

# Wunion.DataAdapter.NetCore 开发人员文档

轻量级的数据库查询组件库，该组件将不同种类的数据库 SQL 命令以 C# 对象结构的方式进行封装。使开发人员进行数据库操作时不必关心不同类型数据库的命令差异对代码的直接影响，而更加专注于业务代码的实现。该组件库采用 .NET Standard 2.0 跨平台标准进行设计，同时支持 .NET Framework 4.6.2 以上版本及 .NET Core 2.0 以上版本的各类 .NET 应用程序。利用 .NET Core 的跨平台特性，使得该组件库可在 Windows、Mac OSX 及 Linux 系统上运行。

该项目完全免费，组件库的版权归开发者本人所有。任何企业或个人皆可自由使用，但禁止除开发者以外的任何企业或个人以任何形式将该组件纳入自己的知识产权范围。使用该组件引起的任何商业损失、利益纠纷，其刑事或民事责任一概由使用者自行承担。

开发环境：Microsoft Visual Studio 2017

开发语言：C#

目标架构：.NET Standard 2.x

库名称：Wunion.DataAdapter.NetCore

根命名空间：Wunion.DataAdapter.Kernel

## 第一章、项目的软件包结构说明

- Wunion.DataAdapter.NetCore 核心主项目，SQL 对象化的核心实现层。
- Wunion.DataAdapter.NetCore.SqlServer Microsoft SQL Server 2008 R2 SP3 及以上版本的数据库支持包。
- Wunion.DataAdapter.NetCore.SQLite3 SQLite 数据库的支持包。
- Wunion.DataAdapter.NetCore.MySQL MySQL 数据库的支持包（在 MariaDB 10.2 进行测试并通过）。
- Wunion.DataAdapter.NetCore.PostgreSQL PostgreSQL 数据库的支持包（在 PostgreSQL 10 进行测试并通过）。
- Wunion.DataAdapter.NetCore.EntityUtils 数据库实体模型支持包（配实体模型代码生成器工具）

## 第二章、使用 Wunion.DataAdapter.NetCore 查询数据库

### 2.1、使用组件库必须要了解的类

#### 2.1.1、DataEngine 类

该类提供对目标数据库进行增、删、改、查的访问操作，以及事务管理。一个 DataEngine 类型的对象实例总是与具体某种类型的数据库引擎相关，同时该类型还提供了多个 DataEngine 数据引擎实例的静态缓存池管理功能。

**初始化 DataEngine 引擎的实例：**通过公共的构造函数来创建并初始化一个实例。第一个参数 dba 为抽象的 DbAccess 类型，它表示目标类型数据库的访问器，由组件库的各类具体的数据库支持包实现（例如 SqlServerDbAccess 或 NpgsqlDbAccess 等）；第二个参数 ParserAdapter 为 DbCommandBuilder 命令对象树的解释适配器，其同样由组件库的各类具体的数据库支持包实现。

**AppendDataEngine 静态方法：**此方法在静态缓存池中创建一个 DataEngine 引擎实例，前两个参数的意义与构造函数一致，第三个参数表示数据引擎实例在缓存中的键名称，默认为 Default。

**GetEngine 静态方法：**获得缓存池中指定键的数据引擎实例，当一个应用中同时使用多种数据库时，可利用缓存池轻松实现多种数据库的交互。

**RemoveEngine 静态方法：**从静态缓存池中删除具有指定键名称的数据引擎。

**CurrentEngine 静态属性：**该属性返回静态缓存池中默认（即键名称为 Default）的数据实例。

#### 2.1.2、DbCommandBuilder 类

用于构建 SQL 命令的对象树，它提供 SELECT、INSERT、UPDATE 及 DELETE 等常规 SQL 命令的对象树构建方法，并支持无限层次的嵌套查询。存储过程无法使用 DbCommandBuilder 类进行构建。使用 DataEngine 执行命令时皆以该类型的对象作为参数，其内部将 DbCommandBuilder 对象树解释为目标数据库的 SQL 命令并执行。

#### 2.1.3、td、exp、fm 及 Fun 静态快捷类

- td 用于创建命令中的字段元素
- exp 用于创建命令中的表达式元素、及 LIKE 子句元素（大部份已由运算符重载完成，勿需手动调用该快捷类）
- fm 用于创建命令中的表元素及 LeftJoin 元素
- Fun 用于创建命令中的函数元素

## 2.2、引用及初始化组件库

**NuGet 包的引用：**

- Wunion.DataAdapter.NetCore 核心库（必须引用）  
命名空间：Wunion.DataAdapter.Kernel  
Wunion.DataAdapter.Kernel.DbInterop  
Wunion.DataAdapter.Kernel.CommandBuilders
- Wunion.DataAdapter.NetCore.SqlServer Microsoft SQL Server 数据库支持包（按需引用），仅支持 SQL Server 2008 R2 SP3 及以上版本，低版本则无法连接数据库。  
命名空间：Wunion.DataAdapter.Kernel.SqlServer  
Wunion.DataAdapter.Kernel.SqlServer.CommandParser
- Wunion.DataAdapter.NetCore.SQLite3 SQLite 数据库支持包（按需引用）  
命名空间：Wunion.DataAdapter.Kernel.SQLite3  
Wunion.DataAdapter.Kernel.SQLite3.CommandParser
- Wunion.DataAdapter.NetCore.PostgreSQL PostgreSQL 数据库支持包（按需引用）  
命名空间：Wunion.DataAdapter.Kernel.PostgreSQL  
Wunion.DataAdapter.Kernel.PostgreSQL.CommandParser
- Wunion.DataAdapter.NetCore.MySQL MySQL 数据库支持包（按需引用）  
命名空间：Wunion.DataAdapter.Kernel.MySQL  
Wunion.DataAdapter.Kernel.MySQL.CommandParser

**初始化数据引擎：**根据 NuGet 包的引用情况 using 引入相关的命名空间，初始化代码如下

```
SqlServerDbAccess DBA = new SqlServerDbAccess();  
DBA.ConnectionString = "Server=(local);Database=test;User ID=sa;Password=*****";  
DataEngine.AppendDataEngine(DBA, new SqlServerParserAdapter()); //添加为默认引擎
```

上例为使用 DataEngine 提供的数据库引擎静态缓存的方式，若不使用此方式，则可自定义 DataEngine 实例的全局化。

## 2.3、数据查询的语法

在本章所有的示例代码中 DataEngine.CurrentEngine 表示一个数据引擎实例，在实际的项目情况中并非总是要按此机制。当实际的项目中采用其它手段来全局化 DataEngine 实例时，以具体的 DataEngine 实例为准。

DataEngine 为查询数据提供了 ExecuteQuery、ExecuteQuery<DataTable>、ExecuteReader 及 ExecuteScalar 四个方法来分别应对不同场景的查询需求。当在执行查询过程中产生错误时，以上四个方法将返回 null 值，并且在 DataEngine.DBA.Error 属性中提供了错误信息及引发错误的目标 SQL 命令原型来为程序的调试工作提供参考依据。

### 2.3.1、单表查询

```
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);  
// 查询所有字段请使用：td.Field("*")
```

复杂的查询及其条件：例如下面的 SQL 原型

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

DbCommandBuilder 实现方式如下

```
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)
    );
```

命令对象树中的元素优先级分组：从下面的 SQL 命令原型来理解何谓优先级分组

```
SELECT ([Field1] + [Field2]) * [Field3]
```

```
FROM [TableName]
```

```
WHERE [Field5] = FALSE AND ([Field4] IS NULL OR LEN([Field4] < 1))
```

其中 SELECT 段中的四则运算表达式中的加法运算优先级，在 DbCommandBuilder 上的实现方式如下

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

在 WHERE 条件中的逻辑运算表达式后段若无分组，其查询结果是不正确的。逻辑运算的优先级分组实现与四则运算同理，在 DbCommandBuilder 的实现方式如下

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

## 2.3.2、多表查询

```
DbCommandBuilder Command = new DbCommandBuilder();
```

```
Command.Select(td.Field("t2", "TestId"), // 查询其中某个表的所字段可用：td.Field("t2", "*")
```

```
    td.Field("t1", "GroupName"),
```

```
    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); // 使用分页功能
```

```
SpeedDataTable dt = DataEngine.CurrentEngine.ExecuteQuery(Command);
```

当在 Microsoft SQL Server 数据库使分页功能时，必须要对查询使用排序，其它数据库则无此要求。多表查询只实现 LEFT JOIN 方式，未实现如 RIGHT JOIN 等其它方式。复合表达式、优先级分组等参见上一小节所述。

### 2.3.3、嵌套查询

```
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、数据的更新/插入/删除

记录的插入、更新和删除皆通过调用 DataEngine 实例的 ExecuteNonQuery 方法执行相应的命令来达到目的。当在执行命令过程中产生错误时，该方法返回 -1 并且在 DataEngine.DBA.Error 属性中提供了错误信息及引发错误的目标 SQL 命令原型来为程序的调试工作提供参考依据。

### 2.4.1、插入记录

```
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.ExecuteNonQuery(Command);
int TestId = Convert.ToInt32(DataEngine.CurrentEngine.DBA.SCOPE_IDENTITY); // 获取插入后自增量字段的值
```

### 2.4.2、更新记录

```
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.ExecuteNonQuery(Command);
```

### 2.4.3、删除记录

```
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、使用 QuickDataChanger 快捷类

使用该类可以不必通过 DbCommandBuilder 来构建插入、更新及删除命令，并且该类提供了将 SpeedDataTable 及 DataTable 数据集批量更新回库的方法。

命名空间：Wunion.DataAdapter.Kernel

### 插入、更新、删除记录

插入及更新记录通过调用 QuickDataChanger.SaveToDataBase 方法进行，该方法定义原型如下

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

参数：

TableName      表名称（数据库表名）  
data            更新或者插入的字段及其值的字典集合  
executeUpdate 当执行更新操作时为 true，插入则为 false

