Wunion.DataAdapter.NetCore developers document

Lightweight database Query component library that encapsulates different kinds of database SQL commands in the form of C# object structure. This allows developers to focus more on the implementation of business code without having to care about the direct impact of different types of database command differences on the code. The component library is designed with the .NET Standard 2.0 Cross-platform standard, and supports the .NET Framework 4.6.2 above and the .NET Core 2.0 version of the various .NET applications. Leverages the cross-platform features of .NET Core to make this component library run on Windows, Mac OS X, and Linux systems.

Development IDE: Microsoft Visual Studio 2017
Development language: C#
Target framework: .NET Standard 2.x
Library Name: Wunion.DataAdapter.NetCore
Root namespace: Wunion.DataAdapter.Kernel

Chapter I: Description of the package structure for the project

- Wunion.DataAdapter.NetCore -- The main project, the core implementation layer of SQL object.
- Wunion.DataAdapter.NetCore.SQLServer -- Microsft SQL Server 2008 R2 SP3 and the above version of the database support package.
- Wunion.DataAdapter.NetCore.SQLite3 -- The support package for the SQLite database.
- Wunion.DataAdapter.NetCore.MySQL -- The MySQL database support package (tested and passed in Mariadb 10.2).
- Wunion.DataAdapter.NetCore.PostgreSQL -- The PostgreSQL database support package (tested and passed in PostgreSQL 10).
- Wunion.DataAdapter.NetCore.EntityUtils -- Database Entity Model support package (with Entity Model Code Builder tool).

Chapter II: Querying database using Wunion.DataAdapter.NetCore

2.1: Classes that you must understand by using the component library

2.1.1: DataEngine class

This class provides access to the target database for insert/deleted/modified/query operations, as well as transaction management. An object instance of a DataEngine class is always associated with a specific type of database engine, and the class also provides static cache pool management capabilities for multiple DataeEgine instances.

Initializes an instance of the Dataengine engine: Creates an instance through a public constructor. The first parameter dba is an abstract DbAccess class that represents the accessor of the target type database, implemented by various specific database support packages for the component library (such as SqlServerDbAccess or NpgsqlDbAccess, etc.) The second parameter, ParserAdapter, is the interpreter adapter for the DbCommandBuilder Command object tree, which is also implemented by various specific database support packages for the component library.

AppendDataEngine static method: This method creates an instance of the DataEngine in the static cache pool, with the first two arguments consistent with the constructor, and the third parameter represents the key name of the DataEngine instance in the cache and defaults to "Default".

GetEngine static method: Gets an instance of the DataEngine for the specified key in the cache pool, where multiple database interactions can be easily implemented using a caching pool when a variety of databases are used simultaneously in one application.

RemoveEngine static method: Deletes the DataEngine with the specified key name from the static cache pool.

CurrentEngine static property: This property return the DataEngine instance of the default (that is: key name is "Default") in the static cache pool.

2.1.2: DbCommandBuilder class

The object tree used to build SQL commands, which provides object tree building methods for general SQL commands such as SELECT, INSERT, UPDATE, and DELETE, and supports an infinite level of nested queries. Stored procedures cannot be built using the DbCommandBuilder class. When executing a command using DataEngine, the object of that type is used as a parameter, and the DbCommandBuilder object tree is interpreted internally as the SQL command for the target database and executed.

2.1.3: Static class td, exp, fm and Fun

- **td**   Create a field element in a command.
- **exp**   Create an expression element in a command, and a LIKE clause element (most of which has been completed by the operator overload, and do not need to call the shortcut class manually).
- **fm**   Create table elements and LEFT JION elements in a command.
- **Fun**   Create a function element in a command.

2.2:   Referencing and initializing component libraries

NuGet Package Reference:
- Wunion.DataAdapter.NetCore   Core library (must be referenced)
  Namespace: Wunion.DataAdapter.Kernel
  Wunion.DataAdapter.Kernel.DbInterop
  Wunion.DataAdapter.Kernel.CommandBuilders
- Wunion.DataAdapter.NetCore.SQLServer   Microsoft SQL Server database support package (referenced on-demand)
  Namespace: Wunion.DataAdapter.Kernel.SQLServer
  Wunion.DataAdapter.Kernel.SQLServer.CommandParser
- Wunion.DataAdapter.NetCore.SQLite3   SQLite database support package (referenced on-demand)
  Namespace: Wunion.DataAdapter.Kernel.SQLite3
  Wunion.DataAdapter.Kernel.SQLite3.CommandParser
- Wunion.DataAdapter.NetCore.PostgreSQL   PostgreSQL database support package (referenced on-demand)
  Namespace: Wunion.DataAdapter.Kernel.PostgreSQL
  Wunion.DataAdapter.Kernel.PostgreSQL.CommandParser
- Wunion.DataAdapter.NetCore.MySQL   MySQL database support package (referenced on-demand)
  Namespace: Wunion.DataAdapter.Kernel.MySQL
  Wunion.DataAdapter.Kernel.MySQL.CommandParser

Initializing DataEngine: The associated namespace is introduced using a reference to the NuGet package, and the initialized Code as follows:

```
SqlServerDbAccess DBA = new SqlServerDbAccess();
DBA.ConnectionString = "Server=(local);Database=test;User ID=sa;Password=******";
```

```
DataEngine.AppendDataEngine(DBA, new SqlServerParserAdapter());   // Add as Default Engine
```

The previous example uses the static cache provided by DataEngine class to globally instantiate. To freely define the global DataEngine instance if not used this idea.

2.3:   Syntax for data query

In all of the sample code in this chapter, DataEngine.CurrentEngine represents a default DataEngine instance in static cache provided by DataEngine class. When the actual project uses other means to the global DataEngine instance, please use the concrete DataEngine instance.

DataEngine provides ExecuteQuery, ExecuteQuery<DataTable>, ExecuteReader, and ExecuteScalar four methods for querying the data to respond to the query requirements of different scenarios respectively. When an error occurs during the execution of the query, the above four methods return null, and the error message and the target SQL command that caused the error are provided in the DataEngine.DBA.Error property to provide a reference for debugging.

2.3.1:   Single-Table Query

```
DbCommandBuilder  Command = new DbCommandBuilder();
Command.Select(td.Field("Field1"), td.Field("Field2"))
        .From("TableName")
        .Where(td.Field("Field1") == 123 & td.Field("Field3") == true);
SpeedDataTable dt = DataEngine.CurrentEngine.ExecuteQuery(Command);
// To query all fields, use:   td.Field("*")
```

Complex queries and their conditions: for example, the following SQL command.

> SELECT *, [Num] * [Price] AS Total FROM [Products] WHERE [Selling] = true AND [Price] BETWEEN 100 AND 998

DbCommandBuilder implementation methods are as follows:
```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Select(td.Field("*"), (td.Field("Num") * td.Field("Price")).As("Total"))
        .From("Products")
        .Where(
            td.Field("Selling") == true &
            Fun.BetweenAnd(td.Field("Price"), 100, 998)
        );
```
Element precedence groupings in the Command object tree: Understand what priority grouping is from the following SQL command

> SELECT ([Field1] + [Field2]) * [Field3]
> FROM [TableName]
> WHERE [Field5] = FALSE AND ([Field4] IS NULL OR LEN([Field4] < 1))

The addition operation priority in the arithmetic expression in the SELECT, implemented on the DbCommandBuilder as follows:

(td.Field("Field1") + td.Field("Field2")).Group * td.Field("Field3")

The query result is incorrect if the logical operation expression in the Where condition is not grouped. The priority grouping implementation of logical operations is similar to arithmetic, implemented one the DbCommandBuilder as follows:

td.Field("Field5") == false & (td.Field("Field4").IsNull() | Fun.Len(td.Field("Field4")) < 1).Group

2.3.2: Multi-Table Query

```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Select(td.Field("t2", "TestId"),          //    using :    td. Field ( "t2", "*")
                td.Field("t1", "GroupName"),    //    to Query the all fields of one table.
                td.Field("t2", "TestName"),
                td.Field("t2", "TestAge"),
                td.Field("t2", "TestSex"))
        .From(fm.Table("DATA_TABLE", "t2"),
                fm.LeftJoin("GROUP_TABLE", "t1")
                .ON(td.Field("t2", "GroupId") == td.Field("t1", "GroupId"))
        )
        .Where(td.Field("t2", "GroupId") == groupId)
        .Paging(pageSize, currentPage, td.Field("t2", "TestId"), OrderByMode.DESC); //Working with paging features
SpeedDataTable dt = DataEngine.CurrentEngine.ExecuteQuery(Command);
```

When you make paging available one the Microsoft SQL Server database, you must use ORDER BY for the query, which is not required for other databases. Multiple-table queries support only the LEFT JOIN method, and not supported RIGHT JOIN. Compound expressions, priority groupings, etc. are described in the chapter 2.3.1

2.3.3: Nested queries

```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Select(td.Field("*")).From("DATA_TABLE")
        .Where(td.Field("GroupId").In(
            Command.Nested(td.Field("GroupId")).From("GROUP_TABLE")
                    .Where(td.Field("GroupName").Like("test", LikeMatch.Center))
            ) & td.Field("TestSex") >= 18
        );
SpeedDataTable dt = DataEngine.CurrentEngine.ExecuteQuery(Command);
```

2.4:   Records -- INSERT/UPDATE/DELETE

Record's insert, update, and delete to execute a command by invoke the ExecuteNoneQuery method of the DataEngine instance. When an error occurs during the execution of a command, the method return -1. and provides the error message and the target SQL command in the DataEngine.DBA.Error property for debugging.

2.4.1:   Insert record

```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Insert("DATA_TABLE", td.Field("GroupId"), td.Field("TestName"), td.Field("TestAge"))
        .Values(2, "The test name.", 12.3f);
int result = DataEngine.CurrentEngine.ExecuteNoneQuery(Command);
// Get the value of an identity field after inserted.
int TestId = Convert.ToInt32(DataEngine.CurrentEngine.DBA.SCOPE_IDENTITY);
```

2.4.2:   Update record

```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Update("DATA_TABLE")
        .Set(td.Field("TestName") == NewName, td.Field("TestAge") == Age)
        .Where(td.Field("TestId") == Id);
int result = DataEngine.CurrentEngine.ExecuteNoneQuery(Command);
```

2.4.3:   Delete record

```
DbCommandBuilder Command = new DbCommandBuilder();
Command.Delete("DATA_TABLE")
        .Where(td.Field("TestId") == Id & td.Field("GroupId") == GrpId);
int result = DataEngine.CurrentEngine.ExecuteNoneQuery(Command);
```

2.5:   Use the QuickDataChanger shortcut class

You can use this class to build insert, update, and delete command without DbCommandBuilder, and that class provides a way to batch update SpeedDataTable and DataTable to the database.
Namespace: Wunion.DataAdapter.Kernel

Insert, Update and Delete

The insert and update record is made by invoke the Quickdatachanger.SaveToDataBase method, which define as follows

```
public int SaveToDataBase(string TableName, Dictionary<string, object> data, bool executeUpdate)
```

Parameters:

TableName   Table name (in the database)

data   A dictionary collection of fields and their values that are updating or inserting.

executeUpdate   True when the updating, False for inserting.

