

## ORM 框架 -VB/C#.Net 实体代码生成工具

### (EntitysCodeGenerate) 【 ECG】 4.2

#### 1 引言

#### 2 内容

##### 2.1 ORM 框架的实现： VB/C#.Net 实体代码生成工具 (EntitysCodeGenerate)

##### 2.2 在开发中的实际应用

###### 2.2.1 单个实体对象的数据库操作

- 1、 获取一个实体 (集)对象信息
- 2、 插入一个实体对象信息
- 3、 更新一个实体对象信息
- 4、 保存一个实体对象信息
- 5、 删除一个实体对象信息
- 6、 取得实体映射表数值字段的最大值 +1

###### 2.2.2 多个实体对象的数据库操作并结合事务处理

###### 2.2.3 数据查询及通用 DML 操作

- 1、 常用实体对象查询
- 2、 ORM结构化查询
  - (1)、 Select 查询
  - (2)、 From 连接查询
  - (3)、 Where 语句的 Condition 条件
  - (4)、 Order By 排序功能
  - (5)、 Group By 分组及分组条件和排序功能
  - (6)、 结合事务处理的功能

###### 3、 Delete 删除

###### 4、 Update 更新

###### 5、 Insert 插入

###### 2.2.4 DbCore+SQL/ 存储过程

###### 1、 DbCore+SQL

###### 2、 DbCore+存储过程

###### 2.2.5 Extend 辅助扩展功能

###### 1、 TableHelp 辅助扩展

###### 2、 CommonHelp 常用方法扩展

###### 2.2.6 ORM的分析及与 Xml 的交互

1、ORM的分析

2、与 XML的交互

3 结束语

4 相关下载地址

## 1 引言

目前大多数项目或产品都使用关系型数据库实现业务数据的存储，这样在开发过程中，常常有一些业务逻辑需要直接用写 SQL 语句实现，但这样开发的结果是：遍地布满 SQL 语句。这些藕合较高的 SQL 语句给系统的改造和升级带来很多无法预计的障碍。也许说可以使用服务端数据库存储子程序实现，这样只是将这种藕合搬迁到后端，问题依然没有根本解决，服务端驻留过多的存储子程序也消耗着服务器的性能并给多人合作维护和更新部署带来许多障碍。为了提高项目的灵活性，特别是快速开发，ORM 是一个不错的选择。举个简单的例子：在使用 ORM 的系统中，当数据库模型改变时，不再需要理会逻辑代码和 SQL 语句中涉及到该模型的所有改动，只需要将该模型映射的对象稍作改动，甚至不做改动就可以满足要求。

ORM 的全称是 Object Relational Mapping，即对象关系映射。它的实质就是将关系数据（库）中的业务数据用对象的形式表示出来，并通过面向对象（Object-Oriented）的方式将这些对象组织起来，实现系统业务逻辑的过程。在 ORM 过程中最重要的概念是映射（Mapping），通过这种映射可以使业务对象与数据库分离。从面向对象来说，数据库不应该和业务逻辑绑定到一起，ORM 则起到这样的分离作用，使数据库层透明，开发人员真正的面向对象。下图简单说明了 ORM 在多层系统架构中的这个作用。

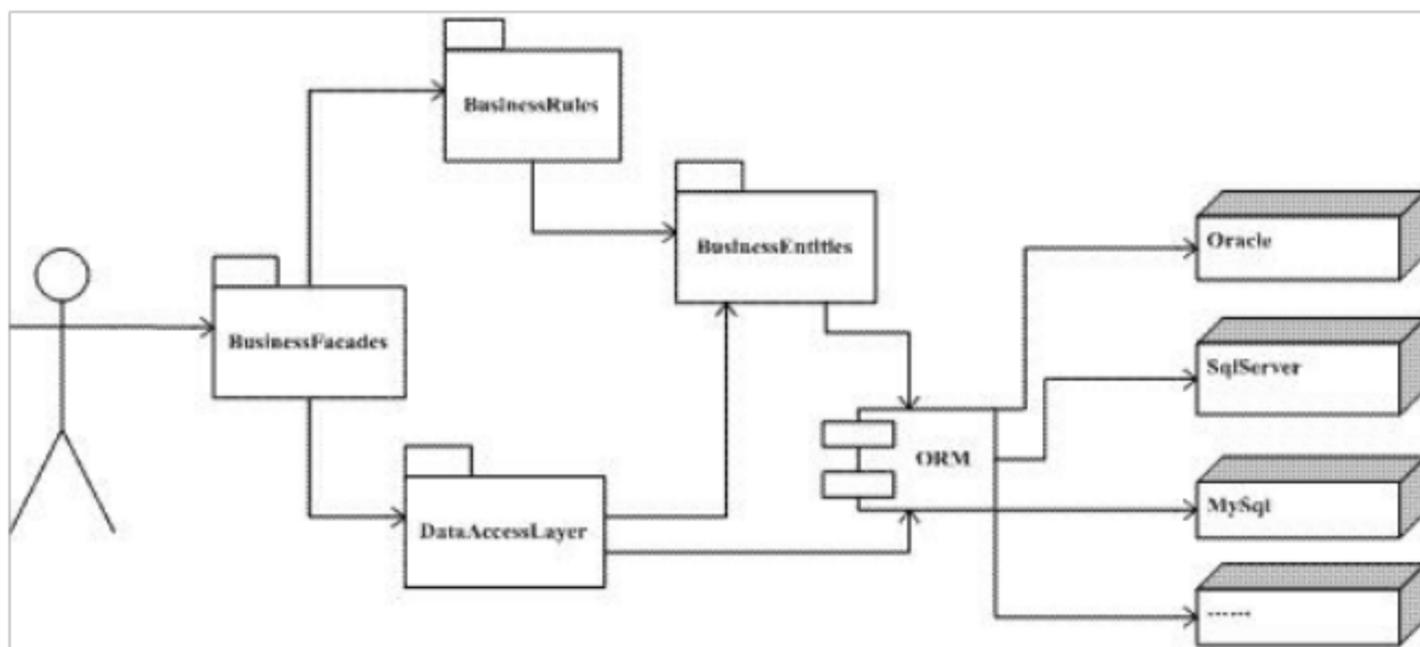


图 1 ORM 在多层系统架构中的作用

诚然 ORM 并非是万能的，面对纷繁复杂的业务逻辑，当遇到特别复杂的数据处理及海量数据运算或弥补设计的不足时还应归结到 SQL 或存储过程来实现才是好的选择，但它却很好地体现了“80/20（或 90/10）法则”（也被称为“帕累托法则”），也就是说：花比较少（10%-20%）的力气就可以解决大部分（80%-90%）

的问题，这样通过利用 ORM框架，我们就仅需要付出极少数时间和精力来解决剩下的少部分问题了，这无疑缩短了整个项目开发的周期，因此快速开发、面向对象和性能优化等必须灵活兼顾才好。ORM产品应当预留适当的接口来做性能优化并对特定功能的补充支持，这样才是一个好的产品，这些该工具都提供了很好的解决方案，下文分别作一简单介绍。

## 2 内容

### 2.1 ORM框架的实现：VB/C#.Net 实体代码生成工具 (EntityCodeGenerate)

好的 ORM工具不仅可以帮助我们很好的理解对象，而且工具本身会帮助我们记住字段属性业务含义并提供辅助的应用。基于这个理念，一个基于 .Net 的 ORM工具——VB/C#.Net 实体代码生成工具 (EntityCodeGenerate) 便应运而生，该工具运行于 .Net2.0，适用性广，开发后的代码部署要求不高，在 .Net 更高版本上也可以很好的运行。

VB/C#.Net 实体代码生成工具 (EntityCodeGenerate) 为 ORM提供对象持久、对象查询、事务处理等功能。数据持久包括对象的 Insert、Update、Save、Delete、Select 等功能，对象查询则提供一些基于对象的查询如 GetEntity 及构造函数获取对象信息和实体集等。该工具是基于 VS.NET2005的开发的应用程序，职责是从数据库中提取数据模型信息并生成实体类代码，帮助开发人员快速映射到关系数据库中的业务数据模型，最优化快速开发。目前提供直接从 Oracle、SqlServer、Access、MySQL Sybase、SQLite、DB2 PostgreSQL、DM(达梦)及支持 OleDb连接的数据库和 Custom(自定义)类型的数据库中生成 VB/C#.Net 代码的支持，可以生成实体和实体集的相关代码，并自动提取数据库表和字段的注释说明和对应的数据类型等信息。

