

管理时间标记，其中包括像DateTimeToSQLTimeStamp、SQLTimeStampToStr和VarSQLTimeStampCreate这样的函数。

一些dbExpress演示

让我们看一下实际的示例，强调这些组件的关键特征，并显示怎样使用ClientDataSet对单向数据集提供缓存和编辑支持。稍后，我们将提供一个示例，以展示单向查询的使用，其中不包括缓存和编辑支持请求。

基于dbExpress的标准可视应用程序使用下列查询组件：

- **SQLConnection**组件，提供与数据库的连接和适当的dbExpress驱动程序。
- **SQLDataSet**组件，它与连接挂钩（通过SQLConnection属性），说明执行哪个SQL查询或打开哪个表格（使用以前介绍的CommandType和CommandText属性）。
- **DataSetProvider**组件，连接到数据集，从SQLDataSet抽取数据并产生适当的SQL更新语句。
- **ClientDataSet**组件，从数据提供者处读取数据并在内存存储所有数据（如果它的PacketRecords属性被设置为-1的话）。该组件有许多额外特征并具有完全的导航和编辑能力，提供真实数据到应用程序。最后我们需要调用它的ApplyUpdates方法，将实际更新发送回数据库服务器（通过提供者）。
- **DataSource**组件，允许我们从ClientDataSet到可视数据敏感控件展示数据。

如前所述，使用SQLClientDataSet能使问题得到简化，它替换了两个数据集和提供者（或者是连接）。SQLClientDataSet将它代替的组件的多数属性组合在一起。

使用单一组件或者多个组件

对于第一个范例，在窗体上放置一个SimpleDataSet组件并且设置它的Connection子组件的连接名称。设置CommandType和CommandText属性，用以指定要取得的数据，并且设置PacketRecords属性来指明在每一个块中要取回多少个记录。

这些是DbxSingle范例中组件的关键属性：

```
object SimpleDataSet1: TSimpleDataSet
    Connection.ConnectionName = 'IBLocal'
    Connection.LoginPrompt = False
    DataSet.CommandText = 'EMPLOYEE'
    DataSet.CommandType = ctTable
end
```

作为另一种办法，DbxMulti范例使用了这些组件的全部序列：

```
object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
```

```

    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
end
object ClientDataSet1: TClientDataSet
    ProviderName = 'DataSetProvider1'
end
object DataSource1: TDataSource
    DataSet = ClientDataSet1
end

```

两个范例都包括有一些可视控件：一个网格和一个基于操作管理器（action manager）结构的工具栏。

应用更新

对于每个基于本地缓存的示例（如ClientDataSet和SQLClientDataSet组件所提供的）来说，什么才是最重要的呢？就是将本地变化写回到数据库服务器。通过调用ApplyUpdates方法，可完成这一过程。但什么时候调用它呢？我们既可以在本地缓存上将变化保存一段时间，然后一次性地应用多个更新，也可以立即传递每个变化。在这两个简单示例中，我使用后一种方法，将下面的事件处理器连接到ClientDataSet组件的AfterPost（在一个编辑或者插入操作发生时激活）和AfterDelete事件上：

```

procedure TForm1.DoUpdate(DataSet: TDataSet);
begin
    // immediately apply local changes to the database
    SQLClientDataSet1.ApplyUpdates(0);
end;

```

如果想在单个批处理中应用所有更新，则可以在窗体关闭或程序结束时更新，或通过选择指定命令让用户更新操作，通常可能是使用Delphi 7中预定义的相关操作。在稍后的章节中，当详细讨论ClientDataSet组件的缓存更新支持时，我们将探讨一些这样的可选项。

监控连接

添加到DbxSingle和DbxMulti示例的另一个特性是SQLMonitor组件提供的监控能力。在示例中，当程序开始时该组件即被激活。在DbxSingle范例中，因为SimpleDataSet嵌入了连接，所以监控器不能够在设计时与它连接，而只是当程序启动的时候才可以：

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    SQLMonitor1.SQLConnection := SimpleDataSet1.Connection;
    SQLMonitor1.Active := True;
    SimpleDataSet1.Active := True;
end;

```

每次有可用的追踪字符串时，组件都会激活OnTrace事件，让我们选择是否在记录中包含字符串。如果该事件的LogTrace参数是True（默认值），则组件在TraceList字符串列表中记录消息并激活OnLogTrace事件，以此说明一个新字符串已被添加到记录中。

通过它的FileName属性，该组件也能自动存储记录到文件，但我在该示例中没有使用这一特性，而是处理OnLogTrace事件，用下列代码添加最新消息到备注组件（产生如图14.6所示的输出）：

```
procedure TForm1.SQLMonitor1Trace(Sender: TObject;
  CBIInfo: pSQLTRACEDesc; var LogTrace: Boolean);
begin
  Memol.Lines := SQLMonitor1.TraceList;
end;
```

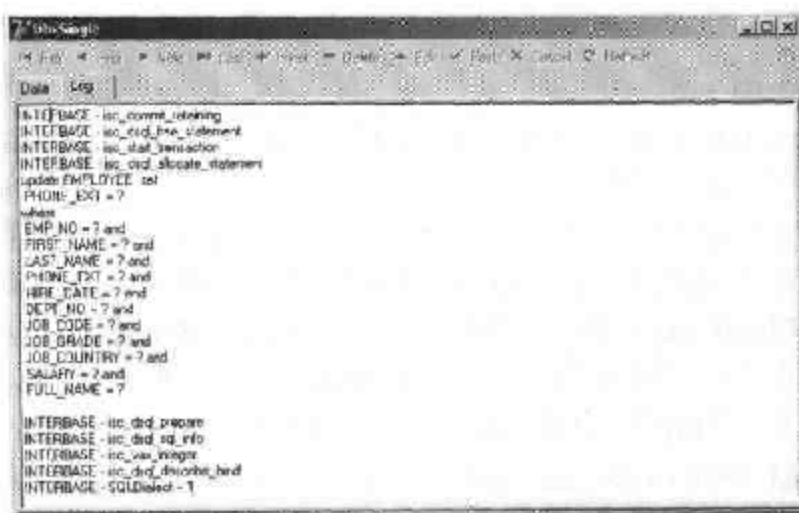


图14.6 在DbxSingle范例中，通过SQLMonitor获得的范例日志

控制SQL更新代码

如果我们运行DbxSingle程序并进行一些更改，如员工的电话号码，则监控器将记录该更新操作：

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  FIRST_NAME = ? and
  LAST_NAME = ? and
  PHONE_EXT = ? and
  HIRE_DATE = ? and
  DEPT_NO = ? and
  JOB_CODE = ? and
  JOB_GRADE = ? and
  JOB_COUNTRY = ? and
  SALARY = ? and
  FULL_NAME = ?
```

通过设定SimpleDataSet的属性，没有办法改变更新代码的产生方式，而且可能会比使用SQLClientDataSet组件的情况还要差，因为后者具有UpdateMode方法，我们可以用它来修改更新的命令。

在DbxMulti范例中，我们能够使用DataSetProvider组件的UpdateMode属性，设定其值为upWhereChanged或者upWhereKeyOnly。在这里，我们将分别获得下面两个语句：

```
update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ? and
  PHONE_EXT = ?

update EMPLOYEE set
  PHONE_EXT = ?
where
  EMP_NO = ?
```

提示：这个结果比在Delphi 6（未打补丁）中的更好，在这里，这个操作将会引发一个错误，因为key字段没有被正确的设定。

如果我们希望对于如何产生更新语句有更全面的控制，则需要对潜在的数据集的字进行操作，它在我们使用合二为一的SimpleDataSet组件（它具有两个字段编辑器，一个用于它所继承的基ClientDataSet组件，一个用于它嵌入的SQLDataSet组件）时也是可用的。我对DbxMulti范例也做了类似的更正，包括为SQLDataSet组件添加持续字段并且修改某些字段的提供者选项，用以在key中包含它们或者从更新中排除它们。

说明：当我们详细讨论ClientDataSet组件、提供者、分解器和本章后面及第16章介绍的其他技术细节时，我们将再次讨论该类型的问题。

用SetSchemaInfo访问数据库元数据

所有的RDBMS系统均使用特殊目的的表格存储元数据，如表格的列表、它们的字段、索引、限制和其他系统信息。正如dbExpress提供了一个统一的API用于不同SQL服务器上工作一样，它也提供了一个普通的访问元数据的方法。TSQLDataSet组件有一个SetSchemaInfo方法，它能够用系统信息填充数据集。该SetSchemaInfo方法有三个参数：

SchemaType 说明请求的信息类型并包含stTables、stSysTables、stProcedures、stColumns和stProcedureParams。

SchemaObject 说明我们引用的对象，如我们正请求的栏所在表格的名字。

SchemaPattern 是一个过滤器，以便我们能限制对表格、栏或用特定字母开始的过程的请求。如果我们使用前缀确定元素的组，这就变得十分简便。

例如，在SchemaTest示例中，用Tables按钮读入连接的数据库的所有表格数据集：

```
ClientDataSet1.Close;
SQLDataSet1.SetSchemaInfo (stTables, '', '');
ClientDataSet1.Open;
```


该程序使用数据集提供者的常用组、客户数据集和数据源组件在网格中显示数据结果，如图14.7所示。检索表格后，我们能挑选网格的一行并且单击Field按钮来察看该表格字段的列表：

```
SQLDataSet1.SetSchemaInfo (stColumns, ClientDataSet1['Table_Name'], '');
ClientDataSet1.Close;
ClientDataSet1.Open;
```

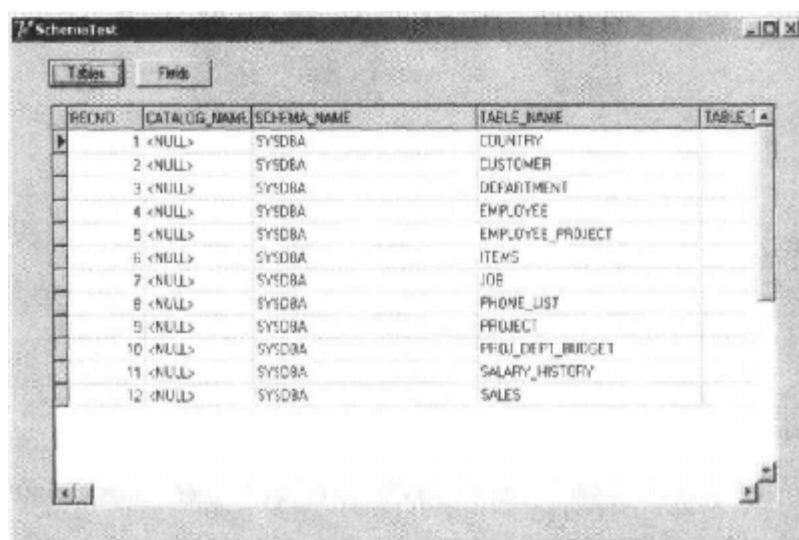


图14.7 SchemaTest范例允许我们察看数据库的表格以及给定表格的列

除了能使我们访问数据库元数据外，dbExpress还提供了一种方法，用于访问它自己的配置信息，包括安装的驱动程序和配置的连接。dbConnAdmin单元为该目的定义了TConnectionAdmin类，但是这种支持被开发者的dbExpress附加工具所限制，因为最终用户通常不能以完全动态的方式访问多个数据库。

提示：包含在Delphi中的DbxExplorer演示程序展示了怎样访问dbExpress管理文件和模式信息。请查看“*The structure of metadata datasets*”下的帮助文件，在“*Developing database applications*”一节内。

参数查询

当我们需要同一个SQL查询的稍微不同的版本时，没有必要在每一次查询时都更改它的文本，而是可以编写一个带有参数的查询，并且更改参数的数值。例如，如果我们决定要让用户选择一个给定国家内的雇员（使用employee表格），则可以编写如下的参数查询：

```
select *
from employee
where job_country = :country
```

在这个SQL子句中，“:country”是一个参数。我们可以使用SQLDataSet组件的Params属性集合的编辑器来设置它的数据类型和起始数值。当Params属性集合编辑器打开后，如图14.8所示，我们看到一个参数的列表，定义在SQL语句中。可以使用对象检验器设定这些参数的数据类型和初始值。

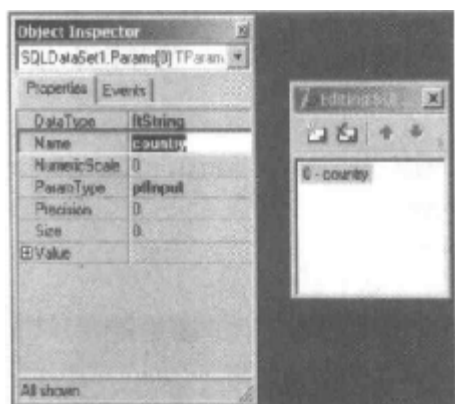


图14.8 编辑一个查询组件的参数集合

由ParQuery程序显示的窗体使用一个组合框提供所有参数的可用值。我们并未在设计时就已准备好这个组合框的条目，而是在程序开始运行后才从同一个数据库表格的可用内容中进行提取。这是使用一个二级查询组件来完成的，使用了下面的SQL语句：

```
select distinct job_country
from employee
```

在激活这个查询之后，程序搜索它的结果集合，提取所有的值并且将它们添加到列表框中：

```
procedure TQueryForm.FormCreate(Sender: TObject);
begin
    SqlDataSet2.Open;
    while not SqlDataSet2.EOF do
    begin
        ComboBox1.Items.Add (SqlDataSet2.Fields [0].AsString);
        SqlDataSet2.Next;
    end;
    ComboBox1.Text := ComboBox1.Items[0];
end;
```

用户可以在组合框中选择不同的条目，然后单击Select按钮（Button1）来改变参数和激活这个查询：

```
procedure TQueryForm.Button1Click(Sender: TObject);
begin
    SqlDataSet1.Close;
    ClientDataSet1.Close;
    Query1.Params[0].Value := ListBox1.Items [ListBox1.ItemIndex];
    SqlDataSet1.Open;
    ClientDataSet1.Open;
end;
```

这段代码在DBGrid中显示了选定国家的雇员，如图14.9所示。还有另外一种办法，就是按照位置来使用Params数组的元素，我们应该考虑使用ParamByName方法，以避免在查询被

屡次修改后以及参数顺序打乱后会产生错误。

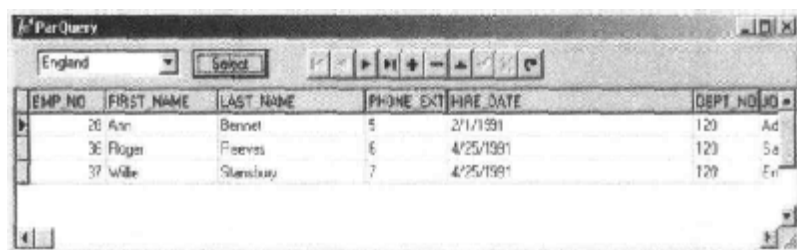


图14.9 运行时的ParQuery程序范例

通过使用参数查询，我们通常可以减少在网络中移动数据的数量，并且仍然使用一个DBGrid和在本地数据库应用程序中常用的标准用户接口。

提示：参数查询通常也用于使用SQL查询获得主/详结构。至少，Delphi倾向于这么做。SQLDataSet组件的DataSource属性会自动使用主数据集中与参数同名的字段来替换参数的值。

单向就足够了：打印数据

我们已经看到，dbExpress库的一个关键元素是它返回单向数据集。此外，我们能使用ClientDataSet组件在本地缓存上存储数据集。现在我们通过简单的示例来看看在什么地方需要单向数据集。

为每条记录按顺序产生信息而不需要更深入地访问数据，这在报告中很普通。它包括生成打印报告（通过一系列报告组件或直接使用打印），发送数据到其他应用程序（如Microsoft Excel或Word），保存数据到文件（包括HTML和XML格式）等等。

此时我不想深入讨论HTML和XML，而是要提供打印示例——没有什么奇特之处，也不基于报告组件，仅仅是一种简单地在监视器和打印机上生成草稿报告的方法。由于该原因，我继续使用Delphi的最简单的技术生成一个打印输出：使用AssignPrt RTL过程分配文件到打印机。

示例叫做UniPrint，它有一个单向SQLDataSet组件，基于下列SQL语句依附于InterBase连接，用部门表格连接员工表格，以显示每个员工工作在哪个部门：

```
select d.DEPARTMENT, e.FULL_NAME, e.JOB_COUNTRY, e.HIRE_DATE
from EMPLOYEE e
inner join DEPARTMENT d on d.DEPT_NO = e.DEPT_NO
```

为了处理打印，我编写了一些基本例程：以参数方式请求打印数据，用于状态信息的进程条，输出字体和每个字段的最大格式尺寸。下面显示的全部例程使用文件打印支持，并以每个字段固定尺寸、左对齐字符串的格式生成分栏类型的报告。对Format函数的调用有一个参数格式字符串，它是使用字段尺寸动态建立的。

在程序清单14.1中，我们能看到核心PrintOutDataSet方法的代码，它使用三层嵌套的try/finally块来正确释放所有的资源。

程序清单14.1 UniPrint范例的核心方法

```

procedure PrintOutDataSet (data: TDataSet;
  progress: TProgressBar; Font: TFont; toFile: Boolean; maxSize: Integer = 30);
var
  PrintFile: TextFile;
  I: Integer;
  sizeStr: string;
  oldFont: TFontRecall;
begin
  // assign the output to a printer or a file
  if toFile then
    begin
      SelectDirectory ('Choose a folder', '', strDir);
      AssignFile (PrintFile,
        IncludeTrailingPathDelimiter(strDir) + 'output.txt');
    end
  else
    AssignPrn (PrintFile);
  // assign the printer to a file
  AssignPrn (PrintFile);
  Rewrite (PrintFile);

  // set the font and keep the original one
  oldFont := TFontRecall.Create (Printer.Canvas.Font);
  try
    Printer.Canvas.Font := Font;
    try
      data.Open;
      try
        // print header (field names) in bold
        Printer.Canvas.Font.Style := [fsBold];
        for I := 0 to data.FieldCount - 1 do
          begin
            sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));
            Write (PrintFile, Format ('%-' + sizeStr + 's',
              [data.Fields[i].FieldName]));
          end;
        Writeln (PrintFile);

        // for each record of the dataset
        Printer.Canvas.Font.Style := [];
        while not data.EOF do
          begin
            // print out each field of the record
            for I := 0 to data.FieldCount - 1 do
              begin
                sizeStr := IntToStr (min (data.Fields[i].DisplayWidth, maxSize));

```

```

        Write (PrintFile, Format ('%-' + sizeStr + 's',
            [data.Fields[i].AsString]));
    end;
    Writeln (PrintFile);
    // advance ProgressBar
    progress.Position := progress.Position + 1;
    data.Next;
end;
finally
    // close the dataset
    data.Close;
end;
finally
    // reassign the original printer font
    oldFont.Free;
end;
finally
    // close the printer/file
    CloseFile (PrintFile);
end;
end;

```

当我们单击Print ALL按钮的时候，程序便调用这个例程。它执行一个单独的查询（select count(*) from EMPLOYEE），返回employee表格中记录的个数。需要使用这个查询来建立进度条（单向的数据集不可能知道它将要获取多少个记录，除非在到达最后一条记录以后）。然后它设置输出字体，通常使用固定大小的字体，并且调用PrintOutDataSet例程：

```

procedure TNavigator.PrintAllButtonClick(Sender: TObject);
var
    Font: TFont;
begin
    // set ProgressBar range
    EmplCountData.Open;
    try
        ProgressBar1.Max := EmplCountData.Fields[0].AsInteger;
    finally
        EmplCountData.Close;
    end;
    Font := TFont.Create;
    try
        Font.Name := 'Courier New';
        Font.Size := 9;
        PrintOutDataSet (EmplData, ProgressBar1, Font, cbFile.Checked);
    finally
        Font.Free;
    end;
end;

```

包和缓存

ClientDataSet组件从信息包读取数据，包中包含有**PacketRecords**属性指定的一定量的记录。该属性的默认值是-1，这意味着提供者将立即推出所有记录。有时，我们能设置该值为0，表示只向服务器请求字段描述符而不是真实数据；或使用正值来指定真实的编号。

当我们浏览本地缓存后，如果我们只恢复部分数据集，而且**FetchOnDemand**属性被设置为**True**，则**ClientDataSet**组件将从它的源获得更多记录。该相同属性也能控制是否BLOB字段和当前记录的嵌套数据集被自动取出（这些值不能是数据信息包的一部分，而是取决于数据集提供者的**Options**值）。

如果我们关闭该属性，将需要通过调用**GetNextPacket**方法手工提取记录，直到方法返回0（我们将为这些其他元素调用**FetchBlobs**和**FetchDetails**）。

警告：注意，我们在为数据设置索引前，应通过到达最后一条记录或设置**PacketRecords**属性为-1来恢复整个数据集。否则我们将有一个基于部分数据的奇索引。

处理更新操作

关于**ClientDataSet**组件的一个核心想法是：将其用做本地缓存来从用户处收集一些输入，然后发送更新请求到数据库。该组件有应用到数据库服务器的变更列表，使用**ClientDataSet**的相同格式进行存储（通过**Delta**属性可以访问）；还有一个完整的更新日志，我们能用几个方法对其进行操作（包含一个**Undo**即撤销功能）。

提示：在Delphi 7中，**ClientDataSet**组件的**ApplyUpdates**和**Undo**操作都可以通过预定义的动作来访问。

记录的状态

该组件有一个特性，允许我们监视数据包中正在进行的操作，这就是**UpdateStatus**方法，它将为当前记录返回下列指示符：

```
type TUpdateStatus = (usUnmodified, usModified, usInserted, usDeleted);
```

为了在客户端数据集中方便地检测每个记录的状态，我们可以向数据集添加字符串类型的计算字段，并使用下面的**OnCalcFields**事件处理程序计算它的值：

```
procedure TForm1.ClientDataSet1CalcFields(DataSet: TDataSet);
begin
    ClientDataSet1Status.AsString := GetEnumName (TypeInfo(TUpdateStatus),
        Integer (ClientDataSet1.UpdateStatus));
end;
```

这个方法（基于RTTI的**GetEnumName**函数）将**TUpdateStatus**列举的当前值转换为一个字符串，我们可以在图14.10中看到它的效果。

访问增量 (Delta)

除了检测每条记录的状态外，了解给定**ClientDataSet**（还没有上传到服务器中）中变化情况的最好方法是查看增量，这是一个准备应用于服务器的变化信息列表。我们定义该属性如下：

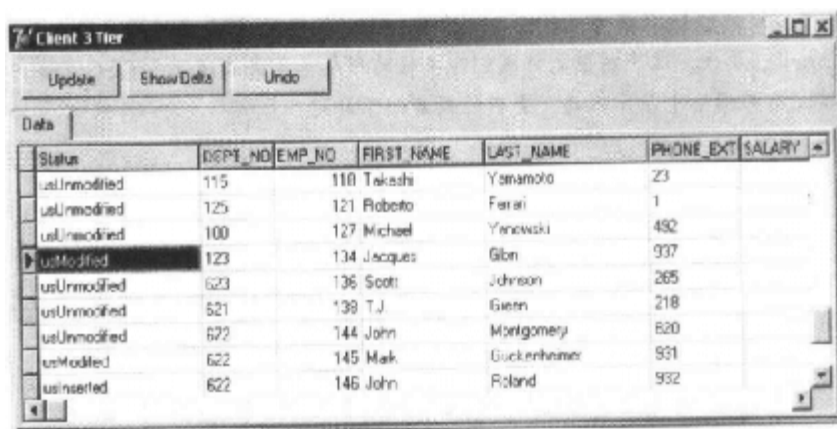


图14.10 CdsDelta程序显示了ClientDataSet中的每一个记录的状态

```
property Delta: OleVariant;
```

Delta属性使用的格式与从客户端向服务器端发送数据的格式相同。我们可以向应用程序添加另一个ClientDataSet组件，并将它与第一个客户端数据集Delta属性中的数据相连：

```
if ClientDataSet1.ChangeCount > 0 then
begin
    ClientDataSet2.Data := ClientDataSet1.Delta;
    ClientDataSet2.Open;
```

在CdsDelta示例，我用两个ClientDataSet组件和一个数据源添加了一个数据模块，这是一个SQLDataSet，映射到InterBase的EMPLOYEE演示表格。两个客户数据集有额外的状态计算字段，比以前讨论的代码更通用，因为事件处理程序在它们之间是共享的。

提示：为了创建与增量状态（运行时）相关联的ClientDataSet的固定字段，我们临时在设计时将其与主ClientDataSet的提供者相连。增量的结构同它引用的数据集一样。创建固定字段后，删除连接。

这个应用程序的窗体有一个页面控件，由两个页面组成，各含一个DBGrid，分别用于真实数据和增量。根据更改日志信息中由ChangeCount方法所返回的值，一些代码可以隐藏或显示第二个选项卡，并当相应的选项卡被选中时，更新增量。用于处理增量的核心代码与上一小节最后的代码相似。在本书选配光碟上有详细的范例源代码。

图14.11显示了CdsDelta应用程序的变化日志信息。请注意Delta数据集对每个被更改的记录具有两个表项（原始值和更改后的值），除非这是一个新的或者已被删除的记录，如它的状态显示的那样。

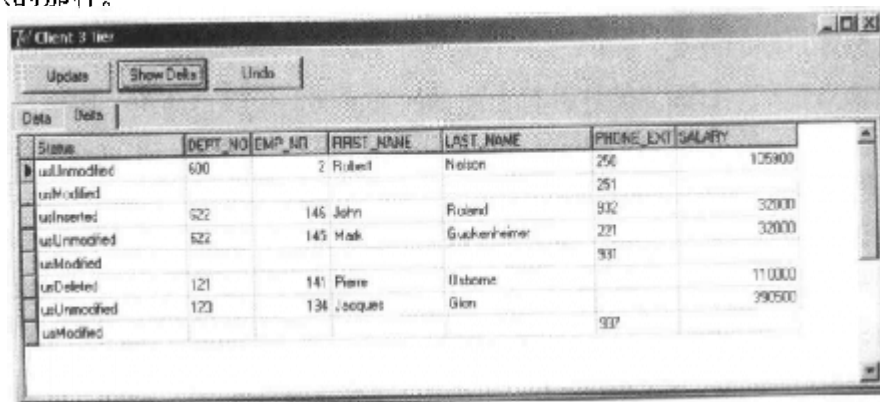


图14.11 CdsDelta范例允许我们察看存储在ClientDataSet的Delta属性中的临时更新请求

提示：我们可以过滤Delta数据集（或者其他的ClientDataSet），但是要根据它的更新状态，并且使用StatusFilter属性。这个属性允许我们在单独的网格中或者是通过在TabControl中选择某个选项经过过滤的网格中显示新的、更新的或者删除的记录。

更新数据

我们现在已经对本地更新期间发生的情况有了很好的理解，现在可以进一步通过将本地更新的数据（存储在增量中）发回到应用程序服务器中去，使该程序正常工作。为了一次性应用数据集中的全部更新，我们可以向ApplyUpdates方法传递“-1”。

如果提供者（或者它内部的Resolver组件）进行数据更新时发生故障，它就会触发OnReconcileError事件。该事件发生在两个不同的操作者同时进行更新的时候。因为我们希望在客户机/服务器程序上使用优化锁定，因此这可以被认为是一种正常情况。

OnReconcileError事件允许我们更改通过引用传递的Action参数，用这个参数来决定服务器的工作方式：

```
procedure TForm1.ClientDataSet1ReconcileError(DataSet: TClientDataSet;  
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
```

该方法有一个参数：客户端数据集组件（在多个客户端应用程序与应用程序服务器交互作用的情况下），引起错误（与错误消息）的异常，失效的操作种类（ukModify、ukInsert或ukDelete）。返回值（存储在Action参数中）可以是以下任何一种：

```
type TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel,  
  raRefresh);
```

raSkip 表示服务器应该跳过发生冲突的记录，并将它留在增量中（这是一个默认的设置）。

raAbort 通知服务器应该放弃全部更新操作，甚至不尝试应用其余的列举在增量中的操作。

raMerge 通知服务器应该将客户端的数据与服务器上的数据合并起来，只应用该客户端的改动字段（并保留由其他客户端改动的其他字段）。

raCorrect 通知服务器应该用当前客户端的数据代替它的数据，亦即覆盖其他客户端的所有字段改变。

raCancel 用于取消更新请求，亦即从增量中删除条目，并恢复最初从数据库（忽视其他客户端的改变）读取的值。

raRefresh 通知服务器应该转储客户增量的更新，可以使用服务器的当前值代替它（保留其他客户端的改变）。

如果读者想在一台独立的计算机上测试冲突情况，可以启动两个客户应用程序，在两个客户端改变同一条记录，然后向服务器发送更新信息。稍后将产生这样一个错误，现在先来介绍处理OnReconcileError事件的方法。

该处理方法实现起来非常简单，但仅仅是因为我们可以获得一些帮助。因为建立一个特殊窗体来处理OnReconcileError事件很常见，所以Delphi已经在对象存储库（在Delphi IDE中选择File►New►Other命令）中提供了这样一个窗体。只需翻到Dialogs页，选择Reconcile Error Dialog选项即可。如CdsDelta范例所示，该单元导出了一个函数，我们可以直接用于

初始化与显示对话框:

```

procedure TmCds.cdsEmployeeReconcileError (DataSet: TCustomClientDataSet;
  E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

```

警告: 就像Reconcile Error Dialog单元的源代码暗示的那样, 我们应该使用Project Options对话框从自动创建窗体的列表中删除该窗体(如果我们不这样做, 当编译项目的时候, 将会产生一个错误)。当然, 只有在没有设置Delphi跳过自动窗体建立的情况下, 才需要这么做。

HandleReconcileError函数创建了对话框窗体并且显示了它, 正如我们在Borland提供的代码中看到的:

```

function HandleReconcileError(DataSet: TDataSet; UpdateKind: TUpdateKind;
  ReconcileError: EReconcileError): TReconcileAction;
var
  UpdateForm: TReconcileErrorForm;
begin
  UpdateForm := TReconcileErrorForm.CreateForm(DataSet, UpdateKind,
    ReconcileError);
  with UpdateForm do
  try
    if ShowModal = mrOK then
    begin
      Result := TReconcileAction(ActionGroup.Items.Objects[
        ActionGroup.ItemIndex]);
      if Result = raCorrect then
        SetFieldValues(DataSet);
    end
    else
      Result := raAbort;
  finally
    Free;
  end;
end;

```

在Reconc单元中, 它具有Reconcile Error对话框(若窗口标题为“Update Error”, 则对于终端用户来说更容易理解), 包含有超过350行的代码, 因此我们无法详细描述它。但是, 读者应该能够通过认真学习源代码来理解它。此外, 也可以使用它查看所有的运行情况。

当发生错误的时候该对话框会出现, 报告所请求的、造成冲突的变化, 并且允许用户选择一个可用的TReconcileAction值。我们可以看到这个窗体在运行时的情况, 如图14.12所示。

提示: 当我们调用ApplyUpdates的时候, 对象存储库启动了一个复杂的更新序列, 我们将在第16章中进行详细介绍。简短地说, 将增量发送给一个提供者, 它就会触发OnUpdateData事件, 然后为每一个要更新的记录接收BeforeUpdateRecord事件。在这两种场合下, 我们必须查看变化并且强制在数据库服务器上的特定操作。

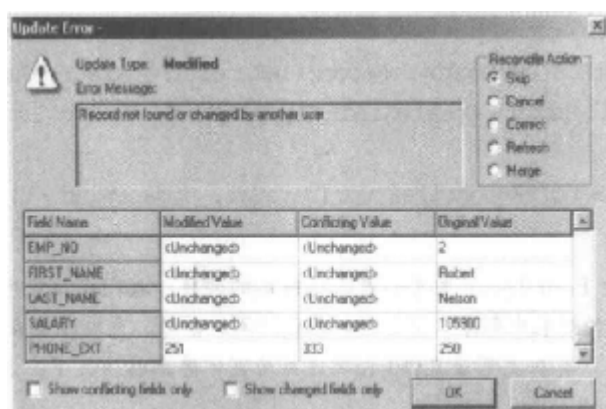


图14.12 Delphi在对象存储库中提供的Reconcile Error对话框，CdsDelta范例使用了它

使用事务

不论何时使用SQL服务器进行工作，我们都应该使用事务让应用程序更强壮。事务可以被描述成一系列单个“原子”整体的、不能分开的操作。

为了弄清楚它的概念，我们举一个例子。假设我们通过固定比率来提升公司员工的薪水，如第13章中的Total示例所示。典型的做法是让程序在服务器上执行一系列SQL语句，针对每一个记录进行更新。如果在操作中有错误发生，则撤销以前的变化。如果我们考虑将“提升每个员工的工资”作为一个单个事务，它或者被全部执行或者被忽视。另外，考虑财务事务分析，如果发生错误，将执行部分操作，亦即加入丢失的账款或额外的款项。

把数据库操作作为事务出于一个有用的目的。我们可以启动事务并完成几个操作（被认为是单个大操作的一部分），然后，或者提交更改或者重新运行事务，放弃所有截止到现在的操作。通常，如果在操作中发生错误，可能需要重新运行事务。

还需要强调另外一个重要的元素：事务的另一个功能是读取数据。当数据被一个事物提交之后，其他连接和（或者）事务才能看到它。一旦数据从一个事物提交上去，其他事物在读取的时候才可以看到变化——除非我们需要打开一个事物并且反复读取同样的数据进行数据分析或者复杂的报表操作。不同的SQL服务器允许我们根据部分或者全部这些选项来读取事务中的数据，正如我们将在讨论事务隔离等级时所看到的。

在Delphi中处理事务是十分简单的。默认情况下，每个edit/post操作均被认为是一个单个的隐式事务，但我们能通过显式地处理它们来改变这个行为。可使用下面三种dbExpress SQLConnection组件的方法（其他数据库连接组件有相似方法）：

StartTransaction 标志一个事务的开端

Commit 在事务执行期间，确认所有对数据库的更新

Rollback 返回数据库开始执行事务之前的状态

还可以使用InTransaction属性来检验一个事务是否是激活状态的。当一个异常产生的时候，我们经常使用try块来返回一个事务，或者在没有任何错误的情况下，作为try块的最后一个操作而提交这个事务。这段代码如下所示：

```
var
    TD: TTransactionDesc;
```

```

begin
    TD.TransactionID := 1;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
    try
        // -- operations within the transaction go here --
        SQLConnection1.Commit(TD);
    except
        SQLConnection1.Rollback(TD);
    end;

```

每一个与事务相关的方法都具有一个描述它操作事务的参数。这个参数使用记录类型 **TTransactionDesc** 并且说明事务隔离等级以及一个事务ID。这个事务隔离等级指明了当其他的事务改动数据的时候，这个事务应该如何行动。下面是三个预定义的值：

tiDirtyRead 使一个事务的更新立即被其他事务和用户看到，甚至在它们被提交之前。这种可能性仅存在于分数数据库，并且符合不支持事务的数据库操作。

tiReadCommitted 只有在更新已被该事务提交之后，才可为其他事务所用。这个设置是对大多数数据库的建议设置，以保护其有效性。

tiRepeatableRead 隐藏在当前事务之后启动的其他事务造成的变动，即使这些变化已经被提交。随后在一个事务中的重复调用将产生相同的结果，好像在当前事务开始时数据库对数据进行了快照一样。只有 **InterBase** 和少数其他数据库服务器在这种模式下工作很有成效。

提示：通常说来，出于性能的考虑，事务应该只涉及最少量的更新（只是那些密切相关的部分原子操作）并且只保留很少的时间。需避免使用那些等候用户输入来完成的事务，因为用户可能会临时离开，这样会使事务长时间地保持激活状态。将变化在本地缓存，就像 **ClientDataSet** 允许的那样，有助于我们使事务更小更快，因为我们能够打开一个进行读取操作的事务，关闭它然后再打开一个事务，写出全部的变化情况。

另一个 **TTransactionDesc** 记录的字段保存着一个事务ID。它只当数据库服务器支持在同一个连接上有多个事务的情况下有用，例如 **InterBase**。我们可以查询连接组件，看看服务器是否支持多个事务或者是否根本不支持事务，这里，我们使用 **MultipleTransactionsSupported** 和 **TransactionsSupported** 属性。

在服务器支持多个事务的情况下，调用 **StartTransaction** 方法时，我们必须给每个事务提供一个唯一的标识符：

```

var
    TD: TTransactionDesc;
begin
    TD.TransactionID := GetTickCount;
    TD.IsolationLevel := xilREADCOMMITTED;
    SQLConnection1.StartTransaction(TD);
    SQLDataSet1.TransactionLevel := TD.TransactionID;

```

我们也可以通过设定每一个数据集的 **TransactionLevel** 属性为事务ID，指明哪一个数据集属于哪一个事务，如前面的语句所示。

为了进一步了解事务以及事务隔离等级，我们可以使用TranSample应用程序。如图14.13所示，单选按钮用于轻易地选择不同的选项，而其他按钮用于操作事务并且应用更新或者刷新数据。为了真正体会不同的效果，可运行程序的多个副本（假设读者在自己的InterBase服务器上有足够的许可证）。

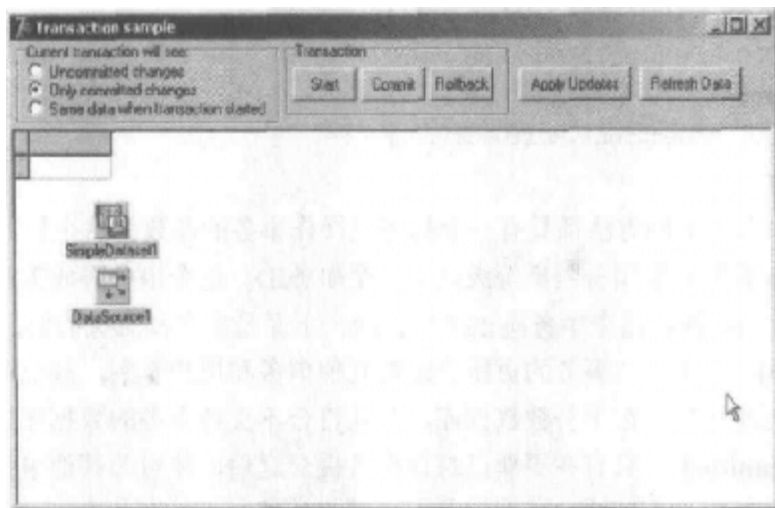


图14.13 设计时TranSample应用程序的窗体。单选按钮用于设定不同的事务隔离等级

说明：InterBase不支持“dirty read”模式，因此，在TranSample程序中，我们不能够使用最后一个选项，除非使用一个不同的服务器。

使用InterBase Express

在本章前面部分建立的示例使用的是新的dbExpress库。使用独立于服务器的方法使我们能够切换应用程序所使用的数据库服务器，尽管这在实际中远谈不上简单。如果我们建立的应用程序总是使用指定的数据库服务器，那么我们可以编写与指定数据库服务器的API直接相关的程序，这种做法将使得程序无法移植到其他SQL服务器上。

当然我们并不总能直接使用那些API，而是往往要将开发工作建立在数据集组件的基础上，这些组件中包括那些API并自然地适应Delphi和它的类库结构。这样的组件系列之一是InterBaseExpress (IBX)。使用这些组件建立的应用程序将工作得更好更快（即使只是在边缘范围），使我们对服务器的特殊功能拥有更多控制。例如，IBX就提供给我们一套特别为InterBase 6建立的管理组件。

说明：我将考察IBX组件，因为它被连接到InterBase（在本章所讨论的数据库服务器），同时也因为该组件集合是惟一可在标准Delphi安装中获得的。其他类似的组件集合（对于InterBase、Oracle以及其他数据库服务器而言）同样十分强大并得到了Delphi程序员组织的重视。InterBase Objects, www.ibobjects.com是一个好示例（对IBX来说是另一个替代方案）。

IBX数据集组件

IBX组件包括了定制的数据集组件以及其他一写组件。这些组件继承自TDataSet基类，可以使用所有常见的Delphi数据敏感控件，而且提供了一个字段编辑器与所有常见的设计时

特性。有多个数据集组件可供挑选。IBX的三个数据集所扮演的角色和所拥有的属性类似于dbExpress系列中的“表/查询/存储过程”组件：

- **IBTable**类似于**Table**组件，用于访问单个的数据表格或视图。
- **IBQuery**与**Query**组件相似，用于执行SQL查询，返回结果集合。**IBQuery**组件可以与**IBUpdateSQL**组件一起用于获得有效的（或可编辑的）数据集。
- **IBStoredProc**与**StoredProc**组件类似，用于执行存储过程。

这些组件与相关的dbExpress组件相似，主要用于在应用程序中提供与旧的BDE组件的兼容性。对于新应用程序来说，通常应使用**IBDataSet**组件，它允许我们使用通过执行select查询获得的有效结果集合。它基本上是将**IBQuery**与**IBUpdateSQL**合并到了一个组件中。实际上，上面的三个组件主要是为复杂的Delphi BDE应用程序提供的。

InterBase Express中的其他很多组件不属于数据集类别，但仍用于需要访问数据库的应用程序。

- **IBDatabase**模拟**DBX SQLConnection**组件并被用于设置数据库连接。**BDE**也使用指定的会话组件来执行一些由**IBDatabase**组件完成的全局任务。
- **IBTransacdon**允许全面控制事务。在**InterBase**中明确地使用事务并相应地隔离每个事务（对于报表使用**Snapshot**隔离级别，对于交互式窗体使用**Read Committed**级别）非常重要。每个数据集都明确地引用给定的事务，所以对于相同的数据库可以有多个并发事务，并且选择哪一个数据集参与哪一个事务。
- **IBSQL**允许我们执行不返回数据集的SQL语句（例如，DDL请求，或更新与删除语句），而不用给数据集组件带来额外的负担。
- **IBDatabaseInfo**用于查询数据库结构与状态。
- **IBSQLMonitor**用于调试系统，因为由Delphi提供的**SQL Minitor**调试器是个BDE专用的工具。
- **IBEvents**接收服务器传递的事件。

这一组组件提供的数据库服务器控制要比**BDE**提供的多得多。例如，一个专门的事务组件允许我们在一个或多个数据库上管理多个并发事务，一个事务也可以横跨多个数据库。**IBDatabase**组件允许我们建立数据库，测试连接，并访问系统数据以及**Database**和**Session** **BDE**组件没有完全提供的一些内容。

提示：IBX数据集能够作为一种自动递增的字段建立生成器的自动性能。这可以通过使用它专门的属性编辑器设置**GeneratorField**属性来实现。在本章稍后的“生成器与ID”一节中将讨论有关范例。

IBX管理组件

Delphi组件面板的**InterBase Admin**中含有**InterBase**管理组件。尽管我们的目标不是建立完整的**InterBase**控制台程序，但在应用程序中包括一些管理特性（如备份处理或用户监控）对于水平较高的用户来讲是很有意义的。

这些组件中的大多数都有自解释名称：它们是**IBConfigService**、**IBBackupService**、**IBRestoreService**、**IBValidationService**、**IBStatisticalService**、**IBLogSenice**、**IBSecurityService**、**IBServerProperties**、**IBInstall**和**IBUninstall**。我们将不建立任何使用这些组件的高级

范例，因为它们更多侧重于服务器管理应用程序的开发，而不是客户程序的开发。我们只是在本章稍后的IbxMon范例程序中包含了两个这样的组件。

建立一个IBX范例

为了建立一个使用IBX的范例，我们将需要在一个窗体或者数据模块上放置至少三个组件：IBDatabase、IBTransaction以及数据集组件（在这个例子中是IBQuery）。任何IBX应用程序至少都会需要这两个组件的一个实例。我们不能在数据集中设置数据库连接，就像在其他数据集中一样，而且至少需要有一个事务对象，用于读取查询结果：

下面是IbxEmp范例中这些组件的主要属性：

```
object IBTransaction1: TIBTransaction
  Active = False
  DefaultDatabase = IBDatabase1
end
object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  CachedUpdates = False
  SQL.Strings = (
    'SELECT * FROM EMPLOYEE')
end
object IBDatabase1: TIBDatabase
  DatabaseName = 'C:\Program Files\Common Files\Borland Shared\Data\employee.gdb'
  Params.Strings = (
    'user_name=SYSDBA'
    'password=masterkey')
  LoginPrompt = False
  SQLDialect = 1
end
```

现在我们将DataSource组件¹与IBQuery1相连，然后简单地在这个应用程序建立用户界面。我们必须输入Borland范例数据库的路径名。然而，不是每个人都有Program Files文件夹（取决于Windows的本地版本），当然Borland范例数据文件可以安装到磁盘的其他任意位置。我们将在下一个范例中解决这些问题。

警告：注意，我们将密码嵌入到了代码当中，这是一种非常“天真”的安全方法。不但任何人可以运行该程序，而且甚至有人可以通过查看可执行文件的十六进制代码取出密码。我们使用该方法是为了在测试程序时不需要反复输入密码，但在真正的应用程序中，如果用户关心自己数据的安全性，则应该设置真正意义上的密码。

建立现场查询

IbxEmp范例有一个不允许编辑的查询。为了激活编辑功能，需要向查询添加IBUpdateSQL组件，即使这个查询是微不足道的。使用具有SQL select语句的IBQuery以及具有insert、update和delete SQL语句的IBUpdateSQL组件，是BDE应用程序通常的方式。这些组件之间的相似之

处使得从一个现有的BDE应用程序向这个体系进行迁移变得简单。下面是这个组件的代码:

```

object IBQuery1: TIBQuery
  Database = IBDatabase1
  Transaction = IBTransaction1
  SQL.Strings = (
    'SELECT Employee.EMP_NO, Department.DEPARTMENT, Employee.FIRST_NAME, '+'
    ' Employee.LAST_NAME, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, '+'
    ' Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    '  INNER JOIN DEPARTMENT Department'
    '    ON (Department.DEPT_NO = Employee.DEPT_NO)'
    '  INNER JOIN JOB Job'
    '    ON (Job.JOB_CODE = Employee.JOB_CODE)'
    '    AND (Job.JOB_GRADE = Employee.JOB_GRADE)'
    '    AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)'
    'ORDER BY Department.DEPARTMENT, Employee.LAST_NAME')
  UpdateObject = IBUpdateSQL1
end
object IBUpdateSQL1: TIBUpdateSQL
  RefreshSQL.Strings = (
    'SELECT Employee.EMP_NO, Employee.FIRST_NAME, Employee.LAST_NAME, '+'
    ' Department.DEPARTMENT, Job.JOB_TITLE, Employee.SALARY, Employee.DEPT_NO, '+'
    ' Employee.JOB_CODE, Employee.JOB_GRADE, Employee.JOB_COUNTRY'
    'FROM EMPLOYEE Employee'
    'INNER JOIN DEPARTMENT Department'
    'ON (Department.DEPT_NO = Employee.DEPT_NO)'
    'INNER JOIN JOB Job'
    'ON (Job.JOB_CODE = Employee.JOB_CODE)'
    'AND (Job.JOB_GRADE = Employee.JOB_GRADE)'
    'AND (Job.JOB_COUNTRY = Employee.JOB_COUNTRY)'
    'WHERE Employee.EMP_NO=:EMP_NO')
  ModifySQL.Strings = (
    'update EMPLOYEE'
    'set'
    '  FIRST_NAME = :FIRST_NAME,'
    '  LAST_NAME = :LAST_NAME,'
    '  SALARY = :SALARY,'
    '  DEPT_NO = :DEPT_NO,'
    '  JOB_CODE = :JOB_CODE,'
    '  JOB_GRADE = :JOB_GRADE,'
    '  JOB_COUNTRY = :JOB_COUNTRY'
    'where'
    '  EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'

```

```

      '(FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, JOB_COUNTRY)'
      'values'
      '(:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, :JOB_GRADE, :JOB_COUNTRY)')
DeleteSQL.Strings = (
  'delete from EMPLOYEE '
  'where EMP_NO = :OLD_EMP_NO')
end

```

对于新的应用程序来说，我们应该考虑使用IBDataSet组件，该组件集中了IBQuery和IBUpdateSQL的特性。使用这两个组件与单独使用哪一个组件的区别很小。使用IBQuery和IBUpdateSQL在下面的情况下是一个很好的解决方法，即当我们把一个现存的基于两个等价BDE组件的应用程序转移到一个IBDataSet组件中的时候，甚至在直接转移一个应用程序的时候，也不需要太多的额外操作。

在IbxUpdSql范例中，我们提供了这两种方法，所以读者可以自己检测它们的不同之处。下面是单个数据集组件的DFM描述的框架：

```

object IBDataSet1: TIBDataSet
  Database = IBDatabase1
  Transaction = IBTransaction1
  DeleteSQL.Strings = (
    'delete from EMPLOYEE'
    'where EMP_NO = :OLD_EMP_NO')
  InsertSQL.Strings = (
    'insert into EMPLOYEE'
    ' (FIRST_NAME, LAST_NAME, SALARY, DEPT_NO, JOB_CODE, JOB_GRADE, ' +
    ' JOB_COUNTRY)'
    'values'
    ' (:FIRST_NAME, :LAST_NAME, :SALARY, :DEPT_NO, :JOB_CODE, ' +
    ' :JOB_GRADE, :JOB_COUNTRY)')
  SelectSQL.Strings = (...)
  UpdateRecordTypes = [cusUnmodified, cusModified, cusInserted]
  ModifySQL.Strings = (...)
end

```

如果将IBQuery或IBDataSet组件与数据源相连，并运行程序，我们会看到效果是相同的。不但组件的功能相似，属性与事件也非常相似。

这一次，在IbxUpdSql程序中，我们使对数据库的引用更为灵活了。我们没有在设计时输入数据库名称，而是从Windows注册表中读取它（在安装程序时，Borland会将它保存在此）。当程序启动时，执行的代码如下所示：

```

uses
  Registry;

procedure TForm1.FormCreate(Sender: TObject);
var
  Reg: TRegistry;
begin

```



```

Reg := TRegistry.Create;
try
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  Reg.OpenKey('\Software\Borland\Borland Shared\Data', False);
  IBDatabase1.DatabaseName := Reg.ReadString('Rootdir') + '\employee.gdb';
finally
  Reg.Free;
end;
EmpDS.DataSet.Open;
end;

```

说明：关于Windows注册表和INI文件的更多信息，参见第8章“Delphi应用程序的结构”中的相关内容。

该范例的另一个特性是使用了事务组件。我们已经说过，InterBase Express组件强制使用事务组件，并明确地遵循InterBase的要求。只需向窗体添加两个按钮，用于提交或返回事务就足够了，因为当编辑与事务相关的任何数据集时，它就会自动启动。

我们还通过添加一个ActionList组件稍稍改进了该程序。该组件包括所有标准的数据库动作，并为事务支持添加了两个定制的动作，Commit和Rollback。当事务启动时，两个动作都可以使用：

```

procedure TForm1.ActionUpdateTransactions(Sender: TObject);
begin
  acCommit.Enabled := IBTransaction1.InTransaction;
  acRollback.Enabled := acCommit.Enabled;
end;

```

当被执行时，它们除了执行主要操作外，还需要在新事务中重新打开数据集（还可以通过“保留”事务环境做到这一点）。事实上，CommitRetaining并不重新开启新事物，但它允许当前事物继续处于打开状态。因此，可以继续使用数据集，它不能被重刷新（因此无法看到其他用户已提交的编辑），但会继续显示已修改的数据。下面是代码：

```

procedure TForm1.acCommitExecute(Sender: TObject);
begin
  IBTransaction1.CommitRetaining;
end;

procedure TForm1.acRollbackExecute(Sender: TObject);
begin
  IBTransaction1.Rollback;
  // reopen the dataset in a new transaction
  IBTransaction1.StartTransaction;
  EmpDS.DataSet.Open;
end;

```

警告：当事务终止时，InterBase会关闭任何打开的数据集，这意味着，即使我们没有做任何改动，我们仍需要重新打开它们并重取数据。当提交数据时，我们通过CommitRetaining命令来请求InterBase保留事务的上下文，而不关闭打开的数据集，如前文所述。InterBase这样工作的原因

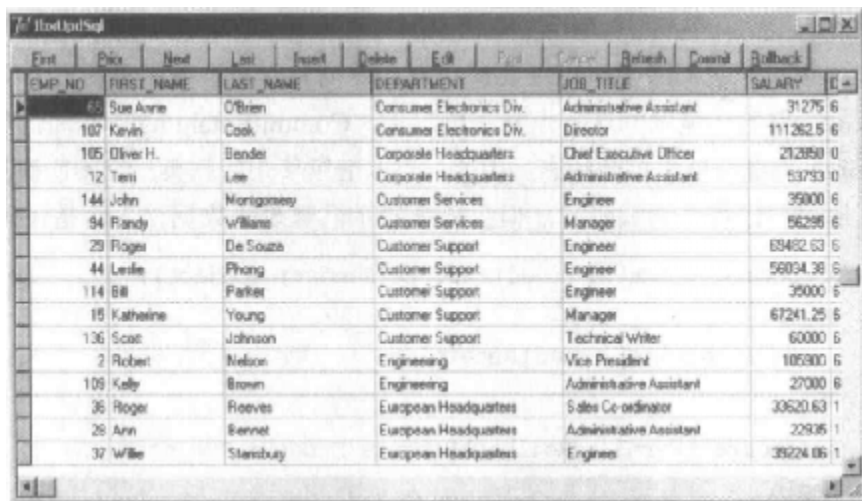
是事务对应着数据的快照。一旦事务结束，我们应该重新读取数据，以重新获取已经被其他用户改动了的记录。InterBase 6.0版本也包括一个我已决定不用的RollbackRetaining命令，因为在回滚操作中，程序将刷新数据集，在屏幕上显示原始值，而不更新已抛弃的更新数据。

最后一步操作引用了通用数据集，而不是特殊的数据集，因为我们打算向程序添加另一个数据集。这些操作与只有文本的工具栏相连，如图14.14所示。程序在启动时打开数据集，并在退出时，在询问了用户要做什么之后，使用下列OnClose事件处理程序，自动关闭当前事务：

```

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
    nCode: Word;
begin
    if IBTransaction1.InTransaction then
    begin
        nCode := MessageDlg ('Commit Transaction? (No to rollback)',
            mtConfirmation, mbYesNoCancel, 0);
        case nCode of
            mrYes: IBTransaction1.Commit;
            mrNo: IBTransaction1.Rollback;
            mrCancel: Action := caNone; // don't close
        end;
    end;
end;

```



| EMP_NO | FIRST_NAME | LAST_NAME | DEPARTMENT | JOB_TITLE | SALARY |
|--------|------------|------------|---------------------------|--------------------------|----------|
| 65 | Sue Anne | O'Brien | Consumer Electronics Div. | Administrative Assistant | 31275.6 |
| 107 | Kevin | Cook | Consumer Electronics Div. | Director | 111262.5 |
| 105 | Oliver H. | Bender | Corporate Headquarters | Chief Executive Officer | 212350.0 |
| 12 | Teri | Lee | Corporate Headquarters | Administrative Assistant | 53793.0 |
| 144 | John | Montgomery | Customer Services | Engineer | 35000.6 |
| 94 | Randy | Williams | Customer Services | Manager | 56295.6 |
| 29 | Roger | De Souza | Customer Support | Engineer | 69462.63 |
| 44 | Leslie | Phong | Customer Support | Engineer | 56034.38 |
| 114 | Bill | Parlier | Customer Support | Engineer | 35000.6 |
| 15 | Katharine | Young | Customer Support | Manager | 67241.25 |
| 136 | Scott | Johnson | Customer Support | Technical Writer | 60000.6 |
| 2 | Robert | Nelson | Engineering | Vice President | 105900.6 |
| 109 | Kelly | Brown | Engineering | Administrative Assistant | 27000.6 |
| 36 | Roger | Reeves | European Headquarters | Sales Co-ordinator | 30520.63 |
| 26 | Ann | Bennet | European Headquarters | Administrative Assistant | 22935.1 |
| 32 | Wille | Stensburg | European Headquarters | Engineer | 35224.06 |

图14.14 IbxUpdSql范例的输出

监控InterBase Express

与dbExpress体系结构类似，IBX还允许我们监控一个连接。为此，我们需要嵌入一个IBSQLMonitor组件的拷贝到应用程序中，然后生成一个定制日志。

我们甚至可以编写更为通用的监控应用程序，如IbxMon范例中那样，在其窗体中放置一个监控组件与一个RichEdit控件，并为OnSQL事件编写如下的处理程序：

```

procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
    if Assigned (RichEdit1) then
        RichEdit1.Lines.Add (TimeToStr (Now) + ': ' + EventText);
end;

```

if Assigned测试在停止运行期间接收消息时非常有用，而且当我们直接向正在监控中的应用程序添加该代码时也需要它。

为了从其他应用程序（或当前的程序）接收消息，我们必须打开IBDatabase组件的追踪选项。在IbxUpdSql范例（在前面“建立现场查询”一节中曾讨论过）中，我们将它们都打开了：

```

object IBDatabase1: TIBDatabase
    ...
    TraceFlags = [tfQPrepare, tfQExecute, tfQFetch, tfError, tfStmt,
        tfConnect, tfTransact, tfBlob, tfService, tfMisc]

```

如果我们同时运行这两个范例，IbxMon将会列举关于IbxUpdSql与InterBase互操作的详细信息，如图14.15所示。

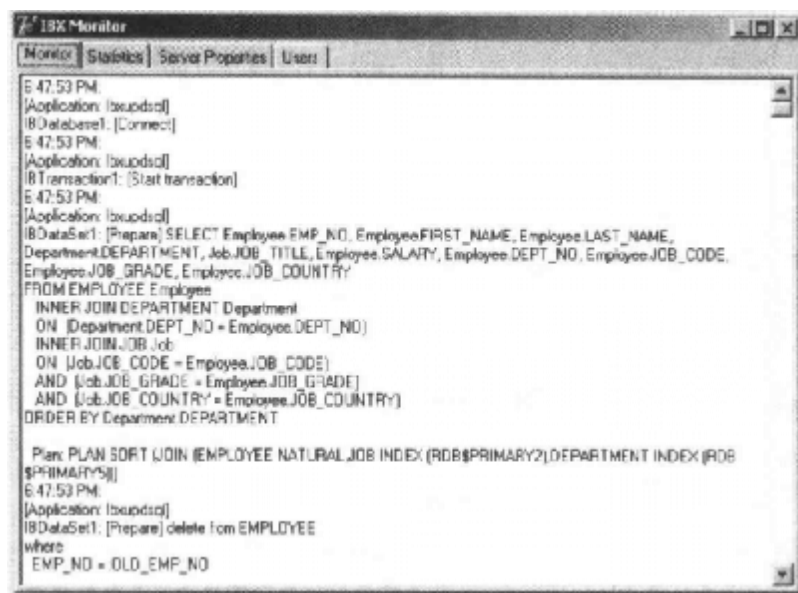


图14.15 IbxMon范例的输出，基于IBMonitor组件

获得更多的系统数据

IbxMon范例不只监控InterBase连接，它还使用页面控件上的各种标签查询服务器的一些设置。该范例包含了一些IBX管理组件，显示了服务器的统计数据、一些服务器属性以及所有连接的用户。我们可以在图14.16中看到服务器属性的范例，下面的代码段则显示了提取用户信息的代码：

```

// grab the user's data
IBSecurityService1.DisplayUsers;
// display the name of each user

```

```
for i := 0 to IBSecurityService1.UserInfoCount - 1 do  
  with IBSecurityService1.UserInfo[i] do  
    RichEdit4.Lines.Add (Format ('User: %s, Full Name: %s, Id: %d',  
      [UserName, FirstName + ' ' + LastName, UserId]));
```

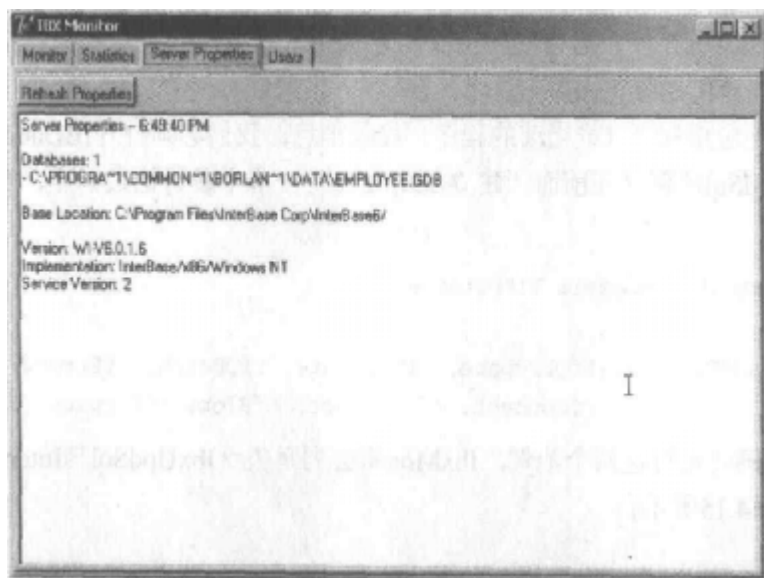


图14.16 IbxMon应用程序显示的服务器信息

实际问题

到现在为止，我们讨论了关于InterBase编程的特殊技术，但没有深入到实际应用程序的开发中，没有涉及实际中存在的问题。在随后的小节中，我们将讨论一些实际技术（没有什么特殊顺序）。

Nando Dessena（他比我更了解InterBase）与我在一个讨论内部Paradox应用程序向InterBase移植的研讨会上使用了所有这些技术。在那种环境下讨论的应用程序更大也更复杂，我将它整理为一些小段，以适合本章的篇幅。

提示：本节讨论的数据库被称为mastering.gdb，本书选配光碟上本章文件夹的data子文件夹中也包含了它。读者可以使用InterBase Console来检验它，可能需要将它复制到硬盘上，这样就可以完全与它交互作用了。

生成器与ID

我在第13章中提到，我很喜欢使用ID扩展来识别数据库每个表格中的记录。

说明：我喜欢对整个系统使用单一的ID序列，通常被称为Object ID (OID)，我们在本章前面已经讨论过。然而，在这种环境下，两个数据表格的ID必须是惟一的。因为我们预先不知道哪一个对象可能用于代替其他对象，所以采取一个全局的OID能在以后提供更大的自由。缺点是，如果我们有大量数据，使用一个整型数作为ID（也就是说，只有40亿个对象）可能不够。因此，InterBase 6支持64位的生成器。

当有多个客户端在运行时，我们如何为这些ID生成惟一的值呢？在一个数据表格中保存最新的值会遇到麻烦，因为多个并发的任务（来自不同用户）会看到相同的值。如果不使用数据表格，可以使用一种独立于数据库的机制，包括更大的Windows GUID或所谓的高低位技术（high-low technique，在启动时将一个基础数值赋给每个客户端——高位值与一个连续值组合在一起，而低位值由客户端自己决定）。

另一种方法与数据库有关，是将内部机制用于序列，在每个SQL服务器中用不同的名称来说明。在InterBase中，它们被称为生成器。这些序列的特征是，它们在事务外操作并递增，所以它们会提供惟一的数值，即使是并发用户也一样（注意InterBase迫使我们打开事务来读取数据）。

我们已经介绍了如何建立一个生成器。下面是演示数据库中一个生成器的定义，随后的视图定义可以用于查询新值：

```
create generator g_master;

create view v_next_id (
    next_id
) as
select gen_id(g_master, 1) from rdb$database
;
```

在RWBlocks应用程序中，我们用下面的SQL语句向数据模块添加了一个IBQuery组件（因为我们不需要它作为一个可编辑的数据集）：

```
select next_id from v_next_id;
```

与直接使用语句相比，这样做的优点是更容易编写与维护，即使生成器发生变化（或一旦我们切换到幕后的另一种方法）。而且，在相同的数据模块中，我们还添加了一个函数，它可为生成器返回一个新值：

```
function TDMMain.GetNewId: Integer;
begin
    // return the next value of the generator
    QueryId.Open;
    try
        Result := QueryId.Fields[0].AsInteger;
    finally
        QueryId.Close;
    end;
end;
```

该方法可以在任何数据集的AfterInsert事件中调用，以充填ID的值：

```
mydataset.FieldByName ('ID').AsInteger := data.GetNewId;
```

我们曾提到，Delphi中的IBX数据集可以直接与生成器相连，这大大简化了整体情况。事实上，由于存在特定的属性编辑器（如图14.17所示），将数据集的一个字段连接到生成器变得非常简单了。



图14.17 用于IBX数据集的GeneratorField属性的编辑器

注意，这两种方法都要比基于服务器端触发器的方法（本章前面曾讨论过）好。事实上，在那种情况下，Delphi应用程序不知道发送给数据库的记录ID，所以就不能刷新它。在Delphi端没有记录ID（也是惟一的关键字段）意味着它几乎不可能将这样的值直接插入到DBGrid中。如果我们试着这样做，会发现插入的值显然被丢失了，只有在完全刷新的情况下才会重新出现。

基于手工代码或GeneratorField属性使用客户端技术不会引起麻烦，因为Delphi应用程序在传递记录之前知道ID（记录的关键字），所以可以轻松地将它放在网格中，并相应地刷新它。

大小写不敏感的查找

通常，使用SQL服务器（不是InterBase）的一个有意思的问题是必须要处理不分大小写的查找。假设我们不想在网格中显示大量的数据（对于客户机/服务器应用程序来说不是一个好主意），那么可选择让用户输入名称的初始部分，然后在该输出上过滤查询，只显示网格中较小的结果记录集合。我们将该方法用到了一个公司的表格上。

通过公司名称进行这样的查询是经常性的，而且可能发生在大型数据表格上。然而，如果我们只是使用startin gwith或like操作符，查找将是分大小写的，如下列SQL语句所示：

```
select * from companies
where name starting with 'win';
```

为了进行不分大小写的查找，我们可以在比较两边使用upper函数检测每个字符串的大写值，但相似的查询速度很慢，因为它没有基于索引。另一方面，用大写字母保存公司名（或其他任何名称）更愚蠢，因为当我们必须打印出这些名称时，结果将非常不自然（即使在旧的信息系统中这很常见）。

如果为了提高速度需要牺牲一些磁盘空间与内存，则可以使用这样一个技巧：将额外的字段添加给数据表格，以存储公司名的大写值；同时使用服务器端的触发器，以生成并更新它。然后，我们可以要求数据库维护名称大写版上的索引，以进一步加快查找操作的速度。

实际上，数据表格的定义如下所示：

```
create domain d_uid as integer;
create table companies
(
```

```

    id          d_uid not null,
    name        varchar(50),
    tax_code    varchar(16),
    name_upper  varchar(50),
    constraint companies_pk primary key (id)
);

```

为了将每个公司的大写名称复制到相关字段中，我们不能依靠客户端的代码，因为任何不协调都会引起问题。在这种情况下，最好使用服务器上的触发器，这样每当公司名发生变化时，它的大写版本就会相应地被更新。另一个触发器将用于插入新公司：

```

create trigger companies_bi for companies
active before insert position 0
as
begin
    new.name_upper = upper(new.name);
end;

create trigger companies_bu for companies
active before update position 0
as
begin
    if (new.name <> old.name) then
        new.name_upper = upper(new.name);
    end;

```

最后，我们使用下面这条DLL语句，将索引添加到表格中：

```
create index i_companies_name_upper on companies(name_upper);
```

使用这个结构的背后意图是，我们现在可以选择所有的具有编辑框（edSearch）中文本为开头的公司——通过在一个Delphi应用程序中编写下面的代码：

```

dm.DataCompanies.Close;
dm.DataCompanies.SelectSQL.Text :=
    'select c.id, c.name, c.tax_code, ' +
    ' from companies c ' +
    ' where name_upper starting with ''' +
    UpperCase (edSearch.Text) + ''';
dm.DataCompanies.Open;

```

提示：使用一个准备好的参数查询，我们可以使代码运行的速度进一步加快。

作为另外一种选择，我们可以在表格定义中创建一个服务器端的计算字段，但是这么做将会阻止我们拥有一个字段上的索引，而这个索引可以显著地加速我们的查询：

```
name_upper  varchar(50) computed by (upper(name))
```

处理地址与人员

我们可以注意到，描述公司的表格很空。事实上，它不含公司地址，也不包括联系信息。原因很简单：我们想能够处理具有多个办公室（或地址）的公司，并列出这些公司多个

员工的联系信息。

每个地址都与一个公司有关。注意，我们决定不使用公司有关的地址标识符（如每个公司的地址号码），而是对所有地址使用一个全局ID。这样，我们可以引用一个地址ID，而不必还引用公司ID。下面是存储公司地址的表格定义：

```
create table locations
(
    id            d_uid not null,
    id_company    d_uid not null,
    address       varchar(40),
    town          varchar(30),
    zip           varchar(10),
    state         varchar(4),
    phone         varchar(15),
    fax           varchar(15),
    constraint locations_pk primary key (id),
    constraint locations_uc unique (id_company, id)
);

alter table locations add constraint locations_fk_companies
    foreign key (id_company) references companies (id)
    on update no action on delete no action;
```

外部关键字的最后定义将地址表格的id_company字段与公司表格的ID字段联系起来。其他表格列出特定公司地址的人员名称与联系信息。为了遵循数据库标准化的规则，我们应该只向该表格添加对地址的引用，因为每个地址都与一个公司有关。然而，为了更容易地改变公司中某个人员的地址，并使查询更高效（避免额外的操作），我们向people表格添加了对地址的引用和对公司的引用。

数据表格还有另一个不寻常的特性：为公司工作的某个人员可以被设置为关键联系人。这可以通过一个Boolean字段（用一个域来定义，因为InterBase不支持Boolean类型）并向表格添加触发器来实现，这样每个公司只有一个员工可以具有该标记：

```
create domain d_boolean as char(1)
    default 'F'
    check (value in ('T', 'F')) not null

create table people
(
    id            d_uid not null,
    id_company    d_uid not null,
    id_location   d_uid not null,
    name          varchar(50) not null,
    phone         varchar(15),
    fax           varchar(15),
    email         varchar(50),
    key_contact   d_boolean,
    constraint people_pk primary key (id),
```



```

constraint people_uc unique (id_company, name)
);

alter table people add constraint people_fk_companies
foreign key (id_company) references companies (id)
on update no action on delete cascade;
alter table people add constraint people_fk_locations
foreign key (id_company, id_location)
references locations (id_company, id);

create trigger people_ai for people
active after insert position 0
as
begin
    /* if a person is the key contact, remove the
       flag from all others (of the same company) */
    if (new.key_contact = 'T') then
        update people
        set key_contact = 'F'
        where id_company = new.id_company
        and id <> new.id;
    end;

    create trigger people_au for people
    active after update position 0
    as
    begin
        /* if a person is the key contact, remove the
           flag from all others (of the same company) */
        if (new.key_contact = 'T' and old.key_contact = 'F') then
            update people
            set key_contact = 'F'
            where id_company = new.id_company
            and id <> new.id;
        end;
    end;
end;

```

建立用户界面

我们目前讨论的三个表格都具有明显的主/详关系。因此，RWBlocks范例需要使用三个IBDataSet组件访问数据，将两个二级表格挂向主表格。主/详支持的代码是一个基于查询的标准数据库范例，所以我们将不进一步讨论它（但建议读者研究该范例的源代码）。

每个数据集都有一整套SQL语句，使得数据可以编辑。只要我们输入一个新的“详”元素，程序就会将它挂到其主表格上，如下列两个方法中所示：

```

procedure TDmCompanies.DataLocationsAfterInsert(DataSet: TDataSet);
begin
    // initialize the data of the detail record
    // with a reference to the master record

```

```

    DataLocationsID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
end;

procedure TDMCompanies.DataPeopleAfterInsert(DataSet: TDataSet);
begin
    // initialize the data of the detail record
    // with a reference to the master record
    DataPeopleID_COMPANY.AsInteger := DataCompaniesID.AsInteger;
    // the suggested location is the active one, if available
    if not DataLocations.IsEmpty then
        DataPeopleID_LOCATION.AsInteger := DataLocationsID.AsInteger;
    // the first person added becomes the key contact
    // (checks whether the filtered dataset of people is empty)
    DataPeopleKEY_CONTACT.AsBoolean := DataPeople.IsEmpty;
end;

```

如代码中所示，一个数据模块管理着数据集组件。实际上，程序对每个窗体都建立了一个数据模块（动态挂接，因为我们可以为每个窗体建立多个实例）。其中每个数据模块都有一个独立的事务，所以在不同页上执行的不同操作都是完全独立的。然而，数据库连接被集中化了。主数据模块控制着相应的组件，它由所有数据集引用。每个数据模块都由引用它的窗体动态建立，而且它的值存储在窗体的dm专用字段中：

```

procedure TFormCompanies.FormCreate(Sender: TObject);
begin
    dm := TDMCompanies.Create (Self);
    dsCompanies.Dataset := dm.DataCompanies;
    dsLocations.Dataset := dm.DataLocations;
    dsPeople.Dataset := dm.DataPeople;
end;

```

这样，我们可以轻松地建立一个窗体的多个实例，并且每个窗体都有一个数据模块实例与之相连。与数据模块相连的窗体有三个DBGrid控件，每个控件都与一个数据模块以及一个相应的数据集相连。图14.18显示了该窗体运行时的情况，其中有一些实际数据。

该窗体实际归一个主窗体管理，它又依次基于一个页控件，还包含着其他窗体。只有与第一页相连的窗体才会在程序启动时建立。我们编写的ShowForm方法负责在删除了窗体边框后，将窗体作为页控件的子控件：

```

procedure TFormMain.FormCreate(Sender: TObject);
begin
    ShortDateFormat := 'dd/mm/yyyy';
    ShowForm (TFormCompanies.Create (Self), TabCompanies);
end;

procedure TFormMain.ShowForm (Form: TForm; Tab: TTabSheet);
begin
    Form.BorderStyle := bsNone;
    Form.Align := alClient;

```

```

Form.Parent := Tab;
Form.Show;
end;

```

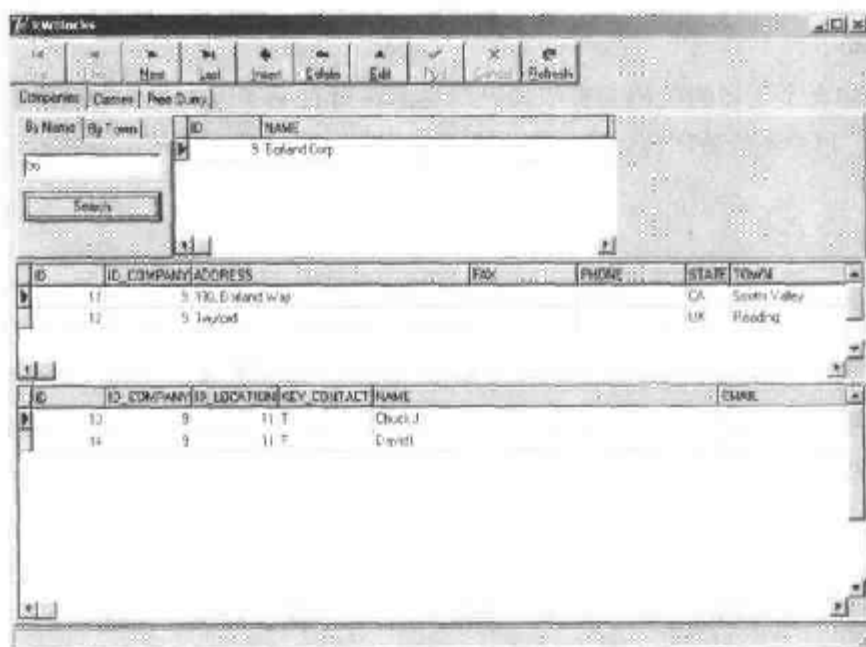


图14.18 一个显示公司信息、办公地点以及人员的窗体（RWBlocks范例的一部分）

另外两个页面在运行时生成:

```

procedure TFormMain.PageControl1Change(Sender: TObject);
begin
    if PageControl1.ActivePage.ControlCount = 0 then
        if PageControl1.ActivePage = TabFreeQ then
            ShowForm (TFormFreeQuery.Create (self), TabFreeQ)
        else if PageControl1.ActivePage = TabClasses then
            ShowForm (TFormClasses.Create (self), TabClasses);
end;

```

该公司窗体带有按照公司名称进行的搜索（在前一节讨论过），以及按照位置进行的搜索。输入一个城镇的名称后，即可获得一个在该城镇中拥有办公室的公司的列表:

```

procedure TFormCompanies.btnTownClick(Sender: TObject);
begin
    with dm.DataCompanies do
        begin
            Close;
            SelectSQL.Text :=
                'select c.id, c.name, c.tax_code' +
                ' from companies c ' +
                ' where exists (select loc.id from locations loc ' +
                ' where loc.id_company = c.id and upper(loc.town) = ' +
                UpperCase(edTown.Text) + ' )';
        end

```

```

    Open;
    dm.DataLocations.Open;
    dm.DataPeople.Open;
  end;
end;

```

该窗体包含了更多的代码。有些代码与关闭许可有关（因为在仍有未决编辑没有发送给数据库时，用户不能关闭窗体），还有大量代码与用做查找对话框的窗体有关，稍后我们会看到。

登记课程

程序与数据库的另一部分涉及登记培训课程和内容（尽管我们建立的该程序只用于演示，但它还是可以帮助我们经营自己的商务活动的）。在数据库中，有一个课程表列出了所有的培训课程，每个课程都有一个标题与计划日期。另一个表格保存了公司的注册信息，包括注册的课程、公司的ID等等。最后，第三个表格列出了签约的人员列表，每个人员都与其公司的注册表以及支付款数相连。

这个基于公司的注册表的基本原理是，将发票发送给公司，由它为其程序员登记课程，并处理折扣。在这种情况下，数据库更规范些，因为人员注册没有直接引用课程，而只是向公司注册课程。下面是所涉及数据表格的定义（忽略了外部约束与其他元素）：

```

create table classes
(
    id            d_uid not null,
    description   varchar(50),
    starts_on    timestamp not null,
    constraint classes_pk primary key (id)
);
create table classes_reg
(
    id            d_uid not null,
    id_company    d_uid not null,
    id_class      d_uid not null,
    notes         varchar(255),
    constraint classes_reg_pk primary key (id),
    constraint classes_reg_uc unique (id_company, id_class)
);
create domain d_amount as numeric(15, 2);
create table people_reg
(
    id            d_uid not null,
    id_classes_reg d_uid not null,
    id_person      d_uid not null,
    amount         d_amount,
    constraint people_reg_pk primary key (id)
);

```

这组数据表格的数据模块使用了一种主/详/详关系，当建立新的从记录时，就用代码设置同主记录的连接。每个数据集的ID都有一个生成器字段，而且每个字段都有相应的update和insert SQL语句。这些语句由相应的组件编辑器来生成，只使用ID字段来识别已有记录并只更新原数据表格的字段。事实上，两个二级数据集都从一个查找表格中检索数据，包括公司列表或人员列表。最后，我们必须手工编辑Refresh SQL语句，重复相应的内部连接。下面是一个范例：

```
object IBClassReg: TIBDataSet
  Database = DmMain.IBDatabase1
  Transaction = IBTransaction1
  AfterInsert = IBClassRegAfterInsert
  DeleteSQL.Strings = (
    'delete from classes_reg'
    'where id = :old_id')
  InsertSQL.Strings = (
    'insert into classes_reg (id, id_class, id_company, notes)'
    'values (:id, :id_class, :id_company, :notes)')
  RefreshSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id = :id')
  SelectSQL.Strings = (
    'select reg.id, reg.id_class, reg.id_company, reg.notes, c.name '
    'from classes_reg reg'
    'join companies c on reg.id_company = c.id'
    'where id_class = :id')
  ModifySQL.Strings = (
    'update classes_reg'
    'set'
    '  id = :id,'
    '  id_class = :id_class,'
    '  id_company = :id_company,'
    '  notes = :notes'
    'where id = :old_id')
  GeneratorField.Field = 'id'
  GeneratorField.Generator = 'g_master'
  DataSource = dsClasses
end
```

为了完成对于IBClassReg的讨论，下面给出它惟一的事件处理程序：

```
procedure TDMClasses.IBClassRegAfterInsert(DataSet: TDataSet);
begin
  IBClassReg.FieldByName ('id_class').AsString :=
    IBClasses.FieldByName ('id').AsString;
end;
```

IBPeopleReg数据集具有相同的设置，但IBClasses数据集在设计时更简单。在运行时，该数据集的SQL代码被动态修改，使用三种方法来显示预定的课程，目前已经开始或结束的课程以及过去的课程。用户可以从带有一个标签控件（它为主表格管理着DBGrid）的数据表格中选择三组记录之一，如图14.19所示。

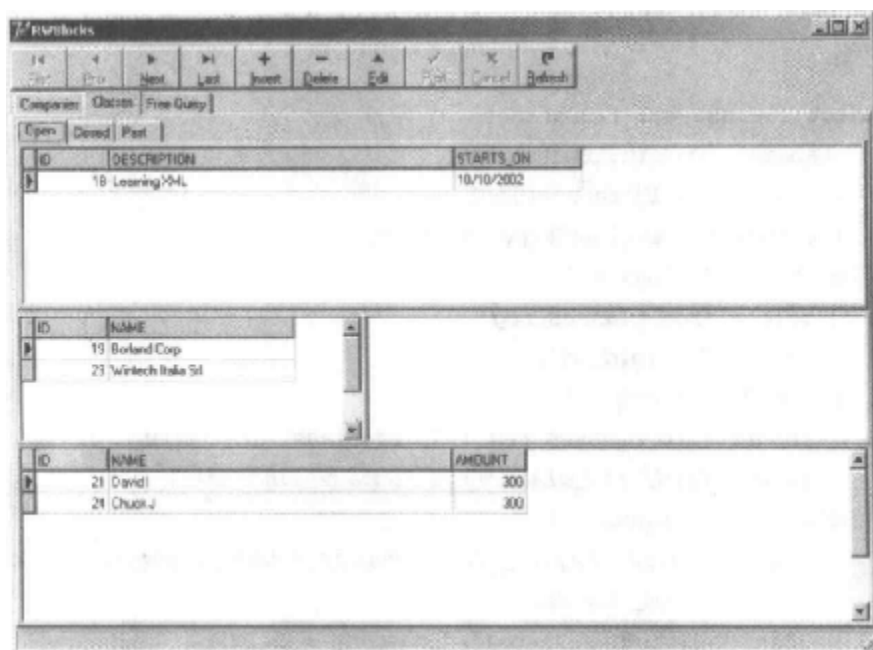


图14.19 用于课程注册的RWBlocks范例

在该程序启动时，或当建立和显示课程注册窗体时，会建立三条SQL语句。而且该程序在一个字符串列表中存储了三条指令（where子句）的最终部分，并在标签改变时选择一条字符串：

```
procedure TFormClasses.FormCreate(Sender: TObject);
begin
    dm := TDMClasses.Create (Self);
    // connect the datasets to the data sources
    dsClasses.Dataset := dm.IBClasses;
    dsClassReg.DataSet := dm.IBClassReg;
    dsPeopleReg.DataSet := dm.IBPeopleReg;
    // open the datasets
    dm.IBClasses.Active := True;
    dm.IBClassReg.Active := True;
    dm.IBPeopleReg.Active := True;

    // prepare the SQL for the three tabs
    SqlCommands := TStringList.Create;
    SqlCommands.Add ( ' where Starts_On > 'now'' );
    SqlCommands.Add ( ' where Starts_On <= 'now'' and ' +
        ' extract (year from Starts_On ) >= extract(year from current_timestamp)'' );
    SqlCommands.Add ( ' where extract (year from Starts_On) < ' +
        ' extract(year from current_timestamp)'' );
```

```

end;
procedure TFormClasses.TabChange(Sender: TObject);
begin
    dm.IBClasses.Active := False;
    dm.IBClasses.SelectSQL [1] := SqlCommands [Tab.TabIndex];
    dm.IBClasses.Active := True;
end;

```

建立一个查找对话框

该课程注册窗体的两个“详”数据集显示了一些查找字段。例如，它不会显示登记课程的公司ID，而是显示公司名称。为此，需要使用SQL语句中的内部连接并配置DBGrdi列，从而不显示公司ID。在一个本地应用程序或带有有限数据的程序中，我们可以使用查找字段。然而，复制整个查找数据集或为了浏览而打开它，应该限制数据表格最多只能有一百个记录，同时嵌入一些查找功能。

如果我们有一个大型的数据表格，如公司表格，则另一种方法是使用二级对话框来进行查找选择。例如，使用已经建立的窗体选择一个公司，并利用它的查找功能。为了将该窗体显示为一个对话框，程序会建立它的一个新实例，显示一些隐藏的按钮（设计时已经存在），并让用户选择一个公司，用于从其他表格中进行引用。

为了简化该查找（它可以在一个大型程序中出现多次）的用法，我们向公司窗体添加了一个类函数，用所选公司的名称和ID作为输出参数。一个初始ID可以被传递给该函数，以确定它的初始选择。下面是该类函数的完整代码，它可建立其类的一个对象，在需要时选择初始记录，显示对话框，最后读取返回值：

```

class function TFormCompanies.SelectCompany (
    var CompanyName: string; var CompanyId: Integer): Boolean;
var
    FormComp: TFormCompanies;
begin
    Result := False;
    FormComp := TFormCompanies.Create (Application);
    FormComp.Caption := 'Select Company';
    try
        // activate dialog buttons
        FormComp.btnCancel.Visible := True;
        FormComp.btnOK.Visible := True;
        // select company
        if CompanyId > 0 then
            FormComp.dm.DataCompanies.SelectSQL.Text :=
                'select c.id, c.name, c.tax_code' +
                ' from companies c ' +
                ' where c.id = ' + IntToStr (CompanyId)
        else
            FormComp.dm.DataCompanies.SelectSQL.Text :=

```

```

        'select c.id, c.name, c.tax_code' +
        ' from companies c ' +
        ' where name_upper starting with 'a''';
FormComp.dm.DataCompanies.Open;
FormComp.dm.DataLocations.Open;
FormComp.dm.DataPeople.Open;

if FormComp.ShowModal = mrOK then
begin
    Result := True;
    CompanyId := FormComp.dm.DataCompanies.FieldByName ('id').AsInteger;
    CompanyName := FormComp.dm.DataCompanies.FieldByName ('name').AsString;
end;
finally
    FormComp.Free;
end;
end;
end;

```

另一个更复杂些的类函数（可以在范例的源代码中找到，这里没有列出）允许选择给定公司的某个人员，为他注册课程。在这种情况下，在禁止查找另一个公司或改变公司的数据之后，才显示窗体。

在这两种情况下，查找都是通过向DBGrid的列添加省略按钮来触发的；例如，网格列中列出了注册课程的公司名称。当单击该按钮时，程序会调用类函数显示对话框，并使用它的结果更新隐含的ID字段和可以看到的名称字段：

```

procedure TFormClasses.DBGridClassRegEditButtonClick(Sender: TObject);
var
    CompanyName: string;
    CompanyId: Integer;
begin
    CompanyId := dm.IBClassReg.FieldByName ('id_Company').AsInteger;
    if TFormCompanies.SelectCompany (CompanyName, CompanyId) then
    begin
        dm.IBClassReg.Edit;
        dm.IBClassReg.FieldByName ('Name').AsString := CompanyName;
        dm.IBClassReg.FieldByName ('id_Company').AsInteger := CompanyId;
    end;
end;

```

添加一个自由查询的窗体

程序的最后一个特性是一个窗体，其中用户可以直接输入一条SQL语句并运行它。作为一个帮助器，该窗体在一个组合框中列出了数据库中可以使用的表格，并在通过调用下列函数建立窗体时获得：

```

DmMain.IBDatabase1.GetTableNames (ComboTables.Items);

```


第15章 使用 ADO

从上个世纪80年代中期开始,数据库程序员就面对着数据库独立性的问题,其理念也就是使用一个API,使应用程序可以同很多不同的数据源交互作用。这样一个API的使用将使开发人员从依赖于单个数据库引擎的局面中解脱出来,并使他们适应社会变化的要求。各厂家为此提出了很多解决方案,早期最有名的两个方案是Microsoft的Open Database Connectivity (ODBC,开放式数据库连接性)与Borland的Independent Database Application Programming Interface (IDAPI,独立数据库应用编程接口),更通俗地称呼为Borland Database Engine (BDE, Borland数据库引擎)。

在20世纪90年代中期,随着COM获得的成功,Microsoft开始用OLE DB代替ODBC。然而,OLE DB被Microsoft归类为系统级的接口,设计用来给系统级的程序员使用。它非常大,非常复杂,容错性差,并且对程序员的要求很高,需要较高水平的知识,因此生产能力较低。ActiveX Data Objects (ActiveX数据对象, ADO)是在OLE DB之上的一个层面,被认为是一种应用程序级的接口。它比OLE DB要简单得多,而且容错性更好。简单地讲,它是为应用程序员设计的。

就像读者在第14章“使用dbExpress的客户机/服务器编程”中已经看到的那样, Borland已经采用一种称为dbExpress的更新的技术取代了BDE。相对于dbExpress而言, ADO与BDE共享了更大的相似性。BDE和ADO支持对数据集合的导航和处理、事务处理、更新缓存(在ADO中称为批更新),所以在ADO涉及的概念和问题都与BDE中的相似。

说明: 我希望感谢最初编写“Mastering Delphi 6”中这一章的Guy Smith-Ferrier先生。他是一个程序员、作者和演讲者,也是一些商业软件产品的作者,为一些独立的公司编写了无数的内部系统。他给Delphi杂志写了许多论文,以及其他一些超出Delphi主题范围的文章,并且经常在北美和欧洲的许多会议上作演讲。Guy现在和他的妻子、儿子和他的猫住在英国。

本章将研究ADO,也将对dbGo进行讨论(这套Delphi组件最初称为ADOExpress,但在Delphi 6中改名为dbGo,因为Microsoft反对在第三方产品名称中使用ADO这个词)。在Delphi中不使用dbGo而是使用ADO也是可以的。通过引入ADO类型库,可以直接访问ADO接口;这实际上就是在Delphi 5之前Delphi程序员使用ADO的方法。然而,这种途径绕过了Delphi的数据库基础结构,并且可以肯定将不能使用其他的Delphi技术,如数据敏感的控件或DataSnap。本章将在所有范例中使用dbGo,不仅因为其非常容易获得并被支持,而且是一个非常可行的方案。不管读者的最终选择是什么,都将发现这里的信息是非常有用的。

说明: 另外,可以借助于Delphi活跃的第三方市场来获得其他ADO套件,如Adonis、AdoSolutio、Diamond ADO与Kamiak等等。

本章主要包括以下内容:

- Microsoft数据访问组件(MDAC)
- Delphi的dbGo
- 数据链接文件

- 获得模式信息
- 使用Jet引擎
- 事务处理
- 不连续性与持续性记录集合
- 公文包模型与配置MDAC

Microsoft数据访问组件 (MDAC)

ADO是Microsoft Data Access Components (Microsoft数据访问组件, MDAC)的一部分。MDAC是Microsoft数据库技术的集合, 包括了ADO、OLE DB、ODBC与RDS (Remote Data Services, 远程数据服务)。读者或许经常听到人们交替地使用术语MDAC与ADO, 其实这并不准确。因为ADO只是作为MDAC的一部分发布的, 所以要依据MDAC的版本来讨论ADO的版本。MDAC的主要版本有1.5、2.0、2.1与2.6。Microsoft独立地发布MDAC, 并且可以免费下载, 并实际地免费发布 (有发布的需要, 但几乎所有的Delphi开发人员都不会在这些需要上遇到麻烦)。MDAC还随着大部分具有各种数据库内容的Microsoft产品一起发布。Delphi 7内置了MDAC 2.6。

这样会产生两个结果。第一, 用户很可能已经在他们的机器上安装了MDAC。第二, 无论用户拥有什么版本, 或程序员升级到什么版本, 实际上有些人——程序员、用户或其他应用程序软件——必然需要将现有的MDAC升级到MDAC的当前版本。这一点是无法改变的, 因为MDAC随着很多常用软件安装, 如Internet Explorer。事实上, Microsoft只支持MDAC的当前版本与之前的版本, 故我们得出这样一个结论: 应用程序必须根据MDAC的当前版本或之前版本设计。

作为一名ADO开发人员, 大家应该经常性地查看Microsoft网站 (www.microsoft.com/data) 上的MDAC页, 从那里免费下载MDAC的最新版本; 在编写本书的时候是MDAC 2.6, 读者可以从www.microsoft.com/data/download_260rtm.htm中下载 (5.2MB), 但首先应该检查更新的版本。当进入该站点时, 如果还没有这个文件或Platform SDK (MDAC SDK是Platform SDK的一部分) 的话, 可以利用这个机会下载MDAC SDK (13MB)。MDAC SDK是各位的“圣经”: 应下载它并经常参考它, 同时使用它来解答ADO问题。在需要MDAC信息时, 应该将它作为首选。

OLE DB供应器

OLE DB供应器能访问一个数据源。它们是相当于dbExpress驱动程序和BDE SQL链接的ADO等效物。安装MDAC时, 将自动地安装表15.1中列出的这些OLE DB供应器:

- ODBC OLE DB供应器用于与ODBC向后兼容。对ADO了解更多以后, 将会发现这个供应器的局限性。
- Jet OLE DB供应器支持MS Access及其他的“桌面”数据库。稍后, 将会继续讨论这些供应器。
- SQL Server供应器支持SQL Server 7、SQL Server 2000和Microsoft Database Engine (MSDE)。MSDE是SQL Server的精简版本, 去除了大部分的工具, 而且当超

过5个活动连接时，会添加一些代码故意降低性能。MSDE是非常重要的，原因有两个。第一，它是免费的。第二，MSDE与SQL Server完全兼容。

表15.1 MDAC中包含的OLE DB供应器

| 驱动程序 | 供应器 | 描述 |
|-------------------------|-----------------|-----------------------|
| MSDASQL | ODBC Drivers | ODBC驱动程序（默认） |
| Microsoft.Jet.OLEDB.3.5 | Jet 3.5 | 只用于MS Access 97数据库 |
| Microsoft.Jet.OLEDB.4.0 | Jet 4.0 | MS Access和其他数据库等 |
| SQLOLEDB | SQL Server | MS SQL Server数据库 |
| MSDAORA | Oracle | Oracle数据库 |
| MSOLAP | OLAP Services | 在线分析处理 |
| SamProv | Sample provider | 用于CSV文件的一个OLE DB供应器范例 |
| MSDAO SP | Simple provider | 为简单的文本数据建立自己的供应器 |

- 用于OLAP的OLE DB供应器可以直接使用，但通常由ADO MultiDimensional (ADOMD) 使用。ADOMD是为提供Online Analytical Processing (OLAP) 而专门设计的一种附加的ADO技术。如果以前用过Delphi的Decision Cube、Excel的Pivot Tables或Access的Cross Tabs，那么就已经使用过OLAP的一些形式了。

除了这些MDAC OLE DB供应器，Microsoft还通过其他产品或可下载的SDK提供了其他一些OLE DB供应器：

- 活动目录服务的OLE DB供应器包含在ADSI SDK中；AS/400和VSAM OLE DB供应器包含在SNA Server中；Exchange OLE DB供应器包含在Microsoft Exchange 2000中。
- 用于索引服务的OLE DB供应器是Microsoft索引服务的一个组成部分，这是Windows通过建立文件信息目录来加快文件查找速度的一种机制。目录服务集成在IIS中，而且通常用于索引网站。
- 用于Internet发布的OLE DB供应器，它允许开发人员使用HTTP处理目录与文件。
- 还有更多的OLE DB供应器以服务供应器的形式出现。正如它们的名称所暗示的那样，OLE DB服务供应器用于向其他OLE DB供应器提供一种服务。通常这些服务供应器会自动按需调用，而不需程序员进行干预。例如，在创建一个客户端光标时，将会调用Cursor Service，并调用Persisted Recordset供应器在本地保存数据。

MDAC还包括许多我们将要讨论到的供应器，但是还有更多可以从Microsoft或第三方市场上获得。要想列出所有可以获得的OLE DB供应器是不太可能的，因为这个列表将会非常巨大，而且会经常变化。除了独立的第三方之外，还应当考虑更多的数据库供应商，因为这些数据库供应商中的大部分都会提供它们自己的OLE DB供应器。例如，Oracle提供的ORAOLEDB供应器。

提示：提供OLE DB供应器的供应商中一个显著的遗漏就是InterBase。除了通过使用ODBC驱动程序访问它之外，我们还可以使用Dmitry Kovalenko的IBProvider (www.lcpi.lipetsk.ru/prog/eng/index.html)。同时也要查看Binh Ly的OLE DB供应器开发工具包 (www.techvanguards.com/products/optk/install.htm)。如果想要编写自己的OLE DB供应器，这个工具将会比大多数的其他工具都易于使用。

使用dbGo组件

对于熟悉BDE、dbExpress或IBExpress的程序员来说，应该可以轻松地确认构成dbGo的组件集合（如表15.2所示）。

表15.2 dbGo组件

| dbGo组件 | 描述 | BDE等效组件 |
|---------------|---------------|------------|
| ADOConnection | 与数据库连接 | Database |
| ADOCommand | 执行一个SQL操作命令 | 没有等效组件 |
| ADODataset | 通用的TDataSet | 没有等效组件 |
| ADOTable | 一个数据表格的封装 | Table |
| ADOQuery | SQL SELECT的封装 | Query |
| ADOStoredProc | 一个存储过程的封装 | StoredProc |
| RDSConnection | 远程数据服务连接 | 没有等效组件 |

四个数据集合组件（ADODataset、ADOTable、ADOQuery、ADOStoredProc）几乎全是被它们直接的父类TCustomADODataset实现的。该组件提供了数据集合的主要功能，而且它的继承者通常是瘦包装器，只是体现了相同组件的不同特性而已。因此，这些组件有很多共同之处。然而，TADOTable、TADOQuery与TADOStoredProc通常被看做是“兼容”的组件，有助于程序员从它们的BDE等效组件转化知识与代码。注意，尽管如此，这些兼容的组件彼此相似但并不完全相同。大家将会在任何应用程序中发现它们的区别。ADODataset是一个可以选择的组件，一部分原因在于它的多功能性，以及它的外表与它基于的ADO Recordset界面比较相似。在本章中，将使用所有的DataSet组件，全面地介绍其各自的用法。

实际范例

理论已经讲的够多了，让我们来看一些具体操作。将一个ADOTable拖到窗体上。ADO使用连接字符串来表示数据集合的连接。如果知道自己正在做什么，就可以手工输入一个连接字符串。通常来说，可以使用连接字符串编辑器（ConnectionString属性的属性编辑器），如图15.1所示。

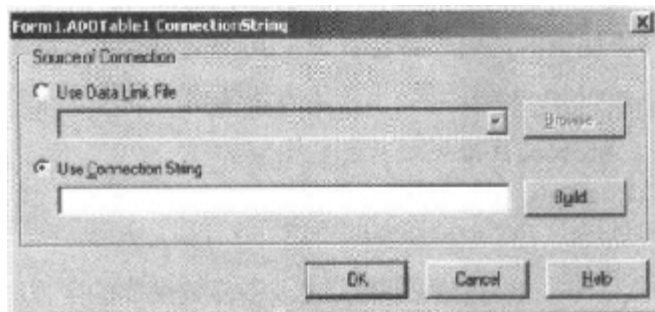


图15.1 Delphi的连接字符串编辑器

该编辑器用于向输入连接字符串的过程添加一些值，所以可以单击**Build**直接进入Microsoft的连接字符串编辑器，如图15.2所示。

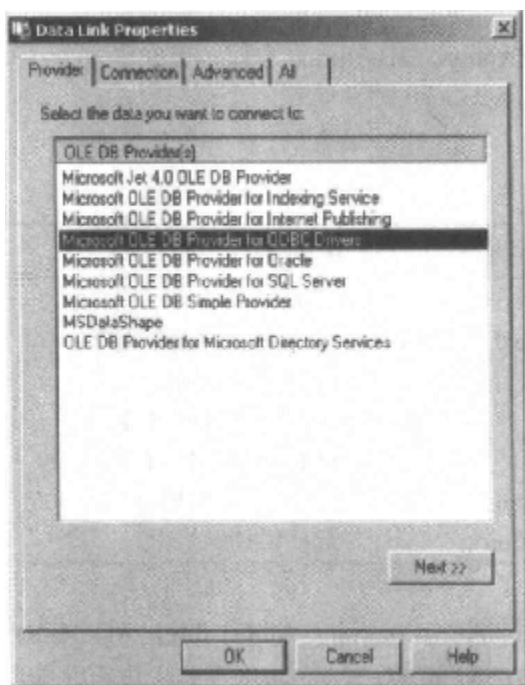


图15.2 Microsoft连接字符串编辑器的首页

这是一种需要大家了解的工具。第一个选项卡显示了安装在计算机上的OLEDB供应器与服务供应器。该列表将根据MDAC的版本以及其他安装在计算机上的软件的不同而有所变化。在这个例子中，选择的是Jet 4.0。双击Jet 4.0 OLE DB供应器，将会看到Connection选项卡。这个页面会根据所选供应器的不同而有所变化；对Jet来说，会要求提供数据库的名称和登陆信息。可以选择一个Access MDB文件，该文件随Delphi 7一起被Borland安装：dbdemos.mdb文件在共享数据文件夹中（默认情况下是C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb）。单击Test Connection按钮，便可以测试所做选择的有效性。

Advanced选项卡负责处理对数据库的访问控制；在这里可以指定对数据库的惟一或只读访问。All选项卡中列出了连接字符串中的所有参数。该列表专门用于在第一页上选择的OLE DB供应器（应该特别关注该页，因为它包含的很多参数是很多问题的答案）。关闭Microsoft连接字符串编辑器之后，将会看到在Borland的连接字符串属性编辑器中，相应的值会返回给ConnectionString属性（这里为了方便阅读，将其分为多行）：

```
Provider=Microsoft.Jet.OLEDB.4.0;  
Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb;  
Persist Security Info=False
```

连接字符串只是一个被分号隔开的、由很多参数组成的字符串。如果想添加、编辑或删除这些参数值，就必须编写子例程查找列表中的参数，并适当地修改它。一个简单的方法就是将该字符串复制到一个Delphi的字符串列表中，并使用它的名称/值；这种技术将在“通过Jet测试文本文件”这一节中用于演示JetText的范例。

既然已经设置了连接字符串,现在就可以选择一个数据表格了。使用对象检验器中的 **TableName** 下拉数据表格列表。选择 **Customer** 表格。添加一 **TDataSource** 与一个 **DBGrid**,并将它们连接起来,现在实际上已经在程序员使用 ADO 了——通过实验(在 **FirstAdoExample** 源代码中可以获得该程序)。要想查看数据,将数据集合的 **Active** 属性设置为 **True** 或者在 **FormCreate** 事件(就像在例子中那样)中打开数据集合,以避免数据库不可用时出现运行时错误。

提示:如果打算将使用 **dbGo** 作为主要的数据库访问技术的话,可能需要将 **DataSource** 组件移动到组件面板的 ADO 页面中,以避免在 **DataAccess** 页与 ADO 页之间来回切换。然而,如果同时使用 ADO 与另一种数据库技术,那么可以通过为 **DataSource** 建立一个组件面板并在 ADO 页上安装它,模拟在多个页上安装 **DataSource**。

ADOConnection 组件

通过这种方法使用 **ADOTable** 组件时,它会在幕后建立自己的连接组件。大家并不一定要接受它创建的默认连接。通常说来,应该使用 **ADOConnection** 组件创建自己的连接,该组件与 **dbExpress SQLConnection** 组件和 **BDE Database** 组件的作用是一样的;允许我们定制登录进程,控制事务,直接执行操作命令,减少在一个应用程序中的连接数目。

ADOConnection 的使用是很简单的。将一个组件放在窗体上并设置它的 **ConnectionString** 属性,方法与 **ADOTable** 相同。另外,还可以双击一个 **ADOConnection** 组件(或使用在其快捷菜单中的组件编辑器的一个特定选项),直接激活连接字符串编辑器。随着将 **ConnectionString** 设置到正确的数据库,就可以通过把 **LoginPrompt** 设置为 **False**,而禁用注册对话框。为了使用前面例子中新的连接,需要把 **ADOTable** 的 **Connection** 属性设置为 **ADOConnection**。大家将看到 **ADOTable1** 的 **ConnectionString** 属性被重置,因为 **Connection** 与 **ConnectionString** 是互斥的。使用一个 **ADOConnection** 的优点之一就是,连接字符串被集中起来了,而不是分散在很多组件中。另一个优点,也是比较重要的优点是,所有共享 **ADOConnection** 的组件将共享一个到数据库的连接。如果没有自己的 **ADOConnection**,则每个 ADO 数据集合都将使用一个独立的连接。

