前面的范例都展示了如何使用TSQLDataSet和TSQLQuery来执行DDL或是直接使用SQL语句来处理数据，不过我们可以发现TSQLDataSet和TSQLQuery一次都只能执行一个SQL语句。当程序员需要执行大量的SQL语句（即所谓的SQL脚本）时，由于dbExpress没有提供类似TSQLScript的组件，因此程序员只能一次执行一个SQL语句，相当不方便。不过在dbExpress提供TSQLScript组件之前，我们也可以使用TSQLDataSet和TSQLQuery来模仿TSQLScript组件，让程序员一次能够执行多个SQL语句。下一小节就展示了这个范例。

2.3.3 执行SQL脚本

本小节要展示的范例是让TSQLDataSet和TSQLQuery执行和2.3.1小节一样的功能，只是本节的范例能够一次执行数个SQL语句，而无需一次执行一个SQL语句。

步骤1：建立数据模块和dbExpress组件

在Delphi/Kylix集成开发环境中点击File|New|CLX Application，再点击File|New|Other...菜单，选择CLX Data Module图标，以建立一个CXL数据模块。接着在此数据模块中加入TSQLConnection组件并且将它连接到范例数据库D7Books，加入TSimpleDataSet组件，设置它的DataSet\CommandText特性值为select * from MYESSAYS。加入TSQLDataSet组件和一个TSQLQuery组件，如图2-35所示。当然，这些组件都需要连接到数据模块中的TSQLConnection组件。

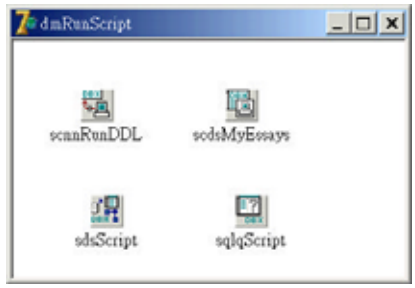

图2-35 范例应用程序的数据模块

本范例要执行的SQL脚本如下：

```
CREATE TABLE MYESSAYS(
    EID INTEGER NOT NULL,
    ETITLE VARCHAR(60),
    MAGAZINE VARCHAR(60),
    PDATE DATE,
    CONTENTS BLOB sub_type 0 segment size 80,
    NOTES VARCHAR(100));

INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(1, '2001年软件巨星-Kylix', 'RUN!PC 2001/3', '03/05/2001 00:00:00',,NULL
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(2, 'Windows原生开发工具的瑰宝-Delphi 7', 'RUN!PC 2001/6', '05/05/2001 00:00:00', NULL );
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(3, 'So much fun, So many possibilities', 'RUN!PC 2001/7', '06/05/2001 00:00:00', NULL );
```

```
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(4, '肥皂的战争与和平', 'RUN!PC 2001/8', '07/05/2001 00:00:00', NULL);
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(5, '.NET的SOAP', 'RUN!PC 2001/9', '08/05/2001 00:00:00', NULL);
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(990, '我的回忆和有趣的故事', 'Programmer深度论坛', '03/01/2001 00:00:00', NULL);
INSERT INTO MYESSAYS(EID, ETITLE, MAGAZINE, PDATE, NOTES) VALUES
(991, '我的回忆和有趣的故事续之一', 'Programmer深度论坛', '04/01/2001 00:00:00',
NULL);

ALTER TABLE MYESSAYS ADD PRIMARY KEY(EID);
CREATE INDEX MYESSAYS_IDX1 ON MYESSAYS(ETITLE);
CREATE INDEX MYESSAYS_IDX2 ON MYESSAYS(PDATE);
```

这个SQL Script会建立MyEssays数据表，新增数个记录，再建立主键值以及数个索引对象。

步骤 2：建立主窗体

回到范例应用程序的主窗体。在主窗体中加入一个 TMemo组件以显示要执行的SQL脚本。放入TDataSource并且连接到数据模块中的 TSimpleDataSet组件，放入TDBNavigator和TDBGrid并且连接到TDataSource。放入一个TOpenDialog组件以便开启SQL脚本文件。最后放入两个 TButton组件，一个加载SQL脚本文件，而另外一个负责执行加载的SQL脚本。这个范例应用程序的主窗体如图 2-36所示。

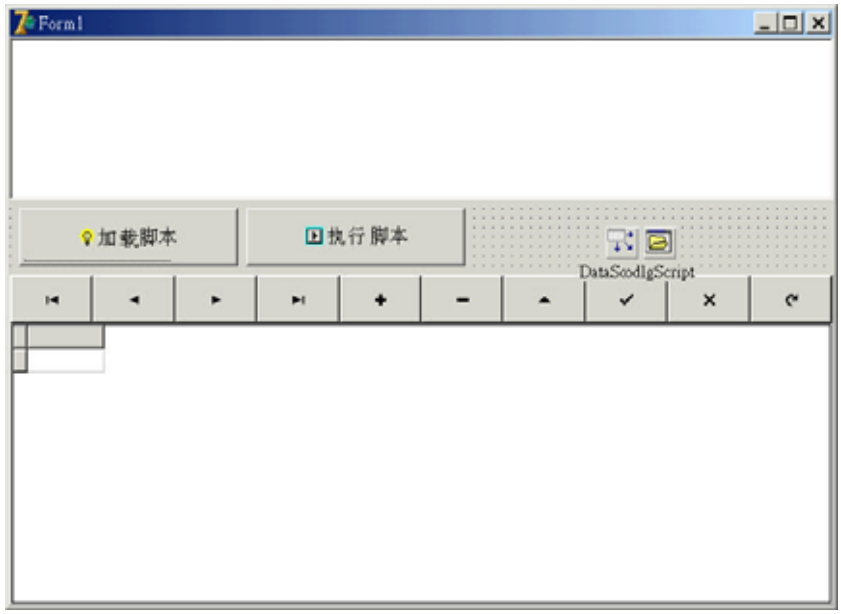

图2-36 范例应用程序的主窗体

现在我们就可以开始实现这个范例应用程序了。

步骤 3：加入实现程序代码

首先双击主窗体中的“加载脚本”按钮，在它的 OnClick事件处理函数中使用TOpenDialog组件加载SQL脚本文件，并且把脚本的内容显示在TMemo中：

```
procedure TForm1.bbtnLoadScriptClick(Sender: TObject);
begin
  if (odlgScript.Execute) then
    mmScript.Lines.LoadFromFile(odlgScript.FileName);
end;
```

接着双击“执行脚本”按钮，在它的 OnClick事件处理函数中编写如下的程序代码：

```
procedure TForm1.bbtnRunScriptClick(Sender: TObject);
const
  sTAG = ';';
var
  sScript : String;
  sSQL : String;
  iPos : Integer;
begin
  sScript := Trim(mmScript.Lines.Text);
  while True do
  begin
    iPos := Pos(sTAG, sScript);
    if (iPos > 0) then
    begin
      sSQL := Copy(sScript, 1, iPos - 1);
      RunScript(sSQL);
      Delete(sScript, 1, iPos);
    end;
    if (Length(sScript) = 0) then
      break;
  end;
  dmRunScript.scdsMyEssays.Active := True;
end;

procedure TForm1.RunScript(const sSQL: String);
begin
{
  dmRunScript.sdsScript.CommandText := sSQL;
  dmRunScript.sdsScript.ExecSQL(True);
```

```
}
  dmRunScript.sqlqScript.SQL.Text := sSQL;
  dmRunScript.sqlqScript.ExecSQL (True) ;
end;
```

上面的程序代码首先从TMemo中取得SQL脚本，然后使用Pos从SQL脚本中一一取出每个SQL语句，再调用RunScript来执行这个SQL语句。在RunScript方法中，可以使用TSQLDataSet或是TSQLQuery组件来执行传入的SQL语句。请注意，在RunScript方法中调用TSQLDataSet或是TSQLQuery组件的ExecSQL方法时，传入了True作为参数。这代表我们要TSQLDataSet或是TSQLQuery组件直接执行SQL语句而不是先准备SQL语句再执行，因为通常SQL脚本中的SQL语句只会执行一次，因此不需要先准备SQL语句，这样可以增加一点效率。

最后当所有的SQL脚本执行完毕之后，再调用数据模块中的TSimpleDataSet的Refresh方法重新从MyEssays数据表中取得所有的数据并且显示在主窗体的TDBGrid中。

步骤 4：执行范例应用程序

现在可以编译和执行这个范例应用程序了，图 2-37显示了范例应用程序加载一个SQL脚本文件，动态建立MyEssays数据表，再新增数个记录，最后建立主键值和索引对象。图2-38使用数据库管理工具证明了这个范例应用程序果然成功地在数据库中建立了各种数据库对象。

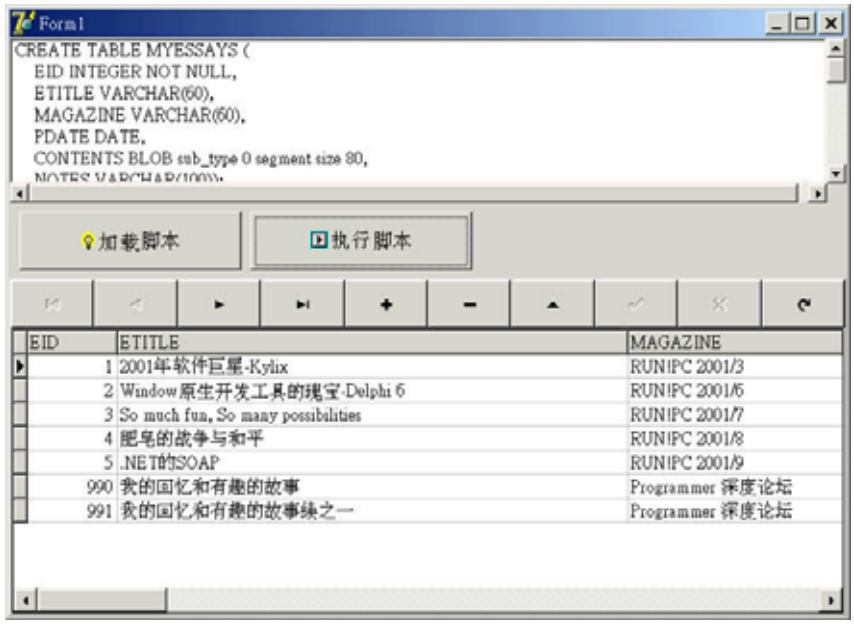

图2-37 执行范例应用程序的画面

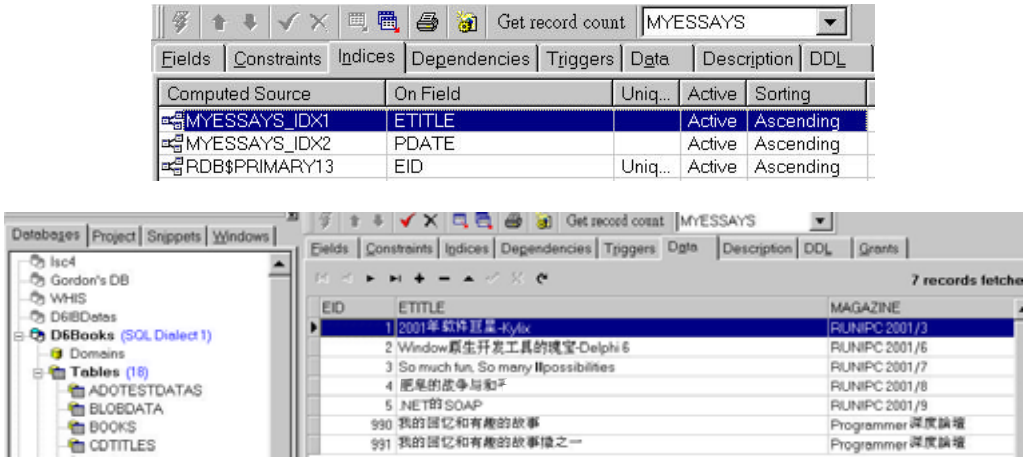

图2-38 范例应用程序建立了数据表以及索引对象

本小节讨论了如何使用TSQLDataSet以及TSQLQuery组件。虽然这两个组件都无法让用户修改它们访问的数据，但是它们可以搭配 TSimpleDataSet组件让程序员可以开发数据库应用程序。对于一些执行 DDL的应用程序或是新增、更新和删除大量数据的情形，使用TSQLDataSet和TSQLQuery组件比使用TSimpleDataSet更适当。在应用系统有这种需求时，程序员便可以使用这两个组件来弥补 TSimpleDataSet组件不足的地方。

2.4 使用TSQLStoredProc组件

dbExpress的TSQLStoredProc组件允许程序员用它执行后端数据源中定义的存储过程。使用TSQLStoredProc组件非常简单，只需要设置它的 DBConnection特性值为一个TSQLConnection，再设置它的StoredProcName特性值为欲执行的存储过程即可。如果欲执行的存储过程需要传入参数，那么只需通过 TSQLStoredProc组件的Params特性值传递参数给存储过程即可。现在让我们使用一个实际的范例来展示如何使用TSQLStoredProc组件。

图2-39是一个在InterBase中实现的存储过程RAISESALARY。这个存储过程接受两个参数，第一个是员工的 ID，第二个参数是欲加薪的百分比。 RAISESALARY可以改变EMPLOYEE数据表中SALARY字段的数值，为特定的员工加薪，现在让我们看看如何使用TSQLStoredProc组件来调用它。

```
CREATE PROCEDURE RAISESALARY (EID VARCHAR(10), RPERCENT FLOAT)
AS
begin
  /* Procedure Text */
```

```
  Update EMPLOYEE
  set
    SALARY = SALARY *(1 + :RPercent)
   where
    EID = :EID;
   suspend;
end
```

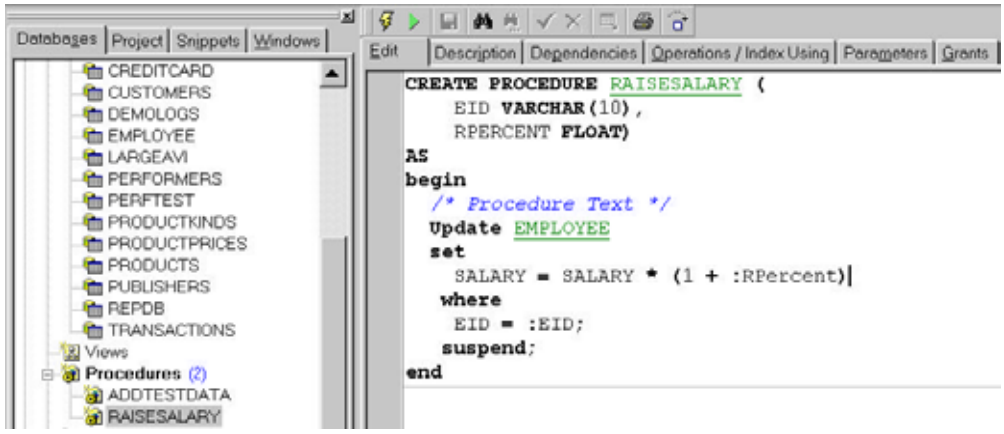

图2-39 在InterBase中定义的RAISESALARY存储过程

步骤1：建立数据模块和dbExpress组件

在Delphi/Kylix集成开发环境中点击File|New|Application建立一个新的Delphi应用程序。接着点击File|New|Data Module建立空白的数据模块。在数据模块中放入TSQLConnection组件并且连接到范例InterBase数据库D7Books.GDB，设置Connected特性值为True。再放入TSimpleDataSet组件，设置它的DBConnection特性值为刚才加入的TSQLConnection，再设置它的DataSet\CommandText特性值为select * from EMPLOYEE，设置Active特性值为True，从EMPLOYEE数据表中选取所有的数据。最后在数据模块中加入一个TSQLStoredProc组件，设置它的SQLConnection特性值为刚才加入的TSQLConnection，再点击它的StoredProcName特性。对象检视器便会显示目前数据库中可以调用的所有存储过程。请从其中选择RAISESALARY这个存储过程。此时数据模块看起来如图2-40所示。

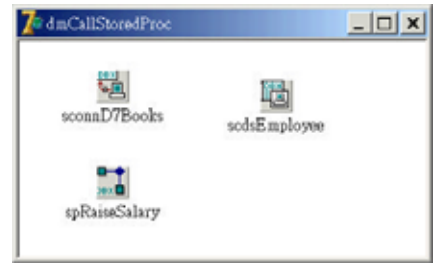

图2-40 范例应用程序的数据模块

步骤2：建立主窗体

在范例应用程序的主窗体中先点击File|Use Unit...菜单，选择使用刚才建立

的数据模块。接着在主窗体中加入TDataSource组件，设置它的DataSet特性值为数据模块中的TSimpleDataSet组件。再加入TDBNavigator组件和TDBGrid组件，设置它们的DataSource特性值为刚才加入的TDataSource。最后放入一个TButton组件、一个TEdit组件和一个TComboBox组件。设置TButton组件的Caption特性值为“加薪”，设置TEdit组件的Text特性值为0.05。此时主窗体看起来如图2-41所示。

图2-41 范例应用程序的主窗体

步骤3：实现范例应用程序

现在到了实现范例应用程序的时候了。首先，在主窗体的 OnActivate事件处理函数中取出EMPLOYEE数据表中所有记录的EID字段值，填入主窗体的TComboBox中以便让用户能够选择要增加那一个员工的薪水。

```
procedure TForm1.FormActivate(Sender: TObject);
var
  aBK : TBookMark;
begin
  if (cbEID.Items.Count = 0) then
  begin
    aBK := dmCallStoredProc.scdsEmployee.GetBookmark;
    try
      cbEID.Items.BeginUpdate;
      dmCallStoredProc.scdsEmployee.First;
      while not dmCallStoredProc.scdsEmployee.Eof do
      begin
        cbEID.Items.Add(dmCallStoredProc.scdsEmployee.FieldByName('EID').Value);
        dmCallStoredProc.scdsEmployee.Next;
      end;
```

```
        cbEID.ItemIndex := 0;
      finally
        cbEID.Items.EndUpdate;
        dmCallStoredProc.scdsEmployee.GotoBookmark (aBK) ;
        dmCallStoredProc.scdsEmployee.FreeBookMark (aBK) ;
      end;
    end;
  end;
```

上面的程序代码首先使用TBookMark的功能记录目前的位置，再一一取出EID字段值填入TComboBox中，最后再通过TBookMark回到起始的位置。

最后就是这个范例应用程序的主体了。在“加薪”按钮的OnClick事件处理函数中我们编写如下的程序代码：

```
procedure TForm1.btnRaiseSalaryClick (Sender: TObject);
begin
  dmCallStoredProc.spRaiseSalary.Params.ParamByName ('EID') .Value :=
    cbEID.Text;
  dmCallStoredProc.spRaiseSalary.Params.ParamByName ('RPERCENT') .Value :=
    StrToFloat (edtPercent.Text) ;
  dmCallStoredProc.spRaiseSalary.ExecProc;
  dmCallStoredProc.scdsEmployee.Refresh;
end;

procedure TForm1.cbEIDChange (Sender: TObject);
begin
  dmCallStoredProc.scdsEmployee.Locate ('EID', cbEID.Text, )[}
end;
```

当用户点击了“加薪”按钮后，我们首先把主窗体中TEdit组件指定的加薪幅度通过数据模块中的TSQLStoredProc组件填入第二个参数中。要想填入存储过程需要的参数，可以使用TSQLStoredProc的Params特性值的ParamByName方法来找到特定的参数。我们也使用相同的方法把用户在主窗体中的TComboBox中选择的员工ID填入存储过程的第一个参数中。最后调用TSQLStoredProc组件的Execute方法执行存储过程，并且调用TSQLClientDataSet的Refresh方法重新显示加薪后的最新数据。

现在编译并且执行这个范例。图2-42中的画面就是执行范例应用程序并且为0000000005员工加薪的画面。从画面中可以看到我们使用TSQLStoredProc组件成功地调用了RAISESALARY存储过程并且完成了加薪的操作。

在上面的范例中，我们调用的是一个不返回数值的存储过程。那么对于会返回数值的存储过程应该如何调用呢？例如，图2-43中的GetRecordCount存储过程会计算Books数据表中所有书籍的总数并且返回这个数值。对于这个存储过程，我们应该如何使用TSQLStoredProc来调用呢？

