Entity Framework 学习初级篇 1--EF 基本概况

最近在学习研究微软的 EF，通过这时间的学习研究，感觉这个 EF 目前来说还不是很完善，半成品。不过，据说在.Net4.0 中，微软将推荐使用此框架，并会有所改善。而且，现在基本上所有数据库均提供了对 EF 的支持。因此，为以后做技术准备可以学习研究以下。但是，我个人觉得就目前来说，在实际项目慎用此框架。

下面简单的介绍以下这个 EF。

在.Net Framework SP1 微软包含一个实体框架（Entity Framework），此框架可以理解成微软的一个 ORM 产品。用于支持开发人员通过对概念性应用程序模型编程（而不是直接对关系存储架构编程）来创建数据访问应用程序。目标是降低面向数据的应用程序所需的代码量并减轻维护工作。Entity Framework 应用程序有以下优点：

- 应用程序可以通过更加以应用程序为中心的概念性模型(包括具有继承性、复杂成员和关系的类型）来工作。
- 应用程序不再对特定的数据引擎或存储架构具有硬编码依赖性。
- 可以在不更改应用程序代码的情况下更改概念性模型与特定于存储的架构之间的映射。
- 开发人员可以使用可映射到各种存储架构(可能在不同的数据库管理系统中实现）的一致的应用程序对象模型。

- 多个概念性模型可以映射到同一个存储架构。
- 语言集成查询支持可为查询提供针对概念性模型的编译时语法验证。

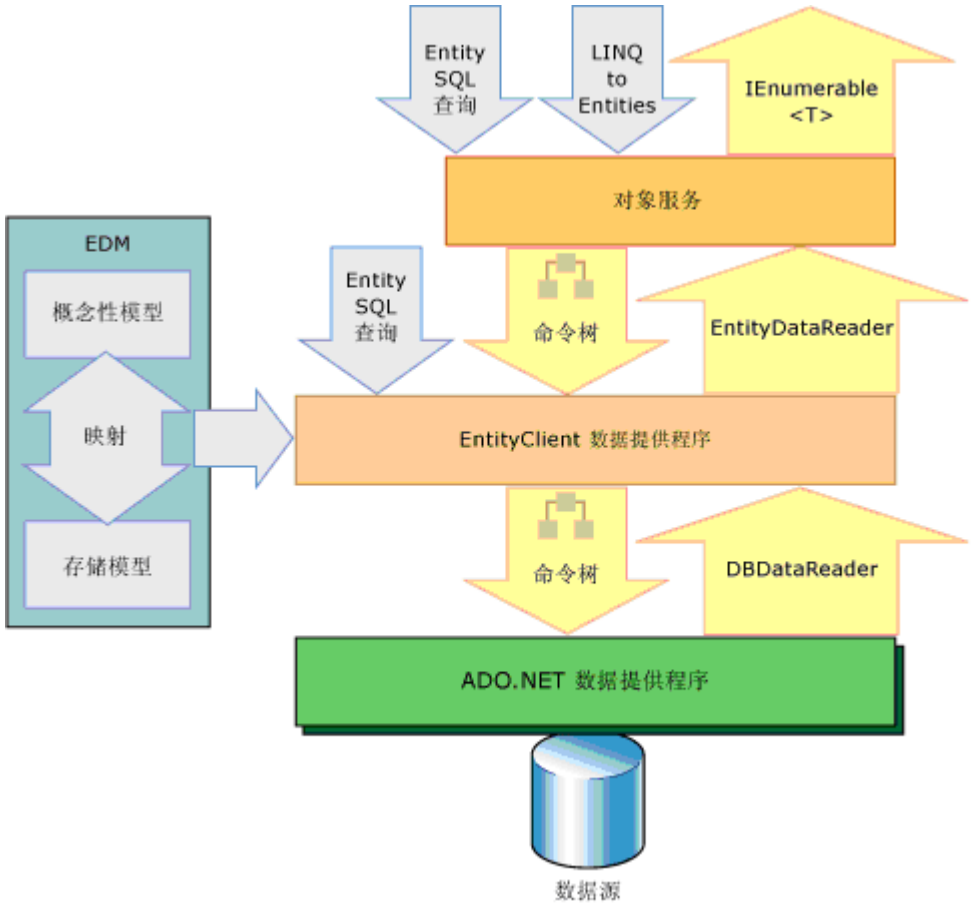
实体框架 Entity Framework 是 ADO.NET 中的一组支持开发面向数据的软件应用程序的技术。在 EF 中的实体数据模型（EDM）由以下三种模型和具有相应文件扩展名的映射文件进行定义。

- 概念架构定义语言文件（.csdl）-- 定义概念模型。
- 存储架构定义语言文件（.ssdl）-- 定义存储模型（又称逻辑模型）。
- 映射规范语言文件（.msl）-- 定义存储模型与概念模型之间的映射。

实体框架 使用这些基于 XML 的模型和映射文件将对概念模型中的实体和关系的创建、读取、更新和删除操作转换为数据源中的等效操作。EDM 甚至支持将概念模型中的实体映射到数据源中的存储过程。它提供以下方式用于查询 EDM 并返回对象：

- LINQ to Entities -- 提供语言集成查询（LINQ）支持用于查询在概念模型中定义的实体类型。
- Entity SQL -- 与存储无关的 SQL 方言，直接使用概念模型中的实体并支持诸如继承和关系等 EDM 功能。
- 查询生成器方法 --可以使用 LINQ 风格的查询方法构造 Entity SQL 查询。

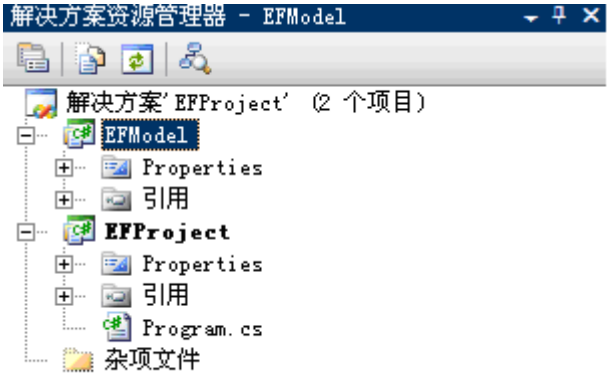
下图演示用于访问数据的实体框架体系结构：



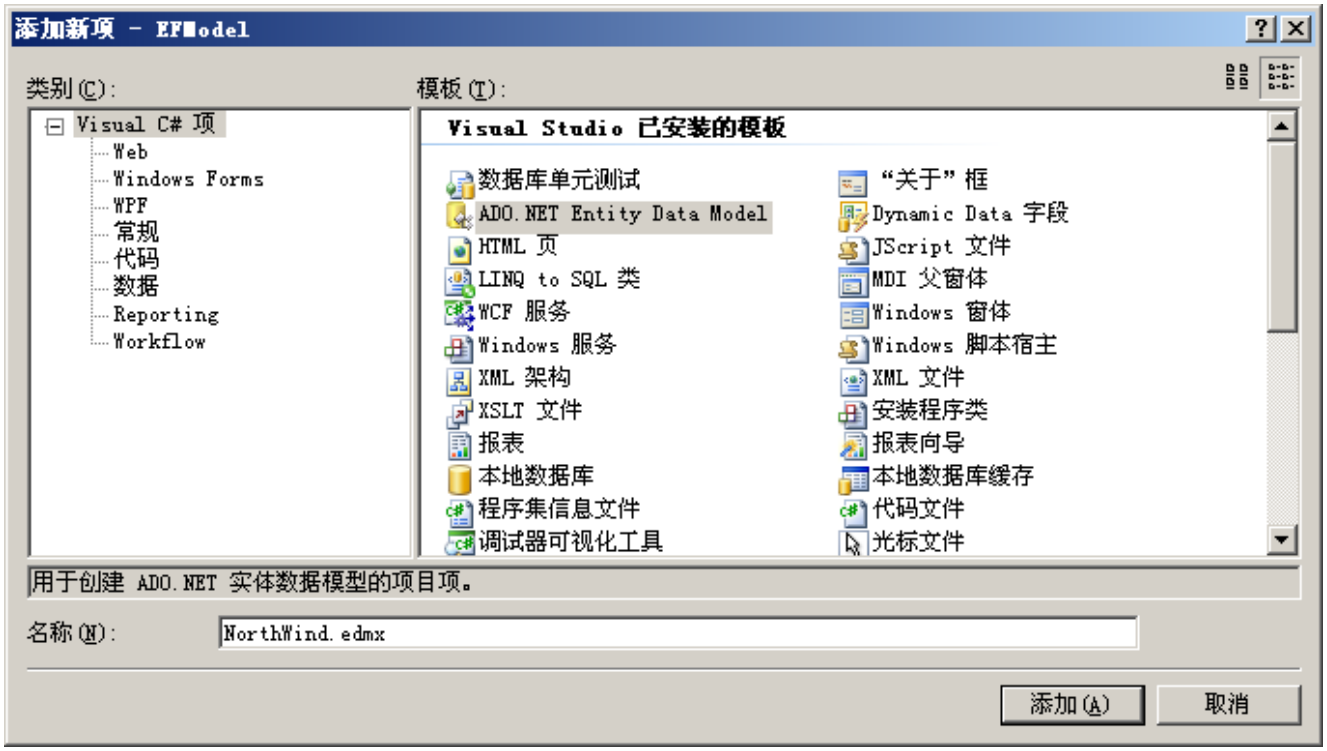
下面，来学习 EF 的基本使用方法。软件环境：：

- Visual Studio 2008 +SP1
- SQL Server2005/2008

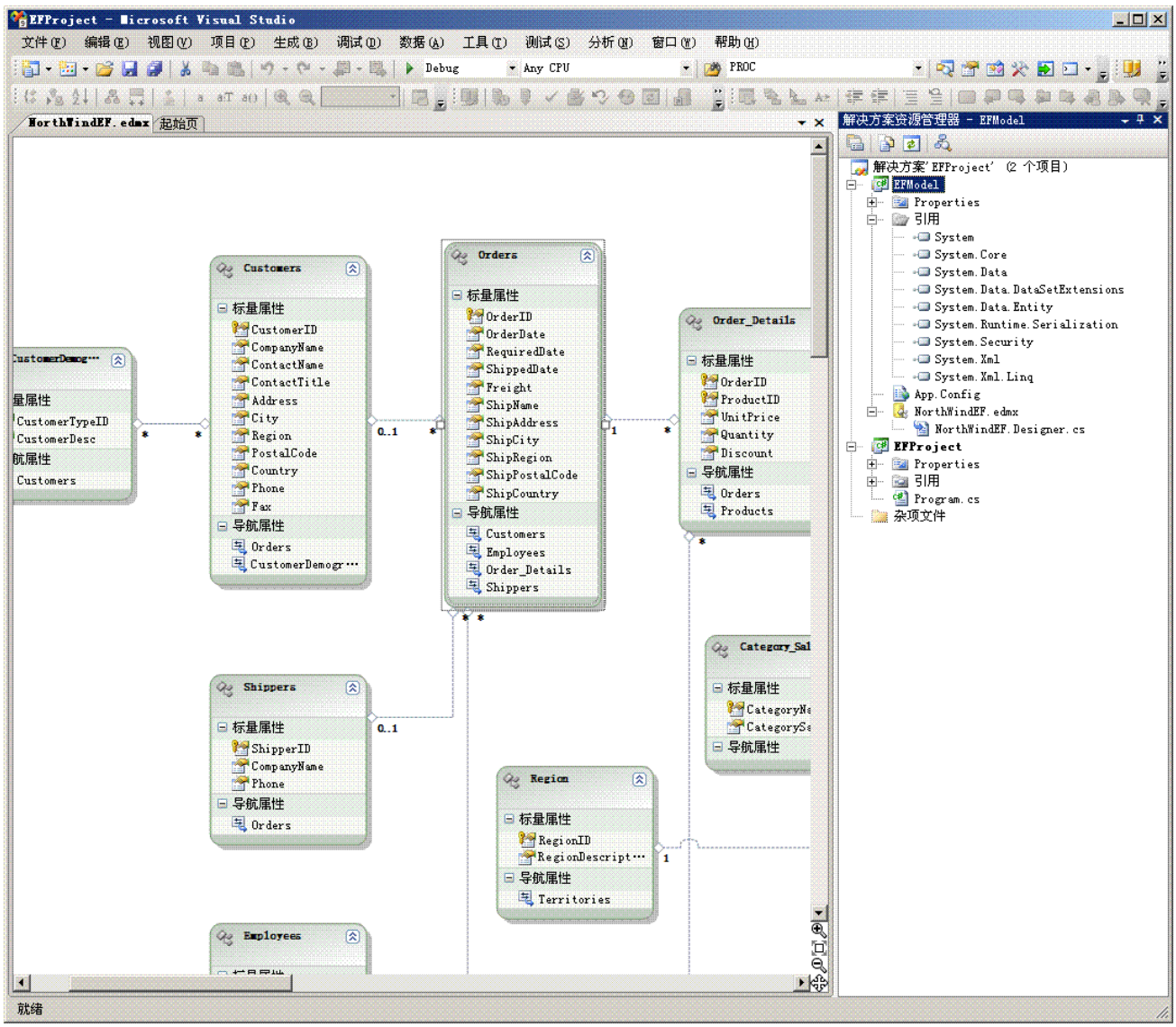
首先，建立一个名为“EFProject”的解决方案，然后添加一个名为“EFModel”的类库项目。如下图所示。



接着，在 EFModel 项目中，添加“ADO.NET Entity Data Model”项目，如下图所示：



名称取为“NorthWindEF.edmx”,然后点击“添加”。然后，在后面的步骤中，数据库选择“NorthWind”后，在选择影射对象是，把表、试图、存储过程全部都选上，其他的均保存默认的即可。最终生成的结果如下图所示。



