DataSource for Entity Framework for WPF/Silverlight

2018.02.20 更新

グレープシティ株式会社

目次

- DataSource for Entity Framework の概要 3
  - ComponentOne Studio for WPF/Silverlight のヘルプ 3
- 主な特長 4-5
- C1DataSource の概要 6
  - 統合データコンテキスト 6
  - 仮想データアクセス 6
  - 強力なデータ連結 6-7
- クイックスタート 8
  - 手順1:データソースの追加 8-9
  - 手順2:C1DataSource へのデータの接続 9-10
  - 手順3:グリッドの追加 10
  - 手順4:プロジェクトの実行 10-11
- DataSource for Entity Framework 12
  - 簡単な連結 12-15
  - サーバー側のフィルタ処理 15-16
  - クライアントデータキャッシュの能力 16-17
  - マスター/詳細連結 17-18
  - 大規模なデータセット:ページング 18-21
  - 大規模なデータセット:仮想モード 21-22
  - グリッドの自動ルックアップ列 22-23
  - ビューのカスタマイズ 23-25
  - コードでのデータソースの操作 25-30
  - ライブビュー 30
  - MVVM の簡略化 30-34
  - 他の MVVM フレームワークを使用した MVVM での C1DataSource の使用 34
  - プログラミングガイド 34
    - ライブビューでサポートされるクエリー演算子 35
    - ライブビューでサポートされるクエリー式 35-36
    - ビューメンテナンスモード 36
    - 更新可能なビュー 36-37
    - ライブビューのパフォーマンス 37

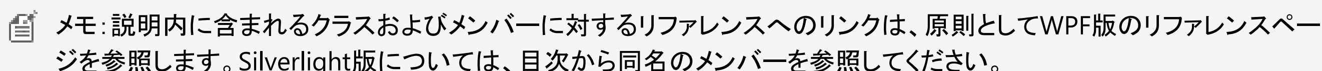
Copyright © GrapeCity Inc. All rights reserved.

LiveLinq クエリーパフォーマンス：論理的な最適化 37-38
LiveLinq クエリーパフォーマンス：インデックスパフォーマンスの調整 38
ライブビュー：他のビューに基づいてビューを作成し、ビューにインデックスを作成する方法 38-40
ライブビュー：ライブビューを使用して非 GUI アプリケーションを作成する方法 40-41
C1LiveLinq 42
LiveLinq の概要 42
LiveLinq (Silverlight の場合) 42
LiveLinq の使用方法 42
LiveLinq でコレクションをクエリーする方法 42-43
組み込みのコレクションクラス IndexedCollection の使用（LiveLinq to Objects） 43-44
ADO.NET データコレクションの使用（LiveLinq to DataSet） 44
XML データの使用（LiveLinq to XML） 44-45
連結可能コレクションクラスの使用（LiveLinq to Objects） 45-46
LiveLinq to Objects：IndexedCollection および他のコレクションクラス 46
インデックスの作成方法 46-47
プログラムでインデックスを使用する方法 47
ライブビューの作成方法 47-48
GUI コントロールをライブビューに連結する方法 48
非 GUI コードでライブビューを使用する方法 49
プログラミングガイド 50
コードでのエンティティの操作 50-51
コードでのエンティティの操作 51
クライアント側トランザクション 51-53
仮想モード 53-54
Unmanaged 仮想モードの制限 54
その他の仮想モードの制限 54

Copyright © GrapeCity Inc. All rights reserved.

DataSource for Entity Framework の概要

DataSource for Entity Framework を使用すると、Entity Framework および RIA サービスがさらに使いやすくなり、そのパフォーマンスを向上させることができます。データのロード、ページング、フィルタ処理、保存などの一般的な問題が解決されるため、これらのフレームワークを使用したデータ連結が強化されると共に容易になります。パフォーマンスの向上には、仮想モードによる大規模なデータセットの高速なロードや透過スクロールもあります。



メモ：説明内に含まれるクラスおよびメンバーに対するリファレンスへのリンクは、原則としてWPF版のリファレンスページを参照します。Silverlight版については、目次から同名のメンバーを参照してください。

ComponentOne Studio for WPF/Silverlight のヘルプ

はじめに

ComponentOne Studio for WPF/Silverlight のすべてのコンポーネントで共通の使用方法については、「ComponentOne Studio for WPF/Silverlight ユーザーガイド」を参照してください。

Copyright © GrapeCity Inc. All rights reserved.

主な特長

以下に、**DataSource for Entity Framework** の便利な機能をいくつか示します。

メモ:このバージョンの **DataSource for Entity Framework** には、Entity Framework 6 以上、.NET Framework 4.5 以上、および Visual Studio 2012 以上が必要です。

- **設計時コンポーネントによる Entity Framework の容易化**
  C1DataSource を使用して、デザインサーフェスでデータソースを直接設定できます。このとき、使いやすいプロパティダイアログを使用することで、記述するコードを最小限に抑えることができます。設計時に、**C1DataSource** コントロールを迅速に設定して、サーバー側のフィルタ、ソート、およびグループディスクリプタを適用できます。もちろん、必要に応じて、機能豊富なデータクラスライブラリを使用して、すべてをコードで実行することもできます。
- **LiveLinq によるライブビュー**
  LINQ は、生データをカスタムビューに変換するために最適なツールです。**C1DataSource** では、LINQ クエリーステートメントを動的に実行できるので、LINQ がさらに強力になります。**C1DataSource** には C1LiveLinq が含まれています。これは、LINQ の機能を補完してクエリーを迅速化し、ライブビューを提供する拡張ライブラリです。LiveLinq では、LINQ の演算子を使用して、更新可能性と連結可能性を損なうことなく、必要に応じたビューを形成できます。"連結可能性" とは、ビューがソースの静的なスナップショットではないことを意味します。ビューは "ライブ" であり、データの変更が自動的に反映されます。LiveLinq を使用すると、データが変更されるたびにデータを再挿入しなくても、クエリー結果が最新の状態に保たれます。
- **MVVM の簡略化**
  C1DataSource は、広く採用されている Model-View-ViewModel パターン(MVVM)を使用したプログラミングを簡略化します。MVVM アプリケーションを開発するには、多くのコードを記述する必要があります。これは、追加コードレイヤ、ViewModel、Model と ViewModel のデータメンバ間の同期などに対応するコードが必要なためです。DataSource を使用すると、ライブビューを ViewModel として使用でき、同期コードを書く手間が不要になります。ライブビューはソースと自動的に同期され、他の方法よりも作成が非常に簡単です。**C1DataSource** は、任意の MVVM フレームワークと組み合わせて使用することができます。
- **仮想モードによる大規模なデータセットの操作**
  C1DataSource の仮想モードを使用して、無制限に大規模なデータセットを操作できます。仮想モードにより、大規模なデータセットを非同期に操作することができます。仮想モードは、データレイヤでのページングと同様に機能しますが、ユーザーは、すべての行がクライアント上にあるかのようにデータをスクロールすることができます。ユーザーがスクロールを行うと、データチャンクが必要に応じてソースからページごとに取得、破棄されます。仮想モードは、**DataGrid**、**C1FlexGrid** などの GUI コントロール、または任意のコントロールで使用できます。データを変更することもできます。この機能は開発者にとって透過的であり、仮想モードは1つの単純なプロパティを設定するだけで有効にできます。
- **データ変更を伴うページング**
  ページングインタフェースを使用するアプリケーションのために、**C1DataSource** は、データ変更に制限のないページングもサポートしています。つまり、ユーザーは、データベースに変更を送信する前に、1つのセッションで複数のページに変更を加えることができます。これは、Microsoft RIA サービスの DomainDataSource が備えているような他のページングの実装よりはるかに優れています。ページング機能が提供されていない WinForms では、**C1DataSource** を使用してページングを行うことができます。
- **高性能なクライアント側キャッシュ**
  DataSource for Entity Framework のほとんどの機能で、組み込みのクライアント側データキャッシュが重要な役割を果たしています。 **C1DataSource** は、エンティティのキャッシュをクライアント上に保持します。新しいクエリーが実行される場合でも、必ずしもサーバーにはアクセスしません。この場合は、最初にクライアント側のキャッシュを確認します。キャッシュに結果が見つかった場合、サーバーにはアクセスしません。サーバーとのやり取りを最小限に抑えることで、パフォーマンスと速度が劇的に向上しました。
- **コンテキストの管理**
  C1DataSource を使用すると、アプリケーション全体で1つのデータコンテキストを使用することができます。このため、複数のビューにまたがる複数のコンテキストに対してプログラミングを行う煩わしさから解放されます。**C1DataSource** がないと、別のコンテキストからエンティティを同時に使用する必要がある場合に、複数のコンテキストを想定する必要があるため、コードを記述することが非常に難しくなります。クライアント側の優れたキャッシュ機能により、この機能が実現されました。
- **メモリ管理**

Copyright © GrapeCity Inc. All rights reserved.

C1DataSource は、パフォーマンスとメモリ消費の両方について最適化されています。メモリリークを防ぐため、必要に応じてキャッシュにある古いエンティティオブジェクトが自動的に解放され、メモリリソースが管理されます。このとき、必要なエンティティとユーザーによって変更されたエンティティが維持されるので、データの一貫性が保持されます。

- **サーバー側のフィルタ処理**
  ネットワークを介して大容量のデータが移動し、クライアント側でメモリが大量に消費されることを避けるため、通常、サーバーからクライアントに提供するデータは、何らかの方法でフィルタ処理または制限する必要があります。**C1DataSource** では、この一般的なタスクを簡単に実行することができます。つまり、コードを使用して手作業で実行する代わりに、**C1DataSource** で単純なプロパティ設定を行うだけで、サーバー側のフィルタ処理を指定できます。
- **クライアント側トランザクション**
  DataSource for Entity Framework には、サーバーを介すことなく、クライアント側で変更をロールバック（キャンセル）したり受け入れたりすることができる、開発者向けの単純かつ強力なメカニズムが用意されています。**C1DataSource** を使用すると、モーダルでもモードレスでも、ネストされた（子）ダイアログボックスやフォームを含む任意の場所に、［**キャンセル/元に戻す**］ボタンや［**OK/保存**］ボタンを簡単に実装することができます。
- **変更済みデータの保存**
  C1DataSource の優れたクライアントキャッシュ機能により、変更済みデータをサーバーに保存するためのコードを簡単に記述することができます。コードを１行記述してサーバーに一度移動するだけで、複数のエンティティを保存することができます。手間のかかる作業がすべて DataSource によって実行されます。変更されたエンティティがキャッシュに保存され、キャッシュの一貫性が常に維持されると共に、メモリ消費量やパフォーマンスが最適化されます。
- **設計時のコードファーストのサポート**
  C1DataSource は、コードをモデルから生成しないコードファーストのシナリオでも使用できます。**C1DataSource** の "ライブ" 機能が必要な場合は、エンティティクラスで **INotifyPropertyChanged** インタフェースを実装します。また、それらのクラスにコレクションナビゲーションプロパティがある場合は、**ObservableCollection** インタフェースを使用します.
- **クロスプラットフォームのサポート**
  DataSource for Entity Framework には **C1DataSource** コンポーネントが含まれています。これにより、Entity Framework や RIA サービスを使用して、複数のクライアントビューソースを組み合わせることができます。**C1DataSource** は、WinForms（.NET 4.5 以上）、WPF、および Silverlight 4 でサポートされています。

Copyright © GrapeCity Inc. All rights reserved.

C1DataSource の概要

マイクロソフトは、Entity Framework の設計に際して、開発者がデスクトップアプリケーション/サーバーアプリケーションの土台となるデータベースや WCF RIA サービスと簡単にやり取りできるように、プラットフォームに依存しない手法の開発に乗り出しました。この原則は、Silverlight プラットフォームにまで受け継がれています。これらのフレームワークは、データの永続化を支援する(基底のデータベースのデータの取得と格納を制御する)ためのほぼ理想的なソリューションを開発者に提供していますが、アプリケーションロジックを作成したり、今日構築されているほとんどのアプリケーションに不可欠な連結 GUI コントロールの操作を簡単に行う手段を提供するには至っていません。**DataSource for Entity Framework** は、この欠点を補うことを目的として設計されました。追加機能が提供され、アプリケーション全体のパフォーマンスが向上したことで、両方のフレームワークが強化されています。換言すれば、開発者は、典型的なビジネスアプリケーションをより高速に開発できるようになり、より少ないコードで目的を達成できるようになりました。

以降のトピックでは、パフォーマンスが強化された **C1DataSource** の機能を詳しく検証します。最初に、主要な2つのフレームワークに加えられた重要な強化点を確認します。

統合データコンテキスト

Entity Framework でも **WCF RIA** サービスでも、コンテキスト(セッション)の管理およびコンテキストの明示的な作成は、開発者が行う必要があります。さらに、場合によっては、アプリケーションのフォームごとに個別のコンテキストを作成する必要があります。あるコンテキストで取得されたオブジェクトと別のコンテキストで取得されたオブジェクトは、本質的には同じオブジェクトであっても、何も関係がありません。このため、フォームどうし(または1つのフォーム内のタブコントロールにあるタブどうしでも)がやり取りする必要がある場合は、深刻な問題が発生することがあります。これまで、これらの問題を解決する従来の方法では、データを基底のデータベースに繰り返し保存し、GUI を継続的にリフレッシュすることで、発生した変更を反映してきました。その結果、多くの繰り返しコードが必要になり、データベースサーバーに繰り返しアクセスするためにアプリケーション全体のパフォーマンスが低下していました。

DataSource for Entity Framework は、統合データコンテキストと高度なクライアントキャッシュ機能を組み合わせることで、より優れたソリューションを提供しています。アプリケーションに必要なデータコンテキストは1つだけで、これがアプリケーション内のすべてのフォームとコントロールで使用されます。高度なクライアントキャッシュ機能により、コンテキストのオブジェクトに加えられた変更がすべて管理されます。つまり、クライアント側トランザクション機能が有効になり、ビューを頻繁に保存したりリフレッシュする必要がなくなるため、アプリケーションのパフォーマンスが向上し、コードが簡略化されます。また、ローカルデータキャッシュ機能により、クライアント側のクエリーですべての LINQ 機能(フィルタ処理、ソート、グループ化など)を使用することができます。ローカルデータキャッシュ機能なしでは、これは不可能です。

メモ:**DataSource for Entity Framework** は、新しい **DbContext**(Visual Studio でのデフォルトのコンテキスト)および従来の **ObjectContext** の両方の Entity Framework コンテキストをサポートしています。

仮想データアクセス

DataSource for Entity Framework は、負荷が大きなページングではなく、独自の仮想モード機能を使用して、数千行から数百万行までの大規模なデータセットへのアクセスとデータ連結をサポートします。このモードでは、**C1DataSource** は、連結コントロールに表示する必要がある行を必要なときにデータベースから自動的に取得し、それらが不要になると破棄します。このように、大規模なデータセットが、メモリ内にあってデータ連結の準備ができているかのように表示されます。追加コードを記述する必要はなく、連結コントロールに現在表示されている行に必要なメモリ以上のメモリリソースを消費することもありません。

大規模なデータセットを操作するために最もよく使用されている方法は、ページングです。データソースに直接連結されているデータグリッドなどの GUI コントロールと比べて、通常、ページングを使用した場合は使用感が悪くなります。仮想モードを使用すると、ページングを使用することなく、機能豊富な GUI コントロールを大規模なデータセットに直接連結することができます。

強力なデータ連結

Copyright © GrapeCity Inc. All rights reserved.

DataSource for Entity Framework for WPF/Silverlight

Entity Framework と RIA サービスの両方でサポートされている組み込みのデータ連結は、サーバーから最初に取得された同じクエリー結果への連結に制限されます。つまり、元のクエリーのサブセットに連結したり、元のクエリーを再形成することはできません。また、組み込みのデータソースコントロール(**DomainDataSource**)は、Silverlight の RIA サービスでのみ使用できます。このコントロールには、すべてのデータの変更がデータベースにコミットされない限り、ページングやフィルタ処理を実行できないといった厳しい制限があります。

DataSource for Entity Framework には、WPF と WinForms、さらに RIA サービス向けに、Entity Framework に直接アクセスするためのデータソースコントロールが用意されています。このコントロールには、標準の DomainDataSource の制限はありません。

データソースコントロールとは別に、**C1DataSource** ではライブビューを作成することができます。ライブビューでは、双方向のライブデータ連結に使用されるコレクションを宣言型で簡単に再形成することができます。これには、ソートやフィルタ処理だけでなく、計算フィールドや LINQ 演算子も含まれます。ライブビューを使用すると、アプリケーションロジックの大半(またはすべて)を宣言型データ連結で表現することができ、より短く信頼性が高いコードを生成できます。また、エンティティコレクションに基づいてライブビューを定義することで、使い慣れた MVVM(Model-View-ViewModel)パターンに従うビューモデルレイヤをアプリケーションで作成することができます。ビューモデルを作成するためのコードが少し必要ですが、モデルと同期させるためのコードは必要ありません。**C1DataSource** なしでは、手作業で大量のコードを記述する必要があります。

Copyright © GrapeCity Inc. All rights reserved.

クイックスタート

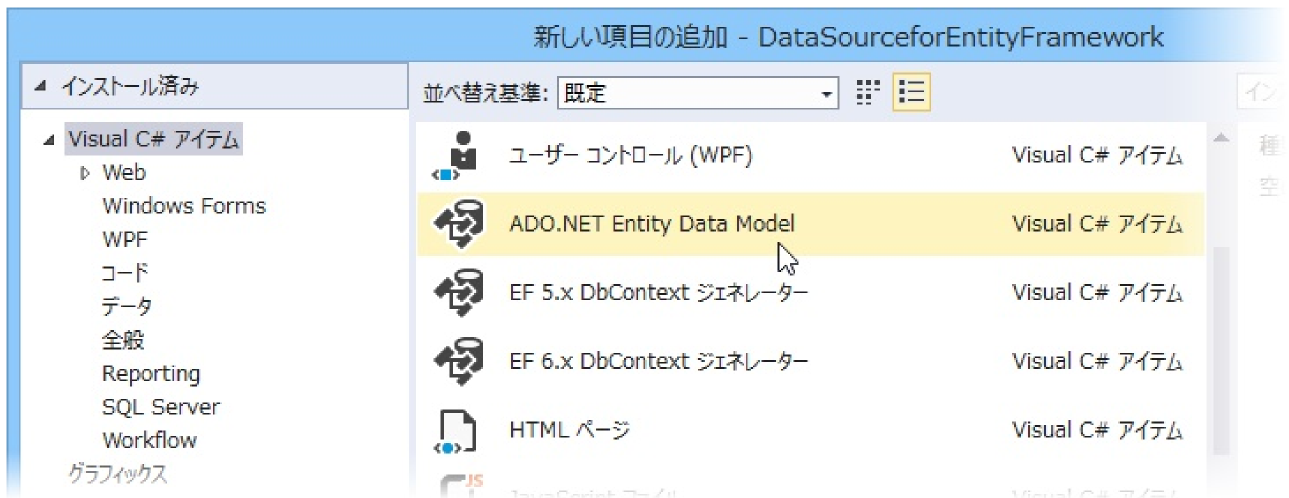
いくつかの簡単な手順を実行するだけで、**DataSource for Entity Framework** の使用を開始することができます。データソースをプロジェクトに追加し、それを **C1DataSource** コンポーネントに接続して、データを表示するグリッドを追加します。このクイックスタートには、連結の最も基本的な手順が示されています。操作の詳細については、このマニュアルの「簡単な連結」トピックを参照してください。

このクイックスタートチュートリアルでは Northwind データベースを使用します。このデータベースは、C:\Users\<username>\Documents\ComponentOne Samples (Windows 7/Vista) or C:\Documents and Settings\<username>\My Documents\ComponentOne Samples (Windows XP) フォルダに、製品と共にインストールされます。

手順1:データソースの追加

最初に、Visual Studio で新しい Windows フォームプロジェクトを作成します。次に、Northwind データベースへの接続を追加します。

1. Visual Studio で、**[ファイル]**→**[新規作成]**→**[プロジェクト]**を選択します。
2. **[WPF アプリケーション]**テンプレートを選択し、**[OK]**をクリックします。
3. 次に、Northwind データベースに基づいてエンティティデータモデルを追加します。このモデルにより、アプリケーション全体のデータが提供されます。
4. **ソリューションエクスプローラー**で、プロジェクト名を右クリックし、**[追加]**→**[新しい項目]**を選択します。
5. **[ADO.NET エンティティデータモデル]**項目を選択し、**[追加]**をクリックします。
6. **エンティティデータモデルウィザード**で、Northwind データベースからモデルを生成するために**[データベースから生成]**を選択し、**[次へ]**をクリックします。



7. **[新しい接続]**ボタンをクリックします。
8. **[データソースの選択]**ダイアログボックスで**[Microsoft SQL Server データベースファイル]**を選択し、**[続行]**をクリックします。
9. NORTHWND.MDF を見つけて選択し、**[開く]**をクリックします。NORTHWND.MDF は、C:\Users\<ユーザー名>\Documents\ComponentOne Samples(Windows 7/Vista の場合)または C:\Documents and Settings\<ユーザー名>\My Documents\ComponentOne Samples(Windows XP の場合)の **Studio for WPF\C1DataSource \Data** フォルダに、製品と共にインストールされます。
10. **[OK]**を選択し、**[次へ]**をクリックします。
11. **[データベースオブジェクトの選択]**ウィンドウで、**[テーブル]**を選択し、**[完了]**をクリックします。
12. ソリューションエクスプローラーで、model1.edmx ノードを展開します。

Copyright © GrapeCity Inc. All rights reserved.