图2-42　执行范例应用程序的画面

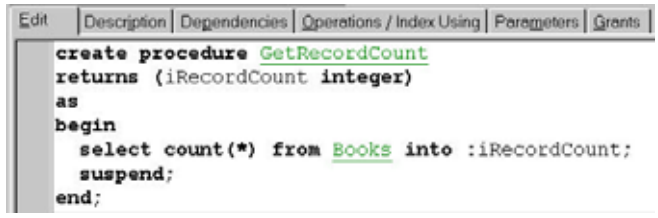

图2-43　在InterBase中定义的GetRecordCount存储过程

这也很简单。对于会返回数值的存储过程而言，当使用 TSQLStoredProc组件调用时，在存储过程执行完毕之后，存储过程会把返回的数值存储在 TSQLStoredProc组件的Params特性值中。程序员只需要调用 Params的ParamByName方法，并且以存储过程返回的数值的名称作为 ParamByName方法的参数即可。但是 Delphi 7联机帮助在这个地方说错了。要想调用会返回数值的存储过程，程序员仍然需要调用TSQLStoredProc组件的ExecProc方法，而不是设置 Active特性值为 True或是调用Open方法。

下面就是调用图2-43中的存储过程的程序代码，在调用ExecProc方法之后，再调用ParamByName方法并且传入IRECORDCOUNT作为参数值即可。

```
procedureTForm1.Button1Click(Sender: TObject);
begin
  dmSP.spBookCount.ExecProc;
  Edit1.Text :=
  IntToStr(dmSP.spBookCount.Params.ParamByName('IRECORDCOUNT').Value);
end;
```

而图2-44就是一个范例应用程序调用GetRecordCount存储过程的画面，GetRecordCount存储过程果然返回了目前Books数据表中的记录数，16。

图2-44 调用GetRecordCount存储过程的结果

dbExpress的TSQLStoredProc简化了调用存储过程的过程，程序员可以使用TSQLStoredProc组件来调用dbExpress支持的关系型数据库中的存储过程。

2.5 使用TSQLMonitor组件

当程序员开发应用程序时，经常会需要进行调试工作。除了调试应用程序的程序代码错误之外，也会想要观察应用程序对于后端数据源使用了什么SQL语句来处理或是修改数据，以便了解应用程序是否使用了正确的SQL语句。当然在许多情形中，能够观察应用程序对于后端数据源使用的SQL语句也有助于进行性能调整的工作，这对于程序员的帮助也非常大。

dbExpress吸引人的地方是它提供了一个TSQLMonitor组件，这个组件使程序员可以追踪前端应用程序对于后端数据源使用的SQL语句。程序员不但可以通过TSQLMonitor组件进行调试，也可以利用它了解dbExpress是如何与后端数据源交互的，以便了解dbExpress和后端数据源的执行行为。

使用TSQLMonitor组件非常简单，只需要将它连接到TSQLConnection并且在它的事件处理函数中编写程序代码，便可以获取dbExpress组件和后端数据源之间互相

传递的信息。现在就让我们说明如何使用它。

步骤1：加入TSQLMonitor组件

首先开启2.1小节建立的范例应用程序，打开数据模块，并且在数据模块中加入一个TSQLMonitor组件，如图2-45所示。

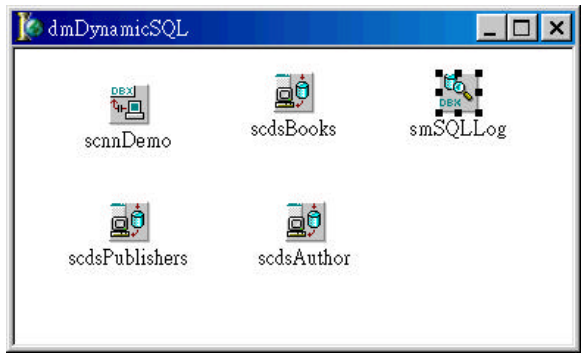

图2-45 范例应用程序的数据模块

接着设置TSQLMonitor的SQLConnection特性值为数据模块中的scnnDemo组件，再设置它的Active特性值为True以激活追踪SQL的功能。

步骤2：修改主窗体

回到范例应用程序的主窗体，在TPageControl中加入一个新的选项卡，并且在其中放入一个TMemo组件，设置它的Name特性值为mmSQLLog，如图2-46所示。

这个TMemo组件将会显示所有由TSQLMonitor组件追踪的消息。

步骤3：实现追踪程序代码

TSQLMonitor组件有两个事件可以让程序员用来追踪dbExpress的消息，分别是OnTrace以及OnLogTrace。OnTrace事件发生在dbExpress和后端数据源间有任何交互消息时，而OnLogTrace则会发生在交互消息已经记录进TSQLMonitor的TraceList特性中之后。要想追踪dbExpress和后端数据源之间的消息，程序员可以根据需要来决定使用哪一个事件。OnTrace以及OnLogTrace事件处理函数的原型如下：

```
type TTraceEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc;
var LogTrace: Boolean) of object;
property OnTrace: TTraceEvent;
type TTraceLogEvent = procedure(Sender: TObject; CBInfo: pSQLTRACEDesc
of object;
property OnLogTrace: TTraceLogEvent;
```

在OnTrace以及OnLogTrace事件处理函数中，程序员可以通过访问pSQLTRACEDesc数据结构中的pszTrace来取得dbExpress和后端数据源之间的命令。

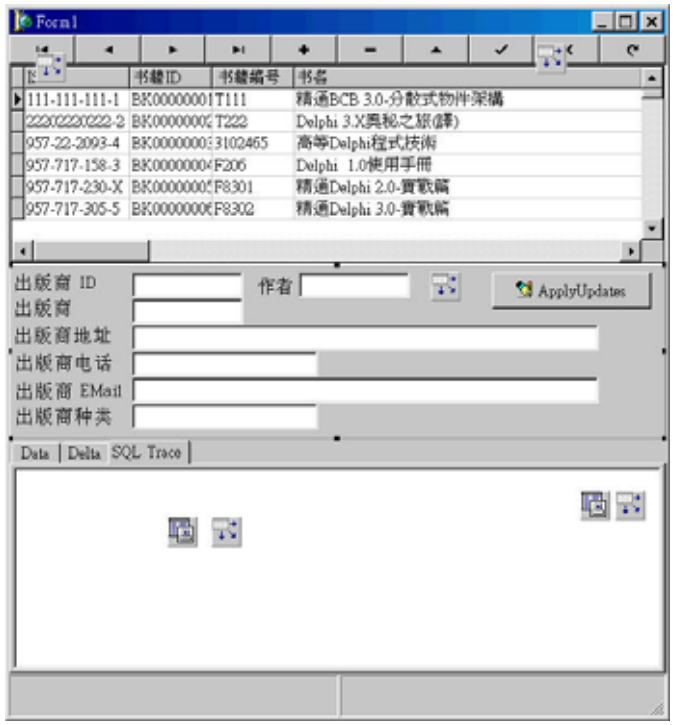

图2-46 在范例应用程序中加入一个新的选项卡和 TMemo组件

了解了TSQLMonitor的这两个事件之后，本节的范例应用程序便可以使用 OnTrace事件来记录dbExpress的命令。请点击数据模块中的 TSQLMonitor组件，再使用对象检视器编写它的OnTrace事件处理函数：

```
procedure TdmDynamicSQL.smSQLLogTrace (Sender: TObject;
  CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
begin
  Form1.mmSQLLog.Lines.Add (CBInfo.pszTrace) ;
end;
```

在OnTrace事件中，我们把 dbExpress和后端数据库之间的命令显示在主窗体的TMemo中。

步骤4：执行范例应用程序

现在可以编译并且执行范例应用程序了，图 2-47是范例应用程序一开始执行时的画面。从画面中可以看到dbExpress下达的命令都显示在主窗体的 TMemo组件中。

现在，先让我们清除主窗体 TMemo组件中的消息，因为我们要开始观察dbExpress如何处理修改数据的工作。首先让我们随便修改一个记录并且 Post修改的数据。图2-48显示了当范例应用程序 Post修改的数据时，TSQLMonitor组件并没有追

踪到dbExpress和后端数据源之间有任何命令，这表示在使用 Post方法进行更新时，dbExpress只是更新暂时存储在内存中的数据，也就是 TSimpleDataSet组件的Data和Delta特性值，并不会真正把修改的数据更新回数据源中。

图2-47 执行范例应用程序时，SQL命令会显示在主窗体的TMemo中

但是如果我们接着点击主窗体中的 ApplyUpdates按钮，那么范例应用程序便会在主窗体的TMemo组件中显示由dbExpress组件向后端数据源下达的命令，这些消息全被TSQLMonitor追踪到了，如图2-49所示。

我们可以通过TSQLMonitor组件证明前面章节讨论的 DataSnap技术的执行行为，也可以观察到dbExpress是如何使用SQL语句的，以及dbExpress使用的SQL语句是不是有效率。TSQLMonitor组件使用起来非常方便，实用性却非常强，连 Borland的工程师也是使用TSQLMonitor来观察和调整dbExpress以及dbExpress引擎的执行行为。

本小节的范例说明了如何使用 TSQLMonitor组件，在稍后讨论性能的章节中本书会说明如何通过TSQLMonitor组件来观察dbExpress应用程序的执行行为并且调整性能。

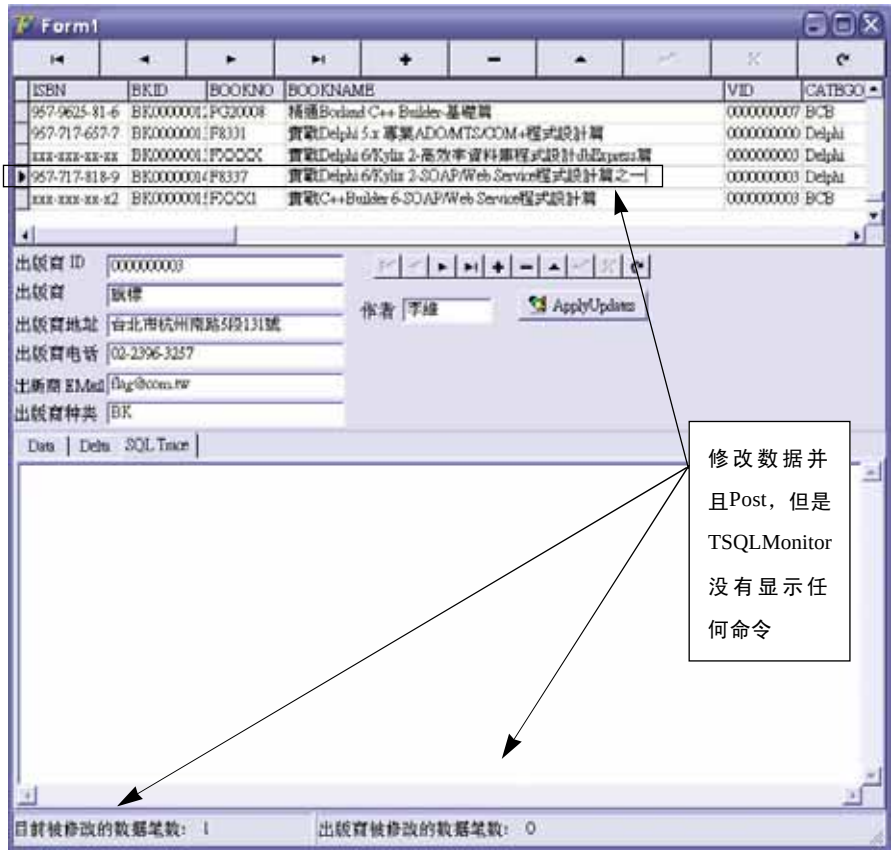

图2-48 修改范例应用程序的数据并且观察 dbExpress的执行行为

2.6 结论

本章说明了如何使用 dbExpress组件组中的 TSimpleDataSet、TSQLDataSet、TSQLQuery以及TSQLStoredProc组件和TSQLMonitor组件。这些组件是程序员在开发数据库应用程序时经常需要使用的，因此切实掌握如何使用这些组件是非常重要的事情。

TSimpleDataSet组件应该是程序员最常使用的组件了，因为它可以帮助程序员访问数据和处理数据的修改，此外它的许多特性值设置也会影响应用程序处理数据的行为和性能，因此 TSimpleDataSet是程序员一定要切实掌握的 dbExpress组件。事实上，TSimpleDataSet组件只是融合了TClientDataSet和TDataSetProvider组件的功能，因此程序员也可以直接使用 TClientDataSet和TDataSetProvider组件，只不过TSimpleDataSet组件简化了程序员需要进行的工作。 TSimpleDataSet组件处理数据的

行为依赖于DataSnap技术，因此本章也详细说明了 DataSnap处理数据的概念，让程序员能够精确地掌握DataSnap技术。

图2-49 应用范例应用程序中的数据更新并且观察 dbExpress的执行行为

除了TSimpleDataSet组件之外，许多数据处理工作也可以使用其他 dbExpress组件来完成，例如 TSQLDataSet或是TSQLQuery组件。虽然这些组件无法处理修改数据的工作，但是却可以进行查询数据或是执行 DDL语句的工作。

如果程序员需要执行后端数据源中的存储过程，那么 TSQLStoredProc组件便是非常好的工具。TSQLStoredProc组件能够轻易地取得数据源中所有存储过程的名称，并且让程序员选择要执行的存储过程，再通过Params特性值来传递存储过程需要的参数。

最后，本章说明了如何使用 TSQLMonitor组件来监视客户端的 dbExpress使用了什么SQL语句与后端数据源交互。通过使用 TSQLMonitor组件，程序员可以掌握客户端dbExpress的执行行为，也可以作为 SQL调试之用。在稍后的章节中也将会使用TSQLMonitor组件来观察和调整 dbExpress应用程序的性能，因此 TSQLMonitor组件可以说是非常重要的dbExpress组件之一。

第3章　更多的dbExpress技巧

本书前面两章讨论的内容应该可以帮助读者顺利地使用 dbExpress来开发基本的数据库应用程序了。但是对于开发真正能够使用的数据库应用系统来说，程序员可能仍然需要许多额外的技巧，例如排序、搜寻数据、如何处理错误、数据库的事务管理等。

从本章开始将会一一介绍这些高级的 dbExpress技巧。读者在掌握了这些经常需要使用的dbExpress技巧之后，才能够轻松地了解后面章节介绍的深入概念和技术。本章将讨论的内容是属于比较容易上手的 dbExpress技巧，在下一章中将会详细讨论如何使用dbExpress来搜寻数据。读者在阅读完本章和下一章的内容之后，将能够切实掌握如何使用dbExpress处理数据。

3.1 数据排序

对数据进行排序是许多数据库应用程序都需要的功能，例如对顾客的身份证号排序、对订单金额排序或是对传票日期排序等。这些功能相信是许多数据库应用程序都经常需要使用的。除了简单的排序之外，许多应用程序也需要做进一步的排序工作。例如当应用程序对顾客的身份证号排序之后，又希望能够根据这个主排序再对顾客名称进行二级排序等。

当然，由于dbExpress使用SQL语句从后端数据源取得数据，因此刚才描述的功能都可以使用SQL语句的Order By和Group By等功能来实现，非常简单。但是对于dbExpress来说，要排序的数据可能已经在客户端的 TSimpleDataSet/TClientDataSet的Data特性值中了，如果使用 SQL语句来重新排序的话，那么就等于浪费了已经存储在客户端的数据。因此对于许多排序的应用而言，可能只是需要对现有的数据排序，而不是从数据源中再次取得数据。如何使用 dbExpress实现这种应用呢？

本小节讨论的内容就在于说明如何使用 dbExpress进行排序的工作。通过讨论各种排序的方法，读者可以了解 dbExpress如何处理排序，并且根据读者自己的需求使用比较适合的方法。

在开始讨论之前，先让我们看看排列范例使用的数据模块。图 3-1是本小节范例应用程序使用的数据模块，这个数据模块使用了TSimpleDataSet组件、TSQLDataSet、TDataSetProvider和TClientDataSet组件。这些组件在稍后的小节中都会一一讨论和使用。其中的scdsDemo和sdsDemo都使用select * from BIOLIFE从BIOLIFE数据表中

选取数据并且进行排序的工作。

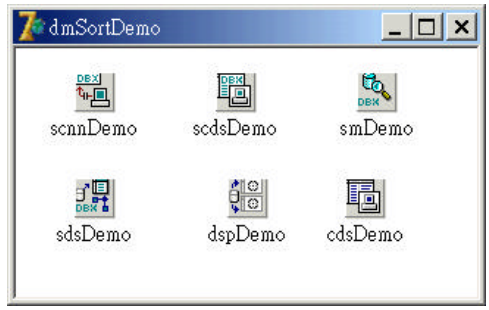

图3-1 本小节使用的范例数据模块

图3-2是范例应用程序使用的主窗体。这个范例应用程序将使用数种方法来进行排序的工作，并且会比较各种排序方法的异同和优缺点。

图3-2 范例应用程序的主窗体

现在我们就可以使用各种排序方式来讨论如何在应用程序中实现排序的功能。

3.1.1 dbExpress/DataSnap默认排序

当使用TSimpleDataSet组件选取数据时，TSimpleDataSet便会自动地帮助程序员建立两个默认的排序方式，第一个是根据选择的数据表的主键建立的默认排序方式，第二个是允许程序员动态修改的排序方式。在dbExpress中分别以DEFAULT_ORDER和CHANGEINDEX来表示。例如，现在如果我们点击数据模块中 scdsDemo的IndexDefs特性值的话，那么就可以看到如图 3-3所示的画面，在IndexDefs特性值中

包含了DEFAULT_ORDER和CHANGEINDEX。如果我们继续点击 DEFAULT_ORDER，那么会在对象检视器中看到这个排序方式是以 SPECIES_NO字段为排序字段的。

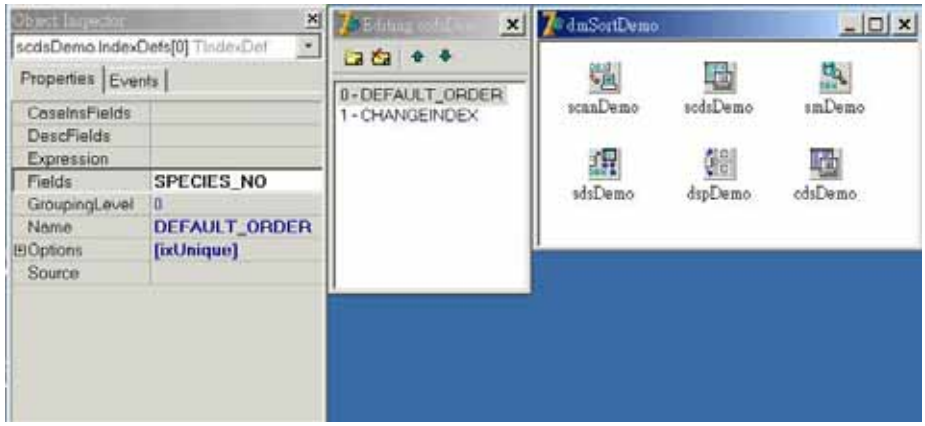

图3-3　TSimpleDataSet组件拥有默认的排序方式

那么为什么DEFAULT_ORDER以SPECIES_NO字段为排序字段呢？如果我们开启BIOLIFE数据表的话，那么从图 3-4的画面中可以看到 SPECIES_NO字段是BIOLIFE数据表的主键字段，因此dbExpress会以这个字段做为默认的排序字段。

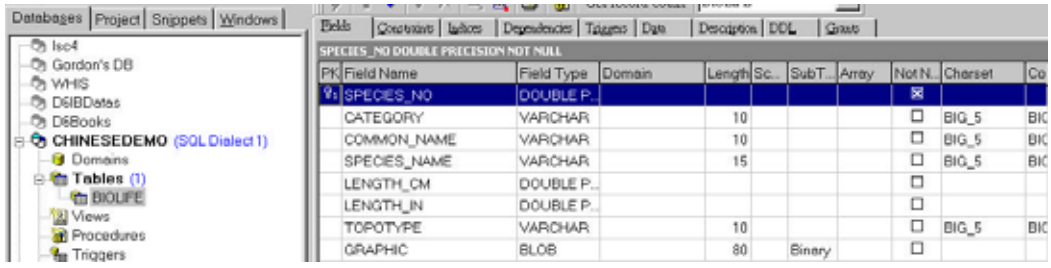

图3-4　范例数据表的纲要，SPECIES_NO是主键字段

因此，如果读者只需要使用主键字段做为排序字段，那么只需要在 TSimpleDataSet的IndexDefs特性值中选择DEFAULT_ORDER为排序方式即可。

但是对于没有定义主键的数据表（虽然这可能是有问题的设计），或是想要动态地使用其他字段来排序的话，那么应该如何做呢？

3.1.2　使用TSQLDataSet的排序特性

如果想要使用除了主键字段之外的其他字段来排序，那么可以使用 TSQLDatsSet组件的SoftFieldNames特性值来指定。例如，现在我们希望范例应用程序使用非主键字段SPECIES_NAME做为排序字段，那么就可以使用对象检视器设置 TSQLDataSet