数据链接文件

ADOConnection 允许程序员将连接字符串的定义集中在窗体或数据模块中。然而,即使这与将相同连接分散在所有 ADO 数据集合中相比是一个进步,但它仍有一个基本的缺陷:如果使用一个数据库引擎通过文件名来定义数据库,那么数据库文件的路径在 **EXE** 中是难以编码的。这将会导致应用程序非常脆弱。为了克服这一问题,ADO 使用了 **DataLink** (数据链接) 文件。

一个数据链接文件就是在 **INI** 文件中的一个连接字符串。例如,Delphi 的安装向系统的 **dbdemos.udl** 文件中添加的下列文本:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=C:\Program Files\Common Files\Borland Shared\Data\dbdemos.mdb
```

尽管可以赋给一个数据链接文件任意的扩展名,但推荐的扩展名为UDL。可以使用任意的文本编辑器来创建一个数据链接,或者右键单击Windows Explorer,选择New,然后选择Text Document,以UDL扩展名重新命名文件(假设扩展名在浏览器的配置中被设为显示),然后双击文件调用Microsoft的连接字符串编辑器。

在Connection编辑器中选择一个文件时,其ConnectionString属性将根据实际的文件名设为“FILE NAME=”后跟实际的文件名,就像在DataLinkFile例子中演示的那样。可以将数据链接文件放置到硬盘中的任何地方,但是如果寻找一个通用的、共享的位置,那么可以使用ADODB_Delphi单元中的DataLinkDir功能。如果还没有改变MDAC的默认值,DataLinkDir将会显示如下:

```
C:\Program Files\Common Files\System\OLE DB\Data Links
```

动态属性

假设要由各位负责设计一个新的数据库中间件的结构。为此,必须协调用于所有数据库的单个API的两个相反目标,并访问特定的数据库特性。可以采用设计界面的方法,用该界面汇总创建数据库的所有特性。每个类都具有可以想像到的属性和方法,但它将只能使用所支持的属性与方法。用不着赘述就可知道这不是一个好的方案。ADO必须解决这些显然相互排斥的目标,而且它为此使用了动态属性。

几乎所有ADO界面及其相应的dbGo组件都有一个名为Properties的属性,这是数据库专用属性的集合。这些属性可以按照它们的顺序位置进行访问,如:

```
ShowMessage(ADOTable1.Properties[1].Value);
```

但是更常见的是使用名称来访问它们,如:

```
ShowMessage(ADOConnection1.Properties['DBMS Name'].Value);
```

动态属性依赖于对象的类型,并且也依赖于OLE DB供应器。为了使读者对其重要性有一个概念上的认识,我们告诉大家一个典型的ADOConnection或Recordset会有大约100个动态属性。在本章中,各位将会看到,很多ADO问题的答案都会归结于动态属性。

提示: 与使用动态属性相关的一个重要事件就是OnRecordsetCreate,在Delphi 6中引入,并且可以在Delphi 7中使用。一旦记录集创建了起来,OnRecordsetCreate即可被调用,即使它还没有打开。在记录集合关闭的情况下才可以设置一些动态属性时,这种事件是很有用的。

获得模式信息

在ADO中,可以使用ADOConnection组件的OpenSchema方法提取模式信息。这种方法接受四个参数:

- 第一个参数是OpenSchema应该返回的数据的类型。这是一个TSchemaInfo值,它是一个40个值的集合,从中可提取关于数据表格、索引、列、视图与存储过程的列表。
- 第二个参数是在数据返回前放置在数据上的过滤器。稍后读者将会看到该参数的一个例子。
- 第三个参数是一个供应器专用查询的GUID,只有当第一个参数是siProviderSpecific时,才可以使用这个参数。

- 第四个参数是一个 `ADODataset`，用于接收返回数据。这个参数说明了 ADO 中的一种常见的主题：任何需要返回大量数据的方法将其返回数据作为 `Recordset`，或者用 Delphi 的术语来说就是一个 `ADODataset`。

为了使用 `OpenSchema`，需要一个打开的 `ADOConnection`。下面的代码（`OpenSchema` 范例的一部分）会将一列用于每个表格的主关键字提取到一个 `ADODataset` 中：

```
ADOConnection1.OpenSchema(siPrimaryKeys, EmptyParam, EmptyParam, ADODataset1);
```

主关键字中的每个字段在结果集合中都拥有单独的一行。所以使用两个字段合成关键字的数据表格就会拥有两行。两个 `EmptyParam` 值说明，这些参数是给定的空值，并且会被忽略。在使用一些定制代码调整网格大小之后的这个代码的结果如图 15.3 所示。

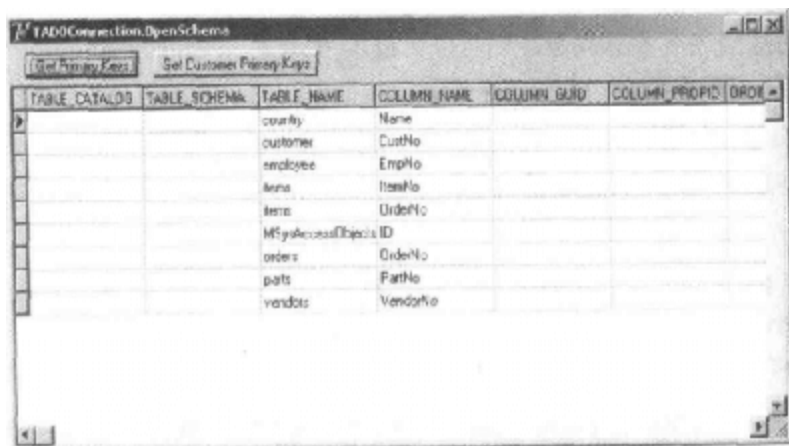


图15.3 OpenSchema范例检索数据表的主关键字

当 `EmptyParam` 作为第二个参数传递时，结果集合会包含整个数据库请求类型的所有信息。对于很多种类的信息来说，需要过滤其结果集合。当然，可以使用 `Filter` 与 `Filtered` 属性或 `OnFilterRecord` 事件，对结果集合应用一个传统的 Delphi 过滤器。然而，在该范例中，这样做会在客户端应用过滤器。使用第二个参数可以在模式信息的源头应用一种更有效的过滤器。该过滤器被指定为一个值的数组。该数组中的每个元素都有一个特定的意义，且与返回的数据种类有关。例如，用于主关键字的过滤器数组有三个元素：第一个是分类表（分类表是数据库的 ANSI 说法），第二个是模式，第三个是数据表格名称。该范例会为 `Customer` 表格返回一列主关键字：

```
var
  Filter: OLEVariant;
begin
  Filter := VarArrayCreate([0, 2], varVariant);
  Filter[2] := 'CUSTOMER';
  ADOConnection1.OpenSchema(
    siPrimaryKeys, Filter, EmptyParam, ADODataset1);
end;
```

说明：可以使用 ADOX 提取相同的信息，而且这保证了 `OpenSchema` 与 ADOX 之间的主要区别。ADOX 是一种附加的 ADO 技术，它允许用户提取与更新模式信息。它在 ADO 中等效于 SQL 的 `Data Defi-`

nition Language (DDL——CREATE、ALTER、DROP) 和Data Control Language (DCL GRANT、REVOKE)。在dbGo中, 并不直接地支持ADOX, 但可以输入ADOX类型库并在Delphi应用程序中成功使用它。遗憾的是, ADOX没有像OpenSchema那样得到普遍的实现, 所以有较大的差距。如果只是需要提取信息而不是更新它, 那么OpenSchema通常是一种更好的选择。

使用Jet引擎

既然我们已经对MDAC与ADO的基础知识有了一定的了解, 现在让我们花一点时间来谈谈Jet引擎。该引擎对于某些人会很有意义, 但对于其他一些人可能没有意义。如果大家对Access、Paradox、dBase、文本、Excel、Lotus1-2-3或HTML感兴趣, 那么本节会对各位有所帮助。如果对这些内容不感兴趣, 则可以跳过本节。

Jet数据库引擎通常是与Microsoft Access数据库相关联的, 而且这确实是它的长处。然而, Jet引擎还是一个通用的桌面数据库引擎, 而且这种鲜为人知的属性也正是其长处的基础所在。因为将Jet引擎用于Access是它的默认模式并且非常简单, 所以本节主要介绍非Access格式的使用, 这并不是如此显而易见的。

说明: Jet引擎已经包含在MDAC的一些(并不是所有)版本中。从MDAC v2.6开始, Jet引擎便从MDAC中脱离出来。关于使用非Microsoft开发工具的程序员是否有权利发布Jet引擎的争论一直在进行。来自官方的答案是积极的, Jet引擎可以免费下载(除了随许多Microsoft的软件产品一起发布之外)。

主要有两个Jet OLE DB供应器: Jet 3.51 OLE DB供应器和Jet 4.0 OLE DB供应器。Jet 3.51 OLE DB供应器使用Jet 3.51引擎并只支持Access 97数据库。如果打算使用Access 97而不是Access 2000, 那么在大多数情况下, 使用这种OLE DB供应器会比使用Jet 4.0 OLE DB供应器获得更好的性能。

Jet 4.0 OLE DB供应器支持Access 97、Access 2000和Installable Indexed Sequential Access Method (IISAM) 驱动程序。Installable ISAM驱动程序是专门为Jet引擎编写的, 用于支持对ISAM格式(如Paradox、dBase与文本)的访问, 而且正是它使Jet引擎如此有用并且功能多样。安装在计算机上的ISAM驱动程序的完整列表要根据安装在计算机上的软件而定。可以通过查看注册表发现该列表:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Jet\4.0\ISAM Formats
```

总之, Jet引擎包含了Paradox、dBase、Excel、文本与HTML的驱动程序。

在Paradox中使用Jet

Jet引擎无疑可以和Access数据库一同使用。为了使任何数据库都能使用它而不仅仅是Access, 需要告诉它使用哪个IISAM驱动程序。这是一个无关痛痒的过程, 包括在连接字符串编辑器中设置Extended Properties连接字符串变元。让我们看一个简单的例子。

将一个ADOTable组件添加到窗体, 并激活连接字符串编辑器。选择Jet 4.0 OLE DB供应器。选择All页面, 选定Extended Properties属性, 并双击它以编辑它的值。

在图15.4所显示的Property Value中输入Paradox 7.x，并单击OK。因为Browse按钮不能为用户提供帮助，所以现在要返回到Connection选项卡并直接输入包含Paradox表格的目录名称（要求输入的是一个文件名，而不是文件夹名）。这时，可以在ADOTable的TableName中选择一个表格，并在设计时或运行时打开它。就像JetParadox范例中所演示的那样，现在正在通过ADO使用Paradox。

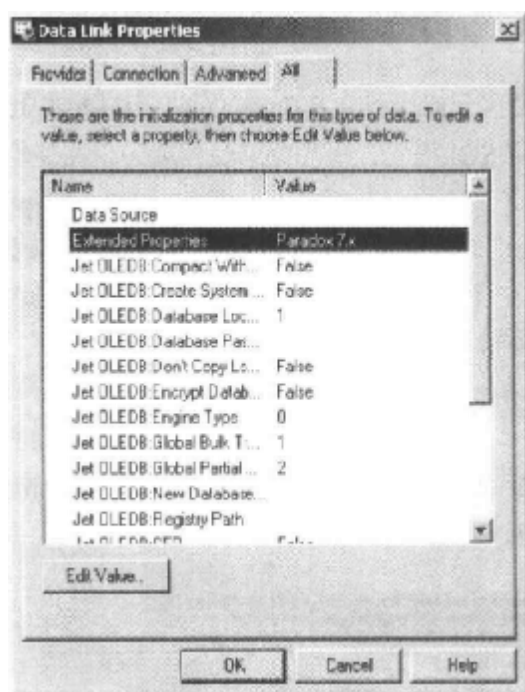


图15.4 设置扩展属性

很遗憾，我有一些坏消息带给Paradox的用户：在特定环境下，除了Jet引擎以外，还需要安装BDE。Jet 4.0需要使用BDE以便能够更新Paradox表格，但它不要求BDE来读取它们。这对于大多数的Paradox ODBC Driver版本也是一样的。Microsoft已承认在这点上的失误，并且制作了一种新的不需要BDE的Paradox IISAM；大家可以通过Microsoft Technical Support获得这些更新的驱动程序。

说明：对ADO了解的越多，就会发现ADO是如何依赖于OLE DB供应器和RDBMS（关系数据库管理系统）的。尽管可以以一种本地文件的格式使用ADO，正如在本范例和后面例子中所示的那样，但通常的建议是在可能的情况下安装一个本地SQL引擎。如果必须使用ADO，则Access和MSDE是个好的选择；否则，可能需要考虑InterBase或Firebird作为可替换的方案，就像在第14章中讨论的那样。

在Excel中使用Jet

使用Jet OLE DB供应器可以很容易地访问Excel。再次重申，可以将Extended Properties属性设置为Excel 8.0。假设有一个名为ABCCompany.xls的Excel电子表格，表单名为Employees，并且想要使用Delphi打开并读取这个文件。当然，可以通过使用自己所知道的一些COM知识自动操作Excel以解决该问题。但是，ADO的解决方案会更容易实现这个操作，并且不需要在计算机上运行Excel。

提示：你也可以使用XLSReadWrite组件读取一个Excel文件（该组件可以从www.axolot.com获得）。它既不需要在计算机上安装Excel程序，也不需要运行Excel（就像OLE Automation技术那样）。

因为ADO要求惟一地访问文件，所以应确保电子数据表格没有在Excel中打开。向窗体中添加一个ADODataset组件。设置ConnectionString以使用Jet 4.0 OLE DB供应器，并设置Extended Properties为Excel 8.0。在Connection选项卡中，数据库的名字应设置为Excel电子数据表的完整路径和文件名（如果打算与程序一起部署这个文件，则可以使用相对路径表示）。

ADODataset组件通过打开或执行在其CommandText属性中的一个值进行工作。该值可能是一个表格的名称、一条SQL语句、一个存储过程或一个文件的名称。大家可以通过设置CommandType属性来指定该值如何被解释。设置CommandType为cmdTableDirect来说明CommandText中的值是表格的名称，而且所有的栏将从该表格返回。在对象检验器中选择CommandText，将看到一个下拉箭头。下拉这个箭头，一个单独的“伪表格”将显示出来：Employees\$（Excel工作簿需后缀一个\$）。

添加一个DataSource和一个DBGrid，并将它们连接在一起，便会看到JetExcel范例在设计时的输出，如图15.5所示。默认情况下，在网格中浏览数据会有些困难，因为每个栏的宽度是255个字符。可以通过向网格中添加栏目并改变它们的Width属性，或者添加持续性字段并改变它们的Size或DisplayWidth属性来改变字段的显示大小。



图15.5 Delphi中的ABCCompany.xls——献给道格拉斯·亚当斯的小礼物

注意，不能在设计时保持数据集合打开并运行这个程序，因为Excel IISAM驱动程序是在惟一的模式下打开XLS文件的。关闭数据集合并向程序中添加一行代码，就可以在开始时打开它。当运行这个程序时，将会注意到该IISAM驱动程序的另一个限制：可以添加新的行并编辑现有的行，但是并不能删除行。

顺便说一下，大家可以使用一个ADOTable组件或者使用一个ADOQuery组件来代替TADODataset，但需要知道ADO是怎样在诸如表格名称和字段名称这样的地方对待符号的。如果使用一个ADOTable并下拉该表格的列表时，将会看到所期待的Employees\$表格。遗憾的是，如果试图打开表格，就会收到一个错误。对于在TADOQuery中使用语句SELECT * FROM Employees\$来说，也会有同样结果。如果使用诸如美元符号、点或其他更重要的符号隔开表格名称或字段名称的话，那么必须用方括号封闭名字（如[Employees\$]）。

在文本文件中使用Jet

随Jet引擎一起的一个非常有用的IISAM驱动程序就是Text IISAM。这个驱动程序允许用户读取并更新几乎是任意结构格式的文本文件。我们将从简单的文本文件开始，然后介绍各种变化。

假设有一个名为NightShift.TXT的简单文本文件，该文本文件包含下列文本：

```
CrewPerson ,HomeTown
Neo        ,Cincinnati
Trinity    ,London
Morpheus   ,Milan
```

向窗体添加一个ADOTable组件，设置它的ConnectionString组件以使用Jet 4.0 OLE DB 供应器，并且设置Extended Properties为Text。由于Text IISAM供应器将一个目录做为数据库，因此我们需要输入包含NightShift.TXT文件的目录作为数据库的名字。在对象检验器中，在TableName属性上下拉表格列表。注意，文件名中的点号已转换为无用信息，例如NightShift#TxT。将Active设置为True，添加一个DataSource和TDBGrid并将它们连接到一起，即可在一个网格中看到文本文件的内容。

警告：如果你的计算机的设置是这样的，即十进制分隔符是用句号而不是用逗号表示（如1,000.00被显示为1.000,00），那么你将需要改变你的Regional Settings（Start>Settings>Control Panel>Regional Settings>Numbers），或者利用SCHEMA.INI，随后将会介绍。

当然，网格表明栏的宽度是255个字符。可以像在JetExcel上那样通过对网格添加持续字段或栏，然后设置有关的宽度属性来改变这些设置。或使用SCHEMA.INI这种更确定的定义文本文件的结构。

在JetText范例中，根据文件夹所包含的程序的不同，可以在运行时确定数据集合文件夹。要想在运行时修改连接字符串，首先要将其装载入一个字符串列表（在转换分离器之后），然后使用Value属性只修改连接字符串的一个元素。下面是该范例中的代码：

```
procedure TForm1.FormCreate(Sender: TObject);
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  sl.Text := StringReplace (ADOTable1.ConnectionString,
    ';', sLineBreak, [rfReplaceAll]);
  sl.Values ['Data Source'] := ExtractFilePath (Application.ExeName);
  ADOTable1.ConnectionString := StringReplace (sl.Text,
    sLineBreak, ';', [rfReplaceAll]);
  ADOTable1.Open;
  sl.Free;
end;
```

文本文件可能会以各种形状和大小出现。通常大家不需要担心一个文本文件的格式，因为文本IISAM会查看前25行，看它是否能确定自己的格式。因此，它要使用这个信息和注

册表中的一些附加信息来决定怎样解释这个文件及其行为是怎样的。如果我们有一个文件，该文件与Text IISAM所能确定的常规格式不相匹配，那么可以通过与其引用的文本文件位于相同目录下的SCHEMA.INI文件提供该信息。这个包含在相同目录中的任意或所有文本文件的模式信息也称为元数据。每个文本文件都有其自己的部分，以文本文件的名称加以识别，如[NightShift.TXT]。

随后可以指定文件的格式：名称、类型和栏的尺寸；使用的任意特殊字符集；以及任意特殊栏格式（如日期/时间、货币）。我们假设将NightShift.TXT文件改变为下列格式：

```
Neo          ,Cincinnati
Trinity      HLondon
Morpheus     IMilan
```

在该示例中，栏的名称不包含在文本文件中，并且分隔符是一个垂直条。一个相关的SCHEMA文件看起来可能和下面所列的有些相似：

```
[NightShift.TXT]
Format=Delimited(1)
ColNameHeader=False
Col1=CrewPerson Char Width 10
Col2=HomeTown Char Width 30
```

不管大家是否使用SCHEMA.INI文件，都将遇到Text IISAM的两个限制：行不能被删除，并且行不能被编辑。

导入和导出

Jet引擎特别擅长于导入和导出数据。对于每个导出格式导出数据的过程是相同的，并由用特殊语法执行的一个SELECT语句组成。让我们先从DBDemos数据库的Access版本向一个Paradox表格导出数据的示例开始。在JetImportExport示例中，需要一个称为ADOConnection1的活动的ADOConnection，它使用Jet引擎打开该数据库。下面的代码将把Customer表格导出到一个Paradox Customer.db文件中：

```
SELECT * INTO Customer IN "C:\tmp" "Paradox 7.x;" FROM CUSTOMER
```

让我们看一下该SELECT语句。INTO子句指定了新的表格，该表格将被SELECT语句创建；它不能是已存在的表格。IN子句指定了添加新表格的数据库；在Paradox中，这是一个已经存在的目录。紧随数据库的子句是IISAM驱动程序的名字，用于执行导出。我们必须在驱动程序名字的最后包含后缀分号。FROM子句是任意SELECT语句的一个常规部分。在范例程序中，该操作通过ADOConnection组件执行，并且需要使用该程序的文件夹，而不是一个固定的文件夹：

```
ADOConnection1.Execute ('SELECT * INTO Customer IN "' +  
    CurrentFolder + '" "Paradox 7.x;" FROM CUSTOMER');
```

所有的导出语句都遵循这些相同的基本子句，尽管有些IISAM驱动程序对什么是数据库会有不同的解释。下面，把相同的数据输出到Excel中：

```
ADOCConnection1.Execute ('SELECT * INTO Customer IN "' +  
CurrentFolder + 'dbdemos.xls' "Excel 8.0;" FROM CUSTOMER');
```

一个称为dbdemos.xls的新的Excel文件在应用程序的当前目录中被创建。一个称为Customer的工作簿也被添加，该工作簿包含dbdemos.mdb中Customer表格的所有数据。

最后一个示例把相同数据导出到一个HTML文件中：

```
ADOCConnection1.Execute ('SELECT * INTO [Customer.htm] IN "' +  
CurrentFolder + '" "HTML Export;" FROM CUSTOMER');
```

在这种情况下，数据库是目录，因为它是用于Paradox的，而不是用于Excel的。因此表格名字必须包含.html扩展名，并且它必须被封在方括号内。注意，IISAM驱动程序的名称是“HTML Export”而不是“HTML”，因为该驱动程序只能用于导出HTML。

在对于Jet引擎的研究中，我们将看到的最后IISAM驱动程序是HTML Export的伙伴：HTML Import。将一个ADOTable添加到一个窗体中，设置它的ConnectionString为使用Jet 4.0 OLE DB供应器，设置它的扩展属性为HTML Import。设置数据库的名字为前面导出所创建的HTML文件的名字——即Customer.htm。现在将TableName属性设置为Customer。打开表格——这才刚刚导入HTML文件！记住，如果试图以任意方式更新数据，将收到一个错误，因为该驱动程序只用于导入。最后，如果创建的HTML文件包含表格，并且需要使用该驱动程序打开这些表格，那么要记住，表格的名字应该是HTML table的caption标记的值。

光标

ADO数据集合有两个属性，会对应用程序产生基础性的影响，并且不可分割地被互相链接在一起，这两个属性分别是：CursorLocation和CursorType。如果想要理解数据集合的行为，就必须理解这两个属性。

光标位置

CursorLocation属性允许程序员指定谁来控制数据的提取和更新。有两个选择：客户机（clUseClient）或服务器（clUseServer）。所做的选择将会影响数据集合的功能、性能和可扩展性。

一个客户端光标是由ADO Cursor Engine负责管理的。该引擎是一个OLE DB服务供应器的优秀示例：它为其他的OLE DB供应器提供一个服务。ADO Cursor Engine管理来自应用程序客户端的数据。当数据集合打开时，结果集上的所有数据将从服务器提取到客户端。从此数据被保存在内存中，并且由ADO Cursor Engine管理更新和操作。这一点和在dbExpress应用程序中使用ClientDataSet组件是很相似的。初始检索之后，为处理数据带来的一个好处就是相当的快。而且，因为操作是在内存中执行的，ADO Cursor Engine比多数服务器端光标拥有更强大的能力并能提供额外的功能。稍后将介绍这些好处以及其他一些技术，这些技术取决于客户端光标（如分离和持续的记录集）。

服务器端的光标是由RDBMS负责管理的。在一个基于客户端/服务器结构的数据库如SQL Server、Oracle或InterBase中，这就意味着光标实质上是在服务器上被管理的。在一个桌面数据库如Access或Paradox中，“服务器”的位置实际上是一个逻辑的位置，因为数据库

就运行在桌面。服务器端的光标通常会比客户端光标装载得更快，这是因为当数据集合被打开时，并不是所有的数据都会被传到客户端。这种行为也使它们更适合于非常大的结果集合。在这种情况下，客户端没有足够的内存来保存整个结果集合。通常可以通过思考光标是怎样工作的来决定每一个光标位置上可用的功能。锁定是一个好的范例，该范例可以解释特性是怎样确定光标类型的。随后将会详细地介绍锁定（为了在记录上放置一个锁，需要有一个服务器端的光标，这是因为在应用程序和DBMS之间必须有一个会话）。

将会影响光标位置选择的另一个问题就是可扩展性。服务器端的光标是由RDBMS管理的；在一个客户端/服务器数据库上，这将被定位在服务器上。当越来越多的用户使用应用程序时，服务器的负载将会随着每个服务器端光标而增长。服务器上更大的工作量意味着DBMS将很快地成为一个瓶颈，因此应用程序的扩展性就会很差。可以通过使用客户端光标以获得更好的可扩展性。在打开光标时，最初的负载是比较重的，因为所有数据都被传到客户机上，但打开光标的维护量是很低的。正如人家所看到的，在为数据集合选择正确的光标位置时将会涉及到许多矛盾的问题。

光标类型

对光标位置的选择将会直接地影响到对光标类型的选择。考虑到各种意图和目的，共有四种光标类型，但是其中有一个值是没有用的：一个意味着“不确定”的光标类型。在ADO中的许多值都表示一个不确定的值，在这里将对它们进行介绍，并解释为什么不需要更多地处理它们。因为它们在ADO上存在，所以它们在Delphi中存在。ADO是为Visual Basic和C程序员设计的。在这些语言中，可以直接使用对象而不需要dbGo提供任何援助。这样，可以像在ADO中调用一样创建并打开记录集合，而不需要为每个属性指定每个值。没有被指定值的属性有一个不确定的值。然而，在dbGo中要使用组件。这些组件都有构造器并且这些构造器都可以初始化组件的属性。因此，从创建一个dbGo组件时起，它通常会有一个用于组件的每个属性的值。该结果就是在许多枚举类型中不需要这种不确定的值。

光标类型会影响数据是怎样被读取和被更新的。就像前面提到的那样，共有四种选择：只向前光标、静态光标、键集光标和动态光标。在我们涉及光标位置和光标类型的所有置换之前，大家应意识到对客户端的光标来说，只有一种光标类型有用：静态光标。所有其他光标类型只对服务器端光标有用。在我们看过各种光标类型后，将返回到光标类型可用性的主题中。

只向前光标 花费最少并且有最佳性能的光标类型是只向前光标，正如其名称所暗示的那样，只向前光标将向前导航。光标读取被CacheSize指定的记录的数目（默认为1）；每次读取完记录后，它将读取另一个CacheSize集合。任何企图通过使用超出结果集合记录数的值来实现向后移动都会产生一个错误。这种行为与dbExpress数据集合的行为很相似。一个只向前光标是并不适合用于用户接口的（用户可以在该用户接口控制通过结果集合的方向）。但是，它非常适合于批操作、报告和无状态Web应用程序，这是因为这些情况从结果集合顶部开始，并逐步向结尾移动，然后结果集合关闭。

静态光标 静态光标的工作就是读取全部的结果集合并提供一个CacheSize记录的窗口到结果集合中。因为全部的结果集合已被服务器提取，故可以在结果集合中向前和向后移动。然而，作为该功能的交换，数据是静态的——因此其他用户所做的更新、插入

和删除等操作不能被看到，这是因为光标的数据已经被读取。

键集光标 把keyset这个词分解成key和set这两个词会对键集光标的理解更深刻些。key这个词在这里指的是一个用于各行的标识符。通常它是一个主键值。因此键集光标就是一个键的集合。当结果集合打开时，用于结果集合的键的整个列表就会被读取。例如，如果数据集合是一个像SELECT * FROM CUSTOMER这样的一个查询，那么键列表将从SELECT CUSTID FROM CUSTOMER中建立。这个键的集合可保持到光标关闭时为止。当应用程序请求数据时，OLE DB供应器将使用键集中的键读取行。因此，数据总是最新的。如果另一个用户在结果集合中改变了一行，那么当数据被重读时，则改变将会被看到。然而，键集本身是静态的，只有当结果集合是第一次打开时，它才会被读取。因此如果另一个用户添加了新的记录，这些增加将不会被看到。删除的记录将不能接受访问，并且对主键值的改变（你不会让你的用户改变主键值，对吗？）也将无法被访问。

动态光标 开销最多的光标类型是动态光标，一个动态光标也几乎和键集光标是相同的。惟一的不同是当应用程序请求的数据不在缓冲区上时，键集将被重读。因为用于ADODataset.CacheSize的默认值为1，因此这种行为将会非常频繁地发生。可以想像，这种行为在DBMS和网络上所产生的额外的负载有多大，以及为什么这是开销最大的光标。然而，结果集合可以看到并响应其他用户所做的增加和删除。

要求与结果

现在我们知道光标位置和光标类型，要提醒大家的是：并不是所有光标位置和光标类型都能结合在一起。通常，这种限制是由RDBMS和/或OLE DB供应器施加的，是数据库功能性和结构的一个结果。例如，客户端的光标总是强制其光标类型为静态的。读者可以自己看到这种行为。向一个窗体中添加一个ADODataset组件，并将其ConnectionString设置到任何数据库，设置ClientLocation为clUseCursor并且设置CursorType为ctDynamic。现在设置Active为True并注意CursorType，它将改变为ctStatic。从该示例大家学到重要一课：所要求的并不一定要是所得到。打开一个数据集合之后，需要经常检查一下其属性，看一下是否是自己实际所要求的效果。

每个OLE DB供应器将依据不同的请求和环境做出不同的改变，但在这里将会给出几个例子，以便读者对此能有一个简单的概念：

- Jet 4.0 OLE DB供应器将大多数光标类型改变为键集光标。
- SQL服务器的OLE DB供应器经常将键集光标和静态光标改变为动态光标。
- Oracle的OLE DB供应器将所有的光标类型改变为只向前光标。
- ODBC的OLE DB供应器依据使用中的ODBC驱动程序做出各种改变。

无记录计数

有时，ADO数据集合的RecordCount的返回值为-1。一个只向前光标只有到达最后一个记录以后才能知道有多少条记录在结果集合中，因此它为RecordCount返回-1。一个静态光标通常会知道在结果集合中有多少个记录，这是因为当集合打开时，它会读取整个集合，所以它会在其结果集合中返回记录数。一个键集光标通常也知道在结果集中有多少个记录，

这是因为当结果集合打开时，它必须提取键值的一个固定集合，所以它也为RecordCount返回一个有用的值。一个动态光标不能很确定地知道在结果集合中有多少记录，这是因为它有规律地重读键集，所以它返回-1。当然，可以完全避免使用RecordCount并执行SELECT COUNT(*) FROM table，但这个结果将是数据库表格中记录数目的准确反映——它没必要和数据集合中的记录数一样。

客户端索引

客户端光标的众多好处之一就是它能创建本地或客户索引。假设有一个用于DBDemos Customer表格的ADO客户端数据集合，该表格中附加有一个网格，而且已将数据集合的IndexFieldNames属性设置为CompanyName。网格将立即以CompanyName的顺序显示数据。很重要的一点要做的是：为了索引数据，ADO不必从数据的源重读数据。索引是从内存中的数据创建的。这意味着不仅能尽可能快地创建索引，而且能使网络和DBMS在以不同的顺序一次次地传递相同的数据时不会超负荷。

IndexFieldNames属性有更大潜力。将它设置为Country;CompanyName，大家会看到数据首先按国家名称进行排序，在同一国家内，按公司名排序。现在设置IndexFieldNames为CompanyName DESC。确保用大写书写DESC（而不是“desc”或“Desc”）。数据现在按降次顺序被保存。

这个简单但有力的特征为数据库开发人员解决了一个大问题。用户好像总是问不可避免又十分合理的问题：“我能单击网格的栏来对我的数据分类吗？”可以这样回答：用非数据敏感控件，如这种分类已经嵌入其中的ListView，代替所有网格；或者触发DBGtid的OnTitleClick事件，并在包含一个合适的ORDER BY子句之后重新提出SELECT语句。然而，诸如此类的答案是远远不能令人满意的。

如果有数据缓存在客户端（就像读者在ClientDataSet组件的使用中已经看到过的那样），就可以使用一个在内存中计算的客户端索引。将下列OnTitleClick事件添加到网格中（该代码可以从ClientIndexes范例中获得）：

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  if ADODataSet1.IndexFieldNames = Column.Field.FieldName then
    ADODataSet1.IndexFieldNames := Column.Field.FieldName + ' DESC'
  else
    ADODataSet1.IndexFieldNames := Column.Field.FieldName
end;
```

这个简单的事件将会检查当前的索引是否和栏建立在相同的字段上。如果是，将会在栏上建立一个新的索引，但是会以降序排列。如果不是，将会在栏上建立一个新的索引。当用户第一次单击栏时，它将按升序排列分类；当用户第二次单击它时，它将按降序排列分类。可以扩展该功能，以允许用户使用Ctrl+click组合键选择多个栏的标题，这样就可以建立更复杂的索引了。

说明：通过对ClientDataSet的使用，所有这些均可以获得，但解决方法之所以不是如此优秀的原因在于：ClientDataSet不支持DESC关键字，所以必须创建一个索引集合条目，以获得一个按降序

排列的索引，这样做也需要更多的代码。而且将一个按升序排列的索引改为一个按降序排列的版本时，ClientDataSet将会执行一个非必需的、可能很慢的操作，以重建这个索引。

复制

ADO拥有太多的特性。可以把“feature-rich（多功能）”理解成“footprint-rich（内容丰富）”，但也能理解为更有力更可靠的应用程序。这样有力的一个特性就是复制。一个复制的记录集合是一个新的记录集合，不过它与被复制的原始记录集合拥有完全相同的属性。首先解释一下可以怎样创建和使用一个复制品，然后解释为什么它们是如此有用。

说明：ClientDataSet也支持复制；这种特性在第13章“Delphi的数据库体系结构”中并没有讨论。

可以使用Clone方法复制一个记录集合（或者按dbGo说法是一个数据集合），以及任意的ADO数据集合，在这个示例中将使用ADOTable。DataClone范例（参见图15.6）使用了两个ADOTable组件，一个连接到数据库的数据上，另一个是空的。这两个数据集合都连接到一个DataSource和网格上。当用户单击一个按钮时，下面这一行代码将会执行，用来复制这个数据集合：

```
ADOTable2.Clone(ADOTable1);
```

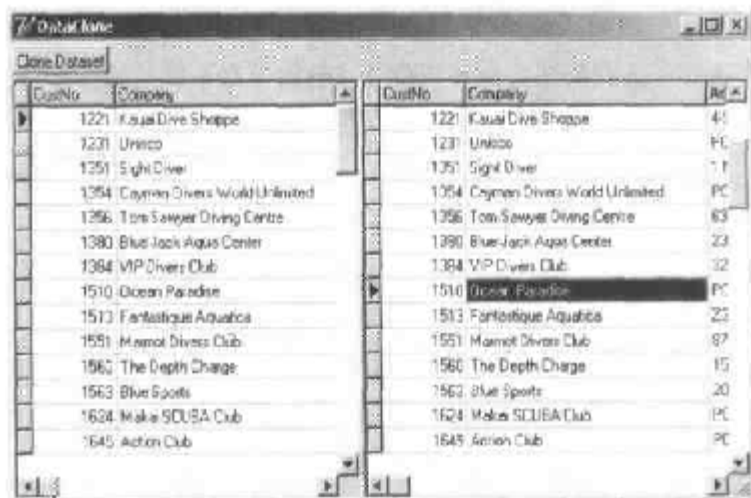


图15.6 DataClone范例的窗体，含有数据集的两个拷贝（源和复制）

该行代码可将ADOTable1复制到ADOTable2。在这个程序中，将看到数据的第二个视图。这两个数据集合都有它们自己的记录指针和其他状态信息，因此复制不会影响到其最初的副本。这种行为使复制操作非常适用于数据集合，而且不会影响到其原始数据。另一个有趣的特性就是，可以有多个不同的活动记录，每个复制一个——在Delphi中用一个单独的数据集合将不能获得正常的功能。

提示：为了能够被复制，记录集合必须支持书签，因此只向前光标和动态光标不能复制。可以确定一个记录集合是否能够通过使用Support方法支持书签（例如ADOTable1.Supports([coBookmark])）。复制的一个有用的效果就是：由一个复制品所创建的书签可以被其他所有的复制品使用。

事务处理

就像我们在第14章中的“使用事务”一节所看到的那样，事务处理允许开发人员将对数据库的单个的更新组织起来，形成一个单个的工作逻辑单元。

ADO的事务处理支持被ADOConnection组件控制——使用BeginTrans、CommitTrans、RollbackTrans等方法，其效果与在dbExpress和BDE中对应的方法相似。为了深入研究ADO事务处理支持，我们将建立一个简单的测试程序，称为TransProcessing。这个程序有一个ADOConnection组件，其ConnectionString设置为Jet 4.0 OLE DB供应器和dbdemos.mdb文件。这是一个ADOTable组件，连接到Customer表、DataSource和用于显示数据的DBGrid。最后，有三个按钮用于执行下列命令：

```
ADOConnection1.BeginTrans;  
ADOConnection1.CommitTrans;  
ADOConnection1.RollbackTrans;
```

通过这个程序，对数据库表格做一些改变，然后退回，便可以看到它们将会像我们期望的那样退回。之所以强调这一点是因为，事务支持会根据数据库和人们所使用的OLE DB供应器的不同而变化。例如，如果我们使用ODBC OLE DB供应器连接到Paradox上，将会收到一条错误消息，告诉我们该数据库或OLE DB供应器不能开始一个事务。可以发现使用连接的Transaction DLL动态属性时，各自所拥有事务处理支持的等级：

```
if ADOConnection1.Properties['Transaction DDL'].Value > DBPROPVAL_TC_NONE then  
    ADOConnection1.BeginTrans;
```

如果使用Jet 4.0 OLE DB供应器试图访问相同的Paradox数据，将不会收到错误消息，但是也不能退回修改。这是由于OLE DB供应器的限制所造成的。

当使用Access时，会出现另一个奇怪的区别：如果使用ODBC OLE DB供应器，将能够使用事务，但不能使用嵌套事务。在一个事务激活的情况下打开另一个事务将会导致错误。通过使用Jet引擎，Access将会支持嵌套事务。

嵌套事务

通过TransProcessing程序，可以试试这个测试：

1. 开始一个事务。
2. 将Around The Horn记录的ContactName从Thomas Hardy改变为Dick Solomon。
3. 开始一个嵌套事务。
4. 将Bottom-Dollar Markets记录的ContactName从Elizabeth Lincoln更改为Sally Solomon。
5. 转回到内部事务。
6. 提交最外层的事务。

实际结果是只有对Around The Horn记录的改变是永久的。然而，如果内部事务已提交并且外部事务被转回，那么实际结果将是没有一个改变会持久（甚至于内部事务的改变）。

这正是我们所期望的，惟一的限制是Access支持五级嵌套事务。

ODBC不支持嵌套事务，Jet OLE DB供应器最多支持五级嵌套事务，SQL Server OLE DB供应器根本就不支持嵌套事务。根据使用的SQL服务器或驱动程序的版本不同，大家可能会得到不同的结果。但是根据文档和我对服务器的经验看起来，这是一个问题。很明显，只有最外层事务才可以决定所有的工作是被提交还是被转回。

ADOConnection属性

如果试图使用嵌套事务，还应考虑另外一个问题。ADOConnection组件有一个属性叫做Attributes，它可以决定当事务提交或转回时，连接应该怎样行动。它是TXActAttributes的一个集合，默认情况下是空的。TXActAttributes中只含有两个值：xaCommitRetaining和xaAbortRetaining（这个值经常被错误地写做xaRollbackRetaining——因为这对它可能是更合理的名字）。当xaCommitRetaining包含在Attributes中并且一个事务被提交时，一个新的事务将会自动开始。当xaAbortRetaining包含在Attributes中并且一个事务被转回时，一个新的事务也将会自动开始。这意味着如果在Attribute中包含这些值，事务将总是在进行中，因为结束一个事务，总是开始另一个事务。

大多数程序员更愿意对他们的事务进行更大程度的控制，而不允许它们自动开始，因此这些值通常是不被使用的。然而，它们对嵌套事务有一个特殊的作用。如果嵌套一个事务并将Attributes设置为[xaCommitRetaining、xaAbortRetaining]，那么最外层的事务将永远不会结束。考虑事件的这种顺序：

1. 一个外部事务被启动。
2. 一个内部事务被启动。
3. 一个内部事务被提交或转回。
4. 一个新的内部事务被自动启动作为Attributes属性的结果。

最外层事务从不结束，这是因为当一个事务结束时，另一个新的事务将会启动。我们的结论就是，Attributes属性的使用和嵌套事务的使用将被认为互斥。

锁定类型

ADO支持四种不同的方法来锁定用于更新的数据，它们是：ltReadOnly、ltPessimistic、ltOptimistic和ltBatchOptimistic（当然还有一个ltUnspecified，但由于前面提到的原因，我们将忽略“不确定”值）。通过数据集合的LockType属性可以获得这四种方法。在这一节中，我将对这四种方法做一个简单的概述，随后各小节将详细地对它们进行介绍。

ltReadOnly的值指定数据是只读的，并且不能更新。因此，不需要锁定控件是非常有效的，这是因为数据不能更新。

ltPessimistic与ltOptimistic的值会提供“pessimistic（消极）”与“optimistic（积极）”锁定控制，与BDE提供的相同。在这方面，ADO与BDE相比，一个重要的优点是为开发人员保留了锁定控制的选择。如果使用BDE，那么实行消极锁定还是积极锁定由BDE驱动程序决定。如果使用诸如dBase或Paradox这样的桌面数据库，那么BDE驱动程序将实行消极锁定；如果使用InterBase、SQL Server或Oracle这样的客户机/服务器数据库，BDE驱动程序将实行积极锁定。

消极锁定

在这里的上下文中，消极与积极这两个词指的是开发人员对用户更新之间产生冲突的期望值。消极锁定就是假设将有很大的可能性，用户会试图同时更新相同的记录，并可能出现冲突。为了防止这样的冲突，当编辑开始时记录将被锁定。记录锁定会一直延续到更新完成或取消时为止。第二个试图同时编辑相同记录的用户将不能成功锁定其记录，并会接收到一条“不能更新，当前记录已被锁定”的异常。

对于这种锁定方法，使用桌面数据库（如dBase与Paradox）的开发人员一定会很熟悉。这种方法的优点是，用户知道他们是否可以开始编辑一个记录，然后是否会成功保存他们的更新。消极锁定的缺点是，用户在放置锁定与解除锁定的时间上受到控制。如果用户熟练掌握了应用程序，那么这可能只是几秒钟的事情。然而，在数据库方面，几秒钟可能意味着永远。比如，用户可能开始了一个编辑，然后去吃午饭，而记录会一直被锁定到用户返回为止。因此，大多数消极锁定的支持者都通过使用一个Timer或其他类似的设备，在键盘与鼠标休止一定的时间后解除锁定，以防止这种可能性的发生。

消极锁定带来的另一个问题就是，它需要一个服务器端的光标。前面我们曾讨论过光标位置，并且看到它们对不同光标类型的使用会有影响。现在，还可以看到，光标位置还会对锁定类型产生影响。在本章稍后，我们将讨论更多的客户端光标的优点；而且如果选择要利用这些优点的话，那么就不能使用消极锁定。

消极锁定是dbGO在Delphi 6中发生了变化的一部分（相对于Delphi 5）。本节将描述消极锁定在Delphi版本6和版本7中的使用方法。为了更好地说明其是怎样工作的，我将建立一个PessimisticLocking范例。该范例与本章中的其他范例相似，但是其CursorLocation属性被设置为clUseServer，其LockType属性被设置为ltPessimistic。为了使用它，需要使用Windows Explorer，运行相同程序的两个拷贝，并尝试编辑在该程序两个实例中的相同记录；但是将会失败，这是因为记录被另一个用户锁定了。

更新数据

用户使用ClientDataSet组件（或在BDE中求助于缓存更新）的一个原因，是为了能够使一个SQL连接进行更新。考虑下面的SQL等值连接：

```
SELECT * FROM Orders, Customer
WHERE Customer.CustNo=Orders.CustNo
```

该语句会提供一个订单列表并设置订单的客户信息。BDE将任何SQL连接都看做是只读的，这是因为插入、更新与删除连接中的行都是不明确的。例如，将一行插入到前面的连接中，是应该导致一种新产品与一个新的供应商，还是只导致一种新产品？ClientDataSet/Provider结构允许我们指定一个主更新表（这种高级特性本书中实际上并未讨论），同时定制更新的SQL，我们在第14章看到过其中的一部分，并且第16章“多层DataSnap应用程序”会做进一步的讨论。

ADO支持一种等效于缓存更新的技术，称做批更新，这种技术与BDE的方法非常相似。在下一节中，我们将进一步地讨论ADO的批更新，它们提供的功能，以及批处理如此重要的原因。然而，在本节中，将不需要它们来解决更新连接的问题，原因很简单，在ADO中，

连接“天生”就是可以更新的。

例如, JoinData范例是建立在一个使用前面提到的SQL连接的ADODataset组件上的。如果运行它,可以编辑其中一个字段并保存变化(通过移开记录)。这不会出现错误,因为更新已经被成功地应用。ADO,相对于BDE而言,采取了一种更实用的方法来解决该问题。在一个ADO连接中,每个字段对象都知道它属于哪一个数据表格。如果更新Products表格中的一个字段并发送所做的改变,那么就会生成一条SQL UPDATE语句,以更新在Products表格中的字段。如果你改变Products表格中的一个字段与Suppliers表格中的一个字段,那么会生成两条SQL UPDATE语句,每个表格一条。

在连接中插入一行也遵循相同的过程。如果插入一行并且只为Products表格输入值,那么会为Products表格生成一条SQL INSERT语句。如果为两个表格输入值,那么会生成两条SQL INSERT语句,每个表格一条。语句执行的顺序是很重要的,因为新产品可能会与新的供应商有关,所以要首先插入新的供应商。

ADO解决方案最大的问题就是,在删除一个连接中的一行时,删除操作将会失败。大家所看到的精确的消息依据正使用的ADO版本与数据库而定,但它将显示在不能删除的行上,因为其他记录与其相关。错误消息可能会令人混淆是非。在当前的场景中,错误消息会说明一种产品不能删除,因为有其他产品与该记录相关。但是无论一种产品是否有相关记录,都会出现错误。按照插入的逻辑来考虑删除过程可能会找到解释。为此,生成两条SQL DELETE语句:一条针对Suppliers表格,而另一条针对Products表格,与显示的消息相反,针对Product表格的DELETE语句会成功,失败的是Suppliers表格的DELETE语句,因为在Supplier仍具有相关记录时,不能删除它。

提示:如果对生成的SQL语句好奇,并且使用的是SQL Server,则可以使用SQL Server Profiler查看这些语句。

除了理解其工作原理之外,研究该问题更好的一种方法是通过用户的眼睛。从用户的角度看,当用户在网格中删除了一行时,他们是想只删除产品,还是同时删除产品与供应商呢?我敢打赌,99%的用户希望执行前一操作,而不是后一操作。幸运的是,可以使用另一个动态属性,得到正确的解决方法——在这里,是Unique Table动态属性。可以使用下面这行代码指定删除只针对Products表格,而不针对Suppliers表格:

```
ADOQuery1.Properties['Unique Table'].Value := 'Products';
```

因为在设计时不能分配这个值,所以最好的方法是将这行代码放在窗体的OnCreate事件中。

批更新

当使用批更新时,对记录做的任何修改都可以在内存中进行;然后,整“批”的修改将作为一个操作被提交。这种方法提供了一些性能上的优点,但需要该技术的更实际的原因是:用户不能在其进行更新时连接到数据库上。“briefcase”应用程序中就有这种情况(稍后在“Briefcase模型”小节,我们将对其进行介绍),但这可能也是使用另一种ADO技术,Remote Data Services (RDS)的Web应用程序中的情况。

在数据集合打开之前,通过将LockType设置为ltBatchOptimistic,就可以在任何ADO数据集合中使用批更新。另外,还需要将CursorLocation设置为clUseClient,这是因为批更新是

由ADO的光标引擎来管理的。从此以后，修改都被做成一个“增量”（也就是一列变化），数据集会查看所有意图与目的，好像数据已经被修改，但其实修改只是在内存中进行；它们并没有实际地应用于数据库上。为了永久使用所做的修改，使用UpdateBatch（等效于缓存更新的ApplyUpdates）即可：

```
ADODataset1.UpdateBatch;
```

为了放弃整批的更新，可以使用CancelBatch或CancelUpdates。在ADO的批更新、BDE的缓存更新以及ClientDataSet之间，其方法与属性名称中有很多相似的地方。例如，UpdateStatus可以完全像在缓存更新中一样地用于识别记录（根据它们是否已被插入、更新、删除或未修改）。这种方法特别适用于在一个网格中用不同颜色突出记录，或在状态栏中显示它们的状态。这在语法上有一些细小的区别，如将RevertRecord修改为“CancelBatch(arCurrent);”，其他的则需要更多的努力。

有一个有用的缓存更新特性没有出现在ADO批更新中，这就是数据集合的UpdatesPending属性。如果已经修改但还没有应用，该属性为True。在窗体的OnCloseQuery事件中，它是特别有用的：

```
procedure TForm1.FormCloseQuery(
  Sender: TObject; var CanClose: Boolean);
begin
  CanClose := True;
  if ADODataset1.UpdatesPending then
    CanClose := (MessageDlg('Updates are still pending' #13 +
      'Close anyway?', mtConfirmation, [mbYes, mbNo], 0) = mrYes);
end;
```

然而，利用一些知识与灵活性，就可以实现适当的ADOUpdatesPending函数。所需要知道的就是ADO数据库具有一个名为FilterGroup的属性，它是一种过滤器。与数据集合的Filter属性（它通过将数据与一个给定条件的对比来过滤数据）不同，FilterGroup过滤器基于记录的状态。这样的一个状态是fgPendingRecords，它包含了已经修改但还没有应用的所有记录。所以，为了允许用户查看目前为止他们所做的全部修改，只需要执行下面两行代码即可：

```
ADODataset1.FilterGroup := fgPendingRecords;
ADODataset1.Filtered := True;
```

自然，结果集合现在还包含已被删除的记录。大家将看到的效果是字段只是保留空白，这也没有很好地解决问题，因为我们不知道哪一个记录已经被删除了（在ADOExpress的第一个版本中并没有这种行为，它将显示被删除记录的字段值）。

解决UpdatesPending问题所需要的“灵活性”就涉及前面讨论过的复制技术。ADOUpdatesPending函数将设置FilterGroup，把数据集合限制为只包含那些还没有应用的修改。程序员所需要做的就是一旦FilterGroup被应用，查看记录集合中是否有一些记录。如果有，那么就存在一些更新还没有提交。然而，如果对实际数据集合这样做的话，FilterGroup的设置将会移动记录指针，并且用户界面将被更新。最好的方法是使用一个复制品：

```
function ADOUpdatesPending(ADODataset: TCustomADODataset): boolean;
var
```



```
Clone: TADODataset;  
begin  
  Clone := TADODataset.Create(nil);  
  try  
    Clone.Clone(ADODataset);  
    Clone.FilterGroup := fgPendingRecords;  
    Clone.Filtered := True;  
    Result := not (Clone.BOF and Clone.EOF);  
    Clone.Close;  
  finally  
    Clone.Free;  
  end;  
end;
```

在该函数中，要复制原来的数据集合，设置FilterGroup，并且检查数据集合是否既在文件头也在文件尾。如果是的话，就没有等待处理的记录了。

积极锁定

我们在前面研究了LockType属性，并且介绍了消极锁定工作的原理。在这一小节中，我们将研究积极锁定，不仅因为它是中或高吞吐量的事务处理的首选锁定类型，而且因为它也是批更新所使用的锁定方法。

积极锁定假设用户在同一时间更新相同记录的可能性很小，所以不太可能引起冲突。这样得出一种结论，所有用户都可以在任何时间编辑任何记录，而且当修改被保存时，可以处理不同用户对相同记录进行更新而产生冲突的结果。这样，冲突被看做是规则的一种异常。这意味着没有为防止两个用户同时编辑相同的记录而采取控制手段。第一个保存其修改的用户将会成功；第二个试图更新相同记录的用户则可能失败。这种行为对于公文包应用程序（本章稍后将会对其进行讨论）与Web应用程序来说是基本要求，因为在这样的程序中，没有与数据库的持久连接，所以无法实现消极锁定。与消极锁定成为对照的是，积极锁定具有很大的优点，资源只是立刻被消耗，所以平均资源使用量低得多，使数据库更具有可扩展性。

让我们来看一个范例。假设有一个ADODataset与dbdemos.mdb数据库的Customer表格相连，其LockType被设置为ltBatchOptimistic，而且其内容显示在一个网格中。假设还有一个称为UpdateBatch的按钮。运行程序两次（如果读者不打算重建，该程序就是BatchUpdates范例），并在程序的第一份拷贝中开始编辑一个记录。尽管出于简单考虑，我们将只使用一台机器来解释一个冲突，但遇到的情况与后续事件都与使用多台机器没有什么不同：

1. 选择在Canada中的Bottom-Dollar Markets公司，并将其名称改为Bottom-Franc Markets。
2. 保存这一改变，移开记录以提交它，并单击按钮进行批更新。
3. 在程序的第二个拷贝中，找到相同的记录，并将公司名称改为Bottom-Pound Markets。
4. 移开记录并单击按钮进行批更新，结果将会失败。

除了很多ADO错误消息外，大家另外所收到的精确消息不但与使用的ADO版本有关，而且与例子中的操作方式有关。在ADO 2.6中，错误消息是“Row cannot be located for

updating. Some values may have been changed since it was last read.”，这是积极锁定的本性。记录的更新通过执行下列SQL语句来执行：

```
UPDATE CUSTOMER SET CompanyName="Bottom Pound Markets"
WHERE CustomerID="BOTTM" AND CompanyName="Bottom-Dollar Markets"
```

被这个UPDATE语句影响的记录数希望是1，因为它使用主关键字与CompanyName字段的内容（与记录第一次读取时的相同）确定了原记录的位置。然而，在这个范例中，被UPDATE语句影响的记录数是0。只有当记录已被删除，或记录的主关键字发生了变化，或正修改的记录已经被别人修改过时才会出现这种情况。于是，更新失败。

如果“第二个用户”修改了ContactName字段，而不是CompanyName字段，那么UPDATE语句会如下所示：

```
UPDATE CUSTOMER SET ContactName="Liz Lincoln"
WHERE CustomerID="BOTTM" AND ContactName="Elizabeth Lincoln"
```

在该范例场景中，这个语句将会成功，这是因为其他用户没有修改主关键字或联系人名称。这种行为与修改即更新（update where changed）这一更新模式下的BDE行为相似。BDE的UpdateMode属性在ADO中被一个数据集合的Update Criteria动态属性所取代。下面的列表显示了可以赋给该动态属性的可能值：

| 常量 | 通过……查找记录 |
|---------------------|--------------|
| adCriteriaKey | 只有主关键字列 |
| adCriteriaAllCols | 所有各列 |
| adCriteriaUpdCols | 只有主关键字列和修改的列 |
| adCriteriaTimeStamp | 只有主关键字列和时间戳列 |

不要掉入这样一个陷阱，这就是认为其中某一个设置在整个应用程序中会比另一个更好。事实上，对设置所做的选择将会受到每个数据表格内容的影响。也就是说，Customer数据表格只有CustomerID、Name与City字段，在这种情况下，对其中任何字段的更新在逻辑上都不会与其他任何字段的更新相互排斥，所以对于该数据表格，一个好的选择应当是adCriteriaUpdCols（也就是默认设置）。然而，如果Customer表格包含一个PostalCode字段，那么PostalCode字段的更新将会与另一个用户的City字段更新相互排斥（因为如果城市发生变化，那么邮政编码也一定要变化，反之亦然）。在这种情况下，可以认为adCriteriaAllCols是一种更安全的解决方案。

另一个要注意的问题是，在多个记录更新期间，ADO是如何处理错误的。使用BDE的缓存更新和ClientDataSet，可以使用OnUpdateError事件在错误发生时处理每个错误，并在移到下一个记录之前解决问题。在ADO中，不能建立这样一种对话。可以使用数据集合的OnWillChangeRecord与OnRecordChangeComplete监控批更新的过程以及成功或失败，但不能像BDE与ClientDataSet那样在该过程期间修改记录并重新提交它。另外，如果在更新过程期间出现一个错误，则更新不会停止。它会继续到最后，直到所有更新都被应用或失败为止。这个过程可以产生一个更无助而且显然不正确的错误消息。如果多个记录都不能被更新，或失败的单个记录并不是最后一个被应用的记录，那么在ADO 2.6中，错误消息将是“Multiple-step OLE DB operation generated errors. Check each OLE DB status value, if available. No

work was done.”。最后一句就有问题：它说明“没有工作被完成”，但这是不正确的。实际上对于失败的记录没有完成工作这句话是正确的，但是对于其他记录，都已成功应用并更新。

解决更新冲突

作为应用更新的一个自然结果，需要对批更新采取的方法是更新一批，让个别记录失败，然后一旦整个过程完毕，再处理失败的记录。可以通过将数据集合的FilterGroup设置为fgConflictingRecords，确定失败的记录：

```
ADODataset1.FilterGroup := fgConflictingRecords;
ADODataset1.Filtered := True;
```

对于每个失败的记录，你可以使用下列TField属性，通知用户三种关于每个字段的关键信息：

| 属性 | 描述 |
|----------|-------------|
| NewValue | 经用户修改后得到的值 |
| CurValue | 从数据库中取到的新值 |
| OldValue | 首次读数据库时得到的值 |

ClientDataSet组件的用户都将会意识到ReconcileErrorForm对话框的方便性，它封装了为用户显示新老记录的过程，并允许他们指定要采取的行动。遗憾的是，在ADO中没有对于该窗体的等效物，而且TReconcileErrorForm已经用ClientDataSet编写了，所以很难转化它并用于ADO数据集合。

关于使用这些TField属性，我将要指出的最后一个问题是：它们都直接取自它们引用的底层ADO Field对象。这意味着，就像ADO中常见的那样，可以选择OLE DB供应器来支持自己所希望使用的特性。这对于大多数供应器都没问题，但Jet OLE DB供应器返回的值与CurValue相同，就像它处理OldValue一样。换句话说，如果使用Jet，就不能确定其他用户对字段的修改，除非借助于自己的手段。然而，通过使用SQL服务器的OLE DB供应器，只能在调用数据集合的Resync方法之后（将AffectRecords参数设置为adAffectGroup并将ResyncValues设置为adResyncUnderlyingValues），才能访问CurValue。代码如下所示：

```
adoCustomers.FilterGroup := fgConflictingRecords;
adoCustomers.Filtered := true;
adoCustomers.Recordset.Resync(adAffectGroup, adResyncUnderlyingValues);
```

断开的记录集合

批更新的所有这些知识允许人们利用下一个ADO特性：断开的记录集合。一个断开的记录集合就是已经从其连接中断开的一个记录集合。该特性的优点是，用户不能识别一个普通记录集合与一个断开的记录集合之间的区别；它们的特性设置与行为几乎完全一样。为了从其连接中断开一个记录集合，必须将CursorType设置为clUseClient，将LockType设置为ltBatchOptimistic。然后，告诉数据集合，它不再拥有连接：

```
ADODataset1.Connection := nil;
```

从此以后，记录集合将继续包含相同的数据，支持相同的定位特性，并允许添加、编辑与删除记录。惟一相关的区别是，不能进行批更新，这是因为需要连接到服务器上更新服务器。可以重新进行连接（并使用UpdateBatch），如下所示：

```
ADODataset1.Connection := ADOConnection1;
```

这个特性通过切换到ClientDataSets，也可以用于BDE与其他数据库技术，但ADO解决方案的优美之处是，可以使用dbGo组件建立完整的应用程序，而且不会察觉断开的记录集合。当发现这个特性并想利用它时，可以继续使用已经在使用的相同组件。

有两个原因使我们可能想断开你的记录集合：

- 为了保持较低的连接总量
- 为了建立一个公文包应用程序

我在本节中将讨论降低连接量的问题，稍后将返回到公文包应用程序。

大多数正规的客户机/服务器商务应用程序都要打开数据表格，并在数据表格打开期间，保持与其数据库的一个永久性连接。然而，与数据库连接的原因通常只有两个：提取数据与更新数据。如果改变正规的客户机/服务器应用程序，在数据表格打开并提取数据之后，数据集合将会断开连接，连接数量将会降低，但用户不会意识到这一点，而且应用程序也不需要维护一个打开的数据库连接。下面的代码显示了这两个步骤：

```
ADODataset1.Connection := nil;
ADOConnection1.Connected := False;

惟一需要连接的地方在需要应用批更新之处。更新代码如下所示：

ADOConnection1.Connected := True;
ADODataset1.Connection := ADOConnection1;
try
  ADODataset1.UpdateBatch;
finally
  ADODataset1.Connection := nil;
  ADOConnection1.Connected := False;
end;
```

如果在应用程序中贯穿始终地遵循这种方法，则打开连接的平均数量在任何时间都将是最少的——连接将只为那些需要少量时间的用户打开。该变化的结果是更具可扩展性；与每个用户维护一个打开的连接相比，应用程序将能够处理更多并发的用户。当然，不利的一面是，重新打开连接可能在一些（但不是全部）数据库引擎上是一个长时间的过程，所以应用程序在批更新上的速度会减慢。

连接池

所有这些关于断开连接并重新打开它们的讨论将我们带到了连接池的主题上来。连接池——不要与会话池混淆——允许重新使用对一个数据库的连接，只要它们完成了连接。如果OLE DB供应器支持它而且它也被激活，这会自动发生，程序员不需要任何动作就可以利用连接池。

将连接汇集起来只有一个原因：性能。数据库连接的问题是，建立一个连接需要时间。在一个桌面数据库，如Access中，需要的时间通常较少。然而，在网络上使用的一种客户端/服务器数据库，如Oracle，可能需要用秒来计时。如果有这样一个昂贵（在性能方面）的资源，它的重复使用是很有意义的。

使用ADO的连接池，ADOConnection对象在被应用程序“解除”时，会被放置在一个池中。以后准备建立一个ADO的连接时，会自动在连接池中查找具有相同连接字符串的连接。如果发现匹配的连接，就会重新使用它；否则，一个新的连接将会建立。连接本身仍保留在池中，直到它们被重复使用、应用程序关闭或它们超时为止。默认情况下，连接会在60秒后超时，但从MDAC 2.5开始，开发人员可以使用HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>\SPTIMEOUT注册表键值设置这个超时周期。连接池会无缝处理发生的情况，与开发人员无关。该过程与BDE的数据库连接池（在Microsoft Transaction Server (MTS) 与COM+下）相似，主要区别是，ADO执行它自己的连接池，而不会求助于MTS或COM+。

默认情况下，连接池可以在所有的MDAC OLE DB供应器上用于关系型数据库（包括SQL Server与Oracle），Jet OLE DB供应器例外。如果使用ODBC，就应该在ODBC的连接池与ADO的连接池之间做出选择，但是这两者不能同时使用。从MDAC 2.1开始，可以使用ADO的连接池，ODBC的连接池不再使用。

说明：连接池没有出现在与OLE DB供应器无关的Windows 95上。

为了真正地适应连接池，读者需要查看连接被汇集与超时的情况。遗憾的是，在编写本书时还没有出现适当的ADO连接池查看工具；但是大家可以使用SQL Server的Performance Monitor，因为它可以准确地查看SQL Server数据库连接。

可以在Registry或连接字符串中使用或禁用连接池。Registry中的键值是OLEDB_SERVICES，位于HKEY_CLASSES_ROOT\CLSID\<ProviderCLSID>中。这是一个1比特的数组，允许开发人员关闭多个OLE DB服务，包括连接池、事务获取和光标引擎。为了使用连接字符串来禁用连接池，要在连接字符串的末尾包含“;OLE DB Services=-2”。为Jet OLE DB供应器打开连接池，则要在连接字符串的末尾包含“;OLE DB Services=-1”，它可以打开所有OLE DB服务。

持久的记录集合

对于公文包模式（将在下一小节中讨论）来说，持久的记录集合是一个非常有用的特性。该特性允许人们将任何记录集合的内容保存到一个本地文件中，之后再被装载。除了面向公文包模式外，该特性还允许开发人员建立真正的单层应用程序——这意味着我们在不必配置数据库的情况下可以配置一个数据库应用程序。这样在客户机上只需要占用很少的空间。

可以使用SaveToFile方法“存留”数据集合：

```
ADODataset1.SaveToFile('Local.ADTG');
```

该方法将把数据及其增量保存到磁盘上的一个文件中。可以使用LoadFromFile方法重新装载该文件，该方法接受一个参数说明要装载的文件。该文件的格式是Advanced Data Table Gram (ADTG)，它是Microsoft的专用格式。然而，它具有非常高效的优点。如果愿意，可以通过向SaveToFile传递第二个参数将文件保存为XML：

```
ADODataset1.SaveToFile('Local.XML', pfXML);
```

但是, ADO没有它自己的一个内置的XML分析器(像ClientDataSet那样), 所以它必须使用MSXML分析器。用户必须安装Internet Explorer 5或更新的版本, 或从Microsoft的网站上下载MSXML分析器。

如果想将文件本地保存为XML格式, 要注意一些缺陷:

- XML文件的保存与装载比ADTG文件的保存与装载慢。
- ADO的XML文件(与通常的XML文件)要比ADTG文件大(XML文件通常要比ADTG文件大两倍)。
- ADO的XML格式专门用于Microsoft, 与大多数公司的XML实现一样。这意味着ADO中生成的XML不能被ClientDataSet阅读, 反之亦然。幸运的是, 这个问题可以使用Delphi的XMLTransform组件来解决, 该组件可以用于不同XML结构之间的转换。

如果准备将这些特性只用于单层的应用程序, 而不作为公文包模式的一部分, 那么可以使用一个ADODataset组件, 将它的CommandType设置为cmdFile, 将它的CommandText设置为文件名称, 这样做比较省事, 可以不必再手工调用LoadFromFile了。然而, 仍必须调用SaveToFile。在公文包应用程序中, 该方法受到太多的限制, 因为数据集合可以用于两种不同的模式。

公文包模式

通过上面介绍的批更新、断开的记录集合与持久的记录集合等知识, 大家可以利用“公文包模式”。公文包模式背后的思想是, 用户可以在外出的路上使用应用程序——他们想得到在办公室桌面上使用的相同应用程序, 并在他们的笔记本电脑的客户端使用它。传统情况下, 这种场景的问题是, 当用户在客户端时, 他们没有与数据库服务器相连, 因为数据库服务器运行在他们办公室的网络上。因此, 在他们的笔记本电脑上没有数据, 而且数据无法更新。

这样就引入了新介绍的知识。假设应用程序已经编写完毕, 用户请求这种新的公文包增强模式, 而且开发人员已在已有的应用程序中将其翻新。需要通过为数据库中的每个数据表格执行SaveToFile, 为用户添加一个新的选项, 允许他们“准备”公文包应用程序。结果是一个ADTG或XML文件的集合, 其中镜像了数据库的内容。这些文件然后被复制到笔记本电脑上, 这里已经提前安装了一份应用程序。

应用程序需要感应到它是本地运行还是与网络相连。这可以通过尝试连接到数据库并查看它是否失败, 通过检查是否存在一个本地的“公文包”文件, 或通过建立自己设计的一些标记来确定。如果应用程序决定运行在公文包模式中, 那么就需要为每个表格使用LoadFormFile, 而不是为ADOConnections将Connected设置为True, 为ADO数据集合将Active设置为True。然后, 在保存数据时, 公文包应用程序需要使用SaveToFile, 而不是UpdateBatch。当用户返回到办公室后, 他们需要有一个更新过程, 从其本地文件中下载每个数据表格, 将数据集合连接到数据库上, 并使用UpdateBatch应用更新。

提示: 要看到一个完整的公文包模式的实现, 请参考前面提到的BatchUpdates范例。

关于ADO.NET

ADO.NET属于Microsoft的新的.NET结构的一部分——它们重新设计了应用程序的开发工具，使之更适合Web开发。ADO.NET是一种意义重大的ADO进化。它研究了Web开发的问题并提出了ADO问题的解决方案。ADO解决方案的问题是，它是基于COM的。对于单层与二层应用程序来说，COM的问题不大，但在Web开发的世界中，它作为传输机制是不可接受的。COM在Web开发中主要存在三个问题，这限制了它在Web开发中的使用：它（主要）只运行在Windows上，来自一个过程的记录集合传输需要COM调度，COM调用不能穿透公司的防火墙。ADO.NET解决所有这些问题的方法是使用XML。

其他一些重新设计的问题集中在如何将ADO记录集合划分为独立的类。这样的结果类适合解决单个问题，而不能解决多个问题。例如，目前被称做DataSetReader的ADO.NET类非常相似于一个只读的、只向前的服务器端记录集合，而且最适合于非常快速地读取结果集合。一个DataTable很像一个断开连接的、客户端的记录集合。一个DataRelation与MSDatashape OLE DB供应器具有相似性。所以，可以看到，对ADO工作原理的了解将有助于理解ADO.NET的基本原理。

小结

本章介绍了ActiveX Data Objects (ADO) 与dbGO，以及用于访问ADO接口的Delphi组件集。讲述了如何利用Microsoft Data Access Components (MDAC) 与不同的服务器引擎，以及在使用ADO时将会遇到的一些好处与障碍。

第16章将介绍Delphi的DataSnap结构，这种结构用于在三层环境中开发定制的客户端与服务器应用程序。你可以通过使用ADO来做，但是因为这是一本介绍Delphi的书，因此我更愿意为读者介绍Delphi的解决方案。在我们深入探讨DataSnap之后，通过介绍定制的数据敏感控件和数据集合组件，我们将继续研究Delphi的数据库结构。

第16章 多层DataSnap应用程序

大公司往往会有比使用本地数据库和SQL服务器的应用程序更广泛的需要。过去几年来，Borland软件公司一直在满足大公司的这种需要，它甚至临时改变自己的名字为Inprise来强调这种新的企业目标。名字虽然最终被改回为Borland，但仍保留了对企业开发方面的侧重。

Delphi面向多种不同的技术：基于Windows NT和DCOM的三层结构、TCP/IP和套接字应用程序及大部分的基于SOAP和XML的Web服务。本章将重点介绍面向数据库的多层结构；而面向XML的解决方案将会在第22章和第23章中进行讨论，这两章将专门讲述XML、SOAP和Web服务。

在正式开始前，我将强调两个主要因素。首先，支持这种开发类型的工具只能在Delphi的企业版本中获得；其次，在Delphi 7中不必再为DataSnap应用程序付费了。因为已经购买了开发环境，就可以在任何所需要的服务器上开发应用程序，而不必再给Borland公司支付任何额外的费用。对于DataSnap的发布策略来说，这是一个意义重大的变化（在Delphi 7中最重要的变化），而在以前，则需要为每个服务器支付费用（最初时非常高，后来明显降低了）。这种新的应用许可证将会增加DataSnap对开发人员的吸引力，这也是为什么要对其做详细介绍的原因。

本章主要包括以下内容：

- 逻辑上的三层结构
- DataSnap技术的基础
- 连接协议和数据包
- Delphi的支持组件（客户端和服务端）
- 连接代理和其他扩展的Delphi特性

Delphi发展历史中的一、二、三层

最初，数据库PC应用程序只是针对客户端的解决方案：即把程序与数据库文件都放在同一台计算机上。后来，一些富有冒险精神的程序员把数据库文件移植到了一台网络文件服务器上。客户端仍控制着应用软件和全部数据库引擎，但是数据库文件则可以在同一时间由多个用户进行访问了。虽然我们仍可以与Delphi应用程序和Paradox文件（或Paradox本身）一起使用这种类型的配置，但是仅仅在几年前，这种方法应用得要更广泛些。

第二个大的转变是客户机/服务器的开发，这样的开发从Delphi的第一版就开始被接受了。在客户机/服务器环境中，客户端计算机需要从一台服务器中获取数据，因为服务器控制着数据库文件及访问它们的数据库引擎。这种结构虽然忽视了客户端的作用，但减少了客户端机器对处理能力的要求。按照程序员实现客户机/服务器的方式，服务器可以完成大部分（如果不是全部）的数据处理操作。在这种方式下，一台功能强大的服务器就可以为多台

功能较弱的客户端提供数据服务了。

显然,使用这种功能集中的数据库服务器还有很多其他原因,如出于数据安全与完整性、更简便的备份策略及数据约束的集中管理等等各方面的考虑。数据库服务器通常也被称做SQL服务器,因为SQL是用于查询数据的一种最常用的语言;但它也可能被称为RDBMS (Relation DataBase Management System, 关系数据库管理系统),这反映了服务器提供了诸如备份和复制等管理数据的工具。

当然,创建的一些应用程序可能不需要RDBMS的全部性能优点,因此一种简单的只是针对客户端的解决方案可能就足够了。但在另一方面,我们可能在一台独立的计算机上却需要RDBMS系统的一些稳定性。在这种情况下,你可以使用一台SQL服务器的一个本地版本,如InterBase。传统的客户机/服务器开发是由一种二层结构完成的。然而,如果RDBMS主要是为了完成数据存储而不是数据与数字处理,那么客户端可能就包含用户界面代码(通过使用定制的报表、数据条目窗体、查询屏幕等方式来格式化输入与输出数据)和管理数据代码(还被称做商务规则)。在这种情况下,通常最好是将程序的这两部分分开,并建立一个逻辑上的三层结构。逻辑——这个术语意味着仍是两台计算机(也就是说,两个物理层),只是现在已经将应用程序划分成了三个不同的部分。

Delphi 2中通过数据模块引入了对逻辑上的三层结构的支持。对于应用程序的数据访问组件(或其他真正的非可视组件)来说,数据模块是一个非可视存储器,但它通常包括了多个与数据库相关的事件。可以在多个不同的窗体之间共享一个数据模块,并为不同的用户界面提供相同的数据;可能还有一个或多个数据输入窗体、报表、主/从窗体与各种图表或动态输出窗体。

逻辑上的三层结构可以解决很多的问题,但它也有一些缺陷。首先,必须将有关数据库管理部分的程序复制到不同的客户端计算机上;这么做可能会影响性能,但更大的问题是它增加了维护代码的复杂性。第二,当多个客户端修改相同的数据时,没有一种简单的方法可以处理更新结果时的冲突。最后,对于逻辑上的三层Delphi应用程序来说,开发人员必须在每台客户端计算机上安装和配置数据库引擎(如果有的话)和SQL服务器客户库。

另一个从客户机/服务器结构得到的逻辑上的提高是,将应用程序的数据模块移植到一个独立的服务器上,并使所有的客户端程序与之交互作用。这其实就是Delphi 3引入远程数据模块的目的。远程数据模块是在一台服务器上运行的,通常被称做应用程序服务器。应用程序服务器会依次与DBMS(它可以在应用程序服务器上运行,也可以在另一台专用计算机上运行)通信。所以,客户端计算机不会与SQL服务器直接相连,但可以通过应用程序服务器间接相连。

在这一点上有一个基本问题:仍然需要安装数据库访问软件吗?传统的Delphi客户机/服务器结构(即使使用逻辑上的三层结构)需要人们在每个客户端上都安装数据库访问软件,这样在配置与维护数百台机器时会非常麻烦。而在物理上的三层结构中,只需要在应用程序服务器上(不用在客户端上)安装与配置数据库访问软件。因为客户端程序只有用户界面代码,而且安装非常简单,所以它们现在被归类为所谓的瘦客户系统,用销售的说法,甚至可以称之为零配置瘦客户结构。但是我们应该把精力集中在技术的问题上,而不是销售的术语上。

DataSnap的技术基础

Borland公司在Delphi中引入这种物理多层结构时,称其为MIDAS(中间层分布式应用程序服务系统)。例如,Delphi 5介绍了该技术的第三个版本MIDAS 3。随后Borland公司将该技术更名为DataSnap,同时还扩展了其性能。

DataSnap要求在服务器上(实际上是中间层计算机)安装特定的库,它可以把从SQL服务器数据库或其他数据源中提取的数据提供给客户端。DataSnap并不需要一台SQL服务器用于数据存储。DataSnap可以从很多种类的数据源中提取出数据,这些数据源包括SQL、其他的DataSnap服务器或正在运算中的数据。

正如人们所期待的,DataSnap的客户端是非常简单的,而且很容易配置。所需要的惟一一个文件就是Midas.dll,一个小DLL文件,它执行了ClientDataSet与RemoteServer组件,并提供与应用程序服务器间的连接。正如在第13章“Delphi的数据库体系结构”中讨论的那样,分配DLL的另一种可替换的方法就是,将MidasLib单元包含在uses声明语句中,以便在可执行文件中嵌入这个库的代码。

IAppServer接口

DataSnap应用程序的两端通过使用IAppServer接口进行通信;程序清单16.1是对这种接口的定义。基本上不需要直接调用IAppServer接口的方法,因为Delphi中包括在服务器端应用程序上执行这个接口的组件,并且这些组件调用客户端应用程序上的这个接口。这些组件简化了对IAppServer接口的支持,有时甚至可以全部隐藏。实际上,服务器可能与其他的定制接口一起,使执行这个接口的对象在客户端有效。

程序清单16.1 IAppServer接口的定义

```
type
  IAppServer = interface(IDispatch)
    ['{1AEFCC20-7A24-11D2-98B0-C69BEB4B5B6D}']
    function AS_ApplyUpdates(const ProviderName: WideString; Delta: OleVariant;
      MaxErrors: Integer; out ErrorCount: Integer;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_GetRecords(const ProviderName: WideString; Count: Integer;
      out RecsOut: Integer; Options: Integer; const CommandText: WideString;
      var Params: OleVariant; var OwnerData: OleVariant): OleVariant; safecall;
    function AS_DataRequest(const ProviderName: WideString;
      Data: OleVariant): OleVariant; safecall;
    function AS_GetProviderNames: OleVariant; safecall;
    function AS_GetParams(const ProviderName: WideString;
      var OwnerData: OleVariant): OleVariant; safecall;
    function AS_RowRequest(const ProviderName: WideString; Row: OleVariant;
      RequestType: Integer; var OwnerData: OleVariant): OleVariant; safecall;
    procedure AS_Execute(const ProviderName: WideString;
      const CommandText: WideString; var Params: OleVariant;
      var OwnerData: OleVariant); safecall;
  end;
```

说明: DataSnap服务器使用COM类型库标明接口, 该技术已在第12章“从COM到COM+”中讨论过。

连接协议

DataSnap只定义了高层结构, 并使用不同的技术将数据从中间层移植到客户端。

DataSnap支持很多不同的协议, 包括:

分布式COM (DCOM) 和无状态COM (MTS或COM+) DCOM可以直接在Windows NT/2000/XP和Windows 98/Me中使用, 并且在服务器上不需要额外的运行时应用程序。DCOM基本上可以看做是COM技术的扩展, 它允许一个客户端应用程序使用现有的服务器对象, 并在一台独立的计算机上执行。DCOM的基础结构允许开发人员使用在COM+和较旧的MTS (Microsoft Transaction Server) 结构中可以获得的无状态COM对象。COM+和MTS中都可以提供包括安全、组件管理及数据库处理等特性, 同时用于Windows NT/2000和Windows 98/Me。

由于DCOM配置复杂而且在通过防火墙时易出现问题, 所以就连微软也要放弃DCOM, 转而支持基于SOAP的解决方案。

TCP/IP套接字 大部分系统都可以使用TCP/IP套接字。通过使用TCP/IP, 可以在不能使用DCOM的Web上分布客户端, 这样就会减少配置上的麻烦。为了使用套接字, 中间层计算机必须运行Borland提供的ScktSrvr.exe应用程序, 该程序既可以当做应用程序运行, 也可以当做服务运行。这种应用程序接收客户端请求并将这些请求转发给使用COM的远程数据模块 (在同一台服务器上执行)。套接字不能为客户端故障提供保护, 因为当客户端意外关闭时, 服务器并不会得到通知, 因此也不能将资源释放出来。

HTTP和SOAP HTTP作为Internet上的传输协议, 简化了与防火墙或代理服务器 (通常与定制的TCP/IP套接字不同) 的连接。开发人员需要一个特定的Web服务器应用程序, httpsrvr.dll, 它可以接收客户端的请求并使用COM创建正确的远程数据模块。这些Web连接也可以使用SSL安全性。最后, 基于HTTP传输的Web连接可以使用DataSnap对象组合支持。

说明: DataSnap HTTP的传输可以用XML作为数据包格式, 激活所有能读取XML的平台或工具, 以使其参与到DataSnap结构中。这是对原始DataSnap数据包格式的扩展, 这种格式也是与平台无关的。在HTTP上使用XML也是SOAP的基础内容。第23章“Web服务与SOAP”将对DataSnap中的SOAP做更详细的叙述。

在Delphi 6之前, 也可以用CORBA (Common Object Request Broker Architecture) 作为一种DataSnap应用程序的传输机制。由于要与Borland公司的VisiBroker CORBA解决方案的更新版本保持兼容, 这种特性在Delphi 7中已经废止了。

最后, 注意作为这种结构的一种扩展, 可以将数据包转换到XML中并将它们发送到Web浏览器。在这种情况下, 主要用额外的一个层次: Web服务器从中间层获得数据, 然后将它发送给客户端。我们将在第22章“使用XML技术”中讨论这种被称做Internet Express的结构。

提供数据包

整个Delphi的多层数据访问结构以数据包的思想为中心。在这里，一个数据包就是从应用程序服务器移动到客户端或从客户端移动到服务器的一个数据块。从技术上讲，数据包就是数据集合的一种子集。它描述了自己所包含的数据（通常是一些数据记录），并列出了数据字段的名称与类型。甚至更重要的是，一个数据包包括了约束——也就是应用于数据集合的规则。通常可以在应用程序服务器中设置这些约束，服务器将会把它们与数据一起发送到客户端应用程序中。

客户端与服务器之间发生的所有通信都是通过交换数据包来实现的。为了尽快地响应用户，服务器端的供应器组件在一个大数据集中管理多个数据包的传送。当客户端在一个ClientDataSet组件中接收到一个数据包时，用户可以编辑它所包含的记录。我们在前面曾提到过，在该进程期间，客户端也会接收与检查约束，并将其在编辑操作期间应用。

当客户端更新记录并发回一个数据包时，该数据包被称做delta。delta包会追踪原记录与新记录之间的区别，记录客户端需要服务器所做的所有改变。当客户端要求应用向服务器更新的数据时，它会向服务器发送delta，而且服务器将试着应用每一个改变。这里之所以说“试”，是因为如果服务器连着多个客户端，数据可能已经改变过了，并且更新请求可能失败。

因为delta包包括了原始数据，所以服务器可以迅速确定另一台客户机是否已经改过它了。如果是这样的话，服务器会触发OnReconcileError事件，这对于瘦客户应用程序来说是极其重要的组成元素。换句话说，这种三层结构使用了一种更新机制，它与Delphi用于高速缓存更新的机制相似。正如大家在第14章“使用dbExpress的客户机/服务器编程”中所看到的那样，ClientDataSet在一种数据高速缓存中管理数据，通常只是读取在服务器端可用的记录的一个子集，只装载更多的所需元素。当客户端更新记录或插入新记录时，它会将这些未决的改变存储在客户端的另一个本地高速缓存（delta缓存）中。

客户端还可以向磁盘保存数据包并脱机工作，非常感谢在第13章讨论的MyBase支持。甚至错误信息与其数据也可以使用数据包协议移动，所以它是这种结构的一个基础组成。

说明：要记住的重要一点就是，数据包是与协议无关的。数据包只是一个字节序列，所以可以在任何地方移动一系列字节。一个数据包，使结构适用于多传输协议（包括DODM、CORBA、HTTP与TCP/IP套接字）以及多个平台。

Delphi支持的组件（客户端）

我们已经介绍了Delphi的三层结构的基础知识，现在让我们开始讨论支持该结构的Delphi组件。为了开发客户应用程序，Delphi提供了ClientDataSet组件，该组件提供所有标准的数据集合功能并嵌入IAppServer接口的客户端。在这种情况下，数据通过远程连接发送。

可以通过客户端应用程序也需要的另一个组件实现与服务器应用程序的连接。应该使用下面三种专用连接组件之一（可以在DataSnap页面上获得）：

- DCOMConnection组件，在客户端使用，用于连接到一台DCOM与MTS服务器上，它或者位于当前计算机上，或者在ComputerName属性指明的另一台计算机上。该连接会有一个注册的对象，它具有给定的ServerGUID或ServerName属性。

- **SocketConnection**组件，通过一个TCP/IP套接字与服务器相连。开发人员应该指出IP地址或主机名，以及服务器对象的GUID（在**InterceptGUID**属性中）。该连接组件有一个额外的属性，**SupportCallbacks**，如果不使用回调函数，并想在Windows 95客户计算机上（没有安装Winsock 2）配置程序，则可以禁用该属性。

说明：在WebServices页也能发现**SoapConnection**组件，它要求一个特定类型的服务器，有关内容将在第23章中进行讨论。

- **WebConnection**组件，用于处理一个HTTP连接，该连接可以轻松通过防火墙。应该向URL说明httpsrvr.dll拷贝的位置，以及服务器上的远程对象的名称或GUID。

在Delphi 6中添加了一些客户端组件，主要用于管理连接：

- **CnnectionBroker**组件，被用做一个实际连接组件的别名，当拥有含多个客户端数据集合的单个应用程序时，这是非常有用的。事实上，为了改变每个数据集合的物理连接，只需要改变**ConnectionBroker**的**Connection**属性。也可以使用这种虚拟连接组件的事件代替那些实际连接，因此如果改变数据传输技术，则并不需要改变任何代码。同样原因，可以引用**CnnectionBroker**的**AppServer**代替一个物理连接的对应属性。
- **SharedConnection**组件，用于连接到一个远程应用程序的二级（或子）数据模块，在现有的到主数据模块的物理连接上传送数据。换句话说，一个应用程序能通过一个单独的共享连接与服务器上的多个数据模块相连。
- **LocalConnection**组件，可将一个本地数据集合供应器作为数据包的源来使用。通过将**ClientDataSet**直接连接到供应器也可以获得相同的效果。然而，使用**Local-Connection**，可以通过“假”连接的**IAppServer**界面，用相同的代码编写一个本地应用程序作为完整的多层应用程序。相对于使用直接连接的程序而言，这将使程序更容易扩展。

几个其他的DataSnap页面组件与将DataSnap数据包向定制XML格式的转变有关。这些组件（**XMLTransform**、**XMLTransformProvider**和**XMLTransformClient**）将在第22章中进行讨论。

Delphi支持的组件（服务器端）

在服务器端（或中间层），需要创建一个应用程序或嵌入到远程数据模块中的一个库，这是**TDataModule**类的特殊版本。另一个可替换的方法是针对事务型COM使用特殊的远程数据模块。在New Items对话框的Multitier页（从File►New►Others菜单中获得）中，有专门的向导用于建立远程数据模块的这些类型。

在服务器端惟一需要的专用组件就是**DataSetProvider**。每个数据表格或在客户端应用程序使用的服务器查询都需要一个这样的组件。它将为每个导出的数据集合使用一个独立的**ClientDataSet**组件。**DataSetProvider**已在第13章进行过介绍。

建立一个范例应用程序

现在，已经准备好了建立一个范例程序。这可以运用我们在前面介绍的一些组件，或者侧重于一些其他问题以及Delphi三层结构的其他方面。我们将分两步建立三层应用程序的

客户端和应用程序服务器。第一步将简单地使用最少的要素测试有关技术，这些程序将是非常简单的。

然后，将向客户端与应用程序服务器添加更多的功能。在每一个范例中，都要使用dbExpress来显示来自本地InterBase表格中的数据，而且完成所有工作，使大家可以在一台独立的计算机上测试程序。这里将不介绍在多台计算机上使用不同技术安装范例所必须遵循的步骤，因为这至少需要一本书的篇幅才能解释清楚。

第一个应用程序服务器

该基本范例的服务器端很容易建立。只需创建一个新的应用程序，并在对象存储库的Multitier页面中使用相应的图标添加一个远程数据模块即可。Remote Dataset Wizard（远程数据集合向导，如图16.1所示）将要求输入一个类的名称与实例样式。当输入一个类的名称，如AppServerOne，并单击OK按钮之后，Delphi将向程序添加一个数据模块。该数据模块通常拥有一般的属性与事件，但它的类却会有如下所示的Delphi语言声明：

```
type
  TAppServerOne = class(TRemoteDataModule, IAppServerOne)
  private
    { Private declarations }
  protected
    class procedure UpdateRegistry(Register: Boolean;
      const ClassID, ProgID: string); override;
  public
    { Public declarations }
  end;
```

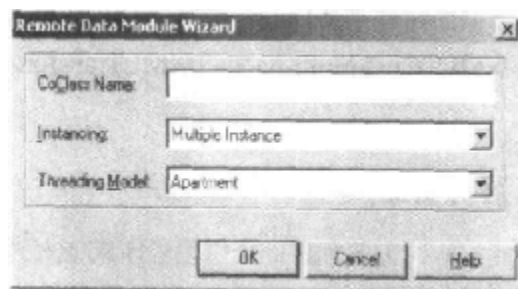


图16.1 远程数据模块向导

除了继承TRemoteDataModule基类之外，该类还实现了定制的IAppServerOne接口，该接口派生自一个标准的DataSnap接口（IAppServer）。该类还覆盖了UpdateRegistry方法，以添加对激活套接字与Web传输的支持，这一点读者可以在向导生成的代码中看到。在该单元的最后，大家将会发现类的出厂声明，如果阅读过第12章，则应该清楚这些：

```
initialization
  TComponentFactory.Create(ComServer, TAppServerOne,
    Class_AppServerOne, ciMultiInstance, tmApartment);
end.
```

现在，可以向数据模块添加一个数据集合组件（我使用的是dbExpress `SQLDataSet`），并将它连接到一个数据库和一个数据表或请求上，启动它，最后添加一个`DataSetProvider`，并将其链接到数据集合组件。这会获得一个与下面所示相似的DFM文件：

```
object AppServerOne: TAppServerOne
  object SQLConnection1: TSQLConnection
    ConnectionName = 'IBLocal'
    LoginPrompt = False
  end
  object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
  end
  object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
    Constraints = True
  end
end
```

该程序的主窗体几乎没有用处，所以可以简单地向它添加一个标签，说明它是服务器应用程序的窗体。当建立服务器之后，应该编译它并运行一次。该操作将会在系统中自动将该服务器注册为Automation服务器，使客户应用程序可以使用它。当然，应该要在要运行服务器的计算机（客户端或中间层）上注册它。

第一个瘦客户端

既然我们已经有了一个正在工作的服务器，下面就可以建立一个与服务器相连接的客户端了。将再次从一个标准的Delphi应用程序开始并向它添加一个`DCOMConnection`组件（或者是想要测试的特殊类型的连接而使用的相应组件）。该组件定义了一个`ComouterName`属性，可以用于指定负责应用程序服务器的计算机。如果想在同一台计算机上测试客户端和应用程序服务器，可以使该属性空着。

一旦选择了一台应用程序服务器计算机，就可以简单地显示`ServerName`属性的组合框列表来查看可用的DataSnap服务器。这个组合框会显示服务器的注册名称（默认情况下，服务器的可执行文件名后面跟着远程数据模块类的名称，如`AppServ1.AppServerOne`）。另外，还可以键入服务器对象的GUID作为`ServerGUID`属性。当设置`ServerName`属性时，Delphi通过在注册表中查找来确定其GUID，并自动输入该属性。

这时，如果将`DCOMConnection`组件的`Connected`属性设置为`True`，将会出现服务器窗体，说明客户端已经启动了服务器。然而通常不需要执行这个操作，因为`ClientDataSet`组件通常会激活`RemoteServer`组件。我已经说过，这样做只是为了强调后台发生的事情。

提示：为了能在Delphi甚至在DataSnap服务器没有被注册的计算机上打开该项目，应在设计时将`DCOMConnection`组件的`Connected`属性设置为`False`。

正如大家所预料的那样，下一步是向窗体添加一个`ClientDataSet`组件。我们必须通过`RemoteServer`属性将`ClientDataSet`与`DCOMConnection1`组件相连接，并借此与它导出的一个

供应器相连。可以通过普通组合框在ProviderName属性中看到可用的供应器列表。在这个范例如中，我们将只能选择DataSetProvider1，因为这是在所选择的服务器中唯一可以使用的供应器。该操作会将客户端内存中的数据集合与服务器上的dbExpress数据集合连接起来。如果激活客户端的数据集合并添加一些数据敏感控件（或一个DBGrid），将会立刻看到在这些数据集合中显示的服务器数据，如图16.2所示。



图16.2 当激活设计时与远程数据模块相连的ClientDataSet组件时，来自服务器的数据将像通常一样可见

这里是用于最小的客户端应用程序的DFM文件，ThinCli1:

```
object Form1: TForm1
  Caption = 'ThinClient1'
  object DBGrid1: TDBGrid
    Align = alClient
    DataSource = DataSource1
  end
  object DCOMConnection1: TDCOMConnection
    ServerGUID = '{09E11D63-4A55-11D3-B9C1-00000100A27B}'
    ServerName = 'AppServ1.AppServerOne'
  end
  object ClientDataSet1: TClientDataSet
    Aggregates = <>
    Params = <>
    ProviderName = 'DataSetProvider1'
    RemoteServer = DCOMConnection1
  end
  object DataSource1: TDataSource
    DataSet = ClientDataSet1
  end
end
```


显然，第一个三层应用程序非常简单，但它们解释了如何创建一个数据集合查看器（分开两个执行文件之间的操作）。在这里，客户端只是一个查看器。如果编辑客户端上的数据，将不会更新服务器上的文件。为了实现服务器文件的更新，需要向客户端应用程序添加更多的代码。然而，在此之前，让我们先向服务器添加一些特性。

向服务器添加约束

在Delphi中编写一个传统的数据模块时，可以很容易地添加一些应用程序逻辑或商务规则，具体方法是编写数据集合事件的处理程序，或设置字段对象的属性并处理它们的事件。应该避免在客户端应用程序上做这样的事情，而是在中间件上编写商务规则。

在DataSnap结构中，可以从服务器向客户端发送一些约束并且在用户输入时，让客户端程序利用那些约束。也可以发送字段属性（如最大、最小值和显示、编辑屏蔽）到客户端，并通过用于访问数据的数据集合（或一个UpdateSql对象）来（使用一些数据访问技术）处理更新。

字段与数据集合约束

当供应器接口创建数据包并向客户端发送时，它包括了字段的定义、数据表格与字段约束，以及一个或多个记录（由ClientDataSet组件请求）。这意味着我们可以通过使用基于SQL的约束来定制中间件，并建立分布式应用程序逻辑。

使用SQL表达式所创建的约束可以分配给一个完整的数据集合或指定的字段。供应器会将这些约束与数据一起发送到客户端，客户端在向服务器端发送更新之前会应用它们。这种方式可以减少网络流量，与客户端向应用程序服务器以及最终的SQL服务器发回更新数据相比，只需查找无效数据即可。在服务器端编写约束的另一个优点是，如果商务规则发生了改变，只需要更新服务器应用程序就行了，而不需要改动在多台计算机上的客户程序。

但是如何编写约束呢？可以使用下面几个属性：

- BDE数据集合有一个Constraints属性，它是TCheckConstraint对象的一个集合。每个对象都有一些属性，包括表达式和错误信息。
- 每个字段对象都定义有CustomConstraint和ConstraintErrorMessage属性。也有一个ImportedConstraint属性用于从SQL服务器引入约束。
- 每个字段对象也有一个DefaultExpression属性，该属性既可以本地使用，也可以传到ClientDataSet。这不是一个实际的约束，只是对终端用户的一个建议。

下一个例子，AppServ2，将为连接到EMPLOYEE InterBase数据库范例上的远程数据模块添加几个约束。在将表格连接到数据库并为其创建字段对象之后，可以设置下列特定的属性：

```
object SQLDataSet1: TSQLDataSet
...
object SQLDataSet1EMP_NO: TSmallintField
    CustomConstraint = 'x > 0 and x < 10000'
    ConstraintErrorMessage =
```

```

        'Employee number must be a positive integer below 10000'
        FieldName = 'EMP_NO'
    end
    object SQLDataSet1FIRST_NAME: TStringField
        CustomConstraint = 'x <> '#39#39
        ConstraintErrorMessage = 'The first name is required'
        FieldName = 'FIRST_NAME'
        Size = 15
    end
    object SQLDataSet1LAST_NAME: TStringField
        CustomConstraint = 'not x is null'
        ConstraintErrorMessage = 'The last name is required'
        FieldName = 'LAST_NAME'
    end
end

```

说明：表达式'x <> '#39#39是字符串x <> "的DFM转换形式，表示不需要一个空的字符串，最终的约束，not x is null，允许空的字符串，但不是空的值。

包含字段属性

通过使用DataSetProvider组件Options属性的poIncFieldProps值，可以控制是否向ClientDataSet发送中间层上的字段对象的属性（并复制给客户端相应的字段对象）。该标记控制了字段属性Alignment、DisplayLabel、DisplayWidth、Visible、DisplayFormat、EditFormat、MaxValue、MinValue、Currency、EditMask和DisplayValues（如果可以在字段中使用）的下载。下面这个例子是拥有定制属性的AppServ2范例中的另一个字段：

```

    object SQLDataSet1SALARY: TBCDField
        DefaultExpression = '10000'
        FieldName = 'SALARY'
        DisplayFormat = '#,###'
        EditFormat = '####'
        Precision = 15
        Size = 2
    end

```

使用该设置，可以像通常设置一个标准客户机/服务器应用程序的字段那样编写中间件。该方法还可以加快从客户机/服务器向多层结构移植现有应用程序的速度。向客户端发送字段的主要缺点是，发送所有的额外信息会花费时间。关闭poIncFieldProps可以极大地改善多列数据集的网络性能。

一台服务器通常还可以过滤返回客户端的字段——通过使用字段编辑器声明持久性字段对象并省略一些字段来实现的。因为滤出的一个字段可能需要在将来更新时用于识别记录（如果是这个字段是主键的一部分），所以还可以在服务器上使用字段的ProviderFlags属性向客户端发送字段值，但它不能用于ClientDataSet组件（与向客户端发送字段并隐藏它相比，这样会提供一些额外的安全性）。

字段与表格事件

可以像往常那样编写中间件数据集合与字段事件处理程序，并让数据集合以传统的方式处理客户端接收到的更新。这意味着，更新被认为是在数据集合上的操作，严格地讲是用户在本机直接编辑、插入或删除字段。

这可以通过设置TDataSetProvider组件的ResolveToDataSet属性来实现，还要连接用于输入的数据集合或用于更新的数据集合。该方法对支持编辑操作的数据集合也适用，包括BDE、ADO和InterBase Express数据集合，但是不包括那些新的dbExpress结构。

使用这种技术，更新可以通过数据集合来执行，这意味着拥有很多控制（标准事件被触发）但通常执行性能较慢。灵活性是更重要的，因为可以使用标准编码惯例。而且使用这种模型配置现有的本地或客户机/服务器数据库应用程序（使用数据集合与字段事件）也更简单。然而，要知道的是，只有当本地缓存（Delta）发回到中间件时，客户端程序的用户才会收到错误消息。向用户通知半小时前准备的一些数据是没有意义的，而且可能有些笨拙了。如果使用这种方法，可能需要在客户端的每个AfterPost事件上应用缓存中的更新。

最后，如果决定选择新数据集合而不是供应器提供更新，Delphi会在处理可能的异常方面提供很多帮助。任何由中间件更新事件（例如，OnBeforePost）引起的异常都会被Delphi自动转换成更新错误，激活客户端的OnReconcileError事件（本章稍后将介绍该事件）。没有异常显示在中间件上，但错误会返回客户端。

向客户端添加特性

在向服务器添加一些约束和字段属性之后，让我们再来看一下客户端应用程序。第一个版本是非常简单的，但是现在可以向它添加一些特性使之可以更好地工作。在ThinCli2范例会中，通过使用第13章讨论的客户数据集合技术，我已经嵌入了对检测记录状态和访问delta信息（更新被返回到服务器）的支持。该程序也负责处理协调错误和支持公文包模式。

注意，当使用该客户端进行本地数据编辑时，如果出现不符合应用程序商务规则的错误，服务器端会使用约束提醒用户。服务器还会为新记录的Salary字段提供一个默认值。在图16.3中，可以看到该客户应用程序显示的其中一条错误消息，该错误消息是从服务器端发出的。只有在本地编辑数据时才会显示该消息，向服务器发回数据时该消息并不会被显示。

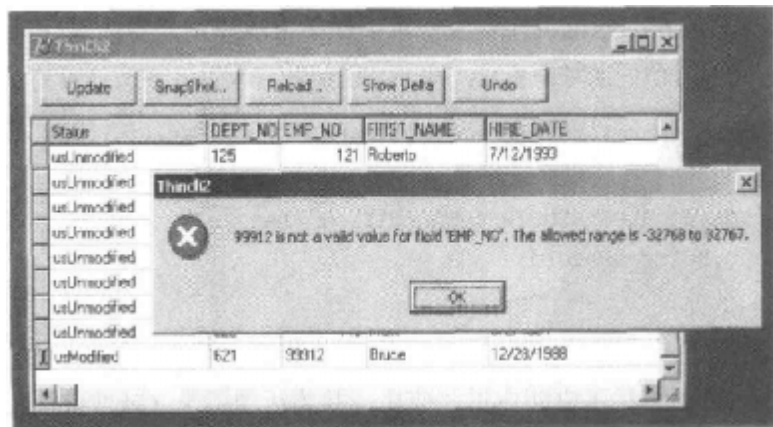


图16.3 当雇员ID号太大时，ThinCli2范例显示出的错误消息

更新操作序列

该客户端程序包含一个按钮，用于对服务器和一个标准的协调对话框进行更新。这里是与一个更新请求和可能的错误事件相关的完整的操作序列汇总：

1. 客户端程序调用ClientDataSet的ApplyUpdates方法。
2. Delta被发送给中间件上的供应器。该供应器会触发OnUpdateData事件，在它们到达数据库服务器之前，我们将有机会查看到请求的改变。在此，可以改动Delta，Delta用一种与ClientDataSet的数据相兼容的格式传递。
3. 供应器（从技术角度上来说，一部分供应器被称做“分辨器”）向数据库服务器应用Delta的每一行。在应用每个更新之前，供应器会接收到一个BeforeUpdateRecord事件。如果已经设置了ResolveToDataSet标记，该更新最终会在中间件上触发数据集的本地事件。
4. 在出现服务器错误的情况下，供应器会触发OnUpdateError事件（在中间件上），而且程序有机会在该级别上对该错误进行修正。
5. 如果中间件程序没有修正错误，相应的更新请求会保存在Delta中。在此，错误会返回客户端，或者在收集到一个给定数量的错误之后，根据ApplyUpdates调用的MaxErrors参数值返回错误。
6. 最后，含有更新的Delta包会被发送回客户端，并为每个保存的更新触发ClientDataSet的OnReconcileError事件。在该事件处理程序中，客户端程序会试图修正问题（可能是为用户提示帮助信息），在Delta中改动更新，并在稍后重新发送它。

刷新数据

可以获得数据的一个更新版本，其他用户也可以通过调用ClientDataSet的Refresh方法改动它。然而，该操作只有在缓存中没有未决的更新操作时才可以执行，如果当更改记录为非空的时候，调用Refresh会引起异常：

```
if cds.ChangeCount = 0 then  
    cds.Refresh;
```

如果只是某些记录发生了变化，则可以通过调用RefreshRecords来刷新其他的记录。这种方法只是刷新当前记录，但是它只应当在用户没有修改当前记录的情况下才使用。事实上，在这种情况下，RefreshRecords将在更改记录中保留未被应用的更改。作为一个范例，大家可以在每次记录成为活动记录时刷新它，除非它已经被修改并且更改还没有被发送到服务器：