另外所生成的代码文件只需修改数据库连接，即可用于目前市场上支持 ADO.NET的各种类型的数据库，如 Oracle、SqlServer、MySQL Access、Sybase、SQLite、DB2 PostgreSQL、Excel、Informix、Firebird、MaxDB DM(达梦)和 OleDb、ODBC连接的数据库等。所生成代码文件的类关系图如下所示：

工具 3.x 版本之后的实体层代码有个基类 (BaseEntity)，基类里其实也正是你项目的数据库连接的配置位置，该工具实际开发时的配置其实也就在这里，默认为生成代码时所填写的数据库连接信息，并可手工扩展修改（如从缓存或 config 配置文件中读取、实现对用户数据库连接信息的加密 /解密等；大多数时候我们只需在 GetConnectionString() 修改数据连接字符串即可）：

```
public class BaseEntity
```

```

{
    public static string GetConnectionString()
    {
        return "User ID=scott;Password=tiger;Data Source= oracle9" ;// 数据库连接可修改从别处读取
    }
    public static DatabaseType GetDatabaseType()
    {
        return DatabaseType . Oracle; // 数据库连接类型设置也可修改从别处读取
    }
    .....
}

```

这里可设置该命名空间当前实体下的数据库连接类型及数据库连接字符串，当然也可以修改成从其它如配置文件中读取（老版本是放在全局设置 DbConnectionString 这个类里面，缺陷是当一个项目有多个数据库时将不好处理；而现在通过使用基类继承及命名空间划分则很容易解决）。

System.Database.DbCore 结合实体类可将简单和复杂的数据库操作及事务处理更为方便的实现，下文着重介绍在实际中的使用。实体对数据库的操作引入保存操作，即将增加和更新合并为一个保存方法，默认以主键为准，当然也可以指定条件，由实体对象本身自己根据主键（没有主键的须指定条件字段）判断是增加还是更新操作。

## 2.2 在开发中的实际应用

VB/C#.Net 实体代码生成工具 (EntitysCodeGenerate) 安装后，在生成对应的实体代码后有个“相关配置”文件夹，里面有“配置说明”文档和相关文件，实际使用时只须按“配置说明”文档操作即可。使用该工具开发的项目，可以做到很好的切换到异构数据库，且只需变动数据库连接接口即可（即 GetDatabaseType()/GetConnectionString()）。同时提供了 ORMapping.dll（1.0 版本引用），System.Database.dll（2.0/3.x/4.x 版本引用）安装目录下有相应的 chm 格式帮助文档，是对这两个 dll 单独使用的说明，支持各种类型的数据库访问，并可结合生成的实体操作数据库，提供良好的事务处理等。其中 ORMapping.dll 支持 Oracle；System.Database.dll 默认支持 Oracle 数据库，并可用于 SqlServer、MySQL Sybase、DB2、SQLite、PostgreSQL、Informix、Firebird、MaxDB DM(达梦)、OleDb、Odbc 等。对没有直接提供的数据库可使用 OleDb 或 Odbc 连接。工具所生成的代码和提供的文件，都是可选配的，可单独使用也可配置使用。实体代码的生成界面比较简单，如下所示：



这里，只须选择数据库类型、输入正确的数据库连接字符串（Custom(自定义)的数据库类型填写正确的自定义程序集信息）、代码文件的输出目录和代码命名空间即可。实体数据类型以“数据类型映射文件”为准，工具提供了系统默认的类型映射，若有自定义的数据类型，在“数据类型映射文件”里修改，可配置到数据类型的精确刻度。数据类型初始值在“数据类型初始值文件”里修改，没有配置的取 .Net 默认值。若修改过程中想恢复默认的配置，只须点击右端“生成”按钮即可（注意不同语言的类型定义可能有所差异，注意区分。如：C#的 byte[]，在 VB 中的定义为 byte()）。准备工作做好后，单击“生成代码”按钮即可（若不想全部生成，单击“选择生成”按钮）。生成成功之后按提示生成的配置说明文档，将实体代码文件和基类文件拷贝到指定目录，并按生成代码文件所在“相关配置”目录下的配置说明文档，添加对应 dll 文件的引用即可。操作简单这里就不在此赘述，下面以 C#.Net 为例介绍实体对象的数据库操作的工作，因篇幅所限，VB.Net 同理对比使用。

## 2.2.1 单个实体对象的数据库操作

这里还是以 Oracle 附带库 DEPT 为例来做说明，首先做对象的声明

```
DEPT entity = new DEPT();
```

这里以该对象来做阐述。

### 1、获取一个实体（集）对象信息

实体的操作默认以主键为准，对单个实体信息的获取可简单如下实现：

```
entity . DEPTNO = 50;
entity = entity . GetEntity();
```

返回主键字段 DEPTNO=50 的对象，若数据库中没有对应的记录则返回 null。DEPT 表的主键字段是 DEPTNO 同时对联合主键也是支持的，下文同此；GetEntity 同时提供其它指定条件的重载，也可以使用 GetEntityByEntityCondition，当指定条件有多个记录符合时只返回首条记录；返回多条记录可使用 GetDataTable 方法。

在 3.3 版本之后也可通过多个构造函数来获取单个实体对象的信息等：

```
DEPT entity = new DEPT(50); // 通过主键字段条件获取实体对象
DEPT entity = new DEPT("DEPTNO 50"); // 指定唯一字段条件获取实体对象
DEPT entity = new DEPT(new string [] { "DEPTNO" }, new object [] { 50 });
```

另外，可通过实体集多个构造函数获取多实体对象信息的查询（实体集对象查询）等，如：

```
DEPTS entity = new DEPT(true); // 获取所有信息
EMPS entitys = new EMPS ("DEPTNO 50") // 获取 DEPTNO=50 所有雇员信息
EMPS entitys1 = new EMP(new string [] { "DEPTNO" }, new object [] { 50 });
DataTable dtbl = entitys . ToDataTable(); // 将获取的实体集对象信息转换到数据表 DataTable
```

### 2、插入一个实体对象信息

插入一个对象代码可如下所示：

```
entity . DEPTNO = 51;
entity . DNAME = "DNAME1";
entity . LOC = "LOC1";
entity . Insert();
```

同时 Insert 提供多种相应的重载和 InsertAll 方法；区别是：Insert 在插入记录时会比较对象初始的字段值，将与初始值不同的值插入，其余的以表字段的默认方式处理；InsertAll 则是插入全部字段，即使是对象的初始值没有改变也会插入。

### 3、更新一个实体对象信息

更新一个对象代码可如下所示：

```
entity . DEPTNO = 51;
```

```
entity . DNAME = " DNAME2";
entity . LOC = " LOC2";
entity . Update();
```

Update 也提供多种相应的重载和 UpdateAll 方法；区别是：Update 在更新记录时会比较对象初始的字段值，将与初始值不同的值进行更新，其余的表字段不更新；UpdateAll 则是更新全部字段，即使是对象的初始值没有改变也会将对象的初始值更新到表里。

#### 4、保存一个实体对象信息

保存是该工具新增的功能，即将新增和更新合并到一个保存功能里，ORM 会根据主键约束自动解析是新增还是更新，保存一个对象的代码可如下所示：

```
entity . DEPTNO = 51;
entity . DNAME = " DNAME3";
entity . LOC = " LOC3";
entity . Save();
```

保存操作默认以主键为准，对多个联合主键也是支持的，不带参数的保存默认是按主键判断，有就更新，没有就插入新记录。同时 Save 提供多种重载和 SaveAll、SaveByEntityCondition 方法，Save 和 SaveAll 区别同插入和更新。

#### 5、删除一个实体对象信息

删除可如下代码所示：

```
entity . DEPTNO = 51;
entity . Delete();
```

删除操作默认以主键为准，对多个联合主键也支持，同时也提供多种指定条件的重载方法。最后附加说明，实体对象的增删改保存操作都会返回一个 int 值，该值返回表中记录受影响的行数。

从这些代码可以明显的看到，这里常用的增、删、改、查操作只需很简单几句即可实现，少写了很多代码，岂不妙哉！

#### 6、取得实体映射表数值字段的最大值 +1

代码可如下：

```
int intID = entity . GetInt32MaxID();
```

这里获取实体对象对应表字段默认第一个主键的最大值 ID+1，类型为整型，同时提供 GetInt?MaxID 多种重载，即有整型和长整型及指定字段等重载。

除此之外，还提供了一系列的 CRUD 扩展方法，如：InsertEx / UpdateEx / SaveEx / DellInsert / DellInsertEx 和 实体集对象的批量操作，如：

entitys.ToDataTable / Save / SaveAll / SaveEx / Delete / DellInsert / DellInsertEx 等；详见示例代码、生成后的实体代码及相关帮助文档。其中实