组件的SoftFieldNames特性值为SPECIES_NAME字段（见图3-5）。

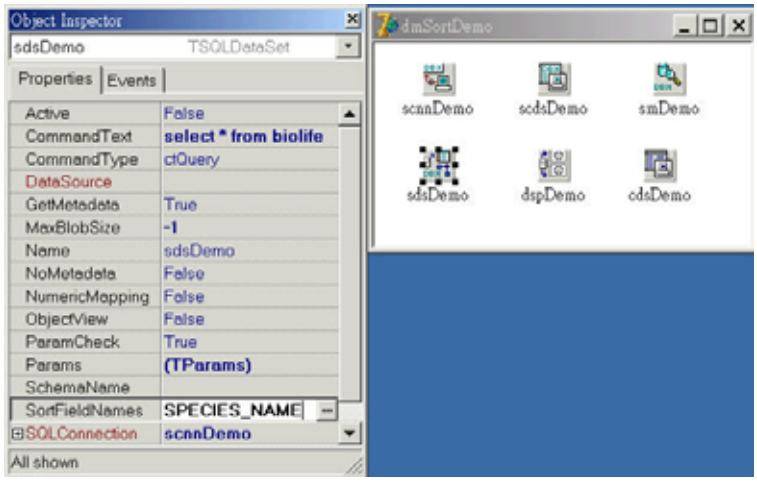

图3-5 TSQLDataSet的SoftFieldNames特性值允许程序员设置排序字段

基本上当程序员设置了TSQLDataSet的SoftFieldNames特性值后，当TSQLDataSet执行CommandText的SQL语句时会在SQL语句之后加上Order By子句，从下面由TSQLMonitor显示的结果可以看得出来：

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from biolife order by SPECIES_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
```

现在如果再次执行范例应用程序，那么我们就可以看到类似于图 3-6的结果。从TDBGrid中可以看到现在数据表中的数据已经按照 SPECIES_NAME字段的次序来排序了。

TSQLDataSet的SoftFieldNames特性除了可以根据一个字段来排序之外，程序员也可以使用 SoftFieldNames特性进行多个字段的排序。例如，现在除了希望以SPECIES_NAME字段作为主排序字段之外，我们还希望以 SPECIES_NO字段做为第二级排序的依据。要进行这样的排序很容易，我们只要在 SoftFieldNames特性中输入这两个字段，并且使用“,”符号分隔每一个字段名称即可。例如，在图 3-7的对象检视器中的SoftFieldNames特性中输入了SPECIES_NAME，SPECIES_NO值，这代表同时使用这两个字段进行数据排序的工作。

现在再次执行应用程序并且使用 TSQLMonitor进行观察，那么我们可以看到下面的SQL语句，这说明dbExpress在order by子句中使用了程序员在 SoftFieldNames特性中输入的字段名称来进行数据排序。

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from biolife order by SPECIES_NAME, SPECIES_NO
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
```

图3-6　在TSQLDataSet的SoftFieldNames特性被设置为排序字段之后，数据便会依据这个特性值来排序

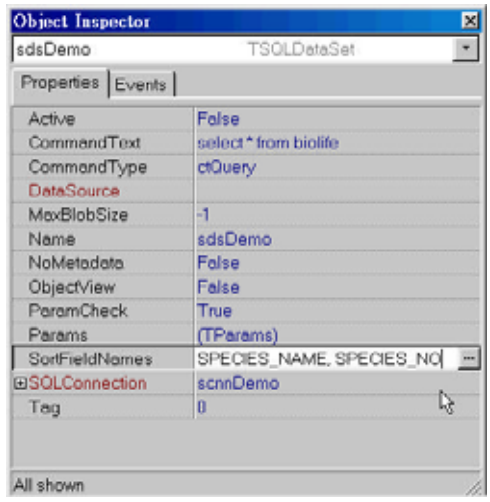

图3-7　在TSQLDataSet的SoftFieldNames特性中设置多个排序字段，每一个字段名称以“,”符号分隔

当然，除了使用TSQLDataSet的SoftFieldNames特性值之外，我们也可以直接使用TSimpleDataSet内建的TInternalDataSet的SoftFieldNames特性值来实现同样的效果，展开TSimpleDataSet的DataSet特性就可以看到TInternalDataSet的SoftFieldNames特性值。

3.1.3 在TSimpleDataSet中进行动态排序

前面讨论的排序技巧大都属于在应用程序设计时期指定排序条件，虽然很方便但是毕竟弹性不够大。在许多的数据应用中，应用程序需要对数据进行动态的排序。例如，在应用程序设计时期程序员在SoftFieldNames特性中使用了SPECIES_NAME字段做为排序条件，但是在应用程序执行时用户可能想使用其他字段来查看数据。这种在应用程序执行时期动态改变数据排序方式的需求是很常见的。那么dbExpress提供了什么方法帮助程序员在应用程序执行时期对数据进行动态排序呢？

dbExpress基本上提供了两种方法让程序员可以在应用程序执行时期进行动态数据排序。第一种是使用AddIndex，第二种是使用IndexDefs。这两种不同的动态排序方式各有特点，在下面的小节中将会讨论如何使用这两种动态排序技巧。

1. 使用AddIndex

TCustomClientDataSet类的AddIndex方法可以为已经存在的数据表动态建立新的索引。在AddIndex方法中，程序员可以指定要作为索引的字段以及如何对这些索引字段的数据进行排序，下面是AddIndex的函数原型：

```
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions; const
DescFields: string = ''; const CaseInsFields: string = ''; const GroupingLevel:
Integer = 0);
```

AddIndex接受数个参数，而这些参数和第3个参数Options有着相当密切的关系。第3个参数Options允许程序员指定使用什么规则来进行排序的工作，例如程序员可以指定数据是以升序、降序或是大小写不敏感的方式排列，或是字段中的所有数据都必须是唯一的值。

第3个参数Options可以指定如下的数值：

```
type
  TIndexOption = (ixPrimary, ixUnique, ixDescending, ixCaseInsensitive,
  ixExpression, ixNonMaintained);
  TIndexOptions = set of TIndexOption;
```

下面的表格总结了TIndexOption值代表的意义：

值	意　义
ixPrimary	此索引是此数据集的主索引对象
ixUnique	此索引是指字段中的数据都必须是唯一的，不能有重复的字段值

（续）

值	意 义
ixDescending	以降序方式排列数据
ixCaseInsensitive	以大小写不敏感的方式排列数据
ixExpression	这个索引是使用dBase表达式来定义的（此值只适合在dBase数据库中使用）
ixNonMaintained	这种索引在修改数据时不会同时自动修改索引对象的信息

如果程序员指定了第2个参数Fields的值，那么数据排序的方式就由第3个参数Options来决定。

如果第3个参数Options是ixDescending，那么程序员在调用AddIndex时就必须指定第4个参数DescFields的值。DescFields是所有需要作为索引的字段名称，每一个字段以分号（;）分隔。如果Options是ixCaseInsensitive，那么AddIndex就必须指定第5个参数CaseInsFields的值，每一个字段也是使用分号（;）分隔。当Options是其他的值时，那么就必须指定第2个参数Fields的值。AddIndex的第一个参数则是新索引的名称。

了解了如何使用AddIndex动态建立索引以便进行数据排序后，我们继续在前面的范例应用程序中加入使用AddIndex进行数据排序的能力。

首先双击图3-6中“排序-AddIndex”按钮，并且在它的OnClick事件处理函数中编写如下的程序代码：

```
procedure TForm1.btnAddIndexClick(Sender: TObject);
var
  sIndexName : String;
begin
  if (AlreadIsIndex(INDEXID + cbFields.Text)) then
    DeleteIndex(cbFields.Text);
  sIndexName := AddIndexForSCDS(cbFields.Text, rbAscending.Checked);
  dmSortDemo.scdsDemo.IndexName := sIndexName;
  Delete(sIndexName, 1, Length(INDEXID));
  slIndex.Add(INDEXID + sIndexName);
  dmSortDemo.scdsDemo.First;
end;
```

在上面的程序代码中，程序代码首先调用AlreadIsIndex方法来检查用户在主窗体edtSortFields控件中输入的要建立索引的字段是否已经是索引字段了。如果是的话，就先调用DeleteIndex删除已经建立的索引对象，再准备重新根据用户输入的字段以及在主窗体中选择的升序或是降序方式建立索引。

接着btnAddIndexClick函数调用AddIndexForSCDS方法，根据edtSortFields控件中的字段名称以及用户选择的排序方式来建立索引对象。在AddIndexForSCDS方法执行完毕之后，新的索引对象便已经建立了。最后，将数据模块中的TSimpleData-

Set组件的IndexName设置为新建立的索引名称，以便使用新建立的索引来进行数据排序。

AlreadIsIndex方法会在slIndex中搜寻是否已经存在参数 sIndex指明的索引名称，并且返回搜寻的结果。

```
function TForm1.AlreadIsIndex (const sIndex: String): Boolean;
var
  iCount : Integer;
begin
  Result := False;

  for iCount := 0 to slIndex.Count do 1
  begin
    if (sIndex = slIndex.Strings[iCount]) then
    begin
      Result := True;
      Break;
    end;
  end;
end;
```

由于slIndex中存储的是索引的名称信息，用于避免用户在 TSimpleDataSet中建立重复的索引信息，因此在范例程序开始执行时我们先调用 TSimpleDataSet的Get-IndexNames方法以便在slIndex中预先存储TSimpleDataSet的所有索引信息。同时调用FillFields在窗体中的TComBox控件中填入TSimpleDataSet的字段名称信息。

```
procedure TForm1.FormShow(Sender: TObject);
begin
  if (slIndex.Count = 0) then
    dmSortDemo.scdsDemo.GetIndexNames (slIndex) ;

  if (cbFields.Items.Count > 0) then
    FillFields (cbFields) ;
  edtSortFields.Text := dmSortDemo.sdsDemo.SortFieldNames;
end;
```

为TSimpleDataSet真正建立索引信息的方法则是 AddIndexForSCDS。在Alread-IsIndex方法成功地查明目前要建立的索引尚不存在之后，范例程序才调用 AddIndex-ForSCDS方法建立索引信息。 AddIndexForSCDS方法会判断用户在窗体中选择的升/降序方式，并且调用AddIndex为TSimpleDataSet实际建立索引信息。

```
function TForm1.AddIndexForSCDS (const sIndex: String;
  bAcsending: Boolean): String;
begin
```

```
    Result := INDEXID + sIndex;
    if (bAcsending) then
      dmSortdemo.scdsDemo.AddIndex(Result, sIndex, [])
    else
      dmSortdemo.scdsDemo.AddIndex(Result, '', [ixDescending], sIndex);
  end;
```

最后，范例程序也提供了删除索引信息的功能。要删除 TSimpleDataSet中的索引信息，只需要调用 DeleteIndex方法即可。DeleteIndex方法接受的参数就是欲删除的索引名称。因此在范例程序的 DeleteIndex方法中就直接调用 TSimpleDataSet的DeleteIndex方法。

```
procedure TForm1.DeleteIndex(const sIndex: String);
begin
  dmSortDemo.scdsDemo.DeleteIndex(INDEXID + sIndex);
end;
```

现在编译此范例程序并且执行它，让我们为数个字段动态建立索引并且观察执行的结果。图3-8和图3-9显示的就是在范例程序中为 SPECIES_NAME和LENGTH_CM字段建立索引的结果。请注意在这些画面中显示了我们不但可以正确地通过AddIndex建立索引，同时AddIndex也可以正确地建立升幂或是降序索引。

到目前为止，我们已经讨论了可以为 TSimpleDataSet/TClientDataSet进行数据排序的数种方法，接下来我们再讨论另外一种数据排序技巧。

图3-8 使用AddIndex动态地为SPECIES_NAME字段建立升序和降序索引

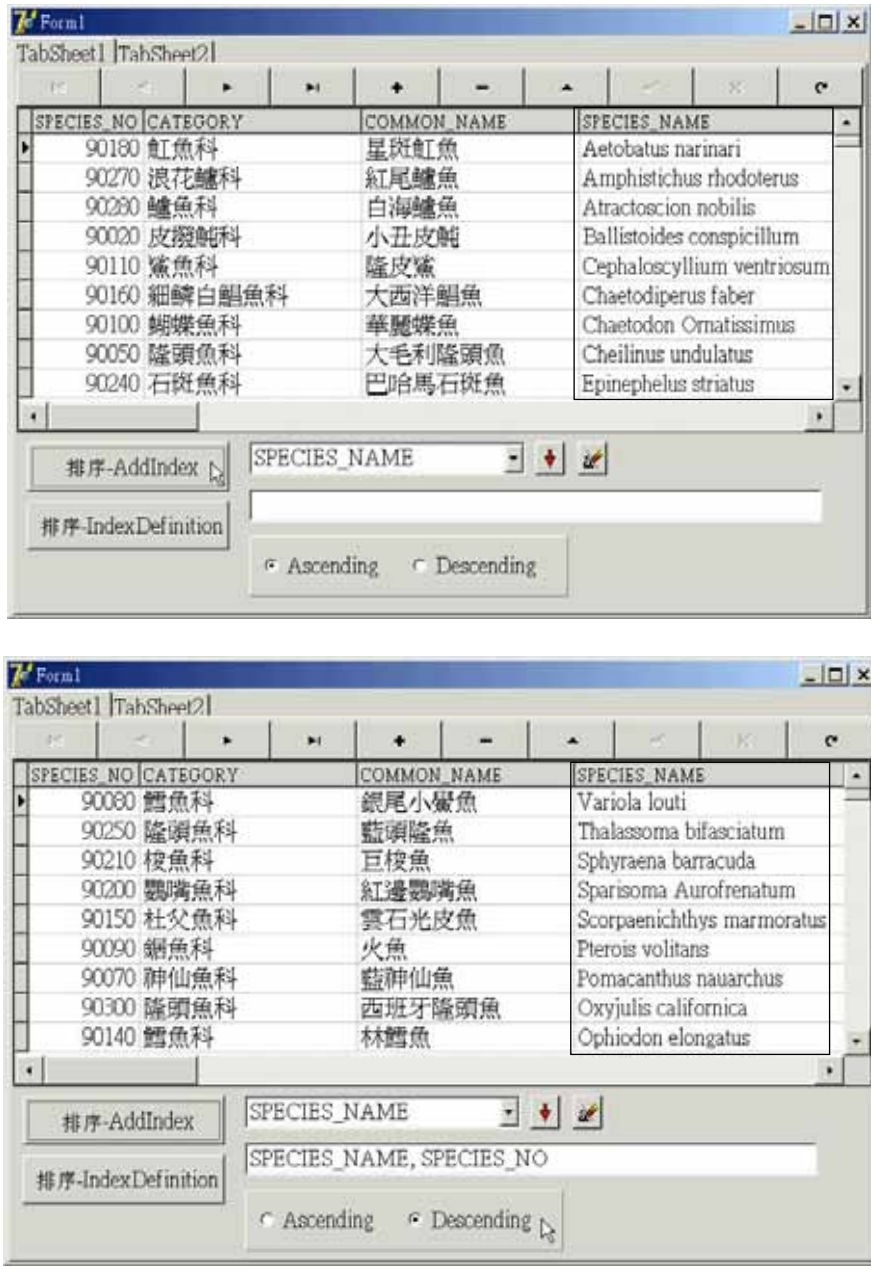

图3-8 （续）

2. 使用IndexDefs和IndexName

除了AddIndex之外，程序员还可以通过TSimpleDataSet的IndexDefs和IndexName特性值同时根据多个字段来进行数据的排序。使用 IndexDefs和IndexName进行排序非常简单。首先程序员调用 IndexDefs的AddIndexDef方法建立一个TIndexDef对象，

然后在TIndexDef对象中设置各种索引信息，例如索引名称、索引的升 /降序方式、索引的目标字段等，和前面的AddIndex使用方式非常相像，最后再把TIndexDef对象的索引名称指定给TSimpleDataSet的IndexName特性值，那么TSimpleDataSet就会使用TIndexDef对象指定的索引方式来对其中的数据进行排序。

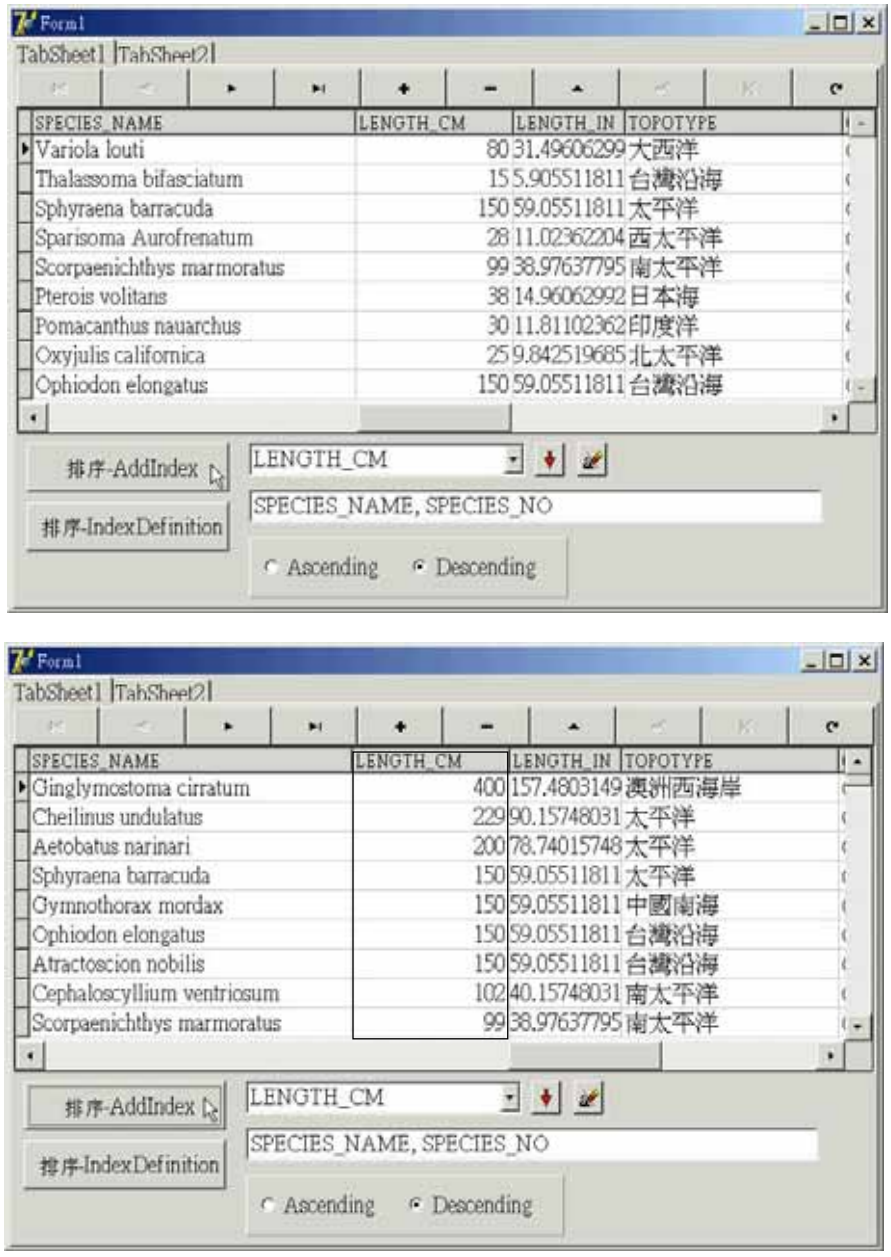

图3-9 使用AddIndex动态地为LENGTH_CM字段建立降序索引

图3-6中的“排序-IndexDefinition”按钮的事件处理函数就使用了TSimpleDataSet/TClientDataSet的IndexDefs和IndexName特性值来进行动态数据排序的工作。在下面的“排序-IndexDefinition”按钮的事件处理函数中，先调用AlreadIsIndexDef方法检查由窗体中TEdit控件edtSortFields指定的字段是否已经建立了索引信息，如果没有的话就调用CreateIndex方法来建立动态索引，最后把CreateIndex建立的索引名称指定给数据模块中TSimpleDataSet的IndexName值，以便对TSimpleDataSet中的数据进行实际排序。

```
procedure TForm1.btnIndexDefsClick(Sender: TObject);
var
  sIndexName : String;
begin
  if (not AlreadIsIndexDef(edtSortFields.Text)) then
  begin
    sIndexName := CreateIndex(edtSortFields.Text, rbAscending.Checked);
    dmSortDemo.scdsDemo.IndexName := sIndexName;
    dmSortDemo.scdsDemo.First;
  end;
end;
```