当执行更新或删除操作时，通过 QuickDataChanger 对象的 Conditions 属性来指定条件。

以下代码演示将数据更新到记录，当更新失败时则尝试进行插入操作，若插入操作仍然失败则表示传入的数据不正确。

```
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(); // 执行插入操作时应清除更新的条件  
    result = DC.SaveToDataBase("DATA_TABLE", Data, false);  
}  
// 若此时 result 的值仍然小于 1 则说明传入的数据并不正确。  
// 删除记录则调用 DC.Delete 方法，如下：  
// result = DC.Delete("DATA_TABLE");
```

### 数据集的批量更新

批量更新的机制需要自己定义，以下代码假定在数据集 speedTable 中定义了 STATE 一列来表示每行数据的状态信息，调用 BatchSaveDataTable 方法即可使用这种规则将数据集批量更新至数据库。批量更新应在事务中执行以提高其执行效率及操作的回滚控制，未在事务中执行当然也是可以的但这对性能不利！

```
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" }; //不更新 TestId 字段  
            RowSubmission.Conditions.Add(td.Field("TestId") == Row.Field<int>("TestId")); //设置更新条件  
            break;  
        case 2:
```

```

        RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Insert);
        RowSubmission.NotInsertFields = new string[] { "TestId" }; //不插入自增量字段
        break;
    case 3:
        RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Delete);
        // 设置删除条件
        RowSubmission.Conditions.AddRange(new object[] {
            td.Field("TestId") == Row.Field<int>("TestId") &
            td.Field("GrpId") == Row.Field<int>("GrpId")
        });
        break;
    default:
        RowSubmission = new DataRowSubmission(QuickDataChangerSubmission.Ignore);
        break;
    }
    return RowSubmission;
});
trans.Commit();
trans.Dispose();

```

## 2.6、使用事务

使用 DataEngine 对象的 BeginTrans() 方法来启动事务，并返回一个 DBTransactionController 事务控制器对象，DBTransactionController 类位于 Wunion.DataAdapter.Kernel.DbInterop 命名空间。

### 在事务中执行命令

在事务中执行命令时，应使用 DBTransactionController.DBA.ExecuteNoneQuery 方法，而非 DataEngine 实例的 ExecuteNoneQuery 方法。如下示例代码：

```

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

```

### 受事务控制的 QuickDataChanger 对象

示例代码：

```

int result = -1;
using (DBTransactionController trans = DataEngine.CurrentEngine.BeginTrans())
{
    QuickDataChanger DC = new QuickDataChanger(trans);
    DC.Conditions.Add(td.Field("GrpId") == 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(); //当分组删除成功时才提交事务
}
```

事务在使用完成后必须被释放。在事务控制范围内的代码应使用 `try...catch` 进行错误处理，当发生异常时应调用事务控制器对象的 `Rollback` 方法进行回滚，若事务控制范围内的所有代码皆正常执行完成时，应手动调用 `Commit` 方法提交事务。在释放事务时，若事务未得到提交则自动进行回滚操作，故，事务的提交必须手动执行。

## 2.7、启用和配置连接池

连接池支持为 1.0.4 版本的新增功能，通过 `DataEngine` 对象实例的 `UseDefaultConnectionPool` 或 `UseConnectionPool` 方法来启用并配置连接池，下面分别介绍这两个方法的区别。

**UseDefaultConnectionPool 方法：** 配置并使用类库自身实现的默认连接池。  
定义形式：`public void UseDefaultConnectionPool(Action<IDbConnectionPool> configure)`  
参数  
`configure` 用于对连接池进行设置的外部方法。

示例：  
`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 方法：** 配置并使用您自己实现的自定义连接池（自定义连接池请按 `Wunion.DataAdapter.Kernel.IDbConnectionPool` 接口规范实现）。  
定义形式：`public void UseConnectionPool(IDbConnectionPool connectionPool)`  
参数  
    `connectionPool` 自定义连接池的对象实例。

**IDbConnectionPool 接口：** 连接池的实现规范接口，其成员描述如下。

属性成员		
RequestTimeout	TimeSpan	获取或设置向连接池请求分配连接的响应超时时间，该时间应以秒种数为判断依据。若超时则应触发异常。
ReleaseTimeout	TimeSpan	获取或设置占用连接的强制释放超时时间，该时间以分钟数为判断依据。一个连接自分配之时起，在该时间内未被释放，连接池则应强制对其进行收回。
MaximumConnections	int	获取或设置连接池的最大连接数量。该设置应根据具体的数据库及服务器操作系统而定。
Count	int	该属性只读，返回连接池中所有连接的数量（包含空闲连接以及正在占用的连接）。
方法成员		
GetConnection(MakeConnectionFactory makeFactory)	IDbConnection	用于向连接池请求一个连接，并返回。 参数： <code>makeFactory</code> 表示当连接池为空时用于创建连接的



		方法，它由各类型的数据库支持包所实现的 DbAccess 提供。其传递由 DataEngine 的内部运作机制完成。
ReleaseConnection(IDbConnection connection)	void	连接池回收指定的连接。该方法只将连接回收到空闲队列中，并不关闭与数据库的连接。

### 第三章、Wunion.DataAdapter.NetCore.EntityUtils 扩展包的使用

该包为 Wunion.DataAdapter.NetCore 类库的实体模型支持扩展包，其依赖于 Wunion.DataAdapter.NetCore 核心基础包。在此包中定义了实体类与数据库表的映射机制。该扩展包的作用旨在帮助开发人员在开发过程中无需对照设计文档进行数据库编程，仅需要通过相应实体的查询代理对象即可完成数据库的增、删、改、查等操作。  
命名空间：Wunion.DataAdapter.EntityUtils

#### 3.1、实体对象的相关基础类型

一个实体对象（数据实体对象）对应于数据表中的一行数据，实体的属性成员与表的字段一一对应。包括名称以及数据类型等。

**DataEntity 类** 该类为所有数据实体的基础类型，所有数据实体都应继承派生自该类。在此类中实现了实体属性成员的动态赋值/取值的标准行为，以及将数据实体输出为字典的方法。

**DynamicEntity 类** 该类为动态实体类，所谓动态实体指的是实体的属性成员根据不同的查询结果动态生成。当开发者进行多表查询时返回的数据集即属于此类实体集合。使用此种类型的实体或实体集合时，应使用 dynamic 关键字来定义实体变量或以 dynamic 关键字作为泛型集合的类型参数，使运行时能够自动推断动态实体的属性成员，而不应使用 DynamicEntity 定义来限制运行时的动态属性成员推断。

**EntityPropertyAttribute 类** 用于实体对象的属性与数据表字段属性的映射，例如：主键字段、自增量字段、是否允许空值、字段默认值等。除 DynamicEntity 类型的数据实体外，其它所有类型的实体的属性成员都必须使用此属性类标记其对应表字段的上述属性特征。

#### 3.2、实体查询的代理对象基础类型

每一个派生于（即继承）DataEntity 类的数据库实体，都对应一个用于辅助创建查询对象树的代理类 Agent 及一个数据表上下文映射代理类 Context。其中查询辅助代理类 Agent 被用于在创建查询的过程中生成实体属性对应的字段或表达式元素，而数据表上下文代理类 Context 则用于创建及执行表的查询（无论增、删、改、查）。Agent 类与 Context 类一起被称之为实体的查询代理。例如有数据表“ProjectInfo”其对应的完整实体及代理类应有如下三个类：

- ProjectInfo 数据实体类，继承自 DataEntity 类
- ProjectInfoAgent 实体的查询辅助代理类，继承自 EntityAgent 类
- ProjectInfoContext 数据表上下文代理类，继承自 TableContext 类

**EntityTableAttribute 类** 用于标记 Agent 类所映射的数据表。

**EntityAgent 抽象类** 所有实体查询辅助类 Agent 的基类（即父类），所有派生于该类的子类型都必须在类上使用 EntityTableAttribute 标记出其对应的数据库中的表。

**TableContext<TEntityAgent>抽象类** 所有数据表上下文映射代理类 Context 的基类。该类为泛型类，泛型类型

参数 TEntityAgent 表示实体查询辅助类 Agent 的具体类型。不同的表应对应其各自的 Agent 类。

上述所有的实体类及代理类，和它们之间的相互联系皆由代码生成器自动生成，使用者并不需要手动去编写它们而只需要大概了解。

### 3.3、DatabaseContext 类的概述

以读写分离的方式实现的数据库上下文对象，通过一个 DatabaseContext 对象实例来进行数据查询及增、删、改操作，这些操作总是与实体及表上下文代理映射相关。该类的构造函数被声明为 protected 受保护级别，故无法直接通过 new 关键字创建它的实例，而需要通过静态方法 Initialize 来初始化并返回一个 DatabaseContext 的对象实例。

当在项目中使用 Wunion.DataAdapter.NetCore.EntityUtils 扩展包进行实体化的数据库交互时，不须进行 DataEngine 实例的初始化及全局化，而只需要对 DatabaseContext 的实例进行全局化即可。原因是在 DatabaseContext 内部分别包含了用于读或写控制的 DataEngine 实例。

#### 3.3.1、DatabaseContext.ContextOptions 类

初始化一个 DatabaseContext 实例所需要的选项。

属性成员	类型	说明
ReadAccess	DbAccess	用于数据库读访问的数据访问器，其中包含连接信息。由 <a href="#">第一章</a> 所讲述的数据库支持包提供。
WriteAccess	DbAccess	用于数据库写访问的数据访问器，其中包含连接信息。由 <a href="#">第一章</a> 所讲述的数据库支持包提供。
Parser	ParserAdapter	内部 DataEngine 实例所需要的命令对象树的解释适配器。由 <a href="#">第一章</a> 所讲述的数据库支持包提供。

#### 3.3.2、初始化 DatabaseContext 实例

以下代码以 PostgreSQL 数据库为例初始化一个 DatabaseContext 实例：

```
DatabaseContext.ContextOptions options = new DatabaseContext.ContextOptions();
options.ReadAccess = new Wunion.DataAdapter.Kernel.PostgreSQL.NpgsqlDbAccess();
// 设置读访问的数据库连接，通常应从配置文件中读取
options.ReadAccess.ConnectionString =
"Host=192.168.1.106;Port=5432;Username=postgres;Password=***;Database=test";
options.WriteAccess = new Wunion.DataAdapter.Kernel.PostgreSQL.NpgsqlDbAccess();
// 设置写访问的数据库连接，通常应从配置文件中读取
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);
// 接下来需要对 DbContext 进行全局化缓存处理，如何全局缓存每个人有自己的喜好，在此不作过多阐述
```