好了，数据模型生成完毕。

最后，谈谈我认为的一些缺点：

- Edmx 包含了所有对象的 csdl，ssdl，msl 文件，过于庞大，如果要手动修改这个文件，一不小心，眼睛看花了，就改错了。（和数据集一样的毛病）。

- 目前 EF 支持表、试图、存储过程，其他的对象不支持，而且对使用存储过程有很多限制（目前有 EFExtension 提供了更多对象的支持）。
- 除了 MS SQL Server 可直接提供这种可视化的设计界面外，其他的数据库目前还没有提供可视化设计界面（但可以自己来实现，后面介绍）。
- 性能问题。（网上看到有说比 ADO.Net 慢 700 百，又有人说比 ADO.net 快的，具体情况我还没测试过，但我觉得像这个些类型的框架，性能肯定是比上原生态的 ADO.net 慢）

好了，接下来，学习以下简单的各种操作。

Entity Framework 学习初级篇 2--ObjectContext、ObjectQuery、ObjectStateEntry、ObjectStateManager 类的介绍

本节，简单的介绍 EF 中的 ObjectContext、ObjectQuery、ObjectStateEntry、ObjectStateManager 这个几个比较重要的类，它们都位于 System.Data.Entity.dll 下的 System.Data.Objects 命名空间下。在后续的章节中，我们经常会用到它们的某些方法，以便完成我们的某些操作或目的。本节，简单的说明一下以后我们可能会用到的各个类的方法,以方便我们后续的学习。

ObjectContext 封装 .NET Framework 和数据库之间的连接。此类用作“创建”、“读取”、“更新”和“删除”操作的网关。

ObjectContext 类为主类，用于与作为对象（这些对象为 EDM 中定义的实体类型的实例）的数据进行交互。

ObjectContext 类的实例封装以下内容：

- 到数据库的连接，以 EntityConnection 对象的形式封装。

- 描述该模型的元数据，以 MetadataWorkspace 对象的形式封装。
- 用于管理缓存中持久保存的对象的 ObjectStateManager 对象。

ObjectContext 类的成员方法以说明如下所示：

- AcceptAllChanges()

接受所有对该实体对象的更改

- AddObject(string,object)

将实体对象添加到制定的实体容器中

- ApplyPropertyChanges(string,object)

将以指派的实体对象属性的更改应用到容器中对应的原对象。

- Attach(System.Data.Objects.DataClasses.IEntityWithKey entity)

将带主键的实体对象附加到默认的容器中

- Attach(string,object)

将实体对象附加到指定的实体容器中

- CreateEntityKey(string,object)

给指定的实体对象创建实体主键或如果已存在实体主键，则直接返回该实体的主键

- CreateQuery<T>(string,params ObjectParameter[])

从给定的查询字符串创建 ObjectQuery 对象。

- DeleteObject(object)

删除指定的实体对象

- Detach(object)

移除指定的实体对象

- ExecuteFunction<TElement>(string,params ObjectParameter[])

对默认容器执行给定的函数。

- GetObjectByKey(System.Data.EntityKey key)

通过主键 KEY 从 ObjectStateManager 中检索对象（如果存在）；否则从存储区中检索。

- Refresh(System.Data.Objects.RefreshMode refreshMode, object entity)

按指定持久更新模式，使用指定实体的存储区数据更新 ObjectStateManager。。

- Refresh(System.Data.Objects.RefreshMode refreshMode, System.Collections.IEnumerable collection)

按指定持久处理模式，使用指定实体集的存储区数据更新 ObjectStateManager。

- SaveChanges(bool)

将所有更新持久保存到存储区中。参数是客户端事务支持所需的参数。参数为 true 则在更新后自动将更改应用到 ObjectStateManager 中的实体。如果为 false，则在更新后还需要调用 AcceptAllChanges（）以便更新 ObjectStateManager 中的实体。

- SaveChanges()

将所有更新持久保存到存储区中

- TryGetObjectByKey(System.Data.EntityKey,out object)

尝试从指定实体主键返回该实体

以上各个方法的具体用法，将在后面介绍。

接着，再看看有用的类 ObjectQuery。

ObjectQuery 有个有用的方法 ToTraceString(),这个方法用于追踪所执行的 SQL 语句，通过此方法我们可以获取所执行的 SQL 语句，以便我们查看、分析具体执行的 SQL 语句。（类似 Nhibernate 配置文件中的 showsql 节）

再了解一下 ObjectStateEntry。

ObjectStateEntry 维护实体实例或关系实例的状态（已添加、已删除、已分离、已修改或未更改）、键值和原始值。还管理已修改属性的列表。其包含一下方法：

- AcceptChanges

接受当前值作为原始值，并将实体标记为 Unchanged()。

- Delete

将实体标记为 Deleted()。如果实体处于 Added()()() 状态，它将为 Detached()。

- GetModifiedProperties

返回标记为 Modified()的属性名称。

- SetModified

将状态设置为 Modified()。

- SetModifiedProperty

将指定的属性标记为 Modified()。

接着，再看看 ObjectStateManager。

ObjectStateManager 用于维护对象映射、对象状态/标识管理以及实体实例或关系实例的持久性。

- GetObjectStateEntries

获取给定 EntityState 的 ObjectStateEntry 集合。

- GetObjectStateEntry

获取给定的 EntityKey 对应的 ObjectStateEntry

现在，几个重要的类简单介绍完毕。后面，我们将具体学习它们的使用。

Entity Framework 学习初级篇 3-- LINQ TO Entities

LINQ 技术（即 LINQ to Entities）使开发人员能够通过使用 LINQ 表达式和 LINQ 标准查询运算符，直接从开发环境中针对 实体框架对象上下文创建灵活的强类型查询。LINQ to Entities 查询使用对象服务基础结构。ObjectContext 类是作为 CLR 对象与实体数据模型 进行交互的主要类。开发人员通过 ObjectContext 构造泛型 ObjectQuery 实例。ObjectQuery 泛型类表示一个查询，此查询返回一个由类型化实体组成的实例或集合。返回的实体对象可供更新并位于对象上下文中。以下是创建和执行 LINQ to Entities 查询的过程：

1. 从 ObjectContext 构造 ObjectQuery 实例。
2. 通过使用 ObjectQuery 实例在 C# 或 Visual Basic 中编写 LINQ to Entities 查询。

3. 将 LINQ 标准查询运算符和表达式将转换为命令目录树。
4. 对数据源执行命令目录树表示形式的查询。执行过程中在数据源上引发的任何异常都将直接向上传递到客户端。
5. 将查询结果返回到客户端。

一、Linq To Entities 简单查询

下面将介绍简单的 Linq To Entities 查询，相关的查询语法可以使用基于表达式或基于方法的语法。本节使用的 TestDriver.Net 配合 Nunit2.4 进行测试。

1， 投影

代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data.Objects;
using NUnit.Framework;
namespace NorthWindModel
{
    [TestFixture]
    public class TestEFModel
    {
        [Test]
        public void Select()
        {
            using (var edm = new NorthwindEntities())
            {
                //基于表达式的查询语法
                ObjectQuery<Customers> customers = edm.Customers;
```

```
                    IQueryable<Customers> cust1 = from c in customers
                                                  select c;
                    Assert.Greater(cust1.Count(), 0);
                    //使用 ObjectQuery 类的 ToTraceString()方法显示查询
SQL 语句
                    Console.WriteLine(customers.ToTraceString());



                }
            }
        }
    }
```

输出：

```
SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax]
```

```
FROM [dbo].[Customers] AS [Extent1]
```

```
1 passed, 0 failed, 0 skipped, took 11.00 seconds (NUnit 2.4).
```

在上面的输出内容中，可以看到使用了 ToTraceString()方法来输出具体的 SQL 语句。同时 Nunit 也输出相关的测试情况，请注意查询所花费的时间，以便我们进行查询速度的分析比较。

2， 条件限制

```
using (var edm = new NorthwindEntities())
{
    //基于表达式的查询语法
    ObjectQuery<Customers> customers = edm.Customers;
    IQueryable<Customers> cust1 = from c in customers
                                  where c.CustomerID == "ALFKI"
                                  select c;

    Assert.AreEqual(cust1.Count(), 1);
    foreach (var c in cust1)
        Console.WriteLine("CustomerID={0}", c.CustomerID);

    //基于方法的查询语法
    var cust2 = edm.Customers.Where(c => c.CustomerID == "ALFKI");
    Assert.AreEqual(cust2.Count(), 1);
    foreach (var c in cust2)
        Console.WriteLine("CustomerID={0}", c.CustomerID);
```

```
}
```

3， 排序和分页

在使用 Skip 和 Take 方法实现分页时，必须先对数据进行排序，否则将会抛异常。

```
using (var edm = new NorthwindEntities())
 {
        //基于表达式的查询语法
     ObjectQuery<Customers> customers = edm.Customers;
     IQueryable<Customers> cust10 = (from c in customers
                                     orderby c.CustomerID
                                     select c).Skip(0).Take(10);

      Assert.AreEqual(cust10.Count(), 10);
      foreach (var c in cust10)
            Console.WriteLine("CustomerID={0}", c.CustomerID);

    //基于方法的查询语法
    var cust = edm.Customers.OrderBy(c => c.CustomerID).Skip(0).Take(10);
    Assert.AreEqual(cust.Count(), 10);
    foreach (var c in cust)
        Console.WriteLine("CustomerID={0}", c.CustomerID);
 }
```

4， 聚合

可使用的聚合运算符有 Average、Count、Max、Min 和 Sum。

```
using (var edm = new NorthwindEntities())
```

```
{
    var maxuprice = edm.Products.Max(p => p.UnitPrice);
    Console.WriteLine(maxuprice.Value);
     }
```

5， 连接

可以的连接有 Join 和 GroupJoin 方法。GroupJoin 组联接等效于左外部联接，它返回第一个（左侧）数据源的每个元素（即使其他数据源中没有关联元素）。

```
using (var edm = new NorthwindEntities())
 {
    var query = from d in edm.Order_Details
                join order in edm.Orders
                on d.OrderID equals order.OrderID
                select new
                 {
                     OrderId = order.OrderID,
                     ProductId = d.ProductID,
                     UnitPrice = d.UnitPrice
                 };
     foreach (var q in query)

    Console.WriteLine("{0},{1},{2}",q.OrderId,q.ProductId,q.UnitPrice);
    }
```

其他一些方法等就不多说了，和 Linq to SQL 基本上是一样的。

二、LINQ to Entities 查询注意事项

- 排序信息丢失

如果在排序操作之后执行了任何其他操作，则不能保证这些附加操作中会保留排序结果。这些操作包括 Select 和 Where 等。另外，采用表达式作为输入参数的 First 和 FirstOrDefault 方法不保留顺序。

如下代码：并不能达到反序排序的效果

```
using (var edm = new NorthwindEntities())
{
    IQueryable<Customers> cc = edm.Customers.OrderByDescending(c => c.CustomerID).Where(c => c.Region != null).Select(c => c);
    foreach (var c in cc)
        Console.WriteLine(c.CustomerID);
}
```

- 不支持无符号整数

由于 实体框架不支持无符号整数，因此不支持在 LINQ to Entities 查询中指定无符号整数类型。如果指定无符号整数，则在查询表达式转换过程中会引发 NotSupportedException 异常,并显示无法创建类型为“结束类型”的常量值。此上下文仅支持基元类型(“例如 Int32、String 和 Guid”)。

如下将会报异常的代码:

```
using (var edm = new NorthwindEntities())
{
    uint id = UInt32.Parse("123");
    IQueryable<string> produt = from p in edm.Products
                                where p.UnitPrice == id
                                select p.ProductName;
    foreach (string name in produt)
        Console.WriteLine(name);
}
```

上面的代码中，由于 id 是 uint 而不是 Int32，String，Guid 的标量类型，所以在执行到 where p.UnitPrice ==id 这个地方时，会报异常。

- 不支持引用非标量闭包

不支持在查询中引用非标量闭包（如实体）。在执行这类查询时，会引发 NotSupportedException 异常，并显示消息“无法创建类型为“结束类型”的常量值。此上下文中仅支持基元类型(‘如 Int32、String 和 Guid’)

如下将会报异常的代码:

```
using (var edm = new NorthwindEntities())
 {
        Customers customer = edm.Customers.FirstOrDefault();
        IQueryable<string> cc = from c in edm.Customers
                                where c == customer
                                select c.ContactName;
         foreach (string name in cc)
              Console.WriteLine(name);
 }
```

上面的代码中，由于 customer 是引用类型而不是 Int32，String，Guid 的标量类型，所以在执行到 where c==customer 这个地方时，会报异常。

好，本节介绍完毕。后面将继续学习 EF.

Entity Framework 学习初级篇 4--Entity SQL

Entity SQL 是 ADO.NET 实体框架 提供的 SQL 类语言，用于支持 实体数据模型(EDM)。Entity SQL 可用于对象查询和使用 EntityClient 提供程序执行的查询。

- 关键字

Value 关键字

ESQL 提供了 SELECT VALUE 子句以跳过隐式行构造。SELECT VALUE 子句中只能指定一项。在使用这样的子句时，将不会对 SELECT 子句中的项构造行包装器，并且可

生成所要形状的集合，例如：SELECT VALUE it FROM NorthwindEntities.Customers as it

it 关键字

it 出现在 ESQL 中，查询对象的别名默认值 "it" 改成其他字符串，例如：

"SELECT VALUE it FROM NorthwindEntities.Customers as it " 。

- 注释：

Entity SQL 查询可以包含注释。注释行以两个短划线 (--) 开头。

"SELECT VALUE it FROM NorthwindEntities.Customers as it -- this a comment "

- Select 查询

例如：

SELECT VALUE it FROM NorthwindEntities.Customers as it

- 参数

参数是在 esql 之外定义的变量，每个参数都有名称和类型，参数名称在查询表达式中定义，并以@符号作为前缀。例如：

Select VALUE c from NorthwindEntities.Customers as c where c.CustomerID=@customerID

- 聚合

Enity SQL 不支持 * ，所以 esql 不支持 count(*)，而是使用 count(0)，例如：

Select count(0) from NorthwindEntities.Customers

- 分页 SKIP/LIMIT

可以通过在 ORDER BY 子句中使用 SKIP 和 LIMIT 子子句执行物理分页。若要以确定的方式执行物理分页，应使用 SKIP 和 LIMIT。如果您只是希望以非确定的方式限制结果中的行数，则应使用 TOP。TOP 和 SKIP/LIMIT 是互斥的

使用 SKIP/LIMIT 分页，esql 代码如下：