AlreadIsIndexDef方法使用IndexDefs的FindIndexForFields方法在TSimpleDataSet中搜寻用户指定的字段是否已经建立索引了。如果用户已经使用相同的字段建立过索引，那么FindIndexForFields会返回代表这些字段的TIndexDef对象。在FindIndexForFields方法检查之后，我们还需要调用IndexDefs的Find方法进行相同的检查，只是这次检查是在字段名称之前加上INDEXID这个代表索引前缀的字符串。

```
function TForm1.AlreadIsIndexDef(const sIndex: String): Boolean;
var
  adxf : TIndexDef;
begin
  Result := False;
  adxf :=nil;

  try
    adxf := dmSortDemo.scdsDemo.IndexDefs.FindIndexForFields(sIndex);
    if (Assigned(adxf)) then
    begin
      Result := True;
      exit;
    end;
  except
    on exception do;
  end;
```

```
    try
      adxf := dmSortDemo.scdsDemo.IndexDefs.FindIndex(INDEXID + sIndex);
      if (Assigned(adxf)) then
        Result := True;
    except
      on Exception do;
    end;
  end;
```

如果AlreadIsIndexDef没有搜寻到根据用户指定字段建立的索引信息，那么CreateIndex便会被执行以便实际建立索引。CreateIndex调用IndexDefs的AddIndexDef方法建立TIndexDef对象，再指定TIndexDef对象的Name特性值以设置索引名称，设置TIndexDef对象的Fields特性值以指定排序的字段名称，每一个字段名称也是使用分号（;）分隔的。TIndexDef对象的Options可以设置排序的特性，例如升序、降序或是大小写不敏感方式排序。最后CreateIndex返回建立的索引名称，以便指定给TSimpleDataSet的IndexName特性值。

```
function TForm1.CreateIndex(const sIndex: String; bAcsending : Boolean: String;
var
  adxf : TIndexDef;
begin
  adxf := dmSortDemo.scdsDemo.IndexDefs.AddIndexDef;
  adxf.Name := INDEXID + sIndex;
  adxf.Fields := sIndex;
  if (not bAcsending) then
    adxf.Options := [ixDescending];
  Result := adxf.Name;
end;
```

现在如果我们再次执行范例程序，那么就可以看到IndexDefs和IndexName对数据进行排序的效果了，如图3-10所示。

前面讨论了许多排序数据技巧，有的非常简单，只需要使用对象检视器设置即可，有的比较有弹性，但是需要使用程序代码来进行动态排序。不管使用哪一种方式似乎都可以达到数据排序的目的，非常方便。不过这是不是表示程序员可以自由选择用自己喜欢的技巧来进行数据排序呢？数据排序有没有什么限制呢？

3.1.4 排序时考虑的因素

答案是“是的，程序员可以选择适合的技巧来进行数据排序，但是在排序时程序员必须注意数据排序实际的执行特性”。这是什么意思呢？简单地说，TSimpleDataSet把dbExpress目前访问的数据存储在TSimpleDataSet管理的缓冲存储器中。由于

图3-10 使用IndexDefs进行多字段动态排序

TSimpleDataSet可以采用分段的方式访问数据，因此可能后端数据库中拥有大量的数据，而TSimpleDataSet目前只有少数的数据。但是在使用前面介绍的数据排序方法排列数据时，TSimpleDataSet会先把后端的所有数据读到客户端，再进行数据排序的工作。这会造成多种不利的后果：

- 把大量数据读到前端会造成网络的瞬间大负荷，造成所有客户端应用程序执行缓慢，降低网络的使用率。
- 客户端应用程序可能因为内存不足而造成应用程序错误。
- 需要花费大量的时间，造成应用程序反应迟钝。
- 这可能不是用户想要的结果。

用户可能只想对目前在TSimpleDataSet中的数据进行排序，而不是对后端的所有数据，因此这样做不但没有效率，而且用户也不会接受。现在让我们以一个实际的范例来说明。图3-11显示的就是我们描述的情形，在后端数据表中拥有20000个记录，假设现在用户只想对目前在TSimpleDataSet中的数据进行排序，因此使用了下面的程序代码来进行数据排序。当然这些程序代码是使用前面介绍的排序技巧。

```
procedure TfrmPerfMain.btnAddIndexClick(Sender: TObject);
var
  sIndexName : String;
begin
  LogStartTime;
//  dmDBExpress.sqlmTest.Active := True;
  sIndexName := AddIndexForSort(edtSortFields.Text, rbAscending.Checked);
  dmDBExpress.cdsTest.IndexName := sIndexName;
  ledtRecordCount.Text := IntToStr(dmDBExpress.cdsTest.RecordCount);
  LogEndTime;
  LogRunTime(mmSorts, 'AddIndex Sort' + IntToStr(STLOOPS) + '笔数据时间 : ');
end;
```

但是在上面的程序代码执行之后，从图3-11下方TClientDataSet显示的数据记录数中可以看到，这个范例程序已经把所有20 000个记录读到了客户端，而且花费了2.374秒，更麻烦的是用户如何在这20 000个排序后的记录中找到他原先想要看到的数据？

因此程序员应该尽量避免直接对TSimpleDataSet/TClientDataSet进行数据排序的工作，以避免拖垮dbExpress客户端应用程序的性能。但是如果程序员真的只需要对目前存在于TSimpleDataSet/TClientDataSet中的数据进行排序的话，那么可以使用一点儿小技巧来达成目的，又不会从后端数据库中将所有数据读取到客户端。

这个技巧就是使用TSimpleDataSet/TClientDataSet的CloneCursor方法为TSimpleDataSet/TClientDataSet建立一个复制的游标管理机制，以切断TSimpleDataSet/TClientDataSet与后端数据源的连接，再进行排序。如此一来，当复制的游标进行数据排序时就不会从后端数据源中取得所有数据。CloneCursor方法有如下的原型：

```
procedure CloneCursor(Source: TCustomClientDataSet;
KeepSettings: Boolean = False); virtual;
```

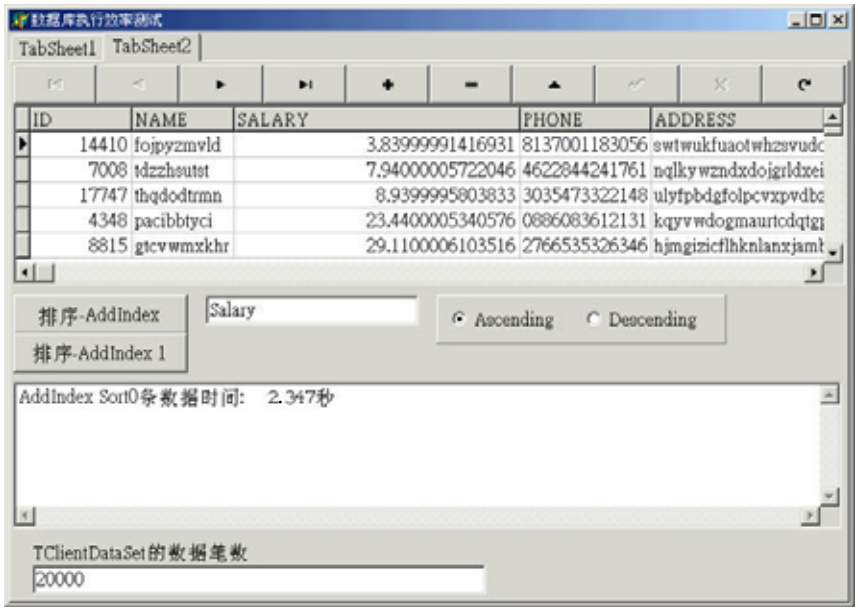

图3-11 TSimpleDataSet在进行数据排序时会把后端的所有数据读到前端，再进行排序工作，这会花费大量的时间

它的第一个参数Source代表要复制的游标的源TSimpleDataSet/TClientDataSet，第2个和第3个参数则代表复制的特性。Reset和KeepSettings参数可以控制下面列出的源TSimpleDataSet/TClientDataSet组件以及目的地TSimpleDataSet/TClientDataSet组件的特性值和事件处理函数：

- Filter、Filtered、FilterOptions和OnFilterRecord
- IndexName
- MasterSource和MasterFields
- ReadOnly
- RemoteServer和ProviderName

下面的表格描述了Reset和KeepSettings这两个参数的设置值对于上面列出的特性值和事件处理函数的影响：

Reset值	KeepSettings值	意　义
False	False	目的地TDataSet在前面列出的特性值都设置成与源TDataSet一样的值
True	N/A	目的地TDataSet在前面列出的特性值设置都会被清除
False	True	目的地TDataSet在前面列出的特性值都不会被改变

程序员可以通过设置Reset和KeepSettings这两个参数值来控制目的地TSimpleDataSet/TClientDataSet的执行特性。

例如，下面的程序代码就是一个很好的范例。在下面的程序代码中，我们首先使

用一个暂时的 TSimpleDataSet/TClientDataSet组件cdsTemp，并且调用它的ConeCursor方法以复制原本拥有数据的 TSimpleDataSet/TClientDataSet组件cdsTest。由于cdsTemp只是复制cdsTest的游标，因此与 cdsTest连接的后端数据源没有连接关系。在这之后，程序代码就只针对 cdsTemp管理的数据进行新增索引的工作来进行数据排序，如此一来就可以避免从后端数据源将大量数据读取到客户端的成本。

```
procedure TfrmPerfMain.Button1Click(Sender: TObject);
var
  sIndexName : String;
begin
  LogStartTime;
  dmDBExpress.cdsTemp.CloneCursor(dmDBExpress.cdsTest, True, False);

  sIndexName := INDEXID + edtSortFields.Text;
  if (rbAscending.Checked) then
    dmDBExpress.cdsTemp.AddIndex(sIndexName, edtSortFields.Text, [])
  else
    dmDBExpress.cdsTemp.AddIndex(sIndexName, edtSortFields.Text, [ixDescending]);

  dmDBExpress.cdsTemp.IndexName := sIndexName;
  ledtRecordCount.Text := IntToStr(dmDBExpress.cdsTemp.RecordCount);
  LogEndTime;
  LogRunTime(mmSorts, 'AddIndex Sort 1' + IntToStr(SLOOPS) + '笔数据时间 : ');
end;
```

图3-12显示的是使用上面的技巧之后与图 3-11中相同的数据排序。从图 3-12可以看到使用CloneCursor技巧之后，客户端的记录只有 100个，而且整个数据排序过程的时间减少到 0.01秒。当然这是因为现在我们只对目前存在于 TSimpleDataSet/TClientDataSet中的数据进行排序。

除了使用CloneCursor之外，如果程序员希望排序的数据能够与原始数据之间有更清楚的分隔，那么程序员可以把拥有原始数据的 TSimpleDataSet/TClientDataSet组件的Data特性值指定给另外一个独立的 TSimpleDataSet/TClientDataSet组件，再针对后一个TSimpleDataSet/TClientDataSet组件进行数据排序。这是因为 TSimpleDataSet/TClientDataSet组件的 Data特性值包含的是目前存在于 TSimpleDataSet/TClientDataSet中的数据。

下面总结了程序员在使用 TSimpleDataSet/TClientDataSet进行数据排序时应该牢记在心的事情：

- 在默认情况下，当程序员针对 TSimpleDataSet/TClientDataSet中的数据进行处理时，TSimpleDataSet/TClientDataSet都会尝试把后端的所有数据读取到客户

端再进行处理，因此程序员必须注意这个执行行为带来的后果。

- 如果程序员只想对目前在TSimpleDataSet/TClientDataSet中的数据进行排序，那么就使用CloneCursor或是额外的TSimpleDataSet/TClientDataSet组件进行排序。
- 如果程序员真的想对所有数据进行排序，那么请直接改变 TSimpleDataSet/TClientDataSet的CommandText特性值，使用SQL语句的Order By子句来取得排序的数据，让后端数据源来帮助进行数据排序。

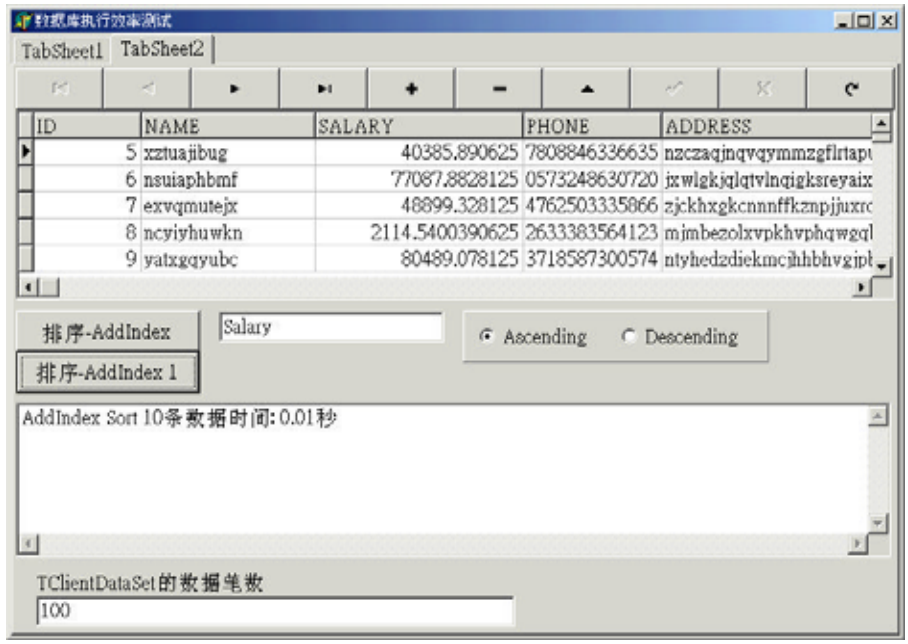

图3-12 通过小技巧可以避免TSimpleDataSet把所有数据读取到前端之后才排序数据

3.2 内存数据表

在开发应用系统时，经常会需要暂时存储一些应用程序执行时期的数据。这些数据可能不存在于数据库中，但是在应用程序执行时却需要经常引用这些数据。对于这种类型的数据以及应用而言，程序员可能会使用各种不同的方法来解决。不过dbExpress中的TSimpleDataSet/TClientDataSet组件却可以提供非常好的解决方案，如果程序员能够再结合上一小节讨论的排序技术，那么就可以在应用程序中实现快速的内存数据表（Memory Table），来进行暂时数据的存储以及对于这些暂时数据的频繁查询的工作。

例如，假设在一个范例应用程序中产生了下列表格中的暂时数据。这些暂时数据可能有数十个记录，而且在应用程序执行时经常需要根据ID来查询相应的字符串值。

ID（整数值）	值（字符串）
1	DataSnap
2	WebSnap
3	Web Service
4	SOAP
5	XML
6	Interface
7	COM+
...	...

对于这种应用，程序员可以使用TSimpleDataSet/TClientDataSet建立内存数据表，然后根据ID字段排序，再让应用程序使用这个内存数据表来存储和查询数据。如此一来，不但可以解决存储暂时数据的问题，也可以很有效率地查询暂时数据。此外，使用TSimpleDataSet/TClientDataSet建立的内存数据表还可以存储多个字段值，也可以存储各种类型的数据，所有的内存数据表内容也可以动态产生，好处非常多。当然，TSimpleDataSet/TClientDataSet的内存数据表也可以在应用程序执行完毕之后把这些暂时数据存储成XML格式的数据，等下次应用程序再次执行时直接从XML中加载这些暂时数据，而无需再次重新建立。本小节讨论的内容就是说明如何使用TSimpleDataSet/TClientDataSet来动态建立内存数据表，以便处理和查询暂时数据。

在下面的范例应用程序中将会使用TSimpleDataSet建立一个内存数据表，并且在这个内存数据表中填入应用程序经常需要使用和查询的数据。由于这些数据存储在内存中，因此查询的速度非常快，也不会增加数据库服务器的负荷。

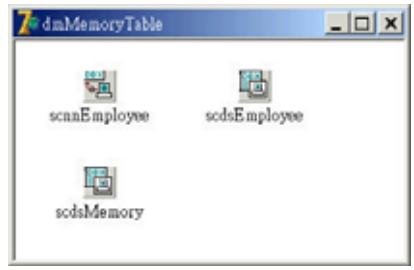

图3-13 范例应用程序的数据模块，使用两个TSimpleDataSet组件，其中scdsMemory是准备作为内存数据表使用的

步骤1：设计范例应用程序

首先在Delphi/Kylix集成开发环境中建立一个新的应用程序项目，再建立一个数据模块。在数据模块中放入TSQLConnection以及两个TSimpleDataSet组件，如图3-13所示。

接着设置数据模块中的dbExpress组件特性值如下：

TSQLConnection：

特性名称	设置特性值
Name	scnnEmployee
Database	e:\Program Files\Common Files\Borland Shared\Data\Employee.gdb

TSimpleDataSet：

特性名称	设置特性值
Name	scdsEmployee
DBConnection	scnnEmployee
CommandText	select * from EMPLOYEE

TSimpleDataSet：

特性名称	设置特性值
Name	scdsMemory

请读者注意，在数据模块中的 scdsMemory中没有设置其他特性值，也没有连接到任何数据表。scdsMemory的内部结构以及数据会在稍后由范例应用程序动态地填入。此外在这个范例中是使用 InterBase的范例数据库Employee.gdb。

接着再设置范例应用程序的主窗体，如图 3-14所示。

图3-14 范例程序主窗体

在图3-14主窗体中有两个 TButton组件，一个用来建立内存数据表以及填入应用程序使用的数据，另外一个 TButton组件则让用户在ID控件中输入搜寻数据的键值之后搜寻内存数据表中的相应数据。主窗体下方的 TDBGrid用来显示应用程序动态建

立的内存数据表，而主窗体右方的 TListBox则会显示建立内存数据表以及搜寻内存数据表花费的时间。

TButton：

特性名称	设置特性值
Name	btnCreateMemoryTable
Caption	产生暂时数据

TButton：

特性名称	设置特性值
Name	btnSearchData
Caption	搜寻暂时数据

设计完范例应用程序的控件之后，现在可以开始实现此范例应用程序，看看如何使用TSimpleDataSet建立内存数据表。

步骤 2：实现范例应用程序

首先实现btnCreateMemoryTable按钮的OnClick事件处理函数。当用户点击了这个按钮之后，btnCreateMemoryTable就会使用数据模块中的 scdsMemory动态建立内存数据表。

下面即是btnCreateMemoryTable按钮的OnClick事件处理函数，首先它会开始计时以计算建立内存数据表和填入暂时数据需要花费的时间，接着它调用CreateMemoryTable函数以建立内存数据表的内部结构，例如字段名称、字段类型等元数据。在成功地建立了内存数据表之后，它会再调用 FillTempData以便在内存数据表中填入应用程序需要使用的暂时数据。最后它显示完成这些工作花费的时间。

```
procedure TForm1.btnCreateMemoryTableClick(Sender: TObject);
begin
  StartTime;
  CreateMemoryTable;
  FillTempData;
  EndTime;
  ShowAppMsg('建立内存数据表和数据时期 : ' + FloatToStr(GetRunTime));
end;
```

CreateMemoryTable是真正建立内存数据表的函数。在 CreateMemoryTable中将会为数据模块中的scdsMemory建立字段信息以存储暂时数据，并且会建立索引字段以增加搜寻数据的速度。

CreateMemoryTable首先调用scdsMemory的FieldDefs特性的AddFieldDef方法，以便在scdsMemory中建立暂时的字段定义对象， TFieldDef。当应用程序建立了字段

定义对象之后，必须设置这个字段定义对象的数据类型、字段长度以及字段名称等重要的特性值。从下面的程序代码中我们可以看到，CreateMemoryTable在scdsMemory中分别建立了两个字段定义对象，第一个字段定义对象是定义为整数类型的字段，这个字段的名称是TID。第二个字段定义对象是字符串类型的，它的字段长度是20个字符，而字段的名称是TSValue。

接着CreateMemoryTable又为scdsMemory定义了一个索引定义对象，TIndexDef。索引定义对象可以为数据集建立字段索引，程序员需要设置要建立索引的字段名称以及这个索引本身的名称。从下面的程序代码中可以看到，这个索引定义对象为TID字段建立索引，并且设置这个索引的名称是idxTID。

```
procedure TForm1.CreateMemoryTable;
begin
with dmMemoryTable.scdsMemory do
  begin
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftInteger;
      Name := 'TID';
    end;
    with FieldDefs.AddFieldDef do
    begin
      DataType := ftString;
      Size := 20;
      Name := 'TSValue';
    end;
    with IndexDefs.AddIndexDef do
    begin
      Fields := 'TID';
      Name := 'idxTID';
    end;
    CreateDataSet;
    IndexDefs.Update;
    IndexName := 'idxTID';
  end;
end;
```

当scdsMemory通过字段定义对象和索引定义对象设置了要定义的字段以及索引之后，就可以调用TSimpleDataSet的CreateDataSet方法建立一个不包含数据的新数据集了。最后记得调用TSimpleDataSet的IndexDefs对象的Update方法以便反映最新的数据表索引信息。

