微型 ORM-FluentData 实例详解

前言：

最近项目中因使用FluentData，查阅部分博客资料觉得不够全面，所以把官网下的文档翻译一下，供大家学习参考。

正文：

开始学习

环境要求

- .NET 4.0.

支持数据库

- `MS SQL Server` 使用本地.NET驱动程序
- `MS SQL Azure` 使用本地.NET驱动程序
- `MS Access`使用本地.NET驱动程序
- `Microsoft SQL Server Compact 4.0` 需使用驱动 Microsoft SQL Server Compact 4.0 driver
- `Oracle`需使用驱动 ODP.NET driver
- `MySQL`需使用驱动 MySQL Connector .NET driver
- `SQLite`需使用驱动 SQLite ADO.NET Data Provider
- `PostgreSql`由Npgsql 提供
- `IBM DB2`
- `Sybase`由https://github.com/FredoKapo/FLUENT-ORM-ASE-PROVIDER提供

安装

使用NuGet

- 搜索`FluentData`并安装。

不使用NuGet

1. 下载zip文件。
2. 解压文件，并将文件复制到您的解决方案或项目文件夹中。
3. 项目中添加对fluentdata.dll引用。

核心概念

DbContext

这是fluentdata的核心类。可以通过配置ConnectionString来定义这个类，如何连接数据库和具体数据库信息

DbCommand

这是负责对数据库执行实际查询的类。

Events

DbContext类有以下的Events(事件)支持：

- OnConnectionClosed
- OnConnectionOpened
- OnConnectionOpening
- OnErrorOnExecuted
- OnExecuting

通过使用其中任何一个，可以在事件中，记录每个SQL查询错误或者SQL查询执行的时间等信息。

Builders

Builders(生成器)提供了一个非常好的API，用于生成SQL，用于插入、更新和删除查询。

Builder用来创建Insert, Update, Delete等相关的DbCommand实例。

Mapping

`FluentData`可以将SQL查询结果自动映射成一个`POCO(POCO - Plain Old CLR Object)`实体类，也可以转换成一个`dynamic`（new in .NET 4）类型：

自动映射为实体类：

1. 如果字段名称不包含下划线（“_”）将自动映射到具有相同名称的属性上。例如，一个名为“Name”的字段将被自动映射到名也为“Name”的属性上。
2. 如果字段名包含下划线（“_”），将自动映射到嵌套属性上。例如，例如，一个名为“Category_Name”的字段值将自动映射到名为“Category.Name”的属性上。

如果数据库中的字段和实体类型之间存在不匹配，则可以使用SQL中的别名关键字，也可以创建自己的映射方法。检查下面的映射部分以获取代码示例。

自动映射为`dynamic`（动态类型）

1. 动态类型的每一个字段都将被自动映射成具有有相同的名称的属性。例如，字段名为Name会被自动映射成名为Name的属性。

什么时候需要释放资源？

- `DbContext` 需要主动释放当你在启用`UseTransaction`或者`UseSharedConnection`时

- DbContext 需要主动释放当你在启用UseMultiResult (or MultiResultSql)时
- StoredProcedureBuilder 需要主动释放当你在启用UseMultiResult时

在所有其他情况下处置将由fluentdata自动处理。这意味着在执行完查询并关闭之前，数据库连接一直是打开状态。

代码实例

创建和初始化一个DbContext

可以在*.config文件中配置connection string，将connection string name或者将整个connection string作为参数传递给DbContext来创建DbContext。

重要的配置

IgnoreIfAutoMapFails – IDbContext.IgnoreIfAutoMapFails返回一个IDbContext，该实例中，如果在字段不能与属性正确映射时是否抛出异常

创建和初始化一个DbContext

通过*.config中配置的ConnectionStringName:MyDatabase创建一个DbContext

```
public IDbContext Context()
{
    return new DbContext().ConnectionStringName("MyDatabase",
            new SqlServerProvider());
}

// *.config文件中内容
<connectionStrings>
    <add name="MyDatabase" connectionString="server=MyServerAddress;uid=uid;pwd=pwd;database=MyDatabase;"  />
</connectionStrings>
```

