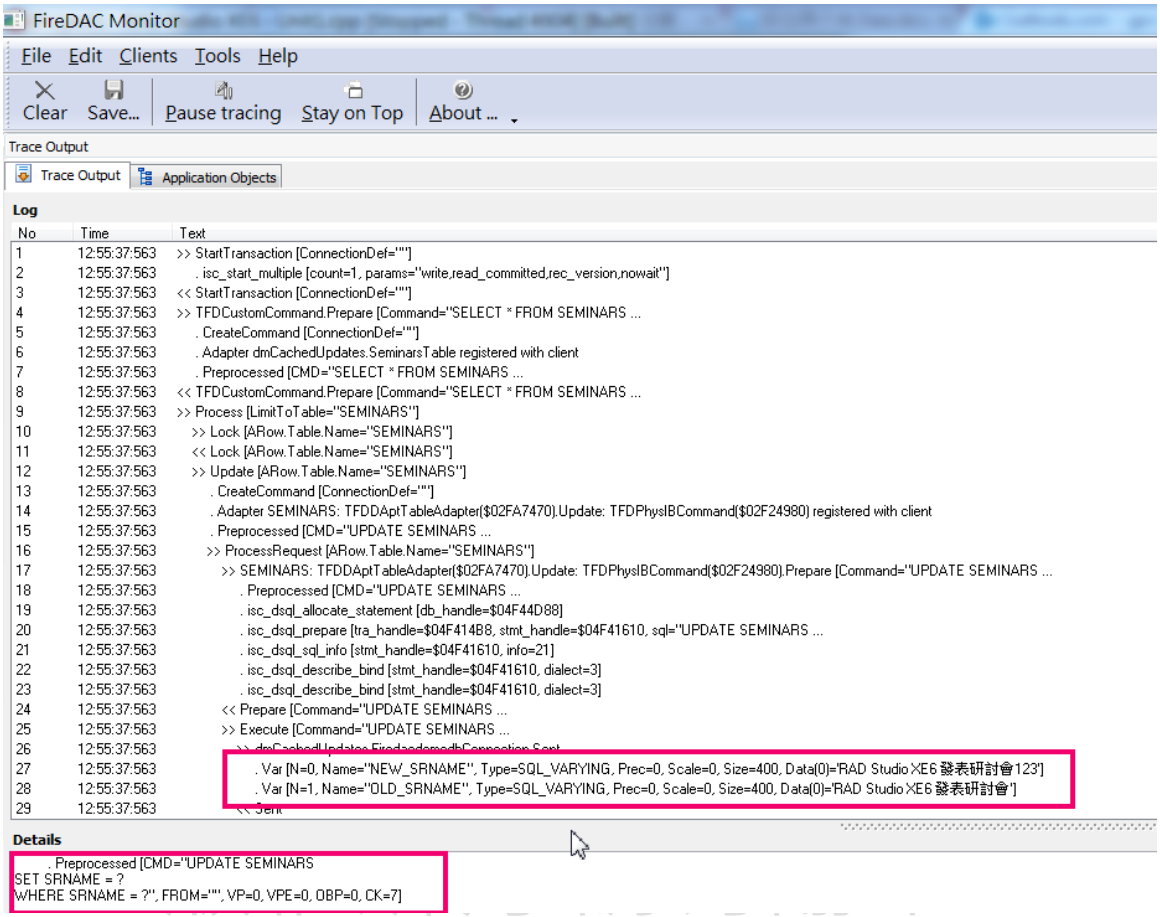



如果我們再次執行前面修改『RAD Studio XE6 發表研討會』為『RAD Studio XE6 發表研討會 123』的步驟，就可以在 FireDAC Monitor Explorer 中看到當我們點選主表單中的『Apply Updates』按鈕後 FireDAC 使用如下的 SQL 命令更新資料：

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



而上面的 2 個 SRNAM 就是『RAD Studio XE6 發表研討會』和『RAD Studio XE6 發表研討會 123』，從下面的 FireDAC Monitor Explorer 也可以看到：



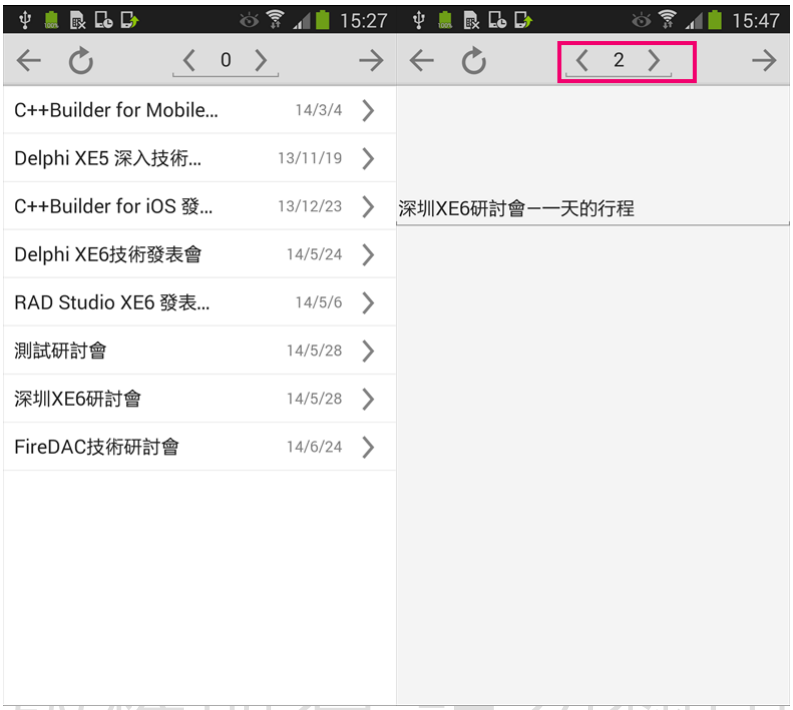

因此上面的 SQL 命令應該是：

```
"UPDATE SEMINARS
SET SRNAME = ” 『RAD Studio XE6 發表研討會 123』”
WHERE SRNAME = ” 『RAD Studio XE6 發表研討會』” ", FROM="", VP=0, VPE=0,
OBP=0, CK=
```

而此時由於後端的 SRNAME 欄位值已經是『RAD Studio XE6 發表研討會 321』了，因此找不到 SRNAME = ” 『RAD Studio XE6 發表研討會』” 的資料，因此發生錯誤 FireDAC 再觸發 OnReconcileError 事件，而且錯誤原因是 ekNoDataFound，也就是找不到資料的錯誤，這一切從 FireDAC Monitor Explorer 中也可以看的一清二楚。

# 2-5 在移動平台使用快儲功能

一般來說 FireDAC 的快儲功能適合使用在多客戶端的應用中，例如 C/S， 多層和 Web 系統中，但同樣的功能也可以使用在移動平台中。例如下圖是本書 pCachedUpdateApp 範例在 Android 手機中使用快儲功能執行的畫面：






當使用者在第 1 個頁面中點選一筆資料之後就可以在第 2 個頁面修改資料，上圖右方的 2 就代表現在在手機 App 中有 2 筆資料被異動了，最後點選上方最右邊的按鈕呼叫 ApplyUpdates 方法後就可以把資料都異動回手機的 InterBase 資料庫了。

雖然手機也可以使用 FireDAC 的快儲功能，但由於手機只有一個客戶端在使用，因此就不一定要使用 FireDAC 的快儲功能，除非您的手機 App 要連結中介伺服器形成 MEAP 的架構，那就另當別論了。

## 2-6 結論

本章的內容主要是說明如何使用 FireDAC 處理資料，在稍後的章節中會討論更多使用其他 FireDAC 元件處理資料的技巧。

# 第3章 使用記憶體資料元件

在 FireDAC 元件組中提供了 TFDMemTable，TFDMemTable 元件的功能是在記憶體中快速建立資料集物件的封裝元件，它類似 dbExpress 中的 TClientDataSet 元件，但 TFDMemTable 的執行速度比 TClientDataSet 快而且 TFDMemTable 也提供了許多 TClientDataSet 沒有的功能。在本章中將介紹如何使用 TFDMemTable 元件，因為 TFDMemTable 不單可使用在一般的資料庫應用程式中，也是最適合使用在移動平台和多層架構中的元件。



## 3-1 使用 TFDMemTable

TFDMemTable 元件  是 FireDAC 框架的記憶體資料集元件，也是 FireDAC 框架中處理資料最快速的元件。簡單的說 TFDMemTable 元件是把資料快儲在記憶體中進行處理，因此 TFDMemTable 元件中的資料基本上是和後端的資料來源是隔離的。

TFDMemTable 元件一般是使用在下面的場景中：

1. 把一些少量但經常會使用的資料放在 TFDMemTable 元件中，可提供最快速的資料處理速度，例如郵遞區號查詢，產品查詢等。
2. 使用 SOAP/REST 取得的資料放在 TFDMemTable 元件中可提供最佳的速度。
3. 使用 TFDQuery 元件取得的資料再拷貝到 TFDMemTable 元件中進行處理。

4. 頻繁異動的資料可暫時快儲在 TFDMemTable 元件中，等待所有異動完成之後再一次更新回後端。

## 3-1-1 使用 TFDMemTable 元件提供快速查詢

TFDMemTable 元件的 CreateDataSet()方法可以讓程式師在記憶體中建立任何架構的資料集物件然後在其中處理資料，因此可提供最快速的資料處理速度。現在假設我們需要在一個手機 App 中開發一個郵遞區號查詢的功能，那麼我們就可以使用 TFDMemTable 元件。

下面的程式碼是一個 Multi-Device Application 主表單的 OnCreate 事件處理函式它先呼叫 CreateZipCodeTable()方法 TFDMemTable 元件在記憶體中建立郵遞區號資料表，再呼叫 FillZipCodeData()方法顯示所建立的郵遞區號資料：

```
void __fastcall TfmMainForm::FormCreate(TObject *Sender)
{
  CreateZipCodeTable();
  FillZipCodeData();
}
```



CreateZipCodeTable()方法在 003 行先存取 FieldDefs 特性取得它的包含的欄位物件，然後在 005 行呼叫 AddFieldDef()方法在 TFDMemTable 元件中建立 TFieldDef 欄位物件，並且設定欄位物件的名稱和資料型態，於 014 行存取 IndexDefs 特性取得它的包含的索引物件，呼叫 AddIndexDef()方法在 TFDMemTable 元件中建立索引物件，最後在 018 行呼叫 CreateDataSet()方法完成在 TFDMemTable 元件中建立 2 個欄位和一個索引的工作。

```
001   void TfmMainForm::CreateZipCodeTable()
002   {
003     TFieldDefs *Defs = fdmtZipCodes->FieldDefs;
004
005     TFieldDef *aField = Defs->AddFieldDef();
006     aField->DataType = ftInteger;
007     aField->Name = L"郵遞區號";
008
009     aField = Defs->AddFieldDef();
010     aField->DataType = ftWideString;
011     aField->Size = 30;
```

```
012     aField->Name = L"區名";
013
014     TIndexDef *anIndex =
fdmtZipCodes->IndexDefs->AddIndexDef();
015     anIndex->Fields = L"郵遞區號";
016     anIndex->Name = "pnIndex";
017
018     fdmtZipCodes->CreateDataSet();
019   }
```

接著就可以在現在 fdmtZipCodes 中新增郵遞區號資料了 FillZipCodeData()方法就呼叫 Insert()和 Post()方法在 fdmtZipCodes 中新增資料：

```
void TfmMainForm::InsertZipData(const int iZipCode, const String
sName)
{
  fdmtZipCodes->Insert();
  fdmtZipCodes->FieldByName(L"郵遞區號")->Value = iZipCode;
  fdmtZipCodes->FieldByName(L"區名")->Value = sName;
  fdmtZipCodes->Post();
}
//---------------------------------------------------------------
void TfmMainForm::FillZipCodeData()
{
  InsertZipData(100, L"中正區");
  InsertZipData(103, L"大同區");
  InsertZipData(104, L"中山區");
  InsertZipData(105, L"松山區");
  InsertZipData(106, L"大安區");
  InsertZipData(108, L"萬華區");
  ..
}
```



一但加入了資料之後就可以呼叫 DisplayZipCodes()方法顯示郵遞區號資料了：

```
void TfmMainForm::DisplayZipCodes()
{
```

```
  lvZipCodes->Items->Clear();
  fdmtZipCodes->First();
  while (! fdmtZipCodes->Eof)
  {
  TListViewItem *alv = lvZipCodes->Items->Add();
  alv->Detail = fdmtZipCodes->FieldByName(L"郵遞區號")->Value;
  alv->Text = fdmtZipCodes->FieldByName(L"區名")->Value;
  fdmtZipCodes->Next();
  }
  fdmtZipCodes->First();
}
```

如果現在執行此範例程式就可以在手機中看到如下的畫面：

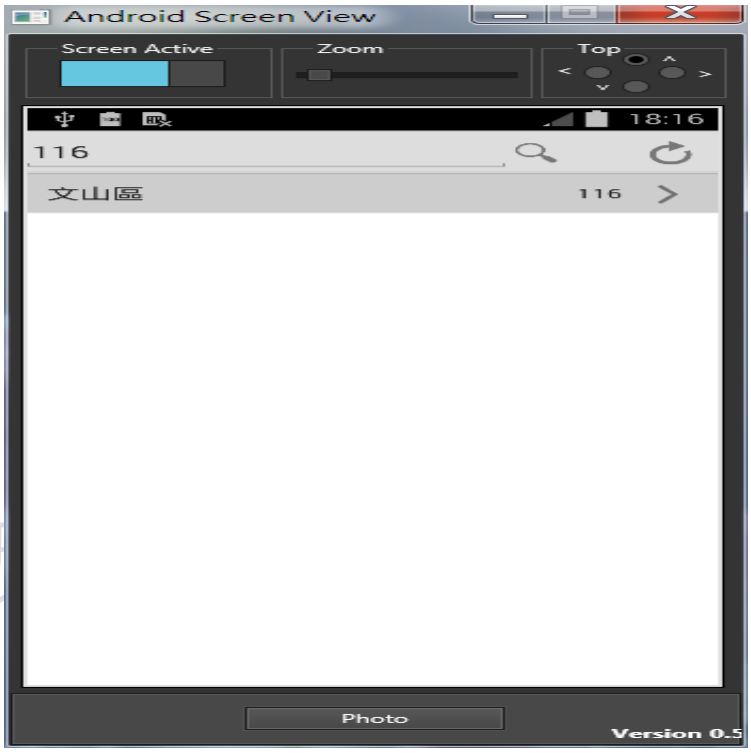



最後加入查詢資料的功能，藉由使用過濾器功能查詢：

```
void TfmMainForm::QueryZipCode(const String sZipCode)
{
  String sFilter = L"\"郵遞區號\"=" + edtZipCode->Text;
  fdmtZipCodes->Filtered = false;
```

```
  fdmtZipCodes->Filter = sFilter;
  fdmtZipCodes->Filtered = true;
  DisplayZipCodes();
}
```

如果現在執行此範例程式就可以在手機中看到可以正確的查詢郵遞區號了：

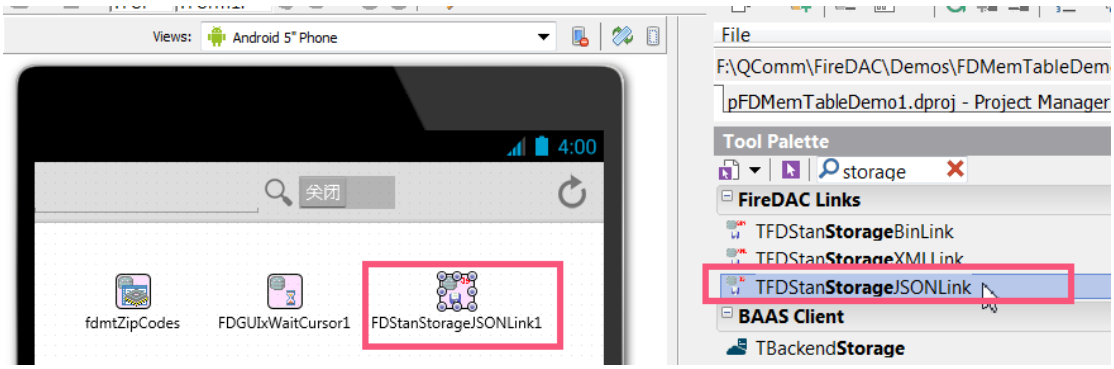



TFDMemTable 元件也可以把它包含的記憶體資料以不同的格式儲存到檔案中，或是再從檔案中再讀取回來，TFDMemTable 的 SaveToFile()和 LoadFromFile()方法就可以完成這 2 個工作。SaveToFile()和 LoadFromFile()方法的第 2 個參數可指定以什麼格式儲存/讀回檔案內容，目前 FireDAC 支援的格式有：

| 格式 | 說明 |
|---|---|
| sfXML | XML 格式 |
| sfBinary | 二進位 |
| sfJSON | JSON 格式 |
| sfAuto | 自動判斷格式 |

現在讓我們繼續為上面的範例加入儲存到檔案的功能，假設璋在我們希望這個 App 在啟動時建立了郵遞區號之後就把這個資料表儲存下來，在下次 App 啟動時就直接從儲存的檔案中讀取資料，而且我們要以 JSON 的格式儲存郵遞區號資料。

首先請在主表格中加入 TFDStanStorageJSONLink 元件讓這個 App 支援以以 JSON 的格式儲存和讀取資料，如下所示：




修改主表單的 OnCreate()事件處理函式，如果發現已經有先前儲存的郵遞區號檔案就呼叫 LoadZipCodeFile()方法從郵遞區號檔案讀回資料，如果沒有的話就像先前一樣建立郵遞區號資料表最後加入呼叫 SaveZipCodeFile()方法儲存遞區號資料到郵遞區號檔案：

```
void __fastcall TfmMainForm::FormCreate(TObject *Sender)
{
  if (!FindZipCodeFile())
  {
  CreateZipCodeTable();
  FillZipCodeData();
  SaveZipCodeFile();
  }
  else
  {
  LoadZipCodeFile();
  swLoadFromFile->IsChecked = true;
  }
}
```



LoadZipCodeFile()方法呼叫 LoadFromFile()方法從 App 的文件目錄中讀取郵遞區號檔案，傳入 LoadFromFile()方法的第 2 個參數使用了 sfJSON 代表是讀取以 JSON 格式儲存的資料：

```
String TfmMainForm::GetZipCodeFile()
{
```

```
  return
IncludeTrailingPathDelimiter( System::Ioutils::TPath::GetDocumen
tsPath()) + sZIPCODEFILE;
}
//---------------------------------------------------------------
void TfmMainForm::LoadZipCodeFile()
{
  fdmtZipCodes->LoadFromFile(GetZipCodeFile(), sfJSON);
}
```

SaveZipCodeFile()則是呼叫 SaveToFile()方法並且以 JSON 格式儲存資料：

```
void TfmMainForm::SaveZipCodeFile()
{
  fdmtZipCodes->SaveToFile(GetZipCodeFile(), sfJSON);
}
```

現在如果執行這個範例 App 就可以發現第 2 次執行 App 時上方的 TSwitch 元件是在打開的狀態，代表資料是從郵遞區號檔案中讀回的了：



如果我們觀察郵遞區號檔案的實際內容也可以看到如下的結果，可以確定郵遞區號資料的確是以 JSON 的格式儲存和讀回的：


```
{"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":"郵遞區號
","SourceName":"郵遞區號
","SourceID":1,"DataType":"Int32","Searchable":true,"AllowNull":
true,"Base":true,"OAllowNull":true,"OInUpdate":true,"OInWhere":t
rue,"OriginColName":"郵遞區號"},{"class":"Column","Name":"區名
","SourceName":"區名
","SourceID":2,"DataType":"WideString","Size":30,"Searchable":tr
ue,"AllowNull":true,"Base":true,"OAllowNull":true,"OInUpdate":tr
ue,"OInWhere":true,"OriginColName":"區名
","SourceSize":30}],"ConstraintList":[],"ViewList":[],"RowList":
[{"RowID":0,"RowState":"Unchanged","Original":{"郵遞區號":100,"區
名":"中正區"}},{"RowID":1,"RowState":"Unchanged","Original":{"郵遞
區號":103,"區名":"大同區"}},
   … ,{"RowID":22,"RowState":"Unchanged","Original":{"郵遞區號
":251,"區名":"淡水區
"}}]}],"RelationList":[],"UpdatesJournal":{"SavePoint":23,"Chang
es":[]}}}}
```



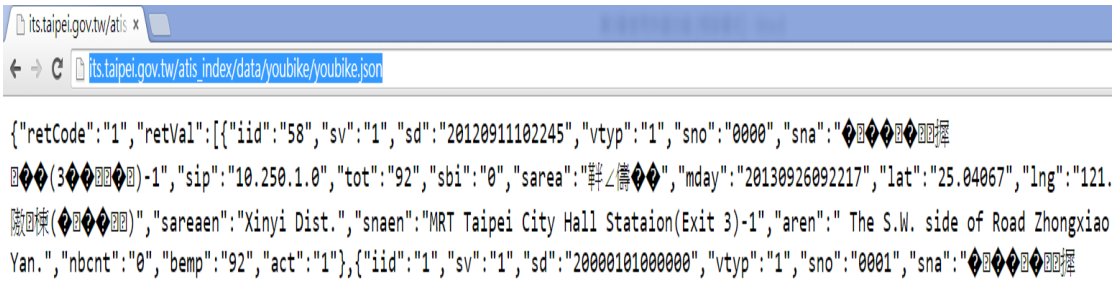

## 3-1-2 使用 TFDMemTable 處理 SOAP/REST 取得的資料

現在的應用環境中資料來源愈來愈多樣化，早已跳脫資料大多只從資料庫來的應用，特別是在移動平台中資料可能是從網站或是遠端服務而來。這些新的資料來源格式可能是文字，ＸＭＬ或是現在最普遍的 JSON 格式。對於這些應用 TFDMemTable 元件也非常的適合，因為我們可以把這些資料來源的資料轉到 TFDMemTable 中之後就可以搭配資料感知元件或是 LiveBidning 技術來使用。

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

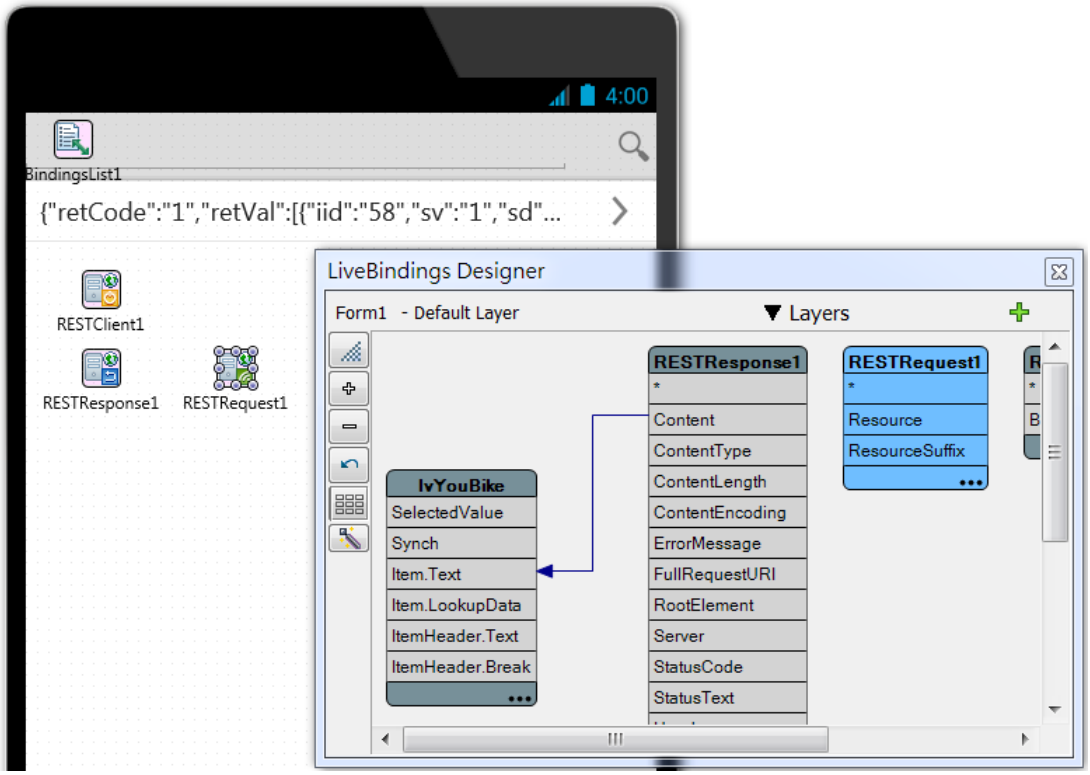
```
http://its.taipei.gov.tw/atis_index/data/youbike/youbike.json
```

如果使用瀏覽器瀏覽上面的 URL 可以看到這些資料的確是以 JSON 的格式提供的：

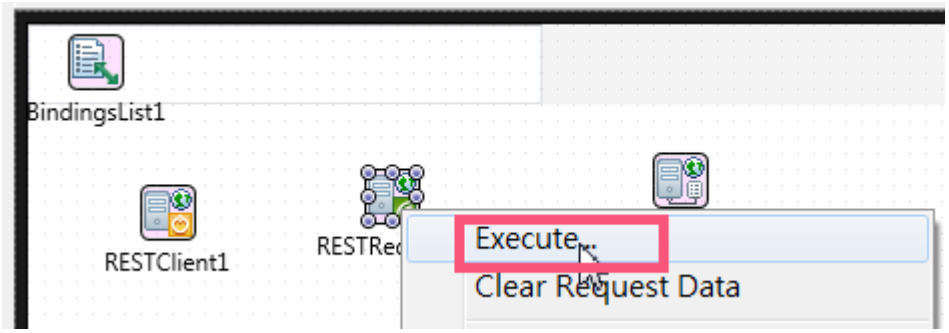



現在讓我們使用 TFDMemTable 元件來處理上述的 JSON 格式資料，讓我們可以在上面的資料中進行搜尋的工作。

建立一個 Multi-Device Application，在其中加入 TRESTClient，TRESTRequest，TRESTResponse 元件和 ToolBar，TEdit，TButton 和 TListView 元件下所示：

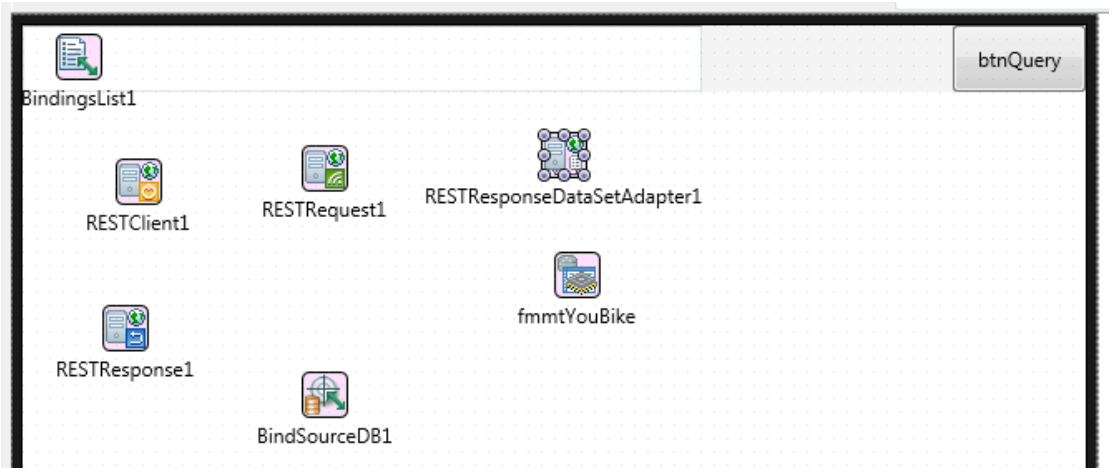



把上面的 URL 設定到 TRESTClient 的 BaseURL 特性中，點選 TRESTRequest 再點選滑鼠右鍵選擇其中的 Execute...選項提出請求欲取得 YouBike 資料：

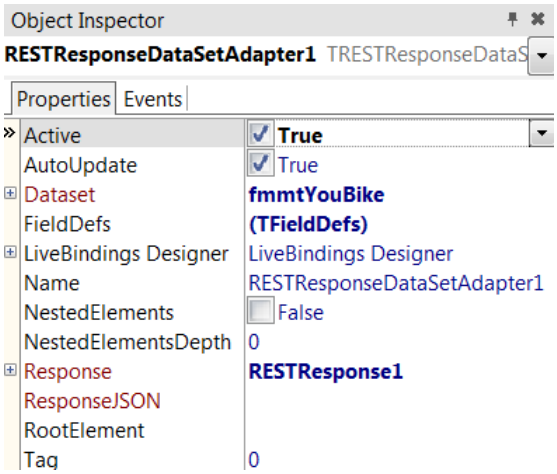



再使用 LiveBinding 技術連結 YouBike 資料和 TListView 元件。

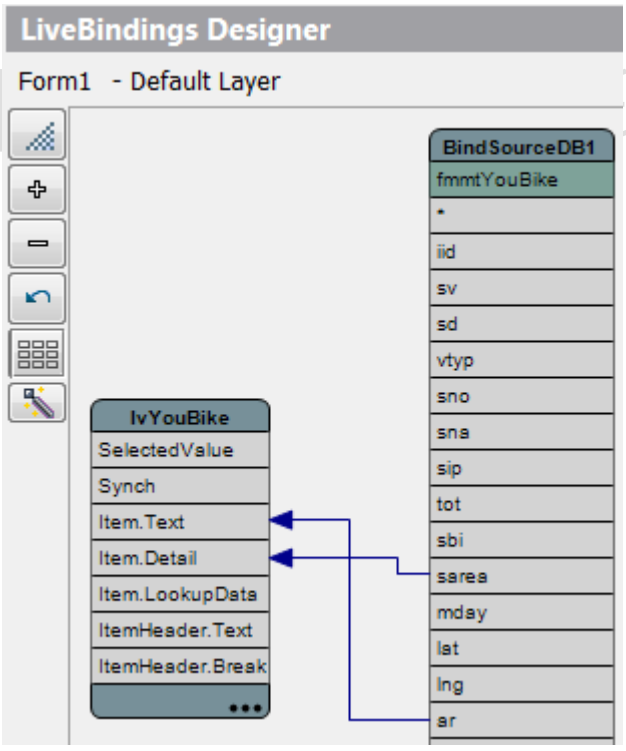
現在就可以準備把 JSON 的 YouBike 資料轉到 TFDMemTable 中，請在主表單中加入 TRESTResponseDataSetAdapter 和 TFDMemTable 元件：




設定 TRESTResponseDataSetAdapter 元件的 Response 特性值為主表單中的 TRESTResponse 元件，設定 DataSet 特性值為剛加入的 TFDMemTable 元件。再把 TRESTResponse 元件的 RootElement 特性值設定為 retVal，最後再設定 TRESTResponseDataSetAdapter 元件的 Active 特性值為 True：




使用 LiveBidining 重新連結 ListView 元件和剛加入的 TFDMemTable 元件：






元在主表單就可以看到如下的畫面 YouBike 的 JSON 格式資料就轉到 TFDMemTable 元件中了：




最後再主表單的 OnActivate 事件中呼叫 TRESTRequest 元件的 Execute()方法在 App 執行時取得 YouBike 資料：

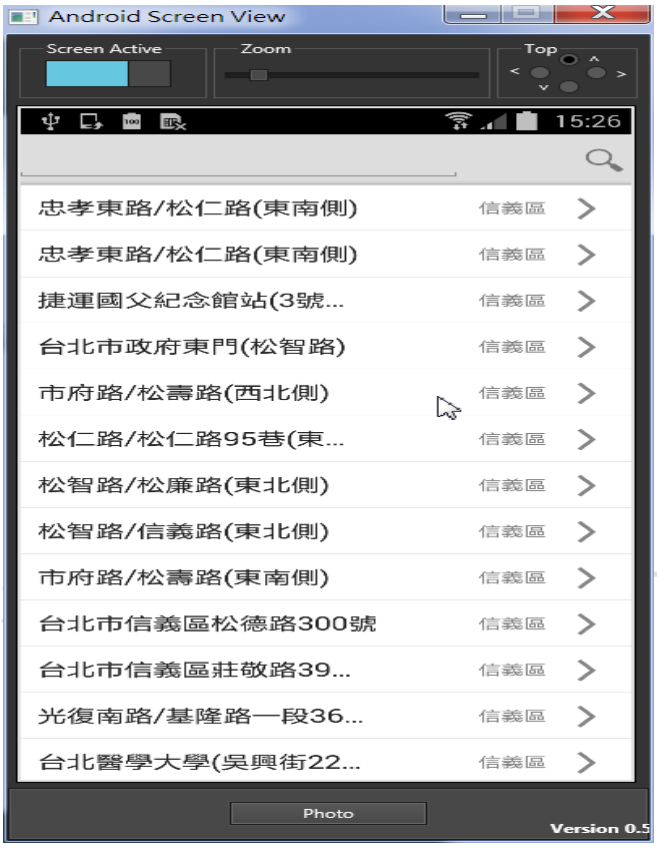
```
void __fastcall TfmMainForm::FormActivate(TObject *Sender)
{
  RESTRequest1->Execute();
}
```

再實作查詢 TButton 的 OnClick 事件如下，使用 TFDMemTable 元件的過濾器功能查詢資料：

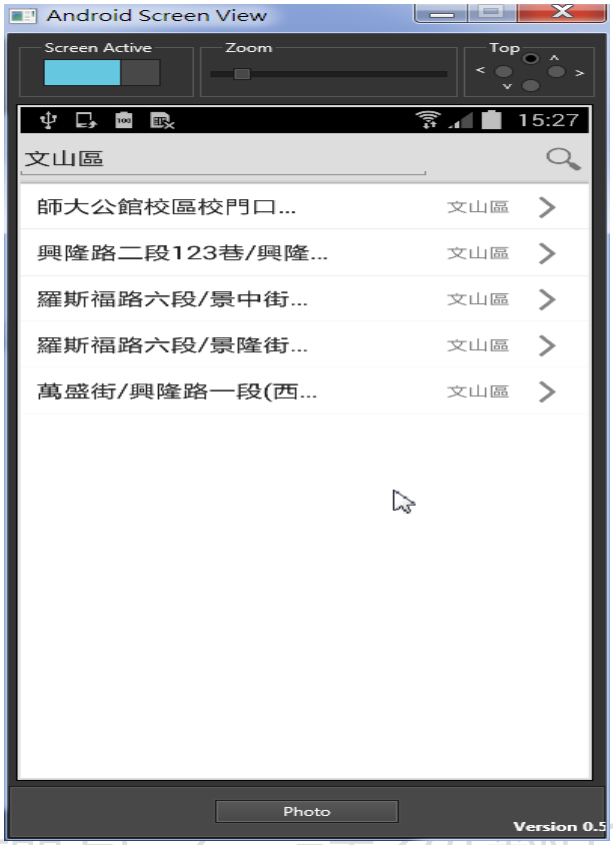
```
void __fastcall TfmMainForm::btnQueryClick(TObject *Sender)
{
  String sFilter = "\"sarea\"='%s'";
  fmmtYouBike->Filtered = false;

  fmmtYouBike->Filter = Format(sFilter,
ARRAYOFCONST((edtArea->Text)));
```

```
  fmmtYouBike->Filtered = true;
  LinkFillControlToField1->BindList->FillList();
}
```

執行範例 App 就可以在手機中看到如下的畫面，顯示了 YouBike 的資料：




也可以在主表單的 TEdit 元件輸入資料查詢每一行政區的 YouBike 資料了：






## 3-1-3 使用 TFDMemTable 處理資料

TFDMemTable 另外一個經常使用的場景就是使用它來共享一個 TDataSet 元件中的資料然後再於 TFDMemTable 中處理資料，這樣做的原因可能有很多，但最常應用程狀況是因為在許多應用中我們不希望影響到原本 TDataSet 連結的資料應知元件，因此在另一個 TFDMemTable 中處理資料即可避免。

另外一個原因則是因為 TFDMemTable 處理資料的速度很快，因此可以在另外的執行緒中使用 TFDMemTable 而不是原本的 TDataSet。

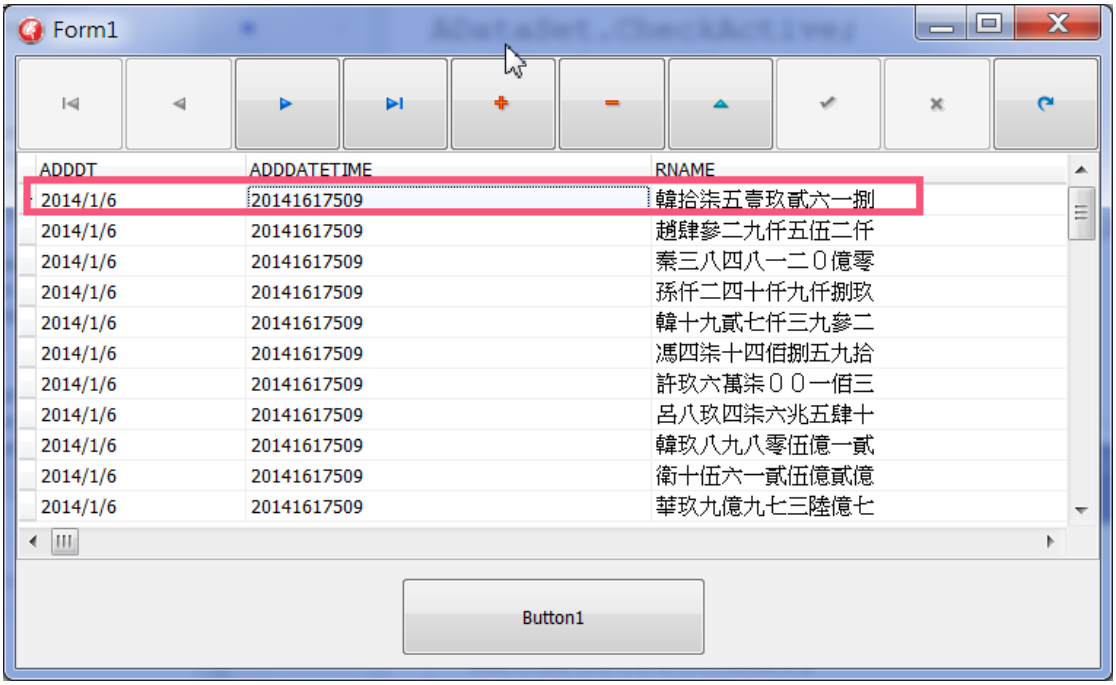
要讓 TFDMemTable 共享 TDataSet 元件中的資料，程式師可以呼叫 CloneCursor 方法，下面是它的宣告原型：

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

CloneCursor 方法接受 3 個參數，第 1 個參數是來源 TDataSet 元件，第 2 個參數 AReset 如果設定為 true 的話代表要設定 TFDMemTable 元件所有的特性值都將設定為內定值，由於通常我們是希望分享來源 TDataSet 元件的設定，因此一般情形下都是設

定 AReset 為 False。第 3 個參數 AKeepSettings 設定為 true 的話代表 TFDMemTable 的任何先前設定都不改變，因此一般情形下也都是設定 AKeepSettings 為 false。

讓我們看個使用 CloneCursor 的範例，下面是一個使用 TFDQuery 取得資料的畫面，假設現在我們要改變其中 ADDDT 欄位中的日期，但又不希望影響到目前的 TDBGrid 顯示資料的狀態：




那我們就可以使用下面的程式碼，先呼叫 TFDMemTable 的 CloneCursor()方法讓 TFDMemTable 可以分享 TbltestdataTable 中的資料，

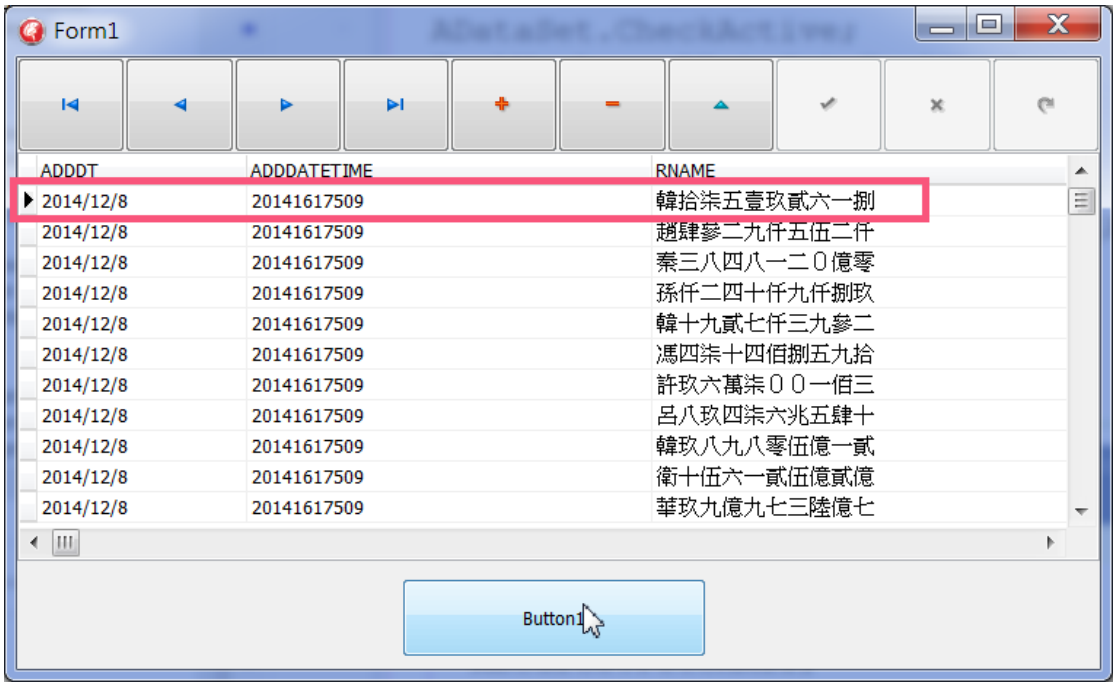
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  FDMemTable1->CloneCursor(TbltestdataTable, false, false);
  Application->OnIdle = UpdateToToday;
}
```

然後再使用 TFDMemTable 修改 ADDDT 欄位中的日期資料：

```
void __fastcall TfmMainForm::UpdateToToday(TObject *Sender, bool
&Done)
{
  FDMemTable1->First();
```

```
  while (! FDMemTable1->Eof)
  {
  FDMemTable1->Edit();
  FDMemTable1->FieldByName("ADDDT")->Value = Now();
  FDMemTable1->Post();
  FDMemTable1->Next();
  }
}
```

從下面的結果可以看到，在點選了主表單中的按鈕之後 TFDMemTable 就立刻修改了 ADDDT 欄位中的日期資料，但 TDBGrid 完全沒有受到影響：




TFDMemTable 處理資料另外一個經常使用的場景就是開發多層的系統，在多層架構中通常是由客戶端向中介的伺服器請求服務，如果客戶端向中介的伺服器請求資料的話，中介伺服器可以回傳 TDataSet 物件回客戶端。一旦客戶端取得了回傳的 TDataSet 物件我們就可以使用 TFDMemTable 來接受並處理回傳的資料。

例如下面就是一個中介伺服器的方法 Divisions，它回傳 TDataSet 物件：

```
public
  { Public declarations }
  ...
```

```
    function Divisions : TDataSet;
```

...

下面是中介伺服器使用 TFDQuery 元件從資料表中取出資料接著直接回傳 TFDQuery 回客戶端：

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

TFDMemTable 的 CopyDataSet()方法可以從一個 TDataSet 中拷貝資料，下面是 CopyDataSet()方法的宣告原型：

```
void __fastcall CopyDataSet(Data::Db::TDataSet* ASource,
TFDCopyDataSetOptions AOptions = (TFDCopyDataSetOptions() <<
Firedac_Comp_Dataset__3::coRestart <<
Firedac_Comp_Dataset__3::coAppend ));
```

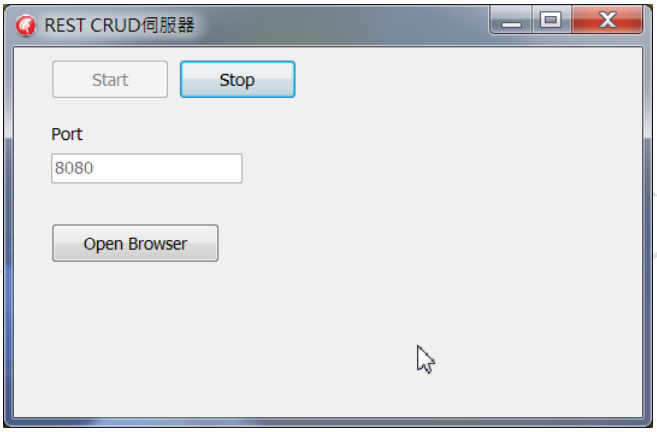


CopyDataSet()方法的接受 2 個參數，第 1 個參數是來源 TDataSet 物件，第 2 個參數 TFDCopyDataSetOptions 則代表要如何從第 1 個參數的 TDataSet 物件中拷貝資料。因此要拷貝一個回傳到客戶端的 TDataSet 物件，我們需要拷貝它的結構和資料。在下面的客戶端程式碼中呼叫了 TFDMemTable 的 CopyDataSet()方法，其中傳入的第 2 個參數值[coStructure, coRestart, coAppend]代表要拷貝來源 TDataSet 的結構，再拷貝資料到 TFDMemTable 中並且取後把目前資料位置移到第 1 筆：

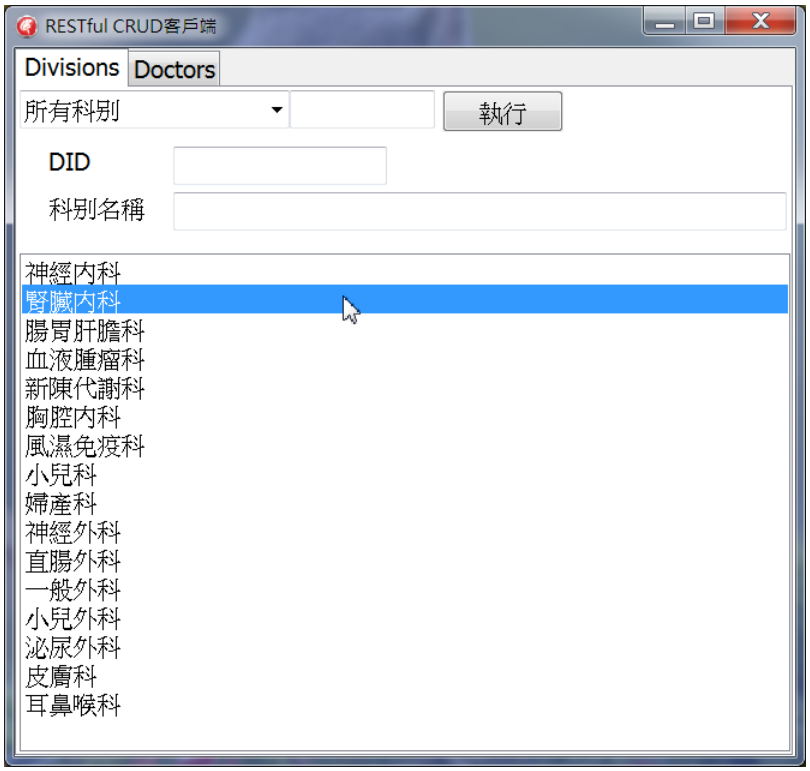
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  TServerMethods1Client* aServer = new
TServerMethods1Client(DSRestConnection1);
  try
  {
  TDataSet *aDS = aServer->Divisions();
  TFDCopyDataSetOptions AOptions;
  AOptions.Clear();
```

```
  AOptions << coStructure << coRestart << coAppend;
  fdmtDivisions->CopyDataSet(aDS, AOptions);
  DisplayDivisions();
  }
  __finally
  {
  delete aServer;
  }
}
```

執行了上面的程式碼之後中介伺服器回傳的資料就存在於 TFDMemTable 元件中了，接著我們就可以使用 TFDMemTable 處理這些資料。例如下面的個畫面就是中介伺服器：




下面則是客戶端 Windows 程式，它從上面的中介伺服器取得資料並拷貝到 TFDMemTable 元件中後就可以進行顯示和處理的工作了：




## 3-2 結論



不論是桌上型，C/S，多層或是移動設備的應用程式，能夠盡量在記憶體中有效的處理資料是開發資料庫相關的應用程式最有效率的方法，本章討論的 TFDMemTable 是 FireDAC 框架中處理資料最快速的方法，在讀者掌握了這些技巧之後應該就可以開發出非常有效率的 FireDAC 資料庫程式系統了。

# 第4章 FireDAC進階功能

FireDAC 提供了豐富的功能讓程式師處理資料，這些資料包含了資料庫中的使用者資料以及資料庫的 MetaData。此外 FireDAC 也提了非常具有彈性的方式讓程式師存取資料，例如 Macro，動態 SQL 等。

本章的內容將說明如何使用這些 FireDAC 的功能，讓我們從 MetaData 開始吧。

## 4-1 存取 MetaData



FireDAC 提了數種方式讓程式師可存取資料庫的 MetaData，例如資料庫的 Catalog，Schema，Package 等資訊，或是資料庫中的資料表，欄位等資訊。在 FireDAC 中程式師可以使用 TFDConnection 元件或是 TFDMetaInfoQuery 元件。

### 4-1-1 使用 TFDConnection 元件存取 MetaData

TFDConnection 元件提供了數個方法可讓程式師存取資料庫中的 MetaData，下面的表格說明了這些相關的函式：

| TFDConnection方法 | 說明 |
|---|---|
| GetCatalogNames | 取得資料庫 Catalog MetaData |
| GetSchemaNames | 取得資料庫 Schema MetaData |
| GetTableNames | 取得資料庫中所有資料表或 View 列表 |
| GetFieldNames | 取得資料庫中資料表的欄位列表 |
| GetKeyFieldNames | 取得資料庫中資料表的鍵值欄位列表 |
| GetGeneratorNames | 取得資料庫中 Generato/Sequence 列表 |
| GetPackageNames | 取得資料庫中 Package 列表 |

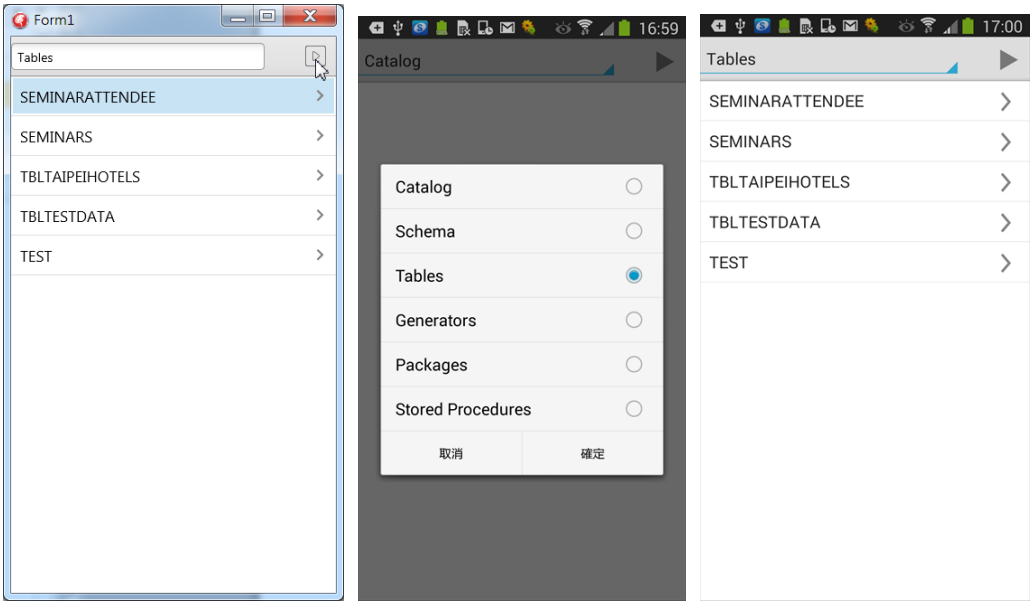
| GetStoredProcNames | 取得資料庫中預儲程序列表 |
|---|---|

例如要取得資料庫中所有的資料表，那就可以呼叫 TFDConnection 元件的 GetTableNames()方法：

```
void TfmMainForm::GetTableMetaData()
{
  TStringList *aSL = new TStringList();
  try
  {
    dmBCBDemoDB->SqlitedemodbConnection->GetTableNames("", "", "",
aSL);
    DisplayMetaData(aSL);
  }
  __finally
  {
    delete aSL;
  }
}
```

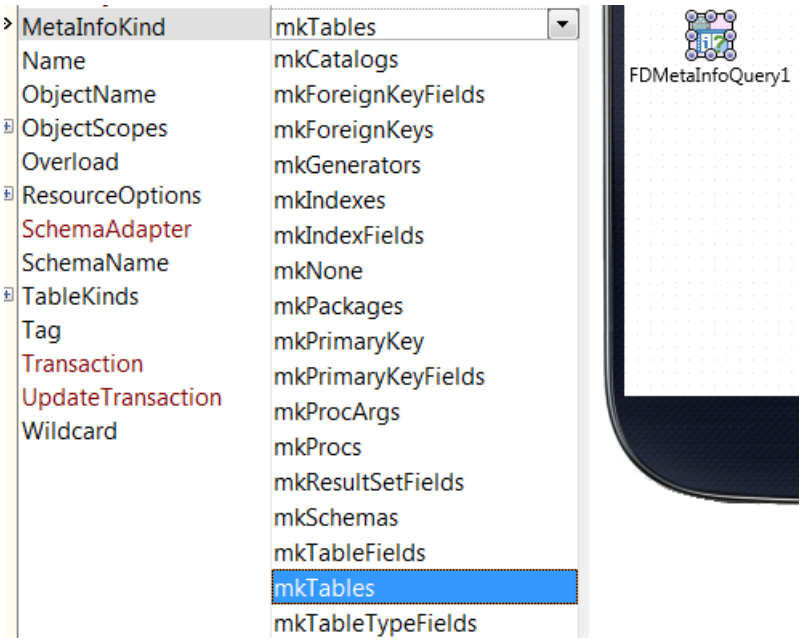


下圖就是此程式碼執行的時取得範例資料庫中所有範例資料表的結果：



## 4-1-2 使用 TFDMetaInfoQuery 元件存取 MetaData

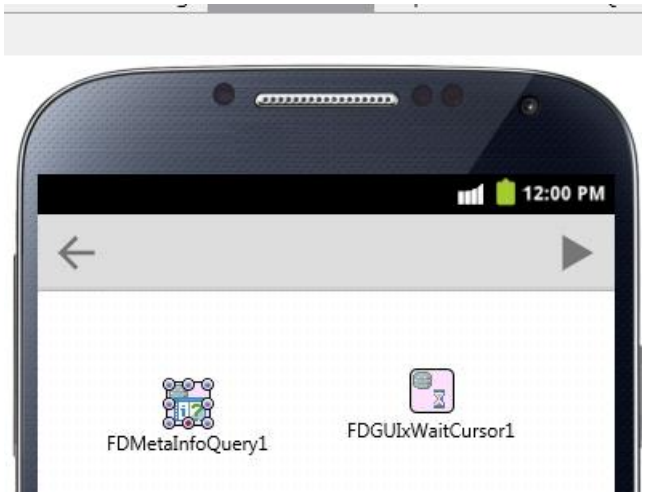
第 2 種方式就是使用 TFDMetaInfoQuery 元件，使用 TFDMetaInfoQuery 元件可以更方便的取得 MetaData，因為程式師只要設定它的 MetaInfoKind 特性即可取得相對應的 MetaData，如下所示：




當程式師使用 TFDMetaInfoQuery 元件取得 MetaData 之後會以資料集的方式儲在在 TFDMetaInfoQuery 元件中，而每一種 MetaData 都定了不同的架構，程式師可以參考下面連結中說明的各種 MetaData 資料結構：

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

例如下面的範例 App 使用了 TFDMetaInfoQuery 元件來查詢範例資料庫中的所有資料表，如果使用者點選了一個查詢到的資料表就可以再查詢其中的所有欄位：




要查詢資料表，我們可以設定 TFDMetaInfoQuery 的 MetaInfoKind 特性值為 mkTables 再開啟 TFDMetaInfoQuery 元件即可：

```
void __fastcall TfmMainForm::SpeedButton1Click(TObject *Sender)
{
  FDMetaInfoQuery1->MetaInfoKind = mkTables;
  FDMetaInfoQuery1->Open();
  DisplayTables();
}
```

DisplayTables() 方法則根據 http://docwiki.embarcadero.com/RADStudio/XE6/en/Metadata_Structure_(FireDAC)中定義的資料表 MetaData 資料結構從其中取出資料表的名稱和型態：

```
void TfmMainForm::DisplayTables()
{
  lvTables->Items->Clear();
  while (! FDMetaInfoQuery1->Eof)
  {
    TListViewItem *alvi = lvTables->Items->Add();
    alvi->Text =
FDMetaInfoQuery1->FieldByName("TABLE_NAME")->AsString;
    TFDPhysTableKind iTK =
TFDPhysTableKind(FDMetaInfoQuery1->FieldByName("TABLE_TYPE")->AsInteg
er);
```

```
    if (iTK == TFDPhysTableKind::tkSynonym)
      alvi->Detail = "Synonym";
    else
      if (iTK == TFDPhysTableKind::tkTable)
        alvi->Detail = "Table";
      else
        if (iTK == TFDPhysTableKind::tkView)
          alvi->Detail = "View";
       else
          if (iTK == TFDPhysTableKind::tkTempTable)
            alvi->Detail = "TempTable";
        else
            if (iTK == TFDPhysTableKind::tkLocalTable)
              alvi->Detail = "LocalTable";
    FDMetaInfoQuery1->Next();
  }
```

當使用者點選了某一資料表就呼叫 DisplayTableFields()方法顯示資料表欄位：

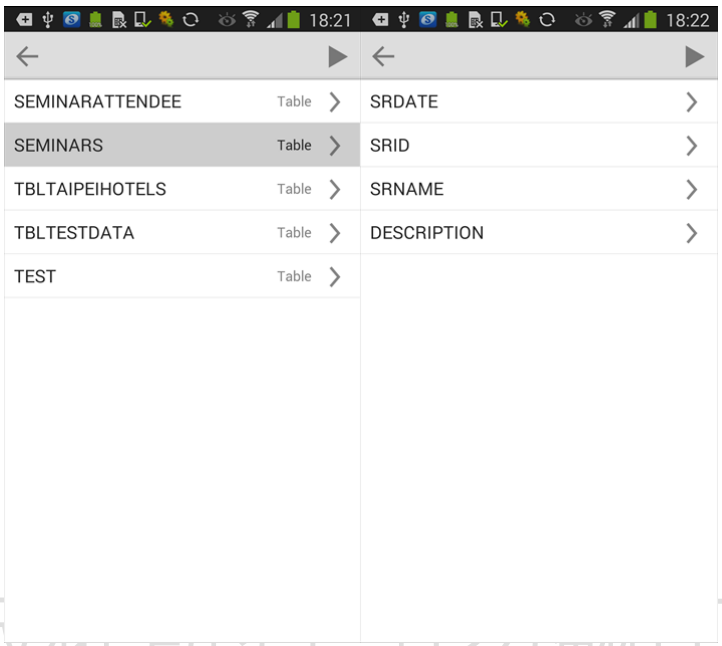
```
void __fastcall TfmMainForm::lvTablesItemClick(TObject * const Sender,
TListViewItem * const AItem)
{
  DisplayTableFields(AItem->Text);
}
```

而 DisplayTableFields 方法也是使用 TFDMetaInfoQuery 元件，設定它的 MetaInfoKind 特性值為 mkTableFields 並且設定 ObjectName 特性值為使用點選的資料表名稱再開啟 TFDMetaInfoQuery 元件即可：

```
void TfmMainForm::DisplayTableFields(const String sTableName)
{
  TLocateOptions AOptions;
  AOptions.Clear();
  if (FDMetaInfoQuery1->Locate("TABLE_NAME", sTableName, AOptions))
  {
    FDMetaInfoQuery1->MetaInfoKind = mkTableFields;
    FDMetaInfoQuery1->ObjectName = sTableName;
    FDMetaInfoQuery1->Open();
    DisplayFields();
```

```
    TabControl1->TabIndex = 1;
  }
}
```

最後執行此範例 App 就可以看到類似如下的結果：






能夠查詢和處理 MetaData 有什麼用呢？再說明之前先讓我們再討論另一個 FireDAC 的功能，那就是 Macro。

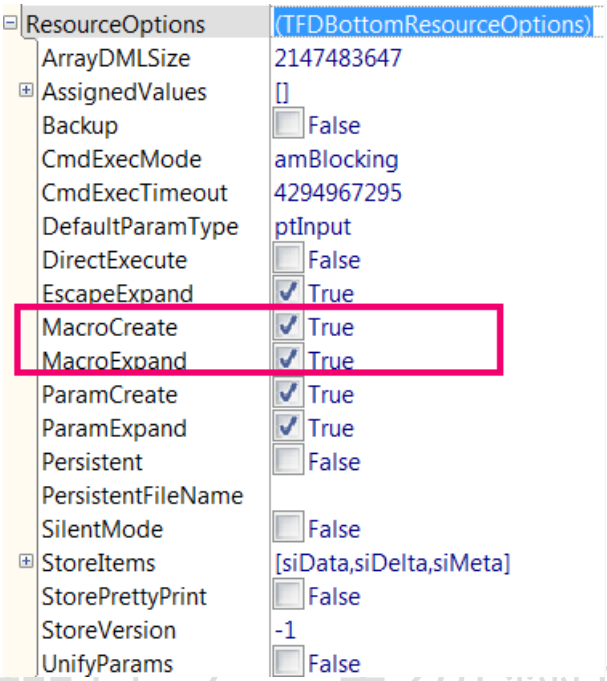
# 4-2 巨集功能(Marco)

FireDAC 提供了豐富的 Macro 功能讓開發人員使用，簡單的說所謂的巨集就是 FireDAC 在把 SQL 命令送到後端執行時會根據巨集功能定義先把 SQL 中使用的巨集替換成相對應的字串之後再把替換的 SQL 命令送出。

FireDAC 基本上提供了 3 種巨集，它們是：

| 巨集功能 | 說明 |
|---|---|
| 替換變數 | 可替換 SQL 命令中的欄位和資料表名稱 |
| Escape sequence | 可在 SQL 命令中撰寫和後端資料庫獨立的命令，FireDAC 會自動根據後端資料庫替換成後端資料庫專屬的命令。Escape sequence 是使用{}包圍的指令 |
| 條件替換 | 可在 SQL 命令中使用判斷條件 |

要使用 FireDAC 的巨集功能，要先把 TFDQuery 元件的 ResourceOptions.MacroCreate 和 ResourceOptions.MacroExpand 特性值設為 True：





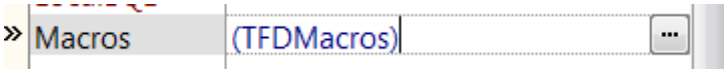

FireDAC 使用'!'或"符號代表巨集，

| 巨集符號 | 說明 |
|---|---|
| ! | 代表"字串"替換模式，直接替換 |
| & | 代表"SQL 命令"替換模式，FireDAC 會根據資料型態使用 RDMBS 的語法替換 |

例如如果我們在 TFDQuery 中撰寫了下面的 SQL 命令：

```
SELECT * FROM &TableName
```

由於在此 SQL 命令中使用巨集來代表要存取的資料表名稱，因此在 TFDQuery 的 Macros 特性中：



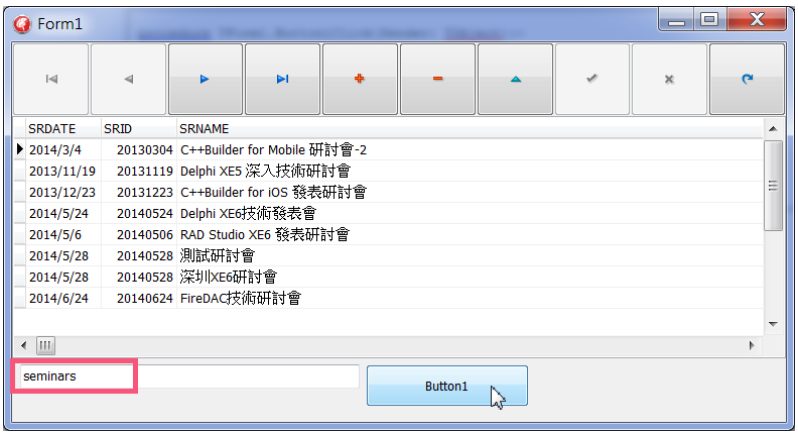

會看到 FireDAC 會解析此 SQL 命令並且發現一個巨集，因此就會在 Macros 特性中有一個名為"TABLENAME"的巨集：


接著我們就可以在程式中使用下面的程式碼在程式執行時動態輸入資料表名稱來存取資料：

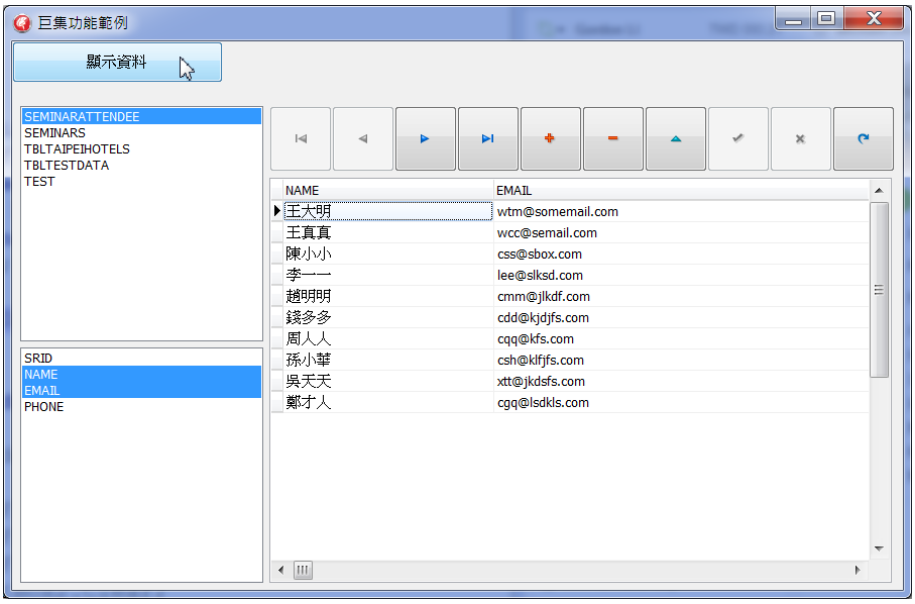
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  FDQuery1->Active = false;
  FDQuery1->MacroByName("TableName")->AsRaw = Edit1->Text;
  FDQuery1->Active = true;
}
```



例如下面就是範例程式執行的結果，我們動態輸入"seminars"資料表名稱後的確可以存取到其中的資料：

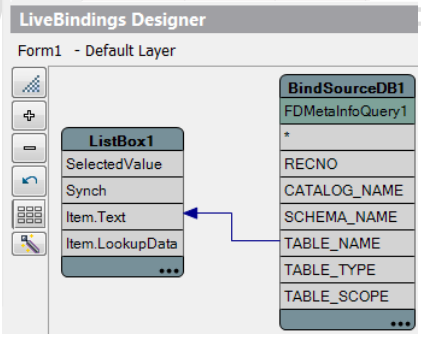



除了可使用巨集替換資料表名稱外也可以替換 SQL 命令其他的內容，現在讓我們看另外一個範例。下面是一個範例程式，在執行時此程式使用 TFDMetaInfoQuery 取得連結資料庫中所有的資料表名稱，使用者點選其中的資料表時左下方的 ListBox 就會顯示此資料表中的欄位。使用者可於其中選擇欄位，最後再點選左上方的"顯示資料"按鈕就可以顯示任何資料表中任何欄位的資料：




此範例程式使用了 2 個巨集 SQL 命令，一個巨集 SQL 命令動態從使用者選擇的資料表中擷取欄位資訊，一個巨集 SQL 命令根據使用者選擇的欄位擷取資料。一開始範例程式使用 TFDMetaInfoQuery 元件取得連結資料庫中所有的資料表名稱，再藉由 LiveBinding 顯示在左上方的 ListBox 中：






當使用者在在左上方的 ListBox 中隨意選擇一個資料表之後，程式就使用下面的巨集 SQL 命令從資料表中取得所有的欄位資訊，由於我們不希望取得任何資料而是只取得欄位資訊，因此在下面的巨集 SQL 命令的 **where** 部份使用了永不成立的條件以避免取得任何資料：

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

因此在左上方的 TListBox 的 OnClick 事件呼叫 LoadFields()方法藉由剛才說明的巨集 SQL 命令從資料表中取得所有的欄位資訊：

```
void __fastcall TfmMainForm::ListBox1Click(TObject *Sender)
```

```
{
  LoadFields();
}
```

LoadFields()方法把使用者在左方上選擇的資料表名稱代入巨集值中，開啟 TFDQuery 元件，取得欄位資訊再顯示在左下方的 ListBox2 之中：

```
void TfmMainForm::LoadFields()
{
  qryFields->Close();
  qryFields->Macros->operator [](0)->AsRaw =
ListBox1->Items->operator [](ListBox1->ItemIndex);
  qryFields->Open();
  ListBox2->Clear();
  for (int iIndex = 0; iIndex < qryFields->FieldCount; iIndex++)
  ListBox2->Items->Add(qryFields->Fields->operator
[](iIndex)->FieldName);
}
```



當使用者在左下方的 ListBox2 中點選了要顯示資料的欄位後，範例程式使用了下面的巨集 SQL 命令來擷取資料：

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

上面的 SQL 命令使用了 2 個巨集，一個可動態指定資料表，另一個&FieldList 巨集則可動態根據使用者選擇的欄位來取得資料。

因此當使用選擇完成欄位之後可點選主表單中的"顯示資料"按鈕，這個按鈕呼叫 LoadData()方法取得資料：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  LoadData();
}
```

LoadData()方法根據使用者在左下方的 ListBox2 中選擇的欄位組合成欄位串列，再於 020~026 行把欄位名稱串列代入到巨集&FieldList1 之中：

```
001    void TfmMainForm::BuildFieldList(String &FieldList, String
```

```
Value)
002    {
003      if (FieldList == "")
004        FieldList = Value;
005      else
006        FieldList = FieldList + ", " + Value;
007    }
008
//---------------------------------------------------------------
009    void TfmMainForm::GetData(String FieldList)
010    {
011      qryData->Macros->operator [](0)->AsRaw = FieldList;
012      qryData->Macros->operator [](1)->AsRaw =
ListBox1->Items->operator [](ListBox1->ItemIndex);
013      qryData->Open();
014    }
015
//---------------------------------------------------------------
016    void TfmMainForm::LoadData()
017    {
018      String FieldList;
019
020      for (int iIndex = 0; iIndex < ListBox2->Items->Count;
iIndex++)
021      {
022        if (ListBox2->Selected[iIndex])
023        {
024          BuildFieldList(FieldList, ListBox2->Items->operator
[](iIndex));
025        }
026      }
027
028      GetData(FieldList);
029    }
```



最後資料就可以正確顯示在程式的資料感知元件中。

除了替換巨集之外，FireDAC 的 Escape sequence 巨集也非常的有用，而且彈性非常大，因為 Escape sequence 巨集允許程式師可在 SQL 命令中呼叫 FireDAC 定義的函式以及特定資料庫中的函式。例如假設你有一個稱為"List Price"的欄位，但在不同的資料庫中有的是使用單引號來指定：

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

有的可能是使用雙引號來指定：

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

這會讓程式師非常的麻煩，因為程式師需要根據不同的資料庫使用不同的符號。在這種情形中 FireDAC 的 escape sequence 巨集就非常方便了，程式師只需要使用：

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

一個 SQL 命令即可，其中的{就是 escape sequence 巨集，而{中的"id"則是 FireDAC 的巨集指令，它可自動判斷連結的資料庫而替換此 SQL 命令為

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



或是

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

FireDAC 提供的巨集指令分成 5 大類，它們是

| 巨集功能 | 說明 |
|---|---|
| 字元巨集 | http://docwiki.embarcadero.com/RADStudio/XE7/en/Character_Macro_Functions_(FireDAC) |
| 數值巨集 | http://docwiki.embarcadero.com/RADStudio/XE7/en/Numeric_Macro_Functions_(FireDAC) |
| 日期巨集 | http://docwiki.embarcadero.com/RADStudio/XE7/en/Date_and_Time_Macro_Functions_(FireDAC) |
| 系統巨集 | http://docwiki.embarcadero.com/RADStudio/XE7/en/System_Macro_Functions_(FireDAC) |
| 轉換巨集 | http://docwiki.embarcadero.com/RADStudio/XE7/en/Convert_Macro_Function_(FireDAC) |

例如在日期巨集中有一個 CURRENT_DATE()巨集函式，它可回傳今天的日期，因此當我們使用如下的程式碼就可以在資料集中加入 Today 欄位：

```
void __fastcall TfmMainForm::Button2Click(TObject *Sender)
{
  qryGeneral->SQL->Text = "select {CURRENT_DATE()} as Today, SRNAME
from Seminars";
  qryGeneral->Active = true;
  DataSource1->DataSet = qryGeneral;
}
```

| 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 資料表中有 2013 和 2014 年的研討會活動，那麼我們如何可以使用巨集功能從其中取得不同年次中的活動資訊呢？

那麼我們可以使用如下的巨集 SQL 命令：

```
select {Current_Date()} as Today, s.* FROM &TableName s WHERE
{Extract(Year, s.SRDATE)} = :iYear
```

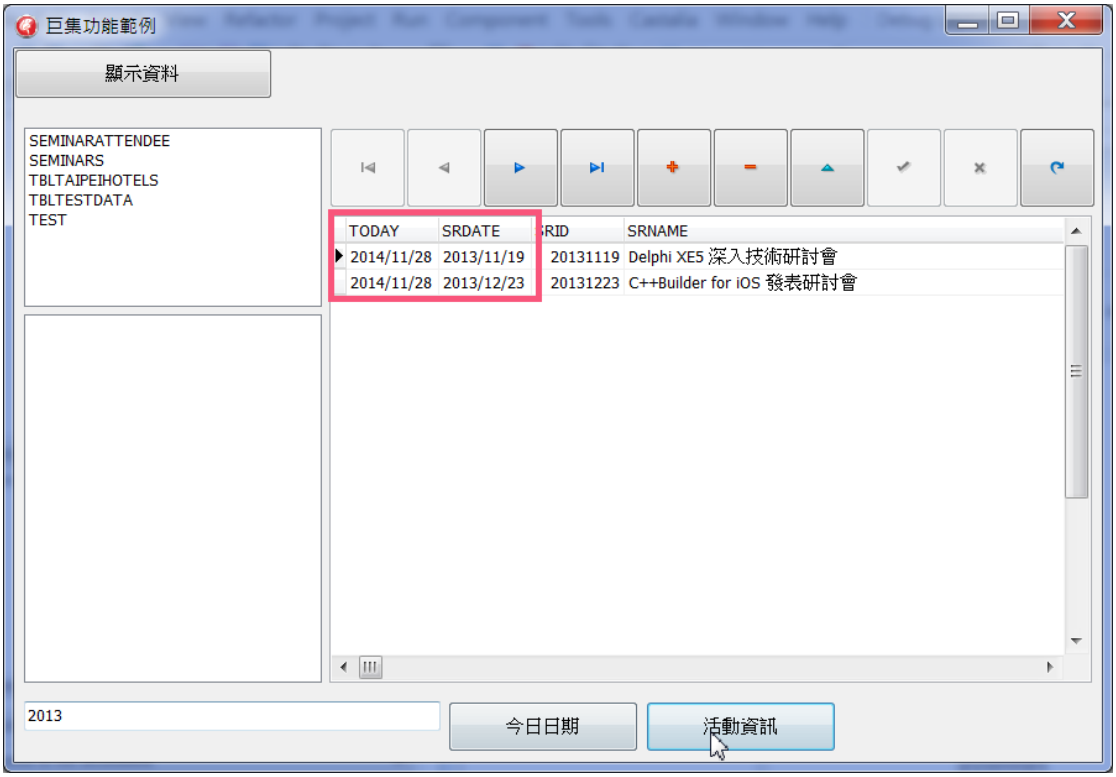
在上的巨集 SQL 命令中 Current_Date()，Extract 和 Year 都是 FireDAC 的巨集函式，Extract(Year, s.SRDATE)可從 SEMINARS 資料表的 SRDATE 欄位中取出年份資料。

最後可使用下面的程式碼動態代入資料表名稱和使用者打入的年份來顯示研討會活動：

```
void __fastcall TfmMainForm::Button3Click(TObject *Sender)
{
  qrySeminars->Macros->operator [](0)->AsRaw = "SEMINARS";
  qrySeminars->ParamByName("IYEAR")->Value =
```

```
StrToInt(edtYear->Text);
  qrySeminars->Active = true;
  DataSource1->DataSet = qrySeminars;
}
```

下面是執行範例程式並打入 2013 年，我們可以看到在今日(2014/11/28)日可以查詢到 2013 年的活動。




# 4-3 Update SQL 處理客製化資料

在實際的資料庫應用程式中經常會遇到使用 Join SQL 從多個資料表中擷取資料的狀況，程式師可以在 TFDQuery 的 SQL 特性中使用 Join SQL 取得資料，例如：

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

就是從 Seminars 和 SeminarAttendee 這 2 個資料表中取得參加每一個研討會的參加人員名單。

使用 TFDQuery 取得上述的資料雖然很簡單，但如果程式師想把 TFDQuery 中的 Join 的資料更新回多個資料表的話，那麼就不是這麼簡單了，因為程式師不能直接呼叫 Post 或是 ApplyUpdates 方法，因為 FireDAC 無法瞭解要把那一個欄位更新回那一個資料

表。那麼程式師如何能把異動資料更新回多個資料表，或是說程式師如何能把異動資料以客製化的方式更新回後端？

這有數種方式可以完成這個需求，例如程式師可以自行撰寫程式碼來更新資料，另外一比較省事的方式就是使用 FireDAC 的 TUpdateSQL 元件。

TUpdateSQL 元件有數個功能，例如它可以自動幫程式師產生處理資料的 DML SQL 命令，例如 Insert SQL，Update SQL 和 Delete SQL 等。但 TUpdateSQL 元件最主要提供了下面 2 項在處理複雜資料時最必要的功能：

1. 允許程式師定義客製化更新

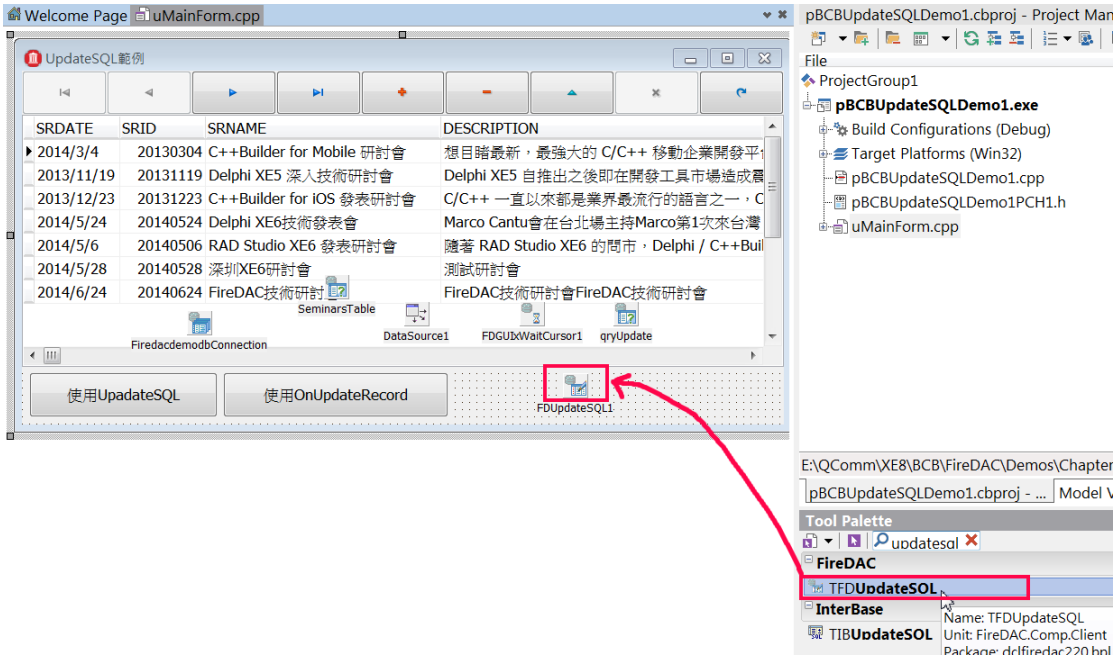
2. 允許程式師在 FireDAC 無法處理類似 Join SQL 的資料更新時提供更新資料的能力

TUpdateSQL 元件和 TFDQuery 元件需要一起工作，在 TFDQuery 元件中有一個 UpdateObject 特性，程式師只需要把這個特性指定為一個 TUpdateSQL 元件即可。或是在 TFDQuery 元件的 OnUpdateRecord 事件中藉由程式碼和 TUpdateSQL 元件來完成工作。讓我們使用 2 個範例來說明讀者就可以很快的瞭解。



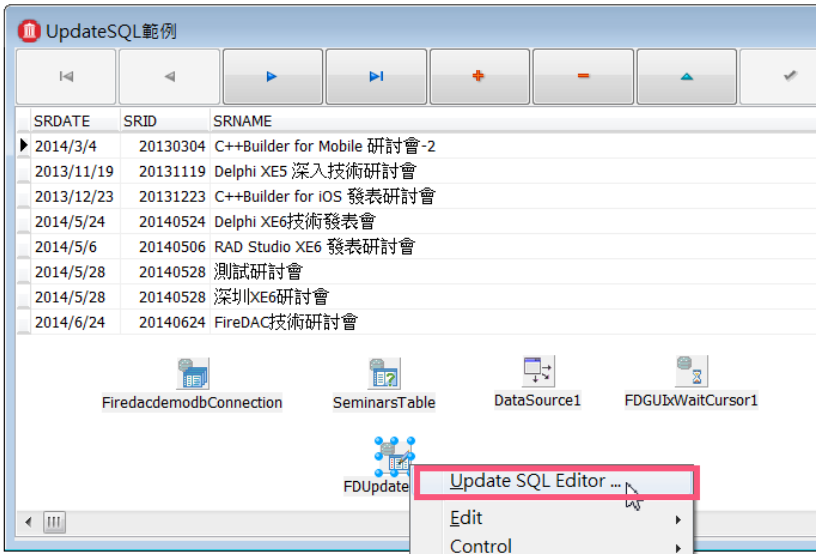
## 4-3-1 使用 TUpdateSQL 元件產生 DML

請建立一個 VCL 或是 FireMonkey 應用程式，雖後在其中放入 TFDConnection 和 TFDQuery 元件並連結到範例資料庫中的 SEMINARS 資料表，例如下面顯示的結果畫面，然後從元件盤拖曳 TFDUpdateSQL 元件到主表單中：