FillTempData非常简单，它只是在scdsMemory建立的内存数据表中新增数个记录，其中第一个字段值是整数，第二个字段值是字符串数据。

```
procedure TForm1.FillTempData;
begin
  InsertData(1, 'DataSnap');
  InsertData(2, 'WebSnap');
  InsertData(3, 'Web Service');
  InsertData(4, 'SOAP');
  InsertData(5, 'XML');
  InsertData(6, 'Interface');
  InsertData(7, 'COM+');
end;

procedure TForm1.ApplyData;
begin
  dmMemoryTable.scdsMemory.ApplyUpdates(0);
end;

procedure TForm1.InsertData(const iID: Integer; const sValue: String);
begin
  with dmMemoryTable.scdsMemory do
  begin
    Insert;
    FieldByName('TID').Value := iID;
    FieldByName('TSValue').Value := sValue;
    Post;
  end;
end;

procedure TForm1.EndTime;
begin
  lEnd := GetTickCount;
end;

function TForm1.GetRunTime: double;
begin
  Result := (lEnd - lStart) / 1000.0;
end;

procedure TForm1.StartTime;
begin
  lStart := GetTickCount;
end;

procedure TForm1.ShowAppMsg(const sMsg: String);
begin
  lbTimes.Items.Add(sMsg);
end;
```

完成了btnCreateMemoryTable的OnClick事件处理函数之后， btnSearchData的OnClick事件处理函数就简单多了。当用户点击了 btnSearchData按钮之后，它就直接

使用scdsMemory刚才建立的内存数据表，调用Lookup方法来根据用户在主窗体中的ID控件中输入的ID键值搜寻数据。

```
procedure TForm1.btnSearchDataClick(Sender: TObject);
begin
  StartTime;
  edtValue.Text:=dmMemoryTable.scdsMemory.Lookup('TID',edtID.Text,
  'TSValue');
  EndTime;
  ShowAppMsg('搜寻数据时间 : ' + FloatToStr(GetRunTime));
end;
```

现在这个建立并使用内存数据表的范例应用程序已经完成了，让我们执行这个范例应用程序来观察执行的结果。

执行范例应用程序，点击“产生暂时数据”按钮建立内存数据表并且填入暂时数据，再在ID控件中输入ID值2搜寻数据，结果见图3-15。在主窗体右方的TListBox中可以看到，建立内存数据表并且填入暂时数据的速度非常理想。而搜寻内存数据表的数据更是快速无比，充分地显示了内存数据表吸引人的地方。

图3-15 使用内存数据表存储数据

内存数据表这种建立和搜寻速度快的特性非常适合在少量、经常使用的数据应用中使用。不过读者必须注意，内存数据表不适合用来存储大量的暂时数据，例如超过数千个记录的情形就不适合使用。读者必须根据数据的特性以及客户端机器的内存情况适当决定是否要使用内存数据表。

3.3 使用计算字段

Delphi/Kylix为客户端的数据集提供了计算字段的功能，以帮助程序员建立在编写应用程序时需要的暂时字段。所谓计算字段（Calculated Field）是指在许多数据库应用程序中经常需要使用一些暂时的字段来存储一些数值，以方便应用程序进行计算工作。由于这些字段只是在应用程序执行时帮助应用程序完成工作，而不永久存储在数据库中。对于这类的应用，计算字段便是很好的解决方案。

由于DataSnap原本就是在客户端的内存中建立数据集结构，因此DataSnap当然也可以在这个内存结构中建立额外的不存在于数据表中的暂时字段。Delphi/Kylix提供了方便的UI和事件处理函数帮助程序员使用计算字段来解决数据库应用程序的需求。在UI方面，程序员可以通过激活数据集组件的字段编辑器来加入任意数量的计算字段，而且能够建立任何类型的暂时字段。另外，数据集的AutoCalcFields特性值可以控制计算字段的执行行为，OnCalcFields事件处理函数则是程序员使用计算字段的地方。在OnCalcFields事件处理函数中，程序员可以使用Object Pascal程序代码来为计算字段进行任何计算。OnCalcFields事件处理函数的原型如下：

```
type TDataSetNotifyEvent procedure (DataSet: TDataSet) of object;
```

其中参数DataSet是指触发此事件的数据集，也就是拥有此计算字段的数据集。

由于计算字段的弹性非常大，因为程序员可以定义任意数量和类型的计算字段，又可以使用Object Pascal在OnCalcFields事件处理函数中进行任何计算工作，因此计算字段在许多数据库应用程序中经常使用。而在Delphi/Kylix中使用计算字段是非常简单的，只需要完成下面的两个操作即可：

- 使用数据集的字段编辑器加入计算字段或是其他类型的暂时字段。
- 在OnCalcFields事件处理函数中使用程序代码执行计算字段的工作。

现在就让我们使用一个简单的范例来说明如何使用简易的计算字段帮助应用程序进行暂时的运算工作。在这个范例中，我们希望能够了解为员工加薪之后每一个员工的薪水，以决定加薪的幅度。由于这个应用只是满足决策者暂时的需求，而不需要把每一个加薪计算的暂时结果存储在数据表中，因此计算字段在这个应用中是非常适合的。因此我们将在数据集中建立一个暂时的浮点类型的计算字段以存储每一次的计算结果。

首先是激活目标数据集组件的字段编辑器以准备新增计算字段（图 3-16）。

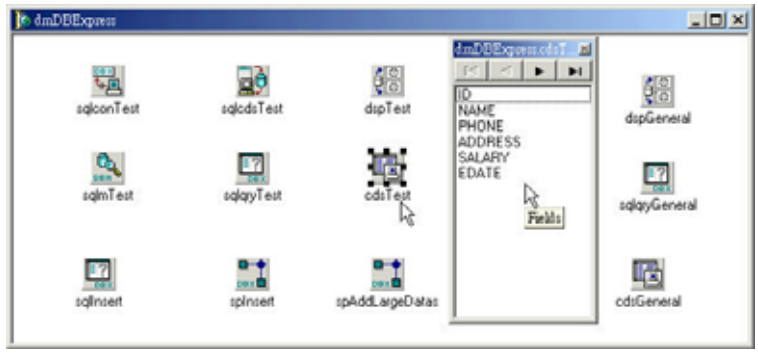

图3-16 激活TClientDataSet的字段编辑器

接着点击鼠标右键激活快捷菜单，从其中选择 New field...项目以激活New Field对话框，定义要建立的字段对象种类，见图 3-17。

图3-17 在字段编辑器中激活快捷菜单，选择 New Field...以建立暂时的字段

此时，Delphi/Kylix会显示如图3-18所示的对话框让程序员定义各种不同类型的暂时字段对象，定义字段的种类以及字段名称等特性。例如，在图 3-18的对话框中便定义了一个类型为浮点数，名称为 AdjustedSalary的计算字段。

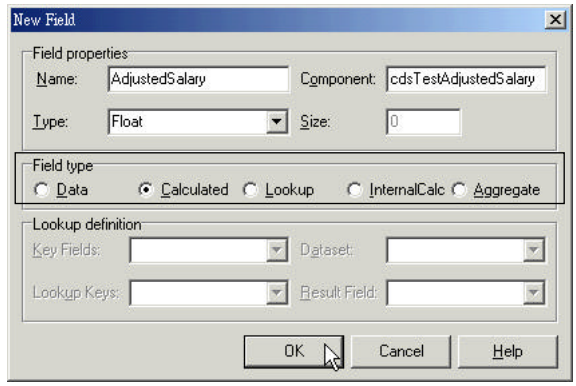

图3-18 在New Field对话框中建立计算字段，设置其名称和字段类型

每一个数据集允许建立的暂时字段并不相同，下面是 Delphi/Kylix允许建立的暂时字段类型，但是有一些暂时字段只能在特定的数据集中使用。例如 TSimpleDataSet便没有提供InternalCalc和Aggregate类型的暂时字段，但是TClientDataSet提供了所有类型的暂时字段。

字段类型	目 的
Data	取代现有数据表中的字段
Calculated	计算字段，在OnCalcFields事件处理函数中动态计算字段值
Lookup	用来查询信息的查询字段
InternalCalc	客户端数据集使用的内部计算字段，用来暂时存储数据的字段。与 Calculated字段不同，InternalCalc并不在OnCalcFields事件处理函数中计算
Aggregate	Aggregate类型的字段，下一小节将会说明

在图3-18中定义了AdjustedSalary计算字段之后，我们便可以在包含这个计算字段的数据集组件的OnCalcFields事件处理函数中编写如下的程序代码在应用程序执行时动态地计算此计算字段的结果，也就是根据不同的加薪幅度来计算每一个员工的薪资信息：

```
procedure TdmDBExpress.cdsTestCalcFields(DataSet: TDataSet);
begin
  try
    DataSet.FieldByName('AdjustedSalary').Value :=
      DataSet.FieldByName('Salary').Value *
       (1 + StrToFloat(frmPerfMain.ledtAdjustedPercent.Text) / 100.0);
  except
    on Exception do;
  end;
end;

procedure TfrmPerfMain.bbtnAdjustClick(Sender: TObject);
begin
  dmDBExpress.cdsTest.Edit;
  dmDBExpress.cdsTest.Cancel;
end;
```

上面的程序代码先取出真正存储员工薪资的字段的值，再乘以用户在应用程序中输入的加薪百分比，并且把计算的结果存储在计算字段 AdjustedSalary中。请读者注意，当在数据集中加入了计算字段之后，程序员便可以按照使用数据表中的真正字段的方式来访问和使用计算字段。

图3-19便是范例程序使用计算字段的画面。从图 3-19中读者可以发现计算字段对于数据感知组件来说就像是真正的数据表字段一样，可以显示或是进行任何修改。程序员当然也可以再根据用户对计算字段的修改使用程序代码来决定如何进行额外的处理。

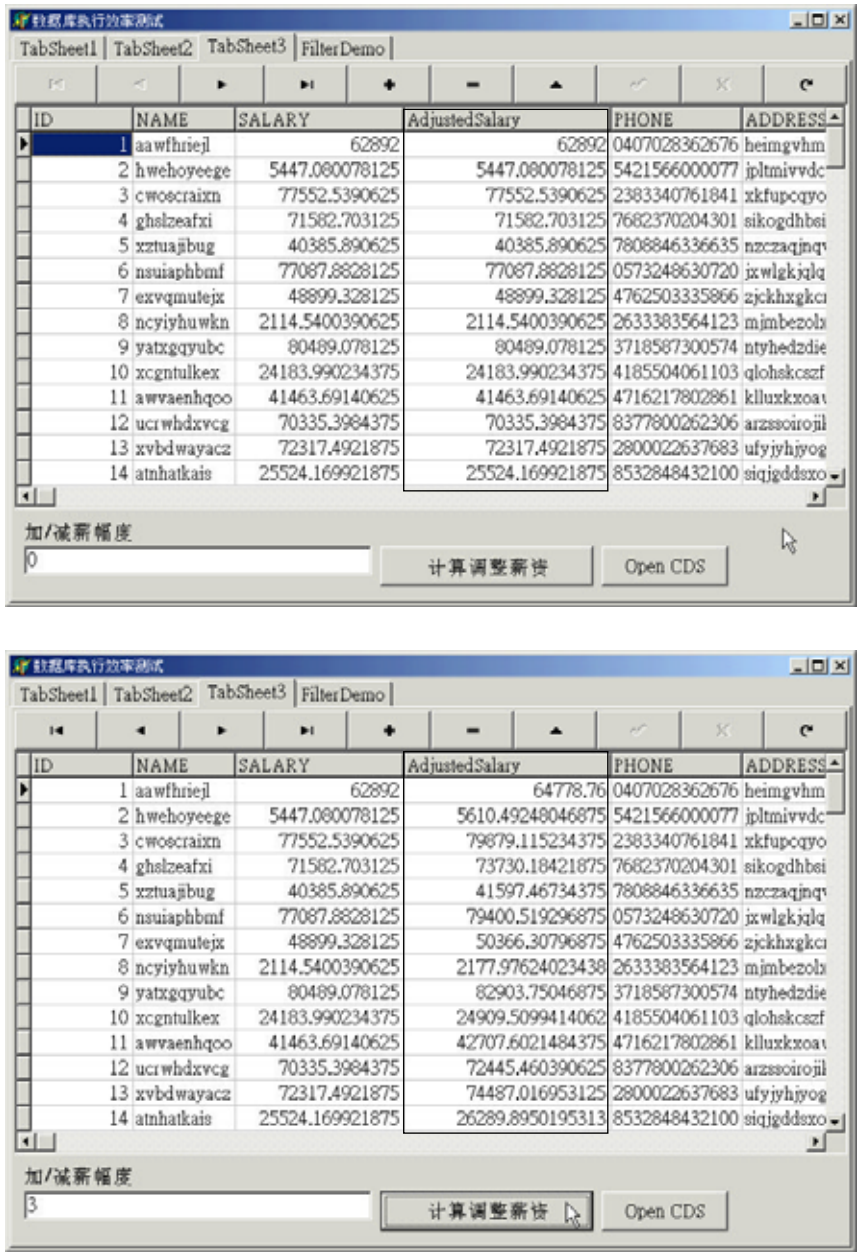

图3-19 计算字段执行的状态

虽然计算字段在使用上很方便，又很具弹性，但是数据集在许多情况下都会触发OnCalcFields事件处理函数，这造成了应用程序大量的计算负荷，因此读者必须注意不要在OnCalcFields事件处理函数中执行太复杂的计算，以避免应用程序的性能被计算字段拖垮。

3.4 使用Aggregate字段

Delphi/Kylix定义了一种特别的字段称为Aggregate字段，从Aggregate的名称便可以了解到它的目的是用来计算客户端数据集中数据的各种特殊值。程序员可以使用Aggregate计算客户端数据集中数据的总和、平均值或最大和最小值。Aggregate字段对象允许程序员使用表达式，并且使用Aggregate预先定义的一些操作数来帮助程序员进行计算的工作，这些预先定义的操作数总结在下面的表格中：

操 作 数	使用意义
Sum	计算数值字段的总和
Avg	计算数值字段或是日期字段的平均值
Count	计算非空字段的总数量
Min	取得数值、日期或是字符串字段的最小值
Max	取得数值、日期或是字符串字段的最大值

代表Aggregate字段的类是TAggregate类，TAggregate类中有一个Expression特性值，在Expression中程序员可以使用上面表格定义的操作数来进行计算的工作。使用Aggregate字段是非常简单的，而且我们在前面图3-18中已经看过了Aggregate字段。因此要使用Aggregate字段，程序员只需采用下面的两个步骤即可：

- 使用字段编辑器定义Aggregate字段。
- 在Aggregate字段的Expression特性值中使用操作数输入要执行的表达式。

之后，程序员就可以在程序代码中访问Aggregate字段，就像访问一般的字段一样。一旦程序员访问Aggregate字段，Aggregate字段就会执行它的Expression特性值中定义的表达式，执行的结果就会成为Aggregate字段的值。

现在让我们继续使用上一小节中的范例来说明如何使用Aggregate字段，现在我们除了为每一位员工计算加薪之后的薪资之外，还希望得到Salary字段的总和。这可以使用Aggregate字段轻易地做到，因为Aggregate字段提供的sum操作数可以计算Salary字段的总和。

因此我们先点击客户端数据集组件，接着在对象检视器中点击Aggregates特性值激活Aggregate编辑器，然后加入一个新的Aggregate字段，如图3-20所示。

接着在Aggregate字段的Expression特性值中输入如下的表达式：

```
Sum (Salary)
```

这个表达式使用Aggregate字段提供的sum操作数来计算Salary字段值的总和。接着我们只需要在程序代码中直接访问这个Aggregate字段，就像访问一般的字段那样，这样就可以取得Aggregate字段代表的Salary字段值的总和：

```
procedure TfrmPerfMain.bbtnAdjustClick (Sender: TObject);
begin
```

```
    dmDBExpress.cdsTest.Edit;
    dmDBExpress.cdsTest.Cancel;

    Self.ledtTotalAdjustedSalary.Text :=
      dmDBExpress.cdsTest.Aggregates.Items[0].Value;
end;
```

最后我们再次执行上一小节的范例，那么当我们输入加薪的百分比之后，下方的TEdit控件就显示了计算出的Aggregate字段，代表所有薪资的总和（见图3-21）。

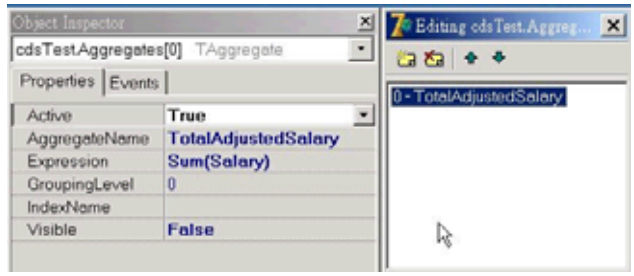

图3-20 建立Aggregate字段

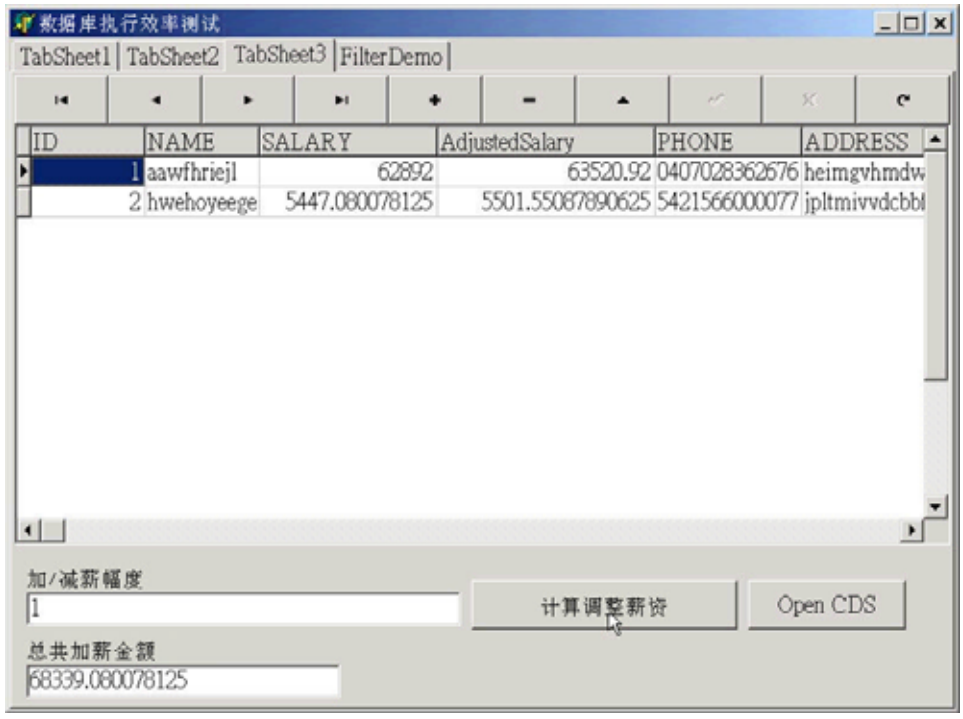

图3-21 Aggregate字段执行的状态

Aggregate字段是Delphi/Kylix提供的简易表达式功能，并且允许程序员把计算出的信息以字段的形式暂时存储在数据集中。不过程序员如果需要比较复杂的计算，

那么仍然应该使用Object Pascal程序代码或是计算字段。

3.5 UpdateStatus

TSimpleDataSet/TClientDataSet的UpdateStatus特性值让程序员能够得知目前的数据的状态。例如，这个记录是未经过修改的原始数据，或是已被修改的数据，或是已被用户删除的数据，亦或是由用户新增的数据。下面的表格列出了 UpdateStatus能够拥有的特性值以及每一个特性值代表的意义：

UpdateStatus数值	意　义
usUnmodified	目前这个记录尚未修改过
usModified	目前这个记录已经被修改过
usInserted	目前这个记录属于新增的数据
usDeleted	目前这个记录已经被删除了

程序员可以直接使用程序代码检查 TSimpleDataSet/TClientDataSet的UpdateStatus特性值来判断目前记录的状态，以便根据需要来处理数据。例如下面的程序代码可以把目前TSimpleDataSet/TClientDataSet中每一个记录的状态显示在主窗体中：

```
procedure TfrmUpdateStatus.NotifyScroll;
var
  aUS : TUpdateStatus;
begin
  aUS := dmUpdateStatus.sdsUpdateStatus.UpdateStatus;

  case aUS of
    usUnmodified    : SetUpdateStatusInfo(True, False, False, False);
    usModified      : SetUpdateStatusInfo(False, True, False, False);
    usInserted      : SetUpdateStatusInfo(False, False, True, False);
    usDeleted       : SetUpdateStatusInfo(False, False, False, True);
  end;
end;

procedure TfrmUpdateStatus.SetUpdateStatusInfo(ck1, ck2, ck3,
  ck4: Boolean);
begin
  Self.cbUnModified.Checked := ck1;
  Self.cbModified.Checked := ck2;
  Self.cbInserted.Checked := ck3;
  Self.cbDeleted.Checked := ck4;
end;

procedure TfrmUpdateStatus.BitBtn1Click(Sender: TObject);
```

```
begin
  if (dmUpdateStatus.sdsUpdateStatus.ChangeCount) then
  begin
    dmUpdateStatus.sdsUpdateStatus.ApplyUpdates(0);
    NotifyScroll;
  end;
end;
```