When an update or delete operation is performed, the conditions is specified by Conditions property of the instance of QuickDataChanger.

The following code demonstrates updating data to a record, attempting an insert operation if the update fails, or incorrect incoming data if the insert operation still fails.

```
QuickDataChanger DC = new QuickDataChanger();
DC. Conditions.Add(td.Field("TestId") == TestId & td.Field("GroupId") == GroupId);
int result = DC.SaveToDataBase("DATA_TABLE", Data, true);
if (result < 1)
{
    DC. Conditions.Clear(); // The condition of the update should be cleared when the insert operation is performed
    result = DC.SaveToDataBase("DATA_TABLE", Data, false);
}
// If the value of result at this time is still less than 1, then the incoming data is incorrect.
// Deleting records invoke the DC.Delete method, as follows:
//  result = DC.Delete("DATA_TABLE");
```

Bulk update of data collection

The mechanism for batch updating needs to be defined itself, and the following code assumes that the "STATE" column is defined in the speedTable to represent the status information for each row of data, and invoke the BatchSaveDataTable method to use this rule to bulk update the data collection to the database.

A batch update should be performed in a transaction to increase it's execution efficiency and rollback control of the operation, it's bad for the performance if not performed in a transaction.

```
QuickDataChanger DC = new QuickDataChanger(trans);
int result = DC.BatchSaveDataTable(speedTable, (SpeedDataRow Row) => {
        DataRowSubmission RowSubmission = null;
        switch (Row.Field<int>("STATE"))
        {
            case 1:
                RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Update);
                RowSubmission.NotInsertFields = new string[] { "TestId" };    // Do not update field TestId.
                RowSubmission.Conditions.Add(td.Field("TestId") == Row.Field<int>("TestId")); //Set update condition
                break;
```

```csharp
            case 2:
                RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Insert);
                RowSubmission.NotInsertFields = new string[] { "TestId" }; //Do net insert field TestId.
                break;
            case 3:
                RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Delete);
                // Set delete conditions
                RowSubmission.Conditions.AddRange(new object[] {
                    td.Field("TestId") == Row.Field<int>("TestId") &
                    td.Field("GroupId") == Row.Field<int>("GroupId")
                });
                break;
            default:
                RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Ignore);
                break;
        }
        return RowSubmission;
});
trans.Commit();
trans.Dispose();
```

2.6:   Use transaction

Use the BeginTrans() method of the DataEngine instance to start a transaction and return a controller object of the DBTransactionController type, DBTransactionController class is located in the Wunion.DataAdapter.Kernel.DbInterop namespace.

Executing command in a transaction

When executing commands in a transaction, you should use the DBTransactionController.DBA.ExecuteNoneQuery method, instead of the ExecuteNoneQuery method of the DataEngine instance. The following sample code:

```csharp
using (DBTransactionController trans = DataEngine.CurrentEngine.BeginTrans())
{
    trans.DBA.ExecuteNoneQuery(Command);
    trans.Commit();
}
```

The object of the QuickCataChanger controlled by transaction
Example:
```csharp
int result = -1;
using (DBTransactionController trans = DataEngine.CurrentEngine.BeginTrans())
{
    QuickDataChanger DC = new QuickDataChanger(trans);
```

```
            DC.Conditions.Add(td.Field("GroupId") == GrpId);
            DC.Delete("DATA_TABLE");
            DC.Conditions.Clear();
            DC.Conditions.Add(td.Field("Id") == GrpId);
            result = DC.Delete("GROUP_TABLE");
            if (result > 0)
                trans.Commit();   // Commit transaction when a group deletion succeeds.
    }
```

A transaction must be freed after it has been used. Code in the scope of the transaction control should use Try...catch for error handling, and When an exception occurs should invoke the transaction controller object's Rollback method for rollback, and invoke the Commit method to commit the transaction manually if all code within the transaction control scope is completed normally. When release a transaction, a rollback operation is performed automatically if the transaction is not committed. so, the commit of the transaction must be performed manually.

2.7: Enable and configure connection pools

Connection Pools support new features for version 1.0.4, using the UseDefaultConnectionPool or UseConnectionPool methods for DataEngine object instances to enable and configure, and the differences between the two methods are described below.

UseDefaultConnectionPool: Configure and use the default connection pool implemented by the class library itself.

Definition: public void UseDefaultConnectionPool(Action<IDbConnectionPool> configure)

Parameter:

            configure   -- An external method used to setup the connection pool.

Example:

```
DbAccess dba = new SqlServerDbAccess();
dba.ConnectionString = "……";
ParserAdapter adapter = new SqlServerParserAdapter();
DataEngine db = new DataEngine(dba, adapter);
db.UseDefaultConnectionPool((pool) => {
    pool.RequestTimeout = TimeSpan.FromSeconds(3);
    pool.ReleaseTimeout = TimeSpan.FromMinutes(5);
    pool.MaximumConnections = 200;
});
```

UseConnectionPool: Configure and use the custom connection pool that you implement yourself (custom connection pool as implemented by Wunion.DataAdapter.Kernel.IDbConnectionPool interface specification).

Definition: public void UseConnectionPool(IDbConnectionPool connectionPool)

Parameter:

            connectionPool   -- An instance of an object that customizes the connection pool.

IDbConnectionPool interface: The implementation specification interface for the connection pool, whose members are described below.

Member of Properties		
RequestTimeout	TimeSpan	Get or set a response timeout for assigning a connection to connection pool request, which should be determined by the number of seconds. If the timeout is over, an exception should be throw.
ReleaseTimeout	TimeSpan	Get or set the forced release timeout that takes up the connection, which is judged by the number of minutes. A connection has not been released since the time it was assigned, and the connection pool should be forced to retract it.
MaximumConnections	int	Gets or sets the maximum number of connections to the pool. This setting should be based on the specific database and server operating system.
Count	int	The property is read-only and returns the number of all connections in the connection pool ,including the free connections and the connections being occupied.
Member of methods		
GetConnection(MakeConnectionFactory makeFactory)	IDbConnection	Request a connection to the connection pool and return. Parameter: makeFactory represents the method used to create a connection when the connection pool is empty, and is provided by DbAccess implemented by the various types of database support packages. Its delivery is done by DataEngine's internal operating mechanism.
ReleaseConnection(IDbConnection connection)	void	Connection pool reclaims the specified connection. This method only recycles connections to the idle queue and does not close the connection to the database.

Chapter III: Wunion.DataAdapter.NetCore.EntityUtils expansion package

This package is an entity model expansion supported of the Wunion.DataAdapter.NetCore class library, which are lazy the Wunion.DataAdapter.NetCore package. The mapping mechanism for entity classes and database tables is defined in this package.

This package is designed to help developers in the development process not need refer to the database design documents for database programming, only through the entity of the query Agent object can complete the database to add, delete, change, query operations.

Namespace: Wunion.DataAdapter.EntityUtils

3.1: Related underlying types of the entity objects

An entity object (data entity object) corresponds to a row of one table in the database, and the property member of the entity corresponds to the field one by one of the table. Include name, data type, etc.

DataEntity class   The class is the parent class for all data entity classes, and all data entities should inherit from the class. The standard behavior of dynamic get and set value of the entity properties members is implemented in this class, as well as the method of outputting data entity as a dictionary.

DynamicEntity class   A dynamic entity class, and the so-called dynamic entity refers to the entity's properties members dynamically generated according to the different query results. The result that is returned when a developer makes a multiple-table query is a collection of entities of this type. When using an entity or an entities list of that type, you should use the **dynamic** to define entity variables or type parameters of a generic collection as **dynamic**, so that the runtime can automatically infer property members of dynamic entities. Instead of using the DynamicEntity definition to limit dynamic property member inference at run time.

EntityPropertyAttribute class   Used to mark the properties of an entity object to the attributes of a table field, such as primary key field, self increment field, null values are allowed, default value, and so on. In addition to the entity of the DynamicEntity type, property members of all other types of entities must use this class to mark the above property characteristics of their corresponding table fields.

3.2: Agent objects base type for entity queries

Each data entity that derives from (i.e. inherits) the DataEntity class corresponds to one agent class (that is used to assist in creating the query object tree), and one context mapper class of a table. Where the query helper class agent is used to generate a field or expression element for an entity property during the creation of a query. the table context class are used to create and execute queries for the table (include insert, delete, update, delete). For example, one table "ProjectInfo" should have the following

three classes:

- ProjectInfo   (Data entity class , inherit from DataEntity class)
- ProjectInfoAgent   (Query helper class, inherit from EntityAgent class)
- ProjectInfoContext   (Table context mapper class, inherit from TableContext<TEntityAgent> class)

EntityTableAttribute class   Used to mark the table mapped by the agent class.

EntityAgent abstract class   Parent class of all the query helper, all the children class of EntityAgent must be marked it's corresponding table in the database by use [Entitytableattribute(TableName = "xxx")] on the class define.

TableContext<TEntityAgent> abstruct class   The parent class for all the table context mapper class. This class is a generic class, and the generic type parameter TEntityAgent is an Entity Query helper Class. Different tables should be corresponding to their respective helper classes.

All of these entity classes and agent classes, and their interrelationships are automatically generated by the code generator, and developers do not need to manually write them and only need to know about them.

3.3、Overview of the DatabaseContext class

The database context object implemented in the way of read-write separation, through a DatabaseContext object instance, you can query and add, delete and update operations, which are always related to data entity and table context mapping. The constructor of the class is declared to be a protected level, so it is not possible to create an instance of it directly through the new keyword, but to initialize and return a DatabaseContext object instance through static method Initialize.

When developers use the Wunion.DataAdapter.NetCore.EntityUtils expansion package for the database interaction in a project, do not have to initialize instance of the DataEngine. you need initialize a instance of the DatabaseContext only, and to globally caching of this instance. The reason is that the DataEngine instance for read and write control is included within the DatabaseContext.

3.3.1、DatabaseContext.ContextOptions class

The option required to initialize a DatabaseContext instance.