```
procedure TForm1.cdsAfterScroll(DataSet: TDataSet);  
begin  
    if cds.UpdateStatus = usUnModified then  
        cds.RefreshRecord;  
end;
```

当数据被许多用户频繁改变并且每个用户都希望立即看到这种改变时，我们应该在AfterPost和AfterDelete方法上立即应用该变化，并为活动记录（就像前面提到的那样）或在

网格内可以看到的每个记录调用**RefreshRecords**。该代码实际上是**ClientRefresh**示例的一部分，连接到**AppServ2**服务器上。为了调试，程序也为其刷新的每条记录记录下了**EMP_NO**字段，如图16.4所示。

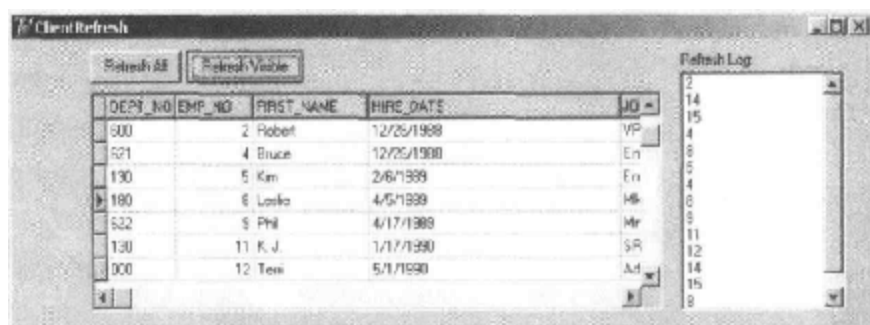


图16.4 ClientRefresh范例的窗体，可自动刷新活动记录，单击按钮还可进行扩展性更新

我通过向**ClientRefresh**示例添加一个按钮完成了这个目的。该按钮的处理器从当前记录移动到网格的第一个可视记录，然后移动到最后一个可视记录。假设第一行是有字段名的固定的行，则可以通过计算**RowCount - 1**，看看有多少行是可视的。该程序并不是每次都调用**RefreshRecord**，因为每次移动都会用前面所显示的代码激活**AfterScroll**事件。下面是更新可视行的代码，它可以通过计时器被触发：

```
// protected access hack
type
  TMyGrid = class (TDBGrid)
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  i: Integer;
  bm: TBookmarkStr;
begin
  // refresh visible rows
  cds.DisableControls;
  // start with the current row
  i := TMyGrid(DbGrid1).Row;
  bm := cds.Bookmark;
  try
    // get back to the first visible record
    while i > 1 do
    begin
      cds.Prior;
      Dec (i);
    end;
    // return to the current record
    i := TMyGrid(DbGrid1).Row;
    cds.Bookmark := bm;
  end;
```

```
// go ahead until the grid is complete
while i < TMyGrid(DbGrid1).RowCount do
begin
    cds.Next;
    Inc (i);
end;
finally
    // set back everything and refresh
    cds.Bookmark := bm;
    cds.EnableControls;
end;
end;
```

这种方法会产生大量的网络流量，因此只有当有实际的变化时，才需要触发更新。可以通过向服务器添加一个回滚技术来实现这个过程，以通知所有连接的用户，一个特定的记录已经发生改变。客户端能决定是否对改变感兴趣并最终触发更新请求。

高级的DataSnap特性

在DataSnap中还包括更多的高级特性，到现在为止我还没有介绍。下面就对该结构中的一些高级特性进行简要介绍，其中一部分特性由AppSPlus⁺与ThinPlus范例来解释。遗憾的是，逐一解释每一个单独的功能会把本章编成一本本书，所以我将只限于对此做一个概括性的介绍。

警告：ThinPlus范例需要运行Delphi的套接字服务器（在Delphi的bin文件夹中提供）。没有这个程序，当客户端试图连接到服务器上时，大家将会看到一个套接字错误。通过在客户端程序中修改服务器的IP地址，可以很容易地通过网络部署这个程序。

除了在随后各节讨论的特性之外，AppSPlus⁺与ThinPlus范例还解释了如何使用套接字连接、服务器端的事件与更新的有限记录，以及在客户端直接读取的记录。最后一个特性可以用下面这个调用来实现：

```
procedure TClientForm.ButtonFetchClick(Sender: TObject);
begin
    ButtonFetch.Caption := IntToStr (cds.GetNextPacket);
end;
```

这样将获得比客户端用户界面（DBGrid）所需请求更多的记录。换句话说，可以直接取得这些记录，而不需要等待用户在网格中操作。我建议大家阅读了本节剩下部分之后，认真研究一下这些复杂范例的细节信息。

参数查询

如果想在查询或存储过程中使用参数，则并不需要建立一个定制的方案（使用定制的方法调用服务器），而是让Delphi提供帮助。首先要在中间件上定义一个带有参数的查询，如：

```
select * from customer where Country = :Country
```

使用Params属性设置参数的类型与默认值。在客户端，可以在将它连接给相应的供应器之后，使用ClientDataSet快捷菜单中的Fetch Params命令。在运行时，可以调用等效的ClientDataSet组件中的FetchParams方法。

现在，通过使用Params属性向参数提供一个本地的默认值。当读取数据时，该参数就会被发送给中间件。ThinPlus范例使用下列代码刷新参数：

```
procedure TFormQuery.btnParamClick(Sender: TObject);
begin
  cdsQuery.Close;
  cdsQuery.Params[0].AsString := EditParam.Text;
  cdsQuery.Open;
end;
```

读者可以在图16.5中看到该范例的二级窗体，它在一个网格中显示了参数查询的结果。在该图中，还可以看到由服务器发送的一些定制数据，就像在“定制数据包”一节中解释的那样。

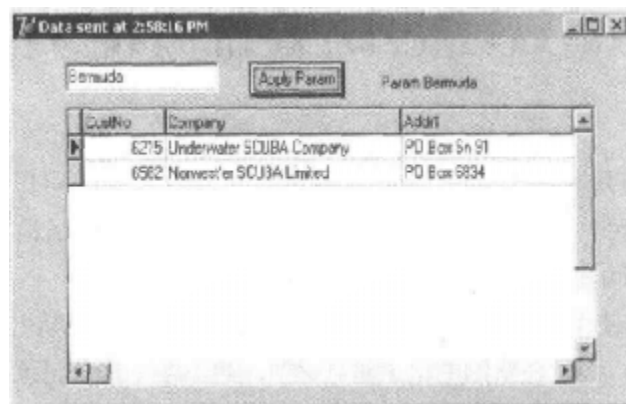


图16.5 ThinPlus范例的第二个窗体，显示了参数查询的数据

定制方法调用

因为服务器有一个标准的COM接口，故可以向它添加更多的方法或属性，并从客户端调用它们。只需打开服务器的类型库编辑器，就像在其他任何COM服务器上一样使用它们即可。在AppSPlus范例中，我已经使用下面的实现添加了一个定制的Login方法：

```
procedure TAppServerPlus.Login(const Name, Password: WideString);
begin
  // TODO: add actual login code...
  if Password <> Name then
    raise Exception.Create ('Wrong name/password combination received')
  else
    Query.Active := True;
    ServerForm.Add ('Login:' + Name + '/' + Password);
end;
```

该程序只执行了一个简单的测试，而并没有像一个正规的应用程序那样，参照一个授权列表进行名称/密码组合的测试。另外，关闭Query不能真正地奏效，因为它可以由供应器激活；实际上，关闭DataSetProvider是一种更为稳定的方法。客户端可以使用一种简单的方法访问服务器：利用远程连接组件的AppServer属性。下面是摘自ThinPlus范例的一个调用，它发生在连接组件的AfterConnect事件中：

```
procedure TClientForm.ConnectionAfterConnect(Sender: TObject);
begin
    Connection.AppServer.Login (Edit2.Text, Edit3.Text);
end;
```

注意，可以通过DCOM调用COM接口的额外方法，也可以使用一个基于套接字或HTTP的连接。因为该程序使用了safecall调用协定，故在服务器上引起的异常会自动向前传递并显示在客户端。这样，当用户选择Connect复选框时，用于客户端数据集合的事件处理程序将会被中断，使用了错误密码的用户将不能看到数据。

说明：除了从客户端到服务器直接进行方法调用外，还可以实现从服务器向客户端的回调。例如，使用这种方法通知特殊事件的每个客户端。使用COM事件是解决该问题的一种方法。另外，还可以添加一个由客户端实现的新接口，它会向服务器传递实现的对象。这样，服务器可以在客户端的计算机上调用方法。尽管如此，在使用HTTP连接时，回调是不可能的。

主/详关系

如果中间件应用程序调出了多个数据集合，开发人员就可以使用客户端上的多个ClientDataSet组件提取它们，并在本地连接它们以形成一个主/详结构。这种方法将会给详数据集合带来很多的问题，除非在本地提取所有记录。

该解决方案还使得应用更新变得非常复杂，在所有相关的详记录被删除之前，通常不能取消一个主记录，而且在新的主记录建立之前，也不能添加详记录（实际上，不同的服务器处理该方法的方法也不同，但是在大多数情况下，这是标准行为）。为了解决该问题，可以根据特定的规则，在客户端编写复杂代码来更新两个表格的记录。

另一种完全不同的方法就是提取一个单独的数据集合，它已经包括了详数据集合字段，这是TDataSetField类型的一个字段。为此，我们需要在服务器应用程序上建立主/详关系：

```
object TableCustomer: TTable
    DatabaseName = 'DBDEMOS'
    TableName = 'customer.db'
end
object TableOrders: TTable
    DatabaseName = 'DBDEMOS'
    MasterFields = 'CustNo'
    MasterSource = DataSourceCust
    TableName = 'ORDERS.DB'
end
object DataSourceCust: TDataSource
    DataSet = TableCustomer
end
```



```

object ProviderCustomer: TDataSetProvider
    DataSet = TableCustomer
end

```

在客户端，详数据表格将会显示为ClientDataSet的额外字段，而且DBGrid控件会将它显示为带有省略号按钮的额外的一列。单击该按钮会显示一个二级窗体，其中的网格显示详数据表格（如图16.6所示）。如果需要在客户端建立一个灵活的用户接口，则可以使用DataSetField属性添加一个二级ClientDataSet，连接到主数据集合的数据集合字段。亦即简单地为主ClientDataSet创建持续性的字段并连接属性：

```

object cdsDet: TClientDataSet
    DataSetField = cdsTableOrders
end

```

通过这样的设置，可以显示一个独立的DBGrid中的详数据集合，这个DBGrid像通常一样是放置在窗体中的（图16.6中的按钮网格）或任何其他需要的形式。要注意的是，采用这种结构，更新将只与主数据表格相关，并且即使在复杂的情况下，都由服务器负责处理正确的更新序列。

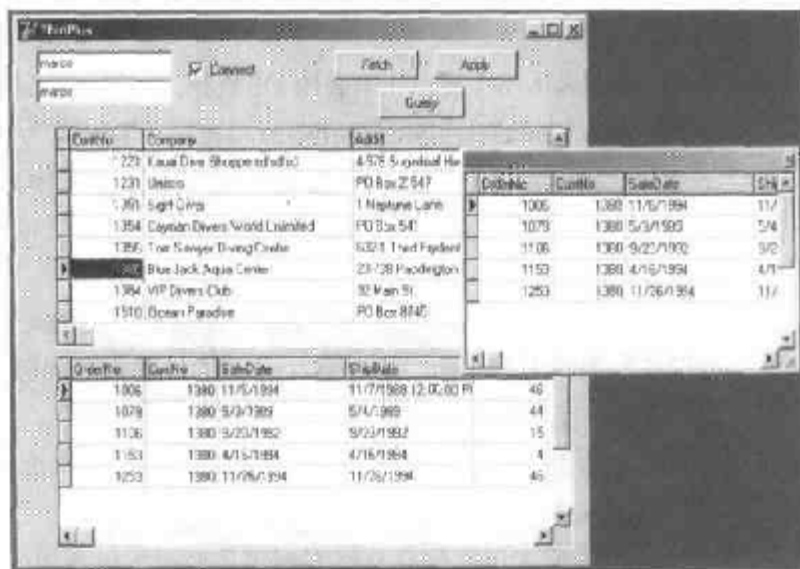


图16.6 ThinPlus范例给出了数据集合字段的两种显示方式：在浮动窗口中的网格内显示，以及由ClientDataSet抽取并在第二个窗体中显示。通常只需二选一。

使用连接代理

前面已经提到过，当我们想改变单个程序的许多ClientDataSet组件使用的物理连接时，ConnectionBroker组件是很有用的。事实上，通过连接每个ClientDataSet组件到ConnectionBroker，能简单地通过更新代理的物理连接来改变它们的物理连接。

ThinPlus范例使用下列这些设置：

```

object Connection: TSocketConnection
    ServerName = 'AppSPlus.AppServerPlus'
    AfterConnect = ConnectionAfterConnect
    Address = '127.0.0.1'
end

```

```

end
object ConnectionBroker1: TConnectionBroker
    Connection = Connection
end
object cds: TClientDataSet
    ConnectionBroker = ConnectionBroker1
end
// in the secondary form
object cdsQuery: TClientDataSet
    ConnectionBroker = ClientForm.ConnectionBroker1
end

```

这基本上就是所有需要我们做的了。为了改变物理连接，放置新DataSnap连接组件到主窗体并为其设置代理的Connection属性。

更多的供应器选项

我已经提到过DataSetProvider组件的Options属性，注意，可以使用它来向数据包添加字段属性。还有其他一些选项，可以用来定制数据包以及客户端程序的行为。下面是一个简单的列表：

- **poFetchBlobsOnDemand**选项用于最小化下载BLOB数据。在这种情况下，客户端应用程序可以通过指定ClientDataSet的FetchOnDemand属性为True或为指定的记录调用FetchBlobs方法来下载BLOB。同样，可以通过设置poFetchDetailsOnDemand选项，关闭详记录的自动下载。在客户端还可以使用FetchOnDemand属性或调用FetchDetails方法。
- 当正在使用一个主/详关系时，可以使用两个选项控制层次。**poCascadeDeletes**标记控制供应器是否应该在删除主记录前，删除详记录。如果数据库服务器执行级联删除操作作为其引用完整性支持的一部分，就可以设置该选项。同样，当一个主/详关系的关键值更新自动由服务器执行时，可以设置**poCascadeUpdates**选项。
- 可以限制在客户端上的操作。最具限制性的选项是**poReadOnly**，它会禁止任何更新操作。如果开发人员想要让用户拥有有限的编辑功能，可以使用**poDisableInserts**、**poDisableEdits**或**poDisableDeletes**。
- **poAutoRefresh**选项用于向客户端重新发送一份它改动过的记录；在其他用户同步进行非冲突操作的情况下，这样做是非常有用的。还可以通过指定**poPropagateChanges**选项向客户端发送回在BeforeUpdateRecord或AfterUpdateRecord事件处理程序中所做的改动。当使用自动递增字段、触发器以及其他技术在服务器或中间件上改动数据超出了客户级请求的变化范围时，该选项也是适用的。
- 最后，如果想让客户端驱动操作，则可以激活**poAllowCommandText**选项。这会让我们使用GetRecords或Execute方法从客户端设置中间件的SQL查询或数据表格名称。

SimpleObjectBroker组件

SimpleObjectBroker组件提供了一种容易的方法来确定一个服务器应用程序在多台服务器计算机之间的位置。只需提供可以使用的计算机的一个列表，客户端就会依次测试它们，直到发现可以使用的应用程序服务器为止。

而且，如果激活LoadBalanced属性，该组件将随机地选择一台服务器；当多个客户端使用相同配置时，将会在台服务器之间自动分布连接。这看起来像“穷人”的对象代理，实际上，一些高度扩展的负载平衡系统并没有提供比此更多的支持。

对象组合

当多个客户端同时与服务器连接时，有两个选择。第一，为每个客户端建立一个远程数据模块对象，让每个请求按顺序进行处理（使用ciMultiInstance样式的COM服务器的默认行为）。第二，让系统为每台客户端建立应用程序的不同实例（ciSingleInstance）。这种方法需要更多的资源与更多的SQL服务器连接（与许可）。

DataSnap中的支持还为对象组合提供了另外一种方法。请求该特性，我们所需要的就是在覆盖的UpdateRegistry方法中添加一个对RegisterPooled的调用。与在这种结构中建立的无状态支持相结合，组合特性还允许我们在大量客户端之间共享一些中间件对象。组合机制被内置到COM+中，但是DataSnap也可以使其供HTTP和基于套接字的连接使用。

客户端上的用户将花费大量时间来读取数据并输入更新，他们通常不能连续地请求数据与发送更新。当客户端不调用中间件对象的方法时，这个相同的远程数据模块就可以用于其他的客户端。事实上，因为无状态，即使服务器专供一台特定的客户端使用时，每个请求也会作为新的操作到达中间件。

定制数据包

有很多方法用于在IAppServer接口处理的数据包中包括定制信息。最简单的方法就是处理供应器自己的OnGetDataSetProperties事件。该事件有一个Sender参数，这是一个数据集集合参数，用于说明数据的来源；还有一个OleVariant数组Properties参数，在该参数中，可以放置额外的信息。需要为每个额外属性定义一个可变数组，并包括额外属性的名称、它的值，以及是否需要将数据与更新Delta一起返回到服务器（IncludeInDelta参数）。

当然，可以传递相关数据集集合组件的属性，或传递其他任何值（额外的伪属性）。在AppSPlus范例中，我向客户端传递了查询执行的时间及其参数：

```
procedure TAppServerPlus.ProviderQueryGetDataSetProperties(
  Sender: TObject; DataSet: TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['Time', Now, True]);
  Properties[1] := VarArrayOf(['Param', Query.Params[0].AsString, False]);
end;
```

在客户端，ClientDataSet组件有一个GetOptionalParameter方法，用于提取给定名称的额外属性值。ClientDataSet还有SetOptionalParameter方法，用于向数据集合添加更多的属性。这些值被保存在磁盘中（用公文包模型），并且最终会发回中间件（通过将可变数组的IncludeInDelta成员设置为True）。下面是在前面代码中提取数据集合的简单例子：

```
Caption := Data sent at ' + TimeToStr (TDateTime (
    cdsQuery.GetOptionalParam('Time')));
Label1.Caption := 'Param ' + cdsQuery.GetOptionalParam('Param');
```

该代码的效果可以参见图16.5。另一种也是更强大的定制发往客户端数据包的方法是，处理供应器的OnGetData事件，它在客户端数据集合的窗体中接收外出数据包。使用该客户端数据集合的方法，可以在将它发往客户端之前编辑数据。例如，对一些数据进行编码或滤出敏感的记录。

小结

Borland最初引入其多层技术是在Delphi 3中，并且不断地从一个版本到另一个版本进行扩展。除了进一步的更新以及将名字从MIDAS改变到DataSnap之外，Delphi 6还提供了对SOAP支持的介绍，引入了用于中间件应用程序的一种可替换和可扩展的结构。我们将在第23章中对这一主题进行详细的讨论。Delphi 7在另一方面则削减了对CORBA的支持。

然而，随着对一种新的许可证机制的介绍（基本上是免费地部署），Delphi 7已经为开发人员对这种技术增强的适应性铺好了道路。这一点在DataSnap的SOAP变体中尤其显得突出，但是套接字连接也提供了一种对于数据传输有效性和易于部署之间的良好平衡。

在此，我们将继续数据库编程，讨论数据库敏感控件并定制数据集合。在本书的最后部分，我们将讨论套接字、互联网编程以及SOAP，届时，读者会对基于Delphi的中间件结构有一个完整的认识，其中包括提供了DataSnap相似特性的第三方工具的可用性。

第17章 编写数据库组件

在第9章“编写Delphi组件”中，我们深入探讨了Delphi组件的开发。在讨论了数据库编程后，我们将回到前面的话题并集中讨论相关数据库组件的开发。

主要有两组这样的组件：数据敏感控件，用来为程序的用户显示一个字段的数据或整个记录；数据集合组件，用来为现有数据敏感控件提供数据，从数据库或其他任何数据源读取数据。本章将讨论这两个话题。

本章主要包括以下内容：

- 数据敏感组件：数据链接
- 面向字段的数据敏感控件
- 数据敏感的TrackBar和ProgressBar
- 面向记录的数据敏感控件
- 记录浏览器
- 建立定制的数据集合
- 保存一个数据集合到本地流

数据链接

编写Delphi数据库程序时，通常总是将一些数据敏感控件连接到一个DataSource组件上，然后再将该DataSource组件连接到一个数据集合中。数据敏感控件和DataSource间的连接就称为数据链接，并且由TDataLink类的一个对象展示。数据敏感控件创建并管理该对象，还描述它对数据的惟一连接。从更实际的角度来看，要做一个数据敏感组件，需要向该对象添加一个数据链接和这个内部对象的一些属性，如DataSource属性和DataField属性。

Delphi使用DataSource和DataLink对象进行双向通信。数据集合使用该连接来通知数据敏感控件，新的数据可以使用了（因为数据集合被激活或当前记录被改变等等）。而数据敏感控件使用该连接来查询字段的当前值或更新它，并通知这个事件的数据集合。

这些组件之间的关系是非常复杂的，因为有些连接可以是一对多的。例如，将多个数据源连接到同一个数据集合中，通常会有多个数据链接到同一个数据源（因为需要一个用于每个数据敏感组件的连接），在大多数情况下，要将多个数据敏感控件连接到每个数据源上。

TDataLink类

在本章中我们将会多次使用TDataLink及其派生类，它们都在DB单元中定义。该类有一组虚拟方法，这些方法与事件的角色相类似。它们是我们能在一个特定子类上覆盖的“almost-do-nothing（几乎什么也不做）”的方法，用于截取用户的操作和其他的数据源事件。下面是从类的源代码中提取的一个程序清单：

```

type
  TDataLink = class(TPersistent)
  protected
    procedure ActiveChanged; virtual;
    procedure CheckBrowseMode; virtual;
    procedure DataSetChanged; virtual;
    procedure DataSetScrolled(Distance: Integer); virtual;
    procedure FocusControl(Field: TFieldRef); virtual;
    procedure EditingChanged; virtual;
    procedure LayoutChanged; virtual;
    procedure RecordChanged(Field: TField); virtual;
    procedure UpdateData; virtual;

```

所有这些虚拟方法都被DataEvent私有方法所调用，这是一种用于数据源的窗口程序，由几种数据事件触发（见例举的TDataEvent）。这些事件最初是在数据集合、字段或数据源中的，通常被应用于一个数据集合。数据集合组件的DataEvent方法将这些事件分配到连接的数据源中。每个数据源调用NotifyDataLinks方法，将事件转发到每个连接的数据链接，然后由数据源触发它的OnDataChange或OnUpdateData事件。

派生的数据链接类

从技术角度上来说，TDataLink类不是抽象类，但是很少会直接使用它。当需要创建数据敏感控件时，就将使用一个派生类或派生出一个新的类。从TDataLink中派生的最重要的类就是TFieldDataLink类，该类被数据敏感控件使用，该控件相关于数据集合的单个字段。多数数据敏感控件符合该类，并且TFieldDataLink类会解决该类型组件的大部分常见问题。

所有面向表格或面向记录的数据敏感控件都可以定义特定的TDataLink子类，我们稍后将会介绍。TFieldDataLink类有一个事件列表，对应它覆盖的基类的虚拟方法。这使得定制类会更加简便，因为我们可以使用事件处理器，而不是必须从其上继承一个新的类。这里是覆盖方法的一个示例，如果可能，它会激活对应的事件：

```

procedure TFieldDataLink.ActiveChanged;
begin
  UpdateField;
  if Assigned(FOnActiveChange) then FOnActiveChange(Self);
end;

```

TFieldDataLink类也包含Field和FieldName属性，用于将数据敏感控件连接到数据集合的一个特定字段中。该链接使用Control属性来保持对当前可视组件的一个引用。

编写面向字段的数据敏感控件

既然理解了数据链接的工作原理，现在让我们开始建立一些数据敏感控件。首先建立的两个示例是ProgressBar和TrackBar数据敏感版本的普通控件。可以使用第一个控件以一种可视方式来显示一个数字值，如一个百分比；用第二个控件来允许用户改变该数字值。

说明：本章中所有组件的代码都在MdDataPack文件夹中，该文件夹中同时也包含有一个相似命名的软件包，用于安装所有的组件。其他的文件夹包含有使用这些组件的示例程序。

只读的ProgressBar

ProgressBar的一个数据敏感控件版本是相对简化了的数据敏感控件，因为它是只读控件。该组件由非数据敏感版本派生而来，并添加了一些它封装的数据链接对象的属性：

```
type
  TMdDbProgress = class(TProgressBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
  protected
    // data link event handler
    procedure DataChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;
```

由于每个数据敏感组件都连接到一个单个字段，故该控件可获得DataSource和DataField属性。不需要编写很多代码；下面的示例可从内部数据链接对象简单地导出属性：

```
function TMdDbProgress.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;

procedure TMdDbProgress.SetDataField (Value: string);
begin
  FDataLink.FieldName := Value;
end;

function TMdDbProgress.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;

procedure TMdDbProgress.SetDataSource (Value: TDataSource);
begin
```

```

    FDataLink.DataSource := Value;
end;

function TMDdbProgress.GetField: TField;
begin
    Result := FDataLink.Field;
end;

```

当然，要想使这些组件工作，必须在组件本身被创建或撤销的同时，创建或撤销数据链接：

```

constructor TMDdbProgress.Create (AOwner: TComponent);
begin
    inherited Create (AOwner);
    FDataLink := TFieldDataLink.Create;
    FDataLink.Control := Self;
    FDataLink.OnDataChange := DataChange;
end;

destructor TMDdbProgress.Destroy;
begin
    FDataLink.Free;
    FDataLink := nil;
    inherited Destroy;
end;

```

在前面的构造器中，注意组件安装它自己的一个方法作为事件处理器，用于数据链接。这里是组件最重要的代码存储的位置。每当数据改变时，就要修改进度条的输出，以反映当前字段的值：

```

procedure TMDdbProgress.DataChange (Sender: TObject);
begin
    if FDataLink.Field is TNumericField then
        Position := FDataLink.Field.AsInteger
    else
        Position := Min;
end;

```

遵循VCL数据敏感控件的约定，如果字段类型无效，组件将不显示错误消息——它只是禁止输出。而当SetDataField方法将它分配给控件时，可能需要检查字段的类型。

从图17.1中可以看到DbProgr应用程序的一个输出示例，该示例使用标签和进度条来展示一个命令的数量信息。感谢这种可视线索，使我们能够跨过记录并轻易识别出用于许多项的命令。该组件的一个明显的好处就是应用程序几乎不包含代码，因为所有重要的代码都在定义组件的MdProgr单元中。

正如大家所看到的那样，编写一个只读数据敏感组件并不太困难。但是，另一方面，在一个DBCtrlGrid容器中使用这样的组件将变得极其复杂。

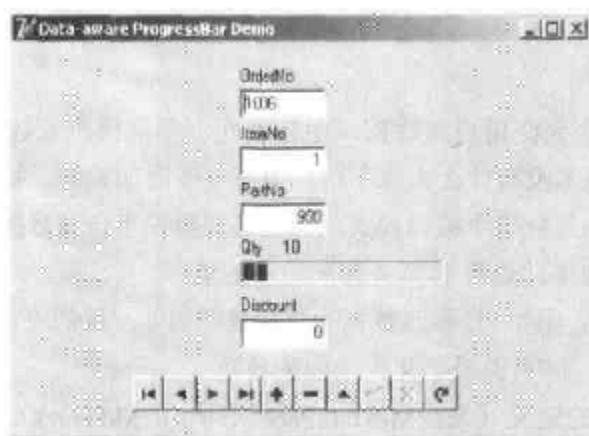


图17.1 数据敏感型控件ProgressBar在DbProgr范例中的样子

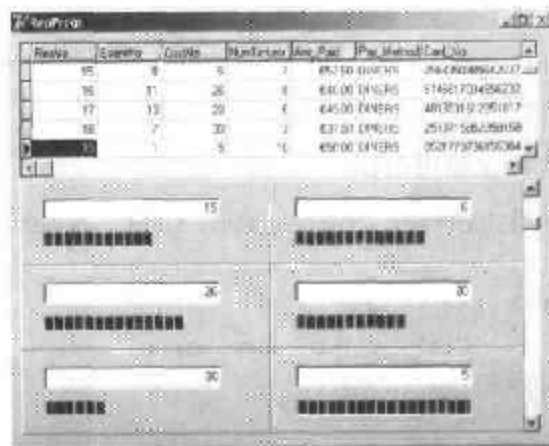
说明：如果读者记得第11章讨论的Notification方法，则可能想知道如果由数据敏感组件引用的数据源被破坏将发生什么。一个好消息就是数据源拥有解除器，它能从自己的数据链接上删除自己。因此不需要为数据敏感控件使用Notification方法，即使许多书籍和文章中建议这样做，并且VCL有许多关于它的无用代码。

可复制的数据敏感控件

扩展一个数据敏感控件来支持它在DBCtrGrid组件内的使用是相当复杂并且不容易讲解的。读者可以在MdDataPack包的MdRepPr单元中找到进度条的全部可复制版本，并在ReProgr文件夹中看到它的使用范例，同时还有描述它的开发过程的一个HTML文件。DBCtrGrid组件有一种特殊的行为：它通过使用smoke和mirrors，能够在屏幕上显示相同物理控件的多个版本。网格能够将控件附加到一个数据缓冲器而不是当前记录上，并且将控件描述操作重定向到监控器的另一部分。

简而言之，为了出现在DBCtrGrid上，一个组件的csReptlicable控件式样必须被设定；这个标志仅仅指示该组件实际支持被控件网格控制。首先，它必须对应cm_GetDataLink-Delphi消息并返回一个指针到数据链接，这样，控件网格就能够使用并改变它。其次，它需要定制Paint方法在相应的画布对象上拖动输出，当ControlState属性的csPaintCopy标记被设置时，该对象将提供在wm_Paint消息的参数上。

示例中所涉及的代码和DBCtrGrid组件并不常用，因此在这里不详细介绍它们，但可以在选配光碟上找到全部代码和更多关于源代码的信息。下图是使用该组件的一个测试程序的输出。



可读写的TrackBar

下一步是编写一个允许用户在数据库中修改而不是仅仅浏览数据的组件。该类型组件的整体结构与以前的版本没有什么太大不同，但有一些附加元素。特别是当用户开始与组件交互时，代码应把数据集合置于编辑模式，然后通知数据集合该数据已修改。而数据集合将使用一个FieldDataLink事件处理器来查询被更新的值。

为了说明怎样才能创建一个修改数据的数据敏感组件，我们决定扩展nxkBar控件。这可能不是最简单的示例，但它说明了几个重要的技巧。

这里是组件的类的定义（来自MdDataPack软件包的MdTrack单元）：

```

type
  TMDbTrack = class(TTrackBar)
  private
    FDataLink: TFieldDataLink;
    function GetDataField: string;
    procedure SetDataField (Value: string);
    function GetDataSource: TDataSource;
    procedure SetDataSource (Value: TDataSource);
    function GetField: TField;
    procedure CNHScroll(var Message: TWMHScroll); message CN_HSCROLL;
    procedure CNVScroll(var Message: TWMVScroll); message CN_VSCROLL;
    procedure CMExit(var Message: TCMExit); message CM_EXIT;
  protected
    // data link event handlers
    procedure DataChange (Sender: TObject);
    procedure UpdateData (Sender: TObject);
    procedure ActiveChange (Sender: TObject);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    property Field: TField read GetField;
  published
    property DataField: string read GetDataField write SetDataField;
    property DataSource: TDataSource read GetDataSource write SetDataSource;
  end;

```

与我们先前建立的只读数据敏感控件相比较，该类更复杂，因为它有三个消息处理器，包括组件通知处理器和两个用于数据链接的新的事件处理器。组件要在构造器上安装这些事件处理器，而构造器也可以禁用组件：

```

constructor TMDbTrack.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.Control := Self;

```

```

FDataLink.OnDataChange := DataChange;
FDataLink.OnUpdateData := UpdateData;
FDataLink.OnActiveChange := ActiveChange;
Enabled := False;
end;

```

get和set方法以及DataChange事件处理器与在TMdDbProgress组件中的相似。惟一的不同就是不管何时，只要数据源或数据字段发生了改变，组件就会检查当前的状态，以便确定是否需要激活其本身：

```

procedure TMdDbTrack.SetDataSource (Value: TDataSource);
begin
  FDataLink.DataSource := Value;
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;

```

这段代码要测试三种条件：数据链接应当是激活状态、该链接应当引用一个实际的字段并且该字段不能是只读的。

当用户修改字段时，组件应当认为该字段的名称是无效的；要想测试这一条件，可在组件中使用一个try/finally块：

```

procedure TMdDbTrack.SetDataField (Value: string);
begin
  try
    FDataLink.FieldName := Value;
  finally
    Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
      not FDataLink.Field.ReadOnly;
  end;
end;

```

当数据集合被激活或关闭时，控件会执行相同的测试：

```

procedure TMdDbTrack.ActiveChange (Sender: TObject);
begin
  Enabled := FDataLink.Active and (FDataLink.Field <> nil) and
    not FDataLink.Field.ReadOnly;
end;

```

这种组件的代码中最有意思的就是与其用户界面相关的部分。当一个用户开始移动滚动条翻页时，组件会将数据集合置于编辑模式，并让基类更新滚动条的位置，同时改变被修改数据的数据链接（以及数据源）。这里是相应的代码：

```

procedure TMdDbTrack.CNHSroll(var Message: TWMHScroll);
begin
  // enter edit mode
  FDataLink.Edit;
  // update data

```

```

    inherited;
    // let the system know
    FDataLink.Modified;
end;

procedure TMDdbTrack.CNVScroll(var Message: TWVScroll);
begin
    // enter edit mode
    FDataLink.Edit;
    // update data
    inherited;
    // let the system know
    FDataLink.Modified;
end;

```

当数据集需要新的数据，例如，要执行一个Post操作时，它将应通过TFieldDataLink类的OnUpdateData事件向组件发出请求：

```

procedure TMDdbTrack.UpdateData (Sender: TObject);
begin
    if FDataLink.Field is TNumericField then
        FDataLink.Field.AsInteger := Position;
end;

```

在合适的条件被满足的情况下，组件将会更新相应的表字段中的数据。最终，如果组件丢失了输入，它将强制进行数据更新（如果数据被改变），这样只要用户移动到不同的字段，任何显示该字段值的其他数据敏感组件都会显示正确的值。如果数据并未发生改变，组件将不会更新表中的数值。这里是VCL所用组件的标准CMExit代码，它也被这个组件所借用：

```

procedure TMDdbTrack.CMExit(var Message: TCMExit);
begin
    try
        FDataLink.UpdateRecord;
    except
        SetFocus;
        raise;
    end;
    inherited;
end;

```

可以获得一个演示程序来测试这个组件；读者可以在图17.2中看到其输出。DbTrack程序包含一个复选框，用于激活或禁用这个表、可视组件和一些按钮，这些按钮用于从其相关的字段中分离垂直的TrackBar组件。我把这些放在了窗体中，以测试激活或禁用跟踪工具栏。

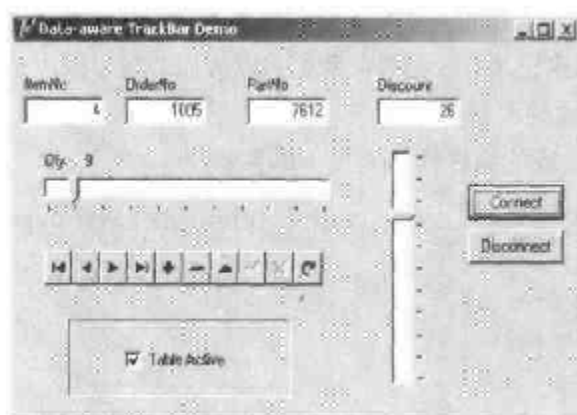


图17.2 用于向数据库表中输入数据的DbTrack范例的跟踪条。复选框和按钮用于测试组件的许可状态

创建定制的数据链接

到目前为止，我们所建立的数据敏感控件全部都引用了数据集合的特定字段，因此可使用一个TFieldDataLink对象来建立与数据源的连接。现在让我们来建立一个数据敏感组件，该组件将与数据集合作为一个整体来运行，是一个简单的记录浏览器。

Delphi的数据库网格可以同时显示几个字段的值和几个记录。我的记录浏览器组件通过使用一种定制的网格，能够列出当前记录的所有字段。该示例将显示怎样建立定制的网格控件以及伴随它的定制的数据链接。

记录浏览器组件

在Delphi中并没有数据敏感组件可以在不显示其他记录的情况下，处理单个记录中的多个字段。事实上，能够在相同表中显示多个字段的组件就两种：DBGrid和DbCtrlGrid，这两种组件可以显示多个字段和多条记录。

我们将在本小节描述的记录浏览器组件都是基于两栏网格的，第一栏显示表格的字段名称，而第二栏显示对应的字段值。网格中行的数目与字段的数目相对应，当它们超出可视区域时使用垂直滚动条。

为了建立该组件所需要的数据链接是一种类，只连接到记录浏览器组件并直接在其单元的实现部分中声明。这与VCL使用的用于某些特定数据链接的方法相同。下面是新类的定义：

```
type
  TMdRecordLink = class (TDataLink)
  private
    RView: TMdRecordView;
  public
    constructor Create (View: TMdRecordView);
    procedure ActiveChanged; override;
    procedure RecordChanged (Field: TField); override;
  end;
```

正如大家所看到的, 该类覆盖了与主要事件有关的方法——在这种情况下, 只是简单地改变活动状态与数据(或记录)。另外, 我们可以导出一些事件, 然后让组件去处理它们, 正如TFieldDataLink所做的那样。

但构造器要求相关联的组件作为它唯一的参数:

```
constructor TMdRecordLink.Create (View: TMdRecordView);
begin
    inherited Create;
    RView := View;
end;
```

在将一个引用保存到相关联的组件之后, 其他的方法就可以直接对其进行操作了:

```
procedure TMdRecordLink.ActiveChanged;
var
    I: Integer;
begin
    // set number of rows
    RView.RowCount := DataSet.FieldCount;
    // repaint all...
    RView.Invalidate;
end;

procedure TMdRecordLink.RecordChanged;
begin
    inherited;
    // repaint all...
    RView.Invalidate;
end;
```

该记录链接代码是非常简单的。创建这个例子大部分的困难在于对网格的使用。为了避免处理这些无用的属性, 我从TCustomGrid类导出了记录浏览器的网格。这种类为网格合并了很多的代码, 但是其大部分的属性、事件和方法都是被保护的。因此, 该类的声明相当长, 因为它需要发布很多现有的属性。这里仅是一个摘录(除了基类的属性):

```
type
    TMdRecordView = class (TCustomGrid)
    private
        // data-aware support
        FDataLink: TDataLink;
        function GetDataSource: TDataSource;
        procedure SetDataSource (Value: TDataSource);
    protected
        // redefined TCustomGrid methods
        procedure DrawCell (ACol, ARow: Longint; ARect: TRect;
            AState: TGridDrawState); override;
        procedure ColWidthsChanged; override;
        procedure RowHeightsChanged; override;
```

```

public
  constructor Create (AOwner: TComponent); override;
  destructor Destroy; override;
  procedure SetBounds (ALeft, ATop, AWidth, AHeight: Integer); override;
  // public parent properties (omitted...)

published
  // data-aware properties
  property DataSource: TDataSource read GetDataSource write SetDataSource;
  // published parent properties (omitted...)
end;

```

除了重新声明属性来公布它们外，该组件还定义了一个数据链接对象和DataSource属性。没有用于该组件的DataField属性，因为它引用了一个完整的记录。该组件的构造器很重要，它设置了许多未公布的属性值，其中包括网格选项：

```

constructor TMDRecordView.Create (AOwner: TComponent);
begin
  inherited Create (AOwner);
  FDataLink := TMDRecordLink.Create (self);
  // set numbers of cells and fixed cells
  RowCount := 2; // default
  ColCount := 2;
  FixedCols := 1;
  FixedRows := 0;
  Options := [goFixedVertLine, goFixedHorzLine,
    goVertLine, goHorzLine, goRowSizing];
  DefaultDrawing := False;
  ScrollBars := ssVertical;
  FSaveCellExtents := False;
end;

```

这个网格有两栏（其中一栏是固定的），并且没有固定的行。固定的栏用于调整网格每行的大小。遗憾的是用户不能拖动固定行来调整栏的大小，因为我们不能重调整固定元素，并且网格已有固定栏。

说明：另一种可选的方法是使用一个附加的空栏，如DBGrid控件那样。在添加固定行后，可以调整其他两个栏的大小。总而言之，我更喜欢自己的设置。

我使用另外一种方法调整栏的大小。第一栏（包含字段名）能通过编程代码或在设计时以可视方式来调整大小，而第二栏（包含字段值）能使用组件剩余部分来调整大小：

```

procedure TMDRecordView.SetBounds (ALeft, ATop, AWidth, AHeight: Integer);
begin
  inherited;
  ColWidths [1] := ClientWidth - ColWidths[0];
end;

```

当组件大小改变和任意一个栏改变时，将发生重新调整大小的情况。使用该代码，组件的DefaultColWidth属性变成第一栏的固定宽度。

所有内容都建立后，组件的主要方法被DrawCell方法覆盖，详见程序清单17.1。在这种方法下，控件将会显示关于字段及其值的信息。需要做三件事。如果数据链接没被连接到数据源，网格显示“空元素”符号（[]）。当画第一栏时，记录浏览器显示字段的DisplayName，这与DBGrid使用的用于标题的值相同。当画第二栏时，组件访问字段值的文本表示，用DisplayText属性提取（或使用备忘录字段的AsString属性）。

程序清单17.1 定制RecordView组件的DrawCell方法

```

procedure TMDRecordView.DrawCell(ACol: Longint; ARow: TRect;
  AState: TGridDrawState);
var
  Text: string;
  CurrField: TField;
  Bmp: TBitmap;
begin
  CurrField := nil;
  Text := '[]'; // default
  // paint background
  if (ACol = 0) then
    Canvas.Brush.Color := FixedColor
  else
    Canvas.Brush.Color := Color;
  Canvas.FillRect (ARect);
  // leave small border
  InflateRect (ARect, -2, -2);
  if (FDataLink.DataSource <> nil) and FDataLink.Active then
    begin
      CurrField := FDataLink.DataSet.Fields[ARow];
      if ACol = 0 then
        Text := CurrField.DisplayName
      else if CurrField is TMemoField then
        Text := TMemoField (CurrField).AsString
      else
        Text := CurrField.DisplayText;
    end;
  if (ACol = 1) and (CurrField is TGraphicField) then
    begin
      Bmp := TBitmap.Create;
      try
        Bmp.Assign (CurrField);
        Canvas.StretchDraw (ARect, Bmp);
      finally
        Bmp.Free;
    end;
  end;

```



```

end;
end
else if (ACol = 1) and (CurrField is TMemoField) then
begin
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_WordBreak or dt_NoPrefix)
end
else // draw single line vertically centered
    DrawText (Canvas.Handle, PChar (Text), Length (Text), ARect,
        dt_vcenter or dt_SingleLine or dt_NoPrefix);
if gdFocused in AState then
    Canvas.DrawFocusRect (ARect);
end;
end;

```

该方法的最后部分是组件认为的备注和图形字段。如果该字段是TMemoField, 那么DrawText函数调用不指定dt_SingleLine标记, 但当没有更多空间时, 则使用dt_WordBreak标记来约束文字。对于一个图形字段, 该组件将使用一种完全不同的方法, 将字段图像指定给临时位图, 然后拉伸它填充单元的表面。

要注意的是, 该组件将DefaultDrawing属性设置为False, 因此它也负责绘制背景和中间的矩形, 就像在DrawCell方法中所做的那样。该组件也调用InflateRect API函数, 在单元边框和输出文本间留下小的空间。而后, 通过调用另一个Windows API函数 (DrawText) 产生的输出, 使文本在单元格中垂直居中。

该绘图代码可在运行时和设计时工作, 如图17.3所示。其输出可能并不很完美, 但该组件在很多情况下是非常有用的。如果想为单个记录显示数据, 则代替用标签和数据敏感控件建立定制窗体, 可以很方便地使用该记录的浏览器网格。要记住的重要的一点就是, 记录浏览器是只读组件。可以扩展它来添加编辑能力 (它们是TCustomGrid类的一部分)。然而, 我决定通过添加对显示BLOB字段的支持来使组件更完整, 而不是添加这种支持。

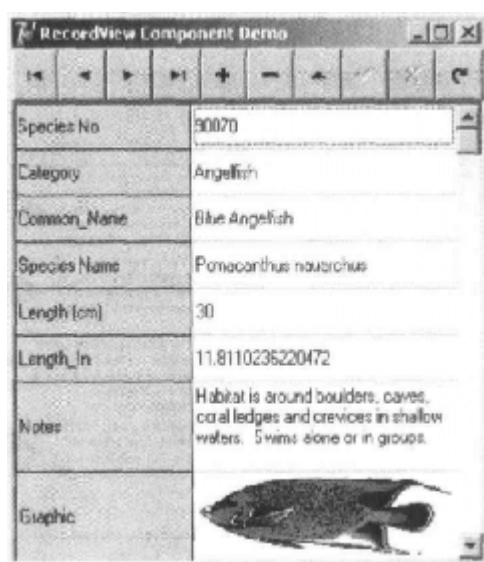


图17.3 ViewGrid范例演示了RecordView组件的输出, 这里使用了Borland的BioLife数据库表格样本

为了改善图形输出, 该控件使那些字段的行高两倍于那些普通文本字段的行高。当连接数据敏感控件的数据集合被激活时, 该操作才完成。数据链接的ActiveChanged方法也被连接到基类的DefaultRowHeight属性的RowHeightsChanged方法所触发:

```

procedure TMDRecordLink.ActiveChanged;
var
    I: Integer;
begin
    // set number of rows
    RView.RowCount := DataSet.FieldCount;
    // double the height of memo and graphics
    for I := 0 to DataSet.FieldCount - 1 do
        if DataSet.Fields[I] is TBlobField then
            RView.RowHeights[I] := RView.DefaultRowHeight * 2;
    // repaint all...
    RView.Invalidate;
end;

```

在此, 我们遇到了一个小问题。在DefineProperties方法中, TCustomGrid类保存着RowHeights和ColHeights属性的值。可以通过覆盖该方法以及不调用inherited来禁用这个数据流(通常这项技术并不好用), 或者通过切换FSaveCellExtents保护型字段来禁用该特征(就像我在该组件的代码中所做的那样)。

定制DBGrid组件

除了编写新的定制数据敏感组件之外, Delphi程序员通常会定制DBGrid控件。下一个组件的目标就是使用在RecordView组件中相同类型的定制输出来增强DBGrid, 以直接展示图形和备注字段。要想实现这种操作, 网格需要使行高可调, 以便拥有用于图形和合理文本的空间。大家可以在设计时看到这种网格的示例, 如图17.4所示。

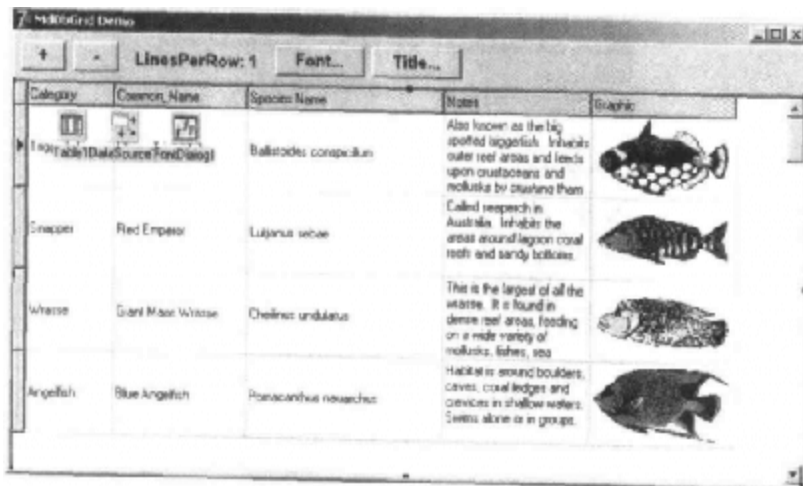


图17.4 MDDBGrid组件在设计时的范例。注意图形和演示字段的输出

然而，创建输出只是简单地改变用于记录浏览器组件的代码，设置网格单元的高度最终成为很难解决的问题。我们将看到用于该操作的代码可能会很少，但它们占用了我们很多时间！

说明：与我们上面使用的一般网格不同，DBGrid在数据集合上是一种虚拟的视图——即显示在屏上的行数和显示在数据集合上的数据行数没有关系。在数据集合的数据记录上滚动时，并不能在DBGrid的行上滚动；当数据从一行移动到下一行时，行是静止的。由于该原因，程序将不会设置单个行的高度来适应数据；但它将所有数据行的高度设置为多行高度值。

这次控件不必创建一个定制的数据链接，因为它可以从已有复杂数据链接的组件中派生出来。新类有新的属性，用于为每行确定文本行的数量并覆盖几个虚拟方法：

```
type
  TmddbGrid = class(TDbGrid)
  private
    FLinesPerRow: Integer;
    procedure SetLinesPerRow (Value: Integer);
  protected
    procedure DrawColumnCell(const Rect: TRect; DataCol: Integer;
      Column: TColumn; State: TGridDrawState); override;
    procedure LayoutChanged; override;
  public
    constructor Create (AOwner: TComponent); override;
  published
    property LinesPerRow: Integer
      read FLinesPerRow write SetLinesPerRow default 1;
  end;
```

构造器将为FLinesPerRow字段设置默认值。这里是对该属性的设置方法：

```
procedure TmddbGrid.SetLinesPerRow(Value: Integer);
begin
  if Value <> FLinesPerRow then
  begin
    FLinesPerRow := Value;
    LayoutChanged;
  end;
end;
```

改变行数的影响是调用LayoutChanged虚拟方法。当许多输出参数被改变时，系统会频繁调用该方法。在该方法的代码中，组件首先调用继承的版本，然后设置每行的高度。作为该计算的基础，它使用了与TCustomDBGrid类相同的公式：文本高度使用当前字体中的Wg范例字来计算（使用该文本是因为它包含高度完整的大写字符，而且带有下行字母的小写字符）。其代码如下：

```
procedure TmddbGrid.LayoutChanged;
var
  PixelsPerRow, PixelsTitle, I: Integer;
```

```

begin
  inherited LayOutChanged;

  Canvas.Font := Font;
  PixelsPerRow := Canvas.TextHeight('Wg') + 3;
  if dgRowLines in Options then
    Inc (PixelsPerRow, GridLineWidth);

  Canvas.Font := TitleFont;
  PixelsTitle := Canvas.TextHeight('Wg') + 4;
  if dgRowLines in Options then
    Inc (PixelsTitle, GridLineWidth);

  // set number of rows
  RowCount := 1 + (Height - PixelsTitle) div (PixelsPerRow * FLinesPerRow);

  // set the height of each row
  DefaultRowHeight := PixelsPerRow * FLinesPerRow;
  RowHeights [0] := PixelsTitle;
  for I := 1 to RowCount - 1 do
    RowHeights [I] := PixelsPerRow * FLinesPerRow;

  // send a WM_SIZE message to let the base component recompute
  // the visible rows in the private UpdateRowCount method
  PostMessage (Handle, WM_SIZE, 0, MakeLong(Width, Height));
end;

```

警告：Font和TitleFont是默认网格设置，该网格能被单独的DBGrid栏对象属性覆盖。该组件现在忽视了那些设置。

这种方法困难的部分就是正确获得最后的语句。可以简单地设置DefaultRowHeight属性，但那种情况下标题行可能太高。首先，我设置DefaultRowHeight，然后是第一行的高度，但这种方法使得在网格中计算可视行的数量（只读VisibleRowCount属性）变得复杂化。如果我们指定行数（为了避免有行隐藏在网格底边之下），基类需要不断重新计算它们。这里是用于画出数据的代码，这些代码从RecordView组件中得到，为适应网格我们稍微做了一定的改变：

```

procedure TMdDbGrid.DrawColumnCell (const Rect: TRect; DataCol: Integer;
  Column: TColumn; State: TGridDrawState);
var
  Bmp: TBitmap;
  OutRect: TRect;
begin
  if FLinesPerRow = 1 then
    inherited DrawColumnCell(Rect, DataCol, Column, State)
  else
    begin
      // clear area
      Canvas.FillRect (Rect);
      // copy the rectangle
    end
  end;

```

```
OutRect := Rect;
// restrict output
InflateRect (OutRect, -2, -2);
// output field data
if Column.Field is TGraphicField then
begin
  Bmp := TBitmap.Create;
  try
    Bmp.Assign (Column.Field);
    Canvas.StretchDraw (OutRect, Bmp);
  finally
    Bmp.Free;
  end;
end
else if Column.Field is TMemoField then
begin
  DrawText (Canvas.Handle, PChar (Column.Field.AsString),
    Length (Column.Field.AsString), OutRect, dt_WordBreak or dt_NoPrefix)
end
else // draw single line vertically centered
  DrawText (Canvas.Handle, PChar (Column.Field.DisplayText),
    Length (Column.Field.DisplayText), OutRect,
    dt_vcenter or dt_SingleLine or dt_NoPrefix);
end;
end;
```

在上面的代码中大家可以看到，如果用户只显示一行，则网格使用不含备注和图形字段输出的标准绘图技术。然而，只要增加行数就能看到更好的输出。

为了看到该代码的实际效果，运行GridDemo示例。该程序有两个可以用来增加或减少网格行高的按钮，并有多改变字体的按钮。这是一个重要的测试，因为每个单元高度的像素值是字体高度乘以行数。

建立定制的数据集合

在第13章“Delphi的数据库体系结构”中讨论TDataSet类和Delphi上的可选数据集合组件家族时，我曾经提到过编写定制数据集合类的可能性。现在来看一下实际的示例。编写定制数据集合的原因是不需要配置数据库引擎，但仍能利用Delphi的数据库结构，包括持续数据库字段和数据敏感控件。

编写定制的数据集合对一个组件开发人员来说是最复杂的任务之一，因此这是本书中最高级的地方。而且，Borland还没有发布任何关于编写定制数据集合的官方文档。如果读者以前使用过Delphi，则可以跳过该章的剩余部分并稍后返回到这里。

TDataSet类是一种抽象类，该类声明了几个虚拟抽象方法，现在已很少使用，因为大部分已经被空的虚拟方法所替代（仍然必须要覆盖）。TDataSet的每个子类必须覆盖所有那些

方法。

在讨论定制数据集合的开发之前，我们需要在特殊的记录缓冲器上探讨TDataSet类的几个技术元素。该类维护着一个缓冲器的列表，该列表存储着不同记录的值。这些缓冲器在存储实际数据的同时，也为管理记录时使用的数据集合存储更多的信息。这些缓冲器没有预定义结构，必须由每个定制数据集合来分配缓冲器，填充它们，并撤销它们。定制数据集合也必须从记录缓冲器中复制数据到数据集合的各个字段，反之亦然。换句话说，定制数据集合负责处理这些缓冲器。

除了管理数据缓冲器之外，该组件也负责在记录间导航，管理书签，定义数据集合的结构并创建正确的数据字段。TDataSet类与框架无关，必须用合适的代码填充它。幸运的是，多数代码遵循标准结构，该结构被派生的TDataSetVCL类使用。一旦理解了关键点，就能通过借用相当多的代码来建立多个定制数据集合。

为了简化该类型的重复使用，我收集了TMDCustomDataSet类中任意定制数据集合所要求的通用特性。然而，我并不会首先讨论基类和后面的特殊实现，因为那相当难以理解。相反，我将详细讲解数据集合所需要的代码，同时根据逻辑流程展示通用类和特殊类的方法。

类的定义

像往常一样，首先声明本小节中讨论的两个类：通用的定制数据集合和在流中存储数据的特殊组件。这些类的声明见程序清单17.2。除了虚拟方法之外，类中还包含用来管理缓冲器、跟踪当前位置和记录数目并处理许多其他特征的一系列保护型字段。同时在开始处还声明了另一个记录：一种用来为放置在缓冲器上的每个数据存储附加数据的结构。数据集合将根据实际数据在每个记录缓冲器上放置该信息。

程序清单17.2 TMdCustomDataSet和TMdDataSetStream的声明

```
// in the unit MdDsCustom
type
  EMdDataSetError = class (Exception);

  TMdRecInfo = record
    Bookmark: Longint;
    BookmarkFlag: TBookmarkFlag;
  end;
  PMdRecInfo = ^TMdRecInfo;

  TMdCustomDataSet = class (TDataSet)
  protected
    // status
    FIsTableOpen: Boolean;
    // record data
    FRecordSize,           // the size of the actual data
    FRecordBufferSize,    // data + housekeeping (TRecInfo)
    FCurrentRecord,        // current record (0 to FRecordCount - 1)
    BofCrack,              // before the first record (crack)
    EofCrack: Integer;     // after the last record (crack)
```

```

// create, close, and so on
procedure InternalOpen; override;
procedure InternalClose; override;
function IsCursorOpen: Boolean; override;
// custom functions
function InternalRecordCount: Integer; virtual; abstract;
procedure InternalPreOpen; virtual;
procedure InternalAfterOpen; virtual;
procedure InternalLoadCurrentRecord(Buffer: PChar); virtual; abstract;
// memory management
function AllocRecordBuffer: PChar; override;
procedure InternalInitRecord(Buffer: PChar); override;
procedure FreeRecordBuffer(var Buffer: PChar); override;
function GetRecordSize: Word; override;
// movement and optional navigation (used by grids)
function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
    TGetResult; override;
procedure InternalFirst; override;
procedure InternalLast; override;
function GetRecNo: Longint; override;
function GetRecordCount: Longint; override;
procedure SetRecNo(Value: Integer); override;
// bookmarks
procedure InternalGotoBookmark(Bookmark: Pointer); override;
procedure InternalSetToRecord(Buffer: PChar); override;
procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
// editing (dummy versions)
procedure InternalDelete; override;
procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
procedure InternalPost; override;
procedure InternalInsert; override;
// other
procedure InternalHandleException; override;
published
    // redeclared dataset properties
    property Active;
    property BeforeOpen;
    property AfterOpen;
    property BeforeClose;
    property AfterClose;
    property BeforeInsert;
    property AfterInsert;

```

```

    property BeforeEdit;
    property AfterEdit;
    property BeforePost;
    property AfterPost;
    property BeforeCancel;
    property AfterCancel;
    property BeforeDelete;
    property AfterDelete;
    property BeforeScroll;
    property AfterScroll;
    property OnCalcFields;
    property OnDeleteError;
    property OnEditError;
    property OnFilterRecord;
    property OnNewRecord;
    property OnPostError;
end;

// in the unit MdDsStream
type
  TMdDataFileHeader = record
    VersionNumber: Integer;
    RecordSize: Integer;
    RecordCount: Integer;
  end;

  TMdDataSetStream = class(TMdCustomDataSet)
  private
    procedure SetTableName(const Value: string);
  protected
    FDataFileHeader: TMdDataFileHeader;
    FDataFileHeaderSize, // optional file header size
    FRecordCount: Integer; // current number of records
    FStream: TStream; // the physical table
    FTableName: string; // table path and file name
    FFieldOffset: TList; // field offsets in the buffer
  protected
    // open and close
    procedure InternalPreOpen; override;
    procedure InternalAfterOpen; override;
    procedure InternalClose; override;
    procedure InternalInitFieldDefs; override;
    // edit support
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
    procedure InternalPost; override;
    procedure InternalInsert; override;
    // fields

```



```

procedure SetFieldData(Field: TField; Buffer: Pointer); override;
// custom dataset virtual methods
function InternalRecordCount: Integer; override;
procedure InternalLoadCurrentRecord(Buffer: PChar); override;
public
    procedure CreateTable;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
published
    property TableName: string read FTableName write SetTableName;
end;

```

当我把这些方法分成几部分时（就像大家通过源代码文件中看到的那样），使用了罗马数字来标记每个部分。在描述方法的注释中可以看到这些数字，这样在浏览这个长长的代码清单时，将能立即知道自己位于哪一部分。

部分I：初始化、打开和关闭

要讨论的第一种方法用来初始化数据集合，并负责打开和关闭保存数据的文件流。除了初始化组件的内部数据外，这些方法还负责初始化并将正确的TFields对象连接到数据集合组件中。为了完成该工作，需要用字段的定义为数据集合初始化FieldsDef属性，然后调用一些标准的方法来生成并绑定TField对象。下面是普通的InternalOpen方法：

```

procedure TMDCustomDataSet.InternalOpen;
begin
    InternalPreOpen; // custom method for subclasses

    // initialize the field definitions
    InternalInitFieldDefs;

    // if there are no persistent field objects, create the fields dynamically
    if DefaultFields then
        CreateFields;
    // connect the TField objects with the actual fields
    BindFields (True);

    InternalAfterOpen; // custom method for subclasses

    // sets cracks and record position and size
    BofCrack := -1;
    EofCrack := InternalRecordCount;
    FCurrentRecord := BofCrack;

    FRecordBufferSize := FRecordSize + sizeof (TMDRecInfo);
    BookmarkSize := sizeof (Integer);

    // everything OK: table is now open
    FIsTableOpen := True;
end;

```

注意，该方法设置了该类的大部分本地字段和基类TDataSet的BookmarkSize字段。在这种方法中，我调用了在我介绍自己定制的数据集合层次时曾提过的两个定制方法：InternalPreOpen和InternalAfterOpen。首先InternalPreOpen用于最开始要求的操作，如检查数据集合是否能真正打开并从文件读取标题信息。该代码也能检查当表格第一次创建时，用于连续保存值的内部版本号，大家随后将会看到这一点。通过在该方法内产生一个异常，最终能停止打开操作。

下面是基于流的派生数据集合中这两种方法的代码：

```

const
    HeaderVersion = 10;

procedure TmdDataSetStream.InternalPreOpen;
begin
    // the size of the header
    FDataFileHeaderSize := sizeof (TmdDataFileHeader);

    // check if the file exists
    if not FileExists (FTableName) then
        raise EmdDataSetError.Create ('Open: Table file not found');

    // create a stream for the file
    FStream := TFileStream.Create (FTableName, fmOpenReadWrite);

    // initialize local data (loading the header)
    FStream.ReadBuffer (FDataFileHeader, FDataFileHeaderSize);
    if FDataFileHeader.VersionNumber <> HeaderVersion then
        raise EmdDataSetError.Create ('Illegal File Version');

    // let's read this, double check later
    FRecordCount := FDataFileHeader.RecordCount;

end;

procedure TmdDataSetStream.InternalAfterOpen;
begin
    // check the record size
    if FDataFileHeader.RecordSize <> FRecordSize then
        raise EmdDataSetError.Create ('File record size mismatch');

    // check the number of records against the file size
    if (FDataFileHeaderSize + FRecordCount * FRecordSize) <> FStream.Size then
        raise EmdDataSetError.Create ('InternalOpen: Invalid Record Size');

end;

```

第二种方法就是InternalAfterOpen，用于执行在字段定义被设置之后的操作，它后面跟随着一段代码，用于将读自文件的记录大小与InternalInitFieldDefs方法中计算出的值进行比较。该代码也会检查读取自标题的记录数是否与文件的实际大小兼容。如果数据集合非正常关闭，该测试可能失败：需要修改该代码，以便让数据集合在标题上重新刷新记录大小。

定制数据集合类的InternalOpen方法专门负责调用InternalInitFieldDefs，它决定着字段的定义（在设计时或运行时）。对于该示例，我决定以一个外部文件——一个简单的INI文件上的字段定义为基础，该INI文件能够为每个字段提供一个部分。每部分都包含字段的名字

和数据类型以及它的大小（如果它是字符串数据）。程序清单17.3是我们将在组件的演示应用程序中使用的Contrib.INI文件。

程序清单17.3 用于Demo应用程序的Contrib.INI文件

```
[Fields]
Number = 6

[Field1]
Type = ftString
Name = Name
Size = 30

[Field2]
Type = ftInteger
Name = Level

[Field3]
Type = ftDate
Name = BirthDate

[Field4]
Type = ftCurrency
Name = Stipend

[Field5]
Type = ftString
Name = Email
Size = 50

[Field6]
Type = ftBoolean
Name = Editor
```

该文件或类似的文件必须使用相同的名字作为表格文件，并且与之位于相同的目录下。**InternalInitFieldDefs**方法（显示在程序清单17.4中）将读取它，使用它发现的值来建立字段定义和确定每个记录的大小。该方法还初始化内部TList对象（存储记录内的每个字段的偏移）。我们将使用该TList来访问记录缓冲器内的字段数据，如程序清单17.4所示。

程序清单17.4 基于流的数据集合的InternalInitFieldDefs方法

```
procedure TMDDataSetStream.InternalInitFieldDefs;
var
  IniFileName, FieldName: string;
  IniFile: TIniFile;
  nFields, I, TmpFieldOffset, nSize: Integer;
  FieldType: TFieldType;
begin
  FFieldOffset := TList.Create;
  FieldDefs.Clear;
  TmpFieldOffset := 0;
```

```

IniFilename := ChangeFileExt(FTableName, '.ini');
IniFile := TIniFile.Create (IniFilename);
// protect INI file
try
  nFields := IniFile.ReadInteger (' Fields', 'Number', 0);
  if nFields = 0 then
    raise EDataSetOneError.Create (' InitFieldsDefs: 0 fields?');
  for i := 1 to nFields do
    begin
      // create the field
      FieldType := TFieldType (GetEnumValue (TypeInfo (TFieldType),
        IniFile.ReadString (' Field' + IntToStr (I), 'Type', '')));
      FieldName := IniFile.ReadString ('Field' + IntToStr (I), 'Name', ' ');
      if FieldName = '' then
        raise EDataSetOneError.Create (
          'InitFieldsDefs: No name for field ' + IntToStr (I));
      nSize := IniFile.ReadInteger ('Field' + IntToStr (I), 'Size', 0);
      FieldDefs.Add (FieldName, FieldType, nSize, False);
      // save offset and compute size
      FFieldOffset.Add (Pointer (TmpFieldOffset));
      case FieldType of
        ftString:                Inc (TmpFieldOffset, nSize + 1);
        ftBoolean, ftSmallInt, ftWord:  Inc (TmpFieldOffset, 2);
        ftInteger, ftDate, ftTime:      Inc (TmpFieldOffset, 4);
        ftFloat, ftCurrency, ftDateTime: Inc (TmpFieldOffset, 8);
      else
        raise EDataSetOneError.Create (
          'InitFieldsDefs: Unsupported field type');
      end;
    end; // for
  finally
    IniFile.Free;
  end;
  FRecordSize := TmpFieldOffset;
end;

```

关闭表格是断开字段连接的一种方法（使用一些标准的定义）。每个类必须处理所分配的数据，并更新文件的标题、记录首次被添加的时间和每次记录数被改变的时间：

```

procedure TMDCustomDataSet.InternalClose;
begin
  // disconnect field objects
  BindFields (False);
  // destroy field object (if not persistent)
  if DefaultFields then
    DestroyFields;

```

```

    // close the file
    FIsTableOpen := False;
end;

procedure TMdDataSetStream.InternalClose;
begin
    // if required, save updated header
    if (FDataFileHeader.RecordCount <> FRecordCount) or
        (FDataFileHeader.RecordSize = 0) then
        begin
            FDataFileHeader.RecordSize := FRecordSize;
            FDataFileHeader.RecordCount := FRecordCount;
            if Assigned (FStream) then
                begin
                    FStream.Seek (0, soFromBeginning);
                    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
                end;
            end;
            // free the internal list field offsets and the stream
            FFieldOffset.Free;
            FStream.Free;
            inherited InternalClose;
        end;

```

另一个相关的功能就是测试数据集合是否被打开，可以使用相应的本地字段解决一些问题：

```

function TMdCustomDataSet.IsCursorOpen: Boolean;
begin
    Result := FIsTableOpen;
end;

```

这些是在任何数据集合中实现打开和关闭操作的方法。然而，更多的是要添加一种方法来创建表格。在这个例子里，**CreateTable**方法创建了一个空的文件，并将下列信息插入到标题中：一个固定的版本号、一个伪记录大小（在初始化字段之前，不需要知道其大小）以及记录的数量（以0作为开始）：

```

procedure TMdDataSetStream.CreateTable;
begin
    CheckInactive;
    InternalInitFieldDefs;

    // create the new file
    if FileExists (FTableName) then
        raise EMdDataSetError.Create ('File ' + FTableName + ' already exists');
    FStream := TFileStream.Create (FTableName, fmCreate or fmShareExclusive);
    try
        // save the header
        FDataFileHeader.VersionNumber := HeaderVersion;

```

```

    FDataFileHeader.RecordSize := 0;    // used later
    FDataFileHeader.RecordCount := 0;   // empty
    FStream.WriteBuffer (FDataFileHeader, FDataFileHeaderSize);
  finally
    // close the file
    FStream.Free;
  end;
end;

```

部分 II：移动和书签管理

正如我们先前提到的一样，每个数据集合都必须实现书签管理，该管理对数据集合导航很必要。从逻辑上来看，一个书签能够引用数据集合的特殊记录，它可以惟一确定记录，以便数据集合能访问它并把它与其他记录相比较。从技术上来说，书签就是指针。我们可以把它们作为指针来确定保存记录信息的数据结构，或者将其作为简单的记录编号。为了简化起见，将使用后一种方法。

给定一个书签，就可以找到所对应的记录；而且给定一个记录缓冲器，也应能检索出对应的书签。这就是将TMdRecInfo结构追加到每个记录缓冲器的记录数据上的原因。该数据结构能为缓冲器中的记录保存书签，并保存一些定义如下的书签标记：

```

type
  TBookmarkFlag = (bfCurrent, bfBOF, bfEOF, bfInserted);

```

系统将会要求人们在每个记录缓冲器中保存这些标签，随后还将要求人们为给定的记录缓冲器恢复标签。

总之，记录缓冲器的结构用于保存数据、书签和书签标记，如图17.5所示。

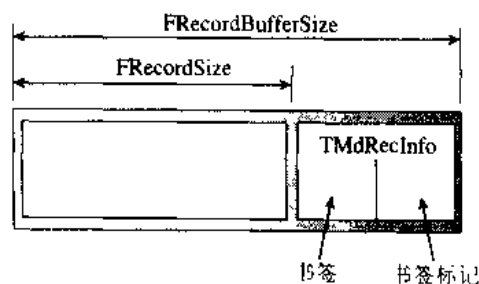


图17.5 定制数据集合的缓冲器结构，用局部字段表示其各个子部分

为了访问书签和标记，我们将简单使用实际数据的尺寸偏移，捕捉PMdRecInfo指针类型的值，然后通过指针访问TMdRecInfo结构的正确字段。用于设置和获得书签标记的两种方法说明了该技术：

```

procedure TMdCustomDataSet.SetBookmarkFlag (Buffer: PChar;
  Value: TBookmarkFlag);
begin
  PMdRecInfo(Buffer + FRecordSize).BookmarkFlag := Value;
end;

```

```

function TMDCustomDataSet.GetBookmarkFlag (Buffer: PChar): TBookmarkFlag;
begin
    Result := PMdRecInfo(Buffer + FRecordSize).BookmarkFlag;
end;

```

我们用来设置和获得记录当前书签的方法类似前面的两种方法，但它们更复杂，因为我们在Data参数中接收书签的指针。通过捕捉该指针作为整数指针，我们可以获得书签值：

```

procedure TMDCustomDataSet.GetBookmarkData (Buffer: PChar; Data: Pointer);
begin
    Integer(Data^) := PMdRecInfo(Buffer + FRecordSize).Bookmark;
end;

procedure TMDCustomDataSet.SetBookmarkData (Buffer: PChar; Data: Pointer);
begin
    PMdRecInfo(Buffer + FRecordSize).Bookmark := Integer(Data^);
end;

```

主要的书签管理方法是InternalGotoBookmark，数据集合用该方法将给定记录作为当前记录。注意，这不是标准的导航技术，因为它通常是移动到下一个或前一个记录（可以使用下一章节中介绍的GetRecord方法完成它），或移动到第一个或最后一个记录（可以使用InternalFirst和InternalLast方法完成）。

很奇怪，InternalGotoBookmark方法不获得书签参数，而是指向书签的指针，因此必须对它解除引用来确定书签值。随后的InternalSetToRecord方法用来跳到给定的书签，但首先它必须将书签作为参数从记录缓冲器中提出。然后通过InternalSetToRecord调用InternalGotoBookmark。下面是这两种方法：

```

procedure TMDCustomDataSet.InternalGotoBookmark (Bookmark: Pointer);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := Integer (Bookmark^);
    if (ReqBookmark >= 0) and (ReqBookmark < InternalRecordCount) then
        FCurrentRecord := ReqBookmark
    else
        raise EMdDataSetError.Create ('Bookmark ' +
            IntToStr (ReqBookmark) + ' not found');
    end;

procedure TMDCustomDataSet.InternalSetToRecord (Buffer: PChar);
var
    ReqBookmark: Integer;
begin
    ReqBookmark := PMdRecInfo(Buffer + FRecordSize).Bookmark;
    InternalGotoBookmark (@ReqBookmark);
end;

```

除了刚才描述的书签管理方法外，还可以使用其他几种导航方法，以便能够在数据集合内移动到指定的位置，如第一个或最后一个记录。实际上，这两种方法并不是真正将当前记录指针移动到第一个或最后一个记录，而是移动到第一个记录前与最后一个记录后的特定位置上，这些并不是实际记录，Borland称之为crack。因为第一个记录的位置为0，所以文件开始的crack或BofCrack为-1（在InternalOpen方法中进行设置）。因为最后一个记录有位置FRecordCount-1，所以文件末尾的crack或EofCrack有记录标号的值。我使用EofCrack和BofCrack两个本地字段使该代码更容易阅读：

```

procedure TMDCustomDataSet.InternalFirst;
begin
    FCurrentRecord := BofCrack;
end;

procedure TMDCustomDataSet.InternalLast;
begin
    EofCrack := InternalRecordCount;
    FCurrentRecord := EofCrack;
end;

```

InternalRecordCount方法是我在TMDCustomDataSet类中引入的一种虚拟方法，因为不同的数据集合可以含有一个用于该值的局部字段（如同基于流的数据集合有FRecordCount字段）或自由地计算它。

另一组可选的方法用来获得当前记录编号（被DBGrid组件用来显示相称的垂直工具栏），设置当前记录编号或确定记录号。这些方法很容易理解，如果大家还记得内部FCurrentRecord字段的记录编号范围是从0到-1。相对应的记录编号向系统报告的范围是从1到记录的条数：

```

function TMDCustomDataSet.GetRecordCount: Longint;
begin
    CheckActive;
    Result := InternalRecordCount;
end;

function TMDCustomDataSet.GetRecNo: Longint;
begin
    UpdateCursorPos;
    if FCurrentRecord < 0 then
        Result := 1
    else
        Result := FCurrentRecord + 1;
end;

procedure TMDCustomDataSet.SetRecNo(Value: Integer);
begin
    CheckBrowseMode;
    if (Value > 1) and (Value <= FRecordCount) then
        begin

```



```

    FCurrentRecord := Value - 1;
    Resync([]);
end;
end;

```

注意它是一般的定制数据集合类，该类实现了该部分的所有方法。基于流派生的数据集合不需要修改它们。

部分Ⅲ：记录缓冲器和字段管理

我们已经介绍了所有的支持方法，现在来分析一下定制数据集合的核心内容。除了打开和创建这些记录并在它们间相互移动之外，组件还需要将数据从流（持续文件）移动到记录缓冲器中，并从记录缓冲器将数据移动到连接数据敏感控件的TField对象内。记录缓冲器的管理十分复杂，因为每个数据集合也需要分配、清空和释放它所需的内存：

```

function TmdCustomDataSet AllocRecordBuffer: PChar;
begin
    GetMem (Result, FRecordBufferSize);
end;

procedure TmdCustomDataSet.FreeRecordBuffer (var Buffer: PChar);
begin
    FreeMem (Buffer);
end;

```

分配内存的原因是数据集合通常需要将更多的信息添加到记录缓冲器，因此系统没办法知道分配多少内存。注意AllocRecordBuffer方法，该组件可为记录缓冲器分配内存，包括数据库数据和记录信息。在InternalOpen方法中我们编写了以下代码：

```
FRecordBufferSize := InternalRecordSize + sizeof (TmdRecInfo);
```

组件也需要实现一种重设置缓冲器的函数InternalInitRecord，通常用数字0或空来填充。

我们也必须实现一种方法，用于返回每个记录的大小，不只是数据部分而是整个记录缓冲器。该方法是执行只读RecordSize属性的必需方法，它将用在整个VCL源代码上的两种特殊情况中。在一般的定制数据集合上，GetRecordSize方法返回FRecordSize字段的值。

我们实际接触到了定制数据集合组件的核心部分。这些方法包括：GetRecord（从文件上读取数据）、InternalPost和InternalAddRecord（更新或添加新数据到文件），以及InternalDelete（删除数据，但没有在我们的数据集合范例中实现）。

该组中最复杂的方法可能是GetRecord，该方法有多种用途。事实上，系统使用该方法来为当前记录检索数据，填充一个被作为参数传递的缓冲区并检索下一个数据或前面的记录。GetMode参数决定了这种作用：

```

type
    TGetMode = (gmCurrent, gmNext, gmPrior);

```

当然，前一个或下一个记录可能不存在。甚至当前记录也不存在，例如，当表格为空时（或在内部出错的情况下）。在这些情况下，我们不恢复数据而是返回错误代码。因此该方法的结果可以是下列值：

```
type
```

```
TGetResult = (grOK, grBOF, grEOF, grError);
```

检查请求的记录是否存在与我们所想像的有点不同。我们不必确定当前记录是否在正确范围内，只需要确定所请求的记录是否在范围内即可。例如，在case语句的gmCurrent分支中，我们使用标准的表达式CurrentRecord>=InternalRecordCount。为了完全理解不同的情况，读者可能需要再阅读一次代码。

几年前我编写第一个定制数据集时，颇费了一番功夫（因为递归调用引起了系统瘫痪）。为了测试它，假设如果我们使用DBGrid，系统将执行一系列GetRecord调用，直到网格满了或GetRecord返回grEOF。下面是GetRecord方法的全部代码：

```
// III: Retrieve data for current, previous, or next record
// (moving to it if necessary) and return the status
function TMDCustomDataSet.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result := grOK; // default
  case GetMode of
    gmNext: // move on
      if FCurrentRecord < InternalRecordCount - 1 then
        Inc (FCurrentRecord)
      else
        Result := grEOF; // end of file
    gmPrior: // move back
      if FCurrentRecord > 0 then
        Dec (FCurrentRecord)
      else
        Result := grBOF; // begin of file
    gmCurrent: // check if empty
      if FCurrentRecord >= InternalRecordCount then
        Result := grError;
  end;
  // load the data
  if Result = grOK then
    InternalLoadCurrentRecord (Buffer)
  else if (Result = grError) and DoCheck then
    raise EMDDataSetError.Create ('GetRecord: Invalid record');
end;
```

如果有一个错误，同时DoCheck参数是True，则GetRecord会产生一个例外。如果在记录选择期间每件事都正常，则组件从流上装载数据，移动到当前记录的位置（记录大小乘以记录数）。另外，我们需要用正确的书签标记和书签值（或记录数）初始化缓冲区。这项工作由我们引入的另一个虚拟方法完成，因此派生类将只实现代码的这个部分，而复杂的GetRecord方法保持不变：

```

procedure TMDDataSetStream.InternalLoadCurrentRecord (Buffer: PChar);
begin
    FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
    FStream.ReadBuffer (Buffer^, FRecordSize);
    with PMdRecInfo(Buffer + FRecordSize)^ do
        begin
            BookmarkFlag := bfCurrent;
            Bookmark := FCurrentRecord;
        end;
    end;

```

有两种不同的情况需要把数据移入文件：修改当前记录（编辑后传递）或添加新记录（插入或附加后传递）。在这两种情况下都可使用InternalPost方法，但应通过检查数据集合的State属性来确定正在执行的传递类型。而且，由于这里不接收记录缓冲器作为参数，因此必须使用TDataSet的ActiveRecord属性，将当前记录指向缓冲器：

```

procedure TMDDataSetStream.InternalPost;
begin
    CheckActive;
    if State = dsEdit then
        begin
            // replace data with new data
            FStream.Position := FDataFileHeaderSize + FRecordSize * FCurrentRecord;
            FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
        end
    else
        begin
            // always append
            InternalLast;
            FStream.Seek (0, soFromEnd);
            FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
            Inc (FRecordCount);
        end;
    end;

```

另外，还有一种相关的方法：InternalAddRecord。该方法被AddRecord方法调用，然后依次被InsertRecord和AppendRecord方法调用。后两个方法是用户能调用的公共方法。因为InsertRecord和AppendRecord调用将收到的字段值作为参数，所以这是将新记录选择插入或附加到数据集合中、编辑各种字段的值或传递数据的另一种方法。这时必须要做的就是复制在InternalPost方法中添加一个新记录的代码：

```

procedure TMDDataSetOne.InternalAddRecord(Buffer: Pointer; Append: Boolean);
begin
    // always append at the end
    InternalLast;
    FStream.Seek (0, soFromEnd);

```

```

FStream.WriteBuffer (ActiveBuffer^, FRecordSize);
Inc (FRecordCount);
end;

```

最后应该实现的文件操作是删除当前记录。该操作很普通但实际上很复杂。如果我们采用一种简单的方法，如在文件上创建空点，则需要跟踪该点并使代码围绕该点读或写这些特殊记录。一种可选的解决方法是复制不包括给定记录的全部文件，然后用复制文件代替原始文件。我们认为对于该范例而言，不需要对删除记录加以支持。

部分 IV：从缓冲器到字段

在前面几个方法中，我们看到了数据集合是怎样将数据从数据文件移动到内存缓冲区的。然而，Delphi对该记录缓冲器不能做什么，因为它不知道怎样在缓冲器解释数据。我们需要再提供两种方法：**GetData**——能将数据从记录缓冲器复制到数据集合的字段对象中，**SetData**——能将数据从字段移回到记录缓冲器中。Delphi将为我们自动做的是从字段对象移动数据到数据敏感控件并返回。

这两个对象方法的代码不是很复杂，主要因为我们在TList对象的记录数据内保存了字段偏移FFieldOffset。通过递增指向当前字段偏移的记录缓冲器中初始位置的指针，我们能够获得指定的数据，这些数据的大小是Field.DataSize字节。

这两种方法中一个容易令人混淆的因素是，它们都接收Field参数和Buffer参数。首先，作为参数传递的缓冲器是记录缓冲器。实际上，Buffer是一个指向字段对象的行数据的指针。如果我们使用一种字段对象的方法来移动数据，它将调用数据集合的GetData或SetData方法，而可能引起一个无限递归。相反，我们应使用ActiveBuffer指针访问记录缓冲器，在记录缓冲器上使用正确的偏移来获得用于当前字段的数据，然后使用已提供的Buffer访问字段数据。两种方法间惟一的不同就是我们移动数据的方向：

```

function TmdDataSetOne.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FieldOffset: Integer;
  Ptr: PChar;
begin
  Result := False;
  if not IsEmpty and (Field.FieldNo > 0) then
  begin
    FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
    Ptr := ActiveBuffer;
    Inc (Ptr, FieldOffset);
    if Assigned (Buffer) then
      Move (Ptr^, Buffer^, Field.DataSize);
    Result := True;
    if (Field is TDateTimeField) and (Integer(Ptr^)=0) then
      Result := False;
  end;
end;

```

```

procedure TMDDataSetOne.SetFieldData(Field: TField; Buffer: Pointer);
var
    FieldOffset: Integer;
    Ptr: PChar;
begin
    if Field.FieldNo >= 0 then
        begin
            FieldOffset := Integer (FFieldOffset [Field.FieldNo - 1]);
            Ptr := ActiveBuffer;
            Inc (Ptr, FieldOffset);
            if Assigned (Buffer) then
                Move (Buffer^, Ptr^, Field.DataSize)
            else
                raise Exception.Create (
                    'Very bad error in TMDDataSetStream.SetField data');
                DataEvent (deFieldChange, Longint(Field));
            end;
        end;

```

GetField方法应返回True或False来说明字段的类型是包含数据还是空的。然而，除非我们为空白字段使用特殊标记，否则很难确定该字段的类型，因此我们存储不同数据类型的值。例如，一个测试，如Ptr^<>#0，只有我们为所有字段使用字符串时才有意义。如果我们使用该测试，则零整数和空字符串都将作为零值显示（数据敏感控件是空的），这正是我们所希望的。问题是Boolean False值不会显示出来。更糟的是没有小数的浮点值和几个数字也不会显现出来，因为该表示的说明部分将是零！然而，为了使该示例在Delphi 6上工作，必须假设每个空日期/时间字段初始值为零。没有该代码，Delphi会试图非法转换内部零日期（日期字段在内部不使用TDateTime数据类型，而是一个不同的表示），从而引起异常。这些代码用来与Delphi的老版本一起工作。

警告：当试图修正问题时，我也发现如果我们为字段调用IsNull，则该请求会通过调用GetFieldData来解决，而不需要通过任何缓冲器来填充，只需要查看函数调用的结果。这就是在代码内包含Assigned测试的原因。

最后的方法是InternalHandleException，它与任何种类无关。通常，该方法用于抑制异常状态，因为它只在设计时才被激活。

测试基于流的数据集合

完成所有工作后，我们准备测试含有定制数据集合组件的应用程序，该应用程序安装在组件的包上。StreamDSDemo程序显示的窗体很简单，如图17.6所示。其面板上有两个按钮、一个复选框、一个导航组件和一个填充客户空间的DBGrid。

图17.6显示的是设计时的窗体，但我们已激活定制数据集合以便它的数据可视。而且，我们也准备好了带有表格定义的INI文件（当讨论数据集合定义时，我们已列出该文件），并且通过执行该程序向文件中添加一些数据。

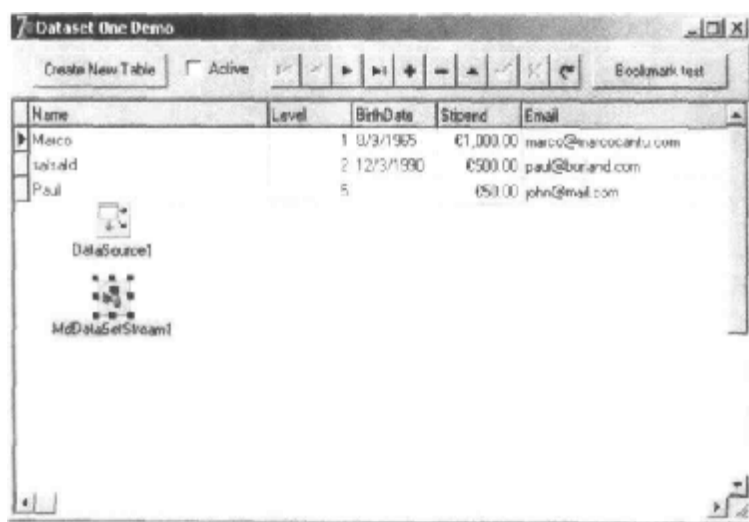


图17.6 StreamDSDemo的窗体。定制数据集合已经激活，因此可以看到设计时的数据

使用Delphi的字段编辑器也可能修改窗体并设置各种字段对象的属性。当它处理一个标准数据集合控件时，一切正常！然而，为了使该数据集合工作，我们需要在TableName属性上输入名字，亦即使用完整路径。

警告：因为演示程序要在设计时定义表格文件的抽象路径，所以如果我们复制示例到不同的驱动器或目录，将需要修改它。示例中TableName属性只在设计时使用。事实上在运行时，该程序要在当前目录中查找该表格。

程序的代码很简单，特别是与定制数据集合的代码相比较。如果表格仍不存在，我们能单击Create New Table按钮：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    MdDataSetStream1.CreateTable;
    MdDataSetStream1.Open;
    CheckBox1.Checked := MdDataSetStream1.Active;
end;
```

注意，我们这里首先是创建文件、在CreateTable调用中打开并关闭该文件，然后是打开表格。这和TTable组件有同样的功能（该组件使用CreateTable方法完成它）。为了打开或关闭表格，我们能单击复选框：

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
    MdDataSetStream1.Active := CheckBox1.Checked;
end;
```

最后我们创建了一种方法，用于测试定制数据集合的书签管理代码（它很有效）。

数据集中的目录

在Delphi中，另一个与数据集相关的想法是用它们简单表示数据的集合，而不管该数据来自何方。SQL服务器或本地文件就是传统数据集的例子，但我们能使用相同的技术来显示系统用户的列表、文件夹的文件列表、对象特性、基于XML的数据等等。

例如，本章出现的第二个数据集就是一个文件列表。我在内存中基于对象列表建立了一般的数据集（使用TObjectList），然后派生出一个对象对应的文件夹文件的版本。实际示例因为它是只读数据集而简化，因此我们甚至可以发现它比以前展示的数据集更简单。

说明：这里出现的一些想法在我为Borland Community站点编写的文章中讨论过，并于2000年6月发表，见bdn.borland.com/article/0,1410,20587,00.html。

数据集的列表

一般基于列表的数据集叫做TMdListDataSet，它包含对象列表，该列表在我们打开数据集时被创建并且在关闭它时被释放。该数据集不在缓冲器内存存储实际记录数据，而只在缓冲器中保存对应于记录数据的表项在列表中的位置。下面是该类的定义：

```
type
  TMdListDataSet = class (TMdCustomDataSet)
  protected
    // the list holding the data
    FList: TObjectList;
    // dataset virtual methods
    procedure InternalPreOpen; override;
    procedure InternalClose; override;
    // custom dataset virtual methods
    function InternalRecordCount: Integer; override;
    procedure InternalLoadCurrentRecord (Buffer: PChar); override;
  end;
```

通过编写一般的定制数据类，我们能覆盖TDataSet类和该定制数据集类的几个虚拟方法，并拥有一个工作中的数据集（尽管它仍是抽象类，但该类要求来自子类的附加代码可用）。当数据集被打开时，我们不得不创建列表并设置记录的尺寸，以说明正在缓冲器中简单地保存列表索引：

```
procedure TMdListDataSet.InternalPreOpen;
begin
  FList := TObjectList.Create (True); // owns the objects
  FRecordSize := 4; // an integer, the list item id
end;
```

此外，这个类型的子类也应用对象填充列表。

提示：类似于ClientDataSet，我们的列表数据集在内存中保存所有数据。然而，使用一些技巧，我们也能创建“假”对象列表，然后只在我们访问它们时装载真实对象。

关闭就是简单地释放列表，这个列表有一个记录数对应着列表尺寸：

```
function TMDListDataSet.InternalRecordCount: Integer;
begin
    Result := fList.Count;
end;
```

只有另一种方法用于在记录缓冲区中保存当前记录的数据，包含书签信息。核心数据只是当前记录的位置，该位置应匹配列表索引（或者书签）：

```
procedure TMDListDataSet.InternalLoadCurrentRecord (Buffer: PChar);
begin
    PInteger (Buffer)^ := fCurrentRecord;
    with PMdRecInfo(Buffer + FRecordSize)^ do
    begin
        BookmarkFlag := bfCurrent;
        Bookmark := fCurrentRecord;
    end;
end;
```

目录数据

派生的目录数据集类提供了一种方法，即当数据集打开时，在内存上装载对象，以定义正确的字段并读和写那些字段的值。当然，它也是一个属性，指示工作目录或更精确地说是目录加上用于过滤文件的标记（如在c:\docs*.txt）：

```
type
    TMDDirDataset = class (TMDListDataSet)
    private
        FDirectory: string;
        procedure SetDirectory(const NewDirectory: string);
    protected
        // TDataSet virtual methods
        procedure InternalInitFieldDefs; override;
        procedure SetFieldData(Field: TField; Buffer: Pointer); override;
        function GetCanModify: Boolean; override;
        // custom dataset virtual methods
        procedure InternalAfterOpen; override;
    public
        function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    published
        property Directory: string read FDirectory write SetDirectory;
    end;
```

GetCanModify函数是TDataSet的另一个虚拟方法，用于确定数据集是否是只读的。这种情况下，它返回False。我们也不必编写任何代码用于SetFieldData程序，但我们不得不定义它，因为它是抽象虚拟方法。

因为正在处理对象列表，故单元中将包含用于那些对象的类。在此例中，用TFileData类构造器的TSearchRec缓冲器来抽取文件数据：

```

type
  TFileData = class
    public
      ShortFileName: string;
      Time: TDateTime;
      Size: Integer;
      Attr: Integer;
      constructor Create (var FileInfo: TSearchRec);
    end;

constructor TFileData.Create (var FileInfo: TSearchRec);
begin
  ShortFileName := FileInfo.Name;
  Time := FileDateToDateTime (FileInfo.Time);
  Size := FileInfo.Size;
  Attr := FileInfo.Attr;
end;

```

当打开数据集时，这个构造器可被每个文件夹调用：

```

procedure TMdDirDataset.InternalAfterOpen;
var
  Attr: Integer;
  FileInfo: TSearchRec;
  FileData: TFileData;
begin
  // scan all files
  Attr := faAnyFile;
  FList.Clear;
  if SysUtils.FindFirst(fDirectory, Attr, FileInfo) = 0 then
    repeat
      FileData := TFileData.Create (FileInfo);
      FList.Add (FileData);
    until SysUtils.FindNext(FileInfo) <> 0;
    SysUtils.FindClose(FileInfo);
  end;

```

下一步是定义数据集的区域，在此例中，这些区域是固定的并且取决于有效的路径数据：

```

procedure TMdDirDataset.InternalInitFieldDefs;
begin
  if fDirectory = '' then
    raise EMDDataSetError.Create ('Missing directory');
  // field definitions

```

```

FieldDefs.Clear;
FieldDefs.Add ('FileName', ftString, 40, True);
FieldDefs.Add ('TimeStamp', ftDateTime);
FieldDefs.Add ('Size', ftInteger);
FieldDefs.Add ('Attributes', ftString, 3);
FieldDefs.Add ('Folder', ftBoolean);
end;

```

最后，该组件从当前记录缓冲器（ActiveBuffer值）引用的列表对象中将数据移动到数据集的每个字段，像GetFieldData方法请求的那样。该函数使用Move或StrCopy，这取决于数据类型，并且为抽取自相关标记的属性代码做些转换（H用于隐藏，R用于只读，S用于系统），这些属性代码也用于决定文件是否是实际的文件夹。下面是其代码：

```

function TMDDirDataset.GetFieldData (Field: TField; Buffer: Pointer): Boolean;
var
  FileData: TFileData;
  Bool1: WordBool;
  strAttr: string;
  t: TDateTimeRec;
begin
  FileData := fList [Integer(ActiveBuffer^)] as TFileData;
  case Field.Index of
    0: // filename
      StrCopy (Buffer, pchar(FileData.ShortFileName));
    1: // timestamp
      begin
        t := DateTimeToNative (ftdatetime, FileData.Time);
        Move (t, Buffer^, sizeof (TDateTime));
      end;
    2: // size
      Move (FileData.Size, Buffer^, sizeof (Integer));
    3: // attributes
      begin
        strAttr := ' ';
        if (FileData.Attr and SysUtils.faReadOnly) > 0 then
          strAttr [1] := 'R';
        if (FileData.Attr and SysUtils.faSysFile) > 0 then
          strAttr [2] := 'S';
        if (FileData.Attr and SysUtils.faHidden) > 0 then
          strAttr [3] := 'H';
        StrCopy (Buffer, pchar(strAttr));
      end;
    4: // folder
      begin
        Bool1 := FileData.Attr and SysUtils.faDirectory > 0;
        Move (Bool1, Buffer^, sizeof (WordBool));
      end;
  end;
end;

```

```

end;
end; // case
Result := True;
end;

```

编写该代码需注意的部分是算出保存在日期/时间字段内的日期的内部格式。这不是普通的Delphi使用的TDateTime格式，甚至不是内部的TTimeStamp，但被内部称为本地日期和时间格式。通过克隆VCL代码中的用于日期/时间字段的函数，我们编写了一个转换函数：

```

function DateTimeToNative(DataType: TFieldType; Data: TDateTime): TDateTimeRec;
var
    TimeStamp: TTimeStamp;
begin
    TimeStamp := DateTimeToTimeStamp(Data);
    case DataType of
        ftDate: Result.Date := TimeStamp.Date;
        ftTime: Result.Time := TimeStamp.Time;
    else
        Result.DateTime := TimeStampToMsecs(TimeStamp);
    end;
end;

```

使用有效的数据集建立演示程序（如图17.7所示），仅仅是连接DBGrid组件并添加文件夹选择组件，即ShellTreeView控件。该控件被设置成只在文件上工作，即设置它的Root属性为C:\。当用户选择一个新文件夹时，ShellTreeView控件的OnChange事件处理器将会更新数据集：

```

procedure TForm1.ShellTreeView1Change(Sender: TObject; Node: TTreeNode);
begin
    MDirDataset1.Close;
    MDirDataset1.Directory := ShellTreeView1.Path + '\*. *';
    MDirDataset1.Open;
end;

```

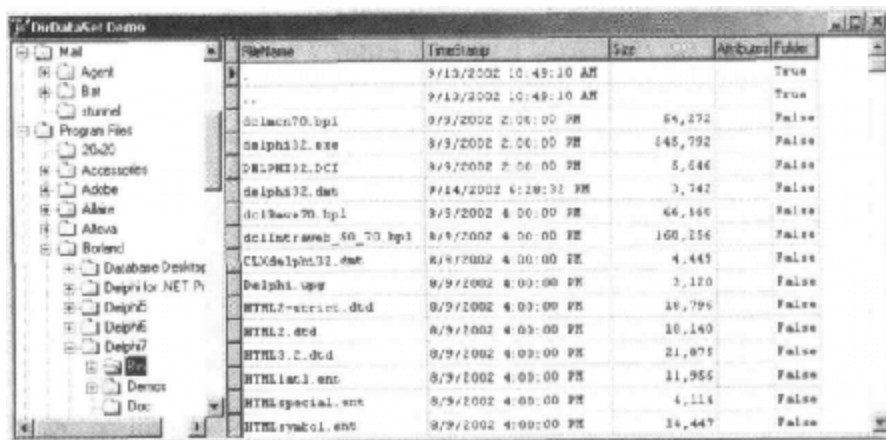


图17.7 DirDemo范例的输出，它使用了独特的数据集显示目录数据

警告：如果各位的Windows版本与Delphi中可以获得的shell范例控件冲突，则可以使用该范例的DirDemoNoShell版本，该版本使用旧风格的Windows 3.1——与Delphi文件的控件兼容。

对象的数据集合

正如大家在前面例子中看到的那样，对象列表从概念上来说和数据集合中的表格行是很相似的。在Delphi中，可以建立一个数据集合来包装一个对象列表，就像TFileData类那样。最有吸引力的地方就是能够扩展这个例子，以建立一个支持普通对象的数据集合，这一点我们要感谢Delphi中扩展的RTTI。

这个数据集合组件就像以前的例子那样，是由TMdListDataSet继承而来的。我们必须提供一个单一的设置：保存在ObjClass属性中的目标类（参见程序清单17.5中的TMdObjDataSet类的完整定义）。

程序清单17.5 TMdObjDataSet类的完整定义

```

type
  TMdObjDataSet = class(TMdListDataSet)
  private
    PropList: PPropList;
    nProps: Integer;
    FObjClass: TPersistentClass;
    ObjClone: TPersistent;
    FChangeToClone: Boolean;
    procedure SetObjClass (const Value: TPersistentClass);
    function GetObjects (I: Integer): TPersistent;
    procedure SetChangeToClone (const Value: Boolean);
  protected
    procedure InternalInitFieldDefs; override;
    procedure InternalClose; override;
    procedure InternalInsert; override;
    procedure InternalPost; override;
    procedure InternalCancel; override;
    procedure InternalEdit; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    function GetCanModify: Boolean; override;
    procedure InternalPreOpen; override;
  public
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    property Objects [I: Integer]: TPersistent read GetObjects;
    function Add: TPersistent;
  published
    property ObjClass: TPersistentClass read FObjClass write SetObjClass;
    property ChangesToClone: Boolean read FChangeToClone
      write SetChangeToClone default False;
  end;

```

该类被InternalInitFieldDefs方法使用，用以基于目标类的公共属性来确定数据集合的字段，该属性可以通过RTTI提取：

```

procedure TMDObjDataSet.InternalInitFieldDefs;
var
    i: Integer;
begin
    if FObjClass = nil then
        raise Exception.Create ('TMDObjDataSet: Unassigned class');
    // field definitions
    FieldDefs.Clear;
    nProps := GetTypeData(fObjClass.ClassInfo).PropCount;
    GetMem(PropList, nProps * SizeOf(Pointer));
    GetPropInfos (fObjClass.ClassInfo, PropList);

    for i := 0 to nProps - 1 do
        case PropList [i].PropType^.Kind of
            tkInteger, tkEnumeration, tkSet:
                FieldDefs.Add (PropList [i].Name, ftInteger, 0);
            tkChar: FieldDefs.Add (PropList [i].Name, ftFixedChar, 0);
            tkFloat: FieldDefs.Add (PropList [i].Name, ftFloat, 0);
            tkString, tkLString:
                FieldDefs.Add (PropList [i].Name, ftString, 50); // TODO: fix size
            tkWString: FieldDefs.Add (PropList [i].Name, ftWideString, 50);
                // TODO: fix size
        end;
    end;

```

相似的基于RTTI的代码用在GetFieldData和SetFieldData方法中，用来在数据集合字段的访问操作被请求时访问当前对象的属性。使用属性访问数据集合数据的最大好处就是，读写操作可以使用对应的方法直接映射到数据。这样，可以通过执行属性的读写方法中的规则来编写应用程序的业务规则——一种更正确的OOP方法，比将代码关联到字段对象并使其有效更好。

这里是GetFieldData的一个相对简单的版本（其他的方法也是相对称的）：

```

function TObjDataSet.GetFieldData (
    Field: TField; Buffer: Pointer): Boolean;
var
    Obj: TPersistent;
    TypeInfo: PTypeInfo;
    IntValue: Integer;
    FlValue: Double;
begin
    if FList.Count = 0 then
        begin
            Result := False;
            exit;
        end;

```

```

end;
Obj := fList [Integer(ActiveBuffer^)] as TPersistent;
TypeInfo := PropList [Field.FieldNo-1]^ .PropType^;
case TypeInfo.Kind of
  tkInteger, tkChar, tkWChar, tkClass, tkEnumeration, tkSet:
    begin
      IntValue := GetOrdProp(Obj, PropList [Field.FieldNo-1]);
      Move (IntValue, Buffer^, sizeof (Integer));
    end;
  tkFloat:
    begin
      FlValue := GetFloatProp(Obj, PropList [Field.FieldNo-1]);
      Move (FlValue, Buffer^, sizeof (Double));
    end;
  tkString, tkLString, tkWString:
    StrCopy (Buffer, pchar(GetStrProp(Obj, PropList [Field.FieldNo-1])));
end;
Result := True;
end;

```

这种基于指针的代码看起来可能非常糟，但是如果大家已经经历了关于开发一个定制数据集合的技术细节的讨论，那么这种代码并不会给各位增加什么复杂性。它使用了一些定义在TypeInfo单元的数据结构（并有简要的注释），如果读者对以前的代码有任何疑问的话，可以很好地参考一下它。

使用这种直接编辑对象数据的方法，读者可能想知道，如果一个用户取消了编辑操作会发生什么情况（Delphi通常会说明一些事情）。我的数据集合可以提供两种手段，通过ChangesToClone属性和基于克隆的概念来复制公共的属性。核心的DoClone程序使用RTTI代码——该代码与大家已经看到的相似——将一个对象的所有公共数据复制到另一个对象中，创建一个有效的复制（或克隆）。

这种克隆发生在两种情况下：根据ChangesToClone属性值的不同，或者是在克隆对象上执行编辑操作，接着在Post操作中将其复制到实际的对象中；或者是在实际的对象上执行编辑操作，并且当编辑操作被一个Cancel请求终止时，由克隆对象取回初始值。这里是涉及的三种方法的代码：