如果执行上面的程序代码，在用户浏览每一个记录时，这个记录的状态便会显示在下方相对应的复选框中，见图3-22。

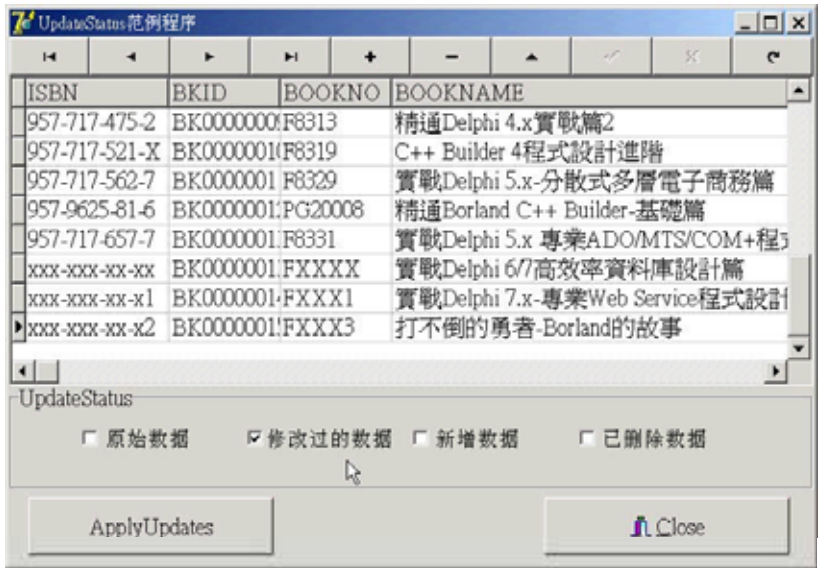

图3-22 范例程序可通过UpdateStatus了解每一个记录的状态

请读者注意，UpdateStatus特性值显示的信息是目前存在于客户端TSimpleDataSet/TClientDataSet缓存内存中的数据的状态。一旦TSimpleDataSet/TClientDataSet调用了ApplyUpdates方法把数据更新回数据源中，TSimpleDataSet/TClientDataSet便会调用MergeChangeLog把客户端的Data和Delta合并，并且清除UpdateStatus特性值。

```
procedure MergeChangeLog;
```

因此，此时TSimpleDataSet/TClientDataSet中所有数据的UpdateStatus特性值会被重置成usUnmodified。另外读者要注意的是，UpdateStatus特性值只会记录一个记录相对于原始情况的状态改变，而不会连续追踪记录的状态改变。这个意义是说，当在TSimpleDataSet/TClientDataSet中新增一个记录时，其UpdateStatus特性值被设置成usInserted；如果稍后这个记录又经过数据修改，那么UpdateStatus特性值将仍然保持usInserted，而不会同时记录这个记录经过了usInserted和usModified两个状态的改变。

3.6 SavePoint

在DataSnap提供的功能中有一个非常好的功能便是，当用户使用 TSimpleDataSet/TClientDataSet/TSQLClientDataSet在客户端修改数据时，DataSnap会记录用户对于数据的每一次修改。由于 DataSnap会记录数据修改的过程，因此 DataSnap也允许用户把数据恢复到先前的状态。提供这个功能的就是 TSimpleDataSet/TClientDataSet/TSQLClientDataSet的SavePoint特性值。

SavePoint和数据源提供的事务管理是不一样的，SavePoint特性值虽然可以允许程序员把数据的修改恢复成先前的状态，但这是对在客户端 TSimpleDataSet/TClientDataSet/TSQLClientDataSet缓存内存中的记录而言，并不是指已经通过ApplyUpdates更新回数据源的数据。

使用SavePoint特性值虽然可以恢复先前对数据的修改，但是程序员只能以向前的方式恢复数据，一旦使用 SavePoint把数据恢复成上一次的状态，那么就无法再以向后的方式恢复数据。另外，当程序代码调用了 ApplyUpdates方法之后，所有先前的SavePoint便失效了而进入重置的阶段，在随后的数据修改中 SavePoint才又开始起作用。

使用SavePoint特性值是非常容易的，程序员只要在应用程序需要的状态下存储TSimpleDataSet/TClientDataSet/TSQLClientDataSet的SavePoint特性值，接着在应用程序随后的阶段想要恢复数据状态时，再把先前存储的 SavePoint值指定回SavePoint特性值即可恢复先前的数据。

例如，下面的程序代码在 TSimpleDataSet的AfterPost事件处理函数中把 TSimpleDataSet的SavePoint特性值存储在一个 TListBox中。而在bbtnSavePointClick事件处理函数中则从TListBox中取出用户选择的先前存储的 SavePoint值，再指定给 TSimpleDataSet的SavePoint特性值，以把数据恢复成当时阶段的状态。

```
procedure TForm1.sdsBooksAfterPost(DataSet: TDataSet);
begin
  ListBox1.Items.Add(IntToStr(Self.sdsBooks.SavePoint));
end;

procedure TForm1.bbtnSavePointClick(Sender: TObject);
var
  iSavePoint : Integer;
begin
  if (ListBox1.ItemIndex <> -1) then
  begin
    iSavePoint := StrToInt(ListBox1.items[ListBox1.ItemIndex]);
    Self.sdsBooks.SavePoint := iSavePoint;
  end;
end;
```

```
procedure TForm1.bbtnApplyClick(Sender: TObject);
begin
  if (Self.sdsBooks.ChangeCount > 0) then
    Self.sdsBooks.ApplyUpdates(0);
end;
```

图3-23～图3-25是执行上面程序代码的结果。首先我们修改FXXX、FXXX1和FXXX3这3个记录的BOOKNAME字段的值，在每一次Post数据之后，SavePoint的值便会被保存在TListBox中。因此我们在图3-23中看到了三个数值：65537、65538和65539。

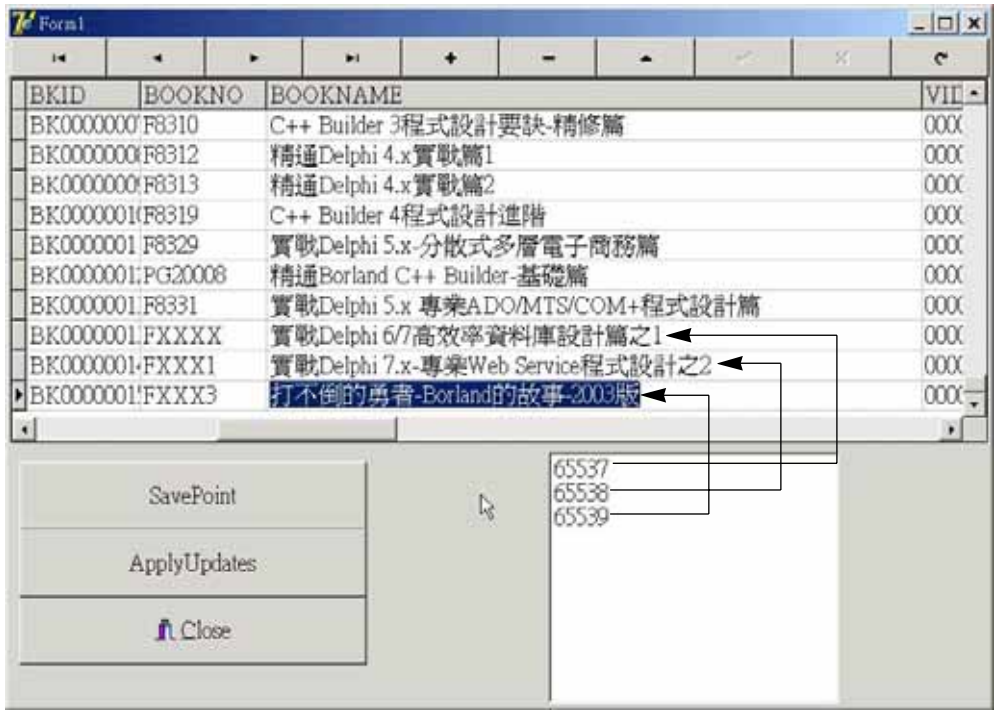

图3-23 修改3个记录并且存储了三个SavePoint值

如果我们点击TListBox中的65538，再点击窗体中的“SavePoint”按钮，那么我们便可以看到类似于图3-24的画面，此时TSimpleDataSet的数据已经恢复到存储第3个SavePoint之前的状态。我们对于FXXX3这个记录的修改已经被恢复成未修改之前的状态了。

如果我们接着又点击TListBox中的65539，再点击“SavePoint”按钮想要向后恢复数据的状态，那么DataSnap便会显示一个Invalid Parameter错误对话框，这说明了SavePoint是只能向前恢复状态，而不能向后恢复数据状态。同样，在调用了ApplyUpdates方法把客户端的所有修改数据更新回数据源之后，所有先前的SavePoint值也无效了，如果程序代码试着使用先前的SavePoint值，那么也会遇到Invalid Parameter

错误对话框（见图3-25）。

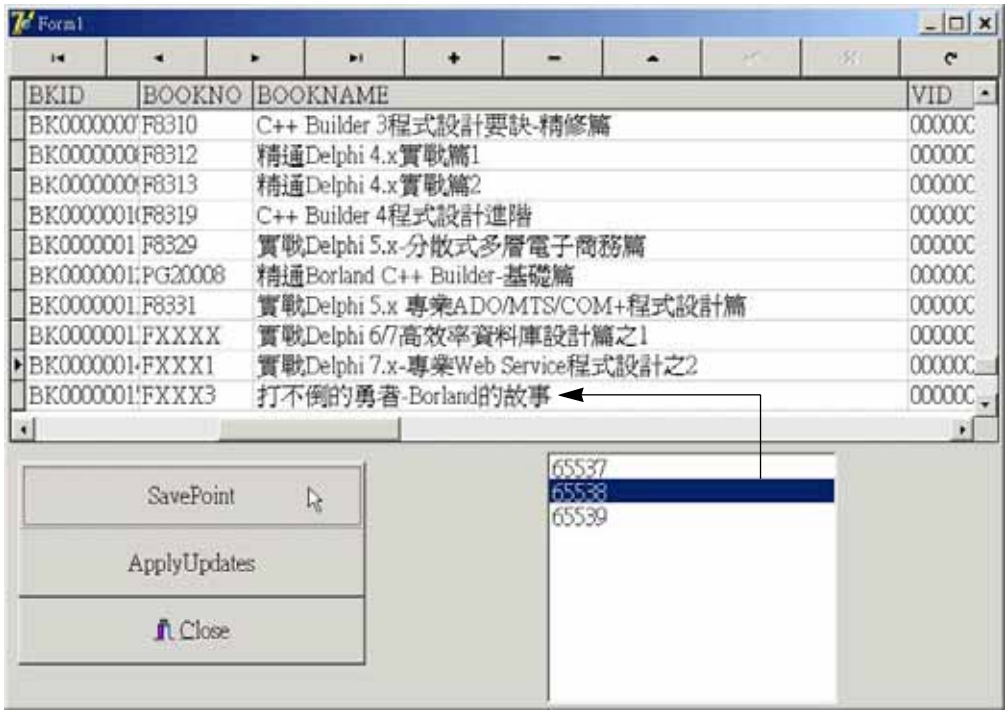

图3-24　点击第二个SavePoint值以恢复当时的数据状态

图3-25　使用向后方式恢复SavePoint或是在ApplyUpdates之后使用SavePoint都会造成Invalid Parameter错误

3.7　MyBase

Delphi的MIDAS/DataSnap技术从推出之后便支持BriefCase模式的数据处理能力。所谓BriefCase模式是指客户端应用程序在使用DataSnap技术从数据源取得了数据之后，可以暂时把数据存储在客户端的机器中，切断和数据源的连接，再继续处理数据。当客户端处理完毕之后，又可以连接数据源，再把客户端修改的数据更新回数据源中。

这种数据处理模式非常适合用于把数据带出企业，在客户端处理数据，最后再回到企业中更新在客户端处理的数据。例如，许多销售人员需要在见顾客之前先从系

统下载产品信息，到顾客处展示，修改数据，最后再回到公司把最新的数据更新回企业的系统中。

在Delphi 6/7和Kylix中，Borland又再次扩充BriefCase数据处理模式，允许程序员使用XML的格式在客户端存储数据，或是与其他系统以XML的格式交换数据，因此把这种功能称为MyBase。

在TSimpleDataSet/TClientDataSet中提供了两个方法允许程序员把数据暂时存储在文件中，以及再从文件中加载先前存储的数据，它们是SaveToFile和LoadFromFile。SaveToFile除了可以存储数据之外，也允许程序员使用不同的格式来存储数据，程序员可以将数据存储为二进制格式或是XML格式。下面是SaveToFile的原型声明：

```
SaveToFile(const FileName: string = ''; Format TDataPacketFormat=dfBinary);
```

SaveToFile的第一个参数是存储的文件名称，第二个参数则是程序员指定的存储格式，在默认情况下是存储为二进制格式。下面的表格列出了TDataPacketFormat能够拥有的值以及这些值代表的意义：

TDataPacketFormat	意　义
dfBinary	以二进制格式存储TSimpleDataSet/TClientDataSet中的数据
dfXML	以XML格式存储TSimpleDataSet/TClientDataSet中的数据
dfXMLUTF8	以XML，UTF8格式存储TSimpleDataSet/TClientDataSet中的数据

LoadFromFile方法更简单了，它只接受一个代表要加载的数据文件名的参数：

```
procedure LoadFromFile(const FileName: string = '');
```

如果程序员使用XML格式存储数据的话，那么就可以与其他应用程序交换数据。程序员可以使用Delphi 6/7提供的XML Data Binding向导来进行数据交换的工作。下面的程序代码片断显示了如何使用SaveToFile和LoadFromFile方法来存储和加载TSimpleDataSet/TClientDataSet中的数据。

```
procedure TForm1.BitBtn3Click(Sender: TObject);
begin
  if (Self.rbtnBinary.Checked) then
    Self.sdsPerformers.SaveToFile(TFILENAME)
  else
  begin
    Self.sdsPerformers.SaveToFile(TFILENAME+ '.XML', dfXMLUTF8);
  end;
end;

procedure TForm1.BitBtn4Click(Sender: TObject);
begin
  Self.sdsPerformers.Active := False;
  if (Self.rbtnBinary.Checked) then
    Self.sdsPerformers.LoadFromFile(TFILENAME)
  else
```

```
    Self.sdsPerformers.LoadFromFile (TFILENAME + '.XML');
end;
```

图3-26显示了当用户在TSimpleDataSet中修改了数据之后通过调用 SaveToFile把数据暂时存储在一个XML文件中。

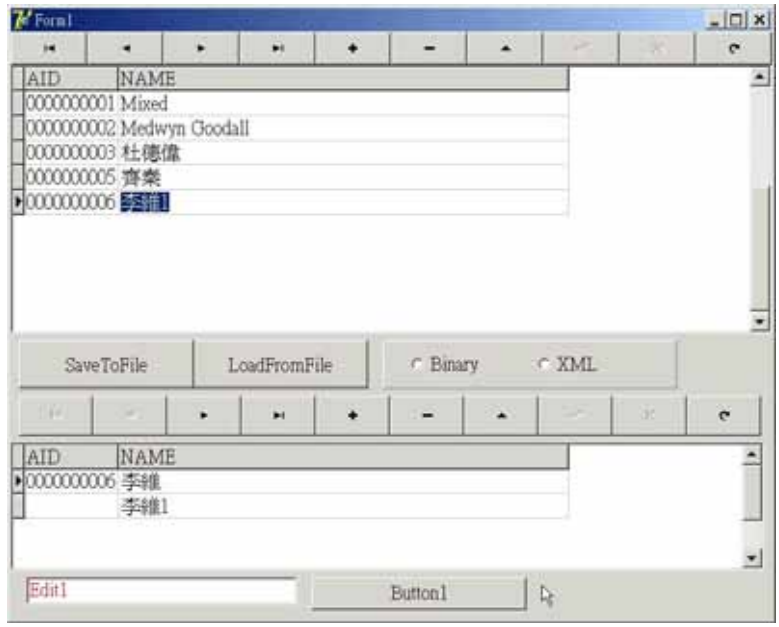

图3-26　将TSimpleDataSet中的数据存储到XML文件中

图3-27则是使用IE显示图3-26存储为XML格式的数据的画面，从图3-27中我们可

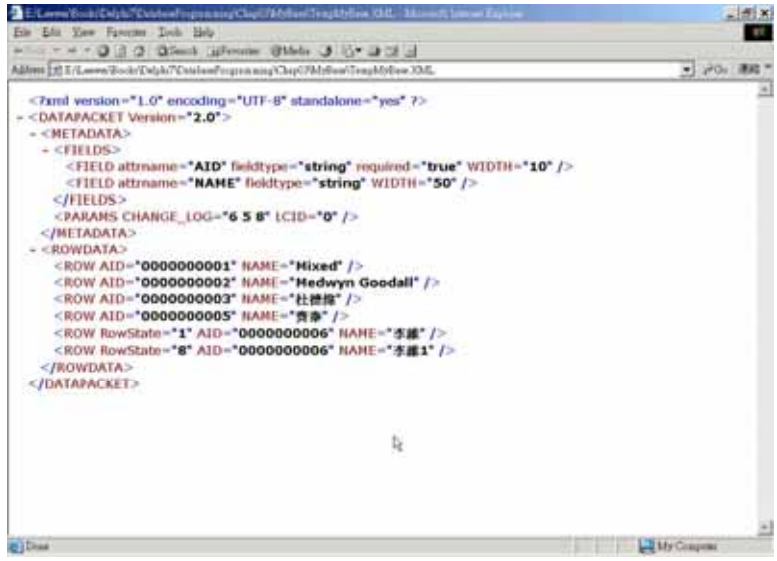

图3-27　IE显示DataSnap存储的XML格式的数据

以看到DataSnap的确是把所有数据正确地存储为 XML格式，如此一来这个文件中的数据就可以顺利地与其他应用程序交换了。

虽然Delphi6/7的DataSnap使程序员能够将数据存储到文件中，以后再将数据加载回来，但是目前 Delphi 7中的DataSnap在这方面似乎有一些小 bug，在将数据加载回来之后没有正确地处理 Delta特性值，在这稍后 Delphi 7的Patch中也许会被纠正。

3.8 TField对象的SetText和GetText事件处理函数

在许多数据库设计中，分析师都会使用键值来代表特定的信息，例如使用 ID值来代表部门代号，使用书籍号码来代表书名等。这些技巧有几个目的，其中最重要的目的通常是在数据表中维持唯一性，以便于搜寻特定的数据。其他目的则可能是为了使数据库正规化（不是所有的分析师都了解数据库正规化），也有可能是为了节省数据库存储空间等考虑因素。

虽然上述的设计是非常正确的，但是用户在使用应用程序时却可能不希望看到对于用户来说是无意义的键值代号，而是希望看到完整的信息。例如用户可能希望看到完整的书籍名称，而不是书籍代号，除非这个用户是书商，能够熟记每一个书籍代号代表的书籍，因此解决这个数据库信息转换问题的工作是许多数据库应用程序都需要处理的工作。

要解决这个问题有许多不同的方法， Delphi/Kylix的TField对象也提供了两个相关的事件处理函数 SetText以及GetText来帮助程序员解决这个问题。 GetText事件被触发的时机是当客户端需要取得字段的值时，因此对于键值的字段程序员可以在这个事件处理函数中使用键值到其他数据表中搜寻这个键值代表的完整信息，再把此完整信息作为字段的值返回即可。下面是 GetText的声明原型：

```
type TFieldGetTextEvent = procedure (Sender: TField; var Text: String;
DisplayText: Boolean) of object;
```

TFieldGetTextEvent的第二个参数 Sender是目前要取得值的字段对象，第二个参数Text是声明为var类型的字符串参数，程序员必须在这个参数中指定代表此字段的值，最后一个参数 DisplayText则是代表这个字段是用来显示的字段，还是允许用户修改的字段。

如果程序员使用 GetText事件处理函数转换字段代表的值，那么通常程序员需要搭配使用SetText事件处理函数。因为当 GetText改变字段代表的值之后，如果用户在客户端修改了这个字段的数据，那么在用户把数据更新回数据源中时，程序员必须先把数据转换回原始的形式，然后再更新到数据源中。而进行这个数据回转的好地方就是SetText事件处理函数。

SetText事件的声明原型如下：

```
type TFieldSetTextEvent=procedure(Sender:TField;const Text:String) of
object;
```

其中的第一个参数Sender代表要更新数据的字段，而Text则是客户端传递来的代表此字段的值，程序员可以将参数Text的值恢复成正确的数据，再把这个正确的数据指定给Sender的Value特性值。