```
Select value c from NorthwindEntities.Customers as c order by c.CustomerID skip 0 limit 10
```

- TOP

SELECT 子句可以在可选的 ALL/DISTINCT 修饰符之后具有可选的 TOP 子子句。TOP 子子句指定查询结果中将只返回第一组行。esql 代码如下：

```
Select top(10) c.CustomerID from NorthwindEntities.Customers as c order by c.CustomerID
```

- NULL 处理

Null 文本与 Entity SQL 类型系统中的任何类型都兼容，可以使用 cast 进行类型转换，例如：

```
select cast(c.region as string) from NorthwindEntities.Customers as c order by c.CustomerID limit 10
```

其中， Nvarchar 等可以成 string，数字类型可以转成 int32，其他的类型转换类似。如果无法完成转换，则将报异常。还有可以处理的方法有 treat。

- 标识符

Entity SQL 提供两种标识符：简单标识符和带引号的标识符

简单标识符：Entity SQL 中的简单标识符是字母数字和下划线字符的序列。标识符的第一个字符必须是字母字符（a-z 或 A-Z）。

带引号的标识符：带引号的标识符是括在方括号（[]）中的任何字符序列。带中文的部分，请使用方括号包括起来，否则会报如下异常信息：“简单标识符“中文”只能包含基本拉丁字符。若要使用 UNICODE 字符，请使用转义标识符”

正确的代码如下：

```
Select c.CustomerID as [中文字符] from NorthwindEntities.Customers as c order by c.CustomerID skip 0 limit 10
```

- ROW

Esql 可使用 row 来构建匿名的结构类型的纪录。例如：

SELECT VALUE row(p.ProductID as ProductID,p.ProductName as ProductName) FROM NorthwindEntities.Products as p order by p.ProductID LIMIT 10

- Key

提取引用或实体表达式的键。如下 esql 语句，直接返回 Customer 表的主键：

string esql = "SELECT value key(c) FROM NorthwindEntities.Customers as c order by c.CustomerID LIMIT 10"

- CreateRef/ref/deref

CreateRef 创建对实体集中的实体的引用。

ref 返回对实体实例的引用，之后就可以当作实体来访问其属性，esql 语句如下：

SELECT ref(c).CustomerID FROM NorthwindEntities.Customers as c order by c.CustomerID LIMIT 10

deref 运算符取消引用一个引用值，并生成该取消引用的结果。

- CASE 语句：

string esql = "using SqlServer;select case when len(trim(c.CustomerID))==0 then true else false end from NorthwindEntities.Customers as c order by c.CustomerID limit 10";

- 运算符

Esql 支持的运算符有：加+、减-、乘*、除/、取模%、-负号。Esql 语句如下：

select 100/2 as OP from NorthwindEntities.Customers as c order by c.CustomerID limit 10

- 比较运算符

Esql 支持的比较运算符有:=,>,>=,IS [NOT] NULL,<,[NOT] BETWEEN,!=,<>,[NOT] LIKE。Esql 语句如下：

select value p from NorthwindEntities.Products as p where p.UnitPrice > 20 order by p.ProductID limit 10

- 逻辑运算符

Esql 支持的逻辑运算符有：and(&&),not(!),or(||)。Esql 语句如下：

```
select value p from NorthwindEntities.Products as p where p.UnitPrice > 20 and p.UnitPrice<100 order by p.ProductID limit 10
```

或

```
select value p from NorthwindEntities.Products as p where p.UnitPrice > 20 && p.UnitPrice<100 order by p.ProductID limit 10
```

- 字符串连接运算符。

加号 (+) 是 Entity SQL 中可将字符串串联起来的唯一运算符。Esql 语句如下：

```
select c.CustomerID + c.ContactName from NorthwindEntities.Customers as c order by c.CustomerID limit 10
```

- 嵌套查询

在 Entity SQL 中，嵌套查询必须括在括号中，将不保留嵌套查询的顺序

```
select c1.CustomerID from( select value c from NorthwindEntities.Customers as c  order by c.CustomerID limit 10) as c1
```

- 日期时间函数

Esql 提供的日期时间函数有：CurrentDateTime()获取当前服务器的日期时间，还有 month,day，year，second, Minute ，Hour 等。例如:

```
select CurrentDateTime()  from NorthwindEntities.Customers as c  order by c.CustomerID limit 10
```

- 字符串函数

Esql 提供的字符串函数有：
Concat,IndexOf,Left,Length,Ltrim,Replace,Reverse,Rtrim,SubString,Trim,ToLower,ToUpper.例如：

```
select Reverse(p.ProductName) as ProductName from NorthwindEntities.Products as p  order by p.ProductID limit 10
```

- GUID

Esql 提供 newguid()函数，产生一个新的 Guid。例如：

```
select newguid()  from NorthwindEntities.Customers as c  order by c.CustomerID limit 10
```

- 数学函数：

Abs,Ceiling,Floor,Round

- 统计函数：

Avg，BigCount,Count,Max,Min,StDev,Sum

- 位计算函数

如果提供 Null 输入，则这些函数返回 Null。这些函数的返回类型与参数类型相同。如果函数采用多个参数，则这些参数必须具有相同的类型。若要对不同类型执行位运算，则需要显式强制转换为相同类型.

BitWiseAnd,BitWiseNot,BitWiseOr,BitWiseXor

- 命名空间

Entity SQL 引入命名空间以避免全局标识符（如类型名称、实体集、函数等）出现名称冲突。Entity SQL 中的命名空间支持与 .NET Framework 中的命名空间支持类似。

Entity SQL 提供两种形式的 USING 子句：限定命名空间（其中，提供较短的别名以表示命名空间）和非限定命名空间，如下例所示：

```
USING System.Data;

USING tsql = System.Data;
```

例如：

```
string esql = "using System; select cast(p.UnitPrice as Int32)  from NorthwindEntities.Products as p  order by p.ProductID limit 10 ";

string esql = "using System;using SqlServer; select (cast(p.UnitPrice as Int32)),SqlServer.ltrim(p.ProductName) as nameLen from NorthwindEntities.Products as p  order by p.ProductID limit 10 ";
```

最后，简单说一下 Esql 与 T-Sql 的某些差异：

- Entity SQL 中的所有列引用都必须用表别名限定.
- Esql 不支持 Any，all 限定运算符以及*运算
- Entity SQL 当前未提供对 DML 语句（insert、update、delete）的支持。
- Entity SQL 的当前版本未提供对 DDL 的支持。

Entity Framework 学习初级篇 5--ObjectQuery 查询及方法

ObjectQuery 类支持对 实体数据模型 (EDM) 执行 LINQ to Entities 和 Entity SQL 查询。ObjectQuery 还实现了一组查询生成器方法，这些方法可用于按顺序构造等效于 Entity SQL 的查询命令。下面是 ObjectQuery 的查询生成器方法以及等效的 Entity SQL 语句：

Distinct,Except,GroupBy,Intersect,OfType,OrderBy,Select,SelectValue,Skip,Top,Union,UnionAll,Where

每个查询生成器方法返回 ObjectQuery 的一个新实例。使用这些方法可以构造查询，而查询的结果集基于前面 ObjectQuery 实例序列的操作。下面来看具体的代码片断：

- Execute 方法:

```
using (var edm = new NorthwindEntities())
{
    string esql = "select value c from NorthwindEntities.Customers as c order by c.CustomerID limit 10";
    ObjectQuery<Customers> query = edm.CreateQuery<Customers>(esql);
    ObjectResult<Customers> results = query.Execute(MergeOption.NoTracking);
    Assert.AreEqual(results.Count(), 10);
    foreach (Customers c in query)
```

```
            Console.WriteLine(c.CustomerID);
        }
```

其中需要说明的是：MergeOption 这个枚举类型的参数项,MergeOption 有四种值分别是：

- AppendOnly： 只追加新实体，不修改以前获取的现有实体。这是默认行为。
- OverwriteChanges： 将 ObjectStateEntry 中的当前值替换为存储区中的值。这将使用服务器上的数据重写在本地所做的更改。
- PreserveChanges： 将替换原始值，而不修改当前值。这对于在发生开放式并发异常之后强制成功保存本地值非常有用。
- NoTracking： 将不修改 ObjectStateManager，不会获取与其他对象相关联的关系，可以改善性能。
- GetResultType 方法：返回查询结果的类型信息.例如：

```
            using (var edm = new NorthwindEntities())
        {
            string esql = "select value c from NorthwindEntities.Customers as c order by c.CustomerID limit 10";
            ObjectQuery<Customers> query = edm.CreateQuery<Customers>(esql);
            Console.WriteLine(query.GetResultType().ToString());
            //输出结果为：
            //NorthWindModel.Customers
        }
```

- ToTraceString 方法：获取当前执行的 SQL 语句。
- Where

实例代码如下：

```
using (var edm = new NorthwindEntities())
 {
```

```
                string esql = "select value c from NorthwindEntities.Customers
as c ";
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql);
                //使用 ObjectParameter 的写法
                query1 = query1.Where("it.CustomerId=@customerid");
                query1.Parameters.Add(new ObjectParameter("customerid",
"ALFKI"));
                //也可以这样写
                //ObjectQuery<Customers> query2 =
edm.Customers.Where("it.CustomerID='ALFKI'");
                foreach (var c in query1)
                    Console.WriteLine(c.CustomerID);
                //显示查询执行的 SQL 语句
                Console.WriteLine(query1.ToTraceString());

     }
```

- First/ FirstOrDefault

实例代码如下：

```
    using (var edm = new NorthwindEntities())
     {
                string esql = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
                ObjectQuery<Customers> query =
edm.CreateQuery<Customers>(esql);
                Customers c1 = query.First();
                Customers c2 = query.FirstOrDefault();
                Console.WriteLine(c1.CustomerID);
                Assert.IsNotNull(c2);
```

```
                Console.WriteLine(c2.CustomerID);
        }
```

- Distinct

实例代码如下：

```
    using (var edm = new NorthwindEntities())
    {
                string esql = "select value c.City from
NorthwindEntities.Customers as c order by c.CustomerID limit 10";
                ObjectQuery<string> query = edm.CreateQuery<string>(esql);
                query = query.Distinct();
                foreach (string c in query)
                {
                    Console.WriteLine("City {0}", c);
                }
     }
```

- Except:返回两个查询的差集。实例代码如下：

```
        using (var edm = new NorthwindEntities())
     {
                string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
                string esql2 = "select value c from NorthwindEntities.Customers
as c where c.Country='UK' order by c.CustomerID limit 10";
                ObjectQuery<Customers> query2 =
edm.CreateQuery<Customers>(esql2);
                query1 = query1.Except(query2);
                foreach (Customers c in query1)
                {
```

```
                Console.WriteLine(c.Country);
                //输出:UK
            }
        }
```

- Intersect: 返回两个查询的交集。实例代码如下：

```
using (var edm = new NorthwindEntities())
    {
            string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
            ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
            string esql2 = "select value c from NorthwindEntities.Customers
as c where c.Country='UK' order by c.CustomerID limit 10";
            ObjectQuery<Customers> query2 =
edm.CreateQuery<Customers>(esql2);
            query1 = query1.Intersect(query2);
            foreach (Customers c in query1)
            {
                Console.WriteLine(c.Country);
            }
     }
```

- Union/UnionAll:返回两个查询的合集，包括重复项。其中 UnionAll 必须是相同类型或者是可以相互转换的。

- Include:可通过此方法查询出与相关的实体对象。实例代码如下：

```
using (var edm = new NorthwindEntities())
    {
            string esql1 = "select value c from NorthwindEntities.Customers
as c WHERE c.CustomerID ='HANAR'";
```

```
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
                query1 = query1.Include("Orders");
                foreach (Customers c in query1)
                {
                    Console.WriteLine("{0},{1}", c.CustomerID,
c.Orders.Count);
                    //输出：HANAR,14
                }

    }
```

- OfType： 根据制定类筛选元素创建一个新的类型。此类型是要在实体模型中已定义过的。
- OrderBy

实例代码如下：

```
using (var edm = new NorthwindEntities())
{
                string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
                query1.OrderBy("it.country asc,it.city asc");
                    //也可以这样写
                    //query1.OrderBy("it.country asc");
                //query1.OrderBy("it.city asc");
                foreach (Customers c in query1)
                {
                    Console.WriteLine("{0},{1}", c.Country, c.City);
                }
```

```
}
```

- Select

实例代码如下：

```
using (var edm = new NorthwindEntities())
{
            string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
            ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
            ObjectQuery<DbDataRecord> records =
query1.Select("it.customerid,it.country");
            foreach (DbDataRecord c in records)
            {
                Console.WriteLine("{0},{1}", c[0], c[1]);
            }
            Console.WriteLine(records.ToTraceString());
            //SQL 输出：
            //SELECT TOP (10)
            //1 AS [C1],
            //[Extent1].[CustomerID] AS [CustomerID],
            //[Extent1].[Country] AS [Country]
            //FROM [dbo].[Customers] AS [Extent1]
            //ORDER BY [Extent1].[CustomerID] ASC
```