体集对象的批量操作自动启用事务处理，操作成功统一提交，失败时统一回滚。

本节介绍的都是单表无事务的操作，下节介绍多表及事务处理的操作。

## 2.2.2 多个实体对象的数据库操作并结合事务处理

这里简略介绍实体对象结合 System.Database.DbCore 及事务处理是如何工作的，先看以下代码（可参见安装示例代码 System.Database.Demo）：

```

Entitys . CommonLC_WORKTYPEEntity = new Entitys . CommonLC_WORKTYPEEntity
entity . ID = 1;
entity . TYPENAME= "TYPENAME"
string strConnection = "Password=cc;User ID=cc;Data Source=oracle9" ;
DbCore dbCore = new DbCore( DatabaseType. Oracle, strConnection);
dbCore. Open();
dbCore. BeginTransaction();
dbCore. Save(new Entitys . CommonLC_WORKTYPEEntity entity);
entity . DESCRIPTION = "类型描述" ;
dbCore. Save(new Entitys . CommonLC_WORKTYPEEntity entity);
entity . TYPENAME= "作业类型" ;
dbCore. Save(new Entitys . CommonLC_WORKTYPEEntity entity);
DataSet dst = dbCore. ExecuteDataSet( "select * from lc_worktype" );

entity . ID = 1;
DataTable dt = dbCore. GetDataTableByEntityKey(entity);
int intRecord = dbCore. Delete(entity);
dt = dbCore. GetDataTableByEntityKey(entity);

dbCore. CommitTransaction();
dbCore. Close();

```

这里使用另外一个实体 LC\_WORKTYPE映射到 XX系统的"作业类型"这张表)，BeginTransaction() 为开始事务的标志；CommitTransaction() 为提交当前事务；还有一个是 RollbackTransaction() 表示回滚当前事务，放弃当前所有数据的更改。这样在事务开始和提交或回滚之间可以进行多个实体的操作，并将结果最终一起提交或回滚撤销。这里 Save 有两个参数第一个是实体对象的初始类用于比较实体的初始值，第二个是要保存的对象，该方法依据主键自动判断是更新还是插入；与 Save 类似的方法有 SaveAll，同时也有重载及 SaveEx Insert、InsertAll、InsertEx、Update、UpdaAll、Update Ex、Delete、IsExitByEntityKey、Exists、Get?MaxId 等方法，可相互结合使用，方法都有详尽的说明及示例代码。执行中可单步跟踪，查看该事务下每步命令执行后对应的数据集信息。

下面再看以 Oracle 自带的 scott 库为例一段代码 Delete、Insert、Update 并结合事务使用的代码：

```

DbCore dbCore = null ;
try
{

```

```

EMP entity1 = new EMP();
DataSet dst = new DataSet ();
entity1 .EMPNO = 7369; // 设置主键 EMPNO为
entity1 = entity1 .GetEntity(); // 取得主键 EMPNO为实体对象信息
// "User ID=scott;Password=tiger;Data Source=oracle9";
dbCore = new DbCore(Entitys . CommarBaseEntity . GetConnectionString());
dbCore. Open();
dbCore. BeginTransaction();
// 选择当前事务下的所有雇员 EMP的信息
dst = dbCore. SelectAll() . From(entity1) . ExecuteDataSet();
dbCore. Delete(entity1); // 删除主键 EMPNO为 7369 的记录
dst = dbCore. SelectAll() . From(entity1) . ExecuteDataSet(); // 查看当前事务下记录，当前删除记录将不在此显示
dbCore. Insert( new EMP(), entity1); // 插入刚才删除主键 EMPNO为 7369 的记录
=dbCore.Save(new EMP(), entity1);
dst = dbCore. SelectAll() . From(entity1) . ExecuteDataSet(); // 查看当前事务下记录，可见刚刚插入的新记录
entity1 .SAL = entity1 .SAL + 100; // 薪水加 100
dbCore. Update( new EMP(), entity1); // 更新=dbCore.Save(new EMP(), entity1);
dst = dbCore. SelectAll() . From(entity1) . ExecuteDataSet(); // 查看当前事务下记录，对应薪水 SAL已更新
entity1 .SAL = entity1 .SAL - 100; // 薪水减 100
dbCore. Update( new EMP(), entity1); // 更新=dbCore.Save(new EMP(), entity1);
dst = dbCore. SelectAll() . From(entity1) . ExecuteDataSet(); // 查看当前事务下记录，对应薪水 SAL已更新
dbCore. CommitTransaction();
dbCore. Close();
}
catch (Exception ex)
{
    if (dbCore != null )
    {
        if (dbCore . IsTransaction)
            dbCore. RollbackTransaction(); // 如果已经开始事务，则回滚事务
        dbCore. Close();
    }
}
}

```

上面的 Insert 、 Update 方法都可以 Save 方法来取代， Save 方法会自动判断是 Update 还是 Insert ，这里只是用来展示之用。

### 2.2.3 数据查询及通用 DML操作

如一般的关系数据库所具有的查询功能一样， EntityCodeGenerate 也有着非常丰富的查询功能，如对象查询、数据集查询、函数查询、条件查询、排序查询、分组等。这里对数据查询做下简单介绍。 ECG提供 ENTITYColumn类(或 ENTITY.s\_静态属性)帮助 System.Database.DbCore.Select 从数据源查询数据， Select 通过 ENTITYColumn实体属性(或 ENTITY.s\_静态属性，下同)，构造查询语句，执行并返回结果数据集。

### 1、常用实体对象查询

常用实体对象查询，见上一节的第一段所述，这里就不在赘述。

### 2、ORM结构化查询

这里先介绍表实体原描述信息类 ENTITYColumn, 这里“ ENTITY”为表实体类对应的占位符，其类下有实体所对应的表的表名和表字段的公共静态只读信息。其中的 TableName为每个该类的对应表名，各属性字段为表名、字段名、字段对应类型的完全限定名，以键盘不能直接输入的偏僻符号 ‘ ’ 为分割符拼接而成。

ORM结构化查询时可使用该类的属性(或 ENTITY.s\_静态属性，下同)。

#### (1)、Select 查询

System.Database.DbCore 类提供 Select 方法及其相应的重载，以 SqlServer 自带的 pubs 库为例，可见如下代码：

```

DataSet dst = new DataSet ();

DbCore dbCore =
new DbCore( DatabaseType.SqlServer, "Server=(local);User id=sa;Pwd=sa;Database=pubs" );

DataSet dst = dbCore.SelectAll().From("sales").ExecuteDataSet();

DataSet dst = dbCore.SelectAll().From(SALESColumnTableName).ExecuteDataSet();

DataSet dst =
dbCore.Select( SALESColumnord_num).From(SALESColumnTableName).ExecuteDataSet();

DataSet dst =
dbCore.Select( SALESColumnord_num).SelectColumn( SALESColumnord_date).From(SALESColumnTable
Name) ExecuteDataSet();

DataSet dst = dbCore.SelectAll().From(new SALE$).ExecuteDataSet();
  
```

其中的 dbCore.SelectAll() 方法缺省默认参数为查询所有字段， Select() 不选择出任何字段，仅初始化实例，同时也可直接使用 SQL语法的字符串或使用相应的 ENTITYColumn类如 SALESColum的属性字段作为参数，或数组集合信息。也可用 SelectColumn 添加要查询的字段， SelectColumn( ...) 接受 SQL表字段名字符串或 ENTITYColumn类的属性字段格式的参数， SelectColumns( ...) 接受符合 SelectColumn 格式的数组参数，同时也提供 SelectCustom( ...) 接受自定义的 SQL语法的字符参数。同时鉴于 SelectColumn 等方法名较长，提供 Add替代

SelectColumn , AddMax Min、 Avg 分别替代 SelectColumnMax、 Min、 AvgValue 各种同构方法，便于代码的抒写。

From(...) 为查询的目标表名，这里使用的是 SALESColumn.TableName 也可直接使用表名字符串，同时接受实体对象作为参数。 ExecuteDataSet() 执行查询并返回在内存当中的 DataSet 格式的结果数据集。 ExecuteReader() 执行查询并返回只读向前的数据结果集流 IDataReader；ExecuteScalar() 快速执行查询并返回第一行第一列的 Object 值，下同。

