Data Providers for ADO.NET

Zen v16

Activate Your Data™

目次

Actian Zen ADO.NET データ プロバイダーへようこそ

このドキュメントでは、次の Actian Zen データ プロバイダーについて説明します。

- Zen ADO.NET データ プロバイダー
- Zen ADO.NET Core データ プロバイダー
- Zen ADO.NET Entity Framework データ プロバイダー
- Zen ADO.NET Entity Framework Core データ プロバイダー

Zen ADO.NET データ プロバイダーとは

Zen ADO.NET データ プロバイダーは、マネージ コードのみで構築されている管理データ プロバイダーです。データ プロバイダーは、ネイティブなワイヤ プロトコル プロバイダーです。つまり、プロバイダーは、データベース クライアントの形式のアンマネージ コード（.NET Framework 外のコード）を呼び出す必要はありません。ただし、アプリケーションが Microsoft Distributed Transaction Coordinator（MS DTC）でコーディネイトされたトランザクションに参加している場合を除きます。

Zen ADO.NET データ プロバイダーは、Zen データベース エンジンに接続できるようにします。32 ビットおよび 64 ビットの両方の .NET で機能し、Zen がサポートするすべての Windows プラットフォームでサポートされます。

Zen でインストールされる ADO.NET データ プロバイダーおよび SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーも参照してください。

本リリースでの新機能

Zen v16 リリースの Zen ADO.NET データ プロバイダーは、次の点で新しくなっています。

- ADO.NET 4.4 はサポートされなくなりました。ADO.NET 4.6 サポートが追加されました。
- 拡張機能

    - Zen データ プロバイダーは、パスワードをより安全にサーバーに送信するように拡張されました。
    - Zen データ プロバイダーは、Zen v16 をサポートするように拡張されました。
    - Zen Core データ プロバイダーおよび Actian Zen Entity Framework Core データ プロバイダーは、.NET 8.0 をサポートするように拡張されました。
    - Zen Entity Framework Core データ プロバイダーは、Entity Framework Core 8.0 をサポートするように拡張されました。
- 変更された動作
    - Zen データ プロバイダーは Visual Studio 2017 をサポートしなくなりました。
    - Zen Entity Framework Core データ プロバイダーは、.NET 8.0 より前の .NET バージョンをサポートしなくなりました。
    - Microsoft Enterprise Libraries は製品ライフ サイクルを終了しており、今後更新情報を受け取ることはありません。そのため、Zen データ プロバイダーはそれらをサポートしなくなりました。

このガイドの使用法

このガイドは、お使いのオペレーティング システムとその各種コマンド、ディレクトリの定義、およびエンド ユーザーアプリケーションからデータベースへのアクセス方法に関して精通しているユーザーを対象としています。

このガイドには以下の情報が記載されています。

- クイック スタートでは、.NET データ プロバイダーを使用したデータベースへの接続について説明します。
- データ プロバイダーの使用では、Zen データ プロバイダーと一緒に .NET アプリケーションを使用する方法、また .NET 環境で .NET アプリケーションを開発する方法について説明します。
- 高度な機能では、接続プール、ステートメント キャッシュ、セキュリティの設定、および Zen Bulk Load の使用を含む、データ プロバイダーの高度な機能について説明します。
- ADO.NET データ プロバイダーでは、Zen Entity Framework データ プロバイダーの接続文字列オプション、データ型などの情報について説明します。

- Zen ADO.NET Core データ プロバイダーでは、Visual Studio で Zen ADO.NET Core DLL を使用してアプリケーションおよび UWP アプリケーションを作成する方法について説明します。
- Zen ADO.NET Entity Framework データ プロバイダーでは、Zen ADO.NET Entity Framework データ プロバイダーの機能について説明します。Zen ADO.NET Entity Framework データ プロバイダーのエンティティ データ モデルの作成方法を説明します。
- Zen ADO.NET Entity Framework Core データ プロバイダーでは、Actian Zen ADO.NET Entity Framework Core データ プロバイダーについて説明し、それを構成および使用する手順を提供します。
- Visual Studio での Zen データ プロバイダーの使用では、Visual Studio 内で Zen データ プロバイダーおよび Performance Wizard を使用する方法について説明します。
- サポートされる .NET オブジェクトでは、Zen データ プロバイダーでサポートされる .NET パブリック オブジェクト、プロパティ、メソッドについて説明します。
- スキーマ情報の入手では、Zen データ プロバイダーでサポートされるスキーマ コレクションについて説明します。
- .NET の SQL エスケープ シーケンスでは、Zen データ プロバイダーでサポートされるスカラー関数について説明します。ご使用のデータ ストアが、ここで説明しているすべての関数をサポートしていないことがあります。
- ロック レベルと分離レベルでは、ロック レベルと分離レベルについて、およびこれらの設定が取得するデータにどのように影響するかについて説明します。
- パフォーマンスの最適化を図る .NET アプリケーションの設計では、アプリケーションのコードを最適化して、パフォーマンスを向上させる方法を説明します。
- .edmx ファイルの使用では、EDM レイヤーに Extended Entity Framework の機能を提供するために、.edmx ファイルに加える必要のある変更について説明します。
- バルク ロード構成ファイルには、バルク ロード操作中に作成されるファイルのサンプルがあります。
- IANA コード ページ マッピングには、最も広範に使用されている IBM コード ページの、IANA コード ページ名へのマッピングが示されています。

メモ： このガイドでは、参考情報が掲載されているオンライン リンクを記載しています。Web コンテンツは頻繁に変わるものであるため、ここに記載されているリンクは、このガイドの発行後に変わる可能性もあります。

クイック スタート

ここでは、Zen ADO.NET データ プロバイダーをインストールした後で、データベースに接続する基本的な操作について説明します。

- Zen でインストールされる ADO.NET データ プロバイダー
- SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダー
- 基本的な接続文字列の定義
- データベースへの接続
- Zen ADO.NET Entity Framework データ プロバイダーの使用

Zen ADO.NET データ プロバイダーの機能を最大限利用するには、当ドキュメント内のほかの Zen ADO.NET 関連トピックもお読みください。

Zen でインストールされる ADO.NET データ プロバイダー

このセクションでは、Zen v16 と一緒にインストールされる Zen ADO.NET データ プロバイダーでサポートされる .NET Framework バージョンについて説明します。SDK をダウンロードすることで提供される .NET Core および Entity Framework Core 用の Zen ADO.NET データ プロバイダーの説明については、SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーを参照してください。

Zen v16 は、ADO.NET データ プロバイダー 4.5 および 4.6 の 2 つのバージョンを提供します。デフォルトで、すべてのバージョンがデータベース エンジンと一緒にインストールされます。

ADO.NET をカスタマイズしないで使用しているのであれば、以前のバージョンの .NET Framework と Zen データ プロバイダー用に作成されたコードは、Zen データ プロバイダー 4.5 および 4.6 と互換性があります。

サポートされる .NET Framework のバージョン

次の表に示すように、Zen ADO.NET データ プロバイダー 4.5 および 4.6 では、Microsoft .NET Framework や Microsoft Entity Framework との組み合わせもサポートされるようになりました。表の各行で、これら 3 製品のサポートされるバージョンの互換性のある組み合わせを表しています。

Zen データ プロバイダー	バージョン	名前空間	アセンブリ ファイル名 (Zen でインストール)	Microsoft .NET Framework	Microsoft Entity Framework
ADO.NET	4.5	Pervasive.Data.SqlClient	Pervasive.Data.SqlClient.dll	2.0、3.0、3.5、3.5 SP1、4.5、4.5.1、4.5.2、4.6、4.6.1、4.6.2、4.7、4.7.1、4.7.2、4.8	—

Zen データ プロバイダー	バージョン	名前空間	アセンブリ ファイル名（Zen でインストール）	Microsoft .NET Framework	Microsoft Entity Framework
ADO.NET	4.6	Pervasive.Data.SqlClient	Pervasive.Data.SqlClient.dll	2.0、3.0、3.5、3.5 SP1、4.5、4.5.1、4.5.2、4.6、4.6.1、4.6.2、4.7、4.7.1、4.7.2、4.8	—
ADO.NET Entity Framework	4.5	Pervasive.Data.SqlClient.Entity	Pervasive.Data.SqlClient.Entity.dll	4.5、4.5.1、4.5.2、4.6、4.6.1、4.6.2、4.7、4.7.1、4.7.2、4.8	6.1、6.1.1、6.1.2
ADO.NET Entity Framework	4.6	Pervasive.Data.SqlClient.Entity	Pervasive.Data.SqlClient.Entity.dll	4.5、4.5.1、4.5.2、4.6、4.6.1、4.6.2、4.7、4.7.1、4.7.2、4.8	6.1、6.1.1、6.1.2

注記

アプリケーションで Zen ADO.NET Entity Framework プロバイダー 4.5 または 4.6 を使用するには、.NET Framework 4.5 以上を使用しなければなりません。

一覧に示したバージョンはすべて、.NET Framework の 32 ビットと 64 ビットのどちらのバージョンにも適用されます。

Zen ADO.NET Entity Framework プロバイダー 4.5 および 4.6 では、Microsoft Entity Framework 6.1.x がサポートされています。

Zen ADO.NET データ プロバイダーの詳細については、各プロバイダーのトピックを参照してください。

SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダー

Zen v16 でインストールされる ADO.NET データ プロバイダーのほかに、.NET Standard 2.0 準拠のアプリケーションをサポートするために追加のプロバイダーを入手することができます。次の表にこれらのプロバイダーを示します。プロバイダーは、弊社 Web サイトでダウンロード可能な SDK に含まれる NuGet パッケージとして入手できます。Zen でインストールされるプロバイダーのバージョンと同様に、準拠アプリケーションの 2 つのバージョンを示します。

Zen データ プロバイダー	バージョン	名前空間	アセンブリ ファイル名	.NET	EF Core	NuGet パッケージ (「メモ」を参照)	SDK ダウンロード
ADO.NET Core	4.5	Pervasive.Data.SqlClient	Pervasive.Data.SqlClientStd.dll	Core 2.1、6.0、7.0	—	Pervasive.Data.SqlClientStd.4.5.0.<build>.nupkg	Zen-SDK-AdoNetDataProvider-4.5-NetStandard-Windows-noarch-<version>.zip
ADO.NET Core	4.6	Pervasive.Data.SqlClient	Pervasive.Data.SqlClientStd.dll	Core 2.1、6.0、7.0	—	Pervasive.Data.SqlClientStd.4.6.0.<build>.nupkg	Zen-SDK-AdoNetDataProvider-4.6-NetStandard-Windows-noarch-<version>.zip
ADO.NET Entity Framework Core	4.5	Actian Entity Framework Core.Zen	Actian.EntityFrameworkCore.Zen.dll	Core 2.1、6.0	3.1、6.0	Actian.EntityFrameworkCore.Zen.4.5.0.<build>.nupkg	Zen-SDK-AdoNetDataProvider-4.5-NetStandard-Windows-noarch-<version>.zip
ADO.NET Entity Framework Core	4.6	Actian Entity Framework Core.Zen	Actian.EntityFrameworkCore.Zen.dll	Core 2.1、6.0	3.1、6.0	Actian.EntityFrameworkCore.Zen.4.6.0.<build>.nupkg	Zen-SDK-AdoNetDataProvider-4.6-NetStandard-Windows-noarch-<version>.zip

メモ： 関連する NuGet パッケージは、表に記載されているダウンロード可能な SDK でのみ入手できます。

Zen ADO.NET Entity Framework Core データ プロバイダーには、同じバージョンの Zen ADO.NET Core データ プロバイダーをアプリケーション プロジェクトに追加する必要があります。

Zen ADO.NET Core データ プロバイダーの詳細については、各プロバイダーのトピックを参照してください。

基本的な接続文字列の定義

データ プロバイダーでは、特定のデータベース サーバーへの接続に必要な情報の指定に接続文字列を使用します。この接続情報は接続文字列オプションで定義されます。

Zen ADO.NET Entity Framework データ プロバイダーは、Entity Framework ウィザードで既存の接続を指定するか、または新しい接続を定義することができます。Zen ADO.NET Entity Framework は接続文字列に含まれている情報を使用して、Entity Framework をサポートする基となる Zen ADO.NET データ プロバイダーに接続します。接続文字列には、必要なモデルおよびマッピング ファイルに関する情報も含まれています。データ プロバイダーは、モデルにアクセスしたり、メタデータをマップしたり、データ ソースに接続したりする場合に接続文字列を使用します。

接続文字列オプションは次のような形式です。

```
" オプション名 = 値 "
```

接続文字列オプション値の各ペアはセミコロンで区切ります。たとえば、次のようになります。

```
"Server DSN=DEMODATA;UID=test;PWD=test;Host=localhost"
```

サポートされる接続文字列オプションの詳細については、**接続文字列プロパティ**を参照してください。

注記

- 接続文字列オプション名内のスペースは任意です。
- 接続文字列オプションでは大文字と小文字が区別されません。たとえば、User ID と user id は同じです。ただし、User ID や Password などいくつかのオプションの値の中には大文字と小文字が区別されるものもあります。
- 接続文字列にポート番号が指定されていない場合、データ プロバイダーは 1583 を使用します。これはデフォルトのポート番号です。

最低限必要な接続文字列オプション

次の表は、Zen サーバーへの接続に必要な最低限のオプションの名前とその説明です。

オプション	説明
Server DSN	接続するサーバーのデータ ソース名を、`DEMODATA` のように指定します。
Host	接続する Zen サーバーの名前または IP アドレスを指定します。たとえば、`Accountingserver` などのサーバー名や、`199.226.22.34`（IPv4）または `1234:5678:0000:0000:0000:0000:9abc:def0`（IPv6）などの IP アドレスを指定できます。 デフォルトの初期値は `localhost` です。

データベースへの接続

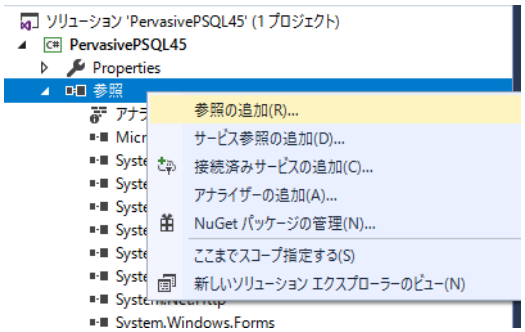
データ プロバイダーをインストールすると、接続文字列を使ってアプリケーションからデータベースに接続できるようになります。接続文字列オプションの一覧については、接続文字列プロパティを参照してください。

メモ：アプリケーションで Zen ADO.NET Entity Framework を使用する場合は、Entity Data Model ウィザードを使って新しい接続を作成したり、既存の接続を使用したりすることができます。詳細については、モデルの作成を参照してください。

例：プロバイダー固有のオブジェクトの使用

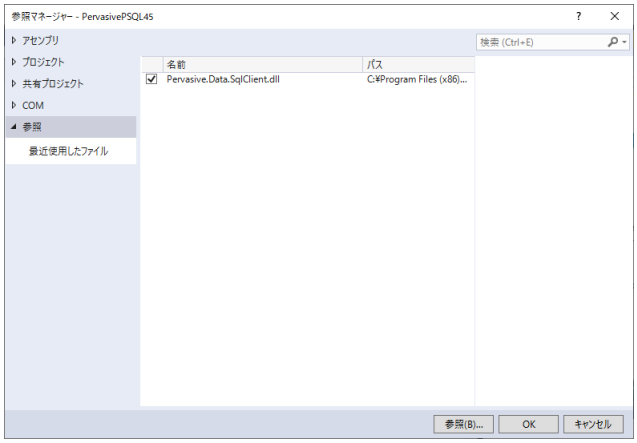
次の例はプロバイダー固有のオブジェクトを使用した例です。C# を使って Visual Studio で開発したアプリケーションから Zen ADO.NET データ プロバイダーを使用してデータベースに接続しています。

1. ソリューション エクスプローラーで［**参照**］を右クリックし、［**参照の追加**］を選択します。

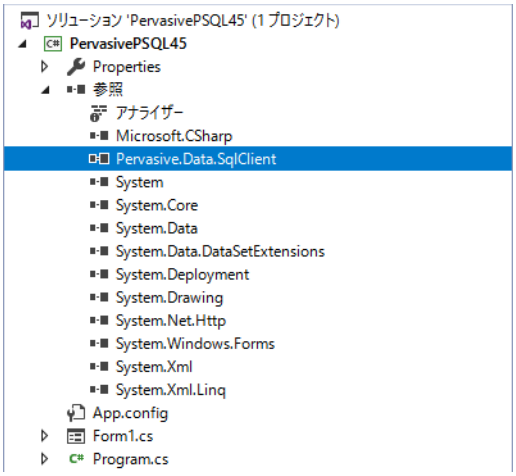


2. ［**参照マネージャー**］ウィザードで［**参照**］ボタンをクリックし、Zen データ プロバイダー アセンブリが含まれるフォルダーに移動します。

3. "**Pervasive.Data.SqlClient.dll**" を選択して［**追加**］をクリックします。［参照マネージャー］ウィザードの［**参照**］タブで、［**最近使用したファイル**］項目に Zen データ プロバイダー アセンブリが表示されます。



4. それを選択し、［**OK**］をクリックします。これで、ソリューション エクスプローラーに Zen データ プロバイダーが含まれるようになります。



5. 次の C# コード例のように、Zen データ プロバイダーの名前空間をアプリケーションの先頭に追加します。

```
// Zen へアクセスします
using System.Data;
using System.Data.Common;
using Pervasive.Data.SqlClient;
```

6. サーバーの接続情報と例外処理のコードを追加します。

```
PsqlConnection DBConn = new PsqlConnection("Server DSN=DEMODATA;Host=localhost");
try
{
```

```
   // 接続を開きます
   DBConn.Open();
   Console.WriteLine(" 接続成功 !");
}
catch (PsqlException ex)
{
   // 接続に失敗しました
   writer.WriteLine(ex.Message);
}
```

7. 接続を閉じます。

```
// 接続を閉じます
DBConn.Close();
```

例：共通プログラミング モデルの使用

次の例は、C# と共通プログラミング モデルを使用して Visual Studio で開発したアプリケーションから Zen データベースに接続する方法を示します。

1. アプリケーションの先頭部分を調べます。ADO.NET 名前空間が記述されていることを確認します。

```
// ファクトリを使用して Zen にアクセスします
using System.Data;
using System.Data.Common;
```

2. サーバーの接続情報および例外処理のコードを追加し、接続を閉じます。

```
DbProviderFactory
factory=DbProviderFactories("Pervasive.Data.SqlClient");
DbConnection Conn = factory.createConnection();
Conn.CommandText = "Server DSN=DEMODATA;Host=localhost;";
try
{
   Conn.Open();
   Console.WriteLine(" 接続成功 !");
}
catch (Exception ex)
{
   // 接続に失敗しました
   Console.WriteLine(ex.Message);
}
// 接続を閉じます
Conn.Close();
```

例：Zen Common Assembly の使用

共通プログラミング モデルに適合するアプリケーションで Zen Bulk Load などの機能を使用したい場合は、Zen Common Assembly を含めることを選択できます。アプリケーションで Zen Bulk Load ユーザーを使用する方法については、Zen Bulk Load の使用を参照してください。

次の例は、C# と共通プログラミング モデルを使用して Visual Studio で開発したアプリケーションで Zen Common Assembly を使用する方法を示します。

1. アプリケーションの先頭部分を調べます。.NET Framework および Zen データ プロバイダーの名前空間が記述されていることを確認します。

```
// ファクトリを使用して Zen にアクセスします
using System.Data;
using System.Data.Common;
using Pervasive.Data.Common;
```

2. サーバーの接続情報および例外処理のコードを追加し、接続を閉じます。

```
// このコードは、バルク コピー操作を
// あるデータベースから別のデータベースに対して行います
DbProviderFactory Factory = DbProviderFactories.GetFactory("Pervasive.Data.SqlClient");
DbConnection sourceConnection = Factory.CreateConnection();
sourceConnection.ConnectionString = "Host=localhost;Server DSN=DEMODATA;";

sourceConnection.Open();

DbCommand command = sourceConnection.CreateCommand();
command.CommandText = "SELECT * FROM test";
DbDataReader reader = command.ExecuteReader();

DbConnection destinationConnection = Factory.CreateConnection();
destinationConnection.ConnectionString =
    "Host= ntsl2003b;Server DSN=DEMODATA";
destinationConnection.Open();

DbBulkCopy bulkCopy = new DbBulkCopy(destinationConnection);
   bulkCopy.DestinationTableName = "test";
   try
   {
      bulkCopy.WriteToServer(reader);
      }// 試行終了
   catch (DbException ex)
   {
       Console.WriteLine( ex.Message );
   }//catch の終了
   finally
   {
      reader.Close();
      MessageBox.Show(" 終了 ");
   }// 完了
```

Zen ADO.NET Entity Framework データ プロバイダーの使用

Entity Data Model ウィザードは、エンティティ データ モデル（EDM）でコンポーネントを定義するための質問をします。その後、ウィザードは Visual Studio でデータのモデルを作成し、そのモデルでコンポーネントの値を自動的に設定します。ウィザードによる EDM の作成方法については、.edmx ファイルの使用を参照してください。

別の方法としては、Visual Studio でほかのツールを使って、値と接続文字列を手動で定義することもできます。

Provider は、EDM のストレージ モデル ファイルに含まれる Schema 要素の属性です。ストレージ モデル ファイルは、ストア スキーマ定義言語（SSDL）で書かれています。

Entity Data Model ウィザードは、Zen ADO.NET Entity Framework データ プロバイダーが選択されているとき、値の割り当てを行います。エンティティ データ モデルを手動で定義することを選ぶ場合は、次の例で示すように、文字列 `Pervasive.Data.SqlClient` を Schema 要素の Provider 属性に代入します。

```
<Schema Namespace="AdventureWorksModel.Store" Alias="Self" Provider="Pervasive.Data.SqlClient"
ProviderManifestToken="Zen" xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/
EntityStoreSchemaGenerator" xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">
```

データ プロバイダーの使用

Zen データ プロバイダーを使用すれば、.NET 対応アプリケーションやアプリケーション サーバーへのデータ アクセスが可能になります。データ プロバイダーはインターネットまたはイントラネット経由で、主要なデータ ストアへの高パフォーマンスなポイント ツー ポイント アクセスおよび *n* 層アクセスを提供します。データ プロバイダーは .NET 環境向けに最適化されているため、現在使用しているシステムに .NET 技術を組み込み、その機能を拡張してパフォーマンスを向上させることができます。

接続プール、ステートメント キャッシュ、セキュリティの設定、Zen Bulk Load、および診断サポートなどの高度な機能については、高度な機能を参照してください。

標準 Zen ADO.NET 環境における Zen ADO.NET データ プロバイダーの使用法については、ADO.NET データ プロバイダーを参照してください。

Zen ADO.NET Entity Framework でのデータ プロバイダーの使用法については、Zen ADO.NET Entity Framework データ プロバイダーを参照してください。

データ プロバイダーについて

Zen データ プロバイダーはマネージ コードのみで構築されています。つまり、完全に共通言語ランタイム（CLR）の内部で、実行およびデータベースへの接続が行えます。

クライアント ライブラリや COM コンポーネントなどネイティブ オペレーティング システムで実行するコードはアンマネージ コードと言います。マネージ コードとアンマネージ コードは 1 つのアプリケーション内に混在させることができます。ただし、アンマネージ コードは共通言語ランタイムの外部にまで影響が及ぶため、現実的には複雑になり、パフォーマンスも低下します。また、セキュリティの危険にさらすことにもなりかねません。

Zen データ プロバイダーでサポートされる .NET Framework バージョンおよび Entity Framework バージョンの詳細については、サポートされる .NET Framework のバージョンを参照してください。

接続文字列の使用

接続の動作は、接続文字列や PsqlConnection オブジェクトのプロパティを使って定義することができます。

しかし、接続文字列内の値の設定を接続プロパティによって変更することはできません。

接続文字列の基本形式は、セミコロンで区切られた一連の「キーワード / 値」のペアを含んでいます。次に、Zen データ プロバイダー用の単純な接続文字列のキーワードと値の例を示します。

```
"Server DSN=SERVERDEMO;Host=localhost"
```

ガイドライン

接続文字列を指定する際には、次のガイドラインを用います。

- 接続文字列オプション名内のスペースは必須です。
- 接続文字列オプションでは大文字と小文字が区別されません。たとえば、Password と password は同じです。ただし、User ID や Password などのオプションの値には大小文字が区別されるものもあります。
- セミコロン、一重引用符、または二重引用符の入った値を含めるには、その値を二重引用符で囲みます。値にセミコロンと二重引用符の両方が入っている場合は、一重引用符を使って値を囲みます。
- 値が二重引用符で始まる場合も一重引用符を使用できます。逆に、値が一重引用符で始まる場合は二重引用符を使用します。値に一重引用符と二重引用符のどちらも入っている場合は、値を囲むのに使用する文字が値の中に現れるたびに、その文字を 2 つ重ねる必要があります。
- 先頭または末尾のスペースを文字列値に含めるには、その値を一重引用符または二重引用符で囲む必要があります。整数値や、ブール値、列挙値を一重引用符または二重引用符で囲んでも、値の前後のスペースは無視されます。ただし、文字列リテラルのキーワード内あるいは値内のスペースは維持されます。接続文字列内で一重引用符または二重引用符を使用する場合、それが値内の先頭または末尾の文字でなければ、区切り文字なしで使用できます（たとえば、`Data Source= my'Server` または `Data Source= my"Server`）。
- 接続文字列オプションの値で特殊文字を使用することができます。特殊文字をエスケープするには、その値を一重引用符または二重引用符で囲みます。

- また、等号文字（=）の繰り返しを接続文字列で使用することもできます。たとえば、次のように指定します。

```
Initialization String=update mytable set col1 == 'foo'"
```

- 接続文字列に無効な接続文字列オプションが含まれている場合には、接続はエラーを返そうとします。たとえば、Alternate Servers を定義していないのに Load Balancing の値を指定すると、エラーが返されます。
- 接続文字列に接続文字列オプションが重複して含まれている場合には、データ プロバイダーは接続文字列内で最後に現れる接続文字列オプションを使用します。たとえば、次に示す接続文字列では、Connection Timeout が異なる値で 2 回現れます。データ プロバイダーは 2 番目の値を使用するので、試行した接続を終了するまでに 35 秒待ちます。

```
"Server DSN=SERVERDEMO;Host=localhost;Connection Timeout=15;Min Pool Size=50;Connection
Timeout=35"
```

サポートされる接続文字列オプションの一覧については、**接続文字列プロパティ**を参照してください。

Zen Performance Tuning Wizard の使用

Performance Wizard を使用して、Zen ADO.NET データ プロバイダーと Zen ADO.NET Entity Framework データ プロバイダーに最適な接続文字列オプションを選択することができます。

詳細については、Zen Performance Tuning Wizard の使用を参照してください。

ストアド プロシージャ

アプリケーションでストアド プロシージャを使用できるようにするには、次の手順に従ってください。

- PsqlCommand オブジェクトの CommandText プロパティにストアド プロシージャ名を設定します。

```
MyCommand.CommandText = "GetEmpSalary";
```

- PsqlCommand オブジェクトの CommandType プロパティに StoredProcedure を設定します。

```
MyCommand.CommandType = CommandType.StoredProcedure;
```

- 必要に応じてパラメーター引数を指定します。アプリケーションでは PsqlCommand オブジェクトのパラメーター コレクションに、ストアド プロシージャの引数の順序どおりにパラメーターを追加してください。ただし、アプリケーションでは PsqlCommand オブジェクトの CommandText プロパティでパラメーター マーカーを指定する必要はありません。

ストアド プロシージャから戻り値を取得するには、アプリケーションで PsqlCommand オブジェクトのパラメーター コレクションにパラメーターを余分に追加する必要があります。このパラメーターの ParameterDirection プロパティを ParameterDirection.ReturnValue に設定します。戻り値の取得用のパラメーターは、パラメーター コレクションのどこにあってもかまいません。これは、このパラメーターが、PsqlCommand オブジェクトの Text プロパティにある特定のパラメーター マーカーに対応していないためです。

ストアド プロシージャからの戻り値がない場合は、ParameterDirection プロパティに ReturnValue としてバインドされているパラメーターは無視されます。

ストアド プロシージャがデータベースから ReturnValue を返しても、アプリケーションでそのパラメーターをバインドしていない場合は、データ プロバイダーによってその値が破棄されます。

Zen ADO.NET Entity Framework ユーザーへの注記：PsqlConnection オブジェクトには、拡張された統計情報機能を提供するためのプロパティおよびメソッドが含まれています。これらのメソッドおよびプロパティは、Zen ADO.NET データ プロバイダーでは標準ですが、Zen ADO.NET Entity Framework レイヤーでは利用できません。代わりに、Zen ADO.NET Entity Framework データ プロバイダーは「擬似」ストアド プロシージャ

を介して同様の機能を公開します。詳細については、ADO.NET Entity Framework でのストアド プロシージャの使用を参照してください。

IP アドレスの使用

データ プロバイダーは、IPv4 形式および IPv6 形式のインターネット プロトコル（IP）アドレスをサポートします。お使いのネットワークが名前付きサーバーをサポートしている場合は、データ ソースに指定されたサーバー名を IPv4 アドレスまたは IPv6 アドレスに解決することができます。

EnableIPV6 接続文字列オプションを True に設定すると、IPv6 プロトコルがインストールされたクライアントを、IPv4 アドレスまたは IPv6 アドレスのいずれかを使用するサーバーに接続できるようになります。IPv6 形式の詳細については、『*Getting Started with Zen*』の IPv6 を参照してください。

トランザクションのサポート

Zen データ プロバイダーは、.NET Framework 内で完全に実装されるトランザクションのサポートにはマネージ コードのみを使用します。

ローカル トランザクションの使用

ローカル トランザクションは、基盤となるデータベースの内部的なトランザクション マネージャーを使用します。

アプリケーションでは、PsqlConnection オブジェクトで BeginTransaction を呼び出すことによって PsqlTransaction オブジェクトを作成します。トランザクションのコミットや中止などその後の操作は PsqlTransaction オブジェクトで実行されます。

スレッドのサポート

PsqlConnection オブジェクトはスレッドセーフです。それぞれ別のスレッドでアクセスする複数の PsqlCommand オブジェクトが、同時に 1 つの接続を使用できます。

別々のスレッドでほかのパブリック オブジェクトやデータ プロバイダー固有のオブジェクトに同時にアクセスするのはスレッド セーフではありません。

Unicode のサポート

データ プロバイダーは、.NET Framework SDK の規定に従って Unicode をサポートします。これは、データ プロバイダーでは Unicode UTF-16 エンコーディングで文字を表わすことを意味します。

データ プロバイダーは UTF-16 文字をデータベースで使用されている形式に変換し、.NET Framework 文字列をアプリケーションに返します。たとえば、Zen データベース コード ページが拡張 ASCII 形式である場合、データ プロバイダーは拡張 ASCII を使用してデータベースに送られた文字を表します。その後、データ プロバイダーはアプリケーションに送り戻される前に、返された拡張 ASCII 文字を変換します。

Unicode と国際的な文字の .NET Framework 実装に関する詳細については、.NET Framework SDK のドキュメントを参照してください。

分離レベル

Zen は ReadCommitted と Serializable 分離レベルをサポートします。レコード レベルのロックをサポートします。詳細については、**ロック レベルと分離レベル**を参照してください。

SQL エスケープ シーケンス

Zen データ プロバイダーでサポートする SQL エスケープ シーケンスの説明については、.NET の SQL エスケープ シーケンスを参照してください。

イベント処理

イベント ハンドラーは PsqlInfoMessageEventArgs 型の引数を受け取ります。これにはイベントに関するデータが含まれています。詳細については、PsqlInfoMessageEventArgs オブジェクトを参照してください。

このイベントは次のように定義されます。

```
public event PsqlInfoMessageEventHandler InfoMessage;
```

データベース サーバーから送られる警告や情報メッセージを処理したいクライアントは、このイベントを受け取るために PsqlInfoMessageEventHandler デリゲートを作成してください。

これらのイベントを使用して、パッケージ、ストアド プロシージャ、またはストアド関数（これらはすべてコマンドを作成します）の作成時に発生するエラーを取得することができます。パッケージ、ストアド プロシージャ、またはストアド関数で作成されたコマンドをコンパイルするときに Zen でエラーが検出されると、有効ではありませんがオブジェクトが作成されます。イベントが送られ、エラー発生したことを示します。

次のコードでは、PsqlConnection オブジェクトの InfoMessage イベントを処理するメソッドを表わすデリゲートを定義しています。

```
[Serializable]
public delegate void PsqlInfoMessageEventHandler(
  object sender
  PsqlInfoMessageEventArgs e
);
```

ここで、*sender* はイベントを生成したオブジェクト、*e* は警告を説明する PsqlInfoMessageEventArgs オブジェクトです。イベントとデリゲートの詳細については、Microsoft .NET Framework SDK のドキュメントを参照してください。

エラー処理

PsqlError オブジェクトは、Zen サーバーで生成されたエラーや警告に関する情報を収集します。詳細については、PsqlError オブジェクトを参照してください。

PsqlException オブジェクトは、Zen サーバーがエラーを返したときに作成されスローされます。データ プロバイダーによって生成された例外は標準のランタイム例外として返されます。詳細については、PsqlException オブジェクトを参照してください。

.NET オブジェクトの使用

データ プロバイダーでは .NET パブリック オブジェクトをサポートし、それらをシールド オブジェクト（封印されたオブジェクト）として公開します。

詳細については、サポートされる .NET オブジェクトを参照してください。

.NET 用アプリケーションの開発

データ コンシューマー アプリケーションの開発者は、Microsoft .NET の仕様やオブジェクト指向のプログラミング技術に精通している必要があります。

Microsoft では以下のような ADO.NET に関する豊富な情報をオンラインで提供しています。

- Microsoft .NET へのアップグレード：ADO プログラマのための ADO.NET
  http://msdn.microsoft.com/ja-jp/library/aa302323.aspx
- .NET Framework データ プロバイダーによるデータのアクセス
  http://msdn.microsoft.com/ja-jp/library/aa735598(VS.71).aspx
- ADO.NET 2.0 基本クラスおよびファクトリによる汎用的なコーディング
  http://msdn.microsoft.com/ja-jp/library/dd278213.aspx
- セキュリティ ポリシーの実施
  http://msdn.microsoft.com/ja-jp/library/vstudio/sa4se9bc(v=vs.100).aspx
- サービス コンポーネントの作成
  http://msdn.microsoft.com/ja-jp/library/3x7357ez.aspx
- DataSets、DataTables、DataViews
  https://msdn.microsoft.com/ja-jp/library/ss7fbaez(v=vs.110).aspx
- DataSet での XML の使用
  https://msdn.microsoft.com/ja-jp/library/84sxtbxh(v=vs.110).aspx

メモ： ここに記載されているリンクは、microsoft.com の新しいオンラインの場所にリダイレクトされる場合があります。

高度な機能

以下のトピックでは、データ プロバイダーの高度な機能について説明します。

- 接続プールの使用
- ステートメント キャッシングの使用
- 接続フェールオーバーの使用
- クライアント ロード バランスの使用
- 接続の再試行機能の使用
- セキュリティの設定
- Zen Bulk Load の使用
- 診断機能の使用

接続プールの使用

接続プールを使用すれば、作成済みの接続を再利用できるので、データ プロバイダーはデータベースに接続するたびに新しい接続を作成する必要がなくなります。データ プロバイダーでは .NET クライアント アプリケーションで自動的に接続プールを使用できるようになりました。

接続プールの動作は、接続文字列オプションによって制御することができます。たとえば、接続プール数、プール内の接続数、接続が破棄されるまでの時間（秒数）を定義することができます。

ADO.NET の接続プールは .NET Framework によって提供されません。ADO.NET データ プロバイダー自体に実装する必要があります。

接続プールの作成

各接続プールは、それぞれ特有の接続文字列と関連付けられます。デフォルトでは、接続プールは一意な接続文字列を使って最初にデータベースに接続したときに作成されます。プールには、プールの最小限のサイズまで接続が格納されます。プールが最大限のサイズに達するまで、接続がプールに追加されていきます。

プールは、その中で接続が開いている限り、あるいは、Connection オブジェクトへの参照を持つアプリケーションによって使用されており、そのオブジェクトに開いている接続がある限り、アクティブであり続けます。

新しい接続を開いたとき、その接続文字列が既存のプールと一致しない場合は、新しいプールを作成する必要があります。同じ接続文字列を使用することで、アプリケーションのパフォーマンスやスケーラビリティを向上させることができます。

以下の C# コード例では、3 つの新しい PsqlConnection オブジェクトが作成されます。ただし、これらのオブジェクトを管理するために必要な接続プールは 2 つのみです。1 番目と 2 番目の接続文字列では、ユーザー ID とパスワードに割り当てられる値と Min Pool Size オプションの値だけが異なることに注意してください。

```
DbProviderFactory Factory = DbProviderFactories.GetFactory("Pervasive.Data.SqlClient");
DbConnection conn1 = Factory.CreateConnection();
conn1.ConnectionString = "Server DSN=DEMODATA;User ID=test;
Password = test; Host = localhost;MinPoolSize=5 ";
conn1.Open();
// プール A が作成されます
DbConnection conn2 = Factory.CreateConnection();
conn2.ConnectionString = "Server DSN=DEMODATA2;User ID=lucy;
Password = quake; Host = localhost;MinPoolSize=10 ";
conn2.Open();
// 接続文字列が異なるため、プール B が作成されます
```

```
DbConnection conn3 = Factory.CreateConnection();
conn3.ConnectionString = "Server DSN=DEMODATA;User ID=test;
Password = test; Host = localhost;MinPoolSize=5 ";
conn3.Open();
// conn3 は conn1 と一緒にプール A に入ります
```

プールへの接続の追加

接続プールは、アプリケーションが使用するそれぞれの一意な接続文字列を作成するプロセス内で生成されます。プールが作成されると、Min Pool Size 接続文字列オプションによって設定される、プールの必要最小限のサイズ条件を満たすだけの接続がプールに格納されます。アプリケーションが Min Pool Size を超える接続を使用する場合、データ プロバイダーは Max Pool Size 接続文字列オプション（プール内の最大接続数を設定します）の値になるまで、プールに接続を追加割り当てします。

Connection.Open(...) メソッドを呼び出すアプリケーションで Connection オブジェクトが要求されたとき、プールから使用可能な接続が入手できる場合には、接続はプールから取得されます。使用可能な接続とは、現在ほかの有効な Connection オブジェクトによって使用されておらず、合致する分散トランザクション コンテキストを持ち（妥当な場合）、サーバーへの有効なリンクを持っている接続と定義付けられます。

プールの最大サイズに達し、かつ入手できる使用可能な接続がない場合には、要求はデータ プロバイダーのキューに入れられます。データ プロバイダーは使用可能な接続をアプリケーションへ返すために、Connection Timeout 接続文字列オプションの値が示す時間だけ待ちます。この時間が経過しても接続が入手可能にならなかった場合は、データ プロバイダーはアプリケーションにエラーを返します。

データ プロバイダーに対し、プールされている接続数に影響を与えないで、指定したプールの最大サイズを超える接続を作成できるようにすることができます。これはたとえば、時折発生する接続要求の急増を処理する場合などに有用です。Max Pool Size Behavior 接続文字列オプションを SoftCap に設定することにより、作成される接続数は Max Pool Size に設定された値を超えることが可能になりますが、プールされる接続数は設定値を超えません。プールの最大接続数が使用されている場合、データ プロバイダーは新しい接続を作成します。接続がプールに返されたとき、そのプールにアイドル状態の接続が入っている場合には、プール メカニズムは接続プールが Max Pool Size を決して超えないよう、破棄する接続を選択します。Max Pool Size Behavior を HardCap に設定した場合、作成される接続数は Max Pool Size に設定された値を超えません。

重要：PsqlConnection オブジェクトの Close() または Dispose() メソッドを使用して接続を閉じると、その接続はプールに追加されるか戻されます。アプリケーションで Close() メソッドを使用した場合、接続文字列の設定は Open() メソッドを呼び出す前の

状態にとどまります。Dispose メソッドを使用して接続を閉じた場合には、接続文字列の設定は消去され、デフォルトの設定に戻されます。

プールからの接続の削除

接続プールから接続が削除されるのは、Load Balance Timeout 接続文字列オプションで決められた存続時間が過ぎた場合や、接続文字列の一致する新しい接続がアプリケーションによって開始された（PsqlConnection.Open() が呼び出された）場合です。

接続プールの接続をアプリケーションに返す前に、プール マネージャーはその接続がサーバー側で閉じられているかどうかを確認します。接続が有効でなくなっていれば、プール マネージャーはその接続を破棄し、プールから別の入手可能で有効な接続を返します。

再使用するための接続プールから接続をどのような順序で削除するかを、Connection Pool Behavior 接続文字列オプションを用いて、接続の使用頻度または使用時期を基に制御することができます。接続をバランスよく使用するには、LeastFrequentlyUsed 値または LeastRecentlyUsed 値を使用します。あるいは、毎回同じ接続を使用した方がパフォーマンスが良くなるアプリケーションの場合は、MostFrequentlyUsed 値または MostRecentlyUsed 値を使用できます。

Connection オブジェクトの ClearPool メソッドおよび ClearAllPools メソッドは、接続プールからすべての接続を削除します。*ClearPool* は特定の接続に関連付けられている接続プールを空にします。対照的に、*ClearAllPools* はデータ プロバイダーによって使用されるすべての接続プールを空にします。メソッドを呼び出すときに使用中だった接続は、閉じるときに破棄されます。

メモ：デフォルトで、無効な接続を破棄することによって、接続数が Min Pool Size 属性で指定した数より少なくなった場合、新しい接続はアプリケーションで必要になるまで作成されません。

プール内の停止接続の処理

アイドル状態の接続が、そのデータベースへの物理的接続を失った場合には何が起こるのでしょうか。たとえば、データベース サーバーが再起動されたり、ネットワークが一時的に中断されるとします。プール内の既存の Connection オブジェクトを使って接続しようとするアプリケーションは、データベースへの物理的接続が失われているため、エラーを受け取る可能性があります。

Control Center はこの状況をユーザーに意識させないで処理します。Connection.Open() では、データ プロバイダーは単に接続プールから接続を返すだけなので、アプリケーションはこのとき何のエラーも受け取りません。SQL ステートメントを実行するために Connection オブジェクトが初めて使用されたとき（たとえば、Command オブジェクトの Execute メソッドを介して）、データ プロバイダーはサーバーへの物理的接続が失われていることを検出し、SQL ステートメントを実行する**前**にサーバーへの再接続を試みます。データ プロバイダーがサーバーへ再接続できた場合、アプリケーションには SQL の実行結果が返され、エラーは何も返されません。データ プロバイダーはこのシームレスな再接続を試みる際、接続フェールオーバー オプションが有効になっていればそれを利用します。プライマリ サーバーが使用できないときはバックアップ サーバーへ接続するようデータ プロバイダーを構成する方法については、**接続フェールオーバーの使用**を参照してください。

メモ：データ プロバイダーは、SQL ステートメントの実行時にデータベース サーバーへの再接続を試みることができるため、ステートメントが実行されるときに接続エラーをアプリケーションへ返すことができます。データ プロバイダーがサーバーに再接続できない（たとえば、サーバーがまだダウンしている）場合、実行メソッドは再接続の試行が失敗したことと、その接続が失敗した理由の詳細を知らせるエラーをスローします。

この接続プール内の停止接続を処理する技術には、接続プール メカニズムを超越して最高のパフォーマンスを得られるという効果があります。一部のデータ プロバイダーは、接続がアイドル状態である間、ダミーの SQL ステートメントを使って定期的にサーバーに ping を実行します。その他のデータ プロバイダーは、アプリケーションから接続プール内の接続の使用を要求されたときにサーバーに ping を実行します。これらの手法はいずれもデータベース サーバーへの往復を追加するため、結局のところ、アプリケーションの通常の操作が発生している間ずっと、アプリケーションの速度が落ちます。

接続プールのパフォーマンスの追跡

データ プロバイダーは、このデータ プロバイダーを使用するアプリケーションの調整およびデバッグが行える PerfMon カウンターのセットをインストールします。PerfMon カウンターの詳細については、PerfMon のサポートを参照してください。

ステートメント キャッシングの使用

ステートメント キャッシュは、プリペアド ステートメントのグループまたは Command オブジェクトのインスタンスで、アプリケーションによって再使用が可能です。ステートメント キャッシュを使用するとアプリケーションのパフォーマンスを向上させることができます。これは、プリペアド ステートメントの動作が、そのステートメントがアプリケーションの存続期間中に何度再使用されたとしても、1 度だけ実行されるためです。キャッシュ内のステートメントの有効性を分析することができます（接続統計情報によるパフォーマンスの分析を参照してください）。

ステートメント キャッシュは物理接続に属します。実行された後、プリペアド ステートメントはステートメント キャッシュに置かれ、接続が閉じられるまで保持されます。

ステートメント キャッシングは複数のデータ ソースにわたって使用でき、抽象化技術の下で使用できます。

ステートメント キャッシングの有効化

デフォルトで、ステートメント キャッシングは無効になっています。既存のアプリケーションのステートメント キャッシングを有効にするには、Statement Cache Mode 接続文字列オプションを Auto に設定します。この場合は、すべてのステートメントをステートメント キャッシュに置くことができます。

キャッシュすると明示的にマークしたステートメントのみをステートメント キャッシュに置くように、ステートメント キャッシングを設定することもできます。これを行うには、そのステートメントの Command オブジェクトの StatementCacheBehavior プロパティに Cache を設定し、Statement Cache Mode 接続文字列オプションに ExplicitOnly を設定します。

次の表はステートメント キャッシング設定とその影響を要約したものです。

動作	StatementCacheBehavior	Statement Cache Mode
ステートメントを明示的にステートメント キャッシュに追加します。	Cache	ExplicitOnly（デフォルト）

動作	StatementCacheBehavior	Statement Cache Mode
ステートメントをステートメント キャッシュに追加します。必要に応じ、ステートメントは、Cache とマークされたステートメント用の場所を空けるために削除されます。	Implicit（デフォルト）	Auto
ステートメントをステートメント キャッシュから明確に除外します。	DoNotCache	Auto または ExplicitOnly

ステートメント キャッシング手法の選択

ステートメント キャッシングを使用すると、アプリケーションの存続期間中にプリペアド ステートメントを複数回再使用するアプリケーションはパフォーマンスが向上します。ステートメント キャッシュのサイズは Max Statement Cache Size 接続文字列オプションに設定します。ステートメント キャッシュのスペースが限られている場合は、1 度しか使用されないプリペアド ステートメントをキャッシュしないでください。

