LINQ To SQL 语法及实例大全

2011/4/5

目录

LINQ to SQL 语句(1)之 Where

Where 操作

适用场景：实现过滤，查询等功能。

说明：与 SQL 命令中的 Where 作用相似，都是起到范围限定也就是过滤作用的 ，而判断条件就是它后面所接的子句。

Where 操作包括 3 种形式，分别为简单形式、关系条件形式、First()形式。下 面分别用实例举例下：

1.简单形式：

例如：使用 where 筛选在伦敦的客户

```
var q =
    from c in db.Customers
    where c.City == "London"
    select c;
```

再如：筛选 1994 年或之后雇用的雇员：

```
var q =
    from e in db.Employees
    where e.HireDate >= new DateTime(1994, 1, 1)
    select e;
```

2.关系条件形式：

筛选库存量在订货点水平之下但未断货的产品：

```
var q =
    from p in db.Products
    where p.UnitsInStock <= p.ReorderLevel && ! p.Discontinued
    select p;
```

筛选出 UnitPrice 大于 10 或已停产的产品：

```
var q =
    from p in db.Products
    where p.UnitPrice > 10m || p.Discontinued
    select p;
```

下面这个例子是调用两次 where 以筛选出 UnitPrice 大于 10 且已停产的产品。

```
var q =
    db.Products.Where(p=>p.UnitPrice > 10m).Where (p=>p.Discontinued);
```

3.First()形式：

返回集合中的一个元素，其实质就是在 SQL 语句中加 TOP (1)。

简单用法：选择表中的第一个发货方。

```
Shipper shipper = db.Shippers.First();
```

元素：选择 CustomerID 为“BONAP”的单个客户

```
Customer cust = db.Customers.First(c => c.CustomerID == "BONAP");
```

条件：选择运费大于 10.00 的订单：

```
Order ord = db.Orders.First(o => o.Freight > 10.00M);
```

LINQ to SQL 语句(2)之 Select/Distinct

[1] Select 介绍 1

[2] Select 介绍 2

[3] Select 介绍 3 和 Distinct 介绍

Select/Distinct 操作符

适用场景：o(∩_∩) o… 查询呗。

说明：和 SQL 命令中的 select 作用相似但位置不同， 查询表达式中的 select 及所接子句是放在表达式最后并把子句中的变量也就是结 果返回回来；延迟。

Select/Distinct 操作包括 9 种形式，分别为简单用 法、匿名类型形式、条件形式、指定类型形式、筛选形式、整形类型形式、嵌套 类型形式、本地方法调用形式、Distinct 形式。

1.简单用法：

这个示 例返回仅含客户联系人姓名的序列。

```
var q =
    from c in db.Customers
    select c.ContactName;
```

注意：这个语句 只是一个声明或者一个描述，并没有真正把数据取出来，只有当你需要该数据的 时候，它才会执行这个语句，这就是延迟加载(deferred loading)。如果，在声 明的时候就返回的结果集是对象的集合。你可以使用 ToList() 或 ToArray()方法 把查询结果先进行保存，然后再对这个集合进行查询。当然延迟加载(deferred loading)可以像拼接 SQL 语句那样拼接查询语法，再执行它。

2.匿名类型 形式：

说明：匿名类型是 C#3.0 中新特性。其实质是编译器根据我们自定义自 动产生一个匿名的类来帮助我们实现临时变量的储存。匿名类型还依赖于另外一 个特性：支持根据 property 来创建对象。比如，var d = new { Name = "s" };编译器自动产生一个有 property 叫做 Name 的

匿名类，然后按这 个类型分配内存，并初始化对象。但是 var d = new {"s"};是编译不 通过的。因为，编译器不知道匿名类中的 property 的名字。例如 string c = "d";var d = new { c}; 则是可以通过编译的。编译器会创建一个叫 做匿名类带有叫 c 的 property。

例如下例：new {c,ContactName,c.Phone};ContactName 和 Phone 都是在映射文件中定义与表中字 段相对应的 property。编译器读取数据并创建对象时，会创建一个匿名类，这个 类有两个属性，为 ContactName 和 Phone，然后根据数据初始化对象。另外编译器 还可以重命名 property 的名字。

```
var q =
    from c in db.Customers
    select new {c.ContactName, c.Phone};
```

上 面语句描述：使用 SELECT 和匿名类型返回仅含客户联系人姓名和电话号码的序 列

```
var q =
    from e in db.Employees
    select new
    {
        Name = e.FirstName + " " + e.LastName,
        Phone = e.HomePhone
    };
```

上面 语句描述：使用 SELECT 和匿名类型返回仅含雇员姓名和电话号码的序列，并将 FirstName 和 LastName 字段合并为一个字段“Name”，此外在所得的 序列中将 HomePhone 字段重命名为 Phone。

```
var q =
    from p in db.Products
    select new
    {
        p.ProductID,
        HalfPrice = p.UnitPrice / 2
     };
```

上面语句描述：使用 SELECT 和匿名类型返回所有产品的 ID 以及 HalfPrice(设置为产品单价除以 2 所得的值)的序列。

3.条件形式：

说 明：生成 SQL 语句为：case when condition then else。

```
var q =
    from p in db.Products
    select new
    {
        p.ProductName,
        Availability =
          p.UnitsInStock - p.UnitsOnOrder < 0 ?
        "Out Of Stock" : "In Stock"
    };
```

上面语句描述： 使用 SELECT 和条件语句返回产品名称和产品供货状态的序列。

4.指定类 型形式：

说明：该形式返回你自定义类型的对象集。

```
var q =
    from e in db.Employees
    select new Name
     {
        FirstName = e.FirstName,
        LastName = e.LastName
    };
```

上面语句描述：使用 SELECT 和已知类型返回 雇员姓名的序列。

5.筛选形式：

说明：结合 where 使用，起到过滤作 用。

```
var q =
    from c in db.Customers
      where c.City == "London"
    select c.ContactName;
```

上面语句描述：使用 SELECT 和 WHERE 返回仅含伦敦客 户联系人姓名的序列。

6.shaped 形式(整形类型)：

说明：其 select 操 作使用了匿名对象，而这个匿名对象中，其属性也是个匿名对象。

```
var q =
    from c in db.Customers
    select new {
        c.CustomerID,
        CompanyInfo = new {c.CompanyName, c.City, c.Country},
        ContactInfo = new {c.ContactName, c.ContactTitle}
    };
```

语句描述：使用 SELECT 和匿名类型返回有关客户的数据的整形子集。查询顾客的 ID 和公司信息 （公司名称，城市，国家）以及联系信息（联系人和职位）。

7.嵌套类型形式：

说明：返回的对象集中的每个对象 DiscountedProducts 属性中，又 包含一个集合。也就是每个对象也是一个集合类。

```
var q =
    from o in db.Orders
    select new {
        o.OrderID,
        DiscountedProducts =
            from od in o.OrderDetails
            where od.Discount > 0.0
            select od,
        FreeShippingDiscount = o.Freight
    };
```

语句描述：使用嵌套查询返回所有订单及其 OrderID 的序列 、打折订单中项目的子序列以及免送货所省下的金额。

8.本地方法调用 形式(LocalMethodCall)：

这个例子在查询中调用本地方法 PhoneNumberConverter 将电话号码转换为国际格式。

```
var q = from c in db.Customers
            where c.Country == "UK" || c.Country == "USA"
            select new
            {
                c.CustomerID,
                c.CompanyName,
                Phone = c.Phone,
                InternationalPhone =
                PhoneNumberConverter(c.Country, c.Phone)
             };
```

PhoneNumberConverter 方法如下：

```
public string PhoneNumberConverter(string Country, string Phone)
{
    Phone = Phone.Replace(" ", "").Replace (")", ")-");
    switch (Country)
     {
        case "USA":
            return "1- " + Phone;
        case "UK":
            return "44-" + Phone;
        default:
            return Phone;
    }
}
```

下面也是使用了这个方法将电 话号码转换为国际格式并创建 XDocument

```
XDocument doc = new XDocument(
    new XElement("Customers", from c in db.Customers
                where c.Country == "UK" || c.Country == "USA"
                select (new XElement ("Customer",
                        new XAttribute ("CustomerID", c.CustomerID),
                        new XAttribute("CompanyName", c.CompanyName),
                          new XAttribute("InterationalPhone",
                            PhoneNumberConverter(c.Country, c.Phone))
                        ))));
```

9.Distinct 形式：

说明：筛选字段中不相同的值 。用于查询不重复的结果集。生成 SQL 语句为：SELECT DISTINCT

[City] FROM [Customers]

```
var q = (
    from c in db.Customers
    select c.City )
    .Distinct();
```

语句描述：查询顾 客覆盖的国家。

LINQ to SQL 语句(3) 之 Count/Sum/Min/Max/Avg

[1] Count/Sum 讲解

[2] Min 讲解

[3] Max 讲解

[4] Average 和 Aggregate 讲解

Count/Sum/Min/Max/Avg 操作符

适用场景： 统计数据吧，比如统计一些数据的个数，求和，最小值，最大值，平均数。

Count

说明：返回集合中的元素个数，返回 INT 类型；不延迟。生成 SQL 语句为：SELECT COUNT(*) FROM

1.简单形式：

得到数据库中客户 的数量：

```
var q = db.Customers.Count();
```

2.带条件形 式：

得到数据库中未断货产品的数量：

```
var q = db.Products.Count(p => !p.Discontinued);
```

LongCount

说明 ：返回集合中的元素个数，返回 LONG 类型；不延迟。对于元素个数较多的集合可 视情况可以选用 LongCount 来统计元素个数，它返回 long 类型，比较精确。生成 SQL 语句为：SELECT COUNT_BIG(*) FROM

```
var q = db.Customers.LongCount();
```

Sum

说明：返回集合中数值类型元素 之和，集合应为 INT 类型集合；不延迟。生成 SQL 语句为：SELECT SUM(…) FROM

1.简单形式：

得到所有订单的总运费：

```
var q = db.Orders.Select(o => o.Freight).Sum();
```

2.映射形式：

得 到所有产品的订货总数：

```
var q = db.Products.Sum(p => p.UnitsOnOrder);
```

Min
说明：返回集合中元素的最小值；不延迟。 生成 SQL 语句为：SELECT MIN(…) FROM
1.简单形式：
查找任 意产品的最低单价：

```
var q = db.Products.Select(p => p.UnitPrice).Min();
```

2.映射形式：

查找任意订单的最低运费：

```
var q = db.Orders.Min(o => o.Freight);
```

3.元素 ：

查找每个类别中单价最低的产品：

```
var categories =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        CategoryID = g.Key,
        CheapestProducts =
            from p2 in g
            where p2.UnitPrice == g.Min(p3 => p3.UnitPrice)
            select p2
    };
```

Max

说明：返回集合中元素的最大值；不延 迟。生成 SQL 语句为：SELECT MAX(…) FROM

1.简单形式：

查 找任意雇员的最近雇用日期：

```
var q = db.Employees.Select(e => e.HireDate).Max();
```

2.映射形式：

查找任意产品的最大库存量：

```
var q = db.Products.Max(p => p.UnitsInStock);
```

3.元素：

查找每个类别中单价最高的产品：

```
var categories =
    from p in db.Products
     group p by p.CategoryID into g
    select new {
          g.Key,
        MostExpensiveProducts =
            from p2 in g
            where p2.UnitPrice == g.Max(p3 => p3.UnitPrice)
            select p2
    };
```

Average

说明：返回集合中的数值类型元素的平均值。集合应为数字类型集合，其返回 值类型为 double；不延迟。生成 SQL 语句为：SELECT AVG(…) FROM

1.简单形式：

得到所有订单的平均运费：

```
var q = db.Orders.Select(o => o.Freight).Average();
```

2.映射形式：

得到所有产品的平均单价：

```
var q = db.Products.Average(p => p.UnitPrice);
```

3.元素：

查找每个类别中单价高于该类别平 均单价的产品：

```
var categories =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        ExpensiveProducts =
            from p2 in g
```

```
            where p2.UnitPrice > g.Average (p3 => p3.UnitPrice)
            select p2
    };
```

Aggregate

说明：根据输入的表达式获取聚合值；不延迟。即 是说：用一个种子值与当前元素通过指定的函数来进行对比来遍历集合中的元素 ，符合条件的元素保留下来。如果没有指定种子值的话，种子值默认为集合的第 一个元素。

LINQ to SQL 语句(4)之 Join

Join 操作符

适用场景：在我们表关系中有一对一关系，一对多关系， 多对多关系等。对各个表之间的关系，就用这些实现对多个表的操作。

说明：在 Join 操作中，分别为 Join(Join 查询), SelectMany(Select 一对多选择) 和 GroupJoin(分组 Join 查询)。

该扩展方法对两个序列中键匹配的元素进 行 inner join 操作

SelectMany

说明：我们在写查询语句时，如果 被翻译成 SelectMany 需要满足 2 个条件。1：查询语句中没有 join 和 into，2：必 须出现 EntitySet。在我们表关系中有一对一关系，一对多关系，多对多关系等 ，下面分别介绍一下。

1.一对多关系(1 to Many)：

```
var q =
    from c in db.Customers
    from o in c.Orders
      where c.City == "London"
    select o;
```

语句描述：Customers 与 Orders 是一对多关系。即 Orders 在 Customers 类中以 EntitySet 形式出现。所以第二个 from 是从 c.Orders 而不是 db.Orders 里进行筛选 。这个例子在 From 子句中使用外键导航选择伦敦客户的所有订单。

```
var q =
    from p in db.Products
    where p.Supplier.Country == "USA" && p.UnitsInStock == 0
    select p;
```

语句描述：这一句使用了 p.Supplier.Country 条件，间接关联了 Supplier 表。这个例子在 Where 子句中使 用外键导航筛选其供应商在美国且缺货的产品。生成 SQL 语句为：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0]. [SupplierID],
[t0].[CategoryID],[t0].[QuantityPerUnit],[t0]. [UnitPrice],
[t0].[UnitsInStock], [t0].[UnitsOnOrder],[t0]. [ReorderLevel],
[t0].[Discontinued] FROM [dbo].[Products] AS [t0]
LEFT OUTER JOIN [dbo].[Suppliers] AS [t1] ON
[t1]. [SupplierID] = [t0].[SupplierID]
WHERE ([t1].[Country] = @p0) AND ([t0].[UnitsInStock] = @p1)
-- @p0: Input NVarChar (Size = 3; Prec = 0; Scale = 0) [USA]
-- @p1: Input Int (Size = 0; Prec = 0; Scale = 0) [0]
```

2.多对多关系(Many to Many)：

```
var q =
    from e in db.Employees
    from et in e.EmployeeTerritories
    where e.City == "Seattle"
    select new
    {
        e.FirstName,
         e.LastName,
        et.Territory.TerritoryDescription
     };
```

说明：多对多关系一般会涉及三个表(如果有一个表是自关 联的，那有可能只有 2 个表)。这一句语句涉及 Employees, EmployeeTerritories, Territories 三个表。它们的关系是 1：M：1。Employees 和 Territories 没有很明确的关系。

语句描述：这个例子在 From 子句中使 用外键导航筛选在西雅图的雇员，同时列出其所在地区。这条生成 SQL 语句为：

```
SELECT [t0].[FirstName], [t0].[LastName], [t2]. [TerritoryDescription]
FROM [dbo].[Employees] AS [t0] CROSS JOIN [dbo].[EmployeeTerritories]
AS [t1] INNER JOIN [dbo]. [Territories] AS [t2] ON
[t2].[TerritoryID] = [t1].[TerritoryID]
WHERE ([t0].[City] = @p0) AND ([t1].[EmployeeID] = [t0]. [EmployeeID])
-- @p0: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [Seattle]
```

3.自联接关系：

```
var q =
    from e1 in db.Employees
    from e2 in e1.Employees
    where e1.City == e2.City
    select new {
        FirstName1 = e1.FirstName, LastName1 = e1.LastName,
        FirstName2 = e2.FirstName, LastName2 = e2.LastName,
        e1.City
     };
```

语句描述：这个例子在 select 子句中使用外键导航筛选成 对的雇员，每对中一个雇员隶属于另一个雇员，且两个雇员都来自相同城市。生 成 SQL 语句为：

```
SELECT [t0].[FirstName] AS [FirstName1], [t0].[LastName] AS
[LastName1],[t1].[FirstName] AS [FirstName2], [t1].[LastName] AS
[LastName2],[t0].[City] FROM [dbo].[Employees] AS [t0],
[dbo].[Employees] AS [t1] WHERE ([t0].[City] = [t1]. [City]) AND
([t1].[ReportsTo] = [t0].[EmployeeID])
GroupJoin
```

像上面所说的，没有 join 和 into，被翻译成 SelectMany，同时有 join 和 into 时，那么就被翻译为 GroupJoin。在这里 into 的 概念是对其结果进行重新命名。

1.双向联接(Two way join)：

此 示例显式联接两个表并从这两个表投影出结果：

```
var q =
    from c in db.Customers
```

```
    join o in db.Orders on c.CustomerID
    equals o.CustomerID into orders
    select new
     {
         c.ContactName,
         OrderCount = orders.Count ()
    };
```

说明：在一对多关系中，左边是 1，它每条记录 为 c（from c in db.Customers），右边是 Many，其每条记录叫做 o (join o in db.Orders)，每对应左边的一个 c，就会有一组 o，那这一组 o，就叫做 orders， 也就是说，我们把一组 o 命名为 orders，这就是 into 用途。这也就是为什么在 select 语句中，orders 可以调用聚合函数 Count。在 T-SQL 中，使用其内嵌的 T- SQL 返回值作为字段值。如图所示：

生成 SQL 语句为：