Property Members	Type	Description
ReadAccess	DbAccess	A data accessor for database read access that contains connection information. Provided by the database support package described in Chapter I.
WriteAccess	DbAccess	A data accessor for database write access that contains connection information. Provided by the database support package described in Chapter I.
Parser	ParserAdapter	An interpreter adapter for the Command object tree required by an

		internal DataEngine instance. Provided by the database support package described in Chapter I.

3.3.2: Initializes a DatabaseContext instance

The following code initializes an DatabaseContext instance with PostgreSQL database:

```csharp
DatabaseContext.ContextOptions options = new DatabaseContext.ContextOptions();
options.ReadAccess = new Wunion.DataAdapter.Kernel.PostgreSQL.NpgsqlDbAccess();
// Set the database connection for read access, which is typically read from the configuration file
options.ReadAccess.ConnectionString =
"Host=192.168.1.106;Port=5432;Username=postgres;Password=***;Database=test";
options.WriteAccess = new Wunion.DataAdapter.Kernel.PostgreSQL.NpgsqlDbAccess();
// Set the database connection for write access, which is typically read from the configuration file
options.WriteAccess.ConnectionString =
"Host=192.168.1.105;Port=5432;Username=postgres;Password=***;Database=test";
options.Parser = new Wunion.DataAdapter.Kernel.PostgreSQL.CommandParser.NpgsqlParserAdapter();
DatabaseContext DbContext = DatabaseContext.Initialize(options);
// Next, we need to DbContext the global cache processing,
// how to globally cache everyone has their own preferences,
// this does not make too much elaboration.
```

3.3.3: Enable and configure connection pools

Connection Pool supports new features for Wunion.DataAdapter.NetCore.EntityUtils 1.0.3 to enable and configure the connection pool by calling the UseConnectionPool method of the DatabaseContext object instance. Here's the code:

```csharp
DbContext.UseConnectionPool((readEngine, writeEngine) => {
    readEngine.UseDefaultConnectionPool((pool) => {
        // Configure the read connection pool here.
    });
    writeEngine.UseDefaultConnectionPool((pool) => {
        // Configure the write connection pool here.
    });
});
```

If the database is not read and write separated, the same connection pool can be applied to the read and write DataEngine as follows:

```csharp
DbContext.UseConnectionPool((read, write) => {
    IDbConnectionPool defPool = null;
    read.UseDefaultConnectionPool((pool) => {
```

```
            // Configure the write connection pool here.
            defPool = pool;
        });
        write.UseConnectionPool(defPool);    // Use the same connection pool as read.
    });
```

3.4:   Query Database

Any insert, delete, update and select operation of the database are carried out through DatabaseContext object. the Agent class and the Context class are the core of grammatical performance during the whole query process. DbContext represents the instantiated DatabaseContext object in all the sample code in this chapter.

3.4.1、Single-Table Query

A query for a single table expands through the DbContext.Table<TContext>() generic method, whose generic type parameter TContext should be the context mapper class for a particular table, and its return type is an object instance of the type specified by TContext. The following sample code demonstrates query   all projects created by a specified account within one week from the "ProjectInfo" table:

```
List<ProjectInfo> Result = DbContext.Table<ProjectInfoContext>().Select(p => {
    // The query agent object that the variable p corresponds to the data entity
    // it's type in this lambda expression is ProjectInfoAgent
    p.AccountID == accountId & Fun.BetweenAnd(p.CreationDate, DateTime.Now.AddDays(-7), DateTime.Now)
});
```

If you need to sort and pagination, see the following example code:

```
List<ProjectInfo> Result = DbContext.Table<ProjectInfoContext>().Select((p, selector) => {
    // The query agent object that the variable p corresponds to the data entity
    // it's type in this lambda expression is ProjectInfoAgent
    // selector is used to build query conditions, sorting and paging, etc.
    selector.Where(p.AccountID == accountId &
            Fun.BetweenAnd(p.CreationDate, DateTime.Now.AddDays(-7), DateTime.Now)
        ).OrderBy(p.CreationDate, OrderByMode.DESC)    // sorting
        .Paging(pageSize, currentPage)    // paging
});
```

The query Command object tree is typically constructed in a lambda expression (that is, an anonymous function) in the Select method, as can be seen from the two example query code. Basically all queries are done that way.

3.4.2: Multi-Table Query

A multiple-table query expands through the DbContext.From<TEntityAgent>() generic method, which has a generic parameter TEntityAgent representing the query agent class corresponding to the data entity. The DbContext.From method return an instance of a MultiQuerySelector type of composite query filter object, that handles the command object tree for multiple table queries. The return result for a multiple-table query is always the List<dynamic>, where a single element in the collection is a dynamic entity object of type DynamicEntity.

The Join() method is implemented in the form of a left join, and not supported right join. Therefore, in the use of multiple table query should first be sorted out the logical order of the table.

Sample code:
```
List<dynamic> Result = DbContext.From<PurchasesAgent>().Join<ProductInfoAgent>()
    .On<PurchasesAgent, ProductInfoAgent>((t0, t1) => {
        // The parameter t0, t1 represents the previous and last table of the left join.
        t0.ProductId == t1.ProductId
    }).Where(t => new object[] {
        // Create a query condition in the Where method.
        // t[0], t[1].....t[n] represents the tables added in the previous From() method and Join() method,
        // indexed in the order in which they were added.
        t[0].Agent<PurchasesAgent>().AccountID == accountId &
        t[1].Agent<ProductInfoAgent>().Selling == true
    }).OrderBy(t => t[0].Agent<PurchasesAgent>().CreationDate, OrderByMode.ASC)
    .Select(t => new object[] {
        // Create the fields you want to discover in the Select method.
        // t[0], t[1].....t[n] represents the tables added in the previous From() method and Join() method,
        // indexed in the order in which they were added.
        t[0].Agent<PurchasesAgent>().Id,
        t[0].Agent<PurchasesAgent>().Quantity,
        t[1].Agent<ProductInfoAgent>().ProductName,
        t[1].Agent<ProductInfoAgent>().Price,
        (t[0].Agent<PurchasesAgent>().Quantity * t[1].Agent<ProductInfoAgent>().Price).As("Total")
        t[0].Agent<PurchasesAgent>().CreationDate
    });
```

3.4.3、QueryScalar method

Used to return the first column of the first row in the query result. all other columns, rows are ignored. In some cases need using the Count function to measure the amount of data that is recorded or queried by this method. For example, when paging data, you need to query the number of records that match the criteria to calculate the total pages.

The following code shows the statistics of all order information generated within a week from the

"Purchases" table:

```
object Result = DbContext.From<PurchasesAgent>().Where(t => new object[] {
        Fun.BetweenAnd(t[0].Agent<PurchasesAgent>().CreationDate, DateTime.Now.AddDays(-7), DateTime.Now)
    }).QueryScalar(t => new object[] {
        Fun.Count("*")
    });
int recordCount = Convert.ToInt32(Result);
```

3.4.4: Insert record

Through the DbContext.Table<TContext>().Add method add new record to the corresponding table. and throws an exception if the new record fails. The following example code shows a new record in table "ProductInfo":

```
ProductInfo product = new ProductInfo();
product. Category = CategoryId;
product.ProductName = "Your product name.";
product.Price = 127.86f;
product.Selling = true;
product.PushDate = DateTime.Now;
product.AccountID = myId;
DbContext.Table<ProductInfoContext>().Add(product);
// If the ID field in the "ProductInfo" table is from incremental,
// the inserted ID is automatically assigned to the product.Id property.
```

3.4.5: Update record

Through the DbContext.Table<TContext>().Update method updates the records in the specified table. and throws an exception when an update condition is not specified at update time. The following example code demonstrates updating the corresponding record in the "ProductInfo" table:

```
// Product is one new created ProductInfo data entity object, or that is obtained through a query.
product.Category = CategoryId;
product.ProductName = pName;
product.Price = price;
product.Selling = true;
product.PushDate = DateTime.Now;
```

```
product.AccountID = myId;
DbContext.Table<ProductInfoContext>().Update(product, t => new object[] {
    // Update conditions
    t.Id == productId
});
```

3.4.6:   Delete records

Through the DbContext.Table<TContext>().Delete method deletes the records in the specified table, and throws an exception when no deletion condition is specified. The following example code demonstrates removing a category from the "ProductCategories" table:

```
int result = DbContext.Table<ProductCategoriesContext>().Delete(t => new object[] {
    // Delete conditions
    t.CategoryId == categoryId
});
// All product information under this category should be deleted after the product classification has been successfully
deleted.
if (result > 0)
{
    result = DbContext.Table<ProductInfoContext>().Delete(t => new object[] {
        t.Category == categoryId
    });
}
```

3.4.7:   Use transaction

Starts a transaction through the BeginTransAction() method of the DbContext object, and return a DBTransactionController controller object, DBTransactionController class is located in the Wunion.DataAdapter.Kernel.DbInterop namespace.
When release a transaction, a rollback operation is performed automatically if the transaction is not committed. so, the commit of the transaction must be performed manually. The following example code demonstrates the use of transactions to remove a category from the "ProductCategories" table:

```
int result = 0;
using (DBTransactionController trans = DbContext.BeginTransAction())
{
    // Delete all product information under the category.
    result = DbContext.Table<ProductInfoContext>().Delete(trans, t => new object[] {
        t.Category == categoryId
    });
```

```
    //Delete a category only if when delete product information without generating an error.
    if (result != -1)
    {
        result = DbContext.Table<ProductCategoriesContext>().Delete(trans, t => new object[] {
            t.CategoryId == categoryId
        });
    }
    if (result > 0)    // Commit a transaction only if the deletion succeeds.
        trans. Commit();
}
```

Insert and update of the transaction are used in much the same way.

3.5:   Use of the code generator

The code generator is used to build the corresponding entity classes and agent classes for the tables in a given database, eliminating the work of developers to write the code manually (let the machine write that code for you). Using code generators even allows developers to not understand how entity classes and query agent classes are implemented, just use them. The tool provides a Windows graphical interface version and a cross-platform command line version in WdaEntityGenerator.zip.