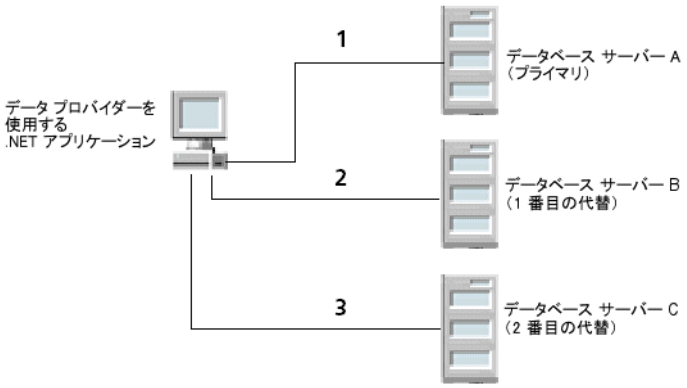
アプリケーションが使用する全プリペアド ステートメントをキャッシュすれば、最高のパフォーマンスを提供できるように思われます。しかし、この手法では、接続プールを使ってステートメント キャッシングを実装した場合、データベースのメモリに負担をかける結果になります。この場合、プールされた接続はそれぞれステートメント キャッシュを持ち、アプリケーションが使用する全プリペアド ステートメントを含むことになります。これらのプールされたプリペアド ステートメントは、すべてデータベースのメモリにも保持されます。

接続フェールオーバーの使用

接続フェールオーバーによって、ハードウェア障害やトラフィックの過負荷などが原因でプライマリ データベース サーバーが利用できなくなった場合でも、アプリケーションは代替またはバックアップ データベース サーバーに接続することができます。接続フェールオーバーは、重要な .NET アプリケーションが依存するデータを常に使用可能な状態にします。

プライマリ サーバーが接続を受け入れない場合に接続を試行する代替データベースのリストを設定することで、データ プロバイダーにおける接続フェールオーバー機能をカスタマイズすることができます。接続が成功するまで、あるいはすべての代替データベースへの接続を指定した回数試行するまで、接続の試行が続けられます。

たとえば、次の図は複数のデータベース サーバーを持つ環境を示しています。データベース サーバー A はプライマリ データベース サーバー、データベース サーバー B は 1 番目の代替サーバー、そしてデータベース サーバー C は 2 番目の代替サーバーとして設計されています。



まず、アプリケーションはプライマリ データベースであるデータベース サーバー A（**1**）に接続を試みます。接続フェールオーバー機能を有効にすると、アプリケーションがデータベース サーバー A に接続できなかった場合はデータベース サーバー B（**2**）に接続を試みます。その接続の試行も失敗した場合、アプリケーションはデータベース サーバー C（**3**）に接続を試みます。

このシナリオで、最低でもどれか 1 つのサーバーへは接続できると思われますが、接続にすべて失敗した場合、データ プロバイダーはプライマリ サーバーと各代替データベースに対し指定の回数分だけ接続を再試行させることができます。**接続の再試行**（Connection Retry Count）機能を使用すれば、試行回数を指定することができます。

接続の試行間隔（Connection Retry Delay）機能を使用すれば、接続の試行間隔を秒数で指定することもできます。接続の再試行の詳細については、**接続の再試行機能の使用**を参照してください。

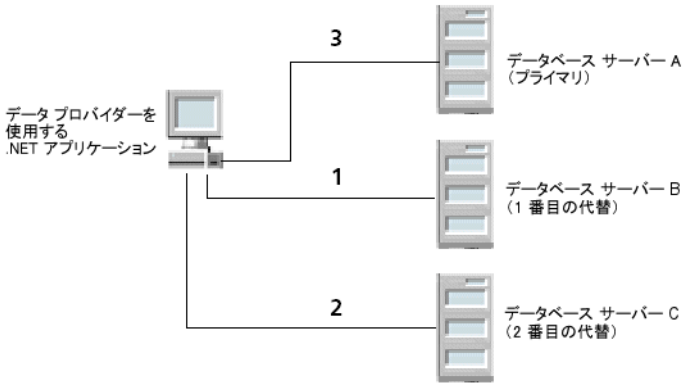
データ プロバイダーは、現在対象としている代替サーバーとの通信が確立できなかった場合にのみ、次の代替サーバーへフェールオーバーします。データ プロバイダーがデータベースとの通信に成功したが、ログイン情報が正しくないなどの原因でそのデータベースが接続要求を拒否した場合、データ プロバイダーは例外を生成し、代替サーバー リスト内の次のデータベースへは接続を試行しません。これは、各代替サーバーがプライマリ サーバーのミラーであり、認証パラメーターや関連情報がすべて同じであることが前提です。

接続フェールオーバーでは新しい接続のみを保護し、トランザクションやクエリの状態は保持しません。ご使用のデータ プロバイダーにおける接続フェールオーバーの設定の詳細については、**接続フェールオーバーの設定**を参照してください。

クライアント ロード バランスの使用

クライアント ロード バランスは接続フェールオーバーと共に動作してユーザーの環境で新しい接続を分散することで、接続要求に対応できないサーバーがないようにします。接続フェールオーバーとクライアント ロード バランスがどちらも有効な場合、プライマリ データベースおよび代替データベースへ接続が試行される順序はランダムです。

たとえば、クライアント ロード バランスが有効になっているとします。



まず、データベース サーバー B への接続が試行されます（**1**）。それからデータベース サーバー C への接続が試行され（**2**）、続いてデータベース サーバー A への接続が試行されます（**3**）。その後、これと同じ順序で接続が試行されます。対照的に、このシナリオでクライアント ロード バランスが無効だった場合、各データベースへの接続の試行はシーケンシャルな順序で行われます。つまり、最初にプライマリ サーバーに試行したら、その後は代替サーバー リスト内のエントリ順に基づいて代替サーバーに試行します。

ご使用のデータ プロバイダーにおけるクライアントとロード バランスの設定の詳細については、接続フェールオーバーの設定を参照してください。

接続の再試行機能の使用

接続の再試行では、データ プロバイダーが最初の接続の試行に失敗した後に、プライマリ サーバーおよび（設定している場合）代替サーバーへ接続を試行する回数を定義します。接続の再試行は、システム復旧の有力な方策となります。たとえば、停電などの電力障害によりクライアントとサーバーの両方が動作しなくなってしまうシナリオを考えてみましょう。電力が復旧してすべてのコンピューターが再起動される際、サーバーが自身のスタートアップ ルーチンを完了する前にクライアントがそのサーバーに接続しようとするかもしれません。接続の再試行が有効な場合、クライアント アプリケーションは、サーバー側で接続が正しく受け入れられるようになるまで接続を何度も試行し続けることができます。

接続の再試行は、サーバーが 1 つしかない環境でも使用できますが、複数サーバーを持つ環境における接続フェールオーバー シナリオの補完機能としても使用できます。

接続文字列オプションを使用すると、データ プロバイダーが接続を試行する回数や接続の試行間隔を秒単位で指定できます。接続の再試行に対する設定の詳細については、接続フェールオーバーの設定を参照してください。

接続フェールオーバーの設定

接続フェールオーバーによって、ハードウェア障害やトラフィックの過負荷などが原因でプライマリ データベース サーバーが利用できなくなった場合でも、アプリケーションは代替またはバックアップ データベース サーバーに接続することができます。

接続フェールオーバーに関する詳細については、**接続フェールオーバーの使用**を参照してください。

接続フェールオーバーを別のサーバーに構成するには、プライマリ サーバーが接続を受け入れない場合に接続を試行する代替データベース サーバーのリストを指定する必要があります。これを行うには Alternate Servers 接続文字列オプションを使用します。接続の確立に成功するまで、あるいはリストにエントリされるすべての代替データベースへの接続を 1 回ずつ試行（デフォルト）するまで、接続の試行が続けられます。

任意で、以下の接続フェールオーバー機能も指定することができます。

- データ プロバイダーが最初に接続を試行した後に、プライマリ サーバーおよび代替サーバーへ接続を試行する回数。デフォルトでは、データ プロバイダーは接続を再試行しません。この機能を設定するには Connection Retry Count 接続文字列オプションを使用します。
- プライマリ サーバーと代替サーバーへの接続の試行間隔（秒数）。デフォルトの間隔は 3 秒です。この機能を設定するには Connection Retry Delay 接続文字列オプションを使用します。
- データ プロバイダーがプライマリ サーバーおよび代替サーバーへの接続を試行する際、ロード バランスを使用するかどうか。ロード バランスが有効な場合、データ プロバイダーは接続の試行順序のパターンをシーケンシャルではなくランダムにします。デフォルトでは、ロード バランスは使用されません。この機能を設定するには Load Balancing 接続文字列オプションを使用します。

接続文字列を使用して、データ プロバイダーに対し接続フェールオーバーを使用するよう指示します。**接続文字列の使用**を参照してください。

次の C# コードには、データ プロバイダーで接続フェールオーバー機能と、その接続オプションであるロード バランス、接続の再試行、および接続の再試行の間隔をすべて使用するように設定する接続文字列が含まれています。

```
Conn = new PsqlConnection Conn = new PsqlConnection();
Conn = new PsqlConnection("Host=myServer;User ID=test;Password=secret;
Server DSN=SERVERDEMO;Alternate Servers="Host=AcctServer, Host=AcctServer2";
Connection Retry Count=4;Connection Retry Delay=5;Load Balancing=true;
Connection Timeout=60")
```

具体的に説明すると、この接続文字列の設定は、データ プロバイダーで 2 つの代替サーバーを接続フェールオーバー サーバーとして使用すること、最初の接続の試行に失敗した場合はさらに 4 回接続を再試行すること、試行間隔は 5 秒間とし、プライマリ サーバーと代替サーバーへの接続の試行順序はランダムとすることを示しています。接続の試行は 1 回当り 60 秒間持続し、試行する順序は最初に行ったランダム順序がそのまま最後まで使用されます。

セキュリティの設定

データ プロバイダーは、接続での暗号化されたネットワーク通信（ワイヤ暗号化とも呼ばれます）をサポートします。デフォルトでは、データ プロバイダーはサーバーの設定を反映します。詳細については、**接続文字列の使用**を参照してください。

データ プロバイダーで許可される暗号化のレベルは、使用される暗号化モジュールによって異なります。デフォルトの暗号化モジュールでは、データ プロバイダーは 40 ビット、56 ビット、および 128 ビット暗号化をサポートしています。

データ暗号化は、データの暗号化と復号で必要となる追加オーバーヘッド（主に CPU 使用）のため、パフォーマンスに悪影響を与えることがあります。詳細については、**パフォーマンスに関する考慮点**を参照してください。

暗号化に加え、Control Center は .NET Framework で定義されるセキュリティ権限を通してセキュリティを実装します。

コードへのアクセス権限

データ プロバイダーをロードして実行するためには、FullTrust 権限を設定する必要があります。これは、System.Data のクラスが FullTrust 権限を継承しなければならないためです。これらのクラスは、すべての ADO.NET データ プロバイダーで DataAdapter を実装するために必要です。

セキュリティの属性

データ プロバイダーには AllowPartiallyTrustedCallers 属性がマークされています。

Zen Bulk Load の使用

Zen Bulk Load は、すべてのバルク ロードの要求に対して 1 か所ですべてを行えるワンストップ手法を提供します。これは、Zen およびこのバルク ロード機能をサポートするすべての DataDirect Connect 製品にバルク ロード操作を行う上で単純かつ一貫性があります。つまり、標準ベースの API バルク インターフェイスを使用してバルク ロード アプリケーションを記述し、それから、その仕事を実行するためにすぐにデータベースのデータ プロバイダーまたはドライバーをプラグインすることができます。

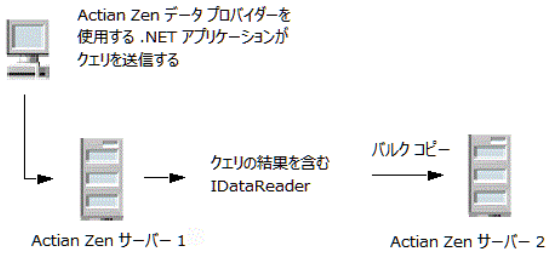
データを Zen、Oracle、DB2、および Sybase にロードする必要があるとします。これまでは、バルク ロード操作のためにデータベース ベンダー製のツールを使用するか、独自のツールを作成する必要がありました。今では、Zen Bulk Load に組み込まれた相互運用性により、作業は非常に簡単になりました。もう 1 つの利点は、Zen Bulk Load は 100% マネージ コードを使用するため、ほかのベンダー製の基盤となるユーティリティやライブラリを必要としないことです。

異なるデータ ストア間でのバルク ロード操作は、クエリの結果をカンマ区切り値（CSV）形式ファイルであるバルク ロード データ ファイルに残すことによって完成されます。このファイルは、Control Center と、バルク ロードをサポートする DataDirect Connect for Zen ADO.NET データ プロバイダー間で使用できます。さらに、バルク ロード データ ファイルは、DataDirect Connect 製品やバルク ロード機能をサポートするデータ プロバイダーで使用できます。たとえば、Zen データ プロバイダーで生成された CSV ファイルは、バルク ロード対応の DataDirect Connect for ODBC ドライバーで使用できます。

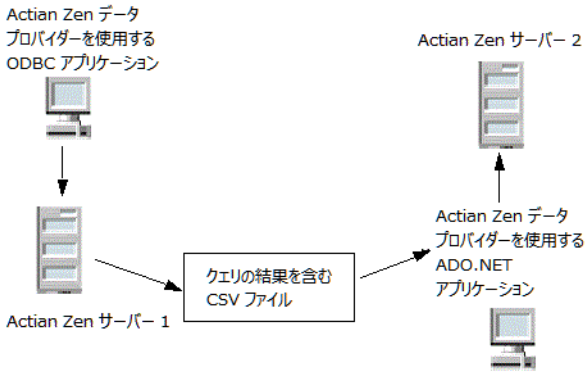
Zen Bulk Load で使用するシナリオ

Control Center で Zen Bulk Load を使用する方法は 2 つあります。

- 新しい Zen バージョンにアップグレードし、次の図に示すように、古い Zen データ ソースから新しいデータ ソースへデータをバルク コピーします。



- データベースからデータをエクスポートし、その結果を Zen データベースに移行します。次の図は、Zen ADO.NET データベース サーバーにデータをコピーする ODBC 環境を示しています。



この図で、ODBC アプリケーションは CSV ファイルへデータをエクスポートするコードを含み、ADO.NET アプリケーションは CSV ファイルを指定して開くコードを含みます。Control Center および DataDirect ODBC ドライバーは一貫した形式を使用するため、これらの標準インターフェイスを介して相互運用性がサポートされます。

Zen Common Assembly

ADO.NET 用の Zen BulkLoad の実装は、事実上の業界標準である Microsoft SqlBulkCopy クラスの定義を使用し、強力な組み込み機能を追加してバルク操作の信頼性をより高めるため、柔軟性に加えて相互運用性も高めます。

データ プロバイダーには、Zen Bulk Load をサポートするためのプロバイダー固有のクラスが含まれます。詳細については、**データ プロバイダー固有のクラス**を参照してください。共通プログラミング モデルを使用する場合は、Zen Common Assembly のクラスを使用できます（Zen Common Assembly を参照してください）。

Pervasive.Data.Common アセンブリには、バルク データ形式間の機能を提供する CsvDataReader および CsvDataWriter などの Zen Bulk Load に対応したクラスが含まれています。

共通アセンブリは、共通プログラミング モデルを使用するバルク ロード クラスのサポート機能も拡張します。つまり、SqlBulkCopy パターンが新しい DbBulkCopy 階層で使用できるようになりました。

データ プロバイダーの将来のバージョンには、共通プログラミング モデル技術を拡張するほかの機能が含められます。Pervasive.Data.Common アセンブリが対応しているクラスの詳細については、Zen Common Assembly を参照してください。

バルク ロード データ ファイル

異なるデータ ストア間でのクエリの結果は、カンマ区切り値（CSV）形式ファイルであるバルク ロード データ ファイルに保存されます。BulkFile プロパティで定義されるファイル名は、バルク データの読み書きに使用されます。ファイル名に拡張子が含まれない場合は、".CSV" と見なされます。

例

Zen ソース テーブルの GBMAXTABLE には 4 つの列が含まれています。次の C# コード例は、CsvDataWriter によって作成される GBMAXTABLE.csv および GBMAXTABLE.xml への書き出しを行います。この例では、DbDataReader クラスを使用していることに注目してください。

```
cmd.CommandText = "SELECT * FROM GBMAXTABLE ORDER BY INTEGERCOL";
DbDataReader reader = cmd.ExecuteReader();
CsvDataWriter csvWriter = new CsvDataWriter();
csvWriter.WriteToFile("¥¥NC1¥net¥Zen¥GBMAXTABLE¥GBMAXTABLE.csv", reader);
```

バルク ロード データ ファイルの GBMAXTABLE.csv には、次のようなクエリの結果が含まれます。

```
1,0x6263,"bc","bc"
2,0x636465,"cde","cde"
3,0x64656667,"defg","defg"
4,0x6566676869,"efghi","efghi"
5,0x666768696a6b,"fghijk","fghijk"
6,0x6768696a6b6c6d,"ghijklm","ghijklm"
7,0x68696a6b6c6d6e6f,"hijklmno","hijklmno"
8,0x696a6b6c6d6e6f7071,"ijklmnopq","ijklmnopq"
9,0x6a6b6c6d6e6f70717273,"jklmnopqrs","jklmnopqrs"
10,0x6b,"k","k"
```

GBMAXTABLE.xml ファイルは、このバルク ロード データ ファイルの形式を指定するバルク ロードの設定ファイルです。これについて次のセクションで説明します。

バルク ロード構成ファイル

バルク ロード構成ファイルは、CsvDataWriter.WriteToFile メソッドが呼び出されたときに作成されます（詳細については CsvDataWriter を参照してください）。

バルク ロード構成ファイルは、バルク ロード データ ファイル内の列の名前とデータ型を定義します。これらの名前とデータ型は、データのエクスポート元のテーブルや結果セットと同様に定義します。

バルク データ ファイルが作成できなかったり XML 構成ファイルに記述されているスキーマに準拠していない場合は、例外がスローされます。XML スキーマ定義の使用法については、**バルク データ構成ファイル用の XML スキーマ定義**を参照してください。

構成ファイルを持たないバルク ロード データ ファイルが読み込まれた場合は、以下のデフォルトが想定されます。

- すべてのデータは文字データとして読み取られます。カンマの間にある値は、それぞれ文字データとして読み取られます。
- デフォルトの文字セットは、バルク ロード CSV ファイルが現在読み取られているプラットフォームの文字セットです。詳細については、**文字セットの変換**を参照してください。

バルク ロード設定ファイルは、バルク データ ファイルについて記述されており、基となる XML スキーマによってサポートされます。

例

前のセクションで示したバルク ロード データ ファイルは、バルク ロード構成ファイルの GBMAXTABLE.xml で定義されています。このファイルには、テーブル内の 4 つの列のそれぞれについてデータ型とその他の情報が記述されています。

```
<?xml version="1.0" encoding="utf-8"?>
<table codepage="UTF-16LE" xsi:noNamespaceSchemaLocation="https://www.datadirect.com/ns/bulk/
BulkData.xsd" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance">
  <row>
   <column datatype="DECIMAL" precision="38" scale="0" nullable=
      "false">INTEGERCOL</column>
   <column datatype="VARBINARY" length="10" nullable=
      "true">VARBINCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252" nullable="true">VCHARCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252" nullable="true">UNIVCHARCOL</column>
  </row>
</table>
```

バルク ロード プロトコルの決定

バルク操作は、専用のバルク プロトコルを使用して実行できます。つまり、データ プロバイダーは基となるデータベースのプロトコルを使用します。場合によっては、専用のバルク プロトコルが使用できない場合があります。たとえば、ロードしようとする

データが、専用のバルク プロトコルが対応していないデータ型の場合です。その場合、データ プロバイダーは配列バインドなどの非バルク手法を自動的に使用してバルク操作を実行し、最適なアプリケーション稼働時間を継続します。

文字セットの変換

時には、異なる文字セットを使用するデータベース間でデータをバルク ロードする必要があります。

Control Center では、デフォルトのソース文字データ、つまり CsvDataReader からの出力および CsvDataWriter への入力は Unicode（UTF-16）形式です。ソース文字データは、常に CSV ファイルのコード ページに変換されます。しきい値を超えてデータが外部のオーバーフロー ファイルに書き出された場合、ソース文字データはバルク構成 XML スキーマで定義されている externalfilecodepage 属性で指定されたコード ページに変換されます（バルク データ構成ファイル用の XML スキーマ定義を参照してください）。構成ファイルで externalfilecodepage の値が定義されていない場合は CSV ファイルのコード ページが使用されます。

不要な変換処理を行わないためには、CSV および外部ファイルの文字データを Unicode（UTF-16）にするのが最良の方法です。以下のいずれかの状況では、アプリケーションにデータを別のコード ページで保存させたい場合があります。

- データが ADO.NET で書き出され ODBC で読み取られる。この場合、読み取り（および関連する文字変換）は ODBC によって行われます。文字データが既に正しいコード ページである場合、文字変換は不要です。
- 空白が考慮の対象となる。文字データがよりコンパクトに表されるかどうかは、コード ページによって異なります。たとえば、ASCII データは 1 文字が 1 バイトで、UTF-16 は 1 文字が 2 バイトです。

構成ファイルでは、任意で文字型の列ごとに 2 番目のコード ページを定義することができます。文字データが CharacterThreshold プロパティで定義した値を超え、別個のファイルに格納される場合（外部オーバーフロー ファイルを参照してください）、この値はそのファイルのコード ページを定義します。

この値を省略したりソース列で定義されたコード ページが不明の場合は、CSV ファイルに定義されたコード ページが使用されます。

外部オーバーフロー ファイル

CsvDataWriter オブジェクトの BinaryThreshold または CharacterThreshold プロパティの値がしきい値を超えた場合、別のファイルが生成されてバイナリまたは文字データを格納します。これらのオーバーフロー ファイルは、バルク データ ファイルと同じディレクトリにあります。

オーバーフロー ファイルに文字データが含まれる場合、ファイルの文字セットは CSV バルク構成ファイルで指定された文字セットが決定します。

ファイル名には CSV ファイル名と拡張子 ".lob" が含まれます（たとえば、CSV_*filename*_*nnnnnn*.lob）。これらのファイルは CSV ファイルと同じ場所に存在します。_000001.lob から開始して 1 ずつ大きくなります。

バルク コピー操作とトランザクション

デフォルトで、バルク コピー操作はトランザクションの一部ではなく分離した操作として実行されます。つまり、エラーが発生しても操作をロール バックするチャンスはないということです。

Zen では、既存のトランザクション内でバルク コピー操作を行うことができます。複数の手順で存在するトランザクションの一部としてバルク コピー操作を定義することができます。この手法を使うと、複数のバルク コピー操作を同じトランザクション内で実行し、トランザクション全体をコミットまたはロール バックすることができます。

エラー発生時にバルク コピー操作のすべてまたは一部をロール バックする方法については、Microsoft のオンライン ヘルプで " トランザクションとバルク コピー操作（ADO.NET）" を参照してください。

診断機能の使用

.NET Framework では、プログラムを再コンパイルすることなくエンド ユーザーが問題を特定するのに役立つ Trace クラスを提供します。

Control Center はさらに以下の診断機能を実現します。

- メソッド呼び出しのトレース機能
- アプリケーションの接続情報の監視を可能にする、パフォーマンス モニターのフック

メソッド呼び出しのトレース

トレース機能は、環境変数または PsqlTrace クラスを使って有効にできます。データ プロバイダーは、パブリック メソッドの全呼び出しの入力引数と、そのメソッドの出力および戻り値（ユーザーが呼び出したものすべて）をトレースします。各呼び出しには、メソッドの開始（ENTER）と終了（EXIT）のエントリがあります。

デバッグ中、機密事項のデータがプライベート変数または内部変数として格納され、アクセスが同じアセンブリに制限されたとしても、そのデータを読み取ることができます。セキュリティを維持するために、トレース ログはパスワードを 5 つのアスタリスク（*****）で示します。

環境変数の使用

環境変数を使用してトレースを有効にするということは、アプリケーションを変更しなくてもよいということです。ただし、環境変数の値を変更したら、その新しい値を有効にするにはアプリケーションの再起動が必要になります。

次の表では、トレースを有効にしたり制御したりするための環境変数について説明します。

環境変数	説明
PVSW_NET_Enable_Trace	1 以上に設定すると、トレースが有効になります。0（デフォルト）に設定すると、トレースは無効になります。
PVSW_NET_Recreate_Trace	1 に設定すると、アプリケーションを開始するたびにトレース ファイルが再作成されます。0（デフォルト値）に設定すると、トレース ファイルに追加されます。

環境変数	説明
PVSW_NET_Trace_File	トレース ファイルのパスと名前を指定します。

注記

- PVSW_NET_Enable_Trace = 1 に設定すると、トレース処理が開始されます。そのため、トレースを有効にする前に、トレース ファイルのプロパティ値を定義しておく必要があります。いったんトレース処理が開始されたら、ほかの環境変数の値を変更することはできません。
- 接続文字列オプションでも環境変数でもトレース ファイルが指定されていないのにトレースを有効にすると、データ プロバイダーは PVSW_NETTrace.txt という名前のファイルに結果を保存します。

静的メソッドの使用

ユーザーによっては、トレースを有効にするのに、データ プロバイダーの Trace クラスの静的メソッドを使用する方が便利だと思われるかもしれません。以下の C# コードでは、.NET Trace オブジェクトの静的メソッドを使用して、トレース ファイルの名前を MyTrace.txt とする PsqlTrace クラスを作成します。この値は環境変数で設定した値よりも優先されます。これ以降、データ プロバイダーへの呼び出しはすべて MyTrace.txt へトレースされます。

```
PsqlTrace.TraceFile="C:¥¥MyTrace.txt";
PsqlTrace.RecreateTrace = 1;
PsqlTrace.EnableTrace = 1;
```

トレースは次のような形式で出力されます。

```
<Correlation#> <Timestamp> <CurrentThreadName>
   <Object Address> <ObjectName.MethodName> ENTER ( または EXIT)
      Argument #1 : <Argument#1 Value>
      Argument #2 : <Argument#2 Value>
      ...
      RETURN:  <Method ReturnValue>  // EXIT の場合のみ
```

各項目の説明は次のとおりです。

Correlation# は重複のない番号で、これによって、アプリケーション内の同じメソッド呼び出しの ENTER エントリと EXIT エントリを符合させます。

Value は、個々の関数呼び出しに固有のオブジェクトのハッシュ コードです。

デバッグ中、機密事項のデータがプライベート変数または内部変数として格納され、アクセスが同じアセンブリに制限されたとしても、そのデータを読み取ることができま

す。セキュリティを維持するために、トレース ログはパスワードを 5 つのアスタリスク（*****）で示します。

PerfMon のサポート

パフォーマンス モニター（PerfMon）や VS パフォーマンス モニター（VSPerfMon）ユーティリティを使用すると、アプリケーションのパラメーターを記録し、その結果をレポートやグラフにして見ることができます。また、パフォーマンス モニターではアプリケーションの CLR（共通言語ランタイム）例外の回数や頻度を確認することもできます。加えて、使用中の接続数や接続プール数を分析して、ネットワークの負荷を調整することができます。

データ プロバイダーは、このデータ プロバイダーを使用するアプリケーションの調整およびデバッグが行える PerfMon カウンターのセットをインストールします。カウンターは、パフォーマンス モニターの［Zen ADO.NET Data Provider］というカテゴリ名の下に置かれます。

次の表では、アプリケーションの接続を調整するために使用できる各種 PerfMon カウンターについて説明します。

カウンター	説明
Current # of Connection Pools	プロセスに関連付けられている現在のプール数。
Current # of Pooled and Non-Pooled Connections	プールされている接続とプールされていない接続の現在の数。
Current # of Pooled Connections	プロセスに関連付けられているすべてのプールにある現在の接続数。
Peak # of Pooled Connections	プロセスの開始後、すべての接続プールにおいてカウントされた最大の接続数。
Total # of Failed Commands	プロセスの開始後、何らかの理由でエラーになったコマンドの実行総数。
Total # of Failed Connects	プロセスの開始後、何らかの理由でエラーになった接続を開くために試行した合計回数。

PerfMon とパフォーマンス カウンターの使用法の詳細については、Microsoft ドキュメント ライブラリを参照してください。

接続統計情報によるパフォーマンスの分析

.NET Framework 2.0 以上では実行時の統計をサポートしており、統計情報は接続単位で収集されます。Control Center は、幅広い種類の実行時の統計情報項目をサポートしています。これらの統計情報項目が提供する情報は、以下に役立ちます。

- アプリケーション パフォーマンスの自動分析
- アプリケーション パフォーマンスの傾向の特定
- 接続での問題の検出および通知の送信
- データ接続での問題修正の優先順位決定

統計情報項目の計測がパフォーマンスに与える影響はわずかです。最良の結果を得るためには、ネットワーク分析またはアプリケーションのパフォーマンス作用分析時にのみ統計情報の収集を有効にすることを考慮してください。

統計情報収集は、すべての Connection オブジェクトで、それらが使用可能である限り有効にできます。たとえば、アプリケーションで、ビジネス分析の実行に関連する複雑なトランザクションのセットを開始する前に統計情報を有効にし、タスクの完了時に統計情報を無効にするように定義できます。データ プロバイダーがサーバーに対して待機させられた時間および返された行数をタスクの完了直後に取得できますし、後から取得することもできます。アプリケーションはタスクの最後で統計情報を無効にするため、統計情報項目は関心のある期間だけ計測されます。

統計情報項目は、機能上 4 つのカテゴリに分類されます。

- ネットワーク レイヤ項目は、送受信されるバイト数およびパケット数、データ プロバイダーがサーバーの応答を待った時間などのネットワーク動作に関連する値を取得します。
- 集計項目は、サーバーとのやりとりごとに送受信されたバイト数などの計算値を返します。
- 行処理の統計情報項目は、アプリケーションが読み取らなかった行の処理に必要な時間とリソースに関する情報を提供します。
- ステートメント キャッシュの統計情報項目は、ステートメント キャッシュ内でのステートメントの動作を説明する値を返します（ステートメント キャッシュの使用法については、ステートメント キャッシングの使用を参照してください）。

統計情報項目の有効化と取得

Connection オブジェクトを作成すると、StatisticsEnabled プロパティを使用して統計情報の収集を有効にすることができます。データ プロバイダーは、接続が開かれると統計情報項目のカウントを開始し、ResetStatistics が呼び出されるまで継続します。接続を閉じた後 ResetStatistics を呼び出さずに再度開くと、統計情報項目のカウントは接続が閉じられた時点から継続されます。

RetrieveStatistics メソッドを呼び出すと、1 つまたは複数の統計情報項目のカウントを取得します。返される値は、RetrieveStatistics メソッドが呼び出された時点での " 適切な時期のスナップショット " となります。

統計情報収集および取得の範囲を定義できます。次の C# コード例では、統計情報項目はタスク A 作業のみを測定し、タスク B 作業の処理後に回収されます。

```
connection.StatisticsEnabled = true;
   // タスク A 作業を行います
connection.StatisticsEnabled = false;
   // タスク B 作業を行います
IDictionary currentStatistics = connection.RetrieveStatistics();
```

すべての統計情報項目を表示するには、次の C# コード例のようなコードを使用することができます。

```
foreach (DictionaryEntry entry in currentStatistics) {
   Console.WriteLine(entry.Key.ToString() + ": " + entry.Value.ToString());
}
Console.WriteLine();
```

SocketReads および SocketWrites 統計情報項目のみを表示するには、次の C# コード例のようなコードを使用できます。

```
foreach (DictionaryEntry entry in currentStatistics) {
   Console.WriteLine("SocketReads = {0}",
   currentStatistics["SocketReads"]);
   Console.WriteLine("SocketWrites = {0}",
    currentStatistics["SocketWrites"]);
}
Console.WriteLine();
```

Zen ADO.NET Entity Framework ユーザーへの注記：PsqlConnection の統計情報用のメソッドおよびプロパティは、Zen ADO.NET Entity Framework レイヤーでは利用できません。代わりに、データ プロバイダーは「擬似」ストアド プロシージャを介して同様の機能を公開します。詳細については、ADO.NET Entity Framework でのストアド プロシージャの使用を参照してください。

ADO.NET データ プロバイダー

Zen ADO.NET データ プロバイダーは、.NET 対応アプリケーションやアプリケーション サーバーにデータ アクセスを提供します。Zen ADO.NET データ プロバイダーにより、インターネットまたはイントラネット経由でデータ ストアにポイント ツー ポイント アクセスおよび *n* 層アクセスすることが可能になります。Zen ADO.NET データ プロバイダーは .NET 環境向けに最適化されているため、これを使用することにより、.NET 技術が組み込まれ、既存のシステムの機能とパフォーマンスを向上させることができます。

以下のトピックでは、Zen ADO.NET データ プロバイダーに関連する機能について説明します。

- Zen ADO.NET データ プロバイダーについて
- Zen ADO.NET データ プロバイダーでの接続文字列の使用
- パフォーマンスに関する考慮点
- データ型
- パラメーター配列

メモ：Zen ADO.NET Entity Framework でのデータ プロバイダーの使用法については、Zen ADO.NET Entity Framework データ プロバイダーを参照してください。

Zen ADO.NET データ プロバイダーについて

Zen ADO.NET データ プロバイダーはマネージ コードのみで構築されています。つまり、完全に共通言語ランタイム（CLR）の内部で、実行およびデータベースへの接続が行えます。

クライアント ライブラリや COM コンポーネントなどネイティブ オペレーティング システムで実行するコードはアンマネージ コードと言います。マネージ コードとアンマネージ コードは 1 つのアプリケーション内に混在させることができます。ただし、アンマネージ コードは共通言語ランタイムの外部にまで影響が及ぶため、現実的には複雑になり、パフォーマンスも低下します。また、セキュリティの危険にさらすことにもなりかねません。

Zen データ プロバイダーでサポートされる .NET Framework バージョンおよび Entity Framework バージョンの詳細については、サポートされる .NET Framework のバージョンを参照してください。

名前空間

Zen ADO.NET データ プロバイダーの名前空間は、Pervasive.Data.SqlClient です。Zen データベースに接続するときに、Pervasive.Data.SqlClient 名前空間で PsqlConnection オブジェクトと PsqlCommand オブジェクトを使用します。

次のコードは、Zen ADO.NET データ プロバイダーの名前空間をアプリケーションに組み込む方法を示しています。

C#

```
// Zen へアクセスします
using System.Data;
using System.Data.Common;
using Pervasive.Data.SqlClient;
```

Visual Basic

```
' Zen へアクセスします
Imports System.Data
Imports System.Data.Common
Imports Pervasive.Data.SqlClient
```

アセンブリ名

Zen ADO.NET データ プロバイダー用の厳密な名前のアセンブリは、インストール時にグローバル アセンブリ キャッシュ（GAC）に配置されます。アセンブリ名は Pervasive.Data.SqlClient.dll です。

Pervasive.Data.Common アセンブリには、バルク ロードのサポートなどの機能が含まれます。

Zen ADO.NET データ プロバイダーでの接続文字列の使用

接続の動作は、接続文字列や PsqlConnection オブジェクトのプロパティを使って定義することができます。しかし、接続文字列内の値の設定を接続プロパティによって変更することはできません。

接続文字列の基本形式は、セミコロンで区切られた一連の「キーワード / 値」のペアを含んでいます。次に、データ プロバイダー用の単純な接続文字列のキーワードと値の例を示します。

```
"Server DSN=SERVERDEMO;Host=localhost"
```

接続文字列を指定する際のガイドラインは、**接続文字列の使用**を参照してください。

接続文字列の構築

PsqlConnectionStringBuilder プロパティの名前は、接続文字列オプションの名前と同じです。ただし、接続文字列オプションの名前は、語と語の間に必要なスペースを入れて、複数の語で構成することができます。たとえば、Min Pool Size 接続文字列オプションは MinPoolSize プロパティに相当します。**接続文字列プロパティ**では、これらのプロパティの一覧を示し、各プロパティについて説明しています。

接続文字列オプションは次のような形式です。

```
オプション名 = 値
```

接続文字列オプション値の各ペアはセミコロンで区切ります。次に、Zen ADO.NET データ プロバイダー用の単純な接続文字列のキーワードと値の例を示します。

```
"Server DSN=SERVERDEMO;Host=localhost"
```

パフォーマンスに関する考慮点

アプリケーションのパフォーマンスは、接続文字列オプション、および、いくつかのデータ プロバイダー オブジェクトのプロパティに設定された値の影響を受けます。

パフォーマンスに影響を与える接続文字列オプション

Encrypt：データ暗号化は、データの暗号化と復号で必要となる追加オーバーヘッド（主に CPU 使用）のため、パフォーマンスに悪影響を与えることがあります。

Max Statement Cache Size：アプリケーションが使用する全プリペアド ステートメントをキャッシュすれば、最高のパフォーマンスを提供できるように思われます。しかし、この手法では、接続プールを使ってステートメント キャッシングを実装した場合、データベース サーバーのメモリに負担をかける結果になります。この場合、プールされた接続はそれぞれステートメント キャッシュを持ち、アプリケーションが使用する全プリペアド ステートメントを含むことになります。キャッシュされたプリペアド ステートメントは、すべてデータベース サーバーのメモリにも保持されます。

Pooling：データ プロバイダーが接続プールを使用できるようにしている場合、パフォーマンスに影響する以下の追加オプションを定義できます。

- **Load Balance Timeout**：接続をプールに保持する時間を定義できます。プール マネージャーは、接続がプールに返されたときに接続の作成時間をチェックします。作成時間を現在の時間と比較し、時間の間隔が Load Balance Timeout オプションの値を超えていたら、接続を破棄します。Min Pool Size オプションの指定によって、一部の接続でこの値を無視させることができます。
- **Connection Reset**：再利用される接続を初期設定に戻すと、その接続はサーバーに対して余分なコマンドを発行する必要が生じるため、パフォーマンスに悪影響を与えます。
- **Max Pool Size**：プールに格納できる接続数の最大値の設定が低すぎると、接続が使用可能になるまでの時間が延期されます。最大値の設定が高すぎると、リソースを無駄に消費します。
- **Min Pool Size**：接続プールは、一意な接続文字列を使って最初にデータベースへ接続したときに作成されます。Min Pool Size が指定されている場合、プールには最小数の接続が格納されます。接続プール内の一部の接続が Load Balance Timeout 値を超えたとしても、接続プールにはこの最小数の接続が保持されます。

Schema Options：ある種のデータベース メタデータを返すことは、パフォーマンスに影響を与えます。アプリケーションのパフォーマンスを最適化するため、データ プロバイダーは、パフォーマンスに悪影響を与えるプロシージャ定義やビュー定義などのデータベース メタデータを返さないようにします。アプリケーションがこれらのデータベース メタデータを必要とする場合は、明確にそれを返すよう要求する必要があります。

複数の種類の除外されたメタデータを返すには、名前をカンマ区切りリストで指定するか、返したい列コレクションの 16 進値の合計を指定します。たとえば、プロシージャ定義とビュー定義を返すには以下のいずれかを指定します。

- Schema Option=ShowProcedureDefinitions, ShowViewDefinitions
- Schema Options=0x60

Statement Cache Mode：ほとんどの場合、ステートメント キャッシングを有効にするとパフォーマンスが向上します。プリペアド ステートメント（コマンド インスタンス）のキャッシングを有効にするには、このオプションを Auto に設定します。アプリケーションが、暗黙的にステートメント キャッシュに含めるとマークされたプリペアド ステートメントを持つ場合は、この設定を使用します。または、暗黙的に含めるステートメントがいくつかあり、その他は明示的に含める場合、この設定を使用します。ステートメント キャッシュに Cache とマークされたプリペアド ステートメントのみを含めたい場合は、1）Command オブジェクトの StatementCacheBehavior プロパティに Cache を設定し、2）このオプションに ExplicitOnly を設定します。

パフォーマンスに影響を与えるプロパティ

StatementCacheBehavior：アプリケーションがその存続期間中にプリペアド ステートメントを複数回再使用する場合、ステートメント キャッシュを使用することによってパフォーマンスに影響を及ぼすことができます。このプロパティは、プリペアド ステートメント（Command オブジェクトのインスタンス）がステートメント キャッシング中にどのように処理されるかを決定します。

Cache に設定すると、プリペアド ステートメントはステートメント キャッシュに含められます。

Implicit に設定して Statement Cache Mode 接続文字列オプションに Auto を設定すると、プリペアド ステートメントはステートメント キャッシュに含められます。

DoNotCache に設定すると、プリペアド ステートメントはステートメント キャッシュから除外されます。

特定のステートメントをキャッシングすることによるパフォーマンスへの影響を判断するには、接続統計情報を使用できます（**接続統計情報によるパフォーマンスの分析**を参照してください）。

データ型

以下のトピックでは、Zen ADO.NET データ プロバイダーでサポートされるデータ型について説明します。

- Zen データ型から .NET Framework データ型へのマッピングでは、Zen データ型を .NET Framework 型にマップします。
- System.Data.DbTypes から PsqlDbTypes へのマッピングでは、System.Data.DbType のみが指定されている場合にデータ プロバイダーが使用するデータ型をマップします。
- .NET Framework 型から PsqlDbType へのマッピングでは、プロバイダー固有のデータ型も System.Data.DbType も指定されていない場合に、データ プロバイダーがデータ型を推定するために使用するデータ型をマップします。
- ストリーム オブジェクトでサポートされるデータ型では、長いデータ パラメーターへの入力としてストリームが使用される場合にデ ― タ プロバイダーが使用するデータ型をマップします。

Zen データ型から .NET Framework データ型へのマッピング

次の表では、Zen ADO.NET データ プロバイダーでサポートされるデータ型と、対応する .NET Framework 型を示します。この表で、DataAdapter を使って DataSet を埋めるときに使用するデータ型を確認してください。

また、この表では DataReader オブジェクトが直接使用される場合のデータの適切なアクセサーも示します。

- ［Zen データ型］列は、ネイティブな型名を示しています。
- ［PsqlDbType］列は、ADO.NET データ プロバイダーのデータ型の列挙を示します。基本的には、ネイティブ データ型と PsqlDbType とは 1 対 1 で対応しています。ただし、Zen データ型 NUMBER は、Decimal または Double のどちらにも対応しているので、この限りではありません。
- ［.NET Framework 型］列は .NET Framework で使用可能な基本データ型を示します。
- ［.NET Framework 型指定されたアクセサー］列は、DataReader を使用する場合に、この型の列へのアクセスに使用する必要があるメソッドを示します。

Zen データ型のマッピング

Zen データ型	PsqlDbType	.NET Framework 型	.NET Framework 型指定されたアクセサー
AUTOTIMESTAMP	Timestamp	DateTime	GetDateTime()
BFLOAT4	BFloat4	Single	GetSingle()
BFLOAT8	BFloat8	Double	GetDouble()
BIGIDENTITY	BigInt	Int64	GetInt64()
BIGINT	BigInt	Int64	GetDecimal()
BINARY	Binary	Byte[]	GetBytes()
BIT	Bit	Byte[]	GetBytes()
CHAR	Char	String Char[]	GetString() GetChars()
CURRENCY	Currency	Decimal	GetDecimal()
DATE	Date	DateTime	GetDateTime()
DATETIME	DateTime1[1]	DateTime	GetDateTime()
DECIMAL	Decimal	Decimal	GetDecimal()
DOUBLE	Double	Double	GetDouble()
FLOAT	Float	Double	GetDouble()
IDENTITY	Identity	Int32	GetInt32( )
INTEGER	Integer	Int32	GetInt32( )
LONGVARBINARY	LongVarBinary	Byte[]	GetBytes()
LONGVARCHAR	LongVarChar	Byte[]	GetBytes()
MONEY	Money	Decimal	GetDecimal()
NCHAR	NChar	String Char[]	GetString() GetChars()
NLONGVARCHAR	NLongVarChar	String Char[]	GetString() GetChars()
NUMERIC	Decimal	Decimal	GetDecimal()
NUMERICSA	DecimalSA	Decimal	GetDecimal()

Zen データ型	PsqlDbType	.NET Framework 型	.NET Framework 型指定されたアクセサー
NUMERICSTS	DecimalSTS	Decimal	GetDecimal()
NVARCHAR	NVarChar	String Char[]	GetString() GetChars()
REAL	Real	Single	GetSingle()
SMALLIDENTITY	SmallIdentity	Int16	GetInt16()
SMALLINT	SmallInt	Int16	GetInt16()
TIME	Time	Timespan[2]	GetValue()
TIMESTAMP、TIMESTAMP2	Timestamp	DateTime	GetDateTime()
TINYINT	TinyInt	SByte	GetByte()
UBIGINT	UBigInt	UInt64	GetUInt64()
UNIQUE_IDENTIFIER	UniqueIdentifier[1]	String	GetString()
UINTEGER	UInteger	UInt32	GetUInt32()
USMALLINT	USmallInt	UInt16	GetUInt16()
UTINYINT	UTinyInt	Byte	GetByte()
VARCHAR	VarChar	String Char[]	GetString() GetChars()

[1] Zen 9.5 以上でサポートされます。

[2] timetype 接続オプションの設定によって異なります。

パラメーター データ型のマッピング

パラメーターのデータ型は、データ プロバイダーごとに固有のものを使用します。Zen ADO.NET データ プロバイダーは、サーバーへ送信する前に、パラメーター値をネイティブの形式に変換する必要があります。アプリケーションでパラメーターを記述する最もよい方法は、データ プロバイダー固有の型の列挙を使用することです。

汎用プログラミングでは、データ プロバイダー固有の型を使用できない場合があります。プロバイダー固有の DB 型が指定されていない場合は、パラメーター値の System.Data.DbType または .NET Framework 型からデータ型が推定されます。

Zen ADO.NET データ プロバイダーは、次の順序を用いてパラメーターのデータ型を推定します。

- データ プロバイダー固有のデータ型が指定されている場合は、それを使用します。
- System.Data.DbType は指定されているけれども、データ プロバイダー固有のデータ型が指定されていない場合は、System.Data.DbType からデータ型を推定します。
- データ プロバイダー固有のデータ型も System.Data.DbType も指定されていない場合は、.NET Framework 型からデータ型を推定します。

System.Data.DbTypes から PsqlDbTypes へのマッピング

次の表は、System.Data.DbType のみが指定された場合に、データ プロバイダーがどのようにデータ型を推定するかを示します。

System.Data.DbType	PsqlDbType
AnsiString	VarChar
AnsiStringFixedLength	Char
Binary	Binary
Boolean	Integer
Byte	Integer
Currency	Currency
Date	Date
DateTime	DateTime[1]
Decimal	Decimal または Money
Double	Double
Float	Float
GUID	UniqueIdentifier*
Int16	SmallInt
Int32	Integer
Int64	BigInt
Int64	BigIdentity

System.Data.DbType	PsqlDbType
Sbyte	Integer
Single	BFloat4
String	NVarChar
StringFixedLength	NChar
Time	Time
Uint16	USmallInt
Uint32	UInteger
Uint64	UBigInt
VarNumeric	Decimal

[1] PSQL 9.5 以上でサポートされます。

.NET Framework 型から PsqlDbType へのマッピング

次の表では、プロバイダー固有のデータ型も System.Data.DbType も指定されない場合に、データ プロバイダーがデータ型を推定するために使用する対応表を示します。

.NET Framework 型	PsqlDbType
Boolean	Integer
Byte	Integer
Byte[]	Binary
DateTime	Timestamp
Decimal	Decimal
Double	Double
Int16	SmallInt
Int32	Integer
Int64	BigInt
Single	BFloat4
String	NVarChar VarChar（PvTranslate=Nothing の場合）