```
SELECT [t0].[ContactName], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t1]
      WHERE [t0].[CustomerID] = [t1].[CustomerID]
) AS [OrderCount]
FROM [dbo].[Customers] AS [t0]
```

2.三向联接(There way join)：

此示例显式联接三个表并分别从每个表投影出结果：

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID
    equals o.CustomerID into ords
    join e in db.Employees on c.City
    equals e.City into emps
    select new
    {
         c.ContactName,
         ords = ords.Count(),
         emps = emps.Count()
     };
```

生成 SQL 语句为：

```
SELECT [t0]. [ContactName], (
    SELECT COUNT(*)
    FROM [dbo].[Orders] AS [t1]
    WHERE [t0].[CustomerID] = [t1].[CustomerID]
) AS [ords], (
SELECT COUNT(*)
    FROM [dbo].[Employees] AS [t2]
    WHERE [t0].[City] = [t2].[City]
) AS [emps]
FROM [dbo].[Customers] AS [t0]
```

3.左外部联接(Left Outer Join)：

此示例说明如何通过使用 此示例说明如何通过使用 DefaultIfEmpty() 获取左外部联接。在雇员没有订单时，DefaultIfEmpty()方 法返回 null：

```
var q =
    from e in db.Employees
      join o in db.Orders on e equals o.Employee into ords
    from o in ords.DefaultIfEmpty()
    select new
    {
          e.FirstName,
        e.LastName,
        Order = o
     };
```

说明：以 Employees 左表，Orders 右表，Orders 表中为空时， 用 null 值填充。Join 的结果重命名 ords，使用 DefaultIfEmpty()函数对其再次查 询。其最后的结果中有个 Order，因为 from o in ords.DefaultIfEmpty() 是对 ords 组再一次遍历，所以，最后结果中的 Order 并不是一个集合。但是，如果没 有 from o in ords.DefaultIfEmpty() 这句，最后的 select 语句写成 select new { e.FirstName, e.LastName, Order = ords }的话，那么 Order 就是一个集合。

4.投影的 Let 赋值(Projected let assignment)：

说明：let 语句 是重命名。let 位于第一个 from 和 select 语句之间。

这个例子从联接投影 出最终“Let”表达式：

```
var q =
    from c in db.Customers
    join o in db.Orders on c.CustomerID
      equals o.CustomerID into ords
    let z = c.City + c.Country
    from o in ords
    select new
    {
          c.ContactName,
        o.OrderID,
        z
     };
```

5.组合键(Composite Key)：

这个例子显示带有组合 键的联接：

```
var q =
    from o in db.Orders
    from p in db.Products
    join d in db.OrderDetails
        on new
        {
            o.OrderID,
              p.ProductID
        } equals
            new
             {
                 d.OrderID,
                   d.ProductID
            }
        into details
    from d in details
    select new
    {
        o.OrderID,
        p.ProductID,
        d.UnitPrice
     };
```

说明：使用三个表，并且用匿名类来说明：使用三个表，并 且用匿名类来表示它们之间的关系。它们之间的关系不能用一个键描述清楚，所 以用匿名类，来表示组合键。还有一种是两个表之间是用组合键表示关系的，不 需要使用匿名类。

6.可为 null/不可为 null 的键关系 (Nullable/Nonnullable Key Relationship)：

这个实例显示如何构造一 侧可为 null 而另一侧不可为 null 的联接：

```
var q =
    from o in db.Orders
  join e in db.Employees
      on o.EmployeeID equals
      (int?)e.EmployeeID into emps
   from e in emps
  select new
  {
      o.OrderID,
      e.FirstName
  };
```

LINQ to SQL 语句(5)之 Order By

Order By 操作

适用场景：对查询出的语句进行排序，比如按时间排序 等等。

说明：按指定表达式对集合排序；延迟，：按指定表达式对集合 排序；延迟，默认是升序，加上 descending 表示降序，对应的扩展方法是 OrderBy 和 OrderByDescending

1.简单形式

这个例子使用 orderby 按雇用日期对雇员进行排序：

```
var q =
    from e in db.Employees
    orderby e.HireDate
    select e;
```

说明：默认为升序

2.带条件形式

注意：Where 和 Order By 的顺序并不重要。而在 T-SQL 中，Where 和 Order By 有严格的位置限制 。

```
var q =
    from o in db.Orders
    where o.ShipCity == "London"
    orderby o.Freight
     select o;
```

语句描述：使用 where 和 orderby 按运费进行排序。

3.降序排序

```
var q =
    from p in db.Products
    orderby p.UnitPrice descending
    select p;
```

4.ThenBy

语句描述：使用复合的 orderby 对客户进行 排序，进行排序：

```
var q =
    from c in db.Customers
    orderby c.City, c.ContactName
    select c;
```

说明：按多个表达式进行排序，例如先按 City 排序，当 City 相 同时，按 ContactName 排序。这一句用 Lambda 表达式像这样写：

```
var q =
      .OrderBy(c => c.City)
       .ThenBy(c => c.ContactName).ToList();
```

在 T-SQL 中没有 ThenBy 语句，其依然翻译为 OrderBy，所以也可以用下面语句来表达：

```
var q =
      db.Customers
      .OrderBy(c => c.ContactName)
      .OrderBy(c => c.City).ToList ();
```

所要注意的是，多个 OrderBy 操作时，级连方式是按逆序。对 于降序的，用相应的降序操作符替换即可。

```
var q =
       db.Customers
      .OrderByDescending(c => c.City)
       .ThenByDescending(c => c.ContactName).ToList();
```

需要 说明的是，OrderBy 操作，不支持按 type 排序，也不支持匿名类。比如

```
var q =
      db.Customers
      .OrderBy(c => new
      {
            c.City,
            c.ContactName
       }).ToList();
```

会被抛出异常。错误是前面的操作有匿名类，再 跟 OrderBy 时，比较的是类别。比如

```
var q =
       db.Customers
      .Select(c => new
      {
             c.City,
            c.Address
      })
      .OrderBy(c => c).ToList();
```

如果你想使用 OrderBy(c => c)，其前提条件是 ，前面步骤中，所产生的对象的类别必须为 C#语言的基本类型。比如下句，这里 City 为 string 类型。

```
var q =
    db.Customers
      .Select(c => c.City)
    .OrderBy(c => c).ToList ();
```

5.ThenByDescending

这两个扩展方式都是用在 OrderBy/OrderByDescending 之后的， 第一个 ThenBy/ThenByDescending 扩展方法 作为第二位排序依据，第二个 ThenBy/ThenByDescending 则作为第三位排序依据 ，以此类推

```
var q =
    from o in db.Orders
      where o.EmployeeID == 1
    orderby o.ShipCountry, o.Freight descending
    select o;
```

语句描述：使用 orderby 先按发 往国家再按运费从高到低的顺序对 EmployeeID 1 的订单进行排序。

6. 带 GroupBy 形式

```
var q =
    from p in db.Products
      group p by p.CategoryID into g
    orderby g.Key
    select new {
        g.Key,
        MostExpensiveProducts =
            from p2 in g
          where p2.UnitPrice == g.Max(p3 => p3.UnitPrice)
          select p2
     };
```

语句描述：使用 orderby、Max 和 Group By 得出每种类别 中单价最高的产品，并按

CategoryID 对这组产品进行排序。

LINQ to SQL 语句(6)之 Group By/Having

Group By/Having 操作符

适用场景：分组数据，为我们查找数据缩小 范围。

说明：分配并返回对传入参数进行分组操作后的可枚举对象。分 组；延迟

1.简单形式：

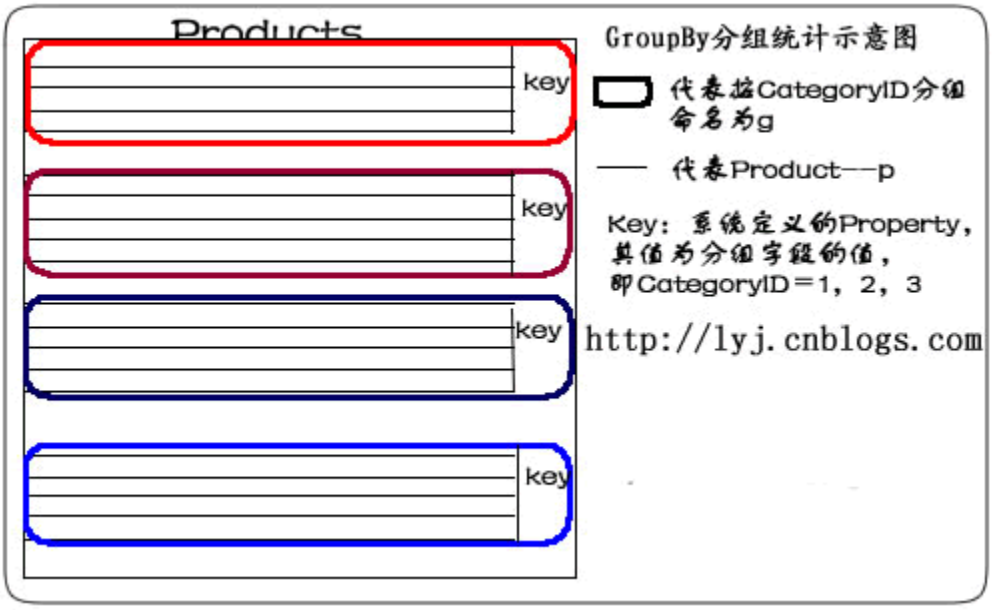
```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select g;
```

语句描述：使用 Group By 按 CategoryID 划分产品。

说 明：from p in db.Products 表示从表中将产品对象取出来。group p by p.CategoryID into g 表示对 p 按 CategoryID 字段归类。其结果命名为 g，一旦重 新命名，p 的作用域就结束了，所以，最后 select 时，只能 select g。当然，也 不必重新命名可以这样写：

```
var q =
    from p in db.Products
    group p by p.CategoryID;
```

我们用示意 图表示：



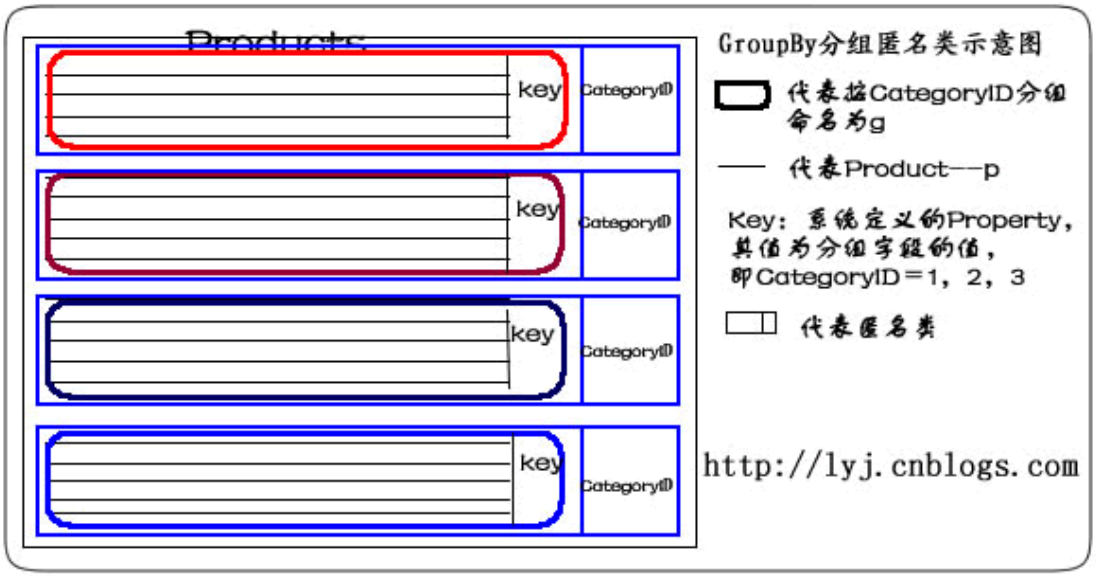
如果想遍历某类别中所有记录，这样：

```
foreach (var gp in q)
{
    if (gp.Key == 2)
    {
        foreach (var item in gp)
        {
            //do something
        }
    }
}
```

2.Select 匿名类 ：

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new { CategoryID = g.Key, g };
```

说明：在这句 LINQ 语句中，有 2 个 property：CategoryID 和 g。这个匿名类，其实质是对返回结果集重新进行了包装。把 g 的 property 封装成一 个完整的分组。如下图所示：



如果想遍历某匿名类中所有记录，要这么做：

```
foreach (var gp in q)
{
    if (gp.CategoryID == 2)
    {
        foreach (var item in gp.g)
        {
            //do something
        }
    }
}
```

3.最大 值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        MaxPrice = g.Max(p => p.UnitPrice)
     };
```

语句描述：使用 Group By 和 Max 查找每个 CategoryID 的最高 单价。

说明：先按 CategoryID 归类，判断各个分类产品中单价最大的 Products。取出 CategoryID

值，并把 UnitPrice 值赋给 MaxPrice。

4.最小 值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        MinPrice = g.Min(p => p.UnitPrice)
     };
```

语句描述：使用 Group By 和 Min 查找每个 CategoryID 的最低 单价。

说明：先按 CategoryID 归类，判断各个分类产品中单价最小的 Products。取出 CategoryID 值，并把 UnitPrice 值赋给 MinPrice。

5.平均 值

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        AveragePrice = g.Average(p => p.UnitPrice)
     };
```

语句描述：使用 Group By 和 Average 得到每个 CategoryID 的 平均单价。

说明：先按 CategoryID 归类，取出 CategoryID 值和各个分类 产品中单价的平均值。

6.求和

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
```

```
    select new {
        g.Key,
        TotalPrice = g.Sum(p => p.UnitPrice)
    };
```

语句描述：使用 Group By 和 Sum 得到 每个 CategoryID 的单价总计。

说明：先按 CategoryID 归类，取出 CategoryID 值和各个分类产品中单价的总和。

7.计数

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        NumProducts = g.Count()
    };
```

语句描述：使用 Group By 和 Count 得到每 个 CategoryID 中产品的数量。

说明：先按 CategoryID 归类，取出 CategoryID 值和各个分类产品的数量。

8.带条件计数

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    select new {
        g.Key,
        NumProducts = g.Count(p => p.Discontinued)
    };
```

语句描述：使 用 Group By 和 Count 得到每个 CategoryID 中断货产品的数量。

说明：先按 CategoryID 归类，取出 CategoryID 值和各个分类产品的断货数量。Count 函数里 ，使用了 Lambda 表达式，Lambda 表达式中的 p，代表这个组里的一个元素或对象 ，即某一个产品。

9.Where 限制

```
var q =
    from p in db.Products
    group p by p.CategoryID into g
    where g.Count() >= 10
    select new {
        g.Key,
        ProductCount = g.Count()
    };
```

语句描述：根据产 品的一ID 分组，查询产品数量大于 10 的 ID 和产品数量。这个示例在 Group By 子句 后使用 Where 子句查找所有至少有 10 种产品的类别。

说明：在翻译成 SQL 语句时，在最外层嵌套了 Where 条件。

10.多列(Multiple Columns)

```
var categories =
    from p in db.Products
    group p by new
    {
        p.CategoryID,
        p.SupplierID
    }
        into g
        select new
            {
                g.Key,
                g
            };
```

语句描述：使用 Group By 按 CategoryID 和 SupplierID 将产品分组。

说明：既按产品的分类，又按供应商分类。在 by 后面，new 出来一个匿名类。这里，Key 其实质是一个类的对象，Key 包含两个 Property：CategoryID、SupplierID。用 g.Key.CategoryID 可以遍历 CategoryID 的值。

11.表达式(Expression)

```
var categories =
      from p in db.Products
    group p by new { Criterion = p.UnitPrice > 10 } into g
    select g;
```

语句描述 ：使用 Group By 返回两个产品序列。第一个序列包含单价大于 10 的产品。第二个 序列包含单价小于或等于 10 的产品。

说明：按产品单价是否大于 10 分类 。其结果分为两类，大于的是一类，小于及等于为另一类。

LINQ to SQL 语句 (7) 之 Exists/In/Any/All/Contains

Exists/In/Any/All/Contains 操作符

适用场景：用于判断集合中元素 ，进一步缩小范围。

Any

说明：用于判断集合中是否有元素满足 某一条件；不延迟。（若条件为空，则集合只要不为空就返回 True，否则为 False）。有 2 种形式，分别为简单形式和带条件形式。

1.简单形式：

仅返回没有订单的客户：

```
var q =
    from c in db.Customers
```

```
    where !c.Orders.Any()
    select c;
```

生成 SQL 语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0]
WHERE NOT (EXISTS(
    SELECT NULL AS [EMPTY] FROM [dbo].[Orders] AS [t1]
    WHERE [t1].[CustomerID] = [t0]. [CustomerID]
   ))
```

2.带条件形式：

仅返回至少有 一种产品断货的类别：

```
var q =
    from c in db.Categories
    where c.Products.Any(p => p.Discontinued)
    select c;
```

生成 SQL 语句为：

```
SELECT [t0]. [CategoryID], [t0].[CategoryName], [t0].[Description],
[t0]. [Picture] FROM [dbo].[Categories] AS [t0]
WHERE EXISTS(
     SELECT NULL AS [EMPTY] FROM [dbo].[Products] AS [t1]
    WHERE ([t1].[Discontinued] = 1) AND
    ([t1].[CategoryID] = [t0]. [CategoryID])
    )
```

All

说明：用于判断集合中所有元素是 否都满足某一条件；不延迟 1.带条件形式

```
var q =
    from c in db.Customers
    where c.Orders.All(o => o.ShipCity == c.City)
    select c;
```