### 3.3.3、启用和配置连接池

连接池支持为 Wunion.DataAdapter.NetCore.EntityUtils 1.0.3 的新增功能，通过调用 DatabaseContext 对象实例的 UseConnectionPool 方法来启用并配置连接池。如下例代码：

```
DbContext.UseConnectionPool((readEngine, writeEngine) => {  
    readEngine.UseDefaultConnectionPool((pool) => {  
        // 在此处配置读的连接池  
    });  
    writeEngine.UseDefaultConnectionPool((pool) => {  
        // 在此处配置写的连接池。  
    });  
});
```

若数据库未进行读写分离则可按如下代码将同一个连接池应用到读及写引擎上：

```
DbContext.UseConnectionPool((read, write) => {  
    IDbConnectionPool defPool = null;  
    read.UseDefaultConnectionPool((pool) => {  
        // 在此处配置连接池。  
        defPool = pool;  
    });  
    write.UseConnectionPool(defPool); // 与读使用同一个连接池。  
});
```

## 3.4、数据库查询

针对数据库的任何增、删、改、查操作皆通过 DatabaseContext 的实例展开。实体代理 [Agent](#) 及 [Context](#) 作为整个查询过程中语法表现的核心。在本节的所有示例代码中 DbContext 表示已实例化的 DatabaseContext 对象。

### 3.4.1、单表查询

针对单张表的查询通过 DbContext.Table<TContext>() 泛型方法展开，该方法的泛型类型参数 TContext 应为特定表的上下文映射代理类，其返回类型即为 TContext 指定的类型的对象实例。以下示例代码演示从工程信息表查出指定用户一周内创建的所有工程：

```
List<ProjectInfo> Result = DbContext.Table<ProjectInfoContext>().Select(p => {  
    // 变量 p 为数据实体对应的查询代理 (即 Agent) 对象  
    // 在该 Lambda 表达式中其类型为 ProjectInfoAgent  
    p.AccountID == accountId & Fun.BetweenAnd(p.CreationDate, DateTime.Now.AddDays(-7), DateTime.Now)  
});
```

若需要进行排序及分页则见下例示例代码：

```
List<ProjectInfo> Result = DbContext.Table<ProjectInfoContext>().Select((p, selector) => {  
    // 变量 p 为数据实体对应的查询代理 (即 Agent) 对象  
    // 在该 Lambda 表达式中其类型为 ProjectInfoAgent  
    // selector 则用于构建查询条件、排序及分页等
```

```

selector.Where(p.AccountID == accountId &
    Fun.BetweenAnd(p.CreationDate, DateTime.Now.AddDays(-7), DateTime.Now)
).OrderBy(p.CreationDate, OrderByMode.DESC) //排序
.Paging(pageSize, currentPage) //分页
});

```

通过上两例查询示例代码可以看出，在 `Select` 方法内通常以 `Lambda` 表达式（即匿名函数）的方式构建查询命令对象树。基本上所有的查询都以该种方式进行。

### 3.4.2、多表联合查询

多表查询通过 `DbContext.From<TEntityAgent>()` 泛型方法展开，该方法的泛型参数 `TEntityAgent` 表示数据实体对应的查询代理(即 `Agent`)类。`DbContext.From` 方法返回一个 `MultiQuerySelector` 类型的复合查询筛选器对象实例，该筛选器用于处理适用于多表查询的命令对象树。多表查询的返回结果总是为 `List<dynamic>`，集合中的单个元素为 `DynamicEntity` 类型的动态实体对象。

联合查询的 `Join` 以 `LEFT JOIN` 的方式实现，未考虑 `RIGHT JOIN` 的实现方式。故，在使用多表查询时应该先整理清楚表的先后逻辑顺序。

示例代码：

```

List<dynamic> Result = DbContext.From<PurchasesAgent>().Join<ProductInfoAgent>()
    .On<PurchasesAgent, ProductInfoAgent>((t0, t1) => {
        // 参数 t0、t1 表示 LEFT JOIN 的前一张及后一张表。
        t0.ProductId == t1.ProductId
    }).Where(t => new object[] {
        // 在 Where 方法中创建查询条件
        // t[0]、t[1].....t[n] 表示前面的 From 及 Join 方法中加入的表，按加入的先后顺序进行索引
        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[] {
        // 在 Select 方法中创建要查出的字段。
        // t[0]、t[1].....t[n] 表示前面的 From 及 Join 方法中加入的表，按加入的先后顺序进行索引
        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 查询

用于返回查询结果中第一行的第一列，所有其他的列和行将被忽略。在某些情况下，需要使用 `COUNT` 函数来统计表或查询的记录数据量时，通过该方法实现。例如在数据分页时需要查询符合条件的记录数量来计算总页数。下面的代码演示从 `Purchases` 订单表中统计一周内产生的所有订单信息：

```

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、插入记录

通过 DbContext.Table<TContext>().Add 方法向对应的表中新增记录，若新增记录失败则引发一个异常。下例代码演示在表 ProductInfo 中新增一条记录：

```

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);
// 若 ProductInfo 表中的 Id 字段为自增量字段，则插入后的 Id 将被自动赋值到 product.Id 属性中。

```

#### 3.4.5、更新记录

通过 DbContext.Table<TContext>().Update 方法更新指定表中的记录，当在更新时未指定更新条件时将引发一个异常。下例代码演示更新 ProductInfo 表中的对应记录：

```

// product 为通过查询得到的某个或新创建的 ProductInfo 数据实体对象。
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[] {
    // 更新条件
    t.Id == productId
});

```

#### 3.4.6、删除记录

通过 DbContext.Table<TContext>().Delete 方法删除指定表中的记录，当未指定删除条件时将引发异常。下例代码演示从 ProductCategories 产品分类信息表中删除一个分类：

```

int result = DbContext.Table<ProductCategoriesContext>().Delete(t => new object[] {
    // 删除条件
    t.CategoryId == categoryId
});
if (result > 0) //成功删除产品分类后应删除此分类下的所有产品信息。

```

```

{
    result = DbContext.Table<ProductInfoContext>().Delete(t => new object[] {
        t.Category == categoryId
    });
}

```

### 3.4.7、使用事务

通过 DbContext 对象的 BeginTransAction()方法启动一个事务，并返回一个 DBTransactionController 事务控制器对象，DBTransactionController 类位于 Wunion.DataAdapter.Kernel.DbInterop 命名空间。

事务在使用完成后必须被释放。释放事务时，若事务未得到提交则自动进行回滚操作，故，事务的提交必须手动执行。下例代码演示使用事务从 ProductCategories 产品分类信息表中删除一个分类：

```

int result = 0;
using (DBTransactionController trans = DbContext.BeginTransAction())
{
    // 删除分类下的所有产品信息.
    result = DbContext.Table<ProductInfoContext>().Delete(trans, t => new object[] {
        t.Category == categoryId
    });
    if (result != -1) //仅在删除分类下的产品信息没有产生错误时删除分类.
    {
        result = DbContext.Table<ProductCategoriesContext>().Delete(trans, t => new object[] {
            t.CategoryId == categoryId
        });
    }
    if (result > 0) //仅在删除成功时提交事务
        trans.Commit();
}

```

插入及更新的事务使用方式同理。

## 3.5、使用代码生成器

代码生成器用于生成给定数据库中的表对应的实体类及代理类，免去开发人员手工编写这些代码的工作（让机器为您编写这些代码）。使用代码生成器甚至让开发者无需理解实体类及查询代理类是如何实现的，只需要使用它们即可。该工具在 WdaEntityGenerator.zip 中提供 Windows 图形界面版及跨平台的命令行版。

### 3.5.1、Windows 图形化代码成器

解压 WdaEntityGenerator.zip 后在 Windows 文件夹内，运行 WdaEntityGenerator.exe 选择语言环境，进入代码生成器主界面，如下图所示：



### 3.5.2、跨平台命令行代码生成器

解压 WdaEntityGenerator.zip 后在 Cross-platform 文件夹内, 命令行代码生成器的语言环境支持简体中文及英文, 默认为英文。在运行命令行代码生成器之前, 首先需在 Linux 或 Mac OSX 上安装 .NET Core SDK, 请参见 Microsoft 官方网站: <https://www.microsoft.com/net/download/linux/build> 下面以 Ubuntu Linux 为例介绍命令行代码生成器的使用。

- 1、使用 cd 命令进入到代码生成器所在目录: `cd /home/cnzhnet/Cross-platform`
- 2、执行命令: `dotnet ./WadEntityGenerator.dll en-US`  
或 `dotnet ./WadEntityGenerator.dll zh-CN`

其中 en-US 为英文语言环境, zh-CN 为简体中文语言环境, 如下图



```
cnzhnet@cnzhnet-ubuntu: ~/Cross-platform
cnzhnet@cnzhnet-ubuntu:~$ cd /home/cnzhnet/Cross-platform
cnzhnet@cnzhnet-ubuntu:~/Cross-platform$ dotnet ./WdaEntityGenerator.dll en-US
=====
Wunion DataAdapter Code Generator for Cross-platform
Version: 1.0.0.0
Locale: English (United States)
Copyright (C) cnzhnet all rights reserved.
-----

Command>: 
```

- 3、执行 help 或 -h 命令可查看该工具支持的所有命令及其帮助信息

```
cnzhnet@cnzhnet-ubuntu: ~/Cross-platform
=====
Wunion DataAdapter Code Generator for Cross-platform
Version: 1.0.0.0
Locale: English (United States)
Copyright (C) cnzhnet all rights reserved.
-----

Command>: help
    connect <ms-sql | sqlite | PostgreSQL | mysql> <ConnectionString>
    set <namespace | output> <option value>
    comments <save | load> <path>
    build <all | -e | -a>
        -e      entity class.
        -a      agent class.
    clear
    exit

Command>: 
```