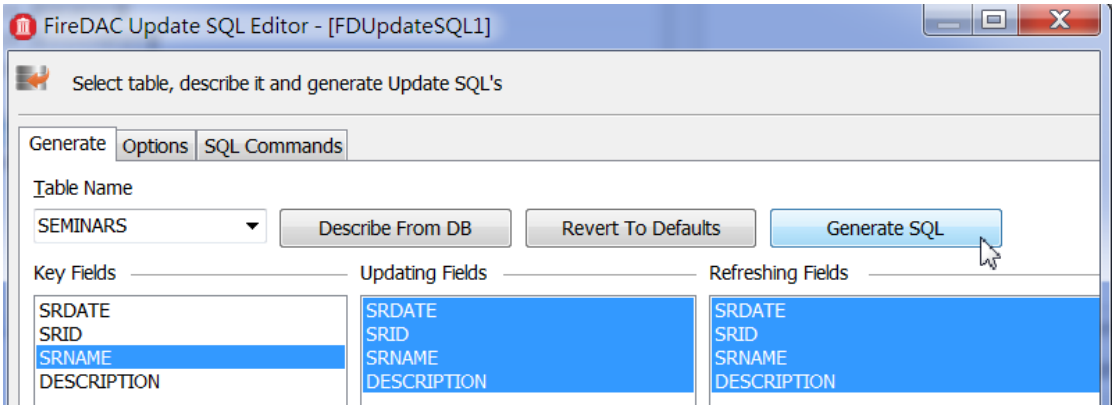

設定 TFDQuery 元件的 UpdateObject 特性為剛才拖曳的 TFDUpdateSQL 元件，點選 TFDUpdateSQL 元件再右擊滑鼠並點選"Update SQL Editor..."選單以啟動 TFDUpdateSQL 的元件編輯器：



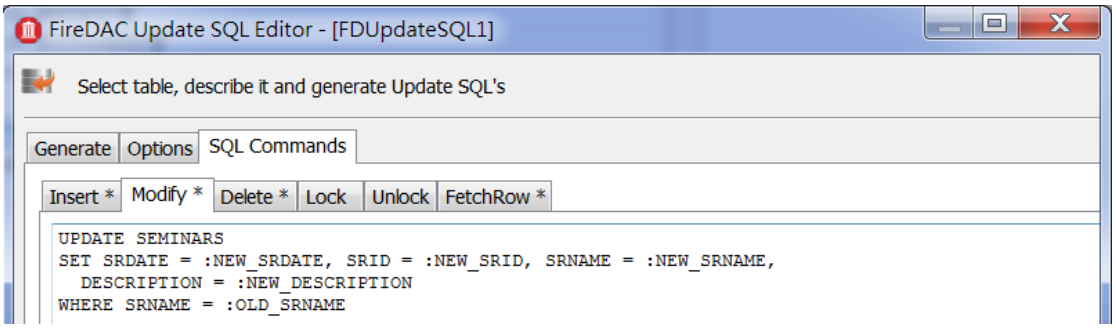



在 TFDUpdateSQL 的元件編輯器可以看到如下的畫面，在下方分成 3 個 ListBox，最左方的是 SEMINARS 資料表的 Key Fields 欄位其中會自動選擇 SEMINARS 資料表的鍵值欄位：SRNAME。中間的 ListBox 則列出所有 SEMINARS 資料表的欄位，程式師可以多選任何需要更新的欄位。最右方的 ListBox 也列出所有 SEMINARS 資料表的欄位，程式師也可多選任何在資料更新之後需要重新顯示的欄位。在下方的畫面中的選

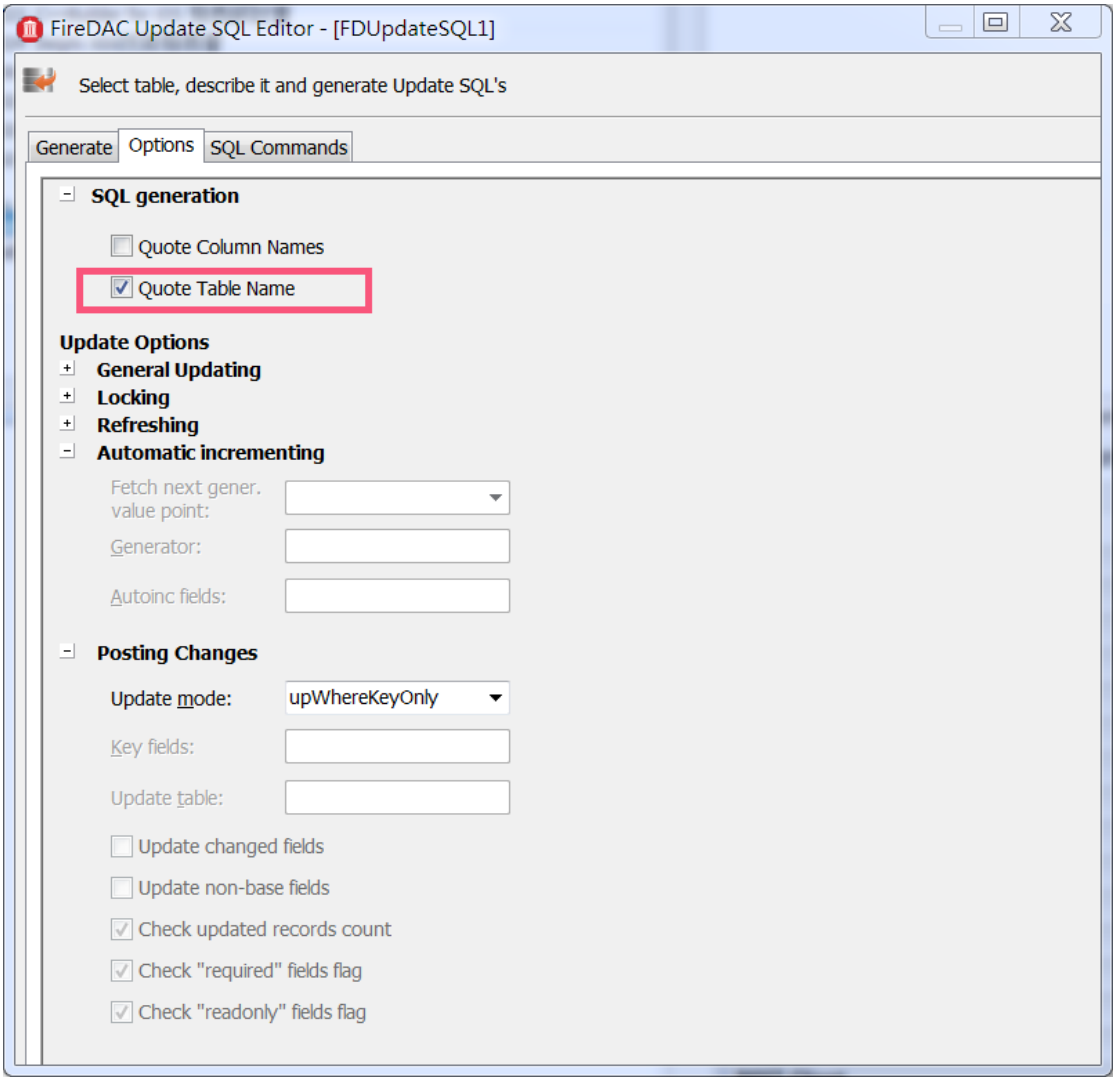
擇代表 SRNAME 是鍵值欄位，需要產生 Update SQL 更新所有的欄位而且在更新資料之後要重新顯示所有的欄位。



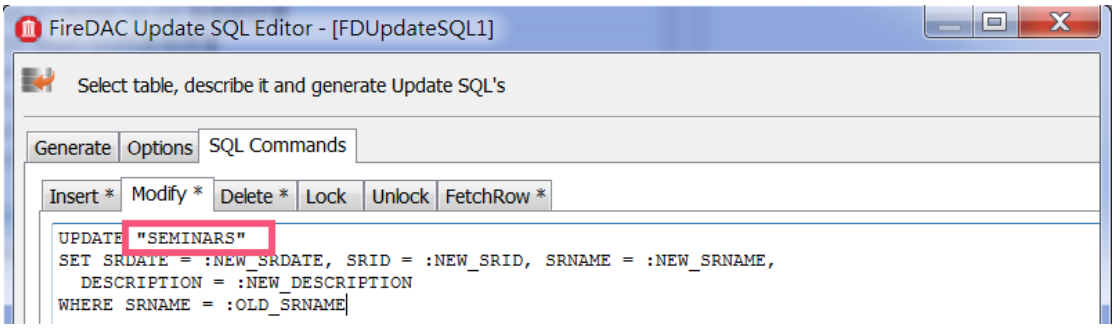

接著點選"Generate SQL"按鈕就可以自動產生所有的 DML SQL，例如下面就顯示了在點選"Generate SQL"按鈕之後再點選 SQL Command 頁次就可以看到 TFDUpdateSQL 元件自動根據我們剛才在"Generate"頁面中的選擇而自動產生的 Update SQL：



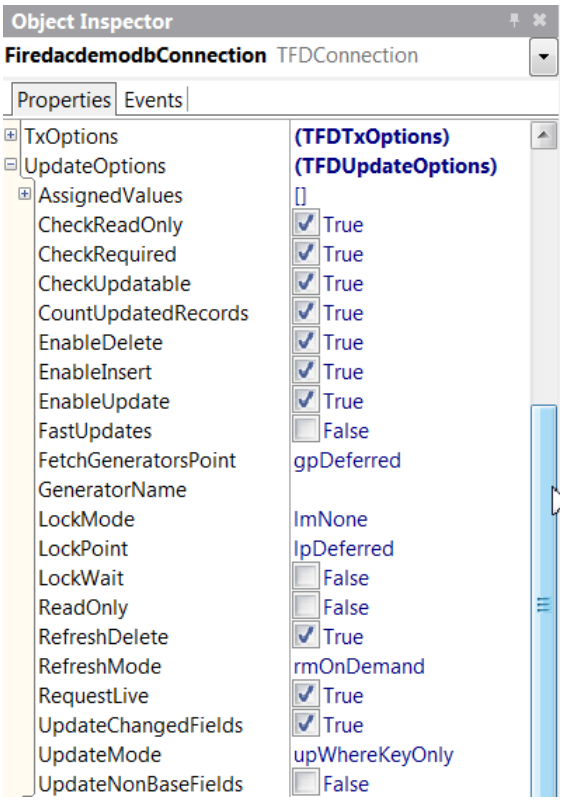

程式師也可以在"Options"頁次做更多的控制設定，例如如果設定"Options"頁次中的"Quote Table Name"：




選項再產生DML的話就可以看到Update SQL中的資料表名稱被雙引號包含起來了。




”Options”頁次中的許多設定和 TFDConnection 的 UpdateOptions 特值以及TFDQuery的特性有關，在稍後的章節中會說明這些特性的意義以及如何使用這些特性。

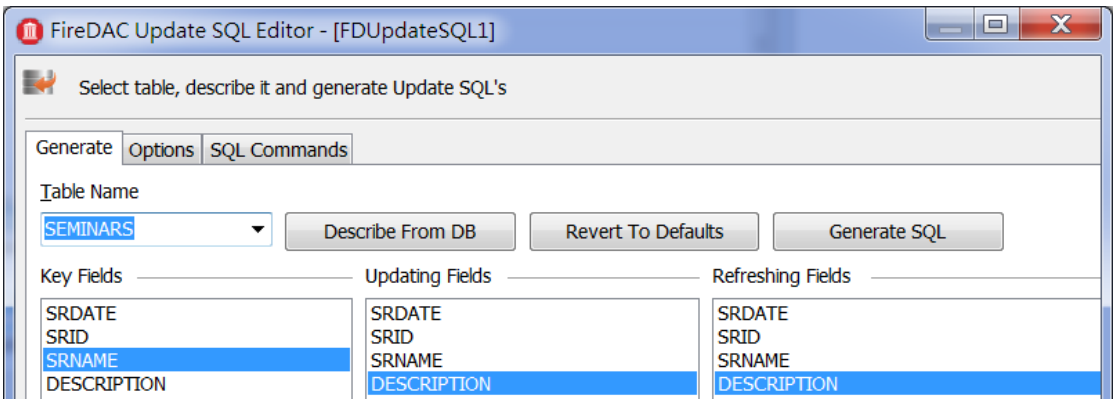




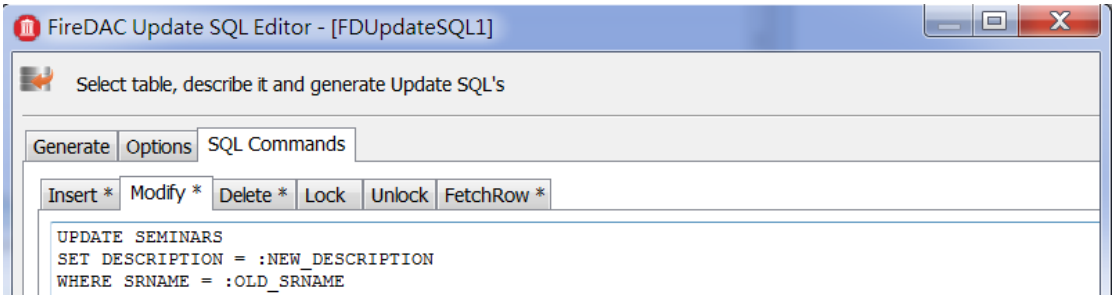

瞭解了如何使用 TFDUpdateSQL 元件自動產生 DML 之後就可以開始說明如何來使用它來更新資料了，

## 4-3-2 使用 TUpdateSQL 元件客製化資料更新

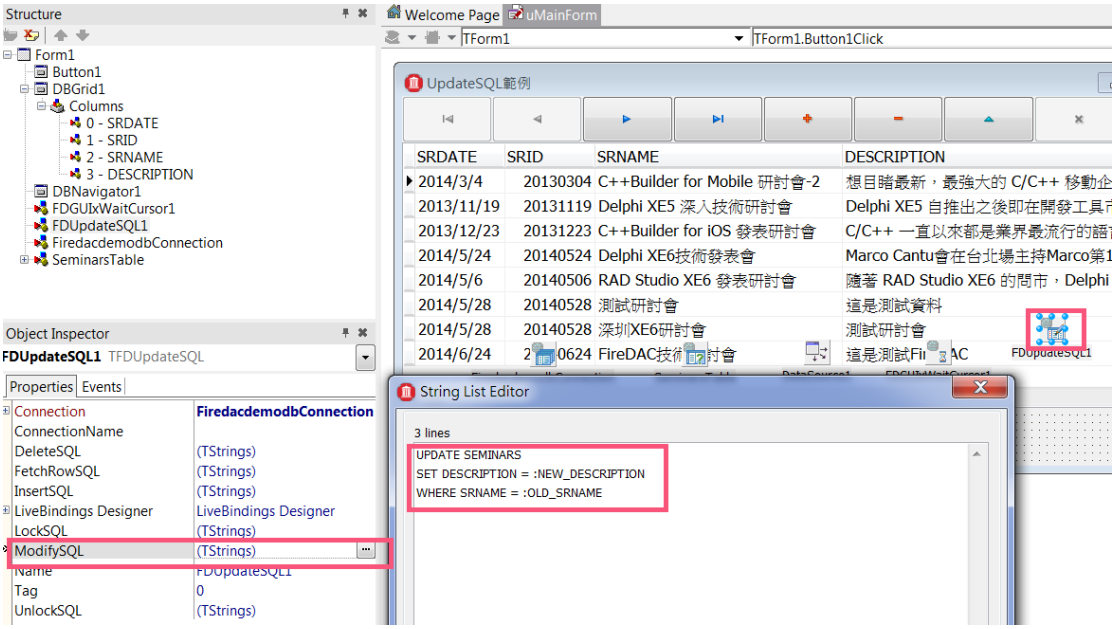
讓我們繼續用前面的 SEMINARS 資料表做為說明，假設在範例應用程式執行時我們只允許 SEMINARS 資料表的 DECCRIPTION 欄位能進行更新，那麼我們可以讓 TFDUpdateSQL 元件自動產生更新 DECCRIPTION 欄位的 Update SQL。請在 TFDUpdateSQL 元件中如下所示在 Update Fields 中只選擇 DECCRIPTION 欄位雖後點選”Update SQL”按鈕：



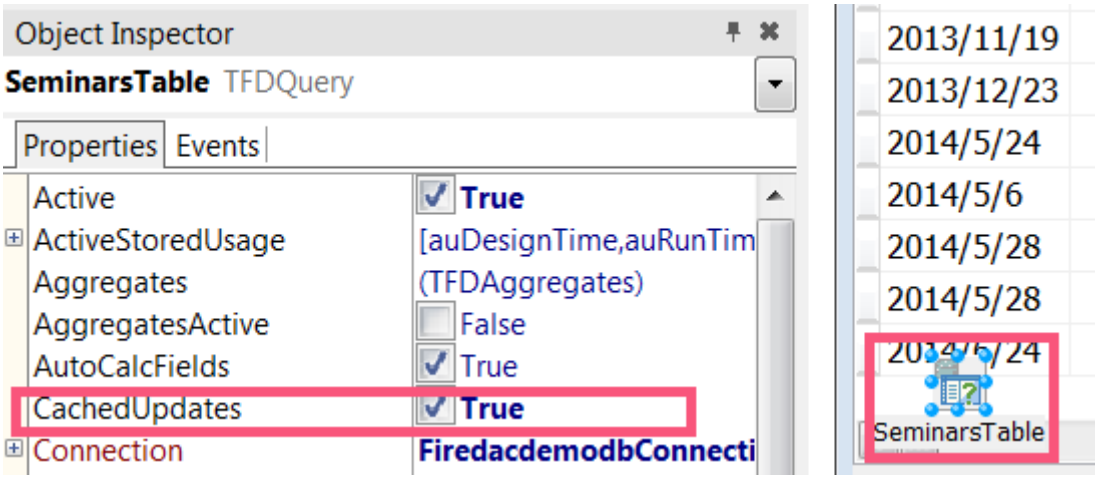
在 TFDUpdateSQL 元件的 Modify 頁面中就會產生如下的 Update SQL：




拷貝這個 Update SQL 並把它貼到 TFDUpdateSQL 元件的 ModifySQL 特性中，如下所示：




接著要設定 SeminarsTable 這個 TFDQuery 元件的 CachedUpdates 特性值為 True 以開啟 CachedUpdates 功能：



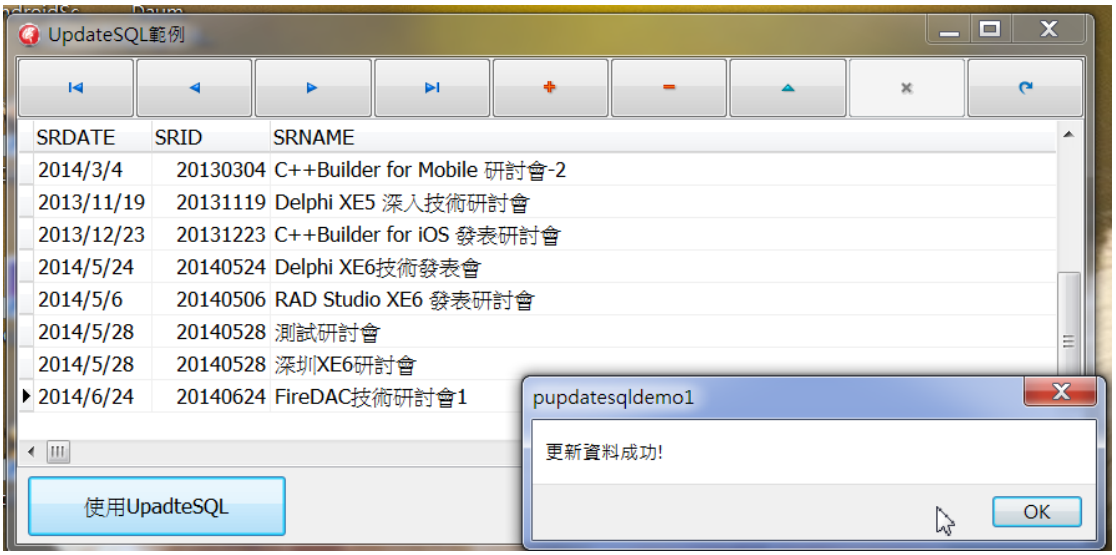

於主表單中放入一個 TButton 元件並在其 OnClick 事件中撰寫如下的程式碼：

```
001   void __fastcall TfmMainForm::Button1Click(TObject *Sender)
002   {
003     SeminarsTable->OnUpdateRecord = NULL;
004     SeminarsTable->UpdateObject = FDUpdateSQL1;
005     int NumErrors = SeminarsTable->ApplyUpdates(0);
006     if (NumErrors == 0)
007     {
008       SeminarsTable->CommitUpdates();
009       ShowMessage("更新資料成功!");
010     }
011     else
012       ShowMessage("產生 " + IntToStr(NumErrors) + " 個更新錯誤");
013   }
```

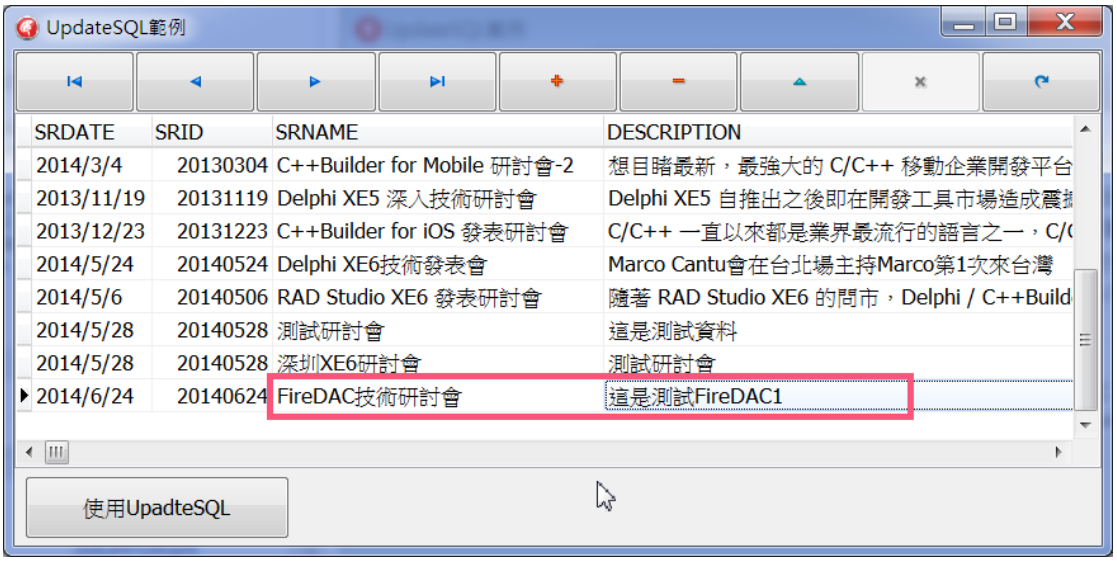


由於現在先展示如何使用 TFDUpdateSQL 元件展示如何客製化更新資料稍後會說明如何使用 OnUpdateRecord 事件客製化資料更新，因此 003 行先把 OnUpdateRecord 事設定為 NULL，004 行設定 SeminarsTable 的 UpdateObject 特性為 FDUpdateSQL1，005 行再呼叫 ApplyUpdates()把更新的資料更新回 SEMINARS 資料表。由於使用了 FDUpdateSQL1 元件，因此 ApplyUpdates() 就會使用 FDUpdateSQL1 元件中的 ModifySQL 特性中的 Update SQL。

現在如果執行此範例程式，並且如下更新 FireDAC 技術研討會那筆資料，讓我們同時異動 SRNAME 和 DESCRIPTION 這 2 個欄位的資料之後再點選”使用 UpdateSQL”按鈕就可以看到顯示” 更新資料成功!”訊息：




但重新顯示資料可以看到 DESCRIPTION 這個欄位的資料果然更新成功了，而 SRNAME 欄位的資料沒有更新：




這當然是因為 FDUpdateSQL1 元件中的 ModifySQL 特性的 Update SQL 只把 DESCRIPTION 欄位更新回 SEMINARS 資料表：

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

這個範例顯示了如何使用 TFDUpdateSQL 元件控制如何更新資料，

## 4-3-3 使用 OnUpdateRecord 事件客製化資料更新

TFDQuery 的 OnUpdateRecord 事件可以讓程式師更精確的控制如何把資料更新回後端。當使用者更新了資料之後，如果程式師使用 OnUpdateRecord 事件，那麼對每一筆異動的資料 TFDQuery 就會呼叫一次 OnUpdateRecord 事件讓程師來控制如何把資料更新回去。下面是 OnUpdateRecord 事件的宣告原型：

```
typedef void __fastcall (__closure
*TFDUpdateRecordEvent)(Data::Db::TDataSet* ASender,
Firedac::Stan::Option::TFDUpdateRequest ARequest,
Firedac::Stan::Intf::TFDErrorAction &AAction,
Firedac::Stan::Option::TFDUpdateRowOptions AOptions);
```

下面的表格說明了 OnUpdateRecord 事件參數的意義：

| 參數 | 說明 |
|---|---|
| ASender: TDataSet | ASender 代表要進行異動資的 TFDQuery 元件 |
| ARequest: TFDUpdateRequest | 代表資料異動的種類，是新增，修改，刪除還是其他的異動 |
| var AAction: TFDErrorAction | 代表發生異動錯誤時要採取的行動 |
| AOptions: TFDUpdateRowOptions | 代表要進行這筆資料異動時要採取的額外選項 |



所以如果要使用 OnUpdateRecord 事件來進行和上一小節一樣的更新動作，我們可以使用如下的程式碼：

```
001    void __fastcall TfmMainForm::Button2Click(TObject *Sender)
002    {
003      SeminarsTable->OnUpdateRecord =
SeminarsTableUpdateRecord;
004      SeminarsTable->UpdateObject = NULL;
005      int NumErrors = SeminarsTable->ApplyUpdates(0);
006      if (NumErrors == 0)
007      {
008        SeminarsTable->CommitUpdates();
009        ShowMessage("更新資料成功!");
010      }
011      else
```

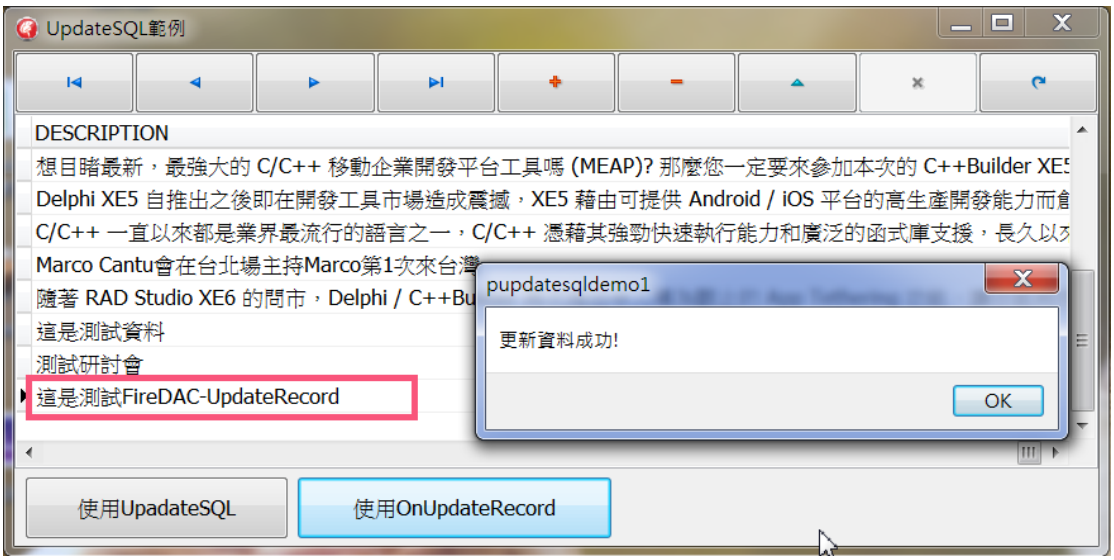
```
012       ShowMessage("產生 " + IntToStr(NumErrors) + " 個更新錯誤");
013    }
014
//---------------------------------------------------------------------
015    void __fastcall
TfmMainForm::SeminarsTableUpdateRecord(TDataSet *ASender,
TFDUpdateRequest ARequest,
016            TFDErrorAction &AAction, TFDUpdateRowOptions
AOptions)
017    {
018      if (ARequest == arUpdate)
019      {
020        FDUpdateSQL1->DataSet = SeminarsTable;
021        FDUpdateSQL1->Apply(ARequest, AAction, AOptions);
022        AAction = TFDErrorAction::eaApplied;
023      }
024    }
```



首先 003 行先設定 SeminarsTable 的 OnUpdateRecord 事件處理函式再於 005 行呼叫 ApplyUpdates() 方法讓 OnUpdateRecord 事件處理函式 SeminarsTableUpdateRecord 來更新資料。

015 行開始的 SeminarsTableUpdateRecord()先於 018 行判斷目前的異動是不是更新的動作，如果是的話就進行 018~023 行呼叫 FDUpdateSQL1 的 Apply()方法真正把資料更新回 SEMINARS 資料表再於 025 行設定 AAction 參數為 TFDErrorAction.eaApplied 代表更新成功，

現在再次執行範例程式，更改同樣一筆資料如下所示，再點選”使用 OnUpdateRecord”按鈕就可以看到也可以成功把資料更新回後端。




## 4-3-4 使用 TUpdateSQL 元件處理複雜資料更新

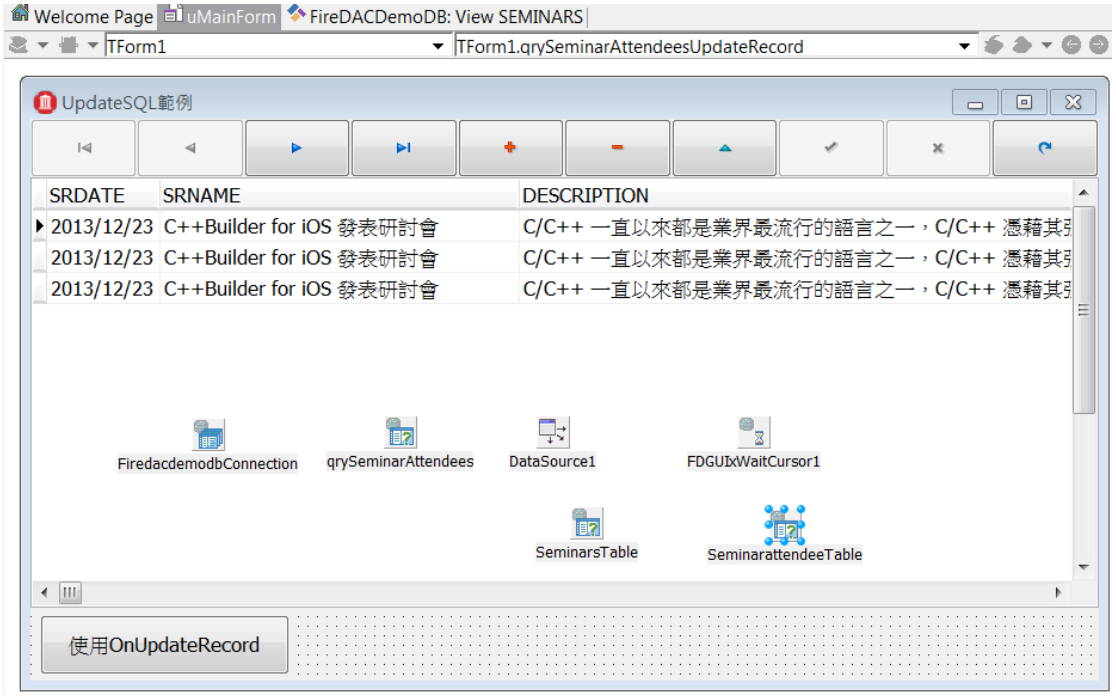
如果資料庫應用程式使用了 JOIN SQL 顯示資料並允許使用者修改資料的話，那麼要如何做到？例如下面的 SQL 命令從 SEMINARS 和 SEMINARATTENDEE 這 2 個資料表中擷取資料，那如何能讓 FireDAC 的應用程式可以修改並把資料更新回後端？



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

一般來說這並不容易，因為使用 Join SQL 取得的資料都是唯讀的，而且 TFDQuery 也無法自動把這種資料更新回後端多個資料表，那麼是否可用 Update SQL 的功能克服這個需求？答案是可以的，Update SQL 不但可以提供 UI 修改 Join SQL 取得的資料，也可以讓程式師使用程式碼這種資料更新回後端多個資料表。

例如下面顯示的第 2 個 Update SQL 的範例程式: pUpdateSQLDemo2.dproj，其主表單中的 qrySeminarAttendees 元件就使用了上面的 Join SQL 從 SEMINARS 和 SEMINARATTENDEE 這 2 個資料表中擷取資料並顯示在 DBGrid 中：




qrySeminarAttendees 元件設定了其 CachedUpdates 特性值為 True 以使用 Update SQL 功能：




但如果現在您執行此範例應用程式會發現在 DBGrid 中只能修改 SEMINARS 資料表的欄位資料：SRDATE，SRNAME 和 DESCRIPTION，而無法修改 SEMINARATTENDEE 資料表的欄位資料：Name，Email 和 Phone。為什麼？

如同前面說的 Join 的資料是唯讀的，打開 CachedUpdates 功能也只能讓使用者可修改主資料表的資料，也就是 SEMINARS，而無法修改從資料表，也就是 SEMINARATTENDEE。因此要允許可修改從資料表，除了開啟 CachedUpdates 功能程式師必須再開啟 TFDQuery 元件的 UpdateOptions | UpdateNonBaseFields 特性：

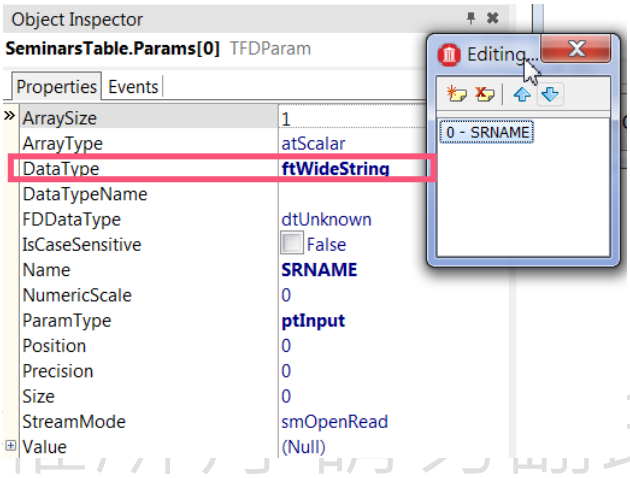

一旦設定 qrySeminarAttendees 的 UpdateOptions | UpdateNonBaseFields 特性為 True 之後現在就可以在 DBGrid 中修改資料了。

為了稍後簡化說明的程式碼，因此讓我們設定要修改並且更新 qrySeminarAttendees 中的 DESCRIPTION 和 NAME 這 2 個欄位。DESCRIPTION 是屬於 SEMINARS 資料表，而 NAME 則是屬於 SEMINARATTENDEE 資料表，這也代表我們需要同時更新 2 個資料表。

在這個範例中使用了 2 個 TFDQuery 元件來更新資料回 SEMINARS 和 SEMINARATTENDEE 資料表，其中的 SeminarsTable 元件使用下面的 SQL 命令：

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

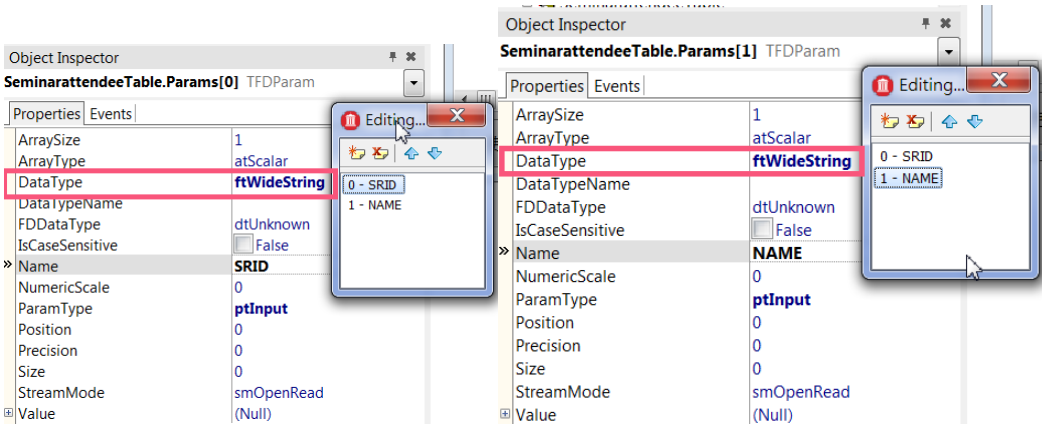
SeminarsTable 元件會使用 SRNAME 欄位到 SEMINARS 資料表中搜尋被異動的資料然後更新它的 DESCRIPTION 欄位值。由於 SeminarsTable 元件使用了動態參數，因此請記得要設定動態參數的資料型態，由於 SRNAME 欄位是字串欄位，因此我們設定它的資料型態是 ftWideString：




SeminarattendeeTable 元件使用下面的 SQL 命令更新 NAME 欄位：

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

它使用了 2 個動態參數，因此我們也需要設定這 2 個動態參數的資料型態如下：



完成了這些設定之後就可以開始撰寫程式碼來進行實際的資料更新工作了。首先在主表單的"使用 OnUpdateRecord"按鈕中撰寫如下的程式碼使用 OnUpdateRecord 事件來更新資料：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  int NumErrors;

  qrySeminarAttendees->OnUpdateRecord =
qrySeminarAttendeesUpdateRecord;
  qrySeminarAttendees->UpdateObject = NULL;
  FiredacdemodbConnection->StartTransaction();
  try
  {
  NumErrors = qrySeminarAttendees->ApplyUpdates(0);
  qrySeminarAttendees->CommitUpdates();
  FiredacdemodbConnection->Commit();
  ShowMessage("更新資料成功!");
  }
  catch (...)
  {
  FiredacdemodbConnection->Rollback();
  ShowMessage("產生 " + IntToStr(NumErrors) + " 個更新錯誤");
  }
}
```



在 OnUpdateRecord 事件處理函式中 004 行先判斷是否為修改的動作，如果是的話就在 006 行呼叫 UpdateSeminars()方法更新資料回 SEMINARS 資料表，007 行呼叫 UpdateSeminarAttendees()方法更新資料回 SEMINARATTENDEE 資料表：

```
001    void __fastcall
TfmMainForm::qrySeminarAttendeesUpdateRecord(TDataSet *ASender,
TFDUpdateRequest ARequest,
002             TFDErrorAction &AAction, TFDUpdateRowOptions
AOptions)
003    {
004      if (ARequest == arUpdate)
```

```
005     {
006       UpdateSeminars(ASender);
007       UpdateSeminarAttendees(ASender);
008       AAction = TFDErrorAction::eaApplied;
009     }
010   }
```

UpdateSeminars()方法先在 004 行呼叫 FindSeminar()方法使用 SeminarsTable 元件在 SEMINARS 資料表中搜尋被修改的資料，如果找到的話就把修改的 DESCRIPTION 欄位值更新回 SEMINARS 資料表：

```
001   void TfmMainForm::UpdateSeminars(TDataSet *ASender)
002   {
003     String sName = ASender->FieldByName("SRNAME")->Value;
004     if (FindSeminar(sName))
005     {
006       if
(!VarIsNull(ASender->FieldByName("DESCRIPTION")->Value))
007       {
008         SeminarsTable->Edit();
009         SeminarsTable->FieldByName("DESCRIPTION")->Value =
ASender->FieldByName("DESCRIPTION")->Value;
010         SeminarsTable->Post();
011       }
012     }
013   }
```



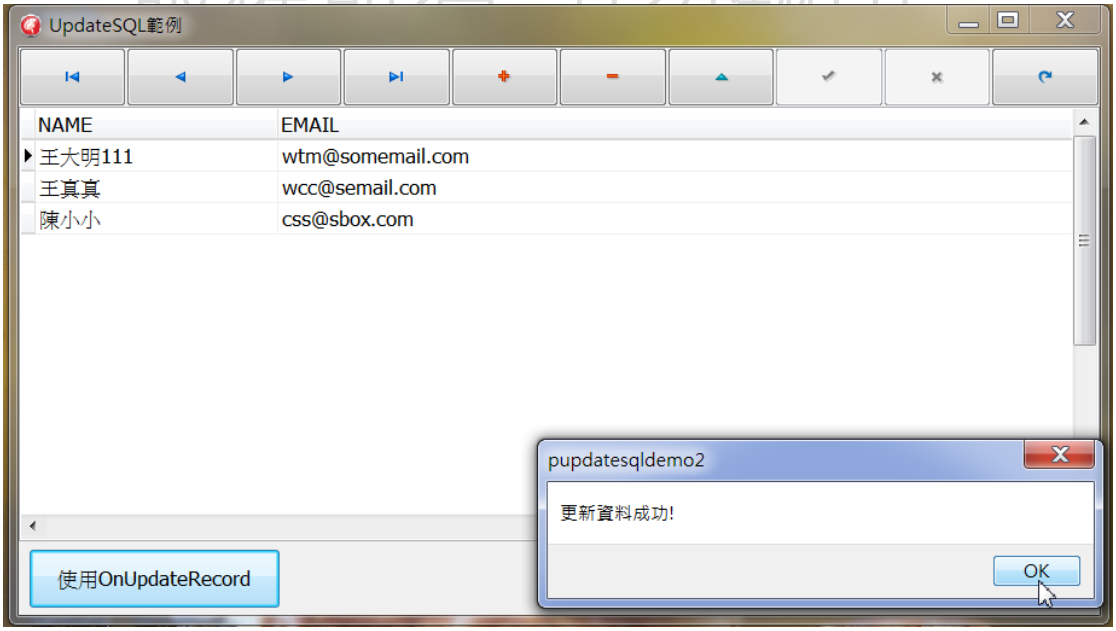
UpdateSeminarAttendees() 方法在 004 行呼叫 FindSeminar() 方法使用 SeminarsTable 元件在 SEMINARS 資料表中搜尋被修改的資料，再藉由連結 SEMINARS 和 SEMINARATTENDEE 資料表和 SRID 欄位值在 008 行呼叫 FindAttendee() 在 SEMINARATTENDEE 資料表中找到被修改的資料再把 NAME 欄位值更新：

```
001   void TfmMainForm::UpdateSeminarAttendees(TDataSet *ASender)
002   {
003     String sName = ASender->FieldByName("SRNAME")->Value;
004     if (FindSeminar(sName))
005     {
```

```
006       if
(FindAttendee(SeminarsTable->FieldByName("SRID")->AsString,
ASender->FieldByName("NAME")->OldValue))
007       {
008         if (! VarIsNull(ASender->FieldByName("NAME")->Value))
009         {
010            SeminarattendeeTable->Edit();
011            SeminarattendeeTable->FieldByName("NAME")->Value =
ASender->FieldByName("NAME")->Value;
012            SeminarattendeeTable->Post();
013         }
014       }
015     }
016   }
```

現在執行此範例程式就可以看到如下的畫面，我們不但可以在 DBGrid 中修改資料，也可以把 DESCRIPTION 和 NAME 欄位的資料分別更新回 SEMINARS 和 SEMINARATTENDEE 資料表。我們藉由 OnUpdateRecord 功能完成了一般狀況下無法做到的工作。






在這個範例中我們是使用 2 個 TFDQuery 元件更新資料，當然程式師也可以直接使用 Update SQL 來更新資料。

不過上面的程式碼雖然可以正常工作，但嚴格來說並不正確，為什麼？再仔細回去看看前面的 Button1Click 事件處理函式，它在呼叫了 qrySeminarAttendees 的

ApplyUpdates()方法後只呼叫 qrySeminarAttendees 的 CommitUpdates()方法確定資料更新動作完成。但在這個範例中真正進行資料更新的動作是由 SeminarsTable 和 SeminarattendeeTable 元件進行的，而且 qrySeminarAttendees 並不能更新資料(它是唯讀的)。因此在上面的程式碼中在內定的模式中 SeminarsTable 和 SeminarattendeeTable 元件分別會啟動一個資料庫交易，因此可能會發生 SeminarsTable 更新成功 SeminarattendeeTable 失敗或是反過來的情形。

因此我們需要更改 Button1Click 事件處理函式，在 qrySeminarAttendees 呼叫 ApplyUpdates() 方法之前先手動啟動資料庫交易，把 SeminarsTable 和 SeminarattendeeTable 元件更新資料的動作置於一個相同的資料庫交易環境中：

```
001    void __fastcall TfmMainForm::Button1Click(TObject *Sender)
002    {
003      int NumErrors;
004
005      qrySeminarAttendees->OnUpdateRecord =
qrySeminarAttendeesUpdateRecord;
006      qrySeminarAttendees->UpdateObject = NULL;
007      FiredacdemodbConnection->StartTransaction();
008      NumErrors = qrySeminarAttendees->ApplyUpdates(0);
009      if (NumErrors)
010      {
011        qrySeminarAttendees->CommitUpdates();
012        FiredacdemodbConnection->Commit();
013        ShowMessage("更新資料成功!");
014      }
015      else
016      {
017        FiredacdemodbConnection->Rollback();
018        ShowMessage("產生 " + IntToStr(NumErrors) + " 個更新錯誤");
019      }
020    }
```



如此一來 SeminarsTable 和 SeminarattendeeTable 元件就可保證可同時更新成功或是同時恢復更新。

但由於我們只是藉由 qrySeminarAttendees 觸發 OnUpdateRecord 事件，資料並不是由 qrySeminarAttendees 更新回後端，因此最後我們可以再修改 Button1Click

如下，使用例外處理程式碼 try…catch(…)來判斷是否是確認更新成功或是恢復資料更新的動作：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  int NumErrors;

  qrySeminarAttendees->OnUpdateRecord =
qrySeminarAttendeesUpdateRecord;
  qrySeminarAttendees->UpdateObject = NULL;
  FiredacdemodbConnection->StartTransaction();
  try
  {
  NumErrors = qrySeminarAttendees->ApplyUpdates(0);
  qrySeminarAttendees->CommitUpdates();
  FiredacdemodbConnection->Commit();
  ShowMessage("更新資料成功!");
  }
  catch (...)
  {
  FiredacdemodbConnection->Rollback();
  ShowMessage("產生 " + IntToStr(NumErrors) + " 個更新錯誤");
  }
}
```



藉由前面的 2 個範例您應該就可以使用 FireDAC 的 Update SQL 功能來更新一個或是多個資料表的資料了。

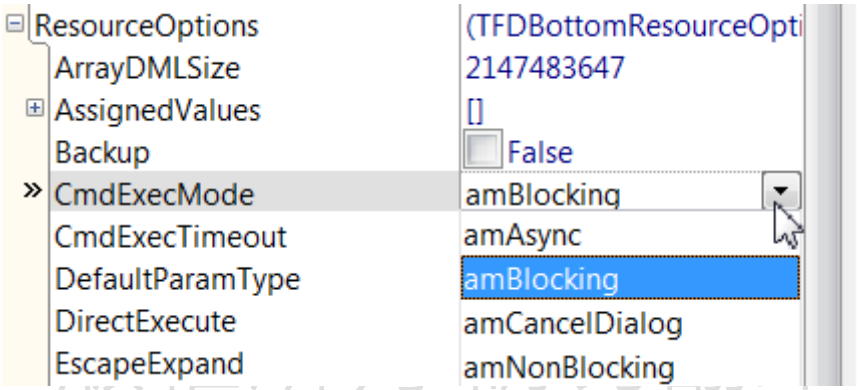
# 4-4 非同步處理資料

FireDAC 和 BDE/dbExpress 不同的地方之一就是 FireDAC 允許開發人員使用非同步的方式來處理資料，雖然 FireDAC 在內定上還無法同時支援使用多個執行緒來查詢和處理資料，但 FireDAC 至少在內定功能中已經建立了非同步處理資料的能力，讓開發人員可以把 UI 和資料處理的動作分離以避免要長時間處理資料時會暫停 UI 的反應。

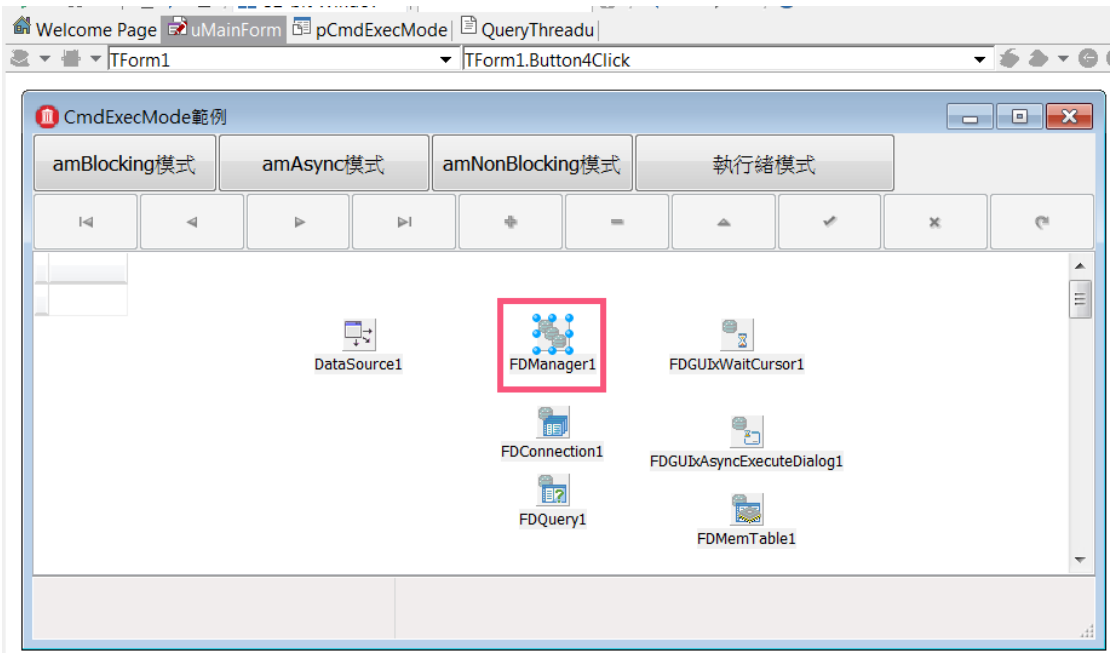
FireDAC 的資料處理功能提供了 4 種模式，下面的表格說明了這 4 種不同的方式：

| 模式 | 說明 |
| --- | --- |
| amBlocking | 呼叫執行緒和 UI 都會暫停一直到資料處理動作完成 |
| amNonBlocking | 呼叫執行緒會暫停一直到資料處理動作完成,但 UI 不會被暫停 |
| amCancelDialog | 呼叫執行緒和UI都會暫停一直到資料處理動作完成.但FireDAC提供一個對話盒可取消資料處理動作 |
| amAsync | 呼叫執行緒和UI都不會暫停. 呼叫資料處理動作之後會立刻回到呼叫執行緒繼續工作. 資料處理動作在後端執行. |

要使用 FireDAC 的非同步資料處理功能，程式師必須搭配使用 TFDManager 元件，再設定 TFDQuery 元件的 ResourceOptions.CmdExecMode 特性值：




下面的範例程式展示了如何使用這 4 種模式，在使用不同的模式時程式師需要注意一些事情，那就是 UI 和執行緒的關係，讓我們使用這個範例程式的程式碼來說明，



在範例程式的主表單中可以看到必須使用 TFDManager 元件，接著看看下面使用"amBlocking 模式"的實作程式碼。由於 amBlocking 模式其實就是使用主執行緒去執行資料處理的工作，因此主表單中的資料感知元件可以直接連結到進行資料處理工作的 FDQuery1 元件：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  FDQuery1->DisableControls();
  lStart = GetTickCount();
  try
  {
  DataSource1->DataSet = FDQuery1;
  FDQuery1->AfterOpen = NULL;
  FDQuery1->ResourceOptions->CmdExecMode = amBlocking;
  FDQuery1->Open();
  }
  __finally
  {
  lEnd = GetTickCount();
  FDQuery1->EnableControls();
  }
  ShowRunTime(lEnd - lStart);
}
```



但在下面的 amNonBlocking 模式中，我們再最在 FDQuery1 進行資料處理工作之前先切斷 FDQuery1 和資料感知元件的連結，也就是要設定連結到 FDQuery1 的 DataSource1 的 DataSet 特性值為 NULL：

```
void __fastcall TfmMainForm::Button3Click(TObject *Sender)
{
  lStart = GetTickCount();
  FDQuery1->Close();
  FDQuery1->DisableControls();
  DataSource1->DataSet = NULL;
  FDQuery1->AfterOpen = QueryAfterOpen;
  FDQuery1->ResourceOptions->CmdExecMode = amNonBlocking;
  FDQuery1->Open();
```

```
  lEnd = GetTickCount();
  ShowRunTime(lEnd - lStart);
}
```

如果是使用 amAsync 模式那麼是一樣要先切斷 FDQuery1 和資料感知元件的連結，如下面 006 行所示，這是因為使用 amAsync 模式時，處理資料工作的執行緒和主執行緒不同，因為在資料工作的執行緒執行時，必須和 UI 隔離。

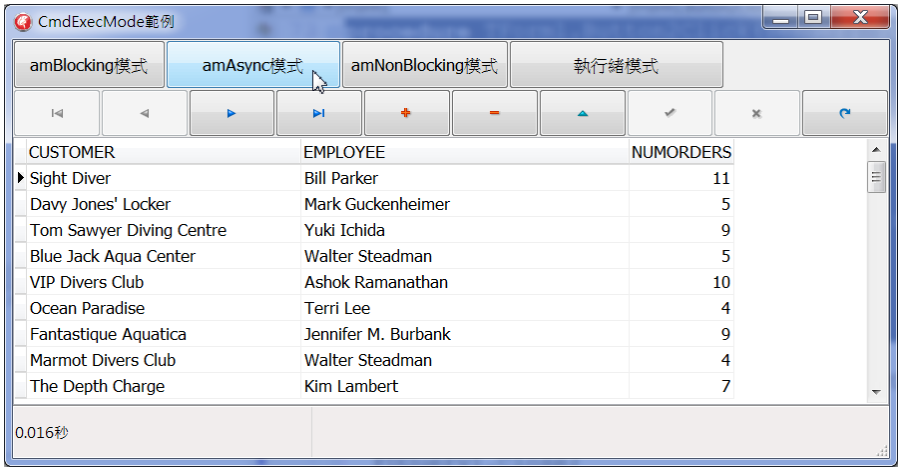
```
001    void __fastcall TfmMainForm::Button2Click(TObject *Sender)
002    {
003      lStart = GetTickCount();
004      FDQuery1->Close();
005      FDQuery1->DisableControls();
006      DataSource1->DataSet = NULL;
007      FDQuery1->AfterOpen = QueryAfterOpen;
008      FDQuery1->ResourceOptions->CmdExecMode = amAsync;
009      FDQuery1->Open();
010      lEnd = GetTickCount();
011      ShowRunTime(lEnd - lStart);
012    }
```



等到處理資料工作的執行緒執行完畢後再連結 UI 的資料感知元件，因此在才 FDQuery1 的因 OnAfterOpen 事件中執行設定 DataSource1 的 DataSet 特性值回 FDQuery1：

```
void __fastcall TfmMainForm::QueryAfterOpen(TDataSet *DataSet)
{
  DataSource1->DataSet = FDQuery1;
  DataSource1->DataSet->EnableControls();
}
```

例如下面的畫面就是 amAsync 模式成功執行的結果：




最後程式師也可以使用 C++Builder 的 TThread 類別使用多個不同的執行緒來執行不同的資料處理工作，例如下面是”執行緒模式”的實作程式碼，它建立了一個 TQueryThread 物件並把一個執行資料處理工作的 SQL 命令和一個 TFDMemTable 元件傳遞給它，再呼叫 Start()方法啟動它：

```
void __fastcall TfmMainForm::Button4Click(TObject *Sender)
{
  TQueryThread *QueryThread = new TQueryThread(FDQuery1->SQL->Text,
FDMemTable1);

  DataSource1->DataSet = NULL;
  QueryThread->FreeOnTerminate = true;
  QueryThread->OnTerminate = ThreadDone;
  lStart = GetTickCount();
  QueryThread->Start();
}
```



當這個 TQueryThread 物件執行完畢之後再連結資料感知元件：

```
void __fastcall TfmMainForm::ThreadDone(TObject *Sender)
{
  lEnd = GetTickCount();
  ShowRunTime(lEnd - lStart);
  DataSource1->DataSet = FDMemTable1;
}
```

TQueryThread 類別是從 TThread 類別繼承下來：

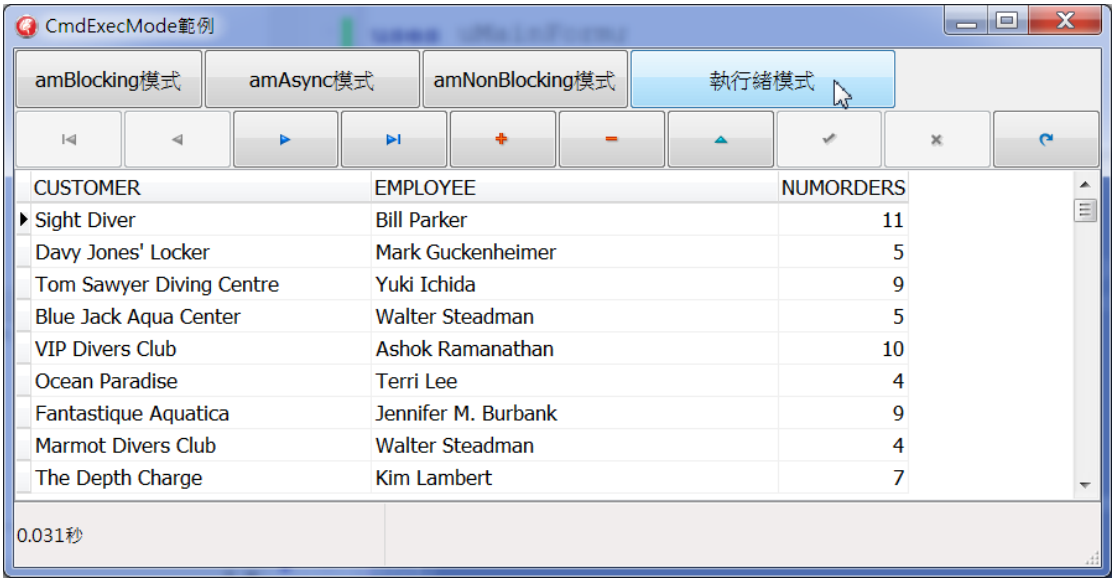
```
class TQueryThread : public TThread
```

它會自行建立一個 TFDConnection 元件再使用它連結到資料庫然後執行傳入的 SQL 命令，最後再把執行結果的資料集拷貝到傳入的 TFDMemTable 元件中：

```
001    __fastcall TQueryThread::TQueryThread(bool CreateSuspended)
002        : TThread(CreateSuspended)
003    {
004    }
005
//---------------------------------------------------------------
-------------
006    __fastcall TQueryThread::TQueryThread(String SQLText,
TFDMemTable *FDMemTable)
007        : TThread(true)
008    {
009      FDConnection = new TFDConnection(NULL);
010      FDConnection->ConnectionDefName = "dbDemos";
011      FDConnection->LoginPrompt = false;
012      FDConnection->Open();
013      FDQuery = new TFDQuery(NULL);
014      FDQuery->Connection = FDConnection;
015      FDQuery->SQL->Text = SQLText;
016      FDQuery->ResourceOptions->CmdExecMode = amBlocking;
017      FFDMemTable = FDMemTable;
018    }
019
//---------------------------------------------------------------
020    __fastcall  TQueryThread::~TQueryThread(void)
021    {
022      delete FDQuery;
023      delete FDConnection;
024    }
025
//---------------------------------------------------------------
026    void __fastcall TQueryThread::Execute()
```

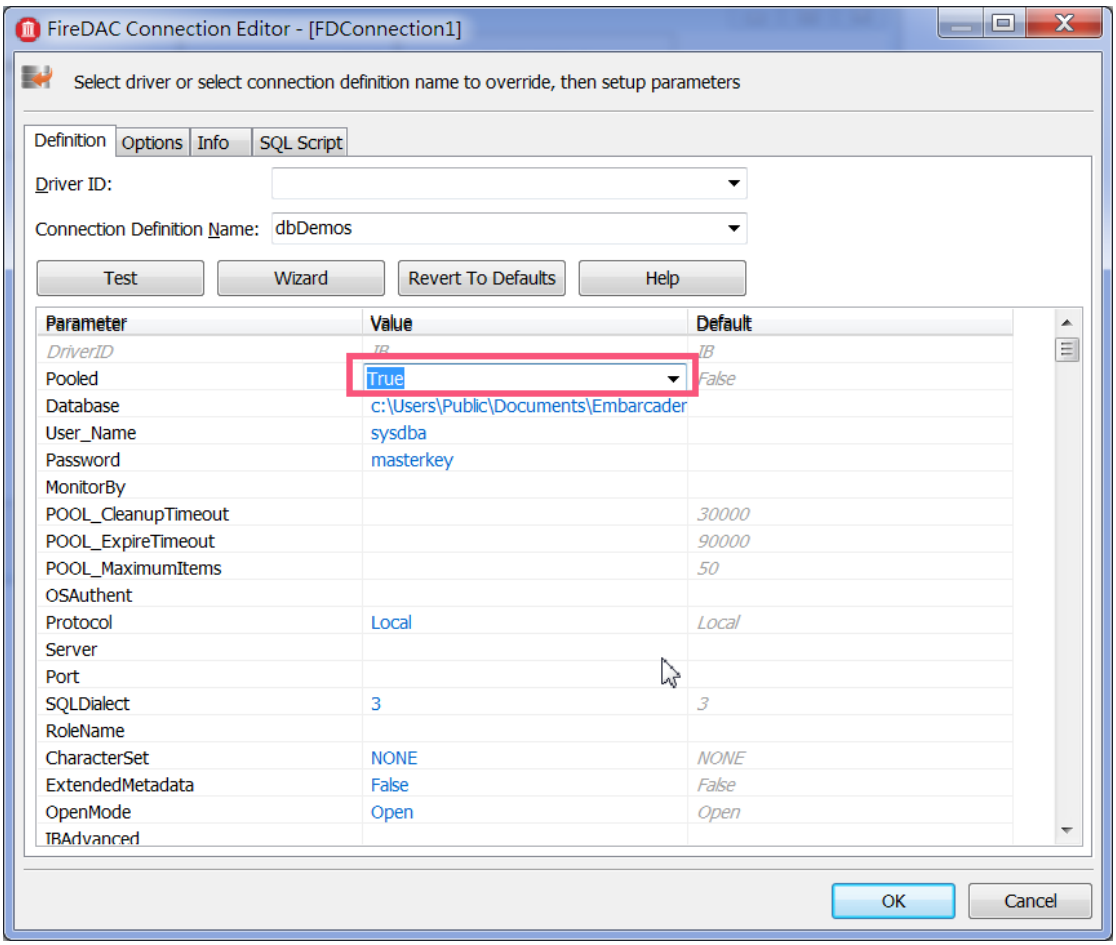
```
027    {
028       //---- Place thread code here ----
029      FDQuery->Open();
030      Synchronize(&CloneCursor);
031    }
032
//---------------------------------------------------------------------------
033    void __fastcall TQueryThread::CloneCursor()
034    {
035      FFDMemTable->CloneCursor(FDQuery, true);
036    }
```

下面的畫面就是執行緒模式成功執行的結果：




不過程式師一定要記得如果使用執行緒模式使用多個執行緒執行資料處理的工作的話，就一定要開啟資料庫的連結池功能以避免建立過多的資料庫連結。

例如下面即顯示了這個範例程式開啟了使用的 InterBase 的連結池功能：




## 4-5 結論

本章的內容主要是說明如何使用 FireDAC 使用一些進階的技巧來處理資料，這些技巧在開發較為複雜的資料庫應用程式時會非常的有幫助，正確使用這些技巧也能夠讓程式師開發出高執行效率的資料庫應用程式，當然本章討論的內容也都可以使用在移動平台中，在稍後的章節中會討論更多使用其他 FireDAC 元件處理資料的技巧。

# 第5章 FireDAC 更多的功能

在前面的章節中已經討論了許多 FireDAC 的功能，在本章中將繼續說明其他的功能，這些功能有的是在較新的 FireDAC 版本才出現的，這些新功能在使用上並不困難但卻非常的實用。



## 5-1 批次處理

在傳統的資料存取中如果我們需要執行 2 個 SQL 命令，例如下面的 SQL

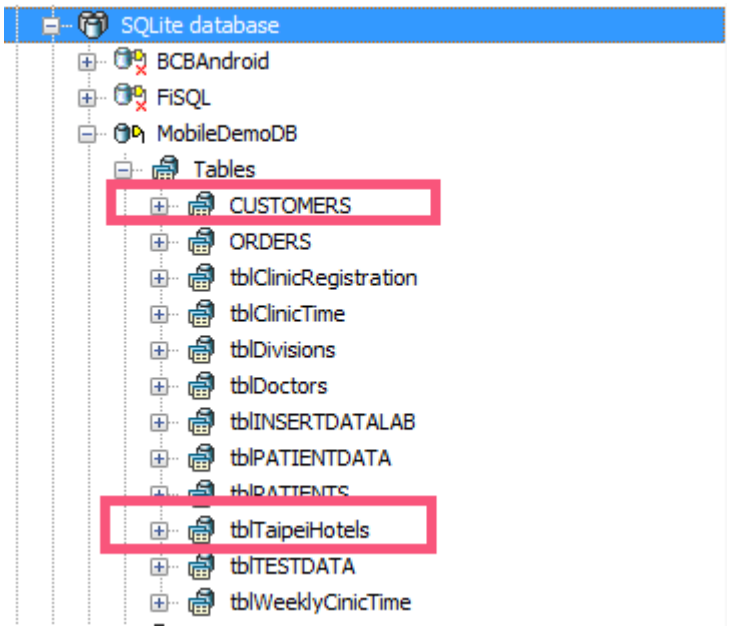
```
SELECT * FROM tblTaipeiHotels;
select * from   CUSTOMERS;
```

在 BDE/dbExpress 中我們需要使用 2 個 Query 元件並執行 2 次以取得 tblTaipeiHotels 和 CUSTOMERS 這 2 個資料表的資料。不過在 FireDAC 中我們可以利用所謂的 Batch SQL 功能一次就執行這 2 個 SQL 命令。

FireDAC 的 Batch SQL 功能可以讓程式師使用一個 TFDQuery 元件一次執行多個 SQL 命令，這樣做有幾個好處，一是可以減少網路的來回次數，二是可以利用資料庫的 Batch 執行功能以增加執行 SQL 的效率，三是可以減少客戶端的資源。

要使用 FireDAC 的 Batch SQL 功能，程式師只需要把 SQL 命令都指定給 TFDQuery 元件的 SQL 特性，再設定 Active 特性值為 True 或是呼叫 Open 方法即可。當 Batch SQL 執行完畢之後後端會傳回多個資料集物件，程式師可以呼叫 TFDQuery 的 NextRecordSet 方法從第 1 個資料集循序存取每一個回傳的資料集物件。

讓我們使用一個簡單的範例來說明，例如在下面的 SQLite 資料庫中假釋我們要存取 CUSTOMERS 和 tblTaipeiHotels 這 2 個資料表中的資料：




那可以使用下面的程式碼，先呼叫 OpenBatchSQL()方法使用 Batch SQL 一次從這 2 個資料表取得資料，再呼叫 CopyToMemoryTable()方法把回傳的資料集物件在客戶端指定給 TFDMemTable，最後再呼叫 DisplayData()顯示資料：



```
void __fastcall TfmMainForm::btnRunBatchClick(TObject *Sender)
{
  OpenBatchSQL();
  CopyToMemoryTable();
  DisplayData();
}
```

OpenBatchSQL() 方法先把執行 Batch SQL 的 TFDQuery 元件的 FetchOptions.AutoClose 特性值設定為 False，否則 TFDQuery 在內定上接到回傳的第 1 個資料集物件之後就會捨棄其他回傳的資料集物件。

接著把 2 個 SQL 命令指定給 TFDQuery 的 SQL 特值，執行 2 個 SQL 命令，最後呼叫 FetchAll()方法一次把 2 個資料表中的資料都存取回客戶端：

```
void TfmMainForm::OpenBatchSQL()
{
  qryGeneral->FetchOptions->AutoClose = false;
```

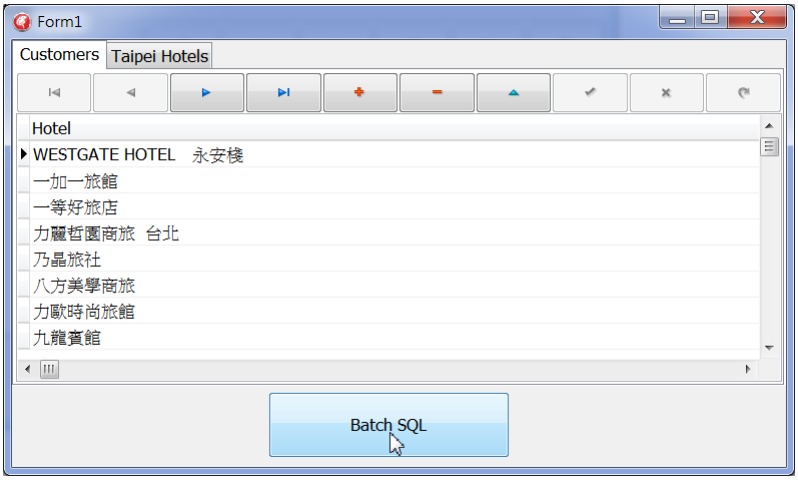
```
  qryGeneral->SQL->Text = "SELECT * FROM tblTaipeiHotels; select *
from CUSTOMERS";
  qryGeneral->Active = true;
  qryGeneral->FetchAll();
}
```

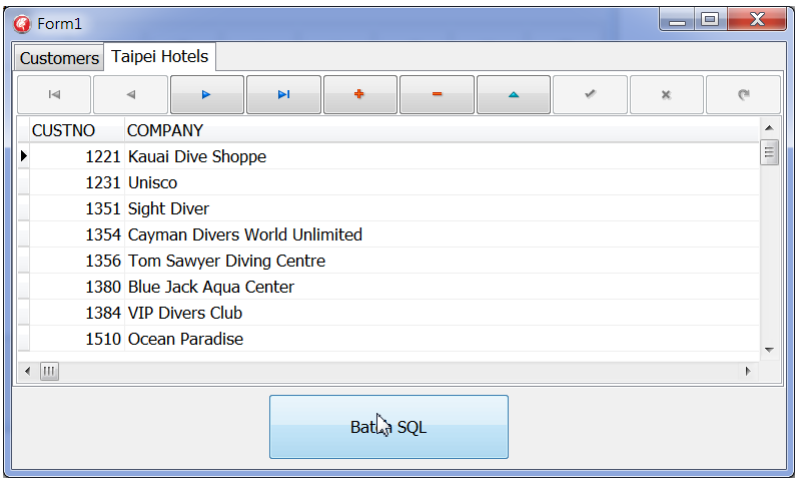
CopyToMemoryTable 方法先把第 1 個回傳的資料集物件中的資料指定給 fdmtCustomers 這個 TFDMemTable 元件，呼叫 TFDQuery 元件的 NextRecordSet() 方法取得下一個回傳的資料集物件之後再指定給 fdmtTaipeiHotels 個 TFDMemTable 元件：