语句描述：这个例子返回所有订单 都运往其所在城市的客户或未下订单的客户。

Contains

说明：用 于判断集合中是否包含有某一元素；不延迟。它是对两个序列进行连接操作的。

```
string[] customerID_Set =
      new string[] { "AROUT", "BOLID", "FISSA" };
var q = (
      from o in db.Orders
      where customerID_Set.Contains(o.CustomerID)
      select o).ToList ();
```

语句描述：查找"AROUT", "BOLID" 和 "FISSA" 这三个客户的订单。先定义了一个数组，在 LINQ to SQL 中 使用 Contains，数组中包含了所有的 CustomerID，即返回结果中，所有的 CustomerID 都在这个集合内。也就是 in。 你也可以把数组的定义放在 LINQ to SQL 语句里。比如：

```
var q = (
      from o in db.Orders
      where (
      new string[] { "AROUT", "BOLID", "FISSA" })
      .Contains (o.CustomerID)
      select o).ToList();
```

Not Contains 则取反：

```
var q = (
      from o in db.Orders
      where !(
        new string[] { "AROUT", "BOLID", "FISSA" })
      .Contains(o.CustomerID)
      select o).ToList();
```

1.包含一个对象：

```
var order = (from o in db.Orders
                  where o.OrderID == 10248
                  select o).First();
var q = db.Customers.Where(p => p.Orders.Contains(order)).ToList();
```

```
foreach (var cust in q)
{
    foreach (var ord in cust.Orders)
    {
        //do something
    }
}
```

语句描述：这个例子 使用 Contain 查找哪个客户包含 OrderID 为 10248 的订单。

2.包含多个值：

```
string[] cities =
    new string[] { "Seattle", "London", "Vancouver", "Paris" };
var q = db.Customers.Where (p=>cities.Contains(p.City)).ToList();
```

语句描述：这个 例子使用 Contains 查找其所在城市为西雅图、伦敦、巴黎或温哥华的客户。

LINQ to SQL 语句 (8) 之 Concat/Union/Intersect/Except

Concat/Union/Intersect/Except 操作

适用场景：对两个集合的处理，例 如追加、合并、取相同项、相交项等等。

Concat（连接）

说明：连接 不同的集合，不会自动过滤相同项；延迟。

1.简单形式：

```
var q = (
            from c in db.Customers
            select c.Phone
          ).Concat(
            from c in db.Customers
            select c.Fax
          ).Concat(
            from e in db.Employees
            select e.HomePhone
            );
```

语句描述：返回所有消费者和雇员的电话和传真。

2.复 合形式：

```
var q = (
            from c in db.Customers
              select new
            {
                  Name = c.CompanyName,
                  c.Phone
            }
            ).Concat(
            from e in db.Employees
            select new
            {
                  Name = e.FirstName + " " + e.LastName,
                  Phone = e.HomePhone
              }
          );
```

语句描述：返回所有消费者和雇员的姓名和电 话。

Union（合并）

说明：连接不同的集合，自动过滤相同项；延迟 。即是将两个集合进行合并操作，过滤相同的项。

```
var q = (
            from c in db.Customers
            select c.Country
          ).Union(
            from e in db.Employees
            select e.Country
          );
```

语句描述：查询顾客和职员所在 的国家。

Intersect（相交）

说明：取相交项；延迟。即是获取不同 集合的相同项（交集）。即先遍历第一个集合，找出所有唯一的元素，然后遍历 第二个集合，并将每个元素与前面找出的元素作对比，返回所有在两个集合内都 出现的元素。

```
var q = (
            from c in db.Customers
            select c.Country
          ).Intersect (
            from e in db.Employees
            select e.Country
          );
```

语句描述：查询顾客和职员同在的国家 。

Except（与非）

说明：排除相交项；延迟。即是从某集合中删除与 另一个集合中相同的项。先遍历第一个集合，找出所有唯一的元素，然后再遍历 第二个集合，返回第二个集合中所有未出现在前面所得元素集合中的元素。

```
var q = (
            from c in db.Customers
              select c.Country
          ).Except(
            from e in db.Employees
            select e.Country
```

```
);
```

语句描述：查询顾客和职员不同的国家。

LINQ to SQL 语句(9)之 Top/Bottom 和 Paging 和 SqlMethods

Top/Bottom 操作

适用场景：适量的取出自己想要的数据，不是全部取出， 这样性能有所加强。

Take

说明：获取集合的前 n 个元素；延迟。即只 返回限定数量的结果集。

```
var q = (
    from e in db.Employees
    orderby e.HireDate
    select e)
     .Take(5);
```

语句描述：选择所雇用的前 5 个雇员。

Skip

说 明：跳过集合的前 n 个元素；延迟。即我们跳过给定的数目返回后面的结果集。

```
var q = (
    from p in db.Products
    orderby p.UnitPrice descending
    select p)
    .Skip (10);
```

语句描述：选择 10 种最贵产品之外的所有产品。

TakeWhile

说明：直到某一条件成立就停止获取；延迟。即用其条件 去依次判断源序列中的元素，返回符合判断条件的元素，该判断操作将在返回 false 或源序列的末尾结束 。

SkipWhile

说明：直到某一条件成立就 停止跳过；延迟。即用其条件去判断源序列中的元素并且跳过第一个符合判断条 件的元素，一旦判断返回 false，接下来将不再进行判断并返回剩下的所有元素 。

Paging（分页）操作

适用场景：结合 Skip 和 Take 就可实现对数据分 页操作。

1.索引

```
var q = (
    from c in db.Customers
    orderby c.ContactName
    select c)
      .Skip(50)
    .Take(10);
```

语句描述：使用 Skip 和 Take 运算 符进行分页，跳过前 50 条记录，然后返回接下来 10 条记录，因此提供显示 Products 表第 6 页的数据。

2.按唯一键排序

```
var q = (
     from p in db.Products
```

```
where p.ProductID > 50
  orderby p.ProductID
select p)
.Take(10);
```

语 句描述：使用 Where 子句和 Take 运算符进行分页，首先筛选得到仅 50 (第 5 页最后 一个 ProductID)以上的 ProductID，然后按 ProductID 排序，最后取前 10 个结果， 因此提供 Products 表第 6 页的数据。请注意，此方法仅适用于按唯一键排序的情 况。

SqlMethods 操作

在 LINQ to SQL 语句中，为我们提供了 SqlMethods 操作，进一步为我们提供了方便，例如 Like 方法用于自定义通配表达 式，Equals 用于相比较是否相等。

Like

自定义的通配表达式。%表示 零长度或任意长度的字符串；_表示一个字符；[]表示在某范围区间的一个字符 ；[^]表示不在某范围区间的一个字符。比如查询消费者 ID 以“C”开 头的消费者。

```
var q = from c in db.Customers
        where SqlMethods.Like(c.CustomerID, "C%")
        select c;
```

比如查询消费者 ID 没有“AXOXT”形式的消费者 ：

```
var q = from c in db.Customers
        where ! SqlMethods.Like(c.CustomerID, "A_O_T")
        select c;
```

DateDiffDay

说明：在两个变量之间比较。分别有：DateDiffDay、 DateDiffHour、DateDiffMillisecond、DateDiffMinute、DateDiffMonth、 DateDiffSecond、DateDiffYear

```
var q = from o in db.Orders
        where SqlMethods
        .DateDiffDay (o.OrderDate, o.ShippedDate) < 10
        select o;
```

语句描述：查询在创建订单后的 10 天内已发货的所有订单。

已编译查 询操作(Compiled Query)

说明：在之前我们没有好的方法对写出的 SQL 语句进 行编辑重新查询，现在我们可以这样做，看下面一个例子：

```
//1. 创建 compiled query
NorthwindDataContext db = new NorthwindDataContext();
var fn = CompiledQuery.Compile(
      (NorthwindDataContext db2, string city) =>
    from c in db2.Customers
    where c.City == city
    select c);
//2.查询城市为 London 的消费者,用 LonCusts 集合表示，这时可以用数据控件 绑定
var LonCusts = fn(db, "London");
//3.查询城市 为 Seattle 的消费者
var SeaCusts = fn(db, "Seattle");
```

语句描述：这个例子创建一个已编译查询， 然后使用它检索输入城市的客户。

LINQ to SQL 语句(10)之 Insert

插入(Insert)1.简单形式

说明：new 一个对象，使用 InsertOnSubmit 方法 将其加入到对应的集合中，使用 SubmitChanges()提交到数据库。

```
NorthwindDataContext db = new NorthwindDataContext();
var newCustomer = new Customer
{
    CustomerID = "MCSFT",
    CompanyName = "Microsoft",
    ContactName = "John Doe",
    ContactTitle = "Sales Manager",
    Address = "1 Microsoft Way",
    City = "Redmond",
```

```
        Region = "WA",
        PostalCode = "98052",
          Country = "USA",
        Phone = "(425) 555- 1234",
        Fax = null
};
db.Customers.InsertOnSubmit(newCustomer);
db.SubmitChanges ();
```

语句描述：使用 InsertOnSubmit 方法将新客户添加到 Customers 表对象。调用 SubmitChanges 将此新 Customer 保存到数据库。

2.一对多 关系

说明：Category 与 Product 是一对多的关系，提交 Category(一端)的数据 时，LINQ to SQL 会自动将 Product(多端)的数据一起提交。

```
var newCategory = new Category
{
        CategoryName = "Widgets",
        Description = "Widgets are the ……"
};
var newProduct = new Product
{
        ProductName = "Blue Widget",
        UnitPrice = 34.56M,
        Category = newCategory
};
db.Categories.InsertOnSubmit(newCategory);
db.SubmitChanges ();
```

语句描述：使用 InsertOnSubmit 方法将新类别添加到 Categories 表中，并将新 Product 对象添加到与此新 Category 有外键关系的 Products 表中。 调用 SubmitChanges 将这些新对象及其关系保存到数据库。

3.多对多关系

说明：在多对多关系中，我们需要依次提交。

```
var newEmployee = new Employee
{
      FirstName = "Kira",
      LastName = "Smith"
};
var newTerritory = new Territory
{
      TerritoryID = "12345",
      TerritoryDescription = "Anytown",
      Region = db.Regions.First()
};
var newEmployeeTerritory = new EmployeeTerritory
{
       Employee = newEmployee,
      Territory = newTerritory
};
db.Employees.InsertOnSubmit(newEmployee);
db.Territories.InsertOnSubmit(newTerritory);
db.EmployeeTerritories.InsertOnSubmit(newEmployeeTerritory);
db.SubmitChanges();
```

语句描述：使用 InsertOnSubmit 方法将新雇 员添加到 Employees 表中，将新 Territory 添加到 Territories 表中，并将新 EmployeeTerritory 对象添加到与此新 Employee 对象和新 Territory 对象有外键关 系的 EmployeeTerritories 表中。调用 SubmitChanges 将这些新对象及其关系保持 到数据库。

4.使用动态 CUD 重写(Override using Dynamic CUD)

说明 ：CUD 就是 Create、Update、Delete 的缩写。下面的例子就是新建一个 ID(主键) 为 32 的 Region，不考虑数据库中有没有 ID 为 32 的数据，如果有则替换原来的数据 ，没有则插入。

```
Region nwRegion = new Region()
{
      RegionID = 32,
      RegionDescription = "Rainy"
};
db.Regions.InsertOnSubmit(nwRegion);
db.SubmitChanges ();
```

语句描述：使用 DataContext 提供的分部方法 InsertRegion 插入一 个区域。对 SubmitChanges 的调用调用 InsertRegion 重写，后者使用动态 CUD 运 行 Linq To SQL 生成的默认 SQL 查询。

LINQ to SQL 语句(11)之 Update

更新(Update)

说明：更新操作，先获取对象，进行修改操作之后，直接调 用 SubmitChanges()方法即可提交。注意，这里是在同一个 DataContext 中，对于 不同的 DataContex 看下面的讲解。

1.简单形式

```
Customer cust =
      db.Customers.First(c => c.CustomerID == "ALFKI");
cust.ContactTitle = "Vice President";
db.SubmitChanges();
```

语句描述：使用 SubmitChanges 将对检索到的一个 Customer 对象做出的更新保持回数据库。

2.多项更改

```
var q = from p in db.Products
              where p.CategoryID == 1
            select p;
foreach (var p in q)
{
      p.UnitPrice += 1.00M;
}
db.SubmitChanges ();
```

语句描述：使用 SubmitChanges 将对检索到的进行的更新保持回数 据库。

LINQ to SQL 语句(12)之 Delete 和使用 Attach

删除(Delete)1.简单形式

说明：调用 DeleteOnSubmit 方法即可。

```
OrderDetail orderDetail =
        db.OrderDetails.First
       (c => c.OrderID == 10255 && c.ProductID == 36);
db.OrderDetails.DeleteOnSubmit (orderDetail);
db.SubmitChanges();
```

语句描述：使用 DeleteOnSubmit 方法从 OrderDetail 表中删除 OrderDetail 对象。调用 SubmitChanges 将此删除保持到数据库。

2.一对多关系

说明：Order 与 OrderDetail 是一对多关系，首先 DeleteOnSubmit 其 OrderDetail(多端)，其次 DeleteOnSubmit 其 Order(一端)。因为一端是主键。

```
var orderDetails =
       from o in db.OrderDetails
       where o.Order.CustomerID == "WARTH" &&
         o.Order.EmployeeID == 3
       select o;
var order =
        (from o in db.Orders
        where o.CustomerID == "WARTH" && o.EmployeeID == 3
        select o).First();
foreach (OrderDetail od in orderDetails)
{
        db.OrderDetails.DeleteOnSubmit(od);
}
db.Orders.DeleteOnSubmit(order);
db.SubmitChanges();
```

语句描述语句描述：使用 DeleteOnSubmit 方法从 Order 和 Order Details 表中删 除 Order 和 Order Detail 对象。首先从 Order Details 删除，然后从 Orders 删除。 调用 SubmitChanges 将

此删除保持到数据库。

3.推理删除(Inferred Delete)

说明：Order 与 OrderDetail 是一对多关系，在上面的例子，我们全部 删除 CustomerID 为 WARTH 和 EmployeeID 为 3 的数据，那么我们不须全部删除呢？ 例如 Order 的 OrderID 为 10248 的 OrderDetail 有很多，但是我们只要删除 ProductID 为 11 的 OrderDetail。这时就用 Remove 方法。

```
Order order = db.Orders.First(x => x.OrderID == 10248);
OrderDetail od =
    order.OrderDetails.First(d => d.ProductID == 11);
order.OrderDetails.Remove(od);
db.SubmitChanges();
```

语 句描述语句描述：这个例子说明在实体对象的引用实体将该对象从其 EntitySet 中移除时，推理删除如何导致在该对象上发生实际的删除操作。仅当实体的关联 映射将 DeleteOnNull 设置为 true 且 CanBeNull 为 false 时，才会发生推理删除行 为。

使用 Attach 更新(Update with Attach)

说明：在对于在不同的 DataContext 之间，使用 Attach 方法来更新数据。例如在一个名为 tempdb 的 NorthwindDataContext 中，查询出 Customer 和 Order，在另一个 NorthwindDataContext 中，Customer 的地址更新为 123 First Ave，Order 的 CustomerID 更新为 CHOPS。

```
//通常，通过从其他层反序列化 XML 来获取要附加的实体
//不支持将实体从一个 DataContext 附加到另一个 DataContext
//因此若要复制反序列化实体的操作，将在此处重新创建这 些实体
Customer c1;
List<Order> deserializedOrders = new List<Order>();
Customer deserializedC1;
using (NorthwindDataContext tempdb = new NorthwindDataContext())
{
    c1 = tempdb.Customers.Single(c => c.CustomerID == "ALFKI");
    deserializedC1 = new Customer
     {
        Address = c1.Address,
        City = c1.City,
```

```
            CompanyName = c1.CompanyName,
            ContactName = c1.ContactName,
            ContactTitle = c1.ContactTitle,
              Country = c1.Country,
            CustomerID = c1.CustomerID,
            Fax = c1.Fax,
            Phone = c1.Phone,
              PostalCode = c1.PostalCode,
            Region = c1.Region
         };
      Customer tempcust =
            tempdb.Customers.Single(c => c.CustomerID == "ANTON");
      foreach (Order o in tempcust.Orders)
      {
            deserializedOrders.Add(new Order
          {
                CustomerID = o.CustomerID,
              EmployeeID = o.EmployeeID,
              Freight = o.Freight,
                OrderDate = o.OrderDate,
              OrderID = o.OrderID,
                RequiredDate = o.RequiredDate,
              ShipAddress = o.ShipAddress,
              ShipCity = o.ShipCity,
                ShipName = o.ShipName,
              ShipCountry = o.ShipCountry,
              ShippedDate = o.ShippedDate,
                ShipPostalCode = o.ShipPostalCode,
              ShipRegion = o.ShipRegion,
              ShipVia = o.ShipVia
           });
      }
}
using (NorthwindDataContext db2 = new NorthwindDataContext())
{
      //将第一个实体附加到当前数据上 下文，以跟踪更改
      //对 Customer 更新，不能写错
        db2.Customers.Attach(deserializedC1);
      //更改所跟踪的实体
      deserializedC1.Address = "123 First Ave";
      // 附加订单列表中的所有实体
      db2.Orders.AttachAll (deserializedOrders);
      //将订单更新为属于其他客户
        foreach (Order o in deserializedOrders)
```

```
    {
        o.CustomerID = "CHOPS";
    }
    //在当前数据上下 文中提交更改
    db2.SubmitChanges();
}
```

语句描述： 从另一个层中获取实体，使用 Attach 和 AttachAll 将反序列化后的实体附加到数据上下文，然后更新实体。更改被提交到数据库。

使用 Attach 更新和删 除(Update and Delete with Attach)

