Fluent Data 入门

由 Primates 根据互联网资源整理

FluentData 是微型 ORM（micro-ORM）家族的一名新成员，旨在比大型 ORM（full ORM）更加易用。FluentData 于本月推出，它使用 fluent API 并支持 SQL Server、SQL Azure、Oracle 和 MYSQL。

FluentData 的设计者 Lars-Erik Kindblad 谈到：

> **当前市面上的 ORM 框架，如 Entity Framework 和 NHibernate，都过于复杂而且难于学习。此外，由于这些框架自身抽象的查询语言以及从数据库到.NET 对象的映射太过麻烦，导致它们生成的 SQL 都很低效。**
>
> **FluentData 另辟蹊径，它是一个轻量级框架，拥有简单的 fluent API 并且很容易学会。**

与其他微型 ORM（如 Dapper 和 Massive）类似，FluentData 关注性能和易用性。它允许开发人员拥有对 SQL 较多的控制，而不是依赖 ORM 进行自动生成。它不仅可以使用 SQL 来执行查询、增添和更新操作，还可以支持使用存储过程和事务。根据文档描述，FluentData 可以在不改动已有结构的情况下，与任何业务对象一同工作。

以下是 FluentData 的一些其他特性：

- 多结果集（Multiple Result Set）：在一次数据库操作下返回多个数据集；
- 开发人员可使用强类型对象或动态对象；
- 可为创建时需要特殊处理的复杂对象自定义实体工厂（Custom Entity Factory）；
- 具有添加其他数据库支持的能力。

FluentData 需要 .NET 4.0，并支持 SQL Server、SQL Azure、SQL Server Compact 以及使用 .NET 驱动的 Oracle 和 MySQL。 想要了解进一步信息，如代码示例和免费下载，请访问 CodePlex 站点上的 FluentData。

一 核心概念

类	说明	备注
DbContext	FluentData 的核心类，可以通过配置 ConnectionString 来定义这个类，如何连接数据库和对具体 DbContext 类。	
DbCommand	负责在相对应的数据库执行具体的每一个数据操作。	
Events	DbContext 类定义了以下这些事件：OnConnectionClosed、OnConnectionOpened、OnConnectionOpening、OnError、OnExecuted、OnExecuting。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	用来创建 Insert, Update, Delete 等相关的 DbCommand 实例。	
Mapping	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 类型。 ● **自动转成实体类：** 如果字段名中不包含下划线("_")，将映射到具有相同名字的属性上，例如：字段名 "Name" 将映射到属性名 "Name"。 如果字段名中不包含下划线("_")，将映射到内嵌的属性上，例如：字段名 "CategoryName" 将映射到属性名 "Category.Name"。 如果数据库字段名和属性名不一致，可以使用 SQL 的 as 让他们一致。 ● **自动转换成 dynamic 类型：** 对应 dynamic 类型，会为每个字段生成一个同名的属性，例如：字段名 "Name" 将映射到属性名 "Name"。 什么时候应该主动释放资源？ 如果使用 UseTransaction 或者 UseSharedConnection，那么 DbContext 需要主动释放。 如果使用 UseMultiResult 或者 MultiResultSql，那么 DbCommand 需要主动释放。 如果使用 UseMultiResult，那么 StoredProcedureBuilder 需要主动释放。其他所有的类都会自动释放，也就是说，一个数据库连接只在查询开始前才进行，查询结束后会马上关闭。的哪个数据库进行数据查询操作。	

其他所有的类都会自动释放，也就是说，一个数据库连接只在查询开始前才进行，查询结束后会马上关闭。

二 创建 DbContext

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。

重要配置

IgnoreIfAutoMapFails – IDbContext.IgnoreIfAutoMapFails 返回一个 IDbContext，该实例中，如果自动映射失败时是否抛出异常

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

```
1：  public IDbContext Context()
2：  {
3：        return new DbContext().ConnectionStringName("MyDatabase",
4：                    new SqlServerProvider());
5：  }
```

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

```
1：  public IDbContext Context()
2：  {
3：        return new DbContext().ConnectionString(
4：        "Server=MyServerAddress;Database=MyDatabase;Trusted_Connection=True;", new SqlServerProvider());
5：  }
```

其他可以使用的 Provider

只有通过使用不同的 Provider，就可以切换到不同类型的数据库服务器上，比如

AccessProvider, DB2Provider, OracleProvider, MySqlProvider, PostgreSqlProvider, SqliteProvider, SqlServerCompact, SqlAzureProvider, SqlServerProvider.