```
void TfmMainForm::CopyToMemoryTable()
{
  fdmtCustomers->Data = qryGeneral->Data;
  qryGeneral->NextRecordSet();
  fdmtTaipeiHotels->Data = qryGeneral->Data;
}
```



最後我們只需要把 fdmtCustomers 和 fdmtTaipeiHotels 這 2 個 TFDMemTable 元件連結到資料感知元件即可把資料顯示出來：






使用 Batch SQL 功能一個網路來回就可以存取到多個資料表的資料，不過並不是每個資料庫都支援 Batch SQL 功能，下面的表格列出了支援 Batch SQL 功能的資料表：

| 資料庫 | Batch SQL |
|---|---|
| IBM DB2 | ✓ |
| Firebird | ✓ |
| Informix | ✓ |
| Microsoft SQL Server | ✓ |
| MySQL | ✓ |
| Oracle | ✓ |
| PostgreSQL | ✓ |
| SQLite | ✓ |
| SQL Anywhere | ✓ |

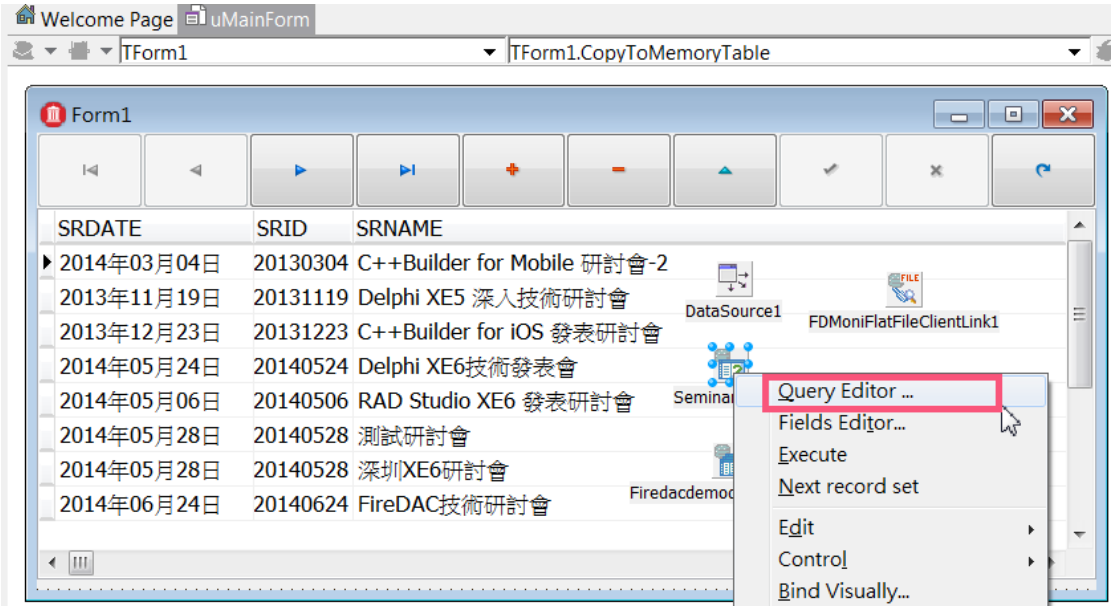


## 5-2 控制資料的顯示和更新

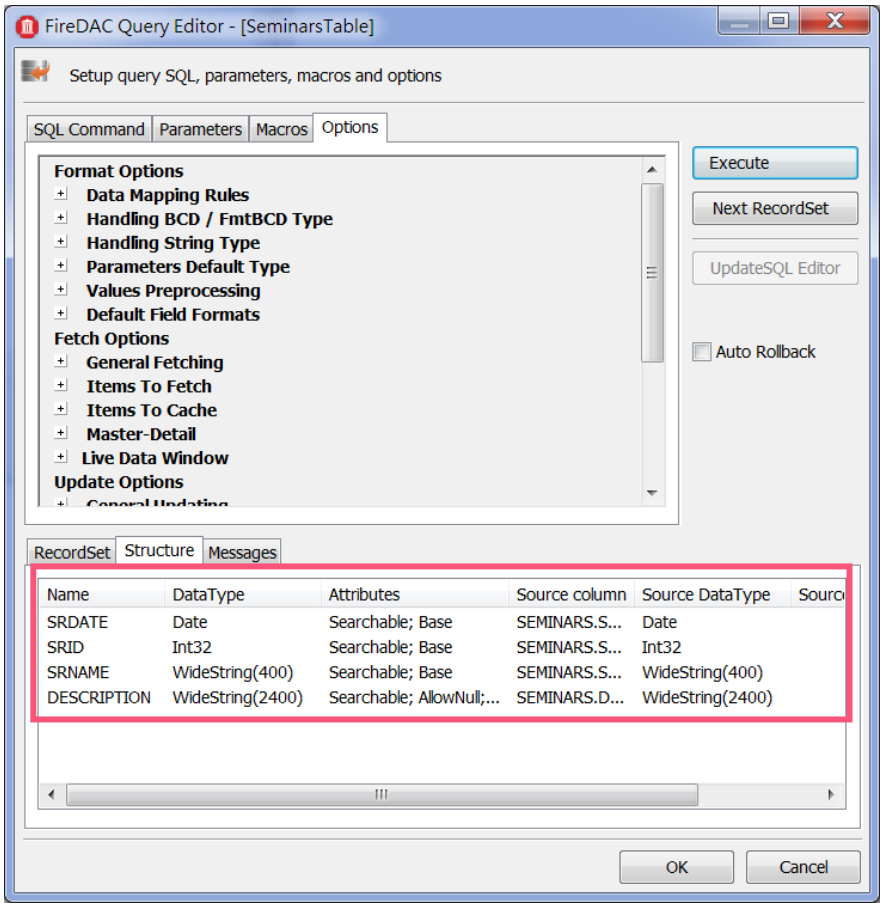
FireDAC 在資料的轉換，對映和顯示都有非常良好的彈性和控制力，能夠讓程式師對進行處理的資料有更大的控制力。在本小節中我們將這些功能進行說明。

在使用 TFDQuery 元件存取後端資料時 FireDAC 允許程式師對於如何處理資料有更進一步的控制，顯示資料來源的結構元資料，如何顯示資料，如何對映資料，一直到如何存取資料和更新資料都可以進生深入的設定和控制。

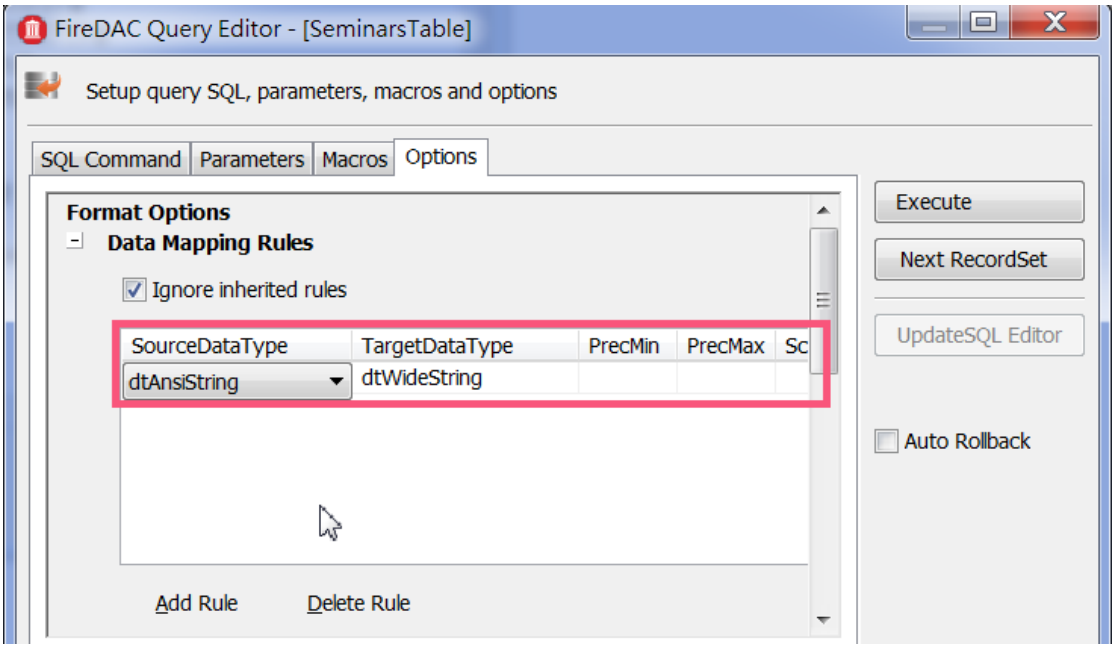
程式師可以藉由右擊 TFDQuery 元件再點選 Query Editor...選項啟動元件編輯器，



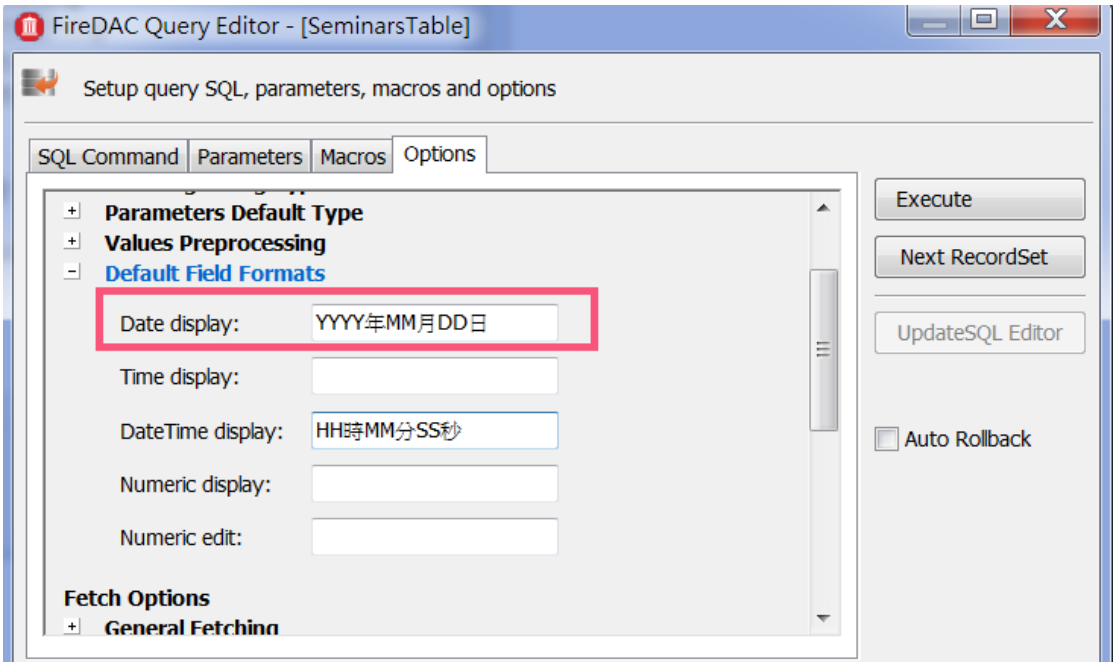

在 TFDQuery 元件編輯器的 Options 頁次可以看到 TFDQuery 元件中的資料集物件的結構元資料：



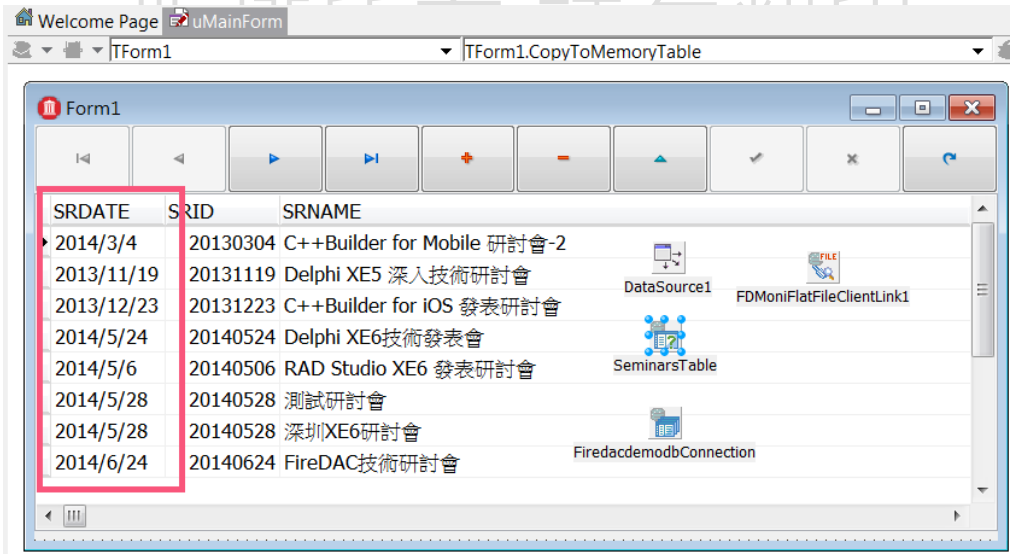
在 Options 頁次中有許多重要的選項可設定，例如在 Format Options 中可以設定資料對映的規則以及資料顯示的內定格式等。例如假設讀者有一個資料表中的一個欄位的資料當初是定義為 AnsiString，現在想以 WideString 顯示和處理，那麼就可以在 Data Mapping Rules 中定義 AnsiString 的資料要對映為 WideString。下圖就是先勾選忽視內定的繼承規則，並定期來源是 AnsiString 的資料要對映成 WideString：

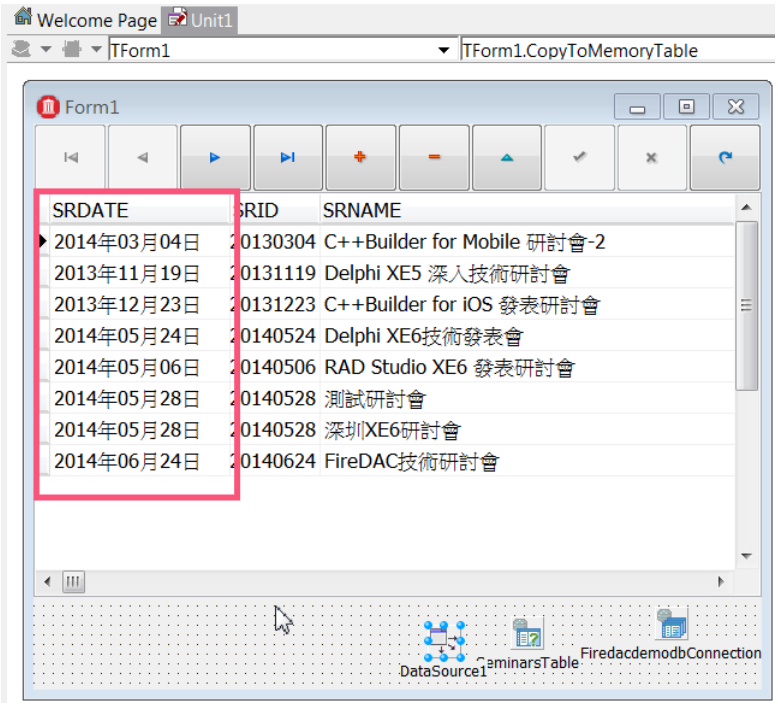



此外也可以在 Options 頁次定義資料的內定顯示格式，例如下面就是在 Options 頁次中定期日期資料要以中文的"年月日"的格式來顯示，時間資料則以"時分秒" 的格式來顯示：

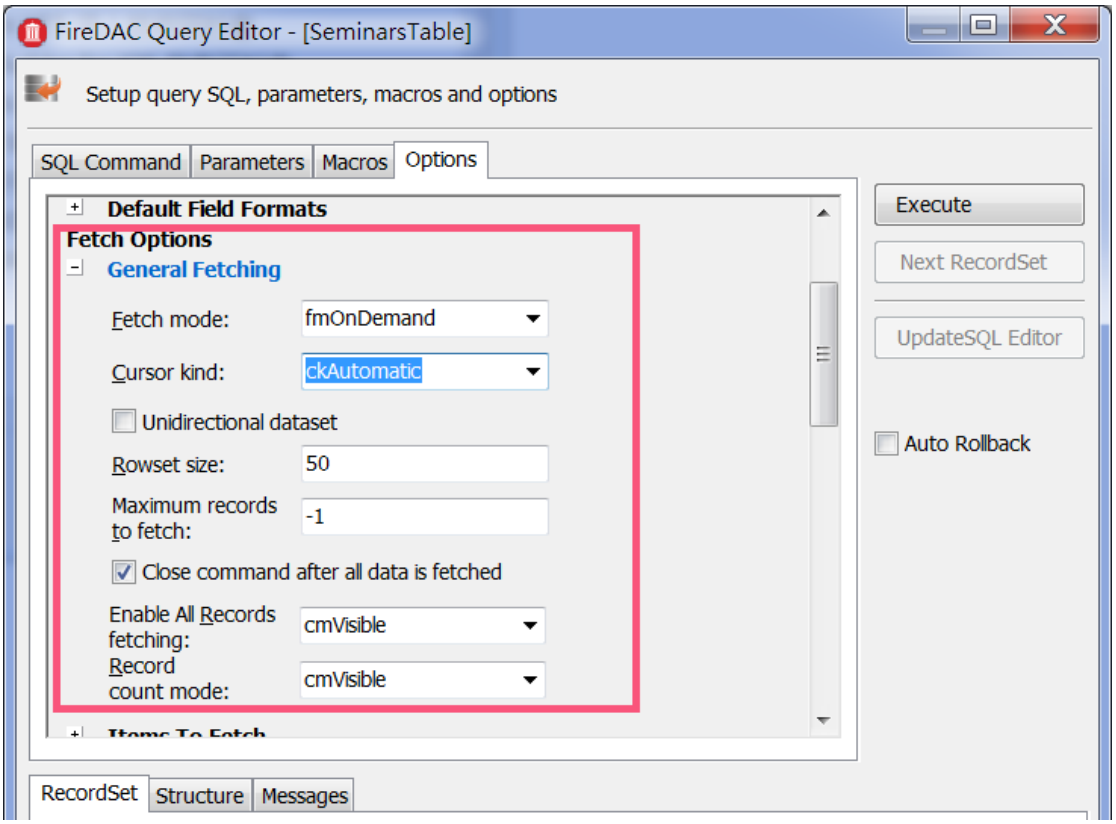



如上在 Options 頁次定義好之後就可以看到原本 SRDATE 欄位的資料格式改變成我們定義的"年月日"格式了：






在 Options 頁次中最重要的應該是 Fetch Options 和 Posting Changes 了，因為這 2 者的設定都和執行效率有很大的關係。下圖說明了 FireDAC 如何存取資料，例如如果應用程式只是依序讀取資料，那設定下圖中的 Unidirectional dataset 選項就可以增加執行速度。Rowset size 特性值控制了 FireDAC 一次從後端資料來源存取的資料筆數，它的內定值 50 代表一次存取 50 筆，因此如果後端資料表中有 1000 筆資料那就要用 20 次的網路來回取得所有資料，因此 Rowset 愈大網路來回次數就愈少。不過設定太大的 Rowset 特性值會讓每一次取取的時間加長，因此程式師應該根據實際的執行環境設定 Rowset 特性值，例如在 PC 環境中也許應該加大 Rowset 特性值，在移動平台則應該稍微減小 Rowset 特性值。在 Options 頁次中的 Rowset size 就是 TFDQuery 的 FetchOptions.RowsetSize 特性值。



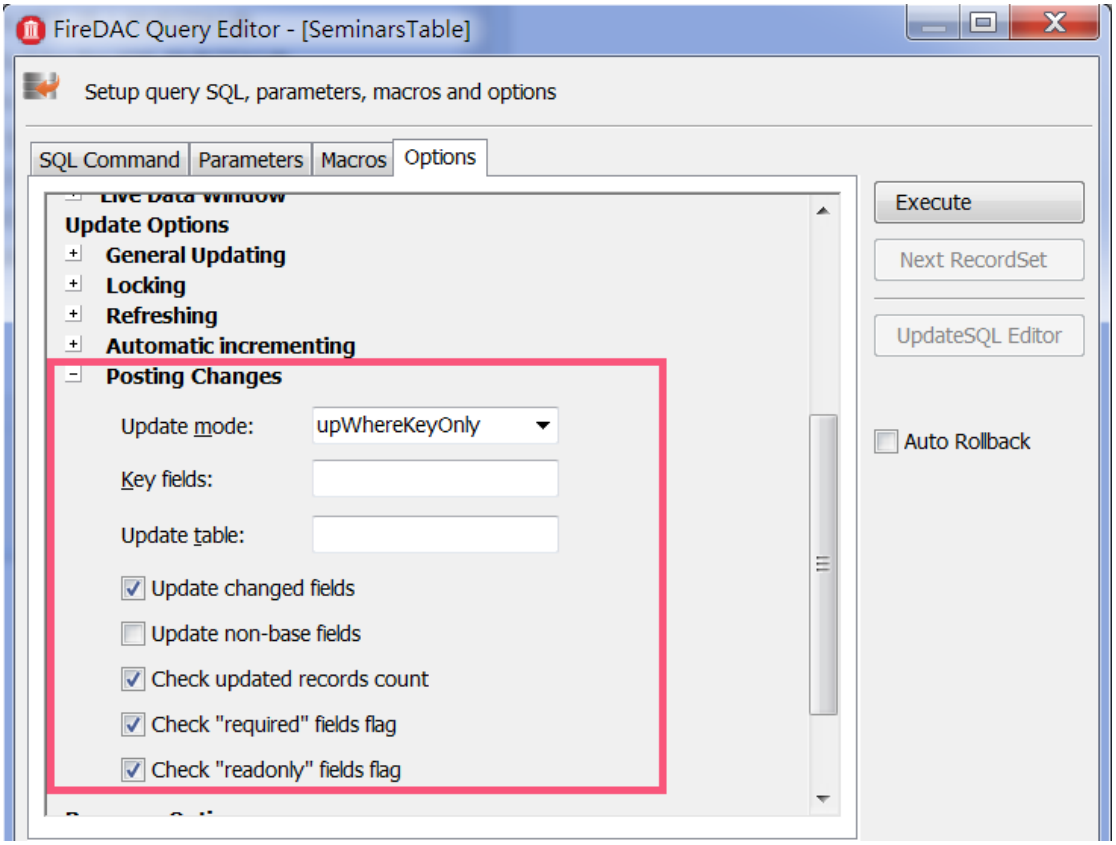



但 FireDAC 一次存取多少資料筆數也受到 Fetch Mode 的影響，Options 頁次中的 Fetch Mode 是就是 TFDQuery 的 FetchOptions.Mode 特性值，它的功能說明如下：

| FetchOptions.Mode | 說明 |
|---|---|
| fmOnDemand | 由 FetchOptions.RowsetSize 特性值自動控制存取資料 |
| fmAll | 一次就取得所有的資料，等於呼叫 TFDQuery 的 FetchAll 方法 |
| fmManual | 由程式師自己呼叫 TFDQuery 的 FetchNext 或 FetchAll 方法取得資料 |
| fmExactRecsMax | 一次取得 FetchOptions.RecsMax 筆的資料，如果取得的資料筆數和 FetchOptions.RecsMax 的特性值不同就會產生例外錯誤。FetchOptions.RecsMax 的內定值是-1，就和 fmAll 是一樣的意思 |

當 FireDAC 拇把資料更新回後端時，如何找到後端要更新的資料再予以更新就是由 Options 頁次中的 Posting Changes 選項來控制。在一般的應用中當使用者在前端異動了資料並實際更新回後端時，FireDAC 會使用被異動資料的鍵值在後端找到這筆資料之

後再對它進行更新，這個規則就是由 TFDQuery 的 UpdateOptions.UpdateMode 控制，也就是 Options 頁次中的 Update mode：

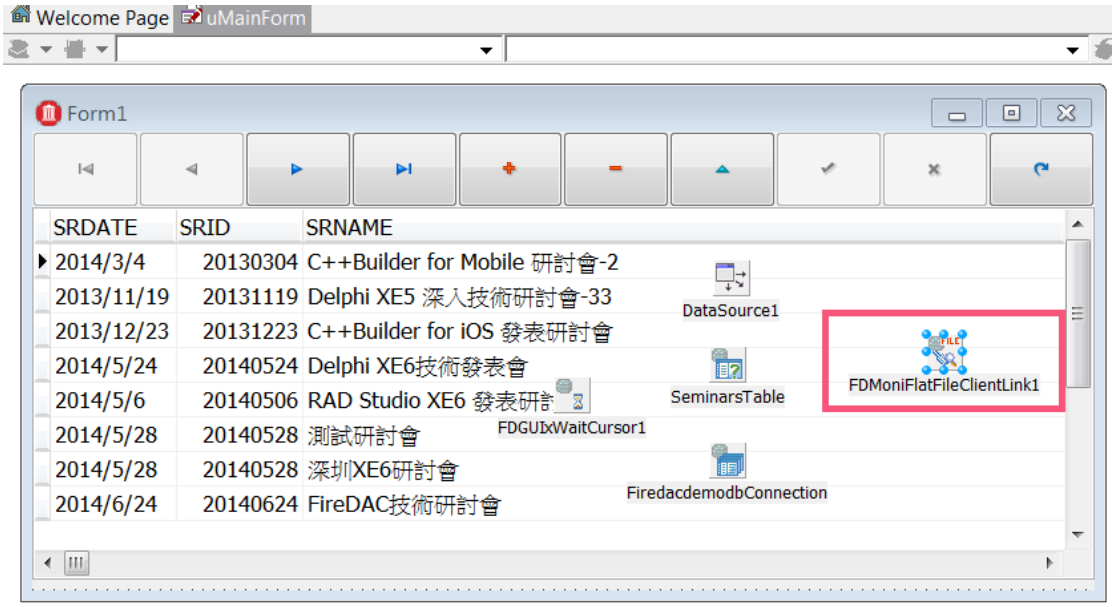



UpdateOptions.UpdateMode 可以有 3 個設定值，說明如下：

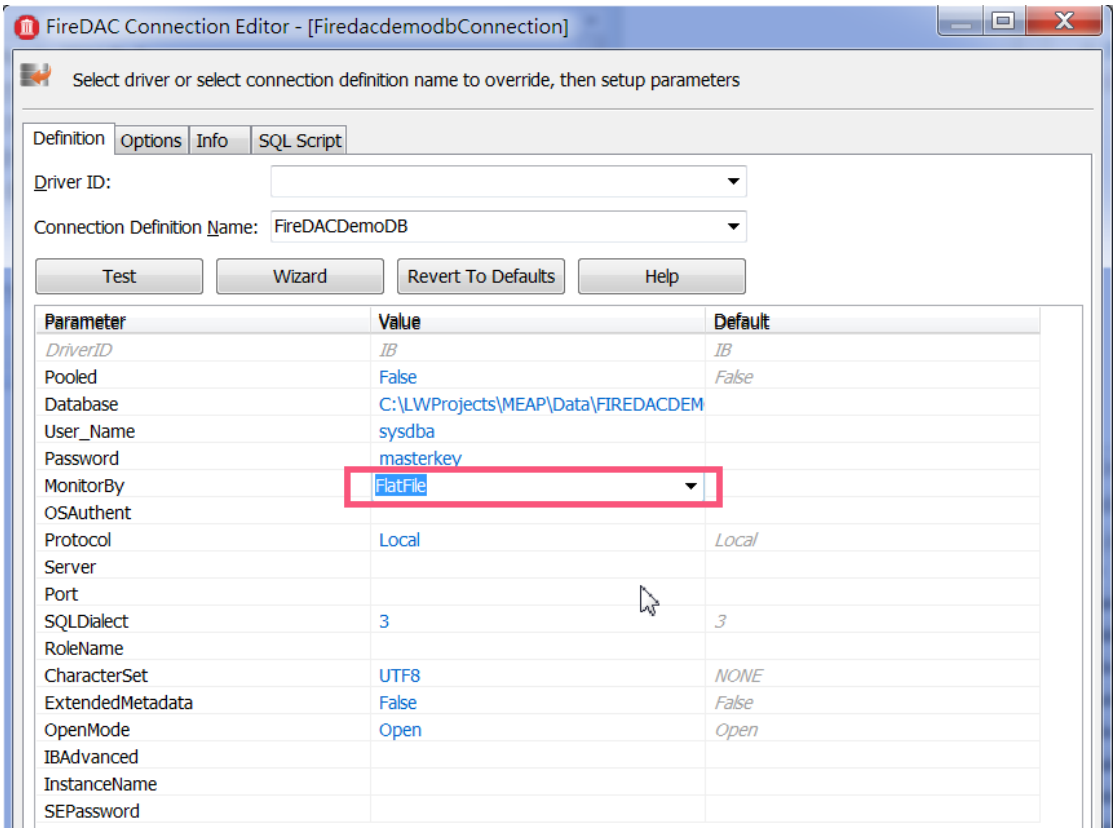
| 特性值 | 說明 |
|---|---|
| upWhereAll | 用所有欄位的舊值來搜尋要更新的後端資料 |
| upWhereChanged | 用更新資料前的鍵值和更新欄位的舊值來搜尋要更新的後端資料 |
| upWhereKeyOnly | 用只更新資料前的鍵值來搜尋要更新的後端資料 |

upWhereKeyOnly 是 UpdateOptions.UpdateMode 的內定值，讓我們使用一個簡單的範例來說明 UpdateOptions.UpdateMode 設定對於更新資料的影響。

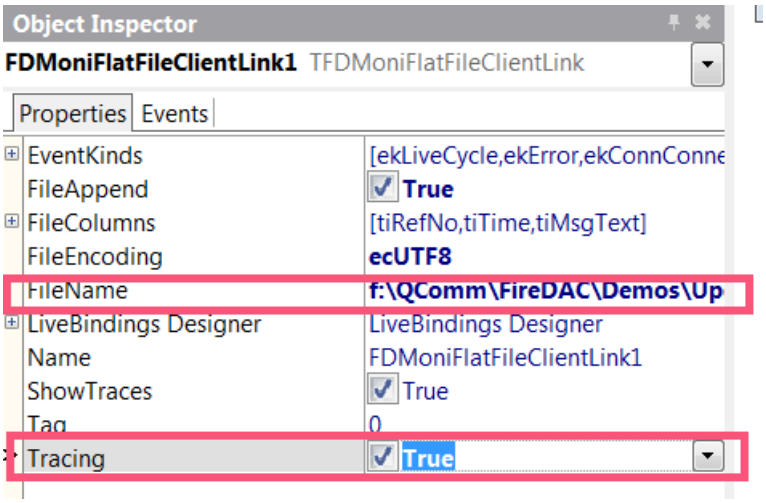
下面是 pUpdateModeDemo.dproj 專案，定使用 TFDQuery 存取 SEMINARS 資料表並且使用了 TFDMoniFlatFileClientLink 元件把前端 FireDAC 程式如何和後端資料庫互動的命令都記錄在一個文字檔中好人讓我們檢查 UpdateOptions.UpdateMode 如何影響資料更新的行為：



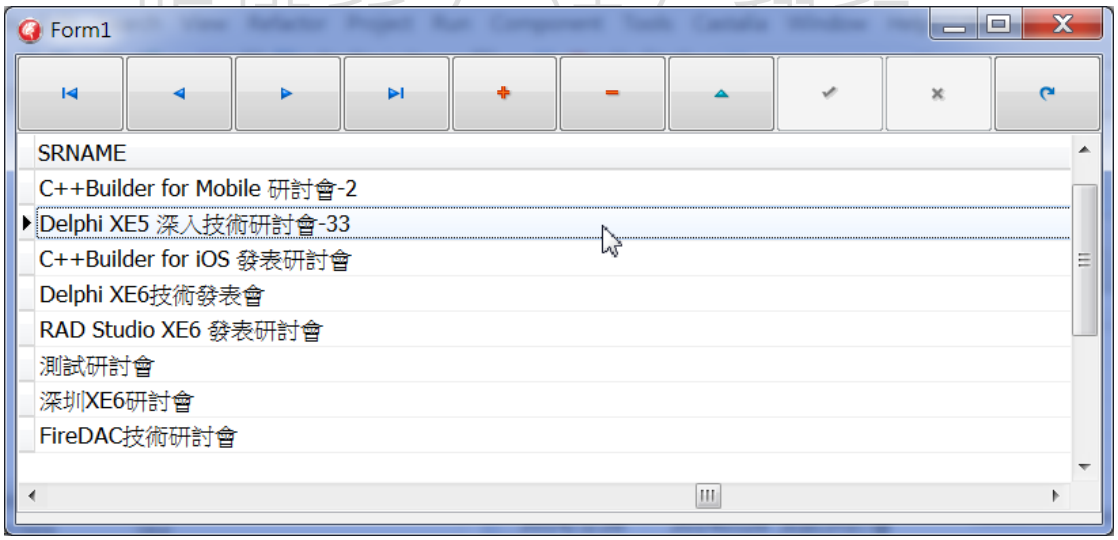

加入 TFDMoniFlatFileClientLink 元件後要在 TFDConnection 元件編輯器中設定 MonitorBy 為"FlatFile"：



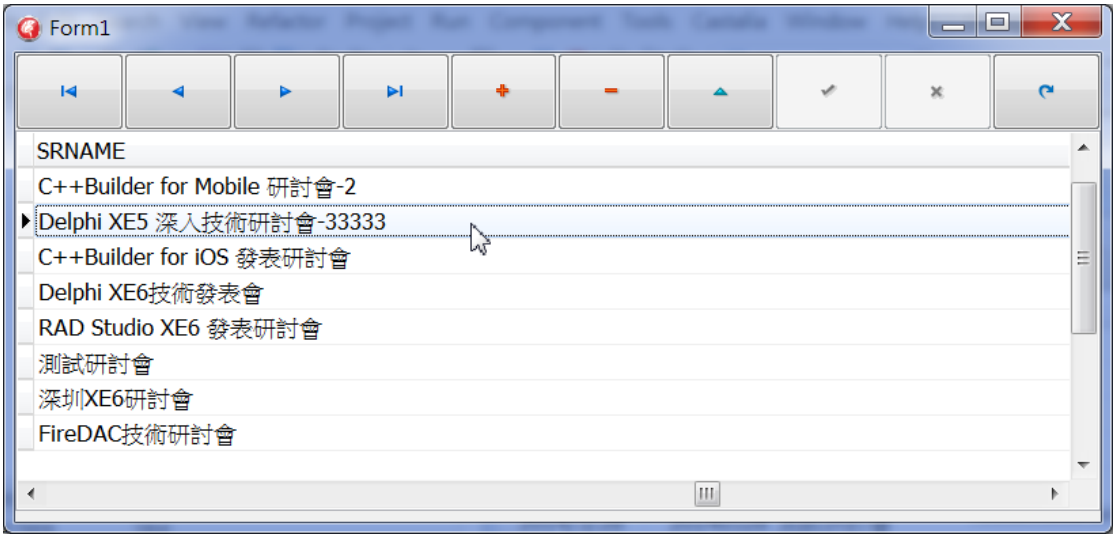
然後設定 TFDMoniFlatFileClientLink 元件的 FileName 特性值為一個文字檔再設定它的 Tracing 特性值為 True：




執行範例程式並修改一筆資料然後更新回資料庫，例如修改”C++Builder XE5 深入技術研討會-33”




為”C++Builder XE5 深入技術研討會-33333”：



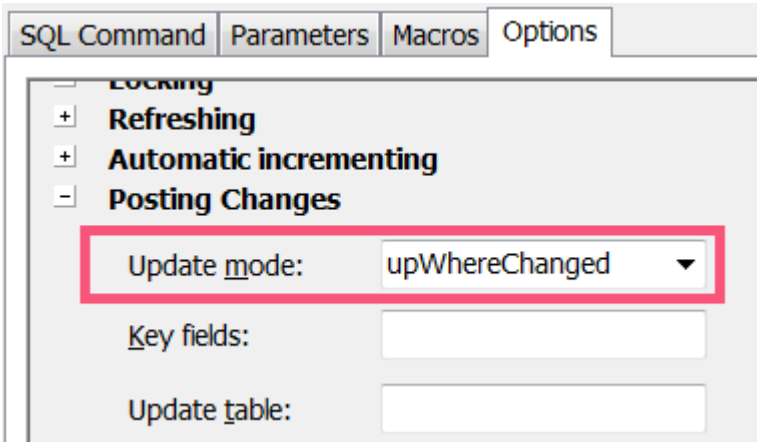

由於 upWhereKeyOnly 是 UpdateOptions.UpdateMode 的內定值，因此如果現在檢查 TFDMoniFlatFileClientLink 元件寫入的文字檔，可以看到 FireDAC 使用了如下的 Update 命令來尋找舊的資料並更新：

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



在 where 中我們可以看到 FireDAC 只使用鍵值欄位” SRNAME”來尋找舊的資料。

但如果我們設定 UpdateOptions.UpdateMode 為 upWhereChanged：

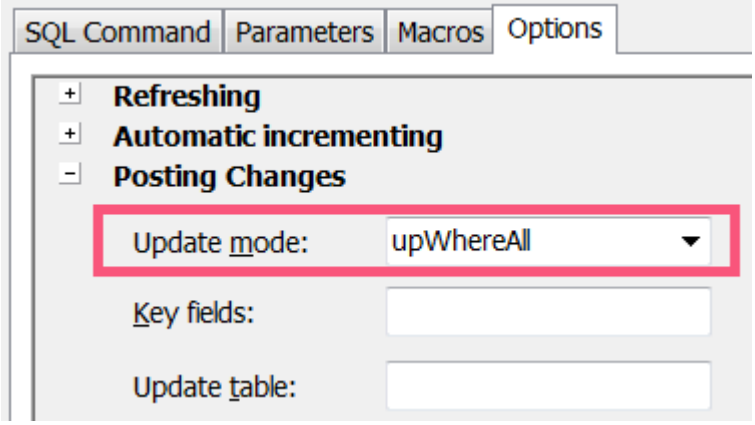



那麼可以看到 FireDAC 使用了如下的 Update 命令來尋找舊的資料並更新：

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

在 where 中我們可以看到 FireDAC 只使用了鍵值欄位”SRNAME”和 DESCRIPTION 欄位來尋找舊的資料，這是因為我們同時修改了 DESCRIPTION 欄位中的數值。

最後如果我們設定 UpdateOptions.UpdateMode 為 upWhereAll：





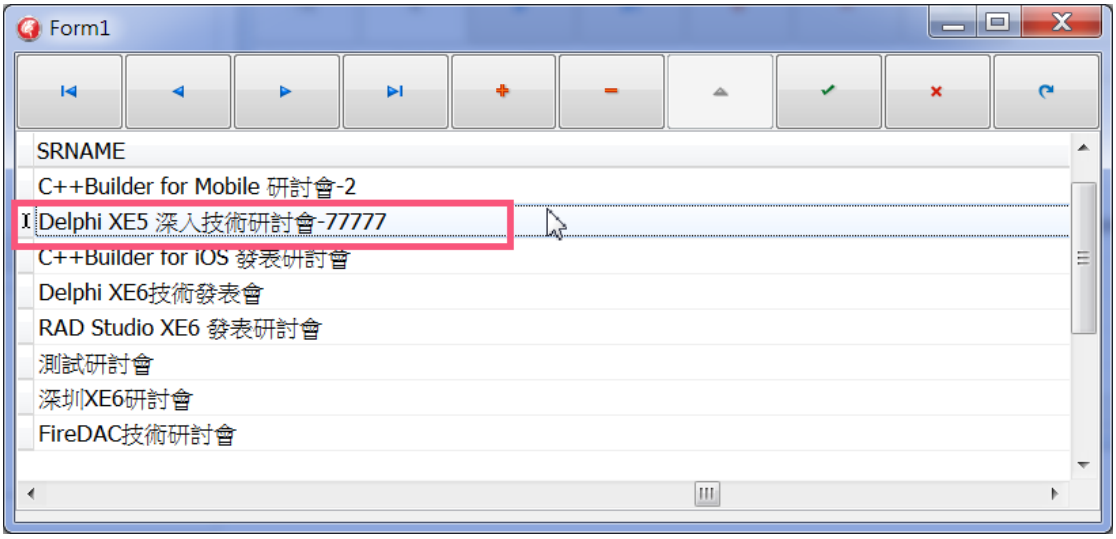

那麼可以看到 FireDAC 使用了如下的 Update 命令來尋找舊的資料並更新：

```
"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"
```

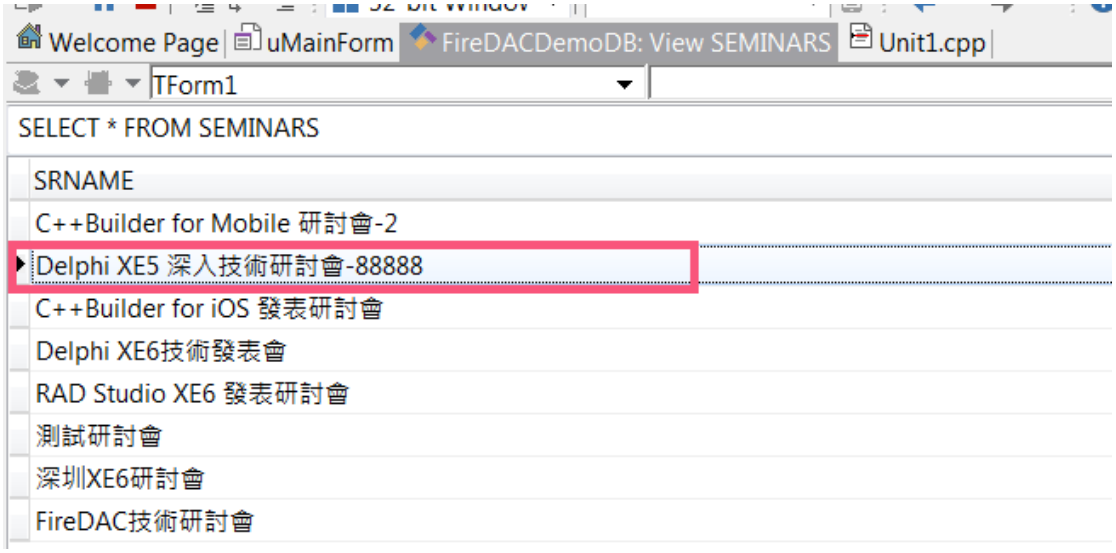
在 where 中我們可以看到 FireDAC 使用了所有未更新資料前的舊欄位值來尋找舊的資料。

在多人使用的環境中 UpdateOptions.UpdateMode 的設定會決定資料是否能成功更新回後端，這是因為剛才說明 UpdateOptions.UpdateMode 控制了在資料更新回資料表之前如何先找到要被異動的資料。讓我們使用一個小小的範例說明讀者就會瞭解了。

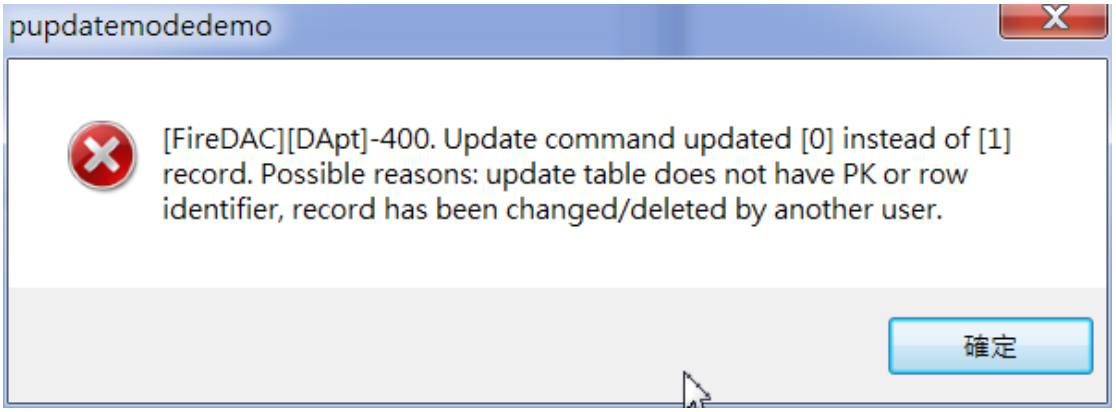
再次執行前的範例程式，讓我們如同下圖修改”C++Builder XE5 深入技術研討會”這筆資料為”C++Builder XE5 深入技術研討會-77777”：




但在按下 Post 按鈕把資料更新回資料表之前，先使用 IDE 中的 Data Explorer(或是任何可修改資料的工具)把"C++Builder XE5 深入技術研討會"這筆資料改成"C++Builder XE5 深入技術研討會-88888"然後立即更新回資料表：




接著再回到範例程式點選更新按鈕後就會看到範例程式出現如下的錯誤：




為什麼會出現錯誤呢？很簡單，因為範例程式在更新資料回資料表時會先以"C++Builder XE5 深入技術研討會"這個鍵值到資料表中搜尋，但由於這筆資料的鍵值已經被另一客戶端 Data Explorer 改成"C++Builder XE5 深入技術研討會-88888"，因此範例程式找不到這筆要被更新的資料，因此就顯示此筆資料已經被其他使用者更改/刪除的錯誤了。

因此在多人使用的環境中 UpdateOptions.UpdateMode 的設定會決定資料是否能成功更新回後端。



## 5-3 資料轉換

從 TOKYO 起 FireDAC 提供了一組新的 ETL(Extract-Transform-Load)元件讓程開發人員可以使用來在不同的資料來源中轉換資料，例如如果開發人員想把資料轉成文字的形式，或是把資料從 SQLite 轉到 InterBase 等，這組元件稱為 FireDAC ETL。

FireDAC ETL 基本上使用了 Reader/Writer 設計樣例，FireDAC ETL 提供了 3 組 Reader/Writer，在資料/文字，資料集/資料集和 SQL 資料/SQL 資料之間轉換。每一個功能都提供一個 FireDAC 元件讓開發人員使用。下面的表格說明了這些元件：

| | |
|---|---|
| TFDBatchMove | 提供實際轉換功能 |
| TFDBatchMoveTextReader | 把資料以文字形式讀出 |
| TFDBatchMoveTextWriter | 把資料以文字形式寫出 |

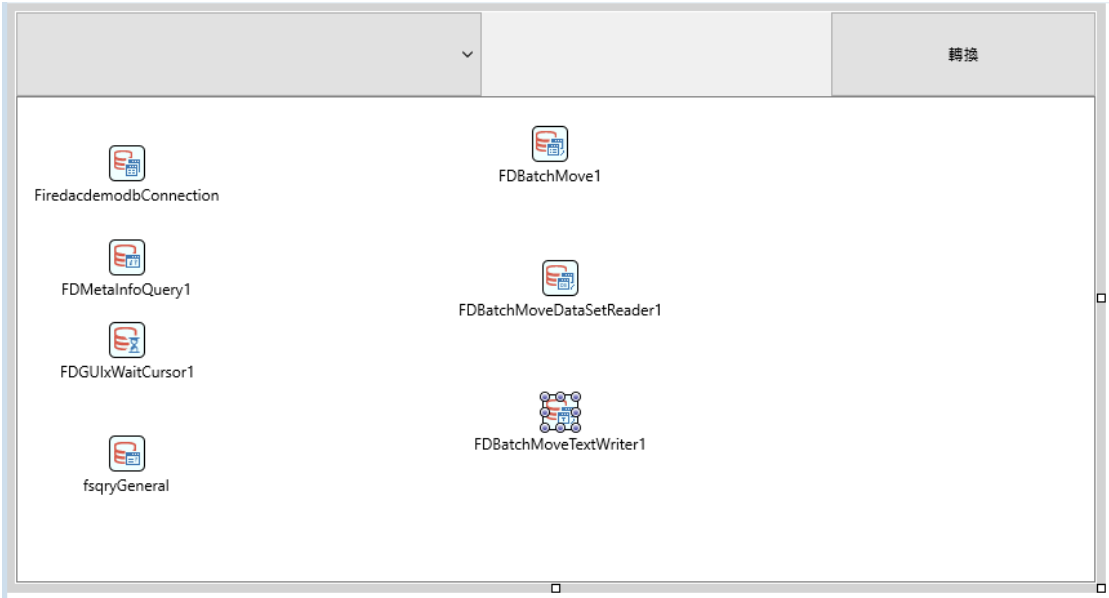
| | |
|---|---|
| TFDBatchMoveDataSetReader | 把資料從資料集中讀出 |
| TFDBatchMoveDataSetWriter | 把資料從資料集中寫出 |
| TFDBatchMoveSQLReader | 把資料從 SQL 命令執行的結果中讀出 |
| TFDBatchMoveSQLWriter | 把資料從 SQL 命令執行的結果中寫出 |
| TFDBatchMoveJSONWriter | 以 JSON 的格式寫出資料 |

FireDAC ETL 元件組使用上很簡單，只需要連結到 TFDConnection 元件，使用 TFDBatchMoveXXXReader 元件讀出資料，再使用 TFDBatchMoveXXXWriter 寫出資料，最後呼叫 TFDBatchMove 元件的 Execute 方法即可，下面的 2 小節分別簡單的說明如何在不同的資料來源中轉換資料。

## 5-3-1 資料換文字格式



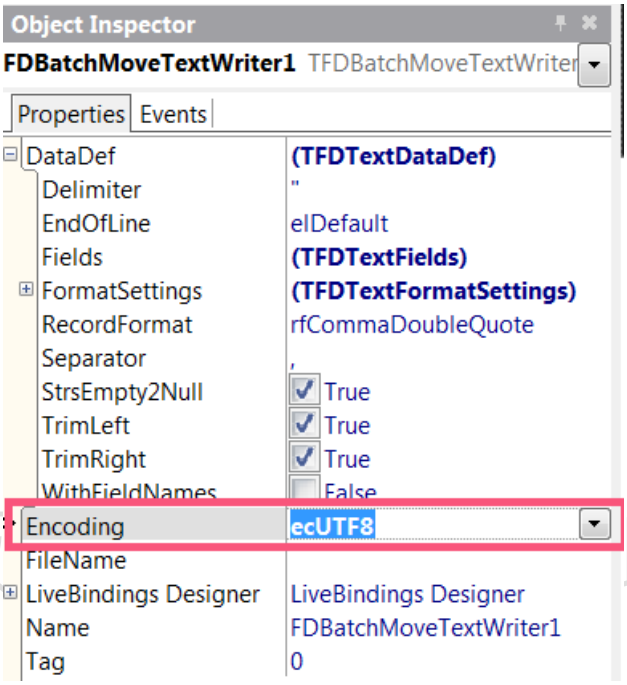
建立一個 Multi-Device Application 專案，在主表單中加入如下的元件，這個範例將可以把範例資料庫 FIREDACDEMODB.GDB 中的任何資料表中的資料轉成文字的形象：



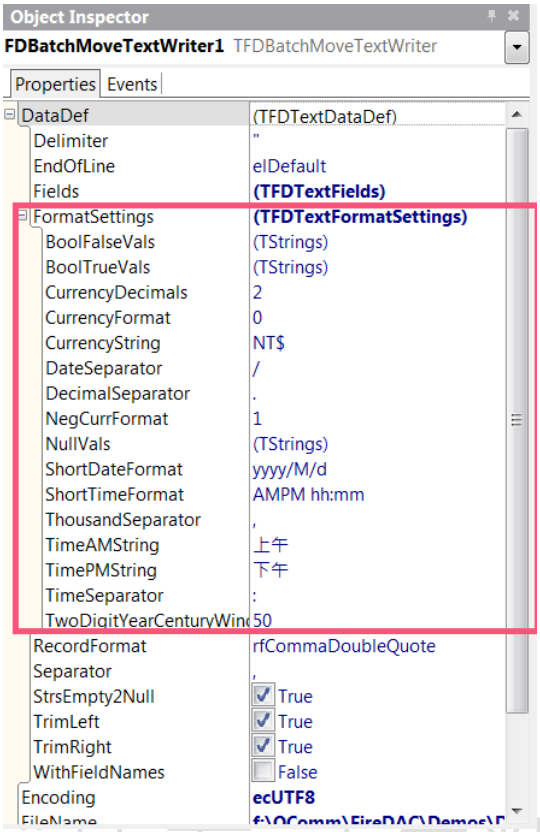

在主表單中使用了 TFDBatchMove，TFDBatchMoveDataSetReader 和 TFDBatchMoveTextReader 這 3 個 ETL 元件，因為我們要把主表單中用 fsqryGeneral

元件取得的資料轉成文字，而 fsqryGeneral 是一個資料集元件因此使用 TFDBatchMoveDataSetReader 從它讀取資料再使用 TFDBatchMoveTextReader 元件寫出資料。

TFDBatchMoveTextReader 元年可以設定使用什麼格式的字串形式寫出資料，例如在物件檢視器中可以設定文字的 Delimiter，Separaor 等：




也可以在 FormatSettings 中進一步設定更多的格式，例如日期格式，布林值格式等等：



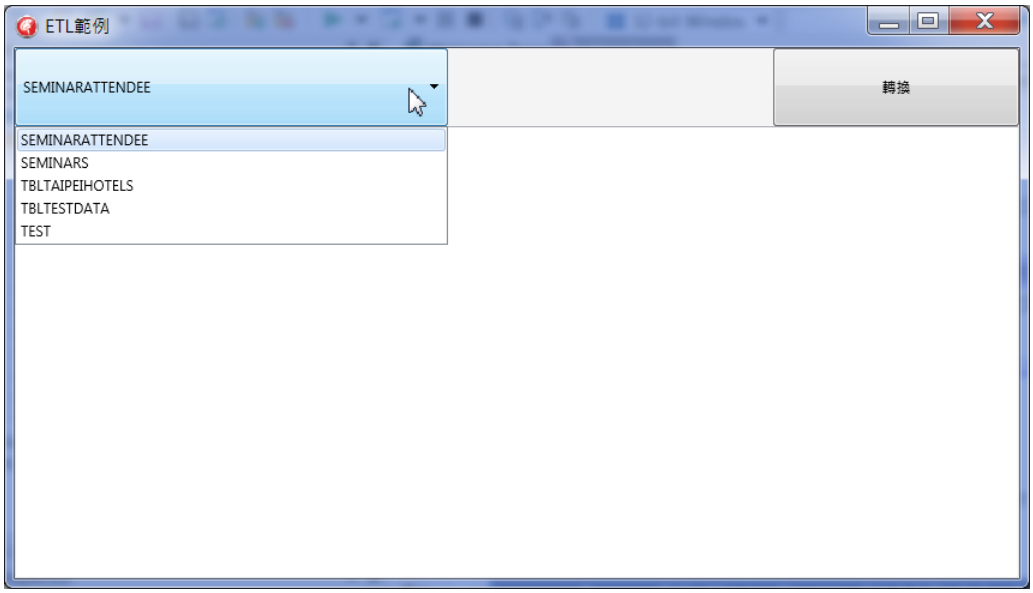



範例應用程式在啟動時使用 TFDMetaInfoQuery 元件取得資料庫中所有資料表的資訊：

```
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;
```

範例應用程式在執行後使用者就可以在左上方的 TComboBox 中選擇要轉換資料的資料表：




接著點選轉換按鈕範例程式就會執行下面的程式碼：

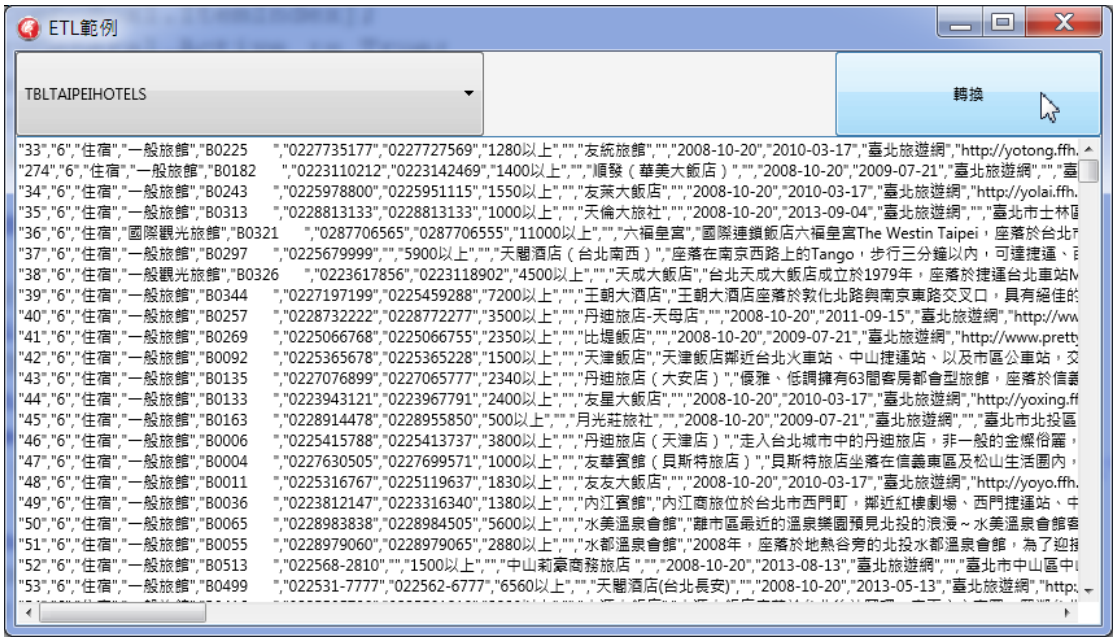
```
001    #include <system.IOUtils.hpp>
002    ..
003
004    void __fastcall TfmMainForm::btnETLClick(TObject *Sender)
005    {
006      if (TFile::Exists(FDBatchMoveTextWriter1->FileName) )
007        TFile::Delete(FDBatchMoveTextWriter1->FileName);
008      fsqryGeneral->Active = false;
009      fsqryGeneral->MacroByName("TABLENAME")->AsRaw =
ComboBox1->Items->operator [](ComboBox1->ItemIndex);
010      fsqryGeneral->Active = true;
011      FDBatchMove1->Execute();
```

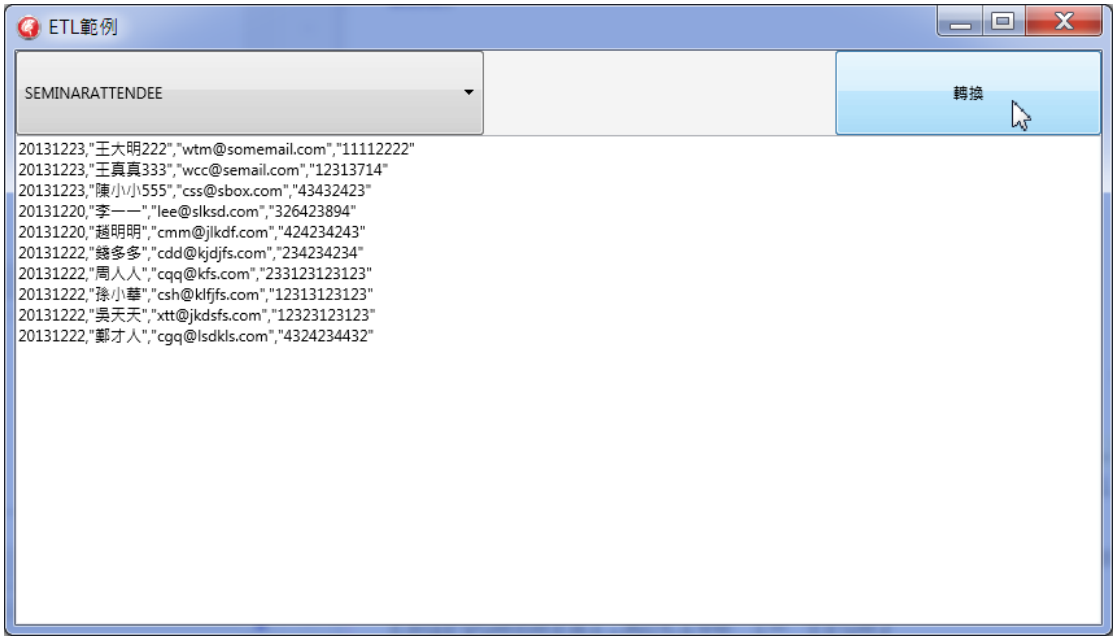
```
012
mmData->Lines->LoadFromFile(FDBatchMoveTextWriter1->FileName);
013    }
```

008~010 使用 FireDAC 的 Macro 功能把使用者選擇的資料表名稱代入 SQL 命令中再執行此 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 元件的 Execute 方法就可以非常簡單的完成資料轉文字的工作了,下面就是範例程式轉換 TBLTAIPEIHOTEL 和 SEMINARATTENDEE 這 2 個資料表的結果：



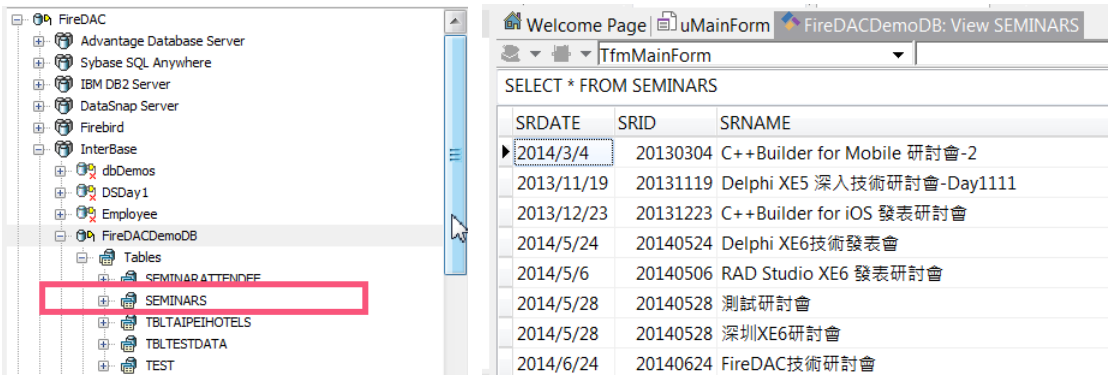



## 5-3-2 在不同資料來源中轉換資料

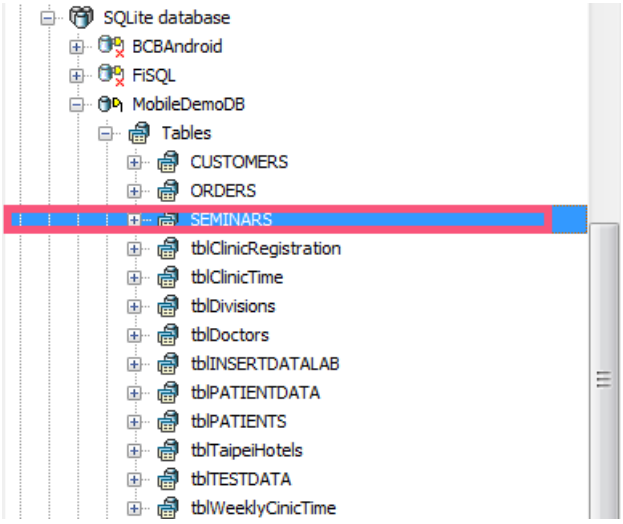
使用 FireDAC ETL 元件組在不同的資料表格中轉換資料非常的容易，程式師只需要使用 2 個 TFDConnection 連結到不同的資料庫，再使用 1 個 TFDQuery 指面來源資料表格，使用另一個 TFDQuery 指面目的地資料表格，最後使用 TFDBacthMove 元件指到這 2 個 TFDQuery 元件再呼叫 TFDBacthMove 的 Execute 方法即可，例如下面是筆者如何使用 FireDAC ETL 元件組把 InterBase 資料庫中的資料自動轉換到 SQLite 資料庫中。



下面的圖形顯示了存在於 InterBase 中的 SEMINARS 資料表中的資料：



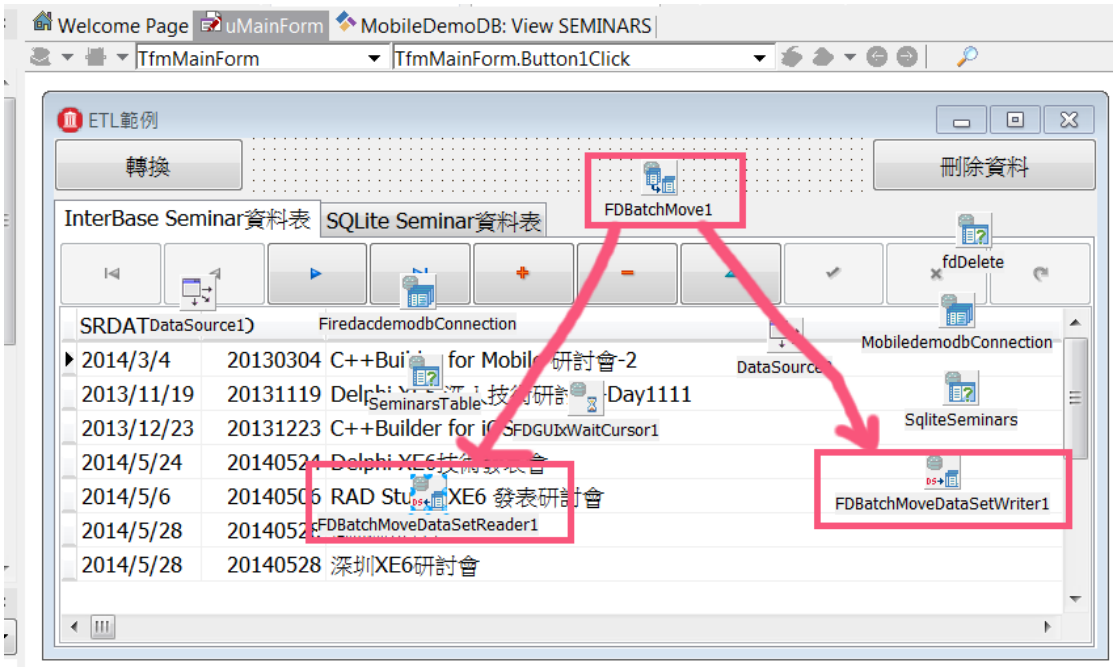

現在我們要使用 FireDAC ETL 元件組把上面的資料轉換到下面的 SQLite 資料表中：



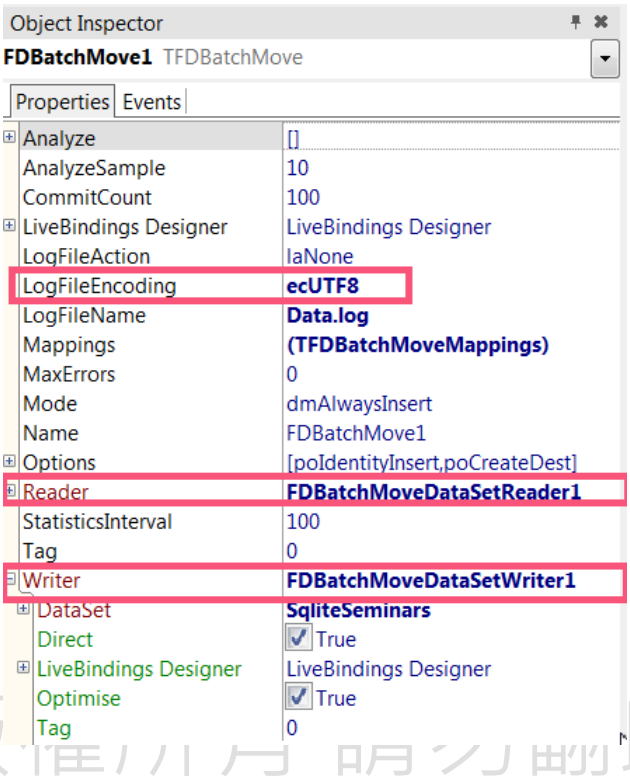

此範例程式使用了 TFDConnection 元件 FiredacdemodbConnection 連結到 InterBase 資料庫，再使用 TFDQuery 元件 SeminarsTable 連結到 InterBase 的 SEMINARS 資料表，再使用 TFDConnection 元件 MobiledemodbConnection 連結到 SQLite 資料庫，再使用 TFDQuery 元件 SqliteSeminars 連結到 InterBase 的 SEMINARS 資料表。



再放入 TFDBatchMoveDataSetReader 元件 FDBatchMoveDataSetReader1 連結到 SeminarsTable，TFDBatchMoveDataSetWriter 元件 FDBatchMoveDataSetWriter1 連結到 SqliteSeminars，最後再放入 TFDBatchMove 元件 FDBatchMove1：

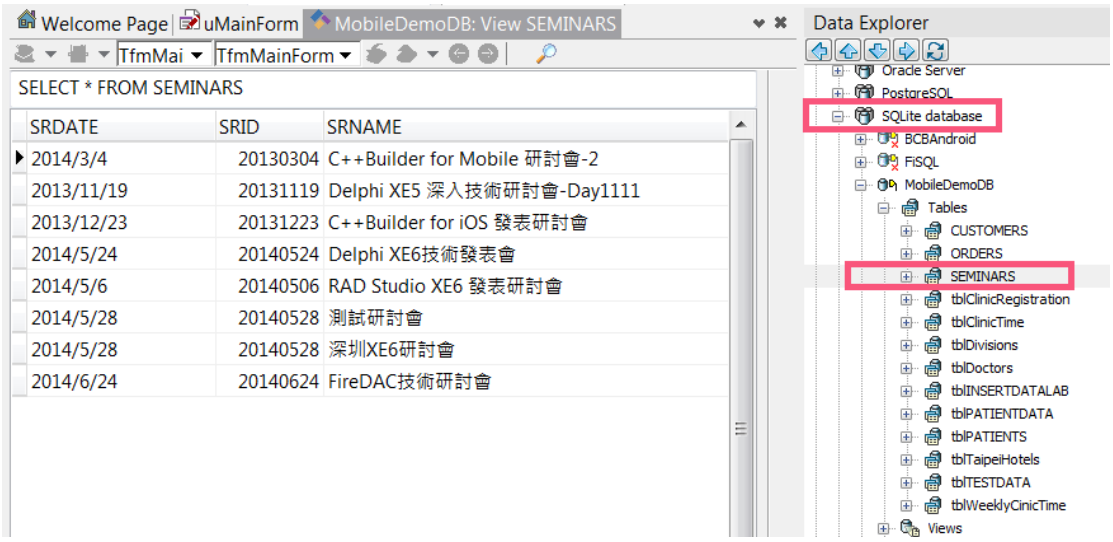


設定 FDBatchMove1 的 LogFileEncoding 為 ecUTF8，Reader 特性值為 FDBatchMoveDataSetReader1，Writer 特性值為 FDBatchMoveDataSetWriter1：






再呼叫 FDBatchMove1 的 Execute 方法後 InterBase 中的資料就立刻的轉換到下面的 SQLite 資料庫的 SEMINARS 資料表中了：




FireDAC ETL 元件組使用上非常的簡單，卻可提供非常方便的資料轉換功能。

# 5-4 處理自動增加值欄位 (Auto-Increment Field)

許多資料庫都支援所謂的自動增加值欄位型態，例如 MS SQL Server 的識別欄位以及 SQLite 的 Autoinc 欄位。這種欄位通常是由後端的資料庫自己維護其數值而不是由程式師撰寫程式碼寫入數值。不過在實際的應用中程式師會需要取得後端資料庫在這種欄位中寫入的數值，例如這種欄位通常都會被定義成鍵值，程式師需要根據此鍵值來搜尋相關的資料或是進行主從資料的處理，那麼要如何使用 FireDAC 處理這種型態的欄位？

比起以前的 BDE/IDAPI 和 dbExpress，FireDAC 提供了非常方便的機製讓程式師來使用和處理自動增加值的欄位。FireDAC 提供了 2 種方式處理自動增加值的欄位：

| 處理模式 | 說明 |
|---|---|
| 自動模式 | 在這種模式中當開啟擁有自動增加值欄位的資料表時，FireDAC 便會試著找出這個自動增加值的欄位並且自動進行適當的特性值設定。 |
| 手動模式 | 如果程式師不想使用自動模式或是 FirDAC 無法辨識出自動增加值的欄位，那麼程式師可以自行設定適當的特性值。 |



讓我們使用一個範例來說明如何讓 FireDAC 處理自動增加值欄位。

下面是一個 SQLite 的範例資料庫，其中的 SRID 欄位就是自動增加值欄位型態：

Create statement

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

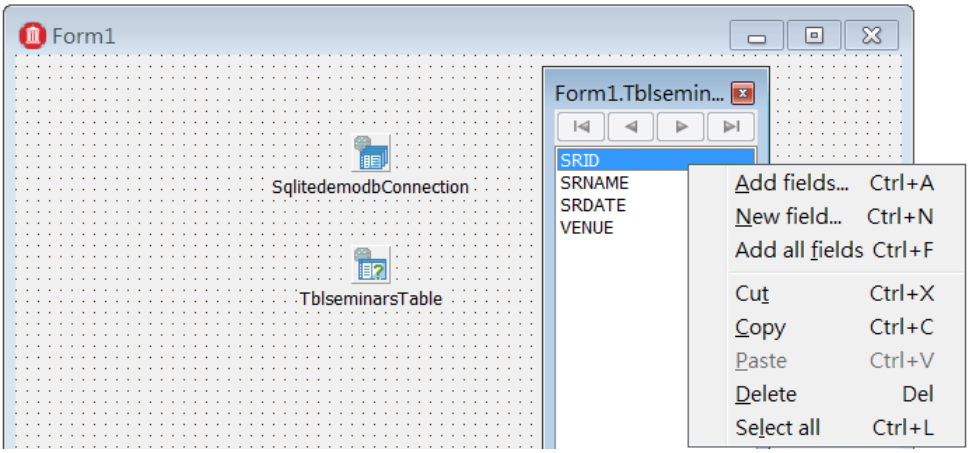
More Info

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

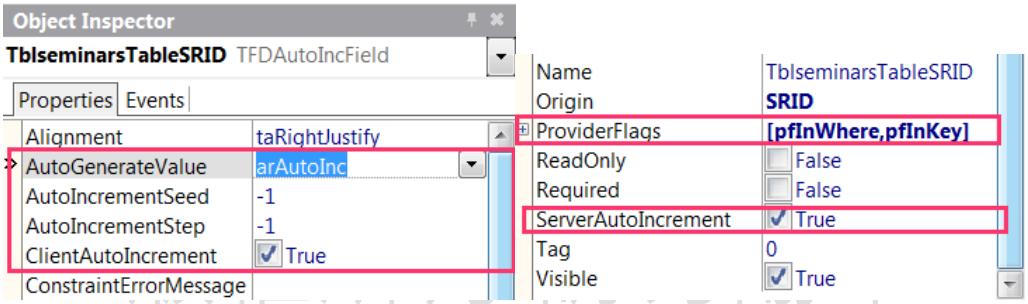
Columns (4)

| 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 |

如果我們使用 FireDAC 連結並開啟上面的 tblSeminars 資料表並且使用 TFDQuery 的元件編輯器點選其中的 SRID 欄位：



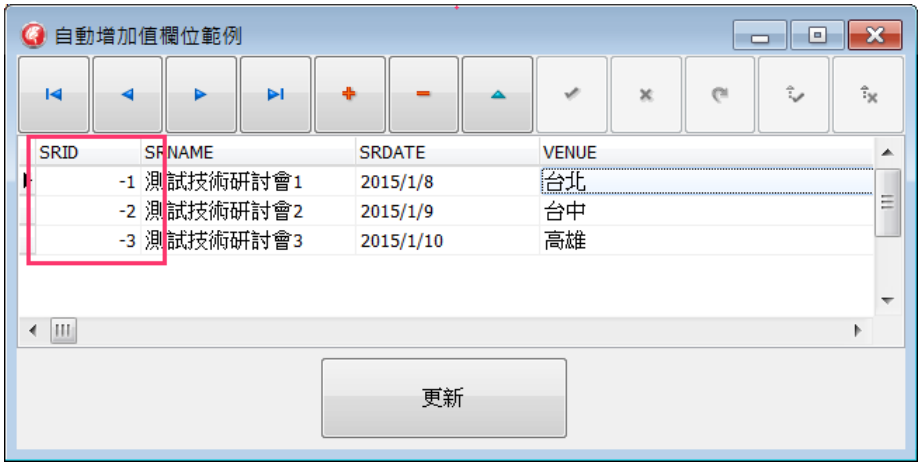

那麼在物件檢視器中就可以看到 FireDAC 自動對 SRID 欄位進行了如下重要的設定：




我們使用下面的表格來說明這些特性值的意義：

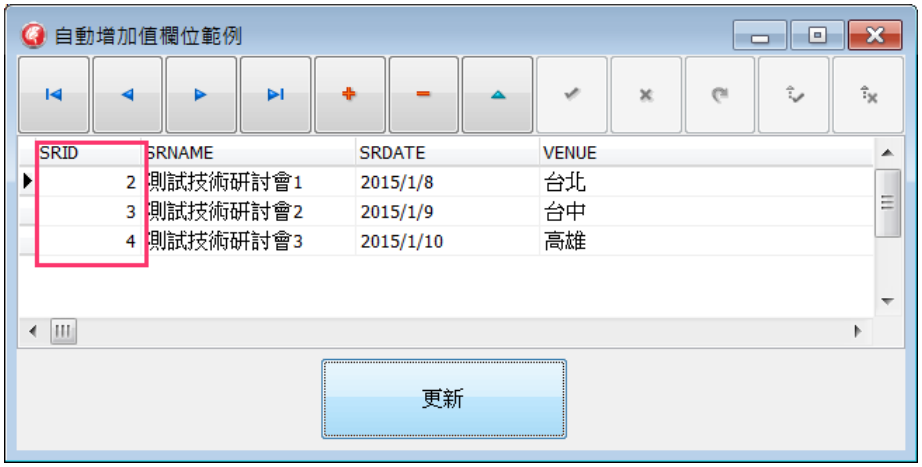
| 處理模式 | 說明 |
|---|---|
| ClientAutoIncrement | 由於自動增加值欄位的數值是由後端的資料庫設定，但在前端 FireDAC 處理自動增加值欄位時仍然暫時需要設定這個欄位值，因此 FireDAC 會設定 ClientAutoIncrement 特性值為 True，如此一來當客戶端新增資料時，會暫時使用 AutoIncrementSeed 和 AutoIncrementStep 這 2 個特性值設定動增加值欄位的暫時數值。例如在上面的圖形中 AutoIncrementSeed 是-1 代表客戶端新增一筆資料時動增加值欄位的數值就設定為 -1，第 2 筆新增的資料就以 AutoIncrementStep 的特性值增加，因此就是-2。 |
| AutoGenerateValue | 設定為 arAutoInc 代表此欄位值是自動增加的 |
| ProviderFlags | 設定為 pfInWhere，如果此自動增加值欄位是鍵值的話就再加入設定 pfInKey |
| ServerAutoIncrement | 代表此欄位的數值是由後端資料庫設定 |

下面的範例顯示了 FireDAC 處理自動增加值欄位的行為，在下面的範例中我們先開啟 Cached Updates 功能以便讓我們可詳細觀察客戶端和後端資料庫如何處理自動增加值欄位。首先我們可看到在客戶端新增資料時自動增加值欄位的暫時數值由 FireDAC 處理，FireDAC 使用苒面說明的 AutoIncrementSeed 和 AutoIncrementStep 這 2 個特性值設定自動增加值欄位的暫時數值，從-1 開始增加：



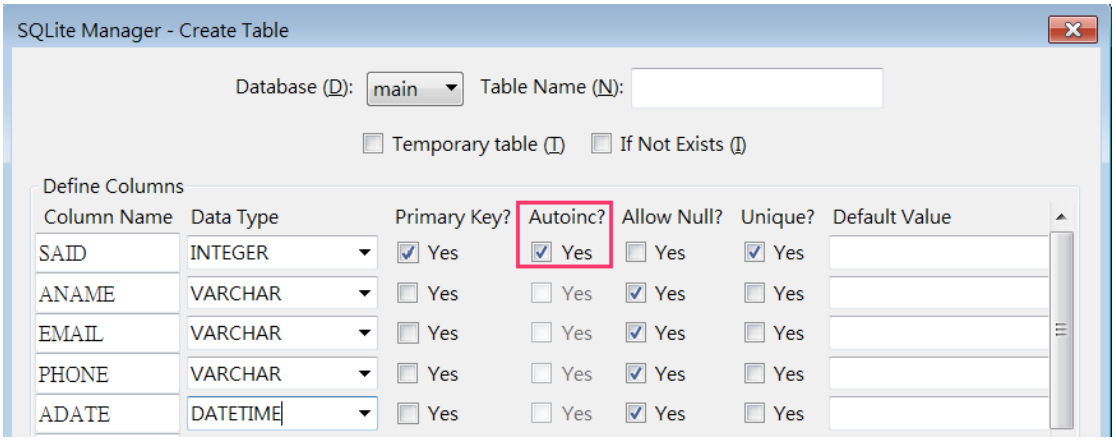

一旦我們呼叫了 ApplyUpdates()方法把資料更新回後端資料庫就可以從下面看到 FireDAC 從後端資料庫取得了自動增加值欄位由後端資料庫真正指定的數值：



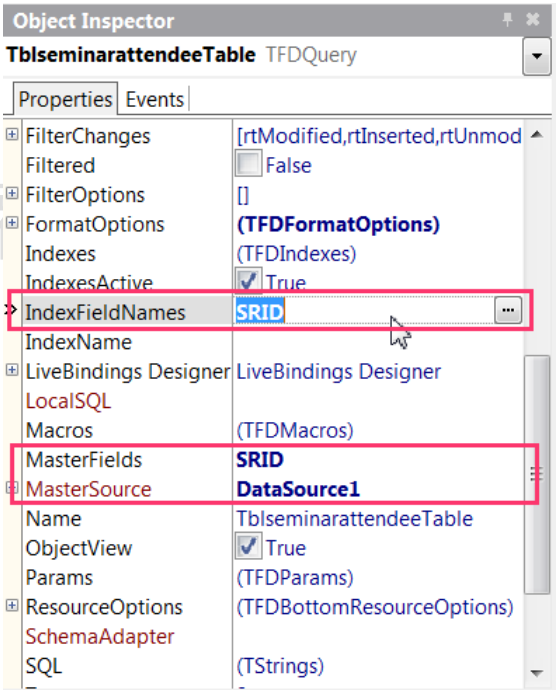



再讓我們看看如何在主/從資料中處理自動增加值欄位。

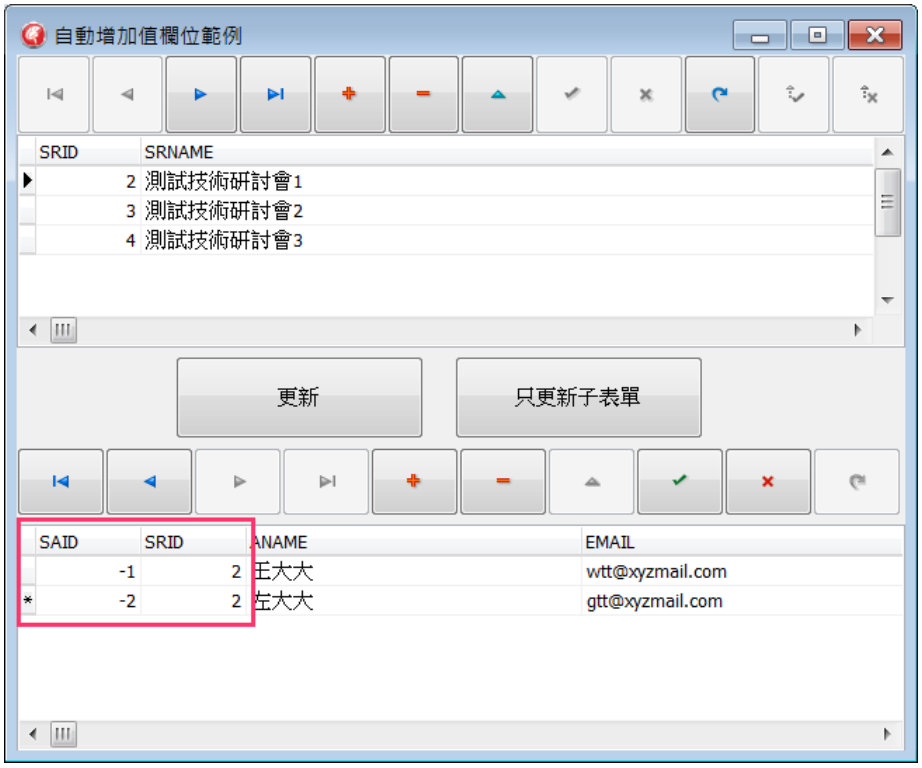
下面是 tblSeminarAttendee 資料表，它也擁有一個自動增加值欄位 SAID 並且使用 SRID 欄位和前面的 tblSeminars 資料表關連：




存取 tblSeminarAttendee 資料表的 TFDQuery 元件 TblseminarattendeeTable 使用 SRID 欄位和存取 tblSeminars 資料表的 TFDQuery 元件 TblseminarsTable 設定關連：

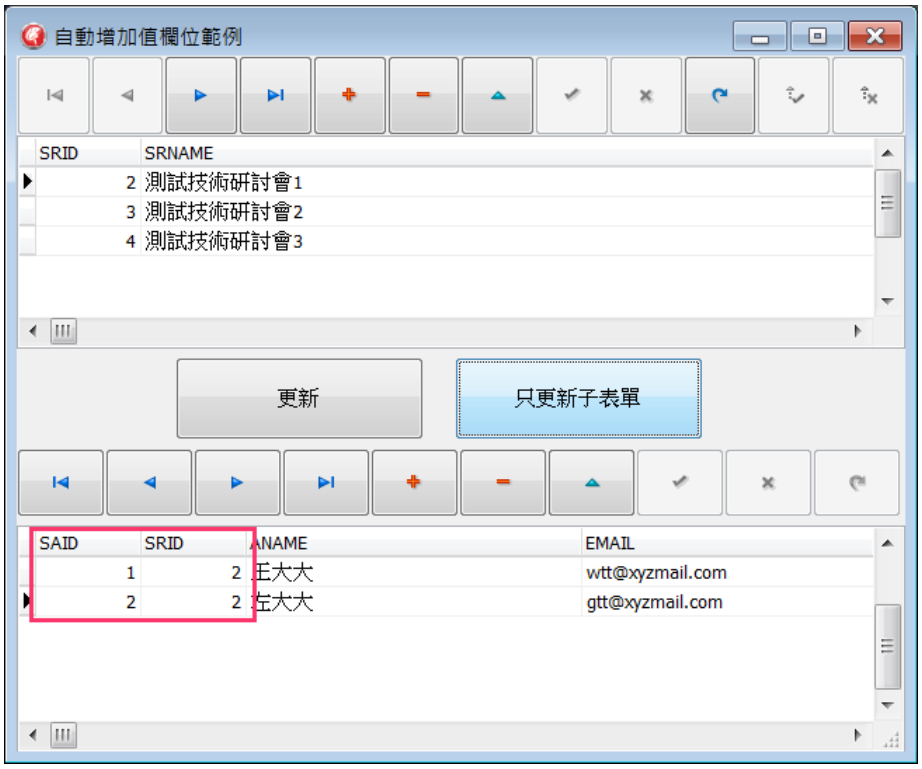



執行此範例程式並且開始在 tblSeminarAttendee 資料表加入資料，從下面的圖形可以看到 tblSeminarAttendee 資料表的自動增加值欄位 SAID 也被客戶端的 FireDAC 指定暫時數值，而關連欄位 SRID 則正確設定成主資料表的 SRID 欄位的數值：




在點選"只更新子表單"按鈕後 tblSeminarAttendee 資料表的自動增加值欄位 SAID 會正確的被後端的資料庫設定數值：



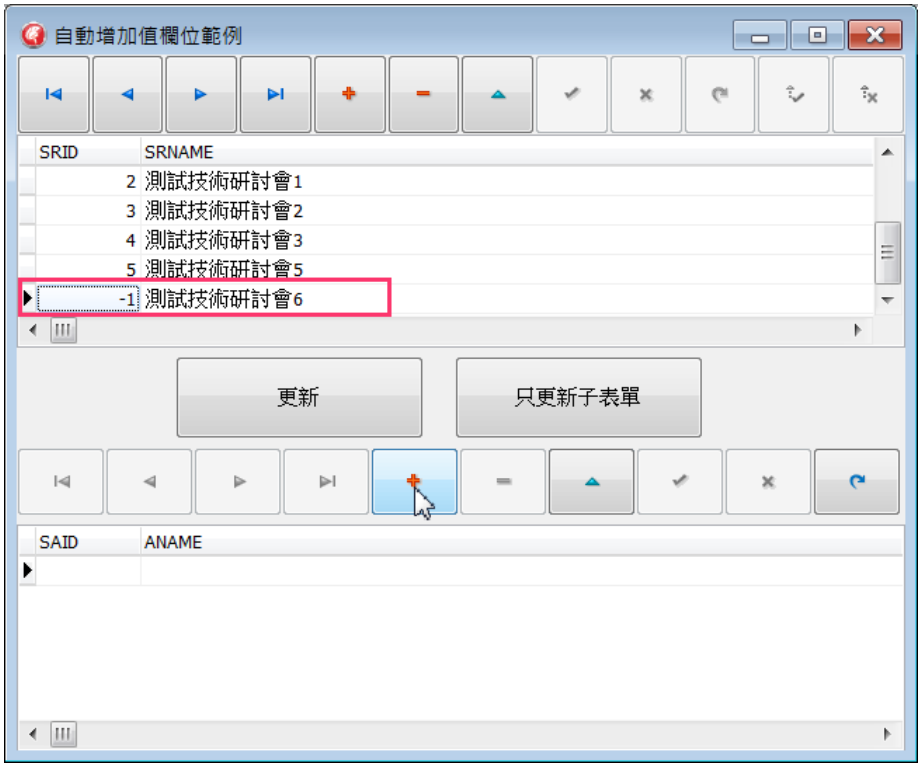


但如果想在子資料表 tblSeminarAttendee 新增資料，那麼我們必須在它的 TFDQuery 元件 TblseminarattendeeTable 的 BeforeInsert 事件中撰寫如下的程式碼：

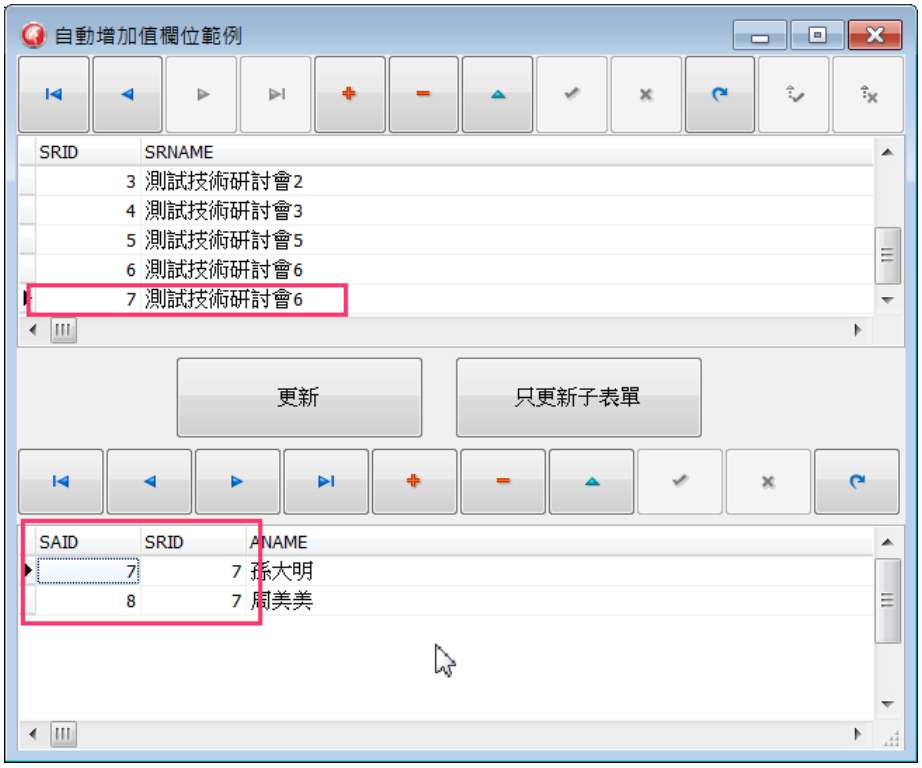
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  TblseminarsTable->ApplyUpdates(0);
  TblseminarattendeeTable->ApplyUpdates(0);
}
```

在 BeforeInsert 事件中我們需要先更新主資料表新增的資料，再取得後端資料庫真正指定給 tblSeminars 資料表的 SRID 欄位值，再由 FireDAC 指定給 tblSeminarAttendee 資料表的 SRID 欄位值。

從下面的執行結果可以看到現在在主資料表 tblSeminars 中新增資料此時 SRID 欄位值為 FireDAC 指定的-1：




接著在子資料表 tblSeminarAttendee 新增資料並更新回後端資料庫後可以看到 tblSeminarAttendee 資料表的 SRID 欄位值和 SAID 欄位值都是正確的：




從這個範例可以證明 FireDAC 對於自動增加值欄位的支援是非常強大又方便使用的。



每個資料庫對於如何支援/實作自動增加值的欄位的方式不同，FireDAC 會儘可能的使用自動模式幫忙程式師處理這種欄位，如果 FireDAC 無法自動處理您使用資料庫的自動增加值欄位型態，請參考您的資料庫和 FireDAC 的手冊說明。