说明：在不同的 DataContext 中，实现 插入、更新、删除。看下面的一个例子：

```
//通常，通过从其他层 反序列化 XML 获取要附加的实体
//此示例使用 LoadWith 在一个查询中预 先加载客户和订单，
//并禁用延迟加载
Customer cust = null;
using (NorthwindDataContext tempdb = new NorthwindDataContext())
{
    DataLoadOptions shape = new DataLoadOptions();
     shape.LoadWith<Customer>(c => c.Orders);
    //加载第一 个客户实体及其订单
    tempdb.LoadOptions = shape;
     tempdb.DeferredLoadingEnabled = false;
    cust = tempdb.Customers.First(x => x.CustomerID == "ALFKI");
}
Order orderA = cust.Orders.First();
Order orderB = cust.Orders.First(x => x.OrderID > orderA.OrderID);
using (NorthwindDataContext db2 = new NorthwindDataContext())
{
     //将第一个实体附加到当前数据上下文，以跟踪更改
     db2.Customers.Attach(cust);
    //附加相关订单以进行跟踪; 否则将 在提交时插入它们
    db2.Orders.AttachAll(cust.Orders.ToList ());
    //更新客户的 Phone.
    cust.Phone = "2345 5436";
    //更新第一个订单 OrderA 的 ShipCity.
     orderA.ShipCity = "Redmond";
    //移除第二个订单 OrderB.
    cust.Orders.Remove(orderB);
    //添加一个新的订 单 Order 到客户 Customer 中.
    Order orderC = new Order() { ShipCity = "New York" };
    cust.Orders.Add (orderC);
```

```
    //提交执行
    db2.SubmitChanges();
}
```

语句描述：从一个上下文提取实体，并使用 Attach 和 AttachAll 附加来自其他上下文的实体，然后更新这两个实体，删除一个实体，添加另一个 实体。更改被提交到数据库。

LINQ to SQL 语句(13)之开放式并发控制和事务

Simultaneous Changes 开放式并发控制

下表介绍 LINQ to SQL 文档 中涉及开放式并发的术语：

术语 说明
并发 两个或更多用户同时尝试更新同一数据库行的情形。
并发冲突 两个或更多用户同时尝试向 一行的一列或多列提交冲突值的情形。
并发控制 用于解决并发冲突的技术。
开放式并发控制 先调查其他事务是否已更改了行中的值，再允许提交更改的技术。相 比之下，保守式并发控制则是通过锁定记录来避免发生并发冲突。之所以称作开 放式控制，是因为它将一个事务干扰另一事务视为不太可能发生。
冲突解决 通过重新查询数据库刷新出现冲突的项 ，然后协调差异的过程。刷新对象时，LINQ to SQL 更改跟踪器会保留以下数据 ：最初从数据库获取并用于更新检查的值 通过后续查询获得的新数据库值。 LINQ to SQL 随后会确定相应对象是否发生冲突（即它的一个或多个成员值是否 已发生更改）。如果此对象发生冲突，LINQ to SQL 下一步会确定它的哪些成员发生冲突。LINQ to SQL 发现的任何成员冲突都会添加到冲突列表中。

在 LINQ to SQL 对象模型中，当以下两个条件都得到满 足时，就会发生“开放式并发冲突”：客户端尝试向数据库提交更改 ；数据库中的一个或多个更新检查值自客户端上次读取它们以来已得到更新。 此冲突的解决过程包括查明对象的哪些成员发生冲突，然后决定您希望如何进行 处理。

开放式并发(Optimistic Concurrency)

说明：这个例子中在 你读取数据之前，另外一个用户已经修改并提交更新了这个数据，所

以不会出现 冲突。

```
//我们打开一个新的连接来模拟另外一个用户
NorthwindDataContext otherUser_db = new NorthwindDataContext();
var otherUser_product =
      otherUser_db.Products.First(p => p.ProductID == 1);
otherUser_product.UnitPrice = 999.99M;
otherUser_db.SubmitChanges();
//我们当前连接
var product = db.Products.First(p => p.ProductID == 1);
product.UnitPrice = 777.77M;
try
{
      db.SubmitChanges();//当前连接执行成 功
}
catch (ChangeConflictException)
{
}
```

说明：我们读取数据之后，另外一个用户获取并提交更新了这个数据，这时，我 们更新这个数据时，引起了一个并发冲突。系统发生回滚，允许你可以从数据库 检索新更新的数据，并决定如何继续进行您自己的更新。

```
//当前 用户
var product = db.Products.First(p => p.ProductID == 1);
//我们打开一个新的连接来模拟另外一个用户
NorthwindDataContext otherUser_db = new NorthwindDataContext() ;
var otherUser_product =
      otherUser_db.Products.First(p => p.ProductID == 1);
otherUser_product.UnitPrice = 999.99M;
otherUser_db.SubmitChanges();
//当前用户修改
product.UnitPrice = 777.77M;
try
{
       db.SubmitChanges();
}
catch (ChangeConflictException)
{
     //发生异常！
}
```

Transactions 事务

LINQ to SQL 支持三种事务模型，分别是：

显式本地事务：调用 SubmitChanges 时，如果 Transaction 属性设置为事务，则在同一事务的上下 文中执行 SubmitChanges 调用。成功执行事务后，要由您来提交或回滚事务。 与事务对应的连接必须与用于构造 DataContext 的连接匹配。如果使用其他连 接，则会引发异常。

显式可分发事务：可以在当前 Transaction 的作用 域中调用 LINQ to SQL API（包括但不限于 SubmitChanges）。LINQ to SQL 检 测到调用是在事务的作用域内，因而不会创建新的事务。在这种情况下， <token>vbtecdlinq</token> 还会避免关闭连接。您可以在此类事 务的上下文中执行查询和 SubmitChanges 操作。

隐式事务：当您调用 SubmitChanges 时，LINQ to SQL 会检查此调用是否在 Transaction 的作用域 内或者 Transaction 属性是否设置为由用户启动的本地事务。如果这两个事务 它均未找到，则 LINQ to SQL 启动本地事务，并使用此事务执行所生成的 SQL 命令。当所有 SQL 命令均已成功执行完毕时，LINQ to SQL 提交本地事务并返 回。

1.Implicit（隐式）

说明：这个例子在执行 SubmitChanges ()操作时，隐式地使用了事务。因为在更新 2 种产品的库存数量时，第二个产品 库存数量为负数了,违反了服务器上的 CHECK 约束。这导致了更新产品全部失败 了，系统回滚到这个操作的初始状态。

```
try
{
      Product prod1 = db.Products.First(p => p.ProductID == 4);
      Product prod2 = db.Products.First(p => p.ProductID == 5);
        prod1.UnitsInStock -= 3;
      prod2.UnitsInStock -= 5;//错误:库存 数量的单位不能是负数
      //要么全部成功要么全部失败
        db.SubmitChanges();
}
catch (System.Data.SqlClient.SqlException e)
{
      //执行异常处理
}
```

2.Explicit（显式）

说明：这个例子使用显式事 务。通过在事务中加入对数据的读取以防止出现开放式并发异

常，显式事务可以 提供更多的保护。如同上一个查询中，更新 prod2 的 UnitsInStock 字段将使 该字段为负值，而这违反了数据库中的 CHECK 约束。这导致更新这两个产品的 事务失败，此时将回滚所有更改。

```
using (TransactionScope ts = new TransactionScope())
{
    try
    {
        Product prod1 = db.Products.First(p => p.ProductID == 4);
        Product prod2 = db.Products.First(p => p.ProductID == 5);
        prod1.UnitsInStock -= 3;
        prod2.UnitsInStock -= 5;//错误:库存数量的单位不能是负数
        db.SubmitChanges();
    }
    catch (System.Data.SqlClient.SqlException e)
    {
        //执行异常处理
    }
}
```

LINQ to SQL 语句(14)之 Null 语义和 DateTime

Null 语义

说明：下面第一个例子说明查询 ReportsToEmployee 为 null 的雇 员。第二个例子使用 Nullable<T>.HasValue 查询雇员，其结果与第一个例 子相同。在第三个例子中，使用 Nullable<T>.Value 来返回 ReportsToEmployee 不为 null 的雇员的 ReportsTo 的值。

1.Null

查找不 隶属于另一个雇员的所有雇员：

```
var q =
    from e in db.Employees
    where e.ReportsToEmployee == null
     select e;
```

2.Nullable<T>.HasValue

查找不隶属于另一个雇员 的所有雇员：

```
var q =
    from e in db.Employees
    where !e.ReportsTo.HasValue
    select e;
```

3.Nullable<T>.Value

返回前者的 EmployeeID 编号。请注意 .Value 为可选：

```
var q =
    from e in db.Employees
    where e.ReportsTo.HasValue
    select new
    {
        e.FirstName,
        e.LastName,
        ReportsTo = e.ReportsTo.Value
    };
```

日期函数

LINQ to SQL 支持以下 DateTime 方法。但是，SQL Server 和 CLR 的 DateTime 类型在范围和计时周期精度 上不同，如下表。

类型 最小值 最大 值 计时周期
System.DateTime 0001 年 1 月 1 日 9999 年 12 月 31 日 100 毫微秒（0.0000001 秒）
T-SQL DateTime 1753 年 1 月 1 日 9999 年 12 月 31 日 3.33… 毫秒（0.0033333 秒）
T-SQL SmallDateTime 1900 年 1 月 1 日 2079 年 6 月 6 日 1 分钟（60 秒）

CLR DateTime 类型与 SQL Server 类型相比，前者范围更 大、精度更高。因此来自 SQL Server 的数据用 CLR 类型表示时，绝不会损失量值 或精度。但如果反过来的话，则范围可能会减小，精度可能会降低；SQL Server 日期不存在 TimeZone 概念，而在 CLR 中支持这个功能。

我们在 LINQ to SQL 查询使用以当地时间、UTC 或固定时间要自己执行转换。

下面用三个 实例说明一下。

1.DateTime.Year

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Year == 1997
     select o;
```

语句描述：这个例子使用 DateTime 的 Year 属性查找 1997 年下的订单。

2.DateTime.Month

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Month == 12
    select o;
```

语句描述：这个例子使用 DateTime 的 Month 属性查找十二月下的订 单。

3.DateTime.Day

```
var q =
    from o in db.Orders
    where o.OrderDate.Value.Day == 31
    select o;
```

语 句描述：这个例子使用 DateTime 的 Day 属性查找某月 31 日下的订单。

LINQ to SQL 语句(15)之 String

字符串（String）

LINQ to SQL 支持以下 String 方法。但是不同的是默认 情况下 System.String 方法区分大小写。而 SQL 则不区分大小写。

1.字符 串串联(String Concatenation)

```
var q =
    from c in db.Customers
    select new
    {
        c.CustomerID,
        Location = c.City + ", " + c.Country
    };
```

语句描述：这个例子使用+运算符在形成经计 算得出的客户 Location 值过程中将字符串字段和字符串串联在一起。

2.String.Length

```
var q =
    from p in db.Products
    where p.ProductName.Length < 10
    select p;
```

语句描述：这个例子使用 Length 属性查找名称短于 10 个字符的所有产品。

3.String.Contains(substring)

```
var q =
    from c in db.Customers
    where c.ContactName.Contains ("Anders")
    select c;
```

语句描述：这个例子使 用 Contains 方法查找所有其联系人姓名中包含“Anders”的客户。

4.String.IndexOf(substring)

```
var q =
    from c in db.Customers
    select new
    {
        c.ContactName,
```

```
        SpacePos = c.ContactName.IndexOf(" ")
    };
```

语句描述：这个例子使用 IndexOf 方法查找每个 客户联系人姓名中出现第一个空格的位置。

5.String.StartsWith (prefix)

```
var q =
    from c in db.Customers
    where c.ContactName.StartsWith("Maria")
    select c;
```

语句描述：这个例子使用 StartsWith 方法查找联系人姓名以 “Maria”开头的客户。

6.String.EndsWith(suffix)

```
var q =
    from c in db.Customers
    where c.ContactName.EndsWith("Anders")
    select c;
```

语句描述：这个例子使用 EndsWith 方法查找联系人姓名以 “Anders”结尾的客户。

7.String.Substring(start)

```
var q =
    from p in db.Products
    select p.ProductName.Substring(3);
```

语句描述：这个例子使用 Substring 方 法返回产品名称中从第四个字母开始的部分。

8.String.Substring (start, length)

```
var q =
    from e in db.Employees
      where e.HomePhone.Substring(6, 3) == "555"
    select e;
```

语句描述：这个例子使用 Substring 方法查找家庭电话号码第七位 到第九位是“555”的雇员。

9.String.ToUpper()

```
var q =
    from e in db.Employees
    select new
    {
        LastName = e.LastName.ToUpper(),
        e.FirstName
    };
```

语句描述：这个例子使用 ToUpper 方法返回姓氏已转换为大 写的雇员姓名。

10.String.ToLower()

```
var q =
    from c in db.Categories
    select c.CategoryName.ToLower();
```

语 句描述：这个例子使用 ToLower 方法返回已转换为小写的类别名称。

11.String.Trim()

```
var q =
    from e in db.Employees
    select e.HomePhone.Substring(0, 5).Trim ();
```

语句描述：这个例子使用 Trim 方法返回雇员家庭电话号码的前五 位，并移除前导和尾随空格。

12.String.Insert(pos, str)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ")"
    select e.HomePhone.Insert(5, ":");
```

语句描述：这个例子使用 Insert 方法返回第五位为) 的雇员电话号码的序列，并在) 后面插入一个 :。

13.String.Remove(start)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ") "
    select e.HomePhone.Remove(9);
```

语句描述：这个 例子使用 Remove 方法返回第五位为) 的雇员电话号码的序列，并移除从第十个 字符开始的所有字符。

14.String.Remove(start, length)

```
var q =
    from e in db.Employees
    where e.HomePhone.Substring(4, 1) == ")"
    select e.HomePhone.Remove(0, 6);
```

语句描述：这个例子使用 Remove 方法返 回第五位为) 的雇员电话号码的序列，并移除前六个字符。

15.String.Replace(find, replace)

```
var q =
    from s in db.Suppliers
    select new
    {
          s.CompanyName,
        Country = s.Country
        .Replace ("UK", "United Kingdom")
        .Replace ("USA", "United States of America")
     };
```

语句描述：这个例子使用 Replace 方法返回 Country 字段中 UK 被替换为 United Kingdom 以及 USA 被替换为 United States of America 的供 应商信息。

LINQ to SQL 语句(16)之对象标识

对象标识

运行库中的对象具有唯一标识。引用同一对象的两个变量实际上 是引用此对象的同一实例。你更改一个变量后，可以通过另一个变量看到这些更 改。

关系数据库表中的行不具有唯一标识。由于每一行都具有唯一的主 键，因此任何两行都不会共用同一键值。

实际上，通常我们是将数据从 数据库中提取出来放入另一层中，应用程序在该层对数据进行处理。这就是 LINQ to SQL 支持的模型。将数据作为行从数据库中提取出来时，你不期望表示 相同数据的两行实际上对应于相同的行实例。如果您查询特定客户两次，您将获 得两行数据。每一行包含相同的信息。

对于对象。你期望在你反复向 DataContext 索取相同的信息时，它实际上会为你提供同一对象实例。你将它们 设计为层次结构或关系图。你希望像检索实物一样检索它们，而不希望仅仅因为 你多次索要同一内容而收到大量的复制实例。

在 LINQ to SQL 中， DataContext 管理对象标识。只要你从数据库中检索新行，该行就会由其主键记 录到标识表中，并且会创建一个新的对象。只要您检索该行，就会将原始对象实 例传递回应用程序。通过这种方式，DataContext 将数据库看到的标识（即主键 ）的概念转换成相应语言看到的标识（即实例）的概念。应用程序只看到处于第 一次检索时的状态的对象。新数据如果不同，则会被丢弃。

LINQ to SQL 使用此方法来管理本地对象的完整性，以支持开放式更新。由于在最初创建对象 后唯一发生的更改是由应用程序做出的，因此应用程序的意向是很明确的。如果 在中间阶段外部某一方做了更改，则在调用 SubmitChanges() 时会识别出这些 更改。

以上来自 MSDN，的确，看了有点“正规”，下面我用 两个例子说明一下。

对象缓存

在第一个示例中，如果我们执行同一查 询两次，则每次都会收到对内存中同一对象的引用。很明显，cust1 和 cust2 是同 一个对象引用。

```
Customer cust1 = db.Customers.First(c => c.CustomerID == "BONAP");
Customer cust2 = db.Customers.First(c => c.CustomerID == "BONAP");
```

下面的示例中，如果您执行返回数据库中同一 行的不同查询，则您每次都会收到对内存中同一对象的引用。cust1 和 cust2 是同 一个对象引用，但是数据库查询了两次。

```
Customer cust1 = db.Customers.First(c => c.CustomerID == "BONAP");
Customer cust2 = (
      from o in db.Orders
      where o.Customer.CustomerID == "BONAP"
      select o )
       .First()
      .Customer;
```

LINQ to SQL 语句(17)之对象加载

对象加载延迟加载

在查询某对象时，实际上你只查询该对象。不会同时自 动获取这个对象。这就是延迟加载。

例如，您可能需要查看客户数据和 订单数据。你最初不一定需要检索与每个客户有关的所有订单数据。其优点是你 可以使用延迟加载将额外信息的检索操作延迟到你确实需要检索它们时再进行。 请看下面的示例：检索出来 CustomerID，就根据这个 ID 查询出 OrderID。

```
var custs =
       from c in db.Customers
       where c.City == "Sao Paulo"
       select c;
//上面 的查询句法不会导致语句立即执行,仅仅是一个描述性的语句,
只有需要的 时候才会执行它
foreach (var cust in custs)
{
       foreach (var ord in cust.Orders)
      {
            //同时查看客 户数据和订单数据
      }
}
```