3.5.1:   Windows Graphical Code Builder

After extracting the WdaEntityGenerator.zip, in the Windows folder, run the WdaEntityGenerator.exe select locale to the main interface of the code generator, as shown in the following illustration:
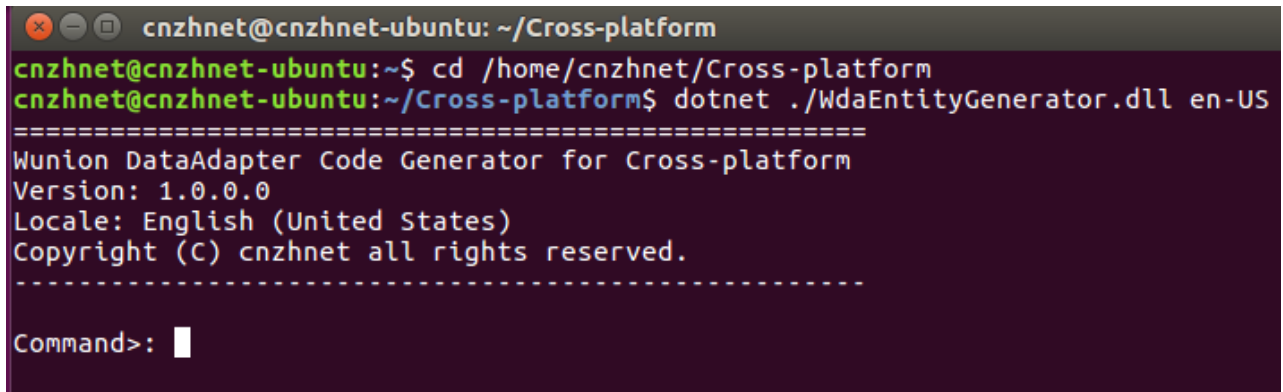
3.5.2: Cross-platform command line code generator

After extracting WdaEntityGenerator.zip in the Cross-platform folder, the language environment of the command-line code generator supports Simplified Chinese and English. Before running the command line code generator, you first need to install the .NET Core SDK on Linux or Mac OS X, see the Microsoft Official Web site: https://www.microsoft.com/net/download/linux/build The following example describes the use of the command-line code generator with Ubuntu Linux.
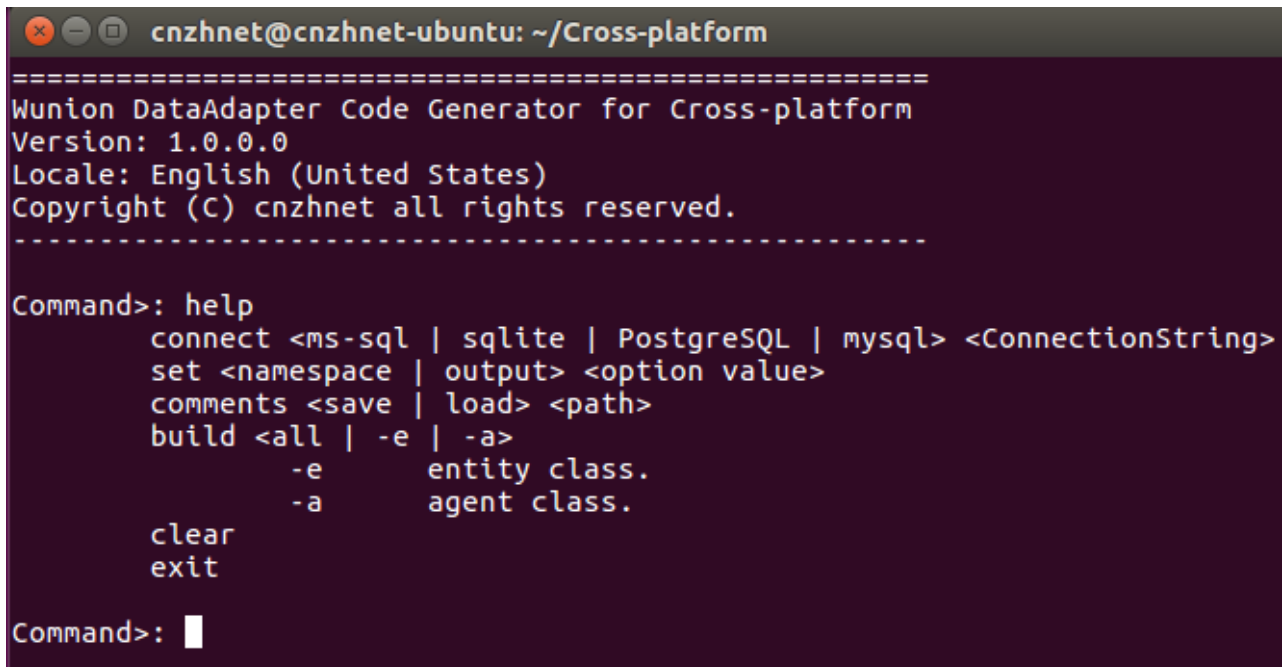
1. Use the "cd" command to enter the directory where the code was born:

    cd /home/cnzhnet/Cross-platform

2. Execute command: dotnet ./WadEntityGenerator.dll en-US

        or   dotnet ./WadEntityGenerator.dll zh-CN

en-US for the English language environment, zh-CN for the Simplified Chinese language environment, the following figure:

3. Execute the "help" or "–h" command to view all of the commands supported by this tool and their assistance information

4. Execute "connect" command connect to the given target database, the "connect" command has two parameter options, the first parameter is the target database type, and the supported database types are listed below:

ms-sql   Represents a Microsoft SQL Server database

sqlite   Represents a SQLite database

PostgreSQL   Represents a PostgreSQL database

mysql   Represents a MySQL database

The following command takes Microsoft SQL Server for example:

connect ms-sql Server=192.168.1.9;Database=Demo;User ID=sa;Password=******

After successfully connecting to the database, the following figure:

If the connection is not successful, an error message is printed.

5. Execute command "set" to set the namespace used by the generated code:

    set namespace Ling.TestCode.Entities

6. Execute command "set" to set the build path of the code:

    set output /home/cnzhnet/CodeOutput

7. Execute command   "build" generating code, and "build" command has an option. Values are: "all" or "–e" or "–a"

    all means to generate entity classes and agent classes; -e means to generate only entity classes; -a means to generate the agent class and the context class;
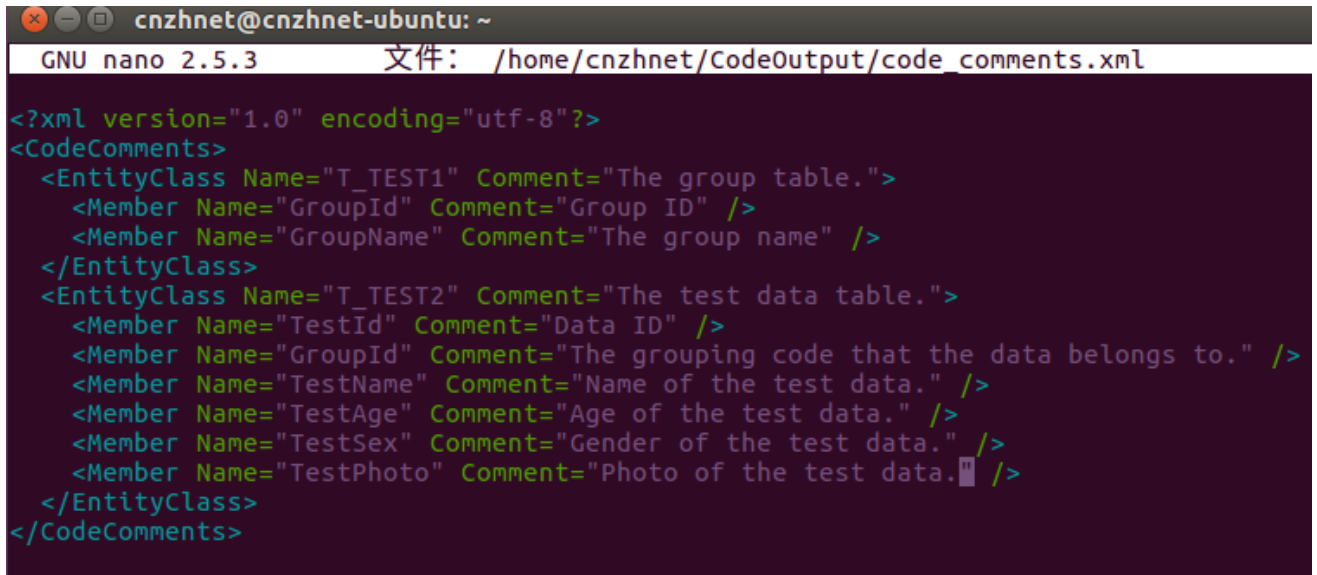
    Execution: "Build all" effect as shown below:

8. If you do not add descriptive information to the tables, and their fields in the database when you create the database, the generated entity classes do not have code comments. At you can generate an XML code comment document by using the "comments" command, and then use the gedit or Nano editor to annotate tables and fields. Note: when the document edits are finished saving it. Use the "comments" command again to load the comments information for the code to be generated. If you already have comments or do not need code comments, the work generated by this code is finished, and you get all the code you generate in the output directory, without having to continue with all the steps that follow.

9. Execute command to generate a comment document:

    comments save /home/cnzhnet/CodeOutput/code_comments.xml

10. Open the comments document for editing in the new terminal:

    nano /home/cnzhnet/CodeOutput/code_comments.xml

11. Execute command load the code comments:

    comments load /home/cnzhnet/CodeOutput/code_comments.xml

12. Regenerate the code in step 7th by executing the "build" command (no need to delete the last generated code, the generator will automatically overwrite)

    To verify that the generated code is executable: The file in the "ls" command list output directory, and open any one of the code files to view

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using Wunion.DataAdapter.EntityUtils

namespace Ling.TestCode.Entities
{
    /// <summary>
    /// The test data table.
    /// </summary>
    [Serializable()]
    public class T_TEST2 : DataEntity
    {
        /// <summary>
        /// Create a <see cref="Ling.TestCode.Entities.T_TEST2" /> object in
        /// </summary>
        public T_TEST2() { }

        /// <summary>
        /// Data ID
        /// </summary>
        [EntityProperty(AllowNull=false, PrimaryKey=true, IsIdentity=true, D
        public int TestId
        {
            get { return GetValue<int>("TestId"); }
            set { SetValue("TestId", value); }
        }