现在让我们使用一个范例来说明如何使用GetText和SetText事件处理函数。在D7Books数据库中，BOOKS数据表有一个字段AID，它代表书籍的作者或是唱片的歌手ID。当我们使用数据感知组件显示BOOKS数据表时，我们并不希望看到AID的值，而是希望看到AID代表的人名。因此这个范例在使用TDBGrid显示BOOKS数据表的内容时，使用GetText和SetText事件处理函数将AID转换成人名，并且在用户修改数据时将人名回转成正确的AID值，再更新回BOOKS数据表中。

首先建立一个Delphi/Kylix项目，在数据模块中放入如图3-28所示的dbExpress组件以连接D7Books数据库。

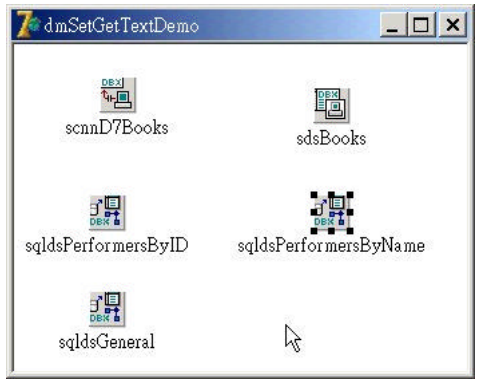

图3-28　使用dbExpress连接D7Books

TSQLConnection：

特性名称	设置特性值
Name	scnnD7Books
Database	e:\LeeWei\Books\Delphi7\Datas\D7Books.GDB

TSimpleDataSet：

特性名称	设置特性值
Name	sdsBooks
DBConnection	scnnD7Books
DataSet\CommandText	select * from BOOKS

TSQLDataSet：

特性名称	设置特性值
Name	sqldsPerformersByID
DBConnection	scnnD7Books
CommandText	select NAME from PERFORMERS where AID = :AID

TSQLDataSet：

特性名称	设置特性值
Name	sqldsPerformersByName
DBConnection	scnnD7Books
CommandText	select AID from PERFORMERS where NAME = :NAME

TSQLDataSet：

特性名称	设置特性值
Name	sqldsGeneral
DBConnection	scnnD7Books

接着激活sdsBooks的字段编辑器加入所有字段，并且使用对象检视器为 AID字段定义GetText和SetText，如图3-29所示。

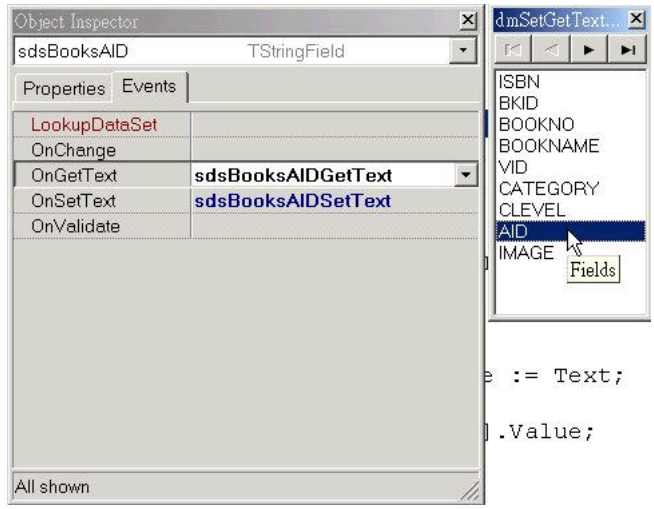

图3-29 激活sdsBooks的字段编辑器，加入所有字段，
再建立AID字段的GetText和SetText事件处理函数

现在我们希望使用 TDBGrid显示BOOKS数据表的内容，并且希望 AID字段能够显示一个下拉框，允许用户在修改此字段的数据时能够使用选择人名的方式。因此在主窗体中加入一个 TDBGrid控件，双击TDBGrid激活字段编辑器，加入所有的字

段，如图3-30所示。由于我们希望在此字段中能够让用户选择人名，因此我们只需要把人名的数据加入到AID的PickList特性值中即可。

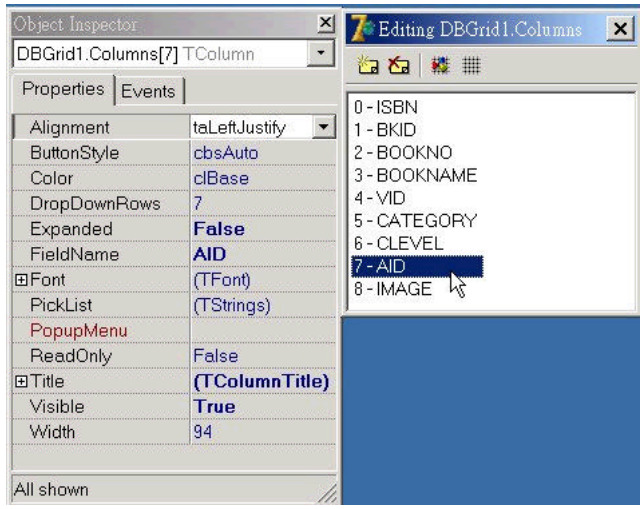

图3-30　为AID字段加入PickList特性值

首先我们实现数据模块中AID字段的GetText和SetText事件处理函数。下面是相关的实现程序代码：

```
procedure TdmSetGetTextDemo.sdsBooksAIDGetText(Sender: TField;
  var Text: String; DisplayText: Boolean);
begin
  try
    Self.sqldsPerformersByID.ParamByName('AID').Value := Sender.Value;
    Self.sqldsPerformersByID.Active := True;
    Text := Self.sqldsPerformersByID.Fields[0].AsString;
  finally
    Self.sqldsPerformersByID.Active := False;
  end;
end;

procedure TdmSetGetTextDemo.DataModuleCreate(Sender: TObject);
begin
  Self.sqldsPerformersByID.Prepared := True;
  Self.sqldsPerformersByName.Prepared := True;
end;

procedure TdmSetGetTextDemo.DataModuleDestroy(Sender: TObject);
begin
  Self.sqldsPerformersByID.Active := False;
```

```
  Self.sqldsPerformersByID.Prepared := False;
  Self.sqldsPerformersByName.Active := False;
  Self.sqldsPerformersByName.Prepared := False;

  Self.scnnD7Books.Connected := False;
end;

procedure TdmSetGetTextDemo.sdsBooksAIDSetText(Sender: TField;
  const Text: String);
begin
  try
    Self.sqldsPerformersByName.ParamByName('NAME').Value := Text;
    Self.sqldsPerformersByName.Active := True;
    Sender.Value := Self.sqldsPerformersByName.Fields[0].Value;
  finally
    Self.sqldsPerformersByName.Active := False;
  end;
end;
```

在sdsBooksAIDGetText事件处理函数中，我们根据 AID字段目前的数值到PERFORMERS数据表中搜寻代表此ID的人名信息，再把找到的人名指定给第二个参数Text。而sdsBooksAIDSetText事件处理函数则是执行相反的工作，它根据传递来的人名数据到PERFORMERS数据表中搜寻此人名的AID值，最后再指定给代表字段对象的Sender参数的Value特性值。

由于GetText和SetText事件处理函数执行得非常频繁，因此为了提高性能，我们在数据模块的OnCreate事件处理函数中先准备sqldsPerformersByID和sqldsPerformersByName组件要执行的SQL语句，并且在数据模块的OnDestroy事件处理函数中取消准备，以释放数据源中使用的资源。

最后再回到范例程序的主窗体，在主窗体中主要的工作就是在显示 TDBGrid时在AID字段的PickList中填入目前在PERFORMERS数据表中的所有人名信息。这个工作是在范例主窗体的OnShow事件处理函数中调用FillPerformerInfos函数完成的。而FillPerformerInfos函数则是使用数据模块中的 sqldsGeneral组件执行SQL语句从PERFORMERS数据表中取得所有人名数据，再填入到 TDBGrid中AID字段的PickList特性值中。

```
procedure TForm1.BitBtn2Click(Sender: TObject);
begin
  if (dmSetGetTextDemo.sdsBooks.ChangeCount > 0) then
    dmSetGetTextDemo.sdsBooks.ApplyUpdates(0);
end;

procedure TForm1.FillPerformerInfos;
```

```
begin
  if (Self.DBGrid1.Columns.Items[7].PickList.Count) then
  begin
    dmSetGetTextDemo.sqldsGeneral.Active := False;
    dmSetGetTextDemo.sqldsGeneral.CommandText := 'select distinct NAME from
    PERFORMERS';
    try
      dmSetGetTextDemo.sqldsGeneral.Active := True;
      while not dmSetGetTextDemo.sqldsGeneral.Eof do
      begin
        Self.DBGrid1.Columns.Items[7].PickList.Add (dmSetGetTextDemo.
        sqldsGeneral.Fields[0].AsString) ;
        dmSetGetTextDemo.sqldsGeneral.Next;
      end;
    finally
      dmSetGetTextDemo.sqldsGeneral.Active := False;
    end;
  end;
end;

procedure TForm1.FormShow (Sender: TObject);
begin
  FillPerformerInfos;
end;
```

最后我们编译此范例并且执行它，那么用户便可以看到类似于图 3-31的画面，请注意此时AID字段显示的不再是代表人名的 ID数值，而是完整的人名数据，而且用户可以使用下拉框来进行数据修改的工作。如果用户修改了 AID字段中的人名数据并且点击主窗体中的 ApplyUpdates按钮，那么数据模块中的 SetText事件处理函数便会被触发，它先把人名回转成正确的ID值，再更新回BOOKS数据表的AID字段中。

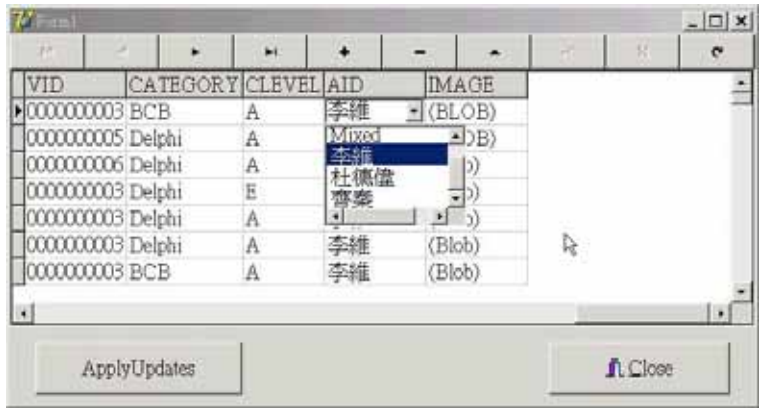

图3-31 范例程序执行的画面

当然本范例只是为了展示如何使用 GetText和SetText这两个事件处理函数。对于这个范例来说，PERFORMERS数据表中的数据并不多。对于这种需要查询的数据表而言，如果数据量不多的话，那么程序员可以使用一个 TSimpleDataSet/TClientDataSet把数据先读到客户端，再在 GetText和SetText事件处理函数中直接使用 TSimpleDataSet/TClientDataSet的Locate方法进行数据转换的工作，这可能会比较有效率。

3.9 结论

在本章中我们讨论了许多使用 dbExpress的技巧，它们包含各种数据排序技巧以及如何高效率地进行数据排序，读者在熟悉了不同的排序方法之后，就可以根据应用程序的需求选择适当的排序方式。

内存数据表是一项非常有用的技巧，特别是对于应用程序频繁使用的数据。由于内存数据表把数据暂时存储在内存中，因此应用程序在处理或是查询这些数据时会非常高效。dbExpress的结构也特别适合使用内存数据表，因为 dbExpress本来就是一个离线的数据集，数据原本就存储在客户端的内存中，此外 dbExpress也允许应用程序动态地建立数据集结构并且建立索引，可以让内存数据表更有效率。不过读者必须知道，内存数据表虽然好用，但是内存数据表并不适合处理大量的数据，例如上千个记录就不适合。如果程序员有需要频繁处理的数据并且在数百个记录左右，那么就可以考虑使用内存数据表。

在本章最后讨论了其他许多有用的技巧，包括计算字段、 SavePoint以及TField对象的事件处理函数等。这些处理数据的技巧在许多场合都非常有帮助，读者在了解了这些技巧之后，可以使用它们更方便地开发数据库应用程序。

第二部分

dbExpress进阶功能篇

第4章 搜寻数据

在前面的章节中，本书讨论了如何使用 dbExpress组件从数据源中取得应用程序需要的数据。通过 dbExpress的TSQLDataSet或是TSimpleDataSet组件，程序员可以从数据库中取得任何数据。当从数据源将数据取到客户端之后，这些数据便存储在客户端的内存中并且形成一个结果数据集（ Result Dataset）。当程序员需要在这个结果数据集中搜寻数据时，就必须使用 TClientDataSet或是TSimpleDataSet组件的搜寻方法来找到特定的数据。

TClientDataSet/TSimpleDataSet组件提供了数个不同的方法让程序员能够在结果数据集中搜寻特定的数据，这些方法包括 Locate、Lookup、过滤器、SetRange等。每一种搜寻数据的方法都有其特定的用途，程序员可以选择最适合的方法来搜寻数据。

当程序员使用 Locate、Lookup等方法搜寻数据时， TClientDataSet/TSimple DataSet组件提供了强大的功能让程序员搜寻特定的数据，程序员可以根据单一字段值来搜寻数据，也可以同时根据多个字段来搜寻数据。此外，被搜寻的字段也不必是索引字段，即使是非索引字段也一样可以使用这些方法搜寻数据。

当TClientDataSet/TSimpleDataSet组件使用Locate、Lookup方法搜寻数据时，这些方法会自动使用最有效率的方法来搜寻数据。如果程序员是以索引字段搜寻数据，那么这些方法便会自动使用索引来搜寻数据。如果搜寻的字段不是索引字段，那么 TClientDataSet/TSimpleDataSet也会根据情况使用最好的方法来搜寻数据。

本章讨论的内容就是如何使用这些方法搜寻数据，除了说明最常使用的搜寻方法之外，也会讨论在什么情况下应该使用什么方法来搜寻数据。更重要的是，本章在后半段会说明如何有效率地搜寻数据。虽然 TClientDataSet/TSimpleDataSet提供的搜寻数据方法都非常好用，但是在许多情况下这些方法也可能非常没有效率。因此程序员必须知道如何能够有效率地搜寻数据。因此在本章中也说明了许多增加搜寻数据的性能的技巧，在了解了这些搜寻技巧之后，程序员就可以非常有信心地在各种结果数据集中有效率地搜寻数据了。

4 1 搜寻数据集数据

从本小节开始，本书将以实际的范例来说明如何搜寻结果数据集中的数据，并且使用范例 InterBase数据库，你可以在本书的光盘中找到这个范例 InterBase数据库。这个范例数据表是中文化之后的 BIOLIFE数据表，它的数据表纲要如图 4 1所示。

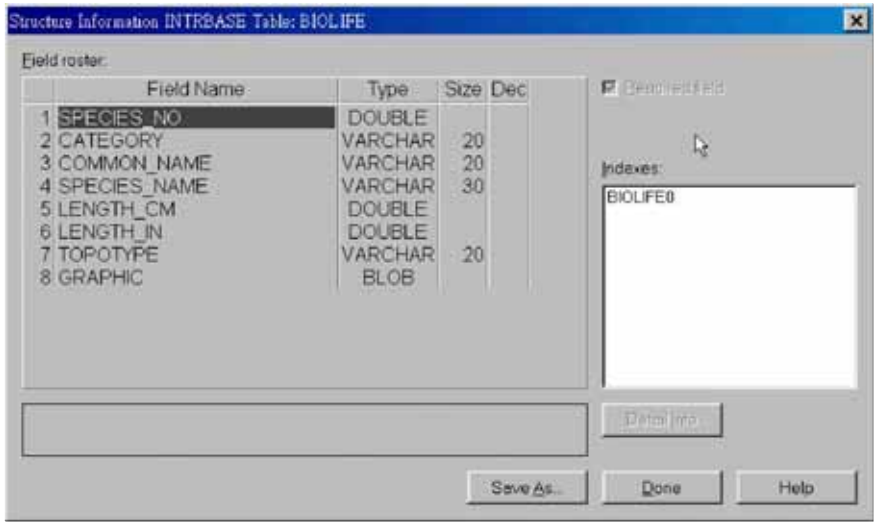

图4-1 本章使用的范例数据表纲要

首先在Delphi中建立一个新的项目，再点击 File New Data Module建立一个数据模块。在数据模块中放入 TSQLConnection组件，双击它激活组件编辑器以便将它连接到范例数据库 CHINESEDEMO.GDB。接着放入 TSimpleDataSet，设置它的DataSet\CommandText特性值为select * from BIOLIFE，此时数据模块如图4 2所示，最后设置这个数据模块的名称为 dmSearchData。

图4-2 范例应用程序使用的数据模块

现在回到主窗体，在上面放入 TDataSouce并且将它连接到数据模块上的TSimpleDataSet，再放入TDBNavigator、TDBGrid、二个TEdit组件以及三个TButton组件。这三个TButton组件将在稍后使用不同的方法搜寻数据。最后在三个 TButton组件的Caption特性值中输入Locate、Lookup和Filter。此时主窗体如图4 3所示。

现在就让我们开始讨论如何使用 dbExpress的搜寻方法。

4 1 1 Locate

TDataSet组件以及它的派生组件（例如 TSimpleDataSet/TClientDataSet等）都可以使用Locate方法在结果数据集中搜寻数据。程序首先必须使用 SQL命令从后端数据

库中取得数据并且形成结果数据集，再使用 Locate方法搜寻数据。

图4-3 范例应用程序的主窗体

当使用Locate方法搜寻数据时，程序员可以使用任何字段来搜寻，而不管这个字段是不是索引字段。当然，当程序员使用索引字段搜寻数据时， Locate会直接使用索引来帮助搜寻，因此速度会非常快。如果程序员使用非索引字段搜寻数据，那么Locate也将使用目前它知道的最好的方式来搜寻数据。

此外，Locate方法不只能够搜寻单一的字段，它还能够同时用数个字段搜寻数据。程序员可以组合数个字段的搜寻条件在结果数据集中搜寻数据。

由于Locate能够搜寻各种不同数据类型的字段，因此 Locate方法在设置搜寻条件时是以Variant类型的变量来存储搜寻值。当程序员要使用多个字段搜寻数据时必须建立一个Variant数组来存储搜寻值。

此外，Locate方法在搜寻数据时也能够使用模糊条件来寻找特定的数据，例如程序员可以要求 Locate在搜寻数据时不区分大小写，或是以部分字符串来搜寻数据，这为程序员提供了非常大的弹性。

下面就是Locate的方法原型：

```
function Locate(const KeyFields:String;const KeyValues:Variant;Options:
  TLocateOptions): Boolean;
```

Locate方法接受三个参数，第一个参数KeyFields是程序员要搜寻的字段名称。如果程序员要搜寻单一字段，那么只需要直接传入此字段名称。如果要按照多个字段

条件进行搜寻，那么程序员需要传入所有的字段名称，并且以分号分隔每一个字段名称。

第二个参数KeyValues是指程序员欲搜寻的条件值。它的类型是 Variant，因为Variant几乎可以代表任何类型，因此程序员可以搜寻整数、小数、字符串或是布尔值。同样，如果程序员只搜寻一个条件值，那么就可以直接在这个参数位置传入搜寻值。如果要按照多个字段条件进行搜寻，那么程序员必须建立一个 Variant数组，然后在这个数组中的每一个元素中指定条件值，再将这个 Variant数组传递到这个参数中。Variant数组可以使用VarArrayOf方法或是使用VarArrayCreate方法来建立，在稍后的范例中会有程序代码说明。

Locate方法的最后一个参数TLocateOptions是让程序员在搜寻字符串字段时指定以什么标准来搜寻数据。程序员可以指明以不区分大小写的方式搜寻字符串数据，或是按照部分字符串值来搜寻数据。下面就是TLocateOptions的类型定义：

```
type
  TLocateOption =(loCaseInsensitive, loPartialKey);
  TLocateOptions setof TLocateOption;
```

在使用Locate时，如果使用loCaseInsensitive就代表不区分大小写，如果使用loPartialKey就代表按照部分字符串搜寻数据。

Locate方法的返回值是一个布尔值，它代表Locate方法是否成功地找到了要搜寻的数据。如果找到的话，就返回True，否则就返回False。当Locate方法成功地搜寻到数据之后，它就会将目前的记录位置移动到这个记录上，否则就会停留在 Locate开始搜寻之前的记录位置上。

请注意，Locate方法搜寻数据的结果是一个记录，因此如果你想搜寻符合条件的多个记录，那么你可以使用稍后介绍的过滤器（Filter）功能。

现在让我们使用数个范例来说明如何使用Locate方法。下面的范例程序代码以一个字段来搜寻数据，它是以数据表的NAME字段来搜寻拥有“李维”这个值的记录，由于最后一个参数是空的，因此这代表 NAME字段必须拥有一模一样的“李维”这个值才算搜寻成功。

```
aSQLClientDataSet.Locate('NAME', 李维', []);
```