调用DbContext的ConnectionString方法显示设置connection string来创建

```
//
public IDbContext Context()
{
    return new DbContext().ConnectionString(
    "Server=MyServerAddress;Database=MyDatabase;Trusted_Connection=True;", new SqlServerProvider());
}
```

其他可以使用的提供器

如果你想连接其他的非SqlServer的数据库，这是非常简单的，只需要替换上面代码中的“new SqlServerProvider()”为下面的数据提供器，比如:AccessProvider, DB2Provider, OracleProvider, MySqlProvider, PostgreSqlProvider, SqliteProvider, SqlServerCompact, SqlAzureProvider, SqlServerProvider。

查询一组数据（Query for a list of items）

返回一组dynamic对象

```
List<dynamic> products = Context.Sql("select * from Product").QueryMany<dynamic>();
```

返回一组强类型对象

```
List<Product> products = Context.Sql("select * from Product").QueryMany<Product>();
```

返回一组自定义的Collection

```
ProductionCollection products = Context.Sql("select * from Product").QueryMany<Product, ProductionCollection>();
```

返回DataTable类型:
查看下方查询单个对象（Query for a single item）

查询单个对象（Query for a single item）

返回一个dynamic对象

```
dynamic product = Context.Sql(@"select * from Product
                where ProductId = 1").QuerySingle<dynamic>();
```

返回一个强类型对象

```
Product product = Context.Sql(@"select * from Product
            where ProductId = 1").QuerySingle<Product>();
```

返回一个DataTable

```
/**
 * 其实QueryMany<DataTable>和QuerySingle<DataTable>都可以用来返回DataTable，
 * 但考虑到QueryMany<DataTable>返回的是List<DataTable>，
 * 所以使用QuerySingle<DataTable>来返回DataTable更方便。
 */
DataTable products = Context.Sql("select * from Product").QuerySingle<DataTable>();
```

查询一个标量值

```
int numberOfProducts = Context.Sql(@"select count(*)
            from Product").QuerySingle<int>();
```

查询一组标量值

```
List<int> productIds = Context.Sql(@"select ProductId
                from Product").QueryMany<int>();
```

查询参数

索引形式参数：

```
dynamic products = Context.Sql(@"select * from Product
            where ProductId = @0 or ProductId = @1", 1, 2).QueryMany<dynamic>();
```

或者：

```
dynamic products = Context.Sql(@"select * from Product
            where ProductId = @0 or ProductId = @1")
            .Parameters(1, 2).QueryMany<dynamic>();
```

命名形式参数：

```
dynamic products = Context.Sql(@"select * from Product
            where ProductId = @ProductId1 or ProductId = @ProductId2")
            .Parameter("ProductId1", 1)
            .Parameter("ProductId2", 2)
            .QueryMany<dynamic>();
```

OutPut形式参数:

```
var command = Context.Sql(@"select @ProductName = Name from Product
            where ProductId=1")
            .ParameterOut("ProductName", DataTypes.String, 100);
command.Execute();

string productName = command.ParameterValue<string>("ProductName");
```

List形式参数-in查询：

```
List<int> ids = new List<int>() { 1, 2, 3, 4 };
// 注意这里,不要在"in(...)"周围有任何空格的操作符.
dynamic products = Context.Sql(@"select * from Product
            where ProductId in(@0)", ids).QueryMany<dynamic>();
```

like查询：

```
string cens = "%abc%";
Context.Sql("select * from Product where ProductName like @0",cens);
```

映射

自动映射-数据库对象与.Net对象自动进行1：1匹配：

```
List<Product> products = Context.Sql(@"select *
            from Product")
            .QueryMany<Product>();
```

自动映射到一个自定义的Collection：

```
ProductionCollection products = Context.Sql("select * from Product").QueryMany<Product, ProductionCollection>();
```