三 查询（Query）

返回单个对象

返回一个 dynamic 对象

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

返回一个强类型对象：

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

返回一个 DataTable：

```
1： DataTable products = Context.Sql("select * from Product").QuerySingle<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。

查询一个标量值

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

返回一组数据

返回一组 dynamic 对象(new in .NET 4.0)

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

返回一组强类型对象

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

返回一个自定义的 Collection

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

返回一组标量值

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

参数化查询

索引顺序形式的参数

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

或者

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

名字形式的参数：

```
1：  var command = Context.Sql(@"select @ProductName = Name from Product where ProductId=1")
2：                  .ParameterOut("ProductName", DataTypes.String, 100);
3：  command.Execute();
4：  string productName = command.ParameterValue<string>("ProductName");
5：  List of parameters - in operator：
6：  List<int> ids = new List<int>() { 1, 2, 3, 4 };
7：  dynamic products = Context.Sql(@"select * from Product
8：                  where ProductId in(@0)", ids).QueryMany<dynamic>();
```

Output 参数：

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

四 映射（Mapping）

自动映射

在数据库对象和.Net object 自动进行1：1匹配

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

自动映射到一个自定义的 Collection：

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

如果数据库字段和 POCO 类属性名不一致，使用 SQL 别名语法 AS：

```
1：  List<Product> products = Context.Sql(@"select p.*,
2：                  c.CategoryId as Category_CategoryId,
3：                  c.Name as Category_Name
4：                  from Product p
5：                  inner join Category c on p.CategoryId = c.CategoryId")
6：                      .QueryMany<Product>();
```

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.

使用 dynamic 自定义映射规则

```
1：  List<Product> products = Context.Sql(@"select * from Product")
2：                  .QueryMany<Product>(Custom_mapper_using_dynamic);
3：
4：  public void Custom_mapper_using_dynamic(Product product, dynamic row)
5：  {
6：      product.ProductId = row.ProductId;
7：      product.Name = row.Name;
8：  }
```

使用 datareader 进行自定义映射：

```
1：  List<Product> products = Context.Sql(@"select * from Product")
2：                  .QueryMany<Product>(Custom_mapper_using_datareader);
4：  public void Custom_mapper_using_datareader(Product product, IDataReader row)
5：  {
6：      product.ProductId = row.GetInt32("ProductId");
7：      product.Name = row.GetString("Name");
8：  }
```

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

```
1：  var products = new List<Product>();
2：  Context.Sql("select * from Product").QueryComplexMany<Product>(products, MapComplexProduct);
3：
4：  private void MapComplexProduct(IList<Product> products, IDataReader reader)
5：  {
6：          var product = new Product();
7：          product.ProductId = reader.GetInt32("ProductId");
8：          product.Name = reader.GetString("Name");
9：          products.Add(product);
10：   }
```

五 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。

```
1：  using (var command = Context.MultiResultSql)
2：  {
3：      List<Category> categories = command.Sql(
4：              @"select * from Category;
5：              select * from Product;").QueryMany<Category>();
6：
7：      List<Product> products = command.QueryMany<Product>();
8：  }
```

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。

六 分页

```
1：  List<Product> products = Context.Select<Product>("p.*, c.Name as Category_Name")
2：                           .From(@"Product p
3：                            inner join Category c on c.CategoryId = p.CategoryId")
4：                           .Where("p.ProductId > 0 and p.Name is not null")
5：                           .OrderBy("p.Name")
6：                           .Paging(1, 10).QueryMany();
```

调用 Paging(1, 10)，会返回最先检索到的10个 Product。

七 数据库操作（Insert, Update, Delete）

插入数据

使用 SQL 语句：

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

使用 builder：

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

使用 builder，并且自动映射