## (2)、From 连接查询

先见如下代码：

```

DataSet dst = new DataSet (); DataSet dst1 = new DataSet ();

DbCore dbCore =
new DbCore( DatabaseType .SqlServer, "Server=(local);User id=sa;Pwd=sa;Database=pubs" );

dst = dbCore. SelectAll() . From() . JoinInner( "sales" , "stor_id" , "stores" , "stor_id" )
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinInner( SALESColumnstor_id, STORESColumrstor_id)
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinLeft( "sales" , "stor_id" , "stores" , "stor_id" )
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinLeft( SALESColumnstor_id, STORESColumrstor_id)
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinRight( "sales" , "stor_id" , "stores" , "stor_id" )
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinRight( SALESColumnstor_id, STORESColumrstor_id)
. ExecuteDataSet()

dst = dbCore. SelectAll() . From() . JoinFull( "sales" , "stor_id" , "stores" , "stor_id" )
. ExecuteDataSet();

dst = dbCore. SelectAll() . From() . JoinFull( SALESColumnstor_id, STORESColumrstor_id)
. ExecuteDataSet();

```

这里是以 SqlServer 系统自带 pubs 示例库的表 sales、 stores 为例介绍的，JoinInner 为内连接、JoinLeft 为左外连接、JoinRight 为右外连接、JoinFull 完全外连接。参数可直接使用表名及表字段名字符串，或使用 SALESColumn STORESColum 类，其中使用实体类静态属性字段代码可读性及维护性更好些。

下面看下 Where 的使用。

## (3)、Where 语句的 Condition 条件

```

DataSet dst = new DataSet ();

```

```

DataSet dst1 = new DataSet ();

DbCore dbCore =
new DbCore( DatabaseType . SqlServer, "Server=(local);User id=sa;Pwd=sa;Database=pubs" );

```

```

dst = dbCore. SelectAll() . From() . JoinInner( "sales" , "stor_id" , "stores" , "stor_id" )
. Where() . ConditionAndEqual( "sales" , "stor_id" , 7067) . ExecuteDataSet();

```

```

dst1 = dbCore. SelectAll() . From() . JoinInner( SALESColumnstor_id, STORESColumrstor_id)
. Where() . ConditionAndEqual( SALESColumnstor_id, 7067) . ExecuteDataSet();

```

```

dst = dbCore. SelectAll() . From() . JoinInner( "sales" , "stor_id" , "stores" , "stor_id" )
. Where() . ConditionAndGreat( "sales" , "stor_id" , 7896)
. ConditionOrLessEqual( "sales" , "stor_id" , 7067) . ExecuteDataSet();

```

```

dst1 = dbCore. SelectAll() . From() . JoinInner( SALESColumnstor_id, STORESColumrstor_id)
. Where() . ConditionAndGreat( SALESColumnstor_id, 7896)
. ConditionOrLessEqual( SALESColumnstor_id, 7067) . ExecuteDataSet();

```

```

dst = dbCore. SelectAll() . From("sales" ) . FromTable( "stores" )
. Where() . ConditionColumnAndEqual( "sales" , "stor_id" , "stores" , "stor_id" ) .
ConditionAndBetweenAnd( "sales" , "stor_id" , 7067, 7896) . ExecuteDataSet();

```

```

dst1 =dbCore. SelectAll() . From( SALESColumnTableName) . FromTable( STORESColumrTableName)
. Where() . ConditionColumnAndEqual( SALESColumnstor_id, STORESColumrstor_id) .
ConditionAndBetweenAnd( SALESColumnstor_id, 7067, 7896) . ExecuteDataSet();

```

这里再次展示了使用字符串和 ENTITYColumn类的相互比较，可见使用 ENTITYColumn更直观，可读及维护性更好些。 Where() 是不带参数的实例化方法，所添加的查询条件均以 Condition ...开头，在这里可以添加关系运算包括 Equal(=), Less(<), Great(>), LessEqual(<=), GreatEqual (>=), NotEqual(<>、!=), BetweenAnd, in, not in, null, not null,like,not like 和各自的关系 and,or 比较 及表字段与字段之间的 =,<, >, <= , >=,<>关系 and,or 的连接等；另外还可以添加自定义的 Where条件语句等。

#### (4)、Order By 排序功能

这里切换到 Oracle 数据库，以 Oracle 自带的 scott 用户为例，先看如下代码：

```

DataSet dst = new DataSet ();

EMP entity = new EMP();

DbCore dbCore = new DbCore( DatabaseType . Oracle, "Password=tiger;User ID=scott;Data
Source=oracle9" );

```

```

dst = dbCore. SelectAll() . From(entity)

```

```
. Where() . ConditionAndGreat(entity,      EMPColumnDEPTNO,20)
. OrderBy( EMPColumnSAL). ExecuteDataSet();
```

```
dst = dbCore. SelectAll() . From(entity)
. Where() . ConditionAndGreat(entity,      EMPColumnDEPTNO,20)
. OrderBy() . Asc( EMPColumnSAL). ExecuteDataSet();
```

```
dst = dbCore.SelectAll().From(entity)
.Where().ConditionAndGreat(entity,      EMPColumn.DEPTNO20)
.OrderBy().Desc(EMPColumn.SAL).ExecuteDataSet();
```

```
dst = dbCore. SelectAll() . From(EMPColumnTableName)
. Where() . ConditionAndGreat( EMPColumnDEPTNO,20) . ExecuteDataSet();
```

其中 entity 为雇员表 EMP对应的实体对象，这里使用查询表名的地方也可直接用实体对象。 OrderBy 默认不带排序参数，只实例化对象，也可带排序字段，排序方式为升序；其后使用 Asc 添加升序字段， Desc 添加降序字段。

#### (5)、Group By 分组及分组条件和排序功能

这里同样以 Oracle 自带的 scott 用户为例，先看如下代码：

```
DataSet dst = new DataSet ();
DataSet dst1 = new DataSet ();
EMP entity = new EMP();
DbCore dbCore = new DbCore( DatabaseType. Oracle, "Password=tiger;User ID=scott;Data
Source=orac" );
```

```
dst = dbCore. Select() . SelectColumnMaxValue( EMPColumnEMPNO)
. SelectColumnMinValue( EMPColumnEMPNO)SelectColumnAvgValue( EMPColumnEMPNO)
. From(EMPColumnTableName)
. GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)
. Having() . ConditionAndGreatEqual( EMPColumnDEPTNO,10)
. ConditionAndLessEqual( EMPColumnDEPTNO,40) . ExecuteDataSet();
```

```
dst1 = dbCore. Select() . AddMax(EMPColumnEMPNO)
. AddMin( EMPColumnEMPNO)AddAvg(EMPColumnEMPNO)
. From(EMPColumnTableName)
. GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)
. Having() . ConditionAndGreatEqual( EMPColumnDEPTNO,10)
. ConditionAndLessEqual( EMPColumnDEPTNO,40) . ExecuteDataSet();
```

```
dst = dbCore. Select() . SelectColumn( EMPColumnDEPTNO)SelectColumn( EMPColumnSAL)
. SelectColumnMaxValue( EMPColumnEMPNO)SelectColumnMinValue( EMPColumnEMPNO)
. SelectColumnAvgValue( EMPColumnEMPNO)
```