- 4、执行 connect 命令连接到给定的目标数据库，connect 命令有两个参数选项，第一个参数为目标数据库类型，支持的数据库类型见下列表：

ms-sql        表示 Microsoft SQL Server 数据库  
sqlite        表示 SQLite 数据库  
PostgreSQL    表示 PostgreSQL 数据库  
mysql        表示 MySQL 数据库

下面的命令以 Microsoft SQL Server 为例：

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

成功连接数据库后如下图

```
cnzhnet@cnzhnet-ubuntu: ~/Cross-platform
=====
Wunion DataAdapter Code Generator for Cross-platform
Version: 1.0.0.0
Locale: English (United States)
Copyright (C) cnzhnet all rights reserved.
-----

Command>: connect ms-sql Server=192.168.1.9;Database=Wunion.DataAdapter.NetCore.Demo;User ID=sa;Password=*****
Successfully connected to the database.
Command>: 
```



若未连接成功将会显示错误信息。

- 5、使用 `set` 命令设置生成的代码使用的命名空间

```
set namespace Ling.TestCode.Entities
```

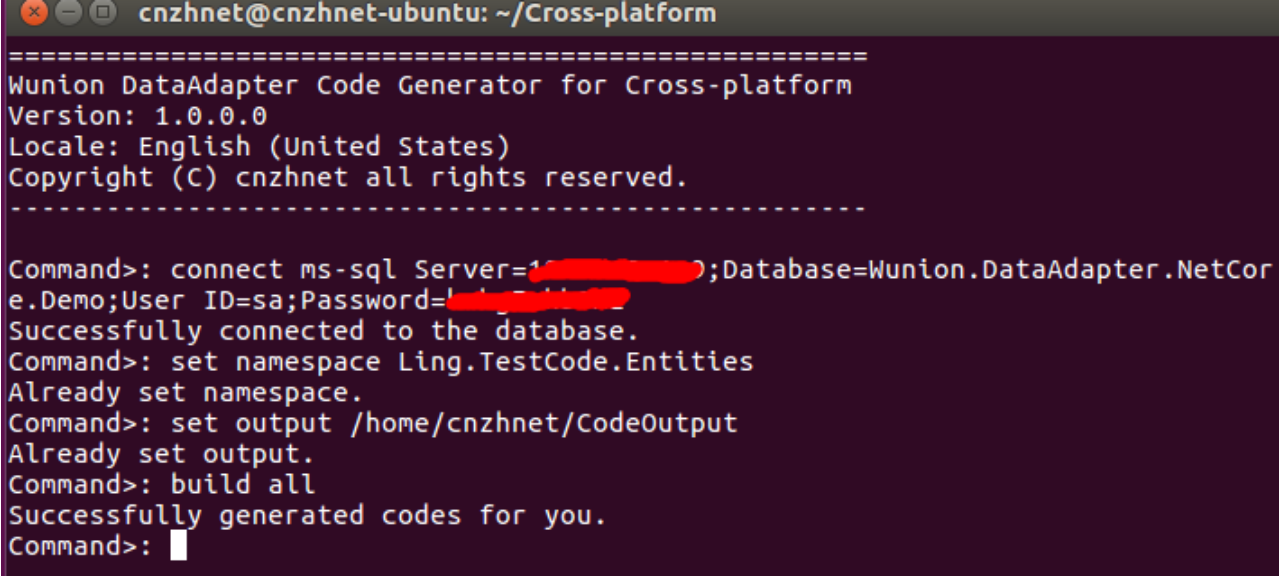
- 6、使用 `set` 命令设置代码的生成路径

```
set output /home/cnzhnet/CodeOutput
```

- 7、使用 `build` 命令生成代码，`build` 命令有一个选项。取值分别为：`all` 或 `-e` 或 `-a`

`all` 表示生成实体类及代理类；`-e` 表示只生成实体类；`-a` 表示只生成代理类 `Agent` 及 `Context`

执行：`build all` 后效果如下图



```
cnzhnet@cnzhnet-ubuntu: ~/Cross-platform
=====
Wunion DataAdapter Code Generator for Cross-platform
Version: 1.0.0.0
Locale: English (United States)
Copyright (C) cnzhnet all rights reserved.
-----

Command>: connect ms-sql Server=192.168.1.100;Database=Wunion.DataAdapter.NetCore.Demo;User ID=sa;Password=123456
Successfully connected to the database.
Command>: set namespace Ling.TestCode.Entities
Already set namespace.
Command>: set output /home/cnzhnet/CodeOutput
Already set output.
Command>: build all
Successfully generated codes for you.
Command>: █
```

- 8、若在创建数据库时未为数据库中的各个表及其字段添加说明信息，则生成的实体类没有代码注释。此时可以通过 `comments` 命令来生成一个 `xml` 代码注释文档，然后使用 `gedit` 或 `nano` 编辑器为表及字段添加注释，注释文档编辑完成保存后再次使用 `comments` 命令来为要生成的代码加载注释信息。若已有注释或不需要代码注释，则至此代码生成的工作结束，在输出目录内可得到您生成的所有代码，不必继续后面的所有步骤。

- 9、执行命令生成注释文档：`comments save /home/cnzhnet/CodeOutput/code_comments.xml`

- 10、在新终端中打开注释文档进行编辑：`nano /home/cnzhnet/CodeOutput/code_comments.xml`



```
cnzhnet@cnzhnet-ubuntu: ~
GNU nano 2.5.3 文件: /home/cnzhnet/CodeOutput/code_comments.xml

<?xml version="1.0" encoding="utf-8"?>
<CodeComments>
  <EntityClass Name="T_TEST1" Comment="The group table.">
    <Member Name="GroupId" Comment="Group ID" />
    <Member Name="GroupName" Comment="The group name" />
  </EntityClass>
  <EntityClass Name="T_TEST2" Comment="The test data table.">
    <Member Name="TestId" Comment="Data ID" />
    <Member Name="GroupId" Comment="The grouping code that the data belongs to." />
    <Member Name="TestName" Comment="Name of the test data." />
    <Member Name="TestAge" Comment="Age of the test data." />
    <Member Name="TestSex" Comment="Gender of the test data." />
    <Member Name="TestPhoto" Comment="Photo of the test data." />
  </EntityClass>
</CodeComments>
```

- 11、执行命令加载代码注释：`comments load /home/cnzhnet/CodeOutput/code_comments.xml`

- 12、按第 7 步执行 `build` 命令重新生成代码（无需删除上一次生成的代码，生成器会自动覆盖）

若要检验生成的代码可执行：`ls` 命令列表输出目录中的文件，并打开任意一个代码文件进行查看如下图

```
cnzhnet@cnzhnet-ubuntu: ~  
GNU nano 2.5.3 文件: /home/cnzhnet/CodeOutput/T_TEST2.cs 已更改  
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
```

## 第四章、使用 CodeFirst ORM

自 Wunion.DataAdapter.NetCore 1.0.5 开始，增加了 CodeFirst 模式的 ORM 支持并正式弃用 Wunion.DataAdapter.NetCore.EntityUtils 包若您希望继续使用 Wunion.DataAdapter.NetCore.EntityUtils 则应选择 Wunion.DataAdapter.NetCore 1.0.4.1 及以前的版本。

示例工程：<https://github.com/cnzhnet/Wunion.DataAdapter.Examples>

### 4.1、安装 CodeFirst ORM

CodeFirst 的支持集成在 Wunion.DataAdapter.NetCore 1.0.5.x 及更高版本的 nuget 包中，请在解决方案的应用程序数据层项目中安装以下 nuget 软件包及工具：

- Wunion.DataAdapter.NetCore 1.0.5.1 及以下
- Wunion.DataAdapter.NetCore.SqlServer 目标数据库支持包  
或 Wunion.DataAdapter.NetCore.MySQL  
或 Wunion.DataAdapter.NetCore.PostgreSQL  
或 Wunion.DataAdapter.NetCore.SQLite3  
请根据您的实际需求安装对应的数据库支持包，不需要全部安装。
- WDA-CF 工具包（CodeFirst 工具：用于生成数据库架构及实体查询数据访问器）  
使用开发者命令提示符或终端 cd 到解决方案的应用程序数据层项目，并执行安装命令：

```
dotnet tool install WDA-CF --version 1.0.1
```

注意：该工具仅支持 .net core 3.1 或 .net 5.0 或 .net 6.0

该项目的目标架构应设置为 .net core 3.1 或 .net 5.0 或 .net 6.0，WDA-CF 将不支持在以 .netstandard 为目标框架的项目中进行 CodeFirst 生成。

### 4.2、CodeFirst ORM 的对象模型概述

整个 ORM 的实现由 DbContext、DbTableContext<TEntity>、QueryDao 以及 QueryBuilder<TDAO>封装构成，各个类的用户说明如下：