.NET Framework 型	PsqlDbType
Uint16	USmallInt
Uint32	UInteger
Uint64	UBigInt

ストリーム オブジェクトでサポートされるデータ型

Zen ADO.NET プロバイダーは、次の表に挙げられているデータ型に対し、長いデータ パラメーターへの入力としてストリームを使用することをサポートしています。

プロバイダー データ型	サポートされるストリーム型
LONGVARBINARY	Stream
LONGVARCHAR	TextReader

ストリームの説明については、長いデータ パラメーターへの入力としてストリームを使用するを参照してください。

長いデータ パラメーターへの入力としてストリームを使用する

ビデオ クリップや大量のドキュメントなど非常に大きなバイナリ値やテキスト値を表すために、不連続なメモリの使用を許可することによって、パフォーマンス、機能性、スケーラビリティが向上します。

バイナリ データの読み取りに使用されるストリーム オブジェクトは System.IO.Stream オブジェクトから派生し、Framework データ型の byte[] を使用します。

- System.IO.BufferedStream
- System.IO.FileStream
- System.IO.MemoryStream
- System.Net.Sockets.NetworkStream
- System.Security.Cryptography.CryptoStream

テキスト データを読み込むために使用するストリーム オブジェクトは System.IO.TextReader オブジェクトから派生し、Framework データ型の string を使用します。

- System.IO.StreamReader
- System.IO.StringReader

ストリームを使用できるようにするには、PsqlParameter オブジェクトの Value プロパティにストリームの特定のインスタンスを設定します（PsqlParameter オブジェクトを参照してください）。コマンドが実行されると、データ プロバイダーはストリームから読み込んで値を抽出します。

データ プロバイダーに付属する用例には、ランダムに生成されるデータを使用して LONGVARCHAR 列と LONGVARBINARY 列にデータを挿入するコード例が含まれています。また、用例では、ストリーム オブジェクトを LONGVARCHAR 列および LONGVARBINARY 列への入力として使用する方法も示しています。

パラメーター マーカー

パラメーター マーカー（ストアド プロシージャで使用するものも含む）は、Zen ADO.NET データ プロバイダーでは SQL ステートメントで疑問符 "?" 記号を使用することによって指定されます。

```
UPDATE emp SET job = ?, sal = ? WHERE empno = ?
```

パラメーターには名前が付いていないため、SQL ステートメントにあるパラメーターの順番どおりにバインドされなければなりません。つまり、PsqlParameterCollection オブジェクトの Add() メソッド（Parameter オブジェクトをコレクションに追加する）の呼び出しは、コマンド テキストに現れる "?" の順番どおりに発生する必要があるということです。

パラメーター配列

パラメーター配列のバインドは通常 INSERT ステートメントで使用され、テーブルを埋めるために必要な時間を短縮します。アプリケーションは 1 回のコマンド実行で複数のパラメーター値の行を指定することができます。それらの値を 1 回の往復（バックエンド データベース本来の能力によって異なる）でデータベース サーバーに送ることができます。

Zen ADO.NET データ プロバイダーでは、INSERT および UPDATE ステートメントに対して入力パラメーター配列をサポートします。

Zen ADO.NET Core データ プロバイダー

Zen ADO.NET Core データ プロバイダーは、.NET 対応アプリケーションやアプリケーション サーバーにデータ アクセスを提供します。これらは、インターネットおよびイントラネットを介して、主要なデータ ストアへの高パフォーマンスなポイント ツー ポイント アクセスおよび *n* 層アクセスを提供します。Zen ADO.NET Core データ プロバイダーは .NET 環境向けに最適化されているため、これを使用することにより、.NET Core 技術が組み込まれ、既存のシステムの機能とパフォーマンスを向上させることができます。

以下のトピックでは、Zen ADO.NET Core データ プロバイダーの機能について説明します。

- Zen ADO.NET Core データ プロバイダーについて
- Visual Studio での Zen ADO.NET Core DLL を使用したアプリケーションの作成
- Visual Studio での Zen ADO.NET Core データ プロバイダーを使用した UWP アプリケーションの作成
- Zen ADO.NET Core データ プロバイダーにない ADO.NET データ プロバイダーの機能

メモ：ADO.NET Entity Framework Core でのデータ プロバイダーの使用法については、Zen ADO.NET Entity Framework Core データ プロバイダーを参照してください。

Zen ADO.NET Core データ プロバイダーについて

Zen ADO.NET Core データ プロバイダーは以下をサポートしています。

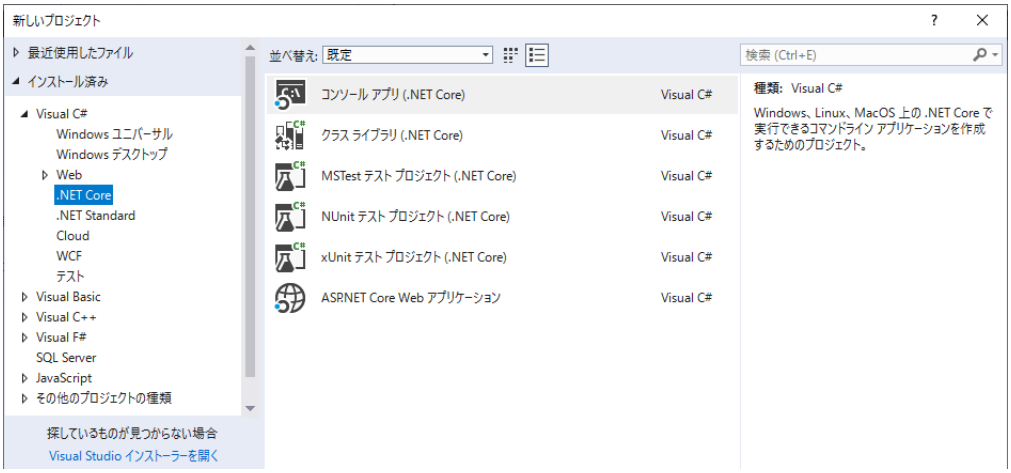
- Visual Studio 2022 での ADO.NET 4.5 および 4.6 プロバイダーの使用
- Zen Data Tools と Visual Studio との統合については、ADO.NET SDK readme を参照してください。
- Visual Studio Code
- .NET 6、7、および 8
- UWP アプリケーション

Visual Studio での Zen ADO.NET Core DLL を使用したアプリケーションの作成

以下の手順を使用する前に、まず、SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーに示されているお使いのバージョンの最新 SDK の .zip アーカイブをダウンロードして、NuGet パッケージ Pervasive.Data.SqlClientStd を展開します。

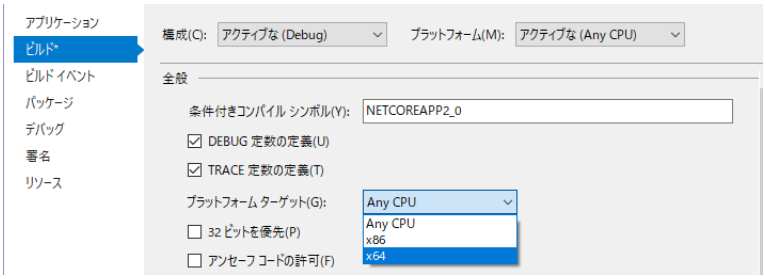
Visual Studio で Zen ADO.NET Core DLL を使用してアプリケーションを作成するには

1. Visual Studio で、［**ファイル**］メニューの［**新規作成**］をポイントして［**プロジェクト**］をクリックします。［新しいプロジェクト］ウィンドウが表示されます。
2. 左ペインの［**インストール済み**］リストで、Visual C# の **.NET Core** テンプレートを選択します。



3. 中央のペインで、必要なプロジェクトの種類を選択します。
4. プロジェクトの名前と場所を適切なフィールドに入力して［**OK**］をクリックします。
5. プロジェクトを右クリックして［**プロパティ**］をクリックします。

6. ［**ビルド**］ペインで、［**プラットフォーム ターゲット**］リストから必要なプラットフォームを選択します。



7. ダウンロードした NuGet パッケージをプロジェクトに追加します。

   NuGet パッケージをローカルに追加する方法については、次を参照してください。
   https://stackoverflow.com/questions/10240029/how-do-i-install-a-nuget-package-nupkg-file-locally/38663739#38663739

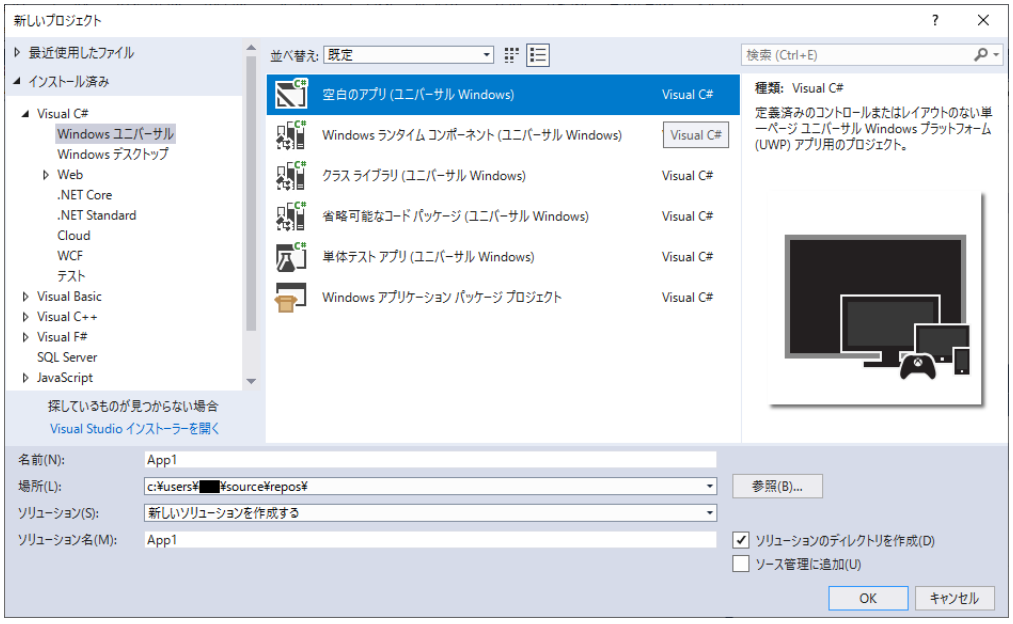
Visual Studio での Zen ADO.NET Core データ プロバイダーを使用した UWP アプリケーションの作成

以下の手順を使用する前に、まず、SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーに示されているお使いのバージョンの最新 SDK の .zip アーカイブをダウンロードして、NuGet パッケージ Pervasive.Data.SqlClientStd を展開します。

メモ：Zen ADO.NET Core DLL を使用する UWP アプリケーションの場合、Windows 10 オペレーティング システムのバージョンは 1709（OS Build 16299）以降が必要です。

Visual Studio で Zen ADO.NET Core データ プロバイダーを使用して UWP アプリケーションを作成するには

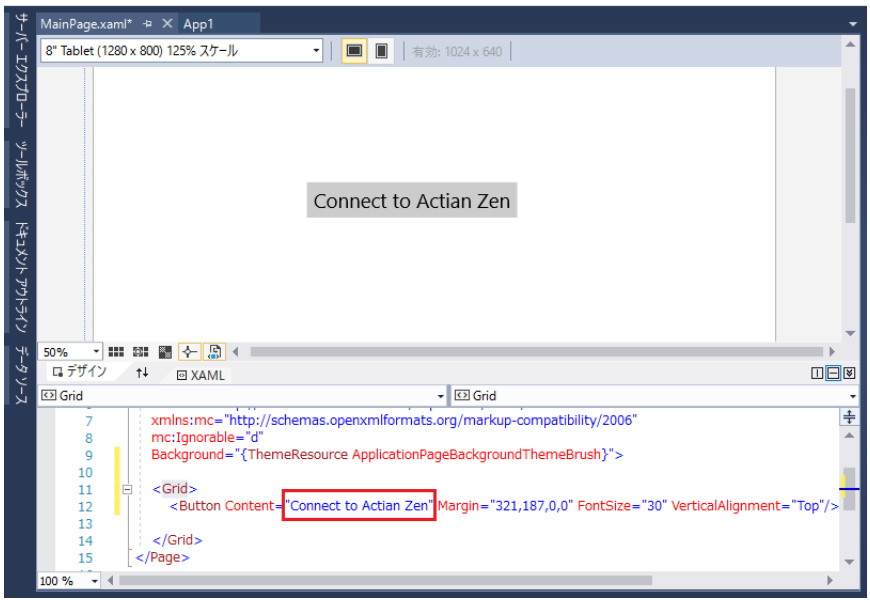
1. Visual Studio で、［**ファイル**］メニューの［**新規作成**］をポイントして［**プロジェクト**］をクリックします。［新しいプロジェクト］ウィンドウが表示されます。
2. 左ペインの［インストール済み］リストで Visual C# の "**Windows ユニバーサル**" を選択し、次に中央のペインで "**空白のアプリ（ユニバーサル Windows）**" を選択します。



3. プロジェクトの名前と場所を適切なフィールドに入力して［**OK**］をクリックします。

4. ［最小バージョン］リストで **Windows 10 Fall Creators Update（10.0; ビルド 16299）**を選択します。

5. MainPage.xaml ファイルでボタンを追加し、そのボタンの名前を **Connect to Actian Zen** に変更します。



6. ボタンをダブルクリックして実装を開きます。

7. ボタンの実装コードに以下を追加します。

```
private void Button_Click(object sender, RoutedEventArgs e)
{


    try
    {
        PsqlConnection conn = new PsqlConnection("Host=someHost;Port=1583;ServerDSN=DEFAULTDB;");

        conn.Open();
        Debug.WriteLine("Connection Opened: ");
    }

    catch (Exception eSql)
    {
        Debug.WriteLine("Exception Message: " + eSql.Message);
    }
}
```

8. ダウンロードした NuGet パッケージをプロジェクトに追加します。

   NuGet パッケージをローカルに追加する方法については、次を参照してください。https://stackoverflow.com/questions/10240029/how-do-i-install-a-nuget-package-nupkg-file-locally/38663739#38663739

9. データベースがリモート コンピューターで利用できる場合は、package.appxmanifest ファイルの［機能］タブで［**プライベート ネットワーク（クライアントとサーバー)**］チェック ボックスをオンにします。

10. アプリケーションをビルドして実行します。

Zen ADO.NET Core データ プロバイダーにない ADO.NET データ プロバイダーの機能

以下の ADO.NET データ プロバイダーの機能は、Zen ADO.NET Core データ プロバイダーにはありません。

- **パフォーマンス カウンター**：パフォーマンス カウンターは ADO.NET Core でサポートされていないので、Zen ADO.NET Core データ プロバイダーでもサポートされません。
- **エンコード**：Zen ADO.NET データ プロバイダーは Windows ベースのアプリケーション専用に作成されたため、デフォルトのエンコードとして Windows-1252 エンコード（コード ページ 1252 で識別される）を使用します。

  Zen ADO.NET Core データ プロバイダーはクロス プラットフォームであるため、デフォルトのエンコードとして現在のオペレーティング システムのデフォルトのエンコードを使用します。
- **PervasiveFactory の CreatePermission**：CreatePermission(PermissionState) メソッドは ADO.NET Core でサポートされていないので、Zen ADO.NET Core データ プロバイダーでもサポートされません。
- **ファイル スキーマ**：Zen ADO.NET データ プロバイダーは、ユーザーにデータをプッシュする際、Char、VarChar、および LongVarChar 列の Encoding.BodyName を使用します。Encoding.BodyName は ADO.NET Core でサポートされていないので、Zen ADO.NET Core データ プロバイダーは Encoding.WebName を使用します。

Zen ADO.NET Entity Framework データ プロバイダー

Zen ADO.NET Entity Framework は、.NET Framework 用のオブジェクト リレーショナル マッピング（ORM）フレームワークです。このフレームワークを使用すると、開発者はリレーショナル ストレージ スキーマに対して直接プログラミングするのではなく、概念アプリケーション モデルに対して行うことで、データ アクセス アプリケーションを作成できます。このモデルにより、データ中心のアプリケーションで書かれ、保守されるコードの量を減らすことができます。

Zen ADO.NET Entity Framework データ プロバイダー（以前の Pervasive ADO.NET Entity Framework データ プロバイダー）は、ADO.NET Entity Framework を使用するアプリケーションで使用できます。

Zen ADO.NET Entity Framework データ プロバイダーは、Microsoft ADO.NET Entity Framework のバージョン 6.1、6.1.1、および 6.1.2 と互換性があります。以下のプログラミング機能をサポートします。

- SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーに記載されている、.NET Framework 4.0 を対象とするアプリケーション
- Database First、Code First、および Model First のワークフロー
- すべてのワークフローでの列挙型のサポート
- Code First Migrations
- POCO（"Plain-old" CLR object）エンティティ
- DbContext クラス
- 複数の DBContext クラス
- 挿入、更新、および削除ストアド プロシージャへの Code First のマッピング
- 構成可能な移行履歴
- 接続の復元性
- Code First Migrations のインデックス属性
- 関数インポートのトランザクションを無効にする
- Enum.HasFlag のサポート

- Migrations コマンドで、プロジェクトではなく参照からコンテキストを使用できるようにする
- web/app.config および DatabaseLogger のインターセプター
- '_' で始まる識別子のサポート
- 文字列と数値が連結されたプロパティの選択

Zen ADO.NET Entity Framework データ プロバイダーは、ADO.NET データ プロバイダーを使用して ADO.NET データベース サーバーと通信します。つまり、Zen ADO.NET データ プロバイダーによって定義された機能は、ここで特に断りがない限り、Zen ADO.NET Entity Framework データ プロバイダーに適用されるということです。同様に、Zen ADO.NET データ プロバイダーに合わせて作られたすべてのパフォーマンス構成が、Zen ADO.NET Entity Framework データ プロバイダーで実現されます。

Zen ADO.NET Entity Framework データ プロバイダーについて

Zen ADO.NET Entity Framework データ プロバイダーはマネージ コードのみで構築されています。つまり、完全に共通言語ランタイム（CLR）の内部で、実行およびデータベースへの接続が行えます。

クライアント ライブラリや COM コンポーネントなどネイティブ オペレーティング システムで実行するコードはアンマネージ コードと言います。マネージ コードとアンマネージ コードは 1 つのアプリケーション内に混在させることができます。ただし、アンマネージ コードは共通言語ランタイムの外部にまで影響が及ぶため、現実的には複雑になり、パフォーマンスも低下します。また、セキュリティの危険にさらすことにもなりかねません。

名前空間

Zen ADO.NET Entity Framework データ プロバイダーの名前空間は、Pervasive.Data.SqlClient.Entity です。

メモ：Pervasive.Data.SqlClient.Entity 名前空間は、Microsoft ADO.NET Entity Framework バージョン 5.0（EF 5）と 6.1（EF 6.1）に共通です。

アセンブリ名

Zen ADO.NET Entity Framework データ プロバイダーは、Pervasive.Data.SqlClient.Entity.dll という名前のアセンブリを使用します。

EF 6.1 を参照する場合は、次を選択します。

```
%windir%¥Microsoft.NET¥assembly¥GAC_MSIL¥Pervasive.Data.SqlClient.Entity¥
v4.0_4.6.0.0__c84cd5c63851e072
```

Entity Framework 6.1 の構成

Zen ADO.NET Entity Framework データ プロバイダーは、Microsoft ADO.NET Entity Framework バージョン 5.0（EF5）および 6.1（EF 6.1）をサポートします。

EF 6.1 を使用するには、まず、次のいずれかの方法を使用して登録する必要があります。

- 構成ファイル登録
- コード ベース登録

メモ：EF 6.1 を登録する場合、アプリケーションをローカルでテストする限りは、開発時にコード ベース登録を実行できます。しかし、プロジェクトを配置する場合は、構成ファイル登録の実行が必要になります。

構成ファイル登録

構成ファイルを更新して EF 6.1 を構成するには

1. **EntityFramework 6.1.2 NuGet** パッケージをインストールします。

   app.config ファイルが作成されます。

2. app.config ファイルから **defaultConnectionFactory** 登録セクションを削除して、次のコードに置き換えます。

```
<providers>
<provider invariantName="Pervasive.Data.SqlClient"
type="Pervasive.Data.SqlClient.Entity.PsqlProviderServices, Pervasive.Data.SqlClient.Entity,
Version=4.6.0.0, Culture=neutral, PublicKeyToken=c84cd5c63851e072" />
</providers>
```

EF 6.1 プロバイダー登録が、app.config ファイルの Entity Framework セクションに追加されます。

コード ベース登録

コード ベース登録を使用して EF 6.1 を構成するには

1. 次の新しい *DbConfiguration* クラスをテスト アプリケーションに追加します。

```
public class MyConfiguration : DbConfiguration
{
public MyConfiguration()
```

```
{
SetProviderServices("PsqlProviderServices.ProviderInvariantName, new PsqlProviderServices());
}
}
```

2. 次のアノテーションを *DBContext* クラスの上に追加します。

```
[DbConfigurationType(typeof(MyConfiguration))]
```

Zen ADO.NET Entity Framework データ プロバイダーでの接続文字列の使用

Zen ADO.NET Entity Framework は接続文字列に含まれている情報を使用して、基となる Entity Framework をサポートする ADO.NET データ プロバイダーに接続します。接続文字列には、必要なモデルおよびマッピング ファイルに関する情報も含まれています。

データ プロバイダーは、モデルにアクセスしたり、メタデータをマップしたり、データ ソースに接続したりする場合に接続文字列を使用します。

Entity Framework ウィザードで既存の接続を指定するか、または新しい接続を定義することができます。接続文字列オプションは、接続文字列で直接定義できますが、Visual Studio の［詳細プロパティ］ダイアログ ボックスでも設定できます（サーバー エクスプローラーでの接続の追加を参照してください）。

サーバー エクスプローラーでの接続文字列の定義

Visual Studio を使って接続を追加、変更する方法については、サーバー エクスプローラーでの接続の追加を参照してください。

接続文字列オプションのデフォルト値の変更

ADO.NET Entity Framework データ プロバイダーで使用される接続文字列オプションのデフォルト値の多くは、Zen ADO.NET データ プロバイダーで使用されるデフォルト値と同じです。次の表は、ADO.NET Entity Framework アプリケーションで接続文字列オプションを使用する場合には、違うデフォルト値になるオプションを示しています。

接続文字列オプション	ADO.NET Entity Framework アプリケーションでのデフォルト値
Parameter Mode	サポートされません。
Statement Cache Mode	ExplicitOnly 値のみサポートされます。

Code First および Model First のサポート

Entity Framework 4.1 以降のプロバイダーは、Model First および Code First の機能をサポートしています。これらの機能のサポートを実装すると、たとえば、長い識別子名の

処理方法など、データ プロバイダーへの変更が必要とされます。ただし、これらの変更はアプリケーションの変更を必要としません。

Code First および Model First の実装は、型マッピングの変更を必要とします。詳細については、データ型および関数のマッピングを参照してください。

長い識別子名の処理

ほとんどの Zen 識別子の最大長は 20 バイトです。サーバー上で作成するオブジェクトの名前はクラス名とプロパティ名から取られるため、識別子名がこのサイズを超える場合があります。また、制約名は多くの場合、いくつかのオブジェクト名を連結して作成されます。このような場合には、識別子の最大長を超える可能性がさらに大きくなります。

データ プロバイダーは、識別子の末尾を整数のハッシュ コードに置き換えることにより、識別子をデータベースで許容される識別子の最大長に短縮します。たとえば、文字列 `ColumnMoreThanTwentyCharacters` は `ColumnMor_2873286151` に短縮されます。DB ツールを使用して DB オブジェクトのアクセスや表示を行った場合、作成されたテーブルの名前が、POCO（Plain Old CLR Object）のクラス名とプロパティ名（Code First）、またはエンティティ名とエンティティのプロパティ名（Model First）に基づいて予想される名前とは異なる場合があります。

同じ先頭文字を持つ 2 つの識別子が短縮されると、識別子間の違いが目で見てわかりづらくなることに注意してください。たとえば、テーブルに 2 つのサポートするシーケンス、`ColumnMoreThanTwentyCharacters` と `ColumnMoreThanTwenty1Characters` があるとします。これらのシーケンスが短縮された場合、それぞれ `ColumnMor_2873286151` と `ColumnMor_672399971` に名前が変更されます。

ADO.NET Entity Framework での Code First Migrations の使用

Entity Framework 4.3 以降では Code First Migrations をサポートしています。これは、POCO クラスを削除して再作成しなくても、このクラスを反映するようにデータベース スキーマを更新することを可能にします。

移行は、モデルの変更に従って、データベース スキーマを増分で更新できるようにします。データベースの変更の各セットは、移行として知られるコード ファイルで表さ

れます。移行は、通常、タイムスタンプを使用して管理され、データベース内のテーブルは、どの移行が適用されたかを追跡します。

Code First Migrations の実装は型マッピングの変更を必要とします。詳細については、データ型および関数のマッピングを参照してください。

Progress DataDirect Connect for Zen ADO.NET データ プロバイダーを使用して Code First Migrations を実装するには、以下の追加設定を実行する必要があります。

1. プロジェクト内の Pervasive.Data.SQLClient.Entity アセンブリに参照を追加します。
2. Configuration Class の変更を継承し、Configuration Class のコンストラクターで SQL Generator を登録します。次のことを行います。
   - PervasiveDbMigrationsConfiguration <TContext> から Configuration Class を継承します。たとえば、次のように指定します。

```
internal sealed class Configuration: PervasiveDbMigrationsConfiguration<%Context Name%>
```

   - Class Generator を登録します。

パッケージ マネージャー コンソールを使用して移行を有効にした後、configuration.cs ファイルの追加設定に加えて、app.config または configuration.cs ファイルのいずれかで接続文字列を指定します。ただし、接続文字列を app.config ファイルに指定する場合は、接続文字列とコンテキストの名前が同じであるようにしてください。

接続文字列を app.config ファイルに指定する場合は、使用するプロバイダーのバージョンに応じて、次の構文を使用して app.config ファイルで SQL Generator を登録します。

```
<providers>
    <provider invariantName="Pervasive.Data.SqlClient"
    type="Pervasive.Data.SqlClient.Entity.PsqlProviderServices, Pervasive.Data.SqlClient.Entity,
    Version=4.5.0.6, Culture=neutral, PublicKeyToken=c84cd5c63851e072" />
</providers>
```

configuration.cs で SQL Generator を登録するには、次の構文を使用します。

```
SetSqlGenerator(PervasiveConnectionInfo.InvariantName, new PervasiveEntityMigrationSqlGenerator());
```

ADO.NET Entity Framework での列挙型の使用

enum キーワードは、列挙型を宣言するために使用されます。列挙型は、列挙子リストと呼ばれる名前付き定数の集まりで構成される固有の型です。すべての列挙型には基になる型があります。デフォルトでは、列挙要素の基になるすべての型は int32 にマップされます。デフォルトでは、最初の列挙子の値は 0 で、後続の列挙子の値は 1 ずつ増加していきます。たとえば、曜日の列挙型は次のように指定します。

```
enum Days {MON, TUE, WED, THU, FRI, SAT, SUN};
```

この列挙型では、`MON` は `0`、`TUE` は `1`、`WED` は `2` などとなります。列挙子は、デフォルト値をオーバーライドする初期化子を持つことができます。たとえば、次のように指定します。

```
enum Days {MON=1, TUE, WED, THU, FRI, SAT, SUN};
```

この列挙型では、要素の並びは 0 ではなく 1 から開始します。列挙型のフィールドの名前は大文字です。これらは定数であるため、慣例により大文字にします。

Microsoft ADO.NET Entity Framework 5.0 以降は列挙型をサポートしています。列挙型の機能を使用するには、.NET Framework 4.5 以降を対象とする必要があります。Visual Studio 2019 の対象はデフォルトで .NET Framework 4.7.2 です。列挙型は 3 つのワークフロー、すなわち、Model First、Code First、および Database First でサポートされています。

Entity Framework では、列挙型の基になる型は次のとおりです。

- Byte
- Int16
- Int32
- Int64
- SByte

デフォルトでは、列挙型は Int32 型です。他の整数型はコロンを使用して指定します。

```
enum Days : byte{MON=1, TUE, WED, THU, FRI, SAT, SUN};
```

基になる型は、列挙子ごとに割り当てるストレージの大きさを指定します。ただし、enum 型を整数型に変更するには、明示的なキャストが必要です。enum の実装は型マッピングの変更もサポートしています。詳細については、**データ型および関数のマッピング**を参照してください。

Entity Framework の一部として、Entity Developer が［モデル エクスプローラー］ウィンドウで新しい Enum ノードを提供することにより、enum 型を完全にサポートしています。Enum プロパティは、他のスカラー プロパティと同じように、LINQ クエリや更新などで使用できます。

データ型および関数のマッピング

ADO.NET Entity Framework を使用すると、開発者はリレーショナル ストレージ スキーマに対して直接プログラミングする代わりに、概念アプリケーション モデルに対してプログラミングすることで、データ アクセス アプリケーションを作成できます。

Database First の型マッピング

Database First モデルでは、データ プロバイダーはストア中心の型マッピング方式を使用します。このマッピングでは、Zen（ストア）型は、モデルの生成時に使用される EDM 型に影響を与えます。

Database First の型マッピング

次の表は、Zen 型から、Database First モデルで使用されるプリミティブ型へのマッピングを示しています。一部の Zen データ型はいくつかの異なる EDM 型にマップすることができます。デフォルト値は斜体で示されています。

列は次のように定義されています。

- ［Zen 型］列は、ネイティブな型名を示しています。
- ［ストア（SSDL）型］列は、ストア スキーマ定義言語（SSDL）ファイルで使用されるデータ型を示しています。ストレージ メタデータ スキーマは、EDM で構築されたアプリケーションのデータを保持する、データベースの形式的記述です。
- ［PrimitiveTypeKind］列は、EDM アプリケーションを定義するのに使用されるエンティティのプロパティの、有効な内容を指定するために使用される共通のデータ プリミティブを示しています。

Zen 型	ストア（SSDL）型	PrimitiveTypeKind
AUTOTIMESTAMP	DateTime	DateTime
BFLOAT4	BFloat4	Single
BFLOAT8	BFloat8	Double
BIGIDENTITY	Bigint	Int64
BIGINT	Bigint	Int64
BINARY	binary	Byte[]

Zen 型	ストア（SSDL）型	PrimitiveTypeKind
BIT	Bit	Boolean
CHAR	Char	String
CURRENCY	Currency	Decimal
DATE	Date	DateTime
DATETIME	DateTime	DateTime
DECIMAL	Decimal	Decimal
DOUBLE	Double	Double
FLOAT	Float	Float
IDENTITY	Identity	Int32
INTEGER	Integer	Int32
LONGVARBINARY	LongVarBinary	Byte[]
LONGVARCHAR	LongVarChar	String
MONEY	Money	Decimal
NCHAR	NChar	String
NLONGVARCHAR	NLongVarChar	String
NUMERIC	Decimal	Decimal
NUMERICSA	DecimalSA	Decimal
NUMERICSTS	DecimalSTS	Decimal
NVARCHAR	NVarChar	String
REAL	Real	Single
ROWID	Rowid	Binary
SMALLIDENTITY	SmallIdentity	Int16
SMALLINT	Smallint	Int16
TIME	Time	Time
TIMESTAMP、TIMESTAMP2	DateTime	DateTime
TINYINT	TinyInt	SByte
UBIGINT	UBigInt	UInt64

Zen 型	ストア（SSDL）型	PrimitiveTypeKind
UNIQUE_IDENTIFIER	Guid	Guid
UINTEGER	UInteger	UInt32
USMALLINT	USmallInt	UInt16
UTINYINT	UTinyInt	Byte
VARCHAR	Varchar	String

Model First の型マッピング

次の表はモデル中心の型マッピングを示しています。このマッピングでは、EDM 単純型は、データベースの作成に使用される Zen（ストア）型に影響を与えます。列は次のように定義されています。

- ［PrimitiveTypeKind］列は、EDM アプリケーションを定義するのに使用されるエンティティのプロパティの、有効な内容を指定するために使用される共通のデータ プリミティブを示しています。
- ［型マッピングに影響を与えるプロパティ値］列は、型マッピングに影響を及ぼす可能性のある、あらゆるプロパティ値を示しています。
- ［ストア（SSDL）型］列は、ストア スキーマ定義言語（SSDL）ファイルで使用されるデータ型を示しています。ストレージ メタデータ スキーマは、EDM で構築されたアプリケーションのデータを保持する、データベースの形式的記述です。
- ［Zen 型］列は、ネイティブな型名を示しています。

PrimitiveTypeKind	型マッピングに影響を与えるプロパティ値	ストア（SSDL）型	Zen 型
Binary	FixedLength：TRUE FixedLength：FALSE	Binary LongVarBinary	Binary(n) LongVarBinary
Boolean		Boolean	Bit
Byte		Tinylint_as_byte	TinyInt
DateTime		DateTime	DateTime
Decimal		Decimal	Decimal
Double		Double	Double
Guid		Guid	Guid

PrimitiveTypeKind	型マッピングに影響を与えるプロパティ値	ストア（SSDL）型	Zen 型
Single		Float	Float
SByte		Smallint_as_Sbyte	Smallint
Int16		SmallInt	Smallint
Int32		Integer	Integer
Int64		Bigint	BigInt
String	MaxLength= (1<=n<=8000) Fixed Length=True Unicode=False	Char	Char(n)
	MaxLength= (1<=n<=8000) Fixed Length=False Unicode=False	Varchar	Varchar(n)
	MaxLength= (>8000) Fixed Length=False Unicode=False	LongVarChar	LongVarchar
	MaxLength= (1<=n<=4000) Fixed Length=True Unicode=True	NChar	NChar(n)
	MaxLength= (1<=n<=4000) Fixed Length=False Unicode=True	NVarChar	NVarChar(n)
	MaxLength= (>4000) Fixed Length=False Unicode=True	NLongVarChar	NLongVarChar
Time		Time	Time
DateTimeOffset		DateTime	DateTime

Code First の型マッピング

次の表はモデル中心の型マッピングを示しています。このマッピングでは、CLR 型は、データベースの作成時に使用される Zen（ストア）型に影響を与えます。一部の CLR

型はいくつかの異なる Zen 型にマップすることができます。デフォルト値は斜体で示されています。

列は次のように定義されています。

- ［CLR 型］列は、共通言語ランタイムの型名を示しています。
- ［Zen 型］列は、ネイティブな型名を示しています。

CLR 型	Zen データ型
Byte[ ]	BINARY
Boolean	BIT
Byte	TINYINT
DateTime	DATETIME
Decimal	DECIMAL
Double	DOUBLE
Guid	UNIQUEIDENTIFIER BINARY
Single	FLOAT
Sbyte	SMALLINT
Int16	SMALLINT
Int32	INTEGER
Int64	BIGINT
String[1]	NCHAR NVARCHAR NLONGVARCHAR
TimeSpan	TIME
DateTimeOffset	DateTime

[1] Code First のワークフローでは、エンティティ内の文字列フィールドの長さが指定されていない場合、データ プロバイダーは Unicode 型と非 Unicode 型のデフォルトの長さをそれぞれ 2048 バイトと 4096 バイトに設定します。ただし、文字列フィールドの長さが最大許容限度に設定されている場合、つまり、Unicode 型では 4000 バイト、非 Unicode 型では 8000 バイトに設定されている場合、データ プロバイダーはそれぞれを 2048 バイトと 4096 バイトにリセットします。その他、文字列フィールドの長さが指定されているすべてのシナリオでは、データ プロバイダーは指定された長さを使用します。

EDM 正規関数から Zen 関数へのマッピング

ADO.NET Entity Framework は、エンティティ データ モデル（EDM）正規関数を Zen 用 ADO.NET Entity Framework データ プロバイダーの対応するデータ ソース機能に変換します。これによって、全データ ソースに共通する形式で表現される関数を呼び出すことができます。

これらの正規関数はデータ ソースから独立しているため、正規関数の引数の型と戻り値の型は、EDM の型の語句で定義されます。Entity SQL クエリで正規関数を使用すると、データ ソースで適切な関数が呼び出されます。

すべての正規関数には、ヌルが入力された場合の動作とエラー状況が明示的に指定されています。ただし、ADO.NET Entity Framework はこの動作を実行しません。詳細は、http://msdn.microsoft.com/ja-jp/library/bb738626.aspx で入手できます。

集計正規関数

次の表は、EDM 集計正規関数から Zen 関数へのマッピングを示します。

集計正規関数	Zen 関数
Avg(*expression*)	avg(*expression*)
BigCount(*expression*)	count(*expression*)
Count(*expression*)	count(*expression*)
Max(*expression*)	max(*expression*)
Min(*expression*)	min(*expression*)
StDev(*expression*)	stdev(*expression*)
StDevP(*expression*)	stdevp(*expression*)
Sum(*expression*)	sum(*expression*)
Var(*expression*)	var(*expression*)
VarP(*expression*)	varp(*expression*)

数学正規関数

次の表は、EDM 数学正規関数から Zen 関数へのマッピングを示します。ただし、処理する列が decimal 値または integer 値のみを含んでいる場合に使用される関数を対象としています。

詳細については、**数値関数**を参照してください。また、弊社 Web サイトで Zen のバージョン番号を選択し、**開発者リファレンス** > **データ アクセス方法** > **SQL Engine Reference** > **SQL 構文リファレンス**を参照してください。

数学正規関数	Zen 関数
Abs(*value*)	abs(*value*)
Ceiling(*value*)	ceiling(*value*)
Floor(*value*)	floor(*value*)
Power(*value*, *exponent*)	power(*value*, *exponent*)
Round(*value*)	round(*numeric_expression1*, *integer_expression2*)
Round(*value*, *digits*)	round(*value*, *digits*)
Truncate(*value*, *digits*)	truncate(*value*, *digits*)

日付と時刻の正規関数

次の表は、EDM の日付と時刻の正規関数から Zen 関数へのマッピングを示します。これらの関数は、DATE や TIME などのデータ型から成るデータの生成、処理、および操作を行います。

日付と時刻の正規関数	Zen 関数
AddNanoseconds(expression, number)	dateadd(millisecond, number/1000000)
AddMicroseconds(expression, number)	dateadd(millisecond, number/1000)
AddMilliseconds(expression, number)	dateadd(millisecond, number)
AddSeconds(expression, number)	dateadd(second, number)
AddMinutes(expression, number)	dateadd(minute, number)
AddHours(expression, number)	dateadd(hour, number)
AddDays(expression, number)	dateadd(day, number)
AddMonths(expression, number)	dateadd(month, number)
AddYears(expression, number)	dateadd(year, number)
CreateDateTime(year, month, day, hour, minute, second)	datetimefromparts(year, month, day, hour, minute, second,0)

日付と時刻の正規関数	Zen 関数
CreateDateTimeOffset(year, month, day, hour, minute, second, tzoffset)[1]	datetimeoffsetfromparts(year, month, day, hour, minute, second, tzoffset)
CreateTime(hour, minute, second)[1]	timefromparts(hour, minute, second, 0, 0)
CurrentDateTime()	now()
CurrentDateTimeOffset()	sysdatetimeoffset()
CurrentUtcDateTime()	current_timestamp()
Day(expression)	datepart(day, expression)
DayOfYear(startexpression, endexpression)	dayofyear(expression)
DiffNanoSeconds(startexpression, endexpression)	datediff(millisecond, startexpression, endexpression)*1000000
DiffMilliSeconds(startexpression, endexpression)	datediff(millisecond, startexpression, endexpression)
DiffMicroSeconds(startexpression, endexpression)	datediff(millisecond, startexpression, endexpression)*1000
DiffSeconds(startexpression, endexpression)	datediff(second, startexpression, endexpression)
DiffMinutes(startexpression, endexpression)	datediff(minute, startexpression, endexpression)
DiffHours(startexpression, endexpression)	datediff(hour, startexpression, endexpression)
DiffDays(startexpression, endexpression)	datediff(day, startexpression, endexpression)
DiffMonths(startexpression, endexpression)	datediff(month, startexpression, endexpression)
DiffYears(startexpression, endexpression)	datediff(year, startexpression, endexpression)
GetTotalOffsetMinutes(DateTime Offset)	datepart(tzoffset, expression)
Year(expression)	datepart(year, expression)
Month(expression)	datepart(month, expression)
Day(expression)	datepart(day, expression)
Hour(expression)	datepart(hour, expression)
Minute(expression)	datepart(minute, expression)
Second(expression)	datepart(second, expression)
Millisecond(expression)	datepart(millisecond, expression)

日付と時刻の正規関数	Zen 関数
TruncateTime(expression)	convert(expression, SQL_DATE)

[1] Zen v11.30 Update 4（May 2013）が必要です。

ビット単位の正規関数

次の表は、EDM ビット単位の正規関数から Zen 関数へのマッピングを示します。

ビット単位の正規関数	Zen 関数
BitWiseAnd(*value1*, *value2*)	bit_and(*value1*, *value2*)
BitWiseNot(*value*)	bit_compliment
BitWiseOr(*value1*, *value2*)	bit_or
BitWiseXor(*value1*, *value2*)	bit_xor

文字列正規関数

次の表は、EDM 文字列正規関数から Zen 関数へのマッピングを示します。

文字列正規関数	Zen 関数
Concat(*string1*, *string2*)	concat(*string1*, *string2*)
Contains(*string*, *target*)	contains(*string*, *target*)
EndsWith(*string*, *target*)	endswith(*string*, *target*)
IndexOf(*target*, *string2*)	instr(*target*, *string2*)
Left(*string1*, *length*)	left(*string1*, *length*)
Length(*string*)	length(*string*)
LTrim(*string*)	ltrim(*string*)
Trim(*string*)	trim(BOTH FROM *string*)
Replace(*string1*, *string2*, *string3*)	replace(*string1*, *string2*, *string3*)
Reverse(*string*)	reverse(*string*)
RTrim(*string*)	rtrim(*string*)
StartsWith(*string*, *target*)	startswith(*string*, *target*)

文字列正規関数	Zen 関数
Substring(*string*, *start*, *length*)	INCOMPLETE regexpr_substr(...)
ToLower(*string*)	lower(*string*)
ToUpper(*string*)	upper(*string*)

その他の正規関数

次の表は、その他の正規関数から Zen 関数へのマッピングを示します。

その他の正規関数	Zen 関数
NewGuid()	newid()

Entity Framework 機能の拡張

ADO.NET Entity Framework は、多くの ADO.NET 機能をマスクし、アプリケーション開発を簡略化することによって、強力な生産性の向上を提供します。ADO.NET データ プロバイダーには、パフォーマンスを最適化するように設計された機能が備わっています。

Entity Framework のパフォーマンスの向上

Entity Framework は強力な生産性の向上を提供しますが、一部の開発者は、アプリケーションでのパフォーマンスの最適化を必要とする機能について、Entity Framework が制御をしすぎると考えています。

XML スキーマ ファイルのサイズの制限

エンティティ データ モデル（EDM）を使って大きなモデルを構築すると、非常に効率が悪くなる可能性があります。最適な結果を得るために、モデルのエンティティが 50 から 100 に達している場合は、モデルを分割することを考慮してください。

XML スキーマ ファイルのサイズは、モデルを生成する基となったデータベースのテーブル、ビュー、またはストアド プロシージャの数にある程度比例します。スキーマ ファイルのサイズが大きくなるにつれ、メタデータの In-Memory モデルの作成や解析にかかる時間が増えます。これは、ObjectContext インスタンスごとに負担する、1 回限りのパフォーマンス コストです。

このメタデータは EntityConnection 文字列に基づいて、アプリケーション ドメインごとにキャッシュされます。つまり、1 つのアプリケーション ドメインの複数の ObjectContext インスタンスで同じ EntityConnection 文字列を使用している場合、アプリケーションがメタデータの読み込みに関してコストを負担するのは 1 回だけということです。しかし、モデルのサイズが大きくなる場合や、アプリケーションが長期にわたるものでない場合には、パフォーマンス コストが重要になる可能性があります。

ADO.NET Entity Framework でのストアド プロシージャの使用

ADO.NET Entity Framework でストアド プロシージャを使用するには、関数のマッピングが必要です。このようなストアド プロシージャの呼び出しは複雑で、いくらかのコーディングを必要とします。

機能の提供

Connection オブジェクトには、拡張された統計情報機能を提供するためのプロパティおよびメソッドが含まれています。これらは、ADO.NET データ プロバイダーでは標準ですが、ADO.NET Entity Framework レイヤーでは利用できません。代わりに、「擬似」ストアド プロシージャを介して同様の機能を公開します。

この方法では、エンティティ データ モデル（EDM）を使用して、ADO.NET の結果に対応する結果を得ます。これは実質的に、擬似ストアド プロシージャから戻されるエンティティおよび関数を提供します。

擬似ストアド プロシージャへのマッピング

次の表は、データ プロバイダーの Connection プロパティから対応する擬似ストアド プロシージャへのマッピングを示します。

Connection プロパティ	擬似ストアド プロシージャ
StatisticsEnabled	Psql_Connection_EnableStatistics Psql_Connection_DisableStatistics
Connection メソッド	**擬似ストアド プロシージャ**
ResetStatistics	Psql_Connection_ResetStatistics
RetrieveStatistics	Psql_Connection_RetrieveStatistics

次の C# コードで示されているように、アプリケーションは ObjectContext を使用してストアド プロシージャ コマンドを作成する必要があります。

```
using (MyContext context = new MyContext())
{
    EntityConnection entityConnection = (EntityConnection)context.Connection;

    // EntityConnection は基となるストア接続を公開します
    DbConnection storeConnection = entityConnection.StoreConnection;
```

```
    DbCommand command = storeConnection.CreateCommand();
    command.CommandText = "Psql_Connection_EnableStatistics";
    command.CommandType = CommandType.StoredProcedure;
    command.Parameters.Add(new PsqlParameter("cid", 1));
}

//

bool openingConnection = command.Connection.State == ConnectionState.Closed;
if (openingConnection) { command.Connection.Open(); }
int result;
try
{
    result = command.ExecuteNonQuery();
}
finally
{
    if (openingConnection && command.Connection.State == ConnectionState.Open) {
command.Connection.Close(); }
}
```

オーバーロードされたストアド プロシージャの使用

オーバーロードされたストアド プロシージャが複数ある場合、Zen Entity Framework データ プロバイダーは各ストアド プロシージャに識別子を追加して、SSDL でそれらを識別できるようにします。データ プロバイダーは、アプリケーションでストアド プロシージャを呼び出す前に、追加した識別子を削除します。

.NET オブジェクトの使用

ADO.NET Entity Framework データ プロバイダーは .NET パブリック オブジェクトをサポートし、それらをシールド オブジェクト（封印されたオブジェクト）として公開します。

詳細については、サポートされる .NET オブジェクトを参照してください。

ADO.NET Entity Framework のプログラミング コンテキストでは、本質的に一部の ADO.NET メソッドおよびプロパティを使用する必要がなくなります。しかし、これらのプロパティおよびメソッドは、標準の ADO.NET アプリケーションでは有用なままです。Visual Studio に組み込まれているオンライン ヘルプで、各クラスのパブリック メソッドおよびパブリック プロパティについて説明されています。