<pre> . From(EMPColumnTableName)  . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)  . Having() . ConditionAndGreatEqual(  EMPColumnDEPTNO,10)  . ConditionAndLessEqual(  EMPColumnDEPTNO,40)  . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet(); </pre>
<pre> dst1  =  dbCore. Select() . Add(EMPColumnDEPTNO)Add(EMPColumnSAL)  . AddMax(EMPColumnEMPNO)AddMin( EMPColumnEMPNO)AddAvg( EMPColumnEMPNO)  . From(EMPColumnTableName)  . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)  . Having() . ConditionAndGreatEqual(  EMPColumnDEPTNO,10)  . ConditionAndLessEqual(  EMPColumnDEPTNO,40)  . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet(); </pre>
<pre> dst  =dbCore. Select() . SelectColumn( EMPColumnDEPTNO)SelectColumn( EMPColumnSAL)  . SelectColumnMaxValue( EMPColumnEMPNO)SelectColumnMinValue( EMPColumnEMPNO)  . SelectColumnAvgValue( EMPColumnEMPNO)  . From(EMPColumnTableName)  . Where() . ConditionAndBetweenAnd( EMPColumnMGR, 7698,  7788)  . ConditionAndGreatEqual(  EMPColumnHIREDATE, new DateTime(1981,  5,  1))  . ConditionAndLessEqual(  EMPColumnHIREDATE, DateTime. Today)  . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)  . Having() . ConditionAndGreatEqual(  EMPColumnDEPTNO,10)  . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet(); </pre>
<pre> dst1  =  dbCore. Select() . Add( EMPColumnDEPTNO)Add(EMPColumnSAL)  . AddMax(EMPColumnEMPNO)AddMin(EMPColumnEMPNO)AddAvg(EMPColumnEMPNO)  . From(EMPColumnTableName)  . Where() . ConditionAndBetweenAnd( EMPColumnMGR, 7698,  7788)  . ConditionAndGreatEqual(  EMPColumnHIREDATE, new DateTime(1981,  5,  1))  . ConditionAndLessEqual(  EMPColumnHIREDATE, DateTime. Today)  . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)  . Having() . ConditionAndGreatEqual(  EMPColumnDEPTNO,10)  . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet(); </pre>
<pre> dst  =  dbCore. Select() . SelectColumn( EMPColumnDEPTNO)SelectColumn( EMPColumnSAL)  . SelectColumnMaxValue( EMPColumnEMPNO)SelectColumnMinValue( EMPColumnEMPNO) </pre>

```

    . SelectColumnAvgValue( EMPColumnEMPNO)
    . From(EMPColumnTableName)
    . Where() . ConditionAndGreatEqual( EMPColumnEMPNO,7654)
    . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL). ExecuteDataSet();
  
```

```

dst = dbCore. Select() . SelectColumn( EMPColumnDEPTNO)SelectColumn( EMPColumnSAL)
    . SelectColumnMaxValue( EMPColumnEMPNO)SelectColumnMinValue( EMPColumnEMPNO)
    . SelectColumnAvgValue( EMPColumnEMPNO)
    . From(EMPColumnTableName)
    . Where() . ConditionAndGreatEqual( EMPColumnEMPNO,7654)
    . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)
    . ConditionAndLessEqual( EMPColumnDEPTNO,40)
    . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet();
  
```

```

dst = dbCore. Select() . SelectColumn( EMPColumnDEPTNO)SelectColumn( EMPColumnSAL)
    . SelectColumnMaxValue( EMPColumnEMPNO)SelectColumnMinValue( EMPColumnEMPNO)
    . SelectColumnAvgValue( EMPColumnEMPNO)
    . From(EMPColumnTableName)
    . Where() . ConditionAndBetweenAnd( EMPColumnMGR,7698, 7788)
    . GroupBy( EMPColumnDEPTNO)Column(EMPColumnSAL)
    . Having() . ConditionAndGreatEqual( EMPColumnDEPTNO,10)
    . ConditionAndLessEqual( EMPColumnDEPTNO,40)
    . OrderBy( EMPColumnDEPTNO)Asc( EMPColumnSAL). Asc( "3" ) . ExecuteDataSet();
  
```

这里 dst、dst1 变量分别对应同样功能不同语句写法的数据集信息。同时展示了 Add 替代 SelectColumn，AddMax、Min、Avg 分别替代 SelectColumnMax、Min、AvgValue 的示例代码。Where 和 GroupBy、Having、OrderBy 可同时使用，也可分开使用。最后三个展示了分别以 Where 和 GroupBy、Having、OrderBy 分开使用的例子。其中分组使用 Column(...) 添加分组字段。可以很明显的看出些语句的功能，和 SQL 语句的书写几乎一致。如：倒数第四、五段的代码（粗体部分，两段代码功能相同，只是使用方法略异）换算成 SQL 语句就是（假设今天是 2009-10-11）：

```

SELECT emp.deptno, emp.sal,
MAX(emp.empno) AS max_emp_empno, MIN(emp.empno) AS min_emp_empno, AVG(emp.empno) AS
avg_emp_empno
FROM emp
WHERE emp.mgr BETWEEN 7698 AND 7788
AND emp.hiredate >= to_date('1981-5-1', 'yyyy-mm-dd') AND
  
```

```
emp.hiredate <= to_date('2009-10-11', 'yyyy-mm-dd')
GROUP BY emp.deptno, emp.sal
HAVING emp.deptno >= 10
ORDER BY deptno ASC, sal ASC, 3 ASC
```

和直接编写 SQL 很相似，并无须编写参数替换等的编码，省去许多代码量，且可读性也高，维护也方便。

### (6)、结合事务处理的功能

(1)-(5) 介绍的没有加入事务处理功能，下面介绍结合事务的使用，先看如下代码：

```
DataSet dst = new DataSet ();
DbCore dbCore = new DbCore( DatabaseType. Oracle, "Password=tiger;User ID=scott;Data
Source=orac" );
try
{
    #region 使用事务
    dbCore. Open(); //-- 打开数据库连接
    dbCore. BeginTransaction(); // 开始事务
    int intRecordCount = dbCore. DeleteFrom( EMPColumnName). ExecuteNonQuery();
    dst = dbCore. SelectAll() . From(EMPColumnName). ExecuteDataSet();
    dbCore. RollbackTransaction(); // 回滚撤销事务
    dst1 = dbCore. SelectAll() . From(new EMP) . ExecuteDataSet();
    dbCore. Close(); //-- 关闭数据库连接
    #endregion
}
catch (Exception ex)
{
    if (dbCore != null )
    {
        dbCore. Close();
    }
    MessageBox Show(ex. Message);
}
```

这里仍然是以 Oracle 自带的 scott 用户为例，其中的 Data Source 可以选择任何 Oracle 的服务名。这里有必要说明的是使用了打开、关闭数据库连接，因为要使用事务所以需要先打开数据库连接，前文介绍没有使用事务的，该步骤略过了。该例的功能是先删除 EMP 表的所有数据再查询该表的所以信息，可看到在当前事务下是没有信息的，然后在回滚事务，再查询就又有数据了，这正是事务所起的作用。同时需要注意的是，数据库连接打开后要关闭，且当中间出现异常时也应关闭已打开的数据库连接，以释放资源。

### 3、Delete 删除

仍然以 Oracle 自带的 scott 用户为例，并结合事务处理，先看如下代码：

```

try
{
    DataSet dst = new DataSet();
    dst = dbCore.SelectAll().From(new EMP).ExecuteDataSet();
    dbCore.Open();/-- 打开数据库连接
    dbCore.BeginTransaction(); // 开始使用事务
    EMP entity = new EMP();
    entity.EMPNO = 7782;
    intRecordCount = dbCore.Delete(entity);
    dst = dbCore.SelectAll().From(EMPColumnName).ExecuteDataSet();
    intRecordCount = dbCore.DeleteFrom(EMPColumnName)
        .Where().ConditionAndEqual(EMPColumnDEPTNO,10).ExecuteNonQuery();
    dst = dbCore.SelectAll().From(EMPColumnName).ExecuteDataSet();
    intRecordCount = dbCore.DeleteFrom(new EMP).ExecuteNonQuery();
    dst = dbCore.SelectAll().From(EMPColumnName).ExecuteDataSet();
    dbCore.RollbackTransaction(); // 回滚结束事务
    dst = dbCore.SelectAll().From(EMPColumnName).ExecuteDataSet();
    dbCore.Close();/-- 关闭数据库连接
}
catch (Exception ex)
{
    if (dbCore != null) dbCore.Close();
}

```

这里先声明一个 EMP 雇员类的实体对象 entity，并将主键 EMPNO 赋以值 7782，然后执行 Delete 操作，即将主键为 7782 的记录删除，接下来的查询跟踪运行可看到。下面的 DeleteFrom 方法，并结合 Where 条件，将部门编号 DEPTNO 等于 10 的雇员信息全部删除，同样再下面的查询跟踪运行可看到。最后将雇员表 EMP 的记录全部删除之，再查询没有任何信息。之后回滚事务，再次查询，又有数据。关于使用实体对象插入 (Insert)、更新 (Update)、删除 (Delete)、保存 (Save) 操作可参见 (上篇) 或示例代码。

#### 4、Update 更新

仍然以 Oracle 自带的 scott 用户为例，并结合事务处理，先看如下代码：

```

DbCore dbCore = null;
try
{
    dbCore = PublicClass.GetNewDbCore();
    dbCore.Open();
    dbCore.BeginTransaction();
    EMP entity = new EMP();
    entity.EMPNO = 7499;