```
}
```

- SelectValue

实例代码如下：

```
using (var edm = new NorthwindEntities())
{
```

```
                string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID limit 10";
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
                ObjectQuery<string> records =
query1.SelectValue<string>("it.customerid");
                foreach (string c in records)
                {
                    Console.WriteLine("{0}", c);
                }
                Console.WriteLine(records.ToTraceString());
                //SQL 输出：
                //SELECT TOP (10)
                //[Extent1].[CustomerID] AS [CustomerID]
                //FROM [dbo].[Customers] AS [Extent1]
                //ORDER BY [Extent1].[CustomerID] ASC
        }
```

- Skip/Top

实例代码如下：

```
    using (var edm = new NorthwindEntities())
    {
                string esql1 = "select value c from NorthwindEntities.Customers
as c order by c.CustomerID ";
                ObjectQuery<Customers> query1 =
edm.CreateQuery<Customers>(esql1);
                query1 = query1.Skip("it.customerid asc", "10");
                query1 = query1.Top("10");
                foreach (Customers c in query1)
                {
```

```
            Console.WriteLine("{0}", c.CustomerID);
        }
        Console.WriteLine(query1.ToTraceString());
        //SQL 输出：
        //SELECT TOP (10)
        //[Extent1].[CustomerID] AS [CustomerID]
        //FROM [dbo].[Customers] AS [Extent1]
        //ORDER BY [Extent1].[CustomerID] ASC
    }
```

本节，简单的介绍一下与 ObjectQuery 查询相关的语法，我个人觉得查询写法比较多，需要在日常的编程中去发现，在这里就不一一复述了。下节，将介绍 EntityClient 相关的内容。

Entity Framework 学习初级篇 6--EntityClient

System.Data.EntityClient 命名空间是 实体框架的 .NET Framework 数据提供程序。EntityClient 提供程序使用存储特定的 ADO.NET 数据提供程序类和映射元数据与实体数据模型进行交互。EntityClient 首先将对概念性实体执行的操作转换为对物理数据源执行的操作。然后再将物理数据源返回的结果集转换为概念性实体。

EntityClient 下的类有以下几个：

- EntityConnection
- EntityCommand
- EntityConnectionStringBuilder
- EntityParameter
- EntityDataReader
- EntityParameterCollection
- EntityProviderFactory
- EntityTransaction

从类的名字上看，我们就知道它们的作用是什么了。在此，就不再一一解释了。直接通过实例代码来学习它们。

- EntityConnection:

实例代码 1：

```
string con = "name = NorthwindEntities";
using (EntityConnection econn = new EntityConnection(con))
{
    string esql = "Select VALUE c from NorthwindEntities.Customers as c where c.CustomerID='ALFKI'";
    econn.Open();
    EntityCommand ecmd = new EntityCommand(esql, econn);
    EntityDataReader ereader = ecmd.ExecuteReader(CommandBehavior.SequentialAccess);
    if (ereader.Read())
    {
        Console.WriteLine(ereader["CustomerID"]);
    }
    Console.WriteLine(ecmd.ToTraceString());
}
```

上述代码中，需要注意的是 EntityConnection 的构造方法。其中，连接字符串写法有多很，如下：

写法 1：

string con ="name = NorthwindEntities" ;其中的”NorthwindEntities”是配置文件中的连接字符串名称

写法 2:

string con = System.Configuration.ConfigurationManager.ConnectionStrings["NorthwindEntities"].ConnectionString; 其中的”NorthwindEntities”是配置文件中的连接字符串名称

写法 3:

string con = @"metadata=res://*/NorthWind.csdl|res://*/NorthWind.ssdl|res://*/NorthWind.msl;provider=System.Data.SqlClient;provider connection string='Data Source=.\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=True;MultipleActiveResultSets=True'";其中的这些字符串是配置文件中的连接字符串的值

写法 4:

```
NorthwindEntities edm = new NorthwindEntities();
string con = edm.Connection.ConnectionString;
```

上述写法中,基于写法简单、方便我比较推荐使用第 1 种或者第 2 种写法。

- EntityCommand

它具有的方法有：ExecuteDbDataReader、 ExecuteNonQuery 、 ExecuteReader 、ExecuteScalar 等。

实例代码 2：

```
        string con = "name = NorthwindEntities";
         using (EntityConnection econn = new EntityConnection(con))
         {
             string esql = "Select VALUE c from NorthwindEntities.Customers as c where c.CustomerID='ALFKI'";
             econn.Open();
             EntityCommand ecmd = econn.CreateCommand();
```

```
                ecmd.CommandText = esql;
                EntityDataReader ereader =
ecmd.ExecuteReader(CommandBehavior.SequentialAccess);
                if (ereader.Read())
                {
                    Console.WriteLine(ereader["CustomerID"]);
                }
                Console.WriteLine(ecmd.ToTraceString());
            }
```

代码中，EntityCommand 创建方式和实例代码 1 中的稍有不同，相信大家都明白，就不多说了。

- EntityConnectionStringBuilder

实例代码 3：

```
            EntityConnectionStringBuilder esb = new
EntityConnectionStringBuilder();
            esb.Provider = "System.Data.SqlClient";
            esb.Metadata =
@"res://*/NorthWind.csdl|res://*/NorthWind.ssdl|res://*/NorthWind.msl";
            esb.ProviderConnectionString = @"Data Source=.\SQLEXPRESS;Initial
Catalog=Northwind;Integrated Security=True;MultipleActiveResultSets=True";
          EntityConnection econn = new
EntityConnection(esb.ConnectionString)//创建连接
```

- EntityParameter

代码实例 4：

```
            string con = "name = NorthwindEntities";
            using (EntityConnection econn = new EntityConnection(con))
```

```
            {
                string esql = "Select value c from NorthwindEntities.Customers
as c order by c.CustomerID skip @start limit @end";
                econn.Open();
                EntityCommand ecmd = new EntityCommand(esql, econn);
                EntityParameter p1 = new EntityParameter("start",
DbType.Int32);
                p1.Value = 0;
                EntityParameter p2 = new EntityParameter("end", DbType.Int32);
                p2.Value = 10;
                ecmd.Parameters.Add(p1);
                ecmd.Parameters.Add(p2);
                 EntityDataReader ereader =
ecmd.ExecuteReader(CommandBehavior.SequentialAccess);
                while (ereader.Read())
                {
                    Console.WriteLine(ereader["CustomerID"]);
                }
                Console.WriteLine(ecmd.ToTraceString());
            }
```

其中，参数是以@符号前缀的，EntityParameter 实体参数类，除了可以直接构造出实例来。为实体命令对象添加参数，我们还可以直接调用 Parameters.AddWithValue 方法。如下代码：

```
        ecmd.Parameters.AddWithValue("start", 0);
        ecmd.Parameters.AddWithValue("end", 10);
```

我比较喜欢用上面的代码，简单、方便。

- EntityDataReader

```
string con = "name = NorthwindEntities";
using (EntityConnection econn = new EntityConnection(con))
{
    string esql = "Select value c from NorthwindEntities.Customers as c order by c.CustomerID limit 10 ";
    econn.Open();
    EntityCommand ecmd = new EntityCommand(esql, econn);
    EntityDataReader ereader = ecmd.ExecuteReader(CommandBehavior.SequentialAccess);
    while (ereader.Read())
    {
        Console.WriteLine("{0},{1},{2},{3},{4}", ereader[0], ereader[1], ereader[2], ereader[3], ereader[4]);
    }
    Console.WriteLine(ecmd.ToTraceString());
}
```

需要注意的是：CommandBehavior.SequentialAccess;这个地方。不同的枚举项，对查询会有不同影响。枚举如下：

- Default 此查询可能返回多个结果集。在功能上等效于调用 ExecuteReader()。
- SingleResult 查询返回一个结果集。
- SchemaOnly 查询仅返回列信息。
- KeyInfo 此查询返回列和主键信息。
- SingleRow 查询应返回一行。

- SequentialAccess 提供一种方法，以便 DataReader 处理包含带有大二进制值的列的行。
- CloseConnection 在执行该命令时，如果关闭关联的 DataReader 对象，则关联的 Connection 对象也将关闭。

需要说明的是，如果使用 SequentialAccess 则需按顺序访问列，否则将抛异常。如下代码，将会抛异常：

```
while (ereader.Read())
{
    //异常信息：从列序列号“1”开始读取的尝试无效。通过 CommandBehavior.SequentialAccess，只能从列序列号“5”或更大值处开始读取
    Console.WriteLine("{0},{1},{2},{3},{4}", ereader[4], ereader[1], ereader[2], ereader[3], ereader[0]);
}
```

- EntityTransaction：

事务类。目前由于 ESQL 仅提供查询的命令，没有提供对 Insert、Update、Delete 等的支持。所以，我觉得目前这个类基本没有用，（不可能我做查询还使用事务吧！）。

从上述简单的介绍，我们可以看到，EntityClient 和 SqlClient 下的类基本上是一致的。所以很容易掌握。其他就不多说了。

Entity Framework 学习初级篇 7--基本操作：增加、更新、删除、事务

本节，直接写通过代码来学习。这些基本操作都比较简单，与这些基本操作相关的内容在之前的 1 至 6 节基本介绍完毕。

- 增加：

方法 1：使用 AddToXXX(xxx)方法：实例代码如下：

```
using (var edm = new NorthwindEntities())
{
```

```
                Customers c = new Customers { CustomerID = "c#", City = "成都市", Address = "中国四川省", CompanyName = "cnblogs", Country = "中国", Fax = "10086", Phone = "1008611", PostalCode = "610000", Region = "天府广场", ContactName = "风车车.Net" };
                edm.AddToCustomers(c);
                int result = edm.SaveChanges();
                Assert.AreEqual(result, 1);
                Customers addc = edm.Customers.FirstOrDefault(cc => cc.CustomerID == "c#");
                Console.WriteLine("CustomerId={0},City={1}", addc.CustomerID, addc.City);
            }
```

方法 2：使用 ObjectContext 的 AddObject(string entitySetName, object entity)方法。实例代码如下：

```
                using (var edm = new NorthwindEntities())
            {
                    Customers c = new Customers { CustomerID = "c2", City = "成都市 2", Address = "中国四川省 2", CompanyName = "cnblogs", Country = "中国", Fax = "10086", Phone = "1008611", PostalCode = "610000", Region = "天府广场", ContactName = "风车车.Net" };
                    edm.AddObject("Customers", c);
                    int result = edm.SaveChanges();
                    Assert.AreEqual(result, 1);
                    Customers addc = edm.Customers.FirstOrDefault(cc => cc.CustomerID == "c2");
                    Console.WriteLine("CustomerId={0},City={1}", addc.CustomerID, addc.City);
```

```
        }
```

其中，在代码中，需要注意的是：AddObject 方法中参数“entitySetName ”就是指对应实体名称，应该是：“Customers”，而不是“NorthwindEntities.Customers”；

- 更新：

```
        using (var edm = new NorthwindEntities())
        {
            Customers addc = edm.Customers.FirstOrDefault(cc => cc.CustomerID == "c2");
            addc.City = "CD";
            addc.ContactName = "cnblogs";
            addc.Country = "CN";
            int result = edm.SaveChanges();
            Assert.AreEqual(result, 1);
            Customers updatec = edm.Customers.FirstOrDefault(cc => cc.CustomerID == "c2");
            Console.WriteLine("CustomerId={0},City={1}", updatec.CustomerID, updatec.City);
        }
```

其中，需要注意的是：不能去更新主键，否则会报“System.InvalidOperationException： 属性“xxx”是对象的键信息的一部分，不能修改。”

- 删除：

实例代码如下：

```
        using (var edm = new NorthwindEntities())
        {
```

```
                    Customers deletec = edm.Customers.FirstOrDefault(cc =>
cc.CustomerID == "c2");
                    edm.DeleteObject(deletec);
                    int result = edm.SaveChanges();
                    Assert.AreEqual(result, 1);
                    Customers c = edm.Customers.FirstOrDefault(cc =>
cc.CustomerID == "c2");
                    Assert.AreEqual(c, null);

        }
```

- 事务：

实例代码如下：

```
            NorthwindEntities edm = null;
            System.Data.Common.DbTransaction tran = null;
            try
            {
                edm = new NorthwindEntities();
                edm.Connection.Open();
                tran = edm.Connection.BeginTransaction();
                Customers cst = edm.Customers.FirstOrDefault(cc =>
cc.CustomerID == "c#");
                cst.Country = "CN";
                cst.City = "CD";
                edm.SaveChanges();
                tran.Commit();
```

```
            }
            catch (Exception ex)
            {
                if (tran != null)
                    tran.Rollback();
                throw ex;
            }
            finally
            {
                if (edm != null && edm.Connection.State != System.Data.ConnectionState.Closed)
                    edm.Connection.Close();
            }
```

至此，初级篇基本介绍完毕。后面，打算写点，中级篇的东西。

Entity Framework 学习中级篇 1—EF 支持复杂类型的实现

本节，将介绍如何手动构造复杂类型(ComplexType)以及复杂类型的简单操作。

通常，复杂类型是指那些由几个简单的类型组合而成的类型。比如：一张 Customer 表，其中有 FristName 和 LastName 字段，那么对应的 Customer 实体类将会有 FristName 和 LastName 这两个属性。当我们想把 FirstName 和 LastName 合成一个名为 CustomerName 属性时，此时，如果要在 EF 中实现这个目的，那么我们就需要用到复杂类型。

目前，由于 EF 不能显示支持复杂类型，所以我们无法在 VS 里的可视化设计器里面来设计我们需要的复杂类型。所以，我们需要手动修改实体模型，以便使其支持复杂类型的属性。修改的主要步骤有以下几步：

- 产生实体模型
- 修改 CSDL 文件

- 修改 msl 文件
- 重新生成模型实体类

在后续的介绍，我使用数据库使用的是 NorthWind，并针对 Customer 表对应的实体类来增加复杂属性 Address，其中复杂属性 Address 由 Address,City,Region,Country 和 PostalCode 这个几个组合而成。

下面，介绍具体的操作步骤：

第一步：产生实体模型

实体模型的产生我们可以直接通过在 VS 可视化设计器来产生（如果不会，请参考《Entity Framework 学习初级篇 1--EF 基本概况》）。或者使用 EdmGen 工具来产生（EdmGen 工具位于：系统盘符:\WINDOWS\Microsoft.NET\Framework\v3.5 下面）。具体步骤就不复述了。

我产生的实体模型文件是：NorthwindEnites.edmx

第二步：修改 csdl 文件

产生了实体模型后，我们使用记事本或其他文本编辑工具打开实体模型，（小技巧：可以把实体模型后缀.edmx 改为.xml，然后把实体模型文件直接拖到 VS 里面进行修改，这样修改起来比较方便，待修改完毕后，将后缀改回来即可。）

接着，开始手动修改 csdl 文件，找到模型文件中关于 csdl 定义的部分，然后找到实体类型名为 Customers 的定义节，删除原来的 Address,City,Region,Country，PostalCode 属性定义，然后添加一个名为 Address 的属性，如下代码所示：