13. Model1.Context.tt ファイルを削除します。
14. Model1.tt ファイルを削除します。
15. モデル図を右クリックし、コンテキストメニューから**[コード生成項目の追加]**を選択します。
16. [コード生成項目の追加]ダイアログボックスで、**[ComponentOne EF 6.x DbContext Generator]**を選択します。

メモ: **DataSource for Entity Framework** をインストールすると、**C1DataSource** がサポートする各バージョンの Visual Studio に **ComponentOne EF6.x DBContext コード生成テンプレート**が追加されます。これらのテンプレートを使用すると、作成する DBContext モデルによって **INotifyPropertyChanged** をサポートするエンティティが提供されます。

ここでプロジェクトをビルドすると、新しいエンティティデータモデルクラスが生成され、プロジェクト全体で使用可能になります。エンティティデータモデルクラスが使用可能になると、データを **C1DataSource** コンポーネントに接続できるようになります。

手順2：C1DataSource へのデータの接続

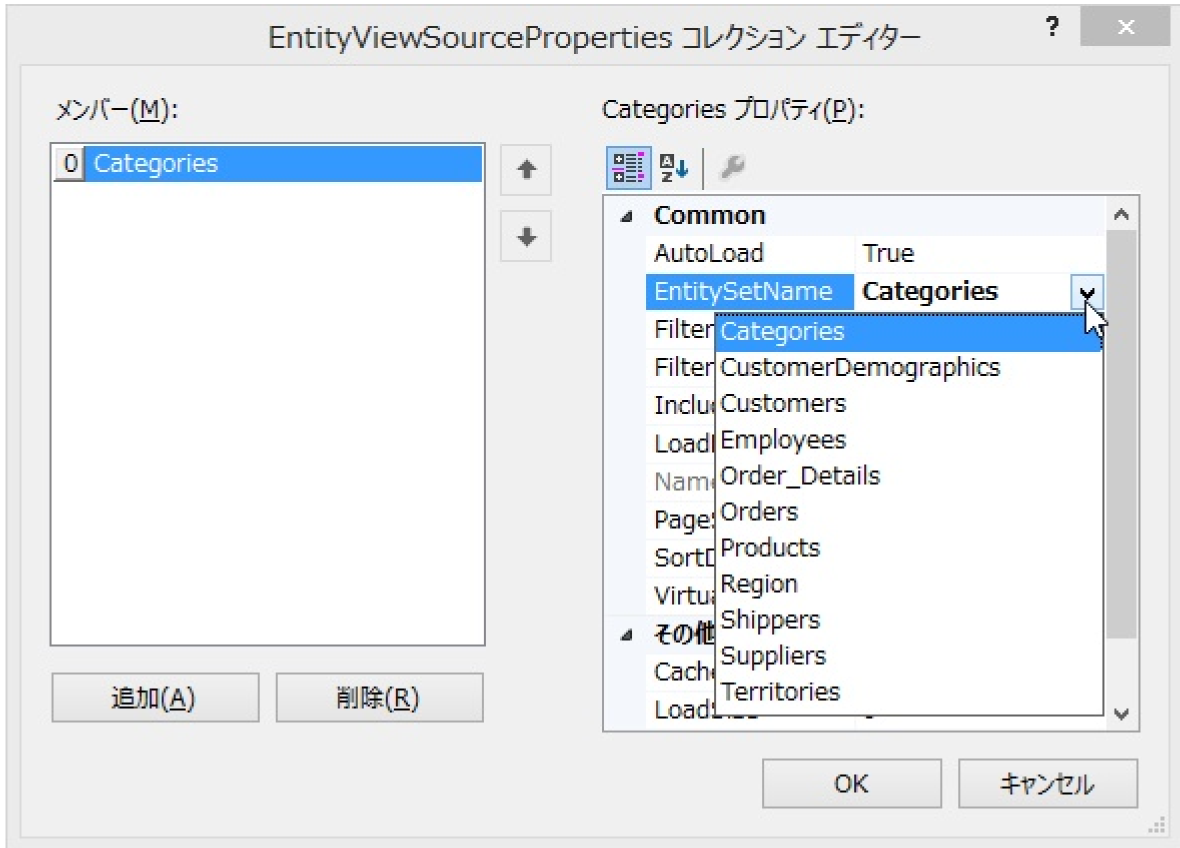
1. この手順では、ウィンドウ（またはページ）に **C1DataSource** コンポーネントを追加し、それをデータソースの Categories テーブルに接続します。ソリューションエクスプローラーで MainWindow.xaml をダブルクリックしてページを開きます。

2. **C1DataSource** コンポーネントをツールボックスからウィンドウにドラッグし、**c1DataSource1** という名前を付けます。これは非ビジュアルコンポーネントです。そのため、ウィンドウの内部コンテンツ内の任意の場所に配置できます。ツールボックスに表示されていない場合は、ツールボックスを右クリックし、[項目の追加]を選択します。[ツールボックス項

Copyright © GrapeCity Inc. All rights reserved.

目の選択]で、WPF コンポーネントを選択します。[参照]をクリックし、**C1.WPF.Data.Entity.4.dll** を参照して追加します。

3. [プロパティ]ウィンドウで、**C1DataSource** の **ObjectContextType** プロパティをドロップダウンリストにある項目に設定します。これは **AppName.NORTHWINDEntities** のような名前になっています。
4. **ViewSources** プロパティの横にある省略符ボタンをクリックして、**ViewSources** コレクションエディタを開きます。
5. [追加]をクリックし、EntitySetName プロパティを Categories に設定します。



6. [OK]をクリックして、エディタを閉じます。

データベースが **C1DataSource** に接続されました。次に、グリッドを追加してデータを表示します。

手順3：グリッドの追加

style="MARGIN: 17pt 0in 0in; mso-style-name: 'Heading 3 Reference'">

この手順では、Northwind データベースの **Categories** テーブルにあるデータを表示するためのグリッドを追加します。**C1FlexGrid** などの使い慣れたグリッドを使用することができます。ただし、この例では **DataGrid** コントロールを使用します。

1. DataGrid コントロールをツールボックスからウィンドウにドラッグします。
2. XAML で DataGrid の ItemsSource プロパティに連結を指定します。
   ItemsSource="{Binding [Categories], ElementName=c1DataSource1}"
3. DataGrid の AutoGenerateColumns プロパティを True に設定します。そうしないと、実行時にグリッドに列が何も含まれません。

次に、プロジェクトを実行してグリッドを表示します。

Copyright © GrapeCity Inc. All rights reserved.

手順4:プロジェクトの実行

[F5]キーを押してプロジェクトを実行します。**Northwind** データベースの **Categories** テーブルにあるデータが表示されます。

CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and se
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd
8	Seafood	Seaweed and fish

Copyright © GrapeCity Inc. All rights reserved.

DataSource for Entity Framework

DataSource for Entity Framework には、**C1DataSource** コンポーネントが含まれています。このコンポーネントを使用すると、少量のコードを記述するだけで、またはコードを追加することなく、高機能なデザインサーフェス上でデータソースを指定し、アプリケーションをすばやく簡単に開発することができます。さらに、**C1DataSource** には、デザインサーフェスから制御できる機能と同じ機能をサポートするクラスや、コードから C1DataSource を制御するための多くの追加機能が含まれています。

最初に、デザインサーフェスから **C1DataSource** コンポーネントを制御する方法について説明します。その後、実行時にこのコンポーネントを動的に制御する方法について説明します。ここに示されている機能以外にも、実行時に使用できる機能があります。クライアント側トランザクションサポートなどの他の実行時機能については、このヘルプの「プログラミングガイド」および「**リファレンス**」を参照してください。

簡単な連結

最初に、Visual Studio で新しい Windows フォームプロジェクトを作成します。

1. Visual Studio で、**[ファイル]**→**[新規作成]**→**[プロジェクト]**を選択します。
2. **[WPF フォームアプリケーション]**テンプレートを選択し、**[OK]**をクリックします。
3. 次に、Northwind データベースに基づいてエンティティデータモデルを追加します。このモデルにより、アプリケーション全体のデータが提供されます。
4. **ソリューションエクスプローラー**で、プロジェクト名を右クリックし、**[追加]**→**[新しい項目]**を選択します。
5. **[データ]**カテゴリで[ADO.NET データモデル]項目を選択し、**[追加]**をクリックします。
6. **エンティティデータモデルウィザード**を使用して、使用するデータベースを選択します。この例では、**Studio for WinForms\C1DataSource\Data** フォルダにインストールされている "NORTHWND.mdf" を使用します。
7. ソリューションエクスプローラーで、*model1.edmx* の横にあるモードを展開します。
8. *Model1.Context.tt* ファイルと *Model1.tt* ファイルを削除します。
9. モデル図を右クリックし、コンテキストメニューから**[コード生成項目の追加]**を選択します。**[コード生成項目の追加]ダイアログボックスで、[ComponentOne EF 6.x DbContext Generator]**を選択します。

メモ: **DataSource for Entity Framework** をインストールすると、**C1DataSource** がサポートする各バージョンの Visual Studio に **ComponentOne EF6.x DBContext コード生成テンプレート**が追加されます。これらのテンプレートを使用すると、作成する DBContext モデルによって **INotifyPropertyChanged** をサポートするエンティティが提供されます。

10. ここでプロジェクトをビルドすると、新しいエンティティデータモデルクラスが生成され、プロジェクト全体で使用可能になります。
11. 次に、**C1DataSource** コンポーネントをアプリケーションに追加し、それをエンティティデータモデルに接続します。
12. **C1DataSource** コンポーネントをツールボックスからフォームにドラッグします。これは非ビジュアルコンポーネントです。そのため、フォーム自体にではなく、フォーム領域の下のトレイに表示されます。
13. 新しいコンポーネントを選択し、**[表示]**→**[プロパティウィンドウ]**を選択します。
14. プロパティウィンドウで、**ContextType** プロパティを、使用するオブジェクトコンテキストのタイプに設定します。この例では、"AppName.NORTHWINDEntities" のようなオプションが1つだけドロップダウンリストに表示されます。

    この時点で、**C1DataSource** により、アプリケーションレベルのオブジェクト(**EntityDataCache**)が作成されています。このオブジェクトは、Northwind データベースを表し、アプリケーションスコープを持ちます。他のフォームに追加される **C1DataSource** オブジェクトも、この同じオブジェクトを共有します。同じアプリケーションに属する **C1DataSource** オブジェクトは、すべて同じ **ObjectContext** を共有します。

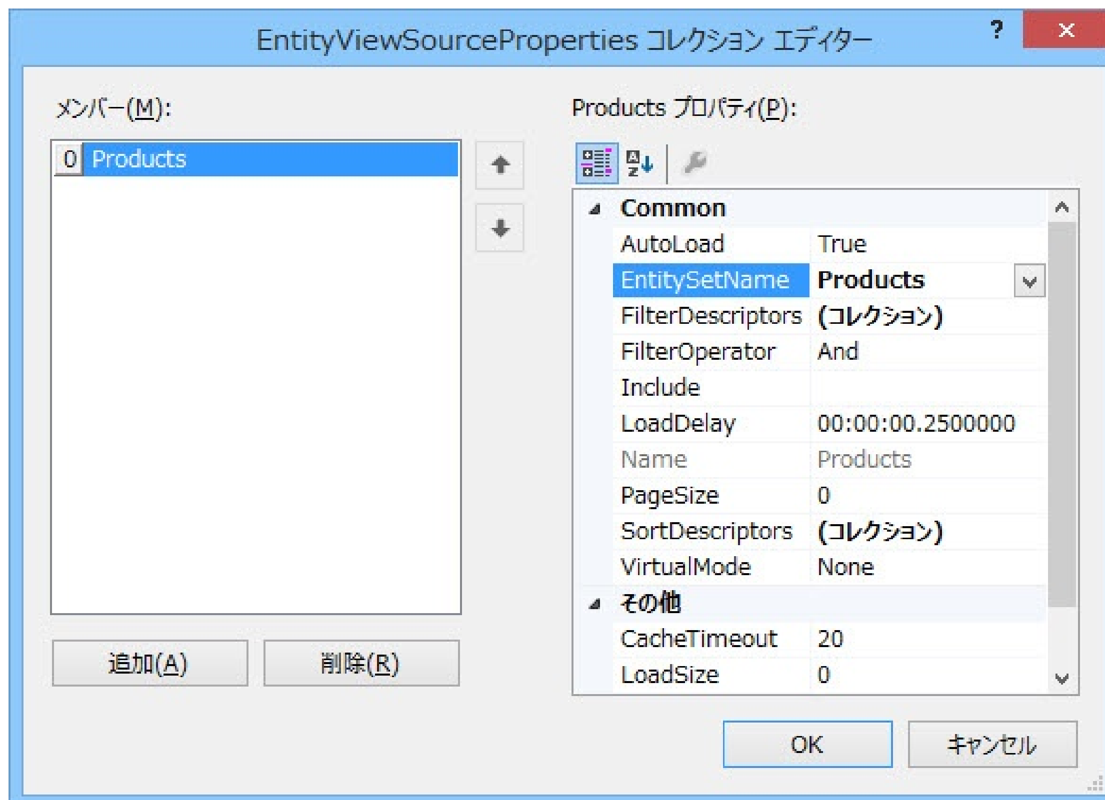
    この統合オブジェクトコンテキストは、**C1DataSource** の大きな特長の1つです。これがない場合は、アプリケーション全体で複数のオブジェクトコンテキストを作成し、それぞれを他のオブジェクトコンテキストや基底のデータベースと個別に同期させる必要があります。これは簡単な仕事ではなく、エラーがあればデータの整合性が損なわれます。統合オブジェクトコンテキストは、この仕事を透過的に実行します。データを効率よくキャッシュし、安全かつ一貫性がある方法ですべてのビューにデータを提供します。

    これで、**C1DataSource** に **ObjectContext** が作成されたので、次にその **ViewSources** コレクションを通してアプリケーションに公開するエンティティセットを指定します。ADO.NET に精通している場合は、**C1DataSource** が **DataSet** に、**ViewSources** コレクションが **DataView** オブジェクトに相当すると考えることができます。

15. **C1DataSource** のプロパティから **ViewSourcesCollection** プロパティを見つけ、そのエディタダイアログを開きます。**[追**

Copyright © GrapeCity Inc. All rights reserved.

加]をクリックし、**[EntitySetName]**ドロップダウンリストから *Products* を選択します。

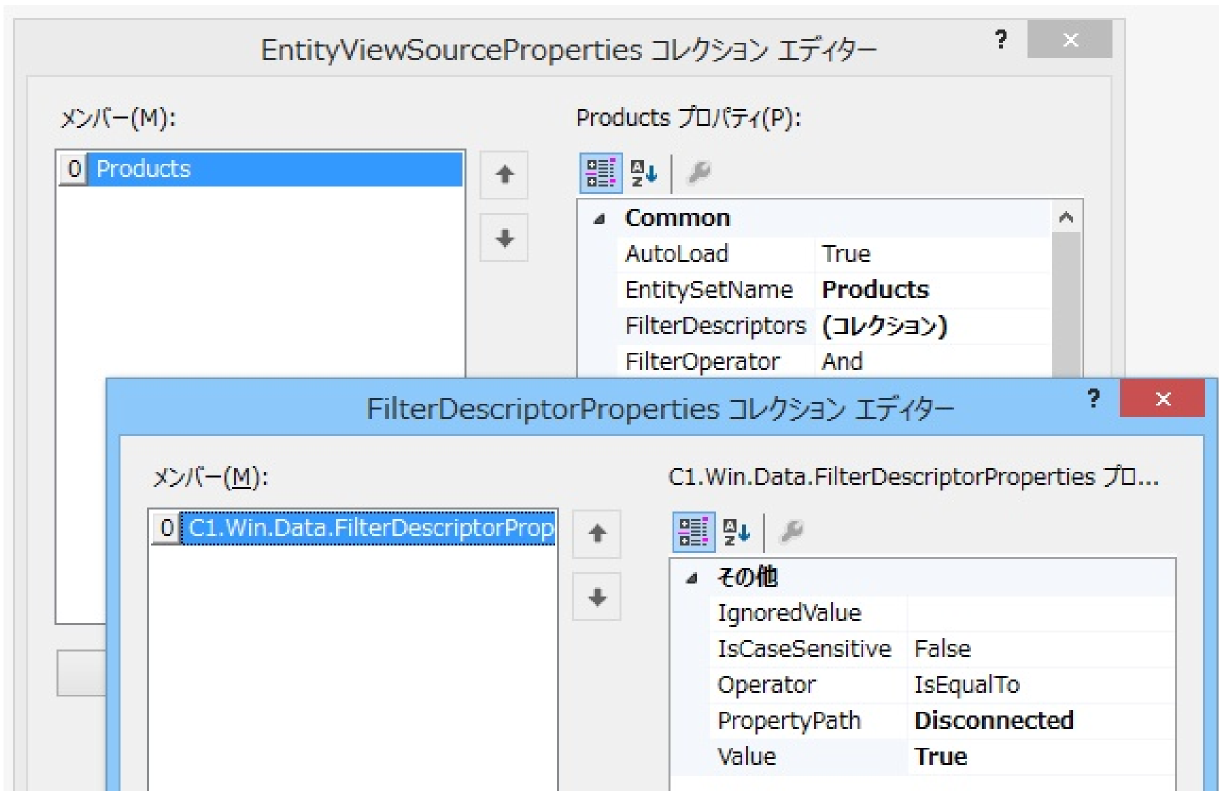


この簡単な例で実際に必要なエンティティセットは *Products* だけです。ただし、引き続き同じ方法で、この **C1DataSource** 内に **ViewSource** を作成してもかまいません。たとえば、*Categories* エンティティセットに基づいて **ViewSource** を作成すると、*Categories* と *Products* 間のマスター/詳細関係を表示するためのフォームを作成することができます。反対に、必要になるすべての **ViewSource** を1つの **C1DataSource** で定義する必要はありません。必要な **ViewSource** ごとに別の **C1DataSource** を作成することができます。必要なことは、使用する **C1DataSource** コンポーネントで同じ **ContextType** を利用することだけです。

実際のところ、クライアントに返したいデータは、常に、データベースに含まれるデータのほんの一部です。そうすることで、クライアントやネットワークに過大な負荷がかかることを回避でき、エンドユーザーが実行しているタスクに関係するデータだけを提示することができます。従来は、コード(通常は SQL クエリー)を作成してデータベースに対して実行することで、この目的を達成していました。**C1DataSource** を使用すると、サーバー側のフィルタをプロパティ設定として指定することで、コードを記述することなく、デザインサーフェスを利用してこの目的を達成することができます。

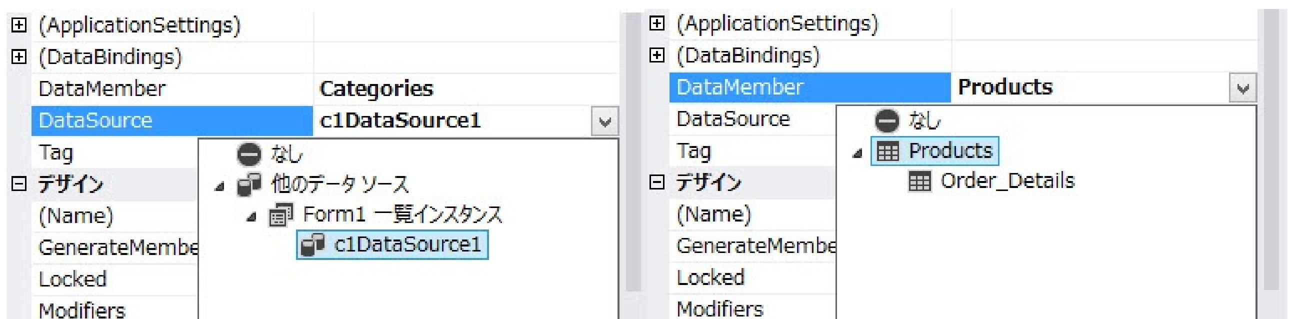
16. **ViewSourceCollection** エディタで、**FilterDescriptor** コレクションエディタを開きます。フィルタディスクリプタを追加し、サーバー側でフィルタ処理するプロパティの名前とその値を入力します。複数のプロパティをフィルタ処理する場合は、フィルタディスクリプタを追加します。

Copyright © GrapeCity Inc. All rights reserved.