```

procedure TObjDataSet.InternalEdit;
begin
  DoClone (fList [FCurrentRecord] as TDbPers, ObjClone);
end;

procedure TObjDataSet.InternalPost;
begin
  if FChangeToClone and Assigned (ObjClone) then
    DoClone (ObjClone, TDbPers (fList [FCurrentRecord]));
end;

procedure TObjDataSet.InternalCancel;

```

```

begin
  if not FChangeToClone and Assigned (ObjClone) then
    DoClone (ObjClone, TPersistent(fList [fCurrentRecord]));
  end;

```

在SetFieldData方法中，必须要修改克隆的对象或者原始的对象。如果要想做得更复杂些，则必须考虑GetFieldData方法中的这种不同之处：如果正从当前的对象中读取字段，可能不得不使用它的修改后的克隆（否则，用户对其他字段的修改将会消失）。

正如大家在程序清单17.5中所见到的，该类还包含一个Objects数组——以一种OOP方式访问数据，以及一个与集合的Add方法相似的Add方法。通过调用Add，代码将创建目标类的一个新的空对象，并将其添加到内部列表中：

```

function TMDObjDataSet.Add: TPersistent;
begin
  if not Active then
    Open;
  Result := fObjClass.Create;
  fList.Add (Result);
end;

```

为了说明这种组件的使用，我编写了ObjDataSetDemo范例。就像读者可以在图17.8中看到的那样，其中包含一个演示目标类，带有一些字段和按钮，可以自动地创建对象。该程序最有意思的特性就是有些东西必须大家亲自来试一下。运行这个程序并查看DbGrid栏。接着编辑这个目标类，TDemo，添加一个新的公开的属性。再次运行这个程序，这时网格将包含一个用于该属性的新列。

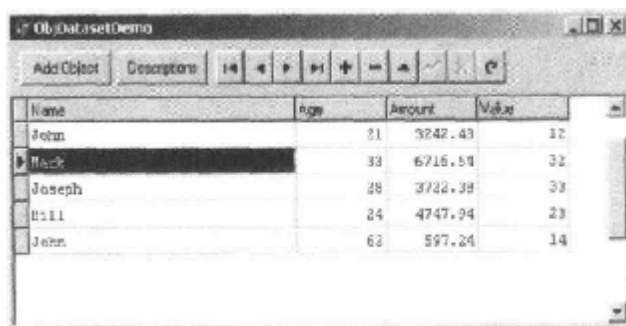


图17.8 ObjDataSet Demo范例显示出使用RTTI将数据集映射到对象的长处

小结

本章我们研究了Delphi的数据库结构：首先谈到数据敏感控件的开发，然后通过研究TDataSet类的本质，编写了两个定制数据集组件。运用该信息以及出现在该部分的其他方法，读者便能依据需要选择数据库应用程序的结构。

数据库编程是Delphi的核心要素，这就是为什么本书专门有几章都讨论该主题的原因。下一章将继续讨论这个问题，介绍在Delphi 7中可以用到的新的报告引擎。第20章和第21章讨论如何在Web上表示数据时，我们还将回到数据库的问题上；同时，在第22章和第23章中讨论XML和SOAP时，我们也将再次涉及到数据库问题。

第18章 使用Rave做报表

数据库应用程序实现了查看和编辑数据，但是其输出数据通常需要物理地打印到纸上。从技术角度出发，Delphi支持多种不同方式的输出：从直接的文本输出到使用打印机Canvas，从一直使用的数据库报表到以多种格式产生文档（从微软的Word到Sun的OpenOffice）。本章专门介绍报表，尤其是使用包含在Delphi 7中的第三方报表引擎Rave。如果读者对Delphi中驱动打印的其他技术感兴趣，请查看笔者网站的相关内容（在附录C中讨论）。

报表工具极为重要，因为它们可以独自完成复杂的进程：报表子系统可以成为一个独立的应用程序。尽管本章主要关注如何使用Delphi程序从数据库中产生一个报表，但当评测该工具时，始终要记住报表的自治性。

必须小心创建报表，因为它代表应用程序的用户接口，有时候比软件本身更加重要。比起使用程序产生报表的用户，更多的人可能更关注打印报表。基于此原因，拥有良好质量的报表和实现用户定制的灵活结构非常重要。

说明：本章在Jim Gunkel的帮助下完成，他来自于Nevrona Designs，正是该公司开发了Rave引擎。

本章主要包括以下内容：

- 介绍Rave（报表创建的可视环境）
- Rave Delphi组件
- Rave设计器组件
- 从数据库到报表
- Rave的高级特性

Rave介绍

检索由应用程序管理的数据信息时，报表是一个首要的方法。为了提供既有含义性又有信息性数据的可视报表，各种传统的可视报表应用程序应运而生，它们提供了适合表格型数据列表的联合规划工具。但是今天，报表的要求也更多、更复杂，无法轻易通过联合规划工具进行处理。

Rave Reports是一个可视的设计环境，提供很多惟一的特性，使报表进程更简单、更迅速、更有效。Rave可以处理多种报表格式，并且包括高级的技术，例如镜像，以鼓励对于报表内容的重新使用，从而加快改动速度，简化维护手段。

本章简要介绍Rave的特性。更多的信息可以从在线帮助文件中获得，该文件可以以下列方式获得：Delphi光碟中的PDF文档，一些演示项目，以及生产商的网站www.nevrona.com。

说明：Rave的一个关键特征，也是Borland看好该解决方案的原因，在于它是一个完全的跨平台解决方案，可以在Windows和Linux两者下使用。不仅仅因为Rave组件集成了VCL和CLX，而且由于Rave设计器本身是一个以CLX编写的跨平台应用程序。

Rave: 报表创建的可视环境

启动Rave可视报表设计环境, 只需双击窗体上的TRvProject组件, 或者选择Delphi IDE中的Tools Rave设计器。激活该环境后, 可以见到如图18.1所示的一个窗口。在这里, Rave设计器包括很多部分: 页面设计器、事件编辑器、属性面板、项目树面板、工具栏、工具栏模板和状态栏。

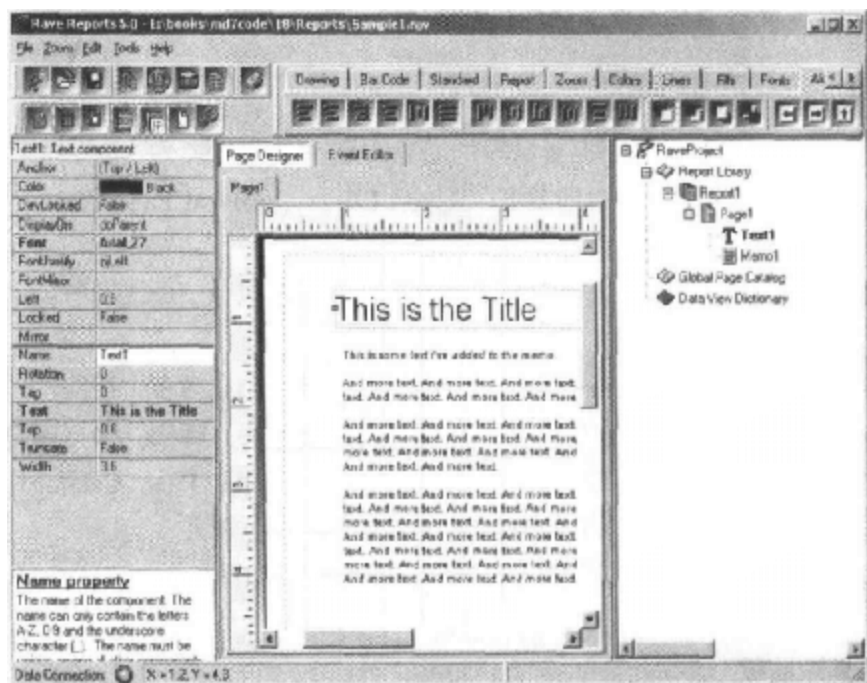


图18.1 含有简单报表的Rave设计器

说明: 无论何时, 若想要检查努力的结果, 只需按Rave设计器中的F9键来预览当前报表。

记住Rave允许最终用户创建或者修改各自的报表。可以通过选择Edit►Preferences对话框 (在Environment部分), 将Rave设计器的级别设置为Beginner、Intermediate或Advanced, 这样最终用户可以在其适合的级别下工作并且不会有更多的权限。还可以锁定报表特性, 使最终用户无法进行修改。

页面设计器与事件编辑器

Rave设计器窗口的中心部分是页面设计器 (放置报表的地方) 和事件编辑器 (运行时提供脚本和定制报表的地方)。

页面设计器是Rave中最值得注意的方面之一。该页是报表的基础, 用来完成所有的设计。它以栅格的形式显示, 但是可以通过参数设置来更改其外观。当前设计页的名称出现在页面设计器上的选项卡中 (图中的Page1)。

事件编辑器允许定制报表组件的脚本代码。每个组件有不同的事件类型用于计算、字符串操作或者定制报表逻辑。事件编辑器是Rave的高级特性, 在本章结尾处会简要介绍。

属性面板

属性面板在Rave设计器的左侧，用于帮助定制组件显示和操作的方式。它的作用和Delphi中的对象检验器相似：当页面上有一个组件被选中后，属性面板便通过显示和该组件相关的属性来反映用户的选择。如果没有选择任何组件，则属性面板为空。

和Delphi IDE中相同的还有，可以通过编辑编辑框的内容、从下拉菜单中选择一个选项、或者打开一个编辑对话框来更改属性值。可以双击任何具备一系列选择的属性（取代单击向下箭头然后选择选项）来进一步操作列表中的下一个项目。

项目树面板

位于设计器右侧的项目树面板可以提供很多信息。它也提供了一个导航报表项目结构的简单方法。项目树包括了三个主要节点：Report Library、Global Page Catalog和Data View Dictionary。

Report Library 包括了项目内的所有报表。每个报表包括一页或者多页。一般而言，每页包括一个或多个组件。

Global Page Catalog 管理报表模板。报表模板包括一个或者多个组件，可以通过Rave的镜像技术重复使用。其中包括的一些项目，诸如信头、脚注、预打印格式、水印设计或者整页定义等，均是其他报表的基础。可以将Global Page Catalog作为一个仓库——一个放置报表项目的中心位置，能够从多个报表中进入。

Data View Dictionary 定义了所有的数据外观和其他报表的数据相关对象。

工具栏和工具栏模板

在Delphi中，Rave包括两种类型的工具栏：组件工具栏和IDE工具栏。但是，不同于Delphi（只为组件使用带标志的组件面板），在Rave中可以任意将两种类型的工具栏放置在设计器顶端的空闲区域或者带标志的工具栏模板下。开始时该过程可能混乱，但是安排工具栏适应要求之后（使用Dock和Undock快捷菜单命令，而不是随意拉动），可以发现它们很灵活。

Rave中的默认组件工具栏是Standard、Drawing、Report和Bar Code。本章稍后，笔者会详细描绘组件工具栏。目前，如果读者已经安装了附加工具包，则还会有其他组件工具栏出现在Rave设计器中，那些组件可以在工具栏中首先安排。

编辑器工具栏可以改变或者修正方案或存在的组件。下面总结一下各种命令：

项目工具栏 和组件工具栏相似，它也允许在报表方案中创建新的报表、页和数据对象组件。项目工具栏还可以用来创建新的报表方案，保存和装载存在的报表方案，以及打印或者预览当前报表。

排列工具栏 每页有很多工具用来排列和定位组件。第一个选择的组件经常决定着排列操作的基准。可以使用顺序按钮来改变组件的打印顺序：前进，后退，提前，放后。Tap按钮可以用来微调组件。

字体工具栏 可以用来改变字体属性，例如名字、大小、类型和队列。

填充工具栏 提供填充类型的形状，例如长方形和圆形。

线条工具栏 用来改变线条和边界的宽度以及类型。

颜色工具栏 用来决定首要和次要的颜色（一般分别对应前景和背景）。鼠标左键用来选择首要颜色，鼠标右键用来选择次要颜色。有八个常用颜色框可满足日常使用的颜色。双击工具栏右边的首要或者次要颜色框来打开颜色编辑对话框，可以提供附加的颜色选择工具。

缩放工具栏 提供很多工具来进行页面设计器的缩放，方便编辑。

设计工具栏 允许通过属性对话框来定制页面设计器和Rave设计器。

状态栏

Rave设计器的底部是状态栏。状态栏可以提供的信息包括直接的数据查看连接状态和鼠标的位置以及大小。数据连接LED的颜色可以提供Rave数据系统的状态信息（DirectDate-Views）：灰色和绿色分别指示非激活和激活的连接；黄色和红色指示特定的数据访问情况（分别是等待答复和超时）。

X和Y的值指示鼠标指示器在页面上的位置。当下拉一个组件到页面上时，如果不放开鼠标按钮，那么组件的大小会通过dX和dY值表现出来（d代表delta）。

使用RvProject组件

图18.1显示了一个小的Rave报表，存在于一个.rav文件中。使用Rave页可将这个报表连接到Delphi 7应用程序，该页提供了一系列组件，中心组件即是RvProject。放置此组件到窗体或者数据模板上，将其ProjectFile属性设置为一个Rave文件，然后为一个按钮编写下面这个事件管理代码：

```
RvProject1.Execute;
```

现在有一个工作中的应用程序（在源代码中的RavePrint例子），它可以打印报表并且包括内置的打印预览特性，如图18.2所示。引用的文件可以单独发布，并且不用更改Delphi程序就可以修正。另一种办法是，把一个.rav文件装载到DFM文件，将其内置入Delphi执行文件。要实现这些，需要使用Rave方案组件的StoreRAV属性。

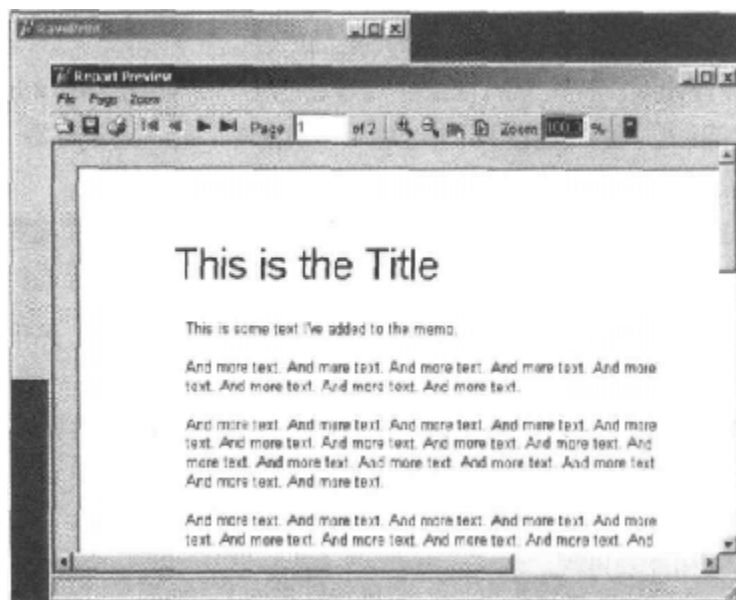


图18.2 一个报表的Rave Report预览窗口

说明：正如笔者早些时候提到的，RvProject组件和其他Rave Reports的组件可以在VCL或者CLX的应用程序中使用。整个Rave引擎是全部跨平台的。

为了控制报表的最重要设置以及预览，可以将RvNDRWriter组件或者RvSystem组件与RvProject组件的Engine属性相关联：

RvNDRWriter组件 执行报表的时候会生成一个NDR格式的文件。NDR格式的文件是专有的二进制格式，保存了绘制组件的所有信息，用来以各种格式重新产生报表。

RvSystem组件 将RvNDRWriter组件与一个标准的打印、预览和绘制用户接口组合起来。有许多属性可以用来定制用户接口，而且可以定义覆盖事件，通过定制版本来转换标准的对话框。

绘制格式

Rave引擎通过RvNDRWriter来产生一个NDR文件或者流。Rave的绘制引擎可以将这个内部表示法转换成多种格式。为了让用户选择一个目标文件格式，需拉下Delphi程序窗体中的需要绘制的组件。当运行RvProject组件的Execute方法时（如RavePrint例中所示），可以通过Rave的对话框选择一个文件格式，如图18.3所示。

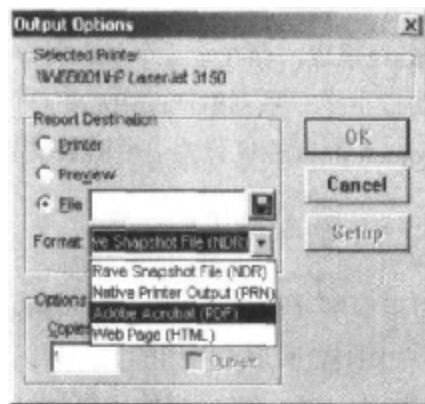


图18.3 执行完Rave方案后，用户可以选择输出格式或者绘制引擎

以下是Delphi组件面板中Rave页的可用绘制组件：

RvRenderPreview 将一个NDR文件或者流显示到屏幕上。ScrollBar组件用来显示报表页，还有很多方法和属性可用来创建用户预览对话框。除非需要一个用户打印预览，否则可以使用RvSystem组件来代替RvRenderPreview进行预览。

RvRenderPrinter 发送一个NDR文件或者流到打印机。除非需要一个用户打印或者预览界面，否则不需要使用这个组件——RvSystem提供了标准的打印功能。

RvRenderPDF 将一个NDR文件或者流转换为PDF（Adobe Acrobat）格式。可以使用属性和事件来定制输入类型，从而创建和提供压缩支持。

RvRenderHTML 将一个NDR文件或者流转换为HTML（DHTML）格式。每页绘制为一个独立的页面，而且可以和模板合并提供更大的输出控制。

RvRenderRTF 将一个NDR文件或者流转换为一个RTF格式。RTF格式使用字段来实现准确定位和报表再现。

RvRenderText 将一个NDR文件或者流转换为文本格式。很多图形命令和组件，例如线条和长方形，会在文本绘制时被忽略。

除了允许用户通过对话框来选择格式外，绘制还可以通过程序实现。例如，为了将一个报表直接转换为一个PDF文件，可以编写下列代码（再次使用RavePrint例子）：

```
procedure TFormRave.btnPdfClick(Sender: TObject);
begin
  RvSystem1.DefaultDest := rdFile;
  RvSystem1.DoNativeOutput := False;
  RvSystem1.RenderObject := RvRenderPDF1;
  RvSystem1.OutputFileName := 'Simple2.pdf';
  RvSystem1.SystemSetups := RvSystem1.SystemSetups - [ssAllowSetup];
  RvProject1.Engine := RvSystem1;
  RvProject1.Execute;
end;
```

数据连接

数据连接组件负责在Delphi应用程序中的数据和Rave设计器中的DirectDataViews之间提供连接。要注意的是，每个数据连接组件定义的Name属性值用来提供和Rave报表的连接。基于这个原因，应注意防止在Rave中创建DirectDataViews后改变组件名。

下面是一些数据连接组件：

RvCustomConnection 使用程序化事件来提供到Rave报表的数据，可以用来发送非数据库数据到一个可视报表。

RvDataSetConnection 将任何TDataSet级的派生组件连接到一个Rave DirectDataView上。使用FieldAliasList属性也可以修改数据集的字段名，将其替换为对开发者和创建报表的最终用户来说更为清晰的字段名。如果需要在Rave报表中对查找项或主/详关系进行分类或过滤，可以操作OnSetSort和OnSetFilter事件。

RvTableConnection和RvQueryConnection 用于将BDE表格或者Query组件连接到一个Rave DirectDataView。Rave天生具备为表格连接提供分类和过滤的功能。

笔者创建了一个RaveSingle例子，用于演示如何构建一个数据库相关的报表，该示例使用Rave方案和一个连接对DbxSingle程序（创建于第14章“使用dbExpress的客户机/服务器编程”）进行了更新：

```
object RvDataSetConnection1: TRvDataSetConnection
  RuntimeVisibility = rtDeveloper
  DataSet = SimpleDataSet1
end
object RvProject1: TRvProject
  ProjectFile = 'RaveSingle.rav'
end
```

在Rave设计器中，笔者还创建了一个新项目，位于RaveSingle文件夹的RaveSingle.rav文件中。为了引用Delphi程序释放的数据，需要增加一个数据视图——单击Project工具栏的New

在Report Library节点卜除了一系列可用报表外,如早些时候介绍的,Rave方案还有一个Global Page Catalog和一个Data View Dictionary。Data View Dictionary是数据通道的详细列表,由集合的Delphi应用程序(使用早些时候介绍的Delphi Rave连接组件)产生,或者从报表到数据库直接激活。

设计报表时,可直接将可视组件放入页面或者另一个容器中,如一个Band或者一个Region。这些组件中的一些不是和数据库数据连接的——例如,设计器标准工具栏中的Text组件、Memo组件和Bitmap组件。其他组件可以通过一个数据库表格的字段连接,例如Report工具栏的DataText和DataMemo组件。

基本组件

标准工具栏中包括七个组件:Text、Memo、Section、Bitmap、Metafile、FontMaster和PageNumInit。很多标准组件在设计报表时频繁使用。

Text和Memo组件

Text组件用来在报表上显示单行文本。它的作用正如可以包含简单文本(不是数据)的标签。放置到报表上时,Text框的周围有一个框来指示它的边界。

Memo组件类似于Text组件,但是包含多行文本。当设置了备注字体后,Memo组件中的所有文本都以相同的字体表示,正如Delphi Memo组件一样。一旦输入文档,不仅Memo框可以自动调节大小,Memo框中的文本也会相应地调整位置。如果输入到Memo Editor中的文本看起来像丢失了一样的话,那么应调整框的大小,使得所有的文本均可见。

Section组件

Section组件用于分组建件,正如Delphi中的Panel一样。它提供了很多有用功能,例如通过单击鼠标来移动组成Section的组件,而不用单独移动每个组件或者在移动前选择所有的组件。

在操作Section组件时,Project Tree比较有用。从一个扩展节点很容易看到哪些组件在各个Section,因为混合的组件可以构成一个有父子关系的树。

提示:镜像时Section组件非常重要,它允许在报表设计中继承可视设计。

绘图组件

Bitmap组件和Metafile组件用来在一个报表上放置图像。Bitmap组件支持带.bmp后缀名的栅格图像文件,Metafile组件支持带.wmf和.emf后缀名的向量图像文件。当在CLX应用程序中使用Rave时,不支持Metafile组件,因为它们是基于特定Windows技术的。

FontMaster组件

一个报表中的每个Text组件都有一个Font属性。通过设置这个属性,可以给组件分配一个特定的字体。在很多情况下,需要并且有必要为一个以上的对象设置一样的字体属性。虽然可以通过同时选择一个以上的组件来实现,但是这个方法有一个缺点:必须跟踪哪种字体应该是同样字形、大小和样式的,对于多个报表而言,这不是一件容易的任务。

FontMaster组件可以允许为报表的不同部分定义标准的字体，例如标头、内容和页脚。FontMaster是一个非可视组件（通过按钮的绿色指定），所以页面上没有可视引用（不同于Delphi）；和其他非可视组件一样，它可以通过Project Tree来访问。

一旦设置了FontMaster组件的属性，就很容易将它链接到文档。只需在报表上选择一个Text/Memo组件，然后使用属性面板中FontMirror属性的下拉键来选择一个FontMaster链接。FontMirror属性设置为FontMaster的所有组件都会被FontMaster的Font属性影响并随之改变。

设置了一个组件的FontMirror属性后，该组件的Font属性就会被FontMaster的Font属性覆盖。使用FontMaster的另一个副作用是，当组件的FontMirror属性设置之后，Font字体栏就被禁止了。

每页可能有多个FontMaster，但是最好可以将FontMaster组件更名来描述其功能。还可以在一个全局页面上定位FontMaster，这样它们就可以被方案中的所有报表使用，并且提供更加一致的排版规划。

页面计数

PageNumInit是一个非可视组件，用于在一个报表中重新开始页面计数。它的用法类似于其他的非可视组件。而且PageNumInit常用于需要更加高级的格式时。

例如，考虑一个检查账目的客户陈述报表。客户每个月收到的账目陈述的页数可能不同。假设第一页定义了账目总结页，第二页定义了客户的信用/存款，第三页定义了提款和贷款。前两份报表可能只需要一页，但是如果客户的账目活动比较频繁，那么提款部分有可能有几页长。如果提供报表的用户希望每个部分页面是独立计算的，那么总结和存款页都要标志“1 of 1”。如果一个活跃的客户账目有三页内容关于提款和贷款，那么陈述的这个部分要标为“1 of 3”、“2 of 3”、“3 of 3”。对于这类页面计数，使用PageNumInit就非常方便。

制图组件

就像标准组件一样，制图组件和数据无关。Rave报表可以包括三种线条组件：常规线可以向任何方向画，包括组成夹角；水平和垂直线有固定的方向。可用的几何图形包括正方形、长方形、圆形和椭圆形。可以在报表上拉下一个图形然后将其移动到其他元素后。例如，在一个DataBand的周围放一个长方形，设置它的大小完全填满带条，然后把它移动到带条里其他组件的后面。

条码组件

条码组件用来在一个报表中创建许多不同的条形码。条码主要用于让用户确切知道自己所需的是什么，必须有关于条码的背景知识和使用方法。要定义一个条码的值，需在属性面板的Text属性框输入值。

Rave条码支持包括下列内容：

- POSTNET（邮政数字编码技术）条码，特别用于美国邮政服务。
- I2of5BarCode（以5间隔2），只用于数字信息。
- Code39BarCode，一个包含数字和字母的条码，可以编码十进制数字、大写字母表和

一些特殊符号。

- **Code128BarCode**，一个高密度包含数字和字母的条码，用于编码所有128个ASCII字符。
- **UPCBarCode**（通用的产品代码），有12位的固定长度，设计用于编码产品。
- **EANBarCode**（欧洲文章编号系统），和UPC一致，但是有13位，包括10个数字字符，两个国家代码字符和一个校验位。

数据访问对象

大多数情况下，一个报表要基于直接来自于数据库或者来自于Delphi应用程序的数据。而数据可能来自连接到数据库的数据集或者Delphi程序的内部进程。选择了Project工具栏的New Data Object按钮后，可以看到这里显示的和下面描述的选项。



RaveDatabase 组件（Database Connection） 为DriverDataView组件提供数据库连接参数。只允许安装了DataLink驱动程序的数据库连接。

RaveDirectDataView组件（Direct Data View） 提供一种方式来恢复位于主Delphi应用程序的数据连接组件中的数据，如上例所示（这个选择称为Direct Data View，尽管不是来自报表的直接数据库连接，但替代的是基于来自数据库数据的间接连接）。

RaveDriverDataView组件（Driver Data View） 通过查询语言如SQL，来提供一种方式，用于定义特定数据库连接查询。而且会显示一个查询构造器来定义查询。

RaveSimpleSecurity组件（Simple Security Controller） 通过使用UserList属性中的一个简单的用户名和密码组合来实现最基本的安全性。UserList每行包含一个用户名和密码组，格式如下：Username = password。CaseMatters是一个布尔值属性，可以控制密码是否大小写敏感。

RaveLookupSecurity组件（Data Lookup Security） 允许用户名和密码在数据库表格中成对地而不是单个地接受检查。DataView属性指定数据视图来检查用户名和密码。

UserField和PasswordField属性用来检查用于校验的用户名和密码。

Region与Band

Region属性包含Band属性。在最简单的方式下，Region可能是整个页面组件。报表是列表类型时会出现这种情况。很多主-详报表用来适应单一的Region设计。但是，不要局限于认为Region是整个页面；Region的属性涉及它的大小和在页面上的位置。设计复杂报表时，

对Region的创造性使用可以提供更多的灵活性。多个Region可以放置到单个页面上；它们的排列顺序可以是并排的、重叠的，或者是与页面交错的。

提示：不要混淆Region和Section。Region组件包含也只包含Band。Section组件可以包含很多组组件，也包括Region组件，但是不能直接包含Bands。

使用Bands时，必须遵循一个简单的规律：**Bands**必须在**Region**中。要注意的是，一个页面上**Region**的数目没有限制，一个**Region**中**Band**的数目也一样。一旦将报表想像出来，就可以使用**Region**和**Band**联合体来解决很多进行报表设计时遇到的困难。

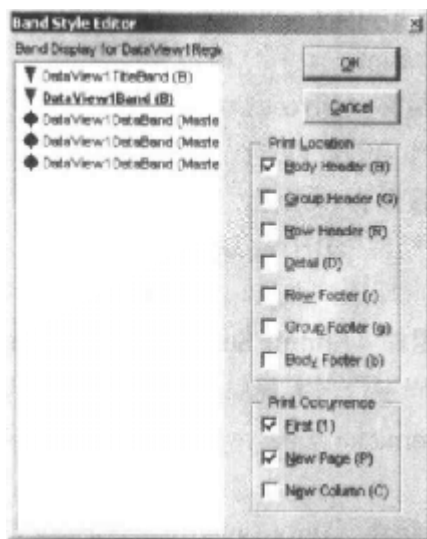
DataBand 用来显示来自**DataView**的迭代信息。一般情况下，一个**DataBand**会包含若干**DataText**组件。一个**DataBand**的**DataView**属性必须要设置为**DataView**组件，这样才能迭代并且代表性地包括其他操作于同一个**DataView**的数据敏感组件。

Band 用于打印一个**Region**内各带条的页头和页脚。支持页头和页脚的类型包括**Body**、**Group**和**Row**；可以通过**BandStyle**属性来选择它们。**Band**中不需要页头和页脚，因为可以在页面上**Region**组件之外的位置直接拉下它们。

ControllerBand是一个重要的属性，这个属性决定了**Band**属于（或者受控于）哪个**DataBand**。当控制**DataBand**设置之后，注意**Band**上的图形符号指向其控制**Band**的方向，而且符号的颜色相匹配。**Band**上的字母代码在下一节中介绍。

Band样式编辑器

进入一个**Band**或者**DataBand**组件的**BandStyle**属性，然后单击省略按钮来打开**Band**样式编辑器。



这个编辑器提供了一种简单的方法，用于使用检查框来选择**Band**想要的属性。注意，一个**Band**可能同时有若干个不同的特征是激活的。这表明一个**Band**可以同时为**Body Header**和**Body Footer**。

Band样式编辑器中的显示区用来表现伪布局样式的报表流程。**DataBands**会被复制三次来表现它们是重复的。当前编辑的**Band**会加亮显示，虽然可以在编辑器中看到其他的**Band**，但是只能修改当前编辑的那一个。

Band样式编辑器在显示区和Page Layout区使用符号和字母来告知每个Band的行为（例如，如图18.6所示）。两种表示方式的主要不同在于：Band样式编辑器显示按照每个Band的定义在伪流中安排Bands。在Band样式编辑器中，Bands按照逻辑流和在设计时放置在报表中的位置来安排。报表输出中Band的次序主要以这种顺序控制。

首先打印标头（大写字母BGR，分别代表Body、Group和Row），后跟DataBand，然后是每级的页脚（小写字母bgr）。但是，如果为某一特殊级别定义不止一个标头，则应按标头Band在Region中安排的顺序处理。所以，在一个主-详报表中，可以将所有级别的所有标头放在顶端，所有的DataBands放在中间，所有的页脚放在Region的底端。另外一种办法是分组各个级，每一级有恰当的标头、页脚和DataBand。Rave允许程序员按照非常直观的方式，使用Region布局来设计流程。要记住，同一级别Band的优先顺序要按照它们在Region中的顺序。

两个符号显示了不同band的父/子或者主/详关系：

- 三角形符号（上/下箭头）指示了band被同样颜色的主band控制，可以通过箭头方向发现。
- 菱形符号代表了一个主要的或者控制的band。

这些符号都是颜色代码并且通常用来表示主-详流程的级别。记住可以有主/详/详的关系，其中两个“详”都可以被同一个主控制，或者一个“详”被另外一个“详”控制。

数据敏感组件

可以将不同的数据敏感的Rave组件放置到一个DataBand。最常用的选择是DataText组件，用来显示来自一个数据集的文本字段，如RaveSingle范例所示。

有两个选项可用来在DataField属性中输入内容。第一个选项是使用下拉列表进行选择；这种方式适合一般的数据库报表，每个DataText项只需要一个单独的数据字段。第二个选项是使用数据文本编辑器。

数据文本编辑器

很多报表需要合并不同的字段。两个常见的例子是城市、州和邮政区号，或者姓和名的组合体。在代码中，可以使用下列语句来实现这些合并：

```
City + ', ' + State + ' ' + Zip  
FirstName & LastName
```

DataField属性有数据文本编辑器，如图18.5所示，可以帮助建立复合字段。单击省略号来打开数据文本编辑器；通过从列表框中选择项目，可以利用字段、参数或者变量来建立复杂的数据敏感文本字段。这个编辑器包括很多组合体；笔者将简单地介绍这些内容，稍后大家可以在练习中尝试。

注意对话框分为五组：数据字段，报表变量，项目参数，初始化变量和数据文本。数据文本是结果窗口；插入项目时要注意这个窗口。这个窗口的右侧有两个键分别是加号（+）和连接符号（&）。+按钮可以将两个项目合并到一起，中间没有空格；&按钮在合并项目时中间加一个空格（只要前面的字段不是空的）。所以，第一步需要决定使用+或者&，然后从数据文本窗口上的三个组中选择文本。

DataText组件不只限于打印数据库数据：还可以使用项目参数和报表变量。进入报表变量组，看一下列表框中的可用变量。项目参数列表可以包含被应用程序初始化的UserName、ReportTitle，或者UserOption参数。要创建项目参数列表，应选择项目树中的项目节点（顶端表项）。然后，在属性面板中单击参数属性旁边的省略号按钮，打开典型的字符串编辑器，然后输入从应用程序到Rave的参数（例如UserName）。

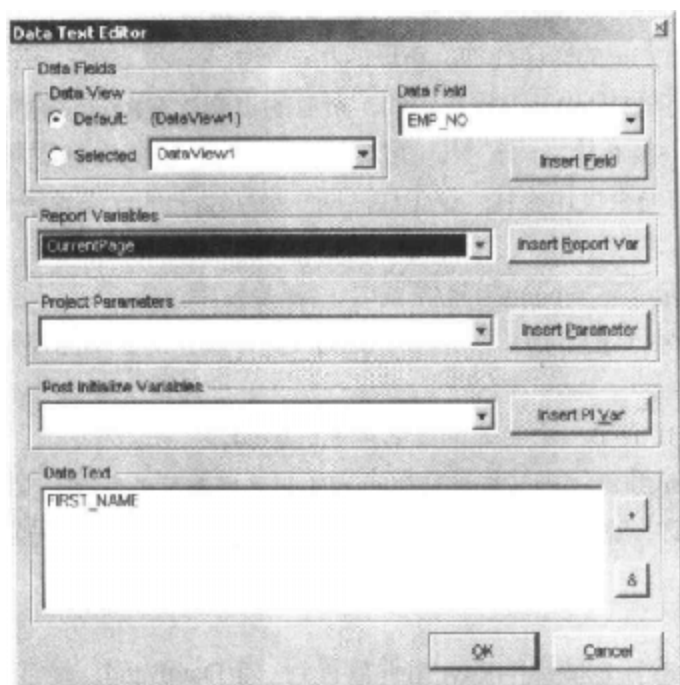


图18.5 Rave设计器的数据文本编辑器

从文本到备注

DataMemo组件显示了DataView的一个备注字段。DataMemo组件和DataText组件的主要不同在于：DataMemo组件用来打印多行文本，因此需要包装。例如，打印发票中每页底端的客户评论。

DataMemo组件的另外一个作用是邮件合并功能。完成这个功能的最简单方式是设置DataView和DataField属性为备注字段的源。然后，通过单击MailMergeItems属性旁的省略号按钮来启用邮件相关的编辑器。这个编辑器允许设置Memo中可能变化的条目。

要使用邮件合并编辑器，需单击Add按钮。在Search Token窗口中，输入Memo中的条目和被替换的内容。然后，在Replacement窗口中输入替换字符串或者单击Edit按钮来启用数据文本编辑器，在它的帮助下选择不同的DataView和字段。

计算总和

CalcText组件是一个数据敏感组件。DataText组件和CalcText组件的主要不同在于：CalcText组件专门设计用来完成计算和显示结果。

CalcType属性决定了要完成计算的类型：取值包括平均、计数、最大、最小和总和。例如，可以使用这个组件来打印一个发票每页顶端的总值。

CountBlanks属性决定在平均和计数方法中是否包括空字段值。如果RunningTotal值为True, 那么每次打印后计算值不会被复位为0。

在页面内轮转数据

DataCycle组件基本上是一个不可见的DataBand, 用来直接向一个页面增加数据迭代功能。这个组件适用于表单形式的报表, 但在使用Region和Band组件时会很麻烦。要确认DataCycle组件先于页面上任何其他组件(用于处理来自同一DataView信息)出现, 这一点很重要。

高级Rave

透过对Rave的长长介绍, 不难发现这个报表系统如此复杂, 可以使用一整本书来讨论。笔者已经创建了一些例子, 并且会继续显示主/详关系和复杂结构的其他报表。但是, 在目前向导和已有信息的帮助下, 读者也可以独立创建相似的例子。所以, 在这一节笔者只创建一个此类型的例子, 然后提供一些Rave的相关特征, 这些特征很难通过反复试验来理解。

说明: 一些不讨论的其他Rave特征包括, 将报表设计器分配给最终用户(允许他们定制报表), 或者与一个Delphi程序捆绑, 或者作为一个独立的工具。Rave还有一个服务器版本, 允许通过Web服务器对报表进行表面处理。

提示: 可以在Nevrona的网站上学习Rave的更多特征和其他方面。特别是在位于www.nevrona.com/rave/tips.shtml的网页上浏览这些提示集合。

主/详报表

如果要在Rave中创建一个主/详报表, 则相应的Delphi应用程序中应有两个数据集, 但是这些数据集不需要在程序中定义的主/详关系——报表可以定义这样一个关系。在RaveDetails演示程序中, 每个数据集合通过一个Rave连接来表现:

```
object dsDepartments: TSimpleDataSet
  Connection = SqlConnection1
  DataSet.CommandText = 'select * from DEPARTMENT'
end
object dsEmployee: TSimpleDataSet
  Connection = SqlConnection1
  DataSet.CommandText = 'select * from EMPLOYEE'
end
object RvConnectionDepartments: TRvDataSetConnection
  DataSet = dsDepartments
end
object RvConnectionEmployee: TRvDataSetConnection
  DataSet = dsEmployee
end
```

该报表有两个相应的数据视图, 每个都与DataBand组件连接(都在Region内)。二级DataBand使用一些属性定义主/详关系。MasterDataView属性对应主数据集的数据视图,

MasterKey和DetailKey属性对应定义连接的字段（都对应DEPT_NO字段）。ControllerBand属性对应DataBand显示主数据集的数据。

在主/详报表中，必须详细设置band，最重要的设置由Band样式编辑器管理。可以在图18.6中看到RaveDetails范例的编辑器。主数据视图的KeepRowTogether属性设置为True，是为了避免详细数据出现在非主数据的不同页面上。

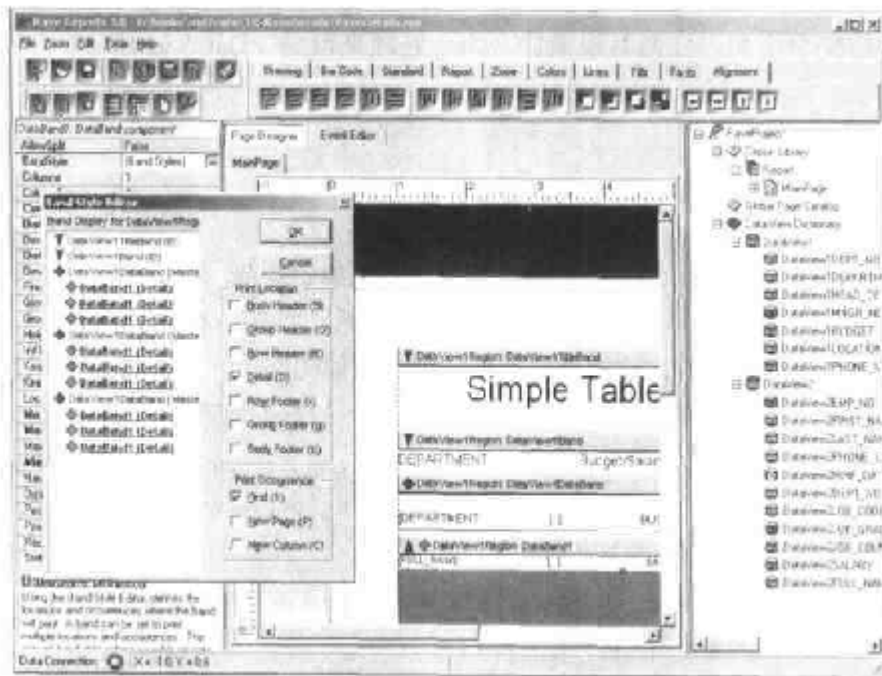


图18.6 主/从报表，其中的Band样式编辑器

警告：为了创建主/详报表，也可以使用Rave提供的相应向导。但在Delphi 7包含的版本中，该向导不能用。在编写本书时，修复此缺陷和其他问题的更新版本仍不可用。

脚本报表

在本章开始已经提到了Rave设计器的事件编辑器窗口，但是还没有用过。这个工具是用来在报表中写代码（脚本）的，如在Delphi中那样，可响应各种组件的事件。在Rave设计器中写脚本可以用老方法定制或调整报表的输出。用于脚本的Rave语言是基于Pascal和Delphi语言变化的，所以应该很容易使用。

RaveDetails范例将大于给定数的Salary值显示为粗体字。显而易见，可以通过编写为每个详细band（employee数据库的每个记录）的实例而运行的脚本来实现。不需要直接更改Font属性，而是向报表页面添加两个不同的FontManager组件，并命名它们使其更易懂：fmPlainFont¹和fmBoldFont。可以打开报表看看它们的属性和布局。

在报表中，为了高亮显示给定范围以外的值，可以处理DataText组件的BeforePrint事件。为此，移动事件编辑器页面，选择连接Salary字段的DataText组件，并选择该事件。在事件代码的编辑窗口中，编写如下代码：

```
if DataView2Salary.AsFloat > 100000 then
    self.FontMirror := fmBoldFont;
```

```

else
    self.FontMirror := fnPlainFont;
end if;

```

该脚本可改变当前对象（self）的FontMirror属性，该对象涉及页面中两个FontManager组件之一——取决于字段的值。注意，DataView2Salary是数据视图（连接当前DataText组件）中一个记录的引用。编译脚本并运行报表，其效果如图18.7所示。

| DEPARTMENT | BudgetSalary | LOCATION |
|------------------------|--------------|---------------|
| Corporate Headquarters | 1,000,000.00 | Menlo Park |
| Lee, Tom | 45,700.00 | |
| Decker, Oliver J. | 212,850.00 | |
| Sales and Marketing | 2,000,000.00 | San Francisco |
| McDonald, Mary S. | 111,262.50 | |
| Yarowski, Michael | 44,000.00 | |
| Engineering | 1,100,000.00 | Menlo Park |
| Anderson, Robert | 105,800.00 | |
| Reusch, Kelly | 37,200.00 | |

图18.7 运行时由脚本确定报表中的粗体文本

警告：每当编辑脚本时，记住单击编译按钮，否则任何修改都无效。

镜像

报告模板可以包含一个或者多个组件，并且可以通过Rave的镜像技术来重复使用。DataMirrorSection组件镜像其他基于一个DataField内容的Sections。镜像部分的DataMirrorSection更加灵活。请记住Sections能够包括任何其他的组件，包括图像、区域、文本等等。

例如，我们可以使用一个DataMirrorSection组件来建立一个简单的报告，产生不同的信封格式，分别对应国际和美国的地址。用于国际用户的模板可能会包括一行来表示国家，它位于信封的中心，而用于美国的格式将不包括国家的信息，而且信息将出现在信封的右下部。

通常，这些设置中有一个会被作为默认设置，如果并未定义默认值，并且这个字段值没有与任何其他设置相匹配，那么使用的格式将会是DataMirrorSection组件的普通内容。

最大限度的计算

除了前面讨论过的简单的CalcText组件外，Rave设计器包含了三个用来处理更复杂情况的组件：CalcTotal、CalcController和CalcOp。

CalcTotal

CalcTotal是CalcText组件的一个不可视的版本。当这个组件被打印的时候，它的值通常存储在一个项目参数中，由DestParam属性定义，并且根据DisplayFormat属性进行格式化。当我们进行总和计算的时候它是很有用的，可以在打印之前用于其他的计算。如果CalcTotal的值只被其他计算组件使用，如CalcOp，我们应该保留DestParam属性为空白。

CalcController

CalcController是一个不可视的组件，可用做CalcText和CalcTotal的控制器——通过它们的Controller属性来实现。当控制器组件被打印时，它会通知所有的计算组件，它要控制他们的操作。这个操作允许一个报告重新计算各组的总和、详细内容，或者是完整的页面——根据CalcController组件的定位来决定。

CalcController组件也能够用一个特定值初始化CalcText或者CalcTotal组件（通过InitCalcVar、InitDataField和InitValue属性）。仅当它被用于CalcText或者CalcTotal组件的Initializer属性中时，CalcController组件将会初始化这些数值。

CalcOp

CalcOp是一个不可视的组件，它允许一个操作（通过Operator属性定义）能够对来自不同数据源的数值进行处理。结果将被存储在一个与CalcTotal类似的项目参数中，如DestParam和DisplayFormat属性所表示的那样。

例如，假设我们需要添加两个DataText组件，如在 $A+B=C$ 中，A和B代表两个DataText组件值，而C代表存储在项目参数之中的结果。这三个源类型具有很多不同的值。

该计算可以使用下面不同类型的数据源开始：

- 一个DataField源是一个表格中的字段，或者在Rave的术语中叫做DataView。因此为了选择一个字段，我们必须首先选择一个DataView。
- 对于一个Value源，我们使用一个数值来填写这个属性。
- 一个CalcVar源代表了其他的计算变量。我们可以从下拉菜单中选择一个，这是一个列举了页面中所有可用的计算变量的菜单。这个数值可以来自其他的CalcOp组件，或者来自某些其他的计算组件。

在选择了数据源之后，我们选择它们之间的操作。Operator属性具有一个下拉菜单，我们能够用它来进行合适的选择。在 $A+B=C$ 的例子中，操作符是coAdd。

有时候一个函数在计算第二个值之前，需要先针对一个数值进行计算。在这个情况下，源的Function属性非常方便。使用一个函数，我们可以转换一个数值，例如将小时转换为分钟、计算一个三角函数，或者进行其他的计算，如平方根或者绝对值。

就像计算的顺序很重要一样，在项目树中，保证组件的顺序也很重要。一个报告执行项目树上的组件。对于CalcOp组件或者其他的计算组件来说，这意味着它们必须处于正确的位置。同样重要的是如果一个源值依赖于其他的组件，那么这些组件就必须首先处于项目树之中。

小结

本章集中讲述了使用Rave为Delphi应用程序增加报表的性能。笔者介绍了基本报表和数据驱动报表两个组件，可以在Delphi应用程序中使用它们连接报表，并且在报表中使用它们显示应用程序传递的数据。本章还介绍了一些Rave的高级特征，如Band样式编辑器、脚本、镜像、计算等。

如果读者想要查找更多的文档，可以参阅Delphi提供的但没有安装在其产品中的PDF文件。这些手册和文件包含在Delphi 7的光碟中。

本章结束了本书涉及的Delphi数据库技术，尽管在本书接下来的部分还会使用数据库（与因特网和网络编程有关）。随后将开始介绍核心技术，如套接字的使用，接下来是HTML，传统Delphi网站开发，和IntraWeb的运用。然后介绍XML和SOAP技术。

第四部分 Delphi, 因特网以及.NET预览

第19章 因特网编程：套接字和Indy组件

随着因特网时代的到来，基于因特网协议编写程序变得很普遍，因此本书将使用五章的篇幅讨论该专题。本章将集中讨论低层套接字编程和因特网协议；第20章介绍服务器端编程，第21章讨论IntraWeb的内容，第22章和第23章介绍Web服务以及XML。

在本章中，我们将从套接字技术的简介开始，然后介绍支持低层套接字编程与多数普通因特网协议的Internet Direct (Indy) 组件。我们还要介绍一些HTTP协议的元素，并且使用数据库数据来建立HTML文件。

虽然读者可能想使用高级协议，但本章仍将从核心概念和低层应用程序开始讨论因特网编程。理解TCP/IP和套接字将有助于读者更容易地理解其他概念。

本章主要包括以下内容：

- 使用套接字
- Internet Direct (Indy) 组件
- 低层套接字编程
- WinInet API
- 邮件协议 (SMTP和POP3)
- HTTP协议
- 创建HTML
- 从数据库到HTML

建立套接字应用程序

Delphi 7带有两个TCP组件：Indy套接字组件 (IdTCPClient与IdTCPServer) 以及本机的Borland组件，它也可以用于Kylix环境，并且位于组件面板的Internet页。Borland组件，TcpClient以及TcpServer是用来代替早期Delphi版本中的ClientSocket与ServerSocket组件的。现在，ClientSocket与ServerSocket组件已经被宣布过时了，虽然它们仍旧可用，但Borland建议使用相应的Indy组件取而代之。

在本章中，我们将集中讨论使用Indy的低层套接字编程，而不只是在讨论支持高级互联网协议的时候才用到它。为了学习更多的关于Indy项目的内容，请参考附加说明“Indy开放

源代码组件”，并且继续阅读，以了解如何使用这些组件进行低层套接字编程。

在我们介绍一个低层的基于套接字的通信范例之前，先来关注一些TCP/IP协议的核心概念，以便于我们了解这个流行最广泛的网络技术的基础知识。

Indy开放源代码组件

Delphi带有一个开放源代码因特网组件的集合，被称为Indy (Internet Direct)。Indy组件，以前被称为WinShoes，是由Chad Hower领导下的一组开发人员建立的，也可以用于Kylix环境。我们可以在www.nevrona.com/indy网址中找到更多的信息以及最新版本的各种组件。

Delphi 7带有Indy 9，但是我们应该检查网站来找到最新的版本。这些组件是免费的，并且带有很多的范例以及帮助文件。与以前的版本相比，Indy 9包括更多的组件（如Indy 8，用于Delphi 6），并且它在组件面板中具有两个新的页面（Indy Intercepts和Indy I/O Handlers）。

Delphi的面板中已经安装了一百多个组件，Indy具有丰富的特性，范围涉及开发各种协议的客户端与服务器TCP/IP应用程序，以及编码与安全性。我们能够从Id前缀来进一步了解Indy。我们不准将所有的组件都列举在这里，而是详细介绍其中的几个。

阻塞与非阻塞连接

当我们在Windows环境中使用套接字的时候，从套接字连接上进行读写操作可以是异步发生的，这样就不会影响网络应用程序中其他代码的执行。这种工作方式被称做非阻塞连接，Windows套接字在数据可用时即发送该数据。除了这种异步方法外，还可以使用阻塞式连接，这样在执行代码的下一行之前，应用程序会等待读或写操作的完成。在这种情况下，必须在两端按顺序编写代码，否则，就不会触发事件。当使用阻塞式连接时，必须在服务器上使用一个线程，而且通常还要在客户端使用一个线程。

Indy组件只使用阻塞连接。因此，任何可能冗长的客户端套接字操作都应该在一个线程内进行；或者使用Indy的IdAntiFreeze组件，这是另一种更简单但是功能有限的解决办法。使用阻塞连接实现协议具有简化程序逻辑的优点，因为这样就无需再使用非阻塞连接的状态机方法了。

所有的Indy服务器都使用多线程的体系结构，我们可以通过IdThreadMgrDefault以及IdThreadMgrPool组件对其进行控制。其中第一个组件是默认使用的，而第二个组件支持线程池，可以提供更快的连接速度。

套接字编程的基础

为了更好地理解套接字组件的工作，我们需要大致了解一些与因特网相关的术语，特别是套接字。因特网的核心是Transmission Control Protocol/Internet Protocol（缩写是TCP/IP），这两种独立协议协同工作，提供了因特网的连接（同时也为专用的内联网提供连接）。简单地说，IP负责定义与选择数据报（因特网传输单元）的传输路径，并指定寻址方案；而TCP负责较高级别的传输服务。

配置局域网：IP地址

如果读者手边有一个局域网可供使用，就能在这个网络中测试后面的程序；否则，只能把同一台计算机既用做客户机，又用做服务器。例如，在这种情况下，使用地址127.0.0.1（或localhost），这个地址总是表示当前计算机的IP地址。如果网络结构非常复杂，就需要请求网络管理员为自己的主机配置相应的IP地址。如果与因特网相连，就要求网络供应商提供该服务。如果想使用几台计算机建立一个简单的网络，可以自己设立IP地址，IP地址是一个32位的二进制数字，通常由圆点分为四部分（称做八位字节，octet）。这些数字背后具有复杂的逻辑关系，其中第一个八位字节说明了地址的类（Class）。

实际上，对于未注册的内部网络，可以使用一些专用的保留IP地址。因特网路由器将忽视这些地址范围，所以可以随意使用这些地址进行测试，而不会妨碍某个现实运行中的网络。例如，将一段“空闲”的IP地址范围192.168.0.0到192.168.0.255，用于规模少于255台设备的小型网络。

局部域名

如何建立IP地址与名称的映射呢？在因特网中，客户程序会搜索域名服务器以获得这个映射关系。此外，也可以从一个本地主机文件中获得相应的信息，该文件是一个易于编辑的文本文件，能够提供本地映射。读者可以查看HOSTS.SAM文件（安装在Windows目录下的一个子目录中，取决于拥有的Windows版本）。我们可以去掉该文件的扩展名，将该文件改名为HOSTS，以激活局部主机映射。

那么在我们的程序中到底应该使用IP地址还是主机名呢？一方面，主机名更容易记忆，而且在IP地址改变时（无论什么原因），主机名不需要改变。另一方面，IP地址不需要进行额外的解析工作（这是一种非常消耗时间的操作，特别是域名解析发生在Web环境中的情况下）。

TCP端口

每个TCP连接都是通过一个端口实现的。端口由16位二进制数字表示，IP地址结合TCP端口可以指定一个因特网连接，或称为一个套接字。运行在同一台计算机上的不同过程不能使用相同的套接字（即相同的端口）。

某些TCP端口是针对一些特殊的高级协议与服务来使用的。换句话说，当需要使用这些特殊服务时应该使用那些特定的端口，而在其他任何情况中都应该避免使用它们。下面是一个简要列表。

| 协议 | 端口 |
|--------------------------------------|-----|
| HTTP (Hypertext Transfer Protocol) | 80 |
| FTP (File Transfer Protocol) | 21 |
| SMTP (Simple Mail Transfer Protocol) | 25 |
| POP3 (Post Office Protocol, 版本3) | 110 |
| Telnet | 23 |

Services文件（与Hosts文件相似的另一个文本文件）列出了服务使用的标准端口号。可以向列表中添加自己定义的表项，来给自己的服务选择名称。客户端套接字总是指向它想要访问的服务器的某个端口或者服务名称。

高层协议

我们在前面多次用到了“协议”这个术语，但它到底是什么意思呢？协议是客户机与服务器之间用于确定通信流程的一系列规则。低层的Internet协议，如TCP/IP，通常由操作系统来实现。但“协议”这个术语也可以用于高层的Internet标准协议（如：http、FTP或SMTP协议）。包含这些协议定义的标准文档位于IETF站点（www.ietf.org）。

如果读者希望自定义通信传输，可以定义自己的（可能是简单的）协议，这是一套规则，用于确定客户可以向服务器发起何种请求，以及服务器如何响应各种不同的请求。稍后大家会看到一个定制协议的范例。与通信协议相比，传输协议处于更高的层次，因为它们来自于TCP/IP提供的传送机制。这样使得协议不仅独立于操作系统而且独立于硬件，同时也独立于物理网络。

套接字连接

如何使用套接字进行通信呢？服务器程序会首先启动，然后它就只是等待来自客户的请求。通常，由客户程序发起一个连接，指向它希望连接的服务器。当客户发出请求时，服务器可以接受这个连接，亦即启动一个特定的服务器端套接字，通过它与客户端套接字相连。

为了支持这种操作模式，可以使用三种不同的套接字连接：

- 客户连接（Client connections）由客户启动，并将本地客户套接字与远程服务器套接字相连。客户套接字必须说明它们想连接的服务器，需要提供服务器的主机名或IP地址及其端口号。
- 监听连接（Listening connections）是等待客户的被动服务器套接字连接。一旦客户发出了新的请求，服务器就会为该特殊连接生成新的套接字，然后返回监听状态。监听服务器套接字必须说明代表其所提供服务的端口号（事实上，客户正是通过该端口进行连接的）。
- 服务器连接是由服务器激活的连接，负责接受来自客户的请求。

这些不同的连接种类只对建立客户与服务器之间的连接具有重要性。一旦连接建立完毕，连接的双方都可以自由发出请求并彼此发送数据。

使用Indy的TCP组件

为了让两个应用程序通过一个套接字进行通信（无论在一个局域网里或者是在因特网1），我们可以使用IdTCPClient和IdTCPServer组件。把其中一个放在一个程序的窗体中，把另一个放在另外一个程序的窗体中，然后，让它们使用相同的端口，并且将客户端程序指向装有服务器端程序的主机，这样我们就可以在两个应用程序之间建立一个连接。例如，在IndySock1项目组中，我们使用了两个组件，带有下面的一些设定：

```
// server program
object IdTCPServer1: TIdTCPServer
```

```

    DefaultPort = 1050
end
// client program
object IdTCPClient1: TIdTCPClient
    Host = 'localhost'
    Port = 1050
end

```

说明：Indy服务器套接字允许到多个IP地址以及端口的绑定——通过使用Bindings集合。

这时，在客户端程序中，我们可以执行下面的代码来连接到服务器上：

```
IdTCPClient1.Connect;
```

服务器程序具有一个列表框，用于记录信息。当一个客户端建立了连接或者中断了连接时，服务器将会记录那个客户端的IP地址，如下面的OnConnect事件处理程序所示：

```

procedure TFormServer.IdTCPServer1Connect(AThread: TIdPeerThread);
begin
    lbLog.Items.Add ('Connected from: ' +
        AThread.Connection.Socket.Binding.PeerIP);
end;

```

现在我们已经建立了一个连接，还需要建立两个程序之间的通信。客户端和服务器的套接字具有读和写的方法，可以用来进行数据发送。但是编写一个可以收到多个不同的命令（通常是基于字符串的）并且在它们上面分别进行不同操作的多线程服务器却决非易事。

然而，Indy利用它的命令结构简化了服务器的开发。在一个服务器中，我们可以定义一系列的命令，并将其存储在IdTCPServer的CommandHandlers集合中。在IndySocket范例中，服务器具有三个处理程序，每一个都用不同的方法实现，以展示一些有用的技巧。

第一个服务器命令称为test，是最简单的一个，因为它是在它的属性中完整定义的。我们在命令处理程序的ReplyNormal属性中设置了一个命令字符串、一个数字代码以及一个字符串结果：

```

object IdTCPServer1: TIdTCPServer
    CommandHandlers = <
        item
            Command = 'test'
            Name = 'TIdCommandHandler0'
            ParseParams = False
            ReplyNormal.NumericCode = 100
            ReplyNormal.Text.Strings = (
                'Hello from your Indy Server')
            ReplyNormal.TextCode = '100'
        end
    end

```

下面是用来执行命令并且显示其响应的客户端代码：

```

procedure TFormClient.btnTestClick(Sender: TObject);
begin

```

```

IdTCPClient1.SendCmd ('test');
ShowMessage (IdTCPClient1.LastCmdResult.TextCode + ' : ' +
  IdTCPClient1.LastCmdResult.Text.Text);
end;

```

对于更复杂的情况，我们应该在服务器上执行这个代码，并且直接从套接字连接上进行读和写的操作。这个方法显示在例子的第二个协议命令中。服务器的第二个命令称为 **execute**，它没有特殊的属性设定（只有命令名称），但是具有下面的 **OnCommand** 事件处理程序：

```

procedure TFormServer.IdTCPServer1TIdCommandHandler1Command(
  ASender: TIdCommand);
begin
  ASender.Thread.Connection.WriteLine ('This is a dynamic response');
end;

```

相应的客户端代码会向套接字连接写入命令名称并且读取一个单行的应答，这是使用不同于前述方法完成的：

```

procedure TFormClient.btnExecuteClick(Sender: TObject);
begin
  IdTCPClient1.WriteLine('execute');
  ShowMessage (IdTCPClient1.ReadLn);
end;

```

这个效果与前面的例子很相似，但是因为它使用了一个低层的方法，因此很容易根据自己的需要来对它进行定制。本范例中第三个和最后一个命令提供了这样的扩展，它允许客户端程序从服务器请求一个位图文件（在一个文件共享体系中）。服务器命令具有一些参数（文件名）并且具有如下定义：

```

object IdTCPServer1: TIdTCPServer
  CommandHandlers = <
    item
      CmdDelimiter = ' '
      Command = 'getfile'
      Name = 'TIdCommandHandler2'
      OnCommand = IdTCPServer1TIdCommandHandler2Command
      ParamDelimiter = ' '
      ReplyExceptionCode = 0
      ReplyNormal.NumericCode = 0
      Tag = 0
    end>

```

这段代码使用第一个参数作为文件的名称并且在一个流中返回它。当发生错误的时候，它产生一个异常，这个异常将被服务器组件截取，然后连接就被中断（虽然不是很理想的解决办法，但是非常简单和安全）：

```

procedure TFormServer.IdTCPServer1TIdCommandHandler2Command(
  ASender: TIdCommand);

```



```

var
  filename: string;
  fstream: TFileStream;
begin
  if Assigned (ASender.Params) then
    filename := HttpDecode (ASender.Params [0]);
  if not FileExists (filename) then
    begin
      ASender.Response.Text := 'File not found';
      lbLog.Items.Add ('File not found: ' + filename);
      raise EIdTCPServerError.Create ('File not found: ' + filename);
    end
  else
    begin
      fstream := TFileStream.Create (filename, fmOpenRead);
      try
        ASender.Thread.Connection.WriteStream(fstream, True, True);
        lbLog.Items.Add ('File returned: ' + filename +
          ' (' + IntToStr (fStream.Size) + ')');
      finally
        fstream.Free;
      end;
    end;
  end;
end;

```

用该参数调用HttpDecode工具函数时，要求将一个带空格的路径名称解码为一个单独的参数，反之，客户端程序要调用HttpEncode。正如我们看到的，服务器还要记录返回的文件和它们的大小，或者是一个错误消息。而客户端程序会读取这个流并且将它复制到一个图像组件中，直接进行显示（如图19.1所示）：

```

procedure TFormClient.btnGetFileClick(Sender: TObject);
var
  stream: TStream;
begin
  IdTCPClient1.WriteLine('getfile ' + HttpEncode (edFileName.Text));
  stream := TMemoryStream.Create;
  try
    IdTCPClient1.ReadStream(stream);
    stream.Position := 0;
    Image1.Picture.Bitmap.LoadFromStream (stream);
  finally
    stream.Free;
  end;
end;

```

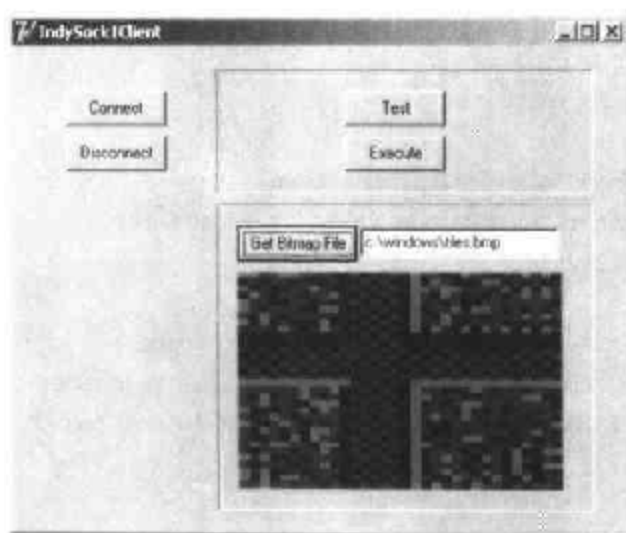


图19.1 IndySock1范例中的客户端程序

在套接字连接上发送数据库数据

使用此技术可以编写一个在套接字上移动数据库记录的应用程序。其设计思想是，为数据输入编写一个前端程序，为数据存储编写一个后端程序。客户应用程序将有一个简单的数据输入窗体，并且使用一个数据库表格，其中有Company、Address、State、Country、Email与Contact等字符串字段，以及一个用于公司ID（被称做CompID）的浮点字段。

说明：在套接字上移动数据库记录其实与操作DataSnap以及一个套接字连接组件（在第16章“多层DataSnap应用程序”中介绍过）一样，或者可以使用内建于Delphi中的SOAP功能，这个课题将在第23章“Web服务和SOAP”中讨论。

这里建立的客户程序基于一个ClientDataSet，其结构存储在当前目录中（读者可以在OnCreate事件处理程序中看到相关代码）。客户端的核心对象方法是Send All按钮的OnClick事件处理程序，它用于向服务器发送所有的新记录。新记录的确定是通过查看该记录是否具有CompID字段的有效值来实现的。事实上，该字段不是用户设置的，而是在发送数据时由服务器应用程序确定的。

对于所有新记录来说，客户程序会将字段信息包装在一个字符串列表中，并使用结构FieldName=FieldValue。然后将与整个列表（一个记录）相关联的字符串发送到服务器中。这时，程序会等待服务器发回公司的ID，然后将其保存在当前记录中。所有这些代码都在一个线程中发生，以避免在长时间操作时，阻塞用户的接口。通过单击Send按钮，用户可以开始一个新的线程：

```
procedure TForm1.btnSendClick(Sender: TObject);
var
    SendThread: TSendThread;
begin
    SendThread := TSendThread.Create(cds);
    SendThread.OnLog := OnLog;
    SendThread.ServerAddress := EditServer.Text;
    SendThread.Resume;
end;
```

这个线程具有一些参数：在构造器中传递的数据集、保存在ServerAddress属性中的服务器地址以及一个用来写入主窗体的日志事件（在一个安全的Synchronize调用中）。这个线程的代码要创建并且打开一个连接，并且不断发送记录直到结束：

```

procedure TSendThread.Execute;
var
  I: Integer;
  Data: TStringList;
  Buf: String;
begin
  try
    Data := TStringList.Create;
    fIdTcpClient := TIdTcpClient.Create (nil);
    try
      fIdTcpClient.Host := ServerAddress;
      fIdTcpClient.Port := 1051;
      fIdTcpClient.Connect;
      fDataSet.First;
      while not fDataSet.Eof do
        begin
          // if the record is still not logged
          if fDataSet.FieldByName('CompID').IsNull or
            (fDataSet.FieldByName('CompID').AsInteger = 0) then
            begin
              FLogMsg := 'Sending ' + fDataSet.FieldByName('Company').AsString;
              Synchronize(DoLog);
              Data.Clear;
              // create strings with structure "FieldName=Value"
              for I := 0 to fDataSet.FieldCount - 1 do
                Data.Values [fDataSet.Fields[I].FieldName] :=
                  fDataSet.Fields [I].AsString;
              // send the record
              fIdTcpClient.Writeln ('senddata');
              fIdTcpClient.WriteString (Data, True);
              // wait for reponse
              Buf := fIdTcpClient.ReadLn;
              fDataSet.Edit;
              fDataSet.FieldByName('CompID').AsString := Buf;
              fDataSet.Post;
              FLogMsg := fDataSet.FieldByName('Company').AsString +
                ' logged as ' + fDataSet.FieldByName('CompID').AsString;
              Synchronize(DoLog);
            end;
            fDataSet.Next;
          end;
        end;

```

```

    finally
        fIdTcpClient.Disconnect;
        fIdTcpClient.Free;
        Data.Free;
    end;
except
    // trap exceptions in case of dataset errors
    // (concurrent editing and so on)
end;
end;
end;

```

现在，再来看看服务器。该程序有一个数据库表格，也存储在本地目录中，含有向客户应用程序数据表格中添加的两个新字段：**LoggedBy**字符串字段，**LoggedOn**数据字段。在服务器收到数据时，两个新添字段的值会自动确定，还有**CompID**字段的值。所有这些操作都是在senddata命令的处理程序中实现的：

```

procedure TForm1.IdTCPServer1TIdCommandHandler0Command(
    ASender: TIdCommand);
var
    Data: TStrings;
    I: Integer;
begin
    Data := TStringList.Create;
    try
        ASender.Thread.Connection.ReadStrings(Data);
        cds.Insert;
        // set the fields using the strings
        for I := 0 to cds.FieldCount - 1 do
            cds.Fields[I].AsString :=
                Data.Values [cds.Fields[I].FieldName];
        // complete with ID, sender, and date
        Inc(ID);
        cdsCompID.AsInteger := ID;
        cdsLoggedBy.AsString := ASender.Thread.Connection.Socket.Binding.PeerIP;
        cdsLoggedOn.AsDateTime := Date;
        cds.Post;
        // return the ID
        ASender.Thread.Connection.WriteLine(cdsCompID.AsString);
    finally
        Data.Free;
    end;
end;

```

除了某些数据有可能丢失以外，当字段的顺序不同或者它们不匹配的时候，不会出现问题，因为数据是存储在**FieldName=FieldValue**结构中的。在收到了所有数据并将其发送给本地数据表格之后，服务器会将公司ID发回客户端。在发送记录之后，客户程序会进入等待

模式，这是由于接收服务器反馈而导致的状态。当接收反馈时，客户程序会保存公司ID，以便在发送时用于标志记录。如果用户改动记录，就无法向服务器发送更新数据。为了实现该操作，可以向客户端数据库表格添加改动的字段，并让服务器检测它是否正接收一个新字段或一个被改动过的字段。使用改动过的字段时，服务器除了更新已有记录外，不会再添加新记录。

如图19.2所示，服务器程序有两个页面，一页显示日志记录，另一页用DBGnd组件来显示服务器数据库表格的当前数据。客户端程序是一个基于窗体的数据表项，并且带有一些按钮，用于发送记录以及删除已发送的记录（针对那些收回的ID）。

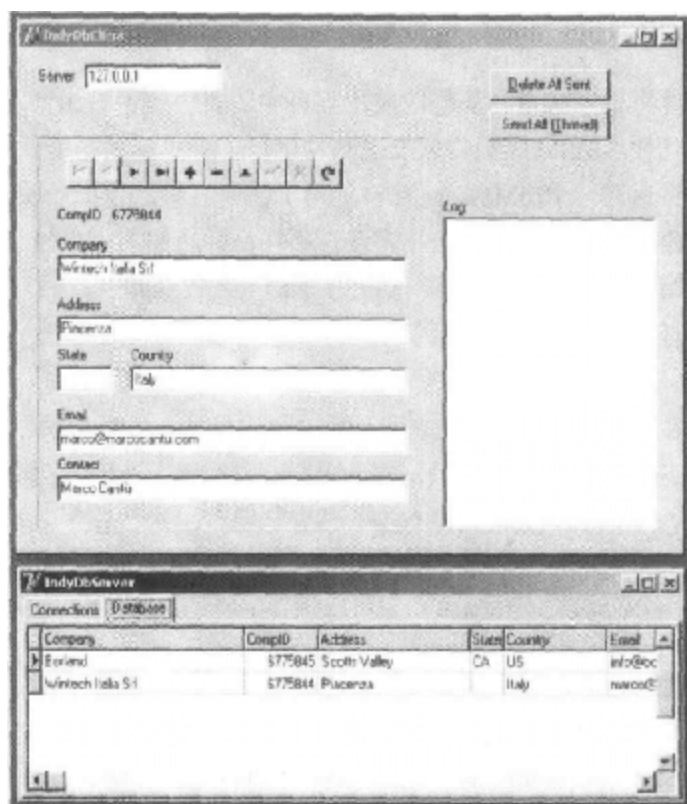


图19.2 数据库套接字范例中的客户端与服务器程序

发送和接收邮件

可能在因特网上最普遍的操作就是发送和接收E-mail了。不需要编写完整的应用程序来处理E-mail，因为现存的程序已相当完整。出于这个原因，这里不想介绍编写一般邮件程序的方法。读者能在那些Delphi的Indy演示示例中发现一些范例。与创建通用的邮件程序相比，我们能对邮件组件和协议做些什么呢？我们下面从两方面进行介绍：

自动生成邮件消息 我们编写的应用程序将包含一个About框，用于将注册消息发送回到营销部门；或者包含一个向技术支持发送请求的指定菜单项。程序员甚至可以在出现异常时决定是否提供技术支持链接。另一个相关的任务是自动将消息分派到人员列表，或从网络站点产生自动化消息，在本章最后将展示一个这样的示例。

使用邮件协议与不经常在线的用户进行通信 当我们必须在那些不经常在线的用户间移动数据时，我们可以在服务器上编写应用程序来对他们进行同步，并且为每位用户提供一个个性化的客户端应用程序，用于与服务器交互作用。使用现有的服务器应用程序也是一种选择，例如一个mail服务器，并基于mail协议编写两个个性化的程序。数据通过该连接发送时将以指定的方式格式化，因此将为这些消息使用指定的E-mail地址。作为一个示例，可以重新编写以前的示例来分派mail消息，以代替使用定制套接字连接。这种做法具有防火墙的优点，并允许服务器暂时离线，因为这意味着请求将被保存在mail服务器上。

发送和接收邮件

使用Indy的邮件协议意味着在我们的应用程序中放置一个消息组件（IdMessage），使用数据填充它，并且使用IdSMTP组件来发送邮件信息。为了从邮箱中获得邮件，可使用一个IdPop3组件，它将返回一个IdMessage对象。为了让读者对这个工作的流程有一个认识，笔者编写了一个程序，用于一次向多个用户发送邮件，而且使用了存储在一个ASCII文件中的列表。最初我编写该程序是为了向在网站上注册的人们发送邮件，后来又通过填加数据库支持以及自动读取订购记录功能，扩展了该程序。但程序的最初版本仍可用来介绍如何使用Indy的SMTP组件。

SendList程序在本地文件保存列表的名字和E-mail地址，它们被展示在列表框中。该程序提供了几个按钮，允许填加和删除项，或通过删除、编辑、然后再次填加项，对其进行修改。当程序关闭时，更新的列表会自动保存。现在让我们讨论该程序中的有趣部分。在设计时，图19.3显示的最顶层面板允许我们输入标题、发送者地址和用来连接mail服务器的信息（主机名、用户名和密码）。

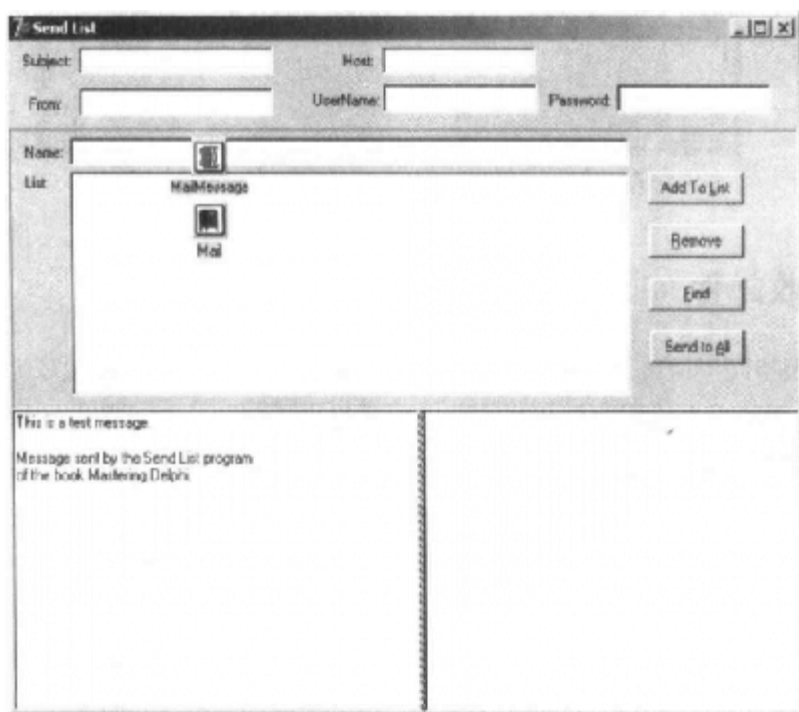


图19.3 设计时的SendList程序

读者可能希望保存编辑框中的值，譬如将其放入一个INI文件中。这里没有这么做，因为我们不希望别人看到自己的邮件连接的详情。编辑框的值以及地址列表允许我们发送邮件信息系列（在对它们进行定制之后），其代码如下：

```
procedure TMainForm.BtnSendAllClick(Sender: TObject);
var
    nItem: Integer;
    Res: Word;
begin
    Res := MessageDlg ('Start sending from item ' +
        IntToStr (ListAddr.ItemIndex) + ' (' +
        ListAddr.Items [ListAddr.ItemIndex] + ')?'#13 +
        '(No starts from 0)', mtConfirmation, [mbYes, mbNo, mbCancel], 0);
    if Res = mrCancel then
        Exit;
    if Res = mrYes then
        nItem := ListAddr.ItemIndex
    else
        nItem := 0;
    // connect
    Mail.Host := eServer.Text;
    Mail.UserName := eUserName.Text;
    if ePassword.Text <> '' then
        begin
            Mail.Password := ePassword.Text;
            Mail.AuthenticationType := atLogin;
        end;
    Mail.Connect;
    // send the messages, one by one, prepending a custom message
    try
        // set the fixed part of the header
        MailMessage.From.Name := eFrom.Text;
        MailMessage.Subject := eSubject.Text;
        MailMessage.Body.SetText (reMessageText.Lines.GetText);
        MailMessage.Body.Insert (0, 'Hello');
        while nItem < ListAddr.Items.Count do
            begin
                // show the current selection
                Application.ProcessMessages;
                ListAddr.ItemIndex := nItem;
                MailMessage.Body [0] := 'Hello ' + ListAddr.Items [nItem];
                MailMessage.Recipients.EMailAddresses := ListAddr.Items [nItem];
                Mail.Send(MailMessage);
                Inc (nItem);
            end;
        finally // we're done
```

```
Mail.Disconnect;  
end;  
end;
```

使用邮件的另一个有趣示例是在应用程序中向开发者通知问题，这些技术可能只适用于内部应用程序，而不适用于广泛传播的应用程序。我们可以修改第2章中的ErrorLog示例获得这种效果，并在异常（或其他值定事件）发生时，发送邮件。

使用HTTP工作

处理邮件消息当然有趣，邮件协议仍是最广泛的因特网协议。另一个流行协议是HTTP协议，被网络服务器和网络浏览器使用。本章的余下部分将讨论该协议，同时也包括一些对HTML的讨论，接下来的两章中我们仍然要对它进行介绍。

网络的客户端其主要行为是浏览——阅读HTML文件。除了建立定制浏览器外，也可以在程序中嵌入Internet Express ActiveX控件（我们在第12章的WebDemo范例中曾经这样做过）。还有一种办法就是直接激活安装在用户计算机上的浏览器，例如，通过调用ShellExecute对象方法打开HTML页（定义在ShellApi单元中）：

```
ShellExecute(Handle, 'open', FileName, '', '', sw_ShowNormal);
```

使用ShellExecute，我们可以简单地执行一个文档，例如一个文件。Windows将使用HTM扩展来启动程序，使用动作作为传递的参数（在本例中，是“open”，但是传递nil将会激活一个能产生同样结果的标准操作）。我们还可使用类似的调用来浏览网站，譬如使用类似“http://www.example.com”这样的字符串代替文件名。这时，系统将识别请求的http部分，从而申请一个网络浏览器并连到它。

在服务器端，创建并获得HTML页。同时，也有办法生成静态页，从数据库表格抽取新数据，按需要更新HTML文件。在其他情况下，需要基于来自用户的请求动态地产生页面。

作为开端，我们将通过建立一个简单但完整的客户机和服务器来讨论HTTP，然后讨论HTML生成器组件。在第20章，将从该“核心技术”级移到Delphi支持的网络RAD开发风格上，简单介绍Web服务器扩展技术（CGI、ISAPI和Apache模块）并讨论WebBroker和WebSnap结构。

理解HTTP内容

作为使用HTTP协议的一个范例，我们这里决定编写一个非常特殊的查找应用程序。该程序与Google网站相连，负责查找一个关键字，并检索发现的前一百个站点。程序没有显示结果HTML文件，而是通过分析它，仅将相关站点的URL提取到一个列表框中。这样，该程序一次解释了两种技术：检索Web页与分析HTML代码。

为了说明怎样用阻塞连接进行工作，例如那些由Indy使用的阻塞连接，我们决定在实现该程序时，使用后台线程进行处理。该方法还有一个优点，就是可以一次启动多个搜索操作。当输入一个要寻找的URL时，由WebFind应用程序使用的线程类将接收到strUrl。

该类有两个输出过程，AddToList与ShowStatus，它们在Synchronize对象方法中被调用。这两个对象方法的代码可以向主窗体发送一些结果或反馈信息，分别依靠向列表框中添加一

行以及改变状态条的SimpleText属性来实现。当然，线程的主要对象方法是Execute方法。然而，在研究它之前，先来看看主窗体是如何激活线程的：

```

const
    strSearch = 'http://www.google.com/search?as_q=';

procedure TForm1.BtnFindClick(Sender: TObject);
var
    FindThread: TFindWebThread;
begin
    // create suspended, set initial values, and start
    FindThread := TFindWebThread.Create (True);
    FindThread.FreeOnTerminate := True;
    // grab the first 100 entries
    FindThread.strUrl := strSearch + EditSearch.Text + '&num=100';
    FindThread.Resume;
end;

```

URL字符串由搜索引擎的主地址组成，后面带有一些参数。第一个参数as_q，说明正在搜索的单词。第二个参数num=100，说明要取回的站点的数量，我们不能随意设置站点的数量，而只能限于少许的选择，100是最大的可能值。

警告：在本书编写和测试的过程中，WebFind程序的编写与测试都是在Google Web站点的服务器上进行的。然而，该站点上的定制软件任何时候都可能改变，这也许会影响WebFind程序的正常操作。这个程序在“Mastering Delphi 6”（中文版名为《Delphi 6从入门到精通》）一书中也用到了，但是那时它没有用户代理HTTP标头，后来Google改变了它的服务器软件并阻止了这种请求。向用户代理添加任何值都可以修补这个问题。

被Resume调用激活的线程的Execute对象方法，通过简单调用两个对象方法进行工作，如程序清单19.1所示。首先是GrabHtml，它使用动态创建的IdHttp组件连接到HTTP服务器，并读取带有查找结果的HTML。第二个对象方法是HtmlToList，用于从结果strRead字符串中抽取出引用到其他网络站点的URL。

程序清单19.1 WebFind程序的TFindWebThread类

```

unit FindTh;

interface

uses
    Classes, IdComponent, SysUtils, IdHTTP;

type
    TFindWebThread = class (TThread)
    protected
        Addr, Text, Status: string;
        procedure Execute; override;
        procedure AddToList;
        procedure ShowStatus;
        procedure GrabHtml;

```

```

    procedure HtmlToList;
    procedure HttpWork (Sender: TObject; AWorkMode: TWorkMode;
        const AWorkCount: Integer);
    public
        strUrl: string;
        strRead: string;
    end;

implementation
    { TFindWebThread }

uses
    WebFindF;

    procedure TFindWebThread.AddToList;
    begin
        if Form1.ListBox1.Items.IndexOf (Addr) < 0 then
        begin
            Form1.ListBox1.Items.Add (Addr);
            Form1.DetailsList.Add (Text);
        end;
    end;

    procedure TFindWebThread.Execute;
    begin
        GrabHtml;
        HtmlToList;
        Status := 'Done with ' + StrUrl;
        Synchronize (ShowStatus);
    end;

    procedure TFindWebThread.GrabHtml;
    var
        Http1: TIdHTTP;
    begin
        Status := 'Sending query: ' + StrUrl;
        Synchronize (ShowStatus);
        Http1 := TIdHTTP.Create (nil);
        try
            Http1.Request.UserAgent := 'User-Agent: NULL';
            Http1.OnWork := HttpWork;
            strRead := Http1.Get (StrUrl);
        finally
            Http1.Free;
        end;
    end;

    procedure TFindWebThread.HtmlToList;
    var

```

```

    strAddr, strText: string;
    nText: integer;
    nBegin, nEnd: Integer;
begin
    Status := 'Extracting data for: ' + StrUrl;
    Synchronize (ShowStatus);
    strRead := LowerCase (strRead);
    repeat
        // find the initial part HTTP reference
        nBegin := Pos ('href=http', strRead);
        if nBegin <> 0 then
            begin
                // get the remaining part of the string, starting with 'http'
                strRead := Copy (strRead, nBegin + 5, 1000000);
                // find the end of the HTTP reference
                nEnd := Pos ('>', strRead);
                strAddr := Copy (strRead, 1, nEnd - 1);
                // move on
                strRead := Copy (strRead, nEnd + 1, 1000000);
                // add the URL if 'google' is not in it
                if Pos ('google', strAddr) = 0 then
                    begin
                        nText := Pos ('</a>', strRead);
                        strText := copy (strRead, 1, nText - 1);
                        // remove cached references and duplicates
                        if (Pos ('cached', strText) = 0) then
                            begin
                                Addr := strAddr;
                                Text := strText;
                                AddToList;
                            end;
                        end;
                    end;
                until nBegin = 0;
            end;
        procedure TFindWebThread.HttpWork(Sender: TObject; AWorkMode: TWorkMode;
            const AWorkCount: Integer);
        begin
            Status := 'Received ' + IntToStr (AWorkCount) + ' for ' + strUrl;
            Synchronize (ShowStatus);
        end;
        procedure TFindWebThread.ShowStatus;
        begin
            Form1.StatusBar1.SimpleText := Status;
        end;
    end.

```

这个程序寻找的是href=http子字符串出现的情况，复制文本直到到达邻近的“>”符号。如果找到包含有“google”一词的字符串，或者它的目标文本包括“cached”一词，它就被从结果中忽略掉。我们可以在图19.4中看到这段代码的输出情况。我们可以同时启动多个搜索，但是注意，搜索的结果将被加载到同一个备注组件中。

WinInet API

当我们需要使用FTP与HTTP协议时，除了使用特殊的VCL组件外，还可以使用Microsoft在WinInet DLL中提供的API函数。该库是操作系统核心的一部分，可以在Microsoft Web站点上找到并下载。它在Windows套接字API的基础上实现了FTP与HTTP协议。

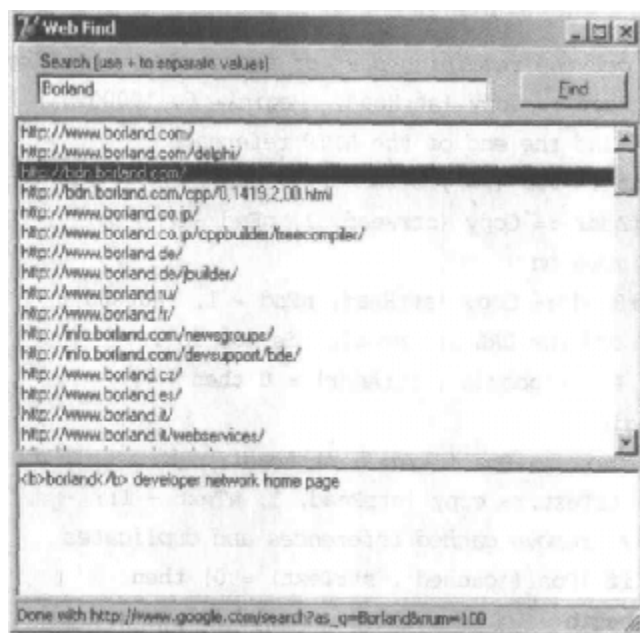


图19.4 WebFind应用程序能够用来在Google的搜索引擎上搜索一系列站点

只需使用三个调用——InternetOpen、InternetOpenURL与InternetReadFile，我们就可以检索与任何URL相应的文件，并存储一份本地拷贝或者对它进行分析。其他简单的对象方法可以用于FTP协议；建议读者查找WinInet.pas Delphi单元的源代码，它列出了所有的函数。

提示：WinInet库的帮助文件不是Delphi的SDK帮助文件的一部分。但是可以在MSDN中找到：
msdn.microsoft.com/library/en-us/wininet/wininet/wininet_reference.asp。

InternetOpen函数可建立一个普通连接并返回一个能在InternetOpenURL调用中使用的句柄。而第二个调用返回一个对URL的处理，使我们能通过InternetReadFile函数读取数据块。在下面的范例代码中，数据被存储在本地字符串中。当所有数据被读取后，程序将两次调用InternetCloseHandle函数：以关闭与URL和因特网会话的连接：

```
var
  hHttpSession, hReqUrl: HInternet;
  Buffer: array [0..1023] of Char;
  nRead: Cardinal;
```

```

strRead: string;
nBegin, nEnd: Integer;
begin
  strRead := '';
  hHttpSession := InternetOpen ('FindWeb', INTERNET_OPEN_TYPE_PRECONFIG,
    nil, nil, 0);
  try
    hReqUrl := InternetOpenURL (hHttpSession, PChar(StrUrl), nil, 0,0,0);
    try // read all the data
      repeat
        InternetReadFile (hReqUrl, @Buffer, sizeof (Buffer), nRead);
        strRead := strRead + string (Buffer);
      until nRead = 0;
    finally
      InternetCloseHandle (hReqUrl);
    end;
  finally
    InternetCloseHandle (hHttpSession);
  end;
end;

```

浏览自己

尽管可能读者对于编写新的网络浏览器不会很感兴趣，但使用在CLX上可用的HTML浏览器（TextBrowser控件）来了解怎样从因特网中获取HTML文件并在本地显示它，可能是比较有意思的事情。将该控件连接到Idny HTTP客户，就可在几分钟内构成一个具有有限导航功能的最简单的纯文本浏览器。代码如下：

```
TextBrowser1.Text := IdHttp1.Get (NewUrl);
```

在上述代码中，NewUrl是想要访问的Web资源的完整地址。在BrowseFast范例中，该URL被输入到一个组合框中，用于追踪近来的请求。一个相似调用的结果将被返回到网页的文本部分（如图19.5所示），因为抓取图形内容要求更复杂的代码。事实上，将TextBrowser控件定义为本地文件查看器比定义为浏览器更为准确。

到目前为止，向程序中添加的都只是一些有限支持超链接的功能。当用户在链接上移动鼠标时，它的链接文本便被复制到本地变量（NewRequest）中，用于单击控件时计算新的HTTP请求。用请求合并当前地址（LastUrl）相当琐碎，即使依靠Idny提供的IdUrl类帮助也是如此。下面是提供的代码，它只处理简单的情况：

```

procedure TForm1.TextBrowser1Click(Sender: TObject);
var
  Uri: TIdUri;
begin
  if NewRequest <> '' then
    begin
      Uri := TIdUri.Create (LastUrl);

```

```

if Pos ('http:', NewRequest) > 0 then
  GoToUrl (NewRequest)
else if NewRequest [1] = '/' then
  GoToUrl ('http://' + Uri.Host + NewRequest)
else
  GoToUrl ('http://' + Uri.Host + Uri.Path + NewRequest);
end;
end;
end;

```

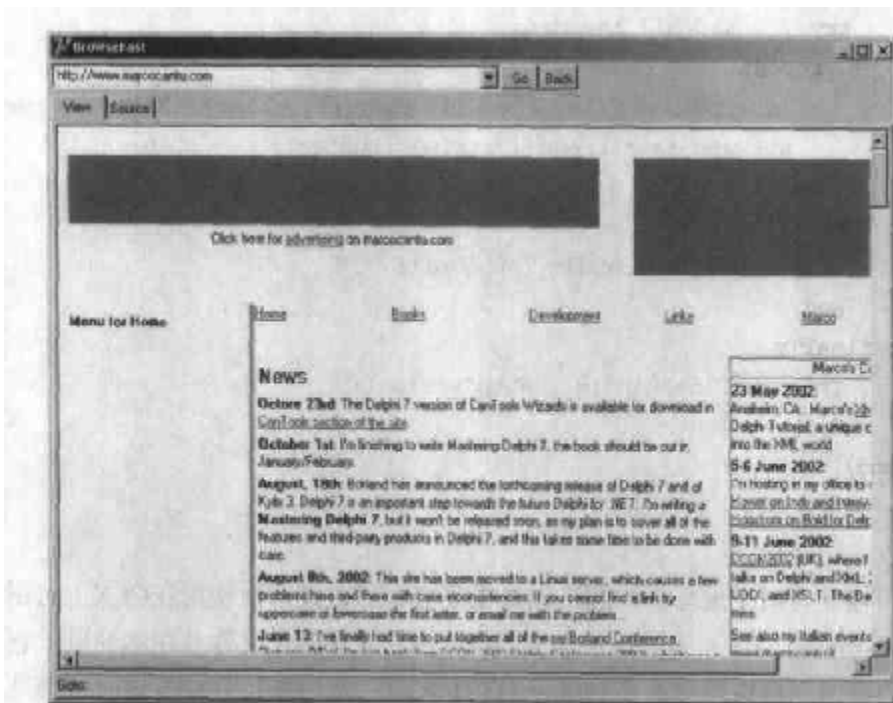


图19.5 BrowseFast纯文本浏览器的输出

该示例相当琐碎而且没有实用价值，而建立一个浏览器多少包含一些通过HTTP连接与显示HTML文件的功能。

一个简单的HTTP服务器

开发HTTP服务器的情况则大不相同了。建立一个服务器来传递基于HTML文件的静态页并不是很简单的，尽管用Indy演示程序为我们提供了良好的开端。但是，一个定制的HTTP服务器可能更关注于建立纯粹的动态站点。相关的内容我们将在第20章进行详细介绍。

为了展示如何着手开发一个定制的HTTP服务器，我们建立了一个HttpServ示例。该程序有一个包含列表框与IdHTTPServer组件的窗体，这个列表框的作用是记录请求信息，IdHTTPServer组件具有下列设置：

```

object IdHTTPServer1: TIdHTTPServer
  Active = True
  DefaultPort = 8080
  OnCommandGet = IdHTTPServer1CommandGet
end

```

该服务器使用端口8080代替标准端口80，因此能与另一个Web服务器共同运行。所有的定制代码都在OnCommandGet事件处理器上，它简单地返回一个固定网页以及一些关于请求自身的信息：

```

procedure TForm1.IdHTTPServer1CommandGet(AThread: TIdPeerThread;
  RequestInfo: TIdHTTPRequestInfo; ResponseInfo: TIdHTTPResponseInfo);
var
  HtmlResult: String;
begin
  // log
  Listbox1.Items.Add (RequestInfo.Document);
  // respond
  HtmlResult := '<h1>HttpServ Demo</h1>' +
    '<p>This is the only page you'll get from this example.</p><hr>' +
    '<p>Request: ' + RequestInfo.Document + '</p>' +
    '<p>Host: ' + RequestInfo.Host + '</p>' +
    '<p>Params: ' + RequestInfo.UnparsedParams + '</p>' +
    '<p>The headers of the request follow: <br>' +
    RequestInfo.RawHeaders.Text + '</p>';
  ResponseInfo.ContentText := HtmlResult;
end;

```

通过使用浏览器命令行传递一个路径和一些参数，我们将看到它们被重新解释和显示。例如，图19.6显示了下面这个命令行的效果：

http://localhost:8080/test?user=marco

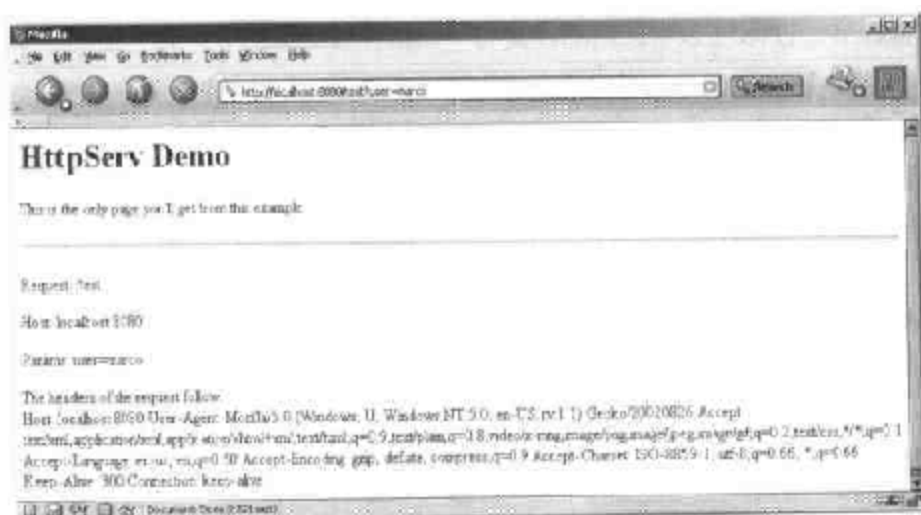


图19.6 通过将 一个浏览器连接到定制的HttpServ程序所显示的页面

这个范例看上去太微不足道了，我们将在下面的一节中提供一个更有趣的版本，那里我们将讨论使用Delphi的生成器组件生成HTML。

说明：如果我们打算使用Delphi建立一个高级的Web服务器或者其他的互联网服务器，那么作为Indy组件的一个替代，我们应该留意DXSock组件，它来自Brain Patchwork DX (www.dxsock.com)。

生成HTML

超文本标记语言 (Hypertext Markup Language) 是Web上一种应用非常广泛的超文本格式, 我们通常对它的缩写HTML更加熟悉。HTML是Web浏览器可以读取的标准格式。它由W3C这个掌控Internet的团体 (World Wide Web Consortium, www.w3.org) 定义。HTML语言的标准文本以及其他一些有趣的链接可以在www.w3.org/markup上找到。

Delphi的HTML生成器组件

Delphi的HTML生成器组件 (在组件面板的Internet页上) 可以用来生成HTML文件, 特别是将数据库表格转换成HTML语言表格。很多开发人员相信, 只有在编写Web服务器扩展程序时使用这些组件才有意义。尽管这些组件确实是为此种目的引入的, 并且是WebBroker技术的组成部分, 但是在必须生成静态HTML文件的应用程序中, 我们仍然可以使用五种生成器组件中的三种来完成任务。

HtmlProd范例解释了HTML生成器组件的用法, 在对这个范例进行研究之前, 让我们先对HTML生成器组件的作用进行一个总结:

- 最简单的HTML生成器组件是PageProducer, 它可以处理含有特殊标记的HTML文件。HTML语言可以存储于一个外部文件内, 也可以存储在一个内部的字符串列表中。该方法的优点是, 可以让程序员使用自己喜欢的HTML编辑器生成这样的文件。在运行时, PageProducer会将特殊标记转换成HTML代码, 并给出一种简明的方法来改动HTML文档的不同部分。特殊标记的基本形式是<#tagname>, 而且还可以在标记中提供命名参数。我们将在PageProducer的OnTag事件处理程序中处理标记。
- DataSetPageProducer可以自动替换与一个相连数据源的字段名称相对应的标记, 本组件扩展了PageProducer的功能。
- DataSetTableProducer组件通常用于显示数据表格、查询或其他数据集合的内容。其意图是用一种非常简单也非常灵活的方法, 从某数据集合中生成一个HTML数据表格。该组件有非常优秀的预览功能, 所以可以在设计时直接看到将要在浏览器中显示的HTML输出形式。
- QueryTableProducer和SQLQueryTableProducer组件与DataSetTableProducer组件非常相似, 但它们 (分别用于BDE或者dbExpress) 专门用于建立基于HTML查找窗体输入的参数化查询。这个组件对于独立的应用程序没有什么意义, 因此, 我们将在第20章对这些组件进行详细介绍。

生成HTML页面

使用标记 (通过“#”符号引入) 的简单例子是建立一个HTML文件, 用它显示当前日期或与当前日期相关的日期值, 如某个终止日期。如果研究HtmlProd范例, 我们会发现在HTMLDoc字符串列表中指定的一个带有内部HTML代码的PageProducer1组件:

```
<html>  
<head>
```



```

<title>Producer Demo</title>
</head>
<body>
<h1>Producer Demo</h1>
<p>This is a demo of the page produced by the <b><#appname></b> application on
<b><#date></b>.</p>
<hr>
<p>The prices in this catalog are valid until <b><#expiration
days=21></b>.</p>
</body>
</html>

```

警告：如果使用HTML编辑器来准备该文件，它可能会自动在标记参数上加引号，如days="21"，因为这是HTML 4和XHTML所需要的。PageProducer组件有一个新的StripParamQuotes属性，可以激活这个属性，以便在组件分析代码（在调用OnHTMLTag事件处理程序之前）时，删除这些额外的引号。

Demo Page按钮用于将PageProducer组件的输出复制到一个Memo的Text中去，当我们调用PageProducer组件的Content函数时，它会读取输入的HTML代码，对它进行分析，并为每个特殊标记触发OnTag事件处理程序。在该事件的处理方法中，程序检测了标记的值（在TagString参数中传递），并返回一个不同的HTML文本（在ReplaceText引用参数中），从而产生如图19.7所示的输出结果。

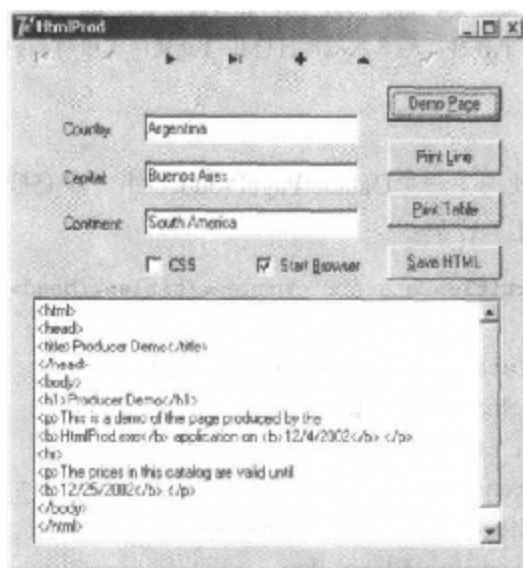


图19.7 HtmlProd程序的输出，它是PageProducer组件的一个简单演示，当用户单击Demo Page按钮的时候，将产生这个输出

```

procedure TFormProd.PageProducer1HTMLTag(Sender: TObject;
  Tag: TTag; const TagString: String; TagParams: TStrings;
  var ReplaceText: String);
var
  nDays: Integer;
begin

```

```

if TagString = 'date' then
  ReplaceText := DateToStr (Now)
else if TagString = 'appname' then
  ReplaceText := ExtractFilename (Forms.Application.Exename)
else if TagString = 'expiration' then
begin
  nDays := StrToIntDef (TagParams.Values['days'], 0);
  if nDays <> 0 then
    ReplaceText := DateToStr (Now + nDays)
  else
    ReplaceText := '<i>{expiration tag error}</i>';
end;
end;

```

特别需要注意用于转换最后一个标记(#expiration)的代码,它需要一个参数。PageProducer组件在一个字符串(是TagParams列表的一部分)中放置了标记参数的全部文本(在本例中,days=21)。为了提取该字符串的数值部分(等号后面的部分),可以使用TagParams字符串列表的Values属性,并同时查找合适的条目。如果不能确定参数的位置,或者参数的数值不是整型的,应用程序将产生一条错误消息。

提示: PageProducer组件支持用户自定义的标记,这些标记可以是喜欢的任何字符串,但是你应该首先检查由Ttags枚举定义的特殊标记。可能出现的值包括tgLink(即link标记)、tgImage(即img标记)、tgTable(即table标记)以及其他的一些标记。如果创建一个自定义的标记,如同在PageProd范例中那样,那么发送给HTML Tag处理程序的Tag参数值将是tgCustom。

生成数据页面

HtmlProd范例中还使用了一个DataSetPageProducer组件,它被连接到一个数据库表格,并且使用了下面的HTML源代码:

```

<html><head><title>Data for <#name></title></head>
<body>
<h1><center>Data for <#name></center></h1>
<p>Capital: <#capital></p>
<p>Continent: <#continent></p>
<p>Area: <#area></p>
<p>Population: <#population></p><hr>
<p>Last updated on <#date><br>
HTML file produced by the program <#program>.</p>
</body></html>

```

通过使用带有连接的数据集字段名称的标记(常用的COUNTRY.DB数据库表格),程序自动取得当前记录字段的值并且自动替换它们。它的输出如图19.8所示:浏览器连接到作为HTTP服务器的HtmlProd范例,如我们稍后讨论的。在程序中与该组件相关的源代码里,并没有对数据库数据的引用:

```

procedure TFormProd.DataSetPageProducer1HTMLTag(Sender: TObject; Tag: TTag;
const TagString: String; TagParams: TStrings; var ReplaceText: String);

```

```

begin
  if TagString = 'program' then
    ReplaceText := ExtractFilename (Forms.Application.Exename)
  else if TagString = 'date' then
    ReplaceText := DateToStr (Date);
end;

```



图19.8 单击Print Line按钮后的HtmlProd程序的输出

生成HTML表格

HtmlProd范例的最后一个按钮是Print Table。该按钮与DataSetTableProducer组件相连，并且也调用了其Content函数，并将结果复制到Memo的Text属性中。通过将DataSetTableProducer的DataSet属性与ClientDataSet1相连，就生成一个标准的HTML数据表格。

实际上，该组件默认情况下只生成20行内容，这是由MaxRows属性设置的。如果想要获得数据表格的所有记录，可以将该属性设置为-1，这是一个简单但未公开的设置。

提示：DataSetTableProducer组件从当前记录启动，而不是从第一个记录开始。这意味着，第二次单击Print Table按钮时，将看到输出中没有记录。在调用生成器组件的Content对象方法之前添加一个对数据表First对象方法的调用可以解决该问题。

为了使该生成器组件的输出更为完整，可以执行两种不同的操作。首先，需要提供一些关于Header与Footer的信息，以生成HTML起始与结束元素，并向HTML数据表格添加一个标题（Caption）。其次，通过设置RowAttributes、TableAttributes与Columns属性来定制数据表格。各列的属性编辑器（也是默认组件编辑器），允许设置这些属性中的大部分，同时为输出提供了非常优秀的预览功能，如图19.9所示。在使用该编辑器之前，可以使用字段编辑器为数据表格的字段设置属性。例如，使用千位分隔符格式化人口与面积字段的输出。

有三种技术可以定制HTML数据表格，接下来我们将分别进行讨论：

- 使用数据表生成器组件的Column属性设置一些其他的属性，如标题的文本与颜色，或者是列中其他单元的颜色与对齐方式。

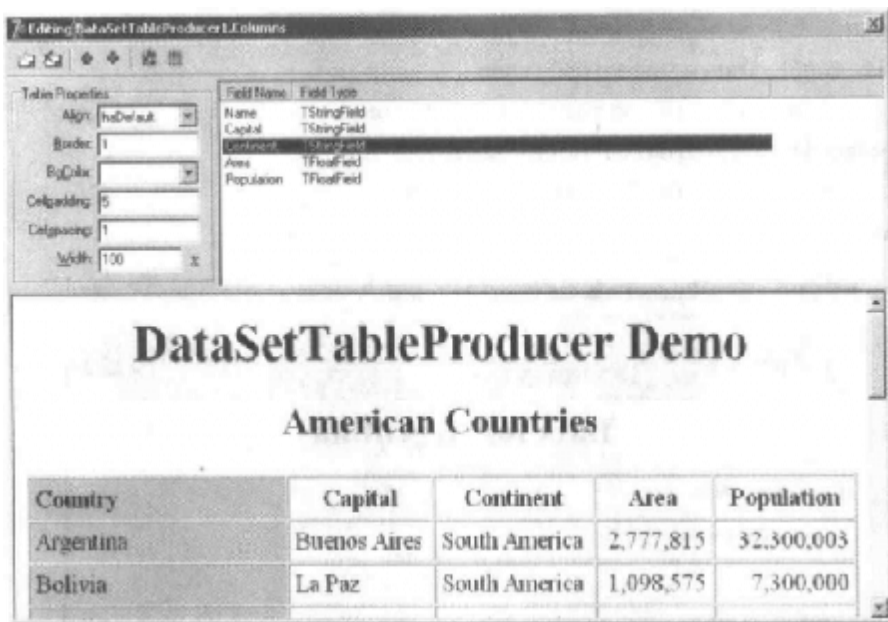


图19.9 DataSetTableProducer组件编辑器的列属性，为我们提供了一个最终的HTML表格的预览（如果数据库表格是激活的）

- 使用TField属性，特别是那些与输出有关的属性。在该范例中，ClientDataSet1Area字段对象的DisplayFormat属性已设置为“###, ###, ###”。这是用于确定每个字段实际输出的正确方法。我们还可以进一步将HTML标记嵌入到字段的输出中。
- 处理DataSetTableProducer组件的OnFormatCell事件来进一步定制输出。在该事件中，可以为一个给定的单元设置不同的列特征，而且还可以定制输出字符串（存储在CellData参数中）以及嵌入的HTML标记。但是不能通过使用Columns属性来完成这个操作。

在HtmlProd范例中，我们针对这个事件使用了一个处理程序，将Population和Area列的文本字体转变为粗体字，并且将较大数值的输出背景颜色设定为红色（除了标题行以外），下面是用到的代码：

```
procedure TFormProd.DataSetTableProducer1FormatCell(
  Sender: TObject; CellRow, CellColumn: Integer;
  var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if (CellRow > 0) and
    (((CellColumn = 3) and (Length (CellData) > 8)) or
     ((CellColumn = 4) and (Length (CellData) > 9))) then
  begin
    BgColor := 'red';
    CellData := '<b>' + CellData + '</b>';
  end;
end;
```

代码的剩余部分大都用于设定数据表生成器组件，其中包括它的Footer以及Header，正如我们打开范例的源代码后所看到的。

使用样式表

最新的HTML包括了一种功能非常强大的机制，用于将内容与表达方式分离开来：级联样式表（CSS）。使用样式表，可以将HTML的格式（颜色、字体、字体大小等等）与实际文本显示（页的内容）分离开。该方法可以使代码更灵活，使Web站点的更新操作更容易。而且，可以将站点上的图形任务（Web设计者的任务）与自动内容生成（程序员的任务）分离开。样式表是一种更为复杂的技术，可以将格式化值赋给主要类型的HTML部分与特殊的“类”（与OOP毫无关系）。详细内容见某种HTML参考资料。

如何在HtmlProd中更新数据表格的生成来包含样式表呢？非常简单，可以使用下面这行代码，在一个二级DataSetTableProducer组件的Header属性中提供一个对样式表的链接：

```
<link rel="stylesheet" type="text/css" href="test.css">
```

然后使用下面这行代码来更新OnFormatCell事件处理程序的代码（而不用两行代码来改变颜色与添加粗字体标记）：

```
CustomAttrs := 'class="highlight"';
```

以上提供的样式表（test.css，请参考范例的源代码）定义了一个强调样式，其中有粗字体与红色背景，并在第一个DataSetTableProducer组件的代码中进行了硬编码。

该方法的优点是，现在可以由一个美术家来改动CSS文件，让表格看起来更漂亮，而不用接触其代码。当想提供很多格式化元素时，使用样式表还可以减小HTML文件的总大小，这也是缩短下载时间的重要因素。

来自定制服务器的动态页

HtmlProd组件能用于生成静态HTML文件：如果使用在HttpServ示例中用到相似方法，它也可以兼做Web服务器，而且更符合现实的语境。事实上，程序要访问一个页生成器的请求，只需简单地通过请求传递组件的名字即可。下面是IdHTTPServer组件的OnCommandGet事件处理器的一部分，它使用FindComponent对象方法来定位正确的生成器组件：

```
var
  Req, Html: String;
  Comp: TComponent;
begin
  Req := RequestInfo.Document;
  if Req [1] = '/' then
    Req := Copy (Req, 2, 1000); // skip '/'
  Comp := FindComponent (Req);
  if (Req <> '') and Assigned (Comp) and
    (Comp is TCustomContentProducer) then
  begin
    ClientDataSet1.First;
    Html := TCustomContentProducer (Comp).Content;
```

```

    end;
    ResponseInfo.ContentText := Html;
  end;

```

在没有参数的情况下，或者参数是无效的，服务器使用一个可用组件的HTML菜单来进行响应：

```

Html := '<h1>HtmlProd Menu</h1><p><ul>';
for i := 0 to ComponentCount - 1 do
  if Components[i] is TCustomContentProducer then
    Html := Html + '<li><a href="/" + Components[i].Name + ".htm" +
      Components[i].Name + '</a></li>';
Html := Html + '</ul></p>';

```

最后，如果程序返回使用CSS的表格，浏览器将从服务器请求文件，因为这里添加了特殊代码来返回它。经过正确的生成步骤，该代码会演示服务器怎样通过返回结果来响应，以及怎样指出响应的MIME类型（ContentType）：

```

if Pos('test.css', Req) > 0 then
begin
  CssTest := TStringList.Create;
  try
    CssTest.LoadFromFile(ExtractFilePath(Application.ExeName) + 'test.css');
    ResponseInfo.ContentText := CssTest.Text;
    ResponseInfo.ContentType := 'text/css';
  finally
    CssTest.Free;
  end;
  Exit;
end;

```

小结

本章重点介绍了因特网的核心技术，包括套接字的使用和因特网的核心协议。我们讨论了主要的概念，并且提供了一些关于邮件与HTTP协议的使用示例。读者可通过更多的使用Indy组件的范例来进一步学习。

在介绍了互联网世界之后，我们准备研究两个主要领域：“现在”和“未来”。“现在”可通过Web应用程序的开发来代表，并且下一章将探讨动态站点开发技术，首先集中讨论旧的WebBroker技术，然后转移到新的WebSnap体系结构。在第21章中，我们将讨论IntraWeb。“未来”由Web服务和XML以及相关技术的使用来代表，将在第22章和第23章中进行讨论。

第20章 使用WebBroker和WebSnap进行Web编程

互联网在世界上的作用越来越大，这很大程度上取决于万维网（World Wide Web）的成功，而万维网是基于HTTP协议的。在第19章“互联网编程：套接字与Indy组件”中我们已经讨论过HTTP协议以及基于它的客户机/服务器端应用程序开发。由于已经有那些可供使用的高性能的、可伸缩和灵活的网络服务器，故我们可能很少想到过创建自己的服务器。事实上，动态网络服务器应用程序一般是通过集成脚本或在网络服务器内的编译程序建立的，而不是使用定制软件代替它们。

本章将集中讨论服务器端应用程序的开发，它将扩充现存的网络服务器。在上一章最后，我们介绍了HTML页的动态生成。现在让我们来看一下怎样在服务器内集成该动态生成脚本的功能。本章是上一章的合理延续，但不全部涉及互联网编程，因为第21章将专著于Delphi 7中的IntraWeb技术，而第22章将再次回到互联网编程的主体，并且从XML的角度来进行阐述。

警告：在本章为了测试一些示例，我们将需要访问一台Web服务器。最简单的解决方法是使用Microsoft的IIS版本或安装在计算机上的个人网络服务器（Personal Web Server）。然而，笔者个人更喜欢使用免费和开放源代码的Apache网络服务器，可从www.apache.org获得。因此，本书不会花费太多时间介绍网络服务器以及使用应用程序的配置细节，想了解相关知识可参考相关文档。

本章主要包括以下内容：

- 动态Web页面
- CGI、ISAPI和Apache模块
- WebBroker体系结构
- Web App调试器（Debugger）
- WebSnap体系结构
- 适配器（Adapters）以及服务器端脚本

动态Web页面

当浏览Web网站时，通常可以从Web服务器向自己的客户机下载静态页面——HTML格式的文本文件。作为Web开发人员，可以手工建立这些页面，但是为了使之更加商业化，从某些类型的数据库（SQL服务器、一系列文件等等）的信息建立静态页显得更有意义。通过这种方法，实际上是在使用HTML格式生成数据的快照；如果数据不是频繁改变的话，这是非常有意义的。该方法在第19章“因特网编程：套接字和Indy组件”中曾讨论过。

除了建立静态的HTML页面之外，还可以建立动态的HTML页面。为此，我们可以直接从数据库中抽取信息来响应浏览器的请求，因此由我们创建的应用程序发送的HTML会显示当前数据，而不是数据的旧快照。如果数据是不断变化的，该方法将会很有意义。

在前面曾提到，有几种方法可以用于编写定制Web服务器的程序；而且对于我们来说，它们是动态生成HTML页的理想方法。除了那些基于脚本的技术以外，编写Web服务器的两个最常见的协议是：通用网关接口，即CGI（Common Gateway Interface）以及Web服务器API。

说明：Delphi的WebBroker技术（包含在Delphi的企业版和专业版中）通过提供一个通用类框架消除了CGI与服务器API之间的区别。这样，我们可以轻松地将CGI应用程序转换成ISAPI库，或者将它集成到Apache之中。

CGI概述

CGI是客户浏览器与Web服务器之间进行通信的一种标准协议。它并不是一个非常有效的协议，但它的用途广泛而且是不限制操作平台的。该协议允许浏览器请求与发送数据，而且基于应用程序（通常是控制台应用程序）的标准命令行输入与输出。当服务器检测CGI应用程序的页面请求时，它会启动应用程序，从页面请求向应用程序传递命令行数据，然后从应用程序向客户发回标准的输出结果。

可供我们进行CGI应用程序编写的工具与语言很多，Delphi只是其中之一。除了要求自己的Web服务器必须是基于Intel的Windows或Linux系统这个限制外，我们可以在Delphi与Kylix中建立一些非常复杂的CGI程序。CGI是一种低级别的技术，因为CGI使用标准的命令行输入与输出，以及一些环境变量来完成从Web服务器接收和发送信息。

为了不使用任何支持类建立一个CGI程序，可以建立一个Delphi控制台应用程序，删除典型的项目源代码，用下列语句代替它：

```
program CgiDate;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  writeln ('content-type: text/html');
  writeln;
  writeln ('<html><head>');
  writeln ('<title>Time at this site</title>');
  writeln ('</head><body>');
  writeln ('<h1>Time at this site</h1>');
  writeln ('<hr>');
  writeln ('<h3>');
  writeln (FormatDateTime(' "Today is " dddd, mmmmm d, yyyy, ' +
    '"<br> and the time is" hh:mm:ss AM/PM', Now));
  writeln ('</h3>');
  writeln ('<hr>');
  writeln ('<i>Page generated by CgiDate.exe</i>');
  writeln ('</body></html>');
end.
```

CGI程序生成使用标准输出的、具有标题的HTML文本。如果我们直接执行该程序，会看到显示在终端窗口中的文本。如果从Web服务器上运行它，并向浏览器发送输出，会出现

格式化的HTML文本，如图20.1所示。



图20.1 CgiDate应用程序的输出，显示在一个浏览器中

使用普通的CGI来建立高级的应用程序需要我们完成很多的工作，例如，为了能够获得HTTP请求的状态信息，我们需要访问一些相关的环境变量，如下所示：

```
// get the pathname  
GetEnvironmentVariable ('PATH_INFO', PathName, sizeof (PathName));
```

使用动态库

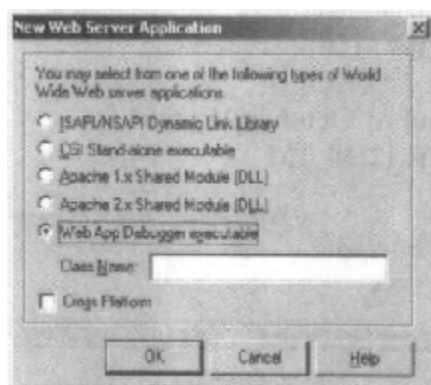
另一种与上述完全不同的方法是，使用Web服务器API，即流行的ISAPI（由微软提供的Internet Server API）、NSAPI（相对较少见的Netscape Server API）以及Apache API。可使用这些专有的API编写一个库文件，服务器会将这个库装载到自己的地址空间中，并且通常保留在内存里。在装入这个库之后，服务器可以通过主进程中的线程执行单个请求，而不必像在CGI应用程序中那样，必须为每个请求启动一个新的EXE。

当服务器接收到页面请求时，如果需要，服务器会装载DLL并执行相应的代码，这样做将可能启动一个新线程或使用一个已有的线程来处理页面请求。接下来，这个库代码会向请求页面的客户端发回相应的HTTP数据。因为该通信通常发生在内存中，所以这种应用程序的执行速度要比CGI快很多，而且给定的系统使用这种方法可以支持更多并发的页面请求。

Delphi的WebBroker技术

到目前为止，我们演示的CGI程序代码片断只解释了该协议普通、直接的用法。我们也可以提供基于ISAPI或者Apache模块的低层次范例。但是在Delphi中，更令人兴奋的是可以使用所谓的WebBroker技术。这是一种特殊的类层次结构，在VCL和CLX中建立，用于简化Web上的服务器端开发。它还是一种特殊类型的数据模块，被称做WebModules，Delphi的企业版与专业版中都包含有这个框架（不同于更高级和更新的WebSnap框架，它只能从Delphi的企业版中获得）。

使用WebBroker技术, 我们可以轻松地开发ISAPI、CGI应用程序或者Apache模块。在New Items对话框的第一个页面(New)上, 选择Web Server Application图标。随后出现的对话框会为用户提供几个选项, 包括ISAPI、CGI、Apache 1或Apache 2模块以及Web App Debugger。



在这种情况下, Delphi将生成一个带有WebModule的项目, WebModule是一个与数据模块非常相似的不可见容器。无论使用什么类型的项目, 这部分代码都是相同的, 只有主项目文件有所改变。对于一个CGI应用程序而言, 它将会如下所示:

```
program Project2;
{$APPTYPE CONSOLE}

uses
  WebBroker,
  CGIApp,
  Unit1 in 'Unit1.pas' (WebModule1: TWebModule);
{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TWebModule1, WebModule1);
  Application.Run;
end.
```

虽然这是一个CGI控制台程序, 但是其代码与标准的Delphi程序看起来颇为相似。然而, 它使用了一个技巧——该程序使用的Application对象不是TApplication类的典型全局对象, 而是一个新类的对象。取决于不同的Web项目, 这个Application对象或来自TCGIApplication类, 或属于派生自TWebApplication的其他类。

最重要的操作发生在WebModule中, 该组件派生自TCustomWebDispatcher, 它为所有的输入与输出提供了支持。事实上, TCustomWebDispatcher类定义了Request与Response属性, 这两个属性存储了客户请求与将要发回客户端的应答。而且它们都是使用一个抽象基类(TWebRequest与WebResponse)来定义的, 但是应用程序使用了一个指定对象(如TISAPIRequest与TISAPIResponse子类)来初始化它们。这些类实现了向服务器传递的所有信息, 因此我们可以使用一种简单的方法访问所有这些信息。对于应答信息也是一样的, 处理非常简单。该方法的优点是, 使用WebBroker编写的代码与应用程序的类型(CGI、ISAPI

或者Apache模块)无关。可以从一个类型转移到另外一个类型——通过更改或者转移项目文件,但是我们不需要更改一个WebModule中的代码。

这就是Delphi框架的体系结构。那么怎样编写应用程序的代码呢?我们可以使用WebModule中的Actions编辑器,根据请求的路径名来定义一系列操作(存储在Accons数组属性中),如下图所示。



该路径名是CGI或ISAPI应用程序URL的一部分,它位于程序名之后、其他参数之前,如下列URL中的path1:

`http://www.example.com/scripts/cgitest.exe/path1?param1=date`

通过提供不同的操作,应用程序可以使用不同的路径名响应请求,而且可以分配不同的制作器组件或者为每个可能的路径名调用不同的OnAction事件处理程序。当然,我们也可以忽略路径名来处理通用的请求。还需要考虑的是,如果我们不将应用程序基于WebModule,则可以使用普通的数据模块并向它添加WebDispatcher组件。如果需要将已有Delphi应用程序转换为Web服务器扩展,那么这是一个好方法。

警告: WebModule是从WebDispatcher基类继承而来的,因此不需要将它作为一个独立的组件。与WebSnap应用程序不同,WebBroker程序不能有多个分派者或者多个Web模块。另外还需要注意的是,WebDispatcher的操作与存储在ActionList或者ActionManager组件上的操作没有关系。

当我们定义相应的HTML页面(用它来触发应用程序)时,链接将为每个这样的路径向URL提出页面请求。拥有一个单独的库(可以根据参数执行不同的操作,在本例中是路径名)允许服务器在内存中保留一份该库的复制,并更快地响应用户请求。对于CGI应用程序来说,某些方面也是一样的:服务器必须运行多个实例,但可以缓存文件并使之运行速度更快。

OnAction事件是用来向特定请求发送特定应答的,需要向该事件处理程序传递两个主要参数,如下例所示:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content :=
    '<html><head><title>Hello Page</title></head><body>' +
    '<h1>Hello</h1>' +
    '<hr><p><i>Page generated by Marco</i></p></body></html>';
end;
```

Response参数的Content属性是输入HTML代码(想让用户看到)的地方。这段代码的惟一缺陷是,在浏览器中的输出会按多行显示文本,但查看HTML源代码,会发现整个字符串只用了一行。为了使HTML源代码更具可读性,需要将它分成多行,可以插入#13换行符(或者使用更好的跨平台sLineBreak值)。

为了使其他操作也可以处理该请求，应将最后一个参数Handled设置为False。否则，其默认值将是True，而且一旦使用自定义的操作处理了请求，WebModule会认为操作完成了。Web应用程序中的大部分代码位于WebModule容器定义的OnAction事件处理程序中。这些操作使用Request与Response参数从客户端接收请求，然后返回应答。

当我们使用生成器(Producer)组件时，OnAction事件通常会使用赋值操作将生成器组件的Content返回为Response.Content。我们还可以通过向操作本身的Producer属性分配一个制作器组件来简化该代码，而不需要再编写这些简单的事件处理程序(但是同时进行这两种操作将可能产生麻烦)。

提示：对于Producer属性来说，可以使用ProducerContent属性来替代它。该新属性允许链接定制的生产器类，该类不是从TCustomContentProducer继承而来的，但可实现IProduceContent接口。ProducerContent属性可以说是一个接口属性：它以同样的方式进行工作，但这是由它的属性编辑器决定的，而并非是基于Delphi对可接口属性提供新支持的缘故。

使用Web APP调试器进行调试

调试用Delphi编写的网络应用程序通常是十分困难的。事实上，我们不能简单地运行应用程序并在其上设置断点，但应相信网络服务器能在Delphi调试器内运行CGI程序或库。这能通过向Delphi的运行参数对话框中指定主机应用程序来完成。但这种做法暗示让Delphi运行网络服务器(它经常是一种Windows服务，而不是独立程序)。

为了解决所有这些问题，Borland公司添加了Delphi特有的Web App调试器程序。该工具通过Tools菜单的对应项被激活。它是一个网络服务器，在程序员设定的端口等待请求(默认情况下是端口1024)。当请求到达时，程序可以将它转发给独立的可执行程序，在Delphi 6中，这个通信过程是基于COM技术的，而在Delphi 7中，它是基于Indy套接字的。这意味着在这两种情况下，都能够从Delphi IDE内运行网络服务器应用程序，设置所有需要的断点。然后调试程序(当程序被Web App调试器激活时)，像为普通可执行文件所做的那样。

Web App调试器也能很好地记录所有接收的请求与返回到浏览器的实际响应。该程序也有一个统计页面，负责追踪响应每个请求的时间，允许我们测试不同情况下应用程序的效率。Delphi 7提供的Web App调试器中的另外一个新的特性是，它现在已经是一个CLX应用程序而不是一个VCL应用程序了。这个用户界面的改变以及从COM到套接字的转变都使得Web App调试器可以用于Kylix之中。

警告：因为Web App调试器使用Indy套接字，所以我们的应用程序将会频繁地收到EidConnClosedGracefully类型的异常。出于这个原因，这个异常在所有Delphi 7的对象中，都被自动的禁用了。

通过使用New Web Server Application对话框的对应选项，我们可以轻松地创建兼容调试器的新应用程序。这个选项定义了标准项目，用于产生主窗体和一个Web模块。这个(并没有实际作用的)窗体包括提供初始化和向Windows注册表添加应用程序的代码：

```
initialization
```

```
  TWebAppSockObjectFactory.Create('program_name');
```

Web App调试器使用该信息获得当前可用程序的列表。当为调试器使用默认URL时，它将完成这个操作，如图20.2所示。这个列表中包括所有注册服务器，而不只是那些正在运行的，同时可以用这个列表来激活某个程序。尽管这不是一个好办法，因为我们需要在Delphi

IDE内运行该程序，以期能够轻松地对其进行调试。注意，我们可以通过单击**View Details**来展开列表进行详细浏览，该列表包括实际可执行文件的列表和许多其他细节。

用于该类型项目的数据模块包括下列初始化代码：

```
uses WebReq;  
  
initialization  
  if WebRequestHandler <> nil then  
    WebRequestHandler.WebModuleClass := TWebModule2;
```

Web App调试器应用程序应该只用于调试。为了部署实际应用程序，应使用一个其他选项。程序员能做的是为网络服务器程序的其他类型创建项目文件，并且添加调试应用程序的相同网络模块项目。

反之则会显得比较复杂。为了调试现有的应用程序，程序员不得不创建该类型的应用程序，删除网络模块，添加现存的程序，并通过添加一行代码设置**WebRequestHandler**的**WebModuleClass**来对它进行修补，就像前面代码片段显示的那样。



图20.2 显示用Web App调试器注册的一列应用程序

警告：尽管在大多数情况下，我们可以将一个程序从一种Web技术迁移到另外一种，但是并不是所有的情况下都可以这样去做。例如，在CustQueP范例（稍后进行讨论）中，我们就不得不尽量避免使用请求的ScriptName属性（它是一个不错的CGI程序），而是使用InternalScriptName属性作为替代。

Web App调试器还提供了其他两个有趣的元素。第一个无需安装网络服务器和不用修改相应的设置，就可测试我们的应用程序。换句话说，展开程序来测试它们——简单地立即测试它们。另一个是，与先前使用CGI技术开发应用程序相反，能立即用多线程结构开始测试，而不用装载和卸载库，它们经常暗示关闭网络服务器甚至计算机。

建立多任务WebModule

为了解释如何方便地依靠Delphi的支持建立一个富于特性的服务器端应用程序，本章建立了一个BrokDemo范例。我们在这个范例中使用了**Web App**调试器技术，但是通过选择相应的项目文件，该范例可以被重编译为CGI或者Web服务器库。

WebBroker范例主要由一系列将用于支持该应用程序的操作组成，读者可以在Actions编辑器或者直接在Object Treeview内查看并管理它们。这些操作还可以直接在Web数据模块的设计器中看到，这样我们就可以通过图形化来了解数据库对象间的关系。如果读者研究一下范例的源代码，还会注意到，每个操作都有一个特定的名称。我们还给OnAction事件处理程序起了有含义的名称，例如，TimeAction作为对象方法的名称应该比Delphi自动生成的WebModule1WebActionItem1Action名称更容易理解。

每个操作都有不同的路径名；即使没有指定路径名，也可以将其中一个确定为默认操作，并执行它。该程序中第一个有趣的想法是将两个PageProducer组件分别用于每页的首尾部分，PageHead与PageTail。将代码集中在一起使得对它的改动更容易了，特别是如果它基于外部HTML文件的话。由这些组件生成的HTML将被添加在Web模块OnAfterDispatch事件处理程序产生的HTML代码的开始与结尾处：

```
procedure TWebModule1.WebModule1AfterDispatch(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageHead.Content + Response.Content + PageTail.Content;
end;
```

我们在页面生成的结尾部分添加初始与终止HTML，只是因为这样可以使组件生成HTML，并且好像一切工作都是由它们完成的。从OnBeforeDispatch事件中的某些HTML开始，意味着不能直接将生成器组件赋给那些操作，或者说生成器组件会覆盖在响应中已经提供的Content。

PageTail组件包括使用该脚本名的定制标记，可以用下列代码替代，它们使用网络模块中提供的当前请求对象：

```
procedure TWebModule1.PageTailHTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'script' then
    ReplaceText := Request.ScriptName;
end;
```

这段代码用于扩充PageTail组件中HTMLDoc属性的<#script>标记。

时间与日期操作的代码非常简单，真正有趣的部分从Menu路径开始，这是默认操作。在其OnAction事件处理程序中，应用程序使用一个for循环来建立所有可用操作的列表（使用它们的名字中不包括前两个字符的部分，在这个例子中是Wa）——通过锚标记（<a>）为每一项操作提供一个链接：

```
procedure TWebModule1.MenuAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  Response.Content := '<h3>Menu</h3><ul>' #13;
  for I := 0 to Actions.Count - 1 do
```

```

Response.Content := Response.Content + '<li> <a href="' +
Request.ScriptName + Action[I].PathInfo + '"> ' +
Copy (Action[I].Name, 3, 1000) + '</a>'#13;
Response.Content := Response.Content + '</ul>';
end;

```

BrokDemo范例的另一个行为向用户提供了一系列与请求有关的系统设置，它们对调试工作非常有用。这对于了解有多少信息、都是些什么信息，以及浏览器与Web服务器之间相互转换的HTTP协议也是非常有用的。为了生成该列表，程序要查找TWebRequest类的每个属性值，如下列初始代码段解释的那样：

```

procedure TwebModule1.StatusAction(Sender: TObject; Request: TWebRequest;
Response: TWebResponse; var Handled: Boolean);
var
I: Integer;
begin
Response.Content := '<h3>Status</h3>'#13 +
'Method: ' + Request.Method + '<br>'#13 +
'ProtocolVersion: ' + Request.ProtocolVersion + '<br>'#13 +
'URL: ' + Request.URL + '<br>'#13 +
'Query: ' + Request.Query + '<br>'#13 + ...

```

动态数据库报表

BrokDemo范例还定义了两个操作，由/table与/record路径名说明。对于最后的两个操作来说，我们的程序要生成一个名称主列表，然后显示一个记录的详细信息；为此，需要使用DataSetTableProducer组件格式化整个数据表格，使用DataSetPageProducer组件建立记录视图。下面是这两个组件的属性：

```

object DataSetTableProducer1: TDataSetTableProducer
DataSet = dataEmployee
OnFormatCell = DataSetTableProducer1FormatCell
end
object DataSetPage: TDataSetPageProducer
HTMLDoc.Strings = (
'<h3>Employee: <#LastName></h3>'
'<ul><li> Employee ID: <#EmpNo>'
'<li> Name: <#FirstName> <#LastName>'
'<li> Phone: <#PhoneExt>'
'<li> Hired On: <#HireDate>'
'<li> Salary: <#Salary></ul>')
OnHTMLTag = PageTailHTMLTag
DataSet = dataEmployee
end

```

为了生成整个表格，这里只是将DataSetTableProducer与对应行为的Producer属性相连，而没有提供任何专门的事件处理程序。通过向指定记录添加内部链接可以使数据表格的功能

更强。下面的代码将被数据表格的每个单元执行，但只是和第一列建立了链接，而非第一行（带有标题的行）：

```

procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject;
  CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
  var Align: THTMLAlign; var VAlign: THTMLVAlign;
  var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + ScriptName + '/record?LastName=' +
      dataEmployee['Last_Name'] + '&FirstName=' +
      dataEmployee['First_Name'] + '"> '
      + CellData + ' </a>';
end;

```

在图20.3中，读者可以看到该行为的结果。当用户选择一个链接时，会再次调用程序，并可以检查QueryFields字符串列表，从URL中读取参数。然后，它使用对应的数据表格的字段值进行记录查找（基于FindNearest调用）：

```

procedure TWebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  dataEmployee.Open;
  // go to the requested record
  dataEmployee.Locate ( 'LAST_NAME;FIRST_NAME',
    VarArrayOf([Request.QueryFields.Values[ 'LastName' ],
      Request.QueryFields.Values[ 'FirstName' ]]), [ ]);
  // get the output
  Response.Content := Response.Content + DataSetPage.Content;
end;

```

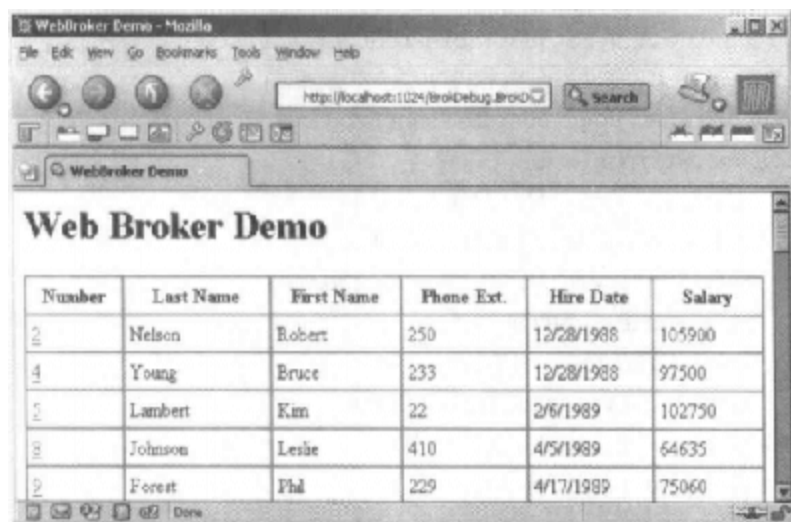


图20.3 对应于BrokDemo示例的表路径的输出，产生一个带有内部链接的HTML表

查询与窗体

前面的范例使用了一些本章前面介绍的HTML生成器组件。但还有一个组件我们没有介绍过，就是QueryTableProducer（用于BDE）以及它的姐妹成员：SQL QueryTableProducer（用于dbExpress）。稍后会看到，该组件使得复杂数据库程序的建立简单化了。

假如在一个数据库中查找一些顾客，就可以建立如下的HTML窗体（还可以嵌入到HTML数据表中，获得更好的格式）：

```
<h4>Customer QueryProducer Search Form</h4>
<form action="#script>/search" method="POST">
<table>
<tr><td>State:</td>
<td><input type="text" name="State"></td></tr>
<tr><td>Country:</td>
<td><input type="text" name="Country"></td></tr>
<tr><td></td>
<td><center><input type="Submit"></center></td></tr>
</table></form>
```

说明：在Delphi中，HTML窗体中有一系列控件（有些像输入字段）。有一些可视工具有助于程序员设计这些窗体，或手工输入相应的HTML代码。可以使用的控件包括按钮、输入文本（或编辑）框、选择（或组合）框、输入（或单选）按钮。可以将按钮定义为特殊类型，如Submit或Reset，它们意味着标准的操作。窗体的另一个重要元素是请求对象方法，它可以是POST（数据在幕后传递，在ContentFields属性中接收它）或GET（数据作为URL的一部分传递，从QueryFields属性中读取它）。

我们应该注意这个窗体中一个非常重要的元素：输入组件（州和国家）的名字，它应该与SQLQuery组件的参数相匹配：

```
SELECT Customer, State_Province, Country
FROM CUSTOMER
WHERE
    State_Province = :State OR Country = :Country
```

这段代码用在CustQueP（Customer Query Producer）范例中。为了建立它，我们在WebModule中放置了SQLQuery组件，而且为它生成了字段对象。同一个WebModule中还添加了与/search行为的Producer属性相连的QueryTableProducer组件。程序将生成相应的响应。当通过调用Content函数启动SQLQueryTableProducer组件时，它可以通过从HTTP请求中获得参数来初始化SQLQuery组件。该组件可以自动检测请求对象方法，然后使用QueryFields属性（如果请求是GET操作）或ContentFields属性（如果请求是POST操作）。

我们使用静态HTML窗体的一个问题是，它不能告诉我们可以查找的州与国家。为了解决该问题，可以在HTML窗体中，使用选择控件代替编辑控件。然而，如果用户向数据库表格添加新记录，还需要自动更新元素列表。最后一种方法是，设计ISAPI DLL，用以临时生成一个窗体，然后向选择控件填充可用的元素。

还应在/form操作中为该页面生成HTML代码，显然，该操作与PageProducer组件相连。PageProducer包含了下列HTML文本，其中有两个特殊标记：

```

<h4>Customer QueryProducer Search Form</h4>
<form action=' <#script> search' method="POST" >
<table>
<tr><td>State:</td>
    <td><select name='State' ><option></option><#State_Provinces></select></td></tr>
<tr><td>Country:</td>
    <td><select name='Country' ><option></option><#Country></select></td></tr>
<tr><td></td>
    <td><center><input type='Submit'></center></td></tr>
</table></form>

```

我们会注意到如同某些表格的字段一样，这些标记具有相同的名字。当PageProducer遇到这些标记的时候，它会为每一个相应字段的不同的数值添加一个<option> HTML标记。下面是OnTag事件处理程序的源代码，它是通用的可以被反复应用：

```

procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
    const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    ReplaceText := '';
    SQLQuery2.SQL.Clear;
    SQLQuery2.SQL.Add ('select distinct ' + TagString + ' from customer');
    try
        SQLQuery2.Open;
        try
            SQLQuery2.First;
            while not SQLQuery2.EOF do
                begin
                    ReplaceText := ReplaceText +
                        '<option>' + SQLQuery2.Fields[0].AsString + '</option>'#13;
                    SQLQuery2.Next;
                end;
            finally
                SQLQuery2.Close;
            end;
        except
            ReplaceText := '{wrong field: ' + TagString + '}';
        end;
    end;

```

这个方法使用了第二个SQLQuery组件，我们可以手工将这个组件放置在窗体中，并且与一个共享的SQLConnection组件进行连接。它将会产生如图20.4所示的输出。

这个Web服务器扩展和很多其他的一样，允许用户查看一个特定记录的详细信息。正如在前面的例子中那样，我们可以通过定制第一列（0列）的输出来完成这个任务，它是由QueryTableProducer组件生成的：

```

procedure TWebModule1.QueryTableProducer1FormatCell(
    Sender: TObject; CellRow, CellColumn: Integer;

```

```

var BgColor: THtmlBgColor; var Align: THtmlAlign;
var VAlign: THtmlVAlign; var CustomAttrs, CellData: String);
begin
  if (CellColumn = 0) and (CellRow <> 0) then
    CellData := '<a href="' + Request.ScriptName + '/record?Company=' + CellData +
      '">' + CellData + '</a>' #13;
  if CellData = '' then
    CellData := '&nbsp;';
end;

```

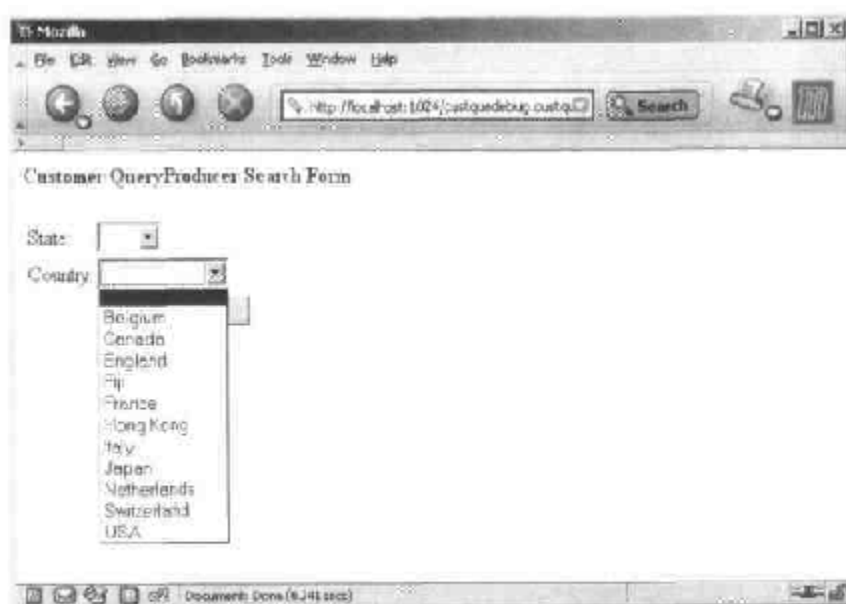


图20.4 CustQueP示例的窗体操作产生一个HTML窗体，它带有动态更新的选择组件，以反映数据库的当前状态

提示：当HTML表格中具有一个空单元（Cell）的时候，大多数浏览器都不会显示它的边框。因此，我们在每一个空单元中添加了一个非中断（Non-Breaking）空间字符（ ），而且需要在Delphi的表格生成器生成的每个表格中都进行这个操作。

对于这个连接的操作是/record，并且要在?参数之后（没有参数名称，因此不是非常标准的）传递一个特定的元素。为了显示该记录而生成HTML表格的这段代码并没有像以往一样使用了生成器组件，相反，它在一个定制表格中显示了每一个字段的数据：

```

procedure TwebModule1.RecordAction(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
var
  I: Integer;
begin
  if Request.QueryFields.Count = 0 then
    Response.Content := 'Record not found'
  else
    begin
      Query2.SQL.Clear;

```

```

Query2.SQL.Add ('select * from customer ' +
  where 'Company=' + Request.QueryFields.Values['Company'] + ' ');
Query2.Open;
Response.Content :=
  <html><head><title>Customer Record</title></head><body>#13 +
  <h1>Customer Record:  + Request.QueryFields[0] + </h1>#13 +
  <table border>#13;
  for I := 1 to Query2.FieldCount - 1 do
    Response.Content := Response.Content +
      <tr><td> + Query2.Fields[I].FieldName + </td>#13 <td> +
      Query2.Fields[I].AsString + </td></tr>#13;
  Response.Content := Response.Content + </table><h1>#13 +
  // pointer to the query form
  <a href=" + Request.ScriptName + '/form"> +
  ' Next Query </a>#13 + </body></html>#13;
end;
end;

```

使用Apache工作

如果计划使用Apache代替IIS或其他网络服务器，可以利用常见的CGI技术在任意网络服务器上展开应用程序。然而，使用CGI意味着将影响速度并且引起处理状态信息的麻烦（因为我们不能将数据保存在内存中）。因此，最好是编写ISAPI应用程序或动态Apache模块。使用Delphi的WebBroker技术，程序员也可容易地为两种技术编译相同的代码，使移动应用程序到不同的网络平台变得更容易。最后，也能用Kylix重编译CGI程序或动态Apache模块并在Linux上展开它。

如前面介绍的，Apache不但能运行传统的CGI应用程序，而且也有特定的技术将服务器扩展程序始终装载在内存中，以实现快速响应。为了用Delphi 6建立这样的应用程序，可简单使用New Web Server Application对话框的Apache Shared Module选项，根据我们打算使用的Web服务器的版本，选择Apache 1或者Apache 2。

警告：虽然Delphi 7支持Apache版本2.0.39，但是它不支持当前流行的版本2.0.40，这是因为一个库接口改变了。我们不建议使用版本2.0.39，因为它有安全漏洞。关于怎样修改VCL代码使之与2.0.40及以后版本的Apache相兼容的信息已经由Borland公司R&D成员在新闻组中发布了。现在可以在Bob Swart的Web站点上找到，URL如下：www.drrob42.com/delphi7/apache2040.htm。

如果选择创建一个Apache模块，那么将要使用一个库，其项目源代码的类型为：

```

library Apache1;

uses
  WebBroker,
  ApacheApp,
  ApacheWm in 'ApacheWm.pas' {WebModule1: TWebModule};

{$R *.res}

exports

```

```

    apache_module name 'apache1_module';

begin
    Application.Initialize;
    Application.CreateForm(TWebModule1, WebModule1);
    Application.Run;
end.

```

注意特殊的exports子句，该句暗示Apache配置文件会引用动态模块使用的名字。在项目源代码中，可以添加其他两个定义，模块名字和内容类型，如下所示：

```

ModuleName := 'Apache1_module';
ContentType:= 'Apache1-handler';

```

如果不设置它们，Delphi将赋给它们一些默认值，这些值，如_module和_handler字符串，将被添加到项目名字中，并以上面提到的名字结束。

Apache模块一般不在脚本文件夹中展开，但会在服务器本身的模块子文件夹中展开（默认情况下是c:\Program Files\Apache\modules）。然后程序员不得不编辑http.conf文件，并添加代码来装载该模块，如下所示：

```
LoadModule apache1_module modules/apache1.dll
```

最后，还必须指明该模块何时被激活。由该模块定义的处理器可与给定的文件扩展名（模块将只处理所有具有该文件扩展名的文件）相关联，或者与物理或虚拟文件夹相关联。在后面这种情况下，文件夹并不存在，但是Apache却假设它存在。下面的代码解释了如何为简单的Apache1模块设置虚拟文件夹的方法：

```
<Location /Apache1>SetHandler Apache1-handler</Location>
```

因为Apache是对大小写敏感的（因为它继承了Linux的属性），故可能需要添加第二个即小写的虚拟文件夹：

```
<Location /apache1>SetHandler Apache1-handler</Location>
```

现在我们能URL激活范例应用程序http://localhost/Apache1。在Apache使用虚拟文件夹的优点是用户不需要在站点的物理和动态端口间进行辨别，如我们在Apache1范例中看到的那样（其中包含了我们这里讨论的代码）。

实际范例

对使用WebBroker进行服务器端应用程序开发的核心概念做了一般性介绍后，可使用两个示例结束该部分。第一个是典型的网络计数器。第二个是扩展第19章中展示的WebFind程序来生成动态页，并以此代替填充列表框。

图形化的网页点击计数器

到目前为止，已建立的服务器端应用程序都只是基于文本的。当然，向已有图形文件添加引用是很容易的。然而，更重要的是建立可以生成图形（不断变化）的服务器端程序。

典型的范例有网页点击计数器。为了编写Web计数器，可将点击网页的当前数字保存在一个文件中，然后每当调用计数器程序时，读取并递增该值。如何返回该信息呢？如果所需的是带点击数值的HTML文本，那么代码就简单了：

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  nHit: Integer;
  LogFile: Text;
  LogFileName: string;
begin
  LogFileName := 'WebCont.log';
  System.Assign (LogFile, LogFileName);
  try
    // read if the file exists
    if FileExists (LogFileName) then
      begin
        Reset (LogFile);
        Readln (LogFile, nHit);
        Inc (nHit);
      end
    else
      nHit := 0;
    // saves the new data
    Rewrite (LogFile);
    Writeln (LogFile, nHit);
  finally
    Close (LogFile);
  end;
  Response.Content := IntToStr (nHit);
end;
```

警告：该简单的文件处理是不可伸缩的。当多个访问者同时点击同一网页时，该代码可以返回假结果或文件I/O错误，这是因为来自于其他线程中的请求已经打开了相应的文件，以便完成读操作，此时如果该线程试图打开该文件，就必须等待。为了支持相似的方案，需要使用一个互斥量（或者是每一个多线程程序中的关键部分）来让每个随后线程等待，直到当前使用文件的线程已完成它的任务。

更有意思的是，可建立能嵌入到任何HTML页中的图形计数器。建立图形化计数器有两种基本方法：预先为每个数字准备一幅位图，然后将它们合并到程序中；或者让程序绘制内存位图来生成想返回的图形。在WebCount程序中，我们选择了第二种方法。

基本上，可以建立一个控制内存位图的Image组件，使用TCanvas类的常用对象方法绘图。然后，将该位图附在TJpegImage对象上。通过JpegImage组件访问位图，可以将图像转换成JPEG格式。这时，能将JPEG数据保存到流中并返回它。正如我们看到的，步骤很多，但代码不复杂：

```

// create a bitmap in memory
Bitmap := TBitmap.Create;
try
  Bitmap.Width := 120;
  Bitmap.Height := 25;
  // draw the digits
  Bitmap.Canvas.Font.Name := 'Arial';
  Bitmap.Canvas.Font.Size := 14;
  Bitmap.Canvas.Font.Color := RGB (255, 127, 0);
  Bitmap.Canvas.Font.Style := [fsBold];
  Bitmap.Canvas.TextOut (1, 1, 'Hits: ' +
    FormatFloat ('###,###,###', Int (nHit)));
  // convert to JPEG and output
  Jpeg1 := TJpegImage.Create;
  try
    Jpeg1.CompressionQuality := 50;
    Jpeg1.Assign(Bitmap);
    Stream := TMemoryStream.Create;
    Jpeg1.SaveToStream (Stream);
    Stream.Position := 0;
    Response.ContentStream := Stream;
    Response.ContentType := 'image/jpeg';
    Response.SendResponse;
    // the response object will free the stream
  finally
    Jpeg1.Free;
  end;
finally
  Bitmap.Free;
end;

```

负责返回JPEG图像的语句有三条，前两条设置Response的ContentStream和ContentType属性，最后一条调用SendResponse。内容类型必须与浏览器公认的一种MIME类型相匹配，而且与这三条语句的顺序有关。在Response对象中，还有一个SendStream对象方法，但只有在发送使用调用的数据类型之后，才可以调用它。下图显示了这个程序的效果。



为了在一个页面中嵌入该程序，可以在HTML代码中加入下面的代码：

```

```

使用网络搜索引擎进行搜索

我们在第19章讨论的Indy HTTP客户组件可在Google网站接收搜索结果。现在将该示例稍稍扩展，把它变成服务器端应用程序。WebSearcher程序可用做CGI应用程序或可执行的Web App调试器，它具有两个操作：第一个操作简单返回每个搜索引擎得到的HTML，另一个操作用于填充客户数据集组件，然后链接到表格页生成器中，以生成最终的输出。下面是第二个操作的代码：

```

const
  strSearch = 'http://www.google.com/search?as_q=borland+delphi&num=100';

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  I: integer;
begin
  if not cds.Active then
    cds.CreateDataSet
  else
    cds.EmptyDataSet;
  for i := 0 to 5 do // how many pages?
  begin
    // get the data form the search site
    GrabHtml (strSearch + '&start=' + IntToStr (i*100));
    // scan it to fill the cds
    HtmlStringToCds;
  end;
  cds.First;
  // return producer content
  Response.Content := DataSetTableProducer1.Content;
end;

```

GrabHtml方法与WebFind范例相同。HtmlStringToCds方法与WebFind范例中的相应方法也很相似（它在列表中添加表项）。它通过如下调用来添加地址和相应的描述：

```
cds.InsertRecord ([0, strAddr, strText]);
```

ClientDataSet组件是由三个字段构成的：两个字符串外加一个计数器。这个额外的空字段是为了在表格生成器中包含额外的列所需要的。下列代码在单元格式化事件中填充了这个列，它同时也加入了超级链接：

```

procedure TWebModule1.DataSetTableProducer1FormatCell(Sender: TObject; CellRow,
  CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign;
  var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
begin
  if CellRow <> 0 then
    case CellColumn of

```



```

0: CellData := IntToStr (CellRow);
1: CellData := '<a href="' + CellData + '">' + SplitLong(CellData) + '</a>';
2: CellData := SplitLong (CellData);
end;
end;

```

SplitLong的调用被加入到输出文本的额外空间之中,用来避免表格中的列过大。因为浏览器不会自动地将这些文本切分为多行的格式,除非其中包含有空格或者其他的特殊字符。这个程序的结果是一个相对缓慢的应用程序(因为它必须转发多个HTTP请求),它产生的输出如图20.5所示。

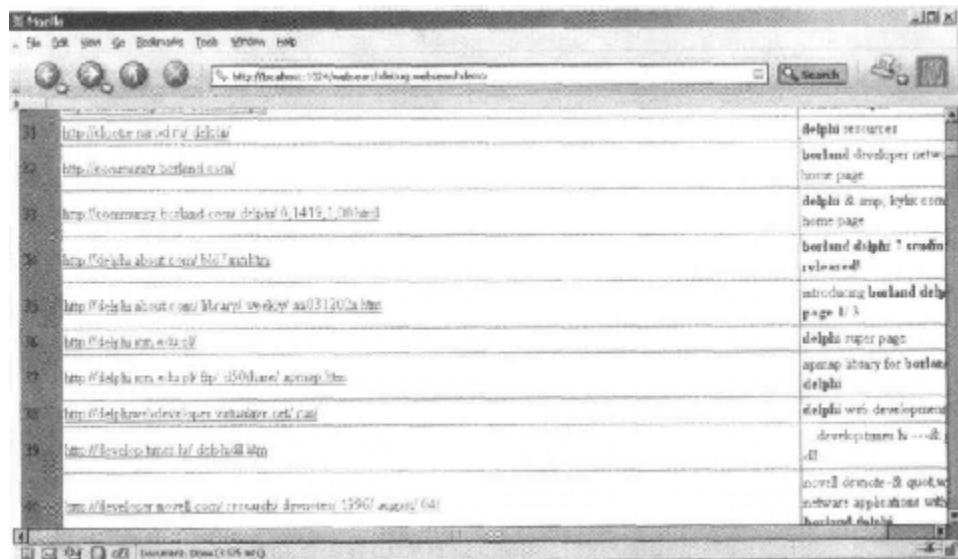


图20.5 WebSearch程序显示出Google上执行的多搜索的结果

WebSnap

介绍完使用Delphi进行网络服务器应用程序开发的核心元素后,我们开始集中介绍由Delphi 6的WebSnap引入的一些更复杂的体系结构。我们没有从本章开始就介绍该专题有两个原因。第一个原因是WebSnap建立在WebBroker的基础上,因此如果我们不知道核心概念就不能学习怎样使用新特性。例如,从技术上讲,WebSnap应用程序是CGI程序,或者是ISAPI或Apache模块。第二个原因是因为WebSnap只引入到了Delphi的企业版,所以不是所有的Delphi程序员都有机会使用它。

WebSnap与普通WebBroker相比有几个明确的优点,如允许用于多个Web模块,且每个模块与一个页相关联;以及集成服务器端脚本、XSL与Internet Express技术(后两个元素将在第22章“使用XML技术”中进行介绍)。而且还有许多备用组件用于处理普通任务,如用户登录、会话管理等等。与立即列出WebSnap的所有特征相反,笔者决定在随后的示例和重点应用程序中介绍它。出于测试的目的,所有这些应用程序都使用Web App调试器建立,但是在实际中我们可以使用下面任何一种其他的可用技术轻易地展开它。

当我们开发WebSnap应用程序时，应从一个对话框开始，通过该对话框我们既能在New Items对话框的WebSnap页激活它，也能在IDE的新互联网工具栏中使用它（默认情况时它是不可见的）。如图20.6显示的结果对话框，它允许用户选择应用程序的类型（像在WebBroker应用程序中那样）并定制初始应用程序组件（还可以在稍后添加更多）。对话框的底部决定第一页的操作，通常是程序的默认页或主页。一个相似的对话框也用于展示随后的页。

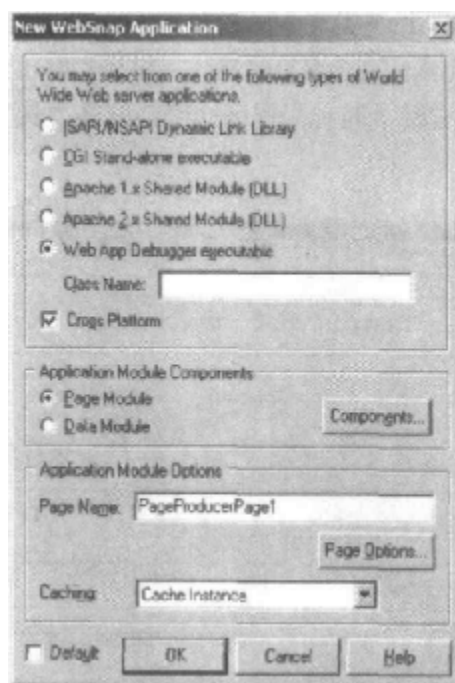


图20.6 New WebSnap Application对话框中的选项包括服务器类型和一个供选择核心应用程序组件的按钮

如果我们选择默认选项并且为这个主页输入一个名称，该对话框将创建一个项目，并打开TWebAppPageModule。默认情况下，该模块包括已选的组件：

- WebAppComponents组件是所有WebSnap应用程序集中服务的容器，如用户列表、核心调度程序、会话服务等。不是所有的属性都必须建立，因为应用程序也许不需要使用所有可以获得的服务。
- 这些核心服务中的一个是由PageDispatcher组件提供的，该组件用于包含所有应用程序的可用页面的列表，并定义默认情况下要使用的页面。
- 另一个核心服务由AdapterDispatcher组件提供，该组件负责处理HTML窗体提交和图像请求。
- ApplicationAdapter是我们遇到的适配器家族的第一个组件。这些组件可提供字段和程序评估的服务器端脚本操作。特别地，ApplicationAdapter是一个字段适配器，表示它自己的ApplicationTitle属性的值。如果我们为该属性输入一个数值，它将被脚本获得。
- 该模块支持PageProducer，它包括页的HTML代码，此例中为程序的默认页。与WebBroker应用程序不同，用于该组件的HTML不会被存储在HTMLDoc字符串列表属性内或被它的HTMLFile属性引用。HTML文件是一个外部文件，默认情况下存储

在宿主项目源代码的文件夹下，并使用类似于源包含的语句{*.html}从应用程序中引用。

- 因为PageProducer包括的HTML文件被作为独立文件保存（LocateFileService组件将帮助程序员进行开发），故它可进行编辑以改变程序页的输出，而不用重新编译应用程序。这些可能的变化不仅与HTML文件的固定部分有关，也与它的动态内容有关，这些都得益于对服务器端脚本的支持。基于标准模板的HTML默认文件已有一些脚本在里面。

警告：资源包含与HTML引用之间的相似之处基本上都是语法上的。HTML引用只能用于设计时的文件位置。但是资源包含可直接链接它引用到可执行文件中的数据。

通过选择对应的低层tab，HTML文件在有适当语法强调的Delphi编辑器内是可视的。这个编辑器同时也具有WebSnap模块的页面，默认包括的HTML结果页面——从中可以看到评估之后生成的HTML代码，以及一个与用户将要在浏览器中看到的一样的Preview页面。Delphi 7的WebSnap模块编辑器中包括了一个比Delphi 6所用的功能更强大的编辑器。它具有更好的提示语法以及自动完成代码的功能。如果用其他更高级的编辑器优先编辑网络应用程序中的HTML，我们就可在环境选项对话框的互联网页中进行选择。单击HTML扩展的Edit按钮，然后从编辑器的快捷菜单中或者是利用Delphi的Internet工具栏上的按钮，选择一个外部编辑器。

用于WebSnap的标准HTML模板会在程序中的所有页面上加入它的标题与应用程序的标题——使用类似下面的代码：

```
<h1><%= Application.Title %></h1>
<h2><%= Page.Title %></h2>
```

我们将复习一下脚本，并在下一节中开始建立WSnap1范例，这是一个具有多页面的应用程序。首先，我们将给出这个范例Web页面的源代码来结束这一部分的介绍：

```
type
  Thome = class(TWebAppPageModule)
    ...
  end;

function home: Thome;

implementation
  {$R *.dfm} {*.html}
  uses WebReq, WebCntxt, WebFact, Variants;

  function home: Thome;

begin
  Result := Thome(WebContext.FindModuleClass(Thome));
end;

initialization
  if WebRequestHandler <> nil then
    WebRequestHandler.AddWebModuleFactory(TWebAppPageModuleFactory.Create(
```

```
Thome, TWebPageInfo.Create([wpPublished {, wpLoginRequired}], '.html'),  
    caCache));  
  
end.
```

该模块使用全局函数代替窗体的典型全局对象以支持页缓存。该Web App调试器应用程序在初始部分也有一些额外代码，特别是一些注册代码让应用程序知道页面的作用和它的行为。

提示：与本章中的WebBroker范例不同，WebSnap范例在它的每一个文件夹中进行编译。这是因为HTML文件是在运行时需要的，并且我们更偏向于尽可能简化开发的过程。

管理多个页面

WebSnap和WebBroker之间第一个显著的区别是，用连接到生成器组件的多个操作代替单个数据模块，WebSnap有多个数据模块，每个都对应一个操作并且有一个使用HTML文件的生成器组件连到其上。实际上，程序员仍能添加多个操作到页面/模块中，但这个思想应是围绕着页面来建立应用程序而不是围绕着各种操作。与操作一样，页面的名字显示在请求路径上。

作为一个范例，我们向WebSnap程序（它是使用默认设置建立的）中又添加了两页。首先针对第一页，在New WebSnap Page Module对话框（如图20.7所示）中，选择标准页生成器并给它起名为date。其次，选择一个DataSetPageProducer并给它起名为country。保存文件后，就可以开始测试应用程序。稍后将讨论一些脚本，每页列出所有可用页面（除非在New WebSnap Page Module对话框中取消了对Published复选框的选择）。

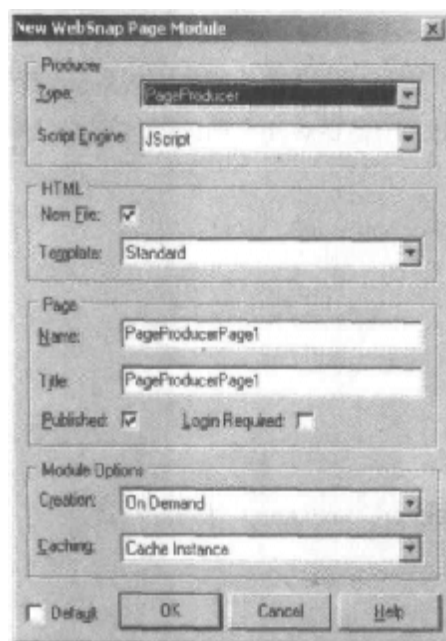


图20.7 New WebSnap Page Module对话框

这些页面可能还是空的，但是至少我们已经将结构部署就位。为了能够完成这个主页，我们将直接编辑它所连接的HTML文件。对于日期页，可以把它当做一个WebBroker应用程序而使用同样的办法。加入一些定制的标记到HTML文本中，如下面的例子所示：

```
<p>The time at this site is <#time>.</p>
```

我们还在生成器组件的OnTag事件处理程序中添加了代码，用当前时间来替换这个标记。

对于第三个页面（country页面），更改HTML，以包括country表格中不同字段的标记，如下所示：

```
<h3>Country: <#name></h3>
```

然后将ClientDataSet附加给页面生成器：

```
object DataSetPageProducer: TDataSetPageProducer
    DataSet = cdsCountry
end
object cdsCountry: TClientDataSet
    FileName = 'C:\Program Files\Common Files\Borland
        Shared\Data\country.cds'
end
```

为了在创建这个页面的时候以及重置它到第一个记录的时候打开这个数据集合，我们可以处理Web页面模块的OnBeforeDispatchPage事件，在其中添加下面的代码：

```
cdsCountry.Open;
cdsCountry.First;
```

事实上，WebSnap页面十分类似于WebBroker应用程序的一部分（基本上来说，是一个与生成器关联的操作），它十分重要，尤其是将已存在的WebBroker代码转化成新架构时。我们甚至也可将现存的DataSetTableProducer组件转换为新结构。从技术上来讲，可以生成新的页面，删除它的生成器组件，用DataSetTableProducer代替它并连接该组件到网页模块的PageProducer属性中。但在实际中，该方法会丢失页面的HTML文件和它的脚本。

在WSnap1程序中，我们使用了更好的技术。添加定制标记（<#htmltable>）到HTML文件中，并使用页生成器的OnTag事件向数据集合表格的HTML结果中进行添加：

```
if TagString = 'htmltable' then
    ReplaceText := DataSetTableProducer1.Content;
```

服务器端脚本

如果在服务器端程序中有多个页，且每页都伴随一个不同的页模块，那么也可以改变编写程序的方式，用手边的服务器端脚本提供一个更强大的方法。例如，WSnap1示例的标准脚本说明了应用程序和页的标题及页的索引。这个索引是由一个计数器产生，该技术用于从WebSnap脚本代码内详细查看列表，如下所示：

```
<table cellpadding="0" cellspacing="0"><tr>
<% e = new Enumerator(Pages)
    s = ''
    c = 0
    for (; !e.atEnd(); e.moveNext())
    {
        if (e.item().Published)
```

```

        {
            if (c > 0) s += '&nbsp;';
            if (Page.Name != e.item().Name)
                s += '<a href=' + e.item().HREF + '>' + e.item().Title + '</a>';
            else
                s += e.item().Title;
            C++;
        }
    }
    if (c>1) Response.Write(s);
}
</td></table>

```

说明：从技术上讲，WebSnap脚本是用JavaScript编写的，这是一种基于对象的语言，非常类似互联网编程，因为它通常只在浏览器上（客户端）可用。技术上，JavaScript作为ECMAScript说明，借助的是C语言的核心语法而与Java并没有什么关系。实际上，WebSnap使用Microsoft的ActiveScripting引擎，该引擎支持JScript（一个JavaScript的变体）和VBScript。

在该表格的单个单元内（奇怪的是，这个表格并没有行），脚本使用Reponse.Write命令输出了字符串。而该字符串可在多个页面间使用for循环产生并存储在Pages全局实体中。只有在公布该页和使用不是指向当前页的超级链接时，每个页的标题才被添加到字符串中。在脚本中使用该代码代替Delphi组件的硬件代码，可允许我们将它传递到一个好的Web设计器中，以更加可视化的方式对其进行修改。

提示：为了公布或不公布某页，不要查找网页模块中的属性。该状况被AddWebModuleFactory对象方法的标志控制，该方法由网页模块的初始化代码调用。读者可简单地设置或不设置该标志，以获得所需的效果。

作为示例，我已对WSnap2示例（WSnap1示例的扩展）添加了演示脚本页。该页的脚本能利用下列脚本代码生成多值的表格（输出如图20.8所示）：

```

<table border=1 cellspacing=0>
<tr>
    <th>&nbsp;</th>
    <% for (j=1;j<=5;j++) { %>
    <th>Column <%=j %></th>
    <% } %>
</tr>
<% for (i=1;i<=5;i++) { %>
<tr>
    <td>Line <%=i %></td>
    <% for (j=1;j<=5;j++) { %>
    <td>Value= <%=i*j %></td>
    <% } %>
</tr>
<% } %>
</table>

```

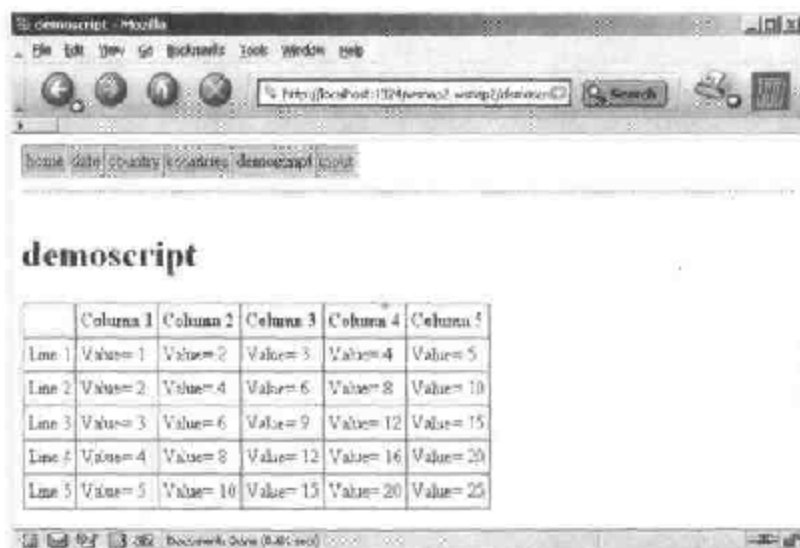


图20.8 WSnap2示例显示了存储在包含文件中的一个普通脚本和一个定制菜单

在这段脚本中，“<%=”符号代替了更长的Response.Write命令。服务器端脚本的另一个重要的特性是在页面中包含其他的页面。例如，如果打算对菜单进行改变，则可以在一个单独的文件中包含相关的HTML语句以及脚本，而不必在多个页面中改变和维护它。文件包含通常是通过下面的代码来实现的：

```
<!-- #include file="menu.html" -->
```

在程序清单20.1中，大家能够发现该菜单完整的包含文件源代码，它可由这个项目中所有其他的HTML文件所引用。图20.9显示了这个菜单的一个例子，它使用前面提到的表格生成脚本，在页面的顶端进行显示。

程序清单20.1 menu.html文件被包含在WSnap2示例的每一页中

```
<html>
<head>
<title><%= Page.Title %></title>
</head>
<body>
<h2><%= Application.Title %></h2>
<table cellpadding="0" cellspacing="2" border="1" bgcolor="#c0c0c0">
<tr>
<% e = new Enumerator(Pages)
for (; !e.atEnd(); e.MoveNext())
{
if (e.item().Published)
{
if (Page.Name != e.item().Name)
Response.Write ('<td><a href="' + e.item().Href + '">' +
e.item().Title + '</a></td>')
else
```

```

        Response.Write ('<td>' + e.item().Title + '</td>')
    }
}
%>
</tr>
</table>
<hr>
<h1><% Page.Title %></h1>
<p>

```

该菜单脚本使用Pages列表以及Page与AppLication全局脚本对象。WebSnap还提供了其他几个全局对象，包括EndUser和Session对象（这时向应用程序添加adapter）、Modules对象以及Producer对象——允许访问网页模块的Producer组件。该脚本也拥有Web模块的可用的Response和Request对象。

适配器

除了这些全局对象外，在脚本内还可访问所有在对应网页模块中可用的适配器（其他模块的适配器包含共享网络数据模块，它们必须应用Module对象和对应模块的前缀名字）。即适配器允许将消息从编译的Delphi代码传递给经过解释的脚本——通过提供Delphi应用程序的脚本界面来完成。适配器允许字段展示数据并支持显示命令的操作。服务器端脚本可通过传递参数访问这些值并发出这些命令。

适配器字段

对于简单的定制，可简单添加新字段到指定适配器。例如，在WSnap2示例中，添加定制字段到应用程序适配器。挑选完该组件后，或者能打开字段编辑器（可通过本地菜单获得），或者能在Object TreeView内简单工作。添加新字段后（在本示例中叫做Count），使能在OnGetValue事件上对它赋值。因为想计算网页上的点击（或请求）次数，所以处理了全局PageDispatcher组件的OnBeforePageDispatch事件，以增加本地字段HitCount的数值。下面是两种对象方法的代码：

```

procedure Thome.PageDispatcherBeforeDispatchPage(Sender: TObject;
    const PageName: String; var Handled: Boolean);
begin
    Inc (HitCount);
end;

procedure Thome.AppHitCountGetValue(Sender: TObject; var Value: Variant);
begin
    Value := HitCount;
end;

```

当然，可以使用页面名称计算每个指定页面的点击次数（因为每次运行程序的新例程时计数都会被重设，所以添加了一些对持续性功能的支持）。既然对现存适配器添加了定制字段（对应Application脚本对象），就能从任意脚本中访问它，如下所示：


```
<p>Application hits since last activation:
<%= Application.AppHitCount.Value %></p>
```

适配器组件

同样也能添加定制适配器到指定页。如果需要传递多个字段，可使用一般的适配器组件。其他的定制适配器（除了已使用的全局ApplicationAdapter）包括：

- PagedAdapter组件，内部支持在多个页上显示内容。
- DataSetAdapter组件，用于从脚本访问Delphi数据集合，我们将在下一节“WebSnap和数据库”中介绍。
- StringValuesList组件，用于存储一个“名/值”列表，如字符串列表，并且能直接使用或为适配器字段提供列表值。继承而来的DataSetValuesList适配器有相同的作用，但它是从数据集合上来获取名/值列表的，可为查找和其他选择提供支持。
- 用户相关的适配器如EndUser、EndUserSession和LoginForm适配器，用于访问用户信息和会话信息，并为应用程序建立注册窗体，自动连接到用户列表。在本章稍后的“会话、用户和许可”一节将做详细介绍。

使用AdapterPageProducer

这些组件中的大多数都将与AdapterPageProducer组件共同使用。事实上，在我们可视化地设计好想要的结果后，AdapterPageProducer就能生成部分脚本。作为一个示例，我们向WSnap2应用程序添加了inout页面，该页面有一个含两字段的适配器，一个是标准型的，另一个是布尔型的：

```
object Adapter1: TAdapter
  OnBeforeExecuteAction = Adapter1BeforeExecuteAction
  object TAdapterActions
    object AddPlus: TAdapterAction
      OnExecute = AddPlusExecute
    end
    object Post: TAdapterAction
      OnExecute = PostExecute
    end
  end
  object TAdapterFields
    object Text: TAdapterField
      OnGetValue = TextGetValue
    end
    object Auto: TAdapterBooleanField
      OnGetValue = AutoGetValue
    end
  end
end
end
```

该适配器还具有一些操作，用于发送当前的用户输入并且添加一个加号到文本中。当Auto字段有效时，也将添加一个相同的加号。使用普通HTML为该窗体开发用户输入和相关字符串将占用一些时间。但AdapterPageProducer组件（在这个页面中使用的）有一个集成的HTML设计器，该设计器被Borland公司叫做Web Surface Designer。使用该工具，可以可视化地添加窗体到HTML页并为其添加AdapterFieldGroup。连接该字段组到适配器可以使编辑器自动显示这两个字段。然后，我们可以添加AdapterCommandGroup，并连接它到AdapterFieldGroup，为适配器的所有操作都提供按钮。图20.9给出了该设计器的示例。

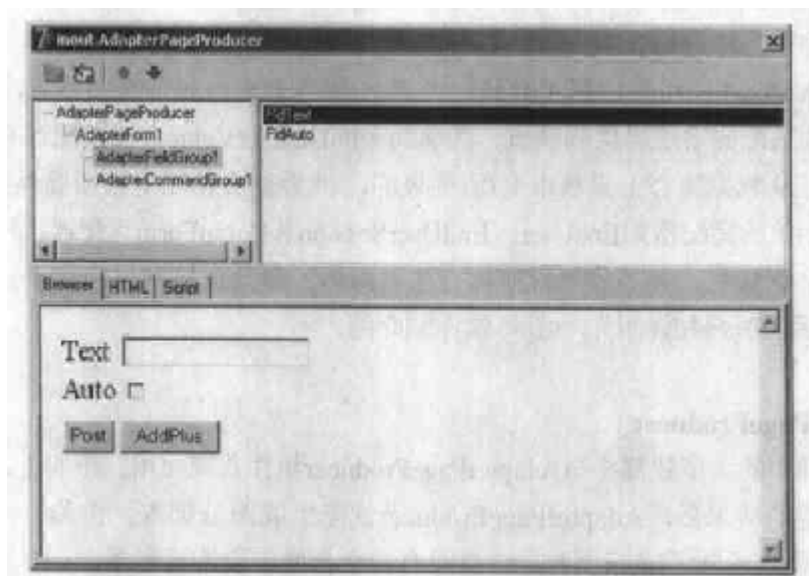


图20.9 在设计WSnap2示例中的输入页时Web Surface Designer的屏幕显示

更精确地说，如果字段组和命令组的AddDefaultFields和AddDefaultCommands属性已经被设定好，那么字段与按钮将自动显示。这些建立窗体的可视操作的效果总结在下面的DFM代码片断中：

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterFieldGroup1: TAdapterFieldGroup
      Adapter = Adapter1
      object FldText: TAdapterDisplayField
        FieldName = 'Text'
      end
      object FldAuto: TAdapterDisplayField
        FieldName = 'Auto'
      end
    end
    object AdapterCommandGroup1: TAdapterCommandGroup
      DisplayComponent = AdapterFieldGroup1
      object CmdPost: TAdapterActionButton
        ActionName = 'Post'
      end
    end
  end
end
```

```

object CmdAddPlus: TAdapterActionButton
    ActionName = 'AddPlus'
end
end
end
end
end

```

现在可以使用有脚本的HTML页来前后移动数据并发出命令，下面我们来看一下可完成这一工作的源代码。首先，要添加该类的两个本地字段，以存储适配器字段并管理它们；并且通过返回字段值，为它们实现OnGetValue事件。当每个按钮被按时，便检索用户传递的文本，该文本不会自动复制到对应的适配器字段中。通过观看这些字段的ActionValue属性就可了解它们的效果，该属性只在输入时设置（出于这个原因，如果没有任何输入，我们就将布尔字段的值设为False）。为了避免在两种操作中出现重复代码，可在网页模块的OnBeforeExecuteAction事件中放置它：

```

procedure Tinout.Adapter1BeforeExecuteAction(Sender, Action: TObject;
    Params: TStrings; var Handled: Boolean);
begin
    if Assigned (Text.ActionValue) then
        fText := Text.ActionValue.Values [0];
        fAuto := Assigned (Auto.ActionValue);
    end;
end;

```

请注意每一个操作可能具有多个数值（在组件允许多个选择的情况下），但是这还不是问题所在，因此我们能够抓住第一个元素。最后，下面是这两个操作的OnExecute事件的代码：

```

procedure Tinout.AddPlusExecute(Sender: TObject; Params: TStrings);
begin
    fText := fText + '+';
end;

procedure Tinout.PostExecute(Sender: TObject; Params: TStrings);
begin
    if fAuto then
        AddPlusExecute (Self, nil);
    end;
end;

```

作为一种选择，适配器字段有公共的EchoActionFieldValue属性，通过该属性可以获取用户输入值并再次在结果窗体中放置它。从技术上来说，该技术用于出错的情况下，以便让用户改变开始的输入值。

说明：AdapterPageProducer组件对级联样式单（CSS）有特殊支持。可以使用StylesFile属性或者Styles字符串列表为页面定义CSS。这时生成器项的编辑器的任意元素都可以定义特殊样式，或选择一个附加CSS的样式。使用StyleRule属性可完成后一种操作。

脚本还是代码

尽管这个使用适配器和适配器页简单组合的示例显示了该结构的功能，但该方法也有一个大的缺陷。通过让组件生成脚本（在HTML中只有<#SERVERSCRIPT>标记），节省了许多开发时间，但同时产生了混合代码，这样就需要在更新用户界面的同时更新程序。Delphi应用程序开发者和HTML/脚本设计者间责任的分工不明显了。相反运行脚本来完成应由Delphi程序立即做的事情可能会更快！

因此笔者的观点是，WebSnap拥有一个十分强大的结构，并且比WebBroker有了很大的进步，但使用它时也应该小心。应避免误用这些技术，因为这些技术是简单而强大的。例如，值得使用AdapterPageProducer的设计器来生成网页的第一个版本，然后获取生成的脚本并复制到普通PageProducer的HTML代码中，以便网络设计者能用指定工具修改该脚本。

对于非平凡的应用程序，可试图用XML和XSL提供的功能，即便它们的作用并不重要，但能够在该结构内使用。我们将在第22章对其进行详细介绍。

定位文件

当我们编写类似上述应用程序的时候，必须将它实现为一个CGI或者动态库。我们可以分派一个子文件夹或者定制的文件夹来存放所有的文件，而不是将模板与包含文件与可执行文件放在一个文件夹内。LocateFileService组件可处理这项任务。

这个组件不是自然而然即可使用的。当需要定位一个文件的时候，系统会触发这个组件的某个事件（这是一个功能很强大的方法），而不是将目标文件夹作为一个属性来设定。有三个事件：OnFindIncludeFile、OnFindStream和OnFindTemplateFile。第一个和最后一个事件在一个var参数中返回所用文件的文件名。OnFindStream事件允许我们直接提供一个流，亦即使用内存中的流，或者是利用匆忙创建、从数据库中提取、通过HTTP连接获得等手段提供流。在OnFindIncludeFile事件最简单的情况下，我们可以编写如下代码：

```
procedure TPageProducerPage2.LocateFileService1FindIncludeFile(  
  ASender: TObject; AComponent: TComponent; const AFileName: String;  
  var AFoundFile: String; var AHandled: Boolean);  
begin  
  AFoundFile := DefaultFolder + AFileName;  
  AHandled := True;  
end;
```

WebSnap和数据库

Delphi经常发挥作用的地方是数据库编程。出于该原因，可以看到许多支持在WebSnap框架内处理数据集合的功能。特别是能使用DataSetAdapter组件连接到一个数据集合并在窗体中或使用AdapterPageProducer组件的可视编辑器的表格上显示它的值。

WebSnap数据模块

为了给大家举例，我建立了一个新的WebSnap应用程序（叫做WSnapTable），使用AdapterPageProducer作为它的主页，用于在网格中展示表格；并在二级页面上使用另一个

AdapterPageProducer来展示有单个记录的窗体。这里也对应用程序添加了WebSnap数据模块作为数据集组件的容器。该数据模块有一个ClientDataSet，它通过供应器被绑定到dbExpress数据集上，并且基于InterBase连接显示如下：

```
object ClientDataSet1: TClientDataSet
    Active = True
    ProviderName = 'DataSetProvider1'
end
object SQLConnection1: TSQLConnection
    Connected = True
    ConnectionName = 'IBLocal'
    LoginPrompt = False
end
object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText =
        'select CUST_NO, CUSTOMER, ADDRESS_LINE1, CITY, STATE_PROVINCE, ' +
        ' COUNTRY from CUSTOMER'
end
object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
end
```

DataSetAdapter

既然有一个可用的数据集，就能添加DataSetAdapter到第一页，并连接它到网络模块的ClientDataSet上。适配器会自动地使得数据集的所有字段可用，并且使几个针对数据集的预定义操作（如Delete、Edit和Apply）有用。此时可以显式地添加它们到Actions和Fields集合，以排斥它们中的一些并定制它们的行为，但这么做并不总是必需的。

像PagedAdapter一样，DataSetAdapter也有PageSize属性，可以在该属性上说明显示在每页上的元素的数量。也有能用于在页中导航的组件。当我们在网格中显示大型数据集时，该方法特别适合。下面是用于WSnapTable示例主页的适配器设置：

```
object DataSetAdapter1: TDataSetAdapter
    DataSet = WebDataModule1.ClientDataSet1
    PageSize = 6
end
```

对应的页生成器有一个包含两个命令组和一个网格的窗体。第一个命令组（在网格上显示）有用于处理页的预定义命令：CmdPrevPage、CmdNextPage和CmdGotoPage 其中最后一条命令会生成页列表编号，因此用户能直接跳过它们。AdapterGrid组件提供了默认栏及一个额外支持的Edit和Delete命令。底部命令组有一个按钮用来创建新记录。读者能在图20.10看到表格输出示例，在程序清单20.2看到AdapterPageProducer的完整设置。

程序清单20.2 WSnapTable主页的AdapterPageProducer设置

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterGrid1
object CmdPrevPage: TAdapterActionButton
    ActionName = 'PrevPage'
    Caption = 'Previous Page'
end
object CmdGotoPage: TAdapterActionButton...
object CmdNextPage: TAdapterActionButton
    ActionName = 'NextPage'
    Caption = 'Next Page'
end
end
object AdapterGrid1: TAdapterGrid
    TableAttributes.CellSpacing = 0
    TableAttributes.CellPadding = 3
    Adapter = DataSetAdapter1
    AdapterMode = 'Browse'
object ColCUST_NO: TAdapterDisplayColumn
    ...
object AdapterCommandColumn1: TAdapterCommandColumn
    Caption = 'COMMANDS'
object CmdEditRow: TAdapterActionButton
    ActionName = 'EditRow'
    Caption = 'Edit'
    PageName = 'formview'
    DisplayType = ctAnchor
end
object CmdDeleteRow: TAdapterActionButton
    ActionName = 'DeleteRow'
    Caption = 'Delete'
    DisplayType = ctAnchor
end
end
end
object AdapterCommandGroup2: TAdapterCommandGroup
    DisplayComponent = AdapterGrid1
object CmdNewRow: TAdapterActionButton
    ActionName = 'NewRow'
    Caption = 'New'
    PageName = 'formview'
end
end
end
end

```



图20.10 WSnapTable示例在启动时显示的页包括分页表的初始部分

在该列表中有几点需要注意的。首先，网格的AdapterMode属性被设置为Browse，其他可能的选项包括Edit、Insert和Query。该适配器的数据集显示模型决定着用户界面的类型（文本或编辑框和其他输入控件）以及其他按钮的可视性（例如，Apply和Cancel按钮只在编辑浏览中显示，对应编辑命令）。

说明：使用服务器端脚本和访问Adapter.Mode，可改变适配器模型。

第二，在网格内修改了命令的显式方式，亦即使用DisplayType属性的ctAnchor值代替了默认按钮样式。类似的属性在该结构的多数组件中均可用，以便使用它们生成的HTML代码。

在窗体中编辑数据

最后一些命令被连接到不同页面上，即在命令使用后将其显示的页面上。例如，Edit命令的PageName属性被设置为formview。应用程序的第二页含有AdapterPageProducer，其组件连接到其他表格的相同DataSetAdapter上，因此所有请求将自动同步。事实上，当我们选择Edit命令时，该程序将打开第二页并显示对应于该命令的记录数据。

程序清单20.3详细显示了该程序第二页的页生成器。我们重申一下，使用Delphi特定设计器（如图20.11所示）建立可视HTML窗体是十分快捷的操作。

程序清单20.3 Formview页的AdapterPageProducer设置

```
object AdapterPageProducer: TAdapterPageProducer
  object AdapterForm1: TAdapterForm
    object AdapterErrorList1: TAdapterErrorList
      Adapter = table.DataSetAdapter1
    end
  end
```

```

object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
object CmdApply: TAdapterActionButton
    ActionName = 'Apply'
    PageName = 'table'
end
object CmdCancel: TAdapterActionButton
    ActionName = 'Cancel'
    PageName = 'table'
end
object CmdDeleteRow: TAdapterActionButton
    ActionName = 'DeleteRow'
    Caption = 'Delete'
    PageName = 'table'
end
end
object AdapterFieldGroup1: TAdapterFieldGroup
    Adapter = table.DataSetAdapter1
    AdapterMode = 'Edit'
object FldCUST_NO: TAdapterDisplayField
    DisplayWidth = 10
    FieldName = 'CUST_NO'
end
object FldCUSTOMER: TAdapterDisplayField
    ...
end
end
end

```

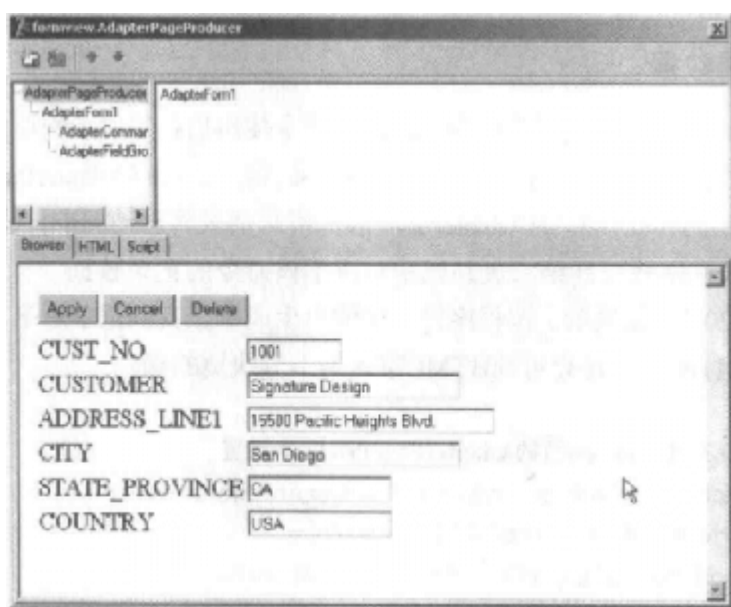


图20.11 WSnapTable在设计时于Web Surface Designer (或 AdapterPageProducer编辑器) 中显示的formview页

在该程序清单中可以看到发送用户返回主页的所有操作，并且看到AdapterMode被设置为Edit，除非出现更新错误或冲突。这时，相同页被再一次展示，并有一个因在窗体顶部添加AdapterErrorList组件而产生的错误描述。

第二页没有展示出来，因为它没有指向任何数据，所以选择它没有任何意义。为了不公布该页，我们可以在对应的初始化代码上加入注释标记。最后，为了保持对数据集合所做改变的持久性，可在控制数据模块的ClientDataSet组件的OnAfterPost和OnAfterDelete事件上调用ApplyUpdates对象方法。

另一个问题（我们还没有解决）是与SQL服务器为每个用户分配ID的操作相关联的，因此当输入新记录时，在ClientDataSet上的数据和在实际数据库上的数据不再对齐。这样就会引起Record Not Found错误。

WebSnap中的主/详关系

DataSetAdapter组件对数据集合间的主/详关系提供了特别支持。在数据集合间像平常一样创建关系后，可为每个数据集合定义适配器，然后连接详细数据集合适配器的MasterAdapter属性。在适配器间建立主/详关系能使它们工作在无缝的条件下。例如，当我们改变主导信息的工作模型或输入新记录时，详细信息会自动输入Edit模型或被刷新。

在WSnapMD范例中使用了一个数据模块来定义这样一种关系。它包括两个ClientDataSet组件，每一组件都通过一个供应器连接到一个SQLDataSet上。每一个数据访问组件都指向一个表格，并且ClientDataSet组件定义了一个主/详关系。相同的数据组件含有两个数据集合适配器，它们指向两个数据集合并且也定义了主/详关系：

```
object dsaDepartment: TDataSetAdapter
    DataSet = cdsDepartment
end
object dsaEmployee: TDataSetAdapter
    DataSet = cdsEmployee
    MasterAdapter = dsaDepartment
end
```

警告：我们曾经尝试使用SimpleDataSet组件来避免数据模块的混乱，但是这个方法并不奏效。程序的主/详部分是正确的，但是当使用相关按钮从一个页面向下一个或者前一个页面移动的时候，总是发生错误。原因是如果我们使用SimpleDataSet，则总存在一个缺陷在每一次互操作的时候关闭数据集合，从而丢失状态信息。

这个WebSnap应用程序中的惟一一个页面具有与两个数据集合适配器都相互连接的AdapterPageProducer组件。这个页面的窗体具有一个连接到主导信息的字段组以及一个连接到详细信息的网格。与其他范例不同的是，我们尝试了通过为不同的元素添加定制属性来改进用户界面。例如，使用灰背景展示一些网格边框（HTML网格经常被Web Surface Designer使用），集中多数元素并增加一些空间。请注意，这里我们为按钮标题添加了额外空间以避免它们显得太小。我们会在下面看到详细的代码，同时这些设置的结果在运行时可以看见，如图20.12所示。

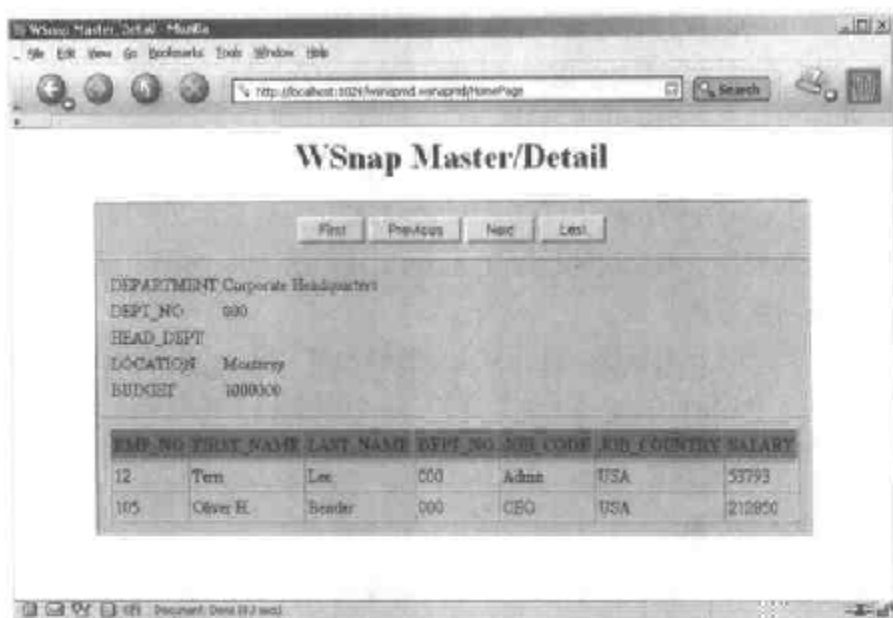


图20.12 WsnapMD示例显示了一个主/详图结构，并有一些定制的输出

```

object AdapterPageProducer: TAdapterPageProducer
object AdapterForm1: TAdapterForm
    Custom = 'Border="1" CellSpacing="0" CellPadding="10" ' +
            'BgColor="Silver" align="center"'
object AdapterCommandGroup1: TAdapterCommandGroup
    DisplayComponent = AdapterFieldGroup1
    Custom = 'Align="Center"'
object CmdFirstRow: TAdapterActionButton...
object CmdPrevRow: TAdapterActionButton...
object CmdNextRow: TAdapterActionButton...
object CmdLastRow: TAdapterActionButton...
end
object AdapterFieldGroup1: TAdapterFieldGroup
    Custom = 'BgColor="Silver"'
    Adapter = WDataMod.dsaDepartment
    AdapterMode = 'Browse'
end
object AdapterGrid1: TAdapterGrid
    TableAttributes.BgColor = 'Silver'
    TableAttributes.CellSpacing = 0
    TableAttributes.CellPadding = 3
    HeadingAttributes.BgColor = 'Gray'
    Adapter = WDataMod.dsaEmployee
    AdapterMode = 'Browse'
object ColEMP_NO: TAdapterDisplayColumn...
object ColFIRST_NAME: TAdapterDisplayColumn...
object ColLAST_NAME: TAdapterDisplayColumn...

```

```

    object ColDEPT_NO: TAdapterDisplayColumn...
    object ColJOB_CODE: TAdapterDisplayColumn...
    object ColJOB_COUNTRY: TAdapterDisplayColumn...
    object ColSALARY: TAdapterDisplayColumn...
  end
end
end

```

会话、用户和许可

WebSnap结构另一个有趣的地方是它对会话和用户的支持。会话是使用典型的方法来支持的，即使用临时cookie。这些cookie被发送到浏览器，因此来自相同用户的随后请求将被系统确认。通过添加数据到会话而不是应用程序适配器，就可决定特殊会话或用户（尽管用户能在同一台计算机上打开多个浏览器窗口运行多个会话）的数据。为了支持会话，应用程序要在内存中保存数据，因此该功能在CGI程序中不可用。

使用会话

为了强调该支持类型的重要性，我们在这里建立了一个WebSnap应用程序，它具有显示点击总数和每次会话总点击数的单页。该程序中包含拥有默认值的SessionService组件，它的MaxSessions和DefaultTimeout属性使用默认的值。对于每个新请求，该程序都要增加页模块的nHits私有字段和当前会话的SessionHits值：

```

procedure TSessionDemo.WebAppPageModuleBeforeDispatchPage(Sender: TObject;
  const PageName: String; var Handled: Boolean);
begin
  // increase application and session hits
  Inc (nHits);
  WebContext.Session.Values ['SessionHits'] :=
    Integer (WebContext.Session.Values ['SessionHits']) + 1;
end;

```

说明：WebContext对象（TWebContext类型）是线程变量，由WebSnap为每个请求创建，用于对程序使用的其他全局变量提供线程安全访问。

相关的HTML将显示状态信息，它使用了由页生成器的OnTag事件求出的定制标记以及由引擎得出的脚本。下面是该HTML文件的核心部分：

```

<h3>Plain Tags</h3>
<p>Session id: <#SessionID>
<br>Session hits: <#SessionHits></p>
<h3>Script</h3>
<p>Session hits (via application): <%=Application.SessionHits.Value%>
<br>Application hits: <%=Application.Hits.Value%></p>

```

这些输出参数是由OnTag事件的处理程序以及字段的OnGetValue事件提供的:

```
procedure TSessionDemo.PageProducerHTMLTag(Sender: TObject; Tag: TTag;  
  const TagString: String; TagParams: TStrings; var ReplaceText: String);  
begin  
  if TagString = 'SessionID' then  
    ReplaceText := WebContext.Session.SessionID  
  else if TagString = 'SessionHits' then  
    ReplaceText := WebContext.Session.Values ['SessionHits']  
end;  
  
procedure TSessionDemo.HitsGetValue(Sender: TObject; var Value: Variant);  
begin  
  Value := nHits;  
end;  
  
procedure TSessionDemo.SessionHitsGetValue(Sender: TObject; var Value: Variant);  
begin  
  Value := Integer (WebContext.Session.Values ['SessionHits']);  
end;
```

该程序的运行效果可以在图20.13中看到, 这里我们在两个不同的浏览器中激活了两个会话。

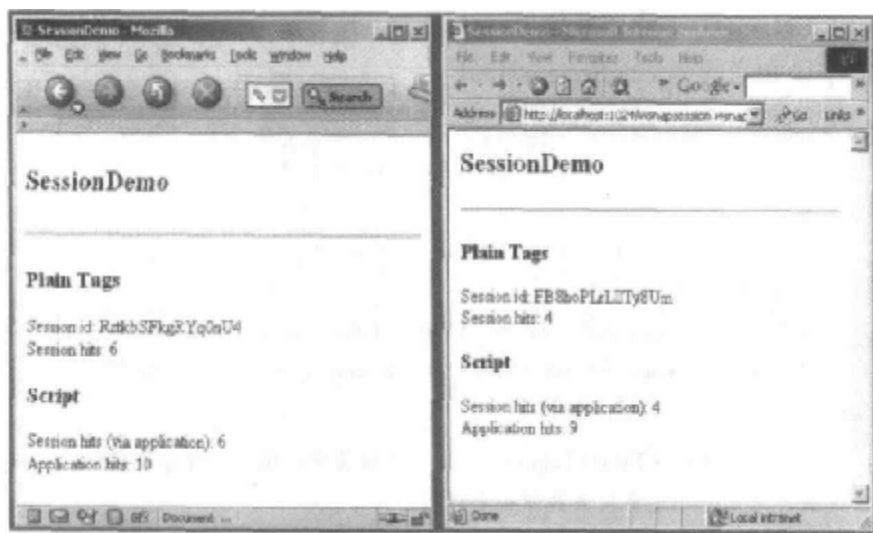


图20.13 浏览器的两个实例在同一WebSnap应用程序的两个不同会话上操作

提示: 在这个例子中, 我们使用了传统的WebBroker标记替代以及新的WebSnap适配器字段和脚本, 因此我们可以对这两个方法进行比较。请记住它们在WebSnap应用程序中都是可用的。

请求登录

除了一般的会话以外, WebSnap也对用户和基于注册权限的会话提供支持。可以添加应用程序用户列表(用WebUserList组件), 每个组件都由名字和密码构成。该组件在数据中相当基本, 它能储存。然而, 除了用用户列表填充它外, 也可在数据库表格(或在其他属性格式)中保存它, 并且使用WebUserList组件的事件来获得定制用户数据并检查用户密码。

一般还需添加应用程序的SessionService和EndUserSessionAdapter组件。这时将要求用户注册，为每页说明它是否能被每个人浏览或只能被注册用户浏览。在网页单元初始化代码中的TWebPageModuleFactory类和TWebAppPageModuleFactory类的构造器上，可以通过设置wpLoginRequire标记来完成该功能。

说明：在工厂中而不是在WebPageModule中提供访问权限和发布信息的原因是，应用程序可以检查访问权限，并且可在没有装载模块的情况下列出相关页。

当用户要查看要求用户身份的页时，在EndUserSessionAdapter组件中指定的注册页将被展示。我们在建立这样一个页面时，可以简单地基于AdapterPageProducer来创建的新网页模块，并为其添加LoginFormAdapter。在页的编辑器上，在窗体内添加字段组，连接字段组到LoginFormAdapter，并用默认注册按钮添加命令组。注册窗体的结果将含有用户名和密码所需的字段，而且请求页也需要它们。最后的值被自动用请求页添加，这时该页将要求权限并且用户还没有注册。完成后，用户将能立即到达所请求的页面而不用被返回到一般的菜单。

从技术上讲，注册后的窗体不被公布，因为当用户没有登录进该系统时，将提供对应的Login命令；当用户注册时，它被Logout命令代替。这可通过网页模块的标准脚本获得，如下所示：

```
<% if (EndUser.Logout != null) { %>
<%   if (EndUser.DisplayName != '') { %>
    <h1>Welcome <%=EndUser.DisplayName %></h1>
<%   } %>
<%   if (EndUser.Logout.Enabled) { %>
    <a href="<%=EndUser.Logout.ASHREF%>">Logout</a>
<%   } %>
<%   if (EndUser.LoginForm.Enabled) { %>
    <a href="<%=EndUser.LoginForm.ASHREF%>">Login</a>
<%   } %>
<% } %>
```

对于WSnapUsers应用程序没有什么别的可以介绍，因为它几乎没有定制代码和设置。用户数据的访问由上面显示的标准模板脚本说明。

单页访问权

除了要求用于访问的登录页外，我们还可以给指定的用户分配更多的权力。事实上任何用户拥有的权力都可由分号或逗号分隔开。用户需要为在适配器组件的ViewAccess和ModifyAccess属性中列出的所有请求页分配所有的权限，当浏览或编辑它们时，它们各自用来说明适配器的属性，即用户是否能看到给定的组件。这些设置比较分散并能应用到整个适配器或某些指定的适配器字段（注意是应用适配器字段而不是设计器内的用户界面组件）中。例如，可以通过隐藏相应字段来隐藏向用户显示的表的相应列（在其他一些情况下也同样，如HideOptions属性指定的那样）。

全局PageDispatcher组件也有OnCanViewPage和OnPageAccessDenied事件，它们能够用来控制在程序代码内对各种程序页的访问，包括更大的控件。

小结

本章介绍了Web服务器应用程序和Delphi库类的两种不同模块：**WebBroker**和**WebSnap**。这些问题不能深入展开，因为一个专题就能写一本书。这里只是介绍它们的起点，为的是让读者能够理解其核心概念，而不是介绍如何创建复杂的示例。

如果想学习**WebSnap**框架的更多知识并查看不同示例的运行情况，可参见Delphi对该部分的扩展演示（在\Demos\WebSnap文件夹中）。其他与XML、XSL和客户端脚本有关的内容将在第22章介绍。

第21章将要讨论Delphi的另一个用来构建Web服务器应用程序的功能强大的框架：**IntraWeb**。这个第三方工具已经存在了好几年，但是它现在变得更加有意义了，因为它被Borland公司包含在Delphi 7之中。正如我们看到的那样，**IntraWeb**能够集成在**WebBroker**或者**WebSnap**程序之中，或者被用做一个单独的体系结构。

第21章 使用IntraWeb进行Web编程

自从Delphi 2以来, Chad Z. Hower建立了一个Delphi体系结构, 以简化Web应用程序的开发, 它的指导思想是使Web编程变得和普通的Delphi程序一样简便而且可视。一些程序员对于动态HTML、JavaScript、Cascading Style Sheets以及最新的Internet技术非常精通。而其他的程序员只是像按照他们构造VCL或者CLX应用程序那样来建造Web应用程序。

IntraWeb适用于第二类开发者, 它非常强大, 即使是专家级的Web程序员也可以从中受益。用Chad的话来说, IntraWeb是用来建立Web应用程序而不是网站的。更进一步地说, IntraWeb组件可以用于一个特定的应用程序, 或者用于WebBroker或WebSnap程序。

在本章中, 我们不能涉及IntraWeb的每一个细节, 在Delphi的面板上和一些模块设计器中有50多个组件。这是一个非常巨大的库。我们将只是介绍它的基础部分, 具体的使用要根据读者自己的需要。

提示: 关于IntraWeb的文档, 参阅Delphi 7随附光碟中的PDF手册。如果没有找到, 也可以从Atozed Software的网站下载。为了获得对于IntraWeb的支持, 参考Borland新闻组。

说明: 本章中有Chad Z. Hower (a.k.a. “Kudzu”, Internet Direct, 即Indy的原作者和项目协调人) 所做的回顾(参考第19章“因特网编程: 套接字和Indy组件”), 他也是IntraWeb的作者。Chad的研究领域包括TCP/IP网络和编程、进程间通信、分布式计算、互联网协议以及面向对象的编程。当不编写程序的时候, 他热衷于户外运动, 如自行车、皮艇等等。Chad还在Kudzu World上发表自由文章、程序、工具等, 其网址为<http://www.Hower.org/Kudzu/>。Chad是位移居国外的美国人, 近年来总是在俄罗斯的圣彼得堡渡夏, 在塞浦路斯的里马索过冬。他的邮件地址是: cpub@Hower.org。

本章主要包括以下内容:

- IntraWeb、Web应用程序和Web站点
- 使用IntraWeb组件
- 集成WebBroker和WebSnap
- Web数据库应用
- 使用客户端组件

IntraWeb简介

IntraWeb是一个组件库, 由Atozed Software创建(www.atozedsoftware.com)。在Delphi 7的专业版和企业版中, 我们可以找到相应的IntraWeb版本。专业版只能够用于Page模式, 如我们将要在本章中看到的那样。虽然Delphi 7是Borland IDE中第一个包含这组组件的, 但是, IntraWeb已经存在很多年了。它接受了各种评估和支持, 包括一些第三方组件的功能。

提示：IntraWeb的第三方组件包括Steema的IWChart（TeeChart的创造者），Centillex的IWBold（用来与Bold集成），Arcana的IWOpenSource、IWTranslator、IWDialogs、IWDataModulePool、TMS的IW组件以及GranPrimo的IWGranPrimo。我们可以在www.atozedsoftware.com中找到第三方组件的更新列表。

虽然我们没核心库的源代码（除非特别付费），但IntraWeb体系结构还是相当开放的，组件的源代码都是可用的。IntraWeb是Delphi标准安装的一部分，但是它也可用于Kylix环境。只要小心编写，IntraWeb应用程序可以用于跨平台环境中。

说明：除了Delphi和Linux版本外，C++ Builder和Java版本的IntraWeb版本也可以获得。一个.NET版本正在开发之中，将会出现在Delphi的.NET版本中。

作为一个Delphi 7的所有者，我们有资格获得最有意义的版本（5.1版）并且将我们的许可证升级到一个完全的IntraWeb企业版，包含有来自Atozed Software的更新与支持（请查看它的网站，以获得更多的详细信息）。更多的正规文档（帮助与PDF文件）也在这个5.1版的更新之中。

从网站到Web应用

就像我在前面提到的，IntraWeb背后的思想是为了建立Web应用而不是Web站点。当我们使用WebBroker或者WebSnap进行工作的时候（在第20章“使用WebBroker和WebSnap进行Web编程”中介绍过），会用到Web页面和页面生成器，并且需要进行HTML生成。当我们使用IntraWeb工作的时候，要用到组件、它们的属性以及它们的事件处理程序，就像我们在Delphi的可视开发中那样。

例如，为了创建一个新的IntraWeb应用，需要在File菜单中选择New Other，移动到New Items对话框中的IntraWeb页面，并且选择Stand Alone Application。在随后的对话框中（它是Delphi而不是IntraWeb的一部分），我们可以选择一个现存的文件夹或者输入一个新的文件夹名称。最终的程序拥有一个项目文件和两个不同的单元，我们将在稍后介绍这个结构。

现在，让我们来创建一个范例程序，在本书的源代码中称为IWSimpleApp。为了建立它，我们使用下面的步骤：

1. 移动到程序的主窗体并且从组件面板的IW Standard页向它添加一个按钮、一个编辑框以及一个列表框。不要从组件面板的Standard页面中添加VCL组件，而应该使用相关的IntraWeb组件：IWButton、IWEdit和IWListbox。
2. 简单地如下改变它们的属性：