```
<EntityType Name="Customers">
        <Key>
          <PropertyRef Name="CustomerID" />
        </Key>
        <Property Name="CustomerID" Type="String" Nullable="false" MaxLength="5"
Unicode="true" FixedLength="true" />
        <Property Name="CompanyName" Type="String" Nullable="false" MaxLength="40"
Unicode="true" FixedLength="false" />
        <Property Name="ContactName" Type="String" MaxLength="30" Unicode="true"
FixedLength="false" />
```

```
            <Property Name="ContactTitle" Type="String" MaxLength="30" Unicode="true"
FixedLength="false" />
            <Property Name="Address" Type="NorthwindModel.CommonAddress"
Nullable="false"></Property>
            <Property Name="Phone" Type="String" MaxLength="24" Unicode="true"
FixedLength="false" />
            <Property Name="Fax" Type="String" MaxLength="24" Unicode="true"
FixedLength="false" />
            <NavigationProperty Name="Orders"
Relationship="NorthwindModel.FK_Orders_Customers" FromRole="Customers" ToRole="Orders" />
            <NavigationProperty Name="CustomerDemographics"
Relationship="NorthwindModel.CustomerCustomerDemo" FromRole="Customers"
ToRole="CustomerDemographics" />
        </EntityType>
```

接着，需要添加一个名为 CommonAddress 复杂类型的定义，具体如下代码：

```
        <ComplexType Name="CommonAddress">
            <Property Name="Address" Type="String" MaxLength="60" Unicode="true"
FixedLength="false" />
            <Property Name="City" Type="String" MaxLength="15" Unicode="true"
FixedLength="false" />
            <Property Name="Region" Type="String" MaxLength="15" Unicode="true"
FixedLength="false" />
            <Property Name="PostalCode" Type="String" MaxLength="10" Unicode="true"
FixedLength="false" />
            <Property Name="Country" Type="String" MaxLength="15" Unicode="true"
FixedLength="false" />
        </ComplexType>
```

至此，csdl 部分修改完毕。

第三步，修改 msl 文件：

找到 msl 部分的定义，修改 Customers 部分的影射定义。具体代码如下（请注意 ComplexProperty 节）：

```
<EntitySetMapping Name="Customers">
        <EntityTypeMapping TypeName="IsTypeOf(NorthwindModel.Customers)">
          <MappingFragment StoreEntitySet="Customers">
            <ScalarProperty Name="CustomerID" ColumnName="CustomerID" />
            <ScalarProperty Name="CompanyName" ColumnName="CompanyName" />
            <ScalarProperty Name="ContactName" ColumnName="ContactName" />
            <ScalarProperty Name="ContactTitle" ColumnName="ContactTitle" />
            <ComplexProperty Name="Address" TypeName="NorthwindModel.CommonAddress">
              <ScalarProperty Name="Address" ColumnName="Address" />
              <ScalarProperty Name="City" ColumnName="City" />
              <ScalarProperty Name="Region" ColumnName="Region" />
              <ScalarProperty Name="PostalCode" ColumnName="PostalCode" />
              <ScalarProperty Name="Country" ColumnName="Country" />
            </ComplexProperty>
            <ScalarProperty Name="Phone" ColumnName="Phone" />
            <ScalarProperty Name="Fax" ColumnName="Fax" />
          </MappingFragment>
        </EntityTypeMapping>
    </EntitySetMapping>
```

至此，msl 部分修改完毕

第四步：重新产生实体类文件。

我们可以使用 EmdGen2 工具来重新实体类.cs 文件。具体操作如下：

将修改好的模型文件（edmx），拷贝到使用 edmgen2.exe 同目录下，然后在命令行中输入：

Edmgen2 /codegen cs NorthwindEnites.edmx

执行此命令后，会在当前的文件夹下生成一个 NorthwindEnites.cs 代码文件，也就是实体类的代码文件。将改文件改名为：NorthwindEnites.Designer.cs（这步主要是和 edmx 对应起来）。

然后，将 NorthwindEnites.edmx 和 NorthwindEnites.Designer.cs 文件添加到项目中。

至此，复合类型的修改完毕。

按照同样的修改过程，我们可以给 Employees 也增加一个 Address 的复杂类型属性。接下来，我们看看具体使用代码：

- 查询：

```
[Test]
public void TestAddress()
{
    using (var db = new NorthwindModel.NorthwindEntities1())
    {
        Console.WriteLine("Get Five customer addresss :");
        var cts = db.Customers.Take(5);
        foreach (var c in cts)
        {
            Console.WriteLine("Address:{0},Country:{1},City:{2},PostalCode:{3}",
c.Address.Address, c.Address.Country, c.Address.City, c.Address.PostalCode);
        }
        Console.WriteLine("Get Five Employess address:");
        var emp = db.Customers.Take(5);
        foreach (var c in emp)
        {
            Console.WriteLine("Address:{0},Country:{1},City:{2},PostalCode:{3}",
c.Address.Address, c.Address.Country, c.Address.City, c.Address.PostalCode);
        }
    }
}
```

- 添加：

```
[Test]
public void AddTest()
{
```

```
            using (var db = new NorthwindModel.NorthwindEntities1())
            {
                var customer = new NorthwindModel.Customers
                {
                    CustomerID = "2009",
                    CompanyName = "Complex Company",
                    ContactName = "xray2005",
                    Address = new NorthwindModel.CommonAddress
                    {
                        Address = "SiChuan,China",
                        City = "ChengDou",
                        Country = "China",
                        PostalCode = "610041",
                        Region = "Chenghua"
                    }
                };
                db.AddToCustomers(customer);
                db.SaveChanges();
                var cst = db.Customers.FirstOrDefault(c => c.CustomerID == "2009");
                Assert.IsNotNull(cst);

Console.WriteLine("CustomerID:{0},CompanyName:{1},ContactName:{2},City:{3},Country:{4}",
cst.CustomerID, cst.CompanyName, cst.ContactName, cst.Address.City, cst.Address.Country);
            }
        }
```

- 条件查询：

```
        [Test]
        public void QueryTest()
        {
            using (var db = new NorthwindModel.NorthwindEntities1())
```

```
            {
                var cst = db.Customers.FirstOrDefault(c => c.Address.City == "ChengDou");
                Assert.IsNotNull(cst);

Console.WriteLine("CustomerID:{0},CompanyName:{1},ContactName:{2},City:{3},Country:{4}",
cst.CustomerID, cst.CompanyName, cst.ContactName, cst.Address.City, cst.Address.Country);
            }
        }
```

最后，补充说明：

1， 在 VS 的可视化设计器里，不支持复杂类型，所以修改后无法再在可视化设计器里修改模型(edmx 文件)。

2， 复杂类型不能单独存在，它必须和某一实体相关起来。

3， 复杂类型不能包含导航属性，如导航到实体或实体集。

4， 复杂类型具有内部结构但没有 Key（主键） 属性的数据类型

下面是示例代码和 EdmGen2 工具的连接。

示例代码 EdmGen2

Entity Framework 学习中级篇 2—存储过程(上)

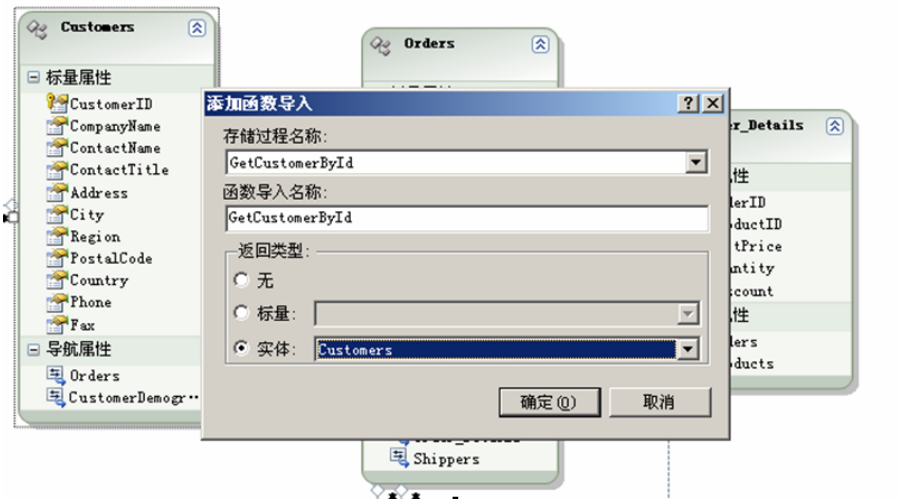
目前，EF 对存储过程的支持并不完善。存在以下问题：

- EF 不支持存储过程返回多表联合查询的结果集。
- EF 仅支持返回返回某个表的全部字段，以便转换成对应的实体。无法支持返回部分字段的情况。
- 虽然可以正常导入返回标量值的存储过程，但是却没有为我们自动生成相应的实体.cs 代码，我们还是无法在代码中直接调用或使用标量存储过程
- EF 不能直接支持存储过程中 Output 类型的参数。
- 其他一些问题。

下面，主要针对如何使用存储过程，以及存储返回实体、表的部分字段这个几个问题，做具体介绍。

- 导入存储过程及返回实体

在 VS 可视化设计器中，打开实体模型（emdx 文件）。然后，鼠标右键点击“Customers”→ “添加”→“函数导入”，然后选择“存储过程名称”，并输入函数导入名称，选择返回类型为实体并选择 Customers。如下图所示：



之后，点击“确定”。之后，存储过程导入。在代码我们就可以使用改存储过程了。如下代码所示：

```
[Test]
public void GetEntityBySP()
{
    using (var db = new NorthwindEntities())
    {
        var cst = db.GetCustomerById("ALFKI").FirstOrDefault();
        Assert.IsNotNull(cst);
        Console.WriteLine("CustomerId:{0},ContactName:{1}",
cst.CustomerID, cst.ContactName);
    }
}
```

- 联合查询结果集的问题

在此版本的 EF 中，是不支持存储过程的多张表联合查询的，也就是说查询的结果集中，一部分字段来自表 A，另一部分字段来自表 B，像这种情况，目前 EF 无法直接进行处理。为此，可以通过写两个或多个存储过程来实现。比如：第一个存储过程返回表 A 的所有字段，第二存储过程返回表 B 的所有字段；然后，我们在代码中来实现联合的查询。

按照前面的思路，增加一个返回所有的 Orders 表字段的存储过程 GetOrdersByCustomerId，再增加一个返回 Order Details 表全部字段的存储过程 GetDetailsByCustomerId，并将它们导入到实体模型中。

其中，GetOrdersByCustomerId 存储过程如下：

```
CREATE PROCEDURE GetOrdersByCustomerId
    @CustomerId varchar(5)
AS
BEGIN
    SET NOCOUNT ON;
    SELECT * FROM orders WHERE orders.CustomerID=@CustomerId;
END
```

GetDetailsByCustomerId 存储过程如下：

```
CREATE PROCEDURE GetDetailsByCustomerId
    @CustomerId varchar(5)
AS
BEGIN
    SET NOCOUNT ON;
    SELECT d.*
  FROM  Orders o,[Order Details] d
    WHERE o.OrderId=d.OrderId  AND o.CustomerId=@CustomerId;
END
```

之后，在我们的代码来实现联合查询。代码如下：

```
[Test]
```

```
        public void GetOrderBySp()
        {
            using (var db = new NorthwindEntities())
            {
                var orders = db.GetOrdersByCustomerId("VINET").ToList();
                var details =
db.GetDetailsByCustomerId("VINET").ToList();
                orders.ForEach(o =>
o.Order_Details.Attach(details.Where(d => d.OrderID == o.OrderID)));
                foreach (var order in orders)
                {
                    Console.WriteLine(order.OrderID);
                    foreach (var d in order.Order_Details)
                        Console.WriteLine(d.ProductID);
                }
            }
        }
```

其中，需要注意的，由于是分别执行了两个存储，在内存中是以两个对立的对象存在，它们之前是没有建立联系的。为此，我们需要使用 Attach 方法来把他们联系起来(红色代码段)，这样我们就可以通过导航来访问对象的实体了，也就是 foreach (var d in order.Order_Details)中的 order.Order_Details。

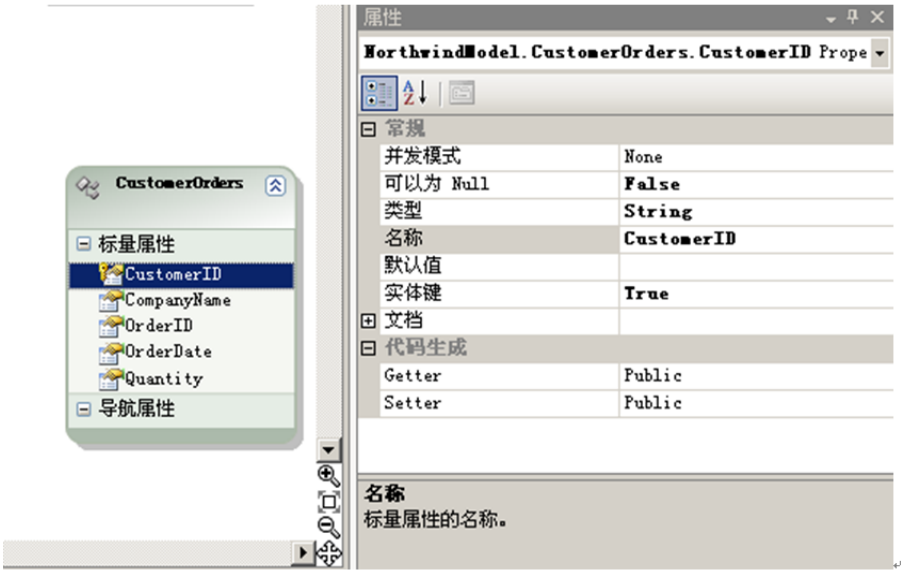
- 返回部分字段的问题

默认情况，目前此版本的 EF 在使用存储过程返回实体的时候，必须返回所有的字段，以便是 EF 能够自动将返回的结果转换成对应的实体。否则会报“数据读取器与指定的 XXX 类型不兼容的异常，....”。

接下来，我们通过建立一个存储过程，并建立新建立一个实体来解决此问题。首先，我们在数据库中建立一个名为 GetCustomerAndOrders 的存储过程，其定义如下：