同じ方法で、取得したデータをソートする **SortDescriptor** を追加することができます。

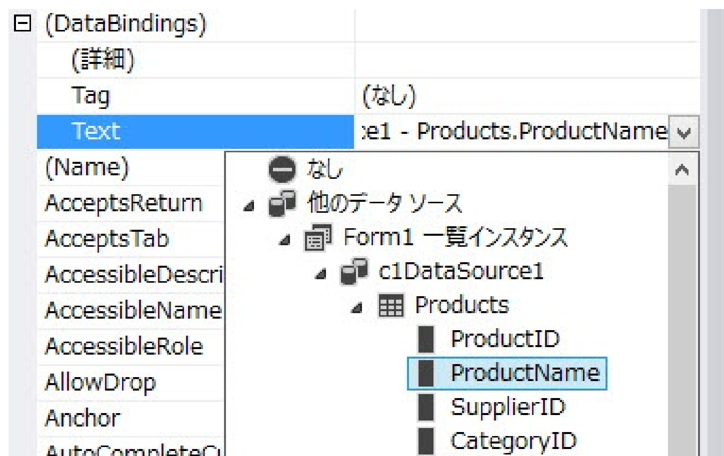
C1DataSource を設定したら、フォームにグリッドコントロールを追加します。**DataGridView**、**C1FlexGrid** など、使い慣れた任意のグリッドを使用することができます。グリッドの **C1DataSource** プロパティを **C1DataSource** の名前に設定し、その **DataMember** プロパティを Products に設定します。**C1DataSource** に名前を付けていない場合は、c1DataSource1 を指定します。実際の **DataMember** プロパティのドロップダウンリストには、**C1DataSource** に対して定義したすべての **ViewSource**（またはエンティティセット）が表示されます。



この時点で、グリッドに、*Product* タイプのすべてのフィールドに対応する列が自動的に生成されます。また、ほとんどのグリッドでは、組み込みのデザイナを使用して、これらの列とレイアウトをカスタマイズすることができます。グリッドのレイアウトが完成したら、アプリケーションを保存、ビルド、および実行します。データが自動的にロードされ、目的のとおり、項目をソート、追加、および削除できることを確認してください。2つの項目（**C1DatSource** とデータグリッド）をフォームに追加し、いくつかのプロパティを設定するだけで、作業が完了しました。1行もコードを記述する必要はありませんでした。

引き続き、このフォームに他のコントロールを追加して、それらを *Products* コレクションの特定の項目に連結することができます。その具体例として、フォームに1つの **TextBox** コントロールを追加します。次に示すように、プロパティウィンドウで **DataBindings** セクションを展開し、その **Text** プロパティを *ProductName* に連結します。

Copyright © GrapeCity Inc. All rights reserved.



アプリケーションを再度保存、ビルド、および実行します。今回は、グリッドで現在選択されている製品の名前が、フォームに追加した **TextBox** に表示されます。一方のコントロールで製品名を編集すると、その変更がもう一方のコントロールにすぐに反映されます。

サーバー側のフィルタ処理

一般にサーバーからクライアントに返されるデータは制限する方が望ましいこと、また C1DataSource では **FilterDescriptor コレクションエディタ**を使用してこれを簡単に実行できることを既に説明しました。次に、サーバー側のフィルタ処理を実行する手段をエンドユーザーに提供する方法について説明します。

他の GUI コントロールでもかまいませんが、たとえば、ユーザーがコンボボックスから *Product Category* を選択し、それによってサーバーから新しいデータが **DataGrid** にロードされるとします。

サーバー側のフィルタ処理を実装するには、次の手順に従います。

1. 「簡単な連結」で使用したプロジェクトに、C1DataSource コンポーネントを含む新しいフォームを追加します。このフォームをプロジェクトのスタートアップフォームにすることで、プロジェクトの実行にかかる時間を短縮することができます。
2. 前と同様に C1DataSource を確立します。今回は、Categories と Products の2つのビューソースを定義します。ViewSourceCollection で、[追加]ボタンを2回クリックします。EntityViewSourceProperties.EntitySetName プロパティの横で、最初のメンバ(0)には「Categories」を、2番目のメンバ(1)には「Products」を入力します。
3. Value は空のままにします。これは、コンボボックスの選択変更イベントに応答するコードで設定するからです。
4. 前に示したように、ItemsSource="{Binding ElementName=c1DataSource1, Path=Products}" を使用してグリッドを連結します。
5. XAML でグリッドの AutoGenerateColumns プロパティを True に設定します。そうしないと、実行時にグリッドに列が何も含まれません。
   <DataGrid AutoGenerateColumns="True"...
6. コンボボックスには、次の連結を使用します。
   ItemsSource="Binding ElementName=c1DataSource1, Path=Categories}"
   DisplayMemberPath="CategoryName"
   グリッドや他のコントロールと同様に、[プロパティ]ウィンドウの連結デザイナを使用して、リストから連結を選択できます。または、XAML で連結を直接入力できます。
7. 最後に、コンボボックスの SelectionChanged イベントを処理する次のコードをフォームに追加します。

Visual Basic

```
Private Sub comboBox1_SelectionChanged(sender As System.Object, e As
System.Windows.Controls.SelectionChangedEventArgs)
     c1DataSource1.ViewSources("Products").FilterDescriptors(0).Value =
         CType(comboBox1.SelectedValue, Category).CategoryID
     c1DataSource1.ViewSources("Products").Load()
End Sub
```


Copyright © GrapeCity Inc. All rights reserved.

C#

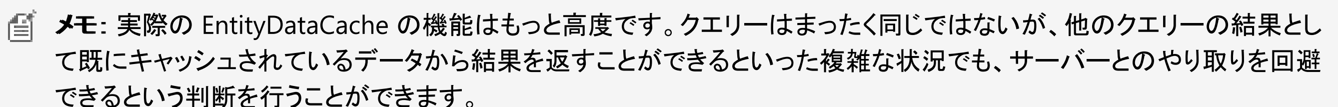
```
private void comboBox1_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    c1DataSource1.ViewSources["Products"].FilterDescriptors[0].Value =
        ((Category)comboBox1.SelectedItem).CategoryID;
c1DataSource1.ViewSources["Products"].Load();
}
```

8. アプリケーションを保存、ビルド、および実行します。コンボボックスでカテゴリを選択します。そのカテゴリに関連する製品がグリッドに表示されます。前回とまったく同様に、グリッド内のデータを編集できます。

クライアントデータキャッシュの能力

「サーバー側のフィルタ処理」の例は、**C1DataSource** により、Entity Framework をこれまでより簡単かつ便利にアプリケーションで使用できるようになったことを示しています。これは、**C1DataSource** の主要な機能であるクライアント側のデータキャッシュによって実現されました。また、いくつかの重要な機能が強化され、アプリケーションコードをはるかに容易に記述できるようになりました。

最初に、「サーバー側のフィルタ処理」の例で見ることができるパフォーマンスの向上について説明します。ユーザーがカテゴリを切り替えると、そのカテゴリに関連する製品がグリッドにロードされます。あるカテゴリを初めて選択すると、関連データが取得されるまでにわずかな遅延が発生します。その後、同じカテゴリを選択すると、データはほぼ即座に取得されます。なぜでしょうか。データが、サーバーからではなく、クライアントメモリのデータキャッシュ（EntityDataCache）から取得されるためです。



メモ：実際の EntityDataCache の機能はもっと高度です。クエリーはまったく同じではないが、他のクエリーの結果として既にキャッシュされているデータから結果を返すことができるといった複雑な状況でも、サーバーとのやり取りを回避できるという判断を行うことができます。

単一のマシンで作業し、ネットワークとのやり取りが必要ないなら、このパフォーマンスの向上は実感されないかもしれません。しかし、現実には、ユーザーアクションのたびにサーバーを呼び出すような反応が遅いアプリケーションと、遅延のない快適なインタフェースとの差は歴然です。

2つめの重要な改良点はメモリ管理です。これまでの説明や実際に目にしたことから、EntityDataCache は、存在する間ずっとデータを蓄積し続けているように思われるかもしれません。もしそうなら、すぐに深刻なパフォーマンスの低下を見ることになるはずです。実際の EntityDataCache は、保存されているデータを常に監視し、データが不要になるとそれを解放して、同時にセルフクレンジング処理を実行しています。これらの処理はすべて、何もコードを追加しなくても実行されます。また、さらに重要なことは、データの整合性が常に維持されるという点です。他のデータによって必要とされるデータは解放されません。また、何らかの変更が加えられたデータは、保存されるまで解放されません。この処理はパフォーマンスにも影響します。メモリ内の不要なオブジェクトを破棄すればパフォーマンスが向上し、反対に、古くなったデータがメモリに大量に保存されていればパフォーマンスが低下します。

C1DataSource では、クライアントキャッシュのお陰で複数のデータコンテキストを作成する必要がなくなり、コンテキスト管理がシンプルになったということを前に説明しました。ここでは、EntityDataCache の詳細と、その知識をさらに活用する方法について説明します。

EntityDataCache は本質的にコンテキストです。**C1DataSource** の名前空間において、キャッシュは **C1.Data.Entities.EntityClientCache** クラスであり、その **ObjectContext** プロパティを通して **ObjectContext** に1対1で対応します。**C1DataSource** コンポーネントを使用している場合は、キャッシュとその基底の **ObjectContext** がどちらも自動的に作成されます。ただし、必要な場合は、コードでこれらを明示的に作成し、**C1DataSource.ClientCache** プロパティを設定することができます。

クライアント側のキャッシュによってどのようにアプリケーションコードが簡略化されるかを見るために、変更データを保存する機能を Server-Side Filtering プロジェクトに追加してみます。

1. ボタン **btnSaveChanges** をフォームに追加し、このコードのハンドラを追加します。

Copyright © GrapeCity Inc. All rights reserved.

VisualBasic

```
private void btnSaveChanges_Click(object sender, EventArgs e)
   {
     c1DataSource1.ClientCache.SaveChanges();
   }
```

C#

```
Private Sub btnSaveChanges_Click(sender As System.Object, e As System.EventArgs)
    C1DataSource1.ClientCache.SaveChanges()
End Sub
```

2. アプリケーションを保存、ビルド、および実行します。
   - カテゴリを選択し、グリッドで製品情報に変更を加えます。
   - 2つ目のカテゴリ(必要に応じて3つ目も)を選択し、グリッドで製品詳細に再度変更を加えます。
   - 追加したボタンをクリックしてアプリケーションを閉じます。再度アプリケーションを開き、前の手順と同じカテゴリを選択します。

変更がどのように保存されているかを確認します。**C1DataSource** では、EntityDataCache により、別のカテゴリを選択するたびに変更を保存しなくても、複数のカテゴリの製品の詳細を変更することができます。**C1DataSource** なしでこのような目的を達成するには、大量のコードを記述するか、複数のカテゴリのエンティティ(製品詳細)を同じコンテキストに保存する必要があります。メモリが解放されないため、メモリが浪費され、メモリリークの原因になります。**C1DataSource** では、このすべての処理が簡略化されると共に、メモリ消費量とパフォーマンスが最適化されます。

クライアント側のキャッシュは、ほかにもクライアント側のクエリーなどの **C1DataSource** の重要な機能を提供しています。特に、ライブビューは重要です。ライブビュー は、複雑なアプリケーションコードの大部分を単純なデータ連結に置き換えることができる機能です。これについては、後のセクションで説明します。

マスター/詳細連結

「サーバー側のフィルタ処理」の例で紹介したように、C1DataSource はマスター/詳細連結をサポートしています。大規模なデータセットでは、サーバー側のフィルタ処理が最適なソリューションですが、規模の小さなデータセットでは、クライアント側のフィルタ処理も同様に効率的です。次のシナリオでは、前の例で使用したコンボボックスではなく、グリッドを使用してクライアント側のマスター/詳細連結を実行し、カテゴリを選択します。

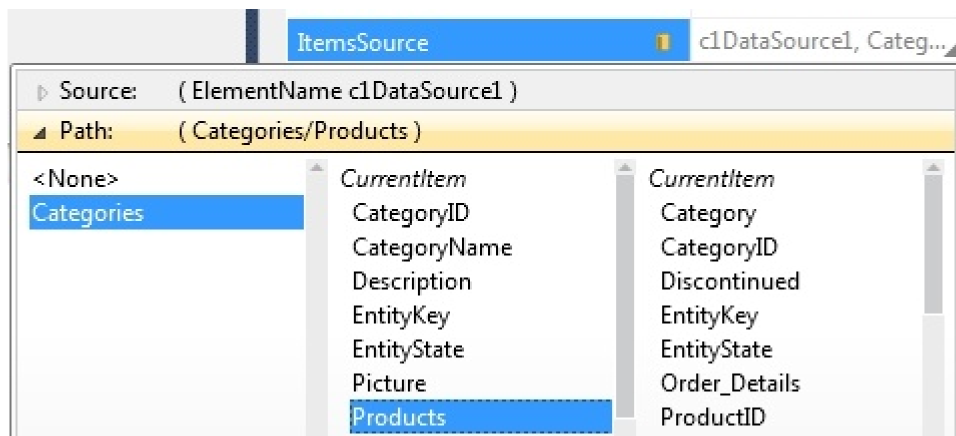
マスター/詳細連結を実装するには、次の手順に従います。

1. 「サーバー側のフィルタ処理」の説明で作成したプロジェクトを使用します。前と同様に、同じ ObjectContextType を使用して、C1DataSource コンポーネントを含むフォームを新しく追加し、Categories に基づく ViewSource を作成します。このフォームをスタートアップフォームにすることで、プロジェクトの実行にかかる時間を短縮することができます。
2. 次に、フォームにマスターグリッドを追加し、それを Categories に連結します。

   ItemsSource="{Binding ElementName=c1DataSource1, Path=Categories}"

3. 次に、フォームで、設定したグリッドの下に2つ目のグリッドを追加します。その DataSource も C1DataSource に設定しますが、DataMember プロパティは Categories の下にある Products ノードに設定します(次の図を参照)。

Copyright © GrapeCity Inc. All rights reserved.



または、XAML で直接入力できます。

```
ItemsSource="{Binding ElementName=c1DataSource1, Path=Categories/Products}"
```

4. アプリケーションを保存、ビルド、および実行します。

連結で "/" を使用した Path=Categories/Products のような形式の **Path** パラメータを指定すると、現在選択されているカテゴリに属する製品がグリッドに自動的に表示されます。これを単一のマシンで実行した場合、マスターグリッドで新しいカテゴリを選択してから関連する製品が詳細グリッドに表示されるまでに、目立った遅延は見られないでしょう。バックグラウンドでは、**C1DataSource** は Entity Framework の "暗黙の遅延ロード" 機能を利用しており、製品は新しいカテゴリが選択されたときにのみ収集されます。多くのシナリオでは、これでまったく問題ありません。ただし、このセクションの冒頭で、特に規模の小さなデータセットのマスター/詳細関係について言及しました。その場合は、フォームをロードするときにすべてのカテゴリのすべての製品をフェッチした方がよく、これで、単一のマシンでもネットワークにアクセスする場合でも瞬時に表示が行われます。それには、**ViewSourceCollection** エディタを開き、*Categories* ビューソースの **Include** プロパティに「*Products*」と入力します。

大規模なデータセット：ページング

クライアントに大量のデータを一度に取り込むことなく表示するには、従来、ページングが使用されてきました。インタフェースが複雑になり、ユーザーにとってもあまり便利ではないため、ページングは理想的なソリューションと言えません。ただし、ページングの方が好ましいアプリケーションもあります。このような場合のために、C1DataSource はページングをサポートしています。しかも、これまでようにデータ変更が制限されることはありません。ユーザーは、1つのセッションで複数のページを変更できます。次のページに進む前に、前のページの変更をデータベースに送信する必要もありません。これは、Microsoft RIA サービスの DomainDataSource で実装されているようなページングよりはるかに優れています。

> **メモ**: **EF DataSource** には、ページングの欠点を補うソリューションが用意されています。これについては、このマニュアルの「仮想モード」で説明します。

ページングを実装するには、次の手順に従います。

「マスター/詳細連結」の説明で作成したプロジェクトを使用します。前と同様に、同じ **ObjectContextType** を使用して、**C1DataSource** コンポーネントを含む新しいフォームを追加します。このフォームをスタートアップフォームにすることで、プロジェクトの実行にかかる時間を短縮することができます。

1. ViewSourceCollection エディタで ViewSource を作成します。EntitySetName には「Orders」と入力します。
2. ここでは、ページングを有効にするために、PageSize プロパティを 10 に設定します。このプロパティには、妥当な値を任意に選択することができます。この値は、ページに表示されるデータ行の数を決定します。
3. 次に、フォームにグリッドを追加し、前と同じ方法でそれを Orders に連結します。

```
ItemsSource="{Binding ElementName=c1DataSource1, Path=Orders}"
```

4. XAML でグリッドの AutoGenerateColumns プロパティを True に設定します。
5. ユーザーがページ間を移動できるようにし、現在のページとページ数を表示するには、さらにいくつかのコントロールを

Copyright © GrapeCity Inc. All rights reserved.

追加する必要があります。たとえば、次の XAML を使用して、2つのボタンと1つのラベルを追加します。

XAML

```
<StackPanel Orientation="Horizontal" Grid.Row="1" Margin="2">
    <Button Padding="10,0,10,0" Margin="1" Content="&lt;"
Click="MoveToPrevPage"/>
    <TextBlock VerticalAlignment="Center" Text="Page: "/>
    <TextBlock VerticalAlignment="Center" x:Name="pageInfo"/>
    <Button Padding="10,0,10,0" Margin="1"  Content="&gt;"
Click="MoveToNextPage"/>
</StackPanel>
```

6. 次のコードを追加します。PropertyChanged イベントの RefreshPageInfo ハンドラは、現在のページ番号とページ数を表示するために使用され、ボタンの Click イベントのハンドラは、次/前のページへの移動に使用されます。

Visual Basic

```
Imports C1.Data.DataSource
Public Class Paging
     Private _view As ClientCollectionView
     Public Sub New()
         InitializeComponent()
         _view = C1DataSource1("Orders")
         RefreshPageInfo()
         AddHandler _view.PropertyChanged, AddressOf RefreshPageInfo
     End Sub
     Private Sub RefreshPageInfo()
         pageInfo.Text = String.Format("{0} / {1}", _view.PageIndex + 1,
_view.PageCount)
     End Sub
     Private Sub btnPrevPage_Click(sender As System.Object, e As
System.EventArgs) Handles btnPrevPage.Click
         _view.MoveToPreviousPage()
     End Sub
     Private Sub btnNextPage_Click(sender As System.Object, e As
System.EventArgs) Handles btnNextPage.Click
         _view.MoveToNextPage()
     End Sub
End Class
```

C#

```
using C1.Data.DataSource;

namespace TutorialsWPF
{
    public partial class Paging : Window
    {
        ClientCollectionView _view;

        public Paging()
        {
            InitializeComponent();
```


Copyright © GrapeCity Inc. All rights reserved.

```
            _view = c1DataSource1["Orders"];

            RefreshPageInfo();
            _view.PropertyChanged += delegate { RefreshPageInfo(); };
        }

        private void RefreshPageInfo()
        {
            labelPage.Text = string.Format("{0} / {1}",
         _view.PageIndex + 1, _view.PageCount);
        }

        private void MoveToPrevPage(object sender, RoutedEventArgs e)
        {
            _view.MoveToPreviousPage();
        }

        private void MoveToNextPage(object sender, RoutedEventArgs e)
        {
            _view.MoveToNextPage();
        }
    }
}
```

7. アプリケーションを保存、ビルド、および実行します。Orders 間をページ移動します。ページ間を移動しながら、グリッドでいくつかのデータを変更してみます。1つのページでデータを変更してから別のページに移動し、そのページでもデータを変更します。C1DataSource では、変更したページをデータベースに保存しなくても、次のページに移動できることに注目してください。これは、Microsoft RIA サービスの DomainDataSource で実装されているようなページングより優れた重要な機能です。Microsoft RIA サービスの DomainDataSource で実装されているページングは Silverlight 用ですが、EF DataSource では、3つすべてのプラットフォーム(WinForms、WPF、Silverlight)で、他の機能と同様にこのページングの実装がサポートされています。
8. また、いくつかの注文を削除してみます。この操作も制限なく実行することができます。さらに、現在のページが自動的に更新されて、ページ内の行数が維持されます。
9. 以前の同様に次のコードを使用して[変更の保存]ボタンを追加すると、そのボタンを使用して、複数のページに加えた変更を一度に保存することができます。

Visual Basic

```
Private Sub btnSaveChanges_Click(sender As System.Object, e As
System.Windows.RoutedEventArgs)
     c1DataSource1.ClientCache.SaveChanges()
End Sub
```

C#

```
private void btnSaveChanges_Click(object sender, RoutedEventArgs e)
{
    c1DataSource1.ClientCache.SaveChanges();
}
```

これらの機能はすべて、データ変更に制限のないページングを実装するために求められるものです。ただし、この機能を実装することは簡単ではありません。たとえば、あるページで変更されたデータが他のページの表示に影響を与えるようなケースをすべて想定する必要があります。このため、ページングでは、データの変更が許可されていても、それに厳しい制限が課されることが普通です。たとえば、Microsoft DomainDataSource では、他のページに移動する前に、すべての変更を保存する必

Copyright © GrapeCity Inc. All rights reserved.