ADO.NET Entity データ プロバイダーとの、プロパティおよびメソッドの相違点

次の表は、ADO.NET Entity Framework アプリケーションでデータ プロバイダーを使用する場合には不要となる、あるいは実装が異なるプロパティおよびメソッドを示しています。

プロパティまたはメソッド	動作
PsqlCommand	
AddRowID	サポートされません。ADO.NET Entity Framework は、返される追加データを処理しません。
ArrayBindCount	サポートされません。アプリケーションは、ADO.NET Entity Framework 上でこのバインド数に影響を与えることができません。
ArrayBindStatus	サポートされません。アプリケーションは、ADO.NET Entity Framework 上でこのバインド数に影響を与えることができません。
BindByName	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。
CommandTImeout	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。
UpdatedRowSource	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。

プロパティまたはメソッド	動作
PsqlCommandBuilder	
DeriveParameters	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。
PsqlConnection	
ConnectionTimeout	接続文字列でのみサポートされます。
StatisticsEnabled	StatisticsEnabled または StatisticsDisabled ストアド プロシージャを使用します。ADO.NET Entity Framework アプリケーションでこの関数を使用する方法については、ADO.NET Entity Framework でのストアド プロシージャの使用を参照してください。
DataAdapter	
UpdateBatchSize	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。
Error	
ErrorPosition	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。
SQLState	サポートされません。代わりに、データ プロバイダーは ADO.NET Entity Framework プログラミング コンテキストを使用します。

モデルの作成

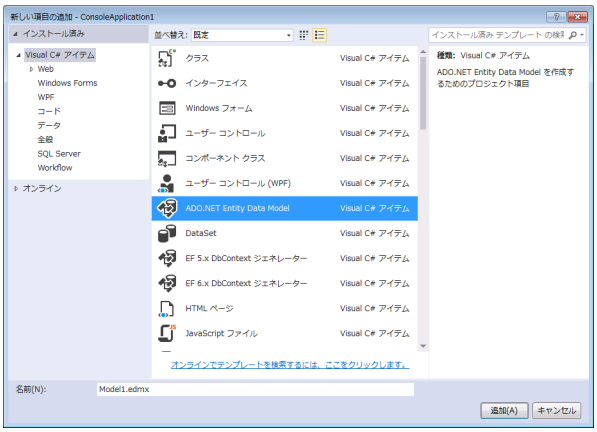
Entity Framework は、Visual Studio でデータのモデルを作成します。

メモ： ADO.NET Entity Framework データ プロバイダーを使って開発するには、バージョン 4.6 の Actian Zen ADO.NET Entity Framework データ プロバイダーと一緒に、Microsoft .NET Framework バージョン 4.5.x、4.6.x、4.7.x、または 4.8 と、Visual Studio 2019 以降を使用する必要があります。

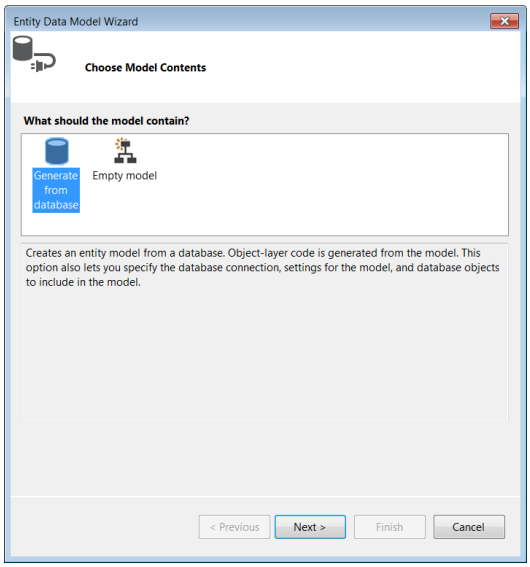
Visual Studio で Entity Framework を使用してデータのモデルを作成するには、まず、データベース スキーマが利用可能になっていることを確認してください。

Visual Studio で Entity Framework を使用してデータのモデルを作成するには

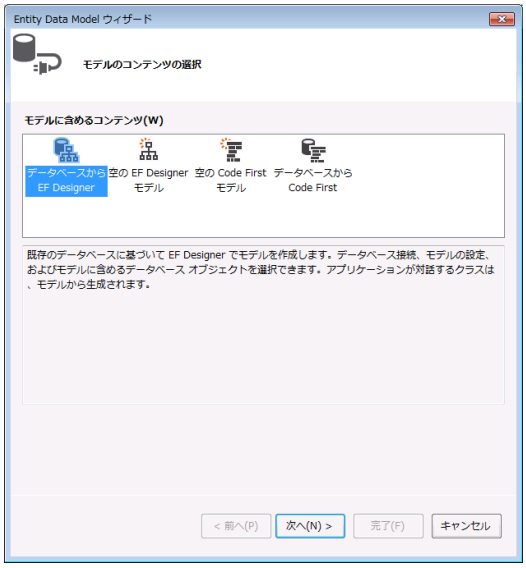
1. Visual Studio で、Windows コンソール、Windows フォームなどの新しい .NET アプリケーションを作成します。
2. ソリューション エクスプローラーでプロジェクトを右クリックし、［**追加**］>［**新しい項目**］をクリックします。
3. ［**ADO.NET Entity Data Model**］を選択したら、［**追加**］をクリックします。



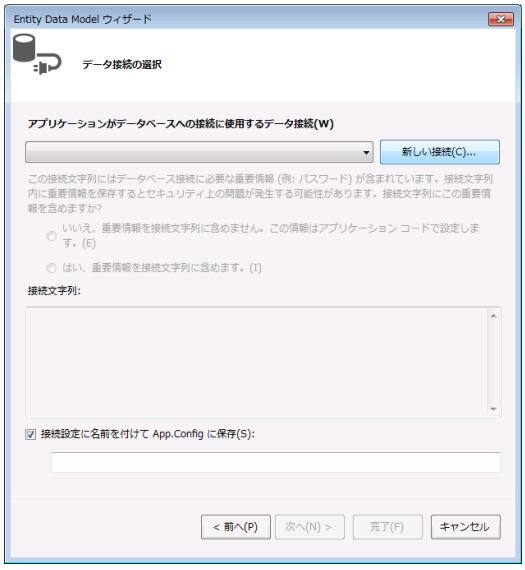
4. Entity Data Model ウィザードが表示されます。Microsoft ADO.NET Entity Framework 6.1（EF 6.1）を構成しているかどうかに基づいて、次のいずれかの操作を行います。
   - EF 6.1 を構成して**いない**場合は、［**データベースから生成**］を選択して［**次へ**］をクリックします。



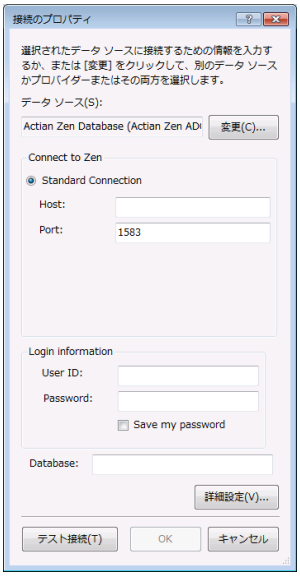
- EF 6.1 を構成して**いる**場合は、［**データベースから EF Designer**］を選択して［**次へ**］をクリックします。



5. ［データ接続の選択］ページで［**新しい接続**］をクリックして、新しい接続を作成します。既存の接続がある場合は、ドロップダウン リストからその接続を選択することができます。



6. ［接続のプロパティ］ウィンドウが表示されます。必要な接続情報を指定して、［**OK**］をクリックします。



7. ウィザードが Entity 接続文字列を作成します。

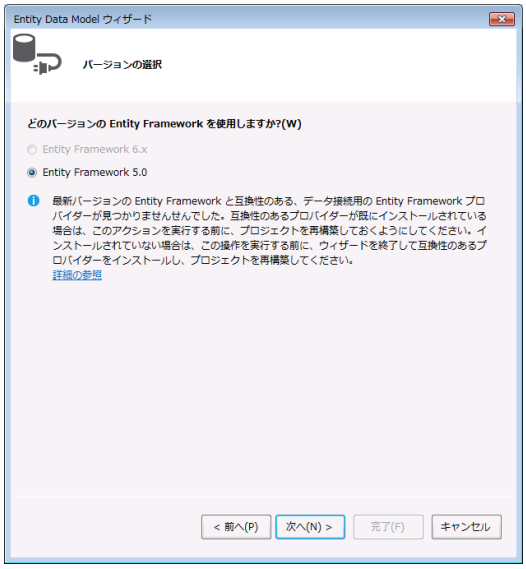
   a. ラジオ ボタンが選択可能であれば、［**はい、重要情報を接続文字列に含めます**］を選択して、接続文字列に重要情報を含めるようにします。

   b. ［**エンティティ接続設定に名前を付けて ... に保存**］フィールドで、メインのデータ アクセス クラスの名前を入力するか、またはデフォルトの名前を受け入れます。

   c. ［**次へ**］をクリックします。

8. 構成されている Entity Framework のバージョンに基づいて、次のいずれかの操作を行います。

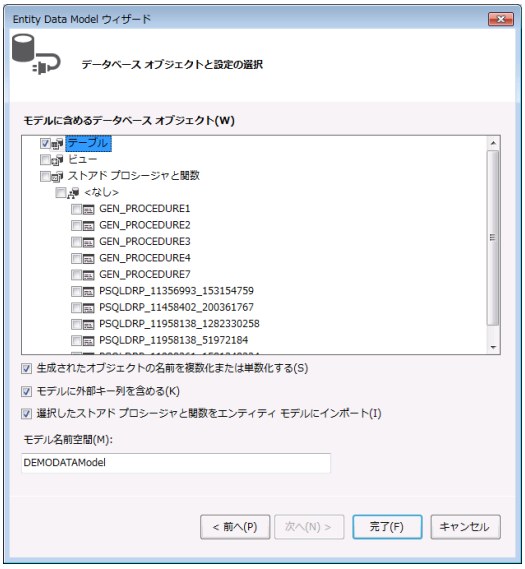
- 現在のプロジェクト用に EF5 を構成している場合は、［バージョンの選択］ページで［**次へ**］をクリックし、デフォルトの［**Entity Framework 5.0**］で次へ進めます。



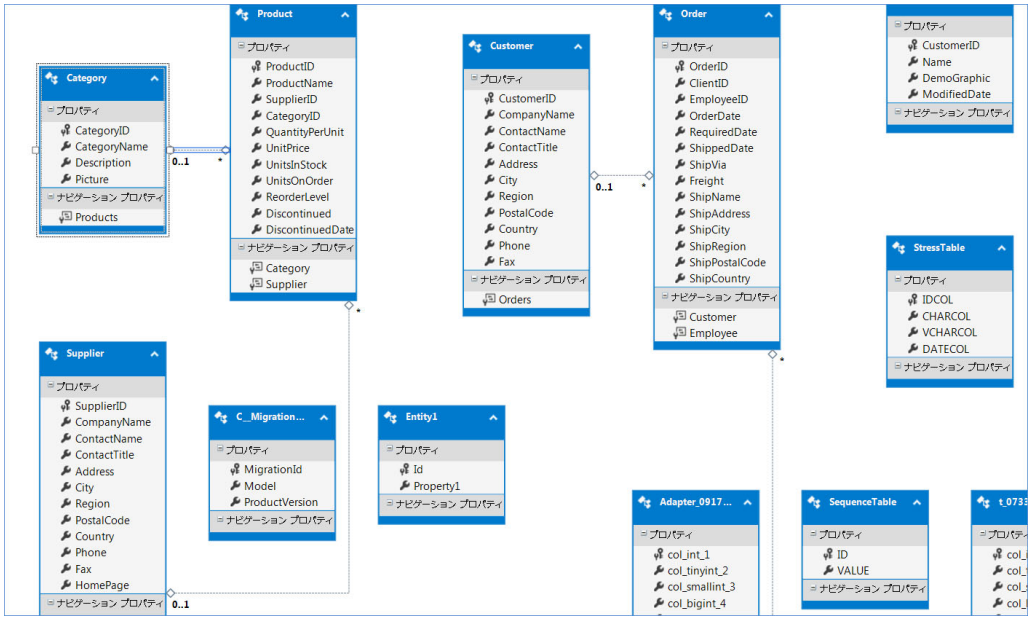
メモ：現在のプロジェクトで EF 6.1 を使用するには、ウィザードを終了し、EF 6.1 を構成した後、プロジェクトをリビルドします。EF 6.1 の構成後にプロジェクトをリビルドすると、ウィザードでは［バージョンの選択］ページが表示されず、次の手順へ直接進めるようになります。

- 現在のプロジェクト用に EF 6.1 を構成している場合は、次の手順に進みます。

9. モデルで使用するデータベース オブジェクトを選択します。



10. [**完了**] をクリックします。モデルが生成され、モデル ブラウザーで開かれます。



Entity Framework 5 アプリケーションから Entity Framework 6.1 へのアップグレード

下記の手順では、Zen ADO.NET Entity Framework プロバイダーを使用して作成された EF 5 アプリケーションを EF 6.1 アプリケーションにアップグレードします。

メモ： Entity Framework 5 Code First アプリケーションで作成されたオブジェクトがターゲット データベースに既に含まれている場合は、それらのオブジェクトを削除してから、移行された Entity Framework 6.1 アプリケーションを実行する必要があります。Entity Framework 5.0 は Entity Framework 6.1 とは異なる外部キー制約名を生成するため、それにより、アプリケーションは " テーブルまたはビューは既に存在します " というエラーで失敗します。

重要！ ここで示されている手順を指定された順序で実行してください。

machine.config ファイルを更新するには

この手順は次のような理由から必要となります。

Entity Framework Power Tools を使用して、EF 6.1 でマッピング ビューを再生成することができます。Entity Framework Power Tools を Zen ADO.NET Entity Framework プロバイダーと連動させるには、.NET Framework 4.0 の machine.config ファイルに Provider の登録エントリを追加する必要があります。

メモ： この手順は、EF 5 アプリケーションを EF 6.1 にアップグレードする場合にのみ必要となります。アップグレード後、machine.config ファイルの編集を元に戻して、以前の内容に戻すことをお勧めします。

1. 以下の手順を続行する前に、すべての Visual Studio ウィンドウを閉じてください。
2. 次の場所にある .NET Framework 4 用の machine.config ファイルを開きます。64 ビット システムを使用している場合は、2 番目の場所にあるファイルも開きます。
   - %windir%¥Microsoft.NET¥Framework¥v4.0.30319¥Config¥machine.config
   - %windir%¥Microsoft.NET¥Framework64¥v4.0.30319¥Config¥machine.config
3. <configSections></configSections> ノード下に次のエントリを追加します。

```
<section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
```

4. <configuration> </configuration> ノード下の </configSections> 閉じるタグより後に、次のエントリを追加します。

```
<entityFramework>
<providers>
<provider invariantName="Pervasive.Data.SqlClient"
type="Pervasive.Data.SqlClient.Entity.PsqlProviderServices, Pervasive.Data.SqlClient.Entity,
Version=4.6.0.0, Culture=neutral, PublicKeyToken=c84cd5c63851e072" />
</providers>
</entityFramework>
```

5. 変更したファイルを保存して閉じます。
6. 対応するプロジェクトに EF バージョン 6.1.2 をインストールする必要があります。これを行うには、プロジェクトを新しい EF 6.1.x ランタイムにアップグレードする必要があります。次の 2 つの方法のいずれかにより実行できます。

   方法 1

   a. プロジェクトを右クリックして、［**NuGet パッケージの管理**］をクリックします。

   b. ［オンライン］タブで、**EntityFramework** を選択して［**インストール**］をクリックします。

   以前のバージョンの EntityFramework NuGet パッケージが EF 6.1.x にアップグレードされます。

   方法 2

   EF 6.1.2 をインストールする対応プロジェクトに対し、パッケージ マネージャー コンソールから次のコマンドを実行します。

```
Install-Package EntityFramework -Version 6.1.2
```

7. System.Data.Entity.dll へのアセンブリ参照が削除されていることを確認します。

   EF6 NuGet パッケージをインストールすると、System.Data.Entity へのすべての参照がプロジェクトから自動的に削除されます。

8. EF 6.x のコード生成を使用するように Enterprise Framework Designer（EDMX）モデルを変更します。EF Designer で作成されたモデルがある場合は、コード生成テンプレートを更新して、EF6 互換コードを生成する必要があります。

メモ： Visual Studio 2012 以降の場合は、EF 6.x DbContext Generator テンプレートのみ使用できます。

a. 既存のコード生成テンプレートを削除します。
これらのファイルには通常、<edmx ファイル名 >.tt や <edmx ファイル名 >.Context.tt という名前が付けられており、ソリューション エクスプローラーで .edmx ファイルの下に入れ子になっています。ソリューション エクスプローラーでテンプレートを選択し、Delete キーを使用してそれらを削除することができます。

メモ： Web サイト プロジェクトでは、ソリューション エクスプローラー内のテンプレートは .edmx ファイルの下に入れ子になっているのではなく、並んで存在しています。VB.NET プロジェクトでは、［すべてのファイルを表示］を有効にして、入れ子になったテンプレート ファイルを表示できるようにする必要があります。

b. 適切な EF 6.x コード生成テンプレートを追加します。
EF Designer でモデルを開き、デザイン画面を右クリックして［コード生成項目の追加］を選択します。

- DbContext API（推奨）を使用している場合は、［データ］タブで EF 6.x DbContext Generator を使用できます。
Visual Studio 2012 を使用している場合は、このテンプレートを使用するために EF 6 Tools をインストールする必要があることに注意してください。詳細については、Microsoft の「Entity Framework を取得する」（https://msdn.microsoft.com/ja-jp/library/ee712906(v=vs.113).aspx）を参照してください。
- ObjectContext API を使用している場合は、［オンライン］タブを選択し、EF 6.x EntityObject Generator を検索する必要があります。

c. コード生成テンプレートにカスタマイズを適用した場合は、更新されたテンプレートに再適用する必要があります。

9. 使用しているすべてのコア EF 型の名前空間を更新します。

DbContext 型および Code First 型は変更されていません。これは、EF 4.1 以降を使用する多くのアプリケーションでは、何も変更する必要がないことを意味します。

以前は System.Data.Entity.dll にあった ObjectContext などの型は、新しい名前空間に移動されました。つまり、EF6 に対してビルドするには、using ディレクティブまたは import ディレクティブを更新する必要があります。

名前空間の変更に関する一般的な規則は、「System.Data.* 内のすべての型が System.Data.Entity.Core.*. に移動される」ということです。言い換えると、System.Data の後に Entity.Core を挿入するだけです。たとえば、次のようになります。

- System.Data.EntityException => System.Data.Entity.Core.EntityException
- System.Data.Objects.ObjectContext => System.Data.Entity.Core.Objects.ObjectContext
- System.Data.Objects.DataClasses.RelationshipManager => System.Data.Entity.Core.Objects.DataClasses.RelationshipManager

これらの型は、ほとんどの DbContext ベースのアプリケーションでは直接使用されないため、Core 名前空間にあります。System.Data.Entity.dll の一部であった一部の型は、DbContext ベースのアプリケーションでよく使用され、直接使用されるため、Core 名前空間に移動されていません。次のものが該当します。

- System.Data.EntityState => System.Data.Entity.EntityState
- System.Data.Objects.DataClasses.EdmFunctionAttribute => System.Data.Entity.DbFunctionAttribute
  メモ：このクラスは名前が変更されました。古い名前のクラスは引き続き存在し、動作しますが、現在は不使用とマークされています。
- System.Data.Objects.EntityFunctions => System.Data.Entity.DbFunctions
  メモ：このクラスは名前が変更されました。古い名前のクラスは引き続き存在し、動作しますが、現在は不使用とマークされています。

10. マッピング ビューを再生成します。

    以前にマッピング ビューを生成した場合は、そのファイルを削除して、マッピング ビューを再生成します。マッピング ビューの詳細については、https://msdn.microsoft.com/en-us/data/dn469601 を参照してください。

これで、Zen ADO.NET データ プロバイダーを使用してビルドされた EF 5 アプリケーションが EF 6.1.2 にアップグレードされました。

メモ： すべての EF 5 アプリケーションが EF 6.1.2 にアップグレードされたら、machine.config ファイルに加えた変更を元に戻し、以前の machine.config ファイルを復元することをお勧めします。

Entity Framework Power Tools は、ビュー モデルを使用する EF 5.0 アプリケーションを EF 6.1 アプリケーションに移行するためのより簡単な方法を提供します。テスト中、Entity Framework Power Tools が Zen Entity Framework プロバイダーで正常に機能することがわかりました。

メモ： Entity Framework Power Tools は、Zen ADO.NET Entity Framework データ プロバイダーでは認定済みのサポートされているツールではありません。

詳細情報

ADO.NET および Entity Framework の追加情報については、以下を参照してください。

- *Programming Entity Framework* by Julie Lerman は、ADO.NET Entity Framework の使い方の網羅的な議論を提供しています。
- ADO.NET Entity Framework では、Entity Framework を紹介するほか、多くの詳細にわたる項目へのリンクを提供しています。
- 接続文字列（Entity Framework）では、Entity Framework による接続文字列の使い方について説明します。接続文字列には、基となる ADO.NET データ プロバイダーへの接続に使用する情報が含まれているほか、必要なエンティティ データ モデルのマッピングおよびメタデータに関する情報も含まれています。
- Entity Data Model ツールでは、EDM を使用してアプリケーションを視覚的に構築するためのツールについて説明します。個別のツールとして、Entity Data Model ウィザード、ADO.NET Entity Data Model デザイナー（エンティティ デザイナー）、およびモデルの更新ウィザードがあります。これらのツールは共同して、エンティティ データ モデルを生成、編集、および更新することができます。
- LINQ to Entities を使用すると、開発者はビジネス ロジックの構築に使用された言語と同じ言語のデータベースに対するクエリを作成できるようになります。

Zen ADO.NET Entity Framework Core データ プロバイダー

Zen ADO.NET Entity Framework (EF) Core は、クロス プラットフォーム開発をサポートする .NET 用の軽量で拡張可能なオブジェクト リレーショナル マッパー（O/RM）です。これにより、開発者は .NET オブジェクトを使用してデータベースを操作できるようになり、データ中心のアプリケーションで記述および保守する必要のあるコードの量を減らすことができます。

Zen ADO.NET Entity Framework Core データ プロバイダーは、ADO.NET Entity Framework Core を使用するアプリケーションで使用できます。以下をサポートしています。

- Microsoft ADO.NET Entity Framework Core 8.0
- .NET Standard 8.0 でサポートされるすべてのプラットフォーム。詳細については、https://docs.microsoft.com/ja-jp/dotnet/standard/net-standard を参照してください。
- Microsoft Entity Framework Core および Actian Zen データベースの両方でサポートされるすべての機能。詳細については、https://docs.microsoft.com/ja-jp/ef/core/what-is-new/ を参照してください。

Zen ADO.NET Entity Framework Core データ プロバイダーは、ADO.NET データ プロバイダーを使用して ADO.NET データベース サーバーと通信します。つまり、Zen ADO.NET データ プロバイダーによって定義された機能は、ここで特に断りがない限り、Zen ADO.NET Entity Framework Core データ プロバイダーに適用されるということです。同様に、Zen ADO.NET データ プロバイダーに合わせて作られたすべてのパフォーマンス構成が、Zen ADO.NET Entity Framework Core データ プロバイダーで実現されます。

メモ：Zen ADO.NET Entity Framework Core のアプリケーションを開発する場合は、Visual Studio 2022 以降が必要です。

Zen ADO.NET Entity Framework Core データ プロバイダーについて

Zen ADO.NET Entity Framework Core データ プロバイダーはマネージ コードのみで構築されています。つまり、完全に共通言語ランタイム（CLR）の内部で、実行およびデータベースへの接続が行えます。

クライアント ライブラリや COM コンポーネントなどネイティブ オペレーティング システムで実行するコードはアンマネージ コードと言います。マネージ コードとアンマネージ コードは 1 つのアプリケーション内に混在させることができます。ただし、アンマネージ コードは共通言語ランタイムの外部にまで影響が及ぶため、現実的には複雑になり、パフォーマンスも低下します。また、セキュリティの危険にさらすことにもなりかねません。

名前空間

Zen ADO.NET Entity Framework Core データ プロバイダーの名前空間は、Actian.EntityFrameworkCore.Zen です。

アセンブリ名

Zen ADO.NET Entity Framework Core データ プロバイダーは、Actian.EntityFrameworkCore.Zen.dll という名前のアセンブリを使用します。

これを使用するには、SDK ダウンロードで入手できる Zen ADO.NET データ プロバイダーに示されているお使いのバージョンの最新 SDK の .zip アーカイブをダウンロードして、NuGet パッケージ Actian.EntityFrameworkCore.Zen を展開します。その後、プロジェクトにパッケージを追加します。

Zen ADO.NET Entity Framework Core データ プロバイダーの構成

ADO.NET Entity Framework Core データ プロバイダーを構成するには

1. .NET Standard 8.0 を対象とするアプリケーションを作成します。詳細については、https://learn.microsoft.com/ja-jp/dotnet/core/tutorials/with-visual-studio?pivots=dotnet-8-0 を参照してください
2. NuGet パッケージ **Actian.EntityFrameworkCore.Zen** をインストールします。
3. 新しいコンテキスト クラスをアプリケーションに追加し、次のコードを使用して OnConfiguring メソッドをオーバーライドします。

```
public class MyContext : DbContext
 {
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
=> optionsBuilder.UseZen(connection string);
 }
```

Zen ADO.NET Entity Framework Core データ プロバイダーでの接続文字列の使用

Zen ADO.NET Entity Framework Core データ プロバイダーは接続文字列に含まれている情報を使用して、Entity Framework Core をサポートする基となる Zen ADO.NET データ プロバイダーに接続します。

接続文字列オプションのデフォルト値の変更

Zen ADO.NET Entity Framework Core データ プロバイダーで使用される接続文字列オプションのデフォルト値の多くは、Zen ADO.NET データ プロバイダーで使用されるデフォルト値と同じです。詳細については、接続文字列プロパティを参照してください。次の接続文字列オプションの一覧は、ADO.NET Entity Framework Core アプリケーションで使用される場合には異なるデフォルト値になるオプションを示しています。

接続文字列オプション	ADO.NET Entity Framework Core アプリケーションでのデフォルト値
Parameter Mode	サポートされません。
Statement Cache Mode	ExplicitOnly 値のみサポートされます。
DB File Directory Path	Environment.SpecialFolder.CommonApplicationData。Environment.SpecialFolder 列挙型およびサポートされるフィールドの詳細については、https://docs.microsoft.com/ja-jp/dotnet/api/system.environment.specialfolder?view=net-8.0 を参照してください。

Code First のサポート

Entity Framework Core は Code First の機能をサポートしています。これらの機能のサポートを実装すると、長い識別子名の処理に必要な変更など、データ プロバイダーへの変更が必要になります。ただし、アプリケーションの変更は必要ありません。

Code First の実装は型マッピングの変更を必要とします。詳細については、Code First の型マッピングを参照してください。

長い識別子名の処理

ほとんどの Zen 識別子の最大長は 20 バイトです。サーバー上で作成するオブジェクトの名前はクラス名とプロパティ名から取られるため、識別子名がこのサイズを超える場合があります。また、制約名は多くの場合、いくつかのオブジェクト名を連結して作成されます。このような場合には、識別子の最大長を超える可能性がさらに大きくなります。

列の場合、データ プロバイダーは識別子の末尾をチルダ文字 ~ に置き換えることにより、識別子をデータベースで許容される識別子の最大長に短縮します。たとえば、文字列 `ColumnMoreThanTwentyCharacters` は `ColumnMoreThanTwent~` に短縮されます。

テーブルの場合、データ プロバイダーは識別子の末尾を整数のハッシュ コードに置き換えることにより、識別子をデータベースで許容される識別子の最大長に短縮します。たとえば、文字列 `ATableWithAVeryLongTableName` は `ATableWit_1738385675` に短縮されます。

DB ツールを使用して DB オブジェクトのアクセスや表示を行った場合、作成されたテーブルの名前が、モデル クラス名とプロパティ名から予想される名前とは異なる場合があります。

ADO.NET Entity Framework Core での Code First Migrations の使用

Entity Framework Core では Code First Migrations をサポートしています。これにより、データベースを削除して再作成しなくても、データベース スキーマを更新することでモデル クラスを反映できるようになります。

移行は、モデルの変更に従って、データベース スキーマを増分で更新できるようにします。データベースに対する各変更セットは、移行として知られるコード ファイルで表されます。移行は、通常、タイムスタンプを使用して管理され、データベース内のテーブルは、どの移行がデータベースに適用されたかを追跡します。

Code First Migrations は型マッピングの変更を必要とします。詳細については、Code First の型マッピングを参照してください。

Code First Migrations を実装するには、Zen ADO.NET Entity Framework Core データ プロバイダーを構成した時点で、NuGet パッケージ **Microsoft.EntityFrameworkCore.Tools（バージョン 3.1）**をインストールします。

リバース エンジニアリングの使用（スキャフォールディング）

データベース スキーマに基づいてエンティティ型クラスおよび DbContext クラスをスキャフォールディングする処理は、リバース エンジニアリングと呼ばれます。これは、EF Core パッケージ マネージャー コンソール（PMC）ツールの `Scaffold-DbContext` コマンド、または .NET コマンド ライン インターフェイス（CLI）ツールの `dotnet ef dbcontext scaffold` コマンドを使用して実行できます。

ADO.NET Actian Zen Entity Framework Core データ プロバイダーでリバース エンジニアリングを使用するには、データ プロバイダーを構成した後、次の手順を実行します。

1. NuGet パッケージ **Microsoft.EntityFrameworkCore.Tools（バージョン 8.0.0）**をインストールします。
2. 次の `Scaffold-DbContext` PowerShell コマンドを実行します。

   ```
   Scaffold-DbContext 'connection string' Actian.EntityFrameworkCore.Zen
   ```

必要に応じて、`Scaffold-DbContext` PowerShell コマンドにパラメーターを追加することができます。詳細については、https://docs.microsoft.com/ja-jp/ef/core/managing-schemas/scaffolding?tabs=vs を参照してください。

.NET Core CLI 環境でリバース エンジニアリングを使用する場合は、https://docs.microsoft.com/ja-jp/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli を参照してください。

Code First の型マッピング

次の表はモデル中心の型マッピングを示しています。このマッピングでは、CLR 型は、データベースの作成時に使用される Zen（ストア）型に影響を与えます。一部の CLR 型はいくつかの異なる Zen 型にマップすることができます。

列は次のように定義されています。

- ［CLR 型］列は、共通言語ランタイムの型名を示しています。
- ［型マッピングに影響を与えるプロパティ値］列は、型マッピングに影響を及ぼす可能性のある、あらゆるプロパティ値を示しています。
- ［Zen 型］列は、ネイティブな型名を示しています。

CLR 型	型マッピングに影響を与えるプロパティ値	Actian Zen データ型
Boolean		BIT
Byte		UTINYINT
Byte[ ]		BINARY LONGVARBINARY[1]
DateTime		AUTOTIMESTAMP DATE TIMESTAMP TIMESTAMP2[2]
DateTimeOffset		DATETIME
Decimal		CURRENCY DECIMAL[2] NUMERIC NUMERICSA NUMERICSTS UBIGINT
Double		BFLOAT8 DOUBLE[2]

CLR 型	型マッピングに影響を与えるプロパティ値	Actian Zen データ型
Float		FLOAT[2] REAL BFLOAT4
Guid		UNIQUEIDENTIFIER
Integer		INTEGER[2] IDENTITY USMALLINT
Long		BIGINT[2] BIGIDENTITY UINTEGER
SByte		TINYINT
Short		SMALLINT[2] SMALLIDENTITY
String		NLONGVARCHAR[2]
	Unicode=False	LONGVARCHAR
	MaxLength= (1<=n<=8000) Fixed Length=False Unicode=True	NVARCHAR
	MaxLength= (1<=n<=8000) Fixed Length=False Unicode=False	VARCHAR
	MaxLength= (1<=n<=8000) Fixed Length=True Unicode=True	NCHAR
	MaxLength= (1<=n<=8000) Fixed Length=True Unicode=False	CHAR
TimeSpan		TIME
1. MaxLength に値が指定されていない場合は、この型にマップされます。 2. デフォルトでは、この型にマップされます。		

EDM 正規関数から Zen 関数へのマッピング

ADO.NET Entity Framework Core は、エンティティ データ モデル（EDM）正規関数を Zen ADO.NET Entity Framework Core データ プロバイダーの対応するデータ ソース機能に変換します。これによって、全データ ソースに共通する形式で表現される関数を呼び出すことができます。

これらの正規関数はデータ ソースから独立しているため、正規関数の引数の型と戻り値の型は、EDM の型の語句で定義されます。Entity SQL クエリで正規関数を使用すると、データ ソースで適切な関数が呼び出されます。

すべての正規関数には、ヌルが入力された場合の動作とエラー状況が明示的に指定されています。ただし、ADO.NET Entity Framework Core はこの動作を実行しません。

集計正規関数

次の表は、EDM 集計正規関数から Zen 関数へのマッピング、およびこれらの関数が適用される CLR 型を示します。

集計正規関数	Actian Zen 関数	CLR 型
BigCount(*expression*)	COUNT_BIG(*expression*)	Long
Count(*expression*)	COUNT(*expression*)	Integer

数学正規関数

次の表は、EDM 数学正規関数から Zen 関数へのマッピング、およびこれらの関数が適用される CLR 型を示します。

数学正規関数	Actian Zen 関数	CLR 型
Abs(*expression*)	ABS(*expression*)	Decimal、Double、Float、Int、Long、SByte、Short
Ceiling(*expression*)	CEILING(*expression*)	Decimal、Double
Floor(*expression*)	FLOOR(*expression*)	Decimal、Double
Pow(*base*, *power*)	POWER(*base*, *power*)	Double
Exp(*expression*)	EXP(*expression*)	Double
Log10(*expression*)	LOG10(*expression*)	Double

数学正規関数	Actian Zen 関数	CLR 型
Log(*expression*)	LOG(*expression*)	Double
Sqrt(*expression*)	SQRT(*expression*)	Double
Acos(*expression*)	ACOS(*expression*)	Double
Asin(*expression*)	ASIN(*expression*)	Double
Atan(*expression*)	ATAN(*expression*)	Double
Atan2(*expression1*, *expression2*)	ATAN2(*expression1*, *expression2*)	Double
Cos(*expression*)	COS(*expression*)	Double
Sin(*expression*)	SIN(*expression*)	Double
Tan(*expression*)	TAN(*expression*)	Double
Sign(*expression*)	SIGN(*expression*)	Decimal、Double、Float、Int、Long、SByte、Short

日付と時刻の正規関数

次の表は、EDM の日付と時刻の正規関数から Zen 関数へのマッピングを示します。これらの関数は、日付時刻データを使用する型の生成、処理、および操作を行います。また、この表はこれらの関数が適用される CLR 型も示しています。

日付と時刻の正規関数	Actian Zen 関数	CLR 型
DateTime.Now	SYSDATETIME()	DateTime
DateTime.Now	SYSDATETIMEOFFSET()	DateTimeOffset
DateTime.UtcNow	SYSUTCDATETIME()	DateTime DateTimeOffset
DateTime.Today	CURDATE()	DateTime DateTimeOffset
AddYears(*expression*)	DATEADD(*year*, *expression*, *column*)	DateTime DateTimeOffset
AddMonths(*expression*)	DATEADD(*month*, *expression*, *column*)	DateTime DateTimeOffset

日付と時刻の正規関数	Actian Zen 関数	CLR 型
AddDays(*expression*)	DATEADD(*day*, *expression*, *column*)	DateTime DateTimeOffset
AddHours(*expression*)	DATEADD*(hour, expression, column)*	DateTime DateTimeOffset
AddMinutes(*expression*)	DATEADD(*minute*, *expression*, *column*)	DateTime DateTimeOffset
AddSeconds(*expression*)	DATEADD(*second*, *expression*, *column*)	DateTime DateTimeOffset
AddMilliSeconds(*expression*)	DATEADD(*millisecond*, *expression*, *column*)	DateTime DateTimeOffset
EF.Functions.DateDiffYear(*column*, *expression*)	DATEDIFF(*year*, *column*, *expression*)	DateTime DateTimeOffset
EF.Functions.DateDiffMonth(*column*, *expression*)	DATEDIFF(*month*, *column*, *expression*)	DateTime DateTimeOffset
EF.Functions.DateDiffDay(*column*, *expression*)	DATEDIFF(*day*, *column*, *expression*)	DateTime DateTimeOffset
EF.Functions.DateDiffHour(*column*, *expression*)	DATEDIFF(*hour*, *column*, *expression*)	DateTime DateTimeOffset TimeSpan
EF.Functions.DateDiffMinute(*column*, *expression*)	DATEDIFF(*minute*, *column*, *expression*)	DateTime DateTimeOffset TimeSpan
EF.Functions.DateDiffSecond(*column*, *expression*)	DATEDIFF(*second*, *column*, *expression*)	DateTime DateTimeOffset TimeSpan
EF.Functions.DateDiffMilliSecond(*column*, *expression*)	DATEDIFF(*millisecond*, *column*, *expression*)	DateTime DateTimeOffset TimeSpan

文字列正規関数

次の表は、EDM 文字列正規関数から Zen 関数へのマッピング、およびこれらの関数が適用される CLR 型を示します。

文字列正規関数	Actian Zen 関数	CLR 型
IndexOf(*expression*)	POSITION(*expression*, *column*)	String
Replace(*toReplace*, *replaceWith*)	REPLACE(*toReplace*, *column*, *replaceWith*)	String
ToLower()	LOWER(*column*)	String
ToUpper()	UPPER(*column*)	String
SubString(*start*, *length*)	SUBSTRING(*column*, *start*, *length*)	String
IsNullOrWhiteSpace()	列の LTRIM および RTRIM と、ヌルチェックの組み合わせ	String
TrimStart()	LTRIM(*column*)	String
TrimEnd()	RTRIM(*column*)	String
TRIM()	列の LTRIM と RTRIM の組み合わせ	String
Contains(*expression*)	POSITION(*expression*, *column*)	String
StartsWith(*expression*)	列の LEFT と LENGTH の組み合わせ	String
EndsWith(*expression*)	列の RIGHT と LENGTH の組み合わせ	String
Length()	LENGTH(*column*)	String
EF.Functions.Position(*column*, *expression*)	POSITION(*expression*, *column*)	String

その他の正規関数

次の表は、その他の正規関数から Zen 関数へのマッピング、およびこれらの関数が適用される CLR 型を示します。

正規関数	Actian Zen 関数	CLR 型
ToString()	CONVERT(*column*, SQL_CHAR)	すべての型
NewGuid()	NEWID()	Guid

メモ：*column* は関数が適用されるプロパティです。

Entity Framework 機能の拡張

ADO.NET Entity Framework Core および Actian Zen Entity Framework Core データ プロバイダーは、簡単に拡張できるように設計されています。以下の例は、Entity Framework Core を拡張する方法を示しています。

- https://docs.microsoft.com/ja-jp/ef/core/managing-schemas/migrations/history-table
- https://docs.microsoft.com/ja-jp/ef/core/modeling/dynamic-model

ADO.NET Entity Framework Core でのストアド プロシージャの使用

Entity Framework Core では、raw SQL クエリを使用してストアド プロシージャを拡張することができます。詳細については、https://docs.microsoft.com/ja-jp/ef/core/querying/raw-sql を参照してください。

Entity Framework 6.x から Entity Framework Core へのアプリケーションのアップグレード

アプリケーションを Entity Framework 6.x から Entity Framework Core にアップグレードするには、https://docs.microsoft.com/ja-jp/ef/efcore-and-ef6/porting/ を参照してください。

制限事項

Actian Zen ADO.NET Entity Framework Core データ プロバイダーには以下の制限事項があります。

- Entity Framework Core には、リバース エンジニアリング（スキャフォールディング）に関連するいくつかの制限があります。それらはすべて、Actian Zen Entity Framework Core データ プロバイダーにも同様に適用されます。これらの制限の詳細については、https://docs.microsoft.com/ja-jp/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli を参照してください。
- Actian Zen Entity Framework Core データ プロバイダーは、リバース エンジニアリング（スキャフォールディング）の同時実行機能をサポートしていません。

詳細情報

ADO.NET および Entity Framework Core の追加情報については、以下を参照してください。

- ADO.NET Entity Framework Core では、Entity Framework Core を紹介するほか、多くの詳細にわたる項目へのリンクを提供しています。
- 機能の比較では、Entity Framework Core と Entity Framework 6.X で使用できる機能を比較しています。
- ASP.NET Core は、ASP.NET Core Razor Pages アプリでの Entity Framework Core の使用方法を示しています。

Visual Studio での Zen データ プロバイダーの使用

Zen データ プロバイダーは Visual Studio への統合に対応しています。つまり、開発者は Microsoft Visual Studio のグラフィカル ユーザー インターフェイスを使用して、さまざまなタスクを実行できます。

以下のトピックでは、Zen データ プロバイダーの機能がどのようにして Visual Studio に組み込まれるかを説明します。

- 接続の追加
- Zen Performance Tuning Wizard の使用
- プロバイダー固有テンプレートの使用
- Zen Visual Studio Wizard の使用
- ツールボックスからのコンポーネントの追加
- データ プロバイダー統合のシナリオ

接続の追加

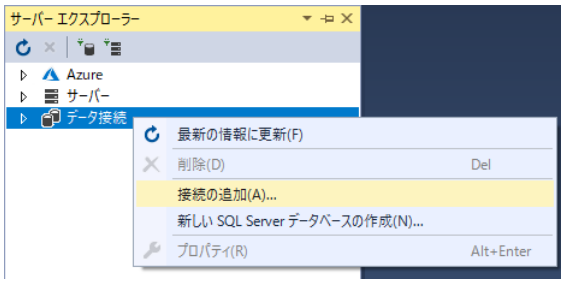
Visual Studio では、いくつかの方法で接続を追加することができます。

- サーバー エクスプローラーでの接続の追加
- データソース構成ウィザードによる接続の追加

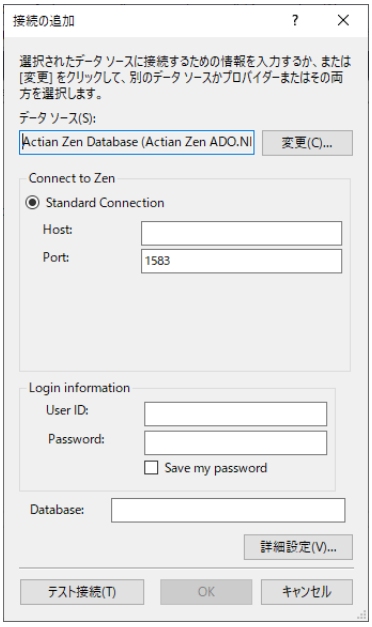
サーバー エクスプローラーでの接続の追加

接続を追加するには

1. サーバー エクスプローラーで［**データ接続**］を右クリックし、［**接続の追加**］を選択します。

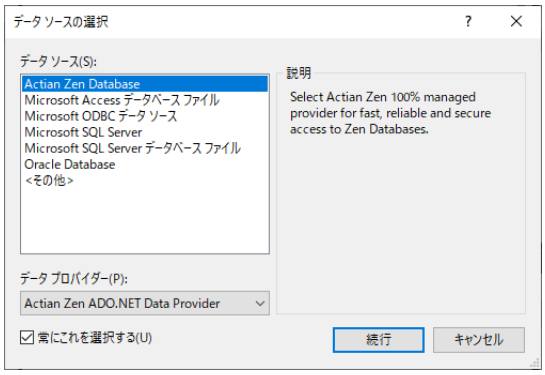


［接続の追加］ウィンドウが表示されます。



2. ［**データ ソース**］フィールドに Zen データ プロバイダーが表示されている場合は、手順 4 へ進みます。そうでない場合は、［**変更**］をクリックします。

3. ［データ ソースの変更］ウィンドウが表示されます。



   a. ［データ ソース］リスト ボックスで "**Actian Zen Database**" を選択します。

   b. ［データ プロバイダー］リストで "**Actian Zen ADO.NET Data Provider**" を選択します。

   c. ほかの接続でもこれらの選択を使用したい場合は、［**常にこれを選択する**］チェック ボックスをオンにします。

   d. ［**OK**］をクリックして［接続の追加］ウィンドウに戻ります。

4. ［接続の追加］ウィンドウで、次の手順を実行します。

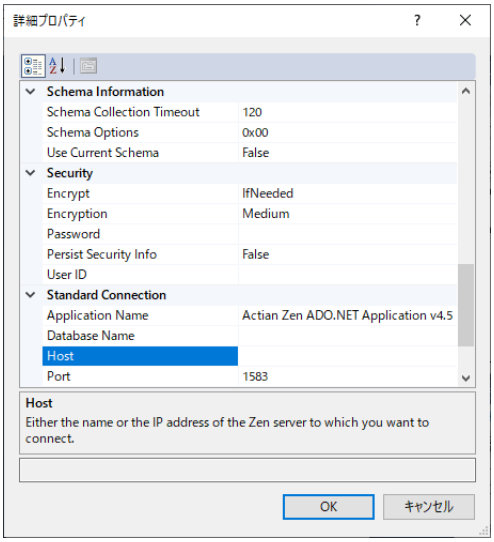
   a. ホスト名（Host）を入力します。

   b. ユーザー ID（User ID）とパスワード（Password）を入力します。これらの値は認証で必要となります。

   c. （任意）サーバー エクスプローラーで定義した接続インスタンスの存続期間中パスワードを保存したい場合は、［**Save my password**］チェック ボックスをオンにします。

   d. （省略可能）［データベース］入力フィールドで、接続するデータベースの名前を入力します。

5. さらにプロバイダー固有のプロパティ値を指定する場合は、［**詳細設定**］ボタンをクリックします。



［詳細プロパティ］ダイアログ ボックスの値を変更するには、対象フィールドで新しい値を選択するか、値を入力して Enter キーを押します。この値が接続文字列に追加され、プロパティの説明の下にあるフィールドに表示されます。デフォルトの値を受け入れる場合は、接続文字列フィールドを変更せずそのままにしておきます。必要な変更を加えたら、［**OK**］をクリックして［接続の追加］ウィンドウに戻ります。

Advanced（詳細）

EnableIPv6：IPv4 アドレスを使用した Zen サーバーへの接続に対応する下位互換性を提供します。

True に設定すると、IPv4 アドレスまたは IPv6 アドレスのいずれかを使用するサーバーに対し、インストール済みの IPv6 プロトコルに対応したクライアントを接続することができます。

False に設定すると、クライアントは下位互換性モードで実行します。クライアントは常に、IPv4 アドレスを使用するサーバーに接続されます。

デフォルト値は、4.0 では True に設定されます。

IPv6 形式の詳細については、『*Getting Started with Zen*』の IPv6 を参照してください。

Encoding：データベースに格納されている文字列データの変換に使用する、ANSI 名または Windows コード ページを入力します。デフォルトでは、Windows コード ページが使用されます。

Initial Command Timeout：データ プロバイダーが実行の試行を終了してエラーを生成するまでのデフォルトの待機時間（秒単位のタイムアウト）を指定します。このオプションは、アプリケーションのコードに変更を加えることなく、PsqlCommand オブジェクトの CommandTimeout プロパティと同じ機能を提供します。その後、アプリケーションは CommandTimeout プロパティを使用して Initial Command Timeout 接続文字列オプションを上書きすることができます。