```
object IWButton1: TIWButton
  Caption = 'Add Item'
end
object IWEdit1: TIWEdit
  Text = 'four'
end
object IWListbox1: TIWListbox
  Items.Strings = (
    'one'
    'two'
```



```
'three')
```

```
end
```

3. 通过在设计时双击这个组件，处理按钮的OnClick事件，并且编写类似的代码：

```
procedure TFormMain.IWButton1Click(Sender: TObject);  
begin  
    IWListBox1.Items.Add (IWEdit1.Text);  
end;
```

这足以创建一个基于Web的应用程序，用来添加文本到一个列表框中，如我们在图21.1中看到的（这里显示了这个程序的最终版本，其中又增加了几个按钮）。运行这个程序时重点需要注意的是，当我们每一次单击按钮的时候，浏览器都会发送一个新的请求到这个应用程序，它运行着Delphi的事件处理程序并且产生一个新的HTML页面——基于窗体中组件的最新状态。

当我们执行这个应用程序时，不会在浏览器中看到程序的输出，而是在IntraWeb的控制器窗体中看到，如图21.2所示。一个独立的IntraWeb应用是一个完备的HTTP服务器，我们将在稍后的章节看到更多详细的信息。我们看到的窗体通过默认创建的项目文件中的IWRun调用来管理，它在每一个IntraWeb应用程序中。调试窗体允许我们选择一个浏览器并且通过它运行应用程序或者复制应用程序的URL到剪贴板中，因此我们可以将它复制到浏览器中。很重要的是，应用程序默认使用一个随机的端口号，它对于每一次执行都是不同的，因此我们必须每一次都使用一个不同的URL。我们也可更改这个行为，通过选择服务器控制器的设计器（与一个数据模块相似）并且设置port属性。在范例中，我们使用端口号8080，但实际上任何值都是可以的。



图21.1 浏览器中的IWSimpleApp应用程序

IntraWeb代码主要是在服务器端，但是IntraWeb也生成JavaScript来控制一些应用程序特性，我们也可以在客户端执行额外的代码。为此，需要通过使用特定的客户端IntraWeb组件

或者编写我们自己的定制JavaScript代码来实现这个功能。作为比较，IWSimpleApp范例中窗体底部的两个按钮使用了两个不同的办法来显示一个消息框。

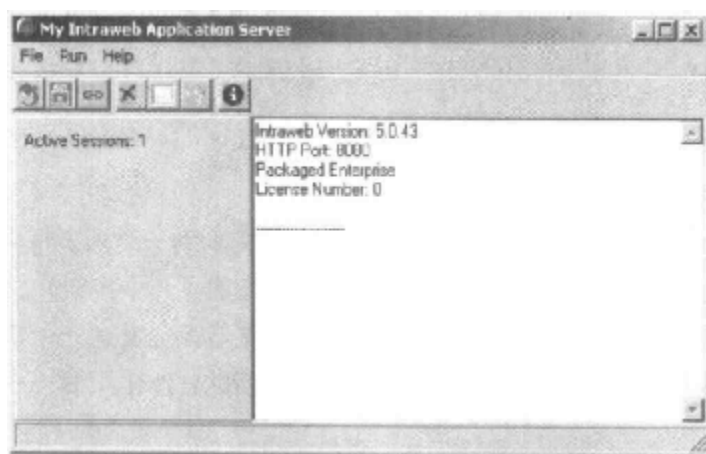
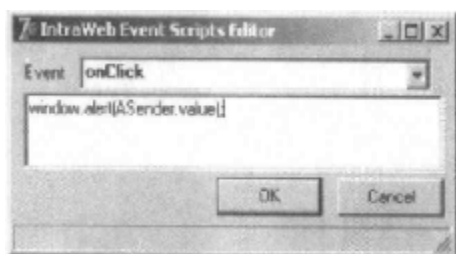


图21.2 一个独立的IntraWeb应用程序的控制器窗体

两个按钮中的第一个（IWButton2）使用了服务器端事件来显示消息，其Delphi代码如下：

```
procedure TFormMain.IWButton2Click(Sender: TObject);
var
    nItem: Integer;
begin
    nItem := IWListbox1.ItemIndex;
    if nItem >= 0 then
        WebApplication.ShowMessage (IWListbox1.Items [nItem])
    else
        WebApplication.ShowMessage ('No item selected');
end;
```

第二个按钮（IWButton3）使用了JavaScript，通过在ScriptEvents属性的特殊属性编辑器中设定正确的JavaScript事件处理程序，即可将其加入Delphi代码中。



了解内幕

我们已经看到了创建一个IntraWeb应用程序与创建一个基于窗体的Delphi程序一样简单，只需将组件放置在一个窗体中并且处理它们的事件。当然，效果是很不一样的，因为应用程序是运行在一个浏览器中的。为了让读者对此有一个了解，我们来简单地看一下这个简单程序幕后所做的工作。这样做有助于大家理解设置组件属性以及其他操作的效果。

这是一个基于浏览器的程序，因此查看它发往浏览器的HTML是对其进行了解的更佳办法。打开IWSimpleApp程序页面（因为篇幅原因没有在这里列出），我们将发现它被分为三个主要部分。第一部分是一个样式列表（基于HTTP的style标记），使用下面的代码：

```
.IWEDIT1CSS {position:absolute;left:40;top:40;z-index:100;
font-style:normal;font-size:10pt;text-decoration:none;}
```

IntraWeb使用样式来决定每一个组件的可视外观，例如字体和颜色，而且使用默认的绝对位置来决定组件的位置。每一种样式都受到IntraWeb组件的某个属性的影响，因此很容易对其进行测试。如果读者对样式表不是很熟悉，那么只要使用属性并且信任IntraWeb可以完成在Web页面上绘制组件的任务就可以了。

第二部分包括JavaScript脚本。主要的脚本块包括初始化代码和组件的客户端事件处理程序，如下所示：

```
function IWBUTTON1_OnClick(ASender) {
    return SubmitClickConfirm('IWBUTTON1','', true, '');
}
```

这个处理程序激活了相应的服务器端代码。如果我们已经直接在IntraWeb应用中提供了JavaScript代码，如前面讨论的那样，我们将看到下列代码：

```
function IWBUTTON3_onClick(ASender) {
    window.alert(ASender.value);
}
```

页面的脚本部分也通过浏览器引用了其他的文件，并且通过IntraWeb来使用。其中有些文件是普通的，其他的则与特定的浏览器挂钩；因此，IntraWeb要检查正在使用的浏览器然后返回不同的JavaScript代码和基本JavaScript文件。

说明：因为JavaScript不是在所有的浏览器中都相同，故IntraWeb只支持其中的一部分，包括所有最近版本的微软IE浏览器、Netscape Navigator以及开放源代码的Mozilla，笔者在编写本章时已用过它们。Opera对于JavaScript的支持更为有限，所以默认情况下，当IntraWeb遇到它时，就会产生一个错误（根据控制器的属性SupportedBrowsers而定）。Opera可以与Arcana组件一起使用，并且将在IW 5.1中得到官方支持。请记住，一个浏览器可以伪造它的身份。例如，Opera经常“伪装”成IE浏览器，这样就可以去访问某些限制浏览器类型的站点，但是可能会导致运行时错误或者不稳定。

生成的HTML的第三部分是对页面结构的定义，在body标记中是一个form标记，带有下一个将要执行的操作：

```
<form onsubmit="return FormDefaultSubmit();" name="SubmitForm"
action="/EXEC/3/DC323E01B09C83224E57E240" method="POST">
```

form标记带有特定的用户接口组件，例如按钮和编辑框：

```
<input type="TEXT" name="IWEDIT1" size="17" value="four"
id="IWEDIT1" class="IWEDIT1CSS">
<input value="Add Item" name="IWBUTTON1" type="button"
onclick="return IWBUTTON1_OnClick(this);"
id="IWBUTTON1" class="IWBUTTON1CSS">
```

该窗体还具有一些隐藏的组件，IntraWeb使用它们来传递信息。但是，URL是在IntraWeb中传递信息的最重要的办法，在程序中，它是像下面这样的：

```
http://127.0.0.1:8080/EXEC/2/DC323E01B09C83224E57E240
```

第一部分是IP地址以及端口，由独立的IntraWeb应用程序使用（当我们使用一个不同的体系结构来部署应用程序的时候，它会改变），后跟EXEC命令、一个前递增请求号以及一个会话ID。我们稍后再讨论会话，但是可以说IntraWeb使用一个URL令牌而不是cookie来使它的应用程序可用，而不管浏览器的设定。如果愿意的话，可以使用cookie来代替URL，但要改变服务器控制器的TrackMode属性。

警告：Delphi 7所带的IntraWeb版本中有一个缺陷，是关于cookies和一些时区设定的。它已经被修补了，补丁文件可以在Atozed软件网站中免费获得。

IntraWeb体系结构

在我们编写更多的范例来演示Delphi 7中IntraWeb组件的使用之前，让我们先来讨论一下IntraWeb的另外一个关键元素：用来在该库的基础上建立和部署应用程序的不同的体系结构。我们可以在Application模式（包含IntraWeb的全部特性）或者Page模式（简单版本，只可放入现有的Delphi WebBroker或WebSnap应用程序）中创建IntraWeb项目。Application模式的应用程序可以被部署为ISAPI库、Apache模块或者是使用IntraWebStandalone模式（Application模式结构的变种）。Page模式可以被部署为任意其他的WebBroker应用程序（ISAPI、Apache模式和CGI等）。IntraWeb具有三个不同的但是有些重叠的体系结构：

Standalone模式 提供一个定制的Web服务器，正如我们在前面的范例中建立的那样。这对于调试应用程序是非常方便的（因为我们可以从Delphi的IDE中运行它，并且在程序的任何地方添加断点）。也可以使用Standalone模式在内部网络中部署应用程序，并且允许用户在各自的计算机上使用一个Web界面进行离线工作。如果我们使用-install标记来运行一个独立的IntraWeb应用程序，它将作为一个服务运行，而不会显示对话框。Standalone模式允许我们使用IntraWeb本身将一个Application模式的IntraWeb程序部署为一个Web服务器。

Application模式 允许我们在商用服务器上部署IntraWeb应用程序，建立一个Apache模块或者IIS库。Application模式包括会话管理以及所有的IntraWeb特性，它是部署一个可扩展的Web应用程序的首选办法。更准确地说，Application模式的IntraWeb程序能够被部署为一个独立的程序、ISAPI库或者Apache模块。

Page模式 为将IntraWeb页面集成到WebBroker以及WebSnap应用程序中提供了办法。我们可以向现有的应用程序中添加特性或者依靠其他的技术来建立基于HTML的动态站点，同时提供与IntraWeb的互操作。Page模式是将IntraWeb用于CGI应用程序之中的唯一选择，但是它缺乏会话管理功能。独立的IntraWeb服务器不支持Page模式。

在本章剩下部分的范例中，我们将使用Standalone模式来简化调试的过程，但是我们将介绍Page模式的支持。

建立IntraWeb应用程序

当我们建立一个IntraWeb应用的时候, 有多个组件可用。例如, 如果我们查看Delphi的组件面板的IW Standard页面, 将看到一个核心组件的列表, 从按钮、复选框、单选按钮、编辑框、列表框、备注等等, 到树状视图、菜单、计时器、网格和链接组件。我们不会列举所有这些组件并且使用范例对它们分别进行介绍, 而是在一些演示中使用某些组件, 以强调IntraWeb的体系结构而不是特定的细节。

我们已经建立了一个范例, 称为IWTree, 显示了IntraWeb的菜单和树状视图组件, 同时也描绘了在运行时创建组件的特性。这个方便的组件在一个动态菜单中可以使用标准的Delphi菜单, 这只需将它的AttachedMenu属性指向一个TMenu组件即可:

```
object MainMenu1: TMainMenu
  object Tree1: TMenuItem
    object ExpandAll1: TMenuItem
    object CollapseAll1: TMenuItem
    object N1: TMenuItem
    object EnlargeFont1: TMenuItem
    object ReduceFont1: TMenuItem
  end
  object About1: TMenuItem
    object Application1: TMenuItem
    object TreeContents1: TMenuItem
  end
end
object IWMenu1: TIWMenu
  AttachedMenu = MainMenu1
  Orientation = iwOHorizontal
end
```

如果菜单项用于在代码中处理OnClick事件, 则它们在运行时将成为链接。我们可以在图21.3中看到一个浏览器中菜单的范例。这个范例的第二个组件是一个树状视图, 并且带有预定义节点的集合。这个组件具有很多的JavaScript代码, 使我们能够在浏览器中直接扩展和折叠节点。同时, 菜单项允许程序在菜单上进行操作, 通过扩展与折叠节点或者改变字体来实现。下面是这两个事件处理程序的代码:

```
procedure TFormTree.ExpandAll1Click(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to IWTreeView1.Items.Count - 1 do
    IWTreeView1.Items[i].Expanded := True;
  end;
procedure TFormTree.EnlargeFont1Click(Sender: TObject);
begin
```

```
IWTreeView1.Font.Size := IWTreeView1.Font.Size + 2;
end;
```

由于IntraWeb组件和标准的Delphi VCL组件相似，故该代码是非常易读和易于理解的。



图21.3 IWTree范例描述了一个菜单，一个树状视图以及动态创建备注组件

这个菜单具有两个子菜单，略显复杂。第一个子菜单显示应用程序ID，它是一个应用程序的执行/会话ID，在WebApplication全局对象的AppID属性中。第二个子菜单，Tree Contents，显示主要层面的第一个树节点以及子节点的数量。有趣的是，显示在备注组件中的信息是在运行时创建的（参考图21.3），与我们将要在VCL应用程序中做的一样：

```
procedure TForm1.TreeContents1Click(Sender: TObject);
var
  i: Integer;
begin
  with TIWMemo.Create(Self) do
  begin
    Parent := Self;
    Align := alBottom;
    for i := 0 to IWTreeView1.Items.Count - 1 do
      Lines.Add (IWTreeView1.Items [i].Caption + ' (' +
        IntToStr (IWTreeView1.Items [i].SubItems.Count) + ')');
    end;
  end;
end;
```

提示：IntraWeb的对齐功能与VCL的相应功能类似。例如，这个程序的菜单使用alTop对齐，树状视图使用alClient对齐，动态备注使用alBottom对齐。作为一个替代的选项，我们可以使用锚（也在VCL中一样）：创建右下按钮，或者位于页面中部的组件，将四个锚全部设定。参见下面的演示范例。

编写多页的应用程序

我们建立的所有程序都只有单个页面。现在让我们创建一个带有第二个页面的IntraWeb应用程序。正如我们看到的，即使在这种情况下，IntraWeb的开发也类似于标准的Delphi或者Kylix开发，并且与大多数其他的互联网开发库不同。这个范例将会介绍一些由IntraWeb应用向导自动产生的源代码。

那么就让我们开始吧！IWTwoForms范例的主窗体具有一个IntraWeb网格。这个强大的组件允许我们在一个HTML网格内放置文本和其他的组件。例如，这个网格的内容在程序开始时就已经确定（在主窗体的OnCreate事件处理程序中）：

```
procedure TFormMain.IWAppFormCreate(Sender: TObject);
var
  i: Integer;
  link: TIWURL;
begin
  // set grid titles
  IWGrid1.Cell[0, 0].Text := 'Row';
  IWGrid1.Cell[0, 1].Text := 'Owner';
  IWGrid1.Cell[0, 2].Text := 'Web Site';
  // set grid contents
  for i := 1 to IWGrid1.RowCount - 1 do
  begin
    IWGrid1.Cell[i, 0].Text := 'Row ' + IntToStr(i+1);
    IWGrid1.Cell[i, 1].Text := 'IWTwoForms by Marco Cantu';
    link := TIWURL.Create(Self);
    link.Text := 'Click here';
    link.URL := 'http://www.marcocantu.com';
    IWGrid1.Cell[i, 2].Control := link;
  end;
end;
```

这段代码的效果如图21.4所示。除了这个输出以外，还有一些有趣的东西值得注意。首先，网格组件使用了Delphi锚（Anchors，都设定为False）来生成使它位于页面中央的代码，即使用户可改变浏览器窗口的大小。第二，我们在第三列中添加了一个IWURL组件，当然，读者可以在这个网格中添加其他的任何组件，包括按钮和编辑框。

第三个也是最重要的考虑是一个IWGrid被转化为HTML网格，它带有或者不带有框架。下面是为一个网格行生成的HTML片断：

```
<tr>
  <td valign="middle" align="left" NOWRAP>
    <font style="font-size:10pt;">Row 2</font>
```

```

</td>
<td valign="middle" align="left" NOWRAP>
  <font style="font-size:10pt;">IWTwoForms by Marco Cant </font>
</td>
<td valign="middle" align="left" NOWRAP>
  <font style="font-size:10pt;"></font>
  <a href="#" onclick="parent.LoadURL('http://www.marcocantu.com')"
    id="TIWURL1" name="TIWURL1"
    style="z-index:100;font-style:normal;font-size:10pt;text-decoration:none;">
    Click here</a>
</td>
</tr>

```

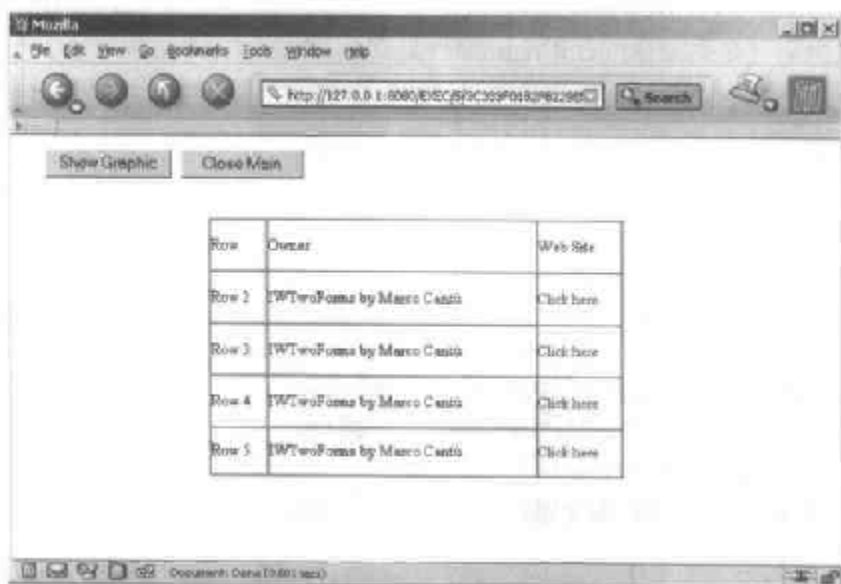


图21.4 IWTwoForms范例中使用了IWGrid组件，嵌入的文本和IWURL组件

提示：在前面的列表中，请注意链接的URL是通过JavaScript来激活的，而不是使用一个直接的链接。之所以发生这种情况是因为在IntraWeb中，所有的操作均允许额外的客户端操作，如确认、检查以及提交。例如，如果我们设定了一个组件的Required属性，而这个字段是空的，那么数据将不会被提交，而且我们将看到一个JavaScript错误（可以通过设置描述性的FriendlyName属性来定制）。

该程序的核心特性是它拥有显示第二个页面的能力，为了实现这一点，我们首先需要添加一个新的IntraWeb页面到这个应用程序中——使用Delphi的New Items对话框（File ►New►Other）中IntraWeb页面的Application Form选项。像往常一样，向这个页面添加一些IntraWeb组件，并且向主窗体添加一个按钮或者其他我们希望在二级窗体中显示的控制（含有存储在主窗体字段中的anotherform引用）：

```

procedure TFormMain.btnShowGraphicClick(Sender: TObject);
begin
  anotherform := TAnotherForm.Create(WebApplication);
  anotherform.Show;
end;

```


即使程序调用了Show方法，它可以被认为像是一个ShowModal调用，因为IntraWeb将可视的页面作为一个堆栈。最后的页面放在堆栈的顶层，并且在浏览器中进行显示。通过关闭这个程序（隐藏或者破坏它），我们可以重新显示前一个页面。在程序中，二级页面通过调用Release方法来关闭自身，就好像在VCL中处理一个正在执行的窗体一样。我们还可以隐藏二级窗体并且再次显示它，以避免每一次都重新创建它，特别是当这样做就意味着丢失用户编辑操作的时候。

警告：我在程序的主窗体中添加了一个Close按钮。注意，它不应该调用Release，而是应该调用WebApplication对象的Terminate方法，传递输出消息，如在WebApplication.Terminate('Goodbye!')中那样。这个示例使用了另一个调用：TerminateAndRedirect。

现在，读者已经看到如何使用两个窗体来创建一个IntraWeb应用程序，下面让我们简要检查一下IntraWeb是如何创建主窗口的。相关代码在创建一个新程序的时候由IntraWeb向导创建，位于项目文件中：

```
begin
    IWRun(TFormMain, TIWServerController);
```

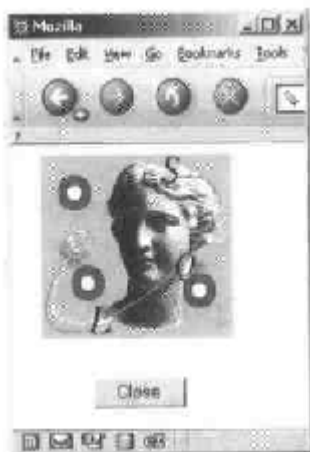
这与Delphi的标准项目文件不同，因为它调用了-一个全局函数，而不是应用一个代表这个应用程序的全局对象的方法。然而效果非常相似。两个参数是主窗体和IntraWeb控制器的类，它们处理会话以及其他我们将要看到的特性。

IWTwoForms范例的二级窗体显示了IntraWeb的另外一个有趣的特性：它支持图像。该窗体中有一个带有经典的Delphi Athena图像的图像控件。这是通过将一个位图载入一个IWImage组件来实现的。IntraWeb将位图转化为一个JPEG，将其存储于一个在应用程序文件夹下创建的缓存文件夹中，并且返回一个指向它的引用，其HTML代码如下：

```

```

由IntraWeb提供的和由程序开发的额外特性是，用户可以使用鼠标单击图像，以启动服务器端代码对图像进行修改。在这个程序中，效果是画若干绿色的小圆圈。



这个效果可以通过下面的代码来获得：

```
procedure Tanotherform.IWImage1MouseDown(ASender: TObject;
    const AX, AY: Integer);
var
```

```

aCanvas: TCanvas;
begin
  aCanvas := IWImage1.Picture.Bitmap.Canvas;
  aCanvas.Pen.Width := 8;
  aCanvas.Pen.Color := clGreen;
  aCanvas.Ellipse(Ax - 10, Ay - 10, Ax + 10, Ay + 10);
end;

```

警告：绘图操作发生在位图画布上。不要使用Image画布（就像在VCL的TImage组件中做的那样），并且不要在其中使用JPEG图像，否则我们将看不到效果，或者会发生一个运行时错误。

会话管理

如果读者曾经从事过Web编程，那么就应该知道会话管理是一个很复杂的课题。IntraWeb提供了预定义的会话管理并且简化了处理会话的工作。如我们在一个特定的窗体上使用会话数据，我们所要做的就是添加一个字段到这个窗体。IntraWeb窗体以及其他组件对于每一个用户会话都有一个实例。例如，在IWSession范例中，我们向窗体添加了一个称为FormCount的字段。作为比较，我们还声明了一个全局单元变量，称为GlobalCount，它被应用程序中的所有实例或者会话共享。

为了加强我们对会话数据的控制，并且使多个窗体共享它，可以定制TUserSession类，它是由IntraWeb Application Wizard放置在ServerController单元中的。在IWSession范例中，我们如下定制了这个类：

```

type
  TUserSession = class
  public
    UserCount: Integer;
  end;

```

IntraWeb可以为每一个新的会话创建一个该对象的实例，正如ServerController单元中TIWServerController类的IWServerControllerBaseNewSession方法显示的那样：

```

procedure TIWServerController.IWServerControllerBaseNewSession(
  ASession: TIWApplication; var VMainForm: TIWAppForm);
begin
  ASession.Data := TUserSession.Create;
end;

```

在程序的代码中，通过访问RWebApplication全局变量的数据字段，能够引用会话对象，用来访问当前的用户会话。

说明：RWebApplication是一个threadvar变量，定义在IWInit单元中。它使我们能够用一种线程安全的方式访问会话数据；但在访问它时需要特别注意，特别是在一个多线程的环境中。这个变量可以用于一个窗体或者控件之外（它原本是基于会话的），这就是为什么它主要用于数据模块、全局例程和非IntraWeb类内部的原因。

另外，默认的ServerController单元提供了一个帮助函数供我们使用：

```
function UserSession: TUserSession;  
begin  
    Result := TUserSession(RWebApplication.Data);  
end;
```

因为大多数代码都已经生成了，故在添加数据到TUserSession类之后，我们可以简单地通过UserSession函数使用它，就像下面从IWSession范例中抽取的代码一样。当我们单击一个按钮的时候，程序会增加几个计数器（一个全局的和两个会话特定的），并且在标签中显示它们的值：

```
procedure TFormMain.IWButton1Click(Sender: TObject);  
begin  
    InterlockedIncrement (GlobalCount);  
    Inc (FormCount);  
    Inc (UserSession.UserCount);  
    IWLabel1.Text := 'Global: ' + IntToStr (GlobalCount);  
    IWLabel2.Text := 'Form: ' + IntToStr (FormCount);  
    IWLabel3.Text := 'User: ' + IntToStr (UserSession.UserCount);  
end;
```

请注意，程序使用了Windows的InterlockedIncrement调用来避免多个线程同时访问全局共享变量。另一个办法包括了使用一个关键部分或者位于IdThreadSafe单元内的Indy的TidThreadSafeInteger。

图21.5显示了程序的输出，带有在两个不同浏览器中运行的两个会话。程序还包括一个复选框，它可激活一个计时器。在IntraWeb应用程序中，计时器与在Windows中的工作大致相同。当计时器间隔到期后，执行代码。在Web上，这意味着可以通过触发一个JavaScript代码中的更新来刷新页面。

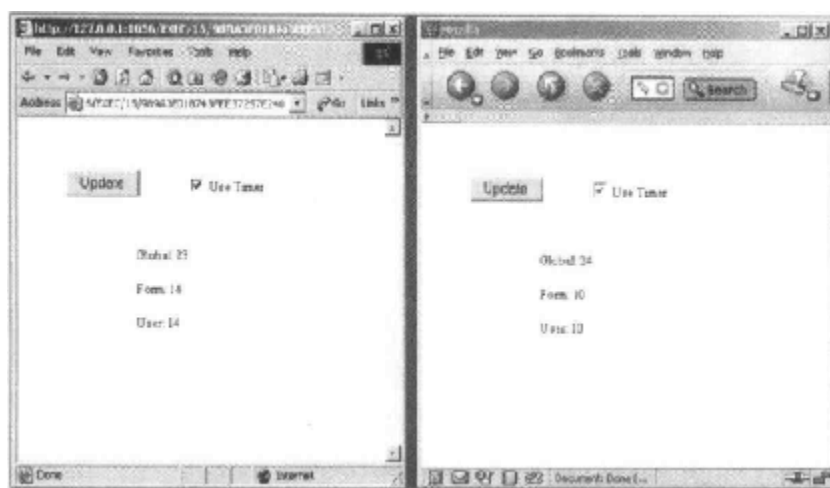


图21.5 IWSession应用程序具有全局和会话特定的计数器，通过在两个不同的浏览器内甚至是在同一个浏览器内运行两个会话便可看到

```
IWTIMER1:=setTimeout('SubmitClick("IWTIMER1","", false)',5000);
```

集成WebBroker和WebSnap

到现在为止，我们已经建立了一个独立的IntraWeb应用程序。当我们在一个库内创建一个IntraWeb应用程序并将其部署在IIS或者Apache内的时候，基本上就处于相同的情形之下。但是，如果我们希望使用IntraWeb的Page模式（它被集成在一个WebBroker或者WebSnap Delphi应用程序的一个IntraWeb页面中），情况就截然不同。

连接这两个世界的桥梁就是IWPageProducer组件。这个组件可连接一个WebBroker动作，这与其他的面生成器组件类似，并且具有一个特殊的事件，我们可以用它来创建和返回一个IntraWeb窗体：

```
procedure TWebModule1.IWPageProducer1GetForm(ASender: TIWPageProducer;
  AWebApplication: TIWebApplication; var VForm: TIWebPageForm);
begin
  VForm := TFormMain.Create(AWebApplication);
end;
```

使用单独这一行代码，并且在Web模块中添加IWModuleController组件，WebBroker应用程序便可以嵌入一个IntraWeb页面，正如CgiIntra程序所做的那样。IWModuleController为IntraWeb提供了核心服务的支持。一个该类型的组件必须存在于每一个IntraWeb项目之中，以保证正常的工作。

警告：Delphi 7中的Web App Debugger和IWModuleController组件有一个问题。这个缺陷已经在一个免费更新中得到了修复。

下面概述一下本例中Web模块的DFM:

```
object WebModule1: TWebModule1
  Actions = <
    item
      Default = True
      Name = 'WebActionItem1'
      PathInfo = '/show'
      OnAction = WebModule1WebActionItem1Action
    end
    item
      Name = 'WebActionItem2'
      PathInfo = '/iwdemo'
      Producer = IWPageProducer1
    end>
  object IWModuleController1: TIWModuleController
  object IWPageProducer1: TIWPageProducer
    OnGetForm = IWPageProducer1GetForm
  end
end
```

因为这是一个Page模式的CGI应用程序，故它不具备会话管理功能。此外，页面中组件的状态不是通过写事件处理程序来自动更新的，就像在标准的IntraWeb程序中那样。为了达到同样的效果，我们需要编写特定的代码来进一步处理HTTP请求的参数。从这个简单的范例程序中我们可以清楚地看到，Page模式要比Application模式功能少，但是它更加灵活。特别是IntraWeb的Page模式允许我们添加可视的RAD设计功能到WebBroker和WebSnap应用程序中。

控制布局

CgiIntra程序具有另一个在IntraWeb中可用的有趣的技术特性：基于HTML来定义一个定制的布局。在前面建立的程序中，最终页面是一系列设计时放置在窗体上的组件的映射，在这里我们可以使用属性来改变最终的HTML。但是如果我们需要在一个复杂的HTML中嵌入数据表项窗体该怎么办呢？使用IntraWeb组件建立完整的页面内容是很笨拙的办法，即使我们使用IWText组件在一个IntraWeb页面中嵌入一个定制的HTML片断。

另一个可以选择的办法是使用IntraWeb的布局管理器。在IntraWeb中，我们总是使用布局管理器，默认是IWLayoutMgrForm组件。其他两个可行的办法是，使用通过外部HTML模板文件进行工作的IWTemplateProcessorHTML和使用通过内部HTML进行工作的IWLayoutMgrHTML。

第二个组件包括一个强大的HTML编辑器，我们可以用它来准备普通的HTML，以及嵌入所需的IntraWeb组件（我们必须使用一个外部HTML编辑器）。此外，当我们从这个编辑器中选择一个IntraWeb组件时（通过双击一个IWLayoutMgrHTML组件来激活），可以使用Delphi的对象检验器来定制组件的属性。如我们在图21.6中看到的那样，IntraWeb中的HTML的Layout Editor是一个强大的可视HTML编辑器，它生成的HTML文本将放在一个单独页面中（这个编辑器还会不断地改进升级，并且修补一些瑕疵）。

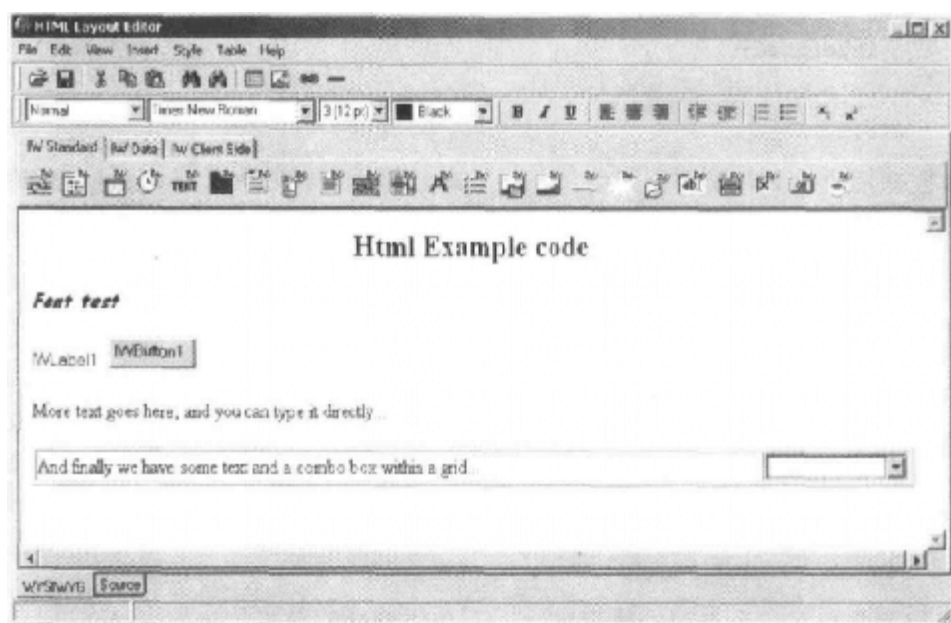


图21.6 IntraWeb的Layout Editor是一个完备的可视HTML编辑器

在生成的HTML中，定义了页面的结构。只可使用特殊的、带花括号的标记对该组件进行标识，像下面的例子中那样：

```
<P> {%TWLabel1%} {%TWButton1%}</P>
```

提示：当我们使用HTML的时候，组件不能使用绝对的位置，而是要跟随HTML进行变化。因此，窗体是惟一的组件持有者，因为窗体中组件的位置和大小是被忽略的。

不用说，我们在HTML Layout Editor的可视设计器中看到的HTML与我们在浏览器中看到的几乎完全一样。

Web数据库应用程序

作为一个Delphi的库，IntraWeb可用控件中的很大部分都与开发数据库应用有关。IntraWeb Application Wizard具有这样一个版本，允许我们创建一个带有数据模块的应用程序。这是开发数据库应用的很好的开端。在这种情况下，应用程序的预定义代码可为每一个会话创建一个数据模块的实例，将它保存在会话数据之中。

这里是TUserSession类的预定义（以及它的构造器），用于一个带有数据模块的IntraWeb应用程序：

```
type
  TUserSession = class(TComponent)
  public
    DataModule1: TDataModule1;
    constructor Create(AOwner: TComponent); override;
  end;

constructor TUserSession.Create(AOwner: TComponent);
begin
  inherited;
  DataModule1 := TDataModule1.Create(AOwner);
end;
```

该数据模块单元并没有全局变量，如果有，那么所有的数据都将会在所有会话之间共享，这样当在多个线程中发生并发请求时，就会有麻烦。但是，该数据模块有一个与Delphi所用的全局变量同名的全局函数，用以访问当前的会话数据模块：

```
function DataModule1: TDataModule1;
begin
  Result := TUserSession(RWebApplication.Data).DataModule1;
end;
```

这意味着可以像下面这样编写代码：

```
DataModule1.SimpleDataSet1
```

但是我们不是访问一个全局的数据模块，而是使用当前会话的数据模块。

在描述数据库数据IWScrollData的第一个范例程序中，我们向数据模块添加了一个SimpleDataSet组件，并且使用下面的配置向主窗体添加了一个IWDBGrid组件：

```

object IWDEGrid1: TIWDEGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  BorderSize = 1
  CellPadding = 0
  CellSpacing = 0
  Lines = tlRows
  UseFrame = False
  DataSource = DataSource1
  FromStart = False
  Options = [dgShowTitles]
  RowAlternateColor = clSilver
  RowLimit = 10
  RowCurrentColor = clTeal
end

```

最重要的设置是删除一个带有控件及其滚动条的框架（UseFrame属性）、从当前数据集的位置处显示数据（FromStart属性）以及浏览器中显示的行数（RowLimit属性）。在这个用户界面中，我们去掉了垂线，并且每隔一行着色一次。我们还为当前行选择一种色彩（RowCurrentColor属性）；否则，替换的颜色将不能正确显示，因为当前行将与背景有同样的颜色，而不论它的位置是什么（设定RowCurrentColor属性为clNone来查看我所说的）。这些设定的输出效果如图21.7所示，运行IWScrollData范例也可看到此效果。

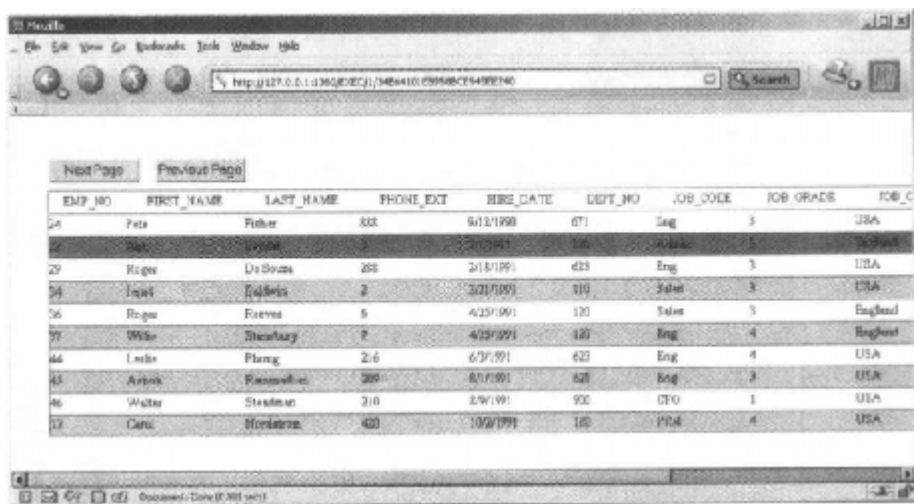


图21.7 IWScrollData范例中的数据敏感网格

当窗体被创建时，程序打开数据集，使用连接到当前数据源的数据集：

```

procedure TFormMain.IWAppFormCreate(Sender: TObject);
begin
  DataSource1.DataSet.Open;
end;

```

范例的相关代码在按钮的代码中，可以用来在数据中移动，显示前页与后页的信息。下面是两个方法之一的代码，另一个也很相似：

```

procedure TFormMain.btnNextClick(Sender: TObject);
var
    i: Integer;
begin
    nPos := nPos + 10;
    if nPos > DataSource1.DataSet.RecordCount - 10 then
        nPos := DataSource1.DataSet.RecordCount - 10;
    DataSource1.DataSet.First;
    for i := 0 to nPos do
        DataSource1.DataSet.Next;
    end;

```

链接到细文

IWScrollData范例的网格显示了一个单独的表格数据页面。使用上面的按钮可以让我们滚动这个页面。IntraWeb中另外一种风格的网格是通过框架式网格来提供的，它可以使用框架以及内部滚动条将大量数据移动到Web浏览器中，它在一个单一固定的屏幕区域内。这个效果由IWGridDemo范例显示。

这个范例中使用了另一个很好的方法来定制网格，就是设定这个网格的Columns集合属性。这个设置允许我们调节输出以及指定列的行为，例如显示超级链接或者处理对项目以及标题单元的单击。在IWGridDemo范例中，其中的一列（Last Name）被转换为超级链接，雇员号码作为参数链传递给后面的命令，我们可以在图21.8中看到。

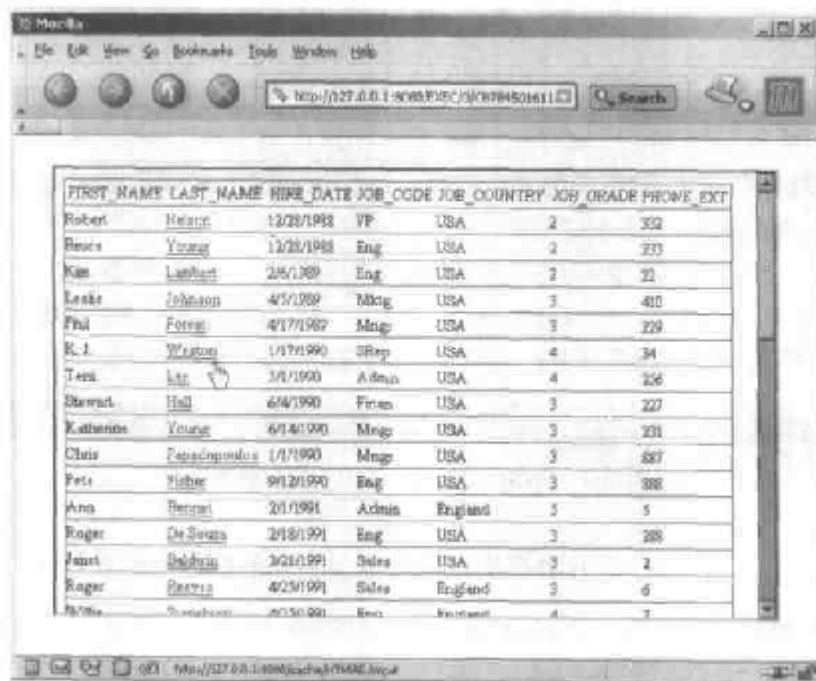


图21.8 IWGridDemo范例的主创体，使用了一个带有帧的网格，并且使用超级连接指向二级窗体

程序清单21.1显示了这个网格的关键属性。请注意“last name”列，它是一个链接字段，亦即该单元内的文本要被转换为一个超级链接，以及事件处理程序，与它的选择相关。在这个方法中，程序创建了一个二级窗体，在其中，用户可以编辑数据：


```

procedure TGridForm.IWDBGrid1Columns1Click(ASender: TObject;
  const AValue: String);
begin
  with TRecordForm.Create (WebApplication) do
    begin
      StartID := AValue;
      Show;
    end;
  end;

```

程序清单21.1 IWGridDemo范例中的IWDBGrid的属性

```

object IWDBGrid1: TIWDBGrid
  Anchors = [akLeft, akTop, akRight, akBottom]
  UseFrame = True
  UseWidth = True
  Columns = <
    item
      Alignment = taLeftJustify
      BGColor = clNone
      DoSubmitValidation = True
      Font.Color = clNone
      Font.Enabled = True
      Font.Size = 10
      Font.Style = []
      Header = False
      Height = '0'
      VAlign = vaMiddle
      Visible = True
      Width = '0'
      Wrap = False
      BlobCharLimit = 0
      CompareHighlight = hcNone
      DataField = 'FIRST_NAME'
      Title.Alignment = taCenter
      Title.BGColor = clNone
      Title.DoSubmitValidation = True
      Title.Font.Color = clNone
      Title.Font.Enabled = True
      Title.Font.Size = 10
      Title.Font.Style = []
      Title.Header = False
      Title.Height = '0'
      Title.Text = 'FIRST_NAME'
      Title.VAlign = vaMiddle
      Title.Visible = True

```

```

        Title.Width = 0;
        Title.Wrap = False;
    end
    item
        DataField = 'LAST_NAME';
        LinkField = 'EMP_NO';
        OnClick = IWDBGrid1Columns1Click;
    end
    item
        DataField = 'HIRE_DATE';
    end
    item
        DataField = 'JOB_CODE';
    end
    item
        DataField = 'JOB_COUNTRY';
    end
    item
        DataField = 'JOB_GRADE';
    end
    item
        DataField = 'PHONE_EXT';
    end>
DataSource = DataSource1;
Options = [dgShowTitles];
end

```

通过设定第一个窗体的StartID属性，我们可以定位正确的记录：

```

procedure TRecordForm.SetStartID(const Value: string);
begin
    FStartID := Value;
    DataSource1.DataSet.Locate('EMP_NO', Value, []);
end;

```

提示：IWDBGrid列也具有一个OnTitleClick事件，我们可以处理它来排序数据或者对该列进行其他的操作。

二级窗体与主窗体一样连接到相同的数据模块上，因此，在数据库的数据被更新后，我们可以在网格中看到它（但是这个更新只保留在内存中，因为程序没有进行ApplyUpdates调用）。二级窗体使用IntraWeb提供的一些编辑控件和一个导航控件。我们可以从图21.9中看到该窗体的运行时状态。

将数据移动到客户端

无论我们怎么使用它，IWDBGrid组件都要依靠嵌入单元的数据库数据来创建HTML文档，但是它不能够在客户端对数据进行操作。一个不同的组件（或者是一组IntraWeb组件）允许我们使用一种不同的模式。亦即使用定制的格式将数据发送到浏览器，由浏览器中的

JavaScript代码生成网格并且对数据进行操作——从一个记录移动到另一个记录，而不需要向服务器请求更多的数据。

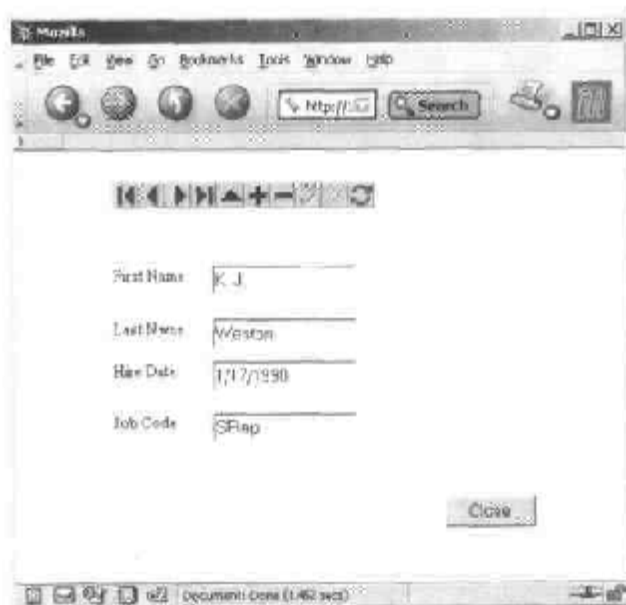


图21.9 IWGridDemo范例的二级窗体，允许用户编辑数据并且在记录中进行定位

说明：这个结构与Delphi的本地Internet Express体系结构类似，我们将要在第22章“使用XML技术”中进行介绍。

我们能够为客户端应用程序使用多种IntraWeb组件，这里给出最重要的一些：

IWClientSideDataSet 一个驻留内存的数据集，由程序中的ColumnNames和Data属性来定义。在未来的更新中，我们将能够编辑客户端数据，对其进行排序、过滤，并且定义主/从数据结构以及其他操作。

IWClientSideDataSetDBLink 一个数据供应器，我们可以连接到任何的Delphi数据集，亦即使用DataSource属性进行连接。

IWDynGrid 一个动态的网格组件，使用Data属性连接到前面两个组件之一。这个组件可将所有数据移动到浏览器，并且通过JavaScript在客户端操作它。

IntraWeb中还有其他的客户端组件，其中包括IWCSLabel、IWCSNavigator以及IWDynamicChart（它只用于IE浏览器）。作为这个方法的一个范例，我们建立了IWClientGrid程序。这个程序具有很少的代码，因为很多已包含在使用的组件中。下面是主窗体的核心元素：

```
object formMain: TFormMain
  SupportedBrowsers = [brIE, brNetscape6]
  OnCreate = IWAppFormCreate
  object IWDynGrid1: TIWDynGrid
    Align = alClient
    Data = IWClientSideDatasetDBLink1
  end
  object DataSource1: TDataSource
```

```

Left = 72
Top = 88
end
object IWClientSideDatasetDBLink1: TIWClientSideDatasetDBLink
    DataSource = DataSource1
end
end
end

```

当该窗体被创建的时候，数据模块的数据集连接到DataSource。最终的网格，如图21.10所示，允许我们对任何单元中的数据进行排序（使用列标题后面的小箭头）并且过滤显示的数据。在图中，我们能够对雇员数据进行排序，例如，根据last name，并且根据国家和工作级别进行过滤。

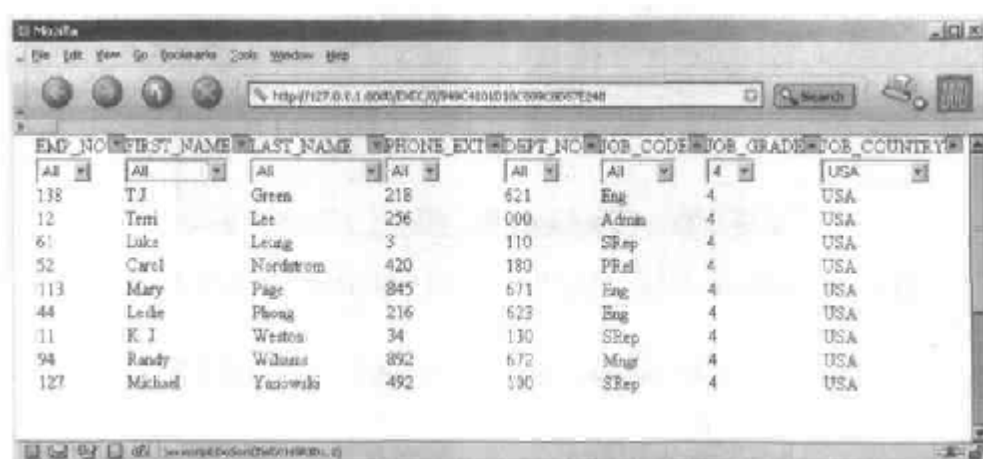


图21.10 IWClientGrid范例的网格支持定制的排序和过滤，而不需要从Web服务器重新取回数据

这个功能是可用的，因为数据在JavaScript代码中即可被移动到浏览器。这里是一个嵌入到HTML页面的脚本代码片断：

```

<script language="Javascript1.2">
var IWDYNGRID1_TitleCaptions =
    [ "EMP_NO", "FIRST_NAME", "LAST_NAME", "PHONE_EXT",
      "DEPT_NO", "JOB_CODE", "JOB_GRADE", "JOB_COUNTRY" ];
var IWDYNGRID1_CellValues = new Array();
IWDYNGRID1_CellValues[0] = [ 2, 'Robert', 'Nelson', '332', '600', 'VP', 2, 'USA' ];
IWDYNGRID1_CellValues[1] = [ 4, 'Bruce', 'Young', '233', '621', 'Eng', 2, 'USA' ];
IWDYNGRID1_CellValues[2] = [ 5, 'Kim', 'Lambert', '22', '130', 'Eng', 2, 'USA' ];
IWDYNGRID1_CellValues[3] = [ 8, 'Leslie', 'Johnson', '410', '180', 'Mktg', 3, 'USA' ];
IWDYNGRID1_CellValues[4] = [ 9, 'Phil', 'Forest', '229', '622', 'Mngr', 3, 'USA' ];

```

说明：之所以使用基于JavaScript的方法而不是其他技术的基于XML的方法，是因为只有IE浏览器支持XML数据。Mozilla以及Netscape缺少这个特性，并且通常对于XML的支持很有限。在JavaScript中进行模拟，就像IE浏览器那样，在运行时是非常昂贵的。

小结

本章对于IntraWeb特性的描述还远不够完善，但是我们的目标主要是对这项技术进行一个评估，以便于读者能够选择是否在他们的Delphi项目中使用这项技术。IntraWeb非常强大，因此我们可以选择Delphi来作为建立Web应用程序的工具，而不需要再求助于其他的各种工具。

读者可以在Delphi随附光碟中找到更多关于IntraWeb的文档。Delphi的默认安装包括IntraWeb演示程序，其中有一些扩展特性的范例显示了这个组件库的大多数特性。还可以参考www.atozedsoftware.com站点，获得更多的更新信息和范例。

在本书中，我们还将要讨论另外一个Web开发方法，它是基于XML和XSLT的。第22章将从Delphi的角度，详细地介绍XML的相关技术。因此，笔者将有机会介绍这项我最喜欢的Web开发技术，虽然它很复杂。

第22章 使用XML技术

为Internet建立应用程序意味着要使用协议并创建基于浏览器的用户接口（如在前两章完成的那样），也意味着创造交换电子事务文档的机会。针对这种类型的行为，所涌现出的标准全集中在XML文档格式上，并包含了SOAP传输协议，用于文档确认的XML方案以及用于将它们作为HTML提交的XSL。

在本章中，我们将介绍核心的XML技术以及从Delphi 6以来所提供的相应扩展支持。由于XML知识并没有大面积地普及，所以我们将只是对它们逐个进行简单的介绍，但读者应该参考有关专著来了解更多的知识。在第23章中，我们将会重点介绍Web服务与SOAP。

本章主要包括以下内容：

- 介绍XML：可扩展标记语言
- 使用XML DOM
- Delphi和XML：接口与映射
- 使用SAX解析XML文档
- Internet Express
- 使用XSLT
- WebSnap中的XSL

XML简介

XML或可扩展标记语言是SGML的简化版本并引起IT界的广泛注意。XML是标记语言（markup language），意味着它使用符号来描述它自己的内容，在这种情况下，标记组成特殊定义的文本，包括在尖括号内。之所以命名它为可扩展的，是因为它允许自由标记（相反HTML有预定义标记）。XML语言是万维网联盟（W3C）提倡的标准。XML建议书位于www.w3.org/TR/REC-xml。

XML被称做是2000年的ASCII，这不仅说明它是一种简洁而普遍的技术，也说明XML文档是明文文件（有选择地用Unicode符号代替普通ASCII文本）。除此之外，XML的重要特点是，它是可描述的，因为每个标记都有一个可读的名称。如果读者从没看过XML文档，可以从下面这个小范例中对其有所了解：

```
<book>
  <title>Mastering Delphi 7</title>
  <author>Cantu</author>
  <publisher>Sybex</publisher>
</book>
```

XML也有缺点，这是从开始就强调的。最大的缺点是若没有正式的描述，文档就没有价值。如果想与其他公司交换文档，就不得不统一每个标记的意思和内容的另一个意思（例

如, 当有一个数量值时, 就不得不统一度量系统或在文档中包含它)。另一个缺点是XML文档比其他格式都大; 例如, 使用字符串效率很低, 重复开关标记占用很大空间。但好在出于相同的原因, XML可以很好地压缩了。

核心XML语法

讨论XML在Delphi中的用法之前, 我们有必要了解一下XML的几个核心技术元素。下面是XML语法关键元素的小结:

- 空白(包含空格字符、回车、换行和跳格)通常被忽视(如在HTML文档中)。这对于通过人工方式格式化XML文档使之具有可读性来说十分重要, 但程序无需关心这些。
- 可以在<!--和-->标记内添加注解, 这通常被XML处理器忽视。也有命令和处理指令, 它们包含在<?和?>标记内。
- 有几个特殊的或保留的字符不能在文本中使用。有两个符号根本不能使用, 那就是小于号(或左尖括号, “<”, 用于定界一个标记)和连字符(&); 前者被<代替, 后者用&代替。其他可选择的特殊字符是>——用于表示大于号(右尖括号, “>”), '——用于表示单引号“'”、"——用于表示双引号“””。
- 为了添加非XML内容(例如, 二进制信息或脚本), 可以使用CDATA段, 包括在“<![CDATA[”和“]]>”内。
- 所有标记被包含在尖括号<与>中。标记是区分大小写的(与HTML相反)。
- 对于每个打开标记, 必须有一个匹配的关闭标记, 由前面加反斜杠符表示, 显示如下:

```
<node>value</node>
```

- 标记不交叠, 它们必须被正确地嵌套, 如下面第一行所示(第二行不正确):

```
<node>xx <nested> yy</nested> </node> // OK
<node>xx <nested> yy</node> </nested> // WRONG
```

- 如果标记没内容(但它的出现很重要), 可以用一个单独的标记代替打开和关闭标记, 它包含了尾符或结束反斜线: <node/>。
- 标记也可以有属性, 用属性名称后跟封闭在引号内的值表示:

```
<node attrib1="aaa">
```

- 任何XML节点可能有多个属性, 多个嵌入标记, 并且只有一个文本块表示节点的值。技术上如果它是可能的话, 则对于XML节点来说应有一个文本值或嵌入标记而不是两个都有。下面这个范例给出了定义一个节点的全部语法:

```
<node attrib1="aaa" attrib2="bbb">
  value1
  <child1>
    value2
  </child1>
</node>
```

- 一个节点可以有多个具有相同标记（标记不惟一）的子节点。对于每个节点来说，属性名是惟一的。

良构的XML

使用前一小节介绍的元素还不足以完整定义XML语法。只有遵循几个额外的规则后，所得XML文档被认为是语法正确或良构的。注意这种检查不能保证文本的内容有意义，而只保证标记被正确放置。

规则之一是每个文档应有一个引言，说明它是真正的XML文档以及它遵守XML的哪个版本和字符编码的可能类型。范例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
```

可能的编码包含各种Unicode字符集（如UTF-8、UTF-16和UTF-32）或某些ISO编码（如ISO-10646-xxx或ISO-8859-xxx）。引言也能包括外部声明，用于确定文档、名称空间声明、相关的XSL文件和一些内部实体声明。关于这些专题的信息参见XML文档或相关书籍。

如果一个XML文档具有引言，它的结构就是完善的，有正确语法（参看前面章节介绍的规则），是一个在单根下有节点的树状结构。多数工具（包括IE浏览器）会在装载它时，检查文档是否是良构的。

说明：正如读者看到的，XML比HTML更正规、准确。W3C正在从事XHTML标准化的工作，这个标准使HTML文档与XML兼容，以便于更好地用XML工具进行处理。在典型的HTML文档中，这暗示了许多变化，如避免无值属性，添加所有闭合标记（如</p>和），添加斜杠到独立标记中（如
和
），适当嵌套等等。W3C网站提供一个从HTML到XHTML的转换器，名为HTML Tidy，请查看W3C网站www.w3.org/People/Raggett/tidy/。

使用XML

为了了解XML格式，可以使用一个能在市场上获得的XML编辑器（包括Delphi本身以及Context，一个用Delphi语言编写的程序员的编辑器）。当我们把一个XML文档装载到Internet Explorer浏览器中时，就可以查看它是否正确。这时将在浏览器内看到它的树状结构。

为了加速这个类型的操作，笔者建立了最简单的XML编辑器，简单说它由一个带有XML语法检测功能的备注以及一个与它相连的浏览器组成。XmlEditOne范例有一个带有三页的PageControl控件。第一页，即Settings中有几个组件，可以在这些组件中插入需要进行操作的文件路径与名称（这样，在我们演示程序的扩展时，没使用标准对话框的原因就变得非常清楚）。含有完整文件名的编辑框由路径和文件名自动更新，前提是AutoUpdate复选框被选中。

第二页提供了一个Memo控件，XML文件的文本通过单击两个工具栏按钮进行装载或者保存。当装载文件或每次修改它的文本时，它的内容被装载到DOM，并且由分析器检查它的正确性（处理自己的代码是一件十分复杂的工作）。为了分析代码，我使用了Delphi中可用的XMLDocument组件，基本是计算机上可用的DOM封装器，并由其DOMVendor属性指定。稍后我们将讨论该组件的使用。现在可以将一个字符串列表赋给其XML属性，并激活它，让它分析XML文本，甚至用于报告异常错误。

对于该范例来说，其功能不是很好，因为当键入XML代码时，将有临时的不正确的XML出现。虽然我们不要用户单击按钮来执行验证的操作，但是我们还是希望让它能够连续运行。由于不可能禁止XMLDocument组件引起的分析异常，所以必须在低层工作，在从XMLDocument组件中（在该代码中名为XmlDoc）抽取IXMLDocumentAccess接口后，抽取DOMPersist属性（引用DOM的持续接口）。这时也可以从文档组件中抽取IDOMParseError接口，用以在状态栏中显示错误消息：

```

procedure TFormXmlEdit.MemoXmlChange(Sender: TObject);
var
    eParse: IDOMParseError;
begin
    XmlDoc.Active := True;
    xmlBar.Panels[1].Text := 'OK';
    xmlBar.Panels[2].Text := '';
    (XmlDoc as IXMLDocumentAccess).DOMPersist.loadxml(MemoXml.Text);
    eParse := (XmlDoc.DOMDocument as IDOMParseError);
    if eParse.errorCode <> 0 then
        with eParse do
            begin
                xmlBar.Panels[1].Text := 'Error in: ' + IntToStr (Line) + '.' +
                    IntToStr (LinePos);
                xmlBar.Panels[2].Text := SrcText + ': ' + Reason;
            end;
    end;

```

读者可在图22.1中看到该程序的输出范例，它位于第三页中的XML树状视图旁（对于正确文档）。程序的第三页使用WebBrowser组件建立，它嵌入了Internet Explorer的ActiveX控件。遗憾的是没有直接的方法将XML文本字符串赋给该控件，因此我们必须首先保存文件，然后移动到它的页上，并在浏览器上触发装载的XML（或单击Refresh按钮）。

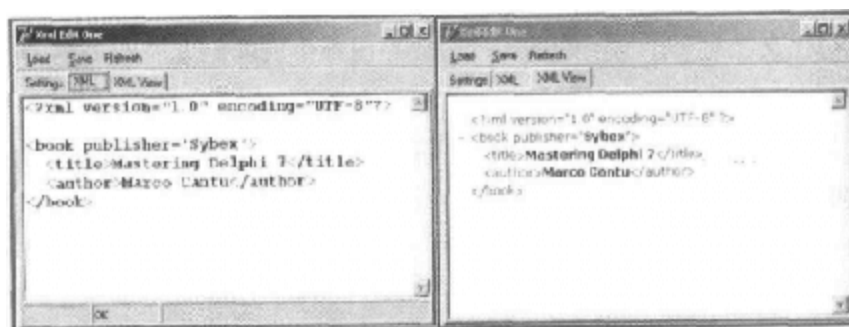


图22.1 XmlEditOne允许我们在一个Memo中输入XML文本，指出输入的错误并且将结果显示在嵌入的浏览器中

说明：为了给一个公司建立完备的、名为XmlTypist的XML编辑器，我使用了这段代码作为开始。它包括语法强调、XSLT支持以及一些额外的特性。参考附录A，可以找到这个免费的XML编辑器。

在Delphi中管理XML文档

现在我们已经知道了XML的核心元素，下面开始讨论在Delphi程序中怎样管理XML文档（或者在普通程序中，因为这里讨论的技术超出了所使用语言的范围）。有两种典型的技术用于管理XML文档，使用文档对象模型（DOM）接口或者是使用XML的Simple API（SAX）。这两种方法完全不同：

- DOM可装载整个文档到一个层次结构的节点树中，允许读取它们并操作它们以改变文档。由于该原因，当我们希望在内存中确定XML结构的位置并编辑它、甚至重新创建新文档时，使用DOM是很适合的。
- SAX用于分析文档，为文档的每个元素激活事件，而不用在内存上建立任何结构。一旦经过SAX分析，文档就会丢失，但通常该操作比DOM树的结构更快。SAX适合一次性读取文件，并可以查找它的部分数据。

第三种处理XML文档的典型方法是：字符串管理。通过添加字符串来创建文档是最快的操作，特别是如果能通过单个关口的话（不需要改变已产生的节点）。即使通过字符串函数读取文档更快，但这个操作难于应对复杂的结构。

除了这些经典的XML处理方法外——这些方法也可以用于其他的编程语言，Delphi 6另外提供了两种技术供我们考虑。第一种用于定义映射文档结构的接口，该接口只用于访问文档而不是通常的DOM接口。读者将看到，该方法可以获得快速编码和更稳定的应用程序。另一个技术用于开发转换过程，即允许读取一般的XML文档到ClientDataSet组件中，或保存数据集到指定结构的XML文件中（不是ClientDataSet或MyBase支持的特定XML结构）。

这里不准备评价哪个选项更适合文档的哪些类型和操作，但在下一节讨论每种方法的范例中，将强调它的一些优缺点。在本章的结尾部分，我们将讨论各种技术处理大文件的速度问题。

用DOM编程

因为XML文档有一个树型的结构，因此在内存中装载XML文档到树中十分适合，这就是文档对象模型所做的事情。DOM是一个标准的接口，以至当编写使用DOM的代码时，可以不用改变源代码就能实现DOM（至少我们不使用任何非定制扩展）。

在Delphi中，我们可安装几个DOM实现，就像COM服务器一样，并使用它们的接口。在Windows中，一个最常用的DOM是Microsoft提供的MSXML SDK的一部分。该DOM也被Internet Explorer使用（出于这个原因在所有最近版本的Windows中都安装了它），但SDK包含一些更详细的文档和范例，这些文档对程序员可能有帮助。其他Delphi 7中可直接使用的DOM引擎包括Apache Foundation的Xerces和开放源代码的OpenXML。

提示：OpenXML是一个本机ObjectPascal DCOM组件，可在www.philo.de/xml上获得。另一个是由TurbPower提供的本机Delphi DOM。这些解决方案的优点是它们不要求用于程序执行的外部库，因为DOM组件被编入我们的应用程序；而且它们是跨平台的。

Delphi将DOM的实现代码封装到一个名为XMLDocument的包装组件中。该组件已在前面的范例中使用，但这里我将以更一般的方式检测它的作用。使用该组件代替实际DOM接

口，程序员可以更独立于各种实现，也可使用简单的方法或者帮助器。

事实上，DOM接口使用起来十分复杂。文档是节点的集合，每个文档有名字、文本元素、属性集合和子节点集合。每个节点集合允许通过位置或名字查找来访问元素。注意，在节点标记内的文本作为节点子集发送并列在其子节点集合中。根节点有额外对象方法用于创建新节点、值或属性。

使用Delphi的XMLDocument，我们实际上可以工作在两个不同层次上：

- 在较低层次，我们能使用DOMDocument属性（IDOMDocument接口类型的一种）来访问标准的W3C文档对象模型接口。官方DOM被定义在xmldom单元中，并包含像IDOMNode、IDOMNodeList、IDOMAttr、IDOMElement和IDOMText这样的接口。使用官方DOM接口，Delphi支持较低层次，标准编程模型。注意，实际的DOM实现是XMLDocument组件在DOMVender属性中指明的。
- 在较高层次，XMLDocument组件也可实现XMLDocument接口。这是由Borland公司定义在XMLIntf单元中的类似于DOM的定制接口，并且组成了诸如IXMLNode、LXMLNodeList和IXMLNodeCollection之类的接口。该Borland接口通过替换多个对象方法和调用，简化了DOM操作，这些被替换的方法调用通常可能需要重复地使用某个单独的属性或方法。

在下面的范例中（特别是在DomCreate演示中），将使用两种方法，因此读者可以对这两种方法的实际区别有一个很好的了解。

TreeView中的XML文档

起始点一般始于从文件中加载文档或从字符串中创建文档，但也可以从一个新文档开始。作为使用DOM的第一个新范例，这里建立了能装载XML文档到DOM中的程序，并在TreeView控件中显示它的结构。此外，还向XmlDomTree程序添加了几个含有样本代码的按钮，用以访问样本文件元素，演示如何访问DOM数据。实际上装载文件很简单，而在树上显示它需要使用递归函数确定节点和子节点的位置。下面是这两个对象方法的代码：

```

procedure TFormXmlTree.btnLoadClick(Sender: TObject);
begin
    OpenFileDialog1.InitialDir := ExtractFilePath (Application.ExeName);
    if OpenFileDialog1.Execute then
        begin
            XMLDocument1.LoadFromFile(OpenDialog1.FileName);
            Treeview1.Items.Clear;
            DomToTree (XMLDocument1.DocumentElement, nil);
            TreeView1.FullExpand;
        end;
    end;

procedure TFormXmlTree.DomToTree (XmlNode: IXMLNode; TreeNode: TTreeNode);
var
    I: Integer;
    NewTreeNode: TTreeNode;

```

```

NodeText: string;
AttrNode: IXMLNode;
begin
    // skip text nodes and other special cases
    if XmlNode.NodeType <> ntElement then
        Exit;
    // add the node itself
    NodeText := XmlNode.NodeName;
    if XmlNode.IsTextElement then
        NodeText := NodeText + ' = ' + XmlNode.NodeValue;
    NewTreeNode := TreeView1.Items.AddChild(TreeNode, NodeText);
    // add attributes
    for I := 0 to xmlNode.AttributeNodes.Count - 1 do
    begin
        AttrNode := xmlNode.AttributeNodes.Nodes[I];
        TreeView1.Items.AddChild(NewTreeNode,
            '[' + AttrNode.NodeName + ' = ' + AttrNode.Text + ']' );
    end;
    // add each child node
    if XmlNode.HasChildNodes then
        for I := 0 to xmlNode.ChildNodes.Count - 1 do
            DomToTree (xmlNode.ChildNodes.Nodes [I], NewTreeNode);
    end;
end;

```

这段代码十分有趣，因为它强调了可以用于DOM的一些操作。首先，每个节点都有 **NodeType** 属性，可用于决定元素、属性、文本代码或特殊实体（如CDATA和其他）。另一方面，不能访问节点的文本表示，亦即它的 **NodeValue**，除非它有文本元素（注意在每次初始测试时，文本节点将被忽略）。在显示了项目名之后，如果文本值可获得，程序（如图22.2所示）将递归调用 **DomToTree** 对象方法直接显示每个属性与每个子节点的内容。

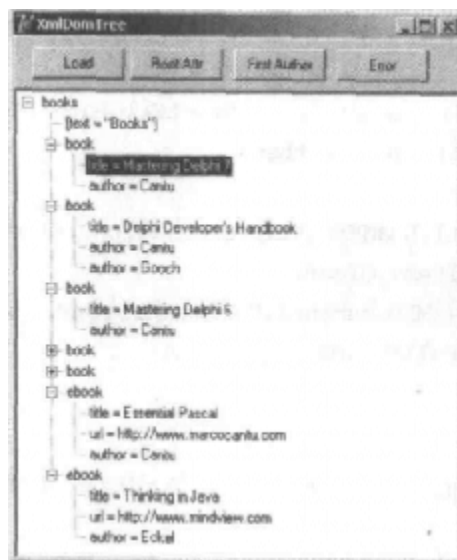


图22.2 XmlDomTree范例可以打开一个普通的XML文档，并且在一个TreeView通用控件中显示它

一旦将XmlDomTree程序（显示在程序清单22.1中）的范例文档装载到XMLDocument组件中，就可以使用各种对象方法访问一般节点，如上面建立树代码那样，或获得特殊元素。例如，通过编写下列代码，获得根节点属性text的值：

```
XMLDocument1.DocumentElement.Attributes ['text']
```

注意如果没有名为text的属性，调用将失败并显示错误消息“无效变量类型转换”。它既不能帮助程序员，也不能帮助最终用户来理解错误。如果需要访问根的第一个属性而不知道它的名字，就可使用下面的代码：

```
XMLDocument1.DocumentElement.AttributeNodes.Nodes[0].NodeValue
```

为了访问实际节点，需要使用类似技术，发挥ChildValues数组的作用。该数组是Delphi对DOM的扩展，它允许把元素的名字或它的数字位置作为参数进行传递：

```
XMLDocument1.DocumentElement.ChildNodes.Nodes[1].ChildValues['author']
```

该代码用来获得第二本书的（第一）作者。这里不能使用ChildValues['book']表达式，因为它们在根节点下有多个节点具有相同的名字。

程序清单22.1 本章范例中使用的XML文档

```
<?xml version="1.0" encoding="UTF-8"?>
<books text="Books">
  <book>
    <title>Mastering Delphi 7</title>
    <author>Cantu</author>
  </book>
  <book>
    <title>Delphi Developer's Handbook</title>
    <author>Cantu</author>
    <author>Gooch</author>
  </book>
  <book>
    <title>Delphi COM Programming</title>
    <author>Harmon</author>
  </book>
  <book>
    <title>Thinking in C++</title>
    <author>Eckel</author>
  </book>
  <ebook>
    <title>Essential Pascal</title>
    <url>http://www.marcocantu.com</url>
    <author>Cantu</author>
  </ebook>
  <ebook>
    <title>Thinking in Java</title>
    <url>http://www.mindview.com</url>
```

```

    <author>Eckel</author>
  </ebook>
</books>
Creating Documents Using the DOM

```

使用DOM创建文档

尽管以前提到, 能通过一起链接一些字符串创建XML文档, 但这不是一种稳定的技术。使用DOM创建文档可确保XML是良构的。如果DOM有一个相关的模式定义, 则当向它添加数据时, 就可确定文档的结构。

为了强调文档创建的不同情况, 我们建立了DomCreate范例。该程序能在DOM内创建XML文档, 在Memo上或者选择用TreeView显示它们的文本。

警告: XmlDocument组件使用doAutoIndent选项通过一种稍好的方法格式化XML来改进XML文本对Memo的输出。可以设置NodeIndentStr属性并选择缩进类型。为了格式化一般XML文本也可使用全局FormatXMLData函数将默认值设置作为缩进。奇怪的是没有一种方法可以传递不同参数到函数。

窗体上的Simple按钮使用低层次的官方DOM接口创建一些简单XML文本。该程序要为每个节点调用文档的createElement对象方法, 添加它们作为其他节点的子节点:

```

procedure TForm1.btnSimpleClick(Sender: TObject);
var
  iXml: IDOMDocument;
  iRoot, iNode, iNode2, iChild, iAttribute: IDOMNode;
begin
  // empty the document
  XMLDoc.Active := False;
  XMLDoc.XML.Text := '';
  XMLDoc.Active := True;

  // root
  iXml := XmlDoc.DOMDocument;
  iRoot := iXml.appendChild (iXml.createElement ('xml'));
  // node "test"
  iNode := iRoot.appendChild (iXml.createElement ('test'));
  iNode.appendChild (iXml.createElement ('test2'));
  iChild := iNode.appendChild (iXml.createElement ('test3'));
  iChild.appendChild (iXml.createTextNode('simple value'));
  iNode.insertBefore (iXml.createElement ('test4'), iChild);

  // node replication
  iNode2 := iNode.cloneNode (True);
  iRoot.appendChild (iNode2);

  // add an attribute
  iAttribute := iXml.createAttribute ('color');
  iAttribute.nodeValue := 'red';
  iNode2.attributes.setNamedItem (iAttribute);

```

```
// show XML in memo
Memo1.Lines.Text := FormatXMLData (XMLDoc.XML.Text);
end;
```

注意文本节点被显式地添加，用特定创建调用建立属性并且使用cloneNode代码复制整个树的分支。总之，代码写起来很烦琐，需要一段时间适应该类型。程序的结果（在Memo和树中格式化）如图22.3所示。

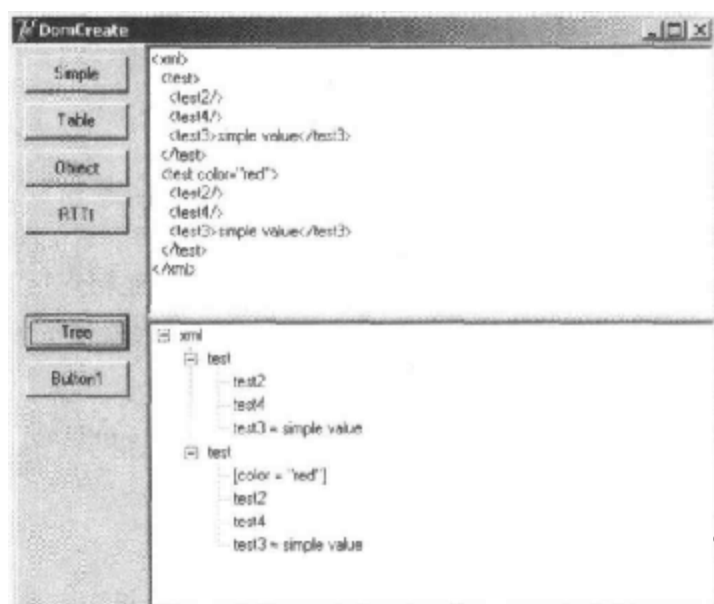


图22.3 DomCreate范例能够使用DOM生成各种类型的XML文档

第二个DOM示例与数据集有关。可为窗体添加一个BDETable组件（任何其他数据集都这样做），并为按钮添加对定制DataSetToDOM过程的调用，代码如下：

```
DataSetToDOM ('customers', 'customer', XMLDoc, SQLDataSet1);
```

DataSetToDOM过程用文本的第一个参数创建根节点，然后获取数据集的每个记录，用第二个参数定义节点，并且为记录的每个字段添加了节点，其相关代码都很普通：

```
procedure DataSetToDOM (RootName, RecordName: string; XMLDoc: TXMLDocument;
  DataSet: TDataSet);
var
  iNode, iChild: IXMLNode;
  i: Integer;
begin
  DataSet.Open;
  DataSet.First;
  // root
  XMLDoc.DocumentElement := XMLDoc.CreateNode (RootName);
  // add table data
  while not DataSet.EOF do
  begin
    // add a node for each record
```

```

iNode := XMLDoc.DocumentElement.AddChild (RecordName);
for I := 0 to DataSet.FieldCount - 1 do
begin
    // add an element for each field
    iChild := iNode.AddChild (DataSet.Fields[i].FieldName);
    iChild.Text := DataSet.Fields[i].AsString;
end;
DataSet.Next;
end;
DataSet.Close;
end;

```

前面的代码使用简单化的DOM访问Borland提供的接口，该接口包含创建子节点的AddChild节点，可直接访问Text属性，使用文本内容定义子节点。这个简单的子例程可以选取数据集的XML表示，也为网络出版打开许多机会，本章将在XSL相关小节中讨论这些内容。

另一个有趣的机会是描述Delphi对象的XML文档。DomCreate程序有用于描述对象可选属性的几个按钮，此时可再一次使用低级的DOM：

```

procedure AddAttr (iNode: IDOMNode; Name, Value: string);
var
    iAttr: IDOMNode;
begin
    iAttr := iNode.ownerDocument.createAttribute (name);
    iAttr.nodeValue := Value;
    iNode.attributes.setNamedItem (iAttr);
end;

procedure TForm1.btnObjectClick(Sender: TObject);
var
    iXml: IDOMDocument;
    iRoot: IDOMNode;
begin
    // empty the document
    XMLDoc.Active := False;
    XMLDoc.XML.Text := '';
    XMLDoc.Active := True;

    // root
    iXml := XmlDoc.DOMDocument;
    iRoot := iXml.appendChild (iXml.createElement ('Button1'));

    // a few properties as attributes (might also be nodes)
    AddAttr (iRoot, 'Name', Button1.Name);
    AddAttr (iRoot, 'Caption', Button1.Caption);
    AddAttr (iRoot, 'Font.Name', Button1.Font.Name);
    AddAttr (iRoot, 'Left', IntToStr (Button1.Left));
    AddAttr (iRoot, 'Hint', Button1.Hint);

```



```
// show XML in memo
Memo1.Lines := XmlDoc.XML;
end;
```

当然，更有趣的是使用一项简单的技术来保存每一个Delphi组件的属性（或者更准确地说，是持续对象），亦即在持续子对象上进行递归并且指明引用组件的名称。我们已在ComponentToDOM过程中实现了这一点，它使用由TypeInfo单元提供的低层次的RTTI信息，包括提取没有默认值的组件属性列。再一次，程序使用了简化的Delphi XML接口：

```
procedure ComponentToDOM (iNode: IXmlNode; Comp: TPersistent);
var
  nProps, i: Integer;
  PropList: PPropList;
  Value: Variant;
  newNode: IXmlNode;
begin
  // get list of properties
  nProps := GetTypeData (Comp.ClassInfo)^.PropCount;
  GetMem (PropList, nProps * SizeOf(Pointer));
  try
    GetPropInfos (Comp.ClassInfo, PropList);
    for i := 0 to nProps - 1 do
      if not IsDefaultPropertyValue(Comp, PropList [i], nil) then
        begin
          Value := GetPropValue (Comp, PropList [i].Name);
          NewNode := iNode.AddChild(PropList [i].Name);
          NewNode.Text := Value;
          if (PropList [i].PropType^.Kind = tkClass) and (Value <> 0) then
            if TObject (Integer(Value)) is TComponent then
              NewNode.Text := TComponent (Integer(Value)).Name
            else
              // TPersistent but not TComponent: recurse
              ComponentToDOM (newNode, TObject (Integer(Value)) as TPersistent);
        end;
      finally
        FreeMem (PropList);
      end;
    end;
end;
```

在此例中，下面两行代码用于触发XML文档的产生（如图22.4所示）：

```
XMLDoc.DocumentElement := XMLDoc.CreateNode(Self.ClassName);
ComponentToDOM (XMLDoc.DocumentElement, Self);
```

XML数据绑定接口

读者会看到用DOM访问数据或生成文档相当单调，因为必须使用位置信息和不合理的访问数据。而且处理一系列不同类型的（如程序清单22.1中显示的）重复节点也不简单。此

外,使用DOM可以创建任何良构的文档,但(除非使用正在验证的DOM)添加任意子节点到任意节点所得到的文档几乎没有任何用处,因为没人能管理它们。

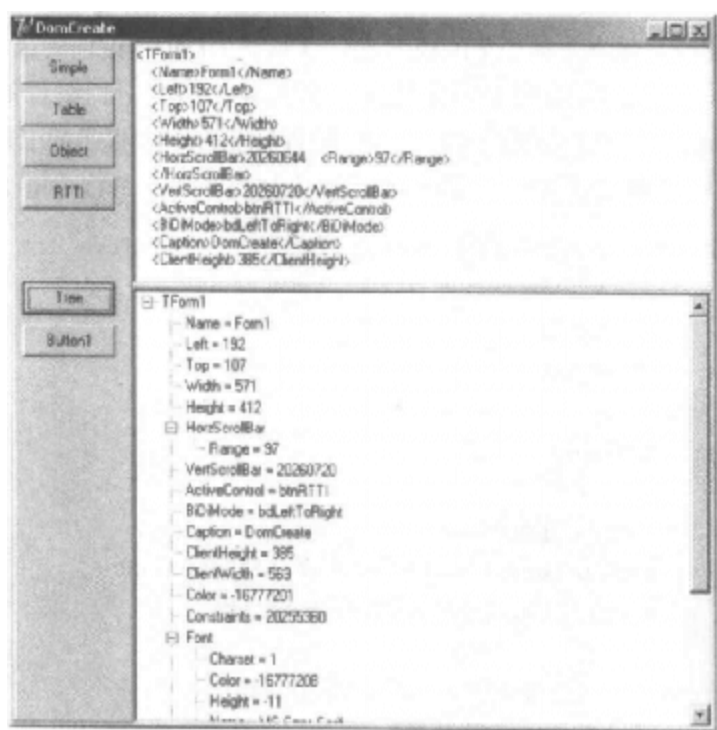


图22.4 为描述DomCreate程序的窗体而生成的XML。请注意在树形图以及Memo文本中,类的属性类型被进一步扩展

为了解决这些问题,Borland对Delphi添加了XML Data Binding Wizard,该向导能检测XML文档或文档定义(模式、文本类型定义即DTD或定义的其他类型),并创建一套接口来处理文档。这些接口对文档和它的结构是特殊的,允许程序员有更易读的代码,但它们远非程序员可以处理的文档类型那么通俗(这比开始的情况要好得多)。

可以使用IDE的New Items对话框首页中对应的图标或直接在XMLDocument组件上双击来激活XML Data Binding Wizard(奇怪的是对应的命令不在组件的本地菜单上)。

在可以选择输入文件的首页之后,该向导向我们形象地显示了文档的结构,如图22.5所示,其源代码来自于程序清单22.1的XML文件范例。在该页上,如果不想使用向导提供的默认值,可以给出生成接口的每个实体的名字。实际上也能改变用向导产生名字的规则,笔者就喜欢在Delphi IDE的其他地方自由扩展。最后一页提供了一个生成接口的预览,并为生成模式和其他定义文件提供了机会。

对于使用作者名称的XML文件范例,XML Data Binding Wizard会生成根节点的接口和两个用于两种不同节点类型的元素列表的接口,以及两个用于两类元素的接口。下面是生成代码的几个摘录,可在XmlInterface范例的XmlIntfDefinition单元获得:

```
type
  IXMLBooksType = 'interface (IXMLNode)
    { '{C9A9FB63-47ED-4F27-8ABA-E71F30BA7F11}' }
    { Property Accessors }
```

```

function Get_Text: WideString;
function Get_Book: IXMLBookTypeList;
function Get_Ebook: IXMLEbookTypeList;
procedure Set_Text(Value: WideString);
{ Methods & Properties }
property Text: WideString read Get_Text write Set_Text;
property Book: IXMLBookTypeList read Get_Book;
property Ebook: IXMLEbookTypeList read Get_Ebook;
end;

IXMLBookTypeList = interface(IXMLNodeCollection)
[ '{3449E8C4-3222-47B8-B2B2-38EE504790B6}' ]
{ Methods & Properties }
function Add: IXMLBookType;
function Insert(const Index: Integer): IXMLBookType;
function Get_Item(Index: Integer): IXMLBookType;
property Items[Index: Integer]: IXMLBookType read Get_Item; default;
end;

IXMLBookType = interface(IXMLNode)
[ '{26BF5C51-9247-4D1A-8584-24AE68969935}' ]
{ Property Accessors }
function Get_Title: WideString;
function Get_Author: IXMLString_List;
procedure Set_Title(Value: WideString);
{ Methods & Properties }
property Title: WideString read Get_Title write Set_Title;
property Author: IXMLString_List read Get_Author;
end;

```

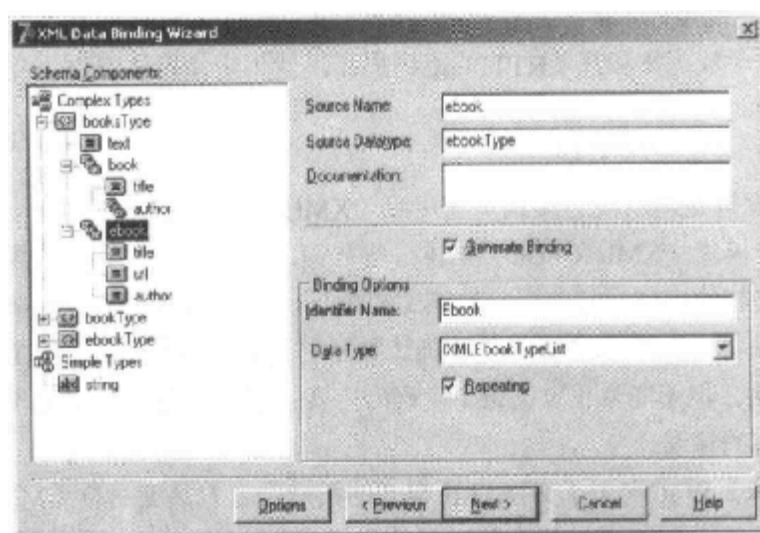


图22.5 Delphi的XML Data Binding Wizard能够检查一个文档或者模式（另一个文档的定义）的结构，以创建一系列用于简化和直接访问DOM数据的接口

对于每一个接口来说,XML Data Binding Wizard也要生成一个实现类,通过将请求翻译为DOM调用来提供接口方法的代码。这个单元包括三个初始化函数,可以从一个装入XMLDocument组件(或者一个提供通用IXMLDocument接口的组件)中的文档返回根节点的接口,或者返回一个文件,或者生成一个全新的DOM:

```
function Getbooks(Doc: IXMLDocument): IXMLBooksType;  
function Loadbooks(const FileName: WideString): IXMLBooksType;  
function Newbooks: IXMLBooksType;
```

在XmlInterface范例中,在使用向导生成这些接口之后,我们重复了XML文档的访问代码,它类似于XmlDomTree范例,但是更加简单,易于阅读。例如,可以编写下面的代码从根节点获取属性:

```
procedure TForm1.btnAttrClick(Sender: TObject);  
var  
    Books: IXMLBooksType;  
begin  
    Books := Getbooks (XmlDocument1);  
    ShowMessage (Books.Text);  
end;
```

如果我们在输入代码的时候重新调用它,情况甚至会变得更加简单。Delphi的代码观察器可以通过列举每个节点的可用属性来协助工作。这要归功于分析器可以读取接口定义的事实(虽然它不能够理解通用XML文档的格式)。访问子列表中的一个节点实际上是编写下面的语句(可能是第二个,带有默认的数组属性):

```
Books.Book.Items[1].Title // full  
Books.Book[1].Title       // further simplified
```

我们可以类似地简化代码,以生成新的文档或者添加新的元素,在每一个基于列表的接口中可用的定制Add对象方法起了很大作用。再者,如果没有XML文档的预定义结构,像前面演示的基于数据集和基于RTTI的范例那样,就不能使用该方法。

确认和模式

XML数据绑定能从现有模式中使用或为XML文档(最终可在文件中用XDB扩展保存它)生成一个模式。XML文档用于描述一些数据,但在公司间交换该数据必须遵从某些结构。模式是文档的定义而且能用来检测文档的正确性,操作通常用validation术语说明。

首先,从广义上讲,XML可用的确认类型使用文档类型定义(DTD)。这些文档描述了XML的结构,但不能真正定义每个代码的内容。DTD本身不是XML文档,但它使用一个不同的难理解的符号。

2000年末,W3C宣布了支持XML模式的第一个官方草案(位于Microsoft的名为XML-Data的DOM内)。XML模式本身是XML文档,它能确认XML树的结构和节点的内容。模式通常基于简单与复杂数据类型的使用和定义,类似于OOP编程语言中的相关内容。

模式用于定义复杂的类型,譬如指出每个可能的节点,它们的可选顺序(sequence, all),每个子节点的发生编号(minOccurs, maxOccurs)和每个特定元素的数据类型。下面

是通过XML Data Binding Wizard定义的XML，用于图书文件范例：

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xdb="http://www.borland.com/schemas/delphi/7.0/XMLDataBinding">
  <xs:element name="books" type="booksType"/>
  <xs:complexType name="booksType">
    <xs:annotation>
      <xs:appinfo xdb:docElement="books"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="book" type="bookType" maxOccurs="unbounded"/>
      <xs:element name="ebook" type="ebookType" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="text" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="bookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="author" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ebookType">
    <xs:annotation>
      <xs:appinfo xdb:repeated="True"/>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="url" type="xs:string"/>
      <xs:element name="author" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

微软和Apache DOM的引擎可以很好地支持模式。另一个控件是XSV（XML模式确认器），它是在适应模式敏感处理器方面所做的开放源代码尝试，能从Web上直接使用，也可下载为可在命令行中执行的命令（参见W3C的XML Schema页面中关于这个工具的链接）。

说明：Delphi的编辑器通过DTD来支持XML文件的代码完成功能。丢弃Delphi的bin目录下的DTD文件并且使用DOCTYPE标记来引用它将会启动这个特性，但是目前Borland并未给它提供官方支持。

使用SAX API

Simple API for XML或称SAX并不会为XML节点创建树，而是简单分析节点——每个节点、属性、值等等的触发事件。因为它不在内存中保存文档，因而使用SAX允许管理更大的文档。它的办法对于一次性检测文档或检索特殊信息很有用。下面列出可由SAX触发的重要事件：

- 用于整个文档的StartDocument和EndDocument
- 用于每个节点的StartElement和EndElement
- 用于节点内文本的Characters

使用堆栈在节点树内处理当前路径十分普遍，通常要为每个StartElement和EndElement事件在堆栈中压入与弹出元素。

Delphi不提供对SAX接口的特殊支持，但可以导入Microsoft的XML支持（MSXML库）。在SaxDemo1范例中我们用的是MSXML版本2，因为这个版本使用广泛。我们已经从该类型库生成了Pascal类型库的导入单元，该导入单元在程序的源代码内，但必须先要在计算机上注册这一特殊的COM库，才能成功地运行程序。

说明：本章末尾的另一个范例（LargeXml）演示了SAX API的用法，包括OpenXml引擎。

为了使用SAX，必须在SAX读取器内安装SAX事件处理器，然后装载文件并分析它。我使用的是MSXML为程序员提供的SAX读取器接口。该官方（C++）接口在其类型库中有几处错误，以防止Delphi可以正确地将它导入。SaxDemo1范例的主窗体做了下述声明：

```
sax: IVBSAXXMLReader;
```

在FormCreate对象方法中，Sax变量被实际COM对象初始化：

```
sax := CoSAXXMLReader.Create;
sax.ErrorHandler := TMySaxErrorHandler.Create;
```

该代码也设置了错误处理器，这是一个类，可使用三个方法实现特殊的接口IVBSAXErrorHandler，至于使用哪一个方法取决于问题的严肃性：error、fatalError和ignorableWarning。

稍微简化一下代码，SAX分析器在将内容处理器赋给它后，通过调用parseURL对象方法被激活：

```
sax.ContentHandler := TMySaxHandler.Create;
sax.parseURL (filename)
```

因此，这些代码最终放在TMySaxHandler类中，它具有SAX事件。因为我们在这个范例中有多个SAX内容处理器，所以我使用核心代码以及一些特殊版本编写了一个基类，以完成特定的操作。下面是基类的代码，它实现了为IVBSAXContentHandler接口提供基础的IVBSAXContentHandler接口和IDispatch接口：

```
type
  TMySaxHandler = class (TInterfacedObject, IVBSAXContentHandler)
  protected
    stack: TStringList;
  public
    constructor Create;
```

```

destructor Destroy; override;
// IDispatch
function GetTypeInfoCount(out Count: Integer): HRESULT; stdcall;
function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
    HRESULT; stdcall;
function GetIDsOfNames(const IID: TGUID; Names: Pointer;
    NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT; stdcall;
function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
    Flags: Word; var Params; var Result, ExcepInfo, ArgErr: Pointer):
    HRESULT; stdcall;
// IVBSAXContentHandler
procedure Set_documentLocator(const Param1: IVBSAXLocator);
    virtual; safecall;
procedure startDocument; virtual; safecall;
procedure endDocument; virtual; safecall;
procedure startPrefixMapping(var strPrefix: WideString;
    var strURI: WideString); virtual; safecall;
procedure endPrefixMapping(var strPrefix: WideString); virtual; safecall;
procedure startElement(var strNamespaceURI: WideString;
    var strLocalName: WideString; var strQName: WideString;
    const oAttributes: IVBSAXAttributes); virtual; safecall;
procedure endElement(var strNamespaceURI: WideString;
    var strLocalName: WideString; var strQName: WideString);
    virtual; safecall;
procedure characters(var strChars: WideString); virtual; safecall;
procedure ignorableWhitespace(var strChars: WideString);
    virtual; safecall;
procedure processingInstruction(var strTarget: WideString;
    var strData: WideString); virtual; safecall;
procedure skippedEntity(var strName: WideString); virtual; safecall;
end;

```

当然，最有意思的部分是最后的SAX事件列表。所有这个基类做的事情就是生成信息来记录分析器在什么时候启动（startDocument）、在什么时候结束（endDocument），并且使用一个堆栈追踪当前节点以及它的父节点：

```

// TMySaxHandler.startElement
stack.Add (strLocalName);
// TMySaxHandler.endElement
stack.Delete (stack.Count - 1);

```

TMySimpleSaxHandler类提供了一个实现方式，覆盖了为任何新节点触发的startElement事件，用以输出在树中的当前位置，语句为：

```
Log.Add (strLocalName + '(' + stack.CommaText + ')');
```

该类的第二个方法是characters事件，它在遇到一个节点值（或一个测试节点）时被触发，并且输出它的内容，如图22.6所示：

```

procedure TMySimpleSaxHandler.characters(var strChars: WideString);
var
    str: WideString;
begin
    inherited;
    str := RemoveWhites (strChars);
    if (str <> '') then
        Log.Add ('Text: ' + str);
end;

```

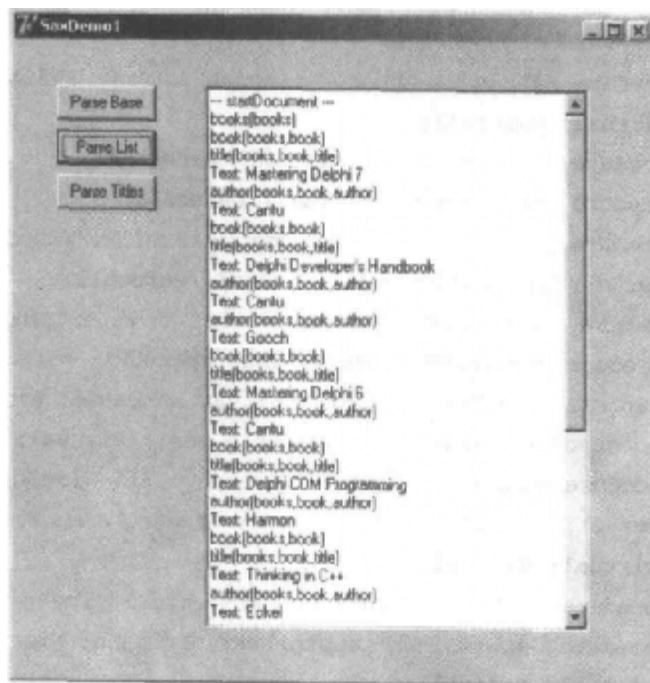


图22.6 在SaxDemo1范例中，通过读取一个带有SAX的XML文档产生的日志信息

一般的分析操作仍将影响整个XML文件。相反，来自SAX内容处理器的第二个类要引用XML文档的特殊结构，抽取给定类型的惟一节点。特别是该程序要查找title类型的节点。当节点有该类型时（在startElement中），此类将设置Boolean类型的isbook布尔变量。节点的文本值在遇到该类型的节点后被认为是惟一正确的：

```

procedure TMyBooksListSaxHandler.startElement(var strNamespaceURI,
    strLocalName, strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
    inherited;
    isbook := (strLocalName = 'title');
end;

procedure TMyBooksListSaxHandler.characters(var strChars: WideString);
var
    str: string;
begin
    inherited;

```



```

if isbook then
begin
    str := RemoveWhites (strChars);
    if (str <> '') then
        Log.Add (stack.CommaText + ': ' + str);
    end;
end;

```

使用转换映射XML

在Delphi中,用以处理XML文档的技术不只一种。可以创建一种转换,在把数据存入MyBase XML文件中时,将XML的一般文档翻译为ClientDataSet使用的格式。与之相反,还需要另一种转换,用于将ClientDataSet(通过DataSetProvider组件)中可用的数据集转换为要求格式(或模式)的XML文件。

Delphi中包含产生这种转换的向导,名为XML映射工具或简称XML映射器,可从IDE的工具菜单调用或作为单独的应用程序执行。XML映射器,如图22.7所示,是设计时的帮助器,协助程序员在一般XML文档的节点和ClientDataSet数据包的字段间定义转换规则。

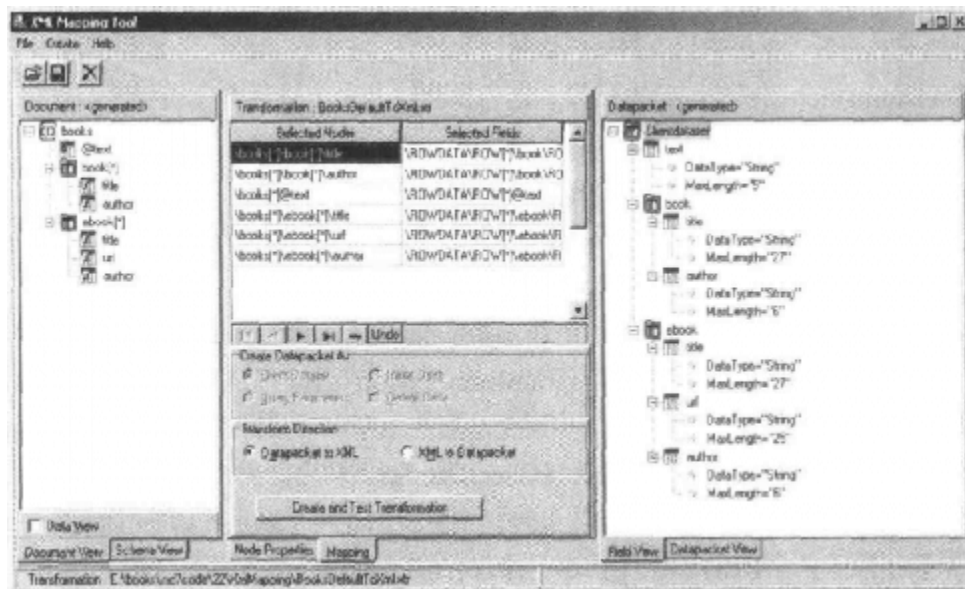


图22.7 XML映射器显示了转换的双方,以定义一个它们之间的映射(使用中间部分指明的规则)

XML映射器窗口由三部分组成:

- 左边是XML文档部分,它在Document View或Schema View的XML模式上显示关于XML文档结构的信息(及其数据,如果相关的检测框被激活的话),显示的内容取决于选中的选项卡。
- 右边是数据包部分,显示数据包中的元数据信息——要么在Field View中显示数据集结构,要么在Datapacket View中报告XML结构。注意实际上,XML映射器也能以本机ClientDataSet格式打开文件。

- 窗口的中间部分是映射部分。这部分包括两页：一个用于映射，在两边的所选元素间能看到的对应项目就是映射；一个用于节点属性，可修改数据类型和其他每个可能映射的细节。

中间面板中的映射页也支持用于产生转换的本地菜单，而其他面板和页中有特殊的本地菜单用来执行各种操作（除了几个在主菜单中的命令）。

可以使用XML映射器映射现存模式到新数据包，映射现存数据包到新模式或文档，或映射现存数据包到现存XML文档（如果匹配合理）。除了转换XML文件的数据到数据包外，也能转换ClientDataSet的增量包。这对于合并文档到现存表格是有益的，就好像用户插入它们一样。特别是能把XML文档变成增量包，用于记录被改变、删除或插入的记录。

使用XML映射器将产生一个或多个转换文件，每个都显示一种转换（因此至少需要两个转换文件来往复地转换文档）。然后这些转换文档在设计时和运行时被XMLTransform、XMLTransformProvider和XMLTransformClient组件使用。

例如，我曾试图打开一般的“books”XML文档，该XML有一个不容易与表格匹配的结构，因为它有两个不同类型值的列表（我们跳过了简单的范例，在那些范例中，XML具有一个普通的矩形结构）。在XML文档部分打开Sample.XML文件后，使用它的本地菜单来选择所有元素（Select All），并创建数据包（Create Datapacket From XML）。这里用数据包自动填充右面板，并用被提议的转换填充中间部分。也能在范例程序上通过选择Create And Test Transformation按钮立即浏览结果。这样做会打开一个普通的应用程序，该应用程序能使用我们已创建的转换将文档装载到数据集。

在这种特殊情况下，读者能看到XML映射器用两个数据集字段产生表格，每个都可能是子元素列表。这是惟一可能的标准解决方案，因为两个子列表有不同的结构，并且允许我们在DBGrid上编辑数据链接到ClientDataSet，同时保存它返回到完整的XML文件，如XmlMapping范例说明的那样。该程序基本是复杂XML文档的基于Windows的编辑器。