        /// <summary>
        /// The grouping code that the data belongs to.
        /// </summary>
        [EntityProperty(AllowNull=false, DefaultValue=0)]
        public int GroupId
        {
            get
```

Chapter IV: Use CodeFirst ORM

Starting with Wunion.DataAdapter.NetCore 1.0.5, Added ORM support for CodeFirst mode and officially deprecated The Wunion.DataAdapter.NetCore.EntityUtils package If you want to continue using Wunion.DataAdapter.NetCore.EntityUtils, you should choose Wunion.DataAdapter.NetCore 1.0.4.1 and earlier.
Example project: https://github.com/cnzhnet/Wunion.DataAdapter.Examples

4.1: Install CodeFirst ORM

CodeFirst support is integrated in the Wunion.DataAdapter.NetCore 1.0.5.x and later nuget packages, please install the following nuget packages and tools in the solution's application data layer project:

● Wunion.DataAdapter.NetCore 1.0.5.1 and above
● Wunion.DataAdapter.NetCore.SQLServer   (The database support packages)
  Or Wunion.DataAdapter.NetCore.MySQL
  Or Wunion.DataAdapter.NetCore.PostgreSQL
  Or Wunion.DataAdapter.NetCore.SQLite3
  Please install the corresponding database support package according to your actual needs, you don't need to install them all.
● WDA-CF Toolkit (CodeFirst Tool: Data Accessor for Generating Database Schemas and Entity Queries)
  Use the developer command prompt or terminal cd to the solution's application data-tier project and execute the installation command:

> dotnet tool install WDA-CF --version 1.0.1

Note: The tool only supports .NET core 3.1 or .NET 5.0 or .NET 6.0
The target framework for this project should be set to .NET core 3.1 or .NET 5.0 or .NET 6.0, and WDA-CF will not support CodeFirst builds in projects with .netstandard as the target framework.

4.2: Overview of the object model of CodeFirst ORM

The entire ORM implementation consists of DbContext, DbTableContext<TEntity>, QueryDao, and QueryBuilder<TDAO> wrappers, and the user descriptions for each class are as follows:
DbContext: A database context that defines the database schema and the behavior of CodeFirst generation and migration, which is an abstract class that you inherit and implement your database schema. A DataEngine object provides the DbContext with read and write access to the database (passed in via a constructor), where you only need to define the tables contained in the database and the generation and data migration behavior of CodeFirst.
DbTableContext<TEntity>:   A table context object that defines a table in a database, and the generic type TEntity is the type name of the entity corresponding to the table. This class encapsulates the insertion, deletion, and modification of table.

QueryDao: The query data accessor for the entity, which is an abstract class. The CodeFirst tool "WDA-CF" automatically generates a QueryDao derived class corresponding to each entity based on the entity, and it's role is to provide field description (FieldDescriotion) information corresponding to the various table field properties of the entity at query time, which describes the field information of the entity property in the SQL command.

QueryBuilder\<TDAO>:   To build various forms of database query commands and create corresponding QueryCommandExecuter instance to execute database queries.

The logical call relationship of the above classes is detailed in the following figure:

4.2.1: DbContext class

Definition

Namespace: Wunion.DataAdapter.Kernel.CodeFirst
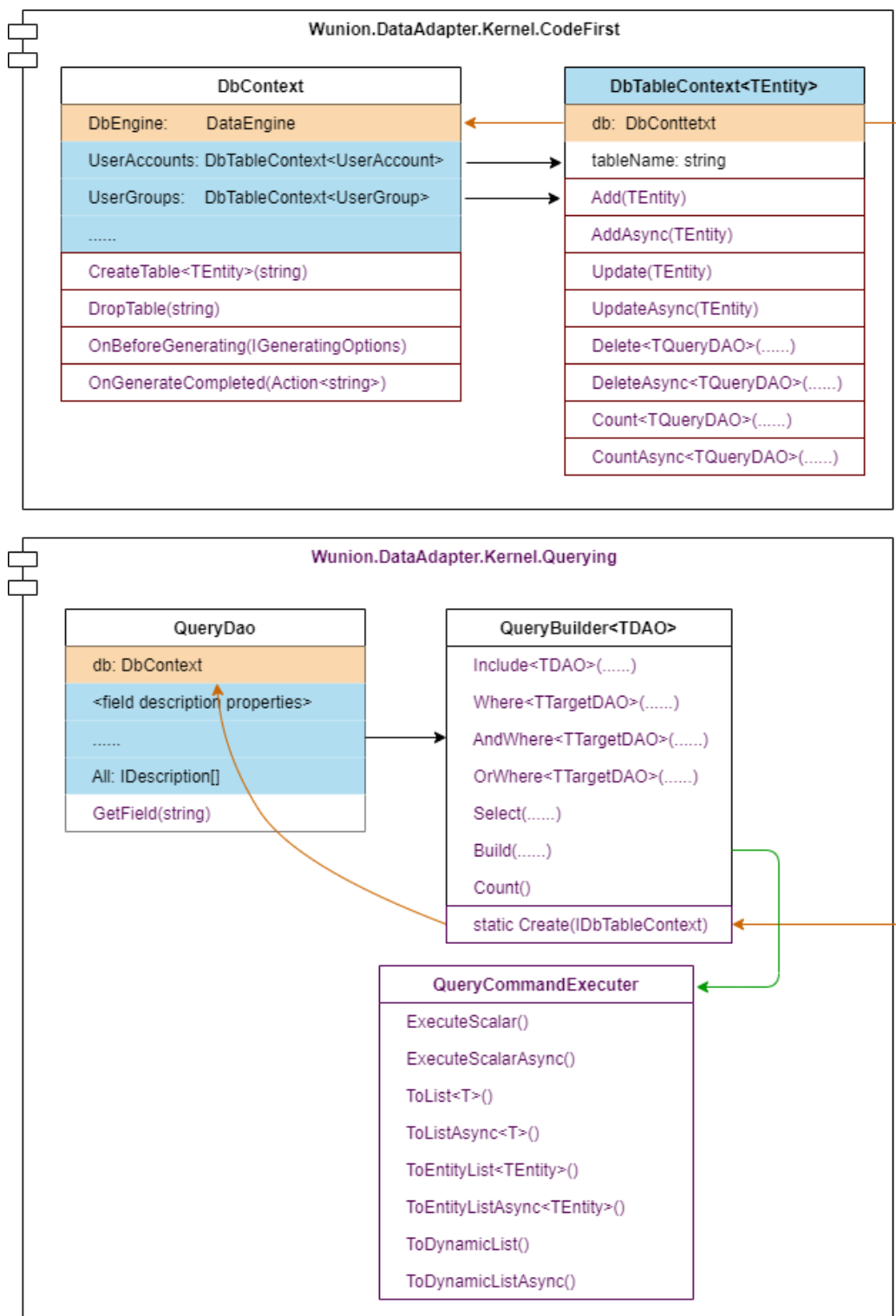Assembly: Wunion.DataAdapter.NetCore.dll
Package: Wunion.DataAdapter.NetCore   v1.0.5.1

Note

Defines the database schema and the behavior of CodeFirst generation and migration, which is an abstract class that you inherit and implement your database schema. A DataEngine object provides the DbContext with read and write access to the database (passed in via a constructor), where you only need to define the tables contained in the database and the generation and data migration behavior of CodeFirst.

Constructors

protected DbContext(DataEngine)	Create an object instance of the database context. The DataEngine parameter is used to provide the basics of read and write access to the database, and the connection string, connection pool, value converter, and other settings of the database are configured on this object.

Properties

DataEngine DbEngine	Gets this database context-dependent database engine for read and write access to the database. Use the object to create transactions, batch execute, or execute commands directly

Methods

TableDeclaration<TEntity>(string)	Gets or defines a table in a database. The String parameter represents the table name, which is defined first when the table with that name is not defined. A table context object is returned directly when a table of that name already exists. Return type: DbTableContext<TEntity>
CreateTable(string, Type, object)	Execute the command to create the table on the database. String param: new table's name Type param: The entity type for which the table corresponds

	Object param: The default value is null. When this operation is to be executed in a transaction, a DBTransactionController object should be passed in; When this operation is to be performed in a batch processor, a BatchCommander object should be passed in; If other values are passed in, an ArgumentException exception is thrown.
`CreateTable<TEntity>(string, object)`	Executes a command to create a table on the database based on an entity of the specified type. String param: new table's name Object param: The default value is null. When this operation is to be executed in a transaction, a DBTransactionController object should be passed in; When this operation is to be performed in a batch processor, a BatchCommander object should be passed in; If other values are passed in, an ArgumentException exception is thrown.
`DropTable(string, object)`	Deletes the specified table from the database. String param: The table name of the table to delete Object param: The default value is null. When this operation is to be executed in a transaction, a DBTransactionController object should be passed in; When this operation is to be performed in a batch processor, a BatchCommander object should be passed in; If other values are passed in, an ArgumentException exception is thrown.
`OnBeforeGenerating(IGeneratingOptions)`	An abstract method for configuring database connections and generating behavior before creating or updating the database schema. This method is called by the WDA-CF
`OnGenerateCompleted(Action<string>)`	This method is called when the CodeFirst tool (WDA-CF) has completed database generation. Typically used to pre-set data to the database, you can do not implement this method. Action<string> param: Used to output log content to WDA-CF.

4.2.2: DbTableContext<TEntity> class

Definition

Namespace: Wunion.DataAdapter.Kernel.CodeFirst
Assembly: Wunion.DataAdapter.NetCore.dll
Package: Wunion.DataAdapter.NetCore   v1.0.5.1

Note

A table context object that defines a table in a database, and the generic type TEntity is the type name of the entity corresponding to the table. This class encapsulates the insertion, deletion, and modification of table.

TEntity generic parameter: Represents the entity type to which the table is to be mapped

Constructors

`public DbTableContext(DbContext, string)`	Create a context-mapped object for a database table. DbContext param: the database context to which the table belongs. String param: the table name of the table in the database.

Properties

`DbContext` db	Get the database context for the table.
`string` tableName	Gets the table name in the database.
`IDescription[]` QueryFields	Gets all query field descriptions for the table.

Methods

`public virtual int` Add(`TEntity`, `object` = null)	Insert a new record into the table. TEntity param: the entity object of the record to be inserted. object parameter: The transaction controller (DBTransactionController) or BatchCommander object that performs data insertion.
`public async Task<int> AddAsync(TEntity, object = null)`	Insert a new record into the table (Asynchronous mode). All parameters are the same as Add(...) method
`public virtual void Update(TEntity, object = null)`	Updates the record specified in the table, which uses the primary key attribute marked in the entity class as the update condition, and throws an exception when the primary key attribute is not identified in the corresponding entity class. TEntity param: Entity object for the record to be updated. object param: The transaction controller (DBTransactionController) or BatchCommander object that executes the update.

`public async Task UpdateAsync(TEntity, object = null)`	Updates the record specified in the table (Asynchronous mode). All parameters are the same as Update(...) method
`public void Delete<TQueryDAO>(Func<TQueryDAO, object[]>, object = null) where TQueryDAO : QueryDao`	Delete the records that meet the criteria from the table. Func<TQueryDAO, object[]> param: to build delete conditions. object param: The transaction controller (DBTransactionController) or BatchCommander object that performs the deletion.
`public async Task DeleteAsync<TQueryDAO>(Func<TQueryDAO, object[]>, object = null) where TQueryDAO : QueryDao`	Delete the records that meet the criteria from the table (Asynchronous mode). All parameters are the same as Delete(...) method
`public int Count<TQueryDAO>(Func<TQueryDAO, object[]> = null, object = null) where TQueryDAO : QueryDao`	Query the table for the number of total records that meet the criteria. Func<TQueryDAO, object[]> param: to build query conditions. object param: The transaction controller (DBTransactionController) or BatchCommander object that performs the deletion.
`public async Task<int> CountAsync<TQueryDAO>(Func<TQueryDAO, object[]> = null, BatchCommander = null) where TQueryDAO`	Query the table for the number of total records that meet the criteria (Asynchronous mode).

4.2.3: QueryBuilder<TDAO> class

Definition

Namespace: Wunion.DataAdapter.Kernel.Querying
Assembly: Wunion.DataAdapter.NetCore.dll
Package: Wunion.DataAdapter.NetCore   v1.0.5.1

Note

Build various forms of database queries. The generic parameter TDAO represents the query for the corresponding Data Access Object type, which is always related to entities and tables. The CodeFirst tool generates a DAO class for each table corresponding to the entity at the same time as generating the database schema, and you don't need to care too much about the names of the tables and fields in the database by creating queries for table.

Constructors

Protected by assembly

Methods

`public static QueryBuilder<TDAO>` `Create(IDbTableContext)`	Static method that create an object instance of the QueryBuilder. IDbTableContext param: A table context object that create queries primarily against that table. Subsequently, more tables can be associated with the Include method.
`public QueryBuilder<TDAO>` `Include<TForeignDao>(Func<IncludeQueryBuilder<T DAO>,object[]>) where TForeignDao : QueryDao`	Associate a table into query. The generic parameter TForeignDao indicates the DAO query accessor type corresponding to the table to be associated. Func<includeQueryBuilder<TDAO>, object[]> param: for building associative conditional expressions (ON in the LEFT JOIN statement)

`public QueryBuilder<TDAO>`
`Include<TForeignDao>(string,`
`Func<IncludeQueryBuilder<TDAO>,object[]>) where TForeignDao : QueryDao`
	Associate a table into query.
	The generic parameter TForeignDao indicates the DAO query accessor type corresponding to the table to be associated.
	string param: The table name of the table to be associated in the database, when the database storage strategy design adopts the split multiple tables, an entity will correspond to multiple tables, and this parameter needs to be used to correctly associate the required table.
	Func<includeQueryBuilder<TDAO>, object[]> param: for building associative conditional expressions (ON in the LEFT JOIN statement)

`public QueryBuilder<TDAO>` `Where<TargetDao>(Func<TargetDao,object[]>) where TargetDao : QueryDao`	Create query criteria. The generic parameter TargetDao represents the DAO query accessor type corresponding to the table existing in the query. If the given TargetDao type is not included in the query, an exception will be thrown. Func<TargetDao, object[]> param: used to construct query condition expressions.
`public QueryBuilder<TDAO>` `AndWhere<TargetDao>(Func<TargetDao,object[]>) where TargetDao : QueryDao`	Create a query condition that forms logic and operations with the previous condition. The all parameters have the same meaning as the Where<TargetDao>(...) method.
`public QueryBuilder<TDAO>` `OrWhere<TargetDao>(Func<TargetDao,object[]>) where TargetDao : QueryDao`	Create a query condition that forms logic or operations with the previous condition. The all parameters have the same meaning as the Where<TargetDao>(...) method.

`public QueryBuilder<TDAO>` `Select(Func<IncludeQueryBuilder<TDAO>, IDescription[]>)`	Select the fields you want to look up in the query. Func<IncludeQueryBuilder<TDAO>, IDescription[]> param: used to build and return fields to be looked up in a query.
`public QueryCommandExecuter` `Build(Action<IncludeQueryBuilder<TDAO>, SelectBlock>)`	Build the command and return a query command executor. Action<IncludeQueryBuilder <TDAO>, SelectBlock> param: more options for setting query commands, such as sorting, grouping, paging, and so on. The default value is null, and null does not append any superfluous query options.
`public int Count(object = null)`	Returns the number of all records that match the query. object param: The transaction controller (DBTransactionController) or BatchCommander object that performs the deletion.

4.2.4: QueryCommandExecuter class

Definition

Namespace: Wunion.DataAdapter.Kernel.Querying
Assembly: Wunion.DataAdapter.NetCore.dll
Package: Wunion.DataAdapter.NetCore   v1.0.5.1

Note

Used to execute query commands and parse query results. The actuator was created by QueryBuilder<TDAO>.Build(...) method, you can't manually create an object instance of the command executor in other scenarios.

Constructors

Protected by assembly

Methods

`public object ExecuteScalar(object = null)`	Executes the query and returns the data of the first column of the first row in the results. object param: The transaction controller (DBTransactionController) or BatchCommander object in which the command is executed, the default value null means that it is not executed in the transaction or BatchCommander.

`public async Task<object>` `ExecuteScalarAsync(object = null)`	Executes the query and returns the data of the first column of the first row in the results (Asynchronous mode).
`public List<T> ToList<T>(object = null) where T :` `class, new()`	Executes the query and returns the query results as a collection of objects of the specified type. The generic parameter T means parsing a single-row record to an object of that type, but note that it is not an entity type. object param: The transaction controller (DBTransactionController) or BatchCommander object in which the command is executed, the default value null means that it is not executed in the transaction or BatchCommander.
`public async Task<List<T>> ToListAsyn<T>(object =` `null) where T : class, new()`	Executes the query and returns the query results as a collection of objects of the specified type (Asynchronous mode).

`public List<TEntity>`
`ToEntityList<TEntity>(object = null) where`
`TEntity : class, new()`
	Executes the query and returns the query results to the specified entity collection.
	The generic parameter TEntity represents the entity type corresponding to record.
	object param: The transaction controller (DBTransactionController) or BatchCommander object in which the command is executed, the default value null means that it is not executed in the transaction or BatchCommander.

`public async Task<List<TEntity>>`
`ToEntityListAsync<TEntity>(object = null) where`
`TEntity : class, new()`
	Executes the query and returns the query results to the specified entity collection (Asynchronous mode).

`public List<dynamic> ToDynamicList(object = null,` `Func<string, object, Type, object> = null)`	Executes the query and returns the query results as a Collection of Wunion.DataAdapter.Kernel.DataCollection.DynamicEntity. object param: The transaction controller (DBTransactionController) or BatchCommander object in which the command is executed, the default value null means that it is not executed in the transaction or BatchCommander. Func<string, object, Type, object> param: The default value is null, which is used to convert the values of individual fields when populating records as DynamicEntity dynamic entity objects.

| ```csharp
public async Task<List<dynamic>>
ToDynamicListAsync(object = null, Func<string,
object, Type, object> = null)
``` | Asynchronous mode of the ToDynamicList(...) method. |