```
CREATE PROCEDURE GetCustomerAndOrders
    AS
BEGIN
    SET NOCOUNT ON;
    SELECT
c.CustomerID,c.CompanyName,o.OrderID,o.OrderDate,d.Quantity
    FROM Customers c,Orders o,[Order Details] d
    WHERE c.CustomerID=o.CustomerID AND o.OrderID=d.OrderID;
END
GO
```

然后，添加一个实体 CustomerOders，并设置属性如下图所示：



然后，在 VS 可视化设计器中，打开实体模型（emdx 文件），通过添加→函数导入，导入存储过程 GetCustomerAndOrders，并取名为 GetCustomerAndOrders，返回类型设置为实体 CustomerOrders。

最后，我们就可以代码实体此实体和存储过程了。如下代码：

```
[Test]
```

```
public void GetOrderCountByCustomerId()
{
    using (var db = new NorthwindEntities())
    {
        var co = db.GetCustomerAndOrders().Take(10).Skip(0);
        foreach(var c in co)
            Console.WriteLine(c.CustomerID);
    }
}
```

- 返回标量值问题

目前，EF 对存储过程返回标量值的支持并没有完全。虽然，我们可以按照前面的步骤导入函数，并设置返回标量值的类型，同时 EF 会在实体模型文件为我们自动生成此存储过程相关的映射配置等。但是，EF 却没有为我们生成在实体模型 cs 文件中，自动生成相应的.cs 代码，所以我们根本无法直接调用此存储过程。为解决此问题，我们需要手动添加代码到实体模型的.cs 代码文件中。

首先,在数据库中建立存储存储过程 GetOrderCntByCustomerId,代码如下：

```
CREATE PROCEDURE GetOrderCntByCustomerId
    @CustomerId varchar(5)
AS
BEGIN
    SET NOCOUNT ON;
    SELECT count(*) FROM Orders WHERE
Orders.CustomerId=@CustomerId;
END
```

接着，按照正常的步骤，导入此存储过程并设置返回类型为标量类型的 Int32。

然后，我们需要添加一个泛型的方法和一个执行存储过程的方法，代码如下：

```
public partial class NorthwindEntities
```

```
  {
      //泛型方法用于执行标量存储过程
        private T ExecuteFunction<T>(string functionName,
System.Data.EntityClient.EntityParameter[] parameters) where T :
struct
        {
            System.Data.EntityClient.EntityCommand cmd =
((System.Data.EntityClient.EntityConnection)this.Connection).Creat
eCommand();
            cmd.CommandType =
System.Data.CommandType.StoredProcedure;
            cmd.Parameters.AddRange(parameters);
            cmd.CommandText = this.DefaultContainerName + "." +
functionName;
            try
            {
                if (cmd.Connection.State !=
System.Data.ConnectionState.Open)
                    cmd.Connection.Open();
                var obj = cmd.ExecuteScalar();
                return (T)obj;
            }
            catch (System.Exception)
            {
                throw;
            }
            finally
            {
                cmd.Connection.Close();
            }
```

```
        }
        //执行数据库中 GetOrderCntByCustomerId 存储过程的方法
        public int GetOrderCountByCustomerId(string CustomerId)
        {
            var param = new
System.Data.EntityClient.EntityParameter("CustomerId",
System.Data.DbType.String);
            param.Value = CustomerId;
            return
ExecuteFunction<int>("GetOrderCountByCustomerId", new[] { param });
        }
    }
```

最后，通过以上修改，我们就可以直接使用返回标量值的存储过程，代码如下：

```
[Test]
  public void GetOrderCountByCustomerId()
  {
      using (var db = new NorthwindEntities())
      {
          var result = db.GetOrderCountByCustomerId("VINET");
          Assert.Greater(result, 0);
          Console.WriteLine(result.ToString());
      }
  }
```

至此，我们就解决了 EF 存储过程返回标量的问题。

Entity Framework 学习中级篇 3—存储过程(中)

目前，EF 对存储过程的支持并不完善。存在以下问题：

- EF 不支持存储过程返回多表联合查询的结果集。
- EF 仅支持返回返回某个表的全部字段，以便转换成对应的实体。无法支持返回部分字段的情况。
- 虽然可以正常导入返回标量值的存储过程，但是却没有为我们自动生成相应的实体.cs 代码，我们还是无法在代码中直接调用或使用标量存储过程
- EF 不能直接支持存储过程中 Output 类型的参数。
- 其他一些问题。

本节，我们将学习如何手动添加/修改存储过程，如何使 EF 能够支持 Output 类型的参数。

- 添加/修改存储过程

有时候，某个 SQL 语句比较复杂，但是数据库中又没有定义相应的存储过程。这个时候，我们又想使上层代码比较简单、方便的方式来完成此项任务。那么，此时，我们便可以手工在实体模型（.edmx 文件）中添加自己需要的存储过程了。这样既方便上层调用又方便后期的修改。

以手动修改实体模型 edmx 文件，添加名为 CustomerByCommandText 的存储过程为例。具体步骤如下：

修改实体模型文件，找到 ssdl 部分，添加如下代码：

```
<Function Name="CustomerByCommandText" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo" >
  <CommandText>
    select c.* from Customers c,Orders o where
c.CustomerID=o.CustomerID
  </CommandText>
</Function>
```

然后，再找到 csdl 部分，添加如下代码：

```
<FunctionImport Name="CustomerByCommandText" EntitySet="Customers"
ReturnType="Collection(NorthwindModel.Customers)"></FunctionImport>
```

接着，找到 msl 部分，添加如下代码：

```
<FunctionImportMapping FunctionImportName="CustomerByCommandText"
FunctionName="NorthwindModel.Store.CustomerByCommandText"/>
```

最后，在实体模型的.cs 文件里面，添加一个执行此存储过程的方法，代码如下：

```
        public global::System.Data.Objects.ObjectResult<Customers>
GetCustomerByCommandText()
        {
            return
base.ExecuteFunction<Customers>("CustomerByCommandText");
        }
```

至此，修改完毕。

现在，我们就可以在代码使用刚才手工定义的存储过程了。如下代码所示：

```
       [Test]
        public void GetCustomerByCmdText()
        {
            using (var db = new NorthwindEntities())
            {
                var csts =
db.GetCustomerByCommandText().Take(10).Skip(0);
                foreach (var c in csts)
                    Console.WriteLine(c.CustomerID);
            }
        }
```

其实，关键的地方就是 CommandText 这个部分的内容，它里面就是要执行的 SQL 语句。另外，我们可以在修改实体模型 emdx 文件的同时，我们可以看到所有的实体类查询的 SQL 语句命令都可以在 edmx 文件里找到，我们都可以进行相应的修改。

- Output 类型参数

在实际应用当中，很多时候，我们需要使用 output 类型的存储过程参数，以便返回我们需要的值。但是，目前，EF 不能直接支持 output 类型的存储过程参数。为此，我们需要对实体模型进行修改，以便使其支持 output 类型的输出参数。具体过程如下：

在数据库中建立一个为名的 GetNameByCustomerId 存储过程，代码如下：

```
CREATE PROCEDURE GetNameByCustomerId
 @CustomerId varchar(5),
 @ContactName varchar(30) output
AS
BEGIN
 SET NOCOUNT ON;
 SELECT @ContactName=ContactName
 FROM Customers
 WHERE CustomerID=@CustomerId;
END
```

然后，开始修改实体模型 edmx 文件。

先找到 ssdl 定义的部分，添加如下代码：

```
<Function Name="GetNameByCustomerId" Aggregate="false"
BuiltIn="false" NiladicFunction="false" IsComposable="false"
ParameterTypeSemantics="AllowImplicitConversion" Schema="dbo">
        <Parameter Name="CustomerId" Type="varchar" Mode="In"
MaxLength="5"></Parameter>
        <Parameter Name="ContactName" Type="varchar" Mode="Out"
MaxLength="30"></Parameter>
```

```
        </Function>
```

接着，在找到 csdl 定义的部分，添加如下代码：

```
      <FunctionImport Name="GetNameByCustomerId">
            <Parameter Name="CustomerId" Mode="In" Type="String"
MaxLength="5"></Parameter>
            <Parameter Name="ContactName" Mode="Out" Type="String"
MaxLength="30"></Parameter>
      </FunctionImport>
```

最后，找到 msl 定义的部分，添加如下代码：

```
<FunctionImportMapping FunctionImportName="GetNameByCustomerId"
FunctionName="NorthwindModel.Store.GetNameByCustomerId"></FunctionImport
Mapping>
```

至此，实体模型 emdx 文件修改完毕。

接下来，我们需要在实体模型的.cs 文件中，增加相应的调用方法。代码如下：

```
    public partial class NorthwindEntities1
   {


     //执行 GetNameByCustomerId 的方法
        public void GetNameByCustomerId(string CustomerId, out string
ContactName)
        {
           ContactName = string.Empty;
            var Pars = new System.Data.EntityClient.EntityParameter[]
            {
```

```
                new
System.Data.EntityClient.EntityParameter{ ParameterName="CustomerId",
DbType=System.Data.DbType.String,Value=CustomerId},
                new
System.Data.EntityClient.EntityParameter{ParameterName="ContactName",
DbType=System.Data.DbType.String,
Direction=System.Data.ParameterDirection.Output}
            };
            this.ExecuteNonQuery("GetNameByCustomerId", Pars);
            ContactName = Pars[1].Value.ToString();

        }

       //辅助方法，执行 SQL 命令
        private void ExecuteNonQuery(string functionName,
System.Data.EntityClient.EntityParameter[] parameters)
        {
            System.Data.EntityClient.EntityCommand cmd =
((System.Data.EntityClient.EntityConnection)this.Connection).CreateComma
nd();
            cmd.CommandType = System.Data.CommandType.StoredProcedure;
            cmd.Parameters.AddRange(parameters);
            cmd.CommandText = this.DefaultContainerName + "." +
functionName;
            try
            {
```

```
                if (cmd.Connection.State !=
System.Data.ConnectionState.Open)
                    cmd.Connection.Open();
                cmd.ExecuteNonQuery();
            }
            catch (System.Exception)
            {
                throw;
            }
            finally
            {
                cmd.Connection.Close();
            }
        }
    }
```

现在，所有的修改工作都做完了。接下来，我们就可以在代码中直接调用此存储过程了。示例代码如下：

```
        [Test]
        public void OutputTest()
        {
            using (var db = new NorthwindModel.NorthwindEntities1())
            {
                string contactname = string.Empty;
                db.GetNameByCustomerId("ALFKI", out contactname);
                Assert.IsTrue(!string.IsNullOrEmpty(contactname));
```

```
            Console.WriteLine(contactname);
        }
    }
```

至此，我们便可以使用 Output 类型的输出参数了。

Entity Framework 学习中级篇 4—存储过程(下)

在 EF 中，各个实体的插入、更新和删除也都通过使用存储过程来完成，以便提高点性能。这个类似于数据集。其步骤是：先定义存储过程，然后在 VS 的可视化设计器，设置存储过程映射即可。

下面，以为 Supplier 实体映射存储过程为例。

分别建立插入、更新和删除存储过程。

InsertSuppliers 存储过程定义如下：

```
CREATE PROCEDURE [dbo].[InsertSuppliers]
    -- Add the parameters for the stored procedure here
@CompanyName nvarchar(40),
@ContactName nvarchar(30),
@ContactTitle nvarchar(30),
@Address nvarchar(60),
@City nvarchar(15),
@Region nvarchar(15),
@PostalCode nvarchar(10),
@Country nvarchar(15),
@Phone nvarchar(24),
@Fax nvarchar(24),
@HomePage ntext
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
```