要があります。**EF DataSource** がサポートしているページングでは、データを無制限に変更することができます。

他の多くの機能と同様に、変更に制限のないページングは、クライアント側のキャッシュによって実現されています(「クライアントトデータキャッシュの能力」を参照)。つまり、**EF DataSource** に実装されているページングは、パフォーマンスとメモリ消費の両面で最適化されています。この最適化は、キャッシュによって実現されています。最近開いたページがメモリに保存されるので、通常は、同じページを再度開くと瞬時に表示されます。また、メモリリソースが管理され(古いページは必要に応じて解放されます)、メモリリークが防止されます。

大規模なデータセット:仮想モード

「大規模なデータセット:ページング」で説明したように、**Entity Framework DataSource**(**EF DataSource**)には、大規模なデータや大量の行を処理するためのソリューションとして、ページングよりはるかに優れた機能があります。

数千から数百万もの行を持つ大規模なデータセットを問題なく使用できるとしたらどうでしょうか。ページングを使用せず、コードを変更することもなく、1つのブール値プロパティを設定するだけで、データセットが小規模な場合と同じコントロールを使用して巨大なデータセットを表示できるとしたらどうでしょうか。**EF DataSource** は、魔法のような **VirtualMode** プロパティによってこれを実現します。

仮想モードを実装するには、次の手順に従います。

1. 「大規模なデータセット:ページング」の説明で作成したプロジェクトを使用します。前と同様に、同じ ObjectContextType を使用して、C1DataSource コンポーネントを含む新しいフォームを追加します。このフォームをスタートアップフォームにすることで、プロジェクトの実行にかかる時間を短縮することができます。
2. ViewSourceCollection エディタで ViewSource を作成します。サンプルデータベースの最大のテーブル Order_Details を使用します。
3. VirtualMode プロパティを Managed に設定します。もう1つの値として Unmanaged がありますが、これは使用時に注意が必要な高度なオプションです。必要な場合にのみ使用してください。Managed オプションを設定すると、サーバーからのデータ取得がグリッドコントロールによって管理されます。Managed オプションを使用して、EF DataSource は、すべての主要な Microsoft および ComponentOne グリッドコントロール(C1FlexGrid、C1DataGrid、および Microsoft DataGrid for WPF)をサポートします。EF DataSource のパフォーマンスは、これらのグリッドコントロールについて最適化されています。Unmanaged オプションに設定した場合、仮想モードは特定のコントロールに基づいて動作するのではなく、任意の連結コントロールと共に動作しますが、こちらで説明されているように、いくつかの制限が適用されます。

4. デザイナにグリッドを追加し、前と同じ方法でそれを Order_Details に連結します。

    ItemsSource="{Binding ElementName=c1DataSource1, Path=Order_Details}"

5. XAML でグリッドの AutoGenerateColumns プロパティを True に設定します。

    Managed オプションを選択したので、データを操作するグリッドコントロールを指定する必要があります。**EF DataSource** は、添付プロパティ C1DataSource.ControlHandler を定義します。これは、C1DataSource|keyword=C1DataSource クラス に連結されるとコントロールの動作に影響を及ぼすプロパティを持つオブジェクトです。C1DataSource.ControlHandler には、**VirtualMode** ブール値プロパティがあります。このプロパティは、このコントロールを *Managed* 仮想モードのメインの "制御中" コントロールとしてマークします。

6. XAML で DataGrid マークアップ内に次のマークアップを追加します(追加されたマークアップは影付きで示します)。

    XAML
    ```
    <DataGrid AutoGenerateColumns="True"  Name="dataGrid1"
           ItemsSource="{Binding ElementName=c1DataSource1, Path=Order_Details}">
                <c1:C1DataSource.ControHandler>
                      <c1:ControlHandler VirtualMode="True"/>
                </c1:C1DataSource.ControHandler>
          </DataGrid>
    ```
7. アプリケーションを保存、ビルド、および実行します。

Copyright © GrapeCity Inc. All rights reserved.

これほど優れたグリッドはほかにありません。このグリッドで、適切に移動およびスクロールしたり、行データを変更することができます。外観も動作も従来のデータグリッドと同様で、それがまさに重要な点です。難点の多いページングもコードの記述もまったくなく、大規模なデータセットを使用できるようになりました。さらに、**DataSource** プロパティを持つ任意の GUI コントロールを使用できるという利点があります。この例では、比較的小さなサイズのデータセットを使用しましたが、仮想モードで実行される **C1DataSource** は、はるかに大きなデータセットでも同様に応答します。その動作が行数に左右されることはありません。その点について確認するために、製品と共にインストールされている **OrdersDemo** サンプルを見てみます。このサンプルでは、より大きなデータベースが使用されており、その行数は、この例で使用されているデータセットより約 65,000 行多くなっています。ただし、応答速度はどちらも変わりません。このように、C1DataSource の動作は、データセットの行数に左右されません。

なぜ、このようなことが可能なのでしょうか。その動作はページングとかなり似ていますが、内部的な動作は GUI コントロールからは見えず、隠れて行われているページングと言えます。GUI コントロールには、データがクライアントにフェッチされ、使用する準備ができているかのように示されます。GUI コントロールがデータを要求すると、**C1DataSource** または **ClientViewSource**（**C1DataSource** コントロールがなく、コードで **ClientViewSource** が使用されている場合）は、メモリから、つまりすべての機能に使用されているクライアント側のキャッシュから、データを取得して提供できるかどうかを最初に確認します。メモリにデータが見つからない場合は、要求されたデータをサーバーから透過的にフェッチします。クライアント側のキャッシュを使用する他の機能と同様に、**C1DataSource** は、フェッチしたデータを無制限には保存しません。無制限に保存すると、メモリリークになってしまいます。C1DataSource は、GUI コントロールに提供する必要があるデータを認識しています。変更されたり、変更された部分に関係するために、保存しておく必要があるデータも認識しています。また、不要になった古いデータを必要に応じて解放します。コードを記述する必要はなく、任意の GUI コントロールを使用できます。

グリッドの自動ルックアップ列

よくあるデータ連結のシナリオとして、データクラスが他のデータクラスへの参照を持つ場合があります。たとえば、**Product** オブジェクトが **Category** オブジェクトや **Supplier** オブジェクトへの参照を持つ場合です。

ADO.NET では、通常、参照は他のテーブルにマップされる外部キー（**Product.CategoryID** や **Product.SupplierID**）として表されます。

Entity Framework でもキー列を取得しますが、実際のオブジェクトも取得します。このため、**Product.CategoryID**（通常は整数）と **Product.Category**（実際の Category オブジェクト）があります。

グリッドに外部キーを表示しても、あまり役に立ちません。これは、Category（カテゴリ）12 が "Dairy Products" であることや、Supplier（仕入れ先）15 が "ACME Imports" であることをユーザーが知っていることはないためです。これらのキーの編集をユーザーに許可することはさらに問題です。一般に、この問題に対処するには、関連するエンティティ列を連結グリッドから削除します。または、それらの列をカスタム列に置き換え、コンボボックスを使用して値を編集できるようにします（"ルックアップ"）。関連する値（**Category.Name** や **Supplier.CompanyName**）がコンボボックスに表示され、コンボボックスの内容がグリッドに表示されている値と同期するように、コンボボックスを関連するテーブルに連結し、そのプロパティを設定する必要があります。この作業はそれほど難しくありませんが、エラーの元になりやすく面倒なタスクなので、プロジェクトの作成と維持が難しくなります。

C1DataSource は、開発者に代わって、この面倒な作業を実行します。コンボボックスルックアップが表示されるように、関連するエンティティ列を自動的に変更します。この作業は、サポートされているいくつかのタイプのデータグリッドで実行可能です。現在サポートされている WinForms グリッドは C1FlexGrid と Microsoft DataGridView です。ここでは、**C1FlexGrid** でこれを実行する方法について説明します。

C1DataSource には、**ControlHandler** という*拡張*プロパティがあります。**C1DataSource** を含むフォームに **C1FlexGrid** コントロールを配置すると、グリッドに **ControlHandler** プロパティが追加されます。ControlHandler は、この時点では1つのブール値プロパティ **AutoLookup** を含むオブジェクトです。このプロパティを *True* に設定すると、**C1DataSource** は、他のエンティティへの参照を含むグリッド列にルックアップコンボボックスが表示されるように設定します。

この動作を確認するには、次の手順に従います。

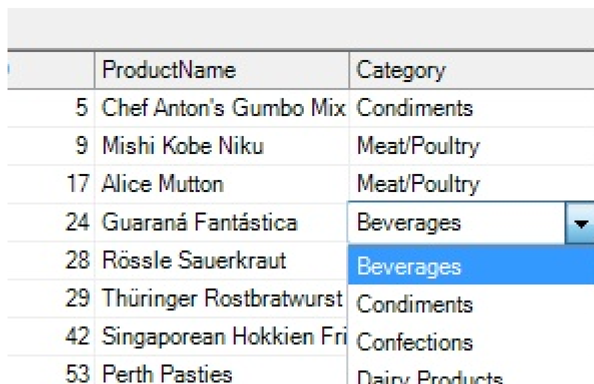
1. 「簡単な連結」で使用したプロジェクトを使用します。前と同様に、同じ **ContextType** を使用して、**C1DataSource** コンポーネントを含む新しいフォームを追加します。
2. **ViewSourceCollection** エディタで **ViewSource** を作成します。**EntitySetName** には「*Products*」と入力します。
3. **C1FlexGrid** をフォームに追加し、その **C1DataSource** プロパティを **C1DataSource** に、**DataMember** プロパティを *Products* に設定します。
4. アプリケーションを保存、ビルド、および実行します。次のようになります。

Copyright © GrapeCity Inc. All rights reserved.

Product ID ▲	Product Name	Category
5	Chef Anton's Gumbo Mix	Category : 2
9	Mishi Kobe Niku	Category : 6
17	Alice Mutton	Category : 6
24	Guaraná Fantástica	Category : 1
28	Rössle Sauerkraut	Category : 7
29	Thüringer Rostbratwurst	Category : 6
42	Singaporean Hokkien Fried Me	Category : 5
53	Perth Pasties	Category : 6

このように、**Category** 列と **Supplier** 列はまったく有効でありません。これらの列を削除したり、新しい列を作成するコードを記述してグリッドをカスタマイズすることができますが、もっと簡単な方法があります。

5. デザイナでグリッドを選択し、次の図に示されているように、プロパティウィンドウで「**ControlHandler on c1DataSource1**」という名前のプロパティを見つけます。
6. これは拡張プロパティで、フォーム内に **C1DataSource** コンポーネントが存在する場合にのみ使用できることに注意してください。**AutoLookup** プロパティを *True* に設定します。
7. 作業が完了したら、プロジェクトを再度実行して、**Category** 列と **Supplier** 列を確認します。グリッドに、汎用文字列ではなく、カテゴリ名と仕入れ先の会社名が表示されていることに注目してください。また、ドロップダウンリストから値を選択して、製品のカテゴリや仕入れ先を編集したり、自動検索機能を使用して値を入力することもできます。



8. 最後に、列ヘッダーをクリックして、**Category** や **Supplier** でグリッドをソートします。グリッドに表示されている値に基づいてソートが実行されることに注目してください。これは誰もが求める機能ですが、通常のデータ連結を使用して実装することは容易でありません。

コンボボックスに表示される文字列値(名前)は、次のルールに従って決定されます。

1. エンティティクラスで **ToString** メソッドをオーバーライドしている場合、エンティティの文字列表現は、オーバーライドされた **ToString** メソッドを使用して取得されます。このメソッドは、エンティティを一意に表現する文字列を返す必要があります。たとえば、**CompanyName** 列の内容、または **FirstName**、**LastName**、**EmployeeID** の組み合わせなどが考えられます。パーシャルクラスを使用して簡単かつ柔軟に実装できるため、これは推奨の方法です(エンティティモデルが再生成されても、実装が影響を受けない)。
2. エンティティクラスで **ToString** メソッドがオーバーライドされていないが、いずれかのプロパティに **DefaultProperty** 属性が含まれる場合は、そのプロパティがエンティティの文字列表現として使用されます。
3. エンティティクラスで **ToString** メソッドがオーバーライドされず、どのプロパティにも **DefaultProperty** 属性が含まれていない場合は、名前に文字列 "Name" または "Description" が含まれている最初の列がエンティティの文字列表現として使用されます。
4. 上のどのルールにも該当しない場合、そのエンティティタイプにはルックアップが作成されません。

ビューのカスタマイズ

Copyright © GrapeCity Inc. All rights reserved.

さまざまな状況で、データベースから提供されるテーブルやビューに直接対応していないカスタムビューを使用したいことがあります。LINQ は、このような状況に最適なツールです。柔軟、簡潔、かつ効率的なクエリーステートメントと任意の言語を使用して、生データを変換できます。**Entity Framework DataSource（EF DataSource）**では、LINQ クエリーステートメントを*動的*に実行できるので、LINQ の機能がさらに強力になります。このため、この LINQ 実装は LiveLinq と呼ばれます。LiveLinq の機能については、「ライブビュー」で説明します。ここでは、完全な更新可能性と連結可能性を損なうことなく、LINQ 演算子を使用して自由にビューを変更し、形成できるということだけを伝えておきます。

さっそく、LINQ 演算子の1つである **Select**（"プロジェクション" とも呼ばれます）を使用して、ビューのフィールド（プロパティ）をカスタマイズしてみます。

ビューをカスタマイズするには、次の手順に従います。

1. 「大規模なデータセット：ページング」の説明で作成したプロジェクトを使用します。前と同様に、同じ ObjectContextType を使用して、C1DataSource コンポーネントを含む新しいフォームを追加します。このフォームをスタートアップフォームにすることで、プロジェクトの実行にかかる時間を短縮することができます。
2. ViewSourceCollection エディタで ViewSource を作成します。サンプルデータベースの Products テーブルを使用します。
3. ウィンドウデザイナにグリッドを追加し、前と同様に、その AutoGenerateColumns プロパティを True に設定します。ただし、今回は、コードでカスタムビューを作成するため、XAML でグリッドを C1DataSource に連結するのではなく、コードで連結します。
4. 次のコードを使用して、カスタムライブビューを作成し、ビューにグリッドを連結します。

Visual Basic

```
dataGrid1.ItemsSource = _
    (From p In c1DataSource1("Products").AsLive(Of Product)()
      Select New With
      {
           p.ProductID,
           p.ProductName,
           p.CategoryID,
           p.Category.CategoryName,
           p.SupplierID,
           .Supplier = p.Supplier.CompanyName,
           p.UnitPrice,
           p.QuantityPerUnit,
           p.UnitsInStock,
           p.UnitsOnOrder
       }).AsDynamic()
```

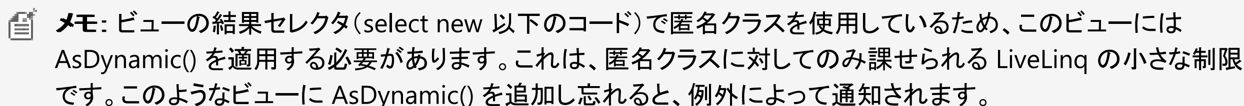
C#

```
dataGrid1.ItemsSource =
    (from p in c1DataSource1["Products"].AsLive<Product>()
     select new
     {
         p.ProductID,
         p.ProductName,
         p.CategoryID,
         CategoryName = p.Category.CategoryName,
         p.SupplierID,
         Supplier = p.Supplier.CompanyName,
         p.UnitPrice,
         p.QuantityPerUnit,
         p.UnitsInStock,
         p.UnitsOnOrder
```


Copyright © GrapeCity Inc. All rights reserved.

```
}).AsDynamic();
```

ここで、c1DataSource1["Products"] は1つの **ClientCollectionView** オブジェクトです。これは、デザイナで設定したビューソースに基づいて作成されるビューです(コードでビューソース自体にアクセスする必要がある場合は、c1DataSource.ViewSources["Products"] としてアクセスすることもできます)。**AsLiveT_Method** メソッド呼び出しは、ビューの項目タイプ(*Product*)を指定するために必要です。これにより、この項目に LiveLinq 演算子を適用できます。c1DataSource1["Products"].AsLive<Product>() の結果は1つの View<Product> です。**C1.LiveLinq.LiveViews.View** は、クライアント側ライブビューで使用される LiveLinq のメインクラスです。ライブビューに適用される LINQ 演算子は、ライブビューの更新可能性と連結可能性を維持します。これらは、通常と同じ LINQ 演算子ですが、ライブビューに適用されることで、データ連結アプリケーションに極めて重要な追加機能を提供します。



メモ: ビューの結果セレクタ(select new 以下のコード)で匿名クラスを使用しているため、このビューには AsDynamic() を適用する必要があります。これは、匿名クラスに対してのみ課せられる LiveLinq の小さな制限です。このようなビューに AsDynamic() を追加し忘れると、例外によって通知されます。

5. アプリケーションを保存、ビルド、および実行します。

これで、この LiveLinq ビューの 'select' 句で定義した列がグリッドに表示されます。すべての列が変更可能であることにも注目してください。これは大したことに思えないかもしれませんが、ここで示すように行の追加や削除もサポートされる場合は特に、カスタマイズされたビューに独自に実装することが難しい重要な機能です。行の削除を試すには、行を選択し、**Delete** キーを押すだけです。行の追加を試すには、次のコードを実行するボタンをウィンドウに追加する必要があります。

Visual Basic

```
Dim newItem = CType(dataGrid1.ItemsSource,
System.ComponentModel.IEditableCollectionView).AddNew()
dataGrid1.ScrollIntoView(dataGrid1.Rows.Count - 1, 0)
```

C#

```
object newItem =
((System.ComponentModel.IEditableCollectionView)dataGrid1.ItemsSource).AddNew();
dataGrid1.ScrollIntoView(newItem);
```

これが必要なのは、単に Microsoft WPF DataGrid には新しい行を追加するためのインタフェースが組み込まれていないためです。

連結可能性は、ビューが常に "ライブ" であり、静的データの単純なスナップショットではないことによって実現されます。これを証明するため、先ほどビルドしたフォームの正確な複製を構築します。この段階では、作成したフォームに簡単にアクセスできるように、アプリケーションにメニューを追加することも簡単でしょう。アプリケーションを保存、ビルド、および実行します。今回は、作成した2つのフォームインスタンスを開き、一方のフォームのグリッドでデータを変更します。もう一方のフォームのグリッドで、対応するデータが自動的に変更されることに注目してください。これらのフォームのグリッドに対して、行った変更を同期するためのコードを1行も記述していないことを思い出してください。

「ライブビュー」では、その他の LiveLinq の機能について確認します。

コードでのデータソースの操作

ここまでは、ほとんどコードを記述せずに、デザインサーフェスで直接データソースを設定してきました。これらの設定は **DataSource for Entity Framework** によってたいへん簡単になりましたが、すべてをコードで行いたい場合もあります。**C1DataSource** はこれも可能にします。これまでに行ったことはすべて、実行時にコードで行うことができます。

その取りかかりとしてわかりやすい方法は、**C1DataSource** の **ViewSourceCollection** の要素として実質的にデザイナで設定してきた **ClientViewSource** オブジェクトを **C1DataSource** なしで独自に作成できるなら、これを使用することです。ただし、一歩下がって、下位レベルのクラス **ClientView**<T> を使用することもできます。これにより、サーバーからのデータのロー

Copyright © GrapeCity Inc. All rights reserved.