```

```

entity = entity . GetEntity(dbCore);
dbCore. Update( EMPColumnNameTableName). Set( EMPColumnSAL, entity .SAL + 100)
    . Set( EMPColumnCOMM,
entity . COMM+ 100) . Set( EMPColumnHIREDATEDateTime. Today)
    . Where() . ConditionAndEqual( EMPColumnEMPNO,7499) . ExecuteNonQuery();
DataSet dst = dbCore. SelectAll() . From(EMPColumnNameTableName)
    . Where() . ConditionAndEqual( EMPColumnEMPNO,7499) . ExecuteDataSet(); // 查询
dbCore. Update( EMPColumnNameTableName). Set( EMPColumnSAL, entity .SAL)
    . Set( EMPColumnCOMMentity . COMM) . Set( EMPColumnHIREDATE,
entity . HIREDATE)
    . Where() . ConditionAndEqual( EMPColumnEMPNO,7499) . ExecuteNonQuery(); // 恢复
原值
dst = dbCore. SelectAll() . From(EMPColumnNameTableName)
    . Where() . ConditionAndEqual( EMPColumnEMPNO,7499) . ExecuteDataSet(); // 查询
dbCore. CommitTransaction(); // 提交事务
dbCore. Close();
}
catch (Exception ex)
{
    if (dbCore != null )
    {
        if (dbCore . IsTransaction)
        {
            dbCore. RollbackTransaction(); // 如果已经开始事务，则回滚事务
        }
        dbCore. Close();
    }
    throw ex;
}

```

这里先声明一个 EMP雇员类的实体对象 entity ，并将主键 EMPNO赋以值 7499 ，然后执行获取信息到对应的实体对象， 查询跟踪运行可见到。之后分别将该员工的薪水 SAL和 COMM分别加 100 和将 HIREDATE更新为今天，再查询可见到刚刚更新的记录数据，然后又将该员工的 SAL、COMM和 HIREDATE更新恢复原值，跟踪可见查询的数据信息，最后提交事务，该段程序只是介绍演示之用。

## 5、Insert 插入

用法同上，不再详细赘述，这里换张表以作区别，可见示例代码，如下所示：

```

DbCore dbCore = null ;
try
{
    dbCore = GetDbCore;
    dbCore. Open();
    int count = dbCore. InsertInto( T_DEMOs_TableName). Values( T_DEMOs_C_ID,GetKeyId)

```

```

        . Values( T_DEMOs_C_NAME, "NameInsert" ). Values( T_DEMOs_C_IDCARD;"3402211966
06066066" )
        . Values( T_DEMOs_C_DATE, DateTime.Today) . Values( T_DEMOs_C_INT, 10)
        . Values( T_DEMOs_C_FLOAT;1.11 ) . Values( T_DEMOs_C_EIDTDATE, DateTime.Now)
        . ExecuteNonQuery();
    dbCore. Close();
    return count;
}
catch (Exception ex)
{
    if (dbCore != null ) dbCore. Close();
    throw ex;
}

```

#### 2.2.4 DbCore+SQL存储过程

诚然 ORM并不是万能的，当遇到特别复杂的数据处理、海量数据运算、性能瓶颈优化或弥补设计的不足时，还应归结到 SQL语句或存储过程来实现才是好的选择。实际项目中仍然会有海量复杂的数据处理或复杂子查询、SQL语句优化和存储过程等，DbCore提供了良好的解决方案，很好的解决了项目中 10%-20%的那部分功能。该组件支持目前市场上 ADO.NET支持的各种类型的数据库，可执行自定义编写的 SQL语句和存储过程等，可针对复杂功能的特殊处理并对项目瓶颈问题作性能优化等操作。

System.Database.DbCore 可直接用于 Oracle、SqlServer、MySql、Sybase、DB2、SQLite、PostgreSQL、Informix、Firebird、MaxDB、Access、DM(达梦)和支持 OleDb、Odbc连接类型的数据库，实例代码分别如下：

DbCore dbCore = new DbCore( DatabaseType. Oracle, " OracleConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. SqlServer, " SqlServerConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. MySql, " MySqlConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. Sybase, " SybaseConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. DB2, " DB2ConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. SQLite, " SQLiteConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. PostgreSQL, " PostgreSQLConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. Informix, " InformixConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. Firebird, " FirebirdConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. MaxDB, " MaxDBConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. Access, " AccessConnectionString " );
DbCore dbCore = new DbCore( DatabaseType. Dm, " DmConnectionString " );

```
DbCore dbCore = new DbCore( DatabaseType. OleDb, " OleDbConnectionString ");
```

```
DbCore dbCore = new DbCore( DatabaseType. Odbc, " OdbcConnectionString ");
```

## 1、DbCore+SQL

下面看一段适合 Oracle 和 SqlServer 访问的通用 SQL代码：

```
DbCore dbCore = PublicClass . GetNewDbCore();
string strParaToken = dbCore.GetCurrentParameterToken; // 对应数据库的参数前导符
string strSql = "INSERT INTO dept (deptno, dname, loc) VALUES
(" + strParaToken + "deptno, " + strParaToken + "dname,
" + strParaToken + "loc) ";
dbCore. Open(); // 打开数据库连接
dbCore. BeginTransaction(); // 开始事务

DBCommandWrapper cmd = dbCore. GetSqlCommandWrapper(strSql);
//cmd.AddParameter(..);// 为命令增加一个参数实例
cmd AddInParameter(strParaToken + "deptno", DbType Int32, 99);
cmd AddInParameter(strParaToken + "dname", DbType. String, "部门名称 ");
cmd AddInParameter(strParaToken + "loc", DbType AnsiString, "locTest" );

int intMaxDeptId = dbCore. GetInt32MaxId( "dept", "deptno" ); // 当前表的 deptno 最大值

dbCore. ExecuteNonQuery(cmd);
intMaxDeptId = dbCore. GetInt32MaxId( "dept", "deptno" ); // 插入数据 deptno=99 之后当前表的
deptno 最大值

strSql = "DELETE dept WHERE deptno = " + strParaToken + "deptno" ;
DBCommandWrapper cmd1 = dbCore. GetSqlCommandWrapper(strSql);
cmd1. AddInParameter(strParaToken + "deptno", DbType Int32, 99);
dbCore. ExecuteNonQuery(cmd1);
intMaxDeptId = dbCore. GetInt32MaxId( "dept", "deptno" ); // 删除数据 deptno=99 之后当前表的
deptno 最大值

dbCore. RollbackTransaction(); // 回滚撤销事务。等于该方法什么都没做，只是演示作用

intMaxDeptId = dbCore. GetInt32MaxId( "dept", "deptno" );
dbCore. Close(); // 关闭数据库连接
```

其中第一句的 PublicClass . GetNewDbCore()方法体代码可以是  
new DbCore( DatabaseType. Oracle, " OracleConnectionString ")也可以是  
new DbCore( DatabaseType. SqlServer, " SqlServerConnectionString ")，当  
new的是 Oracle 时表示操作访问的是 Oracle 数据库，当 new的是 SqlServer  
即表示操作访问的是 SqlServer 数据库。dbCore.GetCurrentParameterToken 即

是获取对应数据库连接参数的前导符（如： Oracle 是“：”，SQL Server 是“@”等），这里也可以结合使用 `dbCore.StandardSqlWithParameters` 方法对当前带参数的 SQL 语句进行标准化通用处理，即所写 SQL 可以用于如 MySQL/Access 数据库等。这里的数据库操作同样也是可以同实体对象一块协同工作。

`dbCore.GetSqlCommandWrapper(...)` 创建一个 SQL 语句的命令，  
`dbCore.GetStoredProcCommandWrapper(..)` 创建一个执行存储过程的命令，可根据项目自身实际需要选择使用。

对专有数据库命令也可以转化为指定数据库命令来使用，这样可针对该数据库特性使用更多的方法，如 Oracle、SqlServer 的命令转化可像下列代码来转化：

```
OracleCommandWrapper cmd = dbCore.GetSqlCommandWrapper(strSql) as OracleCommandWrapper;
SqlCommandWrapper cmd = dbCore.GetSqlCommandWrapper(strSql) as SqlCommandWrapper;
.....
```