```
1：  Product product = new Product();
2：  product.Name = "The Warren Buffet Way";
3：  product.CategoryId = 1;
4：
5：  product.ProductId = Context.Insert<Product>("Product", product)
6：                  .AutoMap(x => x.ProductId)
7：                  .ExecuteReturnLastId<int>();
8：
```

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。

更新数据

使用 SQL 语句：

```
1：  int rowsAffected = Context.Sql(@"update Product set Name = @0 where ProductId = @1")
2：                  .Parameters("The Warren Buffet Way", 1)
3：                  .Execute();
```

使用 builder：

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

使用 builder，并且自动映射：

```
1：  Product product = Context.Sql(@"select * from Product
2：                  where ProductId = 1")
3：                  .QuerySingle<Product>();
4：  product.Name = "The Warren Buffet Way";
5：
6：  int rowsAffected = Context.Update<Product>("Product", product)
7：                  .AutoMap(x => x.ProductId)
8：                  .Where(x => x.ProductId)
9：                  .Execute();
```

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。

Insert and update - common Fill method

```
1：  var product = new Product();
2：  product.Name = "The Warren Buffet Way";
3：  product.CategoryId = 1;
4：
5：  var insertBuilder = Context.Insert<Product>("Product", product).Fill(FillBuilder);
6：
7：  var updateBuilder = Context.Update<Product>("Product", product).Fill(FillBuilder);
8：
9：  public void FillBuilder(IInsertUpdateBuilder<Product> builder)
10：  {
11：        builder.Column(x => x.Name);
12：        builder.Column(x => x.CategoryId);
13：  }
14：
15：  Delete
```

删除数据

使用 SQL 语句：

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

使用 builder：

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

八 存储过程和事务

存储过程

使用 SQL 语句：

```
1：  var rowsAffected = Context.Sql("ProductUpdate")
2：                  .CommandType(DbCommandTypes.StoredProcedure)
3：                  .Parameter("ProductId", 1)
4：                  .Parameter("Name", "The Warren Buffet Way")
5：                  .Execute();
```

使用 builder：

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

使用 builder，并且自动映射

```
1：  var product = Context.Sql("select * from Product where ProductId = 1")
2：                  .QuerySingle<Product>();
3：
4：  product.Name = "The Warren Buffet Way";
5：
6：  var rowsAffected = Context.StoredProcedure<Product>("ProductUpdate", product)
7：                  .AutoMap(x => x.CategoryId).Execute();
```

使用 Lambda 表达式

```
1：  var product = Context.Sql("select * from Product where ProductId = 1")
2：                  .QuerySingle<Product>();
3：  product.Name = "The Warren Buffet Way";
4：
5：  var rowsAffected = Context.StoredProcedure<Product>("ProductUpdate", product)
6：                  .Parameter(x => x.ProductId)
7：                  .Parameter(x => x.Name).Execute();
```

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。

```
1：  using (var context = Context.UseTransaction(true))
2：  {
```

```
3：          context.Sql("update Product set Name = @0 where ProductId = @1")
4：                          .Parameters("The Warren Buffet Way", 1)
5：                          .Execute();
6：
7：          context.Sql("update Product set Name = @0 where ProductId = @1")
8：                          .Parameters("Bill Gates Bio", 2)
9：                          .Execute();
10：
11：          context.Commit();
12：    }
13：
```

实体工厂

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

```
1：  List<Product> products = Context.EntityFactory(new CustomEntityFactory())
2：                          .Sql("select * from Product")
3：                          .QueryMany<Product>();
4：
5：  public class CustomEntityFactory ：  IEntityFactory
6：  {
7：          public virtual object Resolve(Type type)
8：          {
9：                  return Activator.CreateInstance(type);
10：           }
11：   }
```