デフォルトの初期値は 30 秒です。

メモ：CommandTimeout オプションの初期値は、サーバーのデッドロック検出およびタイムアウトの最大値より大きい値に設定します。これによって、アプリケーションはタイムアウトした場合により意味のある応答を受け取ることができます。

Initialization String：セッションの設定を管理するために、データベースへの接続後直ちに発行されるステートメントを入力します。

たとえば、NULL で埋められた CHAR 列を処理するには、次のように値を設定します。

```
Initialization String=SET ANSI_PADDING ON
```

メモ：何らかの理由でステートメントが失敗した場合、サーバーへの接続は失敗します。データ プロバイダーは、サーバーから返されたエラーを含む例外をスローします。

Parameter Mode：ネイティブ パラメーター マーカーおよびバインディングの動作を選択します。これにより、アプリケーションはプロバイダー固有の SQL コードを再利用でき、Zen データ プロバイダーへの移行を容易にすることができます。このオプションは Zen ADO.NET Entity Framework データ プロバイダーには適用されないので、注意してください。

ANSI（デフォルト）に設定すると、? 文字はパラメーター マーカーとして処理され、序数としてバインドされます。

BindByOrdinal に設定した場合、ネイティブ パラメーター マーカーが使用され、序数としてバインドされます。

BindByName に設定した場合、ネイティブ パラメーター マーカーが使用され、名前としてバインドされます。

PVTranslate：クライアントが、適合するエンコードをサーバーとネゴシエイトするかどうかを選択します。

Auto に設定すると、データ プロバイダーは Encoding 接続プロパティをデータベースのコード ページに設定します。また、SQL クエリ テキストは、データ エンコードではなく UTF-8 エンコードを使用して送信されます。これにより、クエリ テキスト内の NCHAR 文字列リテラルが保持されます。

Nothing（デフォルト）に設定すると、Encoding 接続プロパティの設定が使用されます。

Timestamp：Zen のタイムスタンプを文字列として格納および取得するかどうかを選択します。

DataTime（デフォルトの初期値）に設定すると、データ プロバイダーはタイムスタンプを DateTime にマップします。この設定はネイティブな精度が必要な場合、たとえば、タイムスタンプを含む CommandBuilder を使用する場合に適しています。

String に設定すると、データ プロバイダーは Zen タイムスタンプを文字列としてマップします。

TimeType：Zen の Time を Zen ADO.NET データ プロバイダーの Timespan または DateTime として取得するかどうかを指定します。

DateTime に設定すると、データ プロバイダーは SQL の TIME 型を .NET の System.DateTime 型にマップします。

TimeSpan に設定すると、データ プロバイダーは SQL の TIME 型を .NET の System.DateTimeSpan 型にマップします。

Connection Pooling（接続プール）

Connection Reset：アプリケーションで再使用するための接続プールから接続を削除する場合、その状態を接続の初期設定状態にリセットするかどうかを選択します。

False（デフォルトの初期値）に設定すると、データ プロバイダーは接続の状態をリセットしません。

Connection Pool Behavior：再使用するための接続プールからどのような順序で接続を削除するのかを、接続の使用頻度または使用時期に基づいて選択します。

MostRecentlyUsed に設定すると、データ プロバイダーは後入れ先出し法（LIFO）を用いて、最近プールに戻された接続を返します。

LeastRecentlyUsed に設定すると、データ プロバイダーは先入れ先出し法（FIFO）を用いて、最も使用回数の低い接続を返します。この値は、プール内の接続をバランスよく使用できるようにします。

MostFrequentlyUsed に設定すると、データ プロバイダーは最も使用回数の高い接続を返します。この値は、アプリケーションが一番よく使い込んだ接続を優先的に扱うことができるようにします。

LeastFrequentlyUsed に設定すると、データ プロバイダーは最も使用回数の低い接続を返します。この値は、プール内の接続をバランスよく使用できるようにします。

Connection Timeout：サーバーへの接続の試行が失敗した後、接続されないでいる間の秒数を入力します。接続フェールオーバーが有効になっている場合、このオプションは接続の試行ごとに適用されます。

0 に設定すると、データ プロバイダーが接続の試行でタイムアウトすることはありません。

デフォルトの初期値は 15 秒です。

Load Balance Timeout：接続プールで接続を保持する秒数を入力します。接続プール マネージャーは定期的にすべてのプールをチェックし、この値を超過した接続を閉じたり排除します。Min Pool Size オプションの指定によって、一部の接続は Load Balance Timeout に指定された値を無視することができます。

値は、0 から 65535 までの任意の整数を指定できます。

0（デフォルトの初期値）に設定すると、接続のタイムアウトは上限値になります。

接続の存続時間については、プールからの接続の削除を参照してください。

Max Pool Size：1 つのプール内の最大接続数を入力します。最大数に達したら、それ以上の接続を接続プールに追加することはできません。

値は、1 から 65535 までの任意の整数を指定できます。

デフォルトの初期値は 100 です。

Max Pool Size Behavior：接続プール内のすべての接続が使用中である場合に、データ プロバイダーが Max Pool Size オプションで指定された接続数を超えられるようにするかどうかを選択します。

SoftCap に設定すると、すべての接続が使用中であるときにもう 1 つ接続が要求された場合、たとえ接続プールが MaxPool Size オプションで設定された数を超えるとしても、新しい接続が作成されます。プールに接続が返されたとき、そのプールがアイドル状態の接続でいっぱいである場合には、プール メカニズムは接続プールが Max Pool Size を決して超えないよう、破棄する接続を選択します。

HardCap に設定すると、プールに許可されている最大接続数が使用中であるときは、新しい接続要求は Connection Timeout に達するまで、接続が使用可能になるのを待ちます。

Min Pool Size：接続プールが作成されたとき、そのプールで開いている、もしくは置いておく接続の最小数（下限）を入力します。接続プール内の一部の接続が Load Balance Timeout 値を超えたとしても、接続プールにはこの最小数の接続が保持されます。

値は、0 から 65535 までの任意の整数を指定できます。

0（デフォルトの初期値）に設定すると、接続プールが作成されたとき、プール内に追加の接続は置かれません。

Pooling：True（デフォルトの初期値）に設定すると、接続プールは有効になります。

Failover（フェールオーバー）

Alternate Servers：プライマリ データベース サーバーが使用不可能な場合、データ プロバイダーが接続を試みる代替データベース サーバーの一覧を入力します。このプロパティに値を指定すると、データ プロバイダーに対して接続フェールオーバー機能が有効になります。

たとえば、Alternate Servers 値を次のように指定すると、接続フェールオーバー用に 2 つの代替サーバーが定義されます。

```
Alternate Servers="Host=AcctServer;Port=1584,
Host=123.456.78.90;Port=1584"
```

Connection Retry Count：データ プロバイダーがプライマリ サーバーへの接続を試行し、その初期接続が失敗した後に代替サーバーへの接続を試行する回数を入力します。

値は、0 から 65535 までの任意の整数を指定できます。

0（デフォルトの初期値）に設定すると、再接続の試行回数は制限されません。

Connection Retry Delay：初期接続の試行に失敗した後、プライマリ サーバーまたは（指定した場合には）代替サーバーへの接続を再試行するまでの試行間隔を秒数で入力します。

デフォルトの初期値は 3 です。

このプロパティは Connection Retry Count プロパティに 0 を設定している場合は作用しません。

Load Balancing：データ プロバイダーがプライマリ データベース サーバーや代替データベース サーバーへの接続において、クライアント ロード バランスを使用するかどうかを、True または False を選択して決定します。

False（デフォルトの初期値）に設定すると、データ プロバイダーはクライアント ロード バランスを使用しません。

Performance（パフォーマンス）

Enlist：True または False を選択して、スレッドの現在のトランザクション コンテキストの作成中に、データ プロバイダーが接続への参加を自動的に試行するかどうかを指定します。

メモ：Zen は分散トランザクションをサポートしないため、スレッドの現在のトランザクション コンテキストでの接続への参加試行は失敗します。

False（デフォルトの初期値）に設定すると、データ プロバイダーは接続への自動的な参加試行を行いません。

True に設定すると、現在のトランザクション コンテキストが存在する場合にデータ プロバイダーはエラーを返します。現在のトランザクション コンテキストが存在しない場合、データ プロバイダーは警告を発します。

Max Statement Cache Size：この接続のステートメント キャッシュに保持できる、アプリケーションが生成するステートメントの最大数を入力します。

この値には 0、または 1 以上の整数を指定できます。

0 に設定すると、ステートメント キャッシュは無効になります。

1 以上の整数を設定すると、その値はステートメント キャッシュに保持できるステートメント数を決定します。

デフォルトの初期値は 10 です。

Statement Cache Mode：接続存続期間のステートメント キャッシュ モードを選択します。詳細については、ステートメント キャッシングの使用を参照してください。

Auto に設定すると、ステートメント キャッシュは有効になります。Command のプロパティ StatementCacheBehavior で Implicit とマークされているステートメントがキャッシュされます。これらのコマンドは明示的にマークされたコマンドより低い優先順位を持ちます。つまり、ステートメント プールが最大ステートメント数に達した場合、Cache とマークされたステートメント用の余地を作るため、Implicit とマークされたステートメントが最初に削除されます。

ExplicitOnly（デフォルトの初期値）を設定すると、StatementCacheBehavior によって Cache とマークされたコマンドのみがキャッシュされます。これは Zen ADO.NET Entity Framework データ プロバイダーでのみ有効な値なので、注意してください。

Schema Information（スキーマ情報）

Schema Collection Timeout：試行したスキーマ コレクション操作が失敗した後、完了されないでいる間の秒数を入力します。

デフォルトの初期値は 120 です。

Schema Options：返すことができる追加のデータベース メタデータを指定します。デフォルトで、データ プロバイダーは、パフォーマンスに悪影響を与えるデータベース メタデータを返さないようにしてパフォーマンスを最適化します。アプリケーションでこのようなデータベース メタデータを必要とする場合は、メタデータの名前または 16 進値を指定します。

このオプションはパフォーマンスに影響することがあります。

ShowColumnDefaults または 0x04 に設定すると、列のデフォルトが返されます。

ShowParameterDefaults または 0x08 に設定すると、列のデフォルトが返されます。

FixProcedureParamDirection または 0x10 に設定すると、プロシージャ定義が返されます。

ShowProcedureDefinitions または 0x20 に設定すると、プロシージャ定義が返されます。

ShowViewDefinitions または 0x40 に設定すると、ビュー定義が返されます。

ShowAll または 0xFFFFFFFF を設定すると、すべてのデータベース メタデータが返されます。

たとえば、プロシージャ定義の説明を返すには Options=ShowProcedureDefinitions または Schema Options=0x20 を指定します。

複数の除外されたメタデータを表示するには、名前をカンマ区切りのリストで指定するか、制限したい列コレクションの 16 進値の合計を指定します。

データ プロバイダーが追加できるデータベース メタデータの名前および 16 進値については、接続文字列プロパティを参照してください。

Use Current Schema：この接続文字列オプションはサポートされていません。これを設定すると、データ プロバイダーは例外をスローします。

Security（セキュリティ）

Encrypt：データ プロバイダーが暗号化されたネットワーク通信（ワイヤ暗号化とも呼ばれます）を使用するかどうかを選択します。

IfNeeded（デフォルトの初期値）に設定すると、データ プロバイダーはサーバーの設定を反映します。

Always に設定すると、データ プロバイダーは暗号化を使用します。サーバーでワイヤ暗号化が許可されない場合はエラーが返されます。

Never に設定すると、データ プロバイダーは暗号化を使用しません。サーバーでワイヤ暗号化を要求された場合はエラーが返されます。

Encryption：データ プロバイダーが許可する暗号化の最低レベルを選択します。これらの値の意味は使用する暗号化モジュールに応じて変わります。デフォルトの暗号化モジュールでは、値 Low、Medium、High はそれぞれ 40 ビット、56 ビット、128 ビット暗号化に対応しています。

デフォルトの初期値は Medium（中）です。

Password：Zen データベースへの接続に使用するパスワード（大文字小文字の区別なし）を入力します。パスワードは、データベースでセキュリティが有効な場合にのみ必要です。パスワードが必要な場合は、システム管理者からパスワードを入手してください。

Persist Security Info：セキュリティで保護された情報を ConnectionString プロパティにクリア テキストで表示するかどうかを選択します。

True に設定すると、Password 接続文字列オプションの値はクリア テキストで表示されます。

False（デフォルトの初期値）に設定すると、データ プロバイダーはセキュリティで保護された情報をクリア テキストで表示しません。

User ID：Zen データベースへの接続に使用するデフォルトの Zen ユーザー名を入力します。

Standard Connection（標準接続）

Database Name：接続するデータベースの内部名を特定する文字列を入力します。

このフィールドに値を入力した場合は、Server DSN フィールドを使用できません。

Host：接続する Zen サーバーの名前または IP アドレスを入力します。たとえば、accountingserver などのサーバー名を指定できます。あるいは、IPv4 アドレス（199.262.22.34 など）または IPv6 アドレス（2001:DB8:0000:0000:8:800:200C:417A など）を指定できます。

Port：Zen データベースで動作しているリスナーの TCP ポートを入力します。

デフォルトのポート番号は 1583 です。

Server DSN：DEMODATA など、サーバー上のデータ ソースの名前です。

このフィールドに値を入力した場合は、Database Name フィールドを使用できません。

Tracing（トレース）

Enable Trace：1 以上の値を入力すると、トレースが有効になります。0（デフォルト）に設定すると、トレースは有効になりません。

Trace File：トレース ファイルのパスと名前を入力します。指定したトレース ファイルが存在しない場合は、データ プロバイダーがファイルを作成します。デフォルトは空文字列です。

1. ［**接続の確認**］をクリックします。構成処理中のどの時点であっても、［**接続の確認**］をクリックして、［接続の追加］ウィンドウで指定した接続プロパティを使ってデータ ソースへ接続してみることができます。
   - データ プロバイダーは接続に成功したら、その接続を停止した後、"テスト接続に成功しました" というメッセージを表示します。［**OK**］をクリックします。
   - 不適当な環境であるか、または接続値が正しくないためにデータ プロバイダーが接続できない場合は、適切なエラー メッセージが表示されます。
2. ［**OK**］をクリックします。

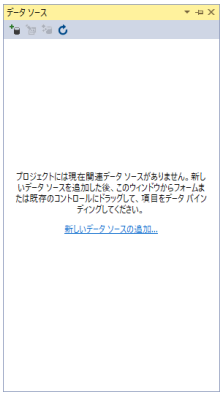
   メモ：接続フェールオーバー機能で使用するための代替サーバーを構成している場合には、［接続の確認］ボタンでテストされるのはプライマリ サーバーのみで、代替サーバーはテストされないということを承知しておいてください。
3. ［**OK**］または［**キャンセル**］をクリックします。［**OK**］をクリックした場合は、指定した値がデータ ソースへ接続するときのデフォルトになります。これらのデフォルト設定は、この手順を使ってデータ ソースを再構成することによって変更できます。また、これらの代わりの値を含む接続文字列を使ってデータ ソースに接続すれば、デフォルト設定を無効にすることができます。

データソース構成ウィザードによる接続の追加

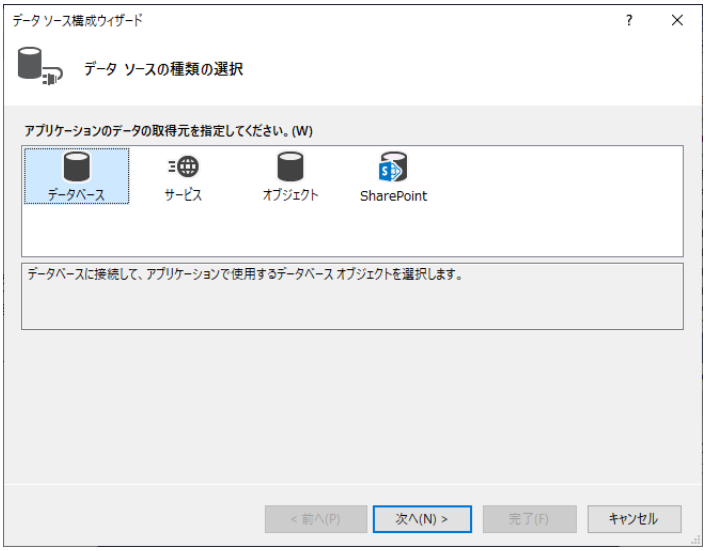
データ ソース構成ウィザードを使用して、アプリケーションに新しい接続を追加することができます。

接続を追加するには

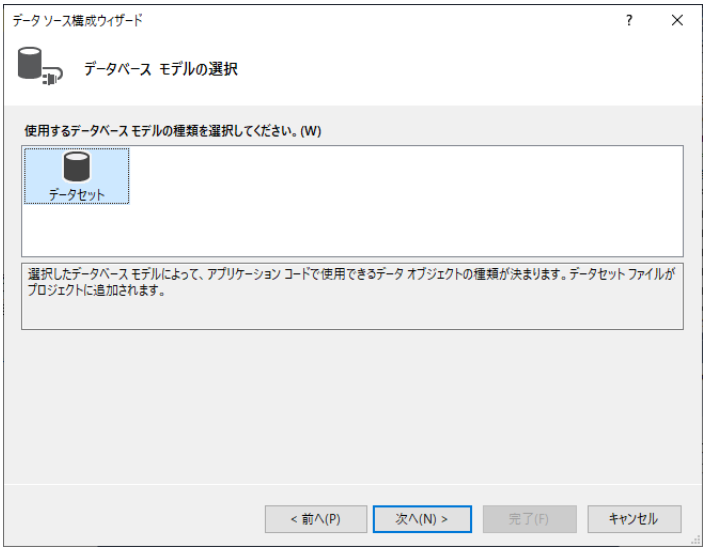
1. Visual Studio の［データ ソース］ウィンドウで、［**新しいデータ ソースの追加**］を選択します。［データ ソース］ウィンドウを開くには、メイン メニューから［**表示**］を選択し、［**その他のウィンドウ**］>［**データ ソース**］の順に選択します。



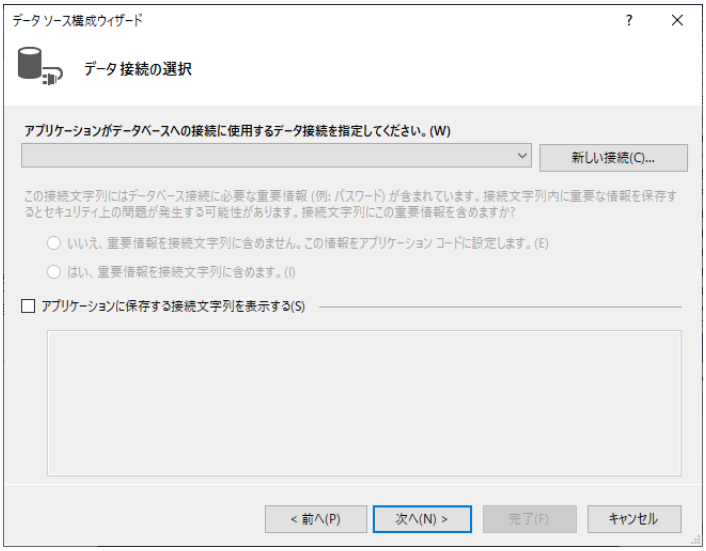
データ ソース構成ウィザードが表示されます。



2. ［**データベース**］を選択し、［**次へ**］をクリックします。［データ接続の選択］ウィンドウが表示されます。



3. ［**新しい接続**］をクリックします。［接続の追加］ウィンドウが表示されます。**サーバー エクスプローラーでの接続の追加**の手順 から処理を続けます。



Zen Performance Tuning Wizard の使用

Zen Performance Tuning Wizard は、アプリケーションについて一連の質問を通して手順を段階的に案内します。ウィザードに対する答えに基づき、Zen データ プロバイダーのパフォーマンスに関する接続文字列オプションに最適な設定が用意されます。

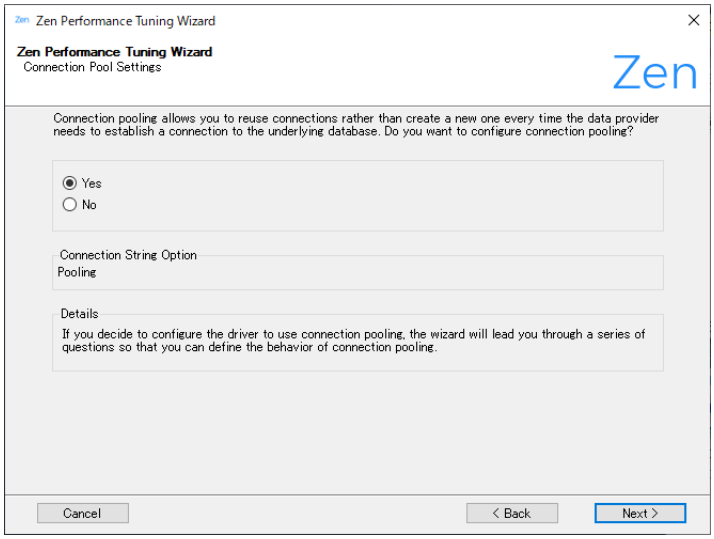
Visual Studio から Zen Performance Tuning Wizard を起動すると、以下のことが行えます。

- パフォーマンスに関する接続文字列オプションの値を生成します。これらの値は接続文字列にコピーされます。
- 既存の接続を変更します。
- お使いの環境に最適化された接続文字列を持つ、あらかじめ設定された新規アプリケーションを生成します。Performance Tuning Wizard はアプリケーションの種類および使用する ADO.NET コードのバージョンを選択するオプションを提供します。

Visual Studio で Zen Performance Tuning Wizard を使用するには

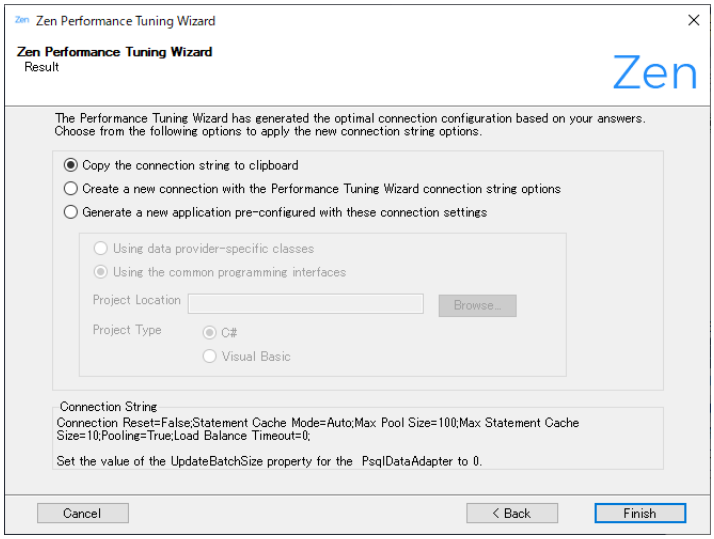
1. Performance Tuning Wizard を起動するには次のいずれかを行います。
   - 新しい接続を作成するには、［**ツール**］>［**Actian Zen**］>［**Run Zen Performance Tuning Wizard**］をクリックします。Performance Tuning Wizard の Welcome ダイアログが表示されたら、［**Next**］をクリックします。手順 2 から処理を続けます。
   - 既存の接続を変更するには、サーバー エクスプローラーでデータ接続を右クリックし、［**Run Zen Performance Tuning Wizard**］をクリックします。Performance Tuning Wizard の Welcome ダイアログが表示されたら、［**Next**］をクリックします。手順 2 から処理を続けます。
2. ウィザードは、お使いの環境に関する一連の質問を行います。デフォルトの答えのままにするか、必要に応じて答えを変更し、［Result］ページにたどり着くまで、［**Next**］をクリックして処理を進めます。

次のスクリーン ショットは、尋ねられる質問の一例を示しています。



3. データ プロバイダーに関する質問に答え終わると、［Result］ダイアログが表示され、接続文字列が表示されます。

   次のスクリーン ショットは、Zen Performance Tuning Wizard が生成するパフォーマンスに関する接続文字列オプションを示しています。



4. 次のいずれかを選択します。

   - 接続文字列をほかのアプリケーションでも使用できるようにするには（デフォルトの初期値）、［**Copy the connection string to clipboard**］をクリックします。この接続文字列をほかのアプリケーションでも使用できます。

- 新規の接続または既存の接続を使用してウィザードを起動したかどうかに基づいて、次のオプションのいずれかを選択します。
  - **Create a new connection with the Performance Tuning Wizard connection string options**

    このオプションを選択して［**Finish**］をクリックすると、［接続の変更］ダイアログ ボックスが表示されます。ここでは、ホスト、パスワード、ユーザー ID など、接続情報を指定する必要があります。
  - **Re-configure the connection with the additional Performance Tuning Wizard connection string options**
  - 新規アプリケーションを作成するには、［**Generate a new application pre-configured with these connection settings**］を選択します。

    このオプションを選択して［**Finish**］をクリックすると、Zen Project テンプレートを使用して Zen アプリケーションが生成されます。プロバイダー固有のテンプレートの詳細については、**プロジェクトの新規作成**を参照してください。

5. 新規アプリケーションの場合は以下の追加情報を定義します。
   - ADO.NET 2.0 仕様に準拠したアプリケーションを作成するには、［**Using data provider-specific interfaces**］を選択します。
   - ADO.NET 共通プログラミング モデルを使用するアプリケーションを作成するには、［**Using common programming interfaces**］を選択します。
   - プロジェクトの場所を入力するか、［**Browse**］をクリックして場所を選択します
   - プロジェクトの種類を選択します。デフォルトでは、ウィザードは C# アプリケーションを作成します。
6. ［**Finish**］をクリックして Zen Performance Tuning Wizard を終了します。

プロバイダー固有テンプレートの使用

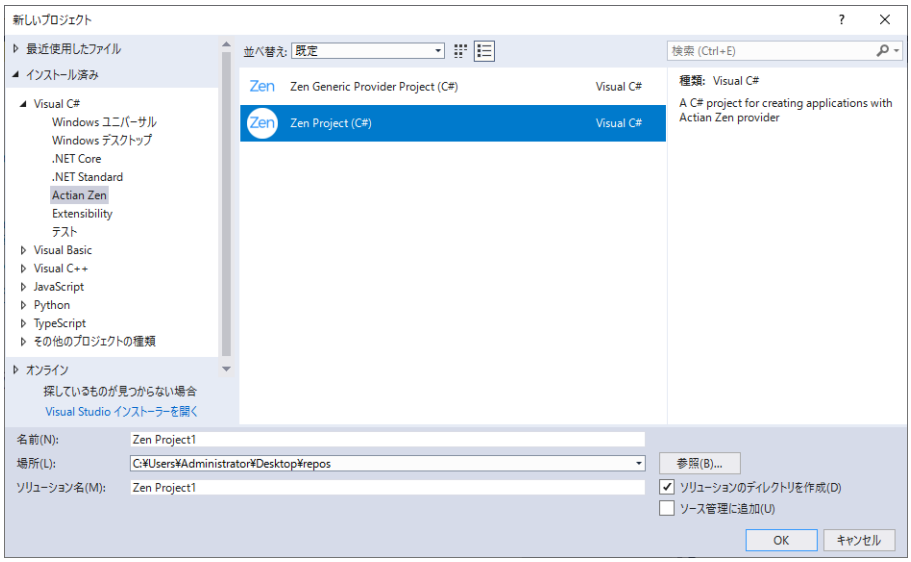
Visual Studio は SQL レベル調整などの機能を自動的に含めるアプリケーションの構築を助けるテンプレートのセットを提供します。

プロジェクトの新規作成

Visual Studio で新規プロジェクトを作成する場合、Zen データ プロバイダー固有のテンプレートを使用するか、汎用コードを持つアプリケーションを作成するテンプレートを使用することができます。

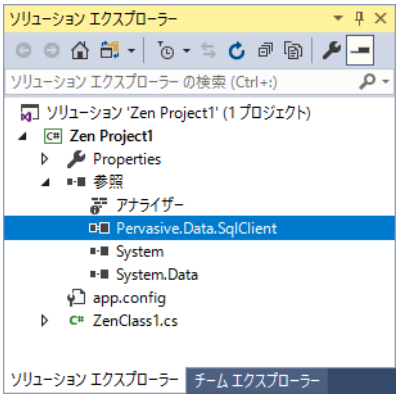
次の例では、Zen データ プロバイダー用のテンプレートを使用して Visual Studio で新規プロジェクトを作成します。

1. ［**ファイル**］>［**新規作成**］>［**プロジェクト**］を選択します。［新しいプロジェクト］ダイアログが表示されます。
2. ［インストール済み］リストで、［**Visual C#**］>［**Actian Zen**］を選択します。
3. 中央のペインで **Zen Project** を選択します。



4. 必要に応じて他のフィールドに変更を加え、［**OK**］をクリックします。

5. 新規プロジェクトがソリューション エクスプローラーに表示されます。Zen ADO.NET データ プロバイダー用の名前空間が、プロジェクトに自動的に追加されます。



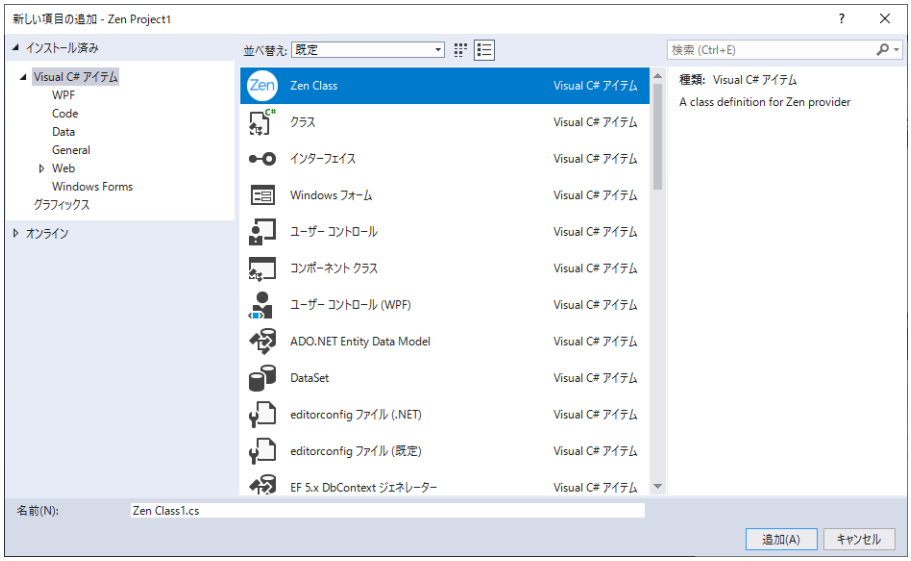
メモ：ADO.NET 2.0 共通プログラミング モデルを使用している場合は、Zen Generic Provider Project テンプレートを選択してください。この場合、プロジェクトはアセンブリへの特定の参照を必要としません。

既存のプロジェクトへのテンプレートの追加

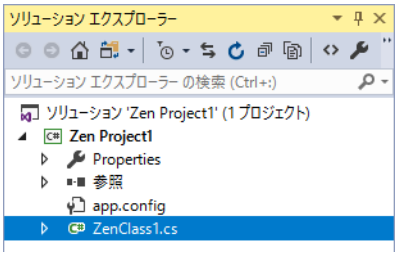
Zen テンプレートを既存のプロジェクトに追加するには

1. ソリューション エクスプローラーでプロジェクトを右クリックし、［**追加**］>［**新しい項目**］をクリックします。

2. ［**新しい項目の追加**］ダイアログで Zen Class を選択します。



3. ［**追加**］をクリックします。Zen データ プロバイダー用のクラスがプロジェクトに追加されます。



Zen Visual Studio Wizard の使用

ウィザードは、アプリケーション作成時に一般的に実行される作業です。

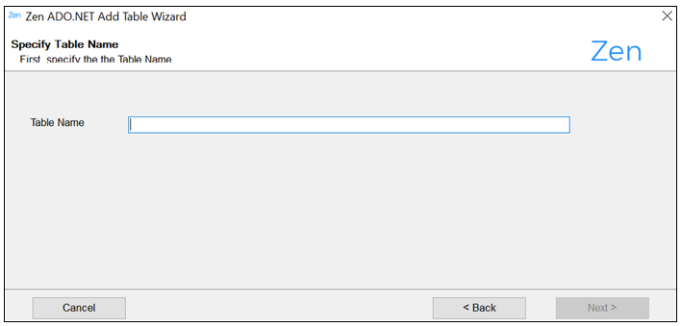
- Add Table Wizard でのテーブルの作成
- Add View Wizard でのビューの作成

この手順を開始する前に、プロジェクトの新規作成で説明したように Zen テンプレートを使用してプロジェクトを作成し、データ接続を追加します。

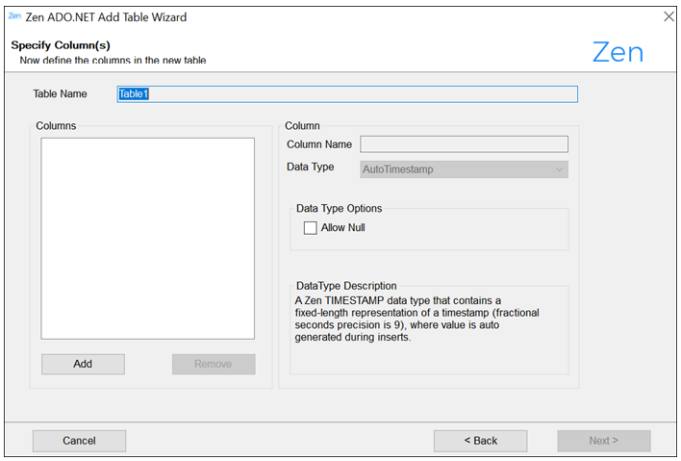
Add Table Wizard でのテーブルの作成

Visual Studio で Zen ADO.NET Add Table Wizard を使用して、新規テーブルをすばやく簡単に定義できます。

1. ［**表示**］>［**サーバー エクスプローラー**］をクリックします。
2. データ ソース接続をダブルクリックして、その下部のノードを表示します。
3. ［**Tables**］ノードを右クリックし、［**Add New Table**］をクリックします。Zen ADO.NET Add Table Wizard が表示されます。
4. ［**Next**］をクリックします。［Specify Table Name］ダイアログが表示されます。
5. ［**Table Name**］フィールドにテーブルの名前を入力します。

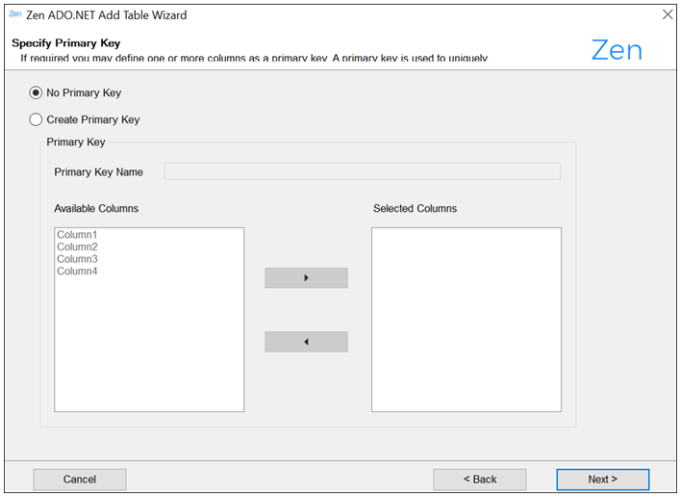


6. ［**Next**］をクリックします。［Specify Column(s)］ダイアログが表示されます。



7. 新規テーブルの列を定義します。この選択により、データ型オプション ペインに追加フィールドが表示されます。
   - ［**Add**］をクリックしてテーブルに列を追加します。列名とデータ型のフィールドが編集可能になります。
   - ［Column Name］フィールドに列の名前を入力します。
   - 列のデータ型を選択し、必要に応じて追加情報を指定します。
     - 文字データ型を選択した場合は、Data Type Options ペインに［Length］フィールドが表示されます。列の最大サイズをバイト単位で入力します。
     - 数値を選択した場合、Data Type Options ペインに［Precision］と［Scale］フィールドが表示されます。
   - 列にヌル値を含むことができる場合は、［**Allow Null**］チェック ボックスをオンにします。
   - テーブルから列を削除するには、列名を選択して［**Remove**］をクリックします。

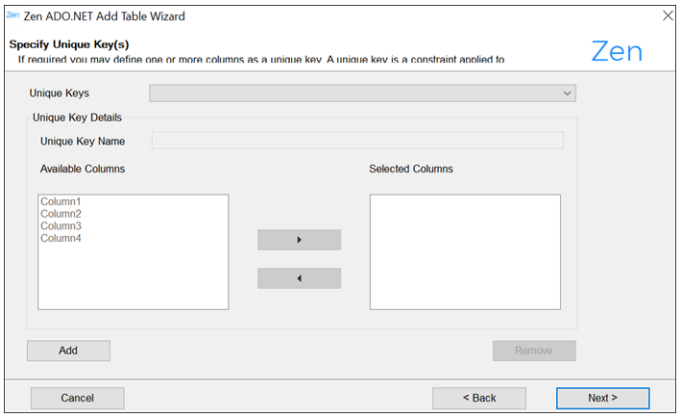
8. ［**Next**］をクリックします。［Specify Primary Key］ダイアログが表示されます。



9. 次のいずれかを実行します。
   - テーブルに主キーを指定したくない場合は［**No Primary Key**］をクリックし、［**Next**］をクリックします。［Specify Unique Key(s)］ダイアログが表示されます。手順 12 から処理を続けます。
   - テーブルに主キーを指定する場合は［**Create Primary Key**］をクリックして手順 10 に進みます。

10. ［Specify Primary Key］ダイアログのフィールドを完成します。
    - ［Primary Key Name］フィールドで主キーの名前を入力するか、デフォルトの名前をそのまま使用します。
    - ［Available Columns］フィールドから列を選択して［Selected Columns］フィールドへ移動します。

11. [**Next**] をクリックします。[Specify Unique Key(s)] ダイアログが表示されます。



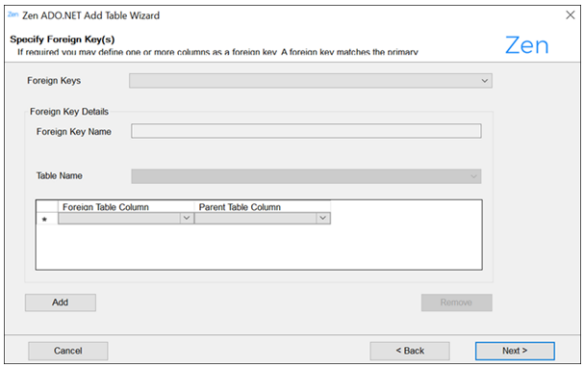
12. 次のいずれかを実行します。

- テーブルに一意なキーを指定したくない場合は、[**Next**] をクリックします。[Specify Foreign Key(s)] ダイアログが表示されます。手順 15 から処理を続けます。
- テーブルに 1 つ以上の一意なキーを指定する場合は、手順 13 から処理を続けます。

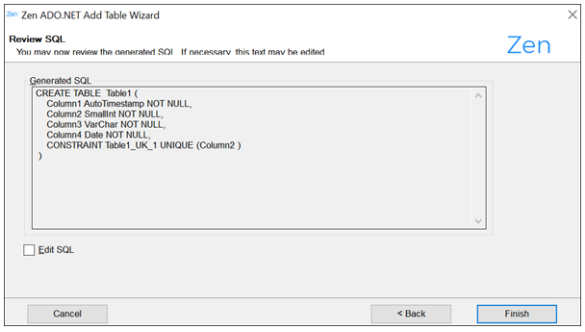
13. [**Add**] をクリックします。ダイアログ上のフィールドが選択可能になります。

- [**Unique Keys**] ドロップダウン リストで一意なキーを選択します。
- [**Unique Key Name**] フィールドで名前を編集するか、デフォルトの名前をそのまま使用します。
- [**Available Columns**] リスト ボックスで一意なキーの指定に使用する列を 1 つ以上選択し、[Selected Columns] リスト ボックスに移動します。

14. [**Next**] をクリックします。[Specify Foreign Key(s)] ダイアログが表示されます。



15. 次のいずれかを実行します。
    - テーブルに外部キーを指定したくない場合は、［**Next**］をクリックします。［Review SQL］ダイアログが表示されます。手順 18 から処理を続けます。
    - テーブルに 1 つ以上の外部キーを指定する場合は、手順 16 から処理を続けます。
16. ［**Add**］をクリックします。ダイアログ上のフィールドが選択可能になります。
    - ［**Foreign Keys**］ドロップダウン リストで外部キーを選択します。
    - ［**Foreign Key Name**］フィールドで名前を編集するか、デフォルトの名前をそのまま使用します。
    - ［**Table Schema**］リストでテーブル スキーマを選択します。
    - ［**Table Name**］リストでテーブルを選択します。
    - ［**Foreign Table Column**］リストで、外部キーの指定に使用する 1 つ以上の外部キーを選択します。
    - ［**Parent Table Column**］リストで、親テーブルから対応する列を選択します。
17. ［**Next**］をクリックします。［Review SQL］ダイアログが表示されます。



18. 選択した内容によって生成された SQL ステートメントを確認します。
    - SQL ステートメントに問題がなければ、［**Finish**］をクリックします。作成したテーブルは、サーバー エクスプローラー内のこの接続の［Tables］ノード下に表示されます。
    - SQL ステートメントを追加したい場合、たとえば、ビューや特定のキーワードを追加するには、手順 19 から処理を続けます。
19. ［**Edit SQL**］チェック ボックスをオンにします。［Generated SQL］フィールド内のテキストが編集可能になります。

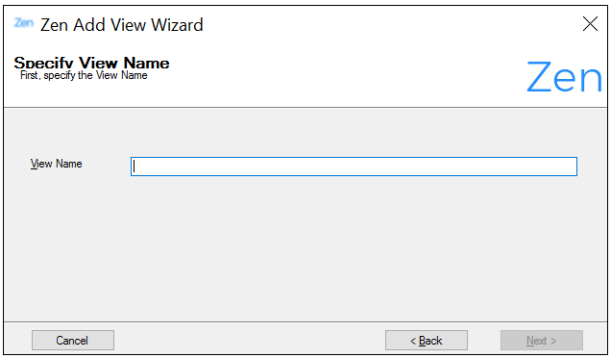
メモ：［Edit SQL］チェック ボックスをオンにすると、［Back］ボタンが無効になります。

20. SQL ステートメントの変更に問題がなければ［**Finish**］をクリックします。作成したテーブルは、サーバー エクスプローラー内のこの接続の［Tables］ノード下に表示されます。

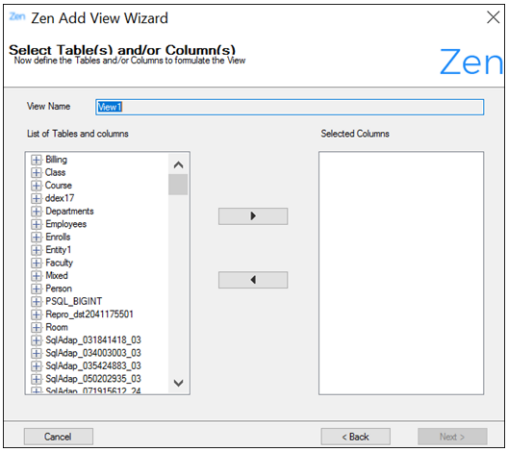
Add View Wizard でのビューの作成

Visual Studio で Zen Add View Wizard を使用して、新規ビューをすばやく簡単に定義できます。

1. まだ開いていない場合は、［**表示**］>［**サーバー エクスプローラー**］をクリックします。
2. データ ソース接続をダブルクリックして、その下部のノードを表示します。
3. ［**Views**］ノードを右クリックし、［**Add New View**］をクリックします。Zen Add View Wizard の Welcome ダイアログが表示されます。
4. ［**Next**］をクリックします。［Specify View Name］ダイアログが表示されます。
5. ［View Name］フィールドにビューの名前を入力します。

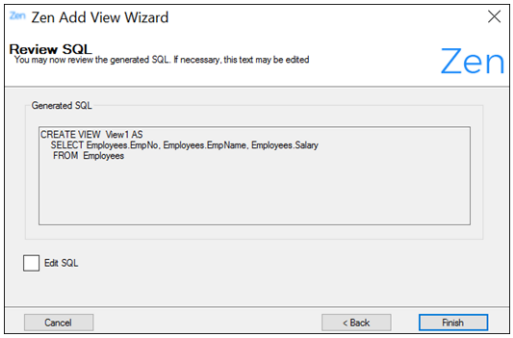


6. ［**Next**］をクリックします。［Select Table(s) and/or Column(s)］ダイアログが表示されます。



7. ［**List of Tables and columns**］リスト ボックスでビューを構成するテーブルまたは列を選択し、それを［Selected Columns］列に移動します。

8. ［**Next**］をクリックします。［Review SQL］ダイアログが表示されます。



9. 選択した内容によって生成された SQL ステートメントを確認します。
   - SQL ステートメントに問題がなければ、［**Finish**］をクリックします。作成したビューは、サーバー エクスプローラー内のこの接続の［ビュー］ノード下に表示されます。
   - SQL ステートメントを追加したい場合、たとえば、ビューや特定のキーワードを追加するには、手順 11 から処理を続けます。

10. ［**Edit SQL**］チェック ボックスをオンにします。［Generated SQL］フィールド内のテキストが編集可能になります。

メモ：［Edit SQL］チェック ボックスをオンにすると、［Back］ボタンが無効になります。

11. SQL ステートメントの変更に問題がなければ［**Finish**］をクリックします。作成したビューは、サーバー エクスプローラー内のこの接続の［ビュー］ノード下に表示されます。

ツールボックスからのコンポーネントの追加

Visual Studio のツールボックスから Windows フォーム アプリケーションにコンポーネントを追加することができます。Windows フォーム アプリケーションの作成については、Visual Studio オンライン ヘルプを参照してください。

この手順を開始する前に、Windows フォーム アプリケーションを作成してデータ接続を追加してください。

Windows フォーム アプリケーションに Zen データ プロバイダー コンポーネントを追加するには

1. ［**表示**］>［**ツールボックス**］をクリックします。ツールボックスで Zen ADO.NET Provider が表示されるまで下へスクロールします。
2. **PsqlCommand** ウィジェットを選択して Windows フォーム アプリケーションにドラッグします。
3. 必要に応じて、アプリケーションへのウィジェットの追加を続けます。

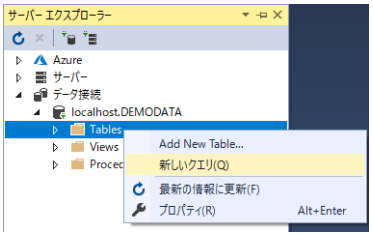
データ プロバイダー統合のシナリオ

Zen データ プロバイダーは Visual Studio に統合されているため、多くのデータ アクセス作業を簡略化することができます。たとえば、データベースへの接続を作成した後、クエリ ビルダーを使用してクエリを作成することができます。