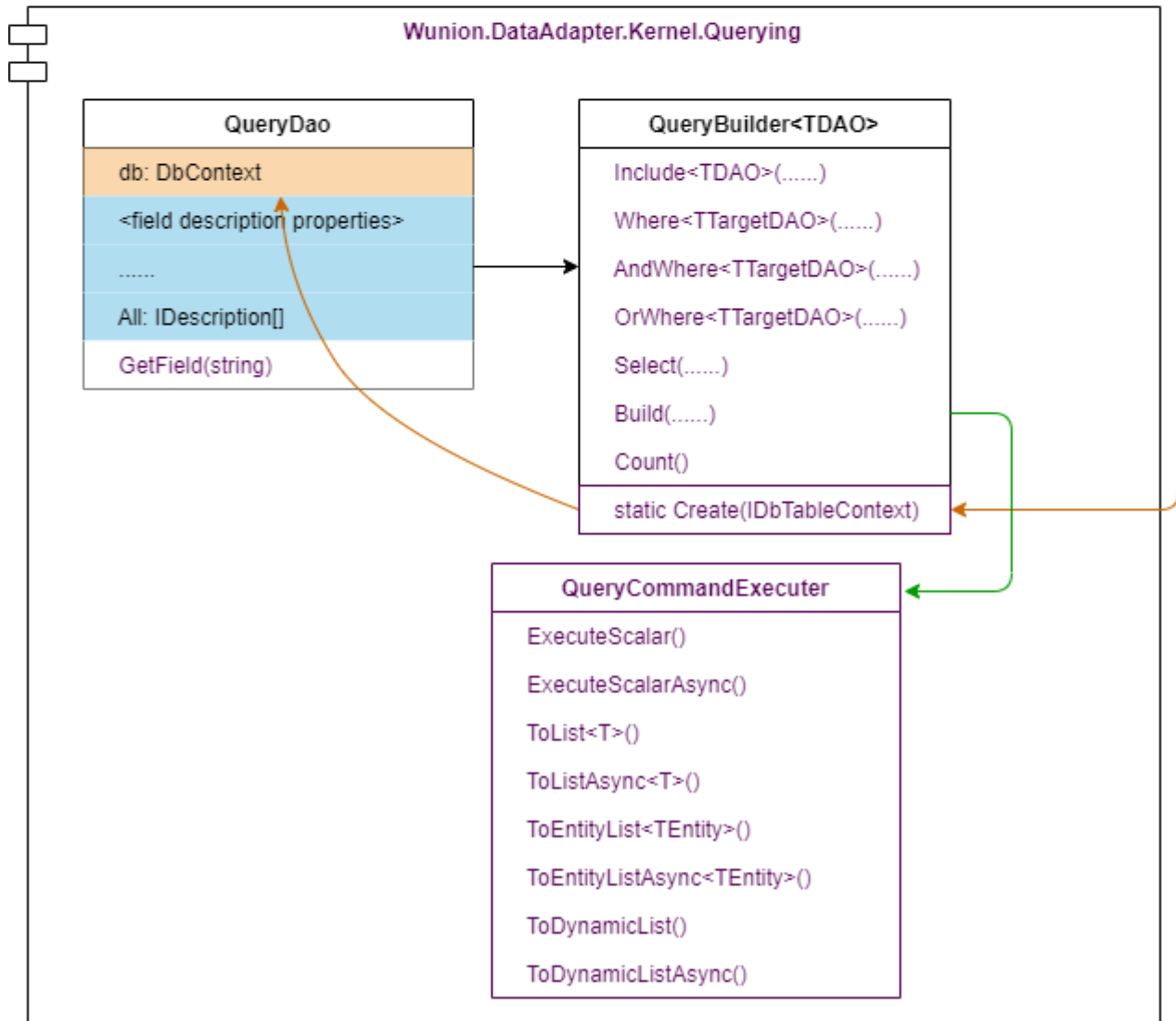
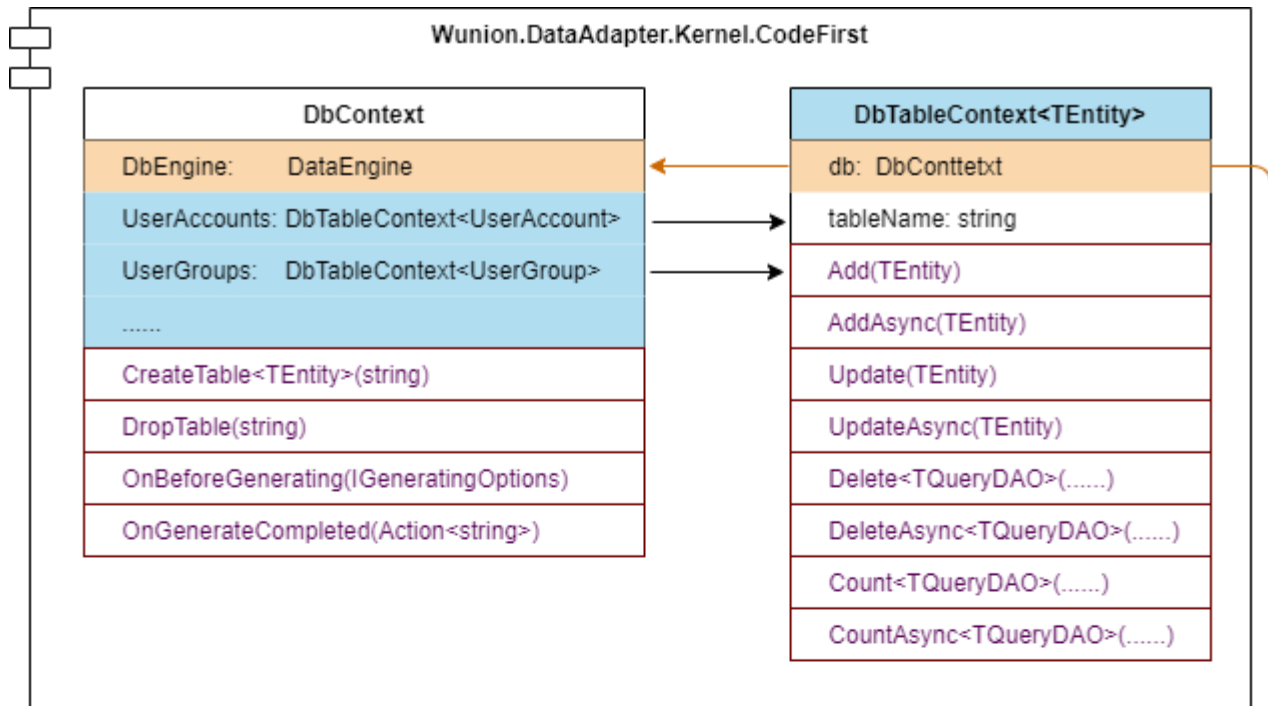
**DbContext**：数据库上下文，用于定义数据库架构及 CodeFirst 生成及迁移的行为，该类为抽象类由您继承并实现您的数据库结构。由一个 DataEngine 对象为 DbContext 提供数据库的读写访问基础（通过构造函数传入），在 DbContext 中您只需要定义数据库包含的表以及 CodeFirst 的生成及数据迁移行为。

**DbTableContext<TEntity>**：表上下文对象，用于定义数据库中的表，泛型类型 TEntity 为表对应的实体的类型名称。该类封装了对表的增、删、改操作。

**QueryDao**：实体对应的查询数据访问器，该类为抽象类。由 CodeFirst 工具“WDA-CF”根据实体自动生成各个实体对应的查询数据访问器派生类，它的作用旨在提供实体的各个表字段属性在查询时所对应字段描述 (FieldDescription) 信息，该信息描述了实体属性在 SQL 命令中的字段信息。

**QueryBuilder<TDAO>**：用于构建各种形式的数据库查询命令，并创建对应的 QueryCommandExecuter 实例执行数据库查询。

上述各个类的逻辑调用关系详见下图：



4.2.1、DbContext 类

定义

命名空间：Wunion.DataAdapter.Kernel.CodeFirst  
程序集： Wunion.DataAdapter.NetCore.dll  
包： Wunion.DataAdapter.NetCore v1.0.5.1

说明

用于定义数据库架构及 CodeFirst 生成及迁移的行为，该类为抽象类由您继承并实现您的数据库结构。由一个 DataEngine 对象为 DbContext 提供数据库的读写访问基础（通过构造函数传入），在 DbContext 中您只需要定义数据库包含的表以及 CodeFirst 的生成及数据迁移行为。

构造函数

<code>protected DbContext(DataEngine)</code>	创建一个数据库上下文的对象实例。 DataEngine 参数用于提供数据库的读写访问基础，数据库的连接字符串、连接池、值转换器等设置皆在该对象上进行配置。
--	--

属性

<code>DataEngine DbEngine</code>	获取此数据库上下文相关的数据库引擎，用于数据库的读写访问。通过该对象创建事务、批处理或者直接执行命令
----------------------------------	--

方法

<code>TableDeclaration&lt;TEntity&gt;(string)</code>	获取或定义数据库中的表。 String 参数表示表名称，当该名称的表未被定义时先定义。当该名称的表已存在时则直接返回表上下文对象。 返回类型：DbTableContext<TEntity>
<code>CreateTable(string, Type, object)</code>	在数据库上执行创建表的命令。 String 参数：要创建的表的表名 Type 参数：该表对应的实体类型 Object 参数：默认值 null。当此操作要在事务中执行时则应传入一个 DBTransactionController 对象； 当此操作要在批处理器中执行时则应传入一个 BatchCommander 对象；若传入其它值则引发 ArgumentException 异常。
<code>CreateTable&lt;TEntity&gt;(string, object)</code>	根据指定类型的实体在数据库上执行创建表的命令。 String 参数：要创建的表的表名 Object 参数：默认值 null。当此操作要在事务中执行时则应传入一个 DBTransactionController 对象；

	当此操作要在批处理器中执行时则应传入一个 <code>BatchCommander</code> 对象；若传入其它值则引发 <code>ArgumentException</code> 异常。
<code>DropTable(string, object)</code>	从数据库中删除指定名称的表。 <b>String</b> 参数：要删除的表的表名。 <b>Object</b> 参数：默认值 <code>null</code> 。当此操作要在事务中执行时则应传入一个 <code>DBTransactionController</code> 对象； 当此操作要在批处理器中执行时则应传入一个 <code>BatchCommander</code> 对象；若传入其它值则引发 <code>ArgumentException</code> 异常。
<code>OnBeforeGenerating(IGeneratingOptions)</code>	抽象方法，用于在创建或更新数据库架构前要配置数据库连接及生成行为。此方法由 WDA-CF 工具调用
<code>OnGenerateCompleted(Action&lt;string&gt;)</code>	当 <code>CodeFirst</code> 工具即（WDA-CF）完成数据库生成时调用此方法。通常用于向数据库中写入预置的数据，您可以不实现此方法。 <b>Action&lt;string&gt;</b> 参数：用于向 WDA-CF 输出日志内容。

### 4.2.2、DbTableContext<TEntity>类

#### 定义

命名空间：Wunion.DataAdapter.Kernel.CodeFirst  
程序集：Wunion.DataAdapter.NetCore.dll  
包：Wunion.DataAdapter.NetCore v1.0.5.1

#### 说明

表上下文对象，用于定义并映射数据库中的表，泛型类型 `TEntity` 为表对应的实体的类型名称。该类封装了对表的增、删、改操作。  
**TEntity** 泛型参数：表示该表所要映射的实体类型