## 5-5 使用計算欄位

除了代表真正資料表欄位的 TField 類別外，FireDAC 也支援計算欄位(Calculated Field)和聚集欄位(Aggregated Field)。計算欄位是所謂的虛擬欄位，計算欄位並不存在於實際的資料表中，而是在應用程式執行時暫時存在的欄位。計算欄位一般是因為應用程式在執行時需要暫時儲存一些必要的計算數值而存在的，在應用程式結束後這些計算數值並不需要儲存在資料庫中。

FireDAC 支援 4 種不同的計算欄位：

| 計算欄位種類 | 說明 |
|---|---|
| fkCalculated | 最簡單的計算欄位，通常只是使用來進行簡單的計算使用。這種計算欄位可在 TDataSet 的 OnCalcFields 事件處理 |

| | |
|---|---|
| | 及式中進行。 |
| fkInternalCalc | 進階的計算欄位，它的數值可像一般的 TField 物件一樣暫時儲存在資料集的快儲記憶體中。這種計算欄位可在 TDataSet 的 OnCalcFields 事件處理及式中進行或是使用 TField 的 DefaultExpression 特性值來計算。 |
| fkLookup | 這種計算欄位可自動執行查詢的結果值。 |
| fkAggregate | 這種計算欄位是屬於聚集欄位，它使用 TAggregateField 的 DefaultExpression 特性值來計算。 |

上面的 fkAggregate 計算欄位提供了下面 5 個運算元讓程式師使用：

| 運算元 | 說明 |
|---|---|
| Sum | 對數值或運算式提供總計值 |
| Avg | 對數值或運算式提供平均值 |
| Count | 對欄位物件或運算式提供非空白值的個數 |
| Max | 提供對字串，數值或運算式中的最大值 |
| Min | 提供對字串，數值或運算式中的最小值 |

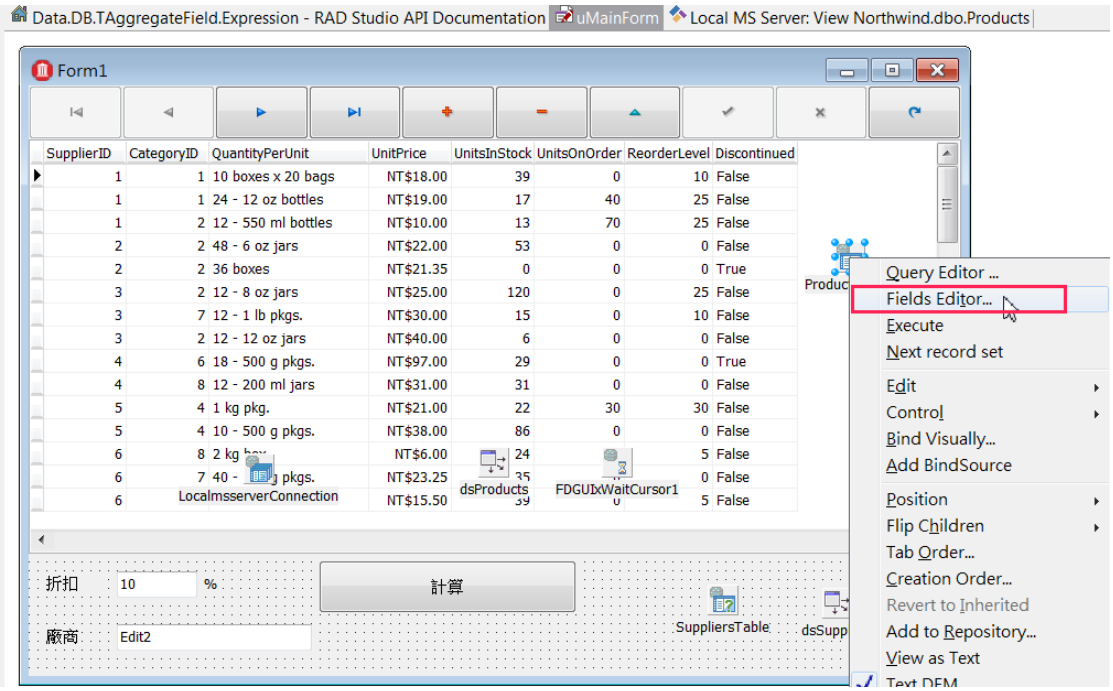
讓我們使用一個範例來說明如何使用計算欄位。



下面是使用 FireDAC 存取 MS SQL Server 的 NorthWind 資料庫中的 Products 資料表程式，現在假設我們想提供其中 UnitPrice 欄位乘上 UnitOnOrder 欄位的收入數值，接著再提供這 2 個欄位乘上折扣的收入數值等資訊，最後又可在 Products 資料表中看到供應廠商的名稱。

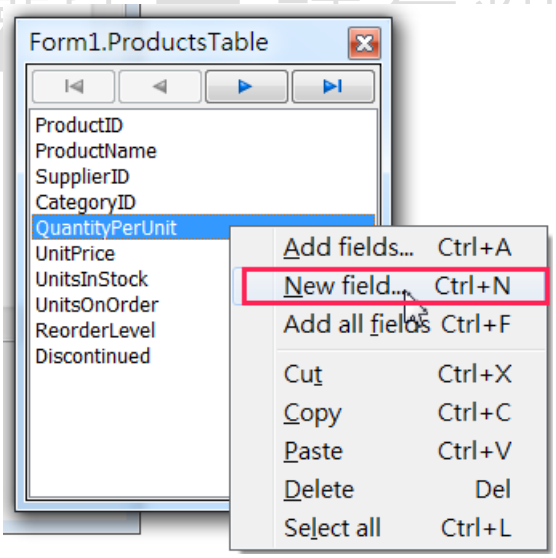
要如此做我們可以建立 2 個計算欄位提供 UnitPrice 欄位乘上 UnitOnOrder 欄位的收入數值以及這 2 個欄位乘上折扣的收入數值，最後再使用 Lookup 型態的計算欄位根據 Products 的 SupplierID 欄位值來查找供應廠商的名稱。

首先點選主表單中的 TFDQuery 元件 ProductsTable，右擊滑鼠從突顯式選單中選擇 Fields Editor...選項：

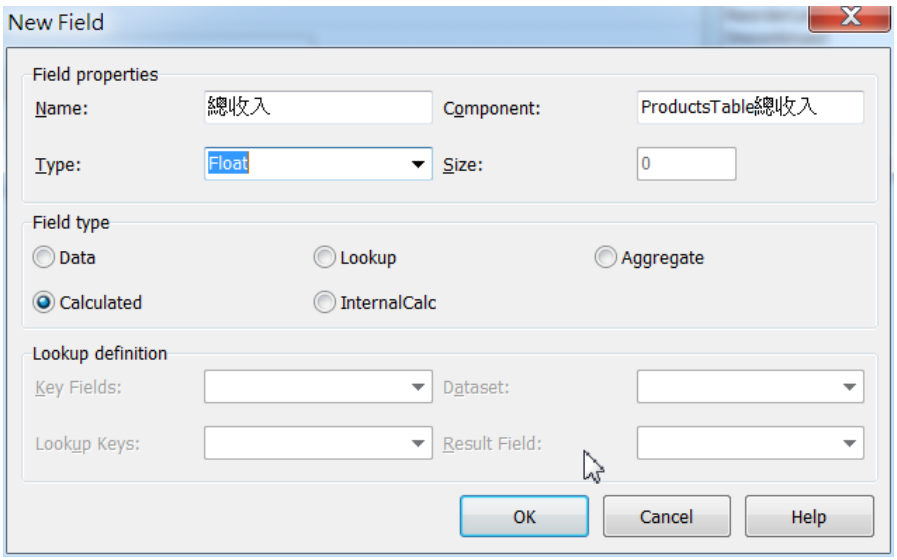



在其中點選 **New field...**選項建立新的計算欄位物件：

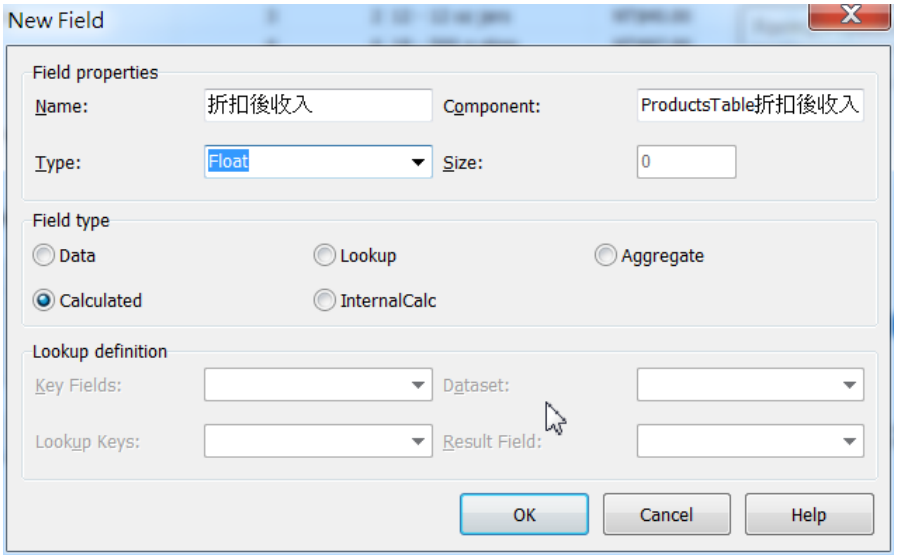




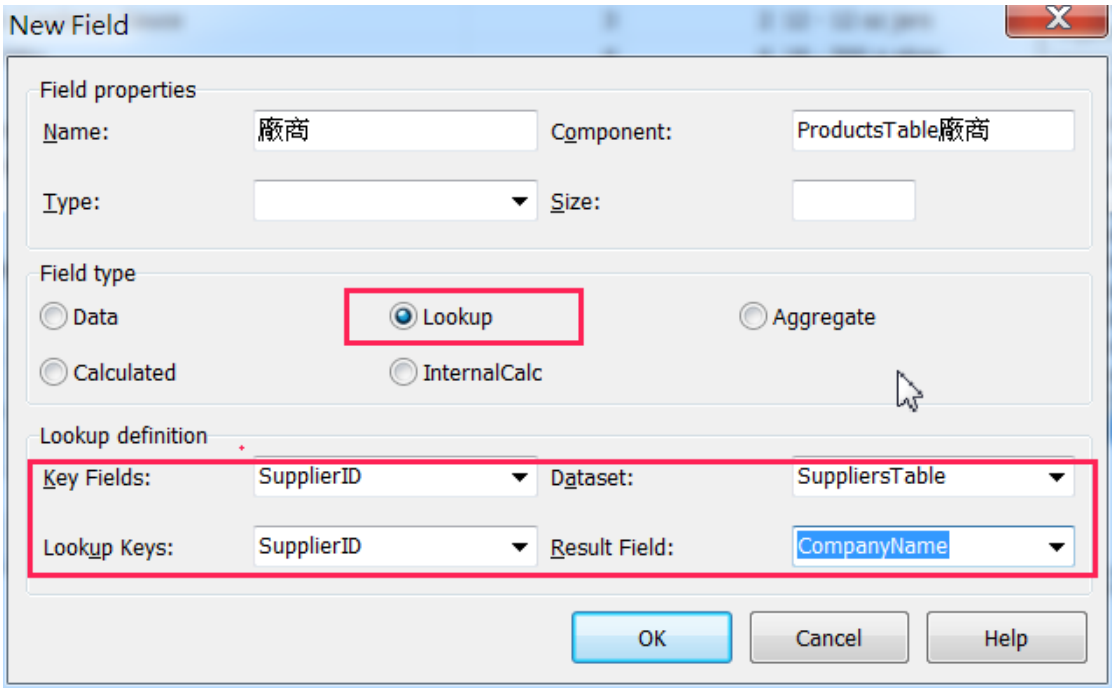

先如下建立一個總收入計算欄位物件：




再建立一個折扣後收入計算欄位物件：




最後再建立一個 Lookup 型態的計算欄位物件：




這個 Lookup 型態的計算欄位物件使用了主表單中的另外一個 TFDQuery 自動到 Supplier 資料表中根據 SupplierID 欄位值查找並回傳 CompanyName 的欄位值。



最後 ProductsTable 的欄位編輯器如握有如下的正常欄位和新增的計算欄位：


接著在 ProductsTable 的 OnCalcFields 事件處理函式中撰寫如下的程式碼：

```
void __fastcall TfmMainForm::ProductsTableCalcFields(TDataSet
*DataSet)
```

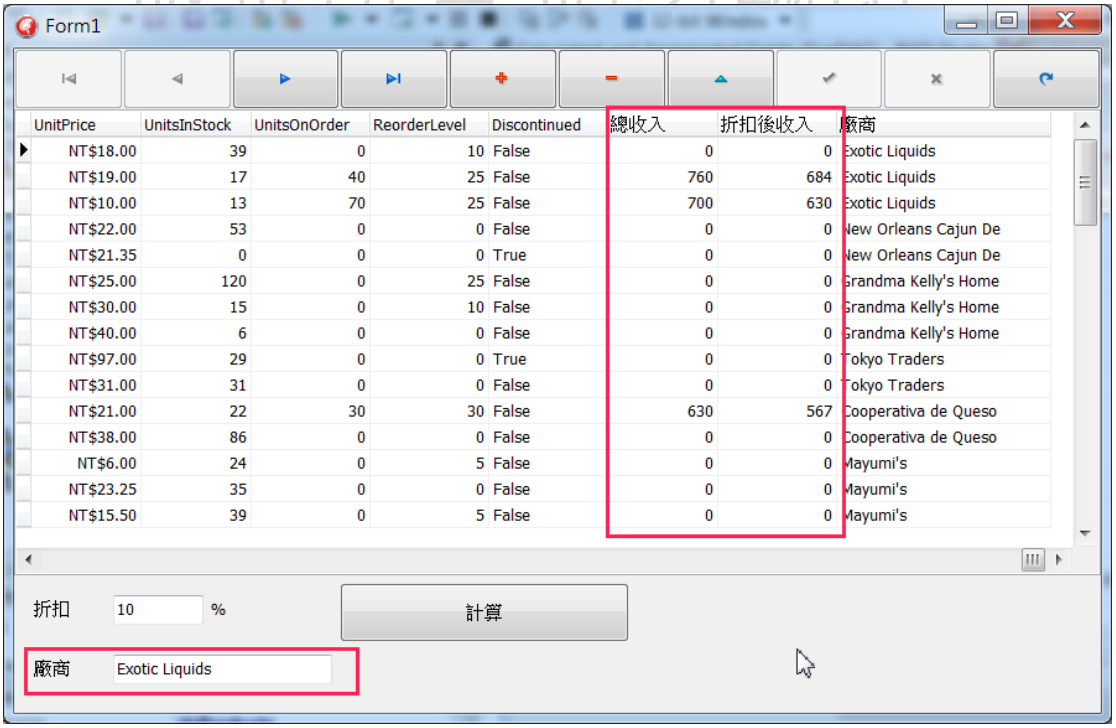
```
{
  int iDiscount = StrToInt(edtDiscount->Text);
  ProductsTable總收入->AsFloat = ProductsTableUnitPrice->AsFloat *
ProductsTableUnitsOnOrder->AsInteger;
  ProductsTable折扣後收入->AsFloat =
(ProductsTableUnitPrice->AsFloat * ((100 - iDiscount) / 100.0) ) *
ProductsTableUnitsOnOrder->AsInteger;
}
```

最後在主表單"計算"按鈕的 **OnClick** 件處理函式中撰寫如下的程式碼以便在使用者輸入新的折扣值之後強迫 **ProductsTable** 重新計算所有計算欄位的數值：

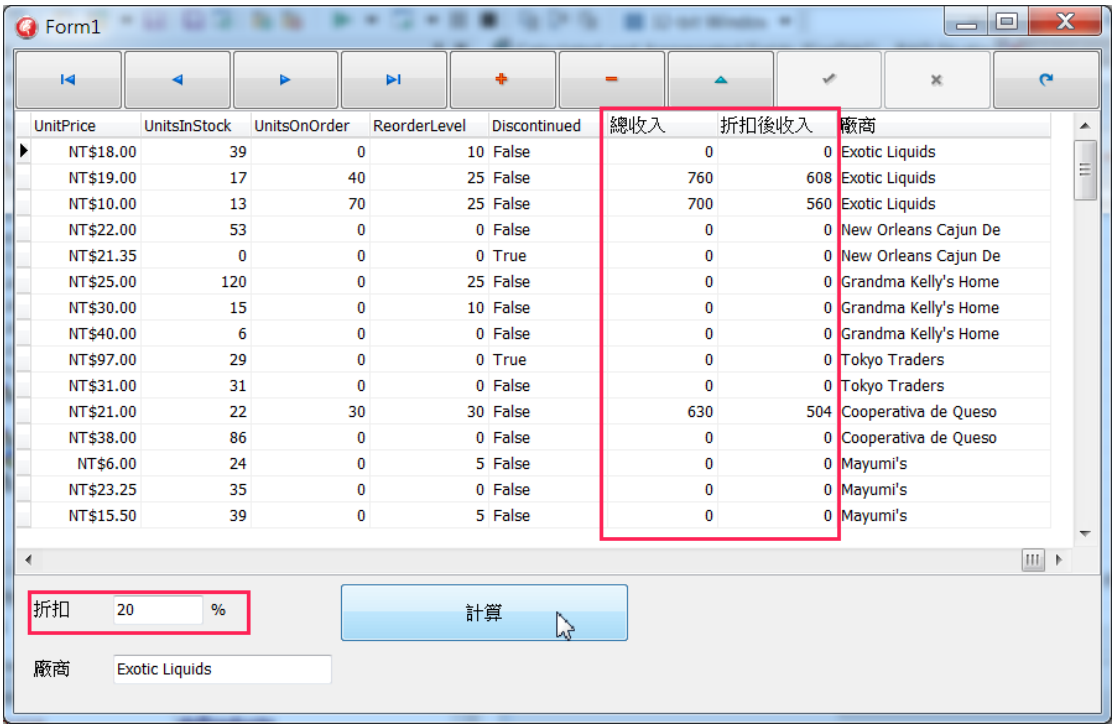
```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
{
  ProductsTable->Refresh();
}
```

執行此範例程式便可看到如下的結果，總收入計算欄位和折扣後收入計算欄位都提供了正確而需要的資訊：






如果輸入新的折扣再點選"計算"按鈕就可以看到總收入計算欄位和折扣後收入計算欄位重新進行計算並提供計算後的數值了：




## 5-6 結論



本章說明了許多 FireDAC 處理資料的技巧，善用這些技巧可以大幅增加您的 FireDAC 應用程式的功能以及執行效率。

# 第6章 MongoDB資料庫開發

NoSQL 型態的資料庫在這幾年非常的流行，也被愈來愈多的公司／企業所接受和使用，FireDAC 也在 TOKYO 的版本中開始加入支援 NoSQL 型態的資料庫，並選擇其中最為流行的 MongoDB 為第 1 個支援的目標。

本節將介紹如何使用 FireDAC 開發 MongoDB 的應用程式。



## 6-1 MongoDB 的基本介紹

MongoDB 是 10gen 開發出來的的 NoSQL 資料庫，Mongo 的資料體結構是以(Key,Value)組合的，儲存的方式是使用 JSON 格式，不過為了執行速度考量，在內部處理上的格上是使用 BSON。所謂 BSON 指的是 Binary JSON 的意思，讀者可以在下面的 URL 找到 BSON 的說明：

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

MongoDB 的特點就是每一筆文件的是欄位的資料型態是不一定的，欄位的存在性也是不一定的，這和傳統的 RDBMS 是很不一樣的。例如在 RDBMS 中一個欄位在使用之前一定要定義欄位的資料型態，一旦定義好之後該欄位儲存的數值就一定必須是該資料型態的性質，但在 MongoDB 中則無需如此，我們不需要事先定義欄位型態，每一筆資料相同欄位的數值的資料型態也可以不同。不過在 MongoDB 中所謂的每筆資料是稱為文件(Document)，下面的表格整理了傳統資料庫中的物件在 MongoDB 中的名稱：

| 關連式資料庫(RDBMS) | MongoDB |
|---|---|
| 資料庫(Database) | DataBase |
| 資料表(Table) | Collection |
| 資料(Record/Row) | Document |
| 欄位(Column) | Field |
| 主索引(PK) | _id |
| 函式(function) | Function() |
| 預儲程序(stored procedure) | mapreduce |

因此在 RDBMS 中一個資料庫有許多的資料表，而在 MongoDB 中則稱為一個資料庫有許多的 Collections，RDBMS 中一個資料表中有許多筆資料，而 MongoDB 中則稱為一個 Collection 有許多 Documents。

MongoDB 有許多的特點，下面是比較重要的特點：

1.MongoDB 可以處理資料庫為 T 級量 的資料庫，也就是處理大數據的資料庫。

2.分散式的資料庫模式，可以把眾多資料庫串聯後處理大數的資料。



3.MongoDB 可直接儲存物件，每個欄位也可以儲存物件

4.Monogo 基本上是使用 JavaScript 和 JSON 的資料庫

例如在 MongoDB 資料庫中假設有一個稱為 employee 的 Collection，在這個 Collection 中我們可以使用下面的 JSON 格式儲存一個 Document：

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

如果我們注意上面的格式就可以發現它是一個 JSON 物件，其中包含了 4 個 JSON Pair，不過上面的"_id" 欄位是由 MongoDB 自行產生的鍵值，由於 MongoDB 的欄位也可以忽略，因此我們又可以使用下面的 JSON 新增另一 Document，而且新增了一個 email 欄位，這個 email 欄位則是一個 JSON 陣列物件：

```
{
	"_id" : ObjectId("55dad3b0a797e189008fec47"),
	"name" : "王小華",
	"account" : "WSH",
	"country" : "tw",
	"age" : 22,
	"email" : [
		"wsh@gmail.com.tw",
		"wsh@hotmail.com"
	]
}
```

有了這些基本的說明後我們就可以開始說明如何使用 FireDAC 來開發 MongoDB 的應用程式了，如果讀者想瞭解更多有關 MongoDB 的觀念，那麼可以到 MongoDB 官方網站或是參考市面上的 MongoDB 專業書籍。

## 6-2 下載和安裝 MongoDB



在使用 FireDAC 開發 MongoDB 應用程式之前讀者必須先下載和安裝 MongoDB，要下載 MongoDB，請到 MongoDB 官方網站：

http://www.mongodb.org/downloads

例如筆者下載和安裝的是 Win32 3.0.4 的版本，

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

由於上面的版本使用了 SSL，因此在您的執行目錄中必須擁有 SSL 的 libeay32.dll 和 ssleay32.dll 這 2 個 DLL。

下載和安裝完 MongoDB 後會在 Program Files (x86)或是 Program Files MongoDB 目錄中看到如下的內容：

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

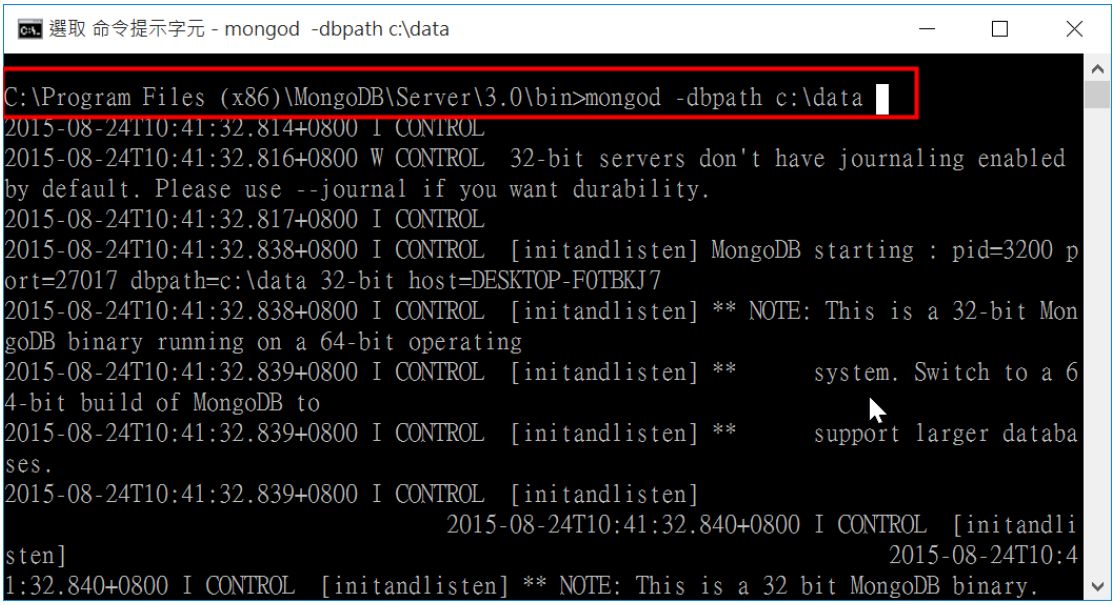
| Name | Ext | Size | Date | Attr |
|---|---|---|---|---|
| [..] | | <DIR> | 2015/08/24 10:39 | ---- |
| bsondump | exe | 3,500,032 | 2015/06/15 16:15 | -a-- |
| mongo | exe | 5,706,752 | 2015/06/15 16:17 | -a-- |
| mongod | exe | 10,802,688 | 2015/06/15 16:20 | -a-- |
| mongod | pdb | 104,747,008 | 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 的資料庫伺服器，執行它就可以啟動 MongoDB，當 MongoDB 執行時它會到預定的資料庫目錄尋找資料庫，在 Windows 平台這個預定的資料庫目錄是

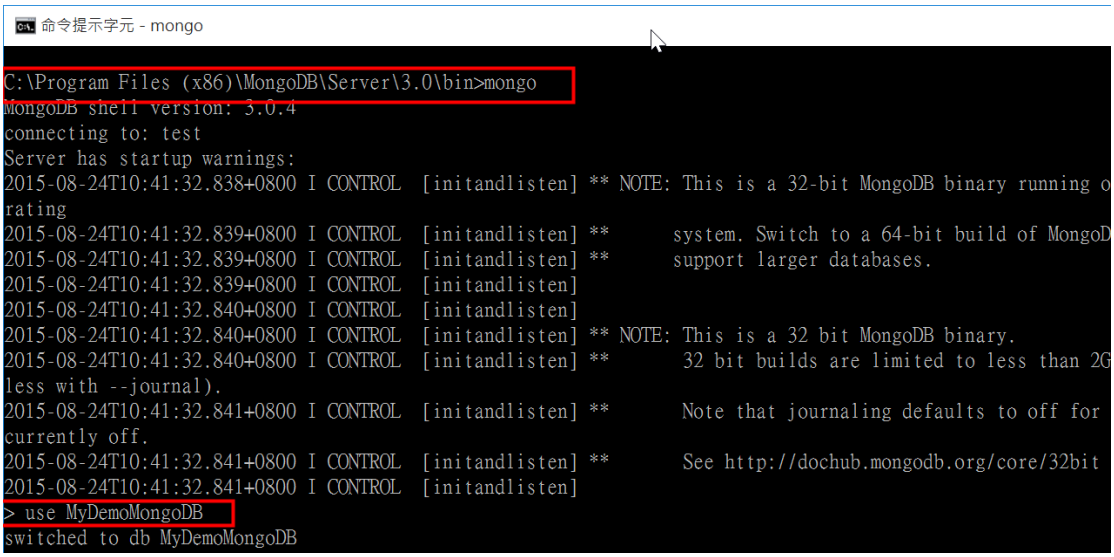
```
C:\data\db\
```



我們可以在啟動 MongoDB 時使用 Mongod.exe 的執行選項—dbpath 來指定想使用的預定的資料庫目錄。例如下面的圖形顯示在 MongoDB 目錄中使用 DOS 命令視窗執行 MongoDB 並使用 c:\data 做為預定的資料庫目錄：



啟動 MongoDB 資料庫後就可以再使用 MongoDB 的命令工具 Mongo.exe 來管理 MongoDB 資料庫，Mongo.exe 使用 JavaScript 語言來執行 MongoDB 命令，例如下面的圖形顯示在 MongoDB 目錄中使用另外一個 DOS 命令視窗執行 Mongo.exe：

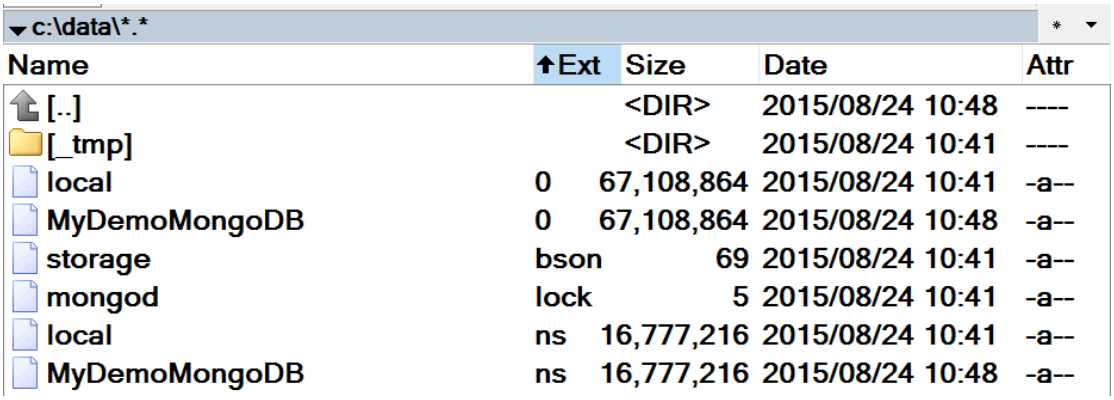



例如現在我們要建立一個本節使用的範例 MongoDB 資料表(即 MongoDB 的 Collection) “MyDemoMongoDB”，我們可以在 Mongo.exe 中使用：



```
use MyDemoMongoDB
```

如上圖所示，那麼在 Mongo.exe 執行完畢之後，我們就可以在 c:\data 目錄中看到如下的結果：




在 c:\data 目錄中就可以看到上面的結果，MongoDB 為我們建立了 MyDemoMongoDB 以及其他相關的檔案。那麼如果我們繼續在 Mongo.exe 中執行

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

如下圖所示，那麼就可以在 MyDemoMongoDB 資料庫中建立一個 employee 資料表(即 MongoDB 的 Document)建立一筆資料了：

```
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, 這些都是使用 MongoDB 的工具和命令來管理和處理 MongoDB，但對於 C++Builder 的開發人員來說使用熟悉的知識和技術來管理和處理 MongoDB 才是最重要的。在 TOKYO 中 FireDAC 加入了支援 MongoDB 的功能，讓 C++Builder 的開發人員可以同時使用 MongoDB 的 API 或是 FireDAC 元件來管理和處理 MongoDB，這正是下一小節的內容。

# 6-3 FireDAC 對 MongoDB 的支援

TOKYO 中的 FireDAC 又增加了許多的新功能，其中最大的新增功能就是支援 MongoDB，FireDAC 同時提供了類別和元件來支援 MongoDB，這也代表 C++Builder 開發人員可以藉由 C++Builder 類別來呼叫 MongoDB API 來管理和處理 MongoDB，或是使使用 FireDAC 的 MongoDB 元件。

基本上 TOKYO 使用了下面的類別來封裝 MongoDB 的 API：

| MongoDB | FireDAC類別 |
|---|---|
| DataBase | TMongoDatabase |
| Collection | TFDMongoDataSet |
| Document | TMongoDocument |
| Field | TField |
| Mongo Environment | TMongoEnv |
| Mongo Command | TMongoCommand |

由於 MongoDB 使用 JSON/BSON 來處理和封裝資料，因此讀者也必須瞭解 TOKYO 中的新 JSON 框架，例如 TJsonReader，TJsonWriter 和 TJSONCollectionBuilder 等類別，請參考本系列的"C++Builder 開發手冊一書"。

此外 FireDAC 也封裝了下面的元件幫助 C++Builder 開發人員使用 TDataSet 的觀念和技術來使用 MongoDB：

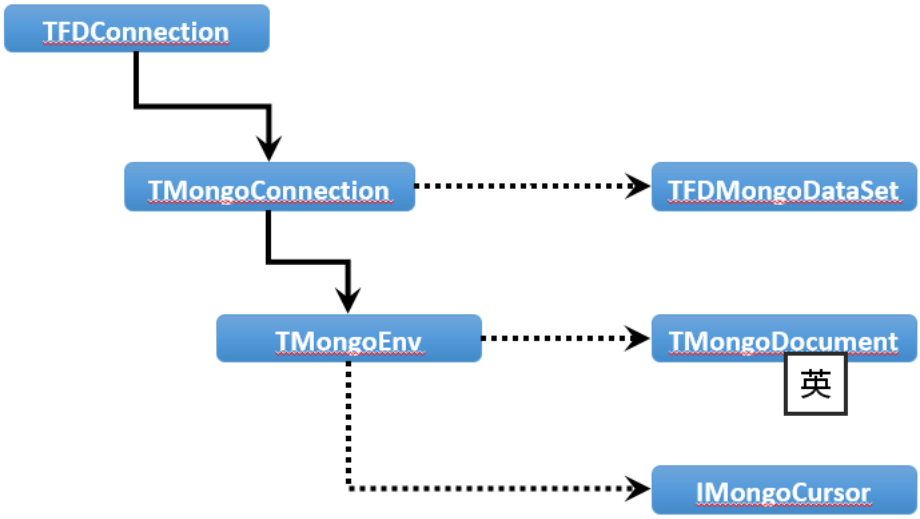
| 元件 | 名稱 | 說明 |
|---|---|---|
|  | TFDPhysMongoDriverLink | 提供連結和驅動 MongoDB 的能力 |
|  | TFDMongoDataSet | 封裝 MongoDB 的 Database 或是 Collection 的元件 |
|  | TFDMongoQuery | 封裝 MongoDB 執行命令的功能 |
|  | TFDMongoPipeline | 封裝 MongoDB Aggregate 功能的元件 |



為了讓讀者瞭解如何使用這些類別的元件，在下面的 2 個小節中將分別以數個範例來說明。

## 6-3-1 使用 C++Builder 類別處理 MongoDB

基本上要使用 C++Builder 類別處理 MongoDB 的話，開發人員可以藉由從 TFDConnection 元件開始取取得 TMongoConnection，TMongoEnv 等類別物件再呼叫其中的方法就可以操作和處理 MongoDB 了，下面圖形顯示出了如從 TFDConnection 元件開始取得相關 MongoDB 相關的類別物件：




在『C++Builder 開發手冊』中討論 TOKYO 新的 JSON/BSON 框架一章中我們使用了如下的範例：

```
{
        "name" : "王小華",
        "account" : "WSH",
        "country" : "tw",
        "age" : "22",
        "email" : [
                "wsh@gmail.com.tw",
                "wsh@hotmail.com"
        ]
}
```



現在讓我們繼續使用這個範例，讓我們建立一個名為"FireDACDemoDB"的資料庫和名為"FireDACColletions"的 Collection 中建立數個類似"name" : "王小華"結構的 Document。

要在 MongoDB 中建立 Collection 和 Document，程式師可以藉由存取 TMongoConnection 和 TMongoEnv 物件完成，一旦取得了 TMongoConnection 物件就可以藉由它的 Collections 特性值在 MongoDB 中建立資料庫和 Collection：

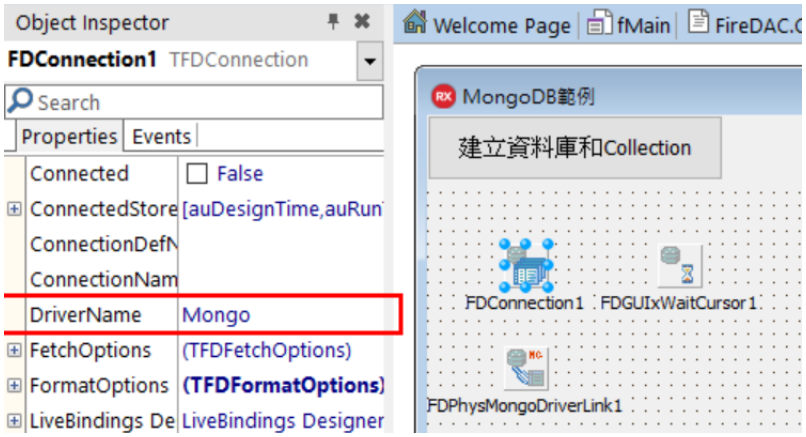
```
  __property TMongoCollection* Collections[const
System::UnicodeString ADBName][const System::UnicodeString
```

```
AColName] = {read=GetCollectionsProp};
```

請注意 Collections 特性接受 2 個索引參數，一個就是資料庫名稱，另外一個就是 Collection 名稱，而使用資料庫名稱和 Collection 名稱存取 Collections 特性時，如果在 MongoDB 中此時沒有此資料庫名稱和 Collection 名稱，那麼 MongoDB 就會自動幫我們建立此資料庫和 Collection，非常的方便，現在就讓我們說明如何完成這 2 個工作。

## 存取 TMongoConnection 和 TMongoEnv 物件

首先建立一個 VCL Application(FireMonkey Application 也可以)，在主表單中加入 TFDConnection，TFDPhysMongoDriverLink 和 TFDGUIxWaitCursor 元件，再設定 TFDConnection 元件的 DriverName 特性為 Mongo，如下所示：




在 TFDConnection 類別中的 CliObj 特性值就封裝了 TMongoConnection 物件，因此藉由存取此特性值再轉換型態為 TMongoConnection 即可：

```
__property void * CliObj = {read=GetCliObj};
```

取特了 TMongoConnection 物件後，在它的 Env 特性值就封裝了 TMongoEnv 物件：

```
__property TMongoEnv* Env = {read=FEnv};
```



因此在上面的範例主表單的 OnCreate 事件中就可以使用下面的程式碼取得 TMongoConnection 物件和 TMongoEnv 物件：

```
void __fastcall TfmMainForm::FormCreate(TObject *Sender)
{
  FDConnection1->Connected = true;
  FDCon = (TMongoConnection*)  FDConnection1->CliObj;
  FDEnv = FDCon->Env;
}
```

當然我們要宣告 FDCon 和 FDEnv 如下：

```
  TMongoEnv *FDEnv;
  TMongoConnection *FDCon;
```

接著就可以在主表單的”建立資料庫和 Collection”按鈕中撰寫程式碼開始建立此資料庫和 Collection 並新增 Document：

```
void __fastcall TfmMainForm::Button1Click(TObject *Sender)
```

```
{
  CreateDBAndCollection();
  InsertDemoData();
  ShowDocuments();
  PageControl1->ActivePageIndex = 0;
}
//--
```

CreateDBAndCollection()方法只需要使用'FireDACDemoDB'資料庫名稱和'FireDACColletions' Collection 名稱那麼 MongDB 就會自動幫我們建立'FireDACDemoDB'資料庫和'FireDACColletions' Collection，而 RemoveAll()方法是先把'FireDACColletions'中所有的 Documents 刪除以便開始新增 Document：

```
void TfmMainForm::CreateDBAndCollection()
{

FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Remove
All();
}
```



InsertDemoData()方法呼叫 InsertDocument()方法取得新增了資料的 TMongoDocument 物件，再呼叫 TMongoCollection 物件的 Insert()方法把 TMongoDocument 物件加入到 TMongoCollection 物件中以便把 Document 物件寫入 Collection 中：

```
void TfmMainForm::InsertDemoData()
{
  TMongoDocument *FDDoc;

  try
  {

FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->BeginB
ulk();

    FDDoc = InsertDocument(L"王小華", "wsh", "tw", "22",
"wsh@gmail->com->tw", "wsh@hotmail->com");
```

```
FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Insert
(FDDoc);
   delete FDDoc;
  …
```

InsertDocument() 方法先呼叫 TMongoEnv 的 NewDoc() 方法建立 TMongoDocument 物件，再使用類似 TJsonObejectBuilder(請參考"C++Builder 開發手冊")的方式新增 JSON 型態的資料：

```
TMongoDocument* TfmMainForm::InsertDocument(const String sName,
const String sAccount, const String sCountry, const String sAge,
const String sEMail1, const String sEMail2)
{
  TMongoDocument* pResult = FDEnv->NewDoc();

  pResult
  ->Add(L"name", String(sName) )
  ->Add(L"account", String(sAccount) )
  ->Add(L"country", String(sCountry) )
  ->Add(L"age", String(sAge) )
  ->BeginArray(L"email")
    ->Add(L"信箱 1", String(sEMail1) )
    ->Add(L"信箱 2", String(sEMail2) )
  ->EndArray();

  return pResult;
}
```



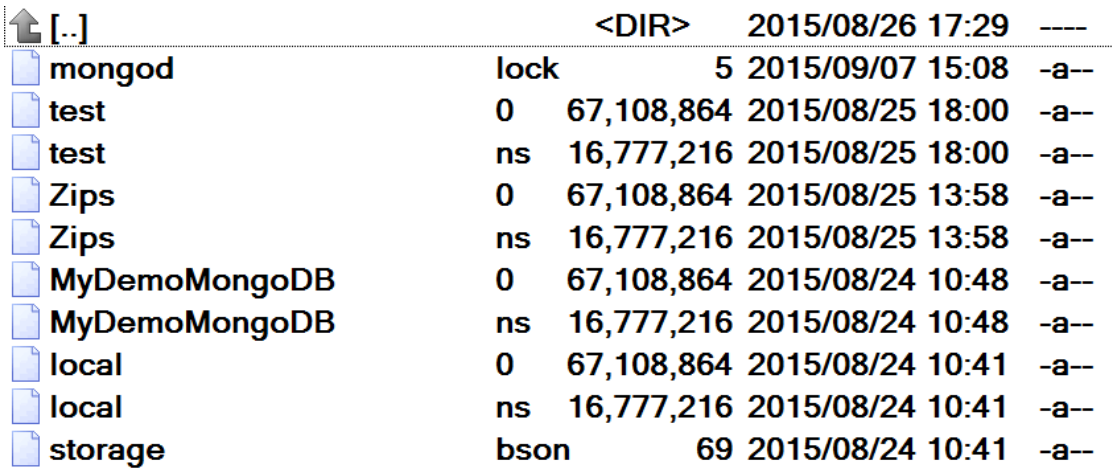
最後的 ShowDocuments()方法是藉由_di_IMongoCursor 介面一一的顯示 'FireDACColletions' Collection 中所有的 Document。

007 行可藉由 TMongoConnection 取得目前使用的 MongoDB 資料庫版本資訊，007 行呼叫 TMongoCollection 的 Find()方法搜尋到所有的 Document，Find()方法會回傳_di_IMongoCursor 介面，_di_IMongoCursor 介面的 Next()方法可以一一的取得每一個 Document 物件，010 行就可以藉由_di_IMongoCursor 介面的 Doc 特性取得 TMongoDocument 物件，再把其中的資料以 JSON 格式回傳並顯示出來：

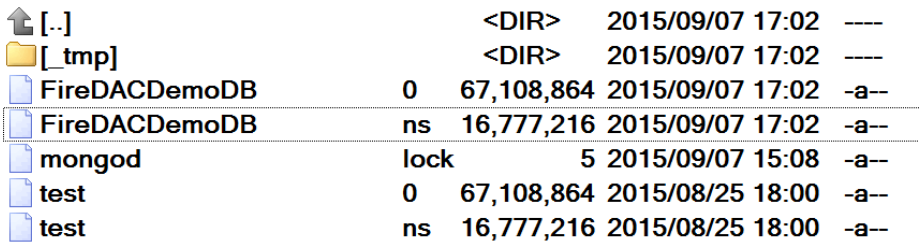
```
001    void TfmMainForm::ShowDocuments()
002    {
003      Memo1->Lines->Clear();
004      Memo1->Lines->Add( IntToStr(FDCon->ServerVersion) );
005      Memo1->Lines->Add("\n");
006
007      _di_IMongoCursor IFDCrs =
FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Find(N
ULL);
008      while (IFDCrs->Next())
009      {
010        String sData = IFDCrs->Doc->AsJSON;
011        Memo1->Lines->Add(sData);
012      }
013
014      Memo1->Lines->Add(L"\n 總筆數 : " +
015       IntToStr
(FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Count
()->Value() ) );
016    }
017
```



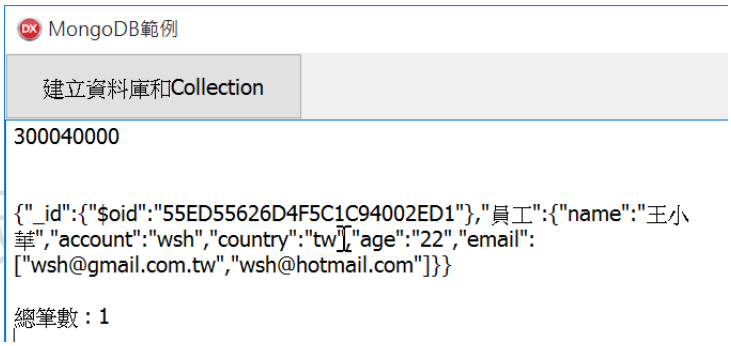
執行上面的範例程式之前下圖是筆者 MongoDB 使用的資料目錄，現在我們沒有看到 FireDACDemoDB 資料庫：



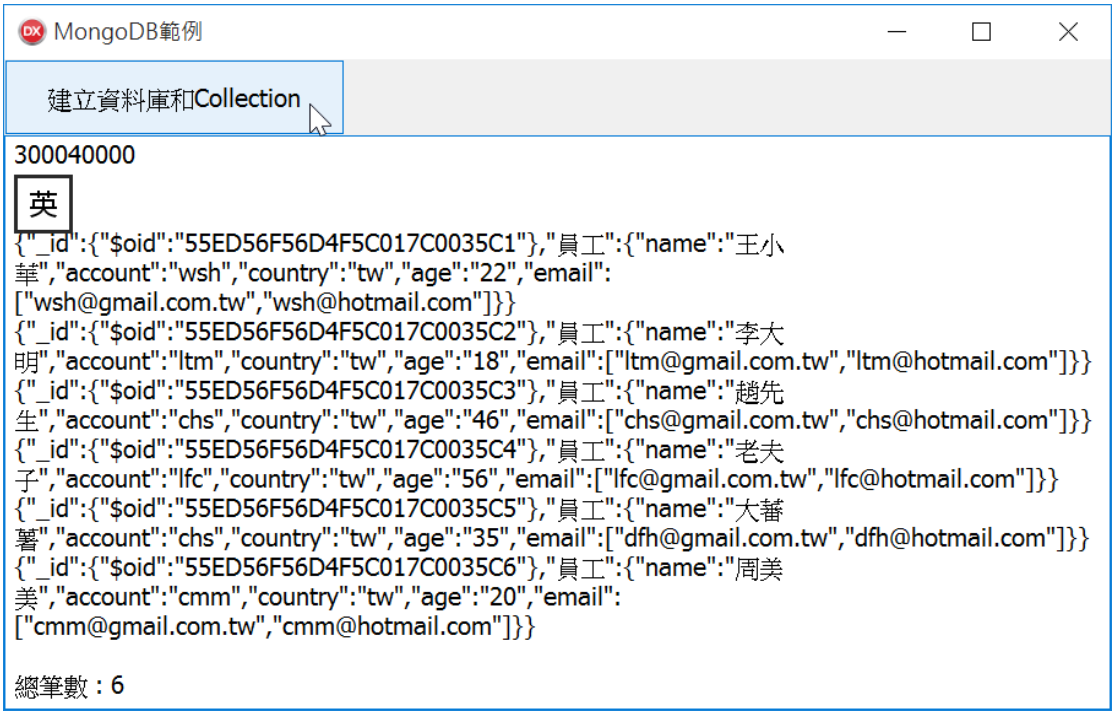
但在執行上面的範例程式之後可以在目錄中看到 FireDACDemoDB 資料庫被自動建立了：




下圖則是此範例程式執行的結果，我們可以看到筆者使用的 MongoDB 資料庫版本是 3.04，而且成功的在 FireDACDemoDB 資料庫的'FireDACColletions' Collection 中新增了"王小華"這個 Document：




而下圖則是在'FireDACColletions' Collection 中新增多個 Document 的結果：




從上面的說明可以瞭解藉由 TFDConnection 元件程式師可以直接取得封裝 MongoDB 功能的類別來處理 MongoDB。



但是 C++Builder 的程式師更習慣於使用 TDataSet 的概念來處理資料，因此 TOKYO 的 FireDAC 也開始封裝 MongoDB 的功能到 TDataSet 元件中，下面的小節將說明如何使用 C++Builder 程式師熟悉的 TDataSet 概念和技術來使用 MongoDB。

## 6-3-2 使用 FireDAC 元件處理 MongoDB

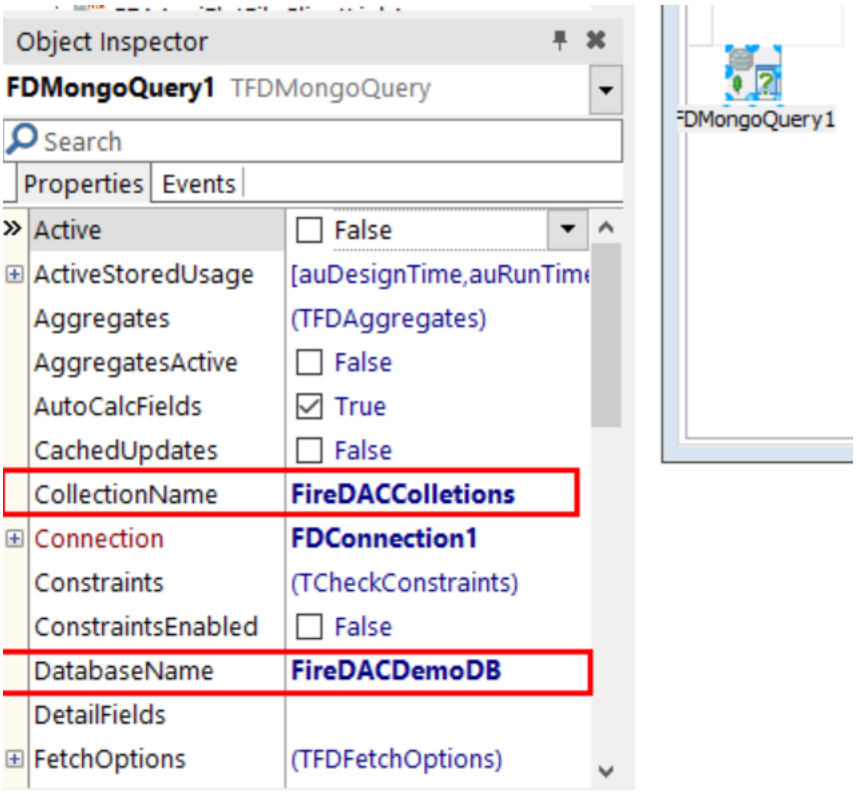
為了讓 C++Builder 程式師避免自己學習和呼叫複雜的 MongoDB API，因此 TOKYO 的 FireDAC 提供了 TFDMongoQuery 和 TFDMongoDataSet 等 TDataSet 元件封裝了許多 MongoDB API 的功能讓 C++Builder 程式師以熟悉的方式來使用 MongoDB，如此一來可以減少許多的開發時間，本小節將簡單的說明如何使用這 2 個 TDataSet 元件。

> TOKYO 的 FireDAC 開始了封裝 MongoDB API 的工作，但目前尚未完全封裝，未來將持續的開發支援 MongoDB 的功能，並未會整合到 IDE 中讓 Data Explorer 也可以直接使用 MongoDB。不過這也不用擔心，因為 C++Builder 程式師如果發現目前不足的地方也可以直接呼叫 MongoDB 類別的方法。

要使用 TFDMongoQuery 元件，程式師需要設定 TFDMongoQuery 元件下面的 3 個特性值：

| 特性 | 說明 |
|---|---|
| Connection | 設定連結到 TFDConnection 元件 |
| DatabaseName | 設定為 MongoDB 的資料庫名稱 |
| CollectionName | 設定為 MongoDB 的 Collection 名稱 |

現在就讓我們使用 TFDMongoQuery 元件來搜尋資料。首先在範例程式的主表單加入 TFDMongoQuery 元件，再如下設定它的 DatabaseName 和 CollectionName 特性值：




接著在主表單加入一個” 搜尋資料”按鈕，於它的 OnClick 事件中使用我們已熟悉的 Close()和 Open()方法執行查詢。但請注意的是 TMongoQuery 不使用 SQL 語法查詢資料，而是使用 JSON 語法查詢資料。因此要查詢資料我們需要把查詢的 JSON 指定給 TMongoQuery 元件的 QMatch 特性：

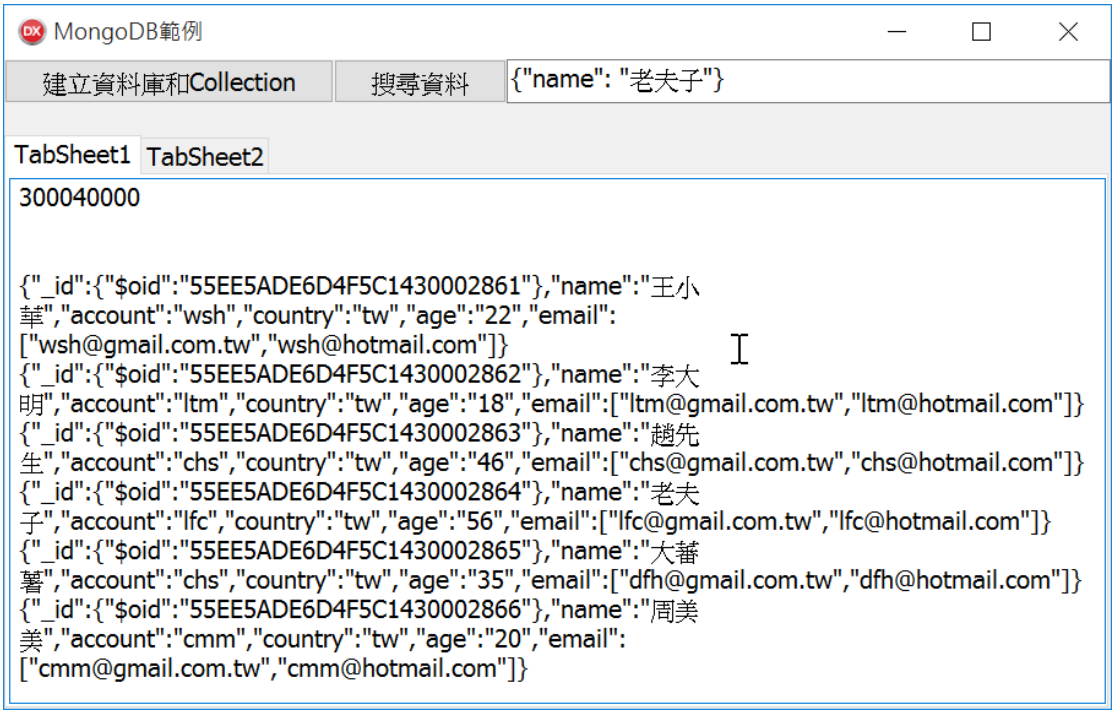
```
void __fastcall TfmMainForm::Button2Click(TObject *Sender)
{
  FDMongoQuery1->Close();
```

```
  FDMongoQuery1->QMatch = Edit1->Text;
  FDMongoQuery1->Open();
  PageControl1->ActivePageIndex = 1;
}
```

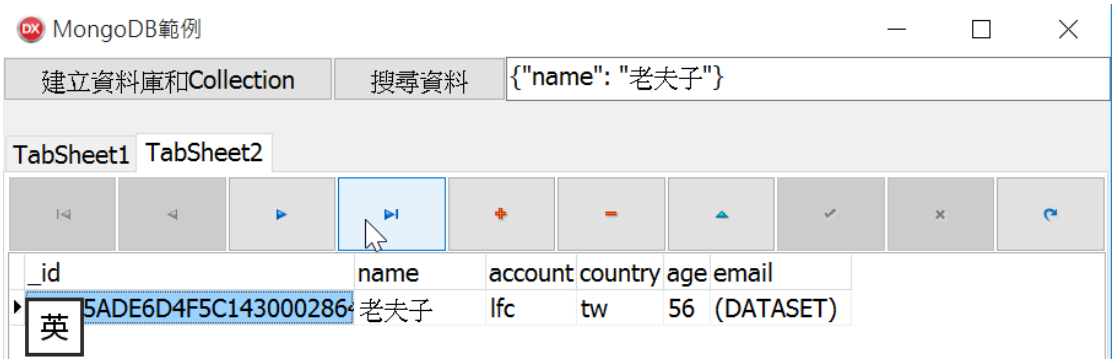
現在執行範例程式並輸入：

```
{"name": "老夫子"}
```

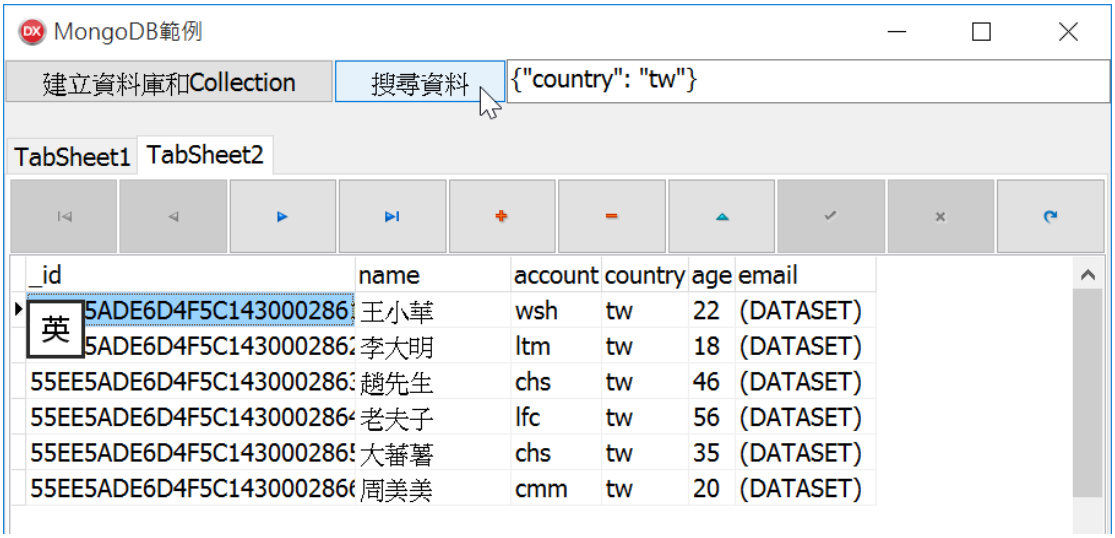
在 Collection 中查詢"老夫子"這筆 Document，下圖是先在'FireDACColletions' Collection 中新增多個 Document 的結果：




下圖則是使用 TFDMongoQuery 元件的 QMatch 特性查詢"老夫子"這筆 Document 的結果：



而下圖則是查詢所有 country=tw Document 的結果：

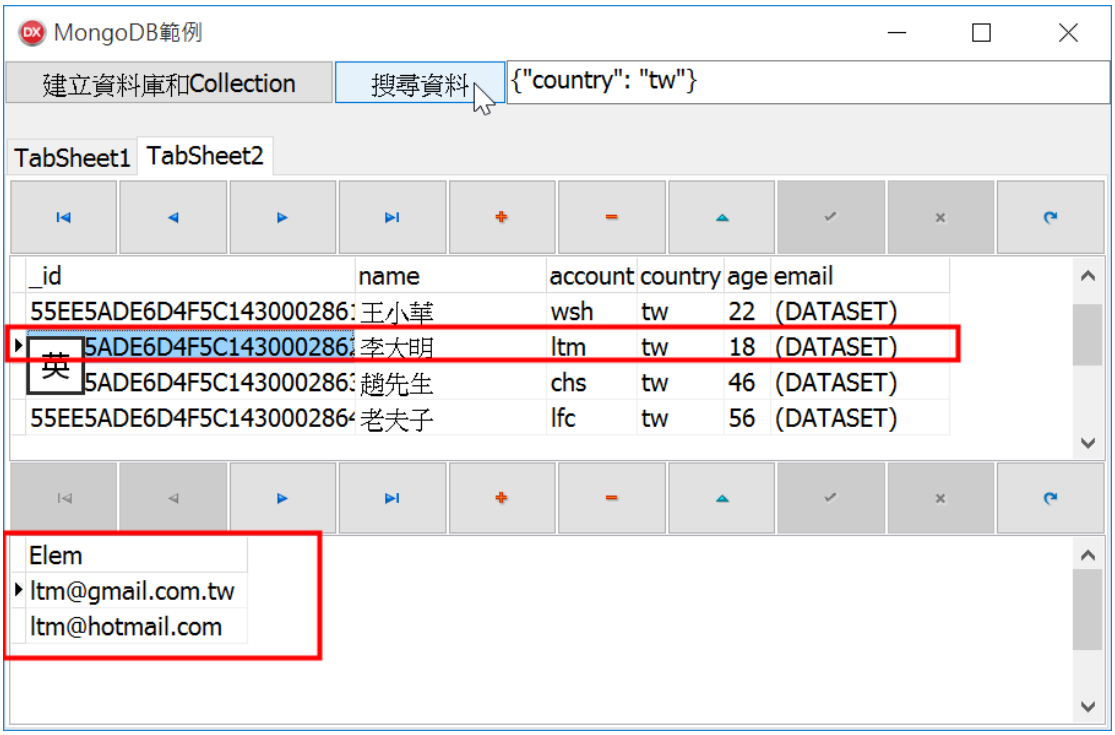



請注意上圖中的 email 欄位顯示的 DATASET，為什麼？還記得在前面 email 是一個 JSON 的陣列物件嗎？當使用 TMongoQuery 元件進行查詢時如果一個 MongoDB Document 的欄位是 JSON 的陣列或是 JSON 物件的話，TMongoQuery 元件就會以內嵌資料集物件(Nested Dataset)來代表，因此我們可以藉由先把這個欄位值轉型為 TDataSetField 物件，再存取 它的 NestedDataSet 特性值就可以取得代表 JSON 的陣列或是 JSON 物件的 TDataSet 物件。

例如我們修改一下” 搜尋資料”按鈕的 OnClick 程式碼，加入下面的 007 行就可以取得 email 欄位 JSON 的陣列值：

```
001    void __fastcall TfmMainForm::Button2Click(TObject *Sender)
002    {
003      FDMongoQuery1->Close();
004      FDMongoQuery1->QMatch = Edit1->Text;
005      FDMongoQuery1->Open();
006
007      dsNestedDataSet->DataSet = ( (TDataSetField
*)FDMongoQuery1->FieldByName("email") )->NestedDataSet;
008      PageControl1->ActivePageIndex = 1;
009    }
```

下面就是在主表單中再加入另外一個 TDBNavigator 和 TDBGrid 元件，就可以看到每一個 Document 資料的 email JSON 陣列的內容：




使用 TFDMongoQuery 元件處理 MongoDB 的 Document 資料是很方便的，也讓 C++Builder 程式師可以使用熟悉的 TDataSet 技術，但是我們仍然可以直接使用 TMongoQuery 類別來處理 MongoDB 的 Document 資料。

## 6-3-3 使用 TMongoQuery 搜尋資料

C++Builder 的 TMongoQuery 類別封裝了 IMongoCursor 介面，C++Builder 程式師可以直接使用 TMongoQuery 類別來處理 MongoDB。例如在 TMongoQuery 類別中提供了 Project()，Match()和 Sort()方法可以查詢和排序 Collection 中的 Document，而 Open()方法則可以回傳包含結果 Document 的_di_IMongoCursor 介面：

```
TProjection* __fastcall Project(const System::UnicodeString AJSON
= System::UnicodeString());

TExpression* __fastcall Match(const System::UnicodeString AJSON =
System::UnicodeString());

TSort* __fastcall Sort(const System::UnicodeString AJSON =
System::UnicodeString());
```

```
_di_IMongoCursor __fastcall Open(void);
```

現在盲讓我們簡單的展示一下如何使用 TMongoQuery 類別。

讓我們在主表單中加入一個" 使用 TMongoQuery 搜尋"按鈕，在 OnClick 事件中撰寫如下的尬式碼：

```
void __fastcall TfmMainForm::Button3Click(TObject *Sender)
{
  Memo2->Lines->Clear();
  _di_IMongoCursor IFDCrs = SearchDocument(Edit1->Text);
  while (IFDCrs->Next())
  {
  String sData = IFDCrs->Doc->AsJSON;
  Memo2->Lines->Add(sData);
  }
}
```

我們先呼叫 SearchDocument() 方法搜尋符合條件的 Document，SearchDocument()方法會回傳搜尋結果的_di_IMongoCursor 介面，我們再使用它把搜尋結果顯示出來。



SearchDocument()方法在 004 行直接建立 TMongoQuery 物件，007~011 行設定搜尋 JSON 條件，最後 012 行呼叫 TMongoCollection 的 Find()方法進行搜尋：

```
001   _di_IMongoCursor TfmMainForm::SearchDocument(const String
sMatch)
002   {
003     _di_IMongoCursor pResult = NULL;
004     TMongoQuery *pMQuery = new TMongoQuery(FDEnv);
005     try
006     {
007       pMQuery
008         ->Match()
009         ->Clear()
010         ->Append(sMatch)
011       ->End();
012       pResult =
```

```
FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Find(p
MQuery);
013      }
014      __finally
015      {
016        delete pMQuery;
017      }
018
019      return pResult;
020    }
```

下面就是使用 TMongoQuery 搜尋"老夫子"這筆 Document 的結果，我們可以看到直接使用 TMongoQuery 物件也可以成功搜尋我們需要的 Document：


此外我們也可以對 Document 進行排序，我們只需要在 TMongoQuery 的 Sort 特性值中寫入排序條件即可，例如現在我們要把所有的 Document 以 Age 欄位排序，那我們可以在 Sort()特性值中寫入：

```
{"age" : 1}
```

上面的數值 1 代表要以升冪來排序，如果是要以降冪來排序就傳入-1：

```
{"age" : -1}
```

現在就可以在主表單中加入另一個” 使用 TMongoQuery 排序”按鈕並寫入如下的程式碼：

```
void __fastcall TfmMainForm::Button4Click(TObject *Sender)
{
  Memo2->Lines->Clear();
```

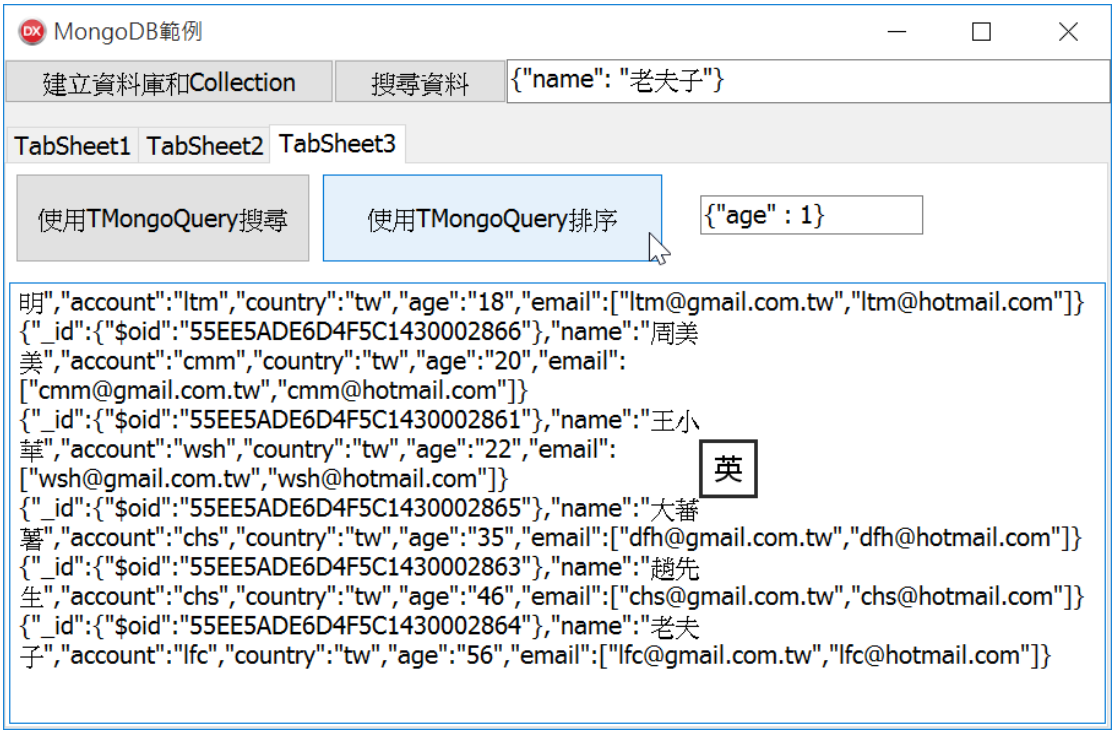
```
  _di_IMongoCursor IFDCrs = SortDocument(Edit1->Text, Edit2->Text);
  while (IFDCrs->Next() )
  {
  String sData = IFDCrs->Doc->AsJSON;
  Memo2->Lines->Add(sData);
  }
}
```

**SortDocument()**方法同樣建立 **TMongoQuery** 物件，並於下面的 **013~017** 行設定排序條件，最後再呼叫 **Find()**方法：

```
001    _di_IMongoCursor TfmMainForm::SortDocument(const String
sMatch, const String sSort)
002    {
003      _di_IMongoCursor pResult = NULL;
004      TMongoQuery *pMQuery = new TMongoQuery(FDEnv);
005      try
006      {
007        pMQuery
008          ->Sort()
009          ->Clear()
010          ->Append(sMatch)
011        ->End();
012
013        pMQuery
014          ->Sort()
015          ->Clear()
016          ->Append(sSort)
017        ->End();
018        pResult =
FDCon->Collections["FireDACDemoDB"]["FireDACColletions"]->Find(p
MQuery);
019      }
020      __finally
021      {
022        delete pMQuery;
023      }
```