语句描述：原始查询未请求数 据，在所检索到各个对象的链接中导航如何能导致触发对数据库的新查询。

预先加载：LoadWith 方法

你如果想要同时查询出一些对象的集合的 方法。LINQ to SQL 提供了 DataLoadOptions 用于立即加载对象。方法包括：

LoadWith 方法，用于立即加载与主目标相关的数据。

AssociateWith 方法，用于筛选为特定关系检索到的对象。

使用 LoadWith 方法指定应同时检索与主目标相关的哪些数据。例如，如果你知道你需 要有关客户的订单的信息，则可以使用 LoadWith 来确保在检索客户信息的同时 检索订单信息。使用此方法可仅访问一次数据库，但同时获取两组信息。

在下面的示例中，我们通过设置 DataLoadOptions，来指示 DataContext 在加载 Customers 的同时把对应的 Orders 一起加载，在执行查询时会检索位于 Sao Paulo 的所有 Customers 的所有 Orders。这样一来，连续访问 Customer 对象 的 Orders 属性不会触发新的数据库查询。在执行时生成的 SQL 语句使用了左连 接。

```
NorthwindDataContext db = new NorthwindDataContext ();
DataLoadOptions ds = new DataLoadOptions();
ds.LoadWith<Customer>(p => p.Orders);
db.LoadOptions = ds;
var custs = (
        from c in db2.Customers
        where c.City == "Sao Paulo"
        select c);
foreach (var cust in custs)
{
      foreach (var ord in cust.Orders)
      {
            Console.WriteLine ("CustomerID {0} has an OrderID {1}.",
                  cust.CustomerID,
                ord.OrderID);
      }
}
```

语句描述：在原始查询过程中使用 LoadWith 请求相关数据，以便稍 后在检索到的各个对象中导航时不需要对数据库进行额外的往返。

LINQ to SQL 语句(18)之运算符转换

运算符转换

1.AsEnumerable：将类型转换为泛型 IEnumerable

使用 AsEnumerable<TSource> 可返回类型化为泛型 IEnumerable 的参数。在 此示例中，LINQ to SQL（使用默认泛型 Query）会尝试将查询转换为 SQL 并在服务器上执行。但 where 子句引用用户定义的客户端方法 (isValidProduct)， 此方法无法转换为 SQL。

解决方法是指定 where 的客户端泛型 IEnumerable<T> 实现以替换泛型 IQueryable<T>。可通过调用 AsEnumerable<TSource>运算符来执行此操作。

```
var q =
    from p in db.Products.AsEnumerable()
    where isValidProduct(p)
    select p;
```

语句描述：这个例子就是使 用 AsEnumerable 以便使用 Where 的客户端 IEnumerable 实现，而不是默认的 IQueryable 将在服务器上转换为 SQL 并执行的默认 Query<T>实现。这很有 必要，因为 Where 子句引用了用户定义的客户端方法 isValidProduct，该方法不 能转换为 SQL。

2.ToArray：将序列转换为数组

使用 ToArray <TSource>可从序列创建数组。

```
var q =
    from c in db.Customers
    where c.City == "London"
     select c;
Customer[] qArray = q.ToArray();
```

语句描述：这 个例子使用 ToArray 将查询直接计算为数组。

3.ToList：将序列转换为 泛型列表

使用 ToList<TSource>可从序列创建泛型列表。下面的示例 使用 ToList<TSource>直接将查询的计算结果放入泛型 List<T>。

```
var q =
    from e in db.Employees
    where e.HireDate >= new DateTime(1994, 1, 1)
    select e;
List<Employee> qList = q.ToList();
```

4.ToDictionary：将序 列转化为字典

使用 Enumerable.ToDictionary<TSource, TKey>方法可 以将序列转化为字典。TSource 表示 source 中的元素的类型；TKey 表示 keySelector 返回的键的类型。其返回一个包含键和值的 Dictionary<TKey, TValue>。

```
var q =
    from p in db.Products
    where p.UnitsInStock <= p.ReorderLevel && ! p.Discontinued
    select p;
Dictionary<int, Product> qDictionary =
    q.ToDictionary(p => p.ProductID);
foreach (int key in qDictionary.Keys)
{
     Console.WriteLine(key);
}
```

语句描述：这个例子使用 ToDictionary 将查询和键表达式直接键表达式直接计算为 Dictionary<K, T>。

LINQ to SQL 语句(19)之 ADO.NET 与 LINQ to SQL

ADO.NET 与 LINQ to SQL

它基于由 ADO.NET 提供程序模型提供的服务。因 此，我们可以将 LINQ to SQL 代码与现有的 ADO.NET 应用程序混合在一起，将 当前 ADO.NET 解决方案迁移到 LINQ to SQL。

1.连接

在创建 LINQ to SQL DataContext 时，可以提供现有 ADO.NET 连接。对 DataContext 的所 有操作（包括查询）都使用所提供的这个连接。如果此连接已经打开，则在您使 用完此连接时，LINQ to SQL 会保持它的打开状态不变。我们始终可以访问此连 接，另外还可以使用 Connection 属性自行关闭它。

```
//新建一个 标准的 ADO.NET 连接:
SqlConnection nwindConn = new SqlConnection (connString);
nwindConn.Open();
// ... 其它的 ADO.NET 数据操作 代码... //
//利用现有的 ADO.NET 连接来创建一个 DataContext:
Northwind interop_db = new Northwind(nwindConn);
var orders =
        from o in interop_db.Orders
        where o.Freight > 500.00M
        select o;
//返回 Freight>500.00M 的订单
nwindConn.Close();
```

语句描述：这个例子使用预先存在的 ADO.NET 连接创建 Northwind 对象，本例中的查询返回运费至少为 500.00 的所有订单。

2.事务

当我们已经启动了自己的数据库事务并且我们希望 DataContext 包含在内时，我们可以向 DataContext 提供此事务。

通过 .NET Framework 创建事务的首选方法是使用 TransactionScope 对象。通过使 用此方法，我们可以创建跨数据库及其他驻留在内存中的资源管理器执行的分布 式事务。事务范围几乎不需要资源就可以启动。它们仅在事务范围内存在多个连 接时才将自身提升为分布式事务。

```
using (TransactionScope ts = new TransactionScope())
{
      db.SubmitChanges();
       ts.Complete();
}
```

注意：不能将此方法用于所有数据库。例 如，SqlClient 连接在针对 SQL Server 2000 服务

器使用时无法提升系统事务 。它采取的方法是，只要它发现有使用事务范围的情况，它就会自动向完整的分 布式事务登记。

下面用一个例子说明一下事务的使用方法。在这里，也 说明了重用 ADO.NET 命令和 DataContext 之间的同一连接。

```
var q =
        from p in db.Products
        where p.ProductID == 3
        select p;
//使用 LINQ to SQL 查询出来
//新建一个标准的 ADO.NET 连接:
SqlConnection nwindConn = new SqlConnection(connString);
nwindConn.Open();
//利用现有的 ADO.NET 连接来创建一个 DataContext:
Northwind interop_db = new Northwind(nwindConn);
SqlTransaction nwindTxn = nwindConn.BeginTransaction();
try
{
      SqlCommand cmd = new SqlCommand("UPDATE Products SET"
        +"QuantityPerUnit = 'single item' WHERE ProductID = 3");
      cmd.Connection = nwindConn;
        cmd.Transaction = nwindTxn;
      cmd.ExecuteNonQuery();
        interop_db.Transaction = nwindTxn;
      Product prod1 = interop_db.Products.First(p => p.ProductID == 4);
      Product prod2 = interop_db.Products.First(p => p.ProductID == 5);
        prod1.UnitsInStock -= 3;
      prod2.UnitsInStock -= 5;//这有一个错 误，不能为负数
      interop_db.SubmitChanges();
        nwindTxn.Commit();
}
catch (Exception e)
{
      // 如果有一个错误，所有的操作回滚
      Console.WriteLine (e.Message);
}
nwindConn.Close();
```

语句描述：这个例 子使用预先存在的 ADO.NET 连接创建 Northwind 对象，然后与此对象共享一个 ADO.NET 事务。此事务既用于通过 ADO.NET 连接执行 SQL 命令，又用于通过 Northwind 对象提交更改。当事务因违反 CHECK 约束而中止时，将回滚所有更 改，包括通过 SqlCommand 做出的更改，以及通过 Northwind 对象做出的更改。

LINQ to SQL 语句(20)之存储过程

存储过程

在我们编写程序中，往往需要一些存储过程，在 LINQ to SQL 中 怎么使用呢？也许比原来的更简单些。下面我们以 NORTHWND.MDF 数据库中自带的 几个存储过程来理解一下。

1.标量返回

在数据库中，有名为 Customers Count By Region 的存储过程。该存储过程返回顾客所在 "WA" 区域的数量。

```
ALTER PROCEDURE [dbo]. [NonRowset]
    (@param1 NVARCHAR(15))
AS
BEGIN
     SET NOCOUNT ON;
     DECLARE @count int
     SELECT @count = COUNT(*)FROM Customers
     WHERECustomers.Region = @Param1
     RETURN @count
END
```

我们只要把这个存储过程拖到 O/R 设 计器内，它自动生成了以下代码段：

```
[Function(Name = "dbo.[Customers Count By Region]")]
public int Customers_Count_By_Region([Parameter
(DbType = "NVarChar (15)")] string param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
    ((MethodInfo) (MethodInfo.GetCurrentMethod())), param1);
    return ((int) (result.ReturnValue));
}
```

我们需要时，直接调用就可以了， 例如：

```
int count = db.CustomersCountByRegion ("WA");
Console.WriteLine(count);
```

语句描述：这 个实例使用存储过程返回在“WA”地区的客户数。

2.单一结 果集

从数据库中返回行集合，并包含用于筛选结果的输入参数。 当我们执行 返回行集合的存储过程时，会用到结果类，它存储从存储过程中返回的结果。

下面的示例表示一个存储过程，该存储过程返回客户行并使用输入参数 来仅返回将“London”列为客户城市的那些行的固定几列。

```
ALTER PROCEDURE [dbo].[Customers By City]
        -- Add the parameters for the stored procedure here
        (@param1 NVARCHAR(20))
AS
BEGIN
        -- SET NOCOUNT ON added to prevent extra result sets from
        -- interfering with SELECT statements.
        SET NOCOUNT ON;
        SELECT CustomerID, ContactName, CompanyName, City from
        Customers as c where c.City=@param1
END
```

拖到 O/R 设计器内，它自动生成了以下代码 段：

```
[Function(Name="dbo.[Customers By City]")]
public ISingleResult<Customers_By_CityResult> Customers_By_City(
[Parameter(DbType="NVarChar(20)")] string param1)
{
      IExecuteResult result = this.ExecuteMethodCall(this, (
      (MethodInfo) (MethodInfo.GetCurrentMethod())), param1);
      return ((ISingleResult<Customers_By_CityResult>)
       (result.ReturnValue));
}
```

我们用下面的代码调用：

```
ISingleResult<Customers_By_CityResult> result =
db.Customers_By_City("London");
foreach (Customers_By_CityResult cust in result)
{
        Console.WriteLine("CustID={0}; City={1}", cust.CustomerID,
```

```
            cust.City);
}
```

语句描述：这 个实例使用存储过程返回在伦敦的客户的 CustomerID 和 City。

3.多个可 能形状的单一结果集

当存储过程可以返回多个结果形状时，返回类型无法强 类型化为单个投影形状。尽管 LINQ to SQL 可以生成所有可能的投影类型，但 它无法获知将以何种顺序返回它们。ResultTypeAttribute 属性适用于返回多 个结果类型的存储过程，用以指定该过程可以返回的类型的集合。

在下 面的 SQL 代码示例中，结果形状取决于输入（param1 = 1 或 param1 = 2）。我 们不知道先返回哪个投影。

```
ALTER PROCEDURE [dbo]. [SingleRowset_MultiShape]
        -- Add the parameters for the stored procedure here
        (@param1 int )
AS
BEGIN
        -- SET NOCOUNT ON added to prevent extra result sets from
        -- interfering with SELECT statements.
        SET NOCOUNT ON;
        if(@param1 = 1)
        SELECT * from Customers as c where c.Region = 'WA'
        else if (@param1 = 2)
          SELECT CustomerID, ContactName, CompanyName from
        Customers as c where c.Region = 'WA'
END
```

拖到 O/R 设计器内，它自动生成了以下代码段：

```
[Function (Name="dbo.[Whole Or Partial Customers Set]")]
public ISingleResult<Whole_Or_Partial_Customers_SetResult>
Whole_Or_Partial_Customers_Set([Parameter(DbType="Int")]
System.Nullable<int> param1)
{
        IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
        return ((ISingleResult<Whole_Or_Partial_Customers_SetResult>)
      (result.ReturnValue));
}
```

但是，VS2008 会把多结果集 存储过程识别为单结果集的存储过程，默认生成的代码我们要手动修改一下，要 求返回多个结果集，像这样：

```
[Function(Name="dbo.[Whole Or Partial Customers Set]")]
[ResultType(typeof (WholeCustomersSetResult))]
[ResultType(typeof (PartialCustomersSetResult))]
public IMultipleResults Whole_Or_Partial_Customers_Set([Parameter
(DbType="Int")] System.Nullable<int> param1)
{
    IExecuteResult result = this.ExecuteMethodCall(this,
     ((MethodInfo)(MethodInfo.GetCurrentMethod())), param1);
     return ((IMultipleResults)(result.ReturnValue));
}
```

我们 分别定义了两个分部类，用于指定返回的类型。WholeCustomersSetResult 类 如 下：

```
public partial class WholeCustomersSetResult
{
    private string _CustomerID;
    private string _CompanyName;
    private string _ContactName;
    private string _ContactTitle;
    private string _Address;
     private string _City;
    private string _Region;
     private string _PostalCode;
    private string _Country;
     private string _Phone;
    private string _Fax;
    public WholeCustomersSetResult()
    {
    }
    [Column (Storage = "_CustomerID", DbType = "NChar(5)")]
    public string CustomerID
    {
        get { return this._CustomerID; }
        set
        {
            if ((this._CustomerID != value))
                this._CustomerID = value;
        }
    }
    [Column(Storage = "_CompanyName", DbType = "NVarChar(40)")]
```

```
  public string CompanyName
{
        get { return this._CompanyName; }
        set
        {
                if ((this._CompanyName != value))
                        this._CompanyName = value;
        }
}
[Column (Storage = "_ContactName", DbType = "NVarChar(30) ")]
public string ContactName
{
         get { return this._ContactName; }
        set
        {
               if ((this._ContactName != value))
                        this._ContactName = value;
        }
}
[Column (Storage = "_ContactTitle", DbType = "NVarChar(30) ")]
public string ContactTitle
{
         get { return this._ContactTitle; }
        set
        {
               if ((this._ContactTitle != value))
                        this._ContactTitle = value;
        }
}
  [Column(Storage = "_Address", DbType = "NVarChar(60) ")]
public string Address
{
        get { return this._Address; }
        set
        {
                if ((this._Address != value))
                    this._Address = value;
        }
}
[Column(Storage = "_City", DbType = "NVarChar(15)")]
public string City
{
        get { return this._City; }
          set
```

```
        {
            if ((this._City != value))
                this._City = value;
        }
    }
    [Column(Storage = "_Region", DbType = "NVarChar (15)")]
    public string Region
    {
        get { return this._Region; }
        set
        {
             if ((this._Region != value))
                this._Region = value;
        }
    }
    [Column(Storage = "_PostalCode", DbType = "NVarChar(10)")]
      public string PostalCode
    {
        get { return this._PostalCode; }
        set
        {
             if ((this._PostalCode != value))
                this._PostalCode = value;
        }
    }
    [Column(Storage = "_Country", DbType = "NVarChar(15)")]
      public string Country
    {
        get { return this._Country; }
        set
        {
            if ((this._Country != value))
                this._Country = value;
        }
    }
    [Column(Storage = "_Phone", DbType = "NVarChar(24)")]
      public string Phone
    {
        get { return this._Phone; }
        set
        {
            if ((this._Phone != value))
                this._Phone = value;
        }
```

```
        }
        [Column(Storage = "_Fax", DbType = "NVarChar(24)")]
        public string Fax
        {
            get { return this._Fax; }
            set
            {
                if ((this._Fax != value))
                    this._Fax = value;
            }
        }
}
```

PartialCustomersSetResult 类 如下：

```
public partial class PartialCustomersSetResult
{
        private string _CustomerID;
        private string _ContactName;
        private string _CompanyName;
        public PartialCustomersSetResult()
        {
        }
        [Column (Storage = "_CustomerID", DbType = "NChar(5)")]
        public string CustomerID
        {
            get { return this._CustomerID; }
            set
            {
                if ((this._CustomerID != value))
                    this._CustomerID = value;
            }
        }
        [Column(Storage = "_ContactName", DbType = "NVarChar(30)")]
          public string ContactName
        {
            get { return this._ContactName; }
            set
            {
                if ((this._ContactName != value))
                    this._ContactName = value;
            }
        }
        [Column (Storage = "_CompanyName", DbType = "NVarChar(40) ")]
```

```
        public string CompanyName
        {
                get { return this._CompanyName; }
            set
            {
                    if ((this._CompanyName != value))
                            this._CompanyName = value;
            }
        }
}
```