### 不带参数 SQL 语句的快捷用法

```
DataSet dst = dbCore.ExecuteDataSet(strSql);
DataTable dtbl = dbCore.ExecuteDataSet(strSql).Tables[0];
```

这里顺便说明一下，当程序执行出现异常时可使用 `dbCore.Close()` 来关闭当前打开的数据库连接，如下代码所示：

```
catch (Exception ex)
{
    if (dbCore != null)
    {
        if (dbCore.IsTransaction)
        {
            dbCore.RollbackTransaction(); // 如果已经开始事务，则回滚事务
        }
        dbCore.Close();
    }
    throw ex;
}
```

最后再说一个

`System.Database.DbCoreConnectLimit.AllDBMaxConnectionCount`，可以设置数据库打开的最大连接数目，默认不受限制。

## 2、DbCore-存储过程

DbCore-存储过程标准用法：

```
DBCommandWrapper cmd = dbCore.GetStoredProcCommandWrapper("[包名.] 存储过程名");
```

```
//cmd.AddInParameter(...  
dbCore. ExecuteNonQuery(cmd);  
OracleCommandWrapper cmd = dbCore.GetStoredProcCommandWrapper("[包名.] 存储过程名  
") asOracleCommandWrapper;  
//cmd.AddInParameter(...  
//cmd.AddOutParameter(...  
//cmd.AddParameter(...  
//cmd.AddCursorOutParameter(...  
dbCore. ExecuteDataSet(cmd);
```

同时 DbCore 也提供了对存储过程的一个快捷用法：

```
dbCore. ExecuteStoredProcudure( "[包名.] 存储过程名 " );
```

下面就以 Oracle 为例，看下 DbCore-存储过程的具体写法：

```
DbCore dbCore = null ;  
try  
{  
    string strConnection = "Password=cc;User ID=cc;Data Source=oracle9" ;  
    DbCore dbCore = new DbCore(DatabaseType. Oracle, strConnection);  
    dbCore. Open();  
    dbCore. BeginTransaction();  
    int count;  
    OracleCommandWrapper cmd =dbCore. GetStoredProcCommandWrapper("storedProcedure.Name  
A") as OracleCommandWrapper;  
    //cmd.AddInParameter(  
    //cmd.AddOutParameter(  
    //cmd.AddParameter(  
    //cmd.AddCursorOutParameter(  
    count = dbCore. ExecuteNonQuery(cmd);  
    cmd = dbCore. GetStoredProcCommandWrapper("storedProcedure.NameB" ) asOracleCommand  
Wrapper;  
    //cmd.AddInParameter(  
    count += dbCore. ExecuteNonQuery(cmd);  
    //count += dbCore.ExecuteStoredProcudure("storedProcedure.NameC");  
    dbCore. CommitTransaction();  
    dbCore. Close();  
    return count;  
}  
catch (Exception ex)  
{  
    if (dbCore != null )  
    {  
        if (dbCore . IsTransaction) dbCore. RollbackTransaction();  
        dbCore. Close();
```

```

    }
    throw ex;
}

```

## 2.2.5 Extend 辅助扩展功能

许多工具都提供例外辅助的功能，该工具也不例外，简要介绍如下：

### 1、TableHelp 辅助扩展

以 Oracle 自带的 scott 用户为例，先看如下代码：

```

DbCore dbCore = PublicClass . GetNewDbCore();
DataTable dt1 = dbCore . SelectAll() . From(EMPColumnTableName). ExecuteDataSet() . Tables[0];
DataTable dt2 = dbCore . SelectAll() . From(DEPTColumnTableName). ExecuteDataSet() . Tables[0];
DataTable dt3 = TableHelp . MergeTable(dt1, dt2, "DEPTNO"); // 按部门编号 DEPTNO 列将表 dt2 的
数据合并到 dt1
DataTable dt3_ = TableHelp . MergeTable(dt2, dt1, "DEPTNO"); // 按部门编号 DEPTNO 列将表 dt1 的
数据合并到 dt2, dt1 中有多行数据对应，取首行的数据，没有对应的数据为空
DataTable dt4 = TableHelp . AddTableRowNumCol(dt3); // 给 dt3 添加行号
DataTable dt5 = TableHelp . GetTableTopRows(dt4, 5); // 获取前 5 行
DataTable dt6 = TableHelp . GetTableSubRows(dt4, 6, 10); // 获取 dt4 从第 6 行到第 10 行
DataTable dt7 = TableHelp . GetTableSubRows(dt4, 11, 20); // 获取 dt4 从第 11 行到第 20 行, 注：
无 20 行取到最后一行
DataTable dt8 = TableHelp . GetTableBottomRows(dt4, 5); // 获取 dt4 后 5 行
dt8 = TableHelp . GetTableBottomRows(dt4, 50); // 获取 dt4 后 50 行；dt4 没有后 50 行, 从后面往前
取到最前面存在行
DataTable dt9 = TableHelp . JoinInner(dt1, dt2, "DEPTNO"); // 内连接
DataTable dt10 = TableHelp . JoinInner(dt1, dt2, "deptno" ); // 内连接
DataTable dt11 = TableHelp . JoinLeft(dt1, dt2, "DEPTNO"); // 左外连接
DataTable dt12 = TableHelp . JoinRight(dt1, dt2, "DEPTNO"); // 右外连接
DataTable dt13 = TableHelp . JoinLeft(dt2, dt1, "DEPTNO"); // 左外连接
DataTable dt14 = TableHelp . JoinFull(dt1, dt2, "DEPTNO"); // 完全外连接

DataTable dt15 = TableHelp . SortTable(dt1, "deptno", SortDirection . Asc);
DataTable dt16 = TableHelp . SortTable(dt1, "deptno" );
DataTable dt17 = TableHelp . SortTable(dt1, "deptno", SortDirection . Asc, "sal", SortDirection
. Asc);
DataTable dt18 = TableHelp . SortTable(dt1, "deptno", "sal" );
DataTable dt19 = TableHelp . SortTable(dt1, "deptno", SortDirection . Desc, "sal", SortDirec
tion . Desc);
DataTable dt20 = TableHelp . SortTableDesc(dt1, "deptno", "sal" );
DataTable dt21 = TableHelp . SortTable(dt1, "deptno", SortDirection . Asc, "sal", SortDirec
tion . Desc);

TableHelp . DataTableToExcel(dt1, @"C:/Documents and Settings/ 楚涛/桌面/temp1.xls" );

```

```

DataSet dst = new DataSet ();
DataTable dt22 = dt1 . Copy();
// 修改表名 ,DataTable 默认 TableName="Table",DataSet 集合的 DataTable.TableName 不能同名
dt22 . TableName = "EMP";
DataTable dt23 = dt2 . Copy();
// 修改表名 ,DataTable 默认 TableName="Table",DataSet 集合的 DataTable.TableName 不能同名
dt23 . TableName = "DEPT";
dst . Tables . Add(dt22);
dst . Tables . Add(dt23);

TableHelp . DataSetToExcel(dst, @"C:/Documents and Settings/ 楚涛/桌面/temp2.xls" );

```

TableHelp 辅助扩展类，提供合并 DataTable 数据表、获取数据表指定行的数据、内连接、左外连接、右外连接、完全外连接、对数据表排序、将数据表 DataTable 或数据集 DataSet 输出到 Excel 文件等，可见上述程序代码的释义说明文字。同时也提供通过过滤条件选择 DataTable 行、合并数据表行信息、转换数据表列值对并以 DataTable 的形式返回的常用方法，如下所示：

```

DataTable dt24 = TableHelp . GetTableSelect(dt1, "deptno=10" ); // 选取 deptno=10 的所有信息,并以 DataTable 的形式返回
DataTable dt25 = TableHelp . GetTableSelect(dt1, "deptno=20" ); // 选取 deptno=20 的所有信息,并以 DataTable 的形式返回
DataTable dt26 = TableHelp . TableAppend(dt24, dt25); // 将 dt23 数据按行附加到 dt22,并以新的结果数据表的形式返回
string [,] strArray = new string [1, 2];
strArray[0, 0] = "SCOTT";
strArray[0, 1] = "scott/tiger" ;
DataTable dt27 = TableHelp . ReplacleTableColValue(dt26, "ename", strArray);

```

MergeTable 和 TableAppend 区别是，前者是横向合并，即列的合并；后者是纵向合并，即行的合并。 ReplacleTableColValue 功能既是如字面意思所示替换 Table 列值，没有匹配的保留原值，功能类似 Oracle 的 decode 函数。