```
024
025      return pResult;
026    }
```

下面就是執行的結果，我們可以看到所有的 Document 都是以 Age 欄位排序了：




瞭解了上面的內容後讀者應該就可以使用 TFDConnection 和 TFDMongoQuery 等 FireDAC 的元件來開發 MongoDB 的應用程式了，而且在瞭解了如何直接使用 FireDAC 封裝的 MongoDB API 類別後，在讀者進一步閱讀 MongoDB 的書籍時現在也知道了如何使用封裝的 MongoDB API 類別來處理 MongoDB 的資料了。

# 資料繫結篇(Data Binding)

# 第7章 開發第1個即時資料繫結應用程式

C++Builder 在 XE2 開始進入跨平台的開發領域，能夠同時使用 C++Builder 程式語言開發 Win32，Win64，MacOS 和 iOS。由於 VCL 框架只能使用在 Win32 和 Win64 平台，因此如果開發人員需要開發跨平台的圖形使用者介面應用程式，那麼必須使用新的 FireMonkey 框架，由於許多的 FireMonkey 框架的控制項都是動態產生的，因此當開發人員需要結合 FireMonkey 控制項和資料功能時，傳統像 VCL 的資料感知元件的使用方式並不適合使用於 FireMonkey 控制項，因為 VCL 的控制項是屬於靜態控制項，因此為了讓 FireMonkey 控制項也能夠像 VCL 的資料感知元件提供類似的資料繫結能力，C++Builder 必須提供另外一種動態繫結資料的能力以便和 FireMonkey 控制項共同使用在一起。



即時資料繫結(Live Data Binding)是 C++Builder XE2 全新推出的資料存取功能，它允許 FireMonkey 控制項使用動態的繫結運算式(Binding Expression)來繫結特定的資料來源，一旦繫結的資料來源資料有變化時，FireMonkey 控制項使用動態的繫結運算式可以自動重新運算並且顯示新的運算結果的資料。由於即時資料繫結不但能夠繫結資料庫，也能夠繫結其他的資料來源，例如讓控制項繫結到其他的控制項，而且能夠使用繫結運算式來進行運算，因此提供了比 VCL 資料感知元件更有彈性的能力。

簡單的說，XE2 的即時資料繫結提供了下面的功能

- 提供執行函數式的服務以及指定運算結果給目的控制項的能力
- 提供資料型態自動轉換的能力
  - 例如當函數式執行完畢之後要指定給目的控制項時，即時資料繫結能夠根據指定的目的控制項的資料型態自動轉換繫結運算式的運算結果為正確的資料型態再指定給目的控制項

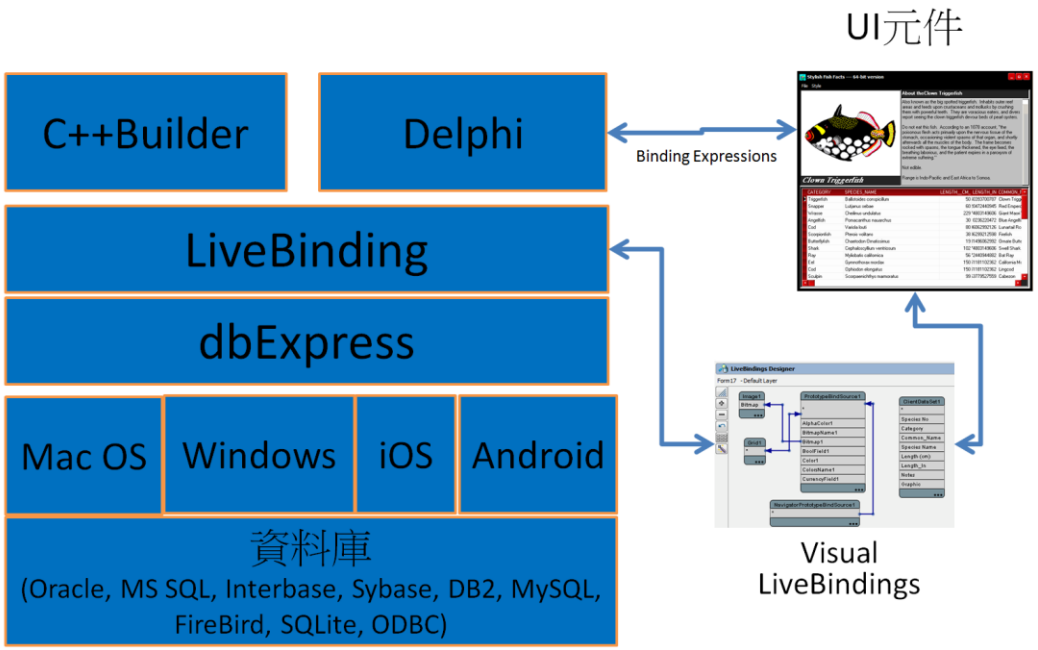
- 即時資料繫結提供運算範圍的能力，這是指當執行繫結運算式時，開發人員可以指定繫結運算式中使用的變數，方法，運算元和特性值等是在什麼範圍中定義的。例如即時資料繫結提供了 TBindScopeDB 元件來指定繫結運算式中使用的變數，方法，運算元和特性值是 TBindScopeDB 元件繫結的資料庫。即時資料繫結也提供 TBindScopeComponent 元件來指定繫結運算式中使用的變數，方法，運算元或特性值是 TBindScopeComponent 元件繫結的元件中定義的。

- 即時資料繫結提供自動更新的能力，當函數式中的資料有變化時，繫結運算式能夠自動重新運算。

不過由於 XE2 的即時資料繫結需要開發人員直接使用繫結運算式來撰寫，因此造成開發人員在學習使用即時資料繫結技術時產生困難。TOKYO 為了幫助開發人員降低學習繫結運算式的困難，因此提供了視覺化即時資料繫結(Visual LiveBindings)技術，讓開發人員藉由使用視覺化的設計家來自動產生繫結運算式，讓開發人員能夠快速完成 FireMonkey 或 VCL 的資料庫應用程式的開發工作。

雖然即時資料繫結技術可以為 FireMonkey 或 VCL 應用程式連結資料來源，但即時資料繫結在存取資料來源時仍然是使用 dbExpress 框架，由於 dbExpress 是跨平台的資料存取引擎，因此使用即時資料繫結技術的應用程式也能夠執行在不同的平台中。下圖說明了即時資料繫結，視覺化即時資料繫結，繫結運算式以及 dbExpress 框架之間的關係:





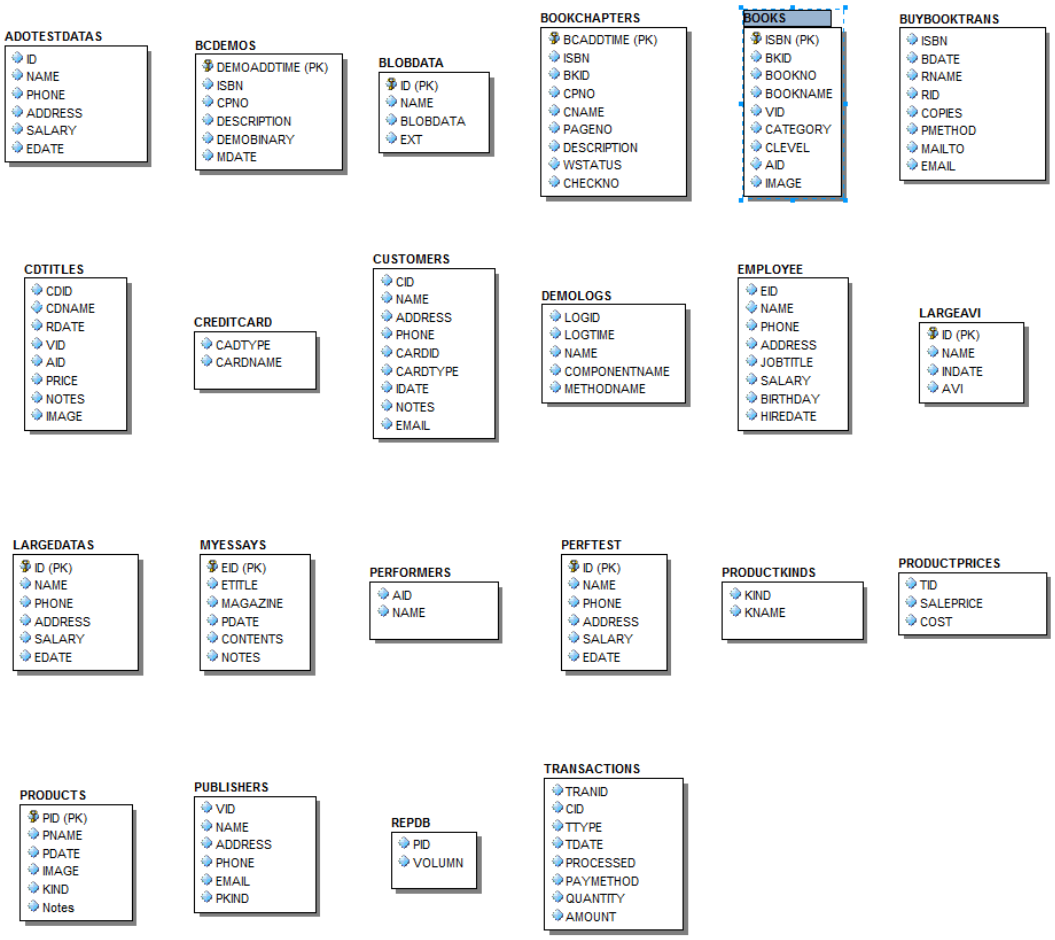
對於使用 TOKYO 開發 FireMonkey 和 VCL 的資料庫應用程式的開發人員來說，學習即時資料繫結和 dbExpress 框架都是必要的，在本書的第一部份將說明如何使用即時資料繫結技術開發 FireMonkey 的資料庫應用程式，第 2 部份將說明如何使用 dbExpress 框架。

雖然即時資料繫結主要是為了 FireMonkey 框架開發出來的資料存取技術，但由於即時資料繫結技術是獨立的框架，因此即時資料繫結也可以使用在 VCL 應用程式中，在本章中我們將先說明如何開發即時資料繫結應用程式，稍後的章節再逐漸討論即時資料繫結框架較為深入的主題和技術。

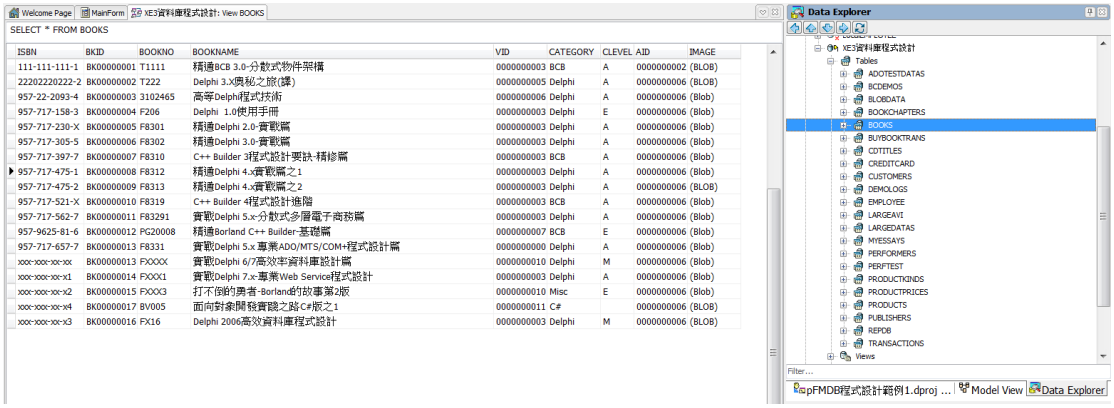
**從本章這裡之後討論資料繫結技術中同時使用了 dbExpress 和 FireDAC 2 種不同的技術展示資料繫結技術可和任一 C++Builder 的資料存取技術共同使用，如果讀者不想使用 dbExpress 技術，那可以自行根據前面章節對於 FireDAC 的瞭解把 dbExpress 元件都置換成使用 FireDAC 元件即可。**

# 7-1 開發第一個 FireMonkey 資料庫應用程式

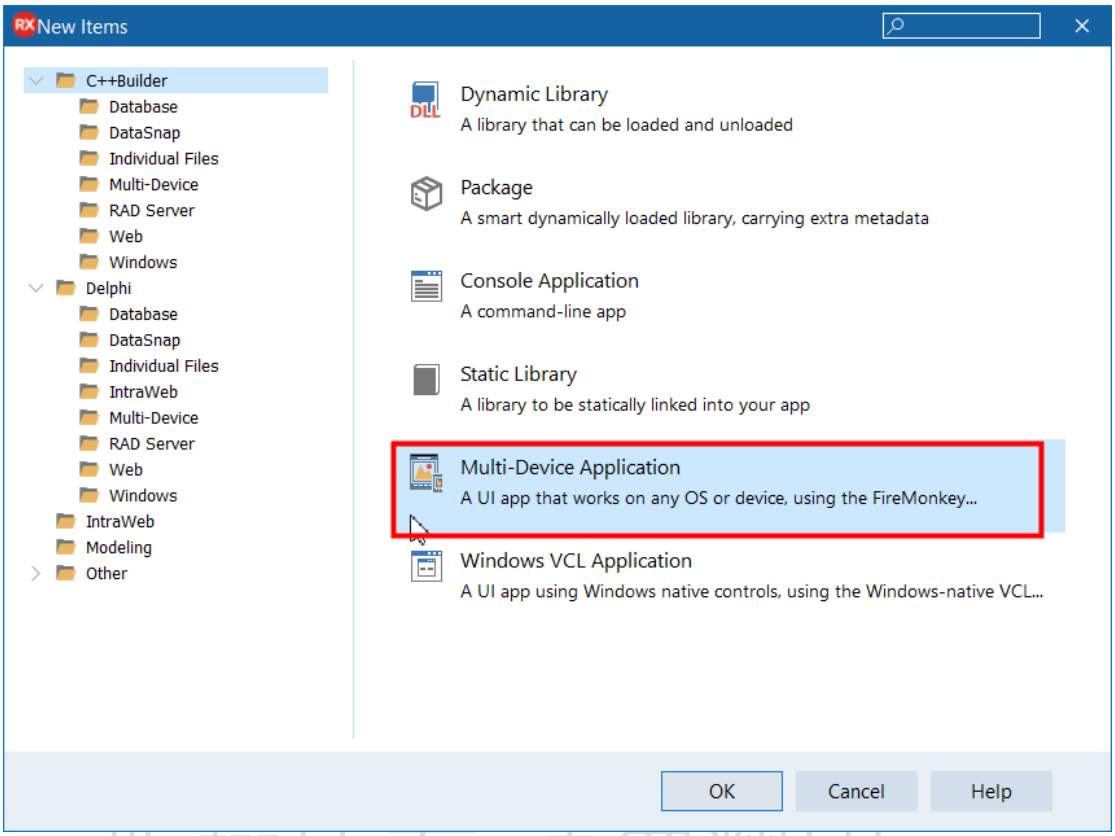
本書將使用 InterBase 做為範例資料庫，在其中有數個資料表之間擁有 Master/Detail 的關係，在說明如何使用視覺化即時資料繫結技術逐漸開發 FireMonkey 資料庫應用程式的流程中，將一一的帶入使用這些資料表。不過讓我們先從簡單的開發工作開始說明如何使用視覺化即時資料繫結。




讓我們先從 BOOKS 範例資料表開始，下面是 BOOKS 資料表中的資料，現在就讓我們使用即時資料繫結開發一個 FireMonkey 資料庫應用程式，讓讀者瞭解 TOKYO 的視覺化即時資料繫結技術是多麼的容易使用。

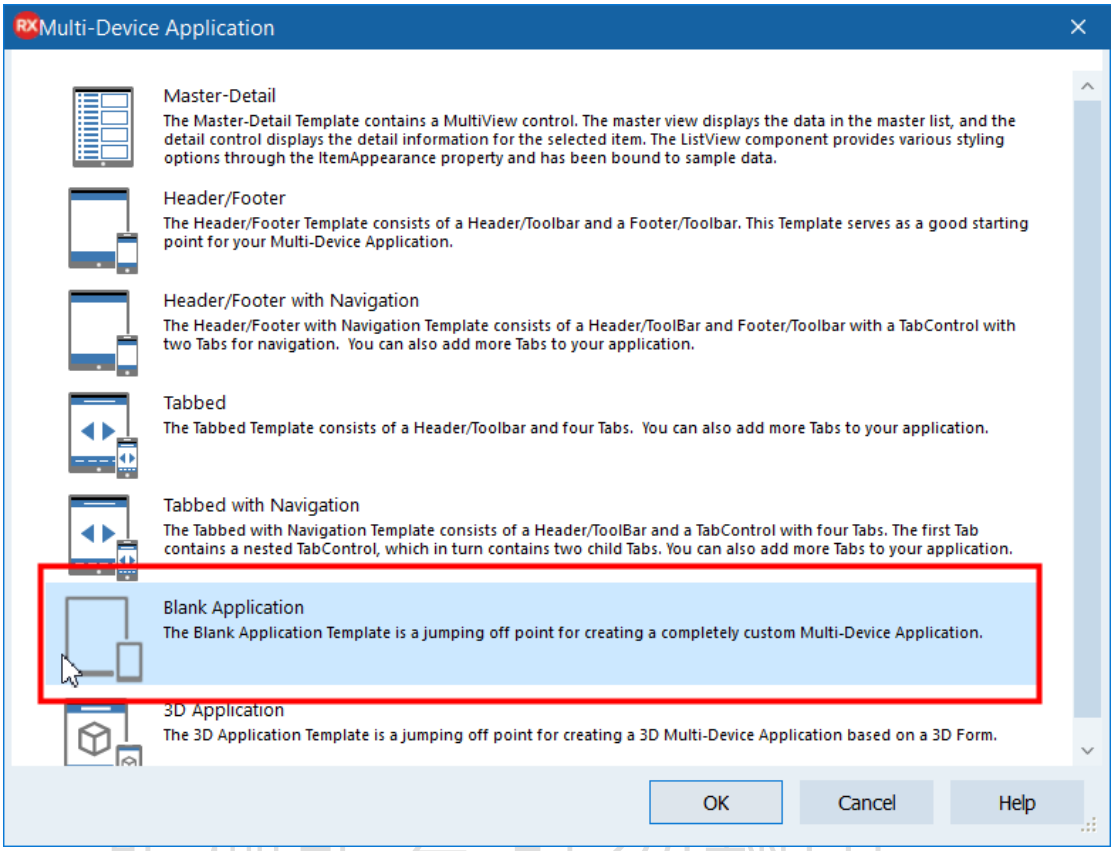



請先建立一個 Multi-Device 桌面應用程式專案:



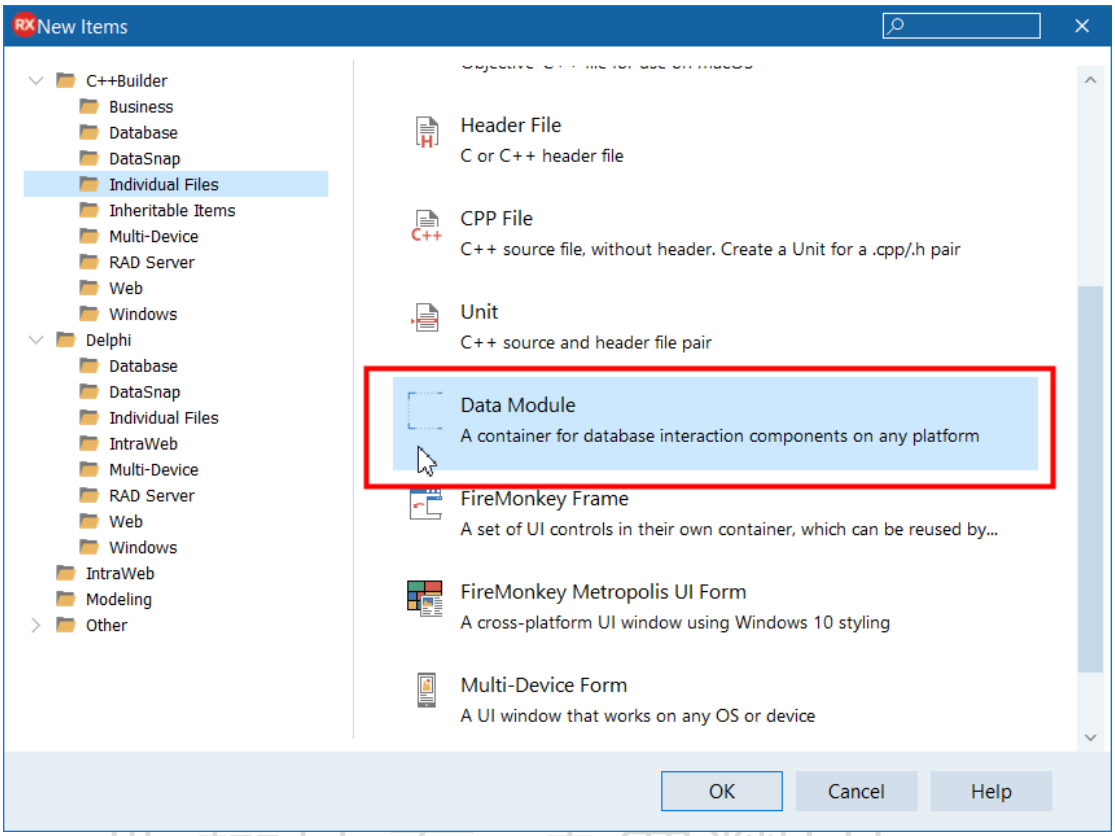



再選擇建立『Blank Application』型態的專案 ：



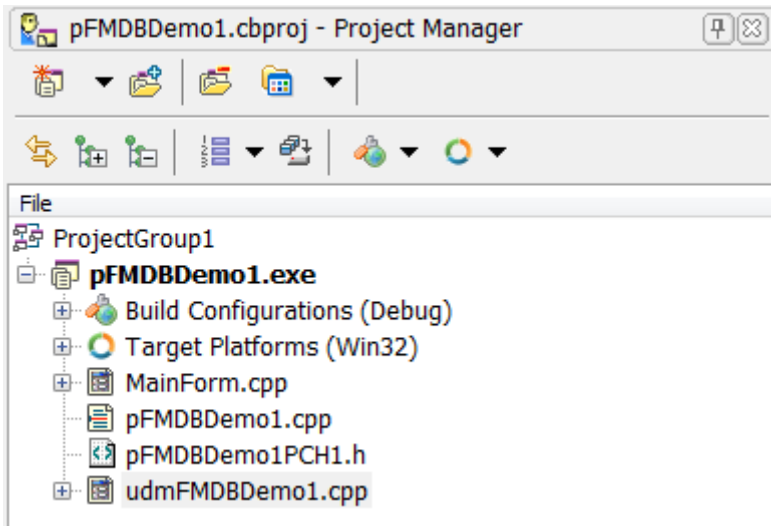



接著在專案中建立一個資料模組:





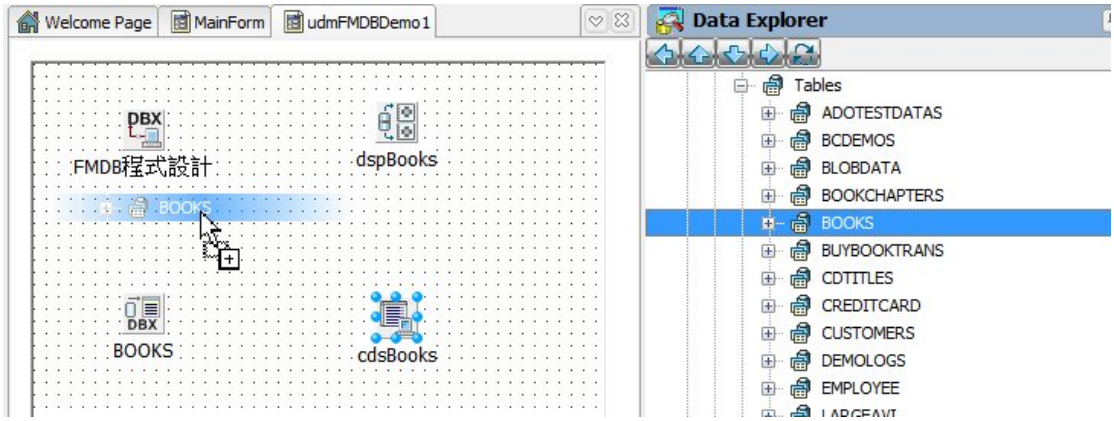

最後儲存此 FireMonkey 桌面應用程式專案如下所示:



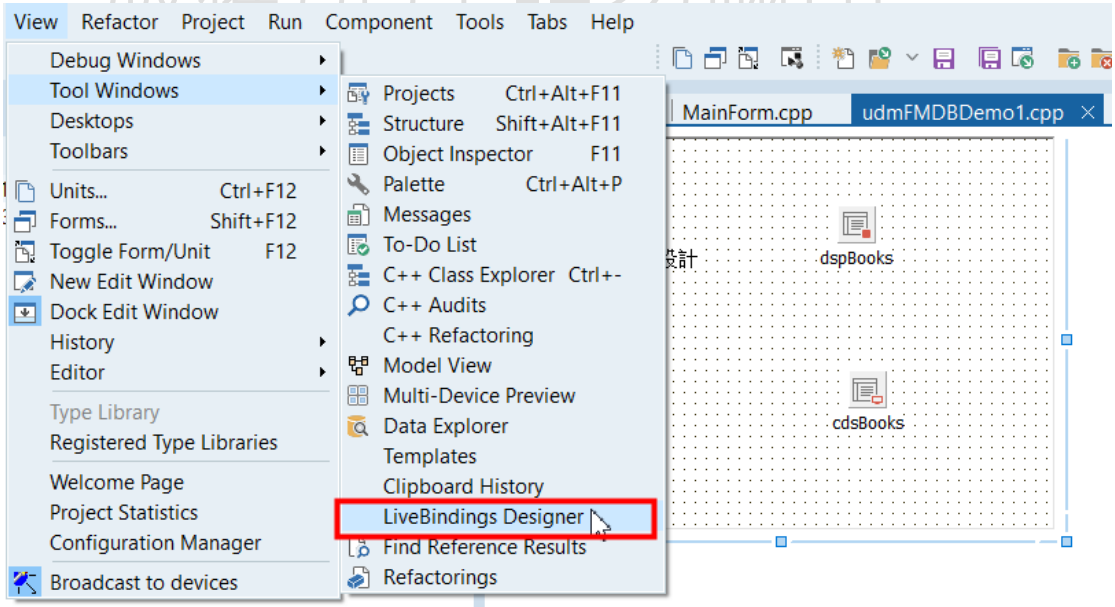

開啟資料模組從 Data Explorer 頁面中拖曳 BOOKS 資料表到資料模組中，資料模組即會自動產生 TSQLConnection 和 TSQLDataSet 元件，接著再於資料模組中放入 TDataSetProvider 和 TClientDataSet 元件，並且設定它們的特性值如下:

| 元件 | 元件名稱 | 設定的特性值 |
|---|---|---|
| TDataSetProvider | dspBooks | DataSet = BOOKS |
| TClientDataSet | cdsBooks | ProviderName = dspBooks |
| TClientDataSet | cdsBooks | Active = True |

此時資料模組應該如下所示:

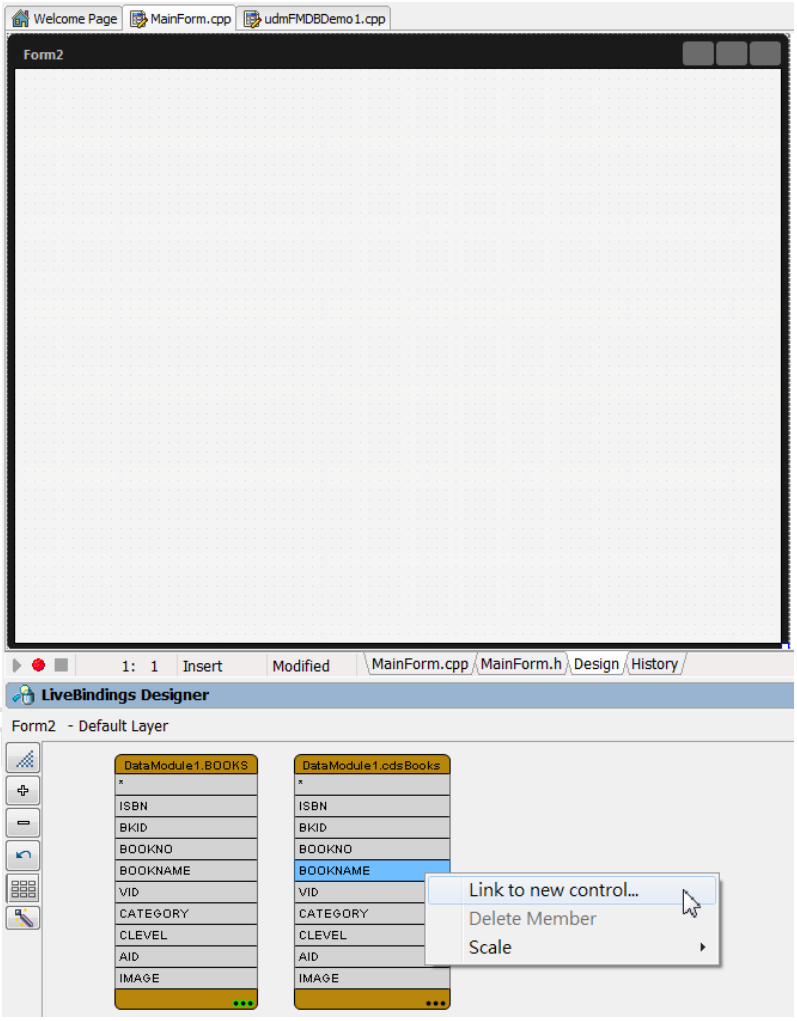



回到主表單，點選 File|Use Unit…功能表使用資料模組，接著點選 View|LiveBindings Designer 啟動視覺化即時資料繫結設計家，如下所示:

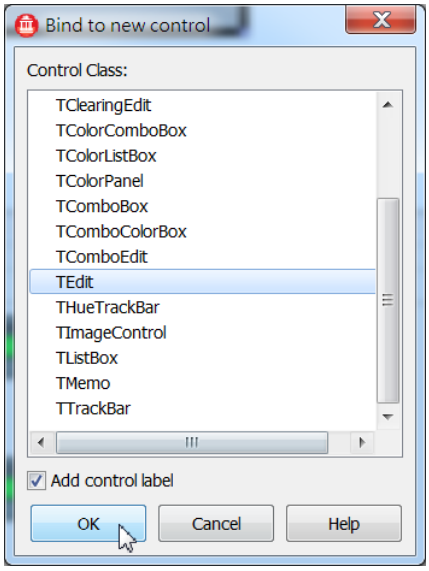



由於主表單使用了資料模組，因此讀者可以從下圖中看到視覺化即時資料繫結設計家中顯示了資料模組中的 cdsBooks 實體(Entity)，假設現在我們希望 cdsBooks 中的 BOOKNAME 欄位顯示在主表單中，那麼我們可以在視覺化即時資料繫結設計家中點選

cdsBooks 的 BOOKNAME 欄位，再點選滑鼠右鍵，就會出現一個快顯功能表，請選擇『Link to new control...』 如下圖所示:

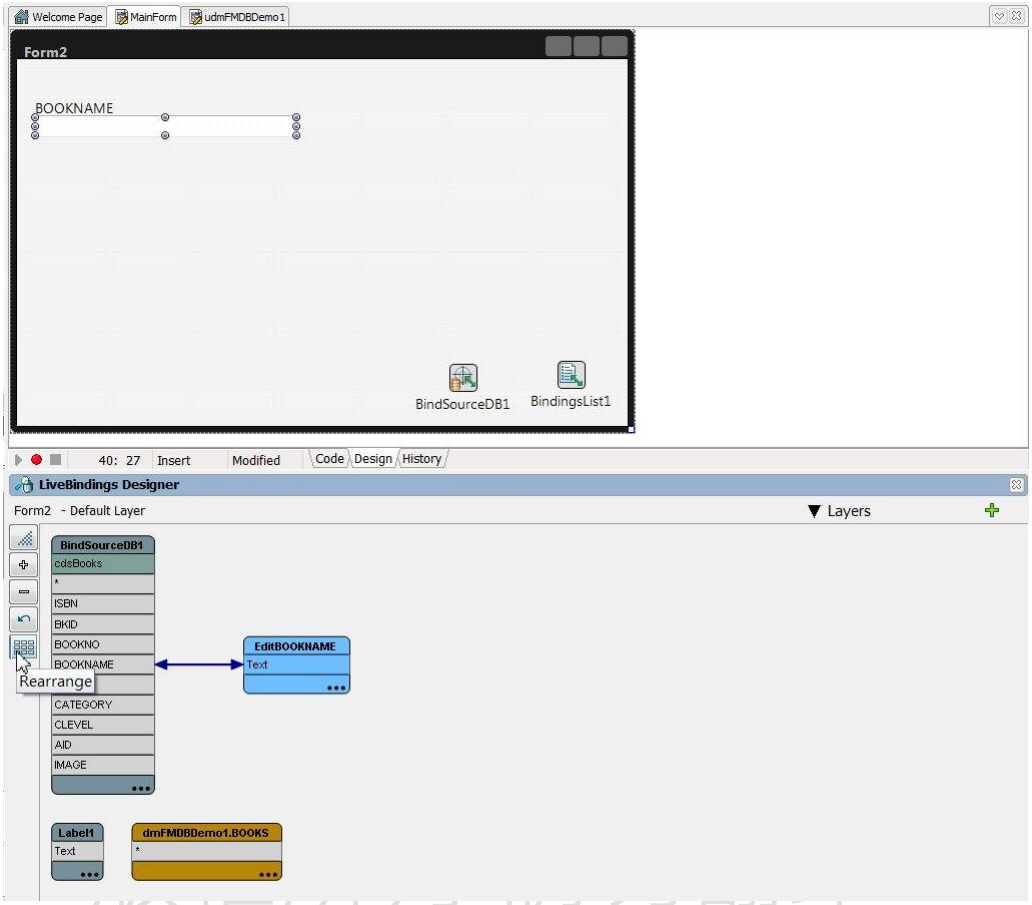



此時視覺化即時資料繫結精靈會顯示如下的對話盒，讓您選擇希望繫結 BOOKNAME 欄位的控制項，您可以瀏覽對話盒中出現的控制項，現在讓我們選擇 TEdit 元件讓 BOOKNAME 欄位的數值顯示在 TEdit 元件中，並且點選 OK:

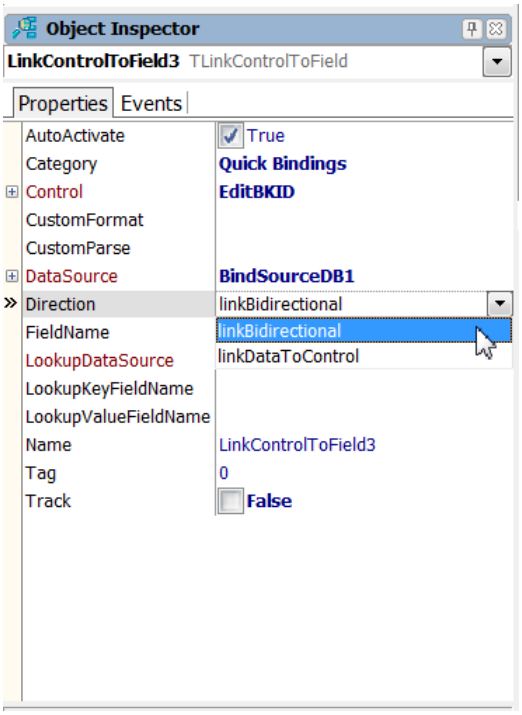



此時視覺化即時資料繫結精靈會顯示如下的對話盒，讓您選擇希望繫結 BOOKNAME 欄位的控制項，您可以瀏覽對話盒中出現的控制項，現在讓我們選擇 TEdit 元件讓 BOOKNAME 欄位的數值顯示在 TEdit 元件中:

接著讀者就可以在主表單中看到一個 TEdit 元件，而且在視覺化即時資料繫結設計家中讀者可以看到 cdsBooks 實體和一個新出現名為 EditBOOKNAME 的實體之間擁有一個雙向箭頭的實線，雙向箭頭的實線代表可雙向更新資料的關係，也就是說 BOOKNAME 欄位的數值可以出現在此 TEdit 元件中，如果使用者修改 TEdit 元件中顯示的 BOOKNAME 欄位數值，那麼這也會更新回 BOOK 資料表中。




要改變實體之間箭頭的方向，例如如果我們希望 BOOKNAME 欄位的數值在 TEdit 元件中只能顯示而不能修改，那麼開發人員可以在視覺化即時資料繫結設計家中點選雙向的箭頭，再於物件檢視器中修改它的 Direction 特性值，例如如果如下圖改變 Direction 特性值為『linkDataToControl』，那麼 TEdit 元件中的 BOOKNAME 欄位值就是唯讀的。




主表單中除了 TEdit 元件之外，也會自動產生 TBindSourceDB 和 TBindingList 元件。TBindSourceDB 元件是連結到資料模組中 cdsBooks 的元件，它負責協調和管理資料來源之中的資料，而 TBindingList 元件則負責產生和執行其包含的繫結運算式，而開發人員使用視覺化即時資料繫結設計家自動產生的繫結運算式也管理在 TBindingList 元件中。

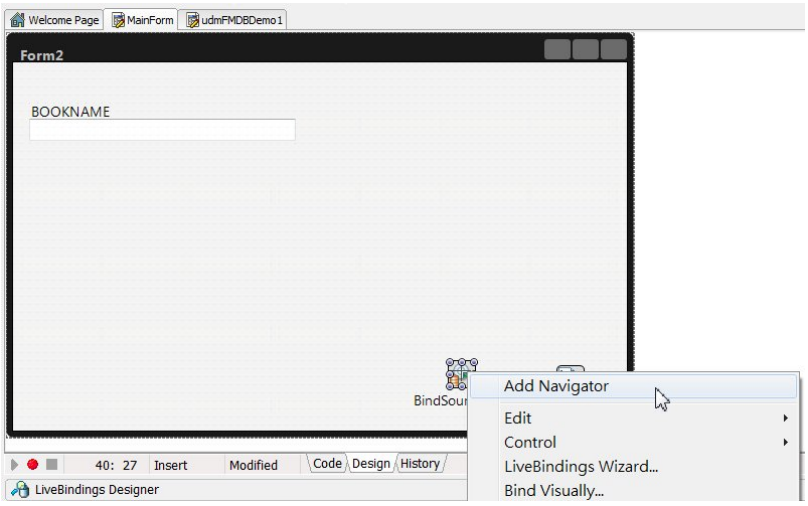


現在如果您資料模組中的 cdsBooks 元件是開啟的，那麼您就可以在主表格中看到 TEdit 元件顯示了 BOOKNAME 欄位的數值。

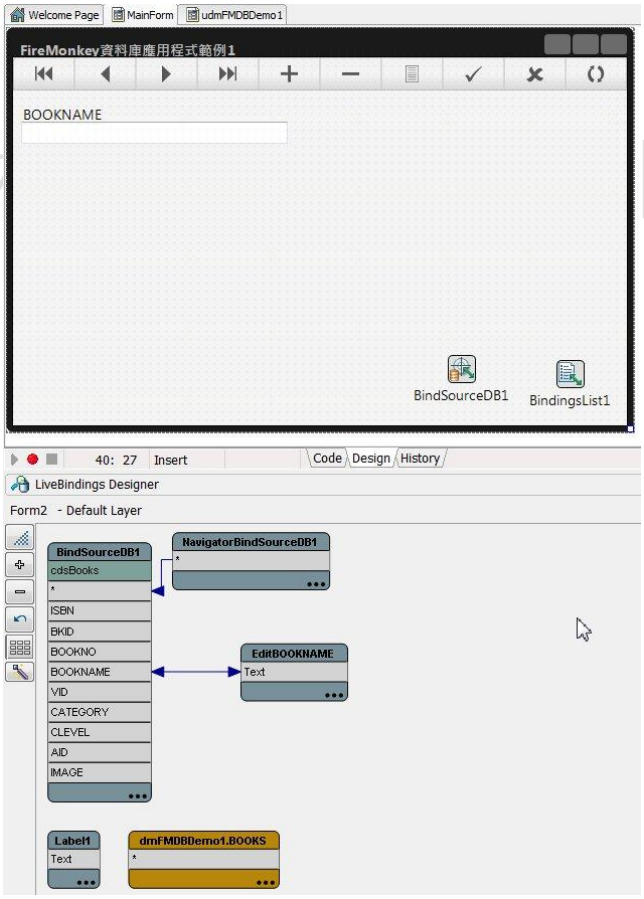
現在讓我們在主表格中加入一個 Navigator 讓我們能夠在主表格中來回瀏覽 BOOKS 資料表中的資料，請點選主表單中的 TBindSourceDB 元件，再點選滑鼠右鍵，從快顯功能表中選擇『Add Navigator』選項，如下所示:

此時主表單中就會出現 TBindNavigator 元件，請設定它的特性值如下:

| 元件 | 特性值 |
|---|---|
| TBindNavigator | Aligh = alTop |

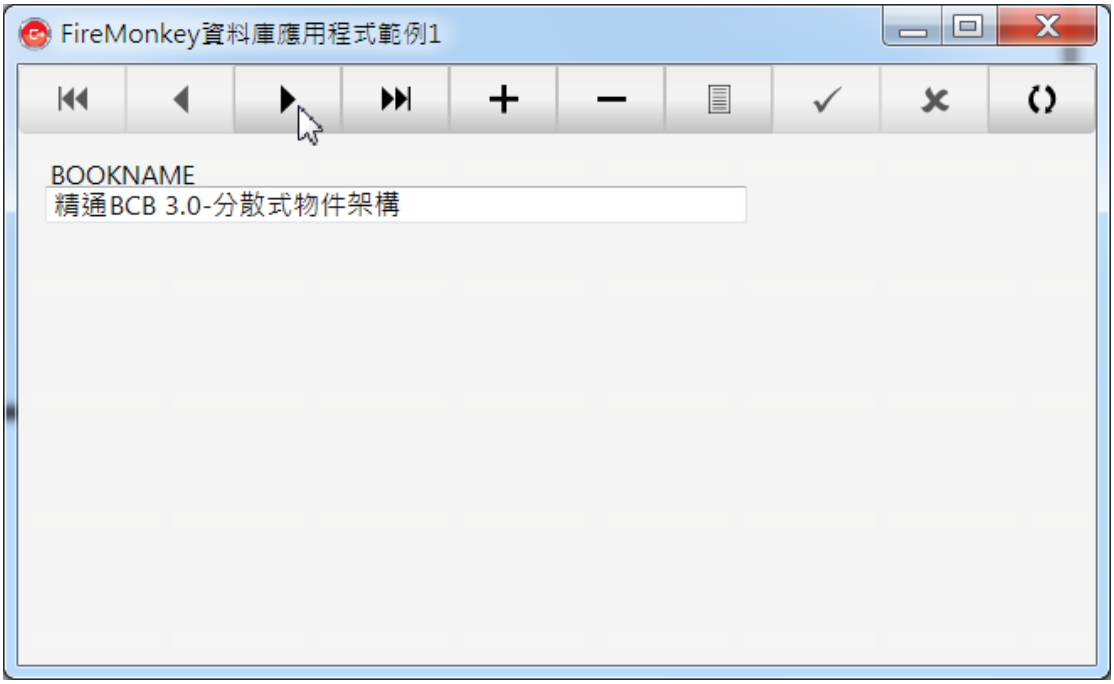



現在主表單應該看起來如下所示，第一個範例 FireMonkey 資料庫應用程式也準備好執行了:




請按下 Shift+Ctrl+F9 執行範例應用程式，就可以看到如下的結果畫面，BOOKS 資料表中的 BOOKNAME 欄位的數值果然可以出現在 TEdit 元件，而且如果點選主表

單上方的 TBindNavigator 元件中的按鈕也能夠在資料之間瀏覽了。現在我們已經成功的使用視覺化即時資料繫結完成了第一個範例 FireMonkey 資料庫應用程式。



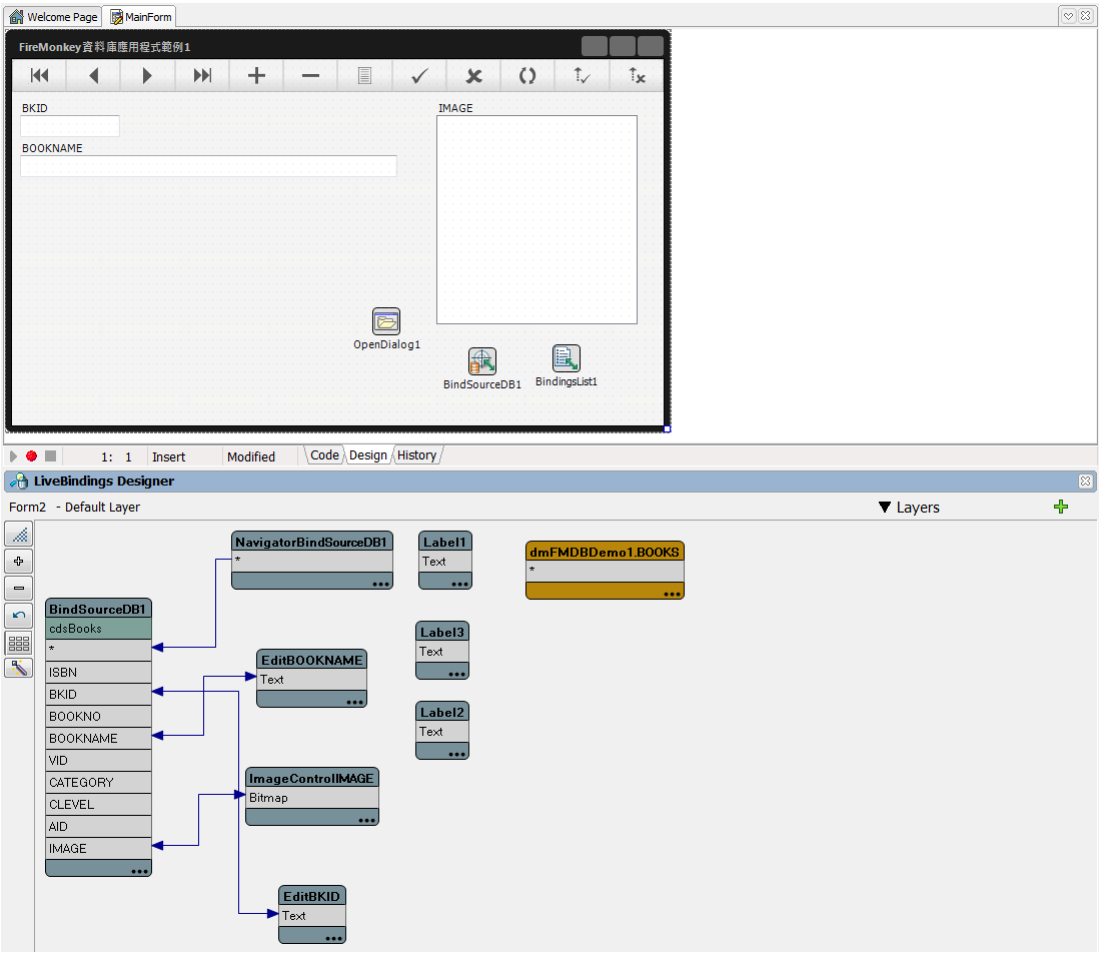



現在請結束範例 FireMonkey 資料庫應用程式回到主表單，再設定 TBindNavigator 元件的 VisibleButtons 特性值如下:

| 元件 | 特性值 |
|---|---|
| TBindNavigator | VisibleButtons = [nbFirst,nbPrior,nbNext,nbLast,nbInsert,nbDelete, nbEdit,nbPost,nbCancel,nbRefresh, nbApplyUpdates,nbCancelUpdates] |

設定 TBindNavigator 元件的 VisibleButtons 特性值如上所述就會加入 ApplyUpdates 和 CancelUpdates 按鈕，讓 TBindNavigator 元件能夠把資料更新回 cdsBooks 代表的 BOOKS 資料表中。

接著再使用視覺化即時資料繫結設計家，如同前面繫結 BOOKNAME 欄位到 TEdit 元件一樣的方式把 BKID 欄位繫結到另外一個 TEdit 元件以及把 IMAGE 欄位繫結到 TImageControl 元件，如下所示:

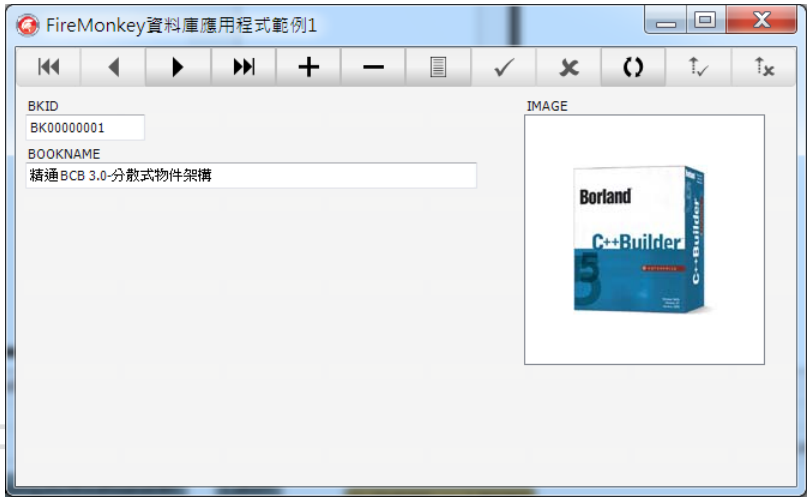



現在主表單能夠顯示兩個額外的 BOOKS 資料表的欄位資料了，請注意 cdsBooks 實體和 TImageControl 元件之間是雙向箭頭，這代表我們可以更改 TImageControl 元件中顯示的圖形而即時資料繫結會把圖形資料自動更新回 BOOKS 資料表中，我們當然也可以直接使用 dbExpress 框架更新圖形資料，請在主表單中加入一個 TOpenDialog 元件並且在 TImageControl 元件的 OnMouseDown 事件處理函式中撰寫如下的程式碼:

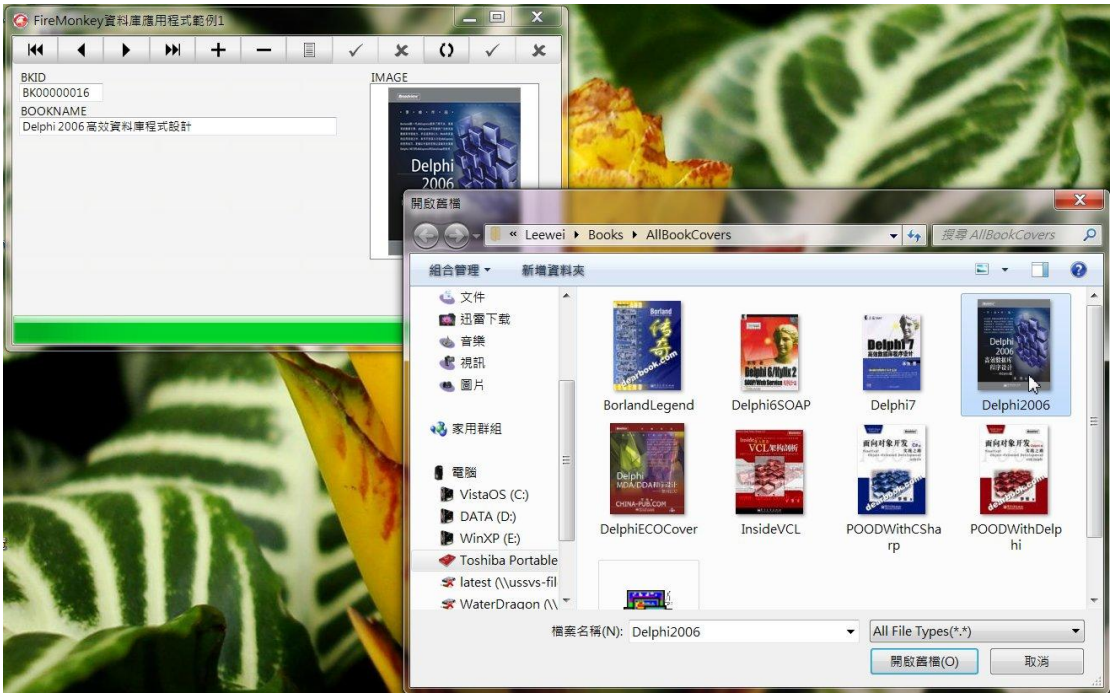
```
void __fastcall TForm2::ImageControlIMAGEMouseDown(TObject *Sender,
TMouseButton Button,
        TShiftState Shift, float X, float Y)
{
  if (Button == TMouseButton::mbRight)
  {
    if (OpenDialog1->Execute())
    {
      dmFMDBDemo1->cdsBooks->Edit();
```