这样就可以使用了，下面代码直接调用，分别返回各自的结果集 合。

```
//返回全部 Customer 结果集
IMultipleResults result = db.Whole_Or_Partial_Customers_Set(1);
IEnumerable<WholeCustomersSetResult> shape1 =
result.GetResult<WholeCustomersSetResult>();
foreach (WholeCustomersSetResult compName in shape1)
{
        Console.WriteLine(compName.CompanyName);
}
//返回部分 Customer 结果集
result = db.Whole_Or_Partial_Customers_Set(2);
IEnumerable<PartialCustomersSetResult> shape2 =
result.GetResult<PartialCustomersSetResult>();
foreach (PartialCustomersSetResult con in shape2)
{
        Console.WriteLine(con.ContactName);
}
```

语句描述：这个实例 使用存储过程返回“WA”地区中的一组客户。返回的结果集形状取决于传入的参数。如果参数等于 1，则返回所有客户属性。如果参数等于 2，则返 回 ContactName 属性。

4.多个结果集

这种存储过程可以生成多个结果 形状，但我们已经知道结果的返回顺序。

下面是一个按顺序返回多个结 果集的存储过程 Get Customer And Orders。 返回顾客 ID 为 "SEVES" 的顾客和他们所有的订单。

```
ALTER PROCEDURE [dbo].[Get Customer And Orders]
```

```
(@CustomerID nchar(5))
      -- Add the parameters for the stored procedure here
AS
BEGIN
       -- SET NOCOUNT ON added to prevent extra result sets from
      -- interfering with SELECT statements.
      SET NOCOUNT ON;
       SELECT * FROM Customers AS c WHERE c.CustomerID = @CustomerID
      SELECT * FROM Orders AS o WHERE o.CustomerID = @CustomerID
END
```

拖到设计器代码如下：

```
[Function (Name="dbo.[Get Customer And Orders]")]
public ISingleResult<Get_Customer_And_OrdersResult>
Get_Customer_And_Orders([Parameter(Name="CustomerID",
DbType="NChar(5)")] string customerID)
{
       IExecuteResult result = this.ExecuteMethodCall(this,
       ((MethodInfo)(MethodInfo.GetCurrentMethod())), customerID);
       return ((ISingleResult<Get_Customer_And_OrdersResult>)
       (result.ReturnValue));
}
```

同样，我们要修改自动生成的代码 ：

```
[Function(Name="dbo.[Get Customer And Orders] ")]
[ResultType(typeof(CustomerResultSet))]
[ResultType(typeof(OrdersResultSet))]
public IMultipleResults Get_Customer_And_Orders
([Parameter (Name="CustomerID",DbType="NChar(5)")]
string customerID)
{
      IExecuteResult result = this.ExecuteMethodCall(this,
      ((MethodInfo) (MethodInfo.GetCurrentMethod())), customerID);
      return ((IMultipleResults)(result.ReturnValue));
}
```

同样，自己手 写类，让其存储过程返回各自的结果集。

CustomerResultSet 类

```
public partial class CustomerResultSet
{
```

```
private string _CustomerID;
private string _CompanyName;
private string _ContactName;
private string _ContactTitle;
private string _Address;
  private string _City;
private string _Region;
  private string _PostalCode;
private string _Country;
  private string _Phone;
private string _Fax;
public CustomerResultSet()
{
}
[Column(Storage = "_CustomerID", DbType = "NChar(5)")]
  public string CustomerID
{
      get { return this._CustomerID; }
      set
      {
            if ((this._CustomerID != value))
                this._CustomerID = value;
      }
}
[Column(Storage = "_CompanyName", DbType = "NVarChar(40)")]
  public string CompanyName
{
      get { return this._CompanyName; }
      set
      {
            if ((this._CompanyName != value))
                  this._CompanyName = value;
      }
}
[Column (Storage = "_ContactName", DbType = "NVarChar(30) ")]
public string ContactName
{
        get { return this._ContactName; }
      set
      {
           if ((this._ContactName != value))
                  this._ContactName = value;
      }
}
```

```
[Column (Storage = "_ContactTitle", DbType = "NVarChar(30) ")]
public string ContactTitle
{
        get { return this._ContactTitle; }
      set
      {
            if ((this._ContactTitle != value))
                  this._ContactTitle = value;
      }
}
 [Column(Storage = "_Address", DbType = "NVarChar(60) ")]
public string Address
{
      get { return this._Address; }
      set
      {
           if ((this._Address != value))
               this._Address = value;
      }
}
[Column(Storage = "_City", DbType = "NVarChar(15)")]
public string City
{
      get { return this._City; }
       set
      {
            if ((this._City != value))
                  this._City = value;
      }
}
[Column(Storage = "_Region", DbType = "NVarChar (15)")]
public string Region
{
      get { return this._Region; }
      set
      {
           if ((this._Region != value))
               this._Region = value;
      }
}
[Column(Storage = "_PostalCode", DbType = "NVarChar(10)")]
 public string PostalCode
{
      get { return this._PostalCode; }
```

```
            set
            {
                if ((this._PostalCode != value))
                    this._PostalCode = value;
            }
        }
        [Column(Storage = "_Country", DbType = "NVarChar(15)")]
         public string Country
        {
            get { return this._Country; }
            set
            {
                if ((this._Country != value))
                    this._Country = value;
            }
        }
        [Column(Storage = "_Phone", DbType = "NVarChar(24)")]
         public string Phone
        {
            get { return this._Phone; }
            set
            {
                if ((this._Phone != value))
                    this._Phone = value;
            }
        }
        [Column(Storage = "_Fax", DbType = "NVarChar(24)")]
        public string Fax
        {
            get { return this._Fax; }
            set
            {
                if ((this._Fax != value))
                    this._Fax = value;
            }
        }
}
```

OrdersResultSet 类

```
public partial class OrdersResultSet
{
        private System.Nullable<int> _OrderID;
       private string _CustomerID;
```

```
private System.Nullable<int> _EmployeeID;
private System.Nullable<System.DateTime> _OrderDate;
private System.Nullable<System.DateTime> _RequiredDate;
private System.Nullable<System.DateTime> _ShippedDate;
private System.Nullable<int> _ShipVia;
private System.Nullable<decimal> _Freight;
  private string _ShipName;
private string _ShipAddress;
private string _ShipCity;
private string _ShipRegion;
private string _ShipPostalCode;
private string _ShipCountry;
public OrdersResultSet()
{
}
[Column(Storage = "_OrderID", DbType = "Int")]
public System.Nullable<int> OrderID
{
      get { return this._OrderID; }
        set
      {
            if ((this._OrderID != value))
                  this._OrderID = value;
      }
}
[Column(Storage = "_CustomerID", DbType = "NChar(5)")]
public string CustomerID
{
      get { return this._CustomerID; }
      set
        {
            if ((this._CustomerID != value))
                  this._CustomerID = value;
      }
}
  [Column(Storage = "_EmployeeID", DbType = "Int")]
public System.Nullable<int> EmployeeID
{
      get { return this._EmployeeID; }
      set
        {
            if ((this._EmployeeID != value))
                  this._EmployeeID = value;
      }
```

```
}
  [Column(Storage = "_OrderDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> OrderDate
{
        get { return this._OrderDate; }
      set
      {
            if ((this._OrderDate != value))
                  this._OrderDate = value;
      }
}
[Column (Storage = "_RequiredDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> RequiredDate
{
      get { return this._RequiredDate; }
      set
      {
            if ((this._RequiredDate != value))
                  this._RequiredDate = value;
      }
}
  [Column(Storage = "_ShippedDate", DbType = "DateTime")]
public System.Nullable<System.DateTime> ShippedDate
{
        get { return this._ShippedDate; }
      set
        {
            if ((this._ShippedDate != value))
                  this._ShippedDate = value;
      }
}
  [Column(Storage = "_ShipVia", DbType = "Int")]
public System.Nullable<int> ShipVia
{
        get { return this._ShipVia; }
      set
      {
            if ((this._ShipVia != value))
                  this._ShipVia = value;
      }
}
[Column (Storage = "_Freight", DbType = "Money")]
  public System.Nullable<decimal> Freight
{
```

```
        get { return this._Freight; }
        set
        {
            if ((this._Freight != value))
                this._Freight = value;
        }
}
[Column (Storage = "_ShipName", DbType = "NVarChar(40)")]
public string ShipName
{
        get { return this._ShipName; }
        set
        {
            if ((this._ShipName != value))
                this._ShipName = value;
        }
}
[Column(Storage = "_ShipAddress", DbType = "NVarChar(60)")]
  public string ShipAddress
{
        get { return this._ShipAddress; }
        set
        {
            if ((this._ShipAddress != value))
                this._ShipAddress = value;
        }
}
[Column (Storage = "_ShipCity", DbType = "NVarChar(15)")]
public string ShipCity
{
        get { return this._ShipCity; }
        set
        {
            if ((this._ShipCity != value))
                this._ShipCity = value;
        }
}
[Column(Storage = "_ShipRegion", DbType = "NVarChar(15)")]
  public string ShipRegion
{
        get { return this._ShipRegion; }
        set
        {
            if ((this._ShipRegion != value))
```

```
                    this._ShipRegion = value;
            }
        }
        [Column(Storage = "_ShipPostalCode", DbType = "NVarChar(10)")]
        public string ShipPostalCode
        {
            get { return this._ShipPostalCode; }
            set
            {
                if ((this._ShipPostalCode != value))
                    this._ShipPostalCode = value;
            }
        }
         [Column(Storage = "_ShipCountry", DbType = "NVarChar (15)")]
        public string ShipCountry
        {
             get { return this._ShipCountry; }
            set
            {
                if ((this._ShipCountry != value))
                    this._ShipCountry = value;
            }
        }
}
```

这时，只要调用就可以了。

```
IMultipleResults result = db.Get_Customer_And_Orders("SEVES");
//返回 Customer 结果集
IEnumerable<CustomerResultSet> customer =
result.GetResult<CustomerResultSet>();
//返回 Orders 结果集
IEnumerable<OrdersResultSet> orders =
result.GetResult<OrdersResultSet>();
//在这里，我们读取 CustomerResultSet 中的数据
foreach (CustomerResultSet cust in customer)
{
    Console.WriteLine(cust.CustomerID);
}
```

语句描述：这个实例使用存储过程返回客户“SEVES”及 其所有订单。

5.带输出参数

LINQ to SQL 将输出参数映射到引用参数 ，并且对于值类型，它将参数声明为可以为 null。

下面的示例带有单 个输入参数（客户 ID）并返回一个输出参数（该客户的总销售额）。

```
ALTER PROCEDURE [dbo].[CustOrderTotal]
@CustomerID nchar(5),
@TotalSales money OUTPUT
AS
SELECT @TotalSales = SUM(OD.UNITPRICE*(1-OD.DISCOUNT) * OD.QUANTITY)
FROM ORDERS O, "ORDER DETAILS" OD
where O.CUSTOMERID = @CustomerID AND O.ORDERID = OD.ORDERID
```

把这个存储过程拖到设 计器中，图片如下：

其生成代码如下：

```
[Function (Name="dbo.CustOrderTotal")]
public int CustOrderTotal (
[Parameter(Name="CustomerID", DbType="NChar(5) ")]string customerID,
[Parameter (Name="TotalSales", DbType="Money")]
   ref System.Nullable<decimal> totalSales)
{
        IExecuteResult result = this.ExecuteMethodCall(this,
        ((MethodInfo)(MethodInfo.GetCurrentMethod())),
      customerID, totalSales);
      totalSales = ((System.Nullable<decimal>)
      (result.GetParameterValue(1)));
      return ((int) (result.ReturnValue));
}
```

我们使用下面的语句调用此存储过 程：注意：输出参数是按引用传递的，以支持参数为“in/out”的方 案。在这种情况下，参数仅为“out”。

```
decimal? totalSales = 0;
string customerID = "ALFKI";
db.CustOrderTotal(customerID, ref totalSales);
Console.WriteLine("Total Sales for Customer '{0}' = {1:C}",
customerID, totalSales);
```

语句描述：这个实例 使用返回 Out 参数的存储过程。

好了，就说到这里了，其增删改操作同 理。相信大家通过这 5 个实例理解了存储过程。

LINQ to SQL 语句(21)之用户定义函数

用户定义函数

我们可以在 LINQ to SQL 中使用用户定义函数。只要把 用户定义函数拖到 O/R 设计器中，LINQ to SQL 自动使用 FunctionAttribute 属性 和 ParameterAttribute 属性（如果需要）将其函数指定为方法。这时，我们只需 简单调用即可。

在这里注意：使用用户定义函数的时候必须满足以下形 式之一，否则会出现 InvalidOperationException 异常情况。

具有正确映 射属性的方法调用的函数。这里使用 FunctionAttribute 属性和 ParameterAttribute 属性。

特定于 LINQ to SQL 的静态 SQL 方法。

.NET Framework 方法支持的函数。

下面介绍几个例子：

1.在 Select 中使用用户定义的标量函数

所谓标量函数是指返回在 RETURNS 子句中定义的类型的单个数据值。可以使用所有标量数据类型，包括 bigint 和 sql_variant。不支持 timestamp 数据类型、用户定义数据类型和非标量类型（如 table 或 cursor）。在 BEGIN...END 块中定义的函数主体包含 返回该值的 Transact-SQL 语句系列。返回类型可以是除 text、ntext、image 、cursor 和 timestamp 之外的任何数据类型。 我们在系统自带的 NORTHWND.MDF 数据库中，有 3 个自定义函数，这里使用 TotalProductUnitPriceByCategory，其代码如下：

```
ALTER FUNCTION [dbo].[TotalProductUnitPriceByCategory]
(@categoryID int)
RETURNS Money
AS
BEGIN
    -- Declare the return variable here
```

```
    DECLARE @ResultVar Money
    -- Add the T-SQL statements to compute the return value here
    SELECT @ResultVar = (Select SUM(UnitPrice)
                         from Products
                         where CategoryID = @categoryID)
    -- Return the result of the function
    RETURN @ResultVar
END
```

我们将其拖到设计器中，LINQ to SQL 通过 使用 FunctionAttribute 属性将类中定义的客户端方法映射到用户定义的函数 。请注意，这个方法体会构造一个捕获方法调用意向的表达式，并将该表达式传 递给 DataContext 进行转换和执行。

```
[Function (Name="dbo.TotalProductUnitPriceByCategory",
IsComposable=true)]
public System.Nullable<decimal> TotalProductUnitPriceByCategory(
[Parameter (DbType="Int")] System.Nullable<int> categoryID)
{
    return ((System.Nullable<decimal>) (this.ExecuteMethodCall(this,
    ((MethodInfo) (MethodInfo.GetCurrentMethod())), categoryID)
     .ReturnValue));
}
```

我们使用时，可以用以下代码来调用：

```
var q = from c in db.Categories
        select new
        {
            c.CategoryID,
             TotalUnitPrice =
                 db.TotalProductUnitPriceByCategory(c.CategoryID)
         };
```

这时，LINQ to SQL 自动生成 SQL 语句如下：

```
SELECT [t0].[CategoryID], CONVERT(Decimal(29,4),
[dbo].[TotalProductUnitPriceByCategory]([t0].[CategoryID]))
AS [TotalUnitPrice] FROM [dbo].[Categories] AS [t0]
```

2.在 Where 从句中 使用用户定义的标量函数

这个例子使用方法同上一个例子原理基本相同 了，MinUnitPriceByCategory 自定义函数如下：

```
ALTER FUNCTION [dbo].[MinUnitPriceByCategory]
```

```
(@categoryID INT
)
RETURNS Money
AS
BEGIN
    -- Declare the return variable here
    DECLARE @ResultVar Money
    -- Add the T -SQL statements to compute the return value here
    SELECT @ResultVar = MIN(p.UnitPrice) FROM Products as p
    WHERE p.CategoryID = @categoryID
    -- Return the result of the function
    RETURN @ResultVar
END
```

拖到设计器中 ，生成代码如下：

```
[Function (Name="dbo.MinUnitPriceByCategory", IsComposable=true)]
public System.Nullable<decimal> MinUnitPriceByCategory(
[Parameter(DbType="Int")] System.Nullable<int> categoryID)
{
    return ((System.Nullable<decimal>) (this.ExecuteMethodCall(
    this, ((MethodInfo) (MethodInfo.GetCurrentMethod())),
      categoryID).ReturnValue));
}
```

这时可以使用了：注意这 里在 LINQ to SQL 查询中，对生成的用户定义函数方法 MinUnitPriceByCategory 的内联调用。此函数不会立即执行，这是因为查询会延 迟执行。延迟执行的查询中包含的函数直到此查询执行时才会执行。为此查询生 成的 SQL 会转换成对数据库中用户定义函数的调用（请参见此查询后面的生成 的 SQL 语句），当在查询外部调用这个函数时，LINQ to SQL 会用方法调用表达 式创建一个简单查询并执行。

```
var q =
    from p in db.Products
    where p.UnitPrice ==
      db.MinUnitPriceByCategory(p.CategoryID)
    select p;
```

它自动生成的 SQL 语句如下：

```
SELECT [t0]. [ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0]. [CategoryID],[t0].[QuantityPerUnit], [t0].[UnitPrice],
[t0]. [UnitsInStock], [t0].[UnitsOnOrder],[t0].[ReorderLevel],
[t0]. [Discontinued]FROM [dbo].[Products] AS [t0]
WHERE [t0]. [UnitPrice] =
```

[dbo].[MinUnitPriceByCategory]([t0].[CategoryID])

3.使用用户定义的表值函数

表值函数返回单个行集（与 存储过程不同，存储过程可返回多个结果形状）。由于表值函数的返回类型为 Table，因此在 SQL 中可以使用表的任何地方均可以使用表值函数。此外，您还 可以完全像处理表那样来处理表值函数。

下面的 SQL 用户定义函数显式 声明其返回一个 TABLE。因此，隐式定义了所返回的行集结构。

```
ALTER FUNCTION [dbo].[ProductsUnderThisUnitPrice]
(@price Money
)
RETURNS TABLE
AS
RETURN
      SELECT *
        FROM Products as P
      Where p.UnitPrice < @price
```