#### 构造函数

<code>public DbTableContext(DbContext, string)</code>	创建一张数据库表的上下文映射对象。 <b>DbContext</b> 参数表示该表所属的数据库上下文。 <b>String</b> 参数表示该表在数据库中的表名。
---	---

#### 属性

<code>DbContext db</code>	获取该表的数据库上下文。
<code>string tableName</code>	获取该表在数据库中的表名。
<code>IDescription[] QueryFields</code>	获取该表的所有查询字段描述。

#### 方法

<pre>public virtual int Add(TEntity, object = null)</pre>	<p>向表中添加一条新的记录.</p> <p><b>TEntity</b> 参数表示要插入的记录的实体对象.</p> <p><b>object</b> 参数: 执行数据插入的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象.</p>
<pre>public async Task&lt;int&gt; AddAsync(TEntity, object = null)</pre>	<p>向表中添加一条新的记录（异步方法）.</p> <p><b>TEntity</b> 参数表示要插入的记录的实体对象.</p> <p><b>object</b> 参数: 执行数据插入的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象.</p>
<pre>public virtual void Update(TEntity, object = null)</pre>	<p>更新表中指定的记录, 该方法以实体类中标记的主键属性作为更新条件, 当对应的实体类中未标识主键属性时将引发异常.</p> <p><b>TEntity</b> 参数: 要更新的记录的实体对象.</p> <p><b>object</b> 参数: 执行更新的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象.</p>
<pre>public async Task UpdateAsync(TEntity, object = null)</pre>	<p>更新表中指定的记录（异步方法）.</p> <p>各个参数意义同上.</p>
<pre>public void Delete&lt;TQueryDAO&gt;(Func&lt;TQueryDAO, object[]&gt;, object = null) where TQueryDAO : QueryDao</pre>	<p>从表中删除符合条件的记录.</p> <p><b>Func&lt;TQueryDAO,object[]&gt;</b>参数: 用于构建删除条件.</p> <p><b>object</b> 参数: 执行删除的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象.</p>
<pre>public async Task DeleteAsync&lt;TQueryDAO&gt;(Func&lt;TQueryDAO, object[]&gt;, object = null) where TQueryDAO : QueryDao</pre>	<p>从表中删除符合条件的记录（异步方法）.</p> <p>各个参数意义同上.</p>
<pre>public int Count&lt;TQueryDAO&gt;(Func&lt;TQueryDAO, object[]&gt; = null, object = null) where TQueryDAO : QueryDao</pre>	<p>查询该表中符合条件的记录数量.</p> <p><b>Func&lt;TQueryDAO,object[]&gt;</b>参数: 用于构建查询条件.</p> <p><b>object</b> 参数: 执行查询的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象.</p>
<pre>public async Task&lt;int&gt; CountAsync&lt;TQueryDAO&gt;(Func&lt;TQueryDAO, object[]&gt; = null, BatchCommander = null) where TQueryDAO</pre>	<p>查询该表中符合条件的记录数量（异步方法）.</p> <p>参数含义同上.</p>



### 4.2.3、QueryBuilder<TDAO>类

## 定义

命名空间：Wunion.DataAdapter.Kernel.Querying  
程序集：Wunion.DataAdapter.NetCore.dll  
包：Wunion.DataAdapter.NetCore v1.0.5.1

## 说明

用于构建各种形式的数据库查询。泛型参数 TDAO 表示查询对应的数据访问对象类型（Data Access Object），DAO 对象总是与实体及表相关。CodeFirst 工具在生成数据库架构的同时会为每一张表对应的实体生成一个 DAO 类，通过该类来创建表的查询您不需在过多关心表及字段在数据库中的名称。

## 构造函数

受程序集密封保护

## 方法

<pre>public static QueryBuilder&lt;TDAO&gt; Create(IDbTableContext)</pre>	静态方法，创建一个查询构建器的对象实例。 IDbTableContext 参数：表上下文对象，创建的查询主要针对该表。后续可通过 Include 方法关联更多的表。
<pre>public QueryBuilder&lt;TDAO&gt; Include&lt;TForeignDao&gt;(Func&lt;IncludeQueryBuilder&lt;T DAO&gt;,object[]&gt;) where TForeignDao : QueryDao</pre>	在查询中关联一张表。 泛型参数 TForeignDao 表示要关联的表对应的 DAO 查询访问器类型。 Func<IncludeQueryBuilder<TDAO>,object[]>参数用于构建关联条件表达式（即 LEFT JOIN 语句中的 ON）
<pre>public QueryBuilder&lt;TDAO&gt; Include&lt;TForeignDao&gt;(string, Func&lt;IncludeQueryBuilder&lt;TDAO&gt;,object[]&gt;) where TForeignDao : QueryDao</pre>	在查询中关联一张表。 泛型参数 TForeignDao 表示要关联的表对应的 DAO 查询访问器类型。 string 参数：要关联的表在数据库中的表名，当数据库设计采用分表存储策略时，一个实体会对应多张表，此时需要用该参数来正确的关联需要的表。 Func<IncludeQueryBuilder<TDAO>,object[]>参数：用于构建关联条件表达式（即 LEFT JOIN 语句中的 ON）
<pre>public QueryBuilder&lt;TDAO&gt; Where&lt;TargetDao&gt;(Func&lt;TargetDao,object[]&gt;) where TargetDao : QueryDao</pre>	构建查询条件。 泛型参数 TargetDao 表示在查询中存在的表对应的 DAO 查询访问器类型，若给定的 TargetDao 类型在查询中未包含则会引发异常。 Func<TargetDao,object[]>参数：用于构建查询条件表达式。
<pre>public QueryBuilder&lt;TDAO&gt; AndWhere&lt;TargetDao&gt;(Func&lt;TargetDao,object[]&gt;) where TargetDao : QueryDao</pre>	构建与上一个构建的条件形成逻辑与（AND）运算的查询条件。 各项参数含义同 Where<TargetDao>(…) 方法。



<pre>public QueryBuilder&lt;TDAO&gt;   OrWhere&lt;TargetDao&gt;(Func&lt;TargetDao, object[]&gt;)   where TargetDao : QueryDao</pre>	构建与上一个构建的条件形成逻辑或（OR）运算的查询条件。 各项参数含义同 Where<TargetDao>(…) 方法。
<pre>public QueryBuilder&lt;TDAO&gt;   Select(Func&lt;IncludeQueryBuilder&lt;TDAO&gt;, IDescription[]&gt;)</pre>	选择在查询中要查出的字段。 Func<IncludeQueryBuilder<TDAO>, IDescription[]>参数：用于构建并返回在查询中要查出的字段。
<pre>public QueryCommandExecuter   Build(Action&lt;IncludeQueryBuilder&lt;TDAO&gt;, SelectBlock&gt;)</pre>	构建命令并返回一个查询命令执行器。 Action<IncludeQueryBuilder<TDAO>, SelectBlock>参数：用于设置查询命令的更多选项，例如排序、分组、分页等等。默认值为 null，为 null 时不附加任何多余的查询选项。
<pre>public int Count(object = null)</pre>	返回符合该查询的所有记录数量。 object 参数：执行查询的事务控制器 (DBTransactionController) 或批处理 (BatchCommander) 对象。

4.2.4、QueryCommandExecuter 类

定义

命名空间：Wunion.DataAdapter.Kernel.Querying  
程序集：Wunion.DataAdapter.NetCore.dll  
包：Wunion.DataAdapter.NetCore v1.0.5.1

说明

用于执行查询命令，并解析查询结果。该执行器由 QueryBuilder<TDAO>.Build(...) 方法创建，您无法在其它场景中手动创建该命令执行器的对象实例。

构造函数

受程序集密封保护

方法

<pre>public object ExecuteScalar(object = null)</pre>	执行查询并返回结果中第一行第一列的数据。 object 参数：在其中执行命令的事务控制器 (DBTransactionController)或批处理(BatchCommander)对象，默认值 null 表示不在事务或批处理器中执行。
<pre>public async Task&lt;object&gt;   ExecuteScalarAsync(object = null)</pre>	执行查询并返回结果中第一行第一列的数据（异步方法） 参数意义同 ExecuteScalar 方法。

<pre>public List&lt;T&gt; ToList&lt;T&gt;(object = null) where T : class, new()</pre>	<p>执行查询并将查询结果返回为指定类型的对象集合。 泛型参数 <b>T</b> 表示将单行记录解析为该类型的对象，但注意不是实体类型。 <b>object</b> 参数：在其中执行命令的事务控制器 (DBTransactionController) 或批处理 (BatchCommander) 对象，默认值 <b>null</b> 表示不在事件或批处理器中执行。</p>
<pre>public async Task&lt;List&lt;T&gt;&gt; ToListAsyn&lt;T&gt;(object = null) where T : class, new()</pre>	<p>执行查询并将查询结果返回为指定类型的对象集合（异步方法）。 各项参数同 <b>ToList&lt;T&gt;()</b> 方法。</p>
<pre>public List&lt;TEntity&gt; ToEntityList&lt;TEntity&gt;(object = null) where TEntity : class, new()</pre>	<p>执行查询并将查询结果返回为指定的实体集合。 泛型参数 <b>TEntity</b> 表示单行记录对应的实体类型。 <b>object</b> 参数：在其中执行命令的事务控制器 (DBTransactionController) 或批处理 (BatchCommander) 对象，默认值 <b>null</b> 表示不在事件或批处理器中执行。</p>
<pre>public async Task&lt;List&lt;TEntity&gt;&gt; ToEntityListAsync&lt;TEntity&gt;(object = null) where TEntity : class, new()</pre>	<p>执行查询并将查询结果返回为指定的实体集合（异步方法）。 各项参数同 <b>ToEntityList&lt;TEntity&gt;()</b> 方法。</p>
<pre>public List&lt;dynamic&gt; ToDynamicList (object = null, Func&lt;string, object, Type, object&gt; = null)</pre>	<p>执行查询并将查询结果返回为 <b>Wunion.DataAdapter.Kernel.DataCollection.DynamicEntity</b> 动态实体集合。 <b>object</b> 参数：在其中执行命令的事务控制器 (DBTransactionController) 或批处理 (BatchCommander) 对象，默认值 <b>null</b> 表示不在事件或批处理器中执行。 <b>Func&lt;string,object,Type,object&gt;</b> 参数：默认值 <b>null</b>，用于在将记录填充为 <b>DynamicEntity</b> 动态实体对象时，转换各个字段的值。</p>
<pre>public async Task&lt;List&lt;dynamic&gt;&gt; ToDynamicListAsync (object = null, Func&lt;string, object, Type, object&gt; = null)</pre>	<p><b>ToDynamicList ()</b> 方法的异步模式。 各项参数相同。</p>