ドを完全に制御できます。また、このクラスは C1.LiveLinq.LiveViews.View<T> から派生しているため、任意の LiveLinq 演算子を適用できます。データソースを View<T> に設定できる任意の GUI コントロールに対してこのクラスを連結できるということは、完全に編集可能なデータビューが手に入るということでもあります。

サーバー側のフィルタ処理は、おそらく最もよく使用されている操作です。普通、データベーステーブル全体を無制限のままクライアントに提供しようとは誰も考えないからです。先ほどは、**C1DataSource** を使用してこれをコードなしで簡単に実行できることを確認しましたが、ここでは実行時コードを使用します。

C1DataSource なしの実行時コードでクライアント側のデータキャッシュを使用するには、プロジェクトのメインクラスに数行を追加して、グローバルなクライアント側データキャッシュを作成します。**C1DataSource** を使用する場合、これはバックグラウンドで作成されました。今回は、次のコードを使用して明示的に作成します。

VisualBasic

```
Imports C1.Data.Entities
Class Application
     Public Shared ClientCache As EntityClientCache
     Private Sub Application_Startup(sender As Object, e As
System.Windows.StartupEventArgs) Handles Me.Startup
         ClientCache = EntityClientCache.GetDefault(GetType(NORTHWNDEntities))
     End Sub
End Class
```

C#

```
using C1.Data.Entities;
public partial class App : Application
{
    public static EntityClientCache ClientCache;
    public static NORTHWNDEntities ObjectContext;
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        ObjectContext = new NORTHWNDEntities();
        ClientCache = new EntityClientCache(ObjectContext);
    }
}
```

このコードは、アプリケーションレベルの(静的な)**ObjectContext** を1つ作成し、それを **EntityClientCache** に関連付けます。先に「クライアントデータキャッシュの能力」トピックで説明したように、アプリケーション全体で1つのコンテキスト(およびキャッシュ)を保持できることは、**C1DataSource** によって可能になった大幅な簡略化です。

実行時コードでサーバー側のフィルタ処理を実行するには、次の手順に従います。

1. 新しいフォームを追加し、フォームにグリッド(**dataGridView1**)、コンボボックス(**comboBox1**)、およびボタン(**btnSaveChanges**)を追加してフォームクラスに次のコードを追加します。

   VisualBasic

   ```
   Imports C1.Data.Entities
           Imports C1.Data
   Public Class DataSourcesInCode
        Private _scope As EntityClientScope
               Public Sub New()
             ' This call is required by the designer.
                   InitializeComponent()
             ' Add any initialization after the InitializeComponent() call.
   ```


Copyright © GrapeCity Inc. All rights reserved.

```
          _scope = Application.ClientCache.CreateScope()
          Dim viewCategories As ClientView(Of Category) = _scope.GetItems(Of
Category)()
          comboBox1.DisplayMemberPath = "CategoryName"
                  comboBox1.ItemsSource = viewCategories
          BindGrid(viewCategories.First().CategoryID)
              End Sub
      Private Sub BindGrid(categoryID As Integer)
                  dataGrid1.ItemsSource =
                         (From p In _scope.GetItems(Of Product)().AsFiltered(
                               Function(p As Product) p.CategoryID.Value =
categoryID)
                           Select New With
                           {
                               p.ProductID,
                               p.ProductName,
                               p.CategoryID,
                               p.Category.CategoryName,
                               p.SupplierID,
                               .Supplier = p.Supplier.CompanyName,
                               p.UnitPrice,
                               p.QuantityPerUnit,
                               p.UnitsInStock,
                               p.UnitsOnOrder
                           }).AsDynamic()
              End Sub
              Private Sub comboBox1_SelectionChanged(sender As System.Object, e
As System.Windows.Controls.SelectionChangedEventArgs) Handles
comboBox1.SelectionChanged
                  If comboBox1.SelectedValue IsNot Nothing Then
                      BindGrid(CType(comboBox1.SelectedValue,
Category).CategoryID)
                  End If
              End Sub
      Private Sub btnSaveChanges_Click(sender As System.Object, e As
System.Windows.RoutedEventArgs) Handles btnSaveChanges.Click
                  Application.ClientCache.SaveChanges()
              End Sub
End Class
```

C#

```
using C1.Data.Entities;
using C1.Data;
public partial class DataSourcesInCode : Window
{
    private EntityClientScope _scope;
    public DataSourcesInCode()
    {
        InitializeComponent();
        _scope = App.ClientCache.CreateScope();
        ClientView<Category> viewCategories = _scope.GetItems<Category>();
        comboBox1.DisplayMemberPath = "CategoryName";
```


Copyright © GrapeCity Inc. All rights reserved.

```
            comboBox1.ItemsSource = viewCategories;
            BindGrid(viewCategories.First().CategoryID);
        }
        private void comboBox1_SelectionChanged(object sender,
SelectionChangedEventArgs e)
        {
            if (comboBox1.SelectedValue != null)
                BindGrid(((Category)comboBox1.SelectedValue).CategoryID);
        }
        private void BindGrid(int categoryID)
        {
            dataGrid1.ItemsSource =
                (from p in _scope.GetItems<Product>().AsFiltered(
                      p => p.CategoryID == categoryID)
                 select new
                 {
                     p.ProductID,
                     p.ProductName,
                     p.CategoryID,
                     CategoryName = p.Category.CategoryName,
                     p.SupplierID,
                     Supplier = p.Supplier.CompanyName,
                     p.UnitPrice,
                     p.QuantityPerUnit,
                     p.UnitsInStock,
                     p.UnitsOnOrder
                 }).AsDynamic();
        }
        private void btnSaveChanges_Click(object sender, RoutedEventArgs e)
        {
            App.ClientCache.SaveChanges();
        }
}
```

2. アプリケーションを保存、ビルド、および実行します。「サーバー側のフィルタ処理」の例と同じ結果が得られますが、今回は、すべてをコードで実装した点が異なります。

それでは、先ほど記述したコードを見ていきます。

プライベートフィールド **_scope** は、フォームからグローバルデータキャッシュへのゲートウェイになります。**C1DataSource** コンポーネントを直接使用する場合はこれが自動的に行われるため、直接使用しない場合はこのパターンに準拠することが推奨されます。これにより、フォームが有効な間、フォームが必要とするエンティティはキャッシュに留まり、それらのエンティティを保持するすべてのフォーム（スコープ）が解放されると、エンティティも自動的に解放されます。

コンボボックスにすべてのカテゴリを表示するビューを作成することは簡単です。

VisualBasic

```
Dim viewCategories As ClientView(Of Category) = _scope.GetItems(Of Category)()
```

C#

```
ClientView<Category> viewCategories = _scope.GetItems<Category>();
```

ビューをグリッドに連結し、コンボボックスで選択されたカテゴリに関連する製品だけを提供するには、追加の演算子 AsFiltered(<predicate>) が必要です。

Copyright © GrapeCity Inc. All rights reserved.

VisualBasic

```
From p In _scope.GetItems(Of Product)().AsFiltered(Function(p As Product)
p.CategoryID.Value = categoryID)
```

C#

```
from p in _scope.GetItems<Product>().AsFiltered(p => p.CategoryID == categoryID)
```

このクエリーが実行されても、要求された製品を取得するために必ずしもサーバーへのラウンドトリップが必要にならないことに注意してください。この要求データがこのフォームまたはアプリケーション内の別のフォームから以前に要求されたことがあるために、既にキャッシュに含まれていないかどうかが最初に調べられます。または、アプリケーション内のどこかで完全に別のクエリーが実行されて、すべての製品を返すように要求されたため、すべての製品データがキャッシュに既に含まれている場合もあります。これも、**C1DataSource** の基本的な長所の1つです。アプリケーションにデータのグローバルキャッシュを提供することで、アプリケーションの全ライフタイムを通してパフォーマンスが継続的に改善されます。

ここでは、ユーザーがコンボボックスで新しいカテゴリを選択するたびに(コンボボックスの **SelectedValueChanged** イベントを参照)、新しいビューを作成し、そこにグリッドを連結することにしました。ただし、いつも新しいビューを作成するのではなく、特別な **BindFilterKey** を使用してビューを1つ作成するだけで済ますこともできます。これについては、「MVVM の簡略化」で詳しく説明します。

要するに、ここでは、「サーバー側のフィルタ処理」で **C1DataSource** を使用してデザインサーフェス上で行った作業をコードでも繰り返したということです。さらに、少しばかり手を加えて、「ビューのカスタマイズ」で LiveLinq ステートメントに **Select** を追加して行ったように、グリッド列に表示されるフィールドをカスタマイズしました。

```
Select New With
 {
     p.ProductID,
     p.ProductName,
     p.CategoryID,
     p.Category.CategoryName,
     p.SupplierID,
     Supplier = p.Supplier.CompanyName,
     p.UnitPrice,
     p.QuantityPerUnit,
     p.UnitsInStock,
     p.UnitsOnOrder
 }
```

```
select new
{
     p.ProductID,
     p.ProductName,
     p.CategoryID,
     CategoryName = p.Category.CategoryName,
     p.SupplierID,
    Supplier = p.Supplier.CompanyName,
    p.UnitPrice,
     p.QuantityPerUnit,
     p.UnitsInStock,
     p.UnitsOnOrder
 };
```


Copyright © GrapeCity Inc. All rights reserved.

特別な書式設定なしで、生の製品データをテーブルから返すだけなら、次のように記述できます。
select p;

ライブビュー

クライアントビューはライブです。クライアントビューは、データの変化に伴って自動的に最新の状態を維持します。ただし、ライブビュー機能はさらに汎用的です。ライブビュー（**View** クラスのオブジェクト。ClientView の派生元）は、エンティティだけでなく、任意の種類のデータに対して定義できます。また、グループ化、ソート、結合などの多くの LINQ クエリー演算子をサポートします（「C1LiveLinq」を参照）。

これらすべての操作（中でもグループ化とソートが最もよく使用される）をクライアントビューに適用できます（**ClientView** は View から派生しているため）。次に例を示します。

LINQ

```
View productsByCategory = products
     .OrderBy(p => p.ProductName).GroupBy(p => p.CategoryID);
LINQ クエリー構文の場合は次のようになります。
View productsByCategory =
    from p in products
    orderby p.ProductName
    group p by p.CategoryID into g
    select new { Category = g.Key, Products = g };
```

結果のビューはライブですが、クライアントビューではありません。これは何らかの欠陥ではありません。単に、このようなビューでは ClientView 機能が必要ないためです。ソートやグループ化は、サーバーに頼らず、完全にクライアント上で実行できます。ただし、開発者は、混乱を避けるためにこの事実を認識しておく必要があります。ビューに LINQ クエリー操作を適用すると、**ClientView** ではなく **View** になります（ただし、ライブであることは変わらず、クライアント上でデータに行われたすべての変更が自動的に反映されます）。したがって、たとえば、サーバー側のフィルタ処理が必要な場合は、LINQ *Where* メソッドではなく、AsFiltered メソッドを使用します。クライアント上でフィルタ処理を行う場合は、LINQ *Where* メソッドを使用します。

MVVM の簡略化

開発者がアプリケーションへのメリットを理解するようになるに従い、Model-View-ViewModel（**MVVM**）パターンがよく使用されるようになっています。その結果、アプリケーションの保守とテストが簡単になり、特に WPF と Silverlight アプリケーションの場合は、UI の設計者とそれを機能させるコードの作成者の間の作業分担が格段に明確になっています。しかし、**MVVM** パターンに基づいてプログラム開発を支援する効果的なツールがないと、余分なレイヤ（ビューモデル）を実装して、そのレイヤとモデルとの間でデータが正しく同期されているかどうかを確認する必要があるため、実質的に作業が難しくなります。少数の単純なコレクションを使用するだけの比較的小さなアプリケーションの場合（さらに MVVM のベストプラクティスとしてデータソースに **ObservableCollection** を使用する場合）、この余分な負担はそれほど大きなものではありませんが、アプリケーションが大きくなるほど、そして生成するコレクションの数が増えるほど、負担も大きくなります。**Entity Framework DataSource**（**EF DataSource**）は、この負担を和らげることができます。

EF DataSource は、ビューモデルとしてライブビューを使用します。それには、モデルコレクションに対してライブビューを作成し、それをビューモデルとして使用するだけです。ライブビューは、ソース（モデル）と自動的に同期されます。つまり、同期コードは*何も*必要ではなく、すべてが自動的に実行されます。また、コードで **ObservableCollection** を使用し、手作業でコレクションにデータを挿入して、独自のビューモデルクラスを記述する作業と比較して、ライブビューは格段に簡単に作成することができます。LINQ のすべての能力を自由に使用して、モデルデータをライブビューに再形成できます。したがって、同期コードが不要になるだけでなく、ビューモデルを作成するコードが劇的にシンプルになります。

ライブビューを使用して簡単に **MVVM** パターンに準拠できることを具体的に示すため、前の2つの例の機能（「コードでのデータソースの操作」の **Category-Products** マスター/詳細および「ライブビュー」のデータの再形成/フィルタ処理/ソート）をすべて組み合わせたフォームを作成します。**Category-Products** ビューに、単価が **30** 以上の販売終了になっていない製品を単価順に表示します。また、マスター/詳細フォームにカスタマイズされた製品フィールドをいくつか表示し、そこでカテゴリを選択してそのカテゴリの製品を表示できるようにします。ここでは **MVVM** パターンに従います。**CategoryProductsView** と

Copyright © GrapeCity Inc. All rights reserved.

いう名前のフォーム(ビュー)は、GUI コントロール(コンボボックスとグリッド)をホストするだけです。データソースをビューモデルに設定するコード以外のコードは含みません。

すべてのロジックは、GUI とは別のビューモデルクラスにあります。さらに正確に言えば、3つのクラス **CategoryViewModel**、**ProductViewModel**、**CategoryProductsViewModel** があります。最初の2つは、追加コードなしでプロパティを定義する単純なクラスです。

Visual Basic

```
Public Class CategoryProductsViewModel
     Private _scope As EntityClientScope
     Private _categories As ICollectionView
     Public Property Categories As ICollectionView
         Get
             Return _categories
         End Get
         Private Set(value As ICollectionView)
             _categories = value
         End Set
     End Property
     Private _products As ICollectionView
     Public Property Products As ICollectionView
         Get
             Return _products
         End Get
         Private Set(value As ICollectionView)
             _products = value
         End Set
     End Property
     Public Sub New()
         _scope = Application.ClientCache.CreateScope()
         Categories =
             From c In _scope.GetItems(Of Category)()
             Select New CategoryViewModel With
             {
                 .CategoryID = c.CategoryID,
                 .CategoryName = c.CategoryName
             }
         Products =
             From p In _scope.GetItems(Of Product)().AsFilteredBound(Function(p)
p.CategoryID.Value)
                 .BindFilterKey(Categories,
"CurrentItem.CategoryID").Include("Supplier")
             Select New ProductViewModel With
             {
                 .ProductID = p.ProductID,
                 .ProductName = p.ProductName,
                 .CategoryID = p.CategoryID,
                 .CategoryName = p.Category.CategoryName,
                 .SupplierID = p.SupplierID,
                 .SupplierName = p.Supplier.CompanyName,
                 .UnitPrice = p.UnitPrice,
                 .QuantityPerUnit = p.QuantityPerUnit,
                 .UnitsInStock = p.UnitsInStock,
```


Copyright © GrapeCity Inc. All rights reserved.

```
                .UnitsOnOrder = p.UnitsOnOrder
            }
        End Sub
End Class
```

C#

```
public class CategoryViewModel
{
    public virtual int CategoryID { get; set; }
    public virtual string CategoryName { get; set; }
}

public class ProductViewModel
{
    public virtual int ProductID { get; set; }
    public virtual string ProductName { get; set; }
    public virtual int? CategoryID { get; set; }
    public virtual string CategoryName { get; set; }
    public virtual int? SupplierID { get; set; }
    public virtual string SupplierName { get; set; }
    public virtual decimal? UnitPrice { get; set; }
    public virtual string QuantityPerUnit { get; set; }
    public virtual short? UnitsInStock { get; set; }
    public virtual short? UnitsOnOrder { get; set; }
}

public class CategoryProductsViewModel
{
    private C1.Data.Entities.EntityClientScope _scope;

    public System.ComponentModel.ICollectionView Categories { get; private set; }
    public System.ComponentModel.ICollectionView Products { get; private set; }

    public CategoryProductsViewModel()
    {
        if (App.ClientCache == null)
            return;

        _scope = App.ClientCache.CreateScope();

        Categories =
            from c in _scope.GetItems<Category>()
            select new CategoryViewModel()
            {
                CategoryID = c.CategoryID,
                CategoryName = c.CategoryName
            };

        Products =
            from p in _scope.GetItems<Product>().AsFilteredBound(p => p.CategoryID)
                .BindFilterKey(Categories,
```


Copyright © GrapeCity Inc. All rights reserved.

```
"CurrentItem.CategoryID").Include("Supplier")
            select new ProductViewModel()
            {
                ProductID = p.ProductID,
                ProductName = p.ProductName,
                CategoryID = p.CategoryID,
                CategoryName = p.Category.CategoryName,
                SupplierID = p.SupplierID,
                SupplierName = p.Supplier.CompanyName,
                UnitPrice = p.UnitPrice,
                QuantityPerUnit = p.QuantityPerUnit,
                UnitsInStock = p.UnitsInStock,
                UnitsOnOrder = p.UnitsOnOrder
            };
    }
}
```

基本的に、2つの LiveLinq ステートメントが含まるほかは、何も含まれていません。

「コードでのデータソースの操作」で説明したように、AsFilteredBound を使用して、サーバー側でフィルタ処理を行うことができます。「コードでのデータソースの操作」では、コンボボックスイベントを使用して選択された **CategoryID** にフィルタキー（**Product.CategoryID** プロパティ）を接続しました。ここでは、コードを GUI から独立に保つために、この方法は使用できません。したがって、**BindFilterKey** メソッドを使用して、**Categories** コレクションで現在選択されている項目の **Category.CategoryID** プロパティにフィルタキーを連結します。

Include（"Supplier"）演算子は厳密には必要ではありませんが、ここでは、パフォーマンスを最適化するために使用されています。これがないと、**Entity Framework** は、まだ **Supplier** がフェッチされていない **Products** コレクションの要素にユーザーがアクセスするたびに、**Supplier** オブジェクトを1つずつ必要に応じてフェッチします。これにより、遅延が発生する可能性があります。また、バッチではなく、単一行のデータをフェッチすると一般に効率が大きく低下します。したがって、ここでは、製品と同じクエリーでサプライヤ情報をフェッチするように **Entity Framework** に指示する **Include**（"Supplier"）を使用して、遅延ロードを回避します。

最後に、フォーム（ビュー）CategoryProductsView.xaml 内の GUI コントロールに対してデータ連結を指定する必要があります。MVVM パターンに準拠し、XAML（コードではない）で次のように指定します。

XAML

```
<Grid>
    <Grid.DataContext>
        <local:CategoryProductsViewModel />
    </Grid.DataContext>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition />
    </Grid.RowDefinitions>
    <ComboBox HorizontalAlignment="Left" Margin="5" Name="comboBox1"
        Width="221" ItemsSource="{Binding Categories}"
        DisplayMemberPath="CategoryName" />
    <DataGrid Grid.Row="1" AutoGenerateColumns="True" Name="dataGrid1"
        ItemsSource="{Binding Products}"/>
</Grid>
```

次のように DataContext を定義します。

XAML

```
<Grid.DataContext>
```


Copyright © GrapeCity Inc. All rights reserved.

```
        <local:CategoryProductsViewModel />
    </Grid.DataContext>
```

フォームのデータソースとして **CategoryProductsViewModel** オブジェクトを作成し、"{Binding Categories}" および "{Binding Products}" を使用してそこにコンボボックスとグリッドを連結しました。

他の MVVM フレームワークを使用した MVVM での C1DataSource の使用

C1DataSource を使用すると、他の任意の Model-View-ViewModel（**MVVM**）フレームワークで MVVM アプリケーションを作成できます。

C1DataSource は、MVVM 開発を容易にするさまざまな機能を提供します。

- **MVVM プログラミングの簡略化**

  「MVVM の簡略化」で説明したように、**C1DataSource** を使用して MVVM プログラミングを簡略化できることから、MVVM に使用できるツールであることは明らかです。

- **ビューモデルクラスの作成を支援し、コードの肥大化を防止**

  MVVM 作業の支援に使用できるツールやフレームワークは多数存在しますが、ビューモデルクラスの作成を支援するツールはほとんどありません。大部分は、ビューとビューモデルの間でコマンドやメッセージを受け渡しするなどのタスクを支援する目的で設計されています。ビューモデルクラスを作成し、それをモデルデータと同期する作業は、ほぼ完全に手作業のコーディングに委ねられています。これが、大部分の MVVM アプリケーションでコードが肥大化する主な原因であり、また他のフレームワークと完全に互換な方法でこれを緩和するように **C1DataSource** が設計されている理由でもあります。

- **任意のフレームワークとライブビューを使用可能**

  MVVM アプリケーションの開発に便利な好みのフレームワークを使用できます。後は、**C1DataSource** を利用してライブビューを提供するだけで、ビューモデルクラスを作成できます。

これらの重要な点を具体的に説明するために、MVVM の考案者の一人である Josh Smith の有名な記事「Model-View-ViewModel デザインパターンによる WPF アプリケーション」（https://msdn.microsoft.com/ja-jp/magazine/dd419663.aspx）に基づくサンプルコードを提供しています。

このサンプルは、**Documents\ComponentOne Samples\Entity Framework DataSource\OrdersDemo\Orders-EF-MVVM** フォルダにあります。

改変したサンプルに含まれるファイルは、1つのファイル ViewModels\OrdersViewModel.cs を除いて、いずれも基本的に元のサンプルと同じです（小さな外観の変更を除く）。

このファイルでは、ライブビューを使用して、**C1DataSource** の方式でビューモデルクラスを作成しています。ビューモデルを構築するために、いくつの再形成関数がモデルデータに適用されているかを確認できます。LINQ だけを使用してすべてが実行されています。このようにビューモデルの構築は容易で、必要なコードも減ります。2つのレイヤのいずれかでデータが変更された場合に、モデルデータと自動的に同期する部分が最も優れた部分です。同期のためのコードは不要です。

ビューモデルクラス自身の作成方法だけを変更したという事実（派生元の基本クラスは 'ViewModelBase' のまま）、さらに Josh Smith が元の例で採用したフレームワークコードを変更していないという事実は、**C1DataSource** が他のフレームワークと完全に互換性があることを示す例となります。MVVM を使用した作業では、引き続き好みのフレームワークを使用できますが、MVVM 開発をさらに容易にするツールが追加されています。

プログラミングガイド

以下のセクションでは、**C1DataSource** プログラミングに関する情報を提供します。LiveLinq 固有の機能の詳細については、「**LiveLinq プログラミングガイド**」を参照してください。

Copyright © GrapeCity Inc. All rights reserved.