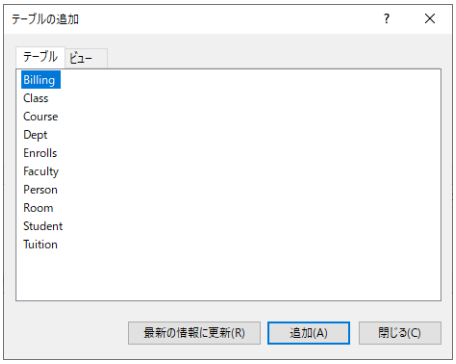
クエリ ビルダーによって、データベース クエリをグラフィカルに設計できます。

単純なクエリを作成するには

1. データ ソース接続を確立します（Zen Visual Studio Wizard の使用を参照してください）。
2. サーバー エクスプローラーでデータ ソースを選択します。
3. ［**Tables**］を右クリックして、［**新しいクエリ**］を選択します。

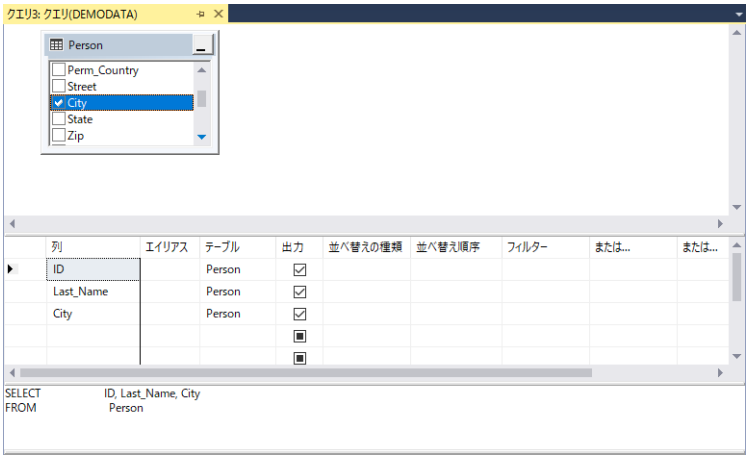


4. ［テーブルの追加］ウィンドウが表示されます。使用するデータが入っているテーブルを選択したら、［**追加**］をクリックします。

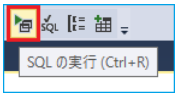


5. ［**閉じる**］をクリックして、［テーブルの追加］ウィンドウを閉じます。

6. 返したい列を選択します。この例では、employee テーブルの id、name および salary 列を選択します。



7. ツール バーの［**SQL の実行**］ボタンをクリックします。



8. 表示された結果を調べます。

A. サポートされる .NET オブジェクト

ADO.NET 2.0 は、アプリケーションおよびデータ ソース間のより汎用的な追加インターフェイスを提供する新しい一連のクラスを導入ました。

ADO.NET 2.0 より以前に使われていたものは、アプリケーションが使用するデータ プロバイダーのそれぞれの特定のインスタンスに、データ プロバイダーをより密接な要因の 1 つとして含めることを選択しました。それとは対照的に、ADO.NET2.0 以降は、アプリケーションが異種のデータ ソースのセットを単一の API で処理できるようにする、基本クラスのセットを提供します。これは、今日の ODBC および JDBC で利用されるものとよく似ています。つまり、ADO.NET2.0 以降では、すべてのデータ クラスは基本クラスから派生し、特定の専用名前空間である System.Data.Common に存在します。

データ プロバイダーでは、以下が使用できます。

- .NET の基本クラス
- データ プロバイダー固有のクラス
- Zen Common Assembly

.NET の基本クラス

ADO.NET 1.0 および ADO.NET 1.1 データ プロバイダーのインターフェイスは、アプリケーションの互換性を維持するために構築されました。ADO.NET 2.0 以降の基本クラスは以下の追加機能を提供します。

- DbCommand
- DbCommandBuilder
- DbConnection
- DbDataAdaptor
- DbDataReader
- DBDataPermission
- DbParameter
- DbParameterCollection
- DbConnectionStringBuilder
- DbTransaction

日々のプログラミングという観点から、これらのクラスは抽象実装として提供されます。つまり、これらは直接インスタンス化できず、プロバイダー ファクトリで使用する必要があります。各データ プロバイダーは、DbFactory クラスから派生する一連の静的メソッドを含む PsqlFactory のような Factory クラスを提供する必要があります。これらの静的メソッドは、それぞれベース クラスのインスタンスを生成するファクトリ メソッドです。

これは、データ プロバイダーのインストール時に .NET Framework に登録されます。これにより、共通 .NET Framework DbFactory は、アプリケーションが必要とする登録済みデータ プロバイダーを見つけて、データ ソースへの接続を確立するための共通メカニズムを提供することができます。最終的に、.NET Framework は ADO.NET データ ソース用の完全に成熟した共通プログラミング API を提供します。

データ プロバイダー固有のクラス

ADO.NET データ プロバイダーは、すべての .NET パブリック オブジェクトをサポートしています。ADO.NET データ プロバイダーは、.NET のパブリック オブジェクトにプロバイダー固有のプレフィックス 「*Psql* 」を付けます（例：PsqlCommand）。

以下のオブジェクトについて説明します。

- PsqlBulkCopy
- PsqlBulkCopyColumnMapping
- PsqlBulkCopyColumnMappingCollection
- PsqlCommand オブジェクト
- PsqlCommandBuilder オブジェクト
- PsqlConnection オブジェクト
- PsqlConnectionStringBuilder オブジェクト
- PsqlCredential オブジェクト
- PsqlDataAdapter オブジェクト
- PsqlDataReader オブジェクト
- PsqlError オブジェクト
- PsqlErrorCollection オブジェクト
- PsqlException オブジェクト
- PsqlFactory オブジェクト
- PsqlInfoMessageEventArgs オブジェクト
- PsqlParameter オブジェクト
- PsqlParameterCollection オブジェクト
- PsqlTrace オブジェクト
- PsqlTransaction オブジェクト

パブリック オブジェクトに関する詳細については、Microsoft .NET Framework Version 2.0 SDK のドキュメントを参照してください。

PsqlBulkCopy

PsqlBulkCopy オブジェクトは ADO.NET Bulk API に類似した API パターンを使用し、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

PsqlBulkCopyColumnMapping

PsqlBulkCopyColumnMapping オブジェクトは ADO.NET Bulk API に類似した API パターンを使用し、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

PsqlBulkCopyColumnMappingCollection

PsqlBulkCopyColumnMappingCollection オブジェクトは Microsoft SqlBulkCopyColumnMappingCollection クラスに類似した API パターンに従い、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

PsqlCommand オブジェクト

次の表では、PsqlCommand オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
AddRowID	ROWID を SQL ステートメント選択リストの一部として追加します。 True に設定した場合、PsqlCommandBuilder を使用する際には、より効率的な Insert、Delete、および Update コマンドを生成するために、ROWID に返された値が利用されます。 False（デフォルトの初期値）に設定すると、データ プロバイダーは ROWID 列を選択リストに追加しません。

プロパティ	説明
ArrayBindCount	使用されるパラメーターの行数を指定します。アプリケーションでは、パラメーター配列バインドを使用するコマンドを実行する前にこのプロパティを設定しておく必要があります。この数はパラメーター値ごとに設定される各配列の長さと等しくなるようにしてください。 デフォルトの初期値は 0 です。アプリケーションはパラメーター配列バインドを使用しません。
ArrayBindStatus	行の状態値の配列を返します。このプロパティは、パラメーター配列バインドを使用するコマンドを実行した後、アプリケーションで行ごとの状態を調べることができるようにします。このプロパティの型は PsqlRowStatus の配列です。 パラメーター配列バインドは単独のアトミック操作として実行されます。つまり、操作が成功した場合はすべてのエントリに OK が設定されますが、失敗した場合にはどのエントリにも OK が設定されないということです。 PsqlRowStatus 列挙で設定可能な値は以下のとおりです。 • OK。操作は正常に終了しました。エントリはすべて OK としてマークされます。 • Failed。操作は失敗しました。データ プロバイダーでは、エラーが発生した行以外のすべてのステータス エントリにこの値を割り当てます。 • SchemaViolation。操作が失敗した場合、データ プロバイダーはこの値をエラーが発生した行に割り当てます。

プロパティ	説明
BindByName	ストアド プロシージャの実行時にデータ プロバイダーが名前付きパラメーターをどのように処理するかを指定します。アプリケーションは、ストアド プロシージャに対して名前付きパラメーターまたはパラメーターのデフォルト値を使用できます。 True に設定すると、データ プロバイダーは Zen サーバーへのパラメーター バインドに PsqlParameter オブジェクトで提供されるパラメーターの名前を使用します。CommandText の例を参照してください。 別の方法として、ユーザーは以下の方法のいずれかを使用して名前付きパラメーターのデフォルト値を指定することができます。 • アプリケーションは名前付きパラメーターを使用してパラメーターをバインドしますが、アプリケーションがデフォルト値を使用したい PsqlParameterCollection には PsqlParameter オブジェクトを追加しません。 • アプリケーションは PsqlParameter オブジェクトの Value プロパティをヌルに設定します。データ プロバイダーはこのパラメーターをサーバーに送信せず、ストアド プロシージャ実行時にパラメーターのデフォルト値を使用します。 BindByName を True に設定し、Parameter Mode 接続文字列オプションが BindByName または BindByOrdinal と定義されている場合、接続文字列に定義されたこれらの値は Command オブジェクトの存続期間中、無効にされます。 False（デフォルトの初期値）に設定すると、データ プロバイダーは PsqlParameter オブジェクトで提供されるパラメーターの名前を無視し、パラメーターが Create Procedure ステートメントで指定されたのと同じ順序であるものと仮定します。

プロパティ	説明

CommandText	データ ソースで実行するテキスト コマンドを取得または設定します。
	ストアド プロシージャを使う場合、CommandText にはそのストアド プロシージャの名前を設定します。たとえば次のようになります。
	cmd.CommandType = System.Data.CommandType.StoredProcedure;
	cmd.CommandText = "call EnrollStudent(!!Stud_id!!,!!Class_Id!!, !!GPA!!)";
	cmd.BindByName = true;
	PsqlParameter Class_Id = new PsqlParameter();
	Class_Id.Value = 999;
	Class_Id.ParameterName = "!!Class_Id!!";
	PsqlParameter Stud_id = new PsqlParameter();
	Stud_id.Value = 1234567890;
	Stud_id.ParameterName = "!!Stud_id!!";
	PsqlParameter GPA = new PsqlParameter();
	GPA.Value = 3.2;
	GPA.ParameterName = "!!GPA!!";
	cmd.Parameters.Add(Class_Id);
	cmd.Parameters.Add(Stud_id);
	cmd.Parameters.Add(GPA);

CommandTimeout	コマンド実行の試行を終了し、エラーを生成するまでの待機時間を取得または設定します。 デフォルトの初期値は 30 秒です。 CommandTimeout プロパティは、サーバーで最大のデフォルトのタイムアウト値よりも大きな値に設定することをお勧めします。これによって、アプリケーションはタイムアウトした場合にもっと意味のある応答を得られるようになります。
CommandType	CommandText プロパティの解釈方法を指示または指定します。 ストアド プロシージャを使用するには、CommandType を StoredProcedure に設定します。
Connection	この IDbCommand のインスタンスで使用する IDbConnection を取得または設定します。
Parameters	PsqlParameterCollection を取得します。

プロパティ	説明
RowSetSize	この Command オブジェクトで実行されるクエリによって返される行数を、実行時に指定される値までに制限します。Read-Write プロパティは符号付き整数です。 有効な値は 0 から 2147483647 です。 0（デフォルトの初期値）に設定すると、データ プロバイダーは返される行数を制限しません。
StatementCacheBehavior	ステートメント キャッシュの動作を取得する、または PsqlStatementCacheBehavior 列挙のいずれかの値のステートメント キャッシュ動作を設定します。詳細については、ステートメント キャッシングの有効化を参照してください。 Implicit（デフォルト）に設定すると、Statement Cache Mode 接続文字列オプションに Auto が設定され、ステートメント キャッシュは暗黙的に行われます。 Cache に設定し、Statement Cache Mode 接続文字列オプションを ExplicitOnly に設定すると、ステートメントは Cache がキャッシュされていると認識します。 DoNotCache に設定すると、ステートメント キャッシュは行われません。
Transaction	PsqlCommand オブジェクトを実行するトランザクションを取得または設定します。
UpdatedRowSource	DataAdapter の Update メソッドで DataRow が使用されたとき、コマンドの結果を DataRow に適用する方法を取得または設定します。 UpdateBatchSize プロパティに 1 以外の値が設定された場合は、UpdateCommand、DeleteCommand、および InsertCommand の UpdatedRowSource プロパティは None または OutputParameters に設定する必要があります。 None に設定すると、返されたパラメーターや行は無視されます。 OutputParameters に設定すると、出力パラメーターは DataSet の変更された行にマップされます。

次の表では、PsqlCommand オブジェクトでサポートされるパブリック メソッドについて説明します。

メソッド	説明
Cancel	IDbCommand の実行のキャンセルを試みます。

メソッド	説明
CreateParameter	IDbDataParameter オブジェクトの新しいインスタンスを作成します。
Dispose	コンポーネントで使用したリソースを解放します。オーバーロードされます。
ExecuteNonQuery	PsqlConnection オブジェクトに対して SQL ステートメントを実行し、影響を受けた行数を返します。これは結果を返さないコマンドを対象としたメソッドです。
ExecuteReader	接続に対して CommandText を実行し、IDataReader を構築します。
ExecuteScalar	クエリを実行し、そのクエリが返す結果セットの最初の行の最初の列を返します。残りの行も列も無視されます。
Prepare	Zen のインスタンスに対する準備済みのコマンドを作成します。 **メモ**：Prepare メソッドは本リリースのデータ プロバイダーでは無効です。

PsqlCommandBuilder オブジェクト

PsqlCommandBuilder オブジェクトを使用すると、パフォーマンスに悪影響を及ぼす恐れがあります。同時実行が制限されるため、PsqlCommandBuilder では効率のよい SQL ステートメントを生成することができません。多くの場合、エンド ユーザーの方が、PsqlCommandBuilder オブジェクトで生成されるものより効率のよい Update ステートメントや Delete ステートメントを作成することができます。

次の表では、PsqlCommandBuilder オブジェクトでサポートされるパブリック プロパティについて説明します。

プロパティ	説明
DataAdapter	この PsqlCommandBuilder に関連付けられている PsqlDataAdapter オブジェクトを取得または設定します。

次の表では、PsqlCommandBuilder オブジェクトでサポートされるパブリック メソッドについて説明します。

メソッド	説明
DeriveParameters	PsqlCommand で指定したストアド プロシージャのパラメーター情報を取得し、指定した PsqlCommand オブジェクトの Parameters コレクションにパラメーターを格納します。
GetDeleteCommand	アプリケーションが PsqlDataAdapter に対して Delete を呼び出したときに、データベースで削除処理を実行するための、自動生成された PsqlCommand オブジェクトを取得します。
GetInsertCommand	アプリケーションが PsqlDataAdapter に対して Insert を呼び出したときに、データベースで挿入処理を実行するための、自動生成された PsqlCommand オブジェクトを取得します。
GetUpdateCommand	アプリケーションが PsqlDataAdapter に対して Update を呼び出したときに、データベースで更新処理を実行するための、自動生成された PsqlCommand オブジェクトを取得します。
QuoteIdentifier	正しいカタログの中に引用符で囲まれていない識別子があると、その識別子の引用符で囲まれた正しい形式を返します。識別子内の埋め込み引用符もすべて適切にエスケープされます。
UnquoteIdentifier	引用符で囲まれた識別子があると、その識別子の引用符で囲まれていない正しい形式を返します。識別子内の埋め込み引用符のエスケープ処理も適切に除かれます。

PsqlConnection オブジェクト

PsqlConnection オブジェクトは、次の表に示すパブリック コンストラクターをサポートしています。

プロパティ	説明
PsqlConnection()	PsqlConnection クラスの新しいインスタンスを初期化します。
PsqlConnection(string connectionString)	指定された接続文字列を含む文字列に対応する、PsqlConnection クラスの新しいインスタンスを初期化します。
PsqlConnection(string connectionString, PsqlCredential credential)	指定された接続文字列、およびユーザー ID とパスワードを含む PsqlCredential オブジェクトに対応する、PsqlConnection クラスの新しいインスタンスを初期化します。

PsqlConnection オブジェクトは、次の表に示すパブリック プロパティをサポートしています。いくつかのプロパティでは、対応する接続文字列オプションに指定された値を返します。接続文字列オプションとは異なり、PsqlConnection のプロパティ名にはスペースが含まれません。

プロパティ	説明
ConnectionString	データベースを開くために使用する文字列を取得または設定します。設定可能な値の説明については、接続文字列プロパティを参照してください。
ConnectionTimeout	データ プロバイダーが接続の試行を中断してエラーを生成するまでの、接続の確立を待機する時間を取得または設定します。 ConnectTimeout プロパティまたは Connection Timeout 接続文字列オプションを使用すると、接続がタイムアウトするのを待つ時間を設定できます。 接続フェールオーバーが有効になっている（AlternateServers プロパティに代替データベース サーバーが 1 つ以上定義されている）場合は、代替サーバーへの接続の試行ごとにこのプロパティが適用されます。接続の再試行も有効になっている（Connection Retry Count 接続文字列オプションに 0 より大きい整数が設定されている）場合は、再試行ごとに ConnectionTimeout プロパティが適用されます。
Credential	Zen サーバーに接続するためのパスワードをより安全に指定する方法を提供します。PsqlCredential は、Zen サーバーへの接続に使用されるユーザー ID とパスワードで構成されます。パスワードを保持する SecureString オブジェクトは、読み取り専用とマークする必要があります。
Database	現在のデータベースの名前、または接続を開くときに使用するデータベースの名前を取得します。
Host	Host 接続文字列オプションに指定された値を返します。読み取り専用です。
Port	Port 接続文字列オプションに指定された値を返します。読み取り専用です。
ServerDSN	Server DSN 接続文字列オプションに指定された値を返します。読み取り専用です。
ServerName	Server Name 接続文字列オプションに指定された値を返します。読み取り専用です。

プロパティ	説明
ServerVersion	このオブジェクトが現在接続している Zen サーバーのバージョンが示された文字列を返します。 PsqlConnection オブジェクトが現在接続されていない場合、データ プロバイダーでは InvalidOperation 例外を生成します。
State	現在の接続の状態を取得します。
StatisticsEnabled	統計情報の収集を有効にします。 True に設定すると、現在の接続に関する統計情報の収集が有効になります。

次の表では、PsqlConnection オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
BeginTransaction	データベース トランザクションを開始します。 オーバーロードされた BeginTransaction(IsolationLevel) メソッドの使用時、データ プロバイダーでは ReadCommitted および Serializable 分離レベルが使用できます。詳細については、分離レベルを参照してください。
ChangeDatabase	開いている Connection オブジェクトの現在のデータベースを変更します。
ClearAllPools	データ プロバイダーの接続プールを空にします。
ClearPool	接続に関連付けられている接続プールを空にします。 呼び出し時、接続プールに関連付けられている追加の接続が使用中である場合、それらには適切なマーク付けがされ、Close を呼び出した時点で破棄されます。
Close	データベースへの接続を閉じます。
CreateCommand	PsqlConnection に関連付けられている PsqlCommand オブジェクトを作成し、返します。
Dispose	PsqlConnection オブジェクトで使用したリソースを解放します。
Open	PsqlConnection オブジェクトの ConnectionString プロパティで指定した設定を使用して、データベース接続を開きます。

メソッド	説明
ResetStatistics	接続上の現在の統計収集セッションのすべての値をゼロにリセットします。 接続が閉じられて接続プールに戻ると、統計情報収集はオフに切り替えられてカウントがリセットされます。
RetrieveStatistics	統計情報収集が有効になっている接続の一連の統計情報を取得します。返される " 名前 = 値 " の組み合わせは、このメソッドが呼び出された時点の接続状態のスナップショットを形成します。

PsqlConnection オブジェクトの InfoMessage イベントを使用すると、データベースから警告や情報メッセージを取得することができます。データベースからエラーが返される場合は、例外が発生します。データベース サーバーから送られる警告や情報メッセージを処理したいクライアントは、PsqlInfoMessageEventHandler デリゲートを作成してこのイベントに登録してください。

InfoMessage イベントでは、このイベントに関するデータを含む PsqlInfoMessageEventArgs の引数を受け取ります。

PsqlConnectionStringBuilder オブジェクト

PsqlConnectionStringBuilder プロパティ名は、PsqlConnection.ConnectionString プロパティの接続文字列オプション名と同じです。ただし、接続文字列オプション名は単語の間にスペースを入れることができます。たとえば、接続文字列オプション Min Pool Size はプロパティ名 MinPoolSize に相当します。

接続文字列の基本形式は、セミコロンで区切られた一連の「キーワード / 値」のペアを含んでいます。次に、ADO.NET データ プロバイダー用の単純な接続文字列のキーワードと値の例を示します。

```
"Server DSN=SERVERDEMO;Host=localhost"
```

接続文字列プロパティ

次の表は、Zen データ プロバイダーでサポートされる接続文字列オプションに対応するプロパティを示します。

プロパティ	説明

AlternateServers	プライマリ データベース サーバーが使用不可能な場合、データ プロバイダーが接続を試みる代替データベース サーバーの一覧を指定します。この接続文字列オプションに値を指定すると、データ プロバイダーに対して接続フェールオーバー機能が有効になります。
	この値は、代替サーバーごとに接続情報を定義する文字列の形式で指定する必要があります。デフォルトのポート値 1583 を使用しない場合は、各代替サーバーの名前または IP アドレスと、ポート番号を指定する必要があります。文字列は次のような形式です。
	`"Host=`*ホスト値*`;Port=`*ポート値*`[, ...]"`
	たとえば、Alternate Servers 値を次のように指定すると、接続フェールオーバー用に 2 つの代替サーバーが定義されます。
	`Alternate Servers="Host=AcctServer;Port=1584,`
	`Host=123.456.78.90;Port=1584"`
	接続フェールオーバーの説明やこの機能で設定できるその他の接続文字列オプションに関する情報については、**接続フェールオーバーの使用**を参照してください。

プロパティ	説明
ConnectionPoolBehavior	{LeastRecentlyUsed | MostRecentlyUsed | LeastFrequentlyUsed | MostFrequentlyUsed}。再使用するための接続プールからどのような順序で接続を削除するのかを、接続の使用頻度または使用時期に基づいて指定します。 MostRecentlyUsed に設定すると、データ プロバイダーは後入れ先出し法（LIFO）を用いて、最近プールに戻された接続を返します。 LeastRecentlyUsed（デフォルトの初期値）に設定すると、データ プロバイダーは先入れ先出し法（FIFO）を用いて、一番長く接続プールに置かれている接続を返します。この値は、プール内の接続をバランスよく使用できるようにします。 MostFrequentlyUsed に設定すると、データ プロバイダーは最も使用回数の高い接続を返します。この値は、アプリケーションが一番よく使い込んだ接続を優先的に扱うことができるようにします。 LeastFrequentlyUsed に設定すると、データ プロバイダーは最も使用回数の低い接続を返します。この値は、プール内の接続をバランスよく使用できるようにします。
ConnectionReset	{True | False}。アプリケーションで再使用するための接続プールから接続を削除する場合、その状態を接続の初期設定状態にリセットするかどうかを指定します。状態をリセットすると、パフォーマンスは低下します。これは、新しい接続では、接続時に指定した値に現在のデータベースをセットし直すなど、サーバーに対して余分なコマンドを発行する必要が生じるためです。 False（デフォルトの初期値）に設定すると、データ プロバイダーは接続の状態をリセットしません。

プロパティ	説明
ConnectionRetryCount	データ プロバイダーがプライマリ サーバーへの接続を試行し、その初期接続が失敗した後に代替サーバーへの接続を試行する回数を指定します。 値は、0 から 65535 までの任意の整数を指定できます。 0（デフォルトの初期値）に設定すると、データ プロバイダーは、初期接続の試行に失敗した後に再接続を試行しません。 接続を再試行している間に接続が成功しなかった場合、データ プロバイダーは、接続を試行した最後のサーバーで生成されたエラーを返します。 このオプションと試行間隔を指定する Connection Retry Delay オプションは、接続フェールオーバーに関して使用することができます。接続フェールオーバーの説明やこの機能で設定できるその他の接続文字列オプションに関する情報については、**接続フェールオーバーの使用**を参照してください。
ConnectionRetryDelay	初期接続の試行に失敗した後、プライマリ サーバーまたは（指定した場合には）代替サーバーへの接続を再試行するまでの試行間隔を秒数で指定します。 値は、0 から 65535 までの任意の整数を指定できます。 デフォルトの初期値は 3（秒）です。0 に設定すると、接続の再試行に待ち時間はありません。 **メモ**：このオプションは、Connection Retry Count 接続文字列オプションを 0 に設定している場合は作用しません。 このオプションと、データ プロバイダーが最初の接続の試行に失敗した後に接続を試行する回数を指定する Connection Retry Count 接続文字列オプションは、接続フェールオーバーに関して使用することができます。接続フェールオーバーの説明やこの機能で設定できるその他の接続文字列オプションに関する情報については、**接続フェールオーバーの使用**を参照してください。
ConnectionTimeout	サーバーへの接続の試行が失敗した後、接続されないでいる間の秒数を指定します。接続フェールオーバーが有効になっている場合、このオプションは接続の試行ごとに適用されます。 0 に設定すると、データ プロバイダーが接続の試行でタイムアウトすることはありません。 デフォルトの初期値は 15 秒です。

プロパティ	説明
DatabaseName	接続するデータベースの内部名を指定します。ServerDSN が定義されていない Zen データ ソースに接続する必要がある場合は、このオプションを使用します。 デフォルト値は空文字列です。 **メモ：**1 つの接続文字列に Database Name と Server DSN 接続文字列オプションを混在させることはできません。 別名 ：DBQ
DbFileDirectoryPath	**メモ：**このオプションは、Zen ADO.NET Entity Framework Core データ プロバイダーでのみサポートされます。 データベース サーバーのどのディレクトリでデータベース ファイルが作成されたかを判断します。 デフォルト値は空文字列です。
EnableIPV6	IPv4 アドレスを使用した Zen サーバーへの接続に対応する下位互換性を提供します。 True に設定すると、IPv4 アドレスまたは IPv6 アドレスのいずれかを使用するサーバーに対し、インストール済みの IPv6 プロトコルに対応したクライアントを識別させることができます。 False に設定すると、クライアントは下位互換性モードで実行します。クライアントはいつでも IPv4 アドレスを使用するサーバーに識別されます。 デフォルト値は、4.0 では True に設定されます。 IPv6 形式の詳細については、『*Getting Started with Zen*』の IPv6 を参照してください。
EnableTrace	{0 | 1}。トレースを有効にするかどうかを指定します。 0（デフォルトの初期値）に設定すると、トレースは有効になりません。
Encoding	データベースに格納されている文字列データの変換に使用する、IANA 名または Windows コード ページを指定します。 デフォルト値は空文字列で、現在の Windows Active Code Page（ACP）が使用されます。

プロパティ	説明
Encrypt	{If Needed | Always | Never}。データ プロバイダーが暗号化されたネットワーク通信（ワイヤ暗号化とも呼ばれます）を使用するかどうかを決定します。 Always に設定すると、データ プロバイダーは暗号化を使用します。サーバーでワイヤ暗号化が許可されない場合はエラーが返されます。 Never に設定すると、データ プロバイダーは暗号化を使用しません。サーバーでワイヤ暗号化を要求された場合はエラーが返されます。 IfNeeded（デフォルト）に設定すると、データ プロバイダーはサーバーのデフォルト設定を使用します。 **メモ：**このオプションは、データの暗号化と復号で必要となる追加オーバーヘッド（主に CPU 使用）のため、パフォーマンスに悪影響を与えることがあります。
Encryption	{Low | Medium | High}。データ プロバイダーが許可する暗号化の最低レベルを決定します。 デフォルトの初期値は Medium（中）です。 これらの値の意味は使用する暗号化モジュールに応じて変わります。デフォルトの暗号化モジュールでは、これらの値はそれぞれ 40 ビット、56 ビット、および 128 ビット暗号化に対応しています。
Enlist	{True | False}。スレッドの現在のトランザクション コンテキストの作成中に、データ プロバイダーが接続への参加を自動的に試行するかどうかを指定します。 **メモ：**Zen は分散トランザクションをサポートしないため、スレッドの現在のトランザクション コンテキストでの接続への参加試行は失敗します。 False に設定すると、データ プロバイダーは接続への自動的な参加試行を行いません。 True（デフォルトの初期値）に設定すると、現在のトランザクション コンテキストが存在する場合にデータ プロバイダーはエラーを返します。現在のトランザクション コンテキストが存在しない場合、データ プロバイダーは警告を発します。

プロパティ	説明
Host	接続する Zen データベース サーバーの名前または IP アドレスを指定します。たとえば、`Accountingserver` などのサーバー名を指定できます。あるいは、`199.226.22.34`（IPv4）または `1234:5678:0000:0000:0000:0000:9abc:def0`（IPv6）などの IP アドレスを指定できます。 デフォルトの初期値は空文字列です。 別名：Server、Server Name

InitialCommandTimeout	データ プロバイダーが実行の試行を終了してエラーを生成するまでのデフォルトの待機時間（秒単位のタイムアウト）を指定します。このオプションは、アプリケーションのコードに変更を加えることなく、PsqlCommand オブジェクトの CommandTimeout プロパティと同じ機能を提供します。その後、アプリケーションは CommandTimeout プロパティを使用して Initial Command Timeout 接続文字列オプションを上書きすることができます。
	デフォルトの初期値は 30 です。0 に設定すると、クエリは絶対タイムアウトしません。
	たとえば、次の C# コード例では、接続文字列はコマンドの実行試行を終了するまでに 60 秒待機するよう指示しています。アプリケーションは、次に、45 秒の CommandTimeout を指定し、接続文字列で指定した値を上書きします。
	`PsqlCommand command = new PsqlCommand();`
	`PsqlConnection conn = new PsqlConnection("…; Initial Command Timeout=60; …");`
	`conn .Open();`
	`command.Connection = connection;`
	`// command.CommandTimeout は 60 を返します`
	`command.CommandTimeout = 45;`
	`// command.CommandTimeout は 45 を返します`
	
	`command = new PsqlCommand();`
	`command.CommandTimeout = 45;`
	`command.Connection = conn;`
	`// command.CommandTimeout は、依然として 45 を返します`
	メモ：CommandTimeout オプションの初期値は、サーバーのデッドロック検出およびタイムアウトの最大値より大きい値に設定します。これによって、アプリケーションはタイムアウトした場合により意味のある応答を受け取ることができます。

プロパティ	説明
InitializationString	セッションの設定を管理するために、データベースへの接続後直ちに発行されるステートメントを指定します。 デフォルトの初期値は空文字列です。 例：NULL で埋められた CHAR 列を処理するには、次のように値を設定します。 `Initialization String=SET ANSI_PADDING ON` **メモ**：何らかの理由でステートメントが失敗した場合、Zen サーバーへの接続は失敗します。データ プロバイダーは、サーバーから返されたエラーを含む例外をスローします。
LoadBalanceTimeout	接続プールで接続を保持する秒数を指定します。接続プール マネージャーは定期的にすべてのプールをチェックし、その存続時間を超過した接続を閉じたり排除します。MinPoolSize オプションの指定によって、一部の接続でこの値を無視させることができます。接続の存続時間については、プールからの接続の削除を参照してください。 値は、0 から 65335 までの任意の整数を指定できます。 0（デフォルトの初期値）に設定すると、接続のタイムアウトは上限値になります。 別名：Connection Lifetime

プロパティ	説明
LoadBalancing	{True | False}。データ プロバイダーがプライマリ データベース サーバーや代替データベース サーバーへの接続において、クライアント ロード バランスを使用するかどうかを決定します。代替サーバーのリストは Alternate Servers 接続オプションで指定されます。 True に設定すると、データ プロバイダーはランダムな順序でデータベース サーバーに接続しようとします。ロード バランスに関する詳細については、クライアント ロード バランスの使用を参照してください。 False（デフォルトの初期値）に設定すると、クライアント ロード バランスは使用されず、データ プロバイダーはシーケンシャル（最初にプライマリ サーバー、次に代替サーバーの指定された順）に各サーバーへ接続します。 **メモ：**このオプションは、Alternate Servers 接続文字列オプションで代替サーバーが指定されていなければ作用しません。 Load Balancing 接続文字列オプションは、接続フェールオーバーに関して使用することができる任意の設定です。接続フェールオーバーの説明やこの機能で設定できるその他の接続文字列オプションに関する情報については、接続フェールオーバーの使用を参照してください。
MaxPoolSize	1 つのプール内の最大接続数を指定します。最大数に達したら、それ以上の接続を接続プールに追加することはできません。Max Pool Size Behavior 接続文字列オプションの指定によって、一部の接続でこの値を一時的に無視させることができます。 値は、1 から 65335 までの任意の整数を指定できます。 デフォルトの初期値は 100 です。

プロパティ	説明
MaxPoolSizeBehavior	{SoftCap | HardCap}。接続プール内のすべての接続が使用中である場合、データ プロバイダーは Max Pool Size オプションで指定された接続数を超えることができるかどうかを指定します。 SoftCap に設定した場合、作成される接続数は Max Pool Size に設定された値を超えることはできますが、プールされる接続数は設定値を超えません。プールの最大接続数が使用されているときに接続要求を受け取った場合、データ プロバイダーは新しい接続を作成します。アイドル状態の接続が入った満杯のプールに接続が返された場合、プール メカニズムは接続プールが Max Pool Size を決して超えないよう、破棄する接続を選択します。 HardCap（デフォルトの初期値）に設定した場合、プールに許可されている最大接続数が使用中であるときは、新しい接続要求は Connection Timeout に達するまで、接続が使用可能になるのを待ちます。
MaxStatementCacheSize	ステートメント キャッシュに保持できるステートメントの最大数を指定します。この値には 0、または 1 以上の整数を指定できます。 キャッシュ サイズを 0 に設定すると、ステートメント キャッシングを無効にします。 デフォルトの初期値は 10 です。 ほとんどの場合、ステートメント キャッシングを使用するとパフォーマンスが向上します。このオプションがパフォーマンスに与える影響については、お使いのデータ プロバイダーの「パフォーマンスに関する考慮点」トピックを参照してください。
MinPoolSize	接続プールが初期化されたときに作成される接続数、およびそのプールに保持される接続数を指定します。接続プール内の一部の接続が LoadBalanceTimeout 値を超えたとしても、接続プールにはこの最小数の接続が保持されます。 値は、0 から 65335 までの任意の整数を指定できます。 0（デフォルトの初期値）に設定した場合、接続が閉じられて接続プールに送られても、プールには、そのプールの作成に使用された元の接続のみが保持されます。 1 から 65535 までの整数に設定すると、指定された数の接続の複製がプールに保持されます。 プールがパフォーマンスに与える影響については、お使いのデータ プロバイダーの「パフォーマンスに関する考慮点」トピックを参照してください。

プロパティ	説明
ParameterMode	ネイティブ パラメーター マーカーおよびバインディングの動作を指定します。これにより、アプリケーションはプロバイダー固有の SQL コードを再利用でき、Zen ADO.NET データ プロバイダーへの移行を容易にすることができます。 ANSI（デフォルトの初期値）に設定すると、? 文字はパラメーター マーカーとして処理され、序数としてバインドされます。アプリケーションは BindByName プロパティの動作をコマンド単位で切り替えられます。 BindByOrdinal に設定した場合、ネイティブ パラメーター マーカーが使用され、ストアド プロシージャおよび標準コマンド用に序数としてバインドされます。 BindByName に設定した場合、ネイティブ パラメーター マーカーが使用され、ストアド プロシージャおよび標準コマンド用に名前でバインドされます。 **メモ：** このオプションは、Zen ADO.NET Entity Framework データ プロバイダーではサポートされません。
Password	Zen データベースへの接続に使用するパスワード（大文字小文字の区別なし）を指定します。パスワードは、データベースでセキュリティが有効な場合にのみ必要です。パスワードが必要な場合は、システム管理者からパスワードを入手してください。 別名：PWD
PersistSecurityInfo	{True | False}。セキュリティ情報を ConnectionString プロパティにクリア テキストで表示するかどうかを指定します。 True に設定すると、Password 接続文字列オプションの値はクリア テキストで表示されます。 False（デフォルトの初期値）に設定すると、データ プロバイダーは接続文字列にパスワードを表示しません。
Pooling	{True | False}。接続をプールするかどうかを指定します。接続プールに関する詳細については、接続プールの使用を参照してください。 True（デフォルトの初期値）に設定すると、接続プールが有効になります。 プールがパフォーマンスに与える影響については、お使いのデータ プロバイダーの「パフォーマンスに関する考慮点」トピックを参照してください。

プロパティ	説明
Port	Zen データベースで動作しているリスナーの TCP ポートを指定します。 デフォルトのポート番号は 1583 です。
PVTranslate	{Auto | Nothing}。クライアントが、サーバーと適合するエンコードをネゴシエイトするかどうかを指定します。 Auto に設定すると、データ プロバイダーは Encoding 接続プロパティをデータベースのコード ページに設定します。また、SQL クエリ テキストは、データ エンコードではなく UTF-8 エンコードを使用して送信されます。これにより、クエリ テキスト内の NCHAR 文字列リテラルが保持されます。 Nothing（デフォルト）に設定すると、Encoding 接続プロパティの設定が使用されます。
SchemaCollectionTimeout	試行したスキーマ コレクション操作が失敗した後、完了されないでいる間の秒数を指定します。 0 に設定すると、データ プロバイダーがスキーマ コレクション操作の試行でタイムアウトすることはありません。 デフォルトの初期値は 120 です。

プロパティ	説明
SchemaOptions	返すことができる追加のデータベース メタデータを指定します。デフォルトで、データ プロバイダーは、パフォーマンスに悪影響を与えるデータベース メタデータの一部が返らないようにすることで、パフォーマンスを最適化しています。アプリケーションでこのようなデータベース メタデータを必要とする場合は、メタデータの名前または 16 進値を指定します。 このオプションはパフォーマンスに影響することがあります。 ShowColumnDefaults または 0x04 に設定すると、列のデフォルトが返されます。 ShowParameterDefaults または 0x08 に設定すると、列のデフォルトが返されます。 FixParameterDirections または 0x10 に設定すると、プロシージャ定義が返されます。 ShowProcedureDefinitions または 0x20 に設定すると、プロシージャ定義が返されます。 ShowViewDefinitions または 0x40 に設定すると、ビュー定義が返されます。 ShowAll または 0xFFFFFFFF（デフォルトの初期値）に設定すると、すべてのデータベース メタデータが返されます。 たとえば、プロシージャ定義の説明を返すには Options=ShowProcedureDefinitions または Schema Options=0x20 を指定します。 複数の除外されたメタデータを表示するには、名前をカンマ区切りのリストで指定するか、制限したい列コレクションの 16 進値の合計を指定します。たとえば、プロシージャ定義とビュー定義（それぞれ 16 進値で 0x20 と 0x40）を返すには、`Schema Options=ShowProcedureDefinitions, ShowViewDefinitions` または `Schema Options=0x60` と指定します。 **メモ：**この接続文字列オプションはパフォーマンスに悪影響を与えることがあります。詳細については、お使いのデータ プロバイダーのパフォーマンスに関する考慮点のドキュメントを参照してください。
ServerDSN	サーバー上のデータ ソースの名前、たとえば Server DSN=SERVERDEMO を指定します。 デフォルト値は DEMODATA です。 **メモ：**1 つの接続文字列に Database Name と Server DSN 接続文字列オプションを混在させることはできません。

プロパティ	説明
StatementCacheMode	ステートメント キャッシュ モードを指定します。Statement Cache Mode はステートメントのキャッシュ動作を制御します。ステートメントは自動的にキャッシュされるか、またはコマンドが明示的にマークした場合にのみキャッシュされます。 Auto に設定すると、ステートメント キャッシングは PsqlCommand プロパティの StatementCacheBehavior に暗黙的にマークされたステートメントに対して有効になります。これらのコマンドは明示的にマークされたコマンドより低い優先順位を持ちます。つまり、ステートメント プールが最大ステートメント数に達した場合、Cache とマークされたステートメント用の余地を作るため、Implicit とマークされたステートメントが最初に削除されます。 ExplicitOnly（デフォルトの初期値）に設定すると、StatementCacheBehavior によって Cache とマークされたステートメントのみがキャッシュされます。 ほとんどの場合、ステートメント キャッシングを有効にするとパフォーマンスが向上します。このオプションが ADO.NET データ プロバイダーのパフォーマンスに与える影響については、パフォーマンスに関する考慮点を参照してください。 **メモ**：このオプションは、Zen ADO.NET Entity Framework データ プロバイダーではサポートされません。
Timestamp	{DateTime | String}。データ プロバイダーで、Zen のタイムスタンプを文字列として格納および取得するかどうかを指定します。 DateTime に設定するか、または定義しない（デフォルト）と、データ プロバイダーはタイムスタンプを .NET DateTime 型にマップします。この設定はネイティブな精度が必要な場合、たとえば、タイムスタンプを含む PsqlCommandBuilder を使用する場合に適しています。 String に設定すると、タイムスタンプは文字列として返されます。データ プロバイダーは、Zen タイムスタンプを .NET String 型にマップします。
TimeType	{DateTime | TimeSpan}。Zen の Time を ADO.NET データ プロバイダーの Timespan または DateTime として取得するかどうかを指定します。 DateTime に設定すると、データ プロバイダーは SQL の TIME 型を .NET の System.DateTime 型にマップします。 TimeSpan に設定すると、データ プロバイダーは SQL の TIME 型を .NET の System.DateTimeSpan 型にマップします。

プロパティ	説明
TraceFile	トレース ファイルのパスと名前を指定します。 デフォルトの初期値は空文字列です。指定したファイルが存在しない場合は、データ プロバイダーがファイルを作成します。
UseCurrentSchema	この接続文字列オプションはサポートされていません。これを設定すると、データ プロバイダーは例外をスローします。
UserID	Zen データベースへの接続に使用するデフォルトの Zen ユーザー名を指定します。 別名：UID

次の表に、データ プロバイダーが返されるデータから除外する列コレクションの名前と 16 進値を挙げます。複数の値を指定するには、名前をカンマ区切りのリストで指定するか、返したい列コレクションの 16 進値の合計を指定します。

名前	16 進数の値	コレクション / 列
ShowColumnDefaults[1]	0x04	Columns/COLUMN_DEFAULT
ShowParameterDefaults	0x08	ProcedureColumns//PARAMETER_DEFAULT
FixParameterDirections	0x10	ProcedureColumns/PARAMETER_TYPE
ShowProcedureDefinitions	0x20	Procedures/PROCEDURE_DEFINITION
ShowViewDefinitions	0x40	Views/VIEW_DEFINITION
ShowAll	0x7F	All

[1] COLUMN_HAS_DEFAULT は常にヌル値を返します。

PsqlConnectionStringBuilder オブジェクトにはプロバイダー固有のメソッドはありません。サポートされるメソッドの情報については、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK ドキュメントを参照してください。

PsqlCredential オブジェクト

PsqlCredential オブジェクトは、Zen サーバー認証を使用して安全にログインする方法を提供します。PsqlCredential は、Zen サーバーによって認識されるユーザー ID とパスワードで構成されます。

PsqlCredential オブジェクトのパスワードは、接続文字列とは違って SecureString 型です。接続文字列のパスワードは、プロバイダーが読み取って SecureString に変換しない限り、セキュリティで保護されていません。パスワードは、メモリに書き込まずに安全な方法で処理されます。パスワードに格納する文字列は、使用後に消去されます。

メモ： PsqlCredential は、ユーザー ID とパスワードが必要な認証方法の場合にのみ使用してください。また、Kerberos 認証またはクライアント認証を使用している場合は、PsqlCredential を使用しないでください。最後に、PsqlCredential オブジェクトを使用する場合は、接続文字列にユーザー ID とパスワードを含めないでください。

次のコード スニペットは、PsqlCredential クラスを使用する方法を示しています。この例で文字列から SecureString への変換に使用されている方法は、考えられる多くの方法のうちの 1 つです。

```
PsqlConnection con = null;
PsqlCredential lobjCredential = null;
string userId = "ABCD";
SecureString password = ConvertToSecureString("XYXYX");
private static SecureString ConvertToSecureString(string value)
{
   var securePassword = new SecureString();
   foreach (char c in value.ToCharArray())
   securePassword.AppendChar(c);
   securePassword.MakeReadOnly();
   return securePassword;
}
try
{
   lobjCredential = new PsqlCredential(userId, password);
   con = new PsqlConnection("Host=nc-xxx;Port=xxxx;Database Name=xxxx"",
      lobjCredential);
   con.Open();
   Console.WriteLine("Connection Successfully Opened...");
   con.Close();
}
catch (Exception e)
{
   Console.Write(e.Message)
}
finally
{
   if (null != con)
   {
      con.Close();
      con = null;
   }
   if (null != lobjCredential)
   {
      lobjCredential = null;
   }
}
```

次の表では、PsqlCredential オブジェクトのプロバイダー固有のパブリック プロパティの実装について説明します。

プロパティ	説明
User ID	PsqlCredential オブジェクトのユーザー ID 構成要素を返します。 String データ型を使用します。ヌルまたは空の値は無効です。
Password	PsqlCredential オブジェクトのパスワード構成要素を返します。 SecureString データ型を使用します。ヌルは無効な値です。

接続が開いているときに PsqlCredential オブジェクトを使用して、プールされている同じ接続を使用したい場合は、使用可能な接続プールから同じ接続が取得されるよう、同じ PsqlCredential オブジェクトを参照する必要があります。

接続ごとに新しい資格情報オブジェクトを作成する場合、ドライバーはオブジェクトを個別に扱い、同じユーザー ID とパスワードが使用されている場合でも、それらを異なる接続プールに入れます。

PsqlDataAdapter オブジェクト

PsqlDataAdapter オブジェクトは PsqlCommand オブジェクトを使って Zen データベースに対して SQL コマンドを実行し、取得したデータを DataSet に読み込んだり、DataSet で変更されたデータをデータベースと照合したりします。

次の表では、PsqlDataAdapter オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
UpdateBatchSize	バッチで実行可能なコマンド数を指定する値を取得または設定します。 アプリケーションで切断された DataSet を使用してこれらの DataSet を更新する場合、このプロパティの値を 1 より大きく設定することにより、積極的にパフォーマンスを操作することができます。デフォルトでは、データ プロバイダーは利用可能な最大バッチ サイズの使用を試みます。ただし、これがアプリケーションでの最適なパフォーマンスに一致するとは限りません。DataSet で通常更新する行数に基づいて値を設定します。たとえば、50 未満の行を更新する場合、このプロパティの推奨設定は 25 です。 0 に設定すると、PsqlDataAdapter はデータ ソースで利用可能な最大バッチ サイズを使用します。InsertCommand、UpdateCommand、および DeleteCommand の UpdatedRowSource プロパティは、None または OutputParameters に設定する必要があります。 1 に設定すると、バッチ更新は無効になります。 1 より大きい値に設定すると、バッチ内で指定した数のコマンドが実行されます。InsertCommand、UpdateCommand、および DeleteCommand の UpdatedRowSource プロパティは、None または OutputParameters に設定する必要があります。
DeleteCommand	Zen データ ソースからレコードを削除するための SQL ステートメントを取得または設定します。
InsertCommand	Zen データ ソースに新しいレコードを挿入するための SQL ステートメントを取得または設定します。
SelectCommand	Zen データベースのレコードを選択するための SQL ステートメントを取得または設定します。
UpdateCommand	データ ソースのレコードを更新するための SQL ステートメントを取得または設定します。