下面的程序代码则是以两个字段City和District来搜寻数据，搜寻的目标是City字段是“台北”而且District字段是“大安区”的记录。

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

下面的程序代码和第一个范例非常相像，只是这个程序代码搜寻的是 NAME字段以“李”开头的第一个记录。

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

最后一个范例则是搜寻ID字段以“A12”开头的第一个记录，而且不区分A的大小写。

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

现在就让我们使用Locate方法在范例应用程序中搜寻数据。

1. 单字段搜寻

现在就让我们使用Locate方法在范例应用程序中搜寻数据，先让我们以单一字段来展示如何搜寻数据，稍后再说明如何以多个字段搜寻数据。现在请双击图 4 2中的Locate按钮，并且在它的事件处理函数中编写如下的程序代码：

```
dmSearchData.sqlcdsTest.Locate('SPECIES_NO',edtID.Text,[loCaseInsensitive,
loPartialKey]);
```

这行程序代码使用数据模块中的TSimpleDataSet在SPECIES NO字段中搜寻用户在TEdit控件edtID中输入的数值。现在请执行这个范例应用程序，并且在主窗体中右边的TEdit控件中输入数值来搜寻数据。例如，图 4 4便是范例应用程序执行的画面。当笔者在TEdit控件中输入90100并且点击Locate按钮之后，TSimpleDataSet便会立刻找到这个记录并且把目前的记录位置移动到这个记录上。

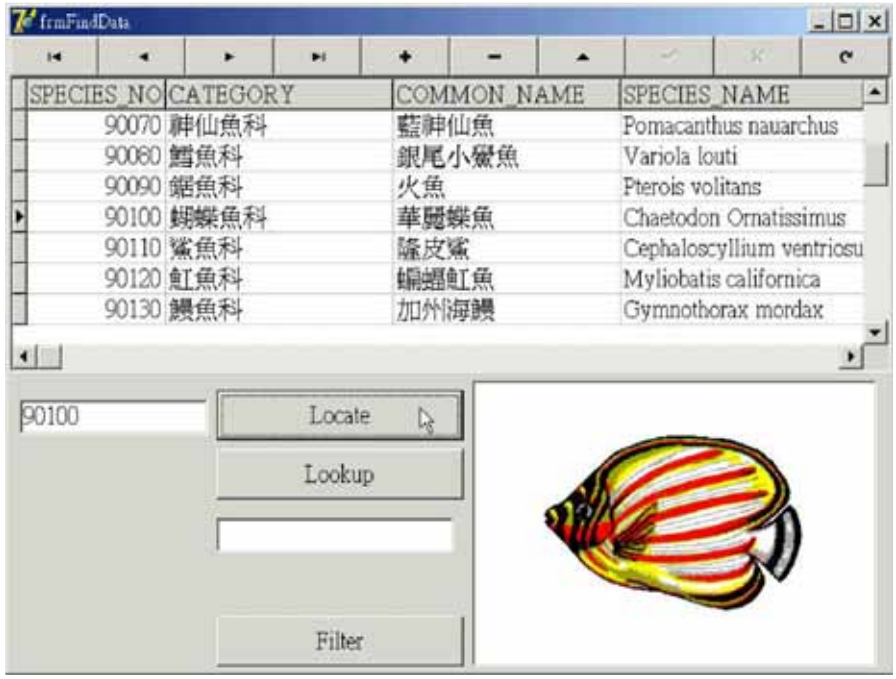

图4-4 Locate找到90100这个记录

使用Locate方法搜寻单一字段的数据是非常简单的，现在再让我们看看如何使用多个字段来搜寻数据。

2. 多字段搜寻

在Delphi/Kylix中建立一个应用程序，再和刚才的范例一样建立一个数据模块，并且放入 TSQLConnection和TSimpleDataSet，连接到相同的范例数据库CHINESEDEMO.GDB。接着在主窗体中放入图 4 5所示的控件。在主窗体中我们使用了一个TComboBox，在这个TComboBox中将会填入范例数据表的所有字段名称以便让用户可以自由选择要用来搜寻的字段。

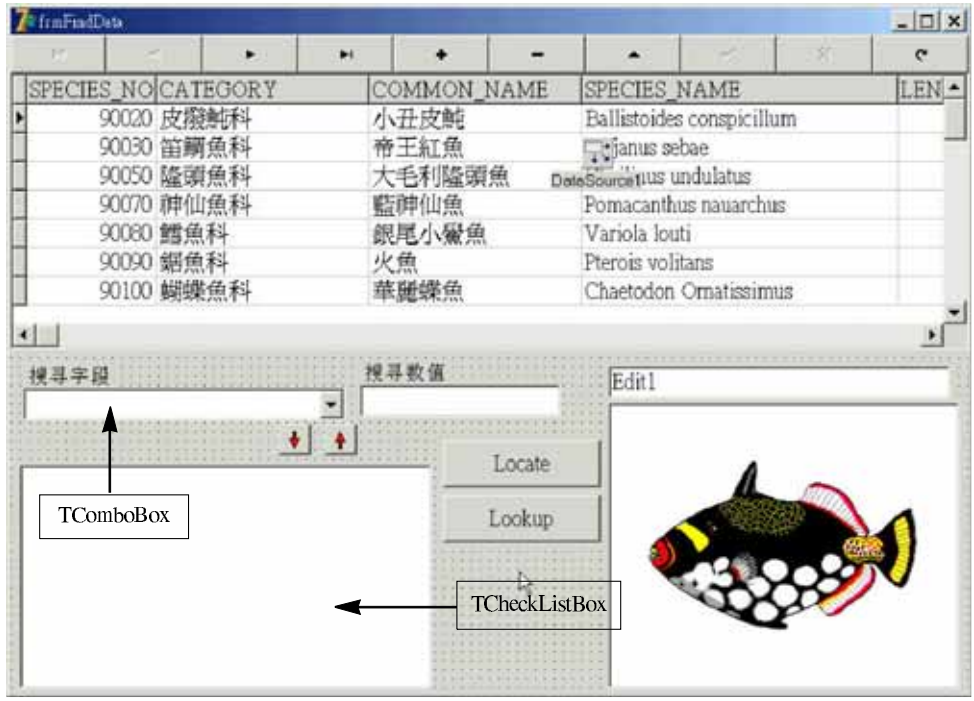

图4-5 范例应用程序的主窗体

主窗体另外使用了一个 TCheckListBox，它主要用来存储用户输入的所有搜寻条件。用户在搜寻字段中选择了一个字段，然后在搜寻值控件中输入欲搜寻的数值，那么就可以点击主窗体中向下箭头按钮把这个搜寻条件加入到 TCheckListBox中。当用户输入完所有搜寻条件之后，就可以点击主窗体中的 Locate按钮开始搜寻数据。此时范例应用程序就会分析 TCheckListBox中的所有搜寻字段和搜寻数值，再调用Locate按照多个字段条件来搜寻数据。

图4 6就是这个范例应用程序执行的画面。当范例应用程序执行后，用户可以在TComboBox中选择欲搜寻的字段，接着可以在搜寻值控件中输入欲搜寻的数值，接着点击向下箭头按钮将搜寻条件加入到 TCheckListBox中，或是点击向上箭头按钮清除某一个搜寻条件。

在TCheckListBox中的搜寻条件是以：

搜寻字段名称\搜寻字段值

的格式存储的。当用户点击了Locate按钮之后，就会从TCheckListBox中一一取出搜寻条件并且分析出搜寻字段名称以及搜寻字段值，再放入到 Locate方法的第一个和第二个参数中。

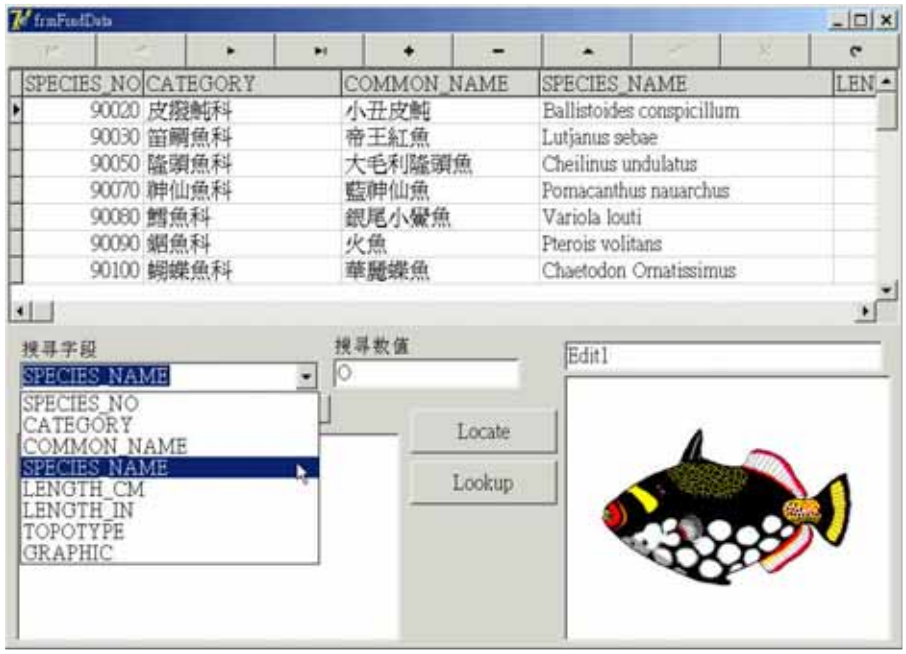

图4-6 执行范例应用程序的画面

最后，当输入完所有的搜寻条件之后，用户就可以点击主窗体中的 Locate按钮来搜寻数据了。例如，图 4 7便是搜寻 TOPOTYPE字段包含“台湾沿海”而且SPECIES NAME字段以字母O开头的记录。在点击了 Locate按钮之后，范例应用程序调用Locate方法并且以多个字段为搜寻条件，果然立刻找到了这个记录。

这个范例应用程序是如何运作的呢？这个范例应用程序基本上执行了下列的工作：

1) 在程序激活时在TComboBox中填入范例数据表中的所有字段名称。

2) 在点击向下箭头按钮时把搜寻字段和搜寻值加入到 TCheckListBox中，以及在点击向上箭头按钮时清除搜寻条件。

3) 在点击Locate按钮时从TCheckListBox中取出搜寻条件并且填入Locate方法的参数中，搜寻数据。

现在就让我们实现以上的工作。首先在范例应用程序激活时，访问数据模块中TSimpleDataSet的Field对象的 FieldName特性值以取得字段名称，再填入TComboBox中：

```
procedure TForm1.FormActivate(Sender: TObject);
var
  iField : Integer;
begin
  for iField := 0 to dmFindData.sqlcdsTest.FieldCount - 1 do
  begin
    cbFields.Items.Add(dmSearchData.sqlcdsTest.Fields[iField].FieldName)
  end;
  cbFields.ItemIndex := 0;
end;
```

当点击向下箭头按钮时，取出用户在 TComboBox中选择的字段名称以及在搜寻值控件中输入的搜寻值，再检查 TCheckListBox中是否已经存在了这个搜寻字段。如果没有的话，就把字段名称和字段值加入到 TCheckListBox中。

图4-7 以数个字段条件来搜寻数据

此外当点击向上箭头按钮时，我们就删除 TCheckListBox中目前被选择的搜寻条件。

```
procedure TForm1.sbtnAddClick(Sender: TObject);
begin
  if (not AlreadyInCond(cbFields.Text)) then
  begin
    clbConditions.Items.Add(cbFields.Text + '\' + lblSearchValue.Text
```

```
    clbConditions.Checked[clbConditions.Count - 1] := True;
  end;
end;

procedureTForm1.sbtnDeleteClick(Sender: TObject);
begin
  try
    clbConditions.Items.Delete(clbConditions.ItemIndex);
  except
    on Exception do;
  end;
end;
```

最后当用户点击了主窗体中的Locate按钮时，范例应用程序就先检查用户是否输入了任何搜寻条件。如果有的话，就调用 GetSerchFields从TCheckListBox中取出所有搜寻字段名称，再调用 GetSearchValues取得用户输入的所有搜寻值，最后调用Locate方法来搜寻数据。

其中的GetSearchValues会先调用VarArrayCreate方法建立一个Variant数组，再在这个Variant数组中一一输入用户指定的搜寻值。

```
procedureTForm1.btnLocateClick(Sender: TObject);
var
  sFields : String;
begin
  lStart := GetTickCount;
  if (CanSearch) then
  begin
    sFields := GetSerchFields;
    dmFindData.sqlcdsTest.Locate(sFields,GetSearchValues[,loCaseInsensitive,
    loPartialKey]);
  end;
  lEnd := GetTickCount;

  Self.Caption := FloatToStr((lEnd - lStart)/1000.0);
end;

functionTForm1.GetSearchValues : Variant;
var
  iCount : Integer;
  sCond : String;
begin
  Result:=VarArrayCreate([0,Self.clbConditions.Items.Count,-1]
  varVariant);;
```

```
  for iCount := 0 to Self.clbConditions.Items.Count - 1 do
  begin
    sCond := Self.clbConditions.Items[iCount];
    Result[iCount] := GetSearchValue(sCond);
  end;
end;

function TForm1.GetSearchFields: String;
var
  iCount : Integer;
  sCond : String;
begin
  Result := '';
  for iCount := 0 to Self.clbConditions.Items.Count - 1 do
  begin
    sCond := Self.clbConditions.Items[iCount];
    Result := Result + GetSearchField(sCond) + ';';
  end;
  Delete(Result, Length(Result), 1);
end;

function TForm1.CanSearch: Boolean;
begin
  Result := Self.clbConditions.Items.Count > 0;
end;

function TForm1.GetSearchField(const sCond: String): String;
var
  iPos : Integer;
begin
  iPos := Pos('\', sCond);
  Result := Copy(sCond, 1, iPos - 1);
end;

function TForm1.GetSearchValue(const sCond: String): String;
var
  iPos : Integer;
begin
  iPos := Pos('\', sCond);
  Result := Copy(sCond, iPos + 1, Length(sCond) - iPos);
end;
```

上面的范例展示了如何使用 Locate方法按照多个字段条件来搜寻数据。由于Locate方法的第二个参数是Variant类型的，因此我们可以搜寻几乎任何类型的字段。

Locate方法非常适合在所有数据已经存在于结果数据集中的应用程序使用，但是对于拥有大量记录的数据表而言却不见得适合，在稍后的小节中本章会继续讨论如何使用Locate在大量数据中搜寻数据。

4.1.2 Lookup

Lookup方法和上一小节介绍的 Locate方法在使用上非常类似，它们的差别是当Locate找到要搜寻的数据后，它会把目前的记录位置移动到找到的这个记录上；但是当Lookup找到要搜寻的数据后，它会返回找到的记录的特定字段值，却不会移动目前的记录位置。下面即是Lookup方法的原型：

```
function Lookup(const KeyFields: String;const KeyValues:Variant;const
ResultFields: String): Variant;
```

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

Lookup方法的第三个参数则是指定当 Lookup找到欲搜寻的数据之后，要返回这个记录的哪些字段值。如果程序员想让 Lookup返回多个字段值，那么每一个字段也是以分号分隔。

Lookup方法返回的数值就是第三个参数指定的字段值。如果 Lookup返回多个字段的话，那么这个返回值就是一个 Variant数组，每一个返回的字段存储在这个Variant数组的元素中。

现在就让我们看看如何使用 Lookup方法搜寻数据，首先先以简单的单一字段来搜寻数据。

1. 单字段搜寻

请使用鼠标双击图4.5中的Lookup按钮，并且在它的事件处理函数中编写如下的程序代码：

```
try
  edtReturn.Text :=
    dmFindData.sqlcdsTest.Lookup('SPECIES_NO', edtID.Text, 'COMMON_NAME');
except
  on e : Exception do
    ShowMessage(E.Message);
end;
```

上面的程序代码用Lookup方法搜寻BIOLIFE数据表中的SPECIES NO字段，并且以用户在TEdit控件edtID中输入的数值作为搜寻的目标，要求Lookup方法在搜寻到数据之后返回COMMON NAME字段的值，并且把返回值显示在TEdit控件edtReturn中。

图4 8便是执行范例应用程序并且输入90100搜寻数据的画面。从画面中可以看到当Lookup找到90100这个记录之后，它果然会返回 COMMON NAME字段的值“华丽蝶鱼”，但是目前记录的位置仍然停留在原先的记录上，而不会移动到 90100这个记录上。

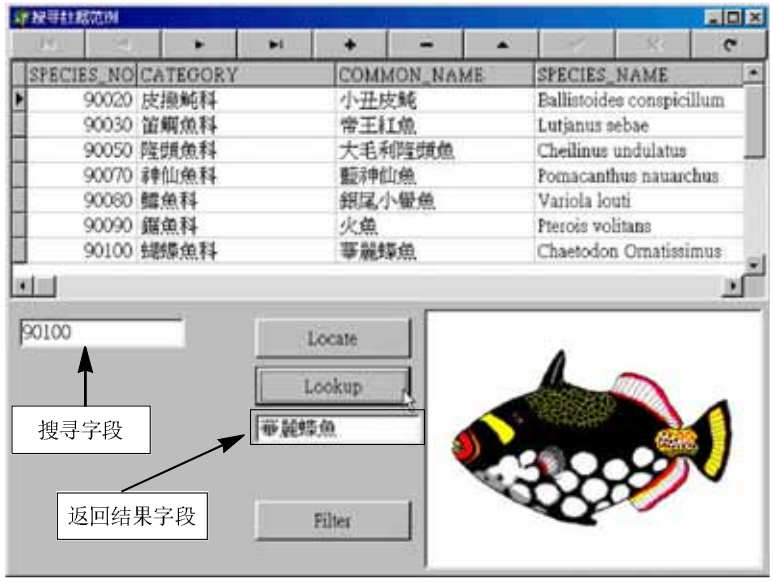

图4-8 执行范例应用程序的画面

使用Lookup搜寻并且返回单一字段的数据是非常简单的，再让我们看看如何搜寻和返回多个字段的数据。

2. 多字段搜寻

请将上一小节的Lookup按钮的事件处理函数修改为如下的程序代码：

```
procedureTForm1.btnLookUpClick(Sender: TObject);
var
  sFields : String;
  vResult : Variant;
  iCount : Integer;
begin
  lStart := GetTickCount;
  if (CanSearch) then
  begin
    sFields := GetSearchFields;

    vResult:=dmFindData.sqlcdsTest.Lookup(sFields,GetSearchValues,edtResult.Text);

    if (VarIsArray(vResult)) then
    begin
```

```
        sFields := '';
        for iCount:=VarArrayLowBound(vResult,1)to VarArrayHighBound(vResult,1) do
        begin
          sFields := sFields + ';' + vResult[iCount];
        end;
        Delete(sFields, 1, )1;
        edtResult.Text := sFields;
      end
      else
        edtResult.Text := vResult;
    end;
    lEnd := GetTickCount;

    Self.Caption := FloatToStr(( lEnd - lStart )/1000.0) ;
  end;
```

上面的程序代码先调用GetSearchValues从主窗体中的TCheckListBox中取得用户欲搜寻的所有字段和值，再从edtResult中取得用户欲取得的字段返回值，然后调用Lookup方法。由于用户可以在edtResult中输入多个字段，因此当Lookup执行完毕之后，我们先检查返回的是否是一个 Variant数组。如果是的话，就进入一个循环，调用VarArrayLowBound以及VarArrayHighBound取得数组中实际的元素值范围，然后再一一从数组元素中取出 Lookup返回的字段值，最后再把这些返回的数值显示在edtResult控件中。

图4 9到图4 11便是执行范例应用程序的画面，图 4 9是用新修改过的程序代码搜寻并且返回一个字段数据的画面。而图4 10是在TOPOTYPE字段中搜寻“台湾沿海”，并且要求返回 COMMON NAME和CATEGORY两个字段的值。图 4 11则是点击了Lookup按钮之后看到应用程序果然返回了 COMMON NAME和CATEGORY这两个字段的值。

请注意，Lookup并不像Locate那样可以使用模糊搜寻，Lookup只能搜寻到完全一样的字段值。因此如果读者使用部分字符串来搜寻数据的话，那么 Lookup不会找到数据，而会产生一个搜寻错误的异常。

4 1 3 过滤器

除了Locate和Lookup之外，事实上Delphi/Kylix的过滤器（Filter）功能是非常有用的，因为过滤器不但能够搜寻数据，更棒的是它可以再把结果数据集中的数据分门别类，让应用程序只访问特定的数据组。这个行为相当于对从后端数据库返回的结果数据集中的数据再使用额外的条件，以此取得子结果数据集。