如果数据库字段和.Net对象类属性名不一致，使用SQL别名语法AS：

```
/*
 * 在这里p.*中的ProductId和ProductName会自动映射到Prodoct.ProductId和Product.ProductName,
 * 而Category_CategoryId和Category_Name将映射到Product.Category.CategoryId和 Product.Category.Name
 */
List<Product> products = Context.Sql(@"select p.*,
            c.CategoryId as Category_CategoryId,
            c.Name as Category_Name
            from Product p
            inner join Category c on p.CategoryId = c.CategoryId")
                .QueryMany<Product>();
```

使用dynamic自定义映射规则：

```
List<Product> products = Context.Sql(@"select * from Product")
            .QueryMany<Product>(Custom_mapper_using_dynamic);

public void Custom_mapper_using_dynamic(Product product, dynamic row)
{
    product.ProductId = row.ProductId;
    product.Name = row.Name;
}
```

使用datareader进行自定义映射：

```
List<Product> products = Context.Sql(@"select * from Product")
```

```
            .QueryMany<Product>(Custom_mapper_using_datareader);

public void Custom_mapper_using_datareader(Product product, IDataReader row)
{
    product.ProductId = row.GetInt32("ProductId");
    product.Name = row.GetString("Name");
}
```

或者当你需要映射到一个复合类型时，可以使用`QueryComplexMany`或者`QueryComplexSingle`：

```
var products = new List<Product>();
Context.Sql("select * from Product").QueryComplexMany<Product>(products, MapComplexProduct);

private void MapComplexProduct(IList<Product> products, IDataReader reader)
{
    var product = new Product();
    product.ProductId = reader.GetInt32("ProductId");
    product.Name = reader.GetString("Name");
    products.Add(product);
}
```

多结果集

FluentData支持多结果集。也就是说，可以在一次数据库查询中返回多个查询结果。使用该特性的时候，记得使用类似下面的语句对查询语句进行包装。需要在查询结束后把连接关闭。

```
/**
 * 执行第一个查询时，会从数据库取回数据。
 * 执行第二个查询的时候，FluentData可以判断出这是一个多结果集查询，所以会直接从第一个查询里获取需要的数据。
 */
using (var command = Context.MultiResultSql)
{
    List<Category> categories = command.Sql(
            @"select * from Category;
            select * from Product;").QueryMany<Category>();

    List<Product> products = command.QueryMany<Product>();
}
```

选择数据和分页

选择一个`builder`使得选择数据和分页更简单：

```
// 通过调用Paging(1, 10)，将返回前10个产品。
List<Product> products = Context.Select<Product>("p.*, c.Name as Category_Name")
                    .From(@"Product p
                     inner join Category c on c.CategoryId = p.CategoryId")
                    .Where("p.ProductId > 0 and p.Name is not null")
                    .OrderBy("p.Name")
                    .Paging(1, 10).QueryMany();
```

插入数据

使用`SQL`：

```
int productId = Context.Sql(@"insert into Product(Name, CategoryId)
            values(@0, @1);")
            .Parameters("The Warren Buffet Way", 1)
            .ExecuteReturnLastId<int>();
```

使用`builder`：

```
int productId = Context.Insert("Product")
            .Column("Name", "The Warren Buffet Way")
            .Column("CategoryId", 1)
            .ExecuteReturnLastId<int>();
```

使用`builder`，并且自动映射：

```
Product product = new Product();
product.Name = "The Warren Buffet Way";
product.CategoryId = 1;

// 将ProductId作为AutoMap方法的参数，是要指明ProductId不需要进行映射，因为它是一个数据库自增长字段
product.ProductId = Context.Insert<Product>("Product", product)
            .AutoMap(x => x.ProductId)
            .ExecuteReturnLastId<int>();
```

更新数据

使用`SQL`：

```
int rowsAffected = Context.Sql(@"update Product set Name = @0
```

```
            where ProductId = @1")
            .Parameters("The Warren Buffet Way", 1)
            .Execute();
```