PsqlDataReader オブジェクト

PsqlDataReader オブジェクトは、データベースから読み取り専用のレコードを取得する前方スクロール カーソルです。PsqlDataAdapter を使用するよりもパフォーマンスは高くなりますが、結果セットは変更できません。

次の表では、PsqlDataReader オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
Depth	現在の行の入れ子の深さを示す値を取得します。
HasRows	この PsqlDataReader に 1 行以上の行が格納されているかどうかを示す値を取得します。
IsClosed	データ リーダーが閉じているかどうかを示す値を取得します。
RecordsAffected	SQL ステートメントの実行によって変更、挿入または削除された行数を取得します。

次の表では、PsqlDataReader オブジェクトのパブリック メソッドの一部について説明します。

メソッド	説明
Close	DataReader を閉じます。DataReader オブジェクトの使用を終了するときは必ず Close メソッドを呼び出してください。
GetSchemaTable	PsqlDataReader の列メタデータを記述する DataTable を返します。詳細については、PsqlCredential オブジェクトを参照してください。
NextResult	バッチ SQL ステートメントの結果を読み込むときに、データ リーダーを次の結果に進めます。
Read	IDataReader を次の結果に進めます。

PsqlError オブジェクト

PsqlError オブジェクトは、Zen サーバーで生成されたエラーや警告に関する情報を収集します。

次の表では、PsqlError オブジェクトでサポートされるパブリック プロパティについて説明します。

プロパティ	説明
Message	Zen サーバーから返されるエラー メッセージ テキストを取得します。
Number	Zen から返されるエラー番号を取得します。

プロパティ	説明
SQLState	Zen データ プロバイダーが例外をスローしたときに SQLState の文字列表記を取得します。例外に対応するエラーがない場合は 0 です。このプロパティは読み取り専用です。 **メモ**：SQLstate 情報が何もない ADO.NET クライアントのエラー メッセージについては、S1000 がデフォルトの SQLState として使用されます。

PsqlErrorCollection オブジェクト

PsqlErrorCollection オブジェクトは PsqlException によって作成され、Zen サーバーによって生成されたすべてのエラーを格納します。

次の表では、PsqlErrorCollection オブジェクトでサポートされるプロバイダー固有のパブリック プロパティについて説明します。サポートされるその他のプロパティおよびメソッドの情報については、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK ドキュメントを参照してください。

プロパティ	説明
Count	Zen サーバーで生成される PsqlError オブジェクトの数を取得します。

PsqlErrorCollection オブジェクトは、次の表に示すパブリック メソッドをサポートしています。

メソッド	説明
CopyTo	PsqlError オブジェクトを ErrorCollection から指定した配列にコピーします。
GetEnumerator	指定した配列の IEnumerator インターフェイスを返します。

PsqlException オブジェクト

プロバイダー固有の例外は System.Data interface インターフェイスから直接派生します。System.Exception オブジェクト全般で直接使用できるのは、Message プロパティなどのパブリック プロパティおよびメソッドのみです。SQLState および Number プロパティは、プロバイダー固有のコードを介して、または参照を使用してのみアクセスできます。

ADO.NET 2.0 は、DbException クラスに新しいプロパティ Data を導入しました。このプロパティは、例外に関する追加のユーザー定義情報を提供する、キーと値の組のコレクションを返します。ADO.NET データ プロバイダーは、SQLState、Number、および ErrorPosition などのキー / 値ペアのコレクションを取得します。

Psql.Data.SqlClient プレフィックスは各キーに適用されます。たとえば、次のようになります。

```
Psql.Data.SqlClient.Data["SQLState"] = 28000;
```

複数のエラーが存在する場合、次の表に示されるプロパティは最後に生成されたエラーに対して適用されます。複数のエラーが発生したかどうかを判断するには、このオブジェクトの Errors プロパティで返される PsqlErrorCollection の Count プロパティをチェックします。詳細については、PsqlErrorCollection オブジェクトを参照してください。

プロパティ	説明
Errors	PsqlError オブジェクトのうち 1 つの PsqlErrorCollection を取得または設定します。
Message	Zen サーバーから返されるエラー メッセージ テキストを指定します。
Number	Zen サーバーから返される数を取得または設定します。
SQLState	Zen データ プロバイダーが例外をスローしたときに SQLState の文字列表記を返します。例外に対応するエラーがない場合は 0 です。このプロパティは読み取り専用です。

PsqlFactory オブジェクト

プロバイダー ファクトリ クラスを使用すると、ユーザーは汎用オブジェクトのプログラムを作成できます。DbProviderFactory からインスタンス化すれば、ファクトリが正しいタイプの具体的なクラスを生成します。

次の表は、ADO.NET データ プロバイダーの選択や DbProviderFactory のインスタンス化を調整するために使用される静的メソッドを示しています。

メソッド	説明
CreateCommand	厳密に型指定された DbCommand インスタンスを返します。

メソッド	説明
CreateCommandBuilder	厳密に型指定された DbCommandBuilder インスタンスを返します。
CreateConnection	厳密に型指定された DbConnection インスタンスを返します。
CreateConnectionStringBuilder	厳密に型指定された DbConnectionString インスタンスを返します。
CreateDataAdapter	厳密に型指定された DbDataAdapter インスタンスを返します。
CreateDataSourceEnumerator	厳密に型指定された PsqlDataSourceEnumerator インスタンスを返します。
CreateParameter	厳密に型指定された DbParameter インスタンスを返します。

PsqlInfoMessageEventArgs オブジェクト

PsqlInfoMessageEventArgs オブジェクトは、PsqlInfoMessageEventHandler に入力として渡され、Zen サーバーで生成される警告に関する情報を格納します。

次の表では、PsqlInfoMessageEventArgs オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
Errors	Zen サーバーから送られる警告のコレクションを格納する PsqlErrorCollection を指定します。詳細については、PsqlErrorCollection オブジェクトを参照してください。
Message	Zen サーバーから返される最新のテキスト メッセージを返します。複数の警告が発生したかどうかを判断するには、アプリケーションはこのオブジェクトの Errors プロパティで返される PsqlErrorCollection の Count プロパティを調べる必要があります。

PsqlParameter オブジェクト

PsqlParameter オブジェクトは PsqlCommand オブジェクトのパラメーターを表します。

次の表では、PsqlParameter オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
ArrayBindStatus	PsqlParameterStatus エントリの配列内のすべての値を null としてバインドする必要があるかどうかを決定します。PsqlParameterStatus 列挙はエントリ NullValue を含んでいます。 このプロパティを設定しなければ、値は null になりません。配列の長さは、PsqlCommand オブジェクトの ArrayBindCount プロパティで指定された値と一致する必要があります (PsqlCommand オブジェクトを参照してください)。 デフォルトの初期値は null です。
DbType	パラメーターの DbType を取得または設定します。
Direction	パラメーターが入力専用、出力専用、双方向、あるいはストアド プロシージャの戻り値パラメーターかどうかを示す値を取得または設定します。
IsNullable	パラメーターが null 値を受け入れるかどうかを示す値を取得または設定します。
ParameterName	PsqlParameter オブジェクトの名前を設定または取得します。
Precision	Value プロパティを表す場合の最大桁数を取得または設定します。
Scale	Value プロパティを解決する小数点以下の桁数を取得または設定します。
Size	列内のデータの最大サイズをバイト単位で取得または設定します。
SourceColumn	DataSet に割り当てられており、Value プロパティの読み込みまたは返却に使用されるソース列の名前を取得または設定します。
SourceColumnNullMapping	ソース列がヌル値を許可するかどうかを示す値を設定または取得します。
SourceVersion	Value プロパティの読み込み時に使用する DataRowVersion を取得または設定します。

プロパティ	説明
Value	パラメーターの値を取得または設定します。 デフォルトの初期値は null です。 **メモ**：配列バインドが有効な場合（PsqlCommand オブジェクトの ArrayBindCount プロパティを参照してください）、このプロパティは値の配列として指定されます。各配列の長さは ArrayBindCount プロパティの値と一致させる必要があります。バイナリ型の列に配列の値を指定する場合、そのデータは実質的には byte[] として指定されます。これはバイトの配列の配列です。データ プロバイダーは複数のパラメーターにバインドするパラメーター配列を使用するような場合には、「ジャグ（Jagged）」配列を予測します。 ストアド プロシージャ パラメーターを null に設定した場合、データ プロバイダーはそのパラメーターをサーバーに送りません。代わりに、ストアド プロシージャを実行する際はパラメーターのデフォルト値が使用されます。

PsqlParameterCollection オブジェクト

PsqlParameterCollection オブジェクトは PsqlCommand に関するパラメーターのコレクションです。また、DataSet の列へのマッピングも含まれています。

次の表では、PsqlParameterCollection オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
Count	コレクション内の PsqlParameter オブジェクトの数を取得します。
IsFixedSize	PsqlParameterCollection が固定サイズかどうかを示す値を取得します。
IsReadOnly	PsqlParameterCollection が読み取り専用かどうかを示す値を取得します。
IsSynchronized	PsqlParameterCollection がスレッド セーフかどうかを示す値を取得します。
Item	指定されたインデックス位置にあるパラメーターを返します。C# の場合、このプロパティは IDataParameterCollection クラス用のインデクサーです。
SynchRoot	PsqlParameterCollection への同期アクセスに使用するオブジェクトを取得します。

次の表では、PsqlParameterCollection オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
Contains	コレクション内のパラメーターに、指定したソース テーブル名が格納されているかどうかを示す値を取得します。
IndexOf	コレクション内における IDataParameter の位置を取得します。
RemoveAt	コレクションから IDataParameter を削除します。

PsqlTrace オブジェクト

PsqlTrace オブジェクトは、開発時に問題をデバッグするためアプリケーションによって作成されます。PsqlTrace オブジェクトのプロパティを設定すると、その値を優先し環境変数の設定を無効にします。アプリケーションが最終的なものになったら、PsqlTrace オブジェクトへの参照を必ず削除してください。

次のコードでは MyTrace.txt という名前の Trace オブジェクトを作成しています。その結果、データ プロバイダーへの呼び出しはすべてそのファイルへトレースされます。

```
PsqlTrace  MyTraceObject = new PsqlTrace();
    MyTraceObject.TraceFile="C:¥¥MyTrace.txt";
    MyTraceObject.RecreateTrace = 1;
    MyTraceObject.EnableTrace = 1;
```

次の表では、PsqlTrace オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
EnableTrace	1 以上に設定すると、トレースが有効になります。 デフォルトの初期値は 0 です。トレースは無効です。
RecreateTrace	1 に設定すると、アプリケーションを開始するたびにトレース ファイルが再作成されます。 0（デフォルトの初期値）に設定すると、トレース ファイルに追加されます。
TraceFile	トレース ファイルのパスと名前を指定します。 デフォルトの初期値は空文字列です。指定したファイルが存在しない場合は、データ プロバイダーがファイルを作成します。

プロパティ	説明
メモ：EnableTrace でトレースを有効にするよう設定するとトレース処理が開始します。したがって、EnableTrace を設定する前にトレース ファイル用のプロパティ値を定義しておく必要があります。いったんトレース処理が開始されたら、TraceFile および RecreateTrace プロパティの値を変更することはできません。	

次の表では、PsqlTrace オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
DumpFootprints	1 つのデータ プロバイダーのすべてのソース ファイルのフットプリントを表示します。

PsqlTransaction オブジェクト

次の表では、PsqlTransaction オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
Connection	トランザクションに関連付けられる PsqlConnection オブジェクトを指定します。詳細については、PsqlConnection オブジェクトを参照してください。
IsolationLevel	トランザクション全体の分離レベルを定義します。値を変更すると、その新しい値が実行時に使用されます。

次の表では、PsqlTransaction オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
Commit	派生クラスで上書きされた場合、その結果生じる 1 つ以上の例外の根本原因である Exception を返します。
Rollback	トランザクションをコミットする前に、そのトランザクションで行った変更を取り消します。

Zen Common Assembly

Zen ADO.NET データ プロバイダーでは、バルク ロードなどの拡張機能を提供する追加クラスを使用できます。すべてのクラスはマネージ コードのみで作成されています。以下のクラスが Pervasive.Data.Common.dll アセンブリで提供されます。

- CsvDataReader
- CsvDataWriter
- DbBulkCopy
- DbBulkCopyColumnMapping
- DbBulkCopyColumnMapping

バルク ロードに使用されるクラスは、汎用プログラミング モデルを実装します。これらは、Zen Bulk Load をサポートする任意の DataDirect Technologies ADO.NET データ プロバイダーまたは ODBC ドライバー、およびサポートされる任意のデータベースと共に使用することができます。

CsvDataReader

CsvDataReader クラスは、Zen Bulk Load が定義した CSV ファイル形式用の DataReader 構文を提供します。

次の表では、CsvDataReader オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
BulkConfigFile	WriteToFile メソッドが呼び出されたときに作成された CSV バルク構成ファイルを指定します。バルク ロード構成ファイルは、バルク ロード データ ファイル内の列の名前とデータ型を、データをエクスポートしたテーブルや結果セットと同じ方法で定義します。バルク ロード構成ファイルは、基となる XML スキーマによってサポートされます。 パスには完全修飾パスを使用します。そうしないと、ファイルは現在の作業ディレクトリにあるものと見なされます。 **メモ：**このプロパティは Open() 呼び出しの前および Close() 呼び出しの後にのみ設定できます。それ以外の場合、InvalidOperationException がスローされます。

プロパティ	説明
BulkFile	CSV 形式のバルク データを含むバルク ロード データ ファイルを指定します。このファイル名は、バルク データの書き出し（エクスポート）および読み込み（インポート）で使用されます。指定したファイル名に拡張子が含まれない場合は、".csv" と見なされます。 パスには完全修飾パスを使用します。そうしない場合、ファイルはデフォルトで現在の作業ディレクトリにあるものと見なされます。この値が設定されていないと、InvalidOperationException がスローされます。 **メモ：**このプロパティは Open() 呼び出しの前および Close() 呼び出しの後にのみ設定できます。それ以外の場合、InvalidOperationException がスローされます。
ReadBufferSize	バルク ロードを使用してデータ ソースからデータをインポートする場合の読み取りバッファーのサイズを指定します。 デフォルトの初期値は 2048 KB です。 ゼロまたはゼロより小さい値を設定すると、System.ArgumentOutOfRangeException がスローされます。
RowOffset	バルク ロード読み取りを開始する行を指定します。RowOffset は先頭行（第 1 行）からの相対位置です。 デフォルトの初期値は 1 です。 ゼロまたはゼロより小さい値を設定すると、System.ArgumentOutOfRangeException がスローされます。 **メモ：**このプロパティは Open() 呼び出しの前および Close() 呼び出しの後にのみ設定できます。それ以外の場合、InvalidOperationException がスローされます。
SequentialAccess	パフォーマンス改善のための手法で列にアクセスするかどうかを決定します。 デフォルトの初期値は False です。 **メモ：**このプロパティは Open() 呼び出しの前および Close() 呼び出しの後にのみ設定できます。それ以外の場合、InvalidOperationException がスローされます。

次の表では、CsvDataWriter オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
Open	処理に必要なバルク ファイルのインスタンスおよび関連付けられたメタデータ ファイルを開きます。

CsvDataWriter

CsvDataWriter クラスは、Zen Bulk Load によって書き出される CSV ファイル形式の Pervasive DataWriter 構文を提供します。

詳細については、データ プロバイダーのオンライン ヘルプを参照してください。

次の表では、CsvDataWriter オブジェクトのパブリック プロパティについて説明します。

プロパティ	説明
BinaryThreshold	バルク アンロード中にバイナリ データを格納するのに、どのサイズでファイルを分けて生成するかを決定するしきい値（KB 単位）を指定します。 デフォルトの初期値は 32 です。 ゼロより小さい値を指定すると System.ArgumentOutOfRangeException がスローされます。
CharacterThreshold	バルク アンロード中に文字データを格納するのに、どのサイズでファイルを分けて生成するかを決定するしきい値（KB 単位）を指定します。 デフォルトの初期値は 64 です。 ゼロより小さい値を指定すると System.ArgumentOutOfRangeException がスローされます。
CsvCharacterSetName	サポートされる IANA コード ページの中から、使用されるコード ページ名を値で指定します。サポートされる値については、IANA コード ページ マッピングを参照してください。 アプリケーションは、PsqlConnection.DatabaseCharacterSetName プロパティを使用してデータベースに設定されている文字セットを取得できます。 認識されない CharacterSetName を使用すると、無効な文字セットが使用されたことを示す例外がスローされます。 デフォルトの初期値は UTF-16 です。 このプロパティは、CSV データ ファイルおよび追加されるオーバー フロー ファイルでこの文字セットが必ず使用されるようにします。

次の表では、CsvDataWriter オブジェクトのパブリック メソッドについて説明します。

メソッド	説明
Open	処理に必要なバルク ファイルのインスタンスおよび関連付けられたメタデータ ファイルを開きます。
WriteToFile	IDataReader の内容をバルク データ ファイルに書き出します。

DbBulkCopy

DbBulkCopy クラスは 1 つのデータ ソースから別のデータ ソースへ行をコピーするのを容易にします。

DbBulkCopy オブジェクトは、事実上の業界標準である Microsoft SqlBulkCopy クラスの定義に従っており、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

DbBulkCopyColumnMapping

DbBulkCopyColumnMapping クラスは、データ ソース テーブルから宛先テーブルへの列のマップを表します。

DbBulkCopyColumnMapping オブジェクトは、事実上の業界標準である Microsoft SqlBulkCopyColumnMapping クラスの定義に従っており、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

DbBulkCopyColumnMappingCollection

DbBulkCopyColumnMappingCollection クラスは DbBulkCopyColumnMapping オブジェクトのコレクションです。

DbBulkCopyColumnMappingCollection オブジェクトは、事実上の業界標準である Microsoft SqlBulkCopyColumnMappingCollection クラスの定義に従っており、プロバイダー固有のプロパティやメソッドを持ちません。サポートされるプロパティおよびメソッドについては、データ プロバイダーのオンライン ヘルプおよび Microsoft .NET Framework SDK のドキュメントを参照してください。

B. スキーマ情報の入手

アプリケーションは、データベースのメタデータを検索して返すよう、データ プロバイダーに要求することができます。各データ プロバイダー固有のスキーマ コレクションは、テーブルや列などのデータベース スキーマ要素を公開します。データ プロバイダーは Connection クラスの GetSchema メソッドを使用します。スキーマ情報は GetSchemaTable メソッドによって返される列で説明されているように、結果セットからも取得できます。

データ プロバイダーにもプロバイダー固有のスキーマ コレクションが含まれています。スキーマ コレクション名 MetaDataCollections を使用すると、サポートされるスキーマ コレクションの一覧と、それらがサポートする制限の数を返すことができます。

GetSchemaTable メソッドによって返される列

PsqlDataReader が開いている間は、結果セットからスキーマ情報を取得することができます。PsqlDataReader.GetSchemaTable() で生成された結果セットは、次の表に示される列を、示されている順序で返します。

列	説明
ColumnName	列の名前。列名は一意でない場合もあります。判断できない場合は、null 値が返されます。この名前は、現在のビュー内またはコマンド テキスト内の列に対する最新の名前変更を常に反映した名前になります。
ColumnOrdinal	列の序数。この列に null 値を含めることはできません。行のブックマーク列（ある場合）はゼロになります。その他の列は 1 から順に番号が付けられます。
ColumnSize	列の値に許容される最大長。固定長データ型を使用する列では、そのデータ型のサイズになります。
NumericPrecision	列の精度。これは ProviderType での列の定義方法によって決まります。 ProviderType が数値データ型の場合、これは列の最大精度になります。 ProviderType が数値データ型以外の場合は、null 値になります。
NumericScale	小数点の右側の桁数。これは、ProviderType が DBTYPE_DECIMAL または DBTYPE_NUMERIC の場合です。それ以外の場合は、null 値になります。 この値は、ProviderType での列の定義方法によって決まります。
DataType	列を .NET Framework の型にマップします。
ProviderType	列のデータ型のインジケーター。この列に null 値を含めることはできません。 列のデータ型が行によって異なる場合は、Object である必要があります。
IsLong	非常に長いデータを格納する BLOB が列に含まれている場合に設定されます。このフラグの設定は、データ型の、PROVIDER_TYPES 行セット内の IS_LONG 列の値に対応します。 非常に長いデータの定義は、プロバイダーによって異なります。
AllowDBNull	コンシューマーが列に null 値を設定できる場合や、コンシューマーが列に null 値を設定できるかどうかをデータ プロバイダーが判断できない場合に、設定されます。それ以外の場合は、設定されません。 列を null 値に設定できない場合でも、null 値が含まれている可能性があります。

列	説明
IsReadOnly	列を変更できるかどうかを決定します。 列が変更できない場合は true、それ以外の場合は false です。
IsRowVersion	書き込み禁止で、行の識別以外に意味のない値を持つ永続的な行 ID が列に格納されている場合に、設定されます。
IsUnique	列自体がキーを構成しているか、あるいは、その列だけに適用される UNIQUE 型の制約があるかどうかを示します。 true に設定した場合、ベース テーブル（BaseTableName で返されるテーブル）の行は、この列に同じ値を格納できません。 false（デフォルトの初期値）に設定した場合は、ベース テーブルでこの列に重複する値を格納できます。
IsKey	列のセットが行セットの 1 行を一意に識別するかどうかを指定します。この列のセットは、ベース テーブルの主キー、一意の制約、または重複のないインデックスから生成することができます。 列が、行セット内の列のセットのうちの 1 つで、これらを基に行を一意に識別する場合は true です。列が行を一意に識別する必要がない場合は false です。
IsAutoIncrement	新しい行に固定インクリメントで値を割り当てるかどうかを指定します。 VARIANT_TRUE に設定すると、列は新しい行に固定インクリメントで値を割り当てます。 VARIANT_FALSE（デフォルトの初期値）に設定すると、列は新しい行に固定インクリメントで値を割り当てません。
BaseSchemaName	列を格納している、データベース内のスキーマの名前。基本スキーマ名を判断できない場合は null 値です。 デフォルトの初期値は null です。
BaseCatalogName	列を格納している、データ ストア内のカタログの名前。基本カタログ名を判断できない場合は null 値が使用されます。 デフォルトの初期値は null です。
BaseTableName	列を格納している、データ ストア内のテーブルまたはビューの名前。基本テーブル名を判断できない場合は null 値が使用されます。 デフォルトの初期値は null です。

列	説明
BaseColumnName	データ ストア内での列名。エイリアスが使用された場合は、ColumnName 列に返される列名とは異なることがあります。基本列名を判断できない場合や、データベース内の列から派生した行セット列であるのにそれと一致しない場合は、null 値が使用されます。 デフォルトの初期値は null です。
IsAliased	列の名前がエイリアスであるかどうかを指定します。列名がエイリアスであれば、値 true が返されます。そうでなければ、false が返されます。
IsExpression	列の名前が式であるかどうかを指定します。列が式であれば、値 true が返されます。そうでなければ、false が返されます。
IsIdentity	列の名前が ID 列であるかどうかを指定します。列が ID 列であれば、値 true が返されます。そうでなければ、false が返されます。
IsHidden	列の名前が非表示であるかどうかを指定します。列が表示されない場合は、値 true が返されます。そうでなければ、false が返されます。

GetSchema メソッドによるスキーマ メタデータの取得

アプリケーションでデータ プロバイダーおよびデータ ソースに関するスキーマ メタデータを取得する場合は、Connection オブジェクトの GetSchema メソッドを使用を使用します。各プロバイダーには 5 つの標準メタデータ コレクションも含め、多数のスキーマ コレクションが実装されています。

- MetaDataCollections スキーマ コレクション
- DataSourceInformation スキーマ コレクション
- DataTypes コレクション
- ReservedWords コレクション
- Restrictions コレクション

追加のコレクションを指定したら、データ プロバイダーからスキーマ情報を返す機能がサポートされるようにしておく必要があります。

データ プロバイダーでサポートされるその他のコレクションの詳細については、Additional スキーマ コレクションを参照してください。

メモ： 各 ColumnName で必要なデータ型など、さらなるバックグラウンド機能要件については、.NET Framework ドキュメントを参照してください。

MetaDataCollections スキーマ コレクション

MetaDataCollections スキーマ コレクションは、ログインしているユーザーが利用できるスキーマ コレクションの一覧です。MetaDataCollection は、次の表に示されているサポートされる列を任意の順序で返すことができます。

列名	説明
CollectionName	コレクションを返すために GetSchema メソッドに渡すコレクションの名前。
NumberOfRestrictions	コレクションに指定されている可能性のある制限の数。
NumberOfIdentifierParts	複合識別子 / データベース オブジェクト名の構成要素の数。

DataSourceInformation スキーマ コレクション

DataSourceInformation スキーマ コレクションは、次の表に示されているサポートされる列を任意の順序で返すことができます。1 行のみ返されることに注意してください。

列名	説明
CompositeIdentifierSeparatorPattern	複合識別子内の複合識別子に一致する正規表現。
DataSourceProductName	データ プロバイダーがアクセスする製品の名前。
DataSourceProductVersion	データ プロバイダーがアクセスする製品のバージョンをデータ ソースのネイティブ形式で示します。
DataSourceProductVersionNormalized	データ ソースの標準化されたバージョン。これにより、String.Compare() を使ってバージョンを比較できるようになります。
GroupByBehavior	GROUP BY 句内の列と、選択リスト内の集計されない列の間の関係を指定します。
Host	データ プロバイダーが接続しているホスト。
IdentifierCase	引用符で囲まれていない識別子の大文字小文字を区別して処理するかどうかを示します。
IdentifierPattern	識別子に一致する正規表現で、識別子の適合値を持ちます。
OrderByColumnsInSelect	ORDER BY 句内の列は選択リスト内にある必要があるかどうかを指定します。true の値は、当該列が選択リストに存在しなければならないことを示し、false の値は選択リストに存在しなくてもよいことを示します。
ParameterMarkerFormat	パラメーターの書式設定方法を示す書式文字列。
ParameterMarkerPattern	パラメーター マーカーに一致する正規表現。もしあれば、パラメーター名の適合値を持ちます。
ParameterNameMaxLength	パラメーター名の最大長（文字単位）。
ParameterNamePattern	有効なパラメーター名に一致する正規表現。
QuotedIdentifierCase	引用符で囲まれた識別子の大文字小文字を区別して処理するかどうかを示します。
QuotedIdentifierPattern	引用符で囲まれた識別子に一致する正規表現で、引用符の付いていない識別子自体の適合値を持ちます。

列名	説明
StatementSeparatorPattern	ステートメント区切り文字に一致する正規表現。
StringLiteralPattern	文字列リテラルに一致する正規表現で、リテラル自体の適合値を持ちます。
SupportedJoinOperators	データ ソースでサポートされる、SQL 結合ステートメントの種類を指定します。

DataTypes コレクション

次の表では、DataTypes スキーマ コレクションでサポートされる列について説明します。列は任意の順序で取得できます。

列名	説明
ColumnSize	数値以外の列またはパラメーターの長さは、この型の最大長もしくは、データ プロバイダーがこの型に対して定義している長さを参照します。
CreateFormat	CREATE TABLE など、この列をデータ定義ステートメントに追加する方法を示す書式文字列。
CreateParameters	このデータ型の列を作成するときに指定する必要のある作成パラメーター。各作成パラメーターは文字列で、それらを提供する順にカンマで区切って列挙します。 たとえば、SQL データ型の DECIMAL には精度と小数点以下の桁数が必要です。この場合は、文字列 `"精度, 小数位"` を作成パラメーターに含める必要があります。 精度 10、小数位 2 の DECIMAL 列を作成するテキスト コマンドでは、CreateFormat 列の値を `"DECIMAL({0},{1})"` とすることができます。完全な型指定は `DECIMAL(10,2)` となります。
DataType	データ型に対する .NET Framework 型の名前。
IsAutoIncrementable	データ型の値を自動インクリメントするかどうかを指定します。 true：このデータ型の値は自動インクリメントできます。 false：このデータ型の値は自動インクリメントできません。

列名	説明
IsBestMatch	このデータ型は、データ ストア内のすべてのデータ型と、DataType 列の値によって示される .NET Framework データ型の間で最適なものであるかどうかを指定します。 true：最適なデータ型です。 false：最適なデータ型ではありません。
IsCaseSensitive	データ型は文字型で、大文字小文字を区別するかどうかを指定します。 true：文字型で、大文字小文字を区別します。 false：文字型でないか、もしくは大文字小文字を区別しません。
IsConcurrencyType	true：このデータ型は行が変更されるたびにデータベースによって更新され、列の値は以前のあらゆる値と異なるものになります。 false：このデータ型は行が変更されるたびにデータベースによって更新されません。
IsFixedLength	true：データ定義言語（DDL）によって作成されるこのデータ型の列は固定長になります。 false：DDL によって作成されるこのデータ型の列は可変長になります。
IsFixedPrecisionScale	true：データ型は固定の精度と小数点以下の桁数を持ちます。 false：データ型は固定の精度と小数点以下の桁数を持ちません。
IsLiteralsSupported	true：データ型をリテラルで表すことができます。 false：データ型をリテラルで表すことはできません。
IsLong	true：データ型は非常に長いデータを格納します。非常に長いデータの定義は、プロバイダーによって異なります。 false：データ型は非常に長いデータを格納しません。
IsNullable	true：データ型はヌル値を許可します。 false：データ型はヌル値を許可しません。
IsSearchable	true：データ型は非常に長いデータを格納します。非常に長いデータの定義は、プロバイダーによって異なります。 false：データ型は非常に長いデータを格納しません。
IsSearchableWithLike	true：LIKE 述部で使用できません。 false：LIKE 述部で使用できません。

列名	説明
IsUnisgned	true：データ型は符号なしです。 false：データ型は符号付きです。
LiteralPrefix	指定されたリテラルに適用するプレフィックス。
LiteralSuffix	指定されたリテラルに適用するサフィックス。
MaximumScale	型インジケーターが数値型の場合は、これは小数点の右側に割り当てられる最大桁数になります。 それ以外の場合、これは `DBNull.Value` になります。
MinimumScale	型インジケーターが数値型の場合は、これは小数点の右側に割り当てられる最小桁数になります。 それ以外の場合、これは `DBNull.Value` になります。
ProviderDbType	パラメーターの型を指定する場合に使用する必要のあるプロバイダー固有の型の値。
TypeName	プロバイダー固有のデータ型名。

ReservedWords コレクション

このスキーマ コレクションは、データ プロバイダーが接続するデータベースで予約されている語句についての情報を公開します。次の表では、データ プロバイダーがサポートする列について説明します。

列名	説明
Reserved Word	プロバイダー固有の予約語。

Restrictions コレクション

Restrictions スキーマ コレクションは、現在データベースに接続しているデータ プロバイダーでサポートされる制限に関する情報を公開します。次の表では、データ プロバイダーから返される列について説明します。列は任意の順序で取得できます。

ADO.NET データ プロバイダーは標準化された名前を制限に使用します。データ プロバイダーが Schema メソッドの制限をサポートしている場合は、制限に対して常に同じ名前を使用します。

制限値の大文字小文字の区別は基になるデータベースによって決定されますが、DataSourceInformation コレクションの IdentifierCase 値および QuotedIdentifierCase 値によっても決定することができます（DataSourceInformation スキーマ コレクションを参照してください）。

列名	説明
CollectionName	指定した制限を適用するコレクションの名前。
RestrictionName	コレクション内の制限の名前。
RestrictionDefault	無視されます。
RestrictionNumber	この制限の、コレクション Restrictions 内における実際の場所。
IsRequired	制限が必要かどうかを指定します。

サポートされる追加のスキーマ コレクションそれぞれに適用する制限については、Additional スキーマ コレクションを参照してください。

Additional スキーマ コレクション

Zen ADO.NET データ プロバイダーは、以下に示す追加のスキーマ コレクションをサポートしています。

- Columns スキーマ コレクション
- ForeignKeys スキーマ コレクション
- Indexes スキーマ コレクション
- PrimaryKeys スキーマ コレクション
- ProcedureParameters スキーマ コレクション
- Procedures スキーマ コレクション
- TablePrivileges スキーマ コレクション
- Tables スキーマ コレクション
- Views スキーマ コレクション

Columns スキーマ コレクション

説明：Columns スキーマ コレクションは、指定したユーザーがアクセスできるカタログに定義されているテーブル（ビューも含む）の列を識別します。次の表は、指定したユーザーがアクセスできるカタログに定義されているテーブルの列を示します。

制限の数：3

使用可能な制限：TABLE_CATALOG、TABLE_NAME、COLUMN_NAME

ソート順：TABLE_CATALOG、TABLE_NAME、ORDINAL_POSITION

列名	.NET Framework データ型[1]	説明
CHARACTER_MAXIMUM_LENGTH	Int32	列の値に許容される最大長。文字、バイナリ、またはビットの列の場合、これは次のいずれかになります。 • いずれかの列が定義されている場合は、それぞれ文字単位、バイト単位、ビット単位の列の最大長。 • 列の長さが定義されていない場合は、それぞれ文字単位、バイト単位、ビット単位のデータ型の最大長。 • 列にもデータ型にも最大長が定義されていない場合、あるいは列が文字、バイナリ、ビットのいずれでもない場合は、ゼロ（0）。
CHARACTER_OCTET_LENGTH	Int32	列の型が文字またはバイナリの場合は、8進数（バイト単位）で表す列の最大長。 値ゼロ（0）は、その列が最大長を持たないか、あるいは、文字またはバイナリの列でないことを意味します。
COLUMN_DEFAULT	String	列のデフォルト値。
COLUMN_HASDEFAULT	Boolean	TRUE：列にはデフォルト値があります。 FALSE：列にデフォルト値がないか、もしくはデフォルト値があるかどうかが不明です。
COLUMN_NAME	String	列の名前。これは一意でない可能性があります。
DATA_TYPE	Object	列のデータ型のインジケーター。 この値をヌルにすることはできません。
IS_NULLABLE	Boolean	TRUE：列はヌル値を許可する可能性があります。 FALSE：列がヌル値を許可するかどうかわかりません。

列名	.NET Framework データ型[1]	説明
NATIVE_DATA_TYPE	String	型のデータ ソース記述。 この値をヌルにすることはできません。
NUMERIC_PRECISION	Int32	列のデータ型が数値データのものである場合、これは列の最大精度（有効桁数）になります。
NUMERIC_PRECISION_RADIX	Int32	どの数を基礎として NUMERIC_PRECISION および NUMERIC_SCALE で値を表現するかを示す基数。2 または 10 を返す場合にのみ有効です。
NUMERIC_SCALE	Int16	列の型が小数位を持つ数値型の場合、これは小数点より右側の桁数です。
ORDINAL_POSITION	Int32	列の序数。列には 1 から始まる番号が付けられています。
PROVIDER_DEFINED_TYPE	Int32	列のデータ ソース定義の型は、データ プロバイダーの型の列挙（たとえば PsqlDbType 列挙）にマップされています。 この値をヌルにすることはできません。
PROVIDER_GENERIC_TYPE	Int32	列のプロバイダー定義の型は System.Data.DbType 列挙にマップされています。 この値をヌルにすることはできません。
TABLE_CATALOG	String	データベース名。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

ForeignKeys スキーマ コレクション

説明：ForeignKeys スキーマ コレクションは、指定したユーザーによってカテゴリ内に定義された外部キー列を識別します。

制限の数：2

使用可能な制限：FK_TABLE_CATALOG、PK_TABLE_NAME

ソート順：FK_TABLE_CATALOG、FK_TABLE_NAME

列名	.NET Framework データ型[1]	説明
DEFERRABILITY	String	外部キーを延期できるか。次のいずれかの値になります。 • INITIALLY DEFERRED • INITIALLY IMMEDIATE • NOT DEFERRABLE
DELETE_RULE	String	削除規則が指定された場合は、次のいずれかの値になります。 CASCADE：CASCADE の参照操作が指定されました。 SET NULL：SET NULL の参照操作が指定されました。 SET DEFAULT：SET DEFAULT の参照操作が指定されました。 NO ACTION：NO ACTION の参照操作が指定されました。
FK_COLUMN_NAME	String	外部キー列の名前。
FK_NAME	String	外部キー名。この制限は必須です。
FK_TABLE_CATALOG	String	外部キーテーブルが定義されているカタログ名。
FK_TABLE_NAME	String	外部キー テーブル名。この制限は必須です。
FK_TABLE_OWNER	String	外部キー テーブルの所有者。この制限は必須です。
ORDINAL	Int32	キーにおける列名の順序。たとえば、あるテーブルに別のテーブルへの外部キー参照がいくつか含まれているとします。序数は参照ごとに最初から始まります。3 列のキーへの 2 つの参照ならば、1, 2, 3, 1, 2, 3 を返します。

列名	.NET Framework データ型[1]	説明
PK_COLUMN_NAME	String	主キー列の名前。
PK_NAME	String	主キー名。
PK_TABLE_CATALOG	String	主キー テーブルが定義されているカタログ名。
PK_TABLE_NAME	String	主キー テーブル名。
PK_TABLE_OWNER	String	主キー テーブルの所有者。この制限は必須です。
UPDATE_RULE	String	更新規則が指定された場合は、次のいずれかの値になります。 CASCADE：CASCADE の参照操作が指定されました。 SET NULL：SET NULL の参照操作が指定されました。 SET DEFAULT：SET DEFAULT の参照操作が指定されました。 NO ACTION：NO ACTION の参照操作が指定されました。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

Indexes スキーマ コレクション

説明：Indexes スキーマ コレクションは、指定したユーザーが所有しているカテゴリ内に定義されたインデックスを識別します。

制限の数：2

使用可能な制限：TABLE_CATALOG、TABLE_NAME

ソート順：UNIQUE、TYPE、INDEX、CATALOG、INDEX_NAME、ORDINAL_POSITION

列名	.NET Framework データ型[1]	説明
CARDINALITY	Int32	インデックスに含まれる一意な値の数。
COLLATION	String	次のいずれかの値になります。 ASC：列の並べ替え順は昇順です。 DESC：列の並べ替え順は降順です。
COLUMN_NAME	String	列名。
FILTER_CONDITION	String	フィルター制限を識別する WHERE 句。
INDEX_CATALOG	String	カタログ名。
INDEX_NAME	String	インデックス名。
ORDINAL_POSITION	Int32	インデックスにおける列の位置を表す、1 から始まる序数。
PAGES	Int32	インデックスの格納に使用されるページ数。
TABLE_CATALOG	String	カタログ名。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。
TABLE_QUALIFIER	String	テーブル修飾子。
TYPE	String	インデックスの種類。次のいずれかの値です。 • BTREE：インデックスは B+- ツリーです。 • HASH：インデックスは、直線的なハッシュや伸縮可能なハッシュを用いたハッシュ ファイルです。 • CONTENT：インデックスはコンテンツ インデックスです。 • OTHER：インデックスは上記の種類以外のインデックスです。
UNIQUE	Boolean	

1. クラスはすべて System.XXX です。たとえば、System.String となります。

PrimaryKeys スキーマ コレクション

説明：PrimaryKeys スキーマ コレクションは、指定したユーザーによってカテゴリ内に定義された主キー列を識別します。

制限の数：2

使用可能な制限：TABLE_CATALOG、TABLE_NAME

ソート順：TABLE_CATALOG、TABLE_NAME

列名	.NET Framework データ型[1]	説明
COLUMN_NAME	String	主キー列の名前。
ORDINAL	Int32	キーにおける列名の順序。
PK_NAME	String	主キー名。
TABLE_CATALOG	String	テーブルが定義されているデータベース名。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

ProcedureParameters スキーマ コレクション

説明：ProcedureParameters スキーマ コレクションは、Procedures コレクションの一部であるプロシージャのパラメーターおよびリターン コードに関する情報を返します。

制限の数：3

使用可能な制限：PROCEDURE_CATALOG、PROCEDURE_NAME、PARAMETER_NAME

ソート順：PROCEDURE_CATALOG、PROCEDURE_NAME、ORDINAL_POSITION

列名	.NET Framework データ型[1]	説明
CHARACTER_MAXIMUM_LENGTH	Int32	パラメーターの最大長。
CHARACTER_OCTET_LENGTH	Int32	パラメーターの型が文字またはバイナリの場合は、8 進数（バイト単位）で表すパラメーターの最大長。 パラメーターが最大長を持たない場合、値はゼロ（0）になります。 上記の型以外のパラメーターでは、値は -1 になります。
DATA_TYPE	Object	列のデータ型のインジケーター。 この値をヌルにすることはできません。
DESCRIPTION	String	パラメーターの記述。たとえば、新しい従業員を追加するプロシージャの Name パラメーターの記述を Employee name としてもかまいません。
IS_NULLABLE	Boolean	TRUE：パラメーターはヌル値を許可する可能性があります。 FALSE：パラメーターはヌル値を許可しません。
NATIVE_DATA_TYPE	String	型のデータ ソース記述。 この値をヌルにすることはできません。
NULLABLE	String	パラメーターにヌル値を指定できるかどうかを示します。指定可能な 2 つの値は、YES と NO です。
NUMERIC_PRECISION	Int32	列のデータ型が数値である場合、これは列の最大精度（有効桁数）になります。 列のデータ型が数値でない場合、これは `DbNull` になります。

列名	.NET Framework データ型[1]	説明
NUMERIC_PRECISION_RADIX	Int32	列のデータ型が数値である場合に適用可能です。 どの数を基礎として NUMERIC_PRECISION および NUMERIC_SCALE で値を表現するかを示す基数。2 または 10 を返す場合にのみ有効です。
NUMERIC_SCALE	Int16	列のデータ型が小数位を持つ数値型の場合、これは小数点より右側の桁数です。 それ以外の場合、これは `DbNull` になります。
ORDINAL_POSITION	Int32	パラメーターが入力、入出力、または出力パラメーターである場合、これはプロシージャ呼び出しにおけるパラメーターの位置を表す、1 から始まる序数です。 パラメーターが戻り値の場合、これは `DbNull` になります。
PARAMETER_DEFAULT	String	パラメーターのデフォルト値。 デフォルト値が NULL の場合、PARAMETER_HASDEFAULT 列は TRUE を返し、PARAMETER_DEFAULT 列は存在しなくなります。 PARAMETER_HASDEFAULT に FALSE を設定すると、PARAMETER_DEFAULT 列が存在しなくなります。
PARAMETER_HASDEFAULT	Boolean	TRUE：パラメーターにはデフォルト値があります。 FALSE：パラメーターにデフォルト値がないか、もしくはデフォルト値があるかどうかが不明です。

列名	.NET Framework データ型[1]	説明
PARAMETER_NAME	String	パラメーター名。パラメーターに名前が付いていない場合、これは `DbNull` になります。
PARAMETER_TYPE	String	次のいずれかの値になります。 INPUT：パラメーターは入力パラメーターです。 INPUTOUTPUT：パラメーターは入出力パラメーターです。 OUTPUT：パラメーターは出力パラメーターです。 RETURNVALUE：パラメーターはプロシージャの戻り値です。 UNKNOWN：データ プロバイダーには不明なパラメーターの種類です。
PROCEDURE_CATALOG	String	カタログ名。
PROCEDURE_NAME	String	プロシージャ名。
PROCEDURE_COLUMN_NAME	String	プロシージャ列名。
PROVIDER_DEFINED_TYPE	Int32	列のデータ ソース定義の型は、データ プロバイダーの型の列挙（たとえば PSQLDbType 列挙）にマップされています。 この値をヌルにすることはできません。
PROVIDER_GENERIC_TYPE	Int32	列のデータ ソース定義の型は System.Data.DbType 列挙にマップされています。 この値をヌルにすることはできません。
SQL_DATETIME_SUB	Object	列のデータ型が DateTime である場合に適用可能です。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

Procedures スキーマ コレクション

説明：Procedures スキーマ コレクションは、カタログに定義されているプロシージャを識別します。可能であれば、接続ユーザーが EXECUTE 権限を持っているプロシージャのみを返すようにしてください。

制限の数：2

使用可能な制限：PROCEDURE_CATALOG、PROCEDURE_NAME、PROCEDURE_TYPE

ソート順：PROCEDURE_CATALOG、PROCEDURE_NAME

列名	.NET Framework データ型[1]	説明
PROCEDURE_CATALOG	String	データベース名。
PROCEDURE_NAME	String	プロシージャ名。
PROCEDURE_OWNER	String	プロシージャ所有者。
PROCEDURE_TYPE	String	次のいずれかの値になります。 UNKNOWN：値が返されるかどうかわかりません。 PROCEDURE：プロシージャ。値は返されません。 FUNCTION：関数。値が返されます。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

TablePrivileges スキーマ コレクション

説明：TablePrivileges スキーマ コレクションは、カタログに定義されているテーブルについて、指定したユーザーが利用できる、つまり許可されている権限を識別します。

制限の数：3

使用可能な制限：TABLE_CATALOG、TABLE_NAME、GRANTEE

ソート順：TABLE_CATALOG、TABLE_NAME、PRIVILEGE_TYPE

列名	型インジケーター[1]	説明
GRANTEE	String	権限が付与されているユーザー名（または PUBLIC）。
PRIVILEGE_TYPE	String	権限のタイプ。次のいずれかのタイプになります。 • DELETE • INSERT • REFERENCES • SELECT • UPDATE
TABLE_CATALOG	String	テーブルが定義されているデータベースの名前。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

Tables スキーマ コレクション

説明：Tables スキーマ コレクションは、指定したユーザーがアクセスできるカタログに定義されているテーブル（ビューも含む）を識別します。

制限の数：3

使用可能な制限：TABLE_CATALOG、TABLE_NAME、TABLE_TYPE

ソート順：TABLE_TYPE、TABLE_CATALOG、TABLE_NAME

列名	.NET Framework データ型[1]	説明
DESCRIPTION	String	テーブルの記述。 列に関連付けられている記述がない場合、データ プロバイダーは `DbNull` を返します。

列名	.NET Framework データ型[1]	説明
TABLE_CATALOG	String	テーブルが定義されているデータベースの名前。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。
TABLE_TYPE	String	テーブルの種類。次のいずれかです。 • ALIAS • GLOBAL TEMPORARY • LOCAL TEMPORARY • SYNONYM • SYSTEM TABLE • SYSTEM VIEW • TABLE • VIEW この列に空文字列を含めることはできません。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

Views スキーマ コレクション

説明：Views スキーマ コレクションは、指定したユーザーがアクセスできるカタログに定義されているビューを識別します。

制限の数：2

使用可能な制限：TABLE_CATALOG、TABLE_NAME

ソート順：TABLE_CATALOG、TABLE_NAME

列名	型インジケーター[1]	説明
TABLE_CATALOG	String	テーブルが定義されているデータベースの名前。
TABLE_NAME	String	テーブル名。
TABLE_OWNER	String	テーブル所有者。