## 4.3、定义数据库上下文

创建一个类并继承 `DbContext` 来定义属于您的数据库上下文，必须在数据库上下文中定义表以及实现 `CodeFirst` 的配置。

### 4.3.1、定义实体

一个实体对象对应数据库某个表中的一行记录，实体类不要求继承自某个特定的类，但您可以根据自己的需求梳理它们之间的继承关系。比如，很多表中的记录需要有创建日期及最后修改日期的字段，那么您可以定义一个抽象实体类来实现这两个字段，然后将所有有需要的实体继承该类型。

所有实体都必须声明其对应的表结构的生成依据，此依据通过在实体类的公共属性上添加 `TableFieldAttribute`、`IdentityAttribute`、`ForeignKeyAttribute` 来声明。如下示例：

```
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 特性：**用于表字段声明，在实体类中只有包含该声明的公共属性才会与数据库表中的字段映射。下表列出该特性的所有属性及其说明：

Name	string 类型，字段名称。不指定该属性则表示字段名与实体的属性名称相同.
DbType	GenericDbType 枚举，字段的数据类型. 必须指定
Size	int 类型，字段的长度
PrimaryKey	布尔值，true 表示该字段为主键。默认值 false
NotNull	布尔值，true 表示该字段不允许为空。默认值 false
Unique	布尔值，唯一值约束，true 表示该字段的值不允许重复。默认值 false，不可与 PrimaryKey 同时声明.
Default	object 类型，该字段的默认值.
ValueConverter	Type 类型，该字段的值转换器类型，此类型必须实现 DbValueConverter<TSource>抽象类。当未指定此属性且实体属性的类型不在数据库支持的范围时，ORM 自动从数据库上下文 (DbContext) 所属的 DataEngine（数据库引擎）对象中搜索注册的转换器，若未搜索到相关类型的转换器则会引发数据库异常。关于如何实现并注册值转换器将在后续章节中讲解。

**IdentityAttribute 特性：**用于声明字段的自动增行为，非必要。在下表中列出该特性的所有属性及其说明：

InitValue	int 类型，表示自动增长的起始值.
Increment	int 类型，表示自动增长的递增步长值.

**ForeignKeyAttribute 特性：**用于声明字段的外键约束，非必要。在下表中列出该特性的所有属性及其说明：

TableName	string 类型，主表的表名.
Field	string 类型，主表中的字段名.
OnDeleteAction	string 类型，主表中的记录被删除时的行为。取值范围： ForeignKeyAttribute.ACTION_CASCADE ForeignKeyAttribute.ACTION_SET_NULL ForeignKeyAttribute.ACTION_RESTRICT 默认值 ForeignKeyAttribute.ACTION_CASCADE
OnUpdateAction	string 类型，主表中的记录被更新时的行为。取值范围： ForeignKeyAttribute.ACTION_CASCADE ForeignKeyAttribute.ACTION_SET_NULL ForeignKeyAttribute.ACTION_RESTRICT 默认值 ForeignKeyAttribute.ACTION_CASCADE

**GenerateOrderAttribute 特性：**用于声明表或字段的生成顺序，非必要。该特性仅有一个 Index 属性，并且可能通过构造函数来初始化。Index 表示生成索引，从 0 开始允许重复。当您希望控制数据库中的表或表中的字段的生成顺序时，可以通过添加此特性声明的方式来实现。并且在整个数据库上下文定义中存在外键约束时，使用此特性声明主表及从表的生成顺序可以避免 CodeFirst 工具（WDA-CF）在运行时产生错误。

### 4.3.2、定义数据库中的表

在您的 `DbContext` 中定义 `DbTableContext<TEntity>` 类型的公共只读属性，并使该属性的 `get` 访问器调用 `TableDeclaration<TEntity>(string tableName)` 方法，如下示例：

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

    /// <summary>
    /// 用户账户表.
    /// </summary>
    [GenerateOrder(1)]
    public DbTableContext<UserAccount> UserAccounts => TableDeclaration<UserAccount>("UserAccounts");

    /// <summary>
    /// 用户账户分组表.
    /// </summary>
    [GenerateOrder(0)]
    public DbTableContext<UserGroup> UserGroups => TableDeclaration<UserGroup>("UserGroups");
}
```

**TableDeclaration<TEntity>方法：**此方法用于在数据库上下文中定义表，并返回该表的上下文对象。该方法不会在同一数据库上下文对象中产生重复的表定义。

**大数据的分表存储支持说明：**当某些数据非常庞大时，您可能希望将它们拆分在多张表中进行存储。拆分的方式可能是水平拆分也可能是纵向拆分，以纵向拆分为例。最开始只会存在一张空表，当这张表中的记录数量达到 3000000 条或者 5000000 条时，程序通过调用 `DbContext.CreateTable<TEntity>(string tableName)` 方法创建一张新表，并将记录存储在新创建的表中。在下次使用这张新创建的表时可通过调用 `DbContext.TableDeclaration<TEntity>(string tableName)` 方法来获得它的表上下文对象。

### 4.3.3、配置 CodeFirst 选项

在数据库上下文定义中必须要实现抽象方法 `OnBeforeGenerating(IGeneratingOptions options)` 来为 CodeFirst 工具配置生成行为。下例代码演示 CodeFirst 的所有配置：

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

    public override void OnBeforeGenerating(IGeneratingOptions options)
    {
        options.Database.ConnectionString = "....."; //设置数据库连接.
        options.Database.SchemaVersion = 1; //当前的数据库架构版本.
        //当表在数据库中已存在时是否强制重建（当修改实体后将该值设置为true，否则不会更新数据库）.
        options.Database.ReCreateExistingTable = false;
        //用于实现在重建表的过程中备份及恢复数据（此项可选择）.
        options.TableUpgradeMigrator = new MyTableUpgradeMigrator();
        //为实体生成的DAO查询访问器使用的命名空间.
        options.DaoGenerateNamespace = "Wunion.DataAdapter.CodeFirstDemo.Data.Domain";
        //DAO查询访问器生成的相对路径（从定义该类的项目的根目录开始）.
        options.DaoGenerateDirectory = System.IO.Path.Combine("Domain", "DAO");
    }
}
```

此方法由 CodeFirst 工具（WDA-CF）调用，若未正确实现此方法将会导致 WDA-CF 运行产生错误。

**预置数据：**当 CodeFirst 生成数据库后向数据库中预置某些必要的数据库，例如至少需要一个用户账户来登录系统或者其它系统运行必备的数据，当然这不是必须的。重写 `OnGenerateCompleted(Action<string> log)` 方法，来达到上述目的，示例：

```

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
            {
                // 预置用户账户组.
                UserAccountGroup group = new UserAccountGroup {
                    Id = 100,
                    Name = "Admin",
                    Description = "管理员用户组",
                    Creation = creation
                };
                UserGroups.Add(group, batch);
                group.Id = Convert.ToInt32(batch.SCOPE_IDENTITY);
                // 预置用户账户.
                log("正在预置超级用户 ...");
                UserAccount ua = new UserAccount {
                    Name = "super-admin",
                    Password = "mE9nJTgxBo3lDGJOg47LzX42a89K+LvjbAGQyfpG5k=",
                    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("数据预置已顺利完成.");
            }
            catch (Exception Ex)
            {
                log(Ex.Message);
            }
        }
    }
}

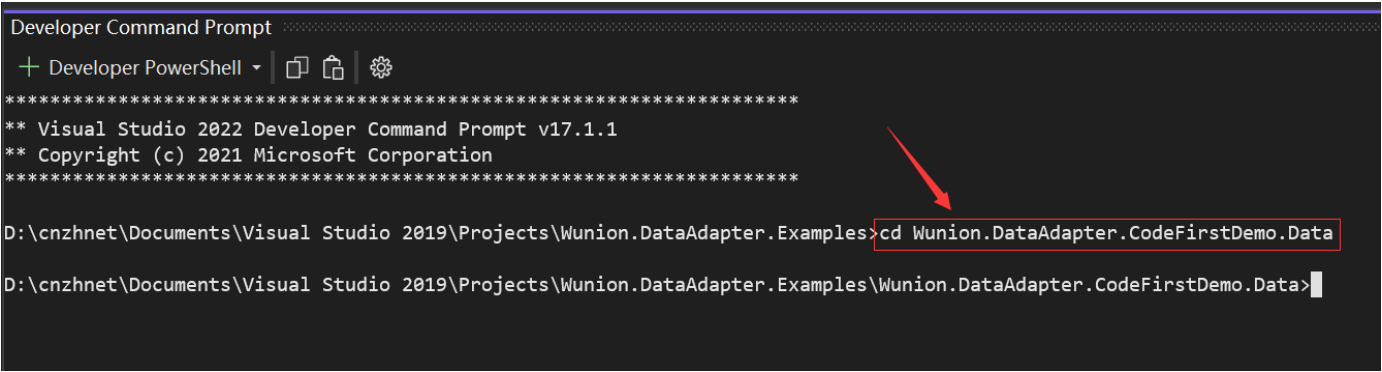
```

## 4.4、生成数据库及 DAO

在完成第 4.3 小节中讲述的工作后，使用 CodeFirst 工具生成数据库架构以及各个实体对应的 DAO 数据查询访问类。在 Visual Studio 2019 或 2022 或 VS Code 中打开终端：视图(V) -> Terminal



使用 Debug AnyCPU 模式编译该项目后，用 cd 命令定位到数据库上下文的定义项目目录，如下图所示：



执行：dotnet wda-cf <mssql | npgsql | mysql | sqlite3>

**CodeFirst 工具的命令行：**工具名称 WDA-CF 不区分大小写，该工具支持简体中文及英文两种语言，默认为简体中文。命令格式 dotnet /local:<zh-CN | en-US> <db-kind>

其中 /local: 为可选择参数，不指定时以简体中文输出日志信息，<db-kind>表示支持的数据库种类代码。数据库种类代码说明如下：

- mssql 表示 Microsoft SQL Server 数据库
- mysql 表示 MySQL 数据库，默认存储引擎为 InnoDB 可显式指定，例如 mysql:InnoDB, mysql:MyISAM
- npgsql 表示 PostgreSQL 数据库
- sqlite3 表示 sqlite3 数据库



## 4.5、数据的插入/更新/删除

### 插入记录