```
dmFMDBDemo1->cdsBooksIMAGE->LoadFromFile(OpenDialog1->FileName);
      dmFMDBDemo1->cdsBooks->Post();
      dmFMDBDemo1->cdsBooks->ApplyUpdates(0);
    }
  }
}
```

再執行範例 FireMonkey 資料庫應用程式，就可以看到 TBindNavigator 多了個 ApplyUpdates 的按鈕，BKID 和 IMAGE 欄位的資料都會出現了，如下所示:

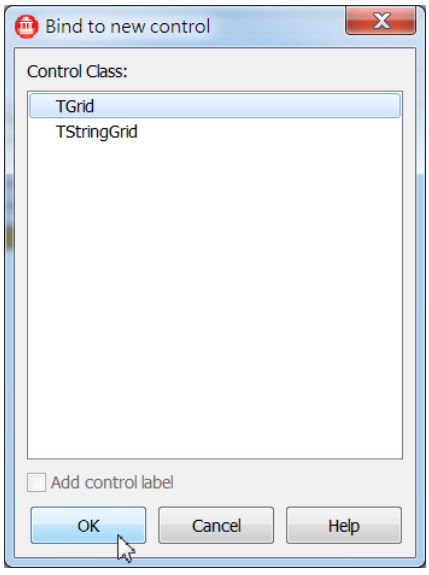



如果在 TImageControl 中點選滑鼠右鍵就可以看到如下圖會出現對話盒讓您載入一個新的圖形並且載入到 BOOKS 資料表中的 IAMGE 欄位中，再點選上方 Navigator 中的 ApplyUpdates 按鈕之後圖形的資料就會真正的更新回資料庫之中了。




讓我們繼續在主表單中加入一些元件來顯示 BOOKS 資料表更多的資料，首先讓我們加入一個 TGrid 元件。請點選視覺化即時資料繫結設計家中 cdsBooks 樣例中的『*』欄位然後再點選滑鼠右鍵，從突顯式選單中選擇『link to new control』選項，視覺化即時資料繫結精靈便會顯示如下的對話盒，請雙擊選擇其中的 TGrid 元件:

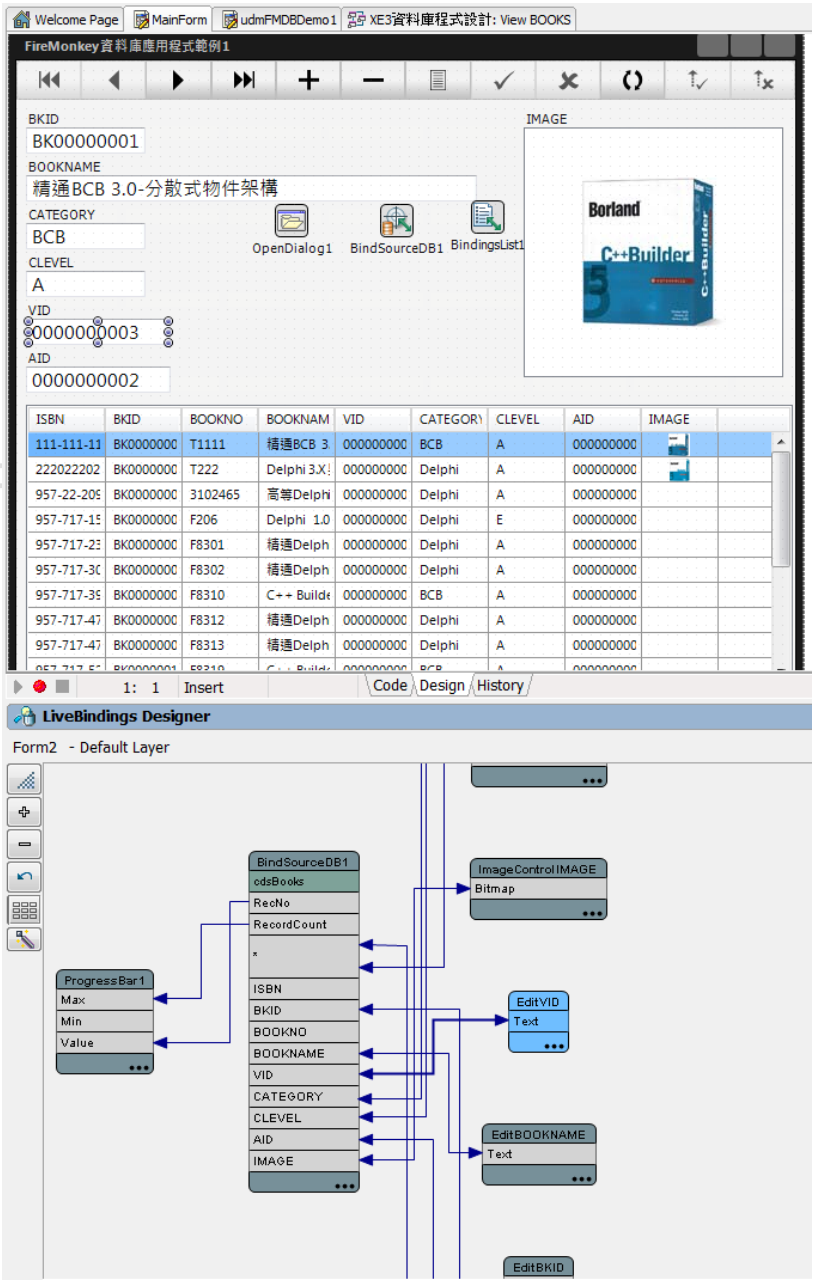




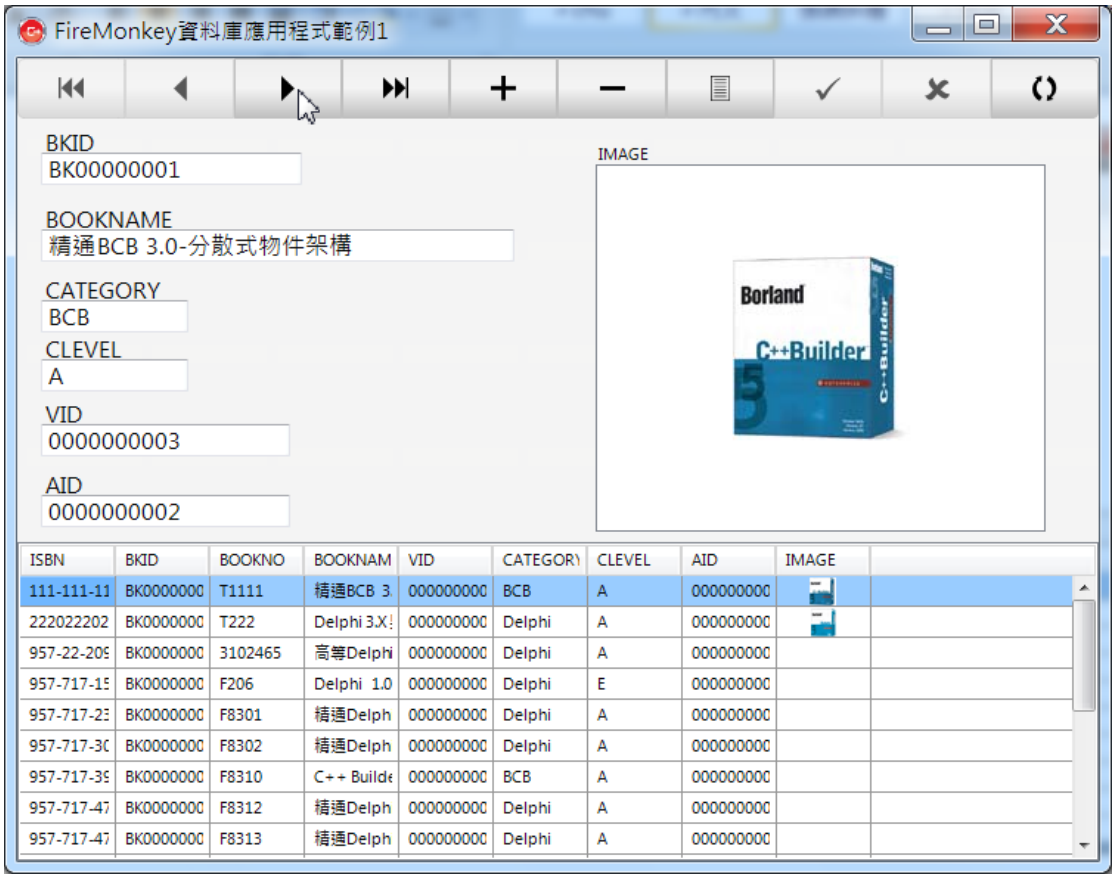

接著再於主表單中放入 2 個 TEdit 元件，分別命名為 EditVID 和 EditAID，於視覺化即時資料繫結設計家中先使用滑鼠左鍵點選 cdsBooks 實體中的 VID 欄位在不放開滑

鼠左鍵的情形下拖曳滑鼠到 EditVID 實體的 Text 特性以繫結這 2 者，此時在視覺化即時資料繫結設計家中這 2 者之間就會出現一個雙向箭頭的線條。

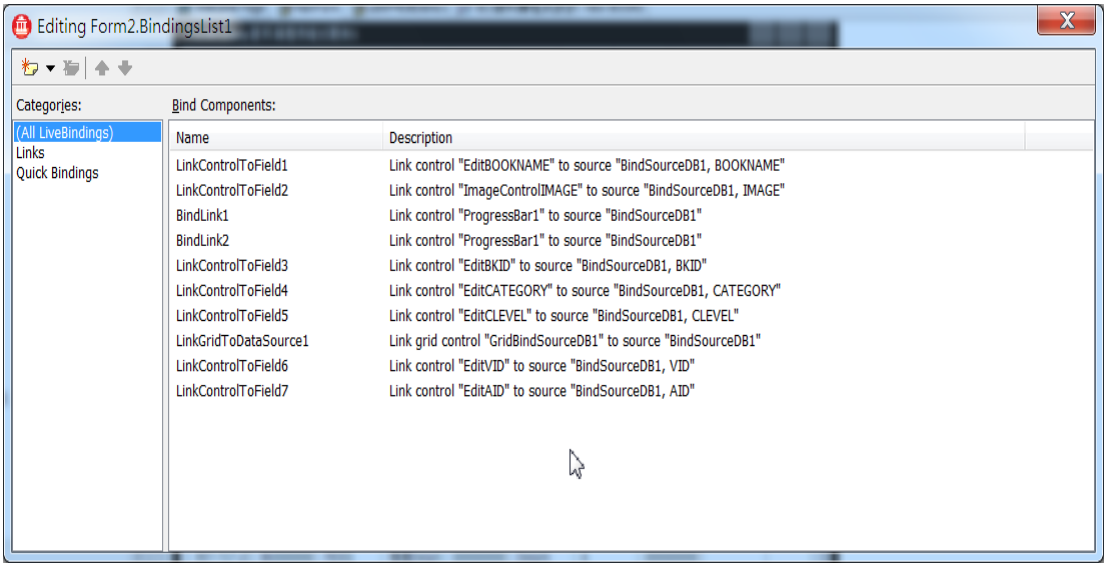
再使用同樣的方法繫結 cdsBooks 實體中的 AID 欄位和 EditAID 的 Text 特性，這 2 者之間也會出現一個雙向箭頭的線條，最後主表單和視覺化即時資料繫結設計家看起來如下所示:



現在您可執行此 FireMonkey 應用程式並且試著修改資料再藉由上方 Navigator 元件中的 ApplyUpdates 按照把資料更新回 BOOKS 資料表中，您會發現即時資料繫結果然可以真的把資料更新回 InterBase 資料庫中，現在您已經完成了 FireMonkey 第 1 個資料庫應用程式開發了，使用視覺化即時資料繫結技術真的很簡單，不是嗎?




在離開本小節之前請您注意在主表單中有一個 TBindingsList 元件，事實上在前面我們所有使用視覺化即時資料繫結設計家進行的繫結都會被自動轉換為繫結運算式儲存在此 TBindingsList 元件中，如果您雙擊主表單中的 TBindingsList 元件就會看到如下的編輯器:

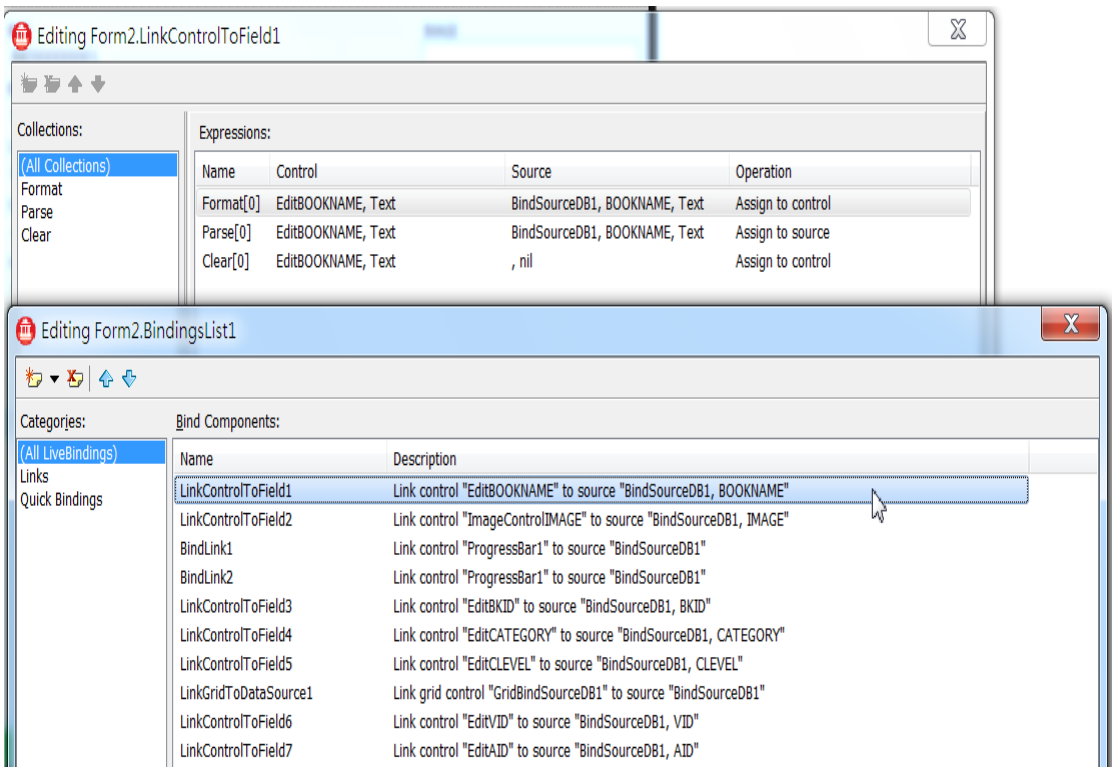



其中的內容就是前面所有在我們視覺化即時資料繫結設計家中進行的工作而自動產生的繫結運算式，您可以點選其中任何的繫結運算式並且在物件檢視器中觀察它，例如請雙擊編輯器中第 1 個繫結運算式您就可以看到這個繫結運算式的詳細資訊，在新出現的編輯器中您可以看到這個繫結運算式的意義就是把 BindSouceDB1(即 cdsBooks)中的 BOOKNAME 欄位的 Text 數值繫結到主表單中 EditBOOKNAME 這個 TEdit 元件的 Text 特性值中，這也就是說把 BOOKS 資料表中的 BOOKNAME 欄位的數值顯示在 EditBOOKNAME 元件中。



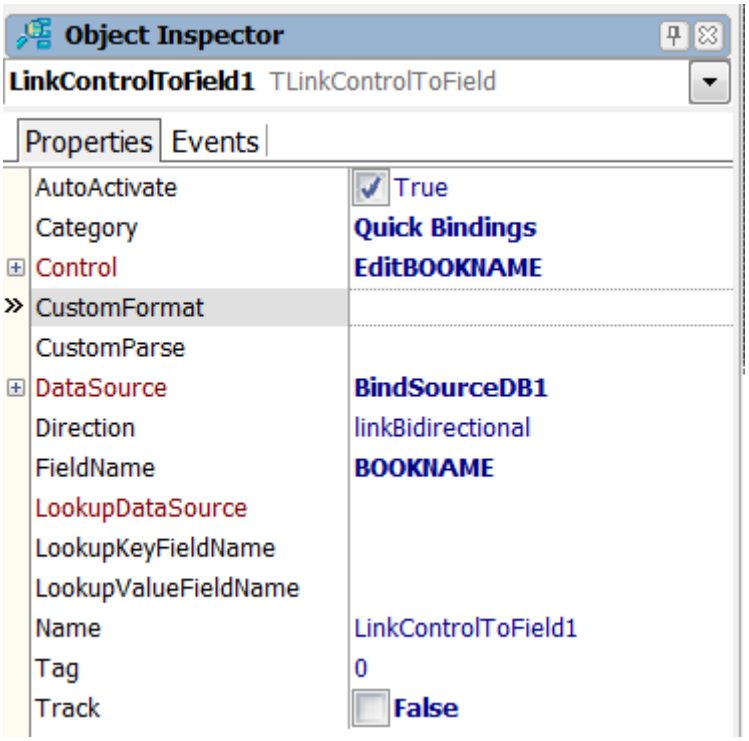
在下圖上方的第 2 個繫結運算式則是反相把 EditBOOKNAME 的 Text 特性值寫回 BindSouceDB1(即 cdsBooks)中的 BOOKNAME 欄位的 Text 數值，這也就是說把 EditBOOKNAME 元件中的數值更新回 BOOKS 資料表的 BOOKNAME 欄位。

至於第 3 個繫結運算式則是代表清除 EditBOOKNAME 中的數值。




如果我們點選上面編輯器中的 LinkControlToField1 物件並且觀察物件檢視器的話就可以看到如下的內容:





我們從上面的物件檢視器中可看到 LinkControlToField1 的 Category 特性值是 Quick Bindings，這種繫結物件是 TOKYO 才出現的新型態的繫結物件，它可快速繫結控制項和資料來源，這種型態的繫結物件主要是搭配視覺化即時資料繫結設計家使用的，但 TOKYO 尚有許多其他型態的繫結物件可讓開發人員使用，現在就讓我們使用其一個簡單型態的繫結物件:TBindLink。

## 7-1-1 淺嘗繫結運算式

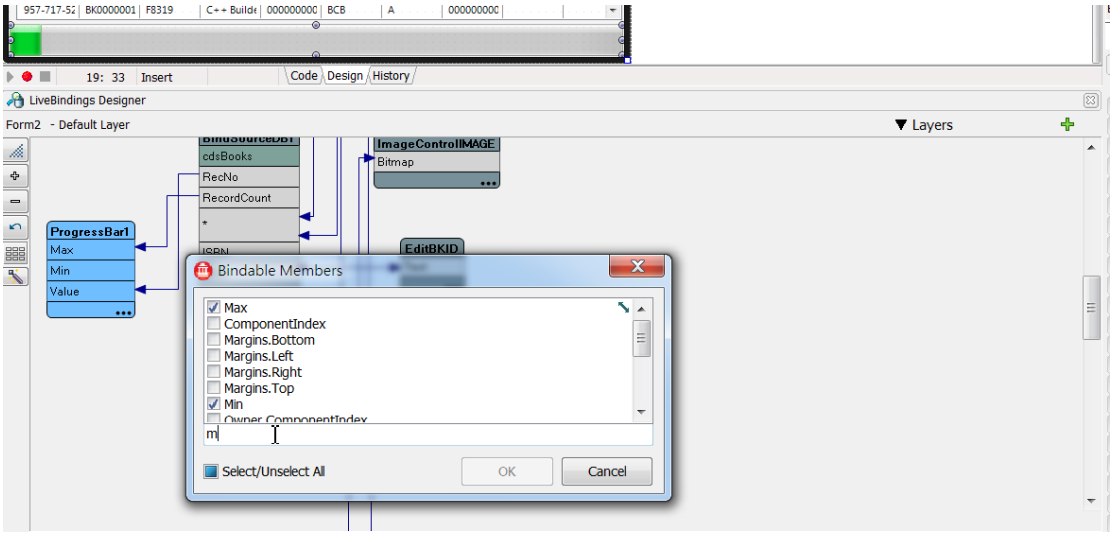
例如現在讓我們在主表單中加入一個 TProgressBar 在主表單的下方，我們希望這個 TProgressBar 能夠顯示目前 cdsBooks 中資料的相對位置，因此我們需要進行下列的 2 個繫結工作:

1. 把 cdsBooks 的 RecordCount 特性值繫結到 TProgressBar 的 Max 特性值

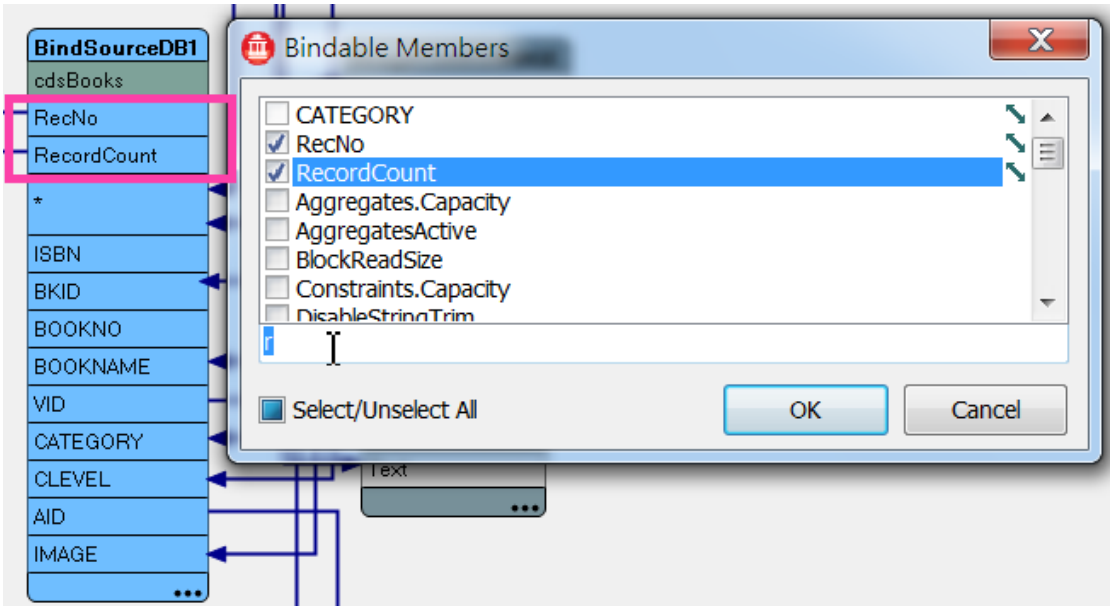
2. 把 cdsBooks 的 RecNo 特性值繫結到 TProgressBar 的 Value 特性值

請先在主表單中加入一個 TProgressBar 元件，設定它的 Align 特性值為 alBottom，回到視覺化即時資料繫結設計家點選其中 TProgressBar 實體右下方的 3 個黑點開啟 TProgressBar 可供繫結的成員對話盒，勾選其中的 Max 和 Value 特性值，再點選成員對話盒的 OK 按鈕之後 Max 和 Value 特性值就會出現在 TProgressBar 實體中，如下所示:

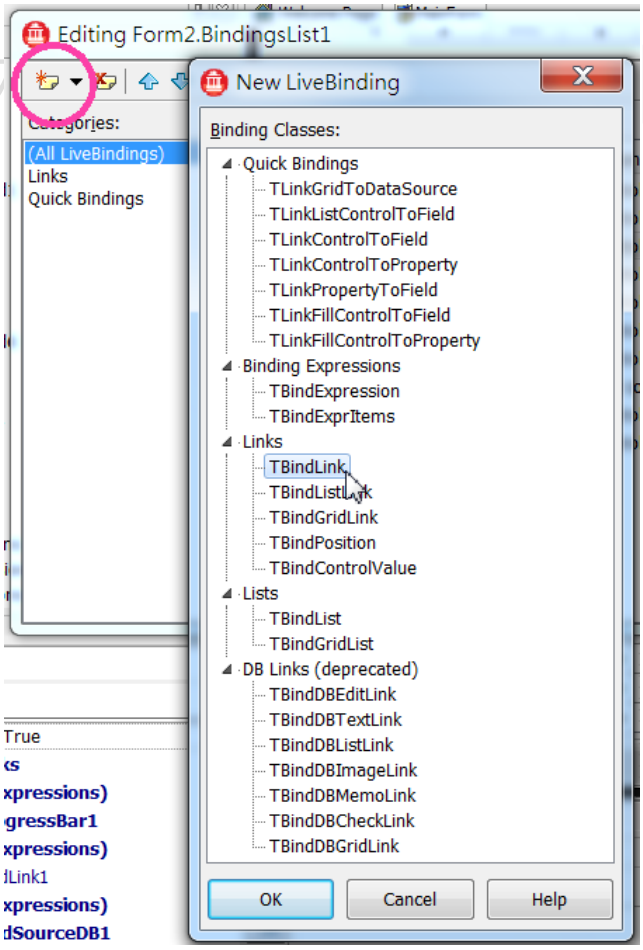




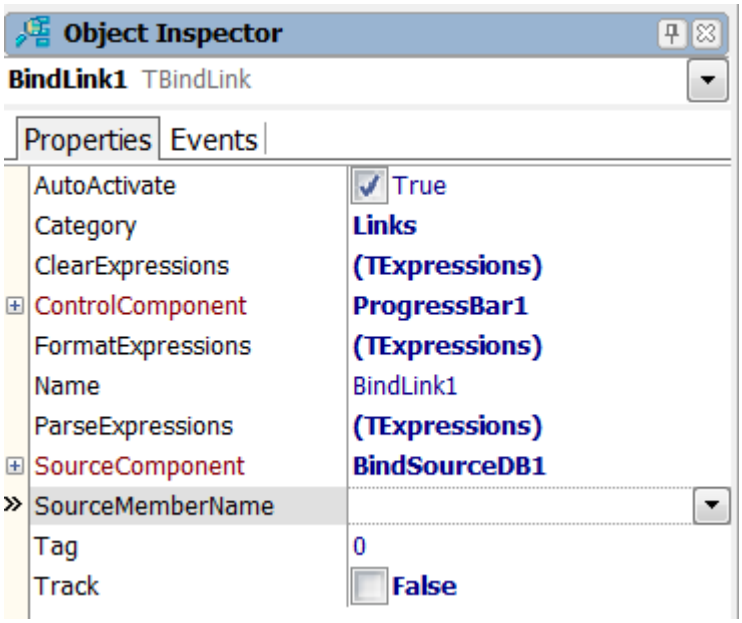

接著使用同樣的方法為 cdsBooks 實體加入 RecNo 和 RecordCount 特性如下所示:




雙擊主表單中的 TBindingsList 元件，再點選左上方的 New Binding 按鈕以加入新的繫結物件並且選擇加入 2 個 TBindLink 物件如下所示:



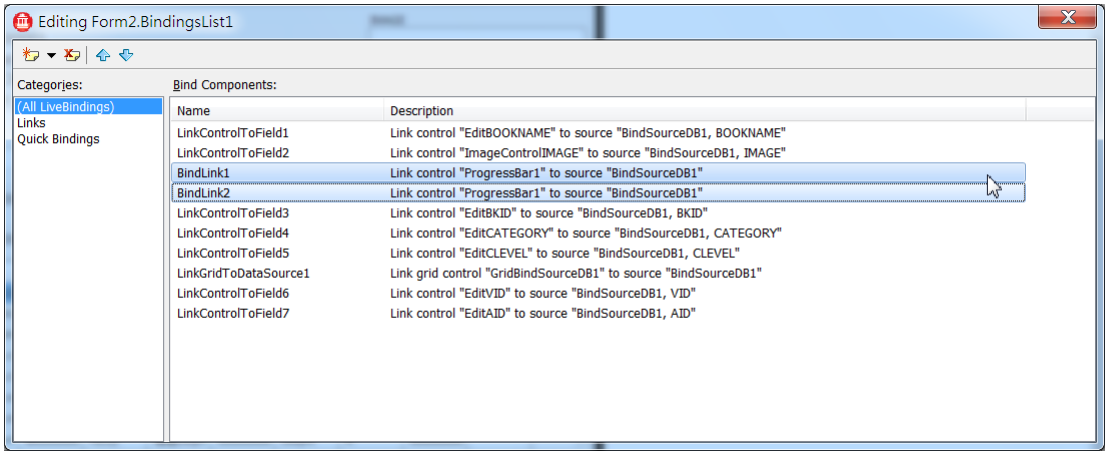
先點選 BindingsList1 編輯盒中的 2 個 TBindLink 物件，於物件檢視器中設定 2 個 TBindLink 物件的 SourceComponent 特性值為 BindSourceDB1，設定 ControlComponent 特性值為 TProgressBar1，如下所示:



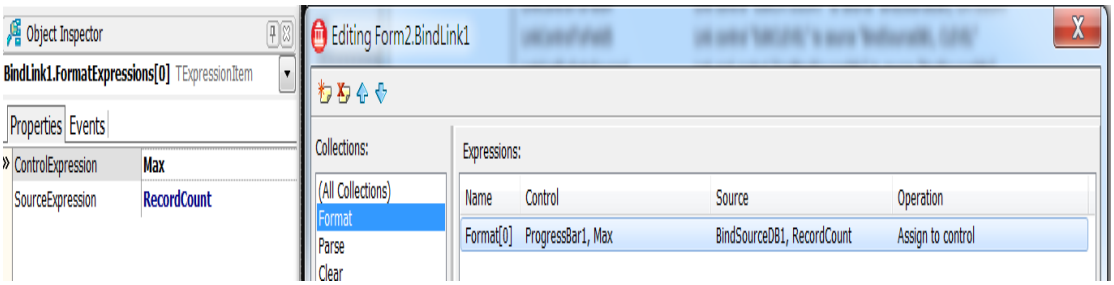



這代表要繫結 BindSourceDB1(即 cdsBooks)到 TProgressBar 物件。

接著雙擊 BindingsList1 編輯盒中的第 1 個 TBindLink 物件 BindLink1 如下所示:

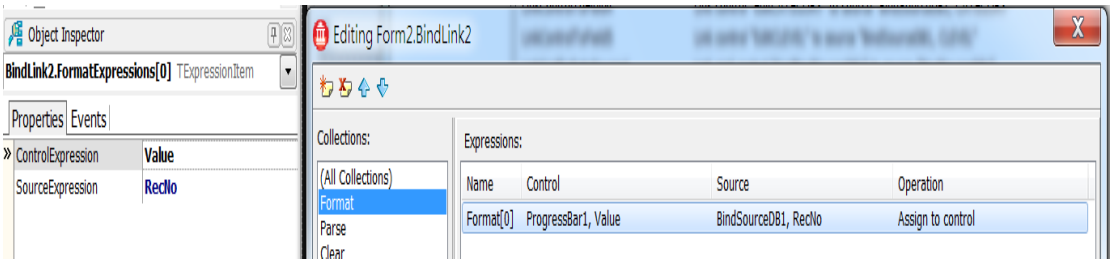



再於物件檢視器中設定第 1 個 TBindLink 物件的 SourceExpression 特性值為 RecordCount，設定 ControlExpression 特性值為 Max:




| Source | Control |
|---|---|
| BindSourceDB1, RecordCount | ProgressBar1, Max |

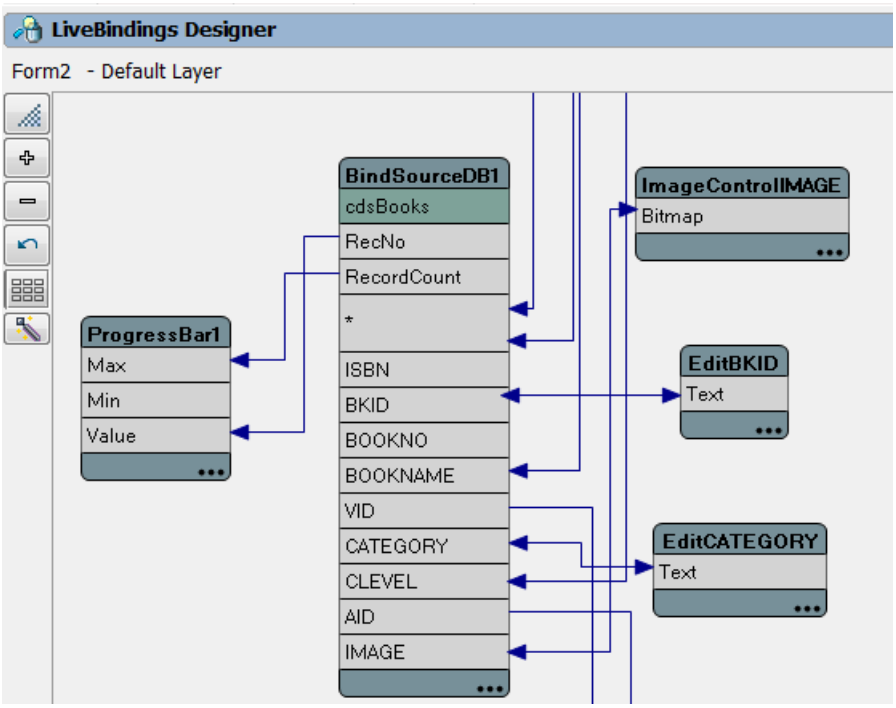
於物件檢視器中設定第 2 個 TBindLink 物件的 SourceExpression 特性值為 RecNo，設定 ControlExpression 特性值為 Value:



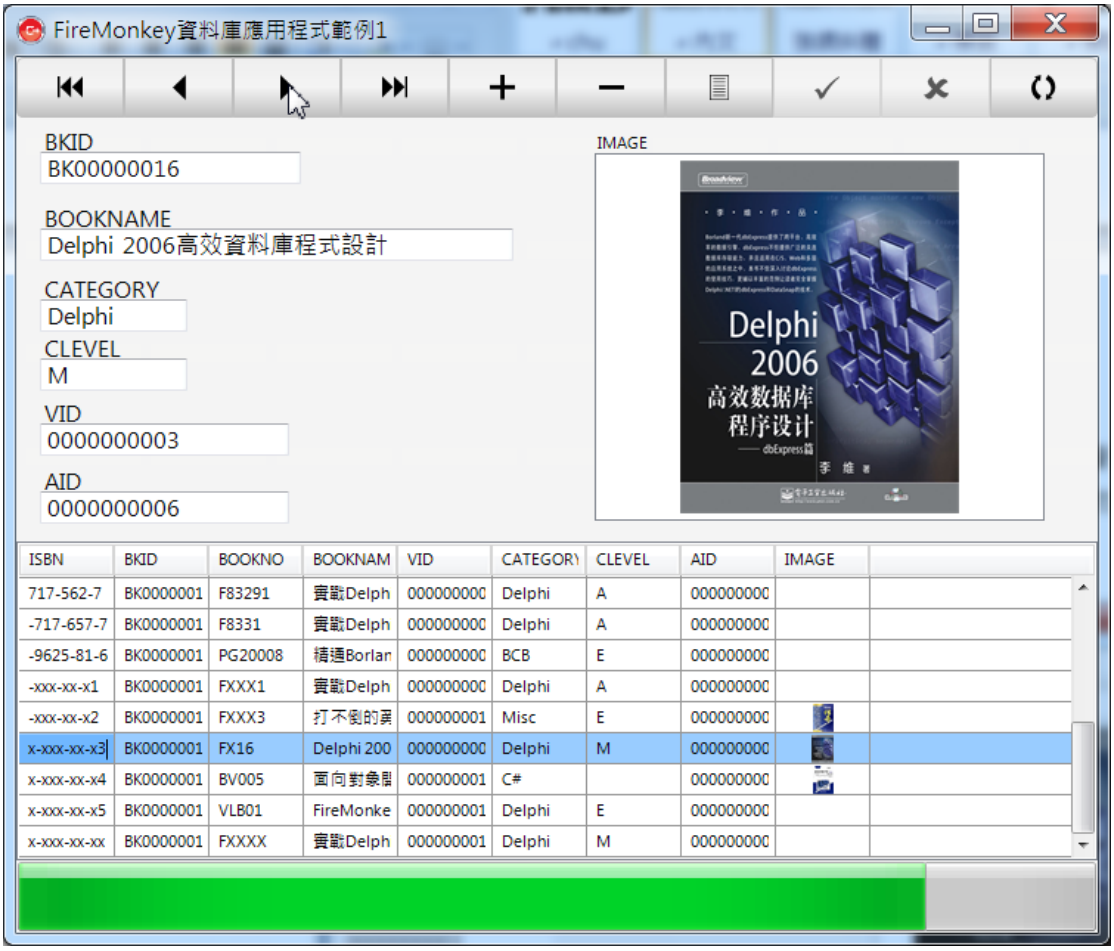



| 元件 | 特性值 |
|---|---|
| BindSourceDB1, RecNo | ProgressBar1, Value |

設定完成之後視覺化即時資料繫結設計家就會顯示 cdsBooks 和 TProgressBar 之間有如下的繫結關係:




最後編譯並執行範例 FireMonkey 應用程式，從下圖我們可以看到當我們使用 Navigator 在資料中瀏覽時，TProgressBar 能夠正確的顯示目前記錄的相對位置了:




從這個範例我們可以證實開發人員在視覺化即時資料繫結設計家中進行的繫結會自動產生繫結運算式並且由即時資料繫結引擎負責執行，當然開發人員也可以直接在 TBindingsList 元件中建立各種不同型態的繫結物件並且直接使用繫結，在稍後的章節中本書會進一步的討論繫結物件。

## 7-2 使用 TBindSourceDBX 元件

TOKYO 的即時資料繫結也增加了數個新的元件能夠幫助開發人員更方便的使用即時資料繫結技術來開發和資料庫/資料來源相關的應用程式。本小節將說明如何使用 TBindSourceDBX 元件。

TBindSourceDBX 元件結合了數個元件於一身讓開發人員只需要使用這一個元件就可以連結資料庫並且繫結控制項。簡單的說 TBindSourceDBX 內部包含了 TSQLDataSet，TDataSetProvider，TClientDataSet 和 TDataSource 等元件，因此開發人員只需要連結 TBindSourceDBX 到 TSQLConnection 元件即可開始進行繫結的

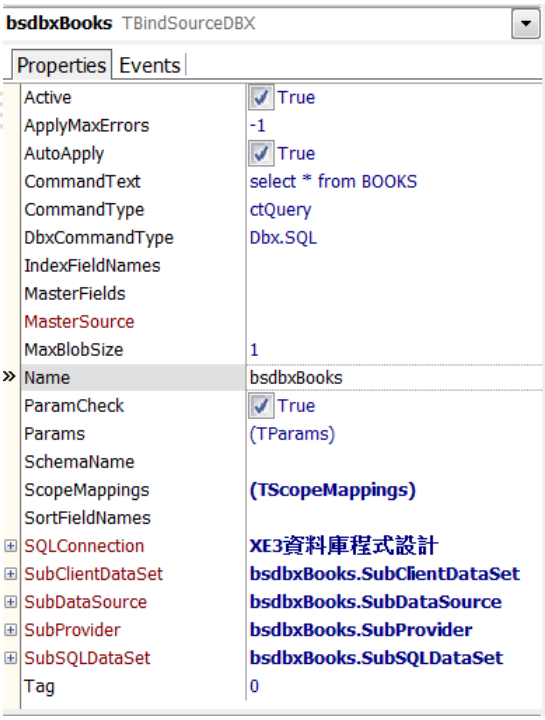
開發工作，可節省許多重覆使用 TDataSetProvider，TClientDataSet 和 TDataSource 元件的時間。在本小節中將以一個簡單的範例說明如何使用 TBindSourceDBX 來進行 Master/Detail 的開發。

首先在 C++Builder 整合發展環境中建立一個 FireMonkey Desktop Application 專案，拖曳 Data Explorer 中的『XE3 資料庫程式設計』節點到主表單中以建立一個 TSQLConnection 元件，再放入一個 TBindSourceDBX 元件，設定它的特性值如下:

| 特性 | 特性值 |
|---|---|
| Name | bsdbxBooks |
| SQLConnection | XE3資料庫程式設計 |
| CommandText | select * from BOOKS |

請注意下面的物件檢視器，在其中我們可以看到 TBindSourceDBX 元件包含了 SubClientDataSet，SubDataSource，SubProvider 和 SubSQLDataSet 四個特性，這四個特性就是 TBindSourceDBX 元件中包含的 TSQLDataSet，TDataSetProvider，TClientDataSet 和 TDataSource 物件。





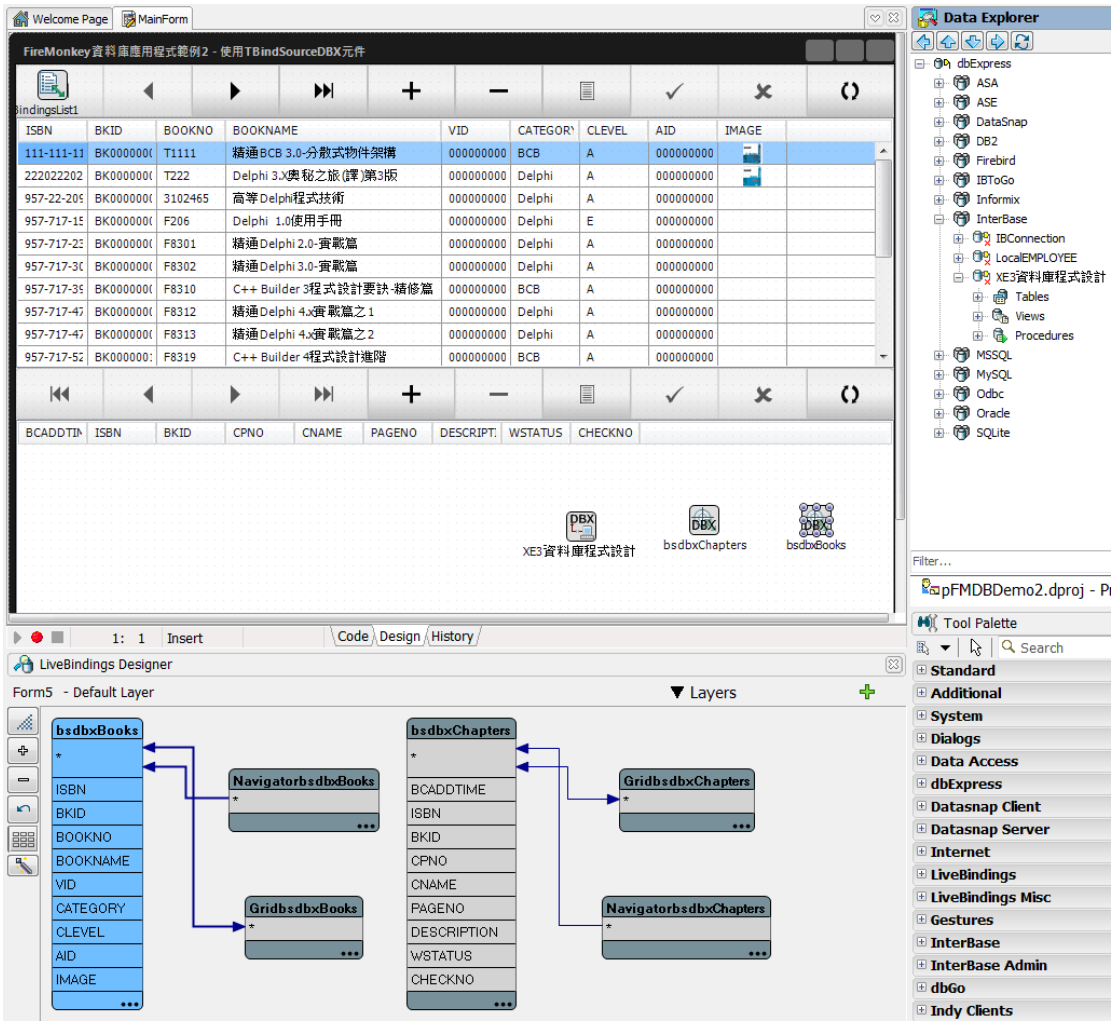

現在再放入另外一個 TBindSourceDBX 元件，設定它的特性值如下:

| 特性 | 特性值 |
|---|---|
| Name | bsdbxChapters |

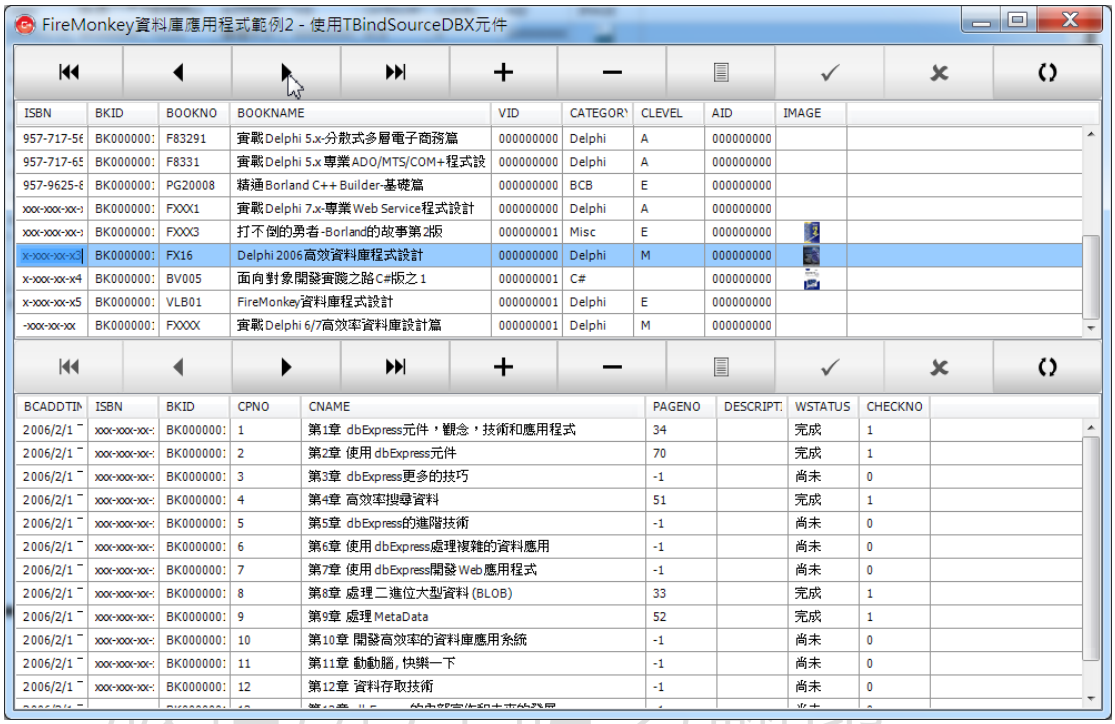
| | |
|---|---|
| SQLConnection | XE3資料庫程式設計 |
| CommandText | select * from BOOKCHAPTERS |
| MasterSource | bsdbxBooks |
| MasterFields | BKID |
| IndexFieldNames | BKID |

由於 BOOKS 和 BOOKCHAPTERS 這 2 個資料表之間是以 BKID 這個鍵值欄位關連在一起，因只要我們設定 bsdbxChapters 的 MasterSource，MasterFields 和 IndexFieldNames 這 3 個特性就可以在 bsdbxBooks 和 bsdbxChapters 之間建立 Master/Detail 的關係。

最後啟動視覺化即時資料繫結設計家，分別繫結 TGrid 和 TBindNavigator 到兩個 TBindSourceDBX 元件，最後在整合發展環境中的設計結果如下所示:



現在讀者就可以編譯和執行這個範例 FireMonkey 應用程式了，從下圖可以看到使用 TBindSourceDBX 元件果然可以輕易的在 FireMonkey 應用程式中建立資料來源之間的 Master/Detail 關係。

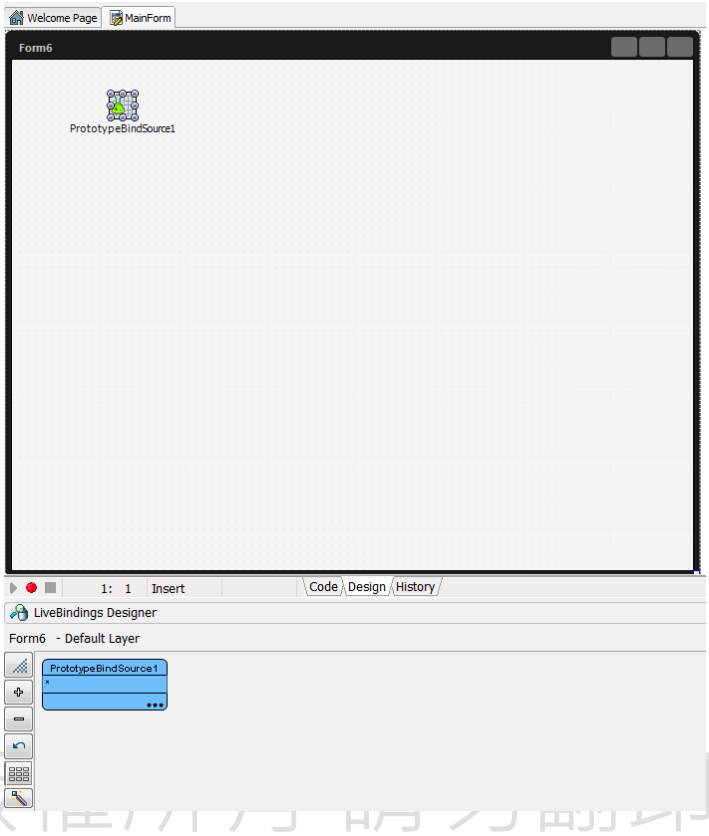



# 7-3 使用 TPrototypeBindSource 元件

TPrototypeBindSource 也是 TOKYO 才出現的新元件，它的主要目的是快速為開發人員建立一個類似虛擬的資料表，其中可包含開發人員需要的任何種類的資料型態的欄位，並且在這個虛擬資料表中產生隨機資料，以便讓開發人員可以進行 POC(Proof of concept)的開發，或是進行快速的雛形開發工作。

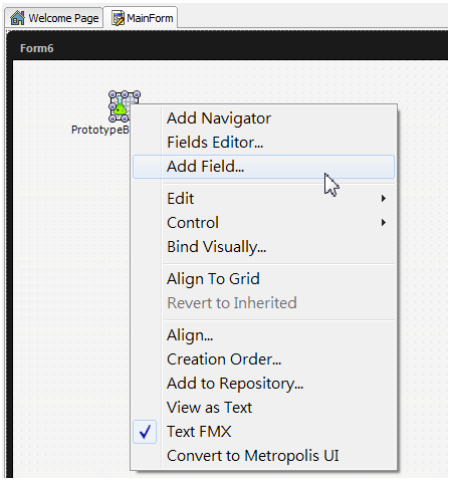
開發人員可以使用 TPrototypeBindSource 元件快速開發 VCL/FireMonkey 的資料相關應用程式，以便向客戶展示應用程式的特定功能。由於 TPrototypeBindSource 元件能夠自動且快速的產生隨機資料，因此也適合使用來進行測試的目的。本小節將說明如何使用 TPrototypeBindSource 元件，以便讓讀者也能夠使用它來快速開發 VCL/FireMonkey 的資料相關應用程式。

首先在整合發展環境中建立一個 FireMonkey Desktop Application 專案，於主表單中加入一個 TPrototypeBindSource 元件，在視覺化即時資料繫結設計家中就會出現 TPrototypeBindSource 實體，如下所示:

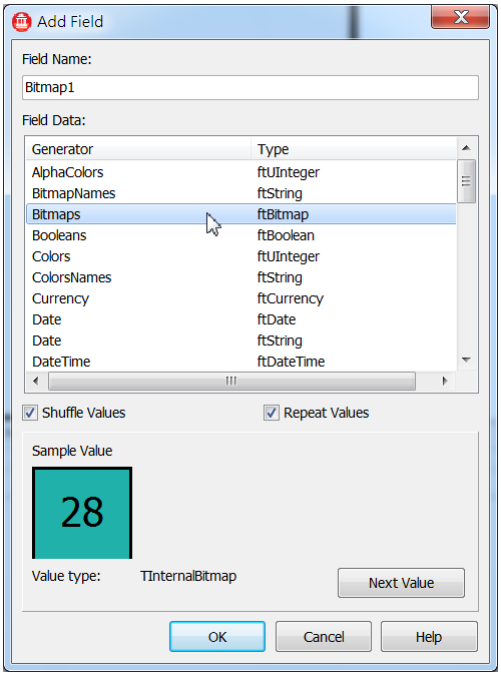





使用滑鼠右擊 TPrototypeBindSource 元件並且從快顯功能表中選擇『Add Field…』選項開始在 TPrototypeBindSource 元件中加入欄位，如下所示:

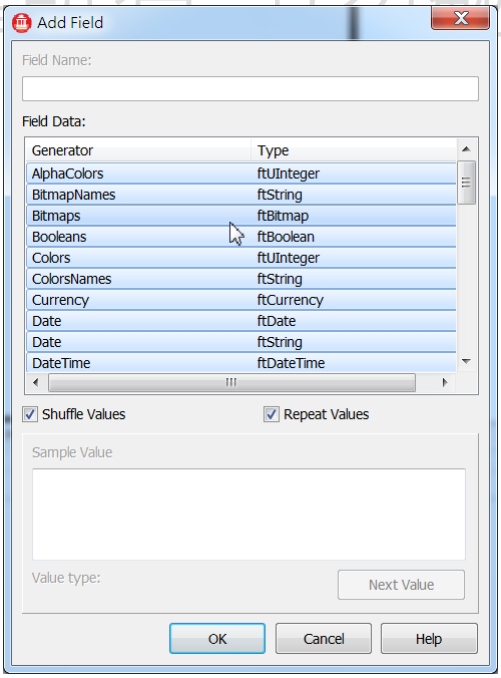



此時便會出現如下的 Add Field 對話盒，其中包含了內定的各種資料型態的欄位，讀者可以選擇要加入到 TPrototypeBindSource 元件中的欄位，當讀者點選其中的欄位時可以在對話盒的下方中看到選擇的資料欄位會產生的隨機範例資料:



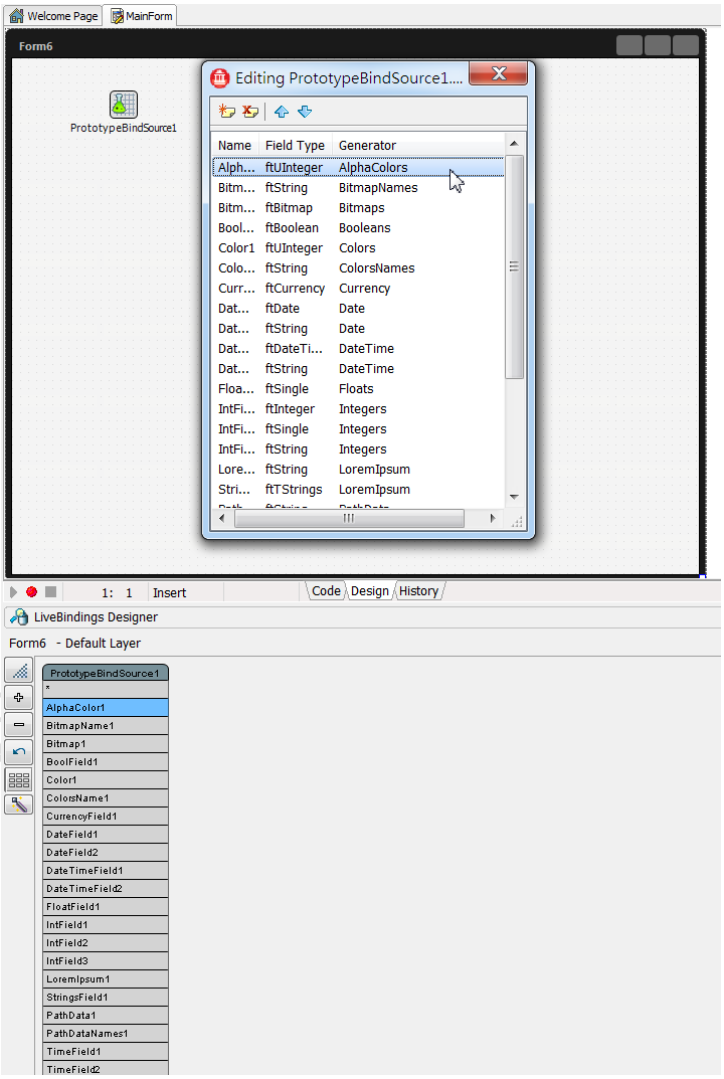

我們可以在按下 Shift 鍵時使用滑鼠點選多個欄位以選擇加入 TPrototypeBindSource 元件中，如下所示，在選擇要加入的欄位之後點選 OK 按鈕關閉對話盒:



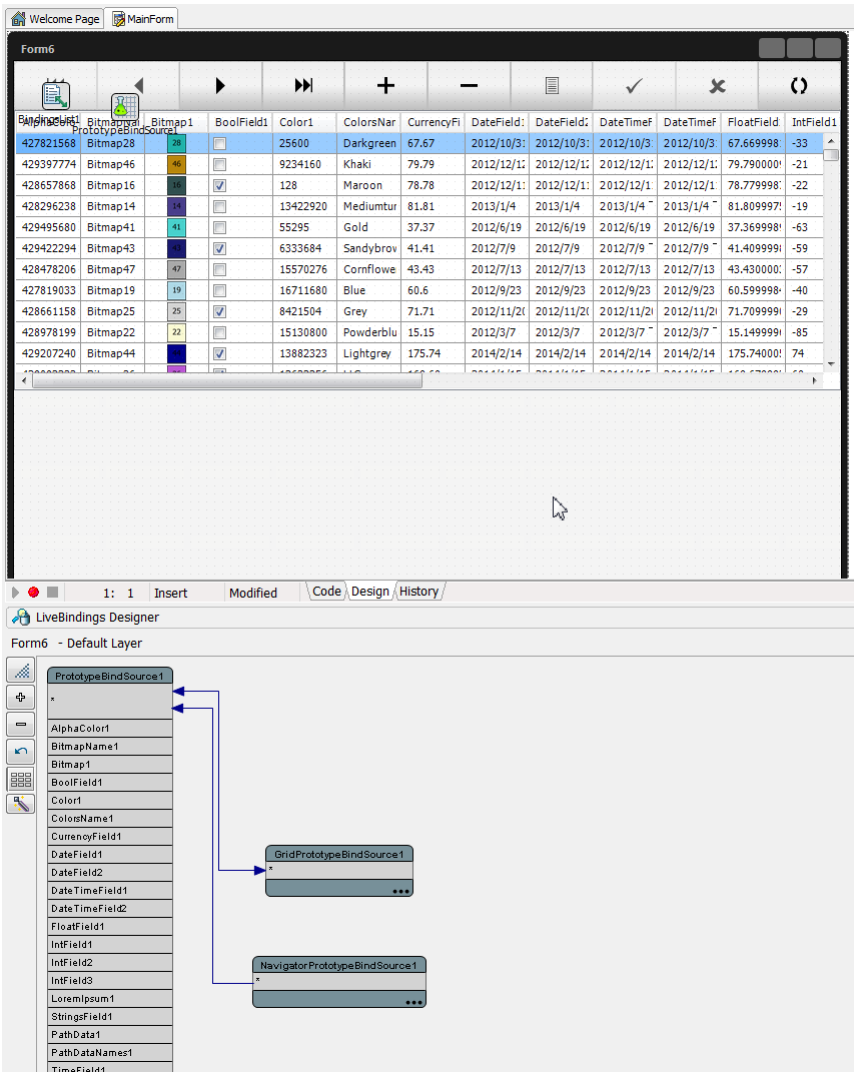



接著再次使用滑鼠右擊 TPrototypeBindSource 元件並且從快顯功能表中選擇『Fields Editor….』選項，就可以在欄位編輯器中看到剛才加入的所有欄位，如下所示，

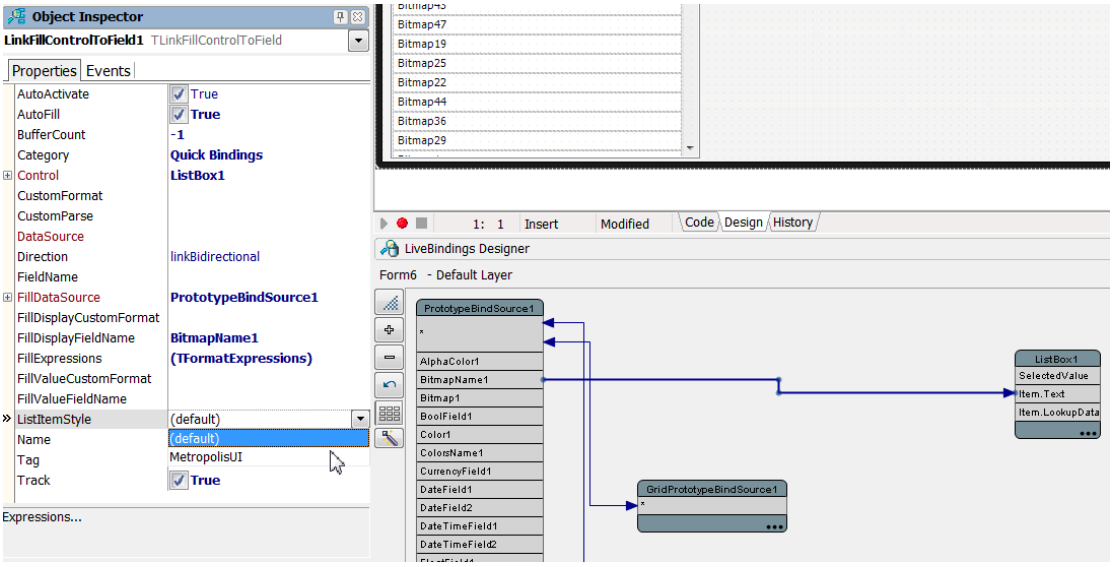
而且視覺化即時資料繫結設計家在 TPrototypeBindSource 實體中也會顯示剛加入的欄位物件:



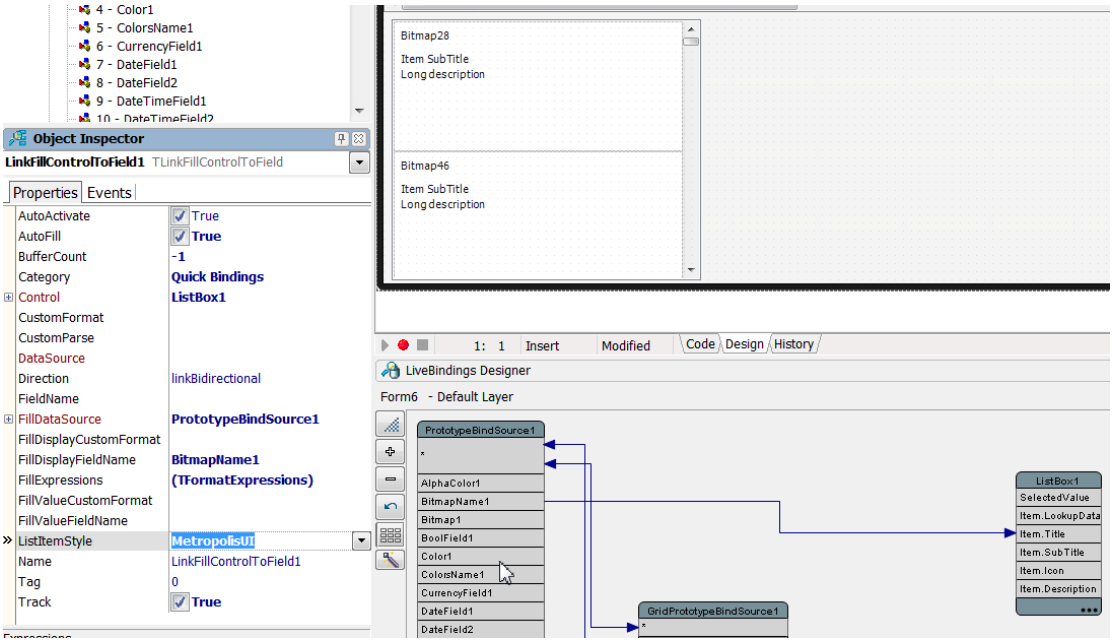

現在請在視覺化即時資料繫結設計家中連結 TPrototypeBindSource 實體和 TGrid 元件，再繫結到 TBindNavigator，在下圖中讀者就可以看到 TPrototypeBindSource 元件中包含的欄位以及隨機產生的資料都出現在主表單的 TGrid 元件中了:



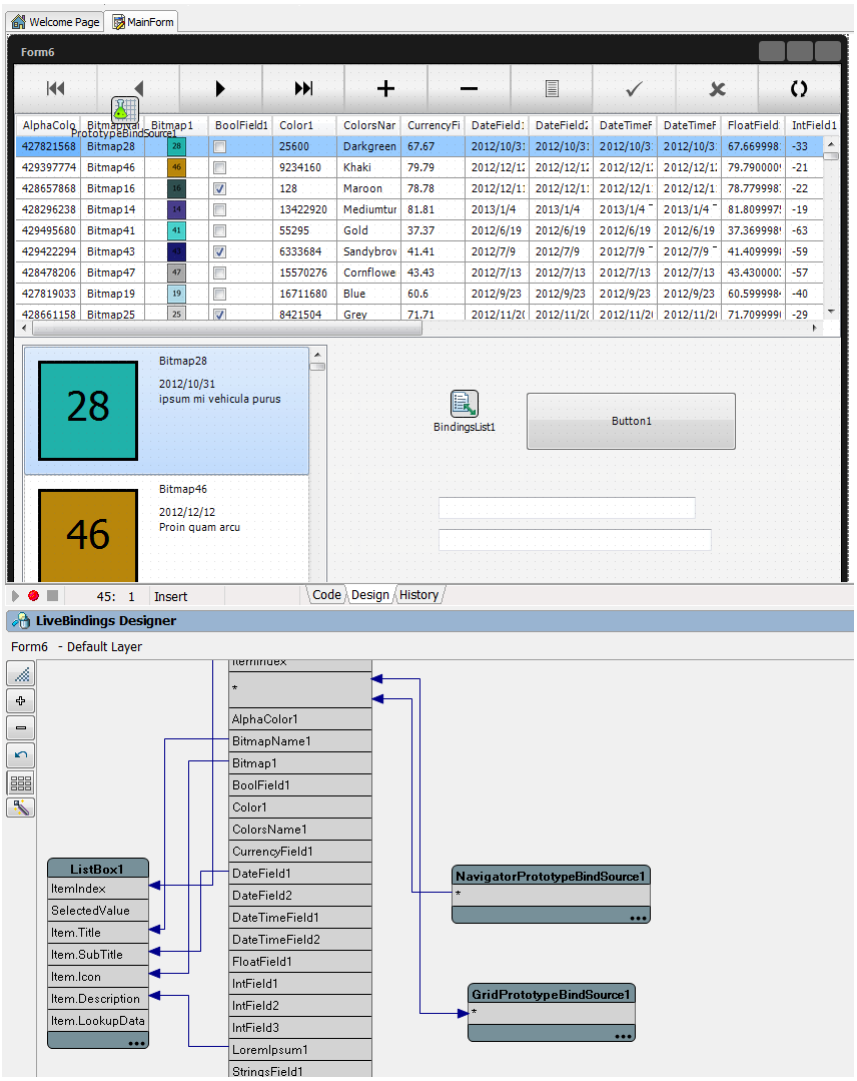

現在再於主表單中放入一個 TListBox 元件，在視覺化即時資料繫結設計家中點選 TPrototypeBindSource 實體的 BitmapName1 欄位並拖曳滑鼠到 TListBox 元件的 Text 特性以繫結這 2 個對象，此時 TListBox 元件中就會顯示 TPrototypeBindSource 元件中 BitmapName1 欄位的所有值，接著在視覺化即時資料繫結設計家中點選繫結 TPrototypeBindSource 實體 BitmapName1 欄位和 TListBox 元件 Text 特性之間的連結線，再於物件檢視器中點選它的 ListItemStyle 特性，選擇 MetropolisUI，如下圖所示:




在選擇了 MetropolisUI 特性值之後主表單中的 TListBox 元件的內容就會變化成使用 MetropolisUI 的格式，如下圖所示:

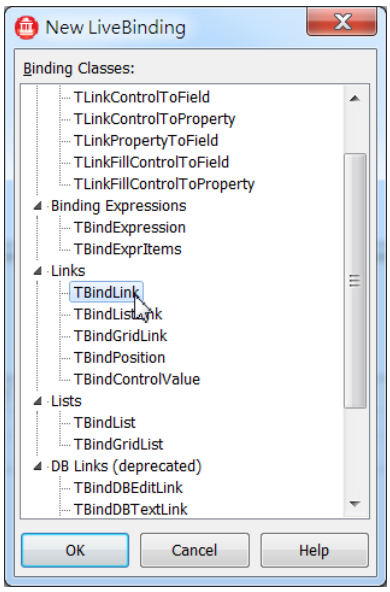



在視覺化即時資料繫結設計家中繫結 BitmapName1 欄位和 ListBox1 實體的 Item.Title 特性，在繫結設計家中繫結 Bitmap1 欄位和 ListBox1 實體的 Item.Icon 特性，在繫結設計家中繫結 DateField1 欄位和 ListBox1 實體的 Item.Subtile 特性，最後這些資料就都會出現在主表單的 TListBox 中，如下圖所示:

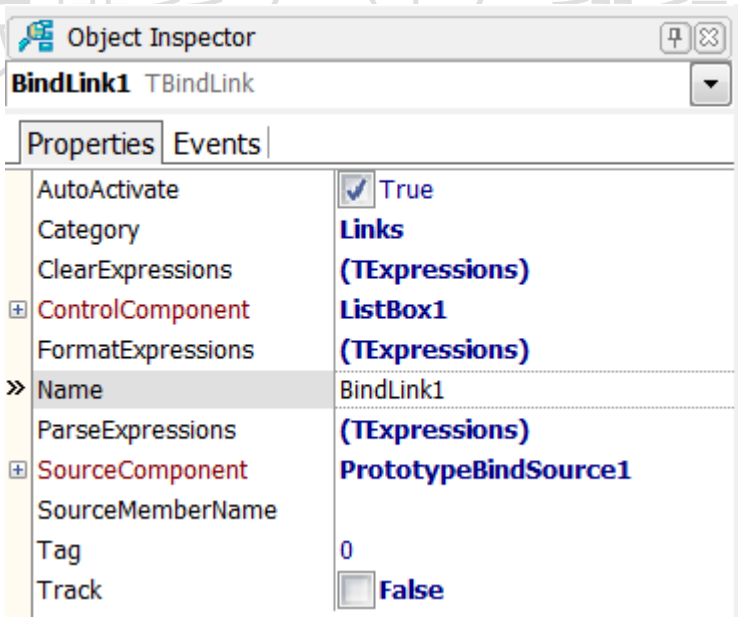



在離開本小節之前讓我們再試著使用另外一個繫結物件來繫結 TPrototypeBindSource 和 TListBox，讓使用者在瀏覽 TPrototypeBindSource 中的資料時也能夠同步 TListBox 中的資料。

請雙擊主表單中的 TBindingsList 物件，點選左上方的 New Binding 按鈕以增加一個新的繫結物件，在 New LiveBinding 對話盒中選擇建立 TBindLink 物件，如下圖所示:

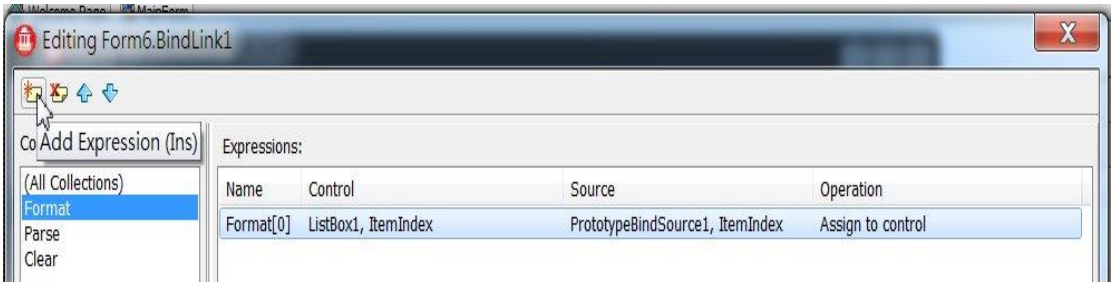



接著在 TBindingsList 對話盒中點選此新建立的 TBindLink 物件，再於物件檢視器中設定它的 SourceComponent 特性值為 PrototypeBindSource1，設定它的 ControlComponent 特性值為 ListBox1，如下圖所示:

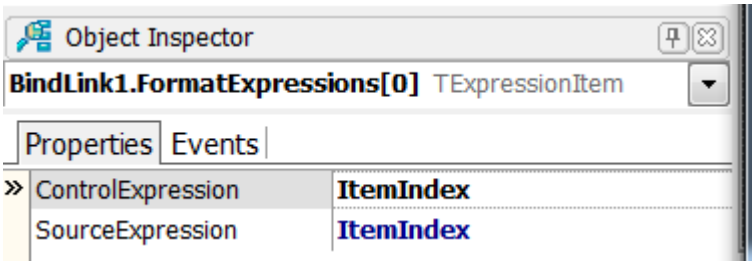



一旦做了如上的設定就代表使用 TBindLink 繫結物件來繫結 TPrototypeBindSource 和 TListBox 物件，接下來我們需要再設定繫結運算式來定義繫結 TPrototypeBindSource 和 TListBox 物件之中的什麼特性。

現在請雙擊 TBindingsList 對話盒中的 TBindLink 物件，此時 TBindLink 物件的繫結運算式編輯器就會出現，如下所示:




接著在物件檢視器中設定 SourceExpression 為 ItemIndex，也設定 ControlExpression 為 ItemIndex，如下所示:



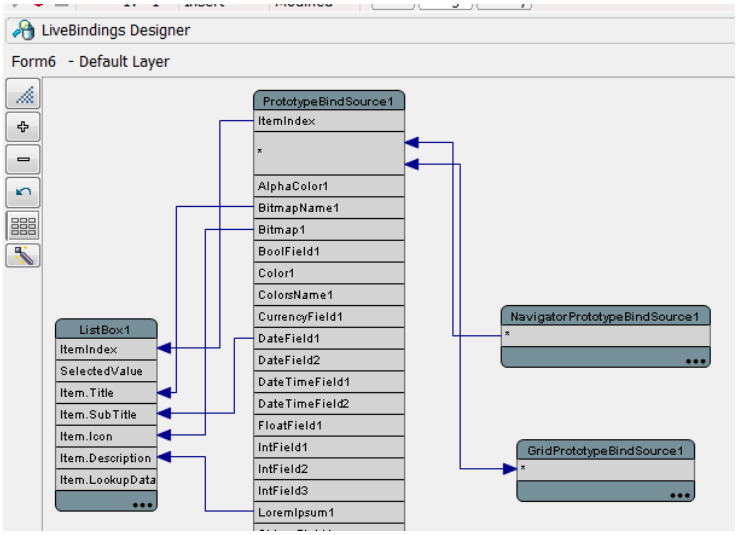



這個繫結運算式就代表在執行時期會進行如下的繫結:

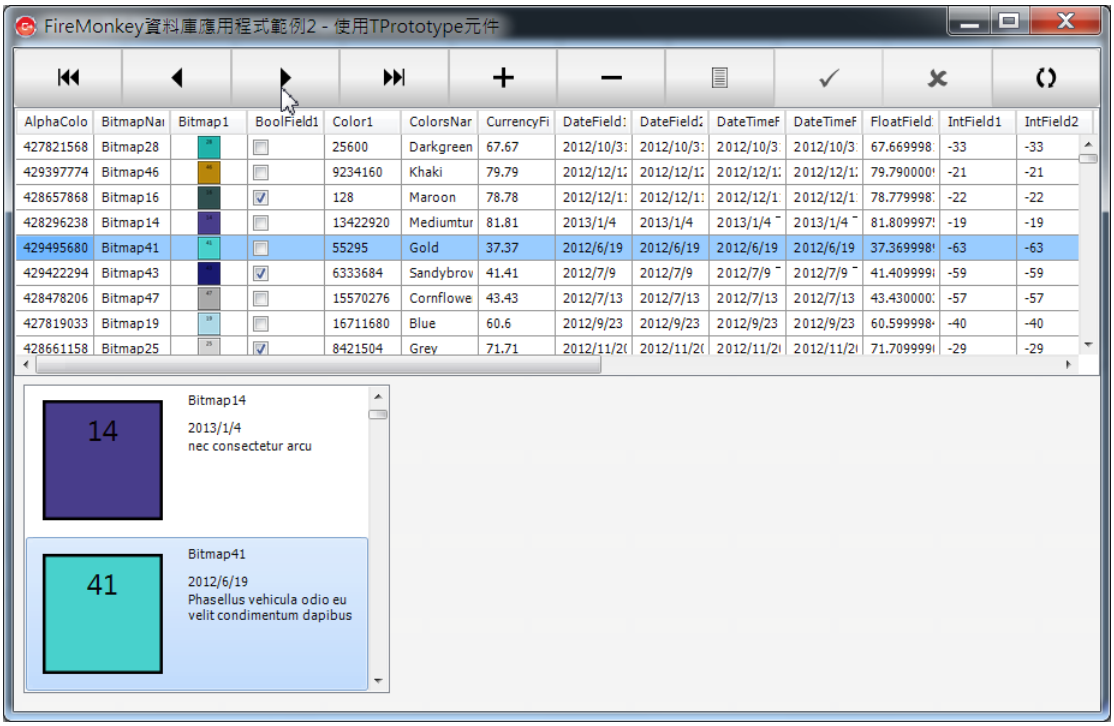
```
ListBox1.ItemIndex := PrototypeBindSource1.ItemIndex;
```

也就是說當我們瀏覽 TPrototypeBindSource 中的資料時，也會相對應的瀏覽到 TListBox 中包含的項目。

現在回到視覺化即時資料繫結設計家就可以看到 PrototypeBindSource1 實體和 ListBox1 實體之間的 ItemIndex 有了繫結的關係了，如下所示:



請編譯並且執行範例應用程式，就可以看到如下的執行結果，當我們使用 TBindNavigator 瀏覽 PrototypeBindSource1 的資料時，ListBox1 也瀏覽到相對應的資料了。




使用視覺化即時資料繫結設計家進行繫結工作雖然非常的簡單，但即時資料繫結仍然有一些深入的技術，我們將在下一章中進行說明。

## 7-4 結論

本章說明了如何使用 TOKYO 的視覺化即時資料繫結技術來繫結資料來源和控制項，藉由視覺化即時資料繫結設計家開發人員可以輕易的繫結 FireMonkey 控制項和資料來源，如此一來開發人員就可以輕易的開發出資料庫相關的 FireMonkey 應用程式了。

在本章討論的過程中也簡單的說明了如何使用繫結物件並且使用簡單的繫結運算式來繫結控制項，讓讀者瞭解除了可使用視覺化即時資料繫結設計家進行繫結工作之外，也可以直接使用繫結物件和繫結運算式來進行其他的繫結工作，在稍後的章節中會更詳細的討論繫結類別和繫結技術。

# 第8章 更多的即時資料繫結技術

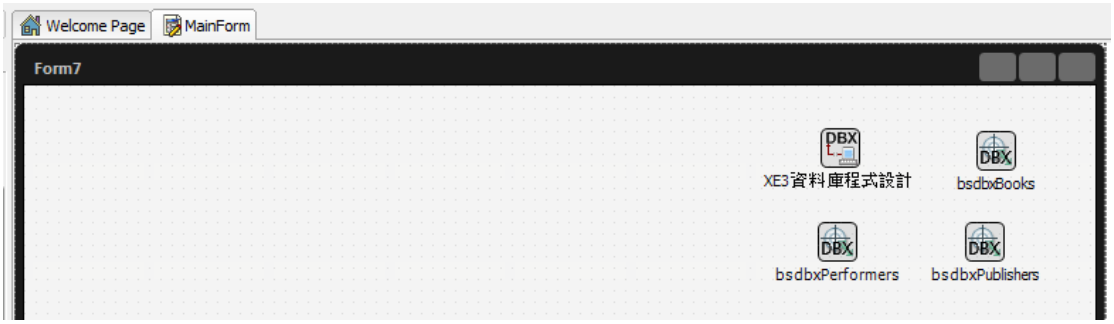
在開發資料庫應用程式時經常需要根據一個欄位來查詢另外一個查詢值，例如當使用者輸入身分證字號時應用程式需要根據身分證字號來查詢並且顯示此身分證字號的人名，這種應用在即時資料繫結技術稱為 Lookup 的關係，在下面的小節中將詳細的討論如何使用這個功能。



## 8-1 使用即時資料繫結技術的 Lookup 功能

本書將使用 InterBase 做為範例資料庫，在其中有數個資料表之間擁有 Master/Detail 的關係，在說明如何使用視覺化即時資料繫結技術逐漸開發 FireMonkey 資料庫應用程式的流程中，將一一的帶入使用這些資料表。不過讓我們先從簡單的開發工作開始說明如何使用視覺化即時資料繫結。

我們還是以第 1 章中的『XE3 資料庫程式設計』為範例資料庫，首先在 IDE 中建立一個 FireMonkey Desktop Application 專案，拖曳 Data Explorer 的『XE3 資料庫程式設計』節點到主表單中建立 TSQLConnection 元件，再放入 3 個 TBindSourceDBX 元件連結到 BOOKS，Publishers 和 Performers 這 3 個資料表，如下所示:




設定主表單中 3 個 TBindSourceDBX 元件的特性值如下:

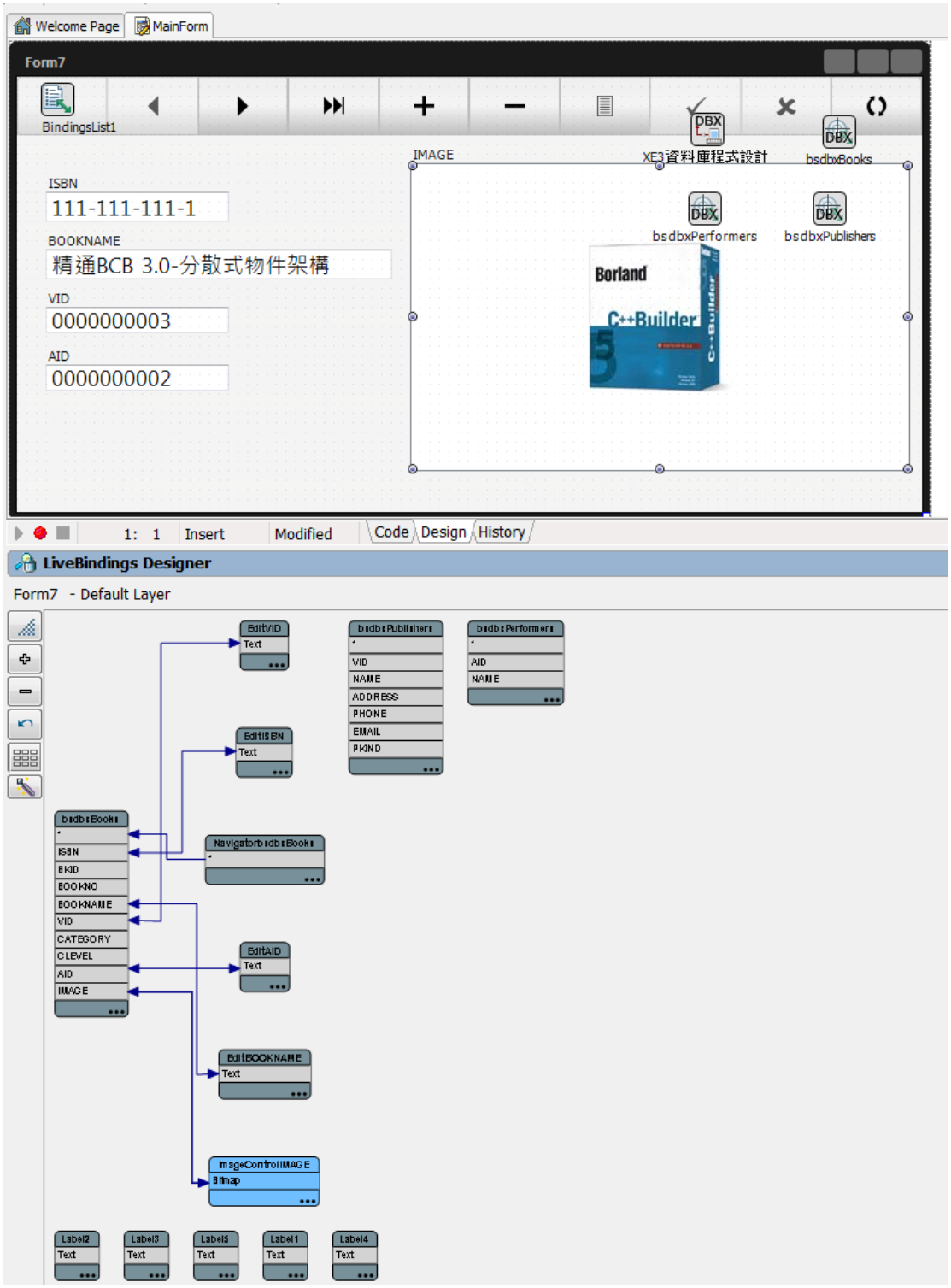
| 元件 | 特性值 |
|---|---|
| Name | bsdbxBooks |
| SQLConnection | XE3資料庫程式設計 |
| CommandType | ctQuery |
| CommandText | select * from BOOKS |

| 元件 | 特性值 |
|---|---|
| Name | bsdbxPublishers |
| SQLConnection | XE3資料庫程式設計 |
| CommandType | ctQuery |
| CommandText | select * from PUBLISHERS |



| 元件 | 特性值 |
|---|---|
| Name | bsdbxPerformers |
| SQLConnection | XE3資料庫程式設計 |
| CommandType | ctQuery |
| CommandText | select * from PERFORMERS |

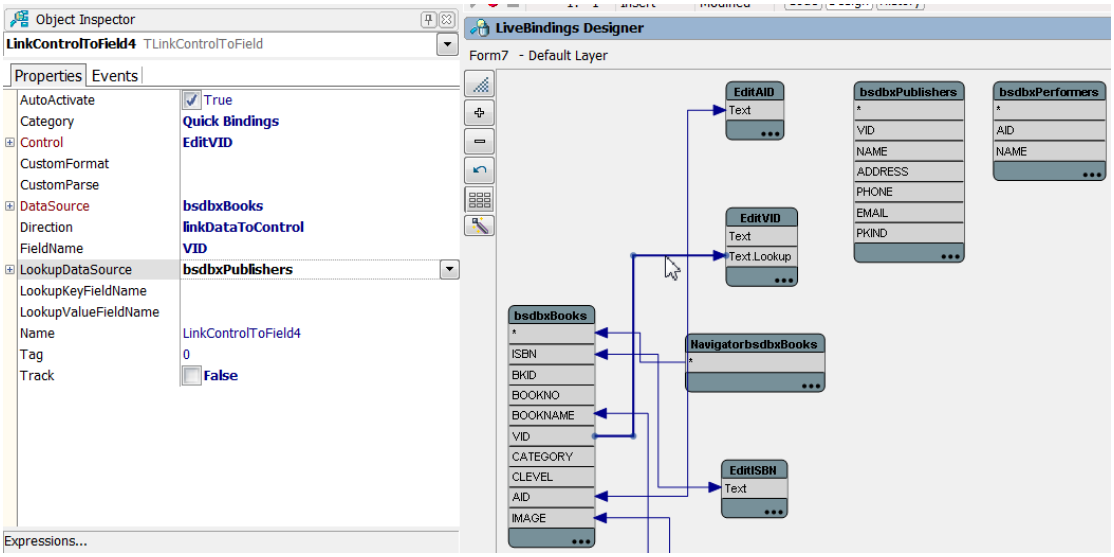
同時設定這 3 個 TBindSourceDBX 元件的 Active 特性值為 True 以開啟資料表:



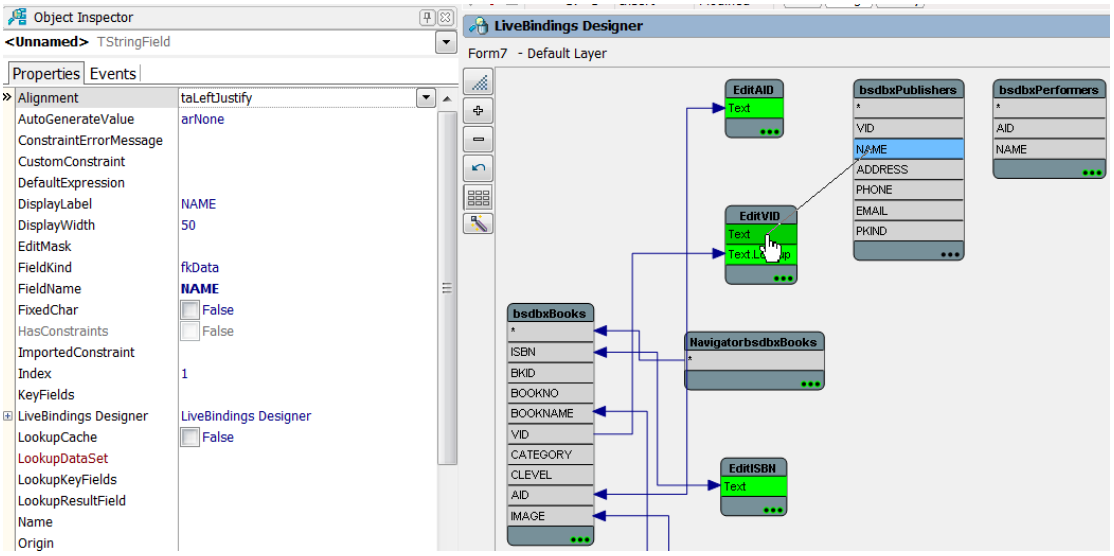

請注意主表單中的 VID 和 AID 這 2 個欄位，它們分別是 Publishers 和 Performers 這 2 個資料表的鍵值，因此我們並不希望顯示 VID 和 AID 的欄位值，而是這 2 個鍵值代表的數值，因此我們需要使用 VID 到 Publishers 資料表中查詢它代表的數值，使用 AID 到 Performers 資料表中查詢它代表的數值，這在即時資料繫結中就稱為 Lookup 的動作。

那麼我們要如何使用即時資料繫結的 Lookup 功能來根據 VID 和 AID 來查詢它們代表的數值呢?

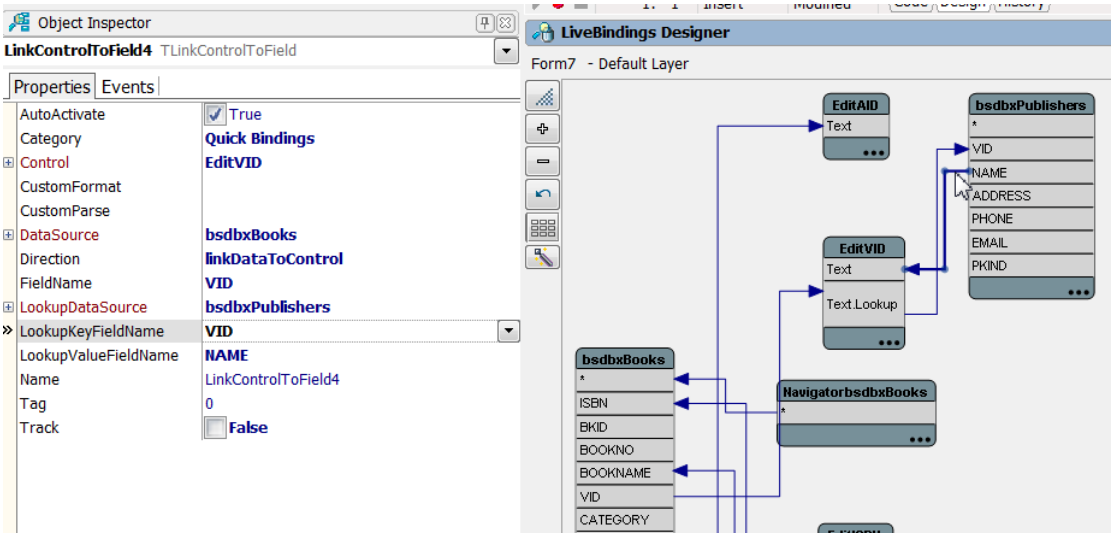
現在請回到視覺化即時資料繫結設計家，點選連結 bsdbxBooks 實體 VID 欄位和 EditVID 實體之間的連結線，接著在物件檢視器中的 LookupDataSource 特性中選擇 bsdbxPublishers，因為 VID 是 bsdbxPublishers 代表的資料表中的鍵值，注意一旦您在物件檢視器中設定了 LookupDataSource 特性值為 bsdbxPublishers，那麼 EditVID 實體立刻會出現一個新的欄位『Text.Lookup』，而且從 bsdbxBooks 實體 VID 欄位來的連結線會改變到連結到 EditVID 實體的『Text.Lookup』欄位，如下所示:




接著請點選滑鼠左鍵，從 bsdbxPublishers 實體的 NAME 欄位資料拖曳到 EditVID 實體的 Text 欄位，如下所示:



最後請點選連結 bsdbxPublishers 實體的 NAME 欄位和 EditVID 實體的 Text 欄位之間的連結線，在物件檢視器中設定 LookupFieldName 特性值為 VID，如下所示:

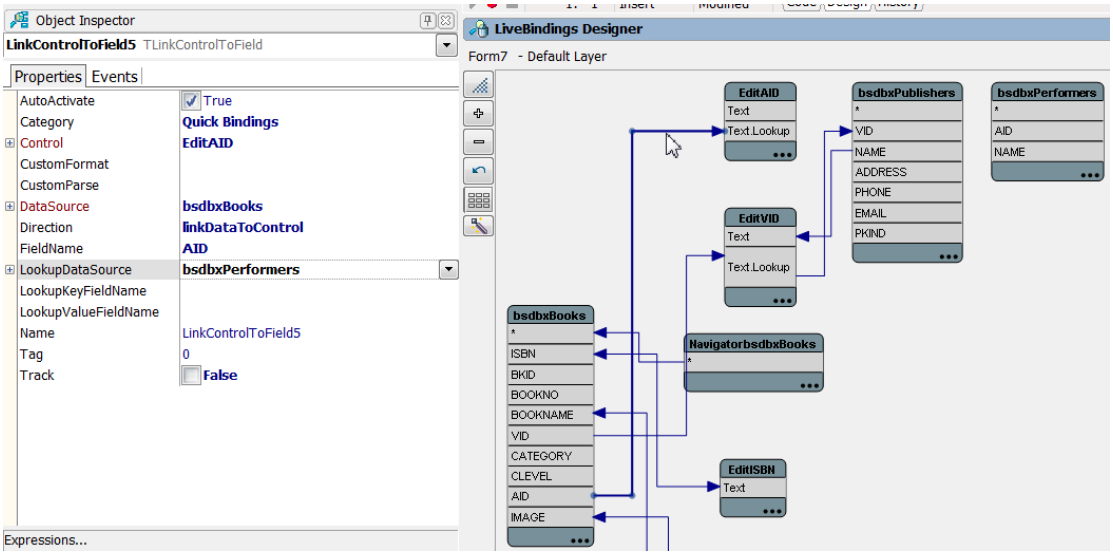



如此一來就完成了使用 bsdbxBooks 中 VID 欄位值到 bsdbxPublishers 進行 Lookup 查詢的工作，如果現在讀者檢視主表單中的 EditVID 元件就會發現它的 Text 特性值改變成顯示 Publishers 資料表中 VID 欄位值代表的數值了。

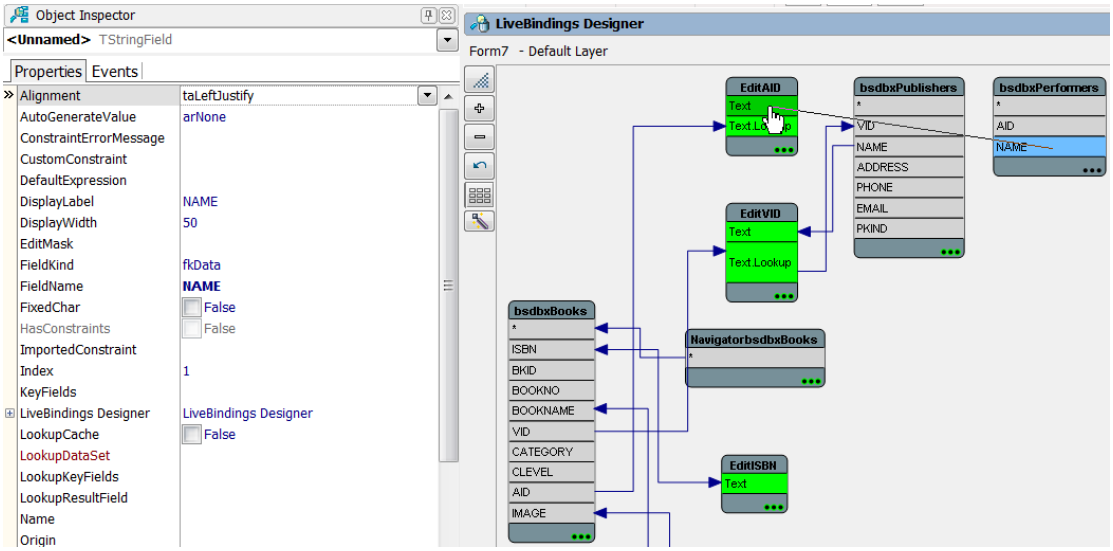


現在讓我們使用相同的方法完成 AID 欄位的 Lookup 查詢工作。

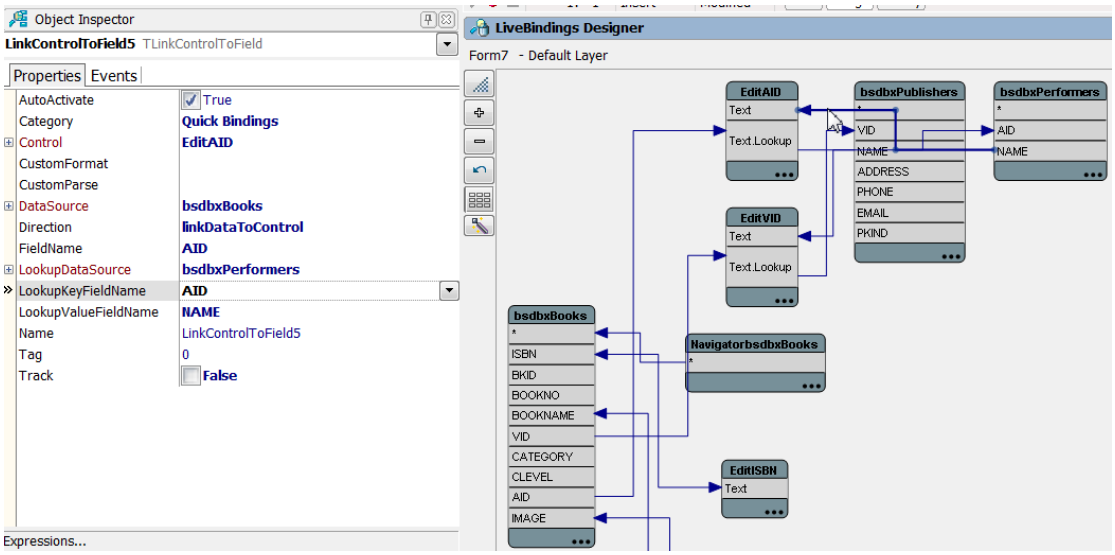
現在請回到視覺化即時資料繫結設計家，點選連結 bsdbxBooks 實體 AID 欄位和 EditAID 實體之間的連結線，接著在物件檢視器中的 LookupDataSource 特性中選擇 bsdbxPerformers，因為 AID 是 bsdbxPerformers 代表的資料表中的鍵值，注意一旦您在物件檢視器中設定了 LookupDataSource 特性值為 bsdbxPerformers，那麼 EditAID 實體立刻會出現一個新的欄位『Text.Lookup』，而且從 bsdbxBooks 實體 AID 欄位來的連結線會改變到連結到 EditAID 實體的『Text.Lookup』欄位，如下所示:



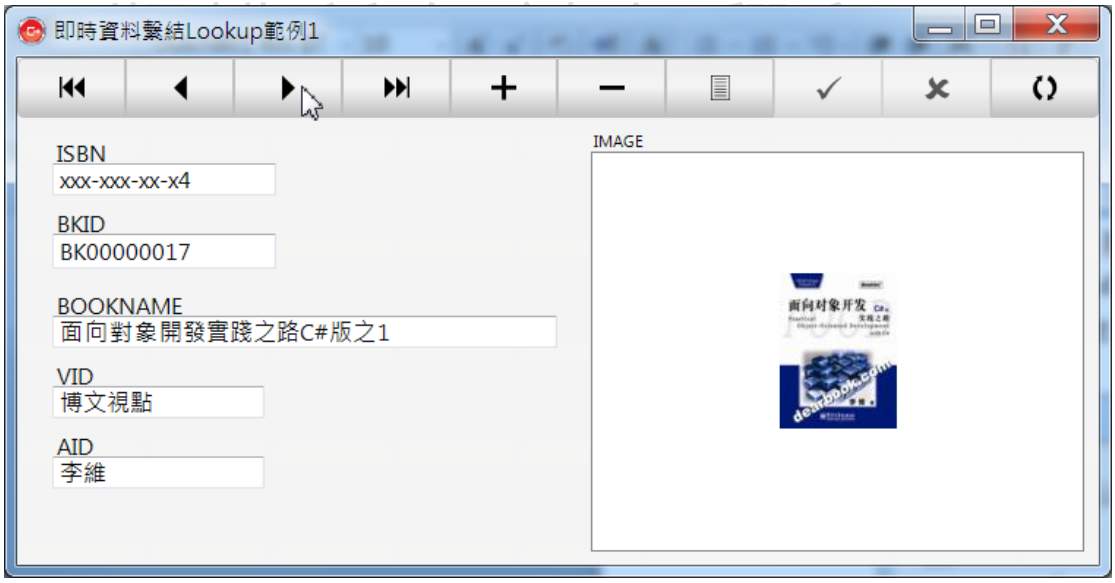

接著請點選滑鼠左鍵，從 bsdbxPerformers 實體的 NAME 欄位資料拖曳到 EditAID 實體的 Text 欄位，如下所示:



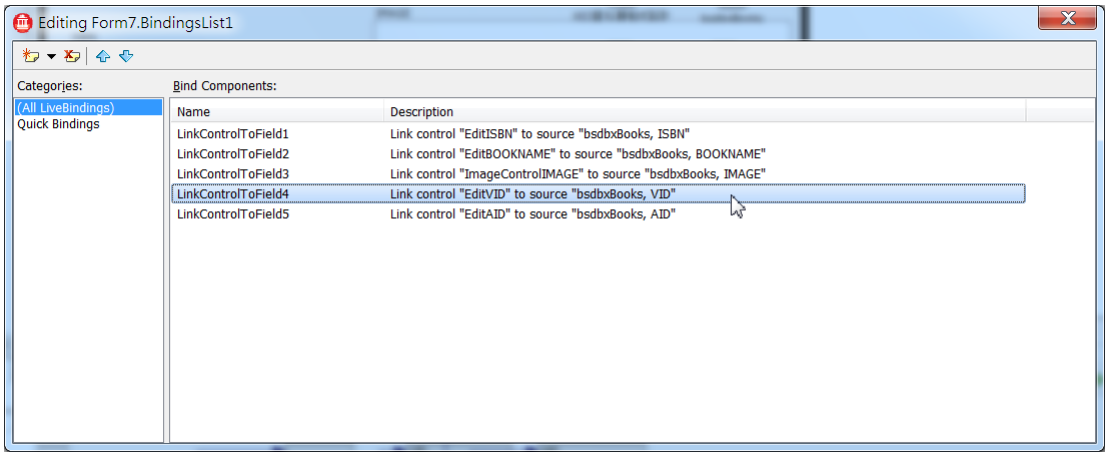

最後請點選連結 bsdbxPerformers 實體的 NAME 欄位和 EditAID 實體的 Text 欄位之間的連結線，在物件檢視器中設定 LookupFieldName 特性值為 AID，如下所示:



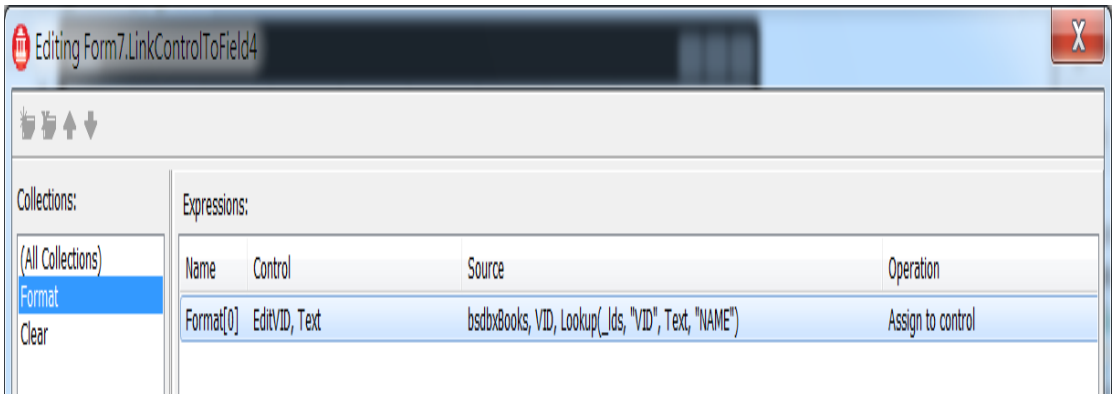

現在請編譯並且執行此範例應用程式，在使用 Navigator 瀏覽資料時，主表單中的 EditVID 和 EditAID 元件就能夠根據 bsdbxBooks 資料表中 VID 和 AID 欄位的數值到 bsdbxPublishers 和 bsdbxPerformers 資料表中查詢正確的代表數值並且顯示在 EditVID 和 EditAID 元件中，如下所示:




在離開本小節之前讓我們雙擊主表單中的 TBindingsList 元件開啟繫結運算式編輯器，點選其中代表 VID 執行查詢的繫結物件，如下所示:




雙擊此繫結物件，我們就可以看到如下的繫結運算式:




上圖中的繫結運算式基本的意思就是:

```
EditVID->Text = bdsdbxBooks->VID->Lookup(_lds, “VID”, Text, “NAME”);
```

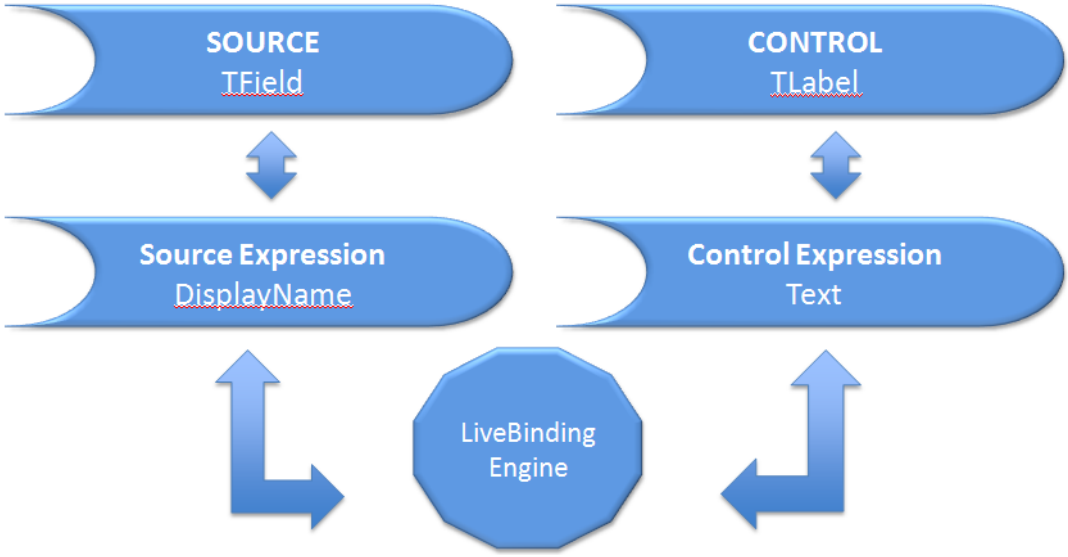
也就是說把 bdsdbxBooks->VID->Lookup(_lds, “VID”, Text, “NAME”)執行的結果指定給 EditVID->Text，但 bdsdbxBooks->VID->Lookup(_lds, “VID”, Text, “NAME”)到底是什麼？如果不是由視覺化即時資料繫結設計家自動產生這個繫結運算式，我們似乎根本寫不出這個繫結運算式，現在連看都不太懂這個繫結運算式的意思，當然更別說要寫出來了。

因此在我們繼續使用即時資料繫結技術之前，也許讀者需要對於即時資料繫結技術有一些基本的觀念。

# 8-2 什麼是即時資料繫結

即時資料繫結技術就是提供應用程式中元件/物件和資料來源之間一個可讀寫的連結，這個可讀寫的連結就是所謂的繫結運算式(Binding Expression) ，在繫結運算式中有個對象，一方稱為來源元件(Source Component) ，另一方則稱為控制元件(Control Component)。

繫結運算式有 2 種型態，即單向繫結(unidirectional)和雙向繫結(bidirectional) ，如果是單向繫結，那麼運作的方式就是由來源元件執行繫結運算式之後再把結果指定給控制元件。如果是雙向繫結，那麼控制元件也可以執行它的繫結運算式再指定回給來源元件，這整個概念可以使用下面的圖形來說明:




從上圖我們可以瞭解，繫結運算式是由即時資料繫結引擎來解釋和執行的，在目前 TOKYO 的實作中即時資料繫結引擎可執行下列 3 種的繫結運算式:

- 簡單的繫結運算式(Simple Expressions)
- 未拖管繫結運算式(Unmanaged Bindings)
- 拖管繫結運算式(Managed Bindings)

繫結運算式是由字串組成的運算式，在這個字串運算式中開發人員可使用下列的元素:

1. +, -, /, *這 4 個運算元

2. 常數值，例如字串常數值 'abc'，數值常數值 1， 1.23，布林常數值 True，False，和 Nil

3. 物件的方法和特性，全域方法和繫結引擎提供的內定方法。

繫結運算式中最重要的元素就是上述的第 3 項，這需要讀者清楚的瞭解它的函意。

當繫結運算式被繫結引擎執行時，繫結引擎會使用一個所謂的**執行範圍**(Scope)來解釋繫結運算式中包含的元素，讓我們使用一些範例來說明這個意思。

例如下面是一個簡單的繫結運算式:

```
Self.Count + 123
```

那麼在上面的繫結運算式中 Self.Count 到底是什麼呢? 這就要看看這個繫結運算式是在什麼執行範圍內執行，例如如果在執行這個繫結運算式之前先繫結執行範圍到一個 TListBox 物件，那麼繫結運算式中的 Self 就指這個 TListBox 物件，因此 Self.Count 就是指存取這個這個 TListBox 物件的 Count 特性值，因此上面的繫結運算式的意義就是指:

```
把 TListBox 物件的 Count 特性值加上 123 再指定給控制物件
```



有了這個基本的觀念之後我們就可以解釋上一節繫結運算式的意義了:

```
bdsdbxBooks->VID->Lookup(_lds, "VID", Text, "NAME");
```

在這個繫結運算式中的 _lds 暫時變數指的就是它的執行範圍，也就是 LookupDataSource 特性值代表的物件 bdsdbxPublishers 又由於 bdsdbxPublishers 元件是 TBindSourceDBX 類別型態，而 TBindSourceDBX 類別實作了 IScopeLookup 介面，在 IScopeLookup 介面中提供了 Lookup 方法的宣告:

```
__interface IScopeLookup;
typedef System::C++BuilderInterface<IScopeLookup> _di_IScopeLookup;
__interface  INTERFACE_UUID("{95C4149E-E1AD-4D21-A8DF-A84A33B6D2D9}")
IScopeLookup  : public System::IInterface
{
public:
    virtual System::Rtti::TValue __fastcall Lookup(const
System::UnicodeString KeyFields, const System::Rtti::TValue &KeyValues,
const System::UnicodeString ResultFields) = 0 ;
    virtual void __fastcall
GetLookupMemberNames(System::Classes::TStrings* AList) = 0 ;
```

```
};
```

因此 bdsdbxPublishers 元件實作了 Lookup 方法，所以這個繫結運算式才能夠呼叫 Lookup 方法，現在這個繫結運算式就不難瞭解了，對吧!

其實 IScopeLookup 介面的 Lookup 方法和 TClientDataSet 的 Lookup 方法功能是類似的，它的第一個參數是搜尋欄位名稱，第二個參數是搜尋值，第 3 個參數則是 Lookup 方法回傳的欄位值，因此 Lookup(_lds, '"VID", Text, "NAME")的意思就是使用 EditVID 元件的 Text 特性值搜尋 bdsdbxPublishers 資料來源的 VID 欄位，搜尋到符合的 VID 值之後就回傳 bdsdbxPublishers 資料來源中 NAME 欄位的數值。

現在我們就可以解釋這 3 種繫結運算式的意義了。

## 簡單的繫結運算式(Simple Expressions)

由執行範圍，TBindingExpression 物件和字串運算式形成的繫結運算式，簡單的繫結運算式一般是使用來計算數值並且和拖管繫結運算式或是未拖管繫結運算式一起使用。

## 拖管繫結運算式(Managed Bindings)



拖管繫結運算式和未拖管繫結運算式是開發人員在使用即時資料繫結技術時最常使用的 2 種運算式，這 2 種運算式都是使用來執行繫結運算式然後把執行結果指定給控制元件，但拖管繫結運算式是由即時資料繫結引擎所擁有並且可自動被執行和重新計算。

## 未拖管繫結運算式(Unmanaged Bindings)

未拖管繫結運算式是由應用程式所擁有而且必須藉由呼叫繫結物件的 Evaluate 方法才會被即時資料繫結引擎解釋和執行，它不像拖管繫結運算式是由即時資料繫結引擎自動解釋和執行的。

在稍後的章節會使用程式碼來真正的說明如何使用這 3 種繫結運算式，現在讓我們對即時資料繫結原理的說明暫時的告一段落，在稍後的章節中本書會詳細的說明，現在讓我們繼續討論即時資料繫的 Lookup 功能。

# 8-3 進階 Lookup 功能

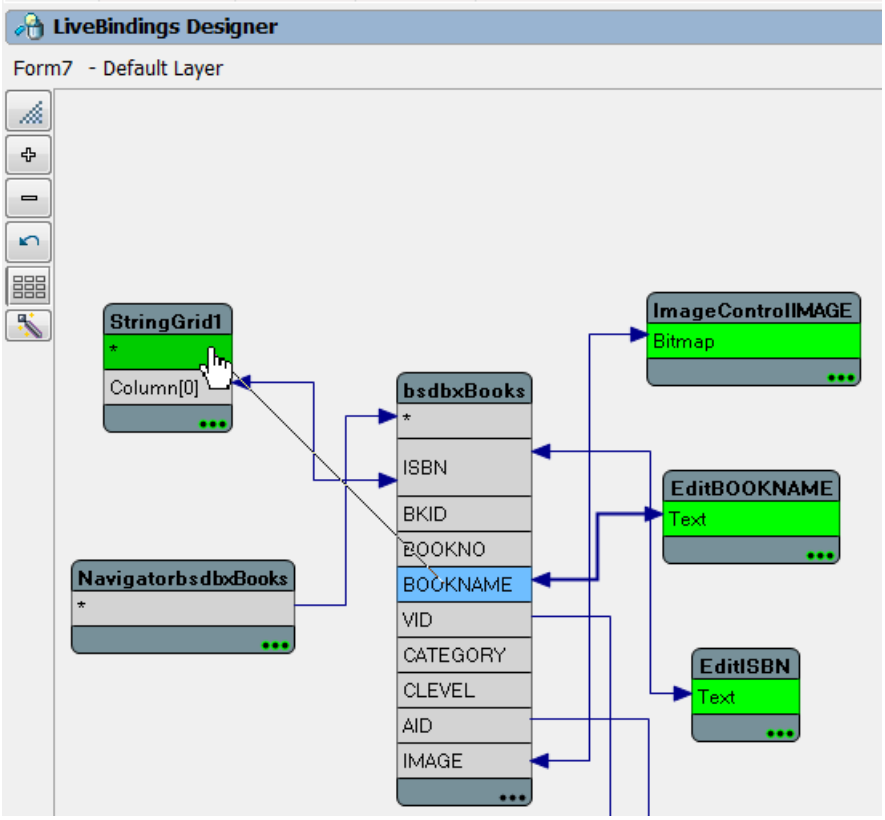
在 2-1 節說明了如何使用即時資料繫結的 Lookup 功能，在簡單的應用中開發人員可以藉由視覺化即時資料繫結設計家來完成 Lookup 功能，但對於比較複雜的 Lookup 應用開發人員仍然需要使用繫結運算式。

在閱讀完 2-2 小節之後讀者應該已經具備了繫結運算式的基本概念，現在就讓我們使用繫結運算式進行比較深入的 Lookup 應用。

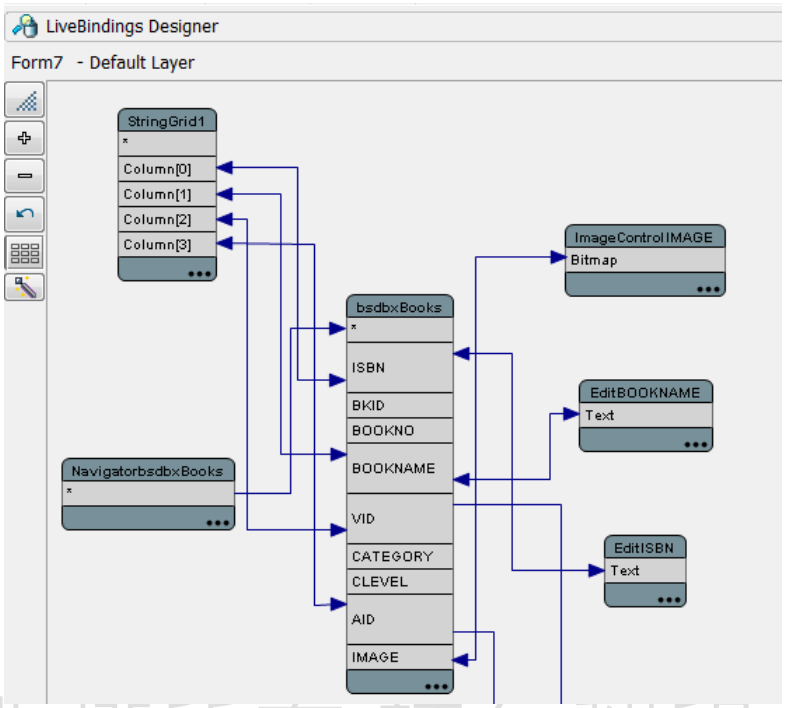
假設現在我們想使用 Grid 元件來顯示 BOOKS 資料表中的資料，但我們仍然希望在 Grid 中顯示 VID 和 AID 欄位時也是顯示 Publishers 和 Performers 資料表中的 NAME 欄位而不是 BOOKS 資料表的 VID 和 AID 欄位值。但問題是如何讓 Grid 元件也能夠執行 Lookup 的查詢?

現在請開啟 2-2 小節中的範例，在主表單中加入一個 TStringGrid 元件並且對齊在主表單的下方，再開啟視覺化即時資料繫結設計家，分別拖曳 ISBN，BOOKNAME，VID 和 AID 到 StringGrid1 實體的『*』欄位中，就可以把這些欄位顯示在 TStringGrid 中，如下所示:



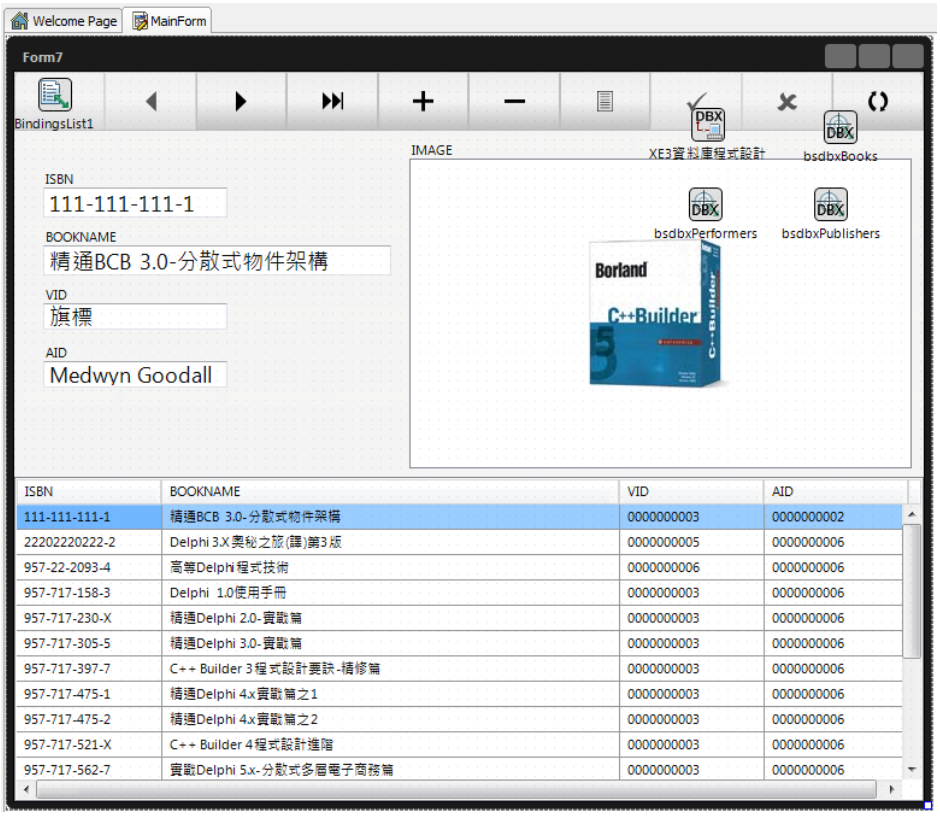


拖曳完這 4 個欄位後，在視覺化即時資料繫結設計家中就可以看到 StringGrid1 實體產生了 4 個 Column 欄位來顯示這個 4 個欄位，如下顯示:






現在主表單應該看起來如下所示，請注意 TStringGrid 元件顯示的 VID 和 AID 欄位數值並不是我們希望看到的數值，我們希望 VID 是顯示 Publishers 資料表中的 NAME 欄位值，而 AID 則是顯示 Performers 資料表中的 NAME 欄位值。






為了要達到這個目的我們就需要撰寫繫結運算式了，但想寫繫結運算式我們需要弄清楚繫結運算式的執行範圍，由於 TStringGrid 元件是繫結到 bsdbxBooks，因此在 TStringGrid 元件中撰寫的繫結運算式其執行範圍就是 bsdbxBooks，由於 bsdbxBooks 實作了 IScopeLookup 介面的 Lookup 方法，因此我們就可以藉由 Lookup 方法到 Publishers 和 Performers 資料表中根據 VID 和 AID 查詢 NAME 欄位值，但是在 bsdbxBooks 的執行範圍中要如何能夠存取代表 Publishers 和 Performers 資料表的 bsdbxPublishers 和 bsdbxPerformers 呢?

基本上我們現在要做的事情等於是要從 bsdbxBooks 執行範圍存取到 bsdbxPublishers 和 bsdbxPerformers 的執行範圍，才能夠再分別呼叫 bsdbxPublishers 和 bsdbxPerformers 實作的 Lookup 方法來根據 VID 和 AID 查詢 NAME 欄位值，如果您還想不通的話，那就想想如果是使用程式碼的話，您是不是也要使用類似下面的程式碼來查詢呢?