```
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    INSERT INTO
Suppliers(CompanyName,ContactName,ContactTitle,Address,City,Region,P
ostalCode,Country,Phone,Fax,HomePage)
    VALUES(@CompanyName,@ContactName,@ContactTitle,@Address,@City,@R
egion,@PostalCode,@Country,@Phone,@Fax,@HomePage);
    SELECT SCOPE_IDENTITY() AS SupplierID;
END
```

DeleteSuppliers 存储过程定义如下：

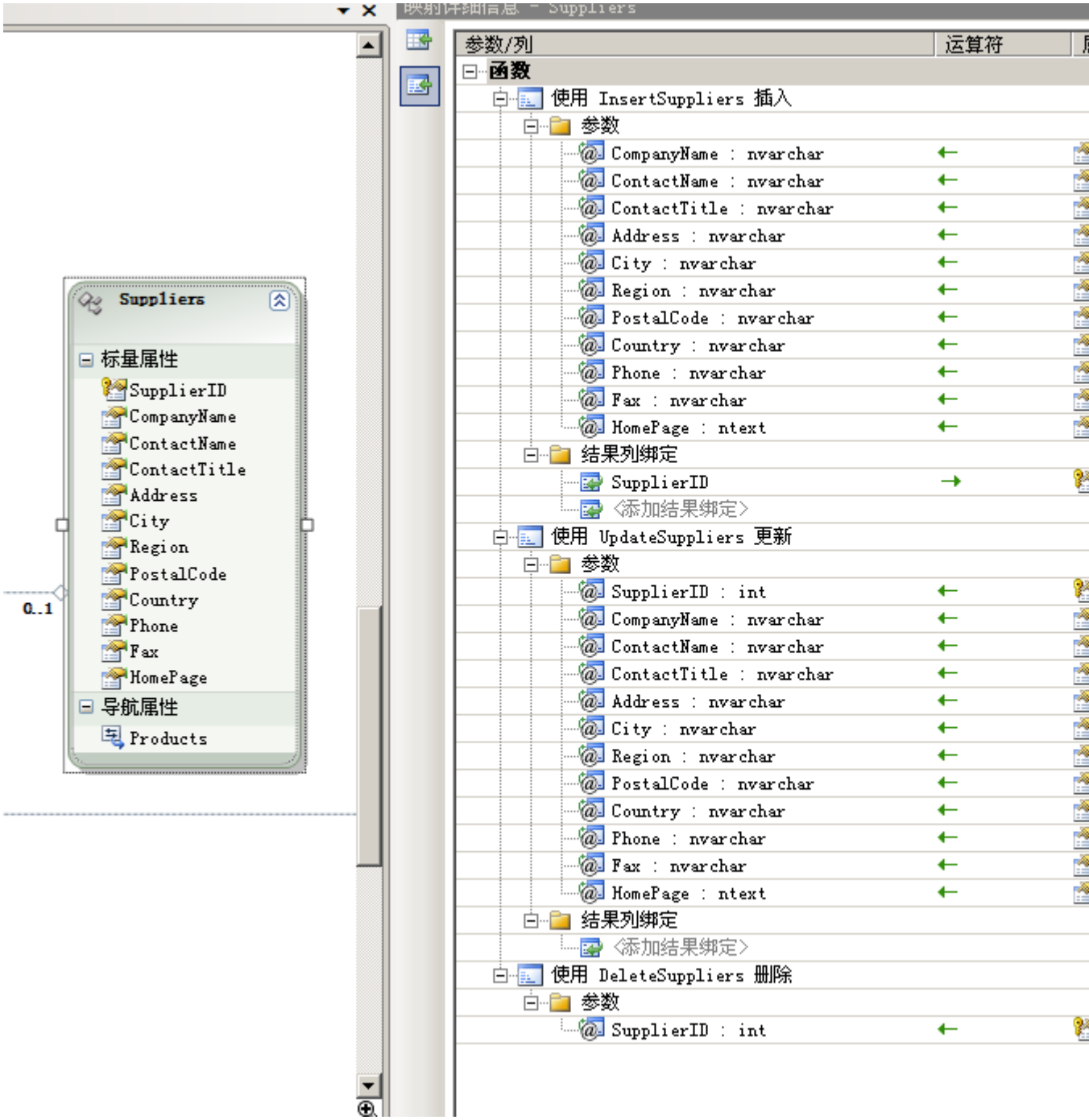
```
CREATE PROCEDURE [dbo].[DeleteSuppliers]
    -- Add the parameters for the stored procedure here
@SupplierID int
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;
    DELETE Suppliers WHERE SupplierID=@SupplierID
END
```

UpdateSuppliers 存储过程定义如下：

```
  CREATE PROCEDURE [dbo].[UpdateSuppliers]
   -- Add the parameters for the stored procedure here
  @SupplierID int,
  @CompanyName nvarchar(40),
  @ContactName nvarchar(30),
  @ContactTitle nvarchar(30),
  @Address nvarchar(60),
  @City nvarchar(15),
```

```
@Region nvarchar(15),
@PostalCode nvarchar(10),
@Country nvarchar(15),
@Phone nvarchar(24),
@Fax nvarchar(24),
@HomePage ntext
AS
BEGIN
 -- SET NOCOUNT ON added to prevent extra result sets from
 -- interfering with SELECT statements.
 SET NOCOUNT ON;
    UPDATE Suppliers SET
CompanyName=@CompanyName,ContactName=@ContactName,ContactTitle=@Co
ntactTitle,Address=@Address,City=@City,Region=@Region,PostalCode=@
PostalCode,Country=@Country,Phone=@Phone,Fax=@Fax,HomePage=@HomePa
ge
 WHERE SupplierID=@SupplierID
END
```

然后，在 VS 可视化设计器中，打开实体模式 edmx 文件，然后在 Suppliers 实体上鼠标右键→“存储过程映射”，然后在在分别设置存储过程即可。如下图所示：



至此，存储过程映射完毕。我们下面来具体的使用，代码如下：

使用存储过程 Insert:

```
[Test]

public void SPInsert()

{

    using (var db = new NorthwindEntities1())
```

```
        {
            var supplier = new Suppliers();
            supplier.CompanyName = "cnblogs";
            supplier.ContactName = "xray2005";
            supplier.ContactTitle = "net";
            supplier.City = "成都";
            supplier.Region = "四川";
            supplier.Country = "中国";
            supplier.PostalCode = "600041";
            supplier.Phone = "028-8001";
            supplier.Fax = "028-8002";
            supplier.HomePage = "http://xray2005.cnblogs.com";
            db.AddToSuppliers(supplier);
            db.SaveChanges();

        }
    }
```

使用存储过程 Update:

```
    [Test]
    public void SPUpdate()
    {
        using (var db = new NorthwindEntities1())
        {
            var supplier = db.Suppliers.FirstOrDefault(s => s.SupplierID == 30);
            Assert.IsNotNull(supplier);
```

```
            supplier.CompanyName = "CNBLOGS";
            supplier.ContactName = "xray2005";
            supplier.ContactTitle = "ぁ风风车.net";
            supplier.City = "成都";
            supplier.Region = "四川";
            supplier.Country = "China";
            supplier.PostalCode = "600040";
            supplier.Phone = "028-1008611";
            supplier.Fax = "028-10086";
            supplier.HomePage = "http://www.cnblogs.com/xray2005";
            db.SaveChanges();

        }
    }
```

使用存储过程 Delete:

```
    [Test]
    public void SPDelete()
    {
        using (var db = new NorthwindEntities1())
        {
            var supplier = db.Suppliers.FirstOrDefault(s => s.SupplierID == 31);
            Assert.IsNotNull(supplier);
            db.DeleteObject(supplier);
            db.SaveChanges();
            var supplier1 = db.Suppliers.FirstOrDefault(s => s.SupplierID == 31);
```

```
            Assert.IsNull(supplier1);
        }
    }
```

至此，实体存储过程映射介绍完毕。本节，内容比较简单。

Entity Framework 学习中级篇 5—使 EF 支持 Oracle9i

从 Code MSDN 上下载下来的 EFOracleProvider 不支持 Oracle9i.但是,目前我所使用的还是 Oracle9i。为此,对 EFOracleProvider 修改了以下，以便使其支持 Oracle9i.

下面说说具体修改地方.（红色部分为添加或修改的代码部分）

一，修改 EFOracleProvider

1,修改 EFOracleProviderManifest.cs 类文件,

```
        internal const string TokenOracle9i = "9i";//add by xray2005
        internal const string TokenOracle10g = "10g";
        internal const string TokenOracle11g = "11g";
```

以下两个地方,不修改也是可以的.但考虑目前我主要是使用 9i,所以也就修改成 9i 了.

```
            private EFOracleVersion _version = EFOracleVersion.Oracle9i;
                //EFOracleVersion.Oracle11g;
        private string _token = TokenOracle9i; //TokenOracle10g;
```

2,修改 EFOracleVersion.cs 类文件，如下代码所示：

```
namespace EFOracleProvider
{
```

```
using System;
/// <summary>
/// This enum describes the current storage version
/// </summary>
internal enum EFOracleVersion
{
     Oracle9i = 9, //add by xray2005
    /// <summary>
    /// Oracle10g
    /// </summary>
    Oracle10g = 10,

    /// <summary>
    /// Oracle 11g
    /// </summary>
    Oracle11g = 11,
    // higher versions go here
}
/// <summary>
/// This class is a simple utility class that determines the version from
the
/// connection
/// </summary>
internal static class EFOracleVersionUtils
{
```

```
        /// <summary>
        /// Get the version from the connection.
        /// </summary>
        /// <param name="connection">current connection</param>
        /// <returns>version for the current connection</returns>
        internal static EFOracleVersion
GetStorageVersion(EFOracleConnection connection)
        {
            string serverVersion = connection.ServerVersion;
            if (serverVersion.StartsWith("9."))
            {
                return EFOracleVersion.Oracle9i; //add by xray2005
            }
            else if (serverVersion.StartsWith("10."))
            {
                return EFOracleVersion.Oracle10g;
            }
            else if (serverVersion.StartsWith("11."))
            {
                return EFOracleVersion.Oracle11g;
            }
            throw new ArgumentException("Could not determine storage
version; " +
                    "a valid storage connection or a version hint is
required.");
        }
```

```
        internal static string GetVersionHint(EFOracleVersion version)
        {
            switch (version)
            {
                case EFOracleVersion.Oracle9i:
                    return EFOracleProviderManifest.TokenOracle9i; //add
by xray2005
                case EFOracleVersion.Oracle10g:
                    return EFOracleProviderManifest.TokenOracle10g;
                case EFOracleVersion.Oracle11g:
                    return EFOracleProviderManifest.TokenOracle11g;
                default:
                    throw new ArgumentException("Could not determine
storage version; " +
                                "a valid storage connection or a version hint
is required.");
            }
        }
        internal static EFOracleVersion GetStorageVersion(string
versionHint)
        {
            if (!string.IsNullOrEmpty(versionHint))
            {
                switch (versionHint)
                {
                    case EFOracleProviderManifest.TokenOracle9i:
```

```
                    return EFOracleVersion.Oracle9i; //add by xray2005
                case EFOracleProviderManifest.TokenOracle10g:
                    return EFOracleVersion.Oracle10g;
                case EFOracleProviderManifest.TokenOracle11g:
                    return EFOracleVersion.Oracle11g;
            }
        }
        throw new ArgumentException("Could not determine storage
version; " +
                "a valid storage connection or a version hint is
required.");
    }

    internal static bool IsVersionX(EFOracleVersion storageVersion)
    {
        return storageVersion == EFOracleVersion.Oracle9i ||
storageVersion == EFOracleVersion.Oracle10g ||
            storageVersion == EFOracleVersion.Oracle11g; //add by
xray2005
    }
  }
}
```

二，使用 EFOracleProvider

修改完毕后,编译一下.如果是自己下载的源代码编译的,那么编译后的 EFOracleProvider 自动已经在 GAC 注册了.如果是手动注册 EFOracleProvider 到 GAC，那么命令如下：

gacutil -I “EFOracleProvider.dll”

其中 gacutil.exe 位于：系统盘符号:\Program Files\Microsoft SDKs\Windows\v6.0A\bin 下面。

接下来，我们需要做的就是,把这个 EFOracleProvider 添加到 Machine.config 中.

第一步,找到 Machine.config 文件.该文件的位置在:

系统盘符号:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG 下面.

第二步,用打开 Machine.config 文件,在 DbProviderFactories 配置节点,增加 EFOracleProvider 的配置,如下所示:

```
<DbProviderFactories>
    <add name="Odbc Data Provider" invariant="System.Data.Odbc" description=".Net Framework Data Provider for Odbc" type="System.Data.Odbc.OdbcFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="OleDb Data Provider" invariant="System.Data.OleDb" description=".Net Framework Data Provider for OleDb" type="System.Data.OleDb.OleDbFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="OracleClient Data Provider" invariant="System.Data.OracleClient" description=".Net Framework Data Provider for Oracle" type="System.Data.OracleClient.OracleClientFactory, System.Data.OracleClient, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    <add name="SqlClient Data Provider" invariant="System.Data.SqlClient" description=".Net Framework Data Provider for SqlServer" type="System.Data.SqlClient.SqlClientFactory, System.Data, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
```

<add name="Microsoft SQL Server Compact Data Provider" invariant="System.Data.SqlServerCe.3.5" description=".NET Framework Data Provider for Microsoft SQL Server Compact" type="System.Data.SqlServerCe.SqlCeProviderFactory, System.Data.SqlServerCe, Version=3.5.0.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91" />

<add name="EF Oracle Data Provider" invariant="EFOracleProvider" description="EF Provider for Oracle" type="EFOracleProvider.EFOracleProviderFactory,EFOracleProvider, Version=1.0.0.0, Culture=neutral, PublicKeyToken=def642f226e0e59b" />

</DbProviderFactories>

第三步,保存即可.

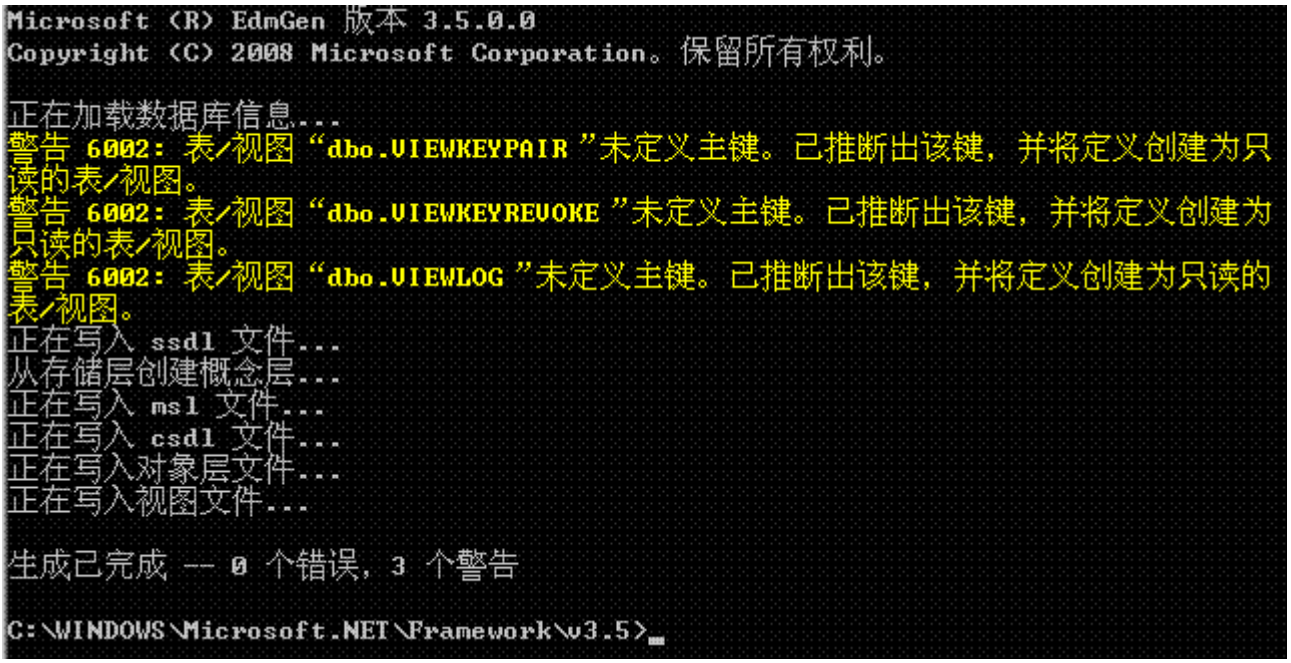
接下来，简单的介绍一下,如何使用这个 EFOralceProvider.

第 1 步:在命令行窗口,将目录定位到提示符，系统盘符:\WINDOWS\Microsoft.NET\Framework\v3.5.如下所示(是我电脑上的目录):

第 2 步,输入相应的生成参数.如下图所示:

将“data source=test;user id=xray;password= 123”成你自己的对应的参数即可.

确定之后,就可以看到生成的结果了,同时会有写信息出来,如下示:



至此,EdmGen 就为我们生成需要的文件.生成的文件如下所示:

- TestEFModel.csdl
- TestEFModel.msl
- TestEFModel.ssdl
- TestEFModel.ObjectLayer.cs
- TestEFModel.Views.cs

然后,通过 EdmGen2 工具, 使用刚刚生成的 TestEFModel .csdl, TestEFModel .msl, TestEFModel .ssdl 三个文件来生成一个模型.

命令如下:

```
Edmgen2.exe /toedmx TestEFModel.csdl TestEFModel.msl TestEFModel.ssdl
```

确定之后,该工具就会为我们生成一个 TestEFModel.edmx 文件了.

然后,把这个文件加入到我们的项目中,同时修改项目的 App.Config 文件连接字符串,如下所示:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <connectionStrings>
```

```
    <add name="NorthwindEFModelContext"
         connectionString="provider=EFOracleProvider;
                           metadata=res://*/TestEFModel.csdl|res://*/TestEFModel.ssdl|res://*/TestEFModel.msl;
                           Provider Connection String='data source=test;user id=xray;password=1111'"
         providerName="System.Data.EntityClient" />
 </connectionStrings>
</configuration>
```

最后保存.

至此，修改 EFOracleProvider 并使用，介绍完毕。

最后提供几个连接，以方便大家学习研究：

- Code.MSDN 上的 EFOracleProvider;
- 经过我修改后，支持 Oracle9i 的 EFOracleProvider.dll
- EdmGen2.exe

Entity Framework 学习高级篇 1—改善 EF 代码的方法（上）

本节，我们将介绍一些改善 EF 代码的相关方法，如 NoTracking, GetObjectByKey, Include 等。

- MergeOption.NoTracking

当我们只需要读取某些数据而不需要删除、更新的时候，可以指定使用 MergeOption.NoTracking 的方式来执行只读查询（EF 默认的方式是 AppendOnly）。当指定使用 NoTracking 来进行只读查询时，与实体相关的引用实体不会被返回，它们会被自动设置为 null。因此，使用 NoTracking 可以提升查询的性能。示例代码如下：

```
        [Test]
        public void NoTrackingTest()
        {
```

```
            using (var db = new NorthwindEntities1())
            {
               //针对 Customers 查询将使用 MergeOption.NoTracking
                db.Customers.MergeOption = MergeOption.NoTracking;
                var cust = db.Customers.Where(c => c.City == "London");
                foreach (var c in cust)
                    Console.WriteLine(c.CustomerID);

                //也可以这样写
                //var cust1 =
((ObjectQuery<Customers>)cust).Execute(MergeOption.NoTracking);

                //Esql 写法
                //string esql = "select value c from customers as c where
c.CustomerID='ALFKI'";
                //db.CreateQuery<Customers>(esql).Execute(MergeOption.No
Tracking).FirstOrDefault();

            }
        }
```

- GetObjectByKey/First

GetObjectByKey：

在 EF 中，使用 GetObjectByKey 方法获取数据时，它首先会查询是否有缓存，如果有缓存则从缓存中返回需要的实体。如果没有则查询数据库，返回需要的实体，并添加在缓存中以便下次使用。

First: 总从数据库中提取需要的实体。

因此，我们应在合适的地方选择 GetObjectByKey 方法来获取数据，以减少对数据库的访问提升性能。示例代码如下：

```
[Test]
 public void GetByKeyTest()
 {
     using (var db = new NorthwindEntities1())
     {
         //从数据库中提取数据
         var cst = db.Customers.First(c => c.CustomerID == "ALFKI");
         Console.WriteLine(cst.CustomerID);

         //将从缓存中提取数据
         EntityKey key = new
EntityKey("NorthwindEntities1.Customers", "CustomerID", "ALFKI");
         var cst1 = db.GetObjectByKey(key) as Customers;
         Console.WriteLine(cst1.CustomerID);


     }
 }
```

此外，需要注意的是如果 GetObjectByKey 没有获取到符合条件的数据，那么它会抛异常。为了避免此情况发生，在有可能出现异常的地方，我们应该使用 TryGetObjectByKey 方法。TryGetObjectByKey 方法获取数据的方式和 GetObjectByKey 类似，只是当没有取到符合条件的数据时，TryGetObjectByKey 会返回 null 而不是抛异常。示例代码如下：

```
        [Test]
        public void TryGetByKeyTest()
        {
            using (var db = new NorthwindEntities1())
            {

                //没有符合条件的数据会有异常抛出
                EntityKey key = new
EntityKey("NorthwindEntities1.Customers", "CustomerID", "风车车.Net");
                var cst = db.GetObjectByKey(key) as Customers;
                Console.WriteLine(cst.CustomerID);

                //没有符合条件的数据会有返回 null
                EntityKey key1 = new
EntityKey("NorthwindEntities1.Customers", "CustomerID", "风车车.Net");
                Object cst1 = null;
                db.TryGetObjectByKey(key1, out cst1);
                if (cst1 != null)
                    Console.WriteLine(((Customers)cst1).CustomerID);

            }
        }
```

- First /FirstOrDefault

First: 当我们使用 First 来获取数据，如果没有符合条件的数据，那么我们的代码将会抛出异常。

FirstOrDefault: 当我们使用 FirstOrDefault 来获取的数据，如果没有符合条件的数据，那么它将返回 null。

显然，对于一个良好的代码，是对可以预见的异常进行处理，而不是等它自己抛出来。示例代码如下：

```
[Test]
public void FirstTest()
{
    using (var db = new NorthwindEntities1())
    {

        //抛异常的代码
        var cst = db.Customers.First(c => c.CustomerID == "风车车.Net");
        Console.WriteLine(cst.CustomerID);//此处将出抛异常

        //推荐的使用如下代码：
        var cst1 = db.Customers.FirstOrDefault(c => c.CustomerID == "风车车.Net");
        if (cst1 != null)
            Console.WriteLine(cst1.CustomerID);
    }
}
```

- 延迟加载/Include

EF 不支持实体的部分属性延迟加载，但它支持实体关系的延迟加载。默认情况，实体的关系是不会加载。如下代码：

```
[Test]
```

```
public void IncludeTest()
{
    using (var db = new NorthwindEntities1())
    {
       var csts = db.Customers;
        foreach (var c in csts)
        {
            Console.WriteLine(c.CustomerID);
            foreach (var o in c.Orders)
                Console.WriteLine("    " + o.OrderID);
        }
    }
}
```

上述代码中，因为 Orders 没有被加载，所以在输出 Orders 的时候，是不会有任何输出的。

当我们需要加载某些关联的关系时，可是用 Include 方法，如下代码所示：

```
[Test]
public void IncludeTest()
{
    using (var db = new NorthwindEntities1())
    {
       var csts = db.Customers.Include("Orders");
        foreach (var c in csts)
        {
            Console.WriteLine(c.CustomerID);
```

```
                foreach (var o in c.Orders)
                    Console.WriteLine("    " + o.OrderID);
            }
        }
    }
```

上述代码中，Customers 关联的 Orders 将被加载。

Entity Framework 学习高级篇 2—改善 EF 代码的方法（下）

本节，我们将介绍一些改善 EF 代码的方法，包括编译查询、存储模型视图以及冲突处理等内容。

- CompiledQuery

提供对查询的编译和缓存以供重新使用。当相同的查询需要执行很多遍的时候，那么我们可以使用 ComplieQuery 将查询的语句进行编译以便下次使用，这样可以免去对同一语句的多次处理，从而改善性能。

示例代码如下：

```
    [Test]
     public void ComplieTest()
     {
         using (var db = new NorthwindEntities1())
         {
             //对查询进行编译
             var customer = CompiledQuery.Compile<NorthwindEntities1,
IQueryable<Customers>>(
                 (database) => database.Customers.Where(c => c.City ==
"London"));
```

```
                //执行 20 次相同的查询
                for (int i = 0; i < 20; i++)
                {
                    DateTime dt = System.DateTime.Now;
                    foreach (var c in customer(db))
                        Console.WriteLine(c.CustomerID);
                    Console.WriteLine(DateTime.Now.Subtract(dt).TotalMil
liseconds);
                    Console.WriteLine("----------------------------------
------------------");
                }

            }
        }
```

- 存储模型视图

在 EF 中，当执行实体查询的时候，运行时首先将实体模型转换成 ESQL 视图，而 ESQL 视图则是根据 msl 文件来生成相应的代码。此外，ESQL 视图包含了相应的查询语句。ESQL 视图被创建后将在应用程序域中进行缓存以便下次使用。这个运行时生成存储模型视图是比较耗时的过程。

为了，免去运行时生成存储模型视图，我们可以预先产生这个的存储模型视图。具体步骤如下：

首先，使用 EdmGen2 来产生存储模型视图，相应的命令如下：

```
Edmgen2 /ViewGen cs NorthwindEntites.edmx
```

执行此命令后，edmgen2 会在当前目录下生成一个名为 NorthwindEntites.GeneratedViews.cs 这个文件，就是我们要使用的存储模型视图文件。

将此文件添加到项目中就行，其他的代码不需要改变，EF 会自动调用此视图文件。如下示例代码：

```
[Test]
public void ViewTest()
{
    using (var db = new NorthwindEntities1())
    {
        var suppliers = db.Suppliers;
        foreach (var s in suppliers)
            Console.WriteLine(s.ContactName);
    }
}
```

没有使用存储模型视图的情况是：

```
1 passed, 0 failed, 0 skipped, took 7.09 seconds.
```

项目中添加了 NorthwindEntites.GeneratedViews.cs 文件，执行情况是：

```
1 passed, 0 failed, 0 skipped, took 5.38 seconds.
```

可见，使用了存储模型视图的确是提高了性能。

- 冲突处理

在 EF 中，默认情况并不会检查并发冲突。因为 EF 实现的是乐观的并发模式，当有并发的冲突发生时，将会抛出 Optimistic Concurrency Exception 异常。我们可以通过使用 RefreshMode 这个枚举来指定当发生冲突时如何处理。

RefreshMode 有两中枚举值：

ClientsWins: 当提交修改，更新数据库中的值。

StoreWins: 放弃修改，使用数据库中的值。

示例代码片段如下：

```
var db2 = new NorthwindEntities1();
```

```
            var customer2 = db2.Customers.FirstOrDefault(c => c.CustomerID
== "2009");
            if (customer2 != null)
            {
                customer2.ContactName = "风车车.Net";
                customer2.City = "CD";
                customer2.Region = "GX";
            }
            try
            {
                db2.SaveChanges();
            }
            catch (OptimisticConcurrencyException ex) //捕获到冲突，则进
行相应的处理
            {
                db2.Refresh(RefreshMode.ClientWins, customer2);
                db2.SaveChanges();
            }
```

上述代码片段，只是说明怎么处理并发冲突，不是具体的并发。（ps：本来是准备开个线程来模拟并发的，但是始终没成功，没明白什么原因，望高人指点呢！）

Entity Framework 学习结束语

前一段时间,一直在学习 EF.对于这个 EF 多少算是有点了解吧. 目前,对于 EF 这个学习系列的文章,就暂时写到这里.当然还有一些相关的知识没有介绍,请各位朋友见谅!很乐意和大家交换关于 EF 学习的心得或技巧.后续,关于 EF 块东西,我还是会随时关注着,如果有什么好的心得或文章,我也会尽量拿出来给大家分享!每次看到一些朋友的相关留言讨论,我都无比的兴奋,有点小小成就感!

最后,感谢各位朋友的热情关注和支持!