范例使用有两个转换文件连接的TransformProvider组件，在XML文档中实现读取功能，并使ClientDataSet组件可以使用它。事实上从它的名字即可看出，该组件是数据集供应器。为了建立用户接口，我们并未直接连接ClientDataSet到网格，因为它含有带文本字段和详细数据集的单个记录。由于该原因，这里向程序中添加了两个以上的ClientDataSet组件，使其与数据集字段和两个DBGrid控件相连。通过它的DFM源代码更容易理解其作用，它的输出如图22.8所示：

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'BooksDefault.xtr'
  TransformWrite.TransformationFile = 'BooksDefaultToXml.xtr'
  XMLDataFile = 'Sample.xml'
end
object ClientDataSet1: TClientDataSet
  ProviderName = 'XMLTransformProvider1'
  object ClientDataSet1text: TStringField
  object ClientDataSet1book: TDataSetField
  object ClientDataSet1lebook: TDataSetField
end
```

```

object ClientDataSet2: TClientDataSet
    DataSetField = ClientDataSet1book
end
object ClientDataSet3: TClientDataSet
    DataSetField = ClientDataSet1ebook
end

```

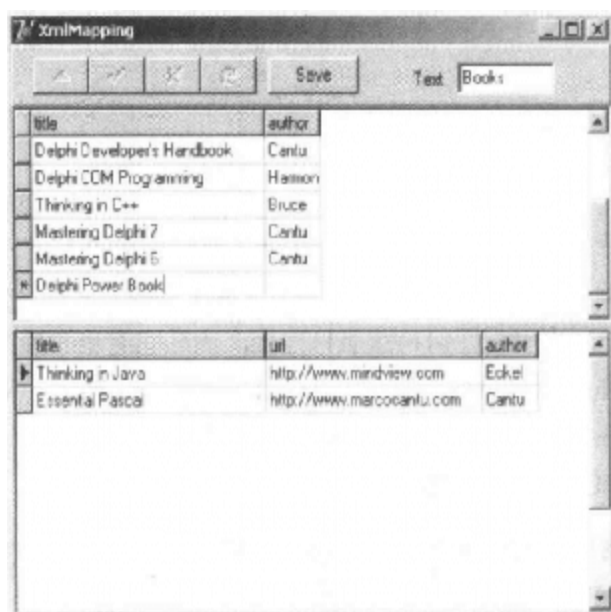


图22.8 XmlMapping范例使用了TransformProvider组件来创建复杂的XML文档，可以在多个ClientDataSet组件中进行编辑

该程序允许在网格内编辑各种节点子列表的数据，修改它们，添加或删除记录。当把发生的改变应用到数据集时（单击Save按钮，调用ApplyUpdates），转换供应器会把文件的更新版本保存到磁盘。

另一种方法是创建一种转换，只映射部分XML文档到数据集。例如XmlMapping范例文件夹中的BooksOnly.xtr文件所示。将要生成的修改后的XML文档与原文档的结构和内容均不同，但只包括用户选择的部分。因此，它能被用于浏览数据但不能编辑它。

说明：转换文件本身就是XML文档一点也不奇怪，因为用户可在编辑器中打开并查看它。该XML文档使用定制格式。

相反读者可以看到转换怎样用于构成数据库表格或查询的结果，并产生XML文件，该文件的可读性比默认情况下用ClientDataSet持续机制提供的更好。为了建立MapTable范例，我在窗体中放置了一个dbExpress SimpleDataSet组件，并将它连接到一个DataSetProvider和一个连接到这个供应器的ClientDataSet上。在打开表格和客户数据集后，就将它的内容保存到一个XML文件中。

在这里，我还打开了XML映射器，将数据包文件装入其中，选择所有数据包节点（用快捷菜单中的Select All命令）并通过Datapacket命令激活Create XML。在随后的对话框中，使用字段的默认名映射并只为记录节点建议一个更可读的名字。如果现在测试转换，XML映射器将在定制的TreeView中显示结果XML文档的内容。

在保存完转换文件后,重新开始开发程序,亦即删除ClientDataSet并且添加一个DBGrid (因为在转换它前,用户能在附加DBGrid上编辑它)以及一个XMLTransformClient组件。该组件与转换文件相连,而非XML文件。事实上,它要通过供应器引用数据。单击按钮将显示备注中的XML文档(经过格式化),而不是将它保存到文件中;通过调用GetDataAsXml对象方法也可做其他一些事情(甚至在HELP文件也很不清楚如何使用该方法的情况下):

```
procedure TForm1.btnMapClick(Sender: TObject);
begin
    Memo1.Lines.Text := FormatXmlData(XMLTransformClient1.GetDataAsXml(''));
end;
```

程序在运行时惟一可见的代码如图22.9所示,我们可以在DBGrid中看到本来的数据集以及在网格下面的Memo控件中显示生成的XML文档。该应用程序比用来产生类似XML文档的DomCreat范例具有更简单的代码,但要求在设计时定义转换。相反,DomCreate范例能在运行时处理任意数据集,而不用与任何特殊表格连接,因为它有更一般的代码。理论上,使用事件的一般XMLTransform组件能产生类似的动态映射,但笔者发现使用前面基于DOM的方法会更简单些。还要注意调用FormatXmlData会产生漂亮的输出,但也会减慢程序运行的速度,因为它需要将XML装载到DOM中。

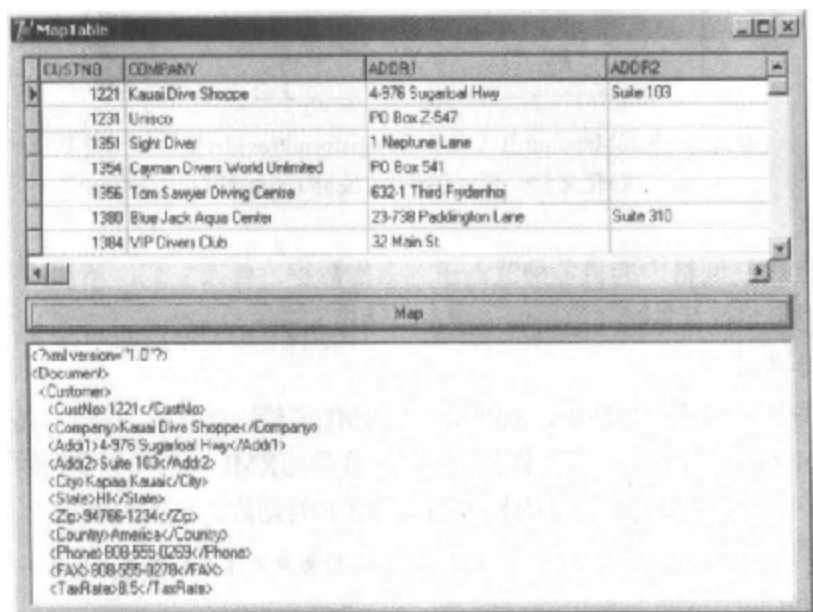


图22.9 MapTable范例使用定制转换文件从数据库表格生成了一个XML文档

XML和InternetExpress

一旦定义了XML文档的结构,就可以让用户在Windows应用程序中或从Web上看到并编辑数据。第二个原因更有趣,Delphi对它提供了特殊支持。Delphi 5包含名为Internet Express的结构,该结构现在已成为Delphi 6 WebSnap平台的一部分。WebSnap也支持XSL,这些在将稍后讨论。

第16章讨论了DataSnap应用程序的开发。Internet Express为该结构提供的客户组件名为XMLBroker, 该组件能用于代替客户数据集, 从中间层DataSnap程序检索数据, 并使特殊类型页生成器InetXpageProducer页生成器可以使用它。可以在标准WebBroker应用程序或WebSnap程序中使用这些组件。InternetExpress背后的理念是, 编写网络服务器扩展(第20章曾经介绍过), 随后它可生成连接到DataSnap服务器的网页。定制的应用程序类似于一个DataSnap客户, 可生成用于浏览器客户的页面。Internet Express提供了便于建立该定制应用程序需要的服务。

这听起来难免令人糊涂, 但Internet Express是四级结构: SQL服务器、应用程序服务器(DataSnap服务器)、定制应用程序的网络服务器和网络浏览器。当然, 可以在相同应用程序内放置数据库访问组件, 以处理HTTP请求并生成HTML结果, 像客户机/服务器解决方案那样。甚至能在双层结构中访问局部数据库或XML文件(服务器程序以及浏览器)。

换句话说, Internet Express是一种基于浏览器建立客户程序的技术, 它允许向客户计算机发送整个数据集, 以及一些HTML和一些JavaScript用于管理XML, 并显示它到HTML定义的用户接口。JavaScript能使浏览器显示数据并处理它。

XMLBroker组件

为达到此目的, Internet Express使用了多种技术。将DataSnap数据包转换成XML格式, 即可由程序把该数据嵌入到用于网络客户端管理的HTML页中。实际上, 增量数据包也在XML中被展示。这些操做由XMLBroker组件完成, 该组件能处理XML并向JavaScript组件提供数据。像ClientDataSet一样, XMLBroker包含有:

- MaxRecords属性, 用于说明添加到单页的记录数量
- Params属性, 用于存放参数, 这些参数将由组件通过供应商转发到远程查询
- WebDispatch属性, 用于说明中介器(broker)响应的更新请求

InetXPageProducer允许从数据集生成HTML窗体, 视觉上类似AdapterPageProducer用户接口的开发。实际上, Internet Express结构、它内部使用的接口和一些DE编辑器能一起被WebSnap体系结构的父类考虑。生成可在服务器端和客户端执行的脚本有很大的不同, 但它们都可提供一个编辑器, 用于放置可视组件和生成类似的脚本。笔者个人并不十分愿意旧的Internet Express比新的WebSnap更加面向XML。

提示: InetXPageProducer和AdapterPageProducer的另一个只有特性是均支持Cascading Style Sheets (CSS)。这些组件具有Style和StylesFile属性, 用于定义CSS, 并且每个元素都有供用户选择类型名称的StyleRule属性。

JavaScript支持

为了使客户端的编辑操作更加强大, InetXPageProducer使用了特殊的JavaScript组件和代码。Delphi嵌入了一个大的JavaScript库, 它必须由浏览器下载。这个操作看起来很麻烦, 但是它是强化浏览器接口惟一的办法(基于动态的HTML), 可以在浏览器中支持字段限制以及其他的业务规则。这些功能对于普通的HTML是不可能的。由Borland提供的JavaScript文件, 也是我们应该在网站应用程序上生成的文件, 包括下面这些:

| 文件 | 描述 |
|---------------|-------------------------------------|
| XmlDOM.js | DOM兼容的XML分析器（用于那些缺少本机XML DOM支持的浏览器） |
| XmlDB.js | 用于HTML控件的JavaScript类 |
| XmlDisp.js | 用于绑定XML数据与HTML控件的JavaScript类 |
| XmlErrDisp.js | 用于处理错误的类 |
| XmlShow.js | 用于显示数据和增量包的JavaScript函数（用于调试功能） |

由Internet Express生成的HTML页面通常包括对于这些JavaScript文件的引用，例如：

```
<script language=Javascript type="text/javascript"
src="IncludePathURL/xmldb.js"></script>
```

我们可以定制JavaScript：直接向HTML页面增加代码，或者创建新的符合Internet Express体系结构并且产生JavaScript代码的Delphi组件。例如，INetXCustom的TPromptQueryButton类可生成下面的HTML代码和JavaScript代码：

```
<script language=javascript type="text/javascript">
function PromptSetField(input, msg) {
    var v = prompt(msg);
    if (v == null || v == "")
        return false;
    input.value = v
    return true;
}
var QueryForm3 = document.forms[ 'QueryForm3' ];
</script>
<input type=button value="Prompt..."
onclick="if (PromptSetField(PromptResult, 'Enter some text\n'))
    QueryForm3.submit();">
```

提示：如果读者打算使用Internet Express，则还需查看InetXCustom额外的演示组件，大家可以在\Demos\Midas\InternetExpress\InetXCustom文件夹中找到它。使用readme.txt文件中的详细指令即可安装这些组件。它们由Borland提供，但是没有任何的支持，允许程序员轻易地添加更多的特性到Internet Express应用程序中。

为了部署这个体系结构，我们不需要在客户端进行任何特殊的设置，因为在任何操作系统上的任何浏览器都支持HTML 4标准。但是，Web服务器必须是一个WIN32服务器（这项技术在Kylix中不可用），此外我们必须在它上面部署DataSnap库。

建立一个范例

为了更好地理解上述内容，作为一种涉及一些技术细节的方法，这里演示一个名为leFirst的简单范例。为了避免配置问题，我们构建一个直接访问数据集的CGI应用程序，亦即本地表格要通过ClientDataSet组件来检索。稍后将向读者显示怎样将一个现存的DataSnap Windows客户程序转换到基于浏览器的接口。为了建立leFirst，我创建了新的CGI应用程序，并给它的数据模块添加了一个连接到本地CDS文件的ClientDataSet组件和连接到数据集的

DataSetProvider组件。下一步是添加XMLBroker组件并连接它到供应器:

```
object ClientDataSet1: TClientDataSet
  FileName = 'C:\Program Files\Common Files\Borland
    Shared\Data\employee.cds'
end
object DataSetProvider1: TDataSetProvider
  DataSet = ClientDataSet1
end
object XMLBroker1: TXMLBroker
  ProviderName = 'DataSetProvider1'
  WebDispatch.MethodType = mtAny
  WebDispatch.PathInfo = 'XMLBroker1'
  ReconcileProducer = PageProducer1
  OnGetResponse = XMLBroker1GetResponse
end
```

ReconcileProducer属性用于在出现更新冲突的情况下显示相应的错误消息。稍后大家会看到, Delphi的一个演示范例中包括了一些定制的代码, 但在这个简单的范例中, 只是将传统的PageProducer与一条常见的HTML错误消息连接起来。在建立了XML调度程序之后, 可以向Web数据模块添加一个InetXPageProducer。该组件有一个标准的HTML框架, 我们已经定制了它来增加一个标题, 而不需要涉及特殊的标记:

```
<HTML><HEAD>
  <title>IeFirst</title>
</HEAD><BODY>
  <h1>Internet Express First Demo (IeFirst.exe)</h1>
  <#INCLUDES><#STYLES><#WARNINGS><#FORMS><#SCRIPT>
</BODY>
```

使用IncludePathURL属性指定目录的JavaScript文件可以自动扩展特殊标记。必须设置该属性, 才能引用这些文件驻留的Web服务器目录。读者可以在Delphi7\Source\WebMidas子目录中发现它们。这五个标记的作用如下:

| 标记 | 效果 |
|-------------|---------------------------------|
| <#INCLUDES> | 生成指令来包括JavaScript库 |
| <#STYLES> | 增加嵌入样式表单的定义 |
| <#WARNINGS> | 用于在设计时显示InetXPageProducer编辑器的错误 |
| <#FORMS> | 生成由Web页面上的组件产生的HTML代码 |
| <#SCRIPT> | 添加一个用于启动客户端脚本的JavaScript块 |

说明: InetXPageProducer组件也处理几个内部标记。<#BODYELEMENTS>对应预定义模板上的所有五个标记。<#COMPONENT Name=WebComponentName>是HTML代码生成部分, 用于声明可视生成的组件。<#DATAPACKET XMLBroker=BrokerName>由实际数据包的XML代替。

为了定制InetXPageProducer的结果HTML, 可以使用其编辑器, 这是一个类似WebSnap服务器端的脚本工具。只需双击InetXPageProducer组件, Delphi便会打开一个窗口, 如图22.10所示(含有该范例的最后设置)。在该编辑器中, 可以从一个查询窗体、一个数据窗

体或一个常见的布局组合开始建立复杂的结构。在范例的数据窗体中，添加了一个DataGrid组件与一个DataNavigator组件，而没有进一步定制它们（如添加子按钮、列与其他对象，以完全代替默认的组件）。

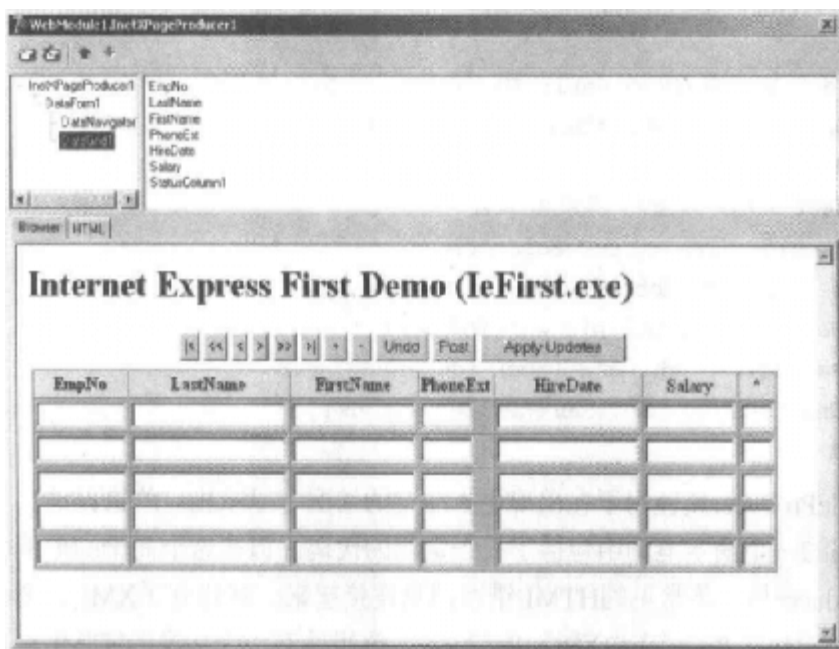


图22.10 InetXPPageProducer编辑器允许我们建立复杂的
可视HTML窗体，与AdapterPageProducer类似

范例中用于InetXPPageProducer和它内部组件的DFM代码如下所示，从中读者能看到核心设置加上有限的图形定制：

```
object InetXPPageProducer1: TInetXPPageProducer
  IncludePathURL = '/jssource/'
  HTMLDoc.Strings = (...)
  object DataForm1: TDataForm
    object DataNavigator1: TDataNavigator
      XMLComponent = DataGrid1
      Custom = 'align="center"'
    end
    object DataGrid1: TDataGrid
      XMLBroker = XMLBroker1
      DisplayRows = 5
      TableAttributes.BgColor = 'Silver'
      TableAttributes.CellSpacing = 0
      TableAttributes.CellPadding = 2
      HeadingAttributes.BgColor = 'Aqua'
      object EmpNo: TTextColumn...
      object LastName: TTextColumn...
      object FirstName: TTextColumn...
      object PhoneExt: TTextColumn...
```



```

        object HireDate: TTextColumn...
        object Salary: TTextColumn...
        object StatusColumn1: TStatusColumn...
    end
end
end

```

这些组件的值位于它们生成的HTML（以及JavaScript）代码中，我们可以通过选择InetXPageProducer编辑器的HTML选项卡来进行预览。下面是HTML代码中针对按钮、数据网格标头以及一个网格单元的部分定义：

```

// buttons
<table align="center">
  <tr><td colspan="2">
    <input type="button" value="|<"
      onclick='if (xml_ready) DataGrid1_Disp.first();'>
    <input type="button" value="<<"
      onclick='if (xml_ready) DataGrid1_Disp.pgup();'>
  ...
// data grid heading
<table cellpadding="2" cellspacing="0" border="1" bgcolor="silver">
  <tr bgcolor="aqua">
    <th>EmpNo</th>
    <th>LastName</th>
  ...
</tr>
<tr>
  // a data cell
  <td><div>
    <input type="text" name="EmpNo" size="10"
      onfocus='if(xml_ready)DataGrid1_Disp.xfocus(this);'
      onkeydown='if(xml_ready)DataGrid1_Disp.keys(this);'>
  </div></td>...

```

当HTML生成器被建立好后，我们可以回到Web数据模块，向它添加一个操作，然后把这个操作通过Producer属性连接到InetXPageProducer。这就足以使程序通过一个浏览器进行工作，正如我们在图22.11中看到的。

如果通过浏览器查看接收到的HTML文件，会发现上面提到的表格定义，一些JavaScript代码，以及XML格式的一些数据库数据（在元数据之后）。该数据由XML调度程序组合，并发送给制作器组件，装入HTML文件中。注意，发送给客户程序的记录的数量取决于XMLBroker，而不是网格中的行数。事实上，一旦将XML数据发送给浏览器，就可以使用导航器组件的按钮移动数据，而不需要进一步访问服务器以索取更多数据。这与WebSnap的页行为十分不同，并不是简单的这个比那个更好的问题，这都取决于建立的特殊应用程序。

同时，系统中的JavaScript类允许用户输入新数据，但要遵循与动态HTML事件相连的JavaScript代码的强制规则。注意，在默认情况下，网格有一个额外的星号列，说明被改动

的记录。更新数据被收集在浏览器中的一个XML数据包里，并在用户单击Apply Updates按钮时，发回服务器。在此，浏览器会激活由XMLBroker的WebDispath.PathInfo属性指定的动作。不需要从Web数据模块中调出该动作，因为这个操作是自动的（尽管可以通过将WebDispath.Enable设置为False来解除它）。

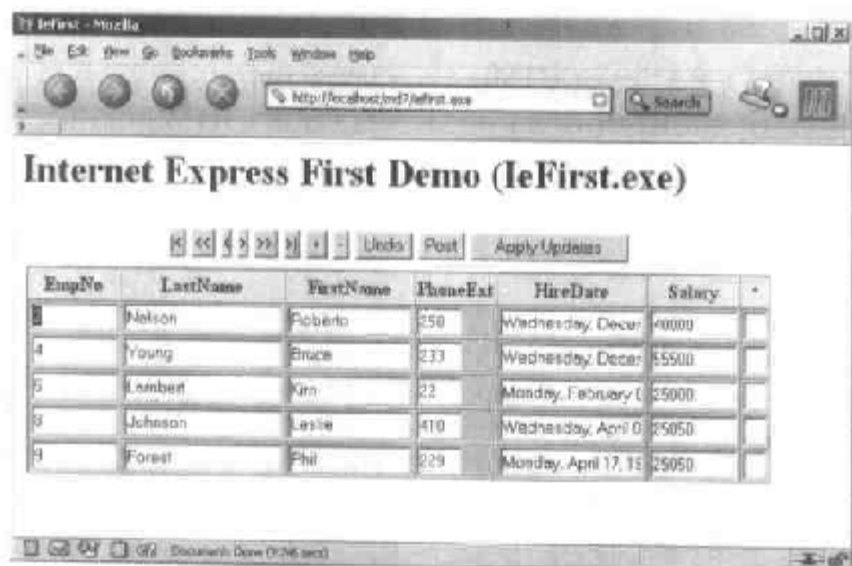


图22.11 IeFirst应用程序向浏览器发送了一些HTML组件、一个XML文档以及JavaScript代码，用以在可视组件中显示数据

XMLBroker可将所做修改应用于服务器中，亦即返回与ReconcileProvider属性相连的供应器的内容（或如果没有定义，则引起一个异常）。当一切运行正常时，XMLBroker会将控制改向包含数据的主页。然而，在使用该技术时，出现了一些问题，因此，IeFirst范例使用下列代码处理OnGetReponse事件，表明这是一个更新的视图：

```
procedure TWebModule1.XMLBroker1GetResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<h1>Updated</h1><p>' + InetXPPageProducer1.Content;
  Handled := True;
end;
```

使用XSLT

另一项用于从XML文档开始生成HTML的技术是使用可扩展样式表语言（XSL），或者更准确地说，是XSL转换子集（XSLT）。XSLT的目的是将一个XML文档翻译成另外一个文档，通常也是一个XML文档。最常用的技术是将一个XML文档转换为一个XHTML文档，并将其从一个Web服务器发送到一个Web浏览器中。另一个有趣的相关技术是XSL-FO，即XSL Formatting Objects。它可以用来将一个XML文档转化为一个PDF或者其他类型的文档。

XSLT文档是构造良好的XML文档。XSLT文件的结构需要根节点，如下所示：

```
<xsl:stylesheet version="1.0" xmlns:xsl="...">
```

XSLT文件的内容是基于一个或者多个模板的（或者规则或函数），它将被引擎进行处理。这个节点是`xsl:template`，通常带有一个`match`属性。在这个最简单的例子里，一个模板对带有给定名称的节点进行操作。我们可以通过将它传递给一个或者多个带有XPath表达式的节点来激活模板：

```
xsl:apply-templates select="node_name"
```

该操作从一个处理根节点的模板开始，它可以是XSLT文件的惟一模板。在这些模板中，我们可以找到其他的命令，例如从一个XML文档中提取值的语句（`xsl:value-of select`）、循环语句（`xsl:for-each`）、条件表达式（`xsl:if`, `xsl:choose`）、排序请求（`xsl:sort`）以及编号请求（`xsl:number`），这里只提到了一些普通的XSLT命令。

使用XPath

XSLT使用了其他的XML技术，特别是用XPath来标识文档部分。XPath定义了一些规则来定位文档中的一个或者多个节点。这些规则基于XML树节点的一个路径结构，因此`/books/book`可标识任何在books文档根下的book节点。XPath使用了特殊的符号来标识这些节点：

- 星号（*），代表任意节点，例如`book/*`说明在book节点下的子节点。
- 点（.），代表当前节点。
- 管道符号（|），说明可选项，如`book | ebook`。
- 双斜线（//），表示任何路径，因此 `//title`指明所有标题节点而不论它们的父节点是什么，`books//author`说明在书节点下的作者节点，而不管它们之间的节点。
- 符号（@），说明属性而不是节点，如`author/@lastname`。
- 方括号（[]），用于选择有给定值的节点或者属性。例如，给定First Name的作者：
`author[@name="marco"]`。

还有很多其他的情况，但是这个简短的关于XPath规则的介绍会给读者以启示，有助于大家理解下面的例子。XSLT文档就是XML文档，它工作在源XML文档结构中，并且生成另一个XML文档的输出，例如一个我们可以在Web浏览器中观看到的XHTML文档。

说明：常用的XSLT处理器包括MS-XML、来自Apache XML项目（xml.apache.org）的Xalan以及来自James Clarke的基于Java的Xt。在Delphi中，我们还可以使用XSLT引擎，包括在TurboPower的XML Partner Pro中（www.turbopower.com）。

实际的XSTL

我们下面将讨论两个范例。首先研究XSL自身，然后集中介绍Delphi应用程序的操作。

为了初始测试，可直接将XSL文件连接到XML文件上。当在Internet Explorer上装载XML时，它将显示转换的结果，通常是XHTML。应在XML文档的头文件中用下列命令指示连接：

```
<?xml-stylesheet type="text/xsl" href="sampleembedded.xml"?>
```

我已在XslEmbed文件夹的sampleembeded.xml文件中实现了这一点。相关的XSL会嵌入各种XSL代码片段，但这里不能详细介绍。例如，它会获取作者的整个列表，或用该代码过滤特殊组：

```
<h2>All Authors</h2>
<ul>
  <xsl:for-each select="books//author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
<h3>E-Authors</h3>
<ul>
  <xsl:for-each select="books/ebook/author">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ul>
```

一些更复杂的代码用于当特殊值被显示在子节点或属性中时抽取代码，而不管高层节点。最后的XSL片段也含有if语句，并在结果节点产生属性，下面的代码会在HTML中建立一个href超链接：

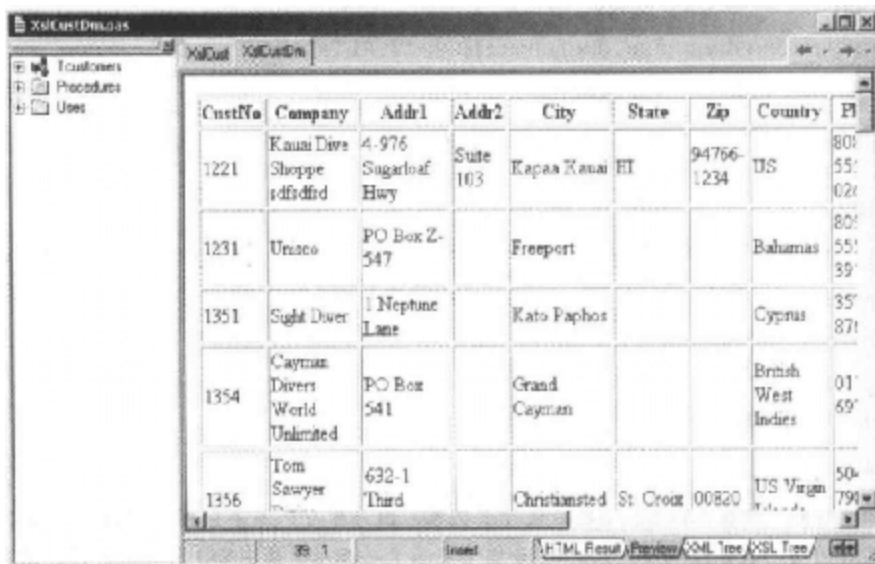
```
<h3>Marco's works (books + ebooks)</h3>
<ul>
  <xsl:for-each select="books/*[author = 'Cantu']">
    <li> <xsl:value-of select="title"/>
      <xsl:if test="url">
        (<a><xsl:attribute name="href"><xsl:value-of select="url"/>
          </xsl:attribute>Jump to document</a>)
      </xsl:if>
    </li>
  </xsl:for-each>
</ul>
```

使用WebSnap的XSLT

在程序代码内，可执行DOM节点的TransformNode对象方法，亦即向该节点传递另一个含有XSL文档的DOM。除了使用该低级方法外，可以让WebSnap帮助我们创建基于XSL的范例。事实上，可以创建新WebSnap应用程序（这时将建立名为XslCust的CGI程序）并选择用于主页的XSLPageProducer组件，让Delphi帮助我们实现应用程序代码。实际上，Delphi也包含处理ClientDataSet数据包的XSL文件框架，并会向编辑器添加新视图。XSL文本将代替HTML文件，XMLTree页面显示数据，XSLTree页面在Internet Explorer ActiveX内显示XSL，HTML结果页显示转换产生的代码，而预览页显示用户将在浏览器上看到什么。

提示：在Delphi 7中，编辑器为XSLT提供了完整的代码完成功能，它使得在编辑器中编辑这种代码与在一些专用的编辑器中编辑代码一样强大。

为了使范例实际工作起来，还必须通过它的XMLData属性为XSLPageProducer组件提供一些数据。该属性能被链接到XMLDocument，也能直接连到XMLBroker组件上，如我在此例中所做的。中介器可以从与本地表格相连的供应器那里获取数据，而这个本地表格与典型DBDEMOS的Customers.cds组件表格相连。结果如下，它含有随后由Delphi生成的XSL，我们能（甚至在设计时）获得如图22.12所示的输出。



The screenshot shows a web browser window titled 'XslCustDemo'. The address bar shows 'XslCustDemo'. The left sidebar has a tree view with 'Customers', 'Procedures', and 'Uses'. The main content area displays a table with the following data:

| CustNo | Company | Addr1 | Addr2 | City | State | Zip | Country | Fl |
|--------|-------------------------------|---------------------|-----------|---------------|-----------|------------|---------------------|----------|
| 1221 | Kaunai Dive Shoppe | 4-976 Sugarloaf Hwy | Suite 103 | Kapaa | Kaunai HI | 94766-1234 | US | 80:55:02 |
| 1231 | Unesco | PO Box Z-547 | | Freeport | | | Bahamas | 80:55:39 |
| 1351 | Sight Diver | 1 Neptune Lane | | Kato Paphos | | | Cyprus | 35:87 |
| 1354 | Cayman Divers World Unlimited | PO Box 541 | | Grand Cayman | | | British West Indies | 01:69 |
| 1356 | Tom Sawyer | 632-1 Third | | Christiansted | St. Croix | 00820 | US Virgin Islands | 50:79:4 |

The bottom of the browser window shows a status bar with '39 1' and a toolbar with buttons for 'HTML Result', 'Preview', 'XML Tree', 'XSL Tree', and 'Help'.

图22.12 由XslCust范例内的XSLPageProducer组件所产生的XSLT转换结果

```

<?xml version="1.0"?>
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="DATAPACKET">
    <table border="1">
      <xsl:apply-templates select="METADATA/FIELDS"/>
      <xsl:apply-templates select="ROWDATA/ROW"/>
    </table>
  </xsl:template>

  <xsl:template match="FIELDS">
    <tr>
      <xsl:apply-templates/>
    </tr>
  </xsl:template>

  <xsl:template match="FIELD">
    <th>
      <xsl:value-of select="@attrname"/>
    </th>
  </xsl:template>

```

```

<xsl:template match='ROWDATA/ROW'>
<xsl:variable name="fieldDefs" select="//METADATA/FIELDS"/>
<xsl:variable name="currentRow" select= current()"/>
<tr>
  <xsl:for-each select="$fieldDefs/FIELD">
    <td>
      <xsl:value-of select="$currentRow/@*[name()='current()']/@attrname'"/><br />
    </td>
  </xsl:for-each>
</tr>
</xsl:template>
</xsl:stylesheet>

```

说明：标准的XSL模板在Delphi 6之后进行了扩展，因为原有版本不能解决空值字段被XML数据包省略的问题。我在2002年的Borland会议上展示了一些对于原有XSL代码的扩展，我的一些建议现在已经被集成到模板之中。

该代码生成的HTML表格由字段元数据的扩展和行数据组成。字段用于生成表格标题，这可以通过在单行中为每个输入使用<th>单元来实现。行数据用每个属性的值（select="@,"）来填充表格的其他行，但这还是不够的，因为一个属性可能会丢失。出于这个原因，要把这个字段列表以及当前行存储在两个变量值中；然后，针对每一个字段，由XSL代码提取一个行项目的值，这个项目具有一个与存储在其attrname属性（@attrname）中的当前字段名称相对应的属性名称（@*[name()='...']。这个代码很复杂，但是它对于要同时检查一个XML文档的不同部分来说，是很简洁的。

用DOM指导XSL转换

使用XSLPageProducer非常容易，但基于同一数据生成多个页以便使用WebSnap处理不同的XSL样式不是好方法。笔者为此建立了一个名为CdsXslt的普通CGI应用程序，它能传递ClientDataSet数据包到不同的HTML类型中，这依赖于作为参数传递的XSL文件名称。其优点是在不重新编译程序的前提下，不但可以修改它，还可以向系统添加新的XSL文件。

为了获得XSL转换，程序要装载XML和XSL文件到两个XMLDocument组件中，分别是xmlDom和XslDom。然后激活XML文档的transformNode对象方法，把XSL文档作为参数并在第二个XMLDocument组件中填充它，这个组件名为HtmlDom：

```

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  xslfile, xslfolder: string;
  attr: IDOMAttr;
begin
  // open the client dataset and load its XML in a DOM
  ClientDataSet1.Open;
  XmlDom.Xml.Text := ClientDataSet1.XMLData;
  XmlDom.Active := True;

```

```

// load the requested xsl file
xslfile := Request.QueryFields.Values ['style'];
if xslfile = '' then
    xslfile := 'customer.xsl';
xslfolder := ExtractFilePath (ParamStr (0)) + 'xsl\';
if FileExists (xslfolder + xslfile) then
    xslDom.LoadFromFile (xslfolder + xslfile)
else
    raise Exception.Create('Missing file: ' + xslfolder + xslfile);
XSLDom.Active := True;
if xslfile = 'single.xsl' then
begin
    attr := xslDom.DOMDocument.createAttribute('select');
    attr.value := '//ROW[@CustNo="' + Request.QueryFields.Values ['id'] + '"]';
    xslDom.DOMDocument.getElementsByTagName ('xsl:apply-templates').
        item[0].attributes.setNamedItem(attr);
end;
// do the transformation
HTMLDom.Active := True;
xmlDom.DocumentElement.transformNode (xslDom.DocumentElement, HTMLDom);
Response.Content := HTMLDom.XML.Text;
end;

```

该代码会使用DOM来修改用于显示单个记录的XSL文档，为了选择由id查询字段指定的记录，可添加XPath语句。该id通过含有记录列表的XSL填充到超链接中，不过，这里我宁愿跳过显示文件列表这一步。读者可以仔细研究它们，它们位于该示例文件夹的XSL子文件夹中。

警告：为了运行该程序，应该在存放该脚本的文件夹下面，在XSL文件夹中部署XSL文件。我们可以从一个脚本文件夹的XSL子文件夹中找到本章的演示文件。为了部署这些文件到一个不同的位置，应更改上面提取XSL文件夹名的代码，它在第一个公共行参数中（就像Forms单元中定义的全局Application对象在一个CGI应用程序中不可用一样）。

处理大型的XML文档

正如我们看到的，XML有很多不同的技术来完成相同的任务。在很多情况下，我们可以选择任何以减少代码和增加可维护性为目的的解决方案。但是当我们需要处理大量XML文档或者是大型XML文档的时候，必须考虑有效性的问题。

讨论理论本身并没有太大的用处，因此我们建立了一个范例用来测试各种解决方案。这个范例称为LargeXml，它涉及一个特定的领域：将数据从数据库移动到一个XML文件，然后再移动回去。这个范例可以打开一个数据集（使用dbExpress），然后多次复制数据到一个内存中的ClientDataSet。这个内存中的ClientDataSet的结构模仿了数据访问组件的结构：

```

SimpleDataSet1.Open;
ClientDataSet1.FieldDefs := SimpleDataSet1.FieldDefs;
ClientDataSet1.CreateDataSet;

```

在使用一个单选按钮组来决定我们需要处理的数据量以后（在比较慢的计算机上，某些选项可能会运行数分钟），数据使用下面的代码进行复制：

```
while ClientDataSet1.RecordCount < nCount do
begin
    SimpleDataSet1.RecNo := Random (SimpleDataSet1.RecordCount) + 1;
    ClientDataSet1.Insert;
    ClientDataSet1.Fields [0].AsInteger := Random (10000);
    for i := 1 to SimpleDataSet1.FieldCount - 1 do
        ClientDataSet1.Fields [i].AsString := SimpleDataSet1.Fields [i].AsString;
    ClientDataSet1.Post;
end;
```

从ClientDataSet到XML文档

现在，程序具有一个大型的数据集，位于内存之中。它提供了三种不同的方式来将这个数据集保存在一个文件内。第一种办法是直接保存ClientDataSet的XMLData到一个文件中，以获得一个基于属性的文档。这可能不是我们期望的格式，因此第二个解决方案是应用XmlMapper转换，这是通过XMLTransformClient组件实现的。第三个解决方案涉及到直接处理数据集并且将每一个记录写到文件中：

```
procedure TForm1.btnSaveCustomClick(Sender: TObject);
var
    str: TFileStream;
    s: string;
    i: Integer;
begin
    str := TFileStream.Create ('data3.xml', fmCreate);
    try
        ClientDataSet1.First;
        s := '<?xml version="1.0" standalone="yes" ?><employee>';
        str.Write(s[1], Length (s));
        while not ClientDataSet1.EOF do
            begin
                s := '';
                for i := 0 to ClientDataSet1.FieldCount - 1 do
                    s := s + MakeXmlstr (ClientDataSet1.Fields[i].FieldName,
                        ClientDataSet1.Fields[i].AsString);
                s := MakeXmlStr ('employeeData', s);
                str.Write(s[1], length (s));
                ClientDataSet1.Next;
            end;
            s := '</employee>';
            str.Write(s[1], length (s));
        finally
            str.Free;
```



```
end;  
end;
```

这段代码使用了一个简单但是有效的支持函数来创建XML节点:

```
function MakeXmlstr (node, value: string): string;  
begin  
    Result := '<' + node + '>' + value + '</' + node + '>';  
end;
```

如果我们运行这个程序,则可以看到每个操作用掉的时间,如图22.13所示。保存ClientDataSet数据是一个最快的方法,但是可能不能得到期望的结果。定义流的方法要稍稍慢一点,但是使用这些代码将不会要求大家先将数据移动到一个ClientDataSet中,因为可以直接应用它,甚至应用于一个单向的dbExpress数据集。我们在遇到一个大型数据集的时候,不应该考虑使用基于XmlMapper的代码,因为它可能会慢数百倍,甚至对一个小的数据集也是如此(我没有尝试一个大的数据集,因为这样将会浪费太多的时间)。例如,针对一个小的数据集使用定制的流大概需要50毫秒的时间,而如果使用映射,在结果相似的情况下,用时超过10秒钟。

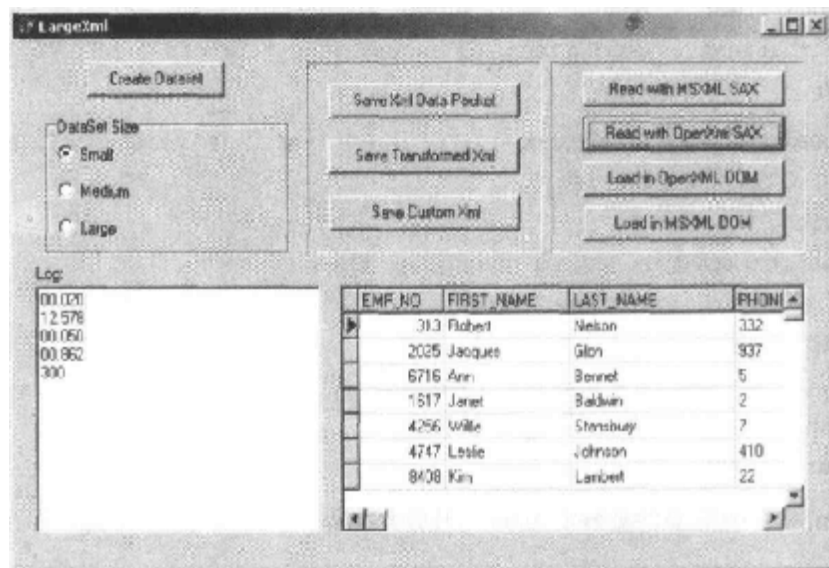


图22.13 LargeXml范例的操作

从XML文档到ClientDataSet

当我们拥有一个大的XML文档的时候,无论它是从一个程序中获得的或者是从一个外部源获得的,我们都需要处理它。正如大家看到的,XmlMapper的支持太慢了,因此我们使用其他三种办法:XSL转换、SAX或者DOM。XSL转换是足够快的,但是在这个例子中,我们使用SAX来打开这个文档,它是最快的方法,而且不要求太多的代码。这个程序可以将一个文档装入一个DOM中,但是我们没有编写定位DOM以及将数据存入一个ClientDataSet的代码。

在这两种情况下,我们测试了OpenXml引擎与MSXML DOM。这就允许我们对两种SAX解决方案进行比较,因为它们的代码还存在一些不同之处。我们对结果进行一个小结:

使用MSXML SAX比使用OpenXml SAX要稍微快一点, 差异在百分之二十左右。而在DOM中装载要比在MSXML中有更大的优势。

MSXML SAX代码与在SaxDemo1范例中讨论过的结构相同, 因此这里我们仅介绍我们使用到的处理程序代码。正如大家看到的, 在EmployeeData的开始部分, 我们插入了一个新的记录, 它在同一个节点关闭的时候被提交。低层次节点被作为当前记录的字段来添加。下面是其程序代码:

```

procedure TMyDataSaxHandler.startElement(var strNamespaceURI, strLocalName,
    strQName: WideString; const oAttributes: IVBSAXAttributes);
begin
    inherited;
    if strLocalName = 'employeeData' then
        Form1.clientdataset2.Insert;
        strCurrent := ' ';
    end;

procedure TMyDataSaxHandler.characters(var strChars: WideString);
begin
    inherited;
    strCurrent := strCurrent + RemoveWhites(strChars);
end;

procedure TMyDataSaxHandler.endElement(var strNamespaceURI, strLocalName,
    strQName: WideString);
begin
    if strLocalName = 'employeeData' then
        Form1.clientdataset2.Post;
    if stack.Count > 2 then
        Form1.ClientDataSet2.FieldByName (strLocalName).AsString := strCurrent;
    inherited;
end;

```

OpenXml版本中的事件处理程序的代码与此相似, 所有的变化就是方法的接口以及参数的名称:

```

type
    TDataSaxHandler = class (TXmlStandardHandler)
    protected
        stack: TStringList;
        strCurrent: string;
    public
        constructor Create(aowner: TComponent); override;
        function endElement(const sender: TXmlCustomProcessorAgent;
            const locator: TdomStandardLocator;
            namespaceURI, tagName: WideString): TXmlParserError; override;
        function PCDATA(const sender: TXmlCustomProcessorAgent;
            const locator: TdomStandardLocator; data: WideString);

```

```

    TXmlParserError; override;
    function startElement(const sender: TXmlCustomProcessorAgent;
        const locator: TdomStandardLocator; namespaceURI, tagName: wideString;
        attributes: TdomNameValueList): TXmlParserError; override;
    destructor Destroy; override;
end;

```

更困难的是调用SAX引擎，就像下面的代码显示的那样（我们在这里已删除了建立数据集、时间和日志的相关代码）：

```

procedure TForm1.btnReadSaxOpenClick(Sender: TObject);
var
    agent: TXmlStandardProcessorAgent;
    reader: TXmlStandardDocReader;
    filename: string;
begin
    Log := memoLog.Lines;
    filename := ExtractFilePath (Application.Exename) + 'data3.xml';
    agent := TXmlStandardProcessorAgent.Create(nil);
    reader := TXmlStandardDocReader.Create (nil);
    try
        reader.NextHandler := TDataSaxHandler.Create (nil); // our custom class
        agent.reader := reader;
        agent.processFile(filename, filename);
    finally
        agent.free;
        reader.free;
    end;
end;

```

小结

在本章中，我们讨论了XML与相关的技术，包括DOM、SAX、XSLT、XML模式、XPath等等。读者已经看到Delphi是如何使用XML访问来简化DOM编程的，它使用了接口以及XML转换。我们还讨论了使用XSL进行Web编程，介绍了XSLT的WebSnap支持以及Internet Express体系结构。

第23章将继续讨论XML近年来的一个最有趣的技术：Web服务。我们将讨论SOAP和WSDL，并介绍UDDI和其他相关的技术。如果读者对这些课题感兴趣，可以阅读一些专门介绍XML、XML模式和XSLT的书籍。

第23章 Web服务与SOAP

在Delphi的所有新特征中，有一个很突出：支持将Web服务建立到产品中。本书最后讨论的事实与这点无关，只与文本的逻辑流程有关，而且也不是学习Delphi编程的起点。

Web服务的课题非常广泛而且包括很多的技术和商业标准。像往常一样，我们将集中介绍Delphi的实现以及与Web服务相关的技术，而不是讨论宏观形式与商业应用。

相对于Delphi 6，Delphi 7中添加了众多的用于Web服务的改进成份，包括对于附件的支持，定制标头以及其他。我们将看到如何创建一个Web服务客户端与Web服务服务器，以及如何使用DataSnap技术通过SOAP来移动数据库数据。

本章主要包括以下内容：

- Web服务
- SOAP和WSDL
- SOAP之上的DataSnap
- 处理附件
- UDDI

Web服务

但什么是网络服务？它是快速涌现的技术，有潜力来改变Internet处理商务工作的方式。依靠浏览网页来输入订单这种做法对个人（B2C，business-to-consumer），而不是对企业（B2B，business-to-business）来说，是好的。例如，为购买图书而登录书商网站，并单击自己的请求。但如果开一个书店并想一天处理数以百计的订单，这便不是一种好方法。特别是在拥有能帮助我们追踪销售和决定再订购的程序时。获取该程序的输出并重输入它到另一个应用程序十分可笑。

网络服务就是解决这种问题的工具：用于追踪销售的程序能自动创建请求并发送它到网络服务上，该服务能立即返回关于订单的信息。下一步是询问追踪出货的数量。这时程序能使用另一个网络服务追踪出货直到它到达目的地，因此可以告诉用户需要等待多长时间。当出货到达时，程序能通过SMS发出提示或呼叫订单尚未完成的人，讨论用银行网络服务付费等问题……读者可以充分发挥自己的想像力。网络服务是计算机和网络及E-mail相结合的结果，用于为用户提供可交互的界面。

SOAP和WSDL

网络服务由Simple Object Access Protocol (SOAP) 提供。SOAP是建立在标准HTTP上的，因此网络服务器能通过防火墙处理SOAP请求和相关的数据包。SOAP定义了一种基于XML的符号，用来请求执行服务器上对象的对象方法，并且向它传递参数，另一个符号定义了响应的格式。

说明: SOAP最初是由DevelopMentor (COM专家Don Box建立的一个培训公司)和Microsoft开发的,用以克服在网络服务器内使用DCOM的缺点。提交给W3C进行标准化后,它被许多公司所承认,特别是受到了IBM的大力推动。但是说它能够成为使微软、IBM、Sun、Oracle及其他应用程序相互结合的标准,或者是这些供应商试图推出该标准的私有版本,还为时尚早。然而,在任何情况下,SOAP都是Microsoft的.NET结构的基石,也是Sun和Oracle当前所推荐的平台。

SOAP将代替COM革新,至少在不同类型的计算机间是这样的。类似地,遵循网络服务描述语言格式的SOAP服务定义将代替IDL和COM、COM+使用的类型库。WSDL文档是XML文档的另一个类型,用于提供SOAP请求的元数据定义。当获得该格式的文件后,就能创建程序调用它。

Delphi特别提供了WSDL和接口间的双向映射。这意味着可以获取WSDL文件并为其生成接口。这时,可以通过这些接口创建嵌入SOAP请求的客户程序,并使用特殊的Delphi组件,该组件允许将本地接口转换为SOAP调用(因为有可能需要手工产生用于SOAP请求的XML请求)。

还有另一个方法,我们可以定义一个接口(或使用一个现有的接口),并且让一个Delphi组件为它生成一个WSDL描述。另外一个组件可以提供一个SOAP到Pascal的映射,因此通过嵌入这个组件以及一个实现接口的对象到服务器端应用程序中,我们就可以使我们的Web服务启动并且运行。

BabelFish翻译

作为使用网络服务的第一个范例,这里建立了一个简单的客户程序用于由AltaVista提供的BabelFish翻译服务。读者会在XMethods网站(www.xmethods.com)上发现它和其他服务来进行试验。

从XMethods(也可在本章的源代码文件中获得)中下载了该服务的WSDL描述之后,在New Items对话框的网络服务页上激活Delphi的Web Services Importer,并且选择文件。这个向导将使我们预览这个服务的结构,如图23.1所示。Delphi将为网络服务生成Delphi接口,如下所示:

```
unit BabelFishService;

interface

uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;

type
  BabelFishPortType = interface(IInvokable)
    ['{D2DB6712-EBE0-1DA6-8DEC-8A445595AE0C}']
    function BabelFish(const translationmode: WideString;
      const sourcedata: WideString): WideString; stdcall;
  end;

function GetBabelFishPortType(UseWSDL: Boolean=System.False;
  Addr: string=''; HTTPRIO: THTTPRIO = nil): BabelFishPortType;

implementation

// omitted
```

initialization

```

InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish', '');
InvRegistry.RegisterDefaultSOAPAction(TypeInfo(BabelFishPortType),
    'urn:xmethodsBabelFish#BabelFish');

```

end.

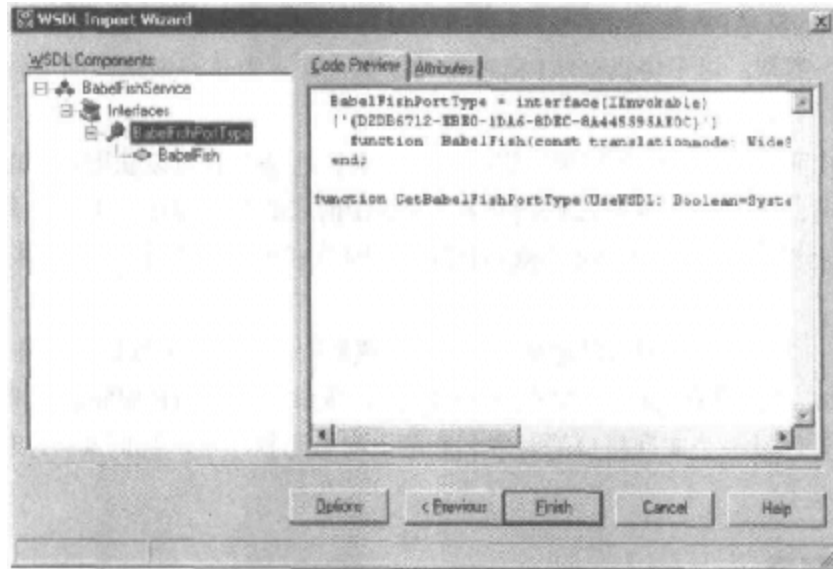


图23.1 工作中的WSDL Import Wizard

注意第一个接口继承自IInvokable接口。该接口不会向Delphi的基接口IInterface添加关于对象方法的事物，但将和标记一起被编译，该标记用于启用RTTI生成功能，{**\$M+**}，与TPersistent类相似。在initialization部分中，读者能看到该接口被注册在全局调用注册表（或InvRegistry）中，负责传递接口类型的类型信息引用。

说明：用于接口的RTTI信息实际上是SOAP最重要的技术优势。这并不是说SOAP到Pascal的映射不重要，因为它对于简化处理操作很重要，但是用于接口的RTTI将使整个结构强大而且健壮。

这个单元的第三个元素由WSDL Import Wizard产生，是一个全局函数，使用该服务在Delphi 7中的名称进行命名。这个函数有助于简化调用Web服务的代码。GetBabelFishPortType函数返回一个正确类型的接口，我们可以使用它直接进行调用。例如，下面的代码将一个句子从英文翻译到意大利语（由第一个参数en_it指明），并且在屏幕上进行显示：

```
ShowMessage (GetBabelFishPortType.BabelFish('en_it', 'Hello, world!'));
```

如果我们查看GetBabelFishPortType函数的代码，我们将会看到它创建了一个内部的THTTPIO类的组件来处理调用。我们还可以将这个组件手动地放置到客户端窗体中（就像我们在范例程序中做的那样），从而更好地控制它的不同设置，并且处理它的事件。

这个组件能够使用两种基本的方法进行配置：一是引用一个WSDL文件或者一个URL，导入它并且从中提取SOAP调用的URL；二是提供一个直接的URL调用。这个范例有两个组件，可提供不同的方法（具有相同的结果）：

```

object HTTPRIO1: THTTPRIO
    WSDLLocation = 'BabelFishService.xml'
    Service = 'BabelFish'
    Port = 'BabelFishPort'
end
object HTTPRIO2: THTTPRIO
    URL = 'http://services.xmethods.net:80/perl/soaplite.cgi'
end

```

这时，需要做的事情很少。我们拥有调用服务所需要的信息，并且知道接口中的可用方法所需的各种类型的参数。通过从HTTTPRIO组件直接抽取我们想调用的接口，利用像HTTPRIO1 as BabelFishPortType这样的表达式，即可合并两个元素。乍一看上去觉得怪怪的，但它其实很简单。下面是该范例完成的网络服务调用：

```

EditTarget.Text := (HTTPRIO1 as BabelFishPortType).
    BabelFish(ComboBoxType.Text, EditSource.Text);

```

该程序的输出结果如图23.2所示，它允许用户学习外国语言（尽管它的功能很少）。这里没有复制与股票、货币、天气预报和许多其他可用服务，它们的实现与此非常相似。

警告：虽然Web服务接口提供给我们不同类型的参数，在很多的情况下，我们需要借助一些具体的关于服务的文档，以了解这些参数数值的实际意义以及服务本身对这些数值的理解。BabelFish Web服务是一个用于此目的的范例。

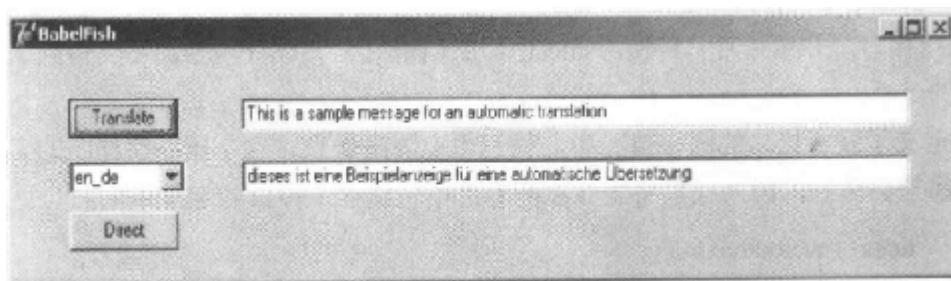


图23.2 一个英译德的范例，是一个来自AltaVista的BabelFishvia网络服务

建立网络服务

如果说在Delphi上调用网络服务很直接，那么也可以说实际服务的开发也很直接。如果进入New Items对话框的网络服务页，就能看到SOAP服务器应用程序选项。选中它，Delphi将展示一个十分类似于WebBroker应用程序选项的列表。事实上，网络服务通常受网络服务器支持，使用一种可行的网络服务器扩展技术（CGI、ISAPI、Apache模块等）或者使用Web App Debugger进行初始化测试。

完成该步骤后，Delphi将添加三个组件到结果网络模块，该模块正好是普通网络模块，没有特殊的添加项：

- HTTPSoapDispatcher组件起接收网络请求的作用，像其他HTTP分配器（dispatcher）那样。

- HTTPSoapPascalInvoker组件做HTTPRIO组件的相反操作，因为它能将SOAP请求翻译为Pascal接口的调用（代替将接口对象方法调用转换为SOAP请求）。
- WSDLHTMLPublish组件能用于从它支持的接口抽取服务的WSDL定义，与Web Services Importer Wizard的作用相反。从技术上讲，这是另一个分配器。

一个货币转换Web服务

一旦该框架就绪，我们也就可以通过在现存网络模块上添加三个组件来做点事——下面可以开始编写服务。例如采用第3章“运行时库”中的欧元转换范例，并把它转换成名为ConvertService的网络服务。首先，向程序中添加一个定义服务接口的单元，如下所示：

```
type
  IConvert = interface(IInvokable)
    ['{FF1EAA45-0B94-4630-9A18-E768A91A78E2}']
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

直接以代码定义接口而不必使用像Type Library Editor这样的工具有很大优点，因为我们可以轻松地为一类创建接口而不必为此去学习如何使用专用工具。请注意，我已将GUID赋给接口，并像往常一样使用stdcall调用惯例，因为SOAP转换器不支持默认的register调用惯例。

在定义了服务接口的同一单元中，我们也应该注册它。这个操作在程序的客户端和服务器端都是必需的，因为我们将能够在两者之中包括这个接口的定义单元：

```
uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(IConvert));
```

现在我们拥有了可以展现给公众的接口，接下来必须为其提供实现，这次仍然是使用标准的Delphi代码，并且依靠预定义的TInvokableClass类的帮助：

```
type
  TConvert = class (TInvokableClass, IConvert)
  protected
    function ConvertCurrency (Source, Dest: string; Amount: Double): Double;
      stdcall;
    function ToEuro (Source: string; Amount: Double): Double; stdcall;
    function FromEuro (Dest: string; Amount: Double): Double; stdcall;
    function TypesList: string; stdcall;
  end;
```

这些函数调用了欧元转换系统的代码（来自第3章），我们在这里不准备对其实现进行叙述，因为它对于开发服务并没有什么帮助。但是，有一点非常值得注意，那就是这个实现

单元在其初始化部分还有一个注册调用：

```
InvRegistry.RegisterInvokableClass (TConvert);
```

发布WSDL

通过注册这个接口，我们就可以使该程序产生一个WSDL描述。这一Web服务应用程序，（从Delphi 6.02更新以来）可以显示一个首页，用来描述它的接口以及每一个接口的详细信息，并且返回WSDL文件。通过使用浏览器连接到该Web服务，我们能够看到与图23.3所示类似的结果。



图23.3 Delphi组件提供的ConvertService Web服务的描述

说明：虽然其他的Web服务体系结构会自动提供给我们一个从浏览器执行Web服务的办法，但是这种技术基本上没有什么用处，因为在有不同的应用程序进行互操作的体系结构中使用Web服务是行得通的，如果我们需要的只是在浏览器中显示数据，那么应该建立一个网站。

这个自动描述特性在Delphi 6提供的Web服务中并没有提供（它只提供了低层的WSDL列表），但是它很容易被添加或者定制。如果我们查看Delphi 7中的SOAP Web模块，就会注意到一个OnAction事件处理程序默认的操作，它调用了下面的默认操作：

```
WSDLHTMLPublish1.ServiceInfo(Sender, Request, Response, Handled);
```

这就是我们为了将这个特性加入到一个现有的Delphi Web服务中所需要做的。为了以手工方式提供类似的功能，我们必须将启用注册（InvRegistry全局对象）集中起来；为此，要使用调用GetInterfaceExternalName和GetMethExternalName。

重要的是，Web服务应用程序应该具有通过展示WSDL向任何程序员或者编程工具证明自己的能力。

创建定制客户端

现在再来看看调用该服务的客户应用程序。这时不需要从WSDL文件开始，因为已经有Delphi接口了。这时该窗体甚至没有HTTPRIO组件，该组件是用代码创建的：

```
private
    Invoker: THTTPrIo;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Invoker := THTTPrIo.Create(nil);
    Invoker.URL := 'http://localhost/scripts/ConvertService.exe/soap/iconvert';
    ConvIntf := Invoker as IConvert;
end;
```

使用WSDL文件的另一种方案是，将SOAP调用器组件与一个URL关联。一旦实现了该关联，并且从组件中抽出所需接口，就能直接开始编写Pascal代码来调用服务，如我们在前面看到的。

用户能填充两个组合框，这将调用TypesList对象方法，由该方法在字符串内返回可用货币的列表（用分号分隔）。该列表将使用换行符来代替分号，然后将多行字符串直接赋到组合框的条目中：

```
procedure TForm1.Button2Click(Sender: TObject);
var
    TypeNames: string;
begin
    TypeNames := ConvIntf.TypesList;
    ComboBoxFrom.Items.Text := StringReplace (TypeNames, ';', sLineBreak,
        [rfReplaceAll]);
    ComboBoxTo.Items := ComboBoxFrom.Items;
end;
```

在选择了两种货币之后，我们就能够使用下面的代码进行转换（图23.4显示了这个结果）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    LabelResult.Caption := Format ('%n', [(ConvIntf.ConvertCurrency(
        ComboBoxFrom.Text, ComboBoxTo.Text, StrToFloat(EditAmount.Text)))]);
end;
```

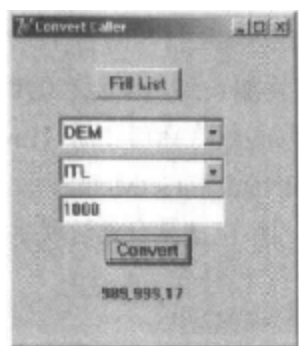


图23.4 ConvertService Web服务的ConvertCaller客户端，显示了德国马克同意大利里拉之间的兑换

请求数据库数据

对于这个范例，我们建立了一个Web服务（基于Web App Debugger），能够显示一位公司雇员的数据。这个数据被映射到InterBase范例数据库的EMPLOYEE表格中，我们在本书中经常使用到这个数据库。Delphi的Web服务接口定义在SoapEmployeeIntf单元中，如下所示：

```
type
  ISoapEmployee = interface (IInvokable)
    ['{77D0D940-23EC-49A5-9639-ADE0751E3DB3}']
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;
```

第一个方法返回了一个公司中所有雇员名字的列表，第二个方法返回了一位指定雇员的详细信息。该接口的实现在SoapEmployeeImpl单元中提供，它使用了下面的类：

```
type
  TSoapEmployee = class (TInvokableClass, ISoapEmployee)
  public
    function GetEmployeeNames: string; stdcall;
    function GetEmployeeData (EmpID: string): string; stdcall;
  end;
```

Web服务的实现有赖于前面两个方法以及一些用来管理XML返回数据的辅助函数。但是在我们的接触范例的XML部分之前，让我们简单讨论一下数据库访问的部分。

访问数据

范例中的所有连接性和SQL代码都在一个数据模块中。当然，也可以在这些方法中动态创建一些连接和数据集组件，但是这样做与Delphi之类的可视开发工具提供的方案相反。该数据模块具有如下所示的结构：

```
object DataModule3: TDataModule3
  object SQLConnection: TSQLConnection
    ConnectionName = 'IBConnection'
    DriverName = 'Interbase'
    LoginPrompt = False
    Params.Strings = // omitted
  end
  object dsEmplList: TSQLDataSet
    CommandText = 'select EMP_NO, LAST_NAME, FIRST_NAME from EMPLOYEE'
    SQLConnection = SQLConnection
    object dsEmplListEMP_NO: TStringField
    object dsEmplListLAST_NAME: TStringField
    object dsEmplListFIRST_NAME: TStringField
  end
```

```

object dsEmpData: TSQLDataSet
  CommandText = 'select * from EMPLOYEE where Emp_No = :id'
  Params = <
    item
      DataType = ftFixedChar
      Name = 'id'
      ParamType = ptInput
    end>
  SQLConnection = SQLConnection
end
end

```

正如大家看到的，数据模块在SQLDataSet组件中具有两个SQL查询。第一个用来获取每一位雇员的名字和ID，第二个返回关于某雇员的整个数据集。

传递XML文档

问题是如何将这个数据返回到远程客户端程序中。在这个例子中，使用了我认为最好的方法：返回XML文档，而不是对复杂的SOAP数据结构进行操作（我不明白为什么XML能作为SOAP引用的消息传递机制——与HTTP提供的传送机制一道——因为它并不适用于被传输的数据。当然，很少有Web服务会返回XML文档，因此我开始对它表示怀疑）。

在这个例子中，GetEmployeeNames方法创建了一个XML文档，它包含了一个雇员的列表，带有他们的名和姓以及作为属性的值和数据库ID，这里使用了两个帮助函数MakeXmlStr（上一章中已经进行了讨论）和下面给出的MakeXmlAttribute：

```

function TSoapEmployee.GetEmployeeNames: string;
var
  dm: TDataModule3;
begin
  dm := TDataModule3.Create (nil);
  try
    dm.dsEmplList.Open;
    Result := '<employeeList>' + sLineBreak;
    while not dm.dsEmplList.EOF do
    begin
      Result := Result + ' ' + MakeXmlStr ('employee',
        dm.dsEmplList.LASTNAME.AsString + ' ' +
        dm.dsEmplList.FIRSTNAME.AsString,
        MakeXmlAttribute ('id', dm.dsEmplList.EMPNO.AsString)) + sLineBreak;
      dm.dsEmplList.Next;
    end;
    Result := Result + '</employeeList>';
  finally
    dm.Free;
  end;
end;

```

```

function MakeXmlAttribute (attrName, attrValue: string): string;
begin
    Result := attrName + '=' + attrValue + '';
end;

```

除了手动产生XML外,我们可以使用XML映射器或者一些其他的技术,但是如第22章所学习到的,我们可能更喜欢直接在字符串中创建XML。我将使用XML映射器来处理在客户端接收到的数据。

说明:读者可能会疑惑,为什么每一次程序都会创建一个数据模块的新实例。这个做法的一个反面效果是每一次程序都要建立一个新的到数据库的连接(一个很慢的操作);而好的方面是我们不会有使用多线程应用程序的风险。如果两个Web服务请求同时被执行,我们可以使用一个共享的连接到达数据库,但是必须使用不同的数据集组件来进行数据访问。可以移动函数代码中的数据集,并且只在数据模块上保留连接,或者对于该连接提供一个共享的全局数据模块(用于多线程)以及一个特定的第二个数据模块的实例,它带有用于每个方法调用的数据集。

现在让我们来看第二个方法,GetEmployeeData。它使用参数查询,并且在不同的XML节点中格式化结果字段(使用另外的帮助函数:FieldsToXml):

```

function TSoapEmployee.GetEmployeeData(EmpID: string): string;
var
    dm: TDataModule3;
begin
    dm := TDataModule3.Create (nil);
    try
        dm.dsEmpData.ParamByName('ID').AsString := EmpID;
        dm.dsEmpData.Open;
        Result := FieldsToXml ('employee', dm.dsEmpData);
    finally
        dm.Free;
    end;
end;

function FieldsToXml (rootName: string; data: TDataSet): string;
var
    i: Integer;
begin
    Result := '<' + rootName + '>' + sLineBreak;;
    for i := 0 to data.FieldCount - 1 do
        Result := Result + ' ' + MakeXmlStr (
            LowerCase (data.Fields[i].FieldName),
            data.Fields[i].AsString) + sLineBreak;
    Result := Result + '</' + rootName + '>' + sLineBreak;;
end;

```

客户端程序(使用XML映射)

在这个范例的最后,要编写一个测试型客户程序。我们可以像通常一样,通过导入定义Web服务的WSDL文件来完成这个任务。在这个例子里,我们必须将收到的XML数据转化

为某些更容易管理的東西，特别是由GetEmployeeNames方法返回的雇员列表。正如前面提到的，我们使用了Delphi的XML映射器，将从Web服务接收到的雇员列表转换为一个可以在DBGrid中显示的数据集。

为了完成这个任务，我首先编写了接收带有雇员信息的XML格式的代码，将它拷贝到一个备注组件中，并从那里复制到一个文件中。接下来，打开XML映射器，载入这个文件，从中生成数据包的结构和转换文件（我们可以从SoapEmployee范例的源代码文件中找到这个转换文件）。为了在一个DBGrid中显示XML数据，程序使用了XMLTransformProvider组件，用以引用转换文件：

```
object XMLTransformProvider1: TXMLTransformProvider
  TransformRead.TransformationFile = 'EmplListToDataPacket.xtr'
end
```

ClientDataSet组件没有和供应器联系起来，因为它将试图打开由转换指定的XML数据文件。在这个例子中，XML数据并没有保存在一个文件中，而是在调用Web服务后传递到该组件。出于这个原因，程序将数据直接转移到ClientDataSet中，其代码如下：

```
procedure TForm1.btnGetListClick(Sender: TObject);
var
  strXml: string;
begin
  strXml := GetISoapEmployee.GetEmployeeNames;
  strXML := XMLTransformProvider1.TransformRead.TransformXML(strXml);
  ClientDataSet1.XmlData := strXml;
  ClientDataSet1.Open;
end;
```

使用这段代码，程序能够在一个DBGrid中显示雇员列表，正如读者在图23.5中看到的。当我们获取给定雇员的信息的时候，程序从ClientDataSet中提取活动记录的ID，然后将XML显示在一个备注之中：

```
procedure TForm1.btnGetDetailsClick(Sender: TObject);
begin
  Memo2.Lines.Text := GetISoapEmployee.GetEmployeeData(
    ClientDataSet1.FieldByName('id').AsString);
end;
```

调试SOAP头部 (Header)

关于这个范例的最后一点说明与使用Web App Debugger来测试SOAP应用程序有关。当然，我们可以从Delphi的IDE中运行这个服务器程序并且轻松地调试它，但是也可以监控通过HTTP连接传递的SOAP头部。虽然从低层来查看SOAP并不太容易，但是这是查清客户端或者服务器应用程序错误的最终办法。例如，在图23.6中，我们看到了一个HTTP日志，它记录了SOAP的请求。

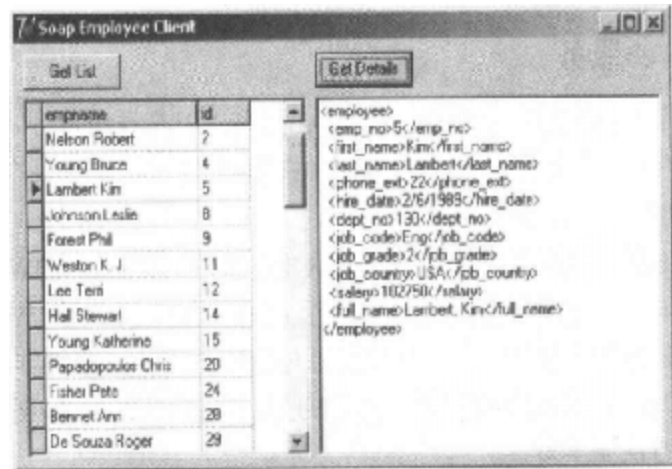


图23.5 SoapEmployee Web服务范例中的客户端程序

Web App Debugger并不总是可用的, 因此, 另一个常用的技术是处理HTTPRIO组件的事件, 就像BabelFishDebug范例中那样。程序的窗体具有两个备注组件, 在这里面, 我们可以看到SOAP的请求以及响应:

```
procedure TForm1.HTTPRIO1BeforeExecute(const MethodName: String;
var SOAPRequest: WideString);
begin
    MemoRequest.Text := SoapRequest;
end;

procedure TForm1.HTTPRIO1AfterExecute(const MethodName: String;
SOAPResponse: TStream);
begin
    SOAPResponse.Position := 0;
    MemoResponse.Lines.LoadFromStream(SOAPResponse);
end;
```

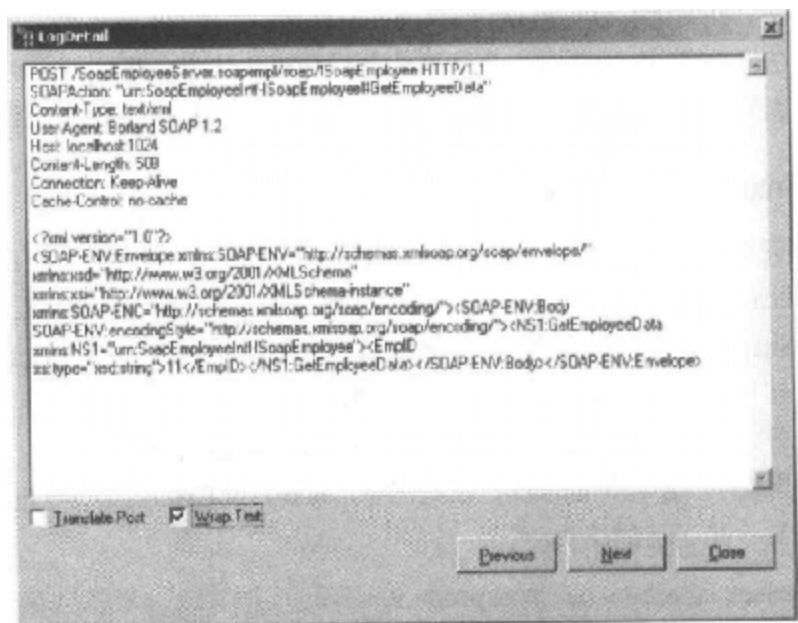


图23.6 Web App Debugger的HTTP日志信息中包括了低层的SOAP请求

将现有的类作为Web服务

虽然有时候我们可能会白手起家建立一个Web服务，但是我们也可以使用现有的代码来完成这个任务。这个过程并不很复杂，因为Delphi在这个领域具有开放的体系结构。为了进行这个操作，我们要经过下面的步骤：

1. 创建一个Web服务应用程序或者添加相关的组件到一个现存的WebBroker项目。
2. 定义一个接口，它继承自IInvokable，并且向它添加我们希望在Web服务中使用的方法（使用stdcall调用约定）。这些方法将与我们想用的那些类非常相似。
3. 定义一个新类，它继承自我们想要使用的类，并且实现接口。这些方法将通过调用相应的基类方法来实现。
4. 编写一个工厂方法，当一个SOAP请求需要一个实现类的对象时，就创建它。

其中最后一个步骤是最复杂的，我们可以定义一个工厂并且对它注册如下：

```
procedure MyObjFactory (out Obj: TObject);
begin
    Obj := TMyImplClass.Create;
end;

initialization
    InvRegistry.RegisterInvokableClass(TMyImplClass, MyObjFactory);
```

但是，这段代码要为每一个调用创建一个新的对象。使用单一的全局对象同样不好：很多不同的用户可能希望使用它，如果这个对象具有状态或者它的方法不是并发的，我们将陷入麻烦之中。为此，我们需要实现一些会话管理功能，它实际上是我们在早期连接数据库的Web服务中遇到问题的变种。

SOAP上的DataSnap

我们现在已经对怎样建立SOAP服务器和SOAP客户有了比较好的了解，接下来让我们进一步学习在建立多层DataSnap应用程序中怎样使用该技术。我们将使用Soap Server Data Module创建新的网络服务，以及连接客户应用程序到其上的SoapConnection组件。

建立DataSnap SOAP服务器

让我们先看一下服务器端。进入New Items对话框的网络服务页中，并首先使用Soap Server Application图标来创建一个新的网络服务，然后使用Soap Server Data Module图标将一个DataSnap服务器端数据模块添加到SOAP服务器中。我已经在SoapDataServer7范例中这样做了（使用Web App Debugger结构用于测试目的）。从这里开始，要做的所有事情就是编写一个普通的DataSnap服务器（或实际的中间层DataSnap应用程序），如第16章“多层DataSnap应用程序”中讨论的那样。在此例中，我通过dbExpress工具将InterBase访问添加到程序中，最终产生下面的结构：

```
object SoapTestDm: TSoapTestDm
    object SQLConnection1: TSQLConnection
```



```

    ConnectionName = 'IBLocal'
end
object SQLDataSet1: TSQLDataSet
    SQLConnection = SQLConnection1
    CommandText = 'select * from EMPLOYEE'
end
object DataSetProvider1: TDataSetProvider
    DataSet = SQLDataSet1
end
end

```

为基于SOAP的DataSnap服务器建立的数据模块定义了定制接口（因此能向它添加对象方法），该接口继承自IAppServerSOAP，现在被定义为公布的接口（即使它不是继承自IInvokable的）。

提示：Delphi 6使用DataSnap的标准IAppServer接口通过SOAP获得数据。Delphi 7使用继承的IAppServerSOAP接口来代替这个默认的设置，它们在功能上相同，但是允许系统根据接口的名称来区别调用的类型。我们将看到如何从一个Delphi 7的客户中调用一个旧的应用程序，因为这个过程不是自动的。

该实现类，即TSoapTestDm，是数据模块本身，如在其他的DataSnap服务器中一样。下面是Delphi生成的代码，以及添加的定制方法：

```

type
    ISampleDataModule = interface (IAppServerSOAP)
    [ '{D47A293F-4024-4690-9915-8A68CB273D39}' ]
    function GetRecordCount: Integer; stdcall;
    end;

    TSampleDataModule = class (TSoapDataModule, ISampleDataModule,
        IAppServerSOAP, IAppServer)
    DataSetProvider1: TDataSetProvider;
    SQLConnection1: TSQLConnection;
    SQLDataSet1: TSQLDataSet;
    public
        function GetRecordCount: Integer; stdcall;
    end;

```

基类TSoapDataModule并非继承自TInvokableClass。只要提供了额外的工厂过程来创建对象（这是TInvokableClass为我们做的）并将它添加到注册代码中（在前面“将现有的类作为Web服务”小节讨论过），这便不成问题：

```

procedure TSampleDataModuleCreateInstance(out obj: TObject);
begin
    obj := TSampleDataModule.Create(nil);
end;

initialization
    InvRegistry.RegisterInvokableClass(
        TSampleDataModule, TSampleDataModuleCreateInstance);
    InvRegistry.RegisterInterface(TypeInfo(ISampleDataModule));

```

该服务器应用程序也发布了IAppServerSOAP和IAppServer接口，这得益于SOAPMidas单元内的代码。作为一个比较，我们能够找到一个SOAP DataSnap服务器，它是由Delphi 6建立的，位于SoapDataServer文件夹。这个范例也可以在Delphi 7中进行重编译并且工作得很好，但是我们应该使用较新的结构来编写新的程序。在SoapDataServer7文件夹中也有一个范例

提示：Delphi 7中的Web服务应用程序能够包括不止一个SOAP数据模块。为了标识一个特定的SOAP数据模块，可使用SoapConnection组件的SOAPServerIID属性，或者将数据模块接口名称添加到URL的末尾。

该服务器具有一个定制方法（Delphi 6版本的程序也具有一个，但是它不能够工作），它利用select count(*) from EMPLOYEE SQL语句使用一个查询：

```
function TSampleDataModule.GetRecordCount: Integer;
begin
    // read in the record count by running a query
    SQLDataSet2.Open;
    Result := SQLDataSet2.Fields[0].AsInteger;
    SQLDataSet2.Close;
end;
```

建立DataSnap Soap客户端

为了建立名为SoapDataClient7的客户应用程序，需要使用普通的程序开始，并向它添加SoapConnection组件（来自组件面板的网络服务页），将它与DataSnap网络服务的URL相连，并且引用正在查找的特殊接口：

```
object SoapConnection1: TSoapConnection
    URL = 'http://localhost:1024/SoapDataServer7.soapdataserver/' +
        'soap/Isampledatabmodule'
    SOAPServerIID = 'IAppServerSOAP - {C99F4735-D6D2-495C-8CA2-E53E5A439E61}'
    UseSOAPAdapter = False
end
```

请注意最后一个属性：UseSOAPAdapter，它表示我们正在操作一个Delphi 7创建的服务器。作为比较，SoapDataClient范例（它使用一个由Delphi 6创建的服务器，并且这个服务器使用Delphi 7重新进行了编译）必须设定这个属性为True。这个值会迫使程序使用普通的IAppServer接口，而不是新的IAppServerSOAP接口。

从此例开始，读者可以照常继续向程序添加ClientDataSet组件、DataSource和DBGrid；为客户数据集选择惟一可用的供应器；并照常连接剩余的部分。对该范例没有可惊讶的，客户应用程序含有一些定制代码：当按钮被单击时，单个调用会打开连接，并且ApplyUpdates调用将把所做改变存回数据库中。

SOAP与其他DataSnap连接

尽管该程序与所有其他的在第16章中创建的数据Snap客户和服务程序在外观上很类似，但一个重要的区别值得强调：SoapDataServer和SoapDataClient程序不使用COM来展开或调用IAppServerSOAP接口。而DataSnap基于套接字和HTTP的连接正相反，仍依赖于本地的

COM对象以及Windows Registry中的服务器注册。然而，基于SOAP的本机支持允许完全定制的解决方案，该方案独立于COM并有更多机会用于其他操作系统（我们可以在Kylix环境中重编译这个服务器，但不是第16章中建立的那个程序）。

客户端程序还可以调用定制方法，我们已将其添加到服务器中，用来返回记录的个数。这个方法可以用来通知用户尚未从服务器上下载的记录的数量。用来调用该方法的客户端代码依赖于一个额外的HTTPRIO组件：

```

procedure TFormSDC.Button3Click(Sender: TObject);
var
    SoapData: ISampleDataModule;
begin
    SoapData := HttpRiOl as ISampleDataModule;
    ShowMessage (IntToStr (SoapData.GetRecordCount));
end;

```

处理附件

Borland在Delphi 7中添加的一个最重要的特性就是完全支持SOAP附件。SOAP中的附件允许我们发送除了XML文本以外的数据，例如二进制文件或者图像等。在Delphi中，附件通过流进行管理。我们可以读取或者指明附件编码的类型，但是需要编写代码来完成从一个原始字节流到给定编码格式的转换。如果考虑到Indy中包含有很多编码组件，那么这个操作也并不非常复杂。

作为如何使用附件的范例，我们编写了一个应用程序，用它转发ClientDataSet的二进制内容（也包括图像）或者单独一个图像文件。该服务器具有下面的接口：

```

type
    ISoapFish = interface(IInvokable)
    ['{4E4C57BF-4AC9-41C2-BB2A-54BCE470D450}']
    function GetCds: TSoapAttachment; stdcall;
    function GetImage(fishName: string): TSoapAttachment; stdcall;
end;

```

GetCds方法的实现使用了一个ClientDataSet，它指向一个经典的BIOLIFE表格，创建一个内存流，向它复制数据并且将这个流附加到TSoapAttachment的结果中：

```

function TSoapFish.GetCds: TSoapAttachment; stdcall;
var
    memStr: TMemoryStream;
begin
    Result := TSoapAttachment.Create;
    memStr := TMemoryStream.Create;
    WebModule2.cdsFish.SaveToStream(memStr); // binary
    Result.SetSourceStream (memStr, soReference);
end;

```

在客户端，我们准备了一个带有连接到DBGrid和DBImage的ClientDataSet组件的窗体。我们要做的就是获取SOAP的附件，将它存入内存的临时流中，然后从内存流中复制数据到本地的ClientDataSet:

```

procedure TForm1.btnGetCdsClick(Sender: TObject);
var
    sAtt: TSoapAttachment;
    memStr: TMemoryStream;
begin
    nRead := 0;
    sAtt := (HttpRIO1 as ISoapFish).GetCds;
    try
        memStr := TMemoryStream.Create;
        try
            sAtt.SaveToStream(memStr);
            memStr.Position := 0;
            ClientDataSet1.LoadFromStream(memStr);
        finally
            memStr.Free;
        end;
    finally
        DeleteFile (sAtt.CacheFile);
        sAtt.Free;
    end;
end;

```

警告：默认情况下，由客户端接收的SOAP附件保存在临时文件中，这与TSOAPAttachment对象的CacheFile属性有关。如果我们不删除这个文件，那么它将会一直存在于这个临时文件目录内。

这段代码产生的视觉效果与客户端应用程序将一个本地文件装载到一个ClientDataSet中相同，正如我们在图23.7中所看到的。在这个SOAP客户中，我们使用了一个HTTPRIO组件，用来监控输入的数据（这可能是非常巨大而且缓慢的）；出于这个原因，我们在调用远程方法之前，将nRead变量设为零。在HTTPRIO对象HTTPWebNode属性的OnReceivingData事件中，我们添加了从nRead变量接收的数据。传递到该事件的Read和Total参数指向由套接字传递的特定数据块，因此它们对于监控的操作几乎是没有什么用的：

```

procedure TForm1.HTTPRIO1HTTPWebNode1ReceivingData(
    Read, Total: Integer);
begin
    Inc (nRead, Read);
    StatusBar1.SimpleText := IntToStr (nRead);
    Application.ProcessMessages;
end;

```

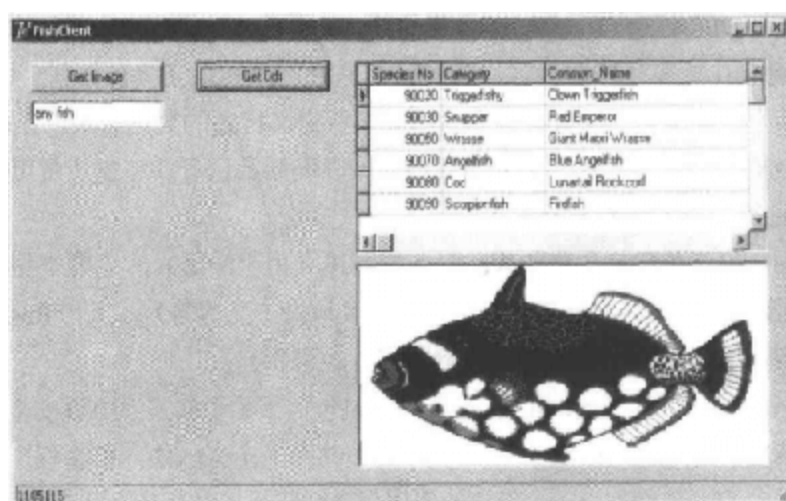


图23.7 FishClient范例在一个SOAP附件中接收了二进制的ClientDataSet

支持UDDI

XML和SOAP的日益流行为B2B通信程序的互操作打开了大门。XML和SOAP提供了一个基础,但是这还远远不够:XML格式的标准化、通信处理的标准化以及商业信息生成的标准化都是现实解决方案的关键组成部分。

在这些克服这个问题的标准程序之中,最著名的是Universal Description, Discovery, and Integration (UDDI, www.uddi.org) 和使用eXtensible Markup Language (ebXML, www.ebxml.org) 的Electronic Business。这两个解决方案有重叠的部分,也有不同的部分,目前正在由OASIS (Organization for the Advancement of Structured Information Standards, www.oasis-open.org) 社团进一步研究开发。我们将不讨论商业操作的问题,而只是介绍一些UDDI的技术元素,因为它是被Delphi 7特别支持的。

什么是UDDI

UDDI, 即Universal Description, Discovery and Integration (统一描述、发现和集成) 规范,是为了创建一个由全球各公司提供的网络服务目录而定制的。它的目的是建立一个开放的、全球化的、与平台无关的框架,使得商业实体能够彼此发现,定义彼此之间如何通过互联网相互操作以及共享全球的商业资源。当然,这个想法也以B2B应用程序的形式加速了电子商务的实现。

UDDI基本上是一个全球的商业注册点。各公司可以在这个系统上进行登记注册,描述它们各自的机构以及它们提供的Web服务(在UDDI的术语中,Web服务的用法很广泛,包括电子邮件地址和网站等等)。UDDI注册表中的信息被分为三块:

白页 包括联络信息、地址等等。

黄页 将公司注册在不同的分类中,包括工业目录、产品分类、地区信息以及其他的分类。

绿页 列举了公司提供的Web服务。每一个服务被列举在一个服务类型中,称为tModel,它可以被预定义或者是一个由公司特别描述的类型,例如, WSDL。

人。因此，我们可以编写自己的UDDI浏览器。

我们来描绘一个简单的解决办法，它是建立一个完整的UDDI浏览器的起点。这个UddiInquiry范例，如图23.9所示，具有一些特性，但是并不是所有的特性都可以顺利地工作（特别是目录搜索特性）。这是因为使用UDDI意味着定位复杂的数据结构，而它并不总是通过WSDL导入器进行映射的。这使得该范例的代码看起来比较棘手，因此，我们将只列出普通搜索的部分代码而非全部，这样做的另一个原因是很多读者对于UDDI并不很感兴趣。

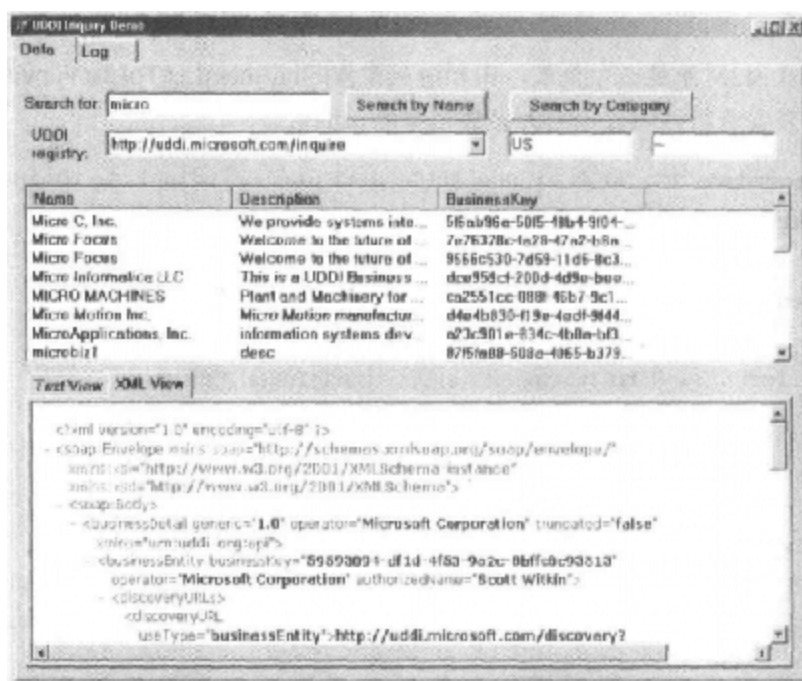


图23.9 带有限制的UDDI浏览器的UddiInquiry范例

在程序开始时，它使用InquireSoap UDDI接口绑定了HTTPRIO组件，该接口定义于Delphi提供的inquire_v1单元中：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    httprio1.Uri := comboRegistry.Text;
    inftInquire := httprio1 as InquireSoap;
end;
```

单击Search按钮，使程序调用find_business UDDI API。因为大多数UDDI函数需要很多的参数，所以使用一个基于记录的FindBusiness类型的参数将其导入进来，并且返回一个businessList2对象：

```
procedure TForm1.btnSearchClick(Sender: TObject);
var
    findBusinessData: Findbusiness;
    businessListData: businessList2;
begin
    httprio1.Url := comboRegistry.Text;
```

```

    findBusinessData := FindBusiness.Create;
    findBusinessData.name := edSearch.Text;
    findBusinessData.generic := '1.0';
    findBusinessData.maxRows := 20;

    businessListData := inftInquire.find_business(findBusinessData);
    BusinessListToListView (businessListData);
    findBusinessData.Free;
end;

```

businessList2对象是一个列表，由程序主窗体的businessListToListView方法进行处理，可在一个列表查看组件中显示了大多数相关的详细信息：

```

procedure TForm1.businessListToListView(businessList: businessList2);
var
    i: Integer;
begin
    ListView1.Clear;
    for i := 0 to businessList.businessInfos.Len do
    begin
        with ListView1.Items.Add do
        begin
            Caption := businessList.businessInfos [i].name;
            SubItems.Add (businessList.businessInfos [i].description);
            SubItems.Add (businessList.businessInfos [i].businessKey);
        end;
    end;
end;

```

虽然该程序只使用普通的文本格式（或者一个基于TWebBrowser的XML视图）来显示最终的XML信息，而且没有进一步对它进行处理，但是通过双击一个列表浏览项目，我们可以进一步浏览它的详细信息。正如前面提到的那样，我们不准备对技术细节做太多的介绍，如果读者感兴趣，可以仔细阅读源代码进行学习。

小结

本章主要涵盖Web服务，包括SOAP、WSDL和UDDI。可参考W3C网站中的UDDI（www.uddi.org）与ebXML（www.ebxml.org）站点，了解面向企业的Web服务领域中的相关信息。这里并未深入探讨这些非技术问题，但是它们也同样值得关注。

在本章中读者可能已经注意到，Delphi在Web服务领域具有强大的开放体系结构，是一个强有力的竞争者。我们可以使用Web服务与为微软的.NET平台开发的应用程序进行互操作。Delphi对这个结构提供了众多的支持，因为它包括了一个Delphi for .NET Preview（我们将在最后两章进行讨论）。

第24章“从Delphi的角度看微软.NET体系结构”将着眼于.NET平台；第25章“Delphi for .NET Preview：语言和RTL”将讨论新的Delphi编译器。

第24章 从Delphi的角度看微软.NET体系结构

每隔几年都会出现一项新的技术，对我们的行业产生巨大的影响。其中的一些技术得到了稳定的发展，还有一些演变为其他的技术，而多数则表现为不成熟的市场产品。在军事应用领域，新技术的应用之前总是面临很多问题。有经验的程序员都知道应该仔细地进行研究，直到其中的一些问题得到解决。毕竟，对这些内容的验证是通过技术试验来完成的。

对Microsoft.NET的反应程度取决于读者的背景。如果读者拥有Java或者Delphi方面的经验，就可能会思考这些混乱的内容都是干什么的。但是，如果读者拥有使用C++编写Windows应用程序的经验的话，那么可能会非常轻松地接受它。

本章的目的就是对组成.NET的一些技术进行解释，并演示它们是如何适应Windows程序设计和Delphi程序设计的。我们首先来安装和配置Delphi for .NET Preview编译器。然后展开来讲解一些.NET技术。最后，更详细地讨论这些技术，并使用一些Delphi代码进行演示。

说明：本章和下一章都是Macro编写的，但是得到了John Bushakra的大力帮助，John工作于Borland的RAD工具文献编制部门。

本章主要包括以下内容：

- 安装Delphi for .NET Preview
- Microsoft的.NET平台
- 中间语言
- 托管代码和CTS
- 无用存储单元收集
- 版本

安装Delphi for .NET Preview

Delphi for .NET Preview需要安装有.NET Framework Runtime，.NET Framework Runtime可以从Microsoft的MSDN网站上自由下载。作为一名开发者，读者可能会采取一些额外的步骤并安装更完整的.NET Framework SDK，SDK为开发者提供了文献说明和工具（当然所占的空间也就更大）。在安装Delphi for .NET Preview之前，应当安装.NET Framework Runtime或SDK。

Preview编译器兼容.NET Framework SDK和服务包1。如果读者已经应用了Service Pack 2或者以后的版本，那么就需要执行一些额外的步骤来重新建立Preview所安装的预编译的单元（dcuil文件）。这个要求在以后的Delphi for .NET Preview编译器中可能会取消。

说明：在Delphi 7发布（2002年11月）后几个月，Borland发布了一个对Delphi for .NET Preview的巨大更新。作为一个注册的Delphi用户，读者可以从Borland Web站点上下载对Preview的更新。甚至在安装Delphi所携带的版本之前就这么做，因为我们需要卸载它，以更新到新版本。

警告：我已经使用了Delphi for .NET Preview 2002年11月的版本来测试本章和.NET章的示例，虽然它们中的多数也可以使用Delphi 7所携带的原始版本进行编译。在大家读到本书的时候，新的版本可能已经出现，读者可以访问作者的Web站点查找更新的示例。

安装完.NET Framework SDK后，插入Delphi for .NET Preview的光碟，并运行安装程序。Preview将把它自己安装在Delphi 7安装之外的一个单独的目录中，并且Delphi 7的任何设置都不会受到Preview安装的影响。

前面我们提到过，如果已经安装了.NET Framework的Service Pack 2，那么应当重新建立编译器所携带的预编译单元。如果是这种情况，打开命令窗口并定位到编译器安装根目录的source\rtl目录下。在那里可以看到一个叫做rebuild.bat的文件，可以执行它来进行重建。读者可能会看到一些错误和警告信息，在编写本书的时候，这些是已知的问题，但它们在编译器后续的更新中将会消失。当批处理文件执行完后，就可以接着往下进行了。

dccil编译器位于bin目录下。除了编译器之外，在这个目录中还有一个叫做dccil.cfg的文件，该文件指定了默认的单元搜索路径（-u编译器开关），实际上就是安装根目录下面的units目录。作为程序包，Delphi for .NET Preview编译器严格地使用命令窗口。

但是Borland已经在Borland Developers Network Web站点上为Delphi 7 IDE提供了一个不被支持的插件，作为Delphi for .NET常用的行编译器IDE集成。使用这个插件，可以在IDE中控制dccil编译器，正如在这个插件所安装的菜单中看到的那样。



但是注意，这个插件并没有提供完整的设计窗体的能力，也没有提供其他Delphi著名的内容。Delphi for .NET Preview编译器会给出一些有关新的语言特性的警告信息，并简单地介绍Delphi运行时库在.NET上下文中可能的样子。读者可以从Borland Developer Network Web站点的Code Central区域（bdn.Borland.com）中下载这个插件（如果找不到它，可以查找编号18889）。

另一个可能会使用到的资源是Reflector（反射器）。这个工具是由Lutz Roeder（顺便说一下，他碰巧在Microsoft工作）编写的，并且在他自己的Web站点（www.aisto.com/roeder/

dotnet) 上提供。

说明: Reflector类似于Microsoft的Intermediate Language Disassembler (ILDASM, 中间语言反汇编器) 工具——它允许用户检查.NET汇编(可执行程序 and DLL) 和它们的类型及成员。

测试安装

现在我们通过编译一个简单的用来打印消息的控制台程序来测试Delphi for .NET Preview的安装。

读者可以使用Delphi 7来启动这个项目, 或者在自己喜欢的编辑器中输入下面的文字:

```
program HelloWorld;
{$APPTYPE CONSOLE}

uses
  Borland.Delphi.SysUtils;

begin
  WriteLn ('Hello, Delphi! - Today is ' +
    DateToStr (Now));
end.
```

注意该代码与过去习惯的Delphi应用程序看上去很相似。惟一不同的是uses子句。Borland已经将Delphi库单元组织成类似于Common Language Runtime (CLR, 公共语言运行时) 的名称空间。在下一章中将会详细讨论这些内容。

将文件保存为HelloWord.dpr, 并打开一个命令窗口。定位到文件所在的位置, 并键入

```
dcc11 HelloWorld.dpr
```

结果应该是一个可执行的映像, HelloWorld.exe, 可以通过从命令行运行它来产生单行的输出。如果正在使用IDE插件, 在启用了IDE hijacking后(读者可以查看上面的屏幕截图中的Hijack IDE菜单项目), 编译以及运行程序就只是按下Ctrl+F9或F9。

如果这个程序看上去很烦人的话, 可以使用HelloWorld程序的Windows版本替换控制台版本来测试.NET的使用。这个程序叫做HelloWin, 它将在屏幕上创建并显示一个窗口, 并将内容写在窗口中。它还显示了对Application.Run的调用, 这个程序用于激活应用程序的消息处理循环:

```
program HelloWin;

uses
  System.Windows.Forms,
  Borland.Delphi.SysUtils;

var
  aForm: Form;

begin
  aForm := Form.Create;
  aForm.Text := 'Hello Delphi';
  Application.Run (aForm);
end.
```

在.NET表示中,前面的代码显示出了程序员对使用Microsoft开发工具的热情。作为Delphi程序员,从1995年以来可能使用的几乎都是同样的代码,因此大家可能会思考这种热情到底来自于哪里。这些程序的输出离有意思还差很远,但是至少可以证明它们并不是在其上运行ILDASM(直接通过插件的菜单来执行)的传统的Win32应用程序。图24.1演示了HelloWorld程序的输出。注意单元的全局代码被包装到一个叫做uUnit的伪类中。与Delphi不一样,.NET运行时并不计算全局过程和函数,只是计算类方法。下面让我们从了解.NET平台开始。

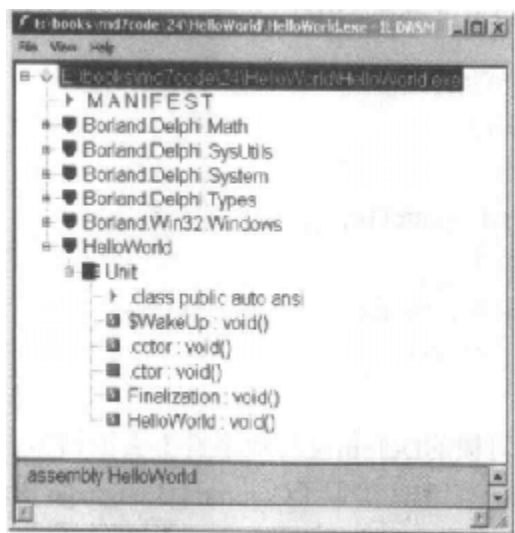


图24.1 HelloWorld的演示

Microsoft的.NET平台

Microsoft的.NET平台由许多不同的规范和创新组成。.NET平台的核心功能已经移交给European Computer Manufacturer's Association (ECMA, 欧洲计算机制造商协会),并且正处于标准化的过程中。在编写本书的时候,C#语言规范刚刚通过ECMA的处理,并且正在成为ISO标准。

说明:读者可以在www.ecma.ch上找到关于C#编程语言和Common Language Infrastructure (CLI, 公共语言基础结构)的标准文档。有关CLI规范的一个有意思的元素就是对变量和方法的标准命名约定。

从程序设计人员的观点来看,.NET平台的核心特性是它的托管环境,这个环境允许执行使用其他程序设计语言所编译的中间代码(倘若它们遵循基础数据类型的公共定义的话)。这个环境嵌入了很多的特性,范围从复杂的内存管理到集成的安全性。在这个托管环境的顶端,Microsoft建立了一个大型的类库——涵盖了各种不同的开发领域(基于Windows的窗体、Web开发、Web服务开发、XML处理、数据库访问等等)。

这只是一个简单的概述。要想获得更详尽的信息,我们需要了解在.NET平台中使用的精确术语,它们中的多数都是通过三个字符的缩写来表示的,在后面的小节中将会进行介绍。

公共语言基础结构 (CLI)

CLI是.NET平台的主要支柱。它在2001年12月被提交给ECMA。CLI由很多不同的规范组成:

公共类型系统 (CTS) CTS (Common Type System, 公共类型系统) 为完全不同的程序设计语言的集成打下了基础。它规定了类型是如何声明和使用的, 并规定了语言编译器必须符合这个规范才能利用.NET平台的跨语句集成。

扩展元数据 CLI规定了每个部署单元(程序集)必须是自描述的实体。因此, 程序集必须携带能够完全标识出程序集(它的名称、版本以及可选的文化和公钥)的数据, 描述在程序集中所定义类型的数据, 列出其他引用的文件以及所需要的特殊安全权限的数据。此外, 元数据系统是可扩展的, 因此程序集可能也包含用户定义的描述元素, 也就是自定义属性。

说明: 术语“文化”被Microsoft所使用, 用来作为术语的语言(用户消息所使用的语言)和区域(国家的日期和数值格式设置)的扩展。这个理念是为了涵盖某个国家或者国家的某个地区专用的东西。