```
PublishersClientDataSet->Lookup("VID"...);
PerformersClientDataSet->Lookup("AID"...);
```

上面的 PublishersClientDataSet 和 PerformersClientDataSet 也就是類似執行範圍的意思，現在讀者瞭解了嗎?

在 TBindSourceDBX 元件中有一個 ScopeMappings 特性，這個特性的功能就是讓開發人員在其中定義『名稱:執行範圍』的數值，如此一來開發人員就可以藉由 ScopeMappings 特性值中的定義來存取其他的執行範圍，再從其他的執行範圍來執行繫結運算式。

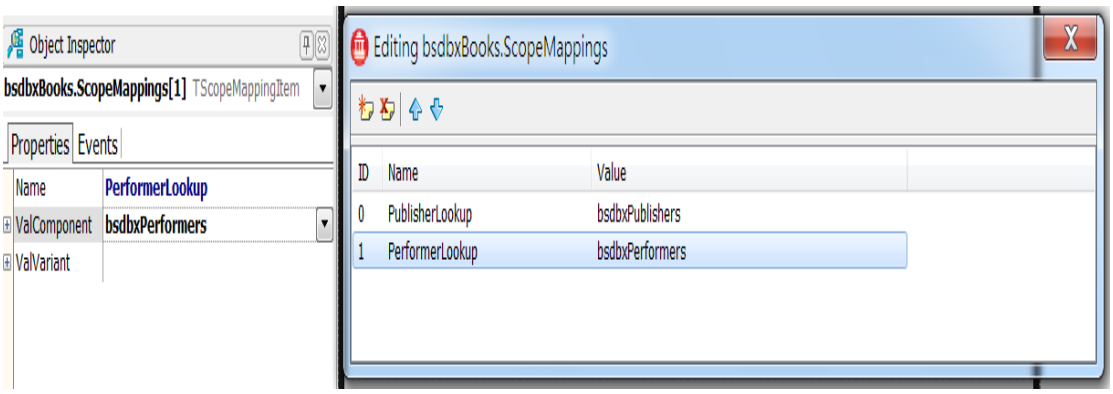
這不就解決了我們的問題了嗎? 讓我們在 bsdbxBooks 的 ScopeMappings 特性中定義如下的數值:

```
PublisherLookup:bsdbxPublishers
PerformerLookup:bsdbxPerformers
```

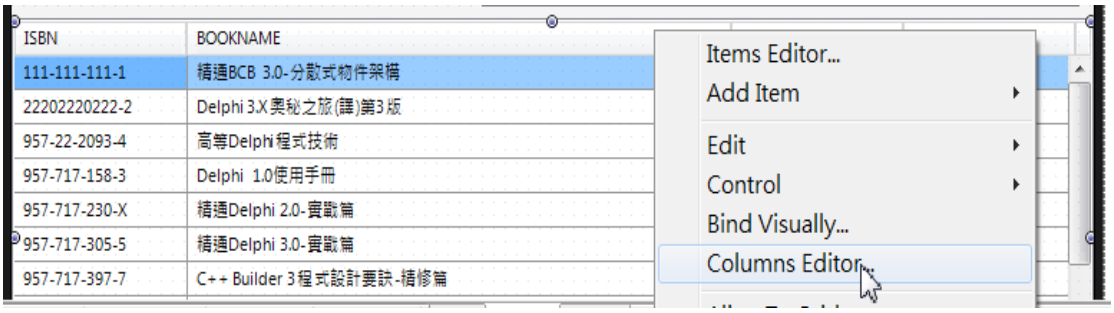
那麼就可以在 bsdbxBooks 的執行範圍中藉由 PublishersLookup 這個識別字來存取 bsdbxPublishers 的執行範圍，再藉由 PerformersLookup 識別字來存取 bsdbxPerformers 的執行範圍了。

現在請點選主表單中的 bsdbxBooks 元件，在物件檢視器中雙擊它的 ScopeMappings 特性，在 ScopeMappings 特性的編輯器中點選左上方的  按鈕以增加新的『名稱:執行範圍』定義，在物件檢視器中於 Name 特性輸入 PublisherLookup，於 ValComponent 特性輸入 bsdbxPublishers。接著用同樣的方法加入『PerformerLookup:bsdbxPerformers』，如下所示:

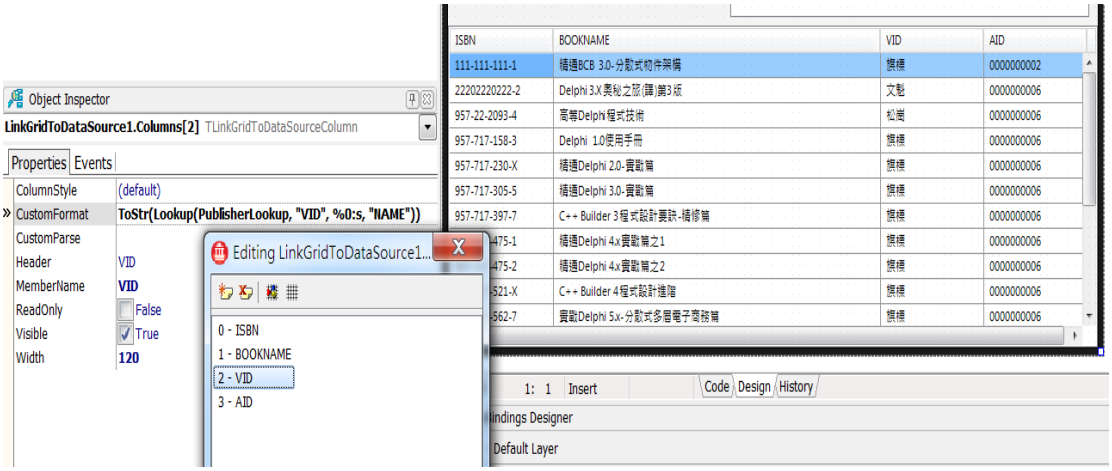





接著點選主表單中的 TStringGrid 元件，右鍵滑鼠從快顯功能表中選擇『Column Editors…』如下所示:




然後從欄位編輯器中選擇 VID 欄位，在它的 CustomFormat 特性中輸入繫結運算式，如下所示:




VID 欄位的 CustomFormat 特性是指當 TStringGrid 元件要顯示 VID 欄位值時，會使用 CustomFormat 特性值中定義的格式來顯示內容，因此我們只需要在 CustomFormat 特性值中以 bsdbxBooks 的 VID 的欄位值從 bsdbxPublishers 執行範圍使用 Lookup 方法根據 bsdbxBooks.VID 來查詢 bsdbxPublishers.NAME 來取代原先要顯示的數值即可，因此請在 VID 欄位的 CustomFormat 特性中輸入如下的繫結運算式:

```
ToStr(Lookup(PublisherLookup, "VID", %0:s, "NAME"))
```

讓我們解釋上面繫結運算式的意義。

先看看

```
Lookup(PublisherLookup, "VID", %0:s, "NAME")
```

它的意思就是在 bsdbxBooks 執行範圍中存取 PublisherLookup 識別字，由於 PublisherLookup 代表 bsdbxPublishers，因此上面的繫結運算式也就代表:

```
Lookup(bsdbxPublishers, "VID", %0:s, "NAME")
```

這個繫結運算式是非常正統的物件導向呼叫慣例，因為在物件導向程式語言中呼叫物件方法時第一個隱藏參數就是物件本身，在 C++Builder 程式語言中就是 **this**，因此上面的繫結運算式可以看成是:

```
bsdbxPublishers->Lookup(bsdbxPublishers, "VID", %0:s, "NAME");
```

也就是:

```
bsdbxPublishers->Lookup("VID", %0:s, "NAME");
```

根據 VID 欄位來查詢並且回傳 NAME 欄位值，上面的第二個參數值%0:s 就是原本 bsdbxBooks 的 VID 欄位值，『%0:s』只是要求 bsdbxBooks 的 VID 欄位值根據這個字串格式帶入到 Lookup 方法中。

上面的繫結運算式中的 ToStr 則是繫結引擎內建的全域方法，把它的參數轉換為字串型態。

現在讀者應該充分瞭解了上面的繫結運算式的意義了吧，繫結運算式並不困難，只要瞭解了執行範圍的觀念之後，就等於呼叫物件的方法或是存取物件的特性值而已。

最後其實上面的繫結運算式中的ToStr方法是不需要呼叫的，為什麼? 因為Lookup方法查詢的 NAME 欄位本身是字串型態的欄位，因此我們可以把上面的繫結運算式最終修改成:



```
Lookup(PublisherLookup, "VID", %0:s, "NAME")
```

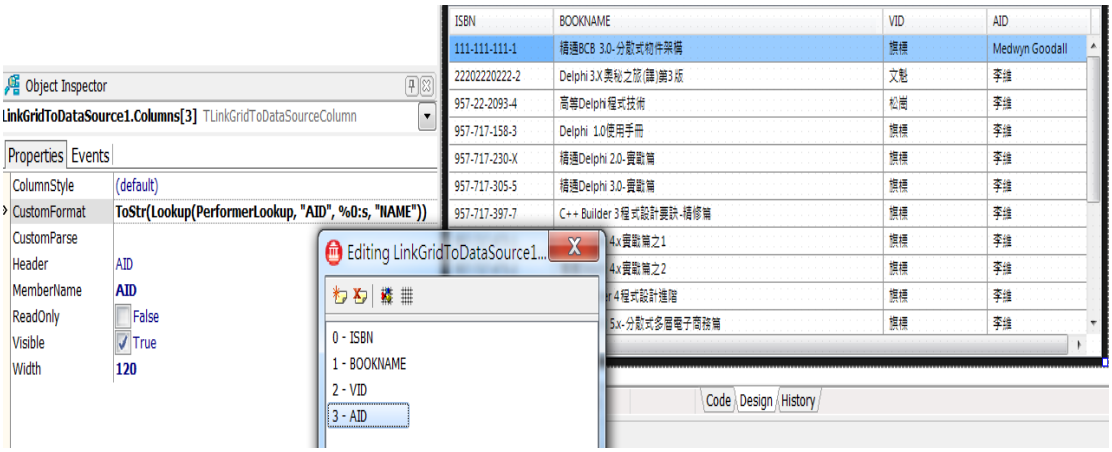
現在請為 AID 欄位進行相同的繫結運算式撰寫流程，當然 AID 欄位的繫結運算式應該是:

```
ToStr(Lookup(PerformerLookup, "AID", %0:s, "NAME"))
```

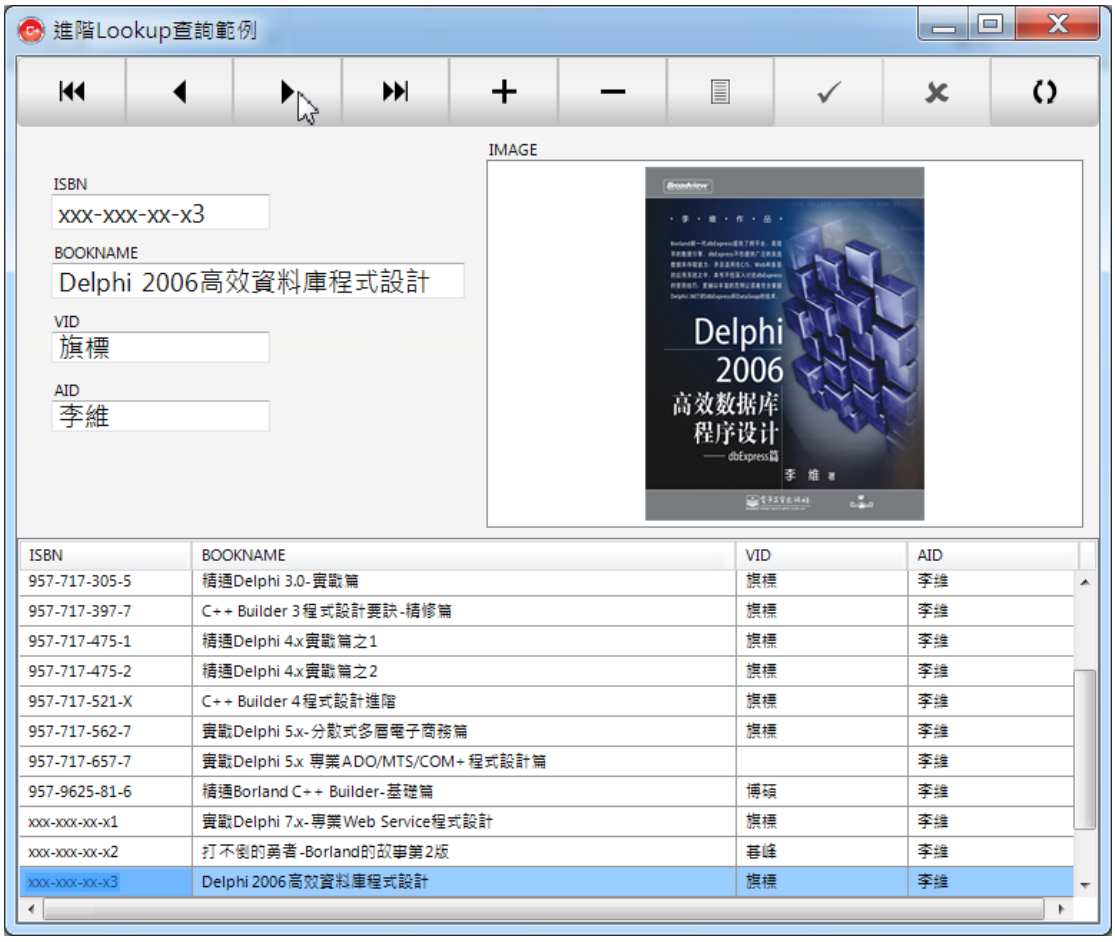
當然 AID 欄位的最終繫結運算式應該寫成:

```
Lookup(PerformerLookup, "AID", %0:s, "NAME")
```

即可，也無需再呼叫 ToStr 全域方法。




現在請編譯和執行此範例 FireMonkey 應用程式，它應該看起來如下所示，讀者可以看到在 TStringGrid 的 VID 和 AID 欄位果然顯示了我們需要的數值了。開發人員在瞭解和掌握了繫結運算式之後就可以在視覺化即時資料繫結設計家無法自動產生繫結運算式的場合中直接使用繫結運算式來進行更具彈性的設計了。



## 8-4 結論

本章說明了如何使用即時資料繫結的 Lookup 功能來進行查詢的功能，也說明了即時資料繫結重要的觀念:繫結運算式的種類，執行範圍等，開發人員必須瞭解這些基本觀念才能夠瞭解視覺化即時資料繫結設計家在做什麼，也能夠自己撰寫繫結運算式和在程式碼中使用繫結物件。

在掌握了即時資料繫結的基本觀念之後開發人員就可以使用繫結運算式來進行高等的 Lookup 功能，以便在視覺化即時資料繫結設計家無法自動產生繫結運算式的場合中直接撰寫繫結運算式。

對於開發一般的 FireMonkey 資料庫應用程式來說，開發人員使用視覺化即時資料繫結設計家，設定 Master/Detail 關係和使用 Lookup 功能應該就能夠完成大多數的開發工作了，再加上開發人員瞭解 dbExpress 框架技術之後，開發有效率的 FireMonkey 資料庫應用程式就應該很簡單了，但要充分掌握即時資料繫結技術，開發人員仍然需要學習繫結類別和繫結物件，這些正是隨後章節討論的重點。

# 第9章 即時資料繫結框架

TOKYO 的視覺化即時資料繫結設計家雖然非常的易於使用，但開發人員要能夠徹底瞭解視覺化即時資料繫結設計家產生的繫結運算式的原理以及正確的使用繫結運算式，那麼開發人員必須掌握即時資料繫結框架。

即時資料繫結框架是由許多類別所組成，不同的類別負責不同繫結運算式的運作，但不管是框架中的那一個類別，基本上開發人員在使用繫結類別和程式碼來運作繫結運算式時，都必須以下列的步驟來執行繫結運算式:



1. 定義繫結執行範圍

2. 撰寫繫結運算式

3. 呼叫繫結物件的 **Evaluate()**方法真正要求繫結引擎根據執行範圍來解釋和執行繫結運算式

在前面討論視覺化即時資料繫結的章節中這 3 個步驟已經由即時資料繫結框架和視覺化即時資料繫結設計家幫我們實行了，因此我們並沒有撰寫任何的程式碼，但為了讓讀者真正的掌握即時資料繫結框架，我們就必須清楚的瞭解如何執行這 3 個步驟，現在讓我們從最簡單的繫結物件開始，藉由程式碼來說明繫結物件如何使用繫結運算式。

## 9-1 建立即時資料繫結概念

讓我們使用繫結運算式框架中最簡單的繫結類別 TBindingExpression 來說明如何前面的觀念，我們將使用 TBindingExpression 類別物件來執行繫結運算式，讓讀者瞭解在使用繫結運算式執行繫結運算式時需要的執行步驟。

首先請在 C++Builder 整合發展環境中建立一個 FireMonkey Desktop Application 專案，並且在主表單中放入一個 TEdit 元件以準備輸入繫結運算式，一個 TMemo 元件以顯示執行繫結運算式的結果，以及一個『Evaluate』按鈕以執行 TEdit 元件中的繫結運算式。

首先在『Evaluate』按鈕的 OnClick 事件處理函式中呼叫 ProcessExpressions() 方法執行 TEdit 元件中使用者輸入的繫結運算式:

```
void __fastcall TForm2::Button1Click(TObject *Sender)
{
  ProcessExpressions();
}
```

下面的程式碼就是 ProcessExpressions()方法的實作程式碼和相關的方法，讓我們詳細的說明這些程式碼的意義:

```
001   void TForm2::DisplayValue(const String sExpression,
System::Bindings::Evalprotocol::_di_IValue AValueIntf)
002   {
003     System::Rtti::TValue LValue;
004     LValue = AValueIntf->GetValue();
005     System::Typinfo::PTypeInfo LType;
006     LType = AValueIntf->GetType();
007     if (LValue.IsObject())
008     {
009     Memo1->Lines->Add("Empty");
010     }
011     else
012     {
013       try
014       {
015         Memo1->Lines->Add(Format("來源運算式 : %s  ==> %s ToString:
'%s'",
016            ARRAYOFCONST(( sExpression, LType->Name,
LValue.ToString()))));
017       }
018       catch(Exception& E)
019       {
```

```
020           Memo1->Lines->Add(E.ClassName() + ": " + E.Message);
021       }
022     }
023   }
024
025   void TForm2::ProcessExpressions()
026   {
027     String LInputExpr;
028     TBindingExpression *pLBindingExpression;
029     System::Bindings::Evalprotocol::_di_IScope pLScope;
030     System::Bindings::Evalprotocol::_di_IScope pLScopes[1];
031     TObject *pLTestObject;
032
033     pLTestObject = new TDictionaryScope();
034     try
035     {
036     pLScope = WrapObject(pLTestObject);
037     pLScopes[0] = pLScope;
038     pLBindingExpression = TBindings::CreateExpression(pLScopes, 0,
edtExpression->Text);
039     try
040     {
041       DisplayValue("繫結運算式 " + edtExpression->Text + " ==> " ,
pLBindingExpression->Evaluate());
042     }
043     __finally
044     {
045       delete pLBindingExpression;
046     }
047     }
048     __finally
049     {
050     pLScope = NULL;
051     delete pLTestObject;
052     }
053   }
```

讓我們先討論 025 行開始的 ProcessExpressions()方法，首先 033 行先建立了 TDictionaryScope 物件，036 行呼叫 WrapObject()方法並且傳遞 TDictionaryScope 物件做為參數，呼叫 WrapObject()方法的目的就是把 TDictionaryScope 物件封裝為執行範圍，也就是說稍後執行的繫結運算式其存取的範圍就是 TDictionaryScope 物件。

TDictionaryScope 類別是繫結運算式框架中一個執行範圍類別，讀者可以把它想成一個空白的執行範圍，它是能夠提供

識別字:執行範圍

的功能，也就是說它可以把一個執行範圍和一個識別字繫結在一起，因此在繫結運算式中使用這個識別字就可以存取到這個執行範圍。

另外執行範圍是可以巢狀化的，這是說開發人員可以在一個執行範圍中再內嵌其他的執行範圍，稍後我們會看到這個巢狀執行範圍的範例，現在讓我們繼續說明上面程式碼的意義。

請讀者看看 038 行，它呼叫 TBindings 類別的類別方法 CreateExpression()以建立一個 TBindingExpression 物件，它的原型宣告如下:

```
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateExpression(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr, const TBindingEventRec &BindingEventRec)/* overload */;
#else /* _WIN64 */
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateExpression(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr, TBindingEventRec &BindingEventRec)/* overload */;
#endif /* _WIN64 */
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateExpression(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr)/* overload */;
```

CreateExpression()方法接受執行範圍和繫結運算式做為參數再回傳建立的TBindingExpression 物件，之後程式碼就在 041 行呼叫 TBindingExpression 物件的Evaluate()方法根據執行範圍來執行繫結運算式。

到這裡為止讀者可以看到上面的程式碼果然是依照:

1. 定義繫結執行範圍 – TDictionaryScope 物件

2. 撰寫繫結運算式 – 038 行

3. 呼叫繫結物件的 **Evaluate()**方法真正要求繫結引擎根據執行範圍來解釋和執行繫結運算式 – 041 行

的步驟來執行繫結運算式。

再讓我們繼續說明下去。

下面是 TBindingExpression 類別的 Evaluate()方法的原型宣告:

```
virtual System::Bindings::Evalprotocol::_di_IValue __fastcall
Evaluate(void) = 0 ;
```

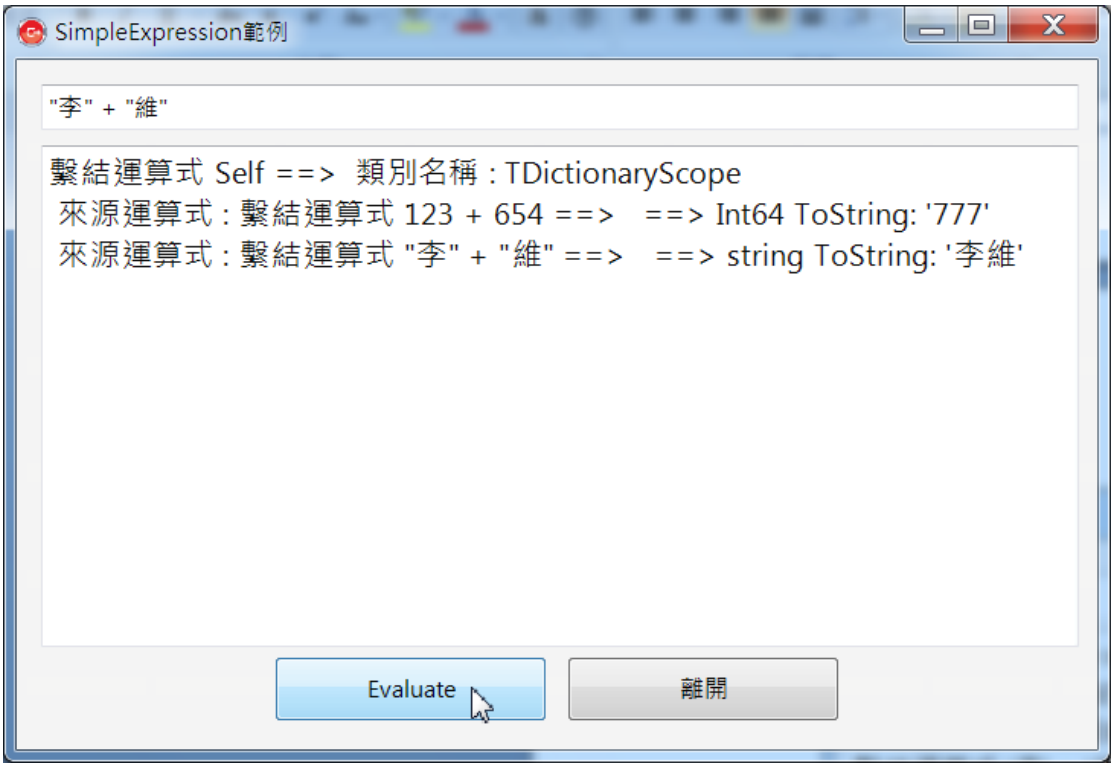
在 TBindingExpression 類別的 Evaluate()方法呼叫繫結引擎執行了繫結運算式之後，Evaluate()方法會回傳 IValue 介面代表執行結果，而下面是 IValue 介面的宣告原型:

```
__interface IValue;
typedef System::C++BuilderInterface<IValue> _di_IValue;
__interface  INTERFACE_UUID("{A495F901-72F5-4384-BA50-EC3B4B42F6C2}")
IValue  : public System::IInterface
{

public:
    virtual System::Typinfo::PTypeInfo __fastcall GetType(void) = 0 ;
    virtual System::Rtti::TValue __fastcall GetValue(void) = 0 ;
};
```

在 IValue 介面中 GetType()方法可以取得執行結果的型態資訊，而 GetValue()則可以取得執行結果值，這些都屬於 C++Builder RTTI 的相關介面和類別，繫結運算式框架使用 RTTI 來動態執行繫結運算式和使用 RTTI 來提供執行結果的資訊。

因此在 DisplayValue()方法中就使用 IValue 介面來取得執行結果並且顯示在主表格的 TMemo 元件中。

現在請編譯和執行此範例程式並且試著在主表格的 TEdit 元件中輸入一些繫結運算式再點選『Evaluate』按鈕來執行繫結運算式。在下面的畫面中您可以看到筆者執行的幾個繫結運算式:



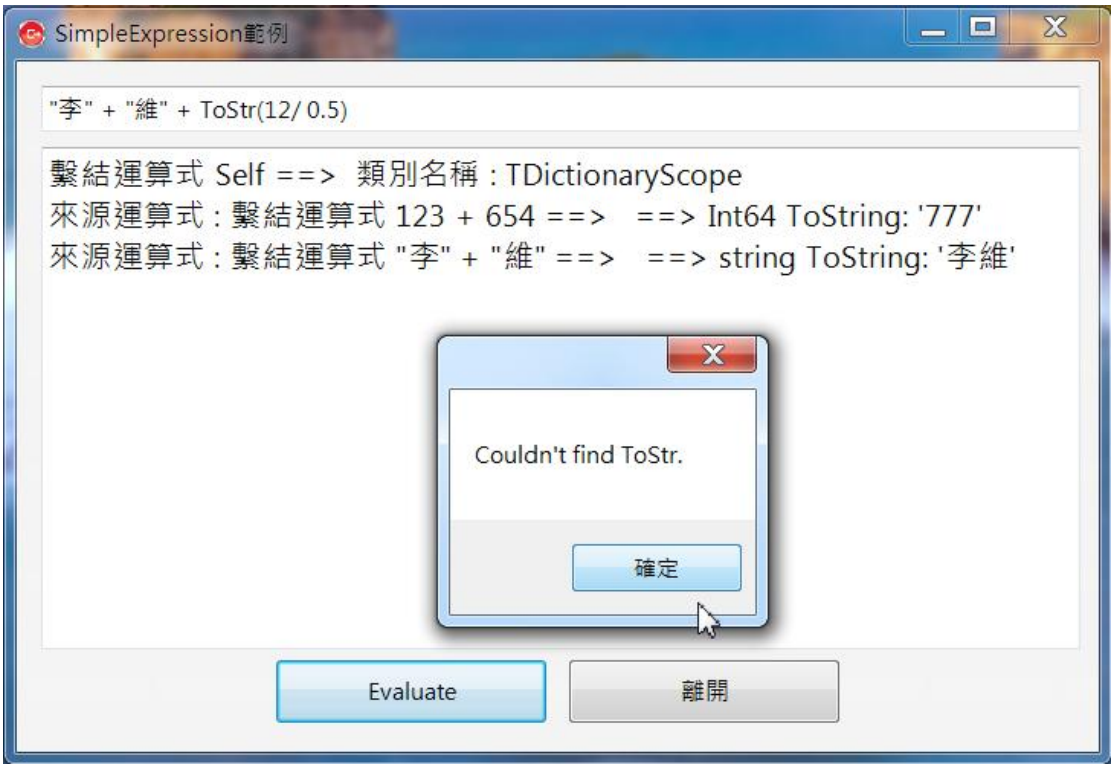

下面的表格說明了這些繫結運算式的意義:

| 繫結運算式 | 執行結果 |
|---|---|
| Self | TDictionaryScope 類別物件，這是因為上面的 031~033 行的程式碼繫結 TDictionaryScope 為執行範圍，而 Self 就是指執行範圍 |
| 123 + 654 | 777，這個繫結運算式可證明簡單的繫結運算式可執行常數值的運算 |
| "李" + "維" | '李維'，這個繫結運算式可證明簡單的繫結運算式可執行字串的簡單運算 |

但現在筆者在下面的畫面中試著執行:

```
"李" + "維" + ToStr(12/ 0.5)
```

卻失敗了，為什麼? 從下圖中我們可以看到範例應用程式說找不到 ToStr()方法，這就是充分的暗示了，因為在目前的執行範圍中沒有 ToStr()方法的定義，因為 TDictionaryScope 類別沒有定義 ToStr()方法。

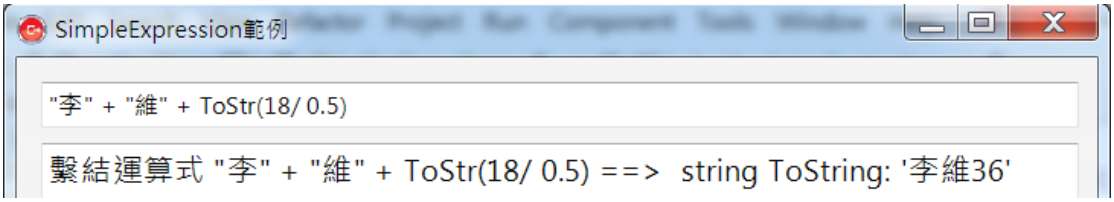



當然如果我們定義的執行範圍中有 ToStr()方法的定義的話那麼在繫結運算式中就可以呼叫 ToStr()，現在讓我們回到範例專案，在 WrapObejct()程式碼下加入新的程式碼 TNestedScope 如下:

```
    pLScope = WrapObject(pLTestObject);
    pNestScope = new TNestedScope(pLScope,
TBindingMethodsFactory::GetMethodScope());
    pLScope = pNestScope->operator _di_IScope();
```

TNestedScope 類別可以在原本的執行範圍中再嵌入另外的執行範圍以形成巢狀式執行範圍，因此在上面的程式碼中是在原本的執行範圍中再加入繫結引擎的全域方法，因為 TBindingMethodsFactory 類別的 GetMethodScope 類別方法即可回傳繫結引擎的全域方法。因此上面這行新加入的程式碼的意思就是在原本只能存取 TDictionaryScope 物件的方法和特性的執行範圍中再加入可存取繫結引擎全域方法的執行範圍。

現在再次執行範例應用程式就可看到在繫結運算式中可以呼叫 ToStr 方法了，因為 ToStr()方法正是繫結引擎的全域方法之一:




我們可以使用下面的程式碼顯示出繫結引擎的全域方法:

```
void TForm2::DisplayMethods()
{
  System::DynamicArray<TMethodDescription> LDescription =
TBindingMethodsFactory::GetRegisteredMethods();

  Memo1->Lines->Add("繫結引擎全域方法: ");
  for (int iIndex = LDescription.Low; iIndex <= LDescription.High;
iIndex++)
  {
    TMethodDescription aMD = LDescription[iIndex];
    if (aMD.DefaultEnabled)
      Memo1->Lines->Add(aMD.Name);
    else
      Memo1->Lines->Add(aMD.Name + " (暫停使用)");
  }
}
```

TBindingMethodsFactory 的類別方法 GetRegisteredMethods()方法可回傳所有在繫結引擎中註冊的全域方法，下面是 GetRegisteredMethods()方法的宣告原型：

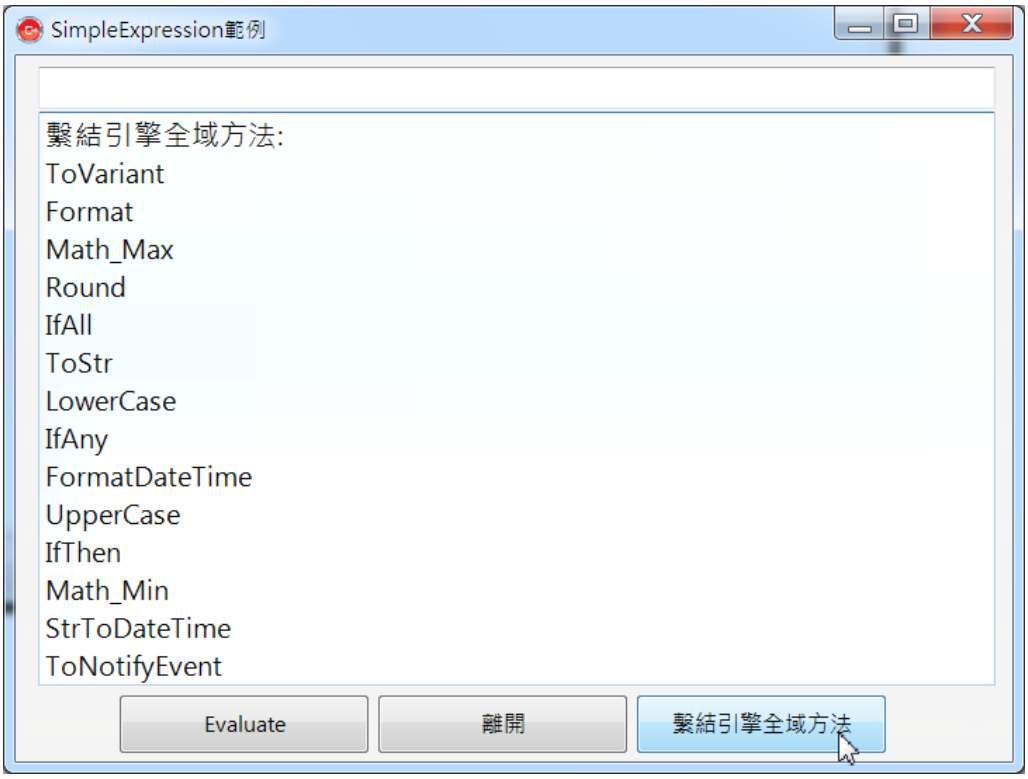
```
__classmethod System::DynamicArray<TMethodDescription> __fastcall
GetRegisteredMethods();
```

GetRegisteredMethods() 方法可回傳 System::DynamicArray<TMethodDescription> 型態的執行結果，在回傳的 DynamicArray 中包含了敘述每一個全域方法的 TMethodDescription 記錄型態物件，從 TMethodDescription 中我們就可以在 for 迴圈中取得全域方法的資訊，例如方法的名稱，方法屬於的程式單元名稱，方法是否作用中，以及可透過 TMethodDescription 記錄型態物件的 Invokable 特性呼叫全域方法等，下面就是 TMethodDescription 記錄型態的定義：

```
struct DECLSPEC_DRECORD TMethodDescription
```

```
{
private:
    System::Bindings::Evalprotocol::_di_IInvokable FInvokable;
    System::UnicodeString FID;
    System::UnicodeString FName;
    System::UnicodeString FUnitName;
    bool FDefaultEnabled;
    System::Classes::TPersistentClass FFrameworkClass;
    System::UnicodeString FDescription;

public:
    __fastcall TMethodDescription(const
System::Bindings::Evalprotocol::_di_IInvokable AInvokable, const
System::UnicodeString AID, const System::UnicodeString AName, const
System::UnicodeString AUnitName, bool ADefaultEnabled, const
System::UnicodeString ADescription, System::Classes::TPersistentClass
AFrameworkClass)/* overload */;
    __property System::UnicodeString ID = {read=FID};
    __property System::UnicodeString Name = {read=FName};
    __property System::UnicodeString UnitName = {read=FUnitName};
    __property bool DefaultEnabled = {read=FDefaultEnabled};
    __property System::Classes::TPersistentClass FrameWorkClass =
{read=FFrameworkClass};
    __property System::Bindings::Evalprotocol::_di_IInvokable
Invokable = {read=FInvokable};
    __property System::UnicodeString Description = {read=FDescription};
    TMethodDescription() {}
};
```

執行上面的程式碼可以看到如下的執行結果:






在 TListBox 中果然顯示出了目前在繫結引擎中註冊的所有全域方法。

再回到範例應用程式，讓我們修改執行範例為一個 TStringList，讀者可以看到在下面的程式碼中我們是建立 TStringList 物件，並且根據 TStringList 物件來建立執行範圍：

```
void TForm2::CreateScopeObejct()
{
  if (pLTestObject == NULL)
  {
    pLTestObject = new TStringList();
    pLScope = WrapObject(pLTestObject);
    pNestScope = new TNestedScope(pLScope,
TBindingMethodsFactory::GetMethodScope());
    pLScope = pNestScope->operator _di_IScope();
  }
}
```

下面的程式碼則是根據上面建立的執行範圍來執行繫結運算式：

```
void __fastcall TForm2::Button4Click(TObject *Sender)
{
```

```
  String LInputExpr;
  TBindingExpression *pLBindingExpression;
  System::Bindings::Evalprotocol::_di_IScope pLScopes[1];

  CreateScopeObejct();
  pLScopes[0] = pLScope;
  pLBindingExpression = TBindings::CreateExpression(pLScopes, 0,
edtExpression->Text);

  try
  {
    DisplayValue("繫結運算式 " + edtExpression->Text + " ==> " ,
pLBindingExpression->Evaluate());
  }
  __finally
  {
    delete pLBindingExpression;
  }
}
```

下面是範例應用程式執行的結果畫面:




下面的表格說明了這些繫結運算式的意義:

| 繫結運算式 | 執行結果 |
|---|---|
| Self | TStringList，這是因為上面的程式碼繫結 TStringList 為執行範圍，而 Self 就是指執行範圍 |
| Self.Add("即時資料繫結") | 使用繫結運算式呼叫 TStringList 物件的 Add 方法在 TStringList 物件中加入一個字串。繫結運算式回傳 0，代表加入的字串索引位置 |
| Self.Add("DataSnap XE7")<br>Self.Add("Mobile Studio") | 同上繫結運算式回傳 1，2，分別代表加入的字串索引位置 |
| Self.Text | 使用繫結運算式存取 TStringList 物件的 Text 特性值 |
| Self.Strings[2] | 使用繫結運算式存取 TStringList 物件的 Strings 特性值，繫結運算式回傳'Mobile Studio' |

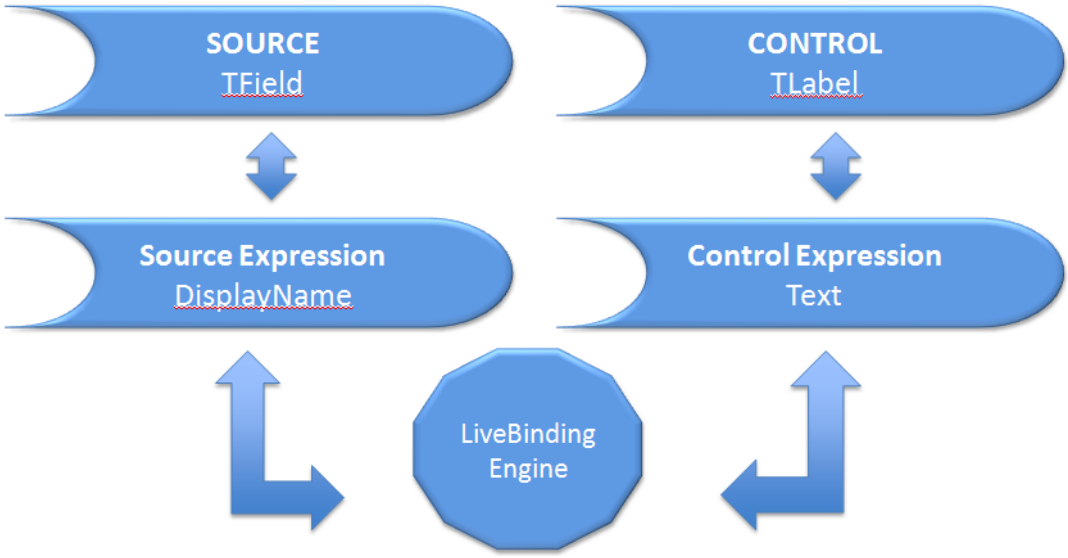
從這個範例我們可以看到在執行範圍內我們可以存取這個執行範圍中物件的方法和特性，就像上面的繫結運算式呼叫 TStringList 物件的方法和存取 TStringList 物件的特性值一樣。這個範例也說明了開發人員幾乎可使用繫結運算式來執行 C++Builder 程式碼可執行的工作，但繫結運算式可在 C++Builder 應用程式執行時動態的改變並重新執行以提供動態語言的機製，這是目前 C++Builder 程式語言無法提供的。

現在讀者應該可以瞭解為什麼繫結框架這麼的強大了，因為 TOKYO 的繫結框架可以為 C++Builder 所有的物件和元件建立執行範圍，一旦建立了執行範圍之後繫結運算式就可以進行處理了。

瞭解了繫結原理之後讓我們直接使用繫結元件來印證一下前面所學習的概念。

## 9-1-1 使用 TBindExpression 元件

在說明如何使用 TBindExpression 元件來使用繫結運算式以印證前面討論的概念之前，請讀者再次觀看一下上一章中已經說明的即時資料繫結架構圖:

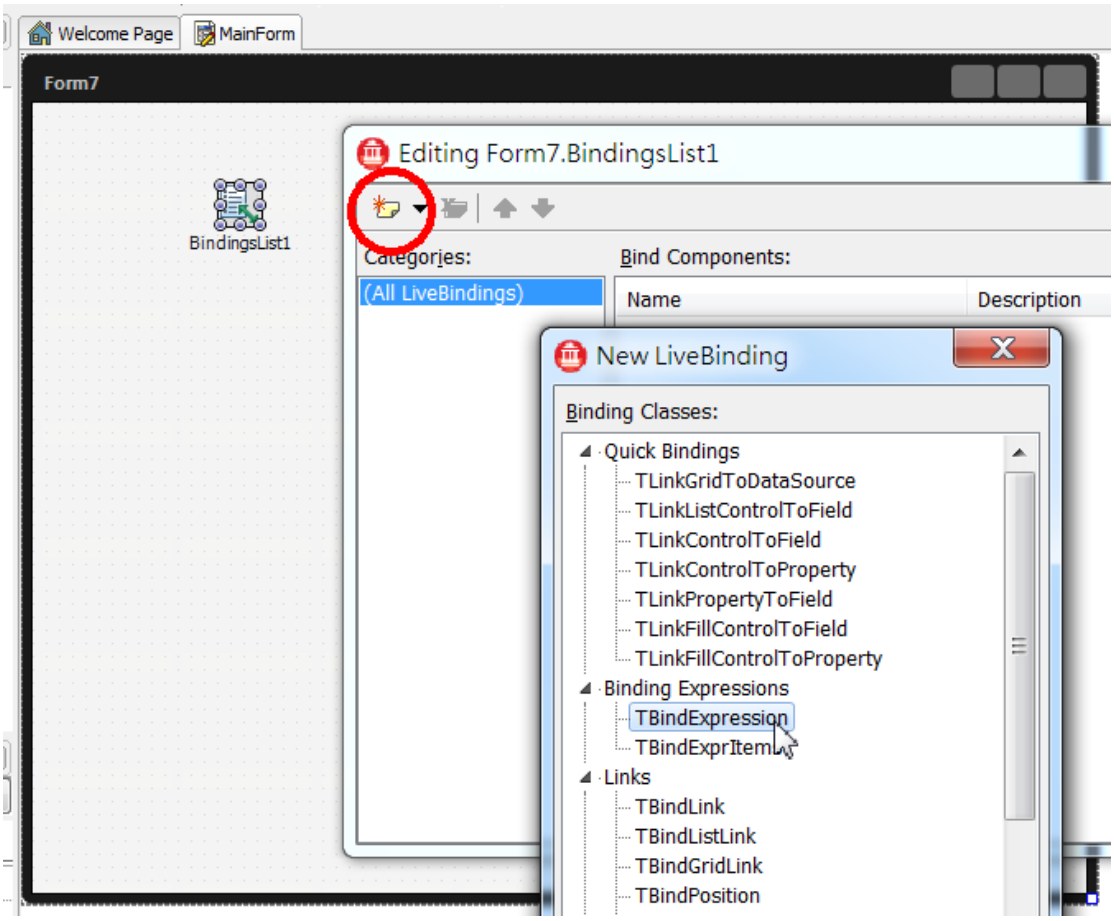



從上圖中可以看到不管是來源元件或是控制元件都有一個繫結運算式，在上一小節中我們只使用了來源繫結運算式，這是因為上一小節是使用程式碼來說明最簡單的繫結運算式應用，但在 C++Builder 支援繫結框架的元件中都是使用上圖的架構，因此 C++Builder 的繫結元件是以下列步驟來提供繫結運算式的運作:

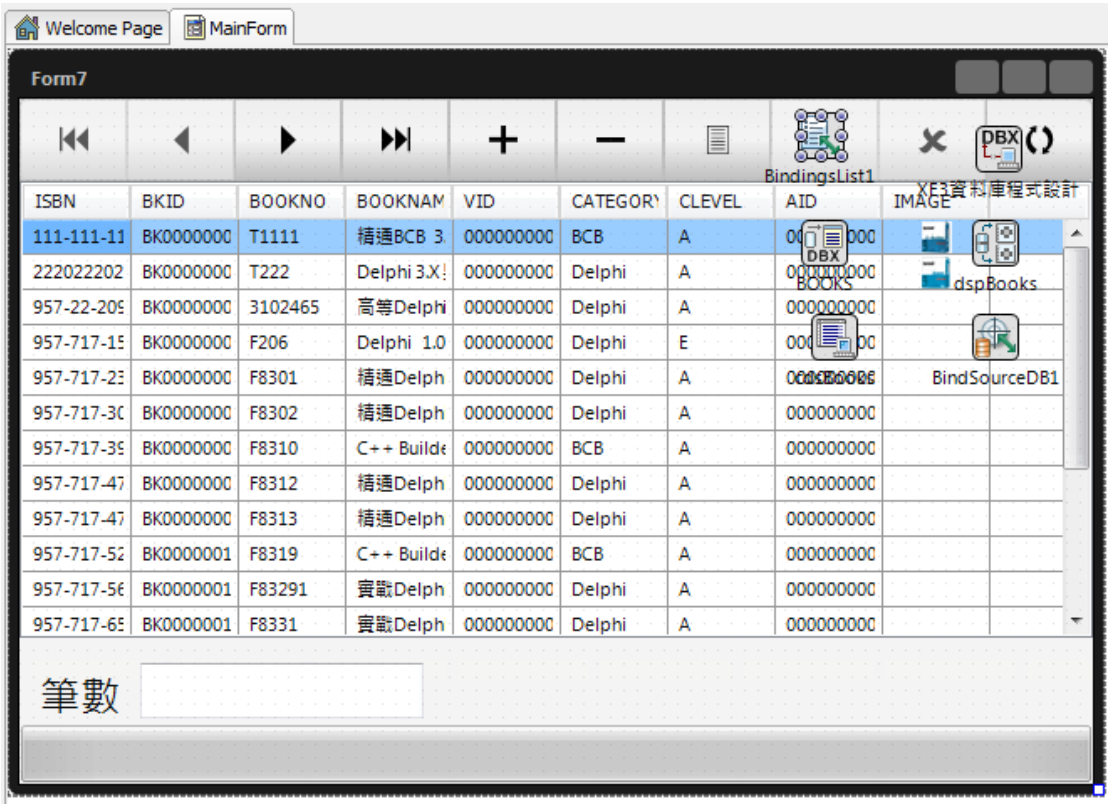
> 來源元件藉由繫結引擎執行來源繫結運算式之後，當來源元件要把繫結運算式的執行結果指定給控制元件時，控制元件也會藉由繫結引擎執行控制繫結運算

式，然後來源元件的執行結果才會根據控制元件的控制繫結運算式的執行結果方式指定給控制元件。

現在我們就可以藉由 C++Builder 的 TBindExpression 元件來印證和說明了，請在 IDE 中建立一個 FireMonkey Desktop Application 專案，在主表單中放入 TBindingsList 元件，雙擊它開啟元件編輯器，再點選左上方的『New Binding』按鈕以新增繫結元件，再於 New LiveBinding 對話盒中選擇建立 TBindExpression 元件，請在其中建立 3 個 TBindExpression 元件，如下所示:




接著在主表單中使用 TSQLConnection 元件連結『XE3 資料庫程式設計』，再使用 TSQLDataSet，TDataSetProvider，TClientDataSet 連結 BOOKS 資料表，再使用 TBindSourceDB 連結 TClientDataSet，最後在連結 TBindNavigator 和 TStringGrid，並且在主表單下方放入 TEdit 和 TProgressBar 元件，我們希望在 TEdit 元件和 TProgressBar 元件中顯示目前記錄的相對位置，我們將使用前面建立的 3 個 TBindExpression 元件來完成這些工作，最後主表單如下所示：






OK，現在想想如何在主表單的 TEdit 元件顯示 cdsBooks 目前記錄的相對位置呢?這很簡單，因為 TClientDataSet 的 RecNo 和 RecordCount 特性值正好是我們要的，我們只需要在 TEdit 中顯示『RecNo/RecordCount』這個資訊即可，因此現在我們就可以寫出兩邊的繫結運算式了。

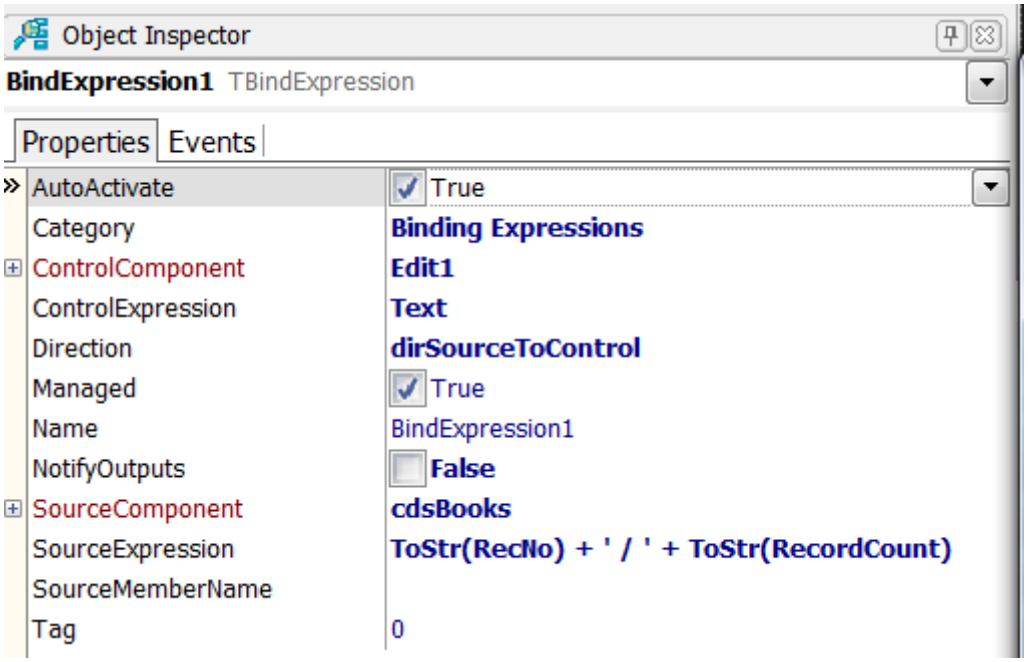
首先是來源方，來源的執行範圍當然是 cdsBooks 元件因為我們要存取 cdsBooks 的 RecNo 和 RecordCount 特性值，而來源繫結運算式則是『RecNo/RecordCount』，但由於控制方的資料來源是字串型態的數值，因此來源繫結運算式應該是 ToStr(RecNo) + '/' + ToStr(RecordCount):

| 來源執行範圍 | 來源繫結運算式 |
|---|---|
| cdsBooks | ToStr(RecNo) + '/' + ToStr(RecordCount) |

控制方的執行範圍當然就是 TEdit 元件了，那麼控制方的繫結運算式呢?由於我們是希望把結果顯示在 TEdit 元件中，因此控制繫結運算式當然就是 TEdit 元件的 Text 特性了，因此最後的結果是:

| 控制執行範圍 | 控制繫結運算式 |
|---|---|
| Edit1 | Text |

瞭解了 2 方的執行範圍和繫結運算式之後，就請點選 TBindingsList 中的第 1 個 TBindExpression 物件，然後在物件檢視器中設定如下的特性值:




在上面的物件檢視器中讀者可以看到 SourceComponent 特性設定為 cdsBooks，SourceExpression 特性設定為 ToStr(RecNo) + '/' + ToStr(RecordCount)，ControlComponent 特性設定為 Edit1，ControlExpression 特性設定為 Text，和前面討論的繫結運算式一模一樣。



同樣的要設定主表單中的 TProgressBar 也可顯示相對位置，那麼我們需要撰寫 2 個繫結運算式，分別繫結 cdsBooks 的 RecNo 和 RecordCount 特性和 TProgressBar 的 Value 和 Max 特性，下面的表格顯示了這 2 個繫結運算式:

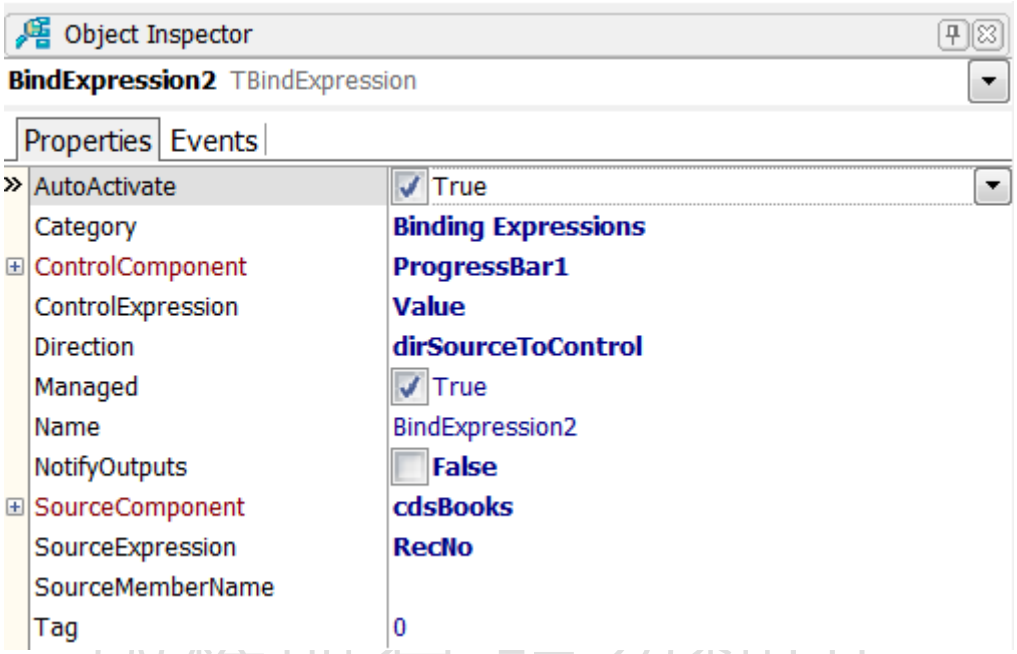
| 來源執行範圍 | 來源繫結運算式 |
|---|---|
| cdsBooks | RecNo |

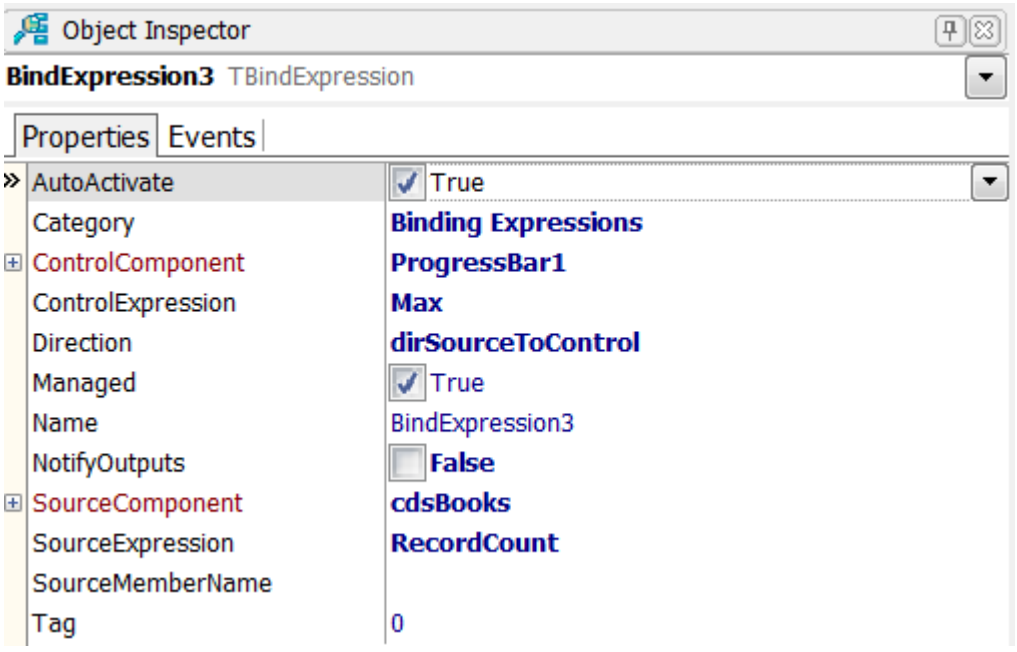
| 控制執行範圍 | 控制繫結運算式 |
|---|---|
| ProgressBar1 | Value |

| 來源執行範圍 | 來源繫結運算式 |
|---|---|
| cdsBooks | RecordCount |

| 控制執行範圍 | 控制繫結運算式 |
|---|---|
| ProgressBar1 | Max |

在下面的 2 個物件檢視器中可以看到根據上面的繫結運算式設定 TBindingsList 元件中第 2 和第 3 個 TBindExpression 物件:

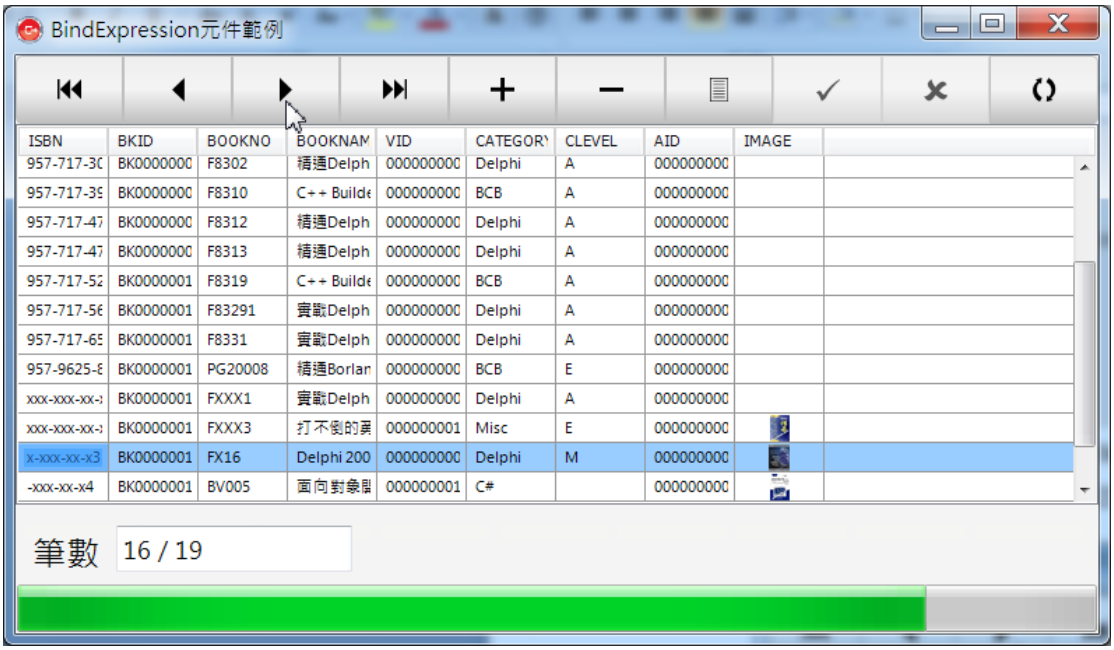



最後回到程式碼中在 cdsBooks 的 AfterOpen 和 AfterScroll 事件處理函式中呼叫 TBindExpression 物件的 Evaluate 方法要求繫結引擎執行上述的繫結運算式:

```
void __fastcall TForm2::cdsBooksAfterOpen(TDataSet *DataSet)
{
  BindExpression1->Evaluate();
  BindExpression3->Evaluate();
}
//---------------------------------------------------------------------------

void __fastcall TForm2::cdsBooksAfterScroll(TDataSet *DataSet)
{
  BindExpression1->Evaluate();
  BindExpression2->Evaluate();
}
```

現在編譯和執行此範例 FireMonkey 應用程式，從下面的畫面中我們可以看到這 2 個繫結運算式果然能夠正確的執行了。




從上面的範例中可以看到我們印證了繫結框架的基本概念，瞭解了上面範例和程式碼的意義後，讀者應該已經掌握了繫結框架的基本概念了，我們就可以再進一步的討論稍微複雜的其他 2 種繫結運算式了。

## 9-1-2 未拖管繫結運算式

未拖管繫結運算式是由應用程式所建立和擁有，當開發人員想執行未拖管繫結運算式中的繫結運算式時必須呼叫未拖管繫結運算式物件的 Evaluate 方法。

未拖管繫結運算式也包含了來源繫結運算式和控制繫結運算式，要建立未拖管繫結運算式開發人員需要呼叫 TBindings 類別的 CreateUnmanagedBinding()類別方法，下面就是 CreateUnmanagedBinding()類別方法的宣告原型：

```
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateUnmanagedBinding(System::Bindings::Evalprotocol::_di_IScope
const *InputScopes, const int InputScopes_Size, const
System::UnicodeString BindExprStr,
System::Bindings::Evalprotocol::_di_IScope const *OutputScopes, const
int OutputScopes_Size, const System::UnicodeString OutputExpr, const
System::Bindings::Outputs::_di_IValueRefConverter OutputConverter,
const TBindingEventRec &BindingEventRec, TCreateOptions Options =
(TCreateOptions() << TCreateOption::coNotifyOutput ))/* overload */;
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateUnmanagedBinding(System::Bindings::Evalprotocol::_di_IScope
const *InputScopes, const int InputScopes_Size, const
System::UnicodeString BindExprStr,
System::Bindings::Evalprotocol::_di_IScope const *OutputScopes, const
int OutputScopes_Size, const System::UnicodeString OutputExpr, const
System::Bindings::Outputs::_di_IValueRefConverter OutputConverter,
TCreateOptions Options = TCreateOptions() )/* overload */;
```

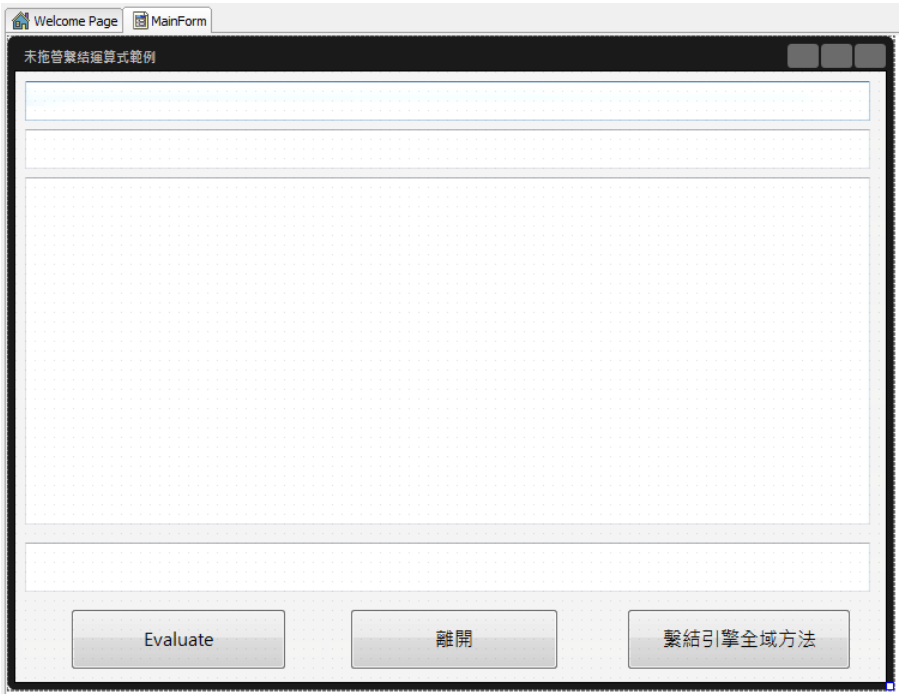
CreateUnmanagedBinding()有 2 個複載原型，分別接受不同的參數，下面的表格說明了這些參數的意義：

| 參數名稱 | 參數說明 |
|---|---|
| InputScopes | 來源執行範圍 |
| InputScopes_Size | 第 1 個參數傳入的陣列元素數目 |
| BindExprStr | 來源繫結運算式 |
| OutputScopes | 控制執行範圍 |
| OutputExpr | 控制繫結運算式 |
| OutputConverter | 輸出資料型態轉換函式 |
| BindingEventRec | 繫結事件處理函式 |
| Options | 影響繫結運算式執行行為的選項 |

CreateUnmanagedBinding()類別方法回傳 TBindingExpression 物件，這個物件就是未拖管繫結物件，它可以使用來執行未拖管繫結運算式。

輸出資料型態轉換函式和繫結事件處理函式在稍後會說明，現在讓我們使用一個範例來說明。

在 IDE 中建立一個 FireMonkey Desktop Application 專案，在主表單中放入 3 個 TEdit 元件，一個 TMemo 元件和 3 個 TButton 元件，如下所示：




在這個範例中我們將繫結 TStringList 物件和主表單中下方的 TEdit 元件，而主表單上方的 2 個 TEdit 元件則是用來輸入來源繫結運算式和控制繫結運算式。

下面是主表單中 Evaluate 按鈕的 OnClick 的實作程式碼中:

```
001   void TForm2::ProcessExpressions()
002   {
003     String LInputExpr;
004     TBindingExpression *pLBindingExpression;
005     System::Bindings::Evalprotocol::_di_IScope pSourceScopes[1];
006     System::Bindings::Evalprotocol::_di_IScope pSourceScope;
007     System::Bindings::Evalprotocol::_di_IScope pControlScopes[1];
008     System::Bindings::Evalprotocol::_di_IScope pControlScope;
```

```
009     TObject *pLTestObject;
010     TNestedScope *pNestScope;
011
012     pLTestObject = new TStringList();
013     ((TStringList*) pLTestObject)->Add("即時資料繫結");
014     ((TStringList*) pLTestObject)->Add("FM2");
015     ((TStringList*) pLTestObject)->Add("Mobile Studio");
016     try
017     {
018     pSourceScope = WrapObject(pLTestObject);
019     pNestScope = new TNestedScope(pSourceScope,
TBindingMethodsFactory::GetMethodScope());
020     pSourceScope = pNestScope->operator _di_IScope();
021     pSourceScopes[0] = pSourceScope;
022
023     pControlScope = WrapObject(edtControl);
024     pNestScope = new TNestedScope(pControlScope,
TBindingMethodsFactory::GetMethodScope());
025     pControlScope = pNestScope->operator _di_IScope();
026     pControlScopes[0] = pControlScope;
027
028      pLBindingExpression = TBindings::CreateUnmanagedBinding(
029       pSourceScopes,
030       0,
031       edtSourceExpression->Text,
032       pControlScopes,
033       0,
034       edtControlExpression->Text,
035       NULL);
036
037     try
038     {
039       DisplayValue("繫結運算式 " + edtSourceExpression->Text + " ==>
" ,  pLBindingExpression->Evaluate());
040     }
041     __finally
```

```
042     {
043       delete pLBindingExpression;
044     }
045     }
046     __finally
047     {
048     pSourceScope = NULL;
049     pControlScope = NULL;
050     delete pLTestObject;
051     }
052   }
```

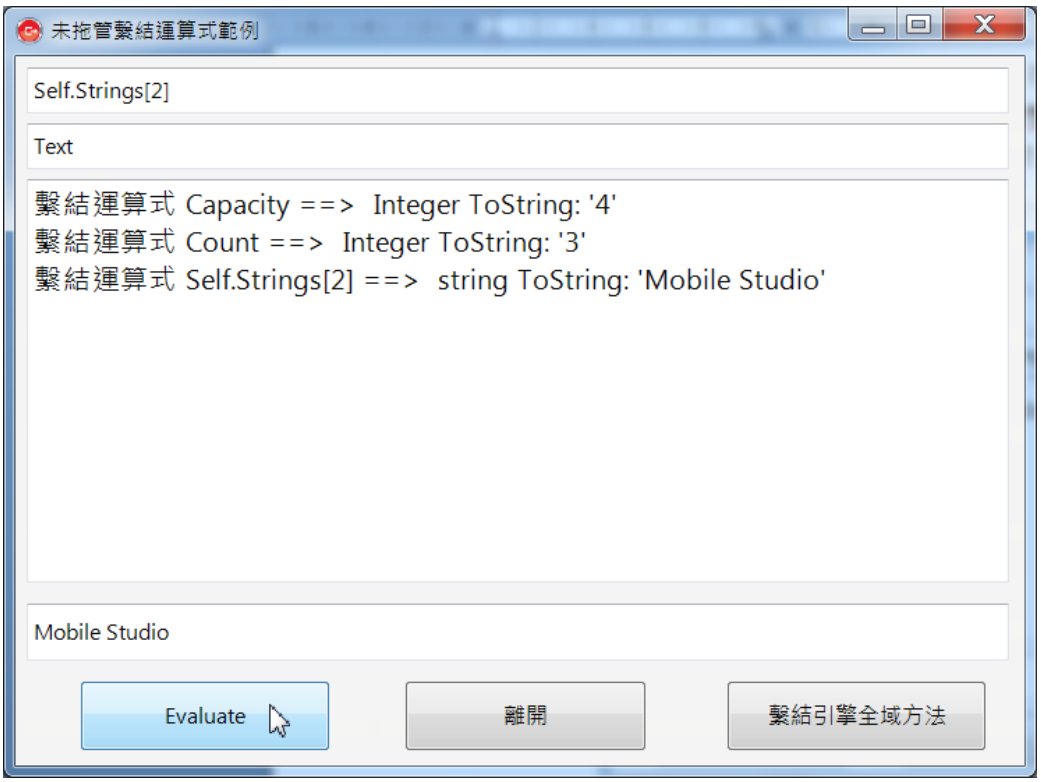
下面的表格說明了上面程式碼的意義:

| 程式碼行數 | 程式碼說明 |
|---|---|
| 012 | 建立來源物件 TStringList |
| 013~015 | 在來源物件 TStringList 中加入 3 個字串資料以準備稍後展示用未拖管繫結運算式來操作 |
| 018~021 | 建立來源執行範圍，即 TStringList 物件 |
| 023~026 | 建立控制執行範圍，即主表單下方的 TEdit 元件 |
| 028~035 | 呼叫 TBindings 的 CreateUnManagedBinding()方法建立未拖管繫結運算式物件，分別把來源執行範圍，來源繫結運算式，控制執行範圍，控制繫結運算式傳入當參數。請注意最後一個參數是 Nil，代表使用繫結引擎內定的輸出資料型態轉換函式 |
| 039 | 呼叫未拖管繫結運算式物件的 Evaluate()方法執行繫結運算式，並且藉由 DisplayValue()方法在主表單中的 TMemo 元件顯示執行資訊 |



從上面的程式碼就可以知道，一旦在主表單上方的 2 個 TEdit 元件中輸入繫結運算式之後，再點選下方的 Evaluate 按鈕就會在主表單下方的 TEdit 中顯示繫結的運算結果。

現在執行範例 FireMonkey 應用程式，在下面的畫面中讀者可以看到筆者在主表單上方的 2 個 TEdit 元件中輸入來源繫結運算式和控制運算式之後，點選『Evaluate』按鈕之後在 TMemo 和下方 TEdit 元件的執行結果:



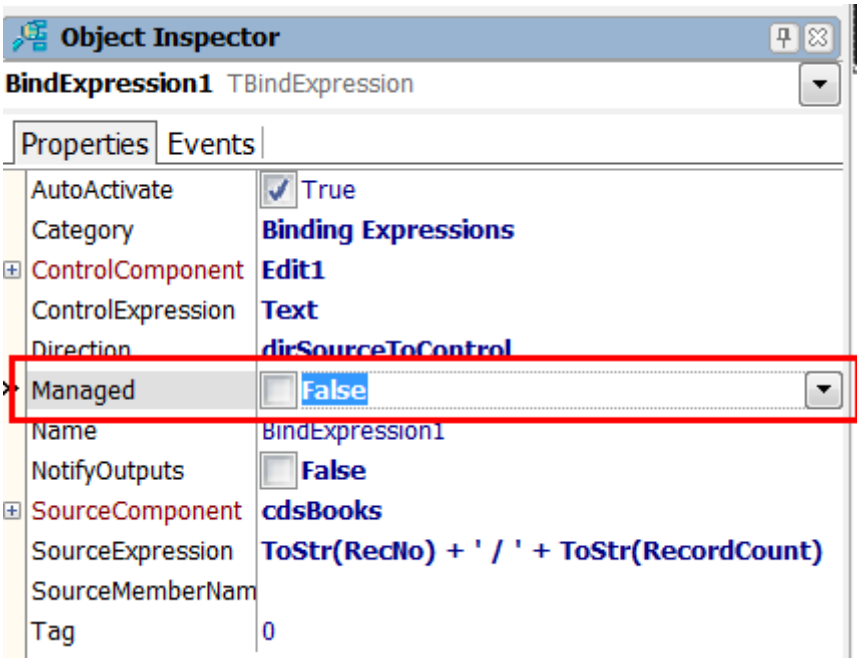



由於範例 FireMonkey 應用程式繫結了 TStringList 和主表單下方的 TEdit 元件，因此在點選『Evaluate』按鈕之後繫結運算式執行的結果會立刻自動出現在主表單下方的 TEdit 元件中，下面的表格說明了執行的繫結運算式以及執行的結果:

| 來源繫結運算式 | 控制繫結運算式 | edtControl.Text |
|---|---|---|
| Capacity | Text | 4 |
| Count | Text | 3 |
| Self.Strings[2] | Text | Mobile Studiio |

從這個範例我們也可以看到未拖管繫結運算式也可以繫結任何的物件和元件，非常具有彈性，此外從這個範例讀者也可以瞭解在使用未拖管繫結運算式時，執行範圍也有 2 個，即來源執行範圍和控制執行範圍，來源繫結運算式是在來源執行範圍中執行而控制繫結運算式是在控制執行範圍中執行。

那麼如果我們是直接在應用程式中使用繫結框架的元件時要如何使用未拖管繫結運算式呢?很簡單，因為 C++Builder 的繫結元件都有一個『Managed』特性，它的內定值是 True，代表 C++Builder 的繫結元件都是拖管繫結運算式，如果開發人員要使用未拖管繫結運算式，那麼只需要取消這個特性即可。例如下圖上面小節使用的範例 TBindExpression 元件，它的『Managed』特性在物件檢視器中是勾選的，要讓它成為未拖管繫結運算式，只需要如下取消『Managed』特性即可。




## 9-1-3 拖管繫結運算式

拖管繫結運算式和未拖管繫結運算式很類似，也擁有來源繫結運算式和控制繫結運算式，但這 2 者不同的地方是拖管繫結運算式是由繫結引擎所擁有，因此當繫結運算式中有任何的繫結元件或是繫結運算式改變時，在繫結引擎中相關的拖管繫結運算式就會自動被重新執行，而無需由開發人員主動呼叫 Evaluate 方式。



這個意思是說，假設在繫結引擎中擁有 2 個拖管繫結運算式，這 2 個拖管繫結運算式中都繫結到了 A 元件的 B 特性，那麼如果第二個拖管繫結運算式改變了 A 元件的 B 特性值，那麼繫結引擎就會自動重新執行第一個拖管繫結運算式。不過開發人員仍然可以呼叫拖管繫結運算式的 Evaluate 方法主動要求執行拖管繫結運算式。

要建立拖管繫結運算式開發人員需要呼叫 TBindings 類別的 CreateManagedBinding()類別方法，下面就是 CreateManagedBinding()類別方法的宣告原型：

```
#ifndef _WIN64
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateManagedBinding(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr, System::Bindings::Evalprotocol::_di_IScope const
*OutputScopes, const int OutputScopes_Size, const System::UnicodeString
```

```
OutputExpr, const System::Bindings::Outputs::_di_IValueRefConverter
OutputConverter, const TBindingEventRec &BindingEventRec,
System::Bindings::Manager::TBindingManager* Manager =
(System::Bindings::Manager::TBindingManager*)(0x0), TCreateOptions
Options = (TCreateOptions() << TCreateOption::coNotifyOutput ))/*
overload */;
#else /* _WIN64 */
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateManagedBinding(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr, System::Bindings::Evalprotocol::_di_IScope const
*OutputScopes, const int OutputScopes_Size, const System::UnicodeString
OutputExpr, const System::Bindings::Outputs::_di_IValueRefConverter
OutputConverter, TBindingEventRec &BindingEventRec,
System::Bindings::Manager::TBindingManager* Manager =
(System::Bindings::Manager::TBindingManager*)(0x0), TCreateOptions
Options = (TCreateOptions() << TCreateOption::coNotifyOutput ))/*
overload */;
#endif /* _WIN64 */
    __classmethod System::Bindings::Expression::TBindingExpression*
__fastcall
CreateManagedBinding(System::Bindings::Evalprotocol::_di_IScope const
*InputScopes, const int InputScopes_Size, const System::UnicodeString
BindExprStr, System::Bindings::Evalprotocol::_di_IScope const
*OutputScopes, const int OutputScopes_Size, const System::UnicodeString
OutputExpr, const System::Bindings::Outputs::_di_IValueRefConverter
OutputConverter, System::Bindings::Manager::TBindingManager* Manager =
(System::Bindings::Manager::TBindingManager*)(0x0), TCreateOptions
Options = (TCreateOptions() << TCreateOption::coNotifyOutput ))/*
overload */;
```

CreateManagedBinding()也有 3 個複載原型，而且其參數的意義和前面介紹 CreateUnmanagedBinding()方法的參數是一樣的，而且 CreateManagedBinding()方法也是回傳 TBindingExpression 物件，不過這個 TBindingExpression 物件是拖管繫結物件。

C++Builder 的繫結元件在內定上都是屬於拖管繫結運算式，在稍後的章節中會有範例說明。

## 9-2 資料型態轉換函式

當開發人員使用繫結引擎把來源運算式的執行結果指定給控制元件時，來源運算式的執行結果的資料型態可能和控制繫結運算式執行結果的資料型態不同，在這種情形中就需要進行資料型態轉換的工作。例如如果把 TStringList 物件的 Count 特性值繫結到 TEdit 元件的 Text 特性，那麼由於 Count 是整數值而 Text 是字串型態，因此來源繫結運算式必須把執行結果從整數型態轉換為字串型態。

在繫結引擎中在來源方和控制方進行自動資料型態轉換的函式就稱為來『輸出轉換元(OutputConverters)』，基本上開發人員在使用繫結運算式時並不需要特別的注意輸出轉換元，因為繫結引擎會藉由 RTTI 自動判斷是否需要呼叫特定的輸出轉換元來進行資料型態轉換的工作。

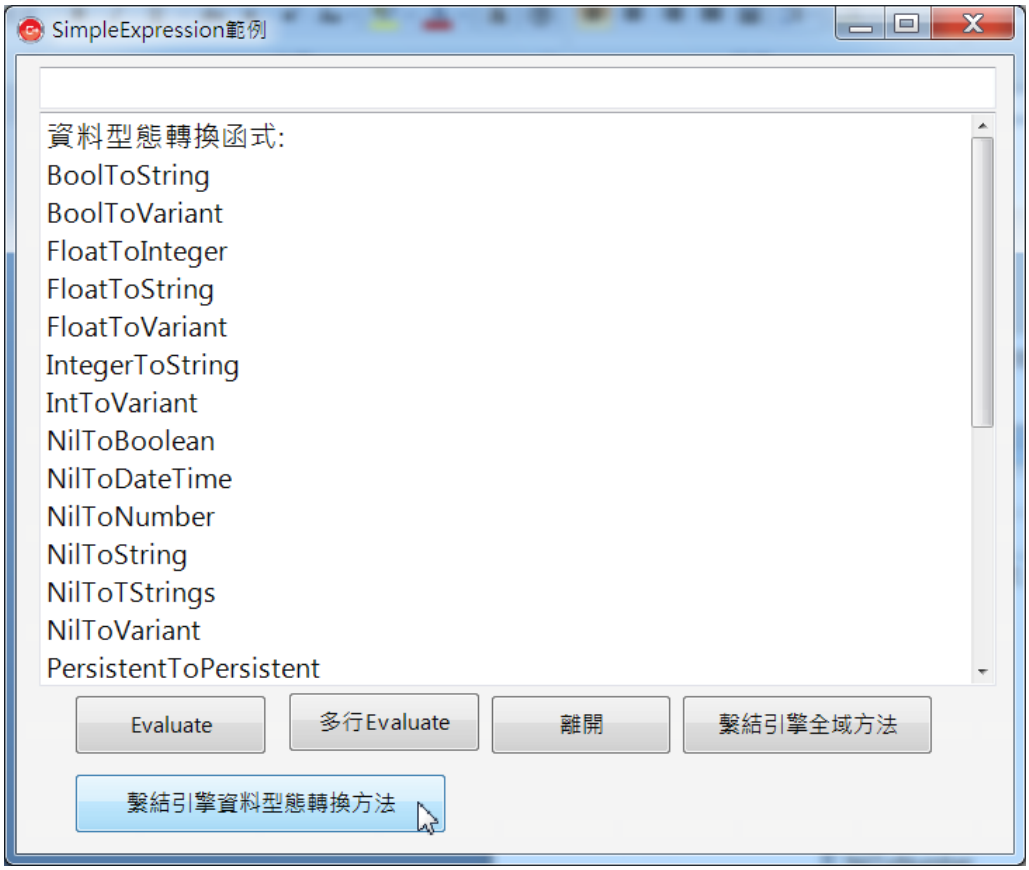
我們可以使用 TValueRefConverterFactory 類別的類別方法 GetConverterDescriptions()來取得目前繫結引擎中所有的輸出轉換元，例如使用下面的程式碼就可以得到輸出轉換元的資訊:



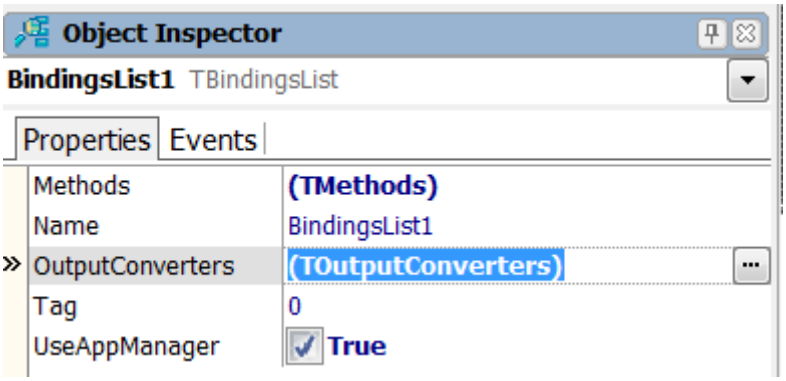
```
void TForm2::DisplayConverters()
{
  System::DynamicArray<TConverterDescription> LDescription =
TValueRefConverterFactory::GetConverterDescriptions();
  TStringList *pLStrings;
  int iCount;

  pLStrings = new TStringList();
  try
  {
    for (iCount = LDescription.Low; iCount <= LDescription.High;
iCount++)
    {
      TConverterDescription aTD = LDescription[iCount];
      if (aTD.DefaultEnabled)
        pLStrings->Add(aTD.DisplayName);
      else
```