```
MyDbContext db = .....
UserAccount ua = new UserAccount {
    Name = "super-admin",
    Password = "mE9nJTgxBo3lDGJ0g47LzX42a89K+LvjbAGQyfpG5k=",
    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);
```

### 更新记录

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

### 删除记录

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

public void DeleteGroup(int gid)
{
    // 使用批处理删除
    using (BatchCommander batch = new BatchCommander(dbContext.DbEngine))
    {
        dbContext.UserAccounts.Delete<UADAO>(p => new object[] { p.Group.Like($"{gid};") }, batch);
        dbContext.UserGroups.Delete<UGDAO>(p => new object[] { p.Id == gid }, batch);
    }
    /* 使用事务删除：事务的使用请参见第二章2.6小节
    using (DBTransactionController trans = dbContext.DbEngine.BeginTrans())
    {
        dbContext.UserAccounts.Delete<UADAO>(p => new object[] { p.Group.Like($"{gid};") }, trans);
        dbContext.UserGroups.Delete<UGDAO>(p => new object[] { p.Id == gid }, trans);
        trans.Commit();
    }
    */
}
```

## 4.6、查询数据

使用 `QueryBuilder<TDAO>` 类构建并执行查询，泛型类型 `TDAO` 表示查询中第一个表对应的 DAO 数据查询访问类，使用 `QueryBuilder<TDAO>.Create(IDbTableContext table)` 静态方法创建查询。

### 4.6.1、查询单个表

根据某张表的 `DbTableContext<TEntity>` 上下文对象来构建并执行查询，获得查询结果。示例：

```
// 同步方式.
List<UserAccount> queryResult = QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
    .Select(p => p.First.All) // p.First 表示查询中的第一个表的DAO
    .Where<UserAccountDao>(p => new object[] {
        // 多条件逻辑如下行取消注释 ----- ↓
        p.Group.Like("0,", LikeMatch.Center) // & p.Name == "cnzhnet"
        // 更多查询条件的语法请参见第二章2.3小节
    }).Build((p, options) => {
        options.OrderBy(p.First.UID, OrderByMode.ASC); // 排序.
        options.Paging(pageSize, currentPage); // 分页.
    })
    .ToEntityList<UserAccount>();

/* 说明: Select 方法用于确定要查询的字段, 任何一个查询都必需要明确在查询结果中须要的字段, Build 方法则用于构建查询并返回该查询的命令执行器 QueryCommandExecutor, 有关 QueryCommandExecutor 的内容请参见4.2.4小节 */

// 异步方式.
List<UserAccount> queryResult = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
    .Select(p => p.First.All)
    .Build((p, options) => {
        options.OrderBy(p.First.UID, OrderByMode.ASC); // 排序.
        options.Paging(pageSize, currentPage); // 分页.
    })
    .ToEntityListAsync<UserAccount>();
```

### 4.6.2、多表联合查询

根据某张表的 `DbTableContext<TEntity>` 上下文对象来构建并执行查询。在构建查询时使用 `QueryBuilder<TDAO>.Include<TForeignDao>(.....)` 方法来关联表，Include 方法最终以 LEFT JOIN 语句来关联多张表，在多表查询时查询结果通常包含多个表中的字段。此时没有特定的实体来承载查询结果中的记录，这种情况下您需要定义数所模型，或者使用动态实体类型 (`DynamicEntity`)。

在此 ORM 中不实现类似 Entity Framework 中的导航属性来关联表中的数据记录，您需要根据实际情况手动进行多表联合查询并设计相应的数据模型。

示例：

```

List<UserDataModel> users = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
    .Include<UserGroupDao>(p => { // p.First 表示查询中的第一个表的DAO
        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[] { // 第一个表的条件.
        p.Group == groupId
    }).AndWhere<UserGroupDao>(p => new object[] { // 第二个表的条件(没有则不需要).
        .....
    })
    .Build((p, options) => {
        options.OrderBy(p.First.UID, OrderByMode.ASC); // 排序.
        options.Paging(pageSize, currentPage); // 分页.
    })
    .ToListAsync<UserAccount>();

```

#### 4.6.3、数据模型或动态实体

某些情况下查询并非总是需要查出表中的所有字段，或者在多表查询时查询结果包含多个表中的字段。此时没有特定的实体来承载查询结果中的记录，这种情况下您需要定义数据模型，或者使用动态实体类型（[DynamicEntity](#)）。在 [QueryCommandExecuter](#) 中提供 [ToList](#)、[ToListAsync](#) 方法将查询结果直接返回为数据模型集合，数据模型对象的属性成员中必须至少包含一个在 [QueryBuilder<TDAO>.Select\(.....\)](#) 方法中选定的查询字段。示例：

```

UserDataModel queryResult = QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
    .Select(p => new object[] { // p.First 表示查询中的第一个表的DAO
        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();

```

**使用动态实体作为查询结果：** [DynamicEntity](#) 类在命名空间 [Wunion.DataAdapter.Kernel.DataCollection](#) 中定义。动态实体根据查询中 [Select](#) 的字段动态地生成实体对象的属性成员，这使得您不必过多的定义数据模型。下面的例子演示用动态实体返回查询结果：

```

List<dynamic> users = await QueryBuilder<UserAccountDao>.Create(dbContext.UserAccounts)
    .Include<UserGroupDao>(p => { // p.First 表示查询中的第一个表的DAO
        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[] { // 第一个表的条件.
        p.Group == groupId
    })
    .Build((p, options) => {
        options.OrderBy(p.First.UID, OrderByMode.ASC); // 排序.
        options.Paging(pageSize, currentPage); // 分页.
    })
    .ToDynamicListAsync(converter: (field, value, fieldType) => {
        // converter用于转换字段的值
        switch (field)
        {
            case "GroupName":
                return JsonConvert.Deserialize<List<int>>(value);
            case "Status":
                return (UserAccountStatus)((int)value);
            default:
                return value;
        }
    });

```

## 4.7、执行自定义 SQL 命令

使用 `DbCommandBuilder` 来创建 SQL 命令，使您不必担心不同种类数据库的 SQL 命令的所差异导致程序运行出错。将 `DbCommandBuilder` 对象传递给 `BatchCommander` 对象或 `DBTransactionController` 事务控制器或 `DbContext.DbEngine` 的相应方法。使用 `DbCommandBuilder` 创建 SQL 命令的相关内容请参见[第二章](#)，以下代码具体演示如何执行自定义 SQL 命令：

```

DbCommandBuilder cb = new DbCommandBuilder();
cb.Update(dbContext.UserAccounts.TableName)
    .Set(td.Field("Password") == newPassword)
    .Where(td.Field("UID") == 0);
// 在 dbContext.DbEngine 上执行.
int result = dbContext.DbEngine.ExecuteNoneQuery(cb);

// 在事务中执行.
using (DBTransactionController trans = dbContext.DbEngine.BeginTrans())
{
    result = trans.DBA.ExecuteNoneQuery(cb);
}

// 在批处理器中执行.
using (BatchCommander batch = new BatchCommander(dbContext.DbEngine))
{
    result = batch.ExecuteNoneQuery(cb);
}

请参考:
https://github.com/cnzhnet/Wunion.DataAdapter.Examples/blob/main/Wunion.DataAdapter.CodeFirstDemo.Application/AppService.cs

```

## 4.8、值转换器的实现及注册

值转换器用于在向数据库中插入或更新记录时将自定义的数据（类或枚举）转换为数据库支持的数据类型，值转换器可以事先在应用程序启动时注册到全局的 **DataEngine** 对象上，也可以在实体类的属性上指定（参见[第 4.3.1 小节](#)）。并且在查询数据时也需要值转换器将数据库中的字段值转换为对应实体相应属性的类型。

### 4.8.1、为自定义类型实现值转换器

通过继承 **DbValueConverter<TSource>** 抽象类来创建特定类型的值转换器，下面的例子演示用户账户状态枚举类型 **UserAccountStatus** 的值转换器：

```

public class UserAccountStatusConverter : DbValueConverter<UserAccountStatus>
{
    /// <summary>
    /// 转换到数据库支持的类型.
    /// </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)) // 转换为字符串.
    {
        buffer = Enum.GetName<UserAccountStatus>(value);
        return;
    }
    // 转换为数字.
    buffer = (int)value;
}

/// <summary>
/// 从数据库支持的类型转换为枚举.
/// </summary>
/// <param name="value"></param>
/// <param name="buffer"></param>
protected override void Parse(object value, ref UserAccountStatus buffer)
{
    // 从数字转换.
    if (value.GetType() == typeof(int))
    {
        int numeric = Convert.ToInt32(value);
        buffer = (UserAccountStatus)numeric;
        return;
    }
    // 从字符串转换.
    string name = value.ToString();
    buffer = Enum.Parse<UserAccountStatus>(name);
}
}

```

## 4.8.2、在 DataEngine 上注册值转换器

下面以 asp.net core 为例演示如何将值转换器注册到全局 DataEngine 中：

```

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);
    }
}

// Startup 类为 asp.net core 默认的启动配置.
public class Startup
{
    .....
    public void ConfigureServices(IServiceCollection services)
    {
        .....
        services.AddDbContext((options =>) { // 在此注册全局值转换器.
            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;
            });
        });
    }
}

```