4.3: Define the database context

Create a class and inherit DbContext to define the database context that belongs to you, you must define the tables in this DbContext, and implement the configuration of CodeFirst.

4.3.1: Define the entity

An entity object corresponds to a row of records in a table in the database, and the entity class does not require inheritance from a specific class, but you can sort out the inheritance relationship between them according to your own needs. For example, if many records in a table need fields for the date of creation and the date of last modification, you can define an abstract entity class to implement these two fields, and then inherit the type from all the entities that need it.

All entities must declare their corresponding table structures by adding TableFieldAttribute, IdentityAttribute, ForeignKeyAttribute to the public property of the entity class.
Here's example:

```csharp
public abstract class DateTimeEntity
{
    [GenerateOrder(160)]
    [TableField(DbType = GenericDbType.DateTime, NotNull = true)]
    public DateTime Creation { get; set; }

    [GenerateOrder(161)]
    [TableField(DbType = GenericDbType.DateTime)]
    public DateTime? LastModified { get; set; }
}

public class UserAccount : DateTimeEntity
{
    [Identity(0, 1)]
    [TableField(DbType = GenericDbType.Int, PrimaryKey = true, NotNull = true)]
    public int UID { get; set; }

    [TableField(DbType = GenericDbType.VarChar, Size = 32, NotNull = true, Unique = true)]
    public string Name { get; set; }

    [TableField(DbType = GenericDbType.VarChar, Size = 64, NotNull = true)]
    public string Password { get; set; }

    [TableField(DbType = GenericDbType.Int, Default = (int)UserAccountStatus.Enabled, NotNull = true,
    ValueConverter = typeof(UserAccountStatusConverter))]
    public UserAccountStatus Status { get; set; }

    [TableField(DbType = GenericDbType.Text, NotNull = true, ValueConverter =
    typeof(IntegerCollectionConverter))]
    public List<int> Groups { get; set; }
        ......
}
```

TableFieldAttribute: Used for table field declarations, where only public properties that contain this declaration are mapped to fields in database tables. The following table lists all the properties of the attribute and their descriptions:

Name	Type of: string, The field name. Not specifying this attribute indicates that the field name is the same as the attribute name of the entity.
DbType	Type of: GenericDbType, The data type of the field. Must be specified.
Size	Type of: int, The length of the field
PrimaryKey	Type of: bool, true indicates that the field is primary. The default value is false

NotNull	Type of: bool, true indicates that the field is not allowed to be null. The default value is false
Unique	Type of: bool, A unique value constraint, The default value is false and cannot be declared at the same time as PrimeKey.
Default	Type of: object, The default value of the field.
ValueConverter	Type of: Type, The value converter type for the field, which must implement the DbValueConverter\<TSource> abstract class. When this property is not specified and the type of the entity property is not in the scope supported by the database, the ORM automatically searches for registered converters from the DataEngine object to which the database context (DbContext) belongs, and throws a database exception if the converter of the relevant type is not searched. How to implement and register a value converter will be explained in subsequent sections.

IdentityAttribute: Used to declare auto-increment of fields, not necessary. All the properties of the attribute and their descriptions are listed in the following table:

InitValue	Type of: int, The starting value.
Increment	Type of: int, Increment step value.