```
        pLStrings->Add(aTD.DisplayName + " (disabled)");
    }
    pLStrings->Sort();

    Memo1->Lines->Add("資料型態轉換函式: ");
    for (iCount = 0; iCount < pLStrings->Count; iCount++)
      Memo1->Lines->Add(pLStrings->Strings[iCount]);
  }
  __finally
  {
    delete pLStrings;
  }
}
```

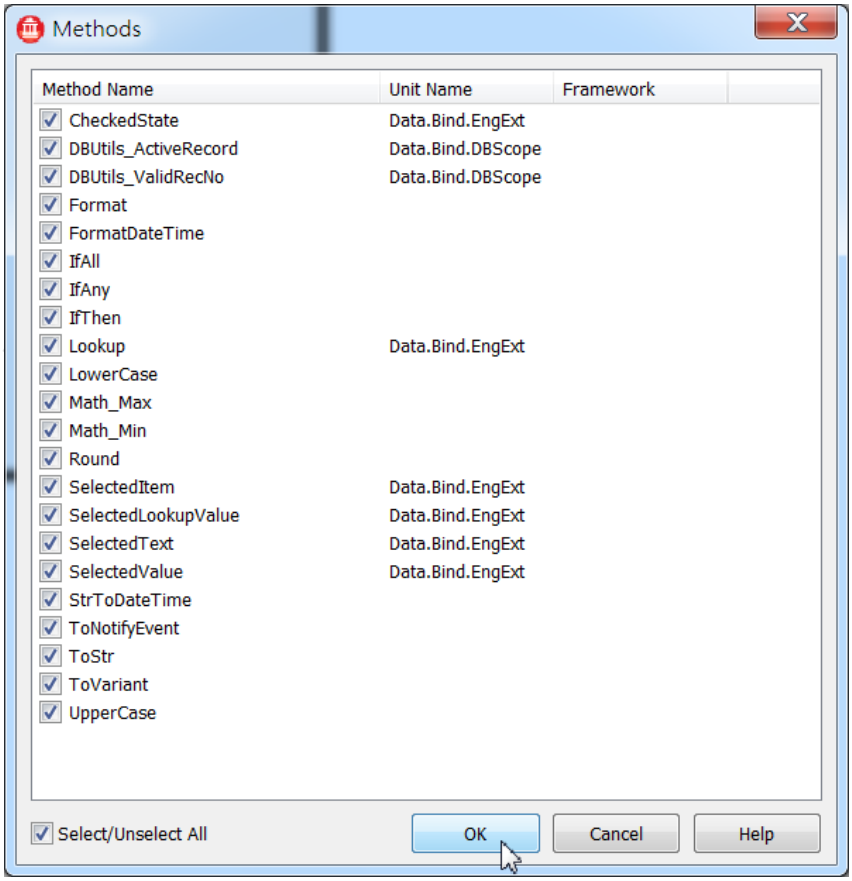
在下面的畫面中讀者可以看到繫結引擎中所有的內建輸出轉換元：

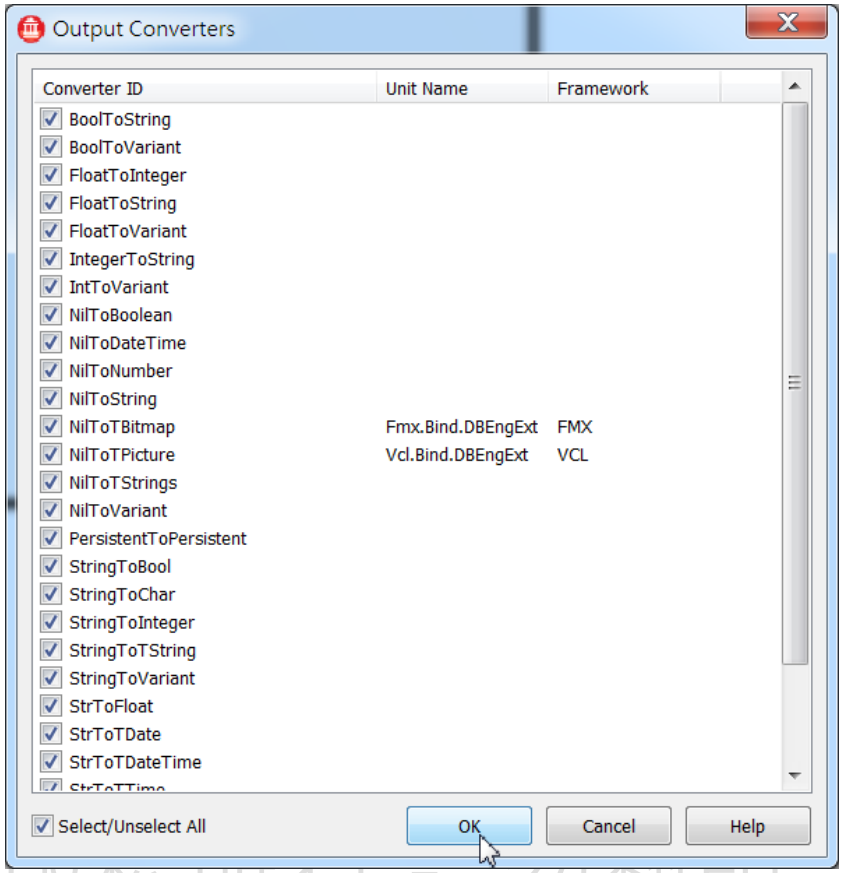



在 TBindingsList 元件中就擁有繫結引擎中的全域方法和輸出轉換元，它們分別在 TBindingsList 元件的 Methods 和 OutputConverters 特性中:




如果讀者雙擊上面的 2 個特性值就可以看到下面的 2 個對話盒其中列出了目前繫結引擎的全域方法和輸出轉換元:








如果開發人員需要特別的輸出轉換元來進行資料型態轉換，那麼開發人員也可以向繫結引擎註冊客製化的輸出轉換元。

## 9-3 即時資料繫結相關的類別

TOKYO 的即時資料繫結框架提供了許多不同的即時資料繫結類別以提供不同的服務，開發人員在使用即時資料繫結技術時可以根據需求選擇使用不同的即時資料繫結類別，在前面討論中的內容中本書已經介紹過數個不同的即時資料繫結類別，例如 TBindExpression 下面的表格說明了每一個即時資料繫結類別的功能:

| 類別 | 說明 |
|---|---|
| TBindExpression | 繫結單一來源物件到控制物件的類別，可提供雙向繫結，或是單向繫結 |
| TBindExprItems | 繫結單一來源物件到控制物件的類別，其中可包含數個不同的函數式，分別儲存在其 FormatExpressions 和 ClearExprerssions 特性中 |
| TBindLink | 繫結單一來源物件到控制物件的類別，其中可包含數個不 |

| | |
|---|---|
| | 同的函數式，分別儲存在其 FormatExpressions，ParseExpressions 和 ClearExprerssions 特性值中。通常是使用在資料庫相關的繫結。 |
| TBindListLink | 繫結單一來源物件到串列型態的控制物件的類別，其中可包含數個不同的函數式，分別儲存在其 FormatExpressions，FormatControlExpressions，ParseExpressions 和 ClearExprerssions，ClearControlExprerssions 特性值中。通常是使用在資料庫相關的繫結。 |
| TBindGridLink | 繫結 Grid 元件到資料來源的類別，例如繫結 FireMonkey 的 TStringGrid 或是 VCL 的 TStringGrid 元件到資料來源。通常是使用在資料庫相關的繫結。 |
| TBindPosition | 繫結具有位置相關特性的元件，例如繫結 TScrollBar，TProgressBar 和 TTrackBar 等元件和資料來源。 |
| TBindControlValue | 繫結控制物件到元件的特性，當控制物件被使用者更新後繫結的元件特性也會自動更新 |
| TBindList | 繫結單一來源物件到串列型態的控制物件的類別，例如繫結資料結構中的資料到 TListBox 等應用。 |
| TBindGridList | 繫結單一來源物件到串列型態的控制物件的類別，例如繫結資料結構中的資料到 TStringGrid 等應用 |



除了這些 XE2 就出現的繫結類別之外，TOKYO 又增加了『Quick Bindings』類別，這些新的快速繫結類別是為了和視覺化即時資料繫結設計家結合以提供開發人員快速開發資料庫相關的繫結工作，因此原先在 XE2 中的『DB Links』繫結類別就宣告為過時的繫結類別，從 TOKYO 之後開發人員應該這些新的快速繫結類別來繫結資料來源。下面的表格說明了這些新的快速繫結類別的意義:

| 類別 | 說明 |
|---|---|
| TLinkGridToDataSource | 繫結 TGrid 或是 TStringGrid 元件到資料來源，因此來源元件是 TGrid 或是 TStringGrid 元件，控制元件是資料來源 |
| TLinkListControlToField | 繫結串列元件到資料集中的欄位，例如繫結 TListBox 等串列到 TClientDataSet 的欄位。來源元件是串列元件，控制元件是欄位物件 |
| TLinkControlToField | 繫結控制項到欄位，例如繫結 TEdit，TComboBox 到 TClientDataSet 的欄位。來源元件是控制項，控制元件是欄位物件 |

| | |
|---|---|
| TLinkControToProperty | 繫結控制項到某元件的特性，例如繫結 TEdit 到 TButton 的 Text 特性，來源元件是控制項，控制元件是元件的特性 |
| TLinkPropertyToField | 繫結元件的特性到欄位，例如繫結 TEdit 的 Text 特性值到 TClientDataSet 的欄位，來源元件是元件的特性值，控制元件是欄位物件 |
| TLinkFillControlToField | 繫結需要填入資料的元件到資料集中的欄位，例如繫結 TComboBox 到 TClientDataSet 的欄位。來源元件是填入資料控制項，控制元件是欄位物件 |
| TLinkFillControlToProperty | 繫結需要填入資料的元件到元件的特性，例如繫結 TComboBox 到 TEdit 的 Text 特性。來源元件是填入資料控制項，控制元件是元件的特性 |

讓我們繼續使用 BOOKS 資料表來說明如何使用這些繫結類別讀者就可以很容易的瞭解了。

## 9-3-1 使用 TBindExprItems 類別



TBindExprItems 繫結類別可同時包含數個繫結運算式，因此我們可以把它看成是多個 TBindExpression 類別物件的集合類別，開發人員藉由在它的 Format 特性中撰寫任意數目的繫結運算式，也可以在它的 Clear 特性中撰寫清除繫結運算式，但 TBindExprItems 仍然是繫結來源元件和控制元件，因此這些多個繫結運算式仍然是執行在這 2 方的執行範圍中。

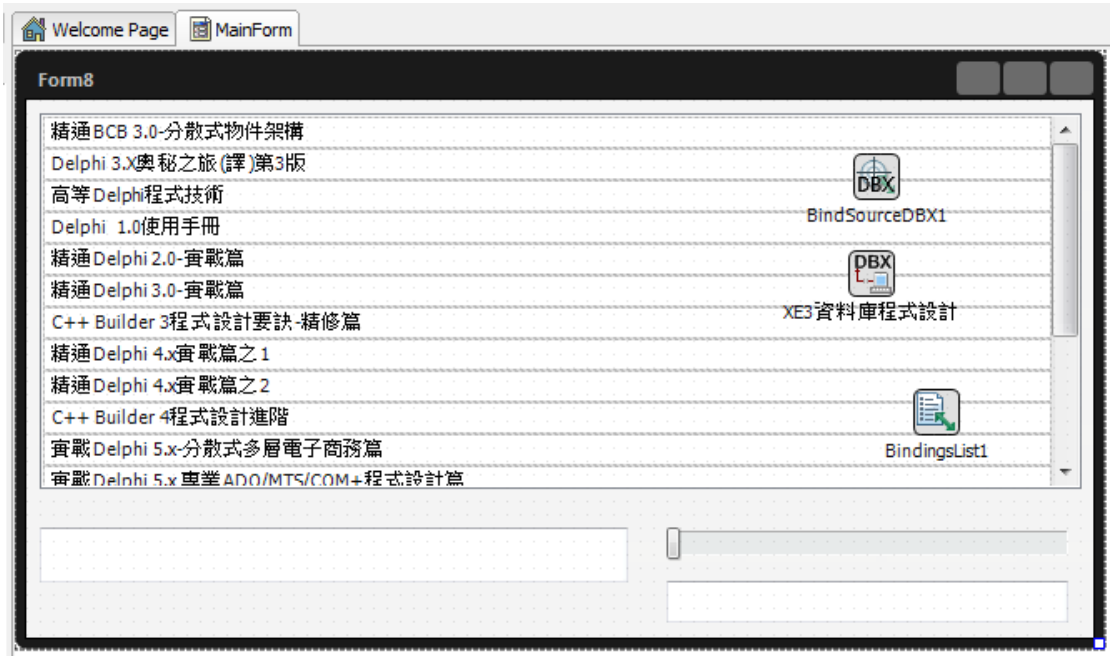
例如如果我們想繫結 TTrackBar 元件和資料來源，那麼我們通常需要把 0 繫結到 TTrackBar 元件的 Min 特性值，資料來源的 RecordCount 繫結到 TTrackBar 元件的 Max 特性值，再把資料來源的 RecNo 繫結到 TTrackBar 元件的 Value 特性值，如此一來當我們捲動 TTrackBar 元件時就可以在資料來源中不同的資料中瀏覽。在上面的說明中有 3 個繫結的工作，我們可以使用 3 個 TBindExpresssion 物件來分別繫結，但我們也可以使用一個 TBindExprItems 物件來撰寫和執行這 3 個繫結工作。

現在就讓我們使用一個範例來說明 TBindExprItems 繫結類別以及拖管繫結運算式的意義。

先在 IDE 中建立一個 FireMonkey Desktop Application 專案，在主表單中拖曳 Data Explorer 中的『XE3 資料庫程式設計』節點到主表單中以建立 TSQLConnectin 元件，再放入 TBindSourceDBX 元件，設定它的特性如下:

| 特性名稱 | 特性值 |
|---|---|
| SQLConnection | XE3 資料庫程式設計 |
| CommandText | Select * from BOOKS |
| CommandType | Dbx.SQL |

再放入一個 TListBox 元件，2 個 TEdit 元件和一個 TTrackBar 元件，最後使用視覺化即時資料繫結設計家繫結 TBindSourceDBX 元件的 BOOKNAME 欄位到 TListBox 的 Text 特性，現在主表單如下所示:




在這個範例中我們將使用繫結類別來完成下列的工作:

1. 點選 TListBox 中的項目時，此項目會自動出現在主表單左下方的 TEdit 元件中

2. 繫結主表單右下方的 TTrackBar 元件和 TListBox 元件，因此當拖曳主表單加下方的 TTrackBar 元件時，TListBox 中的項目也會自動根據 TTrackBar 的位置捲動到相對的項目

3. 當拖曳 TTrackBar 時，TTrackBar 的位置數值會自動出現在主表單右下方的 TEdit 元件中

4. 當輸入數值到主表單右下方的 TEdit 元件中時，也會改變 TTrackBar 的位置

上面的工作看起來很多，但使用繫結運算式卻可以輕易的完成這些工作，而且由於稍後我們會使用拖管繫結運算式，因此會產生連動的效果。現在讓我們一步一步的使用繫結運算式完成上面的 4 個工作。

## 步驟 1-繫結 TListBox 和 TEdit 元件

由於這只需要一個繫結運算式就可以完成，因此我們只需要使用 TBindExpression 繫結物件即可，這個繫結也很簡單，讀者現在應該可以非常容易的完成它。

由於這只需要一個繫結運算式就可以完成，因此我們只需要使用 TBindExpression 繫結物件即可，這個繫結也很簡單，讀者現在應該可以非常容易的完成它，我們只需要把 TListBox 中目前被選擇的項目繫結到 TEdit 的 Text 特性即可，因此我們可以得到下面的繫結運算式:

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TListBox |
| 來源繫結運算式 | ? |
| 控制元件 | TEdit |
| 控制繫結運算式 | Text |

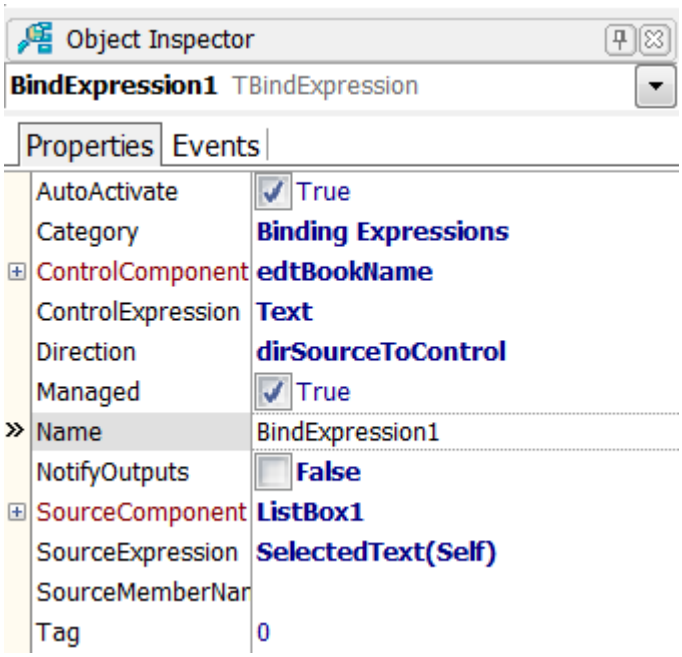


上面唯一要確定的是如何使用繫結運算式取得 TListBox 元件中目前被選擇的項目呢? 這很簡單，因為在繫結引擎中有一個全域函式:SelectedText，它可以從串列元件中取得目前被選擇的項目，因此最後的繫結運算式是:

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TListBox |
| 來源繫結運算式 | SelectedText(Self) |
| 控制元件 | TEdit |
| 控制繫結運算式 | Text |

上面的來源繫結運算式中的 Self 就是來源繫結運算式的執行範圍，它當然就是來源元件 TListBox 了，因此 SelectedText(Self)就等於 SelectedText(ListBox1)，但在繫結運算式中不可以使用 SelectedText(ListBox1)，因為 ListBox1 是 C++Builder 的物件變數名稱，繫結運算式並無法存取 ListBox1，因此必須使用 SelectedText(Self)。

因此現在請在主表單中的 TBindingsList 中新增一個 TBindExpression 元件，並且如下面的物件檢視器一樣設定我們上面的繫結運算式。




接著我們當然需要呼叫 TBindExpression 物件的 Evaluate 方法才能執行上面的繫結運算式，因此請在 TListBox 的 OnChange 事件處理函式中撰寫如下的程式碼:

```
void __fastcall TForm2::ListBox1Change(TObject *Sender)
{
  BindExpression1->Evaluate();
  BindingsList1->Notify(ListBox1, "");
}
```

現在如果您執行此範例應用程式那麼當您點選 TListBox 中的項目時它就會自動出現在主表單左下方的 TEdit 元件中了。

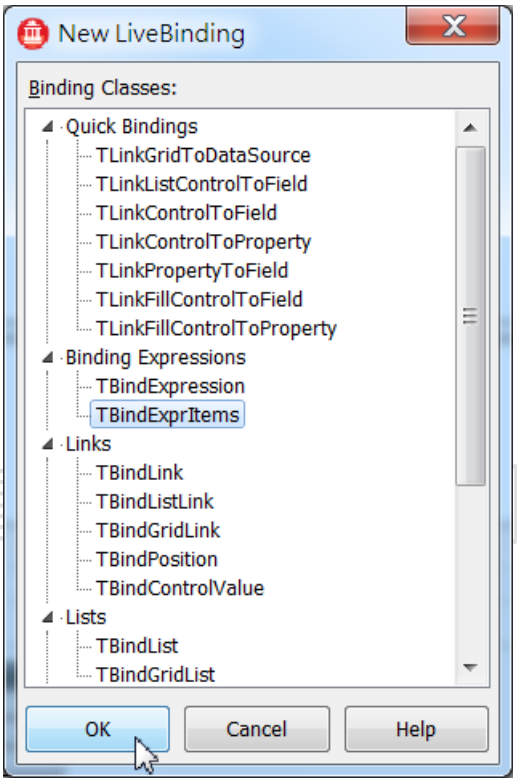
## 步驟 2-繫結 TListBox 和 TTrackBar 元件

要繫結 TListBox 和 TTrackBar 需要數個繫結運算式同時運作，因為我們需要:

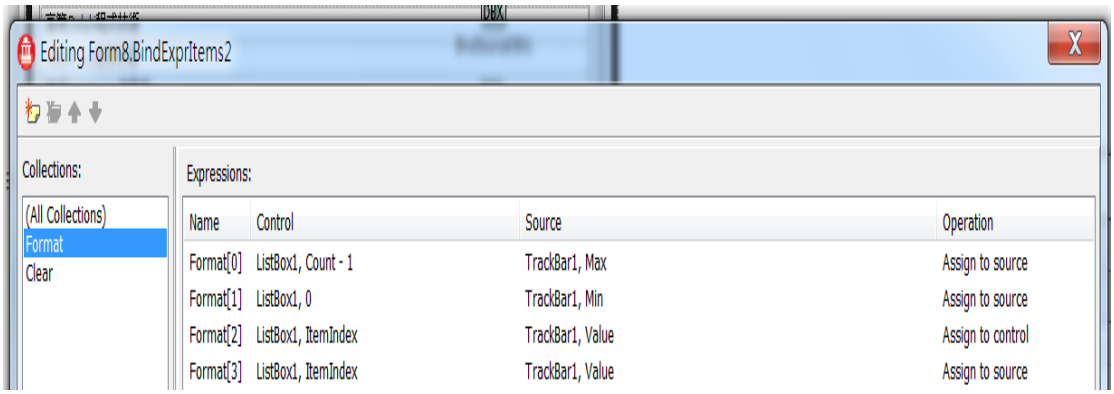
1. 把 TListBox 的 Count - 1 特性值繫結到 TTrackBar 的 Max 特性
2. 把 TListBox 的 0 特性值繫結到 TTrackBar 的 Min 特性
3. 把 TListBox 的 ItemIndex 繫結到 TTrackBar 的 Value 特性
4. 把 TTrackBar 的 Value 特性繫結到 TListBox 的 ItemIndex 繫結

一旦完成了上面的第 3,4 步驟之後 TListBox 和 TTrackBar 就可以連動了，拖曳 TTrackBar 可以改變 TListBox 中目前的選擇項目，而改變 TListBox 中目前的選擇項目則可以連動改變 TTrackBar 的拖曳點(即 Value 特性值)。

由於需要同時繫結上述的工作，因此我們可以建立一個 TBindExprItems 物件並且在它的 Format 特性中同時輸入上面的繫結運算式。因此請在主表單的 TBindingsList 元件中建立一個 TBindExprItems 物件，如下所示:




接著雙擊 TBindingsList 元件編輯器中新建立的 TBindExprItems 物件開啟 TBindExprItems 物件的元件編輯器，點選左上方的『Add Expression』按鈕以新增繫結運算式物件，請在其中建立 4 個繫結運算式，然後分別撰寫如下的繫結運算式以繫結 TListBox 和 TTrackBar:




下面的表格說明了上面 4 個繫結運算式的設定:

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Max |
| 控制元件 | ListBox1 |
| 控制繫結運算式 | Count – 1 |
| 方向 | Assign to Source，即把控制繫結運算式的執行結果指定給來源元件 |
| 繫結運算式意義 | 設定 TrackBar1 的最大值為 ListBox1 中的項目總數 |



| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Min |
| 控制元件 | TListBox |
| 控制繫結運算式 | 0 |
| 方向 | Assign to Source，即把控制繫結運算式的執行結果指定給來源元件 |
| 繫結運算式意義 | 設定 TrackBar1 的最小值為 0 |

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Value |

| 控制元件 | ListBox1 |
|---|---|
| 控制繫結運算式 | ItemIndex |
| 方向 | Assign to Control，即把來源繫結運算式的執行結果指定給控制元件 |
| 繫結運算式意義 | 設定 TrackBar1 的 Value 特性值指定給 ListBox1 的 ItemIndex 特性，這代表拖曳 TrackBar1 就會改變 ListBox1 中目前被選擇的項目 |

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Value |
| 控制元件 | ListBox1 |
| 控制繫結運算式 | ItemIndex |
| 方向 | Assign to Source，即把控制繫結運算式的執行結果指定給來源元件 |
| 繫結運算式意義 | 設定 ListBox1 的 ItemIndex 特性值指定給 TrackBar1 的 Value 特性，這代表改變 ListBox1 中目前被選擇的項目就會改變 TrackBar1 的拖曳位置 |



完成了上面的繫結運算式之後同樣的要藉由 Evaluate 方法執行，不過由於在上面有數個繫結運算式，因此我們可以呼叫 TBindingsList 的 Notify 方法來通知繫結引擎執行繫結運算式。TBindingsList 的 Notify 方法接受 2 個參數，第 1 個是改變繫結條件的物件，第 2 個參數則是改變繫結條件的特性名稱，它的原型如下:

```
    void __fastcall Notify(System::TObject* const AObject, const
System::UnicodeString AProperty);
```

由於在上面的 4 個繫結運算式中會有 2 個物件可能改變繫結條件，那就是 ListBox1 和 TrackBar1，而可能改變繫結條件的特性則有 TracBar1 的 Min，Max 和 Value 以及 ListBox1 的 ItemIndex 特性，因此我們可以傳遞空字串給 Notify 方法的第 2 個參數代表第 1 個參數物件所有的特性都改變了因此要求繫結引擎重新執行所有和第 1 個參數物件相關的繫結運算式。

因此我們需要在 TrackBar1 的 OnChange 事件處理函式中撰寫如下的程式碼，代表 TrackBar1 要求繫結引擎重新執行所有和 TrackBar1 相關的繫結運算式。

```
void __fastcall TForm2::TrackBar1Change(TObject *Sender)
{
    BindingsList1->Notify(TrackBar1, "");
}
```

BindingsList1.Notify(TrackBar1, '')代表 TrackBar1 要求繫結引擎重新計算和執行所有繫結到 TrackBar1 的任何特性相關的繫結運算式，在執行了此行程式碼之後，前面討論的所有和 TrackBar1 相關的繫結運算式都會被重新執行。

另外我們也需要要求繫結引擎重新計算和執行所有繫結到 ListBox1 的任何特性相關的繫結運算式，因此請在 ListBox1 的 OnChange 事件處理函式中修改原先的程式碼如下:

現在我們也只需要呼叫 BindingsList1.Notify(ListBox1, '') 即可讓所有和 ListBox1 相關的繫結運算式都能夠重新執行，因此我們不再需要單獨呼叫 BindExpression1.Evaluate 方法了:

```
void __fastcall TForm2::ListBox1Change(TObject *Sender)
{
//  BindExpression1->Evaluate();
  BindingsList1->Notify(ListBox1, "");
}
```

現在如果執行範例程式的話請讀者注意在拖曳 TrackBar1 時除了 ListBox1 中目前被選擇的項目也會改變之外，主表單左下方的 TEdit 元件的內容也會改變，為什麼？這當然就是因為 TrackBar1 繫結了 ListBox1，而 ListBox1 又繫結了左下方的 TEdit 元件，而這 2 個繫結運物件(TBindExpression 和 TBindExprItems 物件)都是拖管繫結物件，因此當上面 ListBox1Change 事件處理函式藉由 Notify 方法通知繫結引擎 ListBox1 的繫結條件改變時，繫結引擎會自動重新執行所有和 ListBox1 相關的繫結運算式，因此 BindExpression1 就會被繫結引擎重新執行，因此左下方的 TEdit 元件就會重新顯示 ListBox1 中目前被選擇的項目內容，這種會被繫結引擎自動根據繫結條件變化而重新執行繫結運算式的能力就是拖管繫結運算式的特性，未拖管繫結運算式則沒有這種特性。

## 步驟 3-繫結 TTrackBar 和 TEdit 元件

要在拖曳 TTrackBar 時，把 TTrackBar 的位置數值會自動顯示在主表單右下方的 TEdit 元件中就非常的容易了，我們只需要使用下面的繫結運算式即可:

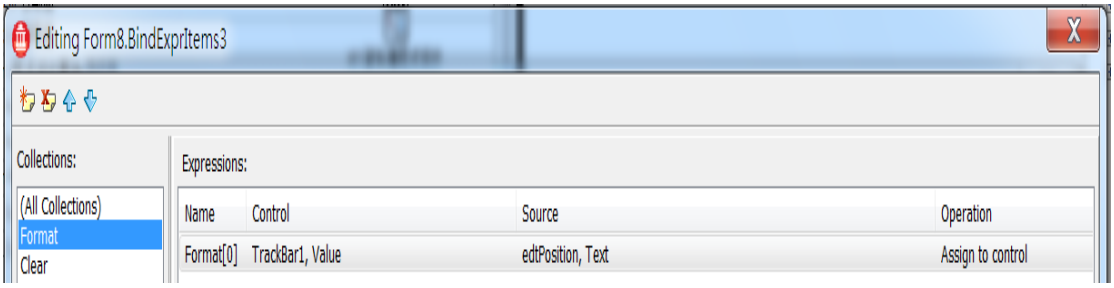

下面的表格說明了要完成這個工作我們只需要把 TrackBar1 的 Value 特性值繫結到主表單右下方的 TEdit 元件 edtPosition 的 Text 特性即可:

| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Value |
| 控制元件 | edtPosition |
| 控制繫結運算式 | Text |
| 方向 | Assign to Control，即把來源繫結運算式的執行結果指定給控制元件 |

## 步驟 4-繫結 TEdit 和 TTrackBar 元件



最後一個繫結工作也正好是步驟 3 的反方向，我們只需把 edtPosition 的 Text 特性繫結到 TrackBar1 的 Value 特性即可，因此我們只需使用下面的繫結運算式：

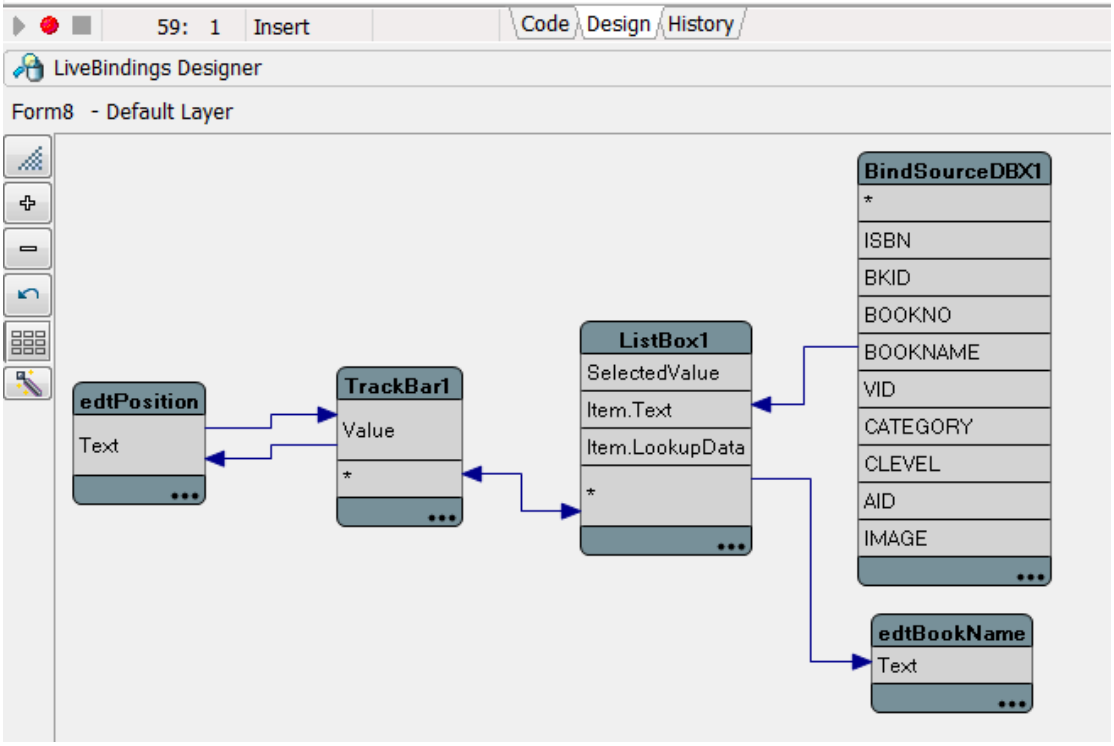



下面的表格說明了要完成這個工作我們只需要把 edtPosition 的 Text 特性值繫結到 TrackBar1 的 Value 特性即可:

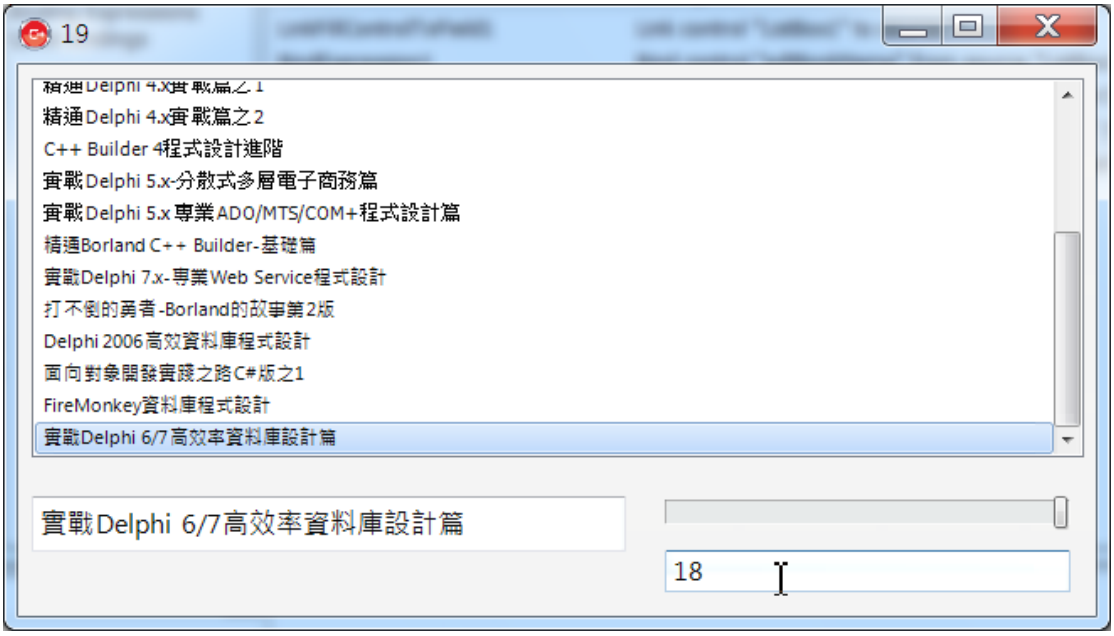
| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | TrackBar1 |
| 來源繫結運算式 | Value |
| 控制元件 | edtPosition |
| 控制繫結運算式 | Text |

| | |
|---|---|
| 方向 | Assign to Source，即把控制繫結運算式的執行結果指定給來源元件 |

完成了這些繫結運算式之後如果讀者現在開啟視覺化即時資料繫結設計家，那麼就會看到類似如下的結果，上面的繫結運算式雖然無法使用即時資料繫結設計家來設計和撰寫(因為即時資料繫結設計家主要是使用快速繫結類別物件)，但這些由開發人員撰寫的繫結運算式仍然能夠展現在即時資料繫結設計家中。



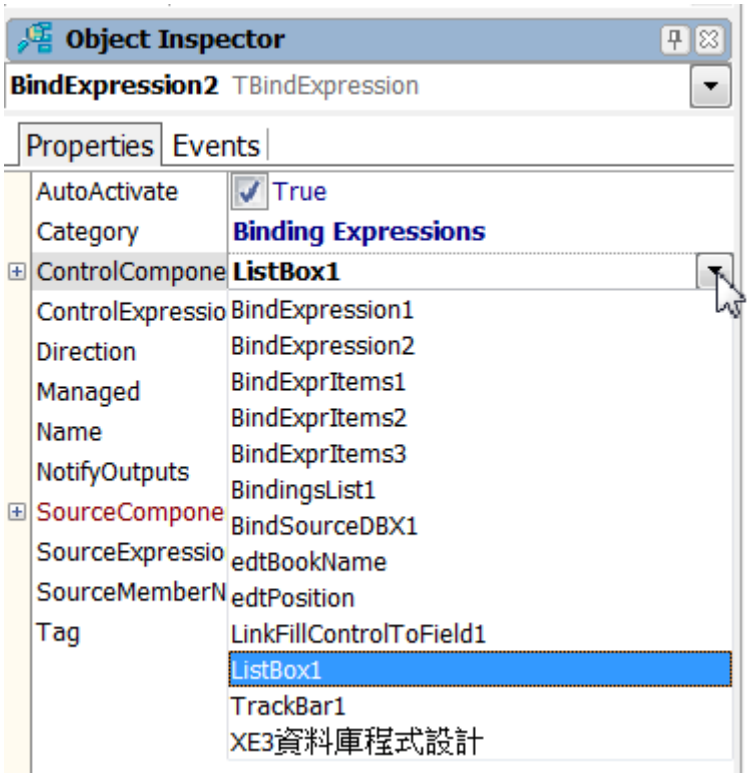

現在請編譯和執行範例 FireMonkey 應用程式，如果拖曳 TTrackBar，點選 TListBox 或是在主表單左下方的 TEdit 中輸入數值都會發現主表單中所有的元件都可連動顯示，如下圖所示:




在離開本小節之前再讓我們撰寫一個繫結運算式讓讀者瞭解如何在繫結運算式中藉由執行範圍存取其他的元件。假設現在我們希望在執行此範例 FireMonkey 應用程式時，當拖曳 TTrackBar 時，我們也希望 ListBox1 中目前被選擇的項目也能夠出現在主表單左上方的表頭中，那麼我們應該如何撰寫這個繫結運算式呢?

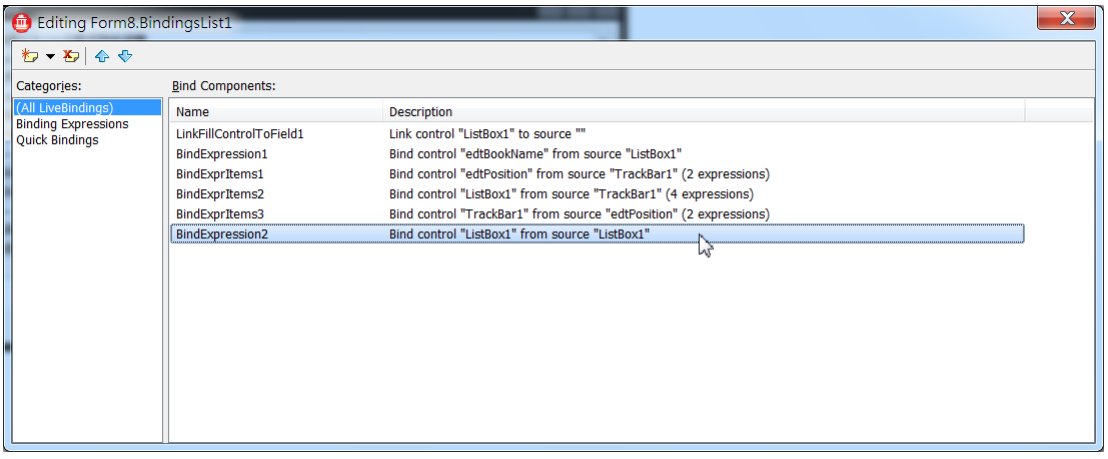


OK，先讓我們找出來源元件和控制元件，它們分別應該是 ListBox1 和主表單，但如果我們在 TBindingsList 中建立一個 TBindExpression 繫結物件，然後在物件檢視器中點選 ControlComponent 特性，在它的下拉盒中卻無法找到主表單物件，如下所示，那麼我們如何把主表單物件設定為控制元件呢?

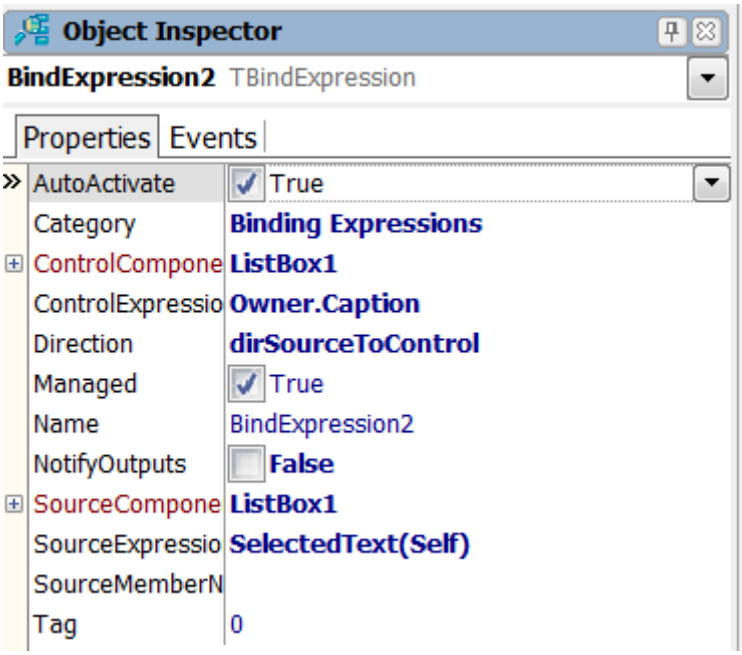



答案是不能直接設定主表單為控制元件，因為目前的即時資料繫結只適用於主表單中的元件，那麼現在我們要如何解決這個繫結工作呢? 很簡單，我們只要把控制元件設定為 ListBox1 的 Owner 不就可以了嗎? 因為 ListBox1 的 Owner 就是主表單物件。因此讓我們先在 TBindingsList 中建立一個 TBindExpression 繫結物件，如下所示:






然後在物件檢視器中設定如下的繫結運算式:

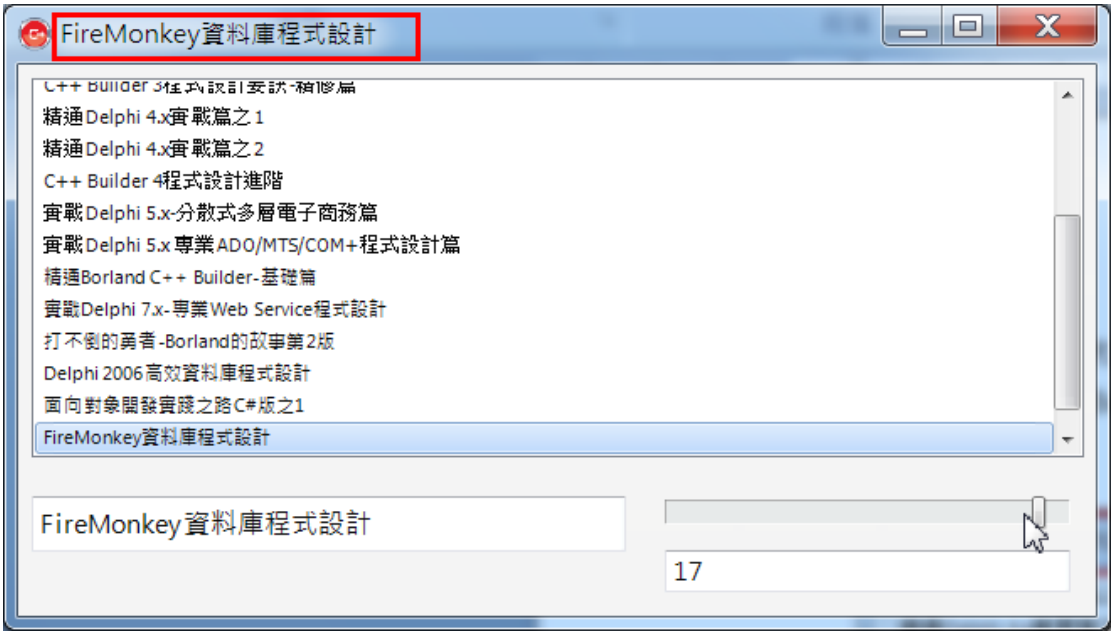



| 繫結運算式 | 設定值 |
|---|---|
| 來源元件 | ListBox1 |
| 來源繫結運算式 | SelectedText(Self) |
| 控制元件 | ListBox1 |
| 控制繫結運算式 | **Owner**.Caption |
| 方向 | Assign to Control，即把控制繫結運算式的執行結果指定給來源元件 |



在上面的繫結運算式中我們把來源和控制元件都設定為 ListBox1，但在控制繫結運算式中先使用關鍵字 Owner 來取得 ListBox1 的 Owner(即主表單)，再繫結到它的 Text 特性即可。

現在再次編譯和執行範例 FireMonkey 應用程式，拖曳主表單中的 TrackBar1 時 ListBox1 中目前被選擇的項目也會自動出現在主表單左上方的表頭中了。




這個範例說明了在繫結運算式中開發人員可以使用 Owner 關鍵字來存取目前執行範圍物件的父代物件，這是很有用的技巧之一。



## 9-4 TBindingsList 提供的可呼叫方法

除了函數式之外，TBindingsList 也提供的一些方法允許開發人員使用在函數式中，例如在前面我們已經看過許多次的 ToStr，SelectedText(Self)等。這些可使用於函數式中的方法可以藉由在物件檢視器中點選 TBindingsList 的 Methods 特性找到，在前面的章節中已經說明過。

這些方法在即時資料繫結框架中是使用如下的程式碼註冊並且提供給開發人員使用:

```
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;
....
```

當然，開發人員也可以實作自己的方法，向即時資料繫結框架註冊之後就可以使用在繫結運算式中。下面的表格說明了這些方法的應用:

| 方法 | 說明 |
|---|---|
| CheckedState(Self) | 繫結布林值和 TCheckedBox 的方法，例如開發人員可以繫結資料表布林值欄位和 TCheckedBox |
| Format | 提供在函數式中呼叫 RTL 的 Format 方法 |
| LowerCase | 把字串轉換為小寫型態 |
| SelectedItem | 存取目前繫結的元件中的項目，例如存取目前 TListBox 或是 TGridBox 中被選擇的項目 |
| SelectedText | 存取目前繫結的元件中被選擇的項目的 Text 特性值，例如存取目前 TListBox 或是 TGridBox 中被選擇的項目的 Text 特性值 |
| ToStr | 把函數式中的運算結果轉換為字串型態，例如把函數式中的整數或是浮點數轉換為字串 |
| ToVariant | 把函數式中的運算結果轉換為 Variant 型態 |
| UpperCase | 把字串轉換為大寫型態 |
| IfThen | 接受 3 個參數，第 1 個是布林值型態的函式參數，如果第 1 個參數是 True 的話就回傳第 2 個參數值，如果第 1 個參數是 False 的話就回傳第 3 個參數值。 |
| IfAny | 接受最多 100 個布林值型態的函式參數，只要有任何的函式參數的結果值是 True 的話，就回傳 True |
| IfAll | 接受最多 100 個布林值型態的函式參數，只要有任何的函式參數的結果值是 False 的話，就回傳 False，只有所有的函式參數都是 True 才會回傳 True |
| DButils_Activerecord | 取得第 1 個參數(TDataSet 型態)的目前記錄值 |
| DBUtils_ValidRecNo | 設定 1 個參數(TDataSet 型態)的目前記錄值為第 2 個參數的數值 |
| FormatDateTime | 格式化 TDateTime 型態的參數並回傳結果 |
| Lookup | 根據執行範例以(鍵值欄位名稱, 鍵值欄位值, 回傳欄位名稱) 的參數資訊查詢回傳欄位值 |
| Math_Max | 接受 2 個整數型態的參數，並且回傳擁有最大整數數值的參 |

| | |
|---|---|
| | 數 |
| Math_Min | 接受 2 個整數型態的參數，並且回傳擁有最小整數數值的參數 |
| Round | 四捨五入參數值並回傳結果值 |
| SelectedLookupValue | 從串列型態的元件中回傳查詢的數值 |
| SelectedValue | 從串列型態的元件中回傳目前版選擇的數值 |
| StrToDateTime | 轉換字串為 TDateTime 型態的數值 |
| ToNotifyEvent | 呼叫事件處理函式 |

TOKYO 的繫結引擎的全域方法比起 XE2 的繫結引擎的全域方法大幅增加了許多，讓開發人員可以在繫結運算式中進行更複雜的控制，在本書的許多內容中都會看到我們在繫結運算式中使用這些繫結引擎的全域方法。

## 9-5 繫結編輯器，觀察元和繫結範例元件

在繫結框架中還有幾個觀念是讀者必須瞭解的，它們是繫結編輯器，觀察元和執行範圍元件。



繫結編輯器(Editors)定義了如何修改控制的抽象介面，實作繫結編輯器的類別可以和許多控制項結合在一起以提供繫結運算式執行時修改控制項的內容，例如繫結編輯器可以和 TEdit，TListBox 結合。在繫結類別中的 TBindList 就使用了繫結編輯器來修改串列控制項的內容。繫結框架為許多的 FMX 和 VCL 控制項都實作了繫結編輯器。

觀察元(Observers)定義了觀察一個控制項是否被修改的抽象介面，實作觀察元的類別可以和控制項結合在一起以提供使用者互動的觀察機制，例如 TBindLink 繫結類別便使用了觀察元來觀察它繫結的 TEdit 元件是否被使用者修改了。繫結框架也為 FMX 和 VCL 控制項實作了許多觀察元。

下面的表格說明了使用繫結編輯器和觀察元的繫結類別:

| 元件 | 使用繫結編輯器 | 使用觀察元 |
|---|---|---|
| TBindExpression, TBindExprItems | | |
| TBindPosition | | X |
| TBindList, TBindGridList | X | |
| TBindLink | | X |
| TBindGridLink, TBindListLink | X | X |

在前面小節介紹繫結框架時討論過了繫結執行範圍的觀念，不過在前面的小節中是使用程式碼來取得執行範圍，C++Builder 已經把一些通用的執行範圍封裝為不同的元件讓開發人員可以直接使用，在前面的範例中我們已經使用了許多的執行範圍元件，例如 TBindScopeDB 和 TBindScopeDBX 等，這兩個元件都是繫結資料來源為繫結執行範圍，下面的表格說明了 TOKYO 中所有的執行範圍元件:



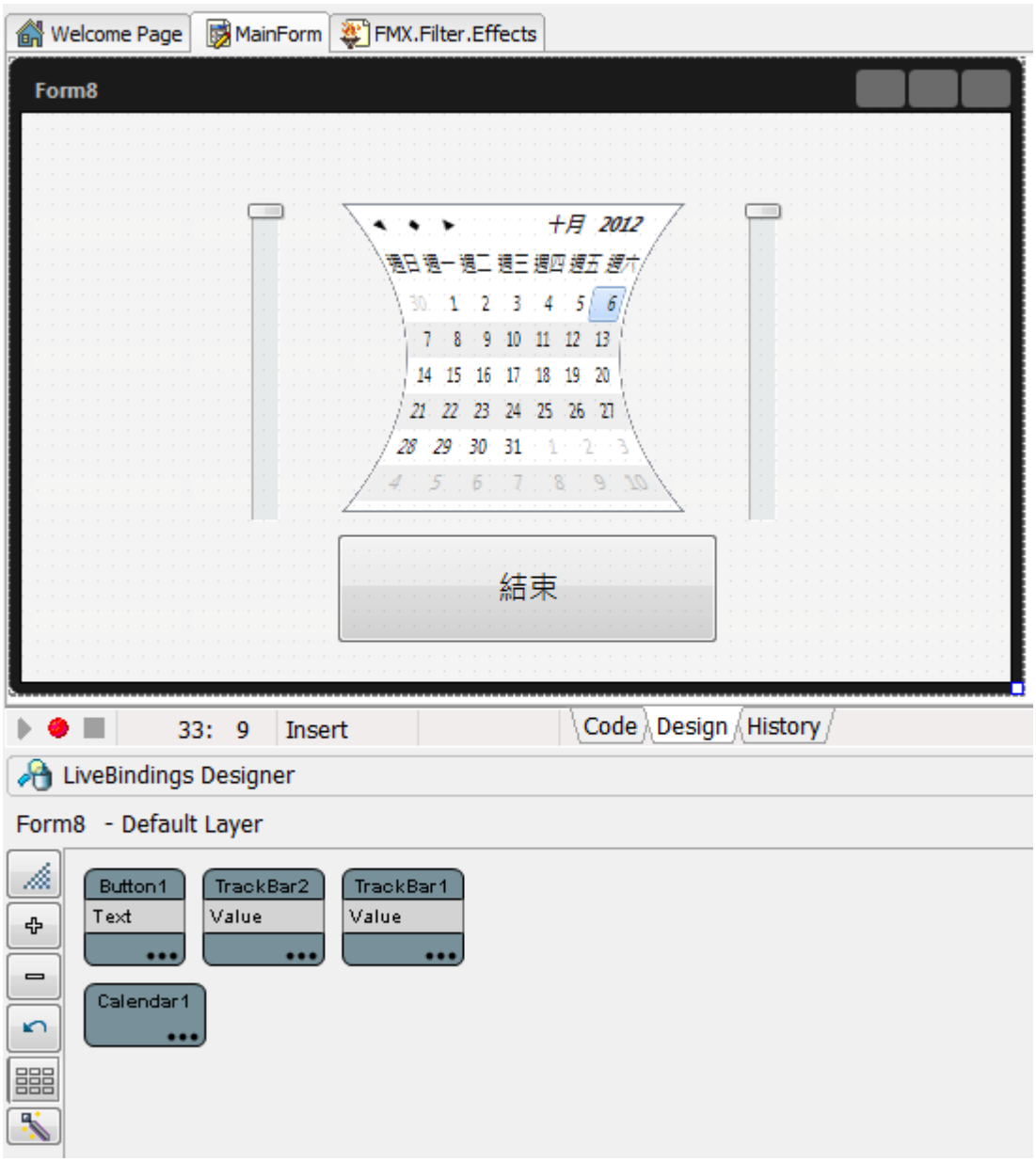
| 繫結範圍元件 | 說明 |
|---|---|
|  TBindScope | 繫結客製化類似物件為執行範圍 |
|  TBindScopeDB | 繫結通用資料來源為執行範圍 |
|  TPrototypeBindSource | 繫結隨機產生資料的虛擬資料表為執行範圍 |
|  TBindSourceDBX | 繫結 dbxExpress 元件為執行範圍，例如 TClientDataSet |
|  TAdapterBindSource | 做為繫結不同繫結範圍元件的 Adapter，使用 TAdapterBindSource 元件開發人員可以在不同的繫結範圍元件間切換 |

在下一章討論客製化類別和即時資料繫結時會討論如何使用 TBindScope 元件和 TAdapterBindSource 件。

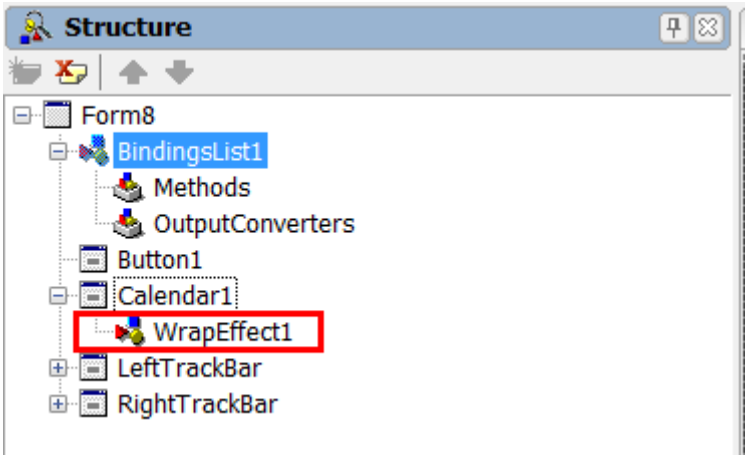
# 9-6 使用即時資料繫結設定

當開發人員使用視覺化即時資料繫結設計家進行繫結設計時，並不是所有的 C++Builder 元件都會出現在設計家中，因為在內定上 C++Builder 整合發展環境只設定了開發人員最常使用的元件才會出現在設計家中，但開發人員可以改變這個內定的設定讓任何開發人員想使用的元件都能夠出現在設計家中。

在 C++Builder 整合發展環境中建立一個 FireMonkey Desktop Application 應用程式，在主表單中放入 2 個 TTrackBar 元件，一個 TCalendar 元件，一個內嵌在 TCalendar 元件中的 TWrapEffect 元件以及一個 TButton 元件，開啟視覺化即時資料繫結設計家，此時主表單和視覺化即時資料繫結設計家如下所示:



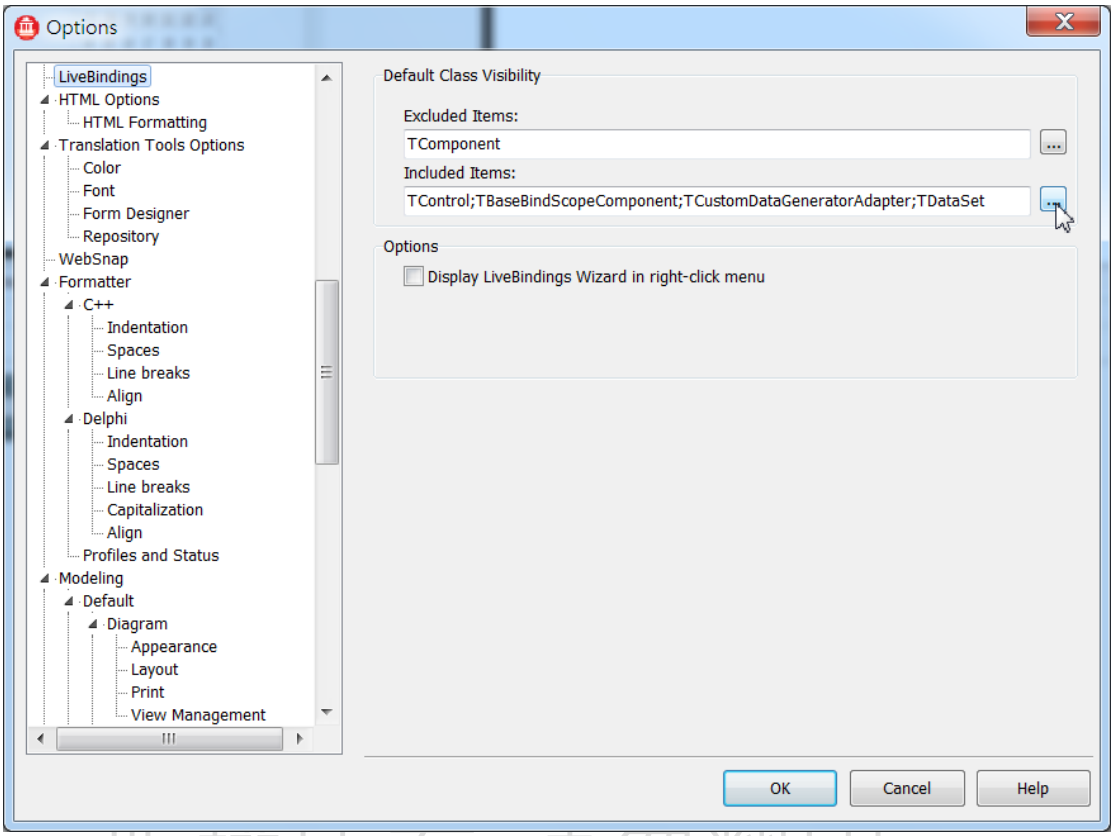

由於 TWrapEffect 元件是內嵌在 TCalendar 元件中，因此我們可以在整合發展環境左上方的樹狀架構視窗中看到如下的架構:




但請讀者觀察上面的視覺化即時資料繫結設計家，我們會發現 TWrapEffect 元件並沒有出現在設計家中，為什麼? 這就是因為 TEffect 相關的類別在內定被設定為不出現在視覺化即時資料繫結設計家。但現在我們希望能夠繫結主表單中的 2 個 TTrackBar 元件到 TWrapEffect 元件以便能夠拖曳 TTrackBar 元件來改變 TWrapEffect 元件對於 TCalendar 元件的影響程度，因此我們需要在即時資料繫結設計家中繫結 TTrackBar 和 TWrapEffect，因此我們需要 TWrapEffect 能夠出現在設計家中。

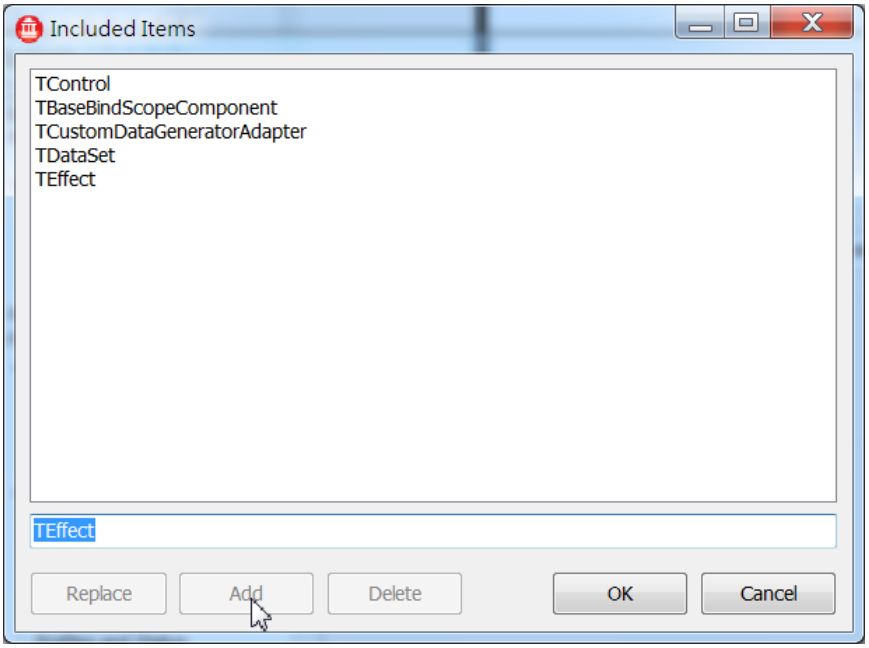


要讓 TWrapEffect 出現在設計家中很簡單，我們只需要改變視覺化即時資料繫結設計家的內定設定即可。請點選 Tools|Options 功能表開啟 Options 對話盒，在 LiveBindings 選項中點選 Included Items 控制項右方的選項按鈕，如下所示:

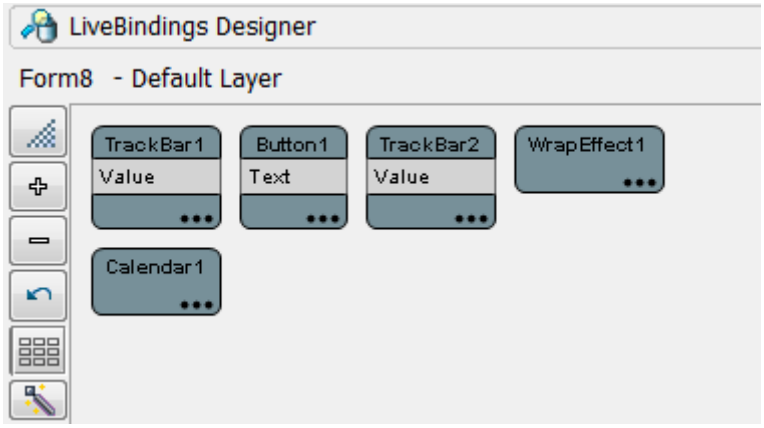





在 Included Items 對話盒中下方的 Edit 控制項中輸入 TEffect 類別名稱把 TEffect 類別和所有的衍生類別都加入到可顯示於視覺化即時資料繫結設計家中的類別，然後點選下方的 Add 按鈕，再點選 OK 按鈕。



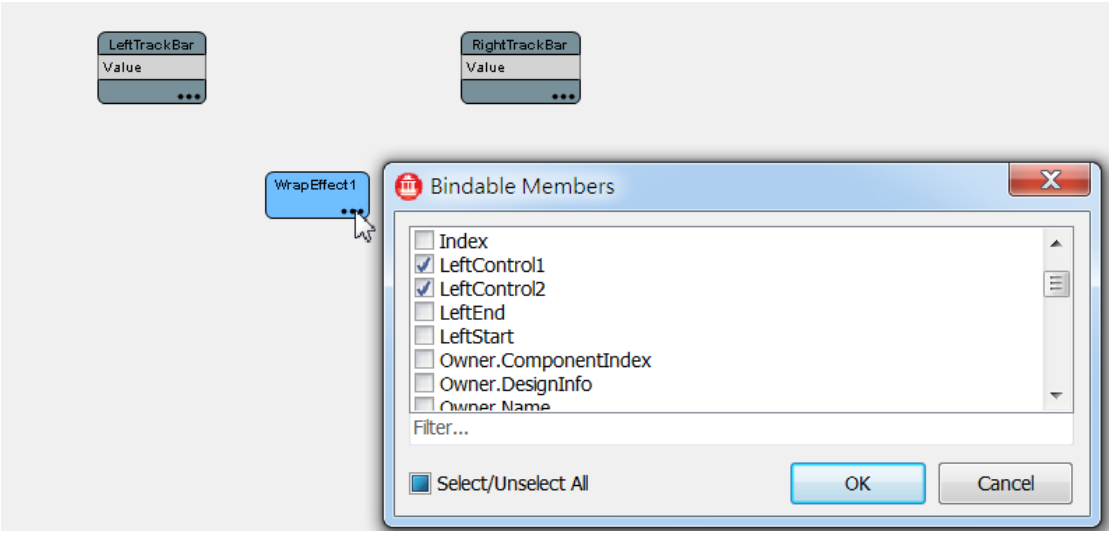
回到視覺化即時資料繫結設計家中此時就可以看到類似下面的結果，現在 TWrapEffect 元件已經出現在視覺化即時資料繫結設計家中了:



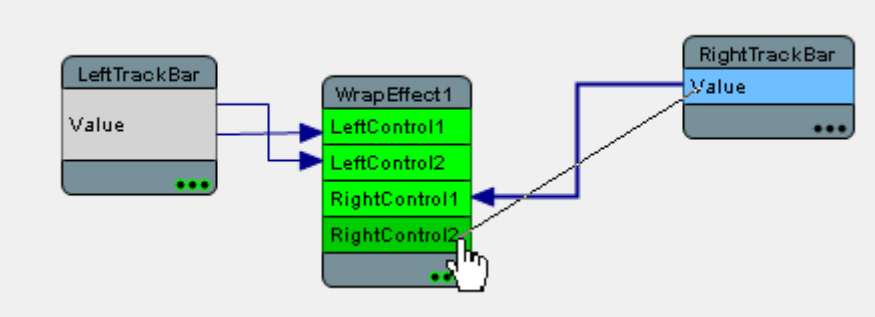

現在我們希望繫結主表單中的 2 個 TTrackBar 元件和 TWrapEffect 元件，當拖曳主表單中左右 2 方的 TTrackBar 元件時能夠改變 TWrapEffect 元件的特性值，如此一來就可以影響主表單中 TCalendar 元件的纏繞效果。

請點選視覺化即時資料繫結設計家中 WrapEffect1 實體右下方的『…』，於 Bindable Members 對話盒中勾選 LeftControl1，LeftControl2，RightControl1 和 RightControl2 特性以把這 4 個特性顯示於 WrapEffect1 實體中，如下所示:






接著拖曳 LeftTrackBar 的 Value 特性到 WrapEffect1 實體的 LeftControl1，LeftControl2 特性，再拖曳 RightTrackBar 的 Value 特性到 WrapEffect1 實體的 RightControl1 和 RightControl2 特性，以繫結左右 2 個 TTrackBar 元件到 WrapEffect1 元件的 4 個特性，如下所示:




現在執行此範例 FireMonkey 應用程式並且拖曳主表單上的 2 個 TTrackBar 元件就可以看到 TWrapEffect 元件對於 TCalendar 元件的影響了，如下所示:


# 9-7 TBindScope 元件

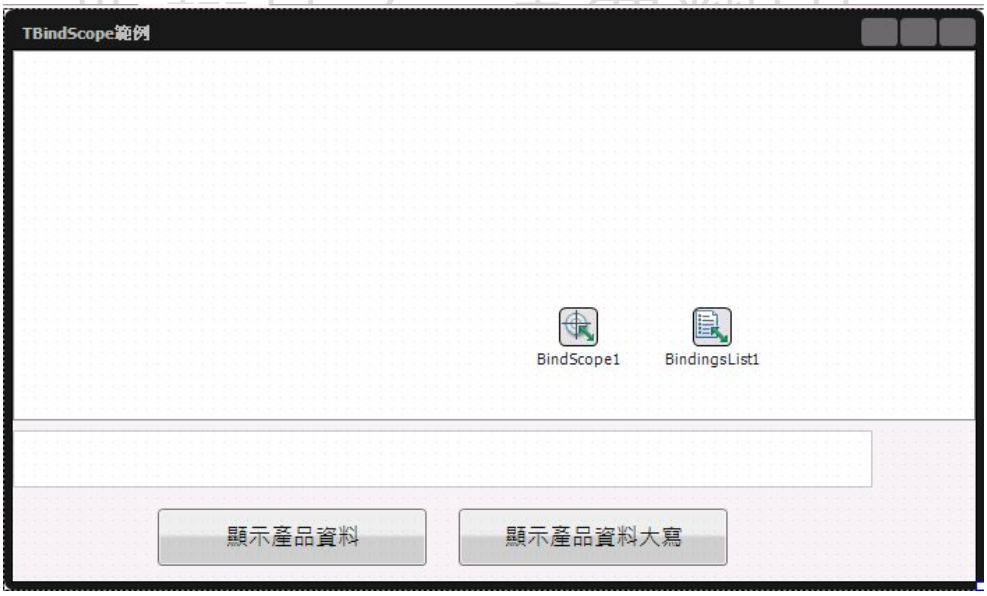
TBindScope 元件的功能是讓不是元件型態的物件也能夠藉由即時繫結技術和 FireMonkey 或是 VCL 的元件繫結在一起，例如 TBindScope 元件允許串列物件型態 (TList，TObjectList)，或是 JSON 物件串列物件和 FireMonkey 或是 VCL 的元件繫結。

TBindScope 物件提供了讓開發人員使用函數式動態繫結串列物件和圖形使用者介面元件的能力，這不但提供了動態繫結的能力，也可以讓程式碼大為簡化。TBindScope 元件提供了 DataObject 特性，開發人員只需要把資料指定給這個特性，再使用函數式即能夠繫結元件。現在讓我們使用一個範例來說明如何使用 TBindScope 的動態繫結能力。

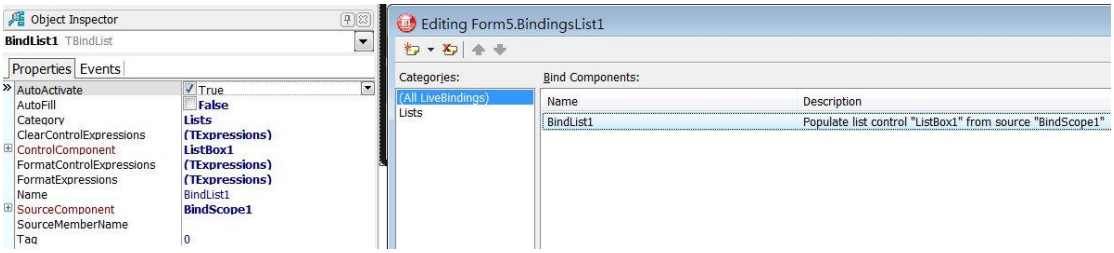
我們在這個範例中將建立一個 TJSONArray 物件，並且在其中建立一些 TJSONString，如下所示:

```
void TForm3::CreateJSONArray()
{
  if (FJSONArray == NULL)
  {
    FJSONArray = new TJSONArray();
    FJSONArray->AddElement(new TJSONString("C++Builder XE2"));
    FJSONArray->AddElement(new TJSONString("C++Builder XE2"));
    FJSONArray->AddElement(new TJSONString("C++Builder Prism XE2"));
    FJSONArray->AddElement(new TJSONString("RadPHP XE2"));
  }
}
```

接著在 C++Builder IDE 中建立一個 FireMonkey HD 應用程式專案，在主表單中放入 TBindingsList，TBindScope，TListBox，TEdit 和兩個 TButton 元件，如下所示:




接著在 TBindingsList 元件中建立一個 TBindList 物件，設定 TBindList 物件的 SourceComponent 特性值為 BindScope1，即主表單中的 TBindList 元件，再設定 ControlComponent 特性值為 ListBox1，即主表單中的 TListBox 元件，如下所示:




設定 TBindList 物件繫結 TBindScope 和 TListBox 的原因是稍後在程式碼中我們將再繫結 TJSONArray 物件和 TBindScope，如此一來 TJSONArray 物件中所有的資料就可顯示在 TListBox 中了。

現在讓我們實作主表單中『顯示人員資料』按鈕的功能，首先在主表單的 OnCreate 事件處理函式中撰寫如下的程式碼:

```
void __fastcall TForm3::FormCreate(TObject *Sender)
{
  CreateJSONArray();
}
```

建立好了 TJSONArray 物件之後，我們就可以在『顯示產品資料』按鈕的 OnClick 事件處理函式中撰寫如下的程式碼:



```
001    void __fastcall TForm3::Button3Click(TObject *Sender)
002    {
003      String sExpression;
004      TVarRec args[1] = {"Current."};
005
006      CreateJSONArray();
007      sExpression = Format("'JSON 字串: ' + %0:sToString() ", args, 0);
008      if (BindList1->FormatExpressions->Count == 0)
009        BindList1->FormatExpressions->Add();
010      BindList1->FormatExpressions->operator
[](0)->SourceExpression = sExpression;
011      BindList1->FormatExpressions->operator
[](0)->ControlExpression = "Text";
012      BindScope1->DataObject = FJSONArray;
013      BindList1->FillList();
014    }
```

在 008 行首先判斷 TBindList 物件中是否已經建立了函數式物件，如果沒有的話就在 009 行建立一個 TExpressionItem 物件，現在先讓我們解釋第 012 行之後再解釋 005 和 007 行讀者會比較容易瞭解。

在 012 行設定 TBindScope 的 DataObject 特性值為 TJSONArray 物件，這代表在以前 TBindingsList 物件中定義的 TBindList 物件是讓 TJSONArray 物件中的資料和 TListBox 物件繫結在一起，因此在在 007 行設定 TExpressionItem 物件的 SourceExpression 特性值為:

```
"'JSON 字串: ' + %0:sToString() "
```

在 007 行執行完成之後，上面的字串就會轉換為真正的函數式:

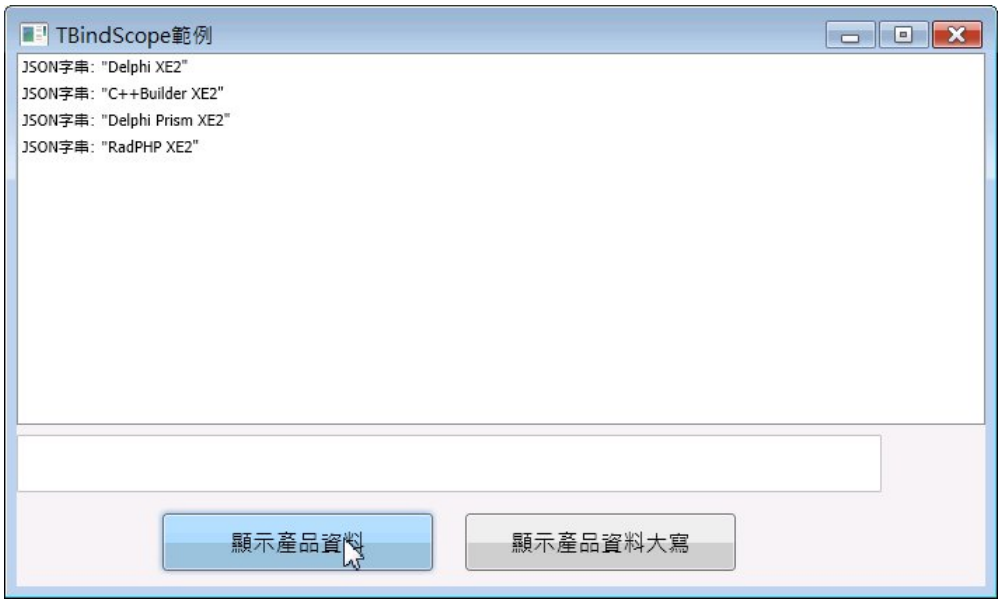
```
"'JSON 字串: ' + Current.ToString() "
```

上面的"Current"代表在 TJSONArray 物件中正要顯示的 TJSONString 物件。

再看 011 行的函數式為什麼可以使用'Text'? 這是因為 TListBox 中的物件其實是 TListBoxItem，因此 Control 目的物件其實就是 TListBox 中的 TListBoxItem 物件，簡單的說最終藉由 TBindScope 和 TBindingsList 中的 TBindList 繫結的雙方其實是 TPerson 和 TListBoxItem，而 Text 正是 TListBoxItem 物件的特性，因此 011 行才能夠使用'Text'這個函數式。



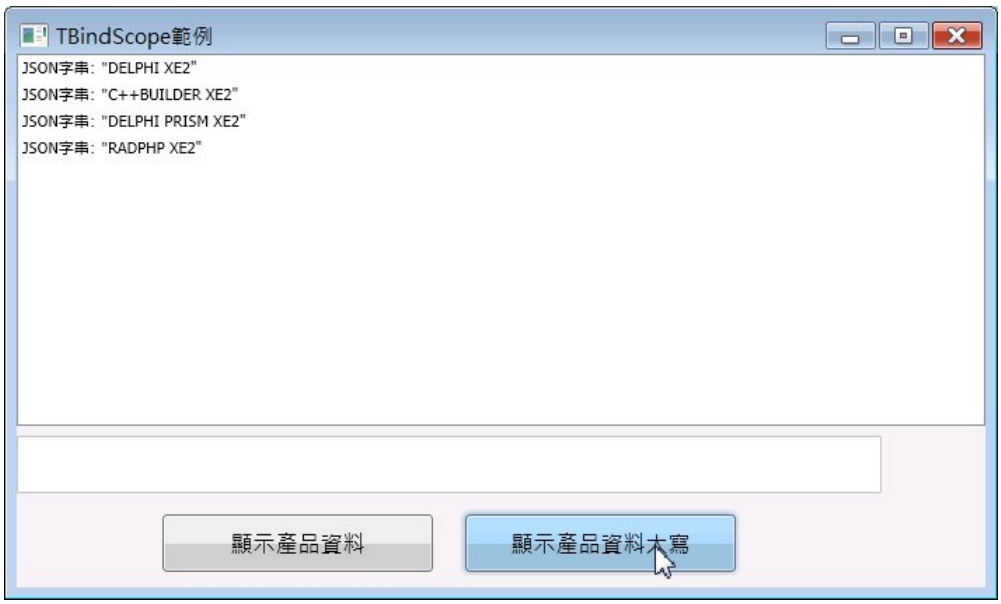
現在如果我們執行這個範例應用程式，點選『顯示人員資料』按鈕就可以看到類似如下的結果，果然每一個 TJSONArray 物件中的 TJSONString 物件成功的顯示在 TListBox 中了。



最後再讓我們在範例應用程式中加入一個『顯示產品資料大寫』按鈕，並且實作它的 OnClick 事件處理函式如下:

```
001    void __fastcall TForm3::Button4Click(TObject *Sender)
002    {
003      String sExpression;
004      TVarRec args[1] = {"Current."};
005
006      CreateJSONArray();
007      sExpression = Format("'JSON 字串: ' + UpperCase(%0:sToString())
", args, 0);
008      if (BindList1->FormatExpressions->Count == 0)
009        BindList1->FormatExpressions->Add();
010      BindList1->FormatExpressions->operator
[](0)->SourceExpression = sExpression;
011      BindList1->FormatExpressions->operator
[](0)->ControlExpression = "Text";
012      BindScope1->DataObject = FJSONArray;
013      BindList1->FillList();
014    }
```

在這個事件處理函式我們希望所有顯示的產品資訊都以大寫形式顯示，因此我們在 007 行的函數式中呼叫 TBindsList 提供的 UpperCase 方法把 TJSONArray 中每一個 TJSONString 自動轉換為大寫的形式，下面是執行範例應用程式的結果:



TBindScope 提供了程式碼和元件之間動態繫結的能力，讓開發人員可以藉由程式碼和函數式開發出更具動態能力的圖形使用者介面應用程式。

## 9-8 結論

本章說明了即時資料繫結概念，讓讀者瞭解在撰寫繫結運算式時，只要瞭解了來源執行範圍，控制執行範圍，以及在執行範圍中能夠存取的元件，方法和特性之後，撰寫繫結運算式就非常的簡單了。

掌握即時資料繫結概念之後本章也使用 C++Builder 的繫結元件來印證即時資料繫結概念，並且介紹了 TOKYO 的時資料繫結類別，也使用了一些時資料繫結類別開發了一些範例讓讀者瞭解如何使用資料繫結類別

最後本章說明了如何設定視覺化即時資料繫結的選項以便讓開發人員能夠控制出現在視覺化即時資料繫結設計家中的類別。

本章討論了更多有關即時資料繫結框架的使用方法和技術，讀者在閱讀完本章的內容之後應該可以使用即時資料繫結框架結合 FireMonkey 框架開發一般的應用程式並且繫結到任何的資料來源了，最後以下面表格總結本書討論框架的使用目的:



| 框架 | 說明 |
|---|---|
| dbExpress 框架 | 提供 C++Builder/C++Builder 跨平台資料存取/處理能力 |
| FireDAC 框架 | 提供 C++Builder/C++Builder 跨平台資料存取/處理能力，所有新的和資料存取有關的應用程式和 App 都應使用 FireDAC |
| FireMonkey 框架 | 提供 C++Builder/C++Builder 跨平台圖形使用者介面能力 |
| 即時資料繫結框架 | 提供 C++Builder/C++Builder 跨平台資料繫結能力 |

這三個框架應該是一般 C++Builder/C++Builder 應用程式共同使用的跨平台框架，並且能夠相互整合提供一致性的服務。

在未來即將推出的 Mobile Studio 中也將使用這 3 個框架來開發 iOS，Android 和其他移動平台的應用程式，因此這 3 個框架也將是未來數年中 C++Builder 最重要的框架，開發人員必須完全的瞭解和掌握它們。


embarcadero
授權代理
揚康科技股份有限公司
電話: 02-23650238
傳真: 02-23650196
信箱: sales@qcomgroup.com.tw
http://embarcadero.qcomgroup.com.tw
版 權 所 有 · 請 勿 翻 印