## 2、CommonHel常用方法扩展

CommonHel提供了许多常用方法，如财务上人民币金额大写的转换、字符串中中文的检查、唯一随机数字固定长度为 20 的数字字符串的产生、HTML代码和对应格式化的字符串的相互转化、整型和浮点型数字字符串检查、Email 地址和日期时间格式字符串的检查、获得中文字符串的汉语拼音码首字母等常用功能，可见以下示例：

```

string str1 = CommonHelpNumberToRMB(1); // " 壹元整 "
str1 = CommonHelpNumberToRMB(102); // " 壹佰零贰元整 "
str1 = CommonHelpNumberToRMB(1000234); // " 壹佰万零贰佰叁拾肆元整 "
str1 = CommonHelpNumberToRMB(1000023456); // " 壹拾亿零贰万叁仟肆佰伍拾陆元整 "
"

```

```

str1 = CommonHelpNumberToRMB(100000234567); // 壹仟亿零贰拾叁万肆仟伍佰陆拾柒元
整 "
decimal dec = 1234007890123.45M;
str1 = CommonHelpNumberToRMB(dec); // 壹万贰仟叁佰肆拾亿零柒
佰捌拾玖万零壹佰贰拾叁元肆角伍分 "
string str = string . Empty;
for (int i = 0; i < 1000; i++)
{
    str += CommonHelpGetOnlyID() + "/r/n" ;// 唯一随机数字固定长度为 20 的数字字符串
}
MessageBox Show(str);
string str1 = "abcdEFGH";
bool isHasChinese = CommonHelpIsHasChineseWord(str1); //false 不含有中文字符
str1 = "abcd 啊 EFGH";
isHasChinese = CommonHelpIsHasChineseWord(str1); //true 含有中文字符
string str2 = "<input type=/'button/' value=/'button/'><input type=/'image/' >";
string str3 = CommonHelpHTML_CodeToString(str2); //Html 代码和对应格式化的字符串的相互
转化
string str4 = CommonHelpStringToHTML_Code(str2); //Html 代码和对应格式化的字符串的相互
转化
string str5 = CommonHelpHTML_CodeToString(str3); //Html 代码和对应格式化的字符串的相互
转化

```

还有其他常用方法和加密 / 解密常用方法扩展类 CryptographyHelp 、 OfficeHelp 常用方法辅助扩展类等就不在此一一列举了。

## 2.2.6 ORM的分析及与 Xml的交互

### 1、ORM的分析

ORM通过对开发人员隐藏 SQL细节可以大大提高生产力。然而很不幸，“ORM和“性能问题”常常一起出现，它容易产生一些未被发现的荒谬查询，虽然几率不是很高，但万一出现，如果没有好的分析方案，也是极为头疼的事。通常情况下，数据库管理员可以通过如交叉引用有问题的存储过程或其它途径来查找问题代码。但是，ORM依赖于动态生成的 SQL，便很难这么做了。所以，需要一些有效的 ORM分析方法。该工具组件 System.Database.DbCore 提供了 DbCore.GetCurrentCommandText等静态方法可以在任何执行 ORM操作时分析当前执行的 SQL语句，这样在任何时候当出现异常时可查看当前执行的 SQL，如果 SQL正常可排除不是 ORM的问题，否则说明所编写的代码结构中出现了问题。

### 2、与 XML的交互

工具生成的实体提供 ToXml和 FromXml两个方法及相应的重载和补充方法，可以方便实体对象与 XML内容直接相互转换。

示例代码如下所示：

```
TSENTITY entity = new TSENTITY(1); // 得到主键为 1 的实体对象信息
TSENTITY entity_ = new TSENTITY(); // 仅实例化实体对象
string sTst = entity . ToXml();
entity . ToXml(@"C:/tst.xml" , Encoding . UTF8, Formatting . Indented);
entity . ToXml_(@"C:/tst1.xml" , Encoding . UTF8, Formatting . Indented);
entity_ = entity_ . FromXml(sTst);

entity_ = new TSENTITY();
entity_ = entity_ . FromXmlFile( @"C:/tst.xml" );
entity_ = new TSENTITY();
entity_ = entity_ . FromXmlFile( @"C:/tst1.xml" );
```

同样实体集也提供 ToXml和 FromXml及相应的重载和补充方法，方便实体集对象与 XML内容直接相互转换。

示例代码如下所示：

```
employeeS entitys1 = new employeeS( true ); // 获取员工对象的所有信息到实体集对象
employeeS entitys2 = new employeeS(); // 仅实例化实体集对象
employeeS entitys3 = new employeeS();
employeeS entitys4 = new employeeS();

string strXml = entitys1 . ToXml();
entitys2 = entitys2 . FromXml(strXml); // 从 Xml 内容中加载对象

strXml = entitys1 . ToXml(Formatting . Indented);
entitys3 = entitys3 . FromXml(strXml); // 从 Xml 内容中加载对象

string strFile = "temp.xml" ;
entitys1 . ToXml_(strFile, Encoding . UTF8, Formatting . Indented);
entitys4 = entitys4 . FromXmlFile(strFile); // 从 Xml文件中加载对象
```

### 3 结束语

以上就是 VB/C#.Net 实体代码生成工具 (EntitysCodeGenerate) 对 ORM框架的实现及使用过程，和对实体 (集) 代码的批量生成及各种数据库访问的统一操作的简单介绍，详细可见示例代码及工具帮助文档。另外，使用该工具，当项目需要切换数据库或使用不同的数据库时，只须适当修改数据库连接类型及连接字符串即可。工具安装后附带大量的示例代码，里面有更多的示例和 chm格式的帮助用户文档。

使用工具在建表之前建议：数据库表名及字段名应当遵循通用规范，如都使用英文字母，字母大写，且首字符以大写英文字母开头且不含有特殊符号，如 ' ' 字符等。另外文字命名可约定俗成为单数形式，可以是英文字母及数字和下划线(\_)组合，此外建议表命名以 "T\_" 开头，表字段以 "C\_" 开头，视图以 "V\_" 开头等，这样做的目的可使数据库设计更规范化，也避免了与关键字的冲突，同时也是因为许多数据库都是英文字母开头的且不能含有特殊字符，如 Oracle 就是这样，且表名称和字段名都是大写（对 PostgreSQL 建议采用小写命名），长度不超过 30 个字符（命名过长无意义，有些数据库也不支持），多出的信息可以在命名注释里多写些注释信息。若采用 Oledb、Odbc 方式连接，表字段名最好不要这样 FIELD1,2,3... 以数字顺序结尾，因为这样容易与参数机制冲突，导致数据访问时带来一些参数上不必要的麻烦，且这样命名也不是好的规范，必要时可以是 FIELD A,B,C...。每个表建议都设置主键，可以是联合主键。若没有主键，则在使用主键判断记录的唯一性时须指定字段。

当前网上也有许多类似的 ORM 工具，但总有许多限制，如：对国产数据库及自定义数据库的不支持、无主键或联合主键的不支持、大量复杂的配置、属性字段注释提取的缺少、数据类型之间的映射死板、不稳定、生成代码的不规范、代码修改困难、复杂数据访问及性能问题、内部配置文件复杂或报错、工具兼容性差、缺少 ORM 分析或其它使用及生产环境运行问题等等。该工具很好的解决了这些问题，并结合多年具体项目的需求，可以像抽屉一样单独或分开使用，也可和其它组件并行使用，如可以与其它组件或平台的集成，动态得从相关平台或组件中读取数据库参数而不必写死在文件里，这样若要实现对用户数据库连接信息加密/解密也可以很好的处理。

理论的实现总是从简单到复杂，覆盖所有可能，实际应用就需要结合实际从复杂到简单，凡是要灵活变通使用，化复杂为简单，将复杂的东西以简单的方式解决为好。就如模型思想提出时需要从众多实践中总结出来，再升华并得到全面的验证，而模型应用则要结合具体实际，拿来适当的从简应用，回归到实践。

软件开发都希望从大量重复繁重的代码中解放出来，缩短工期并提高项目质量，同时当需要的时候可以准确的将绝大多数后台代码快速完成，并要便于后期维护，这些就需要适当借助工具和方法为开发、维护和交流提供良好的保障，显然该工具提供了最佳选择。

由于时间仓促，加之考虑欠缺及水平所限，不足之处在所难免，有待进一步完善，敬请来信交流 ([lxchutao@163.com](mailto:lxchutao@163.com))、批评斧正！Thanks!