使用builder：

```
int rowsAffected = Context.Update("Product")
            .Column("Name", "The Warren Buffet Way")
            .Where("ProductId", 1)
            .Execute();
```

使用builder，并且自动映射：

```
Product product = Context.Sql(@"select * from Product
            where ProductId = 1")
            .QuerySingle<Product>();
product.Name = "The Warren Buffet Way";

// 将ProductId作为AutoMap方法的参数，是要指明ProductId不需要进行映射，因为它不需要被更新。
int rowsAffected = Context.Update<Product>("Product", product)
            .AutoMap(x => x.ProductId)
            .Where(x => x.ProductId)
            .Execute();
```

设置映射失败异常是否抛出（IgnoreIfAutoMapFails）

当从数据库中读取，如果某些数据列不映出实体类，默认情况下，将抛出异常。

如果你想忽略异常，或者属性不需要和数据库对象进行映射，你可以设置IgnoreIfAutoMapFails(true)，即可以在映射错误时不抛出异常

```
context.IgnoreIfAutoMapFails(true);
```

插入和更新 - 常用填充方式

```
var product = new Product();
product.Name = "The Warren Buffet Way";
product.CategoryId = 1;

var insertBuilder = Context.Insert<Product>("Product", product).Fill(FillBuilder);

var updateBuilder = Context.Update<Product>("Product", product).Fill(FillBuilder);

public void FillBuilder(IInsertUpdateBuilder<Product> builder)
{
    builder.Column(x => x.Name);
    builder.Column(x => x.CategoryId);
}
```

删除数据

使用SQL：

```
int rowsAffected = Context.Sql(@"delete from Product
            where ProductId = 1")
            .Execute();
```

使用builder：

```
int rowsAffected = Context.Delete("Product")
            .Where("ProductId", 1)
            .Execute();
```

存储过程

使用SQL：

```
int rowsAffected = Context.Sql(@"delete from Product
            where ProductId = 1")
            .Execute();
```

使用builder：

```
var rowsAffected = Context.StoredProcedure("ProductUpdate")
            .Parameter("Name", "The Warren Buffet Way")
            .Parameter("ProductId", 1).Execute();
```

使用builder，并且自动映射：

```
var product = Context.Sql("select * from Product where ProductId = 1")
            .QuerySingle<Product>();

product.Name = "The Warren Buffet Way";
```

```
var rowsAffected = Context.StoredProcedure<Product>("ProductUpdate", product)
            .AutoMap(x => x.CategoryId).Execute();
```

使用builder，并且自动映射和表达式：

```
var product = Context.Sql("select * from Product where ProductId = 1")
            .QuerySingle<Product>();
product.Name = "The Warren Buffet Way";

var rowsAffected = Context.StoredProcedure<Product>("ProductUpdate", product)
            .Parameter(x => x.ProductId)
            .Parameter(x => x.Name).Execute();
```

使用事务

FluentData 支持事务。如果使用事务，最好使用using语句将代码包起来，已保证连接会被关闭。默认的，如果查询过程发生异常，如事务不会被提交，会进行回滚。

```
using (var context = Context.UseTransaction(true))
{
    context.Sql("update Product set Name = @0 where ProductId = @1")
                .Parameters("The Warren Buffet Way", 1)
                .Execute();

    context.Sql("update Product set Name = @0 where ProductId = @1")
                .Parameters("Bill Gates Bio", 2)
                .Execute();

    context.Commit();
}
```

实体工厂

实体工厂负责在自动映射的时候，生成对象实例。如果需要生成复杂的实例，可以自定义实体工厂：

```
List<Product> products = Context.EntityFactory(new CustomEntityFactory())
            .Sql("select * from Product")
            .QueryMany<Product>();

public class CustomEntityFactory : IEntityFactory
{
    public virtual object Resolve(Type type)
    {
        return Activator.CreateInstance(type);
    }
}
```