拖到设计器中，LINQ to SQL 按如下方式映射此函数：

```
[Function (Name="dbo.ProductsUnderThisUnitPrice",
IsComposable=true)]
public IQueryable<ProductsUnderThisUnitPriceResult>
ProductsUnderThisUnitPrice([Parameter(DbType="Money")]
System.Nullable<decimal> price)
{
      return this.CreateMethodCallQuery
        <ProductsUnderThisUnitPriceResult>(this,
      ((MethodInfo) (MethodInfo.GetCurrentMethod())), price);
}
```

这时我们 小小的修改一下 Discontinued 属性为可空的 bool 类型。

```
private System.Nullable<bool> _Discontinued;
public System.Nullable<bool> Discontinued
{
}
```

我 们可以这样调用使用了：

```
var q = from p in db.ProductsUnderThisUnitPrice(10.25M)
        where ! (p.Discontinued ?? false)
        select p;
```

其生成 SQL 语句如下：

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
[t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
[t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
[t0].[Discontinued]
FROM [dbo]. [ProductsUnderThisUnitPrice](@p0) AS [t0]
WHERE NOT ((COALESCE ([t0].[Discontinued],@p1)) = 1)
-- @p0: Input Money (Size = 0; Prec = 19; Scale = 4) [10.25]
-- @p1: Input Int (Size = 0; Prec = 0; Scale = 0) [0]
```

4.以联接方式使用用户定义的表值函数

我们 利用上面的 ProductsUnderThisUnitPrice 用户定义函数，在 LINQ to SQL 中， 调用如下：

```
var q =
    from c in db.Categories
    join p in db.ProductsUnderThisUnitPrice(8.50M) on
      c.CategoryID equals p.CategoryID into prods
    from p in prods
    select new
    {
        c.CategoryID,
        c.CategoryName,
        p.ProductName,
          p.UnitPrice
    };
```

其生成的 SQL 代码说明对此函数返回 的表执行联接。

```
SELECT [t0].[CategoryID], [t0]. [CategoryName],
[t1].[ProductName], [t1].[UnitPrice]
FROM [dbo].[Categories] AS [t0]
CROSS JOIN [dbo]. [ProductsUnderThisUnitPrice](@p0) AS [t1]
WHERE ([t0]. [CategoryID]) = [t1].[CategoryID]
-- @p0: Input Money (Size = 0; Prec = 19; Scale = 4) [8.50]
```

LINQ to SQL 语句(22)之 DataContext

DataContext

DataContext 作为 LINQ to SQL 框架的主入口点，为我们 提供了一些方法和属性，本文用几个例子说明 DataContext 几个典型的应用。

创建和删除数据库

CreateDatabase 方法用于在服务器上创建数据库。

DeleteDatabase 方法用于删除由 DataContext 连接字符串标识的数据 库。

数据库的名称有以下方法来定义：

如果数据库在连接字符串 中标识，则使用该连接字符串的名称。

如果存在 DatabaseAttribute 属性 (Attribute)，则将其 Name 属性(Property)用作数据库的名称。

如果连接 字符串中没有数据库标记，并且使用强类型的 DataContext，则会检查与 DataContext 继承类名称相同的数据库。如果使用弱类型的 DataContext，则会引 发异常。

如果已通过使用文件名创建了 DataContext，则会创建与该文件 名相对应的数据库。

我们首先用实体类描述关系数据库表和列的结构的 属性。再调用 DataContext 的 CreateDatabase 方法，LINQ to SQL 会用我们的定义 的实体类结构来构造一个新的数据库实例。还可以通过使用 .mdf 文件或只使用 目录名（取决于连接字符串），将 CreateDatabase 与 SQL Server 一起使用。 LINQ to SQL 使用连接字符串来定义要创建的数据库和作为数据库创建位置的服 务器。

说了这么多，用一段实例说明一下吧！

首先，我们新建一 个 NewCreateDB 类用于创建一个名为 NewCreateDB.mdf 的新数据库，该数据库有一 个 Person 表，有三个字段，分别为 PersonID、PersonName、Age。

```
public class NewCreateDB : DataContext
{
    public Table<Person> Persons;
    public NewCreateDB (string connection)
        :
```

```
            base(connection)
        {
        }
        public NewCreateDB(System.Data.IDbConnection connection)
            :
            base(connection)
        {
        }
}
[Table(Name = "Person")]
public partial class Person : INotifyPropertyChanged
{
        private int _PersonID;
        private string _PersonName;
        private System.Nullable<int> _Age;
        public Person() { }
          [Column(Storage = "_PersonID", DbType = "INT",
            IsPrimaryKey = true)]
        public int PersonID
        {
            get { return this._PersonID; }
            set
            {
                if ((this._PersonID != value))
                {
                      this.OnPropertyChanged("PersonID");
                      this._PersonID = value;
                    this.OnPropertyChanged ("PersonID");
                }
            }
        }
        [Column(Storage = "_PersonName", DbType = "NVarChar(30)")]
        public string PersonName
         {
            get { return this._PersonName; }
            set
            {
                if ((this._PersonName != value))
                 {
                    this.OnPropertyChanged ("PersonName");
                    this._PersonName = value;
                    this.OnPropertyChanged ("PersonName");
                }
            }
```

```
        }
    [Column(Storage = "_Age", DbType = "INT")]
    public System.Nullable<int> Age
    {
        get { return this._Age; }
        set
        {
            if ((this._Age != value))
            {
                this.OnPropertyChanged("Age");
                this._Age = value;
                this.OnPropertyChanged("Age");
            }
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged (string PropertyName)
    {
        if ((this.PropertyChanged != null))
        {
            this.PropertyChanged(this,
                new PropertyChangedEventArgs(PropertyName));
        }
    }
}
```

接下来的一段代码先创建一个数据库，在调用 CreateDatabase 后，新的数据库就会存在并且会接受一般的查询和命令。接着插 入一条记录并且查询。最后删除这个数据库。

```
//1.新建一个临时 文件夹来存放新建的数据库
string userTempFolder = Environment.GetEnvironmentVariable
("SystemDrive") + @"YJingLee";
Directory.CreateDirectory (userTempFolder);
//2.新建数据库 NewCreateDB
string userMDF = System.IO.Path.Combine(userTempFolder,
    @"NewCreateDB.mdf");
string connStr = String.Format (@"Data Source=.SQLEXPRESS;
AttachDbFilename={0};Integrated Security=True;
Connect Timeout=30;User Instance=True;
Integrated Security = SSPI;", userMDF);
NewCreateDB newDB = new NewCreateDB(connStr);
newDB.CreateDatabase();
//3.插入 数据并查询
var newRow = new Person
```

```
{
        PersonID = 1,
        PersonName = "YJingLee",
        Age = 22
};
newDB.Persons.InsertOnSubmit(newRow);
newDB.SubmitChanges();
var q = from x in newDB.Persons
                select x;
//4.删除数据库
newDB.DeleteDatabase();
//5.删除临时目录
Directory.Delete (userTempFolder);
```

数据库验证

DatabaseExists 方法用于 尝试通过使用 DataContext 中的连接打开数据库，如果成功返回 true。

下 面代码说明是否存在 Northwind 数据库和 NewCreateDB 数据库 。

```
// 检测 Northwind 数据库是否存在
if (db.DatabaseExists())
        Console.WriteLine("Northwind 数据库存在");
else
        Console.WriteLine("Northwind 数据库不存在");
//检测 NewCreateDB 数据库是否存在
string userTempFolder = Environment.GetEnvironmentVariable("Temp");
string userMDF = System.IO.Path.Combine(userTempFolder,
@"NewCreateDB.mdf");
NewCreateDB newDB = new NewCreateDB(userMDF);
if (newDB.DatabaseExists())
        Console.WriteLine("NewCreateDB 数据库存在");
else
      Console.WriteLine("NewCreateDB 数据库不存在 ");
```

数据库更改

SubmitChanges 方法计算要插入、更 新或删除的已修改对象的集，并执行相应命令以实现对数据库的更改。

无论对象做了多少项更改，都只是在更改内存中的副本。并未对数据库中的实际 数据做任何更改。直到对 DataContext 显式调用 SubmitChanges，所做的更改才会 传输到服务器。调用时，DataContext 会设法将我们所做的更改转换为等效的 SQL 命令。我们也可以使用自己的自定义逻辑来重写这些操作，但提交顺序是由 DataContext 的一项称作“更改处理器”的服务来协调的。事件的顺 序如下：

当调用 SubmitChanges 时，LINQ to SQL 会检查已知对象的集合 以确定新实例是否已附加到它们。如果已附加，这些新实例将添加到被跟踪对象 的集合。

所有具有挂起更改的对象将按照它们之间的依赖关系排序成一 个对象序列。如果一个对象的更改依赖于其他对象，则这个对象将排在其依赖项 之后。

在即将传输任何实际更改时，LINQ to SQL 会启动一个事务来封装 由各条命令组成的系列。

对对象的更改会逐个转换为 SQL 命令，然后发送 到服务器。

如果数据库检测到任何错误，都会造成提交进程停止并引发 异常。将回滚对数据库的所有更改，就像未进行过提交一样。DataContext 仍具 有所有更改的完整记录。

下面代码说明的是在数据库中查询 CustomerID 为 ALFKI 的顾客，然后修改其公司名称，第一次更新并调用 SubmitChanges()方法 ，第二次更新了数据但并未调用 SubmitChanges()方法。

```
//查询
Customer cust = db.Customers.First(c => c.CustomerID == "ALFKI");
//更新数据并调用 SubmitChanges()方法
cust.CompanyName = "YJingLee's Blog";
db.SubmitChanges();
//更新数据没有调用 SubmitChanges()方法
cust.CompanyName = "http://lyj.cnblogs.com";
```

动态查询

使用动态查询，这个例子用 CreateQuery()方法创建一个 IQueryable<T>类型表达式输出查询的语句。这里给个例子说明一下。有 关动态查询具体内容，下一篇介绍。

```
var c1 = Expression.Parameter(typeof(Customer), "c");
PropertyInfo City = typeof(Customer).GetProperty ("City");
var pred = Expression.Lambda<Func<Customer, bool>>(
        Expression.Equal(
       Expression.Property(c1, City),
        Expression.Constant("Seattle")
    ), c1
```

```
);
IQueryable custs = db.Customers;
Expression expr = Expression.Call(typeof(Queryable), "Where",
      new Type[] { custs.ElementType }, custs.Expression, pred);
IQueryable<Customer> q = db.Customers.AsQueryable().
Provider.CreateQuery<Customer>(expr);
```

日志

Log 属性用于将 SQL 查询或命令打印到 TextReader。此方法对了解 LINQ to SQL 功能和调试特定的问题可能很有用。

下面的示例使用 Log 属性在 SQL 代码执行前在控制台窗口中显示此代码。我们可以将此属性与查询、插入、 更新和删除命令一起使用。

```
//关闭日志功能
//db.Log = null;
//使用日志功能：日志输出到控制台窗口
db.Log = Console.Out;
var q = from c in db.Customers
            where c.City == "London"
            select c;
//日志输出到 文件
StreamWriter sw = new StreamWriter(Server.MapPath ("log.txt"), true);
db.Log = sw;
var q = from c in db.Customers
            where c.City == "London"
             select c;
sw.Close();
```

LINQ to SQL 语句(23)之动态查询

动态查询

有这样一个场景：应用程序可能会提供一个用户界面，用 户可以使用该用户界面指定一个或多个谓词来筛选数据。这种情况在编译时不知 道查询的细节，动态查询将十分有用。

在 LINQ 中，Lambda 表达式是许多 标准查询运算符的基础，编译器创建 lambda 表达式以捕获基础查询方法（例如 Where、Select、Order By、Take While 以及其他方法）中定义的计

算。表达式 目录树用于针对数据源的结构化查询，这些数据源实现 IQueryable<T>。 例如，LINQ to SQL 提供程序实现 IQueryable<T>接口，用于查询关系数 据存储。C#和 Visual Basic 编译器会针对此类数据源的查询编译为代码，该代码 在运行时将生成一个表达式目录树。然后，查询提供程序可以遍历表达式目录树 数据结构，并将其转换为适合于数据源的查询语言。

表达式目录树在 LINQ 中用于表示分配给类型为 Expression<TDelegate>的变量的 Lambda 表达式。还可用于创建动态 LINQ 查询。

System.Linq.Expressions 命名空间 提供用于手动生成表达式目录树的 API。Expression 类包含创建特定类型的表达 式目录树节点的静态工厂方法，例如，ParameterExpression（表示一个已命名 的参数表达式）或 MethodCallExpression（表示一个方法调用）。编译器生成 的表达式目录树的根始终在类型 Expression<TDelegate>的节点中，其中 TDelegate 是包含至多五个输入参数的任何 TDelegate 委托；也就是说，其根节点 是表示一个 lambda 表达式。

下面几个例子描述如何使用表达式目录树来 创建动态 LINQ 查询。

1.Select

下面例子说明如何使用表达式树 依据 IQueryable 数据源构造一个动态查询，查询出每个顾客的 ContactName， 并用 GetCommand 方法获取其生成 SQL 语句。

```
//依据 IQueryable 数据 源构造一个查询
IQueryable<Customer> custs = db.Customers;
//组建一个表达式树来创建一个参数
ParameterExpression param =
    Expression.Parameter(typeof (Customer), "c");
//组建表达式树:c.ContactName
Expression selector = Expression.Property(param,
    typeof (Customer).GetProperty("ContactName"));
Expression pred = Expression.Lambda(selector, param);
//组建表达式树:Select (c=>c.ContactName)
Expression expr = Expression.Call(typeof (Queryable), "Select",
    new Type[] { typeof (Customer), typeof(string) },
    Expression.Constant(custs), pred);
//使用表达式树来生成动态查询
IQueryable<string> query = db.Customers.AsQueryable()
     .Provider.CreateQuery<string>(expr);
//使用 GetCommand 方法 获取 SQL 语句
System.Data.Common.DbCommand cmd = db.GetCommand (query);
Console.WriteLine(cmd.CommandText);
```

生成的 SQL 语句为：

```
SELECT [t0].[ContactName] FROM [dbo]. [Customers] AS [t0]
```

2.Where

下面一个例子是“搭建 ”Where 用法来动态查询城市在伦敦的顾客。

```
IQueryable<Customer> custs = db.Customers;
// 创建一个参数 c
ParameterExpression param =
        Expression.Parameter(typeof(Customer), "c");
//c.City=="London"
Expression left = Expression.Property(param,
      typeof(Customer).GetProperty ("City"));
Expression right = Expression.Constant ("London");
Expression filter = Expression.Equal(left, right);
Expression pred = Expression.Lambda(filter, param);
//Where(c=>c.City=="London")
Expression expr = Expression.Call(typeof(Queryable), "Where",
      new Type[] { typeof(Customer) },
      Expression.Constant(custs), pred);
//生成动态查询
IQueryable<Customer> query = db.Customers.AsQueryable()
        .Provider.CreateQuery<Customer>(expr);
```

生成的 SQL 语句为：

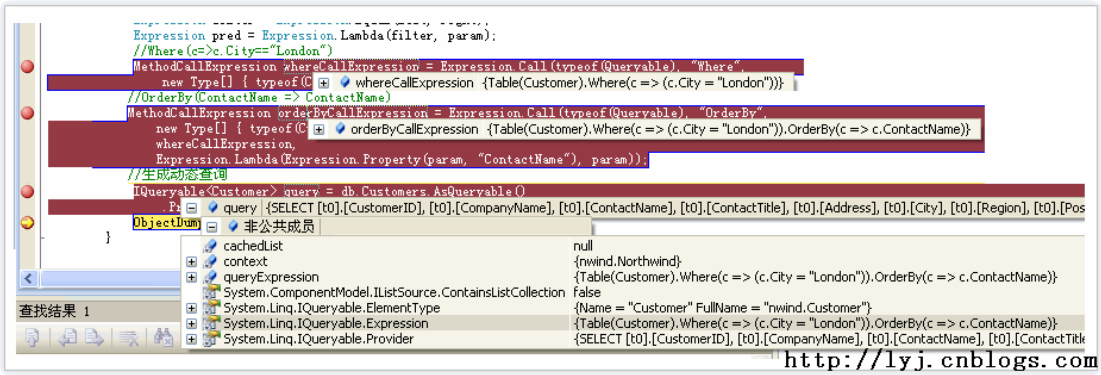
```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0]. [City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0]. [Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0] WHERE [t0]. [City] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

3.OrderBy 本例既实现排序功能又实现了过滤功能。

```
IQueryable<Customer> custs = db.Customers;
//创建一个 参数 c
ParameterExpression param =
    Expression.Parameter (typeof(Customer), "c");
//c.City=="London"
Expression left = Expression.Property(param,
        typeof(Customer).GetProperty ("City"));
```

```
Expression right = Expression.Constant ("London");
Expression filter = Expression.Equal(left, right);
Expression pred = Expression.Lambda(filter, param);
//Where(c=>c.City=="London")
MethodCallExpression whereCallExpression = Expression.Call(
        typeof(Queryable), "Where",
        new Type[] { typeof(Customer) },
        Expression.Constant(custs), pred);
//OrderBy(ContactName => ContactName)
MethodCallExpression orderByCallExpression = Expression.Call(
        typeof(Queryable), "OrderBy",
        new Type[] { typeof(Customer), typeof(string) },
        whereCallExpression,
        Expression.Lambda(Expression.Property
        (param, "ContactName"), param));
//生成动态查询
IQueryable<Customer> query = db.Customers.AsQueryable()
        .Provider.CreateQuery<Customer> (orderByCallExpression);
```