ForeignKeyAttribute: Foreign key constraints for declaring fields, not necessary. All the properties of the attribute and their descriptions are listed in the following table:

TableName	Type of: string, The table name of the master table.
Field	Type of: string, The name of the field in the master table.
OnDeleteAction	Type of: string, The behavior when a record in the master table is deleted. Values range: ForeignKeyAttribute. ACTION_CASCADE ForeignKeyAttribute. ACTION_SET_NULL ForeignKeyAttribute. ACTION_RESTRICT The default value: ForeignKeyAttribute. ACTION_CASCADE
OnUpdateAction	Type of: string, The behavior when a record in the master table is updated. Values range: ForeignKeyAttribute. ACTION_CASCADE ForeignKeyAttribute. ACTION_SET_NULL ForeignKeyAttribute. ACTION_RESTRICT The default value: ForeignKeyAttribute. ACTION_CASCADE

GenerateOrderAttribute: Used to declare the order in which tables or fields are generated, not required. The attribute has only one Index property and may be initialized by a constructor. start with 0 and allowing repetition. When you want to control the generation order of the table or fields, you can do so by adding this attribute declaration. And using this attribute to declare the order in which the primary and slave tables are generated when foreign key constraints exist in the entire database context definition,

4.3.2: Define tables in the DbContext

Define a public read-only property of the DbTableContext<TEntity> type in your DbContext and have the get accessor call the TableDeclaration<TEntity>(string tableName) method, as follows:

```csharp
public class MyDbContext : DbContext
{
    public MyDbContext(DataEngine engine) : base(engine)
    { }

    /// <summary>
    /// User accounts table.
    /// </summary>
    [GenerateOrder(1)]
    public DbTableContext<UserAccount> UserAccounts => TableDeclaration<UserAccount>("UserAccounts");

    /// <summary>
    /// User groups table.
    /// </summary>
    [GenerateOrder(0)]
    public DbTableContext<UserGroup> UserGroups => TableDeclaration<UserGroup>("UserGroups");
}
```

TableDeclaration<TEntity> method: This method is used to define a table in the database context and return the context object for that table. The method does not produce duplicate table definitions in the same database context object.

Split multi-tables support for big data storage: When some data is very large, you may want to split them into multiple tables for storage. The way to split may be horizontal or portrait, take portrait splitting as an example. At first, there will only be one empty table, and when the number of records in this table reaches 3,000,000 or 5,000,000, the program create a new table by calling the DbContext.CreateTable<TEntity>(string tableName) method and stores the records in the newly created table. The next time you use this newly created table, you can get its table context object by calling the DbContext.TableDeclaration<TEntity>(string tableName) method.

4.3.3: Configure the CodeFirst options

The abstract method OnBeforeGenerating(IGeneratingOptions options) must be implemented in the DbContext definition to configure the generated behavior for the CodeFirst tool. The following example code shows all the configurations of CodeFirst:

```csharp
public class MyDbContext : DbContext
{
    public MyDbContext(DataEngine engine) : base(engine)
    { }
    ......

    public override void OnBeforeGenerating(IGeneratingOptions options)
    {
        options.Database.ConnectionString = "……"; // Setup the database ConnectionString.
        options.Database.SchemaVersion = 1; // The current database schema version.
        // Whether to force a rebuild when the table already exists in the database
        //(when the entity is modified, the value is set to true, otherwise the database is not updated).
        options.Database.ReCreateExistingTable = false;
        // Implements the option to back up and restore data during table rebuilding .this option).
        options.TableUpgradeMigrator = new MyTableUpgradeMigrator();
        // The namespace used by the query DAO generated for the entity.
        options.DaoGenerateNamespace = "Wunion.DataAdapter.CodeFirstDemo.Data.Domain";
        // The relative path generated by the query DAO
        // (starting from the root directory of the project that defines the class).
        options.DaoGenerateDirectory = System.IO.Path.Combine("Domain", "DAO");
    }
}
```

Pre-set Data: When CodeFirst generates the database, it is necessary to provision certain necessary data to the database, such as requiring at least one user account to log in to the system or other systems to run the necessary data, of course, this is not necessary. Override the OnGenerateCompleted(Action<string> log) method to achieve the above purposes, for example:

```csharp
public class MyDbContext : DbContext
{
    public MyDbContext(DataEngine engine) : base(engine)
    { }
    ……

    public override void OnGenerateCompleted(Action<string> log)
    {
        using (BatchCommander batch = new BatchCommander(DbEngine))
        {
            DateTime creation = new DateTime(2022, 2, 22, 22, 57, 49);
            try
            {
                // Pre-set user account groups.
                UserAccountGroup group = new UserAccountGroup {
                    Id = 100,
                    Name = "Administrators",
                    Description = "Admin users",
                    Creation = creation
                };
                UserGroups.Add(group, batch);
                group.Id = Convert.ToInt32(batch.SCOPE_IDENTITY);
                // Pre-set supper user.
                log("Super-user is being provisioned ...");
                UserAccount ua = new UserAccount {
                    Name = "super-admin",
                    Password = "mE9nJTgxBo3lDGJOg47LzX42a89K+LvjbeAGQyfpG5k=",
                    Status = UserAccountStatus.Enabled,
                    Groups = new List<int>(new int[] { group.Id }),
                    User = "巨陽道君",
                    Email = "cnzhnet@hotmail.com",
                    Creation = creation
                };
                UserAccounts.Add(ua, batch);
                log("The data pre-set completed.");
            }
            catch (Exception Ex)
            {
                log(Ex.Message);
            }
        }
    }
}
```

4.4: Generate databases and DAOs

After completing the work described in Subsection 4.3, use the CodeFirst tool to generate the database schema and the DAO classes corresponding to each entity. Open Terminal in Visual Studio 2019 or 2022 or VS Code: View (V) - > Terminal
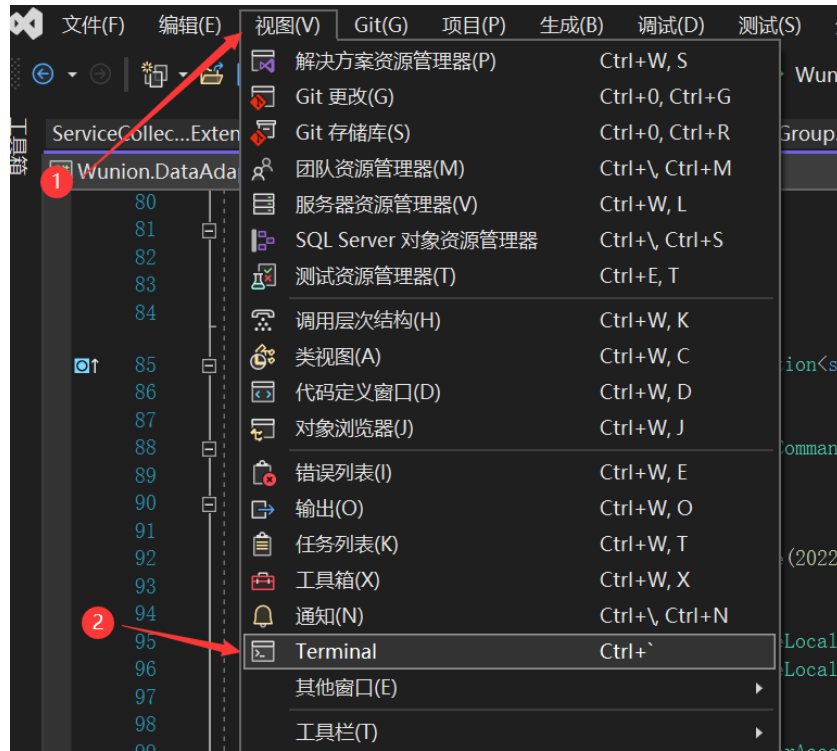
Successfully compile the project using Debug AnyCPU mode, and use the cd command to this project directory, as shown in the following figure:
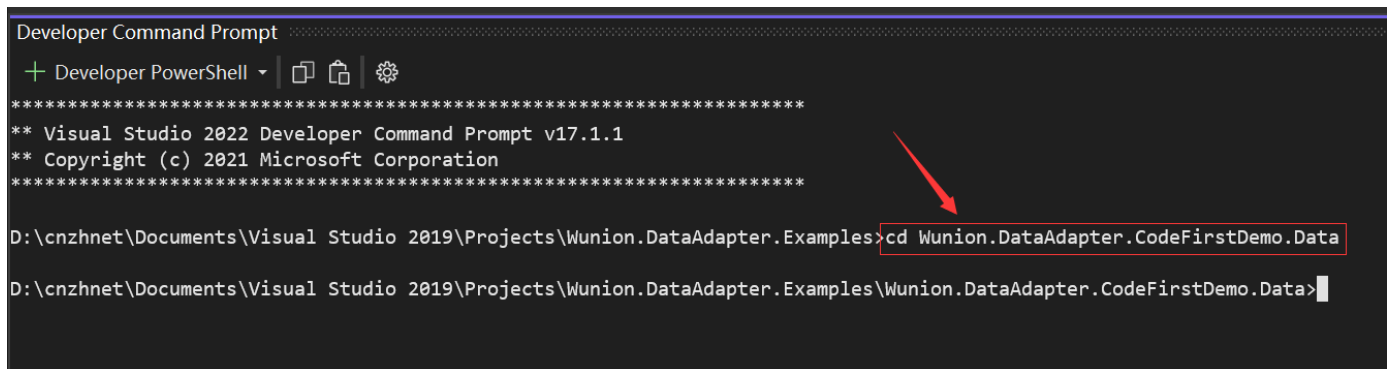
Execute: dotnet wda-cf /local:en-US <mssql | npgsql | mysql | sqlite3>

The tool name WDA-CF is not case-sensitive, the tool supports Chinese Simplified and English, the default is Chinese Simplified. Command format: dotnet /local:<zh-CN | en-US> <db-kind>
where /local: is an optional parameter that, when not specified, outputs log information in Chinese Simplified.
<db-kind> indicating the supported database type codes. The database type code is described as follows:
    mssql   Microsoft SQL Server database

mysql   MySQL database, and the default storage engine is InnoDB, which can be explicitly specified, such as mysql:InnoDB, mysql:MyISAM

npgsql   PostgreSQL database

sqlite3   sqlite3 database

4.5: Insert/Update/Delete

Insert

```
MyDbContext db = ……
UserAccount ua = new UserAccount {
    Name = "super-admin",
    Password = "mE9nJTgxBo3lDGJOg47LzX42a89K+LvjbeAGQyfpG5k=",
    Status = UserAccountStatus.Enabled,
    Groups = new List<int>(new int[] { 0 }),
    User = "巨陽道君",
    Email = "cnzhnet@hotmail.com",
    Creation = creation
};
int result = db.UserAccounts.Add(group);
//int result = await db.UserAccounts.AddAsync(group);
```

Update

```
public void UpdateGroup(UserGroup group)
{
    dbContext.UserGroups.Update(group);
    //await db.UserGroups.UpdateAsync(group);
}
```

Delete

```
using UADAO = Wunion.DataAdapter.CodeFirstDemo.Data.Domain.UserAccountDao;
using UGDAO = Wunion.DataAdapter.CodeFirstDemo.Data.Domain.UserGroupDao;
......

public void DeleteGroup(int gid)
{
    // Execute delete in BatchCommander
    using (BatchCommander batch = new BatchCommander(dbContext.DbEngine))
    {
        dbContext.UserAccounts.Delete<UADAO>(p => new object[] { p.Group.Like($"{gid};") }, batch);
        dbContext.UserGroups.Delete<UGDAO>(p => new object[] { p.Id == gid }, batch);
    }
    /* Execute delete in transaction: See Chapter 2, Subsection 2.6, for use of transaction
    using (DBTransactionController trans = dbContext.DbEngine.BeginTrans())
    {
        dbContext.UserAccounts.Delete<UADAO>(p => new object[] { p.Group.Like($"{gid};") }, trans);
        dbContext.UserGroups.Delete<UGDAO>(p => new object[] { p.Id == gid }, trans);
        trans.Commit();
    }
    */
}
```

4.6: Query Data

Using the QueryBuilder<TDAO> class to build and execute queries, the generic type TDAO represents the data query accessor corresponding to the first table in the query, Invoke the QueryBuilder<TDAO>.Create(IDbTableContext table) Static method create a query.