ライブビューでサポートされるクエリー演算子

LiveLinq は、クエリーですべての LINQ クエリー演算子をサポートします。必ずしもすべての演算子に LiveLinq 固有の実装があるわけではなく、インデックスなどのクエリー実行最適化技術の恩恵を受けることはできませんが、そのような演算子では、単に標準の **LINQ to Objects**（または **LINQ to XML**）実装が使用されるため、ユーザーからは透過的です。

ただし、ライブビューでは、必ずしもすべてのクエリー演算子を使用できません。これは、一部のクエリー演算子は管理のたびに最初から再生成（再クエリー）する必要があり、インクリメンタルビューメンテナンスアルゴリズムを備えていないためです。クエリーにこのような演算子が含まれる場合、そのクエリーを使用してライブビューを作成することはできません。作成しようとすると、コンパイルエラーが発生します。

次に、ライブビューで使用できるクエリー演算子をリストします。

演算子	説明
Select	インデックスに依存する **selector** を使用したオーバーロードは許可されません。
Where	インデックスに依存する **predicate** を使用したオーバーロードは許可されません。
Join	**comparer** を使用したオーバーロードは許可されません。
GroupJoin	**comparer** を使用したオーバーロードは許可されません。
OrderBy	**comparer** を使用したオーバーロードは許可されません。
OrderByDescending	**comparer** を使用したオーバーロードは許可されません。
GroupBy	**comparer** を使用したオーバーロードは許可されません。
SelectMany	インデックスに依存する **selector** および **collectionSelector** を使用したオーバーロードは許可されません。
Union	
Concat	
Aggregate	インクリメンタルビューメンテナンスを使用してパフォーマンスを最適化する場合は、**LiveAggregate** メソッドを使用してください。
Count	インクリメンタルビューメンテナンスを使用してパフォーマンスを最適化する場合は、**LiveCount** メソッドを使用してください。
Min/Max	インクリメンタルビューメンテナンスを使用してパフォーマンスを最適化する場合は、**LiveMin** メソッドを使用してください。
Sum	インクリメンタルビューメンテナンスを使用してパフォーマンスを最適化する場合は、**LiveSum** メソッドを使用してください。
Average	インクリメンタルビューメンテナンスを使用してパフォーマンスを最適化する場合は、**LiveAverage** メソッドを使用してください。

ライブビューでサポートされるクエリー式

クエリーをライブビューとして使用するには、クエリー演算子の制限以外にも、クエリーが満たすべき条件があります。幸い、この条件はほとんどのケースで満たされます。これは、クエリーに特殊な式が含まれる場合にのみ考慮する必要があり、このような状況は比較的まれです（ただし、LiveLinq to Objects で使用されるクラスは、必ずプロパティ通知条件を満たす必要があります。これについては、「組み込みのコレクションクラス IndexedCollection の使用（LiveLinq to Objects）」で既に説明しました）。ただし、この条件は LiveLinq によって自動的には検証されません。したがって、この条件を満たすかどうかは、ユーザー自身の責任において確認する必要があります。この条件が満たされない場合、ビューはベースデータが変更されても反応しません。この条件については2つの説明を用意しています。1つめは基本的な理解のための簡単な説明、2つめは高度な用途

Copyright © GrapeCity Inc. All rights reserved.

に関する詳細な説明です。

ビューメンテナンスモード

ライブビューは、さまざまな種類のアプリケーションで使用できます。対話式の GUI アプリケーションでも、バッチ処理形式の非 GUI アプリケーションでも使用できます(「非 GUI コードでライブビューを使用する方法」を参照)。ライブビューは、GUI(対話式)と非 GUI(バッチ)の両方のモードに対して最適化されています。これらのモードを区別し、モードに応じて動作します。GUI では、変更が発生するとすぐにその変更に反応します。この迅速な処理は、ビューを再作成する代わりにインクリメンタルアルゴリズムを使用することで実現されています(「ビューの保守:インクリメンタルビューメンテナンス」を参照)。ただし、ライブビューが、対話式ではないバッチプロセスに常に適しているとは限りません。データ連結を使用する対話式の GUI プログラムでは、ユーザーが画面上で変更を確認する必要があるため、変更へのすばやい反応が求められることが普通です。一方、バッチ処理の場合は、変更の発生後、しばらくたってからビューにアクセスすることも、ビューにまったくアクセスしないこともあります。このため、実際にビューにアクセスする前にプログラムがビューを更新すると、リソースを不必要に消費することになります。デフォルトでは、ライブビューはこの2つのモードを自動的に区別します。ただし、**MaintenanceMode** プロパティを使用して、プログラマがモードを制御することもできます。**Immediate** モードは、GUI の場合のデフォルトのモードです。このモードでは、変更のたびにすぐにビューが保守されます。**Deferred** モードは、非 GUI の場合(ビューにリスナーが接続されていない場合)のデフォルトのモードです。このモードでは、オンデマンドでビューが保守されます。同期が必要になるまで、またはビューにデータが要求されるまで、ビューはベースデータと同期しません。データの要求を受け取ったビューは、ビューが "ダーティ" な非同期状態にあると見なし、自動的に更新(保守)を行って、変更されたベースデータとの同期を開始します。

更新可能なビュー

ライブビューは双方向に更新可能です。1つめは、ベースデータからビュー方向への更新です。ベース(ソース)データは変更されることがあり、ビューは自動的に更新されて、変更後のベースデータと同期されます。この機能により、ビューはライブになります。2つめは、その逆方向への更新です。データはビューで直接変更されることがあります。この更新は、プログラムを使用して(**ViewRow** を使用)実行することができます。また、データ連結を使用して実行することもできます。グリッドなどの GUI コントロールをビューに連結している場合は特に、ビューで直接データを更新することが普通です。

ビューでデータを直接変更できる場合は、そのビューを "更新可能" と呼びます。この用語は、2つめの方向のデータ更新だけを対象にします。1つめの方向のデータ更新(ベースデータの更新)は、どのライブビューでも制限なく可能です。このため、ビューが "更新可能" ではない("読み取り専用" である)と言う場合でも、そのビューのデータが変化しないという意味ではありません。LiveLinq のすべてのビューはライブであり、ベースデータに加えられた変更は自動的にビューに反映されます。したがって、これは、ビューでは直接データを変更できず、データを修正するにはベースデータを変更する必要があるという意味に過ぎません。

すべてのライブビューが更新可能なわけではありません。さらに、ビュー全体が更新可能でも、一部のプロパティが読み取り専用である場合があります。これらは、ビューのソースデータ内のプロパティに直接対応していないプロパティです。結局のところ、ビューの更新とは、ビューのベースデータ、つまりビューのソースのいずれか1つを更新することを意味します。これは、ビュー自体は仮想の存在で、ビューに独自のデータはなく、ソースのデータを(フィルタ処理および形成して)表示しているに過ぎないからです。このため、ビューフィールド(プロパティ)は、ソース内のプロパティと直接対応させることが可能な場合にのみ、更新することができます。たとえば、次のビューを考えます。

C#

```
for c in customers where c.City == "London"
select new
{
    c.City,
    c.CompanyName,
    FullName = c.FirstName + " " + c.LastName
}
```

City と **CompanyName** は、**customers** ソースの **City** フィールドと **CompanyName** フィールドに直接対応しているため、更新可能です。一方、**FullName** は更新可能ではありません。これは、ソースには FullName に対応する単一のフィールド(プロパティ)がなく、FullName が2つのソースプロパティの組み合わせだからです。

Copyright © GrapeCity Inc. All rights reserved.

ビューの更新(ビュー項目の追加、削除、変更)は、ビューのソースのいずれか1つで対応する操作(単一項目の追加、削除、変更)を実行することによって行われます。通常、操作の影響は明らかであり、意図したとおりのことが行われますが、この単純なルールを知り、正確に理解することは必要です。それは、このルールを考慮しないと、予期しない結果になることがあるためです。たとえば、前述のビューで、**City** の値が "London" 以外になるように項目を変更すると(または新しい項目を追加すると)、その項目はビューから消えます。

ビュー項目を更新することは、ビューのソースのいずれか1つで項目を更新することと同じであるというルールを上で述べました。これは、ビューのソースのいずれか1つだけが更新可能であるということを意味しています。**結合**ビューには2つのソースがあるので、どちらのソースを更新可能にするかを決定する必要があります。デフォルトでは、結合ビューは読み取り専用です。この2つのソースの一方を更新可能にする場合は、拡張メソッド **AsUpdatable()** を使用します。たとえば、次のように指定します。

C#

```
for o in orders.AsUpdatable()
join c in customers on o.CustomerID equals c.CustomerID
select new
{
    o.OrderDate,
    o.Amount,
    c.CompanyName
}
```

ここでは、注文データ(日付と金額)は更新可能で、顧客データ(**CustomerName**)は読み取り専用になります。

ビューが更新可能かどうかを確認するには、プロパティ **View.IsReadOnly** を使用します。データ連結およびプログラムからアクセス可能なビューのすべてのプロパティを、更新可能性を含むすべての情報と合わせて取得するには、プロパティ **ViewRowCollection.Properties** を使用します。

コードでビューのデータを直接更新するには、特別なクラス **ViewRow** を使用します。これは、プログラムからのアクセスやデータ連結に使用されるビュー項目を表すビュー行です。このクラスをコードで使用する方法については、**ViewRowCollection** および **ViewRow** のリファレンスドキュメントを参照してください。

ライブビューのパフォーマンス

ライブビューのパフォーマンスについて最初に考慮する必要があるのは、ライブビューの機能には必然的に代償が伴うということです。ベースデータとの同期を維持するようにビューを保守する作業は最適化されており、高速に実行されます。それでも、ベースデータを変更したりビューを保守するには、いくらかのリソースが消費されます。また、ビューに最初にデータを挿入する際にも、さらに追加リソースが消費されます。この代償はそれほど大きくありませんが、存在することに変わりはなく、主にメモリ消費量の増加という形で現れます。ライブビューにデータが挿入されると、メモリ内に何らかの内部データが作成され、ベースデータが変更されると、その内部データを使用して高速にビューが保守されます。

この追加メモリ消費量は適度なもので、結果のリスト自体に必要なメモリ量とほぼ同じです。このため、ライブビューの機能に必要な追加リソース量が低から中程度であるとしても、ライブビューは、クエリーの結果をライブの状態に保つ必要が本当にある場合、つまりその結果が何度も必要になり、ベースデータが変化しているために結果も変化する場合にのみ使用した方がよいことは明らかです。

ライブビューのパフォーマンスには、このほかにも2つの側面があります。

LiveLinq クエリーパフォーマンス:論理的な最適化

標準の LINQ to Objects は、論理的な最適化を実行せず、クエリーを記述されたとおりに正確に実行します。当然、標準の LINQ to Object は、最適化のためにインデックスを使用することもありません。対照的に、LiveLinq は、物理的な最適化(インデックスがあればインデックスを使用する)と論理的な最適化(実行前にクエリーを効率的な形式に書き換える)を行います。

LiveLinq には、SQL Server や Oracle などのリレーショナルデータベースに含まれているような、完全なクエリーオプティマイ

Copyright © GrapeCity Inc. All rights reserved.

ザは含まれていません。このため、クエリーがどのように記述されているかが引き続き問題となる場合があります。ただし、これは主に結合順序に限られます。LiveLinq は、結合を並べ替えることはしません。結合は、常に、クエリーで指定された順序で実行されます。このため、明らかに非効率的な結合順序でクエリーを記述しないようにする必要があります。ただし、これは比較的珍しい問題であること(そして、同じ問題が標準の LINQ to Objects にも当然当てはまること)に注意してください。次のようなクエリーでは、結合の順序が問題になることがあります。

LINQ
```
from a in A
join b in B on a.k equals b.k                         (1)
where b.p == 1
```

たとえば、**b.p == 1** となる要素が **B** に 10 個しかなく、**a.k == b.k** かつ **b.p == 1** を満たす要素が **A** に 100 個しかないが、**A** の要素の総数がその何倍にもなる(10000 個など)場合です。その場合、上のクエリーには 10000 "サイクル" が必要です。一方、次のように正しい結合順序でクエリーを書き直すと、

LINQ
```
from b in B
join a in Aon b.k equals a.k                          (2)
where b.p == 1
```

このクエリーの実行には 100 サイクルしか必要になりません。**where** の位置は重要ではないことに注意してください。上のクエリーのパフォーマンスは、次のクエリーと同じです。

LINQ
```
from b in B
where b.p == 1                                         (3)
join a in A on b.k equals a.k
```

これは、LiveLinq によってクエリーが最適化されるためです。つまり、LiveLinq は、実行時に内部で(2)を(3)に書き換えます。これは、他の操作を実行する前に条件をチェックして、結果に影響しない要素を除外した方がよいからです。この処理は、LiveLinq によって内部的に実行される論理的な最適化の1つです。

LiveLinq クエリーパフォーマンス:インデックスパフォーマンスの調整

LiveLinq は、インデックスを使用した最適化によってクエリーを高速化しますが、どの程度高速化されるかは、クエリーの記述方法に左右されます。通常、クエリーの記述方法による違いはそれほど大きくありません。LiveLinq は、クエリーの記述方法に関係なく、インデックスを使用した最適化の可能性を正しく認識します。たとえば、複数の項が論理演算子で接続された条件がある場合でも、インデックスを使用して効率よくクエリーを実行します。

ライブビュー:他のビューに基づいてビューを作成し、ビューにインデックスを作成する方法

パラメータを含む複数のクエリーを1つの "大きな" ライブビューに置き換えられることがよくあります。こうすると、コードが簡潔になり、実行速度も向上することがあります。この技術は、ライブビューの次の2つの機能に基づいています。

Copyright © GrapeCity Inc. All rights reserved.

- ベースデータに基づいて作成する方法と同様に、ビューを他のビューに基づいて作成することができます。
- ベースデータにインデックスを作成する方法と同様に、ビューにインデックスを作成することができます。

そこで、パラメータ値を変えて複数のクエリーを発行する代わりに、1つのライブビューを作成し、そのビューにインデックスを作成します。特定のパラメータ値に対応する項目のリストが必要な場合は、その値のインデックスから項目を取得することができます。

この機能は、「LiveLinqIssueTracker デモ」に具体例があります。

Assigned Issues フォームでは、複数のビューが使用されています。これは、従業員ごとの個別ビューで、パラメータ **employeeID** に依存します。ここでは、LiveLinq to DataSet バージョンのビューを使用していますが、Objects および XML バージョンのビューも同様です。

LINQ

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join e in _dataSet.Employees.AsLive()
    on i.AssignedTo equals e.EmployeeID
where i.AssignedTo == employeeID
select new Issue
{
    IssueID = i.IssueID,
    ProductName = p.ProductName,
    FeatureName = f.FeatureName,
    Description = i.Description,
    AssignedTo = e.FullName
};
```

デモアプリケーションは、**Assigned Issues 2** フォームで同じ機能を別の方法でも実装しています。このフォームでは、パラメータに依存する複数のビューではなく、全従業員のデータを含む1つのビューが使用されています。この1つのビューにパラメータはありません。

LINQ

```
_bigView =
    from i in _dataSet.Issues.AsLive()
    join p in _dataSet.Products.AsLive()
        on i.ProductID equals p.ProductID
    join f in _dataSet.Features.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { f.ProductID, f.FeatureID }
    join e in _dataSet.Employees.AsLive()
        on i.AssignedTo equals e.EmployeeID
    select new Issue
    {
        IssueID = i.IssueID,
        ProductName = p.ProductName,
        FeatureName = f.FeatureName,
        Description = i.Description,
        AssignedToID = e.EmployeeID,
```


Copyright © GrapeCity Inc. All rights reserved.

```
        AssignedToName = e.FullName
    };
```

このビューには、従業員 ID フィールドに基づくインデックスが作成されます。

LINQ

```
_bigView.Indexes.Add(x => x.AssignedToID);
```

このため、特定の従業員 ID の項目をいつでも高速に取得できます。

さらに、この大きなビューからビューを作成するだけで、特定の従業員 ID 値(このデモでは **comboAssignedTo.SelectedIndex**)に対応するデータのライブビューを作成することができます。

LINQ

```
from i in _bigView where i.AssignedToID == comboAssignedTo.SelectedIndex select i;
```

ライブビュー:ライブビューを使用して非 GUI アプリケーションを作成する方法

LiveLinq を使用すると、宣言型の GUI アプリケーションを作成できることに加えて、非 GUI のバッチ処理を宣言型として作成するプログラミングスタイル("**ビュー指向プログラミング**" とも呼ばれます)も採用することができます。この技術の概要については、「非 GUI コードでライブビューを使用する方法」を参照してください。

LiveLinqIssueTracker デモアプリケーションでは、**Batch Processing** フォームを使用してこの技術を説明しています。このフォームは、次の2つのアクションを実装します。

1. 未割り当ての課題をできるだけ多く従業員に割り当てます。特定の課題が属している機能を探し、その機能に割り当てられている従業員を見つけます。その従業員に、その機能に関する他の課題が割り当てられていない場合は、その課題を従業員に割り当てます。
2. 特定の従業員に割り当てられている未解決の課題に関する情報を収集します(たとえば、課題のリストを従業員に電子メールで送信するために)。

次のビューを定義して、アクション (1) を実行します(ここでは、ビューの LiveLinq to DataSet バージョンを示します。Objects バージョンと XML バージョンも同様です)。

C#

```
_issuesToAssign =
    from i in issues
        where i.AssignedTo == 0
    join a in _dataSet.Assignments.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { a.ProductID, a.FeatureID }
    join i1 in issues
        on new { i.ProductID, i.FeatureID, a.EmployeeID }
            equals new { i1.ProductID, i1.FeatureID, EmployeeID = i1.AssignedTo }
        into g
    where g.Count() == 0
    join e in _dataSet.Employees.AsLive()
        on a.EmployeeID equals e.EmployeeID
    select new IssueAssignment
    {
        IssueID = i.IssueID,
```


Copyright © GrapeCity Inc. All rights reserved.

```
        EmployeeID = a.EmployeeID,
        EmployeeName = e.FullName
    };
```

次に、このビューに基づいて、単純なコードで操作を実行します。

C#

```
foreach (IssueAssignment ia in _issuesToAssign)
    _dataSet.Issues.FindByIssueID(ia.IssueID).AssignedTo = ia.EmployeeID;
```

アクション (2) は、次のビューを定義することで実行されます。

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join em in _dataSet.Employees.AsLive()
    on i.AssignedTo equals em.EmployeeID
    where i.AssignedTo == employeeID && !i.Fixed
select i.IssueID;
```

操作 (1) や (2) を一度だけ実行する必要がある場合は、ライブビューを使用しても意味がありません。ただし、ここでは、実行するアルゴリズム全体の中のいくつかの手順として、このアクション (1) と (2) を何度も実行するプログラムを記述しているとします。これは、特にサーバー側プログラミングではよくある例です。

ライブビューがない場合は、毎回クエリーし直すか(コストが大きくなります)、何らかのコレクションを作成し、それを処理アルゴリズムの終わりまで維持する必要があります。そのようなコレクションは、手書きコードで維持する必要がある上に、複雑になることが普通です。また、複数のプログラマによって記述され、バグが含まれることもしばしばです。複数のプログラマが別々の機能を記述すれば(アクション (1) や (2) はそのような機能のほんの一例です)、コードの整合性を維持するために、他のプログラマが記述しているコードを把握する必要があります。このような作業の管理は困難です。1 年後に新しいアクションや機能が追加される頃には、各部をつなぐロジックが忘れられ、新しいアクションによって破壊されて、複雑なプログラムが作成されるという悪循環が続いていきます。

ライブビューを使用した場合との比較:

アクション (1) および (2) は、実際のプロシージャではなく、宣言型ルールです。ルール (1) は、このビューに、実行すべき割り当てが含まれることを述べています。データにどのような変更があっても、また今から1年後や他のどの時期に何が追加されても、このビューには常に必要な割り当てが含まれます。このロジックは、手続き型のロジックではなく宣言型ルールなので、正しいことが保証されます。

また、ルール (1) の動作は高速です。このアクションを毎回異なるデータに対して 1000 回実行したとしても、そのたびに、必要な量のデータだけが、つまり変更の影響を受けるデータだけが再計算されます。必要な***インクリメンタル***な再計算だけが実行されます。

ルール (2) も同様に宣言型ルールです。これは、計算を繰り返すことなく、変更の影響を受ける部分だけが再計算されるため、高速に実行されます。

アルゴリズムの主要な部分が***ビュー指向***の方法で、(1) と (2) のような一連のルールとしてプログラミングされている場合は、すべてのルールが連携し、ルール間の整合性が自動的に維持されます。このようなビュー/ルールでアルゴリズム全体を表現することが理想的です。それが不可能な場合は、アルゴリズムの一部をこの方法で実装し、残りを通常の手続き型コードで実装します。

Copyright © GrapeCity Inc. All rights reserved.