公共中间语言 (CIL) CIL是带有抽象微处理机的虚拟执行环境的程序设计语言。把.NET平台作为目标的编辑器不产生本机CPU指令, 而是用抽象处理机的中间语言产生指令。在这个关系上, CLI类似于Java字节代码。

P/Invoke 在.NET运行时中执行的程序都在它们自己专用的沙箱(sandbox)中运行。与传统的Win32程序设计模型不同, 在它们之间以及和操作系统之间有大量复杂的软件层存在。但是.NET运行时并没有完全替代Win32 API, 因此在这两个世界之间必须有一座相互沟通的桥梁。这个桥梁就叫做Platform Invocation Service (平台调用服务, 简称为P/Invoke或者PInvoke)。

框架类库 (Framework Class Library, FCL) .NET Framework Class Library (FCL) 或者简称之.NET Framework (或者是.NET Fx) 是一个类的层次结构, 它的设计类似于Borland的VCL。在VCL中可用的特性也相当相似, 虽然.NET Framework有更多的类集(涵盖了更多的程序设计领域)。.NET Framework和VCL的层次结构的不同之处在于, .NET Framework不仅可以从其他语言中调用, 而且还可以直接被其他语言扩展。这意味着, 作为Delphi程序员, 可以直接继承.NET Framework中的类, 就像继承其他Delphi类一样。此外, CLI能够扩展由其他任何语言(该语言要拥有以.NET运行时为目标的编译器)编写的类。FCL的可分解部分意味着类层次的那个部分可以被提出, 例如, 创建可以在手持设备上使用的精简版本。

扩展的可移植的执行文件格式 Microsoft正在使用它标准的Portable Executable (可移植的执行体, PE) 文件格式(该文件格式被标准的Win32可执行文件使用)来支持CLI的元数据需求。使用这个格式的优点是操作系统加载.NET应用程序的方式和加载本机Win32应用程序是一样的。常规加载关心的是相似性在哪里结束, 因为所有的托管代码都位于一个特定的部分。当框架发现它正在处理.NET实体时, 它会修改加载器, 将控制权交给CLR, 以处理如何调用管理入口点。

公共语言运行时 (CLR)

CLI是一个规范, CLR是该规范的Microsoft实现。不要惊讶, CLR是规范的超级。对于程序员而言, CLR是一个全封装的运行时库, 它将Windows操作系统所提供的各种各样的服务统一在了一起, 并通过一个面向对象的框架将它们展现在用户面前。

从更大范围上来说, CLR负责的是.NET世界中的各个方面: 加载可执行映像, 验证它的身份和类型安全, 将CIL代码编译成本机CPU指令, 以及在应用程序的存活期内对其进行管理。要在CLR中运行的CIL代码叫做托管代码, 而其他代码(例如Delphi 7所产生的Intel可执行代码)是非托管代码。

公共语言规范 (CLS)

CLS与公共类型系统(Common Type System)密切相关, 它是规范的一个子集, 定义了在不同程序设计语言中所创建的类型如何相互合作。并不是所有语言都是被平等创建的, 有些语言所具有的特性在其他语言中是找不到的。CLS就是要找到一个媒质, 规定大部分语言可以实现的项目。Microsoft曾尝试使得CLS在尽可能小的同时, 仍然能够适应大量的语言。

Delphi的一些特性并不是与CLS兼容的。这并不意味着代码无法被CLR执行, 以及无法在.NET平台上工作。它只意味着程序员正在使用无法被其他语言支持的语言特性, 因此如果其他人是用Delphi以外的其他语言编写程序的话, 那么代码的特定部分将无法被其他.NET应用程序使用。

说明: 如果读者拥有使用Java的经验, 就会发现其中的一些项目很熟悉。 .NET平台和Java平台共享了很多的概念, 特别是中间语言和虚拟执行系统, 虽然两者的主要不同在于, 在默认情况下Java是根据代码进行解释的, 而.NET总是JIT编译的。在类库方面它们也有许多相似之处。但是在多数情况下, 不要将相应的Java概念当做是相同的, 因为在具体实现上的差异对于如何使用那些看上去类似的特性可能会产生严重的影响。

CLI的各种规范为跨平台开发提供了希望。但是, 读者可能没有听Microsoft说过“只要写一次, 就可以在任何地方运行”这样的话。这是由于他们认为用户界面是应用程序的关键部分, 例如PC屏幕和移动电话的屏幕具有完全不同的能力。现在有两个方面正在努力, 以便能够在其他操作系统上实现CLI。Rotor项目(官方称之为Microsoft Shared Source CLI或MSSSCLI)是Microsoft Research团队编写的。Rotor由大量的软件(这些软件都遵循Microsoft的Shared Source许可)以及在FreeBSD、Win2和Mac OS X 10.2操作系统上建立的必要工具组成。

第二个著名的在其他操作系统上的CLI实现是Mono Project, 它支持Win32和Linux。Mono正在Linux上建立CLI实现所需要的内容, 并且它的发布带有更多开发的许可证。Borland对Mono持一个观望的态度, 在该舞台上Delphi是否会成为其中的一员还没有相关的说明。

看上去Rotor项目把目标锁定在那些对CLR如何实现相当关心的教育家以及其他一些人身上, 因为它的许可证对学术环境是相当开放的, 但是对商业使用而言则是禁止的。但是Mono项目可能会给Microsoft带来一些激烈的竞争。对于完全可移植的能力而言, 最严重的问题仍然可能是图形化的用户界面。Rotor项目并不包含任何GUI元素, 而Mono已经开始使用WINE库来开发WinForms了。与Mono相关的还有一个GTK#项目, 它是一个对C#语言的

GTK+工具包绑定。

今天的计算环境具有各种变换的可能性。手持设备本身具有复杂的操作系统，并且Microsoft对这个领域已经产生了浓厚的兴趣。Microsoft正在验证一种叫做.NET Compact Framework的将要用于手持设备的.NET平台。.NET平台可能会成为明天技术的起点。64位处理器正在走向舞台，.NET无疑将可以在上面运行。

这是否意味着可以在Linux、64位Windows和PDA上运行自己的托管应用程序吗？不，期待能够将21英寸监视器上显示的1600×1200的用户界面移植到手持设备上是不合理的。因此，使用.NET作为一个跨平台的工具，我们可以从中获得很多的好处，但是不应当期待可以将应用程序直接移植到其他的操作系统或者手持设备上。

在下面的各节中，我们将更详细地测试CLI的一些组件。

程序集

Microsoft使用了术语“程序集（assembly）”在.NET运行时环境中表示“部署单元”。在制造业，指的是一组单独的但是又相关的部分，将它们组合在一起就形成了一个单个的功能单元。有的时候，它只包含一个部分，但是有的时候它也可能由多个部分组成。

一个程序集可以只包含一个文件，可以是单个的可执行程序或者是动态链接库。程序集还可以由一组相关的DLL组成，或者是由一个DLL及其相关的资源（如图片或字符串）组成。返回到制造业，类似地，每个部署单元不管是由一个部分还是由多个部分组成，都包含了一个叫做“装箱单”的包装清单。装箱单描述了所组成的内容，并且包含了部署单元的元数据。装箱单是在CLI中所描述的可扩展的元数据系统的实现。

当查看Delphi for .NET Preview编译器的安装目录时，会发现一个叫做unites的文件夹。在这个文件夹中有大量的扩展名为.dcu和.dcuil的文件。这些并不是PE文件，因此不能使用ILDASM进行测试。

有一个.dcu文件编录了在某个.NET部件中的名称空间和类型（记住，程序集可以包含多个可执行体）。而一个.dcuil文件相对应于名称控件。该.dcuil文件包含名称控件的所有编译器符号，以及对提供给该名称空间（程序集可以将类型提供给多个名称空间）的.dcu文件的参考。

说明：名称空间是一个层次化的保存机制，它用来组织类和其他类型。我们将在第25章“Delphi for .NET Preview：语言和RTL”中详细讨论名称空间。

应用程序是根据一套特定的程序集建立的，应用程序将引用这些程序集。当这套程序集发生变化时，编译器必须重新建立在集合中不再存在的.dcuil文件（包含对.dcu文件的引用）。类似地，如果程序集发生了变化，编译器必须重新建立与程序集相应的.dcu和.dcuil文件。.dcuil文件大体上等价于Delphi的.dcu文件，但是在Delphi for Win32环境中，.dcu文件并不拥有相应的文件类型。

不同的项目类型产生不同类型的.NET程序集：Delphi程序产生可执行的程序集，库项目则产生DLL程序集。

中间语言

公共语言运行时是虚拟执行系统或者说虚拟机的实现。和所有虚拟机一样，CLR有它自己的抽象微处理器。正如前面已经提到的，虚拟处理器的部件语言叫做Common Intermediate Language（公共中间语言，CIL），但是在它成为标准之前，被称为Microsoft Intermediate Language（Microsoft中间语言，MSIL），这个术语现在仍然能够经常看到。以CLR为目标的编译器并不会在任何真正的微处理器本机指令集中产生代码，而.NET编译器则以CLR的抽象处理器为目标。在CLR中建立的硬件抽象似乎具有某些跨平台的生存能力。记住，Microsoft的CLR只不过是CLI的一个实现，任何具有兼容CLI的执行环境的硬件/操作系统平台都可以是目标。

当然，并没有真正的微处理器可以直接执行CIL，因此在执行它之前，必须将它编译成本机硬件的指令集——这是Just in Time（JIT）编译器的工作。下面是CLR与其他虚拟机实现（如Java）的不同之处。CLR并不是一个解释器，它也不执行字节码。在.NET平台上，CIL总是编译成本机CPU指令，并且一旦被编译，它就缓存在内存中，因此不再需要重新编译了。

说明：在有些内存受限制的环境中（例如PDA），编译代码可以被丢弃。在这种情况下，如果代码被重新加载，那么就需要重新进行编译。

编译IL并不是一个非常复杂的操作（MS Research已经花费了多年的时间来开发一种技术，以便能够使JIT编译能够尽可能地被忽略不计），但是仍然需要花费一定的时间，甚至在每次运行同样的程序时，都必须要重复进行操作。多数应用程序在启动时都会花费一定的时间（特别是在Windows会话中使用第一个.NET应用程序加载所有的.NET框架的情况下，就变得非常慢），但是这是有一定限制的，因为并不是在应用程序中所有代码都是立即编译的。JIT编译器和加载器协同工作，因此IL代码将不会被编译，直到它被调用为止。

JIT编译在.NET平台是常见的情况，但是它也可以将托管的执行程序编译成本机指令，并将本机图形存储在磁盘上。这样做就可以避免JIT编译在应用程序启动时所带来的负面影响。 .NET框架运行时包含一个叫做Native Image Generator（本机映像生成器，Ngen.exe）的实用工具来完成这个目的。

Ngen所创建的本机映像被存储在本机映像缓存中。下次CLR尝试加载程序集时，它将首先查看本机映像缓存。如果部件的本机映像被找到，它将首先传递给IL版本。注意，必须部署程序集的IL的版本，因为本机映像并不包含任何元数据。另外，终端用户或者管理员可以从缓存中删除本机映像。在这种情况下，就找不到本机映像了，CLR将返回到IL版本的JIT编译。

产生本机可执行映像的能力是很有用的，但是应当在两个环境（JIT和本机映像）中对应用程序进行配置，看看JIT编译器的不好效果是否就是那么坏。毕竟JIT编译器是用于特定微处理器的真正编译器，因此它能够自己进行一些性能优化——使用好的算法来降低开销，并且为编译代码引入最优化的方式（类似于Delphi在它的编译中所做的）。

查看编译器所产生的IL代码需要有很深的知识。Microsoft提供了一个实用工具叫做IL Disassembler（IL反汇编器，ildasm.exe），可以使用这个工具在最低层对程序集进行解析。ildasm位于.NET Framework SDK安装的bin目录下，它可以加载任何程序集以及元数据；

装箱单、类、它们的方法和属性，当然还有编译器所产生的IL代码。在本章和下一章中读者将会更多地了解ILDASM。而前面所提到的Reflector工具也非常有用。

托管代码和安全代码

简单地说，托管代码是任何被加载、经过JIT编译以及在CLR帮助下执行的代码。与所有Windows平台上的可执行体和库模块一样，托管代码模块是以Microsoft的Portable Executable (PE, 可移植执行体) 文件格式存储的。托管PE文件包含额外的头部信息，并且在被加载时，它将跳入运行时的执行引擎中（跳至MSCorEE.dll的一个函数中）。

运行时可初始化，然后查找模块的入口点。在入口点中的IL代码是JIT编译的本机CPU指令。最后，在模块的入口点开始执行。这种情况类似于库模块，PE文件指导加载器跳转到MSCorEE.DLL的一个不同的函数中。

反之，非托管代码并不包含IL，而是包含传统的本机CPU指令。非托管代码在运行时外执行，因此无法利用CLR所提供的服务——至少是在不采取特殊方式的情况下。

非托管代码能够使用COM Interop服务创建.NET Framework类。.NET Framework类被封装在一个COM代理中，并被暴露给非托管代码，就像它是COM对象一样。COM Interop bridge采用另一种方式，允许COM服务器中的COM对象可以被非托管代码访问。最后，CLR的Platform Invoke服务运行托管代码，以直接调用Win32 API。

说明：Delphi for .NET Preview编译器产生完全的托管代码。和Microsoft的Visual C++ .NET一样（它使用了一个叫做IJW的机制，IJW是It Just Works的缩写），当前并不支持在同一个模块中将托管代码和非托管（本机）代码混合起来。

模块是完全自描述的，因为它既包含IL代码，也包含描述代码所使用的数据元素的元数据。将IL代码和元数据放在一起，CLR能够在编译器所能完成的静态检查之外执行另一种层次的验证。这个过程通常要执行，除非系统管理员关闭它，它用来验证代码是否是安全类型的。被验证为安全类型的代码叫做安全代码。安全代码要通过下列安全类型的检查：

- 在对象上只有有效的操作被调用。这包含参数验证、返回类型验证和可视性检查。
- 对象总是被指派给兼容的类型。
- 代码使用不明确的指针，因为它们可能引用无效的存储位置。

不安全的代码不能通过这些检查。但是，代码没有被验证为安全类型并不意味着它是不安全的，它只意味着代码不能被验证，可能是由于验证过程的限制，也可能是编译器本身的问题。当它作为最终产品发布时，Delphi for .NET编译器应该可以产生验证为安全类型的代码。

有些Delphi语言构造并不是兼容CLS的，但是这与不能被验证为安全类型有所不同。不兼容CLS的语言构造将在第25章中详细讨论。

.NET Framework SDK包含了一个PEVerify实用工具，它可以出于安全性的目的分析托管PE (peverify.exe)。前面提到过的Delphi 7 IDE插件允许程序员自动在代码上运行PEVerify。

公共类型系统

Common Type System (公共类型系统) 就像是推土机, 它为.NET框架中的程序设计语言推平了一块平坦的领域。CTS规定了CLR所需要知道的基本类型和对象。这些类型被用于定义对象模型, 这个模型在所有以CLR为目标的语言中共享。

Component Object Model (COM, 组件对象模型) 已经成为常用的在Windows平台上实现二进制兼容性和语言互用性的方式。除此之外, CTS允许语言(如Eiffel、C#和Delphi)相互进行结合。用这些完全不同的语言所编写的组件可以在它们之间传递对象, 并且能够通过继承性直接扩展它们的能力。这种程序设计语言的集成程度可以说是空前的。

在CTS中定义的所有类型被分成了两类: 数值类型和引用类型。数值类型正如它们的名字所表示的那样, 拥有按值传递的语义。例如, 假设有一个变量是数值类型。如果将这个变量作为函数的参数传递, 并在函数中修改参数, 原始的值是不受影响的。数值类型的例子包括标量类型、枚举和记录。集合类型(如Delphi记录或C#结构)是CTS中的数值等级。

在另一方面, 引用类型拥有别名, 或者按引用传递的语义。如果拥有引用类型的变量(例如类的一个实例)并且将该变量作为参数传递给函数, 对参数的任何更改都会影响到变量。引用类型的例子包含类的类型和接口。指针类型也是引用类型, 另外还有委托, 稍后将对其进行讨论。

对象和属性

类似于Delphi, CTS实现了一个单继承的模式。类必须从一个并且只能从一个祖先那里继承, 并且可以将它声明为零或者多个接口的实现。其他熟悉的面向对象CTS的属性是类和类成员(具有其他可用的可视性说明符, 将在下一章中讨论)的专用性、公共性和受保护的可视性。这些CTS可视性说明符的含义类似于Delphi中的说明符, 但是它们更受限制, 符合C++语义。(在第25章我们将讨论Delphi的可视性说明符如何映射到CTS版本, 并对语言所做的特定更改进行测试, 以便符合CTS额外的特性。)

当细读.NET文献的时候, 我们将会注意到CTS类的类型和Delphi类的性能之间的相似性。传统字段和方法的面向对象特性当然是被支持的。另外在某种程度上, CTS实现属性在概念上类似于我们所熟悉的Delphi通知。在CTS中的属性可以拥有读取和写入访问方法(可以即时限定和计算数值), 或者它们可以简单地屏蔽专用的字段。但是, 它们也有很多不同, 包括属性获取/设置(get/set)方法必须拥有和属性本身同样的可视性, 这样才能确保不支持属性语义的语言仍然可以访问属性。虽然Delphi并没有在源代码中强制使用这一点, 但是如果需要, 它会在后台修改编译代码。

事件和委托

Win32 API之所以仍然存在的一个原因就是它在它的最低层, 它是基于基础概念的, 例如使用函数的地址作为回叫机制。整个Windows用户界面事件系统是基于回叫功能的(并且在VCL框架中的某些事件是建立在那个系统之上的)。回叫机制非常有用, 当然会在CTS中使用。以安全类型、中立语言的方式使用回叫依赖于叫做委托的引用类型。

CTS委托与平常的函数指针不同，平常的函数指针可以引用类的静态和实例方法。委托的声明必须匹配委托将引用的方法签名。在Delphi for .NET中，委托的使用类似于大家所熟悉的过程类型：

```
type
  TMyClass = class
  public
    procedure myMethod;
  end;

var
  threadDelegate: System.Threading.ThreadStart;
  tmc: TMyClass;
  aThread: System.Threading.Thread;

begin
  tmc := TMyClass.Create;
  threadDelegate := @tmc.myMethod;
  aThread := Thread.Create (threadDelegate);
  aThread.Start;
```

threadDelegate变量的类型为System.Threading.ThreadStart（这是一个CLR委托类）。指派给委托的方法拥有的签名与委托的相匹配，在这个例子中是没有参数的过程（在Delegate示例文件夹中可以找到这个代码片段）。

编译器在这里隐藏了大量的复杂性。在后面，编译器必须创建类System.MulticastDelegate的实例。使用MulticastDelegate类上的方法调用委托——正在被封装的函数（在这个例子中是myMethod）。这个例子支持在单个委托中同时封装多个函数。在用户接口事件模型中，它被转化成一个事件拥有多个监听器。

说明：顺便说一下，委托是类的这一事实说明了为什么可以在类的范围之外声明委托。因为委托是System.MulticastDelegate（或者编译器产生的后裔类）的实例，就可以在任何声明类的地方声明它们。

每个特定的语言编译器都实现某些语义形式，以此来创建事件和额外的内容，并且减少删除事件监听器所带来的麻烦。例如，在C#中，Microsoft使用+=和-=操作符从底层委托中添加和删除函数。Delphi for .NET出于同样的目的使用了一套语言。在第25章，我们将探讨这个主题，演示函数Include和Exclude如何用于指派事件处理器。我们还将看看Delphi的:=指派操作符是如何工作的，并关注在.NET世界中如何指派事件处理器。

无用存储单元收集

无用存储单元收集是CLR的一部分，它对内存分配和解除分配执行自动管理。在这个方面，Delphi 7所提供的内容涉及基于接口的变量（有关的详细内容见第2章）。当对象没有更多的引用时，它所使用的内存将会被回收。这也是CLR无用存储单元收集的基础理念，但是相似之处就此为止。CLR的无用存储单元收集能够检测到两个对象除了能够互相引用外没有其他引用的情况，因此它们可以被丢弃，这一点是其他的引用计数系统（如Delphi）所无

法做到的。

使用无用存储单元收集意味着可以创建对象，根据需要分别引用它们，并且可以忘记删除这些对象。系统将会完成程序员所需要做的事情，而且并不需要程序员真正地知道。注意这意味着我们不需要在代码体系结构中对对象的创建和消除进行平衡，也不需要使用`try/finally`块来确保对象被破坏。得益于无用存储单元的收集，我们只需要面对几个特定的环境。

只有在编写低级类时才需要记住要清空非托管资源，例如窗口或者文件句柄。CLR类`System.Object`包含了一个受保护的`Finalize`方法，可以使用该方法进行覆盖来清理非托管资源。在开始覆盖`Finalize`之前，读者需要了解一些重要的内容（如果想了解关于这项内容的详细信息，可以参阅附加说明“关于覆盖`Finalize`的问题”）。

关于覆盖`Finalize`的问题

覆盖`Finalize`方法效率很低，因为它需要对象至少通过无用存储单元收集器产生两个行程。无用存储单元收集器的工作是回收内存，并且它无法这么做直到它运行对象的`Finalize`方法为止。因此无用存储单元收集器必须特殊对待带有`Finalize`方法的所有对象，亦即把它们放置在一个特殊的清单中，但是通过这一回合的无用存储单元收集，对于使它们保持有效而言会带来负面的影响（因为现在对该对象还有其他的引用）。最终，一个线程会检查这个特殊的清单，运行每个对象的`Finalize`方法，并从清单中删除对象。从清单中删除对象将会解除对对象最后的引用，因此下次无用存储单元收集器运行时，对象的内存将会被回收。

但是效率低下并不是`Finalizer`唯一的问题，用户也无法确保对象的`Finalizer`会运行。如果依赖于`Finalize`方法来归还非托管资源，对象占有这些资源的时间可能会超过我们所期待的时间。

当`Finalize`运行时，它并不是和对象运行在同一个执行线程中。这意味着所有的线程同步和阻塞必须要避免，因此它并不是你正在阻止的线程，它是CLR的`Finalizer`线程。如果从CLR捕获到的`Finalize`中抛出了一个异常，这个异常将被忽略。

我们所推荐的清理非托管资源的方法是实现处理模式。这个策略是为对象提供两种自动的方式，一种方法是当它被清理回收时除去它所涉及到的外部资源，另一种方法是在直接的方法调用之上清理同样的资源。为这个模式编写代码就等于实现一个叫做`IDisposable`。`IDisposable`的接口，它由一个方法组成：`Dispose`。`IDisposable`接口为程序员提供了确定的清理对象所用资源的控制权。与`Finalize`不同，`Dispose`方法是公共的，并且是被用户调用而不是被无用存储单元收集器调用的。

读者可能已经阅读了关于C#语言如何通过让编译器在后台创建`Finalize`方法来实现析构器的.NET文献。要在C#中实现`IDisposable`接口，必须直接并将所有的内容捆绑起来，这样`Dispose`方法才可以被直接调用，或者通过自动产生的`Finalize`方法调用。这和Delphi for .NET所工作的方式不同。

为类创建析构器并不会导致编译器在后台实现`Finalizer`。它会导致编译器实现`IDisposable`接口。但是没有事情能够阻止直接实现`IDisposable`，因此如果想依赖编译器后台处理的能力，就必须遵循严格的模式。应按照如下方式明确地声明类析构器：

```
destructor Destroy; override;
```

当编译器看到这个声明时，它将产生代码使类作为IDisposable的实现器。然后，使用CIL的一个特性，将析构器标记为Dispose方法的实现（即便该方法的名称是Destroy，而不是Dispose也可以完成）。

在Delphi中除去对象的常用方法是调用对象上的Free方法。在Delphi for .NET中，Free方法被实现，这样它将测试查看对象是否实现了IDisposable接口。如果遵循前面的析构器签名，编译器就会实现它。Free接着将调用Dispose，Dispose将在Destroy方法中终结。

让我们创建一个项目并使用遵循模式的析构器声明类。然后使用ILDASM来检查产生的代码。该代码列在程序清单24.1和程序清单24.2中。

程序清单24.1 DestructorTest示例项目

```
program DestructorTest;

{$APPTYPE CONSOLE}

uses
  MyClass in 'MyClass.pas';

var
  test : TMyClass;

begin
  WriteLn ('DestructorTest starts');
  test := TMyClass.Create;
  test.Free;
  WriteLn ('DestructorTest ends');
end.
```

程序清单24.2 DestructorTest示例的单元

```
unit MyClass;

interface

type
  TMyClass = class
  public
    destructor Destroy; override;
  end;

implementation

• destructor TMyClass.Destroy;
begin
  WriteLn ('In destructor (which is actually the IDisposable.Dispose method)');
end;

end.
```

编译这个程序后即可运行它，但是我们对运行该程序的兴趣并没有使用ILDASM检查它那么高。启动ILDASM并选择File►Open。定位到DestructorTest.exe所在的目录并打开它，将会看到如图24.2所示的窗口。



图24.2 DestructorTest示例的ILDASM输出

这一树项被标记为MyClass，它表示的是被创建的用来保存单元中所有符号的名称空间。展开MyClass节点会发现TmyClass和Unit条目。Delphi for .NET为每个单元创建了一个CLR类，用以实现初始化和结束，并允许程序员仍然编写全局例程，而它们成为Unit类的方法。展开TmyClass节点，会发现Destroy节点，它有一个粉色的块。这个节点表示TmyClass中的Destroy节点。双击Destroy节点将会打开一个窗口，这个窗口显示了这个方法完整的IL代码，如图24.3所示。

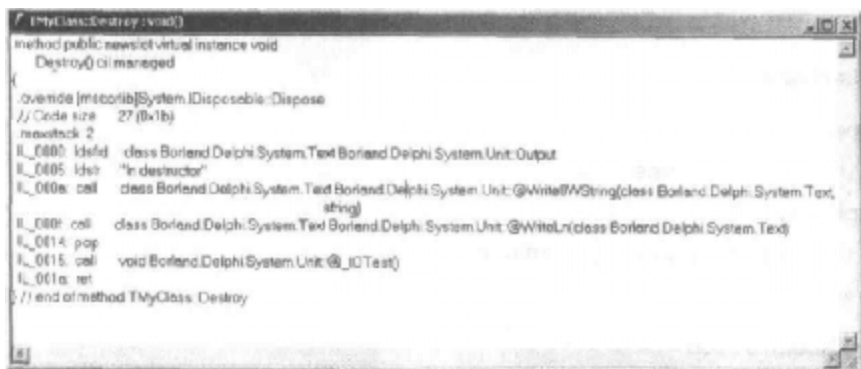


图24.3 Destroy析构器的ILDASM IL窗口

我们正在查找的行使用了.override指令：

```
.override [mscorlib]System.IDisposable::Dispose
```

这是一个明显的覆盖，它表明Destroy方法是IDisposable接口中Dispose的实现。.override指令表示的是当从Free调用Dispose时，执行线程如何在析构器中结束。

说明: IDisposable被处理的方式演示了在Delphi for .NET编译器上正在进行的工作中所常见的一种模式。与C#不同, Delphi是一种老的语言, 它有大量忠实的追随者, 并且具有相当大的代码基础。Borland无法清除现在所有的东西, 并采用一种新的方式打破现在的代码。IDisposable的实现方式应当允许Delphi程序员继续使用熟悉的范例, 并且尽量减小将现有代码移植到.NET平台所带来的影响。

无用存储单元的收集和效率

在程序员中, 对无用存储单元收集(GC)的使用是很有争论的。多数程序员喜欢GC能够帮助他们减少内存管理错误的机会。但是有些程序员害怕GC可能不是很有效。这种担心经常阻止了这项技术的广泛使用。早期Java虚拟机GC的效率确实低下, 在这个方面并不能带来真正的帮助。Microsoft花费了大量的时间来说服所有的程序员使用它的GC, 但是他们过高地估计了他们的解决方案, 并将其描述为完美的。甚至在Microsoft的关于GC的技术文档中, 都可以发现大量骗局和少量事实。

说明: 这并不是说GC并不能完成适当的工作, 实际情况正好相反。但是它的工作情况与它所表现的有些不同。目前, 想知道GC是如何工作的惟一方式就是编写案例程序并研究它们的效果。

作为进行探讨的起点, 读者可以使用GrabageTest示例。使用一个Windows内存分析工具来查看内存是如何受程序执行影响的。GarbageTest示例声明了如下所示的大对象的类(大约10KB):

```
type
  TMyClass = class
  private
    data: Integer;
    list: array [1..10240] of Char;
    s: string;
  public
    constructor Create;
  end;
```

最简单的测试代码如下所示:

```
for i := 1 to 10000 do
begin
  mc := TMyClass.Create;
  WriteLn (mc.s);
end;
```

在Delphi中编写这个简单的程序将会给内存带来洪水般的冲击, 因为这里没有对Free的调用。但是在.NET中, GC会重新使用单个的内存块, 因此内存消费是平坦的。如果按照如下方式将每个对象保存在数组中:

```
objlist: array [1..10000] of TMyClass;
```

那么在每个循环周期, 内存消耗将会增加, 这样就会导致问题的产生。在内存中保留对象引用, 然后有规律地或者随机地释放它们能够提供更复杂的测试。在程序代码中, 读者将会发现几个小片段, 但是大家需要使用自己的知识和想像能力来建立其他有效的测试场景。

部署和版本确定

多数Windows软件开发人员在某种程度上都会感到部署共享库所带来的麻烦。.NET Framework能够适应大量不同的部署场景。在最简单的场景中,终端用户能够将文件复制到一个目录中并运行。当需要从计算机中删除应用程序时,终端用户就可以删除文件,或者删除整个目录(如果它不包含任何数据的话)。在这个场景中,没有安装程序,没有卸载程序,也不需要处理注册表。

提示: 如果想将应用程序部署到单个目录下,并且不希望对注册表带来影响,那么可以对应用程序打包,以适合于Microsoft Installer。

在最简单的场景之外,部署选项变得复杂和有些混乱,因此问题本身是复杂的。如何允许开发者部署新版本的共享组件,而在同时确保新的版本不会对应用程序(它依赖于以前版本所提供的行为)产生影响?惟一的答案是开发一个系统,该系统支持同一组件的不同版本可以并行地进行部署。

需要了解在.NET Framework中进行部署的第一件事就是,将共享组件部署到C:\Windows\System32的日子已经远去了。取而代之的是,.NET Framework包含了两种程序集,定义良好的规则讲述了系统如何搜寻它们,所提供的基础结构支持在同一台机器上部署同一程序集的多个版本。

在我们谈论这两类程序集之前,先来解释这两类部署:

公共或全局的 公共部署的程序集是要在自己编写的应用程序或者在别人所编写的应用程序之间共享的部署单元。.NET Framework规定了一个众所周知的位置,在这个位置可以部署这样的程序集。

私用的 私用部署的程序集是那些不想共享的部署单元,它通常被部署到用户的应用程序所安装的目录中。

所创建的程序集类型要取决于如何部署它。程序集的类型如下所述:

强化名称程序集 强化名称(strong name)程序集是数字签名的。签名由程序集标识组成(名称、版本和可选的文化加上一对带有公共和私用组件的密钥对)。所有的全局程序集必须具有强化名称。名称、版本和文化是程序集的属性,存储在它的元数据中。如何创建和更新这些属性在很大程度上取决于部署工具。

其他内容 没有强化名称的程序集并没有一个正式的名称。没有强化名称的程序集只能被私用地部署,它无法利用CLR并行的版本特性。

密钥对完成了强化名称程序集的签名过程,它是由.NET Framework SDK提供的一个工具所产生的,这个工具叫做SN(sn.exe)。SN创建了一个密钥文件,它被属性中的程序集所引用。当程序集被创建或者建立时,程序集的签名将会发生。但是,在一种叫做延迟签名(delayed signing)的签名形式中,开发者只能使用密钥的公共部署进行工作。稍后,当准备的程序集最后被建立时,私钥将被添加到签名中。当前,Delphi for .NET编译器在自定义属性方面功能不是很强的。因为属性需要表达标识和引用密钥文件,在编写这本书的时候,对强化名称程序集的支持还不很完全。

强化名称程序集通常要进行共享，因此要在Global Assembly Cache (GAC, 全局程序集缓存) 中部署它们（但是它们也可以私用地进行部署）。因为GAC是带有特殊结构的系统文件夹，因此无法只是简单地将它们复制到那里。必须使用.NET Framework运行时中所包含的另一个工具，这个工具叫做GACUTIL (gacutil.exe)。GACUTIL可以把文件安装到GAC中，也可以把文件从GAC中删除。

GAC是一个特殊的系统文件夹，它带有一种层次结构，可以支持并行的程序集部署，如图24.4所示。其目的是隐藏**GAC**的结构，这样程序员就不需要担心所涉及的复杂程度了。**GACUTIL**从它的元数据中读取部件的标识，并使用那些信息在**GAC**中构建文件夹来保存程序集。

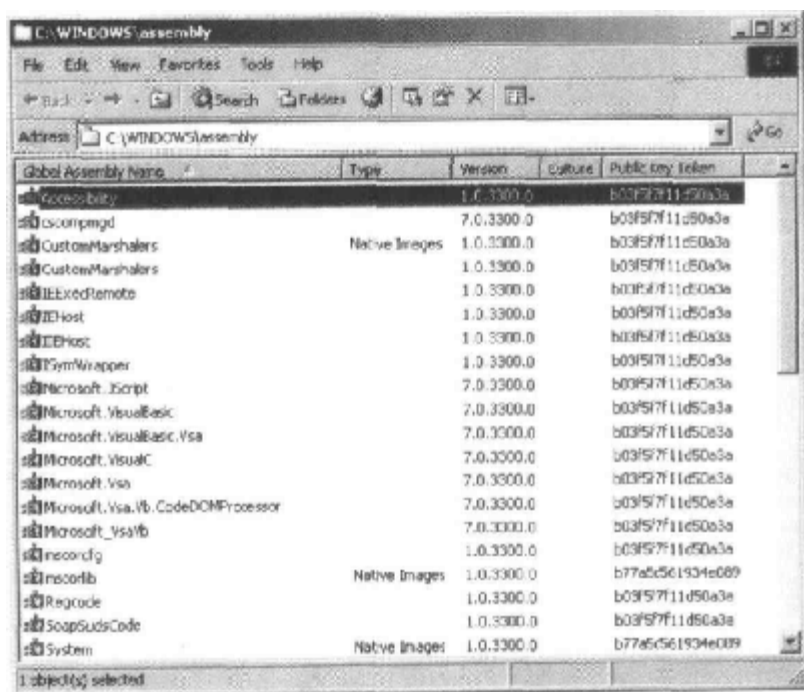


图24.4 在Windows Explorer中显示的.NET的全局部件缓存

说明：通过打开一个命令窗口并定位到C:\Windows\assembly目录就可以看到文件夹的层次。如果使用Windows Explorer查看它，会发现这个层次是被隐藏的。

当在GAC中创建对强化名称程序集的引用时，实际上要创建对该程序集特定版本的引用。如果在该机器上部署同一程序集的更新版本，它将被放置到GAC的一个单独位置中，而保留原始版本的完整性。因为程序集引用了特定版本，因此它总是得到那个版本。

小结

读者对.NET最初的反应要取决于各自的程序设计背景。如果读者以前熟悉Java环境,或者熟悉传统的Windows工具(即使是面向对象的),再或者是熟悉Delphi,则可能会发现很多新的特性或者很多概念你已经很熟悉了。.NET平台是一个技术成果,它提供了令人惊讶的选项和可能性。与早期的Microsoft系统接口技术(Windows API和COM)相比,这个进

步是巨大的。最好的一点就是读者可以使用自认为最趁手的工具和语言（其中包括Delphi）来使用这个平台。

在第25章中，我们将探讨特定于Delphi语言的更改。大家将看到遭到反对的特性和类型、已经被添加到语言中的新特性，以及从Delphi for .NET中使用.NET Framework类的演示示例。

第25章 Delphi for .NET Preview: 语言和RTL

在上一章中介绍了Microsoft的.NET体系结构, 现在将对与Delphi 7一同发行的Delphi for .NET Preview进行集中介绍。本章将介绍对Delphi语言所进行的具体修改, 通过修改使它能够与Common Language Runtime (公共语言运行时) 实现兼容, 还包括对Delphi语言所进行的许多重要添加, 例如名称空间和新的可视性说明符。由于一些曾长期使用的语言特性在安全类型环境中并不受支持, 因此不得不将其去除。

本章将对新型编译器领域进行讨论, 其中包括为了支持.NET和ASP对Microsoft本身的一些类库的使用。

记得在编写本书时, Delphi for .NET编译器仍在制作中。Borland计划在2002年的后期和2003年前期(在Galileo项目完工之前)与Delphi 7一同发行Preview编译器, 并对已注册的Delphi客户提供周期性的更新。这里讨论的一些语言特性可能尚未实现, 或者还处于不同的开发状态, 这要取决于各位所使用的编译器版本。在本章的结尾列举了一些Web资源, 读者可以利用这些资源来跟踪Borland的Delphi for .NET编译器的变化和最新消息。

说明: 和第24章一样, 本章的大部分内容都来源于John Bushakra。

本章主要包括以下内容:

- Delphi语言的变化
- 去除和新增的语言特性
- 运行时库和VCL
- Microsoft库的使用
- 使用Delphi语言实现ASP.NET

去除的Delphi语言特性

我们首先看一下为了与Common Language Runtime (CLR) 实现兼容而必须要去除的一些Delphi特性。然后看一下要在(或计划在) Delphi for .NET中添加的一些新特性, 在不久的将来, Delphi将主要由这些特性所构成, 而且也可能用于Win32和Linux平台。

去除的类型

有一些Delphi类型将不再转移到托管虚拟执行系统中。迄今为止, 下列类型或者已经被去除, 或者还不能确定是否要被去除:

指针 CLR认为指针是一种不安全类型, 因此禁止了所有形式的指针算法。不安全是指代码无法被证明为是安全类型的。最新版本的Delphi for .NET编译器也许会支持不安全的非托管指针, 但是也有可能使用动态数组来代替过去的GetMem、FreeMem和ReallocMem函数(也要被去除)的一些功能。

基于file of <type>的类型 编译器将不支持以过去Pascal中的file of <type>语法为基础

的文件类型，因为编译器无法确定目标平台上的某个指定类型的大小。

以前的Delphi object语法 以前的Delphi object语法已经被去除，而且最新发行的编译器将不再支持这个语法。这个语法是在Turbo Pascal时期提出的，它允许通过type MyClass = object声明新类。这种类型的变量以堆栈为基础，与基于堆(heap-based)的类类型对象正相反。

Real48 and Comp Delphi for .NET并不支持这些类型。Real48类型是一种6字节的浮点类型。将来将使用Int64来取代Comp类型，详细内容参见Delphi 7 Language Reference (Delphi 7语言参考)。

字符串和其他类型

尽管没有去除下面所列举的类型，但将会对它们的基础实现进行修改。这些变化几乎是透明的，但在某些情况下还是能够预料到它们的一些行为的微小变化：

字符串 在Delphi for .NET中，字符串对应于CLR的system.strings类型，而且默认情况下为wide类型，它是指每个字符占用16个比特，如同Delphi 7中的WideString类型。另外，所有的字符在默认情况下都是wide型的。

记录 记录被映射为数值类型。第24章探讨了由Common Type System (CTS) 所指定的两种主要的类型分类：引用类型和数值类型。在.NET平台上，记录将被设为数值类型。CLR要求数值类型不能具有继承性，但可以定义与它们有关的方法（当然，这对于Delphi语言来说是全新的）。关于数值类型方法的定义必须要以final形式进行声明（在“Delphi语言新特性”一节中将讨论这个新增的关键字final）。

TDateTime 在Delphi中，该类型与Microsoft的DATE类型一样都基于相同的实现方法（详见第2章“Delphi编程语言”）。.NET平台使用了不同的实现方法。System.DateTime结构（System.ValueType的派生）从公元0001年1月1日午夜开始计时，一直到公元9999年12月31日11:59:59 P.M.。对于这种时钟，一个滴答等于100个十亿分之一秒。接着Delphi for .NET将其转换为.NET平台的计时标准。当移植到.NET平台上时，要留心依赖浮点运算的日期计算。特别是，如果正在使用Delphi的Trunc和Frac函数区分日期和时间部分的浮点数数值的话，那么将来就可能会遇到一些有趣的漏洞。

货币 货币将被映射到CLR类型System.Decimal。

去除的代码特性

如同前面一节所列举的类型情况一样，一些长期以来一直是组成Delphi语言的部分特性不能被移植到.NET平台。

可变记录 CLR不支持覆盖字段的可变记录。通常，不能对记录声明中有关字段的安排进行设定，这是由于Just In-Time (JIT) 编译器才具有为适应基础平台而进行优化的权利。

ExitProc 事情的发生并不总是像大家想像的那样让它何时发生即何时发生，让它如何进行就如何进行。尽管诸如单元的初始化及其收尾工作问题已经得到了解决（尽管还存在一些需要注意的问题，关于这些内容将在本章的稍后部分进行讨论），但这里并不支持ExitProc。

接口的动态聚合 CLR不支持利用关键字implements实现接口的动态聚合, 因为无法验证它的安全性。类必须要声明它所需要运行的所有接口。

ASM语句 Delphi for .NET Preview编译器不支持ASM语句及插入的装配语言。而且还不能肯定将来最新的版本是否会使用asm语句, 那么编译器将必须具有把受控的IL (Intermediate Language, 中间语言) 与非托管的内部CPU指令混合使用的能力, 如同Microsoft的Visual C++编译器一样。

automated关键字 关键字automated的创建是为了支持OLE的Automation方法, 而在.NET环境中不再需要使用它。同样也不再需要根据编号 (而不是按照名称) 调度COM的Automation方法的关键字dispid。注意, 尽管不再明显地需要它们, 但.NET平台还是支持GUID, 它们以自定义的类型属性形式表示。

直接内存存取 (DMA) 函数 直接内存存取函数, 比如BlockRead、BlockWrite、GetMem、FreeMem、ReallocMem以及Absolute和Addr, 都可以处理非托管指针, 因此它们不能与托管安全代码共同使用。尽管不能输入cast指针类型或者不能进行任何指针算法操作, 但@运算符仍可以在编译器的当前Preview版本中存在 (但估计不会在最新版本中保留它)。

说明: 正如在第2章中讨论的一样, Delphi 7提供了新的一组编译器警告来帮助用户准备移植代码。

这些警告标出了在.NET平台上已知的不安全某些特性和语言结构, 因此要避免使用。默认情况下, 对于新的Delphi 7项目而言, 这些警告是关闭的, 但当对一个现存项目进行重新编译时, 就会激活这些警告。也可以通过{\$WARN UNSAFE_CODE ON}编译器命令或类似的命令启用它们, 详细内容见第2章。

Delphi语言的新增特性

首次发行的dccil编译器包括了CLR所需要的新特性, 而且在随后的更新中添加了更多新的特性。

单元名称空间

单元名称空间在.NET Framework中起了非常重要的作用。它们允许多个第三方对类的分层结构进行扩展而不必担心符号名称发生冲突。Windows和COM利用16字节的GUID来惟一标识组件, 而且该编号必须要被保存在系统注册表中。在.NET平台上, 由于名称空间概念的存在 (再加上有关程序集查询的元数据和严格规则的存在), 导致了放弃使用GUID。

具有讽刺意义的是, Delphi的单元理念与CLR的名称空间相似。如果将单元看做是符号的容器, 将名称空间看做是单元的容器, 那么它并不是多么大的飞跃。在Delphi for .NET中, 单元所属的名称空间在unit子句中进行声明:

```
unit NamespaceA.NamespaceB.UnitA;
```

语句中的点号表示某个名称空间包含在另一个名称空间中, 以及最后在名称空间中的单元。点号将声明划分为多个组件, 每个组件 (一直到右边但不包括最右边的一个) 都是名称空间。整个声明作为一个整体, 包括所有的点号, 是一个单元名。其中的点号只是充当分隔符, 声明中并没有引入任何新的符号。在上面的例子中NamespaceA.NamespaceB是名称空

间, `NamespaceA.NamespaceB.UnitA`是单元名称。`NamespaceA.NamespaceB.UnitA.pas`是源文件名称, 而且编译器产生的输出文件叫做`NamespaceA.NamespaceB.UnitA.dcuil`。

`program`语句(以及最后的`package`和`library`语句)可以为全部项目声明可选的默认名称空间。否则将该项目称为普通项目, 而且默认名称空间是由编译器选项`-ns`所指定的。如果没有使用编译器选项指定默认项目名称空间, 那么就会返回到不使用名称空间的状态, 如同Delphi 7一样(以及以前发行的版本)。

`unit`子句不一定要在任何显式名称空间中声明成员关系, 其格式与传统的Delphi语句一样:

```
unit UnitA;
```

在名称空间中不声明成员关系的单元称为普通单元。普通单元可以自动成为项目名称空间的成员。但是注意这并不会影响源文件名称。

警告: 在编辑本书的时候, 对名称空间的支持是非常有限的。这里说明的是它将来的工作原理, 而非是它现在的工作原理。

在项目文件中, 当编译器在尽量对普通单元的引用进行解析时, 可以指定`namespaces`子句, 为其列举一组名称空间以便于查找。`namespaces`子句必须紧跟在`program`(或`package`或`library`)语句之后, 其他任何子句或块类型之前。名称空间由逗号分隔, 而且该列表由分号结束。例如:

```
program NamespaceA.MyProgram
namespaces Foo.Bar, Foo.Frob, Foo.Nitz;
```

上面的例子向普通的单元查找空间中加入了名称空间`Foo.Bar`、`Foo.Frob`和`Foo.Nitz`。

这个讨论把话题渐渐引向了当构建程序的时候, 显示编译器是如何查找普通单元的。当使用一个单元, 而且将其名称完全限定为使用完整的名称空间声明时, 将不存在任何问题:

```
uses Foo.Frob.Gizmos;
```

在这种情况下, 编译器可以识别`dcuil`文件(或`.pas`文件)的名称。但假设只是用下面的语句形式给出:

```
uses Gizmos;
```

这被称为普通单元引用, 而且编译器一定要具有能够找到其`dcuil`文件的方法。

编译器按照下列顺序对名称空间进行查找:

1. 当前单元名称空间(如果存在的话)
2. 默认的项目名称空间(如果存在的话)
3. 在项目`namespaces`子句中列举的名称空间(如果存在的话)
4. 编译器选项指定的名称空间

对于第一点, 如果当前单元指定了一个名称空间, 那么将首先在当前单元名称空间中查找当前单元的`uses`子句中的普通单元引用。参见下面的例子:

```
unit Foo.Frob.Gizmos;
uses doodads;
```

其中, 单元doodads的第一个搜索位置在名称空间Foo.Frob中。因此编译器将尽量打开Foo.Frob.Doodads.dcuil。如果查找失败, 编译器将继续查找并将默认项目名称空间作为单元名称doodads的前缀, 然后继续沿着列表进行查找。

这个相同的符号可以出现在不同的名称空间中。当发生类似这样的二义性时, 必须要使用它的全部名称空间和单元名称。如果在单元Foo.Frob.Gizmos中存在一个名为Hoozitz的符号, 那么可以使用下面的两种方式中的任何一种来引用该符号:

```
Hoozitz; // if the name is unambiguous
Foo.Frob.Gizmos.Hoozitz;
```

但是不要使用:

```
Gizmos.Hoozitz; // error!
Frob.Gizmos.Hoozitz; // error!
```

单元和名称空间的名称都可能会变得相当长而且不便使用。可以在uses子句中利用关键字as为完全限定的名称创建一个别名:

```
uses Foo.Frob.DepartmentOfRedundancyDepartment.UIToys as ToyUnit;
```

单元别名引入了新的标识符, 因此它们的名称不要与同一个单元中的(别名对于它们的单元来说是本地的)其他标识符发生冲突。即使声明了一个别名, 仍可以使用原来长度较长的名称来引用这个单元。

说明: 在程序集元数据中保留了名称空间声明中字母大小写的区别, 但是对于Delphi, 大小写不同的两个名称空间实际上是相同的。

扩展的标识符

CTS和CLR语言之间的跨语言集成为编译器开发人员带来了一些有趣的情况。例如, 如果程序集中的某个标识符名称与使用语言中的某个关键字相同该怎么办? 例如Delphi语言的关键字type.Type也是CLR的一个类名。由于type是语言的一个关键字, 所以它不能作为标识符名称。可以在Delphi for .NET中通过两种方法来避免这个问题(这些技术并没有在Delphi 7以及以往的版本中实现)。

第一种方法是使用标识符的完全限定名称:

```
var
  T: System.Type;
```

第二种方法更简洁, 用“与”运算符(&)作为标识符的前缀。下面的例子与前面的例子产生相同的效果:

```
var
  T: &Type;
```

在这个语句中, “与”运算符告诉编译器查找带有Type名称的符号, 而且不要将它看做关键字。编译器将在有效单元中查找Type符号, 并可在System中找到它(不管单元是否定义了这个符号, 运行的是相同的机制)。

关键字final和sealed

Common Language Infrastructure (CLI) 所指定的两个概念已经被添加到Delphi for .NET编译器中：类属性sealed和方法属性final。将sealed属性放在类中可以有效地终止将该类用做基类的能力。下面是一个代码片段举例：

```
Type
TDeriv1 = class (TBase)
    procedure A; override;
end sealed;
```

不能通过一个已被密封的类进行派生。同样，在派生类中不能对标有final属性的方法进行覆盖，如下面的代码举例所示：

```
Type
TDeriv1 = class (TBase)
    procedure A; override; final;
end;

TDeriv2 = class (TDeriv1)
    procedure A; override; // error: "cannot override a final method"
end;
```

Borland增加了关键字sealed和final来映射.NET的一个现存特性，但为什么Microsoft要引入这两个属性呢？final和sealed属性可以向使用代码的用户提供重要的说明，以便他们能够了解到将如何使用这些类。况且这些属性为编译器提供了使之能够生成更加有效的Common Intermediate Language (CIL) 的提示。

新的可视性及访问说明符

Delphi可视性概念（公有、保护和私有）与CLI的可视性概念稍有不同。在诸如C++和Java这样的语言中，当将类成员指定为私有的或受保护的可视性时，只能在定义它的类的派生类中才能看到该类成员。但是，正如第2章所述，Delphi强调私有和保护概念只能应用于不同单元中的类，因为在一个单元内部的所有东西都是可见的。如果要遵循CTS规定，则该语言需要新的可视性说明符：

类私有 被声明为类的私有可视性成员符合C++和Java规则。即，只能在声明类的方法或属性中访问类的私有成员。在单元级中声明的过程和函数以及其他类的方法不具有访问权。

类保护 同样，只能在声明类及其派生类中可以看到类的受保护成员。在相同单元中的其他类，只有当它们继承了这个类，才能够对其进行访问。

读者可以参见本章测试案例源代码的LanguageTest文件夹下的ProtectedPrivate例子。

类静态成员

Delphi长期以来一直支持类方法——这些方法既可以以一个整体也可以以一个指定的实例应用到类中，尽管方法代码不能引用当前对象（类方法的Self参数引用的是当前类，而不

是当前对象)。Delphi for .NET通过增加class static (类静态)说明符、类属性、类静态字段和类构造器扩展了这种思想:

类静态方法 与Delphi 7类方法相同,可以不通过对象实例调用类静态成员,而且不存在引用对象的Self参数。但是与Delphi 7不同的地方是不能引用类自身。例如,调用ClassName的方法就会失败。另一个与Delphi 7不同的是不能使用类静态方法关键字virtual。

类静态属性 与类方法相同,可以不通过对象实例访问类静态属性。类静态属性的访问方法或备用字段必须被声明为类静态。类静态属性既不能发布,也不能存储成定义为默认数值。

类静态字段 可以不通过对象实例访问类静态字段。类静态字段和属性通常专门用做设计工具;它们允许在有意义的类声明上下文中声明变量和常量。

类构造器 类构造器是一种私有构造器(必须要用类私有可视性对它进行声明),在第一次使用声明类之前运行。除了能够说这种情况可以在首次使用类之前发生外,CLR并不保证它何时发生。在CLR术语中,这可能变得有点复杂,因为除非(以及直到)代码运行,才可以认为它被“使用”了。一个类只能声明一个类构造器。派生类可以声明它们自己的类构造器,但在所有的类中也只能声明一个类构造器。

不能从源代码中调用类构造器,它是在初始化类静态字段和属性时对其进行的自动调用。甚至连关键字inherited也要被禁止,因为是由编译器对此进行管理的。

下面的类声明说明了这些新说明符的语法:

```
MyClass = class
  class private // can only be accessed within MyClass
    // Class constructor must have class private visibility
    class constructor Create;
  class protected // can be accessed in MyClass and in descendants
    // Class static accessors for class static property P1, below
    class static function getP1 : Integer;
    class static procedure setP1(val : Integer);
  public
    // fx can be called without an object instance
    class static function fx(p : Integer) : Integer;
    // Class static property P1 must have class static accessors
    class static property P1 : Integer read getP1 write setP1;
end;
```

嵌套类型

嵌套类型类似于类字段,因为可以通过类引用对它们进行访问,而且并不需要对象实例。通过在类作用域中声明嵌套类型,提供了以名称空间类型方式使用封装类的方法。

多点传输事件

Delphi通常具有设置事件收听者的能力——即当某个事件被激发时所调用的某个函数。CLR可以支持使用多个事件收听者,因此在某个事件被激发时,会有多个函数发出响应。以

上这些被称做多点传输事件。Delphi for .NET引入了两种新的访问方法特性，`add`和`remove`，用来支持多点传输事件。`add`和`remove`方法只能被用于事件属性。

若要支持多点传输事件，必须要具有能够对所有将其自身注册为收听者的函数进行存储的方法。如同第24章所述，实现多点传输事件是通过使用CLR的`MulticastDelegate`类来完成的。而且，就如那里所讨论的一样，编译器在后台隐藏了许多的复杂性。通过关键字`add`和`remove`可以处理事件收听者的存储和删除问题，但是如何对容器机制的执行细节进行处理是我们根本无法想像的。编译器可以自动地生成`add`和`remove`方法，而且这两种方法可以有效地实现事件收听者的存储工作。

在最新发行的Delphi for .NET中，`add`和`remove`方法应该与标准函数`Include`和`Exclude`联合使用。在源代码中，当想将某个方法注册为某个事件의收听者时，可以调用`Include`函数；要删除某个方法，则可以调用`Exclude`函数。例如：

```
Include(EventProp, eventHandler);  
Exclude(EventProp, eventHandler);
```

在后台，函数`Include`和`Exclude`将分别调用指派给`add`和`remove`访问函数的方法。编写这部分的时候，这种技术还没有付诸实行，因此在本书中的举例没有使用它。

为支持传统代码，Delphi的赋值运算符（`:=`）仍是一种用来为某个单一事件处理器赋值的方法。编译器会生成代码，返回并替换由赋值运算符设置的最后一个事件处理器（而且只能是这个事件的处理器）。而赋值运算符与`add/remove`（或`Include/Exclude`）机制分开单独运行。换句话说，赋值运算符的使用并不会对已经添加到`MulticastDelegate`中的事件处理器列表产生任何影响。

具体的例子可以参照`XmlDemo`程序。下面的代码段（在编写这部分时已经是可执行的代码）可在运行时创建一个按钮，并为其`Click`事件安装两个事件处理器：

```
MyButton := Button.Create;  
MyButton.Location := Point.Create (  
    Width div 2 - MyButton.Width div 2, 2);  
MyButton.Text := 'Load';  
MyButton.add_Click (OnButtonClick);  
MyButton.add_click (OnButtonClick2);  
Controls.Add (MyButton);
```

自定义属性

回忆在第24章中所讲到的，CLI的一个要求是一种可扩充的元数据系统。为了在程序集内部定义的类型中加入元数据，需要所有类型的.NET语言编译器。可扩充元数据中的可扩充性部分是指程序员可以定义他们自己的属性，并可以将它们应用到任何一种实体：程序集、类、方法中。而编译器将把这些属性加入到程序集的元数据中。在运行时，可以利用CLR中的类`System.Type`方法查询应用在某个实体（程序集、类、方法等等）中的属性。

自定义属性是从CLR的类`System.Attribute`中派生的引用类型。对自定义属性类的声明就如同对其他类的声明方法一样（这个代码段是从`LanguageTest`目录下的`NetAttribute`项目部分提取出的）：

```

type
  TMyCustomAttribute = class(TCustomAttribute)
  private
    FAttr : Integer;
  public
    constructor Create(val: Integer);
    property customAttribute : Integer read FAttr write FAttr;
  end;
  ...
  constructor TMyCustomAttribute.Create(val: Integer)
  begin
    inherited Create;
    customAttribute := val;
  end;

```

应用自定义属性的语法与应用C#的属性语法相同:

```

type
  [TMyCustomAttribute(17)]
  TFoo = class
  public
    function Pl(X : Integer) : Integer;
  end;

```

自定义属性被应用到紧跟在其后的构造中。在本例子中, 它被应用在类TFoo中。可以很明显地注意到自定义属性语法几乎与Delphi的GUID语法相同。但这里存在一个问题: GUID是应用在接口中的, 它们必须要紧跟在接口声明之后。而另一方面, 自定义属性必须恰好在它们要应用的声明之前。那么编译器将如何确定在方括号中的属性是传统Delphi类型的GUID (它应该被应用在前面定义的接口声明中) 还是.NET类型的自定义属性 (它应该被应用到接口的第一个成员中) 呢?

这根本就无法区分, 因此必须要由自己来掌握——为自定义属性和接口指定一种特殊字符。如果在接口中使用GUID, 它必须要紧跟在接口声明之后, 并遵守所建立的Delphi语法:

```

type
  interface IMyInterface
    ['(12345678-1234-1234-1234-1234567890ab)']

```

CLR的GuidAttribute自定义属性是用来应用GUID的, 它隶属于System.Runtime.InteropServices名称空间中的一部分。如果使用这个自定义属性来应用一个GUID, 那么就要必须遵守CLR标准, 而且要在接口声明之前进行属性声明。

辅助类

辅助类是Delphi for .NET中一种非常有意思的新增语言特性。支持辅助类的主要原因是它是一种Borland通过自身的RTL类映射.NET核心类的方法, 详细内容参见“RTL辅助类”。这里将从语言方面集中讨论这个特性。

辅助类向用户提供了一种可以在不使用派生的情况下通过添加新方法（但不是新数据）来扩展类的方法。其奇特的方面是，与继承相比，可以创建原始类对象，而且可以扩展使用相同的名称，也就是可以向一个现存类的现存对象中插入方法。用一个简单的例子可以帮助大家理解这种概念。

假设我们具有如下一个类（也许读者还没有进行编写——要不早就对其进行了扩展）：

```
type
  TMyObject = class
  private
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;
```

现在可以通过编写一个辅助类为该对象添加一个Show方法，以对其进行扩展：

```
type
  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;

procedure TMyObjectHelper.Show;
begin
  WriteLn (Text + ' ' + IntToStr (Value) + ' -- ' +
    Self.ClassType.ClassName + ' -- ' + ToString);
end;
```

注意在辅助类方法中的Self是辅助类所辅助的类对象，其使用方法如下所示：

```
Obj := TMyObject.Create;
...
Obj.Show;
```

最后可以在输出中看到TMyObject类名。但是，如果继承了该类，那么也可以在派生类中使用辅助类（因此最终可以在整个分层中添加一个方法），而且所有工作都可以正常运行。如要进行试验，可以参照LanguageTest文件夹中的ClassHelperDemo范例。

运行时库及VCL

在安装Delphi for .NET Preview的source\rtl目录中，可以找到运行时库（run-time library, RTL）源文件。读者可能已经看出单元向CLR名称空间的转移，就如源文件名称所表现出的那样。

Borland采用了保留原单元名称以及将名称空间名称Borland.Delphi作为其前缀的方法（在很大程度上是这样的）。在Borland.Win32的名称空间中加入了指定的操作系统事物（如注册表和ini文件实用程序），因为这些类、过程和函数是Windows专用特性的专用于Borland的

封装。尽管不是所有的单元都要进行这种转换，而且其中的一些还要将它们的内容识别为适当的名称空间，但这种命名方式还应该继续实行。

细细研究一下RTL源文件，就会发现它既具有很高深的知识，而且也是我们所强烈推荐；但是，记住我们所看到的只是该产品发行的预演，并不是最后的版本。仍然还需要对RTL源文件内容进行修改——不应该进行任何假设，而且也千万不要根据在这里所看到的来向自己的代码中引入相关内容。

RTL辅助类

撇开这些警告，让我们看看迄今为止对RTL已经做了什么修改。最有趣的更改可能就是辅助类的引入。

Borland.Delphi.System.pas中包括了如下声明：

```
type
  TObject = System.Object;
```

它告诉用户Delphi的TObject类是CLR类System.Object的别名。这非常重要：TObject并不是System.Object的派生——它只在语法上是等价的。那么通常在TObject中定义的方法会发生什么情况，例如ClassName和ClassParent？这里就是辅助类的用武之地了。

过去通常在TObject中直接声明和执行的方法现在要在称为TObjectHelper的类中声明和执行。因此TObjectHelper被声明为TObject的辅助类。在Borland.Delphi.System.pas中其形式如下：

```
type
  TObjectHelper = class helper for TObject
  procedure Free;
  function ClassType: TClass;
  class function ClassName: string;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: TObject;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress(const Name: string): TObject;
  class function SystemType: System.Type;
  function FieldAddress(const Name: string): TObject;
  procedure Dispatch(var Message);
end;
```

辅助类提供了不用通过派生就可扩展类的方法。

为了在现存的Delphi代码中使用CLR类，大家也许想对某个CLR类进行扩展，而不是通过它派生得到，而且也肯定已经注意到Delphi的类框架和.NET Framework共享了相当数量的功能。在某些情况下，在两个名称之间可能存在着名称冲突——例如，Borland的Exception类和CLR的System.Exception类。一方面，这两个类基本上可完成相同的功能，但它们是以不同的方式实现这个功能的。另一方面，现存的大量Delphi代码一直都在长时间地利用Borland的Exception类。

惟一可行的解决方法是建立一种机制，以允许开发人员（包括Borland的开发人员）均衡CLR类，从而允许CLR类扩展为包含现存Delphi代码所希望的永久性行为。

VCL

在System.Windows.Forms名称空间中的.NET Framework类并不是Win32 API的GUI部分的替代品，同样也不是.NET Framework中其他绝大部分的替代：它的作用是通过提供一种与.NET核心环境服务兼容的面向对象的简单接口，使基础API的使用更加容易。Win32的GUI子集仍然存在，而且占据着它通常所在的位置。System.Windows.Forms会对Win32 API的这种GUI子集进行组织，并以面向对象的方式进行表示，而且在其上层建立一个事件模型。但System.Windows.Forms中的类是以Win32形式调用非托管代码的。也就是说，在使用System.Windows.Forms时，仍就是调用了Win32的API，但目前在我们的代码和Win32之间存在一个称为CLR的大型软件层。

了解上述关于为什么VCL采用了这个相同方法的开场白是非常重要的。通过使用辅助类，TObject可以将System.Object作为根（如果愿意）。TPersistent和TComponent仍是从这里派生得到的。因此举例来说，VCL类TForm并不是类System.Windows.Forms.Form的派生类。相反，整个VCL的分层结构仍保持目前状况。归根结底TForm是从TWinControl中派生得到的，TWinControl本身也是TControl的一个派生类，然后派生TComponent。

如果把整个System.Windows.Forms名称空间看做一个单一的实体，那么VCL就变成了它的兄弟而不是它的孩子。为了从根本上实现对用户接口的控制，这两种框架归根结底都要依赖于本机非托管Win32 API。

剖析VCL.NET源代码

Borland于2002年11月份对Delphi .NET Preview编译器所进行的更新尽管仍处于初步阶段，但提供了有关公司计划的体系结构的详细内容。如果打开Borland.Vcl.Controls单元，大家肯定会对其与Win32版本的相似性感到吃惊。源程序代码几乎是相同的，其不同方面位于TObject和TComponent级。我在前面已经讨论过了这个构造方法，因此现在让我们集中讨论核心组件类，其定义方法分三步进行：

type

```
TComponent = System.ComponentModel.Component;  
TComponentHelper = class helper (TPersistentHelper) for TComponent  
TComponentSite = class (TObject, ISite, IServiceProvider)
```

TComponent类与.NET的Framework类相一致，带有辅助类（提供附加方法和属性）和更深一层的类（提供辅助类需要的附加数据）。这种情况比较复杂，我不想对其进行更加详细的讨论，因为TComponent类还处在试验阶段，并可能在进一步的更新中对其进行修改。

让我们再返回讨论VCL，目前已经存在了一组数量较多的组件，因此可以开始移植代码了。对于2002年11月份所进行的更新，所面临的惟一问题是不支持流式操作，因此必须在窗体构造器中添加创建组件的代码（Delphi for .NET的最新版本将不再需要这个动作）。

下面我们用一个有趣的例子来帮助读者理解VCL类的体系结构, 笔者已经从第3章“运行时库”将其移植到.NET的ClassInfo例子中。NetClassInfo例子在项目源程序中使用了这种修改过的代码(再提醒一次, 将来可能不用进行其中的某些操作):

```
Application.Initialize;
Form1 := TForm1.Create (Application);
Application.MainForm := Form1;
```

正如前面已经提到的, 窗体代码具有一种附加方法, 该方法可由构造器调用, 并用于对控制进行初始化。这种方法的程序段相当长, 因此在这里只提供其中的一些摘要:

```
procedure TForm1.InitializeControls;
begin
    // creating all controls...
    Label3:= TLabel.Create(Self);
    Panel1:= TPanel.Create(Self);
    Label1:= TLabel.Create(Self);
    Label2:= TLabel.Create(Self);
    ...

    // setting form properties and events
    Left:= 217;
    Top:= 109;
    Caption:= 'Class Info';
    OnCreate:= FormCreate;

    // initializing controls (only one is listed here)
    with Label3 do
    begin
        Parent:= Self;
        Left:= 8;
        Top:= 8;
        Width:= 56;
        Height:= 13;
        Caption:= 'Class Name';
    end;
```

应用程序代码的剩余部分几乎都是相同的, 令人惊讶的是这是一个低级的例子。我不得不将调用移植到InstanceSize中, 因为编译器不能解析出.NET体系结构的对象大小, 因而我不得不依据Object(取代TObject)对基类进行测试。下面给出可生成如图25.1所示输出的代码段:

```
procedure TForm1.ListClassesClick(Sender: TObject);
var
    MyClass: TClass;
begin
    MyClass := ClassArray [ListClasses.ItemIndex];
    EditInfo.Text := Format ('Name: %s - Size: %d bytes',
        [MyClass.ClassName, 0 {MyClass.InstanceSize}]);
```

```
with ListParent.Items do
begin
  Clear;
  while MyClass.ClassName <> 'Object' do
  begin
    MyClass := MyClass.ClassParent;
    Add (MyClass.ClassName);
  end;
end;
end;
```

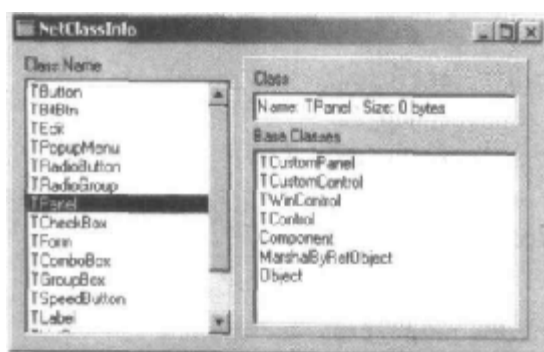


图25.1 NetClassInfo表示给定组件的基类

更深一层的VCL举例

为了向读者提供在.NET条件下对VCL进行试验的起点，这里举两个例子。NetEuroConv是对第3章中基于RTL会话引擎的EuroConv例子的移植。NetLibSpeed是第5章中用于比较VCL和VisualCLX库创建可视组件速度的LibSpeed例子的移植。读者将会看到，尽管是为了相同的目的，VCL.NET将会花费4倍到5倍的时间，这也许会使你感到苦恼，但这个时间量在这种版本的库的初步制作阶段还是很有用的。

正如前面所提到的，举这些例子只是为了给试验提供起点。它们也许无法在进一步更新后的Delphi for .NET Preview中运行。

说明：请与我的站点保持联系，以便对本书这一节及相关举例进行更新。

Microsoft库的使用

虽然VCL还不是相当完备的，但是可以将.NET Framework类库作为对Delphi for .NET Preview编译器进行试验的基础。例如，利用编译器建立程序需要一定的知识，然后可以利用Intermediate Language Disassembler (ILDASM) 来对其进行审查。这就是这一节的目标。如果读者想看一个利用XML支持的更加简单的例子，可以参见本章先前提到的例子XmlDemo。

CLRReflection程序打开了一个程序集，然后使用映像法来检查该部件内部定义的模块和类型。这个程序利用了一个公共对话框 (OpenFileDialog)、构造菜单、事件处理、Delphi的动态数组，当然还有映像方法来举例说明。让我们先看一下这个项目的文件：


```

program CLRReflection;
uses
    System.Windows.Forms,
    ReflectionUnit;
var
    reflectForm : ReflectionForm;
begin
    reflectForm := ReflectionForm.Create;
    System.Windows.Forms.Application.Run(reflectForm);
end.

```

该代码看上去很像一个正确的老式VCL应用程序。可为主窗体定义一个变量，然后创建该窗体。之后，可以使用.NET Framework类System.Windows.Forms.Application中的Run方法。这里的代码模拟了它在VCL中的运行方式（至少从概念上）。

注意在这个例子的全部过程中，为.NET Framework类指定了限定的全名称。这样做是为了确保能够知道这些类所处的位置。因为uses子句中包含了System.Windows.Forms，因此可以将其表达式

```
System.Windows.Forms.Application.Run(reflectForm);
```

缩短为

```
Application.Run(reflectForm);
```

现在，看一下程序清单25.1，它显示了定义主窗体位置的单元。注意这个代码是利用2002年11月份更新的Delphi for .NET Preview进行编译的，而不是利用原先Delphi 7所发行的版本。

程序清单25.1 CLRReflection的ReflectionUnit单元举例

```

unit ReflectionUnit;

interface

uses
    System.Windows.Forms,
    System.Reflection,
    System.Drawing,
    Borland.Delphi.SysUtils;

type
    ReflectionForm = class (System.Windows.Forms.Form)
    private
        mainMenu: System.Windows.Forms.MainMenu;
        fileMenu: System.Windows.Forms.MenuItem;
        separatorItem: System.Windows.Forms.MenuItem;
        openItem: System.Windows.Forms.MenuItem;
        exitItem: System.Windows.Forms.MenuItem;

        showFileLabel: System.Windows.Forms.Label;
        typesListBox: System.Windows.Forms.ListBox;

```

```

    openFileDialog: System.Windows.Forms.OpenFileDialog;
protected
    procedure InitializeMenu;
    procedure InitializeControls;
    procedure PopulateTypes(fileName: String);
    { Event Handlers }
    procedure exitItemClick(sender: TObject; Args: System.EventArgs);
    procedure openItemClick(sender: TObject; Args: System.EventArgs);
public
    constructor Create;
end;

implementation

constructor ReflectionForm.Create;
begin
    inherited Create;

    SuspendLayout;
    InitializeMenu;
    InitializeControls;

    { Initialize the form and other member variables }
    openFileDialog := System.Windows.Forms.OpenFileDialog.Create;
    openFileDialog.Filter := 'Assemblies (*.dll;*.exe)|*.dll;*.exe';
    openFileDialog.Title := 'Open an assembly';

    AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
    ClientSize := System.Drawing.Size.Create(631, 357);
    Menu := mainMenu;
    Name := 'reflectionForm';
    Text := 'Reflection in Delphi for .NET';

    { Add the controls to the form's collection. }

    Controls.Add(showFileLabel);
    Controls.Add(typesListBox);
    ResumeLayout;
end;

{ Build the main menu }
procedure ReflectionForm.InitializeMenu;
var
    menuItemArray : array of System.Windows.Forms.MenuItem;
begin
    mainMenu := System.Windows.Forms.MainMenu.Create;
    fileMenu := System.Windows.Forms.MenuItem.Create;
    openItem := System.Windows.Forms.MenuItem.Create;
    separatorItem := System.Windows.Forms.MenuItem.Create;
    exitItem := System.Windows.Forms.MenuItem.Create;

```

```
{ Initialize mainMenu }
mainMenu.MenuItems.Add(fileMenu);

{ Initialize fileMenu }
fileMenu.Index := 0;
SetLength(menuItemArray, 3);
menuItemArray[0] := openItem;
menuItemArray[1] := separatorItem;
menuItemArray[2] := exitItem;
fileMenu.MenuItems.AddRange(menuItemArray);
fileMenu.Text := '&File';

// openItem
openItem.Index := 0;
openItem.Text := '&Open...';
openItem.add_Click(openItemClick);

// separatorItem
separatorItem.Index := 1;
separatorItem.Text := '-';

// exitItem
exitItem.Index := 2;
exitItem.Text := 'E&xit';
exitItem.add_Click(exitItemClick);
end;

{ Create the controls and populate the form }
procedure ReflectionForm.InitializeControls;
begin
  { Initialize showFileLabel }
  showFileLabel := System.Windows.Forms.Label.Create;
  showFileLabel.Location := System.Drawing.Point.Create(5, 6);
  showFileLabel.Name := 'showFileLabel';
  showFileLabel.Size := System.Drawing.Size.Create(616, 37);
  showFileLabel.TabIndex := 0;
  showFileLabel.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right
  showFileLabel.Text := 'Showing types in: ';

  { Initialize typesListBox }
  typesListBox := System.Windows.Forms.ListBox.Create;
  typesListBox.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Bottom or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right;
  typesListBox.Location := System.Drawing.Point.Create(8, 46);
  typesListBox.Name := 'typesListBox';
```

```

typesListBox.Size := System.Drawing.Size.Create(610, 303);
typesListBox.Font := System.Drawing.Font.Create('Lucida Console', 8.25,
    System.Drawing.FontStyle.Regular, System.Drawing.GraphicsUnit.Point, 0);
typesListBox.TabIndex := 1;
end;

{ Event handler for the Exit menu item }
procedure ReflectionForm.exitItemClick(sender: TObject; Args: System.EventArgs);
begin
    System.Windows.Forms.Application.Exit;
end;

{ Event handler for the Open menu item }
procedure ReflectionForm.openItemClick(sender: TObject; Args: System.EventArgs);
begin
    if openFileDialog.ShowDialog = DialogResult.OK then
    begin
        showFileLabel.Text := Showing types in: ' + openFileDialog.FileName;
        PopulateTypes(openFileDialog.FileName);
    end;
end;

{ Open the given assembly, and reflect over its modules }
{ and types. }
procedure ReflectionForm.PopulateTypes(fileName : String);
var
    assy: System.Reflection.Assembly;
    modules: array of System.Reflection.Module;
    module: System.Reflection.Module;
    types: array of System.Type;
    t: System.Type;
    members: array of System.Reflection.MemberInfo;
    m: System.Reflection.MemberInfo;
    i,j,k: Integer;
    s: String;
begin
    try
        { Clear the listbox }
        typesListBox.BeginUpdate;
        typesListBox.Items.Clear;

        { Load the assembly and get its modules }
        assy := System.Reflection.Assembly.LoadFrom(fileName);
        modules := assy.GetModules;

        { For every module, get all types }
        for i := 0 to High(modules) do
        begin
            module := modules[i];

```

```
types := module.GetTypes;
{ For every type, get all of its members }
for j := 0 to High(types) do
begin
    t := types[j];
    members := t.GetMembers;

    { for every member, get type information and add to list box }
    for k := 0 to High(members) do
    begin
        m := members[k];
        s := module.Name + ':' + t.Name + ':' + m.Name +
            ' (' + m.MemberType.ToString + ')';
        typesListBox.Items.Add(s);
    end;
end;
end;
typesListBox.EndUpdate;
except
    System.Windows.Forms.MessageBox.Show('Could not load the assembly.');
```

```
end;
end;
end.
```

该单元开始就声明了它对.NET Framework dcuil文件以及Borland.Delphi.SysUtils单元的依赖性。并且更进一步地声明了主窗体类，它是.NET Framework类System.Windows.Forms.Form的派生类。窗体类的格式很常见：拥有所有控件的成员变量，而且这些变量被声明为.NET Framework类库中的类型。

函数exitItemClick和openItemClick是事件处理器声明。事件处理器方法的签名由CLR所指定。所有事件处理器都是一个过程，而且这些过程都采用两个参数：激发事件的对象（System.Object的派生）和事件参量，它们均被封装在System.EventArgs（或是某个派生）类中（一会儿就会看到如何接通这些事件处理器）。

让我们转移到类构造器。必须要留心构造器中的第一条语句，它调用了inherited Create。

警告：与过去通常使用的VCL和Delphi相比较，在这里会看到与.NET Framework方式的主要不同点。

在Delphi中，由构造器对成员变量进行初始化，使对象实例成为已知状态，它并不占用任何内存空间。因此利用构造器进行赋值，然后调用继承的构造器的情况并不罕见。但事实上，也许就根本没有调用继承的构造器。而在Delphi for .NET中，就无法逃避这种方法。在构造器中，必须要调用inherited Create，而且它还必须是方法的第一条可执行语句。如果当前无法进行这样的操作，那么编译器就会错报Self没有被初始化，因此在访问祖先的任何字段之前，必须要先调用继承的构造器。

在完成对继承的构造器的调用之后，就会返回到比较熟悉的领域中。尽管这个代码使用了不同的类分层结构，但任何Delphi程序员都应该很清楚它。可以通过调用Create构造器生成System.Windows.Forms.OpenFileDialog实例——这是如何创建任何.NET Framework类实例的具体实现方法。

接下来的几行语句解释说明属性的设置，它们是OpenFileDialogObject对象实例和窗体本身。最后，向窗体的Controls集中增加两个控件（文件名标记和用于保存程序集的ListBox），它是类型Control.ControlCollection的属性。

InitializeMenu过程说明了System.Windows.Forms.MainMenu对象实例的分配及规划。在File菜单进行初始化的地方，用一个动态数组保存每一个菜单条目。然后将这个动态数组传送到AddRange方法。可能已经分别通过为每个菜单条目调用Add方法，完成了对这个代码的编制。

在InitializeMenu中，下一个有趣的事情是编写菜单条目的事件处理器。在第24章和本章前一部分中，提到了包括委托和多点传输事件在内的后台复杂性。在这里可以看到其中的一些复杂性出现在前台。

现在还不能在Delphi for .NET中运行它，但在其他类似于C#的.NET语言中，可以使用语言的关键字event来引入事件处理器委托。事件声明会指定委托为一种回叫机制。因为事件是System.MulticastDelegate的一个派生（在这种情况下是一个System.EventHandler委托），因此其他对象可以添加和删除事件处理器，而且当事件激发时调用这些处理器。

C#语言添加了一点语法修饰，以使这个问题更加容易解决。C#定义了运算符+=和-=来分别添加和删除事件处理器。而Delphi最后利用前面提到的Include/Exclude机制更好地解决了这个问题。CTS要求所有指定这个事件模型的.NET编译器必须产生名为add_<Event>和remove_<Event>的方法。这些add_和remove_方法封装了在System.Delegate中声明的Combine和Remove方法。

目前，要想指派一个事件处理器，必须要使用这些add_和remove_方法；通常程序员可能不会关注它们，因为编译器一般都隐藏了这个复杂性。在当前类声明中，引入了两种方法：openItemClick和exitItemClick，其签名匹配System.EventHandler委托。然后在相应的菜单条目前调用add_Click方法，把事件处理器作为回叫方法传送。

下面让我们先看一看能够反映在程序集内定义的地型的代码。可以通过静态LoadFrom方法给出它的名称来调用任何程序集（从而创建一个对象实例）。一旦具有程序集对象，就可以进行随意的操作，并通过反射从任何角度观察这个程序集。

程序集内部所包含的模块集可以通过GetModules方法得到。从这里可以利用GetTypes一直向下找到模块中定义的类型。如同在InitializeMenu过程中所看到的，可以使用动态数组来暴露具有System.Array的集合。

最后，模块和类型数组的每个单独成员都包含了Name属性，可以利用它来构建一个在ListBox中显示的字符串。该代码最后生成的结果如图25.2所示。

利用Delphi语言实现ASP.NET

不管喜不喜欢（反正我不是那么喜欢），Microsoft的ASP技术在Web应用程序开发中起到了相当重要的作用，至少是在Windows平台上。随着这种技术向ASP.NET的转换，它已经完全接受了.NET Framework。现在，由于Delphi for .NET编译器的存在，Delphi语言可以作为开发ASP应用程序的被选语言。



图25.2 CLRReflection举例，其中加载了一个部件

如要想进行一下试验，可以配置IIS，以支持ASP.NET（这里将不对这些操作步骤进行介绍，它超出了本书的范围，但具体细节可以详见www.asp.net站点上的介绍），然后将web.config文件保存在目标文件夹下，该文件是由Borland与Delphi for .NET Preview（存放在aspx子文件夹下）共同分发的。该配置文件定义了语言到某个特定库的映射，它是由Borland所提供的。文件的核心内容（采用了XML格式）包含以下元素：

```
<compilation debug="true">
<assemblies>
    <add assembly="DelphiProvider" />
</assemblies>
<compilers>
    <compiler language="Delphi" extension=".pas"
        type="Borland.Delphi.DelphiCodeProvider,DelphiProvider" />
</compilers>
</compilation>
```

要测试该配置文件的正确性，没有比尝试一个例子更好的方法了。创建一个新文件（我将我的文件称为`aspbase.aspx`，保存在本章源代码的`DelphiAsp`文件夹中）并输入下列内容：

```
<html>
<body>

<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
procedure HelloMessagef@msg: stringf@;
var
    i: Integer;
begin
    for i := 2 to 7 do
        Response.Write ('<font size=' - inttostr (i) +
            '>' + msg + '</font> <br>')
```

```

end;
</script>

<% HelloMessage('Delphi for .NET Preview made this'); %>

</body>
</html>

```

运行结果是在将这个文件转换为.NET源代码后执行Delphi代码, 利用Delphi Preview编译器对其进行编译, 并将IL编译为程序集代码(记住, 在执行它们之前甚至也要对.NET中的脚本进行编译)。如果一切都运行正常, 浏览器应该显示出如图25.3所示的输出结果。

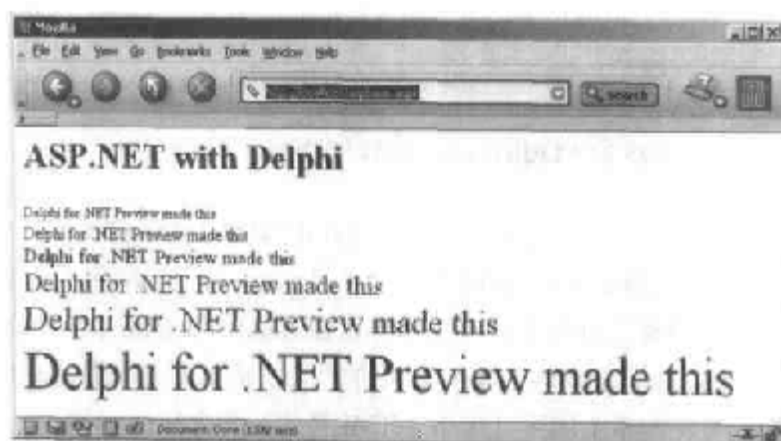


图25.3 在浏览器中的aspbase.aspx举例

从这一点上看, 可以对ASP.NET应用程序提供的任何事物进行操作。我想向读者显示的另一个例子是通过处理器来应用控制, 我已经在以Windows窗体为基础的.NET应用程序的不同环境下对其进行了介绍。

这个例子保存在AspDelphi文件夹下的aspui.aspx文件中, 它使用HTML定义了一个带有一个文本框(即编辑控件)、一个按钮以及一个输出标记的窗体。这个按钮有一个与其相连的Delphi语言事件处理器, 用于将用户输入功能移到这个标记上(程序的输出如图25.4所示)。

```

<html>
<body>
<h1>ASP.NET with Delphi</h1>
<script language="Delphi" runat="server">
  procedure ButtonClick(Sender: System.Object; E: EventArgs);
  begin
    Message.Text := Edit1.Text;
  end;
</script>
<form runat="server">
  <asp:textbox id="Edit1" runat="server"/>
  <asp:button text="Click Me!" OnClick="ButtonClick" runat="server"/>
</form>

```



```
<p><b><asp:label id="Message" runat="server" text="message" /> </b></p>  
</body>  
</html>
```

这只是利用Delphi语言实现ASP.NET的有限介绍。但是，它应该能够使读者感知到.NET的新领域将为Delphi程序员开辟新路的可能性。

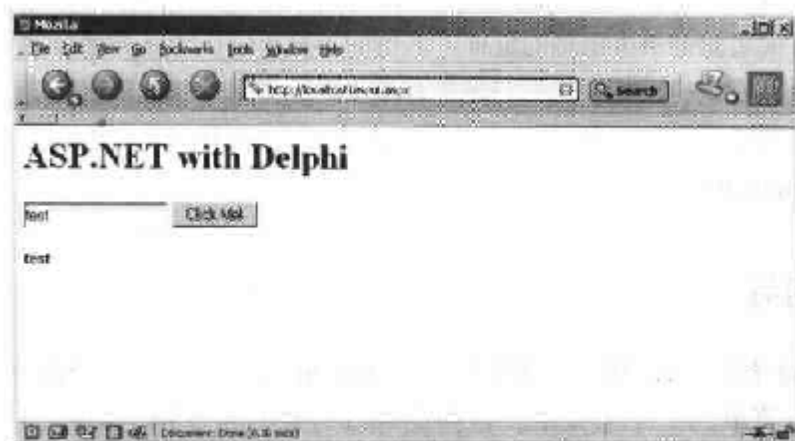


图25.4 在编辑对话框中进行输入和单击按钮后，aspui.aspx的输出举例

小结

在等待Delphi for .NET产品正式发行的同时（当前其代码被称做Galileo），可以开始试着使用与Delphi 7一同发行的Delphi for .NET Preview（现在可以得到Borland随后发布的更新版本）。当然，还应该与Borland的Developer Network站点（bdn.borland.com）和新闻组以及笔者的站点保持联系，以此来获得Delphi领域的更新信息，可以肯定的是这项工作正在进展中。

正是由于Borland要为开发人员提供最好的工具，所以我希望本书能够帮助读者掌握Delphi，它是Borland在最近几年推向市场的最成功的工具。要记住时常查看我的网站中（www.marcocantu.com）所收集的参考资料、基本资料以及高级资料。由于本书的篇幅所限，这些材料中的大部分内容都没有能够收编进来。详细内容见附录C。

在附录A和附录B中讨论了一些我所构造的附件，可以在我的站点上免费得到它们，还包括一些很重要的免费的Delphi工具。另外，注意查看我的站点上有关对本书材料的更新与综述内容。通常情况下，可以使用保存在这里的新闻组，以了解如何解决有关本书和Delphi的问题。

附录A 作者提供的其他Delphi工具

在最近几年中，我曾开发了一些小组件以及Delphi的附加工具。有些工具是为了图书的写做而编制的，或者就是书中范例的延伸。其他的工具可以作为某种操作的一个帮助。所有这些工具都可以免费获得，其中的一些还带有源代码。本附录中提供了一个列表，包括本书中提到的一些特殊工具。对这些工具的支持可以在我的新闻组中找到（请查看www.marcocantu.com）。

CanTools向导

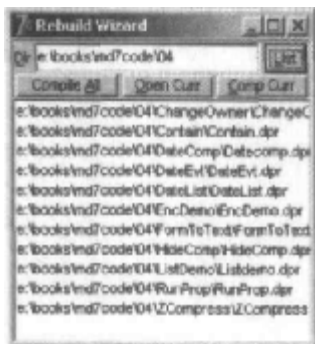
这是一个向导的集合，我们可以将其安装到Delphi中，或者放在一个额外的下拉菜单中，或者作为Tools菜单的一个子菜单。这些向导（可在这里获得：www.marcocantu.com/cantoolsw）是彼此无关的，提供不同的功能：

List Template Wizard 使类似的基于列表的类开发更加简单有效，每一个都带有各自类型的对象。这个向导在第4章中提到过。因为它是在一个库源文件中进行搜索/替代的操作，所以可以在需要重复代码或者类名发生变化的任何时候使用它。

OOP Form Wizard 在第4章中提到过。允许我们隐藏一个窗体的公布组件，让我们的窗体更加面向对象，并且提供一个更好的封装机制。当窗体激活的时候启动它，它将会填写OnCreate事件处理程序。然后我们必须手动地将部分代码移动到单元初始化部分。

Object Inspector Font Wizard 使我们能够改变Object Inspector中的字体（做演示的时候尤其重要，因为Object Inspector的字体太小了，从投影仪的屏幕上很难看清）。另一个设置对话框中的选项允许我们设定一个Object Inspector内部特性，并且使用一个特定字体显示字体名称（在一个下拉组合框中）。

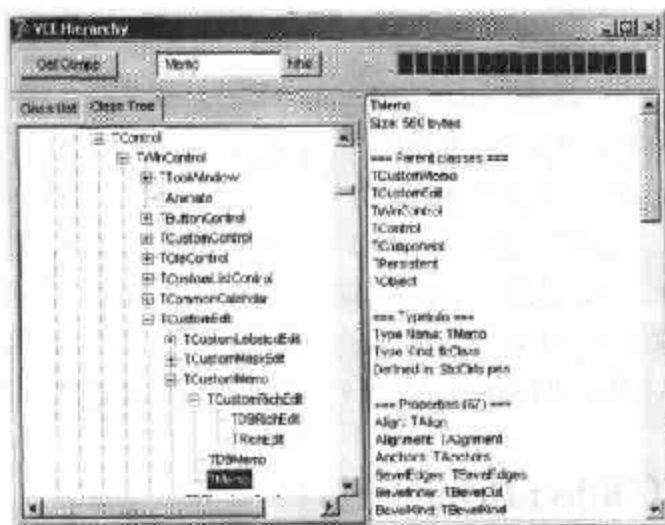
Rebuild Wizard 在向IDE中顺序载入每一个项目后，允许我们在一个给定的子目录中重建所有的Delphi项目。可以使用这个向导来获取一系列的项目（如本书中的那些），并且通过单击打开其中感兴趣的一个。



我们还可以自动编译一个给定的项目或者启动多项目构建；在编译器的结果对话框中，只有当相应的环境选项被选中的时候，才能单击按钮进行操作。如果环境选项没有被设置，我们将不会看到编译器错误，因为编译器的消息在每一次编译的时候都被替换掉了。

Clip History Viewer 跟踪复制到剪贴板中的一个文本单元的列表。在Viewer窗口的一个备注中显示了最近粘贴的100行内容。编辑这个备注（并且单击Save按钮）将更改剪贴板的历史内容。如果我们一直打开着Delphi，那么剪贴板将还会得到来自其他程序的文本（当然只是文本）。偶尔我也曾发现由这个向导产生的剪贴板相关的错误。

VCL Hierarchy Wizard 显示几乎是全部的VCL结构，包括已经安装的第三方组件，并且允许我们搜索一个类和查看很多的详细信息（基类和子类、公布的属性等等）。单击按钮将会重新生成列表与树（按顺序，因此程序栏要运行两次）。



类的列表通过使用预定义的核心类产生，并且向每一个已安装包的组件添加具有一个类类型的所有公布属性的类。但是，那些只作为公共属性的类不包括在内。

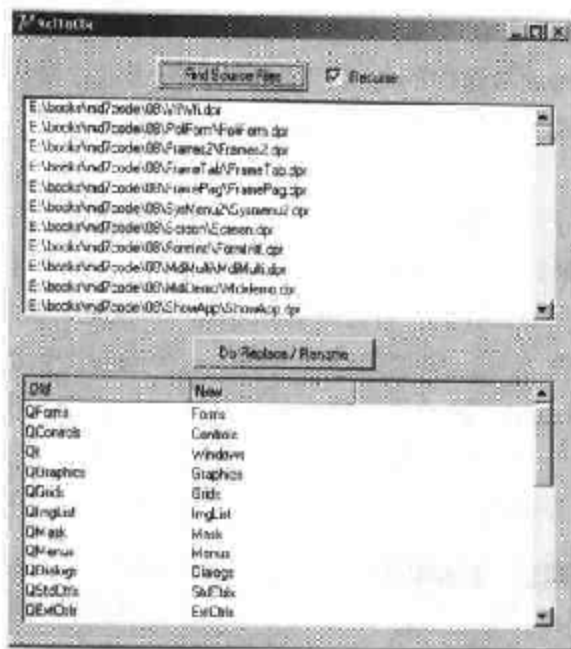
Extended Database Forms Wizard 完成比Delphi IDE中的Database Forms Wizard更多的功能，允许我们选择一个字段将其放置到一个窗体上，还允许使用不是基于BDE的数据集。

Multiline Palette Manager 允许我们将Delphi的组件板转换到一个带有多行标签的TabControl中。



VclToClx转换程序

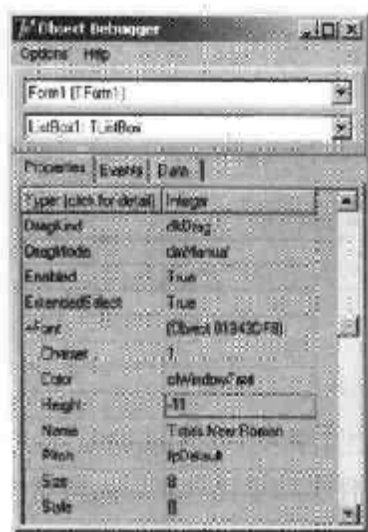
我们可以使用这个独立的工具来将一个Delphi项目从VCL转换到CLX（或者相反）。它可以同时转换位于一个给定文件夹和子文件夹内的所有文件，下图是它的输出范例。



这个程序的源代码可以在本书源代码的Tools目录中找到。VclToClx程序会更改单元的名称（基于一个配置文件）并且处理DMF问题——通过更改DFM文件的名字为XFM并且修复源代码中的引用。该程序是比较简单的，它不能够分析源代码，但是可以查找带有一个逗号或者分号的单元名称，就像一个uses语句那样。它还要求单元的名称前面带有一个空格，但是我们可以修改这个程序来查找一个逗号。不要跳过这个附加的测试，否则Forms单元将被转换为QForms，但是QForms单元还将被再次转换为QQForms！

对象调试器（Object Debugger）

在设计时，我们可以使用Object Inspector来设置窗体或者其他设计器上组件的属性。在Delphi 4中，Borland引入了一个运行时的Debug Inspector，它具有一个类似的界面并且显示类似的信息。在Borland加入这个新的特性之前，我们实现了一个Object Inspector的运行时克隆，用以调试程序，如下图所示。



它允许对一个组件的所有公布属性进行读写访问，并且具有两个组合框，可以让用户选择窗体以及窗体上的组件。某些属性类型具有定制属性编辑器（列表等）。

我们可以将Object Debugger组件放置在一个程序的主窗体上（或者在代码中动态地创建）：它将出现在它自己的窗口内。虽然在目前的状态下，这个工具很方便也拥有众多的用户，但是改进的空间还是有的。

这个组件的源代码可以在本书源代码的Tools目录中找到。

内存快照 (Memory Snap)

在Delphi中有很多的工具可以用于追踪内存的状态。在开发一个项目的过程中，我不得不编写了这样一个工具，之后，我将它最终完成了。

我编写了一个定制的内存管理器，它附加在Delphi默认的内存管理器中，用于追踪所有的内存分配与释放操作。除了报告总数外（这也是Delphi默认所做的），它可以将内存状态的详细描述保存到一个文件中。

Memory Snap在内存中保留了一系列被分配的块（根据一个可以调节的最大许可值），因此它可以将堆的内容保存到一个文件中。这个列表是通过检查每一个内存块并且根据实验技术确定它们的本质而生成的，我们可以看到那些实验技术的源代码（虽然不是很好懂）。输出被保存在一个文件中，因为这是唯一一个不需要进行内存分配的操作，也就不会因此而影响最终的结果。下面是范例文件的一个片断：

```
157) 00C035CC: object: [TList - 16]
158) 00C035E0: buffer with heap pointer [00C032B0]
159) 00C03730: string: [5-1]: Edit1
160) 00C03744: object: [TEdit - 544]
161) 00C03968: object: [TFont - 36]
162) 00C03990: object: [TSizeConstraints - 32]
163) 00C039B4: object: [TBrush - 24]
164) 00C039F4: buffer with heap pointer [00C01FE4]
165) 00C03B34: buffer with heap pointer [00C01F18]
166) 00C03B48: string: [0-0]: dD
167) 00C03B58: string: [11-2]: c:\mman.log
```

该程序可以进行扩展，根据类型（字符串、对象、其他块等）来对内存的使用进行划分，跟踪没有释放的块，并且控制内存的分配。

同样，这个组件的源代码可以在本书源代码的Tools目录中找到。

许可证与贡献

正如我们看到的，某些工具带有全部的源代码，它们得到了LGPL的许可（Lesser General Public License, www.gnu.org/copyleft/lesser.html）。这意味着我们可以自由地使用和分发这些代码——以任何方式，包括进行修改——与此同时，笔者仍然保留对这些代码的原始版权。LGPL不允许程序员封闭自己扩展的源代码，但是可以在销售的程序中使用库代码，

而不论源代码是否可用。如果读者通过修补漏洞或者增加新的特性扩展了这些工具，那么我希望各位将你们的更新发送给我，以便我可以进一步的分发它们，而且也可以避免出现太多的代码版本。当然，许可证本身并不要求大家这么做。

附录B 其他来源提供的Delphi工具

数以千计的Delphi附加组件和工具可以在市场上找到，从简单的赠品到大型的开放源代码项目，从共享件程序到高度专业的组件。这个附录将提供一个值得关注的开放源代码项目的列表。我在这本书中的其他地方曾经提到过这个列表。

Delphi预安装的开放源代码组件

Delphi 7包括两个著名的开放源代码项目的源代码：

Internet Direct (Indy) 这个项目在第19章“因特网编程：套接字与Indy组件”中进行了详细介绍。关于Indy的官方站点是www.nevrona.com/indy，但是我们也应该参考有关Indy Portal的介绍：www.atozedsoftware.com/indy，由Borland新闻组提供支持。

Open XML 这是一个基于Delphi的XML DOM和SAX引擎，我们在第22章“使用XML技术”中介绍过。请参考www.philo.de/xml，以获得更多的信息。对Open XML的支持由一个邮件列表提供，可以在网站上进行注册。

Delphi还包括一个开放源代码项目：ZLib压缩库，我们在第4章“核心类库”中进行了介绍。我们并未将其列在这里是因为它不是基于Delphi的。

其他开放源代码项目

Delphi程序员社区从它诞生之时起就非常活跃，并且创造了众多的工具，在程序员中免费分发。这些工具中的一部分具有完整的源代码，对于一些商业Delphi组件也是如此。

Project JEDI

Project JEDI即Joint Endeavor of Delphi Innovators (www.delphi-jedi.org)，不是一个单独的项目，而是一个最大的开放源代码Delphi开发者社区。这个站点拥有自己的项目，以及其他不同Web站点分发与维护的项目。

Project JEDI开始时是为了翻译特定的、由微软或者其他公司分发的Windows API而建立的。使用这些API的声明即可使Delphi单元可用，也就是说允许任何Delphi开发者方便地使用它们。在最近，Project JEDI的目标得到了进一步扩展，包含很多子项目和组的定义。除了一个不断扩张的API库以外，我们还可以找到很多的项目，包括JEDI可视组件库(JVCL)、JEDI代码库(JCL，一个工具函数和不可视类的集合，包括很好的未处理的异常堆栈跟踪器)，以及很多用于图像、多媒体和游戏的项目。其他内容也很丰富，从JEDI版本的控制系统客户程序到DARTH头部转换工具，从程序员的编辑器到在线教程。

其他JEDI相关的项目包括Indy（前面提到）、Gexperts和Delphree站点（下面介绍）。

Gexperts

Gexperts (www.gexperts.org) 可能是传播最广泛的Delphi IDE附件, 提供的特性包括从多行组件面板到源代码定位帮助。自我描述为: “一个工具的集合, 是为了提高Delphi和C++ Builder程序员的生产效率而建立的”, 它包括一个很大的向导集合, 包含Procedure List、Clipboard History、Expert Manager、Grep Search、Grep Regular Expressions、Grep Results、Message Dialog、Backup Project、Set Tab Order、Clean Directories、Favorite Files、Class Browser、Source Export、Code Librarian、ASCII Chart、PE Information、Replace Components、Component Grid、IDE Menu Shortcuts、Project Dependencies、Perfect Layout、To Do List、Code Proofreader、Project Option Sets以及Components to Code。

Delphree站点

在开放源代码和社区开发领域有很多其他的Delphi相关的项目。虽然我不能够列举出所有这些, 但是我能够向大家介绍一个尝试追踪所有类似项目的站点: Delphree (Delphi Free) 站点, 它位于delphree.clexpert.com。

这个站点具有一个可观的Delphi开放源代码项目列表, 覆盖范围从库到程序员工具以至最终用户应用程序。

DUnit

在众多的由极限编程推崇的思想中, 我发现单元测试 (Unit Testing) 非常有趣。Unit Testing是在一个程序编写之前, 对测试代码的持续开发。为此, 拥有一个合适的测试框架是很重要的。一组Delphi程序员创建了这个体系结构, 称为DUnit。我们可以在SourceForge的dunit.sourceforge.net站点上找到它。

附录C 本书配套的Delphi免费读物

这是第七版的“Mastering Delphi”，笔者也编写过其他的书籍、有关类的资料、参考文献等等。因此，除了这本书中的内容外，我还有很多关于Delphi编程的资料，特别是介绍性的资料。在过去的几年中，我将其中的一部分内容制成电子图书（使用HTML或者PDF的格式），放在我的网站上供免费使用。本附录将列举它们的标题与内容，并且提供下载页面的参考（所有这些都可以在www.marcocantu.com中找到）。

Essential Pascal

“Essential Pascal”是对语言基础的介绍，该语言由Nicklaus Wirth教授发明并且由Borland公司于20世纪80年代在它著名的Turbo Pascal编译器中使用。这本100页的图书没有介绍OOP的扩展，但是详细介绍了多年来Borland公司在核心语言中添加的增强特性。下面是这一电子图书的目录，可以在www.marcocantu.com/epascal中获得：

- 第1章：Pascal历史
- 第2章：在Pascal中编码
- 第3章：类型、变量与常量
- 第4章：用户定义的数据类型
- 第5章：语句
- 第6章：过程与函数
- 第7章：处理字符串
- 第8章：内存（和动态数组）
- 第9章：Windows编程
- 第10章：变量
- 第11章：程序和单元
- 附录A：术语词汇
- 附录B：范例

读者可以在网站上看到，这本书被志愿者翻译成若干种不同的语言。

Essential Delphi

“Mastering Delphi”是一本中级读物，其中尽可能多地介绍了在多年之中，Borland公司在Delphi中添加的特性。在此书编写的过程中，我去掉了一些给入门级程序员学习用的Delphi及其可视工具的相关内容。这些内容被包含在“Essential Delphi”这本电子图书中，可以在www.marcocantu.com/edelphi中获得。下面是该书的目录：

- 第1章：窗体就是窗口
- 第2章：Delphi环境的重点
- 第3章：对象库与Delphi向导
- 第4章：基本组件简介
- 第5章：创建并处理菜单
- 第6章：多媒体
- 第7章：保存设定：从INI文件到注册表
- 第8章：关于窗体的更多话题
- 第9章：Delphi数据库（使用Paradox）
- 第10章：打印
- 附件A：SQL基础
- 附件B：常用VCL属性

Delphi Power Book

最后，我还提供了更多的资料，它们是关于更高级内容的，我把它们放在一个叫做“Delphi Power Book”的集合中。在本书编写的时候，只有四个电子章节可供大家参考：“Delphi中的图像”、“使用接口的范例”、“COM Shell扩展”以及“调试”。

更多的内容将会不断出现，请查阅：www.marcocantu.com/delhipowerbook。

欢迎与我们联系

为了方便与我们联系，我们已开通了网站（www.medias.com.cn）。您可以在本网站上了解我们的新书介绍，并可通过读者留言簿直接与我们沟通，欢迎您向我们提出您的想法和建议。