4.6.1: Query from a single table

```
// Execute: Synchronous mode.
List<UserAccount> queryResult = QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
            .Select(p => p.First.All)// p.First Represents the DAO of the first table in the query.
            .Where<UserAccountDao>(p => new object[] {
                // Uncomments is multiple logical conditions —↓
                p.Group.Like("0,", LikeMatch.Center)// & p.Name == "cnzhnet"
                // For more information about the syntax of query conditions, see Chapter 2.3
            }).Build((p, options) => {
                options.OrderBy(p.First.UID, OrderByMode.ASC); // sorting.
                options.Paging(pageSize, currentPage); // pagination.
            })
            .ToEntityList<UserAccount>();
/* Description: The Select method is used to determine the fields to be queried, any query must specify the
    fields required in the query results, and the Build method is used to build the query and return the query's
    command executor, QueryCommandExecuter, see subsection 4.2.4 for QuerycommandExecuter
*/
// Execute: Asynchronous mode.
List<UserAccount> queryResult = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
            .Select(p => p.First.All)
            .Build((p, options) => {
                options.OrderBy(p.First.UID, OrderByMode.ASC); // sorting.
                options.Paging(pageSize, currentPage); // pagination.
            })
            .ToEntityListAsync<UserAccount>();
```

4.6.2: Query from multiple tables

Build and execute queries based on the DbTableContext<TEntity> object of a table.   Use QueryBuilder<TDAO>.Include<TForeignDao>(…) method to join tables, the Include method eventually using the LOFT JOIN associate multiple tables, and the query results usually contain fields from multiple tables when querying multiple tables. There is no specific entity to hold the records in the query results, in which case you need to define a data model or use a dynamic entity type (DynamicEntity).

In this ORM, no implement navigation property similar to that in the Entity Framework to hold the query results.

```
List<UserDataModel> users = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
            .Include<UserGroupDao>(p => { // p.First Represents the DAO of the first table in the query.
                p.First.Group == p.tbl<UserGroupDao>().Id
            })
            .Select(p => p.First.All.Concat(new object[] {
                p.tbl<UserGroupDao>().Name.As("GroupName"),
                p.tbl<UserGroupDao>().Permissions
            }))
            .Where<UserAccountDao>(p => new object[] { // Condition of the first table.
                p.Group == groupId
            }).AndWhere<UserGroupDao>(p => new object[] {
                // Conditions for the second table (If not, you don't need it).
                ……
            })
            .Build((p, options) => {
                options.OrderBy(p.First.UID, OrderByMode.ASC); // sorting.
                options.Paging(pageSize, currentPage); // pagination.
            }).ToListAsync<UserAccount>();
```

4.6.3: Query results hold to data model or DynamicEntity

In some cases, a query does not always need to look up all the fields in the table, or when a multi-tables query results contain fields from multiple tables. There is no specific entity to hold the records in the query results, in which case you need to define a data model or use a dynamic entity type (DynamicEntity). In QueryCommandExecuter object, the ToList, ToListAsync method is provided to return the query results directly as a collection of the data model objects. And the property members of the data model object, must contain at least one property with the same name as the field in the query results.

```
UserDataModel queryResult = QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
            .Select(p => new object[] {
                p.First.UID, p.First.Name, p.First.Password
            }).Where<UserAccountDao>(p => new object[] {
                p.First.Name == loginName |
                p.First.PhoneNumber == loginName |
                p.First.Email == loginName
            }).Build()
            .ToList<UserDataModel>()
            .FirstOrDefault();
```

Dynamic entities as results of the query: The DynamicEntity class is defined in the namespace Wunion.DataAdapter.Kernel.DataCollection. DynamicEntity dynamically generate property members of entity objects based on the fields in the query results, which eliminates the need to define the data model too much. The following example demonstrates returning query results with dynamic entities:

```
List<dynamic> users = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
            .Include<UserGroupDao>(p => {
                p.First.Group == p.tbl<UserGroupDao>().Id
            })
            .Select(p => p.First.All.Concat(new object[] {
                p.tbl<UserGroupDao>().Name.As("GroupName"),
                p.tbl<UserGroupDao>().Permissions
            }))
            .Where<UserAccountDao>(p => new object[] {
                p.Group == groupId
            })
            .Build((p, options) => {
                options.OrderBy(p.First.UID, OrderByMode.ASC); // sorting.
                options.Paging(pageSize, currentPage); // pagination.
            })
            .ToDynamicListAsync(converter: (field, value, fieldType) => {
                // Param converter:  used to convert the value of the field.
                switch (field)
                {
                    case "GroupName":
                        return JsonConvert.Deserialize<List<int>>(value);
                    case "Status":
                        return (UserAccountStatus)((int)value);
                    default:
                        return value;
                }
            });
```

4.7: Execute custom SQL command

Using DbCommandBuilder to create SQL commands saves you from having to worry about the differences in SQL commands for different kinds of databases that cause the program to run incorrectly. Pass the DbCommandBuilder object to the Corresponding method of the BatchCommander object or the DBTransactionController transaction controller or DbContext.DbEngine. For more information about creating SQL commands using DbCommandBuilder, see Chapter 2, and the following code shows how to execute custom SQL commands:

```
DbCommandBuilder cb = new DbCommandBuilder();
cb.Update(dbContext.UserAccounts.TableName)
  .Set(td.Field("Password") == newPassword)
  .Where(td.Field("UID") == 0);
// Execute on the DbContext.DbEngine.
int result = dbContext.DbEngine.ExecuteNoneQuery(cb);

// Execute on the Transaction.
using (DBTransactionController trans = dbContext.DbEngine.BeginTrans())
{
    result = trans.DBA.ExecuteNoneQuery(cb);
}

// Execute on the BatchCommander.
using (BatchCommander batch = new BatchCommander(dbContext.DbEngine))
{
    result = batch.ExecuteNoneQuery(cb);
}


See Example project:
https://github.com/cnzhnet/Wunion.DataAdapter.Examples/blob/main/Wunion.DataAdapter.CodeFirstDemo.Appli
cation/AppService.cs
```

4.8: Implementation and registration of value converters

Value converters are used to convert custom data (classes or enumerations) into database-supported data types when inserting or updating records into the database, and value converters can be registered on the global DataEngine object beforehand when the application starting, or specified on the property of an entity class (See subsection 4.3.1). And When querying, it is also necessary for the value converter to convert the field values in the database to the type of the corresponding attribute of the corresponding entity.

4.8.1: Implement value converter of a custom type

Inheriting the DbValueConverter<TSource> abstract class to create a value converter of a specific type, the following example demonstrates a value converter of the user account state enumeration type UserAccountStatus:

```csharp
public class UserAccountStatusConverter : DbValueConverter<UserAccountStatus>
{
    /// <summary>
    /// Convert to a type supported by the database.
    /// </summary>
    /// <param name="value"></param>
    /// <param name="dest"></param>
    /// <param name="buffer"></param>
    protected override void ConvertTo(UserAccountStatus value, Type dest, out object buffer)
    {
        if (dest == typeof(string)) // Convert to string.
        {
            buffer = Enum.GetName<UserAccountStatus>(value);
            return;
        }
        // Convert to numeric.
        buffer = (int)value;
    }

    /// <summary>
    /// Convert from a database-supported type to an enumeration.
    /// </summary>
    /// <param name="value"></param>
    /// <param name="buffer"></param>
    protected override void Parse(object value, ref UserAccountStatus buffer)
    {
        // From numeric.
        if (value.GetType() == typeof(int))
        {
            int numeric = Convert.ToInt32(value);
            buffer = (UserAccountStatus)numeric;
            return;
        }
        // From string.
        string name = value.ToString();
        buffer = Enum.Parse<UserAccountStatus>(name);
    }
}
```

4.8.2: Register the value converter in to DataEngine

The following uses asp.net core as an example to demonstrate how to register a value converter with the global DataEngine:

```csharp
public static class ServiceCollectionExtensions
{
    private static MyDbContext db;
    public static void AddDbContext(this IServiceCollection services,
                                    Action<DbValueConverterOptions> configure)
    {
        DbValueConverterOptions DbConverterOptions = DataEngine.CreateConverterOptions();
        configure(DbConverterOptions);
        DataEngine dbEngine = new DataEngine(
            new Wunion.DataAdapter.Kernel.SQLServer.SqlServerDbAccess(),
            new Wunion.DataAdapter.Kernel.SQLServer.CommandParser.SqlServerParserAdapter()
        );
        dbEngine.ConfigureValueConverter(converterOptions);
        db = new MyDbContext(dbEngine);
        services.AddSingleton<MyDbContext>(db);
    }
}


public static class ApplicationBuilderExtensions
{
    public static void UseDbContext(this IApplicationBuilder app, Action<DataEngine> configure)
    {
        MyDbContext context = app.ApplicationServices.GetService<MyDbContext>();
        if (context == null)
            throw new Exception("……");
        configure?.Invoke(context.DbEngine);
    }
}


// The Startup class is the default startup configuration for asp.net core.
public class Startup
{
    ……
    public void ConfigureServices(IServiceCollection services)
    {
        ……
        services.AddDbContext((options =>) { // Register the value converter here.
            options.Add(typeof(UserAccountStatus), new UserAccountStatusConverter());
            //options.Add(typeof(List<int>), new IntegerCollectionConverter());
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
```

```
      ……
    app.UseDbContext((dbEngine) => {
        dbEngine.DBA.ConnectionString = "……";
        dbEngine.UseDefaultConnectionPool((pool) => {
            pool.RequestTimeout = TimeSpan.FromSeconds(10);
            pool.ReleaseTimeout = TimeSpan.FromMinutes(2);
            pool.MaximumConnections = 20;
        });
    });
  }
}
```