C1LiveLinq

C1LiveLinq で LINQ を高速化し、ライブビューを手に入れてください。この独自のクラスライブラリは、インデックスなどの最適化技術を使用して LINQ の機能を強化し、LINQ クエリーを 100 倍以上高速化します。また、ライブビューを使用すると、ベースデータが変更されるたびにデータを再挿入しなくても、LINQ クエリーの結果を最新の状態に保つことができます。

LiveLinq の概要

LiveLinq は、関連する2つの方向で LINQ の機能を強化するクラスライブラリです。

- **LINQ を高速化します**
  LiveLinq は、インデックスなどの最適化によってメモリ内の LINQ クエリーを高速化します。高度な選択条件を持つクエリーでは、速度の向上が数百倍から数千倍に達する可能性があります。典型的/平均的な向上はこれほどになりませんが、それでも大幅なものであり、一般には 10 倍から 50 倍の高速化が図られます。
- **LINQ にライブビューのサポートを追加します**
  ライブビュー は、ベースデータが変更されるたびにデータを再挿入しなくても、LINQ クエリーの結果を常に最新の状態に保ちます。これにより、ビューの内外でオブジェクトが編集またはフィルタ処理されたり、小計が更新されるなど、よく使用されるデータ連結シナリオで、LiveLinq が極めて有用になります。古い用語を使用すると、LINQ クエリーはスナップショットに、LiveLinq ビューはダイナセットに相当します。ライブビューは自動的に変更に反応するため、データ連結や GUI だけでなく、他の多くのプログラミングシナリオにおいても、宣言型プログラミングの可能性が大きく広がります。

LiveLinq (Silverlight の場合)

LiveLinq は、Silverlight（Silverlight 4 以降）と .NET Framework で使用できます。アセンブリは、**C1.LiveLinq.dll** の代わりに、**C1.Silverlight.LiveLinq.dll** を使用してください。

LiveLinq のインデックス機能は Silverlight で使用できませんが、ライブビューは Silverlight と .NET Framework で完全にサポートされています。Silverlight バージョンの LiveLinq に、次の名前空間（Silverlight では使用できないインデックス機能および ADO.NET サポートを含む）はありません。

C1.LiveLinq.AdoNet

C1.LiveLinq.Indexing

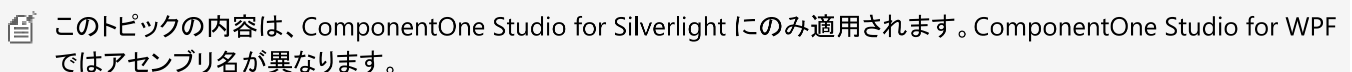
C1.LiveLinq.Indexing.Search

Samples\LiveLinq\HowTo\LiveViews には、次の2つのサンプルプロジェクトが含まれています。それぞれコレクション（サンプルでは ObservableCollection）および XML と組み合わせて、Silverlight で LiveLinq を使用する方法を示しています。

LiveViews-Silverlight-Objects

LiveViews-Silverlight-XML

これらは、データベースなしのインメモリオブジェクトと組み合わせて Silverlight で LiveLinq を使用するサンプルです。ComponentOne Studio for Entity Framework の他のサンプル、チュートリアル、およびマニュアルには、データベースアクセスに RIA サービスを使用する Silverlight のサンプルが多数用意されています。


このトピックの内容は、ComponentOne Studio for Silverlight にのみ適用されます。ComponentOne Studio for WPF ではアセンブリ名が異なります。

LiveLinq の使用方法

LiveLinq でコレクションをクエリーする方法

Visual Studio で LiveLinq を使用するには、最初に、LiveLinq アセンブリ C1.LiveLinq.dll をプロジェクトの参照に追加します。

Copyright © GrapeCity Inc. All rights reserved.

次に、コード内で LiveLinq を利用するための using ディレクティブをソースファイルに追加します。

C#

```
using C1.LiveLinq;
using C1.LiveLinq.Indexing;
using C1.LiveLinq.Collections;
```

(常にこれらすべてが必要になるわけではありませんが、ここで一度にまとめて記述した方が簡単です。)

LiveLinq でコレクションにクエリーを実行するには、コレクションをインタフェース **IIndexedSource<T>** でラップして、LiveLinq に引き継ぎを指示する必要があります。そうしないと、標準の LINQ が使用されます。このラップは、たとえば、拡張メソッド **AsIndexed** の呼び出しで実行されます。

C#

```
from c in source.AsIndexed() where c.p == 1 select source
```

ただし、すべてのコレクションをこの方法でラップできるわけではありません。たとえば、**List<T>** ソースでこれを行うと、コンパイルエラーになります。LiveLinq でコレクションを使用するには、そのコレクションが変更通知をサポートしている必要があります。つまり、LiveLinq オブジェクトに変更が行われたり、コレクションにオブジェクトが追加または削除されたときに、それを LiveLinq に通知する必要があります。データ連結に使用されるコレクション、つまり **IBindingList**（WinForms データ連結）または **INotifyCollectionChanged/INotifyPropertyChanged**（WPF データ連結）を実装するコレクションは、この要件を満たしています。

特に、**AsIndexed()** は、ADO.NET コレクション（**DataTable**、**DataView**）および LINQ to XML コレクションに適用できます。

組み込みのコレクションクラス IndexedCollection の使用（LiveLinq to Objects）

最初に、オブジェクトに対してどのようなコレクションを使用するかは気にしないとします。次のような **Customer** クラスがあります

C#

```
public class Customer
{
    public string Name { get; set; }
    public string City { get; set; }
}
```

また、LiveLinq を使用して必要なコレクションクラスを提供します。

これで、LiveLinq から提供され、LiveLinq の使用に対して特に最適化された **C1.LiveLinq.Collections.IndexedCollection** クラス以外を考慮する必要はなくなります。

C#

```
var customers = new IndexedCollection<Customer>();
customers.Add(cust1);
customers.Add(cust2);
...
var query =
  from c in customers where c.City == "London" select c;
```

customers.AsIndexed() ではなく単に **customers** を使用できることに注意してください。その理由は、**IndexedCollection<T>** クラスには LiveLinq が必要とする **IIndexedSource<T>** インタフェースが既に実装されており、**AsIndexed()** 拡張メソッドを使用してこのインタフェースにラップする必要がないためです。

Copyright © GrapeCity Inc. All rights reserved.

上の **Customer** などの独自のクラスをコレクションの要素に使用する場合は、考慮するべき重要な事項があります。上の **Customer** は基本的なクラスで、プロパティ通知をサポートしていません。コードでプロパティを設定した場合は、コレクションに対して作成されたインデックスやライブビューを含めて、何もその変更を認識できません。したがって、このようなクラスでは、プロパティ変更通知を必ず提供する必要があります。プロパティ変更通知は、さまざまな理由で推奨される標準の .NET 機能ですが、LiveLinq もまたその理由を加えます。**INotifyPropertyChanged** インタフェースを実装することで、独自のクラスでプロパティ変更通知をサポートできます。

C#

```
public class Customer : IndexableObject
{
    private string _name;
    public string Name
    {
        get {return _name} ;
        set
        {
            OnPropertyChanging("Name");
            _name = value;
            OnPropertyChanged("Name");
        }
    }
    private string _city;
    public string City
    {
        get {return _city};
        set
        {
            OnPropertyChanging("City");
            _city = value;
            OnPropertyChanged("City");
        }
    }
}
```

ADO.NET データコレクションの使用（LiveLinq to DataSet）

次の例のように、ADO.NET **DataTable** も LiveLinq で使用できます。

C#

```
CustomersDataTable customers = ...
var query =
  from c in customers.AsIndexed() where c.City == "London" select c;
```

それには、次のステートメントをソースファイルに追加する必要があります。

C#

```
using C1.LiveLinq.AdoNet;
```

使用される **AsIndexed()** は **C1.LiveLinq.AdoNet.AdoNetExtensions.AsIndexed** です。これは、ADO.NET DataSet のデータに対して特に最適化されています。

Copyright © GrapeCity Inc. All rights reserved.

XML データの使用（LiveLinq to XML）

LiveLinq は、メモリ内の XML（LINQ to XML **XDocument** クラスに格納）から直接データにクエリーを実行できます。XML インデックス（通常の LINQ to XML ではサポートされない）をサポートすることで、LINQ to XML のパフォーマンスが劇的に向上します。
LiveLinq to XML を使用するには、ソースファイルに次のコードを追加する必要があります。

C#
```
using C1.LiveLinq.LiveViews.Xml;
```

次に、XML データの上にライブビューをいくつか作成する必要があります。ライブビューなしでデータにクエリーを実行できる LiveLinq to Objects や LiveLinq to DataSet とは異なり、LiveLinq to XML では、クエリーとライブビューが常に一緒に使用されます。ライブビューに適用する場合は、LiveLinq to XML バージョンの LINQ クエリー演算子を使用することになります。そうでない場合、それらは標準の非 LiveLinq クエリー演算子になります。ライブビューの作成を開始するには、拡張メソッド AsLive() を **XDocument** に適用するだけです。

C#
```
XDocument doc = …
View<XDocument> docView = doc.AsLive();
```

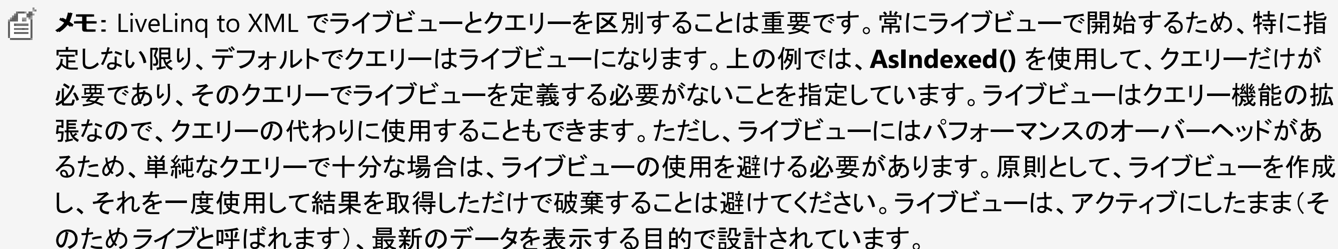
ライブビューが作成されたら、使い慣れた（LINQ to XML）クエリー演算子 **Elements**、**Descendants** など（XmlExtensions クラスを参照）やライブビューでサポートされる標準のクエリー演算子を使用して、ライブビューを定義できます。たとえば、次のコードは、ドキュメント内の注文のコレクションを定義します。

C#
```
View<XElement> orders = docView.Descendants("Orders");
```

このコレクションはライブです。つまり、プログラムによって変更される **XDocument** 内のデータに基づいて自動的に最新の状態に維持されます。その結果、ADO.NET DataTable、DataView などの任意の動的コレクションを使用する場合と同様に、このコレクション内のデータを使用できます。特に、別のセクションで示すように、データにインデックスを作成し、そのインデックスを使用して、クエリーやプログラムによる検索を高速化できます。これは、LiveLinq to XML により、パフォーマンスの低下なくメモリ内の XML データを使用できること、したがってカスタムコレクションクラスを作成したり、ADO.NET などの他のフレームワークを使用して XML データを操作する必要がなくなることを意味しています。LINQ to XML に LiveLinq を追加すれば十分です。

XML データに対していくつかのライブビューが定義できたので、クエリーを実行できます。たとえば、次のコードは、特定の顧客の注文をクエリーします。

C#
```
var query = from Order in orders.AsIndexed()
where (string)Order.IndexedAttribute("CustomerID") == "ALFKI"
```



メモ: LiveLinq to XML でライブビューとクエリーを区別することは重要です。常にライブビューで開始するため、特に指定しない限り、デフォルトでクエリーはライブビューになります。上の例では、**AsIndexed()** を使用して、クエリーだけが必要であり、そのクエリーでライブビューを定義する必要がないことを指定しています。ライブビューはクエリー機能の拡張なので、クエリーの代わりに使用することもできます。ただし、ライブビューにはパフォーマンスのオーバーヘッドがあるため、単純なクエリーで十分な場合は、ライブビューの使用を避ける必要があります。原則として、ライブビューを作成し、それを一度使用して結果を取得しただけで破棄することは避けてください。ライブビューは、アクティブにしたまま（そのためライブと呼ばれます）、最新のデータを表示する目的で設計されています。

連結可能コレクションクラスの使用（LiveLinq to Objects）

任意の連結可能コレクションクラス（データ連結のための変更通知を実装したクラス）を **IndexedCollection<T>** から派生さ

Copyright © GrapeCity Inc. All rights reserved.

れたアダプタクラスでラップした後、LiveLinq クエリーで使用できます。ラップには、**ToIndexed()** 拡張メソッドを適用します。

ToIndexed() を使用することで、LiveLinq をさまざまな一般的なデータコレクションに適用できます。たとえば、ADO.NET Entity Framework の **EntityCollection** クラスや **ObjectResult** クラスは連結可能なので、ADO.NET Entity Framework データを LiveLinq のクエリーやビューで使用できます。

LiveLinq to Objects：IndexedCollection および他のコレクションクラス

LiveLinq による汎用コレクションクラスに対するクエリーは LiveLinq to Objects と呼ばれ、データセットや XML などの特定のデータソースをクエリーする LiveLinq to DataSet や LiveLinq to XML とは区別されます。このため、データセット内のデータをクエリーする場合は LiveLinq to DataSet を、XML データをクエリーする場合は LiveLinq to XML を使用してください。その他の場合については、LiveLinq to Objects の2つのオプションについて既に説明しました。

a. 既存のコレクションクラスがなく、LiveLinq に依存してコレクションクラスを提供する場合は、**IndexedCollection<T>** を使用します。
b. データ連結をサポートしている既存のコレクションクラスには、**ToIndexed() を適用します**。

   また、頻繁には必要とされない高度なオプションもあります。

c. 非標準の機能が必要な場合は、独自のコレクションクラス（通常は **IndexedCollection<T>** から派生）を定義します。

   最後に、高度なオプションと見なすこともできる一方で、実際にはそれほど難しくはないオプションもあります。このオプションを使用すると、ほぼすべてのコレクションに LiveLinq を適用することができます。

d. 変更が発生したことを何らかの方法で通知できるなら、任意の既存のコレクションクラス **C** を LiveLinq で使用可能にすることができます。次に、**C1.LiveLinq.IObservableSource<T>** インタフェースを実装し、ほとんどの機能を既存のコレクションクラスに委ねるアダプタクラスを作成します。**IObservableSource<T>** を実装することで、アダプタクラスに **ToIndexed()** 拡張メソッドを適用することができます。このメソッドは **IIndexedSource<T>** を返します。クエリー演算子を実装する LiveLinq の拡張メソッドは、引数として **IIndexedSource<T>** を受け取ります（**IndexedQueryExtensions** を参照）。

インデックスの作成方法

これで LiveLinq でコレクションをクエリーできるため、コレクションにインデックスを作成する必要があります。そうしないと、LiveLinq は、標準の LINQ より高速にコレクションをクエリーすることができません。

たとえば、次のように、**IIndexedSource<Customer>** を実装するコレクションがあるとします。

C#
```
var customers = new IndexedCollection<Customer>();
```

または、次のようにすることもできます。

C#
```
var customers = CustomersDataTable.AsIndexed();
```

IIndexedSource<T> インタフェースは Indexes コレクションを持つため（実際には唯一のメンバ）、そのコレクションを使用して次のようにインデックスを作成します。

C#
```
var indexByCity = customers.Indexes.Add(x => x.City);
```


Copyright © GrapeCity Inc. All rights reserved.

これで、City に基づいて顧客のインデックスが作成されます。このインデックスは、一旦作成されると、**customers** コレクションが変更されるたびに自動的に管理されます。このメンテナンスにはパフォーマンスコストが伴います。このコストは大きくなく、インデックスメンテナンスは高速で、インデックスが多すぎない限り通常は無視できます。ただし、非常に集中的にコレクションを修正する場合は、検討事項になる可能性があります。

大きなインデックスメンテナンスコストを避けるために、たとえばコレクションにデータを挿入する場合など、インデックス付きのコレクションに大量の変更を行う場合は、**BeginUpdate**/**EndUpdate** メソッドを使用できます。

このインデックスは、次のようなクエリーを極めて高速に実行できます。

C#

```
from c in customers where c.City == "London"
```

これは、LiveLinq が、大きなコレクション全体を横断して必要な項目を検索するのではなく、インデックスを使用して項目に直接アクセスするためです。インデックスは、このような単純なクエリーだけでなく、さまざまなクエリーを高速化でき、（非等価の）範囲条件、結合、グループ化などの LINQ 演算子も高速化できます。インデックスのメリットがあるクエリークラスの詳細については、「LiveLinq クエリーパフォーマンス：インデックスパフォーマンスの調整」を参照してください。

上で説明したように、コレクションにインデックスを追加して明示的にインデックスを作成することは、インデックスのライフタイムを直接制御する場合や、インデックス（**Index<T>** オブジェクト）に直接アクセスしてプログラムで使用する場合（「プログラムでインデックスを使用する方法」を参照）にのみ必要になります。少数の LiveLinq クエリーを最適化する必要があるだけの場合は、代わりに、インデックスを作成する暗黙のメソッドを使用できます。これを LiveLinq クエリーの "*ヒント*" と呼びます。ヒント .Indexed() は、クエリー内のプロパティに適用できる拡張メソッドです。プロパティの値は変更されません。これは、（可能な場合は）そのプロパティにインデックスを作成することを LiveLinq に指示するだけです。したがって、上の例のように、明示的に City に基づいてインデックスを作成する代わりに、次のようにクエリーを記述できます。

C#

```
from c in customers where c.City.Indexed() == "London"
```

これは、まだ作成されていない場合は、City に基づいてインデックスを作成するように LiveLinq に指示します。このヒントは、プロパティの値に影響しません。つまり、**c.City.Indexed()** の値は、**c.City** の値と同じです。

プログラムでインデックスを使用する方法

インデックスは、クエリーの実行を最適化するために LiveLinq によって使用されますが、プログラムからアクセスして使用することもできます。インデックスをコードで直接使用し、**Index<T>** クラスのメソッドを呼び出して、さまざまな検索を高速に実行できます。このように、LiveLinq のインデックスは、LINQ のフレームワークの外部やクエリーの外部でも役立ちます。

たとえば、次のようにして特定の値を検索できます。

C#

```
indexByCity.Find("London")
```

または、次のようにすることもできます。

C#

```
indexByCity.FindStartingWith("L")
```

Index<T> クラスには、**FindGreater**、**FindBetween**、**Join**、**GroupJoin** など、高速な検索、結合、グループ化を実行するために使用できるメソッドもあります。クエリーだけでなく任意のコード内でこれらを使用できます。

ライブビューの作成方法

Copyright © GrapeCity Inc. All rights reserved.

単純なクエリーを考えてみます。

C#

```
from a in A where a.p == 1 select a
```

標準の LINQ では、このクエリーの結果はスナップショットです。結果のコレクションは、クエリーの実行時に形成され、それ以降は変更されません。オブジェクトの1つが変更され、条件を満足しなくなっても、そのオブジェクトは結果のコレクションから削除されません。また、**A** 内のオブジェクトが変更され、条件を満足するようになっても、そのオブジェクトは結果のコレクションに追加されません。このような単純なクエリーの結果でさえ、ライブではなく、動的ではありません。基本コレクション **A** が変化したときに自動的に変化せず、ベースデータと自動的に同期されません。

LiveLinq では、このクエリーに基づいてビューを作成すると、それはライブで動的になります。ベースデータが変化したときに自動的に変化し、ベースデータと自動的に同期されます。

このクエリーからビューを作成するために必要な作業は、拡張メソッド **.AsLive()** を使用することだけです。

C#

```
var view = from a in A.AsLive() where a.p == 1 select a;
```

AsLive() 拡張メソッドは、**AsIndexed()/ToIndexed()** に似ており、**AsIndexed()/ToIndexed()** 拡張メソッドを使用できる場所ならどこでも使用できます。したがって、ライブビューは、LiveLinq to Objects、LiveLinq to XML、LiveLinq to DataSet のすべてのケースでサポートされます（サポートされるクエリー演算子には多少の制限があります。「ライブビューでサポートされるクエリー演算子」を参照）。**AsIndexed()/ToIndexed()** と **AsLive()** の違いは、**AsLive()** がビューを作成することです。したがって、AsLive() の場合は、LiveLinq を使用して、クエリーからビューにデータを挿入することも、初期データを挿入した後のビューを管理することもできます。**AsIndexed()/ToIndexed()** 拡張メソッドは、その前半部分だけを実行し、LiveLinq によるクエリーが有効になります。

上の例は、1つの簡単な Where 条件を示す非常に単純なクエリーです。たとえば、ADO.NET の **DataView** または WPF の **CollectionView** など、ほかにも同様のことを実行できるツールがあります。LiveLinq の威力は、結合を含む大部分の LINQ 演算子をサポートすることにあります。したがって、単純な条件だけでなく、事実上、必要なすべてのクエリーに基づいてライブビューを作成できます。

GUI コントロールをライブビューに連結する方法

LiveLinq ビューを GUI コントロールのデータソースとして使用できます。LiveLinq ビューはデータ連結インタフェースを実装しているため、任意のコントロールを他のデータソースと同様に簡単に LiveLinq ビューに連結できます。次に WinForms の例を示します。

C#

```
View<T> view = from a in A.AsLive() join b in B.AsLive() on a.k equals b.k
     where a.p == 5 select new {a, b};
dataGridView1.DataSource = view;
```

データ連結は長らく、GUI を作成する開発者の大部分に選択されるツールとなってきましたが、その対象は単純なケースに限定されていました。基本コレクション、データセットテーブルなどのベースデータには連結できましたが、派生データや何からのクエリーによって形成されたデータには連結できませんでした。何らかの形成をサポートするツールもありましたが、機能は非常に限定され、ほとんどはフィルタ処理やソートを実行するだけでした。たとえば、結合（極めて普通の例）などのより複雑な方法で派生/形成されたデータがある場合は、データ連結を使用できませんでした（正確に言えば、一度のスナップショット連結とデータ表示にのみ使用でき、変更は不可能）。

LiveLinq は、このような制限を取り去ることで、データ連結を極めて強力な機能にします。これで、連結先となる LINQ の能力を完全に活用できます。

ライブビューがあれば、ほぼ手続き型コードなしで、宣言型データ連結だけを使用して、高度なアプリケーションの GUI 全体を作成できます。

Copyright © GrapeCity Inc. All rights reserved.