列名	型インジケーター[1]	説明
TABLE_QUALIFIER	String	テーブル修飾子。
VIEW _DEFINITION	String	ビュー定義。これはクエリ式です。

1. クラスはすべて System.XXX です。たとえば、System.String となります。

C. .NET の SQL エスケープ シーケンス

通常、外部結合やスカラー関数の呼び出しなどの言語機能の多くは、データベース管理システムによって実装されます。これらの機能の標準的な構文が定義されていても、DBMS 特有の構文が使われる場合がよくあります。.NET では、次の言語機能の標準的な構文を含む、エスケープ シーケンスをサポートします。

- 日付、時刻およびタイムスタンプのリテラル
- 数値、文字列およびデータ型変換関数などのスカラー関数
- 外部結合

.NET で使用するエスケープ シーケンスは次のとおりです。

```
{extension}
```

このエスケープ シーケンスは、ADO.NET データ プロバイダーによって認識および解析されます。データ プロバイダーはこのエスケープ シーケンスをデータ ストア固有の文法で置き換えます。

日付、時刻、タイムスタンプのエスケープ シーケンス

日付、時刻、およびタイムスタンプ リテラルのエスケープ シーケンスは次のとおりです。

```
{literal-type ' 値 '}
```

literal-type は、以下に示す型のうちの 1 つです。

リテラルの型	説明	値の形式
d	日付	`yyyy-mm-dd`
t	時刻	`hh:mm:ss [1]`
ts	Timestamp	`yyyy-mm-dd hh:mm:ss[.f...]`

メモ：Visual Studio で、テーブルの Date フィールドにデータを挿入するクエリを実行しているときにエラーが発生した場合は、システムの日付形式が `yyyy-mm-dd` に設定されていることを確認してください。形式が違う場合は、`yyyy-mm-dd` に変更します。

例

```
UPDATE Orders SET OpenDate={d '1997-01-29'}
WHERE OrderID=1023
```

スカラー関数

SQL 文で使用できるスカラー関数の構文は、次のとおりです。

```
{fn scalar-function}
```

scalar-function は、ADO.NET データ プロバイダーでサポートされるスカラー関数です。

例

```
SELECT {fn UCASE(NAME)} FROM EMP
```

サポートされるスカラー関数を次の表に示します。

文字列 関数	数値関数	日付時刻関数	システム関数
ASCII	ABS	CURDATE	DATABASE
BIT_LENGTH	ACOS	CURRENT_DATE	USER
CHAR	ASIN	CURTIME	
CHAR_LENGTH	ATAN	CURRENT_TIME	
CHARACTER_LENGTH	ATAN2	CURRENT_TIMESTAMP	
CONCAT	CEILING	DAYNAME	
LCASE または LOWER	COS	DAYOFMONTH	
LEFT	COT	DAYOFYEAR	
LENGTH	DEGREES	EXTRACT	
LOCATE	EXP	HOUR	
LTRIM	FLOOR	MINUTE	
OCTET_LENGTH	LOG	MONTH	
POSITION	LOG10	MONTHNAME	
REPLACE	MOD	NOW	
REPLICATE	PI	QUARTER	
RIGHT	POWER	SECOND	
RTRIM	RADIANS	TIMESTAMPADD	
SPACE	RAND	TIMESTAMPDIFF	
STUFF	ROUND	WEEK	
SUBSTRING	SIGN	YEAR	
UCASE または UPPER	SIN		
	SQRT		
	TAN		
	TRUNCATE		

外部結合のエスケープ シーケンス

.NET では SQL92 の左外部結合、右外部結合、および完全外部結合の構文をサポートしています。外部結合用のエスケープ シーケンスは次のとおりです。

```
{oj outer-join}
```

この *outer-join* には次のような構文が入ります。

```
table-reference {LEFT | RIGHT | FULL} OUTER JOIN
```

```
{table-reference | outer-join} ON search-condition
```

各項目の説明は次のとおりです。

table-reference はテーブル名で、*search-condition* はテーブルを結合するのに使用する条件です。

例

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
  FROM {oj Customers LEFT OUTER JOIN
        Orders ON Customers.CustID=Orders.CustID}
  WHERE Orders.Status='OPEN'
```

ADO.NET データ プロバイダーは、以下の外部結合エスケープ シーケンスを Zen 9.x 以上と同様にサポートします。

- 左外部結合
- 右外部結合
- 完全外部結合

D. ロック レベルと分離レベル

さまざまなデータベース システムによって、多様なロック レベルおよび分離レベルをサポートします。以下のトピックでは、ロック レベルおよび分離レベルについて、またこれらの設定が、取得するデータにどのように影響するかについて説明します。

- ロック
- 分離レベル
- ロック モードとレベル

ロック

ロックとは、ユーザーがデータベースのテーブルやレコードにアクセスするのを制限する処理です。複数のユーザーが同じテーブルやレコードに同時にアクセスする可能性がある場合に、ロックをかけます。テーブルやレコードをロックすると、一度に 1 人のユーザーだけがデータに影響する処理を行えるようになります。

ロック制御は、複数のユーザーが、データベースの同じレコードに同時にアクセスしたり変更する場合に特に重要です。複数のユーザーが同時に使えるデータベースは便利ですが、問題が発生することもあります。たとえば、ロックしていない場合、2 人のユーザーが同時に同じレコードを変更しようとすると、正確なデータが取得できなかったり、一方のユーザーが必要とするデータをもう一方のユーザーが削除してしまう可能性があります。しかし、いったん 1 人のユーザーがアクセスしたレコードを、他のユーザーが一時的に変更できないようにロックできるようにしておくと、このような問題は発生しません。ロックすることによって、データベースへの同時アクセスによる問題の発生を最小限に抑えることができます。

分離レベル

分離レベルとは、データの一貫性を高めるためにデータベース システムで採用される特別なロック方法です。分離レベルが高いほど、そのロック方法も複雑になります。データベースで使用される分離レベルによって、データの一貫性に関連する次の事象がトランザクションで発生するかどうかが決まります。

Dirty reads（ダーティー リード）	ユーザー 1 が行を変更します。ユーザー 1 がコミットする前に、ユーザー 2 が同じ行を読み取ります。ユーザー 1 がロールバックします。ユーザー 2 は、実際にはデータベースに存在しない行を読み取ったことになります。これによって、ユーザー 2 は誤ったデータに対して判断を下すことになるかもしれません。
Non-repeatable reads（反復不可能な読み取り）	ユーザー 1 が行を読み取りましたがコミットしません。ユーザー 2 がその同じ行を修正または削除し、それをコミットします。ユーザー 1 がもう一度同じ行を読み取ったときは、その行が変更または削除されています。
Phantom reads（ファントム リード）	ユーザー 1 は、検索条件を使用して複数行のセットを読み取りましたがコミットはしません。ユーザー 2 がこの検索条件を満たす行を挿入し、コミットします。ユーザー 1 が検索条件を使ってもう一度読み取ったときには、前になかった行が存在することになります。

分離レベルとは、データベース システムで、上記の事象が発生するのを防げるかどうかを示すレベルです。ANSI（American National Standards Institute：米国規格協会）によって、次の 4 つの分離レベルが定められています

- Read uncommitted (0) （コミットされていない読み取り）
- Read committed (1) （コミットされた読み取り）
- Repeatable read (2) （反復可能な読み取り）
- Serializable (3) （直列化可能）

0 から 3 へとレベルが高くなるほど、トランザクションのデータの一貫性も高くなります。一番低いレベルでは、上の 3 つの事象がすべて発生する可能性があります。一番高いレベルでは、どの事象も発生しません。このような事象が発生するのを防げるかどうかは、各レベルで使う、次のようなロック方法によって決まります。

Read uncommitted (0)（コミットされていない読み取り）	データベースを変更するときにロックがかかり、トランザクションの終わり（EOT）まで維持されます。データベースの読み取りには、ロックがかかりません。

Read committed (1)（コミットされた読み取り）	データベースの読み取りと変更時にロックがかかります。読み取り後にロックは解除されますが、変更されたオブジェクトのロックは EOT まで維持されます。
Repeatable read (2)（反復可能な読み取り）	データベースの読み取りと変更時にロックがかかります。変更されたすべてのオブジェクトのロックは EOT まで維持されます。データの読み取りロックは、EOT まで維持されます。変更不可能なアクセス構造（インデックスおよびハッシュ構造など）のロックは読み取り後に解除されます。
Serializable (3)（直列化可能）	DataSet の影響を受ける行に、EOT までロックがかかります。変更されたアクセス構造、およびクエリで使われた構造がすべて EOT までロックされます。

次の表では各分離レベルにおいて発生するデータ一貫性の事象を示します。

レベル	Dirty Read	Nonrepeatable Read	Phantom Read
0, Read uncommitted	可	可	可
1, Read committed	不可	可	可
2, Repeatable read	不可	不可	可
3, Serializable	不可	不可	不可

分離レベルが高いほどデータの一貫性は高くなりますが、逆に、個々のユーザーの**同時処理性**は低くなります。同時処理とは、複数のユーザーが同時にデータにアクセスして変更することです。分離レベルが上がるに従って、使用されるロック制御が原因で、同時処理上の問題が発生する確率が高くなります。

検討事項：分離レベルが高くなるほどロック制御が厳しくなり、ユーザーが、別のユーザーによってかけられたロックが解除されるのを待つ時間が長くなります。分離レベルと同時処理性には、このような逆比例的な関係があるので、分離レベルを選択する前にユーザーがデータベースをどのように使用するか検討しておく必要があります。データの一貫性と同時処理性のどちらがより重要かを考慮して、使用する分離レベルを決めてください。

ロック モードとレベル

データベース システムによって使用するロック モードは異なりますが、通常、「共有ロック」と「排他ロック」という 2 種類の基本モードがあります。共有ロックでは、複数のユーザーが単一オブジェクトを参照することが可能です。1 人のユーザーがレコードに共有ロックをかけると、2 番目に同じレコードにアクセスしたユーザーも共有ロックをかけることができます。ただし、2 番目のユーザーは、そのレコードに排他ロックをかけることはできません。排他ロックとは、そのロックを取得したユーザーへの独占的な権利です。1 人のユーザーがレコードに排他ロックをかけると、2 番目のユーザーは、同じレコードにどの種類のロックもかけられなくなります。

データベース システムで使用するロック レベルは、パフォーマンスと同時処理性にも影響します。ロック レベルによって、データベース内でロックされるオブジェクトのサイズが決まります。たとえば、データベース システムの多くで、テーブル全体、および個々のレコードをロックすることができます。また、ページ レベルのロック（ロック レベルは中程度）もよく使われます。1 ページには 1 つまたは複数のレコードが入っていますが、これは通常、1 回のディスク アクセスでディスクから読み取られるデータの量です。ページ レベルのロックの大きな欠点は、1 人のユーザーがレコードをロックすると、同じページに保存されているほかのレコードを 2 番目のユーザーがロックできないことです。

E. パフォーマンスの最適化を図る .NET アプリケーションの設計

パフォーマンス重視の .NET アプリケーションを開発することは容易ではありません。Zen ADO.NET データ プロバイダーは、コードの実行速度が非常に遅くても、それを伝える例外をスローしません。

データの取得

データを効率よく取得するには、必要なデータのみを返す最も効率のよい方法を選択します。ここでは、.NET アプリケーションでデータを取得するときに、システムのパフォーマンスを最適化する方法を説明します。

長いデータの取得

ネットワーク経由で長いデータを取得するのには時間がかかり、リソースも消費するので、特に必要な場合以外は、アプリケーションから長いデータを要求しないようにします。

ユーザーが長いデータを必要とすることはほとんどありません。ユーザーがこのような結果項目の確認を要求する場合は、選択リストに長いデータの列だけを指定して、アプリケーションからもう一度データベースを照会します。このようにすると、平均的なユーザーは、ネットワーク トラフィックのパフォーマンスにさほど影響を与えずに結果セットを取得することができます。

最良の方法は、長いデータを選択リストに入れないことですが、アプリケーションによっては、Zen ADO.NET データ プロバイダーにクエリを送信する前に選択リストを編成しないものもあります（一部のアプリケーションでは、`select * from <テーブル名> ...` などの構文を使用します）。長いデータが選択リストに入っていると、アプリケーションが長いデータを結果セットにバインドしていなくても、フェッチ時に長いデータを取得する必要があるデータ プロバイダーもあります。できれば、テーブルの一部の列のみを取得する方法を試してください。

場合によっては長いデータを取得しなければならないことがあります。このような場合でも、一般に 100 KB を超えるような大量のテキストを画面に表示させるのは望ましくありません。

取得するデータのサイズの縮小

ネットワーク トラフィックを減らしてパフォーマンスを向上させるために、最大行数や最大フィールド サイズの設定を呼び出したり、行サイズやフィールド サイズを制限するほかのデータベース固有のコマンドを呼び出したりして、取得するデータのサイズを扱いやすいサイズまで下げることができます。取得するデータのサイズを下げるもう 1 つの方法は、列のサイズを小さくすることです。データ プロバイダーでパケット サイズを定義できる場合は、必要な最小パケット サイズを指定します。

また、必要な行のみが返されるようにしてください。たとえば、2 列しか必要でないのに 5 列を返すようにしていた場合、不要な列に長いデータが入っていると、パフォーマンスが低下します。

CommandBuilder オブジェクトの使用

CommandBuilder オブジェクトは SQL ステートメントを生成するには便利だと思われがちです。しかし、これを使用するとパフォーマンスに悪影響を及ぼす恐れがあります。同時処理の制限が原因で、CommandBuilder オブジェクトは効率のよい SQL ステートメントを生成することができません。たとえば、次の SQL ステートメントは Command Builder で作成されたものです。

```
CommandText: UPDATE TEST01.EMP SET EMPNO = ?, ENAME = ?, JOB = ?, MGR = ?, HIREDATE = ?, SAL = ?, COMM
= ?, DEPT = ?
```

```
WHERE
  ( (EMPNO = ?)AND ((ENAME IS NULL AND ? IS NULL)
  OR (ENAME = ?))AND ((JOB IS NULL AND ? IS NULL)
  OR (JOB = ?))AND ((MGR IS NULL AND ? IS NULL)
  OR (MGR = ?))AND ((HIREDATE IS NULL AND ? IS NULL)
  OR (HIREDATE = ?))AND ((SAL IS NULL AND ? IS NULL)
  OR (SAL = ?))AND ((COMM IS NULL AND ? IS NULL)
  OR (COMM = ?))AND ((DEPT IS NULL AND ? IS NULL)
  OR (DEPT = ?)))
```

多くの場合、Command Builder で生成される Update ステートメントや Delete ステートメントよりも、エンド ユーザーの方が効率のよいステートメントを記述できます。

もう 1 つの問題は、CommandBuilder オブジェクトの設計にあります。CommandBuilder オブジェクトは常に DataAdapter オブジェクトと関連付けられ、DataAdapter オブジェクトが生成する RowUpdating イベントと RowUpdated イベントのリスナーとして CommandBuilder オブジェクト自身を登録します。したがって、行を更新するたびに、この 2 つのイベントが処理されなければなりません。

正しいデータ型の選択

データ型によっては、取得や送信に時間がかかるものがあります。スキーマを設計するときには、最も効率よく処理できるデータ型を選択してください。たとえば、整数データは浮動小数点データより速く処理できます。浮動小数点データは、内部データベース固有の形式に基づいて定義され、通常、圧縮形式になっています。このようなデータは、ワイヤ プロトコルで処理できるように、解凍して別の形式に変換する必要があります。

処理時間が最も短いデータ型は文字列で、その次が整数です。整数の場合は、通常、何らかの変換またはバイトの並べ替えが必要です。浮動小数点データとタイムスタンプの処理には、整数の少なくとも 2 倍の時間が必要です。

.NET オブジェクトとメソッドの選択

ここでは、.NET オブジェクトとメソッドを選択して使用するときに、システムのパフォーマンスを最適化する方法を説明します。

ストアド プロシージャの引数としてのパラメーター マーカーの使用

ストアド プロシージャを呼び出す場合は、リテラル引数ではなく、常に、引数マーカーのパラメーター マーカーを使用します。

Command オブジェクトの CommandText プロパティにストアド プロシージャ名を設定する場合、そのリテラル引数を CommandText へ物理的にコーディングしないでください。たとえば、次のようなリテラル引数は使いません。

```
{call expense (3567, 'John', 987.32)}
```

Zen ADO.NET データ プロバイダーでは、データベース サーバーのストアド プロシージャを呼び出すことができますが、その際プロシージャをその他の SQL クエリとして実行します。ストアド プロシージャを SQL クエリとして実行すると、データベース サーバーがステートメントを解析し、引数の型を検証し、引数を正しいデータ型に変換します。

次の例で、アプリケーション プログラマは getCustName の引数を整数 12345 であると見なすでしょう。

```
{call getCustName (12345)}
```

しかし、SQL は常に文字列としてデータベース サーバーに送信されます。データベース サーバーが SQL クエリを解析し、引数値を分離しても結果はまだ文字列です。データベース サーバーで文字列 "12345" を整数値 12345 に変換する必要があります。パラメーター マーカーを使用することで文字列変換の必要がなくなるので、データベース サーバーでの処理量を減らすことができます。

```
{call getCustName (?)}
```

.NET アプリケーションの設計

ここでは、.NET アプリケーションを設計するときに、システム パフォーマンスを最適化する方法を説明します。

接続の管理

アプリケーションのパフォーマンスを向上させるには、接続を適切に管理することが重要です。接続を複数回行う代わりに、1 回の接続で複数のステートメント オブジェクトを使用することで、アプリケーションのパフォーマンスを最適化します。初期接続の確立後は、データソースへの接続は行わないようにします。

接続プールを使用すると、特に、ネットワークまたは World Wide Web を介して接続するアプリケーションのパフォーマンスを大幅に向上させることができます。また、接続プールによって接続の再利用が可能になります。接続を閉じても、データベースとの物理的接続を閉じるわけではありません。アプリケーションが接続を要求すると、アクティブな接続が再利用されるので、新しい接続の作成に必要なネットワークへの I/O は発生しません。

接続は、あらかじめ割り当てておきます。まず、どの接続文字列が必要かを確認します。1 つの固有な接続文字列で、新しい接続プールが 1 つ作成されます。

一度作成された接続プールは、アクティブなプロセスが終了するか、Connection Lifetime に指定された時間が過ぎるまで破棄されません。アクティブでないプールや空のプールを維持するのに必要なシステム オーバーヘッドは、ごくわずかです。

接続とステートメントの処理は、実装前に決めておく必要があります。接続方法を慎重に管理することで、アプリケーションのパフォーマンスが向上し、メンテナンスも簡単になります。

接続の開閉

接続は、それが必要になる直前に開いてください。必要になるより早く接続を開くと、ほかのユーザーが使用できる接続の数が減り、リソースの需要が増える可能性があります。

使用可能なリソースを保持するには、接続が必要でなくなったらすぐに接続を明示的に閉じます。有効範囲外になった接続がガベージ コレクターによって暗黙的にクリーンアップされるのを待つ場合は、接続は直ちに接続プールに戻されず、実際には使用されていないリソースに関連付けられます。

finally ブロックの内側で接続を閉じます。finally ブロック内のコードは、例外が発生した場合でも必ず実行されます。これにより、接続が明示的に閉じられることが保証されます。たとえば、次のように指定します。

```
try
{
   DBConn.Open();
… // 必要な作業を行います
}
catch (Exception ex)
{
   // 例外を処理します
}
finally
{
   //  接続を閉じます
   if (DBConn != null)
      DBConn.Close();
}
```

接続プールを使用している場合には、接続の開閉は不経済な操作ではありません。データ プロバイダーの Connection オブジェクトの Close() メソッドを使用すると、接続は接続プールに追加されるか戻されます。ただし、自動的に接続を閉じると、その接続に関連付けられているすべての DataReader オブジェクトが閉じられることを忘れないでください。

ステートメント キャッシングの使用

ステートメント キャッシュは、プリペアド ステートメントのグループまたは Command オブジェクトのインスタンスで、アプリケーションによって再使用が可能です。ステートメント キャッシュを使用するとアプリケーションのパフォーマンスを向上させることができます。これは、プリペアド ステートメントの動作が、そのステートメントがアプリケーションの存続期間中に何度再使用されたとしても、1 度だけ実行されるためです。

ステートメント キャッシュは物理接続に属します。実行された後、プリペアド ステートメントはステートメント キャッシュに置かれ、接続が閉じられるまで保持されます。

アプリケーションが使用する全プリペアド ステートメントをキャッシュすれば、パフォーマンスが向上するように思われます。しかし、この手法では、接続プールを使ってステートメント キャッシングを実装した場合、データベースのメモリに負担をかける結果になります。この場合、プールされた各接続がステートメント キャッシュを持ち、アプリケーションで使用される全プリペアド ステートメントを各自のキャッシュに含むことになります。これらのプールされたプリペアド ステートメントは、すべてデータベースのメモリにも保持されます。

コマンドの複数回使用

Command.Prepare メソッドを使用するかどうかによって、クエリ実行のパフォーマンスは良くも悪くも大きな影響を受けます。Command.Prepare メソッドは、基となるデータ プロバイダーに対し、パラメーター マーカーを使用するステートメントの複数回実行を最適化するように指示します。実行メソッド（ExecuteReader、ExecuteNonQuery、または ExecuteScalar）が使用されているかどうかにかかわらず、あらゆるコマンドの準備が可能であることに留意してください。

Zen ADO.NET データ プロバイダーが、プリペアド ステートメントを含んでいるストアド プロシージャをサーバー上で作成することにより、Command.Prepare を実装する場合を考えてみましょう。ストアド プロシージャの作成には多くのオーバーヘッドを要しますが、ステートメントは複数回実行することができます。ストアド プロシージャの作成はパフォーマンスに悪影響を与えますが、プロシージャの作成時にクエリが解析され、最適化パスが保管されるため、作成したステートメントの実行は最小化されます。同じステートメントを複数回実行するアプリケーションは、Command.Prepare を呼び出してからそのコマンドを複数回実行することで、大きなメリットを得ることができます。

しかし、1 回しか実行しないステートメントに対して Command.Prepare を使用すると、不要なオーバーヘッドが生じる結果になります。さらに、大きな単一の実行クエリ バッチに対して Command.Prepare を使用するアプリケーションではパフォーマンスが低下します。同様に、常に Command.Prepare を使用するか、まったく Command.Prepare を使用しないアプリケーションは、プリペアド ステートメントとアンプリペアド ステートメントを論理的に組み合わせて使用するアプリケーションとは同じように機能しません。

ネイティブの管理プロバイダーの使用

アンマネージ コード、つまり .NET 環境外のコードへのブリッジは、パフォーマンスを低下させます。マネージ コードからアンマネージ コードを呼び出すと、データ プロバイダーの実行速度はマネージ コードのみのデータ プロバイダーよりも著しく遅くなります。このようなパフォーマンスが大きく落ちる方法は、極力避けます。

ブリッジを使用する場合は、そのブリッジのためのコードを書くことになります。後でデータベース固有の Zen ADO.NET データ プロバイダーが使用可能になったとき、このコードを書き直す必要があります。つまり、オブジェクト名、スキーマ情報、エラー処理、およびパラメーターを書き直さなければなりません。ブリッジ用ではなく管理デー

タ プロバイダー用にコード化することによって、貴重な時間とリソースを節約できます。

データの更新

ここでは、データベースのデータを更新するときのシステム パフォーマンスを最適化する方法を説明します。

切断された DataSet の使用

結果セットのサイズが、なるべく小さくなるようにします。サーバーから結果セットをすべて取得してから、DataSet を埋める必要があります。結果セット全体をクライアントのメモリに保存します。

データソースへの変更の同期

データ ソースへの変更の同期を取るには、次の例で示すように主キーを使用して、PsqlDataAdapter にロジックを作成する必要があります。

```
string updateSQL As String = "UPDATE emp SET sal = ?, job = ?"+
     " = WHERE empno = ?";
```

F. .edmx ファイルの使用

.edmx ファイルは XML ファイルで、エンティティ データ モデル（EDM：Entity Data Model）の定義と、ターゲット データベースのスキーマの記述、そして、その EDM とデータベース間のマッピングの定義を行います。.edmx ファイルには、ADO.NET Entity Data Model デザイナー（エンティティ デザイナー）で使用される、モデルを視覚的に表示するための情報も含まれています。

以下のコード例では、EDM レイヤーに Extended Entity Framework の機能を提供するために、.edmx ファイルに加える必要のある変更を示します。

Entity Framework には、ADO.NET と類似した一連のメソッドが含まれています。これらのメソッドは、LINQ、EntitySQL、および ObjectServices などの新しい Entity Framework コンシューマーで利用できるように適応させてあります。

ADO.NET Entity Framework データ プロバイダーは、PsqlStatus および PsqlConnectionStatistics エンティティを公開することによって、EDM のこの機能をモデル化します。これにより、Visual Studio の標準ツールを使用して、この機能をモデル化することが可能となります。

コード例

次のコードは、SSDL モデルの一例です。

```
<!--
SSDL content
-->
<edmx:StorageModels>
  <Schema Namespace="DDTek.Store" Alias="Self" Provider="DDTek.Oracle" ProviderManifestToken="11g"
    xmlns:store="https://schemas.microsoft.com/ado/2007/12/edm/ EntityStoreSchemaGenerator"
    xmlns= "https://schemas.microsoft.com/ado/2006/04/edm/ssdl">
    <EntityContainer Name="DDTek_Connection">
      <EntitySet Name="Connection_Statistics" EntityType="DDTek.Store.Connection_Statistics" />
      <EntitySet Name="Status" EntityType="DDTek.Store.Status" />
    </EntityContainer>
    <Function Name="RetrieveStatistics" Aggregate="false" BuiltIn="false" NiladicFunction="false"
IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
StoreFunctionName=""DDTek_Connection_RetrieveStatistics"" />
    <Function Name="EnableStatistics" Aggregate="false" BuiltIn="false" NiladicFunction="false"
IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
StoreFunctionName=""DDTek_Connection_EnableStatistics"" />
    <Function Name="DisableStatistics" Aggregate="false" BuiltIn="false" NiladicFunction="false"
IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
StoreFunctionName=""DDTek_Connection_DisableStatistics"" />
    <Function Name="ResetStatistics" Aggregate="false" BuiltIn="false" NiladicFunction="false"
IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
StoreFunctionName=""DDTek_Connection_ResetStatistics"" />
    <!--
    <Function Name="Reauthenticate" Aggregate="false" BuiltIn="false" NiladicFunction="false"
IsComposable="false" ParameterTypeSemantics="AllowImplicitConversion"
StoreFunctionName=""DDTek_Connection_Reauthenticate""> -->
    <!-- <Parameter Name="CurrentUser" Type="varchar2" Mode="In" /> -->
    <!-- <Parameter Name="CurrentPassword" Type="varchar2" Mode="In" /> -->
    <!-- <Parameter Name="CurrentUserAffinityTimeout" Type="number" Precision="10" Mode="In" /> --></
Function>
      -->
    <EntityType Name="Connection_Statistics">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="SocketReadTime" Type="binary_double" Nullable="false" />
      <Property Name="MaxSocketReadTime" Type="binary_double" Nullable="false" />
      <Property Name="SocketReads" Type="number" Precision="20" Nullable="false" />
      <Property Name="BytesReceived" Type="number" Precision="20" Nullable="false" />
      <Property Name="MaxBytesPerSocketRead" Type="number" Precision="20" Nullable="false" />
      <Property Name="SocketWriteTime" Type="binary_double" Nullable="false" />
      <Property Name="MaxSocketWriteTime" Type="binary_double" Nullable="false" />
      <Property Name="SocketWrites" Type="number" Precision="20" Nullable="false" />
      <Property Name="BytesSent" Type="number" Precision="20" Nullable="false" />
      <Property Name="MaxBytesPerSocketWrite" Type="number" Precision="20" Nullable="false" />
      <Property Name="TimeToDisposeOfUnreadRows" Type="binary_double" Nullable="false" />
     <Property Name="SocketReadsToDisposeUnreadRows" Type="number" Precision="20" Nullable="false" /
>
      <Property Name="BytesRecvToDisposeUnreadRows" Type="number" Precision="20" Nullable="false" />
      <Property Name="IDUCount" Type="number" Precision="20" Nullable="false" />
      <Property Name="SelectCount" Type="number" Precision="20" Nullable="false" />
      <Property Name="StoredProcedureCount" Type="number" Precision="20" Nullable="false" />
      <Property Name="DDLCount" Type="number" Precision="20" Nullable="false" />
      <Property Name="PacketsReceived" Type="number" Precision="20" Nullable="false" />
      <Property Name="PacketsSent" Type="number" Precision="20" Nullable="false" />
      <Property Name="ServerRoundTrips" Type="number" Precision="20" Nullable="false" />
      <Property Name="SelectRowsRead" Type="number" Precision="20" Nullable="false" />
      <Property Name="StatementCacheHits" Type="number" Precision="20" Nullable="false" />
```

```
      <Property Name="StatementCacheMisses" Type="number" Precision="20" Nullable="false" />
      <Property Name="StatementCacheReplaces" Type="number" Precision="20" Nullable="false" />
      <Property Name="StatementCacheTopHit1" Type="number" Precision="20" Nullable="false" />
      <Property Name="StatementCacheTopHit2" Type="number" Precision="20" Nullable="false" />
      <Property Name="StatementCacheTopHit3" Type="number" Precision="20" Nullable="false" />
      <Property Name="PacketsReceivedPerSocketRead" Type="binary_double" Nullable="false" />
      <Property Name="BytesReceivedPerSocketRead" Type="binary_double" Nullable="false" />
      <Property Name="PacketsSentPerSocketWrite" Type="binary_double" Nullable="false" />
      <Property Name="BytesSentPerSocketWrite" Type="binary_double" Nullable="false" />
      <Property Name="PacketsSentPerRoundTrip" Type="binary_double" Nullable="false" />
      <Property Name="PacketsReceivedPerRoundTrip" Type="binary_double" Nullable="false" />
      <Property Name="BytesSentPerRoundTrip" Type="binary_double" Nullable="false" />
      <Property Name="BytesReceivedPerRoundTrip" Type="binary_double" Nullable="false" />
<!--
Oracle specific
-->
      <Property Name="PartialPacketShifts" Type="number" Precision="20" Nullable="false" />
      <Property Name="PartialPacketShiftBytes" Type="number" Precision="20" Nullable="false" />
      <Property Name="MaxReplyBytes" Type="number" Precision="20" Nullable="false" />
      <Property Name="MaxReplyPacketChainCount" Type="number" Precision="20" Nullable="false" />
      <Property Name="Id" Type="number" Precision="10" Nullable="false" />
    </EntityType>
    <EntityType Name="Status">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="ServerVersion" Type="varchar2" Nullable="false" />
      <Property Name="Host" Type="varchar2" Nullable="false" />
      <Property Name="Port" Type="number" Precision="10" Nullable="false" />
      <Property Name="SID" Type="varchar2" Nullable="false" />
      <!-- <Property Name="CurrentUser" Type="varchar2" Nullable="false" /> -->
     <!-- <Property Name="CurrentUserAffinityTimeout" Type="number" Precision="10" Nullable="false"
/> -->
      <!-- <Property Name="SessionId" Type="number" Precision="10" Nullable="false" /> -->
      <Property Name="StatisticsEnabled" Type="number" Precision="1" Nullable="false" />
      <Property Name="Id" Type="number" Precision="10" Nullable="false" />
    </EntityType>
  </Schema>
</edmx:StorageModels>
```

モデルをさらに分解して、概念レイヤーで CSDL モデルを確立します。このレイヤーは EDM に公開されています。

```
<edmx:ConceptualModels>
  <Schema Namespace="DDTek" Alias="Self"
    xmlns="https://schemas.microsoft.com/ado/ 2006/04/edm">
    <EntityContainer Name="DDTekConnectionContext">
      <EntitySet Name="DDTekConnectionStatistics" EntityType="DDTek.DDTekConnectionStatistics" />
      <EntitySet Name="DDTekStatus" EntityType="DDTek.DDTekStatus" />
      <FunctionImport Name="RetrieveStatistics" EntitySet="DDTekConnectionStatistics" ReturnType=
"Collection(DDTek.DDTekConnectionStatistics)" />
      <FunctionImport Name="EnableStatistics" EntitySet="DDTekStatus"
ReturnType="Collection(DDTek.DDTekStatus)" />
      <FunctionImport Name="DisableStatistics" EntitySet="DDTekStatus" ReturnType=
"Collection(DDTek.DDTekStatus)" />
      <FunctionImport Name="ResetStatistics" EntitySet="DDTekStatus"
ReturnType="Collection(DDTek.DDTekStatus)" />
      <FunctionImport Name="Reauthenticate" EntitySet="DDTekStatus" ReturnType=
"Collection(DDTek.DDTekStatus)">
        <Parameter Name="CurrentUser" Type="String" />
        <Parameter Name="CurrentPassword" Type="String" />
        <Parameter Name="CurrentUserAffinityTimeout" Type="Int32" />
      </FunctionImport>
```

```
    </EntityContainer>
    <EntityType Name="DDTekConnectionStatistics">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="SocketReadTime" Type="Double" Nullable="false" />
      <Property Name="MaxSocketReadTime" Type="Double" Nullable="false" />
      <Property Name="SocketReads" Type="Int64" Nullable="false" />
      <Property Name="BytesReceived" Type="Int64" Nullable="false" />
      <Property Name="MaxBytesPerSocketRead" Type="Int64" Nullable="false" />
      <Property Name="SocketWriteTime" Type="Double" Nullable="false" />
      <Property Name="MaxSocketWriteTime" Type="Double" Nullable="false" />
      <Property Name="SocketWrites" Type="Int64" Nullable="false" />
      <Property Name="BytesSent" Type="Int64" Nullable="false" />
      <Property Name="MaxBytesPerSocketWrite" Type="Int64" Nullable="false" />
      <Property Name="TimeToDisposeOfUnreadRows" Type="Double" Nullable="false" />
      <Property Name="SocketReadsToDisposeUnreadRows" Type="Int64" Nullable="false" />
      <Property Name="BytesRecvToDisposeUnreadRows" Type="Int64" Nullable="false" />
      <Property Name="IDUCount" Type="Int64" Nullable="false" />
      <Property Name="SelectCount" Type="Int64" Nullable="false" />
      <Property Name="StoredProcedureCount" Type="Int64" Nullable="false" />
      <Property Name="DDLCount" Type="Int64" Nullable="false" />
      <Property Name="PacketsReceived" Type="Int64" Nullable="false" />
      <Property Name="PacketsSent" Type="Int64" Nullable="false" />
      <Property Name="ServerRoundTrips" Type="Int64" Nullable="false" />
      <Property Name="SelectRowsRead" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheHits" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheMisses" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheReplaces" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheTopHit1" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheTopHit2" Type="Int64" Nullable="false" />
      <Property Name="StatementCacheTopHit3" Type="Int64" Nullable="false" />
      <Property Name="PacketsReceivedPerSocketRead" Type="Double" Nullable="false" />
      <Property Name="BytesReceivedPerSocketRead" Type="Double" Nullable="false" />
      <Property Name="PacketsSentPerSocketWrite" Type="Double" Nullable="false" />
      <Property Name="BytesSentPerSocketWrite" Type="Double" Nullable="false" />
      <Property Name="PacketsSentPerRoundTrip" Type="Double" Nullable="false" />
      <Property Name="PacketsReceivedPerRoundTrip" Type="Double" Nullable="false" />
      <Property Name="BytesSentPerRoundTrip" Type="Double" Nullable="false" />
      <Property Name="BytesReceivedPerRoundTrip" Type="Double" Nullable="false" />
      <Property Name="PartialPacketShifts" Type="Int64" Nullable="false" />
      <Property Name="PartialPacketShiftBytes" Type="Int64" Nullable="false" />
      <Property Name="MaxReplyBytes" Type="Int64" Nullable="false" />
      <Property Name="MaxReplyPacketChainCount" Type="Int64" Nullable="false" />
      <Property Name="Id" Type="Int32" Nullable="false" />
    </EntityType>
    <EntityType Name="DDTekStatus">
      <Key>
        <PropertyRef Name="Id" />
      </Key>
      <Property Name="ServerVersion" Type="String" Nullable="false" />
      <Property Name="Host" Type="String" Nullable="false" />
      <Property Name="Port" Type="Int32" Nullable="false" />
      <Property Name="SID" Type="String" Nullable="false" />
      <Property Name="CurrentUser" Type="String" Nullable="false" />
      <Property Name="CurrentUserAffinityTimeout" Type="Int32" Nullable="false" />
      <Property Name="SessionId" Type="Int32" Nullable="false" />
      <Property Name="StatisticsEnabled" Type="Boolean" Nullable="false" />
      <Property Name="Id" Type="Int32" Nullable="false" />
    </EntityType>
  </Schema>
</edmx:ConceptualModels>
```

次の単純なマッピングは、各部分を結び付けます。

```
<!--
C-S mapping content
-->
<edmx:Mappings>
  <Mapping Space="C-S"
    xmlns="urn:schemas-microsoft- com:windows:storage:mapping:CS">
    <EntityContainerMapping StorageEntityContainer="DDTek_Connection"
CdmEntityContainer="DDTekConnectionContext">
      <EntitySetMapping Name="DDTekConnectionStatistics">
        <EntityTypeMapping TypeName="DDTek.DDTekConnectionStatistics">
          <MappingFragment StoreEntitySet="Connection_Statistics">
<!--
StoreEntitySet="Connection_Statistics" TypeName="DDTek.DDTekConnectionStatistics">
-->
            <ScalarProperty Name="SocketReadTime" ColumnName="SocketReadTime" />
            <ScalarProperty Name="MaxSocketReadTime" ColumnName="MaxSocketReadTime" />
            <ScalarProperty Name="SocketReads" ColumnName="SocketReads" />
            <ScalarProperty Name="BytesReceived" ColumnName="BytesReceived" />
            <ScalarProperty Name="MaxBytesPerSocketRead" ColumnName="MaxBytesPerSocketRead" />
            <ScalarProperty Name="SocketWriteTime" ColumnName="SocketWriteTime" />
            <ScalarProperty Name="MaxSocketWriteTime" ColumnName="MaxSocketWriteTime" />
            <ScalarProperty Name="SocketWrites" ColumnName="SocketWrites" />
            <ScalarProperty Name="BytesSent" ColumnName="BytesSent" />
            <ScalarProperty Name="MaxBytesPerSocketWrite" ColumnName="MaxBytesPerSocketWrite" />
           <ScalarProperty Name="TimeToDisposeOfUnreadRows" ColumnName="TimeToDisposeOfUnreadRows" /
>
            <ScalarProperty Name="SocketReadsToDisposeUnreadRows" ColumnName=
"SocketReadsToDisposeUnreadRows" />
            <ScalarProperty Name="BytesRecvToDisposeUnreadRows"
ColumnName="BytesRecvToDisposeUnreadRows" />
            <ScalarProperty Name="IDUCount" ColumnName="IDUCount" />
            <ScalarProperty Name="SelectCount" ColumnName="SelectCount" />
            <ScalarProperty Name="StoredProcedureCount" ColumnName="StoredProcedureCount" />
            <ScalarProperty Name="DDLCount" ColumnName="DDLCount" />
            <ScalarProperty Name="PacketsReceived" ColumnName="PacketsReceived" />
            <ScalarProperty Name="PacketsSent" ColumnName="PacketsSent" />
            <ScalarProperty Name="ServerRoundTrips" ColumnName="ServerRoundTrips" />
            <ScalarProperty Name="SelectRowsRead" ColumnName="SelectRowsRead" />
            <ScalarProperty Name="StatementCacheHits" ColumnName="StatementCacheHits" />
            <ScalarProperty Name="StatementCacheMisses" ColumnName="StatementCacheMisses" />
            <ScalarProperty Name="StatementCacheReplaces" ColumnName="StatementCacheReplaces" />
            <ScalarProperty Name="StatementCacheTopHit1" ColumnName="StatementCacheTopHit1" />
            <ScalarProperty Name="StatementCacheTopHit2" ColumnName="StatementCacheTopHit2" />
            <ScalarProperty Name="StatementCacheTopHit3" ColumnName="StatementCacheTopHit3" />
            <ScalarProperty Name="PacketsReceivedPerSocketRead"
ColumnName="PacketsReceivedPerSocketRead" />
           <ScalarProperty Name="BytesReceivedPerSocketRead" ColumnName="BytesReceivedPerSocketRead"
/>
            <ScalarProperty Name="PacketsSentPerSocketWrite" ColumnName="PacketsSentPerSocketWrite" /
>
            <ScalarProperty Name="BytesSentPerSocketWrite" ColumnName="BytesSentPerSocketWrite" />
            <ScalarProperty Name="PacketsSentPerRoundTrip" ColumnName="PacketsSentPerRoundTrip" />
            <ScalarProperty Name="PacketsReceivedPerRoundTrip"
ColumnName="PacketsReceivedPerRoundTrip" />
            <ScalarProperty Name="BytesSentPerRoundTrip" ColumnName="BytesSentPerRoundTrip" />
           <ScalarProperty Name="BytesReceivedPerRoundTrip" ColumnName="BytesReceivedPerRoundTrip" /
>
            <ScalarProperty Name="PartialPacketShifts" ColumnName="PartialPacketShifts" />
            <ScalarProperty Name="PartialPacketShiftBytes" ColumnName="PartialPacketShiftBytes" />
            <ScalarProperty Name="MaxReplyBytes" ColumnName="MaxReplyBytes" />
            <ScalarProperty Name="MaxReplyPacketChainCount" ColumnName="MaxReplyPacketChainCount" />
            <ScalarProperty Name="Id" ColumnName="Id" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
```

```
      <EntitySetMapping Name="DDTekStatus">
        <EntityTypeMapping TypeName="DDTek.DDTekStatus">
          <MappingFragment StoreEntitySet="Status">
            <ScalarProperty Name="ServerVersion" ColumnName="ServerVersion" />
            <ScalarProperty Name="Host" ColumnName="Host" />
            <ScalarProperty Name="Port" ColumnName="Port" />
            <ScalarProperty Name="SID" ColumnName="SID" />
            <!-- <ScalarProperty Name="CurrentUser" ColumnName="CurrentUser" /> -->
            <!-- <ScalarProperty Name="CurrentUserAffinityTimeout"
ColumnName="CurrentUserAffinityTimeout" /> -->
            <!-- <ScalarProperty Name="SessionId" ColumnName="SessionId" /> -->
            <ScalarProperty Name="StatisticsEnabled" ColumnName="StatisticsEnabled" />
            <ScalarProperty Name="Id" ColumnName="Id" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <FunctionImportMapping FunctionImportName="RetrieveStatistics" FunctionName=
"DDTek.Store.RetrieveStatistics" />
      <FunctionImportMapping FunctionImportName="EnableStatistics"
FunctionName="DDTek.Store.EnableStatistics" />
      <FunctionImportMapping FunctionImportName="DisableStatistics" FunctionName=
"DDTek.Store.DisableStatistics" />
      <FunctionImportMapping FunctionImportName="ResetStatistics"
FunctionName="DDTek.Store.ResetStatistics" />
      <FunctionImportMapping FunctionImportName="Reauthenticate" FunctionName=
"DDTek.Store.Reauthenticate" />
    </EntityContainerMapping>
  </Mapping>
</edmx:Mappings>
```

G. バルク ロード構成ファイル

以下のトピックでは、Zen Bulk Load で使用される構成ファイルについて説明します。

- バルク データ構成ファイルのサンプル
- バルク データ構成ファイル用の XML スキーマ定義

この機能の詳細については、Zen Bulk Load の使用を参照してください。

バルク データ構成ファイルのサンプル

バルク形式の構成ファイルは、テーブルまたは DataReader がバルク コピー操作およびバルク ロード操作を使ってエクスポート（アンロード）された場合に作成されます。

```
<?xml version="1.0"?>
<!--
Sample DDL
----------

CREATE_STMT = CREATE TABLE GTABLE (CHARCOL char(10),VCHARCOL varchar2(10), ¥

DECIMALCOL number(15,5), NUMERICCOL decimal(15,5), SMALLCOL number(38), ¥

INTEGERCOL integer, REALCOL number, ¥

FLOATCOL float, DOUBLECOL number, LVCOL clob, ¥

BITCOL number(1),TINYINTCOL number(19), BIGINTCOL number(38), BINCOL raw(10), ¥

VARBINCOL raw(10), LVARBINCOL blob, DATECOL date, ¥

TIMECOL date, TSCOL date) -->

<table codepage="UTF-16" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="file:///c:/Documents and Settings/jbruce/My Documents/Current Specs/
BulkData.xsd">
   <row>
      <column codepage="UTF-16" datatype="CHAR" length="10" nullable="true">CHARCOL</column>
      <column codepage="UTF-16" datatype="LONGVARCHAR" length="10">VCHARCOL</column>
      <column codepage="UTF-16" datatype="DECIMAL" precision="15" scale="5">DECIMALCOL</column>
       <column codepage="UTF-16" datatype="DECIMAL" precision="15" scale="5">NUMERICOL</column>
       <column codepage="UTF-16" datatype="DECIMAL" precision="38">SMALLCOL</column>
       <column codepage="UTF-16" datatype="INTEGER">INTEGERCOL</column>
       <column codepage="UTF-16" datatype="SINGLEPRECISION"">REALCOL</column>

       <!- ここに、さらに定義を続けることができます -->
   </row>
</table>
```

バルク データ構成ファイル用の XML スキーマ定義

バルク構成 XML スキーマは、バルク構成ファイルを管理します。バルク構成ファイルは、Zen Bulk Load で処理されるバルク ロード データ ファイルを管理します。

このスキーマは https://media.datadirect.com/download/docs/ns/bulk/BulkData.xsd で公開されており、この標準を使用してビルドされるサード パーティ機能の基盤を提供します。大量のデータを管理するカスタム アプリケーションまたはツールは、ODBC、JDBC、および ADO.NET の API 間や複数のプラットフォーム間で、疎結合型の Zen Bulk Load としてこのスキーマを採用することができます。

Zen Bulk Load で使用できる CSV データを生成する場合は、XML 構成ファイル用の XML スキーマを提供する必要があります。

各バルク操作では、作成されるバルク データ ファイルを記述する XML 構成ファイルを UTF-8 形式で生成します。バルク データ ファイルが作成できなかったり XML 構成ファイルに記述されているスキーマに準拠していない場合は、例外が返されます。

H. IANA コード ページ マッピング

次の表では、最も広範に使用されている IBM コード ページの IANA コード ページ名へのマップを示します。

IBM 番号	IANA コード ページ名
37	IBM037
38	IBM038
290	IBM290
300	IBM300
301	IBM301
500	IBM500
813	ISO_8859-7:1987
819	ISO_8859-1:1987
857	IBM857
860	IBM860
861	IBM861
897	IBM897
932	IBM-942_P120-2000
939	IBM-939
943	Windows-932-2000（Windows クライアント用）
943	IBM-943_P14A-2000（UNIX クライアント用）
950	Big5
1200	UTF-16
1208	UTF-8
1251	Windows-1251
1252	Windows-1252
4396	IBM-930
5025	IBM5025

IBM 番号	IANA コード ページ名
5035	IBM5035
5297	UTF-16
5304	UTF-8
13488	UTF-16BE