下面一张截图显示了怎么动态生成动 态查询的过程



生成的 SQL 语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Phone], [t0].[Fax]
FROM [dbo].[Customers] AS [t0] WHERE [t0].[City] = @p0
ORDER BY [t0].[ContactName]
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

4.Union

下面的例子使用表达式树动态查询顾客和雇员同在的城 市。

```
//e.City
```

```
IQueryable<Customer> custs = db.Customers;
ParameterExpression param1 =
Expression.Parameter(typeof(Customer), "e");
Expression left1 = Expression.Property(param1,
      typeof (Customer).GetProperty("City"));
Expression pred1 = Expression.Lambda(left1, param1);
//c.City
IQueryable<Employee> employees = db.Employees;
ParameterExpression param2 =
Expression.Parameter(typeof (Employee), "c");
Expression left2 = Expression.Property(param2,
      typeof(Employee).GetProperty ("City"));
Expression pred2 = Expression.Lambda(left2, param2);
//Select(e=>e.City)
Expression expr1 = Expression.Call(typeof(Queryable), "Select",
      new Type[] { typeof(Customer), typeof(string) },
        Expression.Constant(custs), pred1);
//Select(c=>c.City)
Expression expr2 = Expression.Call(typeof(Queryable), "Select",
      new Type[] { typeof(Employee), typeof (string) },
      Expression.Constant(employees), pred2);
//生 成动态查询
IQueryable<string> q1 = db.Customers.AsQueryable()
       .Provider.CreateQuery<string>(expr1);
IQueryable<string> q2 = db.Employees.AsQueryable()
       .Provider.CreateQuery<string>(expr2);
//并集
var q3 = q1.Union(q2);
```

生成的 SQL 语句为：

```
SELECT [t2].[City]
FROM (
      SELECT [t0].[City] FROM [dbo]. [Customers] AS [t0]
      UNION
      SELECT [t1].[City] FROM [dbo].[Employees] AS [t1]
      ) AS [t2]
```

LINQ to SQL 语句(24)之视图

视图

我们使用视图和使用数据表类似，只需将视图从“服务器资源 管理器/数据库资源管理器”拖动到 O/R 设计器上，自动可以创建基于这些 视图的实体类。我们可以同操作数据表一样来操作视图了。这里注意：O/R 设计 器是一个简单的对象关系映射器，因为它仅支持 1:1 映射关系。换句话说，实 体类与数据库表或视图之间只能具有 1:1 映射关系。不支持复杂映射（例如， 将一个实体类映射到多个表）。但是，可以将一个实体类映射到一个联接多个相关表的视图。 下面使用 NORTHWND 数据库中自带的 Invoices、Quarterly Orders 两个视图为例，写出两个范例。

查询：匿名类型形式

我们使用下面代 码来查询出 ShipCity 在 London 的发票。

```
var q =
    from i in db.Invoices
    where i.ShipCity == "London"
    select new
    {
        i.OrderID,
         i.ProductName,
        i.Quantity,
        i.CustomerName
    };
```

这里，生成的 SQL 语句同使用数据表类似：

```
SELECT [t0].[OrderID], [t0].[ProductName], [t0]. [Quantity],
[t0].[CustomerName] FROM [dbo].[Invoices] AS [t0]
WHERE [t0].[ShipCity] = @p0
-- @p0: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

查询：标识映射形式

下例查询出每 季的订单。

```
var q =
    from qo in db.Quarterly_Orders
    select qo;
```

生成 SQL 语句为：

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0]. [City],
[t0].[Country] FROM [dbo].[Quarterly Orders] AS [t0]
```

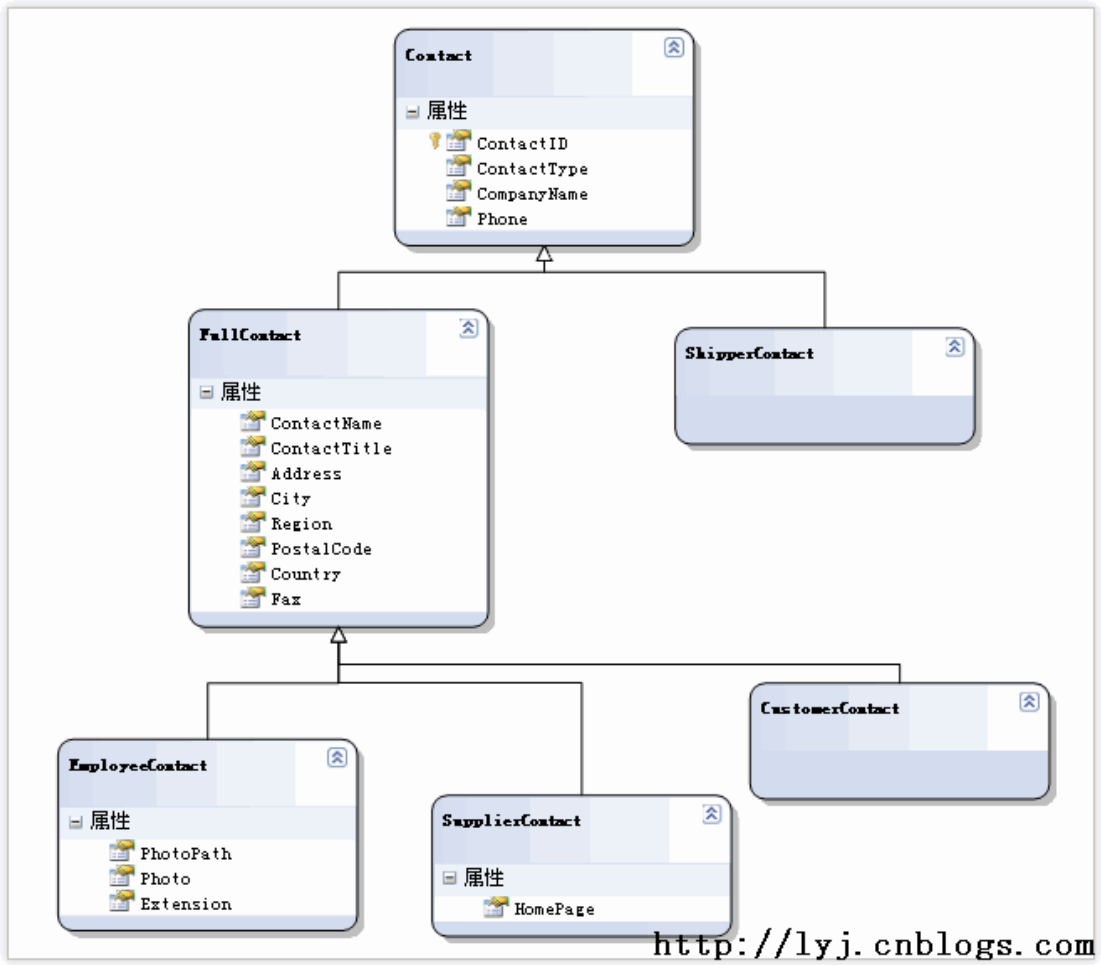
LINQ to SQL 语句(25)之继承

继承支持

LINQ to SQL 支持单表映射，其整个继承层次结构存储在单个数据库表中。该表包含整个层次结构的所有可能数据列的平展联合。(联合是 将两个表组合成一个表的结果，组合后的表包含任一原始表中存在的行。）每行 中不适用于该行所表示的实例类型的列为 null。

单表映射策略是最简单 的继承表示形式，为许多不同类别的查询提供了良好的性能特征，如果我们要在 LINQ to SQL 中实现这种映射，必须在继承层次结构的根类中指定属性(Attribute) 和属性 (Attribute) 的属性 (Property)。我们还可以使用 O/R 设 计器来映射继承层次结构，它自动生成了代码。

下面为了演示下面的几 个例子，我们在 O/R 设计器内设计如下图所示的类及其继承关系。



我 们学习的时候还是看看其生成的代码吧！

具体设置映射继承层次结构有 如下几步：

根类添加 TableAttribute 属性。

为层次结构中的每个 类添加 InheritanceMappingAttribute 属性，同样是添加到根类中。每个 InheritanceMappingAttribute 属性，定义一个 Code 属性和一个 Type 属性。Code 属性的值显示在数据库表的 IsDiscriminator 列中，用来指示该行数据所属的类 或子类。Type 属性值指定键值所表示的类或子类。

仅在其中一个 InheritanceMappingAttribute 属性上，添加一个 IsDefault 属性用来在数据库表中的鉴别器值在继承映射中不与任何 Code 值匹配时指定回退映射。

为 ColumnAttribute 属性添加一个 IsDiscriminator 属性来表示这是保存 Code 值的列 。

下面是这张图生成的代码的框架（由于生成的代码太多，我删除了很 多“枝叶”，仅仅保留了主要的框架用于指出其实质的东西）：

```
[Table(Name = "dbo.Contacts")]
[InheritanceMapping(Code = "Unknown", Type = typeof (Contact),
                                IsDefault = true)]
[InheritanceMapping(Code = "Employee", Type = typeof (EmployeeContact))]
[InheritanceMapping(Code = "Supplier", Type = typeof(SupplierContact))]
[InheritanceMapping(Code = "Customer", Type = typeof (CustomerContact))]
[InheritanceMapping(Code = "Shipper", Type = typeof(ShipperContact))]
public partial class Contact :
INotifyPropertyChanging, INotifyPropertyChanged
{
      [Column(Storage = "_ContactID",IsPrimaryKey = true,
      IsDbGenerated = true)]
      public int ContactID{ }
      [Column(Storage = "_ContactType",IsDiscriminator = true)]
      public string ContactType{ }
}
public abstract partial class FullContact : Contact{ }
public partial class EmployeeContact : FullContact{ }
public partial class SupplierContact : FullContact{ }
public partial class CustomerContact : FullContact{ }
public partial class ShipperContact : Contact{ }
```

1.一般形式

日常我们经常写的形式，对单表查询。

```
var cons = from c in db.Contacts
```

```
                select c;
foreach (var con in cons) {
        Console.WriteLine("Company name: {0}", con.CompanyName);
       Console.WriteLine("Phone: {0}", con.Phone);
        Console.WriteLine("This is a {0}", con.GetType());
}
```

2.OfType 形式

这里我仅仅让其返回顾客的联系方式。

```
var cons = from c in db.Contacts.OfType<CustomerContact>()
                select c;
```

初步学习，我们还是看看生成的 SQL 语句，这样容易理解。在 SQL 语句中查询了 ContactType 为 Customer 的联系方式。

```
SELECT [t0].[ContactType], [t0].[ContactName], [t0].[ContactTitle],
[t0].[Address],[t0].[City], [t0].[Region], [t0].[PostalCode],
[t0].[Country], [t0].[Fax],[t0].[ContactID], [t0].[CompanyName],
[t0].[Phone] FROM [dbo].[Contacts] AS [t0]
WHERE ([t0]. [ContactType] = @p0) AND ([t0].[ContactType] IS NOT NULL)
-- @p0: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [Customer]
```

3.IS 形式

这个例子查找一下发货人的联系方式。

```
var cons = from c in db.Contacts
                where c is ShipperContact
                select c;
```

生成的 SQL 语句 如下：查询了 ContactType 为 Shipper 的联系方式。大致一看好像很上面的一样， 其实这里查询出来的列多了很多。实际上是 Contacts 表的全部字段。

```
SELECT [t0].[ContactType], [t0].[ContactID], [t0]. [CompanyName],
[t0].[Phone],[t0].[HomePage], [t0]. [ContactName],
[t0].[ContactTitle], [t0].[Address], [t0]. [City],
[t0].[Region], [t0].[PostalCode], [t0].[Country],
```

```
[t0].[Fax],[t0].[PhotoPath], [t0].[Photo], [t0].[Extension]
FROM [dbo].[Contacts] AS [t0] WHERE ([t0].[ContactType] = @p0)
AND ([t0].[ContactType] IS NOT NULL)
-- @p0: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [Shipper]
```

4.AS 形式

这个例子 就通吃了，全部查找了一番。

```
var cons = from c in db.Contacts
                select c as FullContact;
```

生成 SQL 语句如下：查询整个 Contacts 表。

```
SELECT [t0]. [ContactType], [t0].[HomePage], [t0].[ContactName],
[t0]. [ContactTitle],[t0].[Address], [t0].[City],
[t0].[Region], [t0]. [PostalCode], [t0].[Country],
[t0].[Fax], [t0].[ContactID], [t0].[CompanyName],
[t0].[Phone], [t0].[PhotoPath],[t0].[Photo], [t0].[Extension]
FROM [dbo].[Contacts] AS [t0]
```

5.Cast 形式

使用 Case 形式查找出在伦敦的顾客的联系方 式。

```
var cons = from c in db.Contacts
                where c.ContactType == "Customer" &&
                            ((CustomerContact)c).City == "London"
                 select c;
```

生成 SQL 语句如下，自己可以看懂了。

```
SELECT [t0].[ContactType], [t0].[ContactID], [t0]. [CompanyName],
[t0].[Phone], [t0].[HomePage],[t0]. [ContactName],
[t0].[ContactTitle], [t0].[Address], [t0].[City], [t0].[Region],
[t0].[PostalCode], [t0].[Country], [t0].[Fax], [t0].[PhotoPath],
[t0].[Photo], [t0].[Extension]FROM [dbo]. [Contacts] AS [t0]
WHERE ([t0].[ContactType] = @p0) AND ([t0]. [City] = @p1)
-- @p0: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [Customer]
```

```
-- @p1: Input NVarChar (Size = 6; Prec = 0; Scale = 0) [London]
```

6.UseAsDefault 形式

当插入一条记录时，使 用默认的映射关系了，但是在查询时，使用继承的关系了。具体看看生成的 SQL 语句就直截了当了。

```
//插入一条数据默认使用正常的映射关系
Contact contact = new Contact()
{
      ContactType = null,
      CompanyName = "Unknown Company",
       Phone = "333-444-5555"
};
db.Contacts.InsertOnSubmit(contact);
db.SubmitChanges();
//查询一条数据默认使用继承映射关系
var con =
      (from c in db.Contacts
       where c.CompanyName == "Unknown Company" &&
                                    c.Phone == "333-444-5555"
       select c).First();
```

生 成 SQL 语句如下：

```
INSERT INTO [dbo].[Contacts] ([ContactType], [CompanyName],
[Phone]) VALUES (@p0, @p1, @p2)
SELECT TOP (1) [t0].[ContactType], [t0].[ContactID],
[t0]. [CompanyName], [t0].[Phone],[t0].[HomePage],
[t0].[ContactName], [t0].[ContactTitle], [t0].[Address],
[t0].[City],[t0].[Region], [t0].[PostalCode], [t0].[Country],
[t0].[Fax], [t0].[PhotoPath], [t0].[Photo], [t0].[Extension]
FROM [dbo].[Contacts] AS [t0]
WHERE ([t0].[CompanyName] = @p0) AND ([t0].[Phone] = @p1)
-- @p0: Input NVarChar (Size = 15; Prec = 0; Scale = 0)
      [Unknown Company]
-- @p1: Input NVarChar (Size = 12; Prec = 0; Scale = 0)
      [333-444-5555]
```

7.插入新的记录

这个例子说明如 何插入发货人的联系方式的一条记录。

//

1.在插 入之前查询一下，没有数据

```
var ShipperContacts =
      from sc in db.Contacts.OfType<ShipperContact>()
      where sc.CompanyName == "Northwind Shipper"
      select sc;
//
```

2.插入数据

```
ShipperContact nsc = new ShipperContact()
{
      CompanyName = "Northwind Shipper",
      Phone = "(123)-456-7890"
};
db.Contacts.InsertOnSubmit(nsc);
db.SubmitChanges();
//
```

3.查询数据，有一条记录

```
ShipperContacts =
      from sc in db.Contacts.OfType<ShipperContact>()
        where sc.CompanyName == "Northwind Shipper"
      select sc;
//
```

4.删除记录

```
db.Contacts.DeleteOnSubmit (nsc);
db.SubmitChanges();
```

生成 SQL 语句如下：

```
SELECT COUNT(*) AS [value] FROM [dbo].[Contacts] AS [t0]
WHERE ([t0].[CompanyName] = @p0) AND ([t0].[ContactType] = @p1)
AND ([t0].[ContactType] IS NOT NULL)
-- @p0: Input NVarChar [Northwind Shipper]
-- @p1: Input NVarChar [Shipper]
INSERT INTO [dbo].[Contacts]([ContactType], [CompanyName], [Phone])
VALUES (@p0, @p1, @p2)
-- @p0: Input NVarChar    [Shipper]
-- @p1: Input NVarChar    [Northwind Shipper]
-- @p2: Input NVarChar    [(123)-456-7890]
SELECT COUNT(*) AS [value] FROM [dbo].[Contacts] AS [t0]
WHERE ([t0].[CompanyName] = @p0) AND ([t0].[ContactType] = @p1)
AND ([t0].[ContactType] IS NOT NULL)
-- @p0: Input NVarChar [Northwind Shipper]
-- @p1: Input NVarChar [Shipper]
DELETE FROM [dbo].[Contacts] WHERE ([ContactID] = @p0) AND
([ContactType] = @p1) AND ([CompanyName] = @p2) AND ([Phone] = @p3)
-- @p0: Input Int    [159]
-- @p1: Input NVarChar    [Shipper]
-- @p2: Input NVarChar    [Northwind Shipper]
-- @p3: Input NVarChar    [(123)-456-7890]
-- @p4: Input NVarChar    [Unknown]
-- @p5: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [Supplier]
-- @p6: Input NVarChar (Size = 7; Prec = 0; Scale = 0) [Shipper]
-- @p7: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [Employee]
-- @p8: Input NVarChar (Size = 8; Prec = 0; Scale = 0) [Customer]
```