非 GUI コードでライブビューを使用する方法

ライブビューの使用は、GUI に限定されません。より一般的には、ライブビューにより、**ビュー指向プログラミング**とも呼ばれる宣言型プログラミングが可能になります。また、非 GUI のバッチ処理コードも、ライブビューを使用して宣言型にすることができます。

一般に、すべてのアプリケーションは何らかのデータに作用し、どのデータも、ベースデータまたは派生データのいずれかと解釈することができます。ベースデータには、アプリケーションが扱う Customers、Orders、Employees などのメインオブジェクトがあります。これらのオブジェクト(特定クラスのすべてのオブジェクトを含むコレクション。クラスの "エクステント" とも呼ばれる)は、ベースデータに何か変更を行う必要がある場合に、アプリケーションによって修正されるオブジェクトです。ただし、アプリケーションがこのベースデータに直接作用したり、クラスのエクステント全体に直接作用することはほとんどありません。通常、アプリケーションは、エクステントのフィルタ処理と形成を行い、エクステントどうしを組み合わせて、特定の操作に必要なデータのスライスを取得します。このデータの形成/フィルタ処理/結合/スライスが、**派生**データ(基底の**ベース**データではなく)を取得するプロセスです。LINQ が世に出る前は、純粋な手続き型コードとそれに伴うすべての複雑な機構によってこのプロセスが実行されていました。LINQ によってこのプロセスは宣言型になり、大きく前進しました。ただし、宣言型と言っても、ライブでも動的でもなく、ベースデータの変化に自動的に反応しません。結果的に、変更に対する反応は宣言型ではありません。プログラマが手続き型の命令コードによって変更に反応する必要があります。LINQ によって複雑性は低下しましたが、変更に対する反応は複雑なままです。変更は至るところで発生するため、変更に対する反応も広汎です。

LiveLinq のライブビューは、変更に対する反応も宣言型にすることで、宣言型の輪を完成させます。これで、GUI だけでなく、アプリケーション全体をほぼ完全に宣言型にすることができます。

ある課題データに対して2つの操作を行います。

1. 保留中の課題、製品の機能(機能に属するすべての課題)、および割り当てに関する情報を使用して、未割り当ての課題を可能な限り従業員に割り当てます。
2. 指定された従業員の未解決の課題に関する情報を収集します。

この2つの操作(もちろん、実際のアプリケーションには、このような操作が多数存在します)はそれぞれ、結合を含むクエリーによって形成されたデータに作用します。どちらの操作も、プログラムの実行中に複数回実行されます。データにこれらの操作および他の操作を実行する間も、データは変化します。これらの操作を実装するライブビューはデータの変化に伴って自動的に変更されるため、操作を単純かつ完全に宣言型に保つことができます。その結果、ビジネス要件の変化に対する堅牢性、信頼性、柔軟性、および変更の容易性が提供されます。

既に知っているライブビューの作成方法に関する知識以外に、このスタイルのプログラミングのために必要なことは何もありません。

Copyright © GrapeCity Inc. All rights reserved.

プログラミングガイド

以下のセクションでは、**LiveLinq** プログラミングに関する情報を提供します。

コードでのエンティティの操作

Entity Framework DataSource（EF DataSource）は、Entity Framework（または RIA）の通常のメソッドやプロパティを使用してコードで自由にエンティティを操作でき、それが **EF DataSource** との組み合わせで正しく機能するという意味で非侵入型です。通常のメソッドを使用してエンティティを追加、削除、および変更できます（そのために、特別な **EF DataSource** オブジェクトモデルは必要ありません）。また、エンティティに対して行った変更は、自動的に **EF DataSource** コレクションに反映されます。たとえば、新しいエンティティを追加するために何か特別な **EF DataSource** メソッドを使用する必要はなく、したがって存在もしません。いつも EF（または RIA）で行っているように、単に新しいエンティティを追加するだけで、対応する **EF DataSource** コレクションにそれが表示されます。**EF DataSource** コレクションが *Where* 条件を持つ場合は、その条件を満足する場合にのみ表示されます。同じことが、エンティティの削除や変更にも当てはまります。通常の EF（または RIA）の方法で削除や変更を行うと、**EF DataSource** に変更が自動的に反映され、したがって連結コントロールにも自動的に反映されます。しかし、厳密に守る必要がある重要な制限が1つあります。実行できる内容は制限されません。（かなりまれな）いくつかのケースでのみ、実行する内容を **EF DataSource** に通知するためのメソッド呼び出しをコードに追加する必要があります。

注意

EF DataSource に通知しないまま、コンテキストを直接クエリーしてサーバーからエンティティをフェッチしてはなりません。

すべてのエンティティは、**EF DataSource** の **GetItems** メソッドの1つを使用してフェッチするか、またはオブジェクトコンテキストを直接クエリーすることによってフェッチする場合は、**ClientScope.AddRef** メソッドを呼び出して、サーバーからエンティティをフェッチしたことを **EF DataSource** に通知する必要があります。

C1DataSource コントロールまたは **ClientViewSource** を使用する場合は、**AutoLoad** = true として暗黙的に、または **ClientViewSource.Load** メソッドを呼び出して明示的にエンティティをフェッチします。どちらの場合も **EF DataSource** を通してフェッチが実行されるため、**EF DataSource** は新しくフェッチしたエンティティを認識することができます。ClientViewSource を使用せずにコード内でエンティティをフェッチする必要がある場合は、**EF DataSource** の **GetItems** メソッド（EntityClientScope.GetItems|keyword=EntityClientScope.GetItems メソッド、RiaClientScope.GetItems|keyword=RiaClientScope.GetItems メソッド）の1つを使用する方法が標準的です。この場合も、**EF DataSource** はフェッチされたエンティティを認識できます。しかし、場合によっては、**EF DataSource** を使用せずに、サーバーに直接クエリーを発行することによってエンティティを取得する必要があることがあります。たとえば、Entity Framework では、次のように、**EF DataSource** によって使用される ObjectContext を取得して、クエリーを作成できます。

```
C#
ObjectContext context = scope.ClientCache.ObjectContext;
// または
ObjectQuery query = ((NORTHWNDEntities)context).Customers;
// または
query = context.CreateObjectSet<Customer>();
// または
query = context.CreateQuery<Customer>("...");
```

RIA サービスでは、次のようなコードになります。

```
C#
DomainContext context = scope.ClientCache.DomainContext;
var query = ((DomainService1)context).Customers;
// または
var entities = context.Load(
```


Copyright © GrapeCity Inc. All rights reserved.

```
((DomainService1)context).GetCustomersQuery()).Entities;
```

クエリーを作成したら、クエリー結果を列挙することで、サーバーから直接エンティティを取得できます。

C#
```
foreach (Customer c in query) /* 何らかの処理 */
```

または、コントロールをクエリー結果に連結します。

C#
```
dataGrid.ItemsSource = query;
```

ClientScope.AddRef を呼び出さずにこれを実行すると、サーバーからエンティティは取得できますが、**EF DataSource** はそれを認識できません。それらのエンティティは、他のエンティティと同じキャッシュに入り、区別できなくなります。したがって、**EF DataSource** は、それらを他のエンティティと同様に管理し、それらを解放したり、不要になれば破棄してしまう可能性があります。これによってエンティティはアクセスできなくなりますが、プログラムではまだ必要かもしれません。したがって、**EF DataSource** が管理するコンテキストに対して **EF DataSource** を使用せずにエンティティをフェッチし、エンティティをフェッチしたことを EF DataSource に通知しなければ、非常に不都合な問題が発生する可能性があります。幸い、その通知は簡単に追加できます。次のように、フェッチしたすべてのエンティティに対して **ClientScope.AddRef** を呼び出すだけです。

C#
```
foreach (Customer c in query)
{
scope.AddRef(c);
// 何らかの処理
}
```

これは、**EF DataSource** に対して、スコープが有効な限りエンティティを解放できないことを通知します。

コードでの ClientView の使用

Entity Framework DataSource(EF DataSource)は、ビジュアルスタイルのプログラミングと "オールコード" スタイルのプログラミングの両方をサポートします。

- ポイント＆クリックで RAD スタイルのアプリケーション開発を行いたい場合は、**C1DataSource** コントロールを使用します。
- **C1DataSource** の使いやすさはそのままに、コードを GUI から分離したい場合は、**ClientViewSource** クラスを使用します。ClientViewSource クラスは GUI から独立しており、任意の GUI プラットフォーム(WPF、Silverlight、WinForms)と組み合わせてコードで使用できます。MVVM アプリケーションのビューモデルレイヤを含めて、GUI とは完全に独立して使用できます。また、**ClientViewSource** は **C1DataSource** が使用するオブジェクトと同じなので、使いやすさという **C1DataSource** の特徴は維持できます(ただし、ビジュアルデザイナなしのコードのみ)。実際に、**C1DataSource** は、単に **ClientViewSource** オブジェクトのコレクションであるだけでなく、ビジュアルデザイナによるサポートを備えています。
- コードを最大限制御する場合は、**EF DataSource** オブジェクトモードの最下位(3番目)にある **ClientView** クラスを使用します。特に MVVM パターンを使用して(ただし、それに限らず)、ピュアコードを指向する場合は、これが推奨のレベルです。プログラミングはやはり容易です。多くは関数型プログラミングを促進する LINQ に基づいており、直感的で表現力豊かです。

ここからは、**ClientView** クラスを使用するプログラミングを中心に説明します。

クライアント側トランザクション

クライアント側トランザクション

Copyright © GrapeCity Inc. All rights reserved.

Entity Framework DataSource(**EF DataSource**)の強力なメカニズムを使用して、サーバーの関与なく、クライアント側で変更をキャンセルすることができます。この操作を「*トランザクション*」と呼びます。これは、複数の変更(作業単位)を全体として実行したりキャンセルすることができるという点で、この操作がデータベーストランザクションの一般的な概念に似ているためです。"全体として" とは、メモリ内のエンティティが不完全に、または一貫性なく変更されることがないということです。このようなトランザクションがデータベーストランザクションには何も影響を与えず、データベーストランザクションとは完全に独立していることを理解しておく必要があります。データベーストランザクションと区別するために、このトランザクションを「*クライアント側トランザクション*」と呼ぶことがあります。
クライアント側トランザクションは、**[キャンセル]/[元に戻す]**ボタン/コマンドの実装時に特に便利です。**EF DataSource** なしで操作をキャンセルしたり元に戻すには、オブジェクトコンテキスト内の変更をすべてキャンセルする必要があります。**EF DataSource** のクライアント側トランザクションでは、変更の一部をキャンセルすることができます。また、トランザクションはネストすることさえできます。たとえば、ダイアログボックスに[キャンセル]ボタンを置き(アプリケーションの他の場所で加えられた、オブジェクトコンテキスト内のすべての変更ではなく、ダイアログボックス内の変更だけをキャンセルするボタン)、そのダイアログボックスから、独自の **[キャンセル]**ボタンを持つ別のダイアログボックスを開くことができます。子ダイアログ内にネストされた(子)トランザクションを使用することで、その **[キャンセル]**ボタンから、子ダイアログボックスに加えられた変更だけをキャンセル(ロールバック)することができます。ユーザーは、親ダイアログボックスのデータの編集に戻り、親トランザクションを使用して、そのダイアログボックスで加えられた変更を受け入れたりキャンセルすることができます。

クライアント側トランザクションを最も簡単に使用するには、トランザクションをライブビューに関連付けます。たとえば、次のようにライブビューに連結されたデータグリッドがある場合は、

C#

```
var ordersView = from o in customer.Orders.AsLive()
                  select new OrderInfo
                  {
                      OrderID = o.OrderID,
                      OrderDate = o.OrderDate,
                  };
dataGrid1.ItemsSource = ordersView;
```

次のようにトランザクションを作成して、ビューに関連付けることができます。

C#

```
var transaction = _scope.ClientCache.CreateTransaction();
 ordersView.SetTransaction(transaction);
```

子(ネストされた)トランザクションを作成するには、メソッド **ClientCacheBase.CreateTransaction** を呼び出す代わりに、**ClientTransaction** コンストラクタを使用し、パラメータとして親トランザクションを渡します。

C#

```
var transaction = new ClientTransaction(parentTransaction);.
```

View.SetTransaction|keyword=SetTransaction メソッド を呼び出してトランザクションをビューに関連付けたら、そのビューからデータ連結を通して加えられた変更(前述の例のように、エンドユーザーがビューに連結されたグリッドで変更を加えた場合など)やコードからプログラムを通して加えられた変更がすべて管理されます。**ClientTransaction.Rollback** を呼び出してトランザクションをロールバックすると、そのトランザクションによって管理されていた変更がキャンセルされます。

GUI コントロールをオブジェクトのコレクションではなく単一のオブジェクトに連結する場合もよくあります。単一のオブジェクトをライブビューで表すことはできないため、View.SetTransaction|keyword=SetTransaction メソッド を利用することはできません。この場合は、**ClientTransaction.ScopeDataContext** メソッドを使用します。WPF および Silverlight では、このメソッドを

 Copyright © GrapeCity Inc. All rights reserved.

使用して **DataContext** を設定できます。XAML で指定されたデータ連結に対しては、次のように使用します。

DataContext = transaction.ScopeDataContext(order);

結果の **DataContext** は、元の DataContext をラップし、同じデータ連結を実行しますが、そのデータ連結を通して加えられた変更がすべて "トランザクション" として管理されるという利点があります。このため、必要に応じて、変更をロールバックすることができます。

WinForms の場合は、同じ **ScopeDataContext** を使用して、次のように結果のオブジェクトに連結することができます。

C#

```
var dataContext = transaction.ScopeDataContext(order);
textBox.DataBindings.Add(new Binding("Text", dataContext, "OrderDate"));
```

最後に、ライブビューやデータ連結ではなく、コードを使用して一部のエンティティを変更（または追加/削除）し、その変更をトランザクションで管理する場合があります。それには、**ClientTransaction.Scope()** メソッドを使用します。このメソッドを使用すると、トランザクションのスコープが開きます。トランザクションのスコープにある間にエンティティを変更すると、変更がそのトランザクションによって管理されます。このメソッドは、次に示すように、終了時に自動的にスコープを閉じてくれる "using" 構文と共に使用するように設計されています。

C#

```
using (transaction.Scope())
 {
     Customer.Orders.Add(order);
 }
```

ここで説明したトランザクションを使用する3つの方法については、**EF DataSource** に付属する **Transactions** サンプルプロジェクトで具体例が示されています。このサンプルプロジェクトでは、子（ネストされた）トランザクションを使用して、**[キャンセル]**ボタンを持つフォームを、**[キャンセル]**ボタンを持つ別のフォームの内部に実装する方法についても説明しています。

仮想モード

仮想モードは、GUI コントロールと大規模なデータセットの間の遅延のないデータ連結を **Entity Framework DataSource（EF DataSource）**に提供する技術です。コードを記述したり、ページングを使用する必要はなく、プロパティを1つ設定するだけで、この技術を利用できます。そのプロパティは **ClientViewSource.VirtualMode** です。これには次の3つの値を設定できます。

- **None:**仮想モードは無効になります。これがデフォルト値です。
- **Managed:**仮想モード（サーバーからのデータのフェッチ）は、グリッドコントロールによって制御されます。仮想モードを制御するグリッドコントロールを指定するには、グリッドにある拡張（添付）プロパティを設定します。広く使用されているグリッドはほぼサポートされていますが、サポートされていないグリッドもあります（以下のリストを参照）。サポートされているグリッドで仮想モードを使用する場合は、**Unmanaged** ではなく **Managed** オプションを使用します。**Managed** オプションを使用すると、使用するグリッドに合わせてパフォーマンスが最適化され、最適なパフォーマンスを得ることができます。グリッドコントロールのほかに単純なコントロール（TextBox コントロールなど）が同じデータソースに連結されていてもかまいません。ただし、仮想モードを制御中のグリッドのほかに、"複雑な" 連結コントロール（別のグリッドやリストなど）は使用しないでください。

  Managed モードは、現在、次のグリッドコントロールでサポートされています。

  - **ComponentOne**：C1FlexGrid（WinForms、WPF、Silverlight）、C1DataGrid（WPF、Silverlight）。
  - **Microsoft**：DataGridView（WinForms）、DataGrid（WPF）、DataGrid（Silverlight。ただし、Microsoft DataGrid for Silverlight の問題については、下記を参照）。
- **Unmanaged**：仮想モード（サーバーからのデータのフェッチ）は、連結されているコントロールのタイプに関係なく、データソース自体によって制御されます。上にリストしたサポートされているグリッドに含まれないグリッド/リストなどのコントロールで仮想モードを使用する場合は、このオプションを使用します。ただし、データセット全体ではなく、現在のレコードにのみ連結する TextBox などの "単純な" コントロールは、**Managed** モードと **Unmanaged** モードの制限な

Copyright © GrapeCity Inc. All rights reserved.

く使用することができます。**Unmanaged** のパフォーマンスは、**Managed** と同様に優れています。ただし、Unmanaged にはいくつかの制限があります。詳細については、「Unmanaged 仮想モードの制限」を参照してください。これらの制限はそれほど強いものではありませんが、自動的にはチェックされません。このため、**Unmanaged** オプションを使用する場合は、制限に該当していないことを事前に確認してください。また、**Unmanaged** モードでは、どの時点で GUI コントロールに表示されるページサイズより大きな数を **PageSize** プロパティに設定しておく必要があります。Managed モードでは、PageSize は無視されます。

Unmanaged 仮想モードの制限

Managed オプションと **Unmanaged** オプションの違いは、データをサーバーから取得する際に見られます。**Managed** モードでは、"仮想モードを制御中" のグリッドがデータを必要とするとき、たとえばユーザーがグリッドをスクロールしたりグリッド内を移動したときに、データが取得されます。**Unmanaged** モードでは、連結コントロールがデータ要求を行うたびにサーバーからデータがフェッチされます（データがキャッシュにない場合）。このとき、どのコントロールがどのような目的でデータを要求したかは考慮されません。データソースは、データ要求に含まれる各行の前後の**ClientViewSource.PageSize** 分の行をフェッチします。ただし、**Managed** モードとは異なり、このモードでは、連結コントロールに表示される行が不明なので、これらの行は保持されません。データソースは、現在の行と最後に要求された行だけを保持します（それらの直前の**ClientViewSource.PageSize**分の行と直後の **ClientViewSource.PageSize**分の行を含む）。他の行は保持されません。つまり、**Entity Framework DataSource** がキャッシュをクリーンアップ/圧縮する必要があるときに、解放されてキャッシュから削除される可能性があります。ただし、多くのケースで、これが問題になることはありません。また、通常は、次の点に注意することで問題を回避できます。

Unmanaged 仮想モードでは、1つのデータソースに複数のスクロール可能コントロールを連結しないでください。 TextBox などの単純なコントロール（1つの行に連結）は、必要な数だけ連結することができます。また、任意のグリッド、リスト、または他の "スクロール可能" なコントロールを（1つの行ではなくデータセット全体に）連結することもできます。ただし、データセット内の互いに **ClientViewSource.PageSize** より離れた2つの位置（行）にスクロールすることがある2つのスクロール可能なコントロールを連結してはなりません。そうすると、最後に要求された一方の行だけが保持され、もう一方の行は **Entity Framework DataSource** によって（予期できないタイミングで）解放されてしまうことがあるためです。

その他の仮想モードの制限

- 仮想モードで1つのデータソースに2つの独立してスクロール可能なコントロール（データセット内の別の場所に独立してスクロール可能なグリッドまたはリスト）を連結することはサポートされません。これは、**Unmanaged** モードだけでなく、**Managed** モードでも同じ理由でサポートされていません。2つのモードの違いは、単に **Managed** モードでは影響がすぐに現れ、したがって動作に予測不能な要素がないため、危険性がより小さいということだけです。メインの "制御中" の連結コントロールのほかに、別のスクロール可能なコントロールが同じデータソースに連結されている場合、そのコントロールは "制御中" ではありません。つまり、そのコントロール内でスクロールしても、データはフェッチされません。このため、メインコントロールに表示されていない行にスクロールすると、空の行が表示されます。
- **Managed** モードでも **Unmanaged** モードでも、（1つのレコードではなく）データセット全体に連結されているグリッド、リストなどのコントロールで、データソースのすべての行に対して何らかの処理を一度に実行してはなりません。言い換えると、データソースのすべての行をループして、各行に何らかの処理を行うようなアクションをコントロールのコードで実行してはなりません。これは、単純に、仮想モードで扱う行数が、数千行から数百万行にまで、極めて多くなることがあるためです。実際、その数は無制限です。データソース内の行数が少ないという仮定には依存しない、適切に設計されたコントロールは、**Entity Framework DataSource（EF DataSource）**の仮想モードで正常に動作します。ただし、小規模なデータソース向けに特別に設計されたコントロール、たとえばすべての行をループするようなコントロールを、大量の行を含むデータソースと一緒に使用すると、長時間の遅延が発生する可能性があります。C1FlexGrid および C1DataGrid（WinForms/WPF/Silverlight 用）、Microsoft DataGridView（WinForms）、Microsoft DataGrid for WP はいずれも正常に動作します。一方、Microsoft DataGrid for Silverlight（現行バージョンの Silverlight 4）は、すべての行をループしてコントロールの高さを計算するため、**EF DataSource** の仮想モードに推奨されません。

Copyright © GrapeCity